游戏开发与编程系列

Java 游戏编程

北京希望电子出版社 总策划
David Brackeen 著
邱仲潘 译

科学出版社
www.sciencep.com

内 容 简 介

Java 游戏编程已经在游戏开发中成为重头戏。使用 Java 的好处是明显的：代码易读易写，而且可以在不同平台上方便快捷地开发。如果你已经有 Java 编程的经验，而且想涉及游戏编程，那么《Java 游戏编程》是不错的选择。

本书介绍如何使用 Java 1.4 平台建立快速的全屏动作游戏，如横向卷轴和三维射击程序。介绍的关键特性包括 Java 游戏编程技术，最新二维图形和声音技术，三维图形和视景管理，路径寻找和人工智能，碰撞探测，使用 BeanShell 的游戏脚本和多方游戏引擎的生成等。

本书主要内容包括：掌握全屏图形与动画、硬件加速图形技术；在游戏中增加人工智能和精彩的多方功能；生成定制鼠标和键盘控制的系统；在全屏游戏中集成 Swing 用户界面；同时播放多个声音，使用回响、伪三维声音过滤之类的实时声音效果；用 BSP 树、碰撞探测和脚本生成三维引擎；学习现代 Java 虚拟机的优化方法。

本书面向游戏开发人员及广大游戏编程爱好者，同时，也是大专院校相关专业师生的有力参考。本书配套光盘内容为书中实例源代码。

需要本书或需要得到技术支持的读者，请与北京中关村 083 信箱（邮编 100080）发行部联系。电话：010-62528991，62524940，62521921，62521724，82610344，82675588（总机）传真：010-62520573，E-mail：yanmc@bhp.com.cn

版权声明

本书英文版名为：*Developing Games in Java*，作者是 David Brackeen，由 New Riders 出版，版权归 New Riders。本书中文版由 New Riders 授权出版，译者邱仲潘。未经出版者书面许可，本书的任何部分不得以任何形式或手段复制或传播。

图书在版编目（CIP）数据

Java 游戏编程/（美）布拉肯（Brackeen,D.）著；邱仲潘译. —北京：科学出版社，2004.3

（游戏开发与编程系列）

ISBN 7-03-012942-3

Ⅰ. J… Ⅱ. ①布…②邱… Ⅲ. ①JAVA 语言—程序设计②游戏—应用程序—程序设计 Ⅳ. ①TP312 ②G899

中国版本图书馆 CIP 数据核字（2004）第 011531 号

责任编辑：杨如林 安源 / 责任校对：周 艳

责任印刷：媛明 / 封面设计：梁运丽

科学出版社 出版

北京东黄城根北街 16 号

邮政编码：100717

http://www.sciencep.com

北京市媛明印刷厂印刷

科学出版社发行 各地新华书店经销

*

2004 年 6 月第一 版 开本：787×1092 1/16

2004 年 6 月第一次印刷 印张：42 1/2

印数：1—5 000 字数：982 000

定价：68.00 元（含 1CD）

程序员　代序

我 98 年离开前导公司的时候手里有一份《游戏制作者通讯录》，里面记录了 83 个人的姓名，单位和联系方法。这里面有公司的同事、兄弟公司的老总、代理商，杂志社的编辑以及相关行业的工程师。在不知不觉中，国内的游戏产业已经开始迎接第 10 个年头了，碰巧又把这份名单翻出来——其中的许多人已经很久没有联系了，估计他们都离开游戏行业了。虽然许多相关的公司早已经销声匿迹或是转行，但是仍能发现为数不多的几个人还在做游戏。不自觉地想在这 10 年的时间里，有多少辛酸的故事被述说着呢？

现在的游戏制作行业终于可以抬起头了。如果我跟别人说“我是做游戏的”，那么再也没有人报以诧异的目光。也许大家觉得难以理解？但就是在短短的 2 年前，这还是每一个游戏制作者曾经尴尬的局面。就拿图书出版来说吧，早先学习游戏制作的人，从来不会见到一本跟游戏沾边的书籍，所有的资料就是依靠阅读人家软件自带的帮助文件，还都是英文的。虽然间接锻炼自己的英文水平，但总想着什么时候能见到用自己国家语言所书写的游戏书籍呢？

上个星期去西单图书大厦，在堆砌着满满一层计算机图书的四楼大厅里，竟然看到有一个书架上赫然写着“游戏开发”四个大字。虽然书架上只有可怜的十几种，而且内容是关于题材的，其中也不乏内容杂乱无章，不成体系之辈，但毕竟表明游戏开发有了自己的书籍。当时站在那里，感觉真的有些受宠若惊，因为从来没有想过游戏制作人会获得如此的待遇，也许梦想真的快实现了。

现在人们的生活要比过去丰富、自由，于是许多人开始寻找自己的价值，游戏圈也是如此——有的人从玩游戏的过程中找到了，有的人从做游戏的过程中找到了。

这又让我想起那些为数不多的游戏制作人，朋友们你们现在在想什么呢！经历过无数个不眠的夜，多年的风风雨雨之后，如何思考中国游戏的现状和未来呢？中国游戏业过去面临的是无奈和彷徨，接受盗版的冲击和生存的考验。终于冬天过去，春天来了——伴随着暖暖的风和悄无声息的细雨。我希望藉此机会，把这套书奉献给那些刚刚踏入游戏行业的初学者们，这里也包括我对你们的期望。

希望通过这套书带给你们在游戏制作中最基本的知识，还有更多前人的经验，使你们更容易的进入行业中，以尽快达到专业的水准。因为只有这样，中国游戏制作的队伍才能不断的壮大。我不希望自己的通讯录中只有 83 的名字，而现在的你们将是它永不停息的动力。

希望通过这套书让你们能够与前人进行心灵上的沟通，从而逐步了解你的内心。**游戏制作人是要求奉献出自己的心灵，因为只有注入了灵魂的游戏才是让人喜欢的游戏，真正优秀的游戏。**游戏制作业里面充满了机遇和竞争，不管你选择留下或是离开，我都希望你们能够找到自我，找到自己的梦想，找到自己的归宿。

希望通过这套书带我们走过更加漫长的人生旅途。在游戏制作的过程中，我们曾经接受过别人的指点，在未来，我们也会成为黑暗道路上照亮他人前程的一盏指路灯。毕竟，前面的道路并不平坦，唯有我们的前赴后继它才能越走越宽。

刘刚

序者介绍

刘刚　1996年毕业于北京工业大学计算机科学与工程系（五年制），本科。

1995年8月实习进入前导软件有限公司，参与制作国内第一个基于Windows 95的游戏《三国志演义－官渡》。毕业后，正式加入前导软件有限公司。成立瞬间工作室，领导开发游戏《三国志演义－赤壁》。1997年5月，前往日本大阪协助进行游戏日本版的制作。同年7月，《三国志演义－赤壁》、《三国志演义—官渡》均销往日本，12月完成《赤壁增强版》并开始《三国志演义－荆州》的前期设计工作。1998年1月，参与另外一个游戏《西游记－齐天大圣》后期的部分程序工作，并开始研制三维引擎。1998年5月《西游记－齐天大圣》完成，6月前导软件有限公司研发部门解散。

1998年8月，带领瞬间工作室加盟中青旅尚洋电子技术有限公司，开始《烈火文明》的制作。这是一款“三维RPG游戏”，自己的三维引擎知识得到第二次应用。

2000年2月只身离开中青旅尚洋电子技术有限公司，进入目标软件（北京）有限公司。作为项目负责人开始《秦殇》的制作，这是一款多人动作RPG游戏。2001年5月《秦殇》作为唯一一款国产游戏参加美国E3，2002年7月《秦殇》上市。2002年8月开始制作网络游戏《天骄-秦殇世界》（MMORPG），秦殇的后续产品《复活-秦殇前传》也开始正式启动。2003年4月网络游戏《天骄-秦殇世界》正式收费，并在韩国进入公测，此年1月单机游戏《复活-秦殇前传》在国内发售。

技术经验

1997年开始使用汇编语言为游戏《赤壁》进行基于Pentium CPU的MMX优化，后应用于该产品中。

1998年开始亲自研发基于Direct3D 5.0上的三维引擎，后应用于《西游记-齐天大圣》和《烈火文明》中。

2000年开始亲自研发专门用于RPG游戏的脚本引擎——EasyRPG，后应用于《秦殇》《天骄》《复活》《天骄2》等项目中。

2001年开始亲自研发基于Winsock的网络引擎，后应用于《秦殇》的战网和《天骄》等网络游戏中。

译者　序

2003 年春天的一场非典狂潮，把许多人关在家里，许多长期没时间看的书，一下子把它看完了；许多没时间欣赏的影视节目，一下子把它欣赏完了；许多没时间上的网站，一下子把它上完了；许多没时间玩的游戏，一下子把它玩完了。一到夏天，又是一场全球高温，把大家赶到水里，险些泡成水饺。看着阳光在水里照出的彩色斑条，我突然想，能自己编个游戏该多好。

真是事如人愿，一本《Java 游戏编程》的书到了我的手上，真是高兴坏了。儿子是个游戏迷，比我还高兴，马上就预订了，“老爸，到时要送我一本样书”。我说，“没问题，但你先得把 Java 语言学好”。

翻译本书时，正值初秋时节，但我和陈凌峰、陈纯颖、张荣、李青、钟锃光、王凌飞一起，在白鹭飞起的地方，怀着巨大的热情，挥毫泼墨，细推慢敲，终于拿出自己略感满意的手稿，厦门市电脑学会理事长李棠秋教授亲自审阅了部分章节，刘云昌、刘昌和、严明英、赖华龙、周阳生、邹能东、李耀平、彭振庆、刘文琼等朋友也在翻译整理和录排方面提供了诸多帮助。套用一句老话，这是集体智慧的结晶，在此对各位深表感谢。也盼望广大读者不吝赐教，让我们精益求精，更上一层楼。

译 者

前 言

如果你选购这本书，则一定想真正进行 Java 游戏编程。

许多现代游戏的外观、声效和表现非常精彩，让人很难想像它是怎么编写出来的。但是，即使最大最复杂的游戏也是从一些基本的游戏编程概念开始的。本书的目的是介绍这些基本的游戏编程概念，使你能够自己创建出精彩出众的游戏。

过去的许多 Java 游戏编程图书都是针对简单 Web 游戏小程序的，如卡通游戏和谜题游戏。但是，Java 的功能远远不止这些。在这本 Java 游戏编程图书中，我们要建立高速全屏动作游戏，如横向卷轴（side scrollers）和三维射击游戏，充分利用 Java 的功能。

其实，本书不是介绍如何编写 Java 游戏，而是介绍如何编写游戏，只是恰好使用 Java 语言而已。我们将介绍许多 Java 游戏编程主题，就像使用某些 API 得到所需的结果，我们还会介绍许多一般性的游戏编程主题，如：碰撞探测、路径寻找、人工智能和 BSP 树。

为什么选择 Java

Java 1.4 可以建立高速全屏的硬件加速游戏，同时充分享受 Java 平台开发的好处。Java 平台开发可以得到简单、先进、安全和面向对象的语言，同时具有全面的 API、简单的多线程、自动内存回收和高度的移植性。另外，Java 平台开发还有大量工具，包括许多开放源代码库和一些精彩的 IDE 开发环境。

Java 的主要问题是速度，但随着 HotSpot 虚拟机和硬件加速图形的出现，速度问题不再是个大问题。HotSpot 在运行时将游戏中的“热点”部分编译成自然代码，而硬件加速图形可以充分利用强大的显示卡功能。

预备要求

学习本书时，并不要求具有游戏编程方面的经验，但应符合下列预备要求。

- 应知道如何用 Java 编程。如果你还在学习 Java 或更熟悉 C++之类其他面向对象语言，则可以先看一些 Java 方面的书籍，如 *Learning Java*，作者是 Patrick Niemeyer 和 Jonathan Knudsen（O'Reilly 出版公司出版，2002）。
- 需要 Java 2 标准版 SDK V1.4.1 以上版本，可以从 http://java.sun.com 下载。
- 手头最好拥有 J2SE 文档。拥有 J2SE API 文档有利于加快编程过程。可以从 http://java.sun.com 下载，也可以直接到该站点浏览。
- 本书大多数代码都是用 Ant 编译的，这是基于 Java 的编译工具。如果使用本书的编译文件，则要熟悉 Ant 的用法。本书稍后“使用 Ant”一节将会介绍 Ant 的用法。
- 最后，要下载本书的所有源代码，地址为 www.brackeen.com/javagamebook。

但是，最重要的动力来自于开发游戏的强烈愿望。建立游戏并不容易，有时会在一个小问题上折腾多日而毫无进展。但是，如果你有建立游戏的强烈愿望，一切困难都是可以克服的。

本书内容

总的说来，本书内容包括创建二维游戏和三维游戏，现简述如下。

第 1 章“Java 线程”是关于 Java 线程的简单入门知识。线程是许多人感到费解的概念之一，本章澄清了许多问题并指出游戏中什么时候使用线程，什么时候不使用线程。

第 2 章“二维图形与动画”到第 5 章“创建二维平台游戏”介绍任何游戏的基础：图形、输入和声音。其中的主题包括全屏硬件加速图形、动画和幽灵，如何创建定制键盘与鼠标控件的系统，如何在全屏游戏中集成 Swing，如何同时播放多种声音，如何播放回声、伪三维声音过滤之类的声音效果。第 5 章用这些概念创建精彩的二维横向卷轴游戏。

第 6 章“多人游戏”介绍用 NIO 创建多人游戏，首先介绍多用户闲聊服务器，然后创建多人游戏框架，同时介绍许多高级问题和技巧。

第 7 章“三维图形”到第 9 章“三维对象”主要介绍三维图形，包括三维基础和纹理贴图、灯光、三维实体与 Z 缓存。第 10 章“用 BSP 树管理三维视景”要用 BSP 树创建三维引擎。

第 11 章“碰撞探测”到第 14 章“游戏脚本”介绍各种游戏编程技术，并将其应用于三维引擎中。介绍各种碰撞探测技术、路径寻找（使用 A*算法）、人工智能（包括模拟进化）和游戏事件与脚本。这些技术既能在三维世界中采用，又能在二维世界中采用。

第 15 章“保存游戏”到第 18 章“游戏设计与最后 10%”介绍游戏的其他方面，包括持久性（保存游戏）、优化、创建图形与声音、游戏设计和游戏的最后 10%（包括调试要点和游戏发布）。

最后一章是 Java 未来概述，包括 Java 平台的发展方向和怎么使 Java 更适合游戏编程。

本书多处强调代码复用和框架建立。前几章建立的代码将在书中多处使用，我们要在现有类中增加代码，扩展其功能。

本书不包括的内容

本书介绍概念而不是介绍 API，不准备介绍 Java 1.4 SDK 中的每个 API。Java API 文档中会详细介绍如何画圆和画矩形及 API 的更多细节。另外，本书也不是 Java 入门书，读者必须会用 Java 编程。

本书针对 Java 1.4 标准版，而不是 Java 1.1 或小程序。Microsoft 的 Java 1.1 虚拟机仍然是小程序最常用的虚拟机，因为它放在 Internet Explorer 浏览器中。但是，本书针对 Java 1.4 以上版本，因此不介绍普通的小程序问题及解决方法。如果对 Java 1.1 或小程序感兴趣，本书仍然提供了大量游戏编程概念，但可能还需要阅读其他书籍。如 *Black Art of Java Game Programming*，作者为 Joel Fan(Waite Group 出版公司出版,1996) *Java Game Programming for Dummies*，作者为 Wayne Holder(IDG Books Worldwide 出版公司出版，1998)。

另外，本书也不介绍 J2ME（移动电话和其他设备上使用的 Java 版本）。关于这方面的信息，见 *Micro Java Game Development*,作者为 David Fox(Addison Wesley Professiona 出版公司出版，2002)。

我们将详细介绍三维图形问题，包括纹理贴图、照明、碰撞探测技术和 BSP 树，但本书不介绍任何硬件加速三维图形。目前，Java 1.4 SDK 中没有包括任何硬件加速三维 API，

但市面上已经有几种 API，包括 Java 3D 与 gl4java。可以用 Java 3D 与 OpenGL 图书作为本书的补充读物。本书的宗旨是详细介绍三维图形编程，有了这个知识之后，就很容易转入学习各种硬件加速三维 API。

最后，本书不是游戏编程的全面解析，而要为游戏编程奠定一个坚实的基础。游戏编程是个庞大的议题，游戏编程的概念似乎是无穷无尽的。许多概念可以自己悟出，有些则可以从其他书籍与文献中学习。

本书代码

本书把几乎所有代码都放在包中，避免在一个目录中出现几百个 Java 文件，包的布局使整个代码库一目了然。不在包中的代码只有那些单纯用于章内显示的简单 Java 文件，其名称通常包含 Test 字样，如 CollisionTest。

代码放在每章的独立目录中，使各章的代码间不会相互干扰。由于本书的所有代码是复用的，因此，每一章的代码通常包含前面各章的所有代码，再加上本章增加的新内容。

另外，每章的大部分代码都有 Ant 建立文件，可以方便地编译和建立代码。

使用 Ant

Apache Ant 是免费的 Java 编译工具，可以从 http://ant.apache.org 下载。由于 Ant 是基于 Java 的，因此 Ant 建立文件可以在任何平台上编译。另外，Ant 已经集成到许多 Java IDE 中，因此可以方便地启动和运行。

Ant 建立文件是 XML 文件，通常命名为 build.xml。清单 1 是个 Ant 建立文件的例子。

清单 1　Ant 建立文件

```
<?xml version “1.0”?>
<project name ="myBuild" default= "build" basedir=".">
  property name ="srcdir" value="src"/>
  property name ="destdir" value="build"/>
  <target name="clean" description="Deletes the class files">
    <delete>
    <fileset dir="${destdir}"includes="**/*.class"/>
  </delete>
 </target>
 < target   name="compile" description="Compiles the source code">
  <javac srcdir="${srcdir}">
          destdir="${destdir}"
          debug="on">
 </javac>
</target>
< target   name="Compile" depends="compiles"
    description="Compiles the source code and creates a jar file">
```

```
  </jar>
</target>
< target   name="rebuild" depends="clean,build"
   description="Performs a clean build"/>
<project>
```

这个 Ant 建立文件的例子几乎是描述自己的。首先，它生成两个属性，指定 srcdir 与 destdir 目录。属性可以用｛varName｝语法指定。

Ant 建立文件包含 4 个目标文件：clean，compile，build 和 rebuild。

Compile 目标编译源目录中的所有 Java 文件，将编译的类文件放在目标目录中。Clean 目标则删除所有这些类。

Build 目标取决于 Compile 目标，因此，运行 Build 目标时先自动运行 Compile 目标。Build 目标将所有编译的类文件包装成 jar 文件。

最后，rebuild 目标本身什么也不干，但根据 clean 与 build 目标进行净建立。

安装 Ant 之后，只要在建立文件的目录中打开命令提示并键入要执行的目标如下：build.xml。

这个命令将 build.xml 文件装入当前目录中，并执行 compile 目标。如果不指定目标，则执行 build 目标，因为它定义为建立文件中的缺省目标：ant。

除了从命令行运行 Ant 程序之外，Ant 还密切集成在许多集成开发环境中和文本编辑器中，如 Jbuilder，jEdit，Eclipse，IDEA 与 NetBeans。例如，在图 1 所示的 jEdit 中，AntFarm 插入件可以浏览 Ant 文件，单击鼠标执行目标和把编译错误发送到 jEdit 的 ErrorList 插入件中，以便单击错误直接进入编译错误的源中。

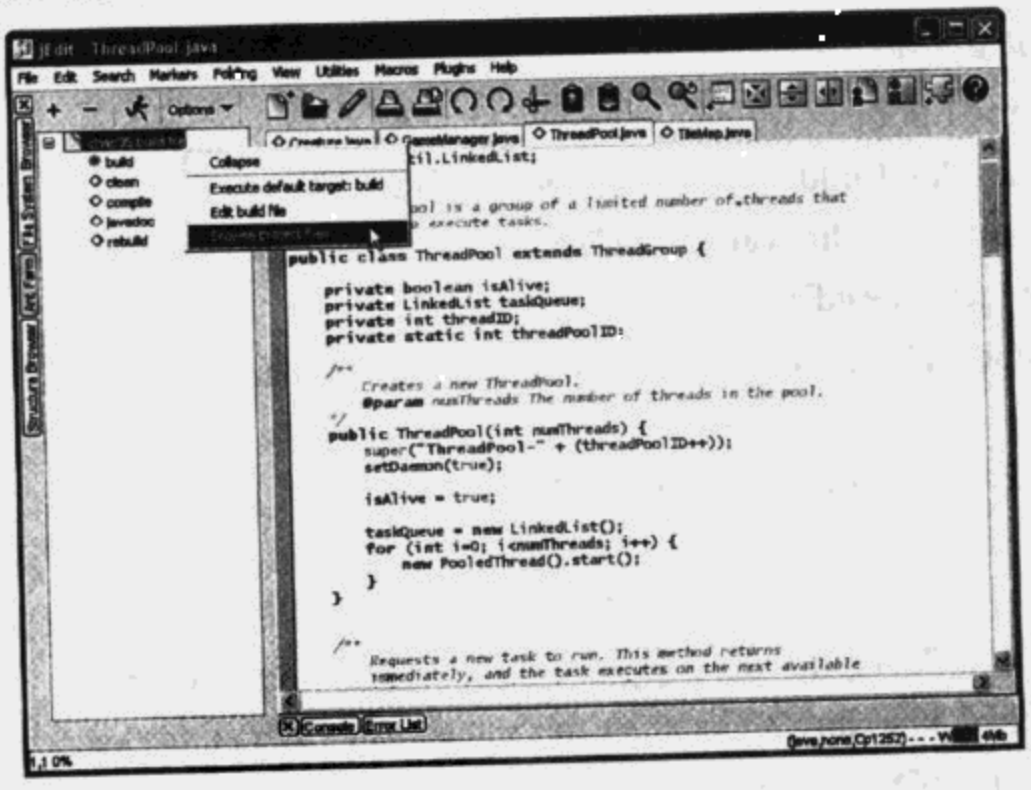


图 1　集成 jEdit 4.1，这是常用和高度可配置的开放源代码编辑器

这个 Ant 建立文件例子只显示了 Ant 功能的一点皮毛。有许多内置 Ant 任务和更多可选任务，甚至可以编写自己的任务。要显示更多任务和使用 Ant 的更多细节，见 http://ant.apache.org/manual/。

希望本书有趣和有用。如果要和作者讨论有关本书的任何问题，可以访问游戏站点：

www.brackeen.com/javagamebook。记住，你还可以从这个游戏站点取得本书的所有源代码。

作者介绍

David Brackeen 生长在德克萨斯洲，获得北德克萨斯洲大学计算机科学专业的学士学位。曾经使用 Java 创建了许多游戏、关卡编辑器和多媒体产品，包括 Scared(一种三维射击游戏)和 Race3d（一种三维比赛引擎，在多种游戏中使用）。有人说他喝过鞋子中的雨水，他既不肯定，也不否定。目前住在洛杉机，网址为 www.brackeen.com。

友情：作者介绍

Bret Barker 生长在纽约，在麻萨渚塞州沃色斯特大学学过电子工程和计算机科学专业。作为旧金山 dot-com 公司爆炸式增长的逃难者，目前住在缅因州波特兰市，是个自由的软件开发人员。他从事为 Java 游戏、移动应用开发和三维图形编程。E-mail 地址为 bret@hypefiend.com。

Laurence Vanhelsuwe 是自学成才的独立软件工程师。他涉及的技术多种多样，包括 X.25WAM 路由器、虚拟实境飞行模拟、Postscript、基于视频的实时数字化交通分析和基于交互式地图的多媒体光盘。Laurence Vanhelsuwe 并不是整天躲在屏幕后面干活，他喜欢爬山、冲浪，这样促进血液循环。

技术审校人员介绍

下列技术审校人员把大量亲身经验投进了整个《Java 游戏编程》项目开发过程中。本书编写过程中，这些技术审校人员审查了所有材料的技术内容、组织编排和流程。他们的反馈保证《Java 游戏编程》满足读者对高质量技术信息的需求。

David Fox 设计和开发了多个光盘、Web 和无线游戏，包括 Fox 交互式公司、Byron Preiss 多媒体公司和 PlayLink 公司的游戏。目前，他是 Next Game 有限公司的股东之一，开发的系统能使人们充分施展游戏技能。David 还是多本关于互联网技术的畅销书的作者。他的著作已经在 Gamasutra,Salin.com 和 Developer.com 等出版社出版，过去几年他一直参加了 JaveOne 会议上的 Java 游戏主题讨论。

Tom Jacobson 在新墨西哥圣菲市的圣约输学院取得本科学历，毕业后开始在 UCSF 与卢格大学的联合研究项目中创建交互式计算机游戏，并帮助朗读有困难的小孩康复。Tom 是 Gamelet.com 公司的联合创办者，这是一家联机 Web 游戏公司，已经为 Snapple/Sony 与 Subaru 等公司创建了多种 Java 游戏。他目前是 www.superdudes.net 的游戏总监。

目　录

第一部分　Java 游戏基础

第 1 章　Java 线程 1
1.1　什么是线程 1
1.2　在 Java 中生成与运行线程 2
1.2.1　扩展 Thread 类 2
1.2.2　实现 Runnable 接口 3
1.2.3　使用匿名内类 3
1.2.4　等待线程完成 3
1.2.5　休眠线程 3
1.3　同步 4
1.3.1　为什么要同步 4
1.3.2　如何同步 4
1.3.3　同步是什么 5
1.3.4　不同步是什么 6
1.3.5　避免死锁 6
1.4　使用 wait()与 notify() 7
1.5　Java 事件模型 8
1.6　何时使用线程 8
1.7　何时不使用线程 8
1.8　线程池 8
1.9　小结 14
第 2 章　二维图形与动画 15
2.1　全屏图形 15
2.1.1　屏幕布局 15
2.1.2　像素颜色与位深度 16
2.1.3　刷新率 17
2.1.4　将显示切换到全屏方式 17
2.1.5　抗齿边 21
2.1.6　选用显示方式 22
2.2　图像 23
2.2.1　透明度 23
2.2.2　文件格式 23
2.2.3　读取图像 24
2.2.4　硬件加速图像 27
2.2.5　图形绘制基准程序 28
2.2.6　动画 32
2.2.7　活动绘制 36
2.2.8　动画循环 36
2.3　避免闪烁与裂开 39
2.3.1　双缓存 40
2.3.2　翻页 41
2.3.3　监视刷新与裂开 42
2.3.4　BufferStrategy 类 42
2.3.5　生成屏幕管理器 43
2.3.6　幽灵 52
2.4　简单效果 59
2.5　小结 65
第 3 章　交互性与用户界面 66
3.1　AWT 事件模型 69
3.2　键盘输入 70
3.3　鼠标输入 74
3.4　通过鼠标移动的观看模式 78
3.5　创建输入管理器 84
3.6　使用输入管理器 97
3.6.1　暂停游戏 97
3.6.2　增加重力 98
3.7　设计直观用户界面 105
3.8　使用 Swing 组件 106
3.8.1　Swing 基础 106
3.8.2　在全屏方式中使用 Swing 107
3.9　创建简单菜单 109
3.10　让游戏者配置键盘 115
3.11　小结 122
第 4 章　声效与音乐 123
4.1　声音基础 123
4.2　Java Sound API 124
4.2.1　打开声音文件 124
4.2.2　使用 Line 124
4.3　播放声音 125
4.4　创建实时声音过滤体系结构 131
4.5　创建实时回响过滤 135
4.6　仿真三维声音 139

4.6.1 生成三维过滤器的思路 139
4.6.2 实现三维过滤器 140
4.6.4 测试三维过滤器 143
4.7 创建声音管理器 147
4.7.1 Sound 类 147
4.7.2 SoundManager 类 148
4.7.3 线程局部变量 157
4.8 播放音乐 158
4.8.1 播放光盘声频 158
4.8.2 播放 MP3 与 Ogg Vorbis 159
4.8.3 播放 MIDI 音乐 159
4.8.4 生成适配音乐 163
4.9 小结 166
第 5 章 创建二维平台游戏 167
5.1 生成基于地砖的地图 167
5.1.1 实现 Tile 地图 168
5.1.2 读取 Tile 地图 172
5.1.3 绘制 Tile 地图 175
5.1.4 绘制幽灵 176
5.1.5 视差滚动 177
5.1.6 充电器 179
5.1.7 简单坏人 181
5.2 碰撞探测 190
5.2.1 探测碰撞 190
5.2.2 处理碰撞 191
5.2.3 幽灵碰撞 194
5.3 完成与加速 196
5.4 生成可执行的 .jar 文件 196
5.5 扩展游戏 197
5.6 小结 198
第 6 章 多人游戏 199
6.1 Java I/O 库的变革 200
6.1.1 JDK 1.4 NIO 库的概述 201
6.1.2 信道 201
6.1.3 缓存 203
6.1.4 选择器与 SelectionKeys 205
6.2 基本多方游戏应用程序 206
6.2.1 服务器端 206
6.2.2 建立与运行服务器 212
6.2.3 客户端 214
6.2.4 建立与运行客户机 215
6.3 多方游戏服务器框架 215
6.3.1 设计目标与战术 215
6.3.2 设计 216
6.3.3 共用类与接口 219
6.3.4 服务器实现 221
6.3.5 客户机 229
6.4 RPS 游戏（石头、布、剪刀） 230
6.4.1 类 230
6.4.2 运行 RPS 游戏 233
6.5 建立框架 235
6.5.1 客户机 GUI 235
6.5.2 持久性 235
6.5.3 伙伴名单、门厅与闲聊 235
6.6 服务器管理 236
6.6.1 登记 236
6.6.2 启动/关闭 236
6.6.3 服务器管理控制台 237
6.6.4 游戏监查 238
6.7 高级课题 238
6.7.1 切断与重接 238
6.7.2 HTTP 隧道 239
6.7.3 量化测试 240
6.7.4 Modem 连接 241
6.7.5 性能剖析与统计 241
6.7.6 性能调整 242
6.8 小结 244
第二部分 三维图形与高级技术
第 7 章 三维图形 245
7.1 三维绘制的类型 246
7.2 回顾曾经学过的数学 246
7.2.1 三角学和直角三角形 247
7.2.2 向量数学 247
7.3 3D 的基础 252
7.4 3D 的数学 254
7.5 多边形 259
7.6 三维变换 262
7.6.1 旋转 263

7.6.2 包装旋转和平移变换 264
7.6.3 采用变换 268
7.6.4 旋转顺序 270
7.7 3D 的简单管道 270
7.8 镜头移动 275
7.9 实体和背面删除 275
7.9.1 点积 276
7.9.2 交积 277
7.9.3 再谈点积和交积 279
7.10 扫描转换多边形 280
7.11 三维剪取 288
7.12 最后绘制管道 291
7.13 小结 299
第 8 章 纹理贴图与照明 301
8.1 透视正确纹理贴图的基础 301
8.2 简单纹理贴图器 305
8.3 优化纹理贴图 314
8.3.1 纹理的存储格式 315
8.3.2 原始优化 319
8.3.3 内联方法 322
8.3.4 快速纹理贴图演示程序 323
8.4 简单灯光 324
8.4.1 扩散反射 325
8.4.2 环境光 325
8.4.3 光源强度 325
8.4.4 距离减弱 325
8.4.5 实现点光源 326
8.5 实现纹理照明 327
8.6 使用阴影图的高级照明 334
8.6.1 求出边界矩形 335
8.6.2 采用色档图 337
8.6.3 建立色档图 338
8.6.4 建立表面 341
8.6.5 缓存表面 344
8.6.6 阴影表面演示程序 351
8.7 其他概念 352
8.7.1 深度提示 352
8.7.2 伪阴影 352
8.7.3 MIP 贴图 352
8.7.4 双线性插值 352
8.7.5 三线性插值 352
8.7.6 法线图与深度 353
8.7.7 其他类型的光 353
8.8 小结 353
第 9 章 三维对象 354
9.1 表面消隐 354
9.1.1 油画算法 354
9.1.2 反油画算法 355
9.1.3 Z 缓存 355
9.1.4 使用 1/Z 的 Z 缓存 356
9.1.5 计算 Z 深度 359
9.2 三维动画 360
9.2.1 空间运动 362
9.2.2 角度运动 364
9.3 多边形组 369
9.4 从 OBJ 文件装入多边形组 375
9.4.1 OBJ 文件格式 375
9.4.2 MTL 文件格式 382
9.5 游戏实体 384
9.6 管理游戏实体 387
9.7 综合起来 389
9.8 改进 396
9.9 小结 397
第 10 章 用 BSP 树管理三维视景 398
10.1 BSP 树简介 398
10.2 二叉树基础 399
10.3 一维 BSP 树 401
10.4 二维 BSP 树 403
10.4.1 建立 BSP 树 404
10.4.2 BSP 树遍历实例 406
10.5 实现二维 BSP 树 407
10.5.1 BSP Line 409
10.5.2 确定点相对于直线的边 409
10.5.3 BSP 多边形 413
10.5.4 遍历 BSP 树 413
10.5.5 中序遍历 415
10.5.6 由前到后遍历 416
10.5.7 树的建立 417

10.5.8 两条线求交 421
10.5.9 用直线剪取多边形 423
10.5.10 删除 T 形接头 426
10.5.11 测试 BSP 树 427
10.6 从前到后画多边形 429
10.7 第一个 BSP 树例子 438
10.8 在视景中画实体 439
10.9 从文中装入贴图 440
10.10 综合起来 444
10.11 改进 445
10.12 小结 446
第 11 章 碰撞探测 447
11.1 碰撞基础 447
11.2 实体间碰撞 448
11.2.1 减少测试 448
11.2.2 边框球 450
11.2.3 边框圆柱 452
11.2.4 离散时间问题 454
11.3 实体与世界碰撞 455
11.3.1 测试地板的边框 455
11.3.2 寻找特定位置的 BSP 叶 456
11.3.3 实现地板和天花板高度测试 457
11.3.4 测试墙的边框 459
11.3.5 BSP 树的线段交点 460
11.3.6 角问题 464
11.3.7 实现实体/世界碰撞探测 464
11.4 基本碰撞探测演示程序 467
11.5 滑动与碰撞处理 467
11.5.1 实体间滑动 468
11.5.2 实体沿墙的滑动 470
11.5.3 重力与爬楼梯（实体沿地面滑动） 473
11.5.4 实现跳动 476
11.6 碰撞探测与滑动演示程序 477
11.7 改进 477
11.8 小结 477
第 12 章 路径搜索 478
12.1 路径寻找基础 478
12.2 初步路径寻找方法 478
12.3 A*算法基础 482
12.4 采用 A*算法 487
12.5 BSP 树与 A*算法 487
12.5.1 门户 488
12.5.2 实现门户 489
12.6 一般性路径寻找 493
12.7 建立 PathBot 496
12.8 改进 A*搜索 500
12.9 小结 500
第 13 章 人工智能 502
13.1 人工智能基础 502
13.2 减少 Bot 的神力 503
13.2.1 视觉 503
13.2.2 听觉 505
13.3 状态机与响应 507
13.4 概率机 509
13.5 决策 512
13.6 模式 513
13.6.1 躲避 514
13.6.2 攻击 517
13.6.3 跑开 519
13.6.4 瞄准 520
13.6.5 射击 521
13.7 实体派生 522
13.8 综合起来 523
13.8.1 大脑 523
13.8.2 健康与死亡 524
13.9 进化 532
13.9.1 再生 532
13.9.2 进化 Bot 534
13.9.3 演示改进 538
13.10 进化 538
13.11 小结 539
第 14 章 游戏脚本 540
14.1 关于脚本 540
14.2 实现接触与释放通知 541
14.3 游戏实体监听器 544
14.4 脚本 548
14.4.1 设计脚本 549

14.4.2 嵌入 BeanShell 解释器.....551
14.5 延迟事件.....556
14.6 综合起来.....560
14.7 改进.....562
14.8 小结.....563
第 15 章 保存游戏.....564
15.1 游戏保存基础.....564
15.2 对游戏状态保存使用 Java 序列化 API.....565
15.2.1 序列化简介.....565
15.2.2 序列化基础.....565
15.2.3 序列化规则.....567
15.2.4 序列化问题.....570
15.2.5 覆盖缺省行为.....573
15.3 生成游戏屏幕图形.....577
15.3.1 生成屏幕略图图形.....579
15.3.2 保存图形.....579
15.4 将游戏保存到正确目的地.....581
15.5 小结.....582
第三部分 调整与完成游戏
第 16 章 优化技术.....583
16.1 优化规则.....583
16.2 剖析.....584
16.2.1 基准测试.....584
16.2.2 使用 HotSpot 剖析器.....584
16.3 HotSpot.....588
16.3.1 Java 特定优化.....589
16.3.2 死亡代码消除.....589
16.3.3 循环不变项提升.....589
16.3.4 公共子表达式消除.....590
16.3.5 常量传递.....590
16.3.6 内联方法.....590
16.4 优化技巧.....591
16.4.1 算法.....591
16.4.2 移位.....592
16.4.3 求模.....593
16.4.4 乘法.....593
16.4.5 指数.....593
16.4.6 更多循环不变项提升.....594
16.4.7 查找表.....594
16.4.8 定点算术.....597
16.4.9 异常.....597
16.4.10 输入/输出.....597
16.4.11 内存贴图文件.....598
16.5 内存使用与回收.....598
16.5.1 Java 堆与内存回收.....599
16.5.2 监视内存回收.....599
16.5.3 监视内存使用.....600
16.5.4 调整堆.....604
16.5.5 调用内存回收单元.....605
16.5.6 减少对象生成.....605
16.5.7 对象复用.....605
16.5.8 对象池.....606
16.6 感觉性能.....606
16.7 小结.....611
第 17 章 生成游戏图形与声音.....613
17.1 选择外观.....613
17.1.1 寻找灵感.....613
17.1.2 保持一致.....614
17.2 取得自由版权游戏媒体.....614
17.3 与艺术家和声效工程师合作.....615
17.4 工具.....615
17.5 生成声音.....615
17.6 生成纹理与幽灵.....616
17.6.1 图形文件格式.....617
17.6.2 生成无缝纹理.....617
17.6.3 生成替换纹理.....618
17.6.4 生成过渡纹理.....619
17.6.5 生成多层纹理.....620
17.7 生成漂亮屏幕与 HUD 图形.....621
17.8 生成 UI 图形.....621
17.9 生成字体.....622
17.10 小结.....628
第 18 章 游戏设计与最后 10%.....629
18.1 最后 10%.....629
18.1.1 效果.....630
18.1.2 游戏状态机.....630
18.2 游戏设计要素.....636

18.2.1 环境.....636
18.2.2 故事要素.....637
18.2.3 玩游戏.....638
18.2.4 告诉用户怎么玩.....639
18.3 生成贴图编辑器.....639
18.4 调试.....640
18.4.1 调试 Java2D 问题.....642
18.4.2 日志.....643
18.5 保护代码.....644
18.6 游戏部署.....645
18.7 Java Web Start 游戏部署.....646
18.7.1 从 jar 文件取得资源.....646
18.7.2 签名 jar 文件.....647
18.7.3 生成 JNLP 文件.....647
18.7.4 设置 Web 服务器.....649
18.8 自然编译游戏部署.....650
18.9 更新与补丁.....650
18.10 带宽问题.....651
18.11 反馈与 Beta 测试.....651
18.12 赚钱.....652
18.13 综合起来.....652
18.14 小结.....653
第 19 章 未来.....654
19.1 Java 的演变.....654
19.1.1 Java 社团.....654
19.1.2 缺陷天堂（Bug Parade）.....654
19.2 Java 1.5（Tiger）.....655
19.2.1 一般（JSR 14）.....655
19.2.2 枚举（JSR 201）.....656
19.2.3 静态导入(JSR 201).....657
19.2.4 改进 for 循环(JSR 201).....657
19.2.5 编译器 API（JSR 199）.....658
19.2.6 网络传输格式（JSR 200）.....658
19.2.7 共享虚拟机（JSR 121）.....658
19.3 Java 平台需要什么.....658
19.3.1 更多鼠标与键盘输入选项.....658
19.3.2 游戏杆输入.....659
19.3.3 硬件加速半透明图像.....659
19.3.4 高精度定时器.....659
19.3.5 Linux 中的硬件加速图形和全屏方式.....660
19.3.6 运行环境中的硬件加速三维特性.....660
19.3.7 HotSpot SIMD 优化.....660
19.3.8 更多字体光顺选项.....661
19.3.9 其他情况.....662
19.4 新设备与 Java Games Profile (JSR 134).....662
19.5 小结.....663

第一部分　Java 游戏基础

第 1 章　Java 线程
第 2 章　二维图形与动画
第 3 章　交互性与用户界面
第 4 章　声效与音乐
第 5 章　创建二维平台游戏
第 6 章　多人游戏

第 1 章　Java 线程

关键主题

- 什么是线程
- 在 Java 中生成与运行线程
- 同步
- 使用 wait()与 notify()
- Java 事件模型
- 何时使用线程
- 何时不使用线程
- 线程池
- 小结

中午你找到一家喜欢的餐厅去吃饭。当坐在位子上环顾四周，发现餐厅里很空，除了你之外只有一位服务员和另一位食客。此时，服务员正在为另一个客人服务，奇怪的是你被冷落了。虽然你向他要求菜单，但服务员仍然不理不睬，只是给另一客人添加啤酒。最后，当那位客人离开了餐厅后，服务员才确认你的存在，并过来为你服务。

问题出在哪里呢？在于没有把这位服务员开除，呵呵！

实际上，这里的服务员不能进行多任务（multitask）。多任务服务员可以同时为 2 个或多个客户服务，而不是对客户逐个进行服务。多任务服务员和计算机的线程（threads）问题大致相近，本章就将介绍 Java 线程是如何工作的，以及线程间同步的问题，其中会介绍相关的要点和技巧。

1.1　什么是线程

可以把服务员看成计算机的处理器，把客户看成任务（task）。每个任务在一个线程中运行，而现代的处理器能够并发运行多个线程，如可以从 Internet 网上下载文件，并同时编

写一篇论文。

下面介绍并发运行多个线程的原理。首先把每个线程都分解成多个小块，称线程的并发性（concurrency）。然后让其中的线程在少量的时间段（time slices）上运行。在下一个时间段前，该线程将被占先（per-empted），好让另一个线程开始运行。依次类推，多线程实现原理如图 1-1 所示，由于每个时间段非常小，所以看起来众多任务是同时发生的。

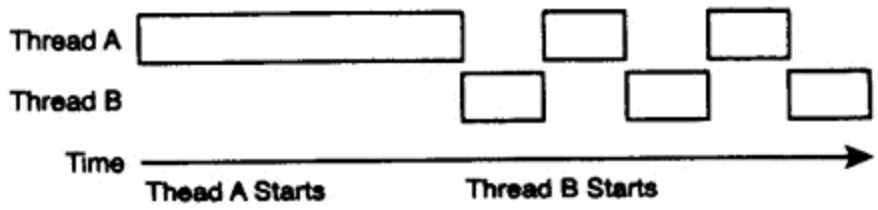


图 1-1　处理器利用并发性运行多个线程

在具有多个处理器的计算机上，众多的线程可能真的同时运行。不过还要取决于 Java 虚拟机的实现方法。

1.2　在 Java 中生成与运行线程

由于 Java 的设计理念考虑到了线程的问题，所以在 Java 中实现线程要比使用其他语言实现线程会更加方便。若要生成和启动新线程，只要生成 Thread 对象的实例并调用 start()方法即可，实现的代码如下所示。

```
Thread myThread =new Thread();
MyThread.start();
```

当然，这段代码干不了任何事情，因为线程还没有被指定任务。当 Java 虚拟机生成新的系统线程并且启动后，就要调用 Thread 对象的 run()方法。但在缺省情况下，run()方法什么也不干，所以线程就等于死亡了。

如果想让 run()方法干点事情，下面就来介绍对线程指定任务的 3 种基本方法：

扩展 Thread 类；

实现 Runnable 接口；

使用匿名内类。

1.2.1　扩展 Thread 类

对线程指定任务的第一个方法就是扩展 Thread 类，覆盖其中的 run()方法，实现代码如下所示。

```
public class MyThread extends Thread{
    public void run(){
        System.out.println("Do something cool here.");
    }
}
```

当然还需要生成和启动线程，实现代码如下所示。

```
Thread myThread =new Thread();
```

```
MyThread.start();
```

现在就形成2个线程：主线程和刚刚启动的线程。

1.2.2 实现Runnable接口

扩展Thread类很简单，但通常是不会为启动线程而写一个新类的，必定实现类扩展也需要运行一个线程来操作。下面介绍实现Runnable接口来对线程指定任务，实现代码如下所示。

```
public class MyClass extends SomeOtherClass implements Runnable{
public MyClass(){
Thread thread = new Thread(this)
Thread. start();
            }
            public void run(){
                System.out.println("Do something cool here.");
        }
}
```

代码中 MyClass 的构造函数启动一个新线程，而 Thread 类的构造函数会引用一个Runnable对象作为参数，这个Runnable对象就是启动线程时执行的任务。

1.2.3 使用匿名内类

假如不能生成新类或不便实现Runnable接口，那么就可以使用匿名内类来对线程指定任务，实现代码如下所示。

```
new Thread（）{
    public void run(){
            System.out.println("Do something cool here.");
        }
} . start();
```

代码十分简单，但如果run()方法太长，则会大大降低可读性，所以要谨慎使用。

1.2.4 等待线程完成

若让当前线程去等待另一个线程，则可以使用join()方法，实现代码如下所示。

```
myThread.join();
```

join()方法应用于玩家退出游戏时，它可以保证在游戏中所有的线程运行完后，再进行最后的清理工作。

1.2.5 休眠线程

由于线程过于繁重，那么可以让它歇息，不过只是让线程暂停一会儿，可以使用静态sleep()方法，实现代码如下所示。

```
Thread.sleep(1000);
```

sleep()方法使当前运行的线程休眠 1000 毫秒（或指定时间量）。休眠线程不会占用任何 CPU 资源，可以说连梦都不做。

1.3 同步

妙哉，现在可以运行一些线程来完成各种奇妙的工作了。但是，并不是到处都有阳光和鲜花，假设多个线程都要访问同一个对象或变量，那么就会遇到线程同步的问题。

1.3.1 为什么要同步

假设要创建一个迷宫游戏 Maze，游戏中任何线程都可以设置游戏者的位置和检查游戏者是否退出，为了简单起见，假设退出条件是 x=0,y=0，请看下面的实例代码。

```
public class Maze{
        private int playerX;
        private int playerY;
        public boolean isAtExit(){
            return (playerX ==0 && playerY ==0);
        }
        public void setposition(int x,int y){
            playerX = x
            playerY = y
        }
}
```

通常上述代码能够顺利工作，但是要记住，线程是随时可能被占先的。假设游戏者从位置（1，0）移动到（0，1），下面来看看代码的执行情况。

（1）开始，对象变量 playerX=1，playerY=0。

（2）线程调用 setPosition(0,1)。

（3）执行语句 playerX = x；之后，playerX =0。

（4）线程 A 被线程 B 占先。

（5）线程 B 调用 isAtExit()。

（6）这时 playerX =0，playerY =0，因此 isAtExit()返回 true 结果。

表面看游戏者已经解完迷宫。其实不然，因为走出迷宫就需要保证 setPosition()和 isAtExit()方法不能同时执行，你做到了吗？

1.3.2 如何同步

下面通过关键字 synchronized 来实现同步问题。在迷宫实例中，实现方法进行同步，则一次只能运行一个方法，下面就是更新后 Maze 的代码。

```
public class Maze{
    private int playerX;
    private int playerY;
```

```
        public synchronized boolean isAtExit(){
            return(playerX ==0 && playerY ==0);
        }
    public synchronized void setPosition(int x,int y){
playerX = x
            playerY = y
}
  }
```

当 Java 虚拟机执行同步方法时，会取得对象的锁（lock），且对象一次只提供一个锁。当方法执行完毕并正常返回或抛出异常时，程序会将锁释放。

当一个同步方法拥有锁时，其他同步方法都要等待当前锁释放之后才能运行。可以把锁看成房门上的锁，一次只有一个人在房间里，而这个人离开时或异常情况下从窗口中跳出时，门锁将被打开。

除了将方法同步之外，还有就是将对象同步。对象同步也可以把任何对象看成锁，而方法同步中的锁则是当前的实例（this）。方法同步实际上是具有 this 的简化对象同步，对于上述迷宫代码，下面 2 个代码实际上是相同的。

```
public synchronized void setPosition(int x,int y){
playerX = x
            playerY = y
}

public void setPosition(int x,int y){
        setPosition(this){
playerX = x
            playerY = y
        }
}
```

惟一的不同就是第 2 个例子多了一些字节码指令。

对象同步适用于需要多个锁，并且要取得 this 以外的锁和不需要同步方法的情况。锁可以是任何对象，甚至是数组。简单地说，可以是除基本类型以外的一切。如果要建立自己的锁，只要生成普通的 Object 即可，实现代码如下所示。

```
Object   myLock = new Object();
...
synchronized (myLock){
    ...
}
```

1.3.3　同步是什么

如何理解同步呢？这个问题就是任何时候都能保证 2 个或以上的线程访问同一个对象

或字段。

1.3.4 不同步是什么

何时让代码实现同步呢，一般原则是不要过份的同步（oversynchronize），即只对需要同步的部分进行同步。否则会在线程执行同步代码时，造成不必要的延迟。

如果要同步方法中的一部分，那么只要在关键代码段的周围放上同步块即可，实现代码如下所示。

```
public void myMethod(){
        synchronized(this){
                //需要同步的代码
        }
//已经线程安全的代码
}
```

使用局部变量的方法则不需要同步。因为局部变量存放在堆栈中，而且每个线程都有自己的堆栈，所以不会遇到同步的问题。如下面的实例使用了局部变量，实例代码如下所示。

```
public int square(int n){
      int s = n * n;
      return s;
}
```

当同步代码不被多个线程访问时将不够成问题，但如果知道有些代码只有一个线程访问，若不进行同步，则一定要在 JavaDoc 文档中指出，因为代码不是线程安全的。

如果无法确定有哪些线程访问代码，那么只要在控制台上打印当前运行的线程名即可知道，实现代码如下所示。

```
System.out.println(Thread.currentThread().getName());
```

1.3.5 避免死锁

死锁就是 2 个线程相互等待对方完成某个事项而形成停滞状态，请观察下面线程的执行步骤。

（1）线程 A 取得锁 1。

（2）线程 B 取得锁 2。

（3）等待释放锁 1。

（4）线程 A 等待释放锁 2。

可以看出 2 个线程都在等待对方的锁，所以它们将永远停滞，即死锁。死锁会在多个线程以不同顺序取得多个锁时出现，那如何避免死锁呢？最好的方法就是防止同步代码发生死锁，只要考虑哪个线程按什么顺序取得什么锁，认真考虑每种可能的情形即可。

如果发现游戏因为死锁而停滞，则可以用几种方法进行探测。在 HotSpot 1.4.1 版虚拟机中，Sun 公司提供了死锁探测器。从控制台运行游戏直到发现停滞时，按 Ctrl+\键(或 Windows 中的 Ctrl+break 键)，虚拟机会显示线程状态的信息，是否等待什么和发现死锁。

1.4 使用 wait()与 notify()

假设 2 个线程需要相互交流，设线程 A 等待线程 B 发一个消息，实现代码如下所示。

```
// Thread A
public void waitForMessage(){
    while (hasMessage == false){
Thread.sleep(100);
        }
}
//Thread B
public void setMessage(String message){
    ...
    hasMessage = true;
}
```

虽然方案可行，但是非常笨拙。因为线程 A 每 100 毫秒要检查线程 B 是否发送消息，即每秒钟要检查 10 次。若线程 A 睡过头（oversleep），将可能太迟得到消息。另外，如果几个线程都等一个消息呢？

更好的方法是线程 A 保持闲机，在线程 B 发消息时再通知线程 A。wait()和 notify()方法提供了这个功能。

wait()方法在同步代码块中使用。执行时，wait()方法将释放锁和线程等待通知。notify()方法也在同步代码块中使用。它通知等待同一锁的一个线程，若多个线程都在等待一个锁，那么就随机通知其中的一个。

下面是线程 A 等待线程 B 发一个消息实例的更新后的代码。

```
//Thread A
pulbic synchronized void waitForMessage(){
    try{
        wait();
    }
    catch (InterruptedException ex){}
}
 //Thread B
pulbic synchronized void setMessage(String message){
    ...
    notify();
}
```

可以看出，线程 B 调用 notify()方法并离开同步方法后（释放 this 的锁），线程 A 取得锁并完成同步代码块，这里它直接返回。

此外，还可以选择等待最大的时间量。wait()方法可以将等待最大的时间量作为参数，实现代码如下所示。

wait(100)

如果线程一直没有收到通知，则等于是让线程休眠指定时间量。但是，无法判断线程返回是因为超时还是线程收到了通知。另一个通知方法 notifyAll（），它通知等待同一锁的所有线程而不只是一个线程。

wait()，Notify（）和 notifyAll（）方法都是 Object 类的方法，任何 Java 对象都有它们，就像任何 Java 对象都可以作为锁一样。

1.5 Java 事件模型

也许认为自己的游戏里只有一个线程，所以不必考虑同步问题。但其实不然，因为所有图形应用程序至少有 2 个可以运行代码的线程：主线程和 AWT 事件派生线程。

主线程就是程序中的“主”线程，它用主类的 main()方法开始执行。AWT 事件派生线程处理用户输入事件，如鼠标单击、键击和窗口缩放等。第 3 章“交互性与用户界面”将介绍这些用户输入事件，目前只要知道它们是通过 AWT 事件派生线程来访问代码的。

1.6 何时使用线程

线程为用户提供更有趣的体验。这意味着当某些代码需要停滞或要较长时间运行时，代码就可以在另一个线程中运行，这样可以使用户不会觉得游戏在停滞。

下面列举一些需要使用线程的情形：

从本地文件系统装入大量文件；

进行任何网络通信时，如将高分成绩发送到服务器中；

进行任何海量计算时，如生成地形。

1.7 何时不使用线程

任何游戏中都可以看到许多同时发生的事件，如敌人在奔跑，门在打开，子弹在飞等。如果认为它们都是在单独的线程中运行，那么就错了。因为这不仅浪费资源，而且会因为同时运行太多线程而使系统性能大大下降，甚至会导致如下问题的产生：

某个敌人可能处于绘图操作的中间，临时出现一个敌人会同时出现在 2 个不同地方的情形；

每个线程的时间段不平衡，造成运动不连贯或不一致；

同步代码可能造成不必要的延迟。

如果想了解让众多个事件同时发生的方法，请参见第 2 章“二维图形与动画”的介绍。

1.8 线程池

在介绍完 Java 线程信息后，下面就要生成一些有用的项目：线程池（thread pool）。顾名思义，它是执行所有任务的一组线程，也许你要限制同时网络连接或 I/O 连接所用的线程数，或要对处理器工作量很大的任务控制最大线程数。

清单 1.1 中的 Threadpool 类可以选择线程池中的线程数和运行定义为 Runnables 的任务。下面的实例将展示如何利用 Threadpool 类生成具有 8 个线程的线程池，然后运行一个简单的任务，直到等待任务的完成，实现代码如下所示。

```
Threadpool myThreadpool =new Threadpool(8);
MyThreadpool.runTask(new Runnable(){
        Public void run(){
            System.out.println("Do something cool here.");
        }
});
myThreadpool.join();
```

从上述代码中可以看出，runTask()方法返回是问题的关键。如果线程池中所有线程都忙于执行自己的任务，则程序会调用 runTask()方法把新的任务存放在队列中，直到有线程来运行它。

清单 1.1　ThreadPool.java

```
import java.util.LinkedList;

/**
    线程池是一组线程，限制执行任务的线程数
*/
public class ThreadPool extends ThreadGroup {

    private boolean isAlive;
    private LinkedList taskQueue;
    private int threadID;
    private static int threadPoolID;

    /**
        创建新的线程池
        @param numThreads 池中的线程数
    */
    public ThreadPool(int numThreads) {
        super("ThreadPool-" + (threadPoolID++));
        setDaemon(true);

        isAlive = true;

        taskQueue = new LinkedList();
        for (int i=0; i<numThreads; i++) {
```

```
            new PooledThread().start();
        }
    }

    /**
        请求新任务。这个方法立即返回，任务在池中下一闲机线程中执行
        <p>任务按收到的顺序开始执行
        @param task 要运行的任务，null 表示不采取操作
        @如果这个线程池已经关闭，则抛出 IllegalStateException
    */
    public synchronized void runTask(Runnable task) {
        if (!isAlive) {
            throw new IllegalStateException();
        }
        if (task != null) {
            taskQueue.add(task);
            notify();
        }
    }

    protected synchronized Runnable getTask()
        throws InterruptedException
    {
        while (taskQueue.size() == 0) {
            if (!isAlive) {
                return null;
            }
            wait();
        }
        return (Runnable)taskQueue.removeFirst();
    }

    /**
        关闭这个线程池并立即返回。所有线程停止，不执行任何等待的任务。关闭一个线程池之
        后，这个线程池上不能再运行任何任务
    */
    public synchronized void close() {
        if (isAlive) {
            isAlive = false;
```

```
            taskQueue.clear();
            interrupt();
        }
    }

    /**
        这个线程池并等待所有运行线程完成，执行任何等待的任务
    */
    public void join() {
        // 告诉所有等待线程，这个线程池不再活动
            synchronized (this) {
                isAlive = false;
                notifyAll();
            }

        // 等待所有线程完成
            Thread[] threads = new Thread[activeCount()];
            int count = enumerate(threads);
            for (int i=0; i<count; i++) {
                try {
                    threads[i].join();
                }
                catch (InterruptedException ex) { }
            }
    }

    /**
        ooledThread 是线程池组中的线程，用于运行任务(Runnables)
    */
    private class PooledThread extends Thread {
        public PooledThread() {
            super(ThreadPool.this,
                "PooledThread-" + (threadID++));
        }

        public void run() {
            while (!isInterrupted()) {

                // 取得要运行的任务
```

```
            Runnable task = null;
            try {
                task = getTask();
            }
            catch (InterruptedException ex) { }

        // 如果 getTask() 返回 null 或中断，则关闭这个线程并返回。
            if (task == null) {
                return;
            }

            // 运行任务，吸收共抛出的任何异常
            try {
                task.run();
            }
            catch (Throwable t) {
                uncaughtException(this, t);
            }
        }
    }
    }
}
```

到目前为止，Threadpool 类的使用概念中还有一个未介绍的人物，它就是 ThreadGroup 类。ThreadGroup 包括一组线程和一些修改线程的方法，如下列代码中，使用 ThreadGroup 类的 interrupt()方法用于在关闭线程池时停止所有等待的线程。

下面用清单 1.2 所示的 ThreadPoolTest 类测试 ThreadPool 对象。测试中，ThreadPoolTest 会在线程池中启动几个线程，并指定几个任务让线程池去运行，不过这里的每个任务只是向控制台打印一个消息。

因为程序要取得任务数和线程数的变元，所以在运行测试程序时，要在命令行中输入如下代码。

```
java ThreadPoolTest 8 4
```

这样在线程池中，将使用 4 个线程和 8 个任务运行测试程序。当然可以任意的组合任务和线程。

清单 1.2　ThreadPoolTest.java

```
public class ThreadPoolTest {

    public static void main(String[] args) {
        if (args.length != 2) {
```

```
        System.out.println("Tests the ThreadPool task.");
        System.out.println(
            "Usage: java ThreadPoolTest numTasks numThreads");
        System.out.println(
            "    numTasks - integer: number of task to run.");
        System.out.println(
            "    numThreads - integer: number of threads " +
            "in the thread pool.");
        return;
    }
    int numTasks = Integer.parseInt(args[0]);
    int numThreads = Integer.parseInt(args[1]);

    // 生成线程池
    ThreadPool threadPool = new ThreadPool(numThreads);

    // 运行例子任务
    for (int i=0; i<numTasks; i++) {
        threadPool.runTask(createTask(i));
    }

    //关闭池并等待完成所有任务
    threadPool.join();
}

/**
    生成简单可运行任务，打印一个 ID，等待 500 毫秒，然后再次打印这个 ID
*/
private static Runnable createTask(final int taskID) {
    return new Runnable() {
        public void run() {
            System.out.println("Task " + taskID + ": start");

            // 模拟长时间任务
            try {
                Thread.sleep(500);
            }
            catch (InterruptedException ex) { }
```

```
                System.out.println("Task " + taskID + ": end");
            }
        };
    }
}
```

由于向控制台打印一个消息几乎不需要时间，所以 ThreadPoolTest 类通过休眠 500 毫秒来模拟时间任务。

1.9 小结

这就是线程！本章介绍了线程的基本工作方法和用法，现在你可以保证游戏中没有线程同步和死锁的问题了。此外，还强调 Java 游戏中不可能不涉及线程，因为所有 Java 图形应用程序至少要有 2 个线程。最后，生成的一般性 ThreadPool 类，它可以在许多情形中使用。

第 2 章　二维图形与动画

关键主题

- 全屏图形
- 图像
- 避免闪烁与裂开
- 简单效果
- 小结

首先确认在 Java 中创建的 3 种图形游戏：小程序游戏、窗口游戏和全屏游戏。

Java 小程序游戏是 Web 浏览器中运行的，好处是用户不必安装任何程序。但是，游戏者要联机并运行 Web 浏览器。而且小程序有几个安全限制，为了防止恶意代码造成的伤害，如小程序无法在用户硬盘上保存选项和游戏之类的信息。另外，小程序只能对小程序所在的服务器建立网络连接。

Java 窗口游戏没有小程序一样的安全限制。事实上，窗口游戏的外观与行为和正常应用程序一样，有标题栏、关闭按钮等。但是，这些用户界面元素可能分散游戏者的注意力，特别是希望游戏者沿漫到游戏中时更为不利。

Java 全屏游戏可以完全控制游戏的外观显示，没有标题栏、任务栏和其他用户界面元素，可以让游戏者完全沿漫到游戏中。

由于本书的重点是介绍全屏游戏，所以本章将介绍全屏图形、动画与幽灵等技术。由于还没有介绍键盘和鼠标的输入（详见第 3 章“交互性与用户界面”），所以本章还要创建一些简单的图形演示程序。

2.1　全屏图形

下面将简要介绍硬件的工作原理，然后再介绍其编程理念。

显示硬件分 2 个部分：显示卡和显示器。显示卡的内存中将存储屏幕上的内容，并有几个修改显示内容的功能。同样，显示卡还要将内存中的内容推到显示器的屏幕上，而显示器则按显示卡的指示来显示内容。

2.1.1　屏幕布局

显示器的屏幕可以用相同尺寸的微小彩色像素表示。像素（pixel）一词是从图形元素（picture element）衍生出来，表示显示器显示的一个光点。屏幕上的垂直与水平像素个数称为屏幕的分辨率（resolution）。

屏幕的原点在屏幕的左上角，如图 2-1 所示。显示内存中存放像素和书本一样，从左上角开始，从左到右，从上到下，直到右下角。屏幕上的位置可以表示成（x,y），其中 x 是距离原点的水平像素个数，y 是距离原点的垂直像素个数。

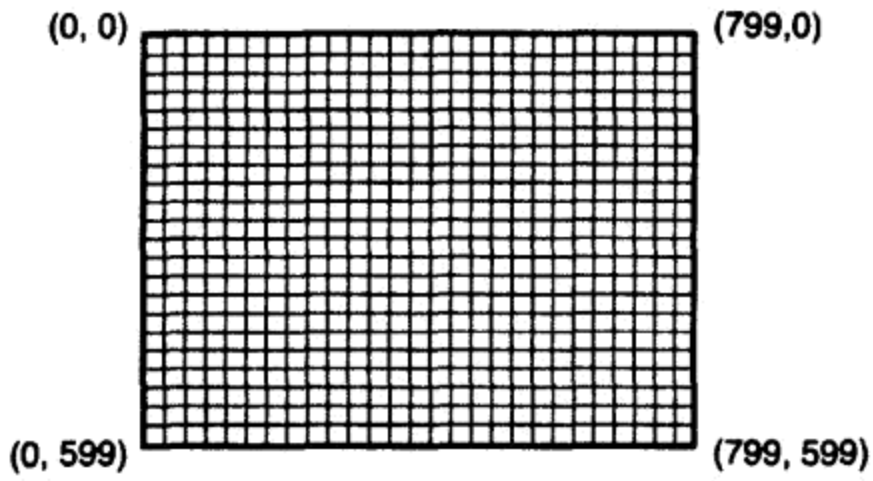


图 2-1 显示器的屏幕分成相同尺寸的微小彩色像素，
屏幕的原点在屏幕左上角。图中描述 800x600 屏幕

屏幕分辨率的选择取决于显示卡和显示器的功能，典型的屏幕分辨率为 640×480,800×600,1024×768 与 1280×1024。

典型显示器和电视机的显示尺寸比例都是 4:3，即显示高度为宽度的 3/4。一些新式显示器屏幕较宽，显示尺寸比例是 3:2 或 16:10。相比而言，宽银幕电影的显示尺寸比例是 16:9。

旧式大型 CRT（阴极射线管）显示器能够整齐地显示各种不同的屏幕分辨率，因为 CRT 是用电子来绘制像素的。便携电脑与新式计算机系统使用的簿型 LCD（液晶显示器）则不然，因为液晶显示器上的每个像素是用物理晶体管照明的，因此液晶显示器有一个自然分辨率。液晶显示器的设计只能显示自然分辨率，而对于非自然分辨率可能出现结块或模糊等现象。因此，最好保证能用 2 种或 3 种不同的分辨率来运行游戏，使游戏者可以选择系统中最合适的屏幕分辨率。

2.1.2 像素颜色与位深度

小时候曾经学过 3 种基本色，它们是红色、黄色和蓝色。由于黄色和蓝色相加可以得到绿色，基于此原理，绘图和使用其他物理媒介时，可以将这 3 种颜色组合起来生成所要的任何颜色，这就是所谓的“减成颜色模型”，当 3 种颜色都不存在时就是白色（实际上，这不是精确的颜色模型，现代打印机会使用更复杂的颜色模型，即使用青、紫、黄、黑 4 种颜色）。

计算机显示器与电视机的工作原理差不多。显示器组合红、蓝、绿 3 种颜色来生成所要的任何颜色。显示器不用油和墨之类物理媒介，而利用发射的光束，因此 RGB 颜色模型是“加成颜色模型”，即所有颜色相加得到白色。

显示器可以显示的颜色数量取决于显示方式的位深度，常见的位深度为 8,15,16,24 与 32 位。

- 8 位颜色有 2^8=256 种颜色。根据调色板情况，一次只能显示 256 种颜色。但是，Java 中目前没有改变调色板的方法，而且规范中也没有指定这些颜色。Java 运行环境可以使用 Web 安全调色板，其中有 216 种颜色，红、蓝、绿 3 种各有 6 个值（即 6x6x6=216）。
- 15 位颜色的红、蓝、绿 3 种各有 5 位，共 2^{15}=32768 种颜色。
- 16 位颜色的红、蓝各 5 位，绿色为 6 位，共 2^{16}=65536 种颜色。

- 24 位颜色的红、蓝、绿 3 种各有 8 位，共 $2^{24}=16777216$ 种颜色。
- 32 位颜色与 24 位颜色相同，但增加 8 位，使像素数据在 32 位边界上对齐。

大多数现代显示卡都支持 8 位、16 位和 32 位方式。由于人眼可以分辨出一千多万种颜色，因此 24 位颜色是很理想的。16 位颜色比 24 位颜色稍快一些，因为传输的数据更少，颜色质量没有那么精确。

2.1.3　刷新率

虽然显示器好象在显示实体图像，但实际上每个像素在几毫秒之后将会消失。为此，显示器要连续刷新图像来防止消失。显示器的刷新速度称为刷新率（refresh rate），它的单位为赫兹（Hz），表示每秒钟循环的次数。人眼通常适合的刷新率为 75Hz 到 85Hz 之间。

2.1.4　将显示切换到全屏方式

介绍分辨率、位深度和刷新率之后，就要编写一些代码。下面将利用几个对象来实现将显示切换到全屏方式：

- Window 对象是屏幕显示内容的抽象，可以看成绘图的画布。通常实际上将使用 Jframe 对象，它是 Window 类的子类，也可以用于建立窗口应用程序。
- DisplayMode 对象是指定将显示切换到何种分辨率、位深度和刷新率。
- GraphicsDevice 对象用来改变显示方式和检查显示属性，可以看作是显示卡的界面。GraphicsDevice 对象从 GraphicsEnvironment 对象中取得。

下面就是将显示切换到全屏方式实例的代码。

```
JFrame window = new JFrame();
DisplayMode displayMode = new DisplayMode( 800, 600, 16, 75);

//取得 GraphicsDevice
GraphicsEnvironment environment =
    GraphicsEnvironment.getLocalGraphicsEnvironment();
GraphicsDevice device = environment.getDefaultScreenDevice();

//用 JFrame 作为全屏窗口
device.setFullscreenWindow(window);

//改变显示方式
device.setDisplayMode(displayMode);
//下面要将显示切换到原先的显示方式时，可以将全屏 Window 设置为 null：//
device.setFullScreenWindow(null);
```

注意：代码并不完整，所以有些系统是不能改变显示方式的。如果没有改变，setDisplayMode()方法将会抛出 IllegalArgumentException 屏幕。

在缺省情况下，即使将显示切换到全屏方式，JFrames 仍然会显示边框和标题栏。为了处理这些问题和简化功能，这里生成一个包装类 SimpleScreenManager，它的代码如清单 2.1

所示。SimpleScreenManager 是将显示切换到全屏方式的简单界面，它在幕后使用 setUndecorated(true)方法捕获异常并删除 Jframes 的边框和标题栏。另外，屏幕恢复时，会处理 Jframes。

清单 2.1 SimpleScreenManager.java

```
import java.awt.*;
import javax.swing.JFrame;

/**
    SimpleScreenManager 类管理初始化并
*/
public class SimpleScreenManager {

    private GraphicsDevice device;

    /**
        生成新 SimpleScreenManager 对象
    */
    public SimpleScreenManager() {
        GraphicsEnvironment environment =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        device = environment.getDefaultScreenDevice();
    }

    /**
        进入全屏方式和改变显示方式
    */
    public void setFullScreen(DisplayMode displayMode,
        JFrame window)
    {
        window.setUndecorated(true);
        window.setResizable(false);

        device.setFullScreenWindow(window);
        if (displayMode != null &&
            device.isDisplayChangeSupported())
        {
            try {
                device.setDisplayMode(displayMode);
```

```
            }
            catch (IllegalArgumentException ex) {
                    // 忽略一本设备中为非法方式
            }
        }
    }
    /**
        将当前使用的窗口返回全屏方式
    */
    public Window getFullScreenWindow() {
        return device.getFullScreenWindow();
    }

    /**
        恢复屏幕显示方式
    */
    public void restoreScreen() {
        Window window = device.getFullScreenWindow();
        if (window != null) {
            window.dispose();
            }
            device.setFullScreenWindow(null);
        }
    }
```

下面就要测试 SimpleScreenManager 类的用途了。如清单 2.2 所示的 FullScreenTest 类利用 SimpleScreenManager 类将显示切换到全屏方式，并显示 Hello World!消息，直到等待 5 秒钟后退出。

FullScreenTest 对象将使用的分辨率是 800×600，位深度是 16。如果要使用不同的分辨率，只要在命令行指定出不同的显示方式。程序把收到的第一、第二、第三个变元看成显示方式的宽度、高度与位深度，如在控制台中输入下列命令时，程序使用的分辨率为 1024x768，位深度为 32。

Java FullScreenTest 1024 768 32

当然代码不让用户设置非法的方式，这样会使显示器烧掉。如果不支持请求的显示方式或系统不支持全屏，则会抛出 IllegalArgumentException 异常，FullScreenTest 将用当前的显示方式模拟全屏方式。

清单 2.2　FullScreenTest.java

```
import java.awt.*;import javax.swing.JFrame;
```

```
public class FullScreenTest extends JFrame {

    public static void main(String[] args) {

        DisplayMode displayMode;

        if (args.length == 3) {
            displayMode = new DisplayMode(
                Integer.parseInt(args[0]),
                Integer.parseInt(args[1]),
                Integer.parseInt(args[2]),
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }
        else {
            displayMode = new DisplayMode(800, 600, 16,
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }

        FullScreenTest test = new FullScreenTest();
        test.run(displayMode);
    }

    private static final long DEMO_TIME = 5000;

    public void run(DisplayMode displayMode) {
        setBackground(Color.blue);
        setForeground(Color.white);
        setFont(new Font("Dialog", 0, 24));

        SimpleScreenManager screen = new SimpleScreenManager();
        try {
            screen.setFullScreen(displayMode, this);
            try {
                Thread.sleep(DEMO_TIME);
            }
            catch (InterruptedException ex) { }
        }
        finally {
            screen.restoreScreen();
```

```
        }
    }

    public void paint(Graphics g) {
        g.drawString("Hello World!", 20, 50);
    }
}
```

注意：FullScreenTest 类的 run()方法中使用了 try/finally 代码块。finally 块中恢复屏幕，保证即使 try 块中抛出异常时，也能够恢复屏幕。

还有需要注意的是 paint()方法中使用的 Graphics 对象。Graphics 对象提供绘制文本、直线、矩形、椭圆、多边形、图像等各种功能，大多数方法都是一目了然，可以从 Java API 规范中找到所要的细节。

也许你会问，paint()方法是如何调用的？实际上 FullScreenTest 是一个 Jframe 对象。显示 JFrame（或任何其他组件）时，Java 的抽象视窗工具库（Abstract Window Toolkit 或 AWT）就会调用组件的 paint()方法。

如果要强制 AWT 调用 paint()方法，则可以调用 repaint()方法，使 AWT 在必要时调用 paint()方法。AWT 是在另一线程中发送 paint 事件，因此要发送 repaint 事件，然后等待绘图完成，实现代码如下所示。

```
public class MyComponent extends SomeComponent {
    ...
    public synchronized void repaintAndWait() {
        repaint();
        try {
            wait();
        }
        catch (InterruptedException ex) { }
    }

    public synchronized void paint(Graphics g) {
    //在此进行绘制
    ...
    //通知绘图完成
    notifyAll();
}
}
```

2.1.5　抗齿边

也许你会注意到，FullScreenTest 中的 Hello World 文本边沿上有些锯齿形状，这是因

为文本没有抗齿边。也许听起来有点奇怪，但抗齿边是将文本的边沿模糊化，使文本颜色与背景颜色相混合，从而达到像素的台阶更光顺，可以将图 2-2 和图 2-3 作一个比较。

Hello World!

图 2-2　正常文本边沿有些锯齿形状

Hello World!

图 2-3　抗齿边文本将边沿模糊，让文本显得清晰

为了文本抗齿边，就要在绘制任何文本之前设置相应的绘制提示。绘制提示功能只由 Graphics2D 类提供，它是 Graphics 的子类。

为了保证向后兼容性，paint()方法取 Graphics 对象作为参数。但是，从 paint()开始，传入方法中的参数实际上是 Graphics2D 对象。让文本抗齿边的 paint()方法的代码如下所示。

```
public void paint(Graphics g) {

        if (g instanceof Graphics2D) {
            Graphics2D g2 = (Graphics2D)g;
            g2.setRenderingHint(
                RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        }
        g.drawString("Hello World!", 20, 50);
    }
```

此外，还有其他绘制提示，包括让所有几何直线与形体抗齿边，设置绘制质量等，具体内容请详见 RenderingHints 类的文档。

2.1.6　选用显示方式

显示方式有很多种，但游戏将如何选择呢？答案是至少要保证游戏可以在 2 种分辨率中运行。一般的原则是让游戏者改变显示方式，即选择出系统中最适合的显示方式。

如果可能，最初可以按当前分辨率运行游戏。在液晶显示器中，当前分辨率通常是液晶显示器的自然分辨率，因此使用当前分辨率会使游戏更好看。

当前游戏的位深度最好选用 16 位、24 位或 32 位颜色。16 位颜色通常更快些，但如果需要更精确的颜色表示，则要使用更高的位深度。

另外，人眼适合的刷新率为 75Hz 到 85Hz 之间。

2.2　图像

在屏幕上绘制文本很有趣，但游戏会更需要图像的参与。在屏幕上绘制图像之前，首先介绍图像的一些基本知识：透明度类型和文件格式。

2.2.1　透明度

假设要显示一个简单图像，如图 2-4 所示。

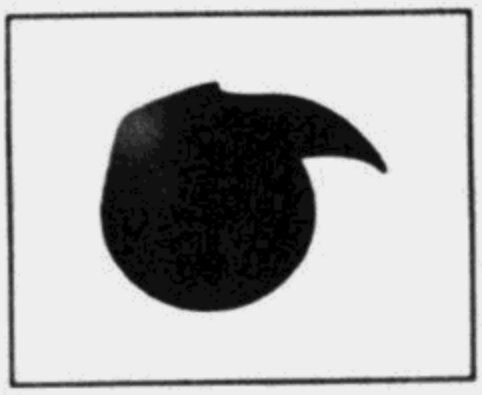


图 2-4　英雄图像，但实际上只是个大球

图像是以白色为背景，但背景是否是图像中的一部分呢？是否也要绘制呢？这取决于图像的透明度。图像的透明度有 3 种：不透明、透明和半透明。

- 不透明图像将显示图像中的每一个像素。
- 透明图像中每一个像素或者完全显示，或者完全穿透。其中白色背景可能是透明的，所以绘制时可以看到图像下面的内容。
- 半透明图像的像素是半透明的，用来产生鬼怪式的部分穿透效果。同样可以在图像边沿使用，来产生抗齿边图像。

在处理角色图像中，就要使用透明效果的白色背景，以便绘制时可以看到图像下面的内容，而且在图像的边沿使用半透明效果，来产生抗齿边图像。

2.2.2　文件格式

图像的基本文件格式有 2 种：光栅格式和矢量格式。光栅格式像显示器一样描述图像，使用指定的位深度像素。矢量格式则利用几何方法描述图像，可以缩放而不会使质量下降。

Java API 没有内置矢量格式，因此这里主要介绍光栅格式。如果你对矢量格式感兴趣，可以参考 Apache 公司的 Scalable Vector Graphics(SVG)版本 Batik，可以在如下网址 http://xml.apache.org/batik 有相关介绍。

Java 运行环境内置了 3 种光栅格式，很容易读取，它们是 GIF,PNG 和 JPEG。

- GIF 图像可以是透明或不透明的，可以使用 8 位以下的颜色深度。尽管 GIF 对颜色变化不大的图形具有很高压缩率，但 PNG 的功能优于 GIF，因此不再需要使用。
- PNG 图像可以是透明、半透明或不透明的，可以使用 24 位以内的任何颜色深度。8 位 PNG 图像的压缩率与 GIF 格式大致相同。
- JPEG 图像只能是不透明的 24 位图形。JPEG 对照片图像的压缩率很高，但它是损失性压缩，因此压缩图像不是原像的精确复制。

这些图像文件可以用常规的绘图程序生成，如 Adobe Photoshop(www.adobe.com), Jasc Paint Shop Pro（www.jasc.com）和 GIMP（www.gimp.org）。

2.2.3 读取图像

如何把GIF，PNG和JPEG文件变成可以显示的内容呢？这是通过工具库中的getImage()方法完成的，它分析图像文件，并返回一个 Image 对象，实现代码如下所示。

```
Toolkit toolkit = Toolkit.getDefaultToolkit();
Image image = toolkit.getImage(fileName);
```

代码看起来很简单，但它并不装入图像，而是在另一线程中实际装入图像。如果在装入图像完成之前显示图像，那么将显示图像的一部分（或什么也不显示）。

可以用 MediaTracker 对象观察图像，等待装入完成，但还有另一个简单方法。ImageIcon 类可以用 MediaTracker 对象装入图像。javax.swing 包中的 ImageIcon 类用工具库装入图像，等待装入完成之后再返回，实现代码如下所示。

```
ImageIcon icon = new ImageIcon(fileName);
Image image = icon.getImage();
```

装入图像后，可以试一试。清单 2.3 的 ImageTest 类与 FullScreenTest 类相似，也使用 SimpleScreenManager 对象。ImageTest 将绘制一个 JPEG 背景图像和 4 个 PNG 前景图像，等待 10 秒钟，然后退出。

背景图像是照片质量的，因此使用 JPEG 格式比 PNG 格式能得到更高的压缩率。

显示的 PNG 图像是透明、半透明和不透明的。一个半透明图像是完全半透明的，而另一个半透明图像只在边沿是半透明的，目的是让图像抗齿边。图 2-5 显示了 ImageTest 的屏幕图形。

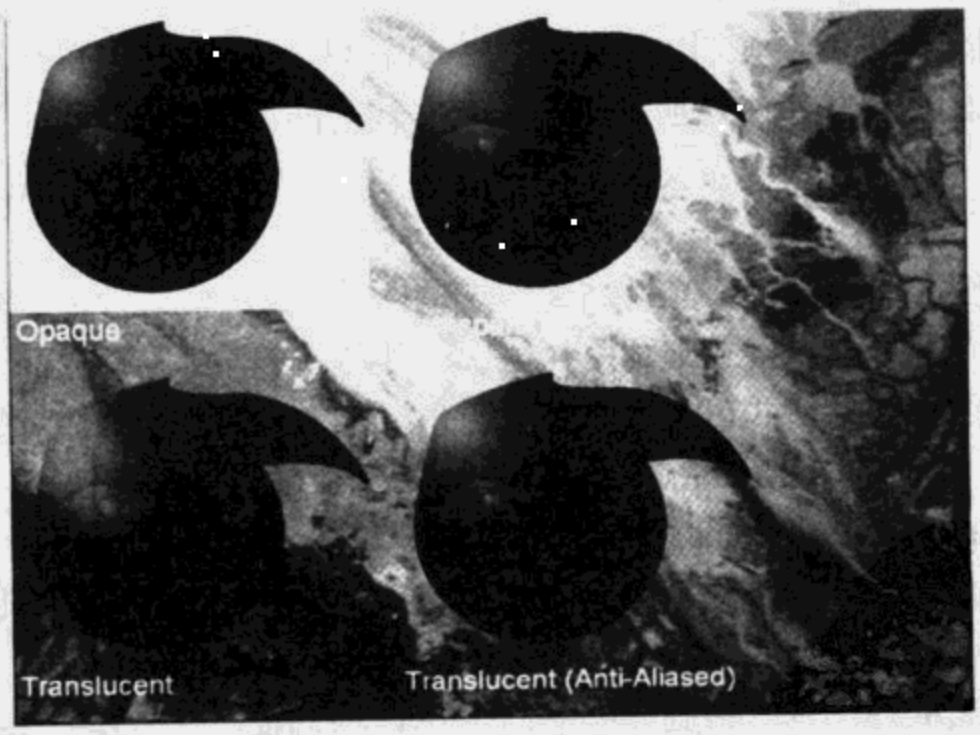


图 2-5　ImageTest 程序显示不同透明度

如果要在不同显示方式中运行 ImageTest，则可以像 FullScreenTest 中一样在命令行指定显示方式。

清单 2.3　ImageTest.java

```
import java.awt.*;
import javax.swing.ImageIcon;
import javax.swing.JFrame;

public class ImageTest extends JFrame {

    public static void main(String[] args) {
        DisplayMode displayMode;

        if (args.length == 3) {
            displayMode = new DisplayMode(
                Integer.parseInt(args[0]),
                Integer.parseInt(args[1]),
                Integer.parseInt(args[2]),
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }
        else {
            displayMode = new DisplayMode(800, 600, 16,
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }

        ImageTest test = new ImageTest();
        test.run(displayMode);
    }

    private static final int FONT_SIZE = 24;
    private static final long DEMO_TIME = 10000;

    private SimpleScreenManager screen;
    private Image bgImage;
    private Image opaqueImage;
    private Image transparentImage;
    private Image translucentImage;
    private Image antiAliasedImage;
    private boolean imagesLoaded;
    public void run(DisplayMode displayMode) {
        setBackground(Color.blue);
```

```
        setForeground(Color.white);
        setFont(new Font("Dialog", Font.PLAIN, FONT_SIZE));
        imagesLoaded = false;

        screen = new SimpleScreenManager();
        try {
            screen.setFullScreen(displayMode, this);
            loadImages();
            try {
                Thread.sleep(DEMO_TIME);
            }
            catch (InterruptedException ex) { }
        }
        finally {
            screen.restoreScreen();
        }
    }

    public void loadImages() {
        bgImage = loadImage("images/background.jpg");
        opaqueImage = loadImage("images/opaque.png");
        transparentImage = loadImage("images/transparent.png");
        translucentImage = loadImage("images/translucent.png");
        antiAliasedImage = loadImage("images/antialiased.png");
        imagesLoaded = true;
        // 让 AWT 重画这个窗口
        repaint();
    }

    private Image loadImage(String fileName) {
        return new ImageIcon(fileName).getImage();
    }

    public void paint(Graphics g) {
        // 设置文本抗齿边
        if (g instanceof Graphics2D) {
            Graphics2D g2 = (Graphics2D)g;
            g2.setRenderingHint(
                RenderingHints.KEY_TEXT_ANTIALIASING,
```

```
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        }

        // 画图
        if (imagesLoaded) {
            g.drawImage(bgImage, 0, 0, null);
            drawImage(g, opaqueImage, 0, 0, "Opaque");
            drawImage(g, transparentImage, 320, 0, "Transparent");
            drawImage(g, translucentImage, 0, 300, "Translucent");
            drawImage(g, antiAliasedImage, 320, 300,
                "Translucent (Anti-Aliased)");
        }
        else {
            g.drawString("Loading Images...", 5, FONT_SIZE);
        }
    }

    public void drawImage(Graphics g, Image image, int x, int y,
        String caption)
    {
        g.drawImage(image, x, y, null);
        g.drawString(caption, x + 5, y + FONT_SIZE +
            image.getHeight(null));
    }
}
```

装入图像之前，ImageTest 类显示一个 Loading images 消息。装入图像之后，调用 repaint() 方法，让 AWT 重画屏幕。这时，画出背景图像和 4 个前景图像。

2.2.4　硬件加速图像

硬件加速图像存放在显示内存中，而不是存放在系统内存中。硬件加速图像可以比非硬件加速图像更快速的拷贝到屏幕上。

Java 对工具库中 getImage()方法装入的任何图像进行硬件加速。由于 Java 使硬件加速自动进行，通常不必增加工作使图像变成硬件加速图像。但是，有几种情形会使图像无法进行硬件加速，它们是：

- 如果图像内容不断改变（如在图像中绘制图形元），则图像无法进行硬件加速。
- 在 Java SDK 1.4.1 中，半透明图像无法进行硬件加速，只有透明和不透明图像才能进行（所以本书很少使用半透明图像）。
- 并不是每个系统都有硬件加速图像功能。

如果在支持硬件加速图像的系统中强制图像硬件加速，则可以创建一个 Volatile Image

对象，它是存放在显示内存中的图像。

要生成 VolatileImage 对象，可以用 Component 的 createVolatileImage(int w, int h)方法或 GraphicsConfiguration 的 createCompatibleVolatileImage(int w, int h)方法。但是，要注意 VolatileImage 只能是不透明的图像。

VolatileImage 可能随时失去内容，所以是“易失的”，如启动屏幕保护程序或改变显示方式等，就要从显示内存中清除 VolatileImage，所以要重画显示的内容。

对检查 VolatileImage 内容是否丢失，可以用 validate()与 contentsLost()方法。validate()方法保证图像与当前显示方式相兼容，而 contentsLost()方法返回的信息则显示上次调用 validate()方法以来图像内容是否丢失。下面将显示如何检查和恢复 VolatileImage，实现代码如下所示。

```
//生成图像
VolatileImage image = createVolatileImage(w, h);
...
//绘制图像
do {
    int valid = image.validate(getGraphicsConfiguration());
    if (valid == VolatileImage.IMAGE_INCOMPATIBLE) {
        //图像与显示器不兼容，重建
        image = createVolatileImage(w, h);
    }
    else if (valid == VolatileImage.IMAGE_RESTORED) {
    //恢复图像
        Graphics2D g = image.createGraphics();
        MyDrawMethod(g);
        g.dispose();
    }
    else {
    //在屏幕上绘制图像
        Graphics g = screen.getDrawGraphics();
        g.drawImage(image, 0, 0, null);
        g.dispose();
    }
}
while (image.contentsLost());
```

上述代码中一直执行循环操作，直到屏幕上成功画出图像为止。

2.2.5 图形绘制基准程序

执行加速操作对透明和不透明图像到底产生多快的影响呢？为了确定到底有多快，下面修改 ImageTest 类并生成 ImageSpeedTest 类，如清单 2.4 所示。ImageSpeedTest 类在指定

时间内重复绘制 ImageTest 中所使用的 4 个图形，然后在控制台上显示每秒钟画出的图形个数。

注意：绝不能让 paint()方法开销大量的时间。虽然在简单实例中不会造成损害，但在实际游戏中，这是行不通的。因为 AWT 事件派生线程调用的 paint()方法，但 AWT 事件派生线程还要处理鼠标、键盘和许多其他事件（详情请见下一章）。如果 paint()方法占用大量的时间，这样实际上会“切开”AWT 线程，不过下节将用更好的方法实现 ImageSpeedTest 类的工作。

清单 2.4　ImageSpeedTest.java

```
import java.awt.*;
import javax.swing.ImageIcon;
import javax.swing.JFrame;

public class ImageSpeedTest extends JFrame {

    public static void main(String args[]) {

        DisplayMode displayMode;

        if (args.length == 3) {
            displayMode = new DisplayMode(
                Integer.parseInt(args[0]),
                Integer.parseInt(args[1]),
                Integer.parseInt(args[2]),
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }
        else {
            displayMode = new DisplayMode(800, 600, 16,
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }

        ImageSpeedTest test = new ImageSpeedTest();
        test.run(displayMode);
    }

    private static final int FONT_SIZE = 24;
    private static final long TIME_PER_IMAGE = 1500;

    private SimpleScreenManager screen;
```

```
    private Image bgImage;
    private Image opaqueImage;
    private Image transparentImage;
    private Image translucentImage;
    private Image antiAliasedImage;
    private boolean imagesLoaded;

    public void run(DisplayMode displayMode) {
        setBackground(Color.blue);
        setForeground(Color.white);
        setFont(new Font("Dialog", Font.PLAIN, FONT_SIZE));
        imagesLoaded = false;

        screen = new SimpleScreenManager();
        try {
            screen.setFullScreen(displayMode, this);
            synchronized (this) {
                loadImages();
                // wait for test to complete
                try {
                    wait();
                }
                catch (InterruptedException ex) { }
            }
        }
        finally {
            screen.restoreScreen();
        }
    }

    public void loadImages() {
        bgImage = loadImage("images/background.jpg");
        opaqueImage = loadImage("images/opaque.png");
        transparentImage = loadImage("images/transparent.png");
        translucentImage = loadImage("images/translucent.png");
        antiAliasedImage = loadImage("images/antialiased.png");
        imagesLoaded = true;
        // 让 AWT 重画这个窗口
        repaint();
```

```
    }

    private final Image loadImage(String fileName) {
        return new ImageIcon(fileName).getImage();
    }

    public void paint(Graphics g) {
        // 设置文本抗齿边
        if (g instanceof Graphics2D) {

            Graphics2D g2 = (Graphics2D)g;
            g2.setRenderingHint(
                RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        }

        // 画图
        if (imagesLoaded) {
            drawImage(g, opaqueImage, "Opaque");
            drawImage(g, transparentImage, "Transparent");
            drawImage(g, translucentImage, "Translucent");
            drawImage(g, antiAliasedImage,

              "Translucent (Anti-Aliased)");

            // 表示测试完毕
            synchronized (this) {
                notify();
            }
        }
        else {
            g.drawString("Loading Images...", 5, FONT_SIZE);
        }
    }

    public void drawImage(Graphics g, Image image, String name) {
        int width = screen.getFullScreenWindow().getWidth() -
            image.getWidth(null);
        int height = screen.getFullScreenWindow().getHeight() -
```

```
            image.getHeight(null);
        int numImages = 0;

        g.drawImage(bgImage, 0, 0, null);

        long startTime = System.currentTimeMillis();
        while (System.currentTimeMillis() - startTime
            < TIME_PER_IMAGE)
        {
            int x = Math.round((float)Math.random() * width);
            int y = Math.round((float)Math.random() * height);
            g.drawImage(image, x, y, null);
            numImages++;
        }
        long time = System.currentTimeMillis() - startTime;
        float speed = numImages * 1000f / time;
        System.out.println(name + ": " + speed + " images/sec");
    }
}
```

不要以为清单 2.4 就是最终的图形绘制基准程序。记住，这只是在一台机器上测试 4 个图像，得到的结果会取决于计算机的显示卡、处理器速度、显示方式和绘制速度。

为了有个直观的概念，下面是 600MHz Athlon 机器上的测试结果，显示卡为 GeForce-256，显示分辨率为 800x600，位深度为 16，目前感觉不错，显示结果如下所示：

Opaque: 5550.599 images/sec

Transparent: 5478.6953 images/sec

Translucent: 85.2197 images/sec

Translucent: (Anti-Aliased): 113.18243 images/sec

可以看出，计算机上的半透明图像比硬件加速不透明和透明图像慢得多。抗齿边图像比完全半透明图像稍快一些，原因可能是图像中有更多实体像素，因此绘图时混合的工作量更小。

这里透明图像与不透明图像几乎一样快。注意有些旧式显示卡可能无法绘制硬件加速透明图像，只好用与半透明图像相同的低速方法。

这样就介绍了硬件加速图像和图形绘制基准程序，下面就介绍游戏中所要用到的动画。

2.2.6 动画

这里要介绍的第一种动画是卡通式动画，它显示为连续的一序列图像。下面就介绍卡通式动画的工作方法。

我们以动态的英雄版本为例，它的头发会飘动，眼睛会眨眼，如图 2-6 所示。

图 2-6　英雄的 3 个动画帧，闭眼时的样子很可爱呦

动画中的图像可以称为帧（frame）。每一帧显示一定时间，但帧不一定都显示相同时间，如第一帧显示 200 毫秒，第二帧显示 75 毫秒等，如图 2-7 所示。

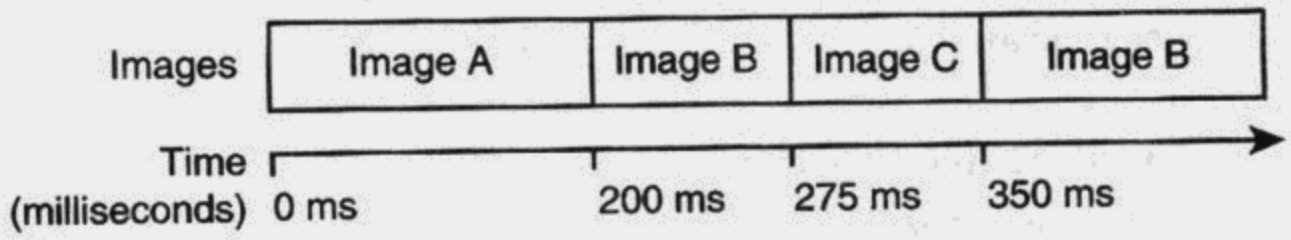


图 2-7　动画显示为连续的一序列图像，每一帧显示一定时间，同一图像可以多次使用

以上就是动画的概念，下面就要介绍代码了。首先明确动画中同一个图像可以多次使用，另外，动画是无限循环的，而不只是播放一次。

清单 2.5 的 Animation 类有 3 个重要方法：addFrame(), update()和 getImage()。addFrame()方法将图形加进动画中，并指定显示时间（毫秒）。update()方法告诉动画已经经过的指定时间。最后，getImage()方法根据经过的时间量取得要显示的图像。

清单 2.5　Animation.java

```
import java.awt.Image;
import java.util.ArrayList;
/**
    Animation 类管理一系列图像（帧）和每个帧显示的时间量
*/
public class Animation {

    private ArrayList frames;
    private int currFrameIndex;
    private long animTime;

    private long totalDuration;

    /**
        生成新的空白 Animation
    */
```

```
    public Animation() {
        frames = new ArrayList();
        totalDuration = 0;
        start();
    }

    /**
        将指定显示时间的图像加进动画中
    */
    public synchronized void addFrame(Image image,
        long duration)
    {
        totalDuration += duration;
        frames.add(new AnimFrame(image, totalDuration));
    }

    /**
        从头开始重新启动这个动画
    */
    public synchronized void start() {
        animTime = 0;
        currFrameIndex = 0;
    }

    /**
        必要时更新这个动画的当前图像（帧）
    */
    public synchronized void update(long elapsedTime) {
        if (frames.size() > 1) {
            animTime += elapsedTime;

            if (animTime >= totalDuration) {
                animTime = animTime % totalDuration;
                currFrameIndex = 0;
            }

            while (animTime > getFrame(currFrameIndex).endTime) {

                currFrameIndex++;
```

```
        }
    }
}

/**
    取得这个 Animation 的当前图像，如果没有图像，则返回 null
*/
public synchronized Image getImage() {
    if (frames.size() == 0) {
        return null;
    }
    else {
        return getFrame(currFrameIndex).image;
    }
}

private AnimFrame getFrame(int i) {
    return (AnimFrame)frames.get(i);
}

    private class AnimFrame {

        Image image;
        long endTime;

        public AnimFrame(Image image, long endTime) {
            this.image = image;
            this.endTime = endTime;
        }
    }
}
```

在 Animation 类中，可以看到代码中有奇怪的符号%，如下所示。

animTime = animTime % totalDuration;

它是求余运算符，将返回整除的余数，如(10 % 3)的值为 1。 因为 10 除以 3 的商为 3，余数为 1。这里用它保证动画完成时，动画时间重新开始，使动画循环。

Animation 的代码很简单。其中 AnimFrame 类包含图像及其显示的时间量。大多数工作是在 Animation 的 update()方法中完成，方法根据经过的时间量取得正确的 AnimFrame。

2.2.7 活动绘制

要实现动画就要以高效的方式连续更新屏幕。过去使用 paint()方法进行任何绘制，通过调用 repaint()方法，让 AWT 事件派生线程重画屏幕，但这样会造成延迟现象，因为 AWT 线程可能忙于其他事务。

另一个方法是使用活动绘制。活动绘制（Active rendering）是指在主线程中直接绘制屏幕，这样就可以控制何时的实际绘制屏幕，同时使代码得到简化。

要使用活动绘制方法，可以调用 Component 类的 getGraphics()方法取得屏幕的图形描述表，实现代码如下所示。

```
Graphics g = screen.getFullScreenWindow() .getGraphics();
draw(g);
g.dispose();
```

代码很简单吧，和本例中一样，记住在绘图完成之后要处理 Graphics 对象，从而清理一些资源，内存回收单元可能要过一阵子才会清理这些资源。

2.2.8 动画循环

下面使用活动绘制在循环中连续绘图，这个循环称为动画循环，首先观察它的步骤。

（1）更新任何动画。

（2）绘制屏幕。

（3）可选休眠一会儿。

（4）再次从第 1 步开始。

然后给出动画循环的代码，实现代码如下所示。

```
while (true) {

    //更新任何动画
    updateAnimations();

    //绘制屏幕
    Graphics g = screen.getFullScreenWindow().getGraphics();
    draw(g);
    g.dispose();

    //休眠一会儿
    try {
        Thread.sleep(20);
    }
    catch (InterruptedException ex) {   }
}
```

由于现实的动画循环是不能永远循环的，所以在本例中，也让动画循环在几秒钟后停

止。

现在就有了测试动画所需的全部知识了，清单 2.6 的 AnimationTest1 程序是个简单实例，看看能否让英雄眨眨眼睛呢。

在 AnimationTest1 类代码中，在每次画图时会更新整个屏幕，但也可能只更新上次画图中发生改变的部分。此方法特别适合静态背景的游戏，如 Pac-Man 中的遗留，不过大多数现代游戏的背景都是动态的或同时发生多件事情。另外，加上后面将要介绍的翻页特性，就无法只更新上次画图中发生改变的部分。简而言之，本书中的实例是直接更新整个屏幕的。

清单 2.6　AnimationTest1.java

```
import java.awt.*;
import javax.swing.ImageIcon;
import javax.swing.JFrame;

public class AnimationTest1 {

    public static void main(String args[]) {

        DisplayMode displayMode;

        if (args.length == 3) {
            displayMode = new DisplayMode(
                Integer.parseInt(args[0]),
                Integer.parseInt(args[1]),
                Integer.parseInt(args[2]),
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }
        else {
            displayMode = new DisplayMode(800, 600, 16,
                DisplayMode.REFRESH_RATE_UNKNOWN);
        }

        AnimationTest1 test = new AnimationTest1();
        test.run(displayMode);
    }

    private static final long DEMO_TIME = 5000;
    private SimpleScreenManager screen;
    private Image bgImage;
```

```
    private Animation anim;

    public void loadImages() {
        // 装入图像
        bgImage = loadImage("images/background.jpg");
        Image player1 = loadImage("images/player1.png");
        Image player2 = loadImage("images/player2.png");
        Image player3 = loadImage("images/player3.png");

        // 生成动画
        anim = new Animation();
        anim.addFrame(player1, 250);
        anim.addFrame(player2, 150);
        anim.addFrame(player1, 150);
        anim.addFrame(player2, 150);
        anim.addFrame(player3, 200);
        anim.addFrame(player2, 150);
    }

    private Image loadImage(String fileName) {
        return new ImageIcon(fileName).getImage();
    }

    public void run(DisplayMode displayMode) {
        screen = new SimpleScreenManager();
        try {
            screen.setFullScreen(displayMode, new JFrame());
            loadImages();
            animationLoop();
        }
        finally {
            screen.restoreScreen();
        }
    }

    public void animationLoop() {
        long startTime = System.currentTimeMillis();
        long currTime = startTime;
```

```
        while (currTime - startTime < DEMO_TIME) {
            long elapsedTime =

                System.currentTimeMillis() - currTime;
            currTime += elapsedTime;

            // 更新动画
            anim.update(elapsedTime);

            // 绘制屏幕
            Graphics g =
                screen.getFullScreenWindow().getGraphics();
            draw(g);
            g.dispose();

            // 休眠一会儿
            try {
                Thread.sleep(20);
            }
            catch (InterruptedException ex) { }
        }
    }

    public void draw(Graphics g) {
        // 绘制背景
        g.drawImage(bgImage, 0, 0, null);

        // 绘制图像
        g.drawImage(anim.getImage(), 0, 0, null);
    }
}
```

2.3　避免闪烁与裂开

运行 AnimationTest1 对象时，你可能发现有个大的问题：动画角色会闪烁，非常讨厌。为什么动画角色会闪烁，如何克服这个问题呢？

这是因为不断直接绘制屏幕的结果，如图 2-8 所示。当要用背景擦除角色，然后重画角色，因此在某个瞬间，可能在角色位置看到背景。由于这一切发生得如此之快，所以会出现闪烁。

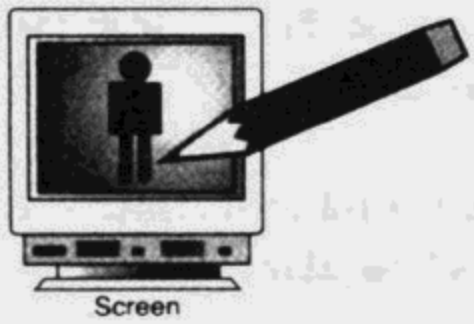


图 2-8 直接绘制屏幕时，用户会看到正在绘制的内容

那如何避免闪烁呢？办法就是用双缓存。

2.3.1 双缓存

缓存区（buffer）是绘图时使用的屏外内存区。使用双缓存不是直接绘制屏幕，而是绘制到后缓存区，然后将整个缓存区复制到屏幕上，如图 2-9 所示。这样就可以同时更新整个屏幕，游戏者只看到要看到的东西了。

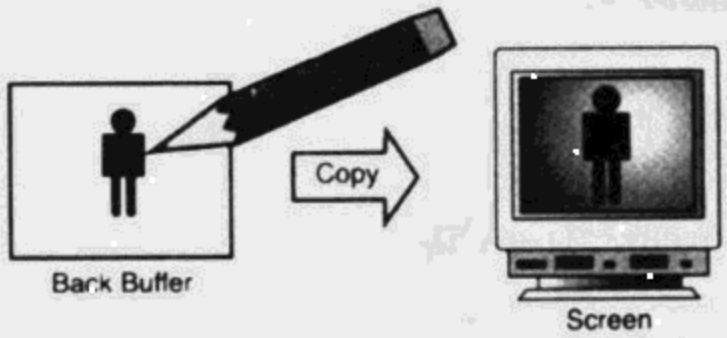


图 2-9 双缓存绘制到后缓存区，然后将整个缓存区复制到屏幕上

后缓存区可能只是普通 Java 图像。可以用 Component 类的 createImage(int w, int h)方法生成后缓存区，如果是不使用活动绘制的小程序使用双缓存，则可以覆盖 update()方法，改用双缓存，并用双缓存的图形描述表调用 paint()方法，实现代码如下所示。

```
private Image doubleBuffer;
...
public void update(Graphics g) {
    Dimension size = getSize();
    if (doubleBuffer == null ||
        doubleBuffer.getWidth(this) ! = size.width ||
        doubleBuffer.getHeight(this) ! = size.height)
    {
        doubleBuffer = createImage(size.width, size.height);
    }

    if (doubleBuffer ! = null) {
        //绘制到双缓存
        Graphics g2 = doubleBuffer.getGraphics();
        paint(g2);
```

```
            g2.dispose();

            //将双缓存复制到屏幕
            g.drawImage(doubleBuffer, 0, 0, null);
        }
        else {
            //不能生成双缓存，只绘制到屏幕
            paint(g);
        }
    }

    public void paint (Graphics g) {
        //在此绘图
        ...
    }
```

2.3.2　翻页

使用双缓存的一个缺点就是将后缓存区复制到屏幕上需要较长时间，以分辨率为 800x600、位深度为 16，那么就需要 800×600×2 字节，即 938kbyte，这样每秒钟 30 帧要交换将近 1Mbyte 内存。尽管复制如此大的内存量在许多游戏中也是相当快的，但如果不用复制缓存区，而能直接把后缓存区变成显示缓存区，那该多好！

翻页技术（page flipping）就可以实现。使用翻页方法时，要用 2 个缓存区，一个是后缓存区，一个是显示缓存区，如图 2-10 所示。

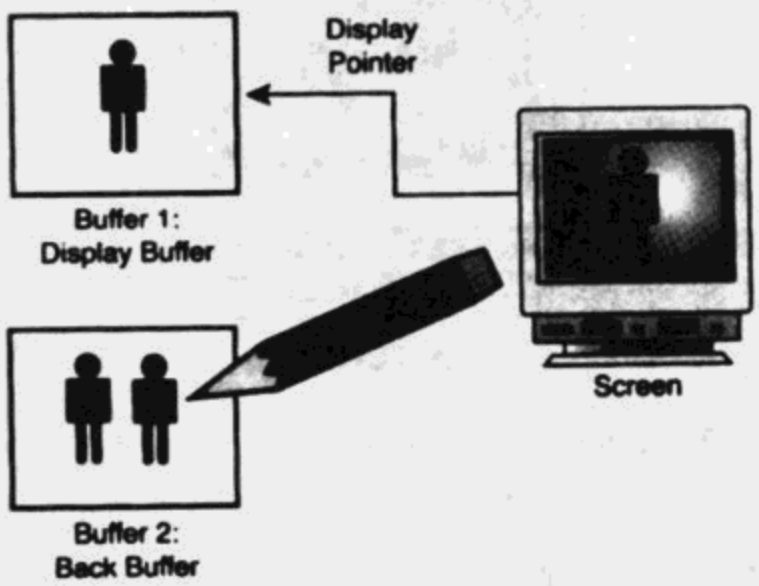


图 2-10　显示指针指向缓存区 1，缓存区 2 用作后缓存区

显示指针指向要显示的缓存区，大多数现代系统都可以改变这个显示指针。向后缓存区绘制完毕后，显示指针可以从当前显示缓存区切换到后缓存区上，如图 2-11。改变显示指针时，显示缓存区立即变成后缓存区，而后缓存区立即变成显示缓存区。

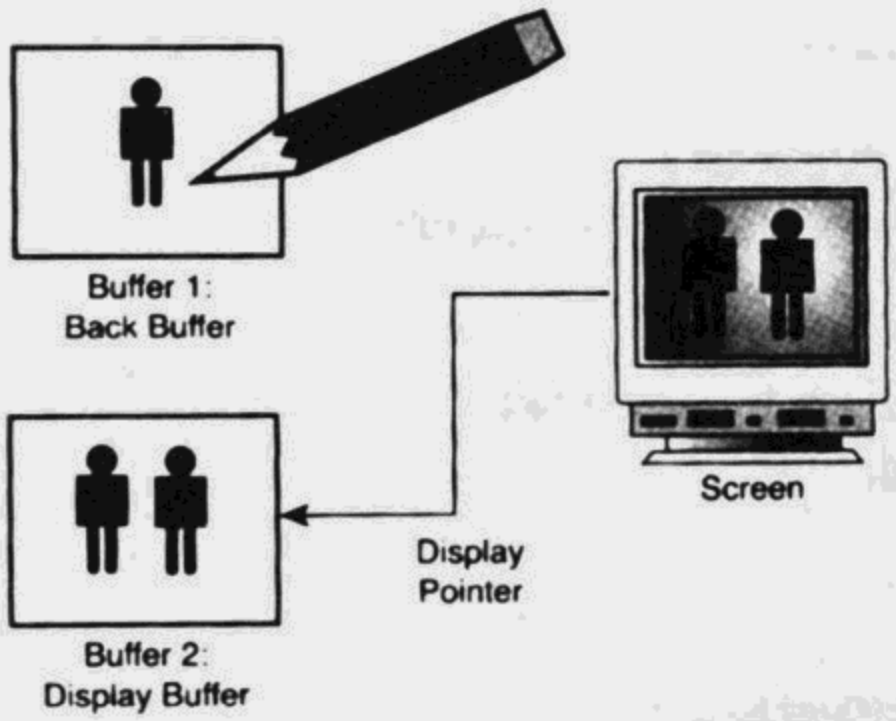


图 2-11　翻页之后，显示指针指向缓存区 2，缓存区 1 用作后缓存区

当然，改变指针比复制大块内存要快得多，从而比双缓存大大提高性能。

2.3.3　监视刷新与裂开

显示器的刷新率通常在 75Hz 左右，表示显示器每秒刷新 75 次。但如果在显示器刷新途中发生翻页或缓存区复制操作，那会发生什么情况呢？答案是同时显示一部分旧缓存区和一部分新缓存区！这个效果与闪烁相似，称为裂开（tearing）（如图 2-12 所示）。裂开的速度很快，人们几乎注意不到，如果能看到，效果就像屏幕上某个地方开裂一样。

图 2-12　发生裂开时，同时显示一部分旧缓存区和显示一部分新缓存区，图中的假想虚线是开裂位置

为了解决问题，可以在显示器刷新之前完成翻页工作。虽然看起来复杂，但不必担心，因为 Java 运行环境会用 BufferStrategy 类完成工作。

2.3.4　BufferStrategy 类

双缓存、翻页和显示器刷新等待都是由 BufferStrategy 类处理的，它会根据系统功能选择最佳的缓存方法。首先，它尝试进行翻页，若无法翻页则尝试进行双缓存。另外，翻页

之前要等待显示器刷新完毕。简而言之，一切都交给这个类，你就不必操心了。

等待显示器刷新的一个缺点是会限制游戏中每秒可以显示的帧数。如果显示器刷新率设置为 75Hz，则游戏中每秒可以显示的帧数最多为 75。这样就不能用游戏帧速率作为测试系统运行速度的基准。

当然，游戏是否每秒运行 200 帧并不重要，因为只能看到显示器能显示的帧数。不管游戏运行速度有多快，刷新率设置为 75Hz 时，就只能在每秒中看到 75 帧。

Canvas 与 Window 对象都可以有 BufferStrategy。利用 createBufferStrategy()方法，根据所要的缓存数生成 BufferStrategy。要进行翻页与双缓存，至少要有 2 个缓存区，实现代码如下所示。

```
frame.createBufferStrategy(2);
```

生成 BufferStrategy 之后，要取得其引用，可调用 getBufferStrategy()方法。然后用 getDrawGraphics()方法取得绘图缓存区的图形描述表。绘图完成之后，调用 show()方法显示绘图缓存区，这样就可以使用翻页，也可以将绘图缓存区复制到显示缓存区，实现代码如下所示。

```
BufferStrategy strategy = frame.getBufferStrategy();
Graphics g =strategy.getDrawGraphics();
draw(g);
g.dispose();
strategy.show();
```

2.3.5　生成屏幕管理器

下面要用这些新特性更新 SimpleScreenManager 类，要增加下列内容：

- 生成 BufferStrategy，实现翻页与双缓存；
- getGraphics(),取得显示的图形描述表；
- update(),更新显示；
- getCompatibleDisplayModes(),取得一列兼容显示方式；
- getCurrentDisplayMode(),取得当前显示方式；
- findFirstCompatibleMode(),从显示方式清单中取得第一个兼容显示方式。

进行活动绘制之后，就不需要用 JFrame 作为全屏窗口，从操作系统接收 paint 事件，因此可以将其关闭，实现代码如下所示。

```
frame.ignoreRepaint(true);
```

但这并不关闭正常的 repaint 事件。因为对 JFrame 调用 repaint()方法时，仍然发送 paint 事件。

清单 2.7 所示的 ScreenManager 类中用这些新特性更新 SimpleScreenManager。

清单 2.7　ScreenManager.java

```
import java.awt.*;
import java.awt.image.BufferStrategy;
import javax.swing.JFrame;
```

```
/**
    ScreenManager 类管理初始化，显示全屏图形方式
*/
public class ScreenManager {

    private GraphicsDevice device;

    /**
        生成新 ScreenManager 对象
    */
    public ScreenManager() {
        GraphicsEnvironment environment =
            GraphicsEnvironment.getLocalGraphicsEnvironment();
        device = environment.getDefaultScreenDevice();
    }

    /**
        对系统缺省设备返回一列兼容显示方式
    */
    public DisplayMode[] getCompatibleDisplayModes() {
        return device.getDisplayModes();
    }

    /**
        返回方法清单中第一个兼容显示方式。如果没有兼容显示方式，则返回 null
    */
    public DisplayMode findFirstCompatibleMode(
        DisplayMode modes[])
    {
        DisplayMode goodModes[] = device.getDisplayModes();
        for (int i = 0; i < modes.length; i++) {
            for (int j = 0; j < goodModes.length; j++) {
                if (displayModesMatch(modes[i], goodModes[j])) {
                    return modes[i];
                }
            }
        }

        return null;
```

```
}

/**
    返回当前显示方式
*/
public DisplayMode getCurrentDisplayMode() {
    return device.getDisplayMode();
}

/**
    确定两个显示方式是否匹配。两个显示方式匹配指具有相同分辨率、位深度和刷新率。如
    果一个显示方式的位深度为 DisplayMode.BIT_DEPTH_MULTI，则忽略位深度。同样，如
    果一个显示方式的刷新率为 DisplayMode.REFRESH_RATE_UNKNOWN，则忽略位深度
*/
public boolean displayModesMatch(DisplayMode mode1,
    DisplayMode mode2)

{
    if (mode1.getWidth() != mode2.getWidth() ||
        mode1.getHeight() != mode2.getHeight())
    {
        return false;
    }

    if (mode1.getBitDepth() != DisplayMode.BIT_DEPTH_MULTI &&
        mode2.getBitDepth() != DisplayMode.BIT_DEPTH_MULTI &&
        mode1.getBitDepth() != mode2.getBitDepth())
    {
        return false;
    }

    if (mode1.getRefreshRate() !=
        DisplayMode.REFRESH_RATE_UNKNOWN &&
        mode2.getRefreshRate() !=
        DisplayMode.REFRESH_RATE_UNKNOWN &&
        mode1.getRefreshRate() != mode2.getRefreshRate())
     {

         return false;
```

```
        }

        return true;
    }

    /**
        进入全屏方式和改变显示方式。如果指定显示方式为 null 或与这个设备不兼容，或这个系
        统中无法改变显示方式，则使用当前显示方式。显示所用 BufferStrategy 有 2 个缓存区
    */
    public void setFullScreen(DisplayMode displayMode) {
        final JFrame frame = new JFrame();
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setUndecorated(true);
        frame.setIgnoreRepaint(true);
        frame.setResizable(false);

        device.setFullScreenWindow(frame);

        if (displayMode != null &&
            device.isDisplayChangeSupported())
        {
            try {
                device.setDisplayMode(displayMode);
            }
            catch (IllegalArgumentException ex) { }
            // fix for mac os x
            frame.setSize(displayMode.getWidth(),
                displayMode.getHeight());
        }
        // avoid potential deadlock in 1.4.1_02
        try {
            EventQueue.invokeAndWait(new Runnable() {
                public void run() {
                    frame.createBufferStrategy(2);
                }
            });
        }
        catch (InterruptedException ex) {
            // ignore
```

```
        }
        catch (InvocationTargetException  ex) {
            // ignore
        }
    }

    /**
        取得显示的图形描述表 ScreenManager 使用双缓存，因此应用程序要调用 update()方法，
        显示绘制的任何图形
        <p>
        应用程序要处置图形对象
    */
    public Graphics2D getGraphics() {
        Window window = device.getFullScreenWindow();
        if (window != null) {
            BufferStrategy strategy = window.getBufferStrategy();
            return (Graphics2D)strategy.getDrawGraphics();
        }
        else {
            return null;
        }
    }

    /**
        更新显示
    */
    public void update() {
        Window window = device.getFullScreenWindow();
        if (window != null) {
            BufferStrategy strategy = window.getBufferStrategy();
            if (!strategy.contentsLost()) {
                strategy.show();
            }
        }
        // Sync the display on some systems.
        // (on Linux, this fixes event queue problems)
        Toolkit.getDefaultToolkit().sync();
    }
```

```
/**
    返回全屏方式中当前使用的窗口。如果设备不在全屏方式中，则返回 null
*/
public JFrame getFullScreenWindow() {
    return (JFrame)device.getFullScreenWindow();
}

/**
    返回全屏方式中当前使用的窗口宽度。如果设备不在全屏方式中，则返回 0
*/
public int getWidth() {
    Window window = device.getFullScreenWindow();
    if (window != null) {
        return window.getWidth();
    }
    else {
        return 0;
    }
}

/**
    返回全屏方式中当前使用的窗口高度。如果设备不在全屏方式中，则返回 0
*/
public int getHeight() {
    Window window = device.getFullScreenWindow();
    if (window != null) {
        return window.getHeight();
    }
    else {
        return 0;
    }
}

/**
    恢复屏幕的显示方式
*/
public void restoreScreen() {
    Window window = device.getFullScreenWindow();
    if (window != null) {
```

```
            window.dispose();
        }
        device.setFullScreenWindow(null);
    }

    /**
        生成与当前显示器兼容的图像
    */
    public BufferedImage createCompatibleImage(int w, int h,
        int transparancy)
    {
        Window window = device.getFullScreenWindow();
        if (window != null) {
            GraphicsConfiguration gc =
                window.getGraphicsConfiguration();
            return gc.createCompatibleImage(w, h, transparancy);
        }
        return null;
    }
}
```

在 ScreenManager 类代码中，注意 update()方法有下列语句。

Toolkit.getDefaultToolkit().sync();

这个方法保证显示与窗口系统同步。在许多系统中，它什么也不干，但在 Linux 中，调用这个方法可以解决 AWT 事件队列的问题。如果不调用这个方法，则有些 Linux 系统可能遇到延迟的鼠标和键盘输入事件。

2 个值得注意的 ScreenManager 类的新方法是 displayModesMatch()与 createCompatibleImage()。

displayModesMatch()方法检查 2 个 DisplayMode 对象是否匹配，匹配是指具有相同分辨率、位深度和刷新率。如果位深度或刷新率在一个 DisplayMode 对象中没有指定，则将其忽略。

createCompatibleImage()方法生成与当前显示器兼容的图像，即与显示器具有相同位深度和颜色模型的图像。生成的图像类为 BufferedImage,是系统内存中存放的非加速图像。这个方法可以生成透明和半透明图像，因为正常的 createImage()方法只生成不透明图像。

下面将要更新 AnimationTest1 类,它使用更新后的 ScreenManager 对象,清单 2.8 显示生成的 AnimationTest2 类，这时程序将不再闪烁!

清单 2.8　AnimationTest2.java

```
import java.awt.*;
import javax.swing.ImageIcon;
```

```
public class AnimationTest2 {

    public static void main(String args[]) {
        AnimationTest2 test = new AnimationTest2();
        test.run();
    }

    private static final DisplayMode POSSIBLE_MODES[] = {
        new DisplayMode(800, 600, 32, 0),
        new DisplayMode(800, 600, 24, 0),
        new DisplayMode(800, 600, 16, 0),
        new DisplayMode(640, 480, 32, 0),
        new DisplayMode(640, 480, 24, 0),
        new DisplayMode(640, 480, 16, 0)
    };
    private static final long DEMO_TIME = 10000;

    private ScreenManager screen;
    private Image bgImage;
    private Animation anim;

    public void loadImages() {
        //装入图像
        bgImage = loadImage("images/background.jpg");
        Image player1 = loadImage("images/player1.png");
        Image player2 = loadImage("images/player2.png");
        Image player3 = loadImage("images/player3.png");

        //生成动画
        anim = new Animation();
        anim.addFrame(player1, 250);
        anim.addFrame(player2, 150);
        anim.addFrame(player1, 150);
        anim.addFrame(player2, 150);
        anim.addFrame(player3, 200);
        anim.addFrame(player2, 150);
    }

    private Image loadImage(String fileName) {
```

```
        return new ImageIcon(fileName).getImage();
    }

    public void run() {
        screen = new ScreenManager();
        try {
            DisplayMode displayMode =
                screen.findFirstCompatibleMode(POSSIBLE_MODES);
            screen.setFullScreen(displayMode);
            loadImages();
            animationLoop();
        }
        finally {
            screen.restoreScreen();
        }
    }

    public void animationLoop() {
        long startTime = System.currentTimeMillis();
        long currTime = startTime;

        while (currTime - startTime < DEMO_TIME) {
            long elapsedTime =
                System.currentTimeMillis() - currTime;
            currTime += elapsedTime;

            //更新动画
            anim.update(elapsedTime);

            // 绘制与更新屏幕
            Graphics2D g = screen.getGraphics();
            draw(g);
            g.dispose();
            screen.update();

            //休眠一会儿
            try {
                Thread.sleep(20);
            }
```

```
            catch (InterruptedException ex) { }
        }

    }

    public void draw(Graphics g) {
        //绘制背景
        g.drawImage(bgImage, 0, 0, null);

        //绘制图像
        g.drawImage(anim.getImage(), 0, 0, null);
    }
}
```

从 AnimationTest1 转到 AnimationTest2 没有太多变化。只是 AnimationTest2 改变了选择显示的方式，不是使用缺省显示方式或从命令行取得显示方式，而是向 ScreenManager 提供一列可用的显示方式，由 ScreenManager 选择清单中第一个兼容显示的方式。

另外，ScreenManager 还生成自己的 JFrame 对象，所以 AnimationTest2 不必生成用作全屏窗口的 JFrame。

2.3.6 幽灵

动画已经顺利运行了，但只是在屏幕上一个地方看到动画并不奇妙，所以下面要生成幽灵。

幽灵（sprite）是在屏幕上独立移动的图形。其实幽灵也是动画，不过可以同时执行动画和移动的操作。除了动画之外，幽灵还有 2 个特性：位置与速度。如果学校的速度课你没有去听，那么我来告诉你，速度就是速率（如 55 米/小时）加上运动方向（如向北）。这里把速度分解为水平和垂直成分，但单位不是英里/小时或米/秒，而是像素/毫秒。

为什么要用速度，而不是直接在每帧中将幽灵位置作一定修改呢？如果直接在每帧中将幽灵位置作一定修改，则不同机器的不同速度会将幽灵移到不同的位置上。帧速率越快，幽灵移动越快。如果幽灵实时移动，则不管帧之间的时间是长是短，幽灵都按一致的路径移动。

和动画一样，幽灵根据上次显示幽灵时经过的毫秒数来更新，如经过 50 毫秒之后，幽灵会根据速度更新其位置与动画。清单 2.9 的 Sprite 类具有动画、位置和速度。

幽灵位置可以使用整数，但幽灵会慢慢移动。假设每次调用 update()方法时，幽灵移动十分之一像素，则 9 次调用 update()方法时，幽灵移动均不明显，到第 10 次调用 update()方法时，才能看到幽灵移动。为此，幽灵位置要使用浮点数值，所以要取得精确的像素位置，可以使用 Math.round()方法。

清单 2.9　Sprite.java

```
import java.awt.Image;
```

```
public class Sprite {

    private Animation anim;
    // 位置（像素）
    private float x;
    private float y;
    // 速度（像素/毫秒）
    private float dx;
    private float dy;

    /**
        用指定 Animation 生成新的 Sprite 对象
    */
    public Sprite(Animation anim) {
        this.anim = anim;

    }

    /**
        根据速度更新 Sprite 的 Animation 及其位置
    */
    public void update(long elapsedTime) {
        x += dx * elapsedTime;
        y += dy * elapsedTime;
        anim.update(elapsedTime);
    }

    /**
        取得 Sprite 的当前 x 位置
    */
    public float getX() {
        return x;
    }

    /**
        取得 Sprite 的当前 y 位置
    */
    public float getY() {
        return y;
```

```
    }

    /**
        设置 Sprite 的当前 x 位置
    */
    public void setX(float x) {
        this.x = x;
    }

    /**
        设置 Sprite 的当前 y 位置
    */
    public void setY(float y) {
        this.y = y;
    }
    /**
        根据当前回像长度取得 Sprite 的宽度
    */
    public int getWidth() {
        return anim.getImage().getWidth(null);
    }

    /**
        根据当前回像长度取得 Sprite 的高度
    */
    public int getHeight() {
        return anim.getImage().getHeight(null);
    }

    /**
        取得 Sprite 的水平速度，（像素/毫秒）
    */
    public float getVelocityX() {
        return dx;
    }

    /**
        取得 Sprite 的垂直速度，（像素/毫秒）
    */
```

```
    public float getVelocityY() {
        return dy;
    }

    /**
        设置 Sprite 的水平速度，（像素/毫秒）
    */
    public void setVelocityX(float dx) {
        this.dx = dx;
    }

    /**
        设置 Sprite 的垂直速度，（像素/毫秒）
    */
    public void setVelocityY(float dy) {
        this.dy = dy;
    }

    /**
        取得 Sprite 的当前图像
    */
    public Image getImage() {
        return anim.getImage();
    }
}
```

Sprite 类很简单，只有一堆 get()和 set()方法，所有的工作均是在 update()方法中完成，其根据速度和经过的时间量更新幽灵位置。

下面要玩点有趣的东西，用 Sprite 类让角色动画和在屏幕上来回弹跳，清单 2.10 的 SpriteTest1 程序来完成这个工作，每次幽灵碰到屏幕边沿时，其速度改变，表示弹回。

清单 2.10 SpriteTest1.java

```
import java.awt.*;
import javax.swing.ImageIcon;

public class SpriteTest1 {

    public static void main(String args[]) {
        SpriteTest1 test = new SpriteTest1();
        test.run();
```

```
    }

    private static final DisplayMode POSSIBLE_MODES[] = {
        new DisplayMode(800, 600, 32, 0),
        new DisplayMode(800, 600, 24, 0),
        new DisplayMode(800, 600, 16, 0),
        new DisplayMode(640, 480, 32, 0),
        new DisplayMode(640, 480, 24, 0),
        new DisplayMode(640, 480, 16, 0)
    };

    private static final long DEMO_TIME = 10000;

    private ScreenManager screen;
    private Image bgImage;

    private Sprite sprite;

    public void loadImages() {
        // 装入图像
        bgImage = loadImage("images/background.jpg");
        Image player1 = loadImage("images/player1.png");
        Image player2 = loadImage("images/player2.png");
        Image player3 = loadImage("images/player3.png");

        // 生成幽灵
        Animation anim = new Animation();
        anim.addFrame(player1, 250);
        anim.addFrame(player2, 150);
        anim.addFrame(player1, 150);
        anim.addFrame(player2, 150);
        anim.addFrame(player3, 200);
        anim.addFrame(player2, 150);
        sprite = new Sprite(anim);

        // 幽灵首先向右向下移动
        sprite.setVelocityX(0.2f);
        sprite.setVelocityY(0.2f);
```

```
    }

    private Image loadImage(String fileName) {
        return new ImageIcon(fileName).getImage();
    }

    public void run() {
        screen = new ScreenManager();
        try {
            DisplayMode displayMode =
                screen.findFirstCompatibleMode(POSSIBLE_MODES);
            screen.setFullScreen(displayMode);
            loadImages();
            animationLoop();
        }
        finally {
            screen.restoreScreen();
        }
    }

    public void animationLoop() {
        long startTime = System.currentTimeMillis();
        long currTime = startTime;

        while (currTime - startTime < DEMO_TIME) {
            long elapsedTime =
                System.currentTimeMillis() - currTime;
            currTime += elapsedTime;

            // 更新幽灵
            update(elapsedTime);

            // 绘制与更新屏幕
            Graphics2D g = screen.getGraphics();
            draw(g);
            g.dispose();
            screen.update();

            // 休眠一会儿
```

```
            try {
                Thread.sleep(20);
            }
            catch (InterruptedException ex) { }
        }
    }

    public void update(long elapsedTime) {
        // 检查幽灵边界
        if (sprite.getX() < 0) {
            sprite.setVelocityX(Math.abs(sprite.getVelocityX()));
        }
        else if (sprite.getX() + sprite.getWidth() >=
            screen.getWidth())
        {
            sprite.setVelocityX(-Math.abs(sprite.getVelocityX()));
        }
        if (sprite.getY() < 0) {
            sprite.setVelocityY(Math.abs(sprite.getVelocityY()));
        }
        else if (sprite.getY() + sprite.getHeight() >=
            screen.getHeight())
        {
            sprite.setVelocityY(-Math.abs(sprite.getVelocityY()));
        }

        // 更新幽灵
        sprite.update(elapsedTime);
    }

    public void draw(Graphics g) {
        // 绘制背景
        g.drawImage(bgImage, 0, 0, null);

        //绘制幽灵
        g.drawImage(sprite.getImage(),
            Math.round(sprite.getX()),
            Math.round(sprite.getY()),
            null);
```

```
    }
}
```

由于 Sprite 对象处理自己的运动，因此 SpriteTest1 类中没有什么新东西。最新的东西是 update()方法，使幽灵碰到屏幕边沿时弹回。幽灵碰到屏幕左右边沿时，水平速度变成相反值。幽灵碰到屏幕上下边沿时，垂直速度变成相反值。

2.4　简单效果

要想增加更多的幽灵，只要生成更多 Sprite 对象，并更新和绘制每个 Sprite 对象，实现代码如下所示。

```
for (int i=0; i<sprites.length; i++) {
    sprites[i].update(elapsedTime);
        g.drawImage(sprite[i].getImage(),
            Math.round(sprite[i].getX()),
            Math.round(sprite[i].getY()),
            null);
    }
```

值得注意的是，因为幽灵更新自己的动画，所以不能共享同一个 Animation 对象，否则会使 Animation 对象更新太多次。如果要使用大量相同的动画，则可以增加一个 clone()方法，使动画更容易复制。

图像变换

一个精彩的效果是图像旋转和比例缩放，可称为图像变换（image transform）。图像变换可以使图像平移、翻折、缩放、剪切和旋转，甚至可以动态进行。

可以用 AffineTransform 对象描述图像变换。AffineTransform 对象用 3x3 矩阵存储变换数据，但不必了解矩阵如何作用于图形，因为 AffineTransform 类提供 rotate(),scale()和 translate()等简单方法，就可以帮助进行计算了。

图像变换实际发生在 Graphics2D 对象中，Graphics2D 类有几个特殊的 drawImage()方法，方法取得 AffineTransform 作为参数，下面的实例将图像画成 2 倍大小，实现代码如下所示。

```
AffineTransform transform = new AffineTransform();
transform.scale(2,2);
transform.translate(100,100);
g.drawImage(image, transform,null);
```

注意：变换操作基于原点，即图像的左上角位置。这样，可能要通过平移图像来取得所要的结果。

下面在 SpriteTest1 对象中增加几个幽灵和一些图像变换。

角色通常面朝右边，但移到左边时，也可以让角色面朝左边。这里不必装入一幅面朝左边的图像，而是通过图像变换动态生成图像的镜像。为此，用-1 缩放图像宽度，然后将

结果平移，使其放到正确位置，实现代码如下所示。

```
transform.scale(-1, 1);
transform.translate(-sprite.getWidth(), 0);
```

为了增加趣味，清单 2.11 的 SpriteTest2 类还增加了淡化效果。当程序首次启动时，图形从黑色变淡，就像打开水平百叶窗时一样，程序退出时则相反。要生成这个效果，可以用 Graphics 对象的 fillRect()方法生成实体黑色“百叶窗”，它的尺寸取决于运行演示的时间量。图 2-13 是 SpriteTest2 的屏幕图形。

图 2-13　SpriteTest2 同时动画几个幽灵，并通过图像变换动态生成图像的镜像

清单 2.11　SpriteTest2.java

```
import java.awt.*;
import java.awt.geom.AffineTransform;
import javax.swing.ImageIcon;

public class SpriteTest2 {

    public static void main(String args[]) {
        SpriteTest2 test = new SpriteTest2();
        test.run();
    }

    private static final DisplayMode POSSIBLE_MODES[] = {
        new DisplayMode(800, 600, 32, 0),
        new DisplayMode(800, 600, 24, 0),
        new DisplayMode(800, 600, 16, 0),
        new DisplayMode(640, 480, 32, 0),
        new DisplayMode(640, 480, 24, 0),
        new DisplayMode(640, 480, 16, 0)
    };
```

```
    private static final long DEMO_TIME = 10000;
    private static final long FADE_TIME = 1000;
    private static final int NUM_SPRITES = 3;

    private ScreenManager screen;
    private Image bgImage;
    private Sprite sprites[];

    public void loadImages() {
        // 装入回像
        bgImage = loadImage("images/background.jpg");
        Image player1 = loadImage("images/player1.png");
        Image player2 = loadImage("images/player2.png");
        Image player3 = loadImage("images/player3.png");

        // 生成与初始化幽灵
        sprites = new Sprite[NUM_SPRITES];
        for (int i = 0; i < NUM_SPRITES; i++) {
            Animation anim = new Animation();
            anim.addFrame(player1, 250);
            anim.addFrame(player2, 150);
            anim.addFrame(player1, 150);
            anim.addFrame(player2, 150);
            anim.addFrame(player3, 200);
            anim.addFrame(player2, 150);
            sprites[i] = new Sprite(anim);

            // 选择随机开始位置
            sprites[i].setX((float)Math.random() *
                (screen.getWidth() - sprites[i].getWidth()));
            sprites[i].setY((float)Math.random() *
                (screen.getHeight() - sprites[i].getHeight()));

            // 选择随机速度
            sprites[i].setVelocityX((float)Math.random() - 0.5f);
            sprites[i].setVelocityY((float)Math.random() - 0.5f);
        }
    }
```

```
    private Image loadImage(String fileName) {
        return new ImageIcon(fileName).getImage();
    }

    public void run() {
        screen = new ScreenManager();
        try {
            DisplayMode displayMode =
                screen.findFirstCompatibleMode(POSSIBLE_MODES);
            screen.setFullScreen(displayMode);
            loadImages();
            animationLoop();
        }
        finally {
            screen.restoreScreen();
        }
    }
    public void animationLoop() {
        long startTime = System.currentTimeMillis();
        long currTime = startTime;

        while (currTime - startTime < DEMO_TIME) {
            long elapsedTime =
                System.currentTimeMillis() - currTime;
            currTime += elapsedTime;

            // 更新幽灵
            update(elapsedTime);

            // 绘制与更新屏幕
            Graphics2D g = screen.getGraphics();
            draw(g);
            drawFade(g, currTime - startTime);
            g.dispose();
            screen.update();

            // 休眠一会儿
            try {
                Thread.sleep(20);
```

```
            }
            catch (InterruptedException ex) { }
        }
    }

    public void drawFade(Graphics2D g, long currTime) {
        long time = 0;
        if (currTime <= FADE_TIME) {
            time = FADE_TIME - currTime;
        }
        else if (currTime > DEMO_TIME - FADE_TIME) {
            time = FADE_TIME - DEMO_TIME + currTime;
        }
        else {
            return;
        }

        byte numBars = 8;
        int barHeight = screen.getHeight() / numBars;
        int blackHeight = (int)(time * barHeight / FADE_TIME);

        g.setColor(Color.black);
        for (int i = 0; i < numBars; i++) {
            int y = i * barHeight + (barHeight - blackHeight) / 2;
            g.fillRect(0, y, screen.getWidth(), blackHeight);
        }
    }

    public void update(long elapsedTime) {

        for (int i = 0; i < NUM_SPRITES; i++) {

            Sprite s = sprites[i];

            // 检查幽灵边界
            if (s.getX() < 0.) {
                s.setVelocityX(Math.abs(s.getVelocityX()));
            }
            else if (s.getX() + s.getWidth() >=
```

```
                screen.getWidth())
            {
                s.setVelocityX(-Math.abs(s.getVelocityX()));
            }
            if (s.getY() < 0) {
                s.setVelocityY(Math.abs(s.getVelocityY()));
            }
            else if (s.getY() + s.getHeight() >=
                screen.getHeight())
            {
                s.setVelocityY(-Math.abs(s.getVelocityY()));
            }

            // 更新幽灵
            s.update(elapsedTime);
        }
    }
    public void draw(Graphics2D g) {
        // 绘制背景
        g.drawImage(bgImage, 0, 0, null);

        AffineTransform transform = new AffineTransform();
        for (int i = 0; i < NUM_SPRITES; i++) {
            Sprite sprite = sprites[i];

            // 平移出灵
            transform.setToTranslation(sprite.getX(),
                sprite.getY());

            // 幽灵移到左边时，翻转图像
            if (sprite.getVelocityX() < 0) {
                transform.scale(-1, 1);
                transform.translate(-sprite.getWidth(), 0);
            }

            // 画幽灵
            g.drawImage(sprite.getImage(), transform, null);
        }
    }
```

```
}
```

在 SpriteTest2 类代码中，对于幽灵图像动态翻转，这里还有一个问题：图像变换不利用硬件加速，即使像平移等简单的图像变换，也不能加速图像。因此，应该少使用图像变换。

注意：这些幽灵演示程序中的 2 个问题：

第一，尽管让所有幽灵保持在屏幕上，但也可以让幽灵部分画到屏外或完全移到屏外，因为 Java2D 中提供了剪取功能；

第二，用系统时钟控制动画，这在 Windows 平台上不够细致。后面第 16 章“优化技术”中将介绍如何解决这个问题。

2.5　小结

本章介绍了许多内容，学也得到了一些有用的可复用代码：ScreenManager 类，Animation 类和 Sprite 类，我们在后面的章节中会用到它们。

记住，这些类不一定是表示的最佳做法，你也可以生成更适合具体需求的代码。记住做一项工作没有“惟一正确”的方法，因此读者不妨多多尝试，寻找更适合具体需求和特定游戏的方法。

第 3 章　交互性与用户界面

关键主题

- AWT 事件模型
- 键盘输入
- 鼠标输入
- 通过鼠标移动的观看模式
- 创建输入管理器
- 使用输入管理器
- 设计直观的用户界面
- 使用 Swing 组件
- 创建简单菜单
- 让游戏者配置键盘
- 小结

如果曾经自己单独玩过象棋游戏的话，就会知道没有交互性的游戏是什么滋味了。交互性实际上就是取得用户输入和根据输入所改变屏幕显示的内容。如果没有交互性，则根本没有游戏性，至少游戏太乏味了。

本章介绍如何取得用户输入，即键盘与鼠标事件。还要介绍把输入集成到游戏中，并介绍如何用 Swing 实现用户界面。

Java SDK 1.4 中没有从游戏摇杆取得输入的功能，尽管这是个缺点，但没有多大影响，因为多数计算机上本没有游戏摇杆，也许多数人更喜欢使用键盘与鼠标呢。

开始介绍之前，首先要说明所用的代码。本书余下部分会把可复用代码组织到光盘中的 com.brackeen.javagamebook 子包中，使代码保持组织，更加便于寻找。

第 2 章“二维图形与动画”生成的 ScreenManager，Animation 和 Sprite 类存放在 com.brackeen.javagamebook.graphics package 包中。本章生成的可复用代码放在 com.brackeen.javagamebook.input 包中。另外，任何仅提供测试的代码将放在缺省无名包中。

最后，要用一个简单类来简化快速测试程序的实现，清单 3.1 的 GameCore 类就是起这个作用。它实现上一章中一些常见的技术，如设置显示方式和运行动画循环等。这里只是扩展这个抽象类，并实现 draw()与 update()方法。

清单 3.1　GameCore.java

```
package com.brackeen.javagamebook.test;

import java.awt.*;
import javax.swing.ImageIcon;
import com.brackeen.javagamebook.graphics.ScreenManager;
```

```
/**
    测试使用的简单抽象类。子类要实现 draw()方法
*/
public abstract class GameCore {

    protected static final int FONT_SIZE = 24;

    private static final DisplayMode POSSIBLE_MODES[] = {
        new DisplayMode(800, 600, 32, 0),
        new DisplayMode(800, 600, 24, 0),
        new DisplayMode(800, 600, 16, 0),
        new DisplayMode(640, 480, 32, 0),
        new DisplayMode(640, 480, 24, 0),
        new DisplayMode(640, 480, 16, 0)
    };

    private boolean isRunning;
    protected ScreenManager screen;

    /**
        告诉游戏循环该退出了
    */
    public void stop() {
        isRunning = false;
    }

    /**
        调用 init() 与 gameLoop()方法
    */
    public void run() {
        try {
            init();
            gameLoop();
        }
        finally {
            screen.restoreScreen();
        }
    }
```

```
/**
    设置全屏方式，将对象初始化
*/
public void init() {
    screen = new ScreenManager();
    DisplayMode displayMode =
        screen.findFirstCompatibleMode(POSSIBLE_MODES);
    screen.setFullScreen(displayMode);

    Window window = screen.getFullScreenWindow();
    window.setFont(new Font("Dialog", Font.PLAIN, FONT_SIZE));
    window.setBackground(Color.blue);
    window.setForeground(Color.white);

    isRunning = true;
}

public Image loadImage(String fileName) {
    return new ImageIcon(fileName).getImage();
}

/**
    运行游戏循环，直到调用 stop()方法
*/
public void gameLoop() {
    long startTime = System.currentTimeMillis();
    long currTime = startTime;

    while (isRunning) {
        long elapsedTime =
            System.currentTimeMillis() - currTime;
        currTime += elapsedTime;

        // 更新
        update(elapsedTime);

       //绘制屏幕
        Graphics2D g = screen.getGraphics();
        draw(g);
```

```
            g.dispose();
            screen.update();

            // 休眠一会儿
            try {
                Thread.sleep(20);
            }
            catch (InterruptedException ex) { }
        }
    }

    /**
        根据经过的时间量更新游戏/动画状态
    */
    public void update(long elapsedTime) {
        // 什么也不干
    }

    /**
        绘制到屏幕上，子类要覆盖这个方法
    */
    public abstract void draw(Graphics2D g);
}
```

缺省情况下，update()方法什么也不干，但子类中用其更新幽灵技术。另外，可以扩展init()方法，装入图像和初始化任何变量。

最后一点要说明的是，从现在开始，代码可以用 Apache Ant 编译。在本书源代码中，把源代码放在 src 文件夹中，Ant 在 Apache 文件夹中编译这个类。如果不熟悉 Ant，可以回到前言或访问 http://ant.apache.org 站点。

下面开始介绍本章的主题——交互性。

3.1　AWT 事件模型

前面曾介绍过 Ant 有自己的事件派生线程，这个线程派生出各种事件，包括来自操作系统中鼠标和键盘的单击事件。

那 AWT 什么时候派生这些事件呢？当特定组件发生一个事件时，AWT 检查为事件准备的监听器（listeners）。监听器是从另一对象接收事件的对象，这里事件来自几个事件派生线程。

每种事件都用不同类型的监听器，如键盘输入事件的监听器是 KeyListener 接口。下面是键盘单击事件在 AWT 事件模型中的工作步骤。

（1）用户单击一个按键。

（2）操作系统向 Java 运行环境发送一个键击事件。

（3）Java 运行环境将键击事件发送到 AWT 事件队列。

（4）AWT 事件派生线程将键击事件发送到任何 KeyListener。

（5）KeyListener 收到键击事件，并对其进行相应处理。

所有监听器都是接口，因此任何对象都可以成为监听器，只要实现监听器接口即可。另外，同一事件类型可以有多个监听器，如允许几个对象同时监听鼠标事件，虽然这是个有用的特性，但代码中不需要对同一事件类型处理多个监听器。

可以捕获所有 AWT 事件，尽管这在实际游戏中用处不大，但可以帮助调试代码和显示实际派生的事件。下列代码捕获所有 AWT 事件，为此它生成 AWTEventListener 对象并将事件打印到控制台上，实现代码如下所示。

```
Toolkit.getDefaultToolkit().addAWTEventListener(
    New AWTEventListener(){
        Public void eventDispatched(AWTEvent event){
            System.out.println(event);
        }
},-1);
```

记住，不要在实际游戏中使用这类代码，而只能在代码调试时使用。

3.2 键盘输入

游戏中需要使用许多按键，如使用箭头方向键移动，Ctrl 键发动武器等。虽然这里不必处理文本输入，但文本输入将会在本章稍后介绍的 Swing 组件中处理。

要捕获键击事件，就要做 2 件事情：生成 KeyListener 和将监听器注册成接收事件。若要注册监听器，则调用键击事件所在组件的 addKeyListener()方法。对这里的游戏而言，这个组件就是全屏窗口，注册监听器的代码如下所示。

```
Window window =screen.getFullScreenWindow();
Window addKeyListener(keyListener);
```

若要生成 KeyListener，只要生成实现 KeyListener 接口的对象，KeyListener 接口包括 3 种方法：keyPressed()，keyReleased()和 keyTyped()。Typed 事件在首次击键时发生，然后在重复击键时将根据重复的速率发生。接收 Typed 事件对游戏毫无用处，因此主要介绍 Pressed（按下）和 Released（放开）事件。

这 3 个方法都取 KeyEvent 为参数。KeyEvent 对象可以检查实际按下与放开的键，它采用虚拟键码（virtu-al key code）。虚拟键码是特定盘键的 Java 定义码，但不同于字符，如 Q 与 q 是不同字符，但其虚拟键码是相同的。

所有虚拟键码在 KeyEvent 中以 VK xxx 形式定义，如 Q 键的键码为 KeyEvent.VK_Q。通常可以猜出虚拟键码名，如 VK__ENTER 与 VK1。虚拟键码的完整清单在 Java API 文档的 KeyEvent 类中。

清单 3.2 的 KeyTest 类实现 KeyListener 接口，它只是在屏幕上显示键按下与放开事件，

还有可以按 Escape 键退出程序。

清单 3.2　KeyTest.java

```
import java.awt.*;
import java.awt.event.KeyListener;
import java.awt.event.KeyEvent;
import java.util.LinkedList;

import com.brackeen.javagamebook.graphics.*;
import com.brackeen.javagamebook.test.GameCore;

/**
    简单键盘测试。在屏幕上显示键按下与放开事件。也用于调试键输入
*/
public class KeyTest extends GameCore implements KeyListener {

    public static void main(String[] args) {
        new KeyTest().run();
    }

    private LinkedList messages = new LinkedList();

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

        Window window = screen.getFullScreenWindow();

        // 可以输入 TAB 键和其他通常用于移动焦点的键
        window.setFocusTraversalKeysEnabled(false);

        // 将这个对象注册为窗口的键监听器

        addMessage("KeyInputTest. Press Escape to exit");
    }

    // KeyListner 接口中的方法
    public void keyPressed(KeyEvent e) {
        int keyCode = e.getKeyCode();
```

```
        // 退出程序
        if (keyCode == KeyEvent.VK_ESCAPE) {
            stop();
        }
        else {
            addMessage("Pressed: " +
                KeyEvent.getKeyText(keyCode));

            // 保证不对其他事项处理键
            e.consume();
            }
    }

    // KeyListner 接口中的方法
    public void keyReleascd(KeyEvent e) {
        int keyCode = e.getKeyCode();
        addMessage("Released: " + KeyEvent.getKeyText(keyCode));

        // 保证不对其他事项处理键
         e.consume();
    }

    // KeyListener 接口中的方法
    public void keyTyped(KeyEvent e) {
        // 在放开键之后调用，忽略
        // 保证不对其他事项处理键
    }

    /**
        将消息加进消息清单
    */
    public synchronized void addMessage(String message) {
        messages.add(message);
        if (messages.size() >= screen.getHeight() / FONT_SIZE) {
            messages.remove(0);
        }

    }
```

```
/**
    绘制消息清单
*/
public synchronized void draw(Graphics2D g) {

    Window window = screen.getFullScreenWindow();

    g.setRenderingHint(
        RenderingHints.KEY_TEXT_ANTIALIASING,
        RenderingHints.VALUE_TEXT_ANTIALIAS_ON);

    // 绘制背景
    g.setColor(window.getBackground());
    g.fillRect(0, 0, screen.getWidth(), screen.getHeight());

    // 绘制消息
    g.setColor(window.getForeground());
    int y = FONT_SIZE;
    for (int i=0; i<messages.size(); i++) {
        g.drawString((String)messages.get(i), 5, y);
        y+=FONT_SIZE;
    }
  }
}
```

在 KeyTest 类代码中可以看出 2 点问题。

第一，init()方法使用如下语句。

```
window.setFocusTraversalkeysEnabled(false);
```

代码关闭了焦点遍历键。当按下焦点遍历键时，键盘焦点从一个组件移动到另一组件，如在 Web 页面中，可以按 Tab 键从窗体上一个字段移到另一字段。通常，Tab 键事件由 AWT 焦点遍历代码处理。由于实例中要接收 Tab 键事件，所以可以调用这个方法。

如果想了解各个焦点遍历键，可以调用 getFocusTraversalKeys()方法。

除了 Tab 键之外，还有几个会产生奇怪行为的键，如 Alt 键。在大多数系统中，Alt 键激活助记符（keyTest）。助记符是特定用户界面元素的快捷键或加速键，如有菜单栏的应用程序中，按 Alt+F 激活 File 菜单。当按下 Alt 键后，AWT 会认为下一个按键就是助记符，所以将忽略 Alt 键。为了关闭这个特性，就要在 keyTest 中阻止每个 keyListener 中的每个 keyEvent 按缺省方式处理，关闭 Alt 键特性的代码如下所示。

```
e.consume();
```

代码保证其他对象不处理 Alt 键（或其他激活助记符的修饰键），从而把 Alt 键当作正常键来处理。

第二，KeyTest 的副作用就是可以测试键在不同系统上的行为。“天哪，不同系统中的键输入还会不同吗”，是的。

下面以重复按键为例，当用户按住一个键时，操作系统会对此键发送多个事件，如在文本编辑器中，按住 Q 键不放，就会出现多个 Q 字符。在有些系统中（如 Linux），当遇到键重复时，都会发送按下和放开事件，而有些系统中（如 Windows）当遇到键重复时，只发送按下事件，只有用户实际释放键时才发送放开事件。

还有其他一些微小差别，如 Windows 系统中的 Start 键随将运行的版本不同而不同，键行为也随运行的 Java 虚拟机不同而不同。

好在这些差别并不太大，通常可以不加考虑。

3.3 鼠标输入

键盘只是上面有一堆键的盘面，而鼠标则复杂得多，不仅有键（根据鼠标类型不同，可能有一个键、两个键、三个键或多个键），还有移动轮，可能还有滚动轮。

可以取得 3 种不同类型的鼠标事件：

- 鼠标键单击事件；
- 鼠标移动事件；
- 鼠标轮滚动事件。

鼠标键单击事件与键盘事件相似，但没有键重复操作。鼠标移动事件可分解成 x 和 y 屏幕坐标。鼠标轮滚动事件表示鼠标轮滚动多远。

每个鼠标事件都有自己的监听器，如：MouseListener，MouseMotinListoner 和 MouseWheelListener，不过监听器都取一个 MouseEvent 作为参数。

和 KeyListener 一样，MouseListener 接口也有探测鼠标接下、放开和单击（按下再放开）的方法，不过在游戏中将忽略鼠标单击事件，就像忽略 KeyTyped 一样，只处理按下与放开事件，可以用 MouseEvent 的 getButton()方法确定按下和放开的键。

另外，MouseListener 接口中还有表示鼠标何时进入与离开组件的方法。由于使用的组件覆盖整个屏幕，因此可以忽略这些方法。

对于鼠标的运动，用 MouseMotinListoner 接口可以探测出 2 种运动事件：普通移动和拖动，其中拖动事件发生在按住键并移动鼠标时。这 2 种运动都可以调用 MouseEvent 类的 getX()和 getY()方法取得鼠标当前的位置。

MouseWheelListener 用 MouseEvent 的子类 MouseWheelEvent，其 getWheelRotation()方法检查鼠标轮运动量，负值表示向上滚动，正值表示向下滚动。

在介绍鼠标输入的基本知识后，下面就要生成一个测试程序。清单 3.3 的 MouseTest 程序在鼠标所在位置画一个 Hello World 消息。单击时，它变成 trail mode（拖尾方式），并画出最近 10 个鼠标的位置，从而产生一个“尾巴”。同时，鼠标轮滚动会改变文本颜色。和前面一样，也可以按 Escape 键退出程序。

清单 3.3 MouseTest.java

```
import java.awt.*;
```

```
import java.awt.event.*;
import java.util.LinkedList;

import com.brackeen.javagamebook.graphics.*;
import com.brackeen.javagamebook.test.GameCore;

/**
    简单鼠标测试。在鼠标所在位置画一个“Hello World”消息。单击时，它变成“trail mode”（拖
    尾方式），画出最近 10 个鼠标位置，从而产生一个“尾巴”（如有）
*/
public class MouseTest extends GameCore implements KeyListener,
    MouseMotionListener, MouseListener, MouseWheelListener

{

    public static void main(String[] args) {
        new MouseTest().run();
    }

    private static final int TRAIL_SIZE = 10;
    private static final Color[] COLORS = {
        Color.white, Color.black, Color.yellow, Color.magenta
    };

    private LinkedList trailList;
    private boolean trailMode;
    private int colorIndex;

    public void init() {
        super.init();
        trailList = new LinkedList();

        Window window = screen.getFullScreenWindow();
        window.addMouseListener(this);
        window.addMouseMotionListener(this);
        window.addMouseWheelListener(this);
        window.addKeyListener(this);
    }
    public synchronized void draw(Graphics2D g) {
```

```
int count = trailList.size();

if (count > 1 && !trailMode) {
    count = 1;
}

Window window = screen.getFullScreenWindow();

//画背景
g.setColor(window.getBackground());
g.fillRect(0, 0, screen.getWidth(), screen.getHeight());

// 画指令
g.setRenderingHint(
        RenderingHints.KEY_TEXT_ANTIALIASING,
        RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
g.setColor(window.getForeground());
g.drawString("MouseTest. Press Escape to exit.", 5,
    FONT_SIZE);

    // 画鼠标尾巴
    for (int i=0; i<count; i++) {
        Point p = (Point)trailList.get(i);
        g.drawString("Hello World!", p.x, p.y);
    }
}

// 来自 MouseListener 接口
public void mousePressed(MouseEvent e) {
    trailMode = !trailMode;
}

// 来自 MouseListener 接口
public void mouseReleased(MouseEvent e) {
    // 什么也不干

}

//来自 MouseListener 接口
```

```
public void mouseClicked(MouseEvent e) {
    // 在释放鼠标后调用－忽略
}

// 来自 MouseListener 接口
public void mouseEntered(MouseEvent e) {
    mouseMoved(e);
}

// 来自 MouseListener 接口
public void mouseExited(MouseEvent e) {
    mouseMoved(e);
}

// 来自 MouseMotionListener 接口
public void mouseDragged(MouseEvent e) {
    mouseMoved(e);
}

//来自 MouseMotionListener 接口
public synchronized void mouseMoved(MouseEvent e) {
    Point p = new Point(e.getX(), e.getY());
    trailList.addFirst(p);
    while (trailList.size() > TRAIL_SIZE) {
        trailList.removeLast();
    }
}

// 来自 MouseWheelListener 接口
public void mouseWheelMoved(MouseWheelEvent e) {
    colorIndex = (colorIndex + e.getWheelRotation()) %
        COLORS.length;

    if (colorIndex < 0) {
        colorIndex+=COLORS.length;
    }
    Window window = screen.getFullScreenWindow();
    window.setForeground(COLORS[colorIndex]);
}
```

```
        // 来自 KeyListener 接口
        public void keyPressed(KeyEvent e) {
            if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
                // exit the program
                stop();
            }
        }

        //来自 KeyListener 接口
        public void keyReleased(KeyEvent e) {
            //什么也不干
        }

        //来自 KeyListener 接口
        public void keyTyped(KeyEvent e) {
            // 什么也不干
        }
    }
```

MouseTest 程序的代码很简单，没有什么复杂之处。当鼠标移动时，程序将新的 Point 对象加进 trailList 中去。Point 对象包含 x 与 y 值，清单中至多保存 10 个 Point 对象。如果打开拖尾方式，则 drsw()方法在清单中每个 Point 位置画出 Hello World 消息，否则只第一个 Point 位置画出 Hello World 消息。单击鼠标可以切换开/关拖尾方式。

3.4 通过鼠标移动的观看模式

如果运行像 Quake III 等第一人称视角的射击游戏时，就会使用鼠标观看（mouselock）特性，它使游戏者可以移动鼠标器达到对视野的观看，如向左移动鼠标，游戏者则向左看。鼠标观看特性的妙处所在是可以随意移动鼠标，由于鼠标不受屏幕的限制，就可以一直移动鼠标，直到头晕目眩或移出鼠标板为止。

MouseTest 程序可以探测屏幕上鼠标的位置，但鼠标若移得太远时，就会碰上屏幕边沿从而停下来。这显然不适合第一人称视角的射击游戏，因为不能限制游戏者的观看范围。所以要达到的目的不是探测鼠标的绝对位置，而要探测鼠标相对移动，那如何实现这个功能呢？

虽然 Java API 没有直接提供探测鼠标相对移动的方法，但可以换个角度想。首先保证鼠标不移出屏幕外，然后每次鼠标移动时，只要将鼠标移到屏幕的中央。这样，鼠标就不会因为碰上屏幕边沿而停下来，即总是可以根据上一位置来计算鼠标的移动量。下面就是实现具体的步骤。

（1）鼠标从屏幕中央开始。

（2）用户移动鼠标，就要计算鼠标的移动量。

（3）发一个事件，将鼠标移到屏幕中央。

（4）发现鼠标重新居中后，将忽略此事件。

可以用 Robot 类移动鼠标位置，Robot 类并不会派一个机器人到桌面上帮你移动鼠标，而是一个自动测试图形的应用程序。除了通过程序移动鼠标外，还有各种其他的功能，如传真键击和进行屏幕捕获。移动鼠标的方法很简单，实现代码如下所示。

```
robot.mouseMove(x,y);
```

注意：在极少数系统中 Robot 类可能无法使用。因为系统不能通过程序来移动鼠标和传真键击，但大多数的 Windows 与 Linux 系统都可以使用 Robot 类。

下面要对 Robot 类作一个测试。虽然不是生成第一人称视角的射击游戏，但也要生成一个简单的背景纹理，它可以用鼠标观看特性无限滚动。清单 3.4 的 MouselookTest 类就是实现这个作用的，可以按空格键关掉鼠标观看特性，还可以按 Escape 键退出程序。

清单 3.4　MouselookTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.SwingUtilities;

import com.brackeen.javagamebook.graphics.*;
import com.brackeen.javagamebook.test.GameCore;

/**
    简单鼠标观看特性测试程序。利用鼠标观看特性，用户可以按任意方向任意移动鼠标，鼠标遇
    到屏幕边沿时停止
    鼠标观看特性在鼠标移动时将鼠标居中，使其总是可以测量相对鼠标运动，使鼠标不会遇到屏
    幕边沿
*/
public class MouselookTest extends GameCore
    implements MouseMotionListener, KeyListener
{

    public static void main(String[] args) {
        new MouselookTest().run();
    }

    private Image bgImage;
    private Robot robot;
    private Point mouseLocation;
    private Point centerLocation;
    private Point imageLocation;
```

```
    private boolean relativeMouseMode;
    private boolean isRecentering;

    public void init() {
        super.init();
        mouseLocation = new Point();
        centerLocation = new Point();
        imageLocation = new Point();

        relativeMouseMode = true;
        isRecentering = false;

        try {
            robot = new Robot();
            recenterMouse();
            mouseLocation.x = centerLocation.x;
            mouseLocation.y = centerLocation.y;
        }
        catch (AWTException ex) {
            System.out.println("Couldn't create Robot!");
        }
        Window window = screen.getFullScreenWindow();
        window.addMouseMotionListener(this);
        window.addKeyListener(this);
        bgImage = loadImage("../images/background.jpg");
    }

    public synchronized void draw(Graphics2D g) {

        int w = screen.getWidth();
        int h = screen.getHeight();

        // 保证位置正确
        imageLocation.x %= w;
        imageLocation.y %= screen.getHeight();
        if (imageLocation.x < 0) {
            imageLocation.x += w;
        }
        if (imageLocation.y < 0) {
```

```
            imageLocation.y += screen.getHeight();
        }

        // 在四个位置画图，盖住屏幕
        int x = imageLocation.x;
        int y = imageLocation.y;
        g.drawImage(bgImage, x, y, null);
        g.drawImage(bgImage, x-w, y, null);
        g.drawImage(bgImage, x, y-h, null);
        g.drawImage(bgImage, x-w, y-h, null);

        // 画指令
        g.setRenderingHint(
                RenderingHints.KEY_TEXT_ANTIALIASING,
                RenderingHints.VALUE_TEXT_ANTIALIAS_ON);
        g.drawString("Press Space to change mouse modes.", 5,
            FONT_SIZE);
        g.drawString("Press Escape to exit.", 5, FONT_SIZE*2);
    }

    /**
        用 Robot 类将鼠标放到屏幕中央。注意并不是所有平台都支持使用这个类
    */
    private synchronized void recenterMouse() {
        Window window = screen.getFullScreenWindow();
        if (robot != null && window.isShowing()) {
            centerLocation.x = window.getWidth() / 2;
            centerLocation.y = window.getHeight() / 2;
            SwingUtilities.convertPointToScreen(centerLocation,
                window);
            isRecentering = true;
            robot.mouseMove(centerLocation.x, centerLocation.y);
        }
    }

    //来自 MouseMotionListener 接口
    public void mouseDragged(MouseEvent e) {
        mouseMoved(e);
    }
```

```
//来自 MouseMotionListener 接口
public synchronized void mouseMoved(MouseEvent e) {
    // 这个事件用于鼠标居中一忽略
    if (isRecentering &&
        centerLocation.x == e.getX() &&
        centerLocation.y == e.getY())
    {
        isRecentering = false;
    }
    else {
        int dx = e.getX() - mouseLocation.x;
        int dy = e.getY() - mouseLocation.y;
        imageLocation.x += dx;
        imageLocation.y += dy;
        // 鼠标居中
        if (relativeMouseMode) {
            recenterMouse();
        }

    }

    mouseLocation.x = e.getX();
    mouseLocation.y = e.getY();

}

// 来自 KeyListener 接口
public void keyPressed(KeyEvent e) {
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
        // 退出程序
        stop();
    }
    else if (e.getKeyCode() == KeyEvent.VK_SPACE) {
        // 改变相对鼠标方式
        relativeMouseMode = !relativeMouseMode;
    }
}
// from the KeyListener interface
public void keyReleased(KeyEvent e) {
```

```
            //什么也不干
        }

        // from the KeyListener interface
        public void keyTyped(KeyEvent e) {
            //什么也不干
        }
    }
```

在 MouselookTest 类代码中，如果不能生成 Robot 对象，则会抛出 AWTException 异常。只要将异常忽略，向控制台打印一个消息即可。

当 Robot 类移动鼠标时，不一定立即发生鼠标移动事件。有些正常鼠标移动事件可能在居中事件之前发送。因此，代码就要检查鼠标移动事件是否使鼠标移到屏幕中央，是则作为居中事件，并将事件忽略，否则把事件作为普通鼠标移动事件。

隐藏光标

在 MouselookTest 程序中，注意到屏幕中的鼠标光标不太好看。居中时它可能会闪烁，可能在短时间看到鼠标移动，下面就要隐藏鼠标光标。

好在 Java API 提供改变鼠标光标的方法，但 Java API 中没有定义隐藏光标。Java API 中定义的鼠标光标包括：

CROSSHAIR_CURSOR　光标显示加号；
DEFAULT_CURSOR　正常箭头光标；
HAND_CURSOR　手形光标，常见于鼠标在 Web 页面超链接上时；
TEXT_CURSOR　文本光标，通常为 I 字形；
WAIT_CURSOR　等待光标，通常为沙漏形。

为了完整起见，Java API 允许定制图像来生成自己的鼠标光标，所以只要生成空白图像的光标即可。为此，调用 Toolkit 类的 createCustomCursor（）方法，实现代码如下所示。

```
Cursor invisibleCursor =
    Toolkit.getDefaultToolkit().createCustomCursor(
    Toolkit.getDefaultToolkit().getImage(“”),
    new Point(0,0),
    “invisible”;
```

实际上，代码只是生成作废的图像，但工具箱却将其解释为隐藏光标。若要改变游戏的光标，可以用 setCursor 方法，实现代码如下所示。

```
Window window =screen.getFullScreenWindow();
Window.setCursor(invisibleCursor)
```

后面可以用 Cursor 的 getPredefinedCursor()方法返回缺省光标。

```
Cursor   normalCursor =
    Cursor.getPredefinedCursor(Cursor.DEFAULT_CURSOR)
```

使用现在的代码就可以不再看到讨厌的光标了。

3.5 创建输入管理器

掌握键盘和鼠标的输入后，下面就要综合起来，生成一个输入管理器，而且上述实例中的缺点都要在输入管理器中加以解决。

也许你已经发现，上述实例中全部使用同步方法实现。记住，所有事件都是 AWT 事件派生线程发生的，它与主线程不是同一线程。因为不希望在绘图操作的中途修改游戏状态（如改变幽灵位置），所以要使用同步方法，防止发生这类情形。

但是在今后的游戏中，要在游戏循环的特定时刻要处理所有的输入，即在按下键时就要设置布尔变量，如按下空格键时，那么可以在 keyPressed()方法中设置 jumpIsPressed 布尔变量，然后在游戏循环的后面部分检查是否设置了 jumpIsPressed 变量，是则代码可以让游戏者跳动。

另外，对于某些操作（如跳动）只对初始键击进行操作，而对另外一些操作（如移动）则按下键时就要移动。跳动时，如果要探测初始键击跳动，则可以在游戏循环中看到 jumpIsPressed 值为 true 时，将其设置为 false。这样，游戏者只有再次按键时才会跳动。

如果用户不喜欢按空格键跳动，而要按 Ctrl 键呢？很简单，只要告诉他，“对不起，不能这么干”。也许是处于好心，你允许用户配置键盘。为此，就要将一般性操作贴图到一个键上，达到让这个操作动态贴图到不同键。目前可以把这个功能放在 InputManager 类的功能表中。

总之，希望输入管理器（InputManager）具有下列功能：

- 处理所有键盘与鼠标事件，包括相对鼠标运动；
- 保存事件，以便要需要时精确地查询，而不是在 AWT 事件派生线程中改变游戏状态；
- 探测某些键的初击，探测其他键的连续按下；
- 将键贴图为一般性游戏操作，如将空格键贴图为跳动操作；
- 动态改变键贴图，使用户可以配置键盘。

首先要看看一般性游戏。清单 3.5 的 GameAction 类很容易使用，其中 isPressed()方法用于键击，getAmount()方法显示鼠标移动量。GameAction 类还处理初击和正常键击行为。最后，GameAction 类提供了让 Input Manager 调用的方法，如 press()与 release()方法。

清单 3.5 GameAction.java

```
package com.brackeen.javagamebook.input;

/**
    GameAction 类抽象用户启动的操作，如跳动，移动。 GameActions 可以用 InputManager 贴图
    键与鼠标
*/
public class GameAction {

    /**
```

```
    正常行为，按下键时 isPressed() 方法返回真值
*/
public static final int NORMAL = 0;

/**
    初击行为，isPressed()方法只在初击后返回真值，然后要放开和再次按下时才返回真值
*/
public static final int DETECT_INITAL_PRESS_ONLY = 1;

private static final int STATE_RELEASED = 0;
private static final int STATE_PRESSED = 1;
private static final int STATE_WAITING_FOR_RELEASE = 2;

private String name;
private int behavior;
private int amount;
private int state;

/**
    生成新的 GameAction，具有正常行为
*/
public GameAction(String name) {
    this(name, NORMAL);
}

/**
    生成新的 GameAction ，具有指定行为
*/
public GameAction(String name, int behavior) {
    this.name = name;
    this.behavior = behavior;
    reset();
}

/**
    取得 GameAction 名
*/
public String getName() {
    return name;
```

```
    }

    /**
        复位 GameAction，使其像没有按过一样
    */
    public void reset() {
        state = STATE_RELEASED;
        amount = 0;
    }

    /**
        按击 GameAction。 等于 press() 再调用 release()
    */
    public synchronized void tap() {
        press();
        release();
    }

    /**
        表示键按下
    */
    public synchronized void press() {
        press(1);
    }

    /**
        表示键按下指定次数，或鼠标移动指定距离
    */
    public synchronized void press(int amount) {
        if (state != STATE_WAITING_FOR_RELEASE) {
            this.amount+=amount;
            state = STATE_PRESSED;
        }

    }
    /**
        表示键放开
    */
    public synchronized void release() {
```

```
        state = STATE_RELEASED;
    }

    /**
        表示上次检查以来键是否按下
    */
    public synchronized boolean isPressed() {
        return (getAmount() != 0);
    }

    /**
        上次检查以来键按下的次数，或鼠标移动的距离
    */
    public synchronized int getAmount() {
        int retVal = amount;
        if (retVal != 0) {
            if (state == STATE_RELEASED) {
                amount = 0;
            }
            else if (behavior == DETECT_INITAL_PRESS_ONLY) {
                state = STATE_WAITING_FOR_RELEASE;
                amount = 0;
            }
        }
        return retVal;
    }
}
```

最后，清单 3.6 生成 InputManager 类。InputManager 对象具有上述输入实例中所有的代码，包括隐藏光标和相对鼠标移动。

另外，它还实现将键盘与鼠标事件贴图到 GameAction 对象上。当按下键时，代码检查是否有贴图这个键的 GameAction 对象，有则调用 GameAction 类的 press()方法。

贴图是如何进行呢？生成一个 GameAction 数组，数组中每个索引对应一个虚拟键码。大多数虚拟键码的值小于 600，因此 GameAction 数组的长度为 600。

鼠标事件的贴图也差不多。由于鼠标事件没有代码，因此 InputManager 中建立了伪鼠标代码。鼠标移动（上、下、左、右）、鼠标轮滚动和鼠标键都有鼠标代码。InputManager 类的代码如下所示。

清单 3.6　InputManager.java

```
package com.brackeen.javagamebook.input;
```

```
import java.awt.*;
import java.awt.event.*;
import java.util.List;
import java.util.ArrayList;
import javax.swing.SwingUtilities;

/**
    InputManage 管理键与鼠标事件的输入，事件贴图 GameActions
*/
public class InputManager implements KeyListener, MouseListener,
    MouseMotionListener, MouseWheelListener
{
    /**
        隐藏光标
    */
    public static final Cursor INVISIBLE_CURSOR =
        Toolkit.getDefaultToolkit().createCustomCursor(
            Toolkit.getDefaultToolkit().getImage(""),
            new Point(0,0),
            "invisible");

    //鼠标代码
    public static final int MOUSE_MOVE_LEFT = 0;
    public static final int MOUSE_MOVE_RIGHT = 1;
    public static final int MOUSE_MOVE_UP = 2;
    public static final int MOUSE_MOVE_DOWN = 3;
    public static final int MOUSE_WHEEL_UP = 4;
    public static final int MOUSE_WHEEL_DOWN = 5;
    public static final int MOUSE_BUTTON_1 = 6;
    public static final int MOUSE_BUTTON_2 = 7;
    public static final int MOUSE_BUTTON_3 = 8;

    private static final int NUM_MOUSE_CODES = 9;

    //键码在 java.awt.KeyEvent 中定义，大多数虚拟键码的值小于 600（除了"alt 图"之类少数键码）
    private static final int NUM_KEY_CODES = 600;
    private GameAction[] keyActions =
        new GameAction[NUM_KEY_CODES];
    private GameAction[] mouseActions =
```

```
        new GameAction[NUM_MOUSE_CODES];

    private Point mouseLocation;
    private Point centerLocation;
    private Component comp;
    private Robot robot;
    private boolean isRecentering;

    /**
        生成新 InputManager ，监听指定组件的输入
    */
    public InputManager(Component comp) {
        this.comp = comp;
        mouseLocation = new Point();
        centerLocation = new Point();

        // 注册键与鼠标监听器
        comp.addKeyListener(this);
        comp.addMouseListener(this);
        comp.addMouseMotionListener(this);
        comp.addMouseWheelListener(this);

        // 允许 Tab 键和其他焦点遍历键的输入
        comp.setFocusTraversalKeysEnabled(false);
    }

    /**
        在这个 InputManager 的输入组件上设置光标
    */
    public void setCursor(Cursor cursor) {
        comp.setCursor(cursor);
    }

    /**
        将相对鼠标方式设置为开/关，对相对鼠标方式，鼠标锁在屏幕中央，只测量鼠标移动变
        化。而在正常方式中，鼠标在屏幕上自由移动
    */
    public void setRelativeMouseMode(boolean mode) {
        if (mode == isRelativeMouseMode()) {
```

```
            return;
        }

        if (mode) {
            try {
                robot = new Robot();
                recenterMouse();
            }
            catch (AWTException ex) {
                // couldn't create robot!
                robot = null;
            }
        }
        else {
            robot = null;
        }
    }

    /**
        返回相对鼠标方式是否打开
    */
    public boolean isRelativeMouseMode() {
        return (robot != null);
    }

    /**
        将 GameAction 贴图特定键。键码在 java.awt.KeyEvent 中定义。 如果键已经贴图
        GameAction，则新 GameAction 将其覆盖
    */
    public void mapToKey(GameAction gameAction, int keyCode) {
        keyActions[keyCode] = gameAction;
    }

    /**
        将 GameAction 贴图特定鼠标。鼠标码在 InputManager 中定义 (MOUSE_MOVE_LEFT
        MOUSE_BUTTON_1,等等)。 如果鼠标已经贴图 GameAction，则新 GameAction 将其覆盖
    */
    public void mapToMouse(GameAction gameAction,
        int mouseCode)
```

```
{
    mouseActions[mouseCode] = gameAction;
}

/**
    对这个 GameAction 清除所有贴图的键与鼠标操作
*/
public void clearMap(GameAction gameAction) {
    for (int i=0; i<keyActions.length; i++) {
        if (keyActions[i] == gameAction) {
            keyActions[i] = null;
        }
    }

    for (int i=0; i<mouseActions.length; i++) {
        if (mouseActions[i] == gameAction) {
            mouseActions[i] = null;
        }
    }

    gameAction.reset();
}

/**
    取得这个 GameAction 的键与鼠标操作贴图清单，清单中每个项目是个字符串
*/
public List getMaps(GameAction gameCode) {
    ArrayList list = new ArrayList();
    for (int i=0; i<keyActions.length; i++) {
        if (keyActions[i] == gameCode) {
            list.add(getKeyName(i));
        }

    }

    for (int i=0; i<mouseActions.length; i++) {
        if (mouseActions[i] == gameCode) {
            list.add(getMouseName(i));
        }
```

```
        }
        return list;
    }

    /**
        复位所有 GameActions，使其像没有按下一样
    */
    public void resetAllGameActions() {
        for (int i=0; i<keyActions.length; i++) {
            if (keyActions[i] != null) {
                keyActions[i].reset();
            }
        }

        for (int i=0; i<mouseActions.length; i++) {
            if (mouseActions[i] != null) {
                mouseActions[i].reset();
            }
        }
    }

    /**
        取得键码名
    */
    public static String getKeyName(int keyCode) {
        return KeyEvent.getKeyText(keyCode);
    }

    /**
        取得鼠标代码名
    */
    public static String getMouseName(int mouseCode) {
        switch (mouseCode) {
            case MOUSE_MOVE_LEFT: return "Mouse Left";
            case MOUSE_MOVE_RIGHT: return "Mouse Right";
            case MOUSE_MOVE_UP: return "Mouse Up";
            case MOUSE_MOVE_DOWN: return "Mouse Down";
            case MOUSE_WHEEL_UP: return "Mouse Wheel Up";
            case MOUSE_WHEEL_DOWN: return "Mouse Wheel Down";
```

```
            case MOUSE_BUTTON_1: return "Mouse Button 1";
            case MOUSE_BUTTON_2: return "Mouse Button 2";
            case MOUSE_BUTTON_3: return "Mouse Button 3";
            default: return "Unknown mouse code " + mouseCode;
        }
    }

    /**
        取得鼠标的 x 轴位置
    */
    public int getMouseX() {
        return mouseLocation.x;
    }

    /**
        取得鼠标的 y 轴位置
    */
    public int getMouseY() {
        return mouseLocation.y;
    }

    /**
        用 Robot 类将鼠标放到屏幕中央。注意并不是所有平台都支持使用这个类
    */
    private synchronized void recenterMouse() {
        if (robot != null && comp.isShowing()) {
            centerLocation.x = comp.getWidth() / 2;
            centerLocation.y = comp.getHeight() / 2;
            SwingUtilities.convertPointToScreen(centerLocation,
                comp);
            isRecentering = true;
            robot.mouseMove(centerLocation.x, centerLocation.y);
        }
    }
    private GameAction getKeyAction(KeyEvent e) {
        int keyCode = e.getKeyCode();
        if (keyCode < keyActions.length) {
            return keyActions[keyCode];
        }
```

```
        else {
            return null;
        }
    }

    /**
        取得这个 MouseEvent 所指定鼠标键的鼠标代码
    */
    public static int getMouseButtonCode(MouseEvent e) {
        switch (e.getButton()) {
            case MouseEvent.BUTTON1:
                return MOUSE_BUTTON_1;
            case MouseEvent.BUTTON2:
                return MOUSE_BUTTON_2;
            case MouseEvent.BUTTON3:
                return MOUSE_BUTTON_3;
            default:
                return -1;
        }
    }

    private GameAction getMouseButtonAction(MouseEvent e) {
        int mouseCode = getMouseButtonCode(e);
        if (mouseCode != -1) {
            return mouseActions[mouseCode];
        }
        else {
            return null;
        }
    }

    // 来自 KeyListener 接口
    public void keyPressed(KeyEvent e) {
        GameAction gameAction = getKeyAction(e);
        if (gameAction != null) {
            gameAction.press();
        }
        // 保证不对其他事项处理键
        e.consume();
```

```
}

// 来自 KeyListener 接口
public void keyReleased(KeyEvent e) {
    GameAction gameAction = getKeyAction(e);
    if (gameAction != null) {
        gameAction.release();
    }
    //保证不对其他事项处理键
    e.consume();
}

// 来自 KeyListener 接口
public void keyTyped(KeyEvent e) {
    //保证不对其他事项处理键
    e.consume();
}

// 来自 MouseListener 接口
public void mousePressed(MouseEvent e) {
    GameAction gameAction = getMouseButtonAction(e);
    if (gameAction != null) {
        gameAction.press();
    }
}

// 来自 MouseListener 接口
public void mouseReleased(MouseEvent e) {
    GameAction gameAction = getMouseButtonAction(e);
    if (gameAction != null) {
        gameAction.release();
    }

}

// 来自 MouseListener 接口
public void mouseClicked(MouseEvent e) {
    // 什么也不干
}
```

```
    // 来自 MouseListener 接口
    public void mouseEntered(MouseEvent e) {
        mouseMoved(e);
    }

    // 来自 MouseListener 接口
    public void mouseExited(MouseEvent e) {
        mouseMoved(e);
    }

    //来自 MouseMotionListener 接口
    public void mouseDragged(MouseEvent e) {
        mouseMoved(e);
    }

    // 来自 MouseMotionListener 接口
    public synchronized void mouseMoved(MouseEvent e) {
        // 这个事件用于鼠标居中一忽略
            centerLocation.x == e.getX() &&
            centerLocation.y == e.getY())
        {
            isRecentering = false;
        }
        else {
            int dx = e.getX() - mouseLocation.x;
            int dy = e.getY() - mouseLocation.y;
            mouseHelper(MOUSE_MOVE_LEFT, MOUSE_MOVE_RIGHT, dx);
            mouseHelper(MOUSE_MOVE_UP, MOUSE_MOVE_DOWN, dy);

            if (isRelativeMouseMode()) {
                recenterMouse();
            }
        }

        mouseLocation.x = e.getX();
        mouseLocation.y = e.getY();

    }
```

```
// 来自 MouseWheelListener 接口
public void mouseWheelMoved(MouseWheelEvent e) {
    mouseHelper(MOUSE_WHEEL_UP, MOUSE_WHEEL_DOWN,
        e.getWheelRotation());
}

private void mouseHelper(int codeNeg, int codePos,
    int amount)
{
    GameAction gameAction;
    if (amount < 0) {
        gameAction = mouseActions[codeNeg];
    }
    else {
        gameAction = mouseActions[codePos];
    }
    if (gameAction != null) {
        gameAction.press(Math.abs(amount));
        gameAction.release();
    }
    }
}
```

哇，代码真够长的。不过其中 InputManager 类还有一个特性，就是可以取得键和鼠标事件的名称。虽然 KeyEvent 类已经对键事件提供这个功能，但还要对鼠标事件进行实现。这个特性在 KeyName()和 getMouseName()方法中使用，方法可以告诉用户哪个键干什么，后面要用其让用户配置键盘。

另外，InputManager 类还有一个 resetAllGameActions()方法，它复位当前“按下”的任何 GameActions 对象，适用于从主菜单切换到游戏的情形。如果不清除 GameActions，则菜单中按空格键可能使游戏者在游戏中发生跳动。

3.6　使用输入管理器

要使用输入管理器，首先要建立一个简单游戏，不过后面还要增加其他功能，并让用户配置键盘。

首先生成让英雄左右移动和跳动的游戏，还要增加暂停游戏的功能。虽然这不是游戏，但起码已经接近一步了。

3.6.1　暂停游戏

用户有时要暂停游戏，不过是喝一口饮料或休息一会儿，但都希望从游戏离开的位置

上继续开始。此外，当游戏切换到主菜单时，也要暂停游戏，否则显示菜单时游戏也将会继续运行。

暂停游戏操作实际发生什么情况呢？主要有 2 点：首先不更新游戏对象和动画，忽略输入（除了取消暂停游戏的键）。为此，需要修改游戏的循环，在不暂停游戏时才可以检查输入和更新游戏对象，实现代码如下所示。

```
if (!paused){
    checkInput();
    updateGameObjects();
}
```

其次，即使暂停游戏，屏幕也要继续绘图，甚至绘制更多的内容，如用动画显示暂停消息等。另外，用户按 P 之类的键时，还要更新暂停状态。

3.6.2 增加重力

重力为了加强游戏者跳动的效果。如果在太空中，当游戏者跳动之后就可以一直移动（直到碰上哈雷慧星或其他东西），但地球上是有重力的（难道实际中真的有超人吗？），所以游戏者跳动之后将落回到地面上。

增加重力就要了解重力的本质，它的效果就是向下加速度。速度是一定时间移动的距离（如英里/小时），而加速度则是速度改变的度量，如汽车可以在 7 秒钟内从 0 英里/小时加速到 60 英里/小时。

在第 2 章的 Sprite 类中，会根据速度来更新幽灵位置。同样，对带重力的幽灵，幽灵速度则根据加速度进行而更新，实现代码如下所示。

```
velocityY =velocityY + GRAVITY * elapsedTime;
```

地球的重力为 9.8 米/秒，但这不适用于游戏中。可以用象素和毫秒作为单位，只要使用感觉合适值即可。这里 GRAVITY 的值为 0.002，使游戏者可以跳动，但不会跳到屏幕以外。

清单 3.7 中的 Player 类增加了重力特性，Player 类只是扩展的 Sprite 类，不过是增加了重力特性和跳动功能。

清单 3.7 Player.java

```
import com.brackeen.javagamebook.graphics.*;
/**
    Player 类扩展 Sprite 类 ，增加了重力特性和跳动功能
    (STATE_NORMAL 或 STATE_JUMPING) 状态
*/
public class Player extends Sprite {

    public static final int STATE_NORMAL = 0;
    public static final int STATE_JUMPING = 1;
```

```
public static final float SPEED = .3f;
public static final float GRAVITY = .002f;

private int floorY;
private int state;

public Player(Animation anim) {
    super(anim);
    state = STATE_NORMAL;
}

/**
    取得 Player 状态(STATE_NORMAL 或 STATE_JUMPING)
*/
public int getState() {
    return state;
}

/**
    设置 Player 状态(STATE_NORMAL 或 STATE_JUMPING)
*/
public void setState(int state) {
    this.state = state;
}

/**
    设置地面位置，游戏者在此起跳和落地
*/
public void setFloorY(int floorY) {
    this.floorY = floorY;
    setY(floorY);
}

/**
    让 Player 跳动
*/
public void jump() {
    setVelocityY(-1);
    state = STATE_JUMPING;
```

```
    }

    /**
        更新游戏者位置与动画，并在游戏者落地后将其状态设置为 NORMAL
    */
    public void update(long elapsedTime) {
        // 设置垂直速度（重力效果）
        if (getState() == STATE_JUMPING) {
            setVelocityY(getVelocityY() + GRAVITY * elapsedTime);
        }

        // 移动游戏者
        super.update(elapsedTime);

        // 检查游戏者是否着地
        if (getState() == STATE_JUMPING && getY() >= floorY) {
            setVelocityY(0);
            setY(floorY);
            setState(STATE_NORMAL);
        }

    }
}
```

Player 类共有 2 个状态：NORMAL 与 JUMPING。由于 Player 的跟踪能力，所以在检查过程中保证只在正常状态下执行跳动，达到跳动时才增加重力。此外，还要告诉 Player 对象发生的地点，用 setFloorY()方法来设置。

这样所有的必要元素都已实现，那么就可以生成一个简单的游戏——InputManager 对象吧，它的代码如清单 3.8 所示。InputManagerTest 类可以让用户移动游戏者并跳动。利用几个 GameActions 对象来提供这些功能，每个 GameActions 至少贴图一个键或鼠标事件，甚至包括暂停游戏。若要移动游戏者，可以用箭头方向键。若要游戏者跳动，可以用空格键。P 键用来暂停游戏，而 Escape 键用来退出游戏。

清单 3.8　InputManagerTest.java

```
import java.awt.*;
import java.awt.event.KeyEvent;
import java.swing.ImageIcon;
import com.brackeen.javagamebook.graphics.*;
import com.brackeen.javagamebook.input.*;
```

```
import com.brackeen.javagamebook.test.GameCore;

/**
    InputManagerTest 用简单跑/跳机制测试 InputManager。要移动游戏者，可以用箭头键，要让游
    戏者跳动，用空格键
    InputManagerTest 还可以暂停游戏而不更新游戏元素
*/
public class InputManagerTest extends GameCore {

    public static void main(String[] args) {
        new InputManagerTest().run();
    }

    protected GameAction jump;
    protected GameAction exit;
    protected GameAction moveLeft;
    protected GameAction moveRight;
    protected GameAction pause;
    protected InputManager inputManager;
    private Player player;
    private Image bgImage;
    private boolean paused;

    public void init() {
        super.init();
        Window window = screen.getFullScreenWindow();
        inputManager = new InputManager(window);

        // 下面两行用干相对鼠标方式
        //inputManager.setRelativeMouseMode(true);

        //inputManager.setCursor(InputManager.INVISIBLE_CURSOR);

        createGameActions();
        createSprite();
        paused = false;

    }
```

```
/**
    测试游戏是否暂停
*/
public boolean isPaused() {
    return paused;
}

/**
    设置暂停状态
*/
public void setPaused(boolean p) {
    if (paused != p) {
        this.paused = p;
        inputManager.resetAllGameActions();
    }
}

public void update (long elapsedTime){
    //不管游戏暂停否，检查可能发生的输入{
    checkSystemInput();

    if (!isPaused()) {
        // 检查游戏输入
        checkGameInput();

        // 更新幽灵
        player.update(elapsedTime);
    }
}

/**
    不管游戏暂停否，检查可能按下的 GameActions 的输入
*/
public void checkSystemInput() {
    if (pause.isPressed()) {
        setPaused(!isPaused());
    }
    if (exit.isPressed()) {
        stop();
```

```
        }
    }

    /**
        游戏不暂停时检查可按下的 GameActions 的输入
    */
    public void checkGameInput() {
        float velocityX = 0;
        if (moveLeft.isPressed()) {
            velocityX-=Player.SPEED;
        }
        if (moveRight.isPressed()) {
            velocityX+=Player.SPEED;
        }
        player.setVelocityX(velocityX);

        if (jump.isPressed() &&
            player.getState() != Player.STATE_JUMPING)
        {
            player.jump();
        }
    }

    public void draw(Graphics2D g) {
        // 画背景
        g.drawImage(bgImage, 0, 0, null);

        // 画幽灵
        g.drawImage(player.getImage(),
            Math.round(player.getX()),
            Math.round(player.getY()),
            null);
    }

    /**
        生成 GameActions 并将其贴图到键
    */
    public void createGameActions() {
        jump = new GameAction("jump",
```

```
        GameAction.DETECT_INITAL_PRESS_ONLY);
    exit = new GameAction("exit",
        GameAction.DETECT_INITAL_PRESS_ONLY);
    moveLeft = new GameAction("moveLeft");
    moveRight = new GameAction("moveRight");
    pause = new GameAction("pause",
        GameAction.DETECT_INITAL_PRESS_ONLY);

    inputManager.mapToKey(exit, KeyEvent.VK_ESCAPE);
    inputManager.mapToKey(pause, KeyEvent.VK_P);

    // 用空格键或鼠标键跳动
    inputManager.mapToKey(jump, KeyEvent.VK_SPACE);
    inputManager.mapToMouse(jump,
        InputManager.MOUSE_BUTTON_1);

    // 用箭头键移动
    inputManager.mapToKey(moveLeft, KeyEvent.VK_LEFT);
    inputManager.mapToKey(moveRight, KeyEvent.VK_RIGHT);

    // ... 或用 A 与 D.
    inputManager.mapToKey(moveLeft, KeyEvent.VK_A);
    inputManager.mapToKey(moveRight, KeyEvent.VK_D);

    // 下列语句将游戏者移动贴图为鼠标
    //inputManager.mapToMouse(moveLeft,
    //    InputManager.MOUSE_MOVE_LEFT);
    //inputManager.mapToMouse(moveRight,
    //    InputManager.MOUSE_MOVE_RIGHT);

}

/**
    装入图像和生成 player 幽灵
*/
private void createSprite() {
    // load images
    bgImage = loadImage("../images/background.jpg");
    Image player1 = loadImage("../images/player1.png");
```

```
        Image player2 = loadImage("../images/player2.png");
        Image player3 = loadImage("../images/player3.png");

        // 生成动画
        Animation anim = new Animation();
        anim.addFrame(player1, 250);
        anim.addFrame(player2, 150);
        anim.addFrame(player1, 150);
        anim.addFrame(player2, 150);
        anim.addFrame(player3, 200);
        anim.addFrame(player2, 150);

        player = new Player(anim);
        player.setFloorY(screen.getHeight() - player.getHeight());
    }
}
```

在 InputManagerTest 类代码中，用户执行暂停游戏或停止暂停时，都要复位 GameActions 对象，从而保证暂停游戏期间按下的键不会在停止暂停后生效。

另一点值得注意的是，有几行代码已标为说明语句。这些代码打开相对鼠标方式和将鼠标运动贴图为游戏者运动。如果要用鼠标移动游戏者，则要删除这些说明语句标志。这样，就可以用鼠标键让游戏者跳动。

3.7　设计直观用户界面

下面要介绍所有游戏共有的元素：用户界面。用户界面不仅指按什么键移动游戏者，而且包括打开菜单、配置屏幕和游戏过程中的按钮。

设计直观的用户界面是游戏最重要的方面之一，没有好的用户界面，游戏就会索然无味。

用户界面设计提示

生成用户界面的基本原则是简单、明确和快速，下面是几个用户界面设计的提示：

用户界面应保持简单整洁，不要一下子显示每个选项，而可以在主屏幕中放上最常用或最有用的选项，并提供访问其他选项的简便方法；

保证每个选项或按钮易于访问。如果访问某个功能需要太多单击，则会让用户感到费解；

使用工具提示。工具提示弹出的描述在鼠标悬置到特定对象时出现，可以显示按钮功能或提供其他用户界面元素的状态。工具提示可以迅速回答“这是什么”的问题。Swing 具有内置工具提示，因此很容易实现；

对每个用户操作提供响应，可以发一个声音，也可以显示等待光标。用户单击与发生

操作之间应避免暂停。

让朋友测试你的用户界面。有的按钮你以为一目了然，别人可能找不着北。朋友测试你的用户界面时，不要说什么，只要静静地看他怎么干，并记录下来。记住，实际中，用户玩游戏时，你不可能坐在旁边提示他！

朋友测试你的用户界面后，听他们谈谈感受，可以作哪些改进，图标的含义是否明确。别对他们说，“但这样一来代码无法工作”，而只要注意倾听，把一切纳入考虑中。也许后面就找到了代码能工作的办法。

如果用户界面不好，要作推倒重来的打算。也许你为这个界面花了许多时间编码和制作图标，但如果用户界面不好也别舍不得抛弃它。为了创建更精彩的游戏，这点牺牲是值得的。

让亲戚或朋友测试你的用户界面，应该不会太难。如果不行，可以用免费比萨饼之类的小恩小惠吸引他们。

3.8 使用 Swing 组件

本书多处提到 Swing，但什么是 Swing 呢？简而言之，Swing 是一组类，用于生成用户界面元素，包括各种常见组件，如窗口、按钮、弹出菜单、下拉清单、文本输入框、复选框和光标。

这里要用 Swing 组件实现用户界面。还记得第 2 章的 ScreenManager 类中用 Swing 的 Jframe 对象作为全屏窗口。实际上，我们用 JFrame 对象而不用 Window 的 Frame 对象就是为了集成 Swing。

由于 Swing 自己绘制组件，所以可以在独立绘制循环中绘制 Swing 组件。这一点非常好，这样就可以在全屏游戏中集成 Swing 的所有功能，而不必重新从头开始，生成自己的用户界面体系结构。此外，还可以用 Swing 定制自己的外观，这样可以对用户界面生成不同外观。

Swing 的 API 很大，本书是介绍游戏编程的，而不是介绍 Swing 的，所以不准备介绍 Swing 的一切内容。如果要了解 Swing 的细节，可以阅读各种图书和网上教程。这里只要让 Swing 用进游戏中，生成一些基本用户界面元素。

3.8.1 Swing 基础

大多数的 Swing 组件很容易生成。由于多数用户界面元素都属于类，因此只要生成类的实例即可，如要生成 OK 按钮，只要生成新的 JButton 对象，实现代码如下所示。

```
JButton okButton = new JButton("OK")
```

刚生成的 JButton 对象具有所有的缺省参数和文本卷标（OK）。Swing 组件以字符 J 开头，如 JLabel,JTextField 等，它们都是从基类 JComponent 派生出的。

下面要将 okButton 对象添加到屏幕中，从而实现单击功能。组件添加到上层容器中，这里的上层容器就是 JFrame 对象，JFrame 的内容窗格会帮助 Swing 处理其所有鼠标单击，而我们不用操心。下面的实例将一个按钮添加到内容窗格中，实现代码如下所示。

```
JFrame frame = screen.getFullScreenWindow();
```

```
frame.getContentPane().add(okButton);
```

将一个按钮加进内容窗格中，但缺省情况下按钮不放在任何地方，可能只放到屏幕左上角。可以用 2 种方法解决这个问题。

第一种方法是用 setLocation()方法显式布置所有组件，实现代码如下所示。

```
okButton.setLocation(200,100)
```

显式布置很容易实现，但并不完美。假设 2 个按钮要靠在一起，则可能在你的系统中看上去不错，而放到不同系统中时，缺省 Swing 外观可能生成更大的按钮，从而造成重叠或其他难看的样子。

第二种方法是使用布局管理器（LayoutManager）。布局管理器考虑每个组件的尺寸，根据一定规则布置组件。布局管理器有多种，而且会用不同方法布置组件，如 FlowLayout 布局管理器按从左到右、从上到下方式在屏幕上布置组件。

要使用布置管理器，可以调用内容窗格的 setLayout()方法，下面是使用 FlowLayout 布局管理器的实例，实现代码如下所示。

```
frame.getContertPane().setLayout(new FlowLayout());
```

其保证加进内容窗格的所有组件都按 FlowLayout 布局管理器的规则布置。

3.8.2　在全屏方式中使用 Swing

从前面的介绍可以看出，所有组件都添加到 Jframe 对象的内容窗格中。内容窗格实际上是另一容器（即 Swing 的分层窗格）中的组件。分层窗格中包含许多不同的组件层，使一些组件可以显示为在另一些组件上方，如工具提示与弹出菜单应在内容窗格中的组件上方，因此其放在分层窗格中的内容窗格上方。

要绘制所有 Swing 组件，只要用 paintComponents()方法画出分层窗格中的内容，这是在动画循环中完成的，实现代码如下所示。

```
//画图形
draw(g);
//画 Swing 组件
JFrame frame = screen.getFullScreenWindow();
frame.getLayeredPane().paintComponents(g);
```

代码中反映出有 2 个问题。

第一，内容窗格实际绘制背景，会隐藏其下面的一切，所以要让 Swing 组件放在自己的图形上方。这个问题很容易解决，只要让内容窗格透明，实现代码如下所示。

```
if (contentPane instanceof JComponent){
    ((JComponent)contentPane).setOpaque(false;)
}
```

这个代码保证内容窗格只绘制组件而不绘制背景。

第二个问题与 Swing 绘制组件的方法有关。在正常 Swing 应用程序中，不必调用 paintComponents()方法，Swing 会自动在 AWT 事件派生线程中进行所有绘制工作。但是，这里没有关掉这个功能，所以当组件改变外观时（如按下按钮后显示按下样子时），组件会调用 ripaint()方法，以请求重画。

虽然，AWT 事件派生线程中不该发生任何绘制工作，否则会与你自己的绘制工作发生冲突，甚至造成闪烁或其他冲突。代码中可以忽略重画请求，因为如果按下按钮，则动画循环中下次绘图时会改变其外观。

要解决这个问题，就要捕获重画请求和将其忽略。所有重画请求都要发送到 RepaintManager 对像上，它管理请求并将请求转发，实际绘制组件。只要生成 NullRepaintManager 覆盖这个功能即可，如清单 3.9 所示，NullRepaintManager 只是让重画请求消失。

清单 3.9 NullRepaintManager.java

```
ackage com.brackeen.javagamebook.graphics;

import javax.swing.RepaintManager;
import javax.swing.JComponent;

/**
    NullRepaintManager 是不进行重画的 RepaintManager，用于应用程序手工进行所有绘制时 do
    any repainting
*/
public class NullRepaintManager extends RepaintManager {

    /**
        安装 NullRepaintManager.
    */
    public static void install() {
        RepaintManager repaintManager = new NullRepaintManager();
        repaintManager.setDoubleBufferingEnabled(false);
        RepaintManager.setCurrentManager(repaintManager);
    }

    public void addInvalidComponent(JComponent c) {
        //什么也不干
    }

    public void addDirtyRegion(JComponent c, int x, int y,
        int w, int h)
    {
        //什么也不干
    }
```

```
    public void markCompletelyDirty(JComponent c) {
        //什么也不干
    }

    public void paintDirtyRegions() {
        //什么也不干
    }
}
```

NullRepaintManager 程序代码非常简单，它扩展缺省的 RepaintManager 类，将键方法覆盖成什么也不干。这样，重画事件就不发到 AWT，从而不会出现闪烁组件。

这样就可以了。要安装 NullRepaintManager，只要调用 NullRepaintManager.install()方法即可。

注意：记住 Swing 组件不是线程安全的。显示 Swing 组件时，不能在 AWT 事件派生线程修改组件状态的同时修改组件状态。如果要在 Swing 组件显示之后修改组件状态，则可以在 AWT 事件派生线程中进行，实现代码如下所示。

```
EventQueue.invokeAndWait(new Runnable(){
public void run(){
doSomething();
}
})
```

可以看出，在 AWT 事件派生线程中调用 Rummable 中的代码并等待其完成，也可以用 invokeLater()方法而不等待其就完成。

3.9　创建简单菜单

现在可以扩展 InputManagerTest 对象中的游戏，在简单的用户界面中加上暂停、配置和退出按钮。虽然目前配置按钮什么也不干，但还是要先把它放到屏幕上。

首先，确定单击按钮时会发生什么情形呢？当 Swing 发现单击事件，并检查按钮是否有任何 ActionListeners。如果有，则在 AWT 事件派生线程中告诉这些 ActionListeners 已经单击按钮。

与 KeyListener 和 MouseListener 接口一样，ActionListeners 也是任何对象都可以实现的接口，它有一个 actionPerformed()方法，并取 ActionEvent 作为参数。可以用 ActionEvent 的 getSource()方法检查事件是哪个组件生成的，实现代码如下所示。

```
public void actionPerformed(ActionEvent e){
object src = e.getSource();
if (src == okButton){
//进行操作
}
}
```

最后，用户界面中可以对按钮进行一些处理，使其更有用：

- 增加工具提示。只要调用 setToolTip("Hello World")之类方法，Swing 会完成其余工作；
- 使用图标。按钮不仅可以使用文本，还可以使用图标。缺省、滚过和按下状态可以用不同图标；
- 隐藏缺省外观。要让图标单独显示，因此关掉按钮边框，并用 setContentAreaFilled(false)方法保证不绘制按钮背景；
- 改变光标。可以用 setCursor()方法使光标经过按钮时显示手形；
- 关掉键焦点。如果按钮可以聚集，则单击时可能从游戏中"窃取"键盘焦点。要解决这个问题，只要调用 setFocusable(false)方法。这个方法的惟一缺点是只有鼠标能激活按钮。

这些改变都在 MenuTest 类的 createButton()方法中完成。清单 3.10 所示的 MenuTest 类扩展 InputManagerTest 类，并增加几个按钮，图 3-1 显示了 MenuTest 的屏幕图形。

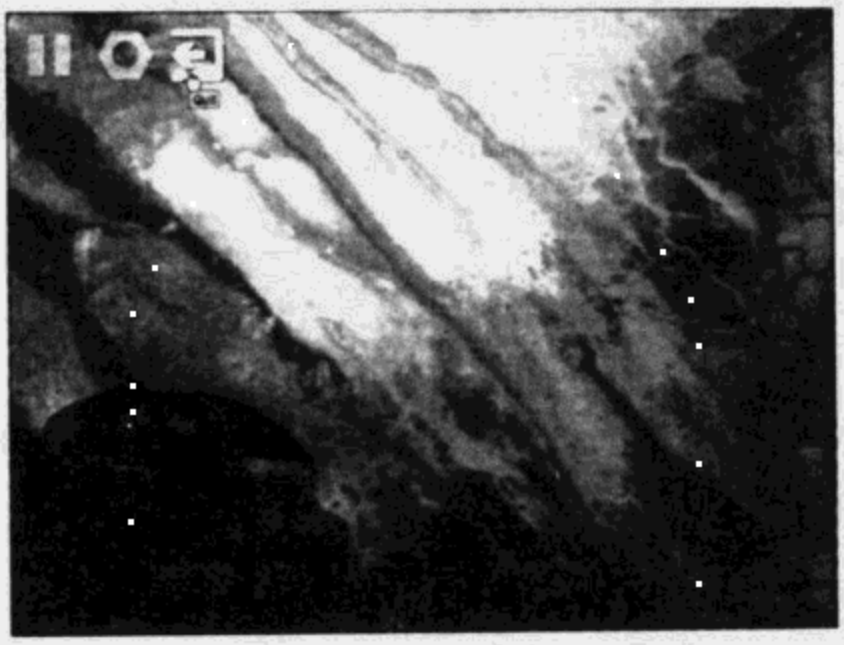


图 3-1　MenuTest 类在游戏上方生成简单用户界面

清单 3.10　MenuTest.java

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

import com.brackeen.javagamebook.graphics.*;
import com.brackeen.javagamebook.input.GameAction;

/**
    扩展 InputManagerTest 中的游戏，在简单的用户界面中加上暂停、配置和退出按钮
*/
public class MenuTest extends InputManagerTest
    implements ActionListener
```

```
{

    public static void main(String[] args) {
        new MenuTest().run();
    }

    protected GameAction configAction;

    private JButton playButton;
    private JButton configButton;
    private JButton quitButton;
    private JButton pauseButton;
    private JPanel playButtonSpace;

    public void init() {
        super.init();
        //让 Swing 组件不画自己
        NullRepaintManager.install();

        //生成配置的 GameAction
        configAction = new GameAction("config");

        // 生成按钮
        quitButton = createButton("quit", "Quit");
        playButton = createButton("play", "Continue");
        pauseButton = createButton("pause", "Pause");
        configButton = createButton("config", "Change Settings");

        // 生成播放/暂停按钮的空间
        playButtonSpace = new JPanel();
        playButtonSpace.setOpaque(false);

        playButtonSpace.add(pauseButton);

        JFrame frame = super.screen.getFullScreenWindow();
        Container contentPane = frame.getContentPane();

        //保证内容窗格透明
        if (contentPane instanceof JComponent) {
```

```
            ((JComponent)contentPane).setOpaque(false);
        }

        // 将组件加进屏幕的内容窗格
        contentPane.setLayout(new FlowLayout(FlowLayout.LEFT));
        contentPane.add(playButtonSpace);
        contentPane.add(configButton);
        contentPane.add(quitButton);

        //显式布置组件（有些系统中需要）
        frame.validate();
    }

    /**
        扩展 InputManagerTest 功能，绘制所有 Swing 组件
    */
    public void draw(Graphics2D g) {
        super.draw(g);
        JFrame frame = super.screen.getFullScreenWindow();

        // 分层窗格包含弹出件（工具提示、弹出菜单）和内容窗格等项目
        frame.getLayeredPane().paintComponents(g);
    }

    /**
        在暂停状态改变时改变播放/暂停按钮
    */
    public void setPaused(boolean p) {
        super.setPaused(p);
        playButtonSpace.removeAll();
        if (isPaused()) {
            playButtonSpace.add(playButton);
        }
        else {
            playButtonSpace.add(pauseButton);
        }

    }
```

```
/**
    按下按钮时由 AWT 事件派生线程调用
*/
public void actionPerformed(ActionEvent e) {
    Object src = e.getSource();
    if (src == quitButton) {
        // 触发 "exit" gameAction
        super.exit.tap();
    }
    else if (src == configButton) {
        // 目前什么也不干
        configAction.tap();
    }
    else if (src == playButton || src == pauseButton) {
        // 触发"pause" gameAction
        super.pause.tap();
    }
}

/**
    生成 Swing JButton。按钮所用图像位于"../images/menu/" + name + ".png"，修改成生成缺
    省外观（半透明）和压下样子（向下向右移动）
    按钮不用 Swing 外观，而用图像
*/
public JButton createButton(String name, String toolTip) {

    // 装入图像
    String imagePath = "../images/menu/" + name + ".png";
    ImageIcon iconRollover = new ImageIcon(imagePath);
    int w = iconRollover.getIconWidth();
    int h = iconRollover.getIconHeight();

    // 取得按钮的光标
    Cursor cursor =
        Cursor.getPredefinedCursor(Cursor.HAND_CURSOR);

    // 生成半透明缺省图像
    Image image = screen.createCompatibleImage(w, h,
        Transparency.TRANSLUCENT);
```

```
        Graphics2D g = (Graphics2D)image.getGraphics();
        Composite alpha = AlphaComposite.getInstance(
            AlphaComposite.SRC_OVER, .5f);
        g.setComposite(alpha);
        g.drawImage(iconRollover.getImage(), 0, 0, null);
        g.dispose();
        ImageIcon iconDefault = new ImageIcon(image);

        //生成压下图像
        image = screen.createCompatibleImage(w, h,
            Transparency.TRANSLUCENT);
        g = (Graphics2D)image.getGraphics();
        g.drawImage(iconRollover.getImage(), 2, 2, null);
        g.dispose();
        ImageIcon iconPressed = new ImageIcon(image);

        // 生成按钮
        JButton button = new JButton();
        button.addActionListener(this);
        button.setIgnoreRepaint(true);
        button.setFocusable(false);
        button.setToolTipText(toolTip);
        button.setBorder(null);
        button.setContentAreaFilled(false);
        button.setCursor(cursor);
        button.setIcon(iconDefault);
        button.setRolloverIcon(iconRollover);
        button.setPressedIcon(iconPressed);
        return button;
    }
}
```

在 MenuTest 类代码中，每个按钮只有一个 PNG 图像，其他图像会在程序启动时通过代码生成。

装入的图像显示为滚过图像，在鼠标位于按钮上方成显示。缺省图像稍淡一些，是 AlphaComposite 为 0.5 时滚过图像的拷贝。AlphaComposite 类只是让 Graphics2D 类将源图像与目标图像混合，生成半透明效果。最后，将滚过图像偏移几个象素，生成按下图像。

MenuTest 类中还有一点要说明的是界面中如何暂停。当游戏暂停时，暂停按钮就会变成播放按钮。要实现这个功能，就要把暂停按钮放在 JPanel 中，Jpanel 是其他组件的容器。用户暂停或停止暂停时，改变 Jpanel 内容，显示正确的按钮。

当然，MenuTest 中的配置按钮目前什么也干不了。

3.10　让游戏者配置键盘

最后，要增加期待已久的功能：让游戏者配置键盘，如图 3-2 所示。

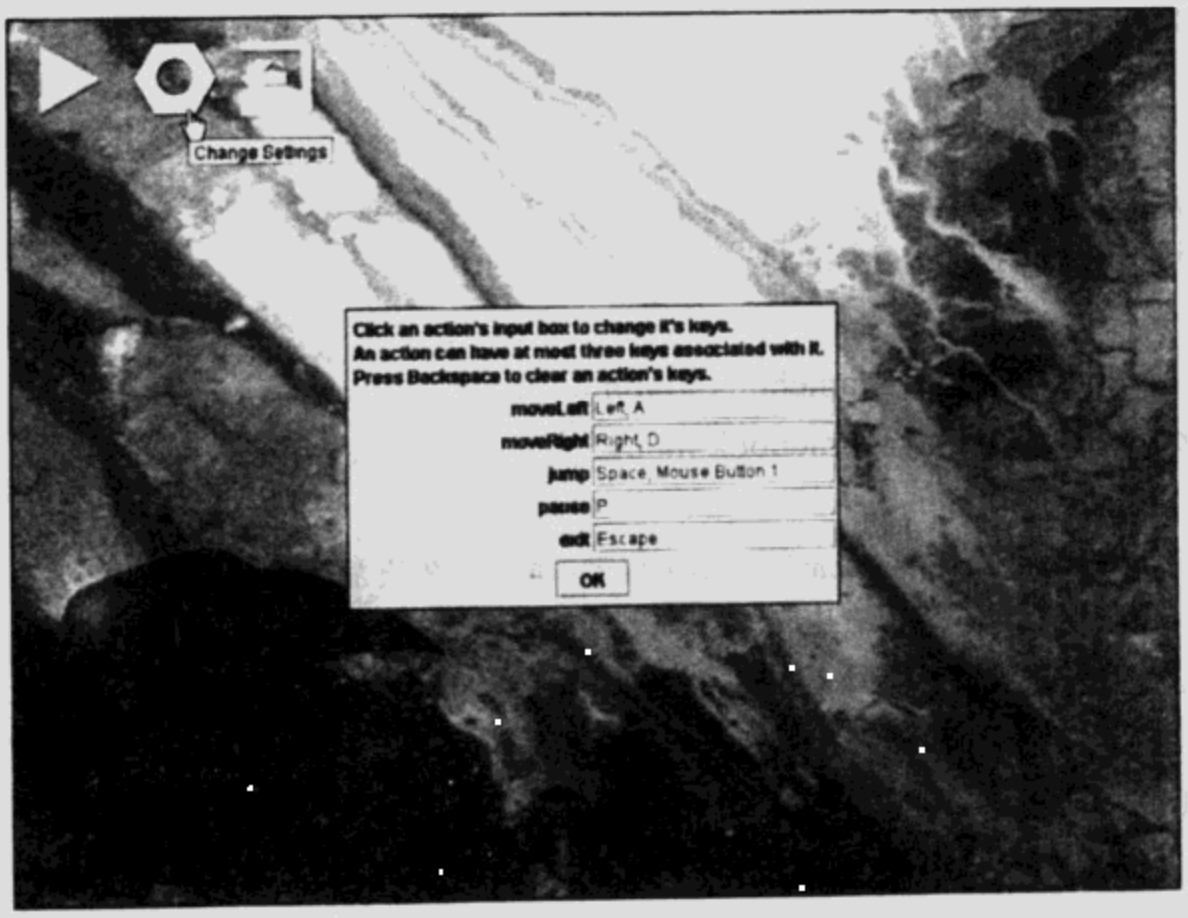


图 3-2　Configure 对话框使用户可以定制游戏控件

这个特性使用户可以将任何的键盘和鼠标事件贴图到任何游戏动作上。用户只要选择游戏动作并按一个键或单击鼠标，即可把这个键指定给游戏动作。

键盘配置特性可以分解为 2 个部分。

（1）首先要实际生成 Configure 对话框。

（2）然后要生成一个特殊组件，让用户输入键或鼠标单击。

生成 Configure 对话框并不难，Configure 对话框中列出游戏的所有游戏动作，并包含指令和 OK 按钮。Configure 对话框本身是个 Jpanel 对象，对话框中的一切都可以用一系列组件、面板与布置生成。

生成特殊输入组件比较困难，这个组件显示当前哪个键贴图游戏动作，并让用户通过按键或单击鼠标改变设置。输入完成后，还要保证输入组件将键盘焦点返回游戏主窗口。

生成的组件称为 InputComponent，是清单 3.11 中 KeyConfigTest 类的内类。

InputComponent 是 JTextField 的子类。尽管用户可以通过正常 JTextField 组件输入任意普通文本，但这里只让用户输入键或鼠标单击，因此要覆盖 JTextField 的输入方法。通常，生成 KeyListener 与 MouseListener 接口以监听输入事件，但这时要用不同方法取键。当然，取得输入事件的方法有多种。

每个 Swing 组件都是 Component 类的一个实例。Component 类具有 processKeyEvent() 和 processMouseEvent()之类的方法。这些方法与 KeyListener 和 MouseListener 中的方法相

似。只在覆盖这些方法，用 enableEvents()方法启用输入事件即可。

结果与使用监听器差不多，这里只是介绍另一种做法而已。

清单 3.11 KeyConfigTest.java

```
import java.awt.event.*;
import java.awt.*;
import java.util.List;
import java.util.ArrayList;
import javax.swing.*;
import javax.swing.border.Border;

import com.brackeen.javagamebook.graphics.*;
import com.brackeen.javagamebook.input.*;
/**
    KeyConfigTest 类扩展 MenuTest 演示程序，增加配置键盘键的对话柜
*/
public class KeyConfigTest extends MenuTest {

    public static void main(String[] args) {
        new KeyConfigTest().run();
    }

    private static final String INSTRUCTIONS =
        "<html>Click an action's input box to change it's keys." +
        "<br>An action can have at most three keys associated " +
        "with it.<br>Press Backspace to clear an action's keys.";

    private JPanel dialog;
    private JButton okButton;
    private List inputs;

    public void init() {

        super.init();

        inputs = new ArrayList();

        //生成 GameActions 与贴图键清单
        JPanel configPanel = new JPanel(new GridLayout(5,2,2,2));
```

```
addActionConfig(configPanel, moveLeft);
addActionConfig(configPanel, moveRight);
addActionConfig(configPanel, jump);
addActionConfig(configPanel, pause);
addActionConfig(configPanel, exit);

//生成包含 OK 按钮的面板
JPanel bottomPanel = new JPanel(new FlowLayout());
okButton = new JButton("OK");
okButton.setFocusable(false);
okButton.addActionListener(this);
bottomPanel.add(okButton);

// 生成包含指令的面板
JPanel topPanel = new JPanel(new FlowLayout());
topPanel.add(new JLabel(INSTRUCTIONS));

// 生成指令对话柜边框
Border border =
    BorderFactory.createLineBorder(Color.black);

// 生成配置对话框
dialog = new JPanel(new BorderLayout());
dialog.add(topPanel, BorderLayout.NORTH);
dialog.add(configPanel, BorderLayout.CENTER);
dialog.add(bottomPanel, BorderLayout.SOUTH);
dialog.setBorder(border);
dialog.setVisible(false);
dialog.setSize(dialog.getPreferredSize());

//居中对话框
dialog.setLocation(
    (screen.getWidth() - dialog.getWidth()) / 2,
    (screen.getHeight() - dialog.getHeight()) / 2);

// 将对话框加进屏幕分层窗格的横态对话框层
screen.getFullScreenWindow().getLayeredPane().add(dialog,
    JLayeredPane.MODAL_LAYER);
}
```

```
/**
    增加包含 GameAction 名与 InputComponent 的标题，用于改变贴图键
*/
private void addActionConfig(JPanel configPanel,
    GameAction action)
{
    JLabel label = new JLabel(action.getName(), JLabel.RIGHT);
    InputComponent input = new InputComponent(action);
    configPanel.add(label);
    configPanel.add(input);
    inputs.add(input);
}

public void actionPerformed(ActionEvent e) {
    super.actionPerformed(e);
    if (e.getSource() == okButton) {
        // 隐藏配置对话框
        configAction.tap();
    }
}

public void checkSystemInput() {
    super.checkSystemInput();
    if (configAction.isPressed()) {
        // 隐藏或显示配置对话框
        boolean show = !dialog.isVisible();
        dialog.setVisible(show);
        setPaused(show);
    }
}

/**
    复位每个 InputComponent 中显示的文本，是贴图键名
*/
private void resetInputs() {
    for (int i=0; i<inputs.size(); i++) {
        ((InputComponent)inputs.get(i)).setText();
    }
}
```

```
/**
    InputComponent 类显示贴图特定操作的键，使用户可以改变贴图键。用户单击选择
    InputComponent，然后可以按任何键或鼠标键（包括鼠标轮），改变贴图值
*/
class InputComponent extends JTextField   {

    private GameAction action;

    /**
        对指定 GameAction 生成新 InputComponent
    */
    public InputComponent(GameAction action) {
        this.action = action;
        setText();
        enableEvents(KeyEvent.KEY_EVENT_MASK |
            MouseEvent.MOUSE_EVENT_MASK |
            MouseEvent.MOUSE_MOTION_EVENT_MASK |
            MouseEvent.MOUSE_WHEEL_EVENT_MASK);
    }

    /**
        将这个 InputComponent 的显示文本设置为贴图键名
    */
    private void setText() {
        String text = "";
        List list = inputManager.getMaps(action);
        if (list.size() > 0) {
            for (int i=0; i<list.size(); i++) {
                text+=(String)list.get(i) + ", ";
            }
            //删除最后的逗号
            text = text.substring(0, text.length() - 2);
        }

        //保证不死锁
        synchronized (getTreeLock()) {
            setText(text);
        }
    }
```

```
/**
    将这个 InputComponent 的 GameAction 贴图指定键或鼠标操作
*/
private void mapGameAction(int code, boolean isMouseMap) {
    if (inputManager.getMaps(action).size() >= 3) {
        inputManager.clearMap(action);
    }
    if (isMouseMap) {
        inputManager.mapToMouse(action, code);
    }
    else {
        inputManager.mapToKey(action, code);
    }
    resetInputs();
    screen.getFullScreenWindow().requestFocus();
}

// 另一种截获键事件的方法
protected void processKeyEvent(KeyEvent e) {
    if (e.getID() == e.KEY_PRESSED) {
        //按空格键时，清除贴图
        if (e.getKeyCode() == KeyEvent.VK_BACK_SPACE &&
            inputManager.getMaps(action).size() > 0)
        {
            inputManager.clearMap(action);
            setText("");
            screen.getFullScreenWindow().requestFocus();
        }
        else {
            mapGameAction(e.getKeyCode(), false);
        }
    }
    e.consume();
}

//另一种截获鼠标事件的方法
protected void processMouseEvent(MouseEvent e) {
    if (e.getID() == e.MOUSE_PRESSED) {
        if (hasFocus()) {
```

```
                int code = InputManager.getMouseButtonCode(e);
                mapGameAction(code, true);
            }
            else {
                requestFocus();
            }
        }
        e.consume();
    }

    //另一种截获鼠标事件的方法
    protected void processMouseMotionEvent(MouseEvent e) {
        e.consume();
    }

    //另一种截获鼠标事件的方法
    protected void processMouseWheelEvent(MouseWheelEvent e) {
        if (hasFocus()) {
            int code = InputManager.MOUSE_WHEEL_DOWN;
            if (e.getWheelRotation() < 0) {
                code = InputManager.MOUSE_WHEEL_UP;
            }
            mapGameAction(code, true);
        }
        e.consume();
    }
}
}
}
```

KeyConfigTest 类扩展 MenuTest 类。Init()方法生成配置对话框并将其加进分层窗格。checkSystemInput()方法覆盖成检查配置按钮是否按下（螺钉图标），如果是，则显示配置对话框。

InputComponent 有几个方法可以取得输入，并可用 InputManager 类的方法将键和鼠标事件贴图到游戏动作。另外，InputComponent 类将游戏动作可以贴图的键数限制为 3 个。这是人为的限制，实际中 InputManager 没有限制游戏动作可以贴图的键数。

另外，InputComponent 把退格键作为特殊键。按下退格键时，清除游戏动作的所有贴图键。如果游戏动作没有贴图键，则贴图退格键。

还有一点没有提到的是， JLabels 可以包含 HTML（Web 语言）。这里可以用 HTML 把包含指令的 Jlabels 分解为多行，这是用 HTML 的分行标志
实现的。

3.11 小结

本章介绍了如何取得键盘与鼠标输入，并创建输入管理器来处理这些输入，还介绍如何实现鼠标观看和隐藏光标，这是设计第一人称视角游戏时非常重要的 2 个小特性。另外，还介绍了如何生成直观用户界面，如何用 Swing 生成这些界面，包括全屏方式。

第 4 章　声效与音乐

关键主题

- 声音基础
- Java Sound API
- 播放声音
- 创建实时声音过滤体系结构
- 创建实时回响过滤
- 仿真三维声音
- 创建声音管理器
- 播放音乐
- 小结

除非你从来没有想到要把音箱装到计算机上，否则你一定注意到游戏中的声音效果和背景音乐。可以说声音是游戏的重要部分，甚至是不可缺少的。

本章将介绍播放声音的基础，然后创建实时声音过滤系统（如回响），最后还要创建声音管理器，来实现常规的操作工作。此外还要介绍播放音乐（如 CD 声频、MP3、Ogg Vorbis 和 MIDI）和根据游戏状态动态来改变音乐技术。

4.1　声音基础

声音是物质在媒介中振动产生的，游戏中的声音以空气为媒介，振动则来自计算机的音箱，当耳膜听到振动并将信息传递给大脑，大脑将其解释为声音。空气中的振动会产生压力波动，高速波动导致高频声波，使声音听起来更尖锐。每次波动的压力量称为振幅（amplitude），振幅越高，声音听起来就越大。简而言之，声波就是随时间改变的振幅，如图 4-1 所示。

图 4-1　声波就是随时间改变的振幅

数字化声音（如光盘声频和音频格式的文件）是声音振幅的一系列离散样本。每秒钟存储的样本量称为采样率，如光盘声频的采样率为 44100Hz。当然，采样率越高，声频表示越精确，而采样率越低，音质越差，但文件相对长度越小。声音的离散样本通常采用 16 位表示，即共有 65536 种振幅值。

许多声音格式都支持多声道技术，如光盘声频有 2 个声道，一个用于左音箱，一个用于右音箱。

4.2 Java Sound API

要在 Java 游戏中播放样本声音，就要使用 javax.sound.sampled 包中的 Java Sound API。

Java Sound 样本通常采用 8 位或 16 位振幅值，采样率为 8000Hz 到 48000Hz，它支持单声道和立体声声道。使用何种声音格式主要由游戏所需的结果决定，本章实例将采用 16 位单声道 44100Hz 的格式。

如果感到自我良好，也可以在代码中生成所需的声音样本，但通常只是从文件中取得它们。Java Sound 支持读取 3 种样本声音的格式：AIFF，AU 和 WAV。这些声音格式都很灵活，而且使用差别不大，本章实例中将使用 WAV 格式。

下面推荐可以生成、录制、编辑声音的声音程序，它们包括：

- Pro Tools FREE（www.digidesign.com/ptfree）；
- Cool Edit（www.syntrillium.com/cooledit）；
- GoldWave（www.goldwave.com）；
- Audacity（audacity.sourceforge.net）。

在第 17 章“生成游戏图形与声音”将介绍如何生成声音文件。

4.2.1 打开声音文件

使用 Java Sound 中的 AudioSystem 类读取声音文件。AudioSystem 有几个静态函数，大多数都用不上。不过它提供单独的 getAudioInputStream()方法可以从系统和其他源中打开音频文件，如 Internet。getAudioInputStream()方法将返回 AudioInputStream 对象。

利用 AudioInputStream 对象，可以读取声音样本，而不必理会文件的标题等多余的信息。另外，还可以用 getFormat()方法查询声音格式，实现代码如下所示。

```
File file = new File(“sound.wav”);
AudioInputStream Stream = AudioSystem. getAudioInputStream(file);
AudioFormat format = stream. getFormat();
```

AudioFormat 类提供取得声音格式中信息的方法，如采样率与声道数。另外，它还可以取得帧长度，即每声道中每个样本所需的字节数。对 16 位立体声而言，它的帧长度为 4，即每个样本 2 字节（左右 2 个声道）。用这些信息可以求出内存中存储声音所需的字节数，如声音格式是 16 位样本、立体声、44100Hz，则 3 秒的声音容量是 44100×3×4 字节，大约 517Kbyte，不过用单声道代替立体声可以将长度减少一半。

4.2.2 使用 Line

在取得声音样本和格式后，那么将如何处理呢？答案是通过 Line 将它们进行输入。

Line 是接收和发送来自声音系统的音频的接口，可以用 Line 将声音样本发送到声音系统中播放，或从麦克风之类声音系统接收声音。

Line 具有若干子接口，本章的实例就使用 Line 的子接口 SourceDataLine，它可以将声

频数据写入声音系统中。

Line 用 AudioSystem 类的 getLine（）方法生成。只要向方法传入 Line.Info 对象，并指定要生成 Line 的类型即可。Line.Info 的 DataLine.Info 子类可以生成 Line，其中包含 Line 的声频格式信息。

除 SourceDataLine 子接口之外，还要介绍另一个 Line 的子接口 Clip。Clip 进行大量工作，如将样本从 AudioInputStream 装入内存并将其自动装入声频系统中。下面就要将 AudioInputStream 扩展成用 Clip 播放，实现代码如下所示。

```
//指定要生成的 Line 类型
DataLine.Info info = new DataLine.Info(Clip.class,format);
//生成 Line
Clip clip = (Clip) AudioSystem。GetLine（info）;
//从流中装入样本
Clip.open(stream);
//开始播放声音片断
Clip.start();
```

Clip 的使用很简单，与 Java SDK 1.0 中引入的 AudioClips 相似。但 Clip 有几个缺点，如 Java Sound 限制同时打开的 Line 个数，通常最多为了 2 个。由于 Clip 是 Line，因此即使还没有播放声音，能打开的声音个数也受到限制。

另外，尽管可以同时播放多个 Clip，但每个 Clip 一次只能播放一个声音，如要同时播放 2 至 3 个爆炸声，则需要多个 Clip 对象。

由于 Clip 的缺点，下节将要建立一个解决方案，其中要装入任意多个声音，并同时播放每个声音的多个拷贝。

4.3　播放声音

首先要生成播放声音的 SimpleSoundPlayer 类，它将样本从 AudioInputStream 装入到字节数组中，并从 Line 中将数据进行复制，然后播放任何 InputStream 中的声音。

如清单 4.1 所示，在 SimpleSoundPlayer 类代码中，装入的样本用 ByteArrayInpu tStream 转换成 InputStream，这样就可以从内存而不是从磁盘中读取样本了。也可以直接将字节数组写入 Line 中，但这样就要从 InputStream 读取，增加一些更高级的功能。

由于使用包装成字节数组中的 ByteArrayInputStream，因此可以对同一声音生成多个 ByteArrayInputStream 对象，从而可以同时播放每个声音的多个拷贝。

清单 4.1　SimpleSoundPlayer.java

```
import java.io.*;
import javax.sound.sampled.*;

/**
```

```
    SimpleSoundPlayer 包装的声音可以从文件系统打开和播放
*/
public class SimpleSoundPlayer   {

    public static void main(String[] args) {
        // 装入声音
        SimpleSoundPlayer sound =
                newSimpleSoundPlayer("../sounds/voice.wav");

        // 生成要播放的流
        InputStream stream =
            new ByteArrayInputStream(sound.getSamples());

        // 播放声音
        sound.play(stream);

        // 退出
        System.exit(0);
    }

    private AudioFormat format;
    private byte[] samples;

    /**
        从文件打开声音
    */
    public SimpleSoundPlayer(String filename) {
        try {
            //打开声频输入流
            AudioInputStream stream =
                AudioSystem.getAudioInputStream(
                new File(filename));

            format = stream.getFormat();

            // 取得声频样本
            samples = getSamples(stream);
        }
```

```
        catch (UnsupportedAudioFileException ex) {
            ex.printStackTrace();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
    }

    /**
        取得声音样本的字节数组
    */
    public byte[] getSamples() {
        return samples;
    }

    /**
        从 AudioInputStream 取得样本的字节数组
        of bytes.
    */
    private byte[] getSamples(AudioInputStream audioStream) {
        //取得要读取的字节数
        int length = (int)(audioStream.getFrameLength() *
            format.getFrameSize());

        //读取整个流
        byte[] samples = new byte[length];
        DataInputStream is = new DataInputStream(audioStream);
        try {
            is.readFully(samples);
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }

        // 返回样本
        return samples;
    }

    /**
```

```
        播放流。这个方法受阻，要播放完声音后才返回
    */
    public void play(InputStream source) {

        // 用短的 100ms (1/10 秒) 缓冲区实时改变声音流
        int bufferSize = format.getFrameSize() *
            Math.round(format.getSampleRate() / 10);
        byte[] buffer = new byte[bufferSize];

        //生成要播放的 Line
        SourceDataLine line;
        try {
            DataLine.Info info =
                new DataLine.Info(SourceDataLine.class, format);
            line = (SourceDataLine)AudioSystem.getLine(info);
            line.open(format, bufferSize);
        }
        catch (LineUnavailableException ex) {
            ex.printStackTrace();
            return;
        }

        // 启动 line
        line.start();

        //将数据复制到 line
        try {
            int numBytesRead = 0;
            while (numBytesRead != -1) {
                numBytesRead =
                    source.read(buffer, 0, buffer.length);
                if (numBytesRead != -1) {
                    line.write(buffer, 0, numBytesRead);
                }
            }
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
```

```
            // 等待播放所有数据
            line.drain();
            //关闭 line
            line.close();

        }
    }
```

在 SimpleSoundPlayer 类代码中，用 getSamples(AudioInputStream)方法读取 AudioInputStream 并将数据存放在样本字节数组中。用 Play()方法将数据从 InputStream 读取到缓冲区并将缓冲区写入 SourceDataLine，它是用来播放声音的。另外，SimpleSoundPlayer 中的 voice.wav()方法播放 voice.wav 声音，测试这个类。

由于 Java Sound 中的缺陷，Java 程序不能自己退出。通常，Java 虚拟机在只运行监控程序线程时退出，但使用 Java Sound 时，总是在后台运行一个非监控程序线程，因此，要退出使用 Java Sound 的 Java 程序，就需要调用 System.exit(0)方法。

现在可以播放声音了，但如果要在循环中重复播放声音呢？如苍蝇声等后台环境声音效。

要在循环中重复播放声音，根本不用改变 SimpleSoundPlayer。清单 4.2 的 LoopingByteInputStream 用法与 ByteArrayInputStream 相似，惟一的差别在于 close()在循环中不断读取字节数组，直到调用其 close()方法为止。

清单 4.2　LoopingByteInputStream.java

```
package com.brackeen.javagamebook.util;

import java.io.ByteArrayInputStream;
import java.io.IOException;

/**
    LoopingByteInputStream 就是 ByteArrayInputStream 不断循环，直到调用其 close() 为止
    扩展这个类的思路之一
    增加只循环一定次数的选项
*/
public class LoopingByteInputStream extends ByteArrayInputStream {

    private boolean closed;
    /**
        用指定字节数组生成新 LoopingByteInputStream。不复制数组
    */
    public LoopingByteInputStream(byte[] buffer) {
        super(buffer);
```

```
        closed = false;
    }

    /**
        从数组中读取 <code>length</code> 字节。如果到达数组末尾，则从数组头重新开始读取。
        如果数组关闭，则返回-1
    */
    public int read(byte[] buffer, int offset, int length) {
        if (closed) {
            return -1;
        }
        int totalBytesRead = 0;

        while (totalBytesRead < length) {
            int numBytesRead = super.read(buffer,
                offset + totalBytesRead,
                length - totalBytesRead);

            if (numBytesRead > 0) {
                totalBytesRead += numBytesRead;
            }
            else {
                reset();
            }
        }
        return totalBytesRead;
    }
    /**
        关闭流，此后调用 read() 方法时返回-1
    */
    public void close() throws IOException {
        super.close();
        closed = true;
    }
}
```

LoopingByteInputStream 类没有什么特别之处，它只扩展 ByteArrayInputStream，遇到流末尾时，用 rest()方法复位，从数组头重新开始读取。

现在即可方便地播放和循环在字节数组中的声音了。另外，由于可以访问所有声音样本，因此可以操纵样本，生成不同效果（过滤）。

4.4 创建实时声音过滤体系结构

声音过滤器（Sound filters）是简单的声频处理器，可以修改现有声音样本，通常进行实时修改。声音过滤器也称为数字信号处理器，在声频生产中经常使用，如增加其他变量或增加回声等。

声音过滤可以使游戏更具动感。游戏中可能有一个关卡，游戏者要在峡谷中漫步。这时，可以在播放声音时增加回声。如果火箭从游戏者身旁飞过，则可以使声音从左音箱转到右音箱。

本节要创建实时声音过滤体系结构和生成 2 个过滤器：一个回响过滤器，一个模拟三维声音过滤器。

首先要创建实时声音过滤体系结构，很简单，只要用一个简单的类修改缓冲区中的声音样本即可。清单 4.3 的 SoundFilter 类可以完成这个工作，它是一个抽象类，可以扩展生成声音过滤器，它有 3 个重要的方法：

- filter(byte[] buffer,int offset,int length)过滤样本数组；
- getRemainingSize()取得声音完成后，过滤器可以播放的其余长度（字节），如回响可以播放比原声更长时间；
- reset()复位声音过滤器，它可以对不同声音再次使用。

由于 SoundFilter 是个抽象类，因此并不实际过滤任何内容，过滤工作要在子类中完成。

清单 4.3 SoundFilter.java

```
package com.brackeen.javagamebook.sound;

/**
    过滤声音样本的抽象类。由于 SoundFilters 可能用内部样本缓存，因此播放的每个声音要生成
    一个新的 SoundFilter 对象。但完成之后可以调用 reset()方法复用 SoundFilters
    假设所有样本都是 16 位带符号小端格式
    @见 FilteredSoundStream
*/
public abstract class SoundFilter{

    /**
        复位这个 SoundFilter.缺省什么也不干
    */
    public void reset() {
        // 什么也不干
    }

    /**
        取得声音完成后这个过滤器可以播放的其余长度（字节）。例如，回响可以播放比原声更
```

```
    长时间。这个方法缺省返回
*/
public int getRemainingSize() {
    return 0;
}

/**
    过滤样本数组。所有样本都是 16 位带符号小端格式
*/
public void filter(byte[] samples) {
    filter(samples, 0, samples.length);
}

/**
    过滤样本数组。所有样本都是 16 位带符号小端格式。这个方法由子类实现，注意偏移量
    与长度是字节数，而不是样本数
*/
public abstract void filter(
    byte[] samples, int offset, int length);

/**
    方便方法，从字节数组取得 16 位样本。所有样本都是 16 位带符号小端格式
*/
public static short getSample(byte[] buffer, int position) {
    return (short)(
        ((buffer[position+1] & 0xff) << 8) |
        (buffer[position] & 0xff));
}

/**
    方便方法，在字节数组中设置 16 位样本。所有样本都是 16 位带符号小端格式
*/
public static void setSample(byte[] buffer, int position,
    short sample)
{
    buffer[position] = (byte)(sample & 0xff);
    buffer[position+1] = (byte)((sample >> 8) & 0xff);
}
```

}

由于 SoundFilter 对象可能包含状态数据，所以播声音对象都要使用不同的 SoundFilter 对象。

为了简单起见，SoundFilter 类假设所有样本都是以 16 位带符号小端格式。小端（little-endian）指数据的字节顺序，在 16 位样本中先保存最低有效字节，而大端（big-endian）则先保存最高有效字节。带符号数据表示样本有符号位，因此范围是-32768 到 32767 而不是 0 到 65536。WAV 格式缺省就是采用带符号小端样本。

Java Sound 希望数据用字节，因此要读取样本的静态方法：getSample()与 setSample()。

现在需要一个对声音采用 SoundFilter 的简便方法。可以对样本数组直接进行过滤，但这只适用于永久过滤，因为它会修改原样本。有些过滤（如三维声音过滤器）依赖于实时改变的特定游戏元素，因此要实时采用样本过滤，而不动原声。

由于 SimpleSoundPlayer 类从 InputStream 播放声音，因此可以生成新 InputStream 子类，采用 SoundFilter。清单 4.4 的 FilteredSoundStream 类就是这么干的。除了 SoundFilter 之外，它还取 InputStream 源（和前面一样，使用 ByteArrayInputStream 或 LoopingByteInputStream）。这样，从输入源读取的每个字节通过 SoundFilter 传递到声音播放器，得到实时过滤，不修改原声。

清单 4.4　FilteredSoundStream.java

```
package com.brackeen.javagamebook.sound;

import java.io.FilterInputStream;
import java.io.InputStream;
import java.io.IOException;

/**
    FilteredSoundStream 类是个 FilterInputStream ，对基础输入流采用 SoundFilter
    @见 SoundFilter
*/
public class FilteredSoundStream extends FilterInputStream {

    private static final int REMAINING_SIZE_UNKNOWN = -1;

    private SoundFilter soundFilter;
    private int remainingSize;

    /**
        生成指定 InputStream 与 SoundFilter 的新 FilteredSoundStream 对象
        specified.
    */
```

```
public FilteredSoundStream(InputStream in,
    SoundFilter soundFilter)
{
    super(in);
    this.soundFilter = soundFilter;
    remainingSize = REMAINING_SIZE_UNKNOWN;
}

/**
    覆盖 FilterInputStream 方法，在读取字节时采用这个过滤
*/
public int read(byte[] samples, int offset, int length)
    throws IOException
{
    //读取和过滤流中的声音样本
    int bytesRead = super.read(samples, offset, length);
    if (bytesRead > 0) {
        soundFilter.filter(samples, offset, bytesRead);
        return bytesRead;
    }

    // 如果声音流中没有其余字节，则检查过滤器中还有没有其余字节（回响）。
    if (remainingSize == REMAINING_SIZE_UNKNOWN) {
        remainingSize = soundFilter.getRemainingSize();
        // 园整化为最近的 4 的倍数（典型帧长度）
        remainingSize = remainingSize / 4 * 4;
    }
    if (remainingSize > 0) {
        length = Math.min(length, remainingSize);

        // 清除缓冲区
        for (int i=offset; i<offset+length; i++) {
            samples[i] = 0;
        }

        //过滤其余字节
        soundFilter.filter(samples, offset, length);
        remainingSize-=length;
```

```
                // 返回
                return length;
            }
            else {
                // 流结束
                return -1;
            }
        }
    }
```

FilteredSoundStream 类扩展 FilterInputStream 类，这个抽象类设计成过滤 InputStream，它只在一个方法 read()就可以工作。

前面曾介绍过，播放回响之类的过滤时，原声播放完毕之后，回响通常要播放更长时间。在 FilteredSoundStream 中，如果 SoundFilter 中还有余音，则 read()方法会处理余高，将源缓冲区清为全零（使其静音），然后过滤这个静音缓冲区。最后，一切完成之后，它返回-1，表示流末尾。介绍完回响过滤器之后，下面要实际生成一个回响过滤器。

4.5　创建实时回响过滤

假如你刚刚买了一辆 SUV，会像电视广告中的那样将车开到山顶上，也许每一个 SUV 车主都会有这个欲望，不是吗？

到达山顶之后，当你走出车门并看到眼前美丽的风景，你情不自禁的高呼“喂！”。一会儿山谷里传来较轻的回声，“喂！”再过一会儿，可能又听到一阵回声，但这时声音更弱了。

可以看出回声有 2 个元素：延迟与衰减，如图 4-2 所示。延迟（delay）指回声发生的时间长度，衰减（decay）指回声比原声弱多少。当你在山顶高喊时，延迟大约为一秒，衰减大约为 50%以下。衰减值为 100%时，表示回声完全消失。

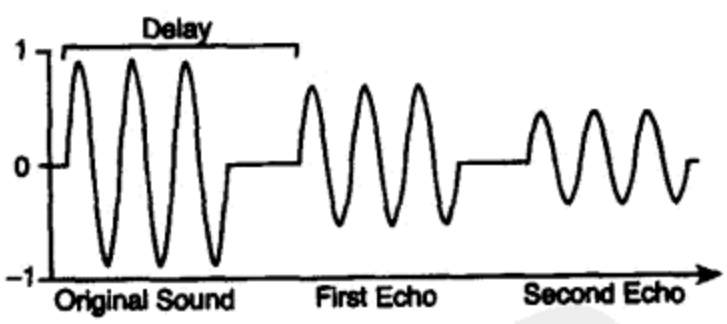


图 4-2　回声在一定延迟之后重复原声，回声比原声弱，衰减大约为 60%

由于 SoundFilter 类与格式无关，所以过滤的声音采样率是 44100Hz 还是 8000Hz 并不重要。因此，无法将延迟时间量告诉回响过滤器，而只能告诉延迟多少个样本。采样率为 44100Hz 而延迟为一秒时，即让过滤器延迟 44100 个样本。记住，延迟从声音开头算起，而不是从声音末尾算起。因此，如果完成 44100Hz 声音之后要延迟一秒，就要将延迟设置为声音中的样本数加 44100。通常所有声音要有相同的回响延迟，即对于长音，第一个回

响可能在原声仍在播放时串入。

衰减本身很简单，取值为 0 到 1，0 表示无回响，1 表示回声与原音的音量相同。但数值接近 1 时要小心。上面的实例很简单，回响在声音完成之后才出现，但对于许多声音，回响可能在原声仍在播放时串入，如果衰减数值接近 1，则可能造成扭曲，因为回声加上部分原因后，可能超过 16 位样本范围。

驾驶 SUV 车登山之后，下面就要生成 EchoFilter 类。清单 4.5 所示的 EchoFilter 类是 SoundFilter 的子类。要实现回声，就要有个延迟缓冲区，这个延迟缓冲区的长度与要延迟的样本量相同。处理样本时，将样本复制到延迟缓冲区中，从而便于生成回声效果。要生成回声效果，将延迟缓冲区中的衰减样本加进原样本中，实现代码如下所示。

```
short newSample = (short)(oldSample + decay *
    delayBuffer[delayBufferPos];
```

由于延迟缓冲区开始为全 0（静音），因此听到的第一位声音是没有过滤的原音。声音播放完毕之后，FilteredSoundStream 用静音作为原音进行过滤，这样就只能听到回声。这就使一切如期工作了。

最后，getRemainingSize()方法计算原音完成之后可以过滤的字节数。这里的计算要保证最后回声的音量为原声音量的 1%，接近静音。

清单 4.5　EchoFilter.java

```
package com.brackeen.javagamebook.sound;

/**
    EchoFilter 类是个仿真回声的 SoundFilter
    @见 FilteredSoundStream
*/
public class EchoFilter extends SoundFilter {

    private short[] delayBuffer;
    private int delayBufferPos;
    private float decay;

    /**
        生成指定延迟样本数和指定衰减率的 EchoFilter
        延迟（delay）指回声发生的时间长度。采样率为 44100Hz 而延迟为一秒时，让过滤器延
        迟 44100 个样本
        衰减（decay）指回声比原声弱多少。衰减率为 0.5 表示回声音量是原声音量的一半
    */
    public EchoFilter(int numDelaySamples, float decay) {
        delayBuffer = new short[numDelaySamples];
        this.decay = decay;
```

```
}

/**
    getRemainingSize()方法计算原音完成之后可以过滤的字节数。这里的计算要保证最后回声
    的音量为原声音量的 1%，接近静音
*/
public int getRemainingSize() {
    float finalDecay = 0.01f;
    // 从 Math.pow(decay,x) <= finalDecay 派生
    int numRemainingBuffers = (int)Math.ceil(
        Math.log(finalDecay) / Math.log(decay));
    int bufferSize = delayBuffer.length * 2;

    return bufferSize * numRemainingBuffers;
}

/**
    清除这个 EchoFilter 的内部延迟缓冲区
*/
public void reset() {
    for (int i=0; i<delayBuffer.length; i++) {
        delayBuffer[i] = 0;
    }
    delayBufferPos = 0;
}

/**
    过滤加进回声的声音样本。播放的样本加进延迟缓冲区的声音，乘以衰减率，然后把结果
    放在延迟缓冲区中，从而听到多个回声
*/
public void filter(byte[] samples, int offset, int length) {

    for (int i=offset; i<offset+length; i+=2) {
        // 更新样本
        short oldSample = getSample(samples, i);
        short newSample = (short)(oldSample + decay *
            delayBuffer[delayBufferPos]);
        setSample(samples, i, newSample);
```

```
                // 更新延迟缓冲区
                delayBuffer[delayBufferPos] = newSample;
                delayBufferPos++;
                if (delayBufferPos == delayBuffer.length) {
                    delayBufferPos = 0;
                }
            }
        }
    }
```

前面曾介绍过，应对播放的每个声音生成新的 SoundFilter。几个声音使用同一 EchoFilter 可能发生一些奇怪的现象，因为每个声音都要复制到延迟缓冲区。如果完成之后要复用 SoundFilter，则要调用 reset()方法。

使用 EchoFilter 的实例请见清单 4.6 的 EchoFilterTest 程序。此实例只在 SimpleSoundPlayer 上增加了 2 步，即生成 EchoFilter 和 FilteredSoundStream。

清单 4.6　EchoFilterTest.java

```
import java.io.*;

import com.brackeen.javagamebook.sound.*;

/**
    用回响过滤器播放声音的例子
    @见 EchoFilter
    @见 SimpleSoundPlayer
*/
public class EchoFilterTest {

    public static void main(String[] args) {
        // 装入声音
        SimpleSoundPlayer sound =
            new SimpleSoundPlayer("../sounds/voice.wav");

        // 生成声音流
        InputStream is =
            new ByteArrayInputStream(sound.getSamples());

        // 生成回响，使用 11025 个样本的缓冲区（44100Hz 声音为 1/4 秒）和 60%衰减
        EchoFilter filter = new EchoFilter(11025, .6f);
        // 生成过滤的声音流
```

```
        is = new FilteredSoundStream(is, filter);

        //播放声音
        sound.play(is);

        // 由于 Java 声音 API 的缺陷，要显式退出虚拟机
        System.exit(0);
    }

}
```

上述实例用 1/4 秒延迟和 60%衰减使播放的声音产生太空时代的回声。

介绍回声过滤器后，下面要介绍另一个过滤器一仿真三维声音的过滤器。

4.6　仿真三维声音

三维声音也称为有向声音（directional hearing），将虚拟声源放在三维空间中，给游戏者丰富的声频体验。声音更真实，游戏的空间感更强，你可能听到面前的一个坏家伙在冷笑，也可能听到远方开门的声音。

可以用许多不同效果仿真三维声音，下面是其中最常用的效果：

- 让声音随着距离而衰减，声源越远，声音越小；
- 将声音移到相应音箱，使游戏者左边的声音在左音箱中播放，游戏者右边的声音在右音箱中播放，还可以扩展为回音箱环绕声；
- 采用房间效果，使声波遇墙回弹，产生回声和振动；
- 采用多普勒效果，使声源移动时影响音调，如快速移向游戏者的声源比静止声源的音调高。如果你听过火力发动机和火车笛声，则 directional hearing 已经有这种体验。

还没有介绍三维图形，怎么介绍起三维声音呢？没关系，三维声音不仅适用于三维空间，而且所有的效果也适用于二维游戏。

4.6.1　生成三维过滤器的思路

下面就要生成三维过滤器。因为前面介绍了如何生成回声，所以这里不准备过多的介绍房间效果，而是主要介绍声音随着距离而衰减的情形。首先，解决如何改变音量？

利用 Java Sound 的 Control 可以动态改变音量和摇动声音，但是，Control 在实时使用中的质量不太好，可能产生滴答声和颤音。下面就要自己改变音量，而不是使用 Control 来实现，就是要避免滴答声和颤音。

滴答声和颤音是怎么产生的呢？问得好！当音量剧烈改变时，可能产生滴答声和颤音，如图 4-3 所示。

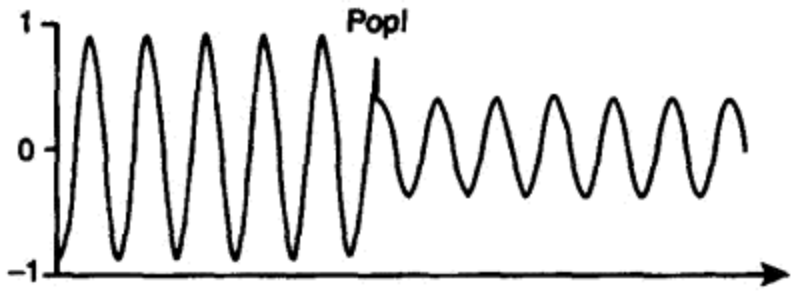


图 4-3 音量剧烈改变时，可能产生滴答声和颤音

为了在三维过滤中避免滴答声和颤音，改变音量时一定要随着时间逐渐改变音量，如图 4-4 所示。

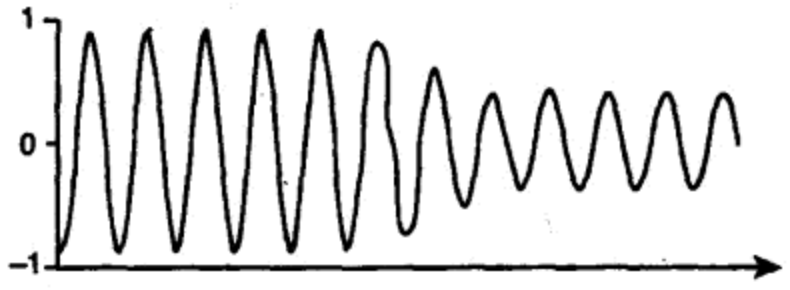


图 4-4 随着时间逐渐改变音量可以得到更自然的听觉体验

由于样本存放成 16 位带符号格式，若要改变音量，只要在将每个样本上乘一个因子，实现代码如下所示。

```
scmple = (short)(sample * volume)
```

如果要按一半音量播放声音，只要将每个样本乘以 0.5 即可。

4.6.2 实现三维过滤器

清单 4.7 所示的 Filter3d 类实现修改音量，使声音随着距离而衰减。它跟踪 2 个 Sprite 对象：一个是声源，一个是听众。源 Sprite 离听众越远，声音就越小。距离按勾股定理测量：$A^2+B^2=C^2$。

Filter3d 类还指定可以听到声音的最远距离，如果听众离声源的距离超过可以听到声音的最远距离，则根本听不到声音。音量从 0 到最远距离线性增减。

Filter3d 类设计成在幽灵位置改变时改变声音，但幽灵位置改变与声音改变之间可能有少量延迟。Line 内部用缓冲区存储声音样本，然后再实际将其发送到自然声音系统。自然声音系统可能有自己的缓冲区，声卡可能有自己的内部缓冲区，所有这些缓冲区都会产生少量延迟。

为了减少延迟，就要在 SimpleSoundPlayer 类中设置 100ms 的短缓冲区，这样可以减少延迟，但无法从根本上消除延迟。

清单 4.7 Filter3d.java

```
package com.brackeen.javagamebook.sound;

import com.brackeen.javagamebook.graphics.Sprite;
```

```
/**
    Filter3d 类是生成三维声效的 SoundFilter，让声音随着距离而衰减，声源越远，声音越小
    扩展这个类的思路之一
    将声音摇到左右音箱
    @见 FilteredSoundStream
*/
public class Filter3d extends SoundFilter {

    // 改变音量时移动的样本数
    private static final int NUM_SHIFTING_SAMPLES = 500;

    private Sprite source;
    private Sprite listener;
    private int maxDistance;
    private float lastVolume;

    /**
        生成指定源和听众幽灵的新 Filter3d 对象。运行这个过滤时可以改变幽灵位置。
        maxDistance 参数是可以听到声音的最大距离
    */
    public Filter3d(Sprite source, Sprite listener,
        int maxDistance)
    {
        this.source = source;
        this.listener = listener;
        this.maxDistance = maxDistance;
        this.lastVolume = 0.0f;
    }

    /**
        过滤声音，使其越远越小
        distance.
    */
    public void filter(byte[] samples, int offset, int length) {

        if (source == null || listener == null) {
            //什么也不过滤一返回
            return;
        }
```

```
        // 计算听众距声源的距离
        float dx = (source.getX() - listener.getX());
        float dy = (source.getY() - listener.getY());
        float distance = (float)Math.sqrt(dx * dx + dy * dy);

        // 设置音量为 0（无声）到 1
        float newVolume = (maxDistance - distance) / maxDistance;
        if (newVolume <= 0) {
            newVolume = 0;
        }

        // 设置样本音量
        int shift = 0;
        for (int i=offset; i<offset+length; i+=2) {

            float volume = newVolume;

            // 从上一音量变到新音量
            if (shift < NUM_SHIFTING_SAMPLES) {
                volume = lastVolume + (newVolume - lastVolume) *
                        shift / NUM_SHIFTING_SAMPLES;
                shift++;
            }

            // 改变样本音量
            short oldSample = getSample(samples, i);
            short newSample = (short)(oldSample * volume);
            setSample(samples, i, newSample);
        }

        lastVolume = newVolume;
    }
}
```

Filter3d 中生成了声音随距离衰减的效果，这是三维效果所需的最少音量。

如果要增加摇动，则要更新 SoundFilter 的体系结构，将单音道变成立体声，然后可以计算-1 到 1 的摇动量，-1 是左音箱，1 是右音箱，0 介于两者之间。最后，可以计算每个音箱的样本，实现代码如下所示。

```
short sampleLeft = (short)(sample * (1-pan))
short sampleRight = (short)(sample * (1+pan))
```

对二维游戏，摇动量可以由声源在屏幕上的位置确定，屏幕左边的声源在左音箱中播放，屏幕右边的声源在右音箱中播放。

4.6.4　测试三维过滤器

尽管 Filter3d 类中没有摇动，但声音随距离衰减的效果对于许多游戏已经是足够了，下面生成清单 4.8 所示的 Filter3dTest 类来进行测试。

Filter3dTest 类是个图形演示程序，可以用鼠标在屏幕上移动一只苍蝇，如图 4-5 所示。除了苍蝇之外，屏幕中央还有一个虚拟“耳朵”。苍蝇发出嗡嗡声，苍蝇离耳朵越近，声音越大。

苍蝇是个简单的三帧动画，在鼠标所在位置绘制。和过去一样，可以按 Escape 键退出演示程序。

清单 4.8　Filter3dTest.java

```
import java.awt.*;
import java.awt.event.KeyEvent;
import java.io.InputStream;
import java.io.IOException;
import javax.sound.sampled.*;

import com.brackeen.javagamebook.graphics.*;
import com.brackeen.javagamebook.input.*;
import com.brackeen.javagamebook.sound.*;
import com.brackeen.javagamebook.test.GameCore;
import com.brackeen.javagamebook.util.LoopingByteInputStream;
/**
    Filter3dTest 类演示 Filter3d 功能
    苍蝇发出嗡嗡声，苍蝇离耳朵越近，声音越大
    @见 Filter3d
    @见 SimpleSoundPlayer
*/
public class Filter3dTest extends GameCore {

    public static void main(String[] args) {
        new Filter3dTest().run();
    }

    private Sprite fly;
    private Sprite listener;
    private InputManager inputManager;
```

```
    private GameAction exit;

    private SimpleSoundPlayer bzzSound;
    private InputStream bzzSoundStream;

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

        // 设置输入管理器
        exit = new GameAction("exit",
            GameAction.DETECT_INITAL_PRESS_ONLY);
        inputManager = new InputManager(
            screen.getFullScreenWindow());
        inputManager.mapToKey(exit, KeyEvent.VK_ESCAPE);
        inputManager.setCursor(InputManager.INVISIBLE_CURSOR);

        createSprites();

        // 装入声音
        bzzSound = new SimpleSoundPlayer("../sounds/fly-bzz.wav");

        // 生成三维过滤器
        Filter3d filter =
            new Filter3d(fly, listener, screen.getHeight());

        // 生成过滤声音流
        bzzSoundStream = new FilteredSoundStream(
            new LoopingByteInputStream(bzzSound.getSamples()),
            filter);

        // 在另一线程中播放声音
        new Thread() {
            public void run() {
                bzzSound.play(bzzSoundStream);
            }
        }.start();
    }
    /**
        装入图像和生成幽灵
```

```
*/
private void createSprites() {
    // 装入图像
    Image fly1 = loadImage("../images/fly1.png");
    Image fly2 = loadImage("../images/fly2.png");
    Image fly3 = loadImage("../images/fly3.png");
    Image ear = loadImage("../images/ear.png");

    //生成苍蝇幽灵
    Animation anim = new Animation();
    anim.addFrame(fly1, 50);
    anim.addFrame(fly2, 50);
    anim.addFrame(fly3, 50);
    anim.addFrame(fly2, 50);

    fly = new Sprite(anim);

    //生成听众幽灵
    anim = new Animation();
    anim.addFrame(ear, 0);
    listener = new Sprite(anim);
    listener.setX(
        (screen.getWidth() - listener.getWidth()) / 2);
    listener.setY(
        (screen.getHeight() - listener.getHeight()) / 2);
}

public void update(long elapsedTime) {
    if (exit.isPressed()) {
        stop();
    }
    else {
        listener.update(elapsedTime);
        fly.update(elapsedTime);
        fly.setX(inputManager.getMouseX());
        fly.setY(inputManager.getMouseY());
    }
}
```

```
public void stop() {
    super.stop();
    // 停止嗡嗡声
    try {
        bzzSoundStream.close();
    }
    catch (IOException ex) { }
}

public void draw(Graphics2D g) {

    // 画背景
    g.setColor(new Color(0x33cc33));
    g.fillRect(0, 0, screen.getWidth(), screen.getHeight());

    // 画听众
    g.drawImage(listener.getImage(),
        Math.round(listener.getX()),
        Math.round(listener.getY()),
        null);

    // 画苍蝇
    g.drawImage(fly.getImage(),
        Math.round(fly.getX()),
        Math.round(fly.getY()),
        null);
    }
}
```

Filter3dTest 程序中有一点要注意，就是在另一线程中播放声音。Play()的 SimpleSoundPlayer 方法受阻，直到声音播放完毕，由于使用循环声音，因此理论上这个方法可能永远受阻。为了停止程序停滞，就要生成新线程。

记住，使用 Java Sound 的 Java 程序要用 System.exit(0)方法退出。为此可以在 Filter3dTest 扩展的 GameCore 类中增加一个 lazilyExit()方法，实现代码如下所示。

```
public void lazilyExit(){
    Thread thread = new Thread(){
        Public void run(){
            //先等待虚拟机自己退出
            try{
```

```
                    Thread.sleep(2000);
                }
                catch(InterruptedException ex){}
                //系统还在运行，因此要强制退出
                System.exit(0);
            }
        };
        };
        thread.setDaemon(true)
    thread.start
    }
```

这个方法生成新的监控程序线程，等待 2 秒钟，然后退出虚拟机。所有非监控程序线程完成后，虚拟机正常退出。如果这个线程只运行其他监控程序线程，则虚拟机自己会整洁地退出，否则经过 2 秒钟后，调用 System.exit(0)方法。恢复屏幕之后，GameCore 类中调用 lazilyExit()方法。

也许你发现了一个问题：每次播放声音都生成一个新线程并不是最整洁的方法，要解决这个问题，就是生成更高级的声音管理器。

4.7　创建声音管理器

当用 SimpleSoundPlayer 播放新的声音时，都会发生下列步骤。

（1）生成新线程。

（2）生成新字节缓冲区。

（3）生成打开和启动新 Line。

这一过程要生成许多对象，加上启动新线程和分配操作 Line 的自然资源，它们都会增加一样东西：延迟，从而使请求播放声音与实际播放声音之间相差太久。

有时延迟并不是问题，如播放启动声音时。但有时延迟是个大问题，如每秒钟发射多枚炮弹时，应尽量消除延迟。

为了减少延迟，首先就要改造声音播放体系结构，不是每次播放声音都生成一个新线程，而是用第 1 章“Java 线程”介绍的 ThreadPool 类，使几个线程作好准备，等待分配要播放声音。

然后，线程池中的每个线程可以包含自己的字节缓冲区和 Line 对象，从而避免在每次播放声音时分配字节缓冲区和 Line 对象。

这个方法的缺点是每个 Line 要事先指定一个 AudioFormat，所以只能播放一种声频格式。但这并不是什么大问题，只要选择游戏使用的声频格式，确保所有声音使用相同的声频格式即可。

4.7.1　Sound 类

下面要实现这个声音播放体系结构，首先要生成一个简单类 Sound，它包含声音样本，

如清单 4.9 所示。

清单 4.9 Sound.java

```
package com.brackeen.javagamebook.sound;

/**
    Sound 类是声音样本的容器。声音样本是格式无关的，存放成字节数组
*/
public class Sound {

    private byte[] samples;

    /**
        用指定字节数组生成新 Sound 对象。不复制字节数组
        The array is not copied.
    */
    public Sound(byte[] samples) {
        this.samples = samples;
    }

    /**
        用字节数组返回这个 Sound 的对象样本
    */
    public byte[] getSamples() {
        return samples;
    }
}
```

Sound 类是字节缓冲区的包装类，其中包含声音样本。Sound 类主要是为了增加代码可读性，隐藏声音内容，以便后面进行改变。

4.7.2 SoundManager 类

下面要生成清单 4.10 所示的 SoundManager 类。除了实现与 SimpleSoundPlayer 相似的功能（如从文件装入声音）外，SoundManager 类减少延迟，并提供暂停声音播放的功能。

清单 4.10 SoundManager.java

```
package com.brackeen.javagamebook.sound;

import java.io.*;
import javax.sound.sampled.*;
```

```
import javax.sound.midi.*;
import com.brackeen.javagamebook.util.ThreadPool;
import com.brackeen.javagamebook.util.LoopingByteInputStream;

/**
    SoundManager 类管理声音播放。SoundManager 是个线程池，每个线程一次播放一个声音。这
    样，SoundManager 很容易限制同时播放的声音数
    <p>扩展这个类的思路之一
    <li>增加 setMasterVolume()方法，用 Controls 设置每个 Line 的音量
    <li>如果请求播放已经超过 500 毫秒，则不再播放
    </ul>
*/
public class SoundManager extends ThreadPool {

    private AudioFormat playbackFormat;
    private ThreadLocal localLine;
    private ThreadLocal localBuffer;
    private Object pausedLock;
    private boolean paused;

    /**
        用最大同时声音数生成新 SoundManager
    */
    public SoundManager(AudioFormat playbackFormat) {
        this(playbackFormat,
            getMaxSimultaneousSounds(playbackFormat));
    }

    /**
        用指定的最大同时声音数生成新 SoundManager
    */
    public SoundManager(AudioFormat playbackFormat,
        int maxSimultaneousSounds)
    {
        super(Math.min(maxSimultaneousSounds,
            getMaxSimultaneousSounds(playbackFormat)));
        this.playbackFormat = playbackFormat;
        localLine = new ThreadLocal();
        localBuffer = new ThreadLocal();
```

```
        pausedLock = new Object();
        //告诉池中线程可以启动了。
        synchronized (this) {
            notifyAll();
        }
    }

    /**
        取得使用缺省混响器能播放的指定 AudioFormat 时的最大同时声音数
    */
    public static int getMaxSimultaneousSounds(
        AudioFormat playbackFormat)
    {
        DataLine.Info lineInfo = new DataLine.Info(
            SourceDataLine.class, playbackFormat);
        Mixer mixer = AudioSystem.getMixer(null);
        return mixer.getMaxLines(lineInfo);
    }

    /**
        关闭之前进行清理
    */
    protected void cleanUp() {
        // 表示结束暂停
        setPaused(false);
        //关闭混响器（停止任何运行的声音）
        Mixer mixer = AudioSystem.getMixer(null);
        if (mixer.isOpen()) {
            mixer.close();
        }
    }

    public void close() {
        cleanUp();
        super.close();
    }

    public void join() {
        cleanUp();
```

```
        super.join();
    }

    /**
        设置暂停状态，声音不一定立即暂停
    */
    public void setPaused(boolean paused) {
        if (this.paused != paused) {
            synchronized (pausedLock) {
                this.paused = paused;
                if (!paused) {
                    // restart sounds
                    pausedLock.notifyAll();
                }
            }
        }
    }

    /**
        返回暂停状态
    */
    public boolean isPaused() {
        return paused;
    }

    /**
        从文件系统装入声音。发生错误时返回 null
    */
    public Sound getSound(String filename) {
        return getSound(getAudioInputStream(filename));
    }

    /**
        从 AudioInputStream 装入声音
    */
    public Sound getSound(AudioInputStream audioStream) {
        if (audioStream == null) {
            return null;
        }
```

```
        // 取得要读取的字节数
        int length = (int)(audioStream.getFrameLength() *
            audioStream.getFormat().getFrameSize());

        // 读取整个流
        byte[] samples = new byte[length];
        DataInputStream is = new DataInputStream(audioStream);
        try {
            is.readFully(samples);
            is.close();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }

        //返回样本
        return new Sound(samples);
    }

    /**
        从文件系统的声音生成 AudioInputStream
    */
    public AudioInputStream getAudioInputStream(String filename) {
        try {
            //打开源文件
            AudioInputStream. Source =
            AudioSysrem.getAudioInputStream(new File(filename));
            //变成播放格式
            return AudioSysrem. GetAudioInputStream(
            playbackFormat,source);
        }
        catch (UnsupportedAudioFileException ex) {
            ex.printStackTrace();
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
        catch (IllegalArgumentException ex) {
            ex.printStackTrace();
```

```
        }

        return null;
    }

    /**
        播放声音。这个方法立即返回
    */
    public InputStream play(Sound sound) {
        return play(sound, null, false);
    }

    /**
        播放声音，用可选 SoundFilter，可选循环。这个方法立即返回
    */
    public InputStream play(Sound sound, SoundFilter filter,
        boolean loop)
    {
        InputStream is;
        if (sound != null) {
            if (loop) {
                is = new LoopingByteInputStream(
                    sound.getSamples());
            }
            else {
                is = new ByteArrayInputStream(sound.getSamples());
            }

            return play(is, filter);
        }
        return null;
    }

    /**
        从 InputStream 播放声音。这个方法立即返回
    */
    public InputStream play(InputStream is) {
        return play(is, null);
```

```
    }

    /**
        从 InputStream 播放声音，用可选声音过滤器，这个方法立即返回
    */
    public InputStream play(InputStream is, SoundFilter filter) {
        if (is != null) {
            if (filter != null) {
                is = new FilteredSoundStream(is, filter);
            }
            runTask(new SoundPlayer(is));
        }
        return is;
    }

    /**
        表示 PooledThread 已经启动。生成线程的 Line 和缓冲区
    */
    protected void threadStarted() {
        // 等待 SoundManager 构造函数完成
        synchronized (this) {
            try {
                wait();
            }
            catch (InterruptedException ex) { }
        }

        // 对实时改变的过滤使用短的 100 毫秒（1/10 秒）缓冲区
        int bufferSize = playbackFormat.getFrameSize() *
            Math.round(playbackFormat.getSampleRate() / 10);

        // 生成、打开与启动 line。
        SourceDataLine line;
        DataLine.Info lineInfo = new DataLine.Info(
            SourceDataLine.class, playbackFormat);
        try {
            line = (SourceDataLine)AudioSystem.getLine(lineInfo);
            line.open(playbackFormat, bufferSize);
        }
```

```
        catch (LineUnavailableException ex) {
            // Line 不可用，表示结束这个线程。
            Thread.currentThread().interrupt();
            return;
        }

        line.start();

        // 生成缓冲区
        byte[] buffer = new byte[bufferSize];

        // 设置线程的局部设置
        localLine.set(line);
        localBuffer.set(buffer);
    }

    /**
        表示 PooledThread 已经停止，排空和关闭线程的 Line
    */
    protected void threadStopped() {
        SourceDataLine line = (SourceDataLine)localLine.get();
        if (line != null) {
            line.drain();
            line.close();
        }
    }

    /**
        SoundPlayer 类是 PooledThreads 要运行的任务，从 ThreadLocal 变量接收线程的 Line 和字
        节缓冲区，并播放来自输入流的声音
        只适用于从 PooledThread 调用时
    */
    protected class SoundPlayer implements Runnable {

        private InputStream source;

        public SoundPlayer(InputStream source) {
            this.source = source;
```

```
    }

    public void run() {
        // 从 ThreadLocal 变量接收线程的 line 和字节缓冲区。SourceDataLine
          line= (SourceDataLine)localLine.get();
        byte[] buffer = (byte[])localBuffer.get();
        if (line == null || buffer == null) {
            // Line 不可用
            return;
        }

        // 将数据复制到 Line
        try {
            int numBytesRead = 0;
            while (numBytesRead != -1) {
                // 如果暂停，等待结束暂停
                synchronized (pausedLock) {
                    if (paused) {
                        try {
                            pausedLock.wait();
                        }
                        catch (InterruptedException ex) {
                            return;
                        }
                    }
                }
                // 复制数据
                numBytesRead =
                    source.read(buffer, 0, buffer.length);
                if (numBytesRead != -1) {
                    line.write(buffer, 0, numBytesRead);
                }
            }
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }

    }
```

```
    }
}
```

由于 SoundManager 类扩展 ThreadPool 类，所以将其移到 com.brackeen.javagame book.util 包中。

SoundManager 类的内类 SoundPlayer 将声音数据复制到 Line 中。SoundPlayer 实现 Runnable 接口，可以作为线程池中一个线程的任务。SoundPlayer 比 SimpleSoundPlayer 更强的一点是 SoundManager 处于暂停状态时停止复制数据。处于暂停状态时，SoundPlayer 调用 wait()方法，使线程闲机等待，直到得到通知。SoundManager 结束暂停时会通知所有等待线程。

4.7.3　线程局部变量

SoundManager 中要保证每个线程都有自己的字节缓冲区和 Line 对象，以便复用，而不必在每次播放声音时生成新对象。为了给线程池中每个线程提供自己的字节缓冲区和 Line 对象，就要利用线程局部变量。

局部变量是代码块局部的变量，而线程局部变量是每个线程中有不同值的变量。实例中，SoundManager 类的线程局部变量为 localLine 和 localBuffer。每个访问这些变量的线程都拥有自己的 Line 和字节缓冲区，所以一个线程无法访问另一线程的线程局部变量。线程局部变量用 ThreadLocal 类生成。

要使用线程局部变量，就要做点手脚，更新 ThreadPool 类。要在线程启动时生成线程局部变量，在线程死亡之时对线程局部变量进行清理。

为此，在 PooledThread 中要告诉 ThreadPool 类每个线程何时启动和停止，实现代码如下所示。

```
Public void run(){
  //表示这个线程启动
  threadStarted();
  while (!isInterrupted()){
  //取得要运行的任务
  Runnable task =null;
  try {
        task = getTask();
   }
   catch (InterruptedException ex){}
   //getTask()返回 null 或中断时
    //关闭这个线程
    if (task = =null){
        break;
    }
    //运行任务，吸收其抛出的任何异常
    try{
```

```
            task.run();
        }
        catch(Throwable t){
            uncaughtException(this,t);
        }
    }
    //表示这个线程启动停止
    threadStopped();
    }
```

在 ThreadPool 类代码中，threadStarted()和 threadStopped()方法什么也不干，而在 SoundManager 类中则派上了用场。threadStarted()方法中生成新的 Line 和字节缓冲区，并将其加进线程局部变量中。threadStopped()方法中清空和关闭 Line。

除了减少延迟和允许暂停声音播放之外，SoundManager 类还提供了更简单的声音播放方法。它会生成 ByteArrayInputStreams 或 FilteredSoundStreams，只要传入一个 Sound 对象和可选 SoundFilter 即可。

这样就完成了新的声音管理器。前面介绍了如何播放声音、使用声音过滤器和仿真三维声音，下面要介绍另一种声音——音乐。

4.8 播放音乐

并不是每个游戏都要使用背景音乐，但背景音乐可以在游戏中起重要作用。音乐可以带动情感，如动作游戏可以播放快节奏的背景音乐，而智力游戏可以播放慢节奏的背景音乐。

此外，音乐还可以改变游戏者对游戏元素的情感。音乐是快乐的吗？是剧烈的吗？快乐的音乐适合简单关卡，如游戏中的前几个关卡。而剧烈音乐适合高难度关卡，如游戏者正在对付一个强大的敌人时。

确定所要的音乐类型后，下一步要确定音乐的来源。游戏通常以 3 种方式播放音乐：

- 光盘声频磁道中的流或音乐；
- 播放 MP3 与 Ogg Vorbis 之类的压缩音乐；
- 播放 MIDI 音乐。

4.8.1 播放光盘声频

一些光盘游戏会在光盘中存放普通或红皮书的声频（标准声频光盘格式），好处是音乐音质好，而且容易实现，只要让声卡开始播放光盘，而不必在游戏中增加任何工作。另一个好处是游戏者可以将游戏光盘放进声频光盘播放器中，录入游戏磁道。

但是，光盘声频要占用很大空间，三分钟歌曲通常要 30Mbyet。如果有四个三分钟歌曲，则要占掉 120Mtyte，这样会使图形和关卡所用的空间减少。此外，Java Sound 版本不支持播放光盘声频，因此无法选择这个选项。

4.8.2 播放 MP3 与 Ogg Vorbis

第二个选项是压缩音乐。MP3 和 Ogg Vorbis 格式比光盘声频小得多，三分钟歌曲只有 3Mbyte 左右，而且音色接近光盘质量，在游戏中越来越普及。

缺点是译码问题，MP3 和 Ogg Vorbis 文件需要大量处理器功能。声卡不能直接播放压缩音乐，因此音乐要边播放边译码。这样，占用的处理器功能可能对游戏其他部分产生干扰。在现代高速机器上，译码并不明显，但在旧式低速机器上，译码要占用处理器功能的 20%到 40%，甚至更多，使游戏其他部分（如动画）速度大大减慢。

如果游戏中处理器时间不重要，则要取得 MP3 或 Ogg Vorbis Java 译码器，两者都在 Java Zoom 站上 www.javazoom.net。站点提供的插入件要使用 Java 服务提供者接口，可以取得 AudioInputStream，译码 MP3 或 Ogg Vorbis Java 流。只要包括必要的.jar 文件，并确保将 AudioInputStream 进行转换如下，实现代码如下所示。

```
//生成播放所用格式（未压缩 44100Hz，16 位，单声道，带符号，小率）
AudioFormat playbackFormat =
      New AudioFormat(44100,16,1,true,false
//打开派文件
AudioInputStream   source =
      AudioSystem.getAudioInputStream(new File(“music.ogg”))
//变成播放格式
source = AudioSystem.getAudioInputStream(playbackFormat, source)
```

不要把样本装入内存中。每分钟音乐的压缩声音文件可能只有 1MB，但解压缩后的样本则可能长达 10MB。应直接从 AudioInputStream 播放任何大的声音，其从磁盘将声音流化。

那么游戏中选 MP3 还是 Ogg Vorbis 呢？尽管 MP3 已经越来越普及，但 Ogg Vorbis 是无需可证的，它的质量更好。基于玩游戏的人并不关心这些，所以可以随便选择更符合需求的音乐格式。关于 Ogg Vorbis 的更多信息见 www.xiph.org/ogg/vorbis，关于 MP3 许可证问题见 www.mp3licensing.com。

4.8.3 播放 MIDI 音乐

最后介绍 MIDI 音乐，它不是采用离散样本方式，而是像作曲家的乐谱，通过指令指示每个乐器上要插的音符来生成音频。声频系统会根据音量、乐器、音调和其他参数合成每个音符。

由于 MIDI 音乐包含指令而不是样本，因此比采样声音格式小得多，通常只有几 KB，而不是几 MB。

基于 MIDI 音乐是合成的，所以它的质量不如采样声音格式那么高。有些乐器听起来不够逼真，甚至有机械的味道，不过创造性的乐手正是能够掩盖 MIDI 缺陷的。

Java Sound API 通过声音乐队合成 MIDI 音乐，它就是一组乐器。但是，Java SDK 中包括声音乐队而 Java 运行环境中却不包括。如果找不到声音乐队，则使用硬件 MIDI 端口。

因为它在 Java Sound API 中的定时不可靠，所以建议使用声音乐队。

Java SDK 中带有质量很差的声音乐队，可以从 http://java.sun.com/products/java-media/sound/soundbanks.html 下载质量很好的声音乐队和将其放进游戏中，Java Sound API 在 javax.sound.midi 包中提供了 MIDI 声音功能。要播放 MIDI 音乐，需要这个包中的 2 个对象 Sequence 和 Sequencer。Sequence 对象包含 MIDI 数据，而 Sequencer 将一个 Sequence 发送到 MIDI 合成器中。下面是播放 MIDI 音乐的实例，代码如下所示。

```
//打开 midi 文件
Sequence   sequence = MidiSystem.getSequence(new File(filename));
//打开序列化器
Sequencer   sequencer = MidiSystem.getSequence（）
Sequencer.open();
//播放 midi 序列
Sequencer.setSequence(Sequence);
Sequencer.start();
```

缺省情况下，Sequencer 将 Sequence 播放一次，然后停止，但游戏中通常要让音乐循环。要让 Sequence 循环，就要在音乐播放完毕时发出通知并重新启动 Sequencer。

清单 4.11 的 MidiPlayer 类是让音乐循环的实例，它实现 MetaEventListener 接口，通过磁道结尾消息在音乐播放完毕时发出通知。

清单 4.11　MidiPlayer.java

```
package com.brackeen.javagamebook.sound;

import java.io.*;
import javax.sound.midi.*;

public class MidiPlayer implements MetaEventListener {

    // Midi 元事件
    public static final int END_OF_TRACK_MESSAGE = 47;

    private Sequencer sequencer;
    private boolean loop;
    private boolean paused;

    /**
        生成新 MidiPlayer 对象
    */
    public MidiPlayer() {
```

```
        try {
            sequencer = MidiSystem.getSequencer();
            sequencer.open();
            sequencer.addMetaEventListener(this);
        }
        catch ( MidiUnavailableException ex) {
            sequencer = null;
        }
    }

    /**
        从系统中装入序列，发生错误时返回 null
    */
    public Sequence getSequence(String filename) {
        try {
            return getSequence(new FileInputStream(filename));
        }
        catch (IOException ex) {
            ex.printStackTrace();
            return null;
        }
    }

    /**
        播放序列，可选循环。这个方法立即返回。无效序列不播放
    */
    public void play(Sequence sequence, boolean loop) {
        if (sequencer != null && sequence != null && sequencer.isOpen()) {
            try {
                sequencer.setSequence(sequence);
                sequencer.start();
                this.loop = loop;
            }
            catch (InvalidMidiDataException ex) {
                ex.printStackTrace();
            }
        }
    }
```

```
/**
    发生元事件时，声音系统调用这个方法。这时，收到磁道结尾元事件时，如果循环打开，
    则序列重新启动
*/
public void meta(MetaMessage event) {
    if (event.getType() == END_OF_TRACK_MESSAGE) {
        if (sequencer != null && sequencer.isOpen() && loop) {
            sequencer.start();
        }
    }
}

/**
    停止序列化器，将其复位为 0
*/
public void stop() {
    if (sequencer != null && sequencer.isOpen()) {
        sequencer.stop();
        sequencer.setMicrosecondPosition(0);
    }
}

/**
    关闭序列化器
*/
public void close() {
    if (sequencer != null && sequencer.isOpen()) {
        sequencer.close();
    }
}

/**
    取得序列化器
*/
public Sequencer getSequencer() {
    return sequencer;
}

/**
```

```
    设置暂停状态，音乐不一定立即暂停
*/
public void setPaused(boolean paused) {
    if (this.paused != paused && sequencer != null && sequencer.isOpen()) {
        this.paused = paused;
        if (paused) {
            sequencer.stop();
        }
        else {
            sequencer.start();
        }
    }
}

/**
    返回停止暂停
*/
public boolean isPaused() {
    return paused;
}

}
```

MidiPlayer 还提供打开 Sequence 和暂停音乐的方法。要暂停音乐，MidiPlayer 调用 Sequencer 对象的 stop()方法，停止序列而不复位其位置。调用 start()方法则恢复暂停的序列。这一切都发生在 MidiPlayer 的 setPaused()方法中。

4.8.4　生成适配音乐

在能够实现播放 MIDI 音乐之后，下面了解它的一个优点：即很容易实现适配音乐。适配音乐（Adaptive）是根据游戏状态改变的音乐，如游戏者正在打击大量敌人，则音乐可能又快又响。相反，游戏者悄悄地在房子中探险时，音乐可能非常小声。

音乐的改变可能随时发生，如游戏者前一秒钟可能还闲逛，后一秒钟突然有 100 个机器人要杀他，想要顺畅地改变音乐可不是件容易的事。

可以用 2 种方法在游戏状态中适配音乐：

- 改变歌曲；
- 修改当前播放的歌曲。

由于游戏者的动作可能随时改变，因此改变歌曲要困难得多。因为改变不能太突然，否则会分散注意力。可以将歌曲设计成具有“改变点”方式，即表示歌曲可以在何处发生改变。

另外，歌曲需要光顺过渡，为此可以让第一首歌慢慢退去，而第二首歌慢慢加强。另

外，在第一首歌慢慢退去时，其速度可以变成第二首歌的速度。

也可以用更简单的方法，插入溜声机刮针的声音。此方法是简单地修改现有歌曲，为此可以改变速度或音量，也可以在其中增加另一乐器。

MIDI 序列中很容易增加和减少乐器。MIDI 音乐通常组织成乐曲，每个乐曲演奏特定乐器，如可能有一个吉他乐曲，一个键盘乐曲等。

可以用一个方法让乐曲静音或取消静音，实现代码如下所示。

```
Sequencer.setTrackMute(trackNum,true)
```

其中参数 trackNum 是个整数，表示要静音的乐曲号。如果不知道 MIDI 文件中哪个乐曲属于哪个乐器，则可以通过一个乐曲一个乐曲静音来试验。

清单 4.12 的 MidiTest 类就是操纵 MIDI 音乐的，这个短测试用 MidiPlayer 开始一首歌，首先关掉鼓声，演奏一遍之后，再打开鼓声演奏一遍。

清单 4.12　MidiTest.java

```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.*;

import com.brackeen.javagamebook.sound.MidiPlayer;

/**
    播放 Midi 序列的例子。首先关掉第 1 个乐曲，演奏一遍，再打开第 1 个乐曲演奏一遍。例子
    Midi 序列中第 1 个乐曲是鼓声
*/
public class MidiTest implements MetaEventListener {

    // 例子 Midi 序列中的是鼓声乐曲
    private static final int DRUM_TRACK = 1;

    public static void main(String[] args) {
        new MidiTest().run();
    }

    private MidiPlayer player;

    public void run() {

        player = new MidiPlayer();
```

```
        // 装入序列
        Sequence sequence =
            player.getSequence("../sounds/music.midi");

        //播放序列
        player.play(sequence, true);

        // 关掉鼓声
        System.out.println("Playing (without drums)...");
        Sequencer sequencer = player.getSequencer();
        sequencer.setTrackMute(DRUM_TRACK, true);
        sequencer.addMetaEventListener(this);
    }

    /**
        发生元事件时，声音系统调用这个方法。这时，收到磁道结尾元事件时，打开鼓声
    */
    public void meta(MetaMessage event) {
        if (event.getType() == MidiPlayer.END_OF_TRACK_MESSAGE) {
            Sequencer sequencer = player.getSequencer();
            if (sequencer.getTrackMute(DRUM_TRACK)) {
                // 打开鼓声
                System.out.println("Turning on drums...");
                sequencer.setTrackMute(DRUM_TRACK, false);
            }
            else {
                // 关闭序列化器并退出
                System.out.println("Exiting...");
                player.close();
                System.exit(0);
            }
        }
    }
}
```

和播放样本声音时一样，完成之后要显式退出虚拟机。收到第二个磁道尾部消息时，MidiTest 退出虚拟机。

4.9 小结

本章介绍了声音、声音过滤器和音乐的基本知识，建立了实时回响与伪三维过滤器，生成了处理游戏声效的精彩声音管理器。我们还建立了一些适配音乐，介绍了如何播放 MP3 与 Ogg Vorbis 声音文件。将这些知识与前几章介绍的图形和交互性结合起来，就可以创建游戏了。

第 5 章　创建二维平台游戏

关键问题

- 生成基于地砖的地图
- 碰撞探测
- 完成与加速
- 生成可执行 java 文件
- 扩展游戏
- 小结

下面就要开始做一些有趣的事情。

前面几章介绍使用 Java 来实现图形、声音和交互性的问题，本章就要用这些内容创建真正的二维平台游戏。平台游戏（Platform games）是指让游戏者在平台上进行跳动和跑动的游戏，主要实现避开敌人和收集物品（HP）等动作。若要生成平台游戏，就要使用基于地砖的地图、地图文件、碰撞探测、收集物、简单敌人和视差背景滚动，如图 5-1 所示，二维平台游戏生成的屏幕图像。

注意：本章不是介绍 Java 技术，而是介绍游戏编程技术，因此没有提供游戏的所有代码，如果想要得到相关代码，可以从网址 www.brackeen.com/javagamebook 下载。

图 5-1　二维平台游戏的屏幕图像

5.1　生成基于地砖的地图

在二维平台游戏中，整个游戏的地图（game map）通常要有几个屏幕的大小，甚至像 20 个或 100 个都不算为过。随着主角在屏幕上走动，地图也将不断地改变。

你可以想像出，使用一幅巨大的图像作为游戏的地图是不理想的。因为这样将占用大量的内存，导致计算机无法装入地图。此外，这样的游戏地图无法定义哪些部分是否让游戏者走动，换句话说，不能定义哪些部分是实的，哪些部分是空的。

通常二维平台游戏主要生成基于 Tile-based maps（地砖地图）的画面，将整个地图分

解成若干栅格，如图 5-2 所示。每个栅格中都包含若干单元，单元表示地图中的图像元素（包括空元素）。

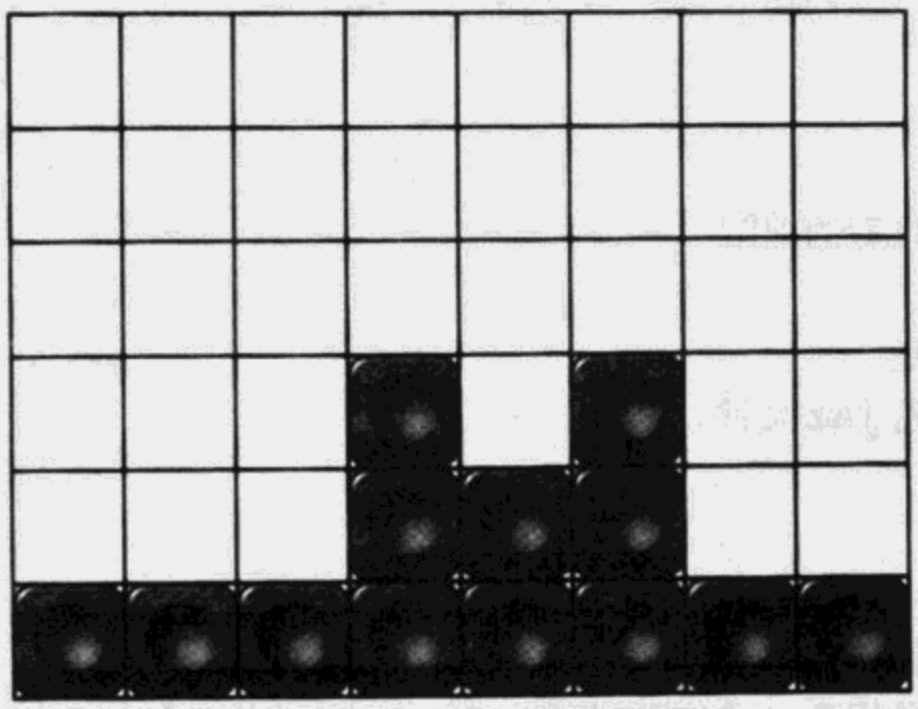


图 5-2　一个地砖地图的构成

生成 Tile-based maps 就像是搭彩色积木游戏，用颜色来表示不同的用途，但每种颜色的数量却不限制。

这里只关心图像元素将归属哪一个栅格，只需将图像元素添加到地砖地图中即可。这样就可以把游戏地图做得任意大，而且不必太担心内存的限制。

一个 Tile-based maps 所采用的栅格数大多是 2 的指数，如 16 和 32。在本章游戏实例中，栅格数是 64，并且如图 5-3 所示，游戏共使用的 9 种图像元素。

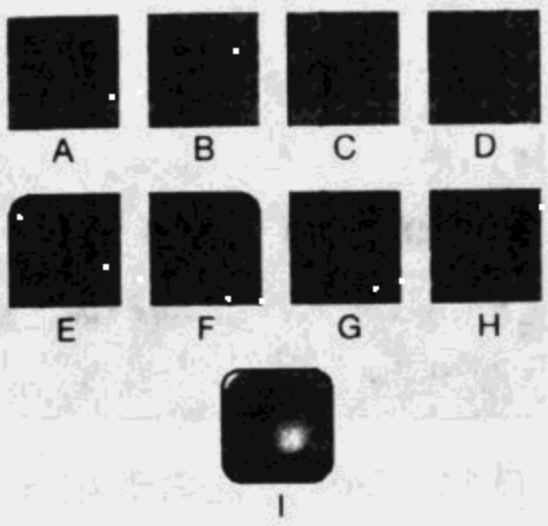


图 5-3　游戏使用的 9 种图像元素

Tile-based maps 还有一个好处，那就是很容易确定哪些部分是实的，哪些部分是空的，从而知道哪些部分是否让游戏者走动，不会让游戏者莫名其妙地穿墙而过。本章稍后在“碰撞探测”一节中实现这个设定，那么首先实现 Tile-based maps 吧。

5.1.1　实现 Tile 地图

如清单 5.1 所示的 TileMap 类，它将包含游戏中所要的地图。TileMap 类使用 Image 类

的二维数组来定义 Tile- based maps，其中把空元素设为 null。

这里要明确一个概念，Image 数组中每一个项目不是唯一的对象，而是对现有对象的引用。如果某个图像元素在地图中出现几次，那么相对的 Image 对象也将在数组中出现几次。32 位 Java 虚拟机执行对象引用要占 4 个字节，因此一维 5000 个 Image 对象的数组要占 20KB（实际上，由于 Java 没有二维数组，所以要用数组的数组表示。一个“二维数组”也只有 30KB）。

清单 5.1　TileMap.java

```
package com.brackeen.javagamebook.tilegame;

import java.awt.Image;
import java.util.LinkedList;
import java.util.Iterator;

import com.brackeen.javagamebook.graphics.Sprite;

/**
    TileMap 类包含地砖地图的数据，包括幽灵。每个地砖是一个图像的引用。当然 Image 在地砖
    地图中多次使用
*/
public class TileMap {

    private Image[][] tiles;
    private LinkedList sprites;
    private Sprite player;

    /**
        生成指定宽和高的新 TileMap（地砖个数）
    */
    public TileMap(int width, int height) {
        tiles = new Image[width][height];
        sprites = new LinkedList();
    }

    /**
        取得 TileMap 的宽度（水平方向地砖个数）
    */
    public int getWidth() {
        return tiles.length;
```

```
    }
    /**
        取得 TileMap 的高度（垂直方向地砖个数）
    */
    public int getHeight() {
        return tiles[0].length;
    }

    /**
        取得指定位置的地砖。指定位置没有地砖时或地址越界时返回 null
     */
    public Image getTile(int x, int y) {
        if (x < 0 || x >= getWidth() ||
            y < 0 || y >= getHeight())
        {
            return null;
        }
        else {
            return tiles[x][y];
        }
    }

    /**
        设置指定位置的地砖
    */
    public void setTile(int x, int y, Image tile) {
        tiles[x][y] = tile;
    }

    /**
        取得游戏者幽灵
    */
    public Sprite getPlayer() {
        return player;
    }

    /**
        设置游戏者幽灵
    */
```

```
    public void setPlayer(Sprite player) {
        this.player = player;
    }

    /**
        将 Sprite 对外加进这个地图
    */
    public void addSprite(Sprite sprite) {
        sprites.add(sprite);
    }

    /**
        从这个地图删除 Sprite 对象
    */
    public void removeSprite(Sprite sprite) {
        sprites.remove(sprite);
    }

    /**
        取得这个地图中所有幽灵的迭代器，游戏者幽灵除外
    */
    public Iterator getSprites() {
        return sprites.iterator();
    }
}
```

除图像元素以外，TileMap 类还包含幽灵。游戏规定幽灵可以在地图上的任何位置出现，如图 5-4 所示游戏中包含的幽灵。

图 5-4　除了游戏者，幽灵还包括 2 个坏家伙和 3 个收集物

TileMap 类会把游戏者作为特殊的幽灵，和其他幽灵相比，会进行不同的处理。

5.1.2 读取 Tile 地图

虽然生成了 Tile-based maps，但还要运用合理的方式来显示它们。

二维平台游戏总是有很多的地图和关卡，本章的游戏也不例外。所以要用简便的方法显示多个地图，而且让地图间的调用更加合理和通顺。

当游戏进行时，要显示每一个地图元素和幽灵，可以调用 TileMap 类的 addTile()和 addSprite()方法。不难想象，这 2 种方法的实现就是关卡困难的标准。

多数二维平台游戏会使用自己的地图编辑器来生成和显示地图。编辑器可以在游戏中直观地增加地图元素和幽灵，使用也方便快捷。它们通常在游戏可以分析的中间地图文件中存放地图。

本章没有必要建立地图编辑器，只定义基于文本格式的地图文件，这样在任何的文本编辑器即可实现。因为 Tile-based maps 在栅格上定义，文本文件中的每个字符均为图像元素或幽灵，如清单 5.2 所示。

清单 5.2 map.txt

```
#基于地砖游戏的地图文件
# (#开头的行是说明行)
# 地砖为:
#    (Space) 空文件
#    A..Z    地砖 A 到 Z
#    o       星星
#    !       音乐音符
#    *       目标
#    1       坏蛋 1 (蛉螬)
#    2       坏蛋 2 (苍蝇)
AD        IIIIIII                IIIIIIIIIIIIIIIIIIIIIIIIIIIII
AD                ooo                I                          E
AD                IIIIIII            I                          C
AD                                   I                 oo       C
AD            2   !       2                            IIIII    C
AD                IIIII                                         C
AD        III                        1    1  1        2        C
AD                                 1  IIIIIIIIIII               C
AD      1     EBBBBBBBBBBBBBBBF  oooooo                        *C
AHBBBBBBBBBBBBBBGAAAAAAAAAAAAAAAHBBBBBBBBBBBBBBBBBBBBBBBBBBBBBG
```

上述地图文件很容易理解和编辑，以符号#开头的行是说明行，而其他的行将定义图像元素。此外，地图不是固定的，可以加行或加长行，从而使地图加大。

地图文件很容易分析，主要有 3 步。

（1）读取每一行，忽略说明行，将每行放在清单中。

（2）生成 TileMap 对象。其中 TileMap 对象的宽度是清单中最长一行的长度，高度是清单中的行数。

（3）分析每一行每个字符，并根据字符在地图中增加相应图像元素和幽灵。

如果遇到“非常”字符，则认为是空元素。

下面把分析代码包装到如清单 5.3 所示的 loadMap()方法中去。

清单 5.3　loadMap()方法

```
private TilMap loadMap (String filename) throws IOException

    ArrayList lines = new ArrayList();
    Int width = 0;
    Int height = 0;
    //读取每一行，将每行放在清单中。
    BufferedReader reader = new BufferedReader（
        new FileReader(filename));
    while (true){
        String line = reader.readLing();
        //没有更多要读的行
        if (line == null){
            reader.close();
            break;
        }
        //增加每个非说明行
        if(!line.startsWith(“#”)){
            lines.add(line)
            width = Math.max (width iine.length())
        }
    }
    //分析行，生成 TileEngine
    height = lines.size();
    TileMap newMap = new TileMap(width, height);
    for(int y=0; y< height;Y++){
        String line = (String)lines.get(y)
        for (int x=0;x<line.length();x++){
            char ch = line.charAt(x);

            //检查字符是否表示地砖 A、B、C，等等
            int tile = ch －  ‘A’;
```

```
            if (tile >=0 && tile < tile.size()){
            newMap.setTile(x,y,(Image) tiles.get(tile))
        }
        //检查字符是否表示幽灵
         else if (ch == '0' ){
             addSprite(newMap,coinSprite, x, y);
         }
         else if (ch == '!' ){
             addSprite(newMap,musicSprite, x, y);
         }
         else if (ch == '*' ){
             addSprite(newMap,goalSprite, x, y);
         }
         else if (ch == '1' ){
             addSprite(newMap,grubSprite, x, y);
         }
         else if (ch == '2' ){
             addSprite(newMap,flySprite, x, y);
         }
        }
    }
    //将游戏者加进地图中
    Sprite player = Sprite playerSprite.clone();
    player.setX (TileMapRenderer.tilesToPixels(3));
    player.setY(0);
    newMap.setPlayer(player);

    return newMap;
     }
```

需要注意的是，将幽灵加进 loadMap()的特殊情形。首先，要对游戏中每个幽灵生成不同的 Sprite 对象，为此可以从“主”幽灵上复制。其次，幽灵的尺寸和图像元素的尺寸不一定相同，所以要图像元素居中和向下对齐每个幽灵。这些实现都是清单 5.4 的 addSprite()方法完成的。

清单 5.4　addSprite()方法

```
Private void addSprite(TileMap map,
        Sprite hostSprite, int tileX,int tileY)
{
        if (hostSprite ! = null){
```

```
        //从“主”幽灵复制幽灵
        Sprite   sprite =( Sprite)hostSprite.clone();
        //居中显示幽灵
        sprite.setX(
            TileMapRenderer.tilesToPixels(tileX) +
            (TileMapRenderer.tilesToPixels(1) -
            Sprite.getWidth())/ 2);
          //向下对齐每个幽灵
        sprite.setY（
            TileMapRenderer.tilesToPixels(tileY  +  1)  -
            Sprite.getHeight());
          //加进地图中
        map. addSprite(sprite)
          }
      }
```

在本章前面的演示中，幽灵的位置是相对于屏幕的，而这里的幽灵位置是相对于 Tile-based maps 的。原因是你可以用 TileMapRender.tilesTopixels()静态函数将图像元素的位置换算成屏幕中象素的位置，只要将图像元素数乘以栅格数。

```
int pixelSize = numTiles * TILE_SIZE
```

这样，幽灵就可以在地图中移到任何位置，而不必和图像元素的边界对齐了。

现在有了灵活的方法，可以生成地图和通过分析地图生成 TileMap 对象。在本章游戏中，所有地图都将放在 map 文件夹中，命名为 map1.txt, map2.txt。移到下一个地图时，代码将寻找下一个地图文件，若找不到，则代码再次从第一个地图文件开始。这样就可以将新的地图文件放进 map 文件夹中，而不必告诉游戏有多少个地图了。

5.1.3　绘制 Tile 地图

前面曾介绍过，Tile-based maps 要比屏幕大得多。因此屏幕上一次只显示地图的一部分，当游戏者移动时，地图也将滚动，并使游戏者保持在屏幕的中央，如图 5-5 所示。

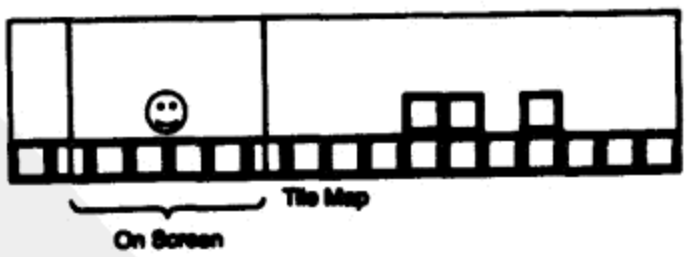


图 5-5　游戏者在地图中的位置

在绘制 Tile-based maps 之前，首先要确定屏幕在地图中的位置，然后要使游戏者保持在屏幕的中央，实现代码如下。

```
int offsetX = screenWidth/ 2  -
```

```
Math.round(player.getX()) - TILE_SIZE;
```

代码指定 offsetX 是屏幕的水平位置。代码很简单，但还要保证游戏者接近地图左边或右边沿时，滚动将停止，以免显示地图边沿以外的“空”部分。为此，offsetX 值要指定一个极限，实现代码如下。

```
int mapWidth = tilesTopixels(map.getWidth())
offsetX = Math.min(offsetX, 0);
offsetX = Math.max(offsetX, screenWidth - mapWidth);
```

为了方便起见，还要生成垂直滚动位置 offsetY 变量，不管屏幕多大，它都保持地图与屏幕底部对齐，实现代码如下。

```
int offsetY = screenHeight - tilesTopixels(map.getHeight());
```

现在就可以绘制 Tile-based maps 了。可以在地图上绘制每图像元素，但实际上只要绘制要显示的元素即可。这里根据 offsetX 的取值绘制的第一块水平栅格，然后根据屏幕宽度计算绘制的最后一块水平栅格，实现代码如下。

```
int   firstTileX = PixelsToTiles(-offsetX);
int   lastTileX = firstTileX +
        PixelsToTiles(screenWidth) + 1
for   (int y=0; y<map.getHeight();y++) {
        for   (int x= firstTileX;x<= lastTileX;x++){
              Image image = map.getTile(x,y);
              If (image != null){
                    g.drawImage(image,
                         tilesToPixels(x) + offsetX
                              tilesToPixels(y) + offsetY
                              null);
                 }
        }
}
```

这样，屏幕上就绘制了地图，可以在游戏者移动时光顺地滚动。下面要实际画出游戏者，同时画出所有其他的幽灵。

5.1.4 绘制幽灵

绘制 Tile-based maps 之后就要绘制幽灵了。由于只绘制显示的地砖，所以也就绘制显示的幽灵，下面先看如下的思路：

- 将幽灵分区到屏幕尺寸的段，只绘制屏幕上显示的幽灵。当幽灵移动时，确保它存放在相应的段中；
- 从左到右按幽灵水平位置将幽灵存放在一个排序清单中。跟踪清单中第一个的显示，随着敌人死亡和屏幕滚动，其可能将会改变。从清单中第一个显示开始绘制，

直到遇到清单中第一个不显示的出现。幽灵移动时，确保将清单重新排序；

- 用强力方法遍历清单中每个幽灵，检查其是否显示。

如果地图中有大量的幽灵，则前 2 种思路无疑很有用。但这里每个地图中只有几个幽灵，所以本章游戏中采用强力方法遍历清单中每个幽灵，检查其是否显示，实现代码如下所示。

```
Iterator I = map.getSpretes();
While (i.hasNext()){
    Sprite sprite = (Sprite) i.next();
    int x = Math.round(Sprite.getX()) + offsetX
    int y = Math.round(Sprite.getY()) + offsetY
    g.drawImage(Sprite.getImage(),x, y, null)
}
```

图形引擎会做这个工作，首先检查每个图像是否显示，然后再绘制每个图像。虽然不是最高效的方法，但是可行的方法。

5.1.5　视差滚动

画完图像元素和幽灵之后，还要画背景。画背景时，就要确定背景相对于地图的画法，可以用如下几种方法实现：

- 背景保持静态，在地图滚动时不滚动；
- 在地图滚动时以相同速率滚动背景；
- 在地图滚动时以较慢的速率滚动背景，让人感到背景离我们更远。

第 3 种方法称为视差滚动（parallax scrolling），是游戏中常见的使用方法。视差就是不同视点观看时，对象位置的明显改变，如开动汽车时若望窗外看，则交通信号灯等近景的动作将比较快，而青山等远景的动作比较慢。如果在地图滚动时以较慢的速率滚动背景，则可以让人感到背景离我们更远，从而产生立体感。

前面曾介绍过，整个地图不用巨大的图像。同样，也适用整个地图背景。由于使用视差滚动，只要生成 2 倍屏幕宽度的背景图像，地图从左滚动到右时，它会从第一个屏幕滚动到第二个屏幕，这是游戏中实现视差滚动的关键。

游戏者在地图左边时，显示地图最左边的部分，绘制背景左部，如图 5-6 所示。

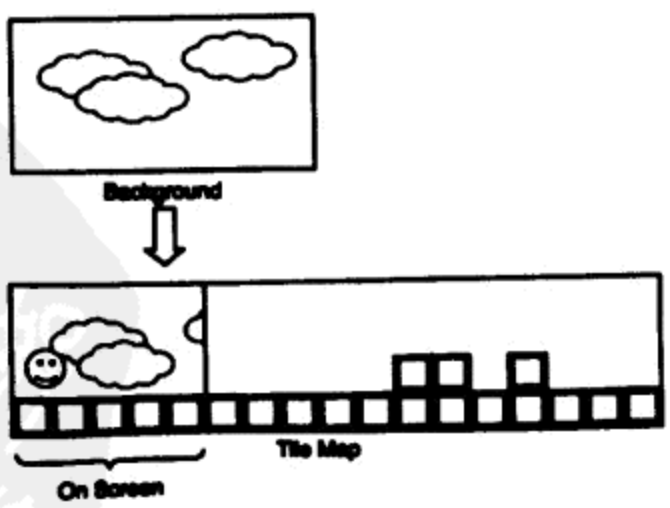


图 5-6　游戏者在地图左边时绘制背景左部

绘制背景左部很简单，实现代码如下。

```
int bcakgroundX = 0;
g.drawImage (background, backgroundX, 0,null)
```

同样，游戏者在地图右边时，绘制背景右部，如图 5-7 所示。

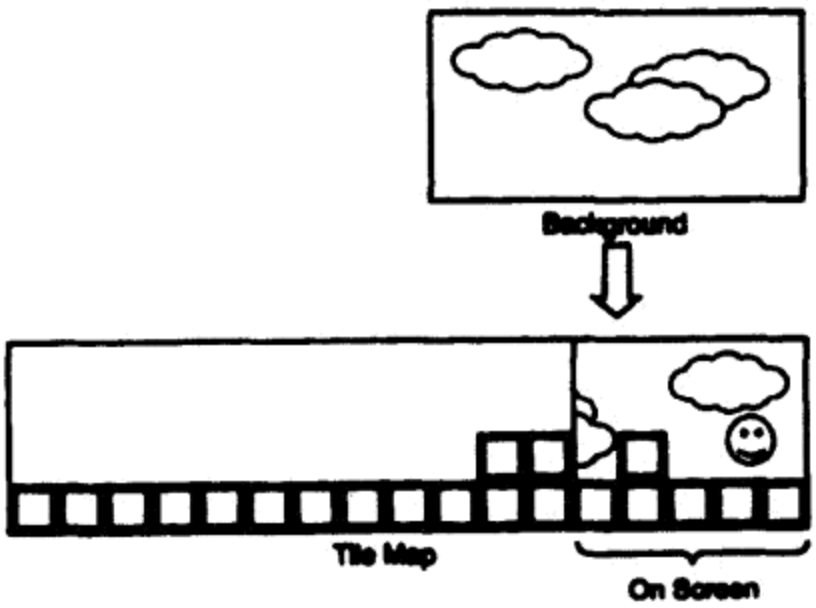


图 5-7 游戏者在地图右边时，绘制背景右部

绘制背景右部很简单，实现代码如下。

```
int bcakgroundX  = screenWidth - background.getWidth(null);
g.drawImage(background, bcakgroundX, 0,null);
```

那么地图左右 2 部分之间的点将怎么办呢？前面计算了 offsetX，是屏幕上绘制的地图位置，只要一个将 offsetX 转换成 bcakgroundX 的公式即可。OffsetX 的范围为 0（所绘制地图的左边）到 screenWidth-mapWidth（所绘制地图的右边），相应 bcakgroundX 的范围为 0 到 screenWidth- background.getWidth(null)。这样，只要在这 2 个点之间进行插值即可，实现代码如下。

```
int bcakgroundX  =  offsetX *
      (screenWidth - background. getWidth(null))/
      (screenWidth - mapWidth);
g.drawImage(background, bcakgroundX, 0,null);
```

多么简单！利用这个公式，游戏者在地图上移动时，背景即可慢慢滚动。这个公式作了 2 个假设：

- 背景比屏幕宽；
- 地图比背景宽。

最后要注意的一点是，背景不一定要使用图像，也可以用另一个 TileMap，从而可以生成任意大的背景。另外，不一定只有一个滚动背景，也可以有 2 个或 3 个滚动背景，它们可以以不同的速度滚动，但前面的层要具有透明度，以便透视后面的层。

背景图像本身也可以平铺。如果这样，则要保证背景右边沿与左边沿匹配，使背景显示为无缝隙。

5.1.6　充电器

有了绘制背景的公式就可以绘制游戏中所需的一切，首先就要实现幽灵。

要实现的第一种幽灵是充电器。充电器幽灵是让游戏者收集它们，通常会给游戏者生命力、能量或进行其他某些动作。

如清单 5.5 所示的 PowerUp 类，它就定义了充电器。下面扩展 PowerUp 抽象类，对游戏中每个充电器包含静态内子类：星星、音符和目标。

清单 5.5　PowerUp.java

```
package com.brackeen.javagamebook.tilegame.sprites;

import java.lang.reflect.Constructor;
import com.brackeen.javagamebook.graphics.*;

/**
    PowerUp 类是个 Sprite ，游戏者可以收集
*/
public abstract class PowerUp extends Sprite {

    public PowerUp(Animation anim) {
        super(anim);
    }

    public Object clone() {
        // 通过反射生成正确子类
        Constructor constructor = getClass().getConstructors()[0];
        try {
            return constructor.newInstance(
                new Object[] {(Animation)anim.clone()});
        }
        catch (Exception ex) {
            // 不能发生
            ex.printStackTrace();
            return null;
        }
    }

    /**
        星星充电器，给游戏者增加生命力
```

```
*/
public static class Star extends PowerUp {
    public Star(Animation anim) {
        super(anim);
    }
}

/**
    音乐充电器，改变游戏音乐
*/
public static class Music extends PowerUp {
    public Music(Animation anim) {
        super(anim);
    }
}

/**
    目标充电器，进入下一帧地图
*/
public static class Goal extends PowerUp {
    public Goal(Animation anim) {
        super(anim);
    }
}

}
```

游戏中的每个幽灵都要实现 clcne()方法，以便得到同一幽灵的多个拷贝。这里不是在 PowerUp 的每个子类中实现 clone()方法，而是在 PowerUp 类中提供一般性的 clone()方法，通过反射克隆对象，即使它是子类。它选择对象类的第一个构造函数，然后用构造函数生成这个对象的新实例。

其实 PowerUp 的子类没干什么事情，只是个占位符，表示各个充电器的作用，那它们的作用是什么呢？

- 游戏者取得星星时，播放一个声音而不采取其他动作。
- 游戏者取得音符时，打开/关掉背景 MIDI 音乐中的鼓声。
- 最后，游戏者取得“目标”充电器时，装入下一个地图。

所有充电器动作都发生在碰撞探测代码中（见本章稍后介绍）。游戏者与 PowerUp 碰撞时，调用 acquirePowerUp()方法，确定要怎么处理。

```
public void acquirePowerUp(PowerUp powerUp){
```

```
        //从地图中删除
      map.removeSprite(powerUp)
if (powerUp instanceof PowerUp.Star){
      //进行一些处理，如增加游戏者生命力
      soundManager.play(prizeSound);
}
else if (powerUp instanceof PowerUp.Music){
     //改变音乐
   soundManager.play(prizeSound);
   toggleDrumPlayback();
}
else if (powerUp instanceof PowerUp.Goal){
        //转入下一地图
      soundManager.play(prizeSound,
              new EchoFilter(2000, .7f), false);
      map = resourceManager.loadNextMap();
  }
    }
```

说明：这只是实现充电器的一种方法。另一种方法是在 powerUp 类中提供一个 acquire()方法，在游戏者取得充电器时调用。这样，每个充电器负责自己的动作。但这个游戏设计的 powerUp 对象不能访问地图和声音管理器，powerUp 对象不执行任何动作。

5.1.7　简单坏人

没有坏人的游戏是没有任何乐趣的，对吗？

但坏人不一定要智能化，本章游戏中的 2 个坏蛋蚧蟾和苍蝇都不是智能化的。因为它们只是向前移动，直到遇到墙壁，然后掉转头再继续向前移动，就这样不断的重复。

在实现坏人之前，首先介绍它的动画，如图 5-4 所示。游戏者和 2 个坏蛋都是脸朝左边的，所以不必生成更多脸朝右边的 PNG 文件，因为只要将脸朝左边的图像进行镜像即可。

第 2 章“二维图形与动画”曾介绍过，AffineTransform 类可以将图像进行镜像，并将生成的镜像图像保存为另一个图像，如清单 5.6 所示，生成的代码还要建立水平镜像（翻折）图像。

清单 5.6　生成变换图像

```
public Image getMirrorImage(Image image){
      return getScaledImage(image, -1, 1);
}
public Image getFlippedImage(Image image){
       return getScaledImage(image, 1, -1);
```

```
}
private Image getFlippedImage(Image image,float x, float y){
        //建立变换
        AffineTransform transform = new AffineTransform()
        transform.scale(x,y);
        transform.translate(
              (x-1) * image.getWidth(null) / 2
              (y-1) * image.getWidth(null) / 2);
       //生成透明（不是半透明）图像
        Image   newImage   =   gc.createCompatibleImage
              Image. getWidth(null),
              Image. getHeight(null),
              Transparency.BITMASK);
      //绘制变换图像
      Graphics2D g = (Graphics2D) newImage.getGraphics();
      g.drawImage (image, transform , null);
      g.dispose();
      return newImage
}
```

这样就可以建立游戏者和坏蛋向右的移动了。此外，还要一个显示坏蛋“死亡”的情形。这时也不准备生成更多的 PNG 图像，而只是用水平镜像（翻折）图像表示坏蛋“死亡”，这时就要用到 getFlippedImage()方法。

游戏中每个坏蛋可以有 4 个不同的动画：

- 向左移动；
- 向右移动；
- 死亡，脸朝左；
- 死亡，脸朝右。

为此，需要新型的幽灵对象，在改变方向或死亡时改变基础动画。

清单 5.7 的 Creature 类可以满足这个要求。除了 4 个不同动画之外，它还跟踪游戏者状态：STATE_NORMAL，STATE_DYING 或 STATE_DEAD。Creature 从 dying 状态转为 dead 状态需要一秒钟。

清单 5.7　Creature.java

```
package com.brackeen.javagamebook.tilegame.sprites;

import java.lang.reflect.Constructor;
import com.brackeen.javagamebook.graphics.*;

/**
```

```
    Creature 幽灵受重力影响，可能死亡。它有四个动画：向左移动、向右移动、死亡，脸朝左、
    死亡，脸朝右
*/
public abstract class Creature extends Sprite {

    /**
        从 STATE_DYING 到 STATE_DEAD 的时间量
    */
    private static final int DIE_TIME = 1000;

    public static final int STATE_NORMAL = 0;
    public static final int STATE_DYING = 1;
    public static final int STATE_DEAD = 2;

    private Animation left;
    private Animation right;
    private Animation deadLeft;
    private Animation deadRight;
    private int state;
    private long stateTime;

    /**
        用指定动画生成新 Creature
    */
    public Creature(Animation left, Animation right,
        Animation deadLeft, Animation deadRight)
    {
        super(right);
        this.left = left;
        this.right = right;
        this.deadLeft = deadLeft;
        this.deadRight = deadRight;
        state = STATE_NORMAL;
    }

    public Object clone() {
        // 用反射生成正确的子类
        Constructor constructor = getClass().getConstructors()[0];
        try {
```

```
            return constructor.newInstance(new Object[] {
                (Animation)left.clone(),
                (Animation)right.clone(),
                (Animation)deadLeft.clone(),
                (Animation)deadRight.clone()
            });
        }
        catch (Exception ex) {
            // 不应发生
            ex.printStackTrace();
            return null;
        }
    }

    /**
        取得这个 Creature 的最高速度
    */
    public float getMaxSpeed() {
        return 0;
    }

    /**
        在屏幕上首次出现 Creature 时唤醒生命。通常，生命首先向左移动
    */
    public void wakeUp() {
        if (getState() == STATE_NORMAL && getVelocityX() == 0) {
            setVelocityX(-getMaxSpeed());
        }
    }

    /**
        取得这个 Creature 的状态，状态为 STATE_NORMAL、STATE_DYING 或 STATE_DEAD
    */
    public int getState() {
        return state;
    }

    /**
        设置这个 Creature 的状态，状态为 STATE_NORMAL、STATE_DYING 或 STATE_DEAD
```

```
*/
public void setState(int state) {
    if (this.state != state) {
        this.state = state;
        stateTime = 0;
        if (state == STATE_DYING) {
            setVelocityX(0);
            setVelocityY(0);
        }
    }
}

/**
    检查这个生物是否还活着
*/
public boolean isAlive() {
    return (state == STATE_NORMAL);
}

/**
    检查这个生物是否在飞
*/
public boolean isFlying() {
    return false;
}

/**
    生物与地砖水平碰撞时在 update() 之前调用
*/
public void collideHorizontal() {
    setVelocityX(-getVelocityX());
}

/**
    生物与地砖垂直碰撞时在 update() 之前调用
*/
public void collideVertical() {
    setVelocityY(0);
```

```
    }

    /**
        更新这个生物的动画
    */
    public void update(long elapsedTime) {
        // 选择正确的动画
        Animation newAnim = anim;
        if (getVelocityX() < 0) {
            newAnim = left;
        }
        else if (getVelocityX() > 0) {
            newAnim = right;
        }
        if (state == STATE_DYING && newAnim == left) {
            newAnim = deadLeft;
        }
        else if (state == STATE_DYING && newAnim == right) {
            newAnim = deadRight;
        }

        // 更新动画
        if (anim != newAnim) {
            anim = newAnim;
            anim.start();
        }
        else {
            anim.update(elapsedTime);
        }

        // 更新为“死亡”状态
        stateTime += elapsedTime;
        if (state == STATE_DYING && stateTime >= DIE_TIME) {
            setState(STATE_DEAD);
        }
    }
}
```

Creature 类中有几个方法可以在 Sprite 类中运用，以增加 Sprite 类的功能。

- wakeUp()方法。当坏蛋在屏幕上首次出现时调用。这里，它调用 setVelocityX

(-getMaxSpeed())方法，使坏蛋开始移动，坏蛋要等你第一次见到时才开始移动。

- isAlive()和 isFlying()方法。它们是个方便方法，检查坏蛋的状态。死的坏蛋不会伤害游戏者，而且重力不作用于正在飞的坏蛋。
- 最后方法 collideVertical()和 collideHorizontal()。在坏蛋碰上地砖时调用。垂直碰撞时，将坏蛋的垂直速度设置为 0。水平碰撞时，将坏蛋的移动方向改变。这样坏蛋只有这么多智能。

注意: Creature 是个抽象类，因此要派生子类才能使用。生成的第一个子类是清单 5.8 的 Grub。

清单 5.8　Grub.java

```
package com.brackeen.javagamebook.tilegame.sprites;

import com.brackeen.javagamebook.graphics.Animation;

/**
    Grub 是在地上慢慢爬的生物
*/
public class Grub extends Creature {

    public Grub(Animation left, Animation right,
        Animation deadLeft, Animation deadRight)
    {
        super(left, right, deadLeft, deadRight);
    }

    public float getMaxSpeed() {
        return 0.05f;
    }

}
```

Grub 类的代码很难读懂，所以要仔细认清。其实它只是覆盖 getMaxSpeed()方法，将蚧蟾的速度设置为低值。

下面要生成的坏蛋是苍蝇，如清单 5.9 所示。

清单 5.9　Fly.java

```
package com.brackeen.javagamebook.tilegame.sprites;

import com.brackeen.javagamebook.graphics.Animation;
```

```
/**
    Fly 是在空中慢慢飞的生物
*/
public class Fly extends Creature {

    public Fly(Animation left, Animation right,
        Animation deadLeft, Animation deadRight)
    {
        super(left, right, deadLeft, deadRight);
    }

    public float getMaxSpeed() {
        return 0.2f;
    }

    public boolean isFlying() {
        return isAlive();
    }
}
```

和 Grub 类一样，Fly 类也是 Creature 的子类。除了定义飞行速度外，只要苍蝇是活的，isFlying()方法就覆盖并返回 true。这样重力不作用于正飞的坏蛋，只在其死亡时才起作用。

Creature 还有一个子类，就是清单 5.10 所示的 Player 类。尽管游戏者不是坏蛋，但也需要 Creature 的特性，如多个动画和状态。

清单 5.10　Player.java

```
package com.brackeen.javagamebook.tilegame.sprites;

import com.brackeen.javagamebook.graphics.Animation;

/**
    游戏者
*/
public class Player extends Creature {

    private static final float JUMP_SPEED = -.95f;

    private boolean onGround;

    public Player(Animation left, Animation right,
```

```
        Animation deadLeft, Animation deadRight)
    {
        super(left, right, deadLeft, deadRight);
    }

    public void collideHorizontal() {
        setVelocityX(0);
    }

    public void collideVertical() {
        // 检查是否碰到地
        if (getVelocityY() > 0) {
            onGround = true;
        }
        setVelocityY(0);
    }

    public void setY(float y) {
        //检查是否摔倒
        if (Math.round(y) > Math.round(getY())) {
            onGround = false;
        }
        super.setY(y);
    }

    public void wakeUp() {
        //什么也不干
    }

    /**
        游戏者在地上或 forceJump 设置为真值时让游戏者跳动
    */
    public void jump(boolean forceJump) {
        if (onGround || forceJump) {
            onGround = false;
            setVelocityY(JUMP_SPEED);
        }
    }
```

```
        public float getMaxSpeed() {
            return 0.5f;
        }
    }
```

简单地说，Player 类在 Creature 中增加的唯一特性就是跳动功能。通常，如果游戏者不在地上，就不能让它跳动了。Creature()方法跟踪游戏者是否在地上，如果游戏者在地上，则可以跳动。也可以向 setY()与 collideVertical()强制游戏者在半空中跳动。

5.2 碰撞探测

前面曾介绍过，要保证游戏者（和坏蛋）不会穿墙而过，但可以在平台上跳动。简而言之，每次移动游戏者时，就要检查是否碰到任何地砖上，若是则要相应调整位置。由于使用地砖地图，所以很容易实现碰撞探测技术。

下面把碰撞探测过程分解成几个部分：探测地砖碰撞和纠正幽灵位置以避免碰撞。

5.2.1 探测碰撞

理论上，幽灵可能一下移过几个地砖，可能同时在 4 个不同地砖上。所以要检查幽灵当前所在的每个地砖和幽灵准备进入的地砖。

如清单 5.11 的 getTileCollision()方法就可以完成这个任务。它检查幽灵从原位到达新位时是否经过任何实物地砖，若不则返回幽灵碰上的地砖位置，否则返回 null。

清单 5.11 getTileCollision()方法

```
point pointCache = new Point();

...
/**
    如果发现碰撞，则返回幽灵碰上的地砖位置，否则返回 null
*/
public Point getTileCollision(Sprite sprite,
            float newX,float newY)
{
float fromX = Math.min(Sprite.getX(),newX);
float fromY = Math.min(Sprite.getY(),newY);
float toX = Math.min(Sprite.getX(),newX);
float toY = Math.min(Sprite.getY(),newY);
//取得地砖位置
int fromTileX = TileMapRenderer.pixelsToTiles(fromX)
int fromTileY = TileMapRenderer.pixelsToTiles(fromY)
int tlTileX = TileMapRenderer.pixelsToTiles(
```

```
        toX + sprite.getWidth()  - 1);
int tlTileY = TileMapRenderer.pixelsToTiles(
        toY + sprite.getHeight()  - 1);
//检查每个地砖是否碰上
for (int x = fromTileX;x<toTileX;x++){
for (int y = fromTileY;y<toTileX;y++){
        if (x< 0 || x>= map. getWidth()    ||
            map. GetTile(x, y) != null)
        {
            //发现碰撞返回地砖
            pointCache.setLocation(x,y);
            return pointCache
          }
      }
    }
  //未发现碰撞
  return     null
}
```

另外，getTileCollision()方法还把移到地图左右边沿也外看成碰撞，以保证幽灵位于屏幕内。

如果帧之间幽灵经过多个地砖，则这个方法并不完善（帧间经常发生的情形），在第 11 章“碰撞探测”将实现更好的幽灵。

环境碰撞方法，但这个代码适用于大多数情形。

5.2.2　处理碰撞

下面要在探测碰撞之后纠正幽灵位置以避免碰撞，如图 5-8 所示，幽灵碰撞后将向下获向右移动。

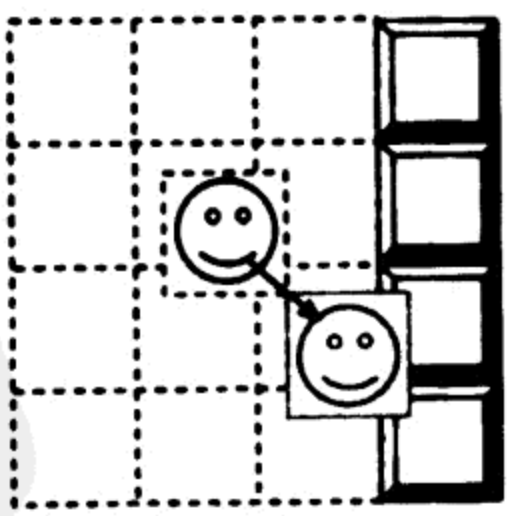


图 5-8　幽灵向下向右移动太远时会撞墙

游戏中，幽灵将沿着对角线移动，若同时碰上 2 个幽灵将如何呢。从图上看好像很容易解决，只要将幽灵放到左边，但怎么算出是向左而不是向上、向下或向右呢？

要进行这种计算，就要把幽灵的运动分为 2 个部分：水平移动和垂直移动。首先要水平移动幽灵，如图 5-9 所示。

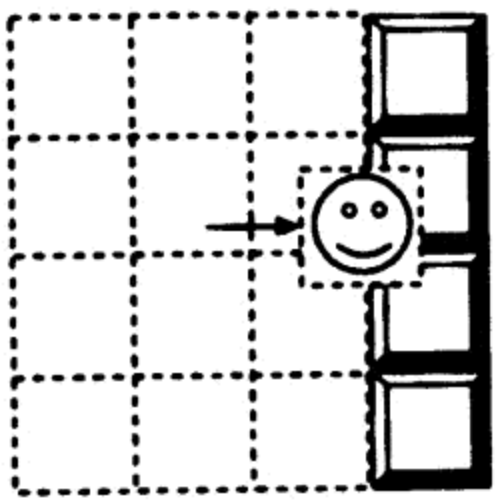


图 5-9　首先要水平移动幽灵，探测碰撞

实例中发现会碰到砖。要纠正这个现象，只要沿运动方向的相反方向移动幽灵，将幽灵与地砖边沿对齐，如图 5-10 所示。

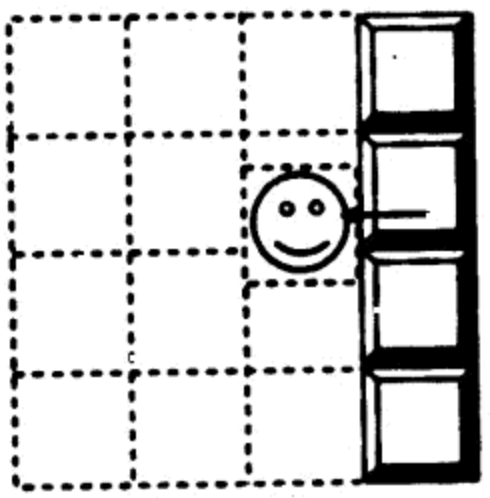


图 5-10　然后要修正幽灵水平位置，避免碰到砖

现在游戏者已经水平移动，其位置已经修正，不会碰到砖。然后用相同方法处理垂直移动，如图 5-11 所示。

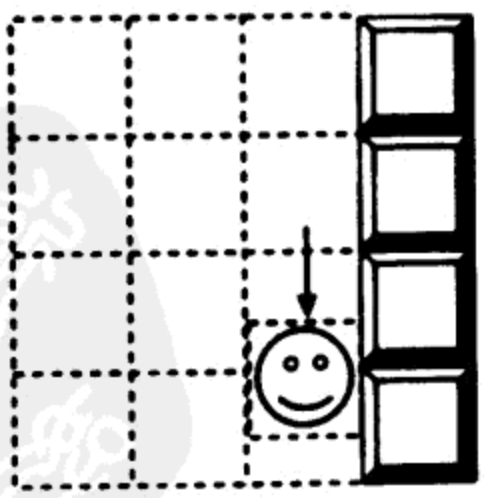


图 5-11　然后用相同方法处理垂直移动，这里没有碰撞

实例中，垂直移动时没有碰撞时，所以完成任务。

这些功能在清单 5.12 所示的 updateCreature()方法中实现，要移动幽灵的水平位置和垂直位置，并在此过程中检查和纠正碰撞。

清单 5.12　updateCreature()方法

```
private void updateCreature(Creature creature, long elapsedTime){

    //采用重力
    if (! creature.isFlying()){
        creature.setVelocityY(creature.getVelocityY) +
                GRAVITY * elapsedTime);
        }
        //改变 x
        float dx = creature.getVelocityX();
        float oldX = creature.getX();
        float newX = oldX + dx * elapsedTime;
        Point tile =
         GetTileCollision(creature, newX, creature .getY());
         if(tile == null){
                creature.setX(newX);
        }
        else{
            //与地砖边界对齐
             if (dx>0) {
                creature.setX(
        TileMapRenderer.tilesToPixels(tile.x) -
        creature.getWidth());
          }
          else if (dx<0){
                creature.setX(
          TileMapRenderer.tilesToPixels(tile.x + 1));
         }
         creature.collideHorizontal();
}
                //改变 y
float dy = creature.getVelocityY();
float oldY = creature.getY();
float newY = oldY + dy * elapsedTime;
tile = getTileCollision(creature, creature.getX(),newY);
```

```
if (tile ==null){
        creature. SetY(newY);
}
else{
   //与地砖边界对齐
     if (dy>0){
       creature.setY(
                    TileMapRenderer.tilesToPixels(tile.y) -
                    creature.getHeight());
          }
          else if (dy<0){
        creature.setY(
             TileMapRenderer.tilesToPixels(tile.y + 1));
        }
        creature.collideVertical();
          }
      }
```

udateCreature()方法还对不飞的生物采用重力。重力总是影响幽灵的，但如果生物站在砖上，则看不到这个效果。

检查和纠正碰撞时，调用生物的 collideVertical()或 collideHorizontal()方法。通常，这些方法改变或停止生物的速度，使碰撞不会在近期内发生。

这样就可以了，这时游戏者和坏蛋可以在地图上漫游，而不会穿墙而过。

5.2.3 幽灵碰撞

然后要探测游戏者何时碰上其他幽灵，如充电器和坏蛋。在游戏中，忽略生物之间的碰撞，只探测与游戏者的碰撞，其实就是观察游戏者的幽灵边界是否与其他幽灵边界相交，如清单 5.13 的 isCollision()方法所示。

清单 5.13 isCollision()方法

```
public boolean isCollision(Sprite s1, Sprite s2)
   //如果 Sprites 相同，则返回 false
   if (s1 ==s2){
        return false;
   }
   //如果一个 Sprite 是死物，则返回 false
   if (s1 instanceof  Creature &&!(( Creature)s1).isAlive()){
         return false;
   if (s2 instanceof  Creature &&!(( Creature)s2).isAlive()){
         return false;
```

```
    }

    //取得幽灵的象素位置
     int s1x = Math.round(s1.getX());
     int s1y = Math.round(s1.getY());
     int s2x = Math.round(s2.getX());
     int s1y = Math.round(s1.getY());

    //检查两个幽灵的边界是否相交
     return (s1x < s2x + s1.getWidth() &&
                s1y < s2y + s2.getHeight() &&
                s2y < s1y + s1.getHeight());
  }
```

由于 TileMap 的清单中包含所有幽灵，因此只要遍历清单，检查每个幽灵是否与游戏者碰撞，见清单 5.14。

清单 5.14　getSpriteCollision 方法

```
public Sprite getSpriteCollision(Sprite sprite)

    //遍历幽灵清单
    Iterator i = map.getSprites();
    While (i.hasNext()){
        Sprite otherSprite = (Sprite)i.next();
        If (isCollision)sprite,otherSprite)){
            //发现碰撞，返回幽灵
            return otherSprite;
        }

     //未发现碰撞
    return null;
}
```

发现碰撞时，可以进行所在的处理。如果幽灵是充电器，则可以给游戏者增加生命力、播放声音或进行与该充电器相关的任何操作。

如果幽灵是坏蛋，则可以杀死坏蛋或杀死游戏者。这时，如果游戏者跳动或掉到生物上，即游戏者的垂直移动增加，则杀死坏蛋。

```
boolean cankill = (oldY < player.getY());
```

否则杀死游戏者，如游戏者在散步时碰上坏蛋时。

今后可能要增加其他杀死坏蛋的方法，如“不可战胜”的充电器或把坏蛋推到悬崖外面。

5.3 完成与加速

这样就有了游戏的各种要素。当然，游戏中还有键盘输入、声音、音乐等基本内容，但这些功能在其他章介绍的类中。除了本章已经列出的类之外，还有 3 个类：

- GameManager 处理键盘输入、幽灵更新、碰撞探测和播放声音与音乐；
- TileMapRenderer 绘制地砖地图、视差背景与幽灵；
- ResourceManager 装入图像、生成动画与幽灵和装入关卡。

别担心，Web 站点中提供了这些类的源代码，以及本章相关的代码。

为了提高游戏的帧速率，GameCore 类作了 2 处改变：

- 首先选择 16 位颜色方式，提高游戏的帧速率；
- 游戏中删除了休眠 20 毫秒的代码，使游戏利用全部处理器功能。

5.4 生成可执行的 .jar 文件

最后，当把游戏提供给朋友时，还要告诉他运行代码的指令。如让编程人员在命令行输入下列合适的内容。

java com.bracken.javagamebook.tilegame.GameManager

但普通用户可能根本不知道什么是命令行，怎么解决遇到的问题。更好的方法是建立可执行.java 文件。利用可执行.java 文件，用户只要双击这个文件就可以启动游戏，编程人员也可以因此大大节省时间。

可执行.java 文件是个 Java 档案文件，是一组类的容器。游戏中的所有类都存放在可执行.java 文件中，通常以压缩格式出现。

为了使.java 文件变成可执行文件，就要在.jar 的 manifest 文件中指定运行哪个类。这个文件在.java 中称为 META_INE/MANIEEST.MF。在这个平台游戏中，.jar 的 manifest 文件如下。

Manifest-Version: 1.0

Main-Class: com.bracken.javagamebook.tilegame.GameManager

双击.java 文件时，Java 虚拟机启动并寻找.jar 的 manifest 文件。如果找到 Main-Class 属性，则运行指定的类。如果没有 manifest 文件或找不到 Main-Class 属性，则 Java 虚拟机就像什么也没有发生，悄悄退出。

如果用 Java 工具生成.jar 文件，则首先要在文本编辑器中生成 manifest 文件，然后用一行选项将这个 manifest 文件加进.java 文件中。

如果用 Ant 建立.java 文件，则 Ant 自动生成 manifest 文件并将其加进.java 文件。清单 5.15 显示的样本 Ant 目标生成游戏的可执行.java 文件。

清单 5.15 样本 Ant 目标生成游戏可执行的 .jar 文件

```
<target name= "build" depends= "compile" >
  <jar jarfile={basedir}/tilegame.jar" >
    <manifest>
```

```
    <attribute name= "Main-Class"
      value= "com.bracken.javagamebook.tikegame.GameManager." />
  </manifest>
      <fileset dir= "{destdir}" imcludes= "**/*.class" />
      </jar>
  </target>
```

manifest 文件中还可以包含各种其他的数据。Java SDK 文档的 Jar Guide 中提供所有的信息，关于 Ant 的更多信息，也可详见 ant.apache.org。

但要注意，并不是每个人都能双击执行.jar 文件。Linux 用户就无法双击执行.java 文件，有些 Windows 机器用 WinZip 而不用 Java 运行环境打开.jar 文件。如果无法双击执行.jar 文件，则可以在控制台中运行下列命令。

```
java - jar tilegame.jar
```

第 18 章“游戏设计与最后 10%”详细介绍了部署游戏的其他方法，如使用 Java Web Start。

5.5　扩展游戏

地砖游戏并不完美，还可以进行各种改进，要解决如下几个问题：

- 在 Windows 机器上，系统定时器的精度不够，无法保证光顺滚动。第 16 章“优化技术”提供了一个解决方法；
- 用幽灵边界进行碰撞探测不够精确。可以改用更小的碰撞边界。第 11 章详细介绍了这个问题；
- 幽灵经过帧间的长距离时，还有更好的碰撞处理方法。第 11 章详细介绍了这个问题；
- 此外，增加关卡和增加地砖图像可以使游戏玩得更久，还可以使引擎更快更高，重力更弱，或进行其他调整。下面是其他一些改进：
- 增加交互式地砖，例如可以打破地砖或开门。这样可以在游戏中增加谜题元素；
- 增加有图无实的“经过”地砖，用于地图装饰；
- 增加多个视差滚动背景，如山和树的透明背景；
- 游戏者跳动时使用不同的幽灵动画；
- 增加岩石滚落和无底地坑，使环境更有破坏力；
- 让游戏者改变弹跳高度与跑动加速，按跳动键的时间越久，就跳得越高；同样，游戏者移动时间越长，速度就越快；
- 增加生物之间的碰撞，使坏蛋仍碰在一起，而不是互相穿过；
- 在苍蝇飞得很近时使用三维嗡嗡声，或让所有声音像山谷中一样回响；
- 提供更好的地图间过渡，如汇总上一地图中收集的星星和在下一幅地图之前出现“准备”屏幕；
- 增加更多标准游戏选项，如暂停、键配置和菜单；

- 增加更多坏蛋、更多图形、更多关卡等。

5.6 小结

我们才等到第五章，已经可以生成有趣的二维平台游戏，具有地砖地图、碰撞探测、坏蛋、充电器和视差滚动。

第 6 章　多人游戏

关键主题

- Java I/O 库的变革
- ChatterBox，基本多方游戏应用程序
- 多方游戏服务器框架
- RPS 游戏（石头、布、剪刀）
- 建立框架
- 服务器管理
- 高级课题
- 小结

多人电脑游戏已经有几十年的历史了。早在 70 年代的远程计算环境中，黑客们就在基于 DEC VAX，UNIX 和 VMS 等操作系统的小型计算机上编写简单的网络游戏。80 年代，多用户地龙（MUDs）游戏非常普及，当时的联机服务商（CompuServe,Prodigy 和 GEnie）都提供了自己的网络游戏。

在 Internet 出现之前，游戏用户仍然很少。因为大多数家庭没有连接 Internet，甚至连电脑都没有。1993 年，Id Software 公司发布了第一款基于网络第一人称视角射击游戏 Doom。Doom 可以使 4 位游戏者在 LAN（局域网）上相互射击，或 2 位游戏者通过 modem 在 Internet 网进行游戏。Doom 的出现标志着新一代高速多人游戏的开始。随 Doom 之后的 Quake 系列游戏，使局域网上可以有更多人的参与，并出现了许多专门提供低锁存游戏服务器的网络公司，但目前大部分已经倒闭了。

随着 Internet 的发展，人们需要能让更多人参加的游戏，而且游戏不单单基于在 LAN 上。新一代游戏主要是基于 Web 的下载游戏，不过速度会稍慢一些。90 年代后期，出现了这类游戏的专业公司，它们提供基本店面游戏（围棋、象棋、扫雷、21 点牌）、文字、谜题游戏和一些原创游戏。尽管游戏属于回合制（比 FPS 游戏稍慢），但 Pogo,Yahoo! Games 和 MSN Gaming Zone 等站点的游戏迷非常多，常常是几千人在线。

90 年代的第一支主力是大量多方联机的角色扮演游戏（MMORPG），最初是 1996 年 3DO 公司推出的 Meridian 59，但是很快被 Ear Ultima Online 和 Sony 公司的 Everquest 所代替。这些游戏都支持更多同时的用户，提供丰富的虚拟世界，带来了全新的气象。

每一代游戏各有不同的设计难点。本章将介绍一般性多方游戏的服务器和客户机的设计，并以 JDK 1.4 NIO 来实现。实例中，介绍如何建立灵活的多方游戏服务器框架，并介绍一些高级课题。最后，建立很好的知识框架，并得到很好的代码基础，从而开始建立自己的多方游戏服务器。

本章和前面几章有所不同，这里没有漂亮的图形，没有悦耳的声音效果。这里最让人激动的就是服务器日志文件中看到下列消息。

New connection from 10.0.0.1, 483 users online

因此，如果你对套接字、登录、线程池包装和远程管理控制台感兴趣，就请往下读吧。

假设读者已经了解命令 Shell，可以是 bash,tcsh 或 Windows shell(cmd)。注意，如果使用 Windows，则 Cygwin(见 www.cygwin.com)可以简化工作，因为它提供了 bash shell 和一组标准命令行实用程序。

服务器应用程序是悄悄运行的，通过日志文件提供输出，而没有应用程序 GUI。所以要学会用 grep 和 tail 等工具浏览和搜索日志。

例如，要浏览日志文件的最后 50 行，请输入如下命令。

```
tail - n 50 server.log
```

要连续监查活动日志文件，请用-f 开关。

```
tail - f server.log
```

GNU tail 命令的最新版本有个-F 开关，即使替换基础文件时也能监查相同日志文件名。这对监查经常换盘的日志非常有用。

grep 工具可以寻找文本文件中符合指定模式的行。将 GNU tail 和 grep 用管道(|)结合起来，就得到分析日志文件的强大工具。假设只要显示表示、客户机切断的日志消息，则可以使用如下命令。

```
tail - F server.log | grep disconnect
```

如果分析旧日志，则可以将 grep 和一个 pager 命令组合，如使用 less 命令如下。

```
grep disconnect server.log | less
```

这些命令的手册页详细介绍了它们的用法和选项。

此外，熟悉 netstat 等网络诊断工具和 ethereal 等分组窃听器也有助于调试服务器操作和客户机/服务器的通信。

6.1 Java I/O 库的变革

网络服务器（特别是游戏服务器）要处理大量在线客户的请求，所以服务器应能够高效地读取和写入大量的消息。

在 JDK 1.4 发布之前，Java 没有处理非防止 I/O 操作的内部机制，所以服务器应用程序至少要连接客户机的一个线程，等待阻止套接的 read()方法。

这样实现的缺点是显而易见的，操作系统的线程数受到政策和调度程序性能的限制，每线程都有基本堆栈会对内存开销，线程的生成也需要时间，情境切换过程中保存和恢复线程状态也需要开销。这些操作都要减少线程数，如果要同时为成百上千的客户服务，则显然不能一个连接客户机一个线程。

在 Sun 公司面对 Java 用户的强大压力下，JDK1.4 引入了新的 I/O 系统——New I/O（NIO）API，它放在 java.nio 包及其子包 java.nio.channels 和 java.nio.charset 中。

6.1.1 JDK 1.4 NIO 库的概述

Java 的 NIO 包括几个新类，它们专门提供高性能的 I/O 操作，是继承 C 语言和其他语言中有用的部分。除了新类之外，几个原先的网络类（如 java.net.Socket）也针对新 API 进行了更新。

Java 的 NIO 类分成几大类：信道、缓存和选择器。读者一定要熟悉这些类，因为这是应用程序的基础。下面将概要的介绍它们，然后举例说明在应用程序中的应用。

6.1.2 信道

信道（Channels）是操作系统资源的抽象表示，如文件、管道和最关心的网络套接。信道和缓存代替旧版 JDK 中 Java 的 Stream 类，能够处理网络和文件资源。

1. 接口

所有信道类的根是 java.nio.channel.Channel 接口，它定义 2 个方法：

- public void close();
- public boolean isOpen()。

将 Channel 接口扩展出几个子接口，其中就有 ReadableByteChannel 接口和 WritableByteChannel 接口,它们也添加了 2 个方法：

- public int read(ByteBuffer dst);
- public int write(ByteBuffer src)。

其中，第一个方法从信道中读取数据并指定到缓冲区。第二个方法将写入信道中的数据指定到缓冲区中。

ByteChannel 接口只是把 ReadableByteChannel 和 WritableByteChannel 合并为一个接口。

GatheringByteChannel 和 ScatteringByteChannel 接口增加了在一个操作中读取和写入多个缓冲区的功能。这适用于将消息的头和荷载读入 2 个不同的缓冲区中，如读取 HTTP POST 响应，则可以把 HTTP 头读入一个缓冲区，把 POST 数据读入另一个缓冲区，实现代码如下所示。

```
long write(ByteBuffer[] srcs);
long write(ByteBuffer[] srcs, long offset, long length);
long read(ByteBuffer[] dsts);
long read(ByteBuffer[] dsts, long offset, long length);
```

其中 write 方法的返回值是实际写入的字节数，可能为 0。read 方法返回读取的字节数，或者返回-1 表示流末尾。许多情况下，终止套接连接时，返回-1 是第一个失败指标。

2. 类

下面简要介绍 java.nio.channels 包中和服务器编程有关的类，其他类的细节和描述请见 API 文档。

（1）ServerSocketChannel。

此类代替（并包装下方的）java.net.ServerSocket 类，能够接收客户端的网络套接。

若要生成新的 ServerSocketChannel 对象，只要调用静态 open()方法。但要想让新对象操作，则先配置它的阻止方式，然后将其关联到一个地址或端口上。

SelectableChannel 对象可以选择阻止或非阻止方式。若设置阻止方式，则信道上所有的 I/O 操作将受阻，等待其完成。若设置非阻止方式，I/O 操作将立即返回，可能不实际进行任何 I/O。ServerSocketChannel 缺省为阻止方式，这样 ServerSocketChann el.accept()方法和旧的 ServerSocket.accept()方法一样，阻止当前线程，直到建立连接。

下面实例将 ServerSocketChannel 对象设置为非阻止方式。

```
// 打开 ServerSocketChannel 并配置非阻止方式
ServerSocketChannel   sSockChan;
sSockChan = ServerSocketChannel.open();
sSockChan.configureBlocking(false);

//将 localhost 关联到指定端口
InetAddress addr = InetAddress.getLocalHost();
sSockChan.socket() .bind(new InetSocketAddress(addr, PORT));
```

后面的 ChatterBox 样本应用程序中要介绍如何将信道注册到选择器上和开始接收连接。

要访问 ServerSocketChannel 的基础 ServerSocket 对象，可以用 socket()方法，实现方法如下。

```
public ServerSocket socket();
```

另一个值得注意的方法是 validOps（），它是从 SelectableChannel 继承，来返回信道注册的操作。对于 ServerSocketChannel 对象，方法总是返回 SelectionKey.OP_ACCEPT。关于套接操作的进一步介绍，详见 java.nio.channels.SelectionKey API 文档。

（2）SocketChannel。

SocketChannel 表示与服务器的没一个客户套接的连接。和 ServerSocketChannel 相似，SocketChannel 代替（并包装下方的）java.net.Socket 对象。

说明：SocketChannel 和 ServerSocketChannel 只用于 TCP 连接，UDP 连接使用下节介绍的 DatagramChannel 类。

SocketChannel 具有 ServerSocketChannel 的所有方法，此外还增加了读取和写入方法，以及管理套接连接的方法。在服务器上，收到 ServerSocketChannel 的连接时打开套接，在客户端，可以用 open(SocketAddress address)方法或 open()方法加上 connect(SocketAddress address)调用。

本章稍后将会介绍客户机和服务器之间的所有读取和写入操作均选用 SocketChannel。

（3）DatagramChannel。

此类用 UDP（用户数据报文协议）而不是 TCP 发送和接收数据。

UDP 在 RFC 768(www.faqs.org/ftp/rfc/rfc768.txt)中定义。与 TCP 不同的是，UDP 是个不可靠协议，即不能保证发送的分组实际到达目的地，也不能保证发送的分组按发送顺序到达目的地。

本章主要介绍 TCP，因为它更容易处理，而且大多数游戏消息都要求可靠送达，使游

戏正确工作。但是，UDP 也有相应的地位。在需要迅速发送大量事件而不管其是否全部送达，此种情况下将非常有用。例如，在多方 FPS 游戏中，要向各方连续传递每个游戏者的位置。在这种任务中，TCP 传送会造成的延迟是无法接受的，特别是在 Internet 网络条件下，路由的分组经常丢失，需要重传。但利用 UDP，则可以发出大量数据，服务器收到顺序不对的分组时，只要把它丢进位桶中，然后继续，因为游戏者已经移到新位置。此外，UDP 分组比 TCP 分组小得多，因此可以更有效地利用 UDP 带宽。

6.1.3 缓存

缓存（buffer）是有限的内存块，用于临时存储基本类型的数据，一般用缓存存储移进和移出的 SocketChannels 数据。

1. 类

java.nio 包中有几个缓存类，用于不同的 Java 基本数据类型：

- ByteBuffer；
- CharBuffer；
- DoubleBuffer(不要与第 2 章“二维图形与动画”介绍的双缓存混起来)；
- FloatBuffer；
- IntBuffer；
- LongBuffer；
- ShortBuffer。

此外，还有一个内存贴图文件的特殊缓存类 MappedByteBuffer，用于处理大文件。FileChannel 类的 map()方法可以将缓存连接到文件。

所有缓存类属于 java.nio 包，是从抽象类 Buffer 派生出的。Buffer 类定义了查询与操纵缓存容量、位置、极限与标志的方法，下面就来介绍：

- 容量是可以包含的指定基本数据类型元素个数，可以在生成时用子类的 allocate()方法设置。后面介绍 ByteBuffers 细节时会介绍缓存分配；
- 极限是读取和写入缓存时的边界。读取时，它是第一个不能读取的元素的索引；写入时，它是第一个不能写入的元素的索引；
- 位置是缓存中读取和写入下一元素的当前位置；
- 标志在调用 reset()方法时将位置移回设置的点。

图 6-1 显示了这些指标，缓存处于随机有效状态。

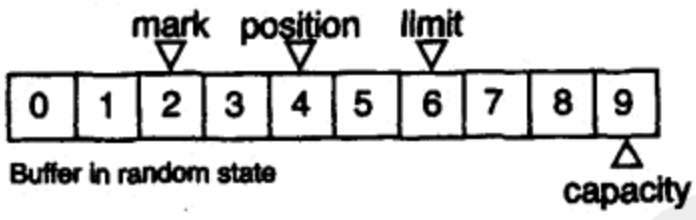


图 6-1 缓存容量、位置、极限与标志

（1）填充和移出。

要填充缓冲区，可以从信道读取，也可以使用子类中定义的各种 put()方法，如要在文

件的偏离量 0 处读取 length 个字节，并放进 ByteBuffers 中，则可以用 FileChannel.read(ByteBuffer src, int o,int length)方法实现。

要移出缓存，可以将缓存写入信道，也可以使用子类中定义的各种 get()方法，如要从 ByteBuffers 写入 SocketChannel 则可以用 SocketChannel.write(ByteBuffer src)方法实现。

记住，缓存的容量是固定的，所以填充和移出时并不实际加长缩短，只是改变上述指标（位置、极限与标志）而已。

（2）操纵缓存。

操纵缓存时要熟悉几个关键方法：

- clear();
- flip();
- rewind();
- compact()。

要准备填充缓冲区，可以调用 clear()方法，将位置设置为 0,将极限设置为容量，如图 6-2。这样调用 clear()方法之后，缓冲区可以从信道读取或 put()操作接收容量指定的元素。

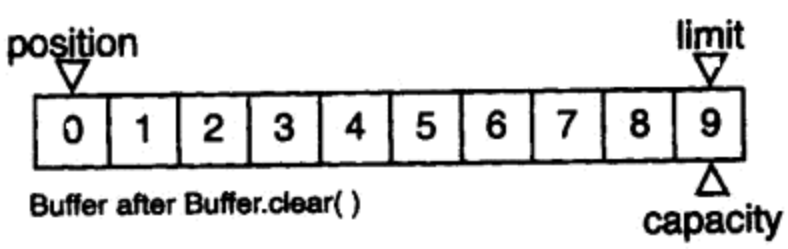


图 6-2　缓冲区调用 Buffer.clear()之后

缓冲区满后（或包含所要数据量），可以调用 flip()方法，准备用 get()方法或信道 write()操作移出缓冲区的内容。flip()方法将极限设置为当前位置，将位置设置为 0，图 6-3 显示了数据用 put()方法或信道 read()操作放进缓存，以及调用 flip()方法后的情形。

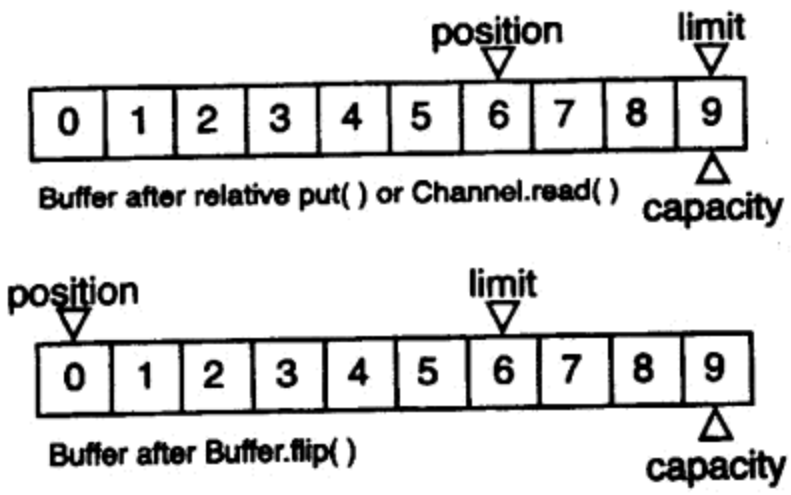


图 6-3　缓存调用 Buffer.flip()方法后的情形

Buffer.rewind()方法只是将位置设置为 0，让缓存准备再次移出，假设已经翻页。例如，可以用这个方法将数据同时写入套接和文件或写入多个客户信道，如图 6-4 描述了这种情形。

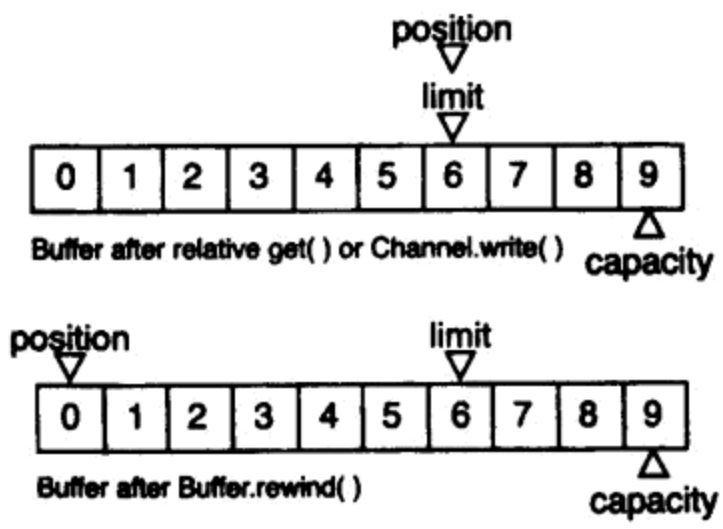


图 6-4　缓存调用 Buffer.rewind()方法后的情形

compact()方法将当前位置和极限之间的元素移到缓冲区开头。移动之后的位置在数据末尾左边，极限设置为容量。稍后要在服务器中使用这个方法，读者应加以注意。

（3）直接和非直接。

Java 缓存的最后一个问题是直接与非直接。直接缓存分配自然数据结构，可以在套接等自然资源间传输数据，而不需将数据复制到 Java 虚拟机内的数据结构中，从而对性能有巨大好处。

同样，分配与自然资源不交互的缓冲区时，就要使用非直接缓存。除了用 allocate()和 allocateDirect()方法分配外，使用直接与非直接缓存时 API 是一样的。

6.1.4　选择器与 SelectionKeys

NIO 库的主要特性之一是能够对套接进行非阻止读取和写入操作。但是，为了利用这个好处，就要按组织方式处理多个连接，其中的关键就是选择器，它提供信道多路复用访问机制。

多路复用（Multiplexing）方法将多个通信信号合并为一个数据流，按频率或时间域让信号交织起来。对于信道和选择器，选择器监听所有注册信道发来的数据，并通过 select()方法的结果串行显示。这样，只要和一个 Selector 对象交流，就可以判断任何信道中何时有数据。

信道首先用 SelectableChannel.register()方法向选择器注册，然后选择器要监视一系列操作，包括 OP_ACCEPT, OP_CONNECT, OP_READ 与 OP_WRITE，定义为 SelectionKeys 类的常量。

说明：ServerSocketChannel 与客户端 SocketChannelS 可以使用单一选择器，监听新连接和有数据的信道。但实际上，为了设计得更清晰，通常要把两者分开。

检查注册信道上的活动时，调用选择器的一个选择方法，select()方法有 3 个版本：

- select()。阻止并直到兴趣集中至少一个信道有活动。兴趣集是选择器对信道监视的一组操作（OP_ACCEPT, OP_CONNECT, OP_READ 与 OP_WRITE）；
- select(long timeout)。与 select()相同，但只阻止指定的最大超时毫秒数；
- selectNow()。立即返回，不管有多少个信道准备（可能是 0）。

调用选择方法时，操作系统查询注册信道。信道上有活动时，其 SelectionKeys 加进选

择器的选择键集合中。选择方法都返回一个整数，表示选择键个数，0 表示没有一个。

选择调用之后，可以用 Selector,selectedKeys()方法返回准备的选择键，其返回一个集合。SelectionKeys 中不仅保存链接的信道，而且提供一个重要特性。每个 SelectionKeys 可以保存任何附件对象，然后用附件存储每个信道中部分读取的消息，但也可能找到其他用途。附件可以在信道向选择器注册时提供，也可以后面用 SelectionKey.attach()方法提供，而且可以用 attachment()方法取得。

Selector 类的另一个重要方法是 wakeup()。wakeup()方法可以让选择器突破防止的选择调用。另一种方法是调用阻止线程的 Thread.interrupt()方法或调用选择器的 close()方法。

6.2 基本多方游戏应用程序

介绍众多基础知识后，下面要介绍一些实际代码。本节将建立客户机/服务器应用程序 ChatterBox。尽管程序很简单，但它演示了如何把前面介绍的内容综合成有用的程序。

6.2.1 服务器端

ChatterServer 是个多用户闲聊应用程序，可以让多个用户连接服务器，并且相互发送文本消息。服务器要完成如下主要功能：

- 接受客户机连接；
- 从客户机读消息；
- 将消息写入客户机；
- 处理切断（优雅与否）。

1. 初始设置

首先介绍一些初始设置，在 ChatterServer 构造函数中有如下的代码。

```
clients = new LinkedList();
writeBuffer = ByteBuffer.allocateDirect(255);
readBuffer = ByteBuffer.allocateDirect(255);
```

其中 LinkedList()跟踪连接的客户机。此外分配 2 个 ByteBuffers，一个从信道读取，一个写入信道，它们都分配直接缓冲区。由于在套接与缓冲区之间传递数据，所以 Java 虚拟机可以使用优化操作系统调用和避免在自然与 Java 代码边界之间传递数据。

服务器听取端口 10997 输入客户连接。端口 10997 的选择是任意的，可以将服务器设置成在任何端口上的通信，但最好注意“著名”端口号（见 www.iana.org/assignments/port-numbers）。端口在清单 6.1 所示的 initServerSocket（）方法中设置。

清单 6.1 初始化服务端

```
private static final int PORT = 10997;

private void initServerSocket() {
    try {
```

```
            sSockChan = ServerSocketChannel.open();
            sSockChan.configureBlocking(false);

            InetAddress addr = InetAddress.getLocalHost();
            sSockChan.socket().bind(new InetSocketAddress(addr, PORT));

        readSelector = Selector.open();
    }
    catch (Exception e) {
        log.error("error initializing server", e);
    }
    }
```

在 initServerSocket（）方法代码中，首先打开 ServerSocketChannel 和将其配置成非阻止操作。然后将其关联到 IP 地址和端口上。最后打开选择器，注册所有新的客户机信道。

2. 主循环

在初始设置后，就可以启动 run()方法了。为了方便 ChatterServer 采用单线程实现，因此所有操作在一个主循环中完成。run()方法只是重复调用 acceptNewConnections()和 readIncomingMessages()方法。

3. 接收连接

接收输入客户机连接很简单，只要在前面打开的 ServerSocketChannel 上调用 accept()方法即可。清单 6.2 显示了 ChatterServer 的 acceptNewConnections 方法。

清单 6.2 接受客户机连接

```
    private void acceptNewConnections() {
        try {
            SocketChannel clientChannel;
            while ((clientChannel = sSockChan.accept()) != null) {
                addNewClient(clientChannel);
                sendBroadcastMessage("login from: " +
                clientChannel.socket().getInetAddress(),
                clientChannel);
                sendMessage(clientChannel, "\n\nWelcome to ChatterBox,
                there are " + clients.size() + " users online.\n");
                sendMessage(clientChannel, "Type 'quit' to exit.\n");
            }
        }
        catch (IOException ioe) {
```

```
            log.warn("error during accept(): ", ioe);
        }
        catch (Exception e) {
            log.error("exception in acceptNewConnections()", e);
        }
    }

    private void addNewClient(SocketChannel chan) {
        clients.add(chan);
        try {
            chan.configureBlocking( false);
            SelectionKey readKey = chan.register(readSelector,
            SelectionKey.OP_READ, new StringBuffer());
    }
        catch (ClosedChannelException cce) {
        catch (IOException ioe) {
    }
```

由于 ServerSocketChannel 配置成非阻止操作，所以调用 accept()方法将立即返回，不管是否有等待的连接，这样就避免了阻止唯一的服务器线程。如果没有等待的连接，则方法迅速结束，否则它一直循环，直到处理所有新连接。对于每个连接，都调用清单 6.2 所示的 addNewClient()方法，进行一些处理工作。客户端信道配置成非阻止，向选择器注册。由于想知道信道何时会有可读的数据，所以要把 SelectionKey.OP_READ 作为兴趣集传入。最后一个参数是对信道指定的选择键的附件，这里用一个简单 StringBuffer 作为附件，放置这个客户机的部分读取消息。

程序最后还要向新客户机发送一个欢迎消息，并向所有客户机发送一个广播消息，告知新连接。

4. 写入消息

ChatterServer 的 sendMessage()方法将消息写入各个 ChatterClients，sendBroadcastMessage()方法将消息发送到所有客户机。清单 6.3 显示了实现方法和帮助方法 prepWriteBuffer()和 channelWrite()。

清单 6.3 写入消息

```
    private void sendMessage(SocketChannel channel, String mesg) {
        prepWriteBuffer(mesg);
        channelWrite(channel, writeBuffer);
    }

private void sendBroadcastMessage(String mesg, SocketChannel from)
```

```
    {
        prepWriteBuffer(mesg);
        Iterator i = clients.iterator();
        while (i.hasNext()) {
            SocketChannel channel = (SocketChannel)i.next();
            if (channel != from)
                channelWrite(channel, writeBuffer);
        }
    }

    private void prepWriteBuffer(String mesg) {
        writeBuffer.clear();
        writeBuffer.put(mesg.getBytes());
        writeBuffer.putChar('\n');
        writeBuffer.flip();
    }

private void channelWrite(SocketChannel channel,
                                ByteBuffer writeBuffer) {
        long nbytes = 0;
        long toWrite = writeBuffer.remaining();

        try {
            while (nbytes != toWrite) {
                nbytes += channel.write(writeBuffer);

                try {
                    Thread.sleep(CHANNEL_WRITE_SLEEP);
                }
                catch (InterruptedException e) {}
            }
        }
        catch (ClosedChannelException cce) {}
        catch (Exception e) {}

    writeBuffer.rewind();
    }
```

在 prepWriteBuffer()中，首先声明 writeBuffer()方法，然后填入 ByteBuffer.put()方法提供的字符串中的数据。然后调用 writeBuffer.flip(),让缓冲区准备写入信道。前面曾介绍过，

flip()方法将缓冲区极限设置为当前位置，是刚刚插入的数据末尾，并将位置复位为 0。

ChannelWrite()方法是包围 Channel.write()调用的循环。循环的原因是非阻止 SocketChannel 不一定能一次写入指定缓冲区的全部内容，而只能写入套接的输出缓冲区提供的字节数。注意可以用 Socket.setSendBufferSize()和 Socket.setReceivBuf ferSize()方法改变套接缓冲区的长度，它们的缺省值是与操作系统相关的。在这个 ChatterBox 应用程序中，消息通常在一次调用中发送到 write()方法，但这是个好习惯，真正的客户机与服务器应用程序中消息长度和数量要大得多，所以需要这个功能。

注意：channelWrite()末尾调用了 writeBuffer.rewind()方法，其设置后续信道写入的缓冲区，将缓冲区位置移回 0，同时极限和缓冲区内容保持不动。

SendBroadcastMessage()对清单中每个客户机调用 sendMessage()方法，只有 from 客户机例外，因为消息源不必再收一份拷贝。

5. 读取消息

从客户机接收消息要比发送消息复杂一些，它问题不在于从信道取字节，而在于需要确定整个消息何时到达。清单 6.4 显示了 ChatterServer 程序完整的 read()方法。

清单 6.4 ChatterServer.readIncomingMessages()

```
private void readIncomingMessages() {
try {
    readSelector.selectNow();
    Set readyKeys = readSelector.selectedKeys();
    Iterator i = readyKeys.iterator();
    while (i.hasNext()) {
        SelectionKey key = (SelectionKey) i.next();
        i.remove();
        SocketChannel channel = (SocketChannel) key.channel();
        readBuffer.clear();
    long nbytes = channel.read(readBuffer);

    // 检查流末尾
        if (nbytes == -1) {
            log.info("disconnect: " +
            channel.socket().getInetAddress() +
            ", end-of-stream");
            channel.close();
            clients.remove(channel);
            sendBroadcastMessage("logout: " +
            channel.socket().getInetAddress() , channel);
        }
        else {
```

```
StringBuffer sb = (StringBuffer)key.attachment();
readBuffer.flip( );
String str = asciiDecoder.decode( readBuffer).toString( );
readBuffer.clear( );
sb.append( str);

String line = sb.toString();
    if ((line.indexOf("\n") != -1) ||
    (line.indexOf("\r") != -1)) {
        line = line.trim();
        log.info ("got mesg: " + line);
        if (line.startsWith("quit")) {
            log.info("got quit msg, closing channel for : " +
                channel.socket().getInetAddress());
            channel.close();
            clients.remove(channel);
            sendBroadcastMessage("logout: " +
          channel.socket().getInetAddress(), channel);
        }
        else {
            log.info("broadcasting: " + line);
            sendBroadcastMessage(
                channel.socket().getInetAddress() +
                ": " + line, channel);
                sb.delete(0, sb.length());
            }
        }
    }
}
catch (IOException ioe) {
        log.warn("error during select(): ", ioe);
    }
catch (Exception e) {
        log.error("exception in run()", e);
    }
}
```

代码首先对 readSelector 对象执行 selectNow()方法，来检查任何信道中是否有感兴趣的活动，即设置 OP_READ 键。Selector.selectNow()方法立即返回，不管有多少准备的键（可

能是 0 个)。这里还是用非阻止调用，避免占用唯一的线程。如果调用 select()方法，则会阻止，直到信道中收到数据，这样会使任何等待的外走消息受阻。后面的完整服务器中会实现让一个线程专门进行套接读取，一个线程专门进行套接写入，就是使用阻止 select()方法。

有了准备键之后，就可以取得 SocketChannel 的引用和调用其 read()方法。在 readBuffer 中填入套接输入缓冲区提供的字节数，处理这个数据前，首先要检查流结尾的条件。SocketChannel.read()方法返回-1，即表示到达流结尾并套接关闭。

知道有一些字节后，取得前面存放成选择键附件的 StringBuffer，然后将 readBuffer 内容添加到 StringBuffer 中，并检查是否有完整消息。

应用程序终止每个消息时会使用新行符\n，所以用 String.indexOf()方法检查该字符是否存在。本章稍后的 GameEvents 一节将详细介绍消息格式，目前先了解这些知识就够了。

6. 切断

客户机最终要关闭与服务器的连接，原因有很多种，方式也是多种多样，所以要保证包括所有情形并采取相应的措施。

客户机关闭连接时可能向服务器发一个 quit 命令。服务器在 readIncomingMessages()方法中检查 quit 消息并关闭客户机信道。这是较好的方法，因为它使客户机和服务器都能进行必要的清理工作。

强力终止客户机连接时，可能是客户机应用程序突然关闭信道或退出，或因为网络连接失败。服务器的第一个反应可能是 readIncomingMessages()方法中 channel.read()返回端口末指示符（如前所述），也可能是 sendMessage()或 readIncomingMessages()方法中捕获 ChannelClosedException 或 IOException 异常。

无论哪种情况，一定要保证关闭信道，并从客户机清单中删除客户机和进行任何必要的清理工作。

说明：不一定要从选择器注销信道。信道在关闭时取消 SelectionKey，使其从选择器的键集中删除、详见 Selector API 文档。

6.2.2 建立与运行服务器

第一个实际的服务器程序终于完成，下面就要建立与运行服务器了，并进行一些测试。但还没有编写客户机呀?没问题，稍后将会看到，由于选择简单消息格式(新行终止字符串)，所以可以使用任何旧式 Telnet 客户机与服务器通信。

首先取得代码拷贝，然后用 Ant 建立 ChatterBox 应用程序，输入命令如下。

```
$ cd chap06
$ ant
```

代码编译后即可测试。测试主要打开 3 个终端窗口（或 use 屏幕，见 www.gnu.org/software/screen/）。

在第一个终端中，用提供的 Shell 脚本运行服务器，输入命令如下。

```
$./bin/chatter_server.sh
```

可以看到屏幕上打印一些初始日志消息。

然后，下面开始介绍使用 Telnet 客户机进行连接。

1983 年 RFC 854(www.faqs.org/rfcs/rfc854.html)中就定义了 Telnet 协议。主要用于本地和远程终端间的通信，通常用于交互式 Shell 会话。由于安全原因，功能大部分已经换成安全 Shell(sh)协议，但 Telnet 还没有消失。让服务器与未发展的 Telnet 协议子集通信，可以利用现代操作系统中无处不在的 Telnet 客户机，把它作为测试客户机。

从另一终端 Telnet 到服务器主机 IP（本例中为 10.0.0.1），并提供如下连接的端口号。

```
$ telnet 10.0.0.1 10997
Trying 10.0.0.1...
Connected to 10.0.0.1.
Escape character is '^]'.

Welcome to ChatterBox, there are 1 users online.

Type 'quit' to exit.
```

这里出现一个欢迎消息，并显示联机用户数。在服务器终端中，可以看到日志消息，表示收到连接。

然后从第 3 个终端进行相同操作，这时在连接之后输入一个消息并按 Return 键，可以看到其他客户机终端回显这个消息。接着自己和自己对话吧。

```
$ telnet 10.0.0.1 10997
Trying 10.0.0.1...
Connected to 10.0.0.1.
Escape character is '^]'.

Welcome to ChatterBox, there are 2 users online.

Type 'quit' to exit.

hello there
/10.0.0.1: wassup?

quit
Connection closed by foreign host.
$
```

这样就有了一个多用户服务器应用程序。虽然很简单，但已经能用了。下面要构造真正的客户机应用程序。

6.2.3 客户端

ChatterClient 是简单的客户机应用程序，与 ChatterServer 有许多共同之处，主要完成 3 个任务：

- 连接服务器；
- 发送消息；
- 接收消息。

1. 设置

首先要进行一些初始化。像 ChatterServer 一样设置 2 个 ByteBuffers，一个用于读取，一个用于写入。此外，进入主循环之前，要启动控制台输入线程，这个线程负责从标准输入读取用户输入。

2. 建立连接

前面曾介绍过，用 SocketChannel 实现套接连接，清单 6.5 显示 connect()方法。

清单 6.5 ChatterServer.connect()

```
private void connect(String hostname) {
try {
        readSelector = Selector.open();
        InetAddress addr = InetAddress.getByName(hostname);
        channel = SocketChannel.open(new InetSocketAddress(addr, PORT));
        channel.configureBlocking( false);
        channel.register(readSelector, SelectionKey.OP_READ,
        new StringBuffer());
    }
    catch (UnknownHostException uhe) { }
    cath  (ConnectException ce) { }
    catch (Exception e) { }
}
```

从上述代码中，首先打开新选择器，并用 InetAddress.getByName()方法取得服务器的 IP 地址。然后用地址和所用的端口号(10997)打开 SocketChannel。接下来将信道配置成非阻止方式。最后要向选择器注册信道，还是用 StringBuffer 作为附件。

注意：在 ChatterBox 例子中，为了简明起见，基本忽略了异常。在生产代码中，一定要捕获所有异常并进行相应处理，以便优雅地恢复，登记错误消息，警告用户或进行组合操作。

3. 主循环

客户机使用 2 个线程，一个用于控制台输入/写入(InputThread)，一个则是主执行线程。主循环在 run()方法中，重复调用 readIncomingMessages()方法与服务器通信。发往服务器的

外走消息在 InputThread 中启动。

4. 发送与接收消息

从代码中可以看出，客户机和服务器中发送、接收消息大致是相同的。后面的游戏服务器框架中要把这类公共代码分解到实用程序类中，来避免冗余。

6.2.4 建立与运行客户机

和建立、运行服务器一样，使用 Ant 建立代码，输入如下命令。

```
$ cd chap06
$ ant
```

然后用提供的 shell 脚本启动客户机。ChatterClient 取客户机 IP 和端口参数，因此调用时和上面的 Telnet 例子一样，输入如下命令。

```
$ ./bin/chatter_client.sh 10.0.0.1 10997
```

交互也和上面的 Telnet 例子一样：

```
$ ./bin/chatter_client.sh 10.0.0.1 10997
Welcome to ChatterBox, there are 1 users online.
Type ‘quit’ to exit.
>
login from /10.0.0.7
> hello
/10.0.0.7: wassup?
>
```

这样就完成了第一个多用户应用程序，下面要做一些更大更好的工作。

6.3 多方游戏服务器框架

6.2 节中介绍了如何用 NIO 库建立基本的客户机/服务器应用程序，下面要把 Chatterbox 实例进一步深化，变成完全特性的多方游戏服务器框架。

6.3.1 设计目标与战术

游戏服务器框架的设计有如下的目标：

- 简单性。核心类应尽量流畅；
- 通用性和可扩展性。游戏要支持不同的游戏风格，最好在服务器的一个实例中支持；
- 伸缩性。服务器要处理大量并发的连接；
- 性能。设计要致力于提供最高的消息通过量，低速事件操作（如需要数据库或文件访问时）不能阻止其他事件的流程。
- 为了达到这些设计目标，要对原先的实例作几个关键性的修改：
- 将通信信息包装到 GameEvents 中；

- 明确分开逻辑与机制；
- 将服务器过程分成管道的各个阶段；
- 每个阶段用线程池支持并发操作；
- 用队列分开每个阶段以缓存事件；
- 提供游戏逻辑插入件体系结构，在相同服务器应用程序中支持多个游戏。

在介绍这些目标之前，先要建立其他一些工具。第 3 章“交互性与用户界面”介绍了 Java 线程，其中的实例 ThreadPool 类。本节要建立线程池的变形，使用 GameEvents 的阻止队列（BlockingQueue）。

ThreadPool 和 BlockingQueue 组合而成的设计模式称为包装（Wrap）。Matt Welsh 及其合作者在论文 A Design Framework for Highly Concurrent Systems(Computer Science Division, University of California, Berkeley, 2002)中指出：

包装模式用队列接口包装一组线程。每个线程通过几个阶段处理单个任务，可能一次或多次受阻。包装使一组线程使用单个输入队列，等于从中建立了任务处理器。这个操作使任务处理器中的处理更可靠，因为任务处理器中的线程数可以固定为某个值，防止线程开销使性能下降，这些线程无法提供服务的其他任务集中到队列中。

BlockingQueue 是个 Queue 数据结构，带有一个阻止 deQueue()方法。换句话说，调用 deQueue()方法时，运行线程受阻，直到队列中提供一个元素，否则可能要永久等待。关于线程的图书大都会介绍 BlockingQueue，因此读者只要参阅实现源代码中的细节(com.hypefiend.javagamebook.common.EventQueue)。包装对这个服务器应用程序非常有利，因为可以让几个线程等待执行管道中的步骤，用入站事件队列从管道其他阶段缓存这些步骤。这里的一个重要细节是，管道每个包装或阶段的线程数是系统的可调参数，特定任务只分配正确的线程数，从而达到最大事件通过量。

6.3.2 设计

服务器设计按照系统中的数据流程进行，如图 6-5 所示。

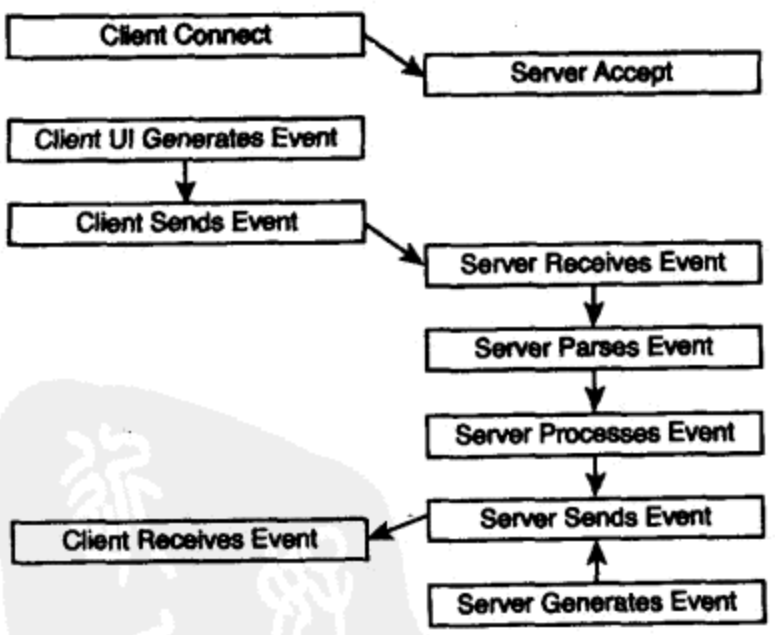


图 6-5 客户机/服务器通信流程

首先是客户机连接服务器和服务器接收这个连接。然后客户机发送事件到服务器中。

服务器读取、分析和处理事件。这个过程通常会生成外走事件和发送到一个或几个客户机中。然后客户机接收事件并进行处理，循环不断重复。

可以看出，大多数交互都是客户机驱动的，但服务器也可能独立于客户机操作而产生一些事件。例如，可能响应服务器中的定时器，表示游戏结束时间或系统级管理消息。

必须进行一些设计选择，帮助建立应用程序。首先要确定事件格式，因为事件是应用程序的主要数据结构。然后要确定用什么类和接口实现上述序列。

1. GameEvents

客户机与服务器和管道每个阶段之间通过事件进行通信。事件格式的选择对各个游戏设计很重要，但对框架是透明的，因为我们提供事件和段与段之间传递事件的接口。这样，每个游戏可以选择不同的事件格式。

系统中可以用许多不同格式传输数据，格式的选择主要取决于放在框架中的游戏特征。在最基本的层次上，可以在 ChatterBox 应用程序中使用简单 ASCII 文本消息。但是，任何具有实际功能级的应用程序都需要更复杂的结构。虽然这里介绍几个选项，但仍会有更多的选项。应该针对游戏设计合适的事件格式，下面是选择事件格式时的设计准则：

- 长度/紧凑性。由于需要通过网络发送事件，所以在消息流量较大时，带宽是个问题。拱廊式游戏显然要比卡片游戏需要更多消息；
- 可读性。提高格式的可读性有助于调试游戏，但与长度准则冲突。一个折衷的方法就是用缩写可读格式来缩短消息的同时保持可读性；
- 序列化/分析难度与速度。事件要从对象格式转换成线上格式，然后从线上格式转换成对象格式，这个过程应易于实现，快速执行；
- 序列化/分析代码长度。根据部署情形，客户端代码长度可能很关键（例如移动设备），因此也许不能用大型代码库来分析事件；
- 灵活性。需要处理各种不同事件类型。

下面介绍几个常见事件格式选项，看看其如何满足这些准则。无论选择什么事件格式，一定要收集特定方法的长度与性能数据，保证其符合带宽、灵活性和处理器开销要求，最好在编写项目主代码之间收集到这些数据。

2. 序列化 Java 对象

这个方法用 Java 内置序列化功能将 GameEvent 对象变成字节流。好处是很容易实现，长度效率高，序列化速度不错，进行转换时增加的代码也很少。从 JDK1.1 开始，Java 内核库中就提供了序列化功能，但这个格式不是人们可读的。

3. XML

XML 在各种网络通信和数据存储中越来越普及。对于 GameEvent，其好处是容易实现，可读和对各种消息类型很灵活，但缺点更大。序列化长度很长（除非使用高度缩略的元素和属性名），处理大量事件时的分析开销也很大。另外，客户机上要增加的分析代码也很成问题。但是，5k NanoXML 分析器（见 http://nanoxml.n3.net）对长度受限的客户机提供了很好的选择。

4. 定制二进制格式

这种格式直接把 GameEvent 的字段写入字节序列，对每种基本数据类型使用尽量少的字节。

可以用某些技术将基本数据类型压缩到比自然长度要求小得多，例如可以用游长编码方法（RLE）压缩字符串，用位将多个布值包装成一个字节，以及量化和压缩整数。

这种方法的长度最佳，很容易编写，也很容易分析。但和序列化对象一样，它的变量不可读的。如果带宽非常紧缺，则这是唯一可行的办法。

5. 二进制编码 XML

编程人员认识到，尽管 XML 的描述性很好，但占用太多带宽和内存，而带宽和内存对无线设备非常重要。万维网联盟（W3C）推出的 WAP 二进制 XML 格式（www.w3.org/TR/wbxml）有望解决这个问题。

这类编码方法提供很好的折衷，传输时是不可读的，但译码后成为正常的 XML。这种格式在传输时长度效率高，但编码/译码会增加不少开销。

6. 术语

实现之前，先要定义框架中使用的几个术语：

- GameName。游戏类，如 Chess, TicTacToe 与 SpaceWar；
- Game。GameName 的特定实例；
- GameID。游戏的唯一标识符；
- Player。连接服务器的可标识用户，可能参与或未参与游戏；
- PlayerID。游戏者的唯一标识符；
- Session。游戏者登录与退出之间的时间，可能跨越多个连接/切断周期；
- SessionID。会话的唯一标识符；
- EventType。一组事件类池之一，确定事件目的，如 LOGIN, CHAT_MSG 或 MOVE。

7. 类框图

在游戏服务器框架中，服务器方和客户机方的类框图如图 6-6 和图 6-7 所示。

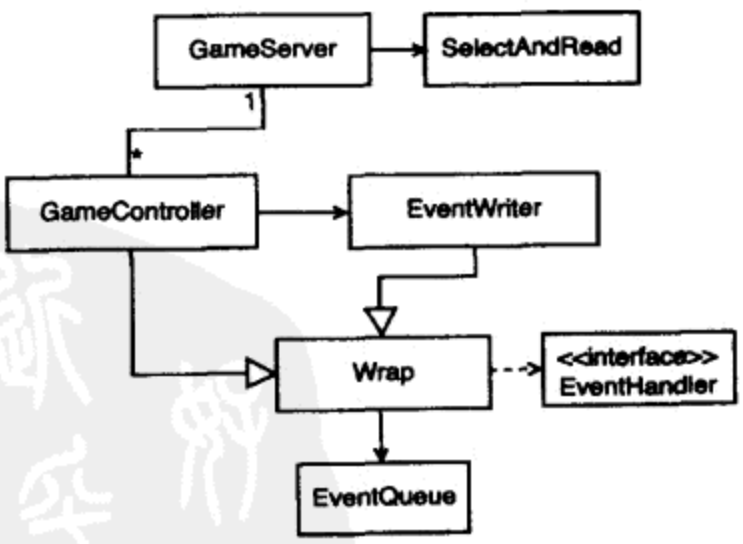


图 6-6 服务器框架类框图

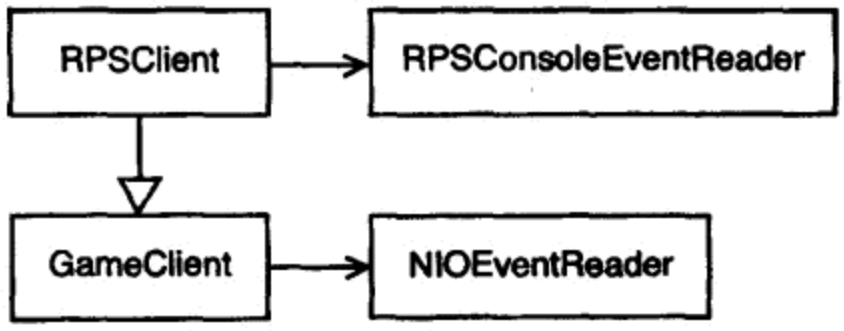


图 6-7　客户机框架类框图

6.3.3　共用类与接口

服务器方和客户机方有几个共用类、接口，它们都放在 com.hypefiend.javagamebook.common 包中：

- Attachment。表示选择键附件的类，用于保存读取缓冲区，检查完整消息荷载；
- EventHandler。只定义一个 handleGameEvent()方法的接口；
- EventQueue。前面介绍过的阻止队列；
- GameConfig。保存 GameController 类配置数据的类；
- GameEvent。GameEvents 的接口，使框架可以按一般性方法处理每个游戏的事件；
- GameEventDefault。GameEvents 的基本实现，在样本游戏中使用；
- Globals。定义服务器方和客户机方公用常量的类；
- NIOUtils。包含几个处理信道和缓冲区的实用程序的函数；
- Player。游戏者的接口，使框架可以按一般性方法处理每个游戏的游戏者；
- PlayerDefault。Player 的基本实现，在样本游戏中使用。

其中一些类和接口需要改进。下面首先详细介绍这些共用类、接口，然后介绍服务器的特定类。

1. GameEvent 接口

框架中定义了一个事件接口并提供缺省的实现，但游戏中要根据需要替换或扩展缺省实现。清单 6.6 显示了完整的 GameEvent 接口。

清单 6.6　GameEvent 接口

```
public interface GameEvent {
    public int getType();
    public void setType(int type);
    public String getGameName();
    public void setGameName(String gameName);
    public String getMessage();
    public void setMessage(String message);
    public String getPlayerId();
    public void setPlayerId(String id);
    public String getSessionId();
```

```
    public void setSessionId(String id);
    public String[] getRecipients();
    public void setRecipients(String[] recipients);
    public void read(ByteBuffer buff);
    public int write(ByteBuffer buff);
}
```

除了 GameEvent 接口外，还需要一个接口以便在服务器事件处理管道的不同阶段之间传递事件。这就是 EventListener 接口，它只有一个方法。

```
public void handleEvent(GameEvent event);
```

此外，所有接收入站事件的服务器和客户机都要实现 GameEventDefault 接口。

GameEvent 的缺省实现 GameEventDefault 直接用 ByteBuffer 的 put()和 get()方法，以原始字节序列形式读取和写入事件数据。例如，要写入 eventType 成员，代码如下。

```
buff.putInt(eventType);
```

要写入字符串成员（如消息），首先要写入表示字符串长度的字节数，然后写入字符串本身的字节，实现代码如下。

```
buff.putShort((short)str.length());
buff.put(str.getBytes());
```

清单 6.7 显示了 GameEventDefault 的 read()与 write()方法。

清单 6.7　GameEventDefault 的读与写方法

```
public int write(ByteBuffer buff) {
    int pos = buff.position();
    buff.putInt(eventType);
    NIOUtils.putStr(buff, playerId);
    NIOUtils.putStr(buff, sessionId);
    buff.putInt(gameId);
    NIOUtils.putStr(buff, gameName);
    buff.putInt(numRecipients);
    for (int i=0;i<numRecipients;i++)
        NIOUtils.putStr(buff, recipients[i]);
    NIOUtils.putStr(buff, message);

    return buff.position() - pos;
}

public void read(ByteBuffer buff) {
    eventType = buff.getInt();
    playerId = NIOUtils.getStr(buff);
```

```
        sessionId = NIOUtils.getStr(buff);
        gameId = buff.getInt();
        gameName = NIOUtils.getStr(buff);
        numRecipients = buff.getInt();
        recipients = new String[numRecipients];
        for (int i = 0;i < numRecipients;i++)
            recipients[i] = NIOUtils.getStr(buff);
        message = NIOUtils.getStr(buff);
    }
```

注意：写入方法返回一个整数值，指定写入的字节数，这对框架的线上事件协议非常重要。

2. 线上事件协议

为了适应不同事件格式，在网络上发送事件时，要在事件数据前面加上一个小头。如图 6-8 显示头和荷载的格式。首先发送 clientId、GameName 散列码和 payloadSize（字节数），它们都是 4 字节整数。然后发送荷载，长度为 payloadSize 字节。

ClientId (4 bytes)	GameName hash code (4 bytes)	PayloadSize (4 bytes)	Payload (PayloadSize bytes)

图 6-8　线上事件格式

在后面服务器实现的 SelectAndRead 和 EventWriter 类时，将会介绍如何实际读取和写入数据，以及在接收端实例化适当的 GameEvents 对象。

6.3.4　服务器实现

框架的服务器方包括如下类：

- EventWriter；
- GameController；
- GameServer；
- SelectAndRead。

下面将详细地介绍每个类，因为它们是框架的基础。

1. GameServer 和 GameController

GameServer 类是服务器应用程序的核心，其中包含应用程序的 main()方法。GameServer 的责任是接收入站套接连接，跟踪连接客户机和管理 GameController。建立服务器套接和接收连接的代码与 ChatterServer 相似，如清单 6.8 所示。

清单 6.8　GameServer initServerSocket()与 run()方法

```
    private void initServerSocket() {
    try {
```

```
            sSockChan = ServerSocketChannel.open();
            sSockChan.configureBlocking(false);

            InetAddress addr = InetAddress.getLocalHost();
            sSockChan.socket().bind(new InetSocketAddress(addr, Globals.PORT));

            selector = Selector.open();
            SelectionKey acceptKey = sSockChan.register(selector,
                SelectionKey.OP_ACCEPT):
        }
        catch (Exception e) {
            log.error("error initializing serverSocket", e);
        }
    }

    public void run() {
        init();

        log.info("******** GameServer running ********");
        running = true;
        int numReady = 0;

        while (running) {
            try {
                selector.select();
                Set readyKeys = readSelector.selectedKeys();
                Iterator i = readyKeys.iterator();
                while (i.hasNext()) {
                    SelectionKey key = (SelectionKey) i.next();
                    i.remove();

                ServerSocketChannel ssChannel =
                    (ServerSocketChannel) key.channel();

                    SocketChannel clientChannel = ssChannel.accept();

                SelectAndRead.addNewClient(client Channel);
                log.info("got connection from: " +
                    clientChannel.socket().getInetAddress());
```

```
                }
            }
        catch (IOException ioe) {
            log.warn("error during serverSocket select(): ", ioe); }
            catch (Exception e) {
            log.error("exception in run()", e);
            }
        }
    }
```

GameServer 代码和 ChatterServer 相似，但有几个差别需要注意。首先，使用 ServerSocketChannel 的选择器。选择器可以在读取时多路复用客户机信道，也可以多路复用 ServerSocketChannel。这适用于服务器应用程序要监听多个地址和端口时。虽然框架目前没有使用这个功能，但在后面介绍远程管理控制台时会对此进行详细的介绍。

run()方法中另一个差别是，当接收连接之后，它将传递给 SelectAndRead 类进一步处理。

2. SelectAndRead

顾名思义，SelectAndRead 放置多路复用客户机信道的选择器，并从这些信道种读取 GameEvents。完整事件根据事件头中的 GameName 散列码传递给 GameControllers。

新的客户机连接从 GameServer 交给 SelectAndRead，使用 SelectAndRead.add NewClient() 方法。客户机的 SocketChannel 加进 LinkedList，并调用 selector.wakeup()方法中断 SelectAndRead 中阻止的 select()调用。这样就从 select()方法中打破 SelectAndRead 线程，然后调用 checkNewConnections()方法，其遍历新客户机 SocketChannel 的 LinkedList，向选择器注册。清单 6.9 显示了这几个方法的代码。

清单 6.9 SelectAndRead, addNewClient()与 checkNewConnections()

```
public void addNewClient(SocketChannel clientChannel) {
    synchronized (newClients) {
        newClients.addLast(clientChannel);
    }
    // 强制选择器返回
    // 使新客户机能在循环中正确工作
selector.wakeup();
}

private void checkNewConnections() {
synchronized(newClients) {
        while (newClients.size() > 0) {
            try {
```

```
                    SocketChannel clientChannel =
                    (SocketChannel)newClients.removeFirst();
                clientChannel.configureBlocking( false);
                    clientChannel.register( selector, SelectionKey.OP_READ,
                    new Attachment());
            }
                catch (ClosedChannelException cce) {
                log.error("channel closed", cce);
                }
            catch (IOException ioe) {
                    log.error("ioexception on clientChannel", ioe);
                }
            }
        }
    }
```

SelectAndRead 使用与 ChatterServer 中相似的方法从客户端信道读取数据，只有几个例外。

它不是从信道读取字符串和检查分隔消息的新行符，而是查找完整的 GameEvents。

由于事件格式是实现特定的，所以框架只读取事件的原始字节，而通过 createGameEvent()让 GameController 实例负责 GameEvents 的实例化。清单 6.10 显示了 select() 方法和读取与委托 GameEvents 的方法。

清单 6.10　读取 GameEvents

```
private void select() {
    try {
        selector.select();
        Set readyKeys = selector.selectedKeys();
        Iterator i = readyKeys.iterator();
        while (i.hasNext()) {
            SelectionKey key = (SelectionKey) i.next();
            i.remove();
            SocketChannel channel = (SocketChannel) key.channel();
            Attachment attachment = (Attachment) key.attachment();

            long nbytes = channel.read(attachment.readBuff);
            // 检查流末尾条件
            if (nbytes == -1) {
                log.info("disconnect: " +
                channel.socket().getInetAddress() +
```

```
                ", end-of-stream");
            channel.close();
        }

        // 检查完整事件
        try {
            if (attachment.readBuff.position() >=
            attachment.HEADER_SIZE) {
                attachment.readBuff.flip();

            // 读取缓冲区中提供的事件
                while(attachment.eventReady()) {
                    GameEvent event = getEvent(attachment);
                    delegateEvent(event, channel);
                    attachment.reset();
            }
            // 准备更多信道读取
                attachment.readBuff.compact();
            }
        }
            catch (IllegalArgumentException e) {
                log.error("illegal argument exception", e);
            }
        }
    }
    catch (IOException ioe) {
        log.warn("error during select(): ", ioe);
    }
    catch (Exception e) {
        log.error("exception during select()", e);
    }
}

private GameEvent getEvent(Attachment attachment) {
    GameEvent event = null;
    ByteBuffer bb = ByteBuffer.wrap(attachment.payload);

    // 取得控制器并让其实例化事件
    GameController gc =
```

```
        gameServer.getGameControllerByHash(attachment.gameNameHash);
        if (gc == null) {
            return null;
        }
        event = gc.createGameEvent();

        // 从荷载读取数据
        event.read(bb);
        return event;
    }

    private void delegateEvent(GameEvent event, SocketChannel channel) {
        if (event != null && event.getGameName() == null) {
            log.error("GameServer.handleEvent() : gameName is null");
            return;
        }

        GameController gc;
    gc = gameServer.getGameController(event.getGameName());
    if (gc == null) {
        log.error("No GameController for gameName: "
        + event.getGameName());
        return;
    }

    Player p = gameServer.getPlayerById(event.getPlayerId());
    if (p != null) {
        if (p.getChannel() != channel) {
        log.warn("player is on a new channel, must be reconnect.");
        p.setChannel(channel);
        }
    }
    else {
        // 首次看到playerId,生成Player对象和建立信道，并加入清单
        p = gc.createPlayer();
        p.setPlayerId(event.getPlayerId());
        p.setChannel(channel);
        gameServer.addPlayer(p);
```

```
gc.handleEvent(event);
}
```

SelectAndRead 从 SocketChannel 读取数据时，将临时存放在 Attachment 类的缓冲区中。Attachment 类只是这个数据的保存单元，只包含验证已经读取头和荷载数据的方法。

每次读取之后，检查事件头，如果 readBuffer 中至少有 payloadSize 字节，则事件是完整的。GameName 散列码用于从 GameServer 取得相应 GameController 的实例。然后调用 GameController 的 createGameEvent()方法实例化新事件，事件从 Attachment.readBuffer 内容读取。

处理初始用户连接和生成游戏者的方法比较复杂一些，需要稍作解释。GameServer 是主游戏者名单的保持者。当 SelectAndRead 取得完整事件时，首先检查 playerId,并询问 GameServer 前面是否发送过这个 ID。如果没有，则是新连接。然后调用 GameController，用 createPlayer()方法实例化新游戏者。

GameController 从游戏者那里取得登录事件时，要查询 GameServer，取得前面实例化的游戏者对象。这好像有点麻烦，但强制 GameServer 严格处理游戏者生成和跟踪。

3. GameController

GameController 包装服务器方游戏的逻辑，可以在一个 GameServer 中同时运行各种不同的游戏。所有 GameController 都要扩展 GameController 基类，这个基类则扩展 Wrap。这样，每个 GameController 都有自己的 EventQueue 和线程池。

GameServer, SelectAndRead 和 GameEvent 实现形成一个组，提供处理客户机/服务器通信的基础，并对系统其他部分隐藏网络实现的细节（这里是 NIO 信道）。换句话说，GameController 不必考虑事件如何读取和写入，只要入站游戏事件出现在 EventQueue 中，出站游戏事件放进 EventWriter 的 EventQueue 中。可以将基础网络代码改写成使用 UDP,HTTP 或其他事件移动方法，而不必修改 GameController。

GameController 在初始化期间由 GameServer 自动装入。这个方法使 GameServer 不会与固定的一组 GameController 相联系。事实上，它要等安装之后才知道 GameController 的名称。清单 6.11 显示了 GameServer 中装入 GameController 实例的代码。

清单 6.11 GameServer,getGameController()与 loadGameController()方法

```
private void loadGameControllers() {
    log.info("loading GameControllers");
    // 取得 GameController 所在目录中的所有类文件
    String baseClass =
    "com/hypefiend/javagamebook/server/controller/GameController.class";
    File f = new File( this.getClass(
    ).getClassLoader().getResource(baseClass).getPath());
    File[] files = f.getParentFile().listFiles( );
    if (files == null) {
        log.error("error getting GameController directory");
```

```
            return;
        }

        for( int i = 0; ( i < files.length); i++) {
            String file = files[i].getName( );
            if (file.indexOf( ".class") == -1)
            continue;
            if (file.equals("GameController.class"))
                continue;
            try {
            String controllerClassName = CONTROLLER_CLASS_PREFIX +
                file.substring(0, file.indexOf(".class"));
            log.info("loading class: " + controllerClassName);
            Class cl = Class.forName(controllerClassName);
            // 保证其扩展 GameController
            if (!GameController.class.isAssignableFrom(cl)) {
                log.warn("class file does not extend GameController: " +
                file);
                continue;
            }

            // 取得实例和初始化
            GameController gc = (GameController) cl.newInstance();
            String gameName = gc.getGameName();
            gc.init(this, getGameConfig(gameName));

            // 加进控制器散列
            gameControllers.put("" + gameName.hashCode(), gc);
            log.info("loaded controller for gameName: " + gameName +
                ", hash: " + gameName.hashCode());
            }
        catch (Exception e) {
            log.error("Error instantiating GameController from file: " + file, e);
            }
    }
        }
```

GameName 的散列码是散列表中存储 GameControllers 的键。线上协议的头中也使用这个键，详见前面的介绍。

GameControllers 基类定义几个抽象方法，并要在子类中实现：

- protected abstract void initController(GameConfig gc)。方法让 GameContro llers 进行所要的初始化；
- public abstract String getGameName()。返回控制器的 GameName；
- public abstract Player createPlayer()。工厂方法，返回实现 Player 接口的对象；
- public abstract GameEvent createGameEvent()。工厂方法，返回实现 GameEvent 接口的对象。

2 个工厂方法 createPlayer()和 createGameEvent()使核心 GameServer 可以在需要时为控制器实例化这些对象，而不必知道特定控制器所用的实现方法。

GameControllers 负责从 EventQueue 取得入站事件（前面曾介绍过，是 SelectAndRead 类放在这里的），处理这些事件，并将出站事件放进 EventWriter 的 EventQueue 中。processEvent()方法(来自 GameControllers 扩展的 Wrap 类)在 EventQueue 中每次 GameEvent 到达时调用，这个方法的签名如下。

protected abstract void processEvent(GameEvent event);

说明：一定要注意，通过扩展 Wrap，所有控制器都成为多线程的，因此处理事件时要相应使用同步。

所有控制器扩展 GameControllers 基类使框架可以在所有控制器中以一致的方式负责一些整理工作，如初始化和登记。基类处理 Wrap 的初始化和控制器的 Log4J 类别。基类的实现还管理运行周期，从入站队列取得事件和将其传递给 processEvent()方法。

将控制器限制为上面定义的方法后，框架提供了非常严格的环境，可以运行游戏逻辑。注意，每个控制器也取得 GameServer 的引用。根据游戏的复杂度不同，可能还需要其他功能。因此可以随意扩展控制器的功能，只要能在中央位置进行修改与补充，影响所有控制器，隔离 GameControllers 逻辑代码与网络通信代码即可。

4. EventWriter

EventWriter 是个 Wrap，处理外走事件队列。事件用 NIOUtils 中的实用程序方法 channelWrite()写入。发送广播事件时，它从 GameEvent 取得接收者名单，然后一一向每个游戏者发送事件。

6.3.5 客户机

游戏客户机所用的代码大部分与前面相同，只是增加了 2 个类：GameClient 和 NIOEventReader。

GameClient 是个客户机应用程序的抽象基类。游戏可以扩展此类，只要实现如下方法：

- getGameName。返回游戏的 GameName；
- createGameEvent。生成游戏特定 GameEvent 的工厂方法；
- createLoginEvent。生成游戏特定 GameEvent 的工厂方法，用于登录服务器；
- createDisconnectEvent。生成游戏特定 GameEvent 的工厂方法，用于切断；
- processIncomingEvents。处理 GameServer 发来的 GameEvent。

GameClient 还处理与服务器的初始连接和写入出站事件。

NIOEventReader 是 SelectAndRead 的客户端版本（简化的），除了客户机从一个

SocketChannel 读取，此外基本一致。NIOEventReader 将事件生成与处理委托给 GameClient 而不像 SelectAndRead 中一样委托给 GameControllers。

提示： JDK1.4 客户机与 JDK1.1GameClient 以及相关的类是用 JDK 1.4 NIO 建立的，必定这是本章的重点。根据部署情况，也许不一定使用 JDK1.4 虚拟机。若要生成使用旧版 JDK 库的客户机版本并不难，主要改变 NIOEventReader 和 GameClient 中的事件写入代码。

6.4 RPS 游戏（石头、布、剪刀）

当具备所有的框架代码后，实现游戏将很简单。只要生成客户机和服务器双方的逻辑和相关数据类即可。

本节的样本游戏是网络上的“石头、布、剪刀”（RPS）游戏。游戏可以使多个用户同时连接服务器，并在游戏中相互挑战。游戏中，2 个客户机可以玩许多局，当游戏结束后，服务器将返回一个结果，显示他们的赢、输和平的局数。同时，客户机还可以在游戏前和游戏期间相互闲聊。

下面首先实现“石头、布、剪刀”（RPS）游戏所需要的类。

6.4.1 类

这里使用 PlayerDefault 和 GameEventDefault 类，因此不用建立定制 Player 和 GameEven 版本。服务器方需要一个 GameControllers，RPSController。此外，增加一个保存游戏数据的类 RPSGame。客户端生成 GameClient 的子类 RPSClient 并增加 RPSConsoleEventReader，用于处理用户的输入。如图 6-9 所示与图 6-6 相似的类框架，只不过增加了 RPS 游戏中的类。

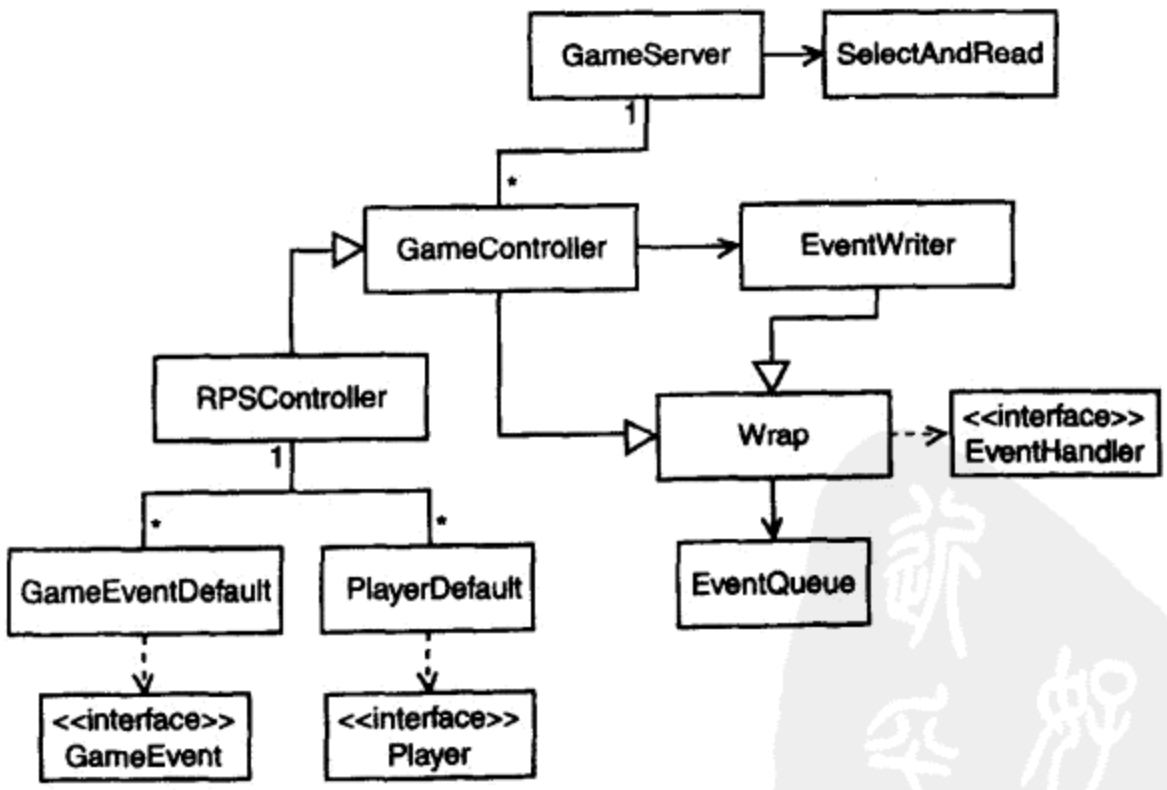


图 6-9 游戏服务器框架，增加 RPS 游戏的类

1. RPSController

游戏逻辑非常简单，但仍然需要几个事件类型和增加大量的逻辑代码。RPSController 跟踪 RPS 游戏者及游戏，这里不准备介绍逻辑的细节，只是介绍 RPSController 是如何与框架交互的。

收到的客户端事件发送到控制器的 EventQueue 中，然后由 GameController 基类中的 run()方法收集并传入 RPSController 的 processEvent()方法。

要将事件发送到客户端，RPSController 用基类的 sendEvent()或 sendBroadcast Event()方法。这些方法调用 GameServer.write()方法，其把事件放进 EventWriter 类的 GameServer.write()中。

注意：RPSController 类在 com.hypefiend.javagamebook.server.controller 包中，而其他类在 com.hypefiend.javagamebook.games.rps 包中。这是因为 GameServer 动态装入 GameController 的方法造成的。

2. RPSGame

RPSGame 表示一对游戏者之间的一组“石头、布、剪刀”（RPS），它跟踪游戏者在每一局的动作，并报告双方的赢、输和平局。

RPSClient 对于客户机，只要编程一些基本逻辑，处理从控制台读取输入和处理服务器的入站事件，以及进行一些整理工作，跟踪游戏者和游戏/挑战状态。

清单 6.12 显示整个 RPSClient 的代码。代码很简单，大多数事件只是把 GameEvent.getMessage()内容写入标准输出，其余都由 GameClient 基类处理。

清单 6.12 RPSClient.java

```
public class RPSClient extends GameClient {
    protected static Logger log = Logger.getLogger("RPSClient");
    protected RPSConsoleEventReader consoleReader;

    public static void main(String args[] ) {
        BasicConfigurator.configure();
        RPSClient gc = new RPSClient();
        gc.init(args);
        gc.start();
    }

    public void init(String args[]) {
    super.init(args);
    consoleReader = new RPSConsoleEventReader(this, inQueue, outQueue);
        consoleReader.start();
    }
```

```
    protected void shutdown() {
        consoleReader.shutdown();
        super.shutdown();
}

    protected void processIncomingEvents() {
        GameEvent inEvent;
        while (inQueue.size() > 0) {
            try {
                inEvent = inQueue.deQueue();

                switch (inEvent.getType()) {
                case GameEventDefault.S_LOGIN_ACK_OK:
                    break;
                case GameEventDefault.SB_LOGIN:
                    stdOut("login: " + inEvent.getMessage());
                    break;
                case GameEventDefault.SB_LOGOUT:
                    stdOut("logout: " + inEvent.getMessage());
                    break;
                case GameEventDefault.SB_CHAT_MSG:
                    stdOut(    inEvent.getPlayerId()    +    ":    "    +
inEvent.getMessage());
                    break;
                case GameEventDefault.S_DISCONNECT:
                    stdErr(    "disconnected    from    server:    "    +
inEvent.getMessage());
                    shutdown();
                    break;
                case GameEventDefault.S_JOIN_GAME_ACK_OK:
                    stdOut( inEvent.getMessage());
                    inGame = true;
                    break;
                case GameEventDefault.S_JOIN_GAME_ACK_FAIL:
                    stdOut( inEvent.getMessage());
                    inGame = false;
                    break;
                case GameEventDefault.SB_PLAYER_QUIT:
                    stdOut( inEvent.getMessage());
```

```
                    inGame = false;
                    break;
                default:
                    stdOut( inEvent.getMessage());
                    break;
                }
            }
        catch (InterruptedException ie) { }
        }
    }

    public String getGameName() {
        return "RPS";
}

    public GameEvent createGameEvent() {
        return new GameEventDefault();
    }
    public GameEvent createLoginEvent() {
        return new GameEventDefault(GameEventDefault.C_LOGIN);
    }

    public GameEvent createDisconnectEvent(String reason) {
        return new GameEventDefault(GameEventDefault.S_DISCONNECT, reason);
    }
}
```

3. RPSConsoleEventReader

ConsoleEventReader 客户端类通过标准输入来读取用户的输入。使用它读取整行文本，用 BufferedReader 从字符串中分解出命令，并根据令牌情况生成相应的动作 GameEvent，然后将其放进客户机的外走 EventQueue 中。

6.4.2 运行 RPS 游戏

首先要取得代码拷贝，然后用 Ant 建立游戏服务器和客户机，输入如下代码。

```
$ cd chap06
$ and
```

和 ChatterBox 实例一样，需要打开 3 个终端窗口。在第一个终端用提供的 Shell 脚本运行服务器，输入代码如下。

```
$ ./bin/server.sh
```

然后启动第一个客户机。脚本取 2 个参数：服务器 IP 地址和游戏者姓名，输入代码如下。

```
$ ./bin/client.sh 10.0.0.1 bret
```

然后启动另一个客户机，输入代码如下。

```
$ ./bin/client.sh 10.0.0.1 brackeen
```

此时，每个客户机显示游戏欢迎文本、帮助文本和一列联机游戏者名单，具体显示如下。

```
Commands:
'/quit'                                    quit the application
'/help'                                    show this help
'/players'                                 list players online
'/newgame <opponent name>'                 start a new game against
                                           opponent
'/move < (r) ock|(p)aper|(s)cissors>'      enter your move
'/endgame'                                 end the game
all other input is treated as a chat message

players online
brackeen
bret
brackeen   >
```

若要启动游戏，则让第一个游戏者输入 newgame 命令。注意所有命令前面都要有个斜杠（象 IRC 命令），输入的其他内容都作为闲聊的消息，如下所示。

```
bret> /newgame bracken
```

当 2 个游戏者都得到游戏开始的通知，就可以输入动作了。

```
brackeen > /move rock 与 bret > /move paper
```

此时，2 个游戏者都可以看到游戏的结果，如下所示。

```
Opponent chooses rock
You Win
bret vs. brackeen >
```

若要同一游戏者玩更多的游戏，只要继续输入动作。若想停下来，则输入 endgame 命令，这时就可以看看总共赢、输和平了多少局。

```
bret vs. brackeen > /endgame

GameOver, player bret has quit.
Final tallies
bret wins: 1
brackeen wins: 3
ties: 1
```

若要退出客户机，输入 quit 命令即可。

这样就利用可扩展游戏服务器框架成功地建立了多方游戏。读者可以浏览其他代码，更多地了解所有细节。

6.5　建立框架

前面的框架中还没有处理完全特性游戏系统中必需的几个项目，这里只介绍其中的一些，具体的实现方法留给读者去选择。

6.5.1　客户机 GUI

只编写了 RPS 游戏的控制台客户机，所以重点是网络和逻辑代码。但现有的大多数游戏显然都要求图形化客户机。其实增加 GUI 和将其连接到基础代码是简单的。基于事件设计可以让 GUI 把事件放进外走 EventQueue 中，就像 RPSConsoleEventReader 一样。同样，如果实现 EventHandler 接口，则 GameClient 可以将入站事件传递到 GUI。

6.5.2　持久性

游戏服务器还有一个重要方面，就是持久性。大多数游戏至少要存储一些游戏数据，通常放在关系型数据库中，通常要存储游戏者数据、最高分、游戏统计信息、伙伴名单和其他游戏特定的数据。

由于用独立 Java 应用程序作为 GameServer，因此不能利用现代 J2EE 应用程序服务器中内置的持久性机制。

怎么办?许多开发人员直接利用 JDBC API 的最小公共特性集，好处是速度快，适合需要的数据较少时。但是，最近的新趋势是利用对象/关系型贴图 API。Sun 公司推出了 Java 数据对象规范（JDO，见 http://java.sun.com/products/jdo），另外 2 个与之抗衡的 AII 是 Castor JDO(www.castor.org)和 Jakarta OJB (http://db.apache.org/ojb)，它们都正在积极开发之中。

6.5.3　伙伴名单、门厅与闲聊

多方服务器要有一种机制，它使游戏者寻找朋友、布置游戏和相互闲聊，这些特性统称为社区特性。

寻找朋友通常需要伙伴名单特性，类似于 IM（立即消息）客户机的功能，在朋友登录时通知你，并显示这个人参加哪个游戏厅或服务器。

游戏者匹配特性主要根据类别，但经常涉及门厅，让技术相近、游戏兴趣相近或符合其他条件的游戏者相聚、闲聊和相互挑战。

闲聊是几乎任何类别的游戏都需要的。除了基本闲聊功能外，还要让管理员能够赶走或禁止某个用户（IRC 服务器的特性）。游戏者要忽略某个游戏者（或所有闲聊），避免遇上罪恶的或讨厌的游戏者。

6.6 服务器管理

和任何服务器应用程序一样，游戏服务器需要定期监视和维护。服务器要产生日志文件，需要管理执行生命周期的工具（启动/关闭）。此外，游戏服务器还要有某种机制进行游戏监视、赶走或禁止某个用户和进行远程管理。

6.6.1 登记

登记是任何服务器应用程序的重要方面，特别是游戏服务器。每个应用程序至少要有一个错误日志和一个调试日志。此外，可能还要有闲聊日志（看看用户说你什么了）、游戏历史日志、连接/访问日志和其他游戏特定的日志。

JDK 1.4 中在 java.util.logging 包中提供了一个新日志的 API，但还太新，许多编程人员都不愿意放弃已经成为日志 API 事实标准的 Log4J。Log4J 是 1999 年诞生的，是 IBM 公司的 Alphaworks 启动和负责多年项目中的一部分，后来转到 Apache Jakarta 项目（http:Jakarta.apache.org）。Log4J 具有可配置、容易使用和高效等特点。下面的实例将使用 Log4J，但只进行基本的配置。

还要对游戏客户机产生日志。通常不想看到数据，但数据在测试期间非常重要。即使部署之后，如果用户遇到问题，可以让用户把日志文件发过来，帮助诊断问题。

有的发布机制要限制客户机应用程序的长度。这时 Log4J 库可能太大，所以就要寻找更袖珍的方法。

无论选择哪种 API，一定要登记适量的数据。数据太多会使日志文件堆砌太满，很难阅读，会浪费磁盘空间，而数据太少则对调试帮助不大。

6.6.2 启动/关闭

由于服务器要连续运行，因此要在系统启动/关闭程序时自动启动或关闭服务器。在基于 Unix 的平台上（Linux，Solaris 等），这是用启动脚本完成的。清单 6.13 的实例就是 Debian GNU/Linux 的 init.d 脚本，若稍作修改，也适用于其他 Linux 版本和 Unix 系统。

清单 6.13 init.d 脚本

```
#! /bin/sh

PATH=/usr/local/gameserver/bin:/usr/local/java/bin:/usr/local/sbin:
/usr/local/bin:/sbin:/bin:/usr/sbin:/usr/bin
DAEMON=/usr/local/gameserver/bin/server.sh
NAME=gameserver
DESC=" Multi-player GameServer"

test - x $DAEMON || exit 0

set - e
```

```
case "$1" in
  start)
        echo -n "Starting $DESC: $NAME"
        start-stop-daemon --start --quiet --exec $DAEMON
        echo "."
        ;;
  stop)
        echo -n "Starting $DESC: $NAME"
        start-stop-daemon --stop --quiet --pidfile /var/run/$NAME.pid
        rm /var/run/$NAME.pid
        echo "."
        ;;
  *)
        N=/etc/init.d/$NAME
        echo "Usage: $N {start|stop}" >&2
        exit 1
        ;;
esac

exit 0
```

这个 init.d 脚本基于 Debian GNU/Linux 系统/etc/init.d/skeleton 目录中的 init.d 框架。除了初始脚本之外，还要在每次启动服务器时把进程号（pid）放进/var/run/gameserver.pid 文件中。为此可以用 Perl 代码从 ps 命令的输出分析 pid。

对于 Windows 系统，可以把服务器应用程序变成 NT 服务，从而得到相同结果。可以用几个工具将 Java 服务器应用程序变成 NT 服务，其中一个是 http://wrapper.sourceforge.net，但用 Google 搜索 Java Windows service 可以得到更多其他选项。

下面介绍关于 Java 关闭挂钩的问题。

JDK 1.3 中可以增加关闭挂钩（shutdown hook）。关闭挂钩可以注册一个 Runnable 对象，在终止虚拟机时激活。这样就可以整洁地关闭服务器应用程序，可以保存游戏状态，优雅地切断用户，关闭数据库连接或其他打开资源，最后退出。

论文 Revelations on Java Signal Handling and Termination(IBM DeveloperWorks，www.ibm.com/developerWorks, January 2002)是个优秀的资料，作者 Chris White 在文中介绍了关闭挂钩及一般信号处理。

6.6.3　服务器管理控制台

另一个监视和调试游戏服务器的工具是管理控制台。管理控制台可以是本地或远程控制台。本地控制台像 RPSConsoleEventReader 一样实现，从标准输入读取命令并将输入变成事件。远程控制台是修改的 GameClient，与 RPSClient 相似，具有命令行或图形用户界面。

无论哪种情况，管理控制台和任何其他客户机一样向 GameControllers 发送 GameEvents。差别在于首先要求用户对管理员名单和可能的远程 IP 地址验证。管理客户机可能与正常 GameControllers 交流，也可能与特殊 AdminController 交流，将功能放在另一类中。

AdminController 可以向管理员提供几个特性，如服务器性能监视、游戏监查、日志浏览和远程关闭/重新启动。

为了增加安全性，可以监听另一端口的管理访问，然后设置相应防火墙规则，限制访问这个端口。然后可以将 GameServer 在这个端口配置另一 ServerSocketChannel，让其与主选择器相关联。

6.6.4 游戏监查

游戏监查和监视就是在游戏进行过程中进行监视，便于查错和分析游戏者行为。

实现这个特性时要将 GameClient 稍作修改，从服务器取得 2 个游戏者的动作，而不从用户取得游戏输入。另外，GameControllers 要改进，使管理客户机可以连接到游戏和开始接收这个游戏的所有事件。可以将 sendEvent()和 sendBroadcastEvent()方法稍作改变，首先检查所连接监视者的存在性，然后将游戏的所有事件多发一个拷贝给这个监视者。

6.7 高级课题

实际部署多方游戏系统时，还要解决另外一些问题，包括处理不良客户机连接，处理网络防火墙和进行性能调整与改进。

6.7.1 切断与重接

游戏服务器开发的主要难题是处理客户机切断问题。网络连接迫使游戏者玩不了游戏时，他们一定很生气。

切断连接的原因很多。如果客户机使用 Modem 连接，则很可能切断连接。在任何连接中，客户机与服务器之间可能发生连接拥塞。所以要尽快发现延滞或切断的连接，加以妥善处理。最好能让游戏者重新连接，从刚刚掉线的地方重新开始。根据游戏类别，有的可行，有的不可行，但至少要加以考虑。

1. Ping 事件

一个保护客户机连接的常用方法是在客户机与服务器之间定期发送 Ping 事件（也称为心跳事件）。Ping 事件不仅可以探测切断，而且可以测量客户机连接的锁存量。

客户机在一定时间内无法发送 Ping 事件时，可以认为客户机发生了连接问题。然后即可让监视这些事件的线程采取措施，如通知客户端游戏中的其他游戏者。

2. Reaper

实例框架目前允许客户机永久保持连接，这在闲机游戏者浪费宝贵服务器资源或堆满游戏门厅时可能造成问题。

为了避免不活动的客户机浪费服务器负载，可以在一定时间间隔后切断闲机客户机。为此可以让一个定期任务（Reaper，死神）检查客户机的最近操作时间，可能是客户机最近发出非 Ping 事件的时间。超过指定时间没有活动的客户机发送一个切断事件，关闭连接。可以用 java.util.Timer 或定制专用线程实现 Reaper。

6.7.2 HTTP 隧道

不久以前，可以在任何端口上运行服务器，大部分客户机都能连接。但是，如今随着 Internet 的发展，安全威胁不断增加，使系统管理员更加警惕。因此，大量公司、家庭用户和整个 Internet 访问提供者开始阻止访问所有非优先端口（大于 1024 的端口），甚至阻止常用端口，如 21(FTP)和 25(SMTP)。

结果只有少数端口是保证打开的，如 HTTP 端口（80）。因此，是否只能在 HTTP 端口（80）运行服务器？游戏客户机使用小程序时，需要用到这个技巧。对无签名小程序，网络访问只限于源服务器（下载小程序的服务器）。如果要从运行 HTTP 端口（80）的 Web 服务器下载小程序，那么如何也能让游戏服务器也在端口 80 运行呢？

更糟的是，许多防火墙进行分组过滤，保证端口 80 的通信流是合法的 HTTP 通信流，但是目前的游戏通信流根本不像 HTTP 协议。

此外，HTTP 是个无状态事务性协议。换句话说，基本流程是请求、响应、请求、响应等。客户机总是向服务器发送取信息的请求。但对于游戏服务器，要求服务器能在任何指定时候将信息推向客户机，要有双向的随时连接的管道。

别着急，这些问题是可以解决的，这里介绍 2 个办法，每种方法都要 2 个技巧。

第一是将通信包装成像合法 HTTP 通信。不用前面介绍的线上协议头，而把事件包装成 HTTP 请求和响应头。此外，利用 XML 和其他 ASCII 事件格式化使用 HTTP 需要更多调整，但必要时还是可以使用二进制事件负载。

第二个技巧是克服没有完全双工通信信道的问题。为此可以使用一对连接，都是从客户机启动的，都使用 HTTP1.1 保持活动（keep-alive）连接。把第一个称为服务器推连接，客户机打开第一个连接，向服务器发一个事件，然后只在服务器中有事件时才从中读取事件。把第二个称为客户机请求连接，让客户机将后面的所有事件发送到服务器。

这样就解决了分组过滤问题，但另一个问题是要在服务器通信的同一地址提供小程序文件和所有相关资源（图像、声音等）。这时有 2 个办法：可以实际从同一服务器提供这 2 种通信流，也可以让客户机以为是来自同一服务器。

方法 1:合并服务器。

第一种方法是让游戏服务器既作为 Web 服务器，也完成正常责任。收到客户机请求时，只要确定其是否游戏通信流。如果是，则要将其路由到正确的 CameController，否则要将其发送到处理文件服务的 DownoladController。

这个方法有几个问题。最大的问题是不够整洁。这里要编写的是游戏服务器，为什么要浪费时间编写 Web 服务器呢？Apache 服务器闭着眼睛都干得更好。

游戏开发员干了些什么？没什么，只要有钱，就可以买上一些现成的 URL 检查负荷平衡器，使硬件能探测收到的消息，相应路由消息。

方法 2：基于 URL 的负荷平衡。

如果不了解高端网络技术，基于 URL 的负荷平衡是目前许多防火墙和负荷平衡设备中使用的特性。这些设备可以根据几个准则路由来自单个 IP 地址和端口组合的通信流。URL 检查特性可以根据 HTTP 请求中指定的 URL 字符串内容进行路由。

下面是负荷平衡逻辑的伪代码：

```
if (URL_PATH matches "/GameServer*")
    originServer = "gameserver.hypefiend.com"
else
    originServer = "webserver.hypefiend.com"
```

因此，下列请求

```
GET /launchgame.html
```

或

```
GET /applet/gameapplet.jar
```

定向到 webserver.hypefiend.com，其可以在端口 80 运行标准 Apache 或其他 Web 服务器，可以保存所有游戏资源（Web 页面，图像、声音等）。

另一方面，下列请求定向到运行游戏服务器的 gameserver.hypefiend.com，这是专门的游戏服务器：

```
GET /GameServer/gamename=rps&playerid=bret
```

从客户端看（这里是 Java 虚拟机），这个过程是透明的。下载和游戏通信流好像都来自同一 IP 地址和端口。

一个好的负荷平衡器对任何大系统部署都很重要，因此这个硬件并不会增加成本。

读者可以将 GameServer 修改成用 HTTP 隧道而不是原始套接连接，基本体系结构和所有 GameController 代码都可以不动，只要进行下列修改即可：

- 让 SelectAndRead 识别 HTTP 头和从 HTTP POST 取得事件负载；
- 让 EventWriter 包装有 HTTP 头的响应；
- 将 GameServer 与 Player 接口修改成每个游戏者允许两个连接。

6.7.3 量化测试

为了实现处理多个同时用户的设计器，以及测试这个功能（而没有多数测试者），就要模拟大量用户的负载。为此，就要进化测试（bot testing）。

量化测试要生成一种客户机应用程序版本，现场模拟大量用户连接和玩游戏的情形。尽管这样无法精确模拟大量用户负载的各个方面，但可以测试几个关键性能、稳定性和持久性因素。量化测试期间要考察下面几点：

- 服务器支持的最大同时用户数；
- 不用同时连接级别的事件通过量（事件/秒）；
- 不用同时连接级别的连接延迟；
- 不用同时连接级别的 CPU 使用情况；
- 事件锁存（事件接收到开始处理之间的时间量）；
- EventQueue 备份日志（用于调整 Wrap 池长度）；

- 一段时间的内存使用情况，认真检查池漏连续增加的指标（在 Java 中是可能的）；
- 任何异常情况和其他错误；
- 任何线程死锁。

建议尽量多用几台机器进行量化测试。每台机器上的连接数越少，越接近模拟的真实用户，因为一台机器上多个 Bot 是顺序向服务器发送分组，而不是真正同时发送。

一定不要在 GameServer 所在机器上运行 Bot，否则会严重歪曲测试结果。

6.7.4 Modem 连接

如果目标用户主要通过 Modem 连接（有的只有 28.8 kbps），则无法通过本地网或连接条件良好的远程机 Bot 测试来模拟。Modem 连接锁存很高（仅仅访问 ISP 就可能要 250 毫秒），通过量低，分组容易丢失，而且经常掉线。

要模拟这类问题，一种方法是使用 Modem 仿真代理服务器。这类工具很多，可以用 Google 进行搜索，它们提供一个代理服务器，限制连接带宽和引入人为锁存。其中一个应用程序是 Sloppy（表示低速代理），在 www.dallaway.com/sloppy 站点。但它只能代理 HTTP 通信，而不能代理一般 TCP 通信流。因此，除非使用 HTTP 包装，否则要寻找另一工具。

代理服务器的能力有限，如果用户使用 Modem，最好使用用户所用硬件。操作系统、ISP 等经常和彻底进行测试。这可能很麻烦，但游戏者会感谢你的。

6.7.5 性能剖析与统计

为了优化应用程序，首先要评估服务器版本的效率和确定瓶颈所在。为此，要进行经验观察。大多数编程人员使用剖析工具，测量程序执行每个方法所花的时间量。Java 的运行环境中带有这样的剖析工具，还有一些第三方剖析工具。但是，通常要在这些剖析基础上增加一些定制代码，产生应用程序特定的性能统计信息。

例如，在 GameServer 中，了解哪些方法占用最多时间虽然有用，但我们感兴趣的关键指标是：

- 事件处理时间；
- 每个 GameEvent 类型的游戏逻辑时间；
- 事件锁存（事件处理之前在队列中等待的时间）；
- 数据库查询时间；
- 增加新连接的时间。

要取得这些性能统计信息，一个简单的方法是在关键方法前后插入一些代码，检查时间并记录到日志文件中，以便进一步分析。清单 6.14 显示了 RPSController 中 processEvent() 方法增加性能统计代码后的结果。

清单 6.14 processEvent()方法增加性能统计代码

```
public void processevent(event) {
long start = System.currentTimeMillis();

    switch (e.getType()) {
```

```
case GameEventDefault.C_LOGIN:
    login(e);
    break;
case GameEventDefault.C_LOGOUT:
    logout(e);
    break;
case GameEventDefault.C_JOIN_GAME:
    join(e);
    break;
case GameEventDefault.C_QUIT_GAME:
    quit(e);
    break;
case GameEventDefault.C_CHAT_MSG:
    chat(e);
    break;
case GameEventDefault.C_MOVE:
    move(e);
    break;
case GameEventDefault.C_GET_PLAYERS:
    getPlayers(e);
    break;
}
```

如果将 Log4J Appender 调整成只输出时间标志和提供的数据，则可以得到漂亮的逗号分隔值（CSV）文件，可以用 Excel 之类外部工具或一些 Perl 脚本进行分析。

6.7.6 性能调整

可以用多种方法调整服务器应用程序的性能。最重要是不要浪费时间优化不需要优化的代码。这好像是显然的，但通常以为是瓶颈的地方，常常并不是瓶颈。要知道哪里是真正的瓶颈，唯一的方法是通过剖析测量哪个代码实际浪费了最多 CPU 时间。

由于大多数服务器的工作是处理 GameEvents，所以处理内核系统时要注意 2 个简单的规则。

- 事件尽量小。事件越小，分配、序列化和传输所要的时间就越少；
- 发送的事件尽量少。每个事件都需要资源，因此如果客户机并不需要这个数据，就不要发送。例如，在 GameController.sendBroadcastEvent()方法中，不把事件发送到消息原创者。

1. 内存回收

缺省情况下，Java 虚拟机在后台运行内存回收线程，其定期触发，进行完整内存回收工作。还有另一种方法，称为增量内存回收，调用时要用-Xincgc 命令行开关。

增量内存回收避免了内存回收的短时间会促发服务器停滞，它频繁地运行内存回收单元，但每次的工作更少。结果就不会有突发工作，但内存回收单元的总体性能较差。

此外，JDK 1.4.1 还有一些试验性的内存回收选项：-XX:+UseConcMarkSweepGC 和 -XX+UseParallelGC。这些选项是非标准的，要慎用，但可能对应用程序性能有利，详见 http://java.sun.com/docs/hotspot/VmOptions.html。

2. 对象复用

就内存回收而言，避免内存回收的最佳方法是减少使用的对象数，为此可以用几个技术。

首先是基本优化，检查和删除不必要的分配。然后可以利用简单对象复用，如 RPSController 中的外走闲聊事件。但对于经常生成的对象，最好的办法是使用对象池。

对象池和线程池与数据库连接池相似，提供了对象复用机制。池中最初生成一定数量的某个类对象。当需要这个类的实例时，不是用 new 运算符生成对象，而是调用一个方法，从池中取实例。

用对象池可以使游戏服务器大大改善，特别是 GameEvents 对象。活动 GameServer 可能经常要生成 GameEvents，而这些事件的寿命又非常短。它们生成之后，发送到一个或几个客户机，然后删除。

将 GameEvents 建立池可以大大提高高负载游戏服务器性能。这里的难点是确定池中需要多少对象，但这可以在自动测试期间确定，也可以使池变成适配性的，用低优先级后台线程调整池的长度。许多 Web 和应用程序服务器都用这个方法管理请求处理线程池。

3. 其他调整

大多数游戏服务器还可以进行如下调整：

- 线程，减少线程数。性能统计信息有助于通过检查平均队列长度确定哪个 Wrap 需要更多或更少线程。每个线程都需要开销，不管其是否闲机，因此开始时尽量减少线程数；
- 同步，尽量紧凑，不需要同步的语句应放在同步块之外；
- 忙循环，避免使用忙等待循环，例如：

```
while (true) {
    checkForSomeCondition();
    try {
        Thread.sleep(SLEEP_TIME);
}
    catch(InterruptedException ie) { }
}
```

  等待时，用阻止队列之类的结构。执行定期任务时，使用 java.util.Timer；
- 日志，紧内循环中应删除所有日志语句，即使它什么也不登记。一定要检查活动代码中启用的日志，太多日志会使服务器性能下降。

6.8 小结

本章首先介绍如何使用 JDK 1.4 NIO API，如何建立多用户闲聊服务器。然后详细介绍了设计与实现灵活的游戏服务器框架，并利用它建立动人的快速动作游戏。现在读者应可自己建立多方游戏服务器了。

第二部分　三维图形与高级技术

第 7 章　三维图形
第 8 章　纹理贴图与照明
第 9 章　三维对象
第 10 章　用 BSP 树管理三维视景
第 11 章　碰撞探测
第 12 章　路径搜索
第 13 章　人工智能
第 14 章　游戏脚本
第 15 章　保存游戏

第 7 章　三维图形

关键问题

- 三维绘制类型
- 数学知识
- 三维基础
- 三维数学
- 多边形
- 三维变换
- 简单三维管道
- 镜头移动
- 实体与背面删除
- 扫描转换多边形
- 三维剪取
- 最后绘制管道
- 小结

曾经有人说过，地球上永远都不会有会飞的汽车。

我当然很失望，因为我很希望能驾驶它。在我看来，不是技术上无法制造会飞的汽车，而是大多数人只能控制车辆向前、向后、向左、向右，却无法控制汽车向上或向下，即只能控制汽车作二维运动（不要考虑爬坡和下坡，因为那必定还是在地面上），或许是人们感觉三维世界太复杂了。

的确，三维概念及三维图形都是让人难以琢磨的。但是，当学完本章后，读者将掌握足够的三维图形的基本知识，这样就可以制造自己的飞车了，虽然这只是在游戏中。

7.1 三维绘制的类型

常见的三维绘制类型有 2 种：光束跟踪和多边形造型。

光束跟踪（Ray tracing）起源于生活，实际上是建立光束的模型。但和现实生活相反，光束跟踪不是建立光源到眼睛的模型，而是建立眼睛到光源的模型。可以想像出，模型中代表每一个象素的光束的计算量将很大。尽管光束跟踪在实时三维图形中不太常用，但可以得到相当逼真的效果，像电影《冰川时代》就用了光束跟踪技术。

现实的游戏和基于计算机生成图像的电影通常会采用多边形造型技术。多边形造型（polygon modeling）把虚拟的三维世界解释为平面多边形。多边形是三边以上的平面闭合形体（如三角形、矩形、八角形），利用多边形可以大大提高速度，但球体之类的图形会很难变成多边形模型（除非用足够多个多边形来逼近）。多边形造型技术的计算量比光束跟踪要小得多，并且可用三维加速卡来提高运行的速度。游戏通常需要三维加速卡的支持，必定目前每台计算机上都配有三维加速卡。

Java 中有 2 种常见的利用三维加速卡的方法：Java3D 和 OpenGL 关联。

Java3D 属于高级 API，它可以实现隐藏表面消除、简单碰撞探测、视景管理等操作。编程人员可以快速地启动和运行用 Java3D，而不必知道太多的三维图形编程技术。尽管 Java3D 主要用 Java 编写，但其核心是用 OpenGL 或 DirectX 绘制三维图形。

Java3D 支持 Window 和许多版本的 UNIX，但 Java 1.4.1 SDK 并没有提供。另外，由于许可证的原因，不是所有系统都提供对 Java3D 的支持，目前 Mac OSX 就是这样。

另一方面，OpenGL 是免费的技术，可以在商业产品中使用，并且许多硬件都支持它，包括 Mac OSX。OpenGL 属于低级 API，所以使用时，要对三维图形编程有更多的了解。OpenGL 关联是使用 Java 代码来访问 OpenGL 功能的。

常见的 2 个 OpenGL 关联是 GL4Java(www.jausoft.com/gl4java.html)和 LWJGL(java-game-lib.sourceforge.net)。到本书编写时，更多的系统支持 GL4Java，还可以同时使用 AWT 和 Swing。但 GL4Java 已经有一年多的时间没有更新，所以其中可能没有最新的 OpenGL 特性。LWJGL 也称为轻量级 Java 游戏库，由于利用 NIO 自然缓冲区和 OpenGL 交互，所以速度更快。目前的 LWJGL 会得到经常的维护和更新，但与 AWT 和 Swing 并不兼容。

本章和下一章“纹理贴图与照明”就要生成基于软件的轻量级三维多边形绘制器。因为基于软件不如基于硬件那么快，所以不一定是大规模游戏的理想方案。在生成绘制器过程中，读者可以更好地了解基于硬件绘制器的工作。还将介绍许多三维图形方法，以及适用基于硬件的方案。

另外，如果要建立可下载的小游戏，而不要求用户安装或下载 Java3D、OpenGL 关联，则基于软件绘制器是个好方案。

7.2 回顾曾经学过的数学

在介绍三维图形编程之前，首先要复习一下向量和三角的一些知识。请读者专心听，因为它们处理三维图形所必要的知识。

7.2.1　三角学和直角三角形

如图 7-1 所示定义直角三角形角和边长的关系。

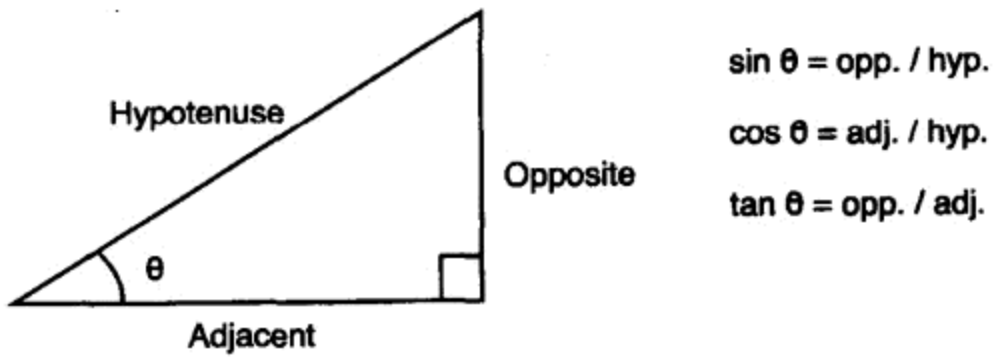


图 7-1　三角函数定义

许多三维图形问题都可以用直角三角形解决。要证明直角三角形或计算 3 个边和角的值，就要至少知道其中的 2 个信息：2 条边或 1 边 1 角。

也许你熟悉勾股定理：$a^2+b^2=c^2$，其中 a，b，c 是直角三角形的 3 条边，c 是斜边。如果知道三角形 2 边的边长，要求第 3 边的边长，使用勾股定理即可。

但是，如果只知道 1 边和 1 角，则要用如图 7-1 所示的三角函数定义。其中的正弦、余弦、正切和直角三角形的边、角都有一定的关系，可以在不同情形中解三角形。

7.2.2　向量数学

处理三维图形时，要把三维空间的一个点表示为三元组（x、y、z）。这里掌握的难点是，可以把三元组看成空间的一个点或一个向量。向量（vector）具有大小和方向，与速度相似，这是向量数学的基础，如方向向东，数量为 55mph。如果 x 轴指向东，则向量（55，0，0）表示向东速度 55mph。同样，如果 z 轴方向向南，则向量（0，0，55）表示向南速度 55mph。

本书只用到三维向量，但向量不一定只有三维，也可能是二维或多维的。本文用大写字母表示向量（而实际数字则用小写），另外，用下标表示向量的成分，如下所示。

$$V=(V_x, V_y, V_z)$$

记住向量有大小和方向，但没有原点，因为向量不管原点在哪里。换句话说，从洛杉矶向东跑 55mph 和从巴黎向东跑 55mph 是一样的（但巴黎可能用公里作为单位），两者是相同的向量。

下面要介绍一些向量数学的知识。首先是基本的加法，如图 7-2 所示。图中显示向量 U 与 V 相加（U+ V），V 与 U 相减（V -U）。图中还显示了向量-U 和-V，它们与 U 和 V 大小相同，但方向相反。

在数学看来，要将 2 个三维向量相加，就是把它们的成分加在一起。

$$U+V=(U_x+V_x \quad U_y+V_y \quad U_z+V_z)$$

例如，如果小鸟以 55mph 的速度向东飞，而风以 10mph 的速度要把小鸟向西推，则小鸟和风的向量相加，得到小鸟实际速度是向东 45mph。注意，2 个向量相加得到的新向量大小、方向或两者均不同。当然，减法也是一样的。此外，向量可以乘一个倍数，改变向

量大小而不改变其方向，向量乘法就是把倍数乘上每个坐标单位。

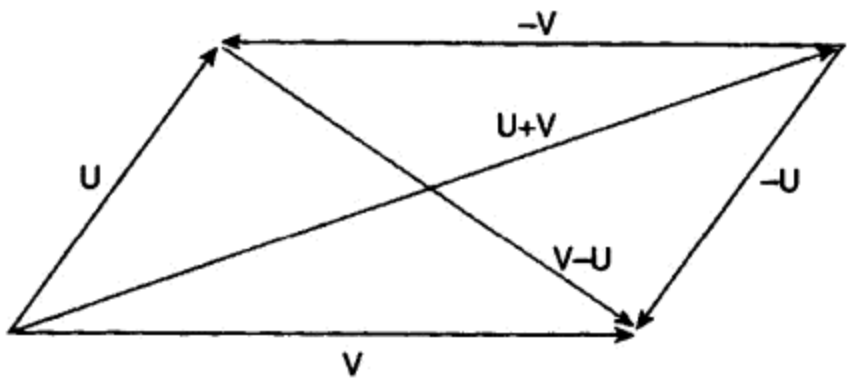


图 7-2　向量可以相加

$s\,V=(s\,V_x,\ s\,V_y,\ s\,V_x)$

因此，向东 55mph 的向量乘以 2 就是向东 110 mph。这个速度可是太快了！

向量的长度（或大小）用绝对值表示，可以用勾股定理的三维版本计算。

$|V|=(V_x^2+V_y^2+V_z^2)^{1/2}$

将长度为 1 的向量称为单位向量或范化向量，若要范化向量，将向量除以其长度即可，范化向量上方用一个折号表示。

$\hat{U}=U/|U|$

以上就简单介绍一些向量的数学知识，本章稍后还会介绍的更多，但现在要生成一个有用的向量类。如清单 7.1 所示的 Vector3D（浮点数描述三维向量）类。本书大多数的三维计算都要用浮点数，主要是它们通常比双精度数更经济，而且精度可以满足我们的要求。

清单 7.1　Vector3D.java

```
package com.brackeen.javagamebook.math3D;

/**
    Vector3D 类浮点数描述三维向量。可以把向量看成(x,y,z)点或从(0,0,0)到(x,y,z)的向量

*/
public class Vector3D implements Transformable {

    public float x;
    public float y;
    public float z;

    /**
        在 (0,0,0) 生成新的 Vector3D
    */
    public Vector3D() {
        this(0,0,0);
```

```
    }

    /**
        生成与指定 Vector3D 相同值的新 Vector3D
    */
    public Vector3D(Vector3D v) {
        this(v.x, v.y, v.z);
    }

    /**
        生成指定(x, y, z)值的新 Vector3D
    */
    public Vector3D(float x, float y, float z) {
        setTo(x, y, z);
    }

    /**
        检查这个 Vector3D 是否等于指定对象，即指定对象也是个 Vector3D 且两个 Vector3D 的
        x, y, z 光标均相等
    */
    public boolean equals(Object obj) {
        Vector3D v = (Vector3D)obj;
        return (v.x == x && v.y == y && v.z == z);
    }

    /**
        检查这个 Vector3D 是否等于指定 x, y, z 光标
    */
    public boolean equals(float x, float y, float z) {
        return (this.x == x && this.y == y && this.z == z);
    }

    /**
        将向量设置为与指定 Vector3D 相同值
    */
    public void setTo(Vector3D v) {
        setTo(v.x, v.y, v.z);
    }
```

```
/**
    将向量设置为指定(x, y, z) 值
*/
public void setTo(float x, float y, float z) {
    this.x = x;
    this.y = y;
    this.z = z;
}

/**
    将指定(x, y, z) 加速这个向量
*/
public void add(float x, float y, float z) {
    this.x+=x;
    this.y+=y;
    this.z+=z;
}

/**
    从这个向量减去指定(x, y, z)
*/
public void subtract(float x, float y, float z) {
    add(-x, -y, -z);
}

/**
    将指定向量加进这个向量
*/
public void add(Vector3D v) {
    add(v.x, v.y, v.z);
}

/**
    从这个向量减去指定向量.
*/
public void subtract(Vector3D v) {
    add(-v.x, -v.y, -v.z);
}
```

```
    /**
        将这个向量乘以指定值，向量新长度为 length()*s
    */
    public void multiply(float s) {
        x*=s;
        y*=s;
        z*=s;
    }

    /**
        将这个向量除以指定值，向量新长度为 length()/s
    */
    public void divide(float s) {
        x/=s;
        y/=s;
        z/=s;
    }

    /**
        用浮点数返回这个向量的长度
    */
    public float length() {
        return (float)Math.sqrt(x*x + y*y + z*z);
    }

    /**
        将这个 Vector3D 变成单位向量（即长度为 1 的向量）。等于调用 v.divide(v.length())
    */
    public void normalize() {
        divide(length());
    }

    /**
        将这个 Vector3D 变成字符串表示
    */
    public String toString() {
        return "(" + x + ", " + y + ", " + z + ")";
    }
}
```

上述代码很简单，Vector3D 类的方法主要实现对向量进行加、减、乘、除、取长度和范化等操作。

向量和空间的点都使用 Vector3D 表示，因为有些数学运行要把点当作向量，有些要把向量当作点，如取得空间 2 个点之间的向量。这时如果 2 个点都是 Vector3D 对象，则两者相减即可得到点之间的向量。

先前说过还要介绍更多的向量知识，因为 Vector3D 类还要增加更多的方法，详情见本章稍后的介绍，目前这里就先介绍这些三维知识。

7.3 3D 的基础

在三维向量中 x，y，z 的值实际指的是什么呢？

在进行二维图形工作时，通常使用和屏幕相同的二维坐标系统：原点在左上角，x 坐标从左向右增加，y 坐标从上向下增加。但三维图形中的 3 个轴并不能直接对应任何设备的坐标，所一我们要把三维世界变换成屏幕上的二维坐标系统。本书使用数学教材中常用的三维坐标系统——“右手”坐标系统：x 轴向右，y 轴向上，z 轴向里（远离浏览者），OpenGL 也使用这种坐标系统，如图 7-3 所示“右手”坐标系统。

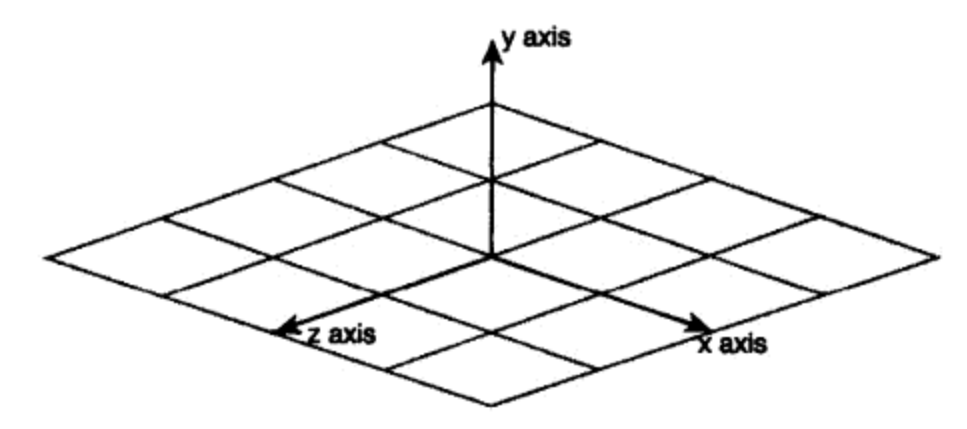


图 7-3 本书使用数学教材中常用的“右手”坐标系统

如果把右手的食指指向 y 轴方向，姆指指向 x 轴方向，则中指指向 z 轴方向，就构成了“左手”坐标系统，如图 7-4 所示。

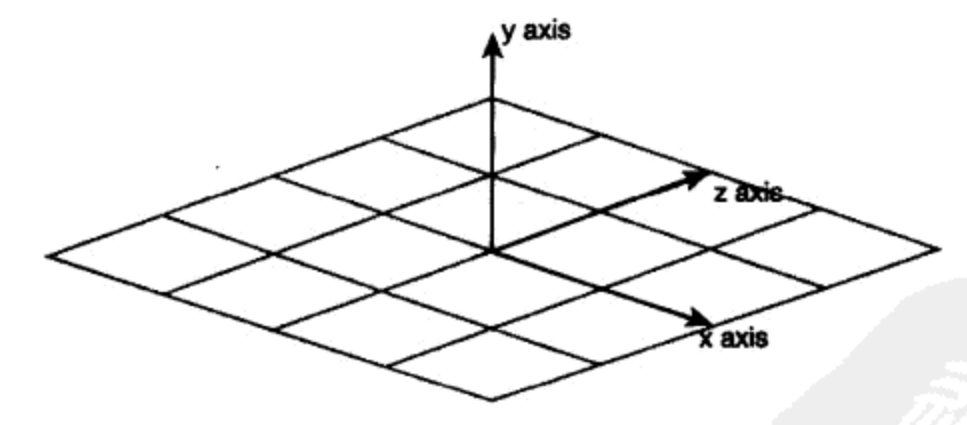


图 7-4 “左手”坐标系统是浏览三维世界的另一种方法

其他的三维坐标系统都是变形的（包括旋转）左手或“右手”坐标系统，本章只用如图 7-3 所示的“右手”坐标系统。下面我们将介绍镜头和视图窗口的概念。镜头（camera）是视图相对于世界上其他物体的位置。视图窗口（view window）是三维空间中与屏幕上三

维窗口大小相同的窗口。在游戏中，视图窗口通常和屏幕尺寸相同。

将镜头放在原点（0，0，0），视图窗口存在于 z 轴相反的方向，如图 7-5 所示，这样，z 轴的呈像深度将随着视图窗口的移动所改变，这是表示视图窗口常用的手段。

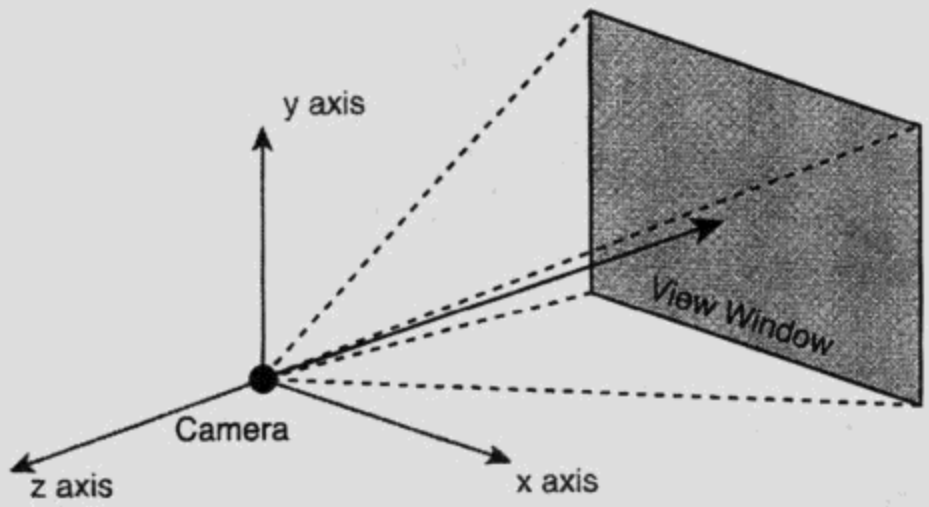


图 7-5　将镜头放在原点（0，0，0），视图窗口沿 z 轴相反方向向下，视图窗口通常是整个屏幕。镜头　视图窗口

将镜头放在原点（0，0，0）并沿着 z 轴看，好像有点奇怪。为什么不移动镜头，或改变镜头所对的方向？如向右转 90 度，朝 x 轴看不行吗？

答案是“差不多”。因为将镜头固定在原点（0，0，0），使视图窗口沿 z 轴方向改变来显示三维图形可以简化许多计算。本章稍后也将介绍模拟镜头的移动方法，但目前先介绍如何绘制固定镜头的三维世界。

镜头能看到的一切都在视图截头锥体（view frustum）内，如图 7-6 所示，视图截头锥体通常是 4 边的棱形，有时部分尖顶会被正截面切掉。但我们只要绘制视图截头锥体以内的部分即可。

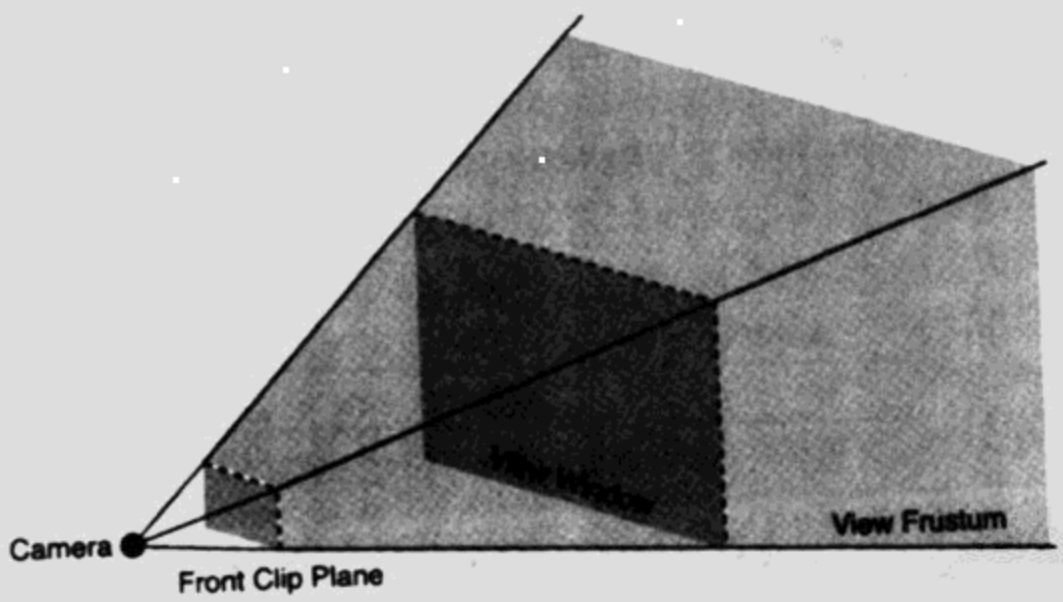


图 7-6　镜头能看到的一切都在视图截头锥体

下面要干什么呢？有了三维世界后，就要把视图截头锥体中的对象显示成二维图像，即对象越远表示越小，越近表示越大，而且要将实际绘制物体投影到视图窗口中，如图 7-7 所示。

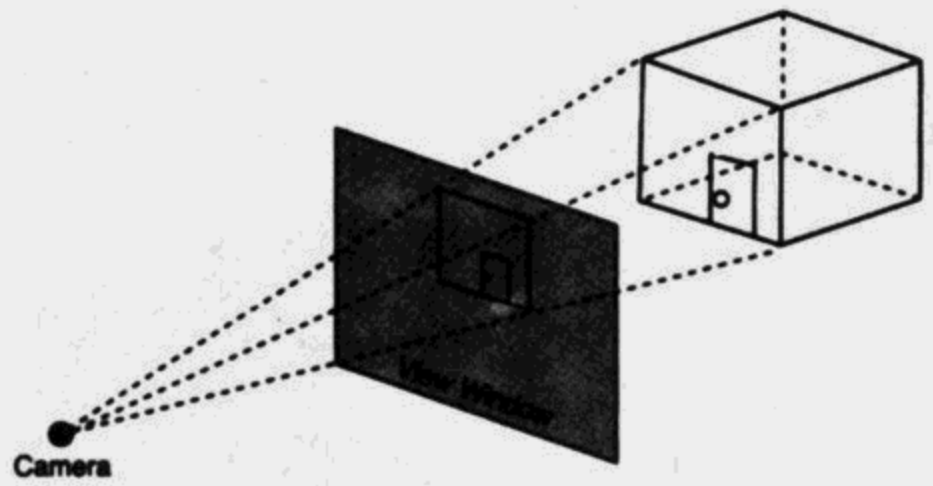


图 7-7 把三维物体投影到视图窗口中

当把三维物体投影到视图窗口中时，可以想像一条线从所画的三维点处连接到镜头上（如图 7-7 所示的虚线）。这条线和视图窗口的交点就是投影点的位置。由于把镜头看成点，因此对象越远越小，越近越大。这样就引出了下一个课题，要用数学方法计算如何把三维物体投影到视图窗口中。

7.4 3D 的数学

把三维物体投影到视图窗口中可以简化成直角三角形的问题，如图 7-8 所示。图中显示了三维点（x,y,z）投影是视图窗口的二维点（x'，y'）。

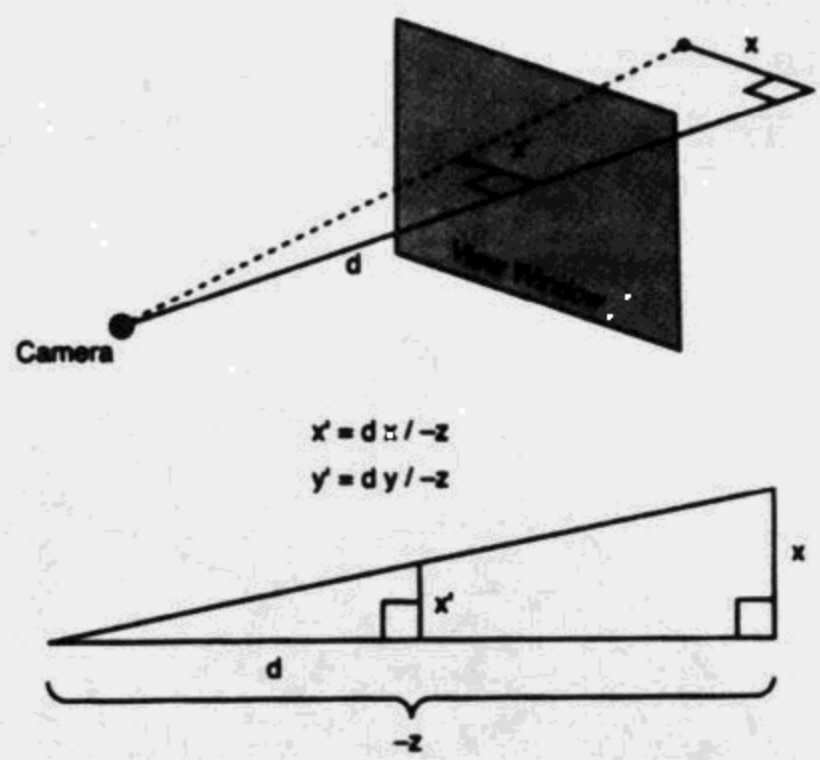


图 7-8 （x,y,z）的投影为视图窗口的二维点（x'，y'）

记住，由于 z 在视图方向上是减少的，所以三维点到原点之间的 z 轴长度是-z。图中处理 x 坐标和 y 坐标可以用同样的方法计算。

线段 x 和-z 构成的三角形与线段 x' 和 d 构成的三角形是相似三角形（角度相等）。若要解三角问题，就要知道从镜头到视图窗口的距离 d，我们称之为视图距离。

好在视图距离 d 可以选取任意值，通常它在游戏中是一个常量，其值只依赖于所要的

视角宽度，例如，如果镜头很接近视图窗口（视图距离短），则镜头可以看到三维世界中较宽的角度。同样，如果镜头很接远视图窗口（视图距离长），则镜头可以看到三维世界中较窄的角度，我们称之为隧道视图。

视角的大小是可选的，通常水平视角值在 45 度到 90 度之间，本书的实例都使用 75 度的视角。当有了水平视角之后，我们就可以计算视图距离，如图 7-9 所示。

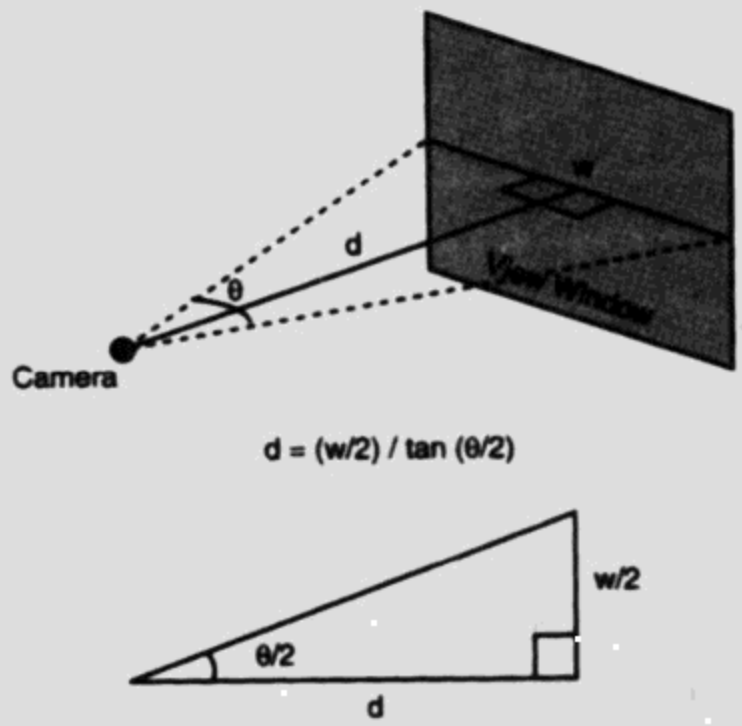


图 7-9　根据视角 θ 和视图窗口宽度 W 计算从镜头到视图窗口的距离

注意：视图距离依赖于窗口的大小，因此，如果让游戏者选择动态改变窗口或屏幕大小，则就要重新计算视图距离了。

知道视图距离后，可以用如图 7-8 所示的公式将一个点投影到视图窗口中。但是，视图窗口的二维坐标和屏幕的窗口不同，如图 7-10 所示，所以要增加一步，将视图窗口坐标换算成屏幕坐标。

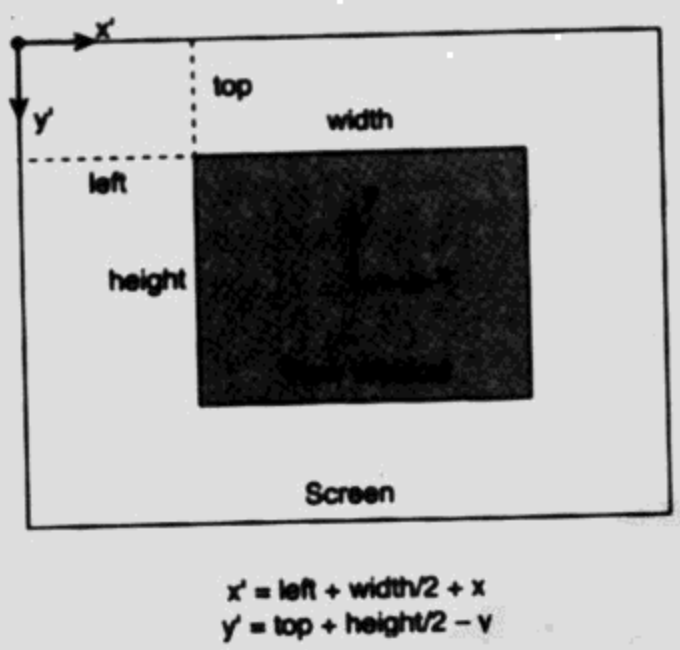


图 7-10　将视图窗口坐标换算成屏幕坐标

利用如图 7-10 所示的方程，就可以把三维点投影到屏幕上。清单 7.2 的 ViewWindow 类就汇总了视图窗口和投影功能。

清单 7.2 ViewWindow.java

```
package com.brackeen.javagamebook.math3D;

import java.awt.Rectangle;

/**
    ViewWindow 类表示三维视图的帧视图窗口几何
*/
public class ViewWindow {

    private Rectangle bounds;
    private float angle;
    private float distanceToCamera;

    /**
        用屏幕上的指定边界和水平视角生成新的 ViewWindow
        screen and horizontal view angle.
    */
    public ViewWindow(int left, int top, int width, int height,
        float angle)
    {
        bounds = new Rectangle();
        this.angle = angle;
        setBounds(left, top, width, height);
    }

    /**
        在屏幕上设置这个 ViewWindow 的边界
    */
    public void setBounds(int left, int top, int width,
        int height)
    {
        bounds.x = left;
        bounds.y = top;
        bounds.width = width;
        bounds.height = height;
        distanceToCamera = (bounds.width/2) /
            (float)Math.tan(angle/2);
```

```
}

/**
    设置这个 ViewWindow 水平视角
*/
public void setAngle(float angle) {
    this.angle = angle;
    distanceToCamera = (bounds.width/2) /
        (float)Math.tan(angle/2);
}

/**
    读取这个 ViewWindow 水平视角
*/
public float getAngle() {
    return angle;
}

/**
    取得这个视图窗口的宽度
*/
public int getWidth() {
    return bounds.width;
}

/**
    取得这个视图窗口的高度
*/
public int getHeight() {
    return bounds.height;
}

/**
    取得这个视图窗口在屏幕上的 y 偏移量
*/
public int getTopOffset() {
    return bounds.y;
}
/**
```

```
    取得这个视图窗口在屏幕上的 x 偏移量
*/
public int getLeftOffset() {
    return bounds.x;
}

/**
    取得这个视图窗口与镜头的距离
*/
public float getDistance() {
    return distanceToCamera;
}

/**
    将视图窗口中的 x 坐标换算为屏幕上的相应 x 坐标
*/
public float convertFromViewXToScreenX(float x) {
    return x + bounds.x + bounds.width/2;
}

/**
    将视图窗口中的 y 坐标换算为屏幕上的相应 y 坐标
*/
public float convertFromViewYToScreenY(float y) {
    return -y + bounds.y + bounds.height/2;
}

/**
    将屏幕中的 x 坐标换算为视图窗口上的相应 x 坐标
*/
public float convertFromScreenXToViewX(float x) {
    return x - bounds.x - bounds.width/2;
}

/**
    将屏幕中的 y 坐标换算为视图窗口上的相应 y 坐标
*/
public float convertFromScreenYToViewY(float y) {
    return -y + bounds.y + bounds.height/2;
```

```
    }

    /**
        将指定向量投影到屏幕上
    */
    public void project(Vector3D v) {
        // 投影到视图窗口
        v.x = distanceToCamera * v.x / -v.z;
        v.y = distanceToCamera * v.y / -v.z;

        // 换算为屏幕坐标
        v.x = convertFromViewXToScreenX(v.x);
        v.y = convertFromViewYToScreenY(v.y);
    }
}
```

代码用 project()方法实现投影功能，将 Vector3D 对象投影到 ViewWindow 对象中去，然后将三维坐标换算成屏幕坐标。调用 project()方法之后，Vector3D 对象的 x 和 y 就是屏幕坐标，而 z 可以忽略不计。

在将三维视图窗口坐标换算成屏幕坐标后，还要生成多边形建模器，下面还是先介绍一下多边形吧。

7.5　多边形

可以把实体多边形分成 2 大类：凸多边形和凹多边形，如图 7-11 所示。凹多边形的外廓有一部分朝内，象个凹槽。本章的代码只处理凸多边形，因为凹多边形比较复杂，不太好处理。当然，任何凹多边形都可以分解成一系列的凸多边形，所一游戏中要模拟凹多边形时，也没有什么问题的。

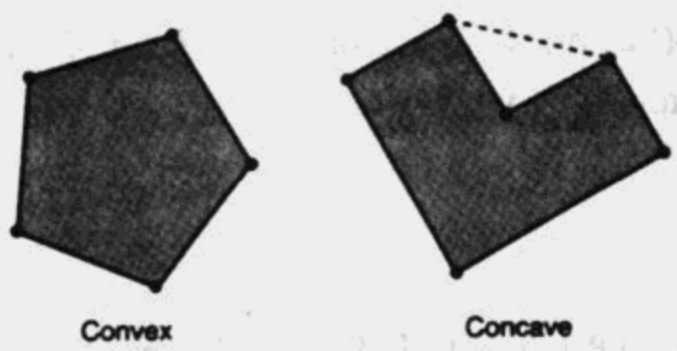


图 7-11　凸多边形和凹多边形。

从图中可以看出多边形的数学描述：多边形就是一系列点或顶点，只有画在屏幕上时才显示为实体。因此，首先生成如清单 7.3 所示的 Polygon3D 类，它目前什么也不干，只是管理一列的顶点。

清单 7.3 Polygon3D.java

```
package com.brackeen.javagamebook.math3D;
/**
    Polygon3D 类将多边形表示为一列顶点
*/
public class Polygon3D implements Transformable {

    // temporary vectors used for calculation
    private static Vector3D temp1 = new Vector3D();
    private static Vector3D temp2 = new Vector3D();

    private Vector3D[] v;
    private int numVertices;
    private Vector3D normal;

    /**
        生成立多边形，可以用作变样、投影等操作用的多边形
    */
    public Polygon3D() {
        numVertices = 0;
        v = new Vector3D[0];
        normal = new Vector3D();
    }

    /**
        用指定顶点生成新 Polygon3D
    */
    public Polygon3D(Vector3D v0, Vector3D v1, Vector3D v2) {
        this(new Vector3D[] { v0, v1, v2 });
    }

    /**
        用指定顶点生成新 Polygon3D，假设所有顶点在同一平面上
    */
    public Polygon3D(Vector3D v0, Vector3D v1, Vector3D v2,
        Vector3D v3)
    {
        this(new Vector3D[] { v0, v1, v2, v3 });
```

```
}
/**
    用指定顶点生成新 Polygon3D，假设所有顶点在同一平面上
*/
public Polygon3D(Vector3D[] vertices) {
    this.v = vertices;
    numVertices = vertices.length;
    calcNormal();
}

/**
    将这个多边形设置为与指定多边形具有相同顶点
*/
public void setTo(Polygon3D polygon) {
    numVertices = polygon.numVertices;
    normal.setTo(polygon.normal);

    ensureCapacity(numVertices);
    for (int i=0; i<numVertices; i++) {
        v[i].setTo(polygon.v[i]);
    }
}

/**
    保证这个多边形的容量足以放置指定个数的顶点
*/
protected void ensureCapacity(int length) {
    if (v.length < length) {
        Vector3D[] newV = new Vector3D[length];
        System.arraycopy(v,0,newV,0,v.length);
        for (int i=v.length; i<newV.length; i++) {
            newV[i] = new Vector3D();
        }
        v = newV;
    }
}

/**
    取得这个多边形的顶点数
```

```
*/
public int getNumVertices() {
    return numVertices;
}

/**
    取得指定索引处的顶点
*/
public Vector3D getVertex(int index) {
    return v[index];
}

/**
    将这个多边形投影到视图窗口
*/
public void project(ViewWindow view) {
    for (int i=0; i<numVertices; i++) {
        view.project(v[i]);
    }
}
```

在上述代码中，Polygon3D 对象用 Vector3D 数组表示三维多边形。和 Vector3D 一样，Polygon3D 也是刚刚开头，本章后面则要增加更多的特性。

缺省无变元构造函数生成没有顶点的空多边形。使用这个构造函数的多边形可以作为临时的多边形。临时多边形可以方便地用于复制、修改和保留原多边形。用 setTo()方法将临时多边形设置为要复制和修改的多边形。

Polygon3D 对象描述多边形的形状而没有描述其颜色和纹理。下一章将会介绍带纹理的多边形，目前只生成绘制实体颜色的多边形。SolidPolygon3D 类是 Polygon3D 的子类，它增加了读取和设置多边形颜色的方法。

现在就可以生成简单的三维演示了，但首先要看看三维的变换，它会使三维演示更加有趣。

7.6 三维变换

要让多边形在三维空间移动或其他的花样，就要对图形进行三维变换。多边形的三维变换通常包括平移、缩放和旋转：

- 平移就是在空间移动多边形，只要在多边形每个顶点上增加平移向量即可；
- 缩放就是缩小或放大多边形的尺寸（不改变其形状）。若多边形的中心在原点上将非常方便，只要将多边形每个顶点乘一个比例因子即可。若多边形中心不在原点上，则就要同时进行平移和缩放多边形的操作。因为三维引擎不用缩放，所以就

不准备介绍了；

- 旋转就是沿 x 轴、y 轴、z 轴或组合轴旋转多边形。

7.6.1　旋转

首先介绍旋转的计算，特别是沿 z 轴旋转。沿 z 轴旋转时，点的 z 值不变，因此相当于沿（0，0）点进行二维旋转，如图 7-12 所示。多边形上的每个点沿（0，0）点进行二维旋转，就是改变其位置，但不改变离（0，0）点的距离。

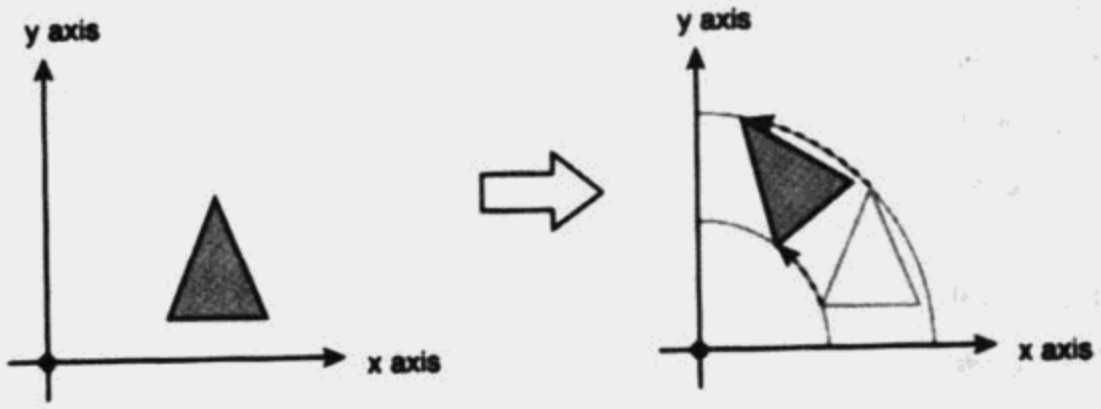


图 7-12　沿 z 轴旋转相当于沿（0，0）点进行二维旋转

如果把 x 和 y 换算成极坐标显示，则可以表示如下。

$x=r\cos\Phi$

$y=r\sin\Phi$

逆时针转一个角度 θ 之后，结果如下（如图 7-13 所示的实例）。

$x'=r\cos(\Phi+\theta)$

$y'=r\sin(\Phi+\theta)$

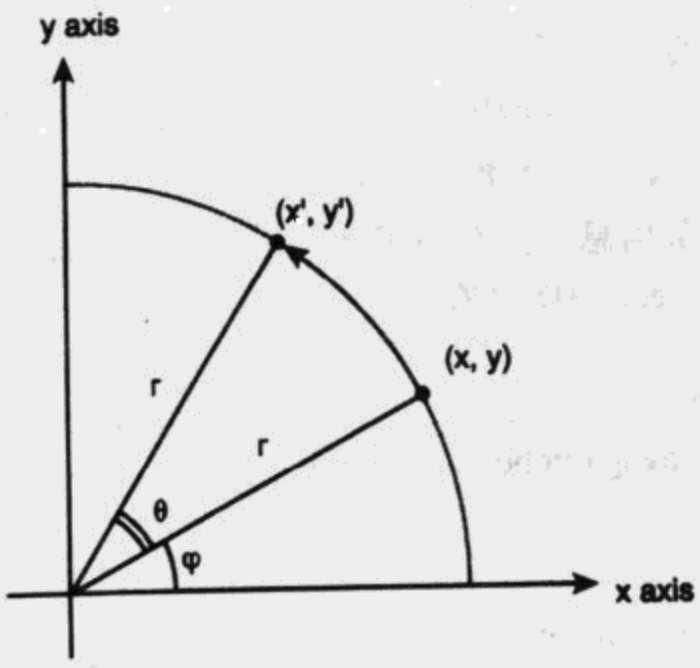


图 7-13　逆时针转一个角度 θ

可以简化成如下所示。

$x'=r\cos\Phi\cos\theta-r\sin\Phi\sin\theta$

$y'=r\sin\Phi\cos\theta+r\cos\Phi\sin\theta$

将公式代入 x 和 y 的公式，即可得到如下结果。

$x' = x\cos\theta - y\sin\theta$

$y' = x\sin\theta + y\cos\theta$

这个公式可以将一个点沿 z 轴逆时针转一个角度 θ 。沿 x 轴和 y 轴旋转时也差不多，下面就是所有旋转的公式。

沿 x 轴逆时针旋转。

$x' = x$

$y' = y\cos\theta - z\sin\theta$

$z' = y\sin\theta + z\cos\theta$

沿 y 轴逆时针旋转。

$x' = z\sin\theta + x\cos\theta$

$y' = y$

$z' = z\cos\theta - x\sin\theta$

沿 z 轴逆时针旋转。

$x' = x\cos\theta - y\sin\theta$

$y' = x\sin\theta + y\cos\theta$

$z' = z$

完整的旋转变换涉及 3 个旋转，分别沿 x，y 和 z 轴。变换多边形时，只要将这些公式应用到多边形的每个顶点上即可。

7.6.2 包装旋转和平移变换

尽管这些旋转公式很简单，但函数 Math.sin()和 Math.cos()的成本很高，实现时要使用大量计算和涉及 JNI 的调用，因此对每个要变换的 Vector3D 对象调用这些函数效率是很低的，所以可以在需要改变变换时预先计算每个角度的正弦和余弦值。由于每一帧最多进行一次变换，所以这样可以大大减少调用 Math.sin()和 Math.cos()的次数。

如清单 7.4 所示的 Transform3D 类，它可以实现这个功能。Transform3D 对象包含采用旋转和平移变换所需的全部信息，但并不实际采用变换。在后面采用变换时，可以用预先计算正弦和余弦值旋转的 Vector3D 对象。

清单 7.4 Transform3D.java

```
package com.brackeen.javagamebook.math3D;

/**
    Transform3D 类表示旋转和平移
*/
public class Transform3D {

    protected Vector3D location;
    private float cosAngleX;
```

```
private float sinAngleX;
private float cosAngleY;
private float sinAngleY;
private float cosAngleZ;
private float sinAngleZ;

/**
    生成新 Transform3D，没有旋转与平移
*/
public Transform3D() {
    this(0,0,0);
}

/**
    生成新 Transform3D，没有旋转，指定平移
*/
public Transform3D(float x, float y, float z) {
    location = new Vector3D(x, y, z);
    setAngle(0,0,0);
}

/**
    生成新 Transform3D
*/
public Transform3D(Transform3D v) {
    location = new Vector3D();
    setTo(v);
}

public Object clone() {
    return new Transform3D(this);
}

/**
    将 Transform3D 设置为指定 Transform3D
*/
public void setTo(Transform3D v) {
    location.setTo(v.location);
    this.cosAngleX = v.cosAngleX;
```

```
        this.sinAngleX = v.sinAngleX;
        this.cosAngleY = v.cosAngleY;
        this.sinAngleY = v.sinAngleY;
        this.cosAngleZ = v.cosAngleZ;
        this.sinAngleZ = v.sinAngleZ;
    }

    /**
        取得这个变换的位置（平移）
    */
    public Vector3D getLocation() {
        return location;
    }

    public float getCosAngleX() {
        return cosAngleX;
    }

    public float getSinAngleX() {
        return sinAngleX;
    }

    public float getCosAngleY() {
        return cosAngleY;
    }

    public float getSinAngleY() {
        return sinAngleY;
    }

    public float getCosAngleZ() {
        return cosAngleZ;
    }

    public float getSinAngleZ() {
        return sinAngleZ;
    }

    public float getAngleX() {
```

```
        return (float)Math.atan2(sinAngleX, cosAngleX);
    }

    public float getAngleY() {
        return (float)Math.atan2(sinAngleY, cosAngleY);
    }

    public float getAngleZ() {
        return (float)Math.atan2(sinAngleZ, cosAngleZ);
    }

    public void setAngleX(float angleX) {
        cosAngleX = (float)Math.cos(angleX);
        sinAngleX = (float)Math.sin(angleX);
    }

    public void setAngleY(float angleY) {
        cosAngleY = (float)Math.cos(angleY);
        sinAngleY = (float)Math.sin(angleY);
    }

    public void setAngleZ(float angleZ) {
        cosAngleZ = (float)Math.cos(angleZ);
        sinAngleZ = (float)Math.sin(angleZ);
    }

    public void setAngle(float angleX, float angleY, float angleZ)
    {
        setAngleX(angleX);
        setAngleY(angleY);
        setAngleZ(angleZ);
    }

    public void rotateAngleX(float angle) {
        if (angle != 0) {
            setAngleX(getAngleX() + angle);
        }
    }
```

```
    public void rotateAngleY(float angle) {
        if (angle != 0) {
            setAngleY(getAngleY() + angle);
        }
    }
    public void rotateAngleZ(float angle) {
        if (angle != 0) {
            setAngleZ(getAngleZ() + angle);
        }
    }

    public void rotateAngle(float angleX, float angleY,
        float angleZ)
    {
        rotateAngleX(angleX);
        rotateAngleY(angleY);
        rotateAngleZ(angleZ);
    }
}
```

7.6.3 采用变换

Transform3D 类中增加一些实际采用变换的方法，如清单 7.5 所示。

清单 7.5　Vector3D.java 的变换方法

```
/**
    用预先计算的转角正弦与余弦值将这个向量沿 x 轴转指定角度
*/
public void rotateX(float cosAngle, float sinAngle) {
        float newY = y*cosAngle - z*sinAngle;
        float newZ = y*sinAngle + z*cosAngle;
        y = newY;
        z = newZ;
    }
/**
    用预先计算的转角正弦与余弦值将这个向量沿 y 轴转指定角度
*/
    public void rotateY(float cosAngle, float sinAngle) {
        float newX = z*sinAngle + x*cosAngle;
        float newZ = z*cosAngle - x*sinAngle;
```

```
        x = newX;
        z = newZ;
    }

    /**
        用预先计算的转角正弦与余弦值将这个向量沿 z 轴转指定角度
    */
    public void rotateZ(float cosAngle, float sinAngle) {
        float newX = x*cosAngle - y*sinAngle;
        float newY = x*sinAngle + y*cosAngle;
        x = newX;
        y = newY;
    }

    /**
        在这个向量中增加指定交换，这个向量先旋转，后平移
    */
    public void add(Transform3D xform) {

        // 旋转
        addRotation(xform);

        // 平移
        add(xform.getLocation());
    }

    /**
        在这个向量中减少指定交换，这个向量先平移，后旋转
    */
    public void subtract(Transform3D xform) {

        //平移
        subtract(xform.getLocation());

        // 旋转
        subtractRotation(xform);
    }

    /**
```

```
        用指定交换角度旋转这个向量
        transform.
    */
    public void addRotation(Transform3D xform) {
        rotateX(xform.getCosAngleX(), xform.getSinAngleX());
        rotateZ(xform.getCosAngleZ(), xform.getSinAngleZ());
        rotateY(xform.getCosAngleY(), xform.getSinAngleY());
    }

    /**
        用指定交换角度反向旋转这个向量
    */
    public void subtractRotation(Transform3D xform) {
        //任意 sin(-x) == -sin(x)而 cos(-x) == cos(x)
        rotateY(xform.getCosAngleY(), -xform.getSinAngleY());
        rotateZ(xform.getCosAngleZ(), -xform.getSinAngleZ());
        rotateX(xform.getCosAngleX(), -xform.getSinAngleX());
    }
```

在上述代码中，注意 “增加” 和 “减少” 变换的方法。增加变换直接采用变换，而减少变换则逆向采用变换，不是转 θ 度和增加位置 V，而是转- θ 度和减去位置 V。本章稍后的自由镜头移动也要用到减法。

其实 Polygon3D 类中也有 add(Transform3D)和 subtract(Transform3D)方法，这些方法也变换多边形的每一个顶点。

7.6.4 旋转顺序

由于放置一次对一个轴进行，所以各个轴的旋转顺序将很重要，不同旋转顺序会得到不同最终的结果。当执行增加变换时，旋转的顺序从 x 到 z 到 y 轴。当执行减去变换时，旋转的顺序则按相反进行。顺序的选择是任意的，如在第一人称视角射击程序中，对三维视景中的走动可提供可预测的结果。

7.7 3D 的简单管道

下面就可以总结出绘制多边形简单的三维管道了。

（1）采用变换。

（2）将其投影到视图窗口。

（3）画图。

如清单 7.6 所示的 Simple3DTest 类，它实现三维数学、变换、投影并画出 2 个构成树的多边形。

清单 7.6　Simple3DTest.java

```
import com.brackeen.javagamebook.test.GameCore;
import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.geom.GeneralPath;

import com.brackeen.javagamebook.input.*;
import com.brackeen.javagamebook.math3D.*;

/**
    演示多边形绘制的三维测试
*/
public class Simple3DTest1 extends GameCore {

    public static void main(String[] args) {
        new Simple3DTest1().run();
    }

    //生成实体颜色多边形
    private SolidPolygon3D treeLeaves = new SolidPolygon3D(
        new Vector3D(-50, -35, 0),
        new Vector3D(50, -35, 0),
        new Vector3D(0, 150, 0));

    private SolidPolygon3D treeTrunk = new SolidPolygon3D(
        new Vector3D(-5, -50, 0),
        new Vector3D(5, -50, 0),
        new Vector3D(5, -35, 0),
        new Vector3D(-5, -35, 0));

    private Transform3D treeTransform = new Transform3D(0,0,-500);
    private Polygon3D transformedPolygon = new Polygon3D();
    private ViewWindow viewWindow;

    private GameAction exit = new GameAction("exit");
    private GameAction zoomIn = new GameAction("zoomIn");
    private GameAction zoomOut = new GameAction("zoomOut");
```

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

    InputManager inputManager = new InputManager(
        screen.getFullScreenWindow());
    inputManager.setCursor(InputManager.INVISIBLE_CURSOR);
    inputManager.mapToKey(exit, KeyEvent.VK_ESCAPE);
    inputManager.mapToKey(zoomIn, KeyEvent.VK_UP);
    inputManager.mapToKey(zoomOut, KeyEvent.VK_DOWN);

    // 把整个屏幕作为视图窗口
    viewWindow = new ViewWindow(0, 0,
        screen.getWidth(), screen.getHeight(),
        (float)Math.toRadians(75));

    //指定多边形颜色
    treeLeaves.setColor(new Color(0x008000));
    treeTrunk.setColor(new Color(0x714311));
}

public void update(long elapsedTime) {
    if (exit.isPressed()) {
        stop();
        return;
    }

    // 包装 elapsedTime
    elapsedTime = Math.min(elapsedTime, 100);

    // 沿 y 轴旋转
    treeTransform.rotateAngleY(0.002f*elapsedTime);

    // 让用户缩小/放大视图
    if (zoomIn.isPressed()) {
        treeTransform.getLocation().z += 0.5f*elapsedTime;
    }
    if (zoomOut.isPressed()) {
        treeTransform.getLocation().z -= 0.5f*elapsedTime;
```

```
        }

    }

    public void draw(Graphics2D g) {
        // 去背景
        g.setColor(Color.black);
        g.fillRect(0, 0, screen.getWidth(), screen.getHeight());

        // 画消息
        g.setColor(Color.white);
        g.drawString("Press up/down to zoom. Press Esc to exit.",
            5, fontSize);

        // 画树的多边形
        trandformAndDraw(g, treeTrunk);
        trandformAndDraw(g, treeLeaves);
    }
    /**
        在视图窗口投影和绘制多边形
    */
    private void trandformAndDraw(Graphics2D g,
        SolidPolygon3D poly)
    {
        transformedPolygon.setTo(poly);

        // 平移和旋转多边形
        transformedPolygon.add(treeTransform);

        // 将多边形投影到屏幕上
        transformedPolygon.project(viewWindow);

        // 将多边形变成 Java2D GeneralPath 并绘制
        GeneralPath path = new GeneralPath();
        Vector3D v = transformedPolygon.getVertex(0);
        path.moveTo(v.x, v.y);
        for (int i=1; i<transformedPolygon.getNumVertices(); i++) {
            v = transformedPolygon.getVertex(i);
            path.lineTo(v.x, v.y);
```

```
            }
            g.setColor(poly.getColor());
            g.fill(path);
        }
    }
```

在上述代码中，Simple3DTest1 对象画出的树可以沿 y 轴不断的移动，只要按“上/下”键即可让树移近或移远，如图 7-14 所示。

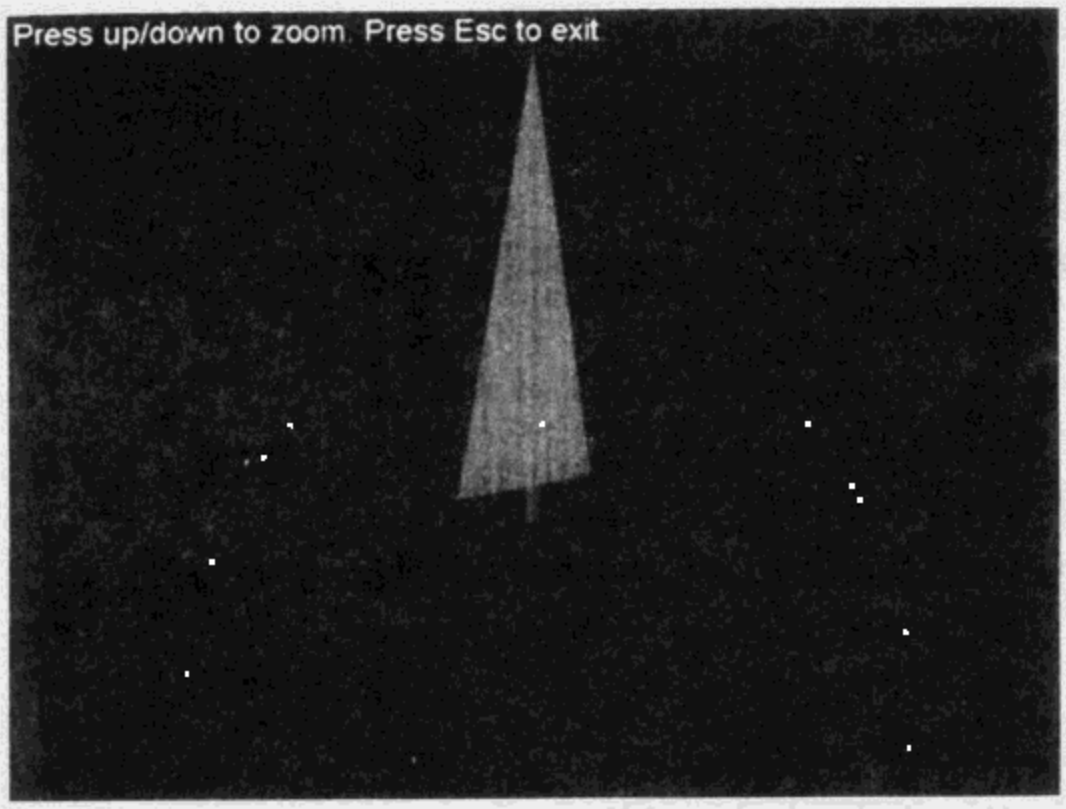


图 7-14　Simple3DTest1 的屏幕图形

Simple3DTest1 对象没有用普通的 Polygon3D 对象，而用其子类 SolidPolygon3D 对象。三维多边形在 transformAndDraw()中绘制，这个方法实现变换和投影多边形的操作。

要绘制多边形，首先要将其变成 GeneralPath（在 java.awt.geom 包中）。Graphics2D 接口提供了填充到 GeneralPath 的方法，这样才可以进行多边形的绘制。

其实树形多边形并不是直接变换的，而是复制到临时多边形中进行变换。如果直接变换，则随着时间的推移会不断累计浮点的误差。一个变换操作不会产生太明显的误差，但经多次变换后，误差将不断的累计，可能造成明显的扭曲变形。

Simple3DTest1 对象中还有几个问题需要注意：

- 镜头固定在原点（0，0，0）。尽管树可以沿 z 轴移近移远远，但我们还想自由移动镜头；
- 当树和镜头很近时，多边形的绘制方式很奇怪。如果放大太多（超过树），则会使树倒过来！这是因为多边形没有按视图截头锥体进行剪取；
- 虽然不明显，但有时多边形不能完全对齐，可以看到一些裂缝，原因有 2 个：第一，Graphics2D 缺省在 GeneralPath 中填入的顶点园整化为整数坐标，而不是保持浮点数坐标；第二，2 个相邻多边形靠在一起会形成 T 形接头，因为它们没有公共顶点。现在先别管 T 形接头，因为第 10 章“用 BSP 树进行三维场景管理要用

BSP 创建三维引擎”会解决这个问题；

- 最后，树是平台的，比较单调，若实现实体树和其他实体形状会更加有趣。

下面就要解决以上的问题，首先从自由移动镜头开始。

7.8　镜头移动

在 Simple3DTest1 对象中，对树形多边形进行变换，使树沿 z 轴移近或移远。实际上，你让移动的是树还是镜头呢？

其实这并不重要，因为从浏览者的角度看，树的相对距离的确在变即可。

下面不改变三维编程方法，来让镜头随意移动。用 Transform3D 对象表示镜头的位置，对视景中每个多边形采用这个变换的逆变换（减法）。但我们只是移动世界，而不是移动镜头。同样可以让镜头保持在（0，0，0）点，从而简化计算，而又给人以移动镜头的感觉。

作为镜头变换的实例，假设虚拟镜头在（10，50，500）点，向左看 5 度。要采用镜头变换，可以将每个多边形平移（-10，-50，-500）和向右转 5 度。换句话说，只要对每个多边形调用 subtract(Transform3D)方法即可。

沿每个轴旋转在三维坐标系统中提供下列效果：

- 沿 x 轴旋转像向上向下看；
- 沿 y 轴旋转像向左向右看；
- 沿 z 轴旋转像向前向后看。

需要提醒的是，在镜头代码中，要实际限制向上和向下看的程度。另外，镜头向前向后只是用来演示，实际三维游戏中通常不需要让镜头向前或向后的效果。

7.9　实体和背面删除

第一个演示 Simple3DTest1 按顺序画出清单中的每一个多边形。本实例中，这个方法没有任何不良影响，因为所画的平面树没有任何重叠多边形。

如果绘制实体形状（即具有前、后、左、右的一组多边形），则要用更好的算法确定按什么顺序画哪些多边形，否则会使后面的多边形出现在前面的多边形之前，远方的多边形出现在近处的多边菜之上，从而得到混乱的三维体验。

后面几章会介绍可见多边形顺序，但现在要先介绍消隐算法，即不要绘制隐藏在其他多边形后面的多边形或部分，因为镜头中是看不到这些内容的。

第一个表面消隐的方法称为背面消隐，也称为背面剔除。如图 7-15 所示背面消隐的基本思想：只画出对着镜头的多边形。

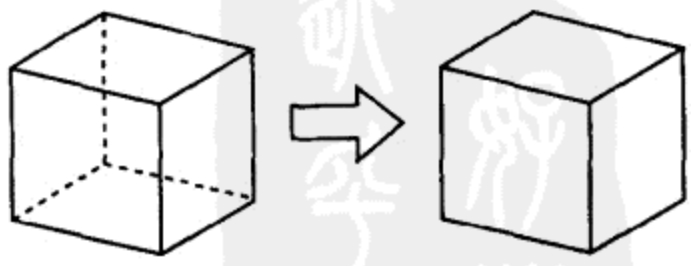


图 7-15　背面消隐

背面消隐平均可以将绘制的多边形减少一半。另外，背面消隐绘制凸多面体时不用增加任何工作。凸多面体是没有向内凹入面的多面体或三维多边形实体。

背面消隐并不是绘制所有多边形的完美方案。对于凹多面体和其他几种实体，要用到另一种算法，称为 z 缓存法，详情见第 9 章“三维对象”。

要确定多边形是否在背面，就要先介绍多边形法线的概念。多边形法线是和多边形垂直或正交的向量，法线的方向就是多边形所朝的方向。

如果法线和镜头方向的夹角小于 90 度，则表明多边形面对镜头，如图 7-16 所示。

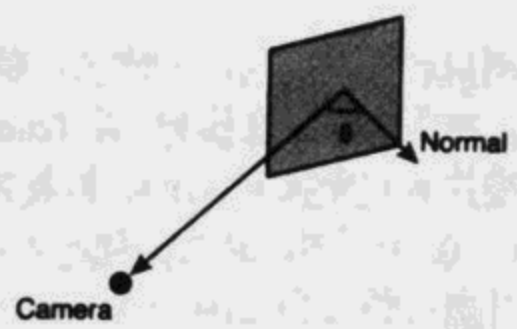


图 7-16　确定多边形面对镜头与否

7.9.1　点积

要计算和 2 个向量的夹角，则可以利用余弦定理，如图 7-17 所示。

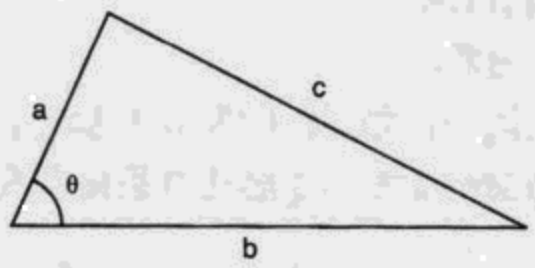


图 7-17　余弦定理

为了对向量采用余弦定理，不用长度 a，b，c，而用向量 U，V 和（U-V），从而把余弦定理变成如下的形式。

$$|U-V|^2=|U|^2+|V|^2-2|U|\ |V|\cos\theta$$

这里，结果的准确角度并不重要，只要知道其是否小于 90 度即可。如果小于 90 度（锐角），则角度的余弦大于 0。换句话说，$\cos\theta>0$，同样，$2|U|\ |V|\cos\theta>0$。下面用这个方法解余弦定理。

$$\begin{aligned}2|U|\ |V|\cos\theta &= |U|^2+|V|^2-|U-V|^2\\ &=|U|^2+|V|^2-(Ux-Vx)^2-(Uy-Vy)^2-(Uz-Vz)^2\\ &=|U|^2+|V|^2-Ux^2-Vx^2-Uy^2-Vy^2-Uz^2-Vz^2\\ &\quad +2UxVx+2UyVy+2UzVz\\ &=2UxVx+2UyVy+2UzV\\ |U|\ |V|\cos\theta &=UxVx+UyVy+UzVz\end{aligned}$$

这样虽然大大简化了，只要 3 个乘法和 2 个加法就可以确定 2 个向量之间的夹角是否为锐角。实际上，这不是什么新发明，这个方程称为点积，是一个基本向量算法，通常用一个点表示，例如。

$U.V = UxVx + UyVy + UzVz = |U|\ |V| \cos\theta$

利用点积方法，可以方便地判断 2 个向量之间的夹角是否锐角、钝角或直角：

- 如果（U.V<0）,则（$\theta > 90^0$）;
- 如果（U.V<0）,则（$\theta = 90^0$）;
- 如果（U.V<0）,则（$\theta < 90^0$）。

三维图形中经常用到点积技，所以可以在 Vector3D 类中增加一个 getDotProduct()方法，计算 2 个向量的点积。

```
/**
    返回这个向量与指定向量的点积
*/
public float getDotProduct(Vector3D v){
      return x*v.x +y*v.y + z*v.z;
}
```

回到背面消隐的问题，现在可以判断多边形法线和镜头向量之间的夹角是否锐角。下面就要求出多边形的法线。

7.9.2　交积

利用点积可以判断 2 个向量是否正交（成直角），即点积为 0，从而求出多边形的法线。

下面要求特定多边形的法线。首先取多边形的任意 2 个向量（U 和 V），这 2 个向量和多边形法线 N 的点积均为 0，则这样表示它。

U.N＝0

V.N＝0

进一步代入，可以用如下公式表示。

UxNx +UyNy+UzNz＝0

VxNx +VyNy+VzNz＝0

这个方程组有无穷解，但可以用行列或求出一个解（行列或是线性代数的内容）。

Nx=UyVz -UzVy

Ny=UzVx -UxVz

Nz=UxVy -UyVx

这样就求出了多边形的法线。此解称为交积，用叉号表示。

U×V＝　(UyVz － UzVy,UzVx －UxVz,UxVy - UyVz)

和点积一样，交积也经常使用，所以也把交积加进 Vector3D 类中。

```
/**
    将这个向量设置为两个指定向量的交积，指定向量之一可以是这个向量
*/
public float setToCrossProduct(Vector3D u Vector3D v){
    //在指定向量之一是这个向量时先解值局部变量
    float x =u.y * v.z － u.z * v.y;
        float y =u.z * v.x － u.x * v.x;
```

```
        float z =u.x * v.y - u.y * v.x;
        this.x = x;
        this.y = y;
        this.z = z;
    }
```

需要注意的是，法线只是和多边形正交的向量，那么如何确定法线的方向呢？它可能指向多边形朝向的面，也可以指向相反方向，因为这 2 个向量都和多边形正交。此外，多边形的哪一面是正面，哪一面是反面呢？

这取决于多边形中如何定义顶点。在右手坐标系统中，多边形正面是多边形顶点按逆时针方向出现的面，如图 7-18 所示。从另一面看，多边形顶点按顺时针方向出现。因此，定义多边形时，一定要保证，从正面看，多边形顶点按逆时针方向出现。

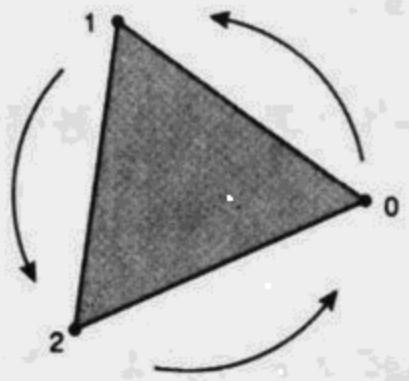


图 7-18 从正面看，多边形顶点按逆时针方向出现

然后要在 Polygon3D 类中增加一些计算法线的方法，法线是个成员变量，因此多边形每一个顶点是一个点。用向量 temp1 和 temp2 计算多边形中的 2 个向量。

```
    /**
        计算这个多边形的单位向量法线，这个方法用第一、第二和第三个顶点计算法线，如果这些顶
        点成一条直线，则这个方法行不通，要用 setNormal()方法显式设置法线。这个方法用 Polygon3D
        类中的静态对象进行计算，因此这个方法并非在 Polygon3D 的所有实例间都是线程安全的
    */
    public Vector3D calcNormal(){
        if (normal == null)
            normal   = new Vector3D
            }
                temp1.setTo(v[2]);
            temp1.subtract(v[1]);
            temp2. setTo(v[0]);
            temp2.subtract(v[1]);
            normal.setToCrossProduct(temp1 temp2);
            normal.normalize();
            return normal;
```

```
}
/**
    取得这个多边形的法线。如果任何顶点发生改变，则要用 calcNormal()
*/
 public Vector3D getNormal(){
 return normal;
 }
/**
    设置这个多边形的法线
*/
  public   void setNormal (Vector3D n){
   if (normal ==null){
   normal = new Vector3D(n)
 }
 else {
      normal.setTo(n)
 }
 }
```

在上述代码中，calcNormal()方法用第一、第二和第三顶点计算法线，如果这些顶点成一条直线，则方法就行不通了。下一章要介绍多边形的边界矩形，这时可以用边界矩形的法线。

最后，可以确定多边形是否朝着镜头。在 Polygon3D 中增加另一个方法 isFacing()，它检查多边形是否朝着某个点。

```
/**
    检查多边形是否朝着某个点，这个方法用 Polygon3D 类中的静态对象进行计算，因此这个方法
    并非在 Polygon3D 的所有实例间都是线程安全的
*/
pubic boolean isFacing(Vector3D u){
    temp1.setTo(u)
    temp1.subtract(v[0])
    return (normal.getDotProduct(temp1)>=0)
     }
```

在上述代码中，calcNormal()和 isFacing()使用临时静态 Vector3D 对象，使 Polygon3D 不是线程安全的，这是因为只在游戏主线程中调用这些方法。

7.9.3　再谈点积和交积

下面介绍点积和交积的几个属性，这些属性将在下一章用到。这里 A，B，C 是向量，S 是个实数。

点积：

- A.B = B.A;
- (sA) .B=s(A.B);
- A.B + A.C = A. (B+C)。

交积：

- A×B = -B×A;
- A×(B+C)=A×B+A×C。

标量三重积：

（A×B）.C=（C×A）.B=（B×C）.A。

向量三重积：

A×(B×C) =B（A.C）-C（A.B）。

7.10 扫描转换多边形

Simple3DTest1 中绘制多边形时，将投影多边形转换成 GeneralPath。可以看出，这样得到的结果并不完美，因为多边形顶点要园整化成最接近的整数。为了解决这个问题，可以生成定制扫描转换器，来保持浮点数顶点。扫描转换器将多边形变成水平扫描，使多边形可以画成一系列的水平线。后面的纹理贴图就要复用这个定制扫描转换器。

保持浮点数顶点好像意义不大，因为屏幕只能处理整数坐标。多边形最终只能显示整数边界，但还是要到最后时刻把浮点换算成整数。在缺省情况下，GeneralPath 将每个点园整化为最近的整数，然后再绘制多边形每个边。在绘制多边形的每个边后，将其转换成整数，这样就能更准确地绘制多边形每个边。

绘制多边形外形就像连点游戏，只在画出每个边或顶点到顶点的连线。在多边形中填上实体颜色初看起来比较复杂，为了简单起见，把它分解为几个小步，如图 7-19 所示，多边形的每行象素分解为水平扫描线。为了填充多边形，只要画出每个水平扫描线即可。

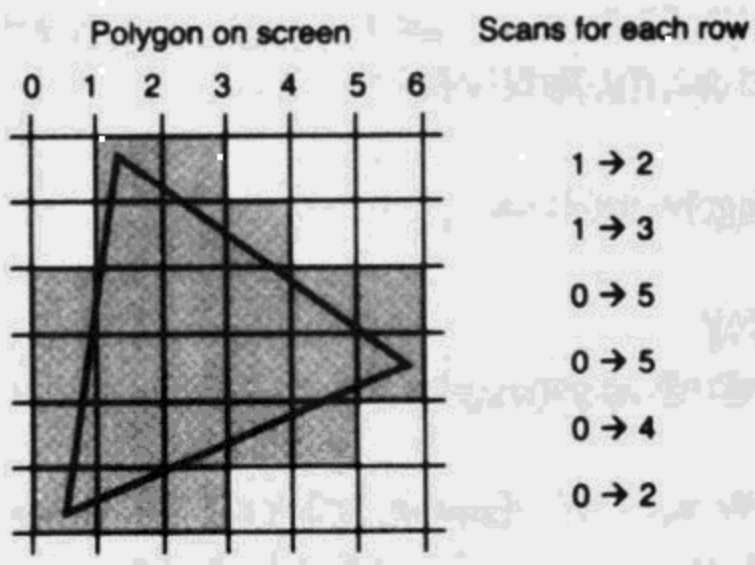


图 7-19 将多边形分解为水平扫描线

从图中可以看出，绘制多边形每个部分的每个象素，但如果 2 个多边形相邻呢？这时多边形共边，因此有些象素会画 2 次，这样会不知道哪个象素属于哪个多边形。

为了解决这个问题，就要区别每个多边形的左边和右边。多边形左边用 ceil()函数园整

化，而右边用 ceil()-1 园整化，如图 7-20 所示。

利用图 7-20 所示的方法，2 个相邻多边形就不会重叠了。另外，MoreMath 类中使用了定制 ceil()函数。

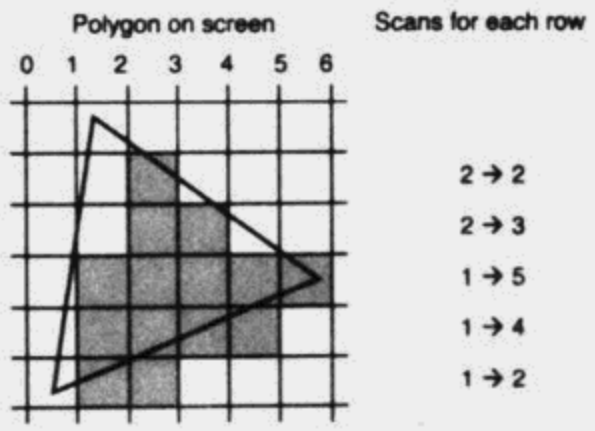


图 7-20　用更保守的方法将多边形分解为水平扫描线

```
public static int ceil (float f){
        if (f >0){
                return (int)f + 1
                }
                else{
                return (int)f
                }
 }
```

这个函数将浮点数换算成整数，Math.ceil()函数取双精度参数并返回双精度值，这样就不必进行双精度换算。下面生成一个简单 Scan 类，如清单 7.7 所示。

清单 7.7　Scan 类

```
/**
      水平扫描线
*/
public int Scan{
        public int left;
        public int right;
/**
      在 x 值超出边界时设置扫描的左右边界
*/
      public void setBoundary(int X){
              if (x < left){
                      left = x;
              }
              if (x-1 > right){
```

```
                right = x-1;
        }

    }

    /**
        清除这个扫描线
    */
    public void clear(){
        left = Integer.MAX_VALUE
        right = Integer.MIN_VALUE
    }

    /**
        确定这个扫描是否有效（左<=右）
    */
    public boolean isValid(){
            return (left <= right);
    }

    /**
        设置这个扫描
    */
    public void setTo (int left,int right){
            this.left = left;
            this.right = right;
    }

    /**
            检查这个扫描是否等于指定值
    */
    public boolean equals (int left,int right){
            return (this.left ==left && this.right == right);
    }
}
```

Scan 类是 ScanConverter 的内类。

每个扫描线都有自己的 Scan 对象，扫描转换器扫描多边形的每个边。每次扫描特定扫描线的边时，扫描转换器就调用 Scan 对象的 setBoundary()方法。方法可能改变扫描的左右边界，如果在同一行扫描 2 个不同边（边具有不同位置），则左边界小于右边界，因此扫描

有效，否则要有多个边才能生成有效扫描。

要实际扫描多边形的边，可以用直线方程。假设多边形的投影边由（x1,y1）和（x2,y2）点构成，对每条水平线（y），可以用如下方程求出相应的 x 值。

$(x - x1)/(x2 - x1) = (y - y1)/(y2 - y1)$

$x = (y - y1)(x2 - x1)-(y2 - y1) + x1$

ScanConverter 类使用这个方程，它还跟踪 ViewWindow(看看是否改变尺寸)，ViewWindow 中每个水平线的扫描清单和最后扫描多边形上下限值。ScanConverter 类的成员变量包括：

ViewWindow　View；

Scan[] scans；

int top；

int bottom。

如清单 7.8 所示，ScanConverter 类中扫描多边形的方法。

清单 7.8　简单多边形扫描换算

```
int minx = view.getLeftOffset();
int maxX = view.getLeftOffset() + view.getWidth()  - 1;
int minx = view.getTopOffset();
int maxY = view.getTopOffset() + view.getHeight()  - 1;

int numVertices = polygon.getNumVertices();
for (int i=0;i<numVertices,i++){
Vector3D v1 = polygon.getVertex(i);
Vector3D v2;
if (i == numVertices  -  1)
      V2 = polygon.getVertex(0);
}
else{
      v2 = polygon.getVertex(i+1)
}
//保证 v1.y < v2.y
if (v1.y > v2.y){
        Vector3D temp = v1;
        v1 = v2;
        v2 = temp;
}
float dy = v2.y -v1.y;

//忽略水平线
```

```
if (dy == 0){
    continue;
}
int startY = Math.max(MoreMath.ceil(v1.y),minY);
int endY = Math.min(MoreMath.ceil(v2.y)-1,maxY);
float dx = v2.x-v1.x;

float gradient = dx / dy;

//这个边进行扫描换算(直线方程)
for (int y = startY; y<=endY, y++)
    int x = MoreMath.ceil(v1.x (y - v1.y) * gradient);
    //保证 x 在视图边界中
    x = Math.min(maxX+1,Math.max(x,minX));
    scans[y].setBoundary(x);
}
}
```

ScanConverter 类将投影多边形换算为一系列水平扫描，并保证所有扫描在视图窗口中。注意 ScanConverter 只适用于凸多边形。凹多边形的有些边会向内凹，ScanConverter 则将其忽略。

下面是所有扫描进行迭代的方法。

```
int y = scanConverter.getTopBoundary();
while (y<= scanConverter.getTopBoundary()){
    scanConverter.Scan scan = scanConverter.getScan(y);
    if (scan.isValid()){
        g.drawLine(scan.left,y,scan.right,y)
    }
    y++
 }
```

用定点算法优化扫描转换

现在这个基本扫描转换器已经可以用了，但速度太慢些，这在屏幕上有许多多边形时会特别明显。

代码之所以不够快，一个原因是要对每个水平扫描线进行太多的浮点数运算。为了加速扫描转换，可以用整数代替浮点数，限制每个水平扫描线的运算量。

当然，整数无法取得理想精度，因此要比较巧妙地解释整数。为此，可以将每个浮点数乘以一个比例，得到整数范围的值。例如，比例为 1000 时，将 10.5 乘以 1000，得到整数值 10500。

选择正确的比例至关重要。如果用 2 的指数，则可以用移位方法在放大和未放大整数

间进行换算。

实例中使用 2^{16}，由于整数长度为 32 位，所以整数部分有 16 位，小数部分有 16 位，这个数字表示方法称为定点数，因为小数点在整数中某个地方固定。

下面首先介绍几个定义比例放大定点整数的常量。

```
final int SCALE_BITS = 16;
final int SCALE = 1<< SCALE_BITS
final int SCALE_MASK = SCALE - 1;
```

<<运行符是移位运行符，1<<16 等于 2^{16}。使用这些常量可以让代码进行定点数换数。

浮点数换算为定点数的公式如下。

```
int fixed = (int) (value * SCALE)
```

定点数换算为浮点数的公式如下。

```
float value = (float)fixed/ SCALE
```

整数换算为定点数的公式如下。

```
int fixed = value << SCALE_BITS;
```

定点数换算为整数的公式如下。

```
int value = fixed>> SCALE_BITS;
```

取得定点数整数部分的公式如下。

```
float   frca = (float)(fixed & SCALE_MASK)/ SCALE
```

2 个定点数相加减和普通加减一样，定点数和整数相乘也和普通乘法一样。但是，2 个定点数相乘除时则有所不同。2 个 32 位相乘会得到 64 位的结果，因此 2 个定点数相乘会使小数点后面的位数加倍。为了解决这个问题，就要进行 64 位乘法（长），以免消去需要的位。然后要移回去，得到正常的定点数。

```
int fixed = int (((long) fixed1 * fixed2)>> SCALE_BITS;
```

同样，除法也有类似问题，但更容易解决。

```
int fixed = int (((long) fixed1<< SCALE_BITS)/ fixed2;
```

代码中实际上不需要进行 2 个定点数的乘法和除法，这里只是顺便介绍一下。下面要利用这些定点数知识优化扫描转换器，实现代码如下所示。

```
int minX = view.getLeftOffset();
int minX = view.getLeftOffset() + view.getWidth() - 1;
int minY = view.getLeftOffset();
int minY = view.getLeftOffset() + view.getHeight() - 1;

int mumVertices = polygon.getNumVertices();
for (int i=0;i<numVertices;i++){
    Vector3D v1 = polygon.getVertex(i);
    Vector3D v2;
```

```
if( i== numVertices - 1){
    v2 = polygon.getVertex(0);
}
else{
    v2 = polygon.getVertex(i+1);
}
//保证 v1.y>v2.y
if(v1.y<v2.y){
    Vector3D temp = v1;
    v1= v2
    v2= temp
}
float dy = v2.y -v1.y;
//忽略水平线
if(dy==0){
    continue;
}
int startY = Math.max(MoreMath.ceil(v1.y),minY)
int endY = Math.max(MoreMath.ceil(v2.y),minY)
top = Math.min (top,startY)
bottom = Math.max(bottom,endY);
float dx = v2.x - v1.x;
//特殊情形：垂线
if (dx ==0){
      int x = MoreMath.ceil(v1.x);
      //忽略水平线
      x = Math.min(maxX+1, Math.max(x, minX));
      for (int y=startY; y<=endY; y++){
         scans[y].setBoundary(x);
      }
  }
}
else{
      //这个边进行扫描换算（直线方程）
       float gradient = dx / dy
      //整理直线开头
      float startX = v1.x + (startY - v1.y) * gradient;
      if (startX < minx){
            int yInt = (int)(v1.y + (minx - v1.x) /
```

```
                gradient);
            yInt = Math.min(yInt, endY);
            while (startY <= yInt){
                scans[startY].getBoundary(minY);
                startY++;
            }
        }
        else if (startX > maxX){
            int yInt = (int)(v1.y +(maxX  -  v1.x) /
                gradient);
            yInt = Math.min(yInt ,endY);
            while (startY <= yInt){
                scans [startY].setBoundary(maxX+1);
                        startY++
                }
        }
        if (startY >endY){
            continue;
        }
        //整理直线末尾
        float endX = v1.x + (endY  -  v1.y) * gradient;
        if (endY < minx){
            int yInt = MoreMath.ceil(v1.y + (minX  -  v1.x)/
                graidient);
            yInt = Math.max (yInt, startY)
            while (endY <= yInt){
                scans [endY].setBoundary(minX);
                        endY--
                }
        }
        else if (endX > maxX){
            int yInt = MoreMath.ceil(v1.y + (maxX  -  v1.x)/
                graidient);
            yInt = Math.max (yInt, startY)
            while (endY <= yInt){
                scans [endY].setBoundary(minX);
                        endY--
            }
        }
```

```
            if (startY > endY){
                continue;
            }
            //使用整数的直线方程
            int xScaled = (int) (SCALE * v1.x +
                SCALE * (startY - v1.y) * dx / dy) + SCALE_MASK;
            int dxScaled = (int)(dx * SCALE / dy);
            for (int y = startY; y<=endY; y++)
                scans[y].setBoundary(xScaled >> SCALE_BITS);
                xScaled+=dxScaled;
        }
        }
    }
```

当然，定点数没有和浮点数一样的精度、范围。由于扫描值可能超出定点数的表示范围，所以代码要在扫描的左右部分超出视图时进行整理。另外，不是每个点都重算交点，而是用斜度（dxScaled）增加每条线的 x 值（xScaled）。

完成的 ScanConverter 还没有完全优化，读者可以让它更快。方法是先实际将二维多边形剪取到视图内再进行扫描转换，以免扫描到屏外多边形或看不到的多边形，但这个 ScanConverter 也可以用。

7.11 三维剪取

Simple3DTest1 的最后一个修正是增加三维剪取功能。前面曾介绍过，树式多边形靠近镜头时的样子非常怪，好像镜头后面的树倒过来了。为了解决这个问题，就要用视图截面锥形剪取多边形。

视图截面锥形是 4 个平面构成的棱形，可以用每个平面剪取多边形，也可以用正截面剪取多边形，其余用二维剪取方法处理，下面就准备使用这种方法。

用平面剪取多边形时，会遇到 3 种情形：多边形完全在平面前面；多边形完全在平面后面；多边形与截面相交。如果在后面，则可以忽略多边形。如果相交，则要用截面剪取多边形，只绘制截面前面的多边形。用截面剪取凸多边形会得到另一个凸多边形，所以这个过程不会产生凹多边形。

如图 7-21 所示。多边形和截面相交，图中正截面在虚线上方。

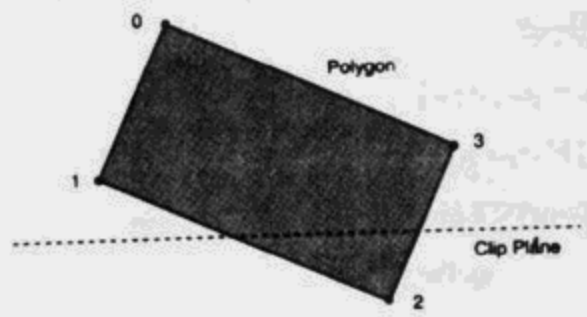


图 7-21 剪取，第 1 步：多边形与截面相交

下一步如图 7-22 所示，要在多边形边沿和截面相交处增加顶点。

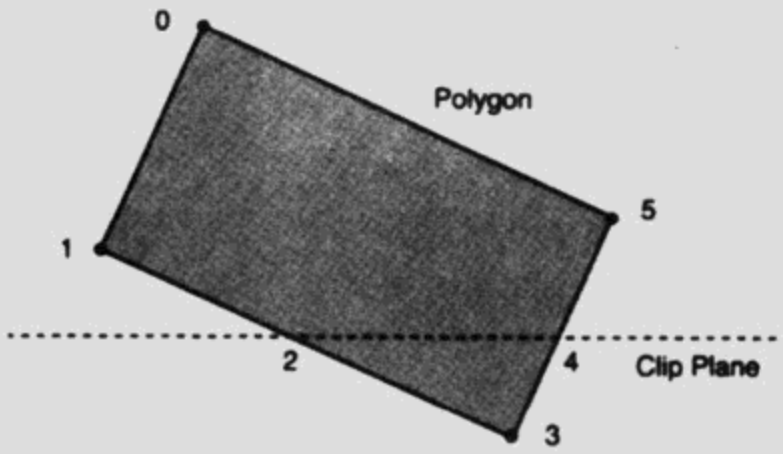


图 7-22　剪取，第 2 步：在多边形边沿与截面相交处增加顶点

这时多边形每边或者都在截面之前，或者都在截面之后。最后一步要删除截面后面的边。或删除截面后面的所有顶点，如图 7-23 所示。

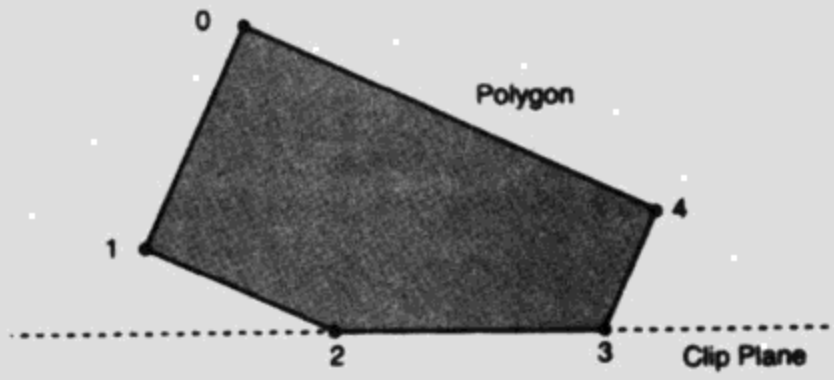


图 7-23　剪取，第 3 步：删除截面后面的所有顶点

由于正截面和视图窗口平行，所以只要选择截面位置即可。可以先选择 z=0，但这样可能造成除数为 0 的问题。本书的演示程序使用 z=-1。

要求出多边形边和截面的交点，可以用直线方程解决。已知截面 z 的设置，就可以得到求交点的方程。

x = (z - z1)(x2-x1)/ (z2 - z1)+x1

x = (z - z1)(y2-y1)/ (z2 - z1)+y1

z = z

剪取算法在 clip()方法中得以实现，将其加进 Polygon3D 类中。

```
/**
    剪取这个多边形，使其所有顶点在截面 clipZ 之前（换句话说，所有顶点的 z<=clipZ）
    ClipZ 值不能为 0，否则可能造成除数为 0 问题
    如果多边形至少有一部分在截面之前，则返回真值
 */
Public boolean clip (float clipZ){
    ensureCapacity (numVertices * 3);

    boolean isCompletelyHidden = true,
```

```
//插入顶点，使各边都在截面之前或之后
for (int i=0;i< numVertices;i++){
        int next = (i + 1) % numVertices;
        Vector3D v1 = v[i]
        Vector3D v1 = v[next]
        if (v1.z < clipZ){
              isCompletelyHidden = false
        }
        //保证 v1.z<v2.z
        if (v1.z>v2.z){
             Vector3D temp = v1
             v1 = v2
             V2 = temp
       }
       if (v1.z < clipZ && v2.z > clipZ){
            float scale = (clipZ- v1.z)/ (v2.z- v1.z)
            insertVertex(next,
                   v1.x + scale * (v2.z- v1.z)
                   v1.y + scale * (v2.y- v1.y)
                   clip Z);
            //跳过刚刚生成的顶点
            i++
       }
 }
 if (isCompletelyHidden){
        return false
 }

 //删除 z > clipZ 的所有顶点
for (int i=numVertices-1;i>=0;i--){
      if (v[i].z >clipZ){
            deleteVertex(i)
       }
}
return (numVertices >= 3)
/**
      在指定索引处插入新顶点
*/
protected void insertVertex (int index, float x, float y ,
```

```
        float z)
{
        Vector3D newVertex = v[v.length-1];
        NewVertex.x = x
        NewVertex.y = y
        NewVertex.z = z
        for (int i=v.length-1;i>index; i--){
                v[i]=v[i-1]
        }
        v[index]=newVertex;
        numVertices++
}

/**
     在指定索引处删除顶点
*/
protected void deleteVertex (int index){
        Vector3D deleted = v[index]
        for (int i= index; i< v.length-1; i++){
                v[i] = v[i+1]
        }
        v[v.length-1]= deleted;
        numVertices--
}
```

这里实现的剪取方法很简单，只是用垂直正截面剪取。此外，也可以用垂直背截面剪取（不绘制太远的多边形）。当然，用视图截面锥形剪取左、右、上下面也能减少扫描转换器的一些工作，但正截面剪取即可提供正确结果。

7.12　最后绘制管道

下面就要扩展 Simple3DTest，使用的简单三维管道，增加消隐、剪取和扫描转换功能。

（1）检查是否面对镜头。

（2）进行变换。

（3）剪取。

（4）投影到视图窗口。

（5）扫描转换。

（6）绘图。

为了最后用绘制管理进行绘图，生成如清单 7.9 所示的抽象类 PolygonRenderer。

清单 7.9 PolygonRenderer.java

```
package com.brackeen.javagamebook.graphics3D;

import java.awt.Graphics2D;
import java.awt.Color;
import com.brackeen.javagamebook.math3D.*;

/**
    PolygonRenderer 类是个抽象类，变换多边形并画到屏幕上
*/
public abstract class PolygonRenderer {

    protected ScanConverter scanConverter;
    protected Transform3D camera;
    protected ViewWindow viewWindow;
    protected boolean clearViewEveryFrame;
    protected Polygon3D sourcePolygon;
    protected Polygon3D destPolygon;

    /**
        用指定 Transform3D (镜头) 与 ViewWindow 生成新 PolygonRenderer。调用 startFrame()
        清除视图
    */
    public PolygonRenderer(Transform3D camera,
        ViewWindow viewWindow)
    {
        this(camera, viewWindow, true);
    }

    /**
        用指定 Transform3D (镜头) 与 ViewWindow 生成新 PolygonRenderer。如果调用
        clearViewEveryFrame 为 true 则调用 startFrame()清除视图
    */
    public PolygonRenderer(Transform3D camera,
        ViewWindow viewWindow, boolean clearViewEveryFrame)
    {
        this.camera = camera;
        this.viewWindow = viewWindow;
```

```
        this.clearViewEveryFrame = clearViewEveryFrame;
        init();
    }

    /**
        生成为扫描转换器和目标多边形
    */
    protected void init() {
        destPolygon = new Polygon3D();
        scanConverter = new ScanConverter(viewWindow);
    }

    /**
        取得这个 PolygonRenderer 使用镜头
    */
    public Transform3D getCamera() {
        return camera;
    }

    /**
        表示开始绘制帧，每个帧在绘制任何多边形之前调用这个方法
    */
    public void startFrame(Graphics2D g) {
        if (clearViewEveryFrame) {
            g.setColor(Color.black);
            g.fillRect(viewWindow.getLeftOffset(),
                viewWindow.getTopOffset(),
                viewWindow.getWidth(), viewWindow.getHeight());
        }
    }

    /**
        表示帧绘制结束。每个帧在绘制所有多边形之后调用这个方法
    */
    public void endFrame(Graphics2D g) {
        // 目前什么也不干
    }

    /**
```

```
        多边形变换与绘制
    */
    public boolean draw(Graphics2D g, Polygon3D poly) {
        if (poly.isFacing(camera.getLocation())) {
            sourcePolygon = poly;
            destPolygon.setTo(poly);
            destPolygon.subtract(camera);
            boolean visible = destPolygon.clip(-1);
            if (visible) {
                destPolygon.project(viewWindow);
                visible = scanConverter.convert(destPolygon);
                if (visible) {
                    drawCurrentPolygon(g);
                    return true;
                }
            }
        }
        return false;
    }

    /**
        绘制当前多边形，这时当前多边形已经变换、剪取、投影、扫描变换和显示
    */
    protected abstract void drawCurrentPolygon(Graphics2D g);
}
```

PolygonRenderer 类提供绘制多边形的框架，主要目的是在 draw()方法中实现多边形的绘制管道。它的任何子类都要实现 drawCurrentPolygon()方法，是实际绘制最终的扫描转换多边形。另外，绘制器可以先清除视图之后再画每个帧，只要调用 startFrame()方法。但需要先清除视图，因为实例中没有填充整个屏幕的多边形。对于三维视景，如果每个象素都被多边形覆盖，则不必先清除视图。

下一章“纹理贴图与照明”会对不同的多边形生成几种不同的 PolygonRenderer 的子类。由于目前只要处理实体颜色多边形，所以生成其子类 SolidPolygonRenderer，如清单 7.10 所示子类的 drawCurrentPolygon()方法。

清单 7.10　SolidPolygonRenderer. drawCurrentPolygon

```
protected void drawCurrentPolygon(Graphics2D g) {

    //设置颜色
    if (sourcePolygon instanceof SolidPolygon3D) {
```

```
            g.setColor(((SolidPolygon3D)sourcePolygon).getColor());
        }
        else {
            g.setColor(Color.GREEN);
        }

        // 绘制器扫描线
        int y = scanConverter.getTopBoundary();
        while (y<=scanConverter.getBottomBoundary()) {
            ScanConverter.Scan scan = scanConverter.getScan(y);
            if (scan.isValid()) {
                g.drawLine(scan.left, y, scan.right, y);
            }
            y++;
        }
    }
}
```

在上述代码中，drawCurrentPolygon()方法从 ScanConverter 的扫描表中绘制多边形。注意方法只在 draw()方法中调用，即这时当前多边形已经变换、剪取、投影、扫描变换和显示，并提供源多边形，可以取得任何需要的数据。

如果源多边形不是 SolidPolygon3D 类的实例（即多边形没有相关联的颜色），则多边形填上绿色。

下面要生成另一个程序，测试新的绘制器。这个程序叫 Simple3DTest2，生成的三维凸多面体样子很奇怪，像个房子，2 个正面多边形像门和窗，如图 7-24 所示。

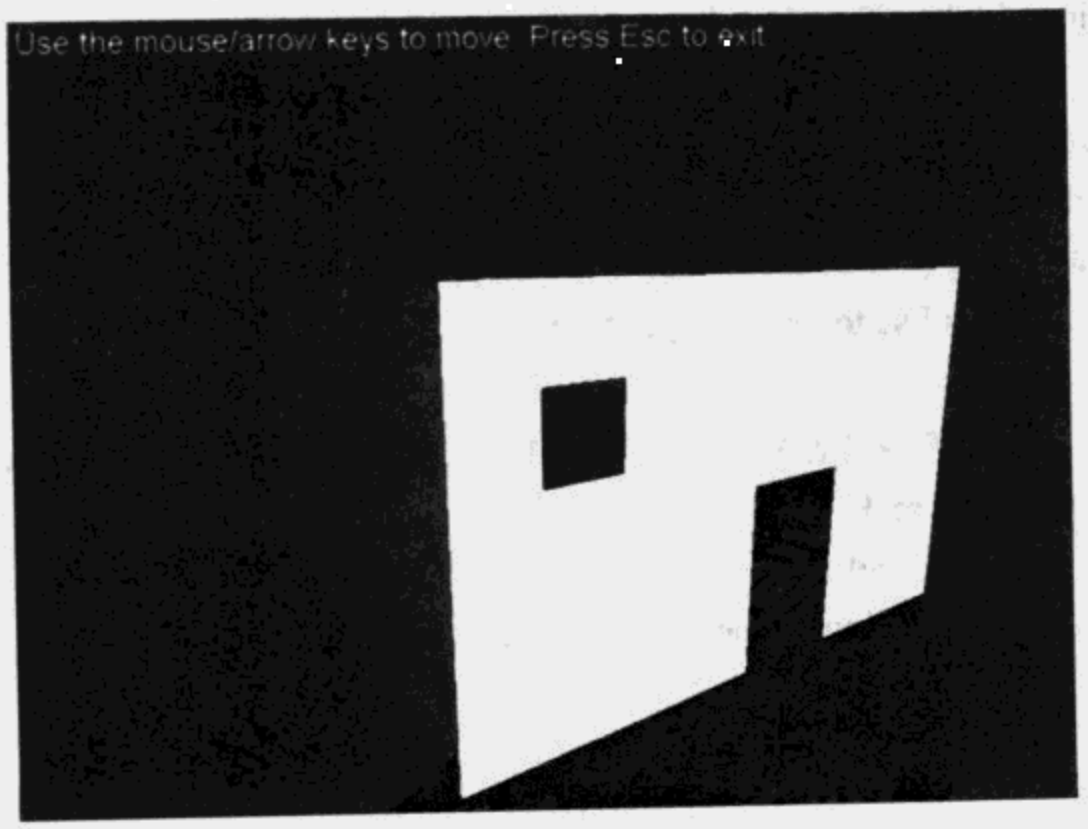


图 7-24　ple3Dtest2 的屏幕图形，包括剪取、消隐、定制多面体绘制器和自由镜头移动

Simple3DTest2 让用户在房子周围自由漫步，向上下左右观看和移动任何地方，包括屋内。注意，如果真的进入屋内，则什么也看不到，因为每个多边形都是背对镜头的。

整个 Simple3DTest2 代码很长（生成多边形就需要大量代码），这里只介绍相关的部分。Simple3DTest2 用 SolidPolygonRenderer 绘制多边形表中每个多边形。

```
public void draw (Graphics2D g) {

    //画多边形
     polygonRenderer.startFrame(g);
     for (int i=0;i<polygons.size();i++){
            polygonRenderer. draw(g,(Polygon3D) polygons.get(i));
     }
   polygonRenderer.endFrame(g);

   drawText(g)
}
```

Simple3DTest2 还绘制一些屏上指令，可以显示绘制器的帧速率：

```
    private boolean drawFrameRate = false;
    private boolean drawInstructions = true;

    private int numFrames;
    private long startTime;
    private float frameRate;

    ...
    public void drawText(Graphics g){
        //绘制文本
      g.setColor(Color.WHITE);
      if(drawInstructions){
     g.drawstring ( "Use the mouse/arrow keys to move." +
            "Press Esc to exit." ,5,fontSize);
      }
            //(为了更精确地测试，可能要关掉 ScreenManager 中的 BufferStrategy)
       if (drawFrameRate){
            calcFrameRate()
            g.drawstring(frameRate + "frame/sec" ,5,
            screen.getHeight() -5);
   }
}
public void calcFrameRate(){
```

```
        numFrames++;
        long currTime = System.currentTimeMillis();
    //每 500 毫秒计算帧速率
    if (currTime > startTime + 500){
        frameRate = (float) numFrames * 1000/
                (currTime - startTime)
        startTime = currTime
        numFrames = 0
    }
  }
```

由于每 500 毫秒计算帧的速率，即把帧数除以经过的时间量。另一种帧速率计算方法是计算动态平均，但不太适合当前帧速率，因为根据绘制器要绘制的内容多少，帧速率变化很大。

镜头变换在 Simple3DTest2 的 update()方法中更新。

```
public void update long elapsedTime){
    if (exit.isPressed()){
        stop();
        return;
    }
    //检查选项
    if (largerView.isPressed()){
        setViewBounds(viewWindow.getWidth() + 64
            viewWindow.getHeight() + 48);
    }
    else if (smallerView.isPressed()){
        setViewBounds(viewWindow.getWidth() - 64
            viewWindow.getHeight()- 48);
    }
    if (frameRateToggle.isPressed()){
        drawFrameRate = ! drawFrameRate;
    }
    //包括 elapsedTime
    elapsedTime = Math.min(elapsedTime, 100);
    float angleChange = 0.0002f*elapsedTime;
    float distanceChange = 05f*elapsedTime;

    Transform3D camera = polygonRenderer.getCamera();
    Vector3D cameraLoc = camera.getLocation();
```

```
//进行移动
if (goForward.isPressed()){
    cameraLoc.x -= distanceChange *  camera.getSinAngleY();
    cameraLoc.z -= distanceChange *  camera.getConAngleY();
}
if (goBackward.isPressed()){
    cameraLoc.x += distanceChange *  camera.getSinAngleY();
    cameraLoc.z += distanceChange *  camera.getConAngleY();
}
if (goLeft.isPressed()){
    cameraLoc.x -= distanceChange *  camera.getConAngleY();
    cameraLoc.z += distanceChange *  camera.getSinAngleY();
}
if (goRight.isPressed()){
    cameraLoc.x += distanceChange *  camera.getConAngleY();
    cameraLoc.z -= distanceChange *  camera.getSinAngleY();
}
if (goUp.isPressed()){
    cameraLoc.y -= distanceChange
if (goDown.isPressed()){
    cameraLoc.y -= distanceChange
}
//向上向下看（沿 x 轴旋转）
int tilt = tiltUp.getAmount() - tiltDown.getAmount();
tilt = Math.min(tilt, 200);
tilt = Math.max(tilt, -200);
//限制向上向下看的范围
float newAngleX = camera.getAngleX() + tilt * angleChange;
newAngleX = Math.max(newAngleX,( float)-Math.PI/2)
newAngleX = Math.min(newAngleX,( float)-Math.PI/2)

//转动（沿 y 轴旋转）
int turn = turnLeft.getAmount() - turnRight.getAmount();
turn = Math.min (turn,200)
turn = Math.max (turn,-200)
camera.rotateAngleY(turn * angleChange);

//左右转头（沿 z 轴旋转）
if (tiltLeft.isPressed()){
```

```
            camera.rotateAngleZ(10*angleChange);
        }
        if (tiltRight.isPressed()){
            camera.rotateAngleZ(10*angleChange);
        }
    }
```

update()方法取得键盘和鼠标的输入并会相应地移动镜头。使用第 3 章“交互性与用户界面”建立的鼠标观看特性让用户在三维世界中观看。另外，还要限制向上向下看的范围，使游戏者不会把头弯得太难受。

下面介绍演示程序的控件：

- 鼠标移动：左转右转，向上向下；
- 箭头键，W，S，A，D：前后、左、右移动；
- Page up/down，上下移动；
- Insert/delete，左右转头；
- +，增加视图窗口尺寸；
- -，减少视图窗口尺寸；
- R，显示帧速率；
- Esc，退出；
- +/-键可以在演示程序的全屏方式中运行速度太慢时减少视图窗口尺寸，加快运行速度。

```
    /**
        设置视图边界，在屏幕上居中显示视图
    */
    public void setViewBounds(int width, int height){
        width = Math.min(width, screen.getWidth());
        height = Math.min(height, screen.getHeight ());
        width = Math.max(64, width );
        height = Math.max(48, height );
        viewWindow.setBounds((screen.getWidth() - width /2),
            (screen.getHeight () - height /2, width, height);
    }
```

运行演示程序时，还要注意房子各边的阴影比其他地方深，这不是三维引擎的阴影特性，而是手工给这些多边形加深颜色的。下一章就要介绍实际照明和阴影。

7.13　小结

本章是三维图形的开头，还要有很多的三维知识要介绍，如纹理贴图，照明和视景管理，但这些课题要放到后面的三章介绍。

本章介绍大量的基础知识，如向量运行、定点运算、点积、交积、多边形法线、视图窗口、消隐、变换、投影、剪取、扫描变换和多边形填充。我们还用到了前几章介绍的代码，如全屏图形、键盘和鼠标输入（包括鼠标观看特性）。还生成了基本三维引擎框架，可供后面几章使用。

第 8 章　纹理贴图与照明

关键问题

- 透视正确的纹理贴图基础
- 简单纹理贴图器
- 优化纹理贴图
- 简单灯光
- 实现纹理照明
- 使用阴影图的高级照明
- 其他概念

小时候的超级任天堂游戏是我最渴望得到，它比原先的任天堂娱乐游戏要高级。因为它采用二维图形、彩色游戏和当时流行的其他特性。不久圣诞节到了，父母就在圣诞树下放了一套超级任天堂游戏。

后来母亲的朋友带来一个比我小几岁的儿子，所以我有责任让他玩得开心，于是就打开全新的游戏机，开始和他玩一般性的二维横向卷轴游戏。

游戏的视差滚动技术让我们非常着迷。画面中前面的树移动得很快，而后面的山却移得很慢，这些实体都有鲜明而真实的纹理。他从来没有见过这么精彩的游戏，于是摇头大喊“这是三维的！”

不可否认其实这不是三维的，它只是在二维视频上采用“模拟景深”技术，若模拟得非常好，这会让小男孩非常着迷的。我没有告诉他游戏其实不是三维的，因为我知道：只要是在电视或显示器等二维表面上绘制的游戏，就只能去模拟三维视景。实际上三维是观众自己体会到的。

这样就引出本章的主题：纹理贴图和照明。在第 7 章“三维图形”的演示程序中提出深度的概念，但屏幕上的一切都是由简单的具备实体颜色的多边形构成，看起来非常抽象。本章就要对多边形提供纹理和阴影效果，生成更逼真的体验。到本章结束时，你就可以运行演示说“这终于是三维了！”

具体地说，要生成基于软件的纹理贴图器和用阴影图实现灯光，这个技术在原先 Quake 等游戏中非常普及。尽管软件绘制器有缺点，但用 Java 实现的软件绘制器要比硬件绘制器具有几个优点，例如不依赖于硬件的 API 或考虑过时和有缺陷的三维驱动器等。此外，虽然建立三维纹理贴图引擎有些意见，但最终要用硬件绘制器在游戏中提供最高的性能。本章介绍的方法可以帮你更彻底地理解和控制任何三维绘制器，不管其是基于硬件或软件的绘制器。

8.1　透视正确纹理贴图的基础

本节的目的是要生成透视正确的纹理贴图器，它使多边形越远越小，纹理也越远越小。

注意纹理可能比贴图的实际多边形小。这时，纹理在多边形上平铺，如图 8-1 所示。

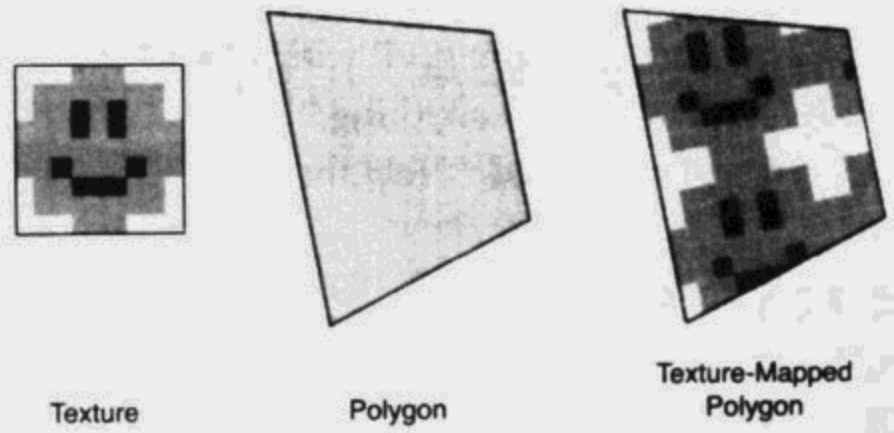


图 8-1　纹理可能比贴图的实际多边形小。这时，纹理在多边形上平铺

纹理中的像素通常称为图素（texel，是 texture element 的简称）。注意纹理是根据镜头与多边形的距离而缩放的，所以图素也会缩放。换句话说，视图窗口中的图素和像素通常不是尺寸相同的。

可以用 2 种方法进行纹理贴图：将每个图素贴图到一个像素；将每个像素贴图到一个图素。由于第一种方法太麻烦，所以有时纹理中并不是每个图素都要画（如从一个角度浏览纹理或从远处浏览纹理时）。另外，图素本身可以根据纹理的方向进行旋转和缩放。简单地说，如果将每个图素贴图到一个像素中，则每个图素画一个实体多边形，在此过程中还要进行一些重画（有些像素要画多次）工作。

可以将每个像素贴图一个图素，使视图窗口中每个像素只画一次。换句话说，对视图窗口中每个像素，都要寻找要画的相应图像。

记住目的后，绘制管道中进行纹理贴图的理想阶段是多边形扫描转换之后，就像上一章的实体填充多边形绘制器一样。

推导纹理贴图方程

Internet 上很容易找到纹理贴图的常用方程，但这里准备更详细地介绍方程的推导过程，使读者有更多的了解。下面首先介绍一些向量算法，如果不熟悉基本的向量算法，则可以看看上一章的介绍。

首先要定义多边形中纹理的朝向。要让纹理能够在多边形中任何地方偏移和旋转，所以要使用如下的向量（如图 8-2 所示）：

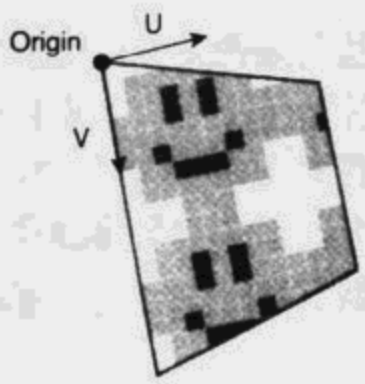


图 8-2　用原点和 2 个方向向量定义多边形中纹理的朝向

其中 O 表示原点；U 表示纹理 x 方向的向量；V 表示纹理 y 方向的向量。

在这个配置中，纹理可以朝向任何方向。另外，纹理原点不一定要在多边形边界之内。

下面要建立几个规则。让 U 和 V 向量相互正交（即夹角为直角），因为不希望表面的纹理发生扭曲。另外，还要保证 U 和 V 向量和多边形在同一平面上，否则 U 和 V 向量如果从多边形向外指，则会得到奇怪的结果，从正面看多边形纹理时会出现一个角度。

下面更详细地看看 O，U 和 V 向量和工作有什么联系。记住，要将每个像素贴图到一个图素。确定要画的图素之前，先要知道视图窗口中一个点在多边形上的具体点 P。下面跳过一步，假设已经有多边形上的点 P，要确定（x, y）纹理坐标（如图 8-3 所示）。

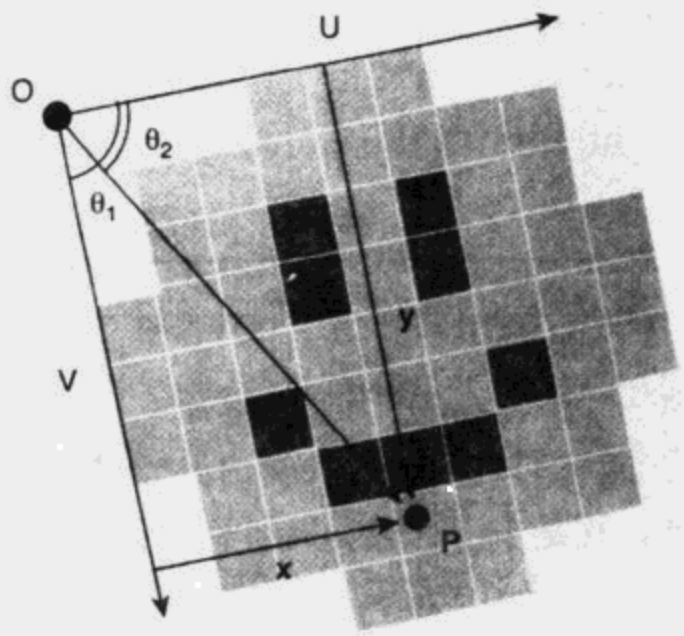


图 8-3　多边形上的点 P 与纹理方向的关系

从图 8-3 可以得到纹理中 x，y 的方程。

$x = |P-O| \cos\theta_1$

$y = |P-O| \cos\theta_2$

在方程中，|P-O|是从 O 到 P 的距离，θ_1是（P-O）和 V 之间的夹角，θ_2是（P-O）和 U 之间的夹角。

再进一步，如果 U 和 V 向量是单位向量，则可以用点积进行如下的简化。

$x = U \cdot (P-O)$

$y = V \cdot (P-O)$

记住（x, y）纹理坐标为负数或不在纹理边界中是正常的，因为要在多边形上平铺纹理。

下面在视图窗口中寻找一个点 W（如图 8-4 所示），已经知道 W 指向哪个多边形，因为在纹理贴图阶段，多边形已经投影到视图窗口。

倒过来看，W 只是 P 在视图窗口上的投影，可以用第 7 章的标准投影方程求解。

$W_x = d\ P_x / P_z$

$W_y = d\ P_y / P_z$

$W_z = d\ P_z / P_z$

W_z等于 d，是从镜头到视图窗口的距离。将其换算为向量形式，会得到如下的结果。

$W = (d / P_z)P$

由于要找到点 P，所以对 P 解这个公式。

$P = (P_z/d)W$

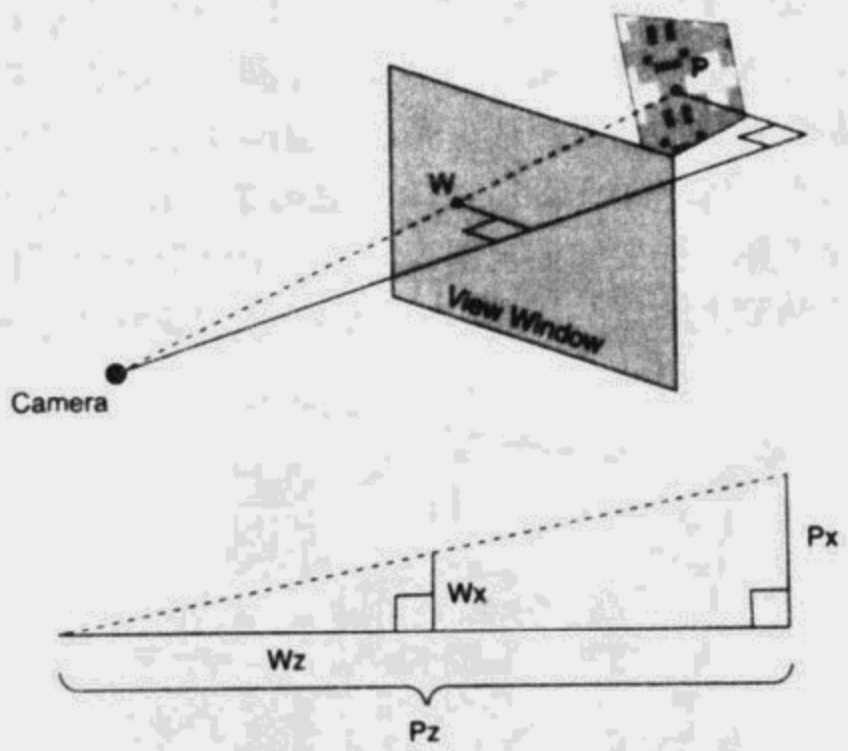


图 8-4 视图窗口中一个点 W 在多边形上的具体点 P

若要找到 P，就要寻找 P_z，这是 z 轴上从镜头到多边形的距离。

若要找到 P_z，就要多边形形式的平台信息。利用平面方程，平面的法线 N 和平面上任一直线是正交的。使用如图 8-3 的（P-0）线段，因为这个向量和多边形在同一平面上。

首先列出如下方程。

$N \cdot (P-0) = 0$

$N \cdot 0 = N \cdot P$

然后插入 P 的上述方程和 P_z 的解。

$N \cdot 0 = N \cdot ((P_z/d)W))$

$N \cdot 0 = (P_z/d)(N \cdot W)$

$P_z = d(N \cdot 0)/(N \cdot W)$

再将 P_z 插入上述公式和取得 P。

$P = W(N \cdot 0)/(N \cdot W)$

所以得到 x 和 y 的如下结果。

$x = U \cdot W(N \cdot 0)/(N \cdot W)-0)$

$y = V \cdot W(N \cdot 0)/(N \cdot W)-0)$

这样就有了一些纹理贴图方程。虽然方程很难看，但采用向量算法可以将其简化。

首先从 x 坐标开始。

$x = U \cdot (W(N \cdot 0) - 0(N \cdot W)/(N \cdot W)$

采用向量三重积定义 $A \times B \times C = A \times B \cdot C$。

$x = U \cdot (N \times W \times 0)/(N \cdot W)$

然后采用标量三重积定义 $A \cdot B \times C = A \times B \cdot C$。

$x = (U \times N \times W \cdot 0)/(N \cdot W)$

如果 $U \times N = -V$ 而 $N = U \times V$（多边形法线和两个纹理方向是正交的），则得到如下结果。

$x = (-V \times W \cdot O) / (U \times V \cdot W)$

最后得到 x 和 y 的如下结果。

$x = (V \times O \cdot W) / (U \times V \cdot W)$

$y = (O \times U \cdot W) / (O \times U \cdot W)$

这样就简单得多了，但还可以继续简化。注意对特定多边形进行纹理贴图时，绘制多边形时只有 W 改变，而向量 O，U，V 则不变，所以可以事先对每个多边形计算这些交积。

$A = V \times O$

$B = O \times U$

$C = U \times V$

这样，进行纹理贴图只需要如下的计算。

$x = A \cdot W / C \cdot W$

$y = B \cdot W / C \cdot W$

这样，有了一些纹理贴图方程，就可用其生成纹理贴图器了。

8.2　简单纹理贴图器

首先要描述纹理方向，生成如清单 8.1 所示的 Rectangle3D 类，它是三维空间中自由方向的矩形。换句话说，它不一定要像 java.awt.geom 中的 Rectangle2D 类一样，要和某个轴对齐。

清单 8.1　Rectangle3D.java

```
package com.brackeen.javagamebook.math3D;

/**
    Rectangle3D j 是三维空间中的自由方向矩形，定义一个原点和指向底（宽）与边（高）的向量
*/
public class Rectangle3D implements Transformable {

    private Vector3D origin;
    private Vector3D directionU;
    private Vector3D directionV;
    private Vector3D normal;
    private float width;
    private float height;

    /**
        在原点生成宽和高为 0 的矩形
    */
    public Rectangle3D() {
```

```
        origin = new Vector3D();
        directionU = new Vector3D(1,0,0);
        directionV = new Vector3D(0,1,0);
        width = 0;
        height = 0;
    }

    /**
        用指定原点和指向底（宽）与边（高）的向量生成新 Rectangle3D
        direction of the base (directionU) and direction of
        the side (directionV).
    */
    public Rectangle3D(Vector3D origin, Vector3D directionU,
        Vector3D directionV, float width, float height)
    {
        this.origin = new Vector3D(origin);
        this.directionU = new Vector3D(directionU);
        this.directionU.normalize();
        this.directionV = new Vector3D(directionV);
        this.directionV.normalize();
        this.width = width;
        this.height = height;
    }

    /**
        将这个 Rectangle3D 的值设置为指定 Rectangle3D
    */
    public void setTo(Rectangle3D rect) {
        origin.setTo(rect.origin);
        directionU.setTo(rect.directionU);
        directionV.setTo(rect.directionV);
        width = rect.width;
        height = rect.height;
    }

    /**
        取得这个 Rectangle3D 的原点
    */
    public Vector3D getOrigin() {
```

```
        return origin;
    }

    /**
        取得这个 Rectangle3D 的底方向。
    */
    public Vector3D getDirectionU() {
        return directionU;
    }

    /**
        取得这个 Rectangle3D 的边方向。
    */
    public Vector3D getDirectionV() {
        return directionV;
    }

    /**
        取得这个 Rectangle3D 的宽度
    */
    public float getWidth() {
        return width;
    }

    /**
        设置这个 Rectangle3D 的宽度
    */
    public void setWidth(float width) {
        this.width = width;
    }

    /**
        取得这个 Rectangle3D 的高度
    */
    public float getHeight() {
        return height;
    }

    /**
```

```
    设置这个 Rectangle3D 的高度
*/
public void setHeight(float height) {
    this.height = height;
}

/**
    计算这个 Rectange3D 的法线失量
*/
protected Vector3D calcNormal() {
    if (normal == null) {
        normal = new Vector3D();
    }
    normal.setToCrossProduct(directionU, directionV);
    normal.normalize();
    return normal;
}

/**
    取得这个 Rectangle3D 的法线
*/
public Vector3D getNormal() {
    if (normal == null) {
        calcNormal();
    }
    return normal;
}

/**
    设置这个 Rectangle3D 的法线
*/
public void setNormal(Vector3D n) {
    if (normal == null) {
        normal = new Vector3D(n);
    }
    else {
        normal.setTo(n);
    }
```

```
    }

    public void add(Vector3D u) {
        origin.add(u);
        // 不平移方向向量和长度
    }

    public void subtract(Vector3D u) {
        origin.subtract(u);
        // don't translate direction vectors or size
    }

    public void add(Transform3D xform) {
        addRotation(xform);
        add(xform.getLocation());
    }

    public void subtract(Transform3D xform) {
        subtract(xform.getLocation());
        subtractRotation(xform);
    }

    public void addRotation(Transform3D xform) {
        origin.addRotation(xform);
        directionU.addRotation(xform);
        directionV.addRotation(xform);
    }

    public void subtractRotation(Transform3D xform) {
        origin.subtractRotation(xform);
        directionU.subtractRotation(xform);
        directionV.subtractRotation(xform);
    }

}
```

简单地说，Rectangle3D 对象包括跟踪 3 个向量（原点、U 和 V）和矩形的法线。

下面生成如清单 8.2 所示的 SimpleTexturedPolygonRenderer 类，这是第 7 章生成的 PolygonRenderer 的简单子类，它只是添加了纹理贴图方程而已。

清单 8.2 SimpleTexturedPolygonRenderer.java

```
package com.brackeen.javagamebook.graphics3D;
import java.awt.*;
import java.awt.image.*;
import java.io.File;
import java.io.IOException;
import javax.imageio.ImageIO;
import com.brackeen.javagamebook.math3D.*;

/**
        SimpleTexturedPolygonRenderer 类演示了透视正确纹理贴图的基础，速率很慢，对每个多边形
        贴图相同纹理
*/
public class SimpleTexturedPolygonRenderer extends PolygonRenderer
{
    protected Vector3D a = new Vector3D();
    protected Vector3D b = new Vector3D();
    protected Vector3D c = new Vector3D();
    protected Vector3D viewPos = new Vector3D();
    protected Rectangle3D textureBounds = new Rectangle3D();
    protected BufferedImage texture;

    public SimpleTexturedPolygonRenderer(Transform3D camera,
        ViewWindow viewWindow, String textureFile)
    {
        super(camera, viewWindow);
        texture = loadTexture(textureFile);
    }

    /**
        从文件装入纹理图像，所有多边形使用这个图像
    */
    public BufferedImage loadTexture(String filename) {
        try {
            return ImageIO.read(new File(filename));
        }
        catch (IOException ex) {
            ex.printStackTrace();
```

```
            return null;
        }
    }
    protected void drawCurrentPolygon(Graphics2D g) {

        //计算纹理边界。最好预先计算纹理边界并和多边形一起存放。为了便于演示，这里计算坐
标
        Vector3D textureOrigin = textureBounds.getOrigin();
        Vector3D textureDirectionU = textureBounds.getDirectionU();
        Vector3D textureDirectionV = textureBounds.getDirectionV();

        textureOrigin.setTo(sourcePolygon.getVertex(0));

        textureDirectionU.setTo(sourcePolygon.getVertex(3));
        textureDirectionU.subtract(textureOrigin);
        textureDirectionU.normalize();

        textureDirectionV.setTo(sourcePolygon.getVertex(1));
        textureDirectionV.subtract(textureOrigin);
        textureDirectionV.normalize();

        // 变换纹理边界
        textureBounds.subtract(camera);

        // 开始纹理贴图计算
        a.setToCrossProduct(textureBounds.getDirectionV(),
            textureBounds.getOrigin());
        b.setToCrossProduct(textureBounds.getOrigin(),
            textureBounds.getDirectionU());
        c.setToCrossProduct(textureBounds.getDirectionU(),
            textureBounds.getDirectionV());

        int y = scanConverter.getTopBoundary();
        viewPos.z = -viewWindow.getDistance();

        while (y<=scanConverter.getBottomBoundary()) {
            ScanConverter.Scan scan = scanConverter.getScan(y);

            if (scan.isValid()) {
```

```
                viewPos.y =
                    viewWindow.convertFromScreenYToViewY(y);
                for (int x=scan.left; x<=scan.right; x++) {
                    viewPos.x =
                        viewWindow.convertFromScreenXToViewX(x);

                    // 计算纹理位置
                    int tx = (int)(a.getDotProduct(viewPos) /
                        c.getDotProduct(viewPos));
                    int ty = (int)(b.getDotProduct(viewPos) /
                        c.getDotProduct(viewPos));

                    // 取得绘图颜色
                    try {
                        int color = texture.getRGB(tx, ty);

                        g.setColor(new Color(color));
                    }
                    catch (ArrayIndexOutOfBoundsException ex) {
                        g.setColor(Color.red);
                    }

                    // 绘制像素
                    g.drawLine(x,y,x,y);
                }
            }
            y++;
        }
    }
}
```

在上述代码中，使用 BufferedImage 对象存储来纹理。在绘制过程中，它对每个像素都生成新的 Color 对象。由于 Graphics2D 类没有 drawPixel()，所以用 drawLine()绘制一个像素长度的直线。注意绘制器并不平铺纹理，而要求纹理和多边形尺寸相同。下节将介绍平铺纹理，这里只是开始，下面生成一个简单的测试。

如清单 8.3 所示的 TextureMapTeat1 类，生成一个简单的多边形，并在屏幕上绘制，使用户可以在其上漫步。

清单 8.3　TextureMapTest1.java

```
import com.brackeen.javagamebook.math3D.*;
```

```
import com.brackeen.javagamebook.graphics3D.*;
import com.brackeen.javagamebook.test.GameCore3D;

public class TextureMapTest1 extends GameCore3D {

    public static void main(String[] args) {
        new TextureMapTest1().run();
    }

    public void createPolygons() {
        Polygon3D poly;

        // 目前为一堵墙
        poly = new Polygon3D(
            new Vector3D(-128, 256, -1000),
            new Vector3D(-128, 0, -1000),
            new Vector3D(128, 0, -1000),
            new Vector3D(128, 256, -1000));
        polygons.add(poly);
    }

    public void createPolygonRenderer() {
        viewWindow = new ViewWindow(0, 0,
            screen.getWidth(), screen.getHeight(),
            (float)Math.toRadians(75));

        Transform3D camera = new Transform3D(0,100,0);
        polygonRenderer = new SimpleTexturedPolygonRenderer(
            camera, viewWindow, "../images/test_pattern.png");
    }
}
```

TextureMapTeat1 从 GameCore3D 类扩展而来，与上一章的 Simple3Dtest2 类大致相同。TextureMapTeat1 只是生成多边形和多边形绘制器，使用户可以在世界上漫步，如图 8-5 所示。

TextureMapTeat1 中的纹理是个简单的非对称的测试纹理，它保证纹理正确贴图到多边形中。很容易看出哪边在上，哪边在下，哪边在左，哪边在右，从而知道是否计算正确。如果纹理中的箭头方向有错或图像翻折，则说明方程有错或丢失符号。

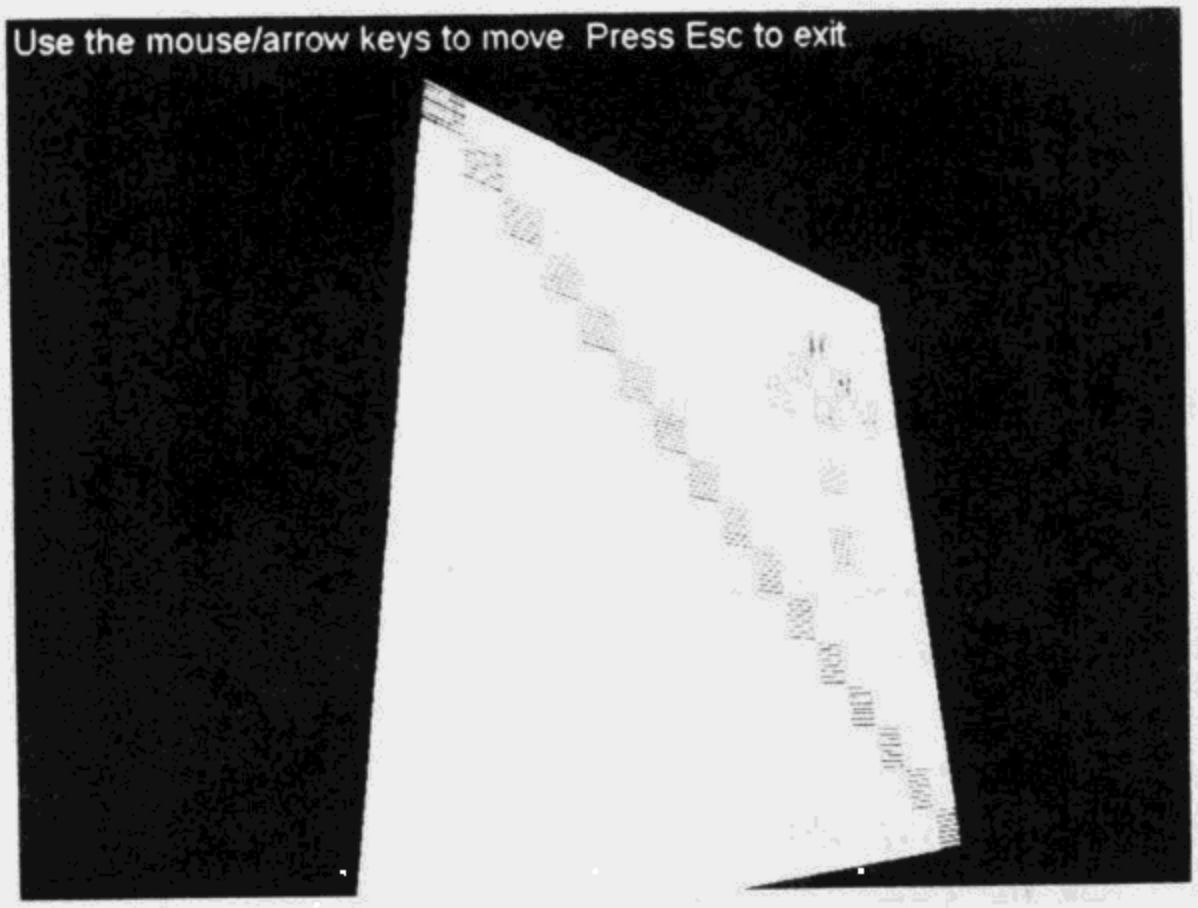


图 8-5 TextureMapTeat1 演示程序在多边形上绘制测试纹理

纹理贴图器的第一个问题

在进行测试时，注意其速度很慢，这根据机器的速度，画一个帧可能超过一秒。造成速度慢有如下的原因：

- 对每个像素生成新的 Color 对象会带来太大的开销，并使内存回收单元工作量太大；
- 调用 setColor()和 drawLine()方法也带来太大的开销；
- BufferedImage 对象的 getRGB()方法要将像素转换成 24 位颜色（如果不是 24 位颜色）；
- 每个像素进行太多的计算；
- 捕获抛出的 ArrayIndexOutOfBoundsException 屏幕比手工检查数组边界更慢，但这里不成问题，因为测试设置成避免这些屏幕了。

在正常任务中以上都不成问题，一般来说，生成 Color 对象和捕获异常并不慢。大多数情况下，这些问题都是正常的，只是在高帧速率的屏幕上对每个像素进行这些处理时才有问题。为了解量的概念，假设 640×480 屏幕的帧速率为 60 帧/秒，则每秒要绘制 18432000 个像素。因此，画单个像素时一定要使用最快的方法，即要避免其他情形中可以接受的一些编程方法。

8.3 优化纹理贴图

不过 Graphics2D 接口并没有提供用不同颜色一次画一个像素的快速接口，Java 中目前也没有直接访问显示内存的接口。理论上，可以用直接指向显示内存的 ByteBuffer 来完成，但一次画一个像素仍然存在 JNI 开销。

为了解决问题，首先将像素画到 BufferedImage 对象中，然后将 BufferedImage 画到屏幕上，这就像第 2 章“二维图形与动画”中介绍的双缓存。利用 BufferedImage，可以用数组从图像中取出数据，直接将像素数据复制到其中。将元素复制到数组是能实现的最快的操作之一。

这样做的缺点是用 Graphics.drawImage()将图像复制到显示内存要增加时间，根据我的测试，这大约要占处理器时间的 5%到 10%，但仍然利大于弊。

一个选择是 BufferedImage 使用什么颜色深度。当然，Graphics.drawImage()应和显示器具有相同的颜色深度和像素布局，使图像能迅速地移到屏幕上，而不需要进行任何的颜色转换。所以要确定显示器颜色深度为 8 位、16 位或 24 位。对 8 位颜色深度，必须使用特定的调色板，通常会限制阴影之类的工作。24 位（或 32 位）颜色提供最佳颜色质量，但 16 位颜色速度更快，因为要移动的数据更少。16 位颜色深度比 24 位颜色的质量稍差，但对使用纹理贴图和灯光的三维游戏已经够好了。使用 16 位 BufferedImage 时，可以取得 BufferedImage 的基础数组如下。

```
BufferedImage   doubleBuffer;
short[] doubleBufferData;
...
 //取得缓存数据
DataBuffer dest = doubleBuffer.getRaster().getDataBuffer();
DoubleBuffer = ((DataBufferUShort)dest).getData();
```

下面要介绍纹理的存储格式。

8.3.1　纹理的存储格式

双缓存和显示器具有相同的颜色深度和像素布局，纹理也应当一样。可以和第一个实例一样对纹理使用 BufferedImage 对象，但并不需要 BufferedImage 提供的所有功能。如清单 8.4 所示生成一个 Texture 抽象类，它只让调用者取得纹理中特定位置（x, y）的颜色。

清单 8.4　Texture.java

```
/**
    Texture 类是表示 16 位颜色纹理的抽象类
*/
public abstract class Texture {

    protected int width;
    protected int height;

    /**
        用指定宽度与高度生成新 Texture
    */
    public Texture(int width, int height) {
```

```
        this.width = width;
        this.height = height;
    }

    /**
        取得 Texture 的宽度
    */
    public int getWidth() {
        return width;
    }

    /**
        取得 Texture 的高度
    */
    public int getHeight() {
        return height;
    }

    /**
        取得纹理中特定位置(x,y)的颜色
    */
        public abstract short getColor(int x, int y);
    }
```

Texture 类的 getColor()方法就是实现取得纹理中特定位置（x, y）的颜色。

记住要在必要时在多边形上平铺纹理。这样可以把每个纹理的宽度和高度限制为 2 的指数，如 16，32，128 等。例如，假设纹理的宽度为 32 像素，则 32 在二进制中的表示如下。

00100000

这个数减 1（31）称为掩码。对于 2 的指数，减去 1 等于将 1 变成 0，将后面的 0 变成 1，在二进制中的表示如下。

00011111

这个掩码表示这个数字范围的所有有效位，这里是纹理的 x 坐标。如果和收到的纹理 x 坐标用&进行位与操作，则可以删除不必要的高位，取得 0 到 31 范围内的值。记住这只适用于 2 的指数。如清单 8.5 所示的 PowerOf2Texture 类就采用这种思路。

清单 8.5　PowerOf2Texture.java

```
package com.brackeen.javagamebook.graphics3D.texture;

/**
```

```
    PowerOf2Texture 类是个 Texture，纹理的宽度与高度限制为 2 的指数，如 32，128 等等
*/
public final class PowerOf2Texture extends Texture {

    private short[] buffer;
    private int widthBits;
    private int widthMask;
    private int heightBits;
    private int heightMask;

    /**
        用指定缓冲区生成新 PowerOf2Texture。绘图宽度是 2 的 widthBits 次幂，或 (1 <<
        widthBits)。同样，绘图高度是 2 的 heightBits 次幂，或 (1 << heightBits)
    */
    public PowerOf2Texture(short[] buffer,
        int widthBits, int heightBits)
    {
        super(1 << widthBits, 1 << heightBits);
        this.buffer = buffer;
        this.widthBits = widthBits;
        this.heightBits = heightBits;
        this.widthMask = getWidth() - 1;
        this.heightMask = getHeight() - 1;
    }

    /**
        取得纹理中特定位置(x,y)的 16 位颜色
    */
    public short getColor(int x, int y) {
        return buffer[
            (x & widthMask) +
            ((y & heightMask) << widthBits)];
    }
}
```

在上述代码中，PowerOf2Texture 类跟踪的短数组保存纹理及其宽度和高度的掩码。getColor()方法用掩码取得正确的 x 和 y 的值，以便在多边形上平捕纹理。

最后，要方便地从 BufferedImage 对象中取得数据以生成纹理。所以要在 Texture 类中增加静态方法 createTexture()。目前，它只生成 PowerOf2Texture 对象，在后面要将其扩展成生成其他类型的纹理。

```
/**
    用指定图像生成 Texture
*/
public static Texture createTexture(BufferedImage image){
      int type = image.getType();
      int width = image.getWidth();
      int height = image.getHeight;

      if (! isPowerOfTwo(width)) || ! isPowerOfTwo(height)){
          throw new IllegalArgumentException(
             "Size of texture must be a power of two.");
      }
      //将图像变成 16 位图像
      if (type != BufferedImage.TYPE_USHORT_565_RGB){
          BufferedImage newImage = new BufferedImage(
              image.getWidth(),image.  getHeight
              BufferedImage.TYPE_USHORT_565_RGB);
          Graphics2D g = newImage.createGraphics();
          g.drawImage(image, 0, 0, null);
          g.dispose();
          image = newImage;
      }
      DataBuffer dest = image.getRaster().getDataBuffer()
      Return new PowerOf2Texture(
            ((DataBufferUShort)dest).getData(),
             countbits(width-1), countbits(hieght-1));
  }
```

createTexture()假设 16 位显示和 BufferedImage 类型 TUPE_USHORT_565_RGB 具有相同的像素格式。虽然还没有见过没有这种像素格式的 16 位显示器，但理论上是有可能的。也可以用 GraphicsConfiguration 的 createCompatibleImage()方法生成和显示器兼容的图像。

然后生成新的 PolygonRenderer 类 FastTexturedPolygonRenderer。这个新类和 SimpleTexturedPolygonRenderer 具有相同的设置程序，计算 A，B，C 向量，并准备 doubleBufferData 数组，这是接收像素的数组，每帧结束时，将双缓存急快移动到显示器中。

这个类还有一个内类 ScanRenderer。它也是一个抽象类，包含对多边形画水平扫描线的方法。

```
/**
```

```
    ScanRenderer 是 FastTexturedPolygonRenderer 的抽象内类，包含对多边形画水平扫描线的方法
*/
  public abstract class ScanRenderer{

     protected Texture currentTexture;

     public void setTexture (Texture texture){
         this. CurrentTexture = texture;
     }
    public abstract void render (int offset,
        int left, int right);
 }
```

不同类型绘图方法都要生成几个不同的 ScanRenderer 对象。

8.3.2 原始优化

最简单的 ScanRenderer 对象只进行多边形填充程序，速度也最快（但什么也不干）。

```
  public void render (int offset,int left, int right){
      for (int x = left;x <=right; x++){
           doubleBufferData[offset++] = (short)0x0007;
      }
    }
```

有了一般概念之后，下面就要生成 ScanRenderer 程序，它完成和 SimpleTextured PolygonRenderer 相同的计算。

```
  public void render (int offset,int left, int right){
      for (int x = left;x <=right; x++){
          int tx = (int) (a.getDotProduct(viewPos)/
               c.getDotProduct (viewPos));
          int ty = (int) (b.getDotProduct(viewPos)/
               c.getDotProduct (viewPos));
          doubleBufferData [offset++]=
              currentTexture.getColor (tx,ty)
          viewPos.x++
      }
  }
```

这时得到的纹理贴图器速度就快多了，以更快速度从纹理将像素数据复制到双缓存中。在 2 台机器上测试这个扫描绘制器，将镜头移到离纹理多边形足够近，使其填充整个 640×480 屏幕。在 2.4Ghz Pentium 4 计算机上，速度可增加 46 倍。而 867MHz G4 计算机，速度增加了 144 倍，真不错，这是个很大的改进，但还可以进一步改进。只要一些乘法和 2 个除法好像没什么，但屏幕上每个像素都要进行这些运算。

下面采用一个最简单的优化，就是把成本较高的代码移到循环的外面。这里，不需要对每个像素都计算 4 个点积，因为 a，b，c 向量是不变的。另外，知道每个像素将 viewPos.x 加 1，因此可以预测点积的改变。ScanRenderer 的下一轮替代就要进行这些优化。

```
public void render (int offset,int left, int right){
    float u = a.getDataProduct(viewPos);
    float v = b.getDataProduct(viewPos);
    float z = c.getDataProduct(viewPos);
    float du = a.x;
    float dv = b.x;
    float dz = c.x;
    for(int x = left;x <=right; x++){
        doubleBufferData [offset++]=currentTexture.getColor(
            (int) (u/z), (int) (v/z);
        u+ = du;
        v+ = dv;
        z+ = dz;
    }
}
```

与上一个扫描绘制器相比，这个优化在 Pentium 4 计算机上速度增加了 1.6 倍。在 G4 计算机上速度增加 1.4 倍。

然后要利用上一章在扫描转换中使用的方法：用整数而不用浮点数。因为每个像素从浮点数换算成整数的成本很高，所以全部用整数。

...

```
public void render (int offset,int left, int right){
    int u = int SCALE * a.getDotProduct(viewPos));
    int v = int SCALE * b.getDotProduct(viewPos))
    int z = int SCALE * c.getDotProduct(viewPos))
    int du = (int) (SCALE * a.x);
    int dv = (int) (SCALE * b.x);
    int dz = (int) (SCALE * c.x);
    for(int x = left;x <=right; x++){
        doubleBufferData [offset++]=
            currentTexture.getColor(u/z,v/z)
        u+ = du;
        v+ = dv;
        z+ = dz;
    }
}
```

Pentium 4 能很好地处理浮点数运行，因此没有加速。但在 G4 中，则比上一个扫描绘制器加速 1.2 倍。这时的速度已经不错了，只不过使用几个数组的引用、一个整数加法和 2 个整数除法。

可以看到，这里的最大瓶颈是 2 个整数除法，加法很方便，但除法会用更多的时钟周期。2 个整数除法是必须的，否则就必然会使纹理贴图的质量下降。质量下降是个关键问题，好在只要视觉上可以接受，可以使质量稍有下降，同时大大增加性能。

一个办法是只对每几个像素计算正确的纹理坐标，然后在这些正确值之间进行插值。数学上，2 个值之间的插值就是估计 2 个已知值之间的值。下一个 ScanRenderer 每隔 16 个像素（如果扫描中留下的像素更少，则少于 16 个像素）计算正确的纹理坐标。这样，这 16 个像素之间只要使用几个加法和移位运算。

```
public void render (int offset, int left, int right){
    float u = a.getDataProduct(viewPos);
    float v = b.getDataProduct(viewPos);
    float z = c.getDataProduct(viewPos);
    float du = INTERP_SIZE * SCALE * a.x;
    float dv = INTERP_SIZE * SCALE * b.x;
    float dz = INTERP_SIZE * c.x;
    int nextTx = (int)( u/z);
    int nextTy = (int)( v/z);
    int x = left;
    while (x <=right){
    int tx = nextTx;
    int tx = nextTy;
    int maxLength = right-x+1;
    if (maxLength > INTERP_SIZE){
        u+=du;
        u+=dv;
        u+=dz;
        nextTx = (int)( u/z);
        nextTy = (int)( v/z);
        int dtx = (nextTx-tx) >> INTERP_SIZE_BITS;
        int dty = (nextTy-ty) >> INTERP_SIZE_BITS;
        int endOffset = offset + INTERP_SIZE
        while (offset < endOffset){
            doubleBufferData [offset++]=
                texture.getColor(
                tx >>SCALE_BITS, ty >> SCALE_BITS);
            tx+=dtx;
            ty+=dty;
```

```
        }
        x+ = INTERP_SIZE
    }
    else{
    //可变插值长度
    int interpSize = maxLength;
    u += interpSize * SCALE * a.x;
    v += interpSize * SCALE * b.x;
    z += interpSize * c.x;
    nextTx = (int)( u/z);
    nextTy = (int)( v/z);
    int dtx = (nextTx-tx) >> interpSize;
    int dty = (nextTy-ty) >> interpSize;
    int endOffset = offset + interpSize;
    while (offset < endOffset){
          doubleBufferData [offset++]=
               texture.getColor(
               tx >>SCALE_BITS, ty >> SCALE_BITS);
             tx+=dtx;
             ty+=dty;
             }
             x+ = interpSize;
          }
      }
      }
```

在 2 台计算机上测试时，Pentium 4 机速度增加 1.6 倍。G4 机速度增加 1.4 倍。

前面曾介绍过，这个 ScanRenderer 会使图像质量稍有下降，但大多数情况下并不明显。如果多边形深度沿水平为措线变化很大，则质量下降比较明显。因此，直视多边形时，多边形深度沿水平措线变化不大，所以效果很好。地面的效果也很好，因为其深度从左到右都不会改变。但是，从一边看垂直多边形时，多边形深度沿水平为措线变化很大，因此纹理会有些扭曲和变形，这在低分辨率屏幕上特别明显，其中 16 个像素要占用很大的空间。

一种限制扭曲的方法是，根据多边形深度改变情况使用可变插值的长度。深度改变较小或不变时，可以用较大的插值长度。而深度改变较大时，则要用较小的插值长度。读者可以自己进行这种改进。

8.3.3 内联方法

还可以进行另一种改进，即采用内联方法。调用方法有一定开销，而这里要对每个像素调用 texture.getColor()方法。好在 HotSpot VM 虚拟机（Java 1.4 运行环境提供的缺省虚拟机）是智能的，所以可以使用一些内联方法。内联方法就是把方法代码移到调用代

码中，从而消除方法的调用。HotSpot VM 内联一些非虚拟的短方法（记住，非虚拟机方法不用子类覆盖）。对于 Texture，只有 PowerOf2Texture 类实现 getColor()方法，这个方法很短，因此 HotSpot VM 采用内联方法。

本章稍后还有其他 Texture 子类，如 ShadedTexture。如果另一个类实现 getColor()方法，则 HotSpot VM 不采用内联方法，每个像素无法避免调用开销。

为了让 HotSpot VM 采用内联方法，声明一个最终局部变量，HotSpot VM 知道最终局部变量的类，因此可以用内联方法。

```
final Texture texture = currentTexture;
```

由于这是最终局部变量，所以不会改变，HotSpot VM 应该能将 getColor()方法内联。其实这是行不通的，虽然我不知道 HotSpot VM 为什么不能将 getColor()方法内联，但事实上就是行不通，因此只好另寻方法。

一个可行的方法是对每种纹理生成不同的 ScanRenderer 对象。每个绘制器的代码大致相同，只是生成局部变量 texture，显式引用这个纹理。

```
public class PowerOf2TextureRenderer extends ScanRenderer{

        public void render (int offset,int left, int right){
            PowerOf2Texture texture = (PowerOf2Texture)currentTexture;

            ...
              //这里放纹理贴图代码
        }
    }
```

对于 Powerof2TextureRenderer, HotSpot VM 知道纹理为 PowerOf2Texture，因此可以相应将 getColor()方法内联。这个方案是可行的，但不是真正的面向对象代码，改变一个 ScanRenderer 时，要到处改变。另一个缺点是，Texture 类（和任何具有 ScanRendere 的其他类）必须是最终类，它不能派生其他子类，否则 HotSpot VM 将无法内联。

有时必须委屈求全，这里就是一个例子。汇编语言、C 语言和 C++的编程人员有时要进行一些古怪的处理，目的是针对硬件或操作系统进行优化，Java 编程人员有时也要进行一些古怪的处理，目的是针对虚拟机进行优化。

在 2 台机算机上测试内联优化的结果时，Pentium 4 机的速度增加 2.3 倍。G4 机的速度增加 1.3 倍。总的来说，与原先的低速纹理贴图器相比，这个绘制器在 Pentium 4 机上速度增加了 284 倍。G4 机上速度增加 436 倍，快多了。

8.3.4 快速纹理贴图演示程序

有了 Texture 类和 FastTexturedPolygonRenderer 之后，下面就要生成另一演示程序 TextureMapTest2，如图 8-6 所示，它生成的凸多面体各面都采用纹理贴图。

TextureMapTest2 使用 TexturePolygon3D 类，与普通 Polygon3D 类相似，但包含多边形纹理和纹理边界的字段。

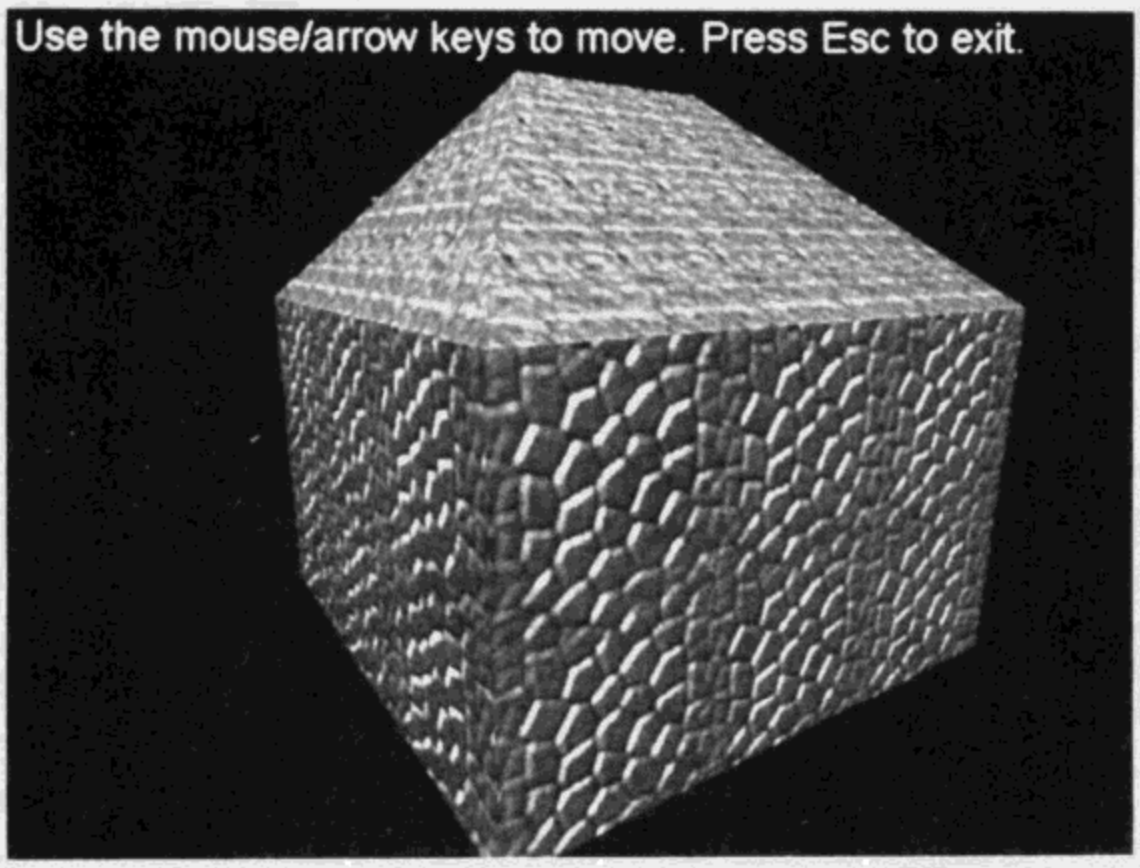


图 8-6 TextureMapTest2 绘制纹理贴图多面体的帧速率比 TextureMapTest1 高得多

TextureMapTest2 的源代码生成一大堆多边形，其中一个重要方法是 setTexture（）。

```
Public void setTexture(TexturedPolygon3D poly, Texture texture){
    Vector3D origin = poly.getVertex(0)

    Vector3D dv = new Vector3D(poly .getVertex(1));
    dv.subtract(origin)

    Vector3D du = new Vector3D();
    Du.setToCrossProduct(poly.getNormal(),dv);

    Rectangle3D textureBounds = new Rectangle3D (origin ,du, dv,
        texture.getWidth(),texture.getHeight());

    poly.setTexture(texture, textureBounds);
}
```

这个方法造成多边形的纹理边界，利用方法就不必对每个多边形显式生成纹理边界。

TextureMapTest2 的速度要快得多，按 R 键可以看到系统的帧速率。下面就要在这个演示程序中增加一些阴影功能。

8.4 简单灯光

本节要在多边形中增加一些逼真的阴影。上章最后一个演示程序是个实体颜色的三维房子。房子侧面的阴影比正面深，这是伪装的多边形阴影。本节要用灯光源建立真正的阴

影，而不是手工给多边形上色。

8.4.1　扩散反射

所用的灯光符合兰伯特定律描述的扩散反射。兰伯特定律指出，反射光的强度取决于表面法线和光源方向之间的夹角，如图 8-7 所示。

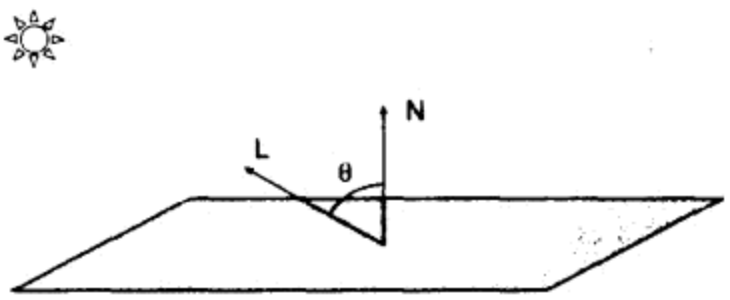


图 8-7　扩散反射光的强度取决于表面法线与光源方向之间的夹角

角度越小，反射光越强。角度越大，反射光越弱。假设将表面法线和光源方向范化，则可以用下列数学方程表示，其中 i 是反射光的强度。

$i = N \cdot L$

实际生活中扩散反射的实例就是房间里的光源。例如，如果暗室中的墙和天花板颜色相同，光若从接近天花板的光源射出，则从墙和天花板相交的角上看，接近角上的天花板比墙上暗。这是因为，墙体直接从光源接收光线，而天花板则是通过一个大角度接收光线，这样，就使天花板比墙上暗。

8.4.2　环境光

白天的室内可能不需要开灯，因为阳光从窗口射入，并在房间里反射，会使室内一片明亮。房间里来回反射的光很难进行模型计算，所以用环境光近似这个效果。环境光限制一个物体的最小光强度。例如，在日照房间里，每个物体的光强度至少为 0.7。为此可以对多边形指定一个环境光强度。

$i = a + N \cdot L$

环境光强度介于 0 和 1 之间。

8.4.3　光源强度

除了光源强度外，光源本身也可以指定强度值，就像 60 瓦的灯泡和 100 瓦灯泡亮度不同。可以用 L 描述光源强度，使方程变成如下。

$I = a + l(N \cdot L)$

8.4.4　距离减弱

光源强度还可以随距离减弱，即物体离光源越远，光的强度越弱。减弱距离是光源看不到的距离。用 l 作为光源强度，分作为减弱距离，d 作为物体与光源之间的距离，则方程如下。

$l' = l(f-d)/(f+d)$

i = a+1' (N • L)

8.4.5 实现点光源

如清单 8.6 所示的 PointLight3D 类描述光源强度和减弱距离。

清单 8.6 PointLight3D.java

```
package com.brackeen.javagamebook.math3D;

/**
    PointLight3D 类描述光源强度(在 0 到 1 之间)，使光的强度越远越弱
    and optionally a distance falloff value,
    which causes the light to diminish with distance.
*/
public class PointLight3D extends Vector3D {

    public static final float NO_DISTANCE_FALLOFF = -1;

    private float intensity;
    private float distanceFalloff;

    /**
        取得指定距离的光强度
    */
    public float getIntensity(float distance) {
        if (distanceFalloff == NO_DISTANCE_FALLOFF) {
            return intensity;
        }
        else if (distance >= distanceFalloff) {
            return 0;
        }
        else {
            return intensity * (distanceFalloff - distance)
                / (distanceFalloff + distance);
        }
    }
}
```

PointLight3D 类是个 Vector3D，它包含光的光源强度和减弱距离值，其 getIntensity() 方法取得指定距离的光强度。

8.5　实现纹理照明

这里要解决的问题是在纹理上实际实现照明。简单地说，多边形光源强度值为 0 到 1，要对纹理采用这个光源强度值。当取值为 1 时，纹理不变，小于 1 时，修改纹理，增加暗色，直到值为 0 时，使纹理变成几乎完全黑色。

这里不是简单地把光源值乘以 16 位颜色值。因为 16 位颜色值是包装数据结构的。实际上，取出红色、绿色、蓝色值，然后乘以光源强度值，然后换算成 16 位颜色值。每个像素都有这样的处理工作。

也可以将光源强度限制为几个值（如 64 个），只装入每个纹理从明到暗的 64 个不同版本，然后就可以根据光源强度值选择每个多边形的正确纹理。

这样可以加快速度，但这样会浪费内存。128×128 位纹理要占用 30K 内存，如果有 64 个版本，则要 2Mbyte 的内存，这样会严重限制内存中的纹理个数。

另一种方法是不用 64 个不同版本，而只有一个纹理版本，但调色板有 64 个不同版本。为此，首先要求所有纹理为 8 位颜色，这样每个纹理有自己的 256 色调色板。然后不是生成纹理的 64 个不同版本，而只要生成调色板的 64 个不同版本。如图 8-8 所示，用简化版本显示了这个基本思路。

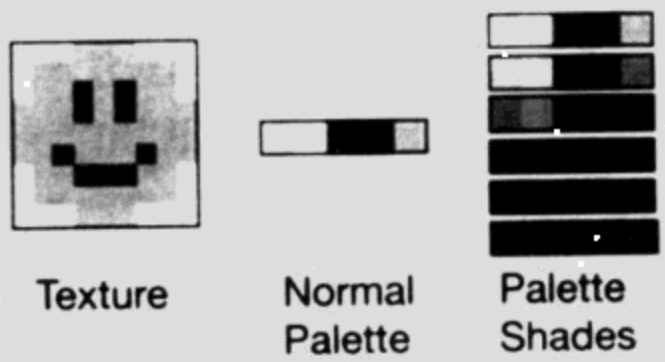


图 8-8　为了照明纹理，保存纹理的正常全照明拷贝和不同版本的调色板，从正常调色板到几乎全黑的调色板

如果调色板本身表示 16 位颜色，则调色板的 64 个不同版本只产生 32K 的开销。对于 8 位的 128×128 纹理，总共只占 80K 的内存，这比 2Mbyte 要小多了。

如清单 8.7 所示的 ShadeTexture 类就实现这个思想。

清单 8.7　ShadeTexture.java

```
package com.brackeen.javagamebook.graphics3D.texture;

import java.awt.Color;
import java.awt.image.IndexColorModel;

/**
    ShadedTexture 类是个 Texture，有多个色档。纹理源图像存放成 8 位图像，每个色档有一个调
    色板
```

```
*/
public final class ShadedTexture extends Texture {
    public static final int NUM_SHADE_LEVELS = 64;
    public static final int MAX_LEVEL = NUM_SHADE_LEVELS-1;

    private static final int PALETTE_SIZE_BITS = 8;
    private static final int PALETTE_SIZE = 1 << PALETTE_SIZE_BITS;

    private byte[] buffer;
    private IndexColorModel palette;
    private short[] shadeTable;
    private int defaultShadeLevel;
    private int widthBits;
    private int widthMask;
    private int heightBits;
    private int heightMask;

    // the row set in setCurrRow and used in getColorCurrRow
    private int currRow;

    /**
        从指定的 8 位图像缓冲区和调色板生成新 ShadedTexture。绘图宽度是 2 的 widthBit 次幂，
        （或 1 << widthBits）。同样，绘图高度是 2 的 heightBits 次幂，（或 1 << heightBits）。
        纹理色档从正常到几乎全黑
    */
    public ShadedTexture(byte[] buffer,
        int widthBits, int heightBits,
        IndexColorModel palette)
    {
        this(buffer, widthBits, heightBits, palette, Color.BLACK);
    }

    /**
        从指定的 8 位图像缓冲区、调色板和图标色档和调色板生成新 ShadedTexture。绘图宽度
        是 2 的 widthBit 次幂，（或 1 << widthBits）。同样，绘图高度是 2 的 heightBits 次幂，（或
        1 << heightBits）。纹理色档从正常到几乎全黑
    */
    public ShadedTexture(byte[] buffer,
        int widthBits, int heightBits,
```

```
        IndexColorModel palette, Color targetShade)
    {
        super(1 << widthBits, 1 << heightBits);
        this.buffer = buffer;
        this.widthBits = widthBits;
        this.heightBits = heightBits;
        this.widthMask = getWidth() - 1;
        this.heightMask = getHeight() - 1;
        this.buffer = buffer;
        this.palette = palette;
        defaultShadeLevel = MAX_LEVEL;

        makeShadeTable(targetShade);
    }

    /**
        生成这个 ShadedTexture 的色档表，调色板中每个项目是从原有颜色到指定目标颜色的色
        档
    */
    public void makeShadeTable(Color targetShade) {

        shadeTable = new short[NUM_SHADE_LEVELS*PALETTE_SIZE];

        for (int level=0; level<NUM_SHADE_LEVELS; level++) {
            for (int i=0; i<palette.getMapSize(); i++) {
                int red = calcColor(palette.getRed(i),
                    targetShade.getRed(), level);
                int green = calcColor(palette.getGreen(i),
                    targetShade.getGreen(), level);
                int blue = calcColor(palette.getBlue(i),
                    targetShade.getBlue(), level);

                int index = level * PALETTE_SIZE + i;
                // RGB 5:6:5
                shadeTable[index] = (short)(
                            ((red >> 3) << 11) |
                            ((green >> 2) << 5) |
                            (blue >> 3));
            }
```

```
        }

    }

    private int calcColor(int palColor, int target, int level) {
        return (palColor - target) * (level+1) /
            NUM_SHADE_LEVELS + target;
    }

    /**
        设置调用 getColor()时使用的缺省色档
        is called.
    */
    public void setDefaultShadeLevel(int level) {
        defaultShadeLevel = level;
    }

    /**
        取得调用 getColor()时使用的缺省色档
    */
    public int getDefaultShadeLevel() {
        return defaultShadeLevel;
    }

    /**
        用缺省色档取得这个 Texture 在指定位置(x,y)处的 16 位颜色
    */
    public short getColor(int x, int y) {
        return getColor(x, y, defaultShadeLevel);
    }

    /**
        用指定色档取得这个 Texture 在指定位置(x,y)处的 16 位颜色
    */
    public short getColor(int x, int y, int shadeLevel) {
        return shadeTable[(shadeLevel << PALETTE_SIZE_BITS) |
            (0xff & buffer[
            (x & widthMask) |
            ((y & heightMask) << widthBits)])];
```

```
        }
    }
```

在上述代码中，纹理本身存放在缓冲区字节数组中。字节数组本身没用，因为它不包含任何颜色数据，只包含 256 色调色板的索引。256 色调色板的 64 个版本存放在 shadeTable 数组中。色档表在 makeShadeTable()方法中生成。在 getColor()方法中，用 shadeTable 将 8 位缓冲区中的索引转换成调色板中的 16 位颜色。

另外，生成色档表时所用的缺省色档值为黑色，因此所有纹理从正常调色板（1）淡化到黑色（0）。也可以用其他颜色生成不同效果。另一种方法是生成点光源效果，从正常调色板（0）淡化到白色（1）。

下面就要试验新的阴影纹理。如清单 8.8 所示的 ShadedTexturedPolygonRenderer 类绘制 ShadedTexture，动态计算每个多边形的灯光强度。

清单 8.8 ShadedTexturedPolygonRenderer.java

```
package com.brackeen.javagamebook.graphics3D;

import java.awt.*;
import java.awt.image.*;
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.graphics3D.texture.*;

/**
    ShadedTexturedPolygonRenderer 类是个 PolygonRenderer，用一个光源动态绘制 ShadedTextured。缺
    省情况下，环境光强度为 0.5，没有点光源
*/
public class ShadedTexturedPolygonRenderer
    extends FastTexturedPolygonRenderer
{

    private PointLight3D lightSource;
    private float ambientLightIntensity = 0.5f;
    private Vector3D directionToLight = new Vector3D();

    public ShadedTexturedPolygonRenderer(Transform3D camera,
        ViewWindow viewWindow)
    {
        this(camera, viewWindow, true);
    }

    public ShadedTexturedPolygonRenderer(Transform3D camera,
```

```
        ViewWindow viewWindow, boolean clearViewEveryFrame)
    {
        super(camera, viewWindow, clearViewEveryFrame);
    }

    /**
        取得这个绘制器的光源
    */
    public PointLight3D getLightSource() {
        return lightSource;
    }

    /**
        设置这个绘制器的光源
    */
    public void setLightSource(PointLight3D lightSource) {
        this.lightSource = lightSource;
    }

    /**
        取得环境光强度
    */
    public float getAmbientLightIntensity() {
        return ambientLightIntensity;
    }

    /**
        设置环境光强度，通常为0到1之间
    */
    public void setAmbientLightIntensity(float i) {
        ambientLightIntensity = i;
    }

    protected void drawCurrentPolygon(Graphics2D g) {
        // 绘图之前设置多边形色档
        if (sourcePolygon instanceof TexturedPolygon3D) {
            TexturedPolygon3D poly =
                ((TexturedPolygon3D)sourcePolygon);
```

```
        Texture texture = poly.getTexture();
        if (texture instanceof ShadedTexture) {
            calcShadeLevel();
        }
    }
    super.drawCurrentPolygon(g);
}

/**
    计算当前多边形色档
*/
private void calcShadeLevel() {
    TexturedPolygon3D poly = (TexturedPolygon3D)sourcePolygon;
    float intensity = 0;
    if (lightSource != null) {

        // 将多边形所有顶点平均
        directionToLight.setTo(0,0,0);
        for (int i=0; i<poly.getNumVertices(); i++) {
            directionToLight.add(poly.getVertex(i));
        }
        directionToLight.divide(poly.getNumVertices());

        //建立从平均顶点到灯光的向量
        directionToLight.subtract(lightSource);
        directionToLight.multiply(-1);

        // 取得从灯光到减弱位置的距离。
        float distance = directionToLight.length();

        // 计算扩散效果
        directionToLight.normalize();
        Vector3D normal = poly.getNormal();
        intensity = lightSource.getIntensity(distance)
            * directionToLight.getDotProduct(normal);
        intensity = Math.min(intensity, 1);
        intensity = Math.max(intensity, 0);
    }
```

```
            intensity+=ambientLightIntensity;
            intensity = Math.min(intensity, 1);
            intensity = Math.max(intensity, 0);
            int level =
                Math.round(intensity*ShadedTexture.MAX_LEVEL);
            ((ShadedTexture)poly.getTexture()).
                setDefaultShadeLevel(level);
        }
    }
```

在上述代码中，在 calcShadeLevel()方法中计算每个多边形的色档。这个类只是一个演示，实际上只有光源改变或多边形移动时才会重算多边形的色档。另外，ShadedTexturedPolygonRenderer 只用一个点光源，所以每个多边形的环境光强度是相同的。

尽管类很简单，但可以动态改变光源的灯光强度，这个特性非常好。下面生成 ShadingTest1 类以使用这个新绘制器，如图 8-9 所示 ShadingTest1 的屏幕图形。

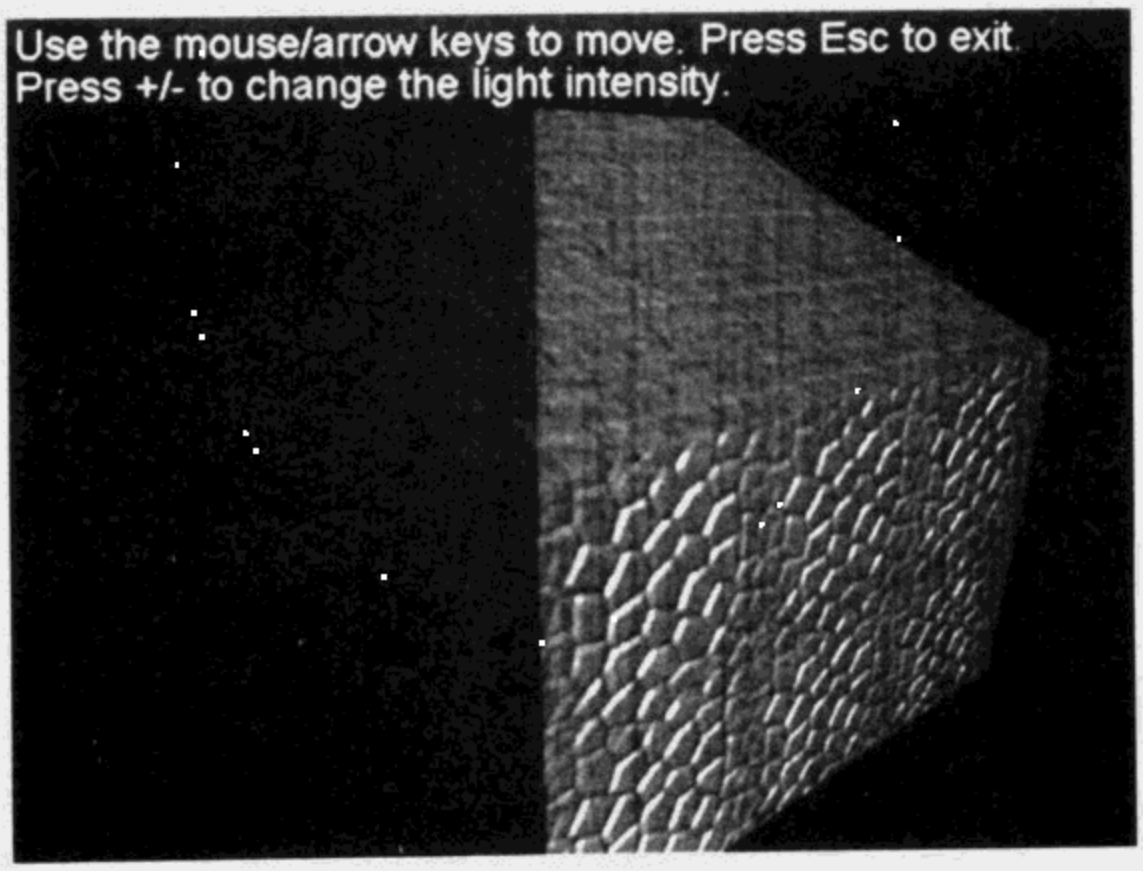


图 8-9　ShadingTest1 类使每个多边形有不同色档，可以动态改变光源的灯光强度

ShadingTest1 和 TextureMapTest2 相似，但用 ShadedTexturedPolygonRenderer 并可以通过+/-键动态改变光强度。

8.6　使用阴影图的高级照明

前面的演示程序很精彩，但整个多边形都使用相同的色档是不够精确的。因为多边形不同部分和光源的角度、距离可能不同，如图 8-10 所示。

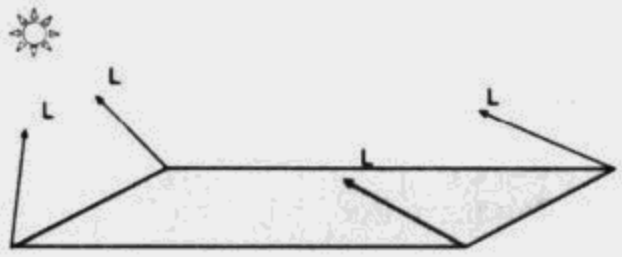


图 8-10　整个多边形使用相同色档不够精确。多边形不同部分与光源的角度和距离可能不同

若要产生更逼真的灯光效果，则可以将所有的多边形分解成 16×16 的多边形，也可以对多边形的每个图素都计算不同的灯光强度值。可以想像，后者的成本将会太高，特别是当需要大量的光源时。所以可以在游戏开始之前计算光强度，然后将这些值存放在多边形的色档图中。

为了减少色档图的内存要求，可以对像素采用隔断来计算光强度，例如分成 16×16 的间隔。这样就可以使色档图比多边形总体尺寸较小。

8.6.1　求出边界矩形

色档图是一个矩形图，它包含多边形的整个表面。可以用 2 种方法求出矩形图。一种方法是求出多边形的最小边界矩形，如图 8-11 所示。最小边界矩形是盖住一个多边形的最小矩形，要保证矩形和多边形至少有一边达到共边。

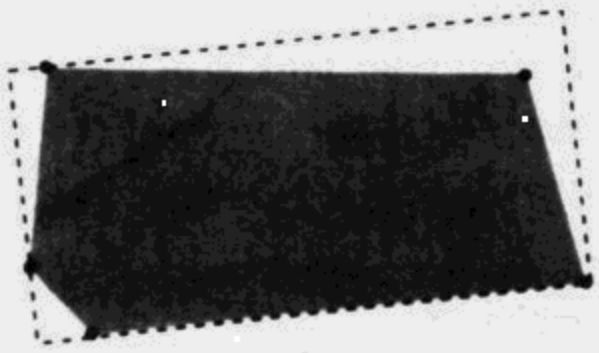


图 8-11　边界矩形和多边形至少有一边达到共边

此外，还要让矩形和多边形纹理边界具有相同的方向，才能使色档图和纹理图正常对齐。虽然使用这种方案，但它有一个缺点：可能浪费一些内存。例如，如图 8-12 所示，矩形边界和纹理边界对齐，很明显，矩形边界要并比多边形区大得多。

在 TexturedPolygen3D 对象中增加一个 calcBoundingRectangle()方法，它计算与纹理边界对齐的多边形边界。方法用真积求出盖住多边形每个顶点的矩形边长。

在 TexturedPolygen3D 对象中增加一个 calcBoundingRectangle()方法，它计算与纹理边界对齐的多边形边界。方法用真积求出盖住多边形每个顶点的矩形边长。

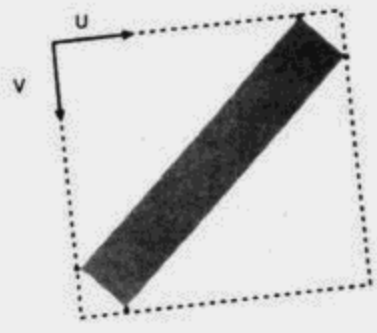


图 8-12　矩形边界和纹理边界对齐可能浪费一些内存

```
/**
    计算与纹理边界对齐的多边形边界
*/
public Rectangle3D calcBoundingRectangle(){

    Vector3D u = new Vector3D(textureBounds.getDirectionU())
    Vector3D v = new Vector3D(textureBounds.getDirectionV())
    Vector3D d = new Vector3D()
    u.normalize()
    v.normalize()

    float uMin = 0
    float uMix = 0
    float vMin = 0
    float vMix = 0
    for (int i = 0;I<getNumVertices();i++){
        d.setTo(getVertex(i));
        d.subtract(getVertex(0));
   float uLength = d.getDotProduct(u);
   float vLength = d.getDotProduct(v)
   uMin = Math.min(uLength,uMin);
   uMix = Math.max(uLength,uMix);
   vMin = Math.min(vLength,vMin);
   vMax = Math.max(vLength,vMix);
}
Rectangle3D boundingRect = new Rectangle3D
Vector3D origin = boundingRect.getOrigin();
Origin.setTo(getVertex(0));
d.setTo(u);
d.multiply(uMin);
origin.add(d);
d.setTo(v);
d.multiply(vMin);
origin.add(d);
boundingRect.getDirectionU().setTo(u);
boundingRect.getDirectionV().setTo(v);
boundingRect.setWidth(uMax-uMin);
boundingRect.setHeight(vMax-vMin);
//显式设置法线，因为纹理方向生成的法线可能与多边形法线反向。
```

```
    boundingRect.setNormal(getNormal());

    return boundingRect
  }
```

8.6.2　采用色档图

下一步要确定何时对多边形采用色档图，可以组合色档图和多边形纹理，生成阴影表面，如图 8-13 所示。

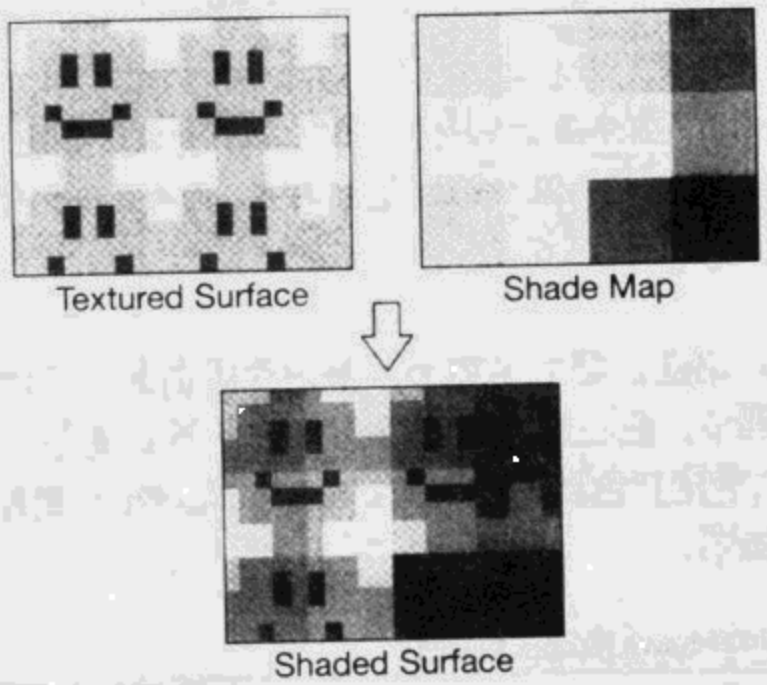


图 8-13　色档图盖住多边形每个顶点，组合色档图和多边形纹理，生成阴影表面

从图中看出，方法生成的阴影表面在阴影边界上有点块状。若要解决问题，可以在色档值之间进行插值操作，这样可以得到如图 8-14 所示的结果。

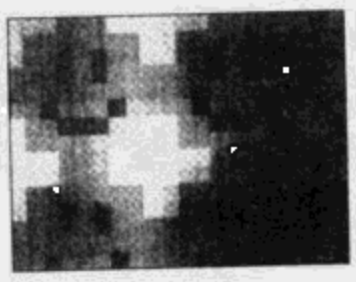


图 8-14　在色档值之间进行插值，得到更光顺的表面

另一种调节阴影的方法是在纹理贴图的同时进行操作。虽然方法可行，但不能通过色档图插值进行优化图像，因为对屏幕上每个像素进行色档值插值计算是非常慢的。

另一方面，建立表面之后，使用表面的速度会受纹理贴图速度的限制。因为表面可能使用大量的内存，我们稍后会解决内存的问题。下面要编写一些代码，来生成色档图和阴影表面。

首先，为了进一步定义几何多边形边界、色档图和表面，请看如图 8-15 所示。图中表面边界要比多边形边界稍大一些，以弥补纹理贴图期间造成的浮点数误差。浮点数误差可能使得到的（x，y）表面坐标不正确，相关一、两个图像。由于不希望纹理贴图计算表面边界以外的值，所以可以让表面边界比多边形边界稍大一些即可克服问题。另外，注意色

档图和多边形边界对齐，而不是和表面边界对齐。

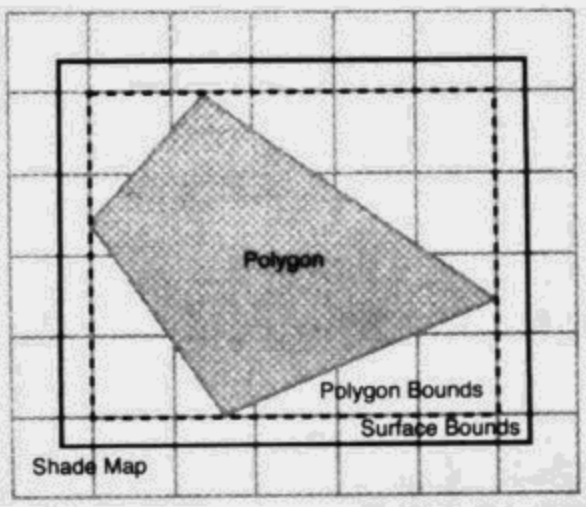


图 8-15 表面边界比多边形边界稍大一些，以弥补纹理贴图期间造成的浮点数误差。色档图和多边形边界对齐，而不是和表面边界对齐

8.6.3 建立色档图

下面要建立色档图。记住，每个多边形只要建立色档图一次即可，除非光线进行改变或多边形移动。在绘制器中，色档图要保持静态，因此不必重新计算。如清单 8.9 所示的 ShadedSurface 类，它就是建立色档图的代码。ShadedSurface 类是 Texture 的子类，包含源纹理、表面数据和色档图。

清单 8.9 用 ShadedSurface.java 类建立色档图

```
public static final int SURFACE_BORDER_SIZE = 1;

public static final int SHADE_RES_BITS = 4;
public static final int SHADE_RES = 1 << SHADE_RES_BITS;
public static final int SHADE_RES_MASK = SHADE_RES - 1;
public static final int SHADE_RES_SQ = SHADE_RES*SHADE_RES;
public static final int SHADE_RES_SQ_BITS = SHADE_RES_BITS*2;

...
/**
    从指定点光源表和环境光强度建立这个表面的色档图
    list of point lights and the ambiant light intensity.
*/
public void buildShadeMap(List pointLights,
    float ambientLightIntensity)
{

    Vector3D surfaceNormal = surfaceBounds.getNormal();
```

```
        int polyWidth = (int)surfaceBounds.getWidth() -
            SURFACE_BORDER_SIZE*2;
        int polyHeight = (int)surfaceBounds.getHeight() -
            SURFACE_BORDER_SIZE*2;
        // 假设 SURFACE_BORDER_SIZE 为 <= SHADE_RES
        shadeMapWidth = polyWidth / SHADE_RES + 4;
        shadeMapHeight = polyHeight / SHADE_RES + 4;
        shadeMap = new byte[shadeMapWidth * shadeMapHeight];

        // 计算色档图原点
        Vector3D origin = new Vector3D(surfaceBounds.getOrigin());
        Vector3D du = new Vector3D(surfaceBounds.getDirectionU());
        Vector3D dv = new Vector3D(surfaceBounds.getDirectionV());
        du.multiply(SHADE_RES - SURFACE_BORDER_SIZE);
        dv.multiply(SHADE_RES - SURFACE_BORDER_SIZE);
        origin.subtract(du);
        origin.subtract(dv);

        // 计算每个样本点的色档
        Vector3D point = new Vector3D();
        du.setTo(surfaceBounds.getDirectionU());
        dv.setTo(surfaceBounds.getDirectionV());
        du.multiply(SHADE_RES);
        dv.multiply(SHADE_RES);
        for (int v=0; v<shadeMapHeight; v++) {
            point.setTo(origin);
            for (int u=0; u<shadeMapWidth; u++) {
                shadeMap[u + v * shadeMapWidth] =
                    calcShade(surfaceNormal, point,
                    pointLights, ambientLightIntensity);
                point.add(du);
            }
            origin.add(dv);
        }
    }

    /**
        确定多边形点的色档。计算平面上点的兰伯特反射。每个点光源具有强度和减弱距离值，
        但不计算其他多边形的光反射和阴影。返回值为 0 到 ShadedTexture.MAX_LEVEL 之间
```

```
    */
    protected byte calcShade(Vector3D normal, Vector3D point,
        List pointLights, float ambientLightIntensity)
    {
        float intensity = 0;
        Vector3D directionToLight = new Vector3D();

        for (int i=0; i<pointLights.size(); i++) {
            PointLight3D light = (PointLight3D)pointLights.get(i);
            directionToLight.setTo(light);
            directionToLight.subtract(point);

            float distance = directionToLight.length();
            directionToLight.normalize();
            float lightIntensity = light.getIntensity(distance)
                * directionToLight.getDotProduct(normal);
            lightIntensity = Math.min(lightIntensity, 1);
            lightIntensity = Math.max(lightIntensity, 0);
            intensity += lightIntensity;
        }

        intensity = Math.min(intensity, 1);
        intensity = Math.max(intensity, 0);

        intensity+=ambientLightIntensity;

        intensity = Math.min(intensity, 1);
        intensity = Math.max(intensity, 0);
        int level = Math.round(intensity*ShadedTexture.MAX_LEVEL);
        return (byte)level;
    }
}
```

上述代码很简单，其中 BuildShadeMap()实现对表面建立色档图，calcShade()方法实现计算色档图中某点光强度的值。

在绘制器中，要在游戏开始之前就建立所有的色档图，但另一个思路是把色档图存放在一个文件中，在游戏进行时，再从文件中装入色档图。这样，就不必在每次启动游戏时重新计算色档图了。

8.6.4　建立表面

下面要建立表面，如清单 8.10 所示，代码涉及 2 个不同的方法：一个实现在色档图中进行插值操作，一个实现在表面上用正确的色档来绘制纹理。

清单 8.10　用 ShadedSurface.java 类建立表面

```
/**
    建立表面。首先，这个方法调用 retrieveSurface()，检查是否需要重建表面。如果不需要重建表
    面，则平铺原纹理和采用色档图，从而建立表面
*/
    public void buildSurface() {

        if (retrieveSurface()) {
            return;
        }

        int width = (int)surfaceBounds.getWidth();
        int height = (int)surfaceBounds.getHeight();

        // 生成新表面（缓冲区）
        newSurface(width, height);

        // 建立表面
        // 假设表面边界与纹理边界对齐（可能原点不同）
        Vector3D origin = sourceTextureBounds.getOrigin();
        Vector3D directionU = sourceTextureBounds.getDirectionU();
        Vector3D directionV = sourceTextureBounds.getDirectionV();

        Vector3D d = new Vector3D(surfaceBounds.getOrigin());
        d.subtract(origin);
        int startU = (int)((d.getDotProduct(directionU) -
            SURFACE_BORDER_SIZE));
        int startV = (int)((d.getDotProduct(directionV) -
            SURFACE_BORDER_SIZE));
        int offset = 0;
        int shadeMapOffsetU = SHADE_RES - SURFACE_BORDER_SIZE -
            startU;
        int shadeMapOffsetV = SHADE_RES - SURFACE_BORDER_SIZE -
```

```
        startV;

    for (int v=startV; v<startV + height; v++) {
        sourceTexture.setCurrRow(v);
        int u = startU;
        int amount = SURFACE_BORDER_SIZE;
        while (u < startU + width) {
            getInterpolatedShade(u + shadeMapOffsetU,
                v + shadeMapOffsetV);

            // 一直绘制，直到需要重算插值阴影（每个 SHADE_RES 像素）。
            int endU = Math.min(startU + width, u + amount);
            while (u < endU) {
                buffer[offset++] =
                    sourceTexture.getColorCurrRow(u,
                        shadeValue >> SHADE_RES_SQ_BITS);
                shadeValue+=shadeValueInc;
                u++;
            }
            amount = SHADE_RES;
        }
    }

    /**
        从色档图中取得指定位置(u,v)的色档。u 和 v 值由 SHADE_RES_BITS 左移位，其他位用
        于数值间插值。例如，色档 1 和 3 之间的中点色档为 2
    */
    public int getInterpolatedShade(int u, int v) {

        int fracU = u & SHADE_RES_MASK;
        int fracV = v & SHADE_RES_MASK;

        int offset = (u >> SHADE_RES_BITS) +
            ((v >> SHADE_RES_BITS) * shadeMapWidth);

        int shade00 = (SHADE_RES-fracV) * shadeMap[offset];
        int shade01 = fracV * shadeMap[offset + shadeMapWidth];
        int shade10 = (SHADE_RES-fracV) * shadeMap[offset + 1];
```

```
        int shade11 = fracV * shadeMap[offset + shadeMapWidth + 1];

        shadeValue = SHADE_RES_SQ/2 +
            (SHADE_RES-fracU) * shade00 +
            (SHADE_RES-fracU) * shade01 +
            fracU * shade10 +
            fracU * shade11;

        // u增加时要增加的值
        shadeValueInc = -shade00 - shade01 + shade10 + shade11;

        return shadeValue >> SHADE_RES_SQ_BITS;
    }

    /**
        从色档图中取得指定位置(u,v)的色档
    */
    public int getShade(int u, int v) {
        return shadeMap[u + v * shadeMapWidth];
    }
```

最好能加快建立表面的速度，因为游戏是要动态建立表面的。因此，buildSurface()方法没有采用优化。

例如，在ShadedTexture类中增加2个方法setCurrRow()和getColorCurrRow()。由于一次一行地建立表面，所以可以使用这些方法而不用getColor()，这样使每个图素不必重新计算行偏移量。

getInterpolatedShade()从色档图中取得指定位置（U，V）的色档，但也不必对每个图素都调用它，因为是水平建立表面上每一行。可以用shadeValueInc值增加每个图素的shadeValue。这样，只要每16个图素取得正确的插值色档值即可，这就像优化纹理贴图时一样。

最后，如清单8.11所示，它实际生成ShadedSurface的代码。

清单8.11 用ShadedSurface类生成ShadedSurface.java

```
    /**
        生成指定多边形的ShadedSurface。从指定点光源表和环境光强度建立这个表面色档图
    */
    public static void createShadedSurface(
        TexturedPolygon3D poly, ShadedTexture texture,
        Rectangle3D textureBounds,
        List lights, float ambientLightIntensity)
```

```
{

    // 生成表面边界
    poly.setTexture(texture, textureBounds);

    Rectangle3D surfaceBounds = poly.calcBoundingRectangle();

    // 对 surfaceBounds 提供边界，纠正纹理贴图造成的少量误差。
    Vector3D du = new Vector3D(surfaceBounds.getDirectionU());
    Vector3D dv = new Vector3D(surfaceBounds.getDirectionV());
    du.multiply(SURFACE_BORDER_SIZE);
    dv.multiply(SURFACE_BORDER_SIZE);
    surfaceBounds.getOrigin().subtract(du);
    surfaceBounds.getOrigin().subtract(dv);
    int width = (int)Math.ceil(surfaceBounds.getWidth() +
        SURFACE_BORDER_SIZE*2);
    int height = (int)Math.ceil(surfaceBounds.getHeight() +
        SURFACE_BORDER_SIZE*2);
    surfaceBounds.setWidth(width);
    surfaceBounds.setHeight(height);

    //生成阴影表面纹理
    ShadedSurface surface = new ShadedSurface(width, height);
    surface.setTexture(texture, textureBounds);
    surface.setSurfaceBounds(surfaceBounds);

    // 生成表面色档图
    surface.buildShadeMap(lights, ambientLightIntensity);

    // 设置多边形表面
    poly.setTexture(surface, surfaceBounds);
}
```

8.6.5 缓存表面

前面提到表面会占用大量的内存，由于不可能预先计算游戏中每个多边形的表面，而且加上要动态生成的表面，所以只能对显示的多边形进行生成。

由于建立表面的成本昂贵，所以对每帧都生成表面是很不理智的。在实际中，让多边形绘制器把表面缓存在“显示”表中。表中只存储当前帧中显示的表面，而其他表面将从内存中删除，这样腾出空间来放更多当前表面的信息。

以上做法适用于许多情形，但有时多边形会经常出现，而且并不是在每帧都出现。例如，游戏者快速左右移动鼠标来观察周围的东西，则表面会出现并会产生重现的现象，而且每次重现时都要重新生成。

对于此种情形，内存回收单元可能根本没机会回收表面内存。我们与其重建表面，还不如让内存回收单元交回还没有回收的表面，这可以用软引用实现。

SofeReference 对象可以对对象进行软引用，而不是用通常的强引用。如果对象采用软引用（即软可访问），则内存回收单元可以将其回收。通常，内存回收单元先回收其他对象之后才回收软可访问对象。另外，所有软可访问对象清理后，虚拟机会抛出 outofMemoryException。

在 ShadedSurface 类中，对表面的缓冲区生成新的 SofeReference，代码如下。

```
/**
    生成新的表面，加上 SofeReference
*/
protected void newSurface (int width ,int height){
     buffer = new short[width*height]
     bufferReference = new SoftReference(buffer);
}
```

后面，如果多边形绘制器认为表面不再显示，则删除强引用：

```
/**
    清除这个表面，使内存回收单元可以在必要时从内存中删除它
*/
public void clearSurface(){
    buffer = null;
}
```

这时，缓冲区数组是软可访问，因此保存回收单元可以将其从内存中永久删除。

后面要恢复这个表面时，首先可以测试是否能从内存回收单元中取得。如果是，则 SofeReference 的 get()方法则返回这个对象，然后可以恢复强引用。

```
/**
    如果缓冲区已经建立和清除，但内存回收单元还没有将其从内存中删除，则这个方法可以从内
    存中取得。成功时返回真值
*/
public boolean retrieveSurface(){
     if(buffer == null){
        buffer = (short[]) bufferReference.get();
        }
        return !( buffer == null);
}
```

如果对象已经从内存中回收，则 get()方法返回 null，并需要重建表面。

简单地说，实现从回收站捡回表面的机制。

Sun 公司建议内存回收单元不要抛弃最近用过或最近生成的软可访问对象，但这不是对内存回收单元的要求。也许还要实现另一层缓存，但使用软引用是个好主意。

这样就完成了阴影表面。如清单 8.12 所示，实现 ShadedSurface 类其余的方法。

清单 8.12 ShadedSurface.java 类的其余方法

```
package com.brackeen.javagamebook.graphics3D.texture;

import java.lang.ref.SoftReference;
import java.util.List;
import com.brackeen.javagamebook.math3D.*;

/**
    ShadedSurface 是预先采用阴影的纹理，贴图到多边形上
*/
public final class ShadedSurface extends Texture {

    public static final int SURFACE_BORDER_SIZE = 1;

    public static final int SHADE_RES_BITS = 4;
    public static final int SHADE_RES = 1 << SHADE_RES_BITS;
    public static final int SHADE_RES_MASK = SHADE_RES - 1;
    public static final int SHADE_RES_SQ = SHADE_RES*SHADE_RES;
    public static final int SHADE_RES_SQ_BITS = SHADE_RES_BITS*2;

    private short[] buffer;
    private SoftReference bufferReference;
    private boolean dirty;
    private ShadedTexture sourceTexture;
    private Rectangle3D sourceTextureBounds;
    private Rectangle3D surfaceBounds;
    private byte[] shadeMap;
    private int shadeMapWidth;
    private int shadeMapHeight;

    // 增量计算色档值
    private int shadeValue;
    private int shadeValueInc;
```

```
    /**
        用指定宽度与高度生成 ShadedSurface
    */
    public ShadedSurface(int width, int height) {
        this(null, width, height);
    }

    /**
        用指定缓冲区、宽度与高度生成 ShadedSurface
    */
    public ShadedSurface(short[] buffer, int width, int height) {
        super(width, height);
        this.buffer = buffer;
        bufferReference = new SoftReference(buffer);
        sourceTextureBounds = new Rectangle3D();
        dirty = true;
    }
/**
    取得绘图中指定位置(x,y)的像素的 16 位颜色。x 与 y 值在表面边界之内，否则抛出
    ArrayIndexOutOfBoundsException
*/
public short getColor(int x, int y) {
    return buffer[x + y * width];
}

/**
    取得绘图中指定位置(x,y)的像素的 16 位颜色。x 与 y 值在表面边界之内，否则返回纹理边
    沿的像素
*/
public short getColorChecked(int x, int y) {
    if (x < 0) {
        x = 0;
    }
    else if (x >= width) {
        x = width-1;
    }
    if (y < 0) {
        y = 0;
    }
```

```
        else if (y >= height) {
            y = height-1;
        }
        return getColor(x,y);
    }

    /**
        表示这个表面是否脏的，如果表面是脏的，则可以外部清除
    */
    public void setDirty(boolean dirty) {
        this.dirty = dirty;
    }

    /**
        表示这个表面是否脏的，如果表面是脏的，则可以外部清除
    */
    public boolean isDirty() {
        return dirty;
    }

    /**
        设置这个 ShadedSurface 的源纹理
    */
    public void setTexture(ShadedTexture texture) {
        this.sourceTexture = texture;
        sourceTextureBounds.setWidth(texture.getWidth());
        sourceTextureBounds.setHeight(texture.getHeight());
    }

    /**
        设置这个 ShadedSurface 的源纹理与源边界
    */
    public void setTexture(ShadedTexture texture,
        Rectangle3D bounds)
    {
        setTexture(texture);
        sourceTextureBounds.setTo(bounds);
    }
```

```
/**
    设置这个 ShadedSurface 与源边界
*/
public void setSurfaceBounds(Rectangle3D surfaceBounds) {
    this.surfaceBounds = surfaceBounds;
}

/**
    取得这个 ShadedSurface 与源边界
*/
public Rectangle3D getSurfaceBounds() {
    return surfaceBounds;
  }
```

除了脏标志之外，ShadedSurface 类中几乎一切都已经介绍了。脏标志和 ShadedSurface 本身并无关。其实 ShadedSurface 不关心标志是否脏，因为外部可以使用脏标志，标明表面是否要清除。如清单 8.13 所示的 ShadedSurfacePolygonRenderer 就显示这个思路，它与其他绘制器差不多，但保存建立的每个表面清单。前面曾介绍过，只建立显示表面。如果不是显示表面（标为脏），则清除表面数据，并从清单中删除这个表面。

清单 8.13 ShadedSurfacePolygonRenderer.java

```
package com.brackeen.javagamebook.graphics3D;

import java.awt.*;
import java.awt.image.*;
import java.util.List;
import java.util.LinkedList;
import java.util.Iterator;

import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.graphics3D.texture.*;

/**
    ShadedSurfacePolygonRenderer 是个 PolygonRenderer ，用 ShadedSurfaces 绘制多边形，它跟踪建
    立的表面，清除上一绘制帧未用的表面以节省内存
*/
public class ShadedSurfacePolygonRenderer
    extends FastTexturedPolygonRenderer
{
```

```
    private List builtSurfaces = new LinkedList();

    public ShadedSurfacePolygonRenderer(Transform3D camera,
        ViewWindow viewWindow)
    {
        this(camera, viewWindow, true);
    }

    public ShadedSurfacePolygonRenderer(Transform3D camera,
        ViewWindow viewWindow, boolean eraseView)
    {
        super(camera, viewWindow, eraseView);
    }

    public void endFrame(Graphics2D g) {
        super.endFrame(g);

        // 清除这一绘制帧未用的表面以节省内存。
        Iterator i = builtSurfaces.iterator();
        while (i.hasNext()) {
            ShadedSurface surface = (ShadedSurface)i.next();
            if (surface.isDirty()) {
                surface.clearSurface();
                i.remove();
            }
            else {
                surface.setDirty(true);
            }
        }
    }

    protected void drawCurrentPolygon(Graphics2D g) {
        buildSurface();
        super.drawCurrentPolygon(g);
    }

    /**
        建立清除了 ShadedSurface 的多边形表面
    */
```

```
    protected void buildSurface() {
        // 必要时建立表面
        if (sourcePolygon instanceof TexturedPolygon3D) {
            Texture texture =
                ((TexturedPolygon3D)sourcePolygon).getTexture();
            if (texture instanceof ShadedSurface) {
                ShadedSurface surface =
                    (ShadedSurface)texture;
                if (surface.isCleared()) {
                    surface.buildSurface();
                    builtSurfaces.add(surface);
                }
                surface.setDirty(false);
            }
        }
    }
}
```

ShadedSurfacePolygonRenderer 对每个多边形都调用 buildSurface()方法。如果表面清除，则要重建（或从内存回收单元取得）并加回建立表面清单中。

8.6.6　阴影表面演示程序

下面要生成阴影表面演示程序。ShadingTest2 类和 TextureMapTest2 相似，它生成几个 Point Light3D 和使用 ShadedSurfacePolygonRenderer，如图 8-16 所示它的屏幕图形。

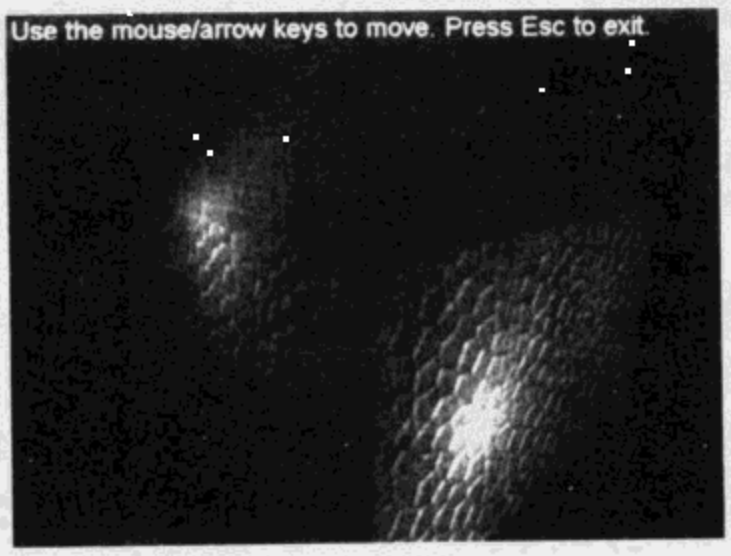


图 8-16　ShadingTest2 阴影表面演示程序

和平常一样，在房子里漫步并从各边观察。房子的环境光强度较弱，但有 2 个强点光源，它们显示一些逼真、醒目的灯光。

8.7 其他概念

本节介绍几个思路，来扩展本章生成的纹理贴图和照明特性技术。

当前只是涉及纹理贴图和照明的皮毛。阴影表面只对纹理采用阴影，同时还可以产生各种其他的效果，包括动态生成的效果。例如。如果游戏者在墙上使用火焰喷射器，则可以采用“燃烧”效果，还可以加一些表面“裂纹”，使纹理不那么单调，或在砸坏机器人时在地上显示机器人爆炸的标志。

8.7.1 深度提示

另一种灯光技巧是让越远的物体光线越暗，使灯光随着距离慢慢减弱。这种方法称为深度提示(Dipth Cueing)，在 Doom 等游戏中非常普及，是任天堂 64 游戏中的常见效果。在游戏中，纹理越远则越暗，但深度提示并不局限于黑色。任天堂 64 游戏中有些游戏采用“雾化”效果，物体淡化成灰色调。

8.7.2 伪阴影

下一章“三维实体”要生成各个物体，可以在三维世界中漫游。但有时很难判断物体相对于地的关系，有时物体像是浮在半空中。为了解决视觉缺陷，一个常见方法是在物体周围画上图形的伪影子。之所以是伪影子，是因为它并不反映物体形状（例如机器人的影子看不到机器人伸出的手臂），但可以方便地表示物体位置。

8.7.3 MIP 贴图

对非常远的多边形生成大表面是多余的，更好的方法是多边形离镜头越远，则表面较小。

这种方法称为 MIP 贴图(MIP mapping)，MIP 指 Parvum，是拉丁语，表示多合一。远处的多边形表面较小，表面越接近镜头，生成的表面越大。这种方法可以节省大量内存，使远离镜头的表面更光顺。

8.7.4 双线性插值

运行本章的演示程序时，注意当很接近纹理时，图素呈块状。同样，远离纹理时，会有些变形，因为屏幕上并不绘制每个图素。这在纹理具有锐边时特别明显。

双线性插值可以解决这 2 个问题。只要像在色档图中相邻值之间进行插值一样，就可以在纹理相邻图素之间进行插值，使其更光顺，使靠近的纹理不会呈块，远处的纹理不会变形。

本书采用的纹理不用或少用锐边，因此这个效果不明显，但双线性插值仍然可以改进效果。如果使用“测试纹理”之类的纹理，则会看到各种奇怪的伪差。双线性插值可以大大减少这类伪差。

8.7.5 三线性插值

将双线性插值和 MIP 贴图结合起来，除了在纹理的相邻图素之间插值外，这种方法还

在相邻 MIP 贴图之间插值（一个 MIP 贴图远离镜头，一个 MIP 贴图靠近镜头）。这样就在从一个 MIP 贴图移到另一 MIP 贴图时消除了明显的变化，因为我们用 2 个 MIP 贴图并在 2 个 MIP 贴图之间插值。

8.7.6　法线图与深度

本章使用的有些纹理本身带有些“伪”光，使纹理显示一定的深度，如石头的突出部分比较亮，四周比较暗，带有阴影。

当然，这种“伪”光并不准确，因为它没有考虑周围环境的光线。“伪”光可以假设光源在左上角，但实际环境中的光源可能来自不同方向。

有些三维加速卡可以用法线图和深度生成与环境光对应的纹理。

这里不是用纹理图，而是用 3 个图：颜色图（没有光线）、深度图（指定颜色图中每个像素的“深度”）和法线图（指定颜色图中每个像素法线向量）。利用颜色、高度和法线向量可以求出每个像素的正确光线，从而生成与环境光对应的纹理。

尽管这个过程的成本很高，但有些三维加速卡可以很快地实时将这 3 个图结合起来。

8.7.7　其他类型的光

最后还要介绍一下光源。这里只用了各个方向均匀辐射的光源，如阳光。为了扩展照明体系结构，可以增加彩色光、锥形光（沿锥形辐射，而不是各个方向均匀辐射）或区域光，其可能来自矩形光源，像三棱镜一样辐射。

虽然本演示中看不到，但光可以穿过多边形，照到另一边的多边形。最好消除这种情形，保证照在表面上的光束没有先照到另一表面。

8.8　小结

本章介绍了大量概念，包括快速纹理贴图、扩展光和用色档图的照明，还介绍了大量优化技术、阴影纹理的高效存储以及用软引用缓存表面。我们还介绍了如何改进生成的绘制器，增加高级光源、MIP 贴图和双线性插值等特性。

下一章要介绍三维游戏实体的动画，最终要将绘制器扩展成能够用 Z 缓冲处理除凸多面体以外的其他实体。

第 9 章　三维对象

关键问题

- 表面消隐
- 三维动画
- 多边形组
- 从 OBJ 文件装入多边形组
- 游戏实体
- 管理游戏实体
- 综合起来
- 改进
- 小结

在典型游戏中，游戏者要控制对象撞击、抓取、跳上、跳过、吃掉、捉住、躲避、推动、战斗、扔、射击等等。

本章要生成三维实体动画系统，移动和旋转世界中的实体。为了绘制三维实体，可以用 Z 缓存按正确的 Z 顺序画多边形。我们还要生成分析器，从 OBJ 文件读取三维实体，以免在 Java 代码中定义大量多边形。最后，我们要综合起来，生成几个简单游戏实体和一个游戏实体管理器。利用这些方法可以更接近生成真实三维游戏，向游戏者提供一些交互实体。

9.1　表面消隐

绘制多边形时的问题是，画面成看不到的被其他多边形隐藏或部分隐藏的多边形。这个问题可以通过正确的表面消隐算法或组合几种表面消隐算法解决。

前面几章只用了背面消除和剪取方法作为表面消隐算法。使用背面消除方法时，只能画凸多面体。本节要介绍另外几个表面消隐算法，可以绘制组成任何形状的多边形。

记住，不同表面消隐算法适用于不同情形。适合静态的算法不一定适合运行三维实体。本章选择的算法最适合运动和旋转的三维实体。

9.1.1　油画算法

一个基本表面消隐算法是油画算法（painter′s algorithm）。油画家绘制油画时，首先画背景，然后画离前景更近的实体，这样一层一层画，直到画完最前面的实体。使用这个方法时，前景中所画的内容遮住布上已有的内容。对三维图形而言，就是由后往前画多边形。

要实现油画算法，就要相对镜头按由后往前的顺序存放所有多边形。这里的难点是取得正确的排序算法。可以按每个多边形的中点排序，但这不能保证完全正确。以图 9-1 所

示的两个投影多边形为例。

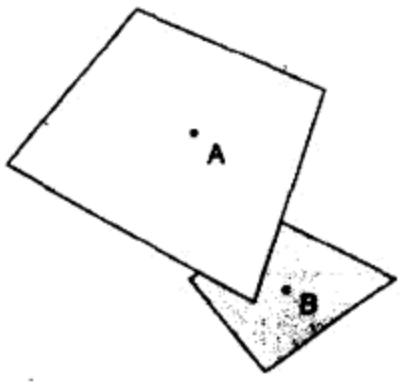


图 9-1 两个投影多边形，油画算法认为要先画多边形 B，因为它在多边形 A 后面

对于图 9-1 所示的两个投影多边形，油画算法认为要先画多边形 B，因为它在多边形 A 后面。但如果用离镜头的距离测量，则会遇到问题。如果按多边形的中点排序，则多边形 A 的中点可能在多边形 B 的中点后面，要先画多边形 A，这显然是错误的。

油画算法的另一个问题是由后往前画涉及大量重画（一个像素绘制多次），因此不是绘制整个视景的理想方案。

9.1.2 反油画算法

要克服油画算法的问题，可以用反油画算法（Reverse painter′ s algorithm）。反油画算法由前往后画，而不是由后往前画。使用反油画算法时，我们不画任何已经绘制的像素，因此不会造成任何重画，只在继续画多边形，直到画完整个视图窗口。但是，反油画算法仍然要有 100%正确的多边形排序算法。另外，它只适用于多边形完全在镜头周围（闭合区）时，这样才能填满视图。

9.1.3 Z 缓存

Z 缓存也称为深度缓存（depth buffering），没有油画算法的问题，而且更容易实现。使用 Z 缓存时，使用与视图窗口一样大小的缓冲区记录多边形中每个像素的深度。绘制像素的条件是像素深度比 Z 缓存中该位置的当前深度更接近。从而得到像素完美的三维视景，不管多边形按什么顺序绘制。图 9-2 显示了这个思路。

在图 9-2 所示的例子中，Z 缓存最初是空的，每个像素的深度设置为无穷大。第一个多边形的每个像素比 Z 缓存中的值更近，因此绘制每个像素并将 Z 缓存设置为多边形的深度值。第二个多边形实际上与第一个多边形相交，因此第二个多边形在第一个多边形后面的部分不画出来。

Z 缓存的缺点是缓存需要更多内存。对 16 位深度，1024×768 的屏幕尺寸需要 1.5MB 内存，但相对于当前大多数计算机的内存量而言，这个内存要求并不高。

主要缺点是画多边形需要更多时间。检查要画的每个像素的 Z 缓存需要一些开销，如果屏幕上每个像素都使用 Z 缓存，则这一点特别明显。在本书许多例子的测试中，如果要绘制屏幕上每个像素，则速度比普通纹理贴图慢 2～3 倍。

但是，由于三维实体通常只占用少量显示屏幕空间，因此开销不像全都用 Z 缓存时那

么糟。

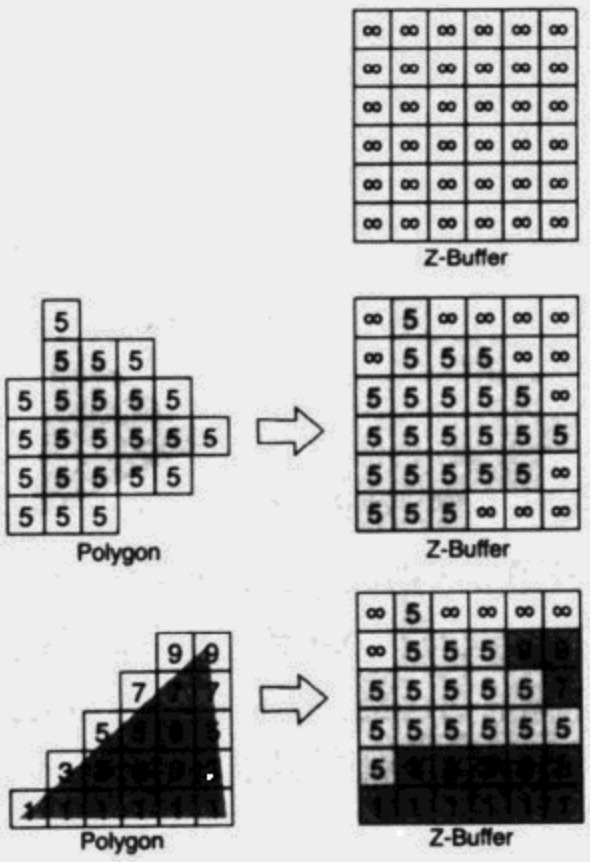


图 9-2　Z 缓存可以正确绘制多边形，数字表示多边形中每个像素的深度

9.1.4　使用 1/Z 的 Z 缓存

Z 缓存的一个问题是，不管多边形离镜头多远，其精度都是相同的。例如，如果 Z 缓存的精度为 1 个单位（或 1 个像素），则靠近镜头的重叠多边形不能很好地合并，部分多边形的深度为 10.7 不是 11。靠近镜头的多边形实际上需要更细的精度，为此可以用两种解决办法：

- 对 Z 缓冲区深度使用 32 位浮点数（或 32 位定点数）。这样更精确，但占用的内存加倍，而且可能更慢。
- 使用深度的倒数 1/Z。

使用 Z 缓存时，实际上并不需要屏幕上每个像素的准确深度值，只要正确比较像素之间的深度，仍然可以用 1/Z 实现。此外，1/Z 在屏幕空间中是线性的（如图 9-3 的例子所示），这样有两个好处：

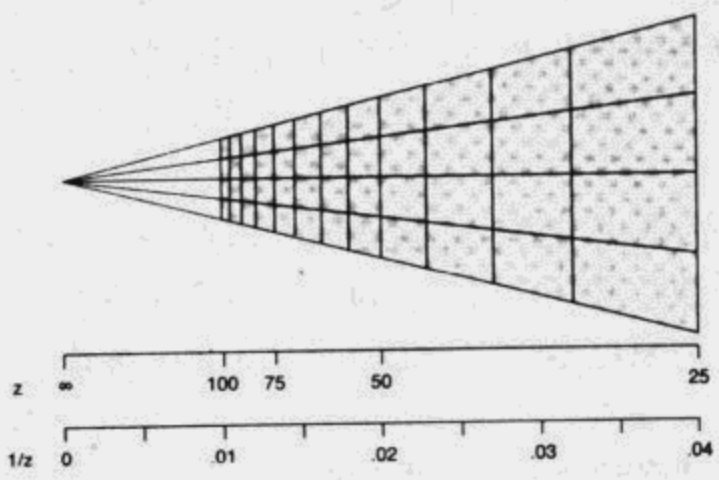


图 9-3　1/Z 在屏幕空间中是线性的

- 用 1/Z 使靠近镜头的实体有更高的深度精度，这是最需要的。
- 由于 1/Z 在屏幕空间中是线性的，与实际 Z 深度不同，因此很容易设置水平扫描的深度，比纹理贴图更容易（其在屏幕空间中不是线性的）。我们不必像纹理贴图中一样对每 16 个像素插值，因此更快更准确。

使用 1/Z 的 Z 缓存是本书的三维实体所用的方法。由于深度使用 16 位整数值，因此深度在 0 与 Short.MAX_VALUE 之间。假设 Z≥1，则可以使用下列公式：

Short.MAX_VALUE/z

使用这个公式时，数值越大深度越接近，数值为 0 时深度为无穷大。

下面要实现 Z 缓存。清单 9.1 所示的 Zbuffer 类生成的基本 16 位 Z 缓存值。

清单 9.1　Zbuffer.java

```
package com.brackeen.javagamebook.graphics3D;

/**
    ZBuffer 类实现 Z 缓存或深度缓存，使用 Z 缓存时，用来记录在 3D 视窗中每个像素的深度。
    每个像素记录的值为深度的倒数(1/Z)，用 1/Z 使靠近镜头的实体有更高的深度精度，这是最需
    要的。越远的实体精度越低（其视觉重要性也越低）
*/
public class ZBuffer {

    private short[] depthBuffer;
    private int width;
    private int height;
    /**
        用指定宽度与高度生成新 Z 缓存
    */
    public ZBuffer(int width, int height) {
        depthBuffer = new short[width*height];
        this.width = width;
        this.height = height;
        clear();
    }
    /**
        取得 Z 缓存宽度
    */
    public int getWidth() {
        return width;

    }
```

```
    /**
        取得 Z 缓存高度
    */
    public int getHeight() {
        return height;
    }
    /**
        清除 Z 缓存，将所有深度值设置为 0
    */
    public void clear() {
        for (int i=0; i<depthBuffer.length; i++) {
            depthBuffer[i] = 0;
        }
    }
    /**
        设置指定偏移量的像素深度，覆盖当前深度
    */
    public void setDepth(int offset, short depth) {
        depthBuffer[offset] = depth;
    }
    /**
        检查指定偏移量的像素深度。如果指定深度较低（大于或等于指定偏移量的当前深度），
        则设置深度，这个方法返回真值，否则不采取任何操作，这个方法返回假值
    */
    public boolean checkDepth(int offset, short depth) {
        if (depth >= depthBuffer[offset]) {
            depthBuffer[offset] = depth;
            return true;
        }
        else {
            return false;
        }
    }
}
```

注意：Zbuffer 类有两个方法操纵特定偏移量处的 Zbuffer：setDepth()与 checkDepth()。setDepth()方法只是设置 Z 缓冲区中特定偏移量处的深度，删除已有的值。checkDepth()方法只在指定深度较低（大于或等于指定偏移量的当前深度）时才设置深度，这个方法返回真值；否则不采取任何操作，这个方法返回假值。

最后，clear()方法清除 Z 缓存，将所有深度值设置为 0，相当于无穷深度。这时每帧都要调用 clear()方法。

9.1.5　计算 Z 深度

下面要求出多边形中计算每一点深度的方程。

实际上我们已经有这个方程。第 8 章“纹理贴图与照明”曾介绍过，我们需要 P_z，是视图窗口中特定点 W 的多边形深度，用于确定纹理贴图的方法：

$P_z = d(U \times V \cdot O) / (U \times V \cdot W)$

这里 d 是从镜头到视图窗口的距离，U 和 V 纹理 x 与 y 方向，O 是纹理原点（或多边形平面上的任何其他点）。这样就可以了（那一章已经推导出纹理贴图方程）。这个方程的逆运算如下：

$1/z = (U \times V \cdot W) / (d(U \times V \cdot O))$

这可以对整个多边形进行计算，而不是对每个像素计算：

```
float w = SCALE * MIN_DISTANCE * Short.MAX_VALUE/
      (viewWindow.getDistance() *
       c.getDotProduct(textureBounds.getOrigin()));
```

记住，向量 C 设置为 U×V，因此不必重新计算 U×V。

在方程中，将数值乘一个比例因子，将其转换成定点值。

另外，还有一个乘数是 MIN_DISTANCE，是使用 Z 缓存的最小距离。这个值设置为 12，在绘制远处实体时更精确，但距离小于 12 时则得不到正确深度。这是没问题的，因为第 11 章“碰撞探测”要实现碰撞探测，我们看不到距离小于 12 时的实体。

最后，纹理贴图时，纹理贴图到水平扫描线之前要计算第一个像素的深度（depth）和深度变化（dDepth）：

```
float z = c.getDotProduct(viewPos);
float dz =c.x
...
int depth = (int) (w*z);
int dDepth = (int) (w*dz);
```

然后对扫描的每个像素，检查深度，在设置深度时设置像素，并增加深度：

```
...
if (zBuffer.checkDepth (offset, (short) (depth>>SCALE_BITS))){
     doubleBufferData [offset] =
        texture.getColor(tx>>SCALE_BITS, ty>> SCALE_BITS);
}
depth+=dDepth;
...
```

这样就完成了使用 Z 缓存的绘制。我们用 ZbufferedRenderer 类总结计算多边形中每个像素深度的过程，这个类是 ShadedSurfacePolygonRenderer 的子类。它和父类差不多，只是绘制时检查所画的每个像素深度，并在每帧开头清除 Z 缓存（或在视图窗口尺寸改变

时生成新的 Z 缓存)：

```
public void starFrame (Graphics2D g) {
    super.startFrame(g);
    //初始化深度缓存
    if zBuffer ==null ||
        zBuffer.getWidth() !=viewWindow.getWidth() ||
        zBuffer.getHeight() !=viewWindow.getHeight ())
    {
        zBuffer = new ZBuffer(
            viewWindow.getWidth(),viewWindow.getHeight ()
    }
    else if (clearViewEveryFrame){
            zBuffer.clear();
    }
}
```

这样就完成了使用 Z 缓存的问题。下面要实际利用 Z 缓存，生成在边界上移动的三维实体。

9.2 三维动画

第 7 章“三维图形”的第一个演示程序中生成了一个简单树，在镜头前面移近移远，沿 y 轴旋转。在这个演示程序中，镜头还没有移动，而是对树的多边形进行变换。

我们用相同的思路移动和旋转游戏中的三维实体：每个三维实体有自己的变换。三维实体指一组多边形联系起来，构成多边形网格（polygon mesh）或多边形模型（polygon modal）。这个多边形组可以独立移动。例如，构成机器人的所有多边形属于一个三维实体，共享相同的变换。

这个方法把三维实体放在独立坐标系中，不同于世界坐标系，如图 9-4 所示。

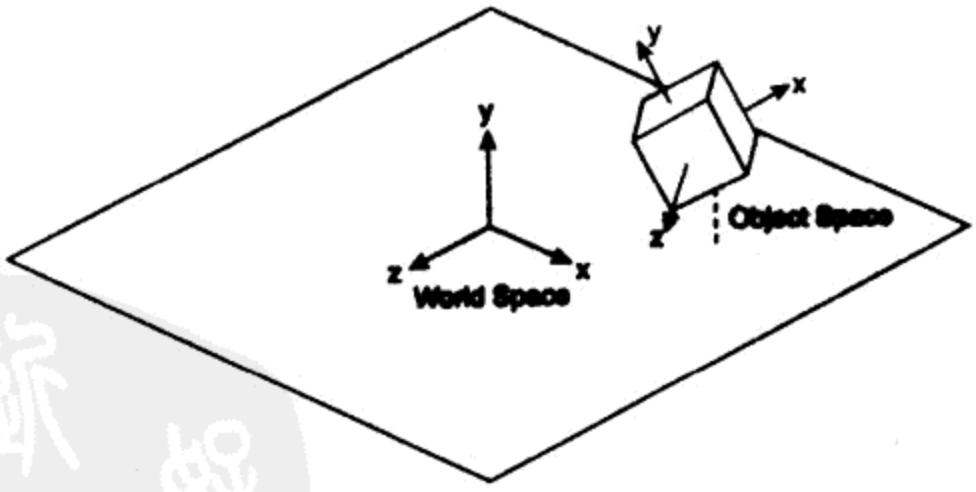


图 9-4 三维实体的坐标系与世界坐标系

三维实体中的多边形并不显式移动与旋转，因为这些多边形是静态的，只改变实体的变换。前几章曾介绍过，如果实际移动与旋转三维实体的多边形，则浮点误差不断积累，

最终会使实体扭曲。绘制过程中，可以将每个多边形复制到一个临时多边形中，通过其实体变换进行变换，像第 7 章的树一样。

要在三维引擎中完成独立运动，只要对每个实体指定自己的 Transform3D，这是第 7 章介绍的变换类。

但是，为了更方便地让实体移动，我们要扩展 Transform3D 类，增加不同类型的运行。

记住，运动有两种：线性运动（实体位移）和角度运动（旋转实体）。角度运动又分为 3 种，分别沿 x 轴、y 轴和 z 轴运动。

另外，我们希望不同实体按不同速度移动与旋转，有时要让一个实体移动与旋转一段时间，然后停止。例如，我们要让机器人转到一定方向，然后停止；或移到靠近游戏者，然后停止。这样，就要让实体移动达到更高级的目的：

- 将实体以速度 s 移到位置(x、y、z)。
- 下一个 t 秒或永远将实体以速度 s 在(x、y、z)方向移动。
- 下一个 t 秒或永远将实体以速度 s 绕一个轴旋转。
- 将实体以速度 s 转到 3π/4 弧度方向。
- 停止一个对象的运动。

所有移动与旋转图标至少可以变成“以速度 x 移动 t 时间”。指定时间到达后，移动停止（有些移动目标没有时间限制，因此要想办法让这个运动一直进行）。另外，除了速度与时间，线性运动还需要方向。

可以在 Transform3D 的子类 MovingTransform3D 中完成这些目标。这个一般性运动主要是在 MovingTransform3D 的 Movement 内类中完成的，见清单 9.2。

清单 9.2　MovingTransform3D 类的 Movement 内类

```
/**
    Movement 类指定运动的速度与时间
*/
protected static class Movement{
    //每毫秒变化
    float speed;
    long remainingTime;
    /**
        将这个运动设置为指定速度与时间（毫秒）
    */
    public void set (float speed, long time){
        this.speed=speed;
        thid.remainingTime = time;
    }

    public boolean isStopped(){
        return (speed ==0)|| (remainingTime ==0);
```

```
    }
    /**
        取得指定时间（毫秒）经过的距离
    */
    public float getDistance (long elapsedTime){
        if (remainingTime ==0){
            return 0;
        }
        else if (remainingTime !=FOREVER){
            elapsedTime = Math.min(elapsedTime, remainingTime);
            remainingTime-= elapsedTime;
        }
        return speed * elapsedTime;
    }
}
```

Movement 内类只是跟踪一般性运动的速度和余下时间。余下时间也可以用 FOREVER，表示运动永不停止。例如，射向天空的子弹不会停止。GetDistance()方法取得指定时间（毫秒）经过的距离（距离＝速度×时间）。这个方法中，remainingTime 减去 elapsedTime，直到结果为 0。

这里跟踪余下时间而不是绝对结束时间。如果跟踪绝对结束时间，则操纵时间流程的项目会使运动不正确，如暂停游戏和保存与重装游戏。例如，如果你在 19：58 说“向前移动，直到 20：00”，但用户在 19：59 暂停游戏，到 20：01 分才恢复，则用个恢复游戏时，运动停止，这是不对的，因为运动只进行了一分钟，而我们要它进行两分钟。

这里 Movement 类的一个实例用于空间运动，3 个实例用于角度运动（各对一个轴）。首先要介绍和实现线性运动。

9.2.1 空间运动

和第 2 章“二维图形与动画”介绍的幽灵一样，变换要指定采用空间运动的速度。首先要在 MovingTransform3D 类中定义两个字段，表示变换速度：

```
//速度（单位/毫秒）
private Vector3D velocity;
private Movement velocityMovement;
```

由于速度有大小和方向，因此这里并不需要 Movement 类的速度字段。

对速度而言，只在速度大小不是 0，就把 Movement 类的速度设置为 1，否则把 Movement 类的速度设置为 0。

下面生成几个方法，显示设置变换速度：

```
/**
    将速度设置为指定向量
*/
```

```
public void setVelocity(Vector3D v){
      setVelocity(v,FOREVER);
}
/**
    设置速度。经过指定时间后，自动将速度设置为 0。如果指定时间为 FOREVER，则永远不会
    自动将速度设置为 0
*/
public void setVelocity (Vector3D v,long time){
     if (velocity != v){
         velocity.setTo(v);
     }
     if (v.x == 0 && v.y == 0 && v.z == 0){
         velocityMovement.set(0,0);
     }
     else{
     velocityMovement.set(1,time);
   }
 }
/**
    在当前速度上增加指定速度。如果这个 MovingTransform3D 当前移动，则余下时间不变，否则
    余下时间设置为 FOREVER
*/
public void addVelocity(Vector3D v){
      if (isMoving()){
        velocity.add(v);
      }
      else{
           setvelocity(v);
      }
}
/**
    如果这个 MovingTransform3D 当前移动，则返回真值
*/
public boolean isMoving() {
    return !velocityMovement. isStopped() &&
         !velocity.equals(0,0,0);
}
```

setVelocity 方法设置速度。经过指定时间后，自动将速度设置为 0。如果指定时间为 FOREVER，则永远不会自动将速度设置为 0。AddVelocity()方法在当前速度上增加指定速

度（例如在运动实体上增加重力）。最后，可以用 isMoving()方法检查变换实体是否运动。

下面要用一个方法显式定义移动哪里：

```
/**
    将速度设置为按指定速率移到下列目标
*/
public void moveTo(Vector3D destination,float speed){
     temp.setTo (destination);
     temp.subtract(location);
     //计算移动所需时间
     float distance = temp.length();
     long time = (long) (distance/speed);

     //范化方向向量
     temp.divide (distance);
     temp.multiply(speed);

      setVelocity(temp,time);
 }
```

moveTo()方法实际上是个方便方法，计算按指定速率移到指定目标地点所需时间和速度（大小与方向）。它用一个临时向量 temp 并调用 setVelocity()方法设置速度。这样就可以不计算方向与时间量而告诉实体要移动哪里。

9.2.2 角度运动

角度运动或旋转类似于速度，但不需要速度向量，只要 3 个 Movement 实例：

```
//角度运动（弧度/毫秒）
private Movement velocityAngleX;
private Movement velocityAngleY;
private Movement velocityAngleZ;
```

每个轴的角度运动差不多，因此我们只介绍 y 轴。首先，下列方法设置角速度并检查实体目前是否绕 y 轴旋转：

```
/**
    设置 y 轴角速度
*/
public void setAngleVelocityY(float speed){
    setAngleVelocityY(speed,FOREVER);
}
/**
    设置一段时间内 y 轴角速度
*/
```

```
public void setAngleVelocityY(float speed,long time){
   setAngleVelocityY(speed,time);
}
/**
    检查实体目前是否绕 y 轴旋转
*/
public boolean isTurningY(){
     return !velocityAngleY.isStopped();
}
```

使用 setAngleVelocityY()方法时，正速率表示逆时针旋转，负速率表示顺时针旋转。

这里的难点是确定实体要朝哪个方向。实体朝某个方向时，要计算顺时针转与逆时针转哪个更快。

通常，解很简单，但也不一定。以图 9-5 为例，图中开始位置在接近-π的一端，而结束位置在接近π的一端。从图上看，很容易判断怎么转更快，但从-π到π的线性标量很难看出，如图 9-2（同样，如果让实体从-π转到π，显然不必让它转一圈）。

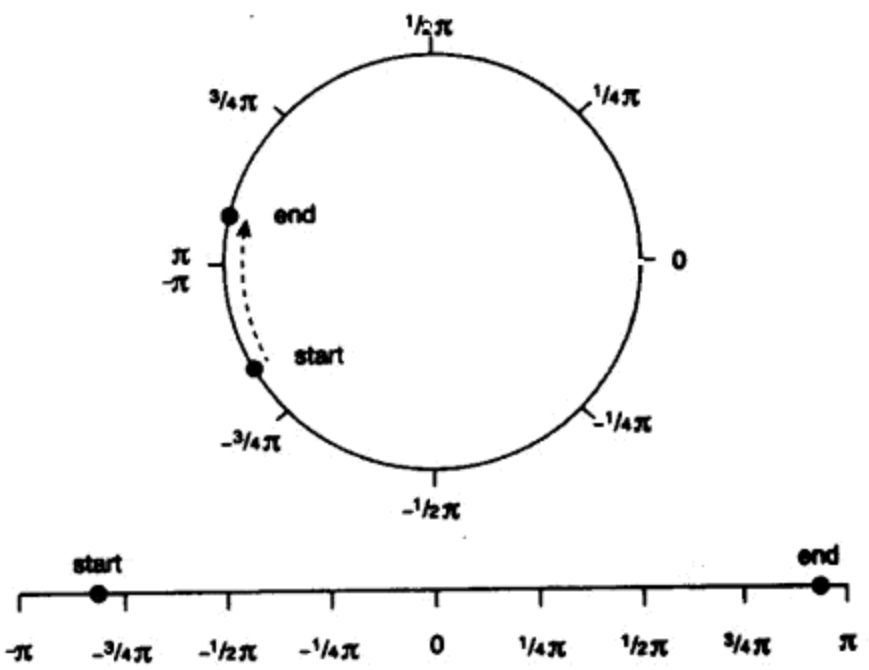


图 9-5　找出两个角度之间的最短角距离

要解决这个问题，可以计算两个距离：顺时针到达目标的角距离和逆时针对达目标的角距离，根据较短的角距离选择转动方向。turnTo()方法如下：

```
/**
    用指定速率将 y 轴转到指定角度
*/
public void  turnYTo (float angleDest, float speed)
    turnTo (velocityAngleY,getAngleY()angleDest, speed);
}
/**
    用指定速率将 y 轴转到向量（x,z）指定的方向
```

```
*/
public void  turnYTo (float x, float z, float angleoffset,
    float speed)
{
    turnYTo((float) Math.atan2(-z,x)+ angleoffset, speed);
}
/**
    用指定速率转动 starAngle 到 endAngle 的角度
*/
protected void turnTo(Movement movement,
      float startAngle, float endAngle,float speed)
{
      startAngle = ensureAngleWithinBounds(startAngle);
      endAngle = ensureAngleWithinBounds(endAngle);
      if (startAngle == endAngle){
          movement.set(0,0);
      }
      else{

          float distanceLeft;
          float distanceRight;
          float pi2 = (float) (2*Math.PI);

          if (startAngle < endAngle){
              distanceLeft = startAngle - endAngle+ pi2;
              distanceRight = endAngle-startAngle ;
          }

          if (distanceLeft < distanceRight){
               speed = -Math.abs(speed);
               movement.set(speed,(long)( distanceLeft / - speed));
          }
          else{
               speed = -Math.abs(speed);
               movement.set(speed,(long)( distanceRight / - speed));
          }
      }
}
/**
```

```
        保证指定角度为-pi 与 pi 之间。返回角度，不在这个范围之内，则纠正到这个范围之内
    */
    protected  float ensureAngleWithinBounds(float angle){
          if(angle < -Math.PI  || angle > Math.PI){
            //将范围变成（0 到 1）
            double newAngle =( angle +  Math.PI) / (2*Math.PI);
           //验证范围
            newAngle = newAngle - Math.floor(newAngle);
           //将范围变成（-pi 到 pi）
            newAngle = Math.PI * (newAngle * 2  - 1);
            return( float) newAngle;
          }
          return angle;
    }
```

ensureAngleWithinBounds()由于 turnTo()方法要求-π到π之间的角度，因此用 ensureAngleWithinBounds()方法在必要时将角度变成-π到π之间的角度。

这样就完成了 MovingTransform3D，清单 9.3 显示了这个类其他部分。

清单 9.3　MovingTransform3D.java 其余部分

```
package com.brackeen.javagamebook.math3D;
/**
    MovingTransform3D 是个 Transform3D，具有较慢速率和绕 x、 y、 z 轴转动的角速率
*/
public class MovingTransform3D extends Transform3D {

    public static final int FOREVER = -1;
   // 计算所用的 Vector3D
    private static Vector3D temp = new Vector3D();

   // 速度（单位/毫秒）
    private Vector3D velocity;
    private Movement velocityMovement;

   // 角速度（弧度/毫秒）
    private Movement velocityAngleX;
    private Movement velocityAngleY;
    private Movement velocityAngleZ;
    /**
        生成新 MovingTransform3D
```

```
    */
    public MovingTransform3D() {
        init();
    }
    /**
        用指定 Transform3D 的值生成新 MovingTransform3D
    */
    public MovingTransform3D(Transform3D v) {
        super(v);
        init();
    }

    protected void init() {
        velocity = new Vector3D(0,0,0);
        velocityMovement = new Movement();
        velocityAngleX = new Movement();
        velocityAngleY = new Movement();
        velocityAngleZ = new Movement();
    }

    public Object clone() {
        return new MovingTransform3D(this);
    }
    /**
        根据经过时间更新 Transform3D ，更新位置和角度
    */
    public void update(long elapsedTime) {
        float delta = velocityMovement.getDistance(elapsedTime);
        if (delta != 0) {
            temp.setTo(velocity);
            temp.multiply(delta);
            location.add(temp);
        }

        rotateAngle(
            velocityAngleX.getDistance(elapsedTime),
            velocityAngleY.getDistance(elapsedTime),
            velocityAngleZ.getDistance(elapsedTime));
    }
```

```
/**
    停止这个 Transform3D，将所有移动速度设置为 0
*/
public void stop() {
    velocity.setTo(0,0,0);
    velocityMovement.set(0,0);
    velocityAngleX.set(0,0);
    velocityAngleY.set(0,0);
    velocityAngleZ.set(0,0);
}
/**
    取得这个运动余下时间量
*/
public long getRemainingMoveTime(){
   if (!isMoving()){
        return 0;
   }
   else{
       return velocityMovement.remainingTime;
   }
    }
}
```

注意：每帧都要调用 Transform3D 的 update()方法，这个方法进行所有工作，根据经过时间更新位置和角度。

另外，由于多边形是静态的，因此绘制多边形时一定要采用 Transform3D。下面要生成一个类，组织相关多边形和对多边形采用变换。

9.3　多边形组

本节实现多边形组的容器，每个多边形组有自己的变换。

前面只介绍了移动整个三维实体，但实际中可能要独立移动三维实体的不同部分。例如，机器人的手与身体可能要单独移动，手指与手臂可能也要单独移动。机器人移动时，手臂跟着移动（除非游戏者很早就把机器人的手臂“截肢”）。

为了实现这类运动，可以让多边形组建立子多边形组，各有自己的变换，使每个多边形组有一个坐标系。例如。图 9-6 的三维实体实际上由两个多边形组构成，即转台与基座。这样，转台可以独立于基座旋转和移动。

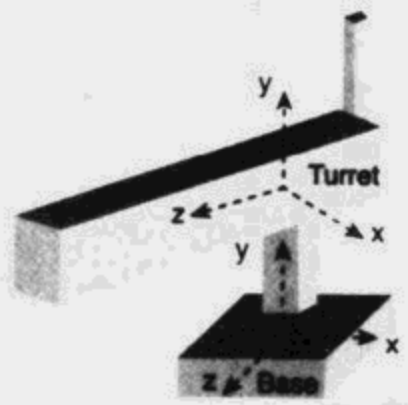


图 9-6 可以让多边形组建立子多边形组，每个多边形组有一个坐标系。本例中转台可以独立于基座旋转和移动

本例中，转台位于基座组右上方居中位置，这样，转台左右旋转时，沿实体的基座旋转。不能让转台沿天线旋转，否则没有和基座连在一起。

即转台与基座有自己的变换，整个实体也有自己的变换。这样表示多边形组即生成变换层次，如图 9-7 所示。

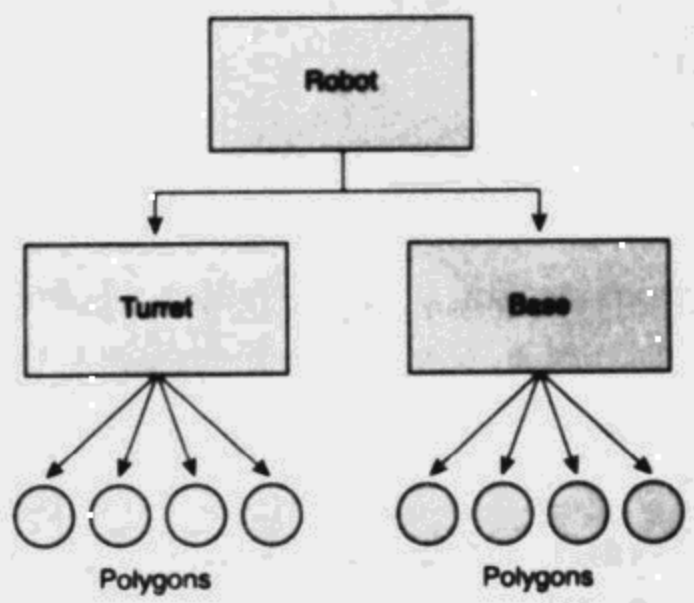


图 9-7 三维实体表示成多边形组（矩形）和多边形（园形）层次

绘制多边形时，首先要采用父变换，然后采用它的变换，这样一直到达根变换。

清单 9.4 的 PolygonGroup 类显示了实现这些多边形组的代码。

清单 9.4 PolygonGroup.java

```
package com.brackeen.javagamebook.math3D;

import java.util.List;
import java.util.ArrayList;
/**
    PolygonGroup 是带 MovingTransform3D 的多边形，PolygonGroup 还可以包含其他 PolygonGroup
*/
public class PolygonGroup {

    private String name;
```

```
private List objects;
private MovingTransform3D transform;
/**
    生成新的空 PolygonGroup
*/
  public PolygonGroup(){
      this ("unnamed");
/**
    用指定名称生成新的空 PolygonGroup
*/
public PolygonGroup(String name) {
    setName(name);
    objects = new ArrayList();
    transform = new MovingTransform3D();
}
/**
    取得这个 PolygonGroup 的 MovingTransform3D
*/
public MovingTransform3D getTransform() {
    return transform;
}
/**
    取得这个 PolygonGroup 名称
*/
public String getName() {
    return name;
}
/**
    设置这个 PolygonGroup 名称
*/
public void setName(String name) {
    this.name = name;
}
/**
    将多边形加入这个组
*/
public void addPolygon(Polygon3D o) {
    objects.add(o);
}
```

```
/**
    将 PolygonGroup 加进这个组
*/
public void addPolygonGroup(PolygonGroup p) {
    objects.add(p);
}
/**
    克隆这个多边形组。 Polygon3D 在这个组与克隆组之间共享，并复制
    Transform3D
*/
public Object clone() {
    PolygonGroup group = new PolygonGroup(name);
    group.setFilename(filename);
    for (int i=0; i<objects.size(); i++) {
        Object obj = objects.get(i);
        if (obj instanceof Polygon3D) {
            group.addPolygon((Polygon3D)obj);
        }
        else {
            PolygonGroup grp = (PolygonGroup)obj;
            group.addPolygonGroup((PolygonGroup)grp.clone());
        }
    }
    group.transform = (MovingTransform3D)transform.clone();
    return group;
}
/**
    取得这个组中指定名称的 PolygonGroup ，如果找不到则返回 null
*/
public PolygonGroup getGroup(String name) {
    // 检查这个组
    if (this.name != null && this.name.equals(name)) {
        return this;
    }
    for (int i=0; i<objects.size(); i++) {
        Object obj = objects.get(i);
        if (obj instanceof PolygonGroup) {
            PolygonGroup subgroup =
                ((PolygonGroup)obj).getGroup(name);
```

```
            if (subgroup != null) {
                return subgroup;
            }
        }
    }

    // 找不到组
    return null;
}
/**
    更新这个组和任何子组的 MovingTransform3D
*/
public void update(long elapsedTime) {
    transform.update(elapsedTime);
    for (int i=0; i<objects.size(); i++) {
        Object obj = objects.get(i);
        if (obj instanceof PolygonGroup) {
            PolygonGroup group = (PolygonGroup)obj;
            group.update(elapsedTime);
        }
    }
  }
}
```

PolygonGroup 类跟踪的实体清单是 Polygon3D 或 PolygonGroup，因此任何多边形组都可以有子多边形组。

PolygonGroup 可选有名称，getGroup()方法可以按名称找到 PolygonGroup。例如，如果要寻找子 PolygonGroup 名 turret 以取得其变换，则可以用这个方法。这个方法查找多边形组本身及所有子多边形组，如果找不到指定名称的多边形组，则返回 null。

Update()方法只是更新 PolygonGroup 变换和任何子 PolygonGroup 的变换。

Clone()方法克隆 PolygonGroup。例如，可以生成游戏中的几个不同机器人。这个方法对新 PolygonGroup 生成新 getGroup()，但使用相同的多边形（因为它们是静态的）。

组中所有多边形迭代

绘制 PolygonGroup 时，实际上并不关心多边形属于哪个（子）组，只是画出组中的每个多边形，包括子组中每个多边形。另外，多边形要变换之后才绘制。

下面在 PolygonGroup 类中生成几个新方法，对每个多边形进行迭代。每个多边形负责对其所有多边形进行迭代。如果组中有子组，则每个子组负责对其所有多边形进行迭代。

```
Private int iteratorIndex;
```

```
...
/**
    对这个组复位多边形迭代器
*/
public void resetIterator(){
    iteratorIndex = 0;
      for (int i=0; i<objects.size(); i++) {
          Object obj = objects.get(i);
          if (obj instanceof PolygonGroup) {
           (PolygonGroup)obj).reaetIterator();
           }
      }
}
/**
    检查当前迭代中是否有另一多边形
*/
public boolean hasNext(){
   return (iteratorIndex < object.size());
}
/**
    取得迭代中下一多边形，对其采用 MovingTransform3D,并将其存放在缓存中
*/
public  void nextPolygonTransformed(Polygon3D cache){
    object obj objects.get(iteratorIndex)

    if (obj instanceof PolygonGroup){
        PolygonGroup group = (PolygonGroup)obj;
        group. NextPolygonTransformed(cache);
        if (! group.hasNext()){
            iteratorIndex++
        }
    }
    else{
        iteratorIndex++
        cache.setTo((Polygon3D)obj);
    }
    cache.add(transform);
}
```

在这些方法中，PolygonGroup 的 iteratorIndex 表示从清单中下一个返回的多边形式

子组。

nextPolygonTransformed()方法返回当前 iteratorIndex 处的多边形。如果清单中 iteratorIndex 处的对象是个 PolygonGroup，则调用这个组的 nextPolygonTransformed()方法。

另外，nextPolygonTransformed()还采用组的变换。因此子组采用变换，然后父组采用变换。这样，每个多边形采用整个变换层次。

要画组中所有多边形，可以在 zBufferedRenderer 类中增加另一个 draw()方法：

```
public boolean draw( Graphics2D g, PolygonGroup group){
    boolean visible = false;
    group.resetIterator();
    while (group.hasNext()){
        roup. NextPolygonTransformed(temp);
        visible |=draw(g, temp);
    }
    return visible;
}
```

这样就生成了 Z 缓存、移动变换、多边形组和多边形绘制器，可以用 Z 缓存绘制多边形组了。

下面要从外部文件读取多边形模型。

9.4　从 OBJ 文件装入多边形组

能够绘制更复杂的多边形模型后，否则要生成的所有多边形编写 Java 代码看来太复杂。本节生成分析器，读取 Alias|Wavefront OBJ 文件，这是三维实体常用的文本格式。

用 OBJ 之类常用的格式使三维艺术家可以在现有三维设计软件中生成三维实体，然后放到游戏中。由于采用文本格式，因此无法访问这类三维设计软件的人也可以用文本编辑器修改现有 OBJ 文件或生成新 OBJ 文件。

在本书代码中，我们实际上只支持 OBJ 文件规范的子集，但代码能够顺利读取现有 OBJ 文件。可以从 Web 上搜索“obj file format”，了解完整的 OBJ 文件格式规范。

9.4.1　OBJ 文件格式

OBJ 文件格式是文本格式，一行一个命令。开头为#的行是说明行，空行忽略。分析器支持 5 个关键字：

mtllib <filename>	从外部.mtl 文件装入材料
v <x> <y> <z>	用浮点坐标（x, y, z）定义顶点
f <v1><v2><v3>	定义新面。面（face）是平面凸多边形，按逆时针顺序列出顶点，面可以有多个顶点。对每个顶点，正数表示顶点的索引，在文件中定义。负数表示相对于最后读取顶点定义的顶点。例如，1 表示文件中第一个

	顶点，-1 是最后读取顶点，-2 是最后读取顶点前一个顶点。
g<name>	定义指定名称的新组，后续面加进这个组中。
Usemtl<name>	后续面用指定材料（从外部.mtl 文件装入）。

MTL 格式定义材料，这里是纹理。稍后将介绍 MTL 格式，但这里要先要生成立方体的例子 OBJ 文件，见清单 9.5。

这个 OBJ 文件列出构成立方体的 8 个顶点和 6 个多边形。本章末尾的演示程序中要使用这类实体。

清单 9.5 cube.obj

```
# 装入材料
mtllib textures.mtl

#定义顶点
v 16 32 16
v 16 32 -16
v 16 0 16
v 16 0 -16
v -16 32 16
v -16 32 -16
v -16 0 16
v -16 0 -16
#指定组名
g myCube
#定义材料
usemtl texture_A
#定义多边形
f 1 3 4 2
f 6 8 7 5
f 2 6 5 1
f 3 7 8 4
f 1 5 7 3
f 4 8 6 2
```

下面生成 OBJ 文件分析器。清单 9.6 显示了分析器第一部分 ObjectLoader，它还不实际分析 OBJ 文件，而是建立一个系统，用来管理材料、顶点和多边形组。另外，它提供一个基本分析框架，忽略空行和说明行，将其他行发送到另一个行分析器中。

清单 9.6 ObjectLoader.java

```
package com.brackeen.javagamebook.math3D;
```

```
import java.io.*;
import java.util.*;
import com.brackeen.javagamebook.graphics3D.texture.*;
/**
    ObjectLoader 类装入 Alias|Wavefront OBJ 文件规范的子集
*/
public class ObjectLoader {
    /**
        Material 类包装 ShadedTexture
    */
    public static class Material {
        public File sourceFile;
        public ShadedTexture texture;
    }
    /**
        LineParser 接口分析文本文件中的行。OBJ 与 MTL 文件使用独立 LineParsers
    */
    protected interface LineParser {
        public void parseLine(String line) throws IOException,
            NumberFormatException, NoSuchElementException;
    }

    protected File path;
    protected List vertices;
    protected Material currentMaterial;
    protected HashMap materials;
    protected List lights;
    protected float ambientLightIntensity;
    protected HashMap parsers;
    private PolygonGroup object;
    private PolygonGroup currentGroup;
    /**
        生成新 ObjectLoader
    */
    public ObjectLoader() {
        materials = new HashMap();
        vertices = new ArrayList();
        parsers = new HashMap();
        parsers.put("obj", new ObjLineParser());
```

```
        parsers.put("mtl", new MtlLineParser());
        currentMaterial = null;
        setLights(new ArrayList(), 1);
    }
    /**
        设置分析实体中多边形所用光。调用这个方法后，调用 loadObject 时用这些光
    */
    public void setLights(List lights,
        float ambientLightIntensity)
    {
        this.lights = lights;
        this.ambientLightIntensity = ambientLightIntensity;
    }
    /**
        将 OBJ 文件作为 PolygonGroup 装入
    */
    public PolygonGroup loadObject(String filename)
        throws IOException
    {
        File file = new File(filename);
        object = new PolygonGroup();
        object.setFilename(file.getName());
        path = file.getParentFile();

        vertices.clear();
        currentGroup = object;
        parseFile(filename);

        return object;
    }
    /**
        从文件中的向量清单取得 Vector3D。正数表示顶点的索引，负数表示相对于最后读取顶点定
        义的顶点。1 表示文件中第一个顶点，-1 是最后读取顶点，0 是无效的，抛出异常
    */
    protected Vector3D getVector(String indexStr) {
        int index = Integer.parseInt(indexStr);
        if (index < 0) {
            index = vertices.size() + index + 1;
        }
```

```
        return (Vector3D)vertices.get(index-1);
    }
    /**
        分析 OBJ (扩展名为".obj")或 MTL 文件 (扩展名为".mtl")
    */
    protected void parseFile(String filename)
        throws IOException
    {
        // 取得相对于源路径的文件
        File file = new File(path, filename);
        BufferedReader reader = new BufferedReader(
            new FileReader(file));

        // 根据文件扩展名取得分析器
        LineParser parser = null;
        int extIndex = filename.lastIndexOf('.');
        if (extIndex != -1) {
            String ext = filename.substring(extIndex+1);
            parser = (LineParser)parsers.get(ext.toLowerCase());
        }
        if (parser == null) {
            parser = (LineParser)parsers.get("obj");
        }

        //分析文件每一行
        while (true) {
            String line = reader.readLine();
            // 读完所有行
            if (line == null) {
                reader.close();
                return;
            }

            line = line.trim();

            // 忽略空行和说明行
            if (line.length() > 0 && !line.startsWith("#")) {
                // 解释行
                try {
```

```
                parser.parseLine(line);
            }
            catch (NumberFormatException ex) {
                throw new IOException(ex.getMessage());
            }
            catch (NoSuchElementException ex) {
                throw new IOException(ex.getMessage());
            }
        }
    }
}
```

ObjectLoader 类的内接口 LineParser 提供了分析文件行的方法，它有两个实现类 ObjLineParser 与 MtlLineParser。

ParseFile()方法一次一行地分析 OBJ 或 MTL 文件，忽略空行和说明行。它用相应 LineParser 分析每一行，根据文件扩展名取得分析器（.obj 或.mtl）。

getVector()方法是个方便方法，取得指定索引处的顶点。记住，索引从 1 算起，负数表示相对于最后读取顶点定义的顶点。

最后，loadObject()方法装入 OBJ 文件和返回 PolygonGroup。

当然，这些工作要用到清单 9.7 所示的 ObjLineParser 内类。

清单 9.7　ObjectLoader 类的 ObjLineParser 内类

```
/**
    分析 OBJ 文件中的行
*/
protected class ObjLineParser implements LineParser {

    public void parseLine(String line) throws IOException,
        NumberFormatException, NoSuchElementException
    {
        StringTokenizer tokenizer = new StringTokenizer(line);
        String command = tokenizer.nextToken();
        if (command.equals("v")) {
            // 生成新顶点
            vertices.add(new Vector3D(
                Float.parseFloat(tokenizer.nextToken()),
                Float.parseFloat(tokenizer.nextToken()),
                Float.parseFloat(tokenizer.nextToken())));
        }
        else if (command.equals("f")) {
```

```
// 生成新面（平面凸多边形）
List currVertices = new ArrayList();
while (tokenizer.hasMoreTokens()) {
    String indexStr = tokenizer.nextToken();

    // 忽略纹理和法线坐标
    int endIndex = indexStr.indexOf('/');
    if (endIndex != -1) {
        indexStr = indexStr.substring(0, endIndex);
    }
    currVertices.add(getVector(indexStr));
}

//生成纹理多边形
Vector3D[] array =
    new Vector3D[currVertices.size()];
currVertices.toArray(array);
TexturedPolygon3D poly =
    new TexturedPolygon3D(array);

    //设置纹理
    ShadedSurface.createShadedSurface(
        poly, currentMaterial.texture,
        lights, ambientLightIntensity);

    // 将多边形加进当前组
    currentGroup.addPolygon(poly);
}
else if (command.equals("g")) {
    // 定义当前组
    if (tokenizer.hasMoreTokens()) {
        String name = tokenizer.nextToken();
        currentGroup = new PolygonGroup(name);
    }
    else {
        currentGroup = new PolygonGroup();
    }
    object.addPolygonGroup(currentGroup);
}
```

```
            else if (command.equals("mtllib")) {
                // 从文件装入材料
                String name = tokenizer.nextToken();
                parseFile(name);
            }
            else if (command.equals("usemtl")) {
                // 定义当前材料
                String name = tokenizer.nextToken();
                currentMaterial = (Material)materials.get(name);
                if (currentMaterial == null) {
                    System.out.println("no material: " + name);
                }
            }
            else {
                // 忽略未知命令
            }
        }
    }
```

ObjLineParser 类用 StringTokenizer 从分析的行中取得分隔词。行的分析很简单，v 命令生成新顶点，f 命令生成新多边形（用 ShadedSurface），g 命令生成新 PolygonGroup，usemtl 命令设置当前材料，mtllib 命令实际触发另一 parseFile()调用，装入 MTL 文件。

值得注意的是，f 命令对每个顶点忽略“/”号及其后面的字符，因为 OBJ 文件规范用斜杠后面的顶点定义多边形法线和纹理光标。这里不用这些顶点，因此我们忽略“/”号及其后面的字符。选择就可以读取包括这个特性的 OBJ 文件。

OBJ 文件阅读器还要一个特性：生成 MTL 文件阅读器。

9.4.2 MTL 文件格式

MTL 文件格式定义材料，如实体颜色、反射表面（如闪光物体或镜面）、鼓包纹理（模拟材料深度）和纹理。它有各种选项与命令，这里只用纹理贴图。我们使用的 MTL 文件命令如下：

newmtl <name>　　定义指定名称的新材料

map_Kd<filename>　　对材料指定纹理图

这里没有问题，生成的 MtlLineParser 读取纹理和生成新的 Material 对象，如清单 9.8 所示。

清单 9.8　ObjectLoader 的 MtlLineParser 内类

```
/**
    分析材料 MTL 文件中的行
*/
```

```
protected class MtlLineParser implements LineParser {

    public void parseLine(String line)
        throws NoSuchElementException
    {
        StringTokenizer tokenizer = new StringTokenizer(line);
        String command = tokenizer.nextToken();

        if (command.equals("newmtl")) {
            // 必要时生成新材料
            String name = tokenizer.nextToken();
            currentMaterial = (Material)materials.get(name);
            if (currentMaterial == null) {
                currentMaterial = new Material();
                materials.put(name, currentMaterial);
            }
        }
        else if (command.equals("map_Kd")) {
            //对材料指定纹理图
            String name = tokenizer.nextToken();
            File file = new File(path, name);
            if (!file.equals(currentMaterial.sourceFile)) {
                currentMaterial.sourceFile = file;
                currentMaterial.texture = (ShadedTexture)
                    Texture.createTexture(file.getPath(), true);
            }
        }
        else {
            //忽略未知命令
        }
    }
  }
}
```

这样就装入了OBJ文件，生成了读取OBJ格式子集的OBJ装入器，这是本书三维实体所需要的。如果要扩展OBJ装入器使其能读取更多OBJ命令，则可以到Web上搜索“OBJ file format”，取得更多这方面的资料。

9.5 游戏实体

本节要建立 PolygonGroup 的容器 GameObject，是本书游戏与演示程序中所有对象的基础类。游戏实体可以操纵自己的状态，包括多边形组的变换。实际上，GameObject 本身什么也不干，要扩展之后才能完成所要功能。

本章生成的 GameObject 类只是个开头，后面几章介绍碰撞探测、路径寻找与人工智能时，要在其中增加更多特性。

首先看看游戏实体的基本状态。游戏实体（game object）可以是任何对象，如静态对象、抛射弹、充电器和敌人。在游戏过程中，游戏实体可以闭置（不动）或活动（运动）。有些实体可能破坏，如敌人爆炸时或抛射弹撞上墙壁时。这样，游戏实体有 3 种状态：IDLE、ACTIVE 与 DESTROYED，如图 9-8 所示。

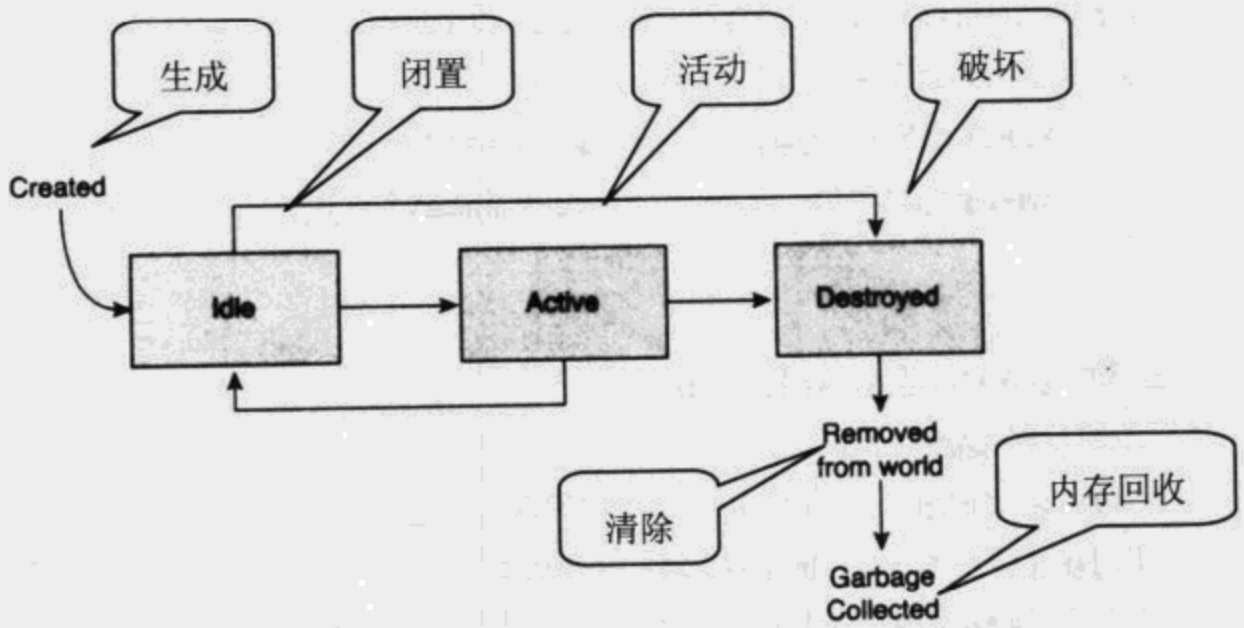


图 9-8 游戏实体生命周期

游戏实体首次生成时，处于 IDLE（闲置）状态，可以随时转入 ACTIVE（活动）状态。同样，ACTIVE 状态也可以随时转入 IDLE（活动）状态。

IDLE 或 ACTIVE 状态的游戏实体可以转入 DESTROYED 状态。在 DESTROYED 状态中，游戏实体不再返回任何其他状态。游戏管理器可以清除 DESTROYED 状态的游戏实体，然后内存回收单元要回收这个游戏实体（假设游戏实体已经没有其他引用）。

不同类型游戏实体以不同方式遵循这个游戏实体生命周期。例如，静态对象在整个游戏中处于 IDLE 状态，抛射弹可能立即转入 ACTIVE 状态，直到撞上某个物体，然后变成 DESTROYED 状态。敌人开始时为 IDLE 状态，显示时则转为 ACTIVE 状态。例如，如果敌人离开屏幕一段时间，则可能再次回到 IDLE 状态。最后，游戏者向它射击足够久后，敌人进入 DESTROYED 状态。

记住这些内容后，下面生成 GameObject 类（见清单 9.9）。这个类管理游戏实体状态并到游戏实体的 PolygonGroup 与变换提供几个方便方法。

清单 9.9 GameObject.java

```
package com.brackeen.javagamebook.game;
```

```
import com.brackeen.javagamebook.math3D.*;
/**
    GameObject 类是游戏中 PolygonGroup 表示的各类对象的基类。例如，GameObject 有静态对象、
    抛射弹、充电器和敌人等对象。GameObject 有 3 种状态：STATE_IDLE, STATE_ACTIVE 与
    STATE_DESTROYED
*/
  public class GameObject {
  protected static final int STATE_IDLE = 0;
  protected static final int STATE_ACTIVE = 1;
  protected static final int STATE_DESTROYED = 2;

  private PolygonGroup PolygonGroup;
  private int state;
  /**
      生成指定 PolygonGroup 表示的新 GameObject ，PolygonGroup 可以是 null
  */
  public GameObject(PolygonGroup polygonGroup) {
      this.polygonGroup = polygonGroup;
      state = STATE_IDLE;
  }
  /**
      从 Transform3D 取得这个 GameObject 的地址的快捷方法
  */
  public Vector3D getLocation() {
      return polygonGroup.getTransform().getLocation();
  }
  /**
      取得这个实体的变换
  */
  public MovingTransform3D getTransform() {
      return polygonGroup.getTransform();
  }
  /**
      取得这个实体的 PolygonGroup
  */
  public PolygonGroup getPolygonGroup() {
      return polygonGroup;
  }
  /**
```

```
    设置这个实体的状态，有三种状态 STATE_IDLE, STATE_ACTIVE 与 STATE_DESTROYED
*/
protected void setState(int state) {
    this.state = state;
}
/**
    设置指定实体的状态。任何 GameObject 可以设置任何其他 GameObject 的状态。有三种状态
    STATE_IDLE, STATE_ACTIVE 与 STATE_DESTROYED
*/
protected void setState(GameObject object, int state) {
    object.setState(state);
}
/**
    这个 GameObject 闲置时返回真值
*/
public boolean isIdle() {
    return (state == STATE_IDLE);
}
/**
    这个 GameObject 活动时返回真值
*/
public boolean isActive() {
    return (state == STATE_ACTIVE);
}
/**
    这个 GameObject 破坏时返回真值
*/
public boolean isDestroyed() {
    return (state == STATE_DESTROYED);
}
/**
    如果 GameObject 处于活动状态，则这个方法更新其 PolygonGroup ，否则这个方法什么也
    不干
*/
public void update(GameObject player, long elapsedTime) {
    if (isActive()) {
        polygonGroup.update(elapsedTime);
    }
}
```

```
    /**
        告知这个 GameObject 上次更新时是否显示。缺省情况下，如果 GameObject 闲置并告知显
        示，则转入活动状态
    */
    public void notifyVisible(boolean visible) {
        if (visible && isIdle()) {
            state = STATE_ACTIVE;
        }
    }
}
```

在这个缺省游戏实体中，游戏实体的多边形组只在 ACTIVE 状态时才更新。这个方法的一个参数是游戏者的游戏实体引用，因此任何游戏实体都知道游戏者的位置。显然，敌人也想知道游戏者的位置。

另外，notifyVisible()方法告知这个 GameObject 上次更新时是否显示。如果 GameObject 闲置并告知显示，则转入活动状态。

游戏实体生命周期的规则在这个类中没有严格执行：理论上可以将处于 DESTROYED 状态的游戏实体改为 ACTIVE 状态，但如果游戏实体已经清除，则不合理。

9.6　管理游戏实体

为了跟踪世界上的所有实体，我们生成清单 9.10 所示的 GameObjectManager 接口。本章使用这个接口的简单实现，后面几章要生成新的实现。

GameObjectManager 接口的方法跟踪世界上所有实体，另外两个方法 markVisible()与 markAllVisible()，标识哪个实体要显示和绘制。markVisible()方法取参数 Rectangle，指定显示的二维区域（鸟瞰视图）。只有显示实体才要绘制，绘制之后，所有实体再次标为不显示。

清单 9.10　GameObjectManager.java

```
package com.brackeen.javagamebook.game;

import java.awt.Rectangle;
import java.awt.Graphics2D;
import com.brackeen.javagamebook.game.GameObjectRenderer;
/**
    GameObjectManager 接口的方法跟踪世界上所有实体并绘制游戏实体
*/

public interface GameObjectManager {
    /**
```

```
    将指定二维边界内的所有实体标为可能显示（要绘制）
*/
public void markVisible(Rectangle bounds);
/**
    将所有实体标为可能显示（要绘制）
*/
public void markAllVisible();
/**
    将 GameObject 加进这个管理器
*/
public void add(GameObject object);
/**
    将 GameObject 加进这个管理器，指定为游戏者实体。现有游戏者实体（如有）不清除
*/
public void addPlayer(GameObject player);
/**
    取得指定为游戏者实体的实体，如果没有指定游戏者实体，则返回 null
*/
public GameObject getPlayer();
/**
    从这个管理器中清除一个 GameObject
*/
public void remove(GameObject object);
/**
    根据上次更新以来经过的时间量更新所有实体
*/
public void update(long elapsedTime);
/**
    画出所有显示实体
*/
public void draw(Graphics2D g, GameObjectRenderer r);
}
```

GameObjectManager 中的 draw()方法绘制游戏实体管理器管理的所有显示实体并将这些实体再次标为不显示。它用 GameObjectRenderer 绘制实体，这只是绘制实体的接口，见清单 9.11。ZbufferedRenderer 类实现这个接口。

清单 9.11　GameObjectRenderer.java

```
package com.brackeen.javagamebook.game;
```

```
import java.awt.Graphics2D;
import com.brackeen.javagamebook.game.GameObject;
/**
    GameObjectRenderer 接口提供绘制 GameObject 的方法
*/
public interface GameObjectRenderer {
    /**
        绘制实体，并在实体有显示部分时返回真值
    */
    public boolean draw(Graphics2D g, GameObject object);

}
```

SimpleGameObjectManager 实现 GameObjectManager 接口，在清单中保持所有实体。实体标为可视时，放在“显示”表中，绘制所有显示实体之后，从显示表中清除。第 11 章后面要生成更高级的 GameObjectManager，但目前这个简单游戏实体管理器就够用了。

9.7　综合起来

本章介绍了许多内容，现在已经快到结尾了，因此，使用所有这些特性生成一个演示程序。

这个演示程序遍历三种不同类型的游戏实体：静态框、抛射器和 Bot。每个实体用自己的 OBJ 文件定义三维模型。

静态框只是个 GameObject，没有什么特别功能，只是摆着好看的。

清单 9.12 显示了 Bot 对象，如图 9-6 所示，包括转台和基座。转台独立于基座移动，并且随时面向游戏者。

清单 9.12　Bot.java

```
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.game.GameObject;
/**
    Bot 游戏实体是个小的静态 Bot，有转台，并且随时面向游戏者
*/
public class Bot extends GameObject {

    private static final float TURN_SPEED = .0005f;
    private static final long DECISION_TIME = 2000;

    protected MovingTransform3D mainTransform;
    protected MovingTransform3D turretTransform;
```

```
    protected long timeUntilDecision;
    protected Vector3D lastPlayerLocation;

    public Bot(PolygonGroup polygonGroup) {
        super(polygonGroup);
        mainTransform = polygonGroup.getTransform();
        PolygonGroup turret = polygonGroup.getGroup("turret");
        if (turret != null) {
            turretTransform = turret.getTransform();
        }
        else {
            System.out.println("No turret defined!");
        }
        lastPlayerLocation = new Vector3D();
    }

    public void notifyVisible(boolean visible) {
        if (!isDestroyed()) {
            if (visible) {
                setState(STATE_ACTIVE);
            }
            else {
                setState(STATE_IDLE);
            }
        }
    }

    public void update(GameObject player, long elapsedTime) {
        if (turretTransform == null || isIdle()) {
            return;
        }

        Vector3D playerLocation = player.getLocation();
        if (playerLocation.equals(lastPlayerLocation)) {
            timeUntilDecision = DECISION_TIME;
        }
        else {
            timeUntilDecision-=elapsedTime;
            if (timeUntilDecision <= 0 ||
```

```
                !turretTransform.isTurningY())
            {
                float x = player.getX() - getX();
                float z = player.getZ() - getZ();
                turretTransform.turnYTo(x, z,
                    -mainTransform.getAngleY(), TURN_SPEED);
                lastPlayerLocation.setTo(playerLocation);
                timeUntilDecision = DECISION_TIME;
            }
        }
        super.update(player, elapsedTime);
    }
}
```

在 Bot 对象中，不能在每一帧检查游戏者位置，只在每两秒检查一次（这是 DECISION_TIME 中定义的时间量）。这样，Bot 只要每 4 秒计算所要的操作，我们让 Bot 迟钝一些，比游戏者延迟一会儿。

下面是 Blast 游戏实体，见清单 9-13。Blast 实体是个抛射器，游戏者可以发射，它沿直线发射（同时转动），在五秒钟后爆炸。SPEED 与 ROT_SPEED 定义其移动与转动速度。

清单 9.13 Blast.java

```
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.game.GameObject;
/**
    Blast 实体是个抛射器游戏者可以发射，它沿直线发射，在五秒钟后爆炸，爆炸时立即破坏 Bot
*/
public class Blast extends GameObject {

    private static final long DIE_TIME = 5000;
    private static final float SPEED = 1.5f;
    private static final float ROT_SPEED = .008f;

    private long aliveTime;
    /**
        用指定多边形组和矢量方向生成新 Blast
    */
    public Blast(PolygonGroup polygonGroup, Vector3D direction) {
        super(polygonGroup);
        MovingTransform3D transform = polygonGroup.getTransform();
        Vector3D velocity = transform.getVelocity();
```

```
        velocity.setTo(direction);
        velocity.multiply(SPEED);
        transform.setVelocity(velocity);
        //transform.setAngleVelocityX(ROT_SPEED);
        transform.setAngleVelocityY(ROT_SPEED);
        transform.setAngleVelocityZ(ROT_SPEED);
        setState(STATE_ACTIVE);
    }

    public void update(GameObject player, long elapsedTime) {
        aliveTime+=elapsedTime;
        if (aliveTime >= DIE_TIME) {
            setState(STATE_DESTROYED);
        }
        else {
            super.update(player, elapsedTime);
        }
    }
}
```

得到所有游戏实体之后，就可以生成 GameObjectTest。和平常一样，这个类扩展 GameCore3D 类，有两个主要方法。首先，用 createPolygon()方法装入 OBJ 文件并建立游戏实体：

```
    public void createPolygon(){
      //生成地面
     Texture floorTexture = Texture. CreateTexture(
          ( "../images/roof1.png" ,true)
      ((ShadedTexture) floorTexture).setDefaultShadedTexture(
            ShadedTexture.MAX_LEVEL*3/4)
     Rectangle3D floorTextureBounds = new Rectangle3D
            new Vector3D(0,0,0),
            new Vector3D(1,0,0),
            new Vector3D(0,0,1),
            floorTexture.getWidth(),
            floorTexture.getHeight());
    float s = GAME_AREA_SIZE;
    floor = new TexturePolygon3D(new Vector3D[]{
            new Vector3D(-s,0,s),
            new Vector3D(s,0,s),
            new Vector3D(s,0, -s),
```

```
            new Vector3D(-s,0, -s)
floor.setTexture(setTexture, floorTextureBounds);

//建立模型的局部灯光
float ambientLightIntensity = .5f
List lights = new LinkedList();
Lights.add(new PointLight3D(-100,100,100,.5f,-1));
Lights.add(new PointLight3D(-100,100,0,.5f,-1));

//装入实体模型
objectLoader loader = new objectLoader();
loader.setLights (lights,ambientLightIntensity);
try{
    robotModel = loader.loadobject("../images/robot.obj");
    powerUpModel = loader.loadobject("../images/cube.obj");
    blastModel = loader.loadobject("../images/blast.obj");
}
catch (IOException ex){
    ex.printStackTrace();
}
  //生成游戏实体
  gameObjectManager = new SimpleGameObjectManager();
  gameObjectManager.addPlayer(new GameObject(
          new PlaygonGroup "(Playerv)))" ;
  gameObjectManager.getPlayer().getLocation().y = 5;
  for (int i=0,i<NUM_POWER_UPS;i++){
      GameObject object =
          new GameObject ((PlaygonGroup) powerUpModel.clone());
      placeObject (object);
  }
}
  //在游戏区随机放置实体
  public void placeObject (GameObject object)
       float size = CAME_AREA_SIZE;
       object. getLocation().setTo(
           (float) (Math.random() * size - size/2);
            0,
           (float) (Math.random() * size - size/2));
       gameObjectManager.add(object);
```

```
}
```

GameObjectTest 中另一个重要方法是 updateWorld()方法，是从 GameCore3D 中修改而成的，使用变换得到游戏者运动。它增加了发射抛射器的功能：

```
Public void updateWorld (long elapsedTime){

    Float angleVelocity;
    // cap elapsedTime
    elapsedTime = Math.min (elapsedTime, 100);

    GameObject player = gameObjectManager.getPlayer();
    MovingTransform3D playerTransform =player.getTransform();
    Vcetor3D velocity = playerTransform.getVelocity();

    playerTransform.stop();
    float x = -playerTransform.getSinAngleY();
    float z = -playerTransform.getCosAngleY();
    if (goForward.isPressed()){
        velocity.add(x,0,z);
    }
    if (goBackward.isPressed(){
        velocity.add(-x,0,-z);
    }
    if (goLeft.isPressed(){
        velocity.add(z,0,-x);
    }
    if (goRight.isPressed(){
        velocity.add(-z, 0, x);
    }
    if (fire.isPressed(){
        float cosX = playerTransform.getCosAngleX();
        float sinX = playerTransform.getSinAngleX();
        Blast blast = new Blast(
            (PolygonGroup) blastModel.clone(),
            new Vector3D (cosX*x, sinX, cosX*Z));
       //发射开始位置要工作。像发射时子弹从你的前面发射出去一样
        Blast.getLocation().setTo(
            Player.getX(),
            Player.getY() + BULLET_HEIGHT,
            Player.getZ());
```

```
        gameObjectManager.add(blast);
    }
    velocity.multiply(PLAYER_SPEED);
    playerTransform.setVelocity(velocity);
    //向上向下看（绕x轴转）
    angleVelocity = Math.min (tiltUp.getAmount(),200);
    angleVelocity + = Math.min (-tiltDown.getAmount(),-200);
    playerTransform.setAngleVelocityX(angleVelocity *
            PLAYER_TURN_SPEED /200);
    //向上向下看（绕y轴转）
    angleVelocity = Math.min (turnLeft.getAmount(),200);
    angleVelocity + = Math.min (-turnLeft.getAmount(),-200);
    playerTransform.setAngleVelocityX(angleVelocity *
            PLAYER_TURN_SPEED /200);
//先把这个帧的整个世界标为显示
gameObjectManager.markAllVisible();
//更新实体
gameObjectManager.update(elapsedTime);

//限制向上向下看
    float angleX = playerTransform.getAngleX();
    float limit = float Math.PI/2;
    if (angleX < -limit){
        playerTransform.setAngleX(-limit);
    }
    else if (angleX < -limit){
        playerTransform.setAngleX(limit);
    }

    //将镜头设置为游戏者上面100个单位
    Transform3D camera = polygonRenderer. ();getcamera();
    camera.setTo(playerTransform);
    camera.getLocation().add(0,100,0)
}
```

我还在GameCore3D类中增加了一个精彩的效果，让屏幕顶部的指令在几秒之后消失。上述演示程序中，文本Use the mouse/keys to move.Press Esc to exit总是在屏幕上方显示，但这个演示程序中，文字在几秒之后逐渐消失，这是用半透明颜色实现的：

```
    //文本在500毫秒之后消失
    long fade = INSTRUCTIONS_TIME - drawInstructionsTime;
```

```
if (fade <500){
    fade = fade * 255/500;
    g.setColor(new Color(0xffffff | ((int) fade <<24),true));
}
else{
     g.setColor(Color.WHITE);
}
```

这样就完成了 GameObjectTest，图 9-9 显示了其屏幕图形。

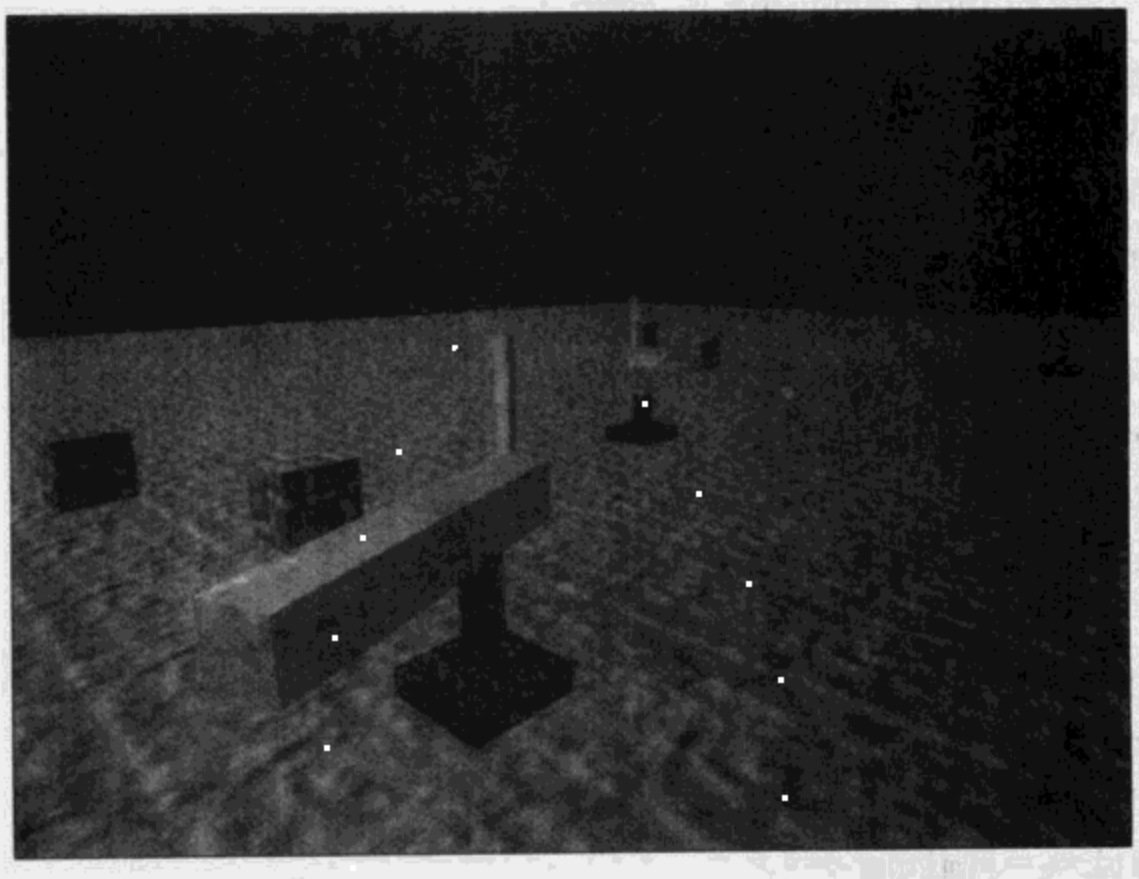


图 9-9　GameObjectTest 屏幕图形显示按正确 Z 顺序绘制的多边形

这个演示程序中要注意的一点是，我们很容易忘记用的是 Z 缓存。如果想看看不用 Z 缓存的样子，可以将 Zbuffer 的 checkDepth()方法修改成总是返回真值，这样就会使绘制的多边形顺序搞乱，看上去奇形怪状。

在 GameObjectTest 演示程序中，如果直接向下看，使绿色地面盖住整个屏幕，则可以发现，帧速率比前几章演示程序的慢得多。这是因为屏幕上每个像素都要检查 Z 缓存和设置 Z 缓存，需要大量开销。前面曾介绍过，Z 缓存适合游戏实体，因为它们一次只在屏幕中的一小部分显示。下一章介绍 BSP 树时要介绍更好的算法，更快绘制世界中的静态多边形和把三维实体合并到视景中。

9.8　改进

下面是扩展本章代码的几个思路：

- 在 MovingTransform3D 中增加加速度，使实体能够光顺地加速，直到达到所要速度，然后保持这个速度。这样就使游戏者的运动更加逼真。
- 在 MovingTransform3D 中增加圆周运动，使实体可以绕另一实体旋转。

- 让使用 MovingTransform3D 的 GameObject 绕一角光顺地转动。游戏者可以用键盘和鼠标光滑地转动，但游戏实体则不然。除了向前、停止、转动、再向前之外，最好能让游戏实体边走边转动。可能要减慢游戏实体速度以限制转角（就像开车时要减速转弯）。
- 扩展 OBJ 与 MTL 支持，MIL 定义的实体颜色很容易实现，只在生成表示一个颜色的 Texture 子类即可。
- 增加更多奇思妙想。游戏实体可以进行多种处理，如让坏蛋转圈，让移动的机器人抬腿，或让鬼怪只在不可见时才追游戏者。第 11 章介绍碰撞探测时，可以对游戏实体做更多工作。

9.9　小结

本书还介绍另一种表面消隐算法：BSP 树。Z 缓存很适合三维移动实体，但用 Z 缓存绘制屏幕上每个像素可能很慢。下章要用 BSP 树解决这个问题。

本章介绍了许多概念，介绍了一些消隐算法，并用 1/Z 深度来实现 Z 缓存。当然，我们生成了用 Z 缓存绘制多边形的绘制器，还生成了一个系统，对多边形组进行三维实体动画，生成了一个分析器，从 OBJ 文件装入三维模型。最后，我们生成了游戏实体的基类，生成几个样本游戏实体，并放进演示程序中。掌握这些概念之后，就可以得到精彩的结果了。后面几章要进一步扩展这些概念。

第 10 章　用 BSP 树管理三维视景

关键主题

- BSP 树简介
- 二叉树基础
- 一维 BSP 树
- 二维 BSP 树
- 实现二维 BSP 树
- 从前到后画多边形
- 第一个 BSP 树例子
- 在视景中画实体
- 从文中装入贴图
- 综合起来
- 改进
- 小结

在现实世界（而不是游戏世界）中，人们投入大量精力建造抗震建筑。

当然，有些地方没有必要建造抗震建筑，例如美国的堪萨斯州。但是，在洛杉矶和日本东京等地，则必须建造抗震建筑。设计建筑时一定要考虑周围的环境。

这就引出了本章的主题。在三维绘图中，我们要根据游戏环境设计多边形管理方法，也称为视景管理。如果视景中的多边形是静态的，则可以用一些方法来加速绘制。

本章使用的方法将多边形组织成一种数据结构，称为二叉空间分区树（binary space partitioned tree）或 BSP 树。BSP 树可以存储多边形，以便迅速绘制视景，后面还可以帮助处理碰撞探测和路径寻找。

10.1　BSP 树简介

上一章“三维对象”生成 Z 缓存，可以在视景中正确绘制多边形，生成完美的绘制视景，先绘制离镜头近的多边形，后绘制离镜头远的多边形。这个方法的缺点是要检查和设置每个像素的 Z 缓存：处理屏幕上每个像素需要太大的开销，而且经常会涉及重画。多边形越多，Z 缓存检查与重画的工作量越大。显然，这个方案的伸缩性不好。

我们要用一个方法代替 Z 缓存，达到下列目标：

- 管理大量多边形；
- 迅速决定视景中任何位置显示哪些多边形；
- 只绘制显示的多边形（而不绘制不显示的多边形）；
- 每个像素只画一次（不重画。）

初看起来，这些目标好像很难达到。当然，达到这些目标的方法取决于游戏类型。例

如，开车游戏要花大量时间看开阔空间，而室内游戏要花大量时间看天花板、地面和墙壁，两者可能采用不同方法。

可以用不同方法组织多边形，使其更容易确定要显示的多边形。除了BSP树之外，另外一些多边形组织方法是八叉树（octrees）和视景图（scene graphs）。BSP树可以在视景中任何位置按从前到后的顺序绘制所有多边形。

通过扩展，BSP树可以达到上述所有目标，但BSP树并不完美，有下列缺点：

- 世界应是静态的（这里墙不能动），但三维游戏实体可以动，因此门和妖怪之类可以使用三维游戏实体。
- 建立BSP树的计算量很大，但理想方案是先建立树，然后在启动时装入树。

Doom之类游戏最初实现BSP树。Doom使用的二维BSP树具有变高度地面和天花板，绘制引擎限制游戏者在游戏中不能向上向下看。在本章建立的代码中，我们也用具有变高度地面和天花板的二维BSP树，但使用前几章的三维绘制引擎，使游戏中能向上向下看。

尽管本章演示二维BSP树，但许多现代游戏中使用的三维BSP树也不难掌握、二维BSP树的例子与方程更容易理解，更容易描述碰撞探测和路径寻找等问题。

使用二维BSP树的缺点是世界中只能有垂直的墙和水平的地面与天花板。但是，在这种限制下，仍然可以建立一些精彩的世界。

10.2 二叉树基础

BSP树是个二叉树，下面先介绍二叉树的一些基本术语和概念。

二叉树数据结构包含节点的层次式结构，二叉树具有下列属性：

- 每个节点至多两个子节点，分别称为左子节点和右子节点；
- 节点有相关联的数据；
- 没有子节点的节点称为叶节点；
- 没有父节点的节点称为根节点；
- 排序二叉树也称为二叉搜索树，节点数据已经排序。

图10-1是个排序二叉树的例子，这个排序二叉树包含一列数：1，3，5，6，8与9。下面要介绍排序二叉树。对每个节点，数值小于该节点值的节点放在该节点左边，数值大于该节点值的节点放在该节点右边。例如，由于根节点值为5，因此，数值小于5的节点放在该节点左边，数值大于该节点值的节点放在该节点右边。认真看看图10-1的例子。

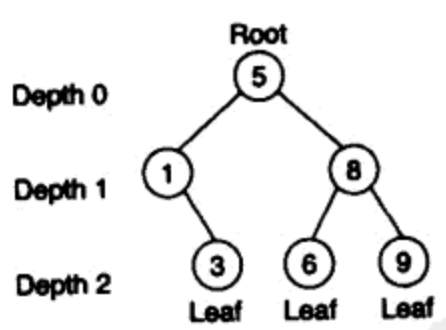


图10-1 这个排序二叉树有3个叶节点，深度为2

从图10-1可以看出，这个树的深度为2。深度也称为高度，是根节点的子节点代数。例子中是平衡的，即每个节点的左边与右边深度至多相差一层。平衡排序二叉树很容易搜

索，而不必查找每个数据。

下面要实现一个简单二叉树，后面要用相同概念建立 BSP 树。我们用整数作为二叉树的数据，像例子中一样。下面先建立节点：

```
public class Node {
    Node left;
    Node right;
    int value;

    public Node(int value) {
        this.value = value;
    }
}
```

Node 类有个整数值和左右子节点的引用（最初为 null，即没有子节点）。记住，节点可能是根节点、叶节点或普通节点。生成新 Node 很简单，这里生成根节点：

```
Node root = new Node(5);
```

然后编写 insertSorted()方法，在排序二叉树中插入新值：

```
public void insertSorted(Node node, int value) {
    if (value < node.value) {
        if (node.left != null){
            insertSorted(node.left, value);
        }
        else {
            node.left = new Node(value);
        }
    }
    else if (value > node.value) {
        if (node.right != null) {
            insertSorted(node.right, value);
        }
        else {
            node.right = new Node(value);
    }
}
else {
        //什么都不干，树中已经有值
    }
}
```

要在排序二叉树中插入数值 8，调用这个方法如下：

```
insertSorted(root, 8);
```

这个方法用递归（recursion，自己调用自己）在树中插值。子节点本身也是个二叉数，通常称为子树（subtree）,因此调用 insertSorted(node.left, value)就是在节点的左子树中插值。

注意：这个方法不允许树中重复，如果值已经在树中，则不再插入。

最后，要用一个方法按树中顺序打印每个值，称为顺序遍历，因为数值按顺序遍历。

```
Public void printInOrder(Node node) {
    if (node != null) {
        printInOrder(node.left);
        System.out.println(node.value);
        printInOrder(node.right);
    }
}
```

要按树中顺序打印每个值，调用如下：

```
printInOrder(root);
```

这个方法也使用递归，表示对这个节点打印左子树，然后打印节点值，然后打印右子树。

这样就介绍完了二叉树，下面要转入本章主题，介绍 BSP 树的概念。

10.3　一维 BSP 树

本章最终要介绍二维 BSP 树，但一开始要了解二维和三维 BSP 树是比较困难的，因此我们先介绍一维 BSP 树。这个例子比较简单，但可以了解 BSP 树的工作方法。

简而言之，BSP 树的节点把世界分成两半，换句话说是二叉空间划分。利用 BSP 树可以从镜头位置由前向后排序多边形。

假设游戏者站在一个只有一排房子的简单世界中，如图 10-2 所示。注意房子按编号排序，且与图 10-1 的例子具有相同编号。

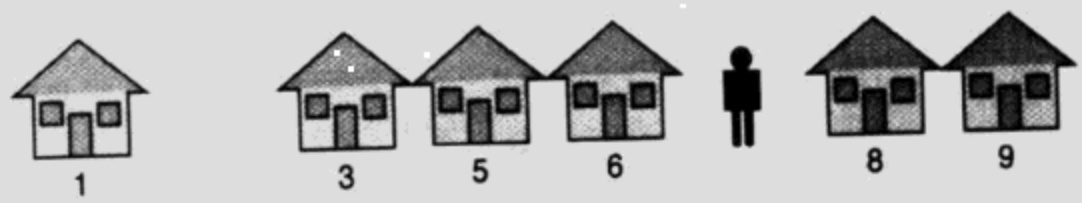


图 10-2　游戏者站在一排房子的 7 号位置

现在假设每个房子是个节点分区。在这个一维例子中，每个房子将世界分成两个空间：左空间（房子左边的一切）和右空间（房子右边的一切）。

建立树的过程和上节建立二叉树的过程一样，本例得到的二叉树如图 10-1 所示。

假设要在三维世界中画这些房子，要从镜头位置由前向后画这些房子，消除重画和避免 Z 缓存的缺点。另外，本例中要先画离游戏者近的房子，后画离游戏者远的房子。

在图 10-2 中，从镜头位置由前向后画这些房子的顺序可以归为两个简单规则：

- 按逆序画镜头左边的房子(6,5,3,1)。
- 按顺序画镜头右边的房子(8，9)。

对于三维透视，左边的房子不会与右边的房子重叠。因此，画房子时先左后右(6,5,3,1,8,9)

还是左右混合(6,8,5,9,3,1)都没事，只要每一边按顺序画即可。因此，不管游戏者面朝哪里，这个方法都是可行的。

要实现这两个简单规则，可以从上节生成的 printInOrder()方法变形，生成 printFrontToBack()方法：

```
public void printFrontToBack(Node node, int camera) {
    if (node = = nu11) return;
    if (camera < node.value) {
        // 按顺序打印
        printFrontToBack (node.left, camera);
        System.out .println（node.value）;
    PrintFrontToBack (node.right, camera);
    }
    else if (camera > node.value )
        //按逆序打印
        printFrontToBack (node.right, camera);
        System.out.println(node.value);
        printFrontToBack(node.left, camera);
    }
else {
    //  顺序不重要
        printFrontToBack(node.left, camera);
        printFrontToBack(node.right, camera);
    }
}
```

这个方法与 printInOrder()方法相似，只是在节点值小于镜头值时按逆序打印。下面用图 10-3 遍历这个方法。

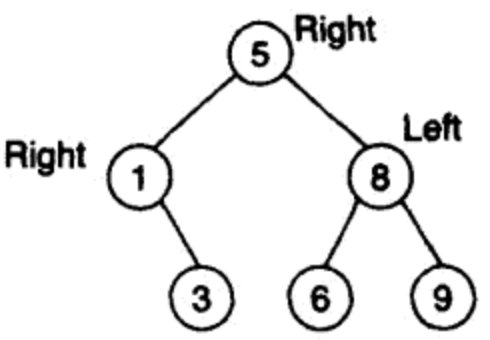


图 10-3　从位置 7 遍历树

对图 10-3 的二叉树，从镜头位置 7 调用 printFrontToBack()方法的步骤如下：

(1) 在根节点，7 大于 5，因此遍历右节点。

(2) 节点 8 中，7 小于 8，因此遍历左节点。

(3) 节点 6 是个叶节点，因此打印 6 并返回父节点。

(4) 打印 8 并遍历右节点。

(5) 节点 9 是个叶节点，因此打印 9 并返回父节点。

(6) 这样就遍历了节点 8，因此返回父节点。

(7) 打印 5 并遍历左节点。

(8) 在节点 1,7 大于 1，因此遍历右节点。

(9) 节点 3 是个叶节点，因此打印 3 并返回父节点。

(10) 节点 1 没有左子节点，因此打印 1 并返回。

(11) 根节点遍历完毕。

因此，从镜头位置 7 调用 printFrontToBack()方法打印顺序为(6,8,9,5,3,1)，左边的一切按从前到后顺序打印(6,5,3,1)，右边的一切按从前到后顺序打印（8,9），正是我们所要的结果。

问题解决了，下面要将这个思想扩展到二维。

10.4　二维 BSP 树

首先要建立和遍历一个二维 BSP 树，然后要生成其实现。

图 10-4 是个二维平面图，我们根据这个二维平面图建立二维 BSP 树。

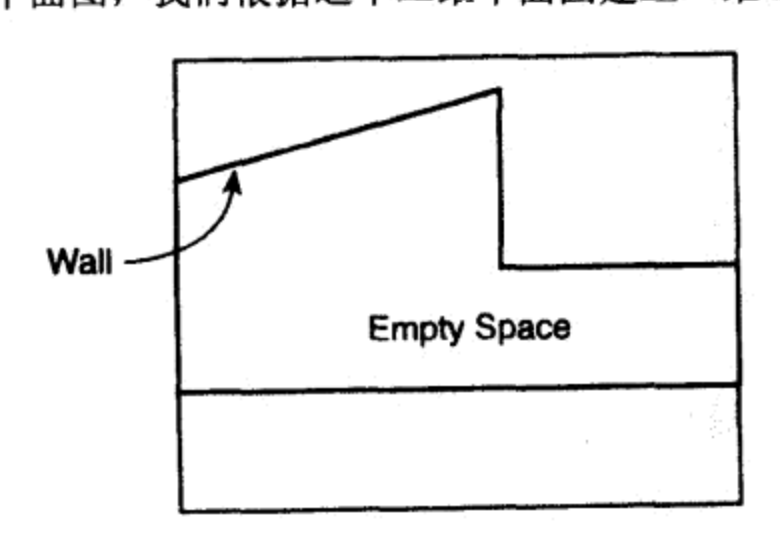


图 10-4　二维平面图的简单房子与四面墙壁

在一维例子中，每个房子是一个分区，因此分区是点，而在二维中，分区是线，在三维中，分区是面。

直线很容易把二维区域分区成两半，但要实际建立 BSP 树，就要知道左边有什么，右边有什么。

实际上，这里左右的概念并不准确，它们适合一维世界，而二维世界中直线的方向是任意的。垂线分出左右，而水平线分出南北。

因此，我们用前后的概念。线段和多边形一样有前有后，法线指向前面，如图 10-5 所示。前面是显示面，即可以用墙进行分区。

由于分区有前有后，因此可以表示镜头相对于每条线的位置：镜头或者在分区前面，或者在分区后面。

掌握这个知识之后，下面通过一个实例建立 BSP 树。如果还不理解直线如何分隔空间，可以从这个例子体会。

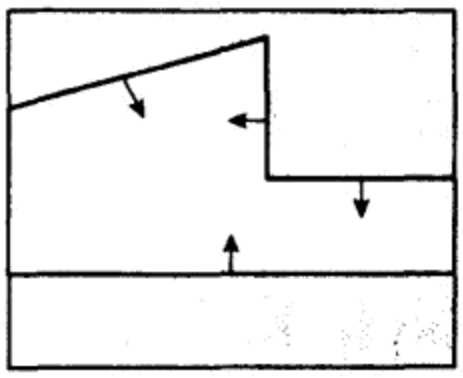


图 10-5 表示墙的线段有前有后

10.4.1 建立 BSP 树

作为例子，我们建立图 10-5 的平面图，每次加一堵墙，增加的顺序随意。

图 10-6 将第一个分区（墙 A）加进树中。注意墙 A 构成的分区扩展到墙 A 以外，见虚线位置。

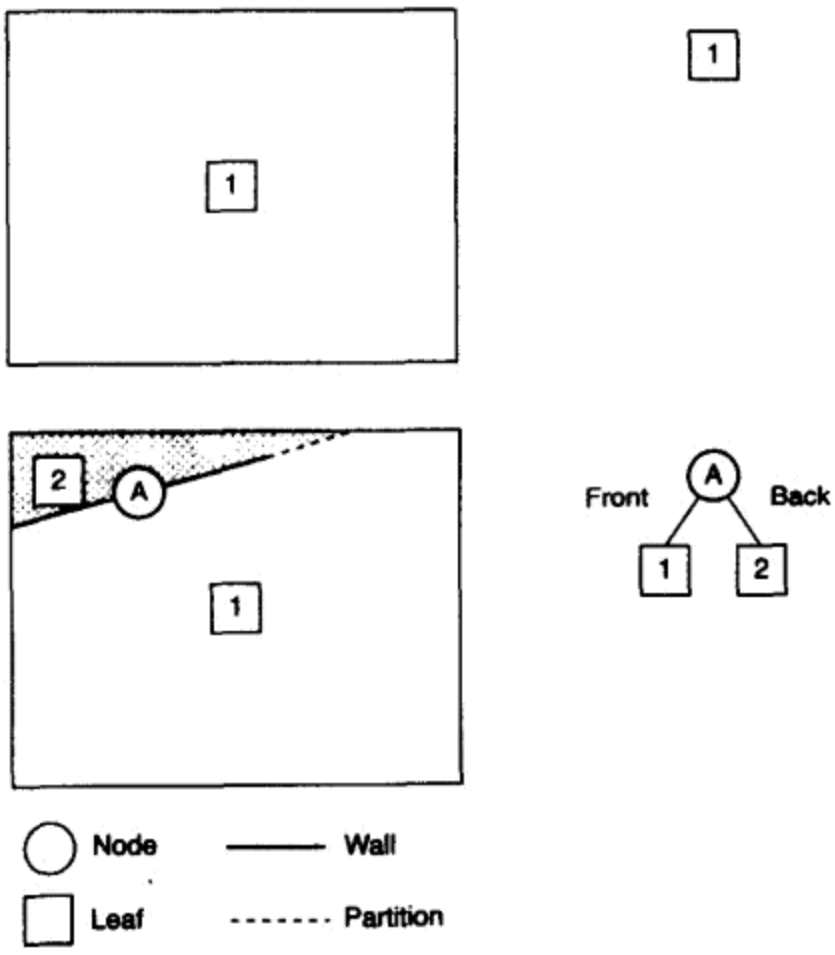


图 10-6 建立 BSP 树之一将第一个分区（墙 A）加进树中

这里，BSP 树的叶是空白空间，这些空白空间的形状与地面和天花板多边形相同。地面和天花板应当有不同高度（从而可以生成楼梯、平台等），因此把地面和天花板高度信息放在这些叶中。图 10-6 中，从叶 1 开始，它是整个空白区。第一个分区（墙 A）把叶一分成二，生成叶 1 和 2。

空白空间总是凸的，因为凸形用直线分割得到两个凸形。因此，地面和天花板多边形也是凸多边形。

图 10-7 把墙 B 和 C 加进树中。墙 B 在墙 A 之前，因此加进 A 的前节点，其把叶 1 分成两个叶（1 和 3）。墙 C 在 A 之前和 B 之后，因此加进 B 节点之后，将叶 3 分成两个叶

(3 和 4)。

注意：增加墙 B 时，叶 3 受墙 B 和墙 A 构成的分区限制。

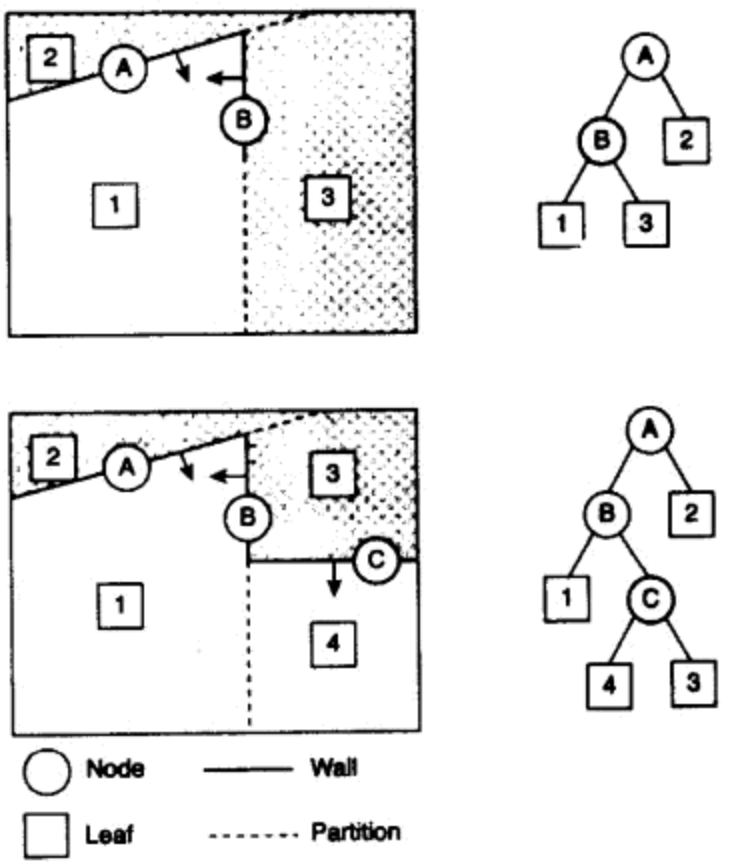


图 10-7　建立 BSP 树之二：把墙 B 和 C 加进树中

图 10-8 继续建立 BSP 树，这时只要再加上墙 D。但墙 D 跨过墙 B 构成的分区。墙不能在树中两个不同位置，因此墙 D 分成两堵墙：D1 与 D2。

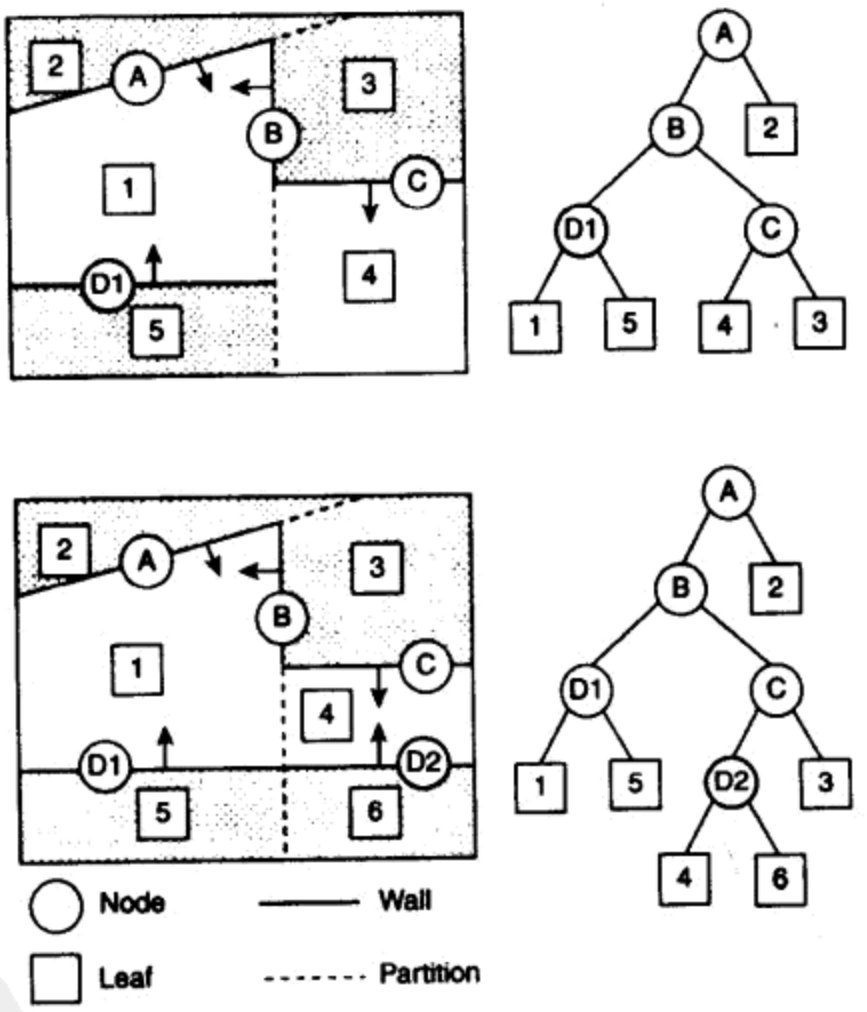


图 10-8　建立 BSP 树之三：墙 D 跨过墙 B 构成的分区，分成两堵墙

这样就建好了树。

在建立 BSP 树的过程中，你会发现，如果按不同顺序加墙，则会得到不同的树。例如，如果按顺序 D、B、C、A 加墙，则得到图 10-9 所示的树。这个树不分割任何墙，因此节点更少。另外，树的深度也较短。但是，这个树不太平衡，所有墙都在第一个节点之前。

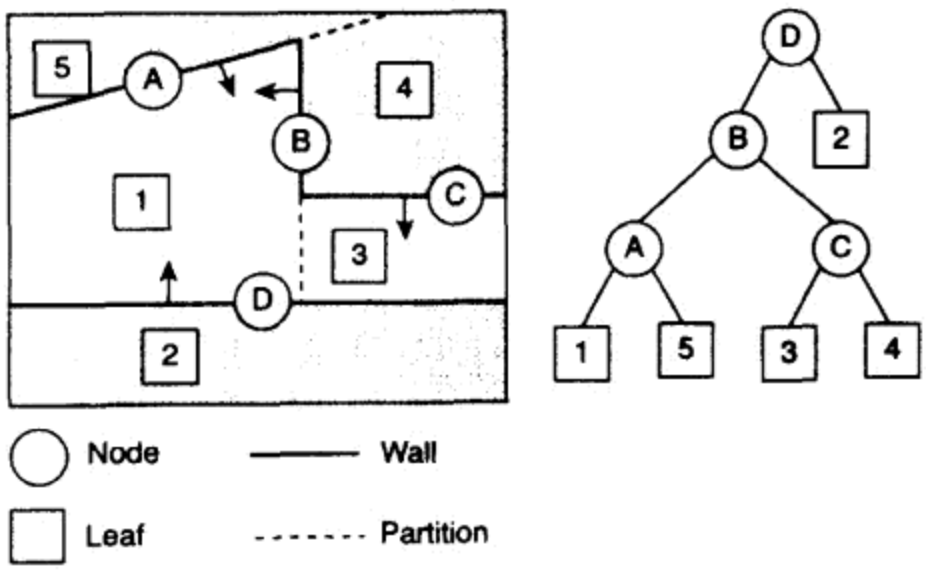


图 10-9　这个 BSP 树节点更少，树的深度也较短

如果有足够的多边形，则更短的平衡树能更快搜索。因此，墙加进树中的顺序非常重要。

确定分区选择顺序的算法要考虑两点：

- 减少分割。分割越少，多边形越少，遍历也越少。
- 树应尽量平衡。更短的平衡树意味着不平衡树更少的遍历。

有时这两个目标是互相矛盾的。让树平衡会产生更多分割，减少分割会让树不平衡，因此要找出正确算法是很复杂的。下面介绍两个思路。

首先，可以减少分割数，但不管树的平衡。每次选择分区时，选择分割最少的分区。例如，不会先选墙 B 后选墙 D，因为墙 B 会分割墙 D。

其次，可以只管树的平衡，而不管分割数，总是选择分区两边墙数相同（至多差 1）的分区。例如，可以选择墙 B 为第一个分区，因为其两边各有两个分区（虽然墙 B 会分割墙）。

也可以将这两种方法结合起来，在一组使树相对平衡的分区中，选择分割数最少的分区。

本章不准备介绍分区选择算法，读者可以自己练习。只要在一列墙中，选择第一个墙作为分区。对于有大量多边形的大图，这不一定是最佳方案，但这些例子中可以这么做。

10.4.2　BSP 树遍历实例

遍历二维 BSP 树与遍历一维 BSP 树相似，有两个规则：

- 按顺序绘制镜头前面的多边形（前节点，当前节点，后节点）。
- 按逆序绘制镜头后面的多边形（后节点，当前节点，前节点）。

以图 10-10 为例，假设镜头在星号所示的位置。这个遍历算法按下列顺序遍历节点：

4,D2,6,C,3,B,1,D1,5,A,2。多边形从前向后遍历，正是所要的结果。

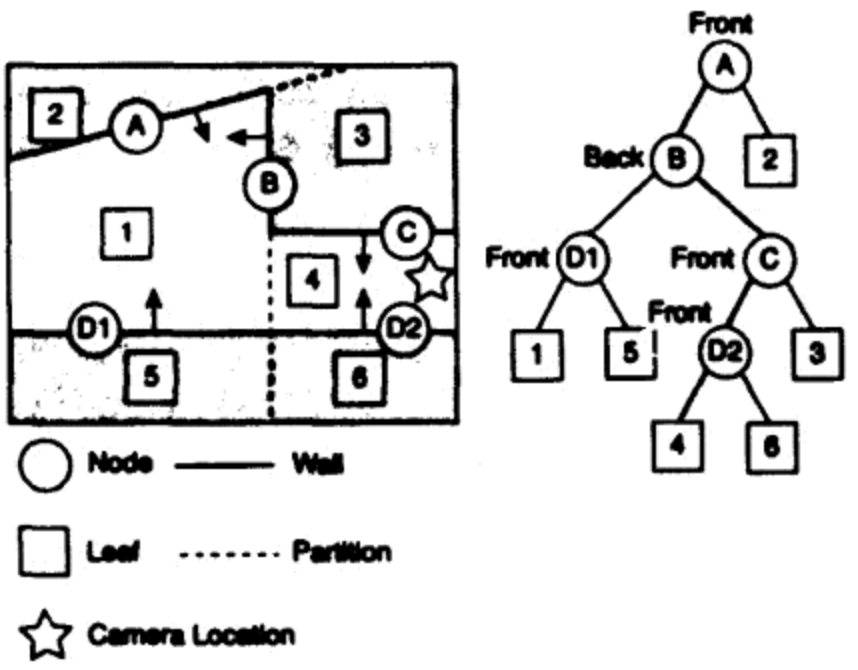


图 10-10　从星形位置遍历树

在这个算法中，第一个遍历的叶节点是镜头所在的叶节点。由于水平多边形可能有不同高度，因此，如果知道游戏者所在的叶节点，则可以确定游戏者应位于什么高度。例如，如果游戏者高度为 100 单位，游戏者在 y=200 的地面上，则游戏者的镜头为 y=300。

建立与遍历 BSP 树中的多边形后，下面要编写二维 BSP 树绘制器的代码。

10.5　实现二维 BSP 树

首先，要生成清单 10.1 所示的 BSPTree 类。这个类定义树的数据结构为节点与叶。和平常一样，这个代码只是个开始，后面要增加更多方法。

清单 10.1　BSPTree.java

```
package com.brackeen.javagamebook.bsp2D;

import java.awt.Rectangle;
import java.util.List;

/**
    BSPTree 类表示多边形的二维 BSP 树。BSPTree 用 BSPTreeBuilder 类建立，可以用
    BSPTreeTraverser 类遍历
*/
public class BSPTree {

    /**
        树的节点，节点的所有子节点在该节点分区的前面或后面
    */
```

```
    public static class Node {
        public Node front;
        public Node back;
        public BSPLine partition;
        public List polygons;
    }
    /**
        树的叶，叶没有分区，没有前后节点
    */
    public static class Leaf extends Node {
        public float floorHeight;
        public float ceilHeight;
        public boolean isBack;
        public List portals;
        public Rectangle bounds;
    }

    private Node root;
    /**
        用指定根节点生成新 BSPTree
    */
    public BSPTree(Node root) {
        this.root = root;
    }
    /**
        取得这个树的根节点
    */
    public Node getRoot() {
        return root;
    }
}
```

BSPTree 类包含内类 Node，它是树中的节点。节点引用前后节点、分区和一列多边形。分区是个 BSPLine，见下节介绍。对非叶节点，多边形表是与分区共线的所有墙多边形表。对叶节点（开阔空间），表中包含地板与天花板多边形。

内类 Leaf 是 Node 的子类，包含地板与天花板高度，后面用其使游戏者和实体保持正确高度。它还有叶的矩形边界，可以确定实体可视性（换句话说，如果叶显示，则其边界中的实体也显示）。

10.5.1　BSP Line

清单 10.2 的 BSPLine 是分区（和其他几个地方）使用的直线。

清单 10.2　BSPLine.java

```
package com.brackeen.javagamebook.bsp2D;

import java.awt.geom.*;
import com.brackeen.javagamebook.math3D.*;

public class BSPLine extends Line2D.Float {
    /**
        直线法线的 X 坐标
    */
    public float nx;
    /**
        直线法线的 Y 坐标
    */
    public float ny;
}
```

BSPLine 是 Line2D.Float 的子类。Line2D.Float 在 java.awt.geom 包中，其浮点字段 x1, y1, x2 与 y2 定义线段。在 BSFLine 中，还要包括直线的法线值，或垂直于直线的方向。法线指向分区“前”面。这个类只是个开头，后面还要在 BSPLine 中增加许多方法。

10.5.2　确定点相对于直线的边

我们要确定多边形在直线前面、后面还是相交。为此可以用第 7 章“三维图形”介绍的多边形剪取方法。对多边形中每个点，确定这个点在直线前面或后面。

还要知道点是否在线上（即与直线共线）。例如，多边形可能有些点与直线共线，有些在直线前面。这时，多边形在直线前面，虽然有些点与直线共线。

确定点在直线前面或后面相当于确定点在多边形前面或后面：只要求出所指向量和直线法线之间的乘积。如果这个值小于 0，则点在后面。如果这个值大于 0，则点在前面；如果这个值等于 0，则点共线。

一个问题是，由于浮点数误差，点很少实际共线（即使 64 位双精度数也不够精确，要保证完全精确，就要无穷位精度）。这可能遇到问题，例如共线点计算成在直线后面，而多边形中其他点都在前面。这时，多边形被看成跨过这条线，显然不准确。

为了解决这个问题，可以把线看成有一定厚度，如图 10-11 所示。

图 10-11 中，虚线所示的实际直线有前线和后线。如果点在前线与后线之间，则认为共线。这样，就可以生成共线区，弥补浮点数误差。

如果直线的法线设定为单位法线（长度为 1），则可以方便地将线前后移动，确定点所

在的边。如果使用一个单位线宽，则可以将直线沿法线方向前后移动半个单位。清单 10.3 的 getSideThick()方法显示了这个做法。

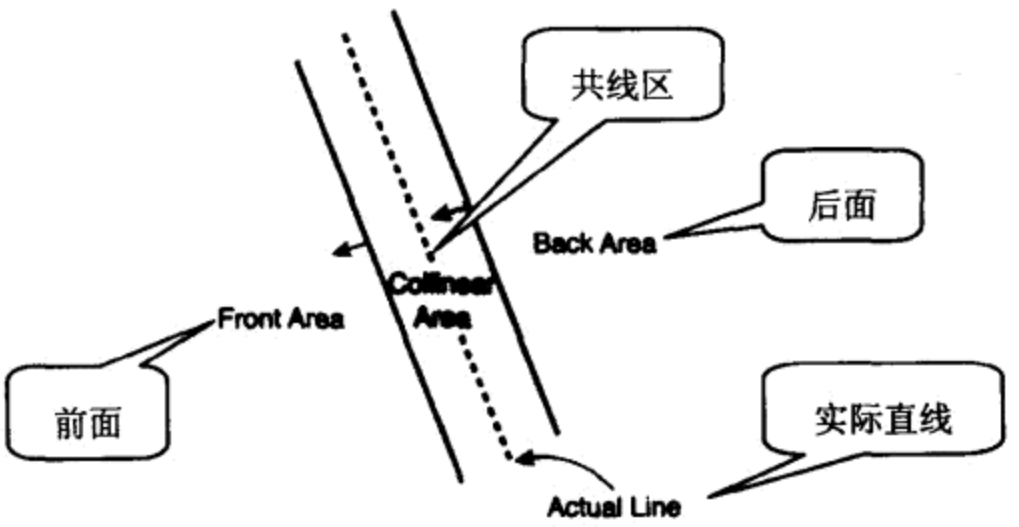


图 10-11　把线看成有一定厚度

清单 10.3　BSPLine.java 的 getSide（）方法

```
    public static final int BACK = -1;
    public static final int COLLINEAR = 0;
    public static final int FRONT = 1;
    public static final int SPANNING = 2;

...

    /**
        范化直线的法线（使法线长度为 1）
    */
    public void normalize() {
        float length = (float)Math.sqrt(nx * nx + ny * ny);
        nx/=length;
        ny/=length;
    }
/**
    取得指定点所在的边。由于浮点数误差，点很少实际共线。为此可以范化直线的法线，将线设
    置为多边形或调用 normalize()方法
    返回 FRONT, BACK 或 COLLINEAR
*/
    public int getSideThin(float x, float y) {
        // 所指向量和直线法线之间的点积
        float side = (x - x1)*nx + (y - y1)*ny;
        return (side < 0)?BACK:(side > 0)?FRONT:COLLINEAR;
    }
```

```
/**
    取得指定点所在的边。这个方法把线看作1个单位厚度，如果点在前线与后线之间，则认
    为共线。为此可以范化直线的法线，将线设置为多边形或调用normalize()方法
    返回FRONT, BACK或COLLINEAR
*/
public int getSideThick(float x, float y) {
    int frontSide = getSideThin(x-nx/2, y-ny/2);
    if (frontSide == FRONT) {
        return FRONT;
    }
    else if (frontSide == BACK) {
        int backSide = getSideThin(x+nx/2, y+ny/2);
        if (backSide == BACK) {
            return BACK;
        }
    }
    return COLLINEAR;
}
```

清单10.3的getSideThick()方法在BSPLine类中。GetSideThin()方法正常处理直线，而getSideThick()方法把线看作1个单位厚度。这两个方法都返回FRONT, BACK或COLLINEAR。

有了这些方法之后，很容易确定另一直线或多边形的两边，清单10.4显示了这些方法。

清单10.4 BSPLine.java的更多getSide（）方法

```
/**
    取得指定线段所在直线的边，返回FRONT, BACK, COLLINEAR或SPANNING
*/

public int getSide(Line2D.Float segment) {
    if (this == segment) {
        return COLLINEAR;
    }
    int p1Side = getSideThick(segment.x1, segment.y1);
    int p2Side = getSideThick(segment.x2, segment.y2);
    if (p1Side == p2Side) {
        return p1Side;
    }
    else if (p1Side == COLLINEAR) {
        return p2Side;
```

```
        }
        else if (p2Side == COLLINEAR) {
            return p1Side;
        }
        else {
            return SPANNING;
        }
    }
    /**
        取得指定多边形所在直线的边，返回 FRONT, BACK, COLLINEAR 或 SPANNING
    */
    public int getSide(BSPPolygon poly) {
        boolean onFront = false;
        boolean onBack = false;

        // 检查每个点
        for (int i=0; i<poly.getNumVertices(); i++) {
            Vector3D v = poly.getVertex(i);
            int side = getSideThick(v.x, v.z);
            if (side == BSPLine.FRONT) {
                onFront = true;
            }
            else if (side == BSPLine.BACK) {
                onBack = true;
            }
        }

        // 将多边形分类
        if (onFront && onBack) {
            return BSPLine.SPANNING;
        }
        else if (onFront) {
            return BSPLine.FRONT;
        }
        else if (onBack) {
            return BSPLine.BACK;
        }
        else {
            return BSPLine.COLLINEAR;
```

```
        }
    }
```

清单 10.4 的方法也在 BSPLine 类中。这两个方法都使用“厚线”。多边形和直线只有一点在前另一点在后时才是 SPANNING,然后要把跨区多边形一分为二，使它一个在前，一个在后。

10.5.3　BSP 多边形

前几章用 TexturedPolygon3D 类处理所有多边形。本章要对 BSP 树中的多边形扩展这个类。清单 10.5 的 BSPPolygon 类是 TexturedPolygon3D 类的子类，包含多边形是墙上还是地上/天花板上的信息；如果是墙，则它包含表示墙的 BSPLine。另外，它还有环境光强度，使每个多边形可以有不同环境光强度。

清单 10.5　BSPPolygon.java

```
package com.brackeen.javagamebook.bsp2D;

import com.brackeen.javagamebook.math3D.*;
/**
    BSPPolygon 是个 TexturedPolygon3D，具有类型(TYPE_FLOOR, TYPE_WALL),环境光强度值和
    BSPLine 表示（如果类型为 TYPE_WALL）
*/
public class BSPPolygon extends TexturedPolygon3D {

    public static final int TYPE_FLOOR = 0;
    public static final int TYPE_WALL = 1;

    private int type;
    private float ambientLightIntensity;
    private BSPLine line;

...
}
```

BSPPolygon 类还有一些 get 与 set 方法，可以取得和设置类型、环境光强度与 BSPLine。

10.5.4　遍历 BSP 树

我们需要两种方法遍历 BSP 树，与一维 BSP 树的例子相似：中序和由前到后顺序。有时只要遍历树中每个多边形（如建立过程中），因此要用中序遍历。绘制时则要按由前到后顺序遍历。

我们生成 BSPTreeTraverser 类进行实际遍历。首先要生成一个监听器接口，使 BSPTreeTraverser 可以在遍历节点中的多边形时通知监听器。清单 10.6 显示了

BSPTreeTraverseListener 接口，通常用于多边形绘制器和处理树中的所有多边形。

清单 10.6 BSPTreeTraverseListener.java

```
package com.brackeen.javagamebook.bsp2D;
/**
    BSPTreeTraverseListener 是 BSPTreeTraverser 的接口，在访问 BSP 多边形时发通知
*/
public interface BSPTreeTraverseListener {
    /**
        访问 BSP 多边形。由 BSPTreeTraverer 调用。如果这个方法返回真值，则 BSPTreeTraverer
        停止当前遍历，否则，如果树中还有未遍历多边形，则 BSPTreeTraverer 继续遍历
    */
    public boolean visitPolygon(BSPPolygon poly,
        boolean isBackLeaf);
}
```

清单 10.6 中，BSPTreeTraverseListener 接口提供 visitPolygon()方法，在访问 BSP 多边形时调用。如果这个方法返回真值，则 BSPTreeTraverser 停止当前遍历，因此可以随时停止遍历。绘图时，屏幕填充后不需要继续遍历，因此可以随时停止遍历。当然，这个方法的参数是多边形，但实际遍历的是节点。因此，BSPTreeTraverser 中提供一个方法，访问节点及其中每个多边形，见清单 10.7。

清单 10.7 BSPTreeTraverser.java 的 visitNode（）方法

```
    private boolean traversing;
    private BSPTreeTraverseListener listener;
    private GameObjectManager objectManager;


...
    /**
        访问树中节点。对节点中每个多边形调用 BSPTreeTraverseListener 的 visitPolygon()方法
    */
    private void visitNode(BSPTree.Node node) {
        if (!traversing || node.polygons == null) {
            return;
        }

        boolean isBack = false;
        if (node instanceof BSPTree.Leaf) {
            BSPTree.Leaf leaf = (BSPTree.Leaf)node;
            isBack = leaf.isBack;
```

```
            // 将这个叶的边界标为在游戏实体管理器中显示
            if (objectManager != null && leaf.bounds != null) {
                objectManager.markVisible(leaf.bounds);
            }
        }

        // 访问每个多边形
        for (int i=0; traversing && i<node.polygons.size(); i++) {
            BSPPolygon poly = (BSPPolygon)node.polygons.get(i);
            traversing = listener.visitPolygon(poly, isBack);
        }
    }
```

这个方法只是访问节点中每个多边形。如果有GameObjectManager(见第9章“三维对象”)，则这个方法通知管理器，显示叶节点边界内的实体。

下面要实际进行一些遍历。

10.5.5 中序遍历

首先，作一个简单的中序遍历，见清单10.8。

清单10.8 BSPTreeTraverser.java的中序遍历方法

```
/**
    中序遍历树
*/
public void traverse(BSPTree tree) {
    traversing = true;
    traverseInOrder(tree.getRoot());
}

/**
    中序遍历节点
*/
private void traverseInOrder(BSPTree.Node node) {
    if (traversing && node != null) {
        traverseInOrder(node.front);
        visitNode(node);
        traverseInOrder(node.back);
    }
}
```

这些方法没什么新东西，只是在traversing布尔值设置为假值时停止遍历。

和平常一样，我们用递归方法遍历树。如果你担心递归速度，先别管它。Java 进行二叉树遍历时，不用递归也无法提高速度。

有些人想消除递归，避免增加更多开销。例如，如果不用递归，则计算被斐波纳契数列的速度要快得多，因为不用递归可以消除方法调用开销和堆栈开销。

对于二叉树遍历，不用递归时可以实现自己的堆栈，但自己的堆栈并不比虚拟机的堆栈更快。因此，我们用递归方法使代码更简洁、更可读。

10.5.6 由前到后遍历

下面要生成按由前到后顺序遍历的方法，和一维 BSP 例子中一样，如清单 10.9 所示。

清单 10.9 BSPTreeTraverser.java 的由前到后遍历方法

```
    private float x;
    private float z;

...

    /**
        用指定视图位置按由前到后顺序
    */
    public void traverse(BSPTree tree, Vector3D viewLocation) {
        x = viewLocation.x;
        z = viewLocation.z;
        traversing = true;
        traverseDrawOrder(tree.getRoot());
    }
    /**
        用当前视图位置按由前到后顺序
    */
    private void traverseDrawOrder(BSPTree.Node node) {
        if (traversing && node != null) {
            if (node instanceof BSPTree.Leaf) {
              // 无分区，只处理多边形
                visitNode(node);
            }
            else if (node.partition.getSideThin(x,z) != BSPLine.BACK) {
                traverseDrawOrder(node.front);
                visitNode(node);
                traverseDrawOrder(node.back);
            }
            else {
```

```
                traverseDrawOrder(node.back);
                visitNode(node);
                traverseDrawOrder(node.front);
            }
        }
    }
```

清单 10.9 的 traverse()方法用指定视图位置按由前到后顺序遍历，对每个节点调用 visitNode()，在遍历多边形时通知监听器。

这样就完成了遍历。当然，还要实现 BSPTreeTraverseListener 类和另外几个类之后才能实际绘制多边形。首先要回头看看树的建立。

10.5.7 树的建立

下面把建立 BSP 树分为 2 个部分：建立树和按直线剪取多边形。

树的建立和遍历一样是个递归过程。基本思路是：取一个节点及其多边形表，选择分区，用分区前面的多边形建立前节点，然后用分区后面的多边形建立后节点，见清单 10.10 的 BSPTreeBuilder 类。

清单 10.10 BSPTreeBuilder.java

```
/**
    建立 BSP 树
*/
public BSPTree build(List polygons) {
    currentTree = new BSPTree(createNewNode(polygons));
    buildNode(currentTree.getRoot());
    return currentTree;
}
/**
    在 BSP 树中建立节点
*/
protected void buildNode(BSPTree.Node node) {

    // 叶节点什么也不建
    if (node instanceof BSPTree.Leaf) {
        return;
    }

    // 相对于分区将多边形分类（前，后，同线）
    ArrayList collinearList = new ArrayList();
    ArrayList frontList = new ArrayList();
```

```
        ArrayList backList = new ArrayList();
        List allPolygons = node.polygons;
        node.polygons = null;
        for (int i=0; i<allPolygons.size(); i++) {
            BSPPolygon poly = (BSPPolygon)allPolygons.get(i);
            int side = node.partition.getSide(poly);
            if (side == BSPLine.COLLINEAR) {
                collinearList.add(poly);
            }
            else if (side == BSPLine.FRONT) {
                frontList.add(poly);
            }
            else if (side == BSPLine.BACK) {
                backList.add(poly);
            }
            else if (side == BSPLine.SPANNING) {
                BSPPolygon front = clipBack(poly, node.partition);
                BSPPolygon back = clipFront(poly, node.partition);
                if (front != null) {
                    frontList.add(front);
                }
                if (back != null) {
                    backList.add(back);
                }
            }
        }

        // 表的清理与赋值
        collinearList.trimToSize();
        frontList.trimToSize();
        backList.trimToSize();
        node.polygons = collinearList;
        node.front = createNewNode(frontList);
        node.back = createNewNode(backList);

        // 建立前后节点
        buildNode(node.front);
        buildNode(node.back);
        if (node.back instanceof BSPTree.Leaf) {
```

```
            ((BSPTree.Leaf)node.back).isBack = true;
        }
    }
    /**
        从多边形表生成新节点。如果多边形中没有墙，则生成叶节点
    */
    protected BSPTree.Node createNewNode(List polygons) {

        BSPLine partition = choosePartition(polygons);

        // 没有分区，因此是叶
        if (partition == null) {
            BSPTree.Leaf leaf = new BSPTree.Leaf();
            leaf.polygons = polygons;
            buildLeaf(leaf);
            return leaf;
        }
        else {
            BSPTree.Node node = new BSPTree.Node();
            node.polygons = polygons;
            node.partition = partition;
            return node;
        }
    }
    /**
        选择多边形表中作为分区的直线。这个方法返回第一个垂直多边形构成的直线，找不到则
        返回null。更好的方法是选择分割最少和提供更平衡的完备树的分区
    */
    protected BSPLine choosePartition(List polygons) {
        for (int i=0; i<polygons.size(); i++) {
            BSPPolygon poly = (BSPPolygon)polygons.get(i);
            if (poly.isWall()) {
                return new BSPLine(poly);
            }
        }
        return null;
    }
    /**
        建立树的叶，计算其他信息，如叶边界、地面高度和天花板高度
```

```
*/
protected void buildLeaf(BSPTree.Leaf leaf) {

    if (leaf.polygons.size() == 0) {
      // 叶表示空白空间
        leaf.ceilHeight = Float.MAX_VALUE;
        leaf.floorHeight = Float.MIN_VALUE;
        leaf.bounds = null;
        return;
    }

    float minX = Float.MAX_VALUE;
    float maxX = Float.MIN_VALUE;
    float minY = Float.MAX_VALUE;
    float maxY = Float.MIN_VALUE;
    float minZ = Float.MAX_VALUE;
    float maxZ = Float.MIN_VALUE;

   // 求最小 y、最大 y 和边界值
    Iterator i = leaf.polygons.iterator();
    while (i.hasNext()) {
        BSPPolygon poly = (BSPPolygon)i.next();
        for (int j=0; j<poly.getNumVertices(); j++) {
            Vector3D v = poly.getVertex(j);
            minX = Math.min(minX, v.x);
            maxX = Math.max(maxX, v.x);
            minY = Math.min(minY, v.y);
            maxY = Math.max(maxY, v.y);
            minZ = Math.min(minZ, v.z);
            maxZ = Math.max(maxZ, v.z);
        }
    }

   // 求叶中的任何平台
    i = leaf.polygons.iterator();
    while (i.hasNext()) {
        BSPPolygon poly = (BSPPolygon)i.next();
      // 对地面
        if (poly.getNormal().y == 1) {
```

```
                float y = poly.getVertex(0).y;
                if (y > minY && y < maxY) {
                    minY = y;
                }
            }
        }

        // 设置叶值
        leaf.ceilHeight = maxY;
        leaf.floorHeight = minY;
        leaf.bounds = new Rectangle(
            (int)Math.floor(minX), (int)Math.floor(minZ),
            (int)Math.ceil(maxX-minX+1),
            (int)Math.ceil(maxZ-minZ+1));
    }
```

在这个代码中，build()方法生成新节点，包含每个多边形。注意这个方法只取BSPPolygons。由于使用二维 BSP 树，因此很容易判断多边形是垂直墙还是水平地面与天花板。createNewNode()方法生成新节点，包含指定表中每个多边形。

buildNode 方法根据节点分区建立节点。这个方法相对于分区将多边形分类（前、后、共线），然后将每个多边形放在相应子节点中。因此，分区前面的多边形放在节点的前子节点中，分区后面的多边形放在节点的后子节点中，跨分区的多边形用稍后介绍的 clipFront()与 clipBack()方法分割。

choosePartition()方法的多边形表中选择分区。前面曾介绍过，我们不作太高明的选择，只是把表中第一个墙作为分区。读者可以优化这个方法。

buildLeaf()方法建立树的叶，计算其他信息，如叶边界、地面高度和天花板高度。

建立 BSP 树之后，下面要编写用直线剪取多边形的方法。

10.5.8 两条线求交

用直线剪取多边形时，实际上忽略多边形 Y 成分，进行二维剪取，类似于第 7 章介绍的多边形剪取，但要求多边形一边与直线的交点。换句话说，就是两条线求交。

图 10-12 显示了两条线求交的思路。

图 10-12 中，I 点是两条线的交点。u 值是直线 A 上交点的比例，如果两个线段相交，则 u 介于 0 和 1 之间。直线 B 上的 v 也一样。

图 10-12 的方程只是直线的基本方程，但可以用这些公式求两条线的交点。如果要解 u，则可以得到：

$$numerator = (B2_y-B1_y)(A2_x-A1_x)-(B2_x-B1_x)(A2_y-A1_y)$$

$$denominator = (B2_y-B1_y)(A2_x-A1_x)-(B2_x-B1_x)(A2_y-A1_y)$$

$$u = numerator/denominator$$

利用 u，即可求出交点：

$x = A1_x + u(A2_x - A1_x)$

$y = A1_y + u(A2_y - A1_y)$

BSPLine 的 getIntersection()方法中使用了这些方程，见清单 10.11。

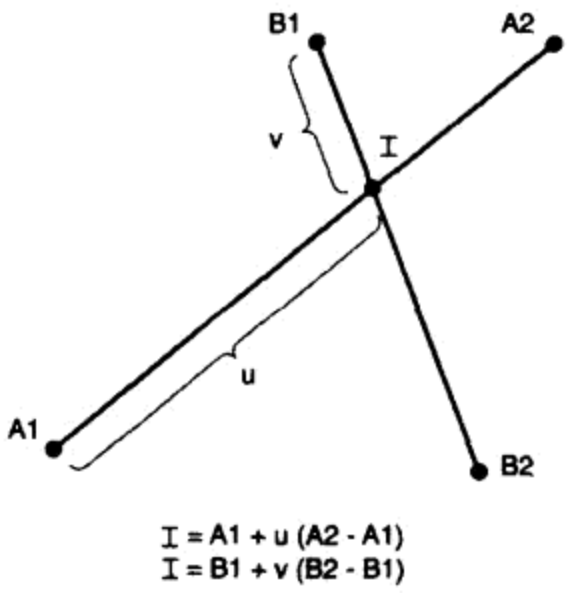


图 10-12　两条线求交

清单 10.11　BSPLine.java 的求交方法

```
/**
    返回直线上交点的比例。如果两条线相交，则返回 0 和 1 之间的值。例如，返回 0 表示交
    点在(x1, y1)，返回 1 表示交点在(x2, y2)，返回 0.5 表示交点在中点。返回-1 时，表示两条
    线平行
*/
public float getIntersection(Line2D.Float line) {
    // 两个向量，A1->A2 与 B1->B2 的交点 I 如下:
     // I = A1 + Ua * (A2 - A1)
     // I = B1 + Ub * (B2 - B1)
     //
    // u 的则得到下列公式
    // 返回 u
     float denominator = (line.y2 - line.y1) * (x2 - x1) -
         (line.x2 - line.x1) * (y2 - y1);

    // 检查两条线是否平行
     if (denominator == 0) {
         return -1;
     }

     float numerator = (line.x2 - line.x1) * (y1 - line.y1) -
         (line.y2 - line.y1) * (x1 - line.x1);
```

```
    return numerator / denominator;
}
/**
    返回这条直线与指定直线的交点
*/
public Point2D.Float getIntersectionPoint(Line2D.Float line) {
    return getIntersectionPoint(line, null);
}
/**
    返回这条直线与指定直线的交点。如果交点为null，则生成新点
*/
public Point2D.Float getIntersectionPoint(Line2D.Float line,
    Point2D.Float intersection)
{
    if (intersection == null) {
        intersection = new Point2D.Float();
    }
    float fraction = getIntersection(line);
    intersection.setLocation(
        x1 + fraction * (x2 - x1),
        y1 + fraction * (y2 - y1));
    return intersection;
}
```

getIntersection()方法求直线u上交点的比例，getIntersectionPoint()方法计算交点的(x,y)值，并在没有指定Point2D.Float时生成新Point2D.Float。

如果处理三维BSP树，则要按平面而不是用直线剪取多边形，这样就要用平面剪取多边形每边，因此是线与平面求交，而不是两条线求交。

计算交点后，就可以剪取多边形了。

10.5.9 用直线剪取多边形

用直线剪取多边形与第7章介绍的多边形剪取相似，是简单二维剪取。清单10.12显示的方法忽略多边形y成分，进行二维剪取。

清单10.12 BSPBuilder.java的剪取方法

```
/**
    剪取指定直线前面的多边形部分。返回直线后面的多边形部分。如果直线不分割多边形，
    则返回null，原多边形不变
*/
protected BSPPolygon clipFront(BSPPolygon poly, BSPLine line) {
```

```
        return clip(poly, line, BSPLine.FRONT);
    }
    /**
        剪取指定直线后面的多边形部分。返回直线前面的多边形部分。如果直线不分割多边形，
        则返回 null，原多边形不变
    */
    protected BSPPolygon clipBack(BSPPolygon poly, BSPLine line) {
        return clip(poly, line, BSPLine.BACK);
    }
    /**
        剪取 BSPPolygon，将多边形指定边（BSPLine.FRONT 或 BSPLine.BACK）消除，返回剪
        取的多边形。如果直线不分割多边形，则返回 null，原多边形不变
    */
    protected BSPPolygon clip(BSPPolygon poly, BSPLine line,
        int clipSide)
    {
        ArrayList vertices = new ArrayList();
        BSPLine polyEdge = new BSPLine();

        // 增加不在剪取边上的顶点
        Point2D.Float intersection = new Point2D.Float();
        for (int i=0; i<poly.getNumVertices(); i++) {
            int next = (i+1) % poly.getNumVertices();
            Vector3D v1 = poly.getVertex(i);
            Vector3D v2 = poly.getVertex(next);
            int side1 = line.getSideThin(v1.x, v1.z);
            int side2 = line.getSideThin(v2.x, v2.z);
            if (side1 != clipSide) {
                vertices.add(v1);
            }

            if ((side1 == BSPLine.FRONT && side2 == BSPLine.BACK) ||
                (side2 == BSPLine.FRONT && side1 == BSPLine.BACK))
            {
              // 保证 v1.z < v2.z
                if (v1.z > v2.z) {
                    Vector3D temp = v1;
                    v1 = v2;
                    v2 = temp;
```

```
            }
            polyEdge.setLine(v1.x, v1.z, v2.x, v2.z);
            float f = polyEdge.getIntersection(line);
            Vector3D tPoint = new Vector3D(
                v1.x + f * (v2.x - v1.x),
                v1.y + f * (v2.y - v1.y),
                v1.z + f * (v2.z - v1.z));
            vertices.add(tPoint);
        // 删除任何生成的t接头
            removeTJunctions(v1, v2, tPoint);
        }

    }

    // 删除相邻等顶点，使(A->A)变成(A)
    for (int i=0; i<vertices.size(); i++) {
        Vector3D v = (Vector3D)vertices.get(i);
        Vector3D next = (Vector3D)vertices.get(
            (i+1) % vertices.size());
        if (v.equals(next)) {
            vertices.remove(i);
            i--;
        }
    }

    if (vertices.size() < 3) {
        return null;
    }

    // 建立多边形
    Vector3D[] array = new Vector3D[vertices.size()];
    vertices.toArray(array);
    return poly.clone(array);
}
```

剪取多边形时还要另外两步。首先要删除相邻等顶点，使(A->A)变成(A)。例如，如果两个相邻顶点共点，则边长度为0，应删除其中一个顶点。

还要删除任何生成的t接头，下面介绍T形接头。

10.5.10 删除 T 形接头

图 10-13 显示了 T 形接头接缝的例子。由于浮点数误差，T 形接头可能在多边形之间产生裂缝，有时只有在某个角度才能看到。本例中，大多边形只有三个顶点。三个多边形变换时，显示裂缝。通常，这个裂缝是看不到的，但有时会显示为一、两个像素或一条多个像素的裂纹。

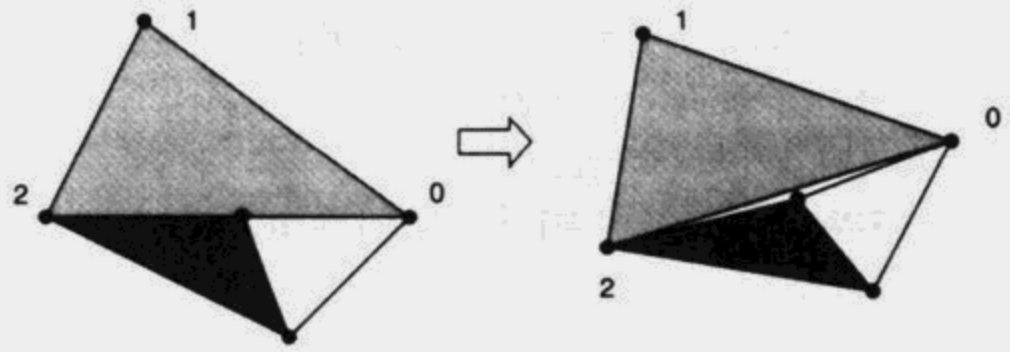


图 10-13 大多边形只有三个顶点。T 形接头可能在多边形之间产生裂缝，有时只有在某个角度才能看到

但是，这个问题也要用无穷精度浮点数才能解决。和平常一样，我们也有解决办法。

要消除 T 形接头间的接缝，可以在跨边多边形的交点处增加一个顶点，如图 10-14 所示。

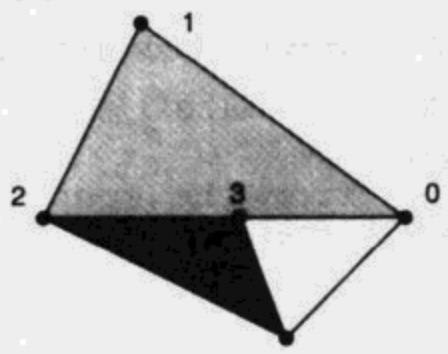


图 10-14 要消除 T 形接头间的接缝，让大多边形加一个顶点，与另外两个多边形共点

因此，要消除 T 形接头间的接缝，代码要在多边形一边发生分割处插入顶点，使建立的 BSP 树不会在分割多边形时生成新的 T 形接头。清单 10.13 显示了这个代码。

清单 10.13 消除 T 形接头间的接缝

```
/**
    沿(v1, v2)指定的直线从当前树中消除 T 形接头间的接缝。求出有这个边的所有多边形并
    在其间插入 T 形交点
*/
protected void removeTJunctions(final Vector3D v1,
    final Vector3D v2, final Vector3D tPoint)
{
    BSPTreeTraverser traverser = new BSPTreeTraverser(
        new BSPTreeTraverseListener() {
            public boolean visitPolygon(BSPPolygon poly,
                boolean isBackLeaf)
```

```
                {
                    removeTJunctions(poly, v1, v2, tPoint);
                    return true;
                }
            }
        );
        traverser.traverse(currentTree);
    }
    /**
        从指定多边形消除T形接头。如果v1与v2点之间没有其他点，则在其间插入T形交点
    */
    protected void removeTJunctions(BSPPolygon poly,
        Vector3D v1, Vector3D v2, Vector3D tPoint)
    {
        for (int i=0; i<poly.getNumVertices(); i++) {
            int next = (i+1) % poly.getNumVertices();
            Vector3D p1 = poly.getVertex(i);
            Vector3D p2 = poly.getVertex(next);
            if ((p1.equals(v1) && p2.equals(v2)) ||
                (p1.equals(v2) && p2.equals(v1)))
                {
                    poly.insertVertex(next, tPoint);
                    return;
                }
            }
        }
```

清单10.13中用强力方法求匹配的T形接头：检查树中每个多边形每一边。这个方法是可行的，但也可以只搜索与T形接头直线共边的多边形，从而加快速度。

注意：建立BSP树时只清除从剪取多边形生成的T形接头。换句话说，它假设传入BSP建立器的多边形没有T形接头。

这样就有了建立所要的一切：建立算法和多边形剪取，还有了BSP遍历，因此下面建立一个简单演示程序。

10.5.11 测试BSP树

尽管我们还没有在三维视景中的BSP树中绘制多边形，但已经可以生成BSP的测试程序。BSPTest2D就是起这个作用的，绘制三维视景的鸟瞰图，像前面的例子与图一样。和前几章一样，这个演示程序扩展GameCore类，清单10.14显示了BSPTest2D中建立BSP树的主要方法。

清单 10.14 用 BSPTest2D.java 建立 BSP 树

```
    public void createPolygons() {
        // 地面多边形
        BSPPolygon floor = new BSPPolygon(new Vector3D[] {
            new Vector3D(0,0,0), new Vector3D(0,0,600),
            new Vector3D(800,0,600), new Vector3D(800,0,0)
            }, BSPPolygon.TYPE_FLOOR);
        polygons.add(floor);

        // 浏览者看墙时从左向右定义的顶点
        BSPPolygon wallA = createPolygon(
            new BSPLine(0, 150, 500, 75), 0, 300);
        BSPPolygon wallB = createPolygon(
            new BSPLine(500, 75, 500, 300), 0, 300);
        BSPPolygon wallC = createPolygon(
            new BSPLine(500, 300, 800, 300), 0, 300);
        BSPPolygon wallD = createPolygon(
            new BSPLine(800, 450, 0, 450), 0, 300);
        polygons.add(wallA);
        polygons.add(wallB);
        polygons.add(wallC);
        polygons.add(wallD);
    }

    public BSPPolygon createPolygon(BSPLine line, float bottom,
        float top)
    {
        return new BSPPolygon(new Vector3D[] {
            new Vector3D(line.x1, bottom, line.y1),
            new Vector3D(line.x2, bottom, line.y2),
            new Vector3D(line.x2, top, line.y2),
            new Vector3D(line.x1, top, line.y1)
            }, BSPPolygon.TYPE_WALL);
    }

    public void buildTree() {
        BSPTreeBuilder builder = new BSPTreeBuilder();
        bspTree = builder.build(polygons.subList(0, numWalls+1));
```

```
}
```

清单 10.14 的 BSPPolygons 符合前面的平面图例子，生成一个盖住整个区域的大地面，在实际中是不真实的（我们看不到墙后面的地面），但作为例子没问题。

BSPTest2D 其余部分从鸟瞰角度画多边形，如图 10-15 所示。

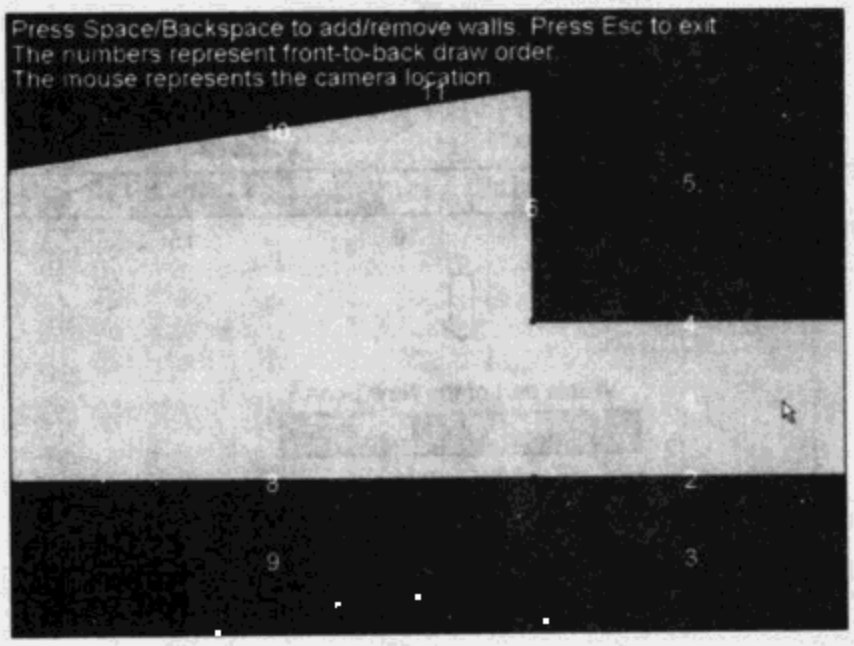


图 10-15　BSPTest2D 的屏幕图形

图 10-15 中，鼠标表示镜头位置，数字表示从镜头位置画多边形的顺序。

这样就有了演示程序，可以建立和遍历树了。下面要转入本章的目的：实际绘制 BSP 树中的多边形。

10.6　从前到后画多边形

有了从前向后遍历多边形的 BSP 树后，要从前向后绘制多边形，需要做两件事：

- 避免重画屏幕上已有的像素。
- 方便地检查视图是否填完（是否画好了每个像素）

可以检查和设置 z 缓存，但我们要避免这个方法。要检查视图是否填完，也可以测试要画的每个像素，但这样也很麻烦。实际上，由于从前向后绘制多边形，只需要 1 位的 z 缓存，因为绘制像素时，它不会重画。

由于我们最终要把多边形变成水平扫描线，因此很容易跟踪视图中已经画好哪些扫描线。为此，对视图窗口中每一行像素保存一个扫描表，如图 10-16 所示。

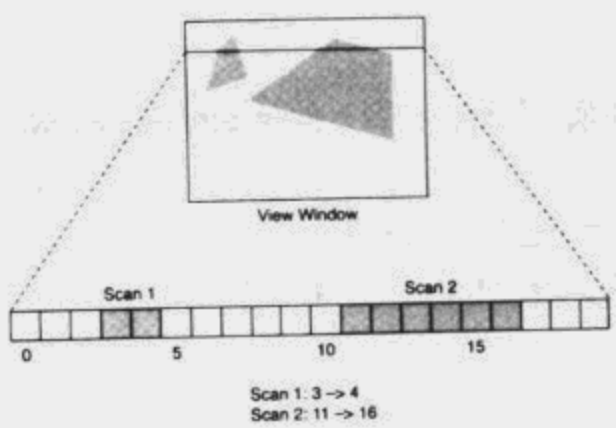


图 10-16　视图窗口中每一行像素保存一个扫描表

绘制新多边形时，进行扫描转换后，将这个扫描线加进视图窗口的扫描表中。图 10-17 显示了一个例子，图中简化了多边形扫描，不画视图中已有部分。另外，这一行的视图窗口的扫描表合并成一个扫描。

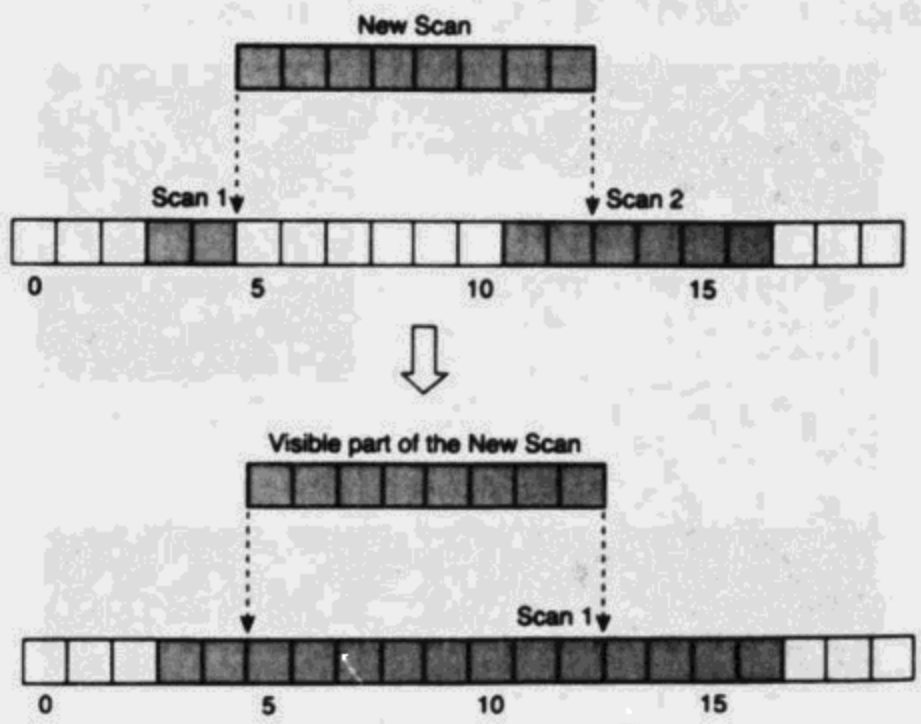


图 10-17　例中将新扫描线加进视图窗口的扫描表中，新扫描只显示可见部分，两个现有扫描线合并成一个扫描

新扫描只显示可见部分利用这个方法，只要检查每个水平扫描是否填上一个扫描，很容易检查视图是否填充。

首先要写一些代码，管理扫描行，见清单 10.15 的 SortedScanList，这是 SortedScanConverter 的内类。

清单 10.15　SortedScanConverter 的内类 SortedScanList

```
/**
    SortedScanList 类表示行的一系列扫描。可以增加新扫描和剪取行中显示部分
*/
private static class SortedScanList {

    private int length;
    private Scan[] scans;
    /**
        生成缺省容量（每行扫描数）的新 SortedScanList。
    */
    public SortedScanList() {
        this(DEFAULT_SCANLIST_CAPACITY);
    }
    /**
        生成指定容量（每行扫描数）的新 SortedScanList。
```

```
*/
public SortedScanList(int capacity) {
    scans = new Scan[capacity];
    for (int i=0; i<capacity; i++) {
        scans[i] = new Scan();
    }
    length = 0;
}
/**
    清除这个扫描表
*/
public void clear() {
    length = 0;
}
/**
    返回这个扫描表的扫描数
*/
public int getNumScans() {
    return length;
}
/**
    取得这个扫描表的第几个扫描
*/
public Scan getScan(int index) {
    return scans[index];
}
/**
    检查这个扫描表是否只有一个扫描，这个扫描是否等于指定左右值
*/
public boolean equals(int left, int right) {
    return (length == 1 && scans[0].equals(left,right));
}
/**
    在这个行中增加和剪取扫描，将显示部分（差）放在指定 SortedScanList 中
*/
public void add(int left, int right, SortedScanList diff) {
    for (int i=0; i<length && left <= right; i++) {
        Scan scan = scans[i];
        int maxRight = scan.left - 1;
```

```
            if (left <= maxRight) {
                if (right < maxRight) {
                    diff.add(left, right);
                    insert(left, right, i);
                    return;
                }
                else {
                    diff.add(left, maxRight);
                    scan.left = left;
                    left = scan.right + 1;
                    if (merge(i)) {
                        i--;
                    }
                }
            }
            else if (left <= scan.right) {
                left = scan.right + 1;
            }
        }
        if (left <= right) {
            insert(left, right, length);
            diff.add(left, right);
        }

    }
    // add() 帮助方法
    private void growCapacity() {
        int capacity = scans.length;
        int newCapacity = capacity*2;
        Scan[] newScans = new Scan[newCapacity];
        System.arraycopy(scans, 0, newScans, 0, capacity);
        for (int i=length; i<newCapacity; i++) {
            newScans[i] = new Scan();
        }
        scans = newScans;
    }

    private void add(int left, int right) {
        if (length == scans.length) {
```

```
            growCapacity();
        }
        scans[length].setTo(left, right);
        length++;
    }

    private void insert(int left, int right, int index) {
        if (index > 0) {
            Scan prevScan = scans[index-1];
            if (prevScan.right == left - 1) {
                prevScan.right = right;
                return;
            }
        }

        if (length == scans.length) {
            growCapacity();
        }
        Scan last = scans[length];
        last.setTo(left, right);
        for (int i=length; i>index; i--) {
            scans[i] = scans[i-1];
        }
        scans[index] = last;
        length++;
    }

    private void remove(int index) {
        Scan removed = scans[index];
        for (int i=index; i<length-1; i++) {
            scans[i] = scans[i+1];
        }
        scans[length-1] = removed;
        length--;
    }

    private boolean merge(int index) {
        if (index > 0) {
            Scan prevScan = scans[index-1];
```

```
                Scan thisScan = scans[index];
                if (prevScan.right == thisScan.left-1) {
                    prevScan.right = thisScan.right;
                    remove(index);
                    return true;
                }
            }
            return false;
        }
    }
```

这个代码管理扫描表，尽可能合并相邻扫描。公开方法 add()可以增加多边形扫描，取得 diff 变元中的显示差别。

要利用这个扫描表类，生成清单 10.16 所示的新 ScanConverter，可以用其代替第 7 章的普通 ScanConverter。

清单 10.16 SortedScanConverter.java

```
package com.brackeen.javagamebook.graphics3D;

import com.brackeen.javagamebook.math3D.*;

/**
    ScanConverter 从前往后绘制排序多边形而不重画。多边形增加与剪取到视图窗口中的表。画每
    个帧之前调用清除方法
*/
public class SortedScanConverter extends ScanConverter {

    protected static final int DEFAULT_SCANLIST_CAPACITY = 8;

    private SortedScanList[] viewScans;
    private SortedScanList[] polygonScans;
    private boolean sortedMode;

    /**
        生成指定 ViewWindow 的新 SortedScanConverter。扫描变换之间可能改变 ViewWindow
        属性。缺省排序方式为关，但可以用 setSortedMode()方法打开
    */
    public SortedScanConverter(ViewWindow view) {
        super(view);
        sortedMode = false;
```

```
}
/**
    清除当前视图扫描。对每帧都调用这个方法
*/
public void clear() {
    if (viewScans != null) {
        for (int y=0; y<viewScans.length; y++) {
            viewScans[y].clear();

        }
    }
}
/**
    设置排序方式，使这个扫描转换器能假设多边形从前向后画，应对已对这个视图扫描的多
    边形进行剪取
*/
public void setSortedMode(boolean b) {
    sortedMode = b;
}
/**
    取得指定行的第几个扫描
*/
public Scan getScan(int y, int index) {
    return polygonScans[y].getScan(index);
}
/**
    取得指定行的扫描个数
*/
public int getNumScans(int y) {
    return polygonScans[y].getNumScans();
}
/**
    检查视图是否填充
*/
public boolean isFilled() {
    if (viewScans == null) {
        return false;
    }
    int left = view.getLeftOffset();
```

```
        int right = left + view.getWidth() - 1;
        for (int y=view.getTopOffset(); y<viewScans.length; y++) {
            if (!viewScans[y].equals(left, right)) {
                return false;
            }
        }
        return true;
    }

    protected void ensureCapacity() {
        super.ensureCapacity();
        int height = view.getTopOffset() + view.getHeight();
        int oldHeight = (viewScans == null)?0:viewScans.length;
        if (height != oldHeight) {
            SortedScanList[] newViewScans =
                new SortedScanList[height];
            SortedScanList[] newPolygonScans =
                new SortedScanList[height];
            if (oldHeight != 0) {
                System.arraycopy(viewScans, 0, newViewScans, 0,
                    Math.min(height, oldHeight));
                System.arraycopy(polygonScans, 0, newPolygonScans,
                    0, Math.min(height, oldHeight));
            }
            viewScans = newViewScans;
            polygonScans = newPolygonScans;
            for (int i=oldHeight; i<height; i++) {
                viewScans[i] = new SortedScanList();
                polygonScans[i] = new SortedScanList();
            }
        }
    }
    /**
        扫描转换一个多边形。如果 sortedMode 打开，则加进并对已在这个视图中的多边形进行剪
        取
    */
    public boolean convert(Polygon3D polygon) {
        boolean visible = super.convert(polygon);
        if (!sortedMode || !visible) {
```

```
            return visible;
        }

        //对已在这个视图中的多边形进行剪取
        visible = false;
        for (int y=getTopBoundary(); y<=getBottomBoundary(); y++) {
            Scan scan = getScan(y);
            SortedScanList diff = polygonScans[y];
            diff.clear();
            if (scan.isValid()) {
                viewScans[y].add(scan.left, scan.right, diff);
                visible |= (polygonScans[y].getNumScans() > 0);
            }
        }

        return visible;

    }
}
```

SortedScanConverter 假设只要多边形处于排序方式，就按从前到后顺序绘制。视图窗口中每一行像素保存一个扫描表。另外，当前多边形每一行像素保存一个扫描表。isFilled()方法检查视图窗口的扫描表，看看视图窗口是否填完。

值得注意的是，isFilled()方法可能成本太高，不能在绘制每个多边形之后调用。也可以每隔三几个多边形调用这个方法，缺点是树遍历时间会加长（而且另外两个多边形要进行扫描转换），但总的来说，会比绘制每个多边形之后调用 isFilled()方法更快。

使用 SortedScanConverter 与使用 ScanConverter 相似，只是每一行可以有多个扫描。例如：

```
int y = scanConverter.getTopBoundary();
while (y <= scanConverter.getBottomBoundary()) {
    for (int i=0; i<scanConverter.getNumScans(y); i++) {
        ScanConverter.Scan scan =  scanConverter.getScan(y, i);

        if (scan.isValid()) {
            int offset = (y - viewWindow.getTopOffset()) *
                viewWindow.getWidth() +
                (scan.left - viewWindow.getLeftOffset());

            scanRenderer.render(offset, scan.left,  scan.right);
        }
```

```
            }
            y++;

        }
```

利用这个 SortedScanConverter，可以从前向后绘制多边形，测试视图窗口是否填完。下面建立一个演示程序。

10.7 第一个 BSP 树例子

我们把 BSP 绘制包装到 SimpleBSPRenderer 类中，这是 ShadedSurfacePolygonRenderer 的子类。新类有一个方法，在树中绘制多边形，也是个 BSPTraverseListener：

```
/**
    根据镜头位置在 BSP 树中绘制显示的多边形。从前向后绘制多边形
*/
    public void draw(Graphics2D g, BSPTree tree) {
        currentGraphics2D = g;
        traverser.traverse(tree, camera.getLocation());
    }

    // 来自 BSPTreeTraverseListener 接口
    public boolean visitPolygon(BSPPolygon poly, boolean isBack) {
        draw(currentGraphics2D, poly);
            return !((SortedScanConverter)scanConverter).isFilled();
    }
```

然后在 BSPTest3D 类中生成基本演示程序。BSPTest3D 类与 BSPTest2D 相似，只是用 SimpleBSPRenderer 绘制多边形：

```
public void draw(Graphics2D g) {

    //绘制多边形
    polygonRenderer.startFrame(g);
    ((SimpleBSPRenderer)polygonRenderer).draw(g, bspTree);
    polygonRenderer.endFrame(g);

    super.drawText(g);
}
```

这样就建立了演示程序。最后，可以看看三维空间中的例子平面图。图 10-18 是 BSPTest3D 的屏幕图形。

这个演示程序不错，但有两个问题：

- 可以画妖怪、宝藏之类的三维实体，但它们不能与视景正确合并，因为实体绘图

要利用 z 缓存。

- 在代码中生成多边形很麻烦，需要更简便的方法。

下面要解决这些问题。

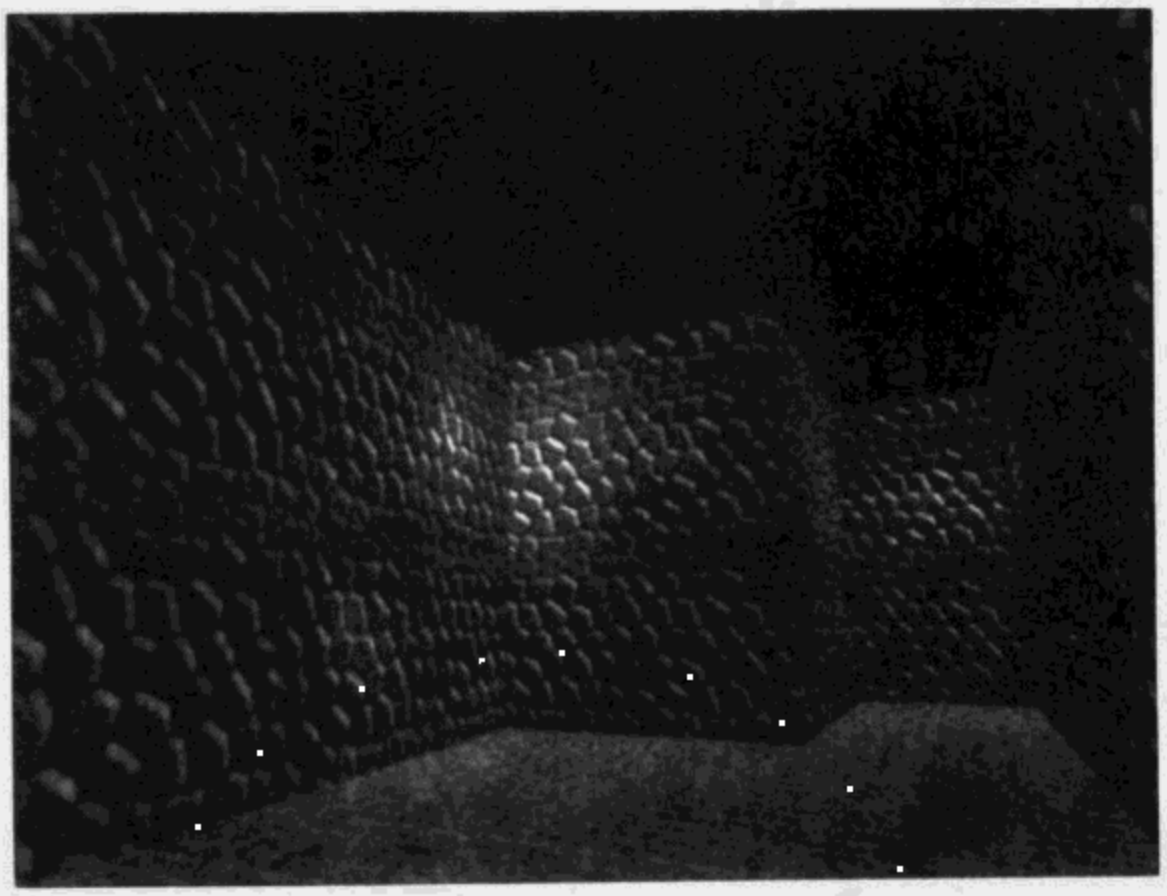


图 10-18　BSPTest3D 的屏幕图形

10.8　在视景中画实体

由于我们已经让 z 缓存绘制三维实体，因此在三维视景中，也可以用 z 缓存绘制三维实体，但要保证多边形在视景中正确绘制。例如，部分遮在墙后的多边形只能显示可见部分。但我们要避免 z 缓存，怎么办？

最简单的方法是照样用 z 缓存，但不是与前面一样的 z 缓存。在 BSP 树中绘制多边形时，视图中每个像素只画一次。因此，对每个像素，不必检查 z 缓存，只要设置 z 缓存即可。换句话说，每个像素只要设置 z 缓存一次，不会像普通 z 缓存一样对性能影响很大。

基本流程如下：

- 在 BSP 树中控制多边形，设置 z 缓存。
- 在三维实体中绘制多边形，检查和设置 z 缓存。

在我的测试机器上，在 BSP 树中设置 z 缓存。占用 3%～10%的处理器能力。尽管有一定性能影响，但可以完美地在视景中合并三维实体。而且，这是三维绘制的常见方法，最初的 Quake 用的正是这种方法。

值得注意的是，可以独立于纹理贴图设置 z 缓存。为此，生成多边形绘制器类 BSPRenderer，对绘制的每个多边形扫描设置 z 缓存如下：

```
/**
    对当前多边形扫描设置 z 缓存深度
*/
```

```
private void setScanDepth(int offset, int width) {
float z = c.getDotProduct(viewPos);
    float dz = c.x;
    int depth = (int)(w*z);
    int dDepth = (int)(w*dz);
    short[] depthBuffer = zBuffer.getArray();
    int endOffset = offset + width;

   // 许多地板和天花板的深度是常量
    if (dDepth == 0) {
        short d = (short)(depth >> SCALE_BITS);
        while (offset < endOffset) {
            depthBuffer[offset++] = d;
        }
    }
    else {
        while (offset < endOffset) {
            depthBuffer[offset++] =
                (short)(depth >> SCALE_BITS);
            depth += dDepth;
        }
    }
}
```

这个 BSPRenderer 类还用普通的检查与设置 z 缓存方法绘制三维实体。

现在就可以绘制三维实体了，最后还要从文件装入贴图。

10.9 从文中装入贴图

通常，一个好的办法是把内容与代码分开。这里，多边形和构成贴图的实体是内容。

上一章常用 Alias|Wavefront OBJ 文件格式定义多边形组。本章对贴图进行类似处理，考虑下列目标：

- 用变高度地板与天花板方便地定义二维贴图；
- 定义点光源和环境光强度；
- 定义三维实体和游戏者的开始位置；
- 方便地编辑贴图；

要达到这些目的，就要生成与 OBJ 格式相似的简单文件格式，称为 MAP 格式和 OBJ 文件格式一样，MAP 也是文本格式，一行一个命令，用#表示说明行。表 10-1 显示了 MAP 格式命令。

表 10-1　MAP 格式命令

格式	说明
mtllib <filename>	从外部.mtl 文件装入材料
usemtl <name>	后续多边形使用指定材料（从外部.mtl 文件装入）
V <x> <y> <z>	用浮点数坐标(x,y,z)定义顶点
ambientLightIntensity<v>	定义下一个房间的环境光强度（0 到 1）
pointlight [v] [intensity] [falloff]	定义指定向量位置的点光源，还可以指定光强度和减弱距离
Room [name]	定义新房间，可以指定房间名。房间包括垂直墙、水平地板和水平天花板。目前不支持凹室，但可以用相邻凸室模拟
Floor [height]	定义当前房间的地面高度，使用当前材料。当前材料为 null 时不生成地面多边形。地面可以在天花板之上，这时生成柱子或块状结构而不是房子
Ceil [height]	定义当前房间的天花板高度，使用当前材料。当前材料为 null 时不生成天花板多边形。地面可以在天花板之上，这时生成柱子或块状结构而不是房子
Wall [x] [z] [bottom] [top]	用指定 x 与 z 坐标定义房中的墙顶点，墙按顺时针方向定义，如果不定义顶和底，则使用地板和天花板高度。如果当前材料为 null 或顶和底相等，则不生成墙多边形
obj [uniqueName] [filename] [v] [angle]	从外部 OBJ 文件定义实体，这个实体可以用惟一名称标识，惟一名称为 null 时表示不必惟一标识。文件名是外部 OBJ 文件，也可以指定沿 y 轴的开始角（弧角）
player [v] [angle]	指定游戏者开始位置，可选指定沿 y 轴的开始角（弧角）

MAP 格式将多边形组织成房间。什么都是房间，如走廊、房间以及里边朝外的柱子与台阶。另外，还可以定义变高度的地板与天花板。

利用 MAP 格式，可以在文本编辑器中生成贴图而不是在代码中手工生成多边形。清单 10.17 的例子贴图生成与本章所用平面图相似的房子。

清单 10.17　basic.map

```
# 装入材料
mtllib textures.mtl

# 定义房子
ambientLightIntensity .5
```

```
room MainRoom

usemtl roof1
floor 0

usemtl roof2
ceil 300

# 定义墙
# 墙定义为这个顶点到下一个顶点
usemtl wall1
wall 0 150
wall 0 450
wall 800 450
wall 800 300
wall 500 300
wall 500 75

# 定义光线
v 400 200 100
pointlight -1 1 300
v 700 200 400
pointlight -1 .5 1000
v 65 200 385
pointlight -1 1 100

# 指定游戏者开始位置
v 400 0 300
player -1
```

这个平面图没有任何实体，但也可以增加一些实体如下：

```
v - 700 0 550
obj bot1 robot.obj - 1 1.57
```

要读取 MAP 文件，可以用 ObjectLoader 的子类 MapLoader,其分析程序用清单 10.18 所示的 RoomDef 类定义房子。

清单 10.18 RoomDef.java

```
package com.brackeen.javagamebook.bsp2D;

import java.util.*;
```

```
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.graphics3D.texture.*;
/**
    RoomDef类表示有墙、地板和天花板的凸房子。地面可以在天花板之上，这时生成柱子或块状
    结构而不是房子。RoomDefs是个快捷键，生成二维BSP树中使用的实际BSPPolygons
*/
public class RoomDef {

    private HorizontalAreaDef floor;
    private HorizontalAreaDef ceil;
    private List vertices;
    private float ambientLightIntensity;

    /**
        HorizontalAreaDef类表示地板或天花板
    */
    private static class HorizontalAreaDef {
        float height;
        Texture texture;
        Rectangle3D textureBounds;

        public HorizontalAreaDef(float height, Texture texture,
            Rectangle3D textureBounds)
        {
            this.height = height;
            this.texture = texture;
            this.textureBounds = textureBounds;
        }
    }
    /**
        Vertex类表示块顶点
    */
    private static class Vertex {
        float x;
        float z;
        float bottom;
        float top;
        Texture texture;
        Rectangle3D textureBounds;
```

```
        public Vertex(float x, float z, float bottom, float top,
            Texture texture, Rectangle3D textureBounds)
        {
            this.x = x;
            this.z = z;
            this.bottom = bottom;
            this.top = top;
            this.texture = texture;
            this.textureBounds = textureBounds;
        }

        public boolean isWall() {
            return (bottom != top) && (texture != null);
        }
    }
}
```

RoomDefs 类跟踪贴图中定义的房子的一切：墙顶点、地板高度、天花板高度和纹理，它还有从房子生成多边形的方法，但这里没有显示这些方法，因为它们没有什么特别之处。这里没有显示 MapLoader 的代码，因为它与 ObjectLoader 相似。

初一看，好象也可以有凹房子，因为 BSP 分区可以把房子分解为凸房子。但 BSP 建立代码无法处理凹房子的复杂细节，如图 10-19 所示。图中分区把凹多边形分解为四个多边形，因此，贴图文件中的房子一定是凸室。

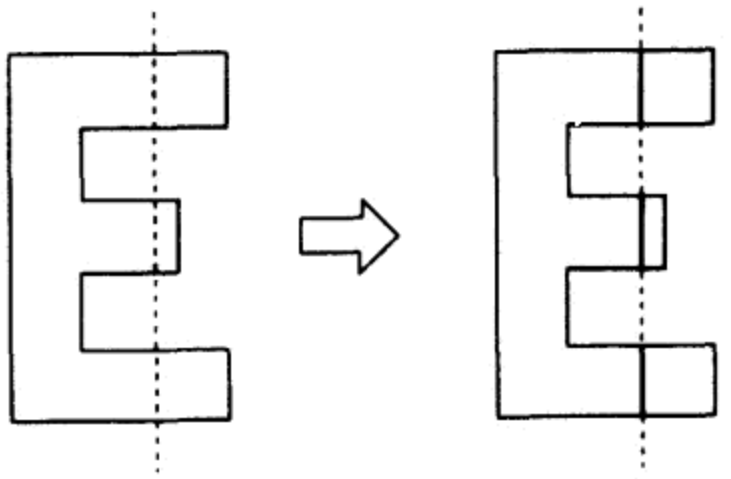


图 10-19　沿直线分割凹多边形可能得到多个多边形，这里生成四个多边形

10.10　综合起来

介绍 BSP 树之后，最后一个演示程序 BSPMapTest 要把所有内容综合起来，从 BSP 树绘制多边形，绘制三维实体多边形，并从文件装入贴图。图 10-20 和图 10-21 显示了 BSPMapTest 的屏幕图形。

这个演示程序并不完美，主要的问题是没有碰撞处理机制：你可以穿墙而过。同样，

还不能上台阶，而只能随时保持相同高度。这个问题将在下一章“碰撞探测”加以解决。

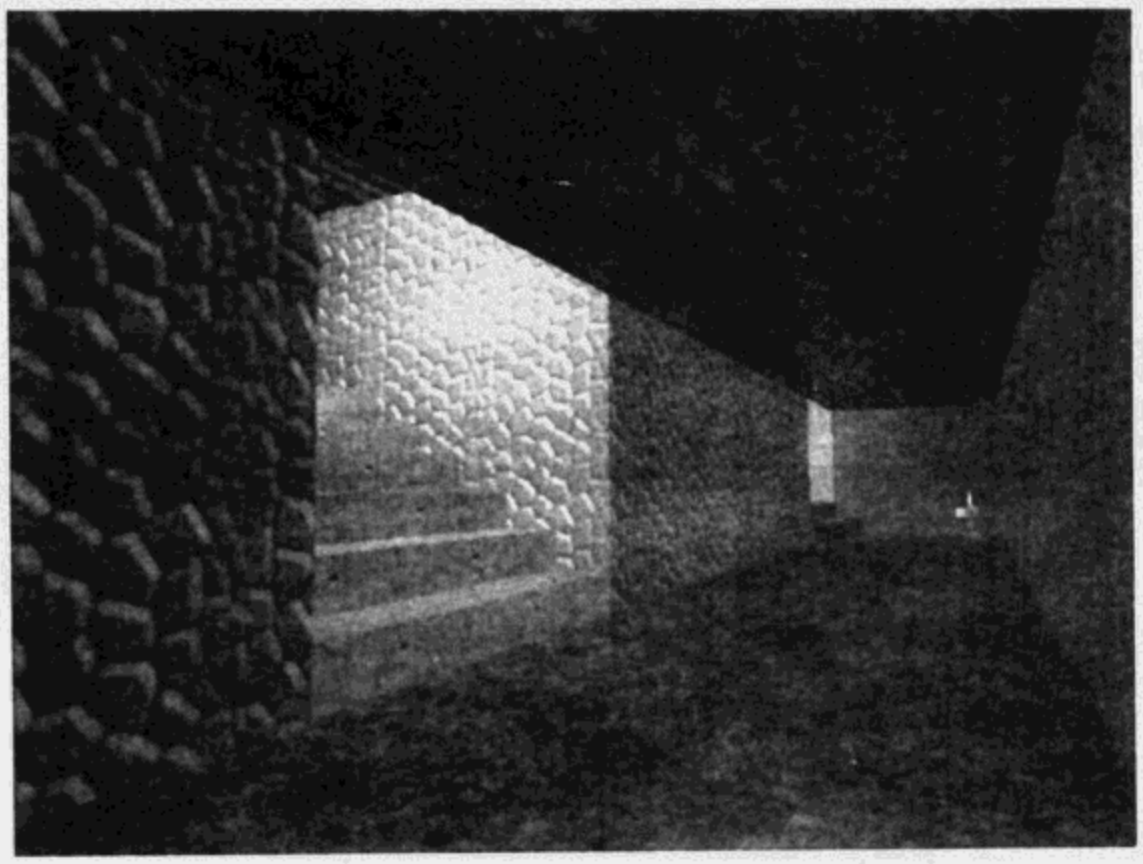


图 10-20　BSPMapTest 的屏幕图形

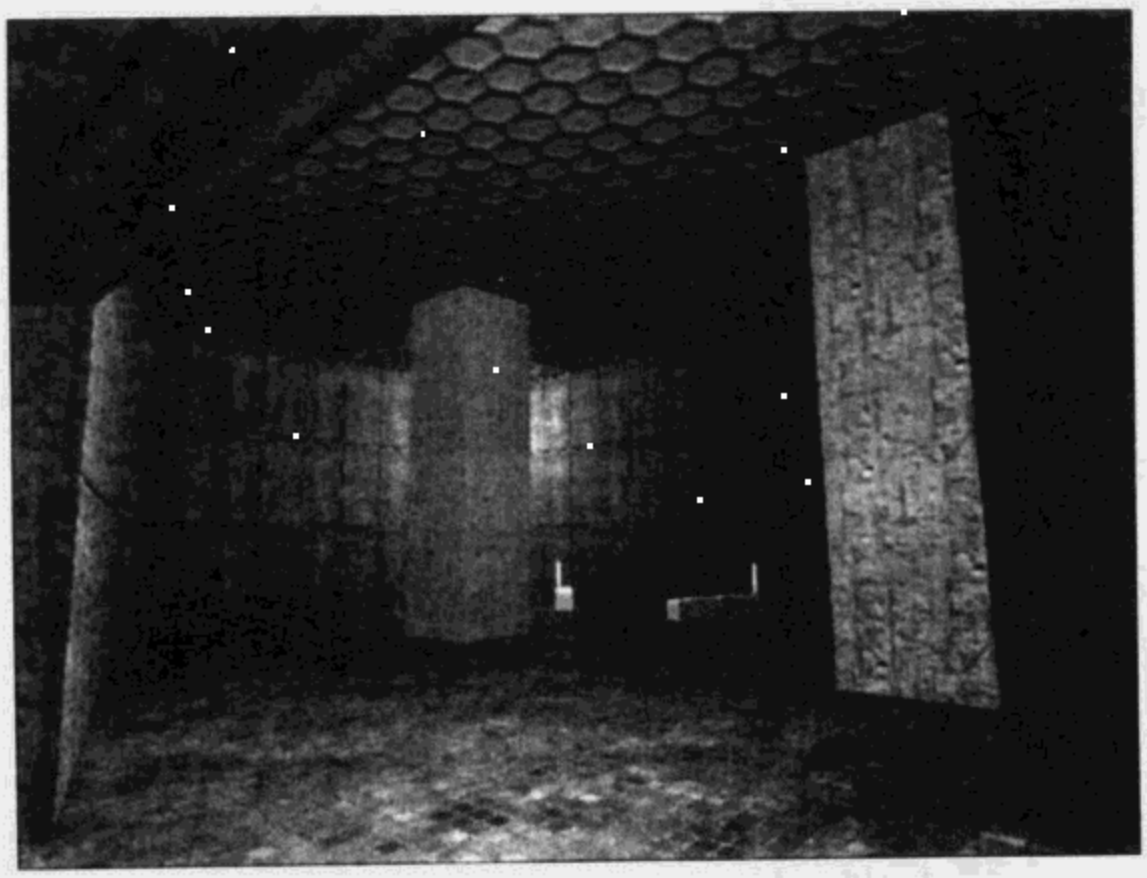


图 10-21　BSPMapTest 的屏幕图形

10.11　改进

本章生成的二维 BSP 树是个精彩的工具，可以生成一些精彩视景，但还有一些可以改进的地方。下面是一些可以改进的方法：

- 建立树时，生成更好的分区选择算法，生成更平衡的树和减少多边形分割。

- 将建立的 BSP 树保存到文件中，以免每次启动时重新计算。
- 用三维 BSP 树而不用二维 BSP 树，以便使用不垂直不水平的多边形，如斜坡和斜天花板。
- 用 z 缓存绘制爆炸之类微粒效果。

10.12 小结

本章介绍了 BSP 树的复杂问题，包括建立二维 BSP 树，从前向后画多边形，绘制三维实体和从文件装入贴图，还介绍了二叉树基础、直线相交、厚线及如何消除 T 形接头接缝。

有了漂亮的三维绘制引擎后，下一步要转入其他一些游戏编程课题，如碰撞探测、路径寻找和人工智能。

第 11 章　碰撞探测

关键主题

- 碰撞基础
- 实体间碰撞
- 实体与世界碰撞
- 基本碰撞探测演示程序
- 滑动与碰撞处理
- 碰撞探测与滑动演示程序
- 改进
- 小结

还记得 Pong 游戏吗？

无论是球跳到拍上、激光射到机器人上、冒险家掉进地沟里、英雄找到弹药、两个妖怪走到一起还是游戏者碰撞到墙壁，几乎任何游戏中都需要某种碰撞探测。

第 5 章“创建二维平台游戏”介绍了一些基本二维碰撞探测，适用于二维地砖平铺游戏。本章要扩展这些概念，提供更彻底的实体间碰撞探测，并对 BSP 树中存放的多边形提供碰撞探测。

同样重要的是，我们要介绍碰撞探测之后的不同处理方法，如沿墙滑动或让游戏实体处理自己的碰撞行为。

11.1　碰撞基础

碰撞探测不仅要探测，还要考虑三个方面：

- 确定测试哪些碰撞。如果两个实体在世界的两头，根本就不必要碰撞探测。如果世界中有 1000 个运动实体，则要测试每个实体与其他实体的碰撞，共有 999000 次测试，因此要尽量限制测试的实体数。一个简单的方法是只测试靠近的实体。
- 碰撞探测。碰撞探测取决于碰撞所要的精度。可以提供完美的碰撞探测，测试一个实体的每个多边形与另一个实体的每个多边形，但涉及的计算量非常大。同样，在二维世界中，可以测试一个幽灵的每个像素与另一个幽灵的每个像素。通常，游戏所用的碰撞探测更粗糙些，但处理速度更快。
- 碰撞处理。如果实体碰上某个东西，则可以根据不同碰撞类型按不同方式处理。例如，抛射弹碰上机器人时，会破坏抛射弹和机器人。实体碰上墙时，可能沿墙滑动，等等。

总之，游戏要实现下列碰撞探测目标：

- 尽量减少碰撞测试。
- 迅速确定是否碰撞。

- 提供对游戏足够精确的碰撞探测。
- 碰撞处理不要影响用户对游戏的注意力。

最后一个目标的意思是，游戏过程中不要过多限制游戏者的运动。例如，游戏者碰墙时，不要让其完全停止，而是让他沿墙滑动或轻轻弹回。

另外，碰撞探测也不能太不精确，以至于游戏者可以在图中某个位置挤过墙去。

下面先介绍一个简单碰撞探测算法。把游戏中移动的一切都看成实体，不管它是妖怪、游戏者、抛射器还是别的东西。对每个实体，步骤如下：

(1) 更新实体位置。

(2) 检查与其他实体或环境的碰撞。

(3) 如果发现碰撞，则让实体退回原位置。

注意：我们在每个实体移动之后检查碰撞，也可以先让所有实体移动，然后再检查碰撞，但这样就要在实体中存储实体原先的位置，如果三个或多个实体发生碰撞，则会遇到问题。

这个基本算法只是在发现碰撞时让实体退回原位置。通常我们要根据碰撞类型使用其他碰撞处理方法。

记住这个算法之后，下面先介绍实体间碰撞。

11.2 实体间碰撞

无论碰撞探测需要怎样的精度，最好都应减少实体间测试，先通过几个其他测试保证两个实体确实很可能发生碰撞时才进行测试。

11.2.1 减少测试

显然，如果实体在帧与帧之间没有移动，则不必对其进行任何碰撞探测。例如，放在那里的棺材不会碰上任何东西。其他实体可能碰上这口棺材，但这些碰撞由其他实体处理。

为了进一步减少测试，可以只测试同一近似范围内的实体。

一种方法是用栅格布置实体，如图 11-1 所示。每个实体在栅格上一个单元中。尽管实体边界可能延伸到其他单元，但实体本身只在一个单元中。这样，实体只要测试本单元和相邻单元上的实体是否发生碰撞。

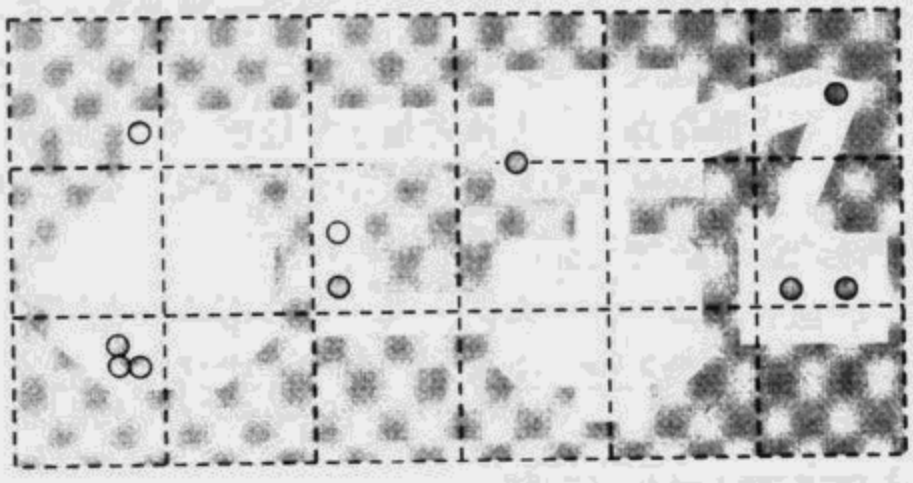


图 11-1 为了减少实体间碰撞，可以用栅格布置实体，只要测试本单元和相邻单元上的实体是否发生碰撞

其他隔离实体的办法包括一维和三维栅格。例如，在横向卷轴游戏中，实体可能在按 x 位置排序的清单中，因此只要对表中的相邻实体测试碰撞。对于三维游戏，可以用三维栅格而不是二维栅格隔离实体，使每个单元为立方体，而不是正方形。本章三维引擎的碰撞探测使用二维栅格。

用栅格隔离实体的好处是很容易进行实体剔除。例如，对自上而下引擎，只要绘制显示单元中的实体。对 BSP 树，只要绘制具有显示叶的单元中的实体。

用栅格布置实体的代码很简单，用本章的 GridGameObjectManager 类实现。更新实体时，调用清单 11.1 所示的 checkObjectCollision()方法。这个方法检查与实体周围单元中任何实体的碰撞。

清单 11.1 检查实体周围单元中的实体碰撞（GridGameObjectManager.java）

```
/**
    Cell 类表示栅格中一个单元，包含一列游戏实体和一个显示标志
*/
    private static class Cell {
        List objects;
        boolean visible;

        Cell() {
            objects = new ArrayList();
            visible = false;
        }
}

...

    /**
        检查指定实体是否与任何其他实体发生碰撞
    */
    public boolean checkObjectCollision(GameObject object,
        Vector3D oldLocation)
    {

        boolean collision = false;

        // 用实体的(x,z)位置（地平面）
        int x = convertMapXtoGridX((int)object.getX());
        int y = convertMapYtoGridY((int)object.getZ());
```

```
        // 检查实体周围九个单元
        for (int i=x-1; i<=x+1; i++) {
            for (int j=y-1; j<=y+1; j++) {
                Cell cell = getCell(i, j);
                if (cell != null) {
                    collision |= collisionDetection.checkObject(
                        object, cell.objects, oldLocation);
                }
            }
        }

        return collision;
    }
```

这个代码调用 CollisionDetection 类的 chectObject()方法，探测一个实体与一列实体间的冲突。下面要实现这个类。

11.2.2 边框球

碰撞探测的一个快速而不够精确的方法是使用图 11-2 所示的边框球（bounding spheres）。

图 11-2 可以用边框球进行碰撞探测

其思路是，如果两个实体的边框球相碰，则认为是这两个实体相碰。测试两个实体的边框球是否相碰的代码如下：

```
dx = objectA.x - objectB.x;
dy = objectA.y - objectB.y;
dz = objectA.z - objectB.z;
minDistance =objectA.radius + objectB.radius
if (Math.sqrt(dx*dx + dy*dy + dz*dz) < minDistance) {
    //发现碰撞
}
```

但是，这个 Math.sqrt()函数调用涉及大量计算，可以对方程两边平方，从而加以简化：

```
if(dx*dx + dy*dy + dz*dz< minDistance*minDistance) {
    //发现碰撞
}
```

如果是二维游戏而不是三维游戏，则可以用圆而不用球进行测试，从方程中去掉 z 坐标。

测试边框球很简单，但不够精确。例如，在图 11-3 中，游戏者的边框球与机器人的边框球碰上，但游戏者与机器人，并没有碰上。

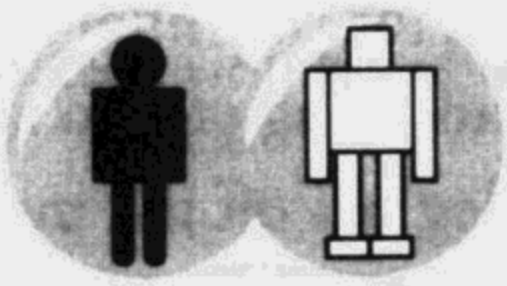


图 11-3　边框球不够精确：球碰上，实体可能没碰上

当然，这点误差在大多数游戏中可以接受。例如，在快速运动动作游戏中，你跑去取食品或弹药时，还没有实际碰上就先取到东西自然没关系。但有些情形中，这种误差就很气人，比如明明一个生物没碰到你，却让你受到伤害。

经过初步边框球测试之后，可以进一步进行第二轮更全面的测试，如检查一个实体的每个多边形与另一个实体的每个多边形。

另一个方法是用另一组球进行更精确的测试，如图 11-4 所示。图中机器人用三个边框球更精确地描述机器人的外形。两个实体边框球测试发现碰撞之后，可以用另一组球进行更精确的测试。如果游戏者的任何第二组球与机器人的任何第二组球发生碰撞，则发生碰撞。

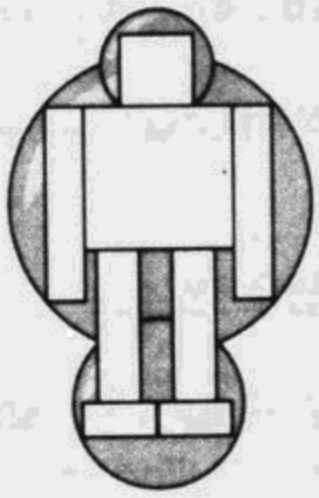


图 11-4　用多个边框球更精确地描述边界测试

这样，每个实体都有两级球。当然，还可以不止两级球测试，可以有更多级，每一级有更多球，从而更加精确。这种方法通常称为球树或球从属。这样可以迅速排除非碰撞实体，更准确地进行碰撞测试。这种方法还可以了解实体哪个部位碰上，从而采取相应动作。例如，如果火箭射中机器人的左脚，则可以让机器人的左脚断掉。

注意：实体转换时，球树要随实体一起转动。例如，机器人的手臂移动时，球要跟着动。在第 9 章“三维对象”的代码中，我们把三维实体定义为嵌套多边形组。由于这已经是树型结构，因此实现球树时可以对每个多边形组指定一组球，保证对球采用这个组的变换。本章并不实际实现球树，读者可以自己思

考这个问题。

总之，对典型帧，大多数实体不用进行碰撞测试，有些实体只要简单碰撞测试，而有些实体则要更复杂、计算量更大的碰撞测试。

11.2.3 边框圆柱

除了边框球外，还有边框圆柱，如图 11-5，可以把碰撞测试简化为二维圆测试和一维垂直测试。

图 11-5 向上边框圆柱可以用于碰撞探测

向上边框圆柱能比球更精确地描述细长的实体（如妖怪和游戏者），本章的三维引擎用向上边框圆柱进行实体间碰撞测试。

本章的所有基本碰撞测试代码都放在 CollisionDetection 类中。清单 11.2 显示了这个类中处理实体间碰撞的方法。

清单 11.2 检查对象（CollisionDetection.java）

```
/**
    检查指定实体是否与指定表中任何其他实体发生碰撞
*/
    public boolean checkObject(GameObject objectA, List objects,
        Vector3D oldLocation)
    {
        boolean collision = false;
        for (int i=0; i<objects.size(); i++) {
            GameObject objectB = (GameObject)objects.get(i);
            collision |= checkObject(objectA, objectB,
                oldLocation);
        }
        return collision;
```

```
}
/**
    如果两个指定实体碰撞，则返回真值。实体 A 是运动实体，实体 B 是要检查的实体。可
    以用向上边框圆柱（上下是圆形）探测碰撞
*/
public boolean checkObject(GameObject objectA,
    GameObject objectB, Vector3D oldLocation)
{
    // 不与自己碰撞
    if (objectA == objectB) {
        return false;
    }

    PolygonGroupBounds boundsA = objectA.getBounds();
    PolygonGroupBounds boundsB = objectB.getBounds();

    // 首先检查 y 轴是否碰撞（假设高度为 pos)
    float Ay1 = objectA.getY() + boundsA.getBottomHeight();
    float Ay2 = objectA.getY() + boundsA.getTopHeight();
    float By1 = objectB.getY() + boundsB.getBottomHeight();
    float By2 = objectB.getY() + boundsB.getTopHeight();
    if (By2 < Ay1 || By1 > Ay2) {
        return false;
    }

    // 然后检查二维 x/z 平面的碰撞（圆底）
    float dx = objectA.getX() - objectB.getX();
    float dz = objectA.getZ() - objectB.getZ();
    float minDist = boundsA.getRadius() + boundsB.getRadius();
    float distSq = dx*dx + dz*dz;
    float minDistSq = minDist * minDist;
    if (distSq < minDistSq) {
        return handleObjectCollision(objectA, objectB, distSq,
            minDistSq, oldLocation);
    }
    return false;
}
/**
    处理实体碰撞。实体 A 是运动实体，实体 B 是要检查的实体。目前碰撞只告知实体 A
```

```
    */
    protected boolean handleObjectCollision(GameObject objectA,
        GameObject objectB, float distSq, float minDistSq,
        Vector3D oldLocation)
    {
        objectA.notifyObjectCollision(objectB);
        return true;
    }
```

在这个代码中，游戏实体的 PolygonGroupBounds 对象用圆柱包围多边形组，边框包括圆柱的半径、底高（通常为 0）和顶高。

checkObject()方法只是检查两个边框圆柱是否发生冲突，如果发生冲突，则调用 handleObjectCollision()方法。

handleObjectCollision()方法只是用 notifyObjectCollision()方法告诉运动实体发生冲突。这类通知方法放在 GameObject 类中，缺省什么也不干，GameObject 的子类可以在需要时覆盖这些通知方法。例如，在 Blast 类中，notifyObjectCollision()方法在发生冲撞时用来杀死 bot：

```
public void notifyObjectCollision(GameObject object) {
    // 删除 Bot 及其本身
    if (object instanceof Bot) {
        setState (object, STATE_DESTROYED);
    }
}
```

目前在 GridGameObjectManager 中，如果实体相互碰撞，则其位置恢复原位。后面可以让实体沿边上滑动，产生更逼真的实体间碰撞效果。

11.2.4 离散时间问题

典型游戏用离散时间段更新游戏状态，例如用上次更新以来经过的时间量更新每个实体。例如，图 11-6 的鸟瞰图显示实体的离散时间运动。运动实体在第 3 帧与大实体相碰。

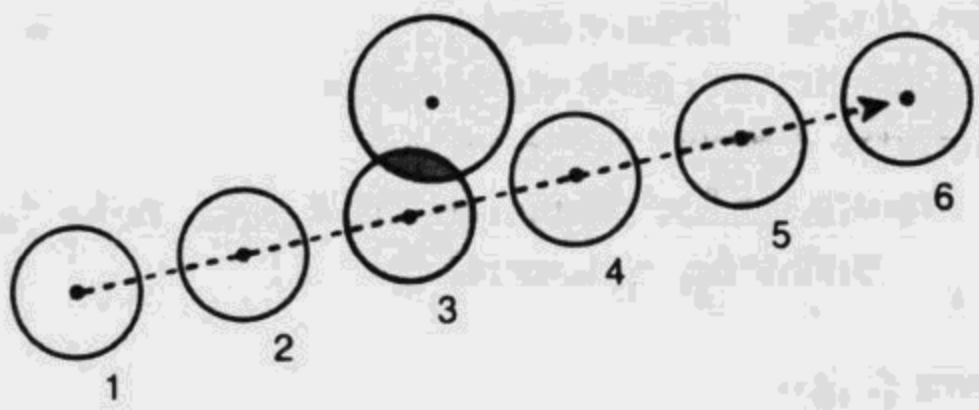


图 11-6 鸟瞰图显示实体的离散时间运动

但是，这个离散时间运动可能在碰撞探测时遇到问题。假设实体移动很快，或帧速率

很慢，则运动实体可能“跳过”相碰的实体。例如，图 11-7 中，运动实体在第 2 帧到第 3 帧之间与大实体相碰。

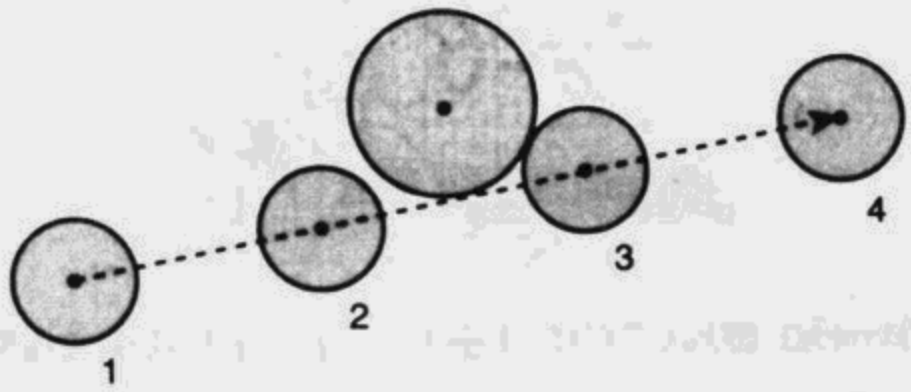


图 11-7　离散时间运动问题，运动实体可能“跳过”相碰的实体

这个问题有两种解决方法。更精确的方法是把运动实体边界看成从开始位置到结束位置的实体形状，如图 11-8，这个方法的计算量很大。

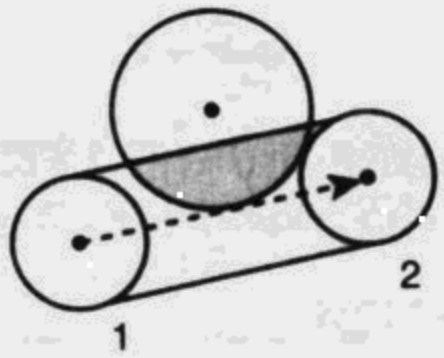


图 11-8　可以把运动实体看成“管道”，解决离散时间运动问题

如果实体移动一定距离，也可以检查从开始位置到结束位置的几个点，看看是否相碰。图 11-7 中，可以在每帧之间再进行一次检查。

11.3　实体与世界碰撞

实体与世界的碰撞通常应尽量准确，不能让游戏者或另一实体穿墙而过或在沿墙移动时发生抖动。

第 5 章实现实体与世界的碰撞时一次只移动一个坐标（先移动 x，后移动 y），适合简单的地砖世界，实体每帧移动的长度不超过一个地砖。

三维世界中没有地砖，但通常用 BSP 树之类结构进行描述，便于进行碰撞探测。我们用上一章的二维 BSP 树实现与墙地板和天花板的碰撞。

11.3.1　测试地板的边框

在第 5 章的二维游戏中，我们对幽灵用边框矩形进行碰撞探测。三维世界中，除了边框球、圆和柱体外，边框是另一个常见的碰撞探测机制。边框有两种：自由边框与轴对齐边框。自由边框可以随意转动和旋转，而轴对齐边框沿 x、y、z 轴对齐。本章用圆柱进行实体间碰撞探测，用轴对齐边框进行实体与世界的碰撞探测。首先要介绍地板与天花板的问题。在变高地板的世界中，实体站在边框下面的最高地板上，如图 11-9。同样，实体不能移到天花板比实体更低的区域。另外，实体还要跨小步而不停止。图中游戏者可以跨小

步走上平台。

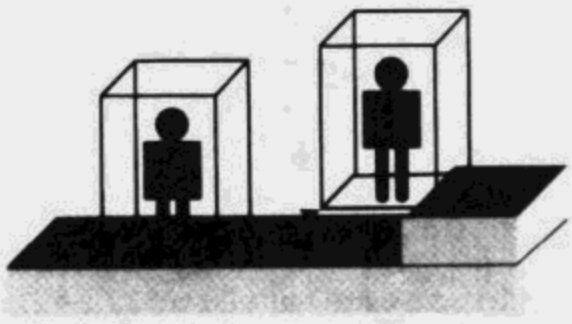


图 11-9 带地板的边框碰撞：游戏者边框有一部分在台阶上，因此台阶高度作为游戏者的“地板”

图 11-9 中，用实体地板下面的最高地板上确定实体站立的高度。用环境中的地板和天花板测试实体时，可以检查框的每个角与地板和天花板是否相交。这涉及四个地板检查，针对边框的四角。需要注意的是，边框也可以用于实体间碰撞探测，甚至可以像球树一样建立边框板。

11.3.2 寻找特定位置的 BSP 叶

对于二维 BSP 树，地板信息存放在树叶中，可以方便地寻找特定位置的 BSP 叶，就像上一章遍历树时一样。清单 11.3 显示了 BSPTree.java 的 getLeaf()方法。

清单 11.3 BSPTree.java 的 getLeaf()方法

```
/**
    取得 x,z 坐标所在的叶节点
*/
public Leaf getLeaf(float x, float z) {
    return getLeaf(root, x, z);
}

protected Leaf getLeaf(Node node, float x, float z) {
    if (node == null || node instanceof Leaf) {
        return (Leaf)node;
    }
    int side = node.partition.getSideThin(x, z);
    if (side == BSPLine.BACK) {
        return getLeaf(node.back, x, z);
    }
    else {
        return getLeaf(node.front, x, z);
    }
}
```

当然，这只是求出一个位置的叶，实体边框可能跨多个叶，因此要检查边框每个角的

叶。

11.3.3 实现地板和天花板高度测试

CollisionDetection 类中的一些方法实现地板和天花板高度测试，见清单 11.4。

清单 11.4 地板和天花板高度测试（CollisionDetection.java）

```
/**
    框住测试 BSP 树相交的游戏对象的四角，四角的顺序可以是顺时针或逆时针
*/
    private static final Point2D.Float[] CORNERS = {
        new Point2D.Float(-1, -1), new Point2D.Float(-1, 1),
        new Point2D.Float(1, 1), new Point2D.Float(1, -1),
    };
...
        getFloorAndCeiling(object);
        checkFloorAndCeiling(object, elapsedTime);
    ...
/**
    取得指定 GameObject 的地板与天花板值。用 object.setFloorHeight()与 object.setCeilHeight()方法设置
    地板与天花板值
*/
    public void getFloorAndCeiling(GameObject object) {
        float x = object.getX();
        float z = object.getZ();
        float r = object.getBounds().getRadius() - 1;
        float floorHeight = Float.MIN_VALUE;
        float ceilHeight = Float.MAX_VALUE;
        BSPTree.Leaf leaf = bspTree.getLeaf(x, z);
        if (leaf != null) {
            floorHeight = leaf.floorHeight;
            ceilHeight = leaf.ceilHeight;
        }

        // 检查周围四点 points
        for (int i=0; i<CORNERS.length; i++) {
            float xOffset = r * CORNERS[i].x;
            float zOffset = r * CORNERS[i].y;
            leaf = bspTree.getLeaf(x + xOffset, z + zOffset);
```

```
            if (leaf != null) {
                floorHeight = Math.max(floorHeight,
                    leaf.floorHeight);
                ceilHeight = Math.min(ceilHeight,
                    leaf.ceilHeight);
            }
        }

        object.setFloorHeight(floorHeight);
        object.setCeilHeight(ceilHeight);
    }
    /**
        检查实体与地板和天花板的碰撞。用 object.getFloorHeight()与 object.getCeilHeight()方法取得地
        板和天花板的值
    */
    protected void checkFloorAndCeiling(GameObject object,
        long elapsedTime)
    {
        boolean collision = false;

        float floorHeight = object.getFloorHeight();
        float ceilHeight = object.getCeilHeight();
        float bottomHeight = object.getBounds().getBottomHeight();
        float topHeight = object.getBounds().getTopHeight();

        if (!object.isFlying()) {
            object.getLocation().y = floorHeight - bottomHeight;
        }
        // 检查是否在地板以下

        if (object.getY() + bottomHeight < floorHeight) {
            object.notifyFloorCollision();
            object.getTransform().getVelocity().y = 0;
            object.getLocation().y = floorHeight - bottomHeight;
        }
        // 检查是否碰上天花板
        else if (object.getY() + topHeight > ceilHeight) {
            object.notifyCeilingCollision();
            object.getTransform().getVelocity().y = 0;
```

```
            object.getLocation().y = ceilHeight - topHeight;
        }
    }
```

getFloorAndCeiling()方法通过检查实体四角的叶节点取得指定 GameObject 的地板与天花板值。记住，实体可能同时在多个 BSP 树叶节点中。

实体 y 坐标在 checkFloorAndCeiling()方法中设置。如果实体不飞，则 y 值为地板高度，否则要检查实体是否碰上地板和天花板。如果碰上，则调用实体的 notifyFloorCollision()或 notifyCeilingCollision()方法。和 notifyObjectCollision()方法一样，这些方法缺省什么也不干，GameObject 的子类可以覆盖这些方法。

实现地板和天花板碰撞的方式目前还很原始，游戏者跳到高层或低层。本章后面要实现重力和平滑登梯的功能，使碰撞显得更逼真。

11.3.4　测试墙的边框

对地板和天花板，只在实体移动之后测试碰撞，只确定游戏者下方的地面有多高。

墙是细线的，因此如果在实体移动后才测试墙的碰撞，则可能错过实体已经穿过的墙。为了精确判断实体是否撞墙，要在一个帧更新到另一个帧期间测试实体经过的整个路径。

对于二维 BSP 树，由于一个帧到另一个帧的实体路径是线段，因此可以测试这个线段是否与表示墙的线相交（与三维 BSP 树中的概念相同，但测试路径与表示墙的平面是否相交）。

当然，实体不是点，而是形体。因此，如果用边框进行碰撞测试，则要用边框的四角测试视景中的墙，如图 11-10 所示。图中测试四个角的四条路径是否相交，其中三条与墙相交。

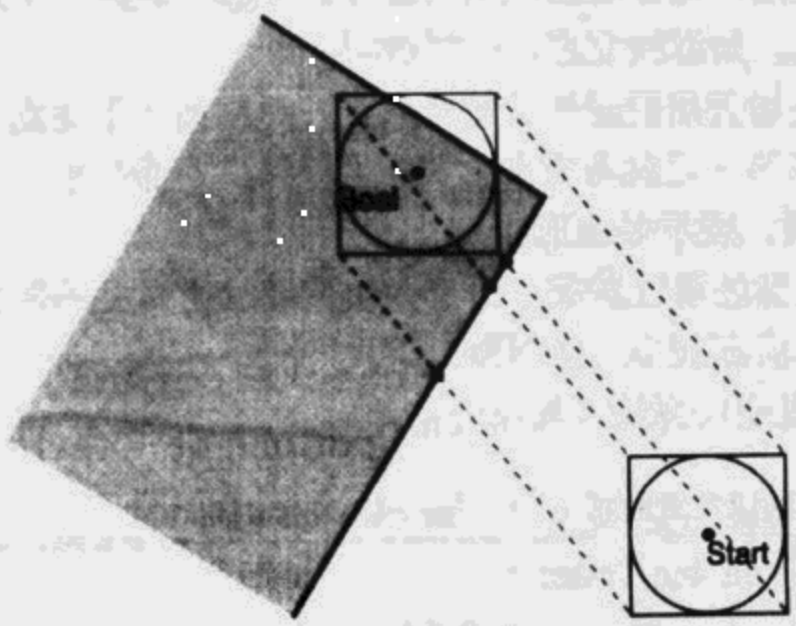


图 11-10　用边框的四角测试视景中的墙并用 BSP 树检查每个线段是否碰撞

如果遇到多个交点，如图 11-10，则碰撞使用从开始位置到交点的最短路径，如图 11-11 所示。这里，实体左上角首先撞墙（是到交点的最短路径），因此用其确定碰撞位置。

现在要用一个简单方法确定路径是否与三维世界中任何多边形相交。这里要求出 BSP 树的第一个线段交点。

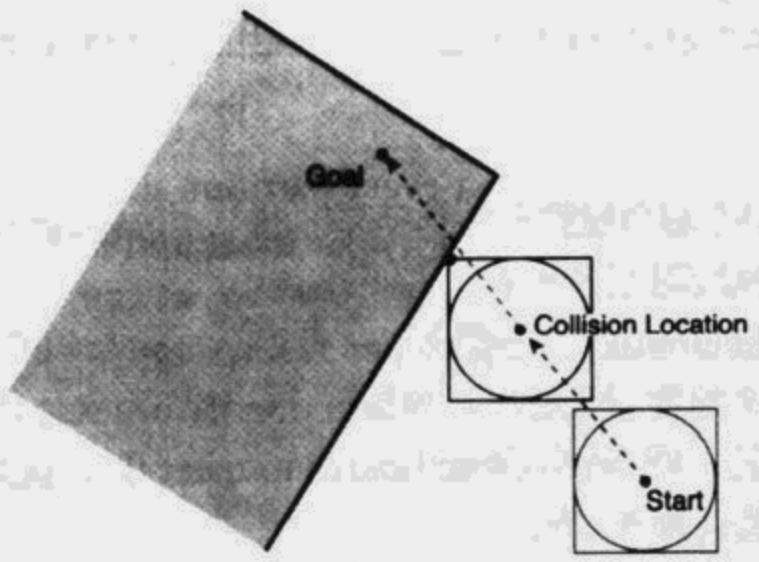


图 11-11 碰撞使用从开始位置到交点的最短路径

11.3.5 BSP 树的线段交点

假设路径从(x1,y1)到（x2,y)，目标是求这个路径与 BSP 树中多边形的第一个交点（如有)。第一个交点是最接近(x1,y1)的交点。

算法如下，首先从 BSP 树的根节点开始：

(1) 检查路径与节点的分区。如果路径完全在节点分区前面或后面，则分别检查前节点或后节点。

(2) 否则，如果路径跨节点分区，则要把路径沿分区处一分为二。一个路径表示路径第一部分，一个路径表示路径第二部分。

- 检查路径第一部分的交点（见第 1 步)。
- 如果找不到交点，则检查这个节点中多边形的交点。
- 如果找不到交点，则检查路径第二部分的交点（见第 1 步)。
- 如果找到交点，则返回交点，否则返回 null。

这个算法由近到远检查路径经过的每个分区，返回第一个交点（如有)。

注意：路径跨分区不一定就有交点。相交要满足三个条件：

- 对二维 BSP 树，表示多边形的线段跨路径。
- 多边形不是实体能跳过的短多边形，实体经过的路上不是太高或太低的多边形。
- 路径从多边形前面到多边形后面。

清单 11.5 显示了求交的算法（及相交条件)。

清单 11.5 BSP 树与线段求交的算法（CollisionDetection.java）

```
    private BSPTree bspTree;
    private BSPLine path = new BSPLine();
    private Point2D.Float intersection = new Point2D.Float();

...

    /**
        取得路径(x1,z1)-> (x2,z2)与 BSP 树中墙的第一个交点（如有）。返回第一个 BSPPolygon
        交点，不相交时返回 null
```

```
*/
public BSPPolygon getFirstWallIntersection(float x1, float z1,
    float x2, float z2, float yBottom, float yTop)
{
    return getFirstWallIntersection(bspTree.getRoot(),
        x1, z1, x2, z2, yBottom, yTop);
}
/**
    从指定节点开始，取得路径(x1,z1)-> (x2,z2)与 BSP 树中墙的第一个交点（如有）。返回第
    一个 BSPPolygon 交点，不相交时返回 null
*/
protected BSPPolygon getFirstWallIntersection(
    BSPTree.Node node, float x1, float z1, float x2, float z2,
    float yBottom, float yTop)
{
    if (node == null || node instanceof BSPTree.Leaf) {
        return null;
    }

    int start = node.partition.getSideThick(x1, z1);
    int end = node.partition.getSideThick(x2, z2);
    float intersectionX;
    float intersectionZ;

    if (end == BSPLine.COLLINEAR) {
        end = start;
    }

    if (start == BSPLine.COLLINEAR) {
        intersectionX = x1;
        intersectionZ = z1;
    }
    else if (start != end) {
        path.setLine(x1, z1, x2, z2);
        node.partition.getIntersectionPoint(path, intersection);
        intersectionX = intersection.x;
        intersectionZ = intersection.y;
    }
    else {
```

```
            intersectionX = x2;
            intersectionZ = z2;
        }

        if (start == BSPLine.COLLINEAR && start == end) {
            return null;
        }

        // 检查第一段线
        if (start != BSPLine.COLLINEAR) {
            BSPPolygon wall = getFirstWallIntersection(
                (start == BSPLine.FRONT)?node.front:node.back,
                x1, z1, intersectionX, intersectionZ,
                yBottom, yTop);
            if (wall != null) {
                return wall;
            }
        }

        // 测试这个边界
        if (start != end || start == BSPLine.COLLINEAR) {
            BSPPolygon wall = getWallCollision(node.polygons,
                    x1, z1, x2, z2, yBottom, yTop);
            if (wall != null) {
                intersection.setLocation(intersectionX,
                    intersectionZ);
                return wall;
            }
        }

        // 检查第一段线
        if (start != end) {
            BSPPolygon wall = getFirstWallIntersection(
                (end == BSPLine.FRONT)?node.front:node.back,
                intersectionX, intersectionZ, x2, z2,
                yBottom, yTop);
            if (wall != null) {
                return wall;
            }
```

```
        }

        // 没找到
        return null;
    }
    /**
        检查指定路径是否与任何共线多边形表相碰。路径与多边形表示的线相交，但多边形不一
        定与路径相交
    */
    protected BSPPolygon getWallCollision(List polygons,
        float x1, float z1, float x2, float z2,
        float yBottom, float yTop)
    {
        path.setLine(x1, z1, x2, z2);
        for (int i=0; i<polygons.size(); i++) {
            BSPPolygon poly = (BSPPolygon)polygons.get(i);
            BSPLine wall = poly.getLine();

            // 检查非墙的情形
            if (wall == null) {
                continue;
            }

            // 检查墙中不垂直的情形（y 轴）
            if (wall.top <= yBottom || wall.bottom > yTop) {
                continue;
            }

            // 检查是否移到墙背面
            if (wall.getSideThin(x2, z2) != BSPLine.BACK) {
                continue;
            }
            // 检查路径是否跨过墙
            int side1 = path.getSideThin(wall.x1, wall.y1);
            int side2 = path.getSideThin(wall.x2, wall.y2);
            if (side1 != side2) {
                return poly;
            }
        }
```

```
        return null;
    }
```

在这个代码中，getFirstWallIntersection()符合上述算法，getWallCollision()方法检查路径与多边形之间相交的条件。

注意：对于三维 BSP 树，还可以用其确定实体下面的地面高度。只要用这个算法检查实体正下方线段的第一个交点，求出游戏者下方最高的多边形。

实现使用 BSP 树的边框碰撞之前，还要考虑一个可能出现的问题。

11.3.6 角问题

边框碰撞可不好办。检查边框每个角与世界的碰撞时，部分世界与边框碰撞，但不是与边框四角碰撞，如图 11-12 所示。

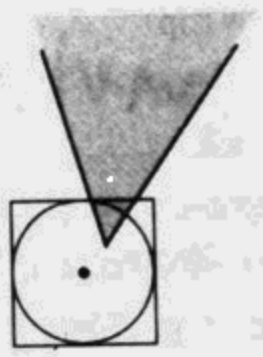


图 11-12 实体与角上相碰时的问题

图中实体与尖角相碰，没有碰上实体的边框角。

要解决这个问题，一种办法是把边框每一边看成一个线段，测试与 BSP 树的相交情况。如果任何一边与 BSP 树中的多边形相交，则实体退回原位。本章的代码中实现这个算法。除了让实体退回原位，也可以一次退一点点，再次测试，直到不会碰上。另一种方法是保证把关卡设计成不会发生这种问题。但是，这就限制了关卡设计，使关卡设计人员很难办，因为很容易出现例子中一样的尖角。

11.3.7 实现实体/世界碰撞探测

这样就完成了用 BSP 树实现实体/世界碰撞探测的工作，清单 11.6 显示了其余代码。

清单 11.6 检查墙（CollisionDetection.java）

```
    // 在 x 或 z 位置改变时检查墙
    if (object.getX() != oldLocation.x ||
        object.getZ() != oldLocation.z)
    {
        chectWalls(object, oldLocation, elapsedTime);
}

...
```

```
/**
    检查游戏实体与 BSP 树中墙的碰撞。返回第一个碰上的墙，不碰时返回 null
*/
public BSPPolygon checkWalls(GameObject object,
    Vector3D oldLocation, long elapsedTime)
{
    Vector3D v = object.getTransform().getVelocity();
    PolygonGroupBounds bounds = object.getBounds();
    float x = object.getX();
    float y = object.getY();
    float z = object.getZ();
    float r = bounds.getRadius();
    float stepSize = 0;
    if (!object.isFlying()) {
        stepSize = BSPPolygon.PASSABLE_WALL_THRESHOLD;
    }
    float bottom = object.getY() + bounds.getBottomHeight() +
        stepSize;
    float top = object.getY() + bounds.getTopHeight();

    // 选择四个角中最近的交点
    BSPPolygon closestWall = null;
    float closestDistSq = Float.MAX_VALUE;
    for (int i=0; i<CORNERS.length; i++) {
        float xOffset = r * CORNERS[i].x;
        float zOffset = r * CORNERS[i].y;
        BSPPolygon wall = getFirstWallIntersection(
            oldLocation.x+xOffset, oldLocation.z+zOffset,
            x+xOffset, z+zOffset, bottom, top);
        if (wall != null) {
            float x2 = intersection.x-xOffset;
            float z2 = intersection.y-zOffset;
            float dx = (x2-oldLocation.x);
            float dz = (z2-oldLocation.z);
            float distSq = dx*dx + dz*dz;
            // 选择距离最近的墙，距离相等时，
            // 如果偏移量与速度符号相同，
            // 则选择当前的墙
            if (distSq < closestDistSq ||
```

```
                (distSq == closestDistSq &&
                MoreMath.sign(xOffset) == MoreMath.sign(v.x) &&
                MoreMath.sign(zOffset) == MoreMath.sign(v.z)))
            {
                closestWall = wall;
                closestDistSq = distSq;
                object.getLocation().setTo(x2, y, z2);
            }
        }
    }

    if (closestWall != null) {
        object.notifyWallCollision();
    }

    // make sure the player bounds is empty
    // (avoid colliding with sharp corners)
    x = object.getX();
    z = object.getZ();
    r-=1;
    for (int i=0; i<CORNERS.length; i++) {
        int next = i+1;
        if (next == CORNERS.length) {
            next = 0;
        }
        // 使用(r-1)，使其不会干扰正常碰撞
        float xOffset1 = r * CORNERS[i].x;
        float zOffset1 = r * CORNERS[i].y;
        float xOffset2 = r * CORNERS[next].x;
        float zOffset2 = r * CORNERS[next].y;

        BSPPolygon wall = getFirstWallIntersection(
            x+xOffset1, z+zOffset1, x+xOffset2, z+zOffset2,
            bottom, top);
        if (wall != null) {
            object.notifyWallCollision();
            object.getLocation().setTo(
                oldLocation.x, object.getY(), oldLocation.z);
            return wall;
```

```
            }
        }
        return closestWall;
    }
```

首先，这个代码测试边框四角的四个路径是否相交。如果找到多个交点，则选择最近的交点。

如果两个交点距离相等，则选择最接近实体速度方向的交点，便于本章稍后实际实现沿墙滑动的功能。

如果找到交点，则把实体位置设置为碰撞位置，靠在墙边。

测试之后，代码要保证实体边框是空的（没有尖角，如图 11-12），如果这样，则实体退回原位。

这样就实现了基本碰撞探测，下面要建立一个演示程序。

11.4　基本碰撞探测演示程序

CollisionTest 演示程序放在本书源代码中，演示了前面介绍的碰撞探测功能，与上一章的 BSPMapTest 演示程序相似，但增加了碰撞探测特性，并使用 GridGameObjectManager。

在这个演示程序中，遇到墙时停止，并可以通过梯子爬到更高的平台上。遇到机器人和棺材之类的实体时也会停止。发射的抛射弹会破坏机器人，为了增加趣味，抛射弹还可以粘到墙上、地上、天花板上和其他实体上，而不是穿过去。

这个演示程序中的碰撞探测特性很不差，但还需要增加一些帮助。

- 碰墙时会“粘住”。在任何三维第一人称或第三人称视角游戏中，这样很容易让人分心，因为游戏者希望沿墙滑动，特别是在墙角移动时。
- 上下楼梯显得有点机械，因为是立即移上移下，而不是平滑地登梯或让重力平滑地把你带下来。
- 沿途不能越过小实体。在地上射一个抛射弹，你居然就走不过去了！
- 不能跳动，当然这不是碰撞探测问题。

但是，本章的目标是良好的碰撞探测，因此我们要解决这些问题。尽管跳动不是碰撞探测问题，但我们也要实现跳动功能，这样就能探测实体何时碰上东西，即使在半空中碰上东西。

11.5　滑动与碰撞处理

下一步要提供非障碍性碰撞处理，即用户希望的碰撞处理行为。

碰上时不应粘住，而应滑动。例如，游戏者碰上一个实体时，不是停止，而是滑到它的旁边。同样，游戏者会沿墙滑动或越过地上的小物体。

本节还要实现一些基本物理特性，使实体可以平滑地上楼梯、采用重力和允许游戏者跳动。

11.5.1 实体间滑动

过去，发生实体间碰撞时，只能让实体恢复原位。这样就使运动实体粘在所碰的静态实体上。

要解决这个问题，一定要保证运动实体在静态实体旁边滑动。一个合理的方法是滑动一定的距离，使两个实体相互避让，即滑动方向由静态实体中心向运动实体中心的向量定义，如图 11-13。图中大实体移动时碰上小实体。然后大实体远离小实体，滑动之后，使两个实体靠在一起而不碰撞。

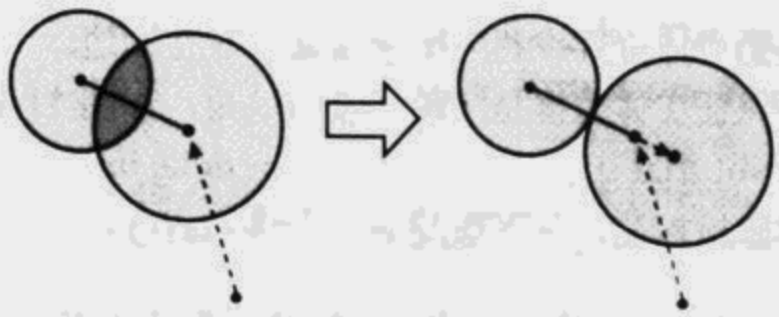


图 11-13 实体间滑动：运动实体被静态实体推开

滑动量是最小距离与实际距离的差：

```
float minDist = objectA.radius + objectB.radius;
float slideDistance = minDist - actualDist;
```

由于由静态实体中心向运动实体中心的向量长度为 actualDist，使公式变成如下：

```
float scale = slideDistance /actualDist;
vector.multiply(scale);
```

距离的平方如下：

```
float scale = (float)Math.sqrt(minDistSq / actualDistSq) - 1;
vector.multiply(scale);
```

尽管平方根函数很慢，但我们只在实体相互碰上时才调用，这种情况并不常有。

如果滑动之后实体滑进墙内或另一实体中，则会遇到问题。这时运动实体可以恢复原位，显示为粘住，游戏者要改变方向。

实体间滑动很简单，所有滑动代码都在 CollisionDetectionWithSliding 类中，这是 CollisionDetection 的子类。清单 11.7 显示了实体间滑动方法。

清单 11.7 实体间滑动（CollisionDetectionWithSliding.java）

```
/**
    处理实体碰撞。实体 A 是运动实体，实体 B 是实体 A 碰上的实体。实体 A 从实体 B 旁边
    滑过或从实体 B 上方越过（如果可能）
*/
protected boolean handleObjectCollision(GameObject objectA,
    GameObject objectB, float distSq, float minDistSq,
    Vector3D oldLocation)
{
```

```
objectA.notifyObjectCollision(objectB);

if (objectA.isFlying()) {
    return true;
}

float stepSize = objectA.getBounds().getTopHeight() / 6;
Vector3D velocity =
    objectA.getTransform().getVelocity();

// 从实体B上方越过（如果可能）
float objectABottom = objectA.getY() +
    objectA.getBounds().getBottomHeight();
float objectBTop = objectB.getY() +
    objectB.getBounds().getTopHeight();
if (objectABottom + stepSize > objectBTop &&
    objectBTop +
    objectA.getBounds().getTopHeight() <
    objectA.getCeilHeight())
{
    objectA.getLocation().y = (objectBTop -
        objectA.getBounds().getBottomHeight());
    if (velocity.y < 0) {
        objectA.setJumping(false);
        // don't let gravity get out of control
        velocity.y = -.01f;
    }
    return false;
}

if (objectA.getX() != oldLocation.x ||
    objectA.getZ() != oldLocation.z)
{
    // 滑到边上
    float slideDistFactor =
        (float)Math.sqrt(minDistSq / distSq) - 1;
    scratch.setTo(objectA.getX(), 0, objectA.getZ());
    scratch.subtract(objectB.getX(), 0, objectB.getZ());
    scratch.multiply(slideDistFactor);
```

```
            objectA.getLocation().add(scratch);

            // 穿墙时恢复原位
            if (super.checkWalls(objectA, oldLocation, 0)
                != null)
            {
                return true;
            }

            return false;
        }

        return true;
    }
```

这个代码中覆盖 handleObjectCollision()方法。首先，它检查运动实体能否从静态实体上方越过。如果不行，则滑到实体旁边。如果滑动会与墙相碰，则实体恢复原位。

怎么实现实体与墙的滑动呢？

11.5.2 实体沿墙的滑动

沿墙滑动好像比较复杂，但其实只要一些简单算术就可以解决。如果知道目标地点（实体在没有墙时要到达的地点），则很容易求出滑动位置，如图 11-14 所示。

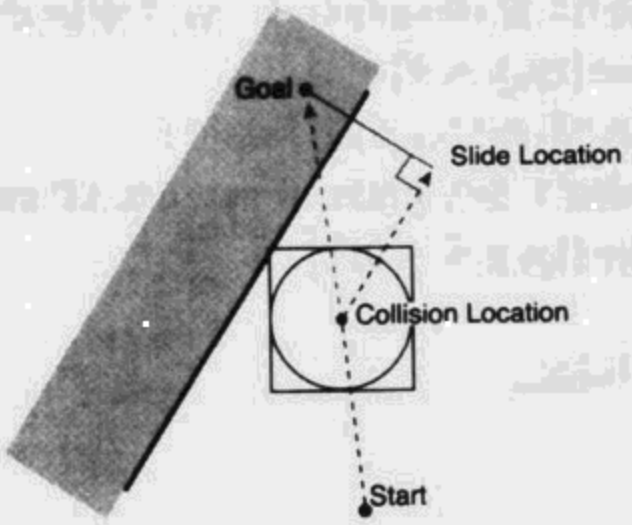


图 11-14　实体沿墙的滑动

图中灰线指向多边形法线方向。如果能求出这条线的长度，则可以求出滑动位置。这是个简单的直角三角形问题。考虑从目标位置到碰撞位置的向量：

```
vector.setTo(actualX, 0, actualZ);
vector.subtract(goalX, 0, goalZ);
```

灰线长度为这个向量与多边形法线的点积：

```
float length = vector.getDotProduct(wall.getNormal());
```

因此，滑动位置如下：

float slideX = goalX + length * wall.getNormal().x;
float slideZ = goalZ + length * wall.getNormal().z;

清单 11.8 显示了实体沿墙的滑动的全部代码。

清单 11.8 实体沿墙的滑动（CollisionDetectionWithSliding.java）

```
    private Vector3D scratch = new Vector3D();
    private Vector3D originalLocation = new Vector3D();


...
    /**
        检查游戏实体与 BSP 树中墙的碰撞。返回第一个碰上的墙，不碰时返回 null。如果有碰撞，
        则实体沿墙滑动，再次检查碰撞。如果滑动会与墙相碰，则实体恢复原位
    */
    public BSPPolygon checkWalls(GameObject object,

        Vector3D oldLocation, long elapsedTime)
    {

        float goalX = object.getX();
        float goalZ = object.getZ();

        BSPPolygon wall = super.checkWalls(object,
            oldLocation, elapsedTime);
        // 发现碰撞，而实体不停止
        if (wall != null && object.getTransform().isMoving()) {
            float actualX = object.getX();
            float actualZ = object.getZ();

            // 墙的法线与到目标的线的点积
            scratch.setTo(actualX, 0, actualZ);
            scratch.subtract(goalX, 0, goalZ);
            float length = scratch.getDotProduct(wall.getNormal());

            float slideX = goalX + length * wall.getNormal().x;
            float slideZ = goalZ + length * wall.getNormal().z;

            object.getLocation().setTo(
                slideX, object.getY(), slideZ);
            originalLocation.setTo(oldLocation);
```

```
        oldLocation.setTo(actualX, oldLocation.y, actualZ);

        // 用较小的半径滑动
        PolygonGroupBounds bounds = object.getBounds();
        float originalRadius = bounds.getRadius();
        bounds.setRadius(originalRadius-1);

        // 检查与滑动位置的碰撞
        BSPPolygon wall2 = super.checkWalls(object,
            oldLocation, elapsedTime);

        // 恢复改变的参数
        oldLocation.setTo(originalLocation);
        bounds.setRadius(originalRadius);

        if (wall2 != null) {
            object.getLocation().setTo(
                actualX, object.getY(), actualZ);
            return wall2;
        }
    }

    return wall;
}
```

这个代码覆盖 CollisionDetection 的 checkWalls()方法。首先采用滑动，然后检查滑动是否碰墙。如果滑动会与墙相碰，则实体恢复原位。

可以关闭实体沿墙滑动。例如，抛射弹覆盖通知方法，使抛射弹碰到墙、地和天花板时停止：

```
public void notifyWallCollision() {
    transform.getVelocity().setTo(0,0,0);
}
public void notifyFloorCollision() {
    transform.getVelocity().setTo(0,0,0);
}

public void notifyCeilingCollision() {
    transform.getVelocity().setTo(0,0,0);
}
```

这样就使抛射弹粘在墙上或其他实体上，可以模仿前面的碰撞处理方法。

实体沿墙滑动之后，下面要爬楼梯了（加点重力）。

11.5.3 重力与爬楼梯（实体沿地面滑动）

另一个常见的滑动效果是游戏者和其他实体平滑地上楼梯，否则像第一个碰撞探测演示程序中一样，游戏者上楼梯时会抖动，因为实体的 y 坐标突然变成上一层楼梯的值。

从高处向下移动时，则会立即掉到下一层，而不是逐渐下降。

重力的用法和第 5 章相似，随着时间增加而使实体向下速度加速。如果实体 y 坐标大于地面 y 坐标，则可以对实体采用重力。

爬楼梯时也一样：如果实体 y 坐标小于地面 y 坐标，则可以对实体采用爬上的加速。

清单 11.9 的 Physics 类总结了游戏使用的物理公式。这只是个开始，后面还要增加更多特性。

清单 11.9 重力与爬力（Physics.java）

```
/**
    缺省重力，单位/平方毫秒
*/
public static final float DEFAULT_GRAVITY_ACCEL = -.002f;

/**
    缺省爬力（上楼梯的加速度），单位/平方毫秒
*/
public static final float DEFAULT_SCOOT_ACCEL = .006f;

private float gravityAccel;
private float scootAccel;
private Vector3D velocity = new Vector3D();

/**
    根据经过的时间量对指定 GameObject 采用重力
*/
public void applyGravity(GameObject object, long elapsedTime) {
    velocity.setTo(0, gravityAccel * elapsedTime, 0);
    object.getTransform().addVelocity(velocity);
}

/**
    根据经过的时间量对指定 GameObject 采用爬力
*/
public void scootUp(GameObject object, long elapsedTime) {
```

```
        velocity.setTo(0, scootAccel * elapsedTime, 0);
        object.getTransform().addVelocity(velocity);
    }
```

对不同情形采用重力与爬力时，需要更复杂的步骤。例如，重力不用于飞行实体。另外，游戏者超过地面而实体不跳动时，要停止爬高加速（稍后将介绍跳动）。

清单 11.10 的代码包括了各种情形。

清单 11.10　检查地板与天花板（CollisionDetectionWithSliding.java）

```
    /**
        检查地板与天花板是否与实体发生碰撞，地板与天花板的值用 object.getFloorHeight() 与
        object.getCeilHeight()取得。如果实体 y 坐标大于地面 y 坐标，则可以对实体采用重力。如
        果实体 y 坐标小于地面 y 坐标，则可以对实体采用爬上的加速。（以便平滑地爬楼梯）
    */
    protected void checkFloorAndCeiling(GameObject object,
        long elapsedTime)
    {
        float floorHeight = object.getFloorHeight();
        float ceilHeight = object.getCeilHeight();
        float bottomHeight = object.getBounds().getBottomHeight();
        float topHeight = object.getBounds().getTopHeight();
        Vector3D v = object.getTransform().getVelocity();
        Physics physics = Physics.getInstance();

        // 检查是否在地板上
        if (object.getY() + bottomHeight == floorHeight) {
            if (v.y < 0) {
                v.y = 0;
            }
        }
        // 检查是否在地板下
        else if (object.getY() + bottomHeight < floorHeight) {

            if (!object.isFlying()) {
                // 如果掉下
                if (v.y < 0) {
                    object.notifyFloorCollision();
                    v.y = 0;
                    object.getLocation().y =
```

```
                    floorHeight - bottomHeight;
            }
            else if (!object.isJumping()) {
                physics.scootUp(object, elapsedTime);
            }
        }
        else {
            object.notifyFloorCollision();
            v.y = 0;
            object.getLocation().y =
                floorHeight - bottomHeight;
        }
    }
    // 检查是否碰上天花板
    else if (object.getY() + topHeight > ceilHeight) {
        object.notifyCeilingCollision();
        if (v.y > 0) {
            v.y = 0;
        }
        object.getLocation().y = ceilHeight - topHeight;
        if (!object.isFlying()) {
            physics.applyGravity(object, elapsedTime);
        }
    }
    // 地板上方
    else {
        if (!object.isFlying()) {
            // 爬上时，停止往上爬
            if (v.y > 0 && !object.isJumping()) {
                v.y = 0;
                object.getLocation().y =
                    floorHeight - bottomHeight;
            }
            else {
                physics.applyGravity(object, elapsedTime);
            }
        }
    }
}
```

这个代码覆盖 CollisionDetection 类中的 checkFloorAndCeiling()方法，剩下的事情就要实现跳动了。

11.5.4 实现跳动

实现跳动的方法与第 5 章相似，只要对实体采用向上的速度。随着重力的采用，这个初始速度的大小减少。

但有时要保证实体所跳的高度，图 11-15 显示了所用的方程。

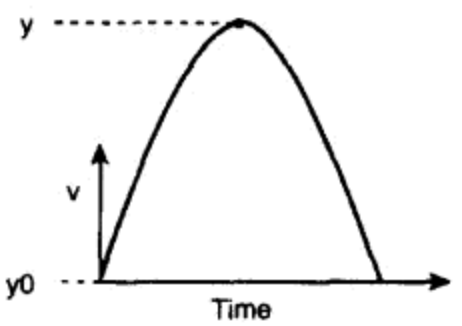


图 11-15 保证实体所跳的高度

图 11-15 中，根据跳动高度(y-y0)和重力加速度（a）求出跳动速度(v)。这是许多物理课本中都有的标准速度/加速度公式。

这样就可以在 Physics 类中增加几个实现跳动的方法，见清单 11.11。

清单 11.11 实现跳动的方法(Physics.java)

```
    /**
        设置指定 GameObject 跳到指定高度的垂直速度。用 getJumpVelocity()方法计算速度，它
        使用 Math.sqrt()函数
    */
    public void jumpToHeight(GameObject object, float jumpHeight) {
        jump(object, getJumpVelocity(jumpHeight));
    }
    /**
        将指定 GameObject 的垂直速度设置为指定跳动速度
    */
    public void jump(GameObject object, float jumpVelocity) {
        velocity.setTo(0, jumpVelocity, 0);
        object.getTransform().getVelocity().y = 0;
        object.getTransform().addVelocity(velocity);
    }
    /**
        根据当前重力返回指定 GameObject 跳到指定高度的垂直速度。使用 Math.sqrt()函数
    */
    public float getJumpVelocity(float jumpHeight) {
```

```
        // 使用速度/加速度公式: v*v = -2 * a(y-y0)
        //(v 是跳动速度, a 是重力加速度, y-y0 是最大高度)
        return (float)Math.sqrt(-2*gravityAccel*jumpHeight);
    }
```

这些方法可以求出一定重力条件下指定 GameObject 跳到指定高度的垂直速度，实际让实体跳动。

11.6　碰撞探测与滑动演示程序

介绍碰撞探测之后，下面把本章的源代码总结到 CollisionTestWithSliding 类中，它与上一个演示程序差不多，但实现滑动。总的说来，它演示了沿墙滑动、沿实体滑动、越过实体、移到实体上、爬楼梯、采用重力和跳动功能。

作为引擎的一个副作用，还可以站在抛射弹上面。由于抛射弹粘在墙上，因此可以发射足够的抛射弹，在墙上“画”出其斜坡，然后走到斜坡上！

当然，这只是我们实现的抛射弹。通常，抛射弹碰到某个东西时会破坏，而不是粘在上面。

11.7　改进

本章介绍了碰撞探测与处理的基础，但还有一些可以改进之处。下面是一些改进的方法：

- 实现球树，得到更精确的实体间测试。
- 检查实体在帧之间是否经过了足够的距离，开始位置与结束位置之间可能碰撞。
- 让游戏者挤在一起。

可以实现摇头，即在游戏者移动时上下移动镜头。

11.8　小结

碰撞探测是二维和三维游戏中必不可少的功能。

本章首先介绍了用栅格隔离实体，限制所要的实际碰撞探测次数，然后介绍了各种碰撞探测机制，如边框球、球树、边框圆柱和边框。我们实现与 BSP 树的墙、地、天花板和其他实体的碰撞探测，我们还实现更好的碰撞处理，使游戏者和其他实体可以沿墙滑动、沿其他实体滑动和爬楼梯，最后还增加重力与跳动特性。

你可以把机器人炸成一堆废铁，但这没什么挑战，因此下面要给敌人增加一些人工智能，使游戏更有趣，更精彩。

第 12 章　路径搜索

关键主题

- 路径寻找基础
- 初步路径寻找方法
- A*算法基础
- 采用 A*算法
- BSP 树与 A*算法
- 一般性路径寻找
- 建立 PathBot
- 改进 A*搜索
- 小结

当你正在街上漫步，考虑自己的业务问题时，突然冲出一帮武士袭击你，你打倒了其中几个人，然后夺路而走，钻进小胡同里，这时再次遇到一帮武士，不管怎么躲，总是躲不开。你遇到过这种事吗？有人遇到过这种事吗？

你很奇怪，这帮武士怎么总能找到你，他们到底要干什么，本书作者是否神经有问题。

本章要解决你的第一个问题，这帮武士怎么总能找到你。具体地说，我们要介绍一些路径寻找算法，实现一般性 A*搜索功能。这个搜索算法使武士总能找到你。

好的路径寻找算法是人工智能的关键，因此我们要在第 13 章介绍人工智能时使用本章建立的结果。

12.1　路径寻找基础

路径寻找可以简化为这样的问题：怎么从 A 点到 B 点。一般来说，从武士（或 Bot）到另一位置的路径可能有多条，但理想路径方案应达到下列目标：

(1) 如何从 A 到 B。

(2) 如何越过沿途障碍。

(3) 如何找到最短路径。

(4) 如何迅速找到路径。

有些路径寻找算法根本没有解决上述问题，而有些则解决了所有上述问题，有些时候任何算法都无法解决这些问题。例如，A 点可能是与 B 点隔离的弧岛，因此根本没有解。

12.2　初步路径寻找方法

如果环境是没有障碍物的平面，则路径寻找非常简单：Bot 只要沿直线到达目标。但 A 点与 B 点之间存在障碍物时，则要复杂一些。例如，图 12-1 中，如果 Bot 走直线，则会撞

墙，粘在墙上。

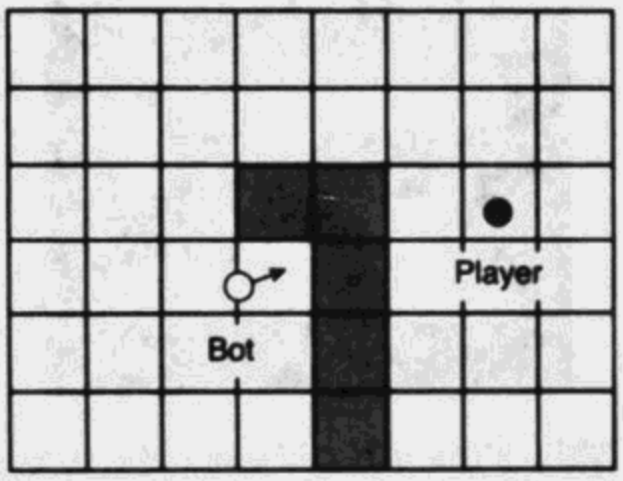


图 12-1　不是最佳方案：Bot（白圆圈）沿直线走向游戏者（黑圆圈），但中间有障碍物

这种行为适合某些游戏，其不关心 Bot 的行为，只关心 Bot 的样子。在游戏者视角中的三维世界中，好像 Bot 只是等在墙后面，等游戏者来送死，而不是粘在墙上。

鸟瞰视角则不同。鸟瞰视角游戏需要有路径寻找功能，如实时战略游戏和 Sims 之类的游戏。你可能要让一队战士攻击某个目标，但他们无法攻击，因为目标在墙的另一头。这时要使用门。

另外，增加 Bot 智能，使其能找到绕过障碍物的路径可以使游戏者感到更有趣，更难哄骗 Bot 做某件事或走某条路。

可以用几种方法绕过障碍物。随机算法最终总能找到一个解（只要有解），这个方法让 Bot 随机绕行，如图 12-2 所示，直到找出所要路径。

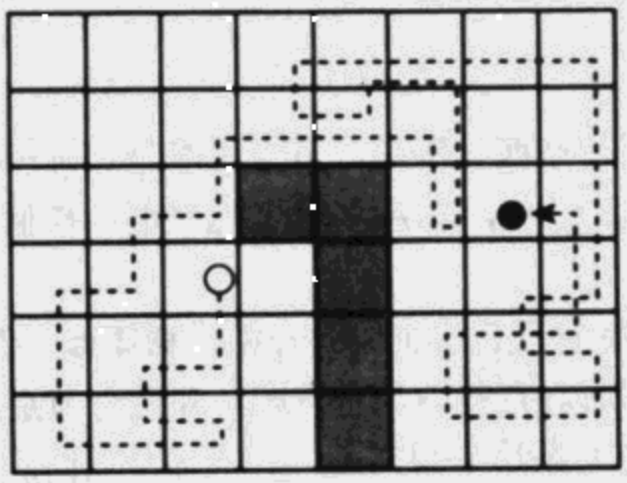


图 12-2　Bot（白圆圈）随机绕行，直到找到游戏者（黑圆圈）

当然，随机算法得不到最佳结果和最短路径。谁愿意看到 Bot 像神经病一样随机绕行呢？

可以将随机算法修改成只是偶尔随机绕行，而大多数情况下直接朝游戏者走，使 Bot 的运动显得更有智能。

另一种解法可以在简单迷宫环境中没有循环时求解，如图 12-3 所示。这里的路径寻找算法是沿墙向右法（或沿墙向左法）。只要在迷宫中坚持沿墙向右走，最终总能找到目的地。

如果 Bot 采用沿墙向右法，则很难找到从 A 到 B 的最短路径。也可以在 Bot 移动之前事先计算路径，然后从路径中删除回溯（例如图中进入右下角大厅的路径），只留下最短路

径解。

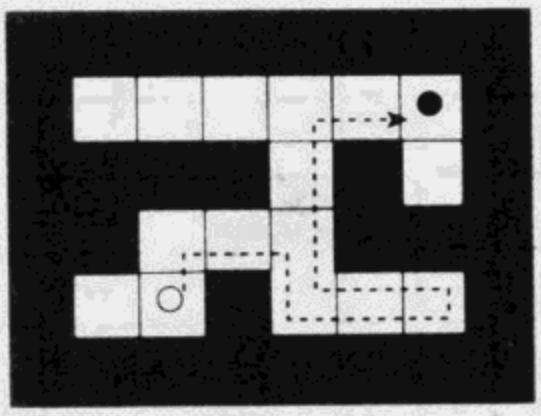


图 12-3　Bot（白圆圈）在迷宫中坚持沿墙向右走直到找到游戏者（黑圆圈）

但是，这个算法只适用于简单迷宫环境中没有循环时。如果路径中有循环，则 Bot 会一直打转转。另外，如果房间很大，则 Bot 永远找不到房子中央的目标，因为 Bot 总是沿墙走。最好能够生成比这更复杂的环境。

如果不把环境看成栅格，也可以把它看成图，如图 12-4 所示。图是一组节点，按不同边组合起来。

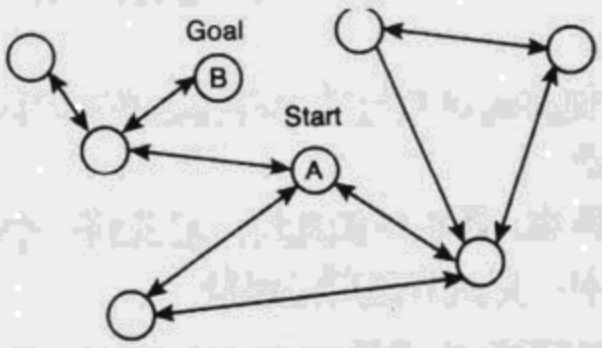


图 12-4　简单图

图中每个节点可以是任何东西。例如，节点可能是二维栅格中的单元，也可能是一个城市，图的边表示高速公路。记住，求 A 到 B 的路径时，任何节点均可作为开始节点或目标节点。

图与第 10 章“用 BSP 树管理三维视景”介绍的树相似，但每个节点不只两个子节点，而可以有无穷个子节点或邻居。有些图是有向同，即两个节点之间的边只能沿一个方向遍历。图 12-4 的例子中，只有一条边是有向的，其余都是双向的。单向边适用于要跳下去的悬崖太高、无法跳回的情形。

本例中，我们要求 A 到 B 的最短路径，或到达目标路径的最少边数。图中的解很容易确定，但计算机程序中怎么求解呢？一个简单方法是采用广度优先搜索法。

广度优先搜索法

和遍历 BSP 树一样，广度优先搜索法一次访问一个节点。广度优先搜索法访问节点时的顺序按离开始节点的距离，即遍历的边数。因此，对于广度优先搜索法，首先访问相距一条边的所有节点，然后访问相距两条边的所有节点，直到访问所有节点。这样就可以求出从开始点到达目标经过最少边数的路径。另一种方法是：访问邻居节点，然后访问邻居的邻居节点，直到访问所有节点。图 12-5 显示了广度优先搜索法的例子，节点按访问顺序

进行编号。

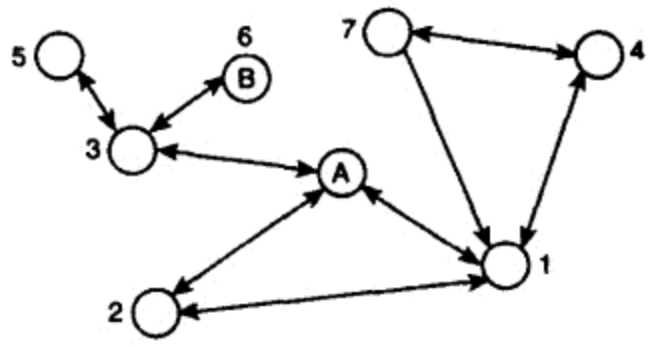


图 12-5 广度优先搜索法的例子，节点按访问顺序进行编号

本章稍后要使用 A*搜索算法，它与广度优先搜索法有几个相似之处。要了解 A*，首先要实现更简单的广度优先搜索法。

首先建立基本节点，其引用所有邻居：

```
public class Node {
    List neighbors;
    Node pathParent;
}
```

邻居表就是所有邻居节点构成的表。

pathParent 节点只用于搜索。可以把从开始节点到目标节点的路径看成树，每个节点只有一个子节点。pathParent 节点是路径树中节点的父节点。找到目标时，可以从图标节点到开始节点遍历路径树，求出路径如下：

```
private List constructPath(Node node) {
    LinkedList path = new LinkedList();
    while (node.pathParent != null) {
        path.addFirst(node);
        node = node.pathParent;
    }
    return path;
}
```

当然，这里假设开始节点在路径中没有父节点。

这样就实现了广度优先搜索法。记住，每个节点只能访问一次。例如，如果 A 有邻居 B 与 C，而 B 有邻居 A，则不能再次访问 A，否则会出现死循环。

因此，我们要跟踪所有已经访问的节点，将其放在“关闭”表中。如果搜索中显示的节点已经在关闭表中，则将其忽略。同样，我们要跟踪要访问的所有节点，放在“打开”表中。打开表是个先进先出（FIFO）表，从起点终点的最少边数到最多边数进行排序。完整代码如下：

```
public List search(Node startNode, Node goalNode) {
// 已访问节点表
LinkedList closedList = new LinkedList();
// 待访问节点表（排序）
```

```
LinkedList openList = new LinkedList();
openList.add(startNode);
startNode.pathParent = null;
while (!openList.isEmpty()) {
Node node = (Node)openList.removeFirst();
if (node == goalNode) {
// 找到路径!
return constructPath(goalNode);
}
else {
closedList.add(node);
// 将邻居加进打开表中
Iterator i = node.neighbors.iterator();
while (i.hasNext()) {
Node neighborNode = (Node)i.next();
if (!closedList.contains(neighborNode) &&
!openList.contains(neighborNode))
{
neighborNode.pathParent = node;
openList.add(neighborNode);
}
}
}
}
// 未找到路径
return null;
}
```

这个函数返回表示路径的节点表，不包括开始节点。如果找不到路径，则返回 null。

这样就完成了广度优先搜索法，回头看看，很容易看到这个算法有一个问题：我们找到了最少边数的路径，但每边的成本可能不同。例如，一个边的成本是 10km，而另一个边的成本是 100km。显然，遍历两个 10km 的边要比遍历一个 100km 的边更快。广度优先搜索法假设各边成本相同，这不一定成立，因此要用到 A*算法。

12.3 A*算法基础

A*搜索与广度优先搜索法相似，但增加两个因素：

- 边的成本，就是从一个节点移到另一节点的成本。估计从任何节点到达目标节点的成本，从而改进搜索，避免在错误方向上搜索。
- 边的成本不一定是距离，也可能是时间，要求所需时间最短的路径。例如，开车

时，抄近路可能距离更短，但走高速公路可能时间更快（洛杉矶的高速公路除外）。另一个例子是地形：穿越森林可能比走草地更花时间。

成本还可以是其他指标。例如，如果 Bot 要偷袭游戏者，则让 Bot 在游戏者前面出现的成本高，而让 Bot 在游戏者后面出现的成本或少没有成本。还可以对某些节点指定特殊的战术优势，例如设在棺材后面的成本比在游戏者前面出现的成本低。

A*搜索与广度优先搜索法相似，但增加两个特点：

- 打开表中的节点按起点到终点的总成本排序，是个优先级队列。总成本是起点成本与终点估计成本之和。
- 关闭表中的节点可以在找到该节点的更短路径（更低成本）时移回打开表。

由于打开表中的节点按估计成本排序，因此这个算法先检查估计成本最小的节点，因此其搜索的节点最可能在到达目标的方向上。因此，估计越佳，搜索越快。

当然，我们要定义成本和估计成本。如果成本是距离，则很容易：节点间的成本是距离，一个节点到目标的成本就是从这个节点到目标的距离。

注意：这个算法要求估计成本不能高于实际成本。如果估计成本高于实际成本，则找到的路径不一定是最短路径。

下面要编写一些代码。首先生成一般性的搜索 A*搜索算法，可以用于各种 A*搜索。不同情形都可以使用这个一般性抽象类，一般性也使代码更可读。首先建立 A*节点，见清单 12.1。

清单 12.1　AStarNode.java

```
package com.brackeen.javagamebook.patli;

import java.util.List;
import com.brackeen.javagamebook.util.MoreMath;
/**
    AstarNode 类和 AstarSearch 类实现一般性 A*搜索算法。AstarNode 类的子类提供搜索功能
*/
public abstract class AStarNode implements Comparable {

    AStarNode pathParent;
    float costFromStart;
    float estimatedCostToGoal;

    public float getCost() {
        return costFromStart + estimatedCostToGoal;
    }

    public int compareTo(Object other) {
        float otherValue = ((AStarNode)other).getCost();
```

```
        float thisValue = this.getCost();

        return MoreMath.sign(thisValue - otherValue);
    }
    /**
        取得这个节点与指定邻居（或子）节点间的成本
    */
    public abstract float getCost(AStarNode node);

    /**
        取得这个节点与指定节点间的估计成本。估计成本不能大于实际成本。估计越佳，搜索越
        快
    */
    public abstract float getEstimatedCost(AStarNode node);

    /**
        取得这个节点的邻居（或子）节点
    */
    public abstract List getNeighbors();
}
```

AstarNode 类是个抽象类，要在子类中提供搜索功能。和广度优先搜索法使用的节点一样，但只包含搜索过程中使用的 pathParent 节点。CostFromStart 与 estimatedCostToGoal 也在搜索期间填入，因为它们随开始节点与目标节点的位置而变化。

getCost(node) 抽象方法取得这个节点与指定邻居（或子）节点间的成本。getEstimatedCost(node)抽象方法取得这个节点与指定节点间的估计成本。估计成本不能大于实际成本。估计越佳，搜索越快。记住，要根据成本定义生成这些函数。

清单 12.2 实现 A*搜索算法。

清单 12.2　AStarSearch.java

```
package com.brackeen.javagamebook.path;

import java.util.*;
/**
    AStarSearch 类和 AStarNode 类实现一般性 A*搜索算法。AStarNode 类的子类提供搜索功能
*/
public class AStarSearch {
/**
    简单优先级表或称优先级队列。表中对象按优先级排列，由对象的 Comparable 接口确定。最高
    优先级项目放在表中开头
```

```
*/
public static class PriorityList extends LinkedList {

    public void add(Comparable object) {
        for (int i=0; i<size(); i++) {
            if (object.compareTo(get(i)) <= 0) {
                add(i, object);
                return;
            }
        }
        addLast(object);
    }
}
/**
    构造路径，不包括开始节点
*/
protected List constructPath(AStarNode node) {
    LinkedList path = new LinkedList();
    while (node.pathParent != null) {
        path.addFirst(node);
        node = node.pathParent;
    }
    return path;
}
/**
    求开始节点到目标节点的路径。返回 AStarNodes 表，找不到路径时返回 null
*/
    public List findPath(AStarNode startNode, AStarNode goalNode)
{

        PriorityList openList = new PriorityList();
        LinkedList closedList = new LinkedList();

        startNode.costFromStart = 0;
        startNode.estimatedCostToGoal =
        startNode.getEstimatedCost(goalNode);
        startNode.pathParent = null;
        openList.add(startNode);
```

```
while (!openList.isEmpty()) {
    AStarNode node = (AStarNode)openList.removeFirst();
    if (node == goalNode) {
        // 构造开始节点到目标节点的路径。
        return constructPath(goalNode);
    }

    List neighbors = node.getNeighbors();
    for (int i=0; i<neighbors.size(); i++) {
            AStarNode neighborNode =
            (AStarNode)neighbors.get(i);
        boolean isOpen = openList.contains(neighborNode);
        boolean isClosed =
            closedList.contains(neighborNode);
        float costFromStart = node.costFromStart +
            node.getCost(neighborNode);

            // 检查邻居节点是否还未遍历或找到了到这个邻居节点的更短路径
        if ((!isOpen && !isClosed) ||
                costFromStart < neighborNode.costFromStart)
        {
            neighborNode.pathParent = node;
            neighborNode.costFromStart = costFromStart;
            neighborNode.estimatedCostToGoal =
                neighborNode.getEstimatedCost(goalNode);
            if (isClosed) {
                closedList.remove(neighborNode);
            }
            if (!isOpen) {
                openList.add(neighborNode);
            }
        }
    }
    closedList.add(node);
}

// 没找到路径
return null;
}
```

}

PriorityList 内类是个简单 LinkedList，用插入排序增加节点。这里只增加 AStarNode 实例，使表由最低成本向最高成本排序，只从表开头删除节点（最低成本）。

findPath()函数与广度优先搜索法实现很相似，但有几处改变，在处理过程中计算 costFromStart 与 estimatedCostToGoal 字段。另外，如果找到该节点的更短路径，则节点从关闭表移到打开表，其余大致相同。

介绍 A*算法基础后，下面要在游戏中实际采用 A*算法。

12.4　采用 A*算法

要在游戏中实际采用 A*算法，就要用图解释游戏环境，即确定节点与边表示什么。

在自上而下地砖世界中，这很简单：节点表示地砖，如图 12-6。图中的游戏环境排成六角形栅格，可以从一个地砖移到六个相邻地砖(除非其中一个或几个相邻地砖是障碍物)。

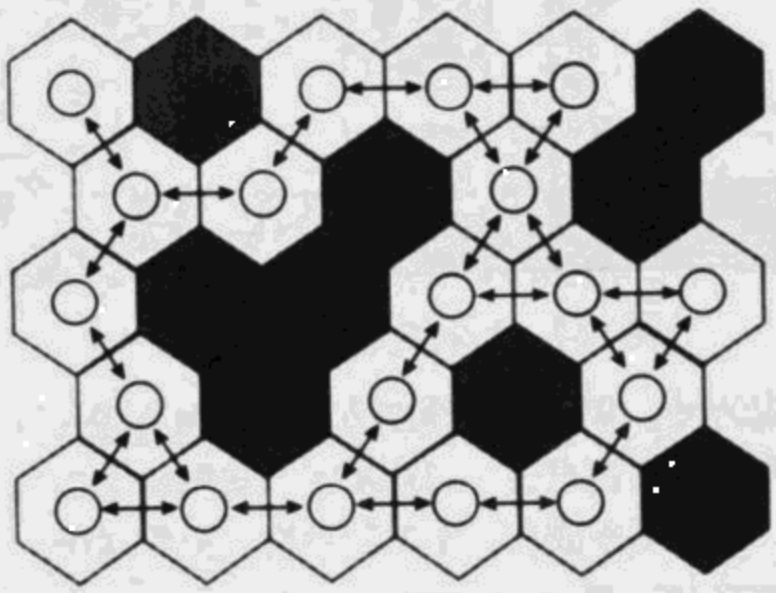


图 12-6　地砖游戏很容易用图解释游戏环境

这里，邻居节点之间的距离是相同的，但邻居节点之间的成本不一定相同。前面曾介绍过，可以让一些地砖（如沼泽地）上的运动比另一些地砖（如过道）上的运动慢。

如果是方形地砖，则对角线移动的成本比正南、正北、正东、正西移动的成本高，因为地砖的对角距离更长。

自上而下地砖世界的一个好处是很容易动态修改图。如果游戏者炸开一个地砖，使其从障碍物变成草地，则很容易修改图。

但是，地砖世界不是采用 A*算法的惟一情形。

12.5　BSP 树与 A*算法

下面看看前几章见过的典型三维世界平面图。在这个世界中，什么是节点？

首先，可以手工放上节点。只要保证相邻节点的边具有通畅的路径（没有障碍物），整个图用足够的节点覆盖。

但是，这可能有些麻烦。谁愿意坐下来定义图上的路径寻找节点呢？下面用自动方法

放置节点。

12.5.1 门户

可以利用这样的事实：平面图中的房子是凸多边形。这样，Bot 从房子的一个通道走到另一通道时，不会碰到任何墙，只要这两个通道不共线。如果通道共线（在同一条直线上），而它们之间有墙，例如宿舍的出入口，则 Bot 可能在它们之间碰上墙。因此，要小心定义房子边界，使同一房子中不会发生共线的通道。

凸室之间的通道称为门户。门户通常是不显示的多边形（或二维世界中不显示的直线），用于分隔两个凸区。图 12-7 的虚线就是门户。

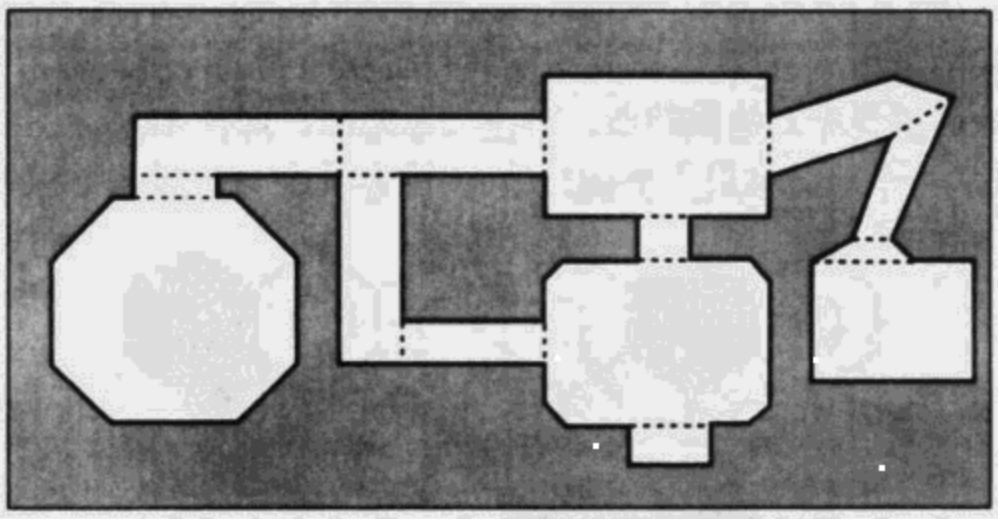


图 12-7 虚线表示凸区之间的门户

门户还可以作为绘制方法。使用门户绘制方法时，首先要画镜头所在房子中的所有多边形。如果门户在视图中，则画门户另一端的房子，直到画完整个视图。

用门户作为节点时，很容易生成从一个房间到另一房间的图，见图 12-8。

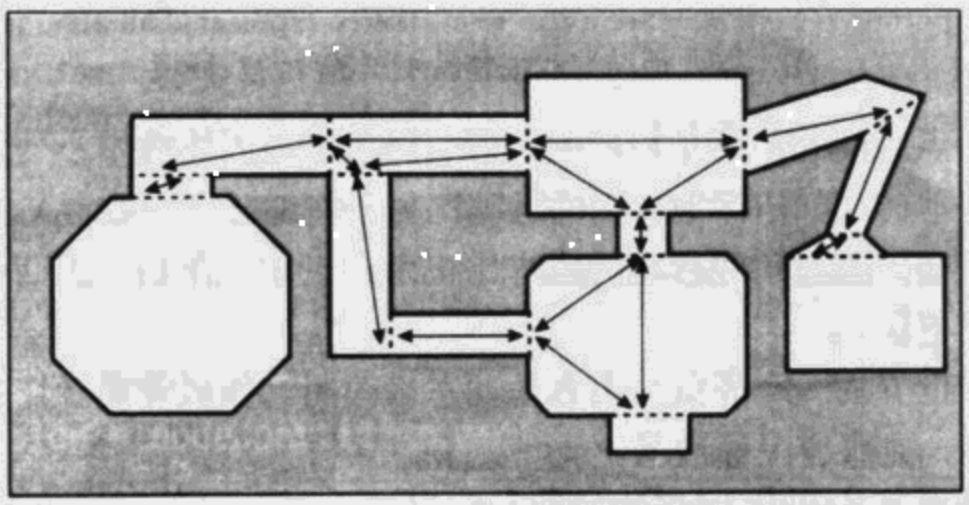


图 12-8 用门户作为节点，用图表示地图

惟一的问题是开始位置和目标位置在房子内，不是在门户上。但由于房子是凸的，因此 Bot 可以从房子中任何位置沿直线走到门户，而不会碰墙。

因此，进行 A*搜索时，可以建立临时节点，表示开始和目标。这些节点的邻居都是所在房子的门户。

注意：用门户作为节点时只考虑世界，而没有考虑基于游戏实体的障碍物。因此，如果沿路有门和其他游戏实体，则要增加代码绕过这些实体，至少要在搜索时加

以考虑。

12.5.2 实现门户

在 BSP 树中，门户可以存放在树的叶节点中，因为叶节点表示凸区。这样就很容易取得房子的所有门户。下面只是在第 10 章生成的 BSP 树的叶节点中增加一列门户：

```
public static class Leaf extends Node {
    public float floorHeight;
    public float ceilHeight;
    public Rectangle bounds;
    public List portals;
}
```

寻找门户很容易。由于二维树中的叶节点表示一个区，因此只要检查这个区的每一边，看看是否实墙。如果不是，则是个门户。三维 BSP 树中也一样，只是检查叶节点的每个面而已。

有了门户之后，要知道叶节点在其哪一面。为此可以进行简单树遍历，见清单 12.3 中 BSPTree 的 getFrontLeaf()与 getBackLeaf()方法。

清单 12.3 BSPTree.java 的 getLeaf()方法

```
/**
    取得指定分区前面的叶节点
*/
public Leaf getFrontLeaf(BSPLine partition) {
    return getLeaf(root, partition, BSPLine.FRONT);
}
/**
    取得指定分区后面的叶节点
*/
public Leaf getBackLeaf(BSPLine partition) {
    return getLeaf(root, partition, BSPLine.BACK);
}

protected Leaf getLeaf(Node node, BSPLine partition, int side)
{
    if (node == null || node instanceof Leaf) {
        return (Leaf)node;
    }
    int segSide = node.partition.getSide(partition);
    if (segSide == BSPLine.COLLINEAR) {
        segSide = side;
```

```
        }
        if (segSide == BSPLine.FRONT) {
            return getLeaf(node.front, partition, side);
        }
        else if (segSide == BSPLine.BACK) {
            return getLeaf(node.back, partition, side);
        }
        else { // BSPLine.SPANNING
            // 不能发生
            return null;
        }
    }
```

这些方法假设分区没有跨过任何其他分区，因为分区是叶节点的边。根据叶节点在分区哪一面，返回正确的 Leaf 对象。

最后，清单 12.4 的 Portal 类是 AStarNode 的子类。

清单 12.4　Portal.java

```
package com.brackeen.javagamebook.bsp2D;

import java.util.*;
import com.brackeen.javagamebook.math3D.Vector3D;
import com.brackeen.javagamebook.path.AStarNode;
/**
    Portal 表示 BSP 树两个叶之间的管道分隔（房子之间的通道）。Portal 类还是个 AStarNode,因
    此人工智能特性可以用 A*算法求出通过 BSP 树的路径
*/
public class Portal extends AStarNode {

    private BSPLine divider;
    private BSPTree.Leaf front;
    private BSPTree.Leaf back;
    private ArrayList neighbors;

    private Vector3D midPoint;
    /**
        用指定分隔和前后叶节点生成新 Portal
    */
    public Portal(BSPLine divider, BSPTree.Leaf front,
        BSPTree.Leaf back)
```

```
{
    this.divider = divider;
    this.front = front;
    this.back = back;
    midPoint = new Vector3D(
      (divider.x1 + divider.x2) / 2,
      Math.max(front.floorHeight, back.floorHeight),
      (divider.y1 + divider.y2) / 2);
}

public float getCost(AStarNode node) {
    return getEstimatedCost(node);
}

public float getEstimatedCost(AStarNode node) {
    if (node instanceof Portal) {
        Portal other = (Portal)node;
        float dx = midPoint.x - other.midPoint.x;
        float dz = midPoint.z - other.midPoint.z;
        return (float)Math.sqrt(dx * dx + dz * dz);
    }
    else {
        return node.getEstimatedCost(this);
    }
}

public List getNeighbors() {
    if (neighbors == null) {
        neighbors = new ArrayList();
        if (front != null) {
            neighbors.addAll(front.portals);
        }
        if (back != null) {
            neighbors.addAll(back.portals);
        }

        // 删除这个节点的所有引用
            while (neighbors.remove(this));
    }
```

```
        return neighbors;
    }
}
```

门户的邻居是前后叶节点的门户。getNeighbors()方法把邻居填入表中。

getEstimatedCost()方法估计门户间的成本，是每个门户中点之间的距离。

注意：对两个相邻节点，getEstimatedCost()与实际成本相同。

最后，还要一种 AStarNode，位于叶节点定义的凸区中任何位置。这些节点通常是临时的，只作为特定路径搜索的开始节点和目标节点。

清单 12.5　AStarSearchWithBSP 的 LeafNode 内类

```
/**
    LeafNode 类是个 AStarNode ，表示 BSP 树中叶节点的位置，作为特定路径搜索的开始节点和
    目标节点
*/
public class LeafNode extends AStarNode {
    BSPTree.Leaf leaf;
    Vector3D location;

    public LeafNode(BSPTree.Leaf leaf, Vector3D location) {
        this.leaf = leaf;
        this.location = location;
    }

    public float getCost(AStarNode node) {
        return getEstimatedCost(node);
    }

    public float getEstimatedCost(AStarNode node) {
        float otherX;
        float otherZ;
        if (node instanceof Portal) {
            Portal other = (Portal)node;
            otherX = other.getMidPoint().x;
            otherZ = other.getMidPoint().z;
        }
        else {
            LeafNode other = (LeafNode)node;
            otherX = other.location.x;
            otherZ = other.location.z;
```

```
        }
        float dx = location.x - otherX;
        float dz = location.z - otherZ;
        return (float)Math.sqrt(dx * dx + dz * dz);
    }

    public List getNeighbors() {
        return leaf.portals;
    }
}
```

LeafNode 类相当简单。LeafNode 的邻居是节点所在 BSP 树叶节点的所有门户，估计成本为节点位置与门户中点之间的距离。

有了 BSP 树 A*搜索所要的一切之后，下面把它们综合到一个包装类中，实现一般性路径搜索接口。

12.6　一般性路径寻找

我们生成的 A*搜索函数返回一列 AStarNodes，可以作为一般性实现，但无法取得更具体内容。实际上，我们可能需要一列位置信息，如清单 12.6 的 PathFinder 接口。

清单 12.6　PathFinder.java

```
/**
    PathFinder 接口是个函数，定义从一个位置到另一位置的路径（表示为 Vector3Ds 表），或从一
    个 GameObject 的路径。
    注意，find()方法可以忽略请求目标，指定任意路径，如侦察一条路线或远离目标
*/
public interface PathFinder {

    /**
        寻找从 GameObject A 到 GameObject B 的路径。路径是 Vector3Ds 的迭代，不包括开始位
        置(GameObject A)，但包括目标位置 (GameObject B)。可以在其他实体中使用 Vector3D
        对象，因此不能修改。找不到路径时返回 null
    */
    public Iterator find(GameObject a, GameObject b);

    /**
        寻找从开始位置到目标位置的路径。路径是 Vector3Ds 的迭代，不包括开始位置，但包括
        目标位置。可以在其他实体中使用 Vector3D 对象，因此不能修改。找不到路径时返回 null
    */
```

```
    public Iterator find(Vector3D start, Vector3D goal);
}
```

在这个接口中，提供了两个方法：一个求两点之间的路径，一个求两个实体之间的路径。接口规范指出，这两个函数应返回 Iterator 对象，对 Vector3D 位置表进行迭代。

因此，使用 PathFinder 时，基础算法并不重要，可以是 A*、随机或其他算法。另外，这个接口也适用于其他类型的路径，如按照房子范围的简单路径或作为背景效果的群鸟纹理。下一章将多处使用 PathFinder 接口。

首先要生成一个 PathFinder 接口实现，用 A*算法寻找 BSP 树中两个点之间的最短路径。清单 12.7 的 AStarSearchWithBSP 类很简单。

清单 12.7 AStarSearchWithBSP.java

```
package com.brackeen.javagamebook.path;

import java.util.*;
import com.brackeen.javagamebook.game.GameObject;
import com.brackeen.javagamebook.math3D.Vector3D;
import com.brackeen.javagamebook.bsp2D.BSPTree;
import com.brackeen.javagamebook.bsp2D.Portal;

/**
    AStarSearchWithBSP 是个 PathFinder 用 A*算法寻找 BSP 树中两个点之间的路径
*/
public class AStarSearchWithBSP extends AStarSearch
    implements PathFinder
{
    private BSPTree bspTree;

    /**
        对指定 BSP 树生成新 AStarSearchWithBSP
    */
    public AStarSearchWithBSP(BSPTree bspTree) {
        this.bspTree = bspTree;
    }

    public Iterator find(GameObject a, GameObject b) {
        return find(a.getLocation(), b.getLocation());
    }

    public Iterator find(Vector3D start, Vector3D goal) {
```

```
BSPTree.Leaf startLeaf = bspTree.getLeaf(start.x, start.z);
BSPTree.Leaf goalLeaf = bspTree.getLeaf(goal.x, goal.z);

// 如果开始与目标在同一叶节点中，则不必进行A* 搜索
if (startLeaf == goalLeaf) {
    return Collections.singleton(goal).iterator();
}

AStarNode startNode = new LeafNode(startLeaf, start);
AStarNode goalNode = new LeafNode(goalLeaf, goal);

// 临时将生成的goalNode加进邻居表中
List goalNeighbors = goalNode.getNeighbors();
for (int i=0; i<goalNeighbors.size(); i++) {
    Portal portal = (Portal)goalNeighbors.get(i);
    portal.addNeighbor(goalNode);
}

// 进行A*搜索
List path = super.findPath(startNode, goalNode);

    // 从邻居表中删除目标节点
    for (int i=0; i<goalNeighbors.size(); i++) {
        Portal portal = (Portal)goalNeighbors.get(i);
        portal.removeNeighbor(goalNode);
    }

    return convertPath(path);
}
/**
    将 AStarNodes 的路径变成 Vector3D 位置的路径
*/
protected Iterator convertPath(List path) {
        if (path == null) {
        return null;
    }
    for (int i=0; i<path.size(); i++) {
        Object node = path.get(i);
        if (node instanceof Portal) {
```

```
                path.set(i, ((Portal)node).getMidPoint());
            }
            else {
                path.set(i, ((LeafNode)node).location);
            }
        }
        return Collections.unmodifiableList(path).iterator();
    }
}
```

find()方法中的实际搜索如下：首先，生成表示开始与目标节点的临时节点，将其加进图中，然后进行搜索。搜索之后，从图中清除临时节点。如果开始与目标在同一叶节点中，则不必进行 A* 搜索，因此这个方法只返回目标地址的 Iterator。

convertPath()方法将 AStarNodes 的路径变成 Vector3D 位置的路径。这里用每个门户的中点作为要用的位置。如果不用门户的中点，也可以用门户中最接近某一边位置的点，只要留足实体放置的空间。但是，门户的中点很方便，因此这个简单方案中只用门户的中点。

12.7 建立 PathBot

也许你已经想过这样的问题：假设 Bot 寻找到游戏者的路径前开始跟踪，但 Bot 到达目标位置时，游戏者已经跑掉了，不是白费劲吗？

可以每帧重算路径，但这样会浪费大量资源。进行 A*搜索要建立不少临时对象，占用大量处理器功能（包括几个 Math.sqrt()调用），因此不能让 Bot 每帧重算路径。另外，游戏者和其他任何目标位置也不一定每帧都改变，因此路径也不必每帧都改变。

可以一段时间重算路径，比如每几秒钟重算一次。这样就可以更新路径，同时又不会浪费太多内存或处理器功能。

另一种思路是，可能有几个 Bot 同时计算路径。这时游戏的帧速率会有所下降，因为几个 Bot 会同时占用处理器功能。为了避免这个问题，可以采用简单办法，保证每个 Bot 在随机时间量之后才开始计算路径。这样就可以保证 Bot 要经过 0 到 1000 毫秒之后才计算路径。

另一种方法是在门户与门户之间缓存公共 A*搜索（不包括开始与目标节点）。这样，同一房中的两个 Bot 可以共享路径，已经计算的路径不必重算。

这样就可以了，清单 12.8 的 PathBot 采用 PathFinder 给出的路径，有几个缺省属性，如速度、转动速度及飞离地面的高度。转动速度是装饰性的，Bot 将脸转向所朝方向，但转动速度不会影响 Bot 沿路径移动的速度。

清单 12.8　PathBot.java

```
package com.brackeen.javagamebook.path;
import java.util.Iterator;
import com.brackeen.javagamebook.path.PathFinder;
```

```
import com.brackeen.javagamebook.math3D.*;
import com.brackeen.javagamebook.game.GameObject;
/**
    PathBot 是个 GameObject 沿着 PathFinder 给定的路径走
*/
public class PathBot extends GameObject {

    // 缺省值
    protected PathFinder pathFinder = null;
    protected Vector3D facing = null;
    protected long pathRecalcTime = 4000;
    protected float turnSpeed = .005f;
    protected float speed = .25f;
    protected float flyHeight = 64;

    // 跟随路径
    protected Iterator currentPath;
    protected Vector3D nextPathLocation = new Vector3D();
    protected long timeUntilPathRecalc;

    public PathBot(PolygonGroup polygonGroup) {
        super(polygonGroup);
        setState(STATE_ACTIVE);
    }

    public boolean isFlying() {
        return (flyHeight > 0);
    }
    /**
        将 PathFinder 类设置成跟随路径
    */
    public void setPathFinder(PathFinder pathFinder) {
        if (this.pathFinder != pathFinder) {
            this.pathFinder = pathFinder;
            currentPath = null;
            // 计算之前的随机时间量，使所有Bot不会同时计算路径
            timeUntilPathRecalc = (long)(Math.random() * 1000);
        }
    }
```

```
/**
    设置实体沿路径走时脸的朝向。这个值可以改变。如果为 null，则这个实体朝向移动方向
*/
public void setFacing(Vector3D facing) {
    this.facing = facing;
}

public void update(GameObject player, long elapsedTime) {

    if (pathFinder == null) {
        super.update(player, elapsedTime);
        return;
    }

    timeUntilPathRecalc-=elapsedTime;

    // 更新到游戏者的路径
    if (timeUntilPathRecalc <= 0) {
        currentPath = pathFinder.find(this, player);
        if (currentPath != null) {
            getTransform().stop();
        }
        timeUntilPathRecalc = pathRecalcTime;
    }

    // 跟随路径
    if (currentPath != null && currentPath.hasNext() &&
        !getTransform().isMovingIgnoreY())
    {
        nextPathLocation.setTo((Vector3D)currentPath.next());
        nextPathLocation.y+=flyHeight;
        getTransform().moveTo(nextPathLocation, speed);

        // 实体朝向移动方向或指定的脸部朝向
        Vector3D faceLocation = facing;
        if (faceLocation == null) {
                faceLocation = nextPathLocation;
        }
        getTransform().turnYTo(
```

```
                faceLocation.x - getX(),
                faceLocation.z - getZ(),
                (float)-Math.PI/2, turnSpeed);
            }

        super.update(player, elapsedTime);
    }
    /**
        发生碰撞时，备份 200 毫秒，然后等待几秒钟再重算路径
    */
    protected void backupAndRecomputePath() {
        //备份200毫秒
        nextPathLocation.setTo(getTransform().getVelocity());
        if (!isFlying()) {
            nextPathLocation.y = 0;
        }
        nextPathLocation.multiply(-1);
        getTransform().setVelocity(nextPathLocation, 200);

        // 等待几秒钟再重算路径
        currentPath = null;
        timeUntilPathRecalc = (long)(Math.random() * 1000);
    }

    public void notifyWallCollision() {
        backupAndRecomputePath();
    }
    public void notifyObjectCollision(GameObject object) {
        backupAndRecomputePath();
    }
}
```

PathBot 很简单，跟踪重算之前等待的时间，如果经过这个时间，则在 update()方法中重算路径。update()方法其余部分按下列方式跟随路径。

(1) Bot 变换设置成移到路径中第一个位置。

(2) Bot 移动完毕后，选择路径中下一个位置。

(3) 如果路径中已经没有更多位置，则 Bot 停止，直到重算路径。

注意：Bot 转头看 facing 向量指定的位置。如果这个值为 null，则 Bot 朝向前进方向。后面让 Bot 将注意力集中到某个地方（如游戏者）时要用到这个方法。

最后，PathBot 还处理碰撞。如果发现撞墙或其他实体，则 Bot 备份 200 毫秒，然后等

待几秒钟再重算路径。

这样就完成了路径寻找。本书的源代码中还有 PathFindingTest 类，生成几个棱形 PathBot，放在前两章使用的三维贴图中。PathBot 跟随游戏者，不管游戏者走到哪里。不管你走到哪里，武士都跟着你。

12.8 改进 A*搜索

运行 PathFindingTest 演示程序时，注意 Bot 相对于门户的一些奇怪行为。由于 BSP 建立过程要分割多边形，因此有时凸室中可能有几个门户。因此，Bot 从一个门户到另一个门户时，有时不是沿 Bot 到游戏者走直线，虽然看上去应该走直线。

要解决这个问题，有几种办法。一种方法是在运行时检查能否在路径中跳过一个门户。例如，不是从门户 A 到 B 到 C，而是直接从门户 A 到 C，只要它们之间没有障碍。

另一种方法是自己手工定义门户，像前面介绍的一样。

第三种方法将在下一章实现。如果 Bot 能看到游戏者，则直奔游戏者。这个方法并不完美，因为有些小洞可以看过去，但无法走过去。但加上其他人工智能特性，可以解决这个问题。

另一种方法是改进路径，用贝齐尔曲线进行四角光顺。这样就不会让 Bot 转尖角，而是转圆角，得到更自然的运动。

最后，一些大世界可能很大，帧与帧之间进行 A*搜索会大大减慢速度。为此可以生成低优先级“工人”线程，惟一任务就是计算路径。Bot 可以请求工人计算路径，后面工人会把求出的路径告诉 Bot。

记住，这个“工人”线程只能有一个，一次只能求一个路径，因为搜索过程中 AStarNode 的各个字段是改变的。两个线程同时修改节点会破坏路径。因此，“工人”线程可以把请求放在队列中（如第 1 章“Java 线程”中介绍的单线程 ThreadPool），在这个线程中进行所有路径计算。

另外，记住 PathBot 对 A*搜索一无所知，只是跟随 PathFinder 提供的路径。前面曾介绍过，可以修改基础搜索算法或向 Bot 提供其他有趣的路径。实际上，下一章要生成几个不同的路径模式。

12.9 小结

本章首先出现了武士。

实际上，我们首先介绍了地砖地图上的一些基本路径搜索程序，如随机搜索和“沿墙右边法”，然后介绍了基于图的高级环境，实现了一般性 A*算法，可以在图环境和地砖环境中使用（因为地砖环境也要解释为图环境）。

最后，我们合并 A*搜索与前几章介绍的 BSP 树，生成一般性路径，可以跟随各种路径，无论是否 A*搜索生成的。和平常一样，我们用一个演示程序显示了本章的材料。

路径搜索是游戏中的必备要素之一，也是游戏中许多人工智能的基础。目前，A*路径 Bot 是神仙一样的，总是知道游戏者在哪里。下一章要用路径搜索算法使 Bot 钝化一些，同

时又要增加一些智能，确定攻击、躲避和追赶游戏者。此外，还要让 Bot 后退，以便打击机器人。

我很奇怪，可以杀死机器人吗？

第 13 章　人工智能

关键问题

- 人工智能基础
- 减少 Bot 的神力
- 状态机与响应
- 概率机
- 决策
- 模式
- 对象派生
- 综合起来
- 进化
- 其他游戏人工智能思路
- 小结

我曾经想建立的第一个游戏是 Pca-Man 克隆游戏，有 16 种颜色，是用 BASIC 语言编写的，在 486 机器中运行，经常闪烁。

我曾经想让鬼子移动及笨一些。鬼子要确定走哪条路时，总是沿游戏者的方向走。这样得到的超级鬼子太厉害了，一直紧跟 Pca-Man ，使他根本没时间吃豆子。但如果他能及时吃上豆子，则所有鬼子就在身边，使 Pca-Man 很容易吃掉，对 Pca-Man 有利。

由此可见，我们很容易使敌人太聪明，也很容易使敌人太笨。关键是要寻找智能的正确平衡，使游戏既有趣又有难度。在 Pca-Man 克隆游戏中，我要使鬼子笨一些，有时会沿随机方向移动，而不是一直跟着游戏者。这个方案与原先的 Pca-Man 不尽相同，但使游戏更好玩了。

人工智能就是模拟智能和行为。人工智能是个很大的课题，应用范围很广，本章只是介绍人工智能的皮毛，介绍与游戏相关的人工智能问题。本章介绍听觉与视觉、状态机、概率机、攻击与避让模式和进化。

此外，我们还要介绍对象派生和图形重量等问题，使演示程序更漂亮。

13.1　人工智能基础

游戏中的人工智能实际上就是使生成的 Bot 具有一定的行为，无论是啮齿动物在地上爬还是人形生物在进行智能决策。本章的 Bot 一词指任何“智能”游戏实体。有了逼真的 Bot，就可以用人工智能定义游戏的感觉，就像游戏的视觉部分一样。

行为的感觉很重要。设计游戏的人工智能时，要自上而下设计。先考虑 Bot 要什么样子，然后再实现这个样子。很多人喜欢自下而上设计人工智能，先设计复杂的模式与基础结构。例如，也许不同 Bot 要有不同搜索模式，像游戏者一样在迷宫中漫步，但在三维世

界中，游戏者可能以为 Bot 是随机出现的，要来攻击的。因此，先要考虑游戏者看到的 AI 生成应是什么样子，然后再进行设计。

当然，许多 AI Bot 首先要有良好的路径寻找算法，使其知道如何在地上任何两点之间移动。如果没有路径寻找功能，则会得到恼人而乱七八糟的人工智能，使 Bot 粘在墙上或无法转变。可以使用上一章实现的 A*算法进行路径寻找。

人工智能 Bot 的总体目标不是分散注意力，而是相反。Bot 之类小东西以不自然的方式移动是很烦人的，有时人工智能只有在做得不好时才会引人注意。

还要让不同角色具有与其外观相符的不同行为。例如，人形生物要进行智能选择，因为用户认为人形生物能进行智能选择，另一方面，蜘蛛状生物可以采用简单模式，仍然是可信。

另一个思路是在游戏过程中让敌人越来越聪明和难斗。不要使用太容易预测的人工智能，否则游戏者会越玩越没劲。另外，首先要记住人工智能的目的：寻找一个合理折衷，使敌人既不太精，也不太笨，使游戏既好玩又有难度，这就需要恰当的平衡。

本章设计的人工智能系统可以本书的三维引擎中使用。Bot 具有不同个性，每个 Bot 用不同运动和攻击模式加以区别。

13.2 减少 Bot 的神力

上一章介绍路径寻找时，我们生成的演示程序中，Bot 像神一样，总是知道游戏者位置，总能找到到达游戏者的路径。这与 Terminatol 或 Pac-Man 鬼子存在相同问题。

我们不希望 Bot 有这种神力，也要让 Bot 有时不知道游戏者位置。

三维游戏中一个可行的方法是让游戏者位于 Bot 的视野中时才知道游戏者位置，即让 Bot 具有视觉智能。

13.2.1 视觉

一般来说，视觉智能的目的是检查 Bot 能否看到游戏者。虽然，Bot 看不到障碍物后面的东西，如墙后面的东西和天花板上面的东西。

可以从 Bot 向游戏者发射一束“视觉光”，看看是否遇到任何障碍物，见图 13-1。

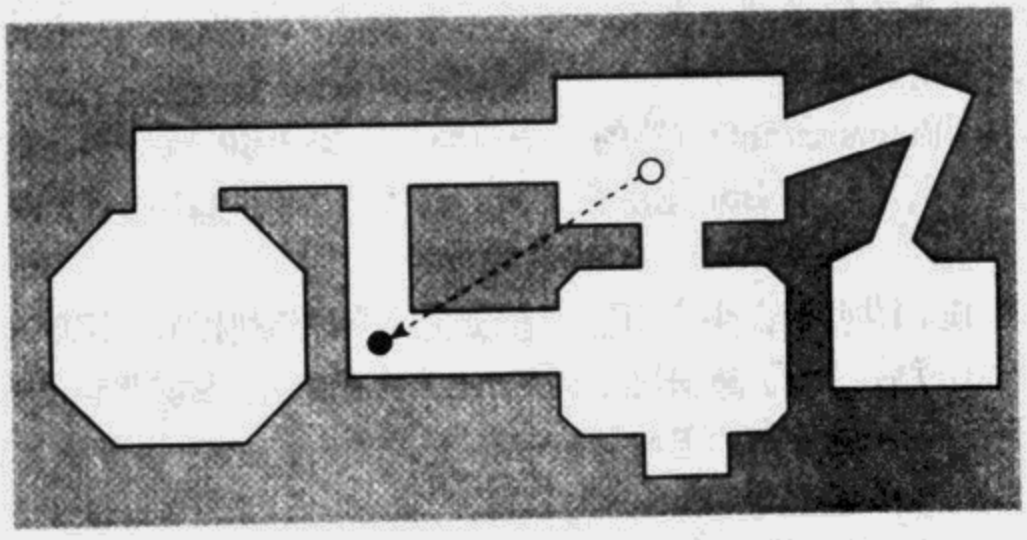


图 13-1 Bot（白圆圈）想找到游戏者（黑圆圈），可以从 Bot 向游戏者发射一束“视觉光”，看看是否遇到任何障碍物

如果视觉光没有遇到任何障碍物，则 Bot 能看到游戏者，因此知道游戏者的位置。知道位置后，Bot 就可以攻击或采取其他措施。但怎么实现视觉光呢？

在地砖世界中，这个问题很容易解决。只要用标准直线方程画一条直线，求出与这条直线相交的所有地砖，然后检查每个地砖，确定是否能看过去。

对 BSP 树，我们已经实现视觉光功能。第 11 章“碰撞探测”中的碰撞探测代码有个 getFirstWallIntersection()方法，求出两点之间第一个墙交点。可以用这个函数检查直线段是否与 BSP 树中任何墙相交。如果不与任何墙相交，则这个方法返回 null，可以看到游戏者。试试看，本章大多数人工智能相关代码都在 PathBot 类中，它是上一章 AIBot 的子类。首先在 PathBot 中增加 canSee()方法，如清单 13.1 所示。

清单 13.1　AIBot. canSee()

```
/**
    检查这个实体能否看到指定实体（假设实体头上有眼睛）
*/
  public boolean canSee(GameObject object){
      //检查这个 Bot 到实体的直线是否与墙相遇
      //hits any walls
      boollean visible =
              (collisonDetection.getFirstWallIntersection(
                getX(),getZ(),object.getX(),object.getZ(),
                getY(),getY() + 1) ==null);
      if (visible){
            timeSincePlayerLastSeen = 0;
      }
      return visible;
  }
```

注意：我们把 timeSincePlayerLastSeen 字段设置为 0。这个变量在 update()方法中递增，因此总是知道上次看到游戏者以来经过的时间。这样，如果游戏者看不到，但最近还能看到，则可以变成某些功能，如用 A*搜索算法搜索最近几秒还能看到的游戏者。

使用 getFirstWallIntersection()方法的一个问题是二维 BSP 树中存在的问题：光束是水平的，则视觉光无法向上，即 Bot 无法往上看。这个问题可以通过组合其他人工智能功能加以解决。

另一个问题是 Bot 的眼睛在头上背面，视觉光是各向发射的，不管 Bot 朝哪里，因此游戏者无法躲在 Bot 背后。为了解决这个问题，可以实现 AI 生物背后的盲区，如图 13-2。

这很简单，只要保证视觉光和 Bot 朝向之间的夹角不太大。这个练习留给读者。

如果游戏者躲在 Bot 背后，突然一声爆叫呢？Bot 听不到，因此没关系。如果游戏者跳跃、摔倒或发射武器时，Bot 能听到呢？

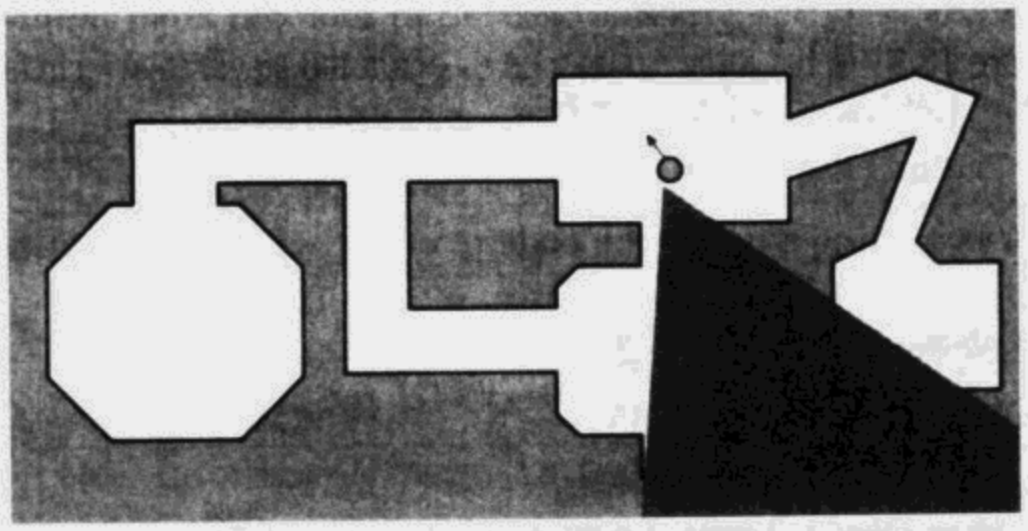


图 13-2 实现 AI 生物背后的盲区，使游戏者能躲在 Bot 背后

13.2.2 听觉

在实际世界中，如果你发出声音，则声波会在墙上回弹，进入另一边，沿着通道和四角传播。声音越远越小，远离声源时就听不到了。

因此，游戏者可以让 Bot 有个听觉雷达，可以听到一定范围内的任何声音，不管中间是否有墙，如图 13-3 所示。

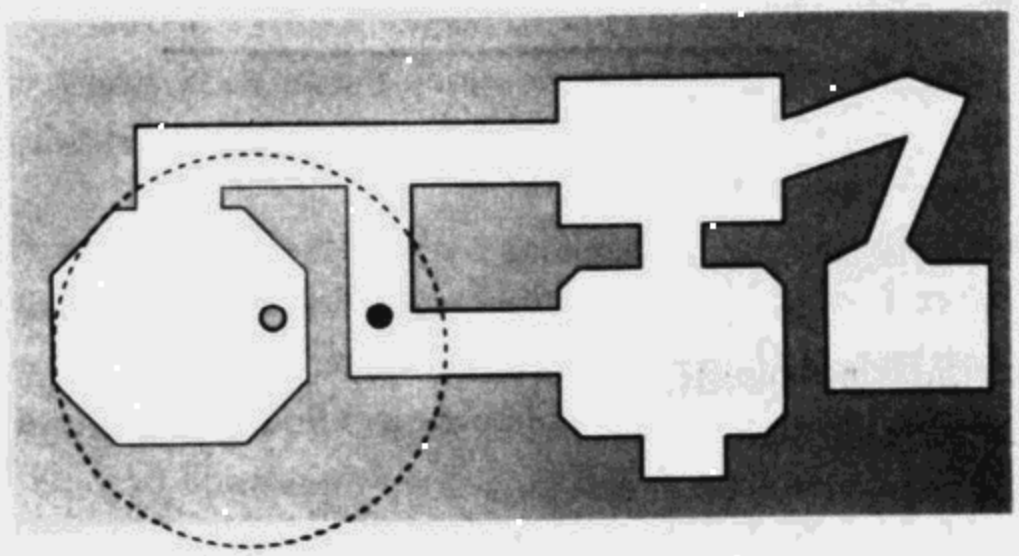


图 13-3 Bot（灰圈）想找到游戏者（白圈），可以听到一定范围内的任何声音

这时，如果游戏者发出声音，则代码检查其是否在听觉范围内，在检查离游戏者的距离，见清单 13.2。

清单 13.2 AIBot.canHear()

```
float hearDistance;
/**
    检查这个实体能否听到指定实体。指定实体要在一定范围内并发声
*/
    public boolean canSee(GameObject object) {

    */
    public boolean canHear(GameObject object) {
```

```
// 检查实体是否发声，Bot 是否有听觉
if (!object.isMakingNoise() || brain.hearDistance == 0) {
    return false;
}

// 检查 Bot 是否足够靠近声音
float distSq = getLocation().
    getDistanceSq(object.getLocation());
float hearDistSq = brain.hearDistance * brain.hearDistance;
boolean heard = (distSq <= hearDistSq);
```

这里实际上使用距离平方，因此不必进行 Math.sqrt()运算。

这很容易，但怎么判断实体是否发声呢？isMakingNoise()方法何时返回真值？

通常，声音不是个立即事件，而是持续几秒钟。因此，可以跟踪每个实体所发声音的余音时间。清单 13.3 的 GameObjcet 基类增加了几个方法。

清单 13.3　GameObjcet 的声音方法

```
private long noiseDuration;

...
/**
    实体发声时返回真值
*/
public boolean isMakingNoise(){
return(noiseDuration > 0);

/**
    表示这个实体发出一定长度的声音，用于确定一个实体能否听到另一个实体。
*/
    public void makeNoise(long duration) {
        if (noiseDuration < duration) {
            noiseDuration = duration;
        }
    }
*/
    public void update(GameObject player, long elapsedTime) {
        if (isMakingNoise()) {
            noiseDuration-=elapsedTime;
        }
        ...
```

```
    }
```

在游戏者代码中，游戏者射击时，调用 makeNoise(500)方法，使 Bot 可以在 500 毫秒时间内听到射击声。注意这与 SoundManager 代码是完全分开的，虚拟声音是假想的，不通过音箱发声。

代码中假设所有声音的音量相同。但实际中声音越大，传播越远。可以扩展这个代码，使每个声音有不同音量或指定声音传播的最大距离。这个练习也留给读者。

为了提供一些公用人工智能路径程序，我在 Vector3D 类中增加两个方法，取得两点之间的距离，见清单 13.4。

清单 13.4　Vector3D 的距离方法

```
/**
    取得这个向量与指定向量之间距离
*/
public float getDistanceSq(Vector3D v) {
    float dx = v.x - x;
    float dy = v.y - y;
    float dz = v.z - z;
    return dx*dx + dy*dy + dz*dz;
    }
/**
    取得这个向量与指定向量之间的距离的平方
*/
public float getDistance(Vector3D v) {
        return (float)Math.sqrt(getDistanceSq(v));
}
```

这样 Bot 能听能看了，但还不能采取不同动作。下面要让 Bot 响应不同情形。

13.3　状态机与响应

Bot 对不同情形采取不同动作。根据环境中发生的情况和能否听到与看到游戏者，Bot 可以采取不同动作，如停下来、巡查、攻击、躲避、追赶、跑开、充电、隐藏或做别的事。

前面几章已经用到状态机，实体在不同状态后采取不同动作。例如，基本游戏实体类有闲置、活动和破坏状态。状态机也称为有限状态机（finite state machines），因为状态数是有限的。

有限状态机还定义实体状态何时改变，如何改变。例如。Bot 可以从停止变成散步变成跑步，但不能直接从停止变成跑步。另外，Bot 要能看到时才开始散步。

此外，可以生成状态层次。例如，本章的 Bot 有三种状态是战斗状态：攻击、躲避、跑开。此外，攻击状态还有不同的攻击模式，因此战斗状态是父状态，有三个子状态，这些子状态又还有子子状态。

图 13-4 是本章人工智能 Bot 使用的状态机。这个状态机有两个基本父状态：正常与战斗状态。Bot 要能看到时才进入战斗状态，看不到时则转为正常状态。

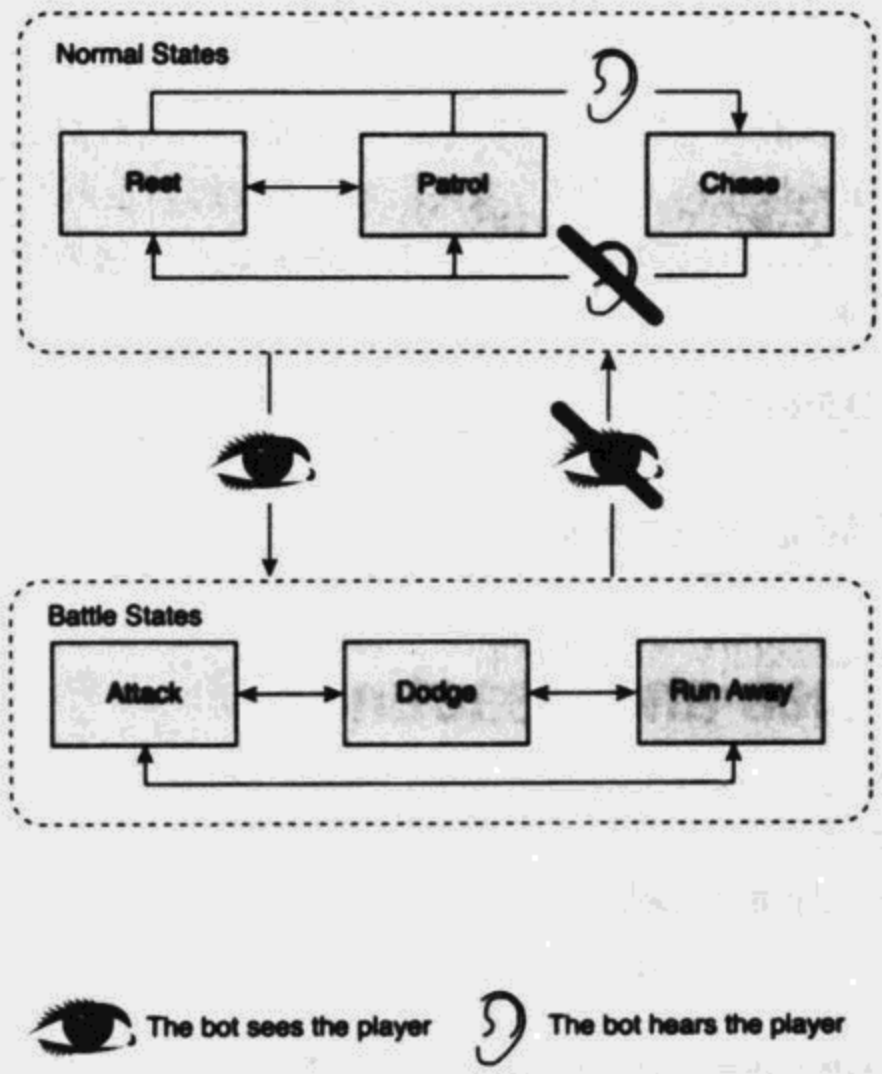


图 13-4　Bot 状态机

虽然状态有层次，但也一定要生成完善的状态代码。这里对前面状态机使用普通整数：

```
public static final int NORMAL_STATE_IDLE = 0;
public static final int NORMAL_STATE_PATROL =1;
public static final int NORMAL_STATE_CHASE = 2;
public static final int NORMAL_STATE_ATTACK = 3;
public static final int NORMAL_STATE_DODGE = 4;
public static final int NORMAL_STATE_RUN_AWAY = 5;
```

我们分开处理人工智能状态和游戏实体状态，使其不会相互干扰。我们实际使用的状态会多一些，但这是个很好的开端。设置正确的人工智能状态很简单，见清单 13.5。

清单 13.5　AIBot 的状态机方法

```
private int aiState;
private long elapsedTimeInState;

...
/**
    设置这个 Bot 的 AI 状态（不同于 GameObject 状态）
```

```
*/
    protected void setAiState(int aiState, GameObject player) {
        if (this.aiState  ! == aiState) {
            this.aiState == aiState
            elapsedTimeInState = 0;
        }
        //在此设置这个状态的相应路径
        ...
    }
    public void update(GameObject player, long elapsedTime) {
        elapsedTimeInState+=elapsedTime;
        ...
    }
```

除了设置状态之外，还要跟踪特定状态所在的时间长度。例如，如果被赶的时间太长时想停止追赶，则可用到这个时间长度。

SetAiState()方法还不完整，还要设置 Bot 路径（用前一章的 PathFinder 接口），使其按适当路径攻击、追赶或进行其他动作。稍后要增加这些代码。

13.4　概率机

从图 13-4 可以看出，不同战斗状态没有什么顺序，因此怎么确定要攻击、躲避或跑开呢？

有时要根据某些原因选择不同状态，例如发现抛射弹射来时，要避开。

有时则随机选择不同状态，可以选择一种状态或另一种状态。图 13-5 显示了一个例子。

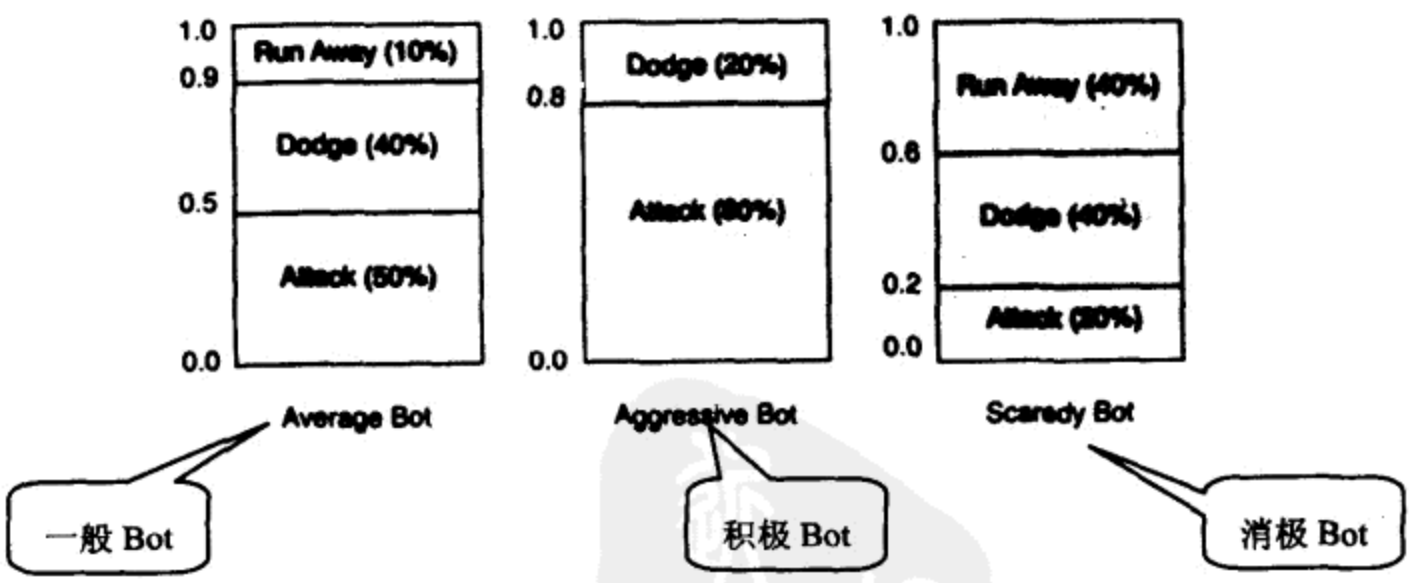


图 13-5　每个 Bot 进行不同战斗状态的概率不同

图中显示了三种不同 Bot，每种 Bot 战斗状态的概率不同，但概率之和为 100%。

也许你习惯于概率用百分比，扔一枚硬币时，正面朝上的概率为 50%。也可以把概率看成分数，例如，骰子出现一个数的概率为 1/6，而电影票的概率百万分之一。本章用概率

的十进制表示，是统计译本中常用的方法。按照这种表示，概率为 0 到 1 之间。概率为 1 表示事件中一定会发生，概率为 0 表示事件中一定不会发生，概率为 0.5 表示事件有一半机会会发生。

图 13-5 可以用饼形图描述，但图中的布局显示了如何确定选择的状态。例如，随机数产生器产生 0 到 1 的数。对一般 Bot,如果结果在 0 到 0.5 之间，则可能是攻击状态；如果结果在 0.5 到 0.9 之间，则可能是躲避；如果结果在 0.9 到 1.0 之间，则可能是跑开状态。

可以跟踪每种战斗状态的概率如下：

```
//每种战斗状态的概率
//（总和应为 1）
float attackProbability;
float dodgeProbability;
float runAwayProbability;
```

注意：和图 13-5 一样，这些概率总和应为 1，至少有一个事件会发生。可以像清单 13.6 一样随机选择一个状态。

由于概率总和应为 1，因此如果没有选择攻击与躲避，则一定是跑开状态。

清单 13.6　AIBot .chooseBattleState()

```
/**
    随机选择一个状态（攻击、躲避或跑开）
*/
public int chooseBattleState(){
        float p = (float)Math.random();
        if (p <=attackProbability){
            return BATTLE_STATE_ATTACK;
        }
        else if (p <=attackProbability + dodgeProbability){
            return BATTLE_STATE_DODGE
        }
        else{
            return BATTLE_STATE_RUN_AWAY;
        }
        }
```

重要随机函数

可以使用 Math.random()中的 chooseBattleState()方法，因为这个随机函数产生 0 到 1 的数，但本章要大量使用随机数，需要更灵活的结果，而不只是产生 0 到 1 的数，因此可以引入一些随机函数。清单 13.7 的 MoreMath 类引入一些随机函数。

清单 13.7　MoreMath.java 的随机函数

```
/**
    返回 0 到 max(包括)的随机整数
*/
public static int random(int max) {
        return (int)Math.round(Math.random() * max);
/**
    返回 min 到 max(包括)的随机整数
*/
public static int random(int min, int max) {
        return min + random(max-min);
}
/**
    返回 0 到 max(包括)的随机浮点数
*/
public static float random(float max) {
    return (float)Math.random()*max;
}
/**
    返回 min 到 max(包括)的随机浮点数
*/
public static float random(float min, float max) {
    return min + random(max-min);
}
/**
    从表面返回随机实体
*/
public static Object random(List list) {
    return list.get(random(list.size() - 1));
}
/**
    在随机事件发生时返回真值，指定值 P 是在随机事件发生的概率（0 到 1）
*/
public static boolean chance(float p) {
    return (Math.random() <= p);
}
```

前四个函数返回一定范围的随机数。后一个函数从表中返回随机实体，最后一个函数在随机事件发生时返回真值，指定值 P 是在随机事件发生的概率。

13.5 决策

决策很容易，但前面曾介绍过，也不能让生物太聪明。虽然，不必每帧都重新决策，因此代码应提供一个限制定期进行决策，例如每秒钟一次。较笨的生物可以使用更长的决策期。

Bot 决策的频率可以根据感觉进行调整。

清单 13.8 的样本代码可以在 AIBot 中进行决策。这里用一个特殊状态 DECISION READY 表示要进行决策。Bot 每几秒决策一次（根据 decisionTime），让闲置时决策次数更多。这样，如果发现游戏者，它可以迅速转入另一状态。走完路径之后，也要进行决策（在 PathBot 类代码中）。

清单 13.8 AIBot 中的决策

```
//决策之间的时间（毫秒）
long decisionTime

...
public void update(GameObject player, long elapsedTime) {

        elapsedTimeSinceDecision+=elapsedTime;
        elapsedTimeInState+=elapsedTime;
        timeSincePlayerLastSeen+=elapsedTime;
//如果闲机和看到游戏者，每 500 毫秒决策一次
        if ((aiState == NORMAL_STATE_IDLE ||
                aiState == NORMAL_STATE_PATROL) &&
                elapsedTimeInState >= 500)
        {
                aiState = DECESION_READY;
        }
//如果时间到，进行决策
        else if (elapsedTimeSinceDecision >= brain.decisionTime) {
                aiState = DECESION_READY;
        }
//如果完成当前路径，进行决策
        else if (currentPath != null && !currentPath.hasNext() &&
                !getTransform().isMovingIgnoreY())
        {
                aiState = DECESION_READY;
        }
//进行新决策
```

```
        if (aiState == DECESION_READY) {
            elapsedTimeSinceDecision = 0;

            if (canSee(player)) {
                setAiState(chooseBattleState(), player);
            }
            else if (timeSincePlayerLastSeen < 3000 ||
                canHear(player))
            {
                setAiState(NORMAL_STATE_CHASE, player);
            }
            else {
                setAiState(NORMAL_STATE_IDLE, player);
            }
        }
    }
```

决策时，Bot 要选择战斗状态、追赶状态或闲置状态。如果看到游戏者，则选择战斗状态；如果最近 3 秒内看到游戏者或听到游戏者的声音，则 Bot 选择追赶状态；否则选择闲置状态。

Bot 选择状态时发生什么情况呢？要让 Bot 实际完成某个动作，大多数动作都是某种运动，这时就要用到模式。

13.6 模式

前面介绍了 Bot 的不同状态，如攻击与躲避。游戏中的生物在特定状态中通常要完成不同运动模式。例如，躲避时，Bot 可能左右摇摆。

可以用上一章的 PathFinder 接口编写一些模式。这个接口的方法返回所循路径中位置的 Iterator。清单 13.9 的抽象 AIPattern 类实现这个接口。AIPattern 类有两个方便方法帮助建立模式。

清单 13.9 AIPattern 中的方便方法

```
/**
    对指定位置计算地板。如果不能确定地板，则使用指定的缺省值
*/
protected void calcFloorHeight(Vector3D v, float defaultY) {
    BSPTree.Leaf leaf = bspTree.getLeaf(v.x, v.z);
    if (leaf == null || leaf.floorHeight == Float.MIN_VALUE) {
        v.y = defaultY;
    }
```

```
        else {
            v.y = leaf.floorHeight;
        }
    }
    /**
     取得游戏者与离游戏者一定距离以外的 Bot 之间的位置
    */
     protected Vector3D getLocationFromPlayer(GameObject bot,
          GameObject player, float desiredDistSq)
     {
         // 取得实际距离（平方）
          float distSq = bot.getLocation().
               getDistanceSq(player.getLocation());

          // 如果在 5 个单位内，则很接近了
          if (Math.abs(desiredDistSq - distSq) < 25) {
               return new Vector3D(bot.getLocation());
          }

          // 计算从 Bot 到游戏者的向量
          Vector3D goal = new Vector3D(bot.getLocation());
          goal.subtract(player.getLocation());

         // 寻找离游戏者的目标距离
          goal.multiply((float)Math.sqrt(desiredDistSq / distSq));

          goal.add(player.getLocation());
          calcFloorHeight(goal, bot.getFloorHeight());

          return goal;
     }
```

第一个方便方法 calcFloorHeight()取得任何位置的地面高度。如果位置越界，则使用指定的缺省值。

第二个方便方法简化几种模式的编写，返回游戏者与 Bot 之间的位置，游戏者与离游戏者在一定距离以外。通常，Bot 要从一定距离攻击或躲避。

13.6.1 躲避

我们介绍的第一个模式是躲避。根据所要结果，躲避可简可繁。图 13-6 显示了一些躲避模式。

躲避模式：　躲避模式　　躲避模式：

前两种躲避模式相对于游戏者把 Bot 移动特定位置，大多数模式都是这么干的。第三种模式（隐藏）要复杂一些。这个模式在环境中找到一个点，让游戏者看不到 Bot，因此模式寻找要对环境有时了解。我们把这个练习留给读者。

下面把前两个躲避模式作为例子。清单 13.10 显示了左右躲闪模式。

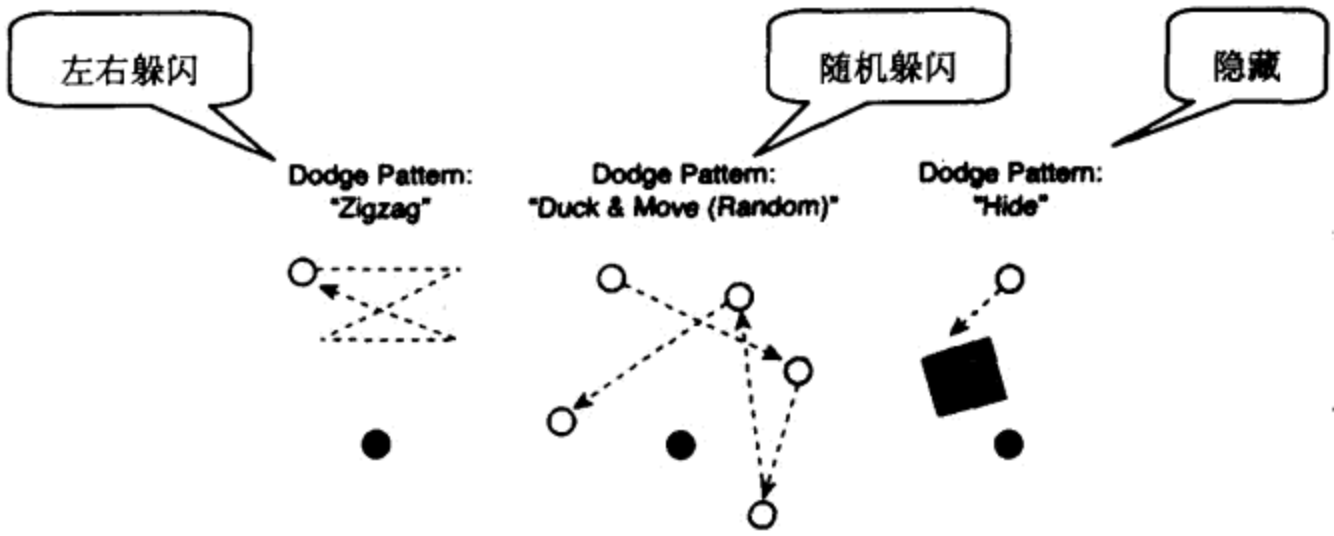


图 13-6　一些躲避模式

清单 13.10　DodgePatternZigZag.java

```
public class DodgePatternZigZag extends AIPattern{

private float dodgDist;

public DodgePatternZigZag(BSPTree tree,float dodgDist){
      super(tree);
      this. DodgDist = dodgDist
}
public Iterator find (GameObject bot, GameObject player){

  //生成到躲闪位置的向量
  Vector3D zig = new Vector3D(bot.getLocation());
  zig.subtract(player.getLocation());
  zig.normalize();
  zig.multiply(dodgDist);
  zig.rotateY((float)Math.PI/2);
  //50%概率，躲到一边或另一边
     if (MoreMath.chance(.5f)){
        zig.multiply(-1)
     }
  //将向量变成绝对位置
```

```
        zig.add(bot. getLocation());
        calcFloorHeight(zig, bot.getFloorHeight());

        List path = new ArrayList();
        path.add(zig);
        path.add(zag);
        return path.iterator();
    }
}
```

左右躲闪模式让 Bot 沿到游戏者的切线方向移动一定距离（dodgeDist）,然后回到 Bot 的开始位置。Bot 随机躲到一边或另一边。

左右躲闪模式非常简单。下一个是随机躲避模式，让 Bot 在游戏者的半圆内移到一个随机位置。限制半圆保证 Bot 不会在进入躲闪位置时遇到游戏者。清单 13.11 显示了随机躲避模式的代码。

清单 13.11　DodgePatternRandom.java

```
public class DodgePatternZigZag extends AIPattern{

    private float radiusSq;

    public DodgePatternZigZag(BSPTree tree,float radius){
        super(tree);
        this. radiusSq = radius *radius
    }
    public Iterator find (GameObject bot, GameObject player){

        Vector3D goal = getLocationFromPlayer(bot,player,
                radiusSq);
          //选择这个半圆上的随机位置（离当前位置-90 度到 90 度）
            float maxAngle = (float) Math.PI/2;
        float angle = MornMath.random(-maxAngle, maxAngle)
        goal.subtract(player.getLocation());
        goal.rotateY(angle)
        goal.add(player.getLocation());
        calcFloorHeight(goal, bot. getFloorHeight);

        return Collections.singleton(goal).iterator();
    }
}
```

随机躲避模式只是增加一些向量运算。

注意这些模式并不进行环境检查，看看是否有模式所需的空间。例如，也许 Bot 只能转到一边而不能转到另一边。理想的模式是进行环境检查，调整相应模式。

当然，这里实现的躲避模式只是个开头，读者可以增加更多躲避模式。另一种躲避模式是跟踪遇到的抛射弹，沿抛射弹路径的垂直位置躲避，或进行其他运动，如俯卧或跳跃。

13.6.2　攻击

下面要实现一些攻击模式。攻击模式随 Bot 的攻击方式而不同。例如，有的发射抛射弹，有的撞击游戏者，有的变成更大，把你吓跑。图 13-7 显示了本章 Bot 实现的一些攻击模式。

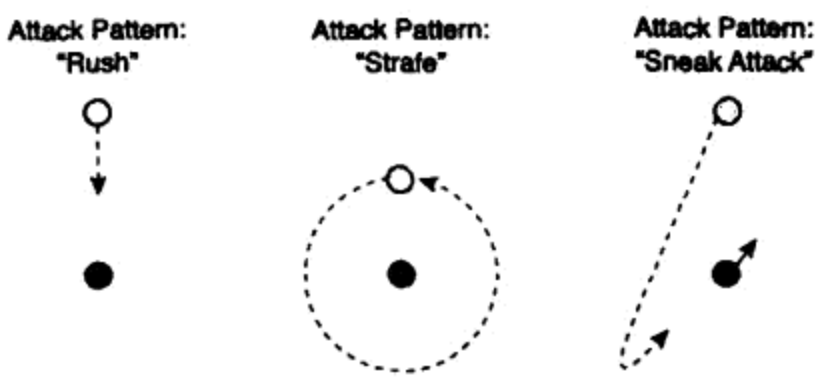


图 13-7　一些攻击模式

突击模式是最简单的攻击模式：Bot 只是移动到游戏者的一定范围内。可以使用 AIPattern 中生成的 getLocationFromPlayer()方法，见清单 13.12。

清单 13.12　AttcakPatternRush.java

```
public class AttcakPatternRush extends AIPattern{

    private float desiredDistSq;

    public AttcakPatternRush (BSPTree tree,float desiredDist){
        super(tree);
        this. desiredDistSq = desiredDist * desiredDist
    }
    public Iterator find (GameObject bot, GameObject player){

        Vector3D goal = getLocationFromPlayer(bot,player,
                desiredDistSq);
            if (goal.equals(bot.getLocation())){
            return null;
              }
              else{
```

```
            return Collections.singleton(goal).iterator();
            }
        }
}
```

如果 Bot 已经处于突击位置，则这个模式返回 null。

包抄游戏者初看起来很难，其实只要设计得好，能大大增加 Bot 的智能感觉。包抄就是绕游戏者走圆圈，使 Bot 很容易发射抛射弹，而游戏者很难向 Bot 开火，因此是很好的防御性战术。

其实不用实际走圆圈，只要让 Bot 绕游戏者转八角形，把路径简化为 8 个位置。这个代码见清单 13.13。

清单 13.13 AttackPatternStrafe.java

```
public class AttackPatternStrafe extends AIPattern{

    private float radiusSq;

    public AttackPatternStrafe (BSPTree tree,float radius){
        super(tree);
        this. radiusSq = radius* radius
    }
    public Iterator find (GameObject bot, GameObject player){

        List path = new ArrayList();
        //寻找所要半径内第一个位置
        Vector3D firstGoal = getLocationFromPlayer(bot,player,
                radiusSq);
        if (!firstGoal.equals(bot.getLocation())){
            path.add(firstGoal);
        }
        //绕游戏者沿逆时针转（实际上是个八角形）
        int angle = (float)(2 * Math.PT/numPoints);
        if (MoreMath.chance(0.5f)){
            angle = - angle
        }
        float lastY = bot.getFloorHeight();
        for (int i=1; i<numPoints,i++){
            Vector3D goal =new Vector3D(firstGoal);
            goal.subtract(player.getLocation());
            goal.rotateY(angle * i)
```

```
            goal.add(player.getLocation());
            calcFloorHeight(goal, lastY);
            lastY = goal.y;
            path.add(goal);
        }
        //增加最后位置（图形开头）
        path.add(firstGoal);
        return path.iterator();
    }
}
```

在这个模式中，利用从游戏者到所要位置的向量可以绕 y 轴将这个向量转 8 次，得到所要的 8 个不同点。

除了这两个模式外，还可以有很多其他模式。上图中的偷袭模式也是给读者练习的，还有其他攻击模式，如撞击、神风突击队攻击、等待与埋伏或跳到游戏者身上。

当然，游戏中 Bot 不一定总要攻击，许多坏蛋（如蜘蛛与小动物）对游戏者完全没有敌意，只是埋头走自己的路，想自己的事。任天堂 Mario 系列之类游戏有大量这类生物。一定要在不同模式中增加一些变化和创造性。

13.6.3 跑开

Bot 有时要远离游戏者。也许游戏者手中有巨大而可怕的武器，也许 Bot 身体不好，想跑开。我们经常要让紧张的 Bot 跑开，或出于防御目的暂时跑开。

跑开模式只要一个跑开点。清单 13.14 显示的模式让 Bot 直接朝游戏者反方向跑开，即使撞上墙也不管。

清单 13.14 RunAwayPattern.java

```
public class RunAwayPattern extends AIPattern{

    public RunAwayPattern (BSPTree tree,float radius){
      super(tree);
    }
  public Iterator find (GameObject bot, GameObject player){
    //重动作：让 Bot 直接朝游戏者反方向跑开，即使撞上墙也不管。
    Vector3D goal =new Vector3D（player.getLocation()）；
    goal.subtract(bot.getLocation());

    //反方向
    goal.multiply(-1)

   //赶紧跑开
```

```
        goal.multiply（100000）
        calcFloorHeight(goal,bot.getFloorHeight());
        //返回代码器
        return Collections.singleton(goal..iterator();
    }
}
```

另一种跑开方法是选择地图上一个位置，通过路径寻找搜索取得到这个位置的路径。这个位置可能是 Bot 的出生地、安全位置或一群 Bot 的集合位置。

在灯光很强的游戏中，Bot 可能隐藏在阴影中，同样，还可以生成许多其他模式。

13.6.4 瞄准

瞄准不是运动模式，但 Bot 经常要将武器瞄准游戏者，保证不同程度的成功率：可以让一些 Bot 的瞄准能力比一些 Bot 更强。图 13-8 显示了几种瞄准模式。

图 13-8 几种瞄准模式

我们只实现一个瞄准模式，可以按 0 到 1 的指定精度瞄准（见清单 13.15）。精度为 0 表示瞄准误差为 10 度，精度为 1 表示对准游戏者，但没有考虑游戏者的速度。

清单 13.15 AimPattern.java

```
public class AimPattern extends AIPattern{
        protected float accuracy;
        public AimPattern (BSPTree tree){
            super(tree);
        }

        /**
            设置瞄准精度，0 为最差，1 为最好
         */
        public void setAccuracy(float p){
            this.accuracy = p;
        }
        public Iterator find (GameObject bot,GameObject player){
        Vector3D goal =new Vector3D（player.getLocation();
```

```
        goal.y + = player.getBounds().getTopHeight()/2
        goal.subtract(g.getLocation());
        //绕 y 轴转 10 度（也可以用上/下随机偏移量）
        if (accuracy<1){
            float maxAngle = 10 * (1- accuracy);
            float angle = MoreMath.random(-maxAngle, maxAngle);
            goal.rotateY((float)Math.toRadians(angle);
        }
        goal.normalize();

        //返回一个迭代器
        return Collections.singleton(goal). iterator();
    }
}
```

注意：这个瞄准模式返回范化向量方向而不是绝对位置，因为我们瞄准的是一个方向。

这就是本章要用的所有模式。模式很容易编写，因此不妨自己试试。但下面要在瞄准之后实际发射抛射弹。

13.6.5 射击

瞄准时还要考虑一个问题，就是 Bot 瞄准所需的时间量。不能让 Bot 每一帧都发射武器，而要让它等一会儿，就像是瞄准所需的时间量，因此只能在瞄准所需的时间量之后才射击。

代码中，我们让瞄准所需的时间量与射击精度成反比。两秒钟以上瞄准效果好，不到这个时间则不够精确。我们把代码加速清单 13.16 的 AIBot 类中。

清单 13.16 AIBot 中的瞄准与射击

```
AimPattern   aimPathFinder;
Long aimTime

Public void update(GameObject player,long elapsedTime){

    ...
    if (aiState == BATTLE_STATE_ATTACK &&
            elapsedTimeInState >= brain.aimTime &&
            aimPathFinder != null)
    {
            elapsedTimeInState != aimTime;
            //瞄准时间越长越精确
            float p = Math.min(1, aimTime / 2000f);
```

```
                aimPathFinder.setAccuracy(p);
                Vector3D direction = (Vector3D)
                    aimPathFinder.find(this, player).next();
                fireProjectile(direction);
            }
    }
    /**
        指定方向发射抛射弹。方向向量要范化
    */
      public void fireProjectile(Vector3D direction) {

        Projectile blast = new Projectile(
            (PolygonGroup)blastModel.clone(),
            direction, this, 3, 6);
        float dist = 2 * (getBounds().getRadius() +
            blast.getBounds().getRadius());
        blast.getLocation().setTo(
            getX() + direction.x*dist,
            getY() + getBounds().getTopHeight()/2,
            getZ() + direction.z*dist);

        // 派生新的游戏实体
        addSpawn(blast);
    }
```

fireProjectile()方法中生成了表示抛射弹的全新游戏实体，怎么处理这个实体呢？游戏实体管理器会负责这个工作。因此，只在将实体标为派生，游戏实体管理器就会选择它。下面介绍实体派生。

13.7 实体派生

和抛射弹例子一样，有时实体要派生另一实体。例如，实体可能炸成多片，得到几个派生碎片在不同方向飞，Bot 可能掉了武器，Bot 被击中时可能冒出火花。

最后让任何游戏实体派生另一实体，因此我们在 GameObject 类中增加了代码，可以增加和取得派生实体，见清单 13.17。

清单 13.17 GameObject 中的派生

```
private List spawns;

...
```

```
/**
    派生一个实体，使 GameManager 可以用 getSawns()取得
*/
    protected void addSpawn(GameObject object) {
        if (spawns == null) {
            spawns = new ArrayList();
        }
        spawns.add(object);

    }
    /**
        返回派生实体表（抛射弹，爆炸物，等等），没有派生实体时返回 null
    */
    public List getSpawns() {
        List returnList = spawns;
        spawns = null;
        return returnList;
    }
```

更新之后，游戏实体管理器应调用 getSawns()方法和将任何派生实体加进所管理的实体表中。

13.8　综合起来

除了 Bot 被击中爆炸之外，还要做几个工作，生成使用人工智能 Bot 的演示程序：

- 可以对每个状态显示不同实体动画。例如，Bot 散步、跑步、包抄、瞄准和射击时可以用不同动画。
- Bot 受伤时要有一些视觉反馈，使游戏者知道打伤了 Bot。
- 由于 Bot 能攻击游戏者，因此游戏者的状态也要有一些视觉反馈。例如，可以使用生命力条和受伤指标。

本章使用与前一章相同的简单棱形 Bot，使实体动画更简单。我们还要生成画生命力条的简单重叠框架。但首先要总结一下要用的不同 AI Bot 属性。

13.8.1　大脑

Bot 的所有运动模式和其他属性存放在一个简单 Brain 类中，见清单 13.18。每个 AIBot 有一个 Brain 类实例。

清单 13.18　Brain.java

```
public class Brain{
```

```
    public PathFinder attackPathFinder;
    public PathFinder dodgePathFinder;
    public PathFinder aimPathFinder;
    public PathFinder idlePathFinder;
    public PathFinder chasePathFinder;
    public PathFinder runAwayPathFinder;
    //每种战斗状态的概率（总和应为 1）
    public float attackProbability;
    public float dodgeProbability;
    public float runAwayProbability;

    public long decisionTime;
    public long aimTime;
    public long hearDistance;

    public void fixProbabilites(){
        //让总和  ==1
        float sum = attackProbability + dodgeProbability +
            runAwayProbability;
        if (sum > 0){
            attackProbability /=sum;
            dodgeProbability /=sum;
            runAwayProbability /=sum;
        }
    }
}
```

这个基类只包括每个大脑属性的字段，这里的惟一功能是修正概率，使其总和为 1。

Brain 类中还可以增加许多内容。可以将类扩展成包括其他 AI 属性，如速度和最大健康指数。还可以让系统更完善些，使 Bot 有一定脑力防止特定武器。

13.8.2 健康与死亡

当然，Bot 能够运动与攻击之后，还要能够破坏。

在上一章的演示程序中，我们采用一击即溃规则，抛射弹击中 Bot 时即把它破坏，使它立即消失。游戏中应该做得更真实些。例如：

- 使有些 Bot 要打击更多次才能破坏。
- 使有些 Bot 对某些武器更脆弱。例如，一个 Bot 能够抵抗激光弹，但无法抵抗火箭弹。
- 击中时要让 Bot 进入受伤状态，停止一会儿。另外，Bot 处于受伤状态时，短期内对其进一步伤害应该更脆弱。

- 根据 Bot 的健康状况进行不同决策。健康 Bot 可能很勇敢，而体弱的 Bot 则见到游戏者就跑。
- Bot 破坏之后，不要让它立即消失，而要让它躺一会儿，几秒钟后再让它消失。有些游戏让死掉的 Bot 在整个游戏中一直躺在那儿。另一种方法是让机器人炸成废物，在地上留下爆炸痕迹，或化成一阵青烟消失。
- 破坏的 Bot 可能丢掉一些东西，如弹药和能量，游戏者可以把它捡起来。
- 游戏中要在游戏者与坏人之间找到适当的平衡，随着游戏的继续，使敌人越战越勇。
- 一定要有创造性。

本章生成简单的健康与死亡程序。每个 Bot 开始的生命力都是 100 点，游戏者的武器会产生随机伤害，减掉 40 到 60 个生命力。记住，游戏者发射的抛射弹是个游戏实体，因此可以用碰撞代码检查抛射弹是否击中任何物体。

首先在 Bot 中增加两个状态，表示 Bot 被击中和死亡：

```
public static final int WOUNDED_STATE_HURT = 6;
public static final int WOUNDED_STATE_DEAD = 6;
```

Bot 被击中后，让它在几秒钟内保持受伤状态。同样，Bot 的 AI 状态为死亡时，让这个状态保持几秒钟，然后再把 Bot 从游戏实体管理器中清除。

首先要在 AIBot 类中增加一些健康函数，见清单 13.19。

清单 13.19 AIBot 的健康方法

```
private static final float DEFAUAT_MAX_HEALTH = 100;
private static final float CRITICAL_HEALTH_PERCENT = 5;

private float maxHealth;
private float health;

...
protected void setHealth(float health){
        this. Health = health;
}
/**
     给这个 Bot 增加一定的生命力。如果数值小于 0，则 Bot 状态设置为 WOUNDED_STATE_HURT
 */
            public void addHealth(float amount) {
                if (amount < 0) {
                    if (health <= 0 || aiState == WOUNDED_STATE_HURT) {
                        return;
                }
                    setAiState(WOUNDED_STATE_HURT, null);
```

```
        }
        setHealth(health + amount);
    }
    /**
        生命力太低时返回真值（小于 CRITICAL_HEALTH_PERCENT）
    */
        public boolean isCriticalHealth() {
            return (health / maxHealth < CRITICAL_HEALTH_PERCENT/100);
        }
```

所有动作发生时都调用 addHealth()方法。

注意：这个方法减少生命力，将不在受伤状态的 Bot 变成受伤状态。

IsCriticalHealth()方法在生命力太低时返回真值，这时可以让它跑开。

下面要根据 Bot 的生命力和受伤状态进行一些决策。要保证 Bot 在短时间内保持受伤状态，要在 Bot 死亡一会儿之后将其破坏。这些代码在清单 13.20 中。

清单 13.20　AIBot 的受伤与死亡方法

```
public void update(GameObject player, long elapsedTime) {

    elapsedTimeSinceDecision+=elapsedTime;
if (aiState == WOUNDED_STATE_DEAD) {
        //五毫秒破坏 Bot
        if (elapsedTimeInState >= 5000) {
            setState(STATE_DESTROYED);
        }
        return;
}
else if (aiState == WOUNDED_STATE_HURT) {
        //经过 500 毫秒后切换到闲置或死亡状态
        if (elapsedTimeInState >= 500) {
                if (health <= 0) {
                    setAiState(WOUNDED_STATE_DEAD, player);
                    return;
                }
                else {
                    aiState = DECESION_READY;
                }
            }
            else {
                return;
```

```
            }
        }
        //生命力太低时跑掉
        if (isCriticalHealth() &&
            brain.runAwayPathFinder != null)
        {
            setAiState(BATTLE_STATE_RUN_AWAY, player);
            return;
        }
        ...
    }
```

代码在 AIBot 的 update()方法中有一些决策程序。Bot 受伤之后，经过 500 毫秒，它转入闲置状态，如果 Bot 的生命力为 0，则转入死亡状态。转入死亡状态时，过五秒钟后破坏。最后，如果 Bot 的生命力太低，则背向游戏者跑掉。

注意： 这里不管如何都让 Bot 在五秒后消失。Bot 突然消失似乎不好看。另一种方法是让 Bot 离开屏幕时才消失，使游戏者不会注意到它消失。还有许多游戏在 Bot 消失时显示一些爆炸效果。前面曾介绍设置 AI 状态时如何设置相应 PathFinder。这时还要改变实体的动画。清单 13.21 扩展 AIBot 中的 setAIState()方法。在这个代码中，Bot 受伤时，停止移动并转一会儿，好像伤得使它晕头转向了。

清单 13.21 扩展 AIBot 中的 setAIState()方法

```
protected void setAiState(int aiState, GameObject player) {
    if (this.aiState == aiState) {
        return;
    }

    this.aiState = aiState;

    elapsedTimeInState = 0;
    PathFinder lastPattern = pathFinder;
    Vector3D playerLocation = null;
    if (player != null) {
        playerLocation = player.getLocation();
    }

    // 更新路径
    switch (aiState) {
```

```
            case BATTLE_STATE_ATTACK:
                setPathFinder(brain.attackPathFinder);
                setFacing(playerLocation);
                break;
            case BATTLE_STATE_DODGE:
                setPathFinder(brain.dodgePathFinder);
                setFacing(null);
                break;
            ...
            case WOUNDED_STATE_HURT:
                setPathFinder(null);
                setFacing(null);
                getTransform().stop();
                getTransform().setAngleVelocityY(
                    MoreMath.random(0.001f, 0.05f),
                    MoreMath.random(100, 500));
                break;
        }
    }
```

这样就有了 AI Bot 演示程序所要的一切。下面要显示游戏者的生命力，因为 AI Bot 会攻击，使游戏者的生命力减弱。为此要增加飞行仪表盘。

据我所知，飞行仪表盘（heads-up 或 HUD）一词来自某些汽车仪表板。在一般汽车中，驾驭员要朝下才能看到车速、油量、里程、水量和仪表板上的其他数据。而在有飞行仪表盘的车上，则把仪表板投影到档风板上或在赛车驾驭员头盔上显示，使驾驭员可以在开车时保持抬姿势。

飞行仪表盘的主要目的是提供游戏本身无法显示的游戏状态信息。飞行仪表盘中常见的信息包括：

- 生命力、弹药量，余下的寿命，等等。
- 游戏消息（包括选择项目时的显示图标）。
- 当前活动的充电器（如超级生命力或能量损失）。
- 余下的游戏时间。

游戏中的飞行仪表盘通常在视图窗口上方写字幕，通常独立于分辨率，即不管屏幕分辨率如何，都在屏幕上保持相同尺寸。相反，分辨率相关飞行仪表盘如果是为 640×480 屏幕设计的，则在 1024×768 之类高分辨率屏幕上变得很小。

这里采用分辨率独立实现方法。首先生成简单 Overlay 接口，见清单 13.22。

清单 13.22　Overlay.java

```
public interface Overlay{
```

```
    /**
        上次更新后经过指定时间更新这个字幕
    */
        public void update (long elapsedTime);
    /**
        在帧上画字幕。ViewWindow 指定视图窗口边界(通常为整个屏幕)。可以用
        g.getDeviceConfiguration().getBounds()方法取得屏幕边界
    */

        public void draw(Graphics2D g, ViewWindow viewWindow);

    /**
        启用这个字幕时返回真值（应该画出）
    */
        public boolean isEnabled();
}
```

字幕和游戏实体一样更新，在每帧画完之后画上，使其在上方显示。

例如，我们生成一个简单飞行仪表盘，显示游戏者的生命力（数字和显示条）。清单 13.23 的 HeadsUpDisplay 类是分辨率独立实现方法，显示游戏者的生命力，而且是动画：比游戏者实际生命力稍后显示生命力。

清单 13.23 HeadsUpDisplay.java

```
public class HeadsUpDisplay implements Overlay {

    // 每秒将生命力显示增加 20 点
    private static final float DISPLAY_INC_RATE = 0.04f;

    private Player player;
    private float displayedHealth;
    private Font font;

    public HeadsUpDisplay(Player player) {
        this.player = player;
        displayedHealth = 0;
    }

    public void update(long elapsedTime) {
        // 一次增加或减少一些 displayedHealth ，而不只是设置成游戏者的生命力
```

```
        float actualHealth = player.getHealth();
        if (actualHealth > displayedHealth) {
            displayedHealth = Math.min(actualHealth,
                displayedHealth + elapsedTime * DISPLAY_INC_RATE);
        }
        else if (actualHealth < displayedHealth) {
            displayedHealth = Math.max(actualHealth,
                displayedHealth - elapsedTime * DISPLAY_INC_RATE);
        }
    }

    public void draw(Graphics2D g, ViewWindow window) {

        // 设置字体（对这个视图窗口缩放）
        int fontHeight = Math.max(9, window.getHeight() / 20);
        int spacing = fontHeight / 5;
        if (font == null || fontHeight != font.getSize()) {
            font = new Font("Dialog", Font.PLAIN, fontHeight);
        }
        g.setFont(font);
        g.translate(window.getLeftOffset(), window.getTopOffset());

        // 绘制生命力值（数字）
        String str = Integer.toString(Math.round(displayedHealth));
        Rectangle2D strBounds = font.getStringBounds(str,
            g.getFontRenderContext());
        g.setColor(Color.WHITE);
        g.drawString(str, spacing, (int)strBounds.getHeight());

        // 绘制生命力条
        Rectangle bar = new Rectangle(
            (int)strBounds.getWidth() + spacing * 2,
            (int)strBounds.getHeight() / 2,
            window.getWidth() / 4,
            window.getHeight() / 60);
        g.setColor(Color.GRAY);
        g.fill(bar);

        //绘制生命力条的加亮部分
```

```
            bar.width = Math.round(bar.width *
                displayedHealth / player.getMaxHealth());
            g.setColor(Color.WHITE);
            g.fill(bar);
        }

        public boolean isEnabled() {
            return (player != null &&
                (player.isAlive() || displayedHealth > 0));
        }
    }
```

字体尺寸和生命力条尺寸由视图窗口尺寸确定，此外，这个飞行仪表盘其实很简单，只是画一个文本字符串和两个矩形而已。但生命力条有很好的动画效果。

本章的演示程序（AIBotText）还有一个字幕，作为消息队列。这里，消息队列显示 AI Bot 状态的不同改变，放在屏幕左上角，有助于检查 AI Bot 的行为，使开发人员了解 Bot 在想什么，包括 Bot 是否能听到和看到游戏者。图 13-9 显示了飞行仪表盘的例子。

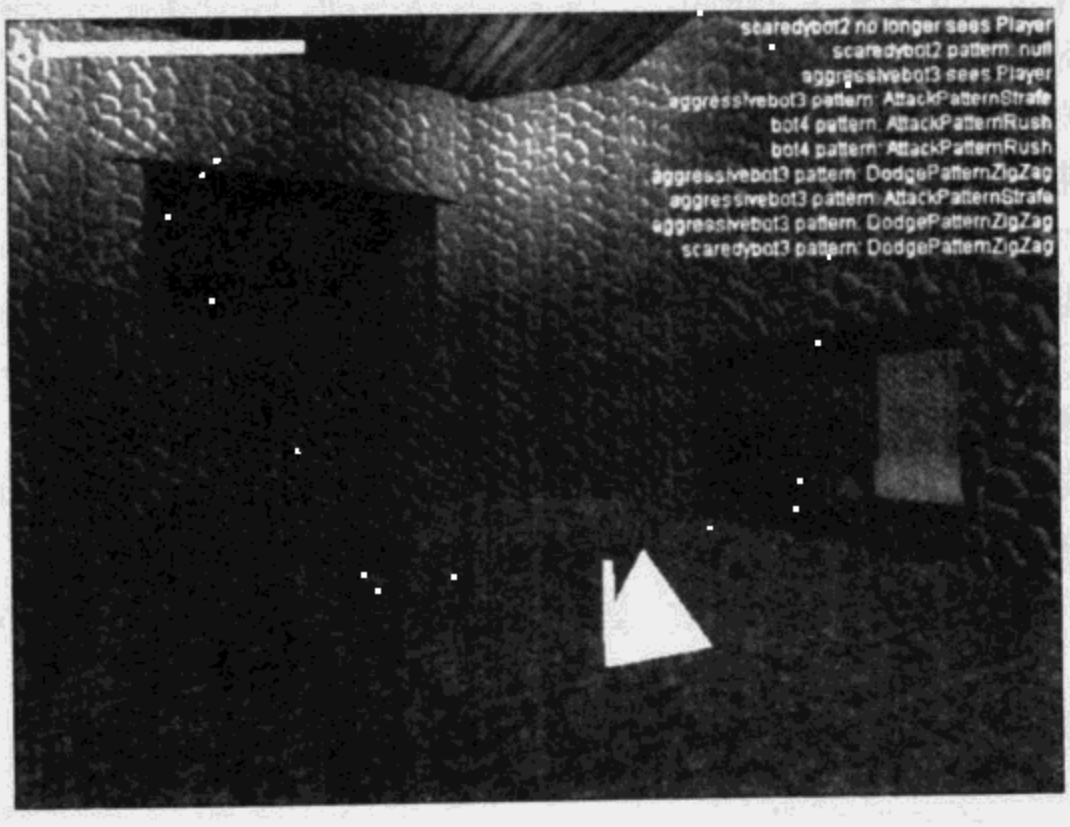


图 13-9　飞行仪表盘的例子：左上角是生命力条，右上角是消息队列，消息队列显示 AI Bot 调试消息

注意：飞行仪表盘只在必要时才用。如果能在游戏本身显示游戏者的生命力，就没必要用飞行仪表盘。例如，在原先的 Mario 游戏中，Mario 的生命力直接与其大小有关。大 Mario 要打击两次才死，而小 Mario 只在打击一次就死。游戏没有用飞行仪表盘显示大小，因为这是一目了然的。

生命力也不一定只能用生命力条显示，有些游戏也用饼形图或心脏图标。读者可以想出更有创造性的生命力表示方法。例如，Doom 游戏显示脸形图，受伤越重，脸上的血越多。

最后，记住，飞行仪表盘也不一定要总是放在屏幕上。其中一部分可以只在发生重大改变时才显示，如游戏者获得更多生命力时或获得新的充电器时。这时，飞行仪表盘可以根据需要在屏幕上滚开或合上。

13.9 进化

最后一个游戏人工智能概念是进化。进化 AI Bot 就是随着时间逐渐改变 Bot 的行为，越来越适应游戏者的攻击和防御战术。

游戏中增加进化的好处是 Bot 更能对付各个用户的游戏风格。每个用户用不同战术玩游戏，进化 Bot 能够相应调整，越来越难对付。

可以用几种方式在游戏中实现进化。可以动态进化，也可以让 Bot 编写修改属性，更好地攻击游戏者。例如：

- Bot 可以改变攻击和瞄准模式，找出命中率更高的模式。
- Bot 可以用不同瞄准模式同时发射几枚“虚”弹，看看哪枚更容易击中游戏者。利用 Bot 能跟踪的虚弹而不用实弹使 Bot 可以同时测试几种不同的瞄准模式，而不必一次发射一枚实弹之后才瞄准。
- Bot 可以动态调整其他属性，如速度、决策时间、听觉距离和决策之间的时间量。

另一种实现进化的办法是老办法：通过再生和基因突变。当然，我们不能让 Bot 交配和生产，但可以实现遗传和变异的思想。方法如下：

- 只复制“最佳” Bot。
- 子代与父代稍有变异，即大致与父代相同，但有几个属性改变。

变异的思想是，如果生成坏脑子，则会在基因库的低端，不会遗传。而好脑子在基因库的高端，不会遗传。因此，随着时间的推移，基因库的高端越来越好，得到更好的 Bot。下面是本章实现进化的方法：

- Bot 破坏时，记录其性能水平，就是对游戏者造成的损伤力。Bot 的脑子保存在基因库中。
- 然后用新脑再生 Bot，或者是基因库中最佳性能的脑子，或者是最佳性能子代变种。

但是，实现进化之前，要补充一个概念：Bot 再生。

13.9.1 再生

再生是许多游戏中常用的方法。这个方法使 Bot 死亡之后能够再生，使自己恢复到开始位置和状态。这在需要生成无限提供的坏蛋时有用。

清单 13.24 的 AIBot 类实现再生。Bot 死亡后不是破坏，而可以再生，完全恢复状态。

清单 13.24 AIBot 的再生代码

```
/**
    返回真值表示 Bot 死亡后可以再生
*/
```

```
public boolean isRegenerating() {
    return isRegenerating;
}
/**
     设置 Bot 死亡后可否再生
*/
public void setRegenerating(boolean isRegenerating) {
    this.isRegenerating = isRegenerating;
}
/**
     让这个 Bot 再生，恢复到开始位置和状态
*/
protected void regenerate() {
        setHealth(maxHealth);
        setState(STATE_ACTIVE);
        setAiState(DECESION_READY, null);
        getLocation().setTo(startLocation);
        getTransform().stop();
        setJumping(false);
        setPathFinder(null);
        setFacing(null);
        //让游戏实体管理器知道这个实体再生（因此旧位置不进行与新位置的碰撞探测）
    addSpawn(this);
}
public void update(GameObject player, long elapsedTime) {
    ...
elapsedTimeInState+=elapsedTime;

//记录第一个地址
if (startLocation == null) {
      startLocation = new Vector3D(getLocation());
}
//死亡 5 秒钟后再生
if (aiState == WOUNDED_STATE_DEAD) {
          if (elapsedTimeInState >= 5000) {
                    if (isRegenerating()) {
                              regenerate();
                    }
                    else {
```

```
                    setState(STATE_DESTROYED);
                }
            }
            return;
        }
    ...
}
```

update()方法修改成在 Bot 死亡几秒钟后调用 regenerate()方法，只在 Bot 有再生能力（isRegenerating()）。regenerate()方法复位 Bot，这里将其位置设置为原先诞生的位置（startLocation）。

regenerate()方法中还要调用 addSpawn（）方法，标为派生。这相当于向游戏实体管理器发一个通知，表示在再生，不要在此时进行碰撞探测。如果进行碰撞探测，则引擎会以为 Bot 从死亡位置移到再生位置，而不是在再生位置重现，Bot 可能粘在这两地之间的墙上或其他实体上。不进行碰撞探测表示 Bot 可以在诞生位置正确重现。

13.9.2 进化 Bot

死了只遗传最佳脑子，要跟踪哪个是最佳脑子。可以用许多不同图素确定最佳脑子，但这里只检查 Bot 的平均杀伤力。

清单 13.25 的 BrainStat 类是 Brain 的子类，踊跃脑子的代数和杀伤力。

清单 13.25　EvolutionGenePool 的 BrainStat 内类

```
private class BrainStat extends Brain implements Comparable {

    long totalDamageCaused;
    int numBots;
    int generation;
    /**
        取得这个脑子的平均杀伤力
    */
    public float getAverageDamageCaused() {
        return (float)totalDamageCaused / numBots;
    }
    /**
        Bot 破坏之后报告这个脑子的 Bot 的平均杀伤力
    */
    public void report(long damageCaused) {
        totalDamageCaused+=damageCaused;
        numBots++;
    }
```

```
/**
    让这个脑子变异。指定 mutationProbability 是每个脑子属性变成不同值（变异）的概率
*/
public void mutate(float Probability) {
        ...
}
/**
    如果这个脑子比指定实体（也是个脑子）适合更大杀伤，则返回较小值
*/
public int compareTo(Object obj) {
    BrainStat other = (BrainStat)obj;
    if (this.numBots ==||other.numBots ==0){
            return (other.numBots - this.numBots );
    }
    else{
    return (int) MoreMath.sign(
            other.getAverageDamageCaused() -
            this.getAverageDamageCaused());
  }
 }
}
```

BrainStat 类实现 Comparable 接口，使脑子表很容易从好到差排序。可以用许多其他标准确定哪个脑子更好，但这里使用的杀伤力很不错。

Bot 的抛射弹击中游戏者时，抛射弹要向 Bot 报告造成的杀伤力。为此要在实现 EvolutionBot 时处理。

这里没有显示 mutate()方法，它只是在发生某个随机情况时改变每个脑子属性。例如，如果 mutationProbability 为 0.10，则每个属性有 10%的改变机会。改变瞄准时间的代码如下：

```
if (MoreMath.chance (mutationProbability)){
    aimTime = MoreMath.random(300,2000);
}
```

脑子变异时，其代数增加。

另外，这里没有显示 clone()方法，因为这个方法很简单。

下面是所有这些脑子的存储机制，放在清单 13.26 的 EvolutionGenePool 类中。

清单 13.26　EvolutionGenePool.java

```
public class EvolutionGenePool {

    private static final int NUM_TOP_BRAINS = 5;
    private static final int NUM_TOTAL_BRAINS = 10;
```

```
private List brains;
...

/**
    从基因库中取得新脑。脑子是最佳脑子或变异后的最佳脑子
*/
public Brain getNewBrain() {

    // 50% 机会生成新的变异脑子
    if (MoreMath.chance(.5f)) {
        BrainStat brain =
            (BrainStat)getRandomTopBrain().clone();

        // 10% 到 25% 机会改变每个属性
        float p = MoreMath.random(0.10f, 0.25f);
        brain.mutate(p);
        return brain;
    }
    else {
        return getRandomTopBrain();
    }
}

/**
    取得随机最佳性能脑子
*/
public Brain getRandomTopBrain() {
    int index = MoreMath.random(NUM_TOP_BRAINS-1);
    return (Brain)brains.get(index);
}

/**
    告知破坏了指定脑子的生物。记录脑子状态。如果脑子状态在最佳总脑子中，则保留这个
    脑子
*/
public void notifyDead(Brain brain, long damageCaused) {
    // 更新脑子状态
    if (brain instanceof BrainStat) {
```

```
                BrainStat stat = (BrainStat)brain;
                // 报告杀伤力
                stat.report(damageCaused);

                // 排序和整理表
                if (!brains.contains(stat)) {
                    brains.add(stat);
                }
                Collections.sort(brains);
                while (brains.size() > NUM_TOTAL_BRAINS) {
                    brains.remove(NUM_TOTAL_BRAINS);
                }
            }
        }
    }
```

其总共跟踪 10 个脑子，只有 5 个最佳脑子可以有后代。

Bot 死亡时，调用 notfyDead()方法，让基因库知道用指定脑子造成多大杀伤力。生成或再生新 Bot 时，调用 getNewBrain（）方法取得脑子。这个方法有 50%机会生成新的变异脑子，50%机会只返回一个最佳脑子。

最后，清单 13.27 中生成 EvolutionBot 类。这个 Bot 是 AIBot 的子类，完成再生的所有功能，并向基因池报告造成多大杀伤力。

清单 13.27　EvolutionBot.java

```
public class EvolutionBot extends AIBot {

    private EvolutionGenePool genePool;
    private long damagedCaused;

    public EvolutionBot(PolygonGroup polygonGroup,
        CollisionDetection collisionDetection,
        EvolutionGenePool genePool, PolygonGroup blastModel)
    {
        super(polygonGroup, collisionDetection,
            genePool.getNewBrain(), blastModel);
        this.genePool = genePool;
        setRegenerating(true);

    }
```

```
        public void regenerate() {
            genePool.notifyDead(brain, damagedCaused);
            brain = genePool.getNewBrain();
            damagedCaused = 0;
            super.regenerate();
        }

        public void notifyHitPlayer(long damage) {
            damagedCaused+=damage;
        }
    }
```

regenerate（）方法只是覆盖 AIBot 方法，进行所要的其他工作。

这样就完成了涉及遗传与变异的进化 Bot。EvolutionBotDemo 演示程序代码首先对每个 Bot 指定基因库中的随机脑子，并在 Bot 死亡之后再生。这个演示程序中的 Bot 无限再生。

此外，游戏退出时，向控制台打印前五个脑子的属性，以便了解哪些是最佳脑子。游戏时间越长，这个表中越可能包含真正有价值的脑子。

注意：游戏者对进化有很大影响。游戏者可能喜欢先杀掉某种 Bot。例如，进行包抄攻击模式的 Bot 很危险，游戏者可能喜欢先杀掉这种 Bot，使其他 Bot 有更大杀伤力，从而影响进化。另外，有些 Bot 在开始位置的战术优势更差，游戏者可以停在再生位置，迅速将 Bot 消灭，即使这些 Bot 是真正聪明的。

前面曾介绍过，Bot 造成的杀伤力不是选择最佳脑子的惟一标准。还可以选择长寿的、射击命中率高的和避开子弹数多的脑子。最佳脑子最好能组合这些优良特性。

13.9.3 演示改进

和平常一样，演示程序中还可以增加许多特性。游戏演示程序还可以使用生命力与弹药充电器、不同武器和“锁定”目标，使游戏者更容易攻击 Bot。

Bot 还可以使用更多模式，包括更好的逃跑模式和更智能的模式，通过检查环境确定最佳模式（例如避免撞墙）。

最后，游戏者可能失去所有生命力而死亡。这时可以使用某种死亡序列。可以用飞行仪表盘中的杀伤指示器显示游戏者何时被击中，只要用几毫秒时间在屏幕上闪烁几色即可。

13.10 进化

还可以实现更多游戏 AI 技术，主要取决于生成的游戏类型。

如果建立二维平台游戏，则人工智能很少，生物大部分时间采用简单模式。但一定要让生物有不同程度的智能。例如，有些生物能够探测平台边沿，躲避平台边沿，而有些则会掉下去。

对第一个称视角射击游戏，Bot 可能花一些时间预测游戏者的运动，相应调整模式。

Bot 可以在预测的游戏者方向射击，而不是射到发射时游戏者所在位置。Bot 还可以“学习”游戏者的常见模式。

除了视觉与听觉之外，有些 Bot 还有其他感觉能力。例如，Bot 可以通过嗅觉跟踪游戏者，有些 Bot 还有热感应或 X 光透视，帮助跟踪游戏者。

可以在游戏中扩展进化，长时间进行再生与变异模拟，然后只在最后游戏中包括最聪明的 Bot。

小组人工智能

在战略游戏中，通常要把 AI Bot 安排成领导层次。小组领导向部队发出命令，对整个小组建立战术优势，而不是每个部队自己决策。这样可以使游戏更难。如果团队领导被杀，则这个部队会混乱，或把一个部队提升为团队领导。

还可以使用基于团队的模式，如集体进攻一个点或从各个方向完全包围游戏者。

集体行动可以显示一些精彩的人工智能行为，也常见于一群鸟和鱼的人工智能行为。集体行动算法要让每个 Bot 靠近组，使每个 Bot 相互间有一定距离（具有“个人空间”），让每个 Bot 按小组平均方向走。小组可以显式定义，也可能只是 Bot 的近邻。

另外，部队也可以备份，这时可能看到部队备份，也可能只是随机显示更多部队。

除了层次之外，附近的部队还可以相互交流，讨论最佳策略。

13.11 小结

本章介绍了大量有用的游戏人工智能，如视觉与听觉、状态机、概率和进化，还建立了一个演示程序，Bot 可以攻击游戏者，演示程序中增加一些基本游戏要素，如 Bot 的生命力与死亡、再生和显示飞行仪表盘的简单字幕框架。

前面曾介绍过，实现人工智能的方法取决于游戏的需要。很大一部分工作是寻找一种平衡，使游戏既好玩又有难度。有时只能通过不断摸索试验来找到这种平衡。因此，诸多多试验，使坏家伙更聪明，使游戏中有更精彩的人工智能特性。

第 14 章　游戏脚本

关键问题

- 关于脚本
- 实现接触与释放通知
- 游戏实体监听器
- 脚本
- 延迟事件
- 综合起来
- 改进
- 小结

你走入一个房间，转动开关。一秒钟后，门开了，灯光从里面射出来。你还没来得及走进门，两秒钟后，门又关上了，房间又暗下来了。怎么回事？这个门只打开几秒钟，因此你要转动开关，冲进门内。

游戏中怎么实现这个场面？本章要解决这个问题。

典型游戏中每个关卡只有一定量的惟一密码，用来与环境交互，这些惟一密码给游戏增加了巨大价值，使游戏更有趣、更好玩、更刺激。

但是，这些独特环境交互不一定是编程人员生成的，通常是关卡设计员设计出来的。因此，本章要介绍如何将游戏实体连接外部脚本，外部脚本很容易编写。这样，关卡设计员不必知道如何实现接口或编译，只需要知道如何编写简单功能。

既然任何人都能生成关卡和编写脚本，用户就可以生成自己的内容和建立自己的游戏修订（mod）。修订就是修改现有游戏，例如，Counter-Strike 是 Half-Life 的修订。

除了脚本之外，本章还要介绍触发器（使事件发生的东西）和延迟事件（在一定延迟之后完成任务）。

14.1　关于脚本

游戏脚本系统至少要生成几个项目：惟一对象名、事件监听器、嵌入脚本语言解释器和延迟事件。详细内容如下：

- 最好每个游戏对象都是来自同一祖先类。这样对象之间更容易交互，因为它们有一组共同方法。我们已经在本书提供了一个公共祖先类 GameObject。
- 要按惟一 ID 引用每个对象，最好用可读名称。这样就可以使用下列命令：游戏者碰到 blueSwitch 时，打开 doorToGrandHall。可以用 GameObject 的 getName()方法得到惟一名称，可以在贴图文件中定义对象名。
- 要监听某些事件并基于这些事件进行相应操作。这样，游戏者碰到 blueSwitch 时，监听器可以得到通知，并进行必要的操作。我们的 GameObject 方法

notifyObjectCollision()已经提供了一个简单通知机制，但还要进行扩展。

- 游戏者（或另一对象）首次接触和最终释放（停止接触）某个对象时要有接触与释放，目前只有碰撞通知。
- 同样，要能执行延迟操作，如经过一定时间后关门。
- 最后，要将系统设计成关卡设计员能方便地编码，应尽量灵活和具有创造性。本章要把游戏连接到脚本语言。这个方法简单而灵活，可以加速开发。

首先要升级事件通知，然后要转入脚本与延迟事件。

14.2　实现接触与释放通知

游戏中，游戏者或另一实体碰到另一实体时，通常要采取某些操作。例如，游戏者接触到开关时要开门，再次接触到开关时要关门。

本章前面的游戏引擎中，实体碰上另一实体时，接收所有碰撞事件，本例中，只要游戏者碰到开关，门总是不断地开或不断关，直到游戏者将它释放，尽管不是我们所要的效果。

在这种情形中，我们感兴趣的只是第一个碰撞事件：游戏者何时首次碰到开关。换句话说，除了接触通知外，还要有一个释放通知，表示游戏者何时停止接触一个实体。

图 14-1 的例子显示了不同事件何时发生。图中游戏者在第 2 帧碰到一个框，发送一个接触和碰撞通知。第 3 帧和第 4 帧游戏者仍然走向框，因此发出碰撞通知。第 5 帧，游戏者停止，不发通知。最后，第 6 帧，游戏者离开框，发出释放通知。

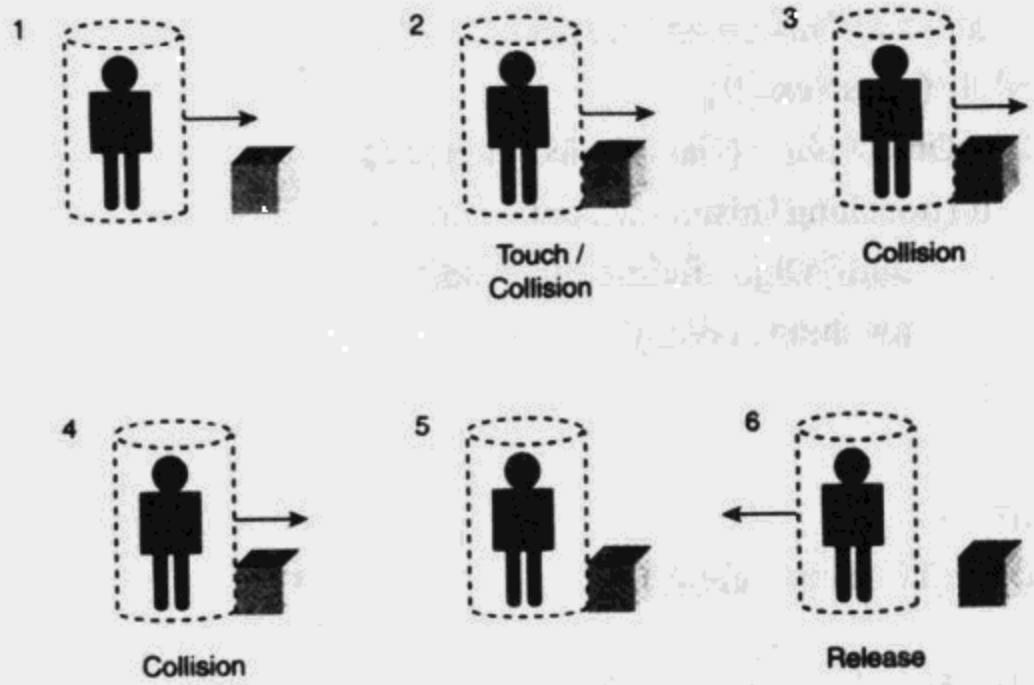


图 14-1　游戏者在第 2 帧发出接触通知，第 6 帧发出释放通知，第 2、3、4 帧发出碰撞通知

记住，有时一个实体可能同时接触好几个实体。

下面要用代码实现接触与释放通知。每个游戏实体保存一个接触的实体表。新实体加进表中时，发出接触通知；删除一个实体时，发出释放通知。这个代码见清单 14.1。

清单 14.1　GameObject.java 中的接触与释放代码

```
private List touching = new ArrayList();
```

```
private List touchingThisFrame = new ArrayList();

...
public void notifyObjectCollision(GameObject otherObject){
        touchingThisFrame.add(otherObject);
}
/**
     移动实体并检查碰撞后，调用这个方法，发出接触与释放通知
*/
public void sendTouchNotifications(){
          //发出释放通知
            Iterator i = touching.iterator();
            While (i.hasNext()){
                  GameObject obj = (GameObject) i. next();
                  If (!touchingThisFrame.contains(obj)){
                        notifyObjectRelease(this ,obj);
                        i.remove();
                  }
          }
           //发出接触通知
             i= touchingThisFrame.iterator();
             while (i.hasNext()){
            GameObject obj = (GameObject) i. next();
                if (!touchingThisFrame.contains(obj)){
                      notifyObjectRelease(this ,obj);
                      touching.add(obj);
                }
            }
            //清理
            touchingThisFrame.clear();
}
```

在这个代码中，另一个 touchingThisFrame 表是上一帧中游戏实体接触的实体表。移动实体并检查碰撞后，游戏实体管理器调用 sendTouchNotifications()方法。这个方法将 touchingThisFrame 表与 touching 表进行比较。如果实体在 touching 表中而不在 touchingThisFrame 表中，则发出实体的释放通知；同样，如果实体在 touchingThisFrame 表中而不在 touching 表中，则发出实体的释放通知，并将实体加进 touching 表中。

注意：sendTouchNotifications()方法只在实体移动时才调用。因此，只有实体移动时才跟踪所接触的东西，图 14-1 的框并不知道碰到什么。同样，在特定帧中没有移动的实体也不在该帧中发任何通知，像图 14-1 的第 5 帧。

触发器

除了接触与释放某些实体外，有时还要让地图的某些区域设置触发器事件。例如，图 14-2 用隐藏触发器区开门和关门。游戏者进入触发区时（接触），门打开；游戏者离开触发区时（释放），门关掉。

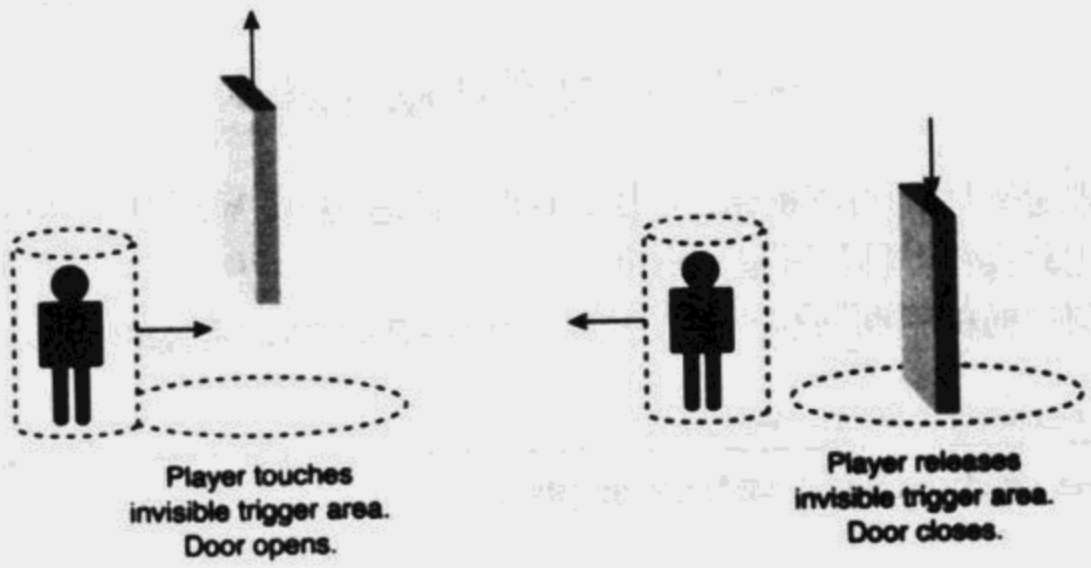


图 14-2　用隐藏触发区开门和关门

游戏者进入触发区时（接触）门打开，游戏者离开触发区时（释放），门关掉。

如果这里用圆柱进行碰撞探测，因此可以在贴图文件中将触发器定义为实体，只是指定半径而已：

```
v 512   32   1150
trigger   doorTrigger  - 1 256
```

贴图文件中的这个命令在位置（512,32,1150）处生成游戏实体 doorTrigger，半径为 256。

触发器只在一种情况下作为特殊游戏实体：碰撞探测。我们要让游戏者和其他实体走过触发器，而不是像第 11 章“碰撞探测”中一样绕触发器滑动。清单 14.2 显示了如何不绕触发器滑动，同时又发送碰撞通知。

清单 14.2　CollisionDetectionWithSliding.java 中的触发器

```
protected boolean handleObjectCollision(GameObject objectA,
        GameObject objectB,float distSq, float minDistSq,
        Vector3D oldLocation)
{
        objectA.notifyObjectCollision(objectB);
        //如果 objectB 没有多边形，则是个触发器区
        if (objectB.getPolygonGroup().isEmpty()){
            return   false;
        }

          ...
        //进行滑动
```

```
}
```

这里如果对象是触发区，则没有多边形，并通知实体碰撞，然后像没有发生碰撞一样从方法中返回，而不进行任何滑动。

这样就完成了触发器。但目前这些修改什么也不干，每个实体要在子类中进行所要的特殊操作。下面要生成游戏实体监听器，监听某些实体的通知并根据这些通知采取操作。

14.3 游戏实体监听器

我们已经熟悉事件/监听器机制，例如 AWT 的 MouseListener 接口接收鼠标事件。由于它是个接口，因此实体可以监听鼠标事件。

游戏实体通知的机制也差不多。首先生成 GameObjectEventListener 接口，见清单 14.3。

清单 14.3 GameObjectEventListener.java

```
package com.brackeen.javagamebook.scripting;

import com.brackeen.javagamebook.game.GameObject;

/**
    Interface 接收 GameObject 通知事件的接口见 GameObject.addListener()
*/
public interface GameObjectEventListener extends EventListener {

    public void notifyVisible(GameObject object, boolean visible);

    public void notifyObjectCollision(GameObject object,
        GameObject otherObject);

    public void notifyObjectTouch(GameObject object,
        GameObject otherObject);

    public void notifyObjectRelease(GameObject object,
        GameObject otherObject);

    public void notifyFloorCollision(GameObject object);

    public void notifyCeilingCollision(GameObject object);

    public void notifyWallCollision(GameObject object);
}
```

这个接口的方法针对每种不同通知：

- 实体、地板、天花板与墙的碰撞。
- 实体接触与释放通知。
- 显示通知（实体在视图中显示或消失）。

当然，可以不限于这些事件通知，还可以增加更多事件通知，如 notifyMoving()、notifyStopped()或 notifyDestroyed()。

对游戏实体采用新的监听器之前，要有一个简便方法让游戏实体有多个监听器。如果游戏实体类有普通动作，而游戏实体本身又有独特动作，则要有多个监听器。作为监听器的类可以向其他监听器派遣事件，类似于清单 14.4 的 AWTEventMulticaster 类。

清单 14.4 GameObjectEventMulticaster.java

```
package com.brackeen.javagamebook.scripting;

import java.util.*;
import com.brackeen.javagamebook.game.GameObject;

/**
    将多点广播 GameObject 通知发送到多个监听器
*/
public class GameObjectEventMulticaster
    implements GameObjectEventListener
{

    private List listeners;

    public GameObjectEventMulticaster(GameObjectEventListener l1,
        GameObjectEventListener l2)
    {
        listeners = new LinkedList();
        addListener(l1);
        addListener(l2);
    }

    public void addListener(GameObjectEventListener l) {
        if (l != null) {
            listeners.add(l);
        }
    }
```

```
    public void removeListener(GameObjectEventListener l) {
        if (l != null) {
            listeners.remove(l);
        }
    }

    public void notifyObjectTouch(GameObject object,
        GameObject otherObject)
    {
        Iterator i = listeners.iterator();
        while (i.hasNext()) {
            GameObjectEventListener l =
                (GameObjectEventListener)i.next();
            l.notifyObjectTouch(object, otherObject);
        }
    }
        ...
}
```

这个 GameObjectEventMulticster 类本身是个 GameObjectEventListener，但只是向其他监听器派遣事件通知。这里只显示了 notifyObjectCollision()方法，但所有其他通知方法也差不多。这里构造函数的参数为两个监听器，因为代码中至少要向两个监听器多点广播。

下面要在游戏实体中使用这些监听器。首先要在清单 14.5 中增加 GameObject 类的读取、增加和删除方法。

清单 14.5　GameObject.java 类的读取、增加和删除方法

```
private GameObjectEventListener listener

...
/**
    读取这个对象的 GameObjectEventListener
*/
public GameObjectEventListener getListener(){
    return listener;
}
/**
    增加这个对象的 GameObjectEventListener
*/
public void addListener(GameObjectEventListener  l){
        if (l ==null){
```

```
        return;
    }
    else if (listener ==null)
        listener = 1;
    }
    else if (listener instanceof GameObjectEventMulticaster){
            ((GameObjectEventMulticaster) (listener, 1);
    }
    else{
            listener = new GameObjectEventMulticaster (listener, 1);
    }
  }
/**
     删除这个对象的 GameObjectEventListener。
*/
public void removeListener(GameObjectEventListener    1)
        if (1 ==null);
        }
        else if (listener ==null){
          iistener =null;
          }
        else if (listener instanceof GameObjectEventMulticaster){
                ((GameObjectEventMulticaster) listener).removeListener(1);
        }
  }
```

这个代码跟踪一个监听器，如果注册多个监听器，则其可以是多目广播监听器。

有了这些监听器之后，就要改变碰撞探测与处理代码。原先通知实体的方法如下：

```
object.notifyWallCollision();
```

使用监听器时则要使用下列代码：

```
object.getListener().notifyWallCollision(object)
```

这样改变之后，就有了新的监听器机制。但还要让现有游戏实体方便地使用新的监听器机制，见清单 14.6。

清单 14.6　用 GameObject.java 来实现新的监听器机制

```
//让监听器适应的设计
listener    = new GameObjectEventListener(){
        public void notifyVisible(GameObject object,
                boolean visible);
    {
```

```
        object notifyVisible(Visible)
    }
    public void notifyObjectCollision(GameObject object,
        GameObject otherObject):
    {
        object notifyObjectCollision(otherObject);
    }
    public void notifyObjectTouch(GameObject object,
        GameObject otherObject);
    {
        object notifyObjectTouch (otherObject);
    }
    public void notifyObjectRelease(GameObject object,
        GameObject otherObject);
    {
        object notifyObjectRelease (otherObject);
    }
    public void notifyFloorCollision(GameObject object){
        object. notifyFloorCollision();
    }
    public void notifyCeilingCollision (GameObject object){
        object. notifyCeilingCollision();
    }
    public void notifyWallCollision (GameObject object){
        object. notifyWallCollision();
};
```

这个代码只是把通知转发到现有游戏实体方法。这样，GameObject 的子类（如 Player 与 PathBot）不用改变即可使用。

有了新的监听器机制和触发器之后，下面要介绍编制动作脚本的不同方法。

14.4 脚本

可以用不同方法编制动作脚本，取决于要达到的不同目的。下面是不同选项及其利弊：

- **编译 Java**。利用编译类的好处是速度快，很容易嵌入，但关卡设计员要下载 Java SDK，要知道如何生成类、设置类路径和进行编译。另外，仅仅为调用 doorl.open() 方法而生成整个类显得太浪费。
- **自己的命令解释器**。编写 doorl.open 类命令很简单，但这个方法不够灵活，因为它限制了能做的工作。此外，还要编写解释不同命令的代码。
- **使用脚本语言**。一般用户很容易使用脚本语言，可以提供所要的灵活性。有些脚本

语言还可以直接访问活 Java 对象。缺点是脚本比编译代码慢，需要单独的解释器。

本章使用脚本语言，除了上述好处之外，脚本还比编译类更适合快速开发，因为只要编辑文件和运动，跳过编译阶段。此外，还可以增加命令，在运行时动态更新脚本，特别适合游戏间关卡设计和调试。

缺点是脚本代码不如 Java 代码那么快，因为它要在运行时解释。但是，通常脚本只调用几个方法，可以把计算量很大的任务留在 Java 代码中，完整修订中可能同时需要编译和脚本代码。

Java 脚本语言解释器很多，例如，BeanShell 可以解释 Java 式代码，其他解释器有（Python）。还有针对其他语言的解释器，如 Tcl、Scheme、Smalltalk 与 Visual Basic。

选择脚本语言时，一定要保证脚本中可以直接访问活 Java 对象。连接活 Java 对象是游戏脚本的关键，利用这个功能，脚本可以调用活 Java 对象的方法，脚本与实际代码之间不必提供任何通信层。

脚本可以利用 Java Reflection API 可以直接访问活 Java 对象，反射调用对象方法时，只要知道方法名即可。

嵌入脚本语言解释器很简单，通常只要几个方法调用即可。本章嵌入的 BeanShell 是免费的 Java 解释器，很容易连接活 Java 对象。BeanShell 代码与 Java 代码相似，但更加灵活。BeanShell 是松散类型的，因此编码时不必进行类型转换和声明任何变量类型。此外，BeanShell 很小，内核解释器只要大约 125KB 内存就可运行。关于 BeanShell 的更多信息见 www.beanshell.org。

实现脚本需要两步：

- 设计脚本调用方法，关卡设计员在编写什么？
- 嵌入解释器，如何调用脚本功能？

换句话说，从需要角度看问题，然后实现它。

下面先介绍设计脚本。

14.4.1 设计脚本

首先要避免让关卡设计员生成类和接口，只要简单函数就可以了。

第二，访问游戏实体不能太麻烦。要避免编写下列代码：

```
openDoor(gameObjectManager.getObject("door1"));
```

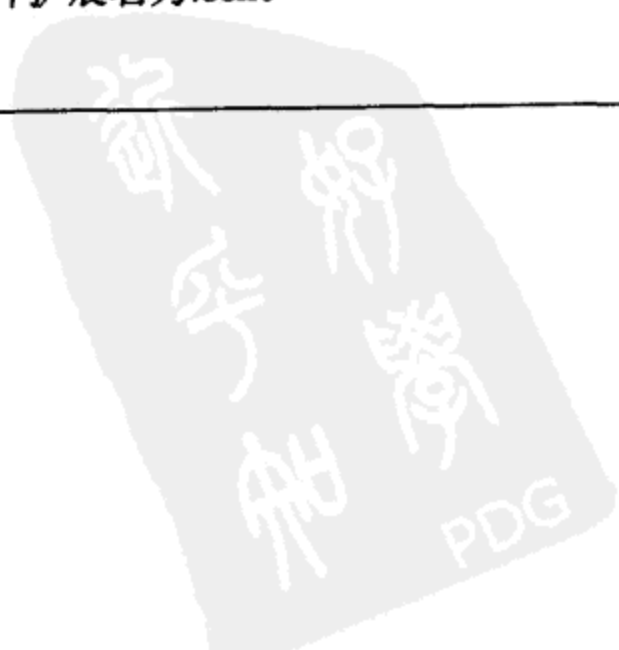
这个代码从游戏实体管理器取得游戏实体 door1，就像引用变量一样引用游戏实体，例如：

```
openDoor(door1)
```

清单 14.7 显示了所要的工作，注意 BeanShell 文件扩展名为.bsh。

清单 14.7 level1.Bsh

```
/**
    开门与关门的函数
*/
player_doorTriggerTouch(){
```



```
        //将门向上移
        moveDoor(180);
    }
    player_doorTriggerRelease(){
        //将门向下移
        moveDoor(0);
    }
    moveDoor（int y）{
        //移动门的 7 个部分
        speed = 0.5f
        moveYTO(doorla,y speed );
        moveYTO(doorlb,y speed );
        moveYTO(doorlc,y speed );
        moveYTO(doorld,y speed );
        moveYTO(doorle,y speed );
        moveYTO(doorlf,y speed );
        moveYTO(doorlg,y speed );
    }
```

这个脚本的贴图用 7 个对象表示门。这些对象样子像个长杆，因此门像扇大门。另外，还有个隐藏触发器区 doorTrigger。

这个脚本提供两个函数：play_doorTriggerTouch()开门。player_doorTriggerRelease 关门。变量 doorla,doorlb 等引用贴图文件中定义的游戏实体。

这很简单：只有两个函数，不需要这里没有的事件通知功能，如撞墙。

脚本的函数名采用这样的命名规则，每个函数名开头为接收通知的对象名。例如，player 对象的碰撞通知方法如下：

playerFloorCollision()

playerWallCollision()

playerCeilingCollision()

同样，如果 player 对象与 box 对象相碰，则调用下列方法：

player_boxCollision()

player_boxTouch()

player_boxRelease()

当然，这些函数不一定要在脚本中定义。

这样实现门有一个问题：人工智能 Bot 目前进行视景测试时只检查 BSP 树而不检查游戏实体，即 Bot 可以透过门看到门另一边的游戏者，在路径寻找时不考虑门。Bot 可能通过门走向游戏者或射击游戏者。要解决这个问题，可以在门上加一个假的隐藏多边形，作为机器人的视觉障碍。

注意：level1 中的 moveYTo()方法还没有定义。可以生成 BeanShell 文件 main.bsh，定义每一关的脚本中有用的公用函数。清单 14.8 显示了这个脚本，目前只定义

了 moveYTo()方法。

清单 14.8 mian.bsh

```
//重要导入
import com.bracken.javagamebook.game.*;
import com.bracken.javagamebook.math3D.*;
import com.bracken.javagamebook.util.*;
import com.bracken.javagamebook.path.*;

moveYTO (GameObject object,int y,float speed){
        object.setFlying(true);
        loc = new Vector3D(object.getLocation());
        loc.y = y
        object.getTransform().moveTo(loc,speed);
}
```

BeanShell 像 Java 一样导入包，因此为了简便起见，要导入几个包。可以在 main.bsh 中放上许多方法，使游戏的 BeanShell 脚本更容易编写。

设计脚本编写方法并提供样本脚本后，下面要在代码中嵌入 BeanShell 解释器。

14.4.2 嵌入 BeanShell 解释器

嵌入 BeanShell 解释器或任何其他解释器很简单。

对于 BeanShell，首先要生成 Interpreter 对象，其在.bsh 包中：

```
Interpreter bsh = new Interpreter();
```

可以用解释器的 eveal()方法求值 Java 表达式如下：

```
bsh.eval("System.out.println(5+3)");
```

还可以用 set()方法设置变量：

```
bsh.set("a",myObject);
```

最后，可以用 source()方法装入整个 BeanShell 脚本：

```
bsh.source("level1.bsh");
```

这样就可以了，但一定要到 www.beanshell.org 站点访问 BeanShell 的完整 API 文档。

清单 14.9 的 ScriptManager 类总结了所有 BeanShell 方法。

清单 14.9 ScriptManager.java

```
package com.brackeen.javagamebook.scripting;

import java.io.IOException;
import java.util.*;
import com.brackeen.javagamebook.game.*;
```

```
import bsh.*;

public class ScriptManager{

    private static final Class[] NO_ARGS = new Class[0];
    private Interpreter bsh;
    private GameObjectEventListener scriptedListener;

    public ScriptManager() {
        scriptedListener = new ScriptedListener();
    }

    /**
        设置关卡的 ScriptManager。执行一列脚本文件，GameObjectManager 中有名称的每个对
        象作为脚本的命名变量加进去
        另外还调用脚本方法 initLevel()（如有）
    */
    public void setupLevel(GameObjectManager gameObjectManager,
        GameTaskManager gameTaskManager, String[] scriptFiles)
    {
        bsh = new Interpreter();
        try {
            // 执行源文件（和装入方法）
            for (int i=0; i<scriptFiles.length; i++) {
                bsh.source(scriptFiles[i]);
            }

            bsh.set("gameTaskManager", gameTaskManager);

            // 把所有名称的对象作为脚本的命名变量
            //注意，这样就把所有对象留在内存中（活动），直到下一个关卡，即使在这一类的
            游戏中已经破坏
            Iterator i = gameObjectManager.iterator();
            while (i.hasNext()) {
                GameObject object = (GameObject)i.next();
                if (object.getName() != null) {
                    bsh.set(object.getName(), object);
                    // 向对象增加脚本监听器
                    if (hasScripts(object)) {
```

```
                        object.addListener(scriptedListener);
                    }
                }
            }
            // 初级关卡代码－调用 initLevel()
            invoke("initLevel");
        }
        catch (IOException ex) {
            ex.printStackTrace();
        }
        catch (EvalError ex) {
            ex.printStackTrace();
        }
    }

    /**
        检查指定的游戏实体是否有任何脚本（检查是否有以该对象名开头的脚本方法）
    */
    public boolean hasScripts(GameObject object) {
        if (object.getName() != null) {
            String[] names = bsh.getNameSpace().getMethodNames();
            for (int i=0; i<names.length; i++) {
                if (names[i].startsWith(object.getName())) {
                    return true;
                }
            }
        }

        // 什么也没找到
        return false;
    }

    /**
        指定方法名为现有脚本方法时返回真值
    */
    public boolean isMethod(String methodName) {
        return (bsh.getNameSpace().
            getMethod(methodName, NO_ARGS) != null);
    }
```

```
    /**
        调用指定脚本方法
    */
    public void invoke(String methodName) {
        if (isMethod(methodName)) {
            try {
                bsh.eval(methodName + "()");
            }
            catch (EvalError e) {
                e.printStackTrace();
            }
        }
    }
}
```

在这个代码中，setUpLevel()方法装入源 BeanShell 文件（我们要装入 main.bsh 与 level1.bsh）并设置 gameTaskManager（见下节介绍）。另外，它用 set()方法对每个游戏实体设置变量，使每个游戏实体很容易按名称引用（如 door1）。然后它检查特定实体是否有脚本，如果有，则增加 ScriptedListener（见后面介绍）。游戏实体有脚本时，就有以这个游戏实体名开头的函数。最后，它调用 initLevel()脚本函数，执行启动关卡所需的代码。

如果指定名称是脚本中的无变元方程，则 isMethod()方法返回真值。

注意：求值 BeanShell 表达式或装入 BeanShell 脚本时可能抛出 EvalError，表示 BeanShell 脚本有问题，其错误消息有助于调试。在这个代码中，我们只是把错误打印到控制台。

当然，ScriptedListener 类是个游戏实体事件监听器，根据函数命名规则调用 BeanShell 脚本。它只向监听器中增加有脚本的游戏实体。清单 14.10 显示了 ScriptedListener 内类的代码。

清单 14.10　ScriptManager.java 的 ScriptedListener 内类

```
/**
    GameObjectEventListener 将所有调用委托给脚本方法。每个有脚本方法的游戏实体都增加
    GameObject
*/
  public class ScriptedListener implements GameObjectEventListener{

      public void notifyVisible(GameObject object,
            boolean visible)
          {
                invoke(object.getName() +
                (visible? “Visible” :"NotVisible"));
```

```
    }
  public void notifyObjectCollision (GameObject object,
    GameObject otherObject)
    {
        if (otherObject.getName() !=null){
            invoke(object.getName() + "_" +
                    otherObject. getName() +"Collision");
        }
    }
  public void notifyObjectTouch (GameObject object,
    GameObject otherObject)
    {
        if (otherObject.getName() !=null){
            invoke(object.getName() + "_" +
                    otherObject. getName() +"Touch");
    }
}
 public void notifyObjectRelease (GameObject object,
    GameObject otherObject)
    {
        if (otherObject.getName() !=null){
            invoke(object.getName() + "_" +
                    otherObject. getName() +"Release");
    }
}
public void notifyFloorCollision(GameObject object){
   invoke(object.getName() +"FloorCollision");
}
public void notifyCeilingCollision(GameObject object){
   invoke(object.getName() +"CeilingCollision");
}
public void notifyWallCollision(GameObject object){
   invoke(object.getName() +"WallCollision");
}
    }
```

这个类是 ScriptManager 的内类，用 ScriptManager 的 invoke()方法调用脚本函数。这样就完成了脚本。下面是个分解例子：

- 游戏者接触 switch1。
- 碰撞探测代码取得游戏者的监听器和发出一个接触事件。

- 其中一个监听器是 ScriptManager，在脚本中调用相应接触函数。

最后还要注意一点，就是要注意掩盖代码的问题。掩盖将在第 18 章“游戏设计与最后 10%”介绍，把类、方法、字段名搞乱，使反编译代码很难读懂。这样就使调用某些方法的脚本遇到问题，因为掩盖代码中的方法名已经变了。因此，如果要掩盖代码，则要保留公开 API（保持公开方法名），只掩盖内部专用代码，保证脚本还能工作。

14.5 延迟事件

还有一个要实现的功能就是方便地生成延迟事件。图 14-3 显示了几种延迟事件。

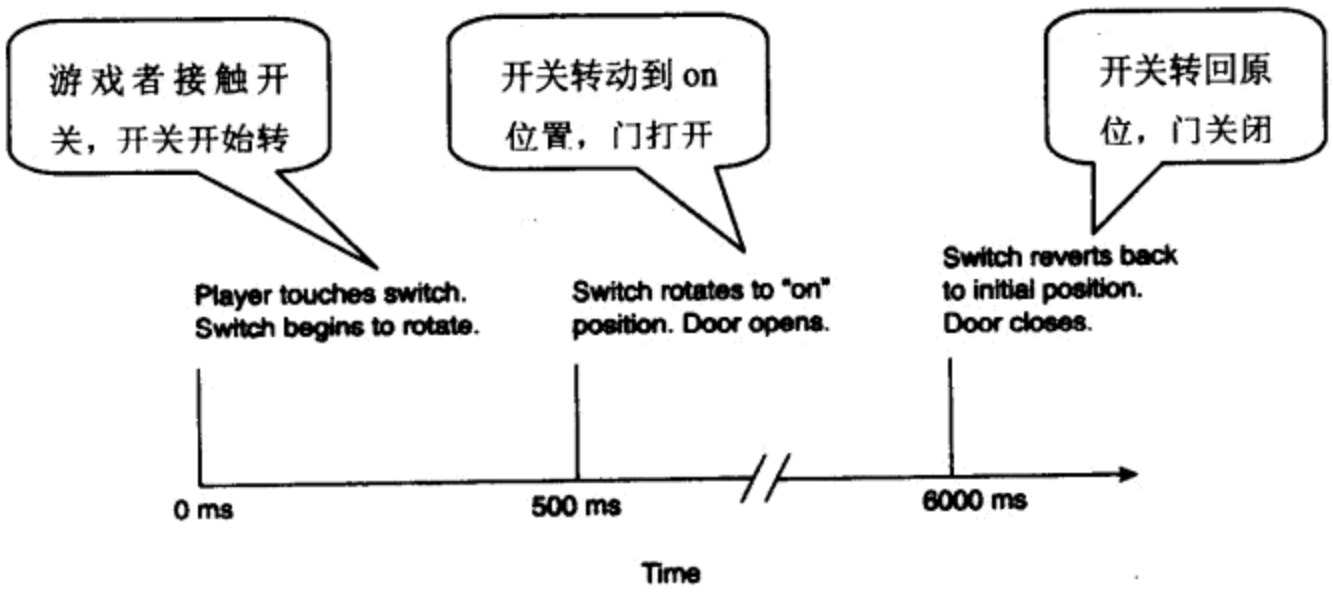


图 14-3 游戏者接触开关时，建立几个延迟事件

图中在游戏者接触开关时，建立几个延迟事件：开门和关门。

要生成处理延迟事件的系统，首先可以用 Timer 实用程序类。Timer 类很容易在指定延迟时间后执行任何。但是，Timer 类不适合典型游戏，原因有两个。第一，Timer 类用系统时间而不是游戏时间跟踪任务，即使游戏暂停，系统时间也还在前进，因此可能在游戏暂停时在错误的时间触发事件。第二，Timer 类在另一线程中执行代码，可能在游戏中造成同步问题。

因此，我们要用替换系统做两件事：在主线程中管理延迟事件，用游戏时间而不是系统时间跟踪任务。

清单 14.11 是个基本 GameTask 类，这个类取一个 Runnable 和延迟作参数，在指定延迟之后执行 Runnable。

清单 14.11 GameTask.java

```
package com.brackeen.javagamebook.scripting;
/**
    可以计划一次性执行的游戏任务
*/
public class GameTask implements Runnable {
```

```
private long remainingTime;
private Runnable runnable;
private boolean done;

/**
    生成新的游戏任务，在指定延迟之后执行 Runnable
*/
public GameTask(long delay, Runnable runnable) {
    this.remainingTime = delay;
    this.runnable = runnable;
}

/**
    取消这个任务
*
public void cancel() {
    done = true;
}

/**
    运行这个任务
*/
public void run() {
    if (runnable != null) {
        runnable.run();
    }
}

/**
    检查这个游戏任务是否准备执行，如果是，则执行。任务完成时返回真值（执行或前面取
    消时）
*/
public boolean check(long elapsedTime) {
    if (!done) {
        remainingTime-=elapsedTime;
        if (remainingTime <= 0) {
            done = true;
            try {
                run();
```

```
                }
                catch (Exception ex) {
                    ex.printStackTrace();
                }
            }
        }

        return done;
    }
}
```

这个类设计成在每帧调用 check()方法，跟踪任务执行前余下的时间，这个值在 GameTaskManager 方法中建档，到达 0 时即执行任务。

下面生成 GameTaskManager，见清单 14.12，其跟踪所有等待的游戏事件。

清单 14.12　GameTaskManager.java

```
package com.brackeen.javagamebook.scripting;
import java.util.*;

/**
    管理 GameTask 对象队列
*/
public class GameTaskManager {

    private List tasks;

    /**
        生成空任务队列的新 GameTaskManager
    */
    public GameTaskManager() {
        tasks = new ArrayList();
    }

    /**
        将任务加进队列中，在指定延迟之后执行 Runnable
    */
    public void addTask(long delay, Runnable runnable) {
        addTask(new GameTask(delay, runnable));
    }
```

```
/**
    将任务加进队列中
*/
public void addTask(GameTask task) {
    tasks.add(task);
}

/**
    清除任务队列
*/
public void clear() {
    tasks.clear();
}

/**
    更新管理器，执行准备的任务
*/
public void update(long elapsedTime) {

    List removeList = null;
    int size = tasks.size();

    //执行任务可能在队列中增加更多任务
    for (int i=0; i<size; i++) {
        GameTask task = (GameTask)tasks.get(i);
        if (task.check(elapsedTime)) {
            // 将对象加进对象表中，后面要清除
            if (removeList == null) {
                removeList = new ArrayList();
            }
            removeList.add(task);
        }
    }

    // 清除执行的任务
    if (removeList != null) {
        tasks.removeAll(removeList);
    }
```

```
        }
    }
```

GameTaskManager 的 update()方法和 GameObject 与本书的许多其他类中一样，调用每个 GameTask 的 check()方法。这里只是保证在每帧调用 check()方法和实现延迟任务。

在这个代码中，我们考虑到，执行任务可能在队列中增加更多任务。代码在删除已执行的任务时很慎重，只在退出循环后才从表中清除任务。

BeanShell 中生成延迟事件

BeanShell 可以生成任意接口，因此很容易在脚本中生成延迟事件。方法如下：

```
gameTaskManager.addTask(delay,new Runnable(){
        run(){
            DoSomething();
        }
});
```

当然，我们要求脚本没有类和接口，我们用 main.bsh 中的 delay()函数简化这个工作，见清单 14.3。

清单 14.13　main.bsh 中的延迟任务

```
/**
    方便方法，用游戏任务管理器在指定延迟之后执行语句
*/
    delay（long delay, String statement）{
        gameTaskManager.addTask(delay,new Runnable(){
      run(){
    this.interpreter.eval(statement);
  }
 })
  }
```

这个函数在指定延迟之后执行语句。例如，如果要在六秒钟后调用 closeDoor()，则可以用下列语句：

```
delay(6000," closeDoor()");
```

注意函数放在引号中，这是因为语句字符串，由 BeanShell 解释。

有了延迟事件系统之后，下面要试试看。

14.6　综合起来

清单 14.14 是演示延迟事件的 BeanShell 脚本。在这个脚本中，游戏者接触 stairTrigger 时，开关移动；楼梯在 750 毫秒之后向上移。五秒钟后，调用 closeStairs()方法，让楼梯降到地上。

清单 14.14　在 level1.bsh 中移动楼梯

```
/**
    打开与关掉楼梯的函数
*/
global.staircsOn = false;
player_stairTriggerTouch() {
    if (global.staircsOn) {
        return;
    }
    openStairs();
}

openStairs() {
    // 将楼梯（转台）转到开位置
    group = stairTrigger.getPolygonGroup().getGroup("turret");
    group.getTransform().turnYTo(3.14159f, 0.005f);
    // 楼梯在 750 毫秒之后向上移
    delay(750, "toggleStairs(true)");
    delay(5000, "closeStairs()");
    global.staircsOn = true;
}

closeStairs() {
    //将楼梯（转台）转到关位置
    group = stairTrigger.getPolygonGroup().getGroup("turret");
    group.getTransform().turnYTo(0, 0.005f);
    // 让楼梯降到地上
    toggleStairs(false);
    delay(750, "global.staircsOn = false;");
}

toggleStairs(boolean raise) {
    moveYTo(stair1, raise?192:0, .1f);
    moveYTo(stair2, raise?192:0, .1f);
    moveYTo(stair3, raise?192:0, .1f);
    moveYTo(stair4, raise?192:0, .1f);
    moveYTo(stair5, raise?192:0, .1f);
    moveYTo(stair6, raise?128:0, .1f);
```

```
        moveYTo(stair7, raise?64:0, .1f);
    }
```

作为演示，本书在本章代码中放了一个 EventTest 类，使用本书的三维引擎并将其与脚本和延迟事件结合起来。这个类中惟一更新的地方是设置脚本管理器和对每个帧更新游戏任务管理器。

要运行这个测试程序，就要在类路径中包括 BeanShell 库。可以用源代码所带 Ant 建立文件中的 run 任务，也可以从命令行的 build 文件夹中运行测试程序如下：

```
java -cp..\lib\bsh-core-1.2b7.jar;.EventTest
```

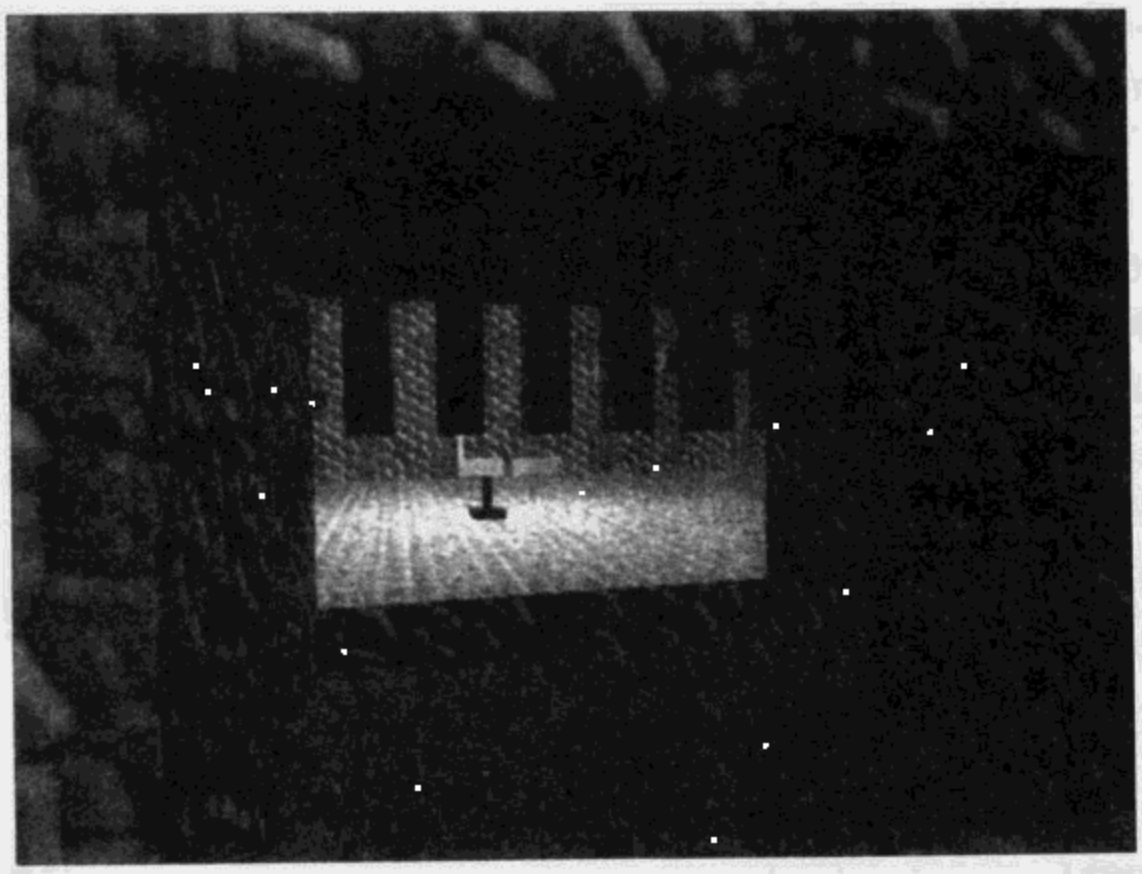


图 14-4 这个屏幕图形是本章的演示，显示开门（条）和背景中的开关，开关激活楼梯

记住，建立这种环境交互的乐趣之一是让游戏者发现发生的事情。一定要使某些环境交互独特和微妙些，有时让游戏者探索出某些奥秘。

14.7 改进

利用这类游戏脚本可以干许多事。下面只是一些可以增加的环境交互。

- （按一定顺序）拉三个杠杆开门。
- 拉一个控制杆在房子中局部灌水，以便在浮到较高位置的浮动平台上走路。
- 切断桥的吊索，让桥掉下，以便寻找下面的密室。
- 向墙上喷火，让它倒塌，以便到另一边。
- 在按钮上压一个档块，使门一直打开。
- 按下按钮，打开全息图，提供地图信息。

另外，本章的代码还可以用不同方法扩展和改进。下面是一些功能扩展方法：

- 建立许多常用脚本对象，如门和开关，以免重复编写。
- 根据实体状态显示不同多边形组。例如，按下红色按钮时，使其变绿。

- 在用户按动作键时执行动作，如单击右鼠标键或按 Enter 键。
- 游戏者可以站在上下移动的平台上，但也可以站在左右移动的平台。
- 可以生成飞行仪表盘。
- 对所有同类型对象编脚本而不是对每个编脚本（如在所有 Projectile 对象中增加脚本方法）。
- 增加其他类型的通知（破坏、移动、停止移动，等等）。
- 在游戏者有特定东西或游戏者要把某个东西给另一角色时执行特定操作。
- 生成用户界面通知，如游戏者接触没见过的实体时激活游戏帮助屏幕。
- 增加热键，动态重装脚本，使关卡设计者很容易在游戏中调整脚本。
- 将脚本压缩和存放在文件中，使游戏者无法或很难编辑脚本（没有秘笈）。

14.8　小结

本章编写游戏脚本，游戏脚本可以使游戏更有趣更刺激。我们演示了开门和短时间出现并消失的楼梯，我们介绍了延迟事件、脚本、接触与斜放事件和新的监听器机制。

本章只是介绍了脚本的技术方面，而更主要的是创造性，要靠读者自己去领会，请动手生成一些具有大量环境交互的游戏吧。

第 15 章　保存游戏

关键问题

- 游戏保存基础
- 对游戏状态保存使用 Java 序列化 API
- 生成游戏屏幕图形
- 将游戏保存到正确目的地
- 小结

Half-Life2,Doom III 最新 Tomb Raider 续集，The Sims 等游戏提供的游戏世界是如此逼真和迷人，许多游戏者希望它每次都能从上次离开的地方继续往下体验。开发人员如果不在自己的杀手锏游戏中提供游戏保存特性，简直就是自寻绝路。

游戏者着迷地希望能从上次离开的地方继续玩下去，体验他们喜欢的冒险世界。游戏者通过解迷题、发现区域、打败敌人、收集物体和积分等手段好不容易达到了目前的状态，要让他在退出游戏时一下子失去这一切，显然是太残酷了。

15.1　游戏保存基础

从游戏编程角度看，保存就是把游戏状态保存到永久存储设备（硬盘或网络），以后可以在游戏过程中随时重装这些游戏状态。

多关游戏中的一个简单方法是只保存游戏者当前所在的关卡。这样很容易实现：只要在文件中写入一个数字（level=5）或代码字（level=forestWorld9）。代码字可以保密，使游戏者不能用秘笈猜出上一关的名称，自己修改保存的游戏文件。

这个方法的缺点是，游戏者装入游戏时，要从这一关的开头开始，即使上次玩到了这一关末尾。另外，这里没有保存游戏者的能力，因此游戏者不能在游戏过程中进化（例如取得新东西或新能力）。

大多数现代游戏提供更高级的游戏保存功能：在游戏任何时刻保存“冻结”游戏状态，装入时回到这个点。这里保存的是整个游戏状态，游戏者回到同一位置，敌人继续从原来的位置跑，游戏者下方爆炸的炸弹还在爆炸。

从头开始的游戏保存方法保存一组核心游戏变量，用普通二进制文件构成游戏的当前游戏者相关状态。文件格式通常是针对游戏调整的定制专属格式。

这种游戏保存状态包括任何无法从全新安装游戏拷贝重新计算的状态：

- 游戏者的进程（在游戏世界中的位置、积分、受伤和其他实体，如武器、魔法、弹药和特殊项目）。
- 所有可移动游戏实体的当前状态（位置、方向、动画循环、速度、人工智能状态，等等），包括游戏者及其敌人和伙伴。
- 子集事件和游戏脚本中定义的变量（关门事件，五秒钟后激活楼梯的事件，等等）。

- 游戏者配置的所有全局游戏配置数据（键盘贴图，游戏难度测试，绘制质量，声音/音乐设置，等等）。

手工保存所有这些数据没有任何难度，但这个方法很费时间，无法利用 Java Serialization API 的复用能力。这个标准 API 可以进行大量基础工作，因此本章主要介绍如何安全地利用 Java Serialization API 特性实现游戏的屏幕保存与重装功能。

但是，Java Serialization API 只解决了游戏屏幕图形问题的一部分。让用户保存游戏时，我们通常让用户对游戏屏幕图形取一个名称，以便后面要退回保存的游戏时浏览和选择保存的游戏屏幕图形名单。随着游戏特性与日俱增，现代游戏通常会显示每个保存游戏的缩略图。这样，游戏者只要直观地浏览缩略图，只在缩略图存在歧义时才用到名称。到本章结束时，我们将介绍如何实现这种特性，用到 AWT 的自动化主管 Mr.Robot。

15.2 对游戏状态保存使用 Java 序列化 API

游戏开发人员如果要使用 Java Serialization API，就只能认真学习这个框架的工作方法。有些 API 可以只学习整个 API 的一小部分，而 Java Serialization API 则没有这种捷径，下面几节详细介绍序列化技术及其问题。

15.2.1 序列化简介

Java Serialization API 的核心功能是把对象变成字节流，把字节流变成对象，分别称为序列化（serialization）和反序列化（deserialization）。

Java Serialization API 在 Java 1.0 向 Java 1.1 升级时就已经加进 Java 平台中（大约在 1996 年末）。这个 API 在当时的主要功能是支持远程方法调用（RMI，即分布式对象）和 Java 可复用组件体系结构 JavaBeans。我们要用这个 API 完成更宏伟的目标：游戏状态保存。

要保存任何对象（防止它在所在 Java 虚拟机中消失），只要将对象变成输出字节流，将这个输出字节流传输到某种独立于当前 Java 虚拟机的外部存储实体，如文件或网络套接（本章其余部分假设读者熟悉 Java 的基本 I/O 流 API），从而保存对象，实现持久性。

15.2.2 序列化基础

Java Serialization API 在 java.io 包中，这个 API 的核心是 Serializable 接口和两个特殊流类，可以序列化和反序列化实现 Serializable 接口的对象：ObjectOutputStream 与 ObjectInputStream。

如果你熟悉 java.io 包中的 DataOutputStream 与 DataInputStream 类，则可以把 ObjectOutputStream 与 ObjectInputStream 类看成这些类的改进版，可以读/写对象，而不只是读/写基本数据和字符串。

但是，介绍这些流类之前，先要介绍参与序列化的对象所要的 Serializable 接口功能：

```
public interface Serializable{
    //empty!
}
```

Serializable 接口没有指定任何行为。与大多数接口不同的是，Serializable 接口不要求

实现任何方法，也没有定义任何常量。这个接口只是把类标为可序列化。

什么意思呢？能否装入所有 Java 源文件，在所有类中加上 implements Serializable？显然不能。本章稍后会介绍为什么不能。

说明：Serializable 接口有个密切相关的子接口 Externalizable，详细信息见 Sun 公司 Java Web 站点 java.sun.com 上的 Java 对象序列化规范。

简单序列化例子。

下面先介绍一个简单序列化例子。清单 15.1 显示的例子程序序列化日常 StringBuffer 对象并将其写入文件中。

清单 15.1 SaveStringBuffer.java

```
import java.io.*;

class SaveStringBuffer {

public static void main (String[] args) {

    StringBuffer sb = new StringBuffer("Beam me up!");

    try {
        FileOutputStream fos = new FileOutputStream("sb.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject(sb);
        oos.close();

    } catch (IOException ioException) {
        System.out.println(ioException);
    }
}
}
```

这个例子的关键是 ObjectOutputStream 对象的 writeObjcet()调用。流对象会序列化变元对象，通过内核反射分析构成 StringBuffer 的实例字段表，将这些字段的值变成字节流。如果看看 StringBuffer 的源代码，则可以看到每个 StringBuffer 包括 char[]、int 与 boolean。这些字段从对象读取并流化成 OutputStream。本例中，我们通过图标 FileOutputStream 将得到的字节流发送到硬盘中。

如果运行这程序，则可以看到当前目录中有个新文件 sb.ser，包含序列化的 StringBuffer。

为了证明这个说明法，可以运行下列互补程序，安装、反序列化和打开 StringBuffer（见清单 15.2）。

清单 15.2 LoadStringBuffer.java

```
import java.io.*;

class LoadStringBuffer {

public static void main (String[] args) {

    StringBuffer sb = null;

    try {
        FileInputStream fis = new FileInputStream("sb.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        sb = (StringBuffer) ois.readObject();

        ois.close();

    } catch (IOException ioException) {
        System.out.println(ioException);
    } catch (ClassNotFoundException classNotFoundException ) {
        System.out.println(classNotFoundException );
    }

    System.out.println("Welcome back aboard, Sir! : " + sb);
  }
}
```

运行这个程序后，可以在控制台上得到下列输出，证明 sb.ser 中包含序列化的 StringBuffer 对象：

Welcome back aboard, Sir! : Beam me up!

尽管这比保存相应内容复杂得多，但这个 LoadStringBuffer 例子的核心是 ObjectInputStream object.readObject 对象的 readObject() 调用。ReadObject() 方法与 writeObject()方法正好相反，将字节流反序列化成原对象的拷贝。由于 readObject()声明返回 Object，因此要将返回值转换成更具体的子类，这里是 StringBuffer。

15.2.3 序列化规则

要让一个类可序列化，就要能用字节序列取出这个对象的状态。对象状态是对象非临时实例字段的数值集合（序列化引擎忽略临时字段）。

任何基本类型字段（boolean、char、byte、short、int、long、float 与 double）都可以顺

利转换成字节，但运行时非 null 的引用字段要序列化时，要求引用所指的对象也实现 Serializable 接口。这个规则递归采用，因此要序列化的根对象直接或间接引用的对象都应可序列化。

实际上，这表明可以将任何只声明基本类型字段的类声明（或标记）为可序列化，而不必操心这种声明的真实性。得到的类总是可序列化的。

另一方面，任何类声明引用其他对象时要经过认真分析确定这个类能否声明可序列化。

类声明任何引用字段时，要一一检查引用的对象类型是否可序列化。如果引用的任何类型不可序列化，则类通常不能声明为可序列化（稍后将会介绍如何定制序列化过程，绕开这个问题）。要声明一个类可序列化，所有引用都应可序列化。

本章稍后将回头分析可序列化的这个基本性质。

序列化的核心类支持

大多数类都声明引用，这些引用通常引用核心库类型（String、Point、Color 等等），因此要注意核心库类的可序列化情况。特别地，要知道 java.lang 与 java.util 包哪些类是可序列化的。表 15.1 和表 15.2 包含 Java 2 平台 1.4 版的这个信息。

从表 15.1 和表 15.2 可以看出，并不是所有 java.lang 与 java.util 包中的类都可序列化，稍后将会介绍具体原因。

表 15.1　JDK 1.4 java.lang 类及其可序列化状态

类名	可序列化	类名	可序列化
java.lang.Booleam	√	java.lang.Process	
java.lang.Byte	√	java.lang.Runtime	
java.lang.Character	√	java.lang.	
java.lang.Class	√	RuntimePerission	√
java.lang.ClassLoader		java.lang.	
		SecurityManager	
java.lang.Compiler		java.lang.Short	√
java.lang.Double	√	java.lang.	
java.lang.Float	√	StackTraceElement	√
java.lang		java.lang.StrictMath	
InheritableThreadLocal		java.lang.String	√
java.lang.Intege	√	java.lang.StringBuffer	√
java.lang.Long	√	java.lang.System	
java.lang.Math		java.lang.Thread	
java.lang.Number	√	java.lang.ThreadGroup	
java.lang.Object		java.lang.ThreadLocal	
java.lang.Package		java.lang.Throwable	√

表 15.2　JDK 1.4 java.util 类及其可序列化状态

类名	可序列化	类名	可序列化
java.util.		ava.util.Collections	
AbstractCollection		java.util.Currency	√
java.util.AbstractList		java.util.Date	√
java.util.AbstractMap		java.util.Dictionary	
java.util		java.util.	
AbstractSequentialList		EventListenerProxy	
java.util.AbstractSet		java.util.EventObject	√
java.util.ArrayList	√	java.util.	
java.util.Arrays		GregorianCalendar	√
java.util.BitSet	√	java.util.HashMap	√
java.util.Calender	√	java.util.HashSet	√
java.util.Hashtable	√	java.util.	
java.util.		PropertyResourceBundle	
IdentityHashMap	√	java.util.Random	√
java.util.LinkedHashMap	√	java.util.ResourceBundle	
java.util.LinkedHashSet		java.util.SimpleTimeZone	√
java.util.LinkedList	√	java.util.Stack	√
java.util.		java.util.StringTokenizer	
ListResourceBundle		java.util.TimerTask	
java.util.Locale √		java.util.TimerZone	√
java.util.Observabl		java.util.TreeMap	√
java.util.Properties	√	java.util.TreeSet	√
java.util		java.util.Vector	√
PropertyPermission	√	java.util.WeakHashMap	

值的一提的是，Object 本身没有实现 Serializable 接口。由此可见，Object 的任何直接子类均不可序列化，除非显式标记（和设计）成可序列化。

如果将前面的 SaveStringBuffer 例子修改成保存普通对象：

```
oos.writeObject(new Object())
```

则运行修改版本时会产生下列运行异常：

```
java.io.NotSerializableException: java.lang.Object
```

NotSerializableException 和 NullPointerException 常常会让你在调试 Java 应用程序的序列化问题时绞尽脑汁。

看看 java.lang 包中的其他类，可以看到大部分基本类都是可序列化的：所有包装类都是可序列化，如 String，StringBuffer，甚至包括 Throwable。由于 Throwable 类可序列化，

因此所有异常都可序列化，不管其在哪个包中声明，是否检查。

在 java.util 包中，除了 Collection 接口的抽象框架实现外，大部分类都是可序列化的。注意，Collection 接口理论上可以扩展为 Serializable（记住，Java 中只能通过接口实现多重继承），因此可以强制所有集合类为可序列化。但是，Sun 公司虽然不支持这种太严格的方法。

一个类不声明为可序列化，通常是有原因的，通常是这个类序列化没有意义。只有静态方法的类（如 Math 与 Collections）不产生任何对象，而对基础 Java 虚拟机具有复杂依赖性的类（如 Thread 与 System）也不支持序列化，其中涉及复杂的技术和类特定原因。

说明：伴随 Web 站点中的控制台实用程序 IsSerializable 可以交互或确定任何完全限定类的可序列化性。如果不向这个实用程序传递任何命令行变元，则其列出 1.4 java.lang 包中所有类的可序列化性。

15.2.4 序列化问题

Java 内置支持对象序列化，能够透明地处理对象序列化，非常方便。但是，如果不了解序列化的微妙之处，则对象序列化还有一些问题，破坏了这种透明性。

1. 未知对象不能反序列化

如果你在运行 SaveStringBuffer 例子与 LoadStringBuffer 例子之间去喝一杯咖啡，有人开了个玩笑，把你的 sb.ser 文件换成另一个 sb.ser 文件，包含序列化的 ButYouDontHaveSuchAClassAnywhere-OnYourClassPath（子虚乌有）对象，则可以看到为什么 ObjectInputStream.readObject()方法会强制你处理可能的 ClassNotFoundException 异常。虽然，readObject()方法背后的反序列化引擎要能访问要重构的任何对象的类。如果 Java 虚拟机中还没有这些类，也不能通过当前类装入器装入，则反序列化失败。

处理 RMI 应用程序时，经常遇到 ClassNotFoundException 异常，这在依赖于传统（即缺省）类装入器行为的独立游戏中不成问题，因为序列化对象通常由序列化对象时所在的同一个游戏程序反序列化。

2. 非可序列化对象图

如果将 SaveStringBuffer 例子修改成保存一个 HashSet 对象，包含下列元素的混合包：

```
Set bag = new HashSet();
bag.add("Hello");
bag.add(new Object());
bag.add(new Integer(2003));

oos.writeObject(bag);
```

则再次运行新版本会产生处理序列化时常见的运行异常：

```
java.io NotSerializableException: java.lang.Object
```

尽管 HashSet 实现 Serializable 接口，表面值应该可序列化，但 HashSet 的运行内容则不一定。这时，包中的 Object 不可序列化。

让 ObjectOutputStream 序列化一个对象而这个对象引用其他对象，其他对象又引用更多对象，对所有这些对象（即整个对象图）都序列化。如果图中任何对象本身不可序列化，则整个操作失败，抛出 NotSerializableException。图 15-1 显示了这个常见的序列化问题。

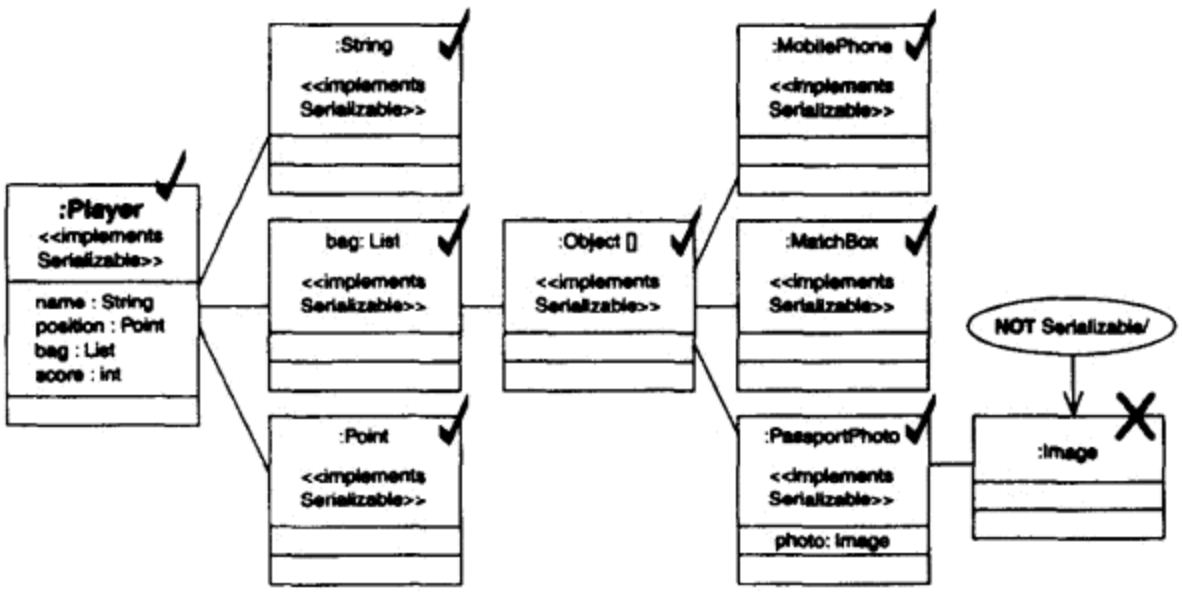


图 15-1 整个对象图都应为可序列化对象。例如，整个对象图中任何一个部分包括图像时，整个对象图的序列化失败

包含普通 Object 的 HashSet 就是序列化的容器对象被污染的典型例子，序列化时，一个或几个元素不可序列化就全盘失败。

3. 数组序列化

下面看看 ObjectOutputStream.writeObject()与 ObjectOutputStream. readObject()方法的精确签名：

```
public final void (Object obj)
        throws   IOExceptionpublic final Object readObject()
        throws   ClassNotFoundException, IOException
```

可以看到，参数与返回值类型初看起来很松散，为什么下列方法用 Serializable 而不用 Object 呢？

```
public final void writeObject(Serializable aSerializable)
                throws IOException
public final Serializable   readObject()
                throws ClassNotFoundException, IOException
```

这些方法只接收和产生 Serializable 对象。现有方法签名可以用规则的准例外一数组进行解释。

数组是 Java 中成熟的对象，任何 Serializable 测试都返回真值，数组传入序列化引擎时不会有任何问题，只要每个数组元素也可序列化。这个问题实际上与容器对象序列化时的问题相同。

4. 类进化与版本

另一个序列化问题是用序列化“长期”保存对象时，这里的“长期”指派生序列化对

象的类在此期间可能改变。

假设编写一个围棋游戏，可以保存任何正在进行的游戏，则下列类是实现游戏的核心：

```
public final ChessGameState implements Serializable{

private ChessBoard theBoard;
private ChessPlayer white;
private ChessPlayer black;
}

class ChessPlayer implements Serializable{
private String name;
}
class ChessPlayer implements Serializable{
private ChessPiece[][] squares;
}
```

这个围棋游戏中保存游戏就是序列化单体对象 ChessGameState 和把序列化对象存放到硬盘中。

假设围棋游戏 V1.0 版取得了巨大成功（也许你采用经典的 Internet 市场战术，免费提供产品，争取更大市场份额），使你可以吸引风险投资，开发改进的 V2.0 版本，准备开始收钱。

V2.0 的任务之一是修复 V1.0 的缺陷：V1.0 中没有实现一个游戏者只剩下国王时的 50 移动规则。这个问题很容易解决，只要在 ChessGameState 中增加一个 int 字段和相关逻辑。

```
public class ChessGameState implements Serializable{
private ChessPlayer theBoard;
private ChessPlayer white
private ChessPlayer black
private int fiftyMovesCounter
}
```

但这时 MyChess V2.0 因为测试遇到了问题。恢复保存的 V1.0 游戏时系统崩溃，出现下列运行异常。

```
java.io.InvalidClassException: ChessGameState; local class
incompatible: stream classdesc serialVersionUID =
8698504323191665099
local class serialVersionUID =3813626932283615031
```

反序列化引擎发现 Java 虚拟机中的 V2.0 ChessGameState 类不同于 V1.0 ChessGameState 类，而对象过去是用 V1.0 类序列化的。换句话说，反序列化引擎发现增加了 fiftyMovesCounter 字段，并表示反对。

为了避免这种类进化问题，要保证进化的类保持序列化版本兼容。

什么意思呢？序列化版本兼容类就序列化而言是一致的，这个相等性可以用 64 位签名

数 serialVersionUID 评估。输出（序列化）时，这些类签名可以由序列化引擎动态计算，也可以由编程人员在可序列化类中显式声明：

private static final long serialVersionUID =3786198910865385080L

输入（反序列化）时，嵌入序列化对象流中的 serialVersionUID 与对象类的 serialVersionUID 比较，如果两者不同，则反序列化失败。

要计算类的 serialVersionUID，实际上要取得完全限定类名和类的所有已声明非临时实例字段，建立类似于对象运行散列码的值。因此，增加、删除和改变实例字段改变计算 serialVersionUID 的值，会使反序列化引擎抛出 InvalidClassException 屏幕。

要解决这个问题，可以显式用静态 serialVersionUID 字段标记一个类，其在类改变时保持常量。这样，任何改变计算 serialVersionUID 值的变化都不会在反序列化时抛出 InvalidClassException 屏幕。只要类的改变是序列化版本兼容的，就可以长期保存进化类的对象。

序列化不兼容的类改变包括：

- 删除或改变实例字段类型。
- 改变实例字段名称（使用掩盖时要注意）。
- 改变继承树中类的位置。
- 对任何专用 readObject()方法进行不兼容逻辑改变。

序列化兼容的类改变包括：

- 增加实例字段。
- 改变任何字段的访问范围。

这里只列出了最常见的改变，完整清单见 Java 对象序列化规范。

15.2.5 覆盖缺省行为

ObjectOutputStream 将对象图序列化成按序列化流协议格式化的字节流。缺省协议不仅是对象字段值的二进制表示加处理对象类型与引用的明智方式。

从游戏编程人员角度看，这种格式可能有问题，因为协议太简单会造成数据透明度。游戏者只要有一定的 Java 知识，很容易破译序列化对象。例如，典型的游戏进度记录游戏者的当前成绩，还有多长寿命和目前累积了哪些物质和能力。这此属性是很好的破译目标。为了防止破译和泄露秘笈，持久对象应尽量不透明。

为此，序列化 API 可以覆盖缺省行为。为此不是扩展 ObjectOutputStream 与 ObjectInputStream 类，覆盖其公开 writeObject()与 readObject()方法（前面曾介绍过，这些方法不能覆盖，是最终方法），而不采用非一般性技术，见表面介绍。

1. 增加专用 writeObject()与 readObject()方法

假设有 writeObject()方法和下列游戏状态类。

```
class GameState implements Serializable{
private Player player;
}
class Player implements Serializable{
```

```
private short livesLeft;
}
```

假设使用缺省序列化的游戏单体 GameState 对象使用户很容易破译余下寿命，则要用另一个数据防护，防止文件编辑器（通常是十六进制编辑器）轻易改变序列化 LivesLeft 字段。

序列化引擎直接支持的方法之一是在 Serializable 类中实现专用方法，手工读取和写入类字段，而不靠序列化引擎。这样就可能用掩盖的或更强加密的格式写入字段值。

要用下列签名声明这些补充专用方法：

```
private void writeObject(ObjectOutputStream s)throws IOException
private void readObject(ObjectOutputStream s)throws IOException
```

清单 15.3 显示了如何用 writeObject()方法改进 Player 类，利用这个技术。

清单 15.3　GameState.java

```
import java.io.*;

public class GameState implements Serializable {

private Player player = new Player(3);

public static void main (String[] args) {

    try {
        FileOutputStream fos = new FileOutputStream("game.ser");
        ObjectOutputStream oos = new ObjectOutputStream(fos);

        oos.writeObject( new GameState() );
        oos.close();

        FileInputStream fis = new FileInputStream("game.ser");
        ObjectInputStream ois = new ObjectInputStream(fis);

        System.out.println( ois.readObject() );

        ois.close();

    } catch (Exception exception) {
        System.out.println(exception);
    }
}
```

```
public String toString() {

    return "GameState[ " + player.toString() + " ]";
}
}

class Player implements Serializable {

private short livesLeft;

/*package*/ Player(int lives) {
    livesLeft = (short) lives;
}

private void writeObject(ObjectOutputStream s) throws IOException {
    System.out.println("Encrypting lives left...");
    s.writeByte(- livesLeft);
}

private void readObject(ObjectInputStream s) throws IOException {
    System.out.println("Decrypting lives left...");
    livesLeft = (short) - s.readByte();
}

public String toString() {

    return "Player[" + livesLeft + "]";
}
}
```

运行 GameState 的 main()方法时，可以从控制台中看到其他 writeObject()与 readObject()方法不是自己的代码调用，而是序列化引擎调用的。序列化 GameState 对象（包含一个序列化 Player 对象）不是把游戏者的 livesLeft 变量存放在普通文本中，而是存放成加密形式。为了使这个例子尽量简短，这里的“加密”不过是对数值进行一些位操作（使用一元负号运算库）。在实际游戏中，可能要使用更高级的加密形式，使游戏黑客更难啃。

2. 显式指定序列化的字段

增加专用 writeObject()与 readObject()方法比缺省序列化引擎前进了一大步，因为这个方法完全负责特定类对象的序列化和反序列化。

对包含大量字段的类，大多数字段不用加密或掩盖，因此这个全是全非的方法会掩盖

使用序列化 API 的一些好处。我们要对大多数字段使用缺省机制，只以定制方式处理极少数可能被破译的目标。

Java 的序列化框架可以处理这个问题，显式指定哪些字段用缺省机制处理。回顾 serialVersionUID 字段，这是在 Serializable 类中定义另一个专用最终静态字段 serialPersistentFields 如下：

```
class Player implements Serializable{
private String playerName;
private short livesLeft;

private static final ObjectStreamField[] serialPersistentFields = {
new ObjectStreamField[]( "playerName" ,String.class)
};
    //..在此放游戏者方法
}
```

可以看出，这个 serialPersistentFields 字段要作为 ObjectStreamField 对象的一维数组输入。这个数组定义当前类中声明和存在的哪个实例字段用缺省机制。每个 ObjectStreamField 数组元素对象通过声明名称和类型定义字段，字段范围和任何其他字段属性不是必要的，因此写入对象并忽略（writeObject 与 objectgnored）。

在 Serializable 类中包括 serialPersistentFields 字段时，还要改变实现 writeObject()与 readObject()方法的方式，分别调用 ObjectInputStream.defaultReadObject()方法：

```
private void writeObject(ObjectOutputStream s)throws IOException
{
    s.defaulWritiObject();
    System.out.println( "Encrypting lives left..." );
    s.writeByte(-livesLeft);
}
private void readObject(ObjectInputStream s) throws IOException{
    s.defaulWritiObject();
    System.out.println( "Decrypting lives left..." );
    livesLeft = (short) - s.readByte();
}
```

混合下面两种方法：

- 在 Serializable 类中增加专用 writeObject()或 readObject()方法。
- 显式指定哪个字段用缺省机制序列化。

就可以一个字段一个字段调整对象序列化的方式，而不是粗糙地处理整个类。利用这种灵活性可以加密游戏保存状态中的任何敏感字段。

15.3 生成游戏屏幕图形

如果显示屏幕可以用某种系统 Java 对象（System,Runtime）访问，而这种假想类实现 Serializable 接口，则取得屏幕图形的方法可以和任何其他 Java 对象取得屏幕图形的方法一样：通过序列化。但是，Java 没有将屏幕贴图方便对象，因此序列化无法满足屏幕图形要求。

AWT 包在 1.3.0 中增加了一个新类 Robot，这个类包含满足屏幕图形要求的方案。

Public BufferedImage createScreenCapture(Rectangla screenRect)

这个方法不是静态的，因此首先要用 Robot 的两个构造函数之一将 Robot 实例化：

public Robot()throws AWTException

public Robot(GraphicsDevice screen)throws AWTException

可能抛出的 AWTExceptions 屏幕反映了 Robot 名称的含义：这个类帮助自动测试 GUI。如果检查 Robot 的完整 API（通过联机 Java API 文档），则可以看到 API 使程序可以触发 GUI 输入（键与鼠标事件），就像是人与物理键盘和鼠标交互时产生的输入一样（第 3 章“交互性与用户界面”用 Robot 类实现鼠标观看特性）。createScreenCapture()方法不是设计成支持游戏开发的，而是用于自动 GUI 测试框架，直观地检查编程的用户界面输入从 GUI 产生所要的图形响应。因此，Robot 表示 GUI 测试机器人（这里比机械手强些），可以击键、移动鼠标和浏览这些操作的结果。因此，Robot 构造函数签名中的 AWTExceptions 是为什么抛出的呢？并不是所有宿主平台都支持直接注入伪输入事件到 GUI 事件子系统中。在这些系统中，实例化 Robot 会失败，抛出 AWTExceptions 异常。

介绍 Robot 的用途和名称来历后，下面调用 Serializable 方法，看看它是否名符其实。清单 15.4 所示的 GrabScreen 例子包含下列 grabScreen 方法。

清单 15.4 grabScreen()方法

```
/*********************************************************************
*抓取显示屏幕
*
*@返回 BufferedImage，包含指定屏幕子矩形
*********************************************************************/
private BufferedImage grabScreen (Rectangle screenRect){

    Robot robot = null;
    try{
        Robot = null Robot();
    }catch (AWTException awtException){
        System.out.println (awtException);
        Return noll;
    }
```

```
        BufferedImage screenImage = robot.createScreenCapture(screenRect)

    Return screenImage
}
```

如果运行演示程序，则可以看到屏幕上重叠了变暗的过滤效果。这是通过 grabScreen 方法建立原屏幕的拷贝，然后用亮度减半的像素覆盖显示屏幕而得到的（将红绿蓝成分减半，见清单 15.5）。

清单 15.5　RGB 成分减半

```
screenImage = grabScreen(screenRect)

...
/*****************************************************************************
*覆盖 Component.paint()


* @param g 要绘制的图形描述表
*****************************************************************************/
public void paint (Graphics g){

        if (screenImage != null){
            g.drawImage(screenImage, 0, 0, null);
        }
}
/*****************************************************************************
*将 BufferedImage 变暗

*****************************************************************************/
public void   darkenImage(BufferedImage image){
        int width = image.getWidth();
        int height= image.getHeight();

            for(int y = 0; y< height;y++){
            int color = image .getRGB(x,y);
            color = color>>1;
            color = color & 0x7F7F7F;
            image.setRGB(x,y,color);
            }
    }
```

```
}
}
```

虽然，在任何支持完全 Robot 实现的 Java 平台上，拷贝屏幕很简单。

下一个目标是将这个屏幕图形变成略图图形。

15.3.1 生成屏幕略图图形

对保存当前游戏屏幕全屏图形的 BufferedImage，下一个目标是将这个图形变成略图图形文件，包括两个工作：缩放原因和将得到的图形保存为 GIF、PNG、或 JPEG 文件。

要缩放图形，可以在新的略图图形上画出图形的缩小版图如下：

```
BufferedImage screenThumbnail = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_RGB);
Graphics2D g = screenThumbnail.createGraphics();
g.drawImage (screenImage, 0, 0,width,height,null);
g.dispose();
```

这里 width 与 height 定义略图尺寸。

但是，使用缺省绘图提示时，从大图缩成小图会出现像素化。可以用第 8 章“纹理贴图与照明”介绍的双线性插值方法使得到的略图更光滑，不会太难看。只要先设置 Graphics2D 对象的插值绘图提示，然后再缩放图形：

```
g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
    RenderingHints.VALUE_ INTERPOLATION_BILINEAR);
```

有了略图之后，要保存到磁盘中。

15.3.2 保存图形

要把图形保存到磁盘中，可以使用 Image I/O 框架。这个框架的体系结构包括高度可插的图形绘制器系统和写入器（编码器与译码器）、相关监听器与服务提供者。好在我们不必了解这个框架的细节就可以保存几个 JPEG 图形。

Image I/O API 的 javax.imageio 根包中包含一个实用程序类 ImageIO，提供了一些高级静态方便方法，可以按很容易选择的图形格式将图形保存到不同目的地：

```
public static boolean write(RendereImage im,String formatName,File output)
public static boolean write(RendereImage im,String formatName,OutputStream output)
public static boolean write(RendereImage im,String formatName,ImageOutputStream
output)
```

三个方法都返回一个布子值，表示图形写入操作是否成功，如果由于 I/O 问题造成操作失败，则还可以抛出 IOException。

这些方法的 RenderedImage 变元与 Image 类型不同，但很接近：BufferedImage 实现 RenderedImage 接口，略图是 RenderedImage，因此就可以保存图形略图了。

formatName 变量应为 jpeg、png、或 gif 之类简单字符串。

利用上述 write()方法，可以用下列代码保存单个略图：

```
File snapshotFile = new File(..);
```

```
Try{
ImageIO.write (thumbnailImage, "jpeg" ,snapshotFile);
}catch (Exception exception){
    //appropriate exception handling here
}
```

缺省情况下，它将图形保存为中等质量 JPEG 文件。

综合起来，可以得到下列全新的高级 saveScreenSnapshotAsThumbnail()程序，可以将屏幕图形保存为 JPEG 略图图形文件（见清单 15.6）。

清单 15.6　saveScreenSnapshotAsThumbnail()方法

```
/*****************************************************************************
*取得显示屏幕，将其变成略图，并保存为 JPEG 格式
*
*@param screenRect 定义屏幕的矩形
*@param thumbnailDimension 所要略图尺寸
*@param thumbnailFilename 生成的 JPEG 文件名
*****************************************************************************/
private void saveScreenSnapshotAsThumbnail(Rectangle screenRect,Dimension
                                            thumbnailDimension,String
                                            thumbnailFilename){
 BufferedImage screenImage = grabScreen (screenRect);

        BufferedImage screenThumbnail = new BufferedImage(
                thumbnailDimension.width,
                thumbnailDimension.height,
                BufferedImage.TYPE_INT_RGB);
        //缩放图形
         Graphics2D g = screenThumbnail.createGraphics();
         g.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
              RenderingHints.VALUE_INTERPOLATION_BILINEAR);
         g.drawImage(screenImage, 0,0,
              thumbnailDimension.width,
              thumbnailDimension.height,null);
              g.dispose();
    try{
         ImageIO.write(screenThumbnail),"jpeg",new File (thumbnailFilename) );
}catch (Exception ignored){
    System.out.println("OOPS" + ignored);
}
```

```
}
```

15.4 将游戏保存到正确目的地

前面只是把序列化对象和 JPEG 略图转储到当前目录。对于高质量游戏，这个方法通常是无法接受的。游戏者希望程序产生的文件能放在整齐组织的文件夹中，放在一个根目录中保存所有游戏文件。一个问题是“在哪里生成根目录呢？”

每个 Java 运行环境都定义一组系统属性，可以通过静态 System.getProperty()程序访问：

String getProperty (String propertyName)

要取得完整属性表，可以调用相关 getProperty()方法（复数），但在游戏保存中，只要知道有个系统属性 user.home，这是个字符串属性，保存当前用户主目录的文件系统路径。在 Linux 与 Mac OSX 系统中，这个目录形如/home/smith。在多用户 Windows 系统（如 Windows XP 与 Windows 2000）中，目录形如 C:\Documents 与 Settings\smith，而在旧的单用户 Windows 系统（如 Windows 98）中，目录通常是 C:\WINDOWS。利用 user.home 可以生成综合游戏目录，保存游戏生成的所有文件。可以用 File.mkdir 帮助生成目录。

清单 15.7 显示了如何实现这个方法。

清单 15.7 SaveGameState.java

```
import java.io.*;

class SaveGameState {

private final static String GAME_ROOT     = "MyGame";
private final static String SAVED_GAMES = "saved";

private void saveGame() {

      File gameDirectory = createGameDirectory();

      saveGameStateTo(gameDirectory);
}

private File createGameDirectory() {

      String userHome = System.getProperty("user.home");

      File dir = new File(userHome);

      dir = new File(dir, GAME_ROOT);
```

```
        if ( ! dir.exists() ) {
            dir.mkdir();
        }

        dir = new File(dir, SAVED_GAMES);
        if ( ! dir.exists() ) {
            dir.mkdir();
        }

        return dir;
    }

    private void saveGameStateTo(File gameDirectory) {
        // 序列化和将所有游戏状态对象存放在指定目录中
    }

    public static void main (String[] args) {
        new SaveGameState().saveGame();
    }
}
```

这个例子的关键是 createGameDirectory()方法。首先要确定用户主目录。从文件系统的用户主目录位置，可以在层次中一次一支增加新分支和子分支。最终得到的 File 对象指向一个目录，可以安全和顺利存储游戏状态。

15.5 小结

在当今竞争激烈的游戏市场上，一定要在游戏中增加保存与重装特性。Java Serialization API 提供了强大的标准机制，可以将大多数游戏内部状态保存到磁盘中或当前执行 Java 虚拟机外部的任何其他目标中。

但是，Java Serialization API 的代价是难写、复杂和问题不少。本章介绍了如何在利用这个 API 的同时清醒地看到最常见问题。

游戏保存的最后一个方面要求另外两个与序列化无关的 API：Java 屏幕抓取 Robot 和 Image I/O 框架的高级 JPEG 保存功能。

第三部分　调整与完成游戏

第 16 章　优化技术
第 17 章　生成游戏图形与声音
第 18 章　游戏设计与最后 10%
第 19 章　未来

第 16 章　优化技术

关键问题

- 优化规则
- 剖析
- HotSpot
- 优化技巧
- 内存使用与回收
- 感觉性能
- 小结

早在 19 世纪，意大利经济学家 Vifredo Pareto 发现许多国家财富分布的一个规律：80%的财富控制在 20%的人手中。

Pareto 的规律称为 80：20 规则，从此以后，这个基本思想应用到各种不同领域，包括商业销售、项目管理和计算机科学。

计算机科学？这个规则的计算机科学版指出：80%的程序时间花在 20%的代码上。

好在 HotSpot 虚拟机能够找到这 20%的“热”代码，将其编译成优化机器码。

但是，HotSpot 并不是什么都做好了，它并不进行每种可能的优化，也不能将代码改写成使用更高效的算法。

本章介绍如何找到这 20%，HotSpot 能够和不能进行哪些优化，以及内存使用与回收问题。

本章没有介绍长度优化（尽管把类缩小）。掩盖能减少代码长度（见第 18 章“游戏设计与最后 10%”），但对于大游戏，主要空间用在贴图、图像、三维模型等资源，而不是用在代码。

16.1　优化规则

首先要注意：优化方法有时比非优化方法可读性差很多。例如，第 8 章“纹理贴图与照明”的纹理贴图代码未优化时只有几行，但最后优化版本变成好几页。尽管代码速度更

快，但可读性更差，更难维护。或曾经把方法优化到自己都忘了它是怎么工作的！

另外，手工优化代码可能引入错误，如少量缺陷和不同情形的不同行为。简而言之，优化代码可能带来混乱。

因此，优化时一定要考虑下面几点：

- 不要优化！首先要尽量生成易读、易维护和易修改的代码。
- 只优化绝对必须优化的方法~20%。
- 优化一个方法时，最好保存原先的再来优化版本，以备不时之用。
- 用测试案例保证优化方法与非优化方法的行为相同。
- 不要重复 HotSpot 已经进行的优化。
- 如果优化之后没什么好处，取消优化。

下面先要寻找这 20%。

16.2 剖析

无论是把幽灵移到显示器中还是绘制三维纹理贴图多边形，游戏总会在少量代码中花掉大部分时间。

尽管我们通常能猜出代码执行时间主要花在哪里，但有时也会出现意外。你以为很慢的方法其实可能并不慢，你以为很快的方法其实可能不快。

16.2.1 基准测试

自己的基准测试方法可能很不准确。可以用下列方法建立基准测试，看看其需要多长时间。

```
int count = 1000000;
long startTime = System.currentTimeMillis();
for (int i=0; i<count; i++){
        myCoolMethod();
}
long totalTime = System.currentTimeMillis() - startTime;
System.out.println("total time: " + totalTime);
```

这个代码一百万次调用一个方法，看看它需要多长时间。

这里编写了一个微型基准测试程序。这个基准测试的结果受许多因素影响，如当前内存条件和系统定时器的精度（见本章稍后介绍）。此外，一个方法可能在一种情形中很好，另一种情形中很差，如最后游戏中。另外，HotSpot 进行的优化可能适合基准测试，但不适合实际游戏。简而言之，最好对整个应用程序进行剖析，而不是对一个方法进行基准测试。

16.2.2 使用 HotSpot 剖析器

HotSpot 剖析器可以方便地检查时间花在哪里，是代码中还是 Java API 中。可以在运行游戏时使用-Xprof 标志：

```
Java - Xprof MyCoolGameClass
```

游戏退出时，剖析器将剖析信息发送到控制台。这些实际结果比手工基准测试更详细、更可读、更容易测量。

剖析数据分解到每个线程。每个线程具有该线程中调用的解释、（HotSpot）编译和自然方法信息。

HotSpot 将编译方法编译成优化代码，解释方法是 HotSpot 没有编译或优化的方法，自然方法是自然代码，通常是操作系统特定代码，HotSpot 不会去碰它。

清单 16.1 的例子是第 11 章“碰撞探测”中三维引擎主线程的剖析器输出。

清单 16.1　剖析主线程

```
Flat profilte of 179.23 secs (16583 total ticks):main
  Interpreted    +  native    Method
  2.3%        0  +     379    sun.java2d.loops.Blit.Blit
  0.2%        2  +      32    sun.awt.windows.Win32SurfaceData.flip
  0.1%        0  +      21    com.bracken.javagamebook.graphics.
                                   ScreenManager.setFullScreen
  0.1%       17  +       0    com.bracken.javagamebook.test.
                                   GameCore.gameLoop
  0.1%        0  +      10    sun.java2d.loops.MaskFill.MaskFill
  0.0%        6  +       0    sun.awt.windows.WComponentPeer.flip
  0.0%        0  +       5    sun.java2d.loops.DrawGlyphList.
                                   DrawGlyphList
  0.0%        0  +       5    java.io .WinNTFileSystem.getLength
  0.0%        1  +       3    java.lang. System.currentTimeMillis
  0.0%        0  +       4    java.lang.Thread.sleep
  3.5%       76  +     509    Total interpreted (including elided)
  Compiled        +  native    Method
  59.9%    9933  +       0    com.bracken.javagamebook.graphics3D
                                   FastFexturedPolygonRenderer$
                                   ShadedSurfaceRenderer.render
  8.1%     1346  +       0    com.bracken.javagamebook.render
                                   BSPRenderer.setScanDepth
  2.6%      435  +       0    com.bracken.javagamebook.bsp2D
                                   BSPRenderer.drawCurrentPolygon
  2.3%      384  +       0    com.bracken.javagamebook.graphics3D
                                   ScanConverter.convert
  1.7%      289  +       0    com.bracken.javagamebook.graphics3D
                                   SortedScanConverter$SortedScanList.add
  1.7%      278  +       0    com.bracken.javagamebook.graphics3D
                                   SortedScanConverter.isFilled
```

```
1.3%    218  +    0   com.bracken.javagamebook.graphics3D
                          SortedScanConverter$SortedScanList.add
1.0%    164  +    0   com.bracken.javagamebook.graphics3D
                          SortedScanConverter. Convert
0.7%    111  +    0   com.bracken.javagamebook.graphics3D
                          SortedScanConverter$
                          SortedScanList.insert
0.5%     87  +    0   com.bracken.javagamebook.math3D
                          Vector3D.rotateY
0.5%     80  +    0   com.bracken.javagamebook.math3D
                          Vector3D.rotateZ
0.5%     78  +    0   com.bracken.javagamebook.math3D
                          Vector3D.rotateX
0.4%     70  +    0   com.bracken.javagamebook.graphics3D
                          ScanConverter.clearCurrentScan
0.4%     60  +    0   com.bracken.javagamebook.math3D
                          ViewWindow.project
0.4%     60  +    0   com.bracken.javagamebook.math3D
                          Polygon3D.isFacing
0.3%     54  +    0   com.bracken.javagamebook.math3D
                          Polygon3D.clip
87.2%  14449  +    6   Total compiled (including elided)
          Stub + native    Method
8.1%      2  +   1343   sun.java2d.loops.Blit.Blit
0.5%      0  +     87   sun.awt.windows.Win32SurfaceData.flip
0.1%      1  +     12   java.lang.Float.floatToIntBits
0.0%      1  +      6   sun.awt.image.BufImgSurfaceData
                          getSurfaceData
0.0%      0  +      6   java.lang.StrictMath.pow
0.0%      2  +      3   java.lang.StrictMath.floor
0.0%      1  +      3   java.lang.Runtime.totalMemory
0.0%      0  +      3   java.lang.StrictMath.atan2
0.0%      0  +      2   java.lang.Runtime.freeMemory
0.0%      0  +      1   sun.java2d.loops.MaskFill.MaskFill
8.9%      8  +   1467   Total stub

Runtime    stub +   native   Method
0.1%      18  +      0   interpreter_entries Runtime1 stub
0.0%       6  +      0   alignment_frame_return Runtime1 stub
```

```
0.1%        24  +        0   Total runtime1 stubs
Thread-local  ticks:
0.0%         8              Blocked(of total)
0.1%        12              Class loader
0.1%        17              Interpreter
0.0%         7              Compilation
```

上方显示了线程运行的时间量（179.23 秒）。

这个清单主要显示每个线程所花的时间量占总时间的百分比，显示解释、编译和自然方法的 tick 数。tick 数是每个方法中所花的时间量。

下方可以看到“Blocked(of total)”为 0.0%，这是这个线程执行受阻的时间量。线程执行受阻的原因很多，如休眠或等待 I/O，或另一线程正在运行。由于这个线程执行受阻的时间量很小，非受阻时间量很大，因此可以看出这个线程占用了游戏的大部分处理器时间。

如果根据清单 16.1 的剖析器输出优化这个代码，则首先是纹理贴图，因为它受阻到 59.9%。另外，整个扫描转换过程涉及几个方法，共达 9.1%，使所有扫描转换方法所花的时间多于单个 setScanDepth()方法（8.1%）。简而言之，扫描转换比设置 Z 缓存占用更多时间。

注意：HotSpot 认为编译方法是“热”代码。对长时间运行的游戏，解释代码所花的时间并不太多。

剖析列出所有线程的信息，而不只是所生成线程的信息。可以看到 AWT-EventQueue-0 与 AWT-Shutdown，等等。这些线程大多数时间受阻。

另一个要显示的线程是 AWT-Windows，见清单 16.2。

清单 16.2 剖析 AWT-Windows 线程

```
Flat profilte of 179.23 secs (16583 total ticks):AWT-Windows
   Interpreted  +  native    Method
99.9%        0  +   16699    sun.awt.windows.Wtoolkit.eventLoop
 0.0%        1  +       1    java.util.EventObject.<init>
 0.0%        0  +       2    sun.awt.PostEventQueue.postEvent
 0.0%        1  +       0    java.awt.event.ComponentEvent. <init>
 0.0%        0  +       1    sun.awt.SunToolkit.wakeupEvent. <init>
 0.0%        0  +       1    java.awt. EventQueue.wakeup
 0.0%        1  +       0    java.awt.AWTEvent. <init>
 0.0%        1  +       0    java.util.WeakHashMap.expungeStaleEntries
99.9%        4  +   16704    Total interpreted

   Compiled  +  native    Method
0.0%        0  +       2    sun.awt.AWTAutoShutdown.setToolkitBusy
0.0%        1  +       0    java.awt.event.InputEvent.getModifiers
0.0%        1  +       0    java.awt.event.GraphicsEnvironment.
```

```
                                        getHeadlessProperty
0.0%        0  +        1    java.awt. EventQueue.wakeup
0.0%        2  +        3    Total stub

Runtime stub  +   native   Method
Thread-local  ticks:
0.0%        2              Blocked (of total)
0.0%        6              Compilation
```

可以看出，这个线程也是受阻时间为 0.0%。是否 AWT-Windows 线程占用处理器时间，eventLoop()自然方法占用了 99.9%的时间？

当然不是，看来剖析器也不一定准。这里 eventLoop()方法大多数时间受阻，等待事件，但它在自然代码中受阻（自然列中显示其 16699 tick）。剖析器不能判断线程是在自然代码中受阻与否，因此这类情况下无法估计自然代码中所花的时间。

最后，剖析器输出的末尾是个总结：

```
Global summary of 181.75 seconds:
100.0%    16830          Received ticks
  0.1%       13          Received GC ticks
  0.1%       25          Compilation
  0.4%       75          Other VM operations
  0.1%       13          Class loader
  0.1%       18          Interpreter
  0.0%        8          Unknown code
```

从这个总体可以看出，内在回收、编译和虚拟机所做的其他工作所花的时间很少。

另一种剖析代码的方法是使用-Xrunhprof 标志，可以得到更详细的信息，可以监视堆，但其格式的可读性差。要得到这个标志的所有选项，可以运行下列命令：

```
java  -Xrunhprof: help
```

16.3 HotSpot

众所周知，Java 源代码不是编译成自然机器码，而是编译成类文件形式的字节码。

最初，Java 虚拟机是个解释器，在循环中一个字节码一个字节码进行译码。

后来出现了快速即时（JIT）编译器，将字节码编译成自然代码。即时编译器就是在运行时编译的快速编译器。装入 Java 类时，它迅速编译，即使类中的方法只使用一次。

HotSpot 是个动态优化编译器。利用 HotSpot 时，先解释方法。一个方法调用足够多次时（即够“热”时），HotSpot 进行编译和优化。另外，HotSpot 还可以通过全局分析更好地优化、在程序运行时调整优化和进行更多内联。内联可以减少调用常用方法的开销。

HotSpot 虚拟机有两个：客户机和服务器。服务器比客户机多几个优化。但是，SDK 包括客户机和服务器虚拟机，而 JRE 只包括客户机虚拟机。

除了几个 Java 特定优化之外，HotSpot 还实现了 C/C++编译器中常见的许多标准优化，

如死亡代码消除、循环不变项提升、公共子表达或消除、常量传递、方法内联和循环展开。我们一一介绍这些优化，读者不必进行这些优化。

16.3.1 Java 特定优化

HotSpot 进行一些 Java 特定优化：

- 快速同步。如果两个线程不运行同一方法（争夺），则方法调用的速度相当于正常的非同步方法调用，但同步方法不能内联。
- Null 检查消除。如果 HotSpot 能证明字段不会为 null，则可以在优化代码中消除 Null 检查。
- 范围检查消除。同样，如果 HotSpot 能证明数组访问不会越界，则可以在优化代码中消除范围检查。

下面几节介绍 HotSpot 对例子代码进行的性能优化。当然，HotSpot 处理字节码和机器码，而不是源代码。但我们将介绍 HotSpot 工作的相应源代码版本。

16.3.2 死亡代码消除

死亡代码消除方法消除对代码执行无效果的代码。以下列函数为例：

```
public int function(int   x){
      int I = 100*x*x;
      return x
}
```

这里设置了变量 i，但没有使用，因此 HotSpot 会消除死亡代码：

```
public int function (int   x){
      return x
}
```

16.3.3 循环不变项提升

不变项就是不变的项目，如常量。循环不变项提升（也称为循环不变代码移动）实际上就是把不变项移到循环外面，使其不必一遍遍重新计算。以下列循环为例：

```
for   (int i= 0; i<array.length; i++){
        int y = 4*x*x + 2*x;
        array [i]  = y + i
}
```

这里 y 值是个不变项，在循环中不改变数值，因此可以把 y 的计算移到循环外面：

```
int y = 4*x*x + 2*x
for(int i =0; i<array.length; i++){
        array [i]  = y + I
}
```

16.3.4 公共子表达式消除

HotSpot 可以消除公共子表达式，例如：

```
int a = 4*x*x + 2*x +3;
int b = 4*x*x + 2*x +5;
```

这两个方程大部分内容可以预先计算：

```
int temp = 4*x*x + 2*x
int a = temp   +   3;
int b = 4*x*x +   5;
```

公共子表达式消除可以增加代码可读性和修改子表达式时的代码可维护性，因此读者可以自己进行这个优化。

16.3.5 常量传递

常量乘以常量还是常量，这是常量传递的基本思想，涉及常量（而不涉及变量）的表达式只计算一次，而不是重复计算。例如下列代码：

```
int x = 10; // constant
int a = 5*x;   //therefore, also   constant
```

可以编译成：

```
int a = 50;
```

16.3.6 循环展开（仅支持服务器虚拟机）

下面是个典型循环：

```
for (int i = 0;i< array.length; i++){
        array[i] *=y
}
```

可以“展开”成多次执行每条语句：

```
//(4 是 array.length 的因子)
for (int i = 0;i< array.length; i+=4){
        array[i] *=y
        array[i+1] *=y
        array[i+2] *=y
        array[i+3] *=y
}
```

这样可以减少循环本身的开销，从而加快速度：例子中四次检查循环末尾和递增计数器。

但是，这个特性仅支持服务器虚拟机。

16.3.6 内联方法

HotSpot 的一个重要功能是积极内联功能。内联代码可以减少方法调用的开销。以下列代码为例：

```
for (int i = 0;i< array.length; i++){
        array[i] = mathObject.calc(i);
}
...
public class MathClass{
      public void calc(int   x){
            return x * 5;
      }
}
```

其内联代码如下：

```
for (int i = 0;i< array.length; i++){
        array[i] = x * 5
}
```

积极内联功能是 HotSpot 的一个重要功能。在 C++之类语言中，所有方法缺省都是非虚方法，即子类不能覆盖，而 Java 中正好相反，所有方法缺省都是虚方法，要声明为 final 才表示非虚方法。好在 HotSpot 可以在运行时分析类，确定方法是否用另一个类覆盖（即是真虚的），如果不是，则这个方法可以内联，即使没有声明为 final。

第 8 章的例子可以调整代码，让 HotSpot 进行内联。在这个例子中，有一个抽象 Texture 和两个子类 PowerOf2Texture 与 ShadedTexture。但是，引用 Texture 而不是其子类之一时，HotSpot 不能内联虚方法 getColor()。但 PowerOf2Texture 与 ShadedTexture 类的 getColor() 方法不是虚方法，因此显式引用这些子类时 HotSpot 可以进行内联。换句话说，不是用下列语句：

```
Texture texture;//有虚方法
```

而是用下列语句：

```
PowerOf2Texture texture;//无虚方法
```

然后 HotSpot 可以内联 getColor()方法。

HotSpot 还不能内联另外两种方法。简而言之，HotSpot 不能内联下列方法：

- 显式抛出异常（非检验方法没问题，如 NullPointerException）。
- 同步方法。
- 另一个类覆盖的方法（真虚方法）。

16.4 优化技巧

本节介绍一些 HotSpot 没有利用的优化技巧，其中许多是强度减少优化，就是更有效的计算函数的方法。我们还要介绍查找表与定点算法。

16.4.1 算法

但首先要介绍算法。选择好的算法可以比采用其他优化技巧得到更好的性能。例如用 BSP 树而不是按 z 深度排序多边形或将对象布置成减少计算碰撞测试。用自己的头脑创造

性地思考：使用“强力”算法时，通常总有更好的方法。

下面是 Java 集合类中使用正确算法的几个例子：

- 使用更快的排序算法，如快速排序而不是冒泡排序。

Arrays.sort()与 Collections.sort()使用调整快速排序算法。

- 用散列图快速搜索，用散列图比对表迭代更快。
- 选择正确的表：LinkedList 插入和删除更快，而 ArrayList 随机访问更快，Vector 类与 ArrayList 相似，但具有同步方法。

对于表，还要考虑如何遍历。由于 ArrayList 随机访问更快，因此可以用下列代码：

```
//适用于 ArrayList，对 LinkedList 较慢
for (int i = 0;i< array.length; i++){
        System.out.println(list.get(i));
}
```

但这不太适用于 LinkedList，因为 get()方法要搜索表以找到所要索引。可以用 Iterator 遍历 LinkedList 中的项目：

```
//更适合
Iterator i = list.iterator();
while (i.hasNext()){
        System.out.println(i.next())
}
```

除了这些基本例子外，还可以阅读一本算法教材（也许必须阅读）。

16.4.2 移位

记住，在计算机内部，数字存放成二进制值，如 1001011，对于二进制算术，可以进行各种方便的操作，如移位。

如果向左移位，则等于乘以 2。在 Java 中，向左移位的代码如下：

```
(x << 1)
```

这个语句左移 1 位，相当于乘以 2。同样，移两位则等于乘以 4，移三位则等于乘以 8。

总之，乘以 2 的指数可以用移位约简。下列语句：

```
int a = x * 32;
```

改成移如下：

```
int a = x << 5;
```

同样，除法也一样，只是向右移位。下列除法：

```
int a = x * 32;
```

变成

```
int a = x >> 5;
```

这个优化的作用随处理器而不同。有些处理器的整数乘除已经很快，因此速度增加不明显，而有些处理器上则有明显改进。

16.4.3 求模

如果对 2 的指数求模，例如：

```
int a = x % 32;
```

则可以改用超快的&运行符。它使用位掩码，位掩码就是模数减 1。例如，32 的二进制值为 100000，其位掩码为 011111（即十进制值 31）。因此，利用&运行和位掩码可以得到：

```
int a = x & 31;
```

注意：这只适用于手工的指数。

16.4.4 乘法

有时要计算一系列乘法，例如：

```
for (int i = 0;i< array.length; i++){
    array[i] = x*i;
}
```

本例中，如果 x 是 5，则结果为 0，5，10，15，20，等等。可以看出其中的模式，不必在循环的每次迭代中相乘，而可以改用比乘法快得多的加法。这里，从 0 开始，在循环的每次迭代中加上 x：

```
value = 0;
for (int i = 0;i< array.length; i++){
    array[i] = value
    value+=x
}
```

16.4.5 指数

乘法可以简化为加法，而指数也可以简化为乘法。例如。

```
for (int i = 0;i< array.length; i++){
    array[i] =(int)Math.pow(x,i);
```

如果 x 为 5，则结果为 1，5，25，125，625，等等。这里每个数只要将上一个数乘 5，与其用成本很高的 pow()函数，不如用一系列乘法：

```
value = 1;
for (int i = 0;i< array.length; i++){
    array[i] = value;
    value*=x;
}
```

乘法与指数的例子表明要识别一系列数的模式，根据前面的结果进行计算，而不是从头开始计算。其他数字模式也要学会利用这种思路。

16.4.6 更多循环不变项提升

循环内的代码移得越少越好，因为循环中的语句是重复执行的。HotSpot 本身不能进行每个循环不变项优化。下面举两个例子。

第一个例子中，循环的每次迭代中执行 if 语句，即使语句结果在循环中并不改变：

```
for (int i = 0;i< array.length; i++){
    if (bool){
        doSomething();
    }
}
```

可以把 if 语句移到循环外面：

```
if (bool){
    for (int i = 0;i< array.length; i++){
        doSomething();
    }
}
```

第二个例子中，每次画一条直线时生成相同的 Color 对象：

```
for (int i = 0;i< array.length; i++){
    g.setColor(new Color   (oxff3366));
    g.drawLine(i*10 ,0, i*10+100,100);
    g.setColor(new Color   (oxff3366));
    g.drawLine(i*10+1, 0, i*10+101,100);
```

可以把对象生成语句移到循环外面：

```
Color   red = new Color   (oxff3366));
Color   blue = new Color   (ox3366ff));
for (int i = 0;i< array.length; i++){
    g.setColor(red);
    g.drawLine(i*10 ,0, i*10+100,100);
    g.setColor(blue);
    g.drawLine(i*10+1, 0, i*10+101,100);
}
```

16.4.7 查找表

另一个古老的优化技巧是对计算量很大的函数使用查找表。查找表计算计算量很大的函数的许多值并将结果存放在一个表中。后面游戏只要从表中取值，而不是每次重算，这样要多占一些内存，但可以提高运行速度。

例如，与其计算下列方程：

```
y = computeFunction(x);
```

不如从表中取值，其中 x 是表中的索引：

y = table[x];

查找表的缺点是预先计算的数值个数有限。但许多情况下，我们并不需要每个可能的值。

例如，可以生成正弦和余弦查找表。对这些三角函数，通常要计算 0 到 2π 的弧度值，但可能用 0 到 2π 的值作为数组索引。这些数之间只有 6 个整数，而我们需要更多预计算值。

这时可以用自己的系统而不用弧度。在这个系统中，可以生成 0 到 4095 的 4096 个预计算值。数值 2048 相当于 π，4096 相当于 2π。要取得直观图形，请看图 16-1。

我随机选择 4096，因为 256 可能太粗，而 4096 可以得到较好结果。

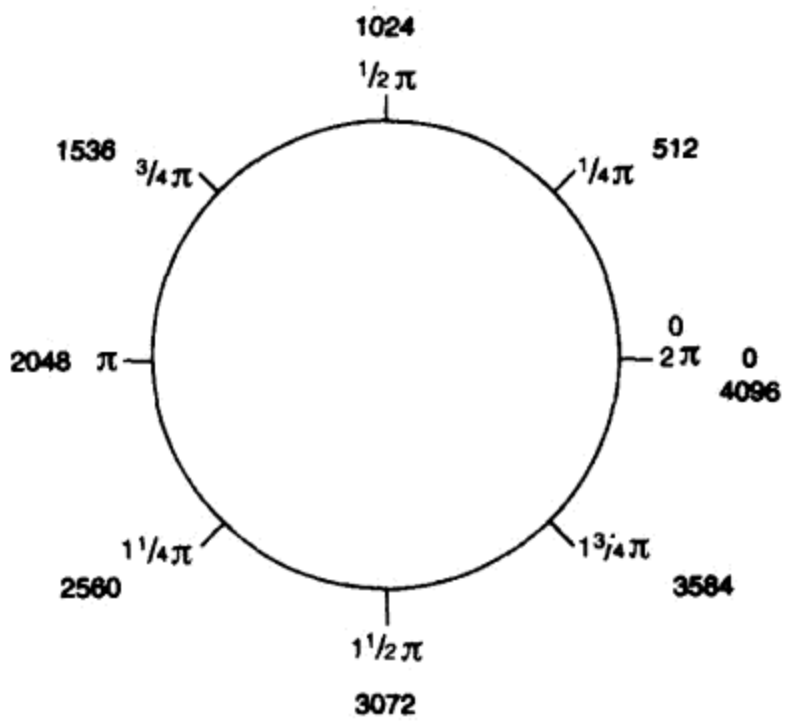


图 16-1 弧度（0 到 2π）到系统（0 到 4096）的换算

由于 4096 是 2 的指数，很容易使数值保持在有效范围内。如果要求-200 的余弦或 5000 的正弦，可以用&运算符和位掩码进行换算：

x = x & 4095

还可以利用下列方程：

cos(x) = sin (π /2-x)

利用这个方程，可以从正弦查找表中取得一个值的余弦。这样，正弦和余弦只要一个表，而不是两个表。

下面要实现查找表。清单 16.3 显示了完整代码，是 MoreMath 类的一部分。

清单 16.3 MoreMath.java 中的三角函数查询函数

```
package com.brackeen.javagamebook.util;

/**
    MoreMath 类提供了 java.lang.Math 和 java.lang.StrictMath 类中没有的函数
*/
```

```
public class MoreMath {

    // 4096 个项目的三角表
    private static final int TABLE_SIZE_BITS = 12;
    private static final int TABLE_SIZE = 1 << TABLE_SIZE_BITS;
    private static final int TABLE_SIZE_MASK = TABLE_SIZE - 1;
    private static final int HALF_PI = TABLE_SIZE / 4;
    private static final float CONVERSION_FACTOR =
        (float)(TABLE_SIZE / (2 * Math.PI));

    private static float[] sinTable;

    // 装入这个类时的初始三角表
    static {
        init();
    }

    private static void init() {
        sinTable = new float[TABLE_SIZE];
        for (int i=0; i<TABLE_SIZE; i++) {
            sinTable[i] = (float)Math.sin(i / CONVERSION_FACTOR);
        }
    }

    /**
        余弦函数，角度为 0 到 4096 而不是 0 到 2pi
    */
    public static float cos(int angle) {
        return sinTable[(HALF_PI-angle) & TABLE_SIZE_MASK];
    }

    /**
        正弦函数，角度为 0 到 4096 而不是 0 到 2pi
    */
    public static float sin(int angle) {
        return sinTable[angle & TABLE_SIZE_MASK];
    }

    /**
```

```
        弧度（0 到 2pi）到系统（0 到 4096）的换算
    */
    public static int angleConvert(float angleInRadians) {
        return   (int)(angleInRadians * CONVERSION_FACTOR);
    }
```

装入这个类时，调用静态 init()方法，启动正弦查找表。这里还在 angleConvert()方法中提供了将弧度换算为类中所用系统的函数。

sin()与 cos()函数返回指定角度的浮点数值。

当然，查找表不限于三角函数，也可以用于更复杂的公式。还可以在服务器方使用，在数据库中缓存成本很高的查找表。尽量用表提高速度。

16.4.8 定点算术

第 7 章“三维图形”介绍了定点算术，这里不再重复。记住，浮点数算术有时太慢，特别是将浮点数换算为整数之类的操作。定点算术可以加快速度，但也有两个缺点：精度较差，两个定点数相乘会产生溢出问题。

16.4.9 异常

使用异常可能影响 HotSpot 编译和限制 HotSpot 的内联功能。

首先，不需要时别捕获异常。例如：

```
try{
    return array[index];
}
catch (ArrayIndexOutOfBoundsException ex){
  //什么也不干
}
```

可以改为自己检查边界：

```
if (index > 0 && index < array.length)
    return array[index];
}
```

当然，异常在面向对象系统中常用，通常抛出异常不会明显影响性能。例如：

```
public void takeItem(Item item)throws InventoryFullExcepetion;
```

游戏者，游戏者可能不经常选择某个项目，因此这类问题不会明显影响性能。

16.4.10 输入/输出

输入/输出可能成为瓶颈，主要是因为要等待一些低速操作，如访问硬盘或从低速 modem 读取字节。下列方法可以简化输入/输出的处理：

- 用 BufferedInputStream（对二进制文件）和 BufferedReader（对文本文件）。这些类一次读一块数据，而不是一次读一个字节。这样很有好处，不管从文件读取还是读取服务器中的流。

- 在不同于主线程的另一线程进行输入/输出。读取/写入文件和与服务器通信需要时间，因此不要在输入/输出时让游戏“冻结”。
- 可以把几个小文件合并为一个大文件，减少打开的文件数。
- 对于几个 MB 的大文件，可以使用内存贴图文件。

下面要介绍内存贴图文件。

16.4.11 内存贴图文件

内存贴图文件（Memory-mapped）可能是硬盘上的文件，贴图到一个内存区。程序读取内存贴图文件时，就像读取内存一样，在需要时从内存中复制和删除文件的局部。

要用内存贴图文件方法进行读取，只要和平常一样打开文件，然后用 FileChannel 类将其贴图到缓冲区：

```
File file = new (filename);
FileInputStream stream = new FileInputStream (file);
MappedByteBuffer buffer = stream.getChannel().map(
        FileChannel.MapMode.READ_ONLY, 0, file.length());
stream.close();
```

map()文件可以贴图文件中所要的任何部分。这里贴图整个文件。这个方法返回 MappedByteBuffer。

贴图文件之后，可以关闭原先的流（stream.close()）。

操作系统会处理文件的内存管理。通常，要开始读取文件时，才将文件内容装入内存中。

可以用任何 MappedByteBuffer 的 get()方法读取文件。下面的例子将部分文件复制到 1KB 的数组中：

```
byte[] array = new byte[1024];
buffer.get(array);
```

内存贴图文件在内存回收之后删除贴图。因此，文件用完之后，可以清除引用，使内存回收单元回收内存：

```
buffer = null;
```

内存贴图文件不一定总是提高性能。对于小文件，内存贴图文件可能更慢，顺序读取文件时，也看不到什么改进。但是，内存贴图文件特别适合需要随机访问的大文件，如数据库和具有嵌入纹理的大贴图文件。

16.5 内存使用与回收

使用内存和回收内存时，要记住两个目标：

- 编写游戏的目标系统不要用太多内存，不管是 64MB 还是 256MB。
- 尽量限制回收内存暂停。

回收内存造成的暂停会使游戏显得不连贯。但是，如果限制每帧生成的对象数和调整 Java 堆设置，则可以解决许多回收内存暂停问题。本节介绍的一些方法监视内存使用和调

整 Java 堆设置，使程序更顺畅。

16.5.1 Java 堆与内存回收

Java 内存放在堆中，堆是大块内存，由虚拟机管理。这样比让操作系统管理内存有几个好处，因为 HotSpot 会对典型 Java 应用程序优化内存管理。

缺省情况下，HotSpot 从小堆开始，如一个堆几个 MB。随着生成的对象增加，堆不断增加。

对象不再有引用时，HotSpot 可以从堆中清除这个对象，称为回收内存。

注意：有些内存在堆之外，如 NIO 缓存、内存贴图文件和显示内存中的图像。

堆分成两个部分：年轻段和终身段。对象首先在年轻段中生成。年轻段中生成的对象较快，通常只涉及指针递增。如果对象存在时间足够长，则转到终身段。

这里的思路是，大多数对象只是短时间存在。例如，Interator 对象用于遍历表中项目，通常只是短时间使用。HotSpot 能够很快清除年轻段中的无引用对象，因此生成几个短命对象不会对性能影响太大。

但终身段中清除引用对象则较慢，因为终身段很大，涉及许多长命对象。这里的回收内存暂停会很明显，最好尽量限制终身段的内存回收。

为此，要看看游戏使用多少内存，何时发生内存回收。

16.5.2 监视内存回收

内存回收会影响游戏性能，好在可以用几个方法减少这种影响。我们不能完全回避内存回收，但可以减少其影响，但首先要监视内存回收，了解内存回收单元如何处理游戏。

可以用-verbose:gc 标志取得何时发生内存回收的信息：

```
java - Verbose:gc MyCoolGameClass
[GC 16373K - >(16311K),0.0028920 secs]
[GC 17316K - >(17311K),0.0028735 secs]
[Full GC 17311K - >6412K(18568K),0.0587969 secs]
```

每一行显示一次内存回收操作。只涉及年轻段的内存回收标为 GC，涉及终身段的内存回收标为 Full GC。

第一个数（17311K->6412K）显示内存回收前面使用的内存量。括号中的数（18568K）是总堆空间（不包括一些 HotSpot 本身使用的堆）。

最后，还显示内存回收所用的时间量。可以看出，每次年轻段的内存回收只要 3 毫秒左右，对 GC 没有太大影响，而终身段的内存回收（Full GC）要大约 60 毫秒，会丢掉几帧。例如，如果画一个帧要 20 毫秒（每秒 50 帧），则内存回收占用三个帧的时间。

可以用-XX:+PrintGCDetails flag 标志得到内存回收更详细的信息：

```
java - Verbose:gc - XX:+PrintGCDetails MyCoolGameClass
[GC [DefNew:1123K - >0K(1152K),0.0027800 secs]
      14013K->14012K(16292K),0.0028174 secs]
[GC [DefNew:376K - >1K(1152K),0.0013854 secs]
      14388K->14388K(16292K),0.0014222 secs]
```

```
[GC [DefNew:1014K - >1K(1152K),0.0026643 secs]
    [Tenured: 15396K - >12400K(16152K),0.0415779 secs]
    15401K - >12400K(17304K),0.0443534 secs]
```

这里把内存回收分为年轻段（DefNew）和终身段，显示每段长度和总长度。这里，年轻段为 1152K，而终身段为 16152K。只要 2 到 3 毫秒，而终身段内存回收总共要 45 毫秒左右。

好在丢掉几帧并不明显，因此这里内存回收单元的影响不大。

可以用-XX:+PrintGCDTimeStamps 标志取得更多内存回收信息。这个标志打印每次内存回收的时间标志，很容易看出内存回收的频度。在日志中增加时间标志后，很容易了解内存回收的发生时间与代码中重要段之间的关系。

16.5.3 监视内存使用

如果每次画一帧要生成 20KB 垃圾，则每秒 60 帧会产生 1.2MB 的垃圾，每 10 秒接近 12MB。这个垃圾量很大，内存回收单元的工作量会很大。因此，一定要监视游戏中内存分配的速率，了解如何根据需要更好地调整堆。

可以用-Xaprof 标志取得每个对象使用的内存信息：

```
java - Xaprof MyCoolGameClass
Allocation profile(sizes in bytes, cutoff=0 bytes):
___Size__Instances__Average__Class_____________
39954200        590        67719     [S
2949360       10406          283     [B
2838936      118289           24     com.bracken...Vector3D
1036904        9062          114     [C
 718848        3744          192     sun.java2d.SunGraphics2D
 475008       29688           16     sun.awt.MostRecentKeyValue
 421400        7525           56     java.awt.event.MouseEvent
 298464       18654           16     java.awt.Point
 273960       11415           24     java.lang.ref.WeakReference
 245936        3942           62     [Ljava.lang.Object;
 239680        3745           64     java.awt.geom. AffineTransform
 212736        8864           24     java.awt.EventQueueItem
 203760        8490           24     java.lang.String
 161540        5047           32     java.util.LinkedList$ListItr
 149760        3744           40     sun.java2d.loops.FontInfo
 141632        8852           16     sun.awt. EventQueueItem
 130560        8160           16     com.brackeen...ScanConverter$Scan
 129600        5400           24     java.util.HashMap$Entry]
 119840        3745           32     sun.java2d.pipe.Region
 114864        4786           24     java.util. LinkedList$Entry
```

这个表中[S 是短数组，[B 是字符数组，[C 是字符数组（常用于 String 类），[Ljava.lang.Object 是对象数组常用于 ArrayList 类。

这样可以看到不同时空中存在的对象，而不一定是任何特定时间的内存使用情况。解释结果时要注意这一点。例如，所画的每帧生成一个新的 sun.java2d.SunGraphics2D 对象，所有这些对象的总长度约为 700KB，但任何时刻可能只引用其中一个对象。

虽然，其中许多对象是 Java API 自己生成的，如 WeakReference 与 MouseEvent。

本例中，短数组占用大部分空间，用于多边形表面，通常只生成一次。利用 MIP 贴图（见第 8 章）对远距离多边形生成小表面可以减少用量。

Vector3D 对象也使用大量空间，可以用两个方法加以解决：首先，可以更经常共享 Vector3D（在当前状态中，一些相邻多边形顶点相同，但使用不同的 Vector3D 实例）。第二，每个对象有大量开销。我们只要 12 字节数据（x、y、z 各 4 个字节），但对象开销园整化为 8 字节边。最后，每个对象要 24 字节。一种解决办法是生成 Vector3Dlist 对象，在如下数组中保存向量表：

Float[] x, y, z;

这样可以减少对象开销，但会产生数组访问开销。

由于-Xaprof 标志只能显示每个类的实例所用的内存，因此要用另一种方法在游戏过程中监视内存。

Runtime 类有几个方法可以监视堆使用，特别是 totalMemory()与 freeMemory()方法，其分别返回堆长度和自由堆空间量。

可以用这些方法在不同帧之间监视内存用量，以便看到每次绘制一个帧时分配多少内存（或产生多少垃圾）。要取得分配的内存量，只要使用 totalMemory()与-freeMemory()，见清单 16.4 的 MemMonitor 类。

清单 16.4 MemMonitor.java

```
/**
    监视堆长度、分配长度、分配改变、堆长度改变和探测内存回收(分配长度减少)。用 takeSample()
    方法取得当前内存状态的样本
*/
public class MenMonitor{
/**
    Data 类包含一系列浮点值的信息，可以取得数据的最小值、最大值、总和与个数
*/
public static class Data{

        float lastValue = 0;
        float min = Float.MAX_VALUE;
        float max = Float.MIN_VALUE;
        float sum = 0;
        int count = 0;
```

```
    public void addValue(float value) {
            lastValue = value;
            sum+=value;
            count++;
            min = Math.min(min, value);
            max = Math.max(max, value);
        }

        public String toString() {
            return "Min: " + toByteFormat(min) + "    " +
                    "Max: " + toByteFormat(max) + "    " +
                    "Avg: " + toByteFormat(sum / count);
        }
    }

    private Data heapSize = new Data();
    private Data allocSize = new Data();
    private Data allocIncPerUpdate = new Data();
    private int numHeapIncs = 0;
    private long startTime = System.currentTimeMillis();

    /**
        取得当前内存状态的样本
    */
    public void takeSample() {
        Runtime runtime = Runtime.getRuntime();
        long currHeapSize = runtime.totalMemory();
        long currAllocSize = currHeapSize - runtime.freeMemory();

        if (currHeapSize > heapSize.lastValue) {
            numHeapIncs++;
        }
        if (currAllocSize >= allocSize.lastValue) {
            allocIncPerUpdate.addValue(
                (currAllocSize - allocSize.lastValue));
        }

        heapSize.addValue(currHeapSize);
        allocSize.addValue(currAllocSize);
```

```
    }
    /**
        将字节数变成表示字节、KB、MB 等的字符串
    */
    public static String toByteFormat(float numBytes) {
        String[] labels = {" bytes", " KB", " MB", " GB"};
        int labelIndex = 0;

        // 确定最合适的标题
        while (labelIndex < labels.length - 1 && numBytes > 1024) {
            numBytes/=1024;
            labelIndex++;
        }
        return (Math.round(numBytes*10)/10f) + labels[labelIndex];
    }

    public String toString() {
        long time = System.currentTimeMillis() - startTime;
        float timeSecs = (float)time / 1000;
        return "Total Time: " + timeSecs + "s\n" +
                "Heap: " + heapSize + "\n" +
                "Allocation: " + allocSize + "\n" +
                "Allocation inc/update: " + allocIncPerUpdate + "\n" +
                "Num Heap Incs: " + numHeapIncs;
    }
}
```

在这个代码中，用 takeSample()方法取得当前内存状态的样本。游戏中，画每帧之后调用这个方法。

ToString()方法返回表示所取样本统计信息的字符串。

我用第 11 章的三维引擎和-verbose:gc 与-XX:+PrintGCDetails 标志运行测试，得到如下结果：

```
Total Time: 241.859g
Heap: Min: 10.7MB   Max: 63.6MB   Avg:   51.8MB
Allocation: Min: 5.5MB      Max: 43.3MB     Avg: 34.1MB
Allocation   inc/update:   Min:   384.0 bytes   Max:   8.5MB      Avg: 6.1KB
Num Heap Incs: 5
```

它报告 73 次年轻段内存回收和 15 次终身段内存回收，即大约 2.75 秒一次内存回收。

利用这个信息，可以了解游戏最多分配多少内存（大约 44MB）。大量内存回收是因为开始时的堆长度很小。堆很小时，虚拟机要进行大量内存回收，使一切能放在内存中。

如果内存监视精确，则画一帧好像至少要 384 字节，但每帧平均生成 6.1KB。本例中，大量分配是生成多边形表面的结果，其通常是永久的，但也生成抛射弹的新对象之类项目。

利用这个信息，我们要调整堆，减少内存回收。

16.5.4 调整堆

我们为游戏建立一个目标：每 10 秒钟一次内存回收，并使内存回收所需时间减少到每次只丢失一帧左右。

首先要选择堆长度。

注意：堆越大，内存回收所需时间越长，因此不能生成太大的堆。

在上述测试中，我们知道游戏大约使用 44MB，最大堆长度为大约 64MB，因此可以从 64MB 的堆开始。可以用-Xms 标志指定开始堆长度如下：

```
java - Xms64m MyCoolGameClass
```

然后，每帧生成大约 6.1MB，这里是个估计不一定准确，但有个大概概念。如果要每 10 秒回收内存，而且要减少回收内存的时间，则要保证 10 秒内生成的所有内存能在堆的年轻段中放下。如果每秒 60 帧，则 10 秒为 3660KB。年轻段可以从大约 4MB 开始。可以用-Xmn 标志指定年轻段开始长度如下：

```
java - Xms4m MyCoolGameClass
```

下面试试看，用下列标志再次运行测试：

```
- Xms64m   - Xms4m   -verbose:gc - XX:+PrintGCDetails
```

I 结果如下：

```
Total Time: 248.094g
Heap: Min: 63.6MB   Max: 63.6MB   Avg:   63.6MB
Allocation: Min: 10.1MB      Max: 42.7MB     Avg: 38.8MB
Allocation   inc/update:   Min:   384.0 bytes   Max:   10.1MB      Avg: 3.1KB
Num Heap Incs: 1
```

另外，它报告 19 次年轻段内存回收，分别在 2 毫秒到 16 毫秒之间，没有终身段内存回收。248 秒钟大约每 13 秒有一次小内存回收，每秒钟为 60 帧，16 毫秒的内存回收只丢掉一帧，没问题。

这样就达到目标了，但这时每帧的平均分配量降低了，是 3.1KB。这里有许多原因，但我认为多边形表面全程放在内存中，因为它有足够的堆空间。堆更小时，多边形表面要刷新和重新生成。

注意：这些测试很简单，因为演示程序只运行了大约四分钟。我们要收集更多数，如长时间玩的情形（如 20 分钟以上）和运行更多次游戏的情形，然后再最后确定堆的调整。

另外，MemMonitor 类很容易扩展成显示游戏中内存使用的直观表示。游戏退出时，类可以输出游戏从开始到结束的内存使用图，从而帮助识别典型视景的内存生成速率和导致内存使用跳跃的情形。

16.5.5 调用内存回收单元

程序中可以用下列方法强制内存回收：

System.gc();

对大多数应用程序，Sun 公司建议不要调用这个方法。但对于游戏，可以在某些情况下进行内存回收，如开始进入新关之前或启动序列之后，如果知道这段时间生成的垃圾较多，就可以内存回收，从而轻装上陈。

除了缺省内存回收单元外，HotSpot 还有几个回收单元，分别适合不同情形。下面介绍这些回收单元。

- 递增内存回收单元（-Xingc），也称为训练内存回收单元。这个内存回收随着时间分布大量内存回收，而不是在主要内存回收期间长时间暂停。但是，这个内存回收使用更多处理器功能，因此对于需要大量处理器功能的动作游戏，这种内存回收单元还适用。
- 并行内存回收单元（-XX:+UseParallelGC），也称为通过量内存回收单元，其在另一线程中进行年轻段内存回收，特别适合多处理器机器。
- 并发内存回收单元（-XX：+UseComcMarkSweepGC，在进行终身段内存回收时暂停时间较短，也特别适合多处理器机器，它在程序运行时并发进行内存回收，而不是停止程序。可以使用并行年轻段内存回收单元（-XX:+UseParallelGC）。

大多数家庭用户没有多处理器机器，因此最好使用缺省内存回收单元。但读者可以试验不同内存回收单元，看看对游戏产生何种结果。

16.5.6 减少对象生成

可以调整堆，使内存回收减少到忽略不生成的临时对象放在堆的年轻段中。但是，仍然应减少生成的对象，使内存回收的次数更少。

注意：我们不可能完全避免生成对象。Java API 中会生成许多对象。如果你想了解某个类或方法是否生成对象，可以看看 SDK 所带的 Java API 源代码，了解何时生成什么对象。

例如，从表中取一个 Itetator 时会生成新的 Itetator 对象。将对象加进 LinkedList 时会生成新对象，将其放在表中。接合字符串对象时，也会生成新对象。

几个避免这类问题的方法是用 ArrayLists 代替 LinkedList、不用 Itetator 遍历 ArrayLists 和尽量用 StringBuffer 代替 Strings。

16.5.7 对象复用

最好能尽量复用对象。在本书的代码中，我们用大量临时对象计算多边形变换等，这样，我们不必对变换的每个多边形每个顶点生成新 Vector3D 对象。

如果能够帧与帧之间使用同一对象而不是每帧生成新对象，对应尽量使用同一对象。

有时只要在内存足够时就尽量保留对象。例如对于多边形下面，当它离开屏幕时，就不再需要了，但我们要保存其拷贝，以便稍后再次使用，只在内存移用。

这时我们使用软引用。如果对象的惟一引用是软引用，则只要有足够内存保存这个对

象，内存回收单元就不回收这个对象。先回收没有引用的对象，只有堆已满而其他对象已经回收时，才回收软引用对象。这样就可以把软引用作为某种对象缓存。

软引用的用法见第 8 章介绍。

16.5.8 对象池

对象池是对象集合，可以在应用程序执行期间复用。对象在需要时从池中取出，用完之后加回池中。对象池可以避免连续分配与去配对象内存的开销。

Sun 公司不提供在 HotSpot 之类现代虚拟机中使用对象池。对象生成速度已经大大加快，如果生成许多短命对象，则堆的年轻段能够很好地处理。

可以把堆的年轻段看成，尽量把临时对象放在堆的年轻段中。如果年轻段中的对象开始溢出，则对象会移到终身段，使内存回收速度减慢。因此，发生这种情形时，应增加年轻段长度，避免溢流。

也许最糟的情形是生成大的临时对象时，其寿命很长，足以移到终身段中，要从终身段中回收内存。如果增加年轻段长度无法解决这个问题，则可以使用对象池。

另一种情形是生成对象的计算量很大时。在三维引擎中建立多边形表面的计算量很大（但这不是对象池的好例子，因为每个多边形表面各不相同）。但生成大多数对象的计算不会很大。

第三种情形是用完一些大的终身段对象而又不想将其回收时，可能要等待这一关完成之后再回收。如果有足够内存，则可以把这些对象放在池中，在更适合的时间再清理对象池。

对象池没有什么特别之处。简单地说，可以通过保存对象表来实现对象池：

```
ArrayList objectPool = new ArrayList()
```

对象池可以填满对象，可以从末尾取出新对象（从末尾取出可以保证不移动数组元素，而从开头取则要移动数组元素）。用完对象之后，将其放回表中。

也可以用软引用扩展对象池的思想。如果池中每个对象是一个软引用，则只在堆内存越界时才清除。另外，HotSpot 只在绝对区要时才回收软对象池中的项目。池可以保留，但池中的对象在需要时回收。

16.6 感觉性能

有时性能与原始数据和基准测试关系不大，而与用户感觉有关，应该看上去运行得顺些。

一种方法是保证游戏与服务器通信或装入大量文件时不要显得“冻结”。可以将这类长期任务放在另一线程中。

后台发生某种情形时，还要给用户一个直观通知，可以用进程条或等待光标。例如，可以在装入游戏时让用户干点事，如提供一个有趣动画或让用户玩简单的 pong 游戏。

定时器精度

感觉性能的最主要因素之一是系统定时器的粒度。但是，在 Windows 机器上，定时器精度并不准确。不同操作系统中 System.currentTimeMillis()的定时器粒度如下：

Windows 95/98/Me:55 毫秒

Windows NT/2000/XP:10-15 毫秒

Mac OS X:1 毫秒

Linux:1 毫秒

定时器精度为 55 毫秒时，每秒大约 18 次更新，即使游戏每秒能画 60 帧。为了了解这个工作原理，请看图 16-2，图中的灰球表示不良定时器精度，白球表示高精度定时器。

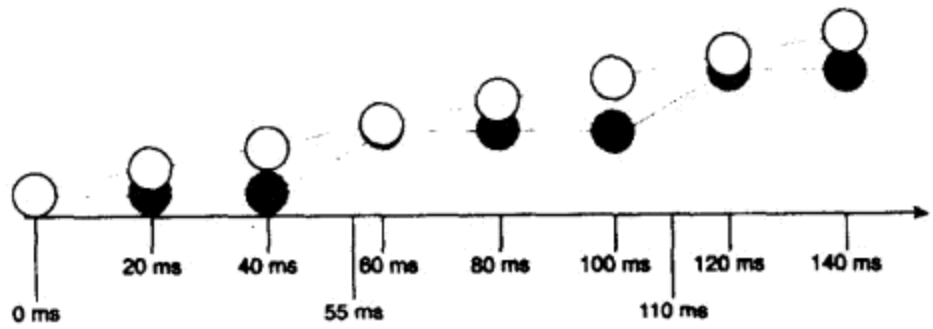


图 16-2 灰球表示不良定时器精度，白球表示高精度定时器，前者的图形更新不连贯

如果你想了解系统的定时器精度，可以运行清单 16.5 的测试。

清单 16.5 GranularityTest.java

```
/**
    测量 System.currentTimeMillis()的粒度
*/
public class GranularityTest {

    public static final int NUM_SAMPLES = 100;

    public static void main(String[] args) {
        long sum = 0;
        int count = 0;

        long lastTime = System.currentTimeMillis();
        while (count < NUM_SAMPLES) {
            long currTime = System.currentTimeMillis();

            // 改变时间时，请记录差值
            if (currTime > lastTime) {
                long granularity = currTime - lastTime;
                // 记录粒度的游和
                sum+=granularity;
                count++;
                lastTime = currTime;
```

```
            }
        }

        // 显示结果
        System.out.println("Average granularity of " +
            "System.currentTimeMillis(): " + ((float)sum / count));
    }
}
```

解决这个问题的方法之一是使用 JNI（Java 自然接口）调用一些自然代码，使用更精确的定时器。这个问题只在 Windows 中存在，因此只在 Windows 机器上要调用 JNI，而其他机器上则继续使用 System.currentTimeMillis()方法。Windows 的多媒体定时器更加精确。Java 文档中提供了 JNI 的更多信息。

但是，如果你不喜欢处理自然代码，则还有另外几种方法。其中一种是用显示器刷新率作为定时器，但这要求游戏能以这个刷新率运行。对大多数游戏而言，刷新率是变化的。

另一个思路是根据上一时间样本让定时器进行估计。为此可以求前几次定时器值的平均。例如，假设定时器表示每帧经过的时间量如表 16-1 所示。

表 16-1　时间样本

实际时间	定时器值	平均时间
第 1 帧：28 毫秒	第 1 帧：0 毫秒	第 1 帧：0 毫秒
第 2 帧：28 毫秒	第 2 帧：55 毫秒	第 2 帧：28 毫秒
第 3 帧：28 毫秒	第 3 帧：0 毫秒	第 3 帧：18 毫秒
第 4 帧：28 毫秒	第 4 帧：55 毫秒	第 4 帧：28 毫秒
第 5 帧：28 毫秒	第 5 帧：0 毫秒	第 5 帧：22 毫秒
第 6 帧：28 毫秒	第 6 帧：55 毫秒	第 6 帧：28 毫秒

这里定时器有 55 毫秒的精度。游戏运行速度为每秒大约 35 帧，或每帧 28 毫秒。平均定时器值并不完美，但比实际定时器值提供了更好的结果。由于有更多定时器样本，因此估计更准确。

下面要实现这个技术。清单 16.6 的 TimeSmoothie 类跟踪最后几个时间样本，通过平均求出当前时间值。它还有计算帧速率的函数。

清单 16.6　TimeSmoothie.java

847（5）

```
package com.brackeen.javagamebook.util;

/**
    理顺定时器精度造成的时间跳动。这个简单算法不太精确（光顺的时间可能比实际时间提前一
    些），但可以得到更好看的结果
*/
```

```
public class TimeSmoothie {

    /**
        重新帧速率的频度
    */

    protected static final long FRAME_RATE_RECALC_PERIOD = 500;

    /**
        不让帧之间的经过时间越过 100 毫秒
    */
    protected static final long MAX_ELAPSED_TIME = 100;

    /**
        取最近 100 毫秒内最后几个样本的平均值
    */
    protected static final long AVERAGE_PERIOD = 100;
    protected static final int NUM_SAMPLES_BITS = 6; // 64 samples
    protected static final int NUM_SAMPLES = 1 << NUM_SAMPLES_BITS;
    protected static final int NUM_SAMPLES_MASK = NUM_SAMPLES - 1;

    protected long[] samples;
    protected int numSamples = 0;
    protected int firstIndex = 0;

    // 计算帧速率
    protected int numFrames = 0;
    protected long startTime;
    protected float frameRate;

    public TimeSmoothie() {
        samples = new long[NUM_SAMPLES];
    }
    /**
        增加指定时间样本返回所有记录的时间样本
    */
    public long getTime(long elapsedTime) {
        addSample(elapsedTime);
        return getAverage();
```

```
    }

    /**
        增加时间样本
    */
    public void addSample(long elapsedTime) {
        numFrames++;

        // 包装时间

        elapsedTime = Math.min(elapsedTime, MAX_ELAPSED_TIME);

        // 将样本加进清单中
        samples[(firstIndex + numSamples) & NUM_SAMPLES_MASK] =
            elapsedTime;
        if (numSamples == samples.length) {
            firstIndex = (firstIndex + 1) & NUM_SAMPLES_MASK;
        }
        else {
            numSamples++;
        }
    }

    /**
        取得录制时间样本的平均
    */
    public long getAverage() {
        long sum = 0;
        for (int i=numSamples-1; i>=0; i--) {
            sum+=samples[(firstIndex + i) & NUM_SAMPLES_MASK];

            // 到达平均时间时，继续并返回平均时间
            // the average.
            if (sum >= AVERAGE_PERIOD) {
                Math.round((double)sum / (numSamples-i));
            }
        }
        return Math.round((double)sum / numSamples);
```

```
}

/**
    取得帧速率（getTime() 或 addSample()的实时调用次数）500 毫秒后重算帧速率

*/
public float getFrameRate() {
    long currTime = System.currentTimeMillis();

    // 500 毫秒后重算帧速率
    if (currTime > startTime + FRAME_RATE_RECALC_PERIOD) {
        frameRate = (float)numFrames * 1000 /
            (currTime - startTime);
        startTime = currTime;
        numFrames = 0;
    }

    return frameRate;
}
}
```

要使用这个类，只要在游戏主循环中增加一行代码，根据时针报告的经过时间返回估计经过时间：

```
public void update (long elapsedTime){
...
//光顺经过的时间
elapsedTime   = timeSmoothie.getTime(elapsedTime);

//根据经过的时间更新世界
updateWorld(elapsedTime);
}
```

这个代码适合帧速率比较一致时。虽然可以改变，但如果帧速率变化太大，则 TimeSmoothis 无法得到精确结果，这时更适合用 JNI 函数。

16.7　小结

本章简要介绍了各种优化问题与思路，我们用剖析器找出需要优化的 20%代码，然后介绍了 HotSpot 已经进行的优化，当然，HotSpot 不是什么都能做，因此我们还要进行 HotSpot 没有进行的优化。

最后，我们介绍了内存使用和内存回收单元，甚至调整堆，使内存回收的影响减少，

每 10 秒钟只在 10 毫秒左右。

当然，优化的最大改进不在计算机上，而在于你的思想。更高效的算法（如 BSP 树）可以取得一些惊人的成效，但我们要不断创新，追求更奇妙更高效的代码。

第 17 章　生成游戏图形与声音

关键问题

- 选择外观
- 取得自由版权游戏媒体
- 与艺术家和声效工程师合作
- 工具
- 生成声音
- 生成纹理与幽灵
- 生成漂亮屏幕与 HUD 图形
- 生成 UI 图形
- 生成字体
- 小结

有了奇妙的游戏思想和开始编程后，还要有图形和声音。

也许你有幸遇到了一位艺术家和声效工程师，愿意在这个项目中投入时间，但这种情况很难遇到。

本章面向编程人员图形，这类游戏图形不太漂亮，但足以用来演示。

建立任何类型的游戏媒体时，要记住下列规则：*“一半的工作都是无效劳动”*。

我个人曾经花许多时间生成完美的纹理，后来才发现用不上，只好忍痛割爱。这一点很重要：要善于放弃！当然，不一定是直接扔掉，也可以保存未用图形，以便后面参考，但不要把不适用的东西放到游戏中。

生成的游戏图形和声音用不上时，不要难受，因为你离生成用得上的媒体又接近一步了。生成的废品可以看成是勋章，而不是缺点。如果一半的工作是无效劳动，那么生成的废品越多，生成的有用材料也就越多。因此，下面开始生成一些废品吧。

17.1　选择外观

首先要确定游戏的总体外观。例如，要暗色而粗糙的样子，还是明亮而光滑的样子？要逼真还是夸张？还可以更有创造性地生成草图样子或 50 年代黑白电影的样子。

这主要取决于目标对象。成人可能喜欢有戏剧灯光和阴影的暗色游戏，而小孩可能喜欢明亮、颜色的游戏。

17.1.1　寻找灵感

有时要找到游戏外观的灵感特别难。想想你喜欢玩什么游戏，喜欢看什么电影和电视节目。科幻片通常有许多奇妙的灵感，其他电影也是，如西部片、神秘片甚至体育节目。

应从周围的世界中寻找灵感。可以根据公寓楼环境建立一个标记游戏，也可以在生成

冒险游戏时借用你家周围的森林和小溪风景。

最后，可以从艺术博物馆寻找更经典的灵感。假设要玩的游戏像梵高的《星光之夜》或玛格丽特的奇怪世界。沙瓦多.达理的温柔钟也是超现实者经常喜欢的。

除了从生活中吸取灵感外，还要记住，寻找灵感与抄袭的界线是微妙的。可以借签某个游戏和电影的灵感，但不要让人觉得太面熟。

17.1.2 保持一致

游戏中是否有些纹理像卡片，有些则暗色而逼真？最好别这样。同样，格斗游戏最好别在跳动时发生很新潮的“啵嘤”声。

最好保证游戏图形、界面图形与声音能和谐组合。例如，三维游戏的纹理应保持相似的亮度（因为照明是在游戏中完成的）。

另外，用户界面与游戏外观也要有某种一致性。例如，卡通式游戏的彩色按钮可以又大又亮，而第一人称视角射击游戏的按钮则可能又暗又恐怖。

17.2 取得自由版权游戏媒体

商店和 Internet 上的光盘中有大量无版权的图形和声音。无版税就是使用图形时不必支付版税（但一定要认真阅读作品所带的许可证）。

许多纹理更适合桌面出版，使用简单的计算机线条画。但也有一些图库是针对图形设计人员的，如纹理集和照片集。通常可以从 Internet 上搜索符合要求的无版权图库。

可以用照片开始生成背景图像与纹理。以图 17-1 为例，这个照片上方可以用于滚动背景（大矩形，其他部分可以用于趣味纹理（小矩形））。

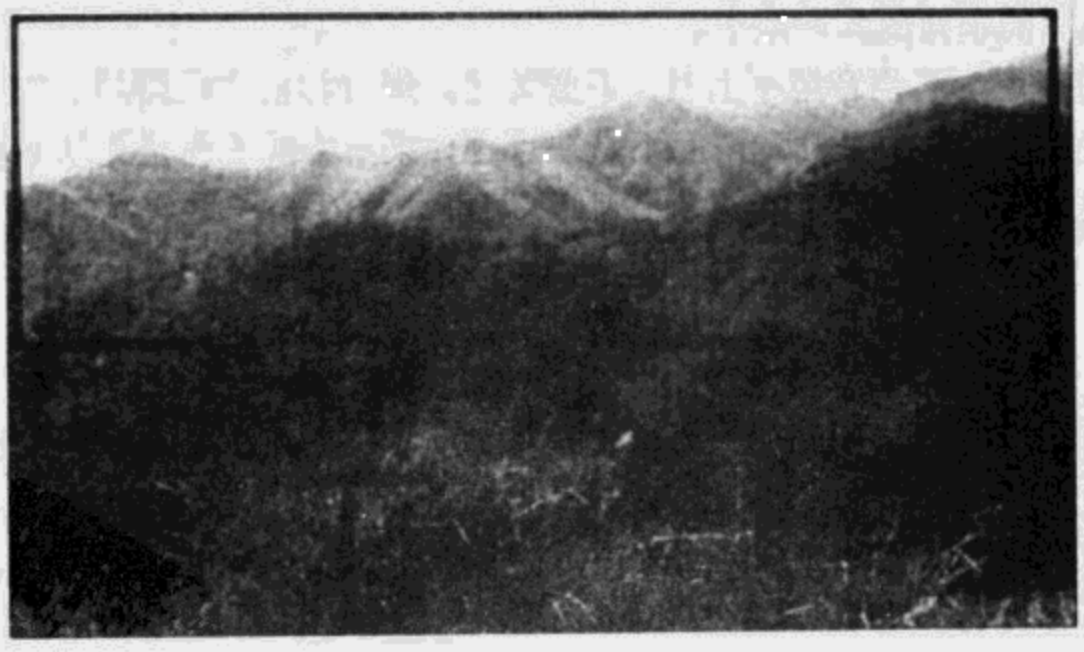


图 17-1 照片上方可以用于滚动背景（大矩形，其他部分可以用于趣味纹理（小矩形））

要找的其他无版权媒体还有图标和字体。图标是很有用的用户界面元素，见第 4 章“声效与音乐”。这些图标来自绘图程序（Paint Shop Pro），稍作修改，增加了下拉阴影。

17.3　与艺术家和声效工程师合作

如果有幸能让别人生成游戏图形和声效，则还要与艺术家很好地配合。下面是要记住的几点：

- 保证生成的媒体不仅有用，而且让艺术家觉得有趣。让他们灵活地按自己的风格所作，而不要控制每个细节。
- 明确游戏需要。例如，我们可能需要三维模型（不同的妖怪、充电器，等等）和不同类型的声音（跳动、发射武器，等等）。
- 同样，游戏媒体的技术要求也要明确。例如，指定纹理尺寸（是否 2 的指数），纹理是否要保证无缝、文件格式、颜色数、多边形数限制，等等。
- 建立一个对双方都合适的时间表。了解建立媒体所需时间。
- 如果某些游戏图形和声效在游戏中不能用或感觉不对，一定要提出来。双方都要确定是采用特定媒体还是重做。

总之，要保持沟通渠道畅通，合作过程有趣。

17.4　工具

工欲善其事，必先利其具。首先要牢牢掌握工具的用法，不怕尝试。如果怕破坏某个游戏图形和声效，可以先建立主备份。

声音工具可以录制、编辑和增加声波效果。一些廉价的声音工具包括 Pro Tools FREE、Cool Edit、GoldWave、GLAME、GSMP 与 LaoE for Java。

图形工具可以生成和编辑纹理与幽灵。一些廉价的图形工具包括 Photoshop Elements(www.adobe.com)，Paint Shop Pro(www.jasc.com)与 GIMP(www.gimp.org)。

三维工具可以生成和绘制三维图形。一些免费三维工具包括 Blender 与 POW-Ray。Blender 是个多边形建模器，而 POW-Ray 是光跟踪绘制器。尽管我们用不上光跟踪，但 POW-Ray 有一些有趣的纹理生成功能，可以加以利用。如果经费够用，则还有一些专业三维工具，如 3ds max 与 Maya。

17.5　生成声音

大多数声效都是以自然录音为基础，然后进行编辑，增加效果。例如，在最初的星球大战电影中，一个星际火箭的引擎声音是狮子的吼声。

可以从生活中的声音开始，如汽车启动声、关门声、冷笑声和喷头喷水的声音。

录音时，音量不要太高，否则声音会剪取和扭曲。最好录制中等音量，不太高也别太低。

另外，用声音工具的消噪过滤功能，从而消除环境噪声，建立便应的低保真麦克风。

得到录制的声音后，采用过滤和效果，使声音更有趣，或把两个或几个有机声音结合起来。可能要避免“伪”音，如明显由人模仿的开枪声音。

记住，在许多声音工具中，过滤是破坏性的，即修改声频样本本身。保留原声的拷贝，以便采用大多过滤之后恢复原者。

当然，声效不一定非要来自录制的有机声音，也可以用许多声音工具产生的声波生成合成音。

最后，声音一定要以 0 帧度开始和结束，避免太突然，如图 17-2 所示。但开始时不要增加太多延迟，否则要过半天才能听到声音。

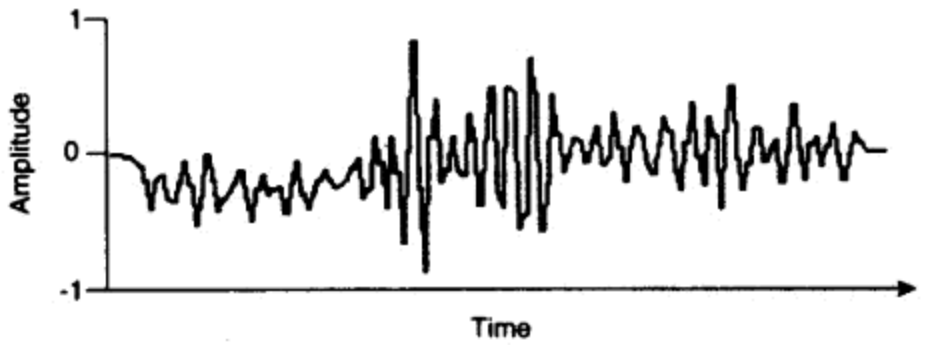


图 17-2　声音一定要以 0 帧度开始和结束

本书生成声音的几个例子如下：

- 第 4 章的声效是我打开邮箱时录制的。
- 第 4 章的“苍蝇嗡嗡声”是我模仿苍蝇声，把音调作了些调整。
- 第 5 章“创建二维平台游戏”中的拳击声是完全用声波和效果用软件生成的。

另一个生成声效的例子是我生成的第一个 Java 游戏 Scared，是个三维第一人称视角射击游戏。这是个标准第一人称视角射击游戏，游戏者在迷宫中跑、打开、找钥匙和进入下一关。这个游戏中的几个例子如下：

- 关门声是录制我的旧门，增加了一些贝斯。
- 激光枪声是我模仿的激光枪声，但调整了音调等参数。
- 游戏者呼噜声和死亡声是录制我的呼噜声和（戏剧性地）死亡声。
- 发电机的声音录制的是我的窗式旧空调，增加了一些贝斯。
- 开门声录制的是打开 20 盎司的 Dr Pepper 瓶子时发出的声音。
- 游戏者捡钥匙时的声音录制的是打开 12 盎司的 Dr Pepper 瓶子时发出的声音。
- 敌人破坏时的声音录制的是压扁 12 盎司的 Dr Pepper 瓶子时发出的声音。

可以看到，这个游戏中大多数声音都借用了 Dr Pepper 的灵感。

声音文件格式

第 5 章介绍过声音文件格式，这里作一个简单复习。

简单地说，Java Sound API 可以读取三种格式：AIFF、AU 与 WAV.Additionally。此外，可以安装第三方的其他格式阅读器，如 OGG 译码器。本书使用 16 位单色 44100Hz WAV 文件，是光盘质量的，但是单色声而不是立体声。

17.6　生成纹理与幽灵

生成纹理与幽灵时，最好在定义像素与图像尺寸时使用实际尺寸。例如，在三维游戏中，可以假设 100 图素为 1 米，或 1 个图素为 1 厘米。这样就可以确定电灯开关和门把手的大小，使物体大小与现实中相符。对于幽灵，有时可以使用基本图形元，如椭圆、直线

和其他简单形状，而不必手工绘制。注意第 5 章平台游戏中使用的幽灵都是简单形状构成的，如图形、星形和心形。大多数图形程序很容易生成这类形体。

如果你更习惯纸上画画，可以扫描图纸，然后用图形程序整理线形和上色。

生成纹理时，要考虑如何用纹理在多边形上平铺。例如，如果纹理上有明显的污染，则平铺纹理时，会出现一点点污染纹理，因此应让纹理尽量微妙。

生成纹理的思路元一是以实体颜色纹理开始，然后不断修改。增加杂波和微妙纹理效果，使其更逼真。许多图形尽管能够使图像显示像沥青、金属或木质，等等。

另外，不好的画缩成更小尺寸时有时会显得更好看。如果游戏需要 64×64 纹理，可以先建立 128×128 或 256×256 纹理，然后缩小。

别怕测验！用 Gamma 值、亮度、对比度等参数调整颜色。用下拉阴影之类的阴影效果使纹理更有深度。但要记住，这是纹理，对于三维引擎，不能在纹理上画太多深度，否则不直视多边形时会很不像样子。

17.6.1　图形文件格式

第 7 章“三维图形”介绍过图形文件格式，这里作的简单回顾。

对照片作品，最好选择 APEG 格式，因为它提供高压缩率和高质量，但它是个损失性格式。

但纹理和幽灵最好保存为 PNG 格式，这是无损失格式。

PNG 格式要说明一点：如果 PNG 图像要使用位掩码透明度（而不是 Alpha 透明度），则要将 PNG 格式变成 8 位 256 色，使用索引透明度。如果保存时超过 8 位颜色，则它会变成半透明图像，无法用硬件加速（在 J2SDK 1.4.1 中）。

有时要减少图形文件的长度，例如减少游戏媒体的总体长度。减少 PNG 图形文件的长度，最好的办法是减少图像中的颜色数。还要保证图像不会抖动，否则在大多数游戏情形中都会很难看。

17.6.2　生成无缝纹理

对于平铺纹理，在多边形上平铺纹理时，要保证纹理边不明显，使平铺纹理显示为无缝纹理。图 17-3 所示的纹理看起来不错，但平铺得不好。好在许多图形程序内置了这个功能。图 17-3 就看不出接缝在哪里了。

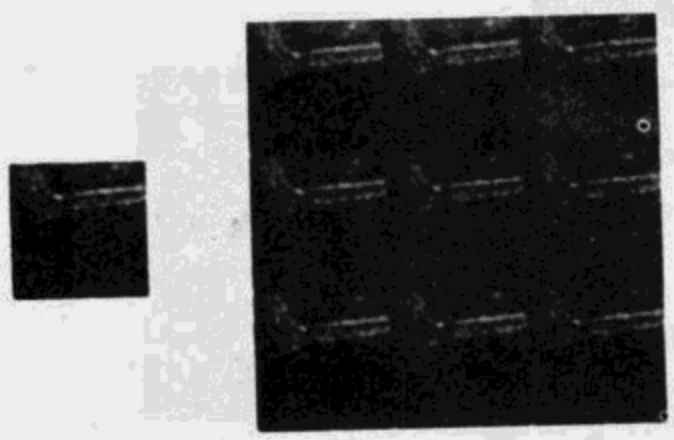


图 17-3　纹理看起来不错，但平铺得不好

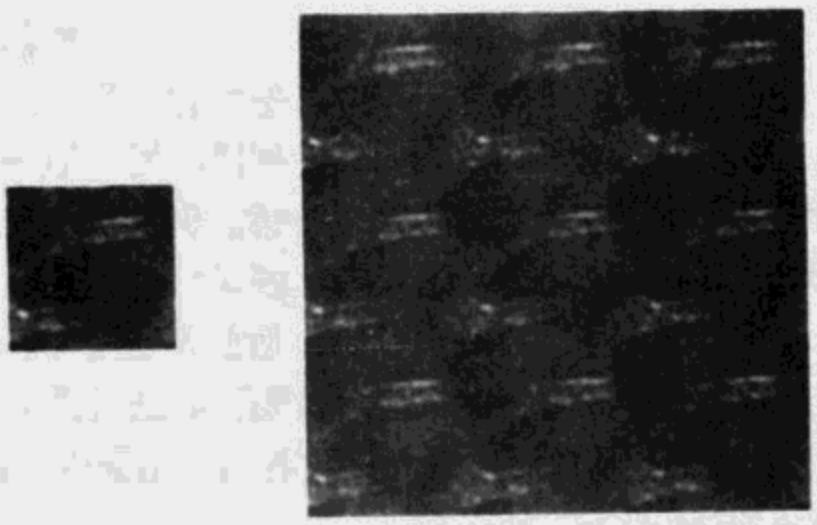


图 17-4 许多图形程序内置无缝平铺纹理功能

生成无缝纹理的一个方法是使用重复纹理，如方形栅格（或六角形栅格）。图 17-5 显示了一些无缝纹理的例子。

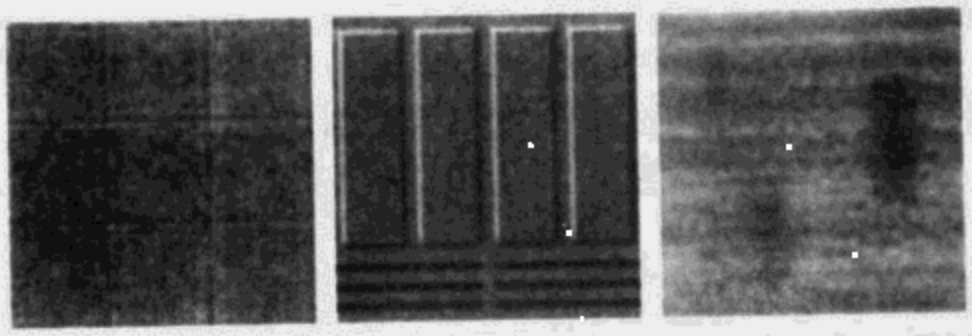


图 17-5 一些无缝纹理的例子

图中第一个纹理是 3×3 栅格，纹理的缝与栅格线没有明显差别。

第二个纹理是实体灰色，但还有背的纹理、一些黑线和中央的蚀斑。背的纹理在纹理中重复四次，黑线断开一次纹理水平平铺时很难看到哪里是接缝，而垂直平铺时则很明显。

最后，第三个纹理是整体无缝地砖，像木纹，用 Paint Shop Pro 制成无缝纹理，可以在各边平铺，不会有太多接缝，但纹理很清楚。

17.6.3 生成替换纹理

有时平铺纹理的样子太单调。三维引擎中的灯光可以缓解部分问题，但另一种方法是使用与平铺纹理稍有不同的替换纹理。

例如，图 17-6 是个替换纹理。

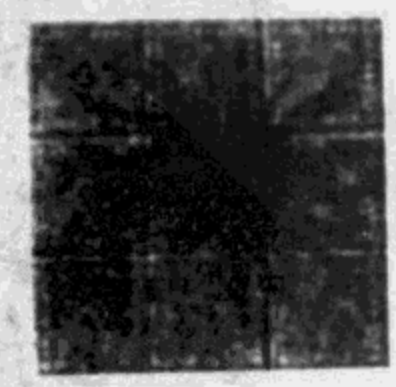


图 17-6 替换地砖的例子

使用这个纹理可以打破平铺纹理的单调，如图 17-7 所示。

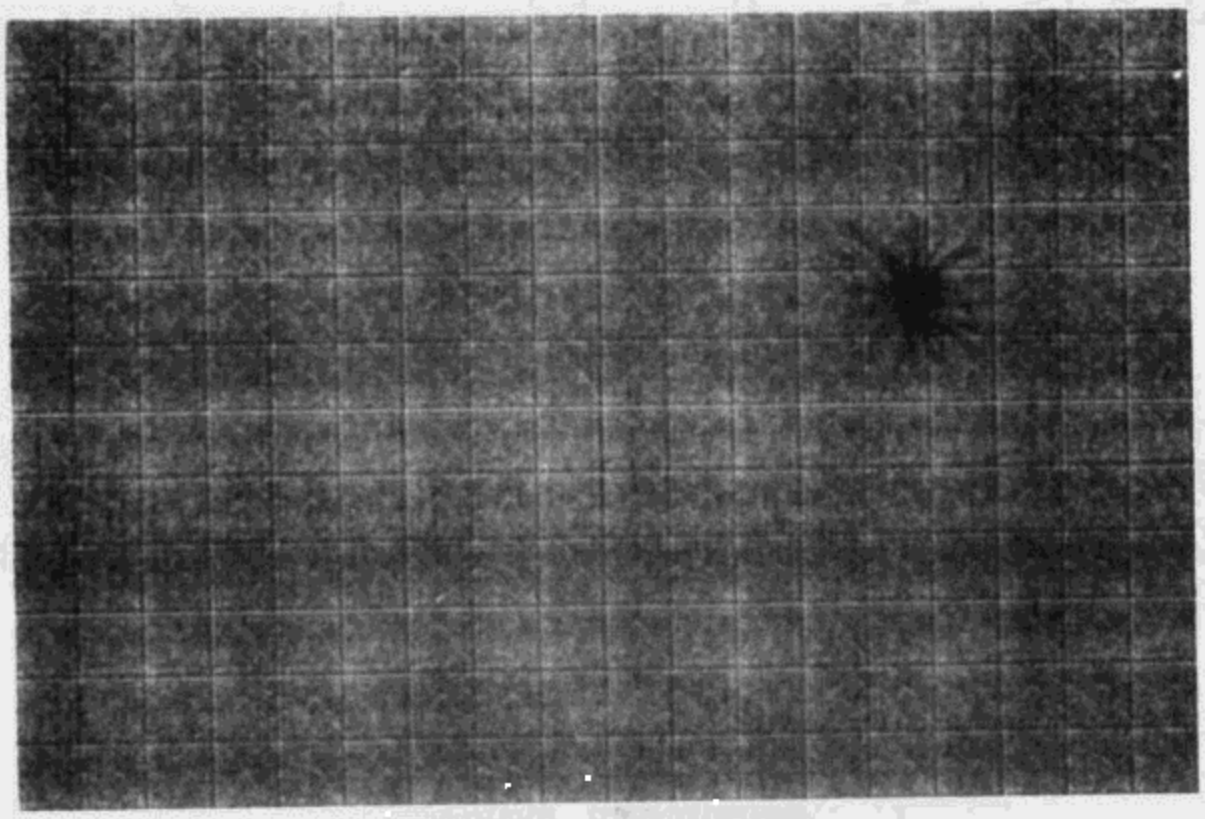


图 17-7　替换纹理在其他纹理中的样子

图 17-8 显示更多替换纹理，用带伪灯光的纹理和带有趣物体的纹理。

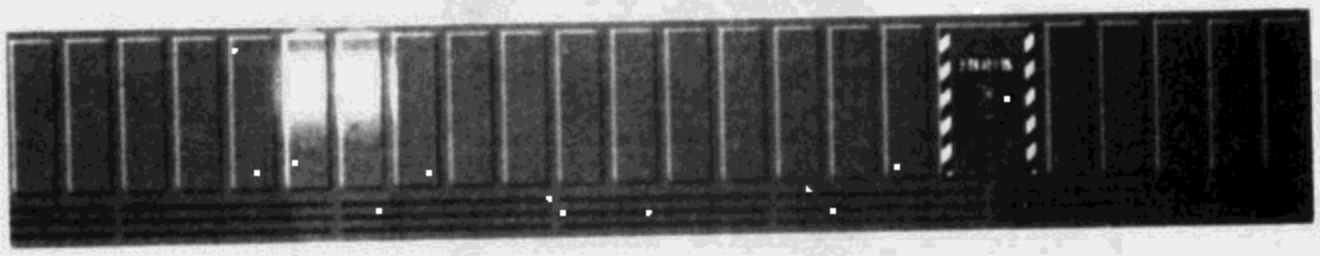


图 17-8　另一个使用替换地砖的例子

17.6.4　生成过渡纹理

在自上而下游戏中，需要用各种不同纹理定义地形，如小纹理和草纹理。但是，不同纹理靠在一起时样子很难看，如小纹理和草纹理靠近时（图 17-9）。

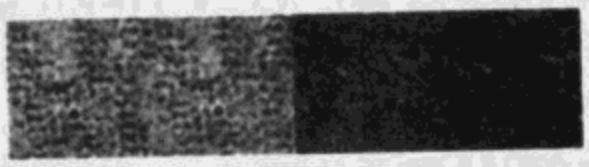


图 17-9　小纹理和草纹理靠近时样子很难看

要解决这个问题，可以用过渡纹理，在水与草之间形成更好看的边界。图 17-10 插入过渡纹理，表示水与草之间的堤岸。

图 17-10　用过渡纹理形成更好看的边界

当然，图 17-10 的方案只解决了水地砖左边是草地砖的情形。在游戏中，会有许多不

同情形，如水地砖右边是草地砖或草的两边都是水。

首先要生成三个不同过渡，如图 17-11 所示。这样就可以让水在过渡纹理的一边、两边或角上。还可以生成更多过滤，使水可以在池或四边，但这个方案可以满足许多情形。

图 17-11　生成三个不同过渡

图 17-11 的过渡纹理上面画了一个非对称的连城游戏，显示过渡纹理不同部分的边。纹理外边要对每个纹理相同：或者完全是水，或者是草，或者从水到草过渡。

好在有这三种过渡就够了。运行时可以将每个过渡纹理转三个角度（90^0，180^0，270^0），得到图 17-12 所示的过渡纹理。

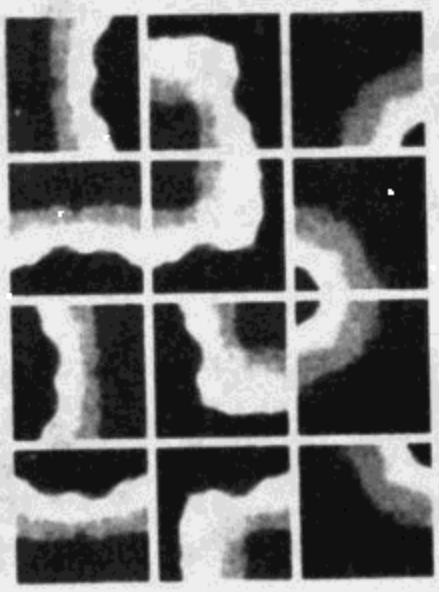


图 17-12　三种过渡纹理转动得到几种过渡纹理

利用几种过渡纹理，可以生成各种漂亮的地形，如岛屿、湖泊、河流，不同纹理靠在一起时样子不再会很难看。

当然，这只是过渡纹理的一点皮毛，还可以生成三种不同纹理相遇时的过渡纹理。有些游戏在同一过滤中使用两种或三种纹理，使过渡花样更丰富。

17.6.5　生成多层纹理

另一种技巧是采用多层纹理，用背景纹理和可以透视的前景纹理。以图 17-13 为例，这里有两层：地形层和道路层。每个纹理由相同尺寸的纹理构成，而层合并成图 17-14。

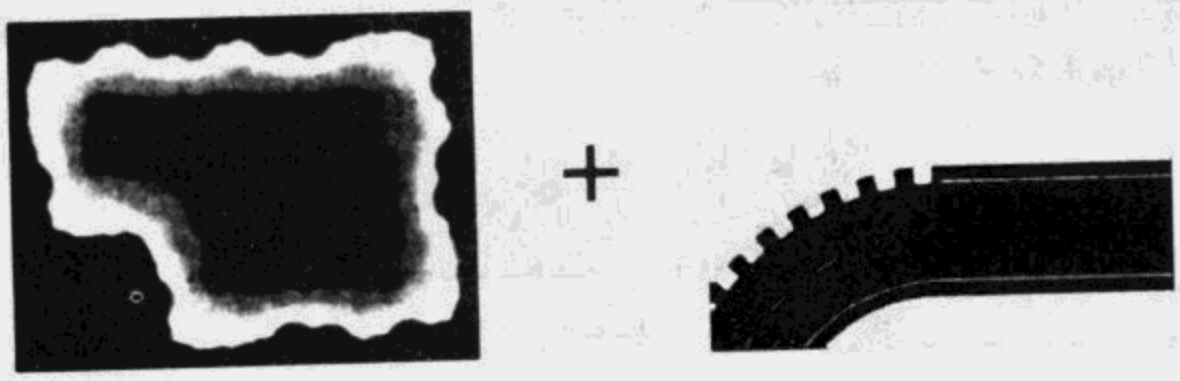


图 17-13　两层：地形层和道路层

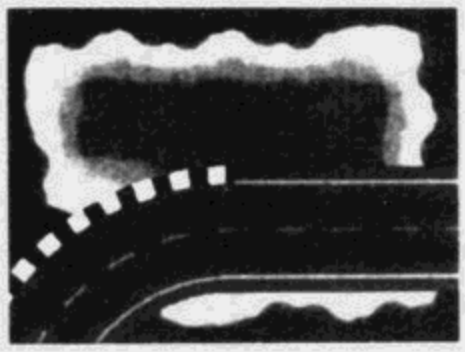


图 17-14　游戏中把道路层放在地形层上面

这样，就可以让道路放在不同地形上，无论是草地、沙滩、海岸，还是别的地形。当然，不仅可以两层，还可以多层，例如路径层，在草地上显示小路。

17.7　生成漂亮屏幕与 HUD 图形

装饰屏幕通常很简单。开始屏幕中只要有精彩徽标的图形。得到程序中有大量工具可以生成基于文本的徽标。

另外，游戏的不同状态也可能需要装饰屏幕，如装入关卡时，完成一关后和游戏结束时。这也很简单，也许有只要徽标即可。

飞行仪表盘（HUD）也需要一些图形，可能要一些图标来表示生命力与弹散，我要有许多按键的整个界面，如战略与模拟游戏中。

17.8　生成 UI 图形

最后还有用户界面图形。通常，我们只需要图像按钮，使游戏具有特定外观。

设计用户界面图形时，记住不同颜色表示的含义。例如，交通灯红色为停，绿色为走，黄色为小心，选择能表达所要意思的灯光。

另外，使用简单而描述性的用户界面图形，使用户迅速判断要单击什么。在第 3 章“交互性与用户界面”的例子中，按钮使用了图标。

定制 Swing

如果用户界面的任何部分使用 Swing，如用户名输入文本框，则可能要根据需要定制 Swing 外观。最灵活的方法是生成自己的外观，但这个过程比较复杂。

另一种方法是根据用户界面颜色改变缺省 Java 外观的颜色。改变 Swing 组件的颜色很简单，只要在用户界面缺省表中存放一个值如下：

```
ColorUIResource myColor = new ColorUIResource(oxf9f9f9);
UIManager.put(“Button.bacdground”, myColor);
```

ColorUIResource 对象是 Color 对象的子类，实现 UIResource 界面，让 Swing 内部使用。这里将所有按钮的背景颜色设置为我们自己的颜色。

Swing 中有几百种不同颜色属性，包括 Button.shadow，Button. highlight 等等。

如果要了解这些属性的意义，可以用清单 17.1 的代码显示这些属性及其缺省值。

清单 17.1 SwingDefaults.java

```
import java.awt.Color;
import java.util.*;
import javax.swing.UIManager;
import javax.swing.UIDefaults;

/**
    打印缺省外观的所有 Swing 缺省清单
*/
public class SwingDefaults {

    public static void main(String[] args) throws Exception {

        List allValues = new ArrayList();

        // 将所有值存放在表中
        UIDefaults defaults = UIManager.getDefaults();
        Enumeration e = defaults.keys();
        while (e.hasMoreElements()) {
            Object key = e.nextElement();
            Object value = defaults.get(key);
            if (value instanceof Color) {
                allValues.add(key + " = " + value);
            }
        }

        // 排序和打印数值
        Collections.sort(allValues);
        for (int i=0; i<allValues.size(); i++) {
            System.out.println(allValues.get(i));
        }
    }
}
```

注意：这个代码显示颜色属性，也可以修改边框等属性。

17.9 生成字体

Java2D 和字体可以进行许多处理，但生成更有趣的新本设计是比较困难的。例如，绘制外形线和渐变填充文本在 Java2D 中就不太容易。

许多游戏使用自己的基于图像字体。基于图像字体中每个字符都是图像，这样，艺术家可以更灵活地设计字体外观，用基于图像字体绘制文本也更快，因为我们可以利用硬件加速。

本节要实现从基于图像字体绘制文本的类。这个版本的一个目的是支持变长度字符，如图 17-15 所示。图中显示 m 比 i 宽。

图 17-15　支持变长度字符，图中显示 m 比 i 宽

要实现支持变长度字符的基于图像字体，一个简单方法是每个字符用一个图像，见清单 17.2 的 ImageFont 类。

清单 17.2　ImageFont.java

```
package com.brackeen.javagamebook.graphics;

import java.awt.*;
import java.awt.image.BufferedImage;
import java.io.*;
import javax.swing.ImageIcon;

/**
    ImageFont 类可以用字符图像装入和绘制文本
    读取目录中所有"charXX.png"形式的 png 图像，其中"XX"是二进制 unicode 值
    字符可以有不同宽度与高度
*/
public class ImageFont {

    public static final int HCENTER = 1;
    public static final int VCENTER = 2;
    public static final int LEFT = 4;
    public static final int RIGHT = 8;
    public static final int TOP = 16;
    public static final int BOTTOM = 32;

    private char firstChar;
    private Image[] characters;
    private Image invalidCharacter;
    /**
        生成没有字符的新 ImageFont
```

```
*/
public ImageFont() {
    this(null);
    firstChar = 0;
    characters = new Image[0];
}

/**
    生成新 ImageFont 并从指定路径装入字符图像
*/
public ImageFont(String path) {
    if (path != null) {
        load(path);
    }

    // 建立无效字符使用的字符
    invalidCharacter =
        new BufferedImage(10, 10, BufferedImage.TYPE_INT_RGB);
    Graphics g = invalidCharacter.getGraphics();
    g.setColor(Color.RED);
    g.fillRect(0,0,10,10);
    g.dispose();
}

/**
    从指定路径装入每个字符的字符图像。例如，如果路径为"../fonts/large"，则这个方法搜索
    该路径中所有名称为"charXX.png" 的文件，其中"XX"是二进制 unicode 值。并非每个字
    符图像都要存在，例如可以只要数字和大写字母
*/
public void load(String path) throws NumberFormatException {
    // in this directory:
    // 在这个目录中装入以 'char'开头的每个文件
    File dir = new File(path);
    File[] files = dir.listFiles();

    // 寻找最小与最大字符
    char minChar = Character.MAX_VALUE;
    char maxChar = Character.MIN_VALUE;
    for (int i=0; i<files.length; i++) {
```

```
            int unicodeValue = getUnicodeValue(files[i]);
            if (unicodeValue != -1) {
                minChar = (char)Math.min(minChar, unicodeValue);
                maxChar = (char)Math.max(maxChar, unicodeValue);
            }
        }

        // 装入图像
        if (minChar < maxChar) {
            firstChar = minChar;
            characters = new Image[maxChar - minChar + 1];
            for (int i=0; i<files.length; i++) {
                int unicodeValue = getUnicodeValue(files[i]);
                if (unicodeValue != -1) {
                    int index = unicodeValue - firstChar;
                    characters[index] = new ImageIcon(
                        files[i].getAbsolutePath()).getImage();
                }
            }
        }
    }

    private int getUnicodeValue(File file)
        throws NumberFormatException
    {
        String name = file.getName().toLowerCase();
        if (name.startsWith("char") && name.endsWith(".png")) {
            String unicodeString =
                name.substring(4, name.length() - 4);
            return Integer.parseInt(unicodeString);
        }
        return -1;
    }

    /**
        取得指定字符的图像。如果没有指定字符的图像，则返回特殊的无效字符图像
    */
    public Image getImage(char ch) {
        int index = ch - firstChar;
```

```
        if (index < 0 || index >= characters.length ||
            characters[index] == null)
        {
            return invalidCharacter;
        }
        else {
            return characters[index];
        }
    }

    /**
        取得指定字符串的字符串宽度（像素）
    */
    public int stringWidth(String s) {
        int width = 0;
        for (int i=0; i<s.length(); i++) {
            width += charWidth(s.charAt(i));
        }
        return width;
    }

    /**
        取得指定字符的字符宽度（像素）
    */
    public int charWidth(char ch) {
        return getImage(ch).getWidth(null);
    }

    /**
        取得指定字符的字符高度（像素）
    */
    public int charHeight(char ch) {
        return getImage(ch).getHeight(null);
    }

    /**
        在 (x, y)位置绘制指定字符串
    */
    public void drawString(Graphics g, String s, int x, int y) {
```

```
        drawString(g, s, x, y, LEFT | BOTTOM);
    }

    /**
        在(x, y)位置绘制指定字符串
    */
    public void drawString(Graphics g, String s, int x, int y,
        int anchor)
    {
        if ((anchor & HCENTER) != 0) {
            x-=stringWidth(s) / 2;
        }
        else if ((anchor & RIGHT) != 0) {
            x-=stringWidth(s);
        }
        // 清除字符绘制的水平标志
        anchor &= ~HCENTER;
        anchor &= ~RIGHT;

        // 绘制字符
        for (int i=0; i<s.length(); i++) {
            drawChar(g, s.charAt(i), x, y, anchor);
            x+=charWidth(s.charAt(i));
        }
    }

    /**
        在(x, y)位置绘制指定字符串
    */
    public void drawChar(Graphics g, char ch, int x, int y) {
        drawChar(g, ch, x, y, LEFT | BOTTOM);
    }

    /**
        在(x, y)位置绘制指定字符串
    */
    public void drawChar(Graphics g, char ch, int x, int y,
        int anchor)
    {
```

```
            if ((anchor & HCENTER) != 0) {
                x-=charWidth(ch) / 2;
            }
            else if ((anchor & RIGHT) != 0) {
                x-=charWidth(ch);
            }

            if ((anchor & VCENTER) != 0) {
                y-=charHeight(ch) / 2;
            }
            else if ((anchor & BOTTOM) != 0) {
                y-=charHeight(ch);
            }
            g.drawImage(getImage(ch), x, y, null);
        }
    }
```

在 ImageFont 构造函数中，可以指定字符图像所在的路径如下：

```
ImageFont medFont = new ImageFont("../fonts/medium");
```

这个路径是每个字符的图像所在目录。代码能够聪明地搜索每个字符的目录位置，而不是装入每个 Unicode 值。这样就可以建立只有数字字符的字体。

字符文件名形式为 charXX.png，其中 XX 是字符的 Unicode 代码。例如，char65.png 是字母 A 的图像。

这个类的文本绘制方法很灵活，可以处理对齐问题。例如，可以用标志自动居中或右齐显示文本字符串。下面的例子使用左上角固定，使文本在屏幕上方显示：

```
medFont.drawstring(g, "Hi" ,0,0, ImageFont.LEFT| ImageFont.TOP);
```

当然，使用图像字体的缺点是不能很好地伸缩能不同尺寸。如果要找多字号字体，则要按照普通 Java2D 字体绘制或对图像字体生成多个字号。

这样就完成了图像字体。本书源代码中有一个使用图像字符的演示程序，演示程序显示左齐、居中、右齐文本和一些简单文本动画，像第 2 章屏幕上移动幽灵时一样。

17.10 小结

这样就完成了编程人员图形。即使你对生成游戏图形不感兴趣，本章也介绍了如何建立精彩的可复用 ImageFont 类，可以用图像代替文本。我们还介绍了如何建立声音和一些制作纹理的提示与技巧。希望本章能帮你生成一些有用材料，当然也难免生成一些无用材料。

第 18 章　游戏设计与最后 10%

关键主题

- 最后 10%
- 游戏设计要素
- 生成贴图编辑器
- 调试
- 保护代码
- 游戏部署
- Java Web Start 游戏部署
- 自然编译游戏部署
- 更新与补丁
- 带宽问题
- 反馈与 Beta 测试
- 赚钱
- 综合起来
- 小结

建立游戏后，怎么办？

建立游戏后，一定有趣和刺激，有了漂亮的图形和声音，然后怎么办？

下一步的工作取决于要达到什么目的。也许要和几位朋友分享，或作为演示，帮你找到工作，或要用光盘发布或在 Web 上发布，希望赚到钞票。

如果准备发布，则要尽量改善游戏。这是游戏的最后 10%，有时游戏的最后 10%要投入 50%的开发时间。

本章介绍游戏中要增加什么进行修饰，还要介绍一些实现技术，如生成游戏状态机和用 Java Web Start 发表游戏。

18.1　最后 10%

游戏和演示要花不少时间来修饰，使它显得更专业，包括增加更多特殊效果，装入屏幕、菜单和开始序列。

一个最重要的工作是让游戏尽量壮实，尽量表现好些。游戏一定要结实，不要老出漏子。尽量消除所有缺陷。如果用户将应用程序最小化或切换到另一应用程序，则要让游戏暂停。另外，用户单击 Exit 按钮时，游戏应实际退出，而不只是隐藏游戏窗口，继续占用处理器和网络连接应使用户计算机处于游戏启动时的状态。

18.1.1 效果

一定要增加一些声效。声效是游戏中必备的元素之一，缺少时才更引人注意。下面介绍何时增加冒险之类游戏所要的声效：

- 游戏者受伤或死亡；
- 游戏者跳动和摔下；
- Bot 碰上或破坏；
- 拉杆和开门；
- 捡到东西时；
- 打开后台发电机或另一机器嗡嗡作响时；
- 背景环境音，如小溪哗哗声、火焰声、风吹进树林的声音、鸟鸣声、海浪声或雨滴声；
- 用户界面声音，如用户单击按钮、转入贴图视图、选择不同武器或用户界面等待输入时；
- “提示”音，如坏家伙要发射武器时和游戏者接近隐藏的洞穴时；
- 负反馈音，如游戏者选择不存在的武器时；

当然，还可以在游戏中增加更多声音。

除了声效之外，视觉效果也有助于改善游戏，使其更精彩。可以增加的视觉效果包括：

- 微粒爆炸；
- 火焰；
- 热型变形效果；
- 灯光照明；
- 急闪灯；
- 环境效果，如下雨、下雪和云在飘；
- 实体动画，如生物在跑，葡萄在生长。

但是，也不能增加太多特殊效果。效果应精彩有趣，但不能影响玩游戏。例如，不能一下子太多声音，无法分辨。还要避免盖住整个屏幕的图形效果，不要读导和搞乱用户（除非确实要达到这样的效果）。

18.1.2 游戏状态机

在本书大多数演示程序中，游戏直接从“游戏”方式开始，一直保持到退出为止，没有装入屏幕、开始序列、菜单和帮助屏幕，但成熟的游戏一定要有这些项目。

如果要增加这些不同屏幕，则可以把整个游戏看成一个状态机，如图 18-1。

在这个简单状态机例子中，游戏从装入屏幕开始，转入开始序列（可能是装饰屏幕、动画或游戏演示）。通常，我们要让用户能够取消这些开始序列，以免每次启动游戏时都坐在那里傻等。

下一个状态是主菜单，然后用户可以转入许多其他游戏状态，如设置和帮助屏幕。

这只是个简单设计。你的游戏中还可以增加更多细节。例如，这里显示的主游戏状态可以分解为几个不同状态：游戏状态可以包括关卡装入序列、关卡开始序列到游戏，然后

到关卡结束序列。

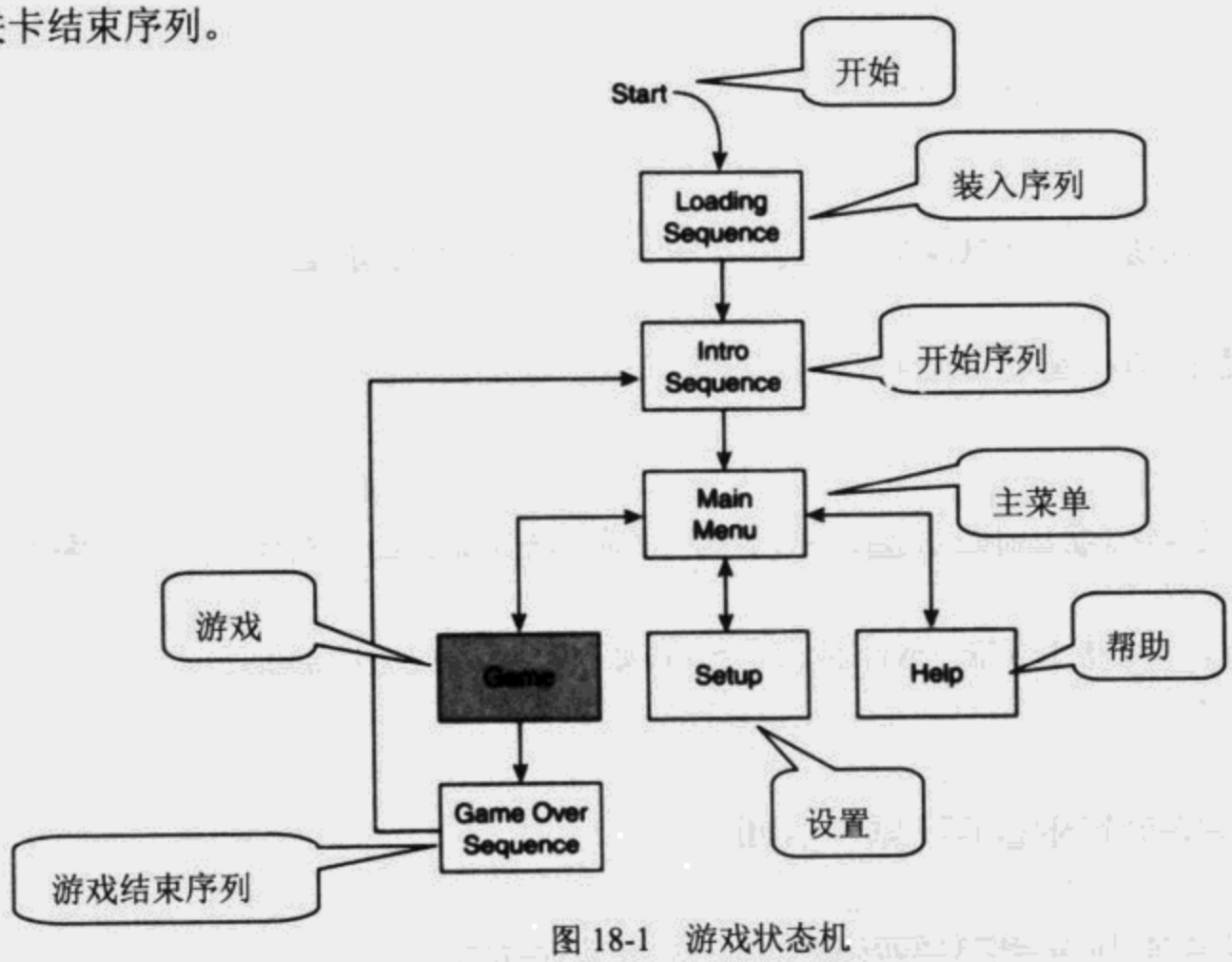


图 18-1　游戏状态机

实现游戏状态机很简单，类似于本书的其他状态机。例如，处理游戏输入时，可以用下列代码：

```
if (state == STATE_INTRO) {
    // 处理游戏输入
}
else if (state == STATE_GAME) {
    // 处理游戏输入
}
else {
    ...
}
```

此外，更新与绘制每一帧时，还需要这些 if/then 语句。

可以想像，这个代码随着更多状态加进游戏中很容易变得麻烦和费解。

也可以把每个状态作为一个对象，处理输入、更新、绘制等操作，避免很长的 if/then 代码块。首先建立清单 18.1 所示的 GameState 接口。

清单 18.1　GameState.java

```
public interface GameState {

    /**
        取得状态名，用于 checkForStateChange()方法
```

```
    */
    public String getName();

    /**
        返回这个状态可以转为另一状态时要改变的状态名，否则返回 null
    */
    public String checkForStateChange();

    /**
        装入这个状态的任何资源，这个方法在设置任何 GameStates 之前在后台线程中调用
    */
    public void loadResources(ResourceManager resourceManager);

    /**
        初始化这个状态和设置输入管理器
    */
    public void start(InputManager inputManager);

    /**
        进行停止这个状态所需的任何操作
    */
    public void stop();

    /**
        更新世界，处理输入
    */
    public void update(long elapsedTime);

    /**
        绘制到屏幕
    */
    public void draw(Graphics2D g);
}
```

这个接口很简单。start()方法贴图这个游戏状态的任何键和鼠标关联。stop()方法在状态完成时进行任何清理工作，如暂停声音或释放资源。update()与 draw()方法更新与绘制每一帧。checkForStateChange()方法返回这个状态可以转为另一状态时要改变的状态名，否则返回 null。例如，这个方法可以在完成开始序列后返回字符串 Main。

注意：loadResources()方法在主线程外的另一线程中调用，使资源装入独立于更新/绘制过程。这样，装入屏幕很容易在后台装入所有资源时显示。最好要在资源

装入遇到错误时让这个方法抛出异常，使游戏引擎能相应响应。GameState 的整个寿命周期如图 18-2 所示。

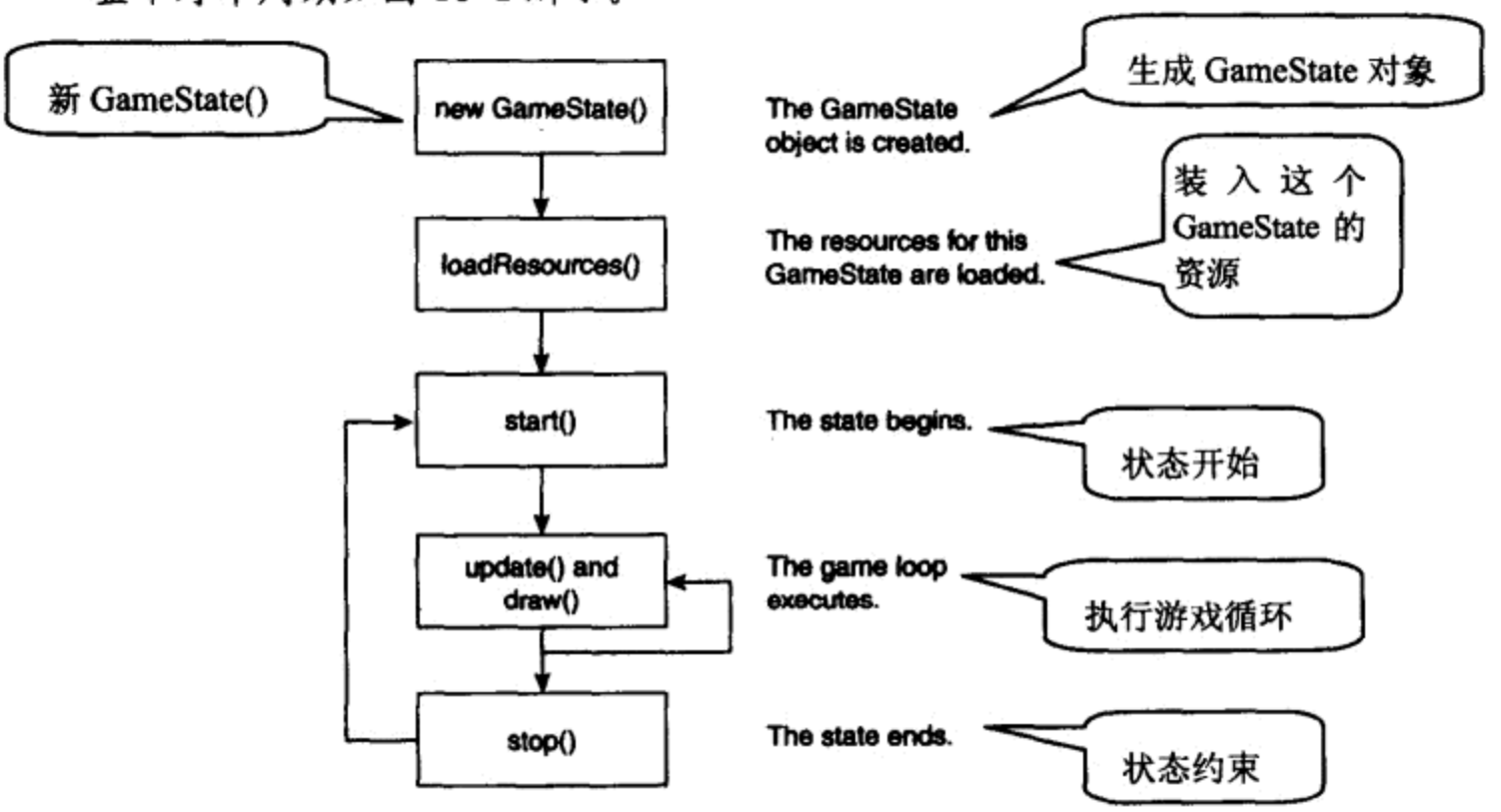


图 18-2　GameState 的整个寿命周期

作为例子，清单 18.2 的 GameStateManager 类管理和改变状态并跟踪当前状态。

清单 18.2　GameStateManager.java 的相关方法

```
    /**
        设置当前状态（名称）
    */
    public void setState(String name) {
        // 清理旧状态
        if (currentState != null) {
            currentState.stop();
        }
        inputManager.clearAllMaps();

        if (name == EXIT_GAME) {
            done = true;
        }
        else {

            // 设置新状态
            currentState = (GameState)gameStates.get(name);
            if (currentState != null) {
```

```
                currentState.start(inputManager);
            }
        }
    }
    /**
        更新世界，处理输入
    */
    public void update(long elapsedTime) {
        // 如果无状态，暂停一会儿

        if (currentState == null) {
            try {
                Thread.sleep(100);
            }
            catch (InterruptedException ex) { }
        }
        else {
            String nextState = currentState.checkForStateChange();
            if (nextState != null) {
                setState(nextState);
            }
            else {
                currentState.update(elapsedTime);
            }
        }
    }
    /**
        绘制到屏幕上
    */
    public void draw(Graphics2D g) {
        if (currentState != null) {
            currentState.draw(g);
        }
        else {
            // 如果无状态，将缺省图像，绘制到屏幕上
            g.drawImage(defaultImage, 0, 0, null);
        }
    }
```

在这个代码中，draw()将工作委托给当前状态，如果状态为 null，则绘制缺省图像。缺省图像可以是装饰图像，表示正在装入游戏。

除了更新状态之外，update()方法还检查当前状态是否指示要转到另一状态。最后，setState()方法实际切换状态。

注意：*转到另一状态时，所有键和鼠标贴图从输入管理器中清除。*

例如，清单 18.3 生成一个简单的装饰屏幕游戏状态。GameState 只是把装饰图像放在屏幕上，保留三秒钟或等待用户按空格条（看哪个在先）。

清单 18.3　SplashGameState.java

```
public class SplashGameState implements GameState {

    private String splashFilename;
    private Image splash;
    private GameAction exitSplash;
    private long totalElapsedTime;
    private boolean done;

    public SplashGameState(String splashFilename) {
        exitSplash = new GameAction("exitSplash",
             GameAction.DETECT_INITAL_PRESS_ONLY);
        this.splashFilename = splashFilename;
    }

    public String getName() {
        return "Splash";
    }

    public void loadResources(ResourceManager resourceManager) {
        splash = resourceManager.loadImage(splashFilename);
    }

    public String checkForStateChange() {
        return done?"Main":null;
    }

    public void start(InputManager inputManager) {
        inputManager.mapToKey(exitSplash, KeyEvent.VK_SPACE);
        inputManager.mapToMouse(exitSplash,
```

```
            InputManager.MOUSE_BUTTON_1);

        totalElapsedTime = 0;
        done = false;
    }

    public void stop() {
        // 什么也不干
    }

    public void update(long elapsedTime) {
        totalElapsedTime+=elapsedTime;
        if (totalElapsedTime > 3000 || exitSplash.isPressed()) {
            done = true;
        }
    }

    public void draw(Graphics2D g) {
        g.drawImage(splash, 0, 0, null);
    }
}
```

这个游戏状态开始时，将空格条和主鼠标按钮贴图到 exitSplash 游戏机。完成之后，状态显示转入主状态。

现在就更容易生成不同游戏状态了。一定要生成所需的一切，包括设置和帮助屏幕。

18.2 游戏设计要素

记住，任何游戏的目的都是娱乐。

首先，游戏应很容易访问，使人们能够娱乐。游戏要易学易玩，更要容易下载和安装。

游戏本身要有好玩的内容、刺激的环境和迷人的故事，使人们得到享受。本节介绍如何改进游戏的这些细节。

另外，为了得到更多游戏设计概念，最好先多玩一些游戏（也许你已经玩过很多游戏），注意这些游戏的各个细节。其中有大量游戏设计概念可以模仿和改进，但不能直接抄袭。

18.2.1 环境

世界是丰富多彩的，这种多样性也可以增加游戏的趣味，使游戏者期望看到后面的东西。

在游戏中增加变化的最佳方法是提供不同环境。不一定要把整个游戏都放在充满异族敌人的废墟之中，可以把游戏的每一关放在不同环境中，如森林里、下水沟里、沙漠中、

洞穴里、星际战舰里、小溪旁或任何别的地方。此外，不同环境中的敌人也不相同，如太空中的机器人和森林里的超大昆虫。

另外，天气变化也可以使游戏更加出彩。可以在一关的中间开始下雨，或者游戏者回到上一关时，原先的草地已经盖上一层白雪，天空中布满乌云。

还可以在每一关增加不同纹理和物体。重复前面关卡中的纹理可以让游戏者增加亲切感，但同一纹理用得太多则也会令人乏味。

环境还应允许某种交互，可以是炸开石头寻找隐藏的门，移动雕像以档住敌人的道路或加进其他效果，如地板上留下爆炸痕迹或破坏草地。

但是,一定不要用环境调戏游戏者。如果一个放射材料箱看样子能爆炸，就让游戏者将它爆炸吧，即使没有什么实际用处，无用的交互也能增加游戏的真实感。一个很好的例子是 Zelda 系列中，游戏者可以破坏草地、砍掉树木和抓住小鸡。

隐藏的宠物也可以在冒险游戏中增加不少乐趣。可以增加许多秘区和宝藏，这不是完成游戏所必须的，但可以帮助游戏者，使游戏更有娱乐性。

18.2.2　故事要素

说到娱乐，就要考虑游戏的故事。为什么游戏者要炸掉所有这些机器人呢？

有趣的故事可以刺激游戏者在游戏中达到不同目标，可以使整个游戏更有乐趣。故事的深度取决于目标对象。小孩几乎不需要故事，而成人则最好多一些故事性。

至少可以在游戏开头提供背景故事，在游戏者取胜的游戏结束序列中提供一些背景故事。更复杂的故事可以在关卡之间显示片断或在游戏本身建立情节。

片断不一定是漂亮的录像，也可以用游戏引擎本身生成，在游戏者交流时使用说话冒泡，就像动画喜剧书中一样。至少，片断可以在装饰屏幕上显示一个小故事。

故事还可以在游戏中进行，让某些角色向游戏者说话。他们不一定要实际产生声音，只要使用说明冒泡就可以了，因为声音要增加大量工作。

故事不能使用太老套的情节，应该有新意。

游戏的最终目标不一定要让游戏者一目了然。游戏的局部可以是神秘的（为什么空间站被机器人占领了？），刺激游戏者去寻找答案。

神秘故事通常在读者答出一个问题之后又出现更多问题。例如，为什么空间站被机器人占领了空间站小组并没有被机器人杀死，而是在机器人到达之前离开了。为什么要放弃空间站？随着情节的深入，可以向游戏者提出更多问题。

此外，曲折的情节可以增加故事的趣味。例如，游戏者发现不是被警察雇来的，而是被悬赏者雇来的。

故事结尾也不一定要回答所有问题，妙手总是喜欢留有悬念。

共享软件演示程序末尾扣人心弦的故事最能吸引用户玩整个游戏，了解接下去要发生什么事。

最后，游戏不一定每次都有相同的结局。可以根据游戏者的决策而改变情节，允许有多种结局。这样，可以增加游戏的回头率。图 18-3 是个例子，图中显示了具有不同结局的情节树和故事每个部分相关的游戏关卡。

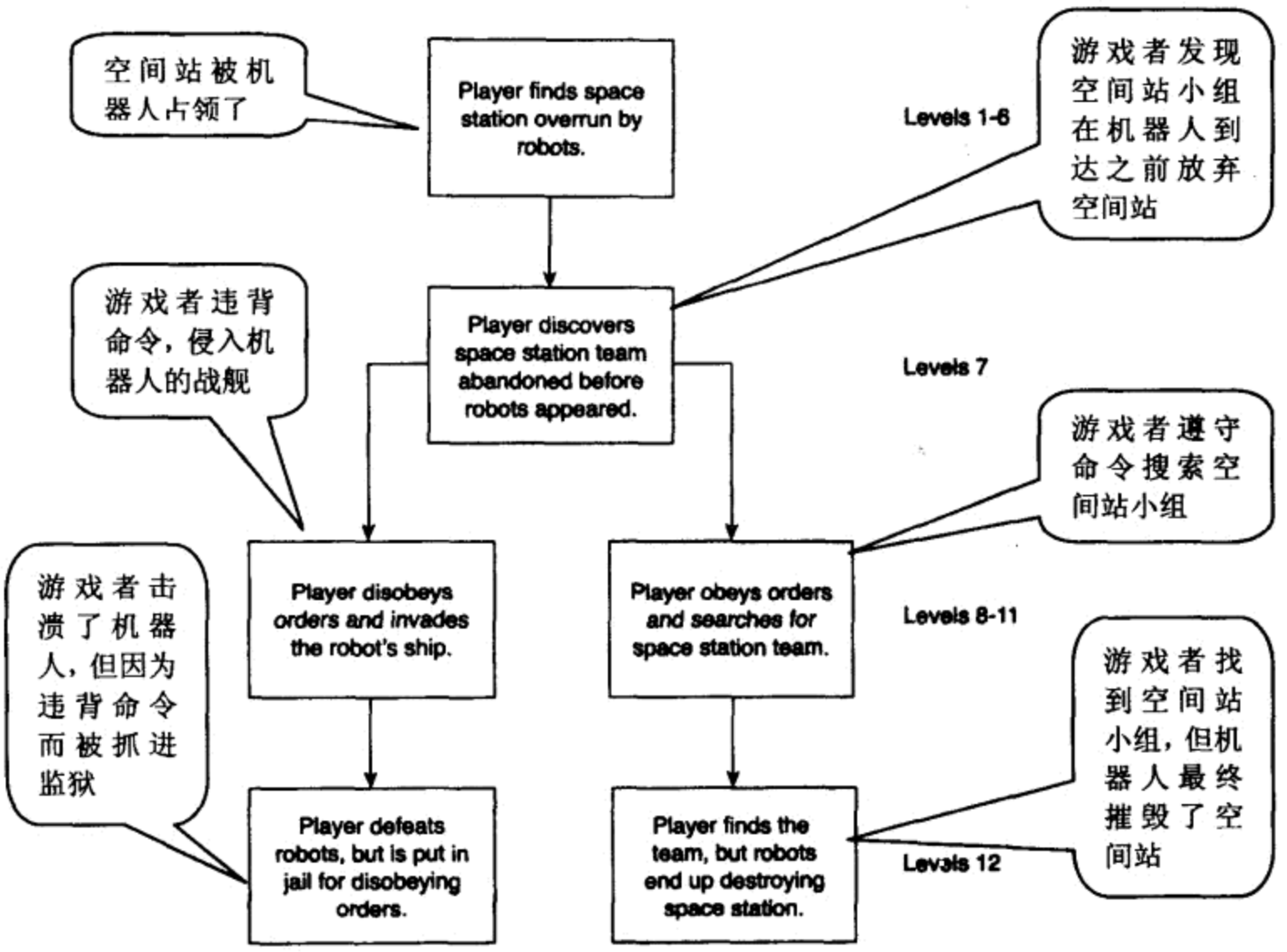


图 18-3　具有三种不同结局的情节树，不一定每次都有相同的结局

18.2.3　玩游戏

除了有趣与刺激之外，玩游戏还可以分解为两个主题：目标和奖励。

目标可以在游戏中显示，也可以由游戏者确定。例如，在冒险游戏中，最终目标可能是寻找某个宝藏，但在 SimCity 之类更开放的游戏中，游戏者可以自己确定目标，如是否建立大都市或无犯罪和环境优美的乡村。

除了游戏的最终目标外，还有其他目标，可以分解为长期目标和短期目标。长期目标可能是进入下一关，取得新武器和新能力，或扩大游戏者的地盘。而短期目标可能是解谜题、找到东西或爬过悬崖。

如果没有目标，游戏者就可能失去兴趣。当然，玩游戏不一定是线性的，但一般来说，要保证游戏者总是有一些目标。

另外，游戏过程中，目标应越来越难解决，使游戏难度不断增加。游戏难度不断增加可以不断增加挑战性，从而使游戏保持趣味性。同样，也可以让游戏有不同难度，吸引不同水平的游戏者。增加难度就是增加敌人、增加挑战或减少物资供应。

但游戏者的动机是什么？游戏者为什么要达到这些目标呢？

答案是奖励。奖励使用户对玩游戏保持兴趣，达到目标或完成某个动作时，都要给予奖励。

奖励不一定针对游戏者，如点数和奖品，也可能针对用户，如特殊视觉效果或音乐序列。下面是游戏中奖励的例子：

- 捡到钱币得到奖金。
- 看到特殊视觉效果或听到音乐序列。
- 转到游戏中的隐藏区（让用户感到聪明）或发现两个地方之间的捷径。
- 进入下一关或没有见过的游戏区。
- Bot 破坏时看到“满意”的压碎或爆炸。
- 打败强大的敌人，可以加一些“老板”敌人。
- 得到新武器、新东西或新能力。

奖励应是意外的，如果每次奖励都是找到钱币，则会令人乏味。另外，还要考虑游戏者得到奖励的频率，可能要保证游戏者每分钟得到几个奖励，太久没有奖励可能会使游戏者感到乏味。

18.2.4　告诉用户怎么玩

告诉用户怎么玩涉及两个部分：游戏输入和玩游戏。显然，要让游戏者知道怎么玩，否则他们就不会去玩，因此你要告诉用户怎么玩。

使用键盘和鼠标的输入应尽量容易理解，不能让游戏者按太多按钮或要求古怪的键组合。另外，使用其他游戏中常用的键盘和鼠标控制，使游戏者更容易上手。例如，第一人称视角射击游戏通常使用 WASD 键盘配置，其中W向前移，A 向左移，S 向后移，D 向右移。

玩游戏也要逐渐教，而不能一下子把全部内容倒出来，如 10 种不同武器和好几种跳跃方式，以及如何使用每种游戏实体。应该先介绍基础，然后逐渐增加用户的游戏知识。这样可能要实现教程或让游戏者一开始只有几个能力，后面不断增加其他能力，或一开始只有几个不同的游戏实体，后面再增加更多游戏实体。

但不能让用户觉得是在学习。有些游戏完全寓教于乐，边玩边会，而不是使用教程对话框或讨厌的角色。

但是，如果学习经历让熟练游戏者感到太慢，则可能要让用户跳过教学过程的某些部分或从高级关卡开始玩游戏。

这样就完成了游戏设计。记住，进行游戏设计时，要特别注意细节，目的是让游戏更有趣，这方面应该不遗余力。

18.3　生成贴图编辑器

关卡设计使游戏真正变得有趣，关卡设计员通常要多次编辑和处理任何关卡。建立不同关卡的游戏时，贴图生成过程应尽量简单。

在文本文件中手工编辑贴图是很麻烦的，此外，如果让别人作为游戏的关卡设计员，则贴图生成过程应尽量理顺，以免花太多时间进行支持和培训。

应当使关卡设计员能够使用他们熟悉的工具。如果他们要使用 x 工具，则要编写一个分析器，读取 x 工具生成的文件，或要编写一个转换器，将 x 工具的文件转换成游戏使用的文件格式。

但是，如果关卡设计员要使用最新的昂贵和全能的三维工具，则会把业余爱好者挡在

门外。

因此，应考虑生成免费易用的贴图编辑器，使任何人都能生成和编辑游戏的贴图。免费贴图编辑器应当是针对特定游戏的，让别人很容易生成和编辑游戏的贴图，包括放置物体和编辑简单脚本。三维实体或纹理不需要提供编辑器，因为它们可以在外部程序中生成。

生成免费贴图编辑器有几个好处：

- 很容易生成关卡，从而扩展游戏。
- 别人很容易生成关卡，使你不必花太多时间进行支持和培训。
- 发布编辑器有助于围绕游戏建立用户社区。
- 用户生成内容可以在游戏中增加变化和回访率。

如果生成贴图编辑器，则应使其尽量易用，同时提供游戏中的每个选项。可以在贴图编辑器中提供下列功能：

- 提供设计环境的简便方法，可以是二维或三维的。在 SimCity 中建路和 Sims 中建房子非常简单。
- 提供导入三维图形格式的导入功能。
- 提供存储纹理与声音的媒体库。媒体应能用拖放夹方便地增加，关卡设计员可以用拖放法将纹理放进环境中（墙、地形，等等）。
- 提供公用结构库，存储常用环境结构。关卡设计员可以存储电梯间、通风道之类的常见项目。
- 拖放放置环境效果，如灯光。
- 拖放放置游戏实体。
- 方便地调整游戏实体属性。
- 在任何实体中增加脚本的方便接口
- 提供预览按钮，显示贴图在游戏中的样子。
- 提供启动按钮，用用户所用的贴图自动启动游戏。
- 提供自动部署按钮，把游戏需要的所有文件放在一个地方。

对 Java 游戏，可能要用 Swing UI 工具库实现贴图编辑器，使贴图编辑器灵活和跨平台。建立贴图编辑器时，要记住 cut,copy,paste,undo 与 redo 之类常用的应用程序命令。

说明：*有时建立贴图编辑器非常有趣，你生成贴图编辑器所花的时间比生成游戏所花的时间还多，一定要合理分配自己的时间。*

18.4 调试

任何项目都会有缺陷，因此要进行调试。缺陷类型包括简单显示问题、异常造成崩溃或游戏不能如期工作。

可以把缺陷分成三类：

- 设计缺陷。例如，碰撞探测代码可能没有处理某些最初设计时没有考虑到的情形。由于这些缺陷涉及重新设计部分游戏，因此较难修复。
- 人为错误造成的缺陷。可能是代码中缺了什么或输入有错。这类缺陷容易修复，但有时很难跟踪。

- 机器差别造成的缺陷。一台机器（或虚拟机）上可行的代码，在另一台机器（或虚拟机）上可能行不通。如果无法访问发生缺陷的机器，则这类缺陷很难修复。

调试的最好方法是不出现缺陷，认真设计和编写代码可以避免后面花更多时间跟踪奇怪的缺陷和重新设计大段代码。要避免机器差别造成的缺陷，只要注意 API 方法中潜在的异常和返回值。

下列缺陷防止方法有助于减少缺陷数量：

- 不要编译、运行就万事大吉。编写代码时，一定要检查是否遗漏了什么。如果感觉不对，再检查一遍。要信号代码可行之后再运行。
- 注意 API 文档。保证了解方法规范、捕获异常、检查返回值和传递合法参数。
- 建档代码。如果对代码的工作方法不熟悉，则很难进行调试。文档可以帮助澄清问题（我在这个方面还不够好）。
- 增加代码可读性。使用有意义的描述性变量名（如 numberOfItems）而不用简短的歧义名称（如 num）。
- 编写单元测试，经常运行游戏。单元测试是简单指令，用不同方式测试方法，保证其符合规范。然后可以在改变方法时运行这个单元测试，保证一切仍然正常工作。但并不是一切都要单元测试。不要花大部分时间编写测试代码，还要生成游戏呢！
- 记录错误、警告和信息，以便在发生错误时跟踪缺陷，最终版本中可以缺省关闭日志。
- 最终你会得到一种“直觉”，一眼看出问题在哪里。

根据我的经验，许多难找的缺陷通常发生在一次编写或改写大量代码（几千行）而不进行任何运行测试。最好一次编写或改写少量代码，以便逐步测试代码补充与修改。

当然，无论如何慎重，还是会出现缺陷。出现缺陷时，可以按下列方法进行修复：

- 遇到异常时，保证在调试方式中编译代码，并从控制台运行游戏。这样就可以得到堆栈跟踪，以便确定方法中抛出异常的准确行号。由于很容易找到异常源，因此这是 Java 中最容易解决的问题之一。
- 用调试器一行一行单步执行代码。
- 编写更多日志语句，缩小缺陷源范围。
- 如果怀疑线程死锁，则可以从控制台运行游戏，等待发生死锁，然后返回控制台并在 Windows 中输入 Ctrl+break（或在 Mac OS X 与 UNIX 中输入 Ctrl+\）。这样就可以显示线程状态、取得的锁和发现的任何死锁。

println debugging 是缩小缺陷范围的简单方法。例如，假设处理两个 Point 实例时有缺陷，则可以向控制台打印每个点的值：

```
Point p1 = new Point(x1, x2);
Point p2 = new Point(y1, y2);
System.out.println(“Two points: “ + p1 + “ and “ + p2);
```

这时，我们用 println debugging 保证生成的点值正确。这个例子很简单，点生成有错，代码应当如下：

```
Point p1 = new Point(x1, x2);
```

```
Point p2 = new Point(y1, y2);
```

这个调试方法对异常很有用，可以打印抛出异常的堆栈跟踪。

```
try {
    ...
}
catch (Exception ex) {
    // 打印堆栈跟踪并继续
    ex.printStackTrace();
}
```

也可以打印堆栈跟踪而不抛出或捕获异常：

```
new Throwable().printStackTrace();
```

也可以只打印堆栈跟踪第一行(最后调用的方法)：

```
System.out.println(new Throwable().getStackTrace()[0]);
```

这里 getStackTrace()方法返回 StackTraceElements 的数组，我们打印第一个堆栈跟踪元素。

当然，并不是所有缺陷都会导致灾难结果或不正确行为。有些缺陷只在极少数情况下出问题。例如，内存泄漏就是这种问题。

内存泄漏在需要去配内存的语言中很常见，但 Java 中有内存回收单元，怎么还有问题呢？这不是内存泄漏的惟一原因。记住，内存回收单元只回收不再有引用的对象。因此，造成内存泄漏的另一个原因可能是应用程序保持不再需要的对象，使这些对象的集合随着时间不断扩大。例如，可以把对象放在 HashMap 中，其在游戏过程中保持活动。如果不断增加新对象而不清除不用的对象，则 HashMap 使用的内存不断增加，从而造成内存泄漏。一定要注意这一点。

如果探测泄漏并发现某种征兆，则可以定期检查 HashMap 长度（或其他 Collection 类长度），看看其在游戏过程中如何改变。如果长度不合理地增加，则可能是内存泄漏。

18.4.1 调试 Java2D 问题

如果怀疑游戏的缺陷与 Windows 机器有关，则首先要检查机器是否有最新显示卡。如果问题还在，则可以用下列命令行标志关闭各种 Java2D 改进功能：

```
-Dsun.java2d.d3d=false
-Dsun.java2d.ddoffscreen=false
-Dsun.java2d.noddraw=true
```

要了解 Java2D 的更多细节，可以登记绘制的一切，包括直线、矩形和图像等。这样可以帮助调试和跟踪性能问题。使用下列命令行标志：

```
-Dsun.java2d.trace=<options>
```

这里<options>是这些选项的逗号分隔清单：

- log 登记调用的每个 Java2D 绘图操作。
- timestamp 增加每个操作的时间标志（毫秒）。
- count 在应用程序退出时显示每个调用操作的次数。

- out:<filename>将输出发送到指定文件，而不是发送到控制台。
- help 显示这个标志的用法。
- verbose 显示所用选项汇总信息。

例如，可以用这个标志将每个调用操作的次数登记到文件 log.txt 中：

-Dsun.java2d.trace=count,out:log.txt

这样就可以对游戏中 Java2D 操作的用法有个一般了解。下面是第 5 章“创建二维平台游戏”地砖游戏的输出例子：

```
85339 calls to sun.awt.windows.Win32BlitLoops::
    Blit(“Short 565 RGB DirectDraw with 1 bit transp”,
    SrcOverNoEa, “Short 565 RGB DirectDraw”)
31698 calls to sun.awt.windows.Win32BlitLoops::
    Blit(“Short 565 RGB DirectDraw”,
    SrcNoEa, “Short 565 RGB DirectDraw”)
...
118723 total calls to 18 different primitives
```

这里只产生两个使用最多的图元一总共显示了 18 个图元。对这两个块移操作，第一个参数是源图像类型，第二个参数是复合类型，第三个参数是目标图像类型。复合类型相当于 AlphaComposite 类中的类型，但后缀 NoEa 指定图像没有更多 Alpha 信息。

18.4.2　日志

第 6 章“多人游戏”和上面曾介绍过，日志可以帮助诊断问题，提供游戏执行方法的警告与信息。当然，如果登记到文件中，则后面还可以分析这个日志。

利用日志还可以方便地控制日志信息的粒度，只看到警告和严重消息而不是显示所有信息。可以将日志输出进行各种改向，如改向到控制台、文件或网络流。

日志不一定是简单文本句子，也可以是数据，后面要分析成另一种格式。例如，如果调试内部二维几何贴图生成过程，则可以将贴图结构登记为 SVG 数据，然后用 SVG 浏览器取得直观化的数据。

第 6 章介绍了 Apache log4j，第 13 章“人工智能”生成了自己的日志程序，用游戏间的消息队列登记 Bot 进行的决策。除了使用这些方法和打印日志外，J2SE 1.4;中还有 java.util.logging 包，下面将简要介绍。

首先要有自己的 Logger 对象：

```
static final Logger log =
Logger.getLogger(“com.bracken.javagamebook.tilegame”);
```

这个语句取得 Logger com.brackeen.javagamebook.tilegame,如果不存在，则生成新的 Logger。可以在代码的每个类中使用这条语句，它返回相同 Logger 对象。可以在不同层次上登记语句，例如：

```
log.info(“Loading resources”);
log.warning(“No MIDI soundbank available.”);
log.severe(“Dude, we can’t find that file!”);
```

不同层次的日志可以得到每个日志语句的严重性，还可以选择只在所要层次输出信息。例如，如果只要取得严重日志信息，则可以用下列代码：

```
log.setLevel(Level.SEVERE);
```

日志程序本身用两个对象登记信息：

Handler 处理日志语句，发送到文件或控制台，Formatter 格式化日志语句的样子，还可以加上时间标志和类/方法信息。缺省情况下，Logger 用父处理器，其使用 ConsoleHandler 与 SimpleFormatter。下面的例子使用自己的处理器和格式化器：

```
//登记到控制台
ConsoleHandler my Handler = new ConsoleHandler();

//使用基本格式化器
myHandler.setFormatter(new Formatter() {
    public String format(LogRecord record) {
        return record.getLevel() + ": " +
            record.getMessage() + "\n";
    }
});

//只用我们的处理器
log.setUseParentHandlers(false);
log.addHandler(myHandler);
```

这里使用 ConsoleHandler，建立自己的 Formatter 和关掉父格式化器。

当然，不同处理器有多种可能性。例如，可以生成一个处理器，将严重日志语句发送到外部服务器。这样就可以看到玩游戏时遇到的严重问题。

如果这样，则一定要弹出一个对话框，告诉用户你要发送什么数据，询问是否可以将这些数据发送到你的服务器上。否则人们可能对计算机发送数据感到不安，即使这些数据不会标识他们或其计算机。

在最后游戏中，可能要关掉所有日志如下：

```
log.setLevel(Level.OFF);
```

注意：*即使关掉日志，仍然生成为日志记录生成的对象，仍然进行日志语句的任何计算。例如，在下列日志语句中，仍然计算距离并非其换算为字符串，然后添加到现有字符串后面：*

```
log.info("Dude, the distance is: " + point.distance(x, y));
```

显然，如果一个帧中多次计算两点之间的距离，则会影响性能。如果要避免这类事情，则可以用查找与替换功能，将 Java 文件中每个日志语句标为说明语句。

18.5 保护代码

也许你已经知道，Java 类文件保护不太严，容易被反编译。大量免费的 Java 类文件反

编译器（如 Jode 与 Jad）可以将类文件反编译成 Java 文件，甚至显示原方法与字段名，只要一个鼠标单击就可以搞定。

如果游戏是开放源代码，则没什么问题，但还是可能要保护代码，例如防止多方游戏中使用秘笈或防止别人复制和利用你的工作成果。

要保护代码，一种方法是生成自己的类加密机制。但是，还要生成“类装入器”类以实际进行解密。这个类装入器可能被反编译，造成问题：如果能够反编译解密方法，则可以解密类文件，将游戏反编译。尽管这不是安全方案，但可以增加游戏反编译的难度。

一个不太完善但非常有用的代码保护方法是使用掩盖器。掩盖器把类名、方法名和字段名搞乱，使反编译得到的类文件更难阅读和理解（但还是可以阅读和理解）。例如，掩盖器将下列代码：

```
public int getSecretValue(int code) {
    return translator[code];
}
```

变成下列代码：

```
public int a(int a) {
    return b[a];
}
```

当然，掩盖器搞乱之后的名称仍然产生有效 Java 代码，虚拟机并不认为类文件有问题。

掩盖方法有两个问题。例如，用 Class.forName()方法取得指定类行不通了。另外，还要防止掩盖插入件和脚本使用的公开 API。

掩盖器的另一个好处是可以大大减小类长度，使发布文件更小。

一些重要掩盖器包括 ProGuard(免费)与 DashO（商业版）。

18.6　游戏部署

准备在 Web 上部署游戏时，要让人们易于下载和安装，希望每个要玩游戏的人都有机会玩。

换句话说，要避免下列情形：

- 用户单击下载 Zip 文件的链接，等待下载。
- 用户解开文件，看到一些 jar 文件、一些类文件、一些图像和一个自读文件（这要求用户知道如何解开文件）。
- 熟悉 jar 文件的用户会双击 jar 文件，希望它运行，但什么也没发生。这时，就不知道如何运行游戏了，许多用户只好放弃。
- 有些用户会打开自读文件，其中要求输入下列代码以运行游戏：

  ```
  java - cp game.jar coolgame.MainClass
  ```

  但许多人根本不知道什么是命令行，这时又会失去许多用户。
- 用户打开命令窗口，但不知道如何转入 zip 文件扩展的文件夹，因此命令行不通。

这是一场恶梦，谁也不希望花时间下载的游戏居然无法运行。即使技术水平很高的人，也会对这么多步骤不胜其烦。

因此，所用的游戏部署方法一定要让大多数电脑盲都知道如何处理，用最少的步骤完成操作。涉及的步骤越多，人们遇到的问题就越多。

下面是另外几种游戏部署方法：

- 小程序非常方便，运行小程序时只要导航到 Web 页面即可。但这要求每次访问平面时下载小程序，而且小程序有许多限制。另外，大多数浏览器只支持 Java1.1 小程序。
- 可执行 jar 文件能节省大量时间，但有时看不出 jar 文件是可执行的。
- 生成自己的安装程序，可以是可执行 jar 文件或自然可执行文件。
- 用 InstallAnywhere 三类现有安装产品（免费和商业版都有）。

另一个问题是许多人的机器上没有最新 Java 运行环境，因此要单独下载。但有些安装产品可以在安装程序中捆绑 Java 运行环境。

考虑安装程序时，要考虑到有些人不知道下载安装程序之后要运行安装程序，他们以为下载的东西会自动安装，不要再干什么了，或者他们会忘记运行安装程序。

最好能把下载、安装和执行过程自动化。换句话说，只要执行下列步骤：

- 用户单击下载 Zip 文件的链接，等待下载。
- 游戏自动安装和运行。

Java Web Start 就是起这个作用的。

18.7 Java Web Start 游戏部署

Java Web Start 是个部署技术，可以把下载、安装和执行过程自动化，用户只要单击 Web 页面上的一个链接，其余一切都自动化。这是最简单的游戏部署方法之一。

要使用 Java Web Start，游戏设置时要注意口令：

- 将所有游戏资源（图像、声音和贴图）放在一个 jar 文件中。
- 签名 jar 文件。
- 生成 JNLP 文件，描述游戏及其资源。
- 将 Web 服务器设置成正确提供 JNLP 文件。

18.7.1 从 jar 文件取得资源

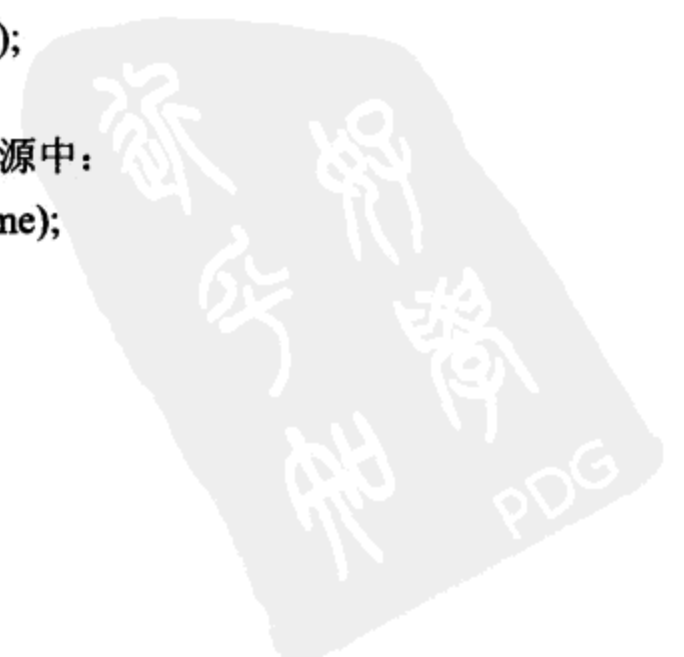
首先要保证所有游戏资源存放在 jar 文件中，可以是独立 jar，例如可以有一个代码 jar 文件，一个声音 jar 文件，一个图像 jar 文件，等等。

在游戏中，可以从 jar 文件取得文件（只要其在类路径中），只要使用 ClassLoader 的 getResource()方法如下：

```
Sering filename = “path/image.png”;
ClassLoader classLoader = this.getClass().getClassLoader();
URL url = classLoader.getResource(filename);
```

其返回资源的 URL。也可以直接将 InputStream 放到资源中：

```
InputStream is = classLoader.getResourceAsStream(filename);
```



18.7.2 签名 jar 文件

然后要签名 jar 文件（所有，即使不包含任何代码）。

Java Web Start 要求签名 jar 文件之后才能对游戏指定正确的安全权限。例如，要有切换到全屏独占方式的安全权限。

签名 jar 文件相当于签合同，签名保证是你写的，程序不会伤害用户机器。首先要用 J2SDK 所带的 keytool 命令行工具生成密钥：

keytool - genkey - keystore samplekeystore - alias Nobody

其生成别名为 Nobody 的新密钥，放在文件 samplekeystore 中。keytool 命令行工具提示输入口令和其他信息，包括姓名、单位、城市、州和国家。

要签名 jar 文件，可以用 And 建立文件中的<signjar>任务如下：

```
<signjar jar=" ${basedir}/tilegame.jar"
    storepass=" secret"
    alias=" Nobody"
    keystore=" ${basedir}/samplekeystore" />
```

这里指定别名 Nobody，口令、密钥库和要签名的 jar 文件。记住，要签名游戏中的所有 jar 文件。

18.7.3 生成 JNLP 文件

然后要生成 JNLP 文件。JNLP 文件指 Java 网络启动协议文件，描述游戏及其所需的安全权限和资源（jar 文件）。

JNLP 文件是个 XML 文件，见清单 18.4 的例子。

清单 18.4 tilegame.jnlp

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp   spec="1.0+"
  codebase="http://www.brackeen.com/javagamebook"
  href="tilegame.jnlp">
  <information>
    <title>Tile-based game</title>
    <vendor>David Brackeen</vendor>
    <homepage href="."/>
    <description>Tile-based game</description>
    <description kind="short">A demo from chapter 5.</description>
    <icon href="icon.gif"/>
    <icon kind="splash" href="splash.gif"/>
    <offline-allowed/>
  </information>
  <security>
```

```
        <!-- get all permissions so we can access
        fullscreen exclusive mode -->
        <all-permissions/>
    </security>
    <resources>
      <j2se version="1.4+" initial-heap-size="32m"/>
      <jar href="tilegame.jar"/>
      <jar href="tilegameres.jar"/>
    </resources>
    <application-desc
      main-class="com.brackeen.javagamebook.tilegame.GameManager"/>
  </jnlp>
```

这里要设置游戏的代码基（www.brackeen.com/javagamebook）和 JNLP 文件相对于代码基的引用(tilegame.jnlp)。

在信息段中，有两个图像的引用：图标和装饰图像。图标用于 Java Web Start 应用程序管理器，如图 18-4 所示。

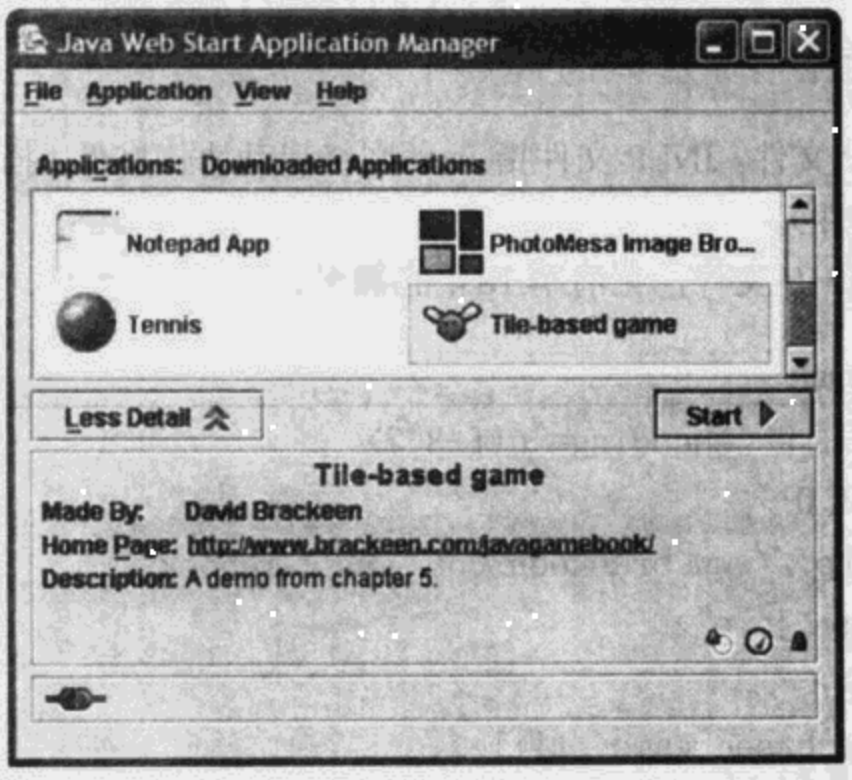


图 18-4　Java Web Start 应用程序管理器显示每个应用程序的图标以及名称、主页和描述

这里显示图标的 32 × 32 版本，下载应用程序时使用图标的 64 × 64 版本（如图 18-5）。Java Web Start 可以将图标缩放到所要的任何尺寸。图标格式为 GIF 或 JPEG 格式，这里使用 GIF 格式，利用其透明度。

操作系统的窗口管理器也使用图标，例如，用户在 Windows 的 Start 菜单中放上应用程序链接时。

装入应用程序时，显示装饰屏幕。Java Web Start 打开 Java 虚拟机新实例，运行应用程序，在此期间显示装饰屏幕。如果没有装饰屏幕，则使用图标，显示图 18-5 所示的下载对话框。

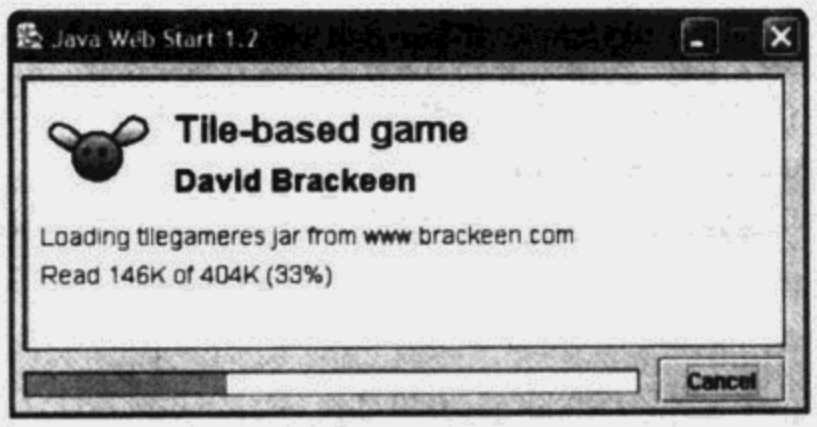


图 18-5　Java Web Start 应用程序管理器下载应用程序时使用图标的 64 x 64 版本

权限在<security>标志中设置，是全屏方式所要的，惟一选项是 All Permissions，因此也可以进行其他操作，如将游戏保存到用户计算机或访问远程服务器。

JNLP 文件其余部分指定所要的 jar 文件（相对于代码基）和要启动的主类。注意 Java Web Start 应用程序忽略 jar 的 manifest 文件：还是要在 JNLP 文件中指定主类。

18.7.4　设置 Web 服务器

对 JNLP 文件设置 Web 服务器其实只有一件事，就是将 JNLP 文件关联到正确的 MIME 类型:application/x-java-jnlp-file。这是必要的，使 Java Web Start 客户机能够装入文件而不是 Web 浏览器。

如果有自己的 Apache web 服务器，则要编辑 mime.types 文件（通常位于 /etc/mime.types）。只要在文件中增加下列语句：

application/x-java-jnlp-file JNLP

然后重新启动服务器：

/etc/rc.d/init.d/httpd restart

如果在共享 Web 服务器上运行或无法取得根访问权限，则要请 Web 管理员进行改变。如果 Web 管理员不能或不愿改变，则还有另一个办法：在 Web 站点的顶层目录中加上.htaccess 文件，其中只有一行：

AddType application/x-java-jnlp-file　　.jnlp

注意：这只适用于 Apache web 服务器设置成读取.htaccess 文件的 AddType 命令时。如果不能确定，可以询问 Web 主人。

有时很难判断你的改变是否凑效。如果前面在浏览器中打开 JNLP 文件，则可能要重新启动浏览器或清除浏览器缓冲区。如果仍然无法判断发生的情形，则可以用清单 18.5 的代码取得远程文件的返回 MIME 类型。

清单 18.5　GetContentType.java

```
public class GetContentType {

    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.out.println("Shows the value of the " +
```

```
                "\"content-type\" header field of a remote file.");
            System.out.println("Usage: GetContentType (url)");
            System.out.println(
                "Example: GetContentType http://www.yahoo.com");
        }
        else {
            URL url = new URL(args[0]);
            URLConnection connection = url.openConnection();
            System.out.println(connection.getContentType());
        }
    }
}
```

可以使用这个简单程序如下：

java GetContentType http://server/path/mygame.jnlp

如果 Web 服务器设置正确，则程序在控制台中打印 application/x-java-jnlp-file，否则打印 text/plain 或其他内容。

这样就完成了 Java Web Start，另一个有用特性是用一些 JavaScript 与 VBScript 代码探测是否安装了 Java Web Start，以及在 Windows 机器上单击安装 Java 运行环境。可以到下列站点下载 JavaScript 代码：java.sun.com/products/javawebstart/1.2/docs/developersguide.html。

18.8 自然编译游戏部署

使用 Java Web Start 的一个缺点是 Java 运行环境很大（10MB 以上），用户如果没有现成 Java 运行环境，则要花时间下载，对 Modem 连接用户很不方便。

另一种还可以在 Web 上发布游戏的方法是用自然编译将 Jar 类编译成自然代码，目的是不需要 Java 运行环境，但这个思想是行不通的。

自然编译 Jar 类的两个产品是 GCJ 和 Excelsior JET。但是，GCJ 目前不支持 AWT 与 Swing。Excelsior JET 不支持 AWT 与 Swing，但实际上要有 Java 运行环境才能工作。简而言之，对图形应用程序，用户要在系统中安装 Java 运行环境，因此用不上自然编译。

18.9 更新与补丁

Java Web Start 的一个妙处是可以自动下载游戏的最新 jar 文件，不必另外提供更新与补包支持。Java Web Start 只下载改变的 jar 文件，因此只是下载代码 jar 或媒体 jar 文件，而不是一次性全部下载。

如果不用 Java Web Start，至少要提供游戏中检查是否有更新的途径。如果发现更新，则可以在游戏中下载更新或提供 Web 站点链接，使用户可以下载。

检查最新版本很简单，只要把文件放在 Web 服务器上包含最新版本号的地方，例如：

latest-version: 5.0.6

更新程序检查这个文件，看看其版本是否低于文件版本。如果是，则说明有新更新，游戏可以相应采取措施。

18.10　带宽问题

如果你的游戏只是个演示，要交给朋友或加进私人资产中，则不必把它放到 Web 站点上。但如果要让更多人玩你的游戏，要把它放到 Web 站点上，则很可能遇到带宽问题。

即使你的游戏在雷达下面，也不一定就不会普及。只要你的站点很普及，或一个站点中有许多朝你链接的通信流，则每天可能有成百上千人下载你的游戏。听起来不错吧。

但是记住，这需要用到带宽。世界上没有真正的“无限带宽”，每个 Web 宿主公司都会限制你的 Web 站点可用的带宽，通常每月 10GB 左右，甚至只有 1GB。如果有专用服务器，则可以得到更高带宽，通常为每月 25GB 甚至到 400GB 以上。

假设游戏用 1MB 的捆绑产品发布，如果宿主限制你的带宽为每月 10MB，不考虑 HTML 页面和图形之类内容，每月也只能下载一万次，每天也只能下载 333 次。此外，宿主公司或者会对这么大的通信流收取巨额费用，或者拨去你的站点连线。

这样，并不是每个想下载游戏的人都能下载你的游戏，你必须解决这个问题。

要解决这个问题，最简单的方法是不在宿主上放游戏，而只在 download.com 之类站点上提供下载。如果实际要靠游戏赚钱，则可以找另一家宿主公司，看看他们的带宽选项，估计要用多少带宽，选择能让你根据需要方便地扩展的公司。

18.11　反馈与 Beta 测试

一般来说，我们要在尽量多机器上测试游戏，看看其在不同硬件和操作系统中的运行情况。尽管 Java 是跨平台的，但游戏在不同操作系统中可能表现稍有不同，取决于处理器、处理器速度、内存、显示卡和计算机上的其他硬件。

如果不能找到不同的计算机，则可以在 Web 上找足够的用户帮你测试游戏。只要让他们能方便地下载游戏，如用前面介绍的 Java Web Start。

测试期间，发布最终游戏之前（通常称为 Beta 测试），要提供缺陷报告 E-mail 地址、Web 窗体或游戏中的窗体，使用户很容易提供反馈。提供游戏中的缺陷报告工具可以使用户更容易报告缺陷，这个工具可以提供用户环境信息，如操作系统和 Java 版本。可以用 System 类的 getProperty()方法取得这个信息。

放上游戏之后，可以从不同用户取得大量反馈，而不只是机器特定问题和缺陷。这个反馈可能是调整与改善游戏的最佳方法。可能得到游戏下列方面的反馈：

- 玩游戏。游戏的难度与平衡性如何？
- 图形与声音。与其他游戏相比如何？
- 性能。是否顺畅运行？
- 易用性。控件是否直观？游戏有没有教游戏者按满意速率玩游戏？
- 趣味性与回访率。是否好玩？能否百玩不厌？
- 安装过程。安装与运行程序是否很容易？

一定要提供很容易访问的 E-mail 地址、反馈窗体或讨论公告牌，使人们能提出建议和提供游戏反馈。

另外还要注意，Internet 中许多人不能提供建设性批评，不要抵触质问和攻击性意见，而要把这些意见理解为有意义的东西。例如：

用户意见：差，你的键盘控件真差，太差了！

理解：我发现游戏中的键盘控件与我习惯的不同或很难控制，建议进行改进。

用户意见：这个游戏真蠢，玩不了。

理解：或者是下载与安装过程太难，或者我的系统中无法运行。可能要提更多问题，帮我找出原因。我的拼写和语法也有问题。

用户意见：差劲，xx 游戏好多了。

理解：我发现另一个游戏更好玩，但不知道为什么，因此我的建议无法帮你改进游戏。

但通常，我们可以收集到大量意见，涉及如何使游戏更好玩更直观。一定要多听意见，采纳良言。但要记住，不一定要实现人们请求的每个特性，否则游戏中特性太多，玩起来负担很大，或者变得不好玩了。要知道该听什么，不该听什么，要知道在适当时候说“不”。

18.12 赚钱

你是否想用 Java 游戏换成桌上的面包？

靠游戏赚钱是最大的难题之一，如何赚钱取决于瞄准的目标用户，游戏的类型和精彩程度。但靠游戏赚钱是可能的，许多 Java 游戏已经赚到钱了。

一种间接用游戏赚钱的方法是建立一个演示程序，帮你谋到更高收入的职位。

如果要把游戏放到零售店中，则要找一家出版商，这是最难的事，也可能是费钱的事。下面还有几种用游戏赚钱的方法：

- 将游戏引擎的许可证发给第三方。如果你的游戏独特、灵活、文档良好，则其他开发公司可能在自己的产品中使用这个游戏引擎的许可证。
- 为商业化 Web 站点生成品牌小程序。例如，饮料公司可能要你帮其 Web 站点生成“狂欢大赛”游戏。
- 在游戏中或旁边打广告。
- 把游戏作为共享软件，免费推出前几关，而游戏完整版本收费。可以用 PayPal,Kagi 与 BMT Micro 之类联机付款系统收费。
- 对多方联机游戏收取月费。

作为 Java 游戏的例子，Race3D 引擎已经向多家公司推出许可证和定制，包括 Snapple 与 Subaru。Jagex 公司的 RuneScape 是个多方联机游戏，已经在收月费。雅虎公司的游戏旁边有个广告。移动电话上也有 J2ME 游戏。另外，商业化光盘游戏“You Don’t Know Jack”与“Who Wants to Be a Millionaire?”也是部分用 Java 写成的。

18.13 综合起来

作为本章一些思路的例子，本章的源代码中带有第 5 章地砖游戏的变形，显示不同游

戏状态（包括一个装入屏幕），用 java.util.logging 包登记，并提供 JNLP 功能。这个游戏还可以继续改进，这个工作留给读者。

18.14　小结

本章介绍了各种课题，帮助读者完成游戏的最后 10%。我们介绍了游戏的状态机、游戏设计与编写情节、玩游戏、调试、日志、部署游戏、Java Web Start 和如何用游戏赚钱。

完成游戏是个难题，需要很多时间，有时好像漫无目的，但最终结果是值得的，因此一定要努力完成游戏。

第 19 章　未来

关键主题

- Java 的演变
- Java 1.5 （Tiger）
- Java 平台需要什么
- 新设备与 Java Games Profile(JSR 134)
- 小结

把“未来”写在白纸黑字上显得有点傻。未来是很难预测的：有些预言会过去，有些则不然，有些事情是谁也意想不到的。读者拿到这本书时，所谓“未来”已经过去，世界又面貌一新了。

但本章不只是介绍编写本书时的未来，还要介绍 Java 平台的未来。

本章介绍未来的三个方面：Java 的演变、Java 的未来和 Java 平台需要什么来改进 Java 游戏开发。

19.1　Java 的演变

尽管 Java 的走向最终要由 Sun 公司确定，但许多特性是在用户请求下建立的。因此，一定要敢于对 Java 的演变提出意见，善于对 Java 的演变提出意见，使 Sun 公司能听到你的意见。两个最主要的渠道是 Java 社区过程和缺陷天堂（Bug Parade）。

19.1.1　Java 社团

Java 技术规范与实现通过 Java Community Process（位于 www.jcp.org）演变。任何 Java 社团的成员都可以提供 JSR（Java 规范请求），定义新的 Java 规范或更新现有 Java 规范。

大多数开发人员都不愿意花时间提交 Java 规范请求，但 JCP Web 站点允许审查和建议现有 JSR。另外，从这个站点还可以看到 Java 平台需要什么的未来走势。

本章介绍几个 JSR,与编程或技术有关，并介绍对 Java 游戏有用的 API。

19.1.2　缺陷天堂（Bug Parade）

Bug Parade,(位于 developer.java.sun.com/developer/bugParade)是提交缺陷报告与 RFE(改进请求)的中央地址，可以检查现有缺陷报告与 RFE 的状态。缺陷报告与 RFE 适用于任何 Java 技术，无论它属于 J2SE、Sun 公司的另一个 Java API 或某个虚拟机。

当然，如果要报告缺陷或请求特性，可以先搜索 Bug Parade，也许别人已经报告这个缺陷或请求这个特性，只要投上一票或把它加进自己的观察表中。

本章介绍几个改进 Java 游戏的特性和与其相关的缺陷 ID。

19.2 Java 1.5（Tiger）

Java 1.5 代码名为“Tiger”，今提供许多潜在技术，但我们很难预测其实际内容。目前它预计在 2004 年上半年推出。

本节介绍 Tiger 中准备推出的新技术及如何使用这些新的 Java 语言补充。

19.2.1 一般（JSR 14）

与其他语言不同的是，Java 中每个对象都是 Object 类的子类。因此，集合类（List,Set,Map，等等）设计成包含 Object，可以在集合中放上任何对象。

但是，这样使用集合是有问题的。例如：

```
//这个表中可以放任何
List stringList = new ArrayList();
stringList.add( “Hello” );
stringList.add( “World” );

//会在后面产生 ClassCastException
stringList.add(new Integer(13));

for (int i=0; i<stringList.size(); i++) {
    String s = (String)stringList.get(i);
    System.out.println(s);
}
```

在这个代码中，字符串表可以放任何类型的对象，而不只是 String 对象的实例。如果在表中加进另一个类（如 Integer），则运行时对象转换成 String 时虚拟机会抛出 ClassCastException 异常。

当然，可以用更小心的编程解决这个问题，但谁也不是完人。

一般性（Generics）或称参数化类型（parameterized types）或模板（templates）可以解决这个问题。Java 语言中要增补的一般性可以生成一组具有相同行为而适合不同类型的类。例如，list 集合类对所包含的对象具有类似行为，但特定 List 可能处理 String、Integer 之类不同类型。这样就在编译时而不是运行时进行类型检查。

代码中生成 List<String>或 List<Integer>而不是 List。上面的代码加上一般性后：

```
// 这个表只放 Strings
List<String> stringList = new ArrayList<String>();
stringList.add( “Hello” );
stringList.add( “World” );

//这个代码会发生编译错误
//stringList.add(new Integer(13));
```

```
for (int i=0; i<stringList.size(); i++) {
    String s = stringList.get(i);
    System.out.println(s);
}
```

本例中，字符串表只能放 String 对象。将 Integer 加进表中时，源代码无法编译。

另外，List<String>的 get()方法返回 String，因此源代码中不必转换为 String。

使用一般性的好处如下：

- 安全性（更严格的编译检查）。
- 代码更可读（知道集合包含何种元素）。
- 更少转换。

说明：使用一般性不要求任何特殊字节码。Java 编译器会生成与旧虚拟机向下兼容的代码。因此，编译类中仍然有转换。

一般性实现方法的缺点是不适用于 int、float 之类基本类型。因此，或者要把基本类型包装成包装类(Integer 或 Float)，或者要实现处理基本类型的集合。

19.2.2 枚举（JSR 201）

本书许多代码中用整数定义枚举，例如：

```
public static final int STATE_IDLE = 0;
public static final int STATE_ACTIVE = 1;
public static final int STATE_DESTROYED = 2;
```

这样虽然行得通，但有问题。以基本 set 方法为例：

```
public void setState(int state) {
    // doesn't check for illegal values!
    this.state = state;
}
```

这里首先没有检查非法值，代码可能将状态设置为 42。也可以检查发给这个方法的值是否合法，只要在枚举改变时维护这个代码。

为了建档，要显式指定这个方法的有效值。

类型安全枚举可以解决这个问题。Java 中的枚举定义与 C 语言中相似：

```
public enum State { IDLE, ACTIVE,DESTROYED }
```

但 Java 中的枚举是对象。setState()方法可以改写如下：

```
public void setState(State state) {
    // null 是惟一非法值
    if (state == null) {
        throw new NullPointerException("Null State");
    }
    this.state = state;
}
```

每个值本身是个实例，因此仍然可以使用==运算符：

```
if (state == State.IDLE) {
    state = State.ACTIVE;
}
```

总之，枚举提供与一般性相似的好处：

- 安全性（没有非法枚举）。
- 代码更可读（定义枚举的代码更少，不需要建档“合法”值）。
- 代码更容易维护。如果在枚举中增加其他值，则不需要对非法值进行改变。

19.2.3 静态导入(JSR 201)

调用静态方法如下：

```
System.out.println(Math.abs(-1));
```

静态导入的基本思路是提供易读的代码。利用新的静态导入特性，可以导入静态类：

```
import static System.*;
import static Math.*;
```

然后直接使用其字段和方法，就像是自己的类成员一样：

```
out.println(abs(-1));
```

这样可以提高可读性。但是，有时静态导入也可能增加混乱。例如，另一个编程人员可能找不到特定字段和方法来自哪个类。一定要在能够提高可读性时才使用这个特性，而不是增加混乱。

19.2.4 改进 for 循环(JSR 201)

JSR 201 中还在数组和集合迭代时改进 for 循环。这个改进使代码更容易编写、更清洁和更可读。

目前，对表进行迭代涉及下列操作（加上一般性）：

```
for (Iterator<String> i = stringList.iterator(); i.hasNext();) {
    String s = i.next();
    System.out.println(s);
}
```

但对于改进 for 循环，可以将循环写成如下：

```
for (String s : stringList) {
    System.out.println(s);
}
```

同样，对任何数组（本例中为 int 数组），可以写成如下：

```
int sum = 0;
for (int value : intArray) {
    sum += value;
}
```

19.2.5 编译器 API（JSR 199）

新的编译器 API 更容易在 Java 应用程序中编译源代码。

编译器 API 不是用命令行调用编译器，然后分析结果，而是让 Java 应用程序直接与 Java 编译器交互，然后取得编译结果的结构化数据。

如果要对游戏脚本进行运行编译，则可以使用这个特性。例如，可以直接在贴图编辑器中编译特定贴图的 Java 源代码。

19.2.6 网络传输格式（JSR 200）

目前，Java 应用程序通常压缩并用 JAR 文件发布。

JAR 文件是个 ZIP 文件。ZIP 文件包含的文件分别压缩，然后合并成一个文件。ZIP 文件中使用的压缩是基于字典的压缩方法。将文件中的常用符号编码成更小的游长位字符串，符号存放在表中。

JAR 文件通常包括几十、几百甚至几千个类文件。但是，分别压缩每个类文件得到的压缩率不如压缩整个类文件高。

例如，假设解压缩 J2SE 1.4.1 所带的 rt.jar 文件。这个文件的长度大约为 22.4MB，但如果是个压缩 JAR 文件，则长度为 11.6MB。使用 tar 与 bzip2 时（常用 Unix 工具），组合和压缩 rt.jar 中的类文件，得到的文件长度大约为 6.2MB.

网络传输格式可以克服 JAR 文件的压缩缺点，用几个技巧压缩类文件。用户可以下载更小的文件，然后在本地机上转换成标准 JAR 文件。

19.2.7 共享虚拟机（JSR 121）

人们迫切希望 Tiger 中推出共享虚拟机。共享虚拟机就是一个虚拟机运行所有 Java 应用程序，而不是对每个应用程序启动新虚拟机。这样可以节省内存，因为只用一个堆，如果虚拟机已经运行，则 Java 应用程序的启动速度也更快。

JSR 121 定义如何在这个共享虚拟机中隔离 Java 应用程序，使其不会相互干扰。

19.3 Java 平台需要什么

Java 平台在游戏编程方面进步很大，特别是 V1.4。但是，还可以增加一些技术和 API，使 Java 平台游戏编程更方便。本节将介绍其中一些技术和 API。

这里介绍的内容几乎都有 Bug Parade ID 或已经在未来版本中考虑，因此不久就可以看到这些特性。

19.3.1 更多鼠标与键盘输入选项

对全屏游戏，AWT 事件体系结构显得太复杂，没有提供典型游戏需要的鼠标与键盘输入。我们不必在每次用户按键或移动鼠标时用另一线程生成 MouseEvent 与 KeyEvent。另外，事件体系结构也可能在用户按键与游戏实际接收这个事件之间产生延迟。

最好轮询键盘上的键击和鼠标的状态。这样就可以在主游戏循环中直接了解键盘上的

键击和鼠标的状态，并相应采取响应。这样还可以减少用户按键与游戏实际响应之间的延迟。

本书提供了实现鼠标观看样式运动的方法，因为目前 API 只提供绝对鼠标位置。但最好能提供相对鼠标位置，使鼠标观看样式运动更壮实。

另外，有些鼠标设备采用强制反馈功能，可以从 Java 游戏中方便地控制。强制反馈指摇动游戏杆、鼠标设备或其他设备，使游戏者对游戏中发生的事件作出反馈。例如，游戏者被妖怪攻击或掉到地上时，鼠标会振动。另外，有些强制反馈游戏杆和鼠标设备还增加阻力，使游戏者在飞行模拟器中控制飞机时，游戏杆更难向前压。

最后，可以发送定制鼠标图像，使鼠标光标隐藏，但这并不是所有平台上都很完善。例如，有些机器上移动鼠标时会发生奇怪的重画行为，即使鼠标是隐藏的。最好能完全关掉光标。

Input polling Bug ID: 4083501

Mouse cursor invisibility Bug ID: 4526950

19.3.2 游戏杆输入

鼠标与键盘输入适合许多游戏者，但许多人喜欢使用游戏杆、游戏板和具有所有按钮的导向轮。需要像轮询键盘上的键击和鼠标的状态一样轮询游戏杆。还要记住强制反馈功能。

Input polling Bug ID: 4083501

19.3.3 硬件加速半透明图像

尽管透明图像在二维游戏中并不重要，但抗齿边图形非常有用，需要让抗齿边图形具有简单半透明度。

目前，要绘制透明图像，就要读取和设置图像中每个媒体，显然很慢。显示卡可以用硬件快速完成这个工作，因此 Java 应提供用硬件绘制透明图像的方法。这一切都不太需要编程人员干预，但要将 API 更新，在 VolatileImage 中支持半透明度。

好在很快就会有硬件加速半透明图形支持了,也许会在 J2SE 1.5 中推出吧。

19.3.4 高精度定时器

前面曾介绍过，Java 使用的定时器精度不高。调用 System.currentTimeMillis()时，在 Windows 95/98/Me 上的精度为 55 毫秒，上的精度为 10 到 15 毫秒，对许多游戏都不够精细。55 毫秒表示定时器每秒只改变 18.2 次，而我们每秒要让游戏运行 30 帧以上。

但是，Linux 与 Mac OS X 的粒度为 1 毫秒。

System.currentTimeMillis()方法要求所有平台上保证粒度为 1 毫秒或要有 1 毫秒或几微秒的高精度定时器 API。

Bug ID: 4478186

19.3.5 Linux 中的硬件加速图形和全屏方式

尽管 Linux 不是家用市场上最著名的操作系统，但仍然要有 Linux 中的硬件加速图形，能够动态改变显示方式和建立全屏游戏。如果不然，则 Linux 上的 Java 游戏赶不上其他操作系统中的 Java 游戏。

另外，硬件加速可以大大加速 Linux 中的 Swing 性能。

据让 J2SE 1.5 中会提供这个特性。

Bug ID: 4498974

19.3.6 运行环境中的硬件加速三维特性

Java3D 目前不在 Java 运行环境中，不是所有平台都支持，特别是 Mac OS X。首先，要让游戏开发人员生成跨平台 Java 游戏，就应该 Java3D 在 Java 运行的所有平台上支持。

其次，运行环境应包括 Java3D API，至少要在运行环境中包括 Java3D 的简化版，因为 Java3D API 中的有些功能不是所有开发人员都需要。另外，除了 Java3D 的简化版，也可以在运行环境中包括简单 OpenGL 关联（如 GL4Java）。

19.3.7 HotSpot SIMD 优化

SIMD（单指令多数据）是个处理器指令集，同时对一组数据执行相同任务。例如，一个指令中不是让处理器将两个数相加，而是将一个数字数组与另一个数字数组相加，如图 19-1 所示。

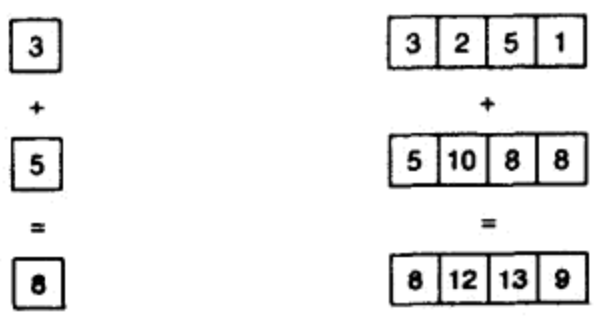


图 19-1 SIMD 同时对一组数据执行相同任务

SIMD 特别适合 MP3 译码、声音处理、图像处理、软件纹理贴图和语音处理等应用程序。

好在目前销售的几乎所有处理器都有某种 SIMD 指令功能：

- AMD 公司 Athlon 具有 3DNow!
- Intels 公司的 Pentium 4 具有 MMX，SSE 与 SSE2。
- Motorola 公司的 PowerPC G4 具有 AltiVec。
- Sun 公司的 UltraSparc 具有 VIS。

过去，开发人员要使用汇编语言以利用这些硬件指令，现在则不然，一些现代 C/C++编译器可以自动进行 SIMD 优化，而不需编程人员的任何输入。

缺点是，如果对某个平台编译 C/C++代码，如 Pentium 4，则其只能在 Pentium 4 以上处理器中运行，而 Pentium 3 或 Athlon 则需要另一个可执行程序。

但 Java 是可移植的，Java 字节码用 HotSpot 编译成自然码时，HotSpot 实际上可以和

C/C++编译器媲美。因此，HotSpot 应该包括自动 SIMD 优化功能。

当然，如果实现这个功能，则 HotSpot 比 C/C++编译器自动 SIMD 优化功能有几个优点：

- HotSpot 可以检查处理器支持的 SIMD 指令（如有），并对这个处理器自动生成 SIMD 代码。
- Java 应用程序不要为每个处理器生成不同可执行程序和代码路径。
- 旧的 Java 应用程序可以利用 SIMD HotSpot 升级，不需要重新编译。

19.3.8 更多字体光顺选项

尽管 Java 平台能够绘制抗齿边字体，但有几个问题：

- Swing 组件（按钮、菜单、清单等）不能抗齿边。
- 抗齿边缺省关闭，即使操作系统打印抗齿边。
- 不支持次媒体字体绘制。

尽管应用程序通常使用 Swing，但第 3 章“交互性与用户界面”介绍了全屏操作系统游戏中如何在用户界面使用 Swing。但是，Swing 组件中的文本不具有字体光顺特性，即使操作系统缺省打开字体光顺特性。因此，Swing 组件的样子不够好看。

就字体光顺特性而言，除了抗齿边，另一个技术是次媒体字体绘制，Microsoft 公司称其为 ClearType。

次媒体字体绘制利用液晶显示器的媒体布局。液晶显示器实际上对屏上每个媒体显示三个瘦长的次媒体：分别为红、绿、蓝成分。图 19-2 显示了一个例子（但这里是黑白的）。红中显示一个白色媒体由红、绿、蓝次媒体构成。我们混合这些次媒体，使这个媒体显示白色。

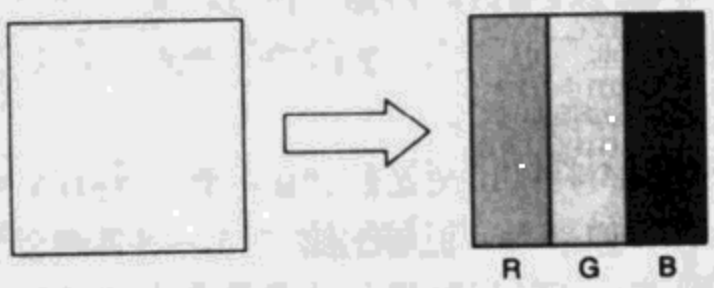


图 19-2 液晶显示器实际上对屏上每个媒体显示三个瘦长的次媒体：分别为红、绿、蓝成分

因此，整个白色显示的次媒体布局如图 19-3 所示。

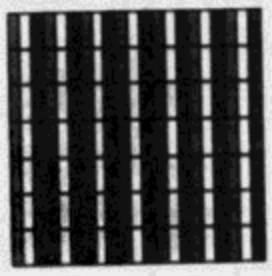


图 19-3 整个白色显示的次媒体布局为红、绿、蓝次媒体交替组合

在典型文本表示中，字体是白色背景上的黑色文本。如果没有字体光顺特性，则每个媒体或者全白，或者全黑。对于抗齿边，字体边沿的媒体可能是不同灰色色档，使文本边

沿更光顺。

次媒体字体绘制使字体光顺特性更进一步，利用次媒体布局，每个媒体不是全关或全关，而是每个次媒体全关或全关，如图 19-4 所示。

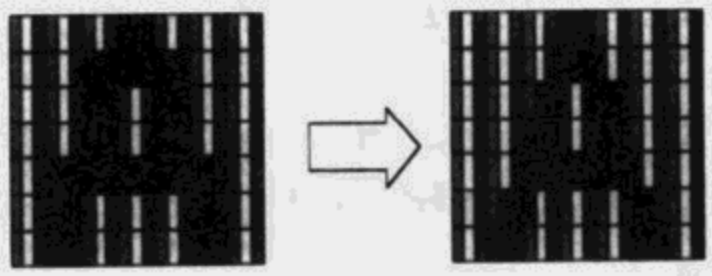


图 19-4　无抗齿边的字体绘制与次媒体字体绘制的字体绘制

例如，如果只要绘制红色和绿色次媒体，则绘制黄色媒体（红+绿=黄）眼睛能把一切混合起来，仍然看到白色背景上的黑色文本。

这只是次媒体字体绘制的基本理论，还可以和抗齿边结合起来，得到更好的结果。

当然，次媒体字体绘制并不是所有情形都很完美。它最适合液晶显示器上的黑白图像（但也对许多 CRT 显示器有用）。另外，它最适合字体，因为字形有许多陡斜坡，但它对缓坡的作用不大。

Sub-pixel Bug ID: 4726365

Anti-aliasing for Swing components Bug ID: 4808567

19.3.9　其他情况

Java 平台还可以通过以下途径更利于游戏：

- Java Sound API 可以更新为支持 OGG（如 NIO 缓冲区）和硬件加速三维声音。
- 缓冲区翻页后，可以关掉垂直追溯同步特性。第 2 章“二维图形与动画”曾介绍过，缓冲区翻页通过垂直追溯同步清除裂开现象。但是，有些系统中的游戏在不需同步时能得到更高的帧速率。
- javac (Bug ID: 4809554)将推出类文件掩盖，但还要有更多保护，防止人们反编译和编辑 Java 类文件。如果人们能够编辑代码，则必然会编辑代码，从而使人们在多方游戏中采用骗术。在多方游戏中采用骗术会使其他游戏者积极性受挫。可能需要某种安全类或安全 JAR 文件支持。

19.4　新设备与 Java Games Profile(JSR 134)

尽管 Windows 没有包括 J2SE 1.4，但计算机厂家开始在销售的机器上包括 J2SE 1.4，这样就把 Java 平台引入了新世界。

另一个激动人心的发展是 Java 平台已经在新设备上出现。Java 已经在手机上普及。Java 2 MicroEdition(J2ME) MIDP 1.0 API 在手机中广为采用，MIDP 2.0 手机已经开始出现。还有将 MIDP 1.0/2.0 引入 Game Boy Advance(JAM 接口设备 www.jamid.com)的技术。Sony 之类控制台游戏系统中也将出现 Java 平台支持。例如，PlayStation 2 市场是个大市场，全球销量超过 5500 万台。

新的 Java Games Profile (JSR 134)是个 J2ME 配置文件，用于游戏控制台（和 PC 机）。这个配置文件中将要包括的一流 Java 技术包括二维与三绘制、流式媒介、物理模型、角色动画、游戏网络和游戏控制器支持。但是，这个配置文件还处于初期，详见 JSR 134。

19.5 小结

本章介绍未来的三个方面：Java 的演变、Java 的未来和 Java 平台需要什么来改进 Java 游戏开发。

当然，未来要靠你我开创。Java 可以创造奇迹，未来需要你的参与，显示你的高超游戏编程技巧。

本书的信息与技术足以建立奇妙的游戏，但要学习的东西还很多。更多的经验要从实践中来，学习游戏编程的最好方法是编一些游戏。要有创造性，要巧妙，利用你的智慧与想像力，建立有趣和畅销的游戏。

记住，可以到本书站点与我联系或下载书中所有源代码与演示程序：

www.brackeen.com/javagamebook。

再见，祝你的游戏好运。

责任编辑：杨如林　安　源
封面设计：梁运丽

本书特点

作者从事游戏编程多年，具有深厚理论基础和实践经验。本书结构清晰，语言简洁明了，代码和实例都注重实用性。深入浅出地介绍了在 Jave 游戏编程中所涉及的众多技术。

本书适用于

游戏开发人员及广大游戏编程爱好者，也可作为高等院校相关专业师生的参考用书。

需要本书或技术支持的读者，请与北京中关村 083 信箱（邮编 100080）发行部联系

市场部电话：010—62520290，82675588	传真：010—62520573
技术支持电话：010—82675588 转 217	E—mail：yanmc@bhp.com.cn
010—62536549（三编室）	投稿：lwm@bhp.com.cn

David Brackeen

生长在德克萨斯洲，获得北德克萨斯洲大学计算机科学专业的学士学位。

曾经使用 Java 创建了许多游戏、关卡编辑器和多媒体产品，包括 Scared（一种三维射击游戏）和 Race3d（一种三维比赛引擎，在多种游戏中使用）。目前住在洛杉机，个人网址为 www.brackeen.com。

ISBN 7-03-012942-3

9 787030 129420 >

ISBN 7—03—012942—3
定价：68.00 元（含 1CD）