Pearson

Broadview
www.broadview.com.cn

游戏编程算法与技巧

[美] Sanjay Madhav 著
刘瀚阳 译

Game Programming Algorithms and Techniques
A Platform-Agnostic Approach

中国工信出版集团

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Game Programming Algorithms and Techniques

A Platform-Agnostic Approach

游戏编程算法与技巧

[美] Sanjay Madhav 著

刘瀚阳 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内容简介

本书介绍了大量今天在游戏行业中用到的算法与技术。本书是为广大熟悉面向对象编程以及基础数据结构的游戏开发者所设计的。作者采用了一种独立于平台框架的方法来展示开发，包括 2D 和 3D 图形学、物理、人工智能、摄像机等多个方面的技术。书中内容几乎兼容所有游戏，无论这些游戏采用何种风格、开发语言和框架。

书中的每个概念都是用 C#、Java 或 C++ 程序员直观明白的伪代码阐述的，并且这些伪代码都已被作者改进和验证过。书中每章末均配有习题或练习，以帮助读者对所学内容进行巩固。本书最后，作者详细分析了两款完整的游戏，清楚展现了前面章节讲到的很多技术和算法。

版权贸易合同登记号　图字：01-2014-4720

图书在版编目（CIP）数据

游戏编程算法与技巧 /（美）马达夫（Madhav,S.）著；刘瀚阳译.—北京：电子工业出版社，2016.10
书名原文：Game Programming Algorithms and Techniques: A Platform-Agnostic Approach
ISBN 978-7-121-27645-3
I. ①游…II. ①马… ②刘…III. ①游戏程序 – 程序设计 IV. ①TP311.5

中国版本图书馆 CIP 数据核字（2015）第 281702 号

策划编辑：张春雨
责任编辑：付　睿
印　　刷：中国电影出版社印刷厂
装　　订：中国电影出版社印刷厂
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　　邮编：100036
开　　本：787×980　1/16　　印张：18.5　　字数：408 千字
版　　次：2016 年 10 月第 1 版
印　　次：2016 年 10 月第 1 次印刷
定　　价：89.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

推荐序

当瀚阳让我为他的译作作序的时候，我还是有些吃惊的。我并非业界名人，虽然在游戏行业摸爬滚打了十几年，但并没有多少建树，为了配得上译者所散发出的光芒，我多少也要发掘出自己的一些亮点。如果资历也算亮点的话，我倒是有些的。1993 年我有了自己的第一台电脑，那时的电脑还处于 386 的时代，硬盘不过 200MB，内存不过 4MB，用着 5 英寸的软驱，光驱还在萌芽状态。当时我一心想做游戏，玩着《Wolf3D》、《Doom》，仰望着 id Soft 的成就。2001 年我终于如愿进入了游戏行业，有幸赶上了西山居《月影传说》的后期开发。之后在三大游戏公司转了一圈，如今沉浮于手机游戏开发的大潮中。其间也曾参与了两款 3D 引擎的早期开发工作，其中的成果已融合在两三款早已上市的游戏当中。我参与程度最深，也算稍微有点名气的游戏就是《斗战神》了。

我和瀚阳相识缘于我们现在共同参与的这个项目，我是项目的第一个客户端程序员，瀚阳是第二个。当项目还只有我一个前台程序员的时候，我一个人要面对十几个美工产出的资源和四五个策划提出的需求，真是压力很大。瀚阳的加入仿佛给项目增加了第二台引擎，各种架构迅速完善起来，功能也越来越完整，项目顺利度过了 Demo 期，得以立项。瀚阳在游戏开发方面涉猎广泛，精通多种语言，做过端游，写过手游，参与过 3D 引擎开发。游戏开发之外瀚阳也有着广泛的兴趣，他的博客就是自己搭建的，跑在租用的服务器上，可以称得上是全栈程序员。游戏行业加班是出了名的，对于瀚阳能在繁忙的工作之余抽出时间来翻译这本书，我是十分佩服的，每一个程序员都不应该局限于自己所在的公司与工作，而应该在更广阔的空间拓展自己的职业生涯。

最后回到本书。如果你想进入游戏开发行业，或者对于游戏开发你已经入门但还缺乏一个全面的了解，本书可以作为不错的入门书籍。本书读起来不会那么累，能够使你快速了解游戏开发所涉及的众多领域，建立一个较为全面的认识。在此之后你可以针对感兴趣的方面购买学习更加深入的书籍。

腾讯互娱前台程序技术指导

陈鹏

译者序

我与本书的一些缘分

我自己是多年的玩家了。在中学时期就暗自下定决心，一定要做游戏。于是不顾反对，大学选择了游戏专业，后来通过一套类似 XNA 的框架进行开发，在学习的过程中一边写游戏，一边扩展框架。从最初的使用 DirectX 绘制 2D 精灵开发俄罗斯方块，到写 3D 骨骼蒙皮动画做 3D 射击游戏，慢慢地学会了很多技术，比如 AI、网络、物理、脚本等。这些知识是“成长”起来的，于是每学到一点新技术都可以往里面塞，逐渐地形成了自己系统的理解。

雕爷有一篇文章说的好，临摹是最好的学习方法。因为临摹不仅需要你拆解分析对象作为输入，还需要以自己的理解作为输出。这种带着主动思考的输入/输出，无形中就对临摹对象有了深刻的理解，这不是走马观花那种被动的学习效率能比拟的。这种经过充分思考的练习，最能够将知识“内化”，而只有站在前人的基础之上，创新才会比较靠谱。在我看来，本书的内容，属于手把手地对各种 GamePlay 拆解，然后实现，让知识一点一点地成长起来。更加难得的是，因为作者从业界转向了教育界，所以本书不仅内容上系统全面，而且语言上简单清晰，让人易于理解。

兴趣是最好的老师，GamePlay 不像操作系统、渲染引擎、物理引擎那么“枯燥”，本书可以很轻松就读完。初学者完全可以从 GamePlay 入手，在 XNA 框架上实现书中的内容，然后在深入的过程中补全游戏开发中各个系统的知识。而有经验者也可以从本书中补全对游戏软件的系统理解，比如一开始作者就阐明了游戏程序的三要素：游戏时间、游戏对象、游戏循环，还有一些有意思的诀窍，比如用 xyzzy 口诀记住叉乘运算，也有一些案例分析——《魔兽世界》的 UI 插件系统，以及工作中的教训——《指环王》中音频引擎的设置。

除此之外，本书还提供了专业方向上的参考资料，让读者在感兴趣的话题上深入学习。每章都会提出关键问题并列出有深度的参考资料，这些资料大部分是行业经典，比如像《游戏引擎架构》、*Real-Time Rendering* 这样的神作当然没有错过。

游戏行业是一个很有意思、充满朝气的行业，有很多真正热爱游戏的人。当然，同时有很多问题，但这何尝不是机会呢？

刘瀚阳
邮箱：lauhonyeung@gmail.com
博客：jjyy.guru

致谢

尽管封面上只有我一个人的名字，但是这本书如果没有那么多人的帮助是不可能出版的。我首先要感谢我的父亲和母亲，他们多年来一直支持我的学习和事业。我还要感谢我的妹妹，小时候我们经常一起经历疯狂又有创意的事情，而她在我长大以后又给我提了很多有用的建议。

在写作的过程中我很荣幸能够与 Pearson 团队一起工作。首先是执行编辑 Laura Lewin，她一直在背后支持这本书。她和她的助手编辑 Olivia Basegio 在我写作的过程中为我提供了很有用的指导。然后是编辑和产品团队，特别是 Chris Zahn，他们让我的书能够面市，包括设计、编辑、排版。

我还想感谢技术指导团队——Alexander Boczar、Dustin Darcy 及 Jeff Wofford，他们在百忙之中抽空确保本书在技术上的准确性。他们的反馈是无价的，而我对本书的信心也来源于此。

还有我的大学里的同事，特别是将我培训成一名导师的信息技术部门的同事。我要特别感谢部门主任 Michael Crowley，以及最早聘我为兼职讲师的主任 Ashish Soni。我还要感谢 Michael Zyda 在南加利福尼亚大学对我的技术领导。我实验室多年来的助手，特别是 Xin Liu 和 Paul Conner 都给了我莫大的帮助。

最后，我还要感谢 Jason Gregory 十多年前对我的辅导。没有他的指导，我就不可能进入游戏行业，更加不会跟随他进入南加利福尼亚大学讲课。他多年来教了我关于游戏开发及游戏教学的很多内容，对此我表示衷心的感谢。

关于作者

Sanjay Madhav 是南加利福尼亚大学的讲师，在那里他教授了几门与游戏编程相关的课程。而在全职加入南加利福尼亚大学之前，他作为程序员在许多公司工作过，包括 Electronic Arts、Neversoft 及 Pandemic Studios。虽然他在很多系统上都有着丰富的开发经验，但是他最感兴趣的还是游戏机制的开发。他所参与的游戏包括《荣誉勋章：太平洋突袭》（*Tony Hawk's Project 8*）、《指环王：征服》（*Lord of the Rings: Conquest*）和 *The Saboteur*。

从 2008 年开始，Sanjay 在南加利福尼亚大学兼职，当时他还是全职游戏程序员当中的一员。在 2009 年年末 Pandemic Studios 关闭之后，他决定专注于教越来越重要的游戏程序员。他的主要课程是面向本科生的游戏编程课程，而且已经连续教了 10 个学期。

前言

不久之前，开发商业游戏的知识还只有少数游戏界精英才知道。在那时候，学习真正用于AAA游戏的算法就像学习一些黑暗禁忌的魔法一样（比如Michael Abrash的*Graphics Programming Black Book*）。如果一个人想要接受游戏编程教育，那么只能去为数不多的培训学校。可是10多年过去之后，游戏编程教育发生了非常大的变化。现在一些顶尖的大学也提供了游戏编程的课程和学位，而且每年加入这个领域的人数也越来越多。

游戏开发课程爆发的结果就是，游戏产业更容易招到需要的人员。21世纪早期的时候，大多数想参与游戏开发的年轻人都拥有着很强的计算机科学背景及创造游戏的热情。这些有着良好基础的年轻人，会在后续工作中不断学习更加高级的游戏编程技巧。今天，有了更多的专注于游戏教育的地方，大量的游戏公司希望他们的年轻程序员可以拥有游戏编程相关的丰富阅历。

为什么需要这么一本游戏编程的书

游戏课程的增长也激发出大学课程设计的需求。可是现在市场上大多数的书都只面向两种读者：希望利用业余时间做点小游戏的爱好者和有着多年经验的专业人员。两种书都会让在校学生感到迷惑。爱好者系列图书不够系统严谨，太过专业的书学生又看不懂。

我在南加利福尼亚大学讲的其中一门课程是ITP 380：游戏编程。课堂上的学生大多数都是大二或者大三的，懂一点编程。有些学生已经能很熟练地运用GameMaker或者Unity开发游戏原型了。但ITP 380是学生们第一门正式的游戏编程课程。我希望本书能够成为这种课程的最佳补充。虽然受众是在校学生，但是其他对游戏编程感兴趣的人也能获得很大价值。

本书的一个特色就是前12章与任何平台和框架无关。这就是说，本书适用于大部分语言和平台。这与市场上其他书籍不一样，那些书往往因为适用某个版本的框架导致几年之后就没用了。正因为这种独立性，本书将会长期适用。这对大学课程设置来说很有帮助，因为他们运用不同的开发框架和语言。这就是说，示例代码总是有价值的。也因此，最后两章会使用两种不同的框架去开发游戏，之前的示例代码会在这些游戏上体现。

谁应该读这本书

本书认为读者已经掌握一门面向对象编程语言（C++、C#、Java），同时也能熟练运用各种数据结构，比如链表、二叉树及哈希表。这些内容通常在计算机科学的前两个学期就会学到，因此对于那些已经完成这些课程的人来说阅读本书是没问题的。更进一步来讲，如果能够掌握微积分，那么后面的线性代数和物理章节读者会更容易理解。

虽然不是必需的，但如果读者提前掌握了基础的游戏策划或者有熟悉的游戏会有不少帮助。同时，本书会专门讨论某种特定的编程机制，所以阅读参考资料会有更多收获。有过GameMaker制作游戏原型经验的话会更好，但仍然不是必需的。

虽然本书意在为学校而写，但是对于已经掌握了通用编程技巧而又对游戏专业感兴趣的人来说，会很有帮助。不像一些学院派的理论书籍，书中的话题总是通过例子形象地展开。

最后，因为本书涵盖非常多话题，对希望掌握游戏编程多个方面的人来说也很有帮助。还有一点需要说明，资深游戏开发者不会在本书中找到太多的最新技术。

本书是怎么组织的

本书的前12章在游戏实战中演示了许多算法与技术。这些话题涵盖2D到3D、物理学、人工智能、摄像机等。虽然设计上第1章到第12章是顺序阅读的，但是有些章节可以直接阅读。图P.1把各章节的依赖关系罗列了出来。所以，最好先阅读依赖章节。

最后两章的游戏示例是为了运用前面12章所学到的算法和技术。两个示例游戏分别是2D滚轴iOS游戏（用Objective-C在Cocos2D上开发）和PC/Mac/Linux平台的3D塔防游戏（用C#在XNA/MonoGame上开发）。两个游戏的源码可以在本书的网站上下载：*http://gamealgorithms.net*。

本书演示风格

这节讲本书的一些演示约定，比如代码和公式。

侧边栏和注意

在本书某些位置你能看到一些“侧边栏”和“注意”版块，像下面这样。

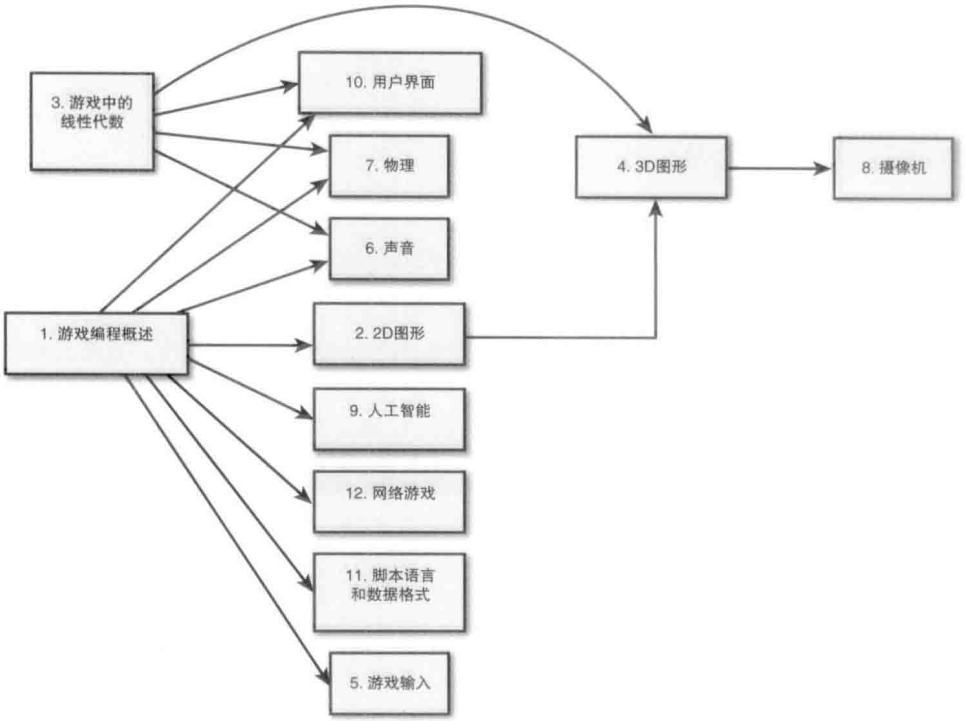


图 P.1　前 12 章的依赖关系

侧边栏

在侧边栏中，会讨论某些运用在某款游戏中特定的算法和技术。有时候会是一些我所开发的游戏中的趣闻轶事，也有时候会是其他游戏的。不管怎样，这里会更加深入地讨论这些在真实游戏开发中会遇到问题的概念。

注意

注意里更多的是不相关的有趣内容。虽然它们不是那么相关，但可能会让我们对话题有更加深刻的认识。

伪代码

为了保持语言的中立性，算法已经以伪代码的形式展现。伪代码的语法与 Lua 比较近似，虽然也有点像 C++ 和 C#。代码展示会像这样：

```
function Update(float deltaTime)
    foreach Object o in world
        // 注释
        o.Update(deltaTime)
    loop
end
```

代码高亮与 IDE 近似。关键词用蓝色，注释用绿色，类名用蓝绿色，变量用斜体。在所有的示例中，访问都是通过点成员方法和变量实现的。

有些情况下，代码要被多次讨论。在这种情况下，展示的不是代码的最终版本。通常最后会将所有代码再展示一次，像清单 P.1 那样。

清单 P.1 示例代码清单

```
function Update(float deltaTime)
    foreach Object o in world
        // 注释
        if o is alive
            o.Update(deltaTime)
        end
    loop
end
```

注意，上面伪代码中简单地检查 Object “is alive”，而不是直接调用某个函数。这样写会更加清晰。

还有就是，一些代码可能会由于篇幅原因省略。通常是由于前面刚演示过，中间有大段重复的部分，我们用省略号替代。

```
function Update(float deltaTime)
    // 更新代码
    ...
end
```

公式

一些章节（主要是线性代数、3D 渲染、物理学）会通过公式来解释。这么做会比使用伪代码更加清晰易懂。公式会居中处理如下：

$$f(x) = a + b$$

本书网站

本书的网站是*http://gamealgorithms.net*，内含第 13 章和第 14 章中游戏的源码及本书的勘误。最后，网站中有一个论坛，读者可以上去提与本书内容或游戏编程相关的问题。

目录

第1章 游戏编程概述

本章先简单地介绍游戏程序员在游戏开发各个时期中所扮演的角色。然后介绍3个任何游戏编程都会用到的重要概念：游戏循环、游戏时间管理、游戏对象。

游戏编程的发展

世界上第一款商业游戏 *Computer Space* 在 1971 年推出。是后来 Atari 的创办人 Nolan Bushnell 和 Ted Dabney 开发的。这款游戏当时并不是运行在传统计算机上的。它的硬件甚至没有处理器和内存。它只是一个简单的状态机，在多个状态中转换。*Computer Space* 的所有逻辑必须在硬件上完成。

随着“Atari 2600”在 1977 年推出，开发者们有了一个开发游戏的平台。这正是游戏开发变得更加软件化的时候，再也不用设计复杂的硬件了。从 Atari 时期一直到现在，仍然有一些游戏技术保留着。大多数的书都不会在这一节展示算法，但接下来我们会看这些仍然保留着的技术。在学习编程之前，对游戏产业有一些认识会更好。

虽然这一节的焦点在于家用机开发，在 PC 机上也仍然适用。在 PC 机上不同之处在于，PC 技术通常都会领先家用机几年。这是因为家用机推出的时候，它的硬件就会被锁定 5 年多，称为一个“世代”，而 PC 机则是以惊人的速度发展着。这就是为什么 PC 上的《孤岛危机》画质可以轻松打败大多数的家用机。可是家用机的优点也在于其锁定了硬件，使得程序员可以有效利用机能。这就是为什么新世代的 *The Last of Us* 与 PC 机上的游戏相比仍然有着惊人的画质表现。

不管如何，家用机自 Atari 在 1977 年推出 Atari 2600 之后并没有大的变化。在此之前，已经有好几种家用游戏系统，但是这些系统都是很受限的。它们的游戏都是只能预装的，而且这些游戏只能在特定平台下玩。在游戏基于卡带后，游戏市场才开始真正地打开。

Atari 时期（1977—1985 年）

虽然 Atari 2600 不是第一个通用的游戏系统，但只有它成功了。不像现在的家用机游戏开发，大多数 Atari 游戏都是一个人负责美术、策划、编程。开发周期也很短，即使最复杂的游戏也只需要几个月。

这个时期的程序员需要对底层硬件有一定的理解。CPU 运行在 1.1MHz，内存只有 128 字节。这些限制使得用 C 语言开发不太实际。大多数游戏都是完全用汇编语言开发的。

更糟糕的是，调试是完全看个人能力的。没有任何开发工具和 SDK（开发代码库）。

虽然有着这些技术限制，Atari 依然很成功。其中一个技术成熟的作品 *Pitfall!*，卖出了 400 万份。这款由 David Crane 在 1982 年设计的游戏，是 Atari 早期的游戏之一，以人类奔跑动画为特色。它被陈列在 GDC 2011 的老游戏面板里，Crane 描述了当时遇到的技能难题。

NES 和 SNES 时期（1985—1995 年）

然而到了 1983 年，北美游戏市场崩溃了。主要原因在于，市场上充斥着大量质量低下的游戏，比如 Atari 版的 *Pac-Man* 及名声不佳的 *E.T.*。

直到 1985 年推出的红白机（NES）才把产业带回正轨。NES 比 Atari 强大得多，这需要更多的“人时”[1]去开发游戏。很多 NES 时期的游戏都需要不少程序员去开发。比如初代《塞尔达传说》需要 3 位著名程序员去开发。

到了超级任天堂（SNES）时代，开发团队进一步扩大。随着开发团队的扩大，不可避免地会变得更加专业化。这使得程序员可以专门为某个游戏的某个功能进行深度开发。比如 1990 年推出的《超级马里奥》就总共有 6 名程序员参与开发。这种专业化包括，一个程序员专门开发马里奥这个主角的功能，而另一个程序员则专门开发关卡。1995 年推出的 *Chrono Trigger* 则更加复杂，总共有 9 名程序员参与开发。他们中的大部分都专门负责做某一块功能。

NES 和 SNES 的游戏仍然完全用汇编语言开发，因为内存依然不足。幸运的是任天堂有提供 SDK（开发代码库）和开发工具，开发者不再像 Atari 时期那么痛苦。

PS 和 PS2 时期（1995—2005 年）

在 20 世纪 90 年代中期推出的 PS 和 N64 终于可以用高级语言去开发游戏了。两大平台的游戏开发语言以 C 语言为主，但仍然有一些性能敏感的功能得用汇编语言完成。

由于高级语言带来了生产力提升，开发团队规模的增长在这个时期刚开始有所减缓。大部分早期游戏仍然只需 8~10 位程序员。即使最复杂的游戏，比如 2001 年推出的 *GTA 3*，工程师团队也是那样的规模。

虽然本时期早期开发团队规模跟 NES 时期差不多，可是到了后期就变庞大了不少。比如 2004 年在 Xbox 推出的 *Full Spectrum Warrior* 总共有 15 名程序员参与开发，大部分都是专门开发某个功能的。但这个增加比起下个时期可以说微不足道。

Xbox360、PS3 和 Wii 时期（2005—2013 年）

家用机的高画质表现导致游戏开发进入了两种境地。AAA 游戏变得更加庞大，也有着相应庞大的团队和费用。而独立游戏则跟过去的小团队开发规模相仿。

对于 AAA 游戏，开发团队的规模增长让人震惊。比如 2008 年推出的 *GTA 4* 仅核心开发者就有大概 30 人，还有额外 15 人来自 Rockstar 的技术团队。近年来团队规模变得更加庞大，《刺

[1]软件工程的一种估算方式。——译者注

客信条：启示录》（*Assassin's Creed: Revelations*）参与开发的程序员总人数达 75 人。

对于独立游戏开发者来说，数字分发平台拯救了他们。随着 XBLA、PSN、Steam 及 App store 等平台的出现，使得游戏不再需要走传统发布渠道。独立游戏的规模远比 AAA 游戏要小，而且在很多方面跟过去的时期差不多。很多独立游戏只要 5 名或更少的开发者。更有甚者，一个人负责编程、美术、策划，完全回到了 Atari 时期。

游戏编程的另一个大的开发趋势就是**中间件**及开源的出现。有的中间件是完整的游戏引擎，比如 Unreal、Unity。有的则是专门地做某个系统，比如物理引擎 Havok Physics。这样就节省了大量的人力、财力。但是缺点就是某个策划功能可能会受到中间件的限制。

游戏的未来

脱离移动和 Web 平台去讨论游戏的未来，是不完整的，因为它们的重要性越来越明显。移动设备的性能提升得飞快，最新的平板已经超越了 Xbox360 和 PS3。这就导致了越来越多的 3D 游戏在移动平台开发。

但这不是说传统的家用机平台就没有进展了。本书编写的时候，任天堂已经开始了 Wii U 项目，而当你读到这里的时候，PS4 和 Xbox One 已经推出。AAA 游戏仍然会扩展它们的团队，团队成员会更加专业。而且 Xbox One 和 PS4 允许自主发行，相当于给独立开发者打开了一扇门。游戏产业的未来真是让人激动和期待。

尽管游戏开发多年来有许多变迁，有趣的是，许多早期概念到现在仍然适用。本章剩余部分，会谈一下那些 20 年都没变的核心概念的基础部分：游戏循环、游戏时间管理和游戏对象模型。

游戏循环

整个游戏程序的核心流程控制称为**游戏循环**。之所以是一个循环，是因为游戏总是不断地执行一系列动作直到玩家退出。每迭代一次游戏循环称为 1 **帧**。大部分实时游戏每秒钟更新 30~60 帧。如果一个游戏跑 60FPS（**帧/秒**），那么这个游戏循环每秒要执行 60 次。

游戏开发中有着各种各样的游戏循环，选择上需要考虑许多因素，其中以硬件因素为主。让我们先讨论传统的游戏循环，然后去了解为现代硬件设计的更高级的技术。

传统的游戏循环

一个传统的游戏循环可以分成 3 部分：处理输入、更新游戏世界、生成输出。一个基本的游戏循环可能是这样的：

```
while game is running
    process inputs
    update game world
    generate outputs
loop
```

比起字面含义，每一个阶段都会有更深层次的含义。例如 **process inputs** 会检查各种输入设备，比如键盘、鼠标、手柄。但是这些输入设备并不只输入已经想到的，任何外部的输入在这一阶段都要处理完成。

举个例子，如果游戏支持多人同时在线。一个重要的输入就是网络数据，因为游戏状态会直接被这些信息所影响。又或者说，支持重新播放的运动类游戏。当要重新看之前玩的录像时，就会将录像数据输入。还有一些移动游戏，输入可能是摄像机、GPS 等。所以会有很多种可能的输入数据，取决于游戏的特殊性及所用到的硬件。

update game world 会执行所有激活并且需要更新的对象。这可能会有成千上万个游戏对象。本章后面会仔细讨论 game object 的表示。

对于 **generate outputs**，最耗费计算量的输出通常就是图形渲染成 2D 或 3D。另外还有其他一些输出，比如音频，涵盖了音效、背景音乐、对话，跟图形输出同样重要。还有就是，家用机手柄通常会有振动效果，随着游戏进程振动。它也是一个需要输出的效果，也叫作**力回馈**。对于多人游戏，游戏中需要输出数据给其他玩家。

本书后面，会详细讨论上面这些游戏循环相关的内容。在此之前，先看一下传统风格的游戏循环在 Namco 的 *Pac-Man* 上是如何应用的。

Pac-Man 的主要输入设备是 4 个方向的摇杆，允许玩家控制 Pac-Man 的移动方向。还有其他输入：投币槽及开始按钮。*Pac-Man* 开始前会进入 Demo 模式吸引玩家。一旦玩家投币，它会请求玩家按开始按钮启动游戏。

进入关卡以后，只有少量对象需要更新主角和 4 个鬼魂。Pac-Man 的位置由摇杆更新。游戏需要检查 Pac-Man 是否靠近鬼魂，靠近的结果就是 Pac-Man 死掉或者杀掉鬼魂，取决于 Pac-Man 是否吃了能力药丸。另外，Pac-Man 能做的事情就是吃途中的药丸和水果，所以更新时也要检查这件事情。由于鬼魂是 AI 操控的，所以还要更新 AI 逻辑。

最后，经典 *Pac-Man* 只输出音频和视频。没有其他网络、力回馈等需要输出。用伪代码描述 *Pac-Man* 逻辑的代码如清单 1.1 所示。

清单 1.1　*Pac-Man* 游戏循环伪代码

```
while player.lives > 0
    // 处理输入
    JoystickData j = grab raw data from joystick
```

```
    // 游戏世界更新
    update player.position based on j
    foreach Ghost g in world
      if player collides with g
          kill either player or g
      else
          update AI for g based on player.position
      end
    loop

    // Pac-Man吃到药丸
    ...

    // 输出结果
    draw graphics
    update audio
loop
```

Pac-Man 还有其他不同的状态，包括之前提到的“吸引模式”，所以这些状态算作全局代码。为简单起见，这里只展示了一个模式下游戏循环的伪代码。

多线程下的游戏循环

虽然很多移动和独立游戏仍然使用传统的游戏循环，但是大部分 AAA 家用机和 PC 机游戏都已经不用了，因为现在 CPU 都有多个核。这意味着 CPU 可以物理上同时执行多指令或者多**线程**。

所有的主流家用机，大部分新 PC，甚至少部分移动设备都支持多核 CPU。为了在这些系统上得到更高的性能，游戏循环需要配合多核工作。有各种方法可以更好地利用多核优势，但大部分都超出了本书的范畴。但是，多线程编程在游戏编程中变得比较流行，所以这里提及一种比较基础的多线程游戏循环技术。

图形渲染对于 AAA 级别游戏来说是非常耗时的。在渲染管线的过程中有多个步骤，而且还有大量的数据需要处理：一些家用机游戏每帧可渲染一百万个多边形。假设对于某个游戏来说，渲染整个场景需要 30 毫秒。它还需要额外的 20 毫秒去更新游戏世界。如果这些都在同一个线程执行，每帧将耗时 50 毫秒，最终导致低帧率——20FPS，这是不可接受的。但如果渲染和更新逻辑同步执行，每帧只要 30ms，30FPS 的目标就可以完成。

为了完成以上想法，主线程必须处理所有输入、更新游戏世界、处理所有图形以外的输出。它必须提交相关数据给第二条线程，那么第二条线程就可以渲染所有图像。

这里有个问题：当渲染线程绘制的时候，主线程该干什么？我们不想它简单地等着渲染结

束，因为这样比单线程还慢。解决的办法是让渲染线程比主线程慢 1 帧。

这个方法的缺点就是会增加**输入延迟**，玩家的输入要更久才能在画面上有所反馈。假设跳跃键在第 2 帧就按下。在多线程游戏循环下，输入直到第 3 帧才开始处理，图形要到第 4 帧结束才能看到，如图 1.1 所示。

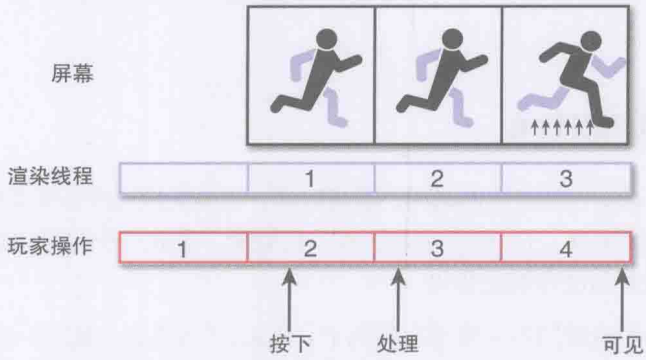


图 1.1　跳跃由于输入延迟的原因被拖后了几帧

如果游戏对于响应速度要求很高，比如类似于 *Street Fighter* 的格斗游戏。长时间的输入延迟是不可接受的。但对于其他类型来说，增加的延迟不会有什么影响。还有其他因素会导致输入延迟。游戏循环只是其中的一种，还有像 LCD 显示器等方面的原因也会导致输入延迟，这个就超出了程序员的控制范围。这个话题的更多信息，可以在本章结尾的参考资料中找到，其中包括有趣的测量和处理游戏输入延迟的方法。

怎么适配多个平台

最初的 Xbox 和 GameCube 都是单核系统，所有这些平台的游戏主要都使用传统游戏循环。可是当 Xbox 360 和 PS3 推出后，传统游戏循环就不能再用了。突然间，习惯于单线程开发的开发者们要面对多核的世界。

大多数游戏工作室早期的尝试都是通过分离渲染和逻辑的方式，就像这一节讲的一样。这个解决方案在早期大多数平台上使用都没问题。

但这种方法的问题在于，没有利用好所有核。Xbox 360 可以同时执行 6 条线程，PS3 可以同时跑 2 条通用线程，6 条专门做数学运算的线程。所以简单地分离渲染和逻辑并没有利用好所有线程。假如机器有 3 条线程，但只有 2 条在用，那么游戏只发挥了 2/3 的机能。

随着时间的推移，开发者开发了新的技术使得完全利用所有核成为可能。这就是 PS3 和 Xbox 360 与上个世代相比进步那么多的部分原因——它们真的完全挖掘了所有潜能。

时间和游戏

大多数游戏都有关于时间进展（progression of time）的概念。对于实时游戏，时间进展通常都很快。比如 30FPS（Frame Per Second）的游戏，每帧大约用时 33ms。即使是回合制游戏也是通过时间进展来完成功能的，只不过它们使用回合来计数。在本节中，考虑一下怎么处理时间比较好。

真实时间和游戏时间

真实时间，就是从真实世界流逝的时间；**游戏时间**，就是从游戏世界流逝的时间。区分它们很重要。虽然通常都是 1：1，但不总是这样。比如说，游戏处于暂停状态。虽然真实时间流逝了，但是游戏时间没有发生变化。

举几个真实时间和游戏时间不一致情况的例子。比如，《马克思·佩恩》运用了“子弹时间”的概念去减慢游戏时间。这种情况下，游戏时间比实际时间要慢得多。与减速相反的例子就是，很多体育类游戏都提升了游戏速度。在足球游戏中，不要求玩家完完全全地经历 15 分钟，而是通常都会让时钟拨快 1 倍。还有一些游戏甚至会有时间倒退的情形，比如《波斯王子：时之沙》就可以让玩家回到之前的时刻。

综合上面的例子，可以看到游戏时间和真实时间大不相同。因此确实需要在游戏循环中考虑游戏时间的流逝。接下来看一下怎么实现这样的需求。

通过处理时间增量来表示游戏逻辑

早期的游戏经常以特定处理器速度来处理逻辑。一个为 8MHz 的处理器开发的游戏，只要处理得当，游戏还是可以运行的。这种情形下，处理敌人位置的代码可能如下：

```
// 更新x位置5个像素
enemy.position.x += 5
```

可是如果敌人的移动速度依赖于 8MHz 的处理器，那么运行在 16MHz 的处理器会怎样呢？可以认为我们的游戏循环会多执行一次，因此敌人的移动速度会快 1 倍。如果要在帧率比过去快上百倍的机器上运行呢？游戏可能会在你眨眼之间就结束。

换句话说，如果之前的敌人移动伪代码每秒执行 30 次（30FPS），敌人在 1 秒就移动 150 个像素。可是在 60FPS 的帧率下，敌人在同样的 1 秒会移动 300 个像素。为了解决这样的问题，需要引入**增量时间**的概念：从上一帧起流逝的时间。

为了应用增量时间，先前的伪代码中的移动不能再使用每帧移动的像素来表示，而是应该使用每秒移动的像素。如果理想的移动速度是每秒 150 像素，那么代码可以改成这样：

```
// 更新x位置150像素/每秒
enemy.position.x += 150 * deltaTime
```

现在代码不管帧率如何都能正常工作。在 30FPS 的帧率下，敌人会每帧移动 5 个像素，总共每秒 150 个像素。在 60FPS 的帧率下，敌人每秒只会移动 2.5 个像素，但总共还是每秒 150 个像素。虽然在 60FPS 下的移动更加平滑，但是总体上每秒的移动速度是一致的。

根据刚才的经验，如果一个对象在游戏中的属性改变需要持续一段时间，则应该以增量时间的形式去处理。这样就可以保证在任何场景下都正常工作，不管是移动、旋转还是缩放。

但你怎么计算每帧的增量时间呢？首先，上一帧所流逝的真实时间是可以得到的。这很大程度上取决于游戏框架，你可以看一下其他游戏是怎么做的。一旦得到流逝的真实时间，我们就可以算出游戏时间。取决于游戏的状态，有可能与真实时间一致也有可能被乘上一个因子。

改进的游戏循环大概像清单 1.2 那样。

清单 1.2 带上增量时间的游戏循环

```
while game is running
   realDeltaTime = time since last frame
   gameDeltaTime = realDeltaTime * gameTimeFactor

   // 进程输入
   ...
   update game world with gameDeltaTime

   // 渲染输出
   ...
loop
```

虽然能够在不同帧率下都正确执行，但还是会有一些问题。首先是任何与物理相关的游戏（比如平台跳跃类）在帧率的表现上都会很不一样。这是数值积分的原因（会在第 7 章“物理”中详细讨论）。这会导致奇怪的角色行为，比如在低帧率下跳得更高。还有就是那些支持多人互动的游戏，可能在不同帧率下导致行为异常。

这个问题有很多解决方法，其中最简单的就是限制帧率，强制游戏循环等待到指定帧率才继续。比如一个目标帧率为 30FPS 的游戏，如果游戏循环本身只用了 30ms，那么还要等待额外的 3.3ms 才能开始执行下一次的游戏循环。清单 1.3 展示了这种做法，但是要记住的是，尽管限制了帧率，还是应该处理增量时间。

清单 1.3 带上帧率限制的游戏循环

```
// 30FPS每帧33.3ms
targetFrameTime = 33.3f
while game is running
   realDeltaTime = time since last frame
   gameDeltaTime = realDeltaTime * gameTimeFactor

   // 处理输入
   ...

   update game world with gameDeltaTime

   // 渲染输出
   ...

   while (time spent this frame) < targetFrameTime
      // 做一些事情将多出的时间用完
      ...
   loop
loop
```

还有一种情况需要考虑：如果游戏突然遇上复杂情形，导致某帧比目标帧率的时长长怎么办？有很多解决办法，最常见的就是为了跟上目标帧率，而丢弃这一帧的渲染。这就是有名的**卡帧**，这么做会引起视觉上的卡顿。你可能会注意到有时候玩游戏的时候，做某些事情就会卡一下（可能是优化太差导致的）。

游戏对象

广义上的**游戏对象**是每一帧都需要更新或者绘制的对象。虽然叫作“游戏对象”，但并不意味着就必须用传统的面向对象。有的游戏采用传统的对象，也有的用组合或者其他复杂的方法。不管如何实现，游戏都需要跟踪管理这些对象，在游戏循环中把它们整合起来。在处理整合游戏对象之前，先看 3 种游戏对象。

游戏对象的类型

在 3 种游戏对象中，最常见的就是更新和绘制都需要的对象。任何角色、生物或者可以移动的物体都需要在游戏循环中的 **update game world** 阶段更新，还要在 **generate outputs** 阶段渲染。在《超级马里奥兄弟》中，任何敌人及所有会动的方块都是这种游戏对象。

只绘制不更新的对象，称为**静态对象**。这些对象就是那些玩家可以看到，但是永远不需要更新的对象。它可以是游戏背景中的建筑。一栋建筑不会移动也不会攻击玩家，但是需要绘制。

第三种游戏对象，就是那些需要更新，但不需要绘制的对象。这么说可能不够直观，一个例子就是摄像机。技术上来讲，你是看不到摄像机的（你可以从摄像机看到外面），但是很多游戏都会移动摄像机。另一个例子就是**触发器**。许多游戏都有这样的设计，当玩家走到某个地点时，就会触发某件事情。比如在恐怖游戏里面，当玩家靠近门口的时候，僵尸突然冒出来。触发器这时就会检测玩家在哪个位置，然后触发合适的行为。所以触发器就是一个看不见的检测与玩家碰撞的盒子。它不应该被绘制出来（除非在调试模式下），否则悬挂在空中只会让玩家感到疑惑。

游戏循环中的游戏对象

为了在游戏循环中使用游戏对象，需要先知道怎么表示它们。之前提到，有很多种方法来表示游戏对象。其中一种就是面向对象中的接口。回想一下，**接口**更像是一种契约。如果一个类实现了这个接口，那么就相当于要按照契约去实现这个接口中所有的函数。

首先，需要有一个基础的游戏对象类，使得 3 种游戏对象都可以从这里继承：

```
class GameObject
   // 成员变量/函数
   ...
end
```

任何游戏对象公用的功能，不管什么对象类型，都应该放在基类里。这样就可以声明两个接口，一个是 Drawable 对象，一个是 Updateable 对象：

```
interface Drawable
   function Draw()
end

interface Updateable
   function Update (float deltaTime)
end
```

一旦有了这两个接口，就可以通过两个接口和一个基类来表示 3 种游戏对象：

```
// 只更新的游戏对象
class UGameObject inherits GameObject, implements Updateable
   // 重载更新函数
   ...
end
```

```
// 只渲染的游戏对象
class DGameObject inherits GameObject, implements Drawable
   // 重载绘制函数
   ...
end

// 更新且绘制的游戏对象
class DUGameObject inherits UGameObject, implements Drawable
   // 重载绘制和更新函数
   ...
end
```

如果采用允许多继承的语言，比如 C++，你可能会直接让 DUGameObject 继承 DGameObject 和 UGameObject。但是这样会让代码变得复杂，因为 DUGameObject 继承了两个对象（DGameObject 和 UGameObject），而这两个对象继承于 GameObject。这种情况称为**钻石问题**[2]，虽然有很多解决方案，但是通常都要尽可能避免，除非有很好的理由这么做。

在实现这 3 种对象后，把它们整合在游戏循环中是很简单的。GameWorld 类拥有两个列表，分别在游戏世界中管理 Updateable 对象和 Drawable 对象：

```
class GameWorld
   List updateableObjects
   List drawableObjects
end
```

游戏对象被创建出来后，必须添加到相应的列表中。相应地，删除游戏对象也要将其从列表移除。在存储了所有游戏对象之后，就可以将游戏循环中的 **update game world** 部分完成，展示在清单 1.4。

清单 1.4 最终的游戏循环

```
while game is running
   realDeltaTime = time since last frame
   gameDeltaTime = realDeltaTime * gameTimeFactor

   // 处理输入
   ...

   // 更新游戏世界
   foreach Updateable o in GameWorld.updateableObjects
      o.Update(gameDeltaTime)
   loop
```

[2]菱形结构。——译者注

```
    // 渲染输出
    foreach Drawable o in GameWorld.drawableObjects
        o.Draw()
    loop

    // 帧数限制代码
    ...
loop
```

这个实现与微软的 XNA 框架相似，这里演示的是一个提炼过的版本。

总结

本章讲了 3 个游戏中至关重要的核心概念。游戏循环决定了游戏对象在游戏世界中每帧是怎么更新的。游戏时间管理则使得我们的游戏速度可以在任何机器上得到保证。最后，一个设计良好的游戏对象模型可以简化游戏世界中的渲染和更新。把它们整合起来，这 3 个概念就是所有实时游戏的基本组件。

习题

1. 为什么早期的游戏使用汇编语言？
2. 什么是中间件？
3. 选择一款经典的街机游戏，讲一下它们在经典游戏循环中每个阶段都做了什么？
4. 在传统游戏循环中，有什么渲染之外的输出？
5. 一个简单的多线程游戏循环是怎么提升帧率的？
6. 什么是输入延迟？多线程游戏循环是怎么导致延迟的？
7. 真实时间和游戏时间有什么不同？什么情况下游戏时间会不同于真实时间？
8. 改变下面依赖于 30FPS 的代码，使其不依赖帧率。

```
position.x += 3.0
position.y += 7.0
```

9. 在传统游戏循环中，怎样强制锁定 30FPS 的帧率？
10. 3 个不同类型的游戏对象是什么？分别给出例子。

相关资料

游戏编程的发展

Crane, David. "GDC 2011 Classic Postmortem on Pitfall!"（*http://tinyurl.com/6kwpfee*）. *Pitfall!* 的作者 David Crane 的 1 小时演讲，关于在 Atari 上开发的心得。

游戏循环

Gregory, Jason. *Game Engine Architecture*. Boca Raton: A K Peters, 2009. 这本书中用了几节篇幅讲了多种多线程下的游戏循环，包括在 PS3 的非对称 CPU 架构上使用的情况。

West, Mick. "Programming Responsiveness" 和 "Measuring Responsiveness"（*http://tinyurl.com/594f6r*和*http://tinyurl.com/5qv5zt*）. 这些是 Mick West 在 Gamasutra 上写的文章，讨论了那些导致输入延迟的因素，同时也对游戏中的输入延迟进行了测量。

游戏对象

Dickheiser, Michael, Ed. *Game Programming Gems 6*. Boston: Charles River Media, 2006. 这卷书中的一篇文章 "Game Object Component System" 讲了一种与传统面向对象游戏对象模型所不同的方法。虽然实现上有点复杂，但是越来越多的商业游戏中的游戏对象通过组合的形式来实现。

第2章
2D图形

随着Web、移动游戏、独立游戏的爆发式增长，2D图形进入了文艺复兴时期。开发者选择2D通常是因为预算和团队规模。玩家选择2D则是因为游戏简洁而纯粹。

虽然本书主要关注3D游戏，但不了解2D的话会很难进入3D。而且在后续章节中，不管是物理、声音，还是UI，在2D和3D下都是通用的。

2D 渲染基础

为了更加全面地认识 2D 渲染，了解这种技术诞生时显示设备的情况是很有必要的。虽然现在基本都用 LCD 或者等离子显示器，但是有很多在老设备上诞生的技术到今天仍然适用。

CRT 显示器基础

在多年前，**阴极射线管（CRT）**显示器是显示器的主流。CRT 里图像的元素就是**像素**。对于彩色显示器，每个颜色由红、绿、蓝组成。显示器的分辨率决定了像素的数量。比如一个 300×200 的显示器有 200 行像素，叫作**扫描线**，每个扫描线可以有 300 个像素，所以总共有 60000 个像素之多。位于 (0,0) 的像素通常在左上角，但不是所有显示器都这样。

CRT 内部，绘制图像是通过电子枪发射电子流完成的。这把枪从左上角开始沿第一条扫描线进行绘制（见图 2.1）。当它完成之后就继续下一条扫描线，然后不断地重复，直到所有扫描线都画完。

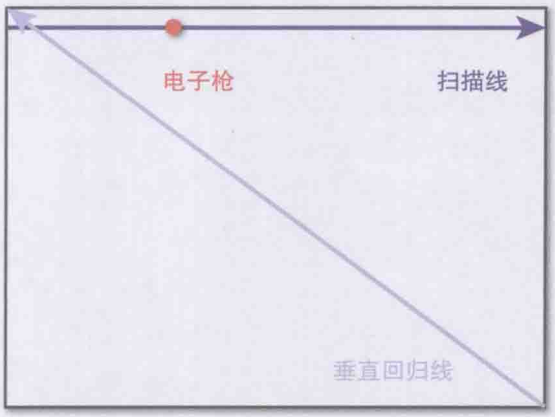


图 2.1 基本的 CRT 绘制

当电子枪刚刚完成一帧的绘制的时候，它的枪头在右下角。喷枪从右下角移动到左上角所花费的时间，我们称为**场消隐期（VBLANK）**。这个间隔以 ms 计，间隔不是由 CRT、计算机或者电视机决定的，而是由用途决定的。

早期像 Atari 这样的游戏硬件，没有足够的内存存储完整的像素数据，这使得渲染系统更复杂。这个话题的更多细节可以看第 1 章提到的 David Crane 在 GDC 的演讲。

像素缓冲区和垂直同步

新的硬件使得有足够的内存将所有颜色保存在**像素缓冲区**中。但这不是说游戏循环就可以完全无视 CRT 喷枪。假设喷枪在屏幕上绘制到一半时，刚好游戏循环到了“generate outputs”阶段。它开始为新的一帧往像素缓冲区写像素时，CRT 还在上一帧的绘制过程中。这就导致了**屏幕撕裂**，具体表现就是屏幕上同时显示了不同的两帧的各自一半画面。像图 2.2 那样。

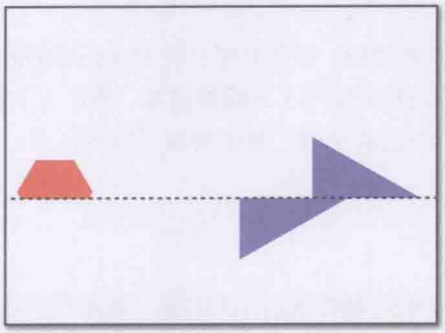


图 2.2　由于 CRT 绘制中提交数据导致的屏幕撕裂

更糟糕的是，新一帧的数据提交时，上一帧还没开始。这就不仅是一帧中有两半不同的画面了，而是连画面都没有。

一个解决方案就是同步游戏循环，等到场消隐期再开始渲染。这样会消除分裂图像的问题，但是它限制了游戏循环的间隔，只有场消隐期期间才能进行渲染，对于现在的游戏来说是不行的。

另一个解决方案叫作**双缓冲技术**。双缓冲技术里，有两块像素缓冲区。游戏交替地绘制在这两块缓冲区里。在 1 帧内，游戏循环可能将颜色写入缓冲区 A，而 CRT 正在显示缓冲区 B。到了下一帧，CRT 显示缓冲区 A，而游戏循环写入缓冲区 B。由于 CRT 和游戏循环都在使用不同的缓冲区，所以没有 CRT 绘制不完整的风险。

为了完全消灭屏幕撕裂，缓冲区交换必须在场消隐期进行。这就是游戏中常见的垂直同步设置。技术上来讲这是不恰当的，因为垂直同步是显示器在场消隐期刚结束时才告诉你的信号。不管怎样，缓冲区交换是一个相对快速的操作，游戏渲染一帧花费的时间则长得多（尽管理想中要比 CRT 绘制一帧快）。所以在场消隐期交换缓冲区完全消除了屏幕撕裂风险。

这就是双缓冲区绘制函数的样子：

```
function RenderWorld()
   // 绘制游戏世界中所有对象
   ...
```

```
    wait for VBLANK
    swap color buffers
end
```

有些游戏确实允许缓冲区交换在绘制完成前尽快进行，这就会导致屏幕撕裂的可能。这种情况通常是因为玩家想要获得远比屏幕刷新速度快的帧率。如果一款显示器有 60Hz 的刷新率，同步缓冲区交换到场消隐期最多只有 60Hz。但是玩家为了减少输入延迟（或者有很快的机器响应速度），可能会消除同步以达到更高的帧率。

虽然 CRT 显示器今天几乎不再使用，但是双缓冲技术在正确的时间交换还是能在 LCD 上消除屏幕撕裂的问题。一些游戏甚至使用了**三缓冲技术**，使用 3 个缓冲区而不是两个。三缓冲区能使帧率在一些特殊情况下更加平滑，但也增加了输入延迟。

精灵

精灵是使用图片中的一个方块绘制而成的 2D 图像。通常精灵用来表示角色和其他动态对象。对于简单的游戏来讲，精灵也可能用于背景，虽然有更加有效的方式来完成，特别是静态的背景。大多数 2D 游戏运用大量的精灵，对于移动端游戏来说，精灵通常就是游戏体积[1]的主要部分。所以，高效利用精灵是非常重要的。

设计精灵时最早决策的部分就是精灵使用哪种图片格式。这在很大程度上取决于用什么图片在哪些平台比较省内存。PNG 格式可能空间占用小，但是通常硬件都不支持以 PNG 格式直接绘制，因此加载到游戏内存的过程中会被转换成其他格式。TGA 格式通常可以直接绘制，可是空间占用比较大。在 iOS 设备上，比较好的格式是 PVR，因为它不仅被压缩过，而且还能够直接绘制。

加载图片到内存的过程也是很大程度上取决于平台和框架的。对于 SDL、XNA 和 Cocos2D 那样的框架，内建了大量图片格式。如果你自己从头开始开发 2D 游戏，可以用一个相对简单的库 stb_image.c（*http://nothings.org/stb_image.c*），它可以加载多种格式，包括 JPEG 和 PNG。因为它是用可移植的 C 语言开发的，所以可以在 C、C++ 和 Objective-C 上运行。

绘制精灵

假设你有一个基本的 2D 场景，上面有着背景图片和在屏幕中间的角色。最简单的绘制场景的方式是，先画背景后画角色。这就像画家在画布上画画一样，也因为这样，这个算法叫作**画家算法**。在画家算法中，所有精灵是从后往前排序的（如图 2.3 所示）。当它绘制场景时，预先排好序的场景可以直接遍历渲染，得到正确的结果。

[1] 磁盘空间占用。——译者注

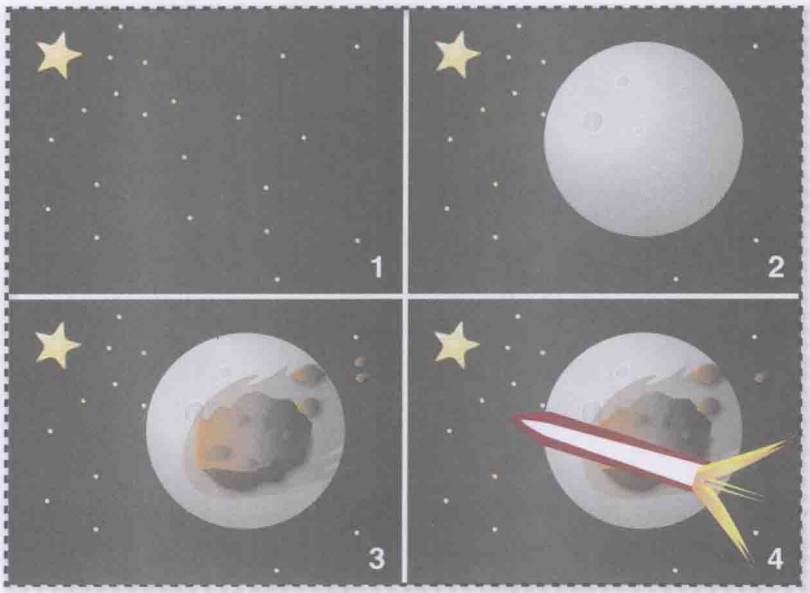


图 2.3　在场景中运用画家算法

这个方法在 2D 游戏中没有什么问题。每个精灵最少有一个绘制顺序，另外还要有图像数据和位置数据。

```
class Sprite
   ImageFile image
   int drawOrder
   int x, y
   function Draw()
      // 把图片在正确的(x,y)上绘制出来
      ...
   end
end
```

然后，游戏世界中的 Drawable 列表就可以根据绘制顺序排序。这样，在绘制期间，排好序的对象就可以线性遍历下去，然后绘制出正确的结果。

```
SortedList spriteList

// 创建新的精灵……
Sprite newSprite = specify image and desired x/y
newSprite.drawOrder = set desired draw order value
// 根据渲染顺序添加到排序列表
spriteList.Add(newSprite.drawOrder, newSprite)
```

```
// 准备绘制
foreach Sprite s in spriteList
   s.Draw()
loop
```

正如将在第 4 章讨论的，画家算法也可以运用在 3D 环境下，但它有很多缺陷。而在 2D 场景中，画家算法工作得很好。

一些 2D 库，比如 Cocos2D，允许场景中的层次任意组合，而每个层次都有一个顺序。比如，游戏可能有一个背景层、有一个角色层，还有 UI 层（这个顺序是从后向前的）。每层可以包含任意数量的精灵。在我们的例子中，所有背景层的精灵会自动放置在角色层精灵的后面，而所有的背景层、角色层精灵又在 UI 层后面。因为采用分层结构，绘制顺序只需相对层内其他精灵就可以了。

精灵的坐标值对应的位置，取决于具体的渲染库。有可能在左上角，也有可能在中间，还有可能在右下角。这个选择是随意的，完全取决于开发者怎么看待这个功能。

动画精灵

对于大多数 2D 游戏，动画原理就跟连环画一样：快速切换静态图片从而产生动画的幻觉（如图 2.4 所示）。为了保证动画的流畅性，帧率最少要达到 24FPS，而电影帧率就是这个帧数。这就是说动画的每 1 秒，都有 24 张不同的图片。有的游戏类型，比如 2D 格斗游戏，可以将动画帧率提高到 60FPS，让图片数量急剧上升。

图 2.4 循环动画

一个常见的方法就是用一组图片去表示一个角色的所有状态，而不是某个特定动画。举个例子，一个有走动和跑步的角色，每个用 10 帧表示，总共用了 20 张图片。为了让问题保持简单，这些图片顺序存储，就是说 0~9 帧表示走路，10~19 帧表示跑步。

但是这就意味着需要一些方法配置哪些帧表示哪个动画。一个简单的方法就是将这些动画信息封装为一个 AnimFrameData 结构体，指定开始帧和帧长度去表示一个动画：

```
struct AnimFrameData
   // 第1帧动画的索引
   int startFrame
   // 动画的所有帧数
   int numFrames
end
```

我们用 AnimData 结构体去存储所有图片的同时，用 FrameData 保存所有动画信息：

```
struct AnimData
   // 所有动画用到的图片
   ImageFile images[]
   // 所有动画用到的帧
   AnimFrameData frameInfo[]
end
```

然后需要 AnimatedSprite 类从 Sprite 类继承下来。因为它继承了 Sprite，它已经有位置和根据绘制顺序进行绘制的功能，简单来讲就是绘制一张图片的能力。但是，由于 AnimatedSprite 比 Sprite 复杂得多，还需要一些额外的变量来完成功能。

AnimatedSprite 要能够跟踪当前的动画数量，知道当前帧属于哪一个动画及当前动画需要用到多长时间。你可能会注意到 FPS 也作为一个成员变量被存储了。这样能够让动画动态加速或者减速。比如说，一个角色获得了加速效果，可能需要让角色跑动得更快一点。

```
class AnimatedSprite inherits Sprite
   // 所有动画数据（包括ImageFiles和FrameData）
   AnimData animData
   // 当前运行中的动画
   int animNum
   // 当前运行中的动画的帧数
   int frameNum
   // 当前帧播放了多长时间
   float frameTime
   // 动画的FPS（默认24FPS）
   float animFPS = 24.0f

   function Initialize(AnimData myData, int startingAnimNum)
   function UpdateAnim(float deltaTime)
   function ChangeAnim(int num)
end
```

Initialize 函数会为这个 AnimatedSprite 引用 AnimData。通过引用传递，多个动画精灵能共享同一份数据，这样大大节省了内存。然后函数要求传入需要播放的动画，而后续初始化工作

由 ChangeAnim 函数完成。

```
function AnimatedSprite.Initialize(AnimData myData, int startingAnimNum)
   animData = myData
   ChangeAnim(startingAnimNum)
end
```

ChangeAnim 函数在 AnimatedSprite 切换动画的时候调用。它设置帧数和时间都为 0，而后设置当前图片为动画的第 1 帧。在 ChangeAnim 和 UpdateAnim 函数中，使用 image 去表示图片。这是因为基类 Sprite 使用 image 绘制。

```
function AnimatedSprite.ChangeAnim(int num)
   animNum = num
   // 当前动画为第0帧的0.0f时间
   frameNum = 0
   animTime = 0.0f
   // 设置当前图像，设置为startFrame
   int imageNum = animData.frameInfo[animNum].startFrame
   image = animData.images[imageNum]
end
```

UpdateAnim 函数是 AnimatedSprite 最重要的函数。一部分复杂的原因是不能假设动画帧率比游戏帧率慢。比如说，一个游戏可以以 30FPS 的帧率运行，但我们想让动画以 24FPS 的两倍的帧率运行，也就是 48FPS。这就是说 UpdateAnim 得跳过某些帧。从这个例子来看，某一帧比某一动画帧所花费的时间要长，那么就要计算到底要跳过多少帧。这也就是说，如果动画是循环动画，那么不能直接设置到第 0 帧。有可能动画太快，需要设置到第 1 或第 2 帧。

```
function AnimatedSprite.UpdateAnim(float deltaTime)
   // 更新当前帧播放时间
   frameTime += deltaTime

   // 根据frameTime判断是否播放下一帧
   if frameTime > (1 / animFPS)
      // 更新当前播放到第几帧
      // frameTime / (1 / animFPS)就相当于frameTime * animFPS
      frameNum += frameTime * animFPS

      // 检查是否跳过最后一帧
      if frameNum >= animData.frameInfo[animNum].numFrames
         // 取模能保证帧数循环正确
         // （比如，If numFrames == 10 and frameNum == 11,
         // frameNum会得到11 % 10 = 1）。
         frameNum = frameNum % animData.frameInfo[animNum].numFrames
      end
```

```
        // 更新当前显示图片
        // （startFrame是相对于所有图片来决定的，而frameNum是相对于某个动画
        // 来决定的）
        int imageNum = animData.frameInfo[animNum].startFrame + frameNum
        image = animData.images[imageNum]

        // 我们用fmod（浮点数运算），相当于取模运算
        frameTime = fmod(frameTime, 1 / animFPS)
    end
end
```

虽然这个 AnimatedSprite 实现在循环动画下工作正常，但还不支持动画切换。如果需要这样的功能，还是建议用动画状态机去实现会更好。第 9 章“人工智能”会讨论状态机设计模式在 AI 中的应用，状态机模式很容易应用在动画状态机中。

精灵表单

为了保证精灵完全对齐，让角色的所有动画用同一个尺寸是尚可接受的。在过去，许多库还要求所有图片尺寸为 2 的幂次方，也就是说 30×30 的图片需要对齐到 32×32。虽然今天很多显卡都不再要求 2 的幂次方了，但是在 mipmap 中还是适用的（超出了本书的范围）。

你当然也可以让每帧图片都单独成为一张图片文件（或**纹理**）。虽然这是个简单的系统，但它通常都会耗费大量资源。这是因为虽然图片是长方形的，但是精灵却不一定是。如果一帧图片大小为 100KB，有 15% 的无用空间，那就意味着有 15KB 是浪费的。如果动画有很多帧，那么浪费就会大大增加。哪怕总共只有 100 帧（大概总共 5 秒的动画）也造成了 1.5MB 的浪费。

一个解决方案就是使用单张图片去存储所有精灵，称之为**精灵表单**。在精灵表单中，可以让精灵打包尽可能地靠近，从而减少浪费的无用空间。但是如果使用精灵表单，那么需要额外做一些工作达到节省内存的目的。而这么做能节省出可观的内存空间。图 2.5 演示了怎么用精灵表单来节省空间。

有个流行的精灵表单打包工具叫作 TexturePacker（*www.texturepacker.com*）。它支持很多 2D 库格式，包括 Cocos2D。TexturePacker 的 Pro 版本还有额外的节省空间的技术，比如旋转精灵使得打包更加紧凑，还有抖动使图片中颜色的数量减少。

精灵表单的另一个优势就是很多 GPU 要纹理加载后才能绘制。如果绘制过程中频繁地切换纹理，会造成相当多的性能损耗，特别是对于大一点的精灵。但是如果所有精灵都在一张纹理中，是可以消除切换产生的损耗的。

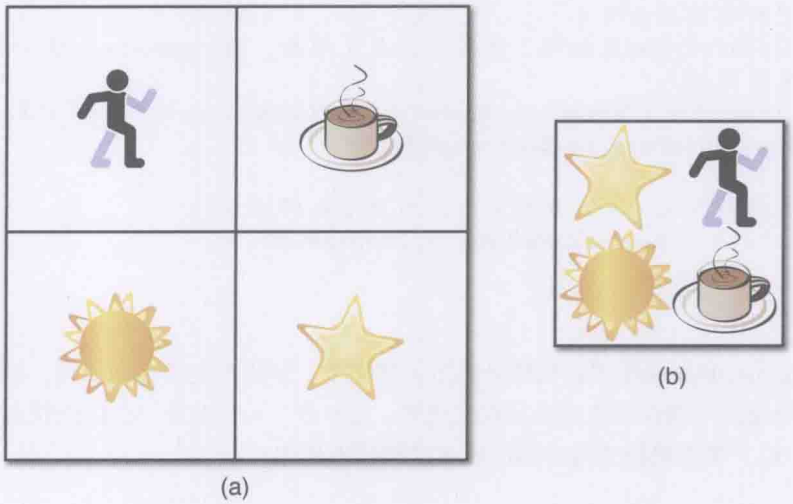


图 2.5 单独的精灵 (a) 和打包的精灵表单 (b)

取决于游戏中精灵的数量，把所有精灵都放入一张纹理是不现实的。大多数硬件都有纹理最大尺寸限制。比如 iOS 设备的纹理最大尺寸为 2048×2048。因此如果 iOS 游戏的精灵放不进一张尺寸为 2048×2048 的表单，那么就需要多张精灵表单。

滚屏

在相对简单的 2D 游戏里，比如 *Pac-Man* 和《俄罗斯方块》，所有元素都在一个屏幕里。在更复杂的 2D 游戏里，游戏世界经常比单个屏幕大。对于这些游戏，一个更常见的方法就是关卡随着角色移动而滚动。在本节，会讨论几种比较复杂的滚屏。

单轴滚屏

在**单轴滚屏**游戏中，游戏只沿 *x* 轴或者 *y* 轴滚动。无限滚轴游戏，比如图 2.6 中的 *Jetpack Joyride* 就属于这种滚轴。至于如何实现，主要取决于关卡大小。对于那些所有纹理一次加载进内存的游戏，就跟本节讨论的游戏一样，算法不会太复杂。

最简单的方法就是把关卡背景按屏幕大小进行切割。一关可能有 20~30 个片段，绝对比单张图片大得多。加载的时候，图片可以以片段为单位放置在游戏世界中。如果背景精灵从左上角开始画，那么每个片段的坐标计算也相当容易。在每个背景放到正确位置后，可以将其加载到一张链表中。水平滚动的初始化代码可以像下面这样：

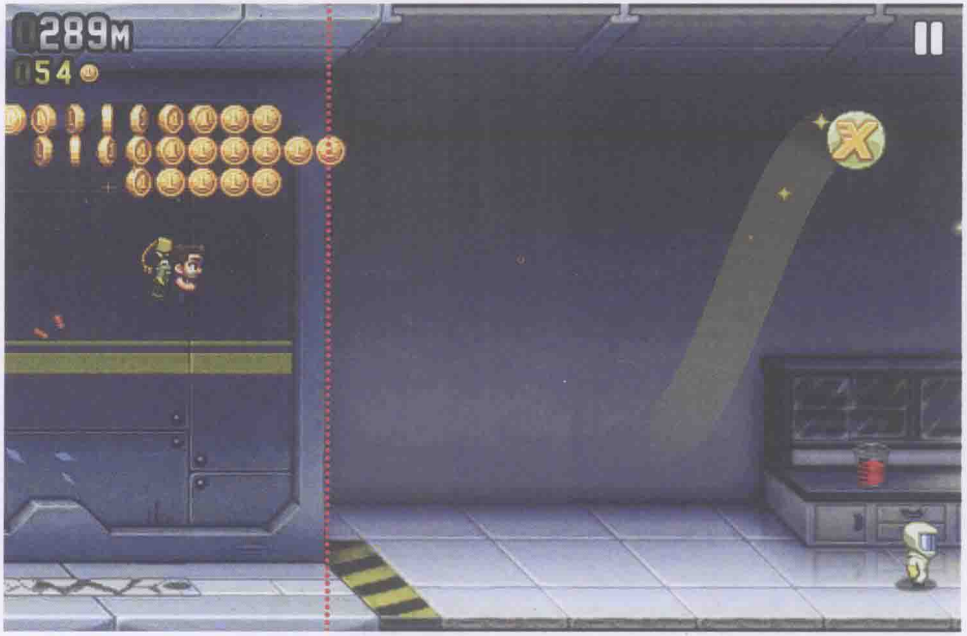


图 2.6　单轴滚动的 *Jetpack Joyride*，虚线画出了图片边界

```
const int screenWidth = 960 // 一台iPhone 4/4S屏幕大小为960×640
// 所有屏幕大小的背景图
string backgrounds[] = { "bg1.png", "bg2.png", /*...*/}
// 所有水平摆放的屏幕大小的背景图数量
int hCount = 0
foreach string s in backgrounds
   Sprite bgSprite
   bgSprite.image.Load(s)
   // 第1个屏幕在x=0处，第2个在x=960处，第3个在x=1920处……
   bgSprite.x = hCount * screenWidth
   bgSprite.y = 0
   bgSpriteList.Add(bgSprite)
   screenCount++
loop
```

在 bgSpriteList 装载完毕后，需要决定哪些背景需要绘制及绘制在什么地方。如果背景片段屏幕尺寸相同，那么同时最多只有两张背景需要绘制。需要一种方法去跟踪什么在屏幕上并显示正确的背景。

一个常见的方法就是让摄像机也在游戏世界中拥有坐标。摄像机最开始放在第一张背景的位置。在水平滚屏的过程中，摄像机的 x 位置设置为玩家的 x 位置，只要位置不超过第一张背景和最后一张背景的范围就没问题。

虽然这样可以通过 if 判断各种情况，但是随着这样的判断越来越多，游戏会变得很复杂。一个不错的方案就是使用 **clamp** 函数，它能让某个值在最大值和最小值之间。在这种情况下，可以设置摄像机 *x* 位置等于玩家 *x* 位置，然后把 *x* 值维护在最大值和最小值之间。

```
// camera.x就是player.x在区间中经过clamp的值
camera.x = clamp(player.x, screenWidth / 2,
                           hCount * screenWidth - screenWidth / 2)

Iterator i = bgSpriteList.begin()
while i != bgSpriteList.end()
   Sprite s = i.value()
   // 找到第1张图片来绘制
   if (camera.x - s.x) < screenWidth
      // 第1张图：s.x = 0, camera.x = 480, screenWidth/2 = 480
      // 0 - 480 + 480 = 0
      draw s at (s.x - camera.x + screenWidth/2, 0)
      // 绘制第1张背景图后，找第2张
      i++
      s = i.value()
      draw s at (s.x - camera.x + screenWidth/2, 0)
      break
   end
   i++
loop
```

上面的代码实现了摄像机和玩家在关卡中的前进和后退，就像《超级马里奥世界》一样。如果希望给摄像机加点限制，比如只能朝前移动，只需在玩家 *x* 位置比摄像机 *x* 位置大的时候更新即可。

值得注意的是，在使用像 Cocos2D 那样的支持层级的引擎时，可以通过使背景在需要的时候改变位置来简化代码。第 13 章的示例游戏中有所应用。但是对于不支持层级功能的框架来说，你需要像上面那样自己编写相应的代码。

无限滚屏

无限滚屏就是当玩家失败才停止滚屏的游戏。当然，这里不可能有无限多个背景来滚屏。因此游戏中的背景会重复出现。在第 13 章的示例游戏中，只有两个星云图片无限循环。当然，大部分无限滚屏游戏都拥有大量的背景图片和很好的随机性来产生丰富的背景。通常游戏都会用连续的四五张图片组成序列，有了不同系列的图片可选后再打乱重组。

平行滚屏

在平行滚屏中，背景拆分成几个不同深度的层级。每一层都用不同的速度来滚动以制造不同深度的假象。一个例子就是游戏中的云朵和地面。如果云朵比地面滚动得慢，就会造成云朵比地面远的感觉。

这种技术在传统动画中运用了几个世纪，但是第一个使用平行滚轴的游戏是 1982 年的街机游戏 *Moon Patrol*。在智能手机商店中，大量的高质量 2D 游戏都使用了某种形式上的平行滚轴技术，比如 *Jetpack Joyride* 和《愤怒的小鸟》。通常打造不错的平行效果，只需要 3 层，如图 2.7 所示。但是更多的层级可以增加更多深度的感觉。

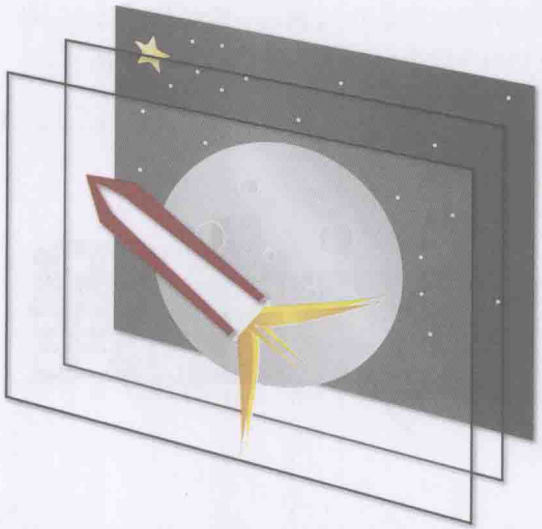


图 2.7　把太空场景分成 3 层完成平行滚屏

单轴滚屏中的伪代码可以替换成多个背景同时滚屏。为了实现平行滚屏效果，当计算精灵相对于摄像机偏移时，需要每层都给出一个额外的因子。如果一个比较靠近的层级以 1/5 的速度移动，偏移计算的代码可以这么做：

```
float speedFactor = 0.2f
draw s at (s.x - (camera.x - screenWidth/2) * speedFactor, 0)
```

对于无限滚屏，背景的层数可能会比一般的要少。云层可能总共只有 10 个在不断重复，这样更能够造成前面提到的随机化。当然，支持层级的框架也能很好地运用平行滚屏功能。

更多平行滚屏的细节，可以在第 13 章的 2D 游戏示例中看到。它实现了无限滚屏和多个背景层共同完成平行滚轴。

四向滚屏

在四向滚屏中，游戏世界会在水平和垂直方向上滚动，大部分《超级马里奥兄弟 2》之后的马里奥游戏都这样。由于两个方向上都滚屏，同一时刻屏幕会显示 4 个背景片段。

为了让之前的水平滚屏代码在两个方向上都能动，需要检查摄像机的 *y* 位置是否需要更新。但在此之前，需要声明一些变量，记录屏幕高度和关卡垂直方向上有多少个片段：

```
const int screenHeight = 640 // iPhone 4/4S的水平方向上的高度
int vCount = 2
```

一个四向滚屏游戏会遇到的问题是原点的放置。因为大多数 2D 框架坐标系统的原点在左上角，如果左上角的游戏世界坐标就是 (0,0) 的话，会让情况简单一些。所以我们的游戏支持两个垂直片段，(0,0) 在初始片段上面一个片段上，如图 2.8 所示。

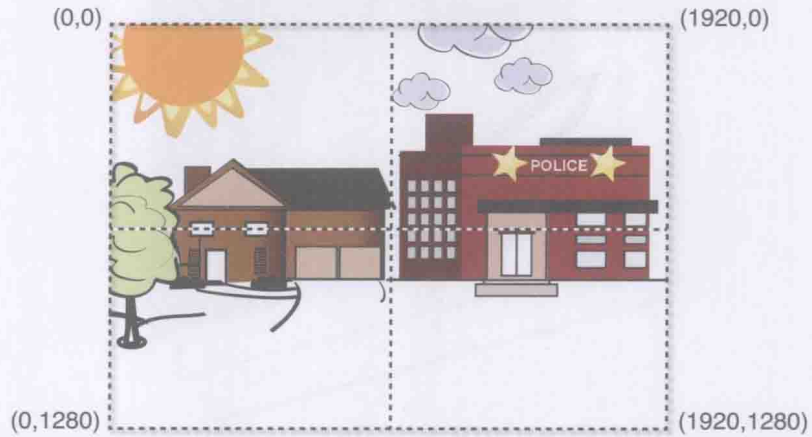


图 2.8 场景分割成 4 块来支持四向滚屏

由于可以垂直滚屏，故需要更新摄像机的 *y* 位置：

```
camera.y = clamp(player.y, screenHeight / 2,
                           vCount * screenHeight - screenHeight / 2)
```

计算精灵相对于摄像机的新位置要考虑 *y* 位置：

```
draw s at (s.x - camera.x + screenWidth / 2,
           s.y - camera.y - screenHeight / 2)
```

要判定哪些背景片段需要绘制，不能再用水平滚屏的方式去做。那是因为不再是更新单个精灵列表。一个基本的方法就是使用二维数组记录片段，然后判断哪一行哪一列需要在屏幕上显示。一旦片段计算出来，计算其他 3 个也很简单：

```
// 这里假设用二维数据array[row][column]记录所有片段
for int i = 0, i < vCount, i++
   // 这是正确的行吗?
   if (camera.y - segments[i][0].y) < screenHeight
      for int j = 0, j < hCount, j++
         // 这是正确的列吗?
         if (camera.x - segments[i][j].x) < screenWidth
            // 这是左上角的可见片段
         end
      loop
   end
loop
```

砖块地图

如果你正在开发就像《塞尔达传说》那样的2D俯视角色扮演动作游戏。会有着大量的野外区域和地下城，你需要一种方式去绘制这些关卡。一个简单粗暴的方法就是每个场景都用一张图片展现。但是这样就会产生大量的重复，比如石头和树木就会在一个场景中重复多次。更糟糕的是，石头和树木在多个场景中有着更大量的重复。这就意味着直接使用图片的方式会造成大量的内存浪费。

砖块地图通过把游戏世界分割成等分的方块(或者其他多边形)来解决这个问题。每个方块代表的精灵占据着某一块网格位置。这些引用的精灵可以放在多张或者一张**砖块集合**里。所以如果树木在砖块集合中的索引号为0，每个表示树木的方块都可以用0表示。如图2.9所示演示了运用砖块地图的场景。

虽然正方形是砖块地图的最常见形式，但这不是必需的。一些游戏采用六边形，有的则采用平行四边形。这主要取决于你希望的视角。

不管什么情况，砖块地图是一种很好的节省内存的方式，也让策划和美工更容易工作。那些动态创建内容的2D游戏，比如*Spelunky*，没有砖块地图将很难实现。

简单的砖块地图

有不少游戏框架，比如Cocos2D，都内建支持免费砖块地图程序Tiled（*www.mapeditor.com*）创建的砖块地图。这是个很好用的程序，如果你的框架支持它，那么我非常推荐你使用这个程序。但是，如果你的框架不支持Tiled，实现砖块地图也不是很困难的事情。特别是那些每关都能在一个屏幕显示完全、不需要滚屏的游戏。

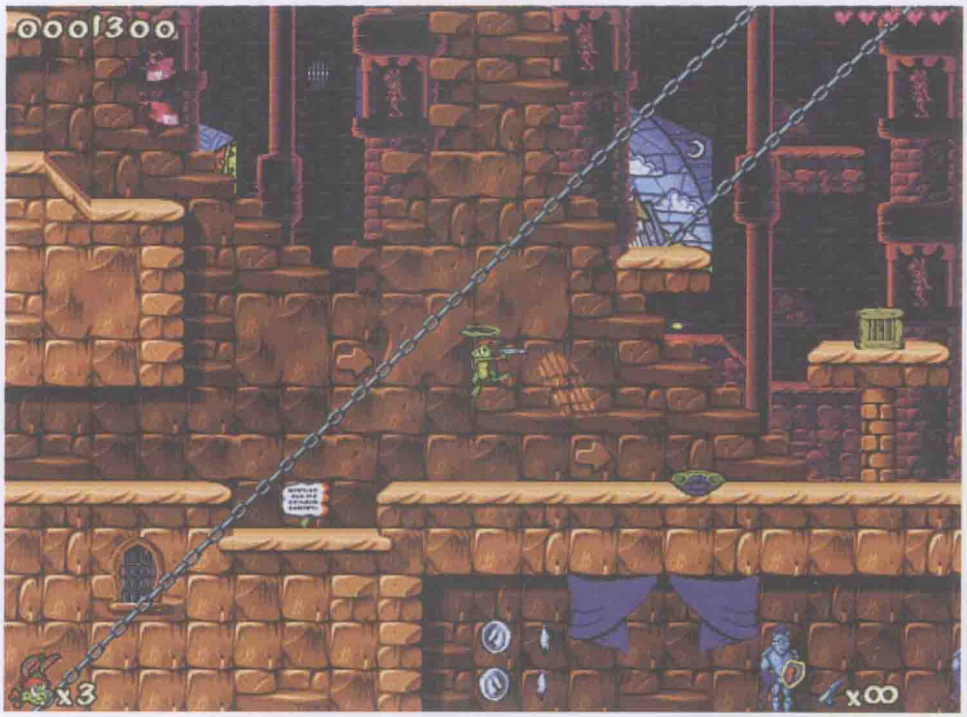


图 2.9 经典的平台游戏 *Jazz Jackrabbit 2* 用砖块地图拼接关卡

首先要确定砖块的尺寸。虽然这在很大程度上取决于设备，但是很多智能手机上的游戏都采用 32×32 像素。这就意味着在分辨率 960×640 的视网膜屏幕的 iPhone 手机上，单个屏幕能显示 30×20 个砖块。

设置好砖块的尺寸后，下一步就是创建游戏使用的砖块。所有砖块都放进一张精灵表单中。渲染程序需要通过砖块 ID 得到真正的图片。一个简单的方法是让左上角的砖块 ID 为 0，往右就加 1，以此类推。

在设计好砖块集合之后，就可以利用砖块集合设计关卡数据了。虽然关卡数据可以硬编码为二维数组，但是存储在外部文件会更好一些。这个文件最常见的格式就是按照屏幕显示的方式，用砖块 ID 去表示。因此 5×5 的砖块关卡会像这样：

```
// 基本关卡文件格式
5,5
0,0,1,0,0
0,1,1,1,0
1,1,2,1,1
0,1,1,1,0
0,0,1,0,0
```

在上面的文件中，忽略第 1 行，那样我们就可以写注释。第 2 行记录了有多少行多少列。然后紧接着就是表示屏幕上的砖块 ID。所以这一关的第 1 行有 4 个 ID 为 0 的砖块[2]，1 个 ID 为 1 的砖块，第 2 行又多了两个 ID 为 1 的砖块。

关卡可以用下面的类表示：

```
class Level
   const int tileSize = 32
   int width, height
   int tiles[][]
   function Draw()
end
```

tiles 数组存储了关卡的 ID，而宽高对应了数组的行列。虽然读取关卡数据的代码是由具体的程序语言决定的，但是解析这个文件非常简单。在完成解析程序之后，下一步就是实现关卡的绘制函数：

```
function Draw()
   for int row = 0, row < height, row++
      for int col = 0, col < width, col++
         // 在(col*tileSize, row*tileSize)绘制tiles[row][col]
      loop
   loop
end
```

虽然怎么绘制纹理也是由特定平台决定的，但是大多数框架都支持在屏幕某个位置上绘制图片中的某一块纹理。因为砖块 ID 转换为图片中某一块纹理，所以绘制也是比较简单的。

值得注意的是，可以为同一个砖块 ID 使用不同图片。这就可以实现游戏中常见的季节性的“皮肤”功能。比如在冬天，可以把某些块换成雪块。它虽然是同一关，但是看上去是在不同节日。

虽然这种基于文本的砖块地图在简单关卡中运行顺利，但是在商业游戏中，会采用更加健壮的格式。在第 11 章中，将会讨论其他常见的文件格式，有些格式能把砖块地图用得更好。

斜视等视角砖块地图

关于砖块地图，一直讨论的都在平面上。虽然能在顶视角中顺利运行，但若希望视觉更具有深度，则需要用不同的视角去观察场景。在**斜视等视角**中，视角通过旋转，让场景更具深度感。《暗黑破坏神》和《辐射》都应用了这种视角，图 2.10 就是这种例子。

[2]原文是 2 个 ID 为 0 的砖块。——译者注

图 2.10 斜视等视角场景的例子

实现斜视等视角与普通视角有一些不同。与正方形不同的是，这些砖块要么是六边形要么是平行四边形。运用斜视等视角砖块地图的时候，使用多个层次去组织砖块是很常见的，这些层次会把相邻的砖块作为一组。为了支持多层次，需要更新数据，使其能够表达多个层次。更进一步讲，我们的渲染代码也需要支持正确的顺序。

总结

自游戏产业创立以来，2D 图形游戏已经随处可见，而本章则讨论了许多相关的核心技术。虽然这些技术是在 CRT 显示屏时期发明的，但是它们在现代游戏中还在使用。即使到了今天，我们还在用双缓冲技术去避免屏幕撕裂。尽管精灵的分辨率不断提升，但它仍然是 2D 游戏的基础组件，不管它带不带动画。最后，讨论了滚屏技术和砖块地图技术，知道了这些稍微复杂的系统在现代游戏中是如何实现的。

习题

1. CRT 中电子枪是如何绘图的？什么是场消隐期？
2. 什么是屏幕撕裂，以及如何避免？

3. 使用双缓冲技术与使用单个颜色缓冲技术相比，有什么优势？
4. 什么是画家算法？
5. 精灵表单与单个精灵文件相比有什么优势？
6. 摄像机位置在单方向滚屏中的更新时机如何判断？
7. 在双向滚屏游戏中，如果所有背景无法一次载入内存，该用什么数据结构去跟踪这些片段？
8. 说一下砖块地图和砖块集合的区别？
9. 对于动画精灵来说，把 FPS 作为成员变量有什么好处？
10. 斜视等视角与顶视角有什么不同？

相关资料

Cocos2D

iOS 游戏开发最流行的 2D 代码库，在*www.cocos2d-iphone.org*可以获取。

Itterheim, Stephen. ***Learn Cocos2d 2: Game Development for iOS.*** **New York: Apress, 2012.** 虽然有很多本书讲 Cocos2D，但这本是相对强的。

SDL

Simple DirectMedia Layer（SDL）是另一种可供选择的代码库。这是用 C 语言开发的流行的跨平台 2D 游戏代码库，同时也可移植到其他语言。SDL 可以在*www.libsdl.org*获取。

第3章

游戏中的线性代数

线性代数是数学中的一个很广泛的分支，在很多专业书籍中都有阐述。对于 3D 游戏来说，其中只有很小的一部分需要使用。本章更加关注游戏开发实战中用得上的部分，而并未涵盖所有线性代数内容和证明。

向量和矩阵在游戏程序员工具箱中有很重要的位置，很多业内的工程师每天都在跟它们打交道。

向量

向量表示了 n 维空间下的长度和方向，每个维度都用一个实数去表示。在游戏中，向量通常要么是 2D 的要么是 3D 的，这个由游戏需求决定。但是在第 4 章会大量用到 4D 向量，尽管游戏本身是 3D 的。

向量在论文中有很多种表示，本书的表示方法为在字母上方加一个箭头。向量的分量都有对应的下标。3D 向量 $\vec{v}$ 可以表示为如下形式：

$$\vec{v} = \langle v_x, v_y, v_z \rangle$$

所以 $\langle 1,2,3 \rangle$ 就是 x 分量为 1，y 分量为 2，z 分量为 3。

在代码中，向量通常用一个类表示，其中每个分量作为一个浮点型成员变量：

```
class Vector3
   float x
   float y
   float z
end
```

向量没有位置的概念。就是说如果两个向量相等的话，那么不仅长度一样，而且方向也一样。图 3.1 展示了包含多个向量的向量场。场内的所有向量都有同样的长度和方向，因此它们都是相等的。

图 3.1　向量场中的向量都是相等的

这种相等使得向量不管在哪里绘制都可以，这种性质是很有价值的。当解决向量问题的时候，你会发现在不同位置都可以绘制向量是非常有帮助的。因为改变向量绘制的位置并不会改变向量本身，这个常用的技巧要铭记在心。

虽然向量在哪里画出来不改变向量本身，但是在固定位置原点画能让向量更加简单。向量的箭头方向可以理解为指向空间中的某个位置，这个位置就恰好是向量的值。所以如果 2D 向量 $\langle 1,2 \rangle$ 把尾部绘制在原点，那么头部位置就是 (1,2)，如图 3.2 所示。

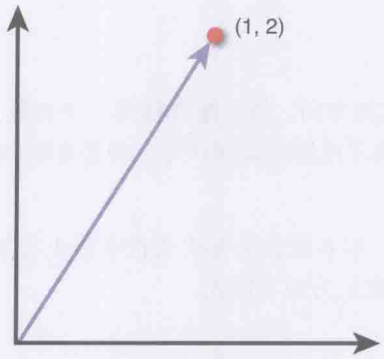


图 3.2 2D 向量 ⟨1, 2⟩ 尾部绘制在原点，那么头部位置就是 (1,2)

向量上有几个基本操作。这些计算是很有价值的，而理解背后的几何意义也同样重要。因为这些几何意义能够帮助你开发游戏。

在下一节，会讨论更多关于向量运算的内容。值得注意的是，虽然演示的是 2D，但是可以轻易拓展到 3D。

加法

两个向量加在一起，把两个向量相对应的分量相加即可：

$$\vec{c} = \vec{a} + \vec{b} = \langle a_x + b_x, a_y + b_y, a_z + b_z \rangle$$

向量相加的几何意义，就相当于把一个向量的尾部串在另一个向量的头部，然后从前一个向量的尾部指向后一个向量的头部，得到的即是相加的向量，如图 3.3 所示。

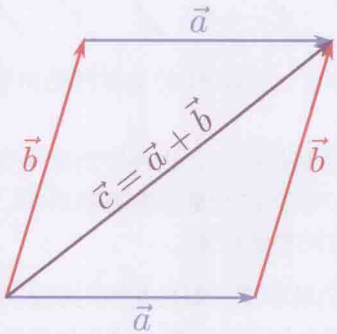


图 3.3 向量相加和平行四边形法则

注意它们是怎么排列的：一个是以 $\vec{a}$为开始向量，另一个是以 $\vec{b}$为开始向量。但不管以哪个开始，结果都一样。这就是**平行四边形法则**，这是因为向量加法支持交换律，就跟实数加法一样：

$$\vec{a}+\vec{b}=\vec{b}+\vec{a}$$

减法

两个向量的减法，就是让两个向量对应的分量相减：

$$\vec{c}=\vec{b}-\vec{a}=\langle b_x-a_x, b_y-a_y, b_z-a_z\rangle$$

为了展示向量相减的几何意义，要将向量的尾部都画在同一个起点，如图 3.4(a) 所示。然后从一个向量的头部指向另一个向量的头部。因为向量减法是不支持交换律的，所以不同顺序有不一样的结果。记住向量相减方向的方法就是，如果你要向量从 $\vec{a}$指向 $\vec{b}$，那么计算就是 $\vec{b}-\vec{a}$。

减法是很重要的，它使得你可以构造两个向量之间的向量。但是如果总是让向量尾部在原点，头部指向一个位置，那么怎么得到两点之间的向量？

假设有 (5,2) 位置的 A点及 (3,5) 位置的 B点。如果把这些点用尾部在原点的向量表示会怎样？就像之前所说的，这个值用哪种方法表示都可以。但是如果从向量的角度去思考，它们的减法会得到向量之间的向量，就像图 3.4(b) 中展示的那样。前面说的规则同样有效：A指 B就是 $B-A$。

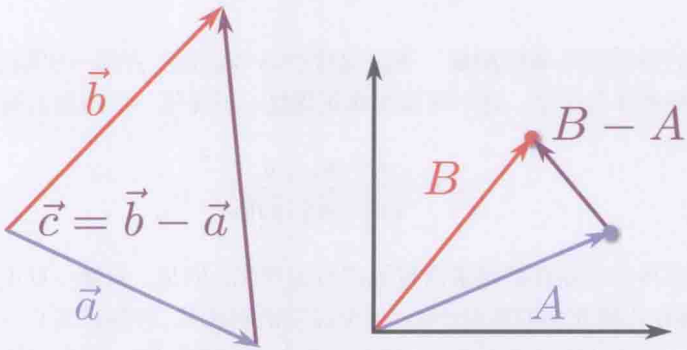


图 3.4　(a) 向量相减 (b) 从原点出发的向量

向量相减在游戏中很常见。比如想象你在为游戏设计 UI 箭头系统。玩家不断得到系统指示，游戏需要一个箭头指引玩家到达目的地。这种情况下，箭头的方向就是从玩家指向目的地，

那么计算方式如下：

```
arrowVector = objective.position - player.position
```

长度、单位向量和正规化

就像之前提到的，向量表示了长度和方向。向量的长度符号表示为在向量两边加竖线，比如 $\|\vec{a}\|$。计算向量长度的公式为：

$$\|\vec{a}\| = \sqrt{a_x^2 + a_y^2 + a_z^2}$$

这跟距离公式非常类似（$d = \sqrt[2]{(x_2 - x_1)^2 + (y_2 - y_1)^2}$），这是因为它就是距离公式，计算向量指向的位置到原点的距离。

计算平方根还是相对昂贵的计算，尽管不像过去那么难。如果你就是要平方根的值，那么平方根是无法避免的。但是，如果你的需求是判断 *A* 与 *B* 哪个更靠近玩家，这种情况下你可能会创建一个向量从玩家指向 *A*，同样也创建另一个向量，从玩家指向 *B*。然后计算长度，比较两个向量哪个更长是完全没问题的。但是由于长度不可能为负，所以比较长度跟比较长度的平方是一样的：

$$\|\vec{a}\| < \left\|\vec{b}\right\| \equiv \|\vec{a}\|^2 < \left\|\vec{b}\right\|^2$$

因此，在比较大小的情况下是可以避免平方根的计算。如果在长度公式上两边取平方，将得到**长度平方**：

$$\|\vec{a}\|^2 = a_x^2 + a_y^2 + a_z^2$$

向量长度等于 1 的时候就是**单位向量**。单位向量的符号就是上方加一顶帽子，就像 $\hat{u}$ 这样。把非单位向量转换成单位向量，这个转换叫作**正规化**。正规化一个向量的时候，用每个分量除以向量长度即可：

$$\hat{a} = \langle \frac{a_x}{\|\vec{a}\|}, \frac{a_y}{\|\vec{a}\|}, \frac{a_z}{\|\vec{a}\|} \rangle$$

下一节讨论到的多种向量操作都要求向量先进行正规化。但是，向量一旦正规化，就会丢失长度信息。所以你在正规化的时候要小心。一个经验法则就是，那些只关心方向的向量（就像之前 UI 的例子），你可以将它们正规化。如果你关心方向和长度，那么向量不应该正规化。

赫赫有名的“卡马克快速平方根”

在 20 世纪 90 年代，个人计算机还不能有效地处理浮点数运算。换句话说，使用整型计算比浮点数快很多。

在3D游戏中，这是个大问题，因为大部分计算都是用浮点数。这就要求程序员使用更加聪明的方法来进行计算。一个问题就是计算平方根，特别是平方根倒数。为什么是倒数？因为正规化一个向量的时候，你要么给每个分量除一个平方根，要么给每个分量乘以一个平方根倒数。乘法比除法要快得多，因此会选择平方根倒数。
当《雷神之锤III竞技场》开源的时候，程序员们发现了一段让人惊叹的平方根倒数计算。以下就是那些源码，附上真正的注释。

```
float Q_rsqrt(float number)
{
   int i;
   float x2, y;
   const float threehalfs = 1.5F;

   x2 = number * 0.5F;
   y  = number;
   i  = * ( int * ) &y; // 对浮点数的邪恶位级hack
   i  = 0x5f3759df - ( i >> 1 ); // what the f***?
   y  = * ( float * ) &i;
   y  = y * ( threehalfs - ( x2 * y * y ) ); // 第1次迭代

   return y;
}
```

第一眼看上去，代码不是那么容易理解。它使用牛顿逼近法来完成计算。那行16进制值是初始估算值，只经过了一次迭代就得到了精确值。
有趣的是，尽管这个平方根总能让人联想到卡马克，但是作为id Softwave公司的创办人和技术指导，这些代码并不是他写的。Rys Sommefeldt一篇有趣的Beyong3D文章中找到了真正的作者：*http://www.beyond3d.com/content/articles/8/*。

标量乘积

向量可以与**标量**进行乘积，标量就是一个实数。将向量与标量进行乘积，你只需将每个分量乘以这个标量即可：

$$s \cdot \vec{a} = \langle s \cdot a_x, s \cdot a_y, s \cdot a_z \rangle$$

如果将向量乘以正数，那么只改变向量长度。如果将向量乘以负数，会得到反方向的向量。图3.5演示了两种不同的标量乘积结果。

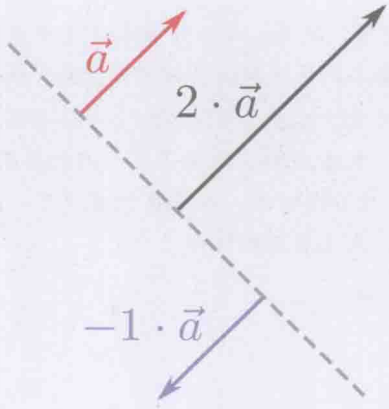


图 3.5 标量乘积

点乘

向量拥有不止一种乘法操作，而是拥有两种不同的乘法操作：**点乘**和**叉乘**。点乘得到标量，而叉乘得到向量。点乘的一个常见用法就是计算两个向量之间的夹角。

点乘计算如下：

$$\vec{a} \cdot \vec{b} = a_x \cdot b_x + a_y \cdot b_y + a_z \cdot b_z$$

注意等式两边都没有关于夹角的值。为了得到向量夹角，向量点乘公式要用三角函数公式来表示：

$$\vec{a} \cdot \vec{b} = \left\|\vec{a}\right\| \left\|\vec{b}\right\| \cos\theta$$

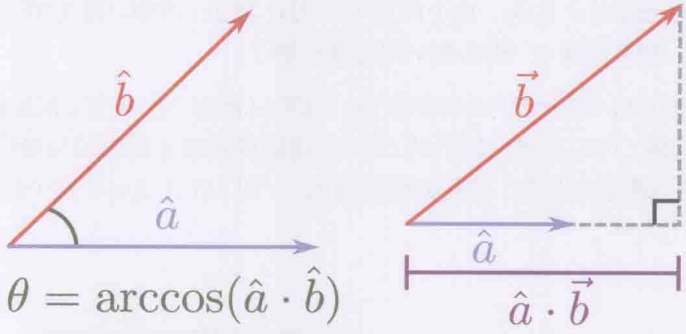
这个公式是基于余弦公式的，完整版本可以看本书相关资料中提到的 Lengyel 的那本书。有了这个公式之后，就可以得到 θ：

$$\theta = \arccos\left(\frac{\vec{a} \cdot \vec{b}}{\left\|\vec{a}\right\| \left\|\vec{b}\right\|}\right)$$

如果 $\vec{a}$和 $\vec{b}$都是单位向量，除法可以忽略，因为长度为 1。如图 3.6(a) 所示。这里的简化也是正规化的理由之一，如果你只关心向量方向的话。

由于点乘可以计算向量之间的夹角，一些特殊情形需要记住。如果两个单位向量的点乘结果为 0，意味着两个向量互相垂直，因为 $\cos(90°) = 0$。如果点乘结果为 1，意味着两个向量平行并且同向。结果为-1，则意味着两个向量平行且反向。

点乘另一个重要的广为人知的部分就是**投影**，如图 3.6(b) 所示。在这种情况下，有单位向量和非单位向量。单位向量延伸到与另一个向量组成的直角三角形的直角处。这种情况下，点乘会得到单位向量延伸后的长度。



(a) (b)

图 3.6 单位向量之间的夹角 (a) 和投影 (b)

就像实数相乘一样，点乘也支持交换律、分配律，以及结合律：

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a}$$
$$\vec{a} \cdot \left(\vec{b} + \vec{c}\right) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c}$$
$$\vec{a} \cdot \left(\vec{b} \cdot \vec{c}\right) = \left(\vec{a} \cdot \vec{b}\right) \cdot \vec{c}$$

另外一个有帮助的提示就是长度的平方等于向量与自身点乘：

$$\vec{v} \cdot \vec{v} = \|\vec{v}\|^2 = v_x^2 + v_y^2 + v_z^2$$

问题举例：向量反射

现在我们学了一些向量运算，下面就试一下解决真实的游戏问题吧。假设你有一颗球朝着墙壁运动。当这颗球撞到墙上的时候，你想计算反射方向。现在假设墙壁是平行于坐标系中某一个轴的，那么问题就比较简单。比如说，如果墙体平行于 x 轴，球从墙上弹开，只要将球的 y 轴速度取反就行。

但是取反这种方法在墙体与坐标轴不平行的情况下就不起作用了。这种常见的反射问题可以用向量来解决。如果可以正确计算出向量的反射，那么在墙体任意朝向下都能用。

向量问题如果没有可视化的帮助，那么是有点棘手的。图 3.7(a) 显示了问题的初始状态。有两个已知值：向量 $\vec{v}$表示球碰撞前的速度，法线 $\hat{n}$就是垂直于反射表面的单位向量。向量 $\vec{v'}$ 表示反射后的速度。

如果我们从 $\vec{v}$的尾连接到 $\vec{v'}$ 的头，这个向量可以用 $\vec{v'}$ 减去 $\vec{v}$得到。图 3.7(b) 画出了辅助向量 $2\vec{s}$。这就意味着，如果得到 $\vec{s}$，那么整个问题就得解了。

如果以某种方式得到将 $\hat{n}$延伸到 $\vec{s}$头部的向量，就可以将这个向量与 $\vec{v}$相加得到 $\vec{s}$。图 3.7(c) 显示了如果将 $\vec{v}$反转，可以通过反转后的 $\vec{v}$在 $\hat{n}$上投影得到刚才所说的向量的长度。因为 $\hat{n}$是单位向量，用 $\hat{n}$乘以刚才的长度，就得到延伸后的 $\hat{n}$。图 3.7(d) 显示了这个延伸后的向量，然后就可以计算出 $\vec{s}$。

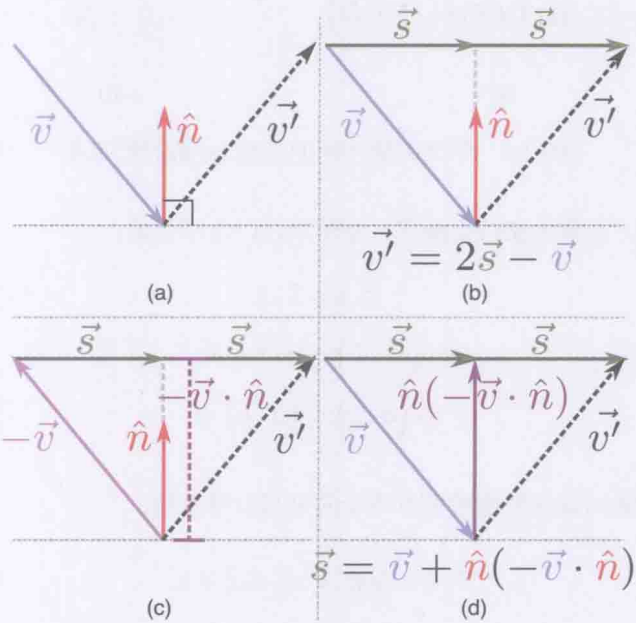


图 3.7 解出反射向量的步骤

现在有了 $\vec{s}$，可以通过一系列代数运算倒推得到 $\vec{v'}$。

$$\vec{v'} = 2\vec{s} - \vec{v}$$
$$\vec{s} = \vec{v} + \hat{n}(-\vec{v} \cdot \hat{n})$$
$$\vec{v'} = 2(\vec{v} + \hat{n}(-\vec{v} \cdot \hat{n})) - \vec{v}$$
$$\vec{v'} = 2\vec{v} + 2\hat{n}(-\vec{v} \cdot \hat{n}) - \vec{v}$$
$$\vec{v'} = \vec{v} - 2\hat{n}(\vec{v} \cdot \hat{n})$$

当你在解向量问题的时候，确认你的向量操作的有效性是很有帮助的。比如说，如果一个向量问题的解法告诉你用向量加上标量，这就意味着解法有问题。这跟在物理上要确保单位统一是一样的。

所以看一下我们的向量发射问题。为了得到 $\vec{v}$，$2\hat{n}\left(\vec{v}\cdot\hat{n}\right)$ 的结果必须是一个向量。可以深入看一下这个算式，先将 $\hat{n}$ 乘以 2，得到一个向量。然后乘以点乘得到的标量。所以最终再将两个向量相减，这个运算是合法的。

这个向量反射问题对于目前所学的向量运算是个绝佳的例子。而且也涉及游戏程序员常见的面试问题。

叉乘

两个向量**叉乘**会得到第 3 个向量。给定两个向量，可以确定一个平面。叉乘得到的向量就会垂直于这个平面，如图 3.8(a) 所示，称之为平面的**法线**。

因为它能得到平面的法线，所以叉乘只能在 3D 向量中使用。这就意味着，为了在 2D 向量中使用叉乘，要先通过将 z 分量设为 0 的方式转换成 3D 向量。

叉乘表达如下：

$$\vec{c}=\vec{a}\times\vec{b}$$

值得注意的是，技术上来讲，平面的垂直向量有两个：与 $\vec{c}$ 反方向的向量。所以你怎么知道叉乘结果向量的朝向？结果取决于坐标系的**手系**。在右手坐标系中，取出你的右手，在拇指和食指间形成 90°。同样地，将中指与食指形成 90°。然后你就可以将食指对准 $\vec{a}$，中指对准 $\vec{b}$。拇指的方向就是叉乘结果的朝向。图 3.8(b) 演示了这个**右手法则**。

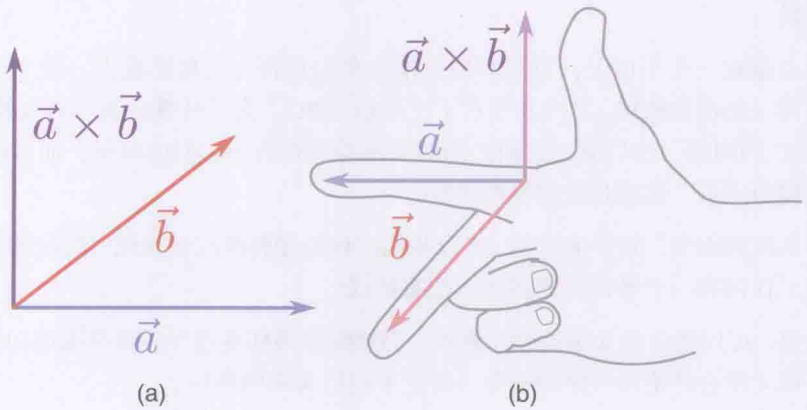


图 3.8　叉乘 (a) 和右手法则 (b)

你可能会注意到，如果你将食指对准 $\vec{b}$，中指对准 $\vec{a}$，拇指的朝向就会是反方向。这是因为叉乘不满足交换律，实际上它满足反交换律：

$$\vec{a} \times \vec{b} = -\vec{b} \times \vec{a}$$

很重要的另一个点就是要记住不是所有游戏都是用右手坐标系的。我们会在后面更加深入地讨论坐标系，在此之前，先假设总是用右手坐标系。

已经在概念上讲述了叉乘，现在看一下叉乘是怎么计算的：

$$\vec{c} = \vec{a} \times \vec{b} = \langle a_y b_z - a_z b_y, a_z b_x - a_x b_z, a_x b_y - a_y b_x \rangle$$

仅根据上面的公式记住叉乘是比较困难的。有一个口诀可以帮助我们记忆："xyzzy"。这句口诀是说，叉乘第一个分量的下标序列：

$$c_x = a_y b_z - a_z b_y$$

只要你记住了基本的公式 $c = a \cdot b - a \cdot b$，"xyzzy" 就是每个字母的下标。只要记住了第 1 行，你就可以通过按照顺序移动下标的方式算出 c_y 和 c_z，移动规则为 $x \to y \to z \to x$。这样就得到叉乘的后续两行：

$$c_y = a_z b_x - a_x b_z$$
$$c_z = a_x b_y - a_y b_x$$

跟点乘一样，叉乘也有要注意的特殊情况。如果叉乘返回的 3 个分量都为 0，意味着两个向量**共线**，也就是在一条直线上。两个共线的向量不能确定一个平面，这就是为什么无法返回该平面的法线。

由于三角形总是在一个平面上，可以利用叉乘去确定是否与三角形垂直。这个三角形的垂线，我们叫作三角形的**法线**。图 3.9 展示了三角形 *ABC*。为了计算法线，首先构造从 *A* 到 *B* 和从 *A* 到 *C* 的向量。然后通过它们之间的叉乘得到垂直于它们的向量，而且根据右手法则，该向量朝向书页，这就是我们要的法线。

因为只关心法线的方向，而不是长度，所以我们应该总是将法线正规化。法线的概念不仅是三角形才有，任何在一个平面上的多边形都有法线。

值得注意的是，可以把向量叉乘的顺序调换，仍然能够得到垂直于三角平面的向量，虽然它朝向相反。第 4 章会讨论顶点序的概念，决定了进行叉乘的顺序。

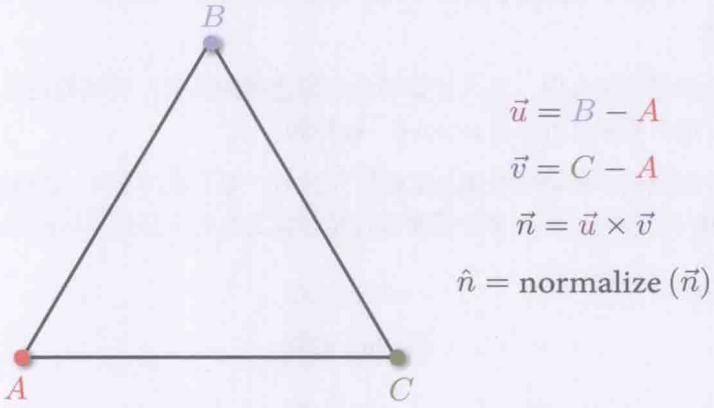


图 3.9　三角形 *ABC* 的法线朝书页方向（在右手坐标系下）

问题举例：旋转一个 2D 角色

看一下另一个例子。假如有一款顶视角的动作游戏，控制一个角色到处移动。我们编写了一个场景事件，在某个地点爆炸，并希望角色可以朝向爆炸的地方。但是，这个场景事件是动态的，玩家的朝向也是动态的。我们不希望玩家突然朝向爆炸点，而是希望玩家在短时间内平滑地旋转到正确的朝向。

为了实现这个平滑的过渡，需要完成两件事。首先，需要知道最终的朝向和最初的朝向。一旦有了这两个值，就可以确定角色应该顺时针还是逆时针旋转。图 3.10 展示了这个问题。

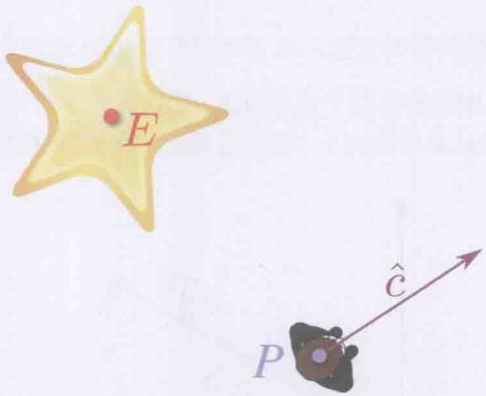


图 3.10　角色需要旋转朝向爆炸

向量 c_y 为玩家当前的朝向。P 是玩家在游戏世界中的位置，E 是爆炸的位置。解题的第一步是得到玩家朝向爆炸的向量。因为想得到 P 到 E 的向量，所以应该是 $E - P$。而且由于我们

只关心该向量的方向而不是距离，所以应该对它进行正规化。因此，可以定义 $\hat{n}$ 为指向爆炸方向的单位向量。

现在有了最初的和最终的方向，怎么计算两个向量之间的夹角？答案就是点乘。由于 $\hat{c}$ 和 $\hat{n}$ 都是单位向量，所以夹角 θ 可以由 $\arccos(\hat{c} \cdot \hat{n})$ 得到。

但是点乘没有告诉我们 θ 是顺时针还是逆时针方向的。为了得到方向，这里使用叉乘。又因为叉乘只能在 3D 向量中使用，$\hat{c}$ 和 $\hat{n}$ 需要通过增加值为 0 的 z 分量来转换成 3D 向量：

$$\hat{c} = \langle c_x, c_y, 0 \rangle$$
$$\hat{n} = \langle n_x, n_y, 0 \rangle$$

在得到 3D 向量之后，可以计算 $\hat{c} \times \hat{n}$。如果得到的向量的 z 分量为正，那么旋转为逆时针；如果为负，那么旋转为顺时针。具体的方法可以用右手法则。如果你回到图 3.10，将你的食指对准 $\hat{c}$，中指对准 $\hat{n}$，那么拇指朝书页外。在右手坐标系中，z 值为正，就是逆时针旋转。

线性插值

线性插值能够计算两个值中间的数值。举个例子，如果 $a = 0$ 而且 $b = 10$，从 a 到 b 线性插值 20% 就是 2。线性插值不仅作用在实数上，它能够作用在任意维度的值上。可以对点、向量、矩阵、四元数等数值进行插值。不管值的维度是什么，都能用一个公式表达：

$$\text{Lerp}(a, b, f) = (1 - f) \cdot a + f \cdot b$$

在公式中，a 和 b 都是即将插值的变量，而 f 则是介于 [0,1] 的因子。

在游戏中，插值的常见应用就是将两个顶点插值。假设有一个角色在 a 点，他需要平滑地移动到 b 点。Lerp 通过 f 值从 0 增加到 1，即可做到将 a 点平滑过渡到 b 点，如图 3.11 所示。

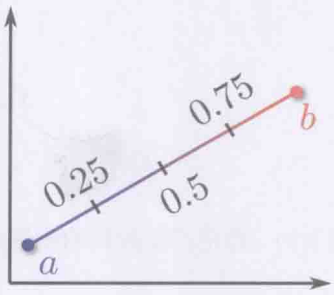


图 3.11 Lerp 通过 f 的变化，将 a 过渡到 b

坐标系

不是所有的 3D 游戏都使用相同的坐标系。有的坐标系 y 轴向上，而有的则 z 轴向上。有的使用右手坐标系，而有的使用左手坐标系。坐标系的选择是很随意的，取决于个人。

因为 3D 系统有不止一种表达的方法，当你使用新框架的时候，了解采用了何种坐标系是很重要的。只要代码中保持一致，最终用哪种坐标系就不是大问题。（见下面“混合坐标系”的幽默示例）

有时候需要将一种坐标系转换或另外一种。常见的情况就是在 3D 模型软件中使用的坐标系与游戏中不同。转换坐标系可能会做去掉分量或者交换分量的操作。这取决于两个坐标系分别是什么。比如说，图 3.12 展示了两个 y 轴向上的系统：一个右手坐标系和一个左手坐标系。在这种情形中，转换很简单：坐标系切换只需要对 z 分量取负。

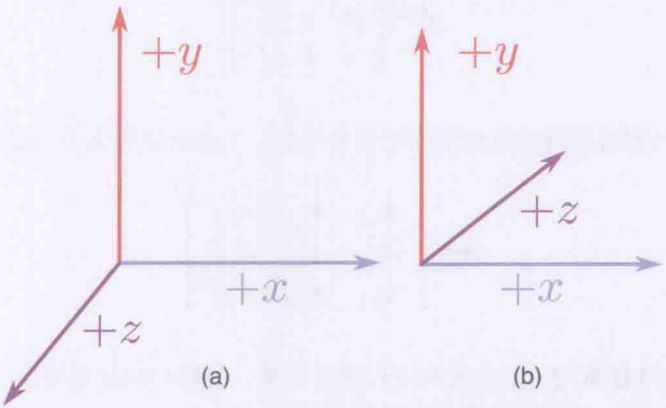


图 3.12　例子中的右手坐标系 (a) 和左手坐标系 (b)

混合坐标系

在《指环王：征服》中，我的责任就是为音效团队开发功能。在项目的早期，音效主设计师遇到一个问题找到了我。他注意到游戏中在屏幕左侧爆炸的时候，右边的音响有声音，反过来也一样。他检查了很多次这个音响，确认音响的连接是完全没有问题的。

随着对问题研究的深入我们发现：我们的游戏中，大部分代码都是用左手坐标系，但是第三方音效库是用右手坐标系的。这就导致了错误的坐标系传入音效系统。解决办法就是在传入音效库的时候将左手坐标系转换成右手坐标系。在经过转换之后，游戏中的音效终于第一次正确运行了。

默认情况下，DirectX 是用左手 y 轴向上的坐标系，OpenGL 是用右手 y 轴向上的坐标系。但这并不意味着所有的游戏在使用这些库的时候必须使用默认坐标系，因为使用两种坐标系都相当简单。

有时候 x 轴、y 轴和 z 轴用**轴向量**$\hat{i}$、$\hat{j}$、$\hat{k}$ 表示。轴向量用于定义坐标系。后面在第 4 章会讨论到一种通用的坐标系变换方法，就是**改变它们的轴向量**。但在此之前，要先了解一下矩阵。

矩阵

矩阵就是一组实数，有着 m 行 n 列，通常用大写字母表示。在游戏中，3×3 和 4×4 矩阵是最常见的。一个 3×3 的矩阵表达方式如下，其中 a 到 i 都是矩阵中的数字：

$$\boldsymbol{A}=\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

另一个表示矩阵的方法是通过坐标下标来表示元素。上面的矩阵 $\boldsymbol{A}$ 可以这么表示：

$$\boldsymbol{A}=\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,1} & A_{3,2} & A_{3,3} \end{bmatrix}$$

在 3D 游戏中，矩阵经常用于对向量和定点进行平移、旋转和缩放等变换。后面会在第 4 章讨论更多变换的内容。现在先学习基本的矩阵操作。

加法/减法

两个矩阵在拥有相同维度的情况下可以进行相加或者相减。相加只需要将对应的分量相加即可：

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}+\begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix}=\begin{bmatrix} a+j & b+k & c+l \\ d+m & e+n & f+o \\ g+p & h+q & i+r \end{bmatrix}$$

减法也同理，分量之间进行相减。上面两个公式在游戏中很少用得上。

标量乘法

跟向量乘以标量类似，矩阵乘以标量计算方式如下：

$$s\cdot\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}=\begin{bmatrix} s\cdot a & s\cdot b & s\cdot c \\ s\cdot d & s\cdot e & s\cdot f \\ s\cdot g & s\cdot h & s\cdot i \end{bmatrix}$$

乘法

矩阵乘法在游戏中应用很广。理解同维度矩阵乘法最简单的方式就是将分量理解成向量进行点乘。假设你有如下两个矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$：

$$\boldsymbol{A}=\begin{bmatrix} a & b \\ c & d \end{bmatrix}\quad \boldsymbol{B}=\begin{bmatrix} e & f \\ g & h \end{bmatrix}$$

计算 $\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}$，先用 $\boldsymbol{A}$ 的第 1 行点乘 $\boldsymbol{B}$ 的第 1 列。点乘的结果就得到了 $\boldsymbol{C}$ 矩阵左上角分量的值。$\boldsymbol{A}$ 的第 1 行和 $\boldsymbol{B}$ 的第 2 列进行点乘就得到 $\boldsymbol{C}$ 的右上角分量的值。对 $\boldsymbol{A}$ 的第 2 行也进行类似操作就得到了完整的矩阵乘法。

$$\boldsymbol{C}=\boldsymbol{A}\times\boldsymbol{B}=\begin{bmatrix} a & b \\ c & d \end{bmatrix}\times\begin{bmatrix} e & f \\ g & h \end{bmatrix}=\begin{bmatrix} a\cdot e+b\cdot g & a\cdot f+b\cdot h \\ c\cdot e+d\cdot g & c\cdot f+d\cdot h \end{bmatrix}$$

矩阵乘法并不要求矩阵有相同的维度，只要求 $\boldsymbol{A}$ 的行数等于 $\boldsymbol{B}$ 的列数。比如下面的乘法也是合法的：

$$\begin{bmatrix} a & b \end{bmatrix}\times\begin{bmatrix} c & d \\ e & f \end{bmatrix}=\begin{bmatrix} a\cdot c+b\cdot e & a\cdot d+b\cdot f \end{bmatrix}$$

4×4 矩阵的乘法与 2×2 一样，只是要计算更多：

$$\begin{bmatrix} C_{1,1} & C_{1,2} & C_{1,3} & C_{1,4} \\ C_{2,1} & C_{2,2} & C_{2,3} & C_{2,4} \\ C_{3,1} & C_{3,2} & C_{3,3} & C_{3,4} \\ C_{4,1} & C_{4,2} & C_{4,3} & C_{4,4} \end{bmatrix}=\begin{bmatrix} A_{1,1} & A_{1,2} & A_{1,3} & A_{1,4} \\ A_{2,1} & A_{2,2} & A_{2,3} & A_{2,4} \\ A_{3,1} & A_{3,2} & A_{3,3} & A_{3,4} \\ A_{4,1} & A_{4,2} & A_{4,3} & A_{4,4} \end{bmatrix}\times\begin{bmatrix} B_{1,1} & B_{1,2} & B_{1,3} & B_{1,4} \\ B_{2,1} & B_{2,2} & B_{2,3} & B_{2,4} \\ B_{3,1} & B_{3,2} & B_{3,3} & B_{3,4} \\ B_{4,1} & B_{4,2} & B_{4,3} & B_{4,4} \end{bmatrix}$$

$$C_{1,1}=A_{1,1}\cdot B_{1,1}+A_{1,2}\cdot B_{2,1}+A_{1,3}\cdot B_{3,1}+A_{1,4}\cdot B_{4,1}$$

……

要注意的是，矩阵乘法是不服从交换律的，但是跟加法一样服从分配律和结合律：

$$A \times B \neq B \times A$$
$$A \times (B + C) = A \times B + A \times C$$
$$A \times (B \times C) = (A \times B) \times C$$

矩阵另一个重要的属性就是**单位矩阵**。如果将一个标量乘以 1，不会改变标量的值。与之类似，矩阵乘以单位矩阵 I 也不会改变矩阵的值。

$$A = A \times I$$

单位矩阵的行数和列数都等于 A 矩阵的行数。单位矩阵上除了对角线上的元素为 1，其余都为 0。因此，一个 3×3 矩阵是这个样子的：

$$I_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

逆矩阵

如果一个矩阵与一个标记了 $^{-1}$ 的矩阵相乘，就会得到单位矩阵：

$$I = A \times A^{-1}$$

计算矩阵的逆矩阵是高级话题，这里就不展开讲了。但是要知道，不是所有的矩阵都有逆矩阵，这一点是很重要的。而在游戏中如果矩阵没有逆矩阵，通常都意味着计算有误。所幸第 4 章讨论的逆矩阵计算是比较简单的。

转置

转置一个矩阵就是将矩阵的每一行与每一列对换，标志为 ^{T}。所以第 1 行成为第 1 列，第 2 行成为第 2 列，以此类推。以下是转置 3×3 矩阵的例子：

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{\mathrm{T}} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

有好几种矩阵都是**正交矩阵**，它的逆矩阵就是它的转置矩阵。旋转矩阵就是其中之一。

用矩阵变换 3D 向量

将矩阵作用在向量上，必须将二者相乘。但在相乘之前，向量需要用矩阵表示。向量有两种潜在的表达方式：只有一行的矩阵或者只有一列的矩阵。这两种表达方式分别称为**行向量**和**列向量**。

给定向量 $\vec{v} = \langle 1, 2, 3 \rangle$，行向量表示如下：

$$\vec{v} = \begin{bmatrix} 1 & 2 & 3 \end{bmatrix}$$

而列向量表示如下：

$$\vec{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

如果你有一个要与行向量相乘的矩阵，然后想换成与列向量相乘，需要先将矩阵转置。看一下 3D 向量怎么与 3×3 矩阵相乘，第一个例子是行向量，第二个例子是列向量：

$$\begin{bmatrix} x' & y' & z' \end{bmatrix} = \begin{bmatrix} x & y & z \end{bmatrix} \times \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix}$$

$$x' = x \cdot a + y \cdot b + z \cdot c$$

$$y' = x \cdot d + y \cdot e + z \cdot f$$

$$\cdots\cdots$$

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \times \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$x' = a \cdot x + b \cdot y + c \cdot z$$

$$\cdots\cdots$$

注意两种表达方式中 x' 的值是一样的。这是因为 3×3 矩阵是经过转置的。就跟坐标手系一样，只要保持一致，就不用关心用行向量还是列向量。由于大多数 3D 游戏库都是用行向量表示的（与 OpenGL 不同），因此本书也用行向量。

总结

线性代数虽然不是游戏开发中最让人激动的部分，但却是那些技术的基础。熟练掌握线性代数是 3D 游戏编程入门的关键。在后续很多地方会用到向量和矩阵，所以往下阅读之前应该先掌握本章内容。

习题

1. 给定向量 $\vec{a} = \langle 2,4,6 \rangle$ 和 $\vec{b} = \langle 3,5,7 \rangle$，以及标量 $s = 2$，计算下面的式子：

   (a) $\vec{a} + \vec{b}$

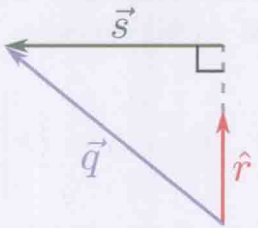
   (b) $s \cdot \vec{a}$

   (c) $\vec{a} \times \vec{b}$

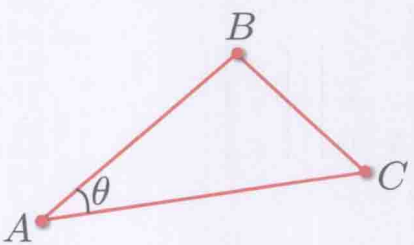
2. 在向量的正规化方面，有没有什么好的经验法则？

3. 给定玩家位置为 P，怎么高效地比较玩家到另外两个点 A 和 B 的距离远近？

4. 给定下图中的向量 $\vec{q}$ 和 $\hat{r}$，利用向量运算算出 $\vec{s}$。



5. 给出下面的三角形，通过向量算出夹角 θ：



$\boldsymbol{A} = \langle -1,2,2 \rangle$

$\boldsymbol{B} = \langle 1,2,3 \rangle$

$\boldsymbol{C} = \langle 2,4,3 \rangle$

6. 使用上面的三角形，在右手坐标系下计算出朝书页外方向的法线。
7. 如果叉乘的顺序改变了，结果会有什么变化？
8. 如果你被委托实现顶视角动作游戏的立体音效，当声音在游戏世界播放时，你必须判断音源相对于角色的位置来控制音响。这个问题使用向量怎么解决？
9. 计算下面矩阵相乘的结果：

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \times \begin{bmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{bmatrix}$$

10. 什么情况下矩阵的转置会等于矩阵的逆矩阵？

相关资料

Lengyel, Eric. ***Mathematics for 3D Game Programming and Computer Graphics (Third Edition).*** **Boston: Course Technology, 2012.** 这本书讨论了本章很多概念的细节，有完整的计算和证明。它也涵盖了超出本书范畴的更加复杂的数学，但是对于一些游戏程序员还是很有用的（特别是专注于图形学领域的程序员）。

第4章

3D图形

虽然第一款3D游戏在街机时代推出，但是直到20世纪90年代中晚期还没有真正意义上的3D游戏。今天，几乎所有AAA级大作都是3D游戏，而且随着智能手机性能的提升，这个趋势也同样出现。

3D游戏显示技术的挑战主要是，怎么将3D的游戏世界显示在2D的显示器上。本章就此讨论了这些技术。

基础

第一款 3D 游戏中的渲染是完全以软件方式实现的（即没有硬件支持）。这意味着即使是画线这种基础功能都要图形程序员去完成。这套将 3D 模型正确渲染到 2D 颜色缓冲的算法称为**软件光栅化**，大部分计算机图形学会花费大量时间在这些算法上。但是现代的计算机已经有了称之为**图形处理单元**（GPU）的图形硬件，这些硬件实现了绘制点、线、三角形等功能。

由此，现代游戏不再需要开发实现软件光栅化了。而焦点则转变为将需要渲染的 3D 场景数据以正确的方式传递给显卡，一般都通过像 OpenGL 和 DirectX 这样的库完成。如果需要进一步自定义这个需求，可以编写一段运行在显卡上称之为**着色器**的小程序来应用传入的数据。在这些数据都传递完成之后，显卡就会将这些数据绘制在屏幕上。编写 Bresenham 画线算法的日子一去不复返。

在 3D 图形中需要注意的是，经常需要计算近似值。这只是因为计算机没有足够的时间去计算真实的光照。游戏不像 CG 电影那样，可以花上几个小时就为了渲染一帧的画面。一个游戏通常需要每秒画 30 帧或者 60 帧，所以精确度需要在性能上做出折中。由于近似模拟而产生的显示错误称之为**图形失真**，没有游戏可以完全避免失真。

多边形

3D 对象在计算机程序中有多种显示方法，在游戏中最广泛应用的就是通过多边形显示，更具体一点来说是三角形。

为什么是三角形？首先，它们是最简单的多边形，它们可以仅用 3 个**顶点**表示。第二点就是三角形总是在一个平面上，而多个顶点的多边形则有可能在多个平面上。最后，任何 3D 对象都可以简单地用细分三角面表示，且不会留下漏洞或者进行变形。

单个模型，我们称为**网格**，是由多个三角片组成的。一个网格的多边形数量主要由用途决定。对于 Xbox 360 游戏来说，一个角色模型可能有一万五千个多边形，而一个木桶模型可能只有几百个多边形。图 4.1 展示了用三角形表示网格。

所有图形库都有方式输入想要显卡渲染的三角形。然后还要提供额外信息，告诉显卡这些数据该怎么渲染，这部分就是本章剩下的篇幅要讲的内容。

坐标系

一个坐标系空间有不同的参考系。比如说，在笛卡儿坐标系中，原点在世界的中间，所有坐标都相对于中心点。与之类似，还有很多坐标系有不同的原点。在 3D 渲染管线中，渲染 3D 模型到 2D 显示器，必须经历 4 个主要的坐标系空间。

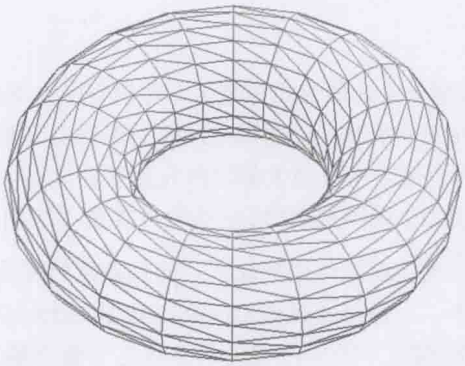


图 4.1 一个三角形网格的例子

- 模型坐标系/局部坐标系
- 世界坐标系
- 视角坐标系/摄像机坐标系
- 投影坐标系

模型坐标系

当我们在建模的时候，比如像在 Maya 这样的软件里面，所有模型顶点的表示都是相对于模型原点的。**模型坐标系**就是那个相对于模型自身的坐标系。在模型坐标系中，原点通常就在模型中心，角色模型的原点在角色两脚中间。这是因为对象的中心点会更好处理。

现在假设游戏场景中有 100 个不同的对象。如果游戏只是加载它们然后以模型坐标系绘制会发生什么？由于所有对象都在模型空间创建，所有对象，包括玩家，都会在原点。相信这样的关卡会很无趣。为了让这个关卡加载正确，需要另一个坐标系。

世界坐标系

有一个新的坐标系称为**世界坐标系**。在世界坐标系中，所有对象都相对于世界的原点偏移。

当加载一个模型之后，它的所有顶点都还在模型坐标系当中。我们不希望手动修改这些数据，因为这样做会导致性能不佳，而且会影响该模型在场景中不同位置的实例。

该用什么办法告诉显卡，我们的模型在哪里绘制呢？而且模型不仅会平移，还会旋转，有时候还有缩放等操作（如图 4.2 所示）。所以最好的方法就是传递模型数据（在模型坐标系）到显卡，同时也附带一些模型在世界中的摆放信息。

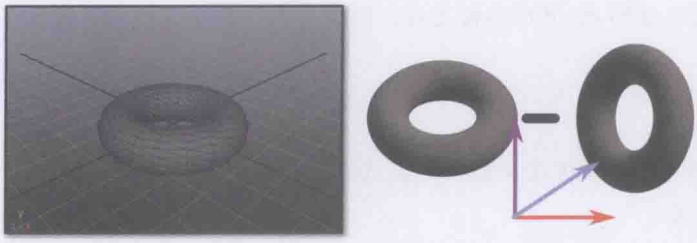


图 4.2　对象在模型坐标系中 (a) 和多个实例对象在世界坐标系中 (b)

最终需要提供刚才所说的额外数据，这将会通过我们所信赖的矩阵来完成。但在可以用矩阵之前，还需要先知道怎么表示三角形的位置。

齐次坐标系

就像之前说过的，经常会有 3D 游戏使用 4D 向量。当 4D 坐标系应用在 3D 空间中时，它们被称为**齐次坐标系**，而第 4 个分量被称为 ***w* 分量**。

在大多数情况下，w 分量要么是 0，要么是 1。如果 $w = 0$，表示这个齐次坐标是 3D 向量。而 $w = 1$，则表示齐次坐标是 3D 的点。但很容易让人疑惑的是，Vector4 类同时用于表示向量和顶点。因此，通过命名规范来保持语义是很重要的：

```
Vector4 playerPosition // 这是个点
Vector4 playerFacing // 这是个向量
```

通过矩阵变换 4D 向量

用于变换的矩阵通常是 4×4 矩阵。为了与 4×4 矩阵相乘，同时也需要 4D 向量。这是齐次坐标的工作方式。

4×4 矩阵乘以 4D 向量跟 3×3 矩阵乘以 3D 向量类似：

$$\begin{bmatrix} x' & y' & z' & w' \end{bmatrix} = \begin{bmatrix} x & y & z & w \end{bmatrix} \times \begin{bmatrix} a & e & i & m \\ b & f & j & n \\ c & g & k & o \\ d & h & l & p \end{bmatrix}$$

$$x' = x \cdot a + y \cdot b + z \cdot c + w \cdot d$$

$$y' = x \cdot e + y \cdot f + z \cdot g + w \cdot h$$

...

在你理解矩阵怎么变换4D齐次坐标之后，你就可以在世界坐标系下构造你想要的变换了。

矩阵变换

矩阵变换就是矩阵用某种方法来影响向量或者顶点。矩阵变换使得我们可以将模型坐标系变换为世界坐标系。

提醒一下，这里所有的矩阵变换都是用行向量表示的。如果你使用的库（比如OpenGL）要求列向量，那么需要将这些矩阵转置。虽然即将展示的矩阵是4×4矩阵，但是缩放和旋转矩阵都可以将第4行和第4列移除，退化成3×3矩阵。

缩放 假设我们的玩家角色模型在创建的时候就给定了大小。在游戏中，有些增强道具可以让角色变大1倍。也就是说，在游戏中需要将角色的模型在世界坐标系中变大1倍。可以通过用模型的每个顶点乘以**缩放矩阵**来完成，它可以将模型在各个轴上进行缩放。除了对角线不一样，它跟单位矩阵差不多。

$$s\left(s_x, s_y, s_z\right) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

如果 $s_x = s_y = s_z$，就是**等比缩放**，就是说向量在每个轴上缩放值一样。所以当所有缩放值等于2时，模型会变为2倍大小。

要想对缩放矩阵取逆，只需将每个缩放因子取倒数即可。

平移 一个**平移矩阵**可以将顶点移动一段距离。它作用在向量上无效，因为向量在哪里绘制都一样。平移矩阵就是将单位矩阵最后一行填上平移的 x、y 和 z 值。

$$T\left(t_x, t_y, t_x\right) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

举个例子，假如模型的中心点放置在世界坐标系的 $\langle 5, 10, -15 \rangle$ 位置，那么平移矩阵会是这个样子：

$$T(5, 10, -15) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 10 & -15 & 1 \end{bmatrix}$$

这个矩阵不会对向量有影响，因此齐次坐标系表示的向量，w 分量为 0。就是说平移矩阵的第 4 行会由于 w 分量为 0 的原因在乘积时不起作用。但是对于点来说，w 分量为 1，那么平移分量会加到顶点上。

平移矩阵取逆，只需要将平移值取负即可。

旋转 旋转矩阵可以将顶点或者向量相对于某个轴旋转。笛卡儿坐标系下每个轴有一个旋转矩阵，总共有 3 种旋转矩阵。因此绕 x 轴旋转跟绕 y 轴旋转的旋转矩阵是不同的。在所有例子中，角度 θ 会传入矩阵当中。

这些旋转称为**欧拉角旋转**，以瑞士数学家名字命名。

$$\text{RotateX}(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{RotateY}(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{RotateZ}(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

以下是一个绕 y 轴旋转 45° 的例子：

$$\text{RotateY}(45°) = \begin{bmatrix} \cos 45° & 0 & \sin 45° & 0 \\ 0 & 1 & 0 & 0 \\ -\sin 45° & 0 & \cos 45° & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3种不同的旋转矩阵看上去很难记忆。小技巧就是每个轴对应一个数值：$x = 1$，$y = 2$，$z = 3$。这是绕 x 轴旋转，在矩阵 (1,1) 上的元素是 1，而第 1 行第 1 列的其他元素都为 0。与之类似，绕 y 轴和绕 z 轴都有类似的情况。而最后一行、一列对于 3 个矩阵总是一样的。只要你记住了 0 和 1 的位置，剩下就是记忆 sin 和 cos 的顺序了。

旋转矩阵是正交的，就是说转置矩阵就是它的逆矩阵。

应用矩阵变换 组合多个矩阵的相乘得到最终世界变换矩阵的做法是很常见的。比如说想将角色放在某个特定位置，然后拉长 2 倍，就需要平移矩阵和缩放矩阵。

组合多个矩阵，你只需要将多个矩阵相乘即可。乘积的顺序是非常重要的，因为矩阵相乘不服从交换律。一般来讲，以行为主来表示的话，你的世界变换矩阵要像下面这样计算：

$$\text{WorldTransform} = \text{Scale} \times \text{Rotation} \times \text{Translation}$$

旋转要最先起作用，因为旋转总是相对于原点而言的。如果你先旋转后平移，那么对象能够自转。但是，如果你先平移，后旋转，这个对象就不会自转，而是绕世界坐标系原点旋转，如图 4.3 所示。这样的行为在某些情况下有可能是有用的，但通常都不会适用于世界变换矩阵。

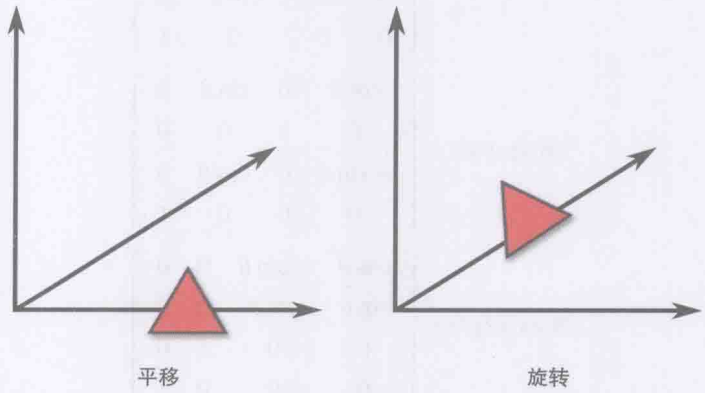


图 4.3 一个先平移后旋转的对象会绕世界坐标系原点转动而不是绕自身转动

视角/摄像机坐标系

在所有对象放置到世界坐标系上正确的位置之后，下一件要考虑的事情就是摄像机的位置。一个场景或者关卡可以完全静止，但是如果摄像机的位置改变，就完全改变了屏幕上的显示。这个称为**视角/摄像机坐标系**，如图 4.4 所示。

图 4.4　摄像机坐标系

所以还需要另外一个矩阵告诉显卡如何将世界坐标系的模型变换到相对于摄像机的位置上。最常见的矩阵是**观察矩阵**。在观察矩阵当中，摄像机的位置通过 3 个轴的额外分量来表示。

对于以行为主的左手坐标系来讲，观察矩阵如下：

$$\text{Look-At} = \begin{bmatrix} L_x & U_x & F_x & 0 \\ L_y & U_y & F_y & 0 \\ L_z & U_z & F_z & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

L 表示左边或者 x 轴，U 表示上方或者 y 轴，F 表示前方或者 z 轴，而 T 则是摄像机的平移。为了构造这个矩阵，需要先计算这 4 个向量。大部分 3D 库可以自动计算这个观察矩阵。但如果你没有这样的代码库，手动计算也不难。

需要 3 个输入来构造观察矩阵：**眼睛**，也就是摄像机的位置、摄像机正在观察的位置、摄像机向上的方向。虽然一些情况下摄像机向上的方向就是世界坐标系向上的方向，但也有特例。在任何情形下，给定这 3 个输入，就可以算出第 4 个向量：

```
function CreateLookAt(Vector3 eye, Vector3 target, Vector3 Up)
   Vector3 F = Normalize(target - eye)
   Vector3 L = Normalize(CrossProduct(Up, F))
   Vector3 U = CrossProduct(F, L)
```

```
    Vector3 T
    T.x = -DotProduct(L, eye)
    T.y = -DotProduct(U, eye)
    T.z = -DotProduct(F, eye)

    // 通过F、L、U和T创建并返回观察矩阵
end
```

在应用视角坐标系之后，整个世界就会变换到从摄像机眼睛的位置去观察。但是，3D 的世界是需要变换到 2D 屏幕上的。

投影坐标系

投影坐标系有时候也叫作屏幕坐标系，是一种将 3D 场景平铺到 2D 平面上得到的坐标系。一个 3D 场景可以通过多种方式平铺在 2D 平面上，两种最常见的方法分别是正交投影和透视投影。

在正交投影中，整个世界挤在 2D 图像中，完全没有深度的感觉。就是说离摄像机远的对象与离摄像机近的对象在视觉上是一样的。任何纯 2D 的游戏都可以看作使用了正交投影。但是，有些正交投影的游戏会传达 3D 信息，比如《模拟人生》和《暗黑破坏神 3》。图 4.5 演示了用正交投影渲染的 3D 场景。

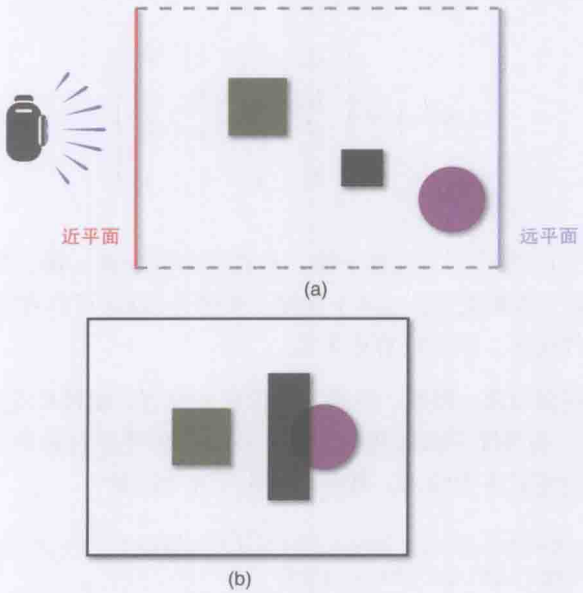


图 4.5 顶视角正交投影 (a) 和在屏幕上得到的结果 (b)

另一种常见的投影则是**透视投影**。在这种投影中，对象在摄像机中会显得近大远小。大部分3D游戏都采用这种投影。图4.6显示了图4.5中的3D场景使用透视投影的结果。

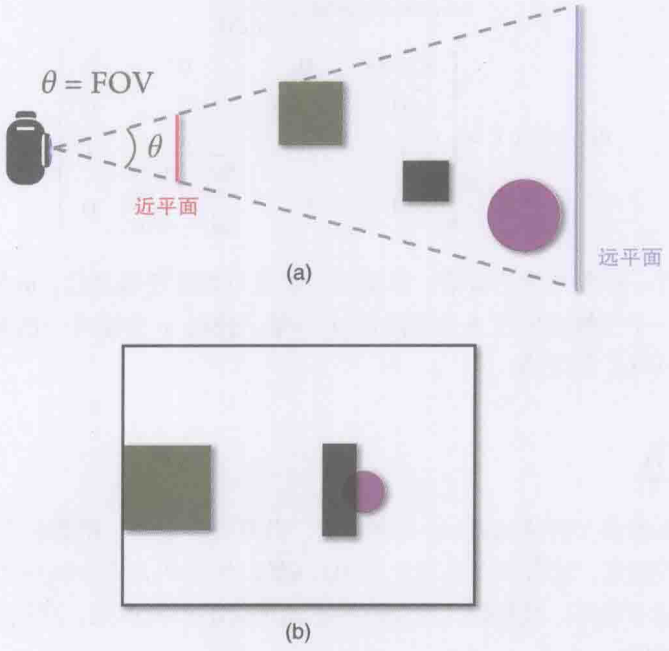


图4.6　顶视角的透视投影(a)和屏幕上得到的结果(b)

两种投影都有近平面和远平面。**近平面**是靠近摄像机的平面，而介于摄像机和近平面之间的物体不参与绘制。游戏中如果某个人物太过于靠近摄像机会突然消失，就是被近平面剔除的原因。与之类似，**远平面**就是远离摄像机的平面，任何物体超过这个平面就不参与绘制了。有的PC游戏让用户选择通过减少绘制次数来提升性能，它们有可能实际上就是将远平面调近了。

正交投影矩阵由4个参数构成，视口的宽和高，还有远平面和近平面到眼睛的最近距离：

$$\text{Orthographic} = \begin{bmatrix} \frac{2}{\text{width}} & 0 & 0 & 0 \\ 0 & \frac{2}{\text{height}} & 0 & 0 \\ 0 & 0 & \frac{1}{\text{far} - \text{near}} & 0 \\ 0 & 0 & \frac{\text{near}}{\text{far} - \text{near}} & 1 \end{bmatrix}$$

透视投影则多了一个**参数视场（FOV）**。就是摄像机的可见角度。视场决定了你能看到多少内容，这一点在后面第8章会讨论更多。加了视场之后就可以计算出透视矩阵：

$$\text{yScale} = \cot \text{fov}/2$$

$$\text{xScale} = \text{yScale} \cdot \frac{\text{height}}{\text{width}}$$

$$\text{Perspective} = \begin{bmatrix} \text{xScale} & 0 & 0 & 0 \\ 0 & \text{yScale} & 0 & 0 \\ 0 & 0 & \dfrac{\text{far}}{\text{far} - \text{near}} & 1 \\ 0 & 0 & \dfrac{-\text{near} \cdot \text{far}}{\text{far} - \text{near}} & 0 \end{bmatrix}$$

这个透视矩阵还有一样要考虑的事情。那就是当顶点与矩阵相乘之后，w 分量不再为 1。**透视分割**需要让每一个变换后的顶点分量除以 w 分量，使得 w 分量再一次为 1。这个过程真正使得透视变换具有了深度感。

光照与着色

到这一步为止，本章讲了传递 GPU 的一堆顶点、世界变换矩阵、摄像机变换矩阵和投影变换矩阵。有了这些信息，就可以在屏幕上将 3D 场景以线框的方式绘制到 2D 的颜色缓冲中。可是线框让人激动不起来。最起码，希望这些三角形能够得到填充，现代 3D 游戏也同样需要颜色、纹理和光照。

颜色

最简单的在 3D 场景中表示颜色的方式就是使用 **RGB 颜色空间**，就是说颜色分开形成红色、绿色和蓝色分量。这是因为 RGB 直接就是显示器的绘图方式。每个屏幕上的像素都是通过红色、蓝色、绿色组成的。如果你仔细观察屏幕，你可能能看出那些颜色分量。

在选择了 RGB 表示颜色之后，下一个要决定的就是**色深**，就是每个像素用多少位来存储。如今，大部分游戏中的 3 颜色分量都用 8 位来存储，就是说红色、蓝色、绿色都有 256 种可能的颜色。这就有了大约 1600 万种不同的颜色。

因为原色的范围从 0 到 255，有时候会用其他方式表达。在网站图像中，#ff0000 表示红色值 255、绿色值 0、蓝色值 0。在 3D 图形中最常见的表示方法是用 0.0 到 1.0 之间的浮点数表示。所以 1.0 就是原色的最大值，而 0.0 就是原色不存在。

根据游戏的不同，有些游戏会有第 4 个分量，称为**不透明度**。不透明通道决定了一个像素的透明度。不透明度为 1.0 就是说像素是 100% 不透明的，而 0.0 则表示看不见。如果游戏需要实现像河流这种效果，就是能看见河流后面的物体，不透明度是必须要有的。

所以如果游戏完全支持 RGBA，每个分量用 8 位表示，那么每个像素总共有 32 位（或者 4 个字节）。这是游戏很常见的渲染模式。图 4.7 演示了 RGBA 下不同颜色的效果。

图 4.7　在 RGBA 颜色空间的不透明红色和紫色 (a) 和一个能看见其下颜色的透明蓝色 (b)

顶点属性

为了让模型有颜色，需要在顶点上存储额外的信息。这些信息被称为顶点属性，在大多数现代游戏中每个顶点都会带多个属性。当然，额外的内存负担会影响模型的顶点数。

有很多参数可以做顶点属性。在**纹理映射**中，2D 的图片可以映射到 3D 的三角形中。每个顶点上都有一个**纹理坐标**指定纹理的特定部分与之对应，如图 4.8 所示。最常见的纹理坐标系是 **UV 坐标系**，纹理上的 x 坐标称为 u，而 y 坐标称为 v。

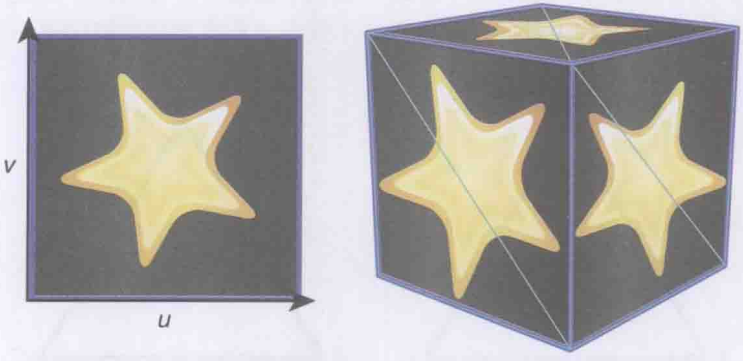


图 4.8　纹理通过 *UV* 坐标系进行映射

仅通过纹理，就可以让场景看起来充满色彩。但是会看起来不够真实，因为场景中没有真实的光照。大部分光照模型依赖于另一种顶点属性：**顶点法线**。回忆一下第 3 章，可以通过叉乘计算出三角形的法线。但是一个顶点，怎么会拥有法线呢？

顶点的法线可以通过多种方法得到。一种方法就是对所有拥有该顶点的三角形的法线取平均值，如图 4.9(a) 所示。如果你希望你的模型看上去很平滑，那么这种方法很好用。但是如果你希望模型看上去边界很清晰就不行了。比如说，你想用平均法线渲染出一个四方体，会得到一个圆角四方体。为了解决这个问题，四方体的每个角都要有 3 个不同的顶点，而每个顶点都存储不同的朝向各自所属平面的法线，如图 4.9(b) 所示。

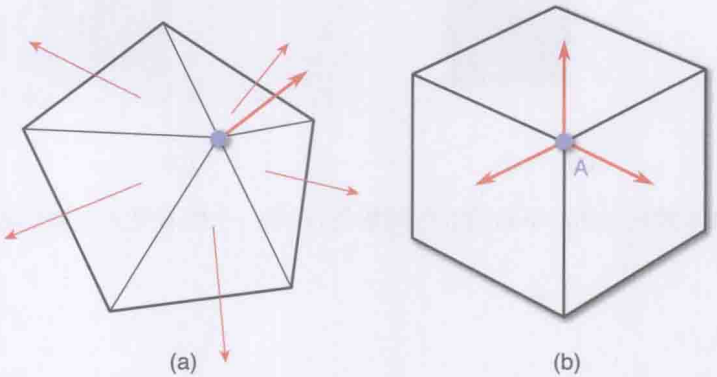


图 4.9 平均顶点法线 (a) 和四方体的顶点根据所属平面有 3 个不同的法线方向 (b)

要记住，一个三角形从技术角度来讲有两个法线，取决于叉乘的顺序。对于三角形来说，这个顺序取决于**顶点序**，可以是顺时针的，也可以是逆时针的。假设一个三角形的顶点 *A*、*B* 和 *C* 有顺时针顶点序，就是说从 *A* 到 *B* 到 *C* 的顺序是顺时针方向，而这些顶点的叉乘结果在右手坐标系中朝书页方向。如果 *A* 到 *B* 到 *C* 的顺序是逆时针方向，那么法线就朝书页外，如图 4.10(b) 所示。就像之前说过的，顶点序并不重要，只要游戏始终保持一致就好。

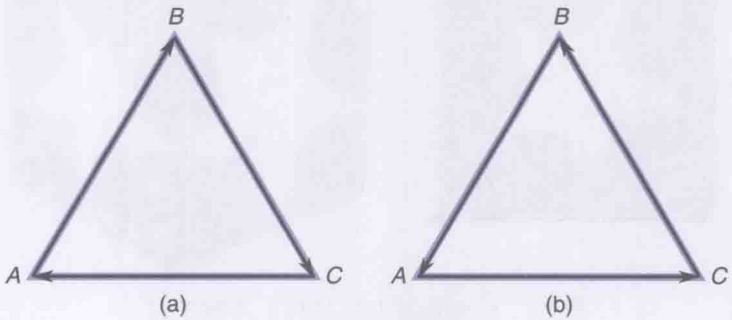


图 4.10 顺时针方向 (a) 和逆时针方向 (b)

顶点序的另一个作用就是用于渲染优化，称为**背面剔除**，就是说没有朝向摄像机的三角片不进行渲染。所以如果你传递的三角片的顶点序与图形库需求的不一样，那么你的三角片不会被看见。图 4.11 演示了这个情况。

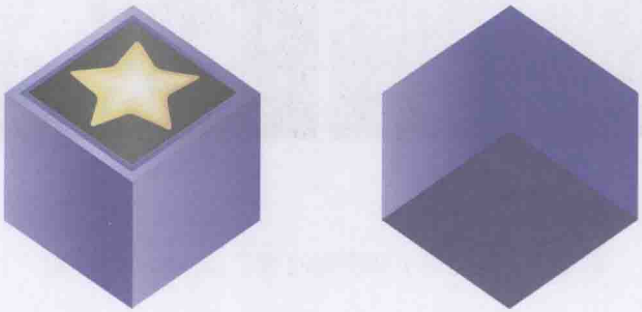


图 4.11 一个四方体使用正确的顶点序渲染 (a) 和同样的四方体使用不正确的顶点序渲染 (b)

光照

一个没有光照的游戏看起来会容易单调枯燥，所以大多数 3D 游戏必须实现某种光照。3D 游戏中使用的光照有好几种类型。一些光照会全局作用于整个场景，而一些光照只作用于光照范围内的区域。

环境光

环境光就是一种添加到场景中每一个物体上的固定光照。在游戏中环境光的值经常会按照不同的关卡需求来设置，取决于关卡所处的时间点。夜间关卡会选用比较阴暗寒冷的环境光，而白天的关卡则会选用比较明亮温暖的。

因为环境光提供了一些光照，环境光对每个物体来说作用都一样。所以可以将环境光想象成多云天气的时候提供最基本的光照。图 4.12(a) 展示了自然界中的多云天气。

方向光

方向光是一种没有位置的光，只指定光照的方向。跟环境光一样，方向光作用于整个场景。但是，由于方向光是带方向的，所以它们会照亮物体的正面，而背面则没有光照。一个方向光的例子就是夏天的阳光。光照的方向向量为太阳指向地球，而面向太阳的一面会亮一些。图 4.12(b) 演示了黄石国家公园中的方向光。

(a) (b)

图 4.12 自然界中环境光的例子 (a) 和方向光 (b)

通常在游戏中，每个关卡只有一个方向光，用来表示太阳或者月亮，但也不一定。一个在黑暗洞穴中的关卡可能会没有方向光。而晚上的体育场则会有可能用多个方向光来表示体育场上的灯光。

点光源

点光源就是某个点向四面八方射出的光照。由于它们从某个点射出，点光源也只会照亮物体的正面。在大多数情况下，不希望点光源无限远。比如说，暗室中的灯泡，如图 4.13(a) 所示。在小范围内有光，但是超了范围光照就马上衰减。点光源不会简单地一直向前。为了模拟这种情形，可以增加**衰减半径**来控制随着距离的增加光照衰减的方式。

聚光灯

聚光灯跟点光源很像，除了点光源向所有方向发射，聚光灯只在锥体内有光外。为了计算锥体范围，需要一个角度作为聚光灯的参数。跟点光源一样，聚光灯只会照亮物体的正面。一个聚光灯的经典例子就是聚光灯，另一个例子是手电筒，如图 4.13(b) 所示。

Phong 光照模型

在光源放入关卡之后，游戏就需要计算光源如何作用于物体上。这些计算都可以通过**双向反射分布函数（BRDF）**来完成。BRDF 用于计算光源怎么作用于物体表面。有很多种不同的 BRDF，下面只讲最基础的一种：**Phong 光照模型**。要注意的是，Phong 光照模型与 Phong 着色模型（稍后会讲到）是不一样的。这两个概念有时候会让人迷惑，但它们是不同的。

图 4.13　点光源的例子，暗室中的电灯泡 (a) 和聚光灯的例子，舞台上的聚光灯 (b)

Phong 光照模型是一种**局部光照模型**，因为它不考虑光的二次反射。换句话说，每个物体都认为在场景中只有自己被渲染。在物理世界中，如果一个红光打到白色的墙上，红光会有少量反射到房屋里其他地方。但是在局部光照模型中是不会发生的。

在 Phong 光照模型中，光被分为几种分量：环境光、漫反射和高光，如图 4.14 所示。这 3 种分量都会影响物体的颜色。**环境光**分量用于整体照亮场景。由于它均匀地作用于整个场景，所以环境光分量与光源的位置和摄像机的位置都无关。

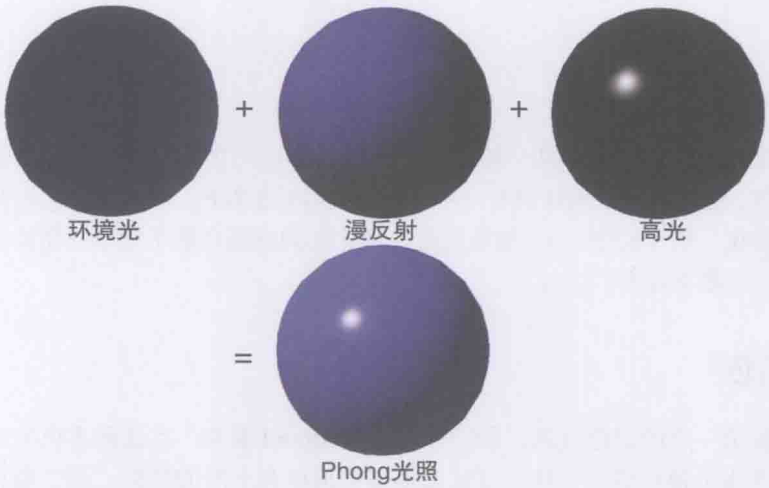


图 4.14　Phong 光照模型

漫反射分量是光源作用于物体表面的主要反射。它会被所有方向光、点光源和聚光灯影响。为了计算漫反射分量，你同时需要物体表面的法线和物体表面到光源方向的向量。但是跟环境光分量一样，漫反射同样也不被摄像机的位置影响。

最后一种分量是**高光**分量，表示物体表面上闪亮的高光。一些有强高光的物体，比如光滑的金属，会比涂上暗色涂料的物体光亮得多。类似漫反射分量，高光分量也同时取决于光源的位置和物体表面的法线。但它还取决于摄像机的位置，因为高光会随着视角方向变换而变换。

总而言之，Phong光照模型的计算并不复杂。这个模型遍历场景中的所有光源，计算表面的颜色，然后以某种方式来决定表面最终的颜色（详见清单4.1）。

清单4.1 Phong光照模型

```
// Vector3 N = 物体表面的法线
// Vector3 eye = 摄像机的位置
// Vector3 pos = 物体表面的位置
// float a = 高光量
Vector3 V = Normalize(eye - pos) // 从物体表面到摄像机
Vector3 Phong = AmbientColor
foreach Light light in scene
   if light affects surface
      Vector3 L = Normalize(light.pos - pos) // 从物体表面到光源
      Phong += DiffuseColor * DotProduct(N, L)
      Vector3 R = Normalize(Reflect(-L, N)) // 计算-L关于N的反射
      Phong += SpecularColor * pow(DotProduct(R, V), a)
   end
end
```

着色

分开光照模型是着色的一种方法，**着色**就是计算表面的三角片如何填充。最基础的着色类型是**平面着色**，就是整个三角片只有一种颜色。使用平面着色，只需每个三角片进行一次光照计算（通常在三角片的中心），然后把通过计算得到的颜色赋予整个三角片。这样做基本能实现着色，可是不好看。

Gouraud着色

有一种稍微复杂一点的着色方法，我们称之为**Gouraud着色**。在这种着色方法中，光照模型的计算需要逐个顶点进行一次。这就使得每个顶点有不同的颜色。而三角片的剩余部分则通过顶点颜色插值填充。举个例子，如果一个顶点为红色，而另一个顶点为蓝色，两点之间的颜色是从红到蓝慢慢混合过渡的。

虽然Gouraud着色比较近似自然色了，但还是有不少问题。首先，着色的质量取决于模型的多边形数量。在低多边形模型上，着色结果会有不少有棱角的边，如图4.13(a)所示。虽然Gouraud着色在高多边形模型上能达到不错的效果，但是会占用不少内存。

另一个问题是高光用在低多边形模型上效果极差，效果可见图 4.13(a)。而在低多边形模型上高光有可能会完全消失，因为只依据顶点来计算光照。虽然 Gouraud 着色流行了好几年，但是随着 GPU 性能的提升，就再也没人使用了。

Phong 着色

Phong 着色是计算三角片上每个像素的光照模型。为了完成这种效果，顶点的法线需要在三角片表面进行插值，然后利用插值得到的法线进行光照计算。

正如大家想象的那样，Phong 着色的计算量比 Gouraud 着色昂贵得多，特别是在场景中有很多灯光的时候。但是，大多数现代硬件都可以轻松处理这部分额外的计算。Phong 着色可以认为是**逐像素光照**，因为光照结果是针对每个像素进行单独计算的。

如图 4.15 所示，比较了 Gouraud 着色和 Phong 着色的结果。你可以看到，Phong 着色结果要平滑得多。

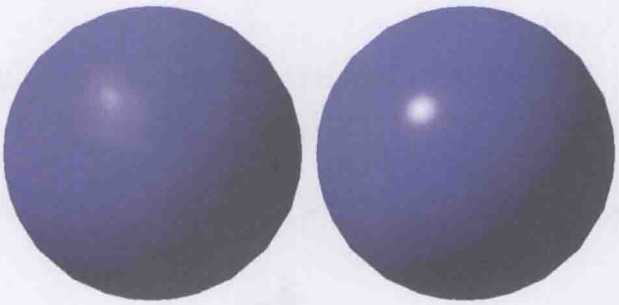


图 4.15　使用 Gouraud 着色的物体 (a) 和使用 Phong 着色的物体 (b)

有趣的是不管选用哪种着色方法，轮廓都是一样的。所以哪怕使用了 Phong 着色，低多边形对象的外轮廓还是很明显。

可见性

在你了解了网格、矩阵、灯光、光照模型、着色方法之后，3D 渲染里最重要的最后一点就是可见性判断。哪个对象可见，哪个不可见？在这个问题上，3D 游戏要比 2D 游戏复杂得多。

再探画家算法

在第 2 章中讨论了画家算法（将场景从后往前绘制），它在 2D 游戏中很好用。这是因为 2D 精灵之间的顺序关系清晰，而且大多数 2D 引擎都内在支持层级的概念。对于 3D 游戏而言，这个顺序很少是静止的，因为摄像机的透视可以改变。

这意味着在 3D 场景中使用画家算法的话，所有场景中的三角片都必须排序，可能每一帧都需要排序，摄像机在场景中移动一次就得排一次序。如果场景中有 10000 个对象，那么依据场景中的深度进行排序的计算会非常昂贵。

这样看上去画家算法非常低效，但是还有更糟的。考虑同屏显示多个视角的分屏游戏。如果玩家 A 和玩家 B 面对面，两个玩家的从后往前的顺序是完全不同的。为了解决这个问题，不仅要每帧排序两次，而且内存中还要有两个排序队列，两个都不是好的解决方案。

另一个画家算法的问题就是会引起大量的**重绘**，有的像素每帧会绘制多次。如果你想做第 2 章中图 2.3 的太空场景，一些像素每帧都会绘制多次：一个是星星区域，一个是太阳，一个是小行星，一个是太空船。

在现代 3D 游戏中，计算最终光照和贴图的过程是渲染管线最昂贵的部分。像素重绘意味着前一次绘制的花费被浪费了。因此，大多数游戏都尽力避免重绘的发生。而在画家算法中这是不能完成的任务。

而且，三角片重叠的时候也有问题。看一下图 4.16 中的 3 个三角片。哪个在前，哪个在后？

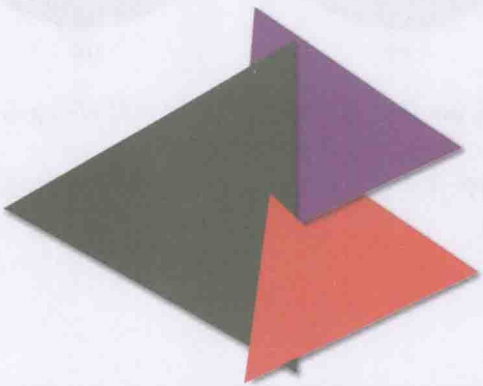


图 4.16 三角形重叠的情况

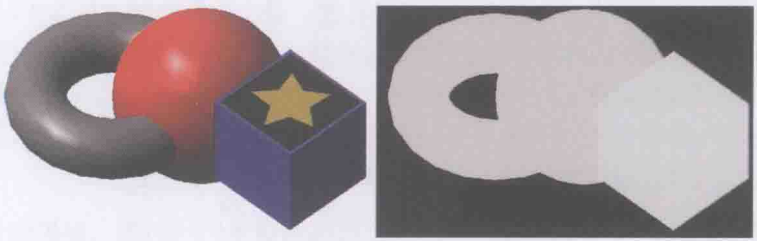
答案就是没有一个三角片在最后。在这种情况下，画家算法唯一的方法就是将这个三角片按照正确的方式切割成更小的三角片。

由于以上各种问题，画家算法在 3D 游戏中非常少用。

深度缓冲区

在**深度缓冲区**中，会有一块额外的缓冲区仅在渲染的过程中使用。这块额外的缓冲区称为**深度缓冲区**，为场景中的每个像素存储数据，就像颜色缓冲一样。但是跟颜色缓冲区存储颜色不同，深度缓冲区存储的是像素到摄像机的距离（或**深度**）。更准确地讲，每帧用到的缓冲区（包括颜色缓冲区、深度缓冲区、模板缓冲区等）统称为**帧缓冲**。

在使用深度缓冲区的每一帧渲染开始之前，深度缓冲区都会清空，确保当前深度缓冲区中的所有像素都无限远。然后，在渲染的过程中，像素的深度会在绘制之前先计算出来。如果该像素的深度比当前深度缓冲区存储的深度要小，那么这个像素就进行绘制，然后新的深度值会写入深度缓冲区。图 4.17 演示了一个深度缓冲区。



场景 **深度缓冲区**

图 4.17 一个示例场景和它相应的深度缓冲区

所以每帧第一个绘制的对象总是能够将它的所有颜色和深度分别写入颜色缓冲区和深度缓冲区。但是当绘制第二个对象的时候，那些比已有像素更远的像素都不会进行绘制。清单 4.2 展示了该算法的伪代码。

清单 4.2 计算深度缓冲

```
// zBuffer[x][y] 存储像素深度
foreach Object o in scene
   foreach Pixel p in o
      float depth = calculate depth at p
      if zBuffer[p.x][p.y] > depth
         draw p
         zBuffer[p.x][p.y] = depth
      end
   end
end
```

通过深度缓冲，场景可以以任意顺序绘制，如果场景中没有透明的物体，那么绘制结果一定正确。这也不是说顺序不相关，比如说，如果场景从后往前绘制，你每帧会绘制很多无用的像素。只是深度缓冲方法背后的思想是无须对场景进行排序，这样会显著提升性能。而且由于深度缓冲针对像素工作，而不是每个对象，因此哪怕遇到图 4.16 中三角片重叠的情况也没问题。

但也不是说深度缓冲可以解决所有可见性问题。比如说，透明对象在深度缓冲中就不太适用。假设有一摊半透明的水，水底有石头。如果使用纯深度缓冲的方法先画水体，那么水体会写入深度缓冲，这样就会影响石头的绘制。为了解决这个问题，应用深度缓冲先画所有不透明的物体。然后可以关掉深度缓冲写入功能，渲染所有透明物体。但为了确保在不透明物体背后的对象不进行渲染仍需进行深度检查。

像颜色一样，深度缓冲的表示方法也有固定的几种。最小的深度缓冲区为 16 位，可是在节省内存的同时也带来了一些副作用。在**深度值冲突**的时候，两个来自不同对象的像素挨得非常近，而且离摄像机很远，会产生每帧交替出现前后像素切换的情况。16 位浮点数精度不够高导致在第 1 帧时像素 A 比像素 B 的深度要低，而在第 2 帧比像素 B 深度要大。为了避免这种情况，大多数现代游戏都会采用 24 位或者 32 位的深度缓冲区。

还有很重要的一点是，仅依靠深度缓冲并不能解决像素重绘的问题。如果你先画了一个像素，然后才发现该像素不该画，那么前一个像素的渲染就纯属浪费。一个解决方案就是采用**先进行深度的 pass**[1]。将深度计算与着色分开，在最终的光照计算 pass 之前完成深度计算，但是实现它超出了本书的话题。

深度缓冲的检查是基于像素的，这一点要铭记在心。举个例子，如果有一棵树完全被建筑遮挡，深度缓冲还是会对这棵树的每个像素进行测试。为了解决这类问题，商业游戏经常使用复杂的**剔除**或者**遮挡**算法去消除那些在某些帧完全看不到的对象。类似的算法有二叉树分区算法（BSP）、入口算法和遮挡体积，这些都超出了本书的话题。

再探世界变换

虽然本章已经讲了很多关于渲染到 3D 场景的内容，但是还有一些值得思考的问题。第一个问题是关于内存的，如果平移、缩放和旋转必须独立修改，那么矩阵就需要分开存储，每个矩阵 16 个浮点数，3 个矩阵就有 48 个浮点数。要减少平移和缩放相对简单，只要将平移存储为 3D 向量，缩放存储为一个标量 (假设游戏只支持统一的缩放) 即可。然后这些值可以在最后的计算阶段才组成矩阵。但是旋转该怎么处理呢？

[1] 一帧画面可以多次渲染，每次为一个 pass，如果某个 pass 什么颜色都没有输出，只输出深度，那么它就是这里说的先进行深度的 pass。——译者注

将旋转存储成欧拉角会有个大问题，主要是因为它们不够灵活。假设有一艘太空飞船在模型坐标系中头部朝向 z 轴，想将它旋转到朝任意点 P。为了在欧拉角中完成这个旋转，你需要判断这个旋转的角度。可是这个旋转不只影响一个轴，需要多个轴配合旋转才能完成。这就导致计算比较困难。

另一个问题就是，如果欧拉角关于某个轴旋转 90°，同时也会影响到原来的轴的朝向。举个例子，如果一个对象关于 z 轴旋转 90°，那么 x 轴、y 轴会重叠，原有角度关系就会丢失。这就称为**万向锁**，如图 4.18 所示，它非常容易让人迷惑。

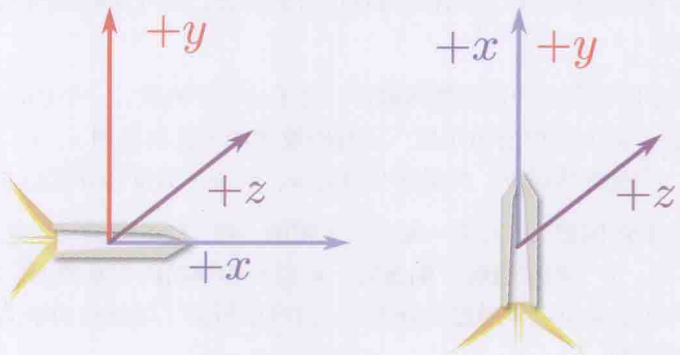


图 4.18　关于 z 轴 90° 旋转会产生万向锁

最后一个是关于将两个朝向进行插值的问题。假设游戏有指向下一个目标的箭头，一旦这个目标达成，这个箭头就应该转向下一个节点。由于视觉原因，箭头不应该突然就指向新节点，它应该有个一两秒的过渡。虽然通过欧拉角可以完成，但很难使得这个插值好看。

由于种种限制，用欧拉角表示世界空间的旋转不是一个好方法。可以采用其他的数学表示方式来做这件事。

四元数

像一位数学家所描述的，四元数相当直白，同时也相当让人迷惑。但是对于游戏程序员而言，只要认为**四元数**表示关于任意轴旋转就可以了。因此有了四元数，再也不是受限于关于某个轴旋转，而是关于任意你想要的轴旋转。

四元数的另外一个优点就是很容易在两个四元数之间进行插值。有两种插值方式：线性插值和**球形插值**。球形插值比起线性插值更加准确，同时计算上也可能更加昂贵，取决于系统。但不管哪种插值类型，之前提到的将箭头指向目标的问题通过四元数都能够轻松解决。

四元数只需要用 4 个浮点数来存储信息，这样更加节省内存。所以就像位置和统一缩放可以分别存储为 3D 向量和浮点标量一样，物体的旋转也可以存储为四元数。

不管哪种用途，游戏中的四元数全部都是**标准四元数**，就像单位向量一样，四元数的长度为1。一个四元数有着一个向量和一个标量，经常都写为 $q = [q_v, q_s]$。向量和标量的计算由旋转的轴 $\hat{a}$ 和旋转的角度 θ 决定：

$$q_v = \hat{a} \sin \frac{\theta}{2}$$
$$q_s = \cos \frac{\theta}{2}$$

要注意的是旋转的轴是必须正规化的。如果没有这样做，你可能会发现物体以奇怪的方式缩放。如果你刚开始使用四元数，物体以奇怪的方式拉扯，很可能就是因为四元数没有以正规化的轴向来创建。

大多数3D游戏数学库都会内建四元数函数库。对于这些库来说，一个CreateFromAxisAngle或者类似的函数会自动为你构造四元数。还有些数学库可能会使用 x、y、z 和 w 分量的形式来表示四元数。在这种情况下，向量部分就是 x、y、z，标量部分就是 w。

现在回顾一下太空船的问题。你有着一架朝向 z 轴的飞船，新的朝向可以通过飞船的位置和目标的位置 P 求得。为了确定绕哪个轴旋转，可以在初始朝向与新的朝向向量之间进行一个叉乘运算。朝向间的夹角可以通过点乘求得。这样就得到了要旋转的轴向和角度，紧接着就可以构造这个四元数。

还可以一个接一个地使用四元数运算。为了这么做，四元数应该以相反的顺序乘在一起。所以如果一个物体先旋转 q 然后旋转 p，那么乘积就是 pq。两个四元数相乘采用 **Grassmann积**：

$$(pq)_v = p_s q_v + q_s p_v + p_v \times q_v$$
$$(pq)_s = p_s q_s - p_v \cdot q_v$$

与矩阵一样，四元数也存在逆四元数。幸运的是，计算标准四元数的逆只需要将向量分量取逆即可。将向量部分取反也成为**共轭四元数**。

由于有逆四元数，那么也有单位四元数。定义如下：

$$i_v = \langle 0, 0, 0 \rangle$$
$$i_s = 1$$

最终，由于GPU还是需要世界变换通过矩阵的形式给出，四元数会在某个时刻变换成矩阵。将四元数变换成矩阵是比较复杂的，所以这里就不演示了。通常3D数学库都会有这个函数。

3D 游戏对象的表示

在有了四元数之后，平移、缩放和旋转都可以用比 4×4 矩阵小的数据表示。这些就是 3D 游戏对象的世界变换应该存储的。然后到了需要将世界变换矩阵传递给渲染代码的时候，可以临时构造一个矩阵。这意味着如果代码要更新对象的位置，只要改变位置向量即可，而不是整个矩阵（见清单 4.3）。

清单 4.3　3D 游戏对象

```
class 3DGameObject
   Quaternion rotation
   Vector3 position
   float scale

   function GetWorldTransform()
      // 先缩放，后旋转，最后平移，顺序很重要
      Matrix temp = CreateScale(scale) *
                    CreateFromQuaternion(rotation) *
                    CreateTranslation(position)
      return temp
   end
end
```

总结

本章讲了 3D 图形的很多方面。坐标系空间是一个重要的概念，它会表示模型在世界中和摄像机中的表现，以及如何将 3D 场景平铺到 2D 的平面上。场景中可能会有多种光照，而且使用 BRDF 决定光照如何作用在场景上，比如 Phong 光照模型。3D 引擎不再采用画家算法，而是使用深度缓冲来判断哪些像素可见。最后，使用四元数是比欧拉角表示旋转要好的方法。

关于渲染的话题还有很多内容可以讲。今天的大多数游戏都使用高级的渲染技术，比如法线贴图、阴影、全局光照、延迟渲染等。如果你对渲染很感兴趣，还有很多东西要学。

习题

1. 为什么游戏中的模型使用三角片来表示？
2. 描述一下渲染管线中的 4 种主要的坐标系。
3. 创建一个世界变化矩阵，平移到 $\langle 2,4,6\rangle$ 而且关于 z 轴旋转 90°。
4. 正交投影和透视投影有什么不同？

5. 环境光和方向光有什么不同？分别给出例子。
6. 描述 Phong 光照模型的 3 种分量。为什么被认为是一种局部光照模型？
7. Gouraud 着色和 Phong 着色之间有什么不同？
8. 在 3D 场景使用画家算法有什么问题？深度缓冲又是怎么解决这个问题的？
9. 为什么四元数在旋转的表示上要比欧拉角优秀？
10. 构造一个关于 x 轴旋转 90° 的四元数。

相关资料

Akenine-Möller, Tomas, et. al. ***Real-Time Rendering (3rd Edition).*** **Wellesley: AK Peters, 2008.** 这是一本图形渲染的资料大全。虽然最新的版本也是几年前出版的，但是它始终是迄今为止对图形程序员而言最佳的渲染读物。

Luna, Frank. ***Introduction to 3D Game Programming with DirectX 11.*** **Dulles: Mercury Learning & Information, 2012.** 讲渲染的书经常都会关注到特定的 API，所以有很多 DirectX 和 OpenGL 的书。Luna 从 2003 年起就开始写 DirectX 相关的书，在我刚开始成为游戏程序员的时候就从他的早期作品中学到了很多内容。所以如果你打算使用 DirectX 来开发游戏（只为 PC 平台），我会推荐这本书。

第5章 游戏输入

没有输入，游戏就成为了一种静止的娱乐形式，就像电影和电视那样。当游戏响应键盘、鼠标、手柄等输入设备的时候，游戏才能够互动。

本章会讨论各种输入设备，包括那些在移动设备上常见到的。还会进一步讨论高层输入系统的设计和实现。

输入设备

如图 5.1 所示，有着各种各样的输入设备，有的很常见而有的很少接触。虽然键盘、鼠标和手柄对于 PC 和家用机游戏来说是最常见的，但是在移动设备上则是以触摸和重力感应为主。还有最近的用于虚拟现实增强的设备，比如 WiiMote、Kinect 和吉他等。有几种核心技术任何设备查询信息的时候都能用到。

图 5.1 多种多样的输入设备

绝大多数的输入形式可以分为两种：数字与模拟。**数字**形式的输入就是那些只有两种状态的："按下"和"没有按"。举个例子，键盘上的按键就是数字的，空格键要么按下要么没按下。绝大多数的键盘不存在两种状态的中间状态。**模拟**形式的输入就是设备可以返回某个数字的范围。一个常见的模拟设备就是摇杆，它会总是返回二维空间的范围值。

一个键盘可能只有数字输入，而大多数输入设备同时有着模拟和数字输入。举个例子，Xbox 360 手柄：方向键和按钮是数字的，但是摇杆和触发器却是模拟的。值得注意的是，摇杆上的按键也不总是数字的，Playstation 2 就是个特殊的例子。

输入系统在其他方面要考虑的是游戏中对**同时按键**和**序列按键**的支持。这种需求在格斗游戏中很流行，玩家会经常同时使用按键和序列按键来搓招。比如最新的《街头霸王》，角色 Blanka 按下前和 3 个拳或者 3 个踢（同时按键），角色会马上跳起来。类似地，所有《街头霸王》游戏都有气功波的概念，能够通过下前拳来施展 (序列按键)。处理同时按键和序列按键超出了本章的主题，但是可以通过状态机的方式来识别玩家的各种输入。

数字输入

与文本类型游戏不同，图形游戏基本不用标准输入（比如 cin）。对于图形游戏来说，控制台通常都会同时关闭，所以不可能使用标准输入。大多数游戏都会采用一些程序库来进行设备级别的查询，可以使用跨平台的库 Simple DirectMedia Layer(SDL;www.libsdl.org)。

通过类似于 SDL 的库，使得查询数字设备的当前状态成为可能。在一个这样的系统中，通常都会获取一组描述每个按键的布尔数组。所以，如果按下空格键，那么空格键对应的索引就被在数组位置中置为真。要记住那个索引对应哪个按键是麻烦的，所以通常都会使用**键码**枚举来映射每个索引到按键的名字上。比如说，如果你想检查空格键的状态，你可能只要检查定义好的索引 K_SPACE。

现在假设你正在开发一款简单的游戏，游戏设定按下空格键可以让飞船发射导弹。代码可以这样写：

```
if IsKeyDown(K_SPACE)
   fire missile
end
```

但是上面代码的问题是，如果玩家很快地按下和释放空格键，上面的判定会在好几帧连续为真。这是因为如果游戏运行在 60 FPS 下，玩家可能会在 16ms 的空档中按下和释放空格键，上面代码只能在一帧中检查。我们不希望这种情况发生，每次按下空格键都能够在连续几帧发射三四个导弹。更进一步来讲，如果玩家只是一直按着空格不放，那么导弹就会每帧都发射，直到玩家释放为止。这可能是一种需求，但对游戏来讲是完全不平衡的。

一个简单但是有效的改进方法就是同时跟踪这个键在上一帧和这一帧的状态。因为现在是两个布尔值，就会总共有 4 种组合，而每一种都对应了不同的状态，如表 5.1 所示。

表 5.1　上一帧和本帧状态组合

上一帧状态	本帧状态	结论
释放	释放	一直释放
释放	按下	刚刚按下
按下	释放	刚刚释放
按下	按下	一直按着

所以，如果上一帧空格没有按下，但是这一帧按下了，就可以判断玩家“刚刚按下”。这就是说，如果导弹发射行为在“刚刚按下”空格键才发生的话，那么飞船每按下一次空格才进行一次发射。

这个系统还可以进一步提升。假设我们的游戏允许玩家长按空格键进行充能。在这种情况下，我们可以在“刚刚按下”的时候增加导弹的能量，在“一直按着”的时候不断增加能量，最后在“刚刚释放”的时候才进行发射。

为了支持这 4 种不同的结果，我们首先必须声明这 4 种枚举：

```
enum KeyState
   StillReleased,
   JustPressed,
   JustReleased,
   StillPressed
end
```

然后，当输入设备更新的时候，当前的输入需要记录下来：

```
function UpdateKeyboard()
   // lastState和currentState是在其他地方定义好的数组，
   // 记录了完整的键盘信息。
   lastState = currentState
   currentState = get keyboard state
end
```

最终，可以通过一个返回 KeyState 的函数来实现之前讨论的功能：

```
function GetKeyState(int keyCode)
   if lastState[keyCode] == true
      if currentState[keyCode] == true
         return StillPressed
      else
         return JustReleased
      end
   else
      if currentState[keyCode] == true
         return JustPressed
      else
         return StillReleased
      end
   end
end
```

通过这个函数，就可以检查每个按键的 KeyState。在稍后，我们会基于此开发一个事件驱动的系统。现在，我们先看一下模拟设备。

模拟输入

因为模拟设备有一组范围值，有输入偏差是很常见的。假设一个摇杆有 x 值和 y 值用 16 位有符号整形表示。这意味着 x 和 y 都可以表示-32768 到 +32768 范围的值。如果一个摇杆放在平面上，玩家不去碰它，理论上由于摇杆没有被控制，这个 x 和 y 应该都为 0。但是，实际上该值会在 0 的附近。

由于这个原因，很少会让模拟输入直接应用到角色的移动上，这会让角色永远停不下来。这意味着即使你放下手柄，角色还会在屏幕上到处乱走。为了解决这个问题，大多数游戏都会采用某种**模拟输入过滤**的方式，用于消除输入的偏差值。图 5.2 演示了这种过滤可以采用**无效区域**的方法实现，就是让摇杆输入在中心范围内无效。

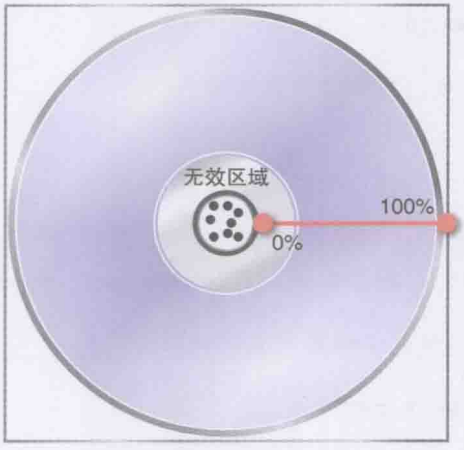


图 5.2　摇杆无效区域使得在内环范围的输入无效

一个简单的无效区域实现就是在 x 和 y 值接近 0 的时候直接设置为 0。如果值的范围在 +/–32K，也就是无效区域范围大约 10% 的区间：

```
int deadZone = 3000
Vector2 joy = get joystick input

if joy.x >= -deadZone && joy.x <= deadZone
   joy.x = 0
end

if joy.y >= -deadZone && joy.y <= deadZone
   joy.y = 0
end
```

但是这个方法有两个问题。首先无效区域是一个正方形而不是圆形。也就是说，如果摇杆的 x 和 y 都稍微比无效区域小一点，角色还是不会移动，哪怕实际上已经超过了 10% 的阈值了。

另一个问题就是没有完全利用所有有效值范围。从 10% 到 100% 都有速度，但是小于 10% 就什么都没有了。我们采用一种解决方案，使得 0% 到 100% 都有速度。换句话说，我们希望有效输入映射到 0% 的速度，而不是映射到 10%。这样会将原来的 10% 到 100% 的中值 55% 映射到 50%。

为了解决这个问题，我们不应该将 x 和 y 看作独立的分量，而是将摇杆的输入视为一个 2D 向量。然后就可以通过向量运算来完成无效区域的过滤。

```
float deadZone = 3000
float maxValue = 32677
Vector2 joy = get joystick input
float length = joy.length()

// 如果长度小于无效区域，那么认为没有输入
if length < deadzone
   joy.x = 0
   joy.y = 0
else
   // 计算无效区域到最大值之间的百分比
   float pct = (length - deadZone) / (maxValue - deadZone)

   // 正规化向量，然后相乘得到最终正确结果
   joy = joy / length
   joy = joy * maxValue * pct
end
```

让我们测试一下上面的伪代码。介于 32677 和 3000 的中间点是 17835，这个长度的向量应该小于等于 50%。这个向量可以是 x=12611 和 y=12611。由于 17835 的长度比无效区域长，那么会走 else 分支，从而会有如下计算：

$$\text{pct} = (17835 - 3000)/(32677 - 3000)$$

$$\text{pct} \approx 0.5$$

$$\text{joy}(x, y) = (12611/17835, 12611/17835))$$

$$\text{joy}(x, y) \approx (0.71, 0.71)$$

$$(0.71, 0.71) \times 32677 \times 0.5 \approx (11600, 11600)$$

在应用模拟过滤之后，得到的向量长度为 16404。换句话说，一个 3000 与 32677 之间 50% 的值，被调整为 0 与 32677 之间的 50%，这样就能够得到从 0% 到 100% 的移动速度范围。

基于事件的输入系统

想象你开车带小孩去儿童乐园。小朋友非常兴奋，所以每分钟都问一次“我们到了吗?”。她不断地问直到你最终到达目的地，这个问题的答案才为“是”。现在想象不止一个小孩，你载着一车小孩去儿童乐园，所以每个小孩都来不断地问这个问题。这不仅影响驾驶，还很浪费精力。

这个场景本质上是个**轮询**系统，每个小孩都来轮询结果。对于输入系统来说，很多代码都想轮询输入，比如主动去检查空格键是否“刚刚按下”。每一帧的多个地方都不断地用 GetKeyState 查询 K_SPACE。这不仅导致需要多写代码，而且还会容易产生更多的 Bug。

我们回到儿童乐园的例子，想象你再次面对一车的孩子。比起让孩子不断地重复提问，不如雇佣一个**事件机制**或者**推送系统**。在基于事件的系统中，小孩需要注册他们关心的事件（这个例子中，关心的事件就是到达儿童乐园），然后会在事件发生的时候通知到他们。现在就可以安静地开车到儿童公园了，只要你到达之后通知所有小孩已经达到就可以了。

输入系统同样也可以设计为基于事件的。可以设计一个接受特定输入事件的系统。当事件发生的时候，系统就通知所有已经注册的代码。所以所有需要知道空格键状态的系统都可以注册关于空格键“刚刚按下”的事件，它们会在事件发生后马上收到通知。

要注意的是，基于事件的输入系统还是要轮询输入的。就好像你作为一个司机，必须在行车的过程中时刻了解是否到达目的地，然后通知孩子们一样。输入系统要不断地轮询空格键，这样就可以正确地发出通知。与之前不同的地方就是，现在我们只有一个地方进行轮询。

基础事件系统

有不少语言，像 C# 那样的，原生就支持事件机制。但是，如果你选择的语言没有内建支持，实现一个事件系统也不难。几乎所有语言都支持将函数作为变量，还有的可以通过匿名函数、lambda 表达式、函数对象或者函数指针来实现。随着函数可以作为变量，就可以得到一个列表关联上具体的事件。每当事件被触发，那些注册了的函数都会被调用。

假设你想实现一个简单的用于鼠标点击的事件驱动系统。游戏的很多逻辑都希望在鼠标点击时获得通知，并且得到鼠标在屏幕上的位置。在这种情况下，就会有一个“鼠标管理中心”的需求，它能让关心鼠标事件的系统去注册。这个类的声明如下：

```
class MouseManager
   List functions

   // 接受那些将传递的参数(int, int)作为信号的函数
   function RegisterToMouseClick(function handler(int, int))
      functions.Add(handler)
   end

   function Update(float deltaTime)
      bool mouseClicked = false
      int mouseX = 0, mouseY = 0

      // 轮询鼠标点击
      ...
```

```
        if mouseClicked
            foreach function f in functions
                f(mouseX, mouseY)
            end
        end
    end
end
```

然后那些对鼠标点击感兴趣的部分都可以通过调用RegisterToMouseClick进行注册，代码如下：

```
// 为鼠标点击注册myFunction
MouseManager.RegisterToMouseClick(myFunction)
```

在注册之后，这个函数就会在事件发生之后自动被调用。让MouseManager可以全局访问的最好方式就是**单件**设计模式，就是整个程序都可以通过一个地方访问得到这个实例。

鼠标管理器使用的事件机制可以轻易扩展到按键处理，这个工作量无非就是为关心的按键注册函数。但是这个基础的事件机制会有不少问题。

虚幻引擎中的输入系统

虚幻引擎面对玩家输入开发了一个可自定义的事件驱动系统。所有按键绑定都存储在ini文件中，每个绑定的按键都直接映射到一个或者多个虚幻脚本语言写好的函数中。比如说，如果有个脚本函数叫Fire，能够使玩家发射武器，通过以下绑定可以同时映射到鼠标左键和手柄的右边触发器：

```
Bindings=(Name="LeftMouseButton",Command="Fire")
Bindings=(Name="XboxTypeS_RightTrigger",Command="Fire")
```

默认情况下，这些绑定会在“刚刚按下”的时候触发。让不同动作在按键“刚刚释放”的时候执行也是可以的，只要用“onrelease”修饰就可以了。比如说，接下来的绑定能够让函数StartFire在鼠标左键按下的时候调用，而在释放的时候调用StopFire：

```
Bindings=(Name="LeftMouseButton",Command="StartFire | onrelease
    StopFire")
```

这个管道操作可以映射到任何绑定函数，这对提升灵活性很有帮助。事件系统不是用于处理UI事件的。对于UI来说，虚幻提供了UpdateInput函数，可以通过重载完成你想要的行为。更多关系虚幻引擎绑定系统，可以查看Epic官网http://udn.epicgames.com/Three/KeyBinds.html。

当前系统的一个问题是具体的案件与事件耦合度比较高。这就是说那些感兴趣的系统必须注册到特定的按键（比如鼠标左键）而不是某个抽象的动作（比如武器发射）。如果玩家想

要发射的时候用空格键而不是鼠标左键呢？这种情况下，所有直接注册到鼠标左键的代码都不能正确工作了。所以理想的输入系统可以支持抽象的动作绑定到特定的按键。

上述的事件系统还有一个问题就是不允许一个注册的系统阻止另一个系统接收消息。假设你在开发简单的 RTS 游戏（比如第 14 章的塔防游戏）。如果游戏是在 PC 平台上的，那么同时支持鼠标点击 UI（菜单）和鼠标点击游戏世界（选择单位）是很合理的。由于 UI 在游戏世界的上层，所以 UI 应该能够优先得到鼠标点击的通知。这意味着当鼠标点击时，UI 系统需要先判断是否点中自己。如果点击到自己，那么事件不应该再往下派发到游戏世界。在我们现有的输入系统中，游戏世界和 UI 都会接收到鼠标事件，这样会导致潜在的意外发生。

一个更复杂的系统

本节讨论的系统不允许直接将函数注册到事件当中。它通过创建一系列输入动作的方式，让感兴趣的系统查询这些动作。所以虽然在严格意义上来讲，它不是一个基于事件的系统，但是也有点类似。这个系统与第 14 章的塔防游戏中的输入系统非常类似。

这个系统依赖于按键映射的字典。回忆一下字典，字典其实就是一系列的键值对组成的集合，而是否排序取决于字典的类型。在这个系统中，键就是绑定的名字（比如“Fire”），而值就是按钮的相关信息，包括哪个按钮以及触发的时机（“刚刚按下”、“一直按着”等）。由于我们希望支持绑定多种行为（比如“Fire”可以是左键或者空格键），我们要选中一种能够支持键值重复的字典。

每一帧，输入系统都检查这个字典的每一个键值并判断哪个键值激活。那些在这帧中激活的绑定会存储到一个激活按键绑定字典中。这个激活的绑定字典首先会提供给 UI 系统，UI 系统就会判断哪些 UI 会受到这些动作影响。接下来剩下的绑定就会传递到游戏世界中再进一步处理。图 5.3 演示了绑定怎么在两个系统间传递。

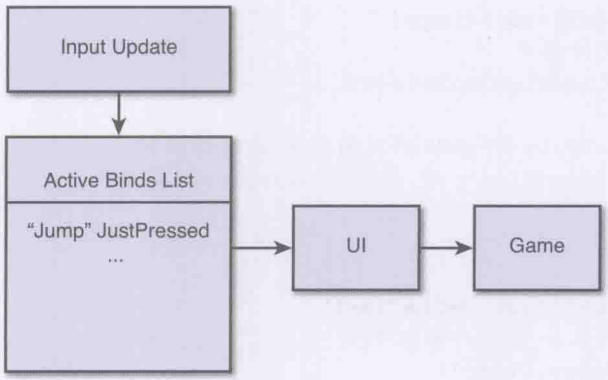


图 5.3　本节提出的输入系统

虽然游戏中的所有绑定，包括“Jump”，都传递给UI看起来有点奇怪，但是这样提供了很好地灵活性。比如说，游戏中的教程经常都会有一个对话框，在玩家按键后才进入下一步。在这种情况下，你可以通过“按键跳过”对话框实现在按键时才进行跳过。

这个系统的代码也不是非常复杂，如清单5.1所示。如果你想看一个具体实现的版本，可以看InputManager.cs，在第14章的塔防游戏中。

清单5.1 一个更复杂的输入系统

```
// KeyState枚举（JustPressed、JustReleased等）在本章的前面有定义
...
// GetKeyState在本章的前面有定义
...

// BindInfo就是字典中的值
struct BindInfo
   int keyCode
   KeyState stateType
end

class InputManager
   // 存储了所有绑定
   Map keyBindings;
   // 只存储那些激活的绑定
   Map activeBindings;

   // 使按键绑定更加方便
   function AddBinding(string name, int code, KeyState type)
      keyBindings.Add(name, BindInfo(code, type))
   end

   // 初始化所有绑定
   function InitializeBindings()
      // 可以从文件解析
      // 然后调用AddBinding进行绑定

      // 比如说，"Fire"可以映射到刚刚释放的回车键
      // AddBinding("Fire", K_ENTER, JustReleased)
      ...
   end

   function Update(float deltaTime)
      // 清除所有上一帧中的激活绑定
      activeBindings.Clear()
```

```
        // KeyValuePair有key和value成员，分别是来自字典键和值
        foreach KeyValuePair k in keyBindings
            // 使用前面定义的GetKeyState得到某个按键的状态
            if GetKeyState(k.value.keyCode) == k.value.stateType
                // 如果与绑定一致，则认为绑定是被激活的
                activeBindings.Add(k.key, k.value)
            end
        end

        // 如果有任何的激活绑定，那么优先发送给UI系统
        if activeBindings.Count() != 0
            // 发送激活绑定给UI
            ...

            // 发送激活绑定给游戏的剩余部分
            ...
        end
    end
end
```

这个系统有一个效率不高的地方，那就是如果有多个按键的同一个按键状态处于绑定，那么每帧都会检查多次。这个完全可以进行优化，但是这里为了简单而将优化去掉了。

这个系统还可以扩展为将函数映射到绑定，这也是 UI 系统或者游戏系统的可选方案。但是对于塔防游戏来说，这么做就会过度设计。如果游戏真的没有必要这么灵活，就没有理由让输入系统更加复杂。

移动设备输入

大多数的智能机和平板设备都有着与传统的键盘、鼠标、手柄不同的输入机制。正因如此，增加一个讨论移动设备输入的小节是不错的。

很容易注意到游戏有很多种输入方式，但是大多数移动游戏都不应该使用它们。移动游戏需要很小心地不要使用过于复杂的操作，这会吓跑很多潜在玩家。

触屏和手势

触屏允许玩家通过手指与屏幕交互。最基本的情形，如果游戏允许玩家只用一根手指点击操作，那么实现起来就和鼠标一样。但是大多数移动设备都支持**多点触摸**，就是说允许用户同时多个手指进行操作。

一些游戏通过多点触摸实现**虚拟手柄**，这会有一个虚拟摇杆和虚拟按键让玩家交互。这个做法在家用机版本的游戏上很流行，但有时候会让玩家很困惑，因为没有真实手柄的反馈。

有一些操作只有触摸屏才能实现，称之为**手势操作**，就是一系列的触摸引发的行为。一个很流行的手势操作就是 iOS 设备上的“两指缩放”，用户可以用食指和拇指同时靠近来进行缩小，同时远离进行放大。

幸运的是，iOS 和 Android 都有提供系统级别的手势识别支持，包括“两指缩放”。iOS 上可以继承 UIGestureRecognizer 类来实现，会有一个函数告诉你出现了什么手势。而在 Android 上，有 android.gesture 可以做同样的事情。

对于自定义手势，实现会更难一些。在 iOS 上，我们要自己写一个 UIGestureRecognizer 的子类来完成手势识别的功能。而在 Android 上，有个很方便的程序“Gestures Builder”可以帮助我们创建自定义手势。

检测手势有很多种方法，其中一种流行的算法就是 **Rubine 算法**，它在 Dean Rubine 在 1991 年的论文“Specifying gestures by example.”中提出。虽然 Rubine 算法是由笔画识别发展出来的，但是不管是单个手指的手势识别还是多个手指的手势识别，它都可以实现。

为了应用 Rubine 算法，我们需要先创建手势识别库。这个库的创建方式是通过一个能画出手势的测试程序来完成的。手势的绘制，有 14 个手势的数学属性，称之为**功能**，是被计算出来的。这些功能的元素包括手势绘制的总时长、总距离、手势的区域、手势的中点、初始点等。图 5.4 展示了原版 Rubine 算法的多个功能。

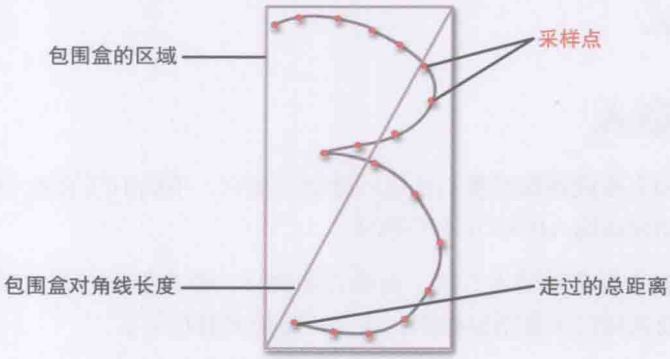


图 5.4 一些 Rubine 算法的功能

在某个特定手势的功能计算之后，会存储到库中。在库完成之后，就是可以在程序中加载这些手势并进行识别。然后当用户开始输入手势时，就会计算这个手势对应的数学属性。在用户完成输入后，这个算法就会开始分析然后判断这是哪个手势。

虽然大多数游戏都不会用到这些高级手势，但还是会有一些情况会用到。比如说任天堂 NDS 上的《脑白金》会要求用户写入数字来解答数学题。另一个例子就是教育游戏，用于教小朋友怎么写字母。

加速器和陀螺仪

加速器和陀螺仪是移动设备用于判断用户细微移动的主要机制。早期的 iOS 和 Android 设备只有加速器，而现在所有设备两者都有。虽然我们在一起讨论它们，但是它们实际上用途是不一样的。

加速器检测设备坐标系轴向上的加速度。在 iOS 设备上，如果你竖着拿手机，y 轴朝屏幕上，x 轴朝屏幕右，z 轴朝屏幕外的方向。举个例子，如果设备竖立持握，然后向上移动，会得到沿着 y 轴的加速度。这些轴如图 5.5(a) 所示。

由于加速器检测加速度，总会有一个常量添加到设备上：重力。这意味着如果设备为空闲状态，加速器可以粗略地通过重力的方向检测设备的朝向。因此，如果设备竖立持握，重力会作用于 y 轴，而如果水平持握，重力会作用于 x 轴。

而**陀螺仪**可以检测设备关于设备轴向的旋转，如图 5.5(b) 所示。这意味着通过陀螺仪很容易测量设备的朝向。举一个非游戏的例子，陀螺仪在检测水平情况，比如判断墙上的照片是否水平。

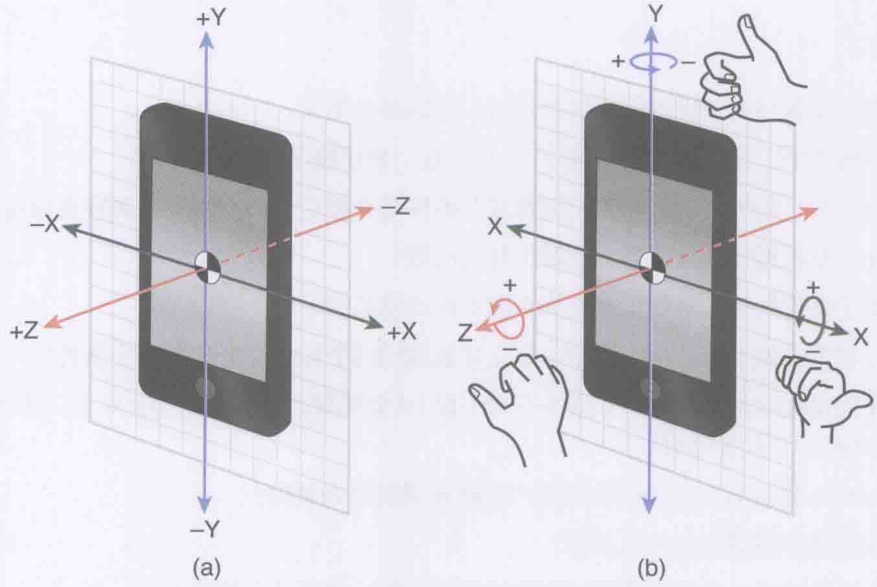


图 5.5　加速器 (a) 和陀螺仪 (b)

从原始的加速器和陀螺仪数据挖掘出有意义的信息是有挑战性的，所以设备 SDK 通常都会有更加高级的功能来使得功能更加易用。比如说，在 iOS 上可以用 CMDeviceMotion 来得到重力向量、用户加速度和**朝向**（设备的朝向）。而这些分量都通过数学的形式表达，比如重力和用户加速度用 3D 向量表示，朝向可以用欧拉角、四元数或者矩阵表示。

其他移动设备输入

一些类型的游戏可能会扩展和使用更多移动平台才有的输入。比如说，一个现实增强游戏可能会使用摄像机的数据然后在物理世界上绘图。还有一些游戏使用 GPS 来查看附近是否有玩同款游戏的玩家。还有的需要我们将穿戴式设备戴着走的（比如虚拟实现设备）。虽然这些领域近几年的发展速度激动人心（比如 Oculus Rift），VR 想要成为主流游戏还有很远的路要走。

总结

因为每个游戏都必须有输入，这是个很基本的概念。数字和模拟设备都有多种不同的形式，但是进行查询的方式都是类似的。在数字设备上，我们通常想要将二进制的开和关转换成有意义的形式。而模拟设备，我们通常都要对输入区间进行过滤。在更高的层面上，我们希望能够通过隔离输入事件、抽象的行为来控制角色，这就是我们基于事件的输入系统。

习题

1. 数字设备和模拟设备的区别是什么？分别举个例子。
2. 在输入中，什么是“同时按下”，什么是“顺序按下”？
3. 为什么“刚刚按下”和“刚刚释放”事件是必须的，而不是直接去查询设备？
4. 什么是模拟输入过滤，它能解决什么问题？
5. 基于事件的输入系统比起轮询系统有什么优势？
6. 在基本的基于事件的系统中，怎么才能跟踪哪个函数注册到了哪个事件？
7. 在用鼠标输入的游戏中，点击行为不仅 UI 会处理，游戏世界也会处理，什么方法能够保证 UI 优先处理？
8. Rubine 算法尝试解决什么问题？粗略地讲讲怎么解决？
9. 加速器和陀螺仪有什么不同？
10. 什么样的输入机制是现实增强游戏会用到的，用于什么地方？

相关资料

Abrash, Michael. Ramblings in Valve Time. http://blogs.valvesoftware.com/abrash/. 这个博客的博主是游戏产业的传奇人物 Michael Abrash，讲了很多有洞见的虚拟实现和现实增强的文章。

Rubine, Dean. "Specifying gestures by example." In SIGGRAPH '91: Proceedings of the 18th annual conference on computer graphics and interactive techniques. (1991): 329–337. 这是 Rubine 手势识别算法的原版论文。

第6章 声音

虽然很容易被忽略，但是声音是游戏的重要组成部分。无论是游戏的提示音效还是整体营造的背景音效，如果没有声音的支持，游戏会大打折扣。为了感受音效的重要性，可以尝试将你最喜欢的一款游戏静音——没有声音之后，感觉会大打折扣。

本章首先讲如何将原始数据转换成代码控制的音效。然后讨论更高级的声音技术，比如多普勒效应、数字信号处理和音效遮挡，它们都会在很多情况下用到。

基本声音

声音播放最基础的级别就是在游戏某个时刻播放某个音频文件。但是在多数情况下一个事件不是必须对应到一个声音上的。假设游戏有个角色在世界中奔跑。每次角色的脚碰地，就应该会有一个脚步声。如果只有一个脚步声音效不断地播放，那么很快就会变得乏味。我们更倾向于在脚碰地面时有多种脚步声随机选择播放。

还有一个要考虑的事情就是只有有限数量的**频道**可以同时播放。想象一个游戏有大量的敌人在玩家附近奔跑，如果他们全部都播放脚步声，那么有可能将所有的频道用完。有很多声音比敌人的脚步声更重要，因此我们需要将音效进行优先级排序。通过以上考虑，我们就会需要比音频文件更多的信息。正因如此，大多数游戏都存储了一组额外的数据描述了音频文件的优先级。

原始数据

原始数据是指音效设计师使用类似 Audacity（*http://audacity.sourceforge.net*）这样的工具来创建的最原始的音频文件。在脚步声的例子中，可能会有许多不同的源文件，比如 fs1.wav、fs2.wav、fs3.wav 等。一个常见的方式为将短音效存储为 WAV 格式或者其他无压缩文件格式，而存储长音效，比如声音或者对话，则会采用压缩格式比如 MP3 或者 OGG。

当在游戏中需要播放这些声音文件时，通常有两种方法。经常会让场景预加载短音效文件到内存中，这样到了播放声音的时刻就不再需要花时间到硬盘加载了。而另一种方法，由于压缩的声音或者对话文件通常会有更大的体积，通常会以**流**的方式加载。就是说当声音文件播放的时候，它的一小片段已经按需从硬盘加载进来了。为了加载和播放源文件，多个平台都会有内建的声音库（比如 iOS 上的 CoreAudio）。但是对于跨平台的支持，OpenAL（*http://kcat.strangesoft.net/openal.html*）则是一个很流行的解决方案。

声音事件

声音事件映射了一个或者多个原始数据文件。声音事件事实上是由代码触发的。所以比起直接播放 fs1.wav 文件，可能调用一个叫“footstep”的声音事件会更好。这个想法就是声音事件可以包含多个声音文件同时还能有元数据，将这些声音文件作为整体。

举个例子，假设有一个爆炸声音事件。这个事件应该随机选择 5 个 WAV 文件中的一个来播放。另外地，由于爆炸是可以在远处就听见的，所以应该有能听见的距离的元数据。而且爆炸声音事件应该具有高优先级，这样即使所有频道都用完了，爆炸还是能播放。这个声音事件的基本布局如图 6.1 所示。

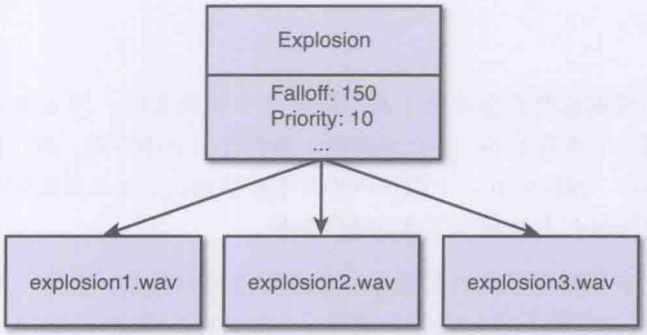


图 6.1 爆炸声音事件

游戏中的元数据有多种实现方式。一个方法是使用 JSON 文件格式，爆炸声音事件大概会像下面这样：

```
{
   "name": "explosion",
   "falloff": 150,
   "priority": 10,

   "sources":
   [
      "explosion1.wav",
      "explosion2.wav",
      "explosion3.wav"
   ]
}
```

在后面的第 11 章中我们会讲 JSON 文件格式。这里也可以用其他格式，只要跟 JSON 一样灵活就可以。在任何情况下，在解析数据的时候，都能够直接将数据映射到声音事件类：

```
class SoundCue
   string name
   int falloff
   int priority
   // 所有源文件的字符串链表
   List sources;

   function Play()
      // 随机选择一个源文件来播放
      ...
   end
end
```

之前提到的系统可能很多游戏已经够用了，但是对于脚步声的例子来说就可能不够。如果角色可以在不同的路面行走——石头、沙地、草地等，声音需要根据当前的地面来判定。在这种情况下，系统需要一些方法知道当前地面所属声音的分类，然后从中随机选择来播放。换句话说，系统需要一些方法根据当前地面而**切换**不同的音频集合。

声音事件的更高级的 JSON 文件如下：

```
{
   "name": "footstep",
   "falloff": 25,
   "priority": "low",
   "switch_name": "foot_surface",

   "sources":
   [
      {
         "switch": "sand",
         "sources":
         [
            "fs_sand1.wav",
            "fs_sand2.wav",
            "fs_sand3.wav"
         ]
      },
      {
         "switch": "grass",
         "sources":
         [
            "fs_grass1.wav",
            "fs_grass2.wav",
            "fs_grass3.wav"
         ]
      }
   ]
}
```

我们可以在原有的 SoundCue 类上增加功能。一个更好的方案是提取出 ISoundCue 接口，然后实现 SoundCue 类和新的 SwitchableSoundCue 类：

```
interface ISoundCue
   function Play()
end

class SwitchableSoundCue implements ISoundCue
   string name
   int falloff
```

```
    int priority
    string switch_name

    // 存储(string,List)键值对的哈希表
    // 比如说("sand", ["fs_sand1.wav", "fs_sand2.wav", "fs_sand3.wav"])
    HashMap sources

    function Play()
        // 得到当前值赋值到switch_name
        // 然后在哈希表中查找链表，然后随机播放一个
        ...
    end
end
```

为了让这个实现顺利工作，我们需要以全局的方式获取和设置切换。通过这样的方法，角色跑动的代码就可以判断地面然后切换到相应的脚步声。然后脚步声的声音事件 Play 函数就可以知道当前所切换的值，然后播放正确的脚步声。

最终，实现声音事件系统的关键就是有足够的配置信息去判断播放什么和怎么播放。有足够多的变量数量能够让音频设计师得到更多的灵活性从而更好地设计真实的交互。

3D 声音

虽然不是绝对，但是大多数 2D 音效都是位置无关的。这意味着对于大多数 2D 游戏，声音的输出在左右两个喇叭是一样的。有些游戏也会考虑音源的位置，比如音量随着距离衰减，但还是少数 2D 游戏才这么做。

对于 3D 音效和 3D 游戏来说，音源的位置就特别重要。大多数音效都有自己独特的随着**监听者**距离增大衰减的方式，一个例子就是游戏中的虚拟麦克风能听到附近的音效。

但也不是说 3D 游戏不使用 2D 音效。一些要素比如用户界面、解说、背景音等还在使用 2D 音效。但是出现在游戏世界中的音效通常都是 3D 音效。

监听者和发射者

不管监听者怎么监听游戏世界中的音效，**发射者**就是发射特定音效的物体。比如说，如果有一堆柴火发出噼里啪啦的声音，就会有一个声音发射器放在那个位置然后放出噼里啪啦的声音。然后基于监听者和火柴声的发射者之间的距离就可以算出音量的大小。发射者相对于监听者的朝向决定了哪个喇叭有声音，如图 6.2 所示。

图 6.2　声音监听者和发射者，这种情况下，声音应该听上去在右边

由于监听者会监听所有 3D 世界中的声音，所以摆放监听者的位置和朝向就很重要。如果监听者摆设不对，3D 声音系统会有问题——要么声音太小或者太大，要么喇叭出声音的方向不对。

对于很多种游戏来说，直接使用摄像机的位置和朝向就可以了。比如第一人称射击游戏，摄像机同时也是玩家位置的参考点，如果监听者使用同样的位置和朝向就没错了。在切场景的时候也同理，只要让监听者与摄像机保持同步即可。

虽然总是跟着摄像机的位置和朝向是很诱人的，但还是有好几种游戏不适用。比如第三人称动作游戏。假设一个游戏，摄像机的跟随距离是 15 米。有一个声音在主角脚底下播放会怎样？

如果监听者与摄像机在同一个位置，这个声音听起来就像在 15 米外。声音的类型和衰减范围等因素，可能会导致声音几乎不能听清，特别是事件在角色身旁的话就会很奇怪。现在如果由角色触发一个音效，我们通常可以认出这些音效并分辨出它们的不同。但是如果音效是由玩家附近的敌人所造，那距离问题就没有太大影响。

能想到的解决方法就是将监听者的位置和朝向设置到角色身上。虽然这样能够解决 15 米外的声音的问题，但是这样会有一个更大的问题。想象一下爆炸在摄像机与玩家之间发生的情形。如果监听者的位置和朝向继承自角色，爆炸从监听者的后面出现，而因此从后面的喇叭放出声音。进一步讲，如果游戏允许摄像机独立于角色旋转，就容易发生发射器在玩家的左侧，而在摄像机的右侧的情况，声音就可能会从左边的喇叭出来。这样就会看起来不正确，因为我们看到了爆炸在屏幕的右侧发生就会预期在右侧的喇叭播放爆炸声，而不管玩家的朝向。

最后，在混战型的第三人称游戏《指环王：征服》中，我们做了一定的妥协。首先就是我们让监听者的朝向等于摄像机的朝向，而不是等于角色的朝向。接着，与监听者位置准确地摆放在玩家身上或者摄像机身上不同，我们放在了两者之间。换句话说，监听者的位置就是摄像机位置和角色位置的差值结果，如图 6.3 所示。两者之间的百分比取决于游戏，但是通常介于 33% 和 66% 之间都能有不错的效果。

图 6.3 第三人称游戏中监听者的位置

通过这个方法，虽然声源在摄像机与角色之间时从另一个喇叭播出的可能性依然存在，但是发生的概率减小了。而虽然在角色身上播放的声音距离不为 0，但是这个距离比摄像机要近一些。最后，监听者的朝向不会出问题，因为它继承了摄像机的朝向。对于一些游戏来说，正确的选择就是总是将监听者的位置放在摄像机身上。但是，由于我们的游戏注重混战，因此放在摄像机上就不太合适。

衰减

衰减描述了音效的音量随着远离监听者会如何减小。可以用任何可能的函数去表达衰减。但是，由于音量的单位**分贝**是一个对数刻度，线性衰减会产生对数变换关系。这种线性分贝衰减函数通常都是默认方法，但是显然不是唯一的方法。

就像点光源一样，可以增加更多的参数。我们可以设定在内半径之内衰减函数不起作用，而外半径之外就完全听不到声音。这样的声音系统允许音频设计师创建多种衰减函数来展现出随着不同距离有不同的音效。

环绕声

不少平台都不支持环绕声的概念——大多数移动设备最多只支持立体声。但是，PC 和家用机游戏是可以有两个以上喇叭的。在 **5.1 环绕**系统中，会有总共 5 个正式的喇叭和 1 个用于表现低频效果的低音炮。

传统的 5.1 配置是放置 3 个喇叭在前面，两个在后面。前面的喇叭放置在左、中、右。而后面的喇叭只有左、右。低音炮的位置不太重要，不过放在室内角落会让低频效果更好。图 6.4 演示了 5.1 环绕系统的布局。

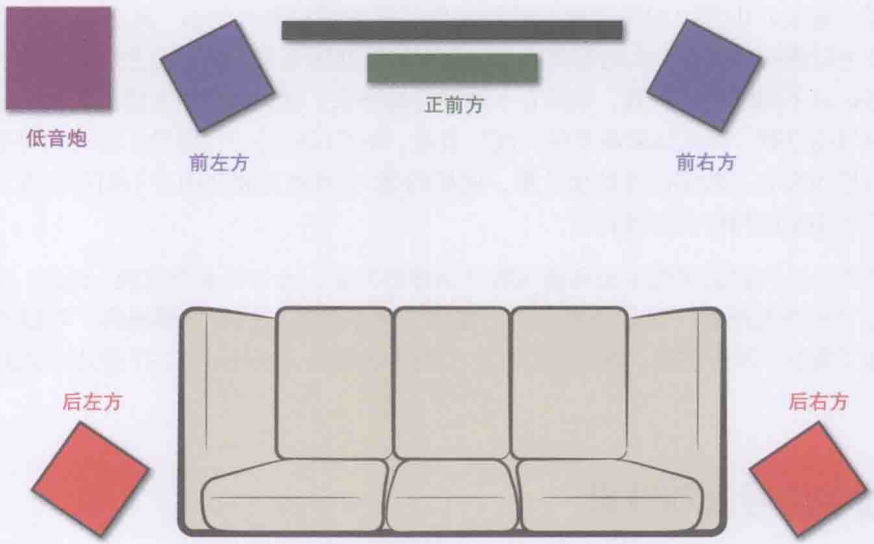


图 6.4　一个标准的 5.1 环绕声配置

5.1 配置的优点就是会感受到更多方向的声音。如果一艘太空船在头顶上方飞过，声音可以从后到前的喇叭顺序播放，感觉上就像是上空飞过一样。或者在恐怖游戏当中，可以判断出敌人偷偷地在左侧或者右侧出现。

虽然它可以很大程度地增加游戏体验，但是事实上大多数玩家都不会有 5.1 环绕配置。正因如此，游戏为 5.1 环绕开发功能就不是很划算。虽然它在 1000 美元配置上听起来当然会更好，但是游戏还是要能够在电视机内置的小喇叭中工作。

虽然你无法分辨出立体声中的前后，但不是说游戏就不需要设置了。监听者和发射者的位置，还有衰减等参数还是会很明显地影响左右侧喇叭的音量。

数字信号处理

广义上讲，**数字信号处理**（DSP）是计算机中表示的信号。在音频领域中，数字信号处理说的是加载音频文件然后在修改之后得到不同的效果。举个数字信号处理相对简单的例子，加载音频文件然后增加或者减小它的音高。

看起来没必要在运行时进行处理，先离线处理好这些效果，然后在游戏中播放也许会更好。但是运行时使用数字信号处理理由就是它能够节省大量内存。假设有一款剑击游戏，有 20 种不同的音效在武器相互碰撞的时候发出。这些音效听起来就像在野外开阔的场地发出。现在想象一下，如果游戏有多种场地会发出声音，除了野外之外，还有洞穴、大教堂，以及大量的其他地方。

问题在于，在洞穴中发出的声音和在野外发出的声音听起来完全不一样。特别是当刀剑在洞穴中发生碰撞时，会有很大的**回声**。不用数字信号处理效果，唯一的方法就是为二十多种刀剑声再针对不同的场地配置。如果有5种不同的场景，就意味着有5倍的增长，从而有一百多种的刀剑音效。现在如果需要所有战斗音效，而不仅仅是刀剑音效，游戏内存很快就被用尽。但是如果有了数字信号处理效果，同样的20个刀剑音效会用在不同的地方，它们只要根据场所调整成相应的音效即可。

实现数字信号处理需要线性系统和高级数学运算的知识，比如傅里叶变换。因此，实现这些效果超出了本书的范畴。但是如果你对于如何实现本章提到的音效感兴趣，可以查看本章最后的相关资料。不管怎样，虽然我们不去实现这些效果，但是讨论一下基本效果还是很值得的。

常见数字信号处理效果

游戏中常见的数字信号处理效果就是之前提到的回声。任何想在狭窄空间创建回声效果的游戏都会琢磨如何实现。一个非常流行的回声效果库叫作Freeverb3，*http://freeverb3.sourceforge.net*有提供。Freeverb3是一个**冲量驱动系统**，就是说为了对任何音效实现回声效果，需要一个音频文件表达在特殊场景播放的数据。

另一个大量使用的数字信号处理效果就是**音高偏移**，特别是多普勒偏移（在本章的后面会提到）。音高偏移会通过调整频率增加或者减小音效的音高。虽然多普勒偏移会很常用，但赛车游戏中引擎的音高会随着速度的变化而变化。

游戏中大多数的数字信号处理效果通常都会修改频率的范围或者输出分贝的级别。举个例子，一个**压缩机**缩小了音量的范围，导致很小的声音得到了增强，同时很大的声音得到了减小。通常用于统一多个音频文件，让它们保持相似的范围。

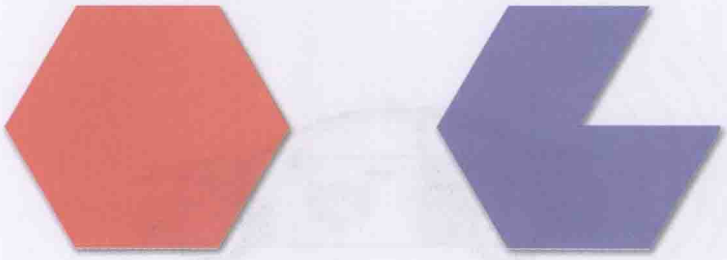
另一个例子是**低通滤波器**，通过删减频率的方式减小音量。在游戏中很常见的就是当玩家附近发生爆炸时的嗡鸣声。为了实现效果，时间会拉长，然后应用低通滤波器，接着播放嗡鸣声。

游戏中还有很多其他效果，但以上4种是游戏中最常见的。

区域标记

有一些效果很少会在整个关卡一直使用，比如说回响效果。通常只有关卡的一些区域才需要使用回响效果。比如说，如果一关里有野外区域和洞穴，回响效果可能只有在洞穴才会有。有很多种方法可以添加区域，最简单的方式就是在地面上标记凸多边形。

回忆一下**凸多边形**，就是所有定点都朝外的多边形。更具体地来讲，凸多边形的内夹角小于180°。图 6.5 展示了凸多边形和凹多边形。



(a) (b)

图 6.5 凸多边形 (a) 和凹多边形 (b)

使用凸多边形的原因就是，给定一个凸多边形我们更容易判断一个点在凸多边形内部还是外部。所以在我们的例子中，用了凸多边形来表示哪些区域应用回响效果。给定角色的位置，就很容易判断角色是在多边形内部还是外部。如果角色在区域内部，回响效果打开，在外部则关闭。判定一个定点是否在凸多边形内部的算法在第 7 章会讲更多，而这个算法超出了数字信号处理效果的范畴。

但是，我们不想玩家一进入区域就马上产生效果。否则效果会比较刺耳。这意味着，玩家进入标记区域之后，我们希望渐渐地开启回响效果，这样效果听上去更加平滑。

值得注意的是使用凸多边形标记区域会有一个问题。就是如果标记区域在关卡中上下重叠，而这些区域又有不同的数字信号处理效果，这个方法会失效。举例来讲，如果有一条通道要穿过草地下方，通道也同时标记了凸多边形区域。如果这个问题必须要解决，你可以使用绑定盒之类的方式来完成标记，在第 7 章会提到。

其他声音话题

虽然本章提及了很多声音相关的话题，但是还有一些不需要一整节来阐述的细节。

多普勒效应

如果你站在街上，同时一辆警车打开报警器向你靠近，音高会随着警车的靠近而提高。相对地，在警车远离之后，声音的音高也会降低。多普勒效应如下图 6.6 所示。

多普勒效应（或者称之为多普勒偏移）的出现是由于声波在空气中传播需要时间。在警车靠近的时候，意味着连续的声波都比前一个要早到。这就导致了频率的增加，就会有更高的

音高。警车就在你身边的时候，你可以听到声音的真实音高。最后，在警车远离你的时候，声波会需要越来越长的时间到达你的位置，导致了低音高的出现。

图 6.6 警报器上的多普勒效应

有趣的是多普勒效应不仅在音波中才会出现，而是所有与波相关的情况中都会出现。最有名的就数光波，如果物体靠近会出现红光，如果物体远离会出现蓝光。但是为了产生能够注意得到的偏移，对象要在非常大的空间里走得非常快。这在地球上的普通速度上不会出现，所以这效果在天文学上才能观测到。

在游戏中，动态多普勒效应只会在高速移动的对象身上应用，比如汽车。技术上来讲，也可以在子弹上应用，但是由于它们太快，我们通常只播子弹飞走的声音。由于多普勒效应会让音高增加或者降低，只有在支持数字信号处理效果处理音高的时候才能用。

声音遮挡

想象你在校园生活。勤奋学习的你，正在努力研读本书。突然间，大堂开了派对。尽管你的门已经关闭，但是声音还是很响亮，声音实在太大，导致你无法专心学习。你认得这首歌，但是听起来有点不同。最主要的差别就是低音为主，高音部分被挡住了。这个**声音遮挡**如图 6.7(a) 所示。

声音遮挡在声音不是直接由发射者传递到监听者的时候发生。声音遮挡主要就是低通滤波的结果，意味着高频率声音的音量被移除了。这是因为低频率的音波比起高频率更容易传播。但是，声音遮挡的另一个输出就是整体音量的降低。

相似但是不同的想法就是**声音衍射**。通过声音衍射，声音可能不再是直线传播的了，但是还是有可能穿透障碍物，如图 6.7(b) 所示。比如说，如果你朝柱子另一边的人大喊，声波会被柱子衍射。一个声音障碍物的有趣例子就是分割之后的音波有可能以不同的顺序到达。所以如果声波到达柱子然后发生分割，左边的音波可能比右边的音波早到达。就是说，柱子另一边的人可能会由于到达的音波时间不同而听到两次。

检测遮挡和衍射的方法就是为发射者构造一系列指向监听者附近的弧形。如果没有一个向量能直接到达，那就是遮挡。如果有一些向量能够达到，那么是衍射。如果全部向量都能达到，那么两者都不是。这个方法称之为 **Fresnel 声学衍射**，如图 6.7(c) 所示。

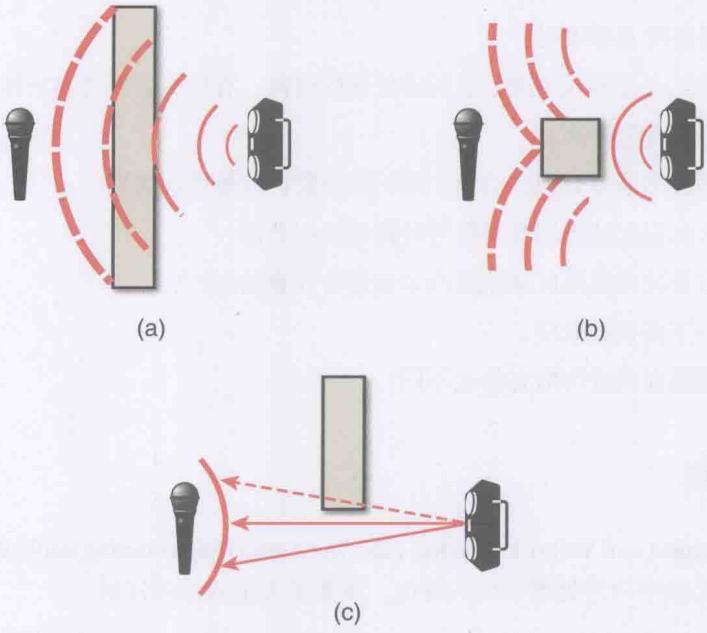


图 6.7　声音遮挡 (a)、声音衍射 (b)、Fresnel 声学衍射 (c)

实现这个方法需要判定向量是否与对象相交，这是我们还没讨论到的。但是，多种类型的相交我们会在第 7 章讨论。值得一提的是，我们可以使用射线判断从发射者到监听者哪个路径是被遮挡的。

总结

声音对任何游戏来说都是重要的组成部分。组织上，我们希望某种方式将声音以元数据的方式组织起来控制在何时如何播放。数字信号处理效果比如回声可以在播放的时候添加到音频文件中。对于 3D 音效，我们同时关心监听者和发射者在世界中的位置。最后，一些游戏可能会实现针对快速移动对象的多普勒效应、声音遮挡、声音衍射等，这取决于环境。

习题

1. 描述音频文件和相关的元数据的不同。
2. 可切换的声音事件与普通的声音事件相比有什么优点？
3. 什么是监听者和发射者？
4. 当在第三人称游戏中决定监听者位置的时候，有什么需要考虑的地方？
5. 分贝刻度是怎样的？
6. 什么是数字信号处理？给出3种不同的数字信号处理效果。
7. 为什么标记区域控制数字信号处理效果很有用？
8. 数字信号处理效果区域使用凸多边形有什么缺点？
9. 描述一下多普勒效应。
10. 声音遮挡和声音衍射有什么不同？

参考资料

Boulanger, Richard and Victor Lazzarini, Eds. ***The Audio Programming Book.*** **Boston: MIT Press, 2010.** 这本书是数字信号处理领域的著作，涵盖了大量的示例代码。

Lane, John. ***DSP Filter Cookbook.*** **Stamford: Cengage Learning, 2000.** 这本书比上一本要高级，它实现了很多特殊效果，包括压缩机和低通道滤波。

第7章 物理

物理都会实现碰撞和运动，但不是所有游戏都需要物理。碰撞检测用于检测两个游戏对象是否相互交错在一起。有大量的算法可以检测碰撞，本章的第一部分就首先讨论这些基础算法。

物理学中的运动部分会考虑作用力、加速度、质量等，以及其他经典力学中用到的属性，用于判断物体每帧的去向。为了做到这样，必须用到微积分，主要是数值积分法，这是本章的第二部分。

平面、射线和线段

在我们讨论如何在游戏世界中使用碰撞检测之前，需要先讨论一些物理学中的数学。也许你已经在高中代数课程中学过，但是在已经熟悉了线性代数之后，我们可以从另一个角度谈谈它们。

平面

平面是平的，在二维上无限延伸，就如同线可以在一维空间无限延伸一样。在游戏中，我们通常用平面作为地面和墙体的抽象，虽然可能还有其他用法。一个平面可以有多种表示方法，但是通常游戏程序员会倾向于用以下表示：

$$P \cdot \hat{n} + d = 0$$

P是平面上任意一点，$\hat{n}$ 是平面法线，d 是平面到原点的最小距离。

回忆一下，三角形是可以确保在一个平面的。上面的平面表达式会采用的一个原因就是，给定一个三角形，很容易根据三角形构造出该平面。在我们计算出 $\hat{n}$ 和 d 之后，可以测试如果任意点 P，也同样满足等式，那么 P也在平面上。

假设我们有 $\triangle ABC$ 以顺时针顶点序排列（如图 7.1 所示）。为了得到三角形所在平面的表达式，我们需要先计算三角形的法线。希望你还记得，对三角形的两条边进行叉乘可以得到三角形的法线——如果你不记得，在往下阅读之前，请回头看看第 3 章。

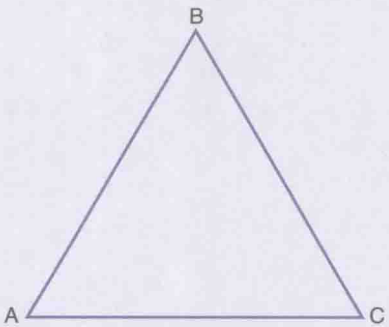


图 7.1 顺时针排列的三角形 ABC

不要忘记了，叉乘的顺序是很关键的，根据顶点序进行叉乘很重要。由于 $\triangle ABC$ 顶点序以顺时针排列，如图 7.1 所示，我们希望构造向量从 A 到 B 和从 B 到 C。在我们有了两个向量之后，我们对两个向量进行叉乘，然后对结果进行正规化得到 $\hat{n}$。

在得到 $\hat{n}$ 之后，我们需要使用平面上的顶点解出 d，由于

$$d = -P \cdot \hat{n}$$

幸运的是，我们已经知道三角形上的 3 个点都在平面上，为 A、B 和 C。我们可以将这些点与 $\hat{n}$ 点乘，就会得到 d 的值。

平面上的所有顶点都会得到相同的 d 值，那是因为这是一个投影：$\hat{n}$ 是正规化过的，P是未正规化过的，所以我们会得到平面到原点在 $\hat{n}$ 方向上的最小距离。这个最小距离不管你采用哪个顶点都一样，因为它们都在一个平面上。

在得到 $\hat{n}$ 和 d 之后，我们可以将这些值存储在我们的 Plane 数据结构体内：

```
struct Plane
   Vector3 normal
   float d
end
```

射线和线段

射线就是从某个点开始出发，朝某个方向无限延伸。在游戏中，通常用参数方程表示射线。回忆一下，参数方程是借助其他参数来表达的，一般称之为 t。对于射线来说，参数方程如下：

$$R(t) = R_0 + \vec{v}t$$

R_0 就是起点，而 $\vec{v}$ 就是射线穿越的方向。由于射线是从某点开始，然后朝着某个方向无限延伸，为了让射线表达式顺利工作，t 必须大于等于 0。就是说当 t 为 0 时，这个参数方程在起点 R_0 就停了。

线段与射线类似，除了既有起点又有终点之外。我们可以使用完全同样的参数方程来表示线段。唯一不同的地方就是现在 t 有了上限，因为线段必须有一个终点。

技术上来讲，**光线投射**就是射出一条射线，然后检查是否打到某个对象。但是，大多数物理引擎都会由于实际上使用的是线段做检测的方法让人迷惑，包括 Havok 和 Box2D。这么做的原因是游戏世界通常都会有一定的约束，所以使用线段更加合理。

我承认这个术语会让人困惑。但是，这是游戏产业的惯用术语，我想保持这个术语的一致性。所以请记得本书中的光线投射实际上使用的都是线段。

那么光线投射有什么用途？它在 3D 游戏中几乎无处不在。举个常见的例子就是发射一颗子弹穿过一条直线。虽然一些游戏采用弹道模拟的方式计算子弹弹道，但是采用光线投射也

差不多，因为子弹运行得很快。还有其他的光线投射应用，包括根据敌友变色、AI 判断敌人是否可见、Fresnel 声学衍射、鼠标选取物品。所有这些场景都可以使用光线投射。

由于光线投射要求使用线段，我们至少需要两个参数——线段的起点和终点：

```
struct RayCast
   Vector3 startPoint
   Vector3 endPoint
end
```

将 startPoint 和 endPoint 转换成线段参数方程的形式相当简单。R_0 就是 startPoint，$\vec{v}$ 就是 endPoint-startPoint。当以这种方式转换之后，t 的值从 0 到 1 会对应到光线投射。也就是说，t 为 0 就在 startPoint，而 t 为 1 则在 endPoint。

碰撞几何体

在现代 3D 游戏中，人形角色拥有 15000 以上条边的多边形很常见。当游戏需要判断两个角色是否碰撞时，检查所有三角片的碰撞效率不会很高。正因如此，大多数游戏都使用简化的几何体做碰撞检测，比如球体、盒子。这些碰撞几何体不会绘制到屏幕上，只是用于提高碰撞检测的速度。在这一节，我们会讲游戏中大部分常见的碰撞几何体。

值得一提的是，游戏对象拥有多个不同级别的碰撞几何体也是很常见的。这样，简单的碰撞体可以先进行第一轮碰撞检测。在简单的碰撞体发生了碰撞之后，再选择更精细的碰撞体进一步检测碰撞。

包围球

最简单的碰撞体就是包围球（在 2D 游戏中则是包围圈）。一个球体可以通过两个变量定义——向量表示球体的中心点，标量表示球体的半径：

```
class BoundingSphere
   Vector3 center
   float radius
end
```

如图 7.2 所示，多个物体，包括小行星都在球体内包围，适配得很好。但是其他类型的物体，包括人形角色，留出了很多空间。这意味着有很多类型的物体使用球形包围体会有很多**漏报** (false positive)。就是说两个游戏对象的包围体发生碰撞，但是两个物体自身还没有碰撞。漏报会让玩家很困惑。

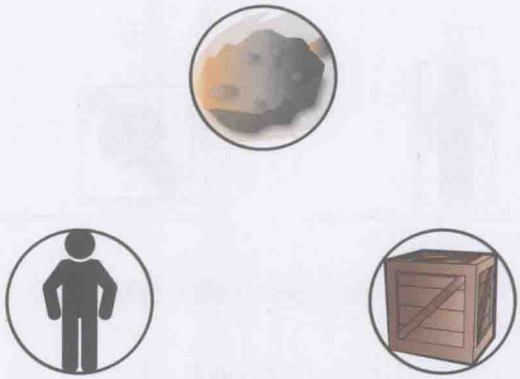


图 7.2　不同物体上的包围球

由于包围球对大多数游戏对象都不够精准，使用包围球作为唯一碰撞体是不合理的。但是包围球的优势就是进行碰撞检测非常简单，所以这种碰撞体是个不错的候补。

轴对齐包围盒

对于 2D 游戏来说，一个**轴对齐包围盒**（缩写 AABB）就是一个每条边都平行于 x 轴或者 y 轴的矩形。类似地，在 3D 游戏中，AABB 就是长方体，而且每条棱都与对应的轴平行。不管 2D 还是 3D，AABB 都可以用两个点表示：最大点和最小点。在 2D 中，最小点就是左下角的点，而最大点则是右上角的点。

```
class AABB2D
   Vector2 min
   Vector2 max
end
```

由于 AABB 必须与对应的轴平行，如果一个对象旋转，那么 AABB 就需要拉长，如图 7.3 所示。但是对于 3D 游戏来说，人形角色通常只绕向上的轴旋转，这种旋转并不会让 AABB 有太多的变化。因此，使用 AABB 作为人形角色的包围体是很常见的，特别是 AABB 和球体之间的碰撞计算量很小。

朝向包围盒

一个**朝向包围盒**（或者 OBB）类似于轴对齐包围盒，只是不再要求与轴平行。就是说，这是一个长方形（2D）或者长方体（3D），而且每条轴不再需要与对应的坐标轴平行。OBB 的优点就是可以随着游戏对象旋转，因此不管游戏对象的朝向如何，OBB 的精准度都很高。但是这个精准度的提升是有代价的，与 AABB 相比，OBB 的计算花费高多了。

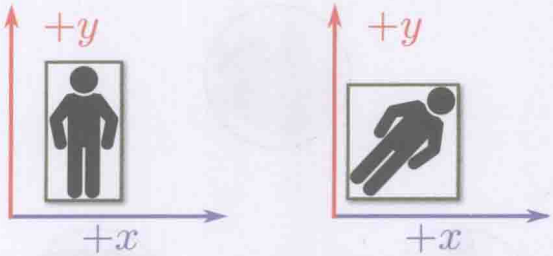


图 7.3 不同朝向下的轴对齐包围盒

OBB 在游戏中有多种表示方式，包括用 8 个顶点或者 6 个平面。但是 OBB 计算比较复杂，本书不打算讲具体如何计算。但是，如果你想在游戏中使用它，可以看一下本章参考资料中的 *Real-time Collision Detection* 一书。

胶囊体

在 2D 游戏中，胶囊体可以看作是一个 AABB 加上两端各一个半圆。之所以叫胶囊体是因为看上去就跟药物胶囊一样。如果我们把胶囊体扩展到 3D，就会变成一个圆柱加上两端各一个半球。胶囊体在人形角色的碰撞体表示中是很流行的，因为他们比 AABB 精准一些，如图 7.4(a) 所示。

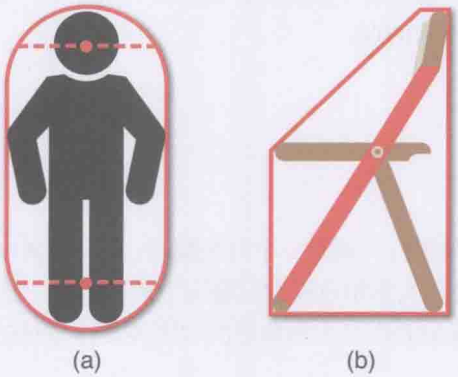


图 7.4 被胶囊体包围的人形 (a) 和被凸多边形包围的椅子 (b)

胶囊体还可以看作带半径的线段，在游戏引擎中就是这么表示的：

```
struct Capsule2D
   Vector2 startPoint
   Vector2 endPoint
   float radius
end
```

凸多边形

另一个碰撞几何体表示的选择就是使用**凸多边形**（在 3D 领域称之为**凸包**）。与你所想的差不多，凸多边形比其他方式效率都要低，但是比它们都精准。图 7.4(b) 演示一张用了凸多边形的椅子。虽然还是有很多漏报，但是漏报的情况比其他方式都要好。

组合碰撞几何体

最后一个增加精准度的选择就是使用**组合碰撞几何体**进行碰撞检测。在人形的例子中，我们可以在头部使用球形，身干用 AABB，凸多边形用于手脚等。通过不同的碰撞几何体组合，我们几乎可以消灭漏报。

虽然检测碰撞几何体组合比检测模型的三角片要快，但还是慢得让你不想用。在人形的例子中，应该先用 AABB 或者胶囊体进行第一轮碰撞检测。然后通过之后再进行更精确的测试，比如组合碰撞几何体。这种方法取决于你是否需要将精准度分级别。在检测子弹是否打中角色的时候会用到，但是阻挡玩家走进墙里就没必要了。

碰撞检测

现在我们看过了游戏中用到的主要碰撞体，我们可以看一下这些碰撞体之间的碰撞检测。任何游戏要响应相互之间的碰撞都要使用这些类型的碰撞检测。可能有的人看到本节的数学会感到难受，但是这些都是游戏中很常见的数学。因此，理解这些数学会非常重要——这些数学不是为了提及而提及的。本节不会讲所有几何体与其他几何体的碰撞，但是会讲一下你可能会用到的最常见的部分。

球与球的交叉

如果两个球的半径之和小于两个球之间的距离，那么就发生了交叉，如图 7.5 所示。但是，计算距离会用到平方根，为了避免平方根的比较，通常都会使用距离的平方与半径之和的平方进行比较。

这个算法只有几行代码，如清单 7.1 所示。它是效率非常高的，用它作为基本的碰撞体是很常见的。

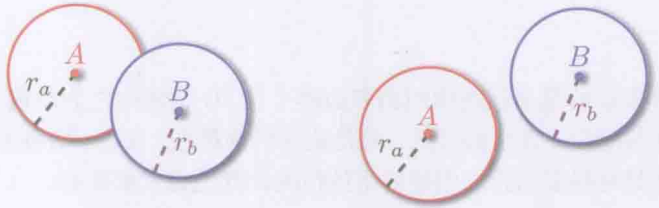


图 7.5 两个球交叉 (a) 和没有交叉 (b)

清单 7.1 球与球的交叉

```
function SphereIntersection(BoundingSphere a, BoundingSphere b)
   // 构造两个中心点的向量，然后求长度的平方
   Vector3 centerVector = b.center - a.center
   // 回忆一下，v的长度平方等于v点乘v
   float distSquared = DotProduct(centerVector, centerVector)

   // distSquared是否小于半径和的平方？
   if distSquared < ((a.radius + b.radius) * (a.radius + b.radius))
      return true
   else
      return false
   end
end
```

AABB 与 AABB 交叉

如同球体交叉一样，AABB 的交叉计算即使在 3D 游戏中也是很廉价的。2D 中的 AABB 看起来会容易理解一些，所以这里谈一下 2D 的 AABB 交叉。

当检测两个 2D 的 AABB 交叉的时候，检测没交叉比检测有交叉要容易些。这 4 种情况如图 7.6 所示。

如果 4 种情况任意一种为真，意味着 AABB 之间没有交叉，所以交叉函数返回假。这意味着返回表达式中的 4 个条件逻辑应该反过来写，如清单 7.2 所示。

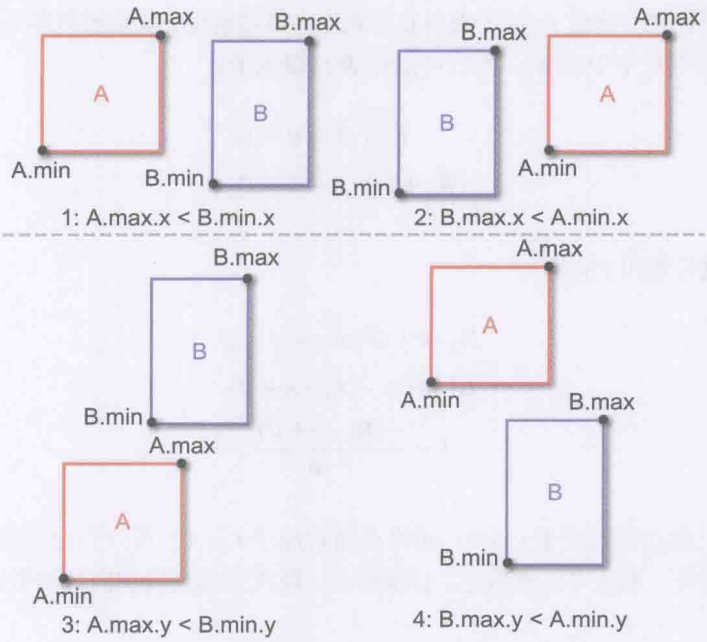


图 7.6 两个 AABB 完全没有交叉的 4 种情形

清单 7.2 AABB 与 AABB 的交叉

```
function AABBIntersection(AABB2D a, AABB2D b)
   bool test = (a.max.x < b.min.x) || (b.max.x < a.min.x) ||
               (a.max.y < b.min.y) || (b.max.y < a.min.y)

   return !test
end
```

值得一提的是，AABB 之间的交叉判定是一种成为轴分离算法的特殊应用。在通用的形式中，这种算法可以应用在任意类型的凸多边形上，虽然它的细节超出了本书的范畴。

线段与平面交叉

检测线段是否与平面碰撞在游戏中是很常见的。为了理解算法是如何运行的，我们先来理解一下背后的线性代数。首先，我们有线段和平面的两个等式：

$$R(t) = R_0 + \vec{v}t$$

$$P \cdot \hat{n} + d = 0$$

我们想判断是否存在一个值 t，使得点落在平面上。换句话说，我们想判断是否存在 t 值，使得 $R(t)$ 满足平面等式中 P 的值。所以可以将 $R(t)$ 带入 P：

$$R(t) \cdot \hat{n} + d = 0$$
$$(R_0 + \vec{v}t) \cdot \hat{n} + d = 0$$

接下来的问题就是解出 t 的值：

$$R_0 \cdot \hat{n} + (\vec{v} \cdot \hat{n})t + d = 0$$
$$(\vec{v} \cdot \hat{n}) = -(R_0 \cdot \hat{n} + d)$$
$$t = \frac{-(R_0 \cdot \hat{n} + d)}{\vec{v} \cdot \hat{n}}$$

回忆一下线段，起点则对应于 $t = 0$，而终点则对应于 $t = 1$。所以当我们解出 t 时，如果 t 的值在这个范围外，那么可以忽略它。特别的是，负值表示线段朝向远离平面的方向，如图 7.7(a) 所示。

同样值得注意的是，如果 $\vec{v}$ 与 $\hat{n}$ 点乘结果为 0，会产生除 0 异常。回想一下，如果两个向量点乘结果为 0，意味着两个向量垂直。在这种情况下，表示 $\vec{v}$ 与平面平行，因此不会有交叉，如图 7.7(b) 所示。唯一交叉的情况就是线段就在平面上。在任何情况下，除 0 异常是必须要考虑的。

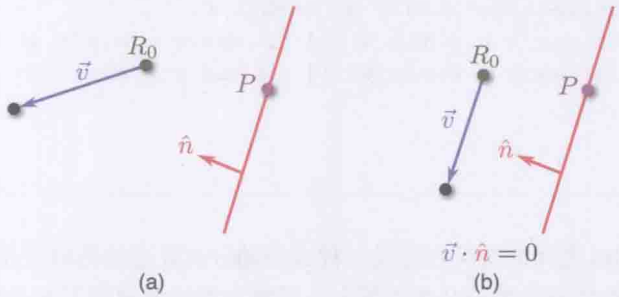


图 7.7 线段远离平面 (a) 和线段与平面平行 (b)

还有一点就是，如果线段与平面真的相交，我们可以将 t 替换为交点的值，如清单 7.3 所示。

清单 7.3 线段与平面交叉

```
// 返回值就是这个结构体
struct LSPlaneReturn
   bool intersects
   Vector3 point
```

```
end

// 记住光线投射实际上就是线段
function LSPlaneIntersection(RayCast r, Plane p)
   LSPlaneReturn retVal
   retVal.intersects = false

   // 计算线段方程的v
   Vector3 v = r.endPoint - r.startPoint

   // 检查线段是否与平面平行
   float vDotn = DotProduct(v, p.normal)
   if vDotn is not approximately 0
      t = -1 * (DotProduct(r.startPoint, p.normal) + p.d)
      t /= vDotn

      // t应该介于起点与终点（0到1）之间
      if t >= 0 && t <= 1
         retVal.intersects = true

         // 结算交点
         retVal.point = r.startPoint + v * t
      end
   else
      // 测试起点是否在平面上
      ...
   end

   return retVal
end
```

线段与三角片交叉

假设你需要算出用线段表示的子弹与某个三角片之间是否发生碰撞。第一步就是算出三角片所在的平面。在有了这个平面之后，你可以看看这个平面是否与线段相交。如果他们相交，你就会得到与三角形所在平面相交的交点。最后，由于平面是无限大的，我们要检测该点是否在三角片之内。

如果三角形 $\triangle ABC$ 以顺时针顶点序表达，第一步就是构造一个从 A 到 B 的向量。然后构造一个向量从 A 到 P，P 就是交点的位置。如果旋转向量 $\vec{AB}$ 到 $\vec{AP}$ 是顺时针，P 就在三角形一侧。这个检测对每一条边（$\vec{BC}$ 和 $\vec{CA}$）进行计算，如果每个都是顺时针，也就是每条边算出 P 都在三角形一侧，就可以得出结论说 P 就在三角形内部，如图 7.8 所示。

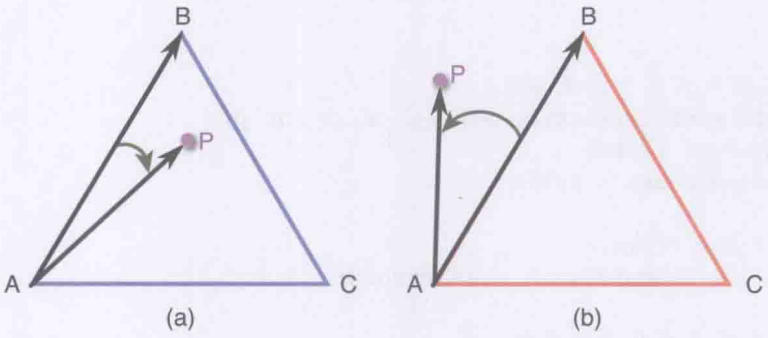


图 7.8 点在三角形内部 (a) 和点在三角形外部 (b)

但是我们怎么判断是顺时针还是逆时针？在图 7.8(a) 中，如果我们在右手坐标系中计算 $\vec{AB} \times \vec{AP}$，得到的向量会朝向书页内。回想一下右手坐标系中顺时针顶点序三角形的法线也是朝向书页内部的。这种情况下，叉乘向量的方向就与三角形法线方向一致。如果两个正规化过的向量点乘值为正值，它们朝向大致一致。所以如果你将 $\vec{AB} \times \vec{AP}$ 的结果正规化后再与法线点乘结果为正数，那么 P 就在 $\vec{AB}$ 所在的三角形一侧。

这个算法可以用于三角形的其他边。它看起来不仅能够作用于三角形，对于任意在同一平面上的多边形也同样适合。伪代码如清单 7.4 所示。

清单 7.4 判断点是否在多边形内部

```
// 这个函数只能在顶点为顺时针顶点序及共面下正常工作
function PointInPolygon(Vector3[] verts, int numSides,
                        Vector3 point)
   // 计算多边形的法线
   Vector3 normal = CrossProduct(Vector3(verts[1] - verts[0]),
                                 Vector3(verts[2] - verts[1])
   normal.Normalize()

   // 临时变量
   Vector3 side, to, cross

   for int i = 1, i < numSides, i++
      // 从上一个顶点到当前顶点
      side = verts[i] - verts[i - 1]
      // 从上一个顶点到point
      to = point - verts[i - 1]

      cross = CrossProduct(side, to)
      cross.Normalize()
```

```
        // 表示在多边形外部
        if DotProduct(cross, normal) < 0
            return false
        end
    loop

    // 必须检测最后一条边，就是最后一个顶点到第一个顶点
    side = verts[0] - verts[numSides - 1]
    to = point - verts[numSides - 1]
    cross = CrossProduct(side, to)
    cross.Normalize()

    if DotProduct(cross, normal) < 0
        return false
    end

    // 在所有边内部
    return true
end
```

球与平面交叉

在游戏中球可以与墙发生碰撞，为了对这个碰撞准确建模，可以使用球与平面的交叉。给定平面的 $\hat{n}$ 和 d，碰撞检测最简单的方法就是建立一个新的平面，对齐球心并且与原有平面平行。如果两个平面距离比球的半径要小，那么就发生了交叉。这个过程如图 7.9 所示。

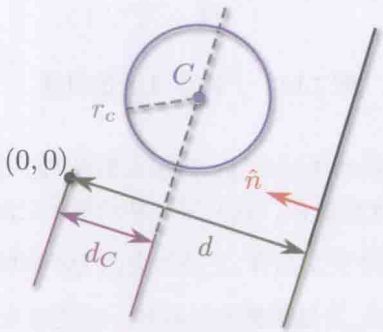


图 7.9 球与平面不交叉的情况

就像球与球的交叉一样，球与平面的交叉也不复杂，如清单 7.5 所示。

清单7.5 球与平面交叉

```
function SpherePlaneIntersection(BoundingSphere s, Plane p)
   // 通过平面的法线p.normal及平面的点s.center计算平面的d
   float dSphere = -DotProduct(p.normal, s.center)

   // 检查是否在范围之内
   return (abs(d - dSphere) < s.radius)
end
```

球形扫掠体检测

到目前为止，我们讲了**即时碰撞检测**算法。就是说那些算法只能检查当前帧中发生的碰撞。虽然很多情况下都有效，但是也有很多不适用的时候。

如果子弹朝着纸张发射，不存在子弹与纸张交错在一起的准确的一帧。这是因为子弹速度很快，而纸张很薄。这个问题通常被称为**子弹穿过纸张问题**，如图7.10所示。为了解决这个问题，能够进行**连续碰撞检查**（CCD）的能力是必要的。通用的CCD超出了本书的话题，但是我们可以只讨论其中一种。

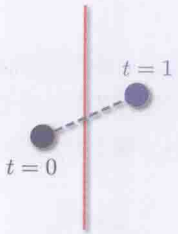


图7.10 子弹穿过纸张问题

在**球形扫掠体检测**中，有两个移动中的球体。而输入则是两个球在上一帧的位置（$t=0$）和这一帧的位置（$t=1$）。给定这些数据，我们可以判断两帧之间两个球是否发生了碰撞。

所以不像即时碰撞检测的球与球交叉那样，它是不会因为不同帧而丢失交叉，如图7.11所示。

你可能注意到，球形扫掠体看上去和胶囊体差不多。那是因为球形扫掠体确实就是胶囊体。球形扫掠体有起点、终点及半径，完全就是一个胶囊体。所以胶囊体与胶囊体的碰撞完全可以在这里使用。

如同线段与平面交叉问题一样，先解等式会对我们有帮助。而且解决胶囊体的碰撞本身也是一个非常流行的问题，所以经常会在面试的时候被问到。由于它涉及很多游戏程序员需要掌握的线性代数概念。

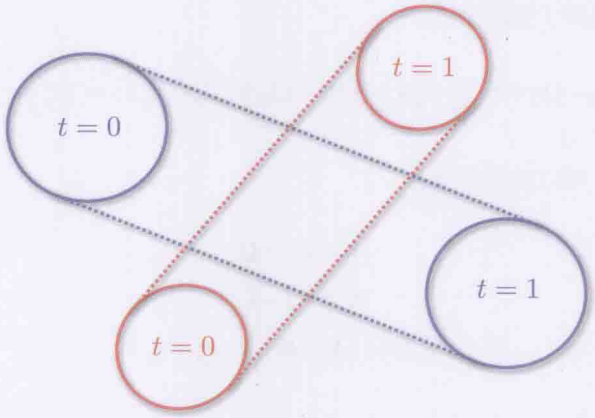


图 7.11　球形扫掠体检测

给定球体的上一帧和这一帧的位置，就可以将球的位置转换为参数方程。这个转换得到的函数可以用于光线投射。所以给定球 P 和球 Q，我们可以用两个参数方程表示：

$$P(t) = P_0 + \vec{v}_p t$$
$$Q(t) = Q_0 + \vec{v}_q t$$

我们想要求的是 t，t 就是两个球距离等于半径之和的时候，因为那就是发生交叉的时候。数学表示如下：

$$\|P(t) - Q(t)\| = r_p + r_q$$

这个等式的问题就是我们需要一些方法来避开长度运算。诀窍就在于向量 $\vec{v}$ 的长度平方就等于对自身进行点乘：

$$\|\vec{v}\|^2 = \vec{v} \cdot \vec{v}$$

因此，如果对公式两边取平方，我们可以得到如下等式：

$$\|P(t) - Q(t)\|^2 = (r_p + r_q)^2$$
$$(P(t) - Q(t)) \cdot (P(t) - Q(t)) = (r_p + r_q)^2$$

现在我们有了这个等式，就可以对 t 求解。这个过程有点复杂。首先，将 $P(t)$ 和 $Q(t)$ 替换进来：

$$(P_0 + \vec{v}_p t - Q_0 - \vec{v}_q t) \cdot (P_0 + \vec{v}_p t - Q_0 - \vec{v}_q t) = (r_p + r_q)^2$$

然后，我们可以提取因子整理公式：

$$\left(P_0 - Q_0 + (\vec{v}_p - \vec{v}_q)t\right) \cdot \left(P_0 - Q_0 + (\vec{v}_p - \vec{v}_q)t\right) = \left(r_p + r_q\right)^2$$

为了更进一步简化，可以继续替换：

$$A = P_0 - Q_0$$
$$B = \vec{v}_p - \vec{v}_q$$
$$(A + Bt) \cdot (A + Bt) = \left(r_p + r_q\right)^2$$

由于点乘被加法隔开了，我们可以对 $(A + Bt)$ 项使用 FOIL（first,outside,inside,last）法则：

$$A \cdot A + 2(A \cdot B)t + (B \cdot B)t^2 = \left(r_p + r_q\right)^2$$

如果我们将 $\left(r_p + r_q\right)^2$ 移到等式左边，然后再做一下替换，会得到更加熟悉的等式：

$$A \cdot A + 2(A \cdot B)t + (B \cdot B)t^2 - \left(r_p + r_q\right)^2 = 0$$
$$a = (B \cdot B)$$
$$b = 2(A \cdot B)$$
$$c = A \cdot A - \left(r_p + r_q\right)^2$$
$$at^2 + bt + c = 0$$

你可能会想起这种二次方程可以用很久没用过的方法来解：

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

平方根下的值，我们称之为判别式，是非常重要的。有 3 种情况：小于 0、等于 0，以及大于 0。

如果判别式小于 0，t 就没有实根，就是说没有交叉发生。如果判别式等于 0，意味着两个球相切。如果判别式大于 0，意味着两个球完全交叉在一起，两个根中比较小的就是最早发生交叉的时候。所有 3 种情况如图 7.12 所示。

在我们解出 t 的值以后，我们可以看一下如果值介于 0 和 1 之间，编码的时候就要注意了。记住 t 值如果大于 1 则是这一帧之后，如果小于 0 则是这一帧之前。因此，t 值超出范围的情况不是这个函数接受的范围。代码实现如清单 7.6 所示。

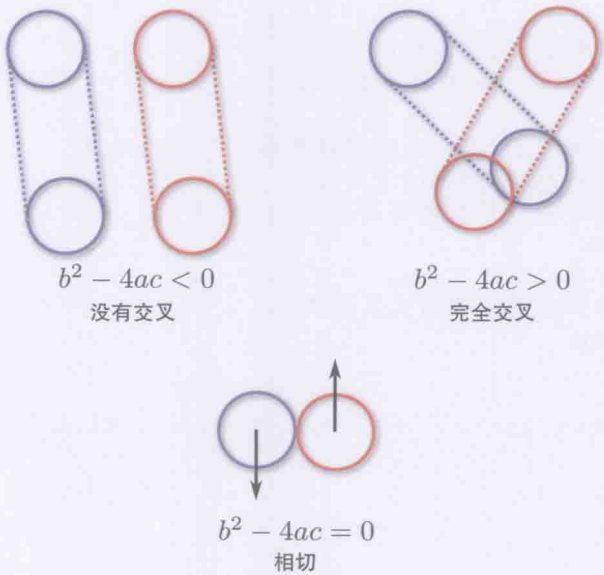


图 7.12　球形扫掠体检测中的所有情形

清单 7.6　球形扫掠体检测

```
// p0/q0是上一帧的球体
// p1/q1是这一帧的球体
function SweptSphere(BoundingSphere p0, BoundingSphere q0,
                     BoundingSphere p1, BoundingSphere q1)
   // 首先计算v用于参数方程
   Vector3 vp = p1.center - p0.center
   Vector3 vq = q1.center - q0.center

   // 计算A和B
   // A = P0 - Q0
   Vector3 A = p0.center - q0.center
   // B = vp - vq
   Vector3 B = vp - vq

   // 计算a、b和c
   // a = B dot B（译注：其中dot表示点乘运算）
   float a = DotProduct(B, B)
   // b = 2(A dot B)
   float b = 2 * (DotProduct(A, B)
   // c = (A dot A) - (rp + rq) * (rp + rq)
   float c = DotProduct(A, A) - ((q0.radius + p0.radius) *
                                 (q0.radius + p0.radius))
```

```
    // 现在计算判别式(b^2 - 4ac)
    float disc = b * b - 4 * a * c
    if disc >= 0
        // 如果我们需要t的值，我们可以用以下的方式解出：
        // t = (-b - sqrt(disc)) / (2a)
        // 但是，这个函数只要回答真就可以了
        return true
    else
        // 没有实数解，所以没有交叉发生
        return false
    end
end
```

响应碰撞

我们可以使用前面提到的各种算法来检测碰撞。但是在检测结果出来之后，游戏应该如何处理呢？这就是**响应**碰撞的问题。一些情况下，响应会很简单：一个或多个对象可能会死亡然后从游戏世界中移除。稍微复杂一点的响应就是一些比如火箭这样的物体会减少敌人的生命值。

但是如果两个对象需要相互弹开呢？比如两个小行星碰撞。一个简单的解决方法就是根据碰撞的方向让速度反向。但这么做会有很多问题。一个问题就是行星会被卡住。假设两个行星在某一帧发生碰撞，那么就会引起速度取反。但是如果它们速度很慢，导致下一帧还继续碰撞呢？额，那么速度就会无限地循环变化下去，然后就会卡住。如图 7.13(a) 所示。

为了解决第一个问题，我们需要找到两个行星发生碰撞的准确位置，尽管它们每帧都会发生。由于行星使用包围球，我们可以使用球与球之间的交叉来找出碰撞发生的时间。在我们算出时间之后，我们要将球回滚到那个时间点的位置。然后我们就可以根据情况添加速度了（我们初步方案是取反）。然后用新的速度在剩余的时间里更新对象的行为。

但是，这里的碰撞响应还是有一个大问题：将速度取反不是一个正确的行为。我们可以看一下为什么，假设你有两个行星朝着同一个方向运动，一个在另一个前面。前面的行星比后面的行星速度要慢一些，所以最终后面的行星会跟上前面的行星。就是说当两个行星碰撞的时候，它们会突然朝另一个方向运动，这肯定是不对的。

所以比起将速度取反，我们实际上是想根据发生碰撞的平面的法线将速度进行反射。向量反射的概念在第 3 章我们就已经讨论过了。如果行星与墙面碰撞，计算碰撞的墙面是很简单的，墙面就是我们要的平面。但是两个球在某个点碰撞的例子中，我们所要的平面就是碰撞点的切线平面，如图 7.13(b) 所示。

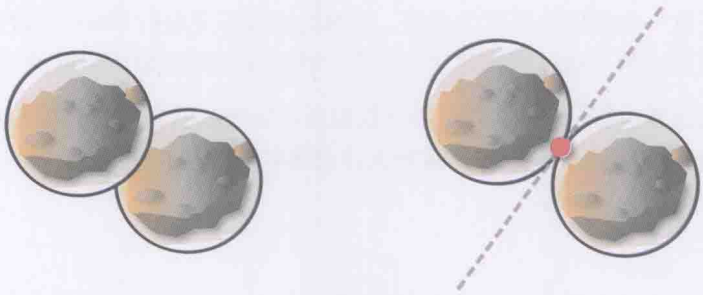


图 7.13　两个行星卡在一起 (a) 和两个行星在切线处发生碰撞 (b)

为了构造切线平面，我们首先要得到发生碰撞的点。这个可以用线性插值算出。如果有两个球体在某个点发生碰撞，这个点肯定就在两个球心所连成的线段上。它所在的位置就取决于两个球的半径。如果我们有两个 BoundingSphere 的实例 A 和 B 在某个点发生碰撞，这个点可以通过线性插值计算出来。

```
Vector3 pointOfIntersection = Lerp(A.position, B.position,
                                   A.radius / (A.radius + B.radius))
```

而找出切线平面也很简单，就是一个球心指向另一个球心的向量，然后正规化。有了平面上的点和平面的法线，我们就可以创建在这个碰撞点上的切线平面了。虽然碰撞响应需要对速度进行反射，但是我们只要有平面的法线就可以了。

有了这个反射之后的速度，行星碰撞看上去好多了，虽然看上去还是很奇怪，因为行星的反射前后都会保持恒定速度。在现实中，两个对象碰撞的时候，有一个**恢复系数**，衡量两个物体在碰撞后的反弹程度：

$$C_R = \frac{\text{碰撞后的相对速度}}{\text{碰撞前的相对速度}}$$

在**弹性**碰撞（$C_R > 1$）的情况下，碰撞后的相对速度大于碰撞前的相对速度。在另一方面，在**无弹性**碰撞 ($C_R < 1$) 就会导致碰撞后相对速度更低。在行星的例子中，我们更倾向于无弹性碰撞，除非它们是魔法行星。

还有就是角度的变化，这个会在本章的后面进行讨论。本节只是希望能够给你一些关于如何实现更加可信的碰撞响应方面的灵感。

优化碰撞

我们讨论的所有碰撞检测算法都只能检测一对物体间的碰撞。一个可能会遇到的问题是，如果有大量的物体需要进行碰撞检测呢？假设我们有 10000 个对象在游戏世界中，然后想检测

我们的角色与任意一个物体是否发生碰撞。原始的方法需要进行 10000 次碰撞检测：将角色和每一个物体进行检测。这样就非常没有效率，特别是在检测距离很远的对象的时候。

所以必须对游戏世界进行分区，这样主角只要跟所在区域的对象进行碰撞检测就可以了。2D 游戏中的一种分区方法就是**四叉树**，游戏世界会递归切割成矩形，直到每一个叶子节点只引用一个对象，如图 7.14 所示。

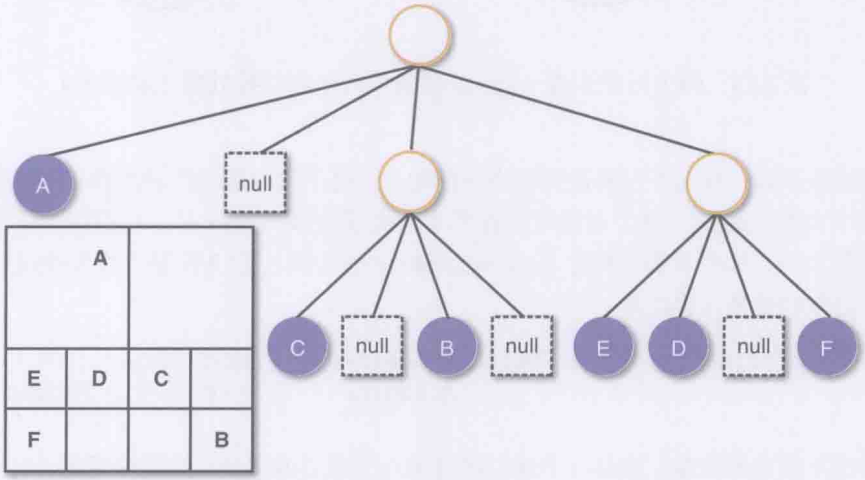


图 7.14 四叉树，字母表示游戏世界中的对象。

在进行碰撞检测的时候，程序会先检测最外层的四叉树矩形中的玩家所在象限的对象是否与玩家发生了碰撞，这样就立刻剔除了 3/4 的对象。然后这个递归算法会不断进行下去，直到找到所有潜在与玩家发生碰撞的对象。在只剩下少数的对象之后，就可以对每个对象进行碰撞体检测了。

四叉树不是唯一的分区方法。还有很多方法，比如二进制空间分割（BSP）及八叉树（3D 版的四叉树）。大多数算法都是基于空间的，还有一些是启发式分组的。分区算法能够轻松地写满一本书，之前提到的 *Real-time Collision Detection* 会谈到这个话题。

基于物理的移动

如果一个游戏对象在游戏世界中移动，就有一些物理会用于模拟这种运动。牛顿物理（也叫作经典物理）在 17 世纪被牛顿用公式表示出来。游戏中会大量使用牛顿物理，这是一个很好的模型，因为游戏对象不会以光速运动。牛顿物理由多个不同部分组成，但是本节聚焦最基础的部分：**线性力学**。就是没有旋转的运动。

在你了解经典力学之前，我想提前说一下，这个话题是很不容易理解的，这就是为什么有整个大学课程围绕它来研究。当然，我不能在本书中面面俱到（老实讲，市面上有大量的物理学书籍）。所以我要很小心地选择要讨论的话题——我想谈一下那些本书目标读者即开发游戏的人关心的话题。

线性力学概览

线性力学的两个基石是力与质量。**力**是一种相互作用，可以导致物体运动。力有着方向和大小，因此可以用向量表示。**质量**表示物体所含物质的量。对于力学来说，主要的关系是质量越大，物体就越难运动。

如果一个足够大的力作用到物体身上，理论上它会开始加速。这个想法就是牛顿第二定律：

$$F = m \cdot a$$

这里，F 是力，m 是质量，a 是加速度。由于力等于质量点乘加速度，所以加速度可以通过力除以质量得到。给定一个力，这个等式就可以计算出加速度。

通常而言，我们想表示加速度为一个接受时间的函数，$a(t)$。现在加速度有了与速度和位置的关系。这个关系就是位置函数 ($r(t)$) 的导数就是速度函数 ($v(t)$)，而速度函数的导数就是加速度函数。这里是符号表达式：

$$v(t) = \frac{\mathrm{d}r}{\mathrm{d}t}$$
$$a(t) = \frac{\mathrm{d}v}{\mathrm{d}t}$$

但是这个公式在游戏中求值不是很方便。在游戏中，我们希望给对象一个作用力，然后这个力就持续产生加速度。在我们有了加速度之后，我们就可以得到这段时间内的速度。最终，得到了速度，我们就可以判断物体的位置。所有这些都要在这段时间的每一帧都进行计算。

换句话说，我们要的是对导数反向操作，你可能会想到积分。但不是所有积分我们都感兴趣。关于积分你可能会想到我们最熟悉的不定积分：

$$\int \cos(x)\,\mathrm{d}x = \sin(x) + C$$

但是在游戏中，不定积分没有什么用，首先因为它不能直接用。还有就是，我们希望每一帧都通过加速度算出速度和位置。意味着使用**数值积分**，一种可以每帧都使用其计算积分近似值的方法。如果你要求曲线下的面积，可以通过梯形法则，算出数值积分。本节讲一下游戏中常用的数值积分。

可变时间步长带来的问题

在我们讨论数值积分之前，需要先解决基于物理的移动问题。在使用数值积分之后，你就或多或少地不能使用可变的时间帧。这是因为数值积分的准确性取决于时间步长。步长越短就越精确。

这意味着如果每帧的时间步长都改变，近似值也会每帧变动。如果准确性改变，行为也会有显著的变化。想象你正在玩一款角色可以跳跃的游戏，就像《超级马里奥兄弟》。平时玩游戏的时候，角色的起跳速度都一样。但是突然间，帧率降低，然后你看到马里奥跳得更高了。这种情况是由于数值积分的百分比误差在低帧率的时候放大了，所以跳得更高了。角色跳跃演示如图 7.15 所示。这意味着玩家在慢的机器上会比快的机器跳得更高。

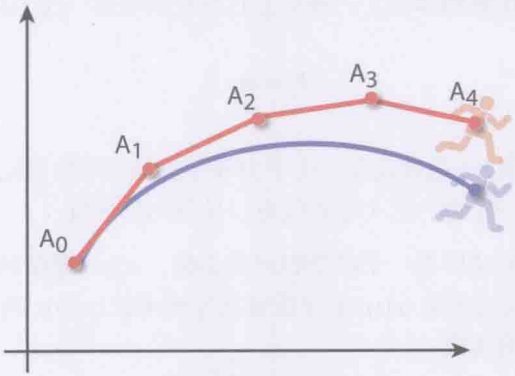


图 7.15 不同时间步长带来不同的跳跃轨迹

由于这个原因，任何游戏使用物理计算位置的时候，都不要使用可变的时间步长。物理计算用可变步长当然是可以的，但是这样就会很复杂。现在，你可以使用第 1 章提到的方法来限制帧率。

力的计算

数值积分让我们可以由加速度算出速度，然后由速度算出位置。但是为了算出加速度，我们需要力和质量。这里有多种多样的力需要考虑。有些力，比如重力，一直作用在物体身上。而有些力可以用**冲量**替代，就是那些只在一帧起作用的力。

举个例子，跳跃可能最先受到冲量的作用而起跳。但是跳跃开始之后，重力的作用下，角色就会回到地面。由于多个力可以同时作用在物体上，在游戏中最常见的做法就是算出所有力的合力，然后除以质量算出加速度：

$$\text{加速度} = \text{合力}/\text{质量}$$

欧拉和半隐式欧拉积分

最简单的数值积分就是**欧拉积分**，以瑞士著名数学家命名。在欧拉积分中，新的位置是由旧的位置加上速度乘以时间步长得到。然后速度以类似的方式通过加速度算出来。一个简单的物体对象的更新函数如清单 7.7 所示。

清单 7.7 物理对象的欧拉积分

```
class PhysicsObject
   // 物体上所有作用力
   List forces
   Vector3 acceleration, velocity, position
   float mass

   function Update(float deltaTime)
      Vector3 sumOfForces = sum of forces in forces
      acceleration = sumOfForces / mass

      // 欧拉积分
      position += velocity * deltaTime
      velocity += acceleration * deltaTime
   end
end
```

虽然欧拉积分很简单，它并没有真正表现得非常准确。一个大问题就是位置是用旧的速度算出来的，而不是时间步长之后的新速度。这样会随着时间的推移让误差不断地积累。

一个简单的改法就是将欧拉积分的位置和速度更新顺序调换。就是说现在位置是使用新的速度来计算。这就是**半隐式欧拉积分**，它会更加合理、更加稳定，著名的游戏物理引擎 Box2D 就用了这种方法。但是，如果我们想要更加准确，我们就要使用更加复杂的数值积分方法。

Verlet 积分法

在 **Verlet 积分法**中，首先算出本次时间步长中点的速度值。然后将它看作平均速度计算整个步长的位置。然后，加速度根据力和质量计算出来，最终利用新的加速度在步长结束的时候计算出速度。Verlet 的实现如清单 7.8 所示。

清单 7.8 Verlet 积分

```
function Update(float deltaTime)
   Vector3 sumOfForces = sum of forces in forces

   // Verlet积分法
```

```
    Vector3 avgVelocity = velocity + acceleration * deltaTime / 2.0f
    // 位置用平均速度算出来
    position += avgVelocity * deltaTime
    // 计算新的加速度和位置
    acceleration = sumOfForces / mass
    velocity = avgVelocity + acceleration * deltaTime / 2.0f
end
```

本质上 Verlet 积分法使用平均速度计算位置。这比起两种欧拉积分都要准确得多。同时计算也更加昂贵，虽然显然比欧拉方法要好，但还不够。

其他积分方法

还有不少其他积分方法可能会在游戏中用到，但是它们有点复杂。它们当中最受欢迎的方法是四阶 Runge-Kutta 方法。它本质上是使用泰勒近似求解的结果表示运动的微分方程的近似解。这个方法无可争议地比欧拉和 Verlet 方法都要准确，但也更慢。对于那些需要高准确度的游戏（比如汽车模拟）来说是有意义的，但是对于多数游戏而言都过重。

角力学

角力学是关于旋转的力学研究。比如说，你可能需要这种物理效果，就是物体围绕另一个物体旋转。就像线性力学有质量、作用力、加速度、速度、位置一样，角力学有转动惯量、力矩、角加速度、角速度和角度。角力学的机制比线性力学还要复杂一些，但也不会复杂很多。就像线性力学一样，角力学也会在旋转的时候用到积分。注重效果的游戏大多会用到角力学，但是由于大多数游戏只用线性力学，所以这里选择只讲线性力学。

物理中间件

回忆一下，第 1 章中提到的中间件就是一些用于解决常见问题的代码库。物理问题的复杂度和广度使得大多数游戏公司都会选择采用中间件而不是自己实现。

3D 游戏最流行的商业物理引擎，毫无疑问就是 Havok 物理引擎。如果你过目一下最近 10 年的 AAA 级游戏，你可以看到大量游戏用的都是 Havok。虽然 Havok 面向商业游戏，但是也有面向微小开发者的版本。

还有一个可选的工业级别的物理引擎就是 PhysX，由于 Unreal3 引擎（和新版本的 Unity）使用而声名大噪。如果你的游戏利润低于 100000 美元则免费提供。

对于 2D 物理引擎来说，目前最流行的是开源的 Box2D（*http://box2d.org*）。核心是用 C++ 编写的，但是有很多其他语言的移植，包括 C#、Java 和 Python。它一个很棒的 2D 物理引擎，基于 2D 物理的游戏，比如《愤怒的小鸟》，也在使用它。

总结

本章是个关于 2D 和 3D 游戏物理的很长的概览。碰撞几何体，比如球体和立方体，经常用于简化碰撞的计算。还有很多其他的交叉测试，包括 AABB vs AABB、线段 vs 平面，而且我们还了解了它们的实现方法。还有就是通过扫略球进行连续性碰撞检测，可以检测两帧之间发生的碰撞。最后，游戏中常见的移动都是用线性力学。所以我们必须使用数值积分方法进行计算，比如用欧拉积分通过速度计算位置，再通过加速度计算速度。

习题

1. 轴对齐包围盒（AABB）与朝向包围盒（OBB）有什么不同？
2. 游戏中最常用的平面表达式是什么？等式中的每个变量是什么意思？
3. 什么是参数方程？如何用参数方程表示光线？
4. 如何高效地检测两个球是否发生交叉？
5. 检测两个 AABB 是否交叉的最佳方法是什么？
6. 在线段与平面交叉中，t 为负值表示什么意思？
7. 即时碰撞检测和连续碰撞检查有什么不同？
8. 在扫略球计算中，判别式有可能为负、为零、为正。这 3 种值分别表示什么意思？
9. 为什么在数值积分中使用可变时间步长不是个好方法？
10. 理论上 Verlet 积分是如何工作的？

相关资料

Ericson, Christer. ***Real-time Collision Detection.*** **San Francisco: Morgan Kaufmann, 2005.** 这本书是碰撞检测大全。书中有各种几何体的碰撞及排列组合。要注意的是，本书涉及大量数学知识，所以在看之前最好先适应数学表达的方法。

Fielder, Glenn. Gaffer on Games – Game Physics. *http://gafferongames.com/game-physics/*. 这个博客讲了很多话题，但是物理学的话题更加引人注目。涵盖了四阶 Runge-Kutta 积分、角力学、弹簧等话题。

第8章 摄像机

摄像机赋予了玩家在3D世界中的视角。在游戏中会用到很多种类型的摄像机，而选择摄像机又是游戏开发初期的基本设计决策。

本章讲了现在的游戏中用到的主要摄像机类型及其背后的计算原理。同时还会深入讲解第4章提过的透视投影。

摄像机的类型

在我们深入了解摄像机的实现之前，了解游戏中常用的摄像机类型会很有帮助。本节不会讨论所有种类的摄像机，但是涵盖了大多数常见的摄像机。

固定摄像机

严格来讲，**固定摄像机**就是那种永远在同一个位置的摄像机。这种固定的摄像机通常只用于非常简单的 3D 游戏。术语“固定摄像机”也可以扩展为根据玩家的位置而摆放在预先定义好的位置。随着玩家在场景中移动，当玩家位置超过某个阈值时，摄像机位置会突然跳到下一个点。摄像机的位置和阈值完全由策划在创建关卡时控制。

固定摄像机一度非常流行，特别在恐怖游戏上，比如早期的《生化危机》。由于玩家不能控制摄像机，这样游戏策划就可以在固定场景里面将敌人藏在玩家看不到的角落。这使得这类型的游戏有了让人紧张的氛围。图 8.1 所示的是场景中几个固定摄像机的顶视图。

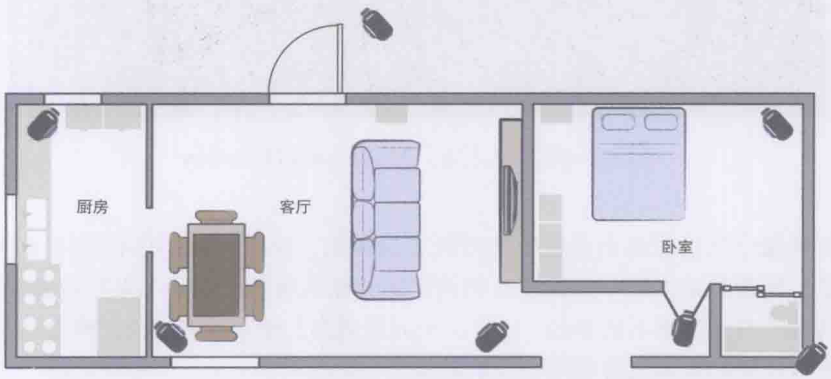


图 8.1　示例场景的顶视图，演示了固定摄像机的摆放

虽然游戏策划知道固定摄像机能够营造紧张气氛，但是玩家会觉得心烦。这也许就是现代的《生化危机》放弃固定摄像机的原因。有的现代游戏仍然使用玩家无法控制的摄像机，但是摄像机会随着玩家的场景移动也动起来。经典的例子就是《战神》系列。

固定摄像机的实现细节就不多说了，但是还会提一个这种早期摄像机的实现方法。一个可能的方法就是让数个凸多边形与一个特定的摄像机对应。然后随着玩家移动，这些凸多边形（见第 7 章的点与多边形的碰撞检测）就能判定玩家进入了哪个摄像机的范围，然后就使用该摄像机。

第一人称摄像机

第一人称摄像机是以玩家的视角来体验游戏世界的，如图 8.2 所示。由于摄像机是角色视角，第一人称摄像机是最让人身临其境的摄像机类型。第一人称摄像机在第一人称射击游戏中非常流行，但是在其他游戏比如《上古卷轴：天际》也会用到。

图 8.2 第一人称的 Quadrilateral Cowboy

第一人称游戏最常见的做法就是在眼睛附近放摄像机，这样其他角色和物体才会有相应的高度。但是，问题是很多第一人称游戏都希望能够显示角色手部。如果摄像机在眼睛位置，当角色向前看的时候是看不到手的。还有一个问题就是，如果玩家的角色模型绘制出来，你会从接近眼睛位置的摄像机看到很奇怪的效果。

为了解决以上问题，大多数第一人称游戏都不会使用普通模型。取而代之的是，使用一个特殊的只有手臂（可能还有腿）的对解剖来讲不正确的位置。这样，就算向前看，玩家总能看到手上有什么。如果使用了这个方法，一些特殊情形，比如看到自己的倒影这种情况就需要考虑了。否则，玩家看到空气中悬挂的手臂，就会被吓到。

跟随摄像机

跟随摄像机会在一个或者多个方向上跟在目标后面。这种摄像机能够在游戏中得到广泛的应用——不管是赛车游戏，跟随着车，就像《火爆狂飙》系列那样，还是第三人称动作/冒险游戏，就像《神秘海域》那样。由于跟随摄像机能够用在非常多的领域中，所以种类繁多。图 8.3 就展示了跟随在汽车后面的摄像机。

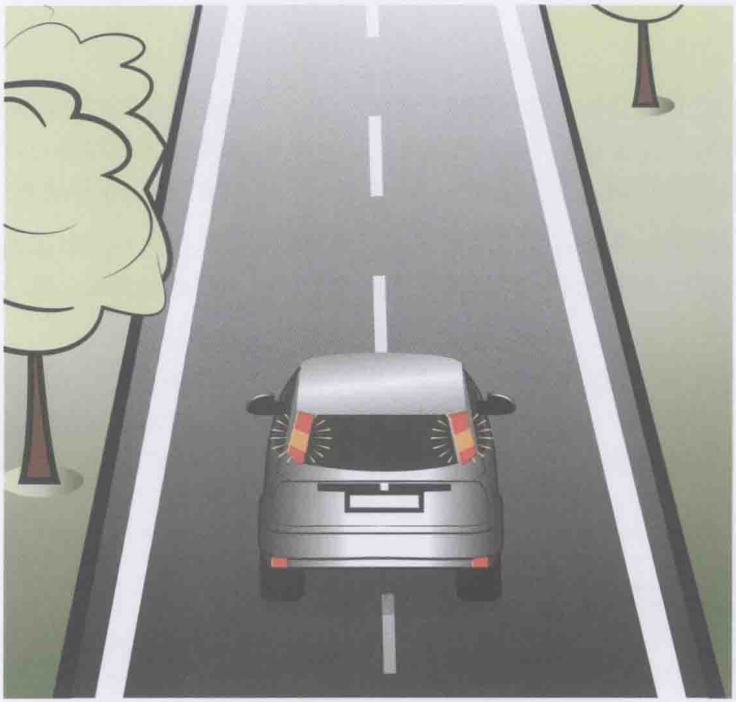


图 8.3　跟随在汽车后面的摄像机

有的跟随摄像机与目标始终保持固定距离，而有的则与目标保持弹性距离。有的在跟随角色的过程中会旋转，而有的不会。有的甚至允许玩家突然转身看背后有什么。有大量的这种类型的摄像机的各种各样的实现。

场景切换摄像机

越来越多的游戏会用到**场景切换**，就是在播放游戏剧情的时候，从玩家的摄像机切过去的一种手法。在 3D 游戏中实现场景切换，要预先在场景中放置动画中用到的固定摄像机。很多场景切换会使用电影设备，比如移动镜头。为了达到效果，会用到样条系统，我们在本章后面会讨论到。

透视投影

就像第 4 章里讲到的，透视投影是具有深度的。就是说，如果物体距离摄像机越来越远，那么看上去就会越来越小。我们在第 4 章简单地接触了一下透视投影的参数，现在是时候讨论更多细节了。

视场

观看世界视野的广度及角度，称为**视场**（FOV）。对人类来说，我们的眼睛提供了 180° 的视野，但是并不是每个角度都有等量的清晰度。双目并视的时候，两只眼都可以同时看到大约 120° 的视场。而剩余的视场在边缘处，能够快速地发现运动，但是不够清晰。

设置投影矩阵的时候，视场是需要考虑的非常重要的因素，因为一个不正确的视场可以导致玩家紧张和眩晕。让我们看一下用高清电视玩家用机玩游戏的玩家，看一下他们对不同视场的反应。

推荐的观看高清电视的距离很大程度取决于向你推荐的人，THX 推荐的观看距离为取对角线长度乘以 1.2。所以 50″ 的电视应该从 60″ 的距离观看。这个距离，电视机会有大约 40° 的**视角度**，就是说电视机会占了观看者 40° 的视场。图 8.4(a) 演示了 THX 推荐的观看距离。

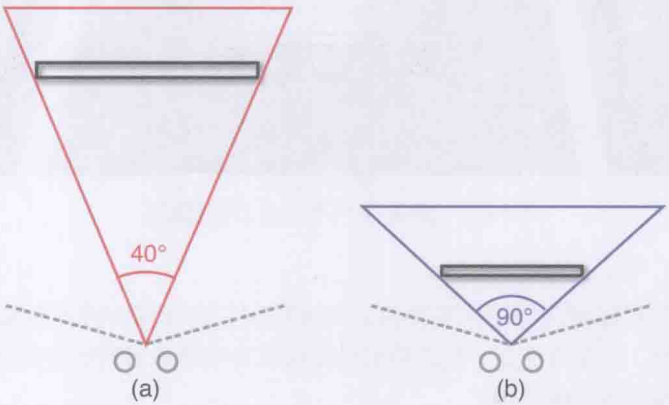


图 8.4 50″ 的高清电视有 40° 的视角度 (a) 和 PC 显示器有 90° 的视角度 (b)

在这种条件下，只要给游戏留下多于 40° 的视场，几乎所有的角色行动都能看得一清二楚。这就是为什么家用机游戏需要大约 65° 的视场。

但是如果将家用机游戏替换成 PC 游戏会怎样？在 PC 的条件下，显示器会占用玩家更多的视场。这种条件下通常有 90° 以上的视场，这个视场及视角度的差异会让一些玩家感到不舒服。这就是为什么游戏世界需要把视场收窄，让大脑回到 90° 的视场感知。由于这个问题，让玩家选择自己习惯的视场是一个不错的选项。

如果视场增加，能够看到的部分便增多。图 8.5 演示了一个示例场景——先看了 65° 的视场，然后看了 90° 的视场。注意一下视场的边缘，可视画面是如何增加的。有的人会争论说，如果玩家采用更大的视场，会得到更大的优势，尤其是在多人游戏当中。但是只要视场限制到最大 120°，就可以认为这个优势是很微弱的。

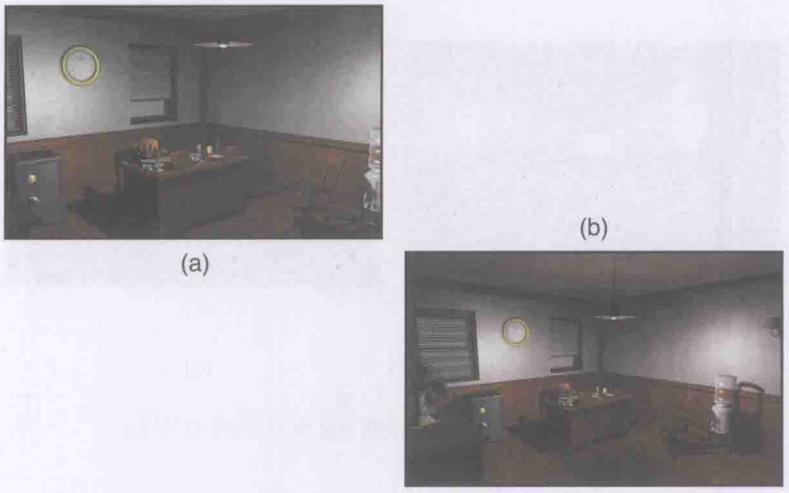


图 8.5　使用 65° 视场的示例场景 (a) 和 90° 的视场

如果视场变得太大，就会有**鱼眼效果**，屏幕的边缘变得弯曲。就类似于摄影中使用了广角镜头一样。大多数游戏不会允许玩家选择太高的视场。

宽高比

宽高比就是观看世界视口的宽度和高度的比率。对于全屏游戏而言，宽高比就通常取决于显示设备所选择分辨率的宽高比。有个例外就是那些可以分屏多人模式的游戏。在这种情况下，游戏世界就有了多个视口，这就是宽高比不等于显示器分辨率的一个例子。最流行的游戏宽高比是 4:3、16:9、16:10。

经典的 1024×768 分辨率（就是横排 1024 个像素，竖排 768 个像素）就是 4:3 的宽高比。但是现在很多设备都不再使用这个宽高比了。今天大多数显示器都使用 16:9，而不是 4:3。标准的高清分辨率 720p（就是 1280×720）就是 16:9 宽高比的例子。高清电视就是为什么 16:9 的宽高比盛行的原因。16:10 是只在电视显示器上看到的宽高比，但是支持这个宽高比的显示器也在下降。

同时支持 4:3 和 16:9 宽高比，需要决定哪个分辨率能够看到更多的游戏世界。常见的方法是 16:9 在横向上看得更多。但是有的游戏，特别是《生化奇兵》的早期版本，采用了相反的方法，16:9 模式下比 4:3 看到得更少。两种方法如图 8.6 所示。

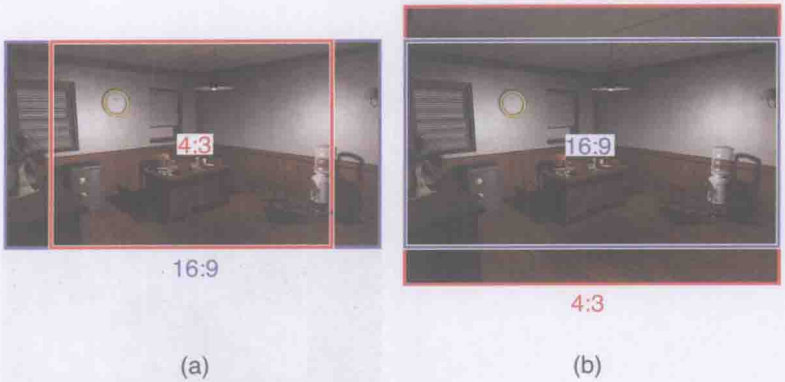


图 8.6 16:9 有更大的 FOV(a) 和更窄的 FOV(b)

摄像机的实现

现在我们已经讨论了数种不同类型的基础摄像机，让我们看一下其中一些类型摄像机的可能实现方式。

基础的跟随摄像机

在基础的跟随摄像机中，摄像机总是直接跟随在某个对象后面，而且保持固定的距离。比如说，一个基础的跟随摄像机跟随着一辆车，如图 8.7 所示。

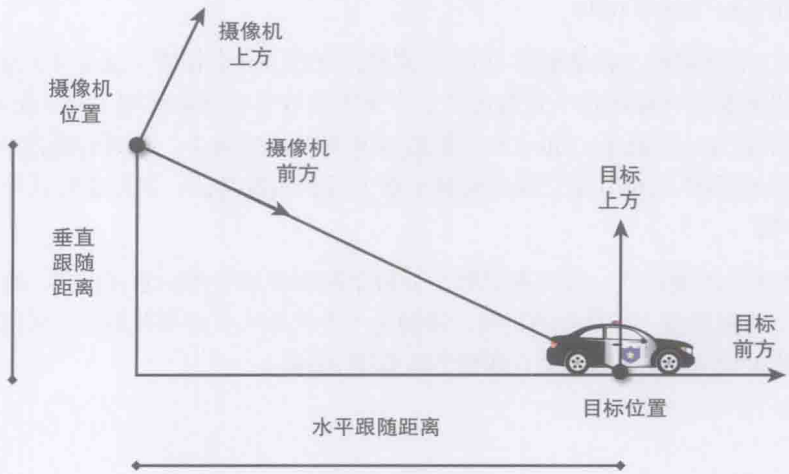


图 8.7 基础跟随摄像机跟随者一辆汽车

回忆一下为摄像机创建观察矩阵，需要 3 个参数：眼睛的位置（摄像机的位置）、摄像机观察的目标，以及摄像机的上方向量。在基础跟随摄像机中，眼的位置可以设置为目标的水平和垂直偏移。在计算出位置之后，就可以计算其他参数，然后传递给 CreateLookAt 函数：

```
// tPos, tUp, tForward = 位置、上方和前方向量
// hDist = 水平跟随距离
// vDist = 垂直跟随距离
function BasicFollowCamera(Vector3 tPos, Vector3 tUp, Vector3 tForward,
                           float hDist, float vDist)
   // 眼睛就是目标位置的偏移量
   Vector3 eye = tPos - tForward * hDist + tUp * vDist

   // 摄像机向前的方向是从眼睛到目标
   Vector3 cameraForward = tPos - eye
   cameraForward.Normalize()

   // 叉乘计算出摄像机的左边及上方向量
   Vector3 cameraLeft = CrossProduct(tUp, cameraForward)
   cameraLeft.Normalize()
   Vector3 cameraUp = CrossProduct(cameraForward, cameraLeft)
   cameraUp.Normalize()

   // CreateLookAt的参数为eye、target，以及up
   return CreateLookAt(eye, tPos, cameraUp)
end
```

虽然基础跟随摄像机会跟着目标在游戏世界中移动，看起来非常僵硬。摄像机总是保持固定距离，没有弹性。当旋转的时候，这个基础跟随行为会让人不知道是世界在转还是人在转。而且，基础跟随摄像机没有一个合理的速度，它的速度就是目标的速度。由于以上种种原因，基础跟随摄像机很少在游戏中使用。虽然它提供了简单的解决方案，但是显得很不优雅。

一个简单的改善方法就是让摄像机有一个随着跟随目标速度调整跟随距离的函数。比如说平时跟随的时候距离为 100，但是当目标全速移动的时候，这个距离为 200。这个简单改变能够提升基础跟随摄像机的速度感，但是还有很多问题没法解决。

弹性跟随摄像机

有了**弹性跟随摄像机**，就不会由于目标的朝向或者位置改变而突然变化，而是摄像机会在几帧的过程中逐渐变化。实现方式是同时设定好理想位置与现实位置。理想位置每帧立刻变化，就跟基础跟随摄像机一样（可能会有跟随距离调整函数）。然后真正的摄像机位置在后续几帧慢慢跟随到理想位置上，这样就能够创造平滑的镜头效果。

这种实现方式是通过虚拟弹簧将理想摄像机和真实摄像机连接到一起实现的。每当理想摄像机位置变化时，弹簧都被拉伸。如果理想摄像机位置不变，随着时间的推移，真实位置总会被调整到理想位置。弹簧、理想摄像机、真实摄像机的配置如图 8.8 所示。

图 8.8 一个弹簧将理想摄像机与真实摄像机连接起来

弹簧的效果可以由弹性常量来控制。这个常量越大，弹簧就越僵硬，就是说摄像机归位得越快。实现弹性跟随摄像机，需要确定每帧的摄像机速度和真实摄像机位置。因此最简单的实现就是使用一个类来实现。这个算法大概的工作方式是首先基于这个弹性常量计算出加速度。然后将加速度通过数值积分计算出摄像机的速度，然后再进一步计算位置。完整的算法如清单 8.1 所示，就像 *Game Programming Gems 4* 中展示的弹性摄像机算法。

清单 8.1 弹性摄像机类

```
class SpringCamera
   // 水平和垂直跟随距离
   float hDist, fDist
   // 弹性常量：  越高表示越僵硬
   // 一个好的初始值很大程度取决于你想要的效果
   float springConstant
   // 阻尼常量由上面的值决定
   float dampConstant

   // 速度和摄像机真实位置向量
   Vector3 velocity, actualPosition

   // 摄像机跟随的目标
   // （有目标的位置、向前向量、向上向量）
   GameObject target

   // 最终的摄像机矩阵
   Matrix cameraMatrix

   // 这个帮助函数从真实位置及目标计算出摄像机矩阵
   function ComputeMatrix()
      // 摄像机的前向是从真实位置到目标位置
      Vector3 cameraForward = target.position - actualPosition
      cameraForward.Normalize()

      // 叉乘计算出摄像机左边，然后计算出上方
```

```
    Vector3 cameraLeft = CrossProduct(target.up, cameraForward)
    cameraLeft.Normalize()
    Vector3 cameraUp = CrossProduct(cameraForward, cameraLeft)
    cameraUp.Normalize()

    // CreateLookAt参数为eye、target及up
    cameraMatrix = CreateLookAt(actualPosition, target.position,
                                cameraUp)
end

// 初始化常量及摄像机，然后初始化朝向
function Initialize(GameObject myTarget, float mySpringConstant,
                    float myHDist, float myVDist)
    target = myTarget
    springConstant = mySpringConstant
    hDist = myHDist
    vDist = myVDist

    // 阻尼常量来自于弹性常量
    dampConstant = 2.0f * sqrt(springConstant)

    // 起初，设置位置为理想位置
    // 就跟基础跟随摄像机的眼睛位置一样
    actualPosition = target.position - target.forward * hDist +
                     target.up * vDist

    // 初始化摄像机速度为0
    velocity = Vector3.Zero

    // 设置摄像机矩阵
    ComputeMatrix()
end

function Update(float deltaTime)
    // 首先计算理想位置
    Vector3 idealPosition = target.position - target.forward * hDist +
        target.up * vDist

    // 计算从理想位置到真实位置的向量
    Vector3 displacement = actualPosition - idealPosition
    // 根据弹簧计算加速度，然后积分
    Vector3 springAccel = (-springConstant * displacement) -
                          (dampConstant * velocity)
    velocity += springAccel * deltaTime
    actualPosition += velocity * deltaTime
```

```
        // 更新摄像机矩阵
        ComputeMatrix()
    end
end
```

弹性摄像机的最大好处就是当目标对象旋转的时候，摄像机会有一定的延迟才开始转动。旋转出来的效果要比基础跟随摄像机要好得多。弹性摄像机效果比基础跟随摄像机好很多，但是计算量没多多少。

旋转摄像机

旋转摄像机会在目标附近旋转。旋转摄像机最简单的实现方法就是存储摄像机的位置及与目标的偏移，而不是直接记录摄像机的世界坐标系位置。这是因为旋转总是关于原点（希望你还记得第 4 章的内容）旋转的。所以如果摄像机位置作为偏移记录下来，旋转就可以以目标对象为原点进行旋转，这样就得到了我们想要的旋转效果。旋转摄像机如图 8.9 所示。

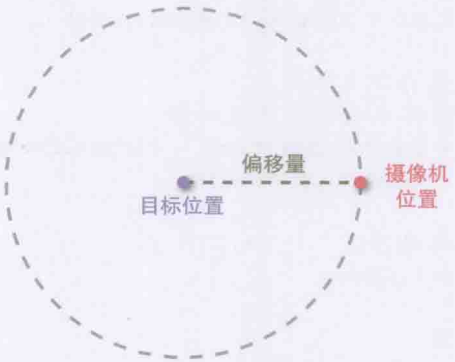


图 8.9 旋转摄像机

旋转摄像机的控制模式通常可以偏航（yaw）和俯仰（pitch），但不能滚转（roll）。由于输入方式通常将偏航和俯仰作为两个独立值输入，将旋转实现为两部分会比较好。首先，关于世界坐标系向上向量旋转摄像机（偏航），然后关于摄像机的左侧向量进行旋转（俯仰）。当摄像机比目标要低的时候，你需要在两个方向上旋转摄像机才能得到正确的行为。

清单 8.2 使用了四元数来实现这个旋转，你也可以使用矩阵来替代。值得一提的是，还有另一种完全不同的方法来实现旋转摄像机旋转，就是使用球形坐标系，但是我发现这里演示的版本更加容易。

清单 8.2 旋转摄像机

```
class OrbitCamera
   // 摄像机向上的向量
   Vector3 up
   // 目标偏移
   Vector3 offset
   // 目标对象
   GameObject target
   // 最终的摄像机矩阵
   Matrix cameraMatrix

   // 初始化摄像机状态
   function Initialize(GameObject myTarget, Vector3 myOffset)
      // 在y轴朝上的世界里，up向量就是Y轴
      up = Vector3(0,1,0)

      offset = myOffset
      target = myTarget

      // CreateLookAt参数为eye、target和up
      cameraMatrix = CreateLookAt(target.position + offset,
                                  target.position, up)
   end

   // 根据这一帧的yaw/pitch增量角度进行更新
   function Update(float yaw, float pitch)
      // 创建一个关于世界向上的四元数
      Quaternion quatYaw = CreateFromAxisAngle(Vector3(0,1,0), yaw)
      // 通过这个四元数变换摄像机偏移
      offset = Transform(offset, quatYaw)
      up = Transform(up, quatYaw)

      // 向前就是就是target.position - (target.position + offset)
      // 刚好就是-offset
      Vector3 forward = -offset
      forward.Normalize()
      Vector3 left = CrossProduct(up, forward)
      left.Normalize()

      // 创建关于摄像机左边旋转的四元数值
      Quaternion quatPitch = CreateFromAxisAngle(left, pitch)
      // 通过这个四元数变换摄像机偏移
      offset = Transform(offset, quatPitch)
      up = Transform(up, quatPitch)
```

```
        // 现在计算矩阵
        cameraMatrix = CreateLookAt(target.position + offset,
                                    target.position, up)
    end
end
```

在一些游戏里面，可能同时想要有弹性摄像机和手动的旋转摄像机。两种方式完全可以组合起来得到跟随而且旋转的特性，但是具体实现就当作练习留给读者。

第一人称摄像机

使用第一人称摄像机，摄像机的位置总是放在角色的相对位置上。所以当角色在世界中移动的时候，摄像机依然是玩家的位置加上偏移。虽然摄像机偏移总是不变，目标位置可以不断变化。这是因为大多数第一人称游戏都支持到处看但是不改变位置的功能。具体情形如图 8.10 所示。

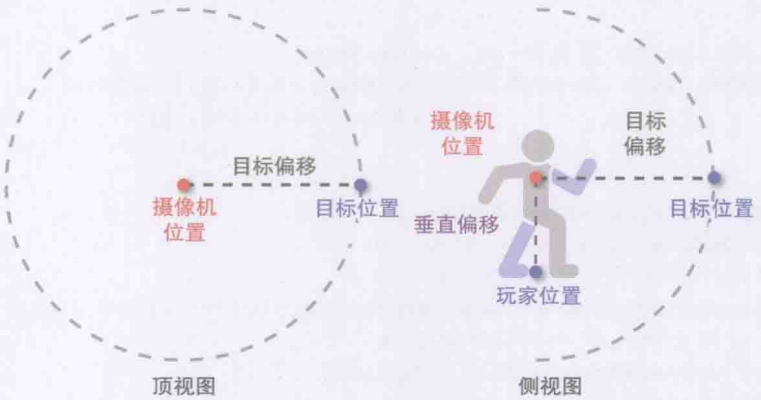


图 8.10 第一人称摄像机的实现

旋转的实现其实和旋转摄像机有点类似。主要区别在于目标偏移在旋转，而不是摄像机偏移。但是这种摄像机还有一些不同。其中之一就是当目标是关于向上向量偏移的时候，角色也需要跟着转动。还有一点就是俯仰度有一个取值范围。

由于这些变化，第一人称摄像机通常会将俯仰度和偏航度记录下来，而不是记录增量。这意味着旋转总是作用到最初的目标偏移，而不是旋转摄像机中增量偏移的做法。第一人称摄像机的实现如清单 8.3 所示。

清单 8.3 第一人称摄像机

```
class FirstPersonCamera
    // 以角色位置为原点的摄像机偏移
    // 对于y轴向上的世界，向上就是(0, value, 0)
    Vector3 verticalOffset
    // 以摄像机为原点的目标位置偏移
    // 对于z轴向前的世界，向前就是(0, 0, value)
    Vector3 targetOffset
    // 总的偏航和俯仰角度
    float totalYaw, totalPitch
    // 摄像机所在的玩家
    GameObject player
    // 最终的摄像机矩阵
    Matrix cameraMatrix

    // 初始化所有摄像机参数
    function Initialize(GameObject myPlayer, Vector3 myVerticalOffset,
                        Vector3 myTargetOffset)
        player = myPlayer
        verticalOffset = myVerticalOffset
        targetOffset = myTargetOffset

        // 最开始，没有任何偏航和俯仰
        totalYaw = 0
        totalPitch = 0

        // 计算摄像机矩阵
        Vector3 eye = player.position + verticalOffset
        Vector3 target = eye + targetOffset
        // 在y轴向上的世界里
        Vector3 up = Vector3(0, 1, 0)
        cameraMatrix = CreateLookAt(eye, target, up)
    end

    // 根据这一帧的增量偏航和俯仰进行更新
    function Update(float yaw, float pitch)
        totalYaw += yaw
        totalPitch += pitch

        // 对俯仰进行Clamp
        // 在这种情况下，范围为角度45°（弧度约为0.78）
        totalPitch = Clamp(totalPitch, -0.78, 0.78)

        // 目标在旋转之前偏移
        // 真实偏移则是在旋转之后
```

```
        Vector3 actualOffset = targetOffset

        // 关于y轴进行偏航旋转，旋转真实偏移
        Quaternion quatYaw = CreateFromAxisAngle(Vector3(0,1,0), totalYaw)
        actualOffset = Transform(actualOffset, quatYaw)

        // 为了俯仰计算左边向量
        // 前向就是偏航之后的真实偏移（经过正规化）
        Vector3 forward = actualOffset
        forward.Normalize()
        Vector3 left = CrossProduct(Vector3(0, 1, 0), forward)
        left.Normalize()

        // 关于左边进行俯仰，旋转真实偏移
        Quaternion quatPitch = CreateFromAxisAngle(left, totalPitch)
        actualOffset = Transform(acutalOffset, quatPitch)

        // 现在构造摄像机矩阵
        Vector3 eye = player.position + verticalOffset
        Vector3 target = eye + actualOffset
        // 在这种情况下，我们可以传递向上向量，因为我们永远向上。
        cameraMatrix = CreateLookAt(eye, target, Vector3(0, 1, 0))
    end
end
```

样条摄像机

讲解数学定义就会有点过多了，简单来讲，**样条**可以看作为曲线，用线上的点定义的。样条在游戏中很常见，因为使用它进行插值能够在整条曲线上得到平滑的效果。这在切换场景的时候很有用，这意味着可以让摄像机跟着预定义的曲线移动。

有很多种不同的样条，最简单的一种是 **Catmull-Rom** 样条。这种样条允许邻近的点插值，这些点里面有一个控制点在前、两个激活点在后。举个例子，如图 8.11 所示，P_1 和 P_2 就是激活点（分别在 $t=0$ 和 $t=1$ 处），而 P_0 和 P_3 就是在前面和后面的控制点。尽管图中只有 4 个点，但实际上是没有限制的。只要在前后加上控制点，曲线就可以无限延长下去。

给定 4 个控制点，这就可以计算 t 值介于 0 到 1 的所有样条，等式如下：

$$\begin{aligned} P(t) = {} & 0.5 \cdot (2 \cdot P_1) + (-P_0 + P_2) \cdot t + \\ & (2 \cdot P_0 - 5 \cdot P_1 + 4 \cdot P_2 - P_3) \cdot t^2 + \\ & (-P_0 + 3 \cdot P_1 - 3 \cdot P_2 + P_3) \cdot t^3) \end{aligned}$$

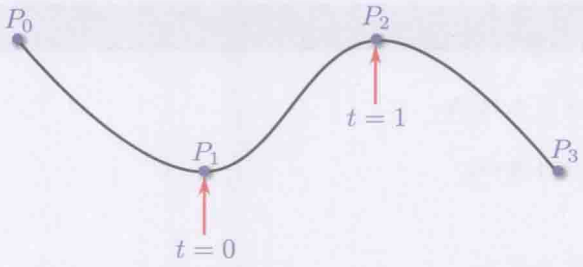


图 8.11 最小顶点数的 Catmull-Rom 样条

注意这个等式要在控制点均匀间距的情况下使用。在任何情况下，有了这个等式，一个简单的支持任意数量点的 Catmull-Rom 样条类实现方法如下：

```
class CRSpline
   // Vector3s的数组（动态数组）
   Vector controlPoints

   // 第一个参数为t=0对应的控制点
   // 第二个参数为t值
   function Compute(int start, float t)
      // 检查start - 1、start、start + 1以及start + 2都要存在
      ...

      Vector3 P0 = controlPoints[start - 1]
      Vector3 P1 = controlPoints[start]
      Vector3 P2 = controlPoints[start + 1]
      Vector3 P3 = controlPoints[start + 2]

      // 使用Catmull-Rom公式计算位置
      Vector3 position = 0.5*((2*P1)+(-P0+P2)*t+
                         (2*P0-5*P1+4*P2-P3)*t*t+
                         (-P0+3*P1-3*P2+P3)*t*t*t)

      return position
   end
end
```

这个公式也可以用于计算 t 介于 0 到 1 之间的切线。首先，计算任意你想要的 t 位置。然后，计算 t 加很小的增量 Δt 的位置。在有了两个位置之后，就可以构造一个 $P(t)$ 到 $P(t+\Delta t)$ 的向量并且正规化，这样就能够近似地得到切线。这种方式可以看作是数值微分。

如果摄像机跟着样条走，那么切线就能够表示摄像机的朝向。提供了样条路径而且不允许摄像机上下颠倒，所以在有了位置和朝向之后，就可以构造摄像机矩阵了。样条摄像机如清单 8.4 所示。

清单 8.4 样条摄像机

```
class SplineCamera
   // 摄像机跟随的样条路径
   CRSpline path
   // 当前控制点索引及t值
   int index
   float t
   // speed是t每秒变化率
   float speed
   // 摄像机矩阵
   Matrix cameraMatrix

   // 给定当期索引和t，计算摄像机矩阵
   function ComputeMatrix()
      // eye就是样条所在的t及index对应的位置
      Vector3 eye = path.Compute(index, t)
      // 给出一个稍微前一点的点
      Vector3 target = path.Compute(index, t + 0.05f)
      // 假定y轴朝上
      Vector3 up = Vector3(0, 1, 0)

      cameraMatrix = CreateLookAt(eye, target, up)
   end

   function Initialize(float mySpeed)
      // 初始index应该为1（因为0是最初的P0）
      index = 1
      t = 0.0f
      speed = mySpeed

      ComputeMatrix()
   end

   function Update(float deltaTime)
      t += speed * deltaTime

      // 如果t>=1.0f，我们可以移动到下一个控制点
      // 这里代码假设速度不会太快，以至于一帧就超过两个控制点
      if t >= 1.0f
         index++
         t = t - 1.0f
      end

      // 应该检查index+1和index+2是否为有效点
      // 如果不是，这条样条就完成了
```

```
        ...

        ComputeMatrix()
    end
end
```

摄像机支持算法

前面提到的实现能够让摄像机基本上顺利工作。但是基本运作方式只是实现摄像机的一小部分。为了实现更好的摄像机，还要考虑额外的支持算法。

摄像机碰撞

摄像机碰撞致力于解决很多类型摄像机都有的问题，那是在摄像机与目标之间有一个不透明物体的时候。最简单的方法（但不是最佳的）就是从目标位置向摄像机位置进行光线投射。如果光线碰撞到任何物体，可以让摄像机移动到阻挡摄像机的物体前面，如图 8.12 所示。不幸的是，这种方法有点突然。一个更好的方法是做一个物理对象表示摄像机，但是那就超出了本节的范围。

图 8.12　摄像机被物体阻挡 (a) 和摄像机向前从而没有阻挡 (b)

另外一个要考虑的问题是在摄像机太过靠近目标的时候。回忆一下，近平面就在摄像机前面一点点，意味着太近的摄像机会让对象消失一部分。一个流行的解决方案是让对象在摄像机太过靠近的时候完全消失或者淡出。

淡出方案有时候也常用于当摄像机与目标之间有阻挡的时候。很多第三人称动作游戏都使用这种方法。

拣选

拣选就是通过点击的方式选择 3D 世界中物体的能力。拣选在 RTS 游戏中很常见，比如第 14 章中的塔防游戏。在那款游戏中，拣选的实现是用来在世界中选择不同的防御塔（以及放置防御塔的位置）。虽然拣选不是摄像机算法，但是它与摄像机和投影紧密相关。

回忆一下，将一个点从世界空间变换到投影空间，它必须乘以一个摄像机矩阵，再乘以一个投影矩阵。但是，鼠标的位置是屏幕空间上的一个 2D 点。我们要做的是将这个 2D 点从屏幕空间变换回世界空间去，称之为**反投影**。为了实现反投影，我们需要一个矩阵可以做逆向矩阵变换操作。对于以行为主的表示，我们需要摄像机矩阵乘以投影矩阵的逆矩阵：

$$\text{unprojection} = (\text{camera} \times \text{projection})^{-1}$$

但是 2D 点不能乘以 4×4 矩阵，所以在这个点乘以矩阵之前，我们必须将其转换到齐次坐标系。这就需要将 z 和 w 分量添加到 2D 点上。z 分量通常设为 0 或 1，取决于该点是放置在近平面还是远平面。而作为一个顶点，w 分量总为 1。下面的 Unproject 函数接受一个 4D 向量，因为它假设已经将它的 z 和 w 分量设置好了：

```
function Unproject(Vector4 screenPoint, Matrix camera, Matrix projection)
   // 计算camera * projection的逆矩阵
   Matrix unprojection = camera * projection
   unprojection.Invert()

   return Transform(screenPoint, unprojection)
end
```

Unproject 可以用来计算两个点：鼠标位置反投影到近平面（$z = 0$）和鼠标位置反投影到远平面（$z = 1$）。这两个点可以作为光线投射的起点和终点。由于光线投射有可能与多个物体交叉，游戏应该选择最近的那个。图 8.13 显示了拣选的基本过程。

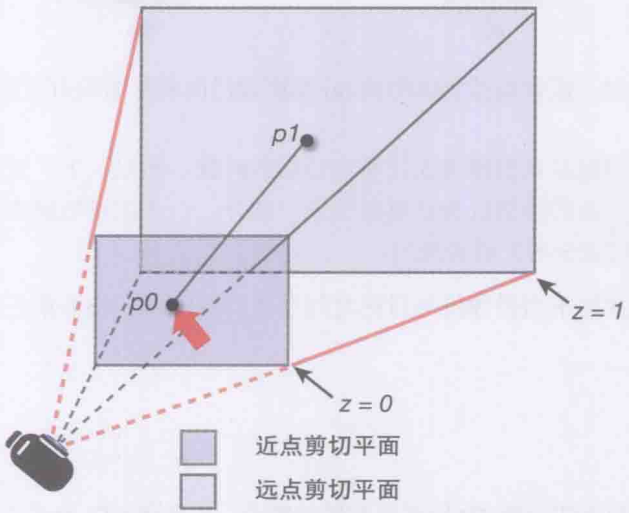


图 8.13 拣选会从近平面到远平面投射一条光线然后拣选与线段交叉的对象

总结

摄像机是3D游戏的核心组件。你可能会用到很多不同类型的摄像机，而本章讨论了很多它们的实现。第一人称摄像机能够让玩家身临其境。跟随摄像机通常用于跟随角色或者汽车。旋转摄像机关于对象进行环绕，而样条摄像机则可能用于切换场景。最后，许多游戏需要实现拣选来让玩家点击并选择目标。

习题

1. 视场表示什么？过窄的视场会引起什么问题？
2. 如何配置基本的以固定距离跟随目标的摄像机？
3. 弹性跟随摄像机在哪方面比基础跟随摄像机要好？
4. 在实现旋转摄像机的时候，该怎么保存摄像机的位置？
5. 第一人称摄像机是如何跟踪目标位置的？
6. 什么是Catmull-Rom样条？
7. 样条摄像机有什么用？
8. 跟随摄像机一直被摄像机与目标之间的物体阻挡着。有哪两种办法可以解决这个问题？
9. 在反投影中，z分量为0和为1是什么意思？
10. 反投影是如何实现拣选的？

相关资料

Haigh-Hutchinson, Mark. *Real-Time Cameras.* Burlington: Morgan Kaufmann, 2009. 这本书广泛地讲了各种不同的游戏摄像机，作者在《银河战士》中实现了各种优秀的摄像机系统。

第9章
人工智能

本章会谈到人工智能在游戏应用中的3个方面：寻路、基于状态的行为机制、策略/计划。寻路算法可以判定非玩家角色（NPC）如何在游戏世界中移动。基于状态的行为驱动角色做出相应的行为。最后，策略/计划在大型AI计划的时候是必要的需求，比如实时战略（RTS）游戏。

“真”AI 与游戏 AI

在传统计算机科学中，许多人工智能的研究都趋向于复杂形式的 AI，包括遗传算法和神经算法。但是这些复杂的算法在计算机和游戏中应用还存在限制。这存在两个主要原因，第一个原因是复杂的算法需要大量的计算时间。大多数游戏只能分出它们每帧的小部分时间在 AI 上，这意味着高效比复杂重要。另外一个主要原因就是，游戏 AI 通常都有良好的行为定义，通常都是在策划的控制之下，而传统的 AI 专注于解决更加模糊而广泛的问题。

在很多游戏中，AI 行为只是一种随机变化的状态机规则的组合，但还是有几个主要的例外。AI 对于复杂的棋牌游戏，比如象棋或者围棋，需要决策树支持，这是传统游戏理论的基石。但是棋牌游戏在某一时刻的行动选择相比起其他游戏来讲还是要小一些。虽然对于象棋 AI 考虑几秒钟做出决策是可以接受的，但是大多数游戏都不会这么奢侈。也有一些游戏实时做出很复杂的算法，令人印象深刻，但那些是特例。一般来讲，游戏中的 AI 就是智能感知。如果玩家觉得敌人的 AI 或者队友的 AI 行为很聪明，这个 AI 系统就已经成功了。

但也不是每个游戏都需要 AI 算法。一些简单的游戏，比如单人跳棋和俄罗斯方块就没有这样的算法。哪怕是一些复杂的游戏也可能没有 AI，比如《摇滚乐队》。对于那些 100% 多人对战没有 NPC 的游戏来说也一样。但是对于任意一款设计指定 NPC 与玩家交互的游戏来说，AI 算法是必需的。

寻路

寻路就是一个看似简单问题的解：给定点 A 和 B，AI 该怎么智能地在游戏世界中行走？

这个问题的复杂来自于实际上 A 和 B 之间存在大量的路径可走，但只有一条是最佳的。比如说，在图 9.1 中，红色和蓝色路径都是 A 到 B 的潜在路径，但是蓝色路径客观上比红色路径要好，简单来讲就是更短。

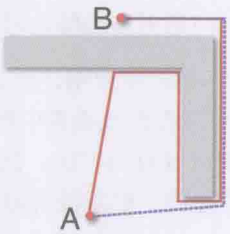


图 9.1　A 到 B 的两条路径

所以只是找到一条两点之间的有效路径是不够的。理想的寻路算法需要查找所有可能的情况，然后比较出最好的路径。

搜索空间的表示

最简单的寻路算法设计就是将**图**作为数据结构。一个图包含了多个**节点**，连接任意邻近的点组成**边**。在内存中表示图有很多种方法，但是最简单的是邻接表。在这种表示中，每个节点包含了一系列指向任意邻近节点的指针。图中的完整节点集合可以存储在标准的数据结构容器里。图 9.2 演示了简单的图的可视化形象和数据表示。

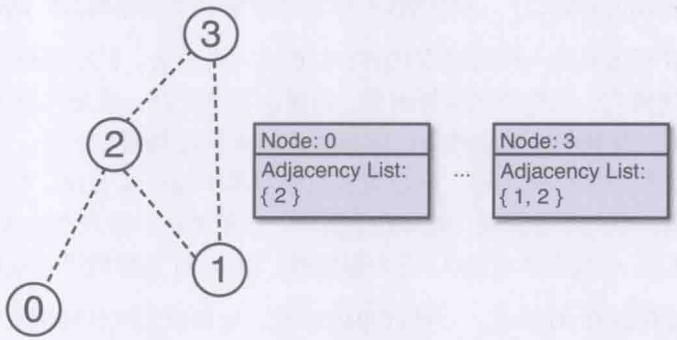


图 9.2 一个图的例子

这意味着在游戏中实现寻路的第一步是如何将游戏世界用图来表示。这里有多种方法。一种简单的方法就是将世界分区为一个个正方形的格子（或者六边形）。在这种情况下，邻近节点就是格子中邻近的正方形。这个方法在回合制策略游戏中很流行，比如《文明》或者 *XCOM*（见图 9.3）。

但是，对于实时动作游戏，NPC 通常不是在网格上一个正方形一个正方形地走。由此，在主流游戏中要么使用路点要么使用导航网格。上面两种方法，都可以手工在场景编辑器中构造数据。

但是手工输入数据不仅烦琐而且容易出错，所以大多数引擎都会让这个过程自动化。自动生成数据的算法超出了本书的范围，但是更多的信息可以在本书的参考资料中找到。

寻路节点最早在第一人称射击游戏（FPS）中使用，由 id Software 在 20 世纪 90 年代早期推出。通过这种表示方法，关卡设计师可以在游戏世界中摆放那些 AI 可以到达的位置。这些路点直接被解释为图中的节点。而边则可以自动生成。比如让设计师手动将节点组合在一起，可以自动处理判断两个点之间是否有障碍。如果没有障碍，那么边就会在两点之间生成。

路点的主要缺点是 AI 只能在节点和边缘的位置移动。这是因为即使路点组成三角形，也不能保证三角形内部就是可以行走的。通常会有很多不能走的区域，所以寻路算法需要认为不在节点和边缘上的区域都是不可走的。

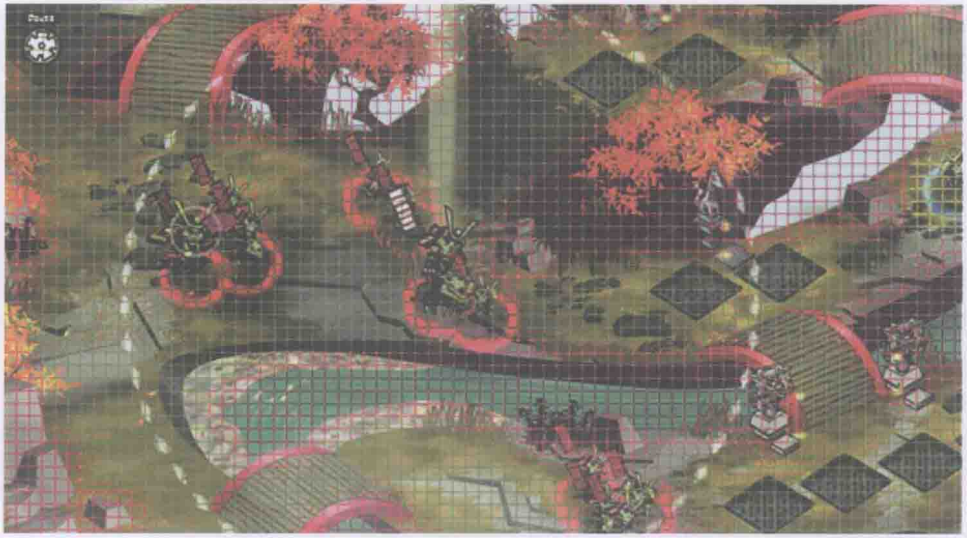


图 9.3 《幕府将军的头骨》用 AI 调试模式演示它用于寻路的格子

实际上，当部署路点之后，游戏世界中就会要么有很多不可到达的区域要么有很多路点。前者是不希望出现的状况，因为这样会让 AI 的行为显得不可信而且不自然。而后者缺乏效率。越多的节点就会有越多的边缘，寻路算法花费的时间就会越长。通过路点，在性能和精确度上需要折中。

一个可选的解决方案就是使用**导航网格**。在这种方法中，图上的节点实际上就是凸多边形。邻近节点就是简单的任意邻近的凸多边形。这意味着整个游戏世界区域可以通过很少数量的凸多边形表示，结果就是图上的节点特别少。如图 9.4 所示的是用游戏中同一个房间同时表示为路点和导航网格的结果比较。

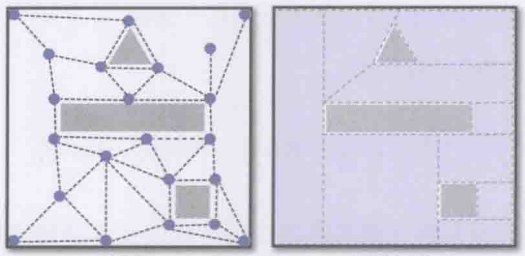


图 9.4 一个房间的两种表示

通过导航网格，在凸多边形内部的任意位置都认为是可走的。这意味着 AI 有了大量的空间可以行走，因此寻路可返回更自然的路径。

导航网格还有其他一些优点。假设游戏中有牛和小鸡在农场中行走。由于小鸡比牛小很多，因此有一些区域只有小鸡可到达，而牛却不行。如果这个游戏使用路点，它通常需要两份图：每份图对应一种生物。这样，牛只能在自己相应的路点行走。与之相反，由于导航网格中每个节点都是凸多边形，计算牛能否进入不会花太多时间。因此，我们可以只用一份导航网格，并且计算哪些地方牛可以到达。

还有一点就是导航网格完全可以自动生成，这也是今天为什么使用路点的游戏越来越少的原因。比如说，多年来虚幻引擎使用路点作为寻路空间的表示。其中一款使用路点的虚幻引擎的游戏就是《战争机器》。而且，最近几年的虚幻引擎已经使用导航网格替代路点。在后来的《战争机器》系列，比如《战争机器 3》就使用的是导航网格，这个转变引起工业上大量转用导航网格。

话虽这么说，但是寻路空间的表示并不完全会影响寻路算法的实现。在本节中的后续例子中，我们会使用正方形格子来简化问题。但是寻路算法仍不关心数据是表示为正方形格子、路点，或是导航网格。

可接受的启发式算法

所有寻路算法都需要一种方法以数学的方式估算某个节点是否应该被选择。大多数游戏都会使用**启发式**，以 $h(x)$ 表示，就是估算从某个位置到目标位置的开销。理想情况下，启发式结果越接近真实越好。如果它的估算总是保证小于等于真实开销，那么这个启发式是**可接受的**。如果启发式高估了实际的开销，这个寻路算法就会有一定概率无法发现最佳路径。

对于正方形格子，有两种方式计算启发式，如图 9.5 所示。**曼哈顿距离**是一种在大都市估算城市距离的方法。某个建筑可以有 5 个街区远，但不必真的有一条路长度刚好为 5 个街区。曼哈顿距离认为不能沿对角线方向移动，因此也只有这种情况下才能使用启发式。如果对角线移动是被允许的，则曼哈顿距离会经常高估真实开销。

在 2D 格子中，曼哈顿距离的计算如下：

$$h(x) = |\text{start.x} - \text{end.x}| + |\text{start.y} - \text{end.y}|$$

第二种计算启发式的方法就是**欧几里得距离**。这种启发式的计算使用标准距离公式然后估算直线路径。不像曼哈顿距离，欧几里得距离可以用在其他寻路表示中计算启发式，比如路点或者导航网格。在我们的 2D 格子中，欧几里得距离为：

$$h(x) = \sqrt{(\text{start.x} - \text{end.x})^2 + (\text{start.y} - \text{end.y})^2}$$

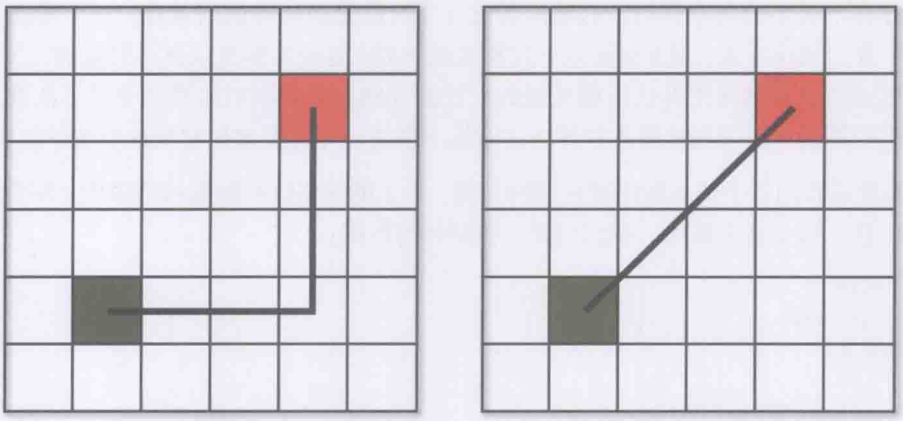


图 9.5　曼哈顿和欧几里得启发式

贪婪最佳优先算法

在有了启发式之后，可以开始实现一个相对简单的算法：**贪婪最佳优先算法**。一个算法如果没有做任何长期计划而且只是马上选择最佳答案的话，则可以被认为是贪婪算法。在贪婪最佳优先算法的每一步，算法会先看所有邻近节点，然后选择最低开销的启发式。

虽然这样看起来理由充足，但是最佳优先算法通常得到的都是次优的路径。看看图 9.6 中的表示。绿色正方形是开始节点，红色正方形是结束节点，灰色正方形是不可穿越的。箭头表示贪婪最佳优先算法的路径。

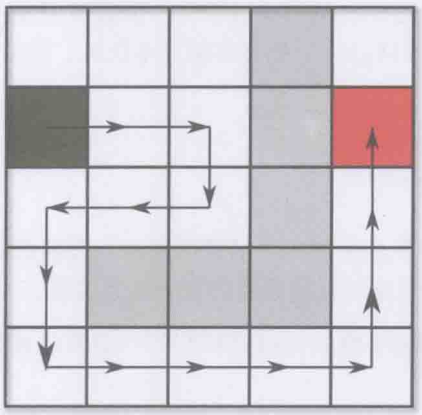


图 9.6　贪婪最佳优先路径

路径上存在不必要的向右移动，这是因为这在当时就是最佳的访问节点。一个理想的路径应该是一开始就往下走，但是这要求一定程度的计划，这是贪婪算法所不具备的。大多数游戏都需要比贪婪最佳优先算法所能提供的更好的寻路。但是本章后续的寻路算法都基于贪婪最佳优先算法，所以先理解贪婪算法才能往下继续，先看看如何实现这个贪婪算法。

首先，先看看我们每个节点所需要存储的数据。为了能够将这些数据构造成图，需要有额外的邻近信息。对于这个算法，我们只要一些额外的数据：

```
struct Node
   Node parent
   float h
end
```

那个 parent 成员变量用于跟踪哪个节点是当前访问的。注意到像 C++ 那样的语言，parent 可能是个指针，而在其他语言中（比如 C#），类可能天然地以引用传递。parent 成员的价值在于构造链表，能够从终点回到起点。当算法完成的时候，parent 链表就可以通过遍历得到最终路径。

浮点数 h 存储了某个节点的 $h(x)$ 的值，这个值导致在选择节点的时候会偏向于 h 值最小的节点。

算法的下一个组件就是用于临时存储节点的容器：开放集合和封闭集合。**开放集合**存储了所有目前需要考虑的节点。由于找到最低 $h(x)$ 值开销节点的操作是很常见的，所以对于开放集合可以采用某种类似于二叉堆或者优先级队列的容器。

而**封闭集合**则包含了所有已经被算法估值的节点。一旦节点在封闭集合中，算法不再对其进行考虑。由于经常会检查一个节点是否存在于封闭集合里，故会使用搜索的时间复杂度优于 $O(n)$ 的数据结构，比如二叉搜索树。

现在我们就有了贪婪最佳优先算法所需要的组件。假设有开始节点和结束节点，而且我们需要计算两点之间的路径。算法的主要部分在循环中处理，但是，在进入循环之前，我们需要先初始化一些数据：

```
currentNode = startNode
add currentNode to closedSet
```

当前节点只是跟踪哪个邻居节点是下一个估值的节点。在算法开始的时候，我们除了开始节点没有任何节点，所以需要先对开始节点的邻居进行估值。

在主循环里，我们首先要做的事情就是查看所有与当前节点相邻的节点，而且把一部分加到开放集合里：

```
do
   foreach Node n adjacent to currentNode
```

```
      if closedSet contains n
         continue
      else
         n.parent = currentNode
         if openSet does not contain n
            compute n.h
            add n to openSet
         end
      end
   loop // 结束循环
```

注意任意已经在封闭集合里的节点都会被忽略。在封闭集合里的节点都在之前进行了估值，所以不需要再进一步估值了。对于其他相邻节点，这个算法会把 parent 设置为当前节点。然后，如果节点不在开放集合中，我们计算 $h(x)$ 的值并且把节点加入开放集合。

在邻近节点处理完之后，我们再看看开放集合。如果开放集合中再也没有节点存在，意味着我们把所有节点都估算过了，这就会导致寻路失败。实际上也不能保证总有路径可走，所以算法必须考虑这种情况：

```
if openSet is empty
   break // 退出主循环
end
```

但是，如果开放集合中还有节点，我们就可以继续。接下来要做的事情就是在开放集合中找到最低 $h(x)$ 值开销节点，然后移到封闭集合中。在新一轮迭代中，我们依旧将其设为当前节点。

```
currentNode = Node with lowest h in openSet
remove currentNode from openSet
add currentNode to closedSet
```

这也是为什么有一种合适的容器能让开放集合变得简单。比起做 $O(n)$ 复杂度的搜索，二叉堆能够以 $O(1)$ 时间找到最低 $h(x)$ 值节点。

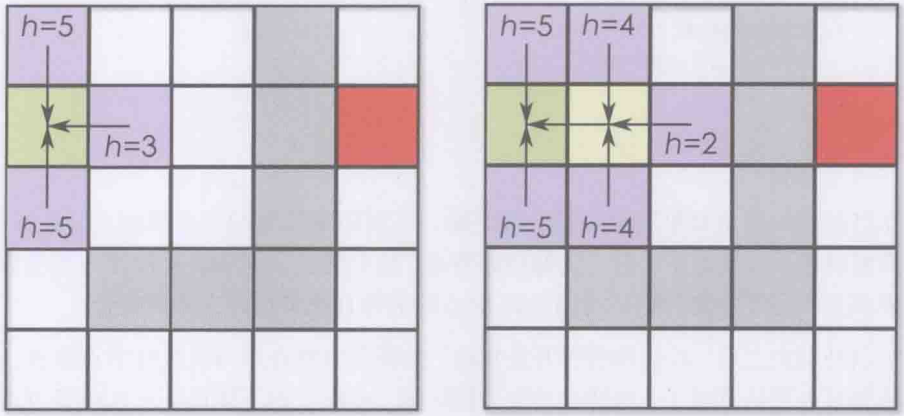
最后，我们要有循环退出的情况。在找到有效路径之后，当前节点等于终点，这样就能够退出循环了。

```
until currentNode == endNode //end main do...until loop
```

如果在成功的情况下退出 do...until 循环，我们会得到一条链表通过 parent 从终点指向起点。由于我们想要得到从起点到终点的路径，所以必须将其反转。有很多种方法反转链表，最简单的方法就是使用栈。

图 9.7 显示了贪婪最佳优先算法作用在示例数据集的开始两次迭代。在图 9.7(a) 中，起点加入封闭集合，而邻接节点则加入开放集合。每个邻接节点（蓝色）都有用曼哈顿距离算出来

的自己到达终点的 $h(x)$ 开销。箭头表示子节点指向父节点。这个算法的下一步就是选择最低 $h(x)$ 值节点，在这里会选择 $h = 3$ 的节点。然后这个节点就会作为当前节点，放到封闭集合里。图 9.7(b) 显示了下一步的迭代，将当前节点（黄色）的邻接节点放入开放集合中。



(a)：第一次迭代　　(b)：第二次迭代

图 9.7　贪婪最佳优先快照

在目标节点（红色）加到封闭集合之后，我们会得到从终点到起点的链表。这个链表可以通过反转得到之前图 9.6 展示的路径。

完整的贪婪最佳优先算法如清单 9.1 所示。注意这个实现假设 $h(x)$ 的值在执行过程中总是不变的。

清单 9.1　贪婪最佳优先算法

```
curentNode = startNode
add currentNode to closedSet
do
   // 把邻接节点加入开放集合
   foreach Node n adjacent to currentNode
      if closedSet contains n
         continue
      else
         n.parent = currentNode
         if openSet does not contain n
            compute n.h
            add n to openSet
         end
      end
   loop
```

```
    // 所有可能性都尝试过了
    if openSet is empty
        break
    end
    // 选择新的当前节点
    currentNode = Node with lowest h in openSet
    remove currentNode from openSet
    add currentNode to closedSet
until currentNode == endNode

// 如果路径解出，通过栈重新构造路径
if currentNode == endNode
    Stack path
    Node n = endNode
    while n is not null
        push n onto path
        n = n.parent
    loop
else
    // 寻路失败
end
```

如果我们不想用栈构造路径，另一个方案就是直接计算起点到终点的路径。这样，在寻路结束的时候就能得到从起点到终点的路径，可以节省一点计算开销。这种优化技巧在第 14 章的塔防游戏中有所介绍（虽然它没有使用贪婪最佳优先算法来寻路）。

A* 寻路

在讲了贪婪最佳优先算法之后，我们就可以考虑怎么提升路径的质量。比起单一地依赖于 $h(x)$ 作为寻路的估价，**A* 算法**（读作 A 星）增加了**路径开销**分量。路径开销就是从起点到当前节点的实际开销，通过 $g(x)$ 计算。A* 中访问一个节点的开销等式为：

$$f(x) = g(x) + h(x)$$

为了能够使用 A* 算法，Node 结构体需要增加 $f(x)$ 和 $g(x)$ 的值，如下：

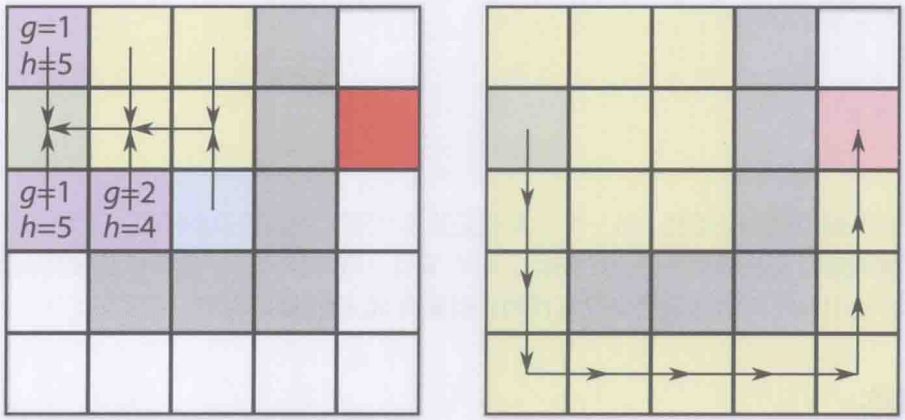
```
struct Node
    Node parent
    float f
    float g
    float h
end
```

当一个节点加入开放集合之后，我们需要计算所有的 3 个分量，而不仅仅是启发式。而且，开放集合会根据 $f(x)$ 的值来排序，因为在 A* 中每次迭代都会选择 $f(x)$ 值最低的节点。

对于 A* 算法只有一个主要变化，那就是**节点选用**的概念。在最佳优先算法中，总是把邻接节点作为父节点。但是在 A* 算法中，已经放在开放集合中的邻接节点需要估值之后才能决定哪个当前节点是父节点。

这是因为 $g(x)$ 的开销取决于父节点 $g(x)$ 的开销。这意味着如果父节点是可选的，那么 $g(x)$ 的开销是可变的。所以在 A* 算法中我们不想自动更改节点的父节点。只有在当前节点是更好的父节点时才改动。

在图 9.8(a) 中，我们可以看到当前节点正在检查邻近节点。这个节点到左边节点的 $g = 2$，如果那个点以当前节点为父节点，$g = 4$，结果会更糟。所以在这种情况下，当前节点的路径应该拒绝选用。图 9.8(b) 显示了 A* 的最终计算路径，显然比贪婪最佳优先算法效果要好。



(a)：当前节点选用失败　　(b)：最终A*路径

图 9.8　A* 路径

除了节点选用，A* 算法的代码与贪婪最佳优先算法非常类似，如清单 9.2 所示。

清单 9.2　A* 算法

```
currentNode = startNode
add currentNode to closedSet
do
   foreach Node n adjacent to currentNode
      if closedSet contains n
         continue
      else if openSet contains n // 选用检查
         compute new_g // n节点以当前节点为父节点的g(x)值
         if new_g < n.g
```

```
            n.parent = currentNode
            n.g = new_g
            n.f = n.g + n.h // 该节点的n.h是不变的
         end
      else
         n.parent = currentNode
         compute n.h
         compute n.g
         n.f = n.g + n.h
         add n to openSet
      end
   loop

   if openSet is empty
      break
   end

   currentNode = Node with lowest f in openSet
   remove currentNode from openSet
   add currentNode to closedSet
until currentNode == endNode
// 清单9.1的路径构造
...
```

第 14 章的塔防游戏有基于六边形网格的 A* 算法的完整实现。

Dijkstra 算法

最后一个寻路算法可以通过稍微修改 A* 算法得到。在 Dijkstra 算法中，没有启发式的估算——或者换个角度：

$$f(x) = g(x) + h(x)$$

$$h(x) = 0$$

$$\therefore f(x) = g(x)$$

这意味着 Dijkstra 算法可以使用与 A* 算法一样的代码，除了启发式为 0 之外。如果将 Dijkstra 算法用于我们的例子中，能够得到与 A* 算法一样的路径。如果只有 A* 才使用启发式，Dijkstra 总能得到与 A* 算法同样的路径。但是，Dijkstra 算法通常会访问更多的节点，所以 Dijkstra 效率更低。

唯一使用 Dijkstra 替代 A* 的场景就是场景中同时存在多个有效目标节点的时候，但是你不会知道哪个更近。但在那种场景下，大多数游戏都不会使用 Dijkstra。这个算法被讨论基本

上都是出于历史原因，因为 Dijkstra 比 A* 早 10 年提出。A* 的创新在于结合了贪婪最佳优先和 Dijkstra 算法。所以虽然本书通过 A* 讨论 Dijkstra，但这不是它被开发出来的原因。

基于状态的行为

大多数基础的 AI 行为无非就是不同的状态。以《乒乓》的 AI 举例，它只需要跟踪球的位置。这个行为在整个游戏的过程中都没有改变，所以这样的 AI 可以被认为是**无状态**的。但是当游戏有点复杂度的时候，AI 就需要在不同的时候有不同的行为。大多数现代游戏的 NPC 在不同的位置都有不同的行为。本节将讨论如何设计带状态的 AI，然后讨论它们是如何实现的。

AI 的状态机

有限状态机可以完美地表达基于状态的 AI。它有着一组可能的状态，由一定条件控制状态转换，而在状态切入切出的时候可以执行动作。

当你为 AI 实现状态机时，应该先设计好有着什么样的行为，这些行为是如何连接的。假设我们为潜行游戏实现警卫的基础 AI。默认情况下，我们会让警卫巡逻预定好的路径。如果警卫在巡逻的时候发现玩家，他应该攻击玩家。最后，如果警卫被杀死了，他应该死亡。所以 AI 描述起来应该有 3 种状态：巡逻、攻击、死亡。

紧接着，我们需要决定哪些状态机之间可以切换。进入死亡状态的条件很明显，就是警卫被杀死，就会进入死亡状态。对于攻击状态，只有玩家被警卫发现才会进入。组合 3 种情形，我们可以为状态机绘制一张图，如图 9.9 所示。

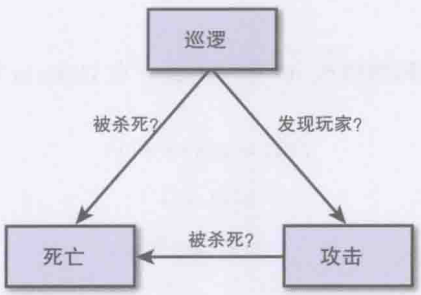


图 9.9 基本的潜行游戏 AI 状态机

虽然这个 AI 能用，但是比起大多数潜行游戏的 AI 来说就弱多了。假设警卫巡逻中听到可疑的声音。以我们目前的状态机，AI 会不闻不问。而理想中的 AI，应该进入“调查”状态，会在玩家附近搜索。更进一步，如果警卫发现玩家，目前它总是攻击。也许我们想要“警报”状态，比起直接进入，需要随机进入“警报”或者攻击状态。

如果增加这些元素，我们的状态机就会更加复杂，如图 9.10 所示。

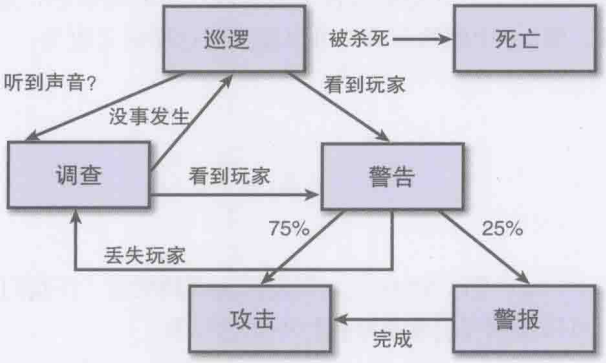


图 9.10　更复杂的潜行游戏 AI 状态机

注意，死亡状态与其他状态是隔离开的。通过简单地去除箭头就能得到图 9.10，这是因为任何状态通过下，只要 AI 被杀死就会进入死亡状态。现在，只要我们希望，就可以加入更多的复杂状态到我们的状态机中，但是各个状态的 AI 行为设计都不太一样。

《吃豆人》中的 AI

初看，街机经典游戏《吃豆人》的 AI 非常初级。鬼魂要么跟在玩家身后追逐要么躲避玩家，所以你可能会认为这是一个二态状态机。但是，实际上鬼魂的 AI 还要复杂一些。

Toru Iwatani 是《吃豆人》的策划，跟 Susan Lammers 谈过这个话题，谈话内容收录于 Susan 1986 年出版的书 *Programmers at Work* 上。他说："希望每个鬼魂敌人都是一个特别的角色，有着独自的移动，所以他们不只是追着吃豆人……因为这样会很无聊"。每 4 个鬼魂有 4 种不同的行为来找到相对于吃豆人或者迷宫的不同位置。而且鬼魂还会在追逐和逃跑的状态中切换。随着玩家玩到后面的关卡，鬼魂追逐的比例会越来越高。

一个很好的关于《吃豆人》鬼魂 AI 实现的文章可以在这个游戏相关的博客上（*http://tinyurl.com/238l7km*）找到。

基础的状态机实现

状态机有多种实现方式。最简单的需求就是当 AI 更新的时候，正确的更新行为必须根据当前状态来完成。理想情况下，我们还想让状态机有进入和退出行为。

如果AI只有两种状态，我们可以在AI的Update函数中用一个布尔值来判断。但是这个方案不够健壮。一个稍微灵活的方式是通过枚举值来表示不同的状态，这经常在简单的游戏中可以看到。比如说，图9.9中的状态机就可以像下面这样定义枚举：

```
enum AIState
   Patrol,
   Death,
   Attack
end
```

然后可以用AIController类以AIState类型作为成员变量。在我们的AIController的Update函数中，可以根据当前状态来执行不同的行为：

```
function AIController.Update(float deltaTime)
   if state == Patrol
      // 执行巡逻行为
   else if state == Death
      // 执行死亡行为
   else if state == Attack
      // 执行攻击行为
   end
end
```

状态的变化和进入/退出行为可以在第二个函数中实现：

```
function AIController.SetState(AIState newState)
   // 退出行为
   if state == Patrol
      // 退出巡逻行为
   else if state == Death
      // 退出死亡行为
   else if state == Attack
      // 退出攻击行为
   end

   state = newState

   // 进入行为
   if state == Patrol
      // 进入巡逻行为
   else if state == Death
      // 进入死亡行为
   else if state == Attack
      // 进入攻击行为
   end
end
```

这个实现有几个问题。首先很明显的一点就是，随着状态机的增加，Update 和 SetState 的可读性会减弱。如果我们的例子中有 20 个状态而不是 3 个，代码看上去就像意大利面条。

第二个主要问题是缺乏灵活性。假如我们有两个 AI，它们有不同的状态机。这样我们就需要为不同的 AI 实现不同的枚举和控制器。现在，假设两个 AI 之间会公用一些状态，比如说巡逻状态。以我们目前的基础代码结构是无法在 AI 之间共享状态的。

一个方法是将巡逻的代码复制到两个类中，但是有着两份同样的重复代码是非常不好的实践。另一个方法就是写一个共同的基类，然后把共有的行为“冒泡”上去。但是这样，还是有很多缺点：意味着任何需要巡逻行为的 AI 都要从这里继承。

所以虽然这个基础的实现能工作，但是除非 AI 状态机非常简单，否则完全不推荐。

状态机设计模式

有一个通用解法的**设计模式**能解决类似的问题。这本在设计模式方面享有盛誉的名著通常被称为“四人帮”《设计模式》，这本书会在本章的引用资料中提及。其中一个设计模式就是状态机模式，允许“一个对象通过改变内在状态来切换行为”。

这可以通过类组合的方式完成。所以 AIController“有一个” AIState 作为成员变量。每个特定的状态都是 AIState 的子类。图 9.11 演示了类图。

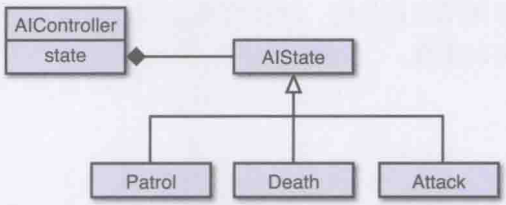


图 9.11　状态机设计模式

基类 AIState 的定义如下：

```
class AIState
   AIController parent
   function Update(float deltaTime)
   function Enter()
   function Exit()
end
```

父引用使得任何 AIState 的实例都可以让 AIController 拥有它。这是必要的，如果我们想要切换到新的状态，需要有一些方法通知 AIController 这些事情。每个 AIState 都有自己的 Update、Enter、Exit 函数，可以为某个特定有需求的状态所实现。

AIController类会保留一个当前AIState的引用，而且需要Update和SetState函数：

```
class AIController
   AIState state
   function Update(float deltaTime)
   function SetState(AIState newState)
end
```

这样，AIController的Update函数只要简单地调用AIState的Update函数即可：

```
function AIController.Update(float deltaTime)
   state.Update(deltaTime)
end
```

通过设计模式，SetState函数也变得清晰多了：

```
function AIController.SetState(AIState newState)
   state.Exit()
   state = newState
   state.Enter()
end
```

通过状态设计模式，所有状态相关的行为都移到AIState的子类当中去了。这使得AIController的代码比之前清晰多了。状态机设计模式也使得系统更加模块化。比如说，如果我们想要在多个状态机中使用巡逻代码，就会容易很多。而且即使我们想让巡逻稍稍不同，只要从巡逻继承出去就可以修改。

策略和计划

很多游戏都需要比基于状态的敌人更复杂的AI。比如即时战略游戏，AI期望看上去与人类玩家相差无几。这个AI需要有一个大局观，知道自己要做什么，然后尽力去做。这就是策略和计划的工作方式。在本节中，我们会通过RTS类型的游戏来了解策略和计划，这个技术在其他类别的游戏中也同样可用。

策略

策略就是从AI的视角来完成游戏。比如说，它要思考的是应该更具侵略性还是防守性。**微观**策略由单位行为组成。这通常可以用状态机来完成，所以就不必再深入讨论了。相对而言，**宏观**策略复杂得多。它是AI的大局观，而且会决定如何完成游戏。当为像《星际争霸》那样的游戏开发AI的时候，开发者通常根据顶级玩家的思维进行建模。一个宏观策略在RTS游戏中可能会是“突击”（尝试尽快攻击玩家）。

策略有时候看上去就像很模糊的使命描述，而且模糊的策略是很难开发的。为了让问题更加形象，策略通常被认为是一系列的特定目标。比如说，如果策略是“科技”（增加装备科技），一个特定的目标可能就是去“扩张”（建立第二个基地）。

> **注意**
>
> RTS 游戏 AI 如果实现得好能够让人印象深刻。年度人工智能互动娱乐大会（AIIDE）有一个《星际争霸》比赛。来自不同大学的参赛队伍开发出来的 AI 要参加数百场比赛。比赛结果非常有趣，AI 看上去就如同熟练的玩家一样。更多相关信息和视频可以在 www.starcraftaicompetition.com 看到。

一个策略通常都不止一个目标，也就是说，我们需要有一个优先级系统来让 AI 选择哪个目标更加重要。所有其他目标如果优先级不是最高，那么会先搁在后面不管。其他目标会在最重要的目标完成时重新参与选择。一个实现目标系统的方式就是像这样写一个 AIGoal 类：

```
class AIGoal
   function CalculatePriority()
   function ConstructPlan()
end
```

每个特定目标都会作为 AIGoal 的子类实现。所以当策略进行目标选择之后，所有策略的目标会放到一个根据优先级排序的容器里。注意，真正高级的策略系统应该支持同时选用多个目标的功能。因为如果两个目标不是互斥的，那么是没有理由不同时选择两个目标的。

计算优先级的启发式函数是 CalculatePriority，可能会相当复杂，而且根据游戏规则的不同而变化。比如说，一个目标是“建立空中单位”，可能会在发现敌人正在建造能够消灭空军的单位时降低优先级。另一方面，如果 AI 发现敌人没有能够伤害空军的单位，那么就会增加这一目标的优先级。

AIGoal 中的 ConstructPlan 函数就是用于构造**计划**的：一系列为了达到目标而计划出来的步骤。

计划

每个目标都需要一个相应的计划。比如说，如果目标是扩张，那么计划可能如下：

1. 为扩张侦查合适的地点。
2. 建立足够多的单位来保护扩张。
3. 派遣工人和战斗单位去扩张点。
4. 开始建造扩张点。

特定目标的计划可以用状态机来实现。计划中的每一步都可以是状态机中的一个状态，而状态机持续为该步骤行动直到达到条件。但是，实践中的计划很少是线性的。根据计划某个步骤的成功或者失败，AI 可能会调整步骤的顺序。

一个需要考虑的事情是计划需要定期查看目标的可行性。如果扩张计划中发现没有适合扩张的位置，那么目标就是不可行的。一旦目标被标记为不可行，大局观需要重新估算。最终，必须要有一个“指挥官”来决定策略的改变。

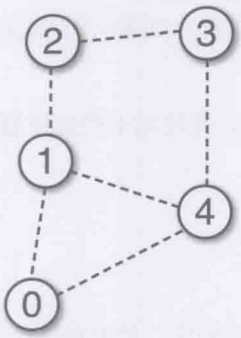
策略和计划的领域可以非常复杂，特别对于一个完整功能的 RTS 游戏来说，比如《星际争霸》。相关话题的更多信息，可以看看本章最后的相关资料。

总结

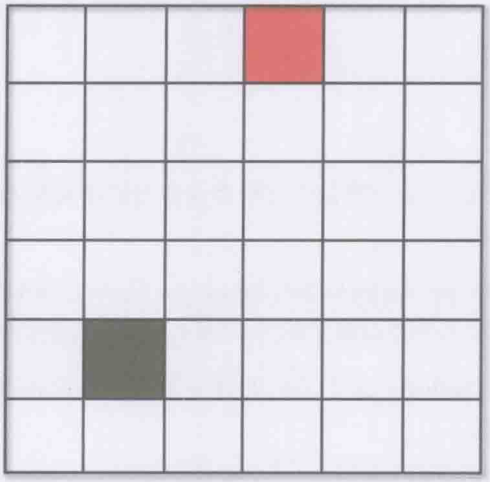
AI 游戏程序员的目标就是让系统看上去比较聪明。这可能不是研究人员眼中的“真”AI，但是游戏通常都不需要这么复杂的 AI。与我们讨论的一样，游戏 AI 领域的一个大问题就是找到从点 A 到点 B 的路径。游戏中最常用的寻路算法就是 A* 算法，而且它可以用于任何搜索空间表达为图（比如格子、路点、导航网格）的问题。大多数游戏都有某种方式实现的通过状态机控制的行为，而实现状态机的最佳方式就是状态机设计模式。最后，策略和计划可能会在 RTS 游戏中创造更加可信真实的行为。

习题

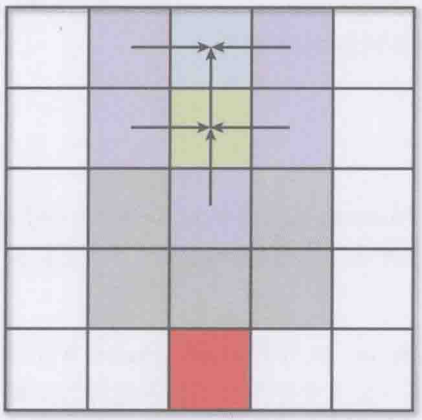
1. 给定下图，每个节点的相邻节点是什么？



2. 为什么导航网格比路点优越？
3. 什么情况下启发式是可取的？
4. 计算下图中两点的曼哈顿距离和欧几里得距离：



5. 把 A* 算法应用于下面的例子中。当前节点的相邻节点已经被加入，所以算法需要选择新的当前节点。使用曼哈顿距离为开放集合中的每个节点计算 $f(x)$ 开销，而且约定不允许对角线行走。哪个节点会被选中为当前节点？



6. 如果 A* 算法将启发式改为 $h(x) = 0$，哪个算法与此类似？
7. 比较 A* 和 Dijkstra 算法。是否其中一个比另一个优越？为什么？
8. 你正在为一头狼设计 AI。构造一个简单的行为状态机，最少要有 5 个状态。
9. 状态机设计模式的优点是什么？
10. 什么是策略，它是如何细分成小组成部分的？

相关资料

通用 AI

Game/AI（*http://ai-blog.net/*）：这个博客中有很多业界经验丰富的程序员谈论了关于 AI 编程相关的话题。

Millington, Ian and John Funge. *Artificial Intelligence for Games* (2nd Edition). Burlington: Morgan Kaufmann, 2009：这本书主要以算法的方式讲了很多常见的游戏 AI 问题。

***Game Programming Gems* (Series)**：这个系列的每一卷都有不少 AI 编程相关的文章。老版本已经绝版了，但新的还在。

***AI Programming Wisdom* (Series)**：与 *Game Programming Gems* 系列类似，但是完全专注于游戏 AI，其中一些已经绝版了。

寻路

Recast and Detour（*http://code.google.com/p/recastnavigation/*）：这是一个优秀的开源寻路库，由 Mikko Mononen 开发，他开发过多款游戏，比如《孤岛危机》。Recast 能够自动生成导航网格，而 Detour 实现了在这些网格上的寻路。

状态

Gamma, Eric et. al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley, 1995. 这本书讲设计模式的同时描述了状态机设计模式，对于所有程序员都很有用。

Buckland, Mat. *Programming Game AI By Example*. Plano: Wordware Publishing, 2005. 这是一本通用的游戏 AI 书籍，它有一个非常好的基于状态的行为实现。

第10章
用户界面

一个游戏的用户界面通常有两个主要部分：一个是菜单系统，一个是平视显示器（HUD）。菜单系统决定玩家如何进入和退出游戏，包括改变模式、暂停游戏、选择选项等。一些游戏（特别是RPG）都会有菜单系统来展示和升级技能。

HUD包括了展示玩家游戏中额外信息的元素，通常包括雷达、子弹数量、指南针和准心。虽然不是所有游戏都有HUD（而一些游戏则有隐藏HUD的选项），但是大多数都会有至少一个。

菜单系统

一个实现良好的菜单系统应该能够提供很多不同方面的灵活性，例如不会有元素个数的限制、窗口个数的限制以及容易添加新的子菜单等。与此同时，它又必须集成尽可能多的功能。本节会讨论一个健壮的菜单系统应该要考虑哪些因素。这里讨论到的很多技术都会在第 14 章的塔防游戏中用到。

菜单栈

典型家用机游戏的菜单系统都以“点击开始”作为开始界面。在用户按键之后，就会进入主菜单界面。也许你还可以通过点击选项进入选项界面，也可以点击开发组或者新手教程。通常来讲，玩家都能够退出当前菜单然后返回之前的界面。这种经典的菜单界面如图 10.1所示。

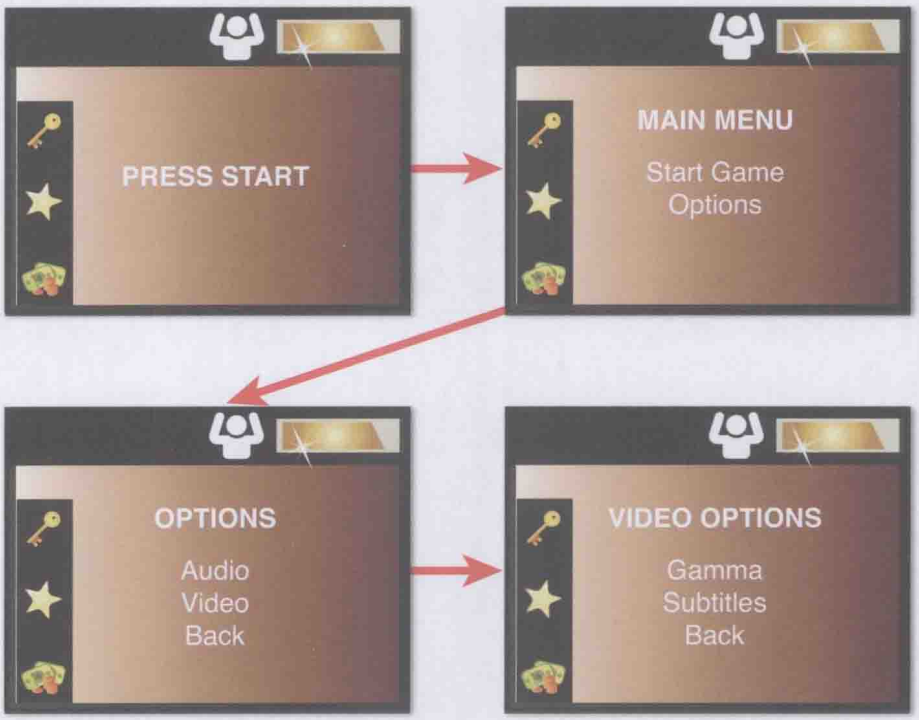


图 10.1 家用机游戏菜单示例

一个确保菜单总能回退到基本界面的方法就是使用栈这种数据结构。栈最上层的元素就是当前活跃的菜单，而打开新菜单就是往栈中压入新的菜单。回退到之前的菜单就是将当前的菜单弹出栈。这个机制还可以改进为支持多个菜单同时可见，比如说，如果需要接受/拒

绝某个请求，一个弹出框可以在某个菜单之前。为了达到目标，菜单系统需要对栈的底部到顶部全部引用。

为了存储栈中的所有菜单，你可能需要所有菜单都继承自同一个基类。这个基类可能存储了比如菜单标题以及按钮列表（或者菜单拥有的子元素）等信息。我使用了类似的方式实现了 *Tony Hawk's Project 8* 的菜单系统，同时也应用于第 14 章塔防游戏的所有 UI 系统。这个系统的一个方面就是它并不以常规的方式来操作栈。这种非常规操作在从主菜单切入到游戏中的时候会用到，因为你通常不想在游戏中还能看到主菜单。

按钮

几乎所有菜单系统都有玩家可以交互的按钮。对于 PC 或者家用机游戏，都需要两种按钮状态：未选中和选中。通过这种方式，当前选中的按钮对于用户来讲就非常清楚。表达按钮选中最简单的方式就是改变颜色，有时候也可以让纹理发生变化或者按钮的大小发生改变。一些 PC/家用机菜单中会使用第三种状态来表示按钮正在被点击，但是第三种状态不是必需的。而在触屏设备上通常采用一个默认状态（未选中）和按下状态。这样，当玩家点击按钮，它就在视觉上发生了变化。

如果菜单只能通过键盘或者手柄控制，按钮的支持就相当直观。一个菜单界面可以用双向链表的形式来组织按钮，而当用户点击按键（比如上、下）时，系统会自动让选中的按钮变为未选中，而选中下一个按钮。由于按钮以双向链表的方式组织，很容易前进和后退，而且如果过了最后一个按钮也很容易回到第一个按钮。

通常在按钮按下的时候，用户会期待发生一些事情。一个足够抽象的支持方式就是让按钮拥有一个函数成员变量。这通常由编程语言决定，但是它可以是一个动作（比如 C#）、一个函数指针（C），还可以是 lambda 表达式（C++）。这样，当创建新按钮的时候，你可以为它分配正确的函数，它能够在按钮按下的时候得到执行。

如果游戏也支持通过鼠标导航，系统就会增加一定的复杂度。每个按钮需要有一个**感应区**，或者用一个 2D 包围盒表示按钮可点击区域。所以随着用户鼠标在菜单中移动，需要检测鼠标是否进入某个按钮的感应区。这个方法在第 14 章的游戏中有所应用，如图 10.2 所示。

允许玩家在鼠标导航和键盘导航之间切换也是可以做到的。一个常见的方式就是当用户按下键盘的时候隐藏鼠标（同时也忽略鼠标位置）。然后，如果鼠标移动，鼠标的选择模式被再次激活。一些支持键盘、鼠标、手柄的游戏就使用了类似的系统。在这种情况下，菜单导航会用同样的规则在键盘、鼠标、手柄之间激活切换。

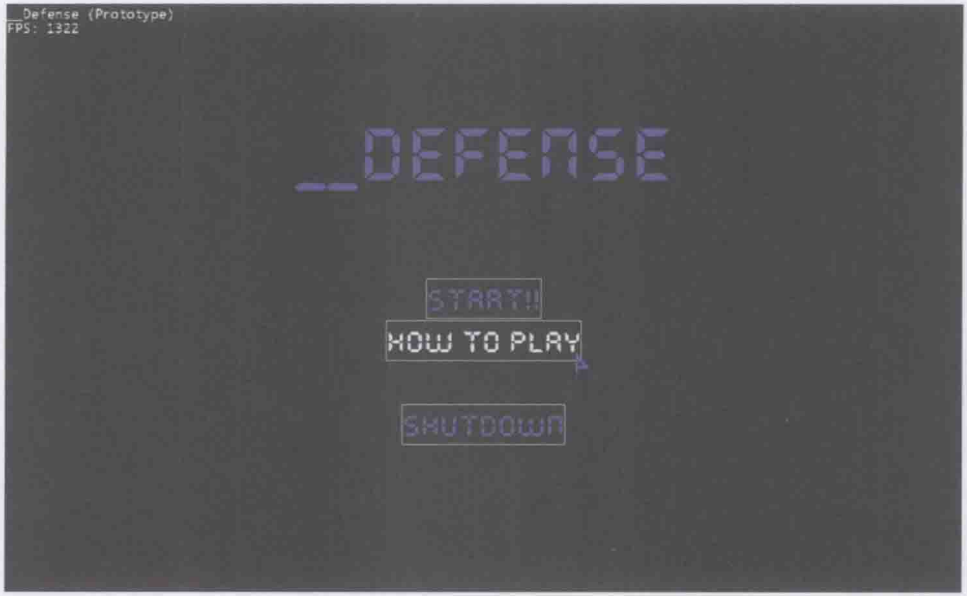


图 10.2 第 14 章的游戏中通过 debug 模式查看菜单按钮的感应区

打字

计算机游戏中最常见的菜单功能就是让用户在其中输入一到两个单词。常见于排行榜系统或者存储文件的文件名等。传统的应用程序支持打字的方法通常是通过标准输入，但是在第 5 章就说过，标准输入不适合游戏。

允许输入的第一步就是准备一个空字符串。每次玩家输入一个字母，我们就可以将字母拼接到字符串后面。但是在这么做之前，要先认识这些字母。回忆一下第 5 章我们也讨论过虚拟键盘（键盘的每个键怎么与枚举值对应起来）。比如说，K_A 就对应键盘上的 A 键。相应地，在打字系统中，字母的虚拟键也会在枚举值中顺序出现。就是说，K_B 在 K_A 之后，K_C 的索引在 K_B 之后，以此类推。而在 ASCII 码中，字符 B 在字符 A 之后，情况也一样。由于这种关系，我们很容易就可以将按键与字母对应起来：

```
function KeyCodeToChar(int keyCode)
   // 确保这是字母键
   if keyCode >= K_A && keyCode <= K_Z
      // 现在，假设大写的情况
      // 这个映射取决于语言
      return ('A' + (char)(keyCode - K_A))
   else if keyCode == K_SPACE
      return ' '
   else
```

```
        return ''
    end
end
```

让我们用几个例子测试一下我们的代码。如果 keyCode 是 K_A，相减的结果为 0，意味着字母会返回“A”。如果用 K_C 作为 keyCode，相减结果为 2，函数返回 A+2，或字母“C”。这些例子都证明了函数能够得到我们期望的结果。

在实现了转换函数之后，剩下的工作就是确认 keyCode 在 K_A 到 K_Z 之间而且是“刚刚按下”。如果发生了，就将 keyCode 转换为字符，然后拼接到我们的字符串中。如果需要，可以进一步扩展 KeyCodeToChar 函数同时支持大小写，可以根据 Shift 键是否按下进行判断。

HUD 元素

最基础的 HUD（平视显示器）元素就是玩家得分和剩余生命值。这种 HUD 实现起来相对琐碎——在主要游戏场景渲染之后，我们只要在顶层绘制文字或者图标以展示相应的信息即可。但是很多游戏都使用了更加复杂的元素，包括路点箭头、雷达、指南和准心。本节将会看看这些元素该如何实现。

路点箭头

路点箭头用于引导玩家到下一个目标点。一个最简单的方法就是让箭头成为真正的 3D 对象，然后放置在屏幕相应的位置。然后随着玩家在游戏世界中移动，3D 箭头会旋转到正确的指向以指引玩家移动。这种类型的路点箭头在《疯狂出租车》中非常常见，如图 10.3 所示。

实现这种路点箭头的第一步就是建立一个在没有任何旋转作用在上面的时候朝前指向的箭头，然后在游戏过程中有 3 个参数需要跟踪：箭头朝向的向量、箭头在屏幕空间中的位置，还有箭头跟踪的路点。

朝向向量应该先初始化轴向使得指向屏幕内。在传统的左手坐标系中，就是 $+z$ 轴。箭头在屏幕空间中所在的位置就是我们所希望箭头在屏幕中的相应位置。由于是屏幕空间的位置，所以它会是一个 (x, y) 坐标，而我们需要将它转换为 3D 世界中的坐标系，这样才能够在正确的位置渲染。为了完成这个目标，我们会用到反投影，与第 8 章的拣选功能类似。最后，路点位置就是箭头朝向所需要的指向位置。

为了更新路点箭头，每一帧都需要构造一个从玩家到路点的向量。然后进行正规化，就可得到箭头所需要的朝向。接下来的问题就是使用点乘和叉乘来确定原始朝向到新朝向所需要的旋转角度。这个旋转可以使用四元数，使用插值得到平滑的箭头过渡。图 10.4 显示了更新路点箭头所必需的计算。

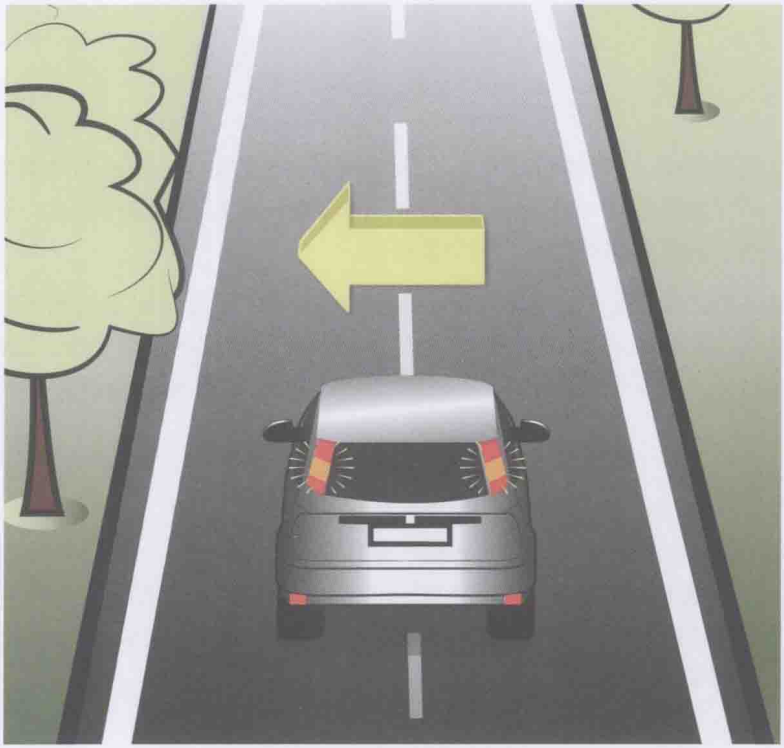
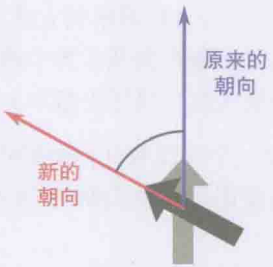


图 10.3 一款通过路点箭头指示玩家向左转的驾驶游戏



$$\theta = \text{acos}(\text{old} \cdot \text{new})$$

$$\text{axis} = \text{old} \times \text{new}$$

图 10.4 路点箭头计算，假设新朝向和旧朝向都经过正规化

这种类型箭头的实现如清单 10.1 所示。这种实现有几件事需要记住。首先就是摄像机在路点箭头之间更新，因为这是保证路点箭头在屏幕上同一位置的唯一方法。否则，当摄像机发生改变时，路点箭头会延后更新一帧。

清单 10.1 路点箭头

```
class WaypointArrow
   // 记录箭头的朝向
   Vector3 facing
   // 箭头在屏幕上的2D位置
   Vector2 screenPosition
   // 箭头指向的当前路点
   Vector3 waypoint
   // 用于渲染箭头的世界变换矩阵
   Matrix worldTransform

   // 通过给定位置/旋转计算世界变换矩阵
   function ComputeMatrix(Vector3 worldPosition, Quaternion rotation)
      // 缩放、旋转、平移（但是这次我们没有缩放）
      worldTransform = CreateFromQuaternion(rotation) *
                       CreateTranslation(worldPosition)
   end

   // 根据屏幕位置计算3D箭头的世界坐标
   function ComputeWorldPosition()
      // 为了计算反投影，我们需要一个3D向量
      // z分量是一个在近平面和远平面之间的百分比
      // 在这种情况下，我选择二者之间10%的一个点（z=0.1）
      Vector3 unprojectPos = Vector3(screenPosition.x,
                                     screenPosition.y, 0.1)

      // 得到摄像机和投影矩阵
      ...

      // 调用第8章的反投影函数
      return Unproject(unprojectPos, cameraMatrix, projectionMatrix)
   end

   function Initialize(Vector2 myScreenPos, Vector3 myWaypoint)
      screenPosition = myScreenPos
      // 对于Y轴向上的左手坐标系
      facing = Vector3(0, 0, 1)
      SetNewWaypoint(myWaypoint)

      // 初始化世界变换坐标系
      ComputeMatrix(ComputeWorldPosition(), Quaternion.Identity)
   end

   function SetNewWaypoint(Vector3 myWaypoint)
      waypoint = myWaypoint
```

```
    end

    function Update(float deltaTime)
        // 得到箭头的当前世界坐标
        Vector3 worldPos = ComputeWorldPosition()

        // 得到玩家位置
        ...
        // 箭头的新朝向是一个正规化向量
        // 从玩家的位置指向路点
        facing = waypoint - playerPosition
        facing.Normalize()

        // 使用点乘得到原始朝向(0,0,1)和新朝向之间的夹角
        float angle = acos(DotProduct(Vector3(0, 0, 1), facing))
        // 使用叉乘得到轴的旋转轴
        Vector3 axis = CrossProduct(Vector3(0, 0, 1), facing)
        Quaternion quat
        // 如果长度为0，意味着平行
        // 意味着不需要旋转
        if axis.Length() < 0.01f
            quat = Quaternion.Identity
        else
            // 计算用来表示旋转的四元数
            axis.Normalize()
            quat = CreateFromAxisAngle(axis, angle)
        end

        // 现在设置箭头最后的世界变换
        ComputeMatrix(worldPos, quat)
    end
end
```

还有，路点箭头渲染时需要在其他 3D 物体渲染之后关闭 z-buffer 才能进行渲染。这样能够确保箭头总能被渲染出来，哪怕有其他 3D 物体在它前面。

准心

大多数第一人称或者第三人称游戏带有的远程攻击都会使用准心这种标准的 HUD，它使得玩家知道自己距离打中目标还差多远（不管目标是敌是友）。不管准心是传统的十字形还是圆形，实现原理都一样。事实上，实现起来就像第 8 章的鼠标拣选。

就像鼠标光标一样，准心是一个在屏幕上的坐标。我们拿到这个 2D 坐标，然后执行两个反投影：一个在近平面，一个在远平面。得到这两个点以后，可以在这两点之间执行光线投射。然后可以使用第 7 章中讨论过的物理计算，得到一组被光线打中的对象。在这组对象当中，我们想要第一个被光线打中的对象。在简单的游戏中，第一个被光线打中的对象就是最靠近近平面的对象。但对于很大的对象来说，这个结论不一定成立。所以对于复杂的游戏来说，可能需要对被光线打中的对象进行比较。

在我们知道哪个对象被准心瞄准之后，就可以检查它是敌是友，或是那些非目标对象。这会决定准心的颜色——大多数游戏使用绿色来标记友军，用红色标记敌军，白色标记那些非目标对象（虽然这种方法对于红绿色盲的玩家没用）。图 10.5 演示了在假想游戏中不同场景下的准心。

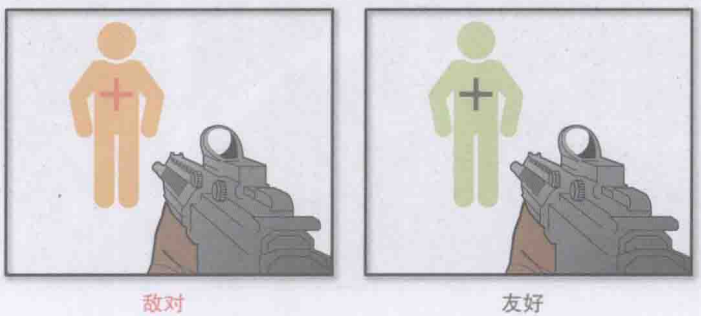


图 10.5　根据目标改变准心的颜色

雷达

有的游戏会有雷达系统用来显示雷达范围附近的敌方（或者友方）。还有几种雷达变种，一种是任何人都会在相应的雷达中显示，另外一种是只显示最近开枪的敌人。不管是哪一种，实现原理几乎都一样。图 10.6 显示了一个关于雷达的游戏截屏。

为了让雷达顺利工作，需要完成两件事情。首先，我们有一种方式可以遍历能够在雷达上显示的所有对象。然后，所有在雷达范围内的对象都需要根据 UI 中心做出相应的偏移。计算距离和转换为 2D 偏移，我们都希望忽视高度，这意味着我们必须投影雷达对象到雷达面板上。

在我们计算之前，应该先定义**雷达光点**结构体，就是那些在雷达上显示的点。这样，就可以根据实际情况让这些点有不同的大小和颜色。

```
struct RadarBlip
   // 雷达光点的颜色
   Color color = Color.Red
```

```
    // 雷达光点的位置
    Vector2 position
    // 雷达光点的缩放
    float scale = 1.0f
end
```

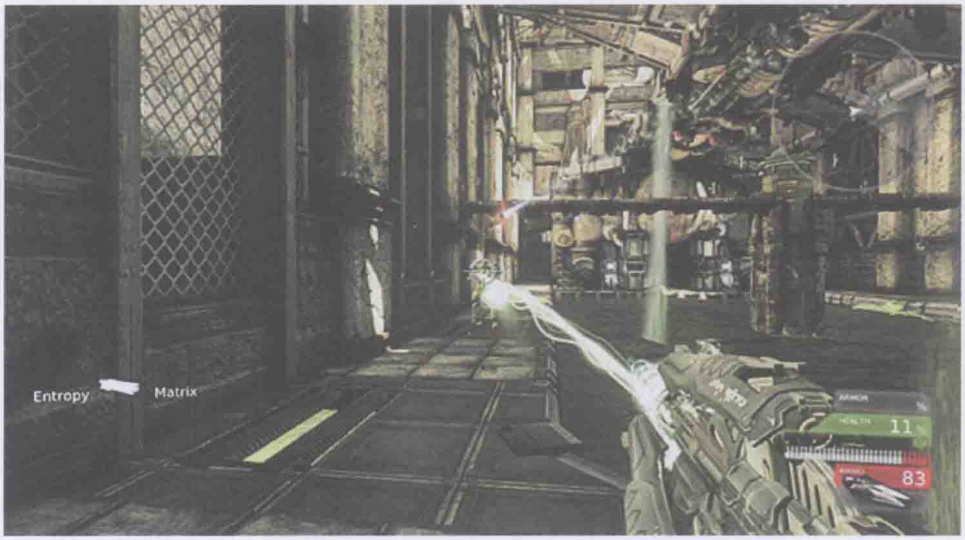


图 10.6 《虚幻竞技场 3》中的雷达（在右上角）

如果我们想为敌人创建更多不同的光标，还可以从 RaderBlip 中继承。但那更多只是显示上的效果，与雷达背后的计算没太大关系。

对于 Radar 类来说，需要有两个参数：游戏世界中的对象能够被探测出来的最大距离，以及屏幕上显示的雷达半径。通过这两个参数，在我们得到光点位置之后，就可转换到屏幕上正确的位置。

假设游戏雷达可以探测 50 个单位那么远。现在想象一下，有一个对象在玩家正前方 25 个单位远。由于对象位置是在 3D 世界中的，我们需要先将这些坐标转换为用于雷达显示的 2D 坐标。如果游戏世界中的 y 轴向上，则就是说雷达平面是 xz 平面。所以投影到这个平面上，我们只要忽略表示高度的 y 分量即可。换句话说，我们拿到的 3D 坐标 (x, y, z) 转换之后就是 (x, z)，虽然在实际代码中 2D 点的第二个分量总是 y。

在玩家和对象的坐标都转换到雷达平面的 2D 坐标系中之后，我们就可以构造从玩家到对象的向量，定义为 $\vec{a}$。虽然 $\vec{a}$ 可以判断对象是否在雷达范围内（通过计算 $\vec{a}$ 的长度），但这个向量还是有一个问题。大多数游戏中的雷达都会随着玩家旋转，雷达中的最高点（90° 单位圆）总是表示玩家的正前方。所以为了允许这样的功能，$\vec{a}$ 必须根据玩家朝向旋转。这个问题如图 10.7 所示。

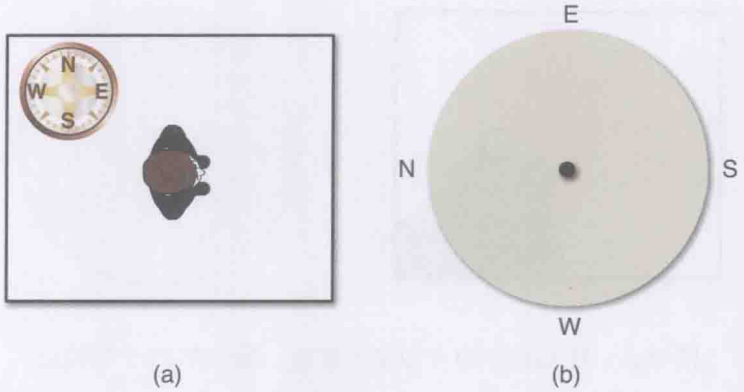


图 10.7　顶视图中玩家朝向东方 (a)，所以雷达 (b) 中的最高点也应该是东方

为了解决这个问题，我们可以取得正规化之后的玩家朝向向量，然后投影到雷达平面上。如果紧接着就对投影后的向量和雷达的向前向量进行点乘，我们就能够确定旋转的角度。然后，剩下的问题就跟第 3 章中的旋转 2D 角色一样，将两个向量通过将 y 分量置 0 转换为 3D 向量，然后使用叉乘确定顺时针还是逆时针旋转。如果两个向量叉乘结果 z 值为正，那么两个向量间的旋转为逆时针方向。

有了旋转的角度，以及旋转的方向之后，我们就可以使用一个 2D 旋转矩阵来将 $\vec{a}$ 旋转到正确的位置。由于旋转总是关于 z 轴的，所以 2D 旋转矩阵只有一种形式。假设以行向量的形式表达，2D 旋转矩阵为：

$$\text{Rotation2D}(\theta) = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}$$

在这种情况下，我们传递的 θ 要么为正要么为负，取决于旋转的方式是逆时针（正值）还是顺时针（负值），它是叉乘的结果。

回到我们的例子中，如果向量 $\vec{a}$ 就是从玩家到 25 个单位远的敌人的向量，那么在旋转之后会得到 $a = \langle 0, 25 \rangle$。一旦得到这个 2D 向量，我们可以将它的每个分量都除以雷达的最大探测距离，然后能够得到雷达以单位圆表示的雷达光点位置。所以在这种情况下，$\langle 0, 25 \rangle / 50 = \langle 0, 0.5 \rangle$。我们紧接着就可以取雷达在屏幕上的半径乘以刚刚得到的 2D 向量。所以如果屏幕上的雷达半径有 200 像素，我们可以得到 2D 向量 $\langle 0, 100 \rangle$，这个向量就表示雷达中心点的相对光点位置。图 10.8 显示了敌人 25 个单位远的例子中的雷达效果。

这样的计算可以应用在范围内的所有敌人，这样就能得到所有光点位置。完整的雷达实现如清单 10.2 所示。

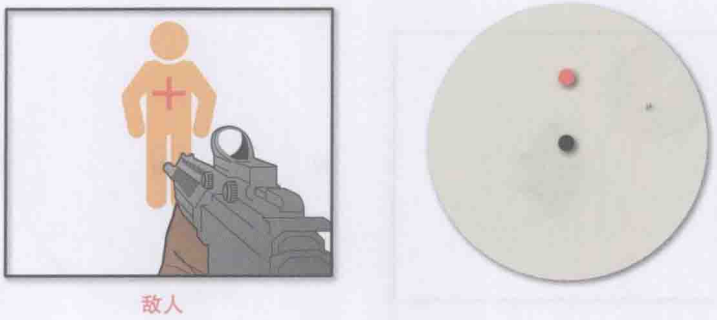


图 10.8 最大距离 50 个单位的雷达，敌人在 25 个单位远

清单 10.2 雷达系统

```
class Radar
    // 雷达在游戏世界单位中的范围
    float range
    // 雷达在屏幕中的位置(x,y)
    Vector2 position
    // 雷达在屏幕中的半径
    float radius
    // 雷达的背景图片
    ImageFile radarImage
    // 所有活跃的雷达光点
    List blips

    // 初始化函数设置range、center、radius及image
    ...

    function Update(float deltaTime)
        // 清除上一帧的光点
        blips.Clear()
        // 获取玩家位置
        ...
        // 将playerPosition转换为2D坐标
        // 以下假设y轴向上
        Vector2 playerPos2D = Vector2(playerPosition.x, playerPosition.z)

        // 计算需要添加到blip上的旋转
        // 得到正规化后的玩家朝向向量
        ...
        // 将playerFacing转换为2D
        Vector2 playerFacing2D = Vector2(playerFacing.x, playerFacing.z)
        // 计算雷达前向与玩家朝向之间的夹角
        float angle = acos(DotProduct(playerFacing2D, Vector2(0,1)))
```

```
        // 为了使用叉乘，需要转换为3D向量
        Vector3 playerFacing3D = Vector3(playerFacing2D.x,
                                         playerFacing2D.y, 0)
        // 使用叉乘判定旋转的方向
        Vector3 crossResult = CrossProduct(playerFacing3D, Vector2(0,1,0))
        // 顺时针为-z，意味着角度应该取负
        if crossResult.z < 0
            angle *= -1
        end

        // 判定哪些敌人在范围之内
        foreach Enemy e in gameWorld
            // 将敌人的位置转换为2D坐标
            Vector2 enemyPos2D = Vector2(e.position.x, e.position.z)
            // 构造从玩家到敌人的向量
            Vector2 playerToEnemy = enemyPos2D - playerPos2D
            // 检查长度，看看是否在距离之内
            if playerToEnemy.Length() <= range
                // 旋转playerToEnemy，使得它相对于玩家朝向旋转（使用2D旋转
                // 矩阵）
                playerToEnemy = Rotate2D(angle)
                // 为敌人创建雷达光点
                RadarBlip blip
                // 取playerToEnemy向量，转换为相对于屏幕上雷达中心点的偏移
                blip.position = playerToEnemy
                blip.position /= range
                blip.position *= radius
                // 将blip添加到blips中
                blips.Add(blip)
            end
        loop
    end

    function Draw(float deltaTime)
        // 绘制雷达图片
        ...

        foreach RadarBlip r in blips
            // 在position + blip.position的位置绘制r
            // 因为blip中存放的是偏移量
            ...
        loop
    end
end
```

这个雷达还有一些可以改善的地方。首先，我们可能想要雷达不仅仅显示敌人，友军也能够显示。在这种情况下，只要将遍历代码改为同时遍历友军和敌军即可，然后根据敌友为 `RadarBlip` 设置相应的颜色。另外一个改善就是根据敌人的垂直高度在雷达上有不同的显示，这取决于敌人在玩家的上层、下层还是同层。为了支持这样的需求，我们可以在雷达系统中比较玩家和敌人的高度，从而可以判断光点的显示。

还有一个修改就是只显示那些最近开枪的敌人的光标，比如《使命召唤》。为了实现这个效果，每次敌人开火，都需要有一个记录来让雷达知道是否显示。当对所有潜在光标的敌人进行遍历的时候，我们只要检查这个记录就可以。如果记录没有查到，雷达上就不显示这个敌人。

其他需要考虑的 UI 问题

其他需要考虑的 UI 问题包括支持多套分辨率、本地化、UI 中间件及用户体验。

支持多套分辨率

对于 PC 游戏来说，通常都会有很高的分辨率，现在最新流行的显示器分辨率为 1920×1080 像素，但是其他分辨率也在用，比如 1680×1050 像素。这意味着每个显示器在 UI 显示像素上都有很大的不同，支持多套分辨率不仅仅在 PC 游戏领域才有。比如说，Xbox 360 和 PS3 都要求游戏支持传统的 CRT 电视机以及宽银幕的显示器（因为 Xbox One 和 PS4 不支持老式电视，所以这些平台不用太担心多套分辨率的问题）。

一个解决多套分辨率问题的办法就是避免使用像素坐标，也称为**绝对坐标**。一个绝对坐标的例子是让 UI 绘制在 (1900, 1000) 像素点。使用这种坐标的问题就是如果显示器只有一种 1680×1050 像素的分辨率，在 (1900, 1000) 位置的 UI 就会完全在屏幕之外。

这种问题的另一个解决方法是使用**相对坐标**，就是坐标是一个相对值。比如说，如果你想让某些东西在屏幕的右下角显示，可能会放置元素在相对于右下角位置的 (-100, 100)。就是说在 1920×1080 像素分辨率下，它的坐标会是 (1820, 980)，而在 1680×1050 像素分辨率下，坐标会是 (1580, 950)，如图 10.9 所示。

相对坐标可以根据屏幕上的一些关键位置（通常是屏幕角落或者中心）来表达，又或者相对于其他 UI 元素来表达。实现第二种方法稍微有点复杂，超出了本书讨论的范围。

一个细微的改良就是根据分辨率进行缩放。原因是如果在非常高分辨率的情况下，UI 元素可能因为太小而导致不可用。所以分辨率越高，UI 就应该越是放大，这样玩家才能看清楚。一些 MMORPG 游戏甚至允许玩家控制 UI 控件的缩放。如果要支持伸缩，使用相对位置就尤为重要。

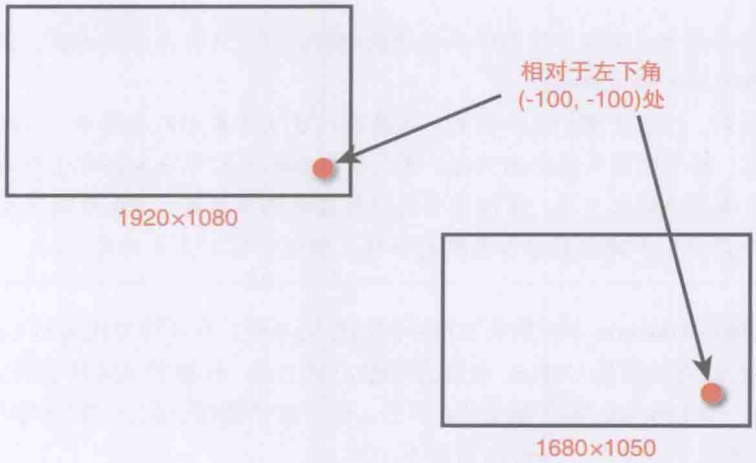


图 10.9 一个放置在相对于屏幕右下角位置的 UI 元素

本地化

虽然一款游戏只支持一种语言（通常是英语）也是可行的，但是大多数商业游戏都需要支持多语言。**本地化**就是支持更多语言的过程。由于许多菜单和 HUD 都有文本显示，在设计 UI 系统的时候就需要重视。哪怕一款游戏不需要本地化，将文本硬编码到代码里面本身就非常不好，这样不利于非程序员修改。但是如果游戏需要本地化，将硬编码移除就特别重要。

最简单的文本本地化方法就是将游戏中这些文本存储到外部文件中。这个外部文件可以使用 XML、JSON 或者类似的格式（在后续章节会更加深入地讨论相关话题）。这样紧接着就可以通过字典映射键来访问特定的字符串。因此不管代码是否需要在屏幕上显示文本，我们都需要用对应的键来使用字典。这意味着比起直接在按钮上显示文本“Cancel”，更应该使用“ui_cancel”键来从字典取得字符串。在这种情况下，支持新语言只需要创建新的字典文件。这种方法在第 14 章的塔防游戏中有所应用。

不幸的是，支持使用不同字符的语言会让问题变得复杂。传统的 ASCII 字符集（大多数语言都用 `char` 类型表示）只能支持英语字母。不支持象形文字，使得支持阿拉伯语和简体中文都要特别开发。

最后一分钟才想起本地化

在一个我曾经工作过的项目上，所有 UI 的开发都通过脚本语言来进行。在开发的过程中，没有人考虑过本地化。结果就是所有屏幕上显示的文本都是硬编码在脚本里面。到了需要本地化的时候，完全没人愿意手工去将脚本中的硬编码改为字典访问。

所以有了这个想法：脚本中的英语字符串就将错就错，只作为字典的键，其他语言的对应的值存储在外部文件中。

这样能起作用，但还是有一点小问题：如果有人改了脚本中的字符串，不知道本地化系统的存在，那个字符串就会出问题。这是因为本地化文件中不会有这个新的英语字符串。这在项目中让人头疼，所以对于大多数游戏都不推荐。这也告诫了我们一个件事情，那就是计划本地化是非常重要的事情，哪怕当时不是立刻需要。

大多数游戏都通过 **Unicode** 字符集来支持不同的编码系统。有多种方式来对 Unicode 字符进行编码，最流行的方法就是 UTF-8，这是使用最广的方法。标准的 ASCII 字符集在 UTF-8 中就是一个字节，而 Unicode 字符可以有 6 字节。由于这个宽度变化，UTF-8 字符串不能只根据内存长度而断定字符个数，每种语言都不一样。

但是改变文本字符串和字符编码不是唯一要考虑的事情。还有一个问题是一个单词在不同语言中的长度不一样。一个经常需要本地化的语言就是德语，德语单词的平均长度要比英语长（我找不到准确数据的文章，但是经验告诉我通常会长 20%~25%）。这就是说，如果一个 UI 元素刚好能够将英语放在上面，换成其他语言就很可能放不进去。这种问题是非常烦人的，因为可能会导致视觉效果要改很长时间才能完成。图 10.10 演示了英语版本的按钮和其他语言版本的按钮（通过 Google 翻译）。

图 10.10 按钮的本地化问题

除了文本和语音之外，本地化的另一个重要方面就是对游戏内容根据国家本地化。比如，《魔兽争霸 2》必须不能违反德国法律，不能使用第三帝国[1]的符号。其他国家也会有一些要求遵守的法律，比如去除流血或者恶心内容。最后，最难本地化的事情就是习俗上的问题，比如一件事在美国是没问题的，但是却冒犯了其他文化。这种类型的本地化就超出了用户界面的领域，但是如果要为其他地区发行游戏就需要注意。

[1]希特勒下的德国。——译者注

UI 中间件

回忆一下第 1 章，中间件就是一些外部代码库用于简化开发工作。谈及 UI，特别是 AAA 游戏，都会提及一个重要的中间件：Autodesk Scaleform。使用 Scaleform 的游戏包括《星球大战：旧共和国》、《生化奇兵：无限》。虽然 Scaleform 对于大型商业游戏非常流行，但是对于小型游戏来说则不一样，因为它不是免费的解决方案。

之所以使用 Scaleform 是因为它允许美工使用 Adobe Flash 来创建所有的 UI 内容。这样，程序员就不用花时间去关心手工设置 UI 元素的位置。大多数的 UI 编程工作都可以在 ActionScript 中完成，这是 Flash 中使用的类似 JavaScript 的语言。使用 Scaleform 需要花时间让系统与游戏引擎融为一体，但是一旦完成，UI 就可以进行多次迭代而无须系统变更。

用户体验

本章关于 UI 讲得比较少的一个重要方面就是**用户体验**（也叫 UX），是当用户使用界面时的交互。一个设计得不好的 UI 会让用户点击多个按钮来完成很简单的动作，但是这个问题在很多家用机 RPG 中都能经常看到。如果你的任务不只是编程，还有 UI 设计，考虑 UX 是非常重要的。但是，这个话题与编程没太大关系，我把更多的信息放在最后的相关资料中了。

总结

游戏 UI 是非常重要的——一个 MMORPG 游戏，比如《魔兽世界》，在屏幕上会同时显示一堆 UI，而极简游戏则尽可能避免 UI。通常而言，UI 是玩家接触游戏的第一个界面，通常通过初始菜单进入游戏。但是游戏过程中通过 HUD 也会有很多功能，比如准心、雷达、路点箭头都为玩家提供了重要信息。虽然有经验的程序员都会表达对于开发 UI 的不满，但是一个设计良好的 UI 可以大大改善玩家的体验。一个实现不足的 UI，会毁掉整个游戏的可玩性。

习题

1. 使用“菜单栈”的好处是什么？
2. 在我们实现将 key code 转换为字符的时候，key code 的什么属性能够让我们开发便利？
3. 在实现路点箭头的时候，如何确定路点箭头在世界空间中的位置？
4. 路点箭头计算中的角度和转向是如何实现的？
5. 描述一下如何让准心判定目标点是敌是友？
6. 在实现雷达的时候，对象的 3D 坐标是如何变换到雷达的 2D 坐标的？

7. 绝对坐标和相对坐标有什么不同?
8. 为什么 UI 文本不应该硬编码?
9. Unicode 字符集解决了什么问题?
10. 什么是用户体验（UX）?

相关资料

Komppa, Jari. "Sol on Immediate Mode GUIs." *http://iki.fi/sol/imgui/*: 这是一个通过 C 和 SDL 实现游戏 UI 不同元素的深入教程。

Quintans, Desi. "Game UI By Example: A Crash Course in the Good and the Bad." *http://tinyurl.com/d6wy2yg*: 这是一篇相对简短的文章，通过游戏例子讲解了什么是好 UI 什么是坏 UI，还讨论了在设计游戏 UI 的时候要考虑的事情。

Spolsky, Joel. ***User Interface Design for Programmers.*** **Berkeley: Apress, 2001.** 这个程序员设计 UI 的方法不是针对游戏，但是他在关于创造高效 UI 方面提供了有趣的视角。

第11章
脚本语言和数据格式

现在越来越多的游戏逻辑倾向于用脚本语言来完成。本章会讨论使用脚本的优点和缺点，以及多种语言之间的比较。

本章还会进一步讨论如何通过数据描述关卡和属性，以及如何存储数据。就像对比脚本语言一样，还会对比多种不同的数据表示，包括二进制格式和文本格式。

脚本语言

多年前，游戏全部使用汇编语言开发。这是因为早期的机器需要汇编级别的优化才能运行。但是随着计算能力的提升，而游戏又变得很复杂，使用汇编开发就变得越来越没意义了。直到某一天，使用汇编语言开发游戏带来的优点被完全抵消了，这就是为什么现在的所有游戏引擎都使用像 C++ 那样的高级语言开发。

同样，随着计算机性能的提升，越来越多的游戏逻辑开始从 C++ 或者类似的语言转移。现在许多游戏逻辑使用**脚本语言**开发，常用的脚本语言有 Lua、Python、UnrealScript 等。

由于脚本代码更加容易编写，所以策划写脚本是完全可行的，这就让他们得到了开发原型的能力，而不用钻进引擎里面。虽然 AAA 游戏的相当部分比如物理引擎或者渲染系统依然使用引擎语言开发，但是其他系统比如摄像机、AI 行为可能会使用脚本开发。

折中

脚本语言并不是万灵药，在使用之前必须考虑很多折中。第一个要考虑的就是脚本语言的性能远不如编译型语言，比如 C++。尽管比起 JIT 或者基于 VM 的语言，比如 Java、C#，那些脚本语言，比如 Lua、Python，在性能上都不具备可比性。这是因为**解释型**语言按需加载文本代码，而不是提前编译好。多数脚本语言都提供了编译为中间格式的选项。虽然始终达不到编译型语言的速度，但还是会比解释型语言要快。

由于这个性能差异的存在，性能敏感的代码不应该使用脚本语言开发。以 AI 系统为例，寻路算法 (比如 A*) 应该是高效的，因此不应该用脚本开发。但是由状态机驱动的 AI 行为应该完全用脚本开发，因为那不需要复杂的计算。哪些系统用脚本开发，哪些用 C++ 开发，表 11.1 中有一些例子。

表 11.1 脚本语言的例子

C++	脚本语言
渲染引擎	摄像机逻辑
AI（寻路）	AI（状态机）
物理系统	一般的游戏逻辑
文件加载	用户界面

使用脚本语言的巨大优势就是使得开发时迭代更加快速。假设某个游戏的 AI 状态机必须以 C++ 开发。在玩游戏的过程中，AI 程序员发现某个敌人的行为不正确。如果状态机使用 C++

开发，程序员需要很多工具去定位问题，而且在玩游戏的过程中通常没法解决。虽然 Visual Studio 的 C++ 确实有“编辑和继续”功能，但实际上它只有在某些情况下才能使用。这意味着通常程序员必须暂停游戏，修改代码，重新生成可执行文件，重新开始游戏，最后才能看到问题是否解决。

但是同样的场景如果出现在 AI 状态机是用脚本语言开发的时候，就可以动态重新加载脚本，然后在游戏仍在运行的时候就把问题解决了。运行中动态加载脚本的能力可以很大程度地提升生产力。

回到 C++ 版本的 AI 行为例子中，假设在警卫 AI 中有 bug，是由访问野指针引起的，那么通常都会引发崩溃。如果 bug 总是出现，游戏就会经常崩溃。但是如果状态机是使用脚本语言开发的，那可能只会让某个特定 AI 的角色行动不正常，而游戏的其他部分都是正常的。第二种情况要比第一种友好得多。

进一步来讲，由于脚本与可执行文件是分开的文件，使得提交工作更加简单。在大型项目中，生成可执行文件需要好几分钟，而且最终文件可能会有 100MB。这意味着如果有新版本，需要的人要下载整个文件。但是，如果使用了脚本语言，用户只要下载几 KB 的文件就可以了，这样会快得多。这不仅对发售后更新补丁非常有帮助，在开发中也同样有用。

由于生产力的优势，一个最好的经验法则就是，只要系统不是性能敏感的，都能从脚本语言中受益。当然，为游戏增加脚本系统本身也要成本，但是如果多个团队因此受益，那么很轻松就能回收成本。

脚本语言的类型

如果你决定用脚本语言开发游戏逻辑，那么接下来的问题就是，该使用哪个脚本语言？有两方面需要考虑：使用现有的，比如 Lua、Python，还是使用自己开发的。一些自己开发的脚本语言有 UnrealScript 和 QuakeC。

使用现有脚本语言的优势就是可减少很多工作量，因为编写脚本语言解释器非常花时间而且容易出错，哪怕你在工具的辅助下工作。进一步来讲，由于自己开发的脚本语言用户量要比现有的少得多，因此隐藏错误的可能性要高得多。请参看下面的“运算优先级中的错误”，这就是一个例子。

运算优先级中的错误

以下的数学运算有什么问题？

```
30 / 5 * 3
```

根据我对这门语言属性的理解，答案为 2。这是因为题目中的语言不恰当地赋予了乘

法的优先级比除法高。所以刚才的代码会按照下面这样解释：

```
30 / (5 * 3) = 2
```

因为乘法和除法应该有同样的优先级，表达式这样计算才对：

```
(30 / 5) * 3 = 18
```

这是这个脚本语言中存在了 10 年的 bug。没人告诉过负责该脚本语言的程序员，大概是因为这个程序员倾向于在数学表达式中写大量的括号。

一旦发现 bug，修复是非常简单的。但这也是会发生在自定义脚本语言上的故事。

另一个要考虑的脚本语言开发的事情就是需要设计者深入了解编译器和虚拟机知识——比大多数游戏开发的话题还要多得多。

但是现有的脚本语言的缺点就是不能与游戏结合得非常好。既有语言设计上的内存和性能问题上的顾虑，也有桥接脚本语言和引擎之间的问题。而自定义的脚本语言是为游戏大量优化过的——这些语言的功能是面向游戏的，比如 UnrealScript。

最后一件使用现有脚本语言需要考虑的事是，很容易找到水平高的程序员。已经有大量的程序员对 Lua 或者 Python 非常熟悉，但却很少有人会了解面向游戏的脚本语言。

Lua

Lua 是一门通用脚本语言，大概是现在游戏领域最流行的脚本语言。使用 Lua 的游戏的例子包括：《魔兽世界》、《英雄连》、《冥界狂想曲》等。Lua 这么流行的一个原因是它的解释器非常轻量——纯 C 实现大概占用内存 150KB。另外一个原因就是它非常容易做**绑定**，也就是在 Lua 中调用 C/C++ 代码的能力。它同时支持多任务，所以它可以让许多 Lua 函数同时运行。

语法上，这门语言有点像 C 族语言，但同时也有一些不同点。表达式结尾的分号是可选的，不再使用大括号控制程序流程。Lua 迷人的一个方面就是它的复杂数据结构只有一种，那就是表格，它可以以很多不同的方式使用，包括数组、链表、集合等。演示代码如下：

```
-- 这样注释 --
-- 这是一个数组
-- 数组从1索引开始
t = { 1, 2, 3, 4, 5 }
-- 输出4
print( t[4] )

-- 这是一个字典
t= { M="Monday", T="Tuesday", W="Wednesday" }
```

```
-- 输出Tuesday
print( t[T] )
```

虽然 Lua 不是面向对象的语言，但是通过表格完全可以做到面向对象。这种技术经常会用到，因为面向对象在游戏中非常重要。

UnrealScript

UnrealScript 是 Epic 为 Unreal 引擎专门设计的严格的面向对象语言。不像很多脚本语言，UnrealScript 是编译型的。由于是编译型的，它有着比脚本语言更好的性能，但也意味着不支持在运行时重新加载。用 Unreal 开发的大部分游戏逻辑都用 UnrealScript 完成。对于使用完整引擎的游戏来说（不是免费版的 UDK），UnrealScript 的绑定允许使用 C++ 实现。例子如图 11.1 所示。

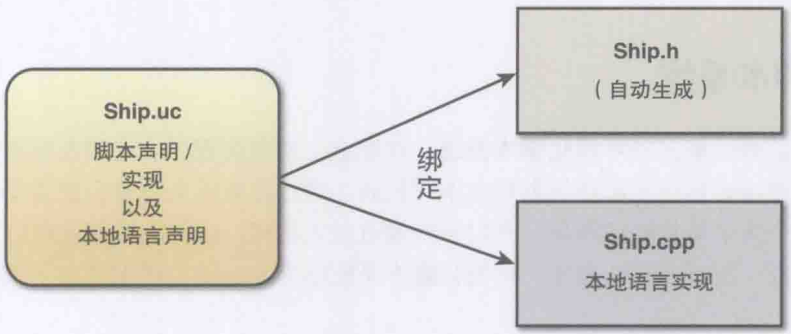


图 11.1 将 Ship 类绑定到 UnrealScript 中

在语法上，UnrealScript 看上去非常像 C++ 或者 Java。因为它是严格面向对象的，每个类都继承自 `Object` 或者 `Object` 的子类，而几乎每个类都表示场景中派生自 `Actor` 的一个角色。UnrealScript 非常特别的功能是内建对状态的支持。可以根据状态有不同的函数重载，这样对于 AI 行为会更加容易设置。以下的代码片段会根据该类当前状态调用不同的 `Tick` 函数（Unreal 对象的更新函数）：

```
// Auto表示进入的默认状态
auto state Idle
{
   function Tick(float DeltaTime)
   {
      // 更新Idle状态
      ...

      // 如果发现敌人，那么进入Alert状态
```

```
        GotoState('Alert');
    }
Begin:
    `log("Entering Idle State")
}

state Alert
{
    function Tick(float DeltaTime)
    {
        // 更新Alert状态
        ...
    }
Begin:
    `log("Entering Alert State")
}
```

可视化脚本系统

越来越多的游戏引擎支持**可视化脚本系统**，它看上去就像流程图。这些系统通常用于设置关卡逻辑。想法来自于比起让策划用文本写代码，通过流程图来设置会更简单一些。比如说，如果一个关卡策划希望在玩家开门的时候有敌人出现，这样的功能就可以用可视化脚本系统来完成。在 Unreal 引擎中，可视化脚本系统称为 Kismet。图 11.2 显示的是其中的截屏画面。

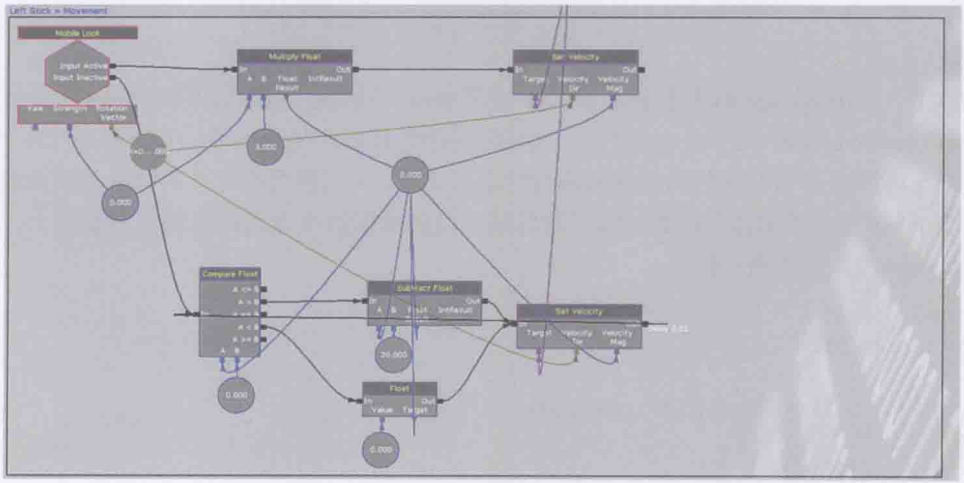


图 11.2 Kismet 可视化脚本系统

实现一门脚本语言

虽然一门脚本语言的完整实现超出了本书讨论的范畴，但是讨论那些必要的核心概念还是很有价值的。这样不仅能知道问题的复杂程度，还能提供入门指导。

实现自定义脚本语言与创建一个通用编译器类似。因此，本节讨论的很多话题都能在专门讨论编译器的书籍中找到（更多细节在相关资料中有提及）。学习编译器如何工作是很重要的，哪怕你不需要实现它，因为它能使编程变得高级。没有编译原理，每个人仍会用汇编语言写代码，这样就会导致有能力编程的人大大减少。

标记化

我们要做的第一步就是将代码文本加载进来，然后分成一块块的**标记**，比如标识符、关键词、操作符和符号。这个过程被称为**标记化**，更正式一点叫作**词法分析**。表 11.2 显示了简单的 C 文件分块成标记之后的样子。

表 11.2　C 语言的标记化

C 代码	标记
`int main(void)`	`int`
`{`	`main`
`  return 0;`	`(`
`}`	`void`
	`)`
	`{`
	`return`
	`0`
	`;`
	`}`

虽然手写标记器（也叫扫描器或者词法分析器）是完全可行的，但这不是推荐的做法。这样的代码写出来容易出错，因为有很多需要处理的情况。比如说，C++ 的标记器需要知道下面这些才是真正的 new 关键词：

```
newnew
_new
new_new
```

```
_new_
new
```

比起手写标记器，更倾向于使用像 flex 这样的工具，它会自动为你生成标记器。flex 的工作方式需要你指定一系列匹配规则，称之为**正则表达式**，然后声明哪些正则表达式匹配哪些标记。它会自动生成代码（用 flex 的话，会生成 C 代码），可以用来根据给定的规则开始标记化。

正则表达式

正则表达式（也叫 regex）在标记化之外还有很多用途。比如说，大多数 IDE 支持在多个文件之间使用正则表达式查找，这样可以找到那些特殊的序列。虽然正则表达式可以很复杂，但是脚本语言用到的模式匹配只用到正则表达式很小的一个子集。

最基本的正则表达式用于匹配关键词，也就是说每个字母必须一样。为了完成匹配，正则表达式只是带括号或者不带括号的一串字符：

```
// 匹配新关键词
new

// 同样匹配新关键词
"new"
```

在一段正则表达式中，还会有特殊含义的操作符。[] 操作符表示在其中的所有字母都会匹配。它还可以通过连字符表达区间，以下是一些例子：

```
// 匹配aac、abc或acc
a[abc]c

// 匹配aac、abc、acc……azc
a[a-z]c

// 你可以组合多种范围
// 类似上面那样，能匹配aAc……aZc
a[a-zA-Z]c
```

操作符 + 表示“一个或者多个特定字母”，而操作符 * 表示“零个或者多个特定字母”。这些可以通过 [] 操作符组合起来，这样就可以匹配某种语言的大多数标记。

```
// 匹配一个或者多个数字（整型标记）
[0-9]+

// 匹配单个字母或者下画线，后接零个或者多个字母、数字和下画线（C++中的
// 标识符）
```

```
[a-zA-Z_][a-zA-Z0-9_]*
```

在任何情况下，一旦一系列的标记和正则表达式对应起来，这些数据就可以提供给像 flex 这样的程序来生成标记器。一旦脚本语言传递到标记器，被分析为标记之后，就可以继续下一步了。

语法分析

语法分析的任务就是遍历所有标记，然后确保它们符合语法规则。比如说，`if` 表达式需要有适当数目和位置的括号、大括号、测试表达式和表达式来执行。在检测脚本语法的过程中，会生成**抽象语法树**（AST），它是基于树的数据结构，定义了整个程序的布局。简单的数学表达式的 AST 如图 11.3 所示。

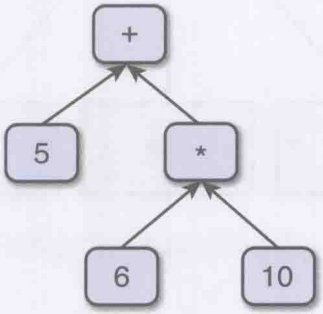


图 11.3　5 + 6 * 10 的 AST

注意，图 11.3 中的树是以后序遍历（左孩子、右孩子、父节点）的方式遍历的，结果会是 `5 6 10 * +`，这个结果就是中序表达式以后序表达式的方式表示的结果。这不是随意决定的，后序遍历在栈上计算非常方便。最后，所有 AST（不管是否是数学表达式）在语法分析之后都会被以后序的方式遍历。

在遍历 AST 之前，我们必须先生成一份 AST。生成 AST 的第一步就是定义一份语法。计算机语言定义语法的经典方式就是通过**巴科斯范式**，一般缩写为 BNF。BNF 的设计是相对简洁的。能够做整型加法和减法的运算子语法可以像下面这样定义：

```
<integer>    ::== [0-9]+
<expression> ::== <expression> "+" <expression>
                | <expression> "-" <expression>
                | <integer>
```

这个 `::==` 操作符表示“定义为”，`|` 操作符表示“或者”，`<>` 操作符用于表示语法规则的名字。所以上面的 BNF 语法的意思是，expression 要么是 expression 加另一个 expression，要么

是 expression 减另一个 expression，要么是一个 integer。这样 5 + 6 是有效的，因为 5 和 6 都是 integer，所以它们都是 expression，所以它们可以相加。

就像标记化一样，语法分析也有可以使用的工具。其中之一就是 bison，它可以在语法规则匹配的时候执行 C/C++ 动作。动作的一般用法就是让它读取 AST 的时候创建合适的节点。比如说，如果加法表达式匹配上了，就会为加法节点创建两个孩子：左右操作数各一个。

最好能有一个类能对应一种类型的节点。所以加/减语法会需要 4 种不同的类：一个抽象的表达式类、一个整型节点、一个加法节点和一个减法节点。层次如图 11.4 所示。

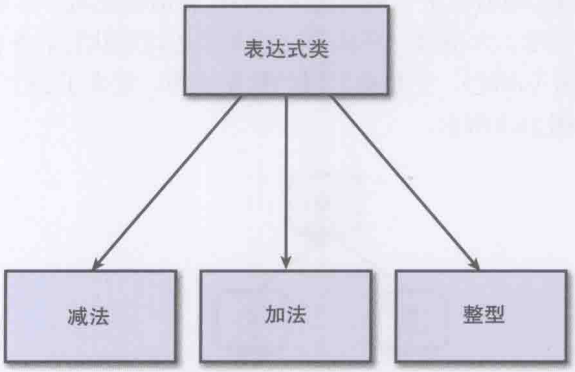


图 11.4 基本的加/减表达式的类层次结构

代码的执行和生成

在从脚本文件到 AST 生成之后，有两个可选方案，两种都需要以后序遍历的方式遍历 AST。对于传统一些的编译器，遍历 AST 的目标就是生成能够在目标机器上运行的代码。所以当 C++ 编译的时候，会遍历 AST，然后生成能够在目标平台上运行的汇编代码。

如果游戏脚本是编译型的，比如 UnrealScript，它生成代码是非常有意义的。但是如果语言是解释型的，代码生成就不是必需的。换言之，遍历 AST 然后执行节点表示的动作会是更好的方案。

对于加法/减法的例子，节点可以像下面这样定义：

```
abstract class Expression
   function Execute()
end

class Integer inherits Expression
   // 存储整数
   int value
```

```
    // 构造函数
    ...

    function Execute()
        // 将结果压到运算符的栈顶
        ...
    end
end

class Addition inherits Expression
    // 左右操作数
    Expression lhs, rhs

    // 构造函数
    ...

    function Execute()
        // 后续表示先访问左孩子，然后右孩子，最后到自己
        lhs.Execute()
        rhs.Execute()

        // 将栈顶的两个值相加，再将结果压到栈里
        ...
    end
end

// 减法节点和加法一样，除了取反之外
...
```

有了上面这些类作为节点，就可以通过调用根节点的 Execute 函数来得到 AST 的结果。在调用返回之后，留在栈上的数据会表示相应的运算结果。

比如说，表达式 5 + 6 的 AST 根节点为 Addition 类，左孩子为值为 5 的 Integer，右孩子为值为 6 的 Integer。如果根节点 Addition 的 Execute 函数被调用，它会先执行 5 的 Execute，这样会导致将 5 压入栈。然后在 6 上执行 Execute 会将 6 压入栈。最后，根节点上的 Execute 会将栈顶的两个值相加，就会得到 11，然后将 11 压入栈，这样就得到了最后的运算结果。

当然，如果脚本语言不仅要支持基本的运算，Execute 函数会变得更加复杂。但是后序遍历的基本原则可以跨越语言的复杂性和节点类型。

数据格式

另外一个游戏开发中要做的决定就是如何通过数据描述像关卡、游戏属性等游戏元素。对于非常简单的游戏来说，你可能不会管这么多，直接将数据硬编码，但这不是一个理想的解决方案。通过将数据存储在外部文件，就可以让非程序员来编辑。同时还使得创建编辑工具（比如关卡编辑器）来处理数据变得可能。

当你确定数据格式的时候，第一个要决定的就是数据是二进制格式还是文本格式。一个**二进制**文件通常都不具备可读性。如果你用文本编辑器打开二进制文件，可以看到一大串乱码。一个二进制格式的例子就是 PNG 格式或者其他的图片文件格式。一个**文本**文件，通常会用 ASCII 码表示，因此具备可读性。就像判断是否使用脚本语言一样，两种方法之间有一种折中。最后，有的情况用文本格式合理，而有的情况用二进制格式合理。

折中

使用二进制文件的第一个优点是体积更小，加载更快。比起花费时间解析文本并转换到内存格式，通常可以直接把整块加载到内存，而不需要任何转换。因为对于游戏来说，效率是非常重要的，所以对于携带大量信息的文件来说，通常都会采用二进制格式。

但是，效率的提升不是没有代价的。二进制文件的一个大缺点就是不支持版本控制系统（如果你对于版本控制不熟悉，可以看看附录 B）。原因是很难分辨出两个二进制文件到底有什么不同。

比如说，假设游戏将关卡数据存储为二进制格式（比如 Unreal 使用的.pkg 格式）。现在进一步假设策划用编辑器打开关卡然后修改了 10 多处地方，然后策划没有经过测试就直接提交了关卡。但是不知道什么原因，在调整之后就无法加载了。这时候，为了让游戏可以加载，唯有将关卡文件回滚到上一个版本，这样所有修改就丢失了。但是如果 10 个修改当中只有 1 个是坏的呢？在二进制格式里，通常没办法找到导致出错的那部分，所以所有修改都被回滚了。

以上场景忽略了一些问题——比如为什么策划提交时不做检查，为什么关卡编辑器保存出错数据？想要说的是，二进制格式的两个版本之间，除了加载场景之外，很难找到变化了什么。

对于文本格式而言，查看两个版本的不同就非常容易了。这意味着，如果关卡中有 10 处修改，只有 1 处导致不能工作，则可以查看文件，然后移除那些导致无法工作的修改。

同时也说明了一个事实，那就是文本文件对于最终用户的编辑来说是相对容易的。比如键盘输入的配置文件，采用文本格式会更好，因为使用标准的文本编辑器就能很容易地进行编辑。但在另一方面，这对于关卡数据不一定是优点，因为这会让玩家很容易就修改数据，然后将其破解。

还有最后一个方案，对于数据文本和二进制表达式都适用。在开发的时候，检查变动是很重要的，所以所有关卡和游戏数据都可以存成文本。然后，在发布的时候，我们可以加入一个**烘培**步骤，将所有文本格式转换为二进制格式。这些二进制文件不会进入版本控制系统，而且开发者只修改文本文件。这样，我们在开发的时候就能够得到文本格式带来的便利，同时又能在发布的时候获得二进制格式的优点。由于两种优点都达到了，因此这种方法是非常流行的。唯一要关心的就是测试——开发组需要确保文本的变动不会导致二进制出问题。

二进制格式

对于存储游戏数据来说，二进制格式通常都没有自定义格式。这是因为有很多方式存储数据，这些很大程度上取决于语言和框架。如果是 C++ 游戏，有时候最简单的方法就是将类的数据直接输出到外部文件，这个过程被称为**序列化**。但是有一些问题需要考虑，比如说，任何类中的动态数据都以指针形式存在，你需要执行深拷贝，然后重新构造数据。但是，二进制格式的设计超出本书的讨论范围。

INI

最简单的文本格式就是 **INI**，经常在用户需要改配置的时候使用。INI 文件是分为几节的，而每一节有一系列的键和值。比如说，INI 文件的图形设置可能会是这样的：

```
[Graphics]
Width=1680
Height=1050
FullScreen=true
Vsync=false
```

虽然对于简单数据来讲，INI 能工作得很好，但是对于复杂的数据而言就显得有些笨重。这对于关卡这种有布局的数据结构而言不太适合，比如说，INI 不支持嵌套的参数和节。

由于 INI 简单而且使用广泛，所以有着大量容易使用的库可供选择。在 C/C++ 中，我特别喜欢的一个库是 minIni，因为它在很多平台上都能工作而且使用简单。

XML

XML，全称 Extensible Markup Language，是一种 HTML 概念扩展出来的文件格式。比起使用 HTML 标签，如 `<html>`、`<head>`、`<body>` 等，在 XML 中，你可以使用任意自定义的标签和属性。所以第一眼看上去，它很像 HTML，尽管它不是。《巫师 2》使用 XML 存储所有能够在游戏中找到的物品的配置。比如说，下面是某一项，其存储了某个剑的状态：

```
<ability name="Forgotten Sword of Vrans _Stats">
   <damage_min mult="false" always_random="false" min="50" max="50"/>
   <damage_max mult="false" always_random="false" min="55" max="55"/>
   <endurance mult="false" always_random="false" min="1" max="1"/>
   <crt_freeze display_perc="true" mult="false" always_random="false"
    min="0.2" max="0.2" type="critical"/>
   <instant_kill_chance display_perc="true" mult="false"
    always_random="false" min="0.02" max="0.02" type="bonus"/>
   <vitality_regen mult="false" always_random="false" min="2" max="2"/>
</ability>
```

关于 XML 的一个批评就是需要很多额外的字符来表示数据。有很多 < 和 > 符号，而且总是需要用名字和引号等修饰每个参数，总是需要确保有配对的标签，所有的组合导致文件比较大。

但是 XML 的一大优势就是可以使用**模式**，就是强制要求哪些字段是必须要有的。这就是说，很容易验证 XML 文件和确保它声明了必要的参数。

像其他常见文件格式一样，有很多解析器都支持 XML。用于 C/C++ 最流行的解析器毫无疑问就是 TinyXML（C++ 会附带 ticpp）。一些语言会有内建的 XML 解析。比如，C# 有 `System.Xml` 命名空间用于处理 XML 文件。

JSON

JSON，全称 JavaScript Object Notation，比起 INI 和 XML 这种新型的文件格式，JSON 在近几年非常流行。虽然 JSON 在互联网交换数据中应用比较多，但在游戏中用于轻量级数据格式也是可以的。回忆一下第 6 章中我们使用了 JSON 来存储声音元数据。有大量的第三方库可以解析 JSON，包括 C++ 的 libjson（*http://libjson.sourceforge.net*）和 C# 的 JSON.NET。

根据存储数据的类型，JSON 可能与 XML 相比速度更快、体积更小。但也不总是这样。比如说，如果我们将《巫师 2》中剑的数据以 JSON 文件表示，通常会比 XML 版本要大：

```
"ability": {
   "name": "Forgotten Sword of Vrans _Stats",
   "damage_min": {
      "mult": false, "always_random": false, "min": 50, "max": 50
   },
   "damage_max": {
      "mult": false, "always_random": false, "min": 55, "max": 55
   },
   "endurance": {
      "mult": false, "always_random": false, "min": 1, "max": 1
   },
```

```
    "crt_freeze": {
        "display_perc": true, "mult": false, "always_random": false,
        "min": 0.2, "max": 0.2, "type": "critical"
    },
    "instant_kill_change": {
        "display_perc": true, "mult": false, "always_random": false,
        "min": 0.2, "max": 0.2, "type": "bonus"
    },
    "vitality_regen": {
        "mult": false, "always_random": false, "min": 2, "max": 2
    }
}
```

哪怕你对 JSON 和 XML 都减少多余的空格和回车，在一些特殊的例子里，JSON 文件还是会比 XML 要稍稍大一些。所以虽然 JSON 在大多数情况下是好用的文本格式文件，但是在《巫师 2》的例子中，开发者做出了正确的选择，使用 XML。

案例学习：《魔兽世界》中的 UI Mod

现在我们已经了解过脚本语言和文本格式的数据表示，那么来看一个两者都用上的系统吧：《魔兽世界》中的用户界面。在大多数游戏中，UI 系统是由设计师和程序员完成的。大多数游戏会有一些选项和设置，但通常都不能超越 UI 自身。但是，在 MMORPG 中，玩家会有很多设置，比如，有的玩家希望敌人的血量显示在屏幕的正中央，而其他玩家则希望显示在右上角。

暴雪想做的是创建一种高度可配置的 UI 系统，他们想要的系统不只是美术替换这么简单，而是会有新的控件产生。为了达到目标，他们需要创建一个非常依赖于脚本和文本数据的系统。

《魔兽世界》中的两个主要控件是布局和行为。布局就是界面中图片、按钮、控件的放置，存储为 XML。而 UI 的行为则使用了 Lua 脚本语言。让我们更深入地了解一下它们。

布局和事件

界面的布局完全是 XML 驱动的，用于设置基本的控件，比如框架、按钮、滑动条、复选框等，UI 开发者都可以使用。所以如果你想让一个菜单看起来像内建的窗口，那么使用正确的元素就可以做到。在这个系统中可以继承某个控件，然后修改属性。因此，会有派生自基础布局的自定义按钮的 XML 布局，然后修改相应的参数。

同样在 XML 文件里，插件指出哪里有事件可以注册。在《魔兽世界》的插件中，有许多不同的事件可以注册，有的关于控件，有的关于游戏。控件事件包括点击按钮、打开面板、滑动滑动条等。但是事件机制的威力来自于丰富的事件。不管你的角色制造伤害、受到伤害，还是与另一名角色聊天、在拍卖行竞标物品，还是其他各种各样的事情，都会有事件产生。UI 可以注册所有这些事件。比如说，一个可以判断你的 DPS（每秒输出伤害）的 UI 插件，可以通过跟踪战斗伤害事件来完成。

行为

每个插件的行为都通过 Lua 实现，这样可以快速实现原型，而且可以在游戏运行中重新加载 UI。由于使用了基于表格的继承系统，可以修改父类的函数实现重载。大多数的插件代码专注于处理事件和完成表现。每个注册了的事件都需要相应的 Lua 代码来处理。

问题：玩家自动操作

虽然刚开始的时候《魔兽世界》的 UI 系统非常成功，但是还是出现了一些问题。一个开发商关心的问题就是**玩家自动操作**，就是玩家玩游戏只通过 UI 来进行，这样游戏乐趣就会降低。为了解决这个问题，游戏设计为系统中的每个插件收到一次按键事件最多只能做一个动作。因此，如果你按下空格键，插件最多能够帮你释放一个法术。

虽然这样阻止了全自动，但还是可以开发一个能够完成大部分工作的插件。比如说，一个治疗型角色可以通过不断按下空格键来完成以下动作：

1. 遍历组中的所有玩家。
2. 找到血量最少的玩家。
3. 根据本局玩家的血量，释放相应的法术。

这样玩家就可以只按空格键来通关整个游戏。如果逻辑顺利，能够以最少的技能让团队保持健康。为了解决这个问题，暴雪修改了系统，使得 UI Mod 再也不能释放法术。虽然这没有完全解决问题（因为总有第三方程序会干这种事），但它消除了通过 UI 来玩整个游戏的方式。

问题：UI 兼容性

另一个 UI 系统的问题就是界面的核心行为会轻易被玩家修改，这很容易使得游戏不能玩。这在新补丁发布的时候经常发生。通常，这个变动会导致 UI API 的变化，也就是说，如果你运行旧的代码，会有可能让 UI 系统崩溃。

这演变成了暴雪技术支持的问题，因为每次发布新补丁，都会被大量的玩家不能攻击、聊天、施法等 UI 崩溃问题困扰。虽然可以关闭过期的插件，但是大多数玩家都不会这么干，因为很多旧插件在新版本中工作得很顺利。最后的解决方案就是增加新的“安全”代码，也就是那些来自内建 UI 的调用。这些安全调用处理了所有的基础功能，比如攻击和聊天，而插件再也不能重载这些函数。不再重载它们之后，只能在函数上建立钩子，就是在每次安全函数执行完毕，自定义钩子才有机会执行。

结论

最终，《魔兽世界》的 UI 插件非常成功，从此以后很多 MMORPG 都提供了类似的系统。由社区创造的插件不断增加，而有的太过成功，以至于暴雪将其并入到游戏核心当中。系统的灵活性和扩展性离不开脚本系统和文本数据的使用。

总结

脚本语言由于能大幅提高生产力，被越来越多的项目使用。因此，许多逻辑程序员大多数时间都在编写脚本。有的游戏使用现有的脚本语言，比如 Lua，而有的则为游戏引擎自定义脚本语言。如果必须使用自定义脚本语言，可以采用很多编译器中用到的技术，包括词法和语法分析。

关于如何存储游戏数据，在文本格式和二进制格式中充满着折中权衡。文本格式版本之间的变化非常清楚，而二进制格式则总是有着更高的运行效率。本章提到的文本格式例子有 INI、XML 和 JSON。最后，我们在《魔兽世界》的 UI 系统中看到了脚本语言和文本数据的结合。

习题

1. 在游戏中使用脚本语言有什么优点和缺点？
2. 举 3 个使用脚本语言是有意义的例子。
3. 举一个自定义脚本语言的例子。比起通用脚本语言有什么优点和缺点？
4. 什么是可视化脚本系统？可以做什么？
5. 将下面的 C++ 表达式分割成相应的标记：

```
int xyz = myFunction();
```

6. 以下正则表达式匹配什么？

```
[a-z][a-zA-Z]*
```

7. 什么是抽象语法树，可以用来做什么？
8. 描述一下文本数据和二进制数据之间的折中。
9. 你会采用哪种文件格式存储基本的配置设定，为什么？
10. 《魔兽争霸》界面系统中的两个主要组件是什么？

相关资料

Aho, Alfred, et. al. *Compilers: Principles, Techniques, and Tools* (2nd Edition). Boston: Addison-Wesley, 2006. 这本是经典书籍“龙书”的改进版，深入讲解了很多编译器背后的概念。其中的很多知识都可以用于实现自定义脚本语言。

“The SCUMM Diary: Stories behind one of the greatest game engines ever made.” Gamasutra. *http://tinyurl.com/pypbhp8*：这是一篇关于 SCUMM 引擎和脚本语言的非常有趣的文章，这个引擎几乎开发了所有 LucasArts 经典冒险游戏。

第12章 网络游戏

网络游戏让玩家们可以通过网络互相连接在一起并在一起玩。近些年最流行的一些游戏全部都支持网络——不管是《光晕》、《使命召唤》还是《魔兽世界》，有的甚至只能在有网络的环境下玩。时至今日，与真人对抗游戏所带来的体验也是 AI 所不及的。

实现网络游戏是一个复杂的话题，这个话题需要一整本书去讲解。本章介绍了数据是如何通过网络传输的，以及游戏为了支持网络需要如何架构的基础。

协议

想象通过快递服务寄出真实的邮件。我们至少需要一个信封，上面写着这封邮件的寄信地址和收信地址。通常，还会需要贴上邮票。在信封里面放着的才是你真正需要传递的信息——邮件本身。**数据包**可以想象成是在网络上传输的电子信封。在它的**数据头**中有地址和其他相关信息，然后才发出真正的**数据帧**。

对于信封来说，地址有较标准的格式。寄信地址在左上角，目的地址在右边中间，而邮票在右上角。这是大多数国家最常见的格式。但是对于网络数据传输，有着很多不同的**协议**或规则来定义数据包以什么格式以及为了发送必须做什么。网络游戏现在通常会让游戏逻辑使用两种协议之一：TCP、UDP。有的游戏会使用第三种协议，ICMP，常用于一些非游戏逻辑的功能。本节将讨论这些不同的协议以及它们的用途。

IP

IP，全称 **Internet Protocol**（网际网络协议），要通过网络发送数据，这是需要遵守的最基本的协议。本章提到的每一个协议，不管是 ICMP、TCP 还是 UDP，都必须附加 IP 协议才能够传输数据。哪怕数据只是在本地网络上传输。这就造就了一个事实，那就是现在所有在本地网络上的机器都需要有一个特定的本地地址，只有这样才能通过 IP 标识某个特定的本地机器。

IP 的数据头有着大量的字段，如图 12.1 所示。我不会对每个字段都进行讲解，因为网络上有无数的资源都讲了这些内容。

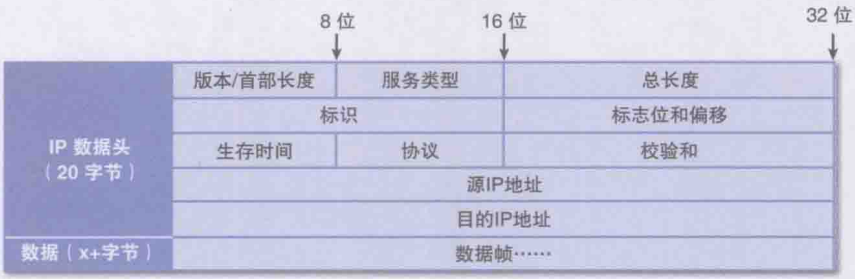


图 12.1 IPv4 数据头的结构

有两个广泛使用的 IP 协议版本：IPv4 和 IPv6。IPv4 地址有 4 个用点号分隔的 8 位数字（也叫八进制数）。所以，举个例子，本地网络地址可能是 192.168.1.1。由于 IPv4 地址是 32 位的，所以大概有 40 亿种可能组合，在 1977 年 IPv4 第一次出现时，这看起来是非常庞大的地址空间，但是到了今天，这个哪怕厨房都会有程序连接到互联网的时代，40 亿就完全不够用了。

为了解决这个问题，IPv6被开发了出来。它使用128位地址，所以它支持惊人的2^{128}的地址空间。由于地址字节数提升了4倍，IPv6数据头肯定是不同的。IPv6的一个缺点就是它们很难记忆，因为它们的完整形式是用冒号隔开的十六进制数，就像是2001:0db8:85a3:0042:1000:8a2e:0370:7334这样。IPv6在2012年6月6日正式启动，而在未来的某个时刻，IPv4将会被抛弃。现在，大多数计算机通过IPv4和IPv6都可以连接到互联网。但是如果你的平台和ISP支持，建议换到IPv6。

ICMP

ICMP，全称**Internet Control Messaging Protocol**（网际网络控制消息协议），并不用于在网络上传输大量数据。因此，它不能用于发送游戏数据。这就是说，在编写多人游戏时ICMP有一个方面是相关的：发送**回声**包的能力。通过ICMP，可以发送数据包给特定地址，然后直接让数据包返回到发送者，这是通过ICMP的回声请求和回声响应机制完成的。这个回声功能可以用于测量两台计算机之间发包所需要的时间。

行程往返的时间，就是**延迟**，在玩家有多台服务器可以选择连接的时候特别有用。在游戏测量所有可以连接的服务器的延迟之后，就可以选择连接延迟最低的服务器。这个测量某个地址的延迟的过程称为**ping**。

当回声请求发送之后，接收者收到数据包接着以正确的回声包回传回去。由于ICMP数据头没有任何的时间戳信息，在发出回声请求之前，发送者需要自己决定当前的时间戳，然后记录到数据帧里面。当回声应答的数据帧返回之后，发送和接收的时间戳的差别就可以计算出来，这样就得到往返时间。这个过程通常会重复很多次，这样就可以得到平均延迟。ICMP回声请求数据包的结构如图12.2所示。

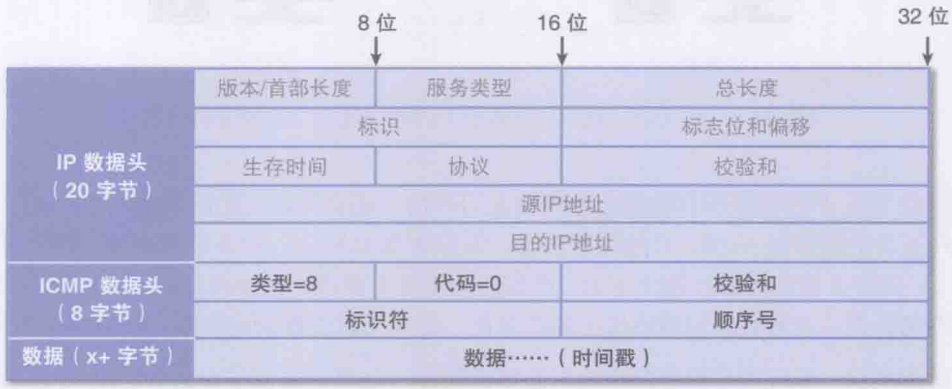


图12.2 ICMP回声请求数据包的结构

ICMP 数据包中的校验和被用来确保数据包在传输的过程中不会有数据出错，校验和的值是用 ICMP 数据头和数据帧计算出来的。如果接收者收到 ICMP 数据包，但是校验和不正确，这个包会被拒绝。因为校验和是基于数据包中的所有其他数据计算得出的，所以在数据包传输之前一定要计算好。标识符和顺序号由回声请求指定——通常每台机器的标识符都不一样，而顺序号应该随着每次发送新的回声请求的时候递增。

TCP

传输控制协议（TCP）是游戏在网络上用来传输数据最常用的两个协议之一。TCP 是一个基于连接的、可靠的、保证顺序的协议。可靠传递听起来很好，但随后我们会进一步讨论，TCP 协议在游戏上的应用通常没有 UDP 流行。

TCP 是基于连接的，就是说两台计算机在任何数据传输之前，必须先建立好彼此的连接。连接完成的方法是通过握手。请求连接的计算机发送一个连接请求到目标计算机，告诉它自己想要如何连接，然后接收者确认这个请求。这个确认在被最初的请求者再次确认之后，三次握手的过程就完成了。

一旦在 TCP 连接建立之后，就可以在两台计算机之间传输数据。之前提到，所有通过 TCP 发送的数据包都是可靠的。可靠的工作原理就是在每当数据包通过 TCP 发送以后，接收者都会告诉发送者我收到数据包。如果发送者在一定时间之内 (超时) 没有收到应答，就会将数据包再发送一次。发送者会不断地发送数据包，直到收到应答为止，如图 12.3 所示。

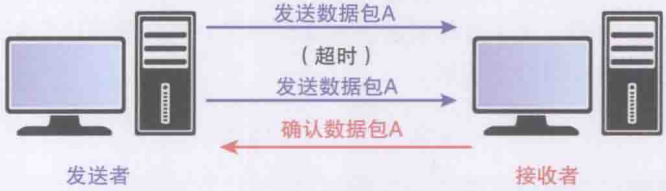


图 12.3 在 TCP 当中，发送者会不断发包，直到收到应答

结果就是 TCP 不仅保证所有数据包的接收是可靠的，还会保证它们的顺序。举个例子，假设现在有 3 个数据包 A、B、C 按顺序发送。如果 A 和 C 到达，而 B 没有到达，接收者不能处理 C，除非 B 到达之后才能往下走。所以需要等待 B 重传，在收到之后，才可以继续。由于**数据包丢失**，或者数据包传不过去的百分比，会大大减慢数据的传输，如图 12.4 所示。

对于游戏来说，保证顺序很容易成为不必要的瓶颈。如果之前例子中的 A、B、C 包含了某个玩家的位置信息：就是最开始玩家在 A 位置，然后在 B 位置，最后在 C 位置。在位置 C 收到以后，游戏就不用关心位置 B，因为玩家不在那个地方了。但是使用 TCP，游戏在收到位置 B 以前是无法使用位置 C 的，这对于 TCP 显然不是理想的场景。

图 12.4　TCP 数据包收到的顺序不正确

还有一个 TCP 需要考虑的有用的方面。所有网络都有 MTC，或者 **maximum transmission unit**（最大传输单元），它会决定数据包的大小限制。如果你尝试发送大于 MTU 的数据包，它会没法通过。幸运的是，TCP 在设计上会由操作系统自动将大的数据块分成合适大小的数据包。所以如果你需要从网站上下载 1MB 的文件，如果使用了 TCP，那么分解为合适大小的数据包以及保证接收顺序的事情，程序员就不用操心了。

在大多数场景中，通常不需要在游戏中传输那么大量的数据。但是还是会有用到的情况，比如说，如果游戏支持自定义地图，就会有一个问题，那就是新玩家试图加入游戏会话的时候是没有这张自定义地图的。通过 TCP，就可以轻松地将自定义地图发送给试图进入游戏的新玩家，而且不用管地图的大小。

也就是说，对于真正的游戏逻辑来说，只有小部分游戏类型需要用到 TCP。对于回合制游戏，使用 TCP 是有意义的，因为通过网络发送的所有信息都是相关的，它定义了某个回合中玩家执行的动作。另一个常用 TCP 的游戏类型是 MMO，特别是《魔兽世界》，它的所有数据都用 TCP 传送。由于《魔兽世界》中的所有数据都是要保证顺序的，所以使用内部就能保证顺序的协议是很合理的。但是对于那些有更加实时动作的游戏来说，比如 FPS 或者动作游戏，通常都不会使用 TCP/IP。这是因为对所有数据包的保证会影响游戏性能。

由于 TCP 是一个相当复杂的协议，TCP 数据头跟 IP 数据头一样大（20 字节）。TCP 数据头的结构如图 12.5 所示。

与 IP 数据头一样，我不会讲解 TCP 数据头的每个元素。但是，一个重要的概念必须要谈及——**端口**。想象某个特定地址的办公室有 50 名雇员。在这个场景下，有 50 个邮箱是非常合理的，每个雇员都有一个。接收员会收到每天来自邮件服务的邮件派送，然后将每个雇员的邮件放到对应的邮箱当中。没有邮箱，雇员找自己的邮件就会非常困惑，因为他们需要在乱七八糟的大堆邮件里面找。

在数据包经过网络发送之后，相同的排序方式也被应用于端口。比如说，大多数 Web 服务器在 80 端口接收连接，这就是说通过其他端口发送的数据都会被 Web 服务器忽略。互联网

服务中还有大量的专用端口，虽然这不是强制的。也就是说，你可能不希望你的游戏端口与 80 端口冲突。总共有大约 65000 个端口可以选择，虽然有的已经被指定使用，但是大多数系统都会有大量的端口空闲着。游戏应该选用一个没有被其他程序使用的端口。要注意的是，源端口和目标端口不一定是一致的，也就是说，虽然 Web 服务器连接建立在 80 端口，但是客户端连接服务器可以使用不同的端口。

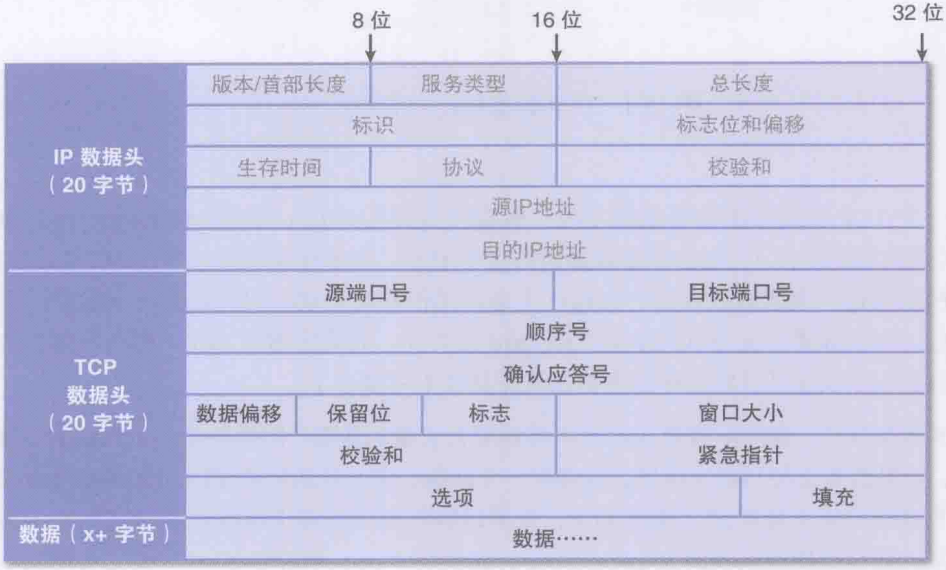


图 12.5 TCP 数据头的结构

UDP

数据包协议（UDP）是一种无连接、不可靠的协议。就是说你可以直接发 UDP 数据包，而不需要与指定目标建立连接。由于它是一个不可靠协议，所以不会有保证数据包到达的手段，也不会保证数据包到达的顺序，也没有接收者应答的功能。由于 UDP 是一种更加简单的协议，数据头比 TCP 要小得多，如图 12.6 所示。

像 TCP 一样，UDP 也支持大约 65000 个端口。UDP 端口和 TCP 端口是独立的，所以如果 TCP 和 UDP 使用同一个端口是不会冲突的。由于 UDP 是不可靠的，UDP 的传输比 TCP 要高效得多。但是由于不知道数据包是否到达，也会造成一些问题。虽然有些数据不太重要（比如对手的位置信息），但还是会有一些重要的保证游戏状态一致的数据。如果玩多人 FPS 游戏，你发射了子弹，这个信息就很重要，要保证它被服务器或者其他玩家所接收。

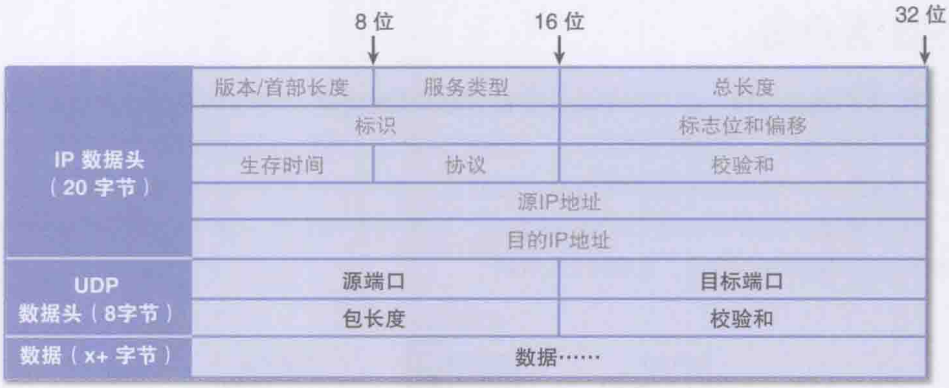


图 12.6 UDP 数据头的结构

一个尝试的解决方案就是将 TCP 用于关键数据，使用 UDP 传输不太重要的数据。但是 INET 97 proceedings paper 指出（在相关资料中有提及），在 TCP 保证系统运行的同时，使用 UDP 会导致 UDP 丢包增加。一个更严重的问题是：虽然移动数据不是那么重要，用 UDP 传送很合理，但我们还是需要数据包的顺序。没有顺序的信息，如果位置 B 在位置 C 之后收到，玩家就会被不正确地移动到旧位置。

大多数游戏处理这个问题都是使用 UDP，然后在所需的数据包里增加一些自定义的可靠层来完成。这个额外的层在 UDP 数据段的开始位置添加——可以认为是自定义的协议数据头。最基本的可靠性数据是**顺序号**，可以跟踪哪个数据包号是哪个，然后通过设置位域来应答。通过使用位域，某个数据包可以同时应答多个数据包，而不需要每个数据包都应答一次。这个系统同时还有灵活性，就是如果某个系统不需要可靠性和顺序信息，那么可以不添加数据头直接发送。这个系统的实现超出了本书的内容，而本章参考资料中关于 *Tribes* 的论文则是这个系统的一个很好的演示。

就像之前所提到的，UDP 在实时游戏领域中是占主导地位的协议。几乎所有 FPS、动作类、RTS 以及其他网络游戏中对时间敏感的信息都会使用 UDP。这也是为什么几乎所有为游戏设计的网络中间件（比如 RakNet）只支持 UDP。

网络拓扑

拓扑决定了不同的计算机在网络游戏会话中是如何相互连接的。虽然配置上有很多种不同的方式，但大多数游戏都支持一种或两种模型：服务器/客户端或是点对点。对于很多情况，两种方法各有优劣。

服务器/客户端

在**服务器/客户端**模型中，有一个中心计算机 (也就是**服务器**)，所有的其他计算机 (也就是**客户端**) 都会与之通信。因为服务器与每一台客户端通信，所以在这个模型中，会需要一台有着比客户端更高带宽和处理能力的机器。比如说，如果客户端发送 10Kbps 数据，在 8 人游戏中，意味着服务器需要接收 80Kbps 的数据。这类模型通常也叫作中心型结构，因为服务器是所有客户端的中心节点，如图 12.7 所示。

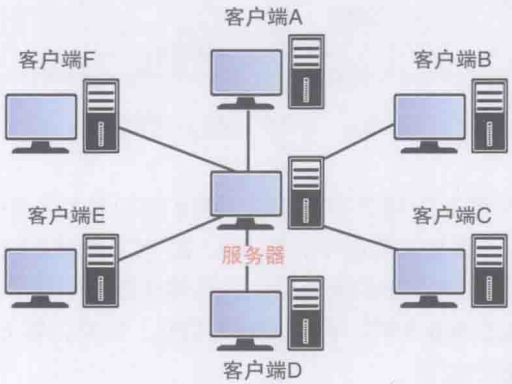


图 12.7 服务器/客户端模型

服务器/客户端模型是今天最流行的网络游戏拓扑结构。大多数 FPS、动作游戏、MMO、策略游戏、回合制游戏等都使用服务器/客户端模型。当然当中会有一些例外，但是确实很多网络游戏都使用服务器/客户端模型。

在常见的服务器/客户端模型的实现中，服务器会被认为是**权威的**，就是说它需要验证大多数客户端行为。假设网络游戏允许玩家向另一名玩家投掷闪避球，在另一名玩家被闪避球打中之后，投掷的玩家会得分。在权威服务器中，当玩家想投掷闪避球的时候，会先向服务器发起请求，服务器会检查这是否是一个合法动作。然后服务器会模拟闪避球的弹道，在每一帧检查这个球是否与某个客户端发生碰撞。如果客户端被击中，服务器会通知客户端被打败。

服务器验证的理由有两个。第一个理由就是服务器会拥有所有客户端的最新位置信息。一个客户端投出闪避球，可能会认为自己投中了，但这可能是因为当时位置不是最新的。而如果客户端能够用闪避球淘汰其他玩家而无须经过服务器验证的话，就很容易有外挂程序作弊淘汰其他玩家。本章后面会讨论到作弊这个话题。

因为服务器需要认证，服务器/客户端模型的游戏逻辑实现起来就比单人游戏更加复杂。在单人游戏中，如果用空格键发射导弹，相同的代码会检测空格键可以创建和发射导弹。但是在服务器/客户端游戏中，空格键代码必须创建发射请求数据包到服务器，然后服务器通知所有其他玩家导弹的存在。

因为这是两种完全不同的方法，对于想要实现两种模式（多人与单人）的游戏来说，最好的方法就是将单人模式当作特殊的多人模式。这在许多游戏引擎中是默认的实现方式，包括 id Softwave 引擎。就是说单人模式实际上是将服务器和客户端都在一台机器上运行。将单人模式看作多人模式的一种特例的优点在于，只需要一套游戏逻辑代码。否则，网络程序员要在权威服务器的游戏逻辑开发上花很多时间。

如果我们回到闪避球游戏的例子，想象一下如果玩家可以选择目标。就是说，他们需要用一些方法以知道对手玩家的运动方向才能够预判出成功的投掷。在最好的情况下，客户端可以以四分之一秒一次地收到服务器下发的对手玩家的位置更新数据。现在想象如果客户端只在收到服务器数据的时候才更新对手位置，就是说每隔四分之一秒，对手玩家都会上传新的位置，而对手位置总是闪来闪去。如果在这种情况下试着去击中一个闪避球——当然听上去游戏似乎不太好玩。

为了解决这个问题，大多数游戏都会实现某种**客户端预测**，也就是客户端会在两次服务器下发数据之间猜测中间的过渡情况。在移动的例子中，如果服务器在下发对手玩家的速度的同时一起下发位置，那么客户端预测就可以工作。然后，在服务器更新之间的几帧里，客户端可以根据最后收到的速度和位置，推算对手玩家的位置，如图 12.8 所示。

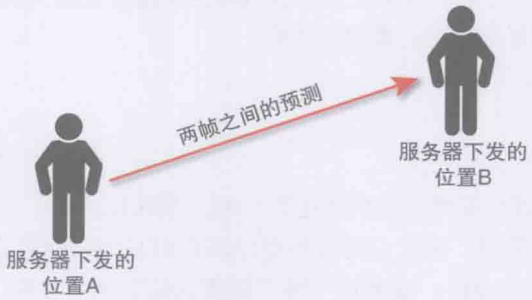


图 12.8　在服务器更新之间，客户端预测推算对手的位置

只要更新足够频繁，客户端就能让对手玩家在所有时刻都有足够准确的表现。但是，如果由于连接问题导致更新不够频繁，客户端预测就会变得不准确。由于服务器是最权威的存在，客户端必须修复预测位置与真实位置之间的差异。但是如果预测工作得足够好，就能看起来非常顺畅。

这个概念还可以延伸到本地终端上执行的动作。如果我们想要一个游戏逻辑很顺畅，就要在玩家按下空格键投掷闪避球的时候，屏幕上立刻有投掷动画。如果客户端要等到服务器确认闪避球投掷之后才开始，游戏会非常卡顿。背后的解决方案就是在服务器确认投掷有效的同时，本地客户端就可以开始播放投掷闪避球动画。如果结果是投掷是非法的，客户端会修复这个问题。只要玩家正确同步，投掷闪避球的时候，游戏的感觉就会非常顺畅。

尽管服务器/客户端模型是一种非常流行的方法，但还是有一些问题需要考虑。首先，有些游戏允许一台计算机同时运行服务器和客户端。在这种情况下，就会有**主机优势**，就是说服务器、客户端同时跑的玩家，会得到最快速的服务器响应。

发生这种问题的一款游戏叫作《战争机器》。问题来自于散弹枪伤害在光线投射的瞬间就发生了。因此如果两个玩家同时进行散弹枪对射会发生什么?——其中一个是主机，另一个是正常的客户端。每一次，主机都会赢得这种对射的战斗。这就导致每场比赛，玩家都会倾向于散弹枪，因为它很强大。这个问题在不久前被修复了，但这是早期散弹枪称霸的一个原因。

另一个服务器/客户端模型的问题就是，如果服务器崩溃，游戏立刻就结束，而所有客户端都失去与服务器的通信。而连接到新的服务器是很困难的（因为所有客户端的信息都不完整），所以不可能修复这个问题。就是说如果玩家是主机，而且马上要失败了，玩家只要退出游戏就可以重启所有玩家，这样就让游戏变得没意思了。但是从服务器/客户端模型来看，如果一个客户端延迟非常多，对其他玩家的体验影响不是特别大。

在任何事件中，为了防止主机带来的问题，许多服务器/客户端游戏只支持**专用服务器**。在大多数例子下，这意味着服务器是安装在一个特别的位置的（通常在数据中心），而所有玩家都需要连接到这些服务器（没有玩家可以做主机）。虽然网速快的玩家还是比网速慢的玩家有优势，但是通过将服务器放在第三方的方法将这种绝对优势大大减弱了。可是，运行专用服务器的缺点就是部署得越多，费用就越高。

点对点

在**点对点**模型中，每个客户端都连接到其他客户端，如图 12.9 所示。这意味着对于所有客户端都要求同样的性能和带宽。由于点对点模型中没有中心的权威服务器，会有很多种可能：每个客户端只认证自己的动作，或者每个客户端都认证其他客户端，又或者每个客户端都模拟整个世界。

RTS 类型中经常会用到点对点模型。正式一点的名称为**帧同步**模型，就是网络更新被分为每次 150ms 到 200ms 的回合更新。每当执行一次输入动作，这个命令都会保存到队列里，在每轮结束的时候执行。这就是为什么在你玩多人游戏《星际争霸 2》的时候，控制单位的命令没有立刻执行——在单位回应命令之前有明显的延迟，因为它们都在等待帧同步回合的结束。

因为输入命令通过网络传递，RTS 游戏中的每个客户端实际上是在模拟所有单位。它们像本地玩家一样处理输入。这也使得记录下所有对手的指令并在游戏结束后查看即时回放成为可能。

客户端的帧同步方法会使所有客户端都紧密地同步，没有任何玩家能够比其他玩家先走。当然，这种同步方式的缺点就是——如果一个客户端开始延迟，其他客户端都要等待，一直到

这个玩家赶上来。但是帧同步方法在 RTS 游戏里面非常流行，因为它通过网络传输的数据相对来讲会更少。比起发送所有单位的信息，游戏只在每分钟发送相对小数量的动作，即使是最好的 RTS 玩家每分钟最多也就发送 300~400 次指令。

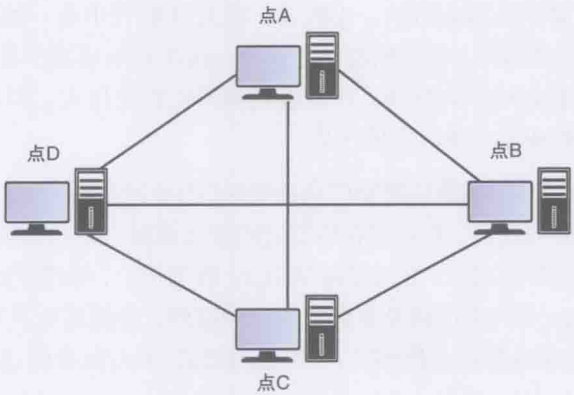


图 12.9　点对点模型

因为在点对点配置中，每个点都要模拟游戏世界的一部分，游戏状态必须保证 100% 的确定。这使得基于变化的游戏逻辑不太好做。如果《星际争霸 2》的狂热者可以根据掷骰来决定攻击伤害，这可能会导致这一点的模拟与另一点的模拟不一致。但这也不是说在点对点模型中完全不能有随机要素（比如《英雄连》），但是要付出更大的努力保证各点同步一致。

虽然 RTS 是将点对点模型用得最多的游戏类型，但还有其他游戏也这么做。一个著名的例子就是《光晕》系列，在多人对战中使用点对点模型。但是可以说，点对点没有服务器/客户端模型使用那么广泛。

作弊

任何网络游戏主要考虑的一点就是要为玩家打造公平的游戏环境。但不幸的是，有的玩家为了胜利会不择手段，哪怕打破游戏的规则。在网络多人游戏中，有很多规则都有可能会被打破，所以不管是否可能，网络游戏需要防止玩家作弊。

信息作弊

信息作弊是一种让玩家获得本不该拥有的额外信息的手段。假设你玩《军团要塞 2》，正在使用可以潜行跟踪的角色跟踪对手。通常情况下，间谍角色在潜行状态下，对于对手玩家来说是看不见的。但是我们已经知道，服务器需要下发其他玩家位置信息。如果服务器下发潜行玩家的位置信息，对于有经验的黑客来说让间谍（即使在潜行状态）可见那实在太容易了。

最简单的方法就是通过限制信息来防止信息作弊。在间谍角色的例子中，服务器应该不再为潜行的角色下发位置信息。在这种情况下，哪怕黑客知道如何让角色显形，最多能看到的就是间谍进入潜行之前的位置。虽然这也算作弊，但优势就相对弱了很多。

但是有的时候这个方案不是那么简单，比如说，在 RTS 游戏中有一种很常见的作弊称为图挂，就是让玩家看到整张地图，包括所有对手玩家。这在 RTS 游戏中是个巨大的优势，因为这使得玩家不仅可以看到对手的行动，还能看到对手正在建什么。因为 RTS 游戏单位之间存在相克关系，图挂就成为一个巨大的优势。

不幸的是，在图挂的例子中，不能够限制信息的发送，因为这会影响帧同步模型的工作。因为每个点都会模拟整个游戏世界，它必须在所有时间都知道敌人单位的位置，所以这种情况下唯一的办法就是建立反外挂对策。在《星际争霸 2》的例子中，暴雪就运行了名叫 Warden 的外挂检测程序。Warden 干的事情就是在游戏运行的时候，检测是否有其他外挂程序在运行。如果检测到了，就会把账号标记为作弊者。一段时间之后，暴雪就会将这些作弊者的账号屏蔽。

但是实际上没有任何外挂检测程序是不可战胜的——这在暴雪更新 Warden 的时候就引发了一场猫捉老鼠的游戏。暴雪更新 Warden，捉到作弊者，然后外挂程序修改之后，Warden 又一次无法检测到它们。一种普遍的对抗 Warden 的方法就是让外挂程序“杀转移”，就是一旦检测到 Warden 更新了就立刻退出，但是这样的系统只能成功一段时间。在任何事件中，暴雪和其他开发公平竞技网络游戏的公司都需要对作弊保持高度警惕。

游戏状态作弊

信息作弊能够让玩家获得不对称的优势，而**游戏状态作弊**则可以完全破坏游戏。一旦玩家可以修改游戏状态，玩家很快就能让游戏变得不可玩。常见于玩家作为主机的时候，如果玩家控制了服务器，就很难阻止玩家作弊。

在闪避球游戏的例子中，主机发送所有对手都被闪避球打中的消息，因此在这一回合中被淘汰了。因为服务器是权威的，因此游戏总是以主机胜利结束。如果没有第三方的专用服务器，那么还有几种解决方案。一种是使用前面提到的外挂检测程序。另一种则是客户端对服务器命令做一些检查。如果客户端看到多个玩家同时被淘汰，而这是不可能发生的，就可以触发一些代码从而认为服务器在作弊。

中间人攻击

影响最坏的一种作弊就是在两台通信机器之间设立一台机器，用于拦截所有数据包。这种称为**中间人攻击**，在通过网络发送明文消息的时候特别容易被人篡改。这种攻击是为什么我们连接到金融机构网站的时候应该使用 HTTPS 而不是 HTTP 的主要原因。

使用 HTTP，所有数据都以明文的方式通过网络发送。就是说，如果在网站上你用 HTTP 提交你的用户名和密码，容易遇到中间人窃取信息。而通过 HTTPS，所有数据都是加密的，这使得中间人几乎不可能访问到这些信息。虽然 HTTPS 协议暴露出几个脆弱的地方，但是对于大多数用户而言这都不是大问题。

但是，在游戏的情况下，中间人攻击最大的优点在于它允许作弊发生在不需要玩游戏的机器。这就是说几乎所有的外挂检测程序都不起作用——它们根本就不知道数据包被拦截而且被篡改了。还有一个问题是，一般通过访问数据包信息可能允许黑客更加深入地发掘其他能够渗透游戏的漏洞。

一种防止这类攻击的方法就是对所有数据包进行加密，这样只有服务器才能解密并理解数据包。但这对于大多数游戏消息来说都过重了，对所有信息加密实际上增加了负荷。但最少，我们可以在玩家登录（比如在 MMO）的时候进行一次加密，通过某种加密算法来有效避免玩家账号被盗。

总结

网络是一个非常深的概念，而本章介绍了重要的基础概念。所有网络游戏的基石就是协议，它使得数据能够通过网络传输。虽然 TCP 保证了可靠性，但是大多数游戏都选择使用更高效的 UDP。一旦选择好协议，下一个重大决定就是网络的拓扑结构类型。虽然今天大多数的实时游戏都使用服务器/客户端模型，但是大多数 RTS 游戏还在使用点对点模型。拓扑的类型会很大程度上影响网络游戏执行每次更新的方式。最后，任何多人游戏的开发者总会遇到的问题就是作弊者，而我们有很多方法可以打败他们。

习题

1. 什么是网际网络协议？IPv4 和 IPv6 有什么区别？
2. ICMP 是什么？如何应用在游戏上？
3. TCP 协议主要起什么作用？
4. 网络中的端口是什么？
5. 为什么对于实时游戏来说 UDP 协议是更好的选择？
6. 什么是服务器/客户端模型，与点对点模型有什么不同？
7. 有什么是客户端预测要做的？它是怎样为玩家提升游戏体验的？
8. 简单说一下 RTS 游戏如何应用点对点下的“帧同步”模型来工作？

9. 给一些信息作弊的例子，以及如何打败它？
10. 什么是中间人攻击，如何阻止？

相关资料

Frohnmayer, Mark and Tim Gift. “**The TRIBES Engine Networking Model.**” 这份经典的论文列出了如何实现为 UDP 增加可靠性和数据流功能，而且应用到了 *Tribes* 上。

Sawashima, Hidenari et al. “**Characteristics of UDP Packet Loss: Effect of TCP Traffic.**” **Proceedings of INET 97.** 这篇文章从技术的角度讨论了为什么 TCP 应答会让 UDP 丢包。

Steed, Anthony and Manuel Oliveira. ***Networked Graphics.*** **Burlington: Morgan Kaufmann, 2009.** 这本书深入地讲解了游戏网络编程的方方面面。

第13章

游戏示例：横向滚屏者（iOS）

《横向滚屏者》是一款横向视角的游戏。有着无尽的卷轴不断地滚动，一直到玩家失败。市面上有一款很流行的无限卷轴游戏叫作 *Jetpack Joyride*。

在本章中，我们先看看这款 2D 无尽卷轴示例游戏 *Ship Attack*。这款游戏用 Objective-C 语言在 Cocos2D 框架下开发，面向 iOS 设备。

概览

本章将在 review 游戏源代码的同时理解游戏，在本书的网站*http://gamealgorithms.net/source-code/shipattack/*有提供源码。不幸的是，代码在 Windows 下无法运行。因为这款游戏是针对 iOS 的，为了运行代码，你需要有一台 Mac（运行 Mountain Lion 或者更高的版本），同时还要有 Xcode（可在 App Store 免费下载）。

Ship Attack 的控制方法相当简单。水平拿着 iPhone，左拇指在屏幕左边控制上下左右移动，右拇指在屏幕右边点击发射子弹。游戏的目标就是在闪避敌人的子弹的同时用激光消灭他们。

Ship Attack 中的精灵由 Jacob Zinman-Jeanes 绘制，可以在 CC-By license 下使用。本书网站上有精灵的链接。这些精灵通过 TexturePacker 打包成了一张精灵表。更多关于精灵表的信息，可以回到第 2 章查看。

现在，在我们详细 review 源码之前，先看看 *Ship Attack* 中用到的技术，如图 13.1 所示。

图 13.1 *Ship Attack* 的截屏

Objective-C

Objective-C 语言在 20 世纪 80 年代早期就开发出来了，致力于为 C 语言添加面向对象特性。它与 C++ 几乎同时开发出来，但是是不同研究室的不同研究项目。Objective-C 用得不多，一

直到 NeXT，一家由乔布斯创办的公司，将这门语言应用到他们的新平台为止。虽然 NeXT 一直不是特别流行，但是随着乔布斯 1996 年回归苹果之后，他将 NeXT 也带了过去。之后 NeXT 操作系统就作为 Mac OS X 以及后来的 iOS 操作系统的基础。今天，所有在 Mac 以及 iOS 设备上运行的 GUI 程序必须至少在某些层级上使用 Objective-C 开发。

C++ 与 Objective-C 有一些显著的不同，最显著的差别就是从技术上讲 Objective-C 没有成员函数的概念。而且，类之间通过发消息以及响应消息来工作——这就是说所有的消息必须动态传递和处理，而 C++ 成员函数只有在用了 virtual 关键字之后才拥有动态特性。尽管从技术上讲没有成员函数，但本章还是这么称呼它们，因为这个术语大家比较熟悉。

Objective-C 中消息传递的语法与 C++/Java 这样的语言风格比起来有点奇怪。调用 C++ 的成员函数，只要像下面这样就可以：

```
myClass->myFunction(arg);
```

但是传递相同的消息到 Objective-C 中，语法是这样的：

```
[myClass myMessage: arg];
```

在浏览代码的同时，你可能也注意到，函数的声明前面可能会有 + 号或者 - 号。一个 + 号标记了这个方法是一个类方法（可以认为是 C++ 中的静态函数），而 - 号标记了该方法是实例方法（就是正常的成员函数）。以下是各个符号的例子：

```
// 类方法（本质上是静态函数）
+(void) func1;

// 实例方法（正常的成员函数）
-(void) funct2;
```

如果你发现你自己被 Objective-C 的语法所迷惑，注意是可以限制 iOS 应用中使用 Objective-C 的。因为它与 C++ 代码的兼容性非常好，只有很少的约束。所以完全可以让应用中 90% 的代码都用 C++ 编写，并且只有与操作系统交互的时候才用 Objective-C。这种方法对于希望稍后在 Android 平台上发布的移动游戏来说是值得推荐的，因为这样减少了需要重写的代码量。但是，因为 *Ship Attack* 用了 Cocos2D 框架，所以完全 100% 使用了 Objective-C。

Cocos2D

Cocos2D（在 www.cocos2d-iphone.org 上可以下载）是一个 2D 游戏框架，可以同时运行在 Mac 和 iOS 平台。由于只有这两个平台，所以所有接口都用 Objective-C 编写。Cocos2D 提供了很多 2D 精灵的内建功能，所以开发 2D 游戏相当不费力。

Cocos2D 中的核心概念是基类 CCNode，用于表示**场景节点**，也就是那些有位置而且可以在屏幕上绘制的对象。CCNode 有很多派生类，包括 CCScene（表示整个场景）、CCLayer（场景中的一个特定层级），以及 CCSprite（一个精灵对象）。在最简单的情况下，一款 Cocos2D 游戏必须有一个 CCScene 和一个 CCLayer，但是大多数游戏都不止有这些。

CCScene 可以看作游戏的一个特定状态。比如说，在 *Ship Attack* 中，有着主菜单场景，也有游戏场景。一个 CCScene 最少要分配一个 CCLayer，但是也可能支持多个 CCLayer 有着不同的 z 排序。游戏场景有着 3 个不同的层级：远处的星云、主要对象层、快速滚动的星星区域。你通常都需要为 UI 单独指定一层，但是由于这款游戏很简单，没有必要将 UI 分离出来。

CCSprite 用于游戏中所有移动和交互的对象，包括玩家的船只、激光、敌人子弹以及敌人的飞碟。比起一个精灵一个文件，我更倾向于使用精灵表，可以通过 TexturePacker 生成。很方便的是，Cocos2D 可以自动导入 TexturePacker 创建的精灵表，所以加载和动画的过程相对于精灵来说简单直观。

Cocos2D 的一个缺点就是它只支持两个 Apple 平台。但是有一个 C++ 版本的库，叫作 Cocos2D-x，它支持很多平台，包括所有的主要移动和桌面平台。虽然类名和函数名与 Cocos2D-x 一样，但还是有不少的语言差异。比如说，Cocos2D-x 将 STL 作为数据结构，而 Cocos2D 必须使用 NSMutableArray 和类似的数据结构。对于 *Ship Attack*，我选择 Cocos2D 是因为相比于更新的 Cocos2D-x 来说，其文档更丰富一些。但是，随着越来越多的 Cocos2D-x 书籍进入市场，这很快就将不成问题。

代码分析

现在我们介绍下 *Ship Attack* 背后的技术，我们深入到源码当中了解这些系统是怎么实现的。总的来讲，本章的代码要远比第 14 章的简单。这是因为 *Ship Attack* 的玩法要比第 14 章塔防游戏的简单。

如果你浏览了所有项目中的文件夹，你可能会看到大量的文件，大部分都在 libs 目录下。但这些源文件都是 Cocos2D 的，它们不是针对 *Ship Attack* 创建的文件。因此看到这么多源文件的时候不要烦恼。你只要关心主要的 shipattack 目录下的几个文件即可。

AppDelegate

AppDelegate 类有 didFinishLaunchingWithOptions 方法，是所有 iOS 程序的入口。对于 Cocos2D 游戏来说，几乎所有 AppDelegate 相关代码都是自动生成的。AppDelegate 是通过 HelloWorld 场景模板创建出来的。对于这个游戏来说，我修改了这个文件，让它进入游戏就跳到主菜单场景。

除此之外，对自动生成的 AppDelegate 的唯一修改就是让它支持多点触摸，使得游戏支持多个手指同时按下。没有多点触摸，游戏的操作模式就无法顺利工作，因为操作的过程中同时需要移动和发射。

MainMenuLayer

MainMenuLayer 有一个类方法叫作 scene，会返回一个包含了 MainMenuLayer 作为唯一 CCLayer 的 CCScene。当一个场景只有一个层级时，这是 Cocos2D 推荐的做法。层本身有少量代码。它的功能就是显示游戏标题和创建有一个选项的菜单（开始游戏）。菜单在 Cocos2D 中可以通过 CCMenu 和 CCMenuItem 创建，而一个特定的菜单项被选中时可以包含一段执行代码。在这个示例中，选中开始游戏选项，游戏就会过渡到 GameplayScene。

GameplayScene

顾名思义，GameplayScene 就是一个表示主要游戏玩法的场景。如果你看了这个文件，可以看到 3 个不同的层：两个 ScrollingLayer 和一个 ObjectLayer。两个 ScrollingLayer 是用于星云和星星区域，而 ObjectLayer 用于所有在游戏世界中的对象（包括分数显示）。

GameplayScene 不做太多事情，只是每帧调用更新函数。这个函数每次都会更新层。最后，加载精灵表到缓存中，而其他类则在缓存中获取相应的精灵。

ScrollingLayer

ScrollingLayer 是一个通用类，用于允许两个及以上的屏幕大小片段无限循环。为了构造 ScrollingLayer，需要传递列出所有背景精灵名字的数组以及层滚动的速度（像素/每秒）。为了简化，ScrollingLayer 假设总是有足够的内存同时加载所有背景精灵。

更新函数是这个文件的主要功能。它所做的就是移动所有背景精灵的位置。在背景精灵滚动出屏幕之后，它将会像任何其他背景精灵那样移动位置。这样，当屏幕需要显示特定图片的时候，将能够正确地滑动。

在我最初实现星云滚动层的图片切换时，图片之间出现了**缝隙**（一个像素的竖线），如图 13.2 所示。在熟读代码之后，我非常困惑，因为没有看到 ScrollingLayer 实现上有任何问题。但是我的一位技术审校指出了问题，问题出现在我使用的星云图片文件。这个问题是由图片当中的一条灰色竖线引起的。在去掉图片文件的灰线之后，星云滚动非常完美。这堂课教育我们应该多检查资源问题，确保问题不是资源引起的。

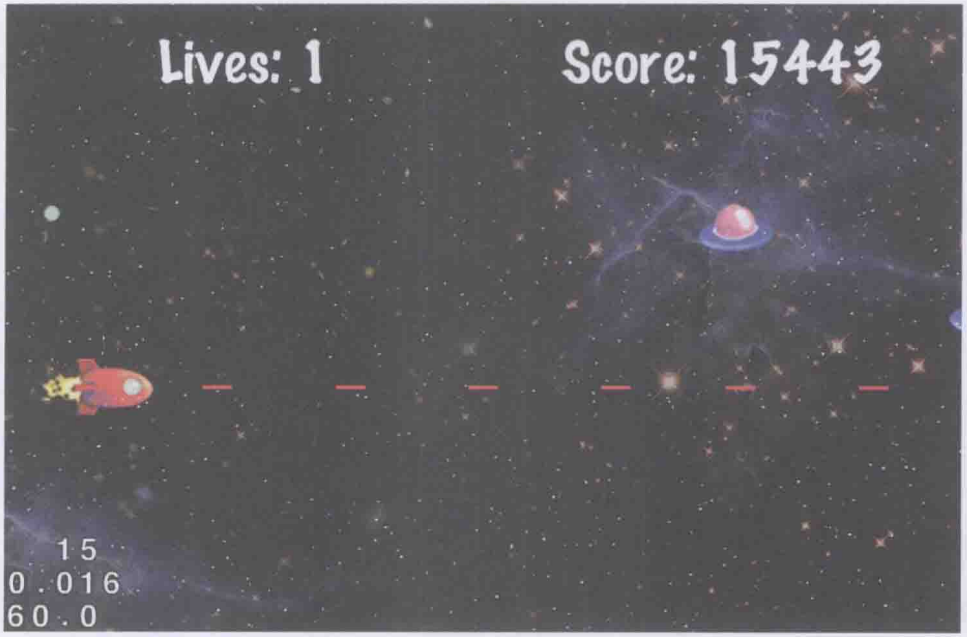


图 13.2 早期版本的 *Ship Attack* 中，滚动层会出现可见的缝隙

任何情况下，*Ship Attack* 使用了两个滚动速度不同的层级来创造平行滚屏效果（见第 2 章）。星星区域滚动比星云快，这样创造了星星区域比星云近的感觉。

Ship

游戏代码中你可能会注意到没有“GameObject”类。取而代之的是，代码中使用的是 `CCSprite`（一个 Cocos2D 的类，用于表示精灵对象），同时也是游戏对象的基类。所以在玩家船只的例子中，`Ship` 类直接继承自 `CCSprite`，而不是任何中间类。

`Ship` 在函数 `init` 中通过**动作**设置了 `Ship` 的循环动画，动作在 Cocos2D 系统中可以让 `CCNode` 做某些事情。比如说，有的动作可以运行动画，有的可以移动到特定位置等。这些动作简化了很多行为，很多命令只要组合动作就可以完成。

`Ship` 中全部 `update` 函数所做的就是让船垂直朝向目标点移动，方向由玩家的左手拇指控制。随着玩家移动拇指，目标点发生变化，这样就使得船朝目标点移动。拇指的移动肯定是连续的，因此不可能发生飞跃。但与此同时，船的速度也足够快，让玩家感到舒适，有控制感。

Projectile

Projectile 类继承自 CCSprite，同时也是船上激光和敌人子弹的基类。Projectile 的 update 函数非常简单。首先，以 Euler 积分（第 7 章讨论过）的方式更新子弹位置。然后，检查子弹是否飞出屏幕边界。一旦子弹飞出边界，会调用 despawn 函数。

有两个类继承自 Projectile：第一个是 Laser（船上激光）和第二个是 EnemyProj（敌人的子弹）。这些继承后的类只要做两件事：设置正确的精灵和重载 despawn 函数以将 ObjectLayer 上的子弹移除。

Enemy

Enemy 类是最复杂的对象类。目前，敌人支持两种不同类型的行为：8 字形（黄色的大型飞碟）和“走过”路线（正常大小的蓝色敌人使用）。两个设置这些路径的函数为 setupFigure8Route 和 setupPostRoute。

注意这两种路径，我用到了动作来设置整个路径。当 ObjectLayer 生成 Enemy 的时候，就会为敌人设置正确的路径，然后动作会指导该怎么做。在走过路径的例子中，敌人会向屏幕中间移动，然后从上面或者下面离开。在 8 字形的例子中，敌人会不断地 8 字飞行，一直到被击败。

两类敌人的发射模型也不同。走过路径的敌人只朝-x 方向发射子弹，而 8 字形的敌人会向玩家位置发射子弹。这个行为在 fire 函数中完成，其中用到了一些基础的向量数学来确定子弹的方向。

Ship 中的其他函数或多或少是一些辅助函数。一个是设置正确的动画配置，取决于船的颜色。另一个会在需要给出一组子弹飞行的时候生成一组 CCMoveTo 动作。

ObjectLayer

ObjectLayer 是主要的游戏逻辑层，你会在类中看到很多的成员变量。最主要的是，指向 Ship 的指针、玩家激光的数组、敌人以及敌人的子弹。不仅如此，还有几个参数可以控制游戏的基本行为，比如船的速度、刷新敌人的时间、敌人出现的间隔等。修改这些变量可以让游戏变样。

ObjectLayer 的更新函数用于更新所有当前活跃的游戏对象。所以它会遍历包含这些对象的数组，然后执行它们的 update 成员函数。在所有对象都被更新之后，会进行碰撞检测。使用 AABB（第 7 章讲过）进行碰撞检测，最先检查激光是否打中敌人。如果敌人被激光打中，

则激光消失且敌人损失血量。如果敌人血量为 0，就会死亡然后从游戏世界中移除。类似地，激光也与敌人的子弹发生碰撞检测，在检测出碰撞之后就销毁激光和子弹。最后更新函数还会检测敌人子弹是否与玩家的船发生碰撞，如果相撞，则损失血量。

敌人的生成是通过 spawnEnemy 函数来配置的，它会每隔 n 秒调用一次。随着玩家玩到后面，这个 n 会越来越小。每当敌人生成，会通过 arc4random 函数随机设置敌人的 y 位置，这个函数是高质量随机数的生成器。游戏还会跟踪生成了多少个敌人，每到一定数量，boss 敌人就会出现。boss 就是更大的敌人，会 8 字形飞行。

在 Cocos2D 中，可以配置层以接收触摸事件。这是通过 init 函数中最开始的 setTouchEnabled 函数来完成的。在配置之后，你可以让函数在特定的触摸事件后被调用。当玩家第一次启动触摸，会触发 ccTouchBegan 的调用。代码中的这个函数检查触摸是在屏幕的左边还是右边。屏幕左边的触摸事件会保存到 m_MoveTouch，因为这是一个尝试移动船的操作。而屏幕右边的触摸则保存在 m_FireTouch 中。

我们之所以保存这些变量，是因为触摸是可以跨过屏幕的。每次活跃触摸的位置发生改变的时候，ccMoveTouch 函数就会被调用。注意到这个函数中的代码会检查如果移动的触摸是控制船移动的，即如果 m_MoveTouch 的位置发生了改变，那就改变船的目标位置。其他的所有触摸都应该被忽略，因为我们不关心移动之外的触摸。

一旦玩家抬起手指，这个触摸就不再活跃，ccTouchEnded 会被调用。在 ObjectLayer 的例子中，它所做的就是检查 m_MoveTouch 或者 m_FireTouch 是否结束，而且也会清除触摸对象引用。

最后，还有在 ObjectLayer 中的函数用于跟踪游戏状态。这包括获得玩家杀敌得分以及存活得分的函数。如果玩家血量为 0，游戏结束，然后显示“game over”。

练习

现在可以看看代码，以下是一些可以添加到游戏中的功能。

1. **更多的敌人类型**

   只有两类敌人会让游戏显得单调。最简单的添加敌人类型的方式就是为 Enemy 类增加额外的路径函数。然后这些路径可以根据不同的条件激活。也可以通过将文件放到 Resources/sprites 下添加更多的敌人图片，然后用 TexturePacker 加载 space.tps 文件。在 TexturePacker 中，你可以点击“publish”来生成新的精灵表。

2. **增加有节奏的敌人**

   如果你玩过经典的基于每一波敌人攻击的飞行游戏，比如《1942》，你可能会想到可以设置每波不同敌人在不同时间点切入游戏。如果你不断地玩同一关，你会发现每

一波的敌人都一样。这比起 *Ship Attack* 中用到的方法更加有趣，我们的游戏中敌人只是通过计时器触发。

为了让每一波都有结构，你会想要添加一些类来管理每波敌人的生成。你还会需要一些方式来定义敌人生成的时机。一种方法就是有着所有敌人的生成清单，实际上就是记录出现的时机、敌人的类型、生成的位置。接着生成管理器可以随着时间流逝生成合适的敌人。

总结

我们关于 *Ship Attack* 的讨论，应该能够让你明白如何实现一款简单又不失有趣的 2D 横向卷轴游戏。因为是一款 2D 游戏，用到了很多第 2 章讲到的技术。而讨论的内容因为用到了 Euler 积分和 AABB 碰撞检测，所以也与物理有一点关系（第 7 章）。通过这个游戏的基础构建块，完全可能创建更加复杂和有趣的版本。

第14章

游戏示例：塔防（PC/Mac）

塔防类型游戏已经流行了很多年，但是它的流行是由游戏自制地图开始，比如《星际争霸》和《魔兽争霸3》。正是因为这种游戏的流行，今天塔防游戏几乎在所有游戏平台上都有。

本章看看一个游戏例子，这是一个未来塔防游戏，叫作 *__Defense*。这是一款完全 3D 渲染的游戏，尽管游戏玩法上是在 2D 平面上进行的。这个游戏用 C# 语言开发，基于 XNA/MonoGame 框架。

概览

在你阅读本章之前，应该先访问本书的网站，下载该游戏的源码。源码的链接会指引你如何运行游戏，地址为*http://gamealgorithms.net/source-code/defense/*。在你让游戏运行起来以及玩起来之后，就可以继续往下阅读了。但是在开始源码分析之前，先对一些 *__Defense*（如图 14.1 所示）中用到的不同技术进行简短的介绍。

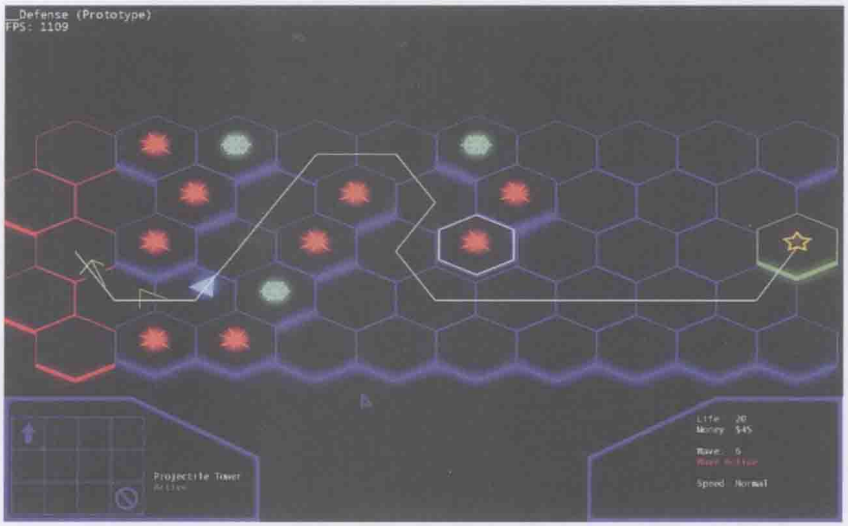


图 14.1 *__Defense* 游戏中

C#

微软所开发的 C#（读作 C sharp）是为了拥有一门类似于 C++/Java 的语言，但是又能基于.NET 框架快速开发 Windows 应用。与 Java 一样，C# 不是直接编译为本地代码，而是编译为字节码，这样就可以在虚拟机上面运行。最开始，只有一个在 PC 上跑的虚拟机，而现在在各个平台上都有可运行 C# 的开源的虚拟机。

虽然最开始 C# 和 Java 很像，但是过去这么长时间以来，语言发生了非常大的变化。但是因为语法上与 C++ 和 Java 类似，所以对于熟悉这两门语言其中之一的读者来说还是相对容易学习的。现在，C# 已经是一门快速开发 Windows GUI 应用的优秀语言。也正因为如此及其他一些原因，C# 在开发游戏工具上非常流行——有些引擎，比如新版本的 Frostbite（用于《战地》系列）在工具链中大量使用了 C#。

C# 中大部分语言结构对于 C++ 程序员来说是非常熟悉的，但是还是有一些部分需要讲一下。首先，C# 通常不需要管理内存。所有基本类型和 struct 总是在栈中分配，而所有类总

会在堆中分配。由于语言有垃圾回收机制，所以不用担心内存泄漏——只要对象不再引用，它就会在未来的某个点被释放。C# 有一个“unmanaged”模式，允许进行低级操作，但是如果不是非常特殊的原因，不推荐使用。

这款游戏代码中用到了这门语言中一个独特的特性。一般来讲，类封装原则下，类中的所有数据应该都是私有的，只有通过 getter 和 setter 函数才能访问。因此在 C++ 中，一个 2D 向量类可能实现如下：

```
class Vector2
{
public:
   int GetX() { return x; }
   void SetX(int value) { x = value; }
   int GetY() { return y; }
   void SetY(int value) { y = value; }
private:
   int x,y;
};
```

然后如果你有 Vector2 的实例，为了设置 x 值为 10，可能会调用 SetX 函数如下：

```
Vector2 v;
v.SetX(10);
```

getter 和 setter 函数的麻烦之处在于它们非常啰唆。为了解决这个问题，C# 增加了**属性**，就是基本 getter 和 setter 函数，但是在使用的时候它们的调用是隐藏的。访问属性就像访问变量一样，但是也可以创建 getter/setter 函数。因此上面的 C# 中的类可以像这样表达：

```
class Vector2
{
   private int x, y;
   public int X
   {
      get { return x; }
      set { x = value; }
   }
   // Y属性也一样
   ...
}
```

而使用 Vector2 的代码可以像这样：

```
Vector2 v = new Vector();
v.X = 10; // 自动调用X的属性
```

属性的优雅之处在于使用了一般变量的语法，尽管实际上不是一个变量。C# 中还有不少独特的其他特性，但是它们很多都没有在这款游戏代码中出现。在阅读代码时，如果发现自己阅读 C# 很吃力，可以在网上找到很多资料，其中有很多是面向有 C++ 或者 Java 背景的工程师写的。

XNA

XNA 是 C# 的 2D、3D 游戏库，由微软开发，用于开发 PC、Xbox 360、Windows Phone 7 上的游戏。PC 开发是完全免费的，但是给 Xbox 360 开发游戏，需要付 100 美金/年的订阅费用（学生可以通过*http://www.dreamspark.com/*得到 1 年免费使用期）。但是，付费可以让你的游戏在 Xbox Live 独立游戏商店中发布。XNA 框架上有不少成功的商业案例。最有名的独立游戏是 *Bastion*，是 2011 年发布于 Xbox 360 平台上的游戏。

我个人认为 XNA 是一款很好的学习游戏编程的框架（特别是 3D）。它处理了很多游戏编程中乏味的部分——比如创建窗口、初始化图形库，而最重要的是加载资源文件。在 XNA 中，只要少量代码就可以加载 3D 模型、初始化摄像机/投影矩阵、渲染模型。这是因为 XNA 内建支持了很多多媒体文件的加载，包括纹理、模型、音频及多种视频格式等。

虽然使用完整的引擎来开发 3D 游戏原型会更加容易，比如 Unity 或者 UDK，但使用这些引擎的问题在于，它们为你做了大部分繁重的编程任务。比如说，如果你想在这些引擎中使用寻路，可以使用内建的寻路，而不需要再次实现 A*。但是在 XNA 中，如果你想使用寻路，必须自己实现，这意味着你才会将算法理解得更加深刻。

不幸的是，微软决定放弃 XNA，而且不再为框架更新。虽然 XNA 被微软抛弃了，但是在 MonoGame 项目中还存活着。

MonoGame

MonoGame 是一款开源且跨平台实现的 XNA 框架。它使用了与 XNA 一模一样的类和命名空间，因此将 XNA 游戏转为 MonoGame 还是挺容易的。在 *__Defense* 的例子中，大概花 15 分钟就能让 XNA 项目运行在 MonoGame 上。MonoGame 最早的两个版本只支持 2D 游戏，但是到了 3.0 版本（2013 年 3 月发布），框架就支持了大部分 3D 功能。

MonoGame 最为强大的地方在于它的跨平台方式，在很多平台上都能跑。在本书写作的时候，它可以运行在 Windows、Windows 8 Metro、Mac、Linux、iOS、Android（但是移动版本不是免费的）上。之前提到的 *Bastion* 就是从 XNA 移植到了 MonoGame，这也是它能够在这么多平台上跑的原因，包括 Mac、Linux、iOS，甚至 Chrome Web Store。在未来，通过 MonoGame 甚至能开发 Xbox One 的游戏，这取决于微软怎么处理开发者证书。

MonoGame 没有完全实现 XNA 的一个重要方面就是资源的格式转换上。所以为了准备 MonoGame 中用到的资源，必须通过 XNA 构建它们，然后复制这些文件到 MonoGame 项目合适的目录下。但是 MonoGame 开发者在 MonoGame 内容管线上非常活跃，所以到你读到这里的时候，这应该不再是个问题。但是目前而言，你想在游戏中使用任何模型和纹理，必须先在 XNA 解决方案中构建资源。

代码分析

比起第 13 章的游戏示例，*__Defense* 会有更多的代码需要浏览。这也是很合理的，因为这款游戏涉及的机制和界面都比 *Ship Attack* 要复杂得多。如果你试着直接阅读代码而不看本节，也许会很难理解它们全部。最好的方法是在阅读本节的同时也去理解代码。

设置

通常，如果设置保存在外部文件的话，这是最好的（就像在第 11 章中所说的）。由于游戏相对简单，感觉上编写代码解析设置文件又没有必要。但是，我不想将所有参数都到处硬编码，所以我将它们中的大多数都合并到了 3 份源文件中。

Balance.cs 有着所有能够影响游戏平衡的参数。这些参数，比如敌人的波数、敌人的个数、敌人的生命值以及调整游戏的难度，都存储在 Balance.cs 中。任何不影响游戏平衡的设置（比如游戏是否全屏以及摄像机速度）放在 GlobalDefines.cs 中。最后，DebugDefines.cs 是几乎没有用的，主要存放与调试游戏相关的参数。

单件

单件是一个可全局访问的类，而且整个程序只有一个实例。这是最常用的设计模式之一，同时也是新引擎中实现的第一个类。虽然可以通过全局类来实现单件，但这不是最好的方法。对于这款游戏的代码来说，单件的实现在 Patterns/Singleton.cs 中。

`Singleton` 的工作方式就是一个拥有模板类型静态实例的模板类，以及一个静态 `Get` 函数可以返回这个静态实例。听上去有点复杂，但是为了最终能够使用单件，所有用到的都需要继承自 `Singleton`，像这样：

```
public class Localization : Patterns.Singleton<Localization>
```

让 `Localization` 类作为单件可以全局访问，只要调用 `Get` 函数就可以，如下：

```
Localization.Get().Text("ui_victory");
```

这款游戏中很多类都是单件，包括 GraphicsManager、InputManager、PhysicsManager、SoundManager 以及 GameState。看上去摄像机也是一个很好的单件候选人。但是，摄像机不是单件，因为如果游戏可以分屏，就会有多个摄像机。为了试图使代码保持流畅，避免这样设定的话，这个想法也不错。

游戏类

Game1.cs 有着主要的游戏类，但是它没有太多的代码。在游戏第一次加载的时候会调用 Initialize，并且实例化一些单件。每帧都会调用 Update 和 Draw 函数，而且它只会在合适的单件上调用 Update 和 Draw。所以总的来说，这个文件没有太多东西。

在以下 Update 函数的代码片段中有一个值得注意的小细节：

```
if (fDeltaTime > 0.1f)
{
   fDeltaTime = 0.1f;
}
```

这使得最低帧率为 10FPS。这么做的原因是如果游戏被调试器暂停了，在恢复的时候有可能过去了很多秒。通过限制最低 FPS 为 10，确保不会有什么时间太长使得游戏行为变得不稳定。

游戏状态

游戏的整体状态由 GameState（在 GameState.cs 中）控制，这也是为什么它是游戏中最大的文件。它是一个状态机，但是因为主菜单和暂停状态行为简单，所以 GameState 中的所有代码几乎都是控制游戏逻辑的。因为某个特定状态中用到特别多的代码，所以我没有使用状态设计模式（像第 9 章那样）。但是如果一个特定游戏有很多不同的状态（比如不同的游戏模式），用设计模式重构是个不错的想法。

任何与游戏中状态相关的东西，比如金钱的数量、血量、下一波敌人的时间等，都存放在 GameState 中。所以 GameState 的成员变量大多数与游戏相关，但是也有一些不是。一个是游戏世界中关于所有游戏活跃对象的链表。所有新游戏对象都要通过 GameState 中的 SpawnGameObject 函数，这样就加入到了游戏世界。然后，在 UpdateGameplay 中，在游戏没有暂停的时候，会更新所有游戏对象。

UpdateGameplay 函数中还有一些内容值得讲一下。首先，实际上它没有迭代所有游戏对象链表，而是在游戏对象链表备份上进行。这就使得在更新的时候可能需要移除游戏对象（包括移除自己）。在 C# 中，foreach 不允许在循环中修改容器，这也是为什么需要备份。

虽然看起来非常没有效率，但是要知道 C# 中的类总是通过引用传递的。所以复制链表只是一个指针数据的浅拷贝，没有太多的计算量。其他要注意的是，UpdateGameplay 接受的间隔时间并不总是传递给所有游戏对象的时间间隔。这样就使得玩家可以通过 +/- 键来调整游戏速度。

GameState 还有一个 Camera 类的实例。这个类中另一个重要的成员变量是 UI 栈，它是当前活跃用户界面屏幕的栈。UI 系统会在本章稍后详细介绍。

游戏对象

基本的游戏对象类是在 Objects/GameObject.cs 中。在 *__Defense* 中，所有游戏对象都是既可绘制又可更新的，因此没有为不同的游戏对象类型做复杂的继承层级。世界变换矩阵用了一个向量表示位置、浮点表示缩放、四元数表示旋转，就像第 4 章讨论的那样。所有游戏对象都有包围球，而任何想使用 AABB（比如砖块）的游戏对象都可以在构造函数中设置 m_bUseAABB 布尔值为真。

Tower 类（Objects/Tower.cs）继承自 GameObject，而且有着所有防御塔的行为。最值得注意的是，它有着建造和升级防御塔的函数，会根据防御塔是否激活而改变纹理。因为目前游戏有两类防御塔，所以理所当然的就有两个 Tower 派生类：ProjectileTower 和 SlowTower。这些防御塔的自定义行为或多或少在 Update 函数中实现。

Tile 类继承自 GameObject，实际上没有太多功能——它只是提供让 Tower 附加到 Tile 的能力。让 Tile 作为一个单独的游戏对象看上去太重，但是这样做是为了让 Tile 可以动态改变高度。虽然 Tile 不是游戏对象也是可能的，但是让它们作为 GameObject 子类显然是更加简洁的实现。Tile 的子类（TileRed 和 TileGreen）与基类 Tile 功能上有所不同，但也只是小部分不同（除了红色砖块不能盖建筑）。

Projectile 是由 ProjectileTower 生成的，而它们所做的就是向敌人发射子弹，让它承受相应的伤害。子弹移动通过线性插值来完成（在第 3 章讨论过），然后就会慢慢地到达敌人的位置。最初，子弹和敌人的碰撞检测是使用连续包围球做检测（见第 7 章）。但是问题在于越高级的敌人包围球越大，导致子弹几乎一出现就消失。所以为了让游戏更具挑战性，改变代码以忽略敌人的大小，而子弹依据距离敌人中心点距离来制造伤害。

两种防御塔都会遍历游戏世界中的所有敌人。在 ProjectileTower 的例子中，它会先找到最近的敌人，而 SlowTower 则是找到所有在射程内的敌人。两个函数都使用暴力算法来确定攻击哪个敌人。一个解决这个问题的方法就是使用分区算法，比如四叉树，在第 7 章有所提及。但是实现该算法超出了本书的范围，所以为了简单起见，代码还是使用暴力算法。但是，如果 *__Defense* 是一款商业游戏，我们应该为了性能原因消除所有暴力算法。

Enemy 游戏对象有一些自定义行为允许在游戏世界中按给定路径移动。它们会根据 Pathfinder 寻路生成的路点链表来行走。在 Update 函数中，它使用线性插值在两点之间移动，一旦到达一个节点，就会继续走路径的下一个节点。敌人的朝向是通过给 SetDirection 函数设置四元数来完成的，还有一些函数，可以让敌人被捕捉（如图 14.2 所示）或者杀死。

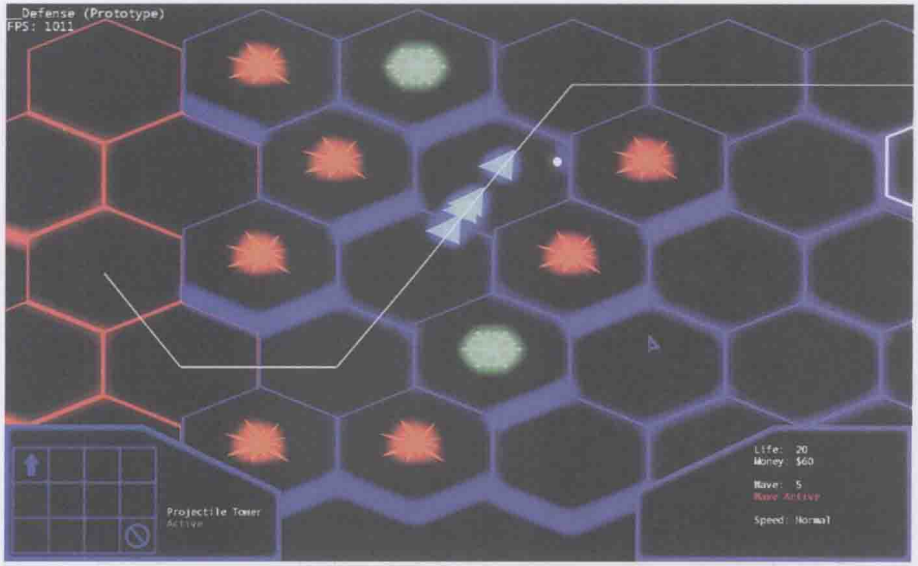


图 14.2　敌人被捕捉之后变成蓝色

关卡

关卡数据完全在 Level.cs 中处理。目前，关卡布局是通过在 LoadLevel 函数中硬编码加载 TileData 数组来完成的。这肯定不是理想的方式，但是对于这款游戏来说从文件加载数据是没必要的。LoadLevel 函数剩下部分基本上就是循环遍历数组并在正确的地方摆放砖块。因为砖块是六边形的，所以计算中心位置会稍微比正方形砖块复杂一些。还有一个函数叫 Intersects，我们稍后在讨论游戏物理部分的时候会详细说明。

计时器

Timer 类（Utils/Timer.cs）是一个工具类，它能够让注册了的函数在一定时间后被调用。因为每个游戏对象都有一个 Timer 类的实例，所以销毁行为就比较简单。比起增加很多浮点变量以跟踪剩余时间，调用特定函数可以简单地添加一个计时器如下：

```
m_Timer.AddTimer("HideError", duration, ClearErrorMessage, false);
```

AddTimer 函数的第 1 个参数是为计时器唯一名称，第 2 个是计时器会在多久之后调用，第 3 个是被调用函数的名字，而最后一个参数表示计时器是否循环。例如，可以通过 Timer 类将某个函数设定为每秒调用一次。

寻路

寻路（Pathfinder.cs）实现上使用的是 A* 算法（第 9 章讨论过）。因为敌人总是从相同的位置出现，因此寻路代码没必要每次都为每个敌人计算路径。相反，Pathfinder 有一条全局路径，作为新敌人的优化路径存储。游戏中这条优化路径以白色线段显示。

在塔防游戏中，不允许玩家完全阻挡敌人行走路径是很必要的。如果敌人无路可走，敌人只能够停留在那里，而游戏难度则变得很低。为了解决这个问题，每次新的防御塔建造之前，GameState 都会通过 Pathfinder 检查确保新的防御塔不会导致路径不可走。如果路径变得不可走，建造就无法进行，然后游戏为玩家显示相应的错误提示。如果路径可走，游戏会建造防御塔，然后设置新的全局路径。图 14.3 显示了防御塔的放置位置非法，因为它阻挡了路径。

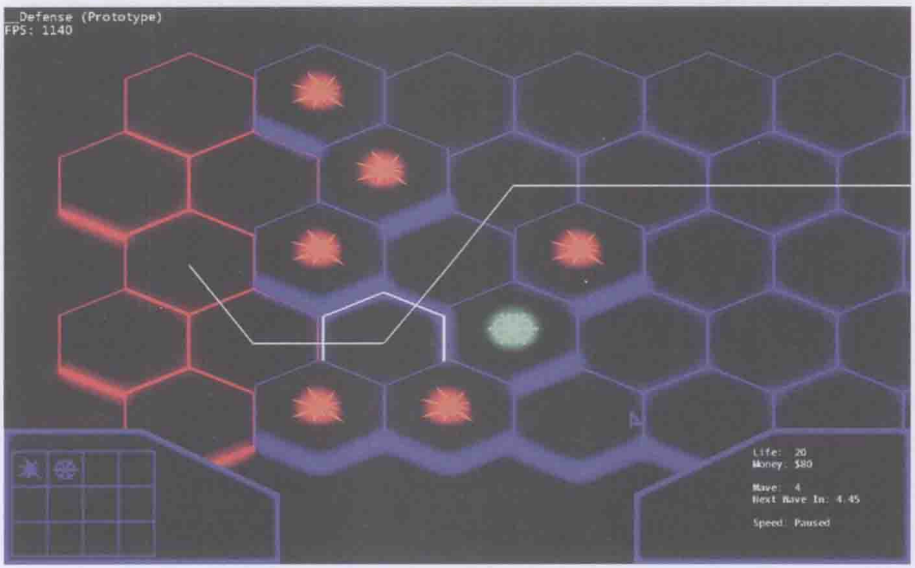


图 14.3 游戏不会允许在选中的砖块（白色勾边）上建造防御塔，因为它阻挡了路径

如果防御塔位置合法，全局路径就会变化。但是这个新防御塔可能会导致当时活跃敌人的路径变为无效。为了修复这个问题，ComputeAStar 函数会为每个单独的敌人调用一次。这样全局路径因为新防御塔而改变，任何活跃敌人都会重新计算一下他们新的最佳路径。敌人被建造的防御塔完全包围也有可能，但是对于整个游戏来说不是很重要，因此系统并不

关心。如果你将敌人圈起来，又无法攻击，那么游戏可能进入无法胜利的状态。但是玩家很难才会遇到。

如果仔细查看 ComputeAStar，你会注意到它会搜索从目标点到起点的路径。这是因为如果用其他的方式计算路径，会得到从终点到起点的链表，但这不是敌人需要的。所以需要反转路径，而不是反向计算。

摄像机和投影

游戏使用了正交投影，这种投影在第 4 章讲过。这意味着深度会没有意义，因为距离摄像机远近并不会改变大小。因此游戏中的摄像机（Camera.cs）就相当简单。除了在这个层级上放置好摄像机之外，它没有任何功能。为了放置好，摄像机的镜头位置和目标位置要以同样的量来改变，通过 AddToPan 函数传递给摄像机。摄像机还可以使用鼠标滚轮缩小放大。最后，还有一些代码用来做将摄像机移动到某个特定位置的动画，但是目前没有使用。

为了让玩家点击和选择砖块，游戏中还有用到鼠标拣选（第 8 章讲过）。尽管拣选是摄像机相关算法，但是相关代码在 InputManager.CalculateMouseRay 中。为了执行拣选，需要同时知道鼠标在屏幕上的位置以及摄像机矩阵。因此方案无非将鼠标位置传递给摄像机或者让输入管理器访问摄像机矩阵。因为绘制代码也需要用到摄像机矩阵，而且摄像机矩阵已经可以通过 GameState 全局访问，所以拣选代码写在 InputManager 中。

输入

所有与键盘和鼠标相关的输入都在 InputManager.cs 中。键盘绑定系统与第 5 章讨论的“更复杂的系统”类似。在 InitializeBindings 函数中，将抽象的动作，比如 Pan_Left，绑定到某个特定的按键以及按键的某个特定状态（比如刚刚按下、刚刚释放、一直按着）。

然后每一帧，UpdateKeyboard 函数都会被调用，然后就会使得在某一特定帧上活跃的所有动作组成一条链表。这个链表最开始传递给 UI，UI 就会处理所有它关心的行为。剩下的动作就会传递给 GameState 来处理。现在，GameState 处理了摄像机放置相关的动作，因为它访问了 Camera 类。

鼠标是与键盘处理分开的（在 UpdateMouse 中），但是处理方式是一样的。鼠标点击事件会先传递给 UI（看看按钮是否被点击），如果 UI 不关心这个点击，就会传递给 GameState。GameState 检查鼠标点击，然后检查恰当的 Tile 是否被选中。因为可能用鼠标放置摄像机（而不是用键盘），UpdateMouse 也会检查鼠标是否到屏幕边缘，这表示放置必须发生。

物理

游戏中的物理主要用于碰撞检测。因此，PhysicsManager 所做的是在模型坐标系下对特定模型生成包围球或者包围盒。因为模型坐标系下的包围数据对于同一个模型而言是不会变化的，所以可以缓存于排序列表。故每次需要某个特定网格（比如“Miner”），就可以检查是否之前已经生成过包围球或者包围盒。这样，每次敌人生成，就没必要重新计算包围数据。

而游戏中的碰撞检测，主要用于检测拣选投射出的光线是否打中砖块。XNA 提供了内建的 Ray、BoundingSphere 和 BoundingBox（AABB）类。所有这些类都可以检测相互之间的交叉，因此我不需要实现任何第 7 章讲到的交叉测试。拣选的光线与砖块之间的交叉出现在 Level.Intersects 函数中。这个函数会创建与光线交叉的所有砖块的链表，然后选择与光线最开始位置最近的砖块。

本地化

就像第 10 章所说的，将显示在屏幕上的文本字符串硬编码是不好的习惯，因为将游戏翻译为其他语言会非常困难。对于这款游戏来说，所有在屏幕上显示的文本字符串（除了调试用的 FPS）都存储在 Languages/en_us.xml 文件中。如果你打开文件，你可以看到里面格式相对简单，每项数据都存储在 <text> 标签中，像这样：

```
<text id='ui_err_money'>You don't have enough money.</text>
```

然后在代码中就可以通过 ID 获取字符串：

```
Localization.Get().Text("ui_err_money")
```

Localization 类的代码不是很复杂。加载的时候，解析 XML 文件，以键值对存储成字典。Text 函数会在字典中进行查找。有了这个系统，游戏翻译成其他语言就相对容易，只要改变 XML 文件就可以了。

图形

GraphicsManager 主要负责大部分的渲染逻辑。在第一次初始化的时候，如果游戏设置为全屏模式，它会检查当前桌面的分辨率，然后设置游戏的分辨率。否则，它会使用 GlobalDefines 中定义的分辨率。但是 GraphicsManager 类中最核心的功能就是处理各种游戏对象的渲染。

游戏对象可以在它的构造函数中选择所进入的绘制顺序组。有 3 种可选：背景、默认、前景。每个绘制顺序组都在 GraphicsManager 中有一条链表，所以游戏对象在添加到游戏世界

的时候，会加载到合适的链表中。不同组的目的在于让某些对象总是在默认组（几乎是所有对象）中对象的前面或者后面。

绘制通常都会使用深度缓冲，如果还记得第 4 章所讲过的深度缓冲的作用，那么就是它只绘制那些深度比它低的像素。但是，GraphicsManager 的绘制函数只为默认组打开深度缓冲。首先深度缓冲关闭，然后开始绘制背景组。该组可以用于**天空盒**，例如天空盒就是一个立方体，包围着整个世界，上面贴着天空（或者空间）的纹理。然后打开深度缓冲，之后绘制默认组，里面有着游戏大部分的游戏对象。最后，再一次关闭深度缓冲，绘制前景组对象。该组可以用来绘制像第一人称游戏模型的对象，而不用担心被其他对象裁剪。

GraphicsManager 中还有一些其他功能，但不是特别重要。你可以找到一些小的辅助函数，用来绘制如 2D 和 3D 线条，或者绘制填充矩形。最后，XNA 版本的 bloom 效果可以使砖块发光。这个效果使用了微软例子（http://xbox.create.msdn.com/en-US/education/catalog/sample/bloom）中的实现。但 bloom 效果的细节以及多渲染目标超出本书的范围，它不是特别重要，因为只是一个视觉增强。

声音

因为这款游戏只有少量 2D 音效，所以 SoundManager 类非常简单。它所做的就是加载 WAV 文件成为字典，字典的键就是文件名，值就是 wave 文件。然后就可以通过 PlaySoundCue 函数播放音乐了：

```
SoundManager.Get().PlaySoundCue("Alarm");
```

这个实现并不难，但是最值得关注的是，声音在游戏暂停的时候并不会暂停。因此，你一定不会想将这个样子的声音系统的游戏进行公测。它只能在很基础的层面上顺利工作。

用户界面

这款游戏中的 UI 系统相当复杂，最主要是因为提示系统。但在我们阅读提示代码之前，先看看 UI 系统的基本布局。UIScreen 类（UI/UIScreen.cs）表示某个菜单或者 HUD。举例来讲，UIMainMenu 中的主菜单就是一个 UIScreen。这个系统支持同时显示多个 UIScreen，通过在 GameState 中的 UI 栈来处理。举个例子，UIGameplay 和 UIPauseMenu 界面在游戏暂停时同时可见。只有栈中的顶层界面才可以接收输入，而次一层则完全无法接收输入。

UIScreen 可以有任意多个按钮（UI/Button.cs）。每一帧，活跃 UIScreen 可以接收光标位置，然后设置相应的按钮高光。如果点击发生，就会调用按钮上注册了的函数进行回调。按钮可以以 3 种不同的方式初始化：只有文本、只有图片或者提示区域（只提示，但不可点击）。对于只有文本和只有图片的按钮来说，构造函数需要给定回调函数，用于按钮被点击

的时候调用。目前，菜单系统不支持键盘导航，因为它被设计只支持 PC 上的鼠标，但是这个功能要加上也不难。

虽然 UIScreen 系统不是很复杂，但是提示系统复杂不少。原因有几个。首先，默认 XNA 的文本渲染不能指定宽度。给定一个字符串，它只是简单绘制字符串直到文本结束。对于提示来讲，根据特定的最大宽度包围起来是很重要的。第二点，我希望提示信息中，个别文本字符串的字体和颜色可以改变，这样可以让提示信息突出重点。这也是 XNA 默认不提供的。为了解决这两个问题，我最后使用非常有用的 MarkupTextEngine，它是随着微软授权发布（*http://astroboid.com/2011/06/markup-text-rendering-in-xna.html*）的“astroboid”附带的。它在解决上面两个问题上都工作得非常好，虽然代码可能有点难懂。

提示系统的复杂度还没完，我还想要动态改变提示中的文本。我不希望提示文本总是固定的（比如花费 $500），因为 Balance.cs 中的值经常变化，提示文本也要随着改变。相反，我想用真实的值替换提示信息中的占位符。所以比起“花费 $500”，提示字符串应该是“花费 ${0}”，而 {0} 会被正确的值所替换。还有更复杂的是，我想颜色能够根据状况来变化。如果建构花费 $500，但你没有 $500，我希望这里的 500 可以标记成红色，来提示玩家金钱不足以购买建筑。最终的提示系统如图 14.4 所示。

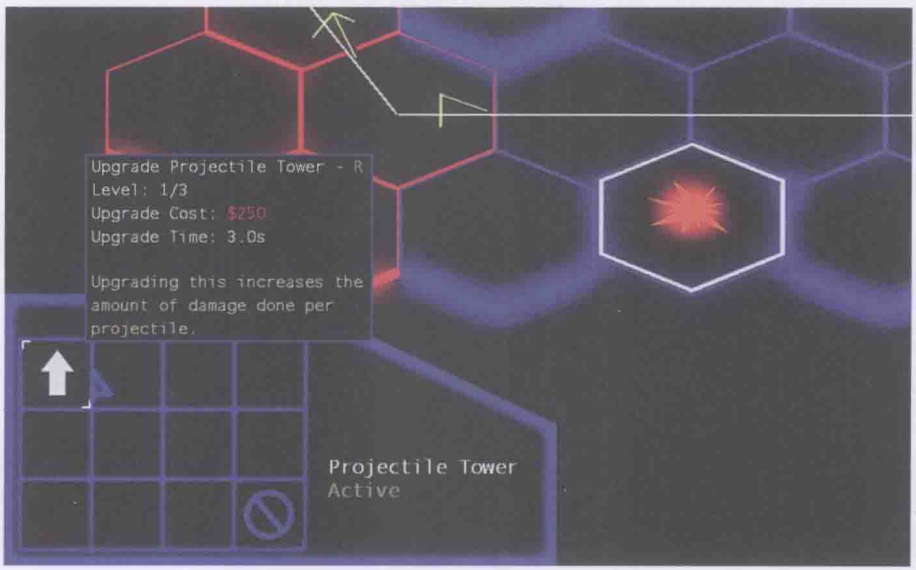


图 14.4 提示信息

这就是 Tooltip 类的功能。想要初始化提示信息，需要提供本地化字符串的键值，以左下角为提示信息中心的屏幕位置坐标，以及 TipData 链表。TipData 链表告诉 Tooltip 类一些占位符要用什么值替换的准备信息。举个例子，之前的 {0} 是指防御塔的花费。这个信息由 CreateTowerBuildCost 以及类似的函数生成。

UIGameplay 中的 InitializeGrid 函数稍微有点复杂。其中的 grid 就是 HUD 中左下角的一系列按钮，这些按钮可以让你选择建造不同的防御塔。因为有多个 grid 按钮，还有一些 grid 按钮上有复杂的提示，创建提示信息的函数会比较长。如果游戏有非常多的 grid 按钮，手动初始化是非常不好的。最好能够存储在外部数据文件中。但是对于这款游戏来说，手动实现会更加简单。

练习

现在我们审阅了 *__Defense* 的代码，以下是一些能够提升游戏的想法。

1. **为游戏增加更多的波数**

   第一个想法就是为游戏增加波数让游戏变得更有挑战性。所有关系到游戏平衡的数据都在 Balance.cs 中。关于难度以及波数存储在 Waves 数组中。增加波数要做的只是给数组添加元素以及增加 TotalWaves 变量。要改变难度或者敌人数量，可以修改 Enemies 数组。

2. **增加新的防御塔**

   增加一个新的闪电塔，可以将多个目标串起来，伤害会在每次跳跃中递减。为了让新防御塔能在基础层级工作，需要修改多个文件。首先需要添加新的 LightningTower 类，然后在 eTowerType 枚举中增加一项。连锁效果需要找到范围内最近的敌人（就像 ProjectileTower 一样），但是只要找到第一个敌人，然后就需要继续找到连锁中的下一个敌人。然后 GameState.TryBuild 需要增加新类型防御塔的支持。最后，UIGameplay 中的 grid 需要更新，这样就可以创建新的防御塔（键盘绑定也是一个不错的想法）。

   在新防御塔功能完成之后，还有一些事能让事情变得漂亮。可以在 en_us.xml 中增加新的自定义提示文本，以及创建一些新的纹理。可以看 XNAContent/Buildings 以及 XNAContent/UI 以获得关于纹理的想法。还需要在 LightingTower 和 UIGameplay 中分别引用这些新纹理。记住如果你要创建新纹理，只有 XNA 项目才能构建资源。所以如果你要在 MonoGame 项目中使用，必须先构建 XNA 版本。

3. **创建外部的平衡数据文件**

   虽然 Balance.cs 文件可以工作，但是如果所有数据都存储在外部文件会更好。创建 JSON 文件存储所有平衡信息，然后在游戏加载的时候加载文件。为了序列化/反序列化 JSON，C# 中最流行的库就是 JSON.NET。在以上内容能够工作之后，还可以将更多数据拆分为几个 JSON 文件（比如关卡和按钮）。

总结

本章的例子是之前几章所提到的技术和算法的实践。实现了游戏对象模型（第 1 章），以及对象通过平移、缩放、旋转构造世界变换矩阵（第 4 章）。对象使用 AABB（第 7 章）而且与光线投射做碰撞检测（第 8 章）。A* 寻路用于决定敌人行走到基地的路径（第 9 章）。最后，*__Defense* 中实现的输入和 UI 系统分别在第 5 章和第 10 章都有讨论。希望本章所讲的能够让你明白如何开发一款复杂的游戏，现在你已经完成了本书所有章节。

附录A 习题答案

该附录为你提供了第 1~12 章章末习题的答案。

第 1 章：游戏编程概览

1. 早期控制台使用汇编语言的原因是，它的内存占用非常小和运行性能非常高。高级语言则会加入很多不必要的机制在里面，特别是在当时编译器优化还没有今天做得那么好的早期。而且，早期控制台并没有开发工具以支持高级语言，尽管这是可行的。
2. 中间件就是第三方代码库，用于解决游戏中的特定问题。比如，Havok 就是物理库，很多游戏公司都使用它而不是实现自己的物理系统。
3. 这个问题有很多种可能答案，以下是其中之一。
   在《大蜜蜂》游戏中，有两种输入要处理：摇杆控制玩家的船只左右移动以及按下发射按钮发射炮弹。在“update world”阶段的主要工作是，游戏循环要为进入的敌军船队生成并模仿 AI。游戏还需要在敌人的子弹和玩家之间做碰撞检测，同样地，玩家的子弹和敌人之间也要做碰撞检测。游戏需要跟踪发起了多少波敌人并相应地增加难度。而游戏的输出则只有视频和音频。
4. 传统游戏循环的输出还可能会有声音、力回馈以及网络数据。
5. 如果渲染大概使用了 30ms，游戏世界的更新用了 20 毫秒，传统的游戏循环会花费每帧 50ms。但是，如果渲染放入一条自己单独的线程，那么就可以和游戏更新并行完成。在这种情况下，每帧时间总花费会降至 30 毫秒。
6. 输入延迟就是你按下按钮到看见屏幕出现按下按钮产生效果的时间。在多线程的游戏循环中，输入延迟会增加。因为渲染总是比游戏更新慢一帧，因此多了一帧的延迟。
7. 真实时间就是在现实世界中流逝的时间，而游戏时间则是在游戏世界流逝的时间。有很多种方式导致游戏时间与真实时间分离——比如“子弹时间”能看到子弹的运动轨迹，这就是游戏时间比真实时间要慢的例子。
8. 以下代码这么写可以不依赖帧率：

```
position.x += 90 * deltaTime
position.y += 210 * deltaTime
```

9. 为了强制 30 FPS 的帧率，每次游戏循环结束，检查这次循环用了多少时间。如果比 33ms 要短，那么等待到 33ms 才开始下一循环开始。如果流逝的时间比 33ms 要长，那么为了赶上帧率，下一帧要跳过渲染过程。
10. 3 种主要的游戏对象类型分别是既绘制又更新的对象、只绘制不更新的对象、不绘制只更新的对象。游戏中既绘制又更新的对象是最常见的，包括玩家、角色、敌人等。只更新的对象包括摄像机和看不见的触发器。只绘制的对象包括静态网格，比如背景中的树木。

第 2 章：2D 图形

1. 电子枪瞄准左上角，然后绘制一行扫描线。之后对准下一行左端再绘制一行，重复这个过程，直到所有扫描线绘制完。场消隐期就是电子枪从右下角移动到左上角所花费的时间。
2. 屏幕撕裂会在电子枪绘制过程中因更新图形数据而引起。避免这个问题的最佳方法就是使用双缓冲技术，在场消隐期交换缓冲区。
3. 如果使用单个颜色缓冲区，那么如果要阻止屏幕撕裂，所有渲染需要在相对较短的场消隐期完成。但是通过双缓冲技术，场消隐期的工作只是交换缓冲区，那么就有充足的时间去渲染场景。
4. 画家算法就是从后往前渲染场景的渲染技术。这就跟画家在画布上绘画类似。这个算法在 2D 场景中工作相对良好。
5. 精灵表单能够节省大量的内存，因为它比单个的精灵文件能让精灵之间排布更加紧密。而且这种方法还有性能优势，因为硬件不需要频繁地切换纹理。
6. 在单方向滚屏游戏中，摄像机需要在玩家越过半屏的时候更新位置。
7. 最好是一个双链表结构，因为它能拿到前一个背景片段和后一个背景片段。
8. 一个砖块集合包含了所有可能用到的砖块图片。每个砖块在砖块集合中都有对应的 ID。另外，砖块地图则列出了特定场景或关卡砖块布局的 ID。
9. 通过存储动画 FPS 为成员变量，使得在播放中能够改变动画速度。比如，当一个角色获得加速的时候，希望动画播放得更快一些。
10. 斜视等视角在一个不同的角度去观察游戏世界，使得场景拥有更强的深度感。

第 3 章：游戏中的线性代数

1. 解：

   (a) $\langle 5, 9, 13 \rangle$

   (b) $\langle 4, 8, 12 \rangle$

   (c)

$$\vec{c} = \vec{a} \times \vec{b}$$

$$c_x = a_y b_z - a_z b_y = 4 \cdot 7 - 6 \cdot 5 = -2$$

$$c_y = a_z b_x - a_z b_z = 6 \cdot 3 - 2 \cdot 7 = 4$$

$$c_z = a_x b_y - a_y b_x = 2 \cdot 5 - 4 \cdot 3 = -2$$

$$\vec{c} = \langle -2, 4, -2 \rangle$$

2. 如果你只关心向量的方向，就很值得归一化你的向量。如果你需要向量方向和大小，就不应该归一化。
3. $\|A - P\|^2 < \|B - P\|^2$
4. 将 $\vec{q}$ 投影到 $\hat{r}$ 上得到 $\vec{r} = \hat{r}\left(\hat{r} \cdot \vec{q}\right)$，这样就很容易得到 $\vec{s}$。向量 $\vec{s}$ 就是从 $\vec{r}$ 到 $\vec{q}$，因此 $\vec{s} = \vec{q} - \vec{r}$。替换 $\vec{r}$ 得到：$\vec{s} = \vec{q} - \hat{r}\left(\hat{r} \cdot \vec{q}\right)$。
5. 解：

   $\vec{u} = B - A = \langle 2, 0, 1 \rangle$

   $\vec{v} = C - A = \langle 3, 2, 1 \rangle$

   将 $\vec{u}$ 和 $\vec{v}$ 归一化。

   $\hat{u} \approx \langle 0.89, 0, 0.45 \rangle$

   $\hat{v} \approx \langle 0.80, 0.53, 0.26 \rangle$

   利用点乘得出 θ。

   $\theta = \arccos(\hat{u} \cdot \hat{v}) \approx 0.59$（弧度）或者34°

6. $\hat{v} \times \hat{u} \approx \langle 0.23, -0.12, -0.47 \rangle$

   你应该对其进行归一化，最终得到 $\langle 0.43, -0.23, -0.86 \rangle$。
7. 这会得到反方向的法线。
8. 将玩家角色向前的向量与角色到音源的向量进行叉乘。如果叉乘结果的 z 值是正的，左边扬声器会有声音。否则，右边有声音。
9. 答：

$$\begin{bmatrix} 2 & 4 & 6 \\ 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

10. 当矩阵是正交的时候。

第 4 章：3D 图形

1. 三角形是最简单的多边形，只需要用 3 个顶点表示，而且能够保证都在一个平面上。
2. 渲染管线中 4 个主要的坐标系分别是：模型坐标系、世界坐标系、视角/摄像机坐标系和投影坐标系。模型坐标系是相对于模型原点的坐标系（通常由建模程序设定）。

世界坐标系是相对于关卡的坐标系，所有世界中的物体都放置于相对世界原点的位置。视角坐标系以摄像机的视角观察世界。投影坐标系是将 3D 视角平铺到 2D 平面的坐标系。

3. 乘积的顺序应该先乘旋转矩阵再乘平移矩阵：

$$\text{RotateZ}(45^\circ) \times \text{Translate}(2,4,6)$$

$$= \begin{bmatrix} \cos 45^\circ & -\sin 45^\circ & 0 & 0 \\ \sin 45^\circ & \cos 45^\circ & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 4 & 6 & 1 \end{bmatrix}$$

$$\approx \begin{bmatrix} 0.71 & -0.71 & 0 & 0 \\ 0.71 & 0.71 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 4 & 6 & 1 \end{bmatrix}$$

$$\approx \begin{bmatrix} 0.71 & -0.71 & 0 & 0 \\ 0.71 & 0.71 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 2 & 4 & 6 & 1 \end{bmatrix}$$

4. 正交投影没有深度感，就是说对象不会随着摄像机的远近而变大或者变小，而透视投影则有 3D 深度感。

5. 环境光和方向光都全局作用于场景。但是，环境光是统一地应用于整个场景，而方向光则是来自某个特定的方向，因此背向光源的地方不会被照亮。

6. Phong 反射模型的 3 个分量分别是环境光、漫反射和高光。环境光分量统一地应用于所有对象上。漫反射分量则取决于表面与光源的位置关系，是主要的反射光。高光分量会在表面上营造出闪亮的高光，它由光源的位置和摄像机的位置决定。Phong 反射被认为是一种局部光照模型，是因为每个对象都认为自己是场景中的唯一对象。

7. Gouraud 着色是逐个顶点地着色，而 Phong 着色则是逐个像素地着色。因此 Phong 着色在低多边形模型上看起来会更好一些，但是相对而言计算量会更大。

8. 3D 场景使用画家算法的最大问题是场景从后向前排序非常耗时，特别是在摄像机位置动态变化的时候。而且它渲染每一帧都会造成过多次重绘问题。最后就是三角形重叠的时候，需要通过画家算法将三角形拆分成更小的三角形。深度缓冲通过检查每个像素的深度之后才进行绘制的方法解决了这些问题——如果已经绘制的像素比当前的像素更靠近摄像机，当前的像素将不会进行绘制。这样允许场景以任意顺序进行绘制。

9. 欧拉角旋转只能绕坐标轴进行，而四元数则可以绕任意轴旋转。欧拉角还会遇到万向锁的问题，而且很难进行插值。
10. 解出向量分量和标量分量如下：

$$q_v = \langle 1,0,0\rangle \cdot \sin\frac{90^\circ}{2} \approx \langle 0.71,0,0\rangle$$

$$q_s = \cos\frac{90^\circ}{2} \approx 0.71$$

第 5 章：游戏输入

1. 数字设备是二进制的——状态要么开要么关。数字设备的示例就是键盘的按键。另一方面，模拟设备就是有着取值范围的设备，比如摇杆。
2. 同时按键就是同时按多个按钮以完成某个特定动作。而顺序按键则是按顺序地按键。
3. 没有“刚刚按下”和“刚刚释放”事件，按键按下会分多帧检测。比如说，如果你快速按着空格键，那么在 60 FPS 下空格键会在多帧被检测到按下，在多个连续帧逐次触发空格键行为。另一方面，如果只使用“刚刚按下”，那么只有空格键按下的第一帧会触发行为。
4. 模拟输入过滤解决的是低精度输入在数值上的偏差。比如说摇杆哪怕是空闲状态也很难得到精确的 (0,0)。就会导致接近 0 的值都被检测出来。过滤会确保接近 0 的取值都会被认为是 0 输入。
5. 轮询系统会让不同位置的代码有可能得到不同的结果，如果按键在一帧当中状态有变化的话。更进一步来讲，轮询系统可能对同一个按钮在多个地方进行了多次不必要的查询。这会导致代码重复。另一方面，一个事件系统试图集中所有输入，让感兴趣的模块注册到这个系统中。同时也保证了一个特定键在一帧中只有一个值。
6. 随着程序语言支持存储函数（或者指向函数的指针）作为变量，简单地存储所有已注册函数的列表到一个特定事件就可以了。
7. 一个保证 UI 优先处理事件的方法就是优先传递触发事件的容器给 UI。UI 可以响应自己选择的事件，然后从容器中移除这些事件。最后剩余的事件被传递给游戏世界代码。
8. Rubine 算法是基于笔画的手势识别算法，用于识别用户绘制的手势。这个算法有两个步骤。第一步是创建已知手势的库以存储一些手势的数学特征。然后在用户绘制手势的时候，将手势的数学特征与库进行比对，看是否匹配，从而进行识别。
9. 加速器测量主要坐标轴上的加速度。而陀螺仪检测轴上的旋转。

10. 增强现实游戏可能会用到摄像机来“增强”世界，或者使用 GPS 来检测附近的玩家或者障碍。

第 6 章：声音

1. 音频文件是声音设计师创造的真实音频。通常是音效的 WAV 文件，而一些较长的音乐则会使用某种压缩格式。元数据则描述了音频在怎样的上下文如何播放。
2. 可切换的声音提示允许根据不同环境使用不同的声音集进行播放。比如说，脚步声会根据角色行走地面的不同而不同。通过切换，就可以根据不同的地表材质播放不同的脚步声音效。
3. 监听者是一个虚拟麦克风，它会选取所有游戏世界中播放的声音，而发射者就是那些实际能够发出声音的对象。在第一人称射击游戏当中（FPS），监听者可能就是摄像机，而发射者可能绑在每个非玩家角色（NPC）身上，当然还有其他会发生声音的对象也是发射者（比如壁炉）。
4. 对于第三人称游戏来说，监听者的位置和朝向同样重要。朝向应该总是与摄像机相关，而不是与角色相关。如果与角色相关，你就会发现一个场景，爆炸发生在屏幕的右边，但是却从左扬声器播出，而这是我们不希望发生的。监听者的位置通常都放置在玩家与摄像机之间。
5. 分贝是一个对数标量。减小 0~-3dB 的音量几乎就可以将声音减半。
6. 数字信号处理涉及以某种方式提取信号以及变换信号。对于音频而言，它常用于改变播放的声音。一个例子就是余响，它会造成回音。另一个例子就是音高调整，会增加或者减小音高。最后一个例子就是压缩机，缩小了音量的范围，导致很小的声音得到了增强，同时很大的声音得到了减弱。
7. 通常，像余响这种 DSP 效果只需要影响一个关卡的一部分。因此，在受 DSP 影响的关卡内能够标记区域很有用。
8. 通过使用凸多边形标记 DSP 的方法在遇到重叠的时候不能正常工作。比如说，一块地形底下有管道，上面标记了凸多边形就会影响下面管道行走的玩家。
9. 多普勒效应会在声音发射者快速靠近或者远离的时候出现。在靠近的时候，音高会上升，而远离的时候，音高下降。在遇到警车报警器的时候它会经常发生。
10. 声音遮挡是在声波必须穿过表面才能达到监听者的时候发生。结果就是低通道过滤效应，就是说高频声音会变得更难听到。在声音衍射中，声音虽然无法直接抵达监听者，但是可以绕过表面。

第 7 章：物理

1. 一个轴对齐包围盒每一侧都必须与坐标轴平行（3D 的话就要与坐标系平面平行），这使得 AABB 更加容易计算。而朝向包围盒则没有这种约束。
2. $P \cdot \hat{n} + d = 0$

   P 是平面上任意一点，$\hat{n}$ 是平面的法线，d 是平面到原点的最短距离。在游戏引擎中，我们通常在平面数据结构中存储 $\hat{n}$ 和 d。
3. 参数方程就是用任意参数定义的函数（通常使用 t 表示）。

   一束光线可以表示为 $R(t) = R_0 + \vec{v}t$。
4. 两个球如果它们中心点之间的距离小于半径和则发生交叉。但是，因为计算距离需要开平方根，所以检查距离的*平方*小于半径和的*平方*会高效得多。
5. 检测 2D AABB 与 AABB 交叉的最好方法就是检查 4 种 AABB 不发生交叉的情况。
6. 在线段与平面交叉的检测中，t 为负数表示线段远离平面。
7. 瞬时碰撞检测算法只能检测两个对象当前帧发生的碰撞，而连续碰撞检测算法则可以检测出两帧之间的碰撞。
8. 对于扫略球，如果判定式为负，意味着两个球没有碰撞。如果等于 0，表示两个球相切。最后，如果为正，表示两个球完全交叉，而较小的根为交叉的第一个点。
9. 数值积分方法的准确性直接由时间步长的大小决定。如果时间步长变化很大，那么数值积分方法会变动很大。
10. 速度的 Verlet 积分法首先计算时间步长中点的速度，所使用的加速度来自上一帧。之然后将这个速度在完整的时间步长上作用于位置积分。之后更新当前帧的加速度，这个加速度在步长的中点到步长的结束区域内作用于速度的计算。

第 8 章：摄像机

1. 视场表示能够观察游戏世界的角度。视场太窄可能会导致头晕，特别是在 PC游戏中。
2. 基本跟随摄像机的位置可以用下面的方程计算：

   ```
   eye = target.position - target.forward * hDist + target.up * vDist
   ```

   在计算出眼睛位置之后，摄像机前向向量就是从眼睛到目标的向量，如下：

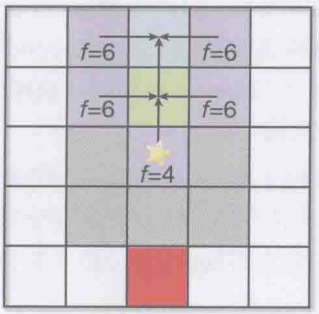
   ```
   forward = eye - target.position
   ```

   通过目标上方和摄像机前方向量的叉乘得到摄像机左方向量，然后摄像机上方向量可以通过将摄像机前方与摄像机左方向量叉乘得到。

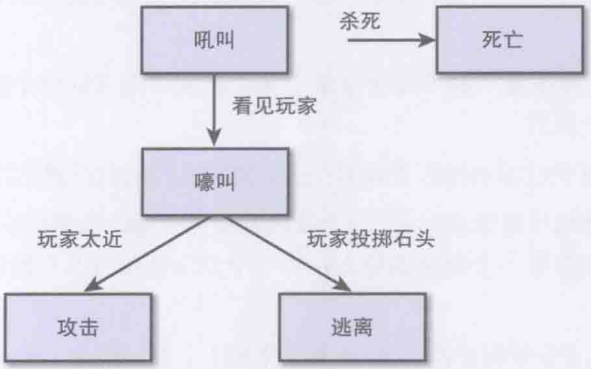
3. 弹性跟随摄像机可以设置理想和真实的摄像机位置。理想位置根据精确的基本跟随摄像机来更新每一帧，而真实的摄像机位置则稍微延后。这样创造了比基本跟随摄像机更加流畅的摄像机体验。
4. 旋转摄像机中的摄像机位置应该以到目标对象的偏移量存储，因为这使得旋转更加容易。
5. 目标位置作为摄像机的偏移量来存储，而随着角色旋转且四处观望，目标位置也需要跟着旋转。
6. Catmull-Rom 样条是一种可以通过最少 4 个控制点定义的样条曲线：一个前、一个后、中间两个插值。
7. 样条摄像机对于过场动画非常有用，这里摄像机需沿着设置的路径。
8. 如果跟随摄像机被对象遮挡了，一个方法就是从目标到摄像机位置投射一条光线，然后将摄像机放在第一个碰撞的地方；另一个方法就是将所有摄像机与目标之间的对象变得透明。
9. 在反投影中，z 分量为 0 表示 2D 点在近平面上，而 z 值为 1 表示同一点在远平面上。
10. 拣选可以通过 2D 点在近平面和远平面之间进行反投影来实现。然后两个点之间的光线投影就可以知道与哪个对象发生了交叉。

第 9 章：人工智能

1. 节点 0：{1,4}；节点 1：{0,2,4}；节点 2：{1,3}；节点 3：{2,4}；节点 4：{0,1}。
2. 通过导航网格而非路径节点来代表典型区域，需要更少的节点。同时，导航网格对一个区域的覆盖也会更广。
3. 如果估算花费比实际花费要少或相等，那么启发式就为可接受的。
4. 曼哈顿距离：6；欧几里得距离：$2\sqrt{5}$。
5. 会选择下面有星星的节点：



6. Dijkstra 算法。
7. 当估算访问节点的开销时，A* 算法同时考虑从开始节点的真实开销和到结束节点的估算开销。而 Dijkstra 算法则只考虑从开始节点的真实开销。A* 算法一般比 Dijkstra 算法更有效，因为这会导致访问更少的节点。
8. 该问题有很多的可能解。以下是其中之一：



9. 状态设计模式使得某个特定状态的行为可以被封装在它自己的类当中。这使得状态可复用以及可交换。同时也意味着“controller”类会比在可选的枚举实现中更加干净。
10. 策略是 AI 玩家的大局观。常见的方法是将策略分为几个小的目标来执行，同时赋予优先级，然后 AI 就通过各计划来完成。

第 10 章：用户界面

1. 菜单栈提供了一系列不同的好处。首先，它能确保用户弹出栈中的当前菜单之后能够轻松回到之前的菜单。更重要的是，它能够使得多个 UI 元素在其他菜单之上绘制，比如弹出确认/拒绝对话框。
2. 字母 key code 是典型的顺序关系，因此 C 的 key code 是紧跟着 B 的 key code，而 B 又紧跟着 A，以此类推。这种属性在检查某个 key code 是否落在 A~Z 范围的时候很有帮助，只需要简单地减去 A 的 key code 即可。这样就可以得到字母表中的偏移，然后再加到“A”上得到相应的字母。
3. 一般而言，路点箭头应该位于称为 2D 屏幕空间的坐标位置上。但是，箭头又确实是一个 3D 对象，我们需要世界空间坐标系来正确绘制。方法就是取 2D 屏幕空间位置，然后做一次反投影，结果就会得到合适的 3D 坐标位置。

4. 为了计算路点箭头简单的旋转轴以及旋转角度，我们需要先构造一个从 3D 玩家到路点的向量。在向量归一化之后，我们使新向量与原有朝向向量做一次点乘，得到所需的旋转角度。相似地，叉乘用于计算旋转轴。
5. 准心就是一个在屏幕上的 2D 点。我们可以取这个 2D 点，然后做两次反投影：一个在近平面做，另一个在远平面做。然后代码可以在两点之间构造光线投射，查看第一个与光线投射相交的对象。如果是敌人，准心可以绘制成红色；如果是友军，那么可以绘制成绿色。
6. 为了将得到的 3D 坐标转换成雷达上的 2D 坐标，我们首先忽略高度分量。在 *y* 轴向上的世界里，这意味着会通过使用 3D 向量的 *x* 分量和 *z* 分量构造 2D 向量。
7. 绝对坐标是指屏幕上具体的像素位置，而相对坐标则是要么相对于屏幕上的关键点（比如中心点或角点），要么相对于其他 UI 元素而言的相对位置。
8. UI 文本硬编码使得非程序员很难修改开发中的文本。还有就是，硬编码文本本地化比较困难。这就是为什么文本应该存储在外部文本文件中。
9. ASCII 字符编码集只能显示英语字母。Unicode 字符编码集解决了多语言编写系统的问题，包括阿拉伯文和中文。最流行的 Unicode 编码表示是 UTF-8，其中字符的长度是可变的。
10. 用户体验涉及用户使用的反馈。一个设计良好的游戏 UI 对用户体验的影响很大。

第 11 章：脚本语言和数据格式

1. 脚本语言让开发迭代更加敏捷，同时也让非程序员也能参与编辑。但是脚本语言的缺点就是性能上没有编译型语言 C++ 那么好。
2. 脚本语言可以用于开发摄像机、AI 状态机以及用户界面。
3. UnrealScript 是一种在游戏开发中比较流行的自定义脚本语言。自定义脚本语言的优点在于可以为游戏开发定制。在 UnrealScript 的例子中，这意味着它支持状态功能。同时它也会针对游戏性能进行优化。缺点则是，自定义语言通常而言会比通用语言在实现上有着更多的问题。
4. 可视化脚本系统允许用户通过基本的流程图开发逻辑。通常用于开发特定的关卡逻辑，比如敌人在何时何地出现。
5. 表达式 `int xyz = myFunction();` 会分成以下标记：

```
int
xyz
=
```

```
myFunction
(
)
;
```

6. 正则表达式 `[a-z][a-zA-Z]*` 匹配了一个小写字母后接着 0 个或者多个小写或者大写字母。
7. 抽象语法树以树的结构存储程序的整个语法布局。AST 通常会在语法分析时生成，然后以后序遍历的方式来生成或者执行代码。
8. 二进制文件格式通常比文本格式加载要更快、更节省内存，但问题是在比较二进制数据文件与文本数据文件的区别的时候比较困难。
9. 对于基本程序配置而言，INI 文件是比较好的格式，因为不仅简单而且终端用户可以编辑。
10. 《魔兽世界》中的两个主要 UI 插件为布局和行为。布局以 XML 实现，而行为用 Lua 实现。

第 12 章：网络游戏

1. 互联网协议是在互联网上发送任何数据都需要遵守的基本协议。IPv4 和 IPv6 的主要区别在于地址的设置。在 IPv4 中，地址是由 4 个字节组成，总数为大概 40 亿个组合。在 IPv4 出现的时候看起来有很多地址，但是近几年已经几乎用光了。IPv6 将地址的组合扩展到 2^{128}，这意味着地址不会马上用完。
2. ICMP，也叫作网际网络控制消息协议，是用于获取网络和连接的状态信息。在游戏中，常用于回声请求和回声应答。这样，游戏就可以通过“ping”确定数据包发送到接受所需要的时间。
3. TCP 是面向连接的，这就是说两台计算机必须先经过握手之后才能够开始传输数据。还有就是 TCP 能保证所发送的所有数据包都到达，而且还与发送的顺序一致。
4. 端口可以认为是虚拟邮箱。每台计算机大约拥有 65000 个 TCP 端口和 65000 个 UDP 端口。许多端口是用于特定类型的数据（比如 80 用于 HTTP 服务），但是也不一定要严格遵守。端口的好处就是使得一台计算机上的多个应用程序访问网络而不会相互影响。
5. 对于实时游戏而言，UDP 是更好的选择，因为 TCP 额外的可靠性会导致不必要的性能损失。有很多信息不是很关键而且是可丢失的。但是用了 TCP，任何数据只要没收到确认都会重发，这会大大降低处理速度。

6. 在服务器/客户端模型中，所有客户端连接到一个中心服务器。这就是说服务器需要比客户端更加强大以及有更大的带宽。在点对点模型中，每台计算机都会连接到其他计算机，这就是说它们是对称的。
7. 服务器会周期性地发送更新游戏中其他玩家的数据。客户端预测则是尝试填补周期中间的空白时间，让游戏更加顺畅。没有客户端预测，远程玩家会在每次收到数据时都在屏幕上闪现，使得趣味性大大降低。
8. 在帧同步模型中，游戏会分为很多 150ms~200ms 延迟的回合。所有回合中的输入指令都会排队，一直到回合结束。然后每段都会根据输入指令模拟出结果。最终同步结果是完全一样的。
9. 信息作弊的例子在雷达上显示本来没有的信息。雷达可以由传递给所有客户端的位置信息创建。一个解决方法就是让服务器根据玩家状态来发送信息。
10. 中间人攻击就是在两台通信的计算机中间加一台计算机。这会导致坏人可以通过拦截数据然后解析篡改数据。主要的解决方案就是对数据包进行加密，这样只有服务器可以读取数据包的内容。

附录 B
对开发者有用的工具

有不少工具可以让编程更少犯错以及提升开发者的能力。本附录介绍了一些最有价值的工具——很多专业的开发者每天都会用到它们。这些工具很通用，因此非游戏程序员也能受益。

调试器

新手程序员经常通过文本输出语句来调试代码问题，比如C++中的cout或者C中的printf。对于控制台应用，这会比较合理。如果结果是确定的，使用文本输出完全没问题。但是对于实时程序比如游戏来说，打印消息就不是那么适合了。

想象你试图解决某个游戏对象的位置问题。你可能会尝试在对象的Update函数中添加打印语句以输出对象每帧的位置。但是这种方法的问题在于这样控制台每秒输出60次对象的位置（假设游戏运行在60 FPS）。要从大量垃圾信息中得到有意义的信息就会比较困难——而且最坏的是，垃圾信息会降低帧率。

这并不是说在游戏中输出消息没有价值。如果这个问题只发生一次，比如加载纹理失败，输出错误文本就完全合适。但更好的方案是在游戏中显示临时的纹理，用大红色块或者其他颜色来标记，所以任何玩游戏的人都会发现这个问题。实际上最大的问题是有的团队成员不太关注文本的控制台输出，但是他们肯定会发现红色的纹理。

所以如果输出消息不是调试游戏（或者说大多数程序）的最佳方式，那么应该用什么方法？答案就是调试器。每个IDE（集成开发环境）都会自带各种调试器。其中有的特别好用，但基本原理都差不多。本节讲讲Visual Studio 2010中的调试器，其中涉及的概念几乎在所有IDE上都有（当然，菜单和按键肯定不同）。截屏显示了第14章塔防游戏的代码。为了调试这款游戏，你需要将它设置为窗口模式，可以通过设置GlobalDefines.bFullScreen为false来完成。

基本断点

在一行代码中设置**断点**，会告诉调试器当执行到这一行时，暂停程序。在Visual Studio中设置断点，只要将光标移动到要断点的那一行，然后按下F9键即可（另外，也可以单击编辑窗口中左侧灰色的一列来设置）。当断点被触发，你可以将鼠标放置在变量上，观察它们的值。图B.1演示了Enemy.cs的Update函数中断点的情况。

在特定断点触发之后，代码逐行往下执行是非常好用的，这样就能快速定位问题。有两种方法可以调试代码。如果你使用逐过程（按下F10键），会执行到下一行代码，而非进入当前行调用函数的代码。举例来讲，假设你有以下两行代码，断点在第1行上：

```
myFunction();
x = 5;
```

如果在第1行上逐过程执行，就可以立刻跳到第2行，然后就能看到myFunction()执行后的结果。你不会看到函数中执行的代码。如果你真的想要跳到特定函数中逐行执行，可以

使用逐语句（按下 F11 键）。这样将进入代码行中的函数，以函数调用的顺序跳转。如果你逐语句执行到函数中，然后发现自己不关心特定函数的逐行行为，可以使用跳出（按下 F12 键）跳到函数外部。

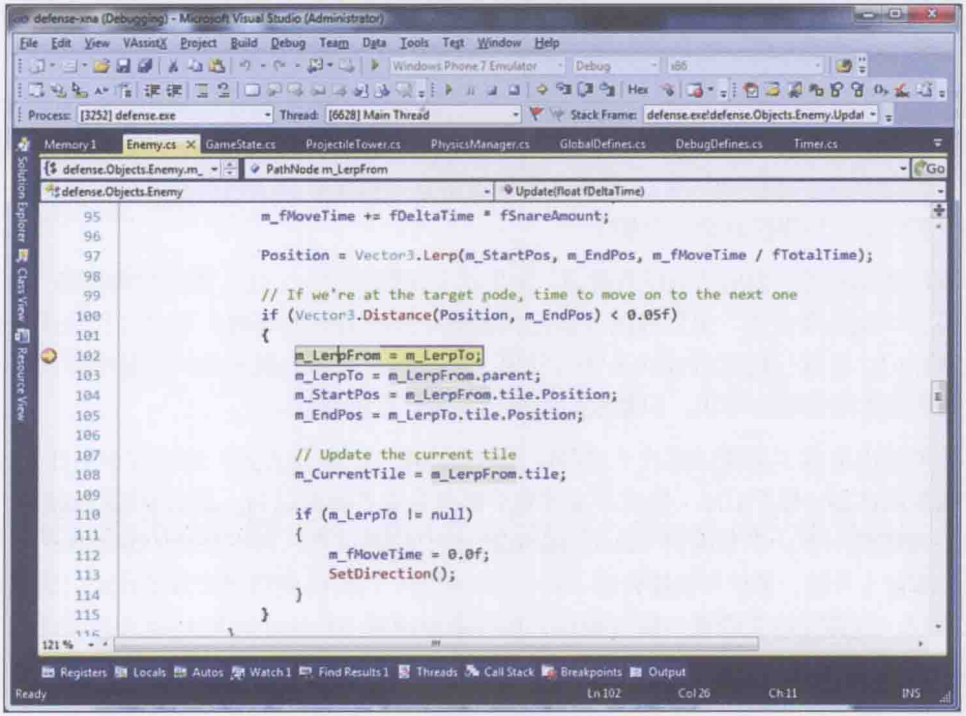


图 B.1 Visual Studio 中的基本断点

监视

监视可以让你跟踪某个变量，而不用每次都用鼠标单击观察它们的值。添加监视最简单的方法就是在调试器的某个变量上单击鼠标右键，然后选择添加监视。当调试的时候，就可以在 IDE 底部的监视窗口中观察变量的当前值。如果变量是结构体或者类，可以单击 + 号来观察具体的成员变量。在代码逐行执行的时候，只要监视的变量发生变化，变量的颜色就会变成红色。这是最简单的跟踪变量变化的方法。监视的例子如图 B.2 所示。

Visual Studio 中还有两个自动监视窗口：局部与自动。局部表示所有当前作用域内的活跃变量。如果在 C# 或者 C++ 的类中，本地变量之一总会有一个 this 指针，自动监视列表则比较受限，只包含当前执行代码行中的变量。

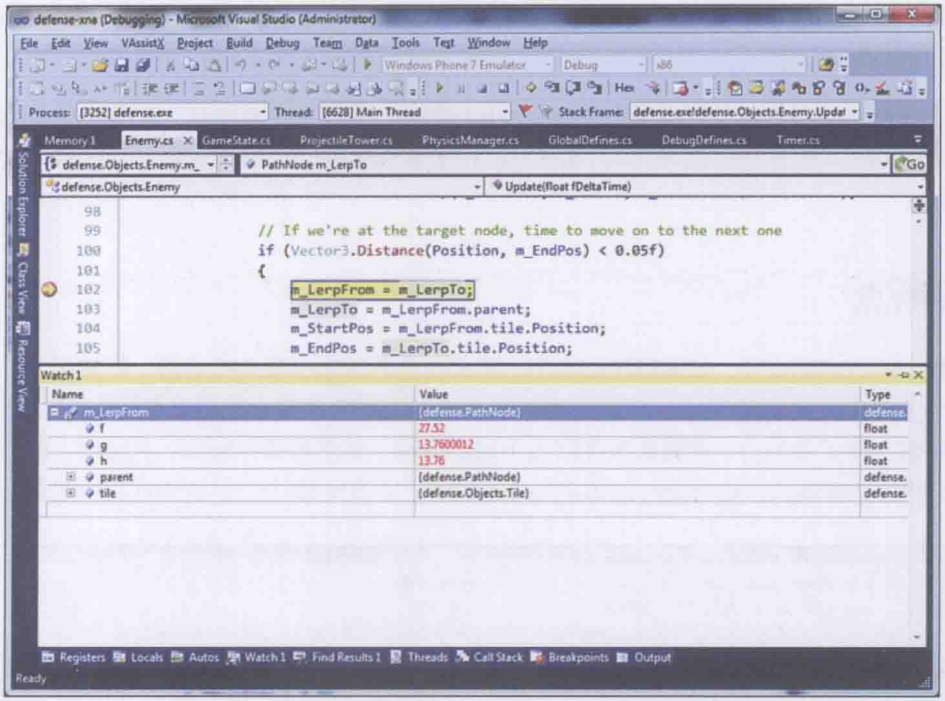


图 B.2　在 Visual Studio 中监视变量

一个值得关注的事情是所有监视类型在“Debug”模式下能够工作，但是在“Release”模式下不一定能工作，这取决于具体的语言。这是因为“Release”版本是经过优化的，而优化的一个方面则是移除所有调试器用于跟踪变量的信息。这在 C++ 中尤为明显，“Release”版本中有积极优化的惯例。

对于 AAA 游戏来说，一个常见的问题就是“Debug”版本通常都不可玩。如果有一款游戏把系统性能用尽，那么“Debug”模式运行低于 5 FPS 也是很常见的，这样测试就变得没有效率。这个问题的一个解决方案就是开启一个中间版本，即拥有 Debug 的功能，但是又有一定程度的优化。另一个选择则是运行在“Release”模式下，但是对于某个要调试的文件关闭优化。在 Visual Studio 中，可以在解决方案浏览器中右键关闭这个文件的优化。但是在调试完之后不要忘记恢复回来。

调用栈

调用栈窗口可以告诉你哪个函数调用了当前调试器所在的函数。这对于没有处理过的异常或者崩溃相当有帮助。当你在 Visual Studio 中遇到崩溃，你会看到带着中断按钮的对话框。如果单击中断按钮，调试器就会在崩溃的地方暂停下来。这使得在程序崩溃的时候还可以

观察当时变量的值，也能看到调用栈中有什么。后者在你找不到在某点调用某函数的原因的时候非常有用。

当你遇到崩溃的时候不要到处使用输出语句以试图找到崩溃发生的点，这是一种找到崩溃源的糟糕方式（而且也不太可靠）。你有专门用于解决这种问题的调试器，那么就应该好好利用它。

条件断点

在函数中断点问题在于，如果每帧都调用，那么就会不断地引发游戏暂停。但是假设你只想在某个条件为真的情况下才在断点位置暂停，比如说实例的某个成员变量为 0 才引发断点，传统的断点就无能为力。这就是条件断点出场的时候。要添加条件断点，右键单击红色断点圆圈，然后选择条件。然后就可以得到一个对话框，在其中输入一定的条件，如图 B.3 所示。

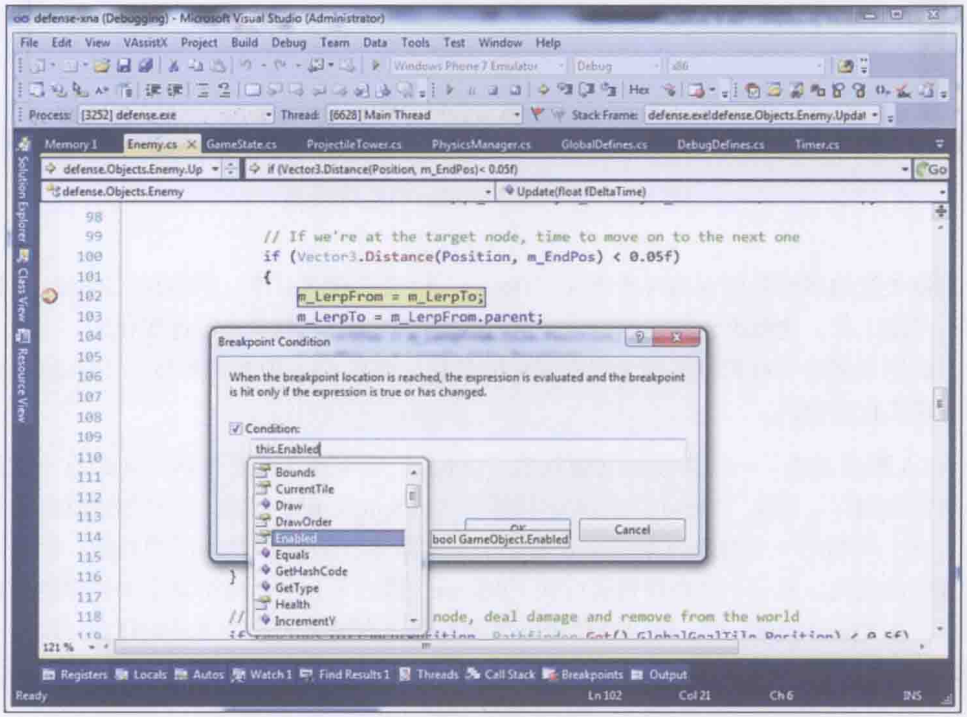


图 B.3 在 Visual Studio 中设置条件断点

在设置好条件断点之后，你就只会在条件成立的时候在那一行停下来。这种方法在你需要特定条件下才引发断点的时候非常好用。另一种条件类型则是**命中次数条件**，就是只有断点被触发一定次数之后才会引发。要使用命中次数条件，可以右键单击断点，然后选择命中次数。

数据断点

假设在 C++ 游戏中你有一个成员变量叫作 `m_Alive`，它会跟踪某个特定的游戏对象是否存活。在调试的时候，会注意到有时候 `m_Alive` 变量改变了，哪怕问题对象没有调用 `Die` 函数也会被设置为 `false`。这就是说存储 `m_Alive` 的内存地址在某个地方以某种方式被坏值重写了。这大概是最让人头疼的问题之一，原因有可能是坏指针修改了代码的任意位置。

Visual Studio 中解决这个问题非常有用的一个功能就是使用数据断点，这类断点只能在 C/C++ 中使用。一个**数据断点**可以在某块内存被修改的时候让调试器暂停程序。内存范围大小通常在 32 位操作系统下是 4 字节，64 位操作系统下则是 8 字节，而数据断点的个数取决于系统。

在我们 `m_Alive` 变量的场景下，要设置数据断点，我们要先在游戏对象构造函数中放置一个常规断点。一旦断点命中，我们可以在菜单中选择调试、新建断点、新建数据断点。其中数据断点的地址，我们可以填写为 `m_Alive` 在内存中的地址 `&m_Alive`。然后只要内存中值发生变化，调试器就会暂停，就知道哪块代码修改了它。

版本控制

假设你自己正在开发一款简单的太空船战斗游戏，大概有着 10 个左右的源文件。有一个 bug 是在按一定顺序捡起道具时才会引起的，结果是飞船无法发射子弹。然后你花了数个小时来解决这个问题（通过调试器），找到问题的根源。你修复了 bug，解决问题之后你很高兴。你继续增加更多的功能，文件修改了无数次。

一周之后，突然这个 bug 又一次出现。你开始困惑。是因为最开始没有把问题修复完吗？还是你做的大量其他变动又一次破坏了代码？现在，一个问题就是你可能不记得最开始修改问题的地方了，因为你没有文件修改的历史记录，也不记得问题原因。哪怕你一整个星期开着计算机开着 IDE，你也不可能回到你修复 bug 的那个点。而不管怎样，你都不想失去一周的工作。

版本控制为所有文件都提供了清晰的历史记录。它能够显示文件之前所有提交的版本，而且通常能告诉你每个版本都修改了哪一行代码。有了版本控制，要浏览历史记录以及找到过去修复的 bug 就简单多了，要找到游戏为什么再次出问题就迎刃而解。

虽然版本控制系统在哪怕只有一个开发者的情况下都非常有用，但是版本控制系统其实是为团队项目开发的。通过版本控制，就可以本地下载一份最新的代码。团队成员可以确保他们总是同步更新的，很清楚地知道谁修改了文件以及为什么修改文件。而且还可以处理多个开发者同时修改一个文件的合并问题，而没有版本控制的日子简直就是噩梦。

有许多类可供选择的版本控制系统，但是只要熟悉其中一个系统之后，学习其他版本控制系统就变得很容易。它们都有着各种各样的工作流，但是基本上都差不多。本节会介绍两种最流行的版本控制系统：SVN 和 Git。Git 会讲得更详细一些，因为在第 13 章、第 14 章的示例代码中用到了它。但这不代表 Git 在任何场合下都是最好的解决方案。

SVN

Subversion，也叫作 SVN，是一种中心版本控制系统。在 SVN 中，需要先从中心服务器（称为 **repository**）“check out”一份代码。每当你对文件做了任何修改，都可以提交到服务器，也就是“commit”。所有操作，包括 check out、commit、update 到最新版本，都需要经过中心服务器。中心服务器的优点就是总是有最新的版本，但是缺点就是操作更慢，因为它总是需要经过网络。

因为 SVN 要求使用中心服务器，所以使用版本控制系统之前必须先设置好。服务器的选择上有许多选项。如果问题项目是**开源项目**（就是说任何人都可以免费访问代码），有不少网站可以部署 SVN。这些流行的网站包括 Google Code（*http://code.google.com*）和 SourceForge（*http://sourceforge.net*）。

如果你不想让别人访问你的项目源码，就需要有一个私人 SVN 服务器。在 Windows 上，最简单的方法就是通过 VisualSVN Server（*http://www.visualsvn.com/server/*）设置 SVN 服务器。相对地，如果你有一个 Web 服务器提供命令行访问，也可以通过命令行设置 SVN 服务器，只是会麻烦一点。

在设置好 SVN 服务器之后，就需要连接上去。虽然 SVN 有提供命令行 SVN 客户端，但是 Windows 上最简单的客户端是 TortoiseSVN（*http://tortoisesvn.net/*）。TortoiseSVN 很好用的一个方面是它集成在浏览器中，因此你可以右键单击文件或文件夹以对其进行操作。

典型的 SVN repository 根目录下有 3 个文件夹：`trunk`、`branches`、`tags`。**trunk** 是当前最新版本，几乎所有当前的开发都在这里。**branch** 是代码的分支备份，通常用于开发新功能。分支的理念就是可以增加大功能而不会影响到其他人。当分支上的大功能经过充分测试之后，可以合并到主分支上。最后，**tag** 是主分支某个版本的备份。举个例子，如果发布了程序的 v1.0 版本，你可能希望在继续开发之前制作一个 v1.0 的 tag 备份。这样，v1.0 的某些代码就可以很容易访问，哪怕以后 trunk 发生了很大变化都没有影响。

Git

不像 SVN，**Git** 是一个分布式版本控制系统。分布式的本质主要体现在两个主要方面。首先，使用 Git 不需要有中心服务器。只要安装好 Git 客户端，就可以在本地创建和使用版本控制

系统。这就是说如果只有你一个人工作，操作会比 SVN 这种中心服务器要快得多，因为不需要通过网络连接服务器。

但是，如果项目有多个开发者，看上去还是需要中心服务器。这是另外一个 SVN 和 Git 不同的地方。比起直接从中心服务器 check out 代码的最新备份，Git 可以 **clone** 整个中心版本库。这意味着不仅可以获取最新的代码，还可以获得所有文件的所有历史版本的备份。这对于趋向于 **fork** 的开源项目非常有用，fork 就是与原项目稍微有点不同的分支版本。一个被 fork 得最频繁的项目就是 Linux，所以说 Linux 的主要开发者 Linus Torvalds 同时也是 Git 的主要开发者就很合情合理。

尽管没有 fork 项目的打算，对版本库进行备份还是有很多好处的，任何操作都只在本地副本上执行，而不是在中心版本库。也就是说提交文件行为会更快，而且不需要连接网络进行提交。但另一方面，提交的文件改动不会自动提交到中心服务器。为了解决这个问题，Git 提供了“push”功能将提交内容从本地副本推到中心版本库，类似地，还有一个从中心版本库更新本地副本的操作，叫作“pull”。

但是如果你只想在本地网络上搭建中心服务器（在专业的环境下是有可能的），使用 Git 的优势就不是那么大。它需要额外同步到服务器，但是如果这正是你需要的，也许选择一个为此设计的系统会更好一些。

使用 Git

第 13 章、第 14 章实现的游戏源码存放在 GitHub（*http://github.com*）上，GitHub 是部署开源 Git 项目最流行的地方。虽然可以使用不同的 Git 客户端，从 GitHub 上备份版本库最简单的方式还是使用 GitHub 自己的客户端。Windows、Mac 平台都提供有相应的版本，而且工作方法大致相同。

首先，为了使用 GitHub 客户端，需要先创建 GitHub 账号。这可以在网站的主页上完成。在创建好账号之后，可以安装 Windows 版 GitHub 客户端（*http://windows.github.com*）或者 Mac 版 GitHub 客户端（*http://mac.github.com*）。在刚开始打开客户端的时候，它会要求输入刚刚注册的用户名和密码。

在客户端设置好之后，就可以开始使用 GitHub 了。打开想要备份的 GitHub 版本库页面（比如第 14 章塔防游戏为*https://github.com/gamealgorithms/defense*）。打开页面之后，在右侧可以看到标记着“Clone in Desktop”或者“Clone in Mac”的按钮。如果单击按钮，GitHub 客户端就开始 Git 备份了。如果过程顺利，项目就会添加到客户端的版本库列表中。

在备份版本库之后，可以在客户端右键单击选择打开 Explorer 或者打开 Finder，就可以在目录下看到所有存储的文件。接着就可以打开方案文件，编辑文件了。

在你准备为本地版本库提交改动的时候，你可以双击 GitHub 客户端版本库的名字，然后可以看到更多细节。在详情界面里，可以看到所有被修改文件的列表，而且可以展开每一项以查看每个文件具体修改了那些内容。选中所有希望提交的文件，然后输入提交注释。其中注释应该能够描述清楚你的工作内容。

一旦单击了 commit 按钮，那么所有选中的文件都被提交到本地的版本库中。但是，如果你想让这些变动推送到中心服务器（如果你有权限），可以单击窗口顶部的同步按钮。这会自动 pull 所有最新变动，然后 push 所有本地修改。要注意的是，如果没有权限，push 变动到版本库就会得到错误提示。

GitHub 客户端还可以做其他事情，比如创建分支、合并等，对于正常使用已经足够。但是为了展示 Git 的强大能力，你需要使用其他客户端。一个选择就是使用 Atlassian SourceTree（*http://www.atlassian.com/software/sourcetree/*），或者直接在命令行中使用 Git。

比较和合并工具

比较工具可以让你查看两个（或者更多）文件之间的不同。许多版本控制客户端都以某种方式集成了这个功能，比如 GitHub 客户端的 commit 窗口。传统的比较工具是一个 UNIX 小程序，它可以标记出两个文本文件之间的差异。幸运的是，更多现代工具可以直接在图形界面上比较出文件的差异。当与版本控制系统一起使用的时候，比较工具可以显示文件具体哪里发生变化。一个这样的开源比较工具是 TortoiseMerge，它是之前提到的 TortoiseSVN 的一部分。

这种工具一个更加复杂的使用是在多个版本之间对变动进行合并。假设你在编辑 GameState.cs，与此同时其他开发者也在编辑这个文件。如果你先将这个文件提交到中心服务器，其他开发者要提交的时候就会发现文件已经过期。版本控制客户端会要求他更新文件。如果你们编辑的是文件的不同地方，客户端通常会自动对不同版本进行合并。

但是如果两个开发者同时编辑了 GameState.cs 的同一行呢？这种情况下，版本控制工具就不知道该使用哪个，然后就会提示出现 conflict（冲突）。为了解决冲突，需要进行**三路合并**。在三路合并的时候，通常文件会分成 3 份，一份是别人的，一份是自己的，还有一份是合并的。对于每块冲突区域，需要选择合并文件中使用哪一块。有可能是用你的，也有可能是用别人的，又或者二者都用。

问题跟踪

问题跟踪涉及开发期间的文档以及对问题的解决方案跟进。问题跟踪也叫 bug 数据库，尽管问题不需要直接对应于 bug。我不太推荐项目早期引入问题跟踪系统，这时会实现某游戏的

主要功能，问题跟踪系统会导致注意力从开发转向修复已存在代码的 bug。在游戏进入开发阶段之后，问题跟踪就几乎是必需的。

常见的问题跟踪工作流就是 QA 团队测试游戏然后为发现的每个 bug 提交 bug 报告。这些 bug 报告会传递给产品经理或者主要开发者，也就是知道谁是最佳解决问题人选的人。这个人选有很多因素考虑，包括谁是最早代码的开发者，或者谁在负责这部分功能。每个问题都会有一个优先级，最高优先级通常会导致崩溃。

问题跟踪一个便利的地方在于，每个开发者都会知道自己的任务列表。这使得每个团队成员都知道自己的方向，知道自己要做什么，这是很重要的。还有一点就是让解决 bug 变得更快，最终能够把所有 bug 解决。每修复一个 bug，就会传递回去给 QA 团队验证。只要验证通过，问题就被关闭，但是 bug 被重新打开也是很常见的。

问题跟踪还提供了存在多少个 bug、存在多少崩溃 bug、哪个开发者修复最多 bug 等的数据。所以问题跟踪不仅提供了每个人清晰的任务列表，还提供了项目进度的统计量。如果在项目后期，新 bug 速度比解决 bug 速度要快，意味着发布日期要延后。

许多开源部署平台都提供了问题跟踪，包括 GitHub。所以如果你在第 13、14 章的代码示例中遇到了 bug，可以提交问题。但是如果不想开源部署，还可以使用其他程序，比如 Bugzilla（*www.bugzilla.org*）和 Trac（*http://trac.edgewall.org*）。

腾讯GAD
游戏开发者平台

GAD平台介绍

腾讯GAD游戏开发者平台，以聚合行业资源、服务从业人群、打造行业连接器为目标，对内依托腾讯互娱10年游戏开发经验，对外联合行业上下游各类优质厂商与机构，共建游戏产业生态联盟。通过《游戏开发者培养计划》和《游戏开发者扶持计划》为广大游戏开发者提供“学习、实践、孵化、发行”等全方位的服务，辅助更多有梦想的游戏人实现梦想，促进行业以“开放、连接、创新”为价值核心的良性生态体系逐步发展、完善。

GAD平台两大核心

1 游戏行业人才培养

腾讯大学 • GAD游戏学院

学习

线上资源
线上课程及文章

书籍编撰
游戏开发丛书编写发行

院校授课
大学开办选修课程

名人分享
行业名人交流分享

实践

在线组队
虚拟团队合作实践

赛事活动
如游戏创意高校大赛等

校园俱乐部
高校俱乐部网络建设

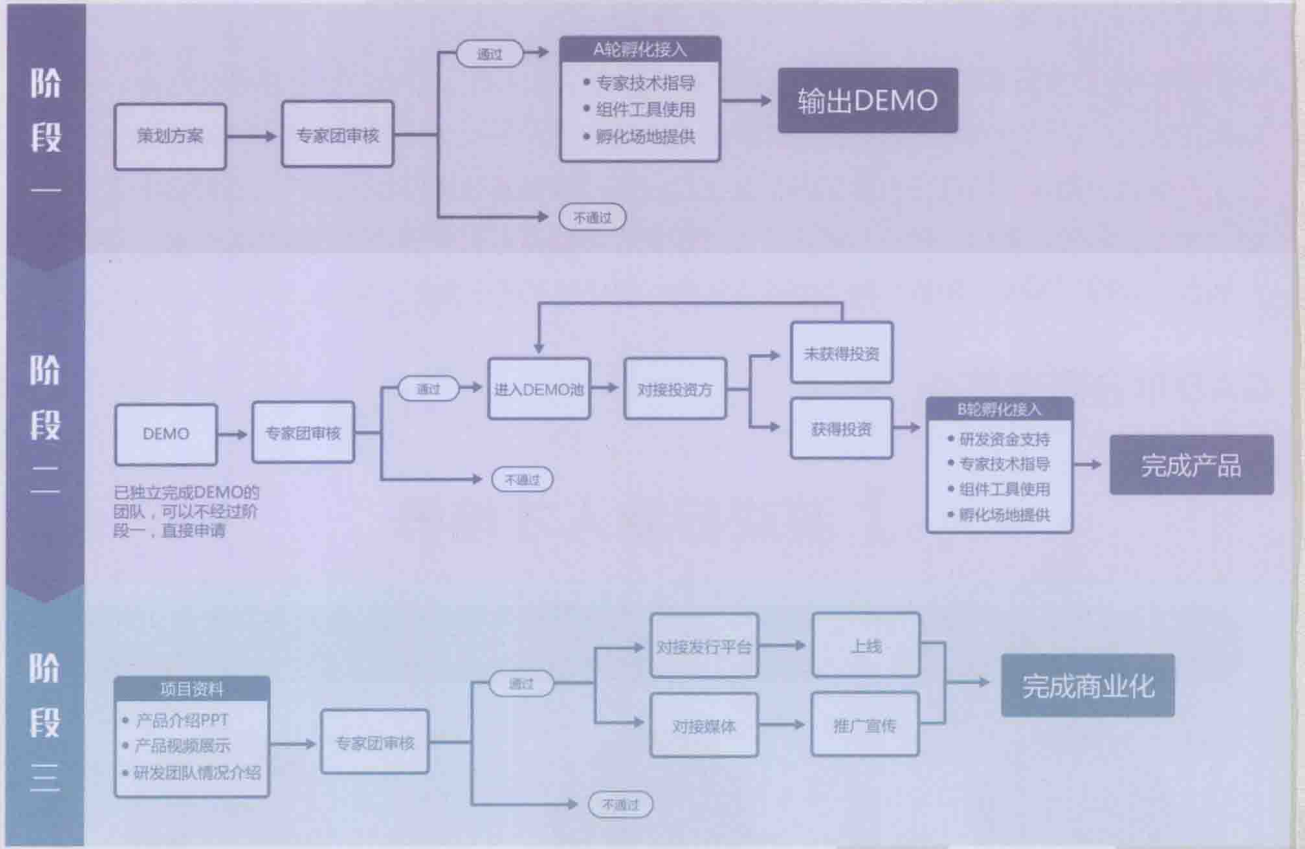
训练营
定期举办线下训练营

考核

学习考核
完成课程学习，修满学分，通过导师团考核可获GAD游戏学院颁发的专业技能证书

项目考核
通过赛事或俱乐部等方式提交的项目，通过专家团鉴定，可获得后续指导、孵化机会

2 独立游戏开发者扶持



关注GAD平台

扫描二维码
关注GAD微信公众号

GAD官方网站
gad.qq.com

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

游戏编程算法与技巧

《游戏编程算法与技巧》介绍了大量今天在游戏行业中用到的算法与技术。是为广大熟悉面向对象编程以及基础数据结构的游戏开发者所设计的，本书专注于游戏行业中的实际应用。

Sanjay Madhav采用了独特的与平台框架无关的方法来展示开发，使其能够在任何游戏、任何风格、任何语言或者框架上开发。他展示了涉及2D、3D图形学，物理，人工智能，摄像机等多个方面的技术。

每个概念的展示都使用了伪代码，很容易就能够被C#、Java或者C++程序员理解。这些伪代码都是Madhav在南加州大学的游戏编程课堂中提炼出来并验证过的。回顾每一章的课后习题能够帮助验证巩固所学的内容，助你继续前进。

Madhav最后详细地以两款完整的游戏作为总结：一款是2DiOS卷轴游戏（使用Cocos2D框架及Objective-C语言开发的）；另一款是3D PC/Mac/Linux塔防游戏（使用XNA/MonoGame框架及C#语言开发的）。这些游戏使用了许多本书所讲的算法与技术，而全部源代码都能在gamealgorithms.net下载。

Sanjay Madhav 从2008年起，在南加州大学教授多门游戏编程相关的课程。在加入南加州大学以前，他曾经在多家大型游戏公司工作过，包括南加州大学Electronic Arts、Neversoft和Pandemic Studios。他的作品包括《荣誉勋章：太平洋突袭》《Tony Hawk's Project 8》《指环王：征服》《The Saboteur》。

本书涵盖以下内容

- 游戏时间管理、速度控制、如何在不同硬件上保持一致
- 现代手机游戏用到的核心2D图形技术
- 3D游戏用到的向量、矩阵及线性代数
- 3D图形包括坐标系变换、材质与着色、深度缓冲
- 处理现在大量使用的数字信号与模拟信号输入设备
- 声音系统，包括声音事件、3D音效及数字信号处理
- 游戏物理基础，包括碰撞检测及数值积分
- 各种摄像机：第一人称、跟随、曲线等
- 人工智能：寻路、状态机、战略规划
- 用户界面，包括菜单系统及平视显示器
- 脚本及基于文本的数据文件：什么情况下应该采取哪种方案
- 网络游戏基础，包括协议及网络拓扑

Pearson
www.pearson.com

策划编辑：张春雨
责任编辑：付　睿

上架建议：游戏算法

ISBN 978-7-121-27645-3

定价：89.00元