CENGAGE Learning

游戏编程精粹 7

GAME PROGRAMMING Gems 7

[美] Scott Jacobs 编
项周臻 陶绍斌 等 译
沙鹰 陈虎 等 审

CD-ROM

人民邮电出版社
POSTS & TELECOM PRESS

图书在版编目（CIP）数据

游戏编程精粹. 7 / (美) 雅各布斯 (Jacobs, S.) 编
; 项周臻等译. -- 北京 : 人民邮电出版社, 2010.7
ISBN 978-7-115-22914-4

Ⅰ. ①游… Ⅱ. ①雅… ②项… Ⅲ. ①游戏—应用程
序—程序设计 Ⅳ. ①G899

中国版本图书馆CIP数据核字(2010)第078184号

版 权 声 明

游戏编程精粹 7

◆ 编　[美] Scott Jacobs
　译　项周臻　陶绍斌　等
　审　沙　鹰　陈　虎　等
　责任编辑　傅道坤

◆ 人民邮电出版社出版发行　北京市崇文区夕照寺街 14 号
　邮编　100061　电子函件　315@ptpress.com.cn
　网址　http://www.ptpress.com.cn
　北京隆昌伟业印刷有限公司印刷

◆ 开本：787×1092　1/16
　印张：28.5　彩插：4
　字数：680 千字　2010 年 7 月第 1 版
　印数：1－4 000 册　2010 年 7 月北京第 1 次印刷
　著作权合同登记号　图字：01-2009-3796 号
　ISBN 978- 7-115-22914-4

定价：69.00 元（附光盘）

读者服务热线：(010)67132692　印装质量热线：(010)67129223
反盗版热线：(010)67171154

游戏编程精粹 7

GAME PROGRAMMING Gems 7

[美] Scott Jacobs 编
项周臻 陶绍斌 等 译
沙鹰 陈虎 等 审

人民邮电出版社
北京

内容提要

本书是游戏编程精粹系列的最新一本，内容涉及通用编程、数学和物理、人工智能、音频、图形学、网络和多人游戏、脚本和数据驱动系统等内容，具有较强的先进性和通用性。随书附带光盘中提供了本书的源程序、演示程序以及需要的各种游戏开发的第三方工具。

因此，无论你是一个刚刚起步的游戏开发新手，还是资深业界专家，都能够在本书中找到灵感，增强洞察力及开发的技能。将书中介绍的开发经验和技巧应用于实际项目中，将缩短开发时间，提高效率。

前　言

游戏编程精粹丛书出到现在已经有6卷了。每一卷都包含许多有用且实际的思路和技术。根据网上的讨论、好奇读者的期望、业余以及专业游戏开发者的咨询，我相信前作起到了让游戏更富有创意，更富有娱乐性和更加令人满意的促进效果。在第7卷里面，我们投入了大量努力来继承这一传统。

游戏开发的激情

游戏开发正越来越成为一个充满幻想的历程。它是一个真正的能让激情与才华闪耀的知识精英的竞技。学位和经验也许能让你进入这个大门，但最终还是要看结果。你的代码容易维护吗？它的效率达到甚至超过目标了吗？视觉和听觉效果令人震撼吗？游戏玩法有趣吗？在这些方面超越别人的挑战毫无疑问让游戏开发令人兴奋，我猜想这也是激发书中文献作者们分享想法和经验的动机。我希望同样是这种向往超越的渴望把你引向这本书，因为这一卷的意义，同时事实上也是这整个系列的意义，就是提供给你提供工具和灵感来做到这一点。

没有几个行业能够像游戏业这样，开发者工作的激情如此之高，以至于在经历了5天的工作后，专业开发者们还会与业余爱好者聚集一起，为了周末的“果酱”继续做和工作日一模一样的事情。也许我在电视上看到的伐木工竞赛和这有点近似。但是有多少伐木工会在家里安排一个伐木计划，却只是为了尝试一些新点子来获得乐趣和经验呢？

专业领域知识的必要性意味着通常游戏开发者被划分到专门的角色上：图形程序员、人工智能程序员之类。这本书的章节也毫无疑问能反映出各方向之间常用的分界线，尽管我必须尊重一些抱怨有个别分类并不总是完全正好属于某一个方向的意见。在我希望专注于某一领域的人能在这些文献里找到他们感兴趣的东西的同时，我十分兴奋于这本书覆盖范围和专业技能的广泛多样，能够展现在对每个方向都充满激情的游戏开发者面前。我希望图形程序员能够阅读音频的文献，反之亦然！

想做游戏的渴望

业内游戏开发者的热情也许可以帮助解释为什么那么多人殷切地想加入游戏开发者行列。虽然独立自学的“勇士”也能踏进游戏行业的大

门（有时候他们甚至制造出他们自己的传奇大门），但对于想把游戏开发作为第一份职业的人来说，合格的教育越来越多的能起到帮助作用。除了传统的数学和计算机科学教育路线和大量高质量对高级出版物的介绍外，全世界的中学和大学里面的专门游戏的开发学位、课程可供使用，有些还是与专业开发工作室紧密合作而设立的。已发行游戏的多种组成都可以被修改，这提供了前所未有的进入使用最前沿技术，价值好几百万美元的游戏引擎，并扩展你的游戏体验或者原型库的机会。不仅如此，对大部分题材的游戏，你都能在高质量的开源游戏或者引擎里面找到灵感、实验方法和扩展方法。

拥有一定量非游戏相关软件开发经验的同时充满热情的业余人士，能够参与游戏开发的机会也相当多的。我们可以使用他们。随着游戏设计、目标硬件和开发团队自身正变得越来越大而且复杂，游戏业继续欢迎来自其他软件开发行业的各种新鲜想法。你的开发团队里面有数据库管理员（DBA）吗（小点声，MMO 开发者们）？在这些人里面，你能找到会给出把你的对象系统整合到关系数据库的天才。我们有一个网络达人，他能把一些网络管理员用的工具用到多玩家模块的开发上，但是在业界这种做法还没有流行起来。当认识到广义领域软件开发的趋势和成功后，游戏开发团队正越来越多地引入正式的项目管理和产品方法论，比如敏捷和 Scrum，在这些领域我们可以受益于游戏业外的同行们。做游戏和做文字处理软件不同，但是优秀的管理日益增长的团队的方法、增强团队内部交流的途径和管理客户（发行商）关系的方案，一定与业外同行们的解决方案相似，因为他们也遇到同样的问题。多核平台的移植，不论是 PC 还是今天的家用机平台，要求开发者超越传统 C/C++编程语言才能解决并行和同步的问题，我们正积极的从精通 Haskel 或者 Erlang 的程序员身上找出能借鉴的经验。

向往快乐的激情

游戏的吸引在于它们充满挑战、恶搞和娱乐。这本书里面的许多文献都是关于幕布背后的灯光道具脚手架，尽管它们并不是制造快乐的直接因素。一个被重定义者(redefiner)一遍又一遍玩弄的类型与一个在打第一个 boss 前丢掉的战车也许使用的是同一套碰撞检测系统或者 C++到脚本语言的接口。真正产生快乐的是玩游戏的体验。所以，除了解决核心难题的文献，书里还有关于直接影响玩家体验的文章，包括音频处理文献和人机互动。人们总是对新的游戏交互方式充满热情且乐于尝试。近来流行乐队（Rock Band）、吉他英雄（Guitar Hero）、劲舞革命（Dance Dance Revolution）当然还有仍天堂的 Wii 的成功，毫无疑问地显示了这点。新的人机接口让长寿的游戏题材有了新的体验，而且也让那些平时不会被游戏诱惑的人产生了想尝试的冲动，从而给那些人带来全新的娱乐方式，顺便也给我们带

来新的市场份额。我很骄傲这本书里面能有 3 篇文献是关于尚未采用的人机交互的想法，同时也很想知道当这些点子被尝试并进一步完善后会有什么东西出来。

在这充满激情的开发者的世界里，殷切的新手、多种多样的产品需求、对创意且娱乐的游戏玩法与设计的渴望创生了这一卷。让一本书符合所有兴趣的需求是一个很高的目标，但是我自信，也希望你会同意，本书后面的内容会作为自己说话。当你的游戏发行的时候通知我一声，我想试试看。

关于封面

Christopher Scot Roby 制作了游戏编程精粹 7 的封面。这个封面表示游戏开发早期产生内容的步骤。从图片最左边界开始，是原始草图的片段。它随后在 Photoshop 里面会被画得更精致，从而得到了左边的原画稿。右边部分体现出后期步骤中把概念转成游戏资源的过程。Google Sketchup 的照片比对功能被使用在这里，来让几何模型贴近原画。取决于不同的流程，这个几何模型可能随后被直接导出成游戏可以使用的形式，从而大大地加快了把设计概念转到可玩元素的过程！

致　　谢

我必须感谢 Jenifer Niles 和游戏编程精粹 6 的编辑 Michael Dickheiser 能够给我编辑这一卷的机会，同时这也是一个愉快的负担。我还要感谢我的章节编辑们，他们既有归来的老兵也有殷勤的新手。没有这些人以及他们多年的行业经验，我可能早就被众多作者提交的天马行空的文章搞得晕头转向了，毕竟许多文章都是在我的专业能力之外的。我再怎么形容他们的辛苦工作在这些书页上的沉淀都不过分。

另外，我想感谢一下这些人，他们耐心地与我一起努力工作，把这本书以及附带的光盘制作出来。他们是来自 Cengage 的 Emi Smith、Kezia Endsley、Brandon Penticuff 以及虽然我不认识但是一直支持他们的人。

我还得感谢我的妻子 Veronica Noechel。在推迟笼子的清理、不得不关掉电视的每一个夜晚，她给予了充分理解，并承担起了一个人做饭的义务。我也应该同样感谢我的父母，但是我想特别指出，我父亲好多次把他办公的电脑带回家给我使用。坦白地说，它曾经相当贵而且沉重，得分成好几段路才能把它从停车场搬到车上。

译者简介

以下是本书中所有译者的背景简介

沙鹰

8681911@qq.com

目前在腾讯互动娱乐部门工作，之前曾在 EA 上海、EA 加拿大、JAMDAT 加拿大和 UbiSoft 上海工作，毕业于南京大学计算机系，是本次翻译特邀监制。

项周臻

hermitbab@qq.com

毕业于上海大学计算机工程系。8 年网络游戏开发经验，在游戏制作流程管理、游戏质量管理、网络编程、物理、脚本、图形图像、性能优化和人工智能方面有浓厚的兴趣。目前在腾讯公司工作，担任一款网络游戏项目的开发经理兼主程序。本次翻译的主要负责人，翻译第 1 章的第 6～第 10 节以及第 4 章和第 7 章。

Shaobin

Tao shaobin_tao@hotmail.com

Shaobin 分别在北京科技大学、清华大学及美国田纳西理工大学获得机械工程类本科及硕士学位。目前在美国艺电（EA Tiburon）从事家用游戏机平台上的游戏开发。在加入 EA 之前，Shaobin 曾在美国参数化技术（Parametric Technology Corp.）及日本森精机（Mori Seiki）等公司服务。他的兴趣主要有图形、动画、物理编程和计算机辅助设计及制造。翻译第 1 章的第 1～第 5 节以及第 5 章。

周克忠

kezhongseu@gmail.com

EA 上海软件工程师，毕业于东南大学计算机科学与技术专业。校译本书时在 EA 新加坡参与《极品飞车：世界在线》的开发。翻译了附录和彩图，并校对了第 2 章和第 7 章。

匡西尼

179741783@qq.com

毕业于北京大学信息技术科学学院，获微电子学士学位和计算机软件辅修学位。毕业后先后任职于 virtuos 和腾讯互动娱乐。截至发稿时，以及在可预见的将来，仍在从事次世代网游开发。翻译了第 2 章和第 6 章，校对了第 3 章。

陈虎

joecgs@gmail.com

在武汉理工大学获得计算机科学学士学位，目前在该校的图形图像研究室攻读计算机科学硕士学位。专业兴趣包括图形、GPGPU 与多核技术。主要校对人员，校对了第 1 章、第 3 章～第 5 章。

徐成龙

floppy@vip.qq.com

自 2003 年投身游戏业以来，致力于用先进的客户端技术开发高品质的网络游戏。校对了第 2 章和第 7 章。

姚远

20036046@qq.com

在北京大学获得计算机与科学学士及硕士学位。毕业后在美国艺电（EA）从事家用机平台游戏和网络游戏开发二年。目前供职于腾讯互动娱乐从事网络游戏开发。在游戏制作流程管理、人工智能、脚本和物理方面有浓厚兴趣。翻译了第 3 章。

作者简介

以下是本书中所有作者的背景简介。

Dmitry Andreev

Dmitry 是比利时 10Tacle 工作室的 3D 图形和工具软件工程师。之前，他是 Burut CT/World Forge 的首席程序员。Burut CT/World Forge 是 X-Tend 游戏技术的创造机构，这个技术被应用到许多的发行版游戏中，其中包括最近的超级战士、古代战争：斯巴达。17 年前，Dmitry 在 ZX-Spectrum 开始了编程生涯。他是远近闻名的演示程序开发者，而且是两届 Assembly demo party 上 64K intro 比赛的赢家。他还有应用数学与机械的学士学位。

Hyun-jik Bae

在 Hyun-jik Bae 开发出 Speed Game 之后（请看他在游戏编程精粹 5 里面的个人简介），他在 11 岁的时候用 Turbo Pascal 开发了 Boom Boom Car（类似于迷魂车 Rally X）。Boom Boom Car 在电影《你是谁》（一个关于玩家和游戏开发者的爱情故事）里面还被主人公角色提到。现在他是 Dyson Interactive 的一个总监，正在开发一款网络游戏。他的主要兴趣包括设计及实现高性能游戏服务器、可扩展数据库应用、真实渲染和物理模拟，另外还有弹钢琴、打高尔夫球、带老婆儿子旅游。

Tony Barrera

Tony Barrera 是自学成才的数学和计算机图形学研究员。他专长于高性能数学计算，尤其是与计算机图形学相关的高性能数学计算。他的第一篇论文《An Integer Based Square-Root Algorithm》发表在 BIT1993 上，到目前为止，总共已经发表了 20 多篇论文。他曾经是好几个公司在计算机图形和相关领域的顾问。目前，他正与 Ewert Bengtsson 和 Anders Hast 一起开发高性能底层图形算法。

Anatoli Beliaev

Anatoli Beliaev（beliaev@trusoft.com）是一名有着超过 15 年多种开发经验的软件工程师。从 2001 年起，他就作为 TruSoft 的首席工程师负责行

为捕捉 AI 技术的架构。他尤其关注适应性编程（Adaptive Programming）和泛型编程（Generic Programming）以及它们在性能要求苛刻的领域构建高性能、高弹性软件上的应用。Beliaev 先生拥有莫斯科鲍曼国立技术大学（Bauman Moscow State Technical University）的计算机科学硕士学位。

Ewert Bengtsson

Ewert Bengtsson 自从 1988 年起就是乌普萨拉大学（Uppsala University）计算图像分析（Computerized Image Analysis）专业的教授，现在是乌普萨拉大学计算图像分析学科的领导。他的主要研究兴趣是开发图像分析和计算机辅助 3D 成像在生物医学上的应用方法和工具，对高性能图像和可视化算法也很有兴趣。迄今为止，他已经发表过约 130 篇国际性研究论文，培养过约 30 名博士生。他现在是 IEEE 的高级会员，而且也是瑞典皇家工程院（Royal Swedish Academy of Engineering Sciences）的院士。

Jacco Bikker

Bikker 是荷兰雷达应用科学大学（University of Applied Sciences, Breda, the Netherlands）国际架构和设计（the International Architecture and Design）课程的讲师。在那之前，他在荷兰游戏业工作了 10 年，曾从业于 Lost Boys Interactive 公司、Davilex 公司、Overloaded PocketMedia 公司和 W!Games 公司。在工作以外，他还写过一些文章，内容关于光线追踪（ray tracing）、光栅化（rasterization）、可见性判断（visibility determination）、人工智能和像 Flipcode.com 与 Gamasutra 这样的开发者网站的游戏开发。

Bill Budge

自从 2 岁的时候得到人生第一套积木，Bill Budge 就爱上了搭造东西。15 岁的时候，他发现计算机编程是人类发明的最伟大的积木。从那以后，他人生的工作就是用这些“积木”来搭造更好的积木，其中有 Bill Budge 的 3D 游戏开发工具集（Bill Budge's 3D Game Toolkit）和弹球游戏建造套装（Pinball Construction Set）。他现在正在索尼计算机娱乐美国公司（SCEA, Sony Computer Entertainment, America）工具和技术组搭造游戏编辑器。

Joaquim Bento Cavalcante-Neto

Joaquim Bento Cavalcante-Neto 是巴西西拉联邦大学（UFS，University of Ceará）计算学院计算机图形学科的教授。1998 年他在里约热内卢教皇天主教大学（Pontifical Catholic University of Rio deJaneiro，PUC-Rio）获

得了土木工程博士学位。他曾在康奈尔大学（Cornell University）计算机图形和土木工程方面工作一年。从 2002～2003 年之间，他是康奈尔大学的博士后研究员。在他博士和博士后期间，他主要从事应用计算机图形的工作。他目前的研究兴趣是计算机图形、虚拟现实和计算机动画。他也做一些其他领域的工作，包括数值方法、计算几何、计算数学。他曾是多个政府资助项目的协调人和西拉联邦大学计算机科学专业硕士和博士毕业流程的协调人。

Michael Dawe

大学毕业后 Micheal 进入游戏业的旅程包括 3 年咨询公司的经历、两次跨国搬迁还有一个专门用来完成答辩论文的暑假。从伦斯勒理工学院（Rensselaer Polytechnic Institute）拿到一个计算机科学的学士学位和哲学的学士学位后，Michael 一边去迪吉彭理工学院（DigiPen Institute of Technology）读计算机科学硕士学位一边在 Amaze Entertainment 磨磨牙齿准备进入游戏业。现在 Michael 是 Big Huge Games 的人工智能和游戏逻辑程序员。

Robert (Kirk) DeLisle

Robert (Kirk) DeLisle 从 20 世纪 80 年代早期就开始编程了，他一直对人工智能、数值分析和算法感兴趣。在研究生期间，他为实验室开发了一些应用软件以分析生物分子数据，这些软件已经在国际上被广泛使用。目前，他是一个计算化学家，开发并应用人工智能方法进行计算机辅助药物设计和化学信息学研究。他是很多期刊的作者，同时是计算化学和药物开发领域很多专利的合伙发明人。

Michael Delp

Michael 是西雅图 WXP 公司（WXP Inc.）的首席人工智能工程师，在那里他曾经只用了 4 个月就从零开始构建一个第一人生射击游戏（FPS）的人工智能系统，此系统获得了评论界的称赞。在他的职业生涯里，他做过人工智能、物理、游戏逻辑软件工程师，涉及第一人生射击游戏、运动游戏和车辆人工智能的工作，既有像他目前的所在的小公司也有像 EA 和世嘉（Sega）这样的大公司。他还在游戏开发者大会（Game Developers Conference）上发表过演讲，并在华盛顿大学（University of Washington Extension）教授一个人工智能课程。他在加利福尼亚大学柏克莱分校（UC Berkeley）拿到了他的计算机科学学位。

Carlos Dietrich

Carlos Augusto Dietrich 在巴西圣玛丽亚联邦大学（Federal University of Santa Maria）获得了计算机科学学士学位，然后在格兰德杜苏尔联邦大学（Federal University of Rio Grande do Sul）获得了计算机科学硕士学位。他的研究兴趣包括计算机图形、可视化和 GPU 通用编程。现在他是巴西格兰德杜苏尔联邦大学计算机图形组 3 年级的博士生。

João Dihl

João Luiz Dihl Comba 在巴西格兰德杜苏尔联邦大学获得了计算机科学的学士学位，然后在巴西里约热内卢联邦大学获得计算机科学的硕士学位。在那之后，他取得了斯坦福大学的计算机科学博士学位。现在，他是格兰德杜苏尔联邦大学计算机科学系的副教授。他的主要研究兴趣是计算机图形、可视化、空间数据结构和应用计算几何。现在他在进行的项目包括开发大规模科学可视化的算法、基于点的建模和渲染的数据结构以及利用图形硬件进行通用计算。他是 ACM SIGGARPH 的会员。

Priyesh N. Dixit

Priyesh N. Dixit 是北卡罗来纳大学夏洛特分校（University of North Carolina at Charlotte）计算机学院的游戏研究员，同时也是 Game Intelligence Group （playground.uncc.edu）的一分子。最近他的主要精力放在通用游戏理解和学习工具集（CGUL toolket，Common Games Understanding and Learning toolkit）的开发上。他是 CGUI PlayerVis 和 HIIVVE 工具的设计者和开发者。他在北卡罗来纳大学夏洛特分校获得了计算机科学学士学位，并在 2008 年春季拿到他的游戏设计与开发硕士学位证书。他的兴趣领域包括从玩与测试（playtesting）中学习和理解、建立支持交互式人工智能的工具和开发所有类型的游戏。

Joshua A. Doss

Joshua Doss 的职业生涯开始于 3Dlabs 的开发者关系部门，为增强专业图形开发者的交流制作一些工具和样例。他在软件方面的贡献包括 ShaderGen、一个开源应用。该开源应用能够动态地产生可编程着色器（programmable shader）以仿真 OpenGL 里大部分固定功能和部分第一版本高层 GLSL 着色器。Joshua 现在 Intel 公司工作，与高级视觉计算组的人一起创造世界水平的图形工具和给游戏开发者的样例。

Nathan Fabian

Nathan 是经验丰富的业余游戏开发者，有 12 年在美国桑迪亚国家实验室（Sandia National Labs）卫星项目的工作经验。他曾经常常考虑要不要在合适的时候加入游戏业，但是从来没有踏入过这个圈子。超过 20 年来，他改进过许多游戏技术，但是始终无法决定他最喜欢的是图形特效、物理模拟还是人工智能。最后，他真的喜欢做一个把 3D 音效作为关键元素的游戏。那个时候，他就能完成他在新墨西哥大学（University of New Mexico）计算机科学专业的硕士学位了。

Marcus Aurelius Cordenunsi Farias

Marcus Aurelius Cordenunsi Farias 2004 年毕业于巴西圣玛丽亚联邦大学（Universidade Federal de Santa Maria）大学计算机科学专业。2006 年，他在计算机图形方向获得了巴西南大河联邦大学（UFRGS，Universidade Federal do Rio Grande do Sul）的硕士学位。去年，他在 Zupple Games（一个巴西的创业公司）参与开发一种用于休闲游戏的新交互技术。他在开发游戏的新交互形式上有一些经验，包括计算机视觉和噪声检测技术。目前，他在巴西格兰德杜苏尔阿雷格港（Porto Alegre, Rio Grande do Sul）的 CWI Software 公司工作。

Mark France

Mark 最近拿到了计算机游戏技术专业的学士学位。他还是独立游戏开发工作室 Raccoon Games 的合作创始人。

Ben Garney

在 GarageGames 还是一个只拥有 8 名雇员和 2 个房间的办公室的时候，Ben Garney 就在那里工作了。他就坐在过道里，给 Torque 游戏引擎（TGE，Torque Game Engine）写文档。那以后，他围绕 Torque 系列引擎做了不少事情，包括图形、网络、脚本和物理。他还参与了几乎 GarageGames 的每一个游戏——最出名的要数彩球闯天关终极版（Marble Blast Ultra）和 Zap 了。最近，他正在为一个角色（Avatar）创建网站学习 Flash 和 PHP。在他的空余时间，Ben 喜欢弹钢琴、爬山以及寻找和他毛茸茸的猫咪 Tiffany 一起和谐共处的方法。

Julien Hamaide

8 岁时 Julien 开始在 Commodore 64 上编写文字游戏，随后是他的第一

个汇编程序。他一直是一个喜欢自学的人，阅读了所有他的父母能买到的书。他 21 岁时以多媒体电子工程师（Multimedia Electrical Engineer）的身份从比利时蒙斯理工学院（FPMS，Faculté polytechnique de Mons）毕业。在 TCTS/Multitel 从事了两年语音和图像处理工作后，他现在是比利时 Elsewhere Entertainment 旗下 10Tacle 工作室的次时代平台首席程序员。Julien 已经多次在游戏编程精粹系列和 AI Game Programming Wisdom 系列上投稿。

Anders Hast

从 1996 年开始，Anders Hast 就是（University of Gävle）计算机科学专业的讲师。2004 年，他以一篇关于计算机图形学基础算法的计算效率的论文获得（Uppsala University）的博士学位。至今他已经在那个领域发表过超过 20 篇文章。他现在作为高级计算科学的可视化专家以兼职方式在（Uppsala Multidisciplinary Center）工作。

Jeremy Hayes

Jeremy Hayes 是 Intel 高级视觉计算组的软件工程师。在加入 Intel 之前，Jeremy 是 3Dlabs 开发者关系组的成员。他的研究兴趣包括程序内容生成（尤其是地形）、独立游戏设计和道奇蝰蛇（Dodge Vipers）跑车。

Scott Jacobs

Scott Jacobs 自从 1995 年就进入游戏业工作了。现在，他是 Destineer 的高级软件工程师。之前，他在一系列游戏公司工作过，包括虚拟英雄（Virtual Heroes）和两个育碧（Ubisoft）的工作室，其中包括 Redstorm Entertainment 和他踏入游戏界的第一个公司 Interactive Magic。另外，他也是游戏编程精粹 6 中网络及多人在线部分的编辑。他和妻子住在北卡莱罗纳州一个充满生物的房子里。

Thomas Jahn

Thomas Jahn 自从 2002 年开始在不来梅专业大学（Hochschule Bremen）学习多媒体工程。在学习的同时，他从 2005 年开始在 KING Art Entertainment 工作，参与一个回合制策略游戏的开发。由于被六边形贴片的特性迷惑，他决定毕业论文就做六边形网格。在 2007 夏天拿到计算机科学的学位之后，Thomas 收到了 KING Art, Bremen 的全职工作邀约（offer）。在那里，他现在正在开发一款冒险游戏 Black Mirror 2。

Alberto Jaspe

Alberto Jaspe 1981 年出生于西班牙拉科鲁尼亚（La Coruña）。他对计算机图形的兴趣起源于 15 岁时一个行走的示例场景。在拉科鲁尼亚大学学习计算机科学专业期间，他在RNASA-Lab开发了3D医疗影像可视化系统。2003 年起，他作为软件工程师和研究员在 VideaLAB 的计算机图形组工作，参与不同的项目和期刊，涉及范围从地形可视化到虚拟现实和渲染。

Mark Jawad

Mark 从 2 年级就开始编程了，而且从来没有想过停下来。21 岁的时候开始了他在游戏业的生涯，然后很快定位于任天堂（Nintendo）的硬件和软件平台。在洛杉矶（LosAngles）的 8 年时间内他编写了很多游戏、工具和类似于 N64 的系统引擎、Game Boy Advance、GAMECUBE 以及任天堂 NDS。2005 年他搬家到雷德蒙 （RedMond），加入任天堂美国开发者支持小组，担任技术带头人（Technical Leader）。在空余时间，他喜欢学习编译器和运行时系统，还有和家人在一起度过时光。

Krzysztof Kluczek

Krzysztof 从 10 岁开始就对游戏编程感兴趣了。随着 3D 技术越来越多的被应用到游戏中，他越来越对 3D 游戏的图形方面感兴趣。他在格但斯克工业大学（Gdansk University of Technology）拿到了计算机科学的硕士学位。现在他正在那里攻读博士学位，开心地学习新技术、制作游戏，以及利用空闲时间参与 gamedev.pl 社团。

David L. Koenig

David L. Koenig 是西雅图（Seattle）The Whole Experience, Inc 的高级软件工程师。他的主要关注点是网络和资源管理的代码。同时他也是华盛顿大学（University of Washington）网络和多玩家编程课程的教师。David 已经是第二次作为游戏编程精粹系列的作者了。他做过很多款大作，包括 SceneIt? Lights、Camera、Action(Xbox 360)、格雷彩弹锦标赛（Greg Hastings' Tournament Paintball Max'd(PS2)、Tron 2.0 (PC)，芝加哥执法者（Chicago Enforcer，Xbox)等。他有休斯顿大学（University of Houston）的计算机科学学士学位。他的个人网站是 http://www.rancidmeat.com。

Adam Lake

Adam Lake 是 Intel 软件解决方案小组的图形软件架构师，领导一个图

形软件开发包的开发。在他进入 Intel 的 9 年里，Adam 做过好几个工作，包括研究非真实渲染和发布 shockwave3D 引擎。他设计了一种流编程架构，其中包含了模拟器、汇编器、编译器和编程模型的设计和实现。在进入 Intel 之前，他在北卡莱罗纳大学查佩尔希尔分校（UNC-Chapel Hill）获得了计算机图形学硕士学位，然后在美国洛斯阿拉莫斯国家实验室（at Los Alamos National Laboratory）的可计算科学方法小组工作。在 www.cs.unc.edu/~lake/vitae.html 可以找到关于他的更多信息。他在计算机图形方向发表过多篇文章，参与 SIGGRAPH、IEEE 的论文和几本关于计算机图形图书的审稿，还有超过 35 项计算机图形和计算机架构上的专利或正在申请的专利。工作之外，他玩山地自行车、公路自行车、远足、露营、疯狂地读书、滑雪，但是却不常常开车。

Dimitar Lazarov

Dimitar Lazarov 是 Luxoflux（Activision 旗下的工作室）的高级软件工程师。他在游戏业有超过 10 年的工作经验，参与过许多种游戏，从面向儿童的 Tyco RC、Casper、功夫熊猫（Kung Fu Panda），到大人游戏例如荣誉勋章（Medal of Honor）和真实犯罪（True Crime）。他自己定位为一个对图形、特效、动画、系统库、底层编程和优化充满激情的通用程序员。他还一直有个不断重复地编写他自己的编程语言的梦想。在空余时间，Dimitar 喜欢看书、旅游、运动还有和他妻子在海滩上放松。

Martin Linklater

Martin Linklater 现在是欧洲索尼电脑娱乐利物浦工作室（SCEE Liverpool，UK）的首席程序员。他有 14 年游戏业的从业经验，开发过很多的游戏，包括反重力赛车 HD（Wipeout HD）、反重力赛车（Wipeout Pure）和极速象限（Quantum Redshift）。

Chris Lomont

Chris Lomont 是安阿伯市 Cybernet Systems 公司政府赞助项目的研究科学家。他现在正在为美国国家航空航天局（NASA）研究图像处理，并为国土安全设计硬件、软件来防止恶意软件感染计算机。在 Cybernet，他参与过量子计算的工作，教学高级 C++编程还释放出一些混乱。Chirs 专长于算法与数学、渲染、计算机安全和高性能科学计算。在拿到数学、物理和计算机科学三个学士学位后，他花了好几年在芝加哥（Chicago）写程序。最终他还是离开了芝加哥的视频游戏公司，去普度大学西拉斐特主校（Purdue, West Lafayette）读研究生，结果他拿到了数学的博士学位，同时

顺便学习了计算机科学的研究生课程。他的爱好包括造桥（http://www.hypnocube.com）、运动、爬山、骑车、皮艇、读书、学数学和物理、旅游、陪妻子 Melissa 看电影和向毫无防备的同事丢弹力球（Superball）。他之前发过两篇游戏编程精粹文献。

Jörn Loviscach

自从 2000 年起，Jörn Loviscach 就是不来梅专业大学（Hochschule Bremen，应用科学学院）计算机图形、动画和模拟方向的教授了。在那之前，他是德国汉诺（Hanover ）的计算机技术杂志的代理主编辑。Jörn 还有个物理学的博士学位。自从他返回学术界以来，他在 GPU Gems、Shader X3、Shader X5 和游戏编程精粹 6 上面都发表过文章。附带一提，他已经在 Eurographics 和 SIGGRAPH 这样的会议上，发表或合作发表过好多篇的计算机图形学和交互技术的文章。

Michael F. Lynch, Ph.D.

Lynch 博士有着电子工程的背景，以前做过很多技术工作，年龄稍大以后进入研究生学院学习。研究生阶段，他跨越了学科的差别进入了社会科学领域。今天他是纽约特洛伊（Troy）区伦斯勒理工学院（Rensselaer Polytechnic Institute）游戏与模拟的艺术和科学（GSAS，Games and Simulations Arts and Sciences）这个新专业的教员。在这里，他讲授游戏的历史和文化以及即将开始的一个游戏中人工智能的课程。在他的课余时间，他做一点美食，玩奇怪的电子音乐还有偶尔品尝交互式情节的刺激（当然，不能少了玩游戏）。

José Gilvan Rodrigues Maia

José Gilvan Rodrigues Maia 是巴西西拉联邦大学（UFC，Federal University of Ceará）计算学院的博士生。他也是在西拉联邦大学获得的计算机科学学士和硕士学位。在他的硕士期间，他花了两年时间从事计算机游戏技术的工作，尤其是渲染和碰撞检测。他现在的研究兴趣是计算机图形、计算机游戏和计算机视觉。他已经参与了西拉联邦大学的研究项目工作。

Joris Mans

Joris Mans 拥有布鲁塞尔自由大学（Free University）的计算机科学硕士学位。结合了学校教育的知识和亲手开发示例场景的经验，他能够毕业后开始工作就直接加入游戏业。他现在是比利时 10Tacle 工作室的首席程序员，

摆弄过从家用机（Console）数据缓存优化到数据库后端等的很多东西。

Enric Martí

Enric Martí 在 1986 年进入巴塞罗那自治大学（UAB，Universitat Autònoma de Barcelona）计算机科学学院。1991 年，他以一篇分析手工线条画 3D 物体的论文拿到了巴塞罗那自治大学的博士学位。同年，他成为副教授。现在，他是计算机视觉中心（CVC，Computer Vision Center）的研究员，对文档分析（尤其是网络文档分析）、图像识别、计算机图形、混合实境和人机交互感兴趣。他现在正在研究以小波从网页中抽取文本信息，来在低分辨率网络图像（比如 CIF 和 JPEG）中进行文本分割的技术。除此之外，他还在开发一个用 3D 接口（数据手套和视觉透镜）混合实境的平台。他还是 *Computer & Graphics* 和 *Electronic Letters on Computer Vision and Image Analysis* (ELCVIA)期刊的评审。

Colt McAnlis

Colt "MainRoach" McAnlis 是微软全效工作室（Microsoft Ensemble Studios）的图形工程师，专长于渲染技术和系统编程。他也是新加坡管理大学 Guildhall 分校（SMU's Guildhall School）的副教授，讲授高级渲染和数学课程。在从内华达州拉斯维加斯的高等技术学院（Advanced Technologies Academy）拿到高级学位以后，Colt 获得了得克萨斯基督教大学（Texas Christian University）计算机科学专业的学士学位。

Curtiss Murphy

Curtiss Murphy 开发和管理软件已经有 15 年了。作为一个项目工程师，他领导了数个为陆军、海军、联合政府和军事机构服务的严肃游戏软件的设计和开发。Curtiss 经常在会议上发表演说，旨在帮助政府使用低成本的基于游戏的技术来增强训练。近来的游戏工作包括十几个基于开源游戏引擎 Delta3D（www.delta3d.org）的项目，以及一个给美国海军研究办公室（Office of Naval Research）的基于美国陆军的公众事件游戏。Curtiss 有弗吉尼亚理工大学（Virginia Polytechnic University）的计算机科学学士学位，现在弗吉尼亚诺福克市的 Alion Science and Technology 工作。

Luciana Nedel

1998 年 Luciana Porcher Nedel 在瑞士洛桑（Lausanne）的瑞士联邦理工学院（Swiss Federal Institute of Technology）拿到了计算机科学的博士学

位，导师是 Daniel Thalmann 教授。她的计算机硕士学位由巴西格兰德杜苏尔联邦大学（Federal University of Rio Grande do Sul）颁发，计算机科学学士学位由巴西圣保罗天主教传信大学（Pontifical Catholic University）颁发。在她 2005 年学术休假期间，她先花两个月在法国图卢兹大学（Université Paul Sabatier in Toulouse），然后又去位于比利时新鲁汶的鲁汶大学（Université Catholique de Louvain）待了两个月，做交互技术的研究。她是格兰德杜苏尔联邦大学的助理教授。自从 1991 年起，她开始参与计算机动画研究；1996 年开始，她就开始做虚拟现实的研究。她目前的项目包括变形法（deformation methods）、虚拟人模拟、交互式动画和使用虚拟现实设备的 3D 交互。

Ken Noland

Ken Noland 是 Whatif Productions 的程序员，已经做过数个扩展内部引擎的技术和基础结构的项目了。他的主要注意力是音频、网络和游戏玩法，还有维护驱动 Whatif 引擎（称为 WorLd 处理器）的难以捉摸的内容驱动技术的代码。他混迹于游戏业一家又一家公司已经超过 6 年了。不工作的时候，你通常可以发现他一脸茫然地在镇上游荡，思索着怎么对付空余时间这种东西。

Jason Page

自从 1988 年起 Jason Page 就在计算机游戏业工作了，他做过游戏编程人员、音频编程人员、音乐家（音频工程师，内容创作者），现在是索尼计算机娱乐欧洲（SCEE，Sony Computer Entertainment Europe）研发部门的音频经理。这个工作包括管理、设计和编写各种 PS3 音频开发包库文件（PS3 音频库 MultiStream 在全世界被很多开发者使用），包括支持所有 PS 平台的开发者关于音频的问题。当然，如果没有他的团队的辛苦工作，这些都是不可能的——所以，这里也谢谢他们。Jason 以前做音频内容创作者的工作包括音乐和声音特效，涉及游戏彩虹岛（Rainbow Islands）、混乱机车（The Chaos Engine）、感官世界足球（Sensible World of Soccer）、极限滑雪板 2（Cool Boarders 2）和 GT 赛车（Gran Turismo）。尽管他不再给游戏做音乐了，他还是会给索尼欧洲开发者大会（SCEE DevStation conference）做一些片段。最后，他想谢谢他的妻子 Emma，在家里忍受了他常有的“我得回一个开发者的邮件”这类话。Jason 个人一直都责怪笔记本电脑和 WiFi 的发明。

Vitor Fernando Pamplona

Vitor Fernando Pamplona 在 Fundação Universidade Regional de

Blumenau 拿到了计算机科学学士学位。自从 2006 年开始，他在巴西南大河联邦大学（UFRGS，Universidade Federal do Rio Grande do Sul）读博士学位。他的研究兴趣有计算机图形学、敏捷开发和免费软件。他还管理一个叫做 JavaFree.org 的 java 虚拟社区，领导 7 个免费软件项目。

Steve Rabin

Steve 是任天堂（Nintendo）北美的首席软件工程师，研究任天堂次世代系统的新技术、开发工具并给任天堂的开发者提供支持。在加入任天堂以前，Wteve 在西雅图（Seattle）好几个创业公司做 AI 工程师，其中有 Gas Powered Games、WizBang Software Productions 和 Surreal Software。他参与管理和编辑过 AI Game Programming Wisdom 系列书籍、Introduction to Game Development、还有数十篇游戏编程精粹系列的文章。他在游戏开发者大会（GDC，Game Development Conference）上发表过演讲，现在还在主持 GDC 的 AI 圆桌会议。Steve 是 University of Washington Extension（UW-Extension）游戏开发认证项目（Game Development Certificate Program）的指导教师和 DigiPen 理工学院的指导教师。Steve 拥有华盛顿大学（University of Washington）的计算机工程学士学位和计算机科学硕士学位。

Arnau Ramisa

Arnau Ramisa 从巴塞罗那自治大学（Universitat Autònoma de Barcelona）毕业获得了计算机科学学位，现在是人工智能研究院（Institut' Investigació in Intelligència Artificial）计算机视觉和人工智能方向的博士在读学生。他对计算机视觉、增强现实、人机接口感兴趣；当然，还有视频游戏。

Mike Ramsey

Mike Ramsey 是 GLR-Cognition Engine 的首席程序员和科学家。在丹佛教会学院（MSCD）拿到计算机科学的学士学位后，Mike 开始开发 Xbox360 和 PC 的核心技术。他已经发行了多个游戏，其中包括越战英豪（Men of Valor ，Xbox and PC）、Master of the Empire 和数个动物园大亨 2 的产品。目前为止，他已经在游戏编程精粹和 AI Wisdom 系列发表过数篇论文。他还有本即将上市的书，书名是 A Practical Cognitive Engine for AI。在空余时间，Mike 喜欢摘草莓和蓝莓，以及与他很厉害的女儿 Gwynn 打羽毛球。

Graham Rhodes

Graham Rhodes 是北卡莱罗纳州罗利市（Raleigh ，North Carolina）应用研究协会公司东南部子公司（Southeast Division ，Applied Research Associates, Inc）的首席科学家。Graham 是许多严肃游戏的主导软件工程师，包括一系列受资助的为 World Book Multimedia Encyclopedia 服务的教育性小游戏，以及最近的为工业安全和人道主义排雷训练服务的第一/第三人称动作/角色扮演游戏。他目前在开发的软件为模拟和训练提供过程式建模和基于物理的解决方案。Graham 在数本游戏编程精粹系列里面提供过文章，编写了 Introduction to Game Development 里面实时物理的章节。他是 gamedev.net 数学和物理部分的主持人和经常性贡献者，在年度游戏开发者大会（GDC）和其他一些业内活动发表过文章，并且经常参加 GDC 和年度 ACM/SIGGRAPH 会议。他是 ACM/SIGGRAPH 会员、国际游戏开发者协会（IGDA ，International Game Developer's Association)会员和北卡莱罗纳高级学习技术协会（NC ALTA ，North Carolina Advanced Learning Technologies Association)的会员。

Timothy E. Roden

Timothy Roden 是得克萨斯州圣安格鲁市安格鲁州立大学（Angelo State University）的副教授兼计算机科学学院的领头人。他讲授游戏开发、计算机图形学和编程。他的研究兴趣包括娱乐计算，尤其专注于过程式内容创作。他在娱乐计算方面发表的论文出现在美国计算机学会计算机和娱乐（ACM，Computers in Entertainment）期刊、国际性的关于娱乐计算的会议国际高级娱乐技术大会（International Conference on Advances in Entertainment Technology）和微软学术日之游戏开发大会（Microsoft Academic Days on Game Development Conference）上。他也在游戏编程精粹 5 上贡献过文章。在加入学术界以前，Roden 作为图形软件开发人员在模拟工业工作 10 之久。

Rahul Sathe

Rahul P. Sathe 是 Intel 高级视觉计算组的软件工程师，开发下一代高端图形硬件的软件开发工具包。他目前正在为游戏开发者设计高级图形算法。在他职业生涯的早些阶段，他做过 CPU 架构和设计方面的很多东西。他持有印度孟买大学（Mumbai University）的电子工程学士学位（1997）和克莱姆森大学（Clemson University）的计算机工程的硕士学位（1999）。他之前的发表作品都是计算机架构领域。他的研究兴趣包括图形、数学和计算机架构。

Stephan Schütze

Stephan Schütze 现在居住在东京。在那里，他一边在日本动画业和游戏业工作，一边学习日语和日本文化。

Antonio Seoane

Antonio Seoane 是 VideaLAB 的研究员。VideaLAB 是西班牙科伦纳大学（University of A Coruña）的工程、建筑和城市设计的可视化小组。他已经有 10 年经验研究和开发集中在模拟、市政工程、城市设计、文化遗产、地形可视化和虚拟现实方面的实时图形应用。他主要研究兴趣和经验是在地形可视化上，最近几年在研究通用 GPU 编程（GPGPU）和人工智能。

Dillon Sharlet

Dillon Sharlet 是博尔德科罗拉多州大学（University of Colorado，Boulder）数学和电子工程系大四的学生。他已经对计算机图形有多年的兴趣。他喜欢探索新的图形算法。在他的空余时间，他喜欢攀岩和滑雪。

Gary Snethen

Gary 对计算的热情在他接触到计算机以前就开始了。他跟着计算机杂志上的例子自学编程，用铅笔和纸编写和运行他的第一个程序。当他父母不愿意买计算机时，Gary 决定自己做一个。当他父母发现了有用的数字加法器和乘法器的手绘图标时，他们觉得 Gary 不只是个暂时的兴趣，所以就给他买了第一台计算机。Gary 的早期兴趣强烈地集中在游戏和 3D 图形上。12 岁时 Gary 编写了自己的线框渲染器，并在青少年时代花了许许多多的夜晚编写 1 对 1 的调制解调器游戏来和朋友们分享。

Gary 现在是 Crystal Dynamics 的首席程序员，发行过“凯恩的遗产：挑衅”（Legacy of Kain: Defiance）和“古墓丽影：传奇”（Tomb Raider Legend）。Gary 现在的专业兴趣包括约束动力学（Constrained Dynamics）、高级碰撞检测、基于物理的动画和在游戏中增强角色逼真度的技术。

Robert Sparks

今年是 Robert 在游戏业音频编程的第 7 年。最近，他是“疤面煞星：掌握世界”（Scarface: The World Is Yours）的技术领队（Radical Entertainment，温哥华，加拿大）。他过去的游戏成就包括辛普森一家横冲直撞（The Simpsons: Hit & Run）、辛普森家庭赛车（The Simpsons: Road Rage）、绿巨人浩克（The

Hulk）、俄罗斯方块（Tetris Worlds）、黑暗天使（Dark Angel）和怪物公司（Monsters, Inc）。

Diana Stelmack

Diana Stelmack 在红色风暴工作室 （Red Storm Entertainment）工作。从 2001 年春天开始，她就参与制作幽灵行动（Ghost Recon）系列。自从这个游戏的 PC 版本发行之后，她的主要开发焦点就是 Xbox Live 在线服务和多人游戏系统的支持。在进入游戏界前，她的网络背景包括电信、IP 安全和美国国防部网络通信。记住，“如果你绊了一下，要假装这是个舞步。”

Javier Taibo

Javier Taibo 1998 年毕业于是西班牙科伦纳大学（University of Coruña）计算机科学专业。他已经进入工程、建筑和城市设计的可视化小组（Visualization for Engineering, Architecture and Urban Design Group，a.k.a. VideaLAB）参与数个计算机图形研发项目。他工作过的领域包括实时 3D 地形渲染和地理信息系统可视化（GIS visualization）、虚拟现实和全景视频和音频。当前，他还在科伦纳大学讲授计算机动画和 3D 交互课程。

Daniela Gorski Trevisan

1997 年 Daniela Gorski Trevisan 从巴西圣玛丽亚联邦大学（UFSM，Universidade Federal de Santa Maria）拿到了信息学学位。2000 年在阿雷格里港她拿到了巴西南大河联邦大学（UFRGS ，Universidade Federal do Rio Grande do Sul）计算机科学（计算机图形方向）的硕士学位。2006 年在比利时法语鲁汶大学（UcL ， Université catholique de Louvain）她拿到了应用科学的博士学位。现在，她是巴西国家科技促进委员会（CNPq）的研究员和巴西南大河联邦大学信息技术学院计算机图形组的成员。她的研究兴趣集中在开发和评估新式交互系统，包括多模式的增强的混合实境的技术。

Iskander Umarov

Iskander Umarov （umarov@trusoft.com）是 TruSoft（www.trusoft.com）的技术总监。TruSoft 开发动作捕捉的人工智能技术，并且向其他公司提供 AI 中间件和顾问服务以帮助他们在游戏和模拟应用中实现创新的 AI 解决方案，应用平台包括 PC、PS2、PS3 和 Xbox360。Umarov 先生创造了 TruSoft

的 Artificial Contender 人工技术的原始思想。Artificial Contender 为游戏和模拟应用提供动作捕捉的 AI 助手。这些 AI 助手能够捕捉人类的动作，学习并且适应人类的动作。Umarov 先生目前负责管理 TruSoft 的 AI 解决方案的开发，同时领导 TruSoft 的研发工作，包括与索尼、EA 和洛克希德马丁公司（Lockheed Martin）的合作项目。Umarov 先生拥有莫斯科国立金属合金学院（Moscow Technological University）应用数学的学士学位和计算机科学的硕士学位，专长于基于实例的学习方法和图论。

Enric Vergara

Enric Vergara 是西班牙巴塞罗那自治大学（Universitat Autònoma de Barcelona）的计算机工程师，最近拿到了庞比犹法布拉大学（Pompeu Fabra University）视频游戏创造专业的硕士学位。他现在是 GeoVirtual 的 C++程序员。

Creto Augusto Vidal

Creto Augusto Vidal 是巴西西拉联邦大学（Federal University of Ceará）计算学院计算机图形学科的副教授。1992 年他拿到了伊利诺伊大学香槟分校（University of Illinois at Urbana-Champaign）土木工程的博士学位，并于 1992～1994 年间，在伊利诺伊大学香槟分校机械与工业工程学院从事博士后研究。他现在是瑞士洛桑理工学院（EPFL，École Polytechnique Fédérale de Lausanne）VRLab 的访问研究员。他目前的研究兴趣是计算机图形学、虚拟现实应用和计算机动画。他曾经是数个政府资助的项目的协调员，这些项目研究基于网络的虚拟环境在培训和教育上的应用。

Jon Watte

除了作为严肃虚拟世界平台公司 Forterra Systems 的首席技术官（CTO）这份工作，Jon 还是独立游戏开发社区经常性的贡献者。作为微软 DirectX/XNA 最有价值专家（MVP）和独立游戏网站 GameDev.Net 上多玩家与网络论坛的主持人，他喜欢分享他在这个行业的经验以及他早年发布的作品：一个基于磁带的 Commodore VIC 20 主机平台的游戏。在带领 Forterra Systems 以前，Jon 参与制作过 There.com、BeOS，还有 Metrowerks CodeWarrior。

G. Michael Youngblood

G. Michael Youngblood 博士是北卡罗莱纳大学夏洛特分校（University

of North Carolina at Charlotte）计算机科学系的副教授、游戏和学习实验室（Games+ Learning Lab）的副总监和游戏智能小组（Games Intelligence Group）的带头人。他的工作内容是研究怎么让人工智能代理和真实人类在虚拟环境里交互，包括计算机游戏和为理解学习行为的元素和模式而进行的高保真模拟，旨在开发出更好的人工智能代理。自从 1997 年以来，他参与开发过实时计算机游戏、智能环境和机器人。他的研究兴趣是交互式人工智能、娱乐计算和智能系统。

目　录

第1章　通用编程

第 2 章 数学和物理

第3章 人工智能

第 4 章 音频

第5章 图形学

第 6 章 网络和多人游戏

第 7 章 脚本和数据驱动系统

第 1 章

通用编程

简　介

Adam Lake，图形软件架构师，高级视觉计算（AVC）小组，英特尔
adam.t.lake@intel.com

游戏开发在每个方面都持续地发生着重大的变化。从以前的几代硬件中提取性能意味着重点要放在调度汇编指令上，利用小矢量指令（如SSE），以及确保被处理的数据保留在寄存器或高速缓存中。虽然这些问题仍然和当代游戏开发相关，但相对于在现代游戏机和个人计算机上利用多线程硬件，这些问题已经不再那么重要了。为此，我们收录了让游戏程序员利用与这些新硬件相关的工具和技术的文章，它们分别是：Michael Ramsey 的“一个多平台线程引擎的设计与实现”，Julien Hamaide 的一个“多线程任务和依赖系统”的实现和 Mark Jawad 的一个“推迟函数调用的唤醒系统”。自《游戏编程精粹》之后，这些问题变得更为普遍。

系统类有 3 篇文章，其中两篇系统软件的文章包括 Dimitar Lazarov 的“高性能堆分配器”和 Colt McAnlis 的“使用年龄和成本指标的高效率缓存替换”。Martin Fleisz 也给我们带来了一篇关于“高级调试技术”的文章。Martin 涉及了关于异常处理、堆栈溢出和内存泄漏的问题（在开发应用程序时，我们都会遇到的问题）。

在这一版中，我们也有一系列令人兴奋的、有关游戏用户界面的文章。Carlos Dietrich 等有一篇关于即时战略游戏的、基于线条界面的文章。Arnau Ramisa 等写了一篇名为“用网络摄像头玩的视频游戏的光流”的文章。还有，在“第一人称射击游戏的脚步导航技术”中，Marcus C. Farias 等描述了一个新的第一人称射击游戏界面。最后，Thomas Jahn 和 Jörn Loviscach 展示了一篇精彩的、使用六边形贴片而不是传统的方形网格的、题为“给蜜蜂和游戏玩家：如何处理六边形贴片”的文章。

所有这些作者都给我们带来了他们自己的独特视角、个性和丰富的技术经验。我的希望是你将受益于这些文章。当你也有精粹需要和大家分享的时候，这些作者会激励着你去这么做。现在，你就尽情地去阅读这些文章吧。

1.1　使用年龄和成本指标的高效率缓存替换

Colt "MainRoach" McAnlis，微软 Ensemble 工作室
cmcanlis@ensemblestudios.com

在内存有限的游戏中，自定义的媒体缓存被用来扩展场景的数据量，同时只使用很小的内存区域而不是一次性将所有媒体装入内存。使用缓存系统时，最困难的因素是：当缓存填充达到其上限时，选择适当的替换页面（victim page）来腾空。当缓存未命中时，页面替换算法的选择至关重要——这种选择和你游戏的硬件内存使用的性能与效率直接相关。不好的算法往往会破坏游戏的性能，而实现得好的算法在不影响其性能的同时，还能成倍地提高游戏质量。流行的缓存替换算法（如最近最少使用算法，LRU）在它们的目标环境下工作得很好，但在需要更多的数据来准确地选择替换页面的情形下，它们往往显得力不从心。本精粹将介绍年龄和成本指标可以作为评估值，应用于构建最符合游戏需求的高速缓存替换算法。

1.1.1　概述

当从主存请求数据时，操作系统将读出的数据放到一个临时的内存区域中（称为高速缓存或缓存，cache），它可以以比主存更快的速度进行存取。高速缓存本身有一个预定义的尺寸，它被分割成较小的称为页面的内存集合。当内存存取时，缓存本身会被填满，此时，操作系统必须从缓存中选择一个页面来存放新来的页面数据。这种情况被称为高速缓存未命中。当需要的页面数大大超过缓存的大小时（通常为 2 倍或更多），缓存抖动（thrash）就发生了，也就是说，为了给新来的整个数据集腾出空间，整个缓存都将被丢弃。缓存抖动被认为是任何缓存算法的最坏情况，也是任何缓存替换性能测试的焦点。

存储器处理程序要从主存获取信息时，每次都有相关的消耗。通常是因为采用了额外的、靠近处理器本身的内存芯片，所以从缓存读取的消耗较小。因而，如果发生缓存未命中，在决定清除存储器的哪个页面时，其中的一个主要目标是：选择一个在缓存中并不需要有效槽的页面。例如，如果在故障发生的过程中随机选择一个页面来删除，但是，这个页面被删除后恰好立即被需要，这将引起另外一个从主存重新获取数据的性能开销。我们的目标是建立一个算法，它能最好地描述要从缓存删除的理想页面，以减少缓存未命中和性能负担。

这些类型的算法被称为牺牲（victim）页面确定或页面替换算法，它们是20世纪60年代和70年代的一个热门话题，并在最近最少使用（LRU）算法（以及其他工作集系统）的引入后达到高峰。自那时起，为了解决LRU在特定问题领域使用时的一些问题，产生了这些算法的衍生算法。例如，[O'Neil93]描述的一个被称为LRU-K的LRU分支，被认为能在软件数据库系统中更有效地工作。自适应替换缓存（ARC）是IBM开发的一个算法，它被使用在硬件控制器以及其他流行的数据库系统中[Megiddo03]（见本精粹末的“参考文献”，以获取更多信息）。

在游戏开发中，程序员必须和硬件和软件层的缓存打交道，尤其是在游戏机领域，程序员不停地努力增加游戏内容，同时还得满足内存限制。如同许多其他的为解决一个具体问题而定做的替换算法，有些共同的游戏和图形系统需要一个更类似于内存模式和使用模式的替换系统。本精粹将介绍两种缓存页面指标，它们可以以更好地融入视频游戏开发环境的方式交织在一起。

1.1.2 缓存替换算法

缓存替换系统自产生起就是一个活跃的研究领域，由此产生了大量的被自定义调整来解决问题空间下各种情况的不同算法。为了有一个参照基准，这里会涉及一些最常用的算法。

Belady的Min（OPT）

最有效的替换算法将永远是取代那些被缓存逐出后，将在最长的一段时间不再需要的页面。在工作系统中，执行这一类算法需要预先知道系统的使用情况，而这是不可能确定的。在已知随时间变化的输入的测试情形下，执行OPT的结果可用做测试其他算法的基准。

在线程之间有生产者和消费者结构的多线程环境中，如果生产者线程比消费者线程超前好几帧，就有可能使用生产者的信息来得到接近于OPT的结果。

最近最少使用（LRU）

LRU算法取代在最长的时间内还没有使用的页面。当一个新的页面加载到缓存中时，按每页来保存描述了给定页面有多久没被使用的数据。一旦缓存未命中，被替换的页面就是那个在最长的时间跨度内未曾被使用的页面。LRU做了一些很酷的事情，但很容易过度抖动。也就是说，在缓存中，你将始终有一个最旧的页面，这意味着除非你很小心，否则当工作量很大时，实际上可能会清除所有缓存。

最近使用的（MRU）

MRU替换刚刚被替换的页面，也就是说缓存中的最新页面。MRU将不会取代整个缓存，在严重抖动时，它将永远选择去取代同一个页面。虽然没有像LRU那么流行和鲁棒，但MRU有它的用途。

在[Carmack00]，John Carmack列出了一个不错的、介于LRU和MRU的纹理缓存页面替换的混合系统。简而言之，它的使用形式是，大部分时间内用LRU，当达到每一帧都必须驱逐一个条目（entry）时，就切换到MRU替换策略。如前所述，LRU的问题是，如果没有

足够的可用空间，它将可能抖动整个缓存。在缓存开始抖动的那一点，系统就切换到 MRU，并在内存中创建一个页面暂存区而让大多数缓存安然无恙。如果在各帧之间使用的纹理数量相对较低，这可能是有用的。当用户处理额外需要的页面是可用缓存大小两倍的情形时，此方法就退化了，这可能使渲染过程停止。

不经常使用的（NFU）

NFU 页面替换算法改变了访问启发式，为缓存中的每个页面保存一个访问计数器。任何在当前时间间隔内被访问的页面，它们的计数器（从零开始）将增加 1，其结果是一个关联页面有多频繁地被使用的数字限定词。然后，要替换一个页面，必须在当前时间间隔内寻找那个有最小计数器的页面。

这个系统最严重的问题是：计数器指标不跟踪访问模式。也就是说，一个在加载时被频繁使用、而以后再也没被用的页面，可能和在同一时间间隔内每隔一帧就被使用的页面有相同的数量。区分这两个使用模式所需的信息是无法从一个单一的变量得到的。下一小节中提出的年龄指标是一个改变了如何表达计数器的 NFU 的变形，于是，在抖动时，用户就可以从它获取额外的信息。

如果需要一个更综合的算法清单，在你最爱的搜索引擎中搜索“Page Replacement Algorithms”（页面替换算法），或者参阅你在大学用过的操作系统教科书。

1.1.3　年龄和成本指标

为了说明这一点，本精粹的剩余部分参照了在当代硬件上大多数游戏面临的一个问题，那就是设计一个自定义的纹理缓存。要做到这一点，用户将保留一个静态的一维数组缓存页面，可以用来加载和卸载数据。假设用户正在使用一个多维纹理缓存，即放在缓存的数据来自纹理，因为各种大小的多纹理可以容纳在一个单一的缓存页面中，所以从这个意义上讲，缓存是多维的。例如，如果缓存页面可容纳一个 256×256 的纹理，那么，你还可以支持 4 个 64×64 的纹理、16 个 32×32 的纹理等，包括可以和谐地存在于同一个页面的每个尺寸的倍数。

即使在这个简单的例子中，你也已经奠定了让标准替换函数表现不佳的基础。考虑多维缓存的情况，你需要插入一个新的 256×256 的页面到一个缓存，而这个缓存只是用一些 32×32 的纹理完全填满。简单的 LRU/MRU 方案不具备必要的数据来合适地计算哪个缓存页面是最佳替换页面。同时，因为访问模式在很大程度上远不止依赖于页面最后被替换的时间，所以它也不能合适地计算哪组 32×32 的纹理需要被清空。因此，在这种情况下，为了更好地分析最佳替换页面，本文提出了一套新的替换指标。

年龄算法

OPT 算法知道一个页面在缓存中的使用数量，并替换那个在最远的时间才会被使用的页面。大多数替换算法尽其所能地用各种数据访问模式来效仿这一算法。为了最好地预测未来的用法，年龄算法通过保留一个在前几帧中使用情况的概念来模拟这个过程。也就是说，你需要掌握，在一个时间窗口内一个页面被访问了多少次。为了完成这项任务，在缓存中的每

个页面保留一个 32 位的整数变量，当一个页面首次进入缓存时，此变量被初始化为 1（0x00000001）。

对于每一帧，在缓存中的所有活跃页面左移一位，标志着它随着时间的折旧。如果一个活跃的页面在此帧被使用，那么年龄变量的最低有效位设置为 1，否则设置为 0。这个移位（Shift）和设置模式允许你对过去的 32 帧保留一个使用评估。

例如，一个每两帧被使用一次的页面将有一个年龄变量 0xAAAAAAAA（010101010…01），而一个首次加载时被大量使用但以后再也没有被用的页面，将有年龄变量 0xFFFF0000（1111…1100000…00）。

为表明年龄变量随着时间的变化，在一个 8 帧的时间窗口，考虑一个在第一、第三、第四和第八帧被使用的页面。年龄变量将进行如下改变：

第 1 帧 — 00000001（使用）
第 2 帧 — 00000010（不使用）
第 3 帧 — 00000101（使用）
第 4 帧 — 00001011（使用）
第 5 帧 — 00010110（不使用）
第 6 帧 — 00101100（不使用）
第 7 帧 — 01011000（不使用）
第 8 帧 — 10110001（使用）

有了这样的信息结构，你可以计算出指定窗口的年龄百分比费用（APC）。通过用页面被使用过的帧数（位为 1 的个数）除以年龄变量中的总帧数，得到一个页面被使用及未被使用的平均帧数。这些数据可以用汇编和处理器启发式得到，而不需要高级代码。虽然可以用自己的方式表达这个数据，但是，提出的 APC 作为一个单位化的在[0，1]之间的单一值存在，如“年龄和成本”一部分所述，此值可以作为一个相对于其他指标的标量。

当使用年龄来确定目标替换页面时，你企图选择在一定的时间内还没有被使用的页面，及一般不常使用的页面。例如，在一个时间窗口，每帧都被使用的一个页面将有一个 100%的 APC，它将几乎是不可能被取代的，而一个有 25%的 APC 的页面被取代的机会会较高。

我喜欢年龄变量是因为它有一些隐性的好处。

- 它不利于旧的纹理，迫使纹理去证明它们是被场景所需要的。一旦证明了这一事实，它们就被保留。一旦获得一个 50%以上的 APC，它就会很难从缓存中释放。
- 它建立了一个暂存区。还没有证明它们是有用的新的纹理被转化为暂存区，这是一件好事，因为新的纹理往往是暂时的，在下一帧消失的概率较高。
- 这是一个修改的 NRU（最近没有使用）方法。在很短的时间内，有较高可见频率的纹理，可以很容易地跳到 50%的 APC，但然后再掉下来，在多帧后，它们的 APC 缓慢回落。年龄提供了访问变量的一个修改了的表示法，并允许额外的分析。因此，如果 APC>60%，但纹理在最近的 X 帧没有被使用过，那么可以检查此情况并早些清除该纹理。

迄今为止提出的 APC 变量是强大的，但并非没有毛病——缓存中的多个页面可以有完全不同的访问模式，但具有相同的 APC 值。也就是说，0xAAAAAAAA 和 0xFFFF0000 有相同

的使用比例，但很容易看到，这两个年龄变量的使用模式明显不同。随后的基本年龄变量的二进制数据的分析模式可以帮助区分这些有相似 APC 值的页面（如分析子窗口来得到二级 APC 值），但这也存在类似的问题。

扩展的年龄算法

前面提出的年龄算法假设：你想要保持相当低的用来推断页面信息的内存消耗。因此，在一个 32 帧的窗口，对每帧存储一个使用/未使用的标志。应当指出的是，在所需页面数额相当大的情况下，年龄算法就像任何其他的算法一样退化。例如，如果有在每一帧都被使用的 50 个纹理，但只有 12 个缓存页面来放置它们，这样在缓存中就不会有足够的空间来同时保留整个内存占用，每一帧都将抖动整个缓存，且替换每个页面。

在这种情况下，页面和纹理的不断加载或者重新加载，会使每个页面都有一个设置成 1 的年龄计数器值，因此，这将缺乏任何其他有助于具体替换页面识别的信息。为了帮助解决这一问题，年龄变量可以每帧存储，比只是一个使用/未用位更多的信息，且事实上存储纹理的使用数量。因此，你可以存储一系列数字，每个数字存储该页面在这一帧中被使用的频度，而不是只在一个 32 位的整数变量储存一个 0/1。这将类似于一个[1，18，25，6，0，0，…，1]，而不是 01001010011…1 的清单。在退化的情况下，这一额外的信息特别有用，因为你现在有更多的数据来协助替换页面的识别。

例如，考虑同时加载到缓存的两个页面（TextureA 和 TextureB），TextureA 被使用在场景中 50%的物体上，而 TextureB 只使用在 10%的物体上。在这一点，两个页面有相同的 APC 值，但很显然，你可以确定这两个页面有着非常不同的使用数量。当必须找出一个替换页面时，应该考虑到，在当前帧 TextureA 被使用了较多次的事实，增加了它将在随后帧也被使用的可能性，因此，较少使用的纹理（TextureB）应该被替代。

通过存储这一额外的每帧数据，你提供了其他的统计分析操作，来帮助确定从缓存清除的最佳页面。

- 用时间窗口内的非零帧的数量除以总帧数，APC 变量仍然可以从扩展的年龄算法得到。
- 在给定的帧窗口寻找最少使用的页面，将找出一般而言最少使用的页面，这将有助于确定替换页面。
- 利用 MAX 分析，你可以确定在窗口内访问最多的页面，来帮助它们避免从缓存中清除。
- 在你的窗口内，找到 AVG 的使用和得到类似于 APC 的第二个简单化了的变量一样简单。

取决于你的实现需要和数据格式，简单的年龄算法或者扩展的年龄算法都是可行的。最好的主意就是坐下来分析你的数据来决定，对你的游戏，哪个是最有效的和最有用的方法。

做替换的成本

大多数替换页面识别算法只使用一个单一的启发式。也就是说，它们的算法是专门针对导致最少的缓存页面未命中的访问模式而量身定做的。例如，LRU 只保留最旧页面的信息。然而，自定义软件缓存常常有涉及缓存未命中的第二个启发式——用新数据填充缓存页面的

成本。对于大多数硬件缓存，都有一个恒定的成本。这个成本与存储器处理程序访问主存及读取所需数据相关。

然而，对于你的软件需要，这个成本通常可以在页面本身之间波动。因此，替换页面识别考虑到，实际上用一个给定内存块来填充一个页面的性能打击量是很明智的。这个性能消耗（或只是消耗）可以有多种来源，它可以用一个外部数据集来手动定义（例如，一个定义哪些纹理被真正使用的 XML 文件），或者它可以用填充页面的实际费用来定义。

考虑到在前面的示例中，传入的纹理页面是通过从光盘驱动器连续传送而生成的。因为较大的纹理有更多的信息要从媒体中传送，而较小的或简单的纹理只是那些消耗的一小部分，所以较大的纹理有较长的、涉及把它们放到缓存的性能时间。在这种情况下，在替换页面识别过程中，考虑涉及有可能取代内存中一个页面的成本将是非常明智的。如果你替换的页面具有较高的相关成本，且接下来的几帧需要该页面，那么将导致不必要的开销。相反，如果替换了一个有较低成本的页面，错误地把它从缓存中删除的性能打击则低得多。概括地说，把成本作为页面替换的一个变量让你可以回答问题“清除 5 个较小的纹理给 1 个大纹理腾出空间会更便宜吗？”

回顾一下，成本可让用户关注将一个给定页面重新装入缓存会如何损害性能。如果需要，这个系统的一个扩展允许替换页面识别功能更加关心缓存未命中的性能成本，而不是帧之间的连贯性。

就成本本身而言，它具有其他缓存替换算法的相同问题。当抖动发生时，你在缓存中找到并清除最便宜的纹理。因为总有一个便宜的页面存在，如果负荷足够大，整个缓存可能抖动。该算法也有这样的问题：它可以把非常昂贵的页面无限期地留在缓存。如果像天空盒纹理这样的东西被装载到缓存，那么这是一个很好的特性，因为天空盒在每一帧都将是活跃的，而且由于它的大尺寸，我们不太可能想把它从缓存中删除。在大多数情况下，这是一个糟糕的特征，需要不断地关注。

和其他启发式相结合，成本是一个功能强大的同盟。通过用替换指标来偏向访问模式算法的替换识别，你允许缓存找到一个在页面替换需求和抖动之间的折中。此外，访问模式的识别帮助消除纯粹的成本指标所涉及的问题，当高费用的项目到达已经不再需要的状态时，让它们最终从缓存中清除。

年龄和成本（A&C）

前面的例子假定：在纹理缓存的每个页面有一个相关的 APC 和相对成本（RC），它们在每一帧被更新。本示例假定：RC 是一个更重要的指标，并允许该值是一个没有上限的整数变量。例如，如果通过从磁盘连续传送纹理至缓存，RC 值可能是纹理尺寸除以从媒体连续传送一小部分纹理所花费的时间。把 APC 当作一个在（0，1）之间的单位化的浮点变量。

在一个简单的实现中，用户可以组合这两个值成一个单一的结果，其中，APC 作为 RC 的一个标量，从而使 ThrashCost=RC×APC。总的来说，这表现为一个非常好的、用来识别适当的替换页面的启发式。为了证实这一点，我已经提供了几个 APC/RC 比率的例子及替换模式的说明。对以下的数据，假定 RC 的最高值可能是 10。

APC	RC	TC	模　式
1.0	10	10	此页面的取代是非常昂贵的，因此很难是一个移出的选择。其 APC 为 1.0，这说明在年龄窗口的每一帧，该页面都被使用。在这一点上，这个页面可以被替换的唯一方法是当它被全缓存清除所强迫，也许你的实现不允许这样
0.2	8	4	本页面有相对较高的 RC，但它的 APC 表明它很少被使用，因此可被视为一个有效的替代。然而，由于 RC 值是如此之高，为了确定这个纹理是不是真正应该被替换，值得在数据的子窗口做第二个 APC 试验
1.0	5	5	此页面处于中间点。这就是说，它的替代相对廉价，但是它的 APC 说它将在下一帧被需要。取代这个页面不会有问题，但由于高 APC，也许值得用额外的搜寻来查找另外一个具有较高 RC 但较低 APC 的页面
0.01	10	0.1	本页面有一个极低的 APC，这指出这样的事实，要么它是刚刚被加入缓存的，要么是不经常使用的。在任何情况下，TC 是如此之低，以至于其他有较低 RC 但有较高 APC 的页面可以轻易地把它从缓存中挤掉。然而，因为 RC 是如此之高，为了确定这个纹理是不是真正应该被替换，值得在数据的子窗口做第二个 APC 试验

正如在此表所见的，使用简单的 RC × APC 值可能会导致页面有不同的 APC/RC 值，但具有相同的 ThrashCost。这将意味着，APC/RC 关系为 0.5/100 的纹理 A，可以和 APC/RC 为 1.0/50 的纹理 B 有相同的成本。这里的问题是你如何确定要取代的页面。从理论上讲，这两个页面的任何一个都是一个有效的目标，都包含相同的潜在替代数值权重。纹理 A 有较高的成本，如果下一帧需要它，替换它将更加昂贵。纹理 B 的成本较低，但有一个 100%的 APC 值，因此立即需要此页面的可能性将很大。

在实践中，我发现：当多个页面返回相同的值时，替换一个有较低 APC 值的效果要好得多。事实上，这就是要使用年龄指标的原因。通过分析使用模式和成本，你可以看到，虽然页面更加昂贵，但是它较少地被使用，它错误地被缓存抖动的概率就较低。在这些情况下，重新扫描缓存来找到一个有较高 ThrashCost 及较低 APC 值的页面是一个好主意。如果没有找到，可以安全地假设，为了缓存，这个页面可能需要被替换掉。然而，取决于你的系统，更好的被替换页面可能会有所不同。

此表中提到了另一个需要讨论的实例。如上所述，A&C 系统有能力引入一个可能成为静态的页面。如果你的缓存包含从每一个镜头角度都可见的纹理，如天空盒纹理或虚拟角色的皮肤贴图，这也可能是一件好事。但是，如果缓存引入太多这样的页面，有效的工作空间就会大规模地缩小，造成更多的缓存未命中和缓存的整体低效率。在这种情况下，对这些静态纹理生成一个单独的缓存也许是明智的。

1.1.4　结论

对任何高性能环境，定制媒体缓存系统是至关重要的。由于外存媒体的使用增加，在游戏环境中，有一个精确的控制模型的需要也增加了。由于旧的替换算法考虑到操作系统的内存管理和硬件内存访问模式而设计，它们缺乏一些关键的、允许它们评估可能存在的更复杂情形的属性。年龄和成本的结合引入了大量的其他信息，而且只需要非常低的开销，这很适合游戏环境并且运作良好。成本指标引入了页面载入的整体性能的概念，对于随时需要的外在系统，页面载入可以成为一个主要瓶颈。年龄指标允许一个更基于每帧的使用模式的视角，这比传统的指标更容易地和游戏模拟的概念相联系，它也包含了足够的、在任何特定环境的临界情形下创建有效替换情形的可用信息。由于缓存替换需求在模拟过程中改变，这也允许

大量的定制和二次分析，来评估最好的替换页面而获得最佳的效果。

利用这一组强大指标的优势是：缓存页面替换性能的全面增加，从而促成一个较低的抖动开销和填充缓存所需费用。在每天结束的时候，这真是你所想要的。

1.1.5 致谢

对于在“扩展的年龄算法”一节的指标，由衷地感谢 Blue Shift 的 John Brooks，以及他让所有这些得以进行的帮助。

1.1.6 参考文献

[Carmack00] Carmack, J. “Virtualized Video Card Local Memory Is the Right Thing,” Carmack .Plan file, March 07, 2000.

[Megiddo03] Megiddo, Nimrod and Modha, Dharmendra S. “ARC: A Self-Tuning, Low Overhead Replacement Cache,” USENIX File and Storage Technologies (FAST), March 31, 2003, San Francisco, CA.

[O’Neil93] O’Neil, Elizabeth J. and others. “The LRU-K Page-Replacement Algorithm for Database Disk Buffering,” ACM SIGMOD Conf., pp. 297–306, 1993.

1.2　高性能堆分配器

Dimitar Lazarov, Luxoflux
dimitar.lazarov@usa.net

本精粹将展示一个创建高性能堆分配器的新技术，它特别侧重于可移植性、完全对齐的支持、低碎片和低的维持开销。此外，它告诉用户如何扩展分配器来得到调试功能以及额外的接口功能，以实现更好的性能和可用性。

1.2.1　简介

有一个共同的看法是：堆分配是缓慢和低效的，它造成从碎片到难以预测的 OS 调用的各种问题，以及对如游戏机等嵌入式系统的其他不良影响。在很大程度上，这是真实的。从历史上讲，这主要是因为控制台制造商没有花费大量的时间或努力去实现高性能的、包括所有支持堆分配的标准 C 语言库。所以，许多游戏开发商建议不要使用堆分配，甚至在游戏引擎的运行组件中彻底禁止它的使用。在需要时，很多团队使用手动调整的内存池，而不幸的是，这是一个持续而艰苦的过程，并且不灵活及容易出错。所有这些都违背了很多现代的 C++使用模式，更具体一点，就是使用 STL 的容器、字符串，智能指针等。可以说，所有这些提供了许多强大的功能，可显著改善代码的开发，所以并不是那么困难地看到你为什么会想两全其美。考虑到这一点，我们的目标是创建一个堆分配器，可以给程序员提供一个内存池的性能特点，但不需要手动修改成千上万行与分配有关的代码。

1.2.2　相关工作

[Lea87]开发的一个流行的开放源码分配器被视为堆分配的基准，它使用一个混合方案，其中，基于请求的分配大小由两个单独的方法来处理分配。小的分配由类似于尺寸组块（chunk）的链表箱来处理，而大的分配由“tries”二叉树结构箱来处理。在这两种情况下，一个“头”结构连同每个请求块（block）一起被分配。在释放操作中，为了确定该块的大小和与周边块的凝聚，这个结构至关重要。非默认对齐的分配用“过分配”（over-allocating）和移动起始地址到正确的对齐来处理。

头结构和过分配这两个因素造成了不良的对齐过度使用模式，这

在游戏编程中是较为常见的。[Alexandrescu01]的一个稍微不同的办法建议：使用每尺寸模板池式分配器。一个池式分配器使用一个大的组块内存，分成较小的由同样大小的块组成的组块，这些块用一个自由块的单链表来管理，通常被称为自由列表。这有一个很好的特性，即不需要头结构，因而具有零内存开销，且它的元素自然得到完美的对齐。不幸的是，在释放时，用户不得不要么执行搜寻以确定此块从哪里来，要么提供一个额外的参数作为块的原始大小。结合先前所述工作的理念，加上我们的小精粹，就得到了我们的解决方案。

1.2.3 我们的解决方案

我们的解决方案采用了一种混合的办法，把我们的分配器分为两个部分，一部分处理小的分配，另外一部分处理其他的。

小分配器

最低和最高的小分配是可以配置的，并分别默认预设为 8 字节和 256 字节，然后，所有在此范围内的尺寸四舍五入到最小分配的最接近的倍数。如图 1.2.1 所示，在默认情况下，这将创建 32 个箱，它们处理的分配大小为 8、16、24、32……一直到 256。

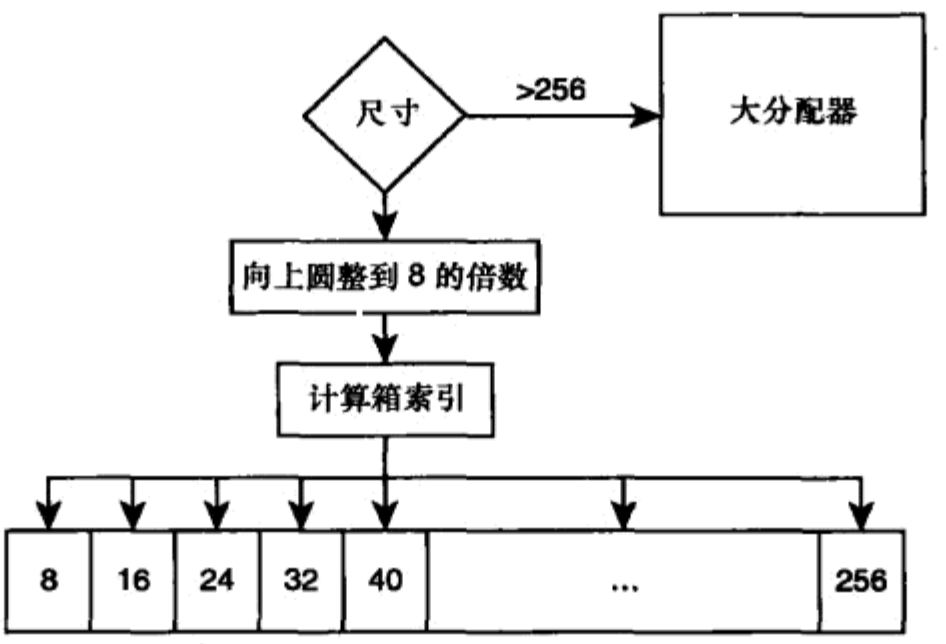


图 1.2.1 基于分配大小选择合适的箱

注意，因为小分配被安排成具体尺寸的箱，所以对整个箱，你可以只为任何尺寸的相关信息保留一次，而不需要对每一个分配都保留。虽然这样可以节省内存开销，但你可能会正确地怀疑：在一个释放操作中，当只给你提供负荷块的地址时，如何正确地找到这一信息。

为了解释这一点，我们需要确认小分配器将如何管理内存。用户可能需要从操作系统分配大块的内存，传统上，这是用池式分配器来做的，但是，让我们在有需求时做这个，以便最大限度地减少碎片和内存的浪费。此外，一旦用户知道当它们将不再被使用时，就把这些同样大小的块（让我们称它们为页面）返回给操作系统，而在其他的地方则需要这些内存。

为了该方法的正确性，这里需要指出的重要事情是：它从操作系统请求自然对齐的页面。换句话说，如果选定的页面大小是 64KB，那么这种方法期望它是 64KB 对齐的。

一旦获得了自然对齐的页面，这个方法就在页面的后边放置管理信息。在那里，它存储一个管理页面内所有元素的空闲链表。一个使用数量决定了页面什么时候是完全空的，而一个箱索引决定了这个页面属于哪个箱。最重要的是，所有属于同一箱的页面用双向链表链接，从而使用户可以方便地添加、删除或安排页面。

拼图的最后一块几乎直截了当——在释放操作中，所提供的负荷地址是和页面边界对齐的，然后用户就可以访问那个空闲链表和其他的管理信息了，如图 1.2.2 所示。

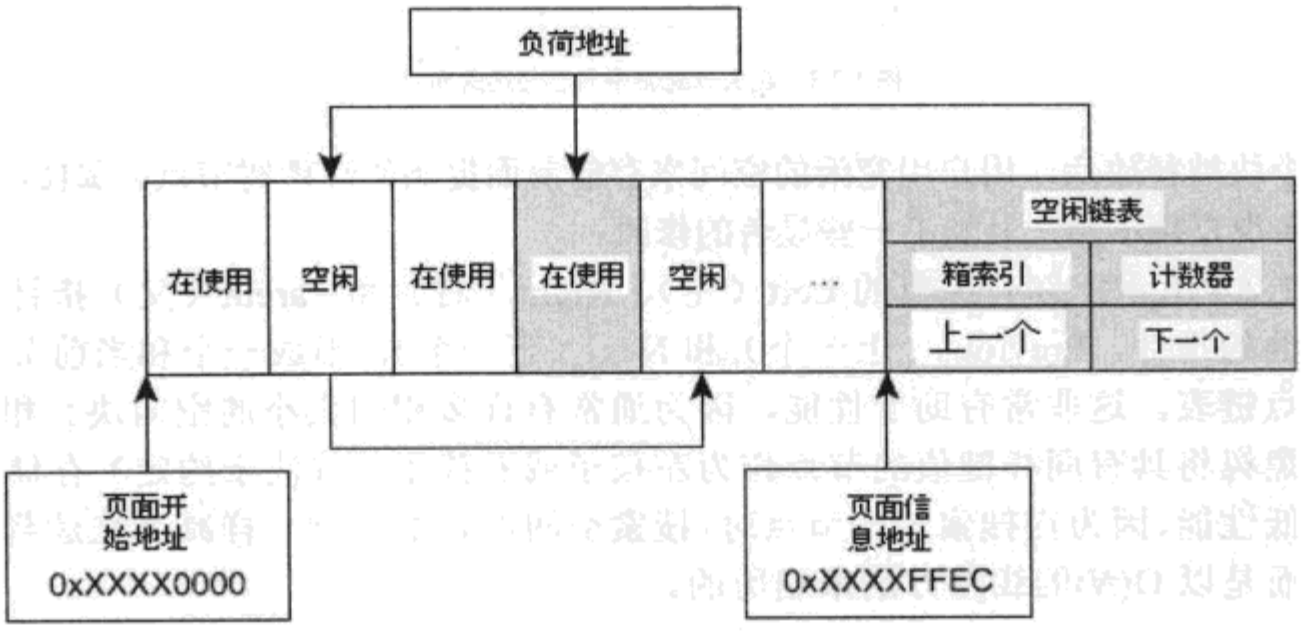


图 1.2.2　在一个箱中的单个页面的布局

因为页面大小不能刚好被元素大小整除，所以在它的后边常常有一小块剩余的内存，由于这个原因，此方法在页面的后边放置管理信息，而不是前面。因此，在这些情况下，管理是免费的。

这种方法还保证，不管什么时候，当一个页面被完全填满时，它是放在相应箱的页面链表的后面。这样，用户只需要检查页面的第一个空闲列表，来看看是否有可用的空闲元素。如果没有元素可用，该方法从 OS 请求额外的页面，初始化它的空闲列表和其他管理信息，并把它插入在箱的页面列表前面。

有了这个设置，小的分配和释放就变得相当简单。例如，当事先知道分配大小时，正如“new”物体的情形，编译器可以完全地内联，并消除任何箱索引的计算，在速度和大小上，最终代码变得和物体特定的池式分配器具有很高的可比性。

大分配器

小分配器是简单而快速的，但是，当分配大小增长时，箱化及池分配带来的好处迅速消失。为了解决这个问题，这个解决方案切换到一个不同的分配器，它使用一个头结构和一个嵌入式红黑树来管理空闲节点。红黑树有几个不错的属性，它们在这种情况下是有帮助的。首先，它自我平衡，从而提供了一个保证的 O(log(*N*))搜索，这里的 *N* 是空闲节点的数量。其次，它还提供了一个排序遍历，当处理对齐限制时，这是非常重要的。最后，有一个嵌入式红黑树的实现是非常方便的。

如图 1.2.3 所示，按它们各自的地址为序，头结构被组织成一个内存块的链表。因为下一个头结构的地址可以从当前头结构加上负荷内存块的大小隐性地计算出来，所以这里没有明确的“Next”指针。此外，用户需要存储关于一个块在当前是否空闲的信息，因为大分配请求的尺寸被向上圆整到头结构的大小（8 字节），所以这些信息可以存储在“尺寸”域的最低有效位。为了使头结构在负荷块间自然对齐，这样的圆整是必要的。

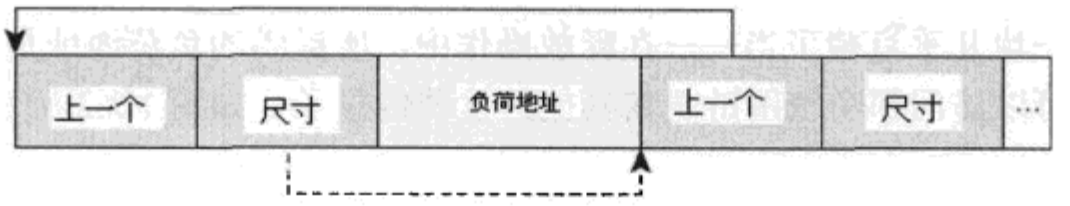


图 1.2.3 在大分配器中内存使用的布局

当一个块被释放后，用户用空闲的空间来存储前面提到的红黑树节点。如[Cormen90]，红黑树的实现直接了当，且加上一些显著的修改。

如图 1.2.4 所示，你有经典的 Left（左）、Right（右）和 Parent（父）指针，另外还有两个其他的指针，Previous（上一个）和 Next（下一个），形成一个和当前节点有相同键值的节点链表。这非常有助于性能，因为通常有许多相同大小的空闲块。相比之下，传统的红黑树将具有同样键值的节点作为左孩子或右孩子（取决于约定）存储。这将可预测地降低性能，因为在搜索这些节点时，搜索空间不是像往常一样减半来达到 O(log(*N*)) 的速度，而是以 O(*N*)的线性方法来遍历的。

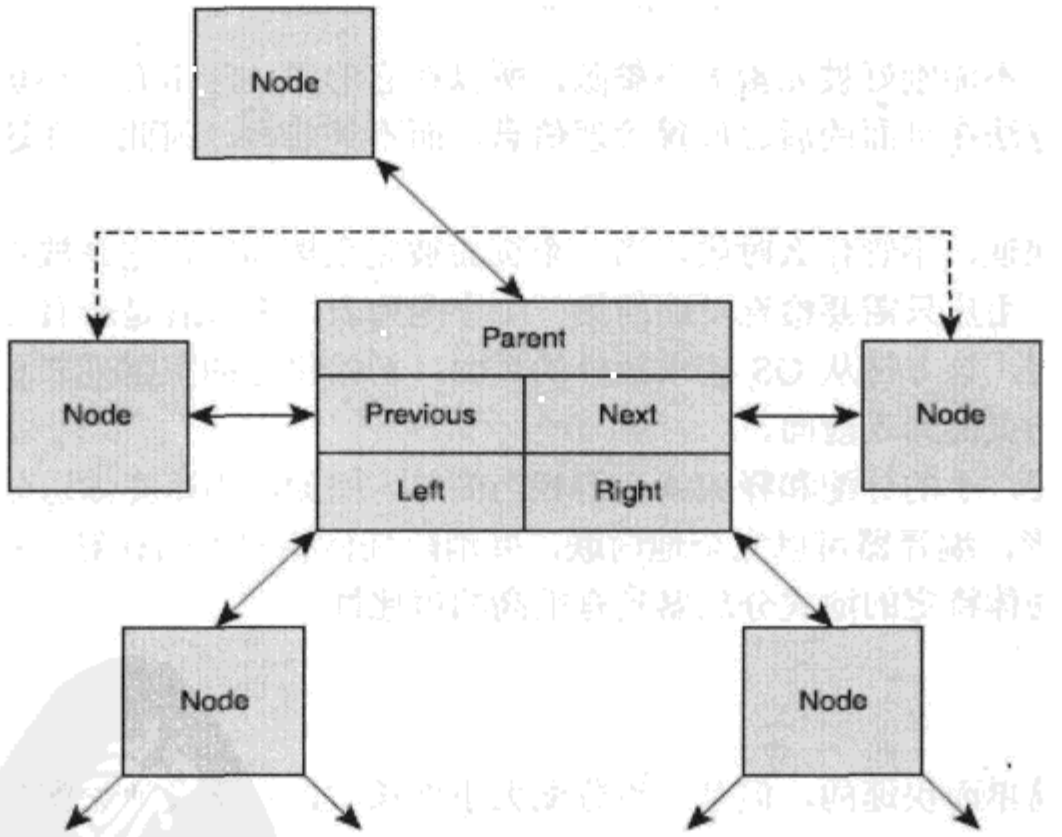


图 1.2.4 红黑树节点的布局

Left/Right（左/右）和 Previous/Next（上一个/下一个）指针都是用两条目的数组来组织，这样做主要是为了简化如“向左旋转”和“前任”的操作，它们通常有对应的“向右旋转”和“后继”。使用索引来表明左或右，然后就可以有一个能成为其中任一个的通用版本。

此外，在每一个节点中，你保留节点附着在父亲的哪一边的信息（左边还是右边）以及它的“颜色”——红色或黑色。这个分配器利用和头结构相似的压缩技巧，且把父亲边索引（side index）和节点颜色放在父指针的两个最低有效位。父亲边索引对性能很重要，特别是与使用所谓的“Nil”节点的红黑树结合起来时，因为重要的“旋转”操作就可以变成完全的无分支了。

如图 1.2.5 所示，“Nil”节点是一个特殊的节点，所有的终端节点都指向它，并且它也是树根附着的节点。根在“Nil”节点的左边这一事实也许看起来像是随机的，但是实际上这对遍历操作是非常重要的。用户很容易看到，在“Nil”节点上运行“前任”操作，将给出树上最大的元素，这刚好是从“Nil”节点开始后向遍历树时想要的。在 STL 术语中，这是容器的“End”。这样一来，“前任”操作不需要处理任何特殊的情形。

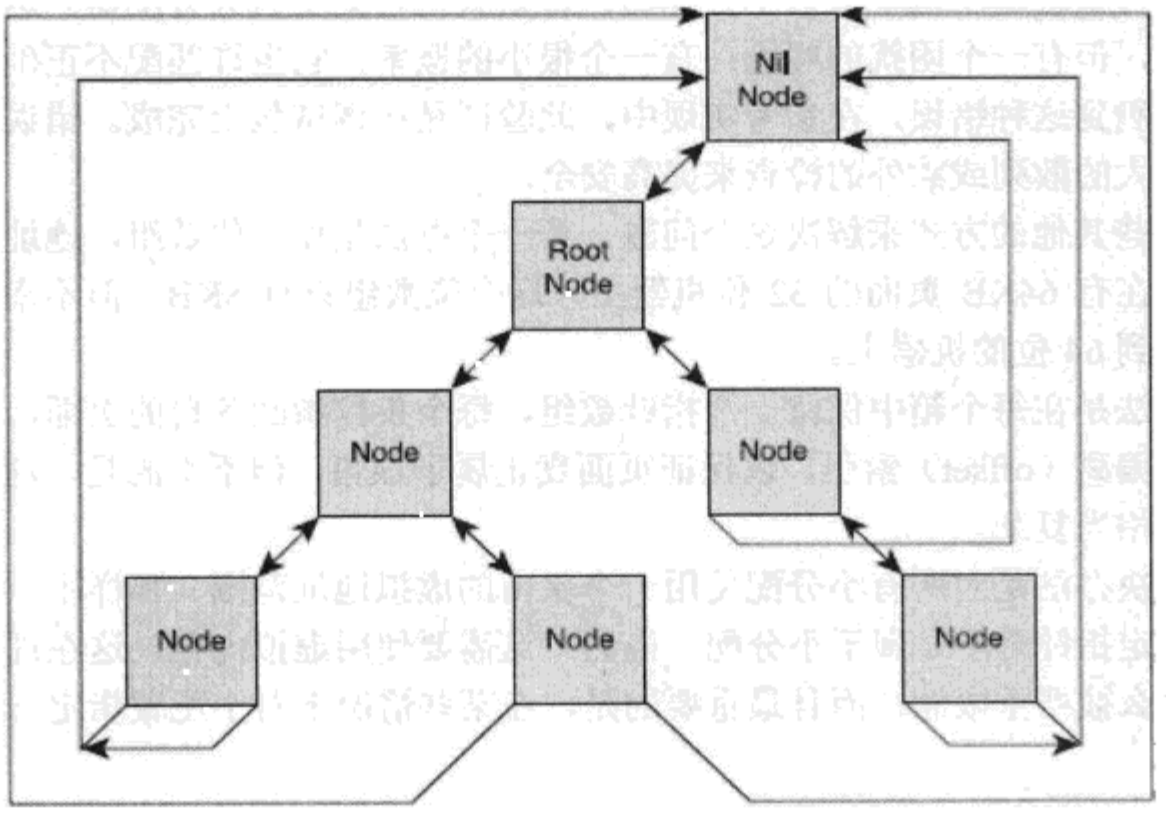


图 1.2.5　“Nil”节点在树上的位置

当所有这些设置完成后，在一个分配中查找红黑树来得到合适的大小。如果获得的块太大，它就被劈开，剩余的部分返回给树；如果没有可用的块，通常以大页面的倍数从 OS 请求更多的内存。

在一个释放操作中，用户搜索头结构，并确定是否有任何空闲的邻居可被凝聚起来。因为对每个被释放的块都做凝聚，在每一边（side）不可能有一个以上的相邻空闲块，所以这个操作只需要检查两个邻居，然后由此产生的空闲块被加入红黑树。

对红黑树更有趣的使用是当用户需要用非默认对齐来分配的时候。这个分配器利用这样的事实：你可以按已经排好的顺序遍历二叉树，并注意，需要用大于或等于所要求的尺寸，但要小于所要求的尺寸加上要求的对齐来检查节点。然后，你可以使用二叉树操作，用所要求的大小作为“下限”，以及用所要求的大小加上对齐作为“上限”，然后遍历此范围，直到找到一个满足对齐约束的块。遍历红黑树是一个 O(log(*N*))的操作，因此，很显然，较大的对齐将需要更长的时间才能找到。需要注意的重要事情是，这将保证最小的适应块

标准，被认为是减少碎片的一个主要因素，而且这是传统的过分配和转移方法所不能很好地满足的。

组合分配器

现在，你有两个分配器，这里有另外一个问题。小分配器依靠页面对齐来找到它的管理信息。由于有两个分配器，你需要一种方法来区分某一地址来自哪个分配器。有几种解决方案，其中没有一个是完美的，但有一个是最简单的，那就是先访问该页面的信息结构，并尝试去识别它。用户需要确保，大分配器始终使用至少和小分配器的页面一样大的页面，这样能够安全地访问页面信息。用户可以在每个箱放置一个用页面信息地址散列的标记，并在页面信息内存储它。然后，在一个释放操作中，用户访问该页面的信息，并再一次计算散列。如果匹配，你可认可那个地址来自小分配器；否则就把它提交给大分配器。这个解决方案非常简单和快速，但有一个困扰的问题：有一个很小的概率，它也许匹配不正确的页面。有一种方法可以检测到这种错误，在参考实现中，此验证是在调试版上完成。错误的检测一旦发生，可以用更大的散列或额外的检查来提高安全。

至少有一些其他的方式来解决这个问题。第一个办法是使用位数组，地址空间的每一个页面有一位。在有 64KB 页面的 32 位机器上，这个位数组只有 8KB，但不幸的是，这并不能很好地缩放到 64 位的机器上。

第二个办法是在每个箱中保留一个指针数组，每个条目指回各自的页面，而页面本身有一个箱和数组偏离（offset）索引，以保证页面真正属于该箱，但不幸的是，这使那个数组的内存管理变得相当复杂。

第三个解决办法是对所有小分配使用一个保留的虚拟地址范围，那样用一个简单的检查就可以立即确定指针是不是属于小分配。当然，这需要使用虚拟内存，这在许多游戏机中要么不存在，要么被严重限制，而且最重要的是，在某些情况下对小配置指定一些上限也许是不可能的。

多线程

当今，具有鲁棒性的多线程是很重要的，这往往是分配器需要适当地实现的第一件事。我们的解决方案不是划时代的，但在轻度竞争时提供了良好的效率。注意，小分配器的每个箱可以有一个互斥体（mutex），因为一旦选定了箱，箱之间就不存在数据共享，这意味着到不同箱的分配可以完全并行处理。另一方面，大的分配较复杂，一旦操作需要访问红黑树，就需要尽快锁定它。

每箱一个互斥体的仅有的快速替代是使用线程本地存储。那也许是可行的，特别是在可以负担一些额外的每线程内存松弛的系统中。不幸的是，线程本地存储仍然是不能移植的，而且有各种各样的怪癖。

调试功能

到目前为止，我们有了一个可轻易地替代 malloc/free 的高性能分配器。现在，用户可以方便地添加一些特征，以调试或提取额外的性能和功能。

有很多方法可以添加调试支持，参考实现用了一个典型的做法，即对所做的每个分配保留调试记录。这些记录都保存在分开管理的内存中，使它们尽可能少地干扰负荷。我们重复使用为主要分配器开发的一些方法，具体地说，我们使用一个嵌入式红黑树来快速搜索调试记录，并使用一种被称为书（book）的新颖容器。这个想法是，我们有一个和链表类似的混合数据结构，用“页面”来管理，并且只能在其末端添加元素，就像一个人在书中写字一样，因此得名。对这样一个特定结构的需要产生于这个事实：用户需要使用直接从 OS 来的大内存组块，而且想使用数据结构来管理那块内存，类似于动态数组的过分配，然后每次一个地填写元素。

当用户把“书”容器和嵌入式红黑色树相结合，就可以相当有效地管理调试记录。除了负荷块的大小和地址外，调试记录还存储了在分配请求时调用堆栈的部分副本，这对跟踪内存泄漏是非常有益的。此外，还可以存储所谓的“调试防卫（debug guard）”，这是一系列的内存字节，它们用负荷来过分配且用一系列的数字填充。

然后，在一个释放操作中检查“调试防卫”来核实块的完整性。如果找不到预期的数字序列，那么极有可能发生了一些内存踩踏。去确定谁做的踩踏有一点困难，但踩踏通常和分配块的代码相关，因此那是一个很好的起点。如果踩踏是可重现的，放置硬件断点能快速地发现踩踏冒犯者。

扩展

到目前为止，这个分配器相当密切地遵循 malloc/free/realloc 接口。我们发现，展示更多的功能可能非常有益于性能。

第一个扩展就是我们所称的“调整大小（resize）”——改变已分配块的大小，但不改变块地址的能力。显然，这意味着块只能从后面增长，尽管这看起来有限制，但是事实证明，它对实现动态数组或其他需要定期扩大的结构非常有用。

另外一个延伸是提供免费的操作，来接受原始请求尺寸及对齐的附加参数。这对分配器是有用的，因为它可以使用该尺寸来确定应该用小的还是大的分配器来释放那个块。这些额外的功能不能用于已分配或已调整大小的块，其仅有的有意义用途是实现高性能的“new”和“delete”的替代。

在本书附带光盘上有分配器的一个参考实现。在实践中，它和一个定制的 STL 实现一起，被使用在下一代游戏引擎中，其中，这个 STL 实现充分利用了分配器的特征和扩展。我们还提供了一个简单的综合指标，以验证该分配器有没有任何的性能优势。

此外，我们阐述了把该分配器与现有或未来的 C++代码集成的几种方式。最简单的方法当然是只覆盖全局的 new 和 delete 运算符。这样做有几个缺点，但最主要的是，它对中间件运行库不是一个妥当的解决方案。实践证明，听起来很诱人的每个类的 new 和 delete 运算符，使用起来是相当困难的，而且往往有令人沮丧的局限性。我们展示了一个去使用模板函数的可能方法，让每个对象有正确的自定义的 new 与 delete 功能。

1.2.4 参考文献

[Alexandrescu01] Alexandrescu, Andrei. *Modern C++ Design*, Addison-Wesley Longman, 2001.

[Berger01] Berger, Emery D. "Composing High-Performance Memory Allocators."

[Cormen90] Cormen, Thomas. *Introduction to Algorithms*, The MIT Press, 1990.

[Hoard00] Berger, Emery. "The Hoard Memory Allocator."

[Lea87] Lea, Doug. "A Memory Allocator."

1.3　用网络摄像头玩的视频游戏的光流

Arnau Ramisa, Institut d'Investigació en Intelligència Artificial
aramisa@iiia.csic.es

Enric Vergara, GeoVirtual
evergara@geovirtual.com

Enric Martí, Universitat Autónoma de Barcelona
Enric.Marti@uab.cat

在计算机视觉的商业游戏中，一项最广泛使用的技术就是光流的概念。光流是视频流连续帧的像素之间的运动，如由一个现代的摄像机提供。

在计算机视觉文献中，多种不同特性的技术被用来确定一对图像间的光流场[Beauchemin95]。本精粹将解释 3 个这样的技术——连续帧的直接相减、运动历史和一个更高级的 Lucas-Kanade 算法。它们有不同程度的鲁棒性和不同的计算成本，因此技术的选择取决于每个应用的要求。

1.3.1　简介

OpenCV 库准备了所提议的算法及许多其他计算机视觉技术，它们随时可以被使用。这个库是一个开源项目，它用 C 语言实现了用于计算机视觉应用的、最重要的算法和数据结构。经过 5 年的开发，在 2006 年，第一个稳定版本—— OpenCV（1.0）在市场上发行。

《游戏编程精粹 6》中有一篇文章使用了这个库来检测玩家的位置[Ramisa06]。在第三或第一人称射击游戏中，这一信息被用来映射围绕着角落的运动。

光流是一个向量数组，它描述了一个视频序列的连续帧之间像素发生的运动。通常，在现实生活中，它被用来近似对象的真实运动。光流信息不仅被许多计算机视觉应用所使用，而且还被利用在如飞行昆虫，甚至人类的生物系统中。这样的一个例子是 MPEG-4 视频压缩标准，它使用像素块的光流来消除视频文件中的多余信息。

在计算机视觉游戏中，光流被用来判断屏幕的某一像素有没有“激活”。如果某一区域有足够的像素被激活，我们可以认为一个“按钮”已被按下，或者一般来说，已经完成了一个动作。

为此，通常没有必要去估计全部的光流场，而使用计算上更便宜的方法，即只有当某一像素相对于前一帧改变后才进行计算的方法。

1.3.2 OpenCV 代码

本小节将介绍 webcamInput 类中最重要的函数。此类的使用围绕着从摄像头获得新帧的主要循环而构建。

```
#define ESC_KEY 27
webcamInput webcam(true,1);
// 方法 = (1: Lukas-Kanade, 2: 差异, 3:运动历史)
while(1)
{
    webcam.queryFlow();

    if(cvWaitKey(10)==ESC_KEY)
        break;
}
```

在此代码中，一个叫 webcam 的新 webcamInput 对象被创建，这个对象封装了通过摄像机获得和处理图像的所有逻辑。构造函数需要两个参数：一个是在对象构造中，相机连接是否应该初始化的布尔值；一个是从 1～3 的整数，表明将使用光流的哪个方法。再后来，在循环体内，queryFlow 函数被用来获取和处理一个新的帧。最后，函数 cvWaitkey 被用来做 10ms 的暂停，等待可能停止这一进程的 Esc 键。

取决于所用方法，在初始化时，所有需要的变量在内存中分配：

```
void webcamInput::queryFlow()
{
    getImage();

    switch(method)
    {
    case 1:
        lucaskanade();
    break;
    case 2:
        differences();
    break;
    case 3:
        flowHistory();
    break;
    }

    flowRegions();

    cvCopy( image, imageOld );
    // 保存前一个图像来做光流计算
}
```

函数 getImage 从摄像头获取一个新图像，并将其转换为灰度，然后根据选择的方法调用适当的函数。最后，在所定义的兴趣区域，flowRegions 被用来计数激活像素，当前的图像被复制到 imageOld，以在之后的光流计算中使用它。

1.3.3　第一种方法：图像差异

第一种方法是最简单的——图像差别。这种方法非常简单，包括从前一帧中减去当前的相机图像。

```
void webcamInput::differences()
{
    cvSmooth( image, image, CV_GAUSSIAN, 5, 5);
    cvAbsDiff( imageOld, image, mask );
    cvThreshold( mask, mask, 5, 255, CV_THRESH_BINARY ); // 使用阈值
    /*可选*/ cvMorphologyEx(mask, mask, NULL,kernel, CV_MOP_CLOSE ,1);
    cvShowImage("differences", mask);
}
```

cvAbsDiff 函数从 imageOld 减去 image，两者分别是网络摄像头的前一个和当前的图像。操作的结果存储在 mask 中，在下一行，其内容被二值化，所有低于值 5 的像素（这表明在 image 和 imageOld 中的类似像素）被丢弃，而那些具有较高值的被标记为“活跃”。这个阈值取决于所用网络摄像头的敏感度，如使用一个较低的值，更多的活跃像素将被正确地检测到，但与此同时，更多由图像噪声所产生的假活跃像素也会出现。为了在估算中降低噪声的影响，第一步包括用 5×5 像素的 Gaussian 内核来平滑图像。这是用 cvSmooth 函数来完成的。

个别或小组的激活像素不太可能对应于实际的移动对象，因此，函数 cvMorphologyEx 使用了一个封闭（closure）来消除这些虚假的激活像素，而不改变正确的激活像素[Morphology]。形态运算符的大小和形状用结构体 kernel 定义。最后，末尾的一个参数指定了侵蚀（erosion）和扩张（dilation）连续进行的次数。封闭是一个功能强大但计算昂贵的操作。如果图像的噪声水平低，或者通过提高阈值，它是可以避免的。

这种方法比 Lucas－Kanade 方法快，但在处理不良的相机质量上并不强大。

1.3.4　第二种方法：运动历史

第二种方法叫运动历史，它使用与图像差异相同的原则，但不仅仅是使用前一帧的图像，而且“记住”最近的活跃像素。

```
void webcamInput::flowHistory()
{
    // 流历史

    cvSmooth( image, image, CV_GAUSSIAN, 5, 5);
    cvCopy( image, buf[last]);
    IplImage* silh;
```

```
    int idx2;
    int idx1=last;
    idx2 = (last + 1) % N; // 第(last - (N-1)) 帧的索引
    last=idx2;
    int diff_threshold=30;

    silh = buf[idx2];
    double timestamp = (double)clock()/CLOCKS_PER_SEC;

    cvAbsDiff( buf[idx1], buf[idx2], silh );

    cvThreshold( silh, silh, diff_threshold, 1, CV_THRESH_BINARY );
    cvUpdateMotionHistory( silh, mhi, timestamp, MHI_DURATION );
    cvCvtScale( mhi, mask, 255./MHI_DURATION,
              (MHI_DURATION - timestamp)*255./MHI_DURATION );

    cvShowImage("M_history",mask);
}
```

在此函数中，buff 是存储网络摄像头的最后 N 个图像的一个循环缓冲区。当一个新的图像进入缓冲区时，最旧的图像与新图像相减，作用阈值于结果来找到发生了可靠变化的像素。然后，利用这一新的信息和最新帧的时间戳来更新运动历史图像 mhi。最后，新的运动历史图像被缩放成一个每像素 8 位的遮罩图像。此外，这个运动历史图像可以用来确定运动的梯度和分隔不同的移动对象。在 OpenCV 例子的 motempl.c 文件中可以找到此方法的一个完整例子。

1.3.5 第三种方法：Lucas-Kanade 算法

前两种方法并不给出实际光流的输出，它们只是检测发生运动的像素。然而，在这种情况下，用户可以用较少的计算量来换取移动像素的目的位置。所给出的最后一种方法是 Lucas 和 Kanade 提出的光流估计方法[Lucas81]。通过使用类似于 Newton-Raphson 寻找函数零值的方法，此方法估计给定移动像素的位置。

```
void webcamInput::lucaskanade()
{
    cvCalcOpticalFlowLK(imageOld, image, cvSize(SIZE_OF,SIZE_OF), flowX, flowY);

    cvPow( flowX, flowXX, 2 );
    cvPow( flowY, flowYY, 2 );
    cvAdd( flowXX, flowYY, flowMOD );
    cvPow( flowMOD, flowMOD, 0.5 );

    cvThreshold( flowMOD, flowAUX, 10, 255, CV_THRESH_BINARY );
    /*可选*/ cvMorphologyEx(flowAUX, flowAUX, NULL,kernel, CV_MOP_CLOSE ,1);

    cvShowImage("LUKAS_KANADE",flowAUX);
}
```

利用 Lucas-Kanade 方法，函数 cvCalcOpticalFlowLK 计算灰度图像 imageOld 和 image 间的光流。函数的其余参数是 cvSize(SIZE_OF, SIZE_OF)指定了窗口的大小，此窗口用于定位其他图像中的对应像素，flowX 和 flowY 将存储光流向量的元素。用户感兴趣的是表明运动强度的向量模块。一旦有了该模块，就对它作用阈值，以消除所有由噪声引起的活跃像素。如果相机确实有太多噪声，也有可能使用形态封闭来删除孤立的像素。

表 1.3.1 中的时间值是在一个运行 OpenSUSE 10.2 操作系统的 Pentium IV 3GHz 计算机上得到的，其中执行时间在超过 100 次的迭代中测量，图片大小为 640 × 480。

表 1.3.1　　运动检测算法每次迭代所需的时间

	均　值	中　值	标　准
图像差异	0.020 867 s	0.025 000 s	0.006 3869 s
运动历史	0.027 794 s	0.025 000 s	0.005 6899 s
Lucas-Kanade 算法	0.211 14 s	0.195 00 s	0.04 3091 s

1.3.6　光流游戏

为了看到光流在实际应用中的效果，我们开发了一个简单的类似于 Eye Toy: Play 的游戏（一个由索尼计算机娱乐公司伦敦分部开发的 PlayStation 2 游戏，使用了类似于网络摄像头的数码相机）。游戏的想法很简单：通过身体运动来清除屏幕上显示的污渍（见图 1.3.1）。

因为玩家可以疯狂地移动他们的手臂，所以一旦当污渍出现时，玩家可以没有任何困难地将它们清除，我们增加了一些有毒污渍，玩家必须避开它们才能继续玩下去，这迫使玩家必须小心他们的动作，引进了让游戏更有趣的技能因素。

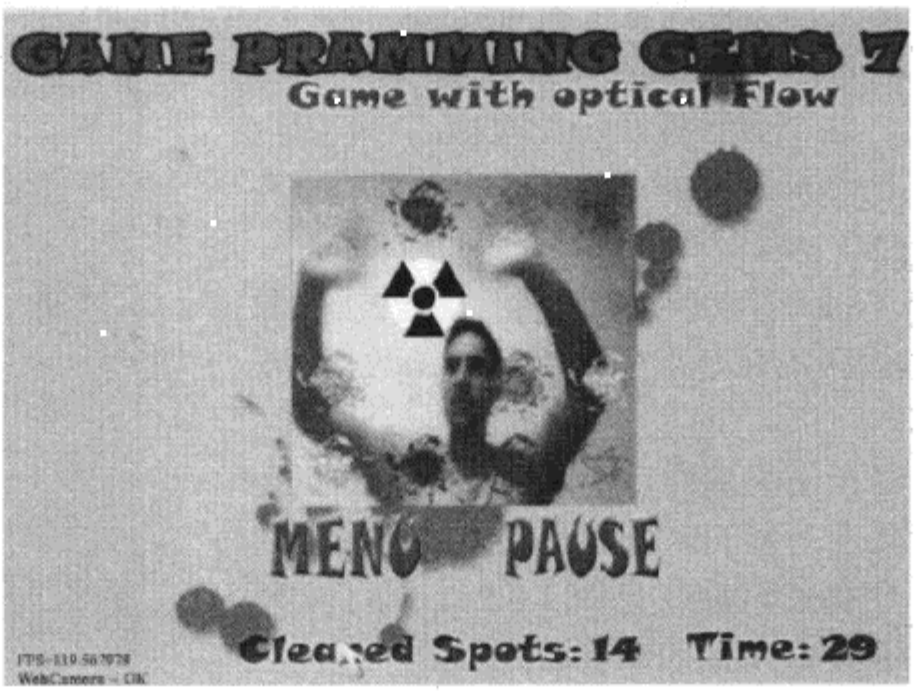


图 1.3.1　游戏截图

问题的关键是在游戏中利用光流，所以我们已经在称为 webcamInput 的 C++类中封装了前面介绍的功能，这使得它更容易地适合于游戏。类的公共接口如下：

```
class webcamInput
{
```

```
public:
    webcamInput( int method=1 );
    // Method= (1: Lukas-Kanade, 2: 图像差异, 3:运动历史)
    ~webcamInput( );

    void   GetSizeImage ( int &w, int &h );
    uchar* GetImageForRender ( void );

    void QueryFlow ( void );
    void AddRegion ( int x, int y, int w, int h );
    std::vector<float> FlowRegions ( void );

private:
    ...
}
```

在类的构造函数中，你可以决定要使用前面介绍的 3 种方法中的哪一种来计算光流。在默认情况下，我们使用似乎产生最佳结果的 Lucas-Kanade 方法。

为了在屏幕上显示来自网络摄像头的彩色图像（玩家的身体），这个类提供了两个重要的函数。第一个函数是 GetSizeImage，告诉用户来自网络摄像头的图像大小。第二个函数是 GetImageForRender，它返回一个 unsigned char 的数组，这些值代表了以行排序的像素的 RGB 元素。通过使用 QueryFlow 函数可以计算当前帧的光流。

然而，对于游戏，用户需要的是能够在图像的某一特定区域查询已经发生了运动的数量（将被擦除的由污渍占据的区域）。要做到这一点，在游戏开始时可以使用 AddRegion(int x, int y, int w, int h)来定义想要的任意多个区域。此函数创建一个区域，它的原点在网络摄像头生成图像的（x，y）坐标上，另外，宽度和高度由（w，h）给出。

一旦确定该区域，每次调用 QueryFlow 时，都可以通过使用函数 FlowRegions 来得到在每个区域的运动范围。这个函数返回一个在[0，1]范围内的浮点值列表（如 std::vector<float>），它们代表了被检测出的运动中的像素比例。

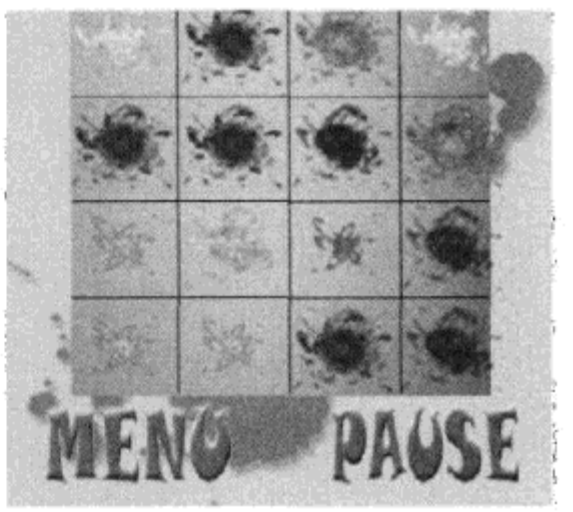


图 1.3.2 分割屏幕成几个区域的游戏截图

在游戏中，用户必须初始化一个此类的对象（mWebCam），随后从代码的几个部分调用它的函数，如下所述：

（1）正如我们已经指出的那样，游戏初始化后，需要将光流的查询分割成几个不同的区域，特别是要把最初的图像分成 16 个 4×4 网格的区域（见图 1.3.2）。为了实现这一目标，需要使用函数 mWebCam.AddRegion：

```
int width, height;
mWebCam.GetSizeImage( width, height );
for(int col = 0; col < 4; col++)
{
   for(int row = 0; row < 4; row++)
   {
      mWebCam.AddRegion(col* (width/4), row*(height/4), width/4, height/4);
```

```
        }
    }
```

（2）在渲染时可以调用函数 mWebCam.GetImageForRender，以绘出来自网络摄像头的彩色图像。在图像的上面，根据它有多清洁用一个特定的 Alpha 分量覆盖污渍。有污渍的图像将占据 16 个区域的每一个所决定的空间。

（3）在游戏更新时，函数 mWebCam.QueryFlow 将先被调用，其次是 mWebCam.FlowRegions 的调用。这样一来，用户知道在 16 个区域的每一个所发生的运动的程度，使越来越多在那个特别的时刻活跃的污渍图像更透明（基于检测出的运动）。当污渍变得完全透明时，它们就消失了。

为了保持一个可以接受的帧速率，需要考虑两点。第一个是游戏的 Update 函数不应该每一次都调用 QueryFlow，因为后者在计算上是非常昂贵的。绕过这个问题，用户可以限制 QueryFlow 的调用在一定的频率，如低于 Update 调用频率的每秒 x 次。这将使游戏在大多数现代化的计算机上平滑运行，而且不影响游戏的响应程度。

另一个考虑影响到来自网络摄像头图像的分辨率。也就是说，分辨率越高，在获得光流之前，CPU 需要进行的计算就越多，这意味着必须设定参数，例如，320 像素×240 像素。

作为最后的评论，本书附带光盘上不仅有在 Microsoft Windows 上的可执行工程版本，而且有它的源代码，以及一个有 UML 类图的 PDF 文件。游戏编程用了 C++语言、Microsoft Visual Studio 2005、DirectX 和 OpenCV 库。

1.3.7　参考文献

[Beauchemin95] Beauchemin, S.S and Barron, J.L. “The Computation of Optical Flow,” ACM Computing Surveys (CSUR), Vol. 27, No. 3, pp. 433–466, ACM Press New York, NY, USA, 1995.

[Lucas81] Lucas, B.D. and Kanade, T. “An Iterative Image Registration Technique with an Application to Stereo Vision.”

[Morphology] “Morphological Image Processing.”

[Ramisa06] Ramisa, A., Vergara, E., and Marti, E. “Computer Vision in Games Using the OpenCV Library,” *Game Programming Gems 6*, edited by M. Dickheiser, Charles River Media, 2006, pp. 25–37.

1.4 一个多平台线程引擎的设计与实现

Michael Ramsey
miker@masterempire.com

引擎的开发正在变化。正如一种古老杰作的色彩会随着时间的推移逐渐褪色，过去时代的单核心引擎开发尝试过的一些真正技术也会一样。现在，开发商必须接受执行他们游戏的多核心系统架构。在某些情况下，基于每个核心的效率都不一样。在一个多核心环境下的架构开发，必须被确认、设计、计划并最终实现。通过提供一个多平台线程系统的一个理论基础和一个实用框架，本精粹协助于实现的方面。

游戏引擎架构的一个基本概念是：它必须能够利用多核环境。对此环境的利用就是一个在定义上允许多个任务并行执行的系统。此外，用户希望正在讨论的系统的性能是实时的。这一实时性能需求要求线程系统也将是轻巧的。在使用该线程系统的对象的开发中，数据结构、对象的构造、缓存的考虑和数据访问模式仅仅是用户必须不断认识到的少数几个问题。

本精粹的重点是一个线程引擎的开发，该引擎已被设计为一个 Xbox 360 和一个标准的多核 PC 游戏引擎。因此，我已经提供了核心系统的详细资料，这一系统适用于所有的架构，而不仅仅是一个多核心台式机（Windows/Linux）或 Xbox 360。当你读到缓存线时，它们将集中于那些使它们在多平台和操作系统上重要的原则。

1.4.1 一个实用线程架构的系统设计

设计一个多线程程序时，其中一个最重要的因素就是提前支出时间来设计和规划游戏架构。一些需要解决的高层次议题包括以下内容：

- 任务相关性；
- 数据共享；
- 数据同步；
- 数据访问模式的确认和流动；
- 解除联系点的耦合，以便能够读取数据，但不一定是写；
- 最小化的事件同步。

线程一个游戏系统时，其中一个最基本的、然而是最有效的原则是：确定有较少相关性的大型系统（或甚至于系统间的通信集中点）并线程它们。如果通信系统间的联系点足够集中，几个简单的同步原语（如自旋锁（spinlock）或互斥体（mutex））通常就够了。如果有一个停顿（stall）被

检测出，这就直截了当地去确定，并通过路由到不同的物体间管理器来降低特定事件的粒度。在设计一个多线程游戏引擎时，不但要保持稳定，而且为扩展性而努力也是重要的。

一个实用的线程架构

在本书的 CD 上，你会发现一个完整的多平台线程系统源代码。GLRThreading 库已经用一个与平台无关的方式设计，并提供了一个在 Windows 操作系统上的实现。接口有利于扩展到 Xbox 360。GLRThreading 库支持 Win32 API 的所有普通线程功能。为便于使用，有些功能已被封装（例如，GLRThreadExecutionProperties）。标准的 Win32 线程模型，在本质上是抢占性的。在本文中，抢占性就是指任何线程可以被操作系统暂停，以让另外一个线程来执行。这使得当只有一个单独的处理器时，OS 可以来模拟多个进程。抢占可直接被一个属性化的 GLRThreadTask 属性影响，但一般来说，用户应该知道，一旦一个任务已经在 GLRThreading 库执行或恢复（即提供给 Windows OS），它可以而且很可能会被抢占，或其执行时间被操作系统减少（增加）了。

GLRThreading 库的基本组成部分

当你学习 GLRThreading 库的结构时，可用图 1.4.1 来更好地了解该系统的组成部分和依赖关系。GLRThreading 系统的基本接口被恰当地命名为 GLRThreadFoundation。GLRThreadFoundation 是一个单例模式，它应当被提供给所有需要获得线程能力的游戏系统。通常情况下，GLRThreadFoundation 被放置在一个预编译头文件中，随后，它被包括在引擎的所有文件中。用户可用 GLRThreadFoundation 来控制任务的提交。但是，在看此之前，用户必须确定和定义一些执行环境的基本属性，这时就涉及系统的描述。

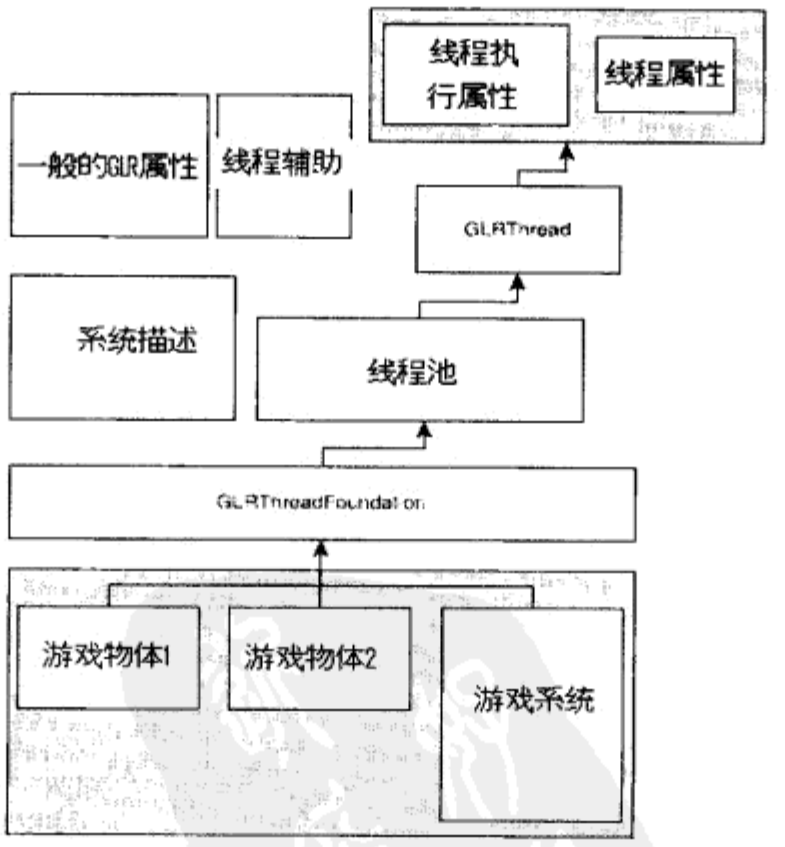


图 1.4.1　GLR 线程库

为了正确地执行，线程系统需要有能够查询相关环境信息的能力，包括确定处理器的数量、内存负荷以及是否支持超线程技术（hyper-threading）。为了以一个平台无关的方式来适应这些，这就有了 GLRISystemDescription。平台相关的具体实施都源自这一基本接口。对于 MS Windows，该系统描述为 GLRWindowSysDesc，Xbox 360 是用 GLRXBox360SysDesc 实现的。

GLRThreadFoundation 的用法

GLRThreadFoundation 是所有线程交互的焦点。用户可以执行的线程交互类型包括从一个游戏对象执行任务的能力，以及从线程池访问线程的能力。在基本代码之内，会有一个线程基础的单个实例。例如：

```
GLRThreadFoundation glrThreadingSystem;
```

若要获得线程系统内的功能，可使用下列方法的语法：

```
glrThreadingSystem.FunctionName();
```

其中，FunctionName 是任何与平台无关的、可以被游戏级部件执行的功能。

1.4.2 线程

线程是可以被操作系统或内部调度系统调度而执行的代码段。图 1.4.2 是一个典型的单线程环境与其对应多线程的比较。这些代码片段可以是单一函数、对象，或者可以是整个系统。GLRThreading 库的接口和所有在 GLRThread 层次上的操作都被设计成与平台无关。

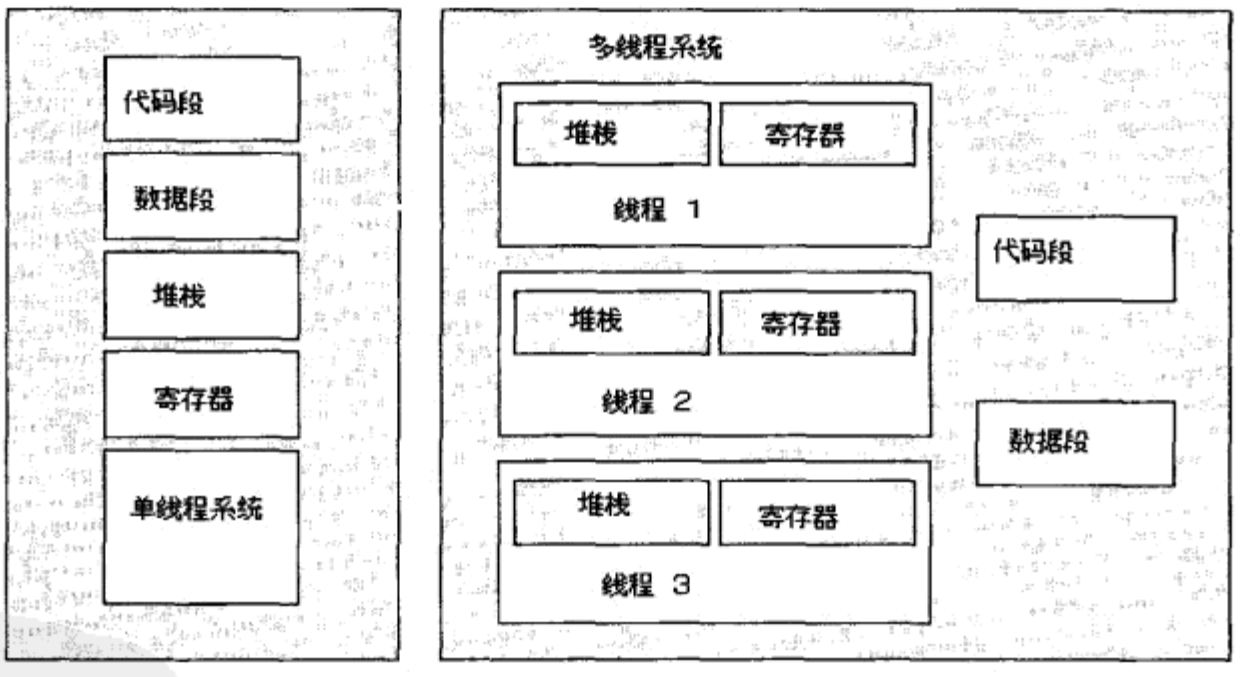


图 1.4.2 单线程和多线程编程模型的比较

在 GLRThreading 库内，GLRThread 是一个线程的与平台无关的实现。作为在一个平台上执行的线程，GLRThread 接口允许以下操作：

- 创建一个线程；

- 执行一个线程；
- 改变一个线程的属性；
- 恢复一个线程；
- 终止一个线程；
- 暂时挂起一个线程；
- 查询一个线程的状态。

属性管理和线程的执行有几个变化。GLRThread 属性和 GLRThreadTask 这两种控制机制也与 GLRThread 相关。这些机制控制了（和其他方面）一个线程在哪里和在一般情况下如何执行一个任务。

抢占的和同时执行的线程

至关重要的是，每一个引擎开发人员都必须知道，不但是他们的引擎所针对的，而且是用来开发的内核的性能特征。这是因为大量的开发首先是在台式 PC 上实现的，它们通常有着和典型的多核心主机不同的执行特性。设计一个线程系统时，其中一个最重要的方面就是考虑可抢占的线程和那些被不可知地称为非抢占的线程。一个线程遵循的一个标准模式是：OS 可以挂起一个线程的执行来让另一个线程执行。当操作系统决定挂起当前线程的执行时，它将存储当前正在执行的线程的上下文，并恢复下一个线程的上下文状态。上下文切换不是免费的，它通常有一些开销，但一般来说，空闲一个线程而不引起上下文切换的费用不值得所增加代码的复杂度。线程的切换造就了一个多任务系统的幻想。

此外，在游戏机上还有对创建基本属于非抢占线程的能力的支持。非抢占线程是一个不能被 OS 中断的线程。这种能力不代表幸福的顶峰。独立执行的线程（在相同的核心）通常共享相同的 L1 缓存，这一般意味着，为了利用数据结构所固有的任何的缓存一致性，你仍要在同一内核上执行类似的任务，这是为了减少当两个不同的系统在相同的内核中执行任务时可能发生的缓存抖动。

线程属性

GLRThreadProperties 是一个存储特定线程句柄和它的相关 ID 的一般机制，此外，它还有改变线程默认堆栈大小的能力。线程的堆栈就是其变量和调用堆栈的位置。OS 可以自动地增长堆栈，但在游戏机上，这是一个性能打击。为了避免动态改变线程堆栈，用户应该事先预见并设置堆栈大小。

线程上下文切换一般非常快，但和其他事情一样，也有相关的费用。其中的一项费用是和线程相关的，来存储它的上下文信息的内存。GLRThread 的默认大小是 64KB。取决于目标平台，这个 64KB 可以而且应该手动调整。如果需要在基于 Windows 的 OS（如 PC 或 Xbox 360）上增加线程堆栈的大小，有一个可以从 Visual Studio 开发环境中设置的属性。

一个需要注意的陷阱是当用户需要大小为 128KB 或 256KB 的堆栈时。通常，这类情况需要减小正在执行的任务规模（即减小粒度），或者是确定还可以进一步分解成更小实现的对象，如[Ramsey05]。

线程执行属性

一个线程的执行属性是一个与平台无关的接口，它允许对任务的执行有更精细的控制。GLRThreadExecutionProps 提供了一些属性，如定义一个任务的首选处理单元、优先级以及它的亲和掩码。线程的执行属性也有能力来定义其理想的处理器元素，这容许一个在特定的处理器上执行类似于群组的任务管理系统。它在文件 GLRThreadExecutionProps.h 内定义。

在有超线程能力的单一物理处理器上，函数 GetProcessAffinityMask 将返回第 1 、2 和 3 个位，来表明用户至少有一个物理 CPU 或者两个逻辑 CPU。在一个有超线程能力的双物理 CPU 机器上，GetProcessAffinityMask 将显示 1+2+4+8=15，这表明有两个物理 CPU，其中它们各有两个逻辑 CPU。CPU 的标识从 1 开始。应当指出的是，GLRThreadExecutionProps 的实现在头文件 GLRThreadExecutionProps.h 中。

处理器亲和

亲和要求线程在指定的处理器上执行，这允许系统开发人员针对特定的处理器做反复的操作。利用反复操作的方式，可以把相关的操作聚合在一起，以进一步提高类似对象的缓存一致性。有些系统可能把线程亲和当成一个提示，而不是要求，因此，在对线程库的任何问题做假定之前，须检查文档。

指定任务的亲和力是通过一个简单的标识 PA_HARD 或 PA_SOFT 来达到的。这些标识分别告诉 OS 去使用一个特定的处理器，或只是用指定的处理单元作为一个暗示。它在 GLRThreadExecutionProps.h 定义。

任务优先级

建议只对某些系统改变线程优先级。GLRThreading 库允许用户按照以下级别来指定线程的优先级：

- TP_Low；
- TP_Normal；
- TP_High；
- TP_Critical；
- TP_TimeCritical。

新创建线程的执行属性优先级的默认设置是 normal，这可取决于某一特定任务的要求而被改变。这在 GLRThreadExecutionProps.h 中定义。

1.4.3 线程分配策略

建立一个多线程系统有许多方法，从天真的动态分配线程到比较复杂的预分配工作系统的实现。对 GLRThreading 系统，一个基本的模式是用户想事前做尽可能多的分配。因为用户不仅要知道在启动时内存的消耗，而且要知道运行时系统级组件的内存使用，所以这在开发一个控制台游戏时是很重要的。

天真的分配

创建一个线程系统的最简单和最直接的方法是拥有一个线程管理器对象，该管理器对象用“创建为了使用”的模式来处理请求，它用一个单例模式来实现。虽然这可能是进行启动的最简单办法，但是，当考虑到产品周期的整个长度和不断分配及释放一个线程的运行效率时，这不是一个很好的决定。

线程池

考虑到前载系统级分配的基本原则，GLRThreadingPool 的内部实现依赖于尽可能多的预分配。线程池是一个前载线程创建的系统，这就避免了在运行时分配资源，而在启动时一下子就处理掉。虽然创建一个单一的线程不昂贵，但在运行时频繁地分配和释放是一个不必要的负担。例如，如果线程不断地被分配和释放，系统就会产生内存碎片。为线程池创建的实际线程数量取决于系统，它可以根据游戏的需要和效率标准进行修改。

线程池是一个子系统，它是基于经过时间考验的任务提交模式。每隔一个正常的间隔，工作线程寻找一个任务来执行。如果有可做的工作，线程继续并设置执行属性。一旦任务完成，线程就暂停，并可被随后的任务所使用。

线程池属性

线程池的有些属性允许定义几个，当系统应用在不同的游戏中时，将被证明是有益的特征。线程池需要能改变已创建线程的数量、线程可以工作的任务数和在任何特定的时间锁定任务池的能力。

多个池

于是，你可能会问自己：假如有一个池是好的，那么在游戏引擎中对不同的子系统创建多个线程池可能甚至是一个更好的主意。引进多个池的问题是多方面的——首要问题是，如果有持不同性能特性的多个池（通过使用线程特性、任务调度等），就必须引进另一层复杂性（池间沟通的需要）进入系统。在这样一个性能很关键的系统中，这种复杂性是不想要的。底层的线程系统越直接，用户就越有可能避免由系统的复杂性引入的困难。

1.4.4　对象的线程

通过建立一个对象，然后提交该新建对象到线程库的过程，GLRThreading 库提供其线程能力。为了提交一个对象给 GLRThreading 系统，只需做如下的调用：

```
GLRThreadFoundation.submitTask(&newGameSystemFunction);
```

为了让工作线程确实从线程池内的任务列表得到任务，需要如此调用 distribute：

```
GLRThreadFoundation.distribute();
```

现在，你的对象就可以在和调用进程相同的处理器上运行了。为了让对象的执行容易些，允许线程执行的过程应被插入在游戏的主要更新中。用这种方式，游戏系统可以只创建并提

交任务，而让线程系统处理分配、执行和所有细节。

1.4.5 线程的安全性、重新进入、对象同步和数据访问

在线程安全的游戏系统的设计问题上，重新进入很棘手，它超出了本精粹的范围。很多问题高度依赖于游戏的结构和数据访问模式。在创建可线程的对象和系统时，需要记住一些原则和做法。

重新进入的一条经验法则是：只有需要时才让一个系统可重新进入。让底层游戏库可以100%重入需要花费大量的时间和努力，但是，事实的真相是大多数系统不需要重新进入。当然，一些库（如内存管理器和任务提交系统）需要保证重入和线程安全，但是，通过使用廉价的同步构造，很多管理系统可用来把关。使用与一些基于硬件的自旋锁一样简单的东西，可以把线程安全的负担推给系统管理器。这是很有道理的，因为它们应该控制自己的数据流。所以，一旦确定了引擎内的一般数据流，决定究竟哪些需要重新进入通常就很清楚了。

1.4.6 使用缓存线（或缓存的一致性）

对一个标准的单核心引擎，沿着缓存边界的对象对齐是很重要的，但对多核心环境，它更为重要。一般来说，一个缓存被分成 32 或 64 字节的缓存线。当主存直接映射到缓存时，一般的策略是无需关心正在被映射的内存量，但要关心正在访问的缓存线数量。有以下 3 种基本类型的缓存未命中。

- 强制性未命中。这在一块内存首次读入缓存时发生。
- 容量未命中。当内存块大得缓存放不下时会发生。
- 冲突性未命中。当内存块映射到相同的缓存线时会发生。在一个多核环境，应该警惕冲突性未命中。冲突性未命中通常是系统性的，这是因为引擎架构包含设计不当的数据结构。这些数据结构和一般线程执行的不确定性模式导致了冲突性未命中，从而消极地影响性能。

GLRThreading 库包括了一个将帮助用户创建缓存对齐的数据结构的基本工具。主要的工具是 GLRCachePad 宏。

```
#define GLRCachePad(Name,BytesSoFar) \
   GLRByte Name[CACHE_ALIGNMENT - (BytesSoFar) % CACHE_ALIGNMENT]]
```

在缓存线的边界上，GLRCachePad 宏将数据按缓存线（尺寸）组块分组。你希望不同 CPU 的访问模式至少被一个缓存线边界分开。对不同的平台，缓存对齐值可能会不同，因此取决于你的目标系统，你可能需要执行一个不同的缓存填充方案。最后要提醒的是，你希望 GLRCachePad 的调用发生在一个数据结构的末端，这将迫使随后的数据结构到一个新的缓存线[Culler99]。

1.4.7 如何使用 GLRThreading 库

本小节将通过一个简单的例子来说明如何使用线程系统。一个典型的游戏对象将被定义

且被称为 TestSystem（见程序清单 1.4.1）。

程序清单 1.4.1　测试游戏对象

```
class TestSystem
{
  public:
     TestSystem();
     ~TestSystem();
     void theIncredibleGameObject( void );
     void objectThreadPump( void );

  private:
     GLRThreadedTask<TestSystem> mThreadedTask;
     GLRThreadExecutionProps *mThreadedProps;
};
```

TestSystem 游戏对象有两个私有的数据结构：一个 GLRThreadedTask 和一个 GLRThread-ExecutionProps。GLRThreadedTask 成员用来引用这一特定对象和对象内的一个函数，该函数将要被线程系统执行。清单 1.4.2 包含了一个如何注册实例对象的例子和将要被分发执行的那个函数。

程序清单 1.4.2　游戏对象可线程函数的实现

```
void TestSystem::objectThreadPump( void )
{
   mThreadedTask.createThreadedTask( this,
                  &TestSystem::theIncredibleGameObject, mThreadedProps );
   glrThreadingSystem.submitTask( &mThreadedTask );
}
```

清单 1.4.3 中的示例代码表明了如何使用清单 1.4.1 中的 TestSystem 对象。清单 1.4.3 也同时显示了如何初始化两个可线程的对象：myTestObject 和 myTestObject2。如前所述，当调用 objectThreadPump 时，将创建一个任务，它接着又从线程池中获得一个线程，然后提交一个新的任务（myTestObject 和 myTestObject2）到 GLRThreadingSystem 去执行。这些任务不立即被执行，它们只是被添加到等待执行的任务队列中。这允许一个调度器根据游戏的负载和任务的相似性来重新整理提交的任务。为了执行这些对象，一个对 distribute 的最终调用是必需的。

程序清单 1.4.3　创建和运行测试对象的代码

```
// 创建一些线程的测试对象
TestSystem myTestObject;
myTestObject.objectThreadPump();

TestSystem myTestObject2;
myTestObject2.objectThreadPump();

// 这一调用应该放在游戏的主要循环内
glrThreadingSystem.distribute();
```

在本书所附 CD 中，你可以找到一个可以编译和执行此例中代码的解决方案。

1.4.8 结论

本精粹覆盖了很多内容，包括一个实用的线程引擎的架构，它在多个平台上工作，且效率高。你也看到了几个不同的分配线程，以及查询有关任务执行的方法，并最终简要地看了一个在多核环境下设计更有效的数据结构的方法。因此，当针对多核心环境来开始开发或改造引擎时，记住本文已覆盖的一些画笔，并利用它们来开始自己的杰作。

1.4.9 参考文献

[Bevridge96] Bevridge, Jim. *Multithreading Applications in Win32: The Complete Guide to Threads*, Addison-Wesley, 1996.

[Culler99] Culler, David E. *Parallel Computer Architecture*, Morgan Kaufmann, 1999.

[Geist94] Geist, A. PVM. MIT Press, 1996.

[Gerber04] Gerber, Richard. *Programming with Hyper-Threading Technology*, Intel Press, 2004.

[Hughes04] Hughes, Cameron. *Parallel and Distributed Programming Using C++*, Addison-Wesley, 2004.

[Nichols96] Nichols, Bradford. *Pthreads Programming: A POSIX Standard for Better Multiprocessing*, O'Reilly, 1996.

[Ramsey05] Ramsey, Michael. "Parallel AI Development with PVM," In *Game Programming Gems 5*, Charles River Media, 2005.

[Ramsey08] Ramsey, Michael. *A Practical Cognitive Engine for AI*, To Be Published, 2008.

[Richter99] Richter, Jeffrey. *Programming Applications for Microsoft Windows*, Microsoft Press, 1999.

1.5　给蜜蜂和游戏玩家：如何处理六边形贴片

Thomas Jahn, King Art
tjahn@kingart.de

Jörn Loviscach, Hochschule Bremen
jlovisca@informatik.hs-bremen.de

网格是一个用来简化、模拟或可视化复杂结构和关系的最主要的工具。它在游戏中的使用包括从掌上游戏机中的 8 像素×8 像素的图形贴片到 AI 代理的空间表示。由于正方形的计算规则简单，正方形网格是一个典型的选择，但在模拟中，它们并没有表现出什么超常的行为。相反，在逻辑和外观上，六边形贴片提供了极具吸引力的特征。然而，在软件开发中，六边形网格有点棘手。本精粹将介绍处理这个问题的概念和技术。

1.5.1　简介

当一个形状或一套固定的形状被不断地重复来覆盖无限的平面，而且没有任何间隙或重叠时，贴片就被创建。贴片有两种变化——尽管非周期的贴片被用来创建纹理，大多数应用还是依赖于周期性的贴片，因为它在计算上很容易处理。常规的贴片被使用来增加对称性，其中，贴片用常规的多边形来形成，如正方形、正六边形或者正三角形。

由于它们的空间填充效率，生物学倾向于六边形网格，而非正方形网格，例如，它们出现在蜂窝结构和视网膜的光感受器的放置中。尽管六边形网格有其他一些好处，但是它需要复杂的代码，因而容易出错。面向对象的抽象可以用来解决此问题。本精粹将介绍一种软件设计来隐藏框架的复杂性。

1.5.2　六边形贴片的利弊

为了判断是正方形还是六边形贴片最适合一个任务，必须考虑从邻接性到一个坐标系选择的若干方面。

邻居和跨越距离

对于一个方形贴片，有两个常见的邻居定义。邻居要么必须有一条共同的边（4-邻居），要么只要它们共用一个顶点（8-邻居）就足够了。这种

模棱两可有其后果。以一个基于方形贴片的策略游戏为例，任何行动被分成一系列的步骤，其中一个步骤意味着从一个贴片到它的一个邻居贴片的移动。现在，作为一个开发人员，你必须做出选择。如图 1.5.1 所示，你可以只允许在 4 个方向上的移动，这意味着，沿纵向或横向轴移动将比沿对角线需要多 41%的步骤来移动相同的距离。然而，允许在 8 个方向过渡并不会好到哪里去，现在，沿着对角线移动若干步骤所涉及的距离比沿垂直/水平轴的距离大 41%。为了避免这样的扭曲，必须不同地处理对角线过渡，这增加了代码和游戏规则的复杂性。

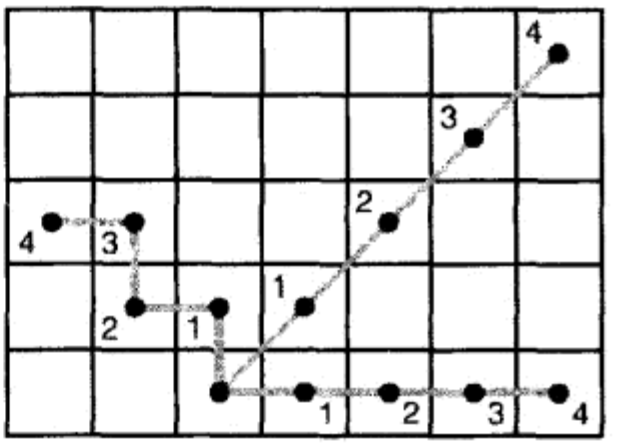


图 1.5.1 在一个方形网格上移动显示了快速和慢速的方向，不管使用哪个邻居定义

这里，六边形贴片有一个优势，每个贴片有 6 个等距离的邻居，其中的每一个连接到一条不同的边。没有两个贴片只共享一个顶点或多于一条边。因此，对于六边形网格来说，一个邻居的概念一点也不含糊。此外，如图 1.5.2 所示，在任意方向移动若干步骤所覆盖的距离只有 15%的变化范围。

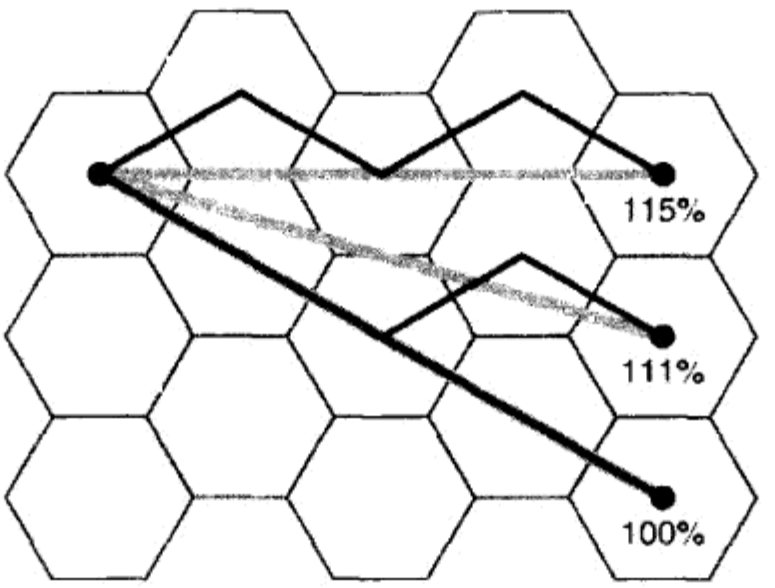


图 1.5.2 在六边形网格上，最短连接的长度接近于一条直线

各向同性和堆积密度

在所有可以贴片平面的形状中，对于一个给定的面积，常规六边形的周长最小，这意味着没有“更圆”形的贴片。因此，每当一个网格要代表一个连续且没有内在首选方向的结构时，六边形贴片是最好的。这是使用六边形像素来进行图像处理的一个重要原因[Middleton05]。

由于其紧凑的形状，正六边形用最高的堆积密度来形成贴片。一个放在正方形之内的圆盘占有其面积的 79%，而放在六边形中的圆盘占 91%。因此，用一个六边形网格，可以用少 10%的单元来达到方形网格的准确性。这是一个减少内存消耗和改善基于网格的算法速度机会，减少的数量大致相同。

外观

在游戏中，网格往往用来代表游戏场，这对外观有重大的影响。方形网格适合于建立城

市和室内场景。然而，六边形的边更平滑地连接，形成 120° 的角度。因为没有平行线段被直接连接，六边形贴片集合的轮廓具有轻微的锯齿状外观。由于此特性以及没有锋利的边，使这种类型的网格更适合于自然场景的表示，正如用户在图 1.5.3 中可以看到的一样。

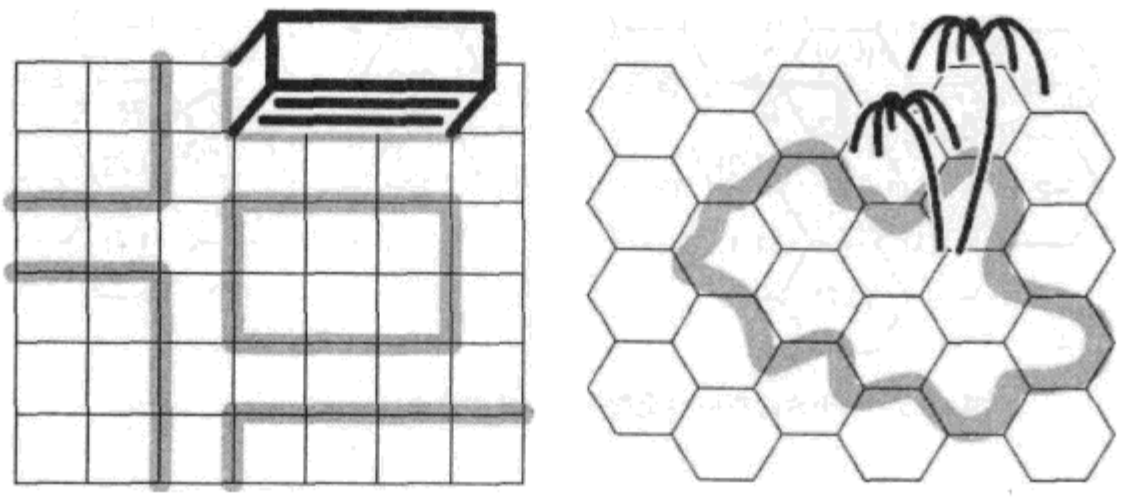


图 1.5.3　方形贴片适合于城市（左），而六边形贴片更适合于有机的形状

轴对称

方形网格可以很容易地被映射到一个普通的整数笛卡儿坐标系。这些也可以用做二维数组的索引。除了平行于正方形两侧的两个对称方向外，也有在对角的两个对称方向。虽然在实践中很少见到，但这些也可以用做坐标轴。

然而，六边形网格拥有可以用做坐标轴的、12 个方向的对称性。如图 1.5.4 所示，有两个基本的布局，一个有水平对齐贴片的六边形网格（每个第二排缩进半个贴片宽度）和一个垂直对齐的网格。

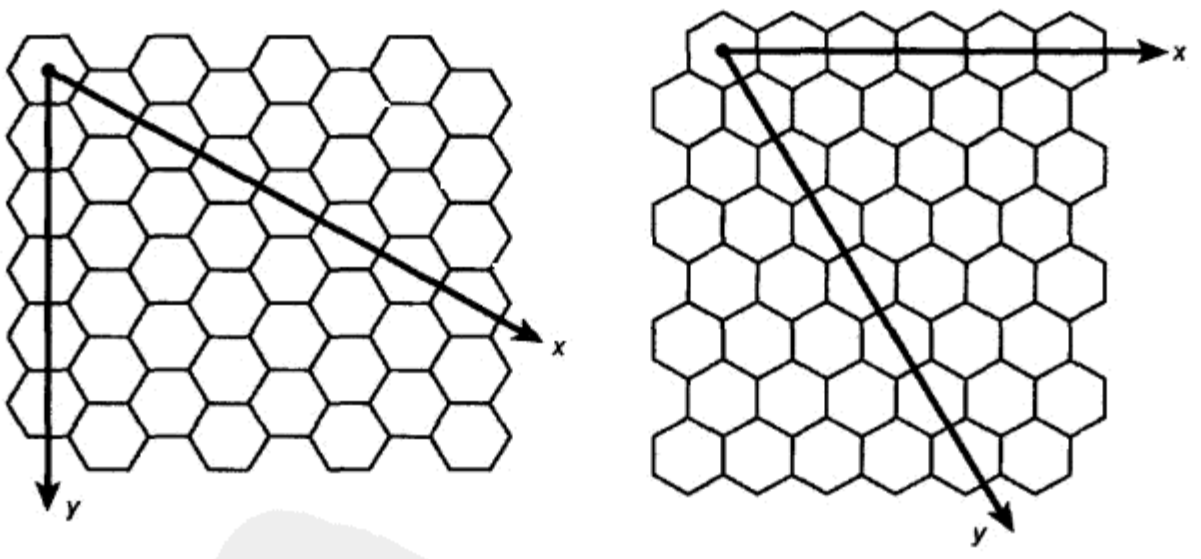


图 1.5.4　取决于哪个对称方向用做 x 轴，六边形的贴片呈现为水平或垂直对齐

用户可以选择的任何一对轴都存在两个中的一个缺陷——如果那些轴彼此垂直，在几何上，它们将不会等同，特别是其中一个轴平行于格子的一些边，而另一个不会。除此以外，用户必须和分数坐标打交道，如图 1.5.5 所示。如果那些轴确实被选择为几何等同，它们将形成 60° 的角度，这意味着，例如，距离不能天真地用勾股定理来计算。为了更好地显示对称性，用户甚至可能得用 3 个重心坐标，其中之一是多余的。

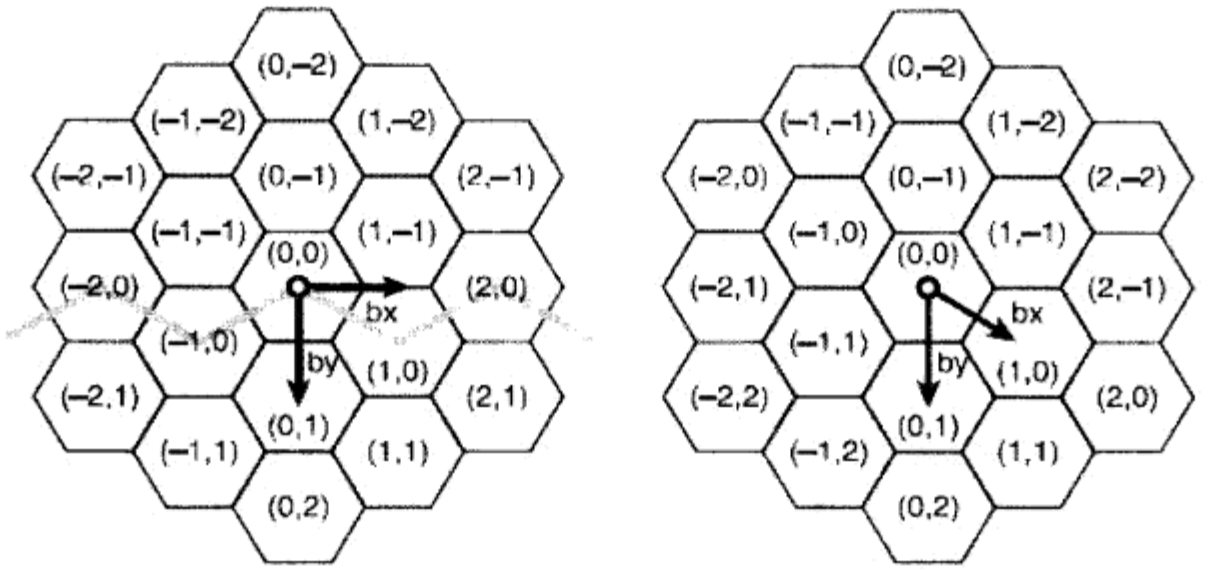


图 1.5.5 一个六边形网格允许有半整数值的垂直坐标轴或倾斜坐标轴

1.5.3 掌握六边形网格

由于面向对象的程序设计，六边形网格的问题可以隐藏在一个优雅的外观之后。实际上，本精粹提出了两个软件层：寻址和访问。它们的抽象层次逐渐增加。

地址层

每一个在地址和六边形网格上的空间位置之间的转换模式都有它的优点和局限性。因此，第一步是将寻址隐藏在一个抽象层之后。支持随机访问的数据容器类和代表寻址方案的类把网格地址映射到容器元素。

容器的第一种选择是一个被索引的随机访问容器，如 C++标准模板库（STL）的 vector，其索引可以是以垂直或倾斜坐标给出的贴片地址。因为容器的索引范围是有限的，所以地址范围也必须是有限的。如果使用垂直坐标轴，一个长方形区域的单元可以通过每个系数的上下界来定义。在这种情况下，该索引可以用 index=y × width+x 来计算。

如果坐标系有倾斜轴，这个方法将得到梯形的单元集（set of cells）。通过改变索引的计算，让索引再次指向一个长方形的单元区域，可以避免上述情况。在这个情况下，索引的计算可能类似于：index =Math.Floor(y×(width+0.5)+x)，如图 1.5.6 所示。

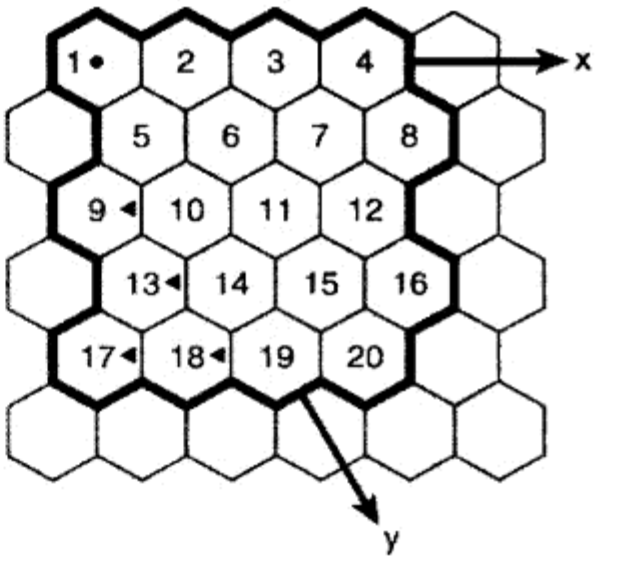


图 1.5.6 通过索引计算的一个变动，倾斜轴也可以用来定义一个长方形的区域

容器的第二个选择是一个基于键的随机访问容器，如 C++的 STL map。虽然这些容器存取数据的速度慢于索引容器，但用户可以直接用单元的地址作为键。最大的好处是地址范围没有内在的限制，而且空单元不消耗内存。因此，对稀疏数据点，map 是一个很好的选择。

为了支持一个非常具体的设置，用户可以使用包装了一个所选择的寻址方式的类的标准容器。迈向灵活性的下一步将是用类型参数将这个类泛型化，以便它不受限于可以存储什么样的数据类型。

为了实现最大的灵活性，用户可以把该功能分为两种类型：实际的存储和寻址模式。那样，有可能选择一个寻址方案和解决手头任务的容器的一个最佳组合。这里处理地址和数据之间的映射的类被称为 AddressingScheme。

访问层

第二层建立在寻址模式之上。这一层采用了迭代器（iterator）的设计模式，这也在 STL 中使用。一个迭代器作为存储在容器内的元素的指针。对于迭代器，所有的 STL 容器都提供了同样的基本接口，它隐藏了容器实现的细节。因此，基于迭代器的代码可被任何容器使用。除了灵活，迭代器也简单易用。用户可以查询任何一个容器来得到它的第一个元素的迭代器，只需调用迭代器的 next 方法，直到到达列表的最后一个元素。

一项类似的策略可以用来避免六边形网格寻址方法的很多麻烦。这里建议两种不同的方法。

- 第一个可以被称为步行器（Walker）类。其实例可以被设定来代表网格的任何单元，并提供了一个接口来读写目标单元的数据。初始化后，被引用的单元可以通过调用类似于一个迭代器的 next 方法来改变。步行器类提供了一个 move(dir)方法，接受一个指定了网格中 6 个自然方向中一个方向的参数，而不是用预先确定的顺序来遍历单元。调用此方法将使步行器对象指向由输入方向指定的旧目标的邻居，如图 1.5.7 所示。这个类提供了在网格上的自由移动，因此而得名。
- 第二种做法是用枚举器（Enumerator）类，与迭代器完全一样地工作，但只经过代表网格的一个特定子集的一系列单元。例如，只有一个给定单元的下一个邻居们。该框架提供了枚举器来遍历不同的邻居，甚至于一些自定义的（如在一给定中心单元的）在某一半径之内的所有单元，或当前在屏幕上的所有可见单元。在一个策略游戏中，一个枚举器可以提供对一个特定单位的攻击或可视范围之内的所有单元的访问，或者是被敌人占领了的所有单元的访问。从实际游戏逻辑中解除耦合网格的逻辑使代码更加简洁，更易于维护。

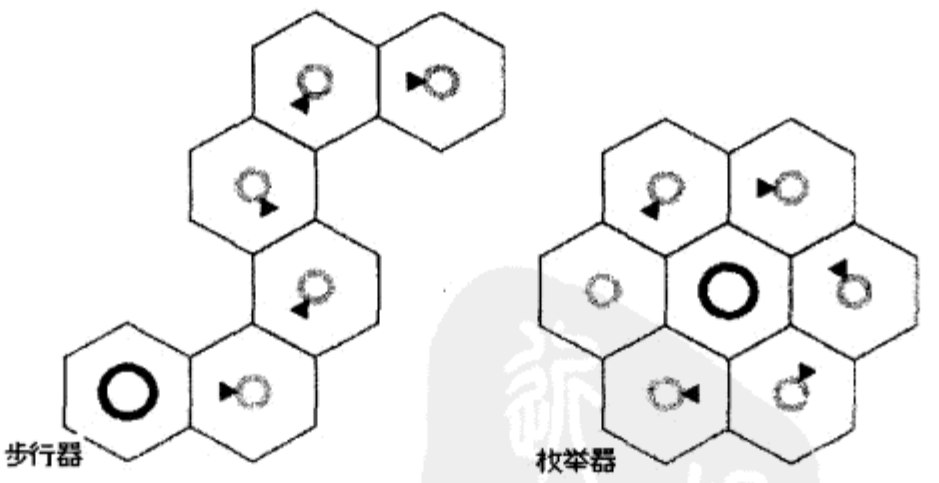


图 1.5.7　步行器可以访问贴片的任何邻居，而枚举器可在贴片附近以预定义的模式遍历

1.5.4　实现技巧

许多算法从来不需要知道实际地址，它们可以使用步行器对象来读/写数据。因此，我们

建议让尽可能多的代码基于一个抽象的步行器基类，这个基类定义了一个共同的接口，但不依赖于特定的寻址方案。在决定想要哪一种寻址方式和实现对应的AddressingScheme类之后，就可以写一个从抽象类继承的兼容的步行器。

另一个建议是使用泛型（generics）来实现这些类，如C++的类模板，这将允许指定独立于数据类型的数据访问。

C#和Java（但不是C++）提供了具体的实现接口，从而促成了客户端的整洁代码。例如，通过实现C#的IEnumeration接口，有可能用和通常的for循环有相同语法的foreach循环（不像C++ STL的for_each）来遍历具体选定的所有单元。假设网格的一个单元中的类型是CellData，并且有一个叫做Neighborhood的泛型枚举器和一个步行器类CellPointer，通过传递一个定义了你想要邻居列出哪些邻居单元的步行器实例中心，Neighborhood的实例可被创建和初始化。因为Neighborhood实现了IEnumeration接口，便遍历中心单元的所有邻居变得如此简单：

```
foreach(CellPointer<DataType> cell in new Neighborhood<DataType>(center))
{
    // 对单元做些事情
}
```

AddressingScheme的核心功能是在一个网格寻址层提供数据访问。然而，这个类可以提供额外的功能。许多应用情形要求用户基于屏幕或世界坐标来找到一个单元。对矩形网格，这是微不足道的，因此我们给出以下的建议。将一个六边形网格分区成如图1.5.8所示的矩形区域。为了求解一对坐标*xy*，第一步是决定该点所在的区域。一个区域有两种可能的布局，在这两种情况下，它都被分为3个子区域，每一个与不同的贴片相联系。一旦用户求解坐标到一个区域的某个位置后，那就只剩下3个选择，进而去计算正确的单元地址就变得轻而易举了。

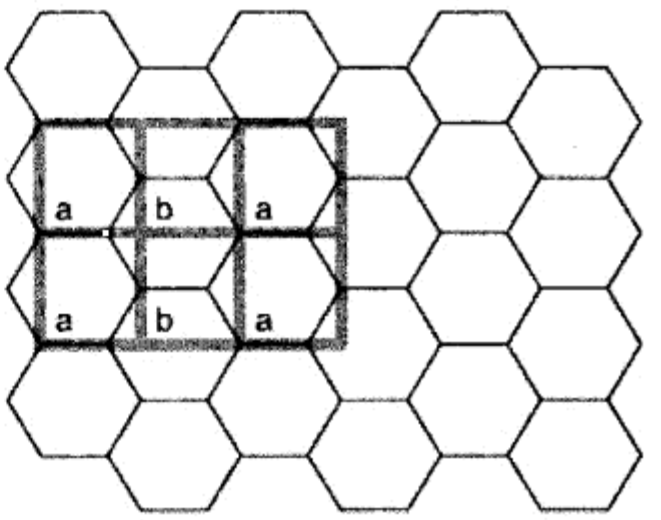


图1.5.8 把网格划分成矩形单元，允许简单的击中测试计算

1.5.5 应用

为了显示出一些实际的好处，可以考虑3种六边形网格可能被使用的3种情况。

空间搜索

一个游戏世界由许多实体组成，当它们在对方的一定范围内时，它们就有相互作用的能力。当一个具体的实体需要决定互动是否可能时，去考虑其他所有的活跃实体在计算上将会很昂贵。相反，一个网格可以用来在一定的半径之内预选对象。网格把游戏世界分成和单元一样表现的贴片。要查找在一个游戏对象的、一定半径内的所有实体，去考虑那些与单元注册的对象就足够了，这些单元包含搜索半径内的点。

如果被搜索的区域是圆形的，六边形网格将是最佳选择。此外，用足够的抽象来执行搜

索代码可以很简单。

```
foreach(CellPointer<List<GameObject>> cell in new SearchZone<List<GameObject>>
        (center))
{
    foreach(GameObject obj in cell.GetData())
    {
        // 用 obj 做些事情
    }
}
```

每个单元包括一个 GameObject 的列表。一个枚举器类的子 SearchZone 被定义成允许遍历在搜索区的所有单元，它产生一个指向特定单元的步行器对象。因为单元的数据是 GameObjects 的一个列表，另一个 foreach 循环可以用来遍历与此单元相关的游戏物体。

路径搜索

一些游戏使用贴片作为它们游戏世界的基石。如果代理必须在世界内移动，路径搜索必须在网格层进行。只有在相邻的贴片间才可能移动，有一些邻居甚至可能被阻塞，例如，因为它们包含了墙壁。这就需要在面向对象的框架中实现一个标准的算法，如 Dijkstra 的算法或 A*算法[Mesdaghi04]。

基本思路是扩展起始节点，直到实现此目标。在寻址层，这是不方便的。必须对当前单元作用 6 种不同的偏移量来访问邻近单元。对一个正交的寻址模式，根据当前的单元是在一个偶数还是奇数行（或列，如果网格垂直对齐），这些偏移量会不同。

下面的程序清单在一个单元网格上执行了一个简单的宽度优先搜索，来寻找连接 startCell 和 goalCell 的最短可移动单元序列。一个单元的邻居用一个匹配的移动邻居枚举器来访问。由于面向对象的抽象性，这个算法就独立于一个具体的网格布局或寻址模式。

```
Queue<CellPointer<PathCell>> openCells = new Queue<CellPointer<PathCell>>();
openCells.Enqueue(startCell);

// 扩展
while(openCells.Count > 0)
{
    CellPointer<PathCell> current = openCells.Dequeue();
    foreach(CellPointer<PathCell> cell in new Neighborhood(current))
    {
        if(cell.GetData().Moveable && cell.GetData().ExpandedFrom == null)
        {
            cell.GetData().ExpandedFrom = current.GetData();
            openCells.Enqueue(cell);
        }
    }
}
// 求解
Stack<PathCell> path= new Stack<PathCell>();
PathCell pc = goalCell.GetData().ExpandedFrom;
while(pc != null && pc != m_Start.GetData())
```

```
{
    path.Push(pc);
    pc = pc.ExpandedFrom;
}
```

细胞自动机（Cellular Automata）

通过让一个几乎无限多的简单部件在局部相互作用，细胞自动机[Wolfram02]可以用来建模和模拟复杂的动态系统。在一个二维设置下，相互作用的部件通常是在一个网格上的单元，其中，一个单元的下一个状态是基于它自己的现有状态和邻近的单元计算出来的。

六边形网格的方向不敏感性也使它们对细胞自动机颇具吸引力，例如对液体的模拟。对确定一个单元的下一个状态的单元集合，显而易见的选择是细胞本身及其 6 个直接邻居（见图 1.5.9），即使有时候使用其他邻居。一个只有 3 个邻居的较小集合可能就足够了，并允许更快的仿真。在其他情况下，6 个邻居可能无法提供足够的数据，所以选择去扩展到最接近的 12 或甚至 18 个单元。

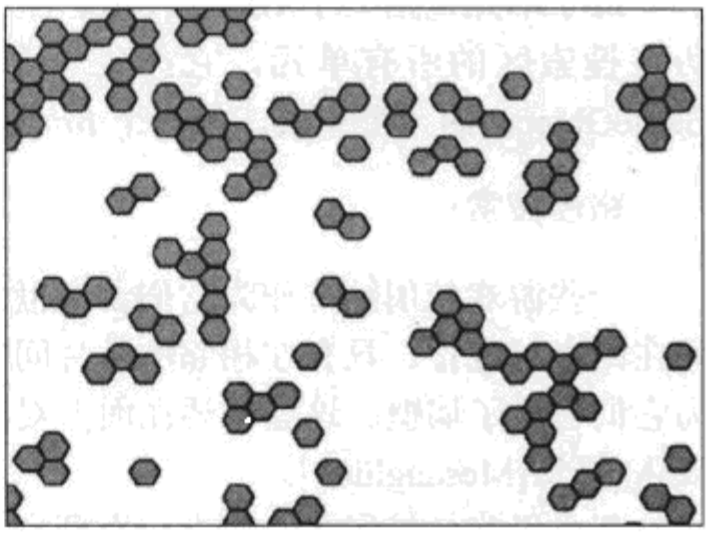


图 1.5.9 从一套简单的规则开始，Conway 的 Game of Life（经典的二维细胞自动机）可被移到六边形贴片

如果仿真代码直接作用于单元地址，则很难用不同的影响单元来实验。但是，如果枚举器提供了影响单元的集合，即使在运行时，也可以轻松地改变单元的选择。

1.5.6 结论

通过在寻址层上引入另一抽象层，就有可能写一些与寻址方法和单元数据的存储高度解耦的代码。这不仅增加了可维护性和灵活性，而且还极大地简化了和六边形网格相关的工作。用户可以让游戏逻辑保持干净，而不存在所有让六边形网格用起来很烦琐的讨厌细节。

1.5.7 参考文献

[Mesdaghi04] Mesdaghi, Syrus. "Path Planning Tutorial," *AI Game Programming Wisdom 2*, CD-ROM, Charles River Media Inc., 2004.

[Middleton05] Middleton, Lee, and Sivaswamy, Jayanthi. *Hexagonal Image Processing—A Practical Approach*, Springer New York Inc., 2005.

[Wolfram02] Wolfram, Stephen. *A New Kind of Science*, Wolfram Media Inc., 2002.

1.6　服务于即时战略游戏的基于细胞多孔机器（Cellular Automaton）的线条主界面

Carlos A. Dietrich
Luciana P. Nedel
João L. D. Comba

即时战略游戏（RTS）是世界上最流行的游戏之一。把动作和战略结合起来虽然只是一种简单的组合，却引来了无数狂热的玩家整日整夜地在网络上为了冠军而追逐厮杀。

不过这几年我们倒没看到 RTS 游戏有多少长进。把这几年的几个游戏和早期几个游戏进行比较，你可能会说屏幕上的元素越来越多了（可能是几百个之多），图形引擎更漂亮了，战场更宽广和美丽了，但是游戏性呢，还是那个样子——选择军队，用鼠标单击来定义它们的任务。这个过程需要一个很简单且有效的界面来操作，而且玩家要很熟悉。但是当游戏需要更多的功能时，如何用一种真实且有效的方式来控制数以百计的军队？另外，这个界面是为手动操作的战斗来设计的，当军队变得很多，又没有很明显的路线来引导这些军队时，该怎么办呢？现在我们开发的游戏就遇到了这样的情况，尽管目前我们玩的游戏都提升了许多，但是普通军队单位控制的界面有时还是会让玩家觉得沮丧。

本精粹将介绍一种替代原始操控的方法，这种方法很可能会提升游戏的可玩性。我们提出了一种一键式的高级功能界面，它可以控制整个军队或一群士兵的移动。这种方法背后隐藏的点子其实很简单，可以对历史上的任何战斗进行描述，就好比在图 1.6.1 所示的例子一样。

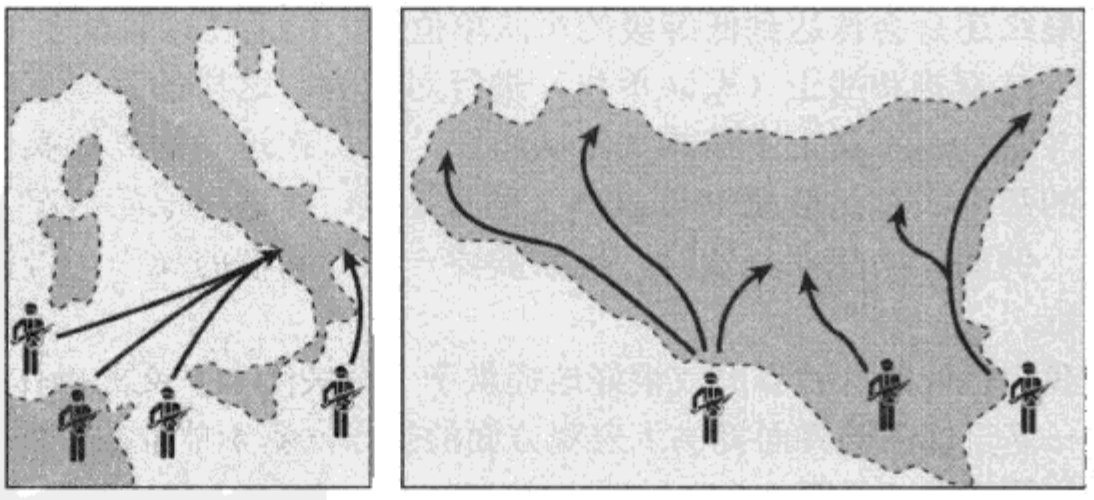


图 1.6.1　在第二次世界大战期间的 1943 年，入侵意大利（左）和西西里（右）过程中军队的移动

在图 1.6.1 中，部队的移动情况是由箭头来描述的，这些移动会是军队中的某些部队，也会是整条战线。注意，图中没有对某个单独战士的个别任务进行描述的信息。为什么会这样？在战场上，你基本不可能吼着某个士兵的名字，并且对他分配攻击这个或者那个敌军部队的任务。一般都是部队指挥官把任务分配给营长，营长再把任务分配给每个连长，然后就是前线队长，最后才是前线队长把这些命令告诉每个战士。

我们设计的工具就是希望能够模拟这个流程，允许玩家画出这个流程中的路径和目标，然后其中的每个部队都要听从这样的指挥。用户界面其实很简单：使用鼠标在屏幕上画线（或者是点），在战场上建立一条路径（或者一个目标），如图1.6.2 所示。这条线在战场上建立了一个“影响区域”，该区域的每个部队单元都要跟随这条线或者这个目标。

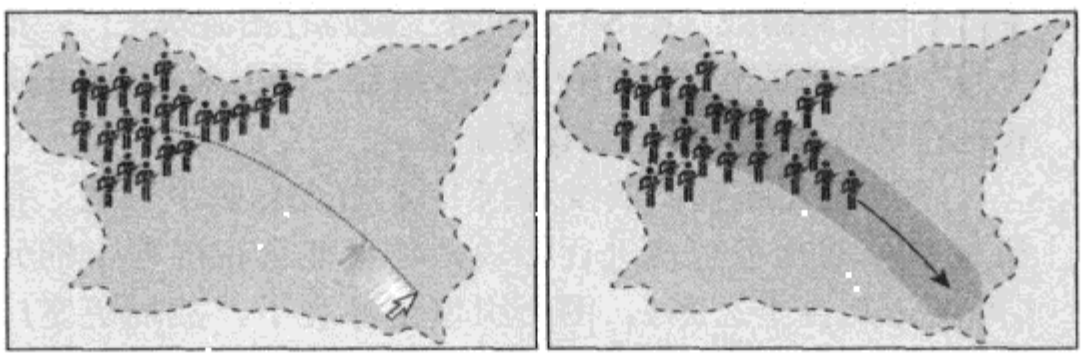


图 1.6.2　通过使用鼠标，玩家在屏幕上画出了一个线条，把线条所影响区域内的军队单位都推向了玩家希望的目标

最近的 RTS 游戏[Bosch06]中有一个工具也采用了类似的界面。虽然这样已经提升了许多，但是我们相信还有很大的提升空间，主要就是界面等这些基本设施的实施。我们这样做的目标也就是尽可能使用更多的动态控制来扩大这一提升的路径。在游戏战斗内设计并提升游戏性和战略计划可以为游戏提供更精彩的内容。下面是总结我们的方法，以及说明一些验证的例子。

1.6.1　关注上下文的控制等级

在军队编制的结构上，将军并不直接和战士们对话，但是沿着命令转达的线路，他们的命令最终还是会传达到低等级的军队单位上。但是在现代战争的 RTS 界中，将军（也就是玩家）是直接和战士（军队单位）进行对话的。这样的界面把玩家的注意力都集中到手动的战斗上，而不是上下结构（军队组织）上。在这样的界面操作下，不管玩家有多棒，按得快的那个人总是会赢[Philip07]。也就是说，在这种原始情况下，玩家必须操作游戏中的小兵。例如，当战斗开始时，玩家需要一系列细节的操作来告诉自己控制的单位如何去袭击目标。

而我们建议的这种界面就很好地提供了一个关注上下文的操作，它把两者很好地捆绑了起来——基于线条的界面提供了宏观方面的操作，基于单位的界面提供了微观方面的操作，例如单位的移动（见图 1.6.3）。

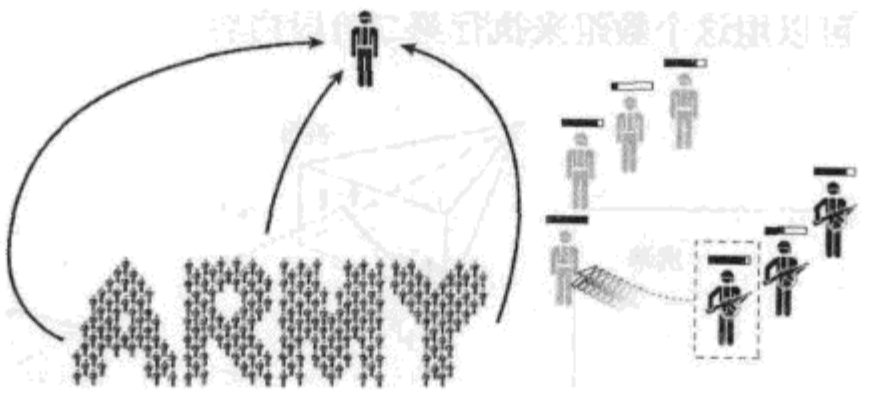


图 1.6.3　关注整体结构的界面：将基于线条的界面（军队部署，左图）和基于军队单位的微操界面（手动战斗，右图）很好地绑定在一起

从远视角能很容易地看到如何去方便地使用线条来布置军队的位置，而当小兵进入战斗以后，你又可以转换成基于军队单位的界面来控制战斗。

1.6.2　实现细节

我们有许多种方法来实现基于线条的界面，其中最重要的一个方面就是采取一种最有效的手段，让玩家的意图在战场的军队单元间传达。这里我们提出一种基于细胞多孔机器的实现方法[Weisstein07b]，这种方法很快也很简单，可以加到游戏引擎中去。

这种实现有两个主要的步骤：

- 获取用户输入；
- 战场内的命令传播。

获取用户输入很简单，我们会在下面的“画路径线条”部分进行介绍。处理用户命令使用的就是细胞多孔机器，由它在战场中迭代地传播命令。这个细胞多孔机器在“移动士兵”部分也会被讨论到。最后，在“综述”部分，我们会介绍如何在游戏界面中使用该方法。

画路径线条

像前面说的那样，我们提出了一个通过玩家直接在战场上画曲线或者点来控制一个军队的界面。这个实现很直接，而且必须完成两个任务：

- 截取用户输入的屏幕坐标；
- 把这个坐标投影到战场上。

在第一个任务里，让我们假设玩家简单地在屏幕上单击并拖曳鼠标（形成一个路径），或者在屏幕上单击鼠标（形成一个目标）。这个操作产生的结果其实就是一组包含屏幕二维坐标的数组（见图 1.6.4）。把这些点保存好，不要管它们是不是能够形成一条封闭的线段，然后进入下一步。

第二个任务就是把每个二维的点投影到三维的战场上。使用标准的图形 API，例如 OpenGL 的函数 gluUnProject，就可以用变换以及视口矩阵把窗体坐标映射到对象坐标上去。这么做时一定要小心战场上的障碍物和在战场上没有对应位置的屏幕坐标。要解决这个问题，可以先把战场用低分辨率渲染一下，然后把生成的深度缓存作为 gluUnProject 的输入。这个方法可以方便地测试那些无效的投影以及减少战场障碍物的干扰。最后的结果会产生战场上

的一个 3D 点的数组，可以用这个数组来执行第二阶段内容。

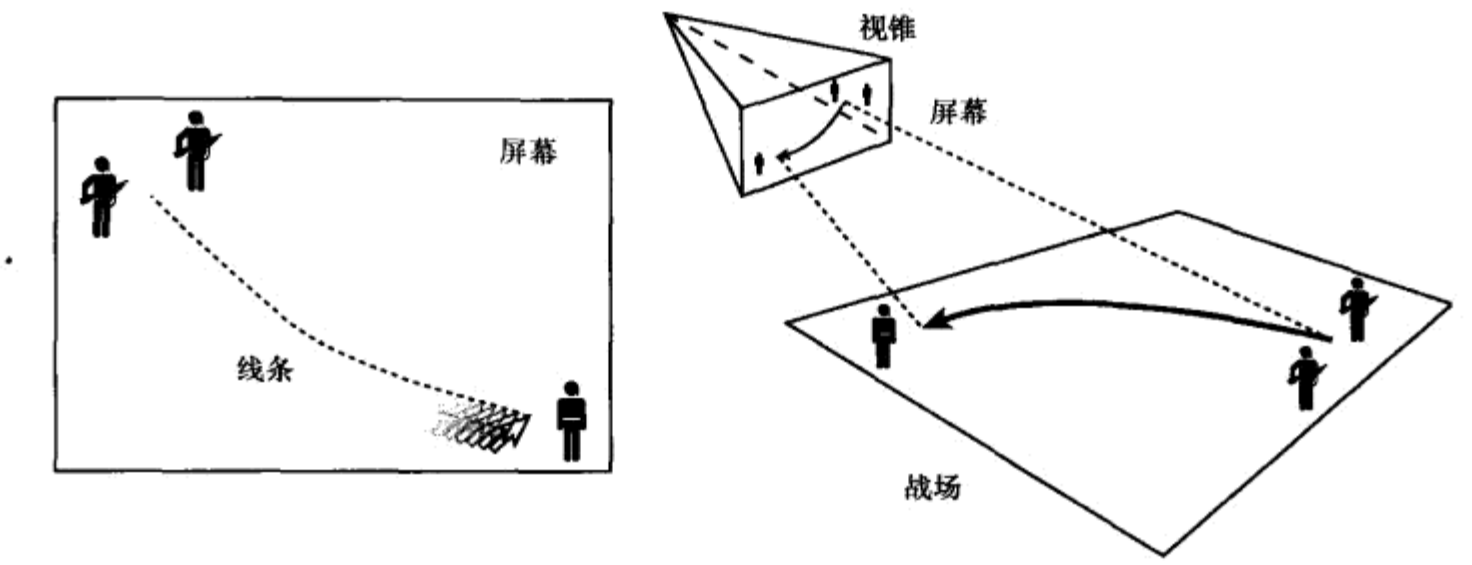


图 1.6.4 在第一阶段的实现中，线条的 2D 坐标（左图）被捕获并投影到战场上（右图）。投射的结果就是一个 3D 坐标的数组，这个数组就是细胞多孔机器的输入

移动士兵

第二阶段的任务就是让军队单位对线条做出响应。在这个方法中，每个线条都被转换成推动军队单位移动的力量，把这些军队单位移动到战场上玩家期望的那个位置。这里运用了一个离散的网格系统，在网格的每个格子中都用一个向量来描述这股力量的方向。当然，这股力量每时每刻都会根据玩家画的线条来改变。如图 1.6.5 所描述的，离线条近的格子，在上面作用的力量会更强一些，这股力量根据距离线条的远近而线性衰减。这就把线条范围的概念带入了游戏中，也就把真实战争中的命令传播的实际情况联系起来了。

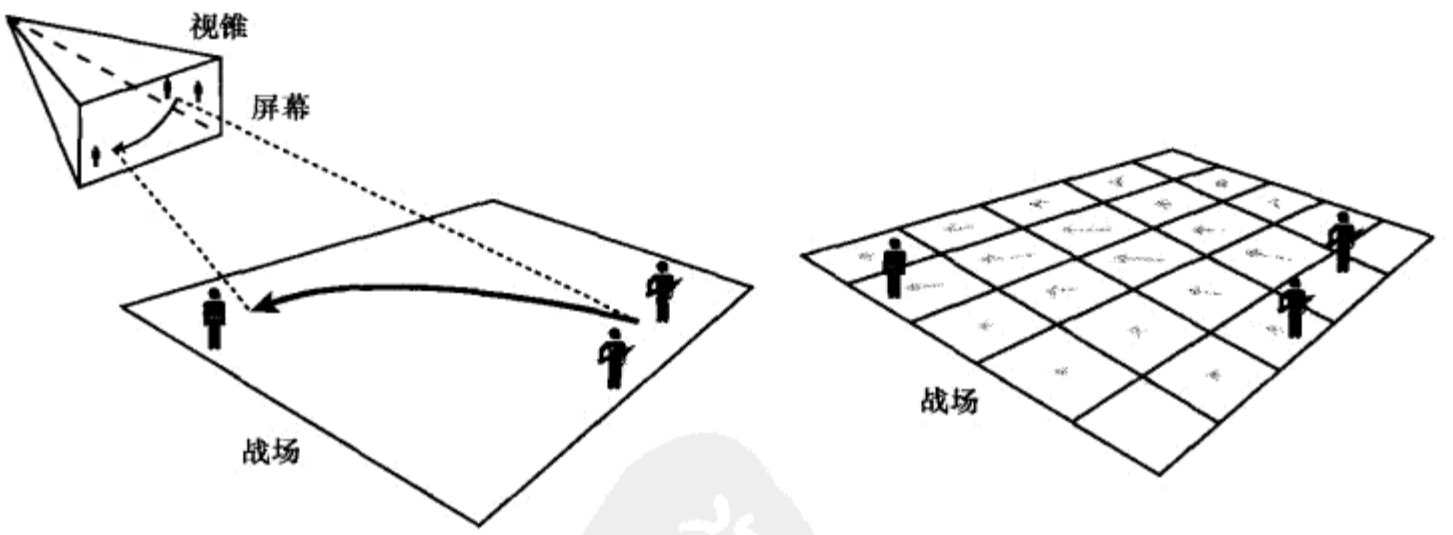


图 1.6.5 线条（左图）以及被离散化成作用力的格子的战场效果（右图）。作用力影响军队单位的移动，把它们推向玩家期望的位置

提出的表示方法以及线条的更新可以通过细胞多孔机器很有效地完成。细胞多孔机器其实也就是一个很简单的格子系统，这个格子系统里面包含了根据一套规则而随时间演化的信息而已[Weisstein07]。基于存储在邻居格子里的信息，每条规则都被用来为每个格子做评估。

图 1.6.5 显示了一个长方形的格子系统，里面的每个格子都是正方形，这是细胞多孔机器的一个基本配置。格子信息的更新是由一个很简单的集权规则来形成的。如前所述，每个格子都保存了一个力量数值（向量），这都是由格子上线条的位置和方向给予的（见图 1.6.6）。这些力量在战场上扩散开来，并且根据距离线条的远近而衰减。我们通过使用一条规则将每个格子的状态迭代地传播到周围的格子上去，直到整个系统达到一个平衡。每个格子的更新总是和周围格子的平均量相关。

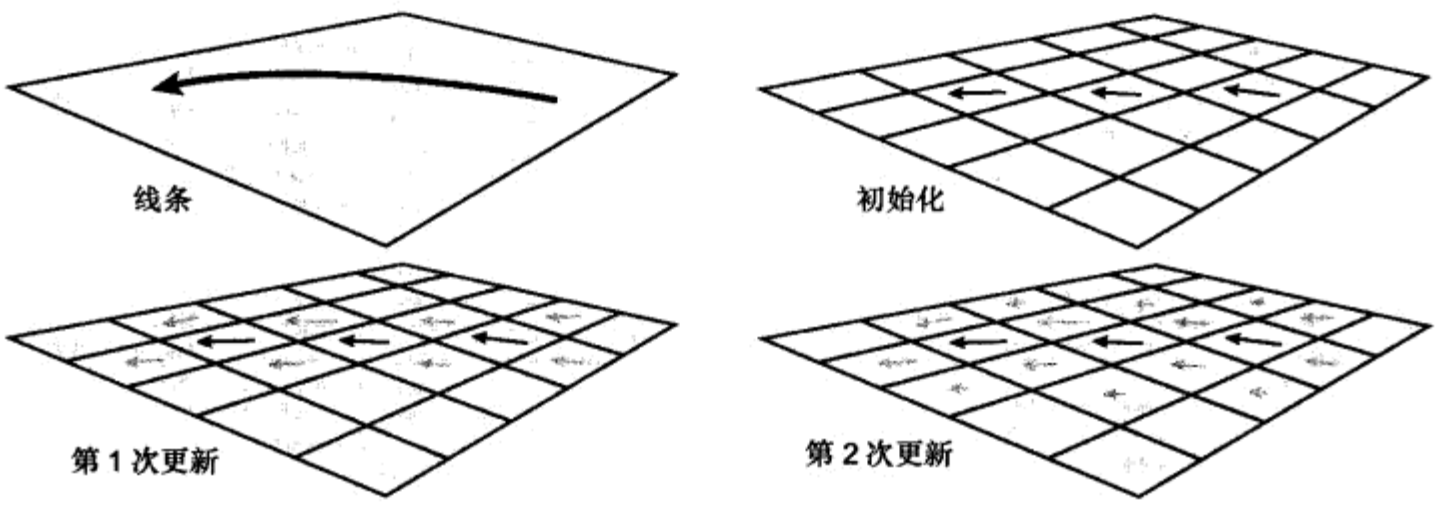


图 1.6.6　线条的 3D 坐标（见图 1.6.4）在细胞多孔机器格子里面被转换成 2D 坐标。格子里面的每个和线条相关的点都被标记上了力向量，这个向量在每次更新时都被很平滑地迭代过去

前面我们说过，这样的机器被正式地称为集权细胞多孔机器（totalistic cellular automaton）。在这个系统中，你有一个连续的状态范围（因为每个格子上的力的大小都可以是任意的）、一个简单的邻居关系（你只是根据相邻的格子来更新当前格子的状态），以及依靠周围格子的平均值来定义当前格子的数值的规则。最终，我们得到的就是集权细胞多孔机器[Weisstein07b]，这是很简单的技术，却有一个很绕口的名字。

综述

为了在真实的 RTS 环境中使用这套机制，你需要找到一个入口线索，那就是通过线条来定义你的命令，并且把目前基于单位的界面整合进基于线条的界面中去。我们推荐把一些有用的命令放到力量格子系统中去，同时也建议使用简单的方法把基于线条的单位管理整合进现有的系统中去。

在 RTS 的游戏中，你会经常指挥自己的军队在战场上移动，比如指引他们转圈子或者找掩体，直到你发现一些让你感兴趣的目标。那这两个命令（指引和单击目标）就自然而然地需要放到力量格子系统中去了。指引单位可以通过线条来完成，线条可以转换成格子中的力向量（见图 1.6.7）。然而，确保线条是由网格上足够数量的点来表示的，是有必要的。你可以通过把点之间定义的线条光栅化，然后直接覆盖到格子上来完成这点。

在战场上单击目标可以通过一个小圈的线条（或者甚至是一个点）来完成，这个线条或者点就会被转化成一组围绕着线条的向量，它们都指向线条的中心点，如图 1.6.7 所示。线条周围向量的更新创建了一个指向线条中心点的向量场，这也会把军队单元推

向目标。

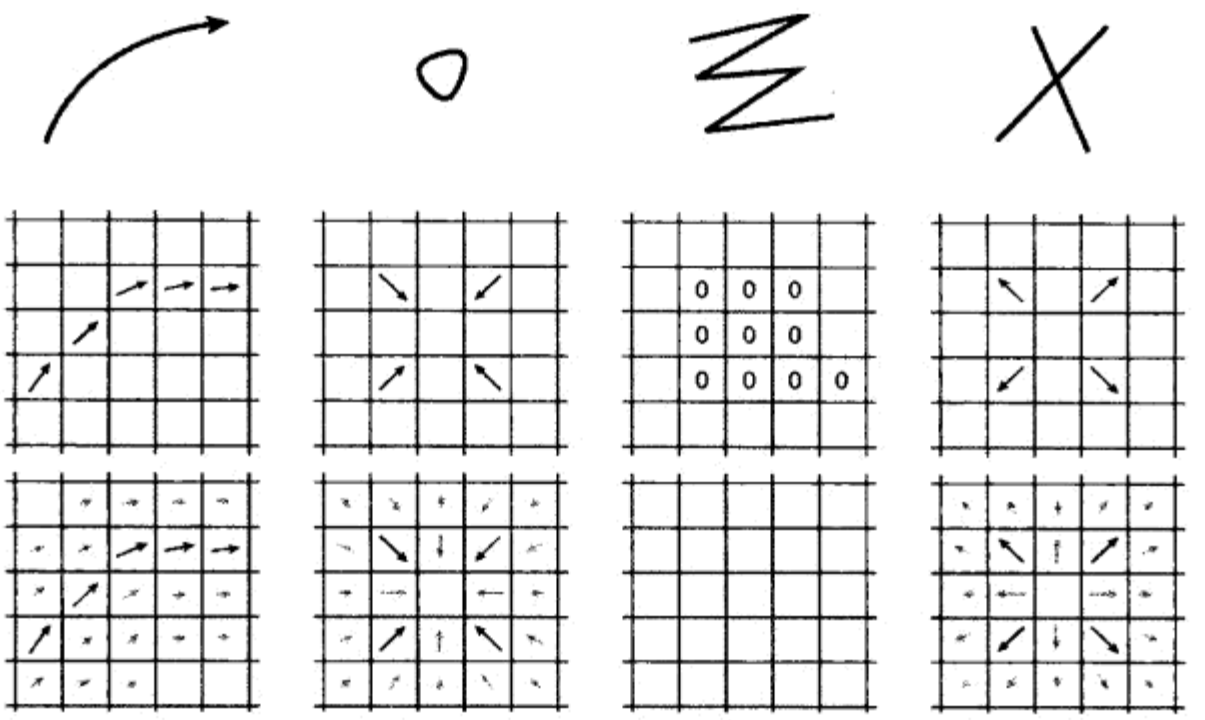


图 1.6.7 不同类型的线条（第一行），这些线条在格子中的描述（中间一行）以及得到的向量场（最后一行）。线条的轨迹被保存在格子中，每时每刻对向量场进行自动更新。“删除”命令（在第三列里）是一个特殊的例子，在那里线条删除了网格的力量值

我们很容易就可以发现，在格子中插入一些向量并不能满足建立一个稳定的向量场的需求。更新机制在每次更新时平滑地对格子中的内容进行更新，但是这些信息会很快地消逝，因为第二次更新马上就来了。为了避免这种情况的发生，我们建议采取定义命令生命周期的这种做法。命令的生命周期就是我们给每个格子一个数字，在这个数字定义的时间内，我们不会去更新命令，这样我们就有足够的时间让命令扩散到周围的格子中去了。生命周期可以根据命令的不同而不同，一般来说线条长的，生命周期就长（给予军队单位足够的时间穿越战场），并且短线条标识了目标或者更小的移动。

基于线条的界面和现有的基于单元的界面的整合其实很简单。原因就是这两种实现方法都很独立（互不影响）。基于线条的方法只是在单位移动的方程式上增加了一个新的项目而已，这个项目就是标识移动方向和距离的向量。所有的向量都被保存在格子中，格子和战场的映射是一一对应的，这也就是说任何一个军队单元都可以通过战场的数据来查询格子的数据，并且找到自己应该去的地方。格子的更新可以是并行的，因为它是独立于任何其他过程的。也就是说，你可以把线条控制封装成一个黑盒子，只需要留出一个接收 2D 坐标数组的接口就可以了。然后，这个线条控制就能够提供力量的查询，并找到战场的位置（见图 1.6.8）。

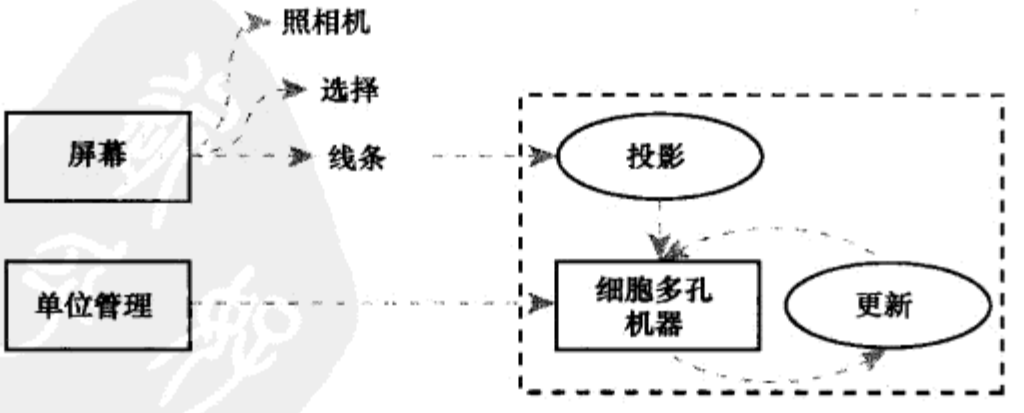


图 1.6.8 将基于线条的界面和基于单元的界面整合在一起的方法。线条控制可以看成是一个黑盒子，这个黑盒子接收 2D 点数组，并且可以提供任何战场位置上的力量查询

1.6.3　结论

本精粹讨论了一个实现基于线条的界面的简单有效的方法。效率主要是从被使用的算法的简单性上得到的（例如点的投影和格子的更新），这种方法为所有的 RTS 游戏引擎提供了一个简单的接口。然而我们发现，选定的网格密度在很大程度上决定了系统的效率和内存需求。在实验中，我们找到了在合理的线条笔画和性能（或者内存消耗）两者之间一个好的折中点，就是采用很小的格子（30×30 或者 50×50 的格子）。也就是说，太大的格子系统的代价通常会太昂贵。

我们的想法也可以很方便地扩展到其他的格子系统上去，这对其他应用程序可能会很有用。例如，你可以采用六边形的格子取代矩形的格子，这样在移动方向上就不会出现重复，你也可以采用更多不规则的格子来描述战场上那些不可跨越的地区，例如山脉和河流。采用其他类型的格子所需要注意的唯一一个地方就是格子之间的更新规则，即通过获得格子的邻居信息来简单地同每种不同类型的格子相适应。

1.6.4　参考文献

[Bosch06] Bosch, Marc ten. “InkBattle.”

[Philip07] Philip G. “Too Many Clicks! Unit-Based Interfaces Considered Harmful.”

[Weisstein07] Weisstein, Eric W. “Cellular Automaton.”

[Weisstein07b] Weisstein, Eric W. “Totalistic Cellular Automaton.”

1.7 第一人称射击游戏的脚步导航技术

Marcus Aurelius C. Farias
Daniela G. Trevisan
Luciana P. Nedel

在第一人称射击游戏（FPS）中的交互控制是一件纷繁复杂的事情，通常是通过同时控制鼠标和键盘来完成的，你还必须记下一些快捷键才行。因为 FPS 游戏是一种角色在虚拟世界中移动的游戏，所以左右手的合作（键盘和鼠标）就成了导向和行动的基础。本精粹提出了一种根据玩家脚的位置来引导移动，并空出双手来做其他动作的方法，例如空出双手做射击、武器选择或者物体操作。

这种基于脚步的导航技术能够提供前进后退、左转或者右转以及加速控制等功能。跟踪脚的技术可以由任何一种动作捕捉设备来完成，而这种设备至少提供两个自由度：一个是移动，另外一个是旋转。当然，提供的功能越多越好。

1.7.1 介绍

我们通过两种途径来实现导航技术。第一种方法就是采用一种非常精密的磁性跟踪器（Ascension 科技公司的禽类群聚技术）来捕捉脚的移动和旋转（见图 1.7.1 中的例子）。尽管通过这种技术可以得到非常精确的数据，但是该设备因太昂贵而不适合家庭用户。

既然如此，我们就对这个问题采取了低成本的无线设备解决方案。在第二种方法中，我们使用了 ARToolKit（一个开源库）和一个普通的摄像头来捕获并识别贴在玩家脚上的一个颜色标记的移动和定向（见图 1.7.1）。因为这种方法可以很方便地实现，而且不用多高端的编程技术，所以我们会在本精粹中详细地介绍这种方法，而不是第一种方法。不过第一种方法的源代码还是能够在光盘中找到的。

后面的章节将向大家介绍基于脚的导航技术的基础，以及如何在计算机上实现这个技术（即探索 ARToolKit 的特性）。本精粹也向大家提供了一个示例游戏，帮助大家测试这个技术的可用性和精确性。根据用户测试显示，因为大多数玩家都习惯于使用键盘鼠标来玩这样的游戏，所以当他们用脚来控制时会很不习惯，不像先前那样又快又准确。不过，他们都在规定的时间内完成了游戏任务，而且可以轻而易举地躲开那些障碍物。这个

结果给我们的鼓励很大，让我们相信玩家在经过一定的训练之后，可以快速地提高他们的适应能力，并逐渐对这个新的互动技术熟悉起来。同样的事情也发生在任天堂的 WII 手柄上。

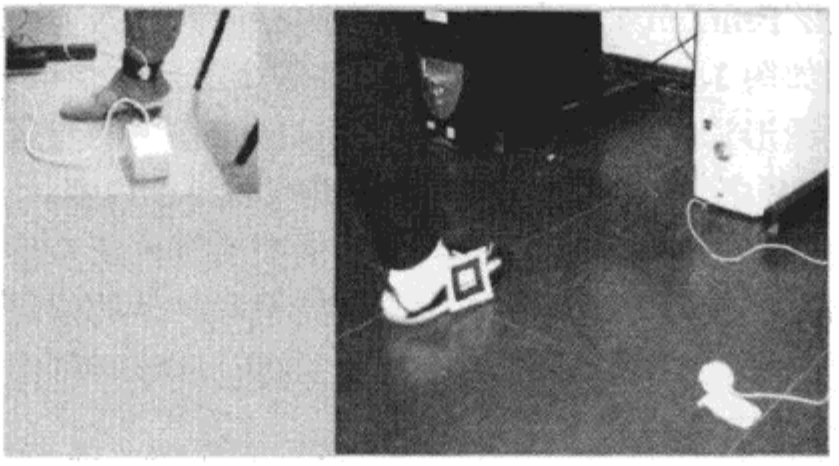


图 1.7.1　使用禽类群聚技术的动作捕获器（左图）和把一个方形标记贴在脚上的摄像头捕捉技术（右图）

1.7.2　用脚来导航

我们推荐的导航技术允许玩家用他们的一只脚在 FPS 游戏中控制他们的移动速度和方向。首先，玩家可以选择是站着玩还是坐着玩。然后，为了以一个固定的速度开始行走，游戏玩家必须把他们的脚向前移动（见图 1.7.2(c)）。如果玩家把脚迈得更靠前，那么在虚拟环境中的角色的移动速度就会更快。要停止移动，玩家只要把脚放回原地就可以了（见图 1.7.2(b)）。向后走就是把脚往后稍微放一点（见图 1.7.2(a)）。如果玩家想要左转或者右转，只需要把脚往左或往右转就可以了，见图 1.7.2(d)～图 1.7.2（f）。

下面将介绍如何使用计算机视觉来实现这个导航技术。

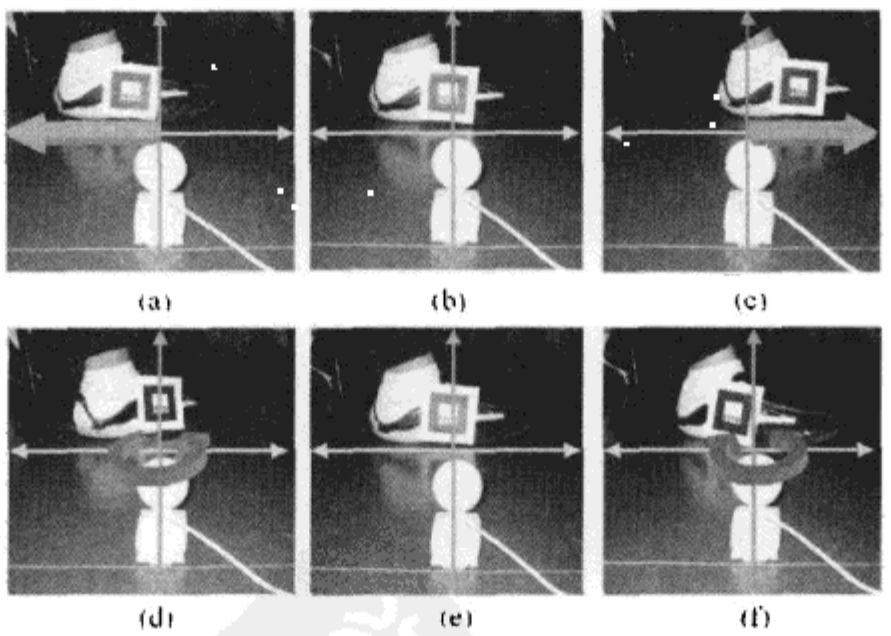


图 1.7.2　使用右脚的游戏导航控制：后（a）；休息位置/重置位置（b）；前进（c）；左转（d）；休息位置/重置位置（e）；和右转（f）

基于计算机视觉的实现需求

因为计算机可以解释用户的移动、手势和眼光，所以计算机视觉是促进人机互动的一个

潜在的强有力工具。我们已经有许多视觉相关的基本算法了，如跟踪、形状辨识和动作分析等。本精粹推荐使用由 ARToolKit 提供的基于标志点的方法，ARToolKit 是一个开源的库，被用来创建增强现实的应用程序，我们用它来捕捉玩家脚的移动。

我们使用的是 3GHz Pentium 4 CPU 和 1GB 的内存，以及 NVIDIA GeForce 5200 显卡。在这个硬件环境下做到如图 1.7.1 所示的情况，比如这个图是 320×240 大小的，那么我们在一毫秒内就可以做一次图形识别工作。这个过程不包括游戏中的一些响应延时。我们可以在 ARToolKit 的网站上找到更多关于性能和最小配置需求的详细内容。

ARToolKit 能够跟踪一些特殊标记（中间有个图案的黑色正方形，这个很容易绘制）的位置和方向，通过这种方法就可以很容易地对玩家的手和身体的位置做出交互式的响应。在这个例子中，将绘制出的标记贴在玩家的脚上，而且标记应该时刻出现在监控摄像头的可视范围之内，就像在图 1.7.1 中所示的那样。

下面描述的技术需要用户把在文件 hiroPatt.pdf 中定义的基准点给打印出来，光盘中有这个文件。如果贴在脚上的是一张很平的硬板纸，图形识别的效率会很高。总之，你要保证识别的图形是在一张很平的纸上。

当你了解到如何把需要的信息从 ARToolKit 上获取下来时，做交互的实现就变得很简单了。下面的第一步就是要检查脚的旋转方向，看看是向左还是向右，是指向前进的还是指向后退的。取决于用左脚还是右脚来控制程序，对于不同的方向可能需要不同的放大器，因为大多数人都会觉得朝一个方向转比另一个方向要容易得多。

然后，我们就要定义一个让角色开始移动的最小值了。当你察觉到玩家的脚的移动超过了这个阈值，角色就要开始做相关的移动了，比如前进或者后退。玩家的脚移动得越远，角色就走得越快。下面的“实现”部分将对这个细节进行讲述。

实现

初始化 ARToolKit 是需要一些功夫的，但是不难做到。我们需要设置一下摄像头，然后把需要检测的图片加载进去。这里还需要对一个 XML 文件（这里没有给出）进行一些配置，例如像素格式等。你也可以把这些作为起始的选项，让玩家自己去配置。下面的代码就是 ARToolKit 例子中的。

```
#include <AR/config.h>
#include <AR/video.h>
#include <AR/param.h>
#include <AR/ar.h>
#include <AR/gsub_lite.h>

ARGL_CONTEXT_SETTINGS_REF argl_settings = NULL;
int patt_id;

void setup()
{
    const char *cparam_name = "Data/camera_para.dat";
    const char *patt_name = "Data/patt.hiro";
```

```
    char vconf[] = "Data/WDM_camera.xml";

    setup_camera(cparam_name, vconf, &artcparam);

    // 为当前上下文建立 argl 库
    // 不要忘记捕获这些异常:-)
    if((argl_settings = arglSetupForCurrentContext()) == NULL){
        throw runtime_error("Error in arglSetupForCurrentContext().\n");
    }
```

从默认的图案文件 Data/patt.hiro 中读取要识别的图案：

```
    if((patt_id = arLoadPatt(patt_name)) < 0){
        throw runtime_error("Pattern load error!!");
    }
    atexit(quit);
}
```

patt_id 是先前已经被识别的图案。

```
void setup_camera(const char *cparam_name, char *vconf, ARParam *cparam)
{
    ARParam wparam;
    int xsize, ysize;

    // 打开视频路径
    if(arVideoOpen(vconf) < 0){
        throw runtime_error("Unable to open connection to camera.\n");
    }

    // 获得窗口大小
    if(arVideoInqSize(&xsize, &ysize) < 0)
        throw runtime_error("Unable to set up AR camera.");
    fprintf(stdout, "Camera image size (x,y) = (%d,%d)\n", xsize, ysize);

    // 加载摄像头参数，为窗口调整大小并初始化
    if (arParamLoad(cparam_name, 1, &wparam) < 0) {
        throw runtime_error((boost::format(
              "Error loading parameter file %s for camera.\n") % cparam_name).str());
    }
```

下一步，根据当前图片的尺寸改变摄像头的参数，因为摄像头的参数改变是基于图像尺寸的，即便使用的是同一个摄像头。

```
    arParamChangeSize(&wparam, xsize, ysize, cparam);
```

摄像头的参数设置好后会在屏幕上打印出来：

```
    fprintf(stdout, "*** Camera Parameter ***\n");
    arParamDisp(cparam);
    arInitCparam(cparam);
    if(arVideoCapStart() != 0){
        throw runtime_error("Unable to begin camera data capture.\n");
    }
}
```

在函数 setup 中使用的函数 quit 释放之前由 ARToolKit 申请的资源。

```
void quit()
{
    arglCleanup(argl_settings);
    arVideoCapStop();
    arVideoClose();
}
```

下面让我们看一些示例代码，这些代码介绍了如何使用 ARToolKit 来得到标记的位置。首先通过下面的方式使用 arDetectMarker 来检测标志：

```
ARMarkerInfo *marker_info;
int marker_num; // 计数检测到的标记数量
arDetectMarker(image, thresh, &marker_info, &marker_num);
```

通过使用库的功能进而找到的标记是以数组的形式返回的。为什么要用数组呢？因为如果一次要检测多个标志，数组就会很有帮助，它可以保存多个标志。用户可以通过比较 marker_info[i].id 和 arLoadPatt 返回的值来找到需要的标识。在下面的代码中，你可以看到如何得到 marker_info[i]标志的变换矩阵。

```
double patt_centre[2] = {0.0, 0.0};
double patt_width = 80.0;
double patt_trans[3][4];
double m[16];
arGetTransMat(&marker_info[i], patt_centre, patt_width, patt_trans);
arglCameraViewRH(patt_trans, m, 1.0);
```

就像你看到的那样，你需要调用 arGetTransMat 和 arglCameraViewRH 这两个函数。前一个函数获取 ARToolKit 需要用到的变换矩阵，后一个函数把它转换成 OpenGL 可以用的格式，这样你就可以使用它去变换场景对象了（这里还没有用到），或者从一种更熟悉的格式来观察变换。

然后，你可以从矩阵 m 中获取围绕 y 轴的平移和旋转。

```
// m[12], m[13], m[14] == x, y, z
if(m[12] > start_position_x + mov_eps){
    walk_fwd(m[12] * mov_mult);
}else if(m[12] < start_position_x - mov_eps){
    walk_bck(m[12] * mov_mult);
}
double angle_y = asin(mat[8]);
if(angle_y > rot_eps){
    turn_left(angle_y);
}else if(angle_y < -rot_eps){
    turn_right(angle_y);
}
```

就像我们解释的那样，前一段代码可以用在当摄像头放在了玩家右面的时候，这样监视的是玩家的右脚。如果想跟踪左脚，只需要翻转符号即可。

你还需要一些放大器和精度值来调整控制敏感性。我们对 move_eps 的建议值是 10，rot_eps

的起始值是 0.3（用户可以自己调整）。虽然我们给 mov_mult 的值是 0.0625，但是这要根据用户在虚拟世界中的比例来定。start_position_x 这个变量必须用需要的值来初始化；也就是说处于这个位置时，保证角色不会移动。最简单的方法就是把 ARToolKit 检测到的第一个位置 *m*[12] 赋给它。

矩阵 *m* 中还有一些其他有用的数据，包括 *m*[13]和 *m*[14]，因为它们包含了另外两个轴向的平移信息。例如，*m*[13]可以用来做跳跃的数据采集，*m*[14]是侧向平移。但是，你会发觉同时控制平移和旋转几乎是不可能的，所以需要在自己的游戏中明智地选择控制方式。其他的旋转轴在这里都没有什么意义，所以就不讨论了。

1.7.3　一个简单的游戏

为了验证使用脚步导航的 FPS 的可操作性和可玩性，我们这里实现了一个简单的 FPS 游戏，游戏中有一个简单的地图，玩家可以用我们的导航技术来探索这个区域。

第一次接触游戏时，玩家可以在训练区域活动，也就是在地图的第一个区域中（见图 1.7.3）调整输入的协调性以及建立自信。当玩家穿越了蓝绿色的条以后（见图 1.7.4），游戏才正式开始。在剩下的区域中，我们设置了几个障碍物和一些红色的检查点，当玩家正确通过以后，这些地板上的检查点会变成绿色（见图 1.7.5）。与障碍物（包括墙）的碰撞会被检测出来并且以明显的方式反馈给游戏玩家（屏幕将会变色，见图 1.7.6）。当玩家通过所有的检查点并且到达出口（图 1.7.3 中的“结束点”）时，游戏就结束了。游戏的目的就是让玩家在有限的时间内通过所有的检查点，同时碰到障碍物和墙的次数尽可能地少。

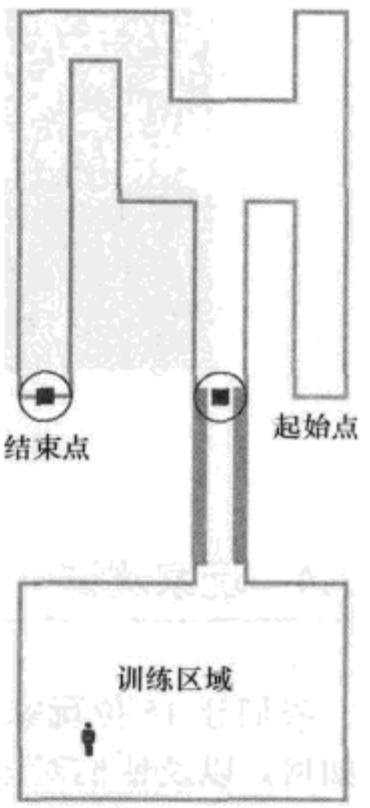


图 1.7.3　游戏的路线

所有的游戏事件都会被记录在一个文本文件中，所以用户可以发现玩家是否在穿越障碍物或寻找一个检查点时遇到了问题。

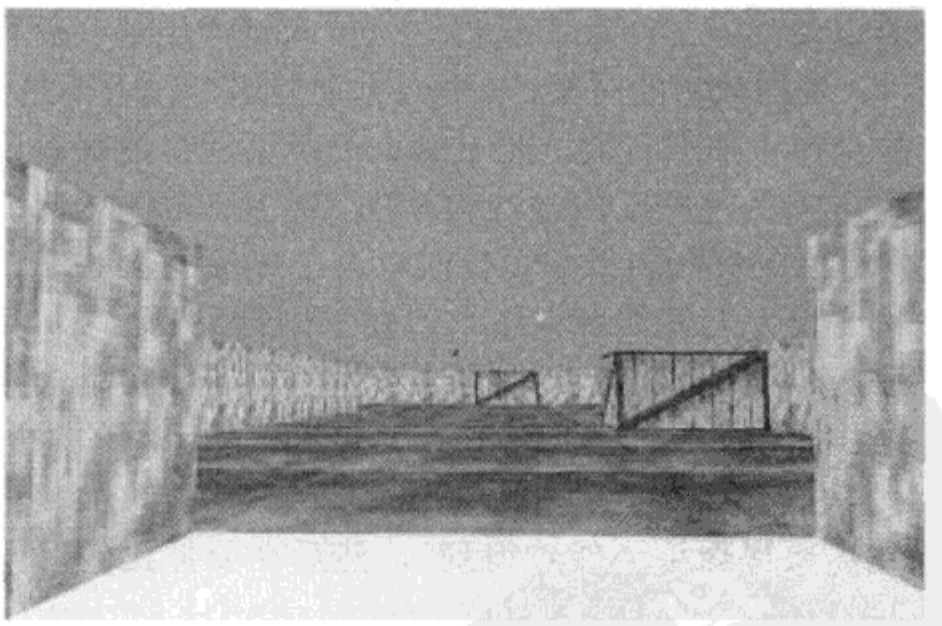


图 1.7.4　游戏开始，通过前景发亮的条指出

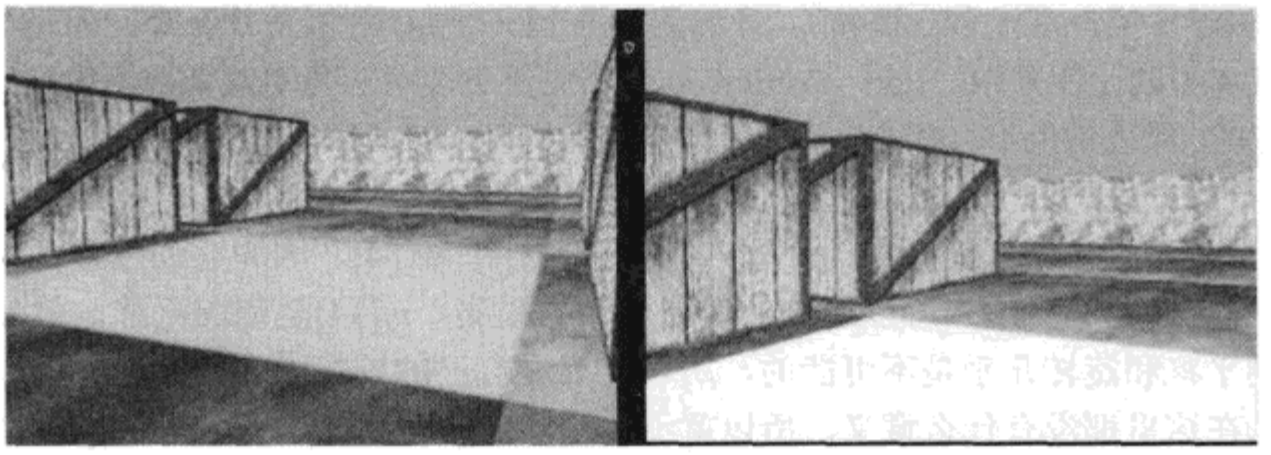


图 1.7.5 在场景中的一个待访问的检查点（左图）和访问该检查点之后的视图（右图）

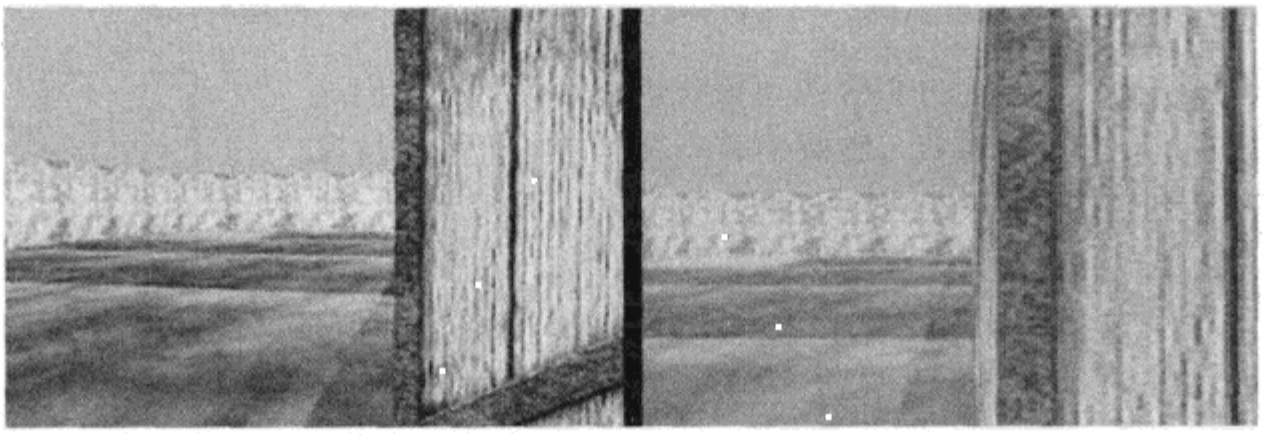


图 1.7.6 游戏中的两帧与障碍物碰撞之前（左图）以及碰撞之后（右图）

1.7.4 玩家测试

我们让 15 位玩家使用这种导航技术试玩了这个简单的游戏，希望看到他们在玩时的表现如何，以及他们对该技术的建议。我们提了几个问题，包括玩这个游戏时感觉是否舒服，玩起来是否简单，学起来是否简单，玩家是否觉得这个设备很有效率。其中的 6 个玩家觉得脚步导航技术很舒服，另外有 4 个玩家觉得非常舒服，4 个玩家觉得不是很舒服，只有一个玩家觉得这个技术难以使用。三个玩家觉得这个技术挺难用的（就像我们说的那样“在开始的时候比较难用”）。说到效率的时候，3 个玩家觉得这个技术不是很有效，另外 7 个玩家有的觉得挺好，有的觉得不那么有效，剩下的 5 个玩家觉得很有效。

这些数据告诉我们玩家的这些反应是很自然的，而且在经过一些训练以后，他们自然会变得很轻松。在我们提供的样板游戏中，我们记录了碰撞的次数和玩家穿过所有检查点并完成游戏所需的时间。因为大多数玩家还是习惯于用键盘和鼠标，所以自然而然地，我们会认为他们在用脚来控制时会比键盘和鼠标慢许多。然而，实际上这些玩家在游戏中并没有遇到多大的问题，而是轻而易举地做到了避免和那些障碍物的碰撞。同时我们注意到，在移动中的突然停止比以前要容易多了，这是因为加速的控制变得更自然了（只要把脚移动到最远的距离，当然需要摄像头还能看得到），然后在要停止的时候把脚放回原位就可以了。

1.7.5　结论

本精粹描述了一种新的技术，这种新技术允许玩家用他们的一只脚来控制 FPS 游戏中角色的行进。在精粹中也介绍了一种很便宜的计算机视觉实现方法，主要是使用 ARToolKit 开源库来简单地跟踪脚的移动，然后把这些移动转换到游戏环境的交互控制中。

在纯计算机视觉系统中有一些限制。很自然地我们就可以想到，如果那些标志点被移动到摄像头的视野范围之外，那么计算机就不能进行跟踪了。这可能会限制到交互的活动，也就是说如果玩家遮住了部分标识图案，导航系统就会停止。其他因素如范围的问题、图案的复杂度、标志相对于摄像头的朝向以及光照环境对图案的辨别影响等，都会或多或少地影响到导航系统的工作。

而从另外一个方面来说，这种技术相对于传统技术有 3 项优势。首先，通过使用跟踪系统，玩家有更多的自由空间可以发挥。玩家可以在一维、二维和三维空间移动他们的脚，或者绕着某个轴转动他们的脚（左右转）。而且，鼠标和键盘可以另进行它用，因为我们已经不需要它们来控制移动了。比如，鼠标和键盘可以用来对屏幕上的物体做瞄准、射击、选择和操作。最后，使用整个身体来玩游戏给玩家带来更多的投入感，就好比任天堂 WII 带给大家的感觉一样。

1.7.6　以后的工作

在这个交互技术上，我们还有许多路要走。第一就是要增加一些新的指令，例如跳跃和侧移，另外就是对鼠标和键盘下新的定义，因为它们不必再做任何移动上的事情了。

多人游戏也是一个还未被触及的领域，因为基于标志的交互技术和禽类群聚技术都具有多人的功能。为每一个玩家增加一个不同的标志来进行标识并不是一件难事，只要你使用的摄像头可以看清楚它们。如果不行，我们就需要用两个或更多的摄像头来捕捉了。

为了解决设备的便携性，也为了避免摄像头和标识之间的遮挡问题，这里也可以用到一些交互的无线控制设备。不过我们还是要坚持主要的想法：把控制器绑在玩家的脚上来实现角色的移动。最后，我们希望类似“超级猴子球”这样的游戏会因为控制器的不同而变得更好玩。

1.7.7　致谢

作者衷心地感谢 Fábio Dapper 和 Alexandre Azevedo，他们在概念和技术的实现方面做了许多的工作，同时也感谢参加游戏测试的人们，他们给予了许多有价值的反馈，最后还要感谢巴西国家科技委员会（CNPq）在财政上的支持。

1.8 推迟函数调用的唤醒系统

Mark Jawad, Nintendo of America Inc.
mark.jawad@gmail.com

当前，大多数计算系统都围绕着多处理器来设计。事实上，在工程领域中，这样的设计已经成为一个核心设计方式，即便是现在的游戏机也无一例外，现在卖的游戏机都是有着多核心 CPU 的。此外，它们中的大多数还依靠辅助处理加速芯片，例如可编程的 IO 控制器、DMA 引擎、数学协处理器等。这些芯片都是和 CPU 并行运行的，并且作为系统中的一部分存在，所以如果我们是美术的，就可以把技术层次推向一个更高的高度。上面说的这些模块可以相对于其他模块独立工作，但是当工作完成以后，都会发送消息给游戏（通常是中断形式），通过这种方法，游戏可以调整其他模块的工作进度。

游戏管理这些消息的方法可能让人很轻松并且没有 BUG 存在，也可能使你进入一个令人头疼的复杂世界，BUG 也会到处都是。本文将讨论异步事件的联系以及其他与时间相关的问题，还提供了一个系统来很好地处理它们。

1.8.1 时间问题

从游戏的角度来看，这些消息在任何时刻都可能发生，这样我们就应该把它们作为异步事件来看待。这些事件消息提醒的好处就是游戏可以顺着主处理器的步伐进行运作，其他处理器可以根据它们的情况来做一些辅助的工作，但是这样做也会有问题出现。如果消息不能被合理地处理，就会造成与时间相关的错误或者使系统不稳定。与时间相关的问题一直都是很难跟踪出来的，这种问题会发生在你试着去访问另一个模块还在使用的内存时，或者是你在没有使用同步原语的情况下改变游戏状态时。使系统不稳定的 BUG 会更糟糕，这种情况会发生在回调或中断句柄在一个中断期里运行了太长时间时，因为这样会造成其他中断信号（或者是随后的相同类型的中断信号）的丢失。在这样的情况下，你会看到各种各样奇怪的程序行为出现。

工作在掌机游戏系统上的开发者更可怜，因为他们面对的不仅仅是这些问题，还有一些其他的。在这些系统中，我们需要关注的一个共同概念是：垂直空白期（vertical blanking period）。这个垂直空白期就是图形引擎在显示设备准备下一帧时悬挂起的时间段。只有在这个空白期内，开发者

才被允许去访问图形系统的内存和寄存器。在这段时间里面，你必须快速地做出决定，比如什么数据需要被载入，显示芯片上的什么设置需要被修改等。开发人员必须把修改的内容和数据尽快上传到图形系统里面去。如果不能在这个时间窗口内完成上传，那么图形就会被毁坏或者有其他明显的瑕疵。

问题都以某种形式出现在时间的把控上（时间总是游戏开发者的敌人，我们却从来无法完全控制它）。这是因为我们从来没有得到足够的时间来完成这个工作，所以我们必须聪明地运用我们所得到的时间。一种方法就是你把一大块事情分成好几个小块来做。也就是说，你可以现在就决定要做什么，但是把实际的工作延迟到以后再完成。因为大多数工作是由函数调用来完成的，所以本书就会介绍一个系统，它把函数的调用和相应的参数排成队列，然后在某个时候唤醒它们——即为一个延迟的函数调用系统。

1.8.2　案例分析

让我们以垂直空白期的使用作为一个例子，因为你在那段时间里面拥有的时间是很有限的。其实最理想的情况就是在这段时间里面不要做任何消耗时间的逻辑；也就是说在这段时间里面，你要做的事情就是尽可能快地上传新的数据。那为什么不把那些和上传数据相关的函数逻辑在前一帧做掉呢？你可以提前把资源和目标地址整理出来，把传输的大小或者可能的话把数据的类型也标记上（贴图、调色板等），然后等到了垂直空白期，你就可以把整理好的东西一股脑儿地都传上去了。

但是如果数据的类型决定了使用何种函数来上传，那我们应该怎么办？其实，你可以用 switch 语句来完成，或者跳转表也是可以的，再或者用一连串的 if 语句来确定调用哪个函数也没问题。在 PC 或者家用主机上，使用这个方法是理所当然的。然而，在时钟速度较慢的便携式机器上，就毫无疑问地需要再三考虑了，尤其是考虑到如此短的垂直空白期的每个时间片实在是太宝贵了。因此，最理想的方法就是在你配置地址和传输尺寸的时候就预先确定调用什么函数。然后在垂直空白期，你需要做的全部事情就是：加载函数的地址和必要的参数，并且跳转到所加载的函数地址中去。综上所述，你要在游戏循环里面早点把函数调用配置好，然后在垂直空白期到来的时候就延迟地调用它们。

同样的策略可以用在家用主机上来应付异步消息通知。这些通知可能是你得到一个回调，这个回调告诉你读取某个文件的操作已经完成，或者内存卡已经被插入，或者是玩家插入或者拔出游戏手柄。这些情况中，有些情况会复杂一些，需要更多的操作在里面，但是它们中的大多数并不需要你立即做出什么反应。有些读者会说，游戏逻辑需要知道文件是不是读完了，或者是不是有新的手柄插入了，但是为什么不把这些事件放到当前被处理的帧的最后面去处理呢？

把传进来的通知消息参数（注意保存那些重要的容易丢失的临时数据）排列好，稍后再处理它们。通过这种方法，你可以尽可能快地结束回调/中断，这样会对你很有帮助。通过把消息处理悬挂起来并且在游戏循环中的指定位置恢复执行的方法，你可以很好地保持游戏行为的确定性。这样做可以基本消除和时间相关的问题。还有一个好处就是你可以保证：在团队里面不会有人错误地运行一个“过长”的进程，而导致程序不能响应中断。这是因为现在的处理过程处于游戏循环的一个已知位置，而不在中断的处理程序中。那么，系统就可以在

任何需要的时候响应中断了。

1.8.3 对函数调用分类

大多数函数都直接获取参数中的值，例如下面这个 C 标准库里的函数：

```
void *memccpy(void *dest, const void *src, int c, size_t count);
```

这个函数有 4 个参数。这样的函数在从 C 继承下来的函数中来看再常见不过了，你可以把这样的函数调用归类为“直接”获取参数类型。其他的函数，例如 Windows 开发包中的这个：

```
ATOM RegisterClassEx(CONST WNDCLASSEX *lpwcx);
```

它只有一个参数，但是这个参数是指向一个控制结构的，在这个结构中存储的是这个函数真正想得到的 12 个参数。我们可以将这样的函数归类成“间接”获取参数类型。通常来说，这些间接类型的参数都存储在被调用函数的栈区中（而不是存储在堆区），因此它们在被调用函数出栈后就丢失了。

从表面来看，第二种类型只是第一种类型的一个分支而已。然而，你需要注意到一个很重要的不同点，有时候你需要临时保存一些数据，在延迟调用函数中，你在哪里保存间接的参数数据块？栈里的被调用函数在出栈之后，它的那些参数早就不知道去了哪里（数据出栈以后就没有了）。解决方法就是把参数保存在一个足够大的内存池里。这个内存池是在延迟系统中的，参数也不会去栈那里，而是在内存池中直接被构建好的。

1.8.4 检视这个系统

系统的头文件 deferred_proc.h 很小，可以很容易地被整合到你的游戏中。头文件包含了一个初始化本系统实例的函数，以及用来将延迟调用加入到直接获取参数（DA）或者间接获取参数（IA）的函数列表中的一对函数和宏定义，或者另外一个负责延迟函数的执行（完成调用以后重置列表）。

系统的主要部分是用 C 完成的，虽然其中有个函数必须用汇编语言来完成。那个函数就是延迟函数的调用器，并且根据使用的游戏机不同而不同，还必须考虑到系统运行平台的应用程序二进制接口（ABI）。所以为了移植这个系统，用户只需要重写这个延迟函数的调用器就可以了。

注意，这里介绍的系统对函数的调用是有限制的，那些函数用的参数都是要保存在通用寄存器中的。也就是说，浮点数指针的值、内部浮点数的向量或者其他任何不能保存在通用寄存器中的数据都是不允许作为延迟函数的参数出现的。这样做的目的就是保持这个系统的简单性。当然，如果你要支持那些参数，那也是可以做到的。同样是为了保持系统的简单性，我们只允许最多有 4 个参数。

我发现 4 个参数可以满足大多数的情况。不过考虑到有时候会超过 4 个参数，对于不同的目标系统，你需要用寄存器来传递一些参数，而另外的参数要用到栈来传递，这又会增加

复杂性，然而复杂性是这个系统希望避免的问题。

由于这个系统只允许最多 4 个参数，你必须使用间接获取参数的方法来处理超过 4 个参数的函数（别忘了，this 参数是 C++实例函数中的一个隐藏参数）。

延迟函数调用器 dfpProcessAndClear 是用汇编写的，所以可能会比较难懂一点，但是它的意思很简单。它要做的事情就是循环访问函数列表的内容，并调用每个入口函数。在每个循环迭代里，它至少会加载用来描述以下各方面的一个控制字：

- 函数的类型（DA，直接访问参数类型，或者 IA，间接访问参数类型）；
- 要载入的通用寄存器的数量；
- 函数调用从列表中获取的任意的额外字节数目的数据。


ON THE CD

当然，这个函数还需要载入目标函数的地址，然后它就会把所有这个函数用得到的参数也载入进去。一旦所有的信息都载入好了，它就会让这个函数自己去执行。等函数执行完毕并返回，这个函数就会进入下一个循环的迭代过程。一旦完成了所有的迭代，你就需要重置这个函数调用列表，然后退出。注意，这些代码很简单，没有考虑到多线程的情况。参见光盘中完整的源代码。

1.8.5　结论

对于所有平台的游戏开发者而言，延迟函数是一种非常有用的工具，对于家用机和手掌机的开发者们来说更是如此。这个系统的存在对于绝大多数与时间相关的比较棘手的问题都很有帮助，例如垂直空白期、中断处理序列以及跟踪中断和时间的系统。这个点子是很灵活、很容易扩展、有效率的，并且是可移植的（这些都是为开发人员所欣赏的特性）。

1.8.6　参考文献

[Earnshaw07] Earnshaw, Richard. “Procedure Call Standard for the ARM Architecture.”

1.9 多线程任务和依赖系统

Julien Hamaide
Julien.hamaide@gmail.com

本精粹着力于让程序员轻而易举地运用到次世代技术中多核的功能，而在建立任务时不需要了解复杂的多线程概念。通过本文提供的一个简单的系统来自动管理任务之间的依赖性之后，你基本上不需要其他任何同步管理机制来帮你完成任务了。这个系统适用于中小型任务，例如动画融合和粒子系统的更新等。

1.9.1 介绍

当考虑多线程时，脑海里会立即浮现出同步的问题。死锁对于每个程序员来说都是噩梦。甚至对于一些小项目来说，我们每次都要依据项目和模块的独特需求和内部的依赖将多线程解决方案重写一遍。因此，实现这些系统的程序员需要具备同步原语的知识及其内在的复杂性。例如，如果系统对时间的要求很苛刻，我们就必须采用一个无锁的算法。另外，系统越复杂，产生 BUG 的几率就越大。

这个系统的主要目的就是提供一个跨平台的框架，这个框架可以把多线程问题的复杂性隐藏起来。这样，任何程序员都可以使用这个系统，即便是他们对于基本的多线程概念都不是很了解。在这个系统中是没有线程概念的，线程的概念被定义为工作单元的任务所取代了。

建立任务的过程通过下面的代码给出。跨平台的兼容问题通过将临界区和线程封装到类中的方式来解决。然后，框架的代码就由这些基础类建立起来。光盘中的演示程序包含了任务的管理类和包装类。

```
void MultithreadEntryFunction( void * context )
{
    // 在这里完成任务
}
PARALLEL_JOB_HANDLE handle;
handle = PARALLEL_JOB_MANAGER_CreateJob(
    &MultithreadEntryFunction, context);
PARALLEL_JOB_MANAGER_ScheduleJob( handle );
```

通过阻止小型任务的创建和删除，这个系统的性能也得到了优化。这个

框架建立了一个线程池，线程池中的线程就会去执行系统中的任务。所有线程都是在程序启动时就被建立好了的。线程的数量依赖于不同的平台，如果有必要，它还会被动态地调整（在个人计算机上，在编译时可能还不知道目标平台的 CPU 的核到底有多少个）。这种方法和 OpenMP 在多线程上的分布式工作方法是类似的[OpenMP]。

这个系统也支持给任务划分优先等级。如果一些任务需要更多的执行时间或者有许多依赖的任务，那么就可以提高它们的优先等级，以保证它们会被更早地执行。对于空闲任务（idle tasks）的支持被安排在今后的工作中，所以目前还没有实现。

任务的同步是由一个依赖系统所管理的。这个系统允许同步点和任务依赖的建立。下面就来看看这个依赖系统。

1.9.2　任务系统

任务系统主要由 4 种对象组成。

- 任务（job），由工作单元组成，在回调函数中被限定。任务必须是线程安全的。
- 管理器（manager），通过任务的类型和优先级来维护任务列表。
- 进度管理器（scheduler），根据任务的优先级和类型来选择最合适的任务。
- 工人（workers），执行分配到的任务。

任务

下面代码中的类描述了任务。这其实就是一个很简单的封装（函数的参数都是预定义好的）。这个结构也包含了任务的优先级和句柄。PARALLEL_JOB_PRIORITY 是一个枚举，包含了标准的优先级值（高、低等）。句柄则在系统中标识任务，然后由系统外部的对象来引用任务。Type 域稍候会被介绍到。

```
class PARALLEL_JOB
{
public:
    void Execute()
    {
        Function( Context );
    }
private:
    PARALLEL_JOB_HANDLE JobHandle;
    PARALLEL_JOB_PRIORITY Priority;
    PARALLEL_JOB_TYPE Type;
    PARALLEL_JOB_FUNCTION Function;
    void * Context;
};
```

管理器

管理器的类是个单件，提供了整个系统的公共接口。作为系统的中心部分，它以一种多线程安全的方式维护了任务列表。这个管理器还对工人和进度管理器线程的创建以及初始化负

责。管理器的接口显示在下面的代码中。建立任务和调度任务是分开的，这样用户就能够把依赖性加到建立好的任务上。在示例代码中，每个调用都有临界区保护[MSDN1]。虽然说无锁算法是一个更好的选择，但是这样会增加系统的复杂度。为了简化概念的理解，这里只给出了临界区这个方法。如果大家有兴趣，可以尝试着去实现无锁算法。

```
typedef void (*PARALLEL_JOB_FUNCTION )( void* );

PARALLEL_JOB_HANDLE CreateJob(
        PARALLEL_JOB_FUNCTION function,
        void * context,
        PARALLEL_JOB_PRIORITY priority = PARALLEL_JOB_PRIORITY_Default
        );

PARALLEL_JOB_HANDLE CreateAndScheduleJob(
        PARALLEL_JOB_FUNCTION function,
        void * context,
        PARALLEL_JOB_PRIORITY priority = PARALLEL_JOB_PRIORITY_Default
        );

void ScheduleJob(
        PARALLEL_JOB_HANDLE job_handle
        );
```

PARALLEL_JOB_HANDLE 是一个结构，它包含了唯一的识别符和依赖索引。依赖索引在下一小节介绍。

进度管理器

进度管理器是一个线程对象，它专门等待工人线程完成工作。一旦有工人线程空出来了，它就将下一个任务赋给这个线程，然后让自己继续在旁边休息。下面给出了进度管理器主循环的伪代码。

PARALLEL_JOB_MANAGER_GetNextJob 返回的是下一个最优的任务。如何来决定这个是最优的任务呢？我们会在后面给大家介绍。WaitForJobEvent 允许进度管理器在没有新任务和一些工人线程是空闲的情况下去休息。不过一旦有新的任务出现，WaitForJobEvent 就会产生信号，否则就被重置[MSDN2]。

```
while ( !ThreadMustStop )
{
    thread_index = PARALLEL_WaitMultipleObjects( worker_thread_table );
    if( PARALLEL_JOB_MANAGER_GetNextJob( next_job, thread_index ) )
    {
        worker_thread_table[ thread_index ]->SetAssignedJob(next_job);
        worker_thread_table[ thread_index ]->WakeUp();
    }
    else
    {
        WaitForJobEvent.Wait();
    }
}
```

工人线程

工人就是真正干活的。当操作系统允许时，工人线程就会被分配到一个处理器上。下面给大家看一下伪代码。工人线程等待着 DataIsReadyEvent 的信号，这个信号就代表它已经被分配给一个新的任务了，然后分配到了任务就要去执行。当任务完成以后，工人就会通知依赖性管理器任务完成了。最后，它会发信号给进度管理器，告诉它自己在等待任务。

在实现的时候，工人线程的数量被设置为可用的处理器数量。如果主线程总是很忙，可以将工人线程的数量设置为可用的处理器数量减一。

```
while ( !ThreadMustStop )
{
    PARALLEL_WaitObject( DataIsReadyEvent, INFINITE );
    AssignedJob.Execute();
    PARALLEL_DEPENDENCY_MANAGER_SetJobIsFinished( AssignedJob );
    WaitingForDataEvent.Signal();
}
```

高速缓存一致性

为了保证代码和数据的高速缓存一致性，所有的任务和工人线程都有自己的类型。任务的类型可以是任何用户想要的类型，这也要看用户的系统如何。在这个实现中，我们选择的类型如粒子系统、动画、寻路等。类型会在任务选择时派上用场，所以在选择类型时需要注意到这一点。高速缓存一致性是特定于每个系统的，这里需要做一些微调，才能获得最大的速度。

任务选择

当进度管理器要求执行下一个任务时，管理器会采用一个简单的法则来选择任务。下面的伪代码介绍了这个法则。选择主要是在线程类型的改变和高优先等级的任务之间保持一个平衡。这个算法是挺难适用在任何情况下的，但是最高优先级的线程总能够比低优先级的线程优先执行这点是不变的，即便在类型选择时选择的是低优先级的任务。在这种情况下，你可以允许进度管理器让两三个低优先级的线程在最高优先级的线程被计划执行以前执行。

```
current_type = type of worker thread
if( job of type current_type exists )
{
    new_job = job of type current_type with highest priority
    if( priority of new_job - highest priority available > 2 )
    {
        new_job = job with highest priority
        worker thread type = new job type
    }
}
else
{
    new_job = job with highest priority
```

```
        worker thread type = new job type
    }
```

1.9.3 依赖性管理器

依赖性管理器是一个基于对象的系统，可以保证任务的同步性。在当前的实现中有两种类型的节点，分别是任务和组（group）。这个系统在节点之间建立了一个依赖关系图。组允许用户建立同步点，例如渲染的起点。图 1.9.1 显示了一个由系统建立的典型的依赖关系图。

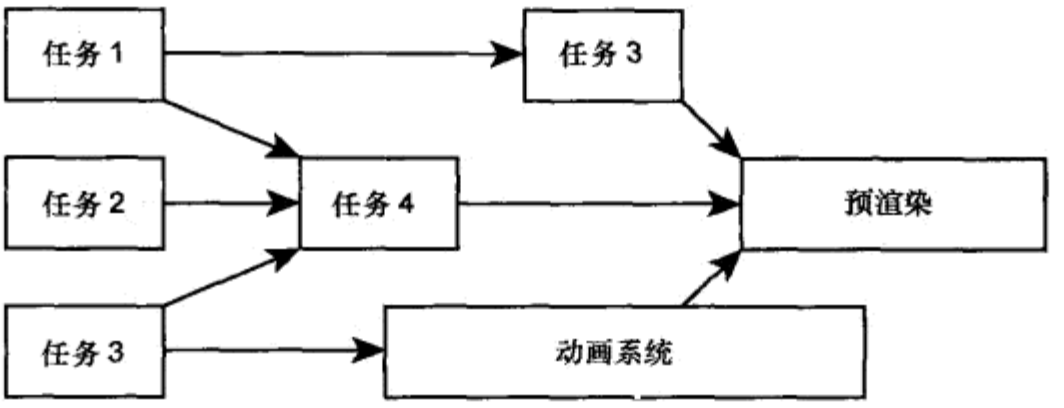


图 1.9.1 依赖关系图例子

依赖关系图

依赖关系图被保存在依赖节点中。每个依赖节点都保存了一个依赖对象的列表。依赖性有两种状态：吻合和阻塞。如果有一个依赖节点是吻合的，那就说明任何其他依赖于它的依赖节点都可以被执行了。即便这个节点上只有一个相关联的节点是阻塞的，这个节点也会是阻塞的。

为了避免依赖性出错，每个节点都会包含一个依赖计数。在节点上每增加一个阻塞依赖，计数就会加一。当计数为零时，依赖就被吻合了，否则就是阻塞的。当依赖进入了吻合状态，它就会迭代它所有的依赖对象来降低它们的计数。同样的，当依赖变成阻塞时，它也会迭代所有的对象来增加它们的计数。吻合和阻塞信息是在依赖关系图中传播（propagation）的。图 1.9.2 和图 1.9.3 显示了传播的步骤。当任务 1 变成吻合的时候，动画组也就成了吻合状态。任务 3 的依赖计数就减一，与此同时，预渲染组的依赖计数就减二。

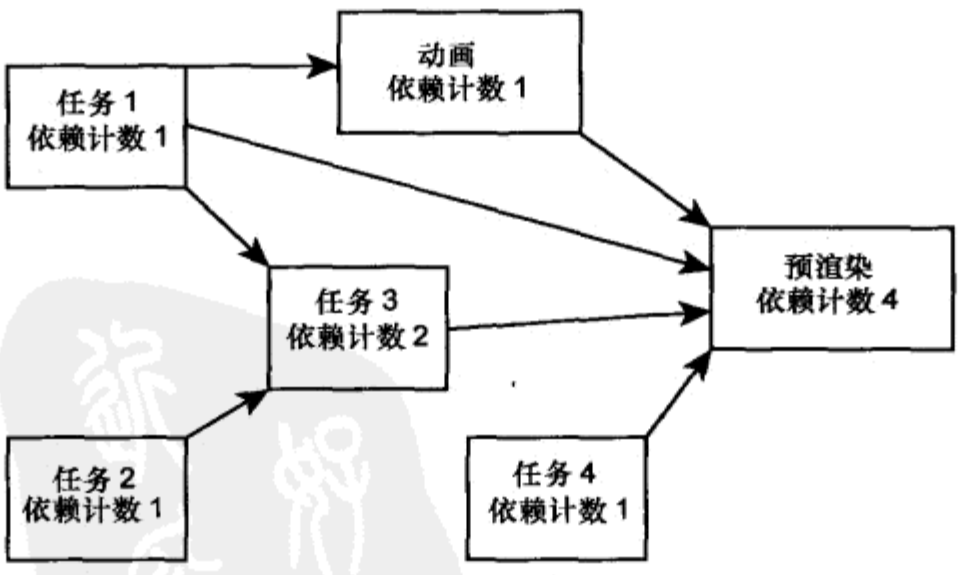


图 1.9.2 依赖计数的初始化

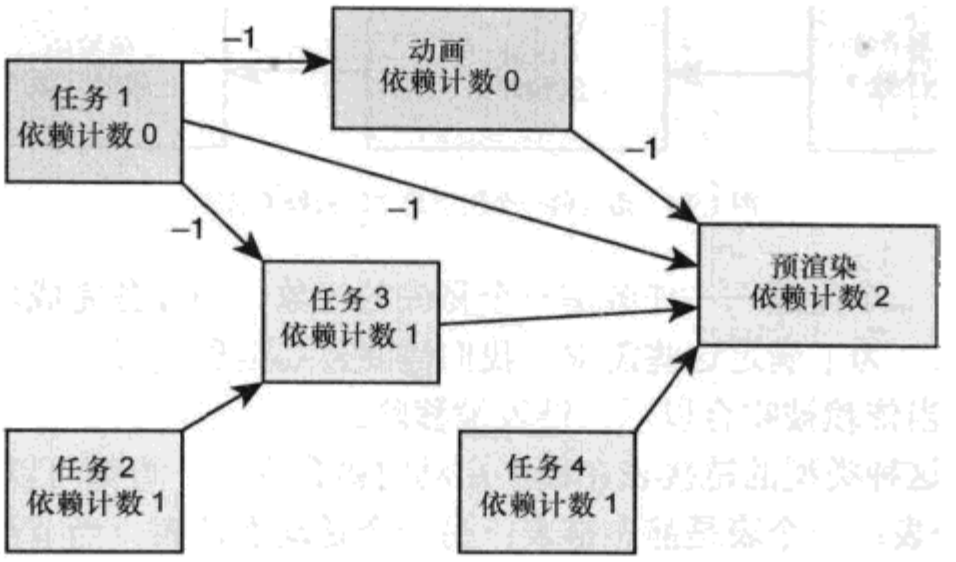


图 1.9.3　信息被传播之后的图

依赖存储

依赖表是一个指向节点的稀疏指针向量。这里提供了一个索引的自动分配机（index dispenser）来把一个空闲的位置分配给目前的依赖。这个表会根据需要增长，但是从不缩短。当一个节点被建立的时候，自动分配机就要分配一个索引给它。当节点被删除的时候，自动分配机就会收回索引以备下次使用。这种循环使用索引的机制表明，依赖不能只靠它的索引来识别。PARALLEL_DEPENDENCY_IDENTIFIER 就是用来协助解决这个问题的。该结构包括了依赖节点表的索引和一个唯一（unique）的索引。这个标识作为一个句柄供用户使用。

下面的代码显示了建立依赖组和链路的方法。在这个例子中，预渲染组会一直等到动画组吻合完成，这样就能保证在渲染以前所有的动画都被计算好了。所有的动画任务都会在动画组之上建立一个依赖。图 1.9.4 显示了由这样的创建方式建立的图。如果一个依赖不存在了（也就是说任务完成了），它就被认为是吻合了。

```
PARALLEL_DEPENDENCY_IDENTIFIER
       animation_identifier, prerender_identifier;

prerender_identifier
   = PARALLEL_DEPENDENCY_MANAGER_CreateEntry( "PreRender Synchronization");

animation_identifier
   = PARALLEL_DEPENDENCY_MANAGER_CreateEntry( "Animation Synchronization");

PARALLEL_DEPENDENCY_MANAGER_AddDependency(
       animation_identifier, prerender_identifier );

// 在游戏过程中
blend_job_handle
   = PARALLEL_JOB_MANAGER_CreateJob( &MultithreadBlendAnim, context);

PARALLEL_DEPENDENCY_MANAGER_AddDependency(
       blend_job_handle, animation_identifier );

PARALLEL_JOB_MANAGER_ScheduleJob( handle );
```

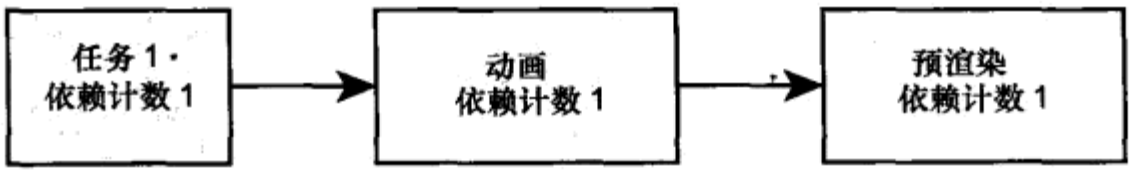


图 1.9.4 通过前一个例子建立的依赖关系图

这个例子列出了一个问题——任务是一个易失的对象，当任务完成后就被销毁了；而组都是一些稳定的节点。为了满足这些需求，我们需要建立两种链表。

- 动态链表：当依赖被吻合以后，链表被移除。
- 静态链表：这种类型的链接表在程序启动时被建立，一直都可以被使用。

节点包含了两个表：一个表是静态链表；另一个是动态链表。当节点吻合的时候，动态链表就被清空了。在前一个例子中，动画同步节点和预渲染同步节点间的链表是被作为一个静态链表而存在的。演示程序中使用了这个例子。

组节点

一个组节点会建立一个同步点，管理器会提供一个方法来等待依赖被吻合。对于 PARALLEL_DEPENDENCY_MANAGER_WaitForDependency 的调用会等到依赖吻合后才开始。这样做是有好处的，比如，我们必须等待所有的计算工作都完成后才可以开始渲染。这种情况是通过事件（events）来实现的，对每个组的实例，我们都会建立一个事件，这就意味着即便用户不想在代码中等待依赖吻合，每个组仍然会有一个事件来对应。如果不想就这样浪费事件，可以建立两种类型的组（需要等待的和不需要等待的）。

任务节点

任务节点有两个用处：

- 通过任务来同步其他的节点；
- 同步任务的启动。

节点建立的时候，它的依赖计数被设置成 1，这就描述了执行这个节点任务的依赖性。为了避免节点的两个需求同时出现，一个是任务开始，一个是任务结束，节点的计数是由任务管理器来统计的。如果节点的依赖计数是 1，这样就不存在依赖关系，而只剩下任务执行自身了，那么这个任务就可以被执行了。在工人线程的伪代码中已经显示了，当任务完成时，工人会向依赖管理器汇报，然后依赖计数就会被设置成零，之后这个节点就变成了吻合的，并把自己的信息传播到所有依赖它的节点中。

1.9.4 后续的工作

下面来讨论后续的工作，也就是这个系统可以提高和扩展的方面。

空闲任务和先占操作（Preemption）

在目前这个系统中，每个已经开始的任务都会占用自己的工人线程，直到完成为止。这

种方法会阻碍用户调度低优先级的任务，而这些低优先级的任务可能需要好几帧才能完成。这里提供了 3 种方法来解决这个问题。

- 先占操作：如果一个新任务被加入到系统中，那么低优先级的任务会被它抢占。并不是所有的操作系统都允许用户对线程上下文切换进行编码的，所以这个方法和平台有关。
- 合作多线程（Cooperative multithreading）：低优先级的任务可以通过合作多线程来执行，但是每个任务应该使用一个特殊的函数来允许系统有机会执行高优先级的任务。这样的系统可以使用像 Windows fibers[MSDN3]这样的技术来实现。
- 低优先级线程：这个系统可以创建新的低优先级线程，当其他工人线程执行的时候，可以让操作系统抢占这些低优先级的线程。这个解决方案是跨平台的，只要操作系统的调度程序足够好，它就能够派上用场。

第一种方法从控制的角度来说是最好的。在任务代码中，用户不需要做额外的工作，只有当任务被抢占时才需要进行控制，但是这个方法是和平台相关的，而且在实现时需要一些技巧。第二种方法相对于第一种方法来说实现起来更简单，但还是平台相关的。第二种方法的另外一个弱点就是任务代码也必须经过修改来满足抢占操作。第三种方法是目前最简单的方法，但用户对线程何时以及如何被抢占没有太多的控制手段。观念上，应该首先实现第三种方法，然后如果还有必要的话，再看看是否转而实现其他两种解决方案。

将同步原语整合到依赖系统中

依赖系统只支持两种节点：任务和组。也就是说，用户不能将这个系统和一个现有的同步机制结合起来。例如，一个载入线程可以使用信号量的方式和其他线程通信。如果你要你的任务等待某些资源载入到内存以后再执行，这个系统就不支持这样的操作了。这里有一个方法可以提供节点的扩展：例如扩展为信号量和事件等。依赖系统在设计时就考虑到了可扩展性。PARALLEL_DEPENDENCY_ENTRY 能够被继承，并方便地支持各种原语。同时，还必须实现用来通知新同步机制下的依赖状态的虚函数。当这些虚函数被实现后，扩展也就做好了，新的系统也就可以和原来的依赖系统进行互动了。

1.9.5　结论

这里介绍的系统为所有的程序员提供了一个强有力的多线程工具，它抽象了多线程的复杂性，把同步的风险降到了最低。有风险的代码都被合并到了一处，同步的问题就在那里被一次性地解决了。通过提供一个依赖图系统（dependency graph system），各个任务就只需要通过建立一个依赖链表而被串联起来。这个系统不仅仅是完成了，而且在设计时就考虑了可扩展性。它的实现还是跨平台的，用户唯一需要移植的就是在演示包中提供的包装类（主线程、临界区和事件）。另外，通过限制线程创建的开销，性能也被提高了。借助于 PARALLEL_DEPENDENCY_ENTRY 接口，这个系统还支持已存在的同步技术。所以，所有的程序员现在都可以通过本精粹来顺利地创建和调度一个任务了。希望大家喜欢！

1.9.6 参考文献

[MSDN1] “Critical Section Objects (Windows).”

[MSDN2] “Event Objects (Windows).”

[MSDN3] “Fibers (Windows).”

[OpenMP] OpenMP Architecture Review Board. “OpenMP: Simple, Portable, Scalable SMP Programming.”

1.10　高级调试技术

Martin Fleisz
Martin.fleisz@kabsi.at

因为现在的玩家对游戏的期望都很高，所以游戏的开发就变得越来越复杂了。增加新特性或者支持新技术往往会增加许多代码量。而正因为这样，导致的问题也就多了起来。游戏项目的进度一般安排都很紧，而且不容易找到的 BUG 经常会造成项目的延期。虽然用户不能阻止代码中的错误，但可以试着改善寻找并修正这些错误的过程。如果一个应用程序出现什么问题，最重要的就是找到这些问题的线索，这样就能找到到底是哪里出了问题，哪个函数引起的。因此，本精粹将介绍一个小型的框架来检测和汇报大量的程序错误，并且这个框架可以很容易地整合到现有的项目中去。

1.10.1　程序崩溃

程序崩溃的原因是多种多样的。堆栈溢出、除以零、访问了一个无效的内存地址，这些仅仅是造成崩溃的一小部分原因。所有的这些错误都是通过异常信号发送给应用程序的，也就是说，如果没有合理地处理它们，游戏就会异常终止。你可以把这些异常分成两种类型：异步异常经常都是在出乎意料的情况下发生的，并且很多时候是由硬件引起的（例如，当你的程序在访问一个无效的内存地址时）；同步异常一般都在你的掌控之下，并且你可以有条理地处理它们。这些异常都是被程序显式地抛出的——也就是说，使用 C++语言中的 throw 关键字。

异常处理

C++语言为处理同步异常提供了内在的支持（通过 try/throw/catch 等关键字）。然而，这不能用来处理异步异常。这些异步异常的处理取决于系统是如何实现它们的。

微软在他们的 Windows 平台上统一了这两种异常的处理，并称它为结构化异常处理（Structured Exception Handling，SEH）。如果你想知道 SEH 以及新的向量化异常处理技术，可查看[Pietrek97]和[Pietrek01]。

用户必须弄清楚的事情是：在操作系统没有找到对应于一个异常的句柄时会发生什么。在这种情况下，操作系统内核会调用函数 Unhandled

ExceptionFilter。[Pietrek97]展示了这个函数如何工作的详细伪代码示例。最令人感兴趣的部分是，这个函数会调用由 SetUnhandledExceptionFilter 这个 API 注册的用户自定义的回调函数。当应用程序崩溃时，操作系统内核就会调用这个回调函数，进而就可以报告错误了。

基于 UNIX 的操作系统使用的是另外一套方法：在发生异步异常的情况下，操作系统会向应用程序发送一个信号，这样就会立即中止正常的执行流程，并调用应用程序的信号处理模块。你可以使用函数 sigaction 来设置自定义的信号处理模块，这样就可以在这个信号处理模块中报告错误了。

除了做一些错误汇报以外，你还可以考虑在异常处理中保存游戏的进度。你可以想想，玩家们最讨厌因为游戏崩溃而失去了自己连续几个小时的游戏进度。当然，这样的工作应该在完成了错误汇报以后就去做。

汇报没有处理的异常

如果在调试进程的过程中产生了一个没有被处理的异常，那么调试过程就会停在产生异常的代码位置。即便没有为应用程序关联调试机制，我们也希望能获得关于崩溃的一些信息。你可以利用内存转储文件（crash dump file）来达到这个目的，这个文件包含了程序崩溃时的一个进程快照。

在 Windows 平台上，你可以使用微软调试工具的 API 来完成这个任务，该功能是由 dbghelp.dll 提供的。通过函数 MiniDumpWriteDump，你就可以创建当前正在运行的线程的一个小型转储文件。不像通常意义上的内存转储文件（类似于 Watson 医生建立的那种转储文件），小型的转储文件不需要保存整个进程空间，因为其中的一些区域在大多数调试情况中都用不到，比如说包含被载入模块的那些区域。你需要知道的就是这些文件的版本信息，后面你会需要把这些信息提供给调试器。这就意味着用这个 API 创建的转储文件会比通常意义的完整的转储文件小得多。

在转储文件中保存的信息可以通过参数 DumpType 来控制。MiniDumpNormal 只包含类似堆栈跟踪和线程列表的基本信息，而 MiniDumpWithFullMemory 会把所有能够访问到的进程内存记录到文件中。如果想要知道哪种类型是最适合你的，建议去看看[Starodumov05]。如果想知道更多关于 MiniDumpWriteDump 这个 API 的使用方法及其参数，可以去看看[MSDNDump07]。因为老版本的 dbghelp.dll 不包含以上接口（比如 Windows 2000 所提供的版本），所以在发布游戏时应该考虑加入这个库的最新版本。还需要记住的就是由 dbghelp.dll 提供的 API 并不是线程安全的，也就是说调用者必须负责所有函数的同步调用。

在 UNIX 平台上，我们必须使用一个第三方工具去显式地建立一个核心转储。Google[Google05]的 coredumper 项目就是一个开源的库，它提供了一个简单的 API，用来为一个运行进程建立核心转储。函数 WriteCoreDump 会把当前进程的一个转储写进一个指定的文件中。你也可以使用函数 WriteCompressedCoreDump 来把转储写进 bzip2 或 gzip 的压缩格式中。最后还有一个函数 GetCoreDump，可以用来建立进程的一个写时拷贝（copy-on-write）的快照，并将文件句柄返回给你。目前这个库支持 x86、x64 和 ARM 系统，同时也可以很容易地将这个库加入到你的项目中去。

处理堆栈溢出

到目前为止，我们前面介绍的实现可以处理大部分程序崩溃的情况，唯一不适用的就是堆栈溢出这种错误了。在能够处理这个特殊的错误情形之前，你需要了解堆栈溢出后线程的状态。当程序启动时，堆栈就被初始化成一个很小的空间（见图 1.10.1（a））。在堆栈最后一页的后面放置了一个所谓的守卫页面（guard page）。如果游戏用完了所有的堆栈内存，就会发生下面的事情（见图 1.10.1(b)）：

（1）当在访问堆栈的时候发现堆栈指针指向了守卫页面，就会抛出 STATUS_GUARD_PAGE_VIOLATION 异常。

（2）移除这个页面上的守卫保护，这个页面就变成了堆栈的一部分。

（3）紧接着堆栈的最后一页，分配一个新的守卫页面。

（4）从造成异常的指令处继续执行。

如果堆栈达到了它的最大尺寸，第三步分配一个新的守卫页面就会失败。在这种情况下，就会引起一个堆栈溢出的异常，这个异常可以由线程的异常模块来处理。正如在图 1.10.1（c）中看到的那样，堆栈现在只剩下一个可供使用的空闲页面。如果打算把这些空间也用掉，就会引起一个访问冲突，紧接着应用程序就会被操作系统终止。

这也就是说你没有那么大的空间来工作了。例如，如果调用 MiniDumpWriteDump 或者甚至是 MessageBox，游戏就会引起一个访问冲突的错误，因为这些函数的堆栈开销太大。然而，这里有个简单的小技巧可以解决这个问题。因为你有足够的空间来调用 CreateThread 函数，所以可以把所有的异常处理转移到新的线程中。然后就可以在一个空间较为充裕的堆栈中继续工作，写你想写的错误报告了。在异常处理中，你可以等到这个工作线程结束自己的工作以后才返回。

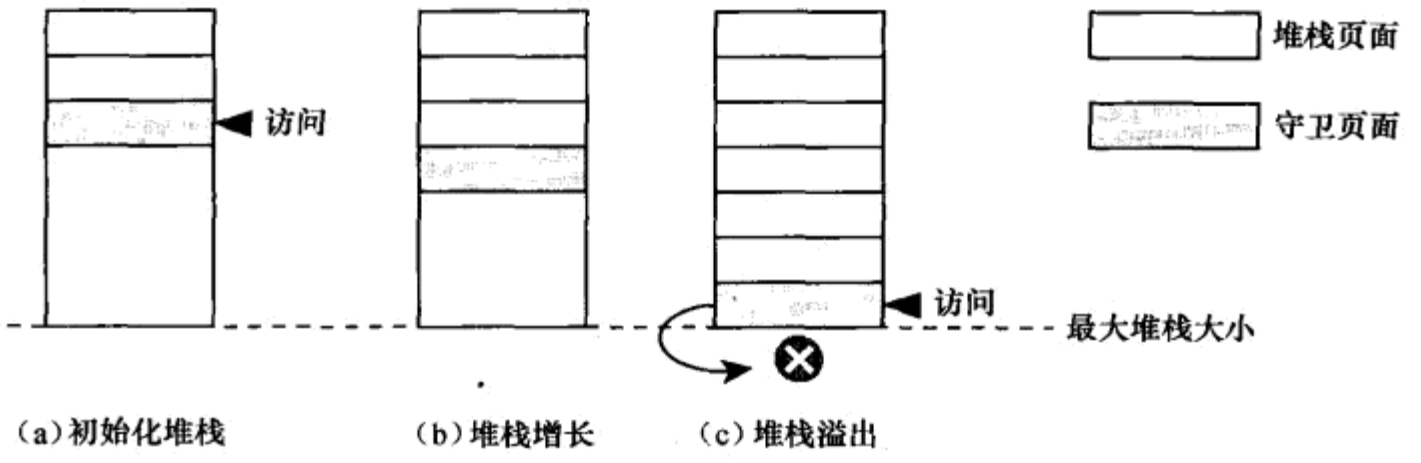


图 1.10.1　初始化堆栈、堆栈增长和堆栈溢出

不幸的是，UNIX 操作系统不支持这么做。当信号处理模块被调用时，堆栈已经被消耗光了，所以你基本不可能正常地汇报错误。能够解决这个问题的唯一方法就是使用 sigaltstack 函数建立另外一个堆栈来执行信号处理模块。所有通过 SA_ONSTACK 标志引入的信号处理模块都会在那个堆栈中执行，所有其他的处理模块还是在当前堆栈上执行。

为了成功地建立一个核心转储，你必须为新的堆栈申请至少 40KB 的内存空间。这要按照实际情况来看，你要做的事情越多，需要申请的空间越大。因为在多线程环境下使用某些 sigaltstack 的实现会造成一些问题，所以你应该查看目标平台的相关文档。

1.10.2 内存泄露

如果在游戏中采用的不是全局的内存管理机制，最好还是考虑使用一个内存泄露检测器来帮助检测内存泄漏。目前有大量的工具可以提供这方面的帮助。但不是所有的工具都很好用，其中一些还会引起一些问题。比如说这些工具中用得最多的方法就是重载 new/delete 操作符。然而这个方法有如下一些缺陷。

- 系统的头文件都必须包含在工程头文件的前面。如果不按照这个规则去做，就会造成内存泄露，甚至程序崩溃。
- 使用函数 malloc 分配的内存无法被检测出泄露。
- 会和其他重载了这些操作符的库冲突（比如 MFC）。

当然，我们也可以找到许多外部的内存泄露检测工具，但一般来讲它们都比较贵。本文介绍的内存泄露检测工具使用了一种叫做分配钩子（allocation hooks）的技术来跟踪应用程序的内存请求。分配钩子是由 C 运行库提供的，允许用户重载内存管理函数，例如 malloc 或者 free。使用钩子的优势就是无论内存在哪里或者如何被分配或者回收，运行库都会通过钩子函数通知你。你要做的就是在做任何分配动作以前把钩子加入进去。

加入分配钩子

微软的 CRT 允许你通过它的_CrtSetAllocHook 函数来导入分配钩子。那么用户应该在什么时候、什么地方加入钩子呢？答案就是越快越好，这样就不会错过任何一次内存分配请求。不过这还是挺难做到的。在 main 函数中加入也会比较晚，因为存在一些全局对象在其构造期间动态分配内存。即便把泄露检测器定义为一个全局实例，也无法保证这个全局实例会在其他的全局实例之前被初始化。解决这个问题的一个方法就是使用微软提供的#pragma init_seg(lib)指令。通过使用这个预处理命令，可以告诉编译器首先初始化在当前源文件定义的对象，然后才初始化其他对象（当然，CRT 库的初始化不包括在内）。

GNU C 的库甚至允许用户用定义在 malloc.h 中的全局钩子变量来完全地替换内存函数。通过将自己的处理模块传递给__malloc_hook、__realloc_hook 和__free_hook，就可以完全控制所有的内存管理请求了。用__malloc_initialize_hook 处理模块可以很简单地实现钩子的初始化。这是一个简单的函数，没有参数或者返回值，在库导入了自己默认的分配钩子以后就会去执行这个函数。这里保存好默认的钩子是很重要的，因为在你自己的钩子函数中会用到它。否则，为了重载内存函数，你必须提供一个完整的实现。

实现分配钩子

泄露检测管理了一个分配注册表，这个注册表包含了被分配的内存块信息。每个块包含了一个唯一的标示符、块的尺寸和分配时的调用堆栈。调用堆栈的数据会在写错误报告时提供一些有意义的信息，这对解决问题很有帮助。在内存块被释放的时候，它的块信息就会从注册表中移除。当应用程序结束的时候，你就必须把注册表中遗留的节点列出来，以此来检查是不是存在内存泄露。分配钩子的任务就是在每次分配内存时用一些有用的信息去更新这个注册表。

传递给_CrtSetAllocHook 的钩子函数必须具有以下函数签名：

```
int AllocHook(int allocType, void* data, size_t size, int blockType,
        long request, const unsigned char* filename, int line);
```

这里最有用的参数就是 allocType、blockType 和 request。参数 allocType 指定何种操作被触发（_HOOK_ALLOC、_HOOK_REALLOC 或者_HOOK_FREE）。blockType 指定能够被用来过滤内存分配的内存块类型（这个实现中不会对 CRT 的内存分配进行泄漏检测）。最后，request 指定了分配的顺序，你可以用唯一的 ID 来预订顺序。函数结束时会返回一个整数，告诉你分配内存是成功了（返回 TRUE）还是失败了（返回 FALSE）。想知道关于分配钩子函数的更多信息，可以去看看[MSDNHook07]。

GNU C 库的钩子函数的签名和它们重载的运行时函数的签名很相似。每个函数收到一个额外的参数，这个参数包含了堆栈上的地址，而这些地址是在 malloc、realloc 或者 free 操作时找到的。钩子函数是按照如下方式工作的：

（1）重建原始的钩子函数；

（2）调用 malloc、realloc 或者 free；

（3）更新内存泄露检测的信息；

（4）备份当前的钩子函数；

（5）设置自定义的钩子函数。

这是分配钩子建议的工作流程，在 GNU C 库的文档中也有类似的描述[GNUC06]。这里没有要求提供 ID，而是使用由 malloc 或者 realloc 返回的地址作为唯一的内存块 ID。

1.10.3　Windows 错误汇报（WER）

通过 Windows Vista，微软把 Windows 反馈平台介绍给了大家，这个平台允许其他制造商和开发者访问通过 WER 发送给微软的内存转储信息。这是通过 Windows 质量在线服务（WinQual）来完成的，是微软提供的一个在线通道。这个服务是免费的，但需要一个验证了的代码 ID 来辨认正在提交软件或者访问 WER 数据库的公司。WinQual 把内存转储信息组织到一个叫水桶（bucket）的地方，在每个水桶中都包含了由相同的 BUG 引起的崩溃报告。下面的参数会被水桶机构用到：

- 应用程序名称——例如，game.exe；
- 应用程序版本——例如，1.0.1234.0；
- 模块名称——例如，input.dll；
- 模块版本——例如，1.0.123.1；
- 模块内部偏移——例如，00003cbb。

默认情况下，WER 会调用 dwwin.exe，这个程序会收集水桶里的数据，并为崩溃的进程建立最小的转储文件。WinQual 还提供这样的可能性，可以对水桶进行指定，或者获取在一个水桶中的更多信息。如果用户碰到了一个已知的 BUG，那么他们会收到 WER 的提醒。用户可以到供应商的网站上下载及时更新的补丁，也可以获取目前任何 BUG 的修补情况。供应商也可以通过执行 WMI 队列列出注册表主键或者询问用户，让他们填写问题表来获取更多的信息。

客户端的 API

在 Windows XP 操作系统上，WER 的实现很简单，只对开发者开放两个 API（ReportFault

和 AddERExcludedApplication）。ReportFault 这个 API 会唤醒 WER，而它本身则必须在应用程序的异常处理模块中被调用。这个 API 还会执行操作系统的应用工具 dwwin.exe，这个工具会向用户弹出错误报告，并询问用户是否填写和发送错误汇报，然后把汇报发送给微软。AddERExcludedApplication 可以被用来把一个应用程序从错误汇报分离出来（这个函数需要对操作系统注册表的 HKEY_LOCAL_MACHINE 有写的权限）。

Windows Vista 提供了一个全新的 API 来让开发者操作 WER。这个库包含了好几个用来建立和提交错误报告的函数。和旧的 WER API 的另一个不同点是：这个报告可以在执行的任何时候生成。下面的表格显示了几个由 Windows Vista 系统中的 WER 提供的重要函数的用法。如果想要知道所有可用的 WER 函数，可参考[MSDNWER07]。

函　数	描　述
WerAddExcludedApplication	从错误汇报中排除指定的程序
WerRegisterFile	为目前的进程注册一个文件来写报告
WerRemoveExcludedApplication	撤销前一个 WerAddExcludedApplication 的调用
WerReportAddDump	把一个转储信息加入到报告中
WerReportAddFile	把一个文件加入到报告中
WerReportCloseHandle	关闭报告
WerReportCreate	新建报告
WerReportSetUIOption	设置用户界面选项
WerReportSubmit	提交报告
WerUnregisterFile	从为当前进程生成的报告中删除一个文件

另一个由 Windows Vista 引进的特性就是新的应用程序恢复（Application Recovery）和重启 API（Restart API）。这个特性允许应用程序把自己注册为在崩溃之后可以重启或者恢复的程序。特别重要的就是 RegisterApplicationRecoveryCallback 函数，它可以让你注册一个恢复回调函数。这个函数在程序无响应或者出现一个没有被处理的异常时会被 WER 调用，那么也就是在这个理想的地方，我们可以存储游戏和玩家数据，让玩家在后面的游戏中可以调出这些存档。读者可以参考[MSDNARR07]，以获取关于这个新的 Vista API 的详细信息。

1.10.4 框架

本小节对光盘上提供的调试框架会做一个简单的概述。CDebugHelp 类包含了建立转储信息和调用堆栈的帮助函数。在 DebugHelp.cpp 中，你会找到 CdbgHelpDll，这是封装了微软调试工具 API 的一个类。它会自动载入最新版本的 dbghelp.dll 文件，而不是在系统中选择默认的那个版本。你是不是在代码中看到了 MiniDumpCallback 函数？这个函数允许你指定一个回调函数来控制能够加入到转储文件的信息。在我的实现中，我把异常处理线程给去除了，这个线程是你在控制堆栈溢出异常时启动的线程。这些函数的 UNIX 实现很简单，这里不需要做更多的说明了（用户可以在[GNUC06]中找到关于 backtrace 和 backtrace_symbols 的文档）。

异常处理

异常处理句柄都是 CDebugFx 类的一部分，被定义在 DebugFx.h 中。在 Windows 操作系统下，你需要设置 UnhandledExceptionFilter，这个函数可以建立一个最小化转储文件，然后调用一个用户自定义的回调函数来完成异常产生后的处理工作。在 UNIX 操作系统下，这个框架会为 SIGSEGV 和 SIGABRT 注册一些信号句柄。我把 Google 的 cordumper 源代码也包含了进来，以消除对这个外部库的依赖。

内存泄露检测

内存泄露检测器是一个全局的实例，当把调试框架链接到你的程序时，这个实例就会自动地被激活。报告工作是由报告函数来完成的，所以除了这些汇报以外，用户还能制作自己的报告。在 Windows 操作系统上，默认的报告会把内存泄露的信息写进调试输出窗口，而在 UNIX 上则写到错误输出上。默认的汇报还有一个功能，就是从调用堆栈上把无用的信息给过滤掉。把调用 new 操作的函数或者 CRT 的分配函数列出来只会使文件变大，而对找到 BUG 没有一点好处。

有时候会出现一些关于符号解析的问题，尤其是在一个模块被释放后又在调用堆栈中被引用时。在这种情况下，用于解析符号的 API 无法解析出一个符号名称的地址。在 Windows 下，你可以使用函数 SymLoadModule64 重新载入一个模块的符号表。为了能够使用这个函数，你只需要把模块的基地址（也就是模块的句柄）、模块的名称和分配信息保存起来就可以了。令人遗憾的是，GNU C 库为符号解析只提供了一个函数 backtrace_symbols。在这个平台上，唯一的解决方案是存储解析后的符号，而不是所有分配的调用堆栈。因为这种方法会消耗很多内存，并且会造成性能损失，所以目前在 UNIX 平台上的内存泄漏检测不支持这个特性。

1.10.5　结论

通过合理地处理应用程序的崩溃，你可以在调试时得到非常宝贵的信息。根据不同的平台，你可以有不同的方法来处理异常，例如 Windows 的未被处理异常的过滤器或者 UNIX 信号句柄等。内存转储是一个非常好的用于程序调试的事后剖析工具，因为它能在程序崩溃时为你提供一个快照。通过在 Windows 上使用线程异常处理和在 UNIX 操作系统上使用信号句柄堆栈，你甚至可以处理堆栈溢出。你还可以通过内存泄露检测把讨厌的内存泄露赶走。通过 CRT 提供的分配钩子，你可以在游戏结束时通过一个完整的堆栈跟踪把所有的泄露都汇报出来。最后，通过使用 CDebugFx 类，你可以得到一个完整的调试框架，这个框架通过建立内存转储来处理未被处理的异常。内存泄露检测是通过 CMemLeakDetector 类来实现的，当你把这个库链接到游戏中时就自动被激活了。有了这些工具，在你今后的项目中，BUG 就不会有存活的机会了。

1.10.6 参考文献

[GNUC06] GNU C library documentation. "The GNU C library."

[Google05] Google coredumper library.

[Pietrek97] Pietrek, Matt. "A Crash Course on the Depths of Win32 Structured Exception Handling."

[Pietrek01] Pietrek, Matt. "New Vectored Exception Handling in Windows XP."

[MSDNDump07] MSDN Library. "MiniDumpWriteDump."

[MSDNHook07] MSDN Library. "Run-Time Library Reference—_CrtSetAllocHook."

[MSDNWER07] MSDN Library. "WER Functions."

[MSDNARR07] MSDN Library. "Application Recover and Restart Reference."

[Starodumov05] Starodumov O. "Effective Mini-Dumps."

[Wikipedia] Wikipedia. "Signal (Computing)."

第 2 章

数学和物理

简　介

Graham Rhodes, Applied Research
Associates, Inc.
grhodes@nc.rr.com

数学让世界运转！至少在电子游戏的领域中，这是绝对真理。在电子游戏的每一个环节中，我们以各种各样的方式使用数学。当下应用最多的领域包括二维及三维游戏世界的渲染、各种类型的人工智能算法以及飞速发展的物理模拟。这些领域应用的复杂程度是如此令人敬畏。正是因为有了可编程图形硬件，有了像素级别的基于物理的光照和材质模型的计算能力，实时 3D 渲染的模型表面不再像发光塑料。

角色 AI 的智能水平已经接近 Mori 理论的诡异谷（uncanny valley），我们很快将看到它被跨过。在实时游戏模拟的领域中，要想实现这个预言，不仅需要 AI，也要依靠基于物理的动画技术。这种技术让游戏角色可以与动态的游戏世界交互。在这个动态的游戏世界中，可收集的物体不再只是放在预先设定好的位置；游戏世界本身也会因为玩家的动作发生几何变化。这个动态的游戏世界必须和很多复杂的角色进行交互，同时他也越来越更基于物理计算。通过应用各种强大的物理引擎，一些当代的游戏，已经发布的或即将发布的，已经实现了完全的动态环境。当然，是模拟人类和自然的数学模型使这一切伟大的娱乐技术变为可能。

在这里，我想对游戏开发中一个基本由数学驱动的重要新兴领域做一点简单评论。这个领域就是“程序建模”，或者被称为“程序生成”或“可生成的模型”，可能还有一打别的叫法。现在商业游戏开发中的一个现实情况是：用传统数字化工具来开发游戏内容的成本在大幅提高。这是由于计算机硬件和游戏引擎已经可以支持前面提到的那些特性，游戏内容可以变得越来越复杂。当然使用的工具也在发展，美术的工作效率比以前任何时候都更高，但通常游戏内容庞大的绝对数量还是个问题。

因此，大工作室已经开始在某种程度上向老的游戏开发方式表达敬意，并从演示阶段就开始模仿那些已经多年致力于发明内容生成算法的开发者。程序建模能以各种形式极大地减少内容开发的成本。使用这项技术，美术团队就不再需要为每一种在场景中出现的树、灌木、草丛建模并摆放到场景中。在商业游戏的开发中，已经广泛应用程序建模来制作关卡中的植被。对游戏关卡中其他元素进行程序建模的工具也已经有了，但还不普及。这种工具能创建一些拥有多个细节程度从内到外都已模型化的关卡元素，比如建筑物，并在其中自动填充家具或者杂物，以及其他一些从第一

人称视角观察完美无缺的东西，甚至有一个正在开发中的饱受期待的游戏声称它将使用程序建模来生成细胞生物体、行星、恒星、星云以及在之间的许多物体。

在我看来，程序建模技术前途远大，美术家完全不必担忧会因此而丢掉工作。当然，要想让其实现，工具和游戏开发者必须运用合适的数学和物理技术。

在本章中，你将了解到多种有用的精粹。这一章将深入剖析一些经典的技术，以及一些能帮用户解决核心问题的新技术。其中有两个小节可以让用户更深入地理解应用于人工智能、物理技术以及程序建模的随机数生成技术。Chiris Lomont 对现有的随机数生成技术做了一个全面的综述，包括它们的优劣以及偶尔失效的情况，而 Steve Rabin 关于高斯随机性技术的心得提供了一个非常有效的实现。

Tony Barrera、Anders Hast 和 Ewert Bengtsson 总结了一项高效的技术来计算三角样条曲线，可用来生成包含直线段和完美圆弧的曲线。相对于其他分段曲线技术，例如传统三次多项式样条曲线，组合多段三角样条曲线可以生成视觉上更优美、更少曲率缺陷的曲线。这项技术非常适用于创建数字内容的工具以及内置于引擎的几何模型程序生成模块。Krzysztof Kluzek 在他的章节里继续介绍了其他一些对于程序生成模型非常有用的技术，向我们展示了使用投影空间来使几何运算更加健壮。相对于其他一些能获得类似结果的技术，该技术能节省存储空间和计算开销。

其余的 4 个章节专注于一直以来工业和学术研究的热点领域：碰撞检测以及其他几何查询。Jacco Bikker 对 k 维树空间分割技术做了一个极好的总结。他着重介绍了如何减少存储空间，以及如何创建基于不同查询类型优化过的树状结构，同时他也介绍了一种处理动态场景的方法。Jose Gilvan Rodrigues Maia、Creto Augusto Vidal 和 Joaquim Bento Cavalcante-Neto 描述了变换矩阵语义（transformation semantics），给出了一种变换矩阵的直观解释。他们的讨论向我们展示了这种直观的解释是如何运用到现代通用的碰撞检测算法中各个阶段的。

Rahul Sathe 和 Dillon Sharlet 介绍了一种碰撞检测的新技术，在粗检测阶段和细检测阶段都有应用。最后，Gary Snethen 介绍了一种由 GJK 算法衍生而来的碰撞检测技术，这种算法简单、直观、优美，而且相当灵活。使用这些技术去开发梦想吧，年轻人！

2.1 随机数生成

Chris Lomont

这是一篇介绍随机数生成器（Random Number Generators, RNGs）的文章。本文的主要目的为程序员展示一个起点，帮助他们选择最适合的 RNG，以及如何去实现，其次介绍了一个算法，来替代经常用到的马特赛特旋转算法（Mersenne Twister，MT），最后介绍各种随机数生成器以及其各自的优劣。

2.1.1 背景：随机数生成

随机数生成器（RNG）是许多计算机应用程序必不可少的要素。对于一些问题，含有随机选择要素的算法，比任何已有的、不含随机选择要素的算法都表现得好。如果允许随机性的存在，去寻找一个解决特定问题的算法就要容易得多（在配备随机数生成器的图灵机上可有效地解决的这类问题被称为 BPP，如果 BPP = P，这个问题就是一个开放式的问题，其中 P 是在没有随机性的计算机上可以解决的那类问题）。

大多数在计算领域用到的随机数都不是真正随机的，而是由伪随机数生成器（PRNG）生成的。伪随机数生成器都是确定性的算法，并且是唯一一种可不需要使用外界熵，诸如热噪声或用户行为来生成随机数的算法。

设计一个好的随机数生成器是非常困难的，最好由专家来完成（美国橡树岭国家实验室的 Robert R. Coveyou 曾很幽默地将一篇文章命名为“随机数生成对我们来说太重要了，我们不能让它随机生成”）。就像密码学一样，随机数生成的历史充满了各种坏算法以及其带来的恶果。一些历史上的错误将在章节的末尾提到。

应　用

随机数在许多应用程序中被用到，包括以下几种。

- AI 算法，比如遗传算法和自动化的对手。
- 随机游戏内容和关卡生成。
- 模拟复杂现象，比如天气和火焰。
- 数学方法，比如蒙特卡洛积分（Monte-Carlo intergration）。
- 素性证明之前一直在使用随机算法，直到最近才有所改变。
- 加解密算法，比如 RSA 使用随机数来生成关键码。
- 气象模拟和其他统计物理测试。

- 最优化算法大量地使用了随机——模拟退火、大空间搜索以及组合搜索。

硬件随机数生成器

因为一个算法是无法生成真正随机数，所以人们设计了很多基于硬件的随机数生成器。一些量子力学上的事件无法被预测，所以被视为很好的随机性来源。这些量子现象包括：

- 原子衰减检测，类似于烟雾检测器；
- 电路中的量子力学噪声，被称为散粒噪声；
- 穿过一个部分镀银镜面的光子流；
- 由高能 X 射线产生的粒子旋转。

还有其他一些物理随机源如下：

- 大气噪声；
- 电子学中的热噪声。

其他一些物理现象也经常在计算机上使用，诸如时钟偏移、键盘和鼠标输入、网络流量、附加硬件设备，或者是从移动风景采集到的图像。每个来源都必须通过分析来决定这个来源中有多少熵，以及能从中抽取中多少个高质量的随机位。

伪随机数生成器

伪随机数生成器通过对一个内部状态进行运算，来生成一个看似随机的数列。这个初始的状态被称为种子。为一个特定算法选一个好种子是非常困难的。通常，算法使用返回的随机值作为内部状态。因为这个内部状态是有限的，伪随机数生成器会在一些情况下重复。一个随机数生成器的周期就是在它开始重复之前所能返回的随机数个数。一个使用 n 位内部状态值的伪随机数算法最多只有 2^n 的周期。用相同的种子开始一个伪随机数生成器，将得到相同的随机数列，不过这个特性在调试其他模块时非常有用。当伪随机数生成器需要一个“随机”的种子时，通常会使用来自系统或者外部硬件的熵来源。

考虑到计算需要、内存需求、安全需要以及随机数质量的不同要求，人们设计了多种不同的随机数生成算法。没有一个算法是放之四海而皆准的，就像没有一个排序算法在所有情况下都表现最优一样。许多人默认使用 C/C++的 rand()函数或者是马特赛特旋转算法（Mersenne Twister），这两种算法都非常有用，会在本节中介绍到。

常见分布

大多数随机数生成器返回一个从[0, m]之间均匀选择的整数，最大值为 m。C/C++实现提供了一个 rand()函数，其中 m 被定义为 RAND_MAX，通常是 15 位整数——32 767。srand(seed)函数用来设置初始种子，一般使用当前的时间来作为熵来源，如 srand(time(NULL))。大多数 C/C++实现的 rand()函数都是使用线性同余生成器。线性同余生成器在加密方面是很糟糕的选择。大多数 C/C++实现（其他语言也一样）都生成质量很低的随机数列，表现出多种偏向性。

在游戏中最常用的分布是均匀分布。均匀分布需要从[a, b]间同等随机地挑出一个整数。一个常见的错误是使用如下的C代码：(rand()%($b-a$+1))+a。这种错误的用法会导致不是所有的值都以同等概率出现。只有当($b-a$+1)可整除 RAND_MAX+1 时，上面的用法才没有问题。

举例来说，假设 RAND_MAX 是 32 767，试生成一个在[0, 32 766]之间的随机数，使用上述方法就会导致 0 出现的概率是其他值的 2 倍。一个可行（但是更慢）的解决方案是将随机输出放缩至[0, 1]区间，然后再放缩至[*a*, *b*]区间，如下所示：

```
double v = (static_cast<double>( rand()) ) / RAND_MAX;
return static_cast<long>(v*(b-a+1)+a);
```

其次比较常用的分布是高斯分布。高斯分布可由一个均匀分布生成。先由 randf()返回[0, 1]间的均匀分布的实数，再调用 Box-Muller 变换，它的极坐标形式将每次产生两个高斯值，y1 和 y2。代码如下：

```
float x1, x2, w, y1, y2;
do {
    x1 = 2.0 * randf() - 1.0;
    x2 = 2.0 * randf() - 1.0;
    w = x1 * x1 + x2 * x2;
    } while ( w >= 1.0 );
w = sqrt( (-2.0 * log( w ) ) / w );
y1 = x1 * w;
y2 = x2 * w;
```

Boost[Boost07]的文档从均匀分布开始介绍了生成其他分布的技术。

2.1.2 随机性测试

要测试一个序列是否随机，首先必须定义什么是随机，但随机性是很难精确定义的。在实践中，许多测试被设计来评测随机数生成器的质量，方法是检测有没有不像随机数列的一连串表现。

最著名的随机性测试集是 DIEHARD 集[Marsaglia95]，它由 12 个测试组成。DIEHARD 集被扩展为一个开源（GPL）测试集 DIEHARDER[Brown06]。DIEHARDER 集囊括了 DIEHARD 集并增加了许多其他新测试，它还包括了许多随机数生成器和一套框架，使添加新的生成器非常容易。第三套测试框架是 TestU01 [L'Ecuyer06]。这些框架确保了通过测试的随机数生成器不至于有明显问题。

2.1.3 软件漂白

许多随机源的bit位都有一些偏向性或者相关性。用于去除这些偏向性和相关性的方法被称为漂白算法（whitening）。一些选择包括如下几种。

- John von Neumann 算法。每次取两位，当 00 和 01 时丢弃，01 时输出 1，10 时输出 0。去除了均匀偏向性，代价是需要更多的位。
- 每隔一位翻转，去除均匀偏向性。
- 与另一个已知的好的随机位来源做异或操作，就像 Blum Blum Shub 算法一样。
- 应用像 Whirlpool 或 RIPEMD-160 的加密哈希算法。注意 MD5 已经不再安全。

这些漂白过的数据流不经过进一步的处理还是不能视为安全的随机源。

2.1.4　不加密随机数生成算法

不加密算法一般会比加密的算法快，但是不能在需要安全的情况下使用，因此我们分类介绍。以下的每个算法都是伪随机数生成器，产生序列 *Xn*。其中有一些算法使用一个隐藏的内部状态 *Sn*，由此生成 *Xn*。*S*0 或者 *X*0 都可以是种子，视情况而定。

平方取中法

这是由 John von Neumann 在 1946 年提出的，取一个 10 位数作为种子，取平方，返回中间 10 位作为下一个数和种子。这个方法在 ENIAC 中用到，是一个有统计缺陷的差算法，已经不再使用。

线性同余生成器（LCG）

这是一个广泛使用的最常见的算法，但逐渐被一些新算法所取代。数列由 $X_{n+1}=(a\times X_n+b) \bmod m$ 生成，其中 *a* 和 *b* 是常数。模数 *m* 通常会使用 2 的乘方作为高效的位掩码。*a* 和 *b* 需要小心挑选来保证最大的周期，并避免其他一些问题。LCG 算法有多种缺陷，其中之一就是：3 个一组作为空间中的点坐标，在空间绘出这些点后，你会发现它们都在一些平面上。这将在 2.1.6 节介绍 RANDU 时展示，这个缺陷是由于连续点线性相关。使用 2 的乘方为模数 $m=2^e$ 的 LCG 已知表现很差，尤其最低的几个有效位[L'Ecuyer90]。举例来说，《C 的数学库》[06 年出版]推荐使用 *a*=1 664 525，*b*=1 013 904 223，$m=2^{32}$，产生的随机数最低几位几乎不变。

LCG 的优点在于它是相对较快的算法，并且使用一个较小的状态，因此它被广泛地使用，包括在嵌入式程序中。如果模数不是 2 的乘方，取模运算通常会非常费时。

表 2.1.1 给出了一些常用的 LCG，使用 LCG(*m*,*a*,*b*)来表示一个 LCG。

表 2.1.1　　一些常用的 LCG

LCG	应　用
LCG(2^{31},65 539,0)	臭名昭著的 RANDU，后面会提到
LCG(2^{24},16 598 013,12 820 163)	微软 Visual Basic 6.0
LCG(2^{48},25 214 903 917,11)	UNIX 标准库的 drand48，在 java.util.Random 中被用到
LCG(10^{12}−11,427 419 669 081,0)	Maple 9.5 和 MuPAD 3 中用到， 在 Maple 10 中被 MT19937 替换（下面会提到）

截断线性同余生成器（TLCG）

这个算法保存一个内部状态 S_i，S_i 是使用 LCG 来得到的，然后依次生成输出 X_i。公式为：$S_{n+1}=(aS_n+b) \bmod m$，$X_{n+1}=Floor\left[\frac{S_{n+1}}{K}\right]$。这种算法允许使用一个快速的模数 $m=2n$，同时

避免了 LCG 中低位的缺陷。如果 K 也是 2 的乘方，除法操作也会很快。这个算法在微软的产品中被广泛使用（有可能是使用 VC++编译的缘故），VC++中的 rand()也使用了这个算法。rand()实现如下：

```
/* 微软对于 ramd()的算法 */
static unsigned long seed;
seed = 214013L * seed + 2531011L;
return (seed>>16)&0x7FFF; // return bits 16-30
```

这个算法是不安全的。实际上，对于一个加密分析工程，只需要得到该算法两个连续的输出就可以破解它的内部状态（包括一个不需要的最高位），也就获得了之后它所有的输出。一个简单的破解内部状态的方法是：注意该状态的最高位与之后的输出是没有关系的，所以只需要破解 31 位。第一个输出会给出该状态的 15 位，只有 17 位未知。再有两个输出，使用已知的 15 位并测试所有可能的 2^{17} 个状态，就可以知道哪个状态可以产生哪两个输出。这样已经可能得到内部状态了。两个输出是不够的，因为它们不能决定一个唯一的状态。

Borland C++和 TurboC 也都使用了 TLCG，设定 a=22 695 477 和 b=1。虽然 C 语言标准不强制要求实行一个 rand 函数，但《C Programming Language》 [Kernighan91]举出了一个例子是使用 TLCG，设定 a=113 515 245 和 b=12 345，RAND_MAX 最小为 32 767。

线性反馈移位寄存器法（LFSR）

线性反馈移位寄存器（LFSR，如图 2.1.1 所示）通过将一个内部状态一次一位地移出来产生随机位。新的数位将通过当前状态的线性函数获得并移入这个状态。LFSR 的流行是因为它非常快速，在硬件上容易实现，并且能生成大范围的数列。通过选择合理的流水序列（反馈输入位）可使一个 n 位的 LFSR 拥有 2^n-1 的周期。只需要 2^n 个输出位就可以推导出 LFSR 的结构和反馈连接，所以这个方法肯定是不安全的。

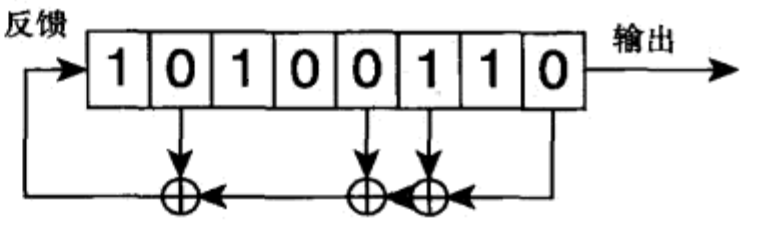


图 2.1.1 线性反馈移位寄存器（LFSR）

逆同余生成器

这个算法类似于 LCG 但不是线性的，$X_{n+1}=(aX_n^{-1}+b)\bmod m$，其中 X_n^{-1} 是 X_n 模 m 的乘法逆数，也就是 $X_n\ X_n^{-1}\equiv 1\bmod m$。因为求逆操作非常费时，所示这个算法并不常用。

滞后斐波那契生成器（LFG）

使用 k 个状态，$X_n=(X_{n-j}\otimes X_{n-k})\bmod m$，$0<j$，其中 $\otimes$ 是一个二元运算（加、乘、异或、其他）。这种算法很难让其良好运行，也很难初始化。周期取决于初始种子，而值空间分割成很难预测的循环。因为马特赛特旋转算法和新的生成器的出现，这个算法已经被抛弃。Boost[Boost07]收录了多种 LFG。

细胞自动机

Mathematica 在 6.0 之前的版本使用 Wolfram rule 30 细胞自动机来生成大整数。版本 6.0 使用了多种方法。

线性递归生成器

这个算法是 LFSR 的一个泛化。现代大多数在二进制有限域的快速伪随机数生成器都是由它衍生而来。注意，这些算法都不能通过线性递归测试，因为都使用线性函数。接下来介绍的是这类 PRNG 的一些特殊例子，它们被视为最佳的通用 RNG。

马特赛特旋转法

在 1997 年，Makoto Matsumoto 和 Takuji Nishimura 发表了马特赛特旋转算法[Matsumoto98]。这种算法避免了之前生成器的很多问题。他们展示了两个版本，MT11213 和 MT19937，周期分别是 $2^{11\,213-1}$ 和 $2^{11\,937-1}$（大约 $10^{6\,001}$）。MT19937 的周期应该能运算到远比整个宇宙的生命还要更长的时间。它使用 624 个长整型或者 19 968 位作为内部状态，这大约符合超大周期的预期。它比 LCG 更快（惊人吧），在高达 623 个维度是等分布的。它已经成为统计模拟中主要使用的 RNG。

它之所以这么快，是因为每次生成随机数时，它只更新内部状态的一小部分，通过多次调用来遍历这个状态。马特赛特旋转算法是一个旋转广义反馈移位寄存器。它不是密码安全的：观察 624 个连续输出就可以破解它的内部状态而预测之后的输出。马特赛特旋转算法有一些缺陷，将在下面的“WELL 算法”部分提到。

LFSR113、LFSR258

[L'Ecuyer99]介绍了组合 LFSR Tausworthe 生成器 LFSR113 和 LFSR258，特别为 32 位和 64 位计算机设计。它们的周期分别为大约 2^{113} 和 2^{258}。它们非常快速、简单，只使用极小的内存。举例来说，以下是 LFSR113 的 C/C++代码实现，返回一个 32 位值。

```
unsigned long z1, z2, z3, z4; /* the state */
/* 注意，种子必须满足
     z1 > 1, z2 > 7, z3 > 15, 和 z4 > 127 的条件 */
unsigned long lfsr113(void)
{ /* 产生随机的 32 位数字 */
unsigned long b;
b = (((z1 << 6) ^ z1) >> 13);
z1 = (((z1 & 4294967294) << 18) ^ b);
b = (((z2 << 2) ^ z2) >> 27);
z2 = (((z2 & 4294967288) <<  2) ^ b);
b = (((z3 << 13) ^ z3) >> 21);
z3 = (((z3 & 4294967280) <<  7) ^ b);
b = (((z4 << 3) ^ z4) >> 12);
z4 = (((z4 & 4294967168) << 13) ^ b);
return (z1 ^ z2 ^ z3 ^ z4);
}
```

因为 2^{113} 大约为 10^{34}，这个算法已经可以生成超大数量的值了，而且占用的内存比 MT19937 要小。这个 LFSR 生成器也是良好等分布的，避免了 LCG 的问题。

WELL 算法

Matsumoto（马特赛特旋转算法的发明者之一）、L'Ecuyer（一个 RNG 的主要研究人员），和 Panneton 在 2006 年推出了另一类 TGFSR PRNG[Panneton06]。这类算法产生的随机数比 MT19937 拥有更好的等分布性，并且在“位混合”（bit-mixing）属性上做了改进。WELL 是良好等分布长周期线性（Well Equidistributed Long-Period Linear）的缩写，这类算法似乎在任何情况下都比现在用的 MT19937 要好。它们非常快速，有多种周期，最重要的是能产生更高质量的随机数。

根据状态的大小，WELL 的周期可以是 2^n，n=512、521、607、800、1 024、19 937、21701、23 209 和 444 497。这样允许用户牺牲周期长度来换取较小的状态。所有的 WELL 算法速度相当。2^{512} 大约为 10^{154}，大概没有视频游戏需要用到这么多的随机数，因为这个数字已经远远超过了宇宙中的粒子数。更大周期的 WELL 算法只有在天气建模和地球模拟之类的计算中才用到。一台标准 PC 需要超过 1 *googol* 年来计算 2^{512} 个随机数（1 *googol* 是 10^{100}，google 它。“Google”就是来源于该词）。

相对于 MT19937，WELL 的一个显著优点是能快速跳过大量 0 位的内部状态。如果 MT19937 的种子有大量的位是 0，或者在某种情况下进入了一个有大量位是 0 的状态，它接下来生成的几个随机数会很大程度地偏向 0。WELL 算法表现得要好得多，很快就能跳过偏向 0 的状态。

唯一的缺点是 WELL 算法比 MT19937 要慢一点，但不会慢很多，而优点是随机数的质量更高，而且代码更简单。以下是作者写的实现 WELL512 的 C/C++代码（如果你使用这段代码，请注明出处或者写封邮件感谢作者）。这段代码比 L'Ecuyer 网站上的实现大概要快 40%，比 Matsumoto 网站上的 MT19937 实现要快 40%。

```
/* 初始化状态到随机位 */
static unsigned long state[16];
/* 初始化必须为零 */
static unsigned int index = 0;
/* 返回 32 位随机数 */
unsigned long WELLRNG512(void)
{
  unsigned long a, b, c, d;
  a = state[index];
  c = state[(index+13)&15];
  b = a^c^(a<<16)^(c<<15);
  c = state[(index+9)&15];
  c ^= (c>>11);
  a = state[index] = b^c;
  d = a^((a<<5)&0xDA442D20UL);
  index = (index + 15)&15;
  a = state[index];
  state[index] = a^b^d^(a<<2)^(b<<18)^(c<<28);
```

```
    return state[index];
}
```

2.1.5　加密 RNG 方法

密码安全的 PRNGS（CSPRNGS）让攻击者即使获得了大量的输出，也很难破解生成器的内部状态或者预测未来输出。一些 CSPRNG 已经被标准化而且可以在网站上找到。两个关于随机性安全标准的 RFC 文件是 RFC1750 和 RFC4086。这些算法都应该小心实现，以避免陷阱。如果可能，尽量使用来源可靠的实现，并确认它是有效的。

Blum Blum Shub

由 Lenore Blum、Manuel Blum 和 Michael Shub 发表于 1986 年的 Blum Blum Shub 算法[Blum86]被视为一个安全的 PRNG。它通过计算 $S_{n+1}=(S_n^2) \bmod m$，其中 $m=pq$，p、q 是两个合理选择的大素数。然后随机数 X_{n+1} 由 S_{n+1} 计算出，通常是 S_{n+1} 的奇偶校验得出或者取 S_{n+1} 的特定位。它的强度依赖于整数因式分解的难度，RSA 的公共密钥安全同样依赖这个问题。Blum Blum Shub 只是在需要加密时使用，因为它比不加密的 PRNG 要慢很多（鉴于 Shor 的量子因式分解算法可高效地分解整数，所以一旦量子计算机投入使用，Blum Blum Shub 也就不再安全了）。

ISAAC、ISAAC+

[Jenkins96]介绍了 ISAAC，一个基于 RC4 密码变体的 CSPRNG。作为一个 CSPRNG 来说，它是比较快的，产生一个 32 位值平均需要 18.75 条指令。ISAAC 没有小于 240 的周期，预期的周期为 28 295。ISAAC-64 是 64 位机上的版本，每产生一个 64 位随机值需要 19 条指令。

/dev/random

虽然不是一个特定的算法，Linux 和许多 UNIX 衍生版都在/dev/random 命令下实现了一个随机源，它返回一个基于系统熵的随机数，因此可被视为真随机数生成器。/dev/random 是阻塞的，意味着在收集到足够多的熵之前不会返回。因此，大多数程序使用不阻塞的/dev/urandom。但这样，得到的随机数就不是那么安全，而且使用/dev/urandom 会耗尽系统熵，稍差的实现很容易被攻击。没有特定的底层算法，有些系统使用了 Yarrow，将在下面的部分中提到。

[Gutterman06]揭示了当时 Linux 实现中一个可被利用的漏洞，当然现在可能已经被修补了。总的来说，在 Linux 上，/dev/random 还是较受青睐的 CSPRNG。

微软的 CryptGenRandom

微软 CryptoAPI 函数库中的 CryptGenRandom 可以生成密码安全的随机字节来填充一个缓冲区。和/dev/random 类似，它也可以被视为真随机数生成器。虽然未开放源文件，但它通过了 FIPS 验证，所以被认为是安全的。笔者还不知道该函数最新实现有任何缺陷。在 Windows

上，这是不错的 CSPRNG。

Yarrow

[Kelsey99]介绍了 Yarrow，一种使用系统熵来生成随机数的算法。它没有申请专利，绝对免费，不需要任何许可就可以使用。Mac OS X 和 FreeBSD 用 Yarrow 来实现了/dev/random。Yarrow 的原作者已经不再对其进行支持，而是发布了一个改进的 Fortuna 算法。

Fortuna

Fortuna 是《Practical Cryptography》[Ferguson03]书中介绍的另外一个 CSPRNG。这个生成器基于任意良好的分组密码，并在计数模式下加密，加密后续计数器的值。它的密钥是周期性变动的，以避免一些统计缺陷。它使用一个熵池来收集对系统有效的随机源信息。因为它使用外部熵，所以被视为真随机数生成器。

2.1.6 创造随机数生成器的常见错误

创造一个好的随机数生成器是非常重要的。在 RNG 的历史上，坏的设计随处可见。用户总是可以从别人的错误中学到经验，所以让我们来看看这个领域中的一些常见错误。

Knuth 例子

即使是算法大师也会犯错，Donald Knuth 在[Knuth98]中提到：他曾经试图创造一个“超级”随机数生成器来产生随机数。他第一次运行停在一个 10 位数上然后不断重复，第二次运行在返回 7 401 个值之后陷入一个周期为 3 178 的循环。

这里还有一些例子，希望能警示人们，不要在重要程序里再使用自制的 RNG。

RANDU

RANDU 是一个臭名昭著的 LCG，从 20 世纪 60 年代开始使用。RANDU 是 LCG ($2^{31,65539,0}$)，初始种子必须是奇数。这些常量的选择是想使实现简单快速。像所有的 LCG 一样，这个算法的输出一样有线性相关的缺陷。图 2.1.2 展示了 10 000 个 3D 绘制出来的三元组。它们全在一些平面上。

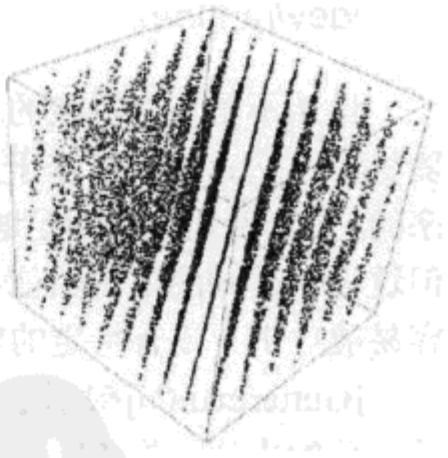


图 2.1.2 LCG 偏向性

Netscape

Netscape 早期的一个版本需要使用 CSPRNG，用它的结果来作为密码。但是这个 CSPRNG 使用的 3 个种子没有完全散布开（当前时间、本进程 ID、父进程 ID）。[Goldberg96]发表了一个对 Netscape SSL 协议的成功攻击，并指出这个缺陷是由于选择了很差的种子。

民间算法

1992 年前后，笔者看见一个游戏程序员自己写的算法。他解释说，他为他的 NES 代码

写了一个快速简单的 PRNG，基本想法是将数位从种子中移出，每当从种子中移出一位时，与一个常量做异或操作。这个算法是很快，粗粗一看也很好。但作为一个怀疑论者，笔者仔细考虑了随机性的要求，并提出这个算法会产生一个递减的序列，然后不时地跳回，形成锯齿状的输出。这个结论帮助原作者发现了一个他的 AI 代码中的 BUG，是由这个他从未怀疑过 PRNG 引起的（他已经使用这个 PRNG 好多年了）。虽然很麻烦，但把一个新的 PRNG 放入你的工具库之前，最好先测试并观测它的表现是否符合预期。

最后一个特别有趣的例子是一个随机数生成器的 xkcd Webcomic 版本。在 http://xkcd.com/c221.html 上可以看到。转载如下：

```
int getRandomNumber()
{
     return 4;          // 由骰子选择
                        // 确保是随机的
}
```

2.1.7　代　码

在网上有很多地方可以获得本文提到的那些算法的源代码。Boost [Boost07]包含了其中许多算法的高效实现，Wikipedia 有本文中大多数主题的相关信息和链接，L'Ecuyer 的网页中可以找到相关论文和许多实现。另外，《C++标准技术报告 1》（TR1）包含了许多分布和生成器（包括 MT19937），说不定哪一天 C++将内置这样特性。

2.1.8　结　论

本文提供了一些 RNG 的基本知识，包括一些常用的算法。LFSR113、LFSR258 和 WELL 生成器为许多程序提供了一个比马特赛特旋转算法更好的选择，希望更多的读者了解到这几个算法的知识。前面介绍了各种算法的优劣，能帮助用户决定该算法的使用场合。一个专业的开发者应该了解 RNG 的各种类型，并且知道什么时候用哪一种，就像应该了解多种排序算法和多种树结构。希望本文能为你提供这些基础知识并作为一个有益的参考。

2.1.9　参考文献

[Blum86] Blum, Lenore, Blum, Manuel, and Shub, Michael. "A Simple Unpredictable Pseudo-Random Number Generator," *SIAM Journal on Computing,* Vol. 15, pp. 364–383, May 1986.

[Boost07] "The Boost C++ Library," 2007.

[Brown06] Brown, Robert G., and Eddelbuettel, Dirk. "DieHarder: A Random Number Test Suite Version 2.24.4."

[Ferguson03] Ferguson, Niels, and Schneier, Bruce. *Practical Cryptography*, Wiley, 2003. ISBN 0-471-22357-3.

[Goldberg96] Goldberg, Ian, and Wagner, David. "Randomness and the Netscape Browser,"

Dr. Dobb's Journal, January 1996, pp. 66 – 70.

[Gutterman06] Gutterman, Pinkas, and Reinman. "Open to Attack: Vulnerabilities of the Linux Random Number Generator," March 2006, Black Hat 2006.

[Jenkins96] Jenkins, Bob. "ISAAC and RC4."

[Kelsey99] Kelsey, J., Schneier, B., and Ferguson, N. "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator," Sixth Annual Workshop on Selected Areas in Cryptography, Springer Verlag, August 1999.

[Kernighan91] Kernighan, B., and Ritchie, Dennis. *The C Programming Language, Second Edition*, Prentice-Hall, 1991.

[Knuth98] Knuth, Donald. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition,* Addison-Wesley, 1998.

[L'Ecuyer90] L'Ecuyer, P. "Random Numbers for Simulation," Communications of the ACM, 33 (1990), pp. 85–98.

[L'Ecuyer99] L'Ecuyer, P. "Tables of Maximally-Equidistributed Combined LFSR Generators," *Mathematics of Computation,* 68, 225 (1999), pp. 261–269.

[L'Ecuyer06] L'Ecuyer, P. and Simard, R. "TestU01: A C Library for Empirical Testing of Random Number Generators," May 2006, Revised November 2006, *ACM Transactions on Mathematical Software,* 33, 4, Article 1, December 2007.

[Marsaglia95] Marsaglia, George. "DIEHARD."

[Matsumoto98] Matsumoto, M., and Nishimura, T. "Mersenne Twister: A 623- Dimensionally Equidistributed Uniform Pseudorandom Number Generator," *ACM Trans. Model. Comput. Simul.* 8, 3 (1998).

[Panneton06] Panneton, F., L'Ecuyer, P., and Matsumoto, M. "Improved Long-Period Generators Based on Linear Recurrences Modulo 2," *ACM Transactions on Mathematical Software,* 32, 1 (2006), pp. 1–16.

[Press06] Press, William H. (Editor), Teukolsky, Saul A., Vetterling, William T., and Flannery, Brian P. *Numerical Recipes in C++: The Art of Scientific Computing*, Cambridge University Press, 2nd edition, 2002.

2.2　游戏中的快速通用光线查询

Jacco Bikker, IGAD/NHTV University of
Applied Sciences—Breda, The Netherlands
bikker.j@nhtv.nl

最近，实时的光线追踪在消费级 PC 上也变为可能了。使用指令级的并行（特别是 SIMD 代码[Wald04]），一个光线追踪器在一个核上每秒能追踪数百万条光线。使用线程级的并行系统，允许用户追踪的光线数随着核数线性增长。

光线追踪在渲染之外的领域也非常有用，本节将介绍一个通用光线追踪器的实现细节。它可用于游戏的多个环节，比如视线查询、物理和声音广播。本节的重点是单条光线的快速遍历。

使用表面面积启发式算法（Surface Area Heuristic, SAH [McDonald90]）创建的 K 维树，在静态场景中对于光线追踪来说是一种很有效的空间细分结构。本节将描述一些方法，把这个算法拓展至动态场景中。

2.2.1　光线追踪介绍

光线追踪是一个很简单的算法。当应用于渲染时，核心算法如下：

```
for each screen pixel                    // 对每个屏幕像素
construct a ray from the camera to the pixel
  //创建一条从相机指向该像素的光线
intersect the ray with each primitive in the scene
  //该光线与场景中每个图元做相交测试
shade the pixel based on the intersection results
  //基于相交测试结果为该像素着色
```

在这里，一条光线是指一根无限长的射线，拥有一个起点 O 和一个方向 D，如图 2.2.1 所示。

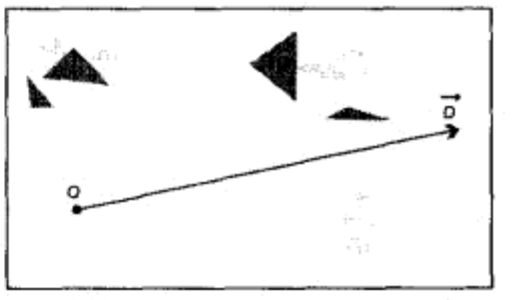


图2.2.1　一条光线包含一个起点和一个方向

追踪单条光线被称为光线投射（ray casting）。基础算法是由 Apple 在 1963 年发明的[Appel63]。在 1979 年，Whitted 通过加入递归拓展了这个算法 [Whitted79]。在交点处，向每个光源方向创建一条新的光线，来查询这个光源是否会影响该交点。如果交点处的材质具有反射性或者折射性，还要递归地生成新的光线。

如果你注意单条光线的追踪过程，就会发现这是一个可见性查询。摄像机需要知道通过某个屏幕像素的图元中哪个是可见的；交点需要知道从该点可以看见哪些光源。下面这类查询在游戏中是非常有用的。

- 一个 AI 敌人想要知道是否能看见或听见玩家。
- 玩家选择场景中的某个物件，使其高亮显示。
- 当一个狙击手向玩家发射一颗子弹时，如果未直接击中玩家，子弹将在场景中弹跳。也有可能子弹击穿玩家后继续飞行。
- 一辆悬浮的载具试图避开障碍物，为此需要侦测周围环境。
- 被杀死的敌人掉落了一件可拾取的物品，这件物品应该一直下落直至碰到地板。

上述一些情景很明显需要进行光线查询（玩家可见性、子弹发射、射线选取），还有一些通常依赖于物理引擎内置的碰撞检测系统（当物品掉落时），甚至使用一条或多条光线追踪会很容易实现。大多数 3D 引擎都以某种形式支持光线查询，但为了效率，游戏开发人员一般都尽量不使用光线查询，因为都认为光线查询会很慢。如果使用一种幼稚的搜索方式来对场景几何体做光线追踪，当然会很慢。但是，我们可以做得更好。通过使用优化过的空间细分结构，一次光线查询可以简化为几步树遍历和有限的几次射线与三角面片相交测试。

本节的其余部分将介绍了 *K* 维树以及如何高效实现之。*K* 维树被认为是最能加快光线追踪的结构。因为这种结构需要时间来做预处理，所以最好应用在静态场景中。但由于动态场景在游戏以及其他实时交互程序中的重要性，本节也将演示一些为动态场景设计的替代方法。在一个混合环境中，用户可以结合这两种方法。CD-ROM 中附带有演示程序作为示范。

2.2.2 *K* 维树的概念和存储考虑

K 维树（K-dimensional tree）是一种将空间递归分割为两半的树状结构。从这个意义上来说，它是一个 BSP 树。但是，相对于通用 BSP 树可以使用任意的分割平面，*K* 维树的分割平面只能是轴对齐的。下面是一个例子，如图 2.2.2 所示。

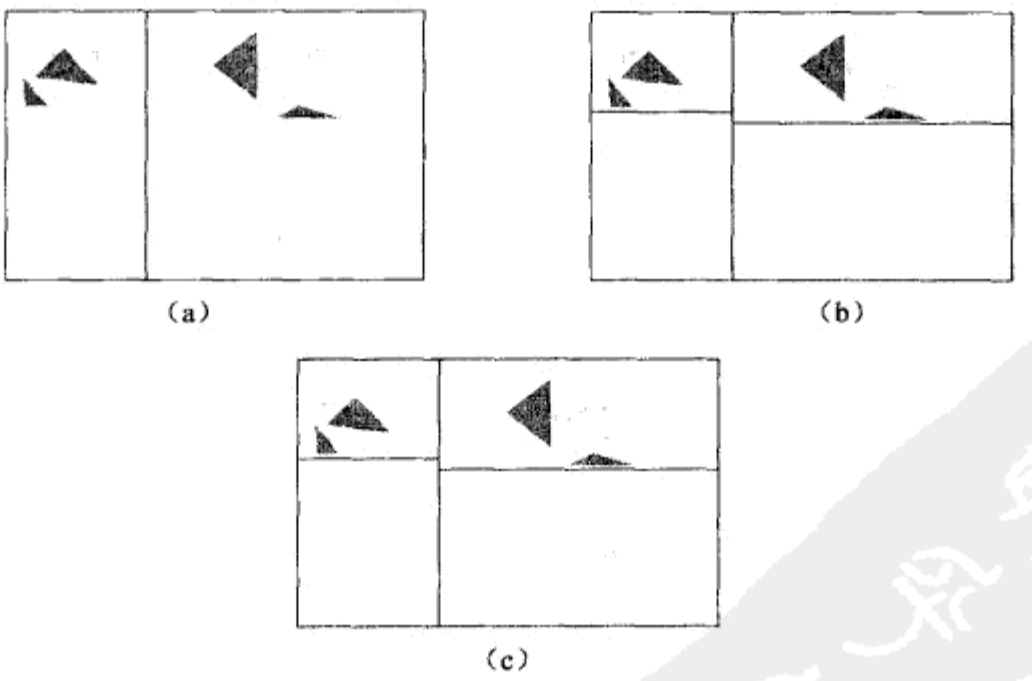


图 2.2.2 *K* 维树分割空间过程中的 3 次迭代

你可以看到在第三次迭代时左下角的部分没有被分割，这是因为它不包含任何几何体。同时也可以注意到，由于分割平面可以任意选择位置，几何体和空白空间很快就被区分开来。

只使用轴对称的分割平面看起来具有局限性，但在实践中，它会使你的射线遍历更加高效。求一条射线与一个轴对齐平面的交点是非常快速的，如下式所示：

```
t = (splitpos[axis] - ray.origin[axis]) / ray.direction[axis]          (1)
```

这里，t 是在射线上从起点到交点的距离。通过预计算射线方向各个值的倒数，这个公式可简化为只需要减法和乘法操作。

因此，一个 K 维树节点需要包含一个分割平面的位置和一个朝向。因为朝向可以是 x、y 或者 z 轴，两位就足以存储这个朝向的信息了。除了分割平面外，一个节点还需要存储图元的列表，或者指向左右子节点的指针。最后，需要存储一个标志位来标识这是一个内部节点（不包含几何体，只是被分割为两个子节点）还是一个叶节点（包含几何体，但是没有子节点）。

```
struct KdTreeNode
{
  float splitpos;
  int axis;
  KdTreeNode* left, *right;
  bool leaf;
  Primitive** primitive;
  int primcount;
};
```

如果精心规划一下，这个结构体可被存储为仅仅 8 字节。

- 在每次分割时成对地分配子节点，右子节点的指针就会是简单的 left+1。这样的话，只需要存储一个子节点指针。
- 把指向图元的指针存在一个独立的数组中，就只需在 K 维树节点中存储一个索引和计数（见图 2.2.3）。为了把索引和计数存储在同一个无符号整型数中，必须使用 5 位来存储计数，剩下的 27 位用来存储数组中的索引。

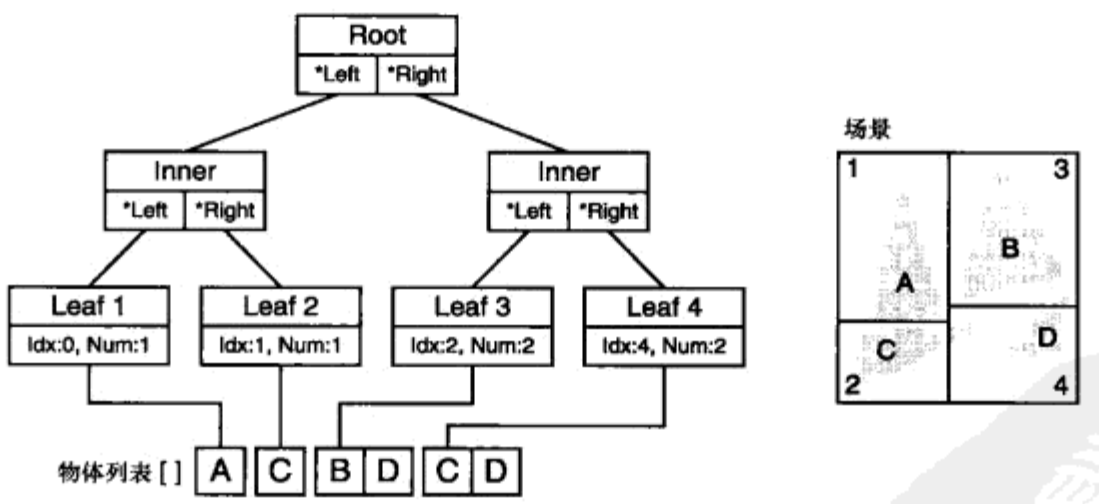


图 2.2.3　一个使用了共享数据结构从而减少了节点大小的 K 维树

- 如果这是一个叶节点，它就不要存储子节点指针。如果它是一个内部节点，它就不需要指向图元。因此这两个数据可安全地使用 kdTreeNode 数据结构中的同一个变量。
- 因为一个优化过的 kdTreeNode 需要 8 字节来存储，通过将 kdTreeNodes 做一个 8 字

节的边界对齐，kdTreeNode 的地址（因此左节点指针的值也同样是）将肯定是 8 的倍数。左节点指针的最低 3 位将是 0，这 3 位可用来存储叶节点标志（1 位）和分割平面轴（2 位）。因为这个数据被图元列表索引和计数所共享，所以存储索引及计数时，这个数据也必须向左移动 3 位。

现在 kdTreeNode 结构变为了：

```
struct KdTreeNode
{
    // 成员数据访问方法
    void SetAxis( int a_Axis ) { m_Data = (m_Data & -4) + a_Axis; }
    int GetAxis(){ return m_Data & 3; }
    void SetLeft( KdTreeNode* a_Left )
    { m_Data = (unsigned long)a_Left + (m_Data & 7); }
    KdTreeNode* GetLeft(){ return (KdTreeNode*)(m_Data & -8); }
    KdTreeNode* GetRight(){ return GetLeft() + 1; }
    int IsLeaf() { return (m_Data & 4); }
    void SetLeaf( bool a_Leaf )
    {m_Data = (a_Leaf)?(m_Data|4):(m_Data & -5);}
    int GetObjOffset() { return (m_Data >> 8); }
    int GetObjCount() { return (m_Data & 248) >> 3; }

    // 成员数据
    float m_Split;
    unsigned long m_Data;
};
```

在这个结构中，成员变量 m_Data 包含了子节点标志位，分割平面轴向，一个指向左子节点的指针或者是图元指针列表的一个入口和计数。多种 get 和 set 方法过滤出需要信息的相关位。

创建 K 维树

前面介绍了 K 维树的基本概念以及一个高效的存储方式，现在最大的悬而未决的问题是如何决定分割平面的位置。

如何才算一个好的 K 维树完全取决于你要如何使用它。如果你的目的是以尽可能少的次数从前到后遍历图元列表，可能需要的是一个平衡树，同时保证尽可能少的图元跨越分割平面。但对于光线追踪，你的目标就不同了——尽可能减少需要做相交测试的三角面片的数量，这就意味着你情愿去遍历一个空白的空间。以一个场景为例，只在左上角有一个小三角面片，如图 2.2.4 所示。

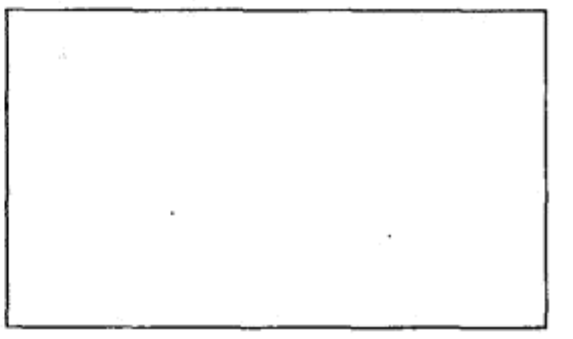


图 2.2.4 找到一个最优的分割平面位置

在这种情形下，最优的树是怎样的呢？这里有一些选择：

- 完全不分割；
- 通过该三角面片的右顶点的一次垂直分割；
- 通过该三角面片的下顶点的一次水平分割；
- 多次分割。

你可以注意到所有的分割都应该发生在包围盒的边界处，而所有其他的分割方法要么会切开该三角面片，要么向包含三角形的节点中添加多余的空白空间。完全不分割意味着每次

光线追踪都必须与这个三角面片做相交测试。一次水平或垂直的分割可以避免一部分光线的相交测试。多次分割将需要做相交测试的光线比例进一步减小。结论是，在如图 2.2.4 所示的情况下，垂直分割是最好的选择。在三维空间中，光线击中包含该三角面片的节点的几率，与该节点包围盒的面积成正比。垂直分割生成的非空叶节点的包围盒面积比水平分割要小，因此减少了光线需要与三角面片做相交测试的几率。

一次分割是否需要完全取决于与三角面片做相交测试的代价和遍历一级 K 维树的代价的对比。通常，与三角面片做相交的代价更大，所以情愿去做一次分割，甚至可能多次。这就是表面面积启发算法（Surface Area Heuristic, SAH）的核心思路——确定最优的分割平面位置，计算每一个备选分割位置的预期代价，并选择最小代价的分割。只有当代价小于不分割时，才去执行一次分割。

不分割的代价可使用下面的等式来计算：

```
cost_nosplit = primitive_count * total_area * intersection_cost;          (2)
```

一次分割的代价可由以下公式得到：

```
left_cost = left_count * intersection_cost;
right_cost = right_count * intersection_cost; (3)
split_cost = traversal_cost + left_area * left_cost+ right_area * right_cost;
```

现在你有了一个启发式原则来找到最好的分割平面，和一个终止条件：如果你不能找到一个分割，它的代价小于不分割，就不再继续分割了。另外，最好在特定的深度停止分割，以控制内存的使用和构建时间。

虽然 SAH 算法非常好，并能产生高质量的树结构，它还是需要一些干预来生成更适合快速光线追踪的树。SAH 算法本身是不会去隔离空白空间的。想要鼓励空白空间的裁剪，就必须降低产生空白叶节点的分割的记分。

快速创建 K 维树

使用表面面积启发算法创建 K 维树是一个非常费时的过程。因为对于每一个可能的分割，你需要得到分割平面两侧的三角面片数量，每次分割必须遍历三角面片列表 $2N$ 次，而且是每个轴，则一共会有 $6N^2$ 次内部循环。因为你是在创建一棵树，所以创建过程的预期执行时间是 $O(N^2 \log N)$。对于任何现实场景，这都要花非常长的时间。但是，这个过程是可以被改进的，如 Wald 和 Harvan 在[Wald06a]所介绍的。正如我们所指出的，SAH 算法的主要瓶颈是三角面片计数。幸运的是，你能将这些费时的过程一起解决。

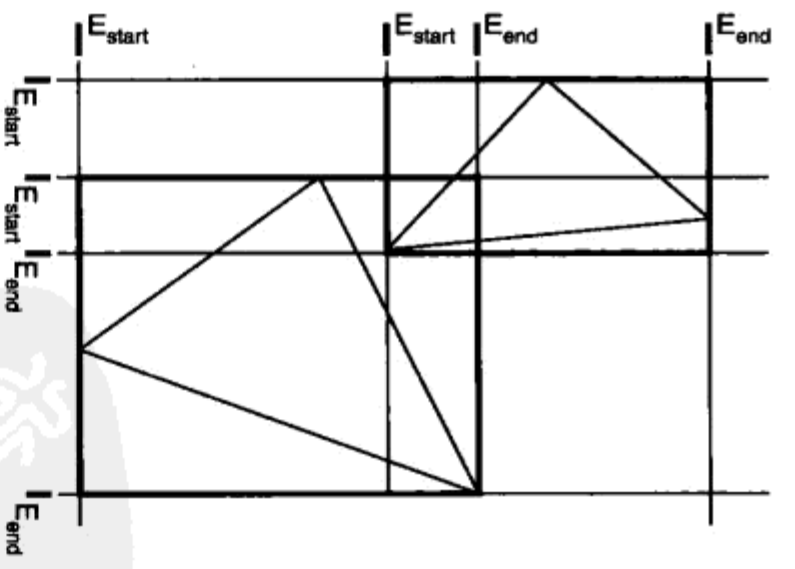


图 2.2.5　包围盒的开始和终止事件

图 2.2.5 是一个包含两个三角面片的简单场景。在水平轴上有 4 个可能的分割平面位置。在第一个位置，分割平面左边的三角面片数为 0，右边的为 2。这样，无论何时，当一个新的三角面片开始分割（从包围盒的左边），分割平

面左边的三角面计数将增加。无论何时，当一个三角面片做最后一次分割，右边的计数将减少。所以，如果你创建一个排好序的这类事件列表（开始事件：开始分割，和终止事件：最后一次分割），并从左到右地遍历这个列表，左右两边地三角面片计数就可以逐步更新。

虽然这个方法已经很好地改进了效率，但还有一些遗留问题：第一，这些事件需要对每个轴排序；第二，即在树的每个层次，这些事件需要重新排序，因为有很多三角面片已经不在列表中了，而且有一些被裁减了，引入了新的事件。

只做一次排序被证明是可能的，即在树的最顶层做一次排序。为了实现它，你需要使用一个非常复杂的数据结构。EventBox 结构（参见以下的代码）是一个包含不少于 6 个 next 指针的链表，一个面片包围盒在三维空间的每个轴向上的每个边一个指针。

对于图 2.2.5 中的场景，需要两个 EventBox，存储每个轴向上的 4 个事件。考虑左边三角面片的 EventBox：它在每个轴向上有两个 next 指针——一个指向第二个 EventBox 的开始事件；另一个指向第二个 EventBox 的结束事件。

```
struct EventBox
{
    EventBoxSide side[2];
    Primitive* prim;
};
```

第一个 EventBoxSide 包含在 x、y 和 z 轴向上的开始事件，第二个包含了终止事件。使用 EventBoxSide 对象的两个实例允许你从一边指向另外一边。每个 EventBoxSide 实例存储每个轴上的一个位置和边（side）信息（开始事件还是终止事件）。为了效率，指针和边信息被存储在一个 32 位值中。

```
struct EventBoxSide
{
    EventBoxSide* next( int axis ) { return (EventBoxSide*)
    (n[axis] & -3); }
    void next( int axis, EventBoxSide* p )
    {
    n[axis] = (n[axis] & 3) + (unsigned long)p;
    }
    int side( int axis ) { return n[axis] & 3; }
    void side( int axis, int side ) { n[axis] = (n[axis] & -3)+ side; }
    unsigned long n[3];
    float pos[3];
};
```

一旦这个链表被创建，它就可以在运行时实时更新了。这样，只需要在树的最顶层做一次排序即可。树的创建时间减少到 O (Nlog$N$)。

*K*维树遍历

K 维树的顺序遍历（从前到后，用于光线追踪）与 BSP 树遍历是一样的。首先找到包含光线起点的叶节点。对于每个节点，起点在分割平面的哪一边是确定的，继续遍历该边的分

支，将另外一边压栈。一旦找到一个叶节点，从栈中弹出一个节点，这个过程一直继续到栈为空。对于光线查询，还有一个额外的终止条件——一旦找到一个交点，就没有必要再做遍历了。

这个过程是以正确的顺序访问节点。但是，对于光线查询还需要做一些调整。比如说，你需要得到光线在当前节点确切的入口点和出口点 tmin 和 tmax。这些值是必需的，因为有些交点可能在当前节点之外，如图 2.2.6 所示。

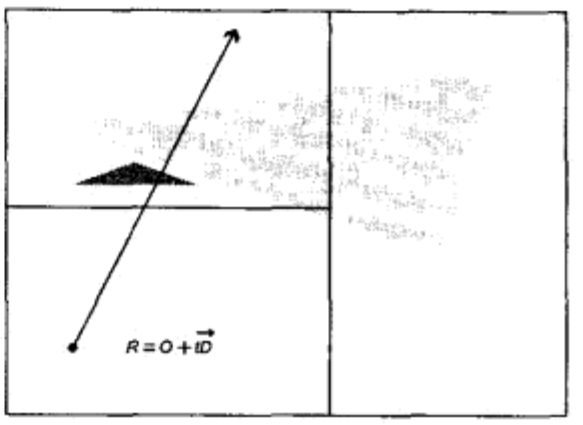


图 2.2.6　在第一个节点中错误地击中了较大的三角面片

最大的三角面片有部分在包含光线起点的节点中。如果你允许在当前节点之外做相交，与这个最大的三角面片的相交测试将会返回一个击中结果，这样遍历就会结束。较小的那个三角面片将不会被涉及。

因此，你必须留意 t_{min} 和 t_{max}。首先你必须通过用场景包围盒来裁剪光线得到 t_{min} 和 t_{max}，之后，t_{min} 和 t_{max} 需要逐步更新。

在遍历中可能出现的 3 种情况如图 2.2.7 所示。

- （图 2.2.7(a)）光线与分割平面（t_{split}）的交点在 t_{min} 和 t_{max} 之间。左边的节点先被遍历，使用 $t_{max}=t_{split}$。右边的节点被压入栈中，使用 $t_{min}=t_{split}$。
- （图 2.2.7(b)）t_{split} 在 t_{max} 之外，你只需要遍历左节点，t_{min} 和 t_{max} 都不要改动。
- （图 2.2.7(c)）t_{split} 在 t_{min} 之前，你只需要遍历右节点，t_{min} 和 t_{max} 都不要改动。

从右边击中分割平面的光线可以用相同的方式处理；唯一的不同是以上规则中的左右节点现在需要调换。

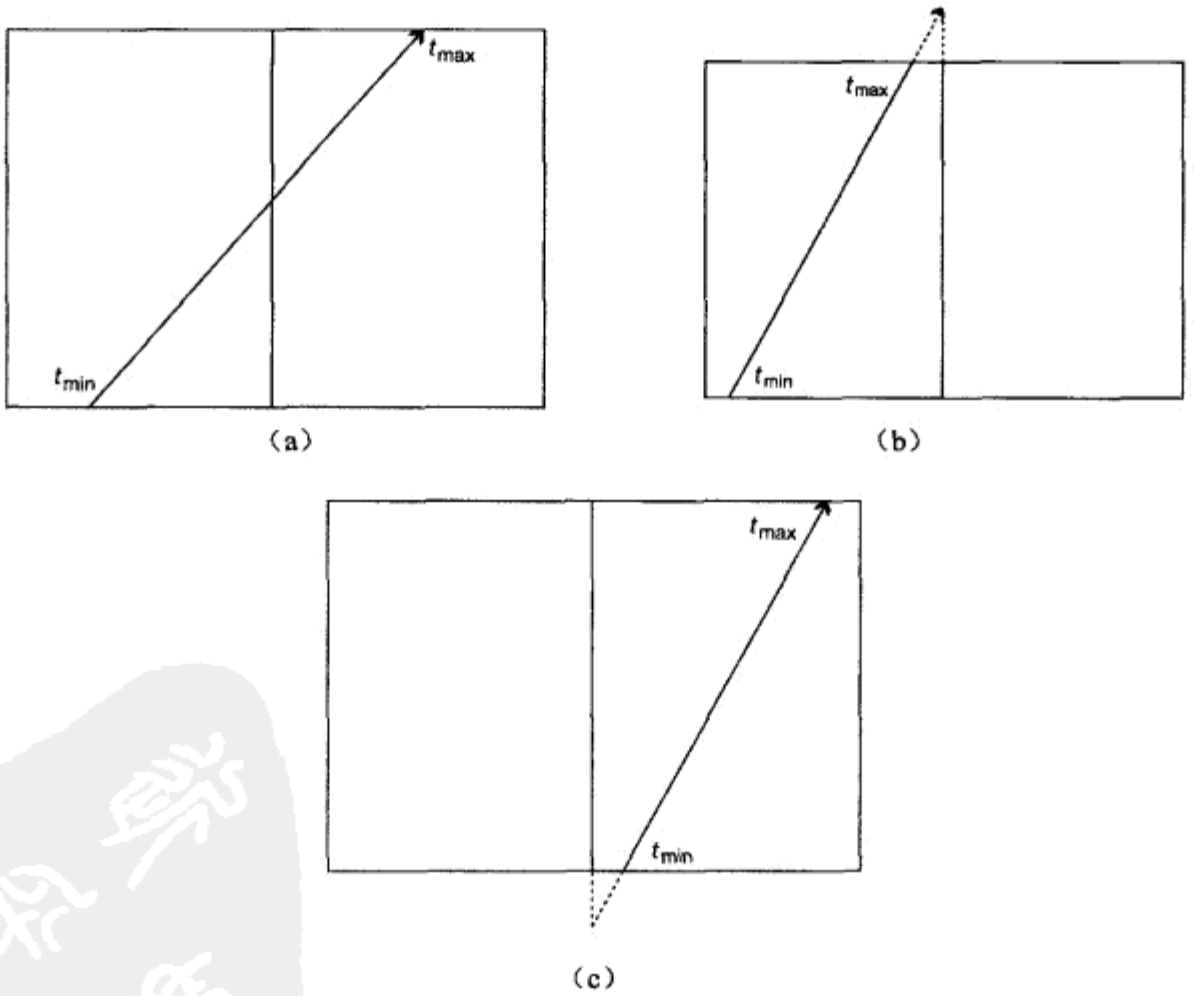


图 2.2.7　k 维树遍历的可能情况

其实现代码如下：

```
// 预计算数据
int raydir[8][3][2];
for ( int i = 0; i < 8; i++ )
{
    int rdx = i & 1;
    int rdy = (i >> 1) & 1;
    int rdz = (i >> 2) & 1;
    raydir[i][0][0] = rdx, raydir[i][0][1] = rdx ^ 1;
    raydir[i][1][0] = rdy, raydir[i][1][1] = rdy ^ 1;
    raydir[i][2][0] = rdz, raydir[i][2][1] = rdz ^ 1;
}
// 准备要遍历的数据
KdTreeNode* node = Scene::GetKdTree()->GetRoot();
vector3 O = ray.origin;
vector3 D = ray.direction;
vector3 R( 1 / ray.direction.x,
           1 / ray.direction.y,
           1 / ray.direction.z );
int oct = ((D.x < 0)?1:0) + ((D.y < 0)?2:0) + ((D.z < 0)?4:0);
int* rdir = &raydir[oct][0][0];
int stackptr = 0;

// 实际遍历
while (1)
{
    while (!node->IsLeaf())
    {
        int axis = node->GetAxis();
        KdTreeNode* front = node->GetLeft() + rdir[axis * 2];
        KdTreeNode* back = node->GetLeft() + rdir[axis * 2 + 1];
        float tsplit = (node->m_Split - O.cell[axis]) * R.cell[axis];
        node = back;
        if (tsplit < tnear) continue;
        node = front;
        if (tsplit > tfar) continue;
        stack[stackptr].tfar = tfar;
        stack[stackptr++].node = back;
        tfar = MIN( tfar, tsplit );
    }

    // 找到叶节点，处理三角面片
    int start = node->GetObjOffset();
    int count = node->GetObjCount();
    for (int i = 0; i < count; i++ ) // 交换三角面片

    // 交换式出栈
    if ((dist < tfar) || (!stackptr)) break;
    node = stack[--stackptr].node;
    tnear = tfar;
    tfar = stack[stackptr].tfar;
}
```

数组 raydir 是用来快速调换左右子节点的。数组的结构为[octant][axis][child]。该数组的每个条目包含了对于每个轴向、每个八分圆的近端节点和远端节点的偏移(相对于光线方向)。

2.2.3　动态物体

以上所述的光线查询方法都依赖于一个高质量的 K 维树，这个结构通常需要离线的预计算。场景必须是静态的，但在游戏中这基本是不可能的。这个问题没有简单的解决方案。在动态场景中做光线追踪一直是个研究热点。针对不同的需求，这里有一些可能的方案。

不能每帧更新 K 维树的主要原因是 SAH 非常费时，但是你可以使用两个树。第一个包含静态几何体，而第二个只包含动态的物体。在一个混合环境（静态和动态三角面片）中，光线追踪被实现为一个双场景遍历的过程。首先，光线先遍历包含静态场景的树，然后遍历包含动态场景的树。初看这样做效率不高，但实际上，大多数光线都不会击中动态几何体，因为这类几何体只占场景中的一小部分，因此大多数光线只是遍历一小部分空白节点，不会与任何三角面片相交。

在上述方法中，包含动态场景的树每帧都需要重建。如果使用 SAH，决定分割平面位置将是最费时的部分。如果固定分割平面位置，包含动态三角面片的树将在很短的时间内创建好。一个方法就是选择各个轴的空间中平面，这样实际上生成了一个八叉树。为动态三角面片使用一个独立树的前提是场景中的静态几何体远多于动态几何体，通常场景都是这样的。

其实我们也可以完全不使用 K 维树。其他加速结构也可以得到不错的结果，而创建时间要比 K 维树短。比如层次包围体（Bounding Volume Hierarchies，BVHS）[Wald07]、层次包围区间（Bounding Interval Hierarchy）[Wächter06]和嵌套网格[Wald06b]。

2.2.4　示例程序

光盘上的示例程序演示了上述概念。它读取一个场景文件（OBJ 格式）并以线框模式显示这个网格。一个小的动态物体也被加载。每一帧，这个动态物体都被旋转，一个小的 K 维树被创建。一个光线束包含 5 000 根光线，每第 64 根画一根光线。这些光线通过静态和动态的 K 维树追踪来决定从场景中心哪些部分是可见的。图 2.2.8 是这个示例的一个屏幕快照。注意，尽管这个图像是二维的，实际的数据是三维的，光线查询也同样是三维的。

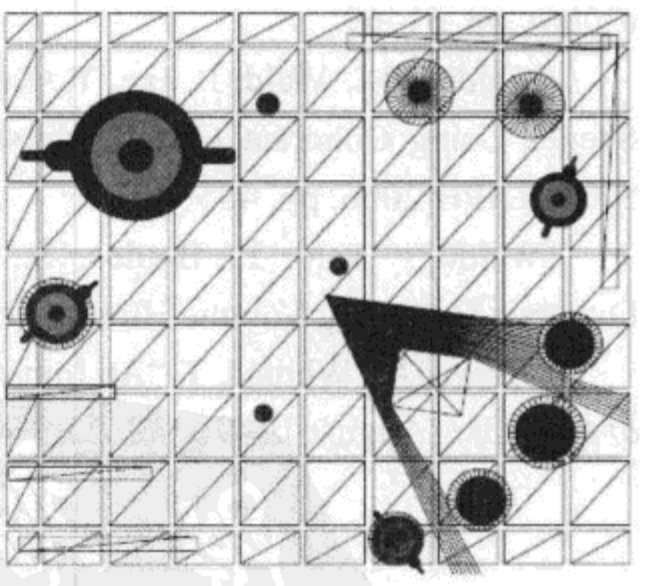


图 2.2.8　示例程序以线框模式显示网格

2.2.5 结 论

多边形环境中通用的光线查询提供了一个强大的工具，以帮助我们实现游戏中的许多功能。本节展示了使用一个基于表面面积启发算法的高质量的 K 维树是如何实现这些查询的。动态几何体存储在一个遍历较低效的 K 维树，但这个树创建时间更短。

如果实现得好，本节所示的方法可以让你每秒轻松地追踪 100 万条光线。你应该需要更多，可以考虑开发 SIMD 遍历加强包（同时遍历 4 倍数的光线）。

甚至你可能想开发使用实时光线追踪生成图像的梦幻世界，这是让你觉得计算能力永远不够用的罕见算法之一。你会喜欢光线追踪的直观性——无论你用它来生成图像还是为了其他的目的。

2.2.6 参考文献

[Appel63] Appel, A. "Some Techniques for Shading Machine Renderings of Solids," *Proceedings of the Spring Joint Computer Conference* 32 (1968), pp. 37–45.

[Havran01] Havran, V. "Heuristic Ray Shooting Algorithms," PhD thesis, Czech Technical University, Praha, Czech Republic, 2001.

[McDonald90] MacDonald, J., and Booth, K. "Heuristics for Ray Tracing using Space Subdivision," *The Visual Computer,* Vol. 6, No. 3 (June 1990), pp. 153–166.

[Wächter06] Wächter, C., and Keller, A. "Instant Ray Tracing: The Bounding Interval Hierarchy," In *Rendering Techniques 2006, Proceedings of the 17th Eurographics Symposium on Rendering* (2006), pp. 139–149.

[Wald04] Wald, I. "Realtime Ray Tracing and Interactive Global Illumination," PhD thesis, Saarland University, 2004.

[Wald06a] Wald, I., and Havran, V. "On Building Fast kD-trees for Ray Tracing, and On Doing That in O(NlogN)," *Proceedings of the 2006 IEEE Symposium on Interactive Ray Tracing* (2006), pp. 61–69.

[Wald06b] I. Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. "Ray Tracing Animated Scenes Using Coherent Grid Traversal," *ACM Transactions on Graphics,* Proceedings of ACM SIGGRAPH 2006, pp. 485–493.

[Wald07] Wald, I., Boulos, S., and Shirley, P. "Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies," *ACM Transactions on Graphics* (2007), Vol. 26, No. 1.

[Whitted79] Whitted, T. "An Improved Illumination Model for Shaded Display," *Communications of the ACM* (August 1979), Vol. 23, No. 6, pp. 343–349.

2.3　使用最远特性图的快速刚体碰撞检测

Rahul Sathe, Advanced Visual Computing,
SSG, Intel Corp.
rahul.p.sathe@intel.com

Dillon Sharlet, University of Colorado at Boulder
dillon.sharlet@colorado.edu

随着处理器和图形加速卡性能的不断增强，在实时交互图形程序和游戏中，物理已经成为一个非常重要的领域。如果没有物理，虽然可能每一帧看起来都很真实，但玩起来就会没有真实感。在真实世界里，物理现象的建模都是数学性质的，而且计算强度非常大。游戏开发者通常都试图在不牺牲视觉真实度的情况下简化这些模型，同时他们还希望改进这些模型的计算效率。计算强度最大的任务之一就是物体的碰撞检测。这个任务涉及确定两个物体是否碰撞（或者在上一步碰撞了）。如果它们碰撞了，物理模拟需要一些更多的工程量，如透深（Penetration Depth）和分开轴（Separating Axis）。

本节将尝试使用一些快速数据结构来解决这个过程涉及的一些复杂问题。我们将介绍一个新的数据结构，被称为最远特性图（farthest feature map），用来加速三角面片可能碰撞集（Potentially Colliding Set，PCS）的实时查找。这个算法就像我们以前的基于距离立方图（Cube-Map）的算法一样，只能用于凸多面体刚体，并且需要一个预处理步骤以及一个运行时步骤。

这个新方法的主要原理如下——凸刚体在彼此相距很远时的行为就像质点。但是，当它们接近时，质点近似就不够好了。这时，你必须使用接近接触面的刚体局部属性来做更多的细节分析。

2.3.1　背　景

碰撞检测可以用两种方式来建模——离散碰撞检测或者连续碰撞检测。前者在每个时间步长更新游戏中物体的位置，从而及时发现是否有两个物体在某个时刻相交了。一种离散碰撞检测的实现方法是使用递归细分层次的包围盒，轴对齐的（AABB）[Bergen97]或者是定向的（OBB）[Gottschalk96]，来找到相交的三角面片。连续碰撞检测系统使用一个可变的时间步长，并为每对可能的碰撞体修改这个时间步长，以使这两个物体

不至于相交。连续碰撞检测的建模算法一般都类似 Gilbert Johnson 和 Keerthy 发明的算法[GJK88]。请查阅参考资料，以了解两种方法基本原则的更多细节。

我们最近在研究把大量工作放到刚体预处理步骤中来做[Sathe07]。在运行时,你只需要访问距离立方图来找到碰撞。距离立方图的访问是可以用硬件加速的，但这还是有改进和优化的空间，这恰恰是这个新概念——最远特性图可以做到的。

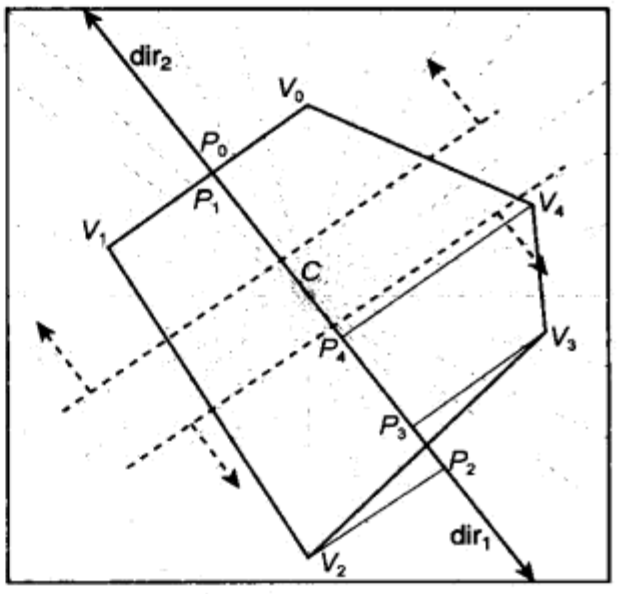
以下是本节中经常要使用的一些名词。

- 主曲率：在微分几何中，一个表面任意点的主曲率是通过那点曲线的最大及最小曲率值。这两条曲线总是成直角。
- 平均曲率：两个主曲率的平均值被称为平均曲率。
- 最佳匹配球：游戏中的典型网格是使用平面三角面片来搭建的，相邻边处有折缝，而不是使用在每条缝处都有平滑导数的连续几何体。这样曲率定义不能很有意义的应用到需要几何查询（Geometric Interrogation）的许多算法，所以你必须使用曲率的一些近似值。一个顶点的第 1 邻环（*ring-1 neighborhood*）是指这个顶点周围的三角形扇。我们考虑一个给定顶点周围的第 1 邻环，并去找一个球心在该点法线上的球。这个圆是半径等于这个三角网格平均曲率的球的大致近似。

2.3.2 预处理

这个方法使用一个类似于立方图的数据结构，也就是说这是一个方向查找，虽然存储的数据已经超过了现代图形硬件中立方图格式的容量。将物体的质心放在起点上，然后想象一个以该起点为中心的轴对齐的立方体，并且从该起点发射通过所有立方体表面中心的射线。对于每条射线，你将从中心到各顶点的线段投射到该射线上，挑出投影中的最大值，并将相应的顶点存储起来。我们把这个数据结构称为在该采样方向上的一个最远特性。

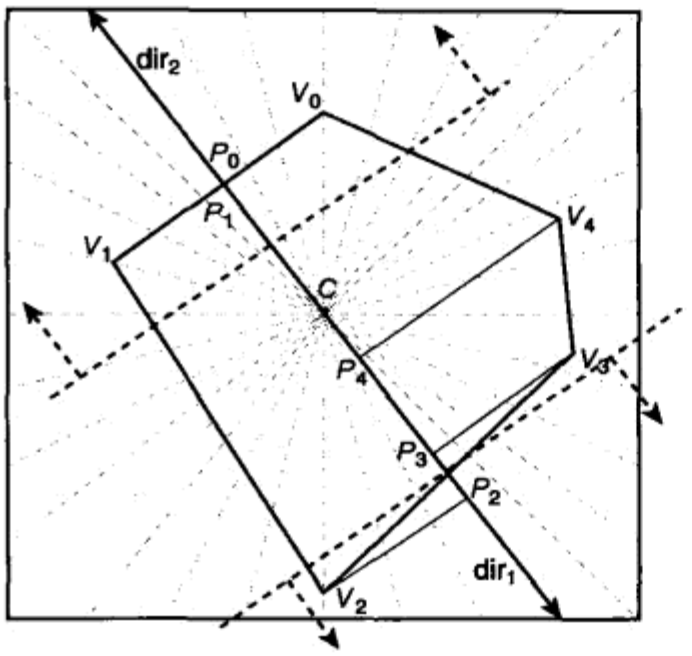
一个直观的描述最远特性图的方式是想象一个垂直于指定采样方向的平面，然后将该平面从质心沿射线方向向外移动，并保持其与采样方向垂直。这个平面将与凸多面体相交，并在该平面上形成一些多边形。如果你一直向外移动这个平面，最终这个平面将与凸面物体的一个或多个顶点相交。当你继续移动该平面远离中心，最后它将离中心足够远，以致不与凸多面体有任何相交。那时，质心到该垂直平面的距离将是该方向上最远的，而在这之前，该几何体上与平面相交的顶点就是最远特性。



$CP_n = CV_n$ 在 dir_1 和 dir_2 上的投影

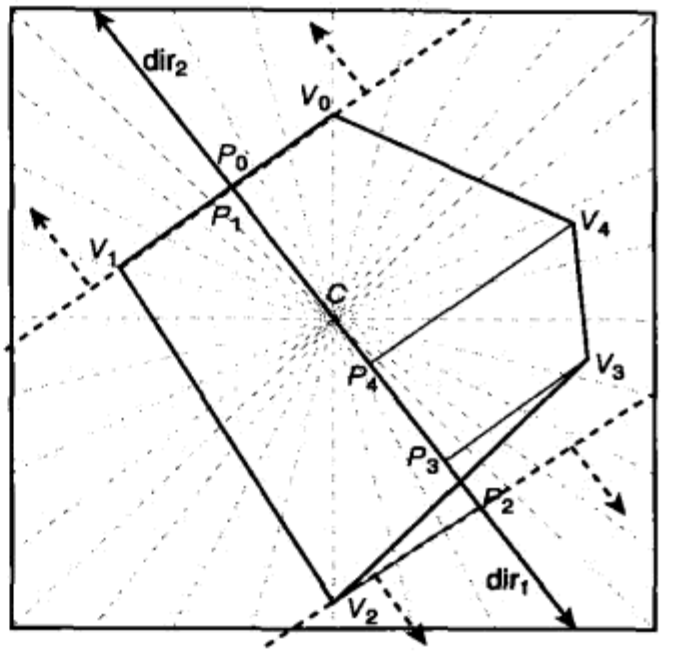
图 2.3.1 垂直与采样方向 dir1 和 dir2 的平面以虚线表示,它们向远离质心的方向移动

图 2.3.1～图 2.3.3 以二维的形式展示了这个过程。这里，粗的实线是游戏中被处理的凸多面体。这个凸多面体被定义为从 V_0 到 V_4 的一个多边形。被移动的平面以粗虚线表示。箭头表示为了找到最远特性，两个平面的移动方向，在两个采样方向上 dir_1 和 dir_2。



CP_n=CV_n 在 dir_1 和 dir_2 上的投影

图 2.3.2　平面从质心向远处移动



CP_n=CV_n 在 dir_1 和 dir_2 上的投影

V_2=dir_1 上的最远特性

V_0,V_1=dir_2 上的最远特性

图 2.3.3　在最远处垂直与采样方向 dir_1 和 dir_2 的平面。V_0 和 V_1 构成了 dir_2 上的最远特性,而 V_2 是 dir_1 方向上的最远特性

图 2.3.4 和图 2.3.5 解释了 3D 环境中最远特性图的创建过程，展示了在两个不同采样方向上的最远特性。

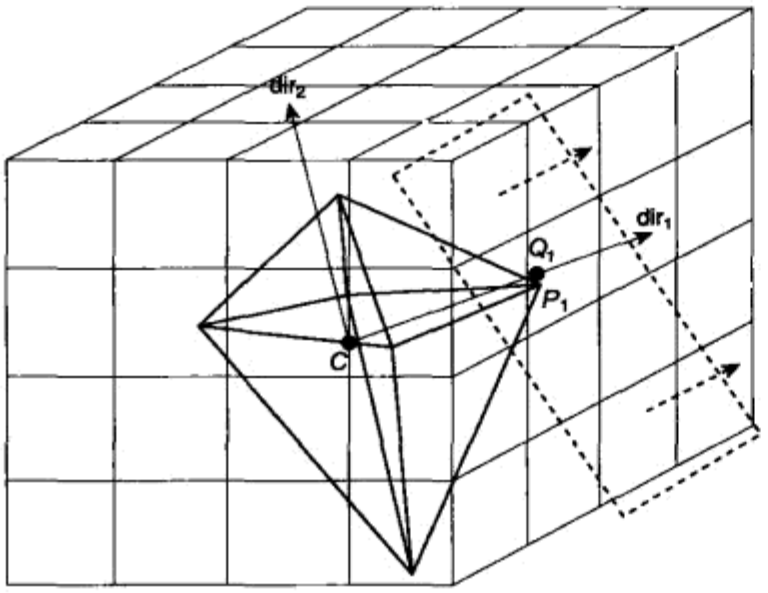


图 2.3.4　垂直于 dir_1 方向向外移动的平面现在在最终位置上。CQ_1 是 CP_1 在 dir_1 方向上的投影。P_1 和 Q_1 都在平面上

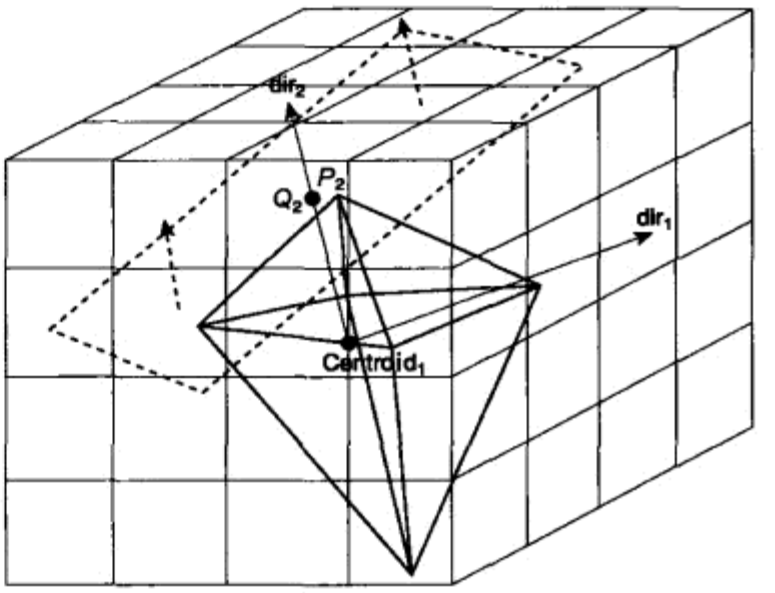


图 2.3.5　垂直于 dir_2 方向向外移动的平面现在在最终位置上。CQ_2 是 CP_2 在 dir_2 方向上的投影。P_2 和 Q_2 都在平面上

你在每个采样方向的数据（类似于立方图中的单个像素中心）都是存储在类 Node 中的。这里，你将该方向上代表最远特性的顶点的索引存储起来。如果是一个以上的顶点，你把所有的顶点索引都存储起来。在运行时，你可以从任何一个开始查询。

最远特性顶点的细节信息是分开存储的。对于每个顶点，你还需要存储它的邻居信息，如类 VertexNeighborhood 所示。对于每个顶点，你需要存储以下数据：最佳匹配球的中心、平面的方程、面的 ID 和边。存储边是用来做边边相交测试的。同样需要存储的还有给定方向上、每个最远特性上的顶点的最佳匹配球的中心和法线。

将每个顶点和每个方向（最远特性图）分开是为了优化存储。对于低面数物体，如果你对

最远特性图过度采样，你会希望存储尽可能少的信息，比如只存储 VertexNeighborhood 缓存的一些索引。

```
Class Node
{
    int *Index;
}
Class VertexNeighborhood
{
    Vector < Plane > Faces;
    Vector < DWORD > Triangles;
    Vector < Pair < DWORD, DWORD > > Edges;
    Vector3 CenterOfBestFittingSphere;
}
```

在最远特性顶点的最佳匹配球是最匹配该顶点附近的三角形扇的球。在二维情况下，这相当于最匹配该顶点第 1 邻环的圆。对于二维分段线性曲线，该顶点的第 1 邻环总是只有两个顶点。你总是能找到一个通过 3 个点的圆（除非这 3 个点在同一条直线上）。如图 2.3.6 所示。但在 3D 情况下，通常是不可能找到一个通过所有第 1 邻环顶点的球，因为第 1 邻环通常会包含比确定一个球更多的顶点，而且一般在给定顶点附近的表面也不可能是标准球面。

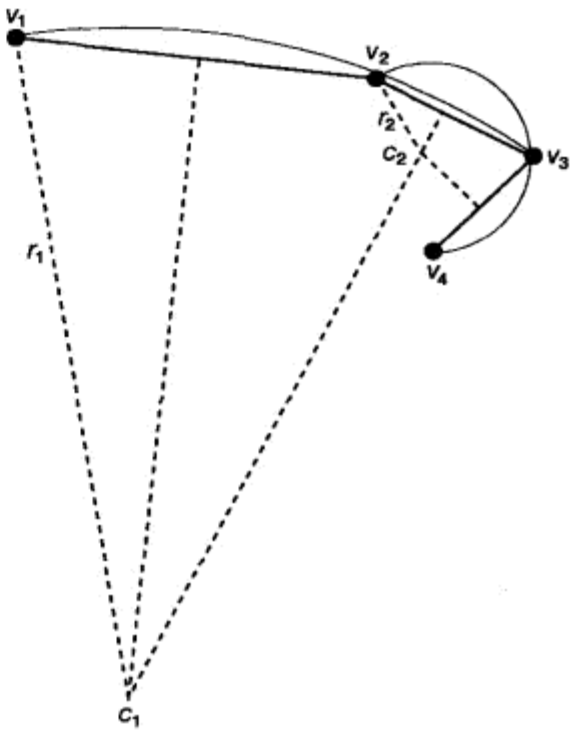


图 2.3.6 二维情况下的最佳匹配圆

2.3.3 运行时查询

在运行时，假设两个凸多面体的行为跟质点一样，当它们足够远的时候，这个假设是成立的。你必须回答的问题是多远才是足够远呢？结论是当它们的包围体彼此没有相交时，你可以把这些物体作为质点。用来近似表示物体的包围体决定了这些物体必须相距多远。包围球、轴对齐包围盒（Axis Aligned Bounding Boxes, AABB）和方向包围盒（Oriented Bounding Box, OBB）是一些很流行的近似包围体。

在运行时，可按如下步骤操作。首先，连接两个物体的质心。在两个质心连线方向上，使用预处理过程中创建的最远特性图找到这个两个物体的最远特性。从某种意义上说，这是在使用最远特性图来快速生成包围体。这与传统的包围体是不同的，因为它们在每个方向上都是不一样的。像以上这种情况，你就是在查找两个质心连线上的边界。

如果从相应的质心到最远特性顶点的连线，在两个质心连线上的投影的距离之和大于两个质心间的距离，那么你就知道这个包围体测试失败了，这两个物体都属于可能碰撞集。这样，你就不能再将这两个物体视为处于它们质心的质点了，而且你必须做进一步的分析处理。

在这种情况下，你必须查询最远特性图来找到两个物体在各自最远特性顶点的最佳匹配球的球心。然后，连接这两个最佳匹配球的球心。从某种意义上来说，这个步骤近似于认为

两个相交物体的局部是两个以最佳匹配球半径为半径的球体。需要注意的是，这个近似是局部意义上的近似。沿两个球心连线方向找到最远特性，做一个距离测试，就像刚才连接质心时做的一样。如果这个距离测试失败了，你继续在最远特性图中查找最佳匹配球，直到在两个连续步骤中找到的是同样的最远特性。这时，我们可以总结一下这个算法。

算法的伪代码如下：

```
lastC1 = lastC2 = null;
连接两个质心 C1C2
计算两个质心间的距离 C1C2
计算质心到最远特性顶点沿 C1C2 的投影距离 d1 和 d2

while (C1C2 < d1+d2 || lastC1 != C1 || lastC2 != C2)
{
    lastC1 = C1
    lastC2 = C2
    C1 = 物体 1 最远特性处的最佳匹配球的球心
    C2 = 物体 2 最远特性处的最佳匹配球的球心
    d1 = 从 C1 到最远特性在 C1C2 上的距离
    d2 = 从 C2 到最远特性在 C1C2 上的距离
}
```

最后你会得到两个最远特性分别来自两个物体和它们的第 1 邻环。对于游戏中的典型网格模型，这两个三角形扇将各自包含 6 个左右的三角面片。这时，你必须确定一个三角形扇中的某一个三角面片是否与另外一个三角形扇中任意一个三角面片相交。因此你需要做 36 对三角面片相交测试。当然，使用合适的处理器技术和编程语言，这些三角面片相交测试可以并行执行。

2.3.4　性能分析和结束语

在大多数情况下，当这两个凸多面体的顶点密度在各个方向上的分布没有太大变化时，这个算法将在 $O(1)$时间内返回。但如果相交的点在顶点更加密集的区域，这个算法将花费更长的时间。

如果能看到这个算法如何拓展到凹面物体，那将非常有趣。而如今，图形硬件的通用可编程性不断进步，我们非常期待看到谁能开发出一个灵活的立方图。这个立方图能存储更多的信息，而不仅仅是固定格式的贴图信息。采样自适应的立方图可节省很多存储空间，但会增加查找代价（译者注：采样自适应意为有些采样方向只有一个顶点，但有一些有多个，存储时各个采样方向上的占用空间是不同的）。如果有一个聪明的编码方式来解决这个问题，那就太好了。

2.3.5　致　谢

我们要感谢 Intel 高级图像计算组的管理团队，允许我们从事这项研究。同样，我们要感谢我们的合作者 Adam Lake 和 Oliver Heim，感谢他们在这项研究多个阶段给予的帮助。

2.3.6 参考文献

[Bergen97] Van den Bergen, G. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees," *Journal of Graphics Tools,* Vol. 2, No. 4, pp. 1–14, 1997.

[GJK88] Gilbert, E.G., Johnson, D.W., and Keerthi, S.S. "A Fast Procedure for Computing the Distance Between Objects in 3 Dimensional Space," *IEEE Journal of Robotics and Automation,* Vol. RA-4, pp. 193–203, 1988.

[Gottschalk96] Gottschalk, S., Lin, M.C., and Manocha, D. "OBBTree: A Hierarchical Structure for Rapid Interference Detection," *Proc. SIGGRAPH 1996,* pp. 171–180.

[Sathe07] Sathe, Rahul. "Collision Detection Shader Using Cube-Maps," *ShaderX^5 Advanced Rendering Techniques,* Charles River Media, 2007.

2.4　使用投影空间来提高几何计算精度

Krzysztof Kluczek, Gda´nsk University
of Technology
krzych82@poczta.onet.pl

在建模、渲染、物理和AI模块中用到的许多几何算法都依赖于点线测试，在三维空间中则是点面测试。当需要检测共线和共面时，计算的有限精度就会导致问题。这类问题的一个常用的解决方案是设置一个最小距离（epsilon），低于这个值时就被认为是共线或者共面。这种方案只解决了部分问题，因为当不共线的点线的距离接近最小距离（epsilon）时，可能被错误地检测成共线，因此降低了算法的整体健壮性。

为了创建在任何情况下都正确、有效、健壮的几何算法，你不能使用任何会截断中间结果的操作（比如一些浮点数操作）。截断会导致信息丢失，虽然某些情况下要获得结果，截断是必须的。这样的话，只有使用基于整数的数学运算，才能保证截断不会发生。直接使用整数笛卡儿坐标表示点是行不通的，因为端点为整数坐标的两条线段，其交点不一定是整数坐标。使用有理数能解决这个问题，但是这需要为每个向量存储 6 个整数（每个分量有分子、分母），而且在这种向量上的高效地运算实现在三维空间中并不直观。使用投影空间能将每个向量减少到 4 个整数。在向量上增加一个维度，但向量的每个分量只使用一个整数。这将在本节中进行详细的解释。

2.4.1　投影空间

投影空间的概念对于理解下面展示的算法至关重要。$\boldsymbol{RP}^2$ 投影空间可用空间中所有通过[0, 0, 0]点的直线来描述，如图 2.4.1 所示。每条线都可用 $\boldsymbol{R}^3$ 空间中除[0, 0, 0]外的一个在线上的点来唯一地表示，这样你可以使用$[x, y, z]$坐标向量来标示在这个空间中的每个元素，其中$[x, y, z] \neq [0, 0, 0]$。要注意的是，如果向量 $\boldsymbol{P}$ 和 $\boldsymbol{Q}$ 满足 $\boldsymbol{P} = \boldsymbol{Q}c$，其中 $c \neq 0$，则它们表示的是同一条直线。使用$[x,y,z]$向量来表示 $\boldsymbol{RP}^2$ 空间中的元素被称为

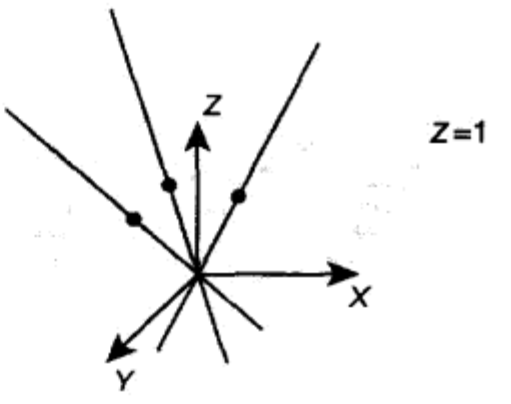


图 2.4.1　$\boldsymbol{RP}^2$ 投影空间是由所有 R^3 空间中通过原点的直线组成的。每条线都可用 R^3 空间中除[0,0,0]外的一个点来唯一地表示，每条直线都会穿过 $Z=1$ 平面刚好一次，除非这条直线是平行于这个平面的

齐次坐标表示法（homogeneous coordinate）。注意 RP^2 空间中的元素都是穿过 R^3 空间原点的直线，我们可以在 R^3 空间中定义一个 $z=1$ 平面。每条直线都会穿过这个 $z=1$ 平面刚好一次，除非这条直线是平行于这个平面的。交点可由直线表达式算出，在直线穿过原点的情况下，交点就是$[x/z, y/z, 1]$，其中$[x, y, z]$是直线上除$[0, 0, 0]$点外的任意一点。因此，你可以轻松地将 R^2 中的点和 RP^2 空间中的元素关联起来。R^2 空间中的任意点$[s, t]$可表示 RP^2 空间中的元素$[s, t, 1]$，或者任意的$[s, z, tz, z]$，其中 $z\neq0$。而表示 RP^2 空间中的元素的任意向量$[x, y, z]$可表示 R^2 空间中的$[x/z, y/z]$点，其中 $z\neq0$。这种表示法可以轻松拓展到更高维度的空间，RP^n 空间中的$[p1, ..., pn, w]$与 R^n 空间的$[p1/w, ..., pn / w]$相关。

本精粹关注于使用 RP^2 投影空间来表示 R^2 空间以及 RP^3 投影空间来表示 R^3 空间。本精粹的印刷部分主要是介绍使用 RP^2 投影空间来实现 R^2 空间的操作。因为在 RP^2 投影空间中的一个点可由一个 3 个分量的向量来表示，在这种数据上的运算相对于用来表示 RP^3 投影空间点的 4 个分量的向量运算更容易想象和理解。在随书附带的 CD-ROM 上有一个单独的文档，介绍将 RP^2 空间中的这些概念拓展至 RP^3 空间上，以满足三维数据的运算。

2.4.2 R^2 空间中的基本对象

在二维空间 R^2 中，你会关注两种对象：点和直线，这两种基本对象让你可以定义更复杂的结构。将 R^2 空间中的有向直线定义为一条有特定延伸方向的线，这样，针对直线的延伸方向，你可以定义这条直线的左右两边。有向直线定义中的方向属性对于高效的多边形表示是至关重要的，这样我们才可以假设多边形内部是在有向直线的确定的一边。如果是一个凸多边形，可以使用一个有向直线的序列来定义其边界，以上假设可以使得在凸多边形上的运算非常高效。就像凸多边形一样，许多 R^2 空间中的对象可以用点和直线来表示，比如线段和射线，所以仅仅集中注意力于这两类对象是可靠的。

2.4.3 RP^2 空间中的点和直线

就如开篇时所述，一个在 R^2 空间中的点$[s, t]$可由 RP^2 空间中的元素$[s, t, 1]$来表示，所以从 R^2 空间到 RP^2 空间的转换是相当直观的。因为缩放 RP^2 空间中的向量不会影响它的意义，所以点$[s, t]$可用任意的向量$[sz, tz, z]$来表示，其中 $z\neq0$。为了简化将来的运算，我们假设 $z>0$。如果 $z<0$，只需将整个向量乘以−1 即可。注意 $z=0$ 的向量不是 R^2 空间中的有效点。要获得把 RP^2 投影空间的向量转换为 R^2 空间点的公式，可以使用同样的规则。每个 RP^2 空间的向量$[x, y, z]$表示 R^2 中的点$[x/z, y/z]$。这类似于透视投影于 $z=1$ 平面上，而投影中心在原点。因此你可以认为 RP^2 空间基本上与 R^3 空间一样，但记住向量$[x, y, z]$的值并不重要，而是它在 $z=1$ 平面上的透视投影代表了 RP^2 空间中的点$[x/z, y/z]$。你可以看到这非常类似于有理数坐标表示法，只是使用一个公共的分母。把点定义为 RP^2 空间中的向量会为你在这些点上的运算提供很有效的工具。用来在 R^2 空间和 RP^2 空间之间做变换的函数如式 2.4.1 和式 2.4.2。注意这些函数是在 R^3 空间的向量上做操作的，而不是直接在 RP^2 空间的元素上操作，但这些 R^3

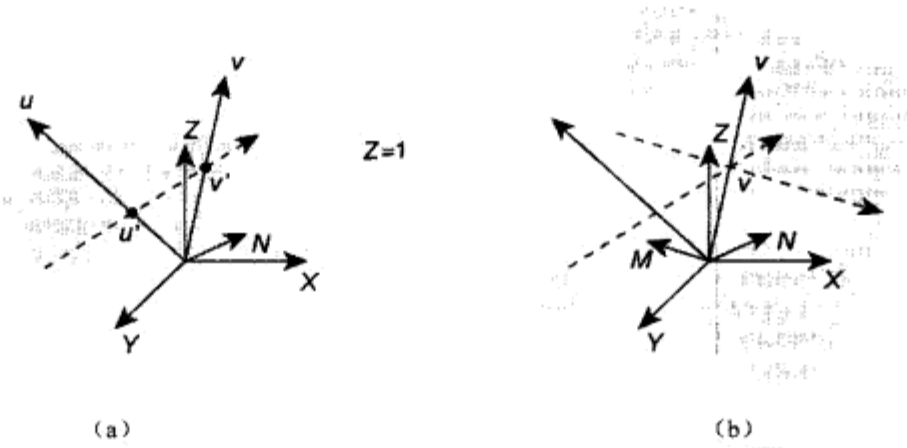
空间的向量都代表了 RP^2 空间中的元素。

$$f:R^2 \to R^3, f([s,t])=[s,t,1] \tag{2.4.1}$$

$$g:R^3_{z\neq 0} \to R^2, g([x,y,z])=[x/z, y/z] \tag{2.4.2}$$

R^2 空间中另一个具有特殊意义的对象是直线。就像点对应于向量在 z=1 平面上的透视投影，在 RP^2 空间中的直线是平面在 $z=1$ 平面上的投影。一个平面的透视投影为一条直线的唯一可能就是这个平面穿过投影中心，因此，所有用来定义直线的平面都必须穿过[0, 0, 0]点。实际上，根据使用的 RP^2 空间的定义，你也不能定义一个不穿过原点的平面，因为 RP^2 空间的元素都是一些穿过圆点的直线，所以你定义的每个平面都必须由这些直线组成。$z=1$ 平面是唯一一个例外，因为它是用来做 RP^2 空间到 R^2 空间的投影。考虑到每个平面都是穿过原点的，直线是 RP^2 空间平面在 $z=1$ 平面的投影这个定义可以被简化。被定义的直线就是给定平面与 $z=1$ 平面的交线。再一次，因为每个平面都通过原点，所以存储它的法线向量就足够了。你不需要将这个法线向量归一化，实际上通常它也不可能是归一化的，因为你将只使用整数坐标来表示这个法线向量。

之前直线的定义也可以拓展到有向直线的定义，只需要简单地将定义中平面法线向量的方向考虑进去就可以了。平面的正半边可以定义为包含所有与法线点乘得到正值向量的半边空间。简单来说，平面的负半边是法线向量指向的那一边。同样，你也可以定义这个平面的负半边。因为由平面表示的直线是这个平面与 $z=1$ 平面的交线，这个平面将 z=1 平面分割为两个半平面，一个处于平面的正半空间；另一个处于负半空间。你可以将处于正半边的那一半平面称为该有向直线的右边；另一半为左边。这样，也就派生出有向直线的方向。我们也就给出了使用 RP^2 空间的平面来表示有向直线的完整定义。图 2.4.2（a）包含了一个使用 RP^3 空间中的平面来表示一条有向直线的例子。



（a）　　（b）

图 2.4.2　RP^2 空间中的点和直线（a）一条通过一对点的直线：向量 u 和 v 表示了点 u'和 v'，直线是用法线为 N 的平面表示的；（b）一对直线的相交：N 和 M 是表示直线的平面的法线，它们叉乘得到的向量代表了交点

2.4.4　在 RP^2 空间中的基本运算

对于点和直线，有 3 种基本运算。给定一对有序点，你可以找到一条有向直线按指定顺序通过这两个点。给定一个点和一条有向直线，你可以确定这个点在直线的哪边或者是在直

线上。最后，给定一对有向直线，你可以找到它们的交点。所有这些操作在 RP^2 空间中都是非常简单且直观的。

给定 RP^2 空间中的两个向量表示两个点，你可以找到一条通过这两个点的有向直线。由于 RP^2 空间的本质，这两个向量都必须处于表示你要找的直线的表面上，否则这两个向量的透视投影点将不在这个平面上，也就不可能在该平面与 $z=1$ 平面的交线上。给定一对这样的向量，你只要将它们叉乘，就可以得到该平面的法线。一条通过两点直线的例子在图 2.4.2a 中演示。以刚才方式计算出的法线可正确地定义你需要寻找的 RP^2 空间中的平面。这条直线的方向取决于叉乘的顺序，因为调换叉乘的两个向量会反转结果。因为要得到的直线的方向必须从第一个向量表示的点指向第二个，所以你必须正确地计算法线，以保证正确地定义了 RP^2 空间中该平面的正半空间，从而正确定义这条有向直线的右半边。因为这取决于所使用的坐标系是左手系还是右手系，所以在实现时注意检查这个运算的结果，必要时反转两个向量的叉乘顺序。

另一个具有特殊意义的运算是点线测试。该测试检查一个给定点是否在一条给定有向直线上。就如之前所指出的一样，当一个顶点处于这条直线上时，RP^2 空间中相应的向量将处于表示该直线的平面上。因此该向量垂直于这个平面的法线。这样点乘将是这个测试最完美的工具。如果表示该点的向量与平面的法线的点乘结果为 0，则该点在直线上。因为平面的法线定义了该有向直线的左边和右边，如果这个点不在直线上，你可以使用点乘结果的正负号来确定点在该直线的哪一边。由于之前我们假设了向量坐标中 $z>0$，当点在该直线右边时，点乘的结果将总是正的，而在左边时结果将总是负值。

最后一个重要的基本运算是找到一对直线的交点。再一次，表示你所要寻找的点的向量必须垂直于那两个代表直线的平面的法线，如图 2.4.2（b）所示。通过将两个法线叉乘，就可以得到 RP^2 空间中代表交点的向量。为保证 $z>0$，当 $z<0$ 时，将整个向量乘以−1 即可。如果所给的直线是平行的，所得到的向量 $z=0$，这就表示这两条直线的交点不存在。

2.4.5 在 RP^2 空间中使用整数坐标进行精确的几何运算

之前介绍的所有运算都主要基于基本的比较运算和向量的点乘及叉乘。点乘和叉乘都需要实现加法、减法和乘法操作。鉴于此，如果只是使用整数坐标的向量，所有这些运算都不会产生分数，也不会产生一个包含分数分量的向量。因此，如果只使用整数坐标的点，将确保这些运算返回精确的结果，不会有浮点数计算固有的数值误差。不幸的是，要使这个系统有用，你必须让它能正确接受浮点数据输入。因为现有的大部分系统，如建模工具包，只能提供浮点数据。在大多数情况下，你可以选择一个确定的有限精度（比如，10^{-3}），将数据放大，凑整为一个最接近的整数，在进行完所有必要的操作之后，再将数据缩小回去。注意，凑整操作只是影响到输入点的位置，但并不影响进一步的运算，尤其是那些需要确定一个点是否在一条直线上的测试。

2.4.6 在 RP^2 空间中几何运算的数值范围限制

因为整数乘法在本节描述的运算中经常用到，你必须注意一个整数可以表示的范围

是有限制的。连续的乘法使用来运算的数值很快就变得巨大。为了估计每个运算阶段可能用到的数值范围，可以使用一个对称的估计范围$[-a, a]$。这个范围定义了在一个给定的阶段最差情况下可能达到的最小和最大值。如果知道输入值的范围，你可以很容易地估计像加法、减法和乘法之类操作的结果范围。使用这个结果范围，一些复杂操作像点乘和叉乘的结果范围也可以被估计出来。如果有两个值，范围分别为$[-a, a]$和$[-b, b]$，你可以确定它们的和必定在$[-a-b, a+b]$之间。因为估计的范围是对称的，不同的给定输入值得到的结果范围都是一样的$[-a-b, a+b]$。类似地，两个范围为$[-a, a]$和$[-b, b]$的值相乘的结果范围可被证明为$[-ab, ab]$。知道了这个，你就可以分析每一步运算所需要的数值范围了。

因为连续对点运算会导致连续的乘法，从而生成大数，你可以限制对 3 类对象的运算，这对于大多数几何运算已经足够了。第一类是作为输入数据的点。这样的点具有明确给定 $\boldsymbol{R}^2$ 空间中的坐标（比如从数字内容创建程序中导入的数据），因此可以轻易地预测它基于游戏关卡大小的坐标范围和表示它所需要的精度。这个范围和精度被用来在数据导入过程中放缩模型坐标。虽然可能需要增大输入点坐标的允许范围来获得更大的游戏关卡或者更高的精度，这个范围始终是影响所有运算中数值范围的主要因素。数值范围分析的目的在于在算法效率（依赖于中间值的范围）和输入数据的范围和精度间找到一个平衡，因为大数不仅需要更多的存储空间，而且需要更多的处理器资源。

第二类对象是有向直线，由计算穿过输入点的直线得到。输入点的坐标是明确给出的，所以表示这类有向直线的平面的法线向量只使用一次叉乘就可以得到。用户可以轻易地估计出该平面法线向量各分量的范围。

最后一类要使用的对象是前一类直线的交点。在 $\boldsymbol{RP}^2$ 空间中表示这类点的向量是由平面法线的叉乘所得，所以这类向量是两次连续叉乘的结果。实际上，这类向量的范围要大于表示其他类对象的向量。

由于使用多类对象数值范围会不断增大，你应该尽量只使用这 3 类对象（输入点、通过输入点的直线以及这类直线的交点）。应该避免使用会产生这 3 类对象之外对象的运算，比如定义一条穿过交点的有向直线，因为这会让你的数值增大到超出控制。幸运的是，许多几何算法，包括构造实体几何算法（Constructive Solid Geometry, CSG），都可以不使用产生 3 类对象之外对象的运算来实现。如果由于某些原因，一个算法不能仅仅依靠这 3 类对象实现，你可以引入新的对象，比如穿过交点的直线。但是要注意产生这类对象的运算所需的数值范围，因为这个数值范围将随着对象的数量指数级增长，并且你需要注意对每个新引入的类都要估计其范围。

表 2.4.1 展示了以上所述操作的数值范围分析。你可以看到连续的操作让结果的数值范围快速膨胀。表 2.4.2 展示了运算数位数不同的情况下，输入点坐标的最大范围和进一步操作结果的范围。即使是使用 64 位整数，允许的输入点坐标范围也仅仅是$[-20\,936, 20\,936]$。在一些程序中，$[-20\,936, 20\,936]$这个范围可能是足够的，在这种情况下，以上算法就可以方便地实现而不需要更多的工作，这样甚至可以在缺乏浮点数支持的设备上（比如手机）进行几何运算。但是，在很多程序中，这个范围对于所需的精度和游戏关卡大小还是不够。对于这类程序，需要长整型来解决问题。

表 2.4.1　　二维几何体上投影空间操作结果的数值范围

对　象	等　式	坐　标	范　围
输入点（P）	$P=[x, y, 1]$	x, y z	$[-n, n]$ 1
平面法线（N）	$N=P_1 \times P_2$	x, y z	$[-2n, 2n]$ $[-2n^2, 2n^2]$
交点（Q）	$Q=N_1 \times N_2$	x, y z	$[-8n^3, 8n3]$ $[-8n^2, 8n^2]$
直线与输入点测试结果	$N \cdot P$		$[-6n^2, 6n^2]$
直线与交点测试结果	$N \cdot Q$		$[-48n^4, 48n^4]$

表 2.4.2　　使用不同长度的整数表示，每一步允许的最大数值

范　围	16 位	32 位	64 位
$[-n, n]$	5	81	20 936
$[-2n, 2n]$	10	162	41 872
$[-2n^2, 2n^2]$	50	13 122	876 632 192
$[-8n^2, 8n^2]$	200	52 488	3 506 528 768
$[-8n^3, 8n^3]$	1 000	4 251 528	73 412 686 286 848
$[-48n^4, 48n^4]$	30 000	2 066 242 608	9 221 808 000 608 698 368
最大值	32 767	2 147 483 647	9 223 372 036 854 775 807

对于一个给定程序，要找到表示整数的长度，必须确定游戏关卡的大小以及所需要的精度。有了这些信息，就可以推导出所需的整数坐标范围。比如说，如果所需的工作空间大小为[－1 000, 1 000]，要求精度为 0.01。在将输入点数据放大后，用于进一步操作的输入坐标范围为[－100 000, 100 000]。然后，你就可以找到合理的范围，以进行几何操作不会导致溢出。这样就可以得到所需整数的位数（记住要记入整数的标识位）。

虽然可能的最大范围需要一个很大位数的整数表示（经常超过 64 位），但这么大的数值可能只是在最复杂的情况下才需要；大多数基本运算可以使用较短的整数来进行。为找到每一种情况下所需的整数位数，每种情况都要做一个与之前类似的分析。使用长整数运算，表 2.4.1 所给的估计范围也是很有用的，因为这些估计可以用来找到不同输入点坐标范围下所需的整数范围。

2.4.7　RP^2 空间运算的例子程序

为证明 RP^2 空间中几何运算的有用性，我们将展示一个简单的算法来进行多边形上的布尔运算。为了描述简单，我们假设多边形都是凸多边形，这样所有输入多边形的集合都是一个凸多边形集的子集。一个凸多边形可以用一个由一些边所围成的圈来表示，每条边由它的两个顶点以及它的方向来定义，每条边的方向都应该使多边形的内部区域处于该边的右边。这个例子也假设任意一个多边形都不存在 3 点共线的情况。

在凸多边形的布尔运算中，一个很有用的基本运算就是使用一条有向线段来分割一个多边形。这个运算可用以下的算法实现，图解见图 2.4.3。

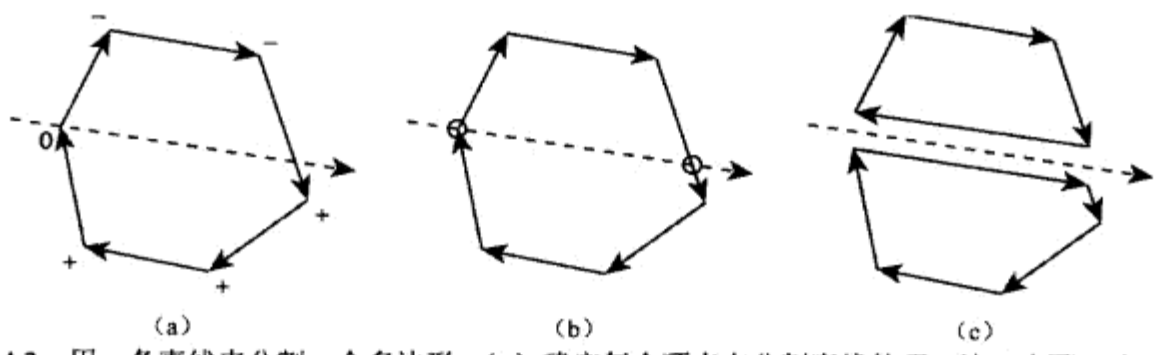


图 2.4.3　用一条直线来分割一个多边形：（a）确定每个顶点在分割直线的哪一边（步骤(1)和(2)）；（b）找到交点并分割边（步骤(3)）；（c）将两边的边分开，并合拢两个新的多边形（步骤(4)和(5)）

（1）对于每个多边形顶点，确定它在分割直线的哪一边还是在直线上。

（2）如果分割直线的其中一边（左边或者右边）没有任何顶点，则算法可以停止。这条分割直线不会与多边形相交。

（3）对于起点和终点分别在分割直线的左右两边的多边形边，分割之。用两条新边来替代被分割的边。分割的中间点为边与分割直线的交点。

（4）使用所有在分割直线右边的边来创建一个新的多边形。增加一条新边来合拢这个多边形，这条新边处于分割直线上的顶点之间（如果初始的多边形是正确的，应该刚好有两个这样的顶点），方向就是分割平面的方向。

（5）类似地，使用处于分割直线左边的边来创建第二个多边形。增加一条在分割直线上但方向相反的新边来合拢这个多边形（反转在 RP^2 空间中表示该分割直线的平面的法线即可）。

值得注意的是，这个算法只使用了之前所述的 RP^2 空间中的 3 类基本对象。在步骤（3）中，多边形添加了新的交点，在步骤（3）、步骤（4）和步骤（5）中使用了已经存在的有向直线来定义多边形的边。这个算法没有使用存在的点来创建新的直线，所以初始多边形的顶点是输入数据还是交点根本不重要。

实际上在这两类顶点上，这个算法都能完美地执行。同样值得注意的是：在步骤（3）中，边的分割点是一对直线的交点。如果使用基于浮点数的几何运算，这个点需要使用边上的点到分割直线距离的线性插值计算来得到。但使用 RP^2 空间中的基于整数的几何运算，就可以回避点线距离计算。

虽然用直线分割多边形本身就很有用处，但更重要的是，用户可以使用这个运算来实现多边形上的布尔集合运算。以下算法演示了进行这些运算的初始步骤，图解见图 2.4.4。

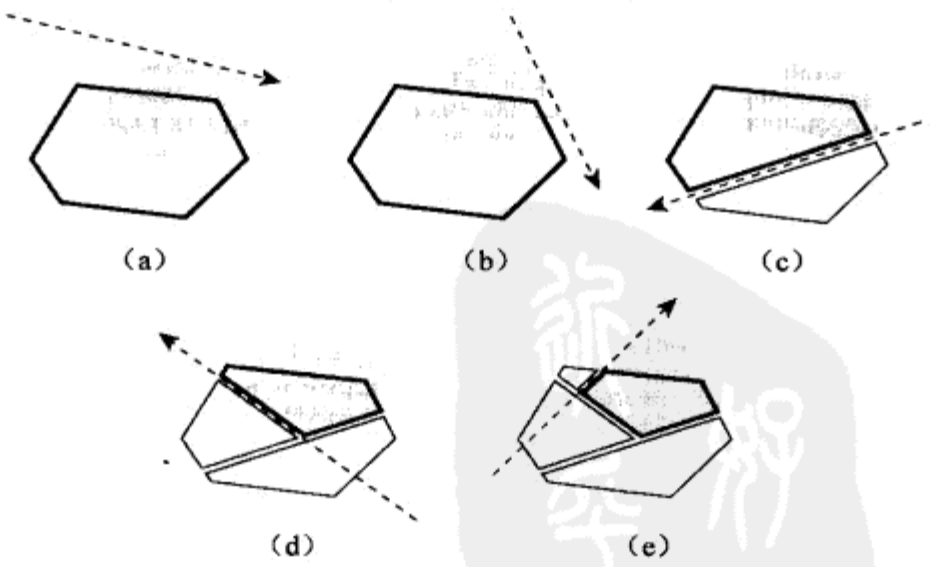


图 2.4.4　找到两个多边形的交集。步骤（a）～步骤（c）展示了在分割算法步骤（3）中每次分割所生成的多边形。初始多边形 ***B*** 以灰色填充，当前多边形 ***C*** 以粗线表示，在集合 ***outA*** 中的多边形以普通线表示

1．***A***、***B*** 为给定的多边形。

2．***C*** 是 ***A*** 的一个副本，用来找出与 ***B*** 的交集。

3．对于 ***B*** 的每条边，使用与该边相关的有向直线切割多边形 ***C***。把在分割直线左侧的之前 C 的部分加入到 ***outA*** 集合中，并且将在分割直线右侧的部分作为新的多边形 ***C***。如果 ***C*** 都在直线的左侧，则停止，***A*** 和 ***B*** 没有交集。

4．重复步骤（3），直到 ***B*** 的所有边都遍历过（除非算法在步骤（3）停止了）。

当算法停止时，如果证明 ***A*** 和 ***B*** 相交（算法没有在步骤（3）停止），最终的多边形 *C* 就是初始多边形 ***A*** 和 ***B*** 的交集，而 ***outA*** 包含了初始多边形 ***A*** 不在 ***B*** 中的部分。在 ***A*** 上的基本的布尔运算可如下表示，图解见图 2.4.5。

$$A \cup B = out\,A \cup B$$

$$A \cap B = C$$

$$A \backslash B = outA$$

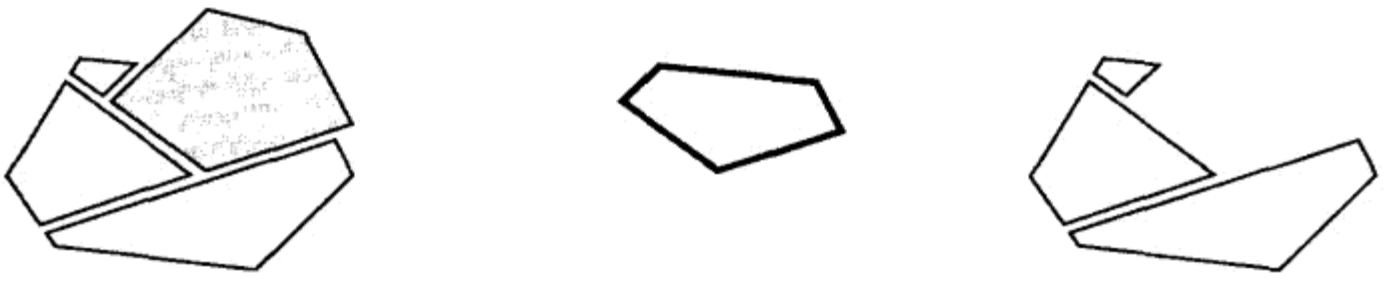


图 2.4.5　在多边形上的布尔运算：（a）多边形 ***A*** 和 ***B*** 的和即 ***outA*** 和 ***B*** 的和，（b）***A*** 和 ***B*** 的交集为多边形 C，（c）***A*** 和 ***B*** 的异集为集合 ***outA***

在以上这些公式中，∪是两个多边形的并集，∩是它们的交集，而\是它们的异集。当在多边形集合上运算时，∪是两个多边形集的并集，而当在集合 ***outA*** 中的多边形与 ***B*** 不重叠时，并集 $out\,A \cup B$ 可由简单将 ***B*** 加入 ***outA*** 中得出。

因为这个算法完全是基于 RP^2 空间中的整数运算的，它已被证明可用于各种可能的有效输入数据。当然，在要用到布尔运算算法的时候也不太可能是这种情况。但是这个算法不能用于所有的应用程序，因为它有可能在生成的多边形网格中产生 T 型相交（T-intersections）。为解决这个问题，这个算法需要加入连通信息，以拓展到网格数据上执行——比如说，使用一个半边数据结构（half-edge data structure）。这个拓展已经超出了本节的范围，就不在这里展示了——这个算法仅仅是用来作为 RP^2 空间中的整数运算的概念证明。

2.4.8　拓展到第三维

之前的讨论介绍了在二维空间中使用投影空间来进行高效且无误差的几何运算。而拓展到第三维度，对于大多数当代游戏是至关重要的。这个拓展也是相当直观的，只是需要使用四维的向量以及在此之后的运算。随书附带的 CD-ROM 上有一个文档讨论了在三维空间上使用 RP^3 投影空间。

2.4.9　结　论

许多几何算法都有因为使用的数值表示而引起的凑整和精度丢失的问题，这些问题会导致这些算法在处理共线或共面点时出现错误。这在构造实体几何算法（CSG）中尤其明显，因为共线和共面情况会导致很多实现下的严重问题。本节所展示的投影空间中的运算使得大多数这类算法的实现可在所有有效输入数据上正确地执行，代价是需要额外的计算能力来做大整数上的运算。但为了保证基于投影空间运算的算法的健壮性，这个代价是值得的。另外，因为 64 位处理器以及相应的指令集扩展会变得越来越通用，长整数数学被高效地实现，这些算法所需的额外计算代价也不会很大。

用于数字内容创建工具或引擎内置关卡编辑器的健壮的 CSG 算法会让美工和“mod”开发人员有更多自由发挥的空间，并且节省他们的开发时间。因为如果 CSG 算法不可行，他们就不得不另寻解决方法。同样，当一个 CSG 算法用于游戏本身时，比如要允许玩家以各种可能的方式与环境交互而不用担心由于 CSG 失败而产生灾难性的 BUG 时，它的健壮性也是至关重要的。

2.4.10　参考文献

[Hollash91] Hollasch, Steven R. “2.1 Vector Operations and Points in Four-Dimensional Space,” in “Four-Space Visualization of 4D Objects,” Chapter 2.

[Young02] Young, Thomas. “Using Vector Fractions for Exact Geometry,” *Game Programming Gems 3,* pp. 160–169.

2.5 使用 XenoCollide 算法简化复杂的碰撞

Gary Snethen, Crystal Dynamics
gary@snethen.com

本精粹将介绍一种可用于多种凸多面体的算法，包括长方体、球体、椭圆体、胶囊体、圆柱体、圆锥体、锥体、截头锥体、角柱体、扇形柱体、凸多胞形和其他一些常用的形体。这个算法能检测重合并且可以得到接触点、法线以及动态刚体的穿透深度。随书附带的光盘提供了一个可用的实现，以及一个简单的刚体模拟器。用户可以直接拿去使用或者尝试实验。

这个算法很简单，在本质上是个几何学算法，所以很容易将之可视化并调试。这个算法可归结为一系列的点面裁剪检测，所以只要熟悉点乘和叉乘，就能理解它的数学原理。

2.5.1 介 绍

创建一个健壮的碰撞系统需要大量的时间和精力。最通用的方法是选择一些基本形体并创建 $O(N^2)$ 个独立的碰撞程序，每一个针对一对可能的形体。使用这个方法，编码、测试和调试的工作量会快速膨胀，以致失去控制。即使是一个只有 4 种基本形体的简单几何，比如球体、箱体、胶囊体和三角网格，就需要 10 种独立的碰撞程序。每个程序都会有一些特殊情况以及失效模式需要测试和调试。每次增加一种新的碰撞形体时，需要创建多个碰撞程序，一个每种已有的形体，还需要一个和自己碰撞的程序。

本精粹将介绍一个高效、紧凑的碰撞算法，适用于在实时碰撞系统中可见的任何凸多面体。新的多面体可快速、轻松地添加进来，而不用改变算法的实现，所需要添加的仅仅是该形体的一个简单的数学描述。如果需要，碰撞多面体也可以以一个较小的代价实时地修改，比如在动画物体上使用时。

这个算法被命名为 XenoCollide。它是闵可夫斯基入口简化技术（Minkowski Portal Refinement，MPR）拓展算法中的一个例子。

本精粹介绍的 XenoCollide 和 MPR 技术是新的技术，但它们都与在[GJK88]中介绍的 GJK 碰撞检测算法很相似，其不同之处在 2.5.4 小节的“比较 MPR 和 GJK”中给出。

本精粹接下来介绍支撑映射（Support Mappings）及其与闵可夫斯基法的不同。如果你已经了解这些概念，可以直接跳到 2.5.3 小节进行学习。

2.5.2　用支撑映射来表示形体

支撑许多种类形体的算法需要一种统一的方式来表示这些形体。XenoCollide 依赖支持映射来做到。支撑映射提供了一种简单、优雅的方式来表示无限种凸多面体。

一个支撑映射是一个数学函数（通常是一个非常简单的函数），这个函数以一个方向向量作为输入，返回一个在凸多面体上该方向上最远的点（支持映射经常被定义为返回法线反方向的最远点。我选择打破常规的方式，以避免等式中出现多余的负数让人迷惑）。如果有多个点满足条件，可以选择任何一个，只要保证总是返回同一个值即可。

支撑映射是直观的，且易于图形化。想象你知道一个平面的法线，如果你沿着它的法线向着凸多面体移动，最终这个平面将与这个凸多面体接触，如图 2.5.1 所示。

当它们接触时，这个凸多面体就被这个平面所支撑，而凸多面体上与该平面接触的点就是对于该平面的支撑点。

如果一整条边或一整个面与该平面接触，可选择边上或面上的任意一个点来作为在该法线上的支撑点，如图 2.5.2 所示。

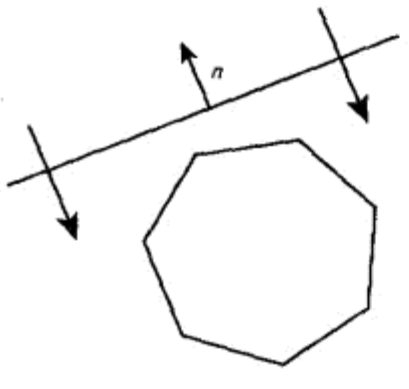
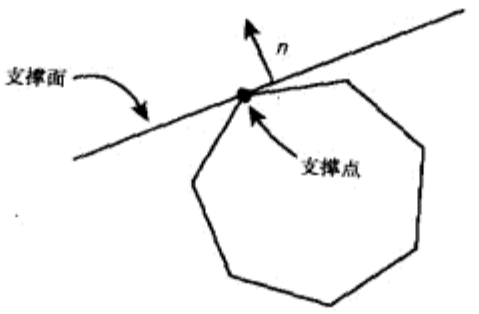


图 2.5.1　一个支撑映射以一个移动平面来图形化表示



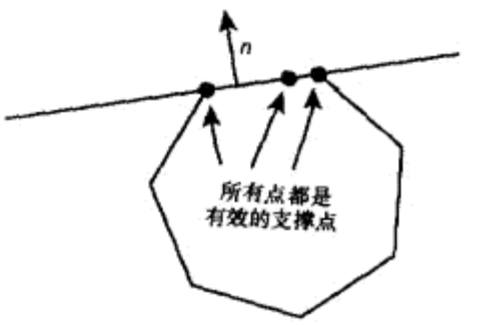


图 2.5.2　当整条边支撑该平面时，任意选择一个支撑点

基本形体

许多通用形体的支撑映射都很简单。考虑一个半径为 r 的球体，以原点为中心。如果你从任意一个方向 $\boldsymbol{n}$ 移动一个平面，你碰到的第一个点将会是在从原点发出的方向为 $\boldsymbol{n}$ 直线上，离原点距离为 r，如图 2.5.3 所示。

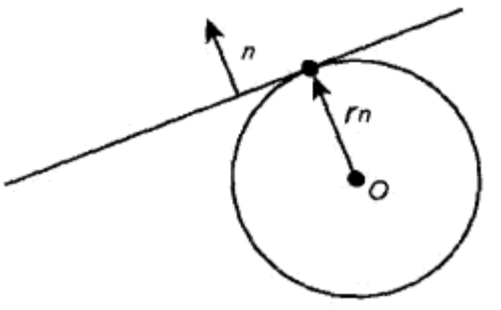


图 2.5.3　一个球体的支撑映射

球体的支撑映射以一个函数表示如下：

$$\boldsymbol{S}_{\text{sphere}}(\boldsymbol{n}) = r\,\boldsymbol{n} \qquad (2.5.1)$$

表 2.5.1 列出了一些其他通用形体的支撑映射。

表 2.5.1　简单基本形体的支撑映射

形　体	描　述	支撑映射
	点	$\boldsymbol{P}$
	线段	$[r_x \operatorname{sgn} n_x \quad 0 \quad 0]$
	长方形	$[r_x \operatorname{sgn} n_x \quad 0 \quad 0]$

续表

形　体	描　述	支撑映射
	长方体	$\left[r_x \operatorname{sgn} n_x \quad r_y \operatorname{sgn} n_y \quad r_z \operatorname{sgn} n_z\right]$
	碟形	$r\frac{\left[n_x \quad n_y \quad 0\right]}{\left\|\left[n_x \quad n_y \quad 0\right]\right\|}$
	球体	rn
	椭圆形	$\frac{\left[r_x^2 n_x \quad r_y^2 n_y \quad 0\right]}{\left\|\left[r_x n_x \quad r_y n_y \quad 0\right]\right\|}$
	椭圆体	$\frac{\left[r_x^2 n_x \quad r_y^2 n_y \quad r_z^2 n_z\right]}{\left\|\left[r_x n_x \quad r_y n_y \quad r_z n_z\right]\right\|}$

平移和旋转支撑映射

表 2.5.1 中的支撑映射都是轴对称的，而且以原点为中心。为支持世界空间中的形体，你需要平移和旋转支撑映射。

一个旋转过和平移过的物体的支撑映射可以如下步骤得到：首先将 $\boldsymbol{n}$ 变换到物体的本地空间，然后在本地空间中找到支撑点，最后将得到的支撑点变换回世界空间。

$$\boldsymbol{S}_{\text{world}}(\boldsymbol{n})=\boldsymbol{R}\boldsymbol{S}_{\text{local}}(\boldsymbol{R}^{-1}\boldsymbol{n})+\boldsymbol{T} \tag{2.5.2}$$

在本精粹的剩余部分，所有的支撑映射都在世界坐标下。

组合支撑映射

支撑映射提供了一个高效紧凑的方式来表示基本形体。它也可以用来表示更复杂的形体。这可由精确地组合两个或更多简单形体的支撑映射来得到。

你可以用以下方式“包裹”一个形体集合：先找到每个形体的支撑点，然后返回在该方向上最远的点。比如，一个以原点为中心的碟型与一个平移过的点可“包裹”为一个圆锥体，见式 2.5.3。

$$\boldsymbol{S}_{\text{cone}}(\boldsymbol{n})=\mathbf{maxsupport}(\boldsymbol{S}_{\text{Point}}(\boldsymbol{n}), \boldsymbol{S}_{\text{disc}}(\boldsymbol{n})) \tag{2.5.3}$$

为了简化组合支撑映射的表达式，我们从函数名中取掉(n)。每一个支撑映射都需要一个法线，所以可以假设法线总是存在：

$$\boldsymbol{S}_{\text{cone}}(\boldsymbol{n})=\mathbf{maxsupport}(\boldsymbol{S}_{\text{Point}}, \boldsymbol{S}_{\text{disc}}) \tag{2.5.4}$$

第二种组合支撑映射的方式是将它们相加，这会导致一个形体被另一个形体“吹胀”或“清扫”。比如，如果将一个小球的支撑映射与一个大长方体相加，会得到一个有球形角的大长方体，见式 2.5.5。

$$\boldsymbol{S}_{\text{smoothbox}}=\boldsymbol{S}_{\text{box}}+\mathbf{S}_{\text{sphere}} \tag{2.5.5}$$

表 2.5.2 列出了一些有用的组合支撑映射。

表 2.5.2　　组合形体的支撑映射

形　体	描　述	支撑映射
	胶囊体	maxsupport (S_{Sphere}, S_{Sphere}+[lenght　0　0]) 或者 S_{edge}+ S_{sphere}
	菱形	$S_{rectangle}$+ S_{sphere}
	圆角长方体	S_{box}+ S_{sphere}
	圆柱体	maxsupport (S_{disc}, S_{disc}+[0　0　height]) 或 S_{edge}+ S_{disc}
	圆锥体	maxsupport (S_{disc}, S_{point}+[0　0　height])
	轮体	S_{disc}+ S_{sphere}
	截头锥体	maxsupport ($S_{rectangle1}$, $S_{rectangle2}$+[0　0　height])
	多边形 多面体	maxsupport (S_{rert1}, S_{rert2},…)

极度复杂的形体也可轻易地由许多的基本形体和组合形体以代数方式组合得到，而这些代数运算也可以转化为内容创建工具中直观的控制。美术人员和策划人员可以使用基本形体来勾勒出一个物体主要的外在特性，然后使用交互的包裹和清除/逐出来处理余下细节。

2.5.3　使用闵可夫斯基（Minkowski）差异来简化碰撞检测

每一个凸多面体都可当作世界空间中的一个凸点集来对待。如果你用一个固态形体中的每一个点去减另一个固态形体中的每一个点，会发生一些有趣的事情。如果这两个形体有重合，那么至少在形体 A 中有一个点与形体 B 中的某个点在世界空间中的位置完全一样。

当一个这样的点去减另一个位置相同的点时，就会产生一个零向量（也就是原点）。相似的，如果两个形体完全不重合，这两个形体中将不存在位置完全相同的点，那么新产生的形体也就不会包含原点。

因此，如果能检测到新产生的形体是否包含原点，就能确定这个初始形体是否碰撞了。

用一个凸多面体减去另一个，生成的新多面体被称为这两个多面体的闵可夫斯基差异（Minkowski Difference）。两个凸多面体的闵可夫斯基差异，$B-A$，同样也是凸多面体。如果 $B-A$ 包含原点，A 和 B 一定相交。如果 $B-A$ 不包含原点，A 和 B 不相交。

真正使用一个形体中的每一个点去减去另一个中的每一个点几乎不可能，代价太昂贵。但是，你可以轻松地确定闵可夫斯基差异 $B-A$ 的支撑映射，如果已知 A 和 B 的支撑映射，使用式 2.5.6，可将两个凸多面体 A 和 B 的碰撞检测问题归结为确定原点是否在一个凸多面体 $B-A$ 之中。

$$S_{B-A}(\mathbf{n}) = S_B(\mathbf{n}) - S_A(\mathbf{n}) \tag{2.5.6}$$

2.5.4 使用闵可夫斯基入口简化（Minkowski Portal Refinement，MPR）来检测碰撞

到现在为止所述的步骤，GJK 算法与 XenoCollide 都是一样的。本精粹的剩下部分将主要讨论 XenoCollide 和 MPR 技术。如果你对 GJK 感兴趣，想要了解更多，[van den Bergen03]、[GJK88]和 [Cameron97]都是很好的参考资料。如果你想了解 XenoCollide 其他背景知识和细节，可以参阅[Snethen07]。

以下是 XenoCollide 的伪码。

```
// 第一阶段：发现入口
find_origin_ray();
find_candidate_portal();
while ( 当原射线和候选者无交叉 )
{
    choose_new_candidate();
}
// 第二阶段：入口重构
while (true)
{
    if (原点在入口内) return hit;
    find_support_in_direction_of_portal();
    if (原点在支持面的外侧) return miss;
    if (支持面靠近入口) return miss;
    choose_new_portal();
}
```

对于这个算法的每一步，我们接下来都会详细地解释。

find_origin_ray()；步骤

首先要找到闵可夫斯基差异 $B-A$ 中的一个点，这个点可以由 B 中的一个已知点减去 A 中的一个已知点来得到。几何中心（或质量中心）是方便使用的选择。实际上，任意内部的点都可行。相减所得的点是 $B-A$ 的内部点。这个内部点在图 2.5.4 中被标记为 $V0$。

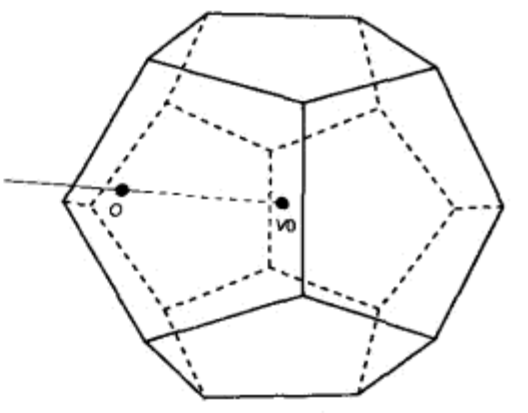


图 2.5.4 步骤 1 包括找到原点射线

这个内部点很重要，因为它是在 $B-A$ 内部。如果从这个内部点向原点画一条射线，该射线称为原点射线，而这条射线在到达原点之前先穿过 $B-A$ 的表面，那么原点在 $B-A$ 外部。反之，如果这条射线在到达 $B-A$ 表面前先穿过原点，那么原点在 B—A 内部。

find_candidate_portal()；步骤

算法的这个步骤将使用 $B-A$ 的支撑映射来找到 $B-A$ 表面上不共线的 3 个点。这个 3

个点形成一个原点射线可能通过（也可能不通过）的三角形入口。见图 2.5.5。

有很多方法来获得 3 个不共线的表面点。XenoCollide 使用以下的支撑顶点：

```
// 在原先射线的方向上找到一个支持点
V1 = S( normalize(-V0) );

// 找到与平面垂直的点
// 包括原点、内部点和第一个支持点
V2 = S( normalize(V1 × V0) );

// 找到垂直于平面的点
// 包括内部点和第一次找到的那两个支持点
V3 = S( normalize((V2-V0) × (V1-V0)) );
```

while(origin ray does not intersect candidate)；步骤

现在你需要测试原点射线是否与该备选三角形相交。你可以通过测试原点是否在由三角形的 3 条边和内部点形成的 3 个平面的内侧：（*v*0, *v*1, *v*2）；（*v*0, *v*2, *v*3）和（*v*0, *v*3, *v*1）。如果原点在这 3 个平面的内侧，你已经找到一个有效的入口，就可以继续下一步骤了，否则必须选择一个新的可能入口并且再次尝试。

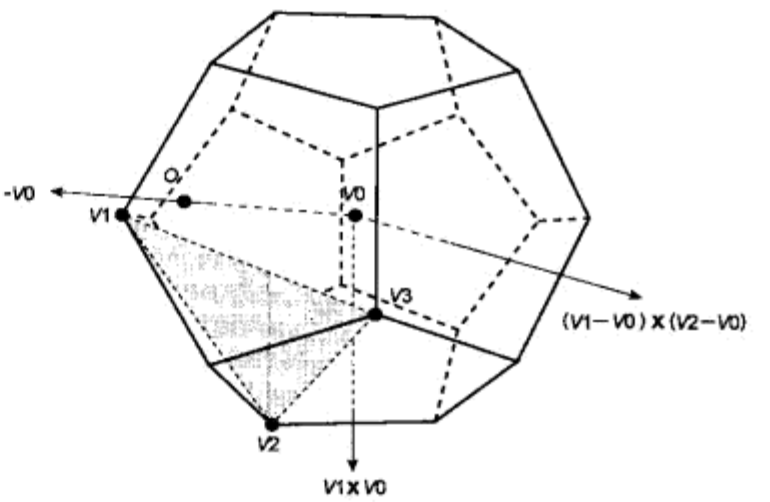


图 2.5.5　步骤 2 包括找到一个备选入口

choose_new_candidate()；步骤

如果原点在某个平面的外侧，使用这个平面的向外法线来找到一个新的支撑点，如图 2.5.6 所示。

新的支撑点用来取代那个在该平面内部的三角形顶点。这个新的支撑顶点提供了一个新的备选入口（见图 2.5.7）；重复以上步骤直到你找到一次击中。

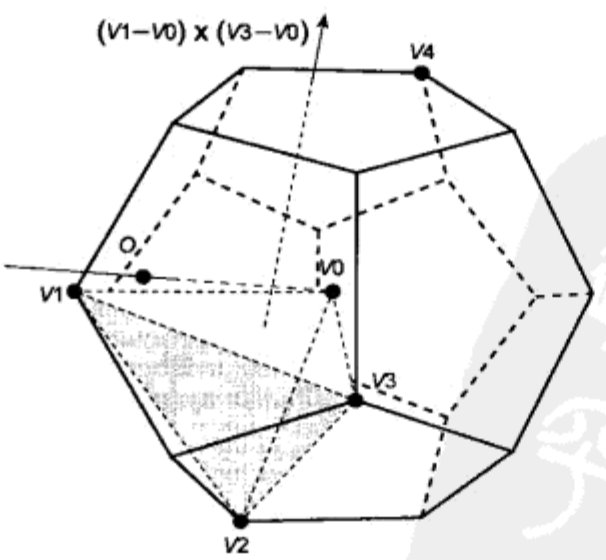


图 2.5.6　步骤 3 包含找到一个新的备选入口

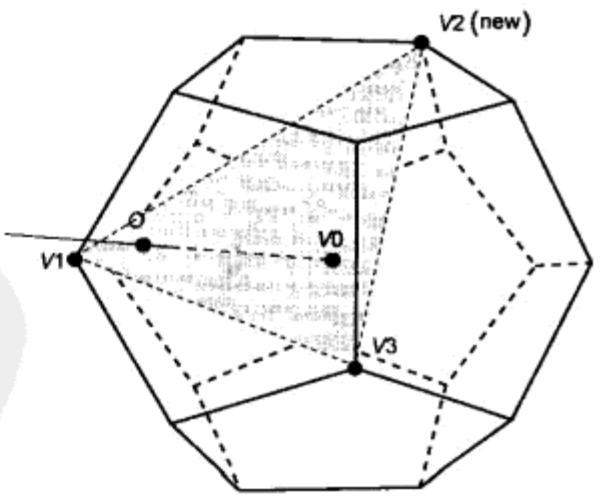


图 2.5.7　步骤包括找到一个有效的入口

if(origin inside portal) return hit; 步骤

点 *V*0、*V*1、*V*2 和 *V*3 构成了一个四面体。由于 *B*－*A* 的凸面性，这整个四面体将在 *B*－*A* 的内部。如果原点在该四面体内，那它也一定在 *B*－A 的内部。现在你知道原点在该四面体的 3 个平面的内部，因为原点射线从 *V*0 出发并且穿过了作为四面体底面的三角形入口。如果原点在入口的内侧，那么它就在这个四面体的内部，也就是在 *B*－*A* 的内部。这种情况下，就可以返回击中。

find_support_in_direction_of_portal(); 步骤

如果你执行到这一步，说明原点在三角形入口的外侧。但是，你并不知道原点是在入口 *B*－*A* 的内部接近入口的地方还是完全在 *B*－*A* 的外部。你需要更多的关于三角形入口外侧的信息，所以你应该使用该入口的外侧法线来获得一个新的支撑点。这个支撑点在入口平面的外侧，见图 2.5.8。

if(origin outside support plane) return miss; 步骤

如果原点位于由新的支撑顶点以及支撑法线所决定的支撑平面的外侧，那么原点就在 *B*－*A* 的外部，这个算法应该返回不相交。

choose_new_portal(); 步骤

原点处于入口和新的支撑平面之间，所以你需要找到一个更靠近 *B*－*A* 表面的入口来使你的查找更精确。考虑由支撑顶点以及入口所组成的四面体。

原点射线通过入口进入该四面体，所以它应该通过该四面体由支撑点和入口的 3 条边所组成的 3 个面穿出该四面体。这个步骤找到 3 个平面中射线穿出的那一个，并以此替代入口作为新的入口。为此，你需要检测原点相对于以下 3 个平面的位置：（*V*4, *V*0, *V*1）、（*V*4, *V*0, *V*2）和（*V*4, *V*0, *V*3）。

原点会在其中两个平面的同一侧，则在这两个平面中间的四面体的侧面就是新的入口，如图 2.5.9 所示。

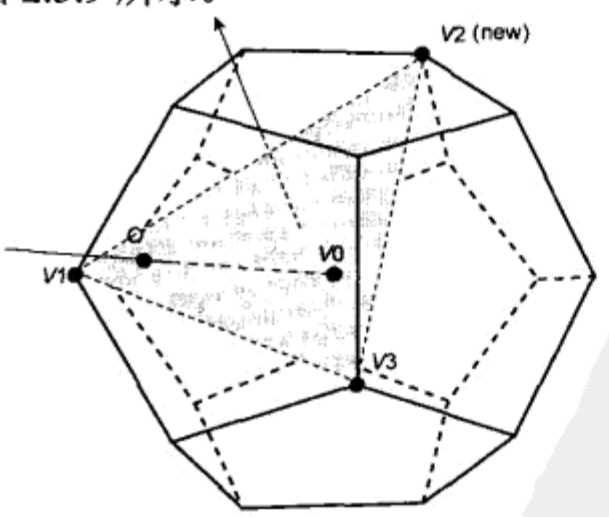


图 2.5.8 步骤 5 包括在入口法线方向上找到一个新的支撑顶点

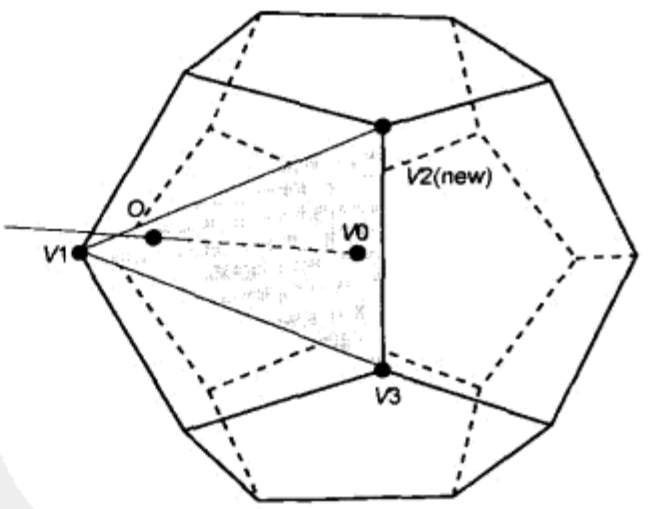


图 2.5.9 步骤 6 包括建立一个新的入口

if (support plane close to portal) return miss; 步骤

随着算法的不断迭代，精炼的入口会迅速贴近 *B*－*A* 的表面；但是，如果 *B*－*A* 包含一个曲线表面，原点有可能在无限接近它的曲线表面的地方。这种情况下，精炼入口可能需要一个任意次

的迭代来测试原点。要在这种情况下中止算法，你需要一个阈值。当入口与表面足够接近时（可由入口到与之平行的支撑平面的距离确定），你就可以终止该算法了。你可以选择相交或者不相交作为结果来中止算法，这完全取决于你想要一个不精确的正面还是负面结果。

对于物理模拟，通常来说宁可返回一个错误的负面结果（当该点可能稍在表面之内，返回一个不相交）。轻微的穿透很难被发现，而在接触处的可见的间隙看起来就不太自然了。

终止条件

这个算法会一直继续运行，直到以下条件被满足：

- 原点在某个入口之内（相交）；
- 原点在某个支撑平面之外（不相交）；
- 入口和与其平行的支撑平面的距离小于阈值（若即若离——可被视为相交或者不相交，取决于程序）。

终止条件的正式证明已经超出了本节的范围。一个非正式证明可在[Snethen07]中找到。

2.5.5　使用 MPR 算法得到相交信息

如果你只是检测相交，只要原点在入口内侧就可以停止 MPR 算法了。但是，如果你需要相交的详细信息，MPR 可以继续执行以获得相交点、相交法线和透深。

MPR 提供了多种方法来获得相交信息。XenoCollide 采用的方法是简单地将原点射线继续发射到闵可夫斯基差异的表面上，这相当于将两个物体沿着它们内部点连线向外推，直到它们刚刚相交，使用它们刚刚相交时的法线作为碰撞法线。这个方法简单高效，并可以产生稳定的相交信息。

第二种方法是找到一对重合点的相对速度，然后顺着该速度向量的反方向发射一条从原点到表面的射线，这样可以得到在运动方向上的透深，产生更为真实的动态碰撞。

额外优化

XenoCollide 效率表现很好，但是，针对特殊形体组合进行深入优化的算法肯定会比通用目的的算法更快。致力于获得理想效率的读者可以以 XenoCollide 技术作为基础来处理所有形体，然后再在开发过程中添加特殊算法来处理那些会使效率受益最多的形体对。一个明显的优化候选就是球球检测，当它被特殊处理时，代价是最小的。

另外一个重要的优化是缓存每一个碰撞测试的结果，来辅助下一个时间步的相同测试。如果找到一个支撑平面来证明两个物体没有碰撞，那么在后续帧中，这个平面也可用来做一个早期的分离测试。同样，如果找到一个入口在原点之外，可使用相同的入口来快速确定这两个物体是否还是相交。

支撑映射函数在每个碰撞检测过程中被多次调用。为优化效率，应尽可能减少坐标转换、归一化和函数调用的开销。在有些情况下，在世界空间中计算支撑映射会更好。举例来说，在世界空间中使用 dot($\boldsymbol{n}$, $\boldsymbol{d}$)来计算会比先把 $\boldsymbol{n}$ 转换到本地空间，只取 x 分量，然后再把结果转换回世界空间中会更省时。

比较 MPR 和 GJK

GJK 是 MPR 的灵感源泉之一。GJK 也支持各种凸多面体，但是它有一些局限在 MPR 中得到了解决。

- GJK中单一的精炼算法是基于一个对于大多数游戏编程人员都不太熟悉的代数公式。这个公式需要使用行列式和余子式，这些计算都很容易引起浮点数精度问题。精度问题和难以想象的数学使得大部分游戏开发人员难以实现一个健壮的 GJK。
- 由于之前的这些问题，许多 GJK 的变种在设计时是把 GJK 作为一个几何问题的解决方案而不是一个代数问题。每一个 GJK 的实现都需要考虑一个四面体中 8～15 种特性（点、边、面和内部）的沃罗诺伊区域（Voronoi region），来检查哪一个更靠近原点。因为有许多不同的条件和分支操作，这将是一个复杂并可能很费时的工作。
- 通常情况下，GJK 不会提供一个精确的相交法线、透深或者表面相交点。需要一个独立的算法，比如 EPA[van den Bergen03]，来获得这些信息。

MPR 解决了所有这些问题。

- MPR 简单而且是几何直观的。这项技术的每一步都可以轻易图形化并在屏幕上查证，这使得它易于理解，测试和调试。
- MPR 只需要很少的分支测试，不需要在 8～15 种特性中选择，只有 2～3 种需要计算。
- MPR 可以用来检测碰撞，也可以用来鉴定碰撞细节，它的表现良好，在各帧中结果保持一致。

2.5.6 结 论

本节介绍了一个可用于多种形体碰撞检测的简单算法。这些形体可以从一个无限的有用凸多面体的设计池中选出。引入一个新的形体非常简单，并且封装成一个可视化的用户界面供美术和策划人员使用。这个算法为游戏玩法和动态刚体运动提供了一个通用目的碰撞系统的健壮基础。这个算法是高效的，但是，如果需要额外的优化，优化过的针对特殊形体对的算法也可以加入这个通用的框架。

2.5.7 致 谢

笔者要感谢 Crystal Dynamics 的管理人员、程序员和开发人员，感谢他们的支持以及对本文的意见。还要感谢 Erin Catto，感谢他对分享这项工作的善意和鼓励。

2.5.8 参考文献

[Cameron97] Cameron, S. "Enhancing GJK: Computing Minimum and Penetration Distances Between Convex Polyhedra," *Proceedings of IEEE International Conference on Robotics and Automation (1997)*, pp. 3112–3117.

[GJK88] Gilbert, E.G., Johnson, D.W., and Keerthi, S.S. "A Fast Procedure for Computing the

Distance Between Complex Objects in Three-Dimensional Space," *IEEE Journal of Robotics and Automation*, Vol. 4, No. 2 (1988), pp. 193–203.

[Snethen07] Snethen, Gary. "XenoCollide Website."

[van den Bergen03] van den Bergen, Gino. *Collision Detection in Interactive 3D Environments*, Morgan Kauffman, 2003.

2.6 使用变换语义的高效碰撞检测

José Gilvan Rodrigues Maia, UFC
gilvanmaia@gmail.com

Creto Augusto Vidal, UFC
cvidal@lia.ufc.br

Joaquim Bento Cavalcante-Neto, UFC
joaquimb@lia.ufc.br

第一人称射击游戏是如何判断一次射击是否击中了敌人的头部呢？如何避免游戏物体穿过墙体或者掉落至无尽的虚空呢？如何检查玩家是否碰到一个物件或者是否到达了一个存盘点呢？一个引擎如何确定动态模拟物体的相交呢？虽然有很多方法来解决以上问题，碰撞检测（CD）是处理这些问题的一个很重要的工具。CD 能够判断一个给定的几何体集合是否有重合。因此，它是几乎所有计算机游戏中至关重要的组成部分。

现代渲染技术支持用变换矩阵来表示几何体在场景中的位置。因此，对于一个游戏程序员来说，了解如何创建和使用这些矩阵是非常重要的。此外，碰撞检测方法不仅需要考虑物体的形状，还需要考虑它们相应的矩阵，以进行相交测试。

本精粹将向用户展示一个方法，这个方法能够转化在游戏中用来摆放模型的变换矩阵，并从中抽取有用的语义信息。本精粹也会解释这些语义信息是如何用来实现高效的碰撞检测的。

2.6.1 仿射变换和游戏

本小节简要地讨论线性代数中的一些概念。仿射变换是用来在两个向量空间之间做映射的，使用一个线性变换（也就是一个矩阵乘法）跟着加上一个原点偏移向量。一个仿射变换（也被称为仿射映射）定义如下：

$$q = Ap + t \tag{2.6.1}$$

仿射映射被用在很多计算机图形程序中。比如说，在变换管线中，从世界空间到摄像机空间的顶点变换就是使用轴变换来实现的——这是一类特别的仿射映射。因为计算机图形系统使用 4×4 矩阵来实现线性变换，我们使用分块矩阵形式来定义式 2.6.1 中的映射。

$$M=\begin{bmatrix} A & t \\ 0 & 1 \end{bmatrix} \tag{2.6.2}$$

观察式 2.6.2 中的仿射映射，A 是一个 3 × 3 的矩阵，而原点偏移 t，是一个 3 × 1（列）的矩阵。所有的在第 4 行中的分量被定义为 0，除了最后一个分量外。对于计算机游戏，可假设在矩阵乘法执行之前输入顶点已经做过的变换，而它们的齐次坐标 w 分量被设为 1。三维几何体是使用正常的笛卡儿坐标来记录的，但顶点是使用齐次坐标来处理的。

从游戏编程的角度来看，要解决的问题是检测场景中模型之间的碰撞，而每个模型都有相应的矩阵以摆放在世界坐标中。因此，在对一个给定模型做碰撞检测时，必须考虑到这个矩阵。

注意到将模型摆放至世界空间的典型矩阵符合式 2.6.2，因为这些矩阵通常是由一些基本变换组合得到的。而这些基本变换本身也是仿射映射——缩放、旋转和平移。这些基本变换可用来解释变换矩阵，并且回答一些关于模型位置的常见问题。比如，它的大小、模型面朝的方向、模型的原点在哪里等。因此，放缩、旋转和平移被称为变换语义（Transformation Semantics）。

变换语义可用来简化和加速碰撞检测过程中的一些计算。下一小节将展示一个高效的分解方法，以从一个变换矩阵中反转并获得变换语义。这将回答以下的问题——给定一个 4 × 4 数组表示一个仿射映射，它的逆矩阵是什么？它相应的缩放、旋转和平移部分是什么？

2.6.2　从矩阵中抽取语义

一些语义信息可在通常情况下获得，以下会简要解释。但是，在游戏开发中，你通常对分解任意仿射映射不感兴趣。因而，本小节的大部分将关注于之前描述的用于在典型游戏中摆放模型的矩阵。这将使用户可应用一个更高效的方法。

通用仿射映射

你能从任意仿射映射中抽取一些语义吗？是的，你可以。如式 2.6.2 所示，t 表示一个原点偏移。因此，平移部分唾手可得——你只需要从给定矩阵的最后一列中取出。一些朝向信息也可以轻易地抽取出来。仔细观察在给定顶点上的矩阵乘法：

$$Ap=\begin{bmatrix} A_{11} & A_{12} & A_{13} \\ A_{21} & A_{22} & A_{23} \\ A_{31} & A_{32} & A_{33} \end{bmatrix}\begin{bmatrix} p_x \\ p_y \\ p_z \end{bmatrix}=\begin{bmatrix} p_xA_{11}+p_yA_{12}+p_zA_{13} \\ p_xA_{21}+p_yA_{22}+p_zA_{23} \\ p_xA_{31}+p_yA_{32}+p_zA_{33} \end{bmatrix} \tag{2.6.3}$$

将式 2.6.3 按 p 中的轴对齐的分量分组，将得到：

$$Ap=p_x\begin{bmatrix} A_{11} \\ A_{21} \\ A_{31} \end{bmatrix}=p_y\begin{bmatrix} A_{12} \\ A_{22} \\ A_{32} \end{bmatrix}=p_z\begin{bmatrix} A_{13} \\ A_{23} \\ A_{33} \end{bmatrix} \tag{2.6.4}$$

从式 2.6.4 可以清楚地看到 A 的第一、第二、第三列分别表示了一个点在变换之后相应的新的 x、y、z 轴。这就意味着你可以使用仿射变换的各列来解决关于朝向和缩放的问题——一个简便的例子将在之后详细讲解。虽然式 2.6.4 清楚地指出了矩阵是如何改变顶点的朝向的，但只有平移和旋转语义从这些分析中得到。

一个仿射矩阵的逆矩阵已经众所周知了，如下所示：

$$\boldsymbol{M}^{-1}=\begin{bmatrix}\boldsymbol{A} & \mathrm{t}\\ 0 & 1\end{bmatrix}^{-1}=\begin{bmatrix}\boldsymbol{A}^{-1} & -\boldsymbol{A}^{-1}\mathrm{t}\\ 0 & 1\end{bmatrix} \tag{2.6.5}$$

对式 2.6.2 中仿射映射的逆矩阵求法感兴趣的读者可参阅 Kevin Wu 的文章[Wu91]。

我们的特殊情况：模型矩阵

考虑一个变换矩阵 $\boldsymbol{M}$ 将一个模型从它的本地空间转换至世界坐标。你可以假设 $\boldsymbol{M}$ 是由一个不均匀缩放矩阵以及后面许多的旋转以及平移矩阵按顺序组成的，这个假设是合理的，因为这与大多数场景表示法是兼容的，比如场景图。为简便起见，$\boldsymbol{M}$ 可写成仅仅 3 个矩阵的乘积——$\boldsymbol{S}$（放缩）、$\boldsymbol{R}$（旋转）和 $\boldsymbol{T}$（平移）：

$$\boldsymbol{M}=\boldsymbol{M}_1\ldots\boldsymbol{M}_k\boldsymbol{S}=\boldsymbol{TRS} \tag{2.6.6}$$

这里，每个 $\boldsymbol{M}_i$ 表示旋转或者平移矩阵。

Kevin Wu[Wu94]在他的书中使用了一种等价的矩阵形式，并且描述了一种简单、高效的方法来反转保持向量间角度的矩阵和保持向量长度的矩阵。但是，他的方法只考虑了均匀放缩。我们的矩阵表达式更通用（考虑到了不均匀缩放），并且可使用一个分块矩阵形式重写：

$$\boldsymbol{M}=\boldsymbol{TRS}=\begin{bmatrix}I & t\\ \mathbf{0} & 1\end{bmatrix}\begin{bmatrix}i & j & k & 0\\ 0 & 0 & 0 & 1\end{bmatrix}\begin{bmatrix}s_x & 0 & 0 & 0\\ 0 & s_y & 0 & 0\\ 0 & 0 & s_z & 0\\ 0 & 0 & 0 & 1\end{bmatrix}=\begin{bmatrix}s_x i & s_y j & s_z k & t\\ 0 & 0 & 0 & 1\end{bmatrix} \tag{2.6.7}$$

你可以看到这个表达式的一些细节：$\boldsymbol{t}$ 表示平移部分；$\boldsymbol{i}$，$\boldsymbol{j}$ 和 $\boldsymbol{k}$ 是形成正规化轴的列矩阵；而 $\mathbf{S_x}$，$\mathbf{S_y}$ 和 $\mathbf{S_z}$ 是非零（否则 $\boldsymbol{M}$ 是异常的）的放缩因子。由于 $\boldsymbol{M}$ 是一些基本变换的乘积，它的逆矩阵是已知的，如下所示：

$$\boldsymbol{M}^{-1}=(\boldsymbol{TRS})^{-1}=\boldsymbol{S}^{-1}\boldsymbol{R}^{-1}\boldsymbol{T}^{-1}=\begin{bmatrix}\boldsymbol{i}^T/\mathrm{s}_x & -\boldsymbol{i}\cdot\boldsymbol{t}/\mathrm{s}_x\\ \boldsymbol{j}^T/\mathrm{s}_y & -\boldsymbol{j}\cdot\boldsymbol{t}/\mathrm{s}_y\\ \boldsymbol{k}^T/\mathrm{s}_z & -\boldsymbol{k}\cdot\boldsymbol{t}/\mathrm{s}_z\\ \mathbf{0} & \mathbf{1}\end{bmatrix} \tag{2.6.8}$$

现在，让我们看看如何反转这种矩阵而不用计算任何平方根。因为 i 是个单位向量，在 x 轴上放缩因子的平方可由第 1 列向量点乘自身得到：

$$(s_x\boldsymbol{i})\cdot(s_x\boldsymbol{i})=s_x^2(\boldsymbol{i}\cdot\boldsymbol{i})=s_x^2(\mathbf{1})=s_x^2 \tag{2.6.9}$$

对第 2 列和第 3 列也是一样：

$$(s_y \boldsymbol{j}) \cdot (s_y \boldsymbol{j}) = s_j^2 (\boldsymbol{j} \cdot \boldsymbol{j}) = s_y^2 (\mathbf{1}) = s_y^2 \tag{2.6.10}$$

$$(s_z \boldsymbol{k}) \cdot (s_z \boldsymbol{k}) = s_z^2 (\boldsymbol{k} \cdot \boldsymbol{k}) = s_z^2 (\mathbf{1}) = s_z^2 \tag{2.6.11}$$

这个属性非常有用，因为放缩因子的平方可用来计算 M^1（式 2.6.8）中 3 × 3 部分中的第 1 行，只需一个简单的除法：

$$\left(\frac{s_x \mathbf{i}}{s_x^2}\right) = \frac{\mathbf{i}^T}{s_x} \tag{2.6.12}$$

知道了这个，你就可以如下计算逆矩阵了：使用式 2.6.9、式 2.6.10 和式 2.6.11 计算放缩因子的平方；转置式 2.6.2 中的子矩阵 A，并将结果各行除以相应的放缩因子平方。注意到，至此，你已经计算出了 $\boldsymbol{A}^{-1}$。最后，最后一列可由- $\boldsymbol{A}^{-1}\boldsymbol{t}$ 轻松计算得到。

这个求逆矩阵方法可灵活地支持不均匀放缩，并且非常高效。只需使用 27 次乘法、3 次除法和 12 次加法。表 2.6.1 比较了这个方法、强力方法（余子式和选主元高斯消元法）、Kevin Wu 用于仿射映射的方法，以及 Kevin Wu 用于保持角度矩阵支持均匀放缩的方法。表 2.6.1 中只考虑了一列 C 实现，鼓励读者使用 SIMD/MIMD 指令——SSE、SSE2 等来实现这些方法。

表 2.6.1　基于所需运算来比较矩阵求逆方法

求逆方法	除　法	乘　法	加　法
[Wu94]（保持角度）	1	21	8
我们的方法	3	27	12
[Wu91]（通用情况）	1	48	22
余子式	1	280	101
部分选主元的高斯消元法	10	51	47

变换语义的抽取需要在这个方法上稍做改动。如式 2.6.2 所示，平移部分直接可以得到。在式 2.6.9、式 2.6.10 和式 2.6.11 上求平方根可以得到放缩因子。旋转部分可通过将前 3 列除以相应的缩放因子获得。因此，语义抽取需要 3 次开平方运算、3 次除法、18 次乘法和 6 次加法。

当只考虑均匀缩放时，很明显求逆方法应使用 Kevin Wu 针对保持角度矩阵的方法 [Wu94]。此外，在这种情况下，语义抽取需要一次开平方、一次除法、12 次乘法和 2 次加法。

现在你可以非常高效地对将模型放入场景的典型矩阵进行求逆操作和语义抽取。下一小节将介绍语义信息是如何在碰撞检测中使用的。

2.6.3　在碰撞检测中使用变换语义

现实程序中的碰撞检测方法必须考虑一个给定的模型集合，而这项工作通常可分为两个连续阶段。第一个阶段被称为粗检测阶段（Broad Phase）或者碰撞选择（Collision Culling），

负责大致找到所有的物体对——可能碰撞集。更重要的是，这个阶段致力于丢弃那些相隔很远不可能碰撞的物体对，以避免费时的相交测试。下一个阶段被称为细检测阶段（narrow phase）或碰撞对处理（pair processing），包括在粗检测阶段中收集到物体对之间有效的相交测试。变换语义在这些阶段中都可用到。

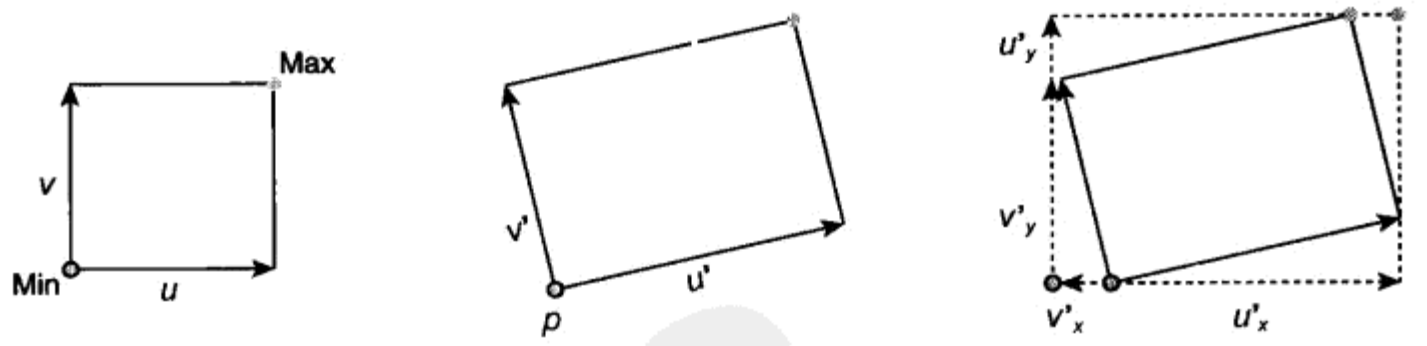
加速粗检测阶段

给定 N 个物体，有 $O(N^2)$个可能碰撞对需要检测。然而，大多数碰撞对中的物体彼此相距很远，因此，许多对可以快速地丢弃——这就解释了这个阶段为什么被称为碰撞选择。空间哈希和 sweep-and-prune 法就是专为这个目的设计的，这些技术的通用实现需要轴对齐包围盒（AABB）的知识。

场景表示法通常都为每一个几何模型维持一个本地 AABB；比如，一个在模型自己本地空间的 AABB。这个盒子也需要使用模型的矩阵来变换，这样它就变为一个世界空间中的方向包围盒（OBB）。这时的问题如下：你如何才能高效地从模型的本地 AABB 和变换矩阵 $\boldsymbol{M}$ 计算出它的全局 AABB？一个暴力的方法是首先将 AABB 的 8 个角顶点转换至世界坐标中，然后计算转换后顶点的 AABB——这种方法需要21个分支。避免计算量和避免分支，对改进效率都是非常重要的。虽然现代 CPU 可以在代码执行前预测它的行为，但较少分支的代码还是执行得稍快。

针对这个问题，Charles Bloom 在他的游戏引擎[Bloom06]中提议了一种优美、高效的解决方案。他的方法是纯几何的，并且基于式 2.6.4 的朝向语义。考虑一个 AABB 的最小-最大表示。首先，最小的顶点使用模型矩阵来变换，获得一个点 p。朝向语义被用来获得模型经过 $\boldsymbol{M}$ 变换后在世界空间中新的轴向。然后这些轴向乘以各自的尺寸，获得变换后盒子的边向量，之后这些边每个分量的正负号也会被检查。

注意到负分量将 p 朝向全局 AABB 的最小顶点移动，所以极小点可由 p 加上边向量所有的负分量得到。相应地，p 加上边向量所有的正分量可以获得极大点——这样只需要 9 个分支，如图 2.6.1 所示。



（a）本地盒子（轴对齐）　（b）变换之后的盒子（转向并缩放）　（c）全局 AABB

图 2.6.1　一个贴合给定模型的本地 AABB（a）通过一个矩阵变换到世界坐标下（b）。注意到 p 处于贴合变换过的 AABB 的全局 AABB 的边界上。全局盒子可通过最小点加上边向量的分量得到（c）

ON THE CD

在上个版本的一篇文章中，Chris Lomon 解释了如何使用浮点数来高效地实现许多“把戏”[Lomon07]。他的方法可以用来抽取正负标志位，从而得到一个本方法较少分支的一个实现，详见 CD-ROM。

细检测阶段

碰撞检测的最后一步是处理粗检测阶段得到的所有物体对。这个阶段必须考虑两个模型 A 和 B，以及它们各自的变换矩阵 $\boldsymbol{M}_A$ 和 $\boldsymbol{M}_B$。以下的问题必须解答：这些模型在变换之后是否相交？如果相交的话，报告一个描述接触表面的相交图元对（三角面片）的列表。

层次包围盒（bounding volume）非常适合解决这个问题，因为它提供了模型的一个多分辨率的表示法，可以有效地丢弃不相交的部分。AABB 树在处理可变形模型时特别有用（比如人物），因为当几何体变形时，它可以便捷地更新[Bergen97]。

我们定义两个很有用的矩阵:

$$\boldsymbol{M}_{AB} = \boldsymbol{M}_B^{-1}\boldsymbol{M}_A \tag{2.6.13}$$

$$\boldsymbol{M}_{BA} = \boldsymbol{M}_B^{-1}\boldsymbol{M}_A \tag{2.6.14}$$

$\boldsymbol{M}_{AB}$ 将几何体从 A 的本地空间映射到 B 的本地空间。相反，$\boldsymbol{M}_{BA}$ 将几何体从 B 的本地空间映射到 A 的本地空间。这些矩阵可用之前所述的求逆方法以及一个矩阵乘法获得。三角面片能从一个物体的本地空间映射到另一个物体的本地空间，从而直接进行相交测试。

实际上，变换语义可灵活地将一个给定空间的几何体变换到一个通用空间，在这个通用空间中进行操作，以使运算更为便利。使用这个方法，你可以从 7 个坐标系中挑选一个来进行运算，如图 2.6.2 所示。

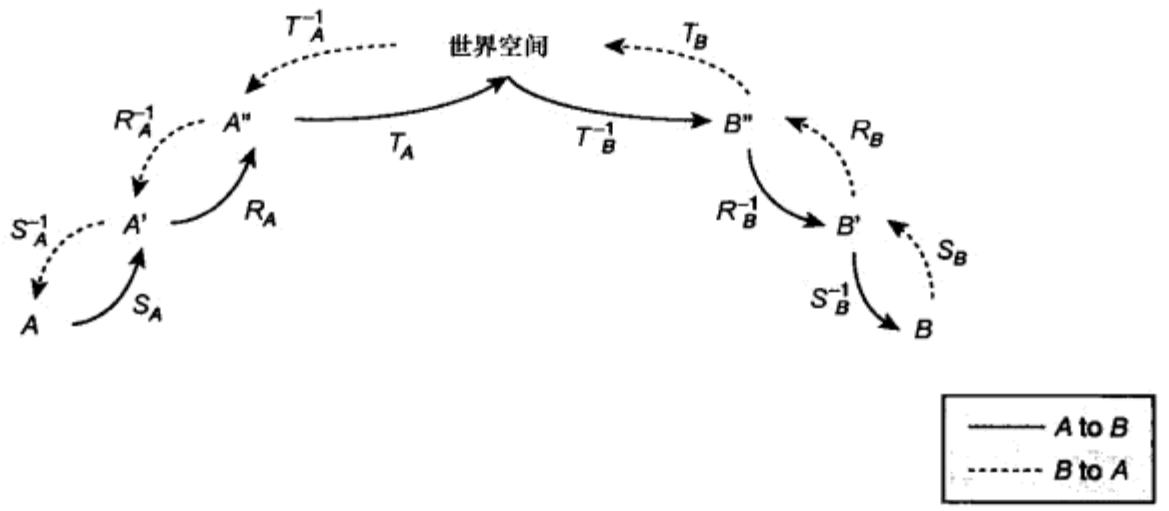


图 2.6.2　使用变换语义来处理不同本地空间中的几何体

在图 2.6.2 中，箭头表示几何体如何从一个给定空间（A、B 或者世界空间）中变换到一个更适合处理的通用空间。根据这个结构，你可以看到从 B 的本地空间到 A 的本地空间的映射为式子 $\boldsymbol{R}_A^{-1}\boldsymbol{T}_A^{-1}\boldsymbol{T}_B\boldsymbol{R}_B\boldsymbol{S}_B$（变换顺序从右到左）。

为了检测一对 AABB 树的碰撞，盒子与盒子相交测试也是必须的。注意，找到不相交的盒子对可以避免许多三角面片相交测试，因为如果包围盒不相交，整棵子树也不可能相交。此外，盒-盒相交测试也考虑到了一个模型与一个摆放在世界坐标下的方向盒的碰撞检测。在游戏中，这可以用来确定玩家是否触碰到某个重要物件，并以此来触发一些 AI。

盒-盒测试一般是使用分离轴定理（Separating Axis Theorem，SAT）[Gottschalk96]来实现的。这个方法基于迭代地搜寻一个分离轴，使在轴上的投影不相交，在某些情况下可以很快地返回。给定的盒子只有在 15 个可能的分离轴上都没有分离才是相交的。

Bergen 指出在 SAT 中找到的分离轴约有 60%是与某个盒子表面法线相关的[Bergen97]。基于

这个结论，他采用了一个粗略的相交测试（SAT-lite），只检测这类分离轴。虽然不是所有的分离情况都被处理了，但是这个测试提供了更快的碰撞选择，因为只有 15 根中的 6 根轴被测试了。

使用变换语义，一个使用 SAT 的盒-盒测试可如下执行：首先，使用从各盒子相应模型的变换矩阵中抽取的放缩语义来放缩每个盒子；之后，只需考虑旋转和平移语义，使用 SAT 或 SAT-lite 来搜寻一个分离轴。注意，放缩调整时的这个测试的盒子是从 A 和 B 的本地空间变换出来的（见图 2.6.2）。当然，任何盒-盒相交测试方法都会从提供缩放模型支持的小改动中获益。此外，只需要 12 个额外的乘法，考虑到可以高效地实现缩放，这些开销并不过分。

在这个例子中，碰撞测试中只考虑 AABB 树。实际上，其他有用的相交测试可通过使用世界空间中的形体、射线以及直线来测试一个模型。一些形体的例子：球体、圆锥体、AABB、OBB 和胶囊体。

基本上，要支持一个给定图元对一个 AABB 树的碰撞测试，需要两个相交测试方法。第一个方法检测这个图元是否与变换过后 AABB 树的一个盒子相交，提供快速的子树碰撞选择。第二个方法检测这个图元与模型中三角面片的相交。第二个方法可在执行相交测试前先使用模型的矩阵将三角面片进行变换。

如图 2.6.2 所示，变换语义抽取允许在 7 个坐标系中任选一个来进行相交测试。这个选择影响到相交测试的复杂度和效率。比如，选择将一个世界坐标中的球体变换到一个模型（比如说 A）的本地空间，由于不均匀缩放，会导致这个球变形为一个椭球体，从而增加了相交测试的复杂度。另外，将球体变换到 A' 的空间可以使相交测试更简单：只需要在计算前先放缩 A 中的三角面片和盒子。

你可能会认为，避免在本地模型空间中进行相交测试，可以使相交测试更简单和快速。实际上，当考虑到射线与模型的测试时，这并不成立。将一条射线变换到模型的本地空间会扭曲它的方向和长度。世界空间中表示射线方向的单位向量，变换到模型本地空间中可能不再是单位向量了。

但是，注意这个过程是基于一个线性变换的。因此，如果使用同样的参数，任何本地空间中返回的参数点将映射到世界空间中的同一个点。因此，可通过使用一个不依赖单位方向向量的相交测试，并预计算 A 空间中的射线来获得一个更快的算法。我们在奔腾 D 处理器上做了对不同模型的射线追踪算法的测试，结果显示在模型空间（A）中的相交测试比一个基于 A' 空间的实现要快 57%。

2.6.4 结 论

本精粹主要介绍了如何使用语义信息来加速和简化碰撞检测各个阶段的计算。你可以了解到如何高效地对模型矩阵求逆和抽取语义。此外，现在你也可以利用变换语义来进行高效的碰撞检测。虽然这里只演示了 3D 的情况，本精粹所述的概念可直接推广并应用于其他维度和其他种类问题。

这里展示的创意被用来修改一个由 Pierre Terdiman 写的最优化碰撞检测库 OPCODE [Terdiman03]，以增加对缩放模型的支持。修改过的版本也支持非索引几何体，以及三角形扇、三角形条和表示地形的点网格。

2.6.5　参考文献

[Bergen97] Bergen, G. Van Den. "Efficient Collision Detection of Complex Deformable Models Using AABB Trees," *Journal of Graphics Tools,* Vol. 2, No. 4, pp. 1–13, 1997.

[Bloom06] Bloom, Charles. "Galaxy3."

[Cohen95] Cohen, J.D., Lin, M.C., Manocha, D., and Ponamgi, M.K. "I-COLLIDE: An Interactive and Exact Collision Detection System for Large-Scale Environments," *Symposium on Interactive 3D Graphics,* pp. 189–196, 1995.

[Gottschalk96] Gottschalk, S. "Separating Axis Theorem," Technical Report TR96–024, Dept. of Computer Science, UNC Chapel Hill, 1996.

[Lomont07] Lomont, Chris. "Floating-Point Tricks,"*Game Programming Gtms 6, ChaHes River Media. 2007.*

[Terdiman03] Terdiman, Pierre. "OPCODE."

[Wu91] Wu, Kevin. "Fast Matrix Inversion," *Graphics Gems II,* Academic Press, Inc., 1991.

[Wu94] Wu, Kevin. "Fast Inversion of Length- and Angle-Preserving Matrices," *Graphics Gems IV,* Academic Press, Inc, 1994.

2.7 三角样条

Tony Barrera, Barrera Kristiansen AB
tony.barrera@spray.se

Anders Hast, Creative Media Lab,
University of Gävle
aht@hig.se

Ewert Bengtsson, Centre For Image Analysis,
Uppsala University
ewert@cb.uu.se

现代游戏（包括 2D 游戏和 3D 游戏）在用户界面设计、关卡设计和人物设计时，美术家都希望创建一些完美圆弧或者椭圆弧的几何体。实际上，在某些情况下，游戏玩家也要求游戏引擎能够高效并无错地生成这类形状。有多种数学方法来生成弧线，每种方法都有各自的优缺点。如果开发人员能使用统一的方法来生成几何图形，而不是针对不同的基本图形使用不同的数学方法，那数字内容创建工具、关卡编辑器和游戏运行时的健壮性将会最大化。

本精粹将展示如何创建样条曲线。该样条可生成直线和完美的圆弧，后者是不可能通过普通的三次样条生成的。三角样条将使我们可以用新的形式来创建 3D 模型。此外，本精粹还将展示在内循环中所使用的三角函数，甚至可以不使用 sin 和 cos 函数就计算出来，使它的效率更高。

2.7.1 背景知识

三次样条不能用来生成完美的圆弧，而三角样条可以做到。三角样条由 Schoenberg[Schonberg64]发明，有时也被称为三角多项式。三角样条在数学和计算机辅助几何的文献中被广泛地研究，一些例子可以在[Lyche79]和[Han03]中找到。但是，三角样条之前并没有引起计算机图形社群太多的兴趣，可能因为它需要引入相对耗时的三角函数。但随着硬件的飞速发展，他们对计算机图形中建模工具这一领域越来越感兴趣，因为所有的模型都可从直线到圆弧直接生成。之后的一个小节将展示如何在不使用 sin 和 cos 函数的情况计算出三角多项式的结果，这样即使没有特殊的图形硬件支持，计算也能更加快速。

三角样条

一个三角样条[Alba04]可由一个截断的傅里叶级数来构建。Hermite 样条是由两个点以及在这些点的切线来定义的，如图 2.7.1 所示。

$$\begin{aligned}\boldsymbol{P}(0)&=\boldsymbol{P}_0\\ \boldsymbol{P}(1)&=\boldsymbol{P}_1\\ \boldsymbol{P}'(0)&=\boldsymbol{T}_0\\ \boldsymbol{P}'(1)&=\boldsymbol{T}_1\end{aligned} \tag{2.7.1}$$

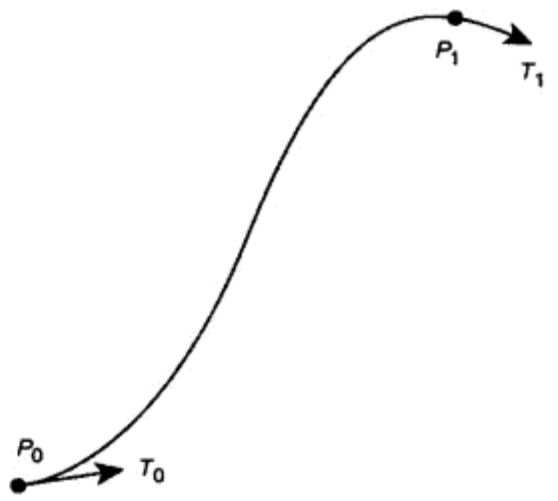


图 2.7.1　一条三角曲线和它的约束

因此，你需要傅里叶级数中的 4 个元素。这个三角曲线是定义在区间[0, π/2]上的：

$$\boldsymbol{P}(\theta)=\boldsymbol{a}+\boldsymbol{b}\cos\theta+\boldsymbol{c}\sin\theta+\boldsymbol{d}\cos 2\theta \tag{2.7.2}$$

这条曲线的系数可通过代入等式 2.7.1 得到，解以下方程即可：

$$\begin{bmatrix}1&1&0&1\\1&0&1&-1\\0&0&1&0\\0&-1&0&0\end{bmatrix}\begin{bmatrix}\boldsymbol{a}\\\boldsymbol{b}\\\boldsymbol{c}\\\boldsymbol{d}\end{bmatrix}=\begin{bmatrix}\boldsymbol{p}_0\\\boldsymbol{p}_1\\\boldsymbol{T}_0\\\boldsymbol{T}_1\end{bmatrix} \tag{2.7.3}$$

该方程的解为：

$$\begin{aligned}\boldsymbol{a}&=\frac{1}{2}(\boldsymbol{P}_0+\boldsymbol{P}_1-\boldsymbol{T}_0+\boldsymbol{T}_1)\\ \boldsymbol{b}&=-\boldsymbol{T}_1\\ \boldsymbol{c}&=-\boldsymbol{T}_0\\ \boldsymbol{d}&=\frac{1}{2}(\boldsymbol{P}_0+\boldsymbol{P}_1-\boldsymbol{T}_0+\boldsymbol{T}_1)\end{aligned} \tag{2.7.4}$$

还需要注意的是，三角 Hermite 样条也可以用以下形式表示：

$$\boldsymbol{P}(\theta)=\boldsymbol{A}\cos\theta+\boldsymbol{B}\cos^2\theta+\boldsymbol{C}\sin\theta+\boldsymbol{D}\sin^2\theta \tag{2.7.5}$$

注意，系数 A、B、C 和 D 与 a、b、c、d 是不一样的。证明如下。对等式 2.7.2，首先展开 cos2 θ 项，得到下式：

$$\boldsymbol{P}(\theta)=\boldsymbol{a}+\boldsymbol{b}\cos\theta+\boldsymbol{c}\sin\theta+\boldsymbol{d}(2\cos^2\theta-1)=\boldsymbol{a}-\boldsymbol{d}+\boldsymbol{b}\cos\theta+\boldsymbol{c}\sin\theta+2\boldsymbol{d}\cos^2\theta \tag{2.7.6}$$

然后将式 2.7.4 代入式 2.7.6 中，可得：

$$\begin{aligned}\boldsymbol{P}(\theta)=&\frac{1}{2}(\boldsymbol{P}_0+\boldsymbol{P}_1-\boldsymbol{T}_0+\boldsymbol{T}_1)-\frac{1}{2}(\boldsymbol{P}_0+\boldsymbol{P}_1-\boldsymbol{T}_0+\boldsymbol{T}_1)\\ &-\boldsymbol{T}_1\cos\theta+\boldsymbol{T}_0\sin\theta+2\frac{1}{2}(\boldsymbol{P}_0+\boldsymbol{P}_1-\boldsymbol{T}_0+\boldsymbol{T}_1)\cos^2\theta\end{aligned} \tag{2.7.7}$$

化简可得：

$$\boldsymbol{P}(\theta)=\boldsymbol{P}_1-\boldsymbol{T}_0-\boldsymbol{T}_1\cos\theta+\boldsymbol{T}_0\sin\theta+(\boldsymbol{P}_0+\boldsymbol{P}_1-\boldsymbol{T}_0+\boldsymbol{T}_1)\cos^2\theta \tag{2.7.8}$$

你可以将最后一项分为两部分，并使用三角恒等式改写其中一个，可得：

$$\boldsymbol{P}(\theta)=\boldsymbol{P}_1-\boldsymbol{T}_0-\boldsymbol{T}_1\cos\theta+\boldsymbol{T}_0\sin\theta+(\boldsymbol{P}_0+\boldsymbol{T}_1)\cos^2\theta+(\boldsymbol{T}_0-\boldsymbol{P}_1)(1-\sin^2\theta) \tag{2.7.9}$$

最后，化简可得：

$$\boldsymbol{P}(\theta)=-\boldsymbol{T}_1\cos\theta+\boldsymbol{T}_0\sin\theta+(\boldsymbol{P}_0+\boldsymbol{T}_1)\cos^2\theta+(\boldsymbol{P}_1-\boldsymbol{T}_0)\sin^2\theta \tag{2.7.10}$$

刚才展示了这个曲线的不同表示法，你可以使用等式 2.7.2 或等式 2.7.10。当然，你也可以使用等式 2.7.8。这种灵活性在计算函数输出时很有用，这在下一小节将会介绍。

三角函数快速求值

[Barrera04]介绍了一种向量和四元素的高效插值技术，主要的创意是使用球面线性插值（Spherical Linear Interpolation，SLERP）。SLERP 是由 Shoemake 引入到计算机图形社群的。SLERP 与线性插值的不同表现在确保每个向量和四元素之间的角度是常量；也就是说，移动速度是常量。两个正交单位向量的 SLERP 如下：

$$\boldsymbol{P}(\theta)=\boldsymbol{A}\cos\theta+\boldsymbol{B}\sin\theta \tag{2.7.11}$$

这种情况下，在插值的每一步都需要对 cos 和 sin 函数求值。[Barrera04]中展示了如何用 C++实现 k 步插值，代码如下：

```
#include <math.h>
#include <stdio.h>
#define M_PI        3.14159265358979323846
void main() {
    int k=10; // nr of steps
    double A[2]={1,0};
    double B[2]={0,1};

    double tm1[2];
    double t0[2];
    double tp1[2];
    double t=M_PI/2.0;
    double kt=t/k; // step angle
    tm1[0]=A[0]*cos(kt)-B[0]*sin(kt);
    tm1[1]=A[1]*cos(kt)-B[1]*sin(kt);

    t0[0]=A[0];
    t0[1]=A[1];

    double u=2*cos(kt);

    tp1[0]=t0[0];
```

```
        tp1[1]=t0[1];
        printf("%f %f\n", tp1[0], tp1[1]);

        for(int n=2; n<=k+1; n++) {
            tp1[0]=u*t0[0]-tm1[0];
            tp1[1]=u*t0[1]-tm1[1];
            printf("%f %f\n",tp1[0], tp1[1]);

            // switch
            tm1[0]=t0[0];
            tm1[1]=t0[1];
            t0[0]=tp1[0];
            t0[1]=tp1[1];
        }
    }
```

这段代码将 ***A*** 和 ***B*** 之间的 sin 和 cos 值打印出来，结果存放在变量 tp1 中。如果要计算两个任意向量之间的 cos 和 sin 值，可以使用 Gram Schmidt 正交算法计算出一个正交向量，详见参考文档。

2.7.2 讨　论

在等式 2.7.2 中强制 ***d*** 等于 0，得到：

$$\boldsymbol{P}(\theta)=\boldsymbol{a}+\boldsymbol{b}\cos\theta+\boldsymbol{c}\sin\theta \tag{2.7.12}$$

这明显是个表示椭圆的等式，这也就证明三角样条可用来创建完美的圆弧和椭圆弧。此外，因为这个曲线是参数化的，所以也可以用三角样条来创建直线。所有系数都是向量，而该函数输出一个空间中的点。每个坐标都有自己的表达式，唯一的不同是系数不同。因此就算使用三角函数，也可以创建一条直线。图 2.7.2 展示了一条由三角样条绘制的完美圆弧。

注意，在等式 2.7.4 中将 d 设为 0，$T_0+T_1=P_1-P_0$ 可得到同样的结果。

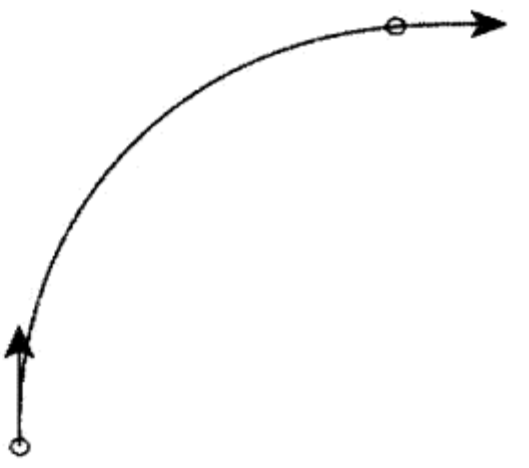


图 2.7.2　使用三角曲线绘制的一条圆弧

2.7.3 结　论

三角样条很有意思，因为它可以用来创建完美的圆弧。这种样条的本质让它可以创建普通三次样条无法实现的样条曲线和表面。当然，你也可以使用多段三次样条来近似地表示这个表面，但这样的话，曲率的波动造成的几何瑕疵在视觉上非常明显，而且无法接受。

虽然我们的讨论是纯理论的，但我们相信三角样条有很大的潜力去解决现代游戏开发中的一些特定问题。

三角样条一个很自然的应用领域就是数字内容创建工具。对于需要原始圆锥截面的游戏

几何体的建模问题，这些样条是简单完美的解决方案。

除了数字内容创建工具外，这些样条也将对游戏的实时环境做出贡献。我们期待新技术的出现，例如，使用三角样条来实现特定的游戏物理模拟。比如，使用在切线上完美贴合的多段弧形来模拟多种“机械”和复杂机器，这样在视觉上将无比优美，因为完全没有分段折线或者传统三次样条所引起的锯齿。

我们同样也相信极有可能将这项技术应用到多种程序建模、程序贴图和程序动画技术当中。这项技术相当高效，很可能适合在最新图形硬件的几何体着色器中实现。想象这无限的可能性，这可能会是一个游戏的全新时代，令人震撼的新世代！

2.7.4 参考文献

[Alba04] Alba-Fernandez, V., Ibanez-Perez, M.J., and Jimenez-Gamero, M. D. “A Bootstrap Algorithm for the Two-Sample Problem Using Trigonometric Hermite Spline Interpolation,” *Communications in Nonlinear Science and Numerical Simulation* (April 2004), Vol. 9, No. 2, pp. 275–286.

[Barrera04] Barrera, T., Hast, A., and Bengtsson, E. “Incremental Spherical Linear Interpolation,” *Proceedings of SIGRAD (2004),* Vol. 13, pp. 7–10.

[Han03] Han, X. “Piecewise Quadratic Trigonometric Polynomial Curves,” Mathematics of Computation (July 2003), Vol. 72, No. 243, pp. 1369–1377.

[Lyche79] Lyche, T. “A Newton Form for Trigonometric Hermite Interpolation,” *BIT Numerical Mathematics (June 1979),* Vol. 19, No. 2, pp. 229–235.

[Schoenberg64] Schoenberg, I. J. “On Trigonometric Spline Interpolation,” *Journal of Mathematics and Mechanics* (1964), Vol. 13, No. 5, pp. 795–825.

[Shoemake85] Shoemake, K. “Animating Rotation with Quaternion Curves,” *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH (1985),* Vol. 19, No. 3, pp. 245–254.

2.8　使用高斯随机性来拟真发射轨迹的变化

Steve Rabin, Nintendo of America Inc.
steve.rabin@gmail.com

无论是开枪或者是射箭，我们都有一种关于靶子上着弹点分布的经验和直觉。着弹点应该散落在中心周围，只有一些偏离得很远。这不是由 rand()产生的那类分布，而是一种特殊的随机性，通常可用一个钟形曲线来表示。幸运的是，我们有一些随机数生成器可产生这类高斯随机性。本精粹将讨论一个非常高效的高斯随机数生成器，并详述如何将它应用到发射轨迹的自然模拟上。

2.8.1　高斯分布

像 rand()这样的随机数生成器都可以产生均匀分布，这样在给定范围中任意一个给定数字都将以相同（或者均匀）的概率出现。比如，在区间[0, 1]，得到 0.3 和 0.5 的概率是一样的。而高斯分布有时也被称为正态分布，倾向于以 0 为中心附近的正负数。当这个分布的标准偏差为 1.0 时，我称之为标准正态分布，如图 2.8.1 所示。这个分布也经常因它的形状被称作钟形曲线。

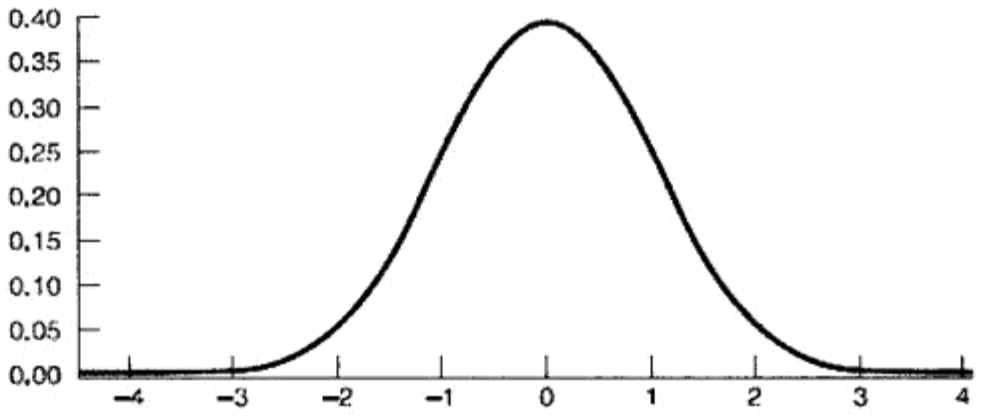


图 2.8.1　高斯分布（正态分布），以 0 为中点，标准偏差为 1.0。水平轴表示所生成的随机数值，而竖直轴表示任意特定值的出现概率，这个分布的尾巴是 3 个偏差之外的极小概率出现的数值（小于-3.0 或大于 3.0）

为更好地理解图 2.8.1，需要了解 68-95-99.7 法则。根据这个法则，68%的值将落在距离中点一个标准偏差的区间[−1, 1]，95%的值落在两个标准偏差区间[−2, 2]，而 99.7%的值落在 3 个标准偏差区间[−3, 3]区间。其余的 0.3%落在这个区间之外，而得到在[±]5.0 之外的数字只有少于百万分之一的概率。

现在你对高斯分布的形状有了一个更好的了解，让我们看几个可以产生高斯分布的随机数生成器。

2.8.2 生成高斯随机性

高斯随机数生成器（GRNGs）在统计分析中非常有用，所以一直在速度和质量上被深入地研究[Thomas07]。速度必须考虑，因为大型的模拟常常需要以 10 亿计的正态分布随机数。这些模拟通常用于通信建模和金融建模，也需要考虑到质量以及分布长尾（6 个标准偏差之外）的精度，理由是极低概率事件也可以影响这些模拟所要研究的重要特性的结果。

幸运的是，视频游戏并不需要这种级别的严密性。实际上，游戏会有一个相反的问题：GRNGs 不应该产生几率很小但极大或极小的值，因为这可能在玩家看来是个错误。比如说，如果树的高度是由一个 GRNG 来决定的，大多数树是 10～15m 高，而出现一个 30m 高的树就会很奇怪。因此，大多数情况下，你使用的 GRNGs 都是该丢弃（而不是取最大、最小边界值）3 个标准偏差之外的值。

高斯随机数生成器

一个非常流行的高质量 GRNG 是 polar-rejection [Knop69]（也被称为 Box-Mueller 变换的极坐标形式）。这个算法因被《Numerical Recipes in C》[97 年出版]一书采录而流行，并因为它最费时的操作只有一个对数操作和一个平方根操作而著名（它规避了最初 Box-Mueller 变换中所需要的 sin 和 cos 操作）。虽然这是一个很好的 GRNG，但是还有更快的算法。

Ziggurat 方法[Marsaglia00]是第二流行的 GRNG，并经常被列为在速度和质量的平衡上做的最好的算法[Thomas07]。它比 polar-rejection 算法要快，因为它使用了一个大小为 1KB 的查询表并且极少调用超越函数。但是，对于游戏开发来说，访问这些查询表会污染数据缓存，从而导致实际运行比使用 polar-rejection 反而更慢。因为在大多数平台上相对于 CPU 的速度，访问内存的代价很高。举例来说，当 Ziggurat 方法在高速执行，它的查询表就会将游戏程序中的其他数据挤出缓存，从而减缓整个游戏的速度，相对 polar-rejection 更甚。

以上两种方法对于游戏程序都不太合适，最佳的方法是一种简单高效的技术，被称为中心极限定理（central limit theorem），有时候也被叫做均匀数和（sum-of-uniforms）[Thomas07]。这个算法采用一些均匀随机数，比如，一些由一个像 rand()这样的 PRNG 产生的随机数，并将它们相加。根据中央极限定理，这些均匀随机数的和将产生一个高斯分布随机数。这个算法在分布的尾部表现很差，但这是可以接受的，因为你通常也不会对在 3 个标准偏差之外的数值感兴趣。

更精确地说，中央极限定理指出 K 个在区间[−1, 1]均匀随机数的和将逼近一个以 0 为中心、标准偏差为 $\sqrt{K/3}$ 的高斯分布。比如说，如果你将 3 个均匀随机数相加，它们的分布将以 0 为中点，标准偏差为 $\sqrt{3/3} = 1.0$（这样非常便利，因为中点和标准偏差与一个标准的正态分布一致）。以下的代码通过将 3 个 32 位的带符号随机数（通过一个非常快速的异或移位 PRNG [Marsaglia03]生成）相加来生成一个高斯分布。

```
double gaussrand(void)
{
```

```
  static unsigned long seed = 61829450;
  double sum = 0;
  for(int i=0; i<3; i++)
  {
    unsigned long hold = seed;
    seed^=seed<<13; seed^=seed>>17; seed^=seed<<5;
    long r = hold+seed;
    sum += (double)r * (1.0/0x7FFFFFFF);
  }
  return sum; //返回 [-3.0,3.0]
}
```

函数 gaussrand()返回一个区间[−3.0, 3.0]内的双精度数。如果想要一个在区间[−1.0, 1.0]内的数，只需简单地将结果除以 3.0（这会相应地将标准偏差收缩为 0.33）。这个分布大致遵守 68-95-99.7 法则，但由于分布的尾部丢失，这个特定算法（使用这个种子）的分布实际上是 66.7-95.8-100。

对更多的数值（增加 K）求和可使中心极限定理方法更为精确，尤其是在 3 个标准偏差之外的尾部。但是，这会使得算法更慢而又没有太大的好处，因为通常你也不关心尾部。

不同的发射轨迹

在游戏 GRNG 的一个理想应用就是为发射轨迹加上随机变动。如之前所述，发射物体，像子弹和箭都有遵循高斯分布的变动（这可能是由于许多随机因素像风、手的抖动和发射体的不规则，一起作用影响最终的轨迹）。但是，所用的高斯分布必须拓展到 2D，如图 2.8.2 所示。

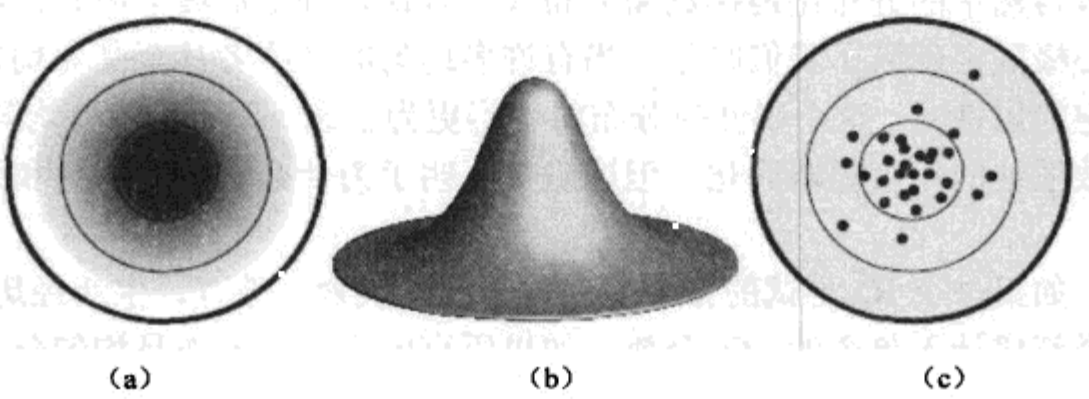


图 2.8.2　（a）展示了一个在 2D 中的高斯概率分布。最为形象的表示是（b），这是一个围绕着中心的高斯分布。（c）是一个 30 发子弹以一个高斯分布的扰动图。因为这些圆环是以一个和两个标准偏差距离摆放的，所以基本上 68%的弹点在最小的环内，而 95%的弹点在第二小的环内

图 2.8.2 中的分布是使用极坐标生成的。这样每个 2D 点就需要两个随机数：一个角度和一个距离。在图 2.8.2 中的子弹是这样产生的：先生成一个区间[0, 2π]内均匀随机的角度，然后生成一个区间[−1, 1]内的高斯随机数，取其绝对值作为距离。通过使用一均匀随机数作为角度，可以确保子弹均匀分布在中心周围的各个角度上。而使用一个高斯随机数作为距离，就可以确保子弹都集中在中心附近，遵守一个正态分布和 68-95-99.7 法则。

图 2.8.2 中的 2D 分布从技术上来说并不是一个 2D 高斯分布（也被称为多元正态分布）。一个真正的 2D 高斯分布是通过两个高斯随机数在笛卡儿坐标（而不是极坐标）下描点绘制

出来。这个分布在统计中很有用，但并不适合刚才的建模。

这类分布用于弹道建模的缺陷可在以下的例子中看到，如果 x、y 是两个高斯随机数，坐标（1.41, 1.41）统计上出现的概率将小于坐标（2.0,0.0），即使这两个坐标到原点是等距离的。因此，一个真正的 2D 高斯随机分布总是更倾向于坐标轴相对于对角线，这对于 2D 弹道分布是不合适的。

2.8.3 其他应用

高斯随机性不仅可应用于弹道，在其他游戏领域也是很有用的。比如说，如果有多个人物或者车辆在一起移动，可能会看见步调一致的移动，这时可以将各个个体的加速度、最大速度或者动画速度通过使用一个 GRNG 来扰乱开从而避免。这样就可以产生一些在平均值附近的小差异来破坏同步的移动或者动画。结果只会产生很小的差异，并且极少的例外。

GRNG 的另一个应用是用来扰乱各个人物、树或者建筑的高度。如果你通过算法控制游戏中的几何物体，那使用一个 GRNG 就可以产生很真实的差异。当游戏每时每刻都可以看见的很多物体，你又需要很自然的差异化时，这个方法就非常有用。通常，许多物理特性或者属性都是在一个平均值附近随机变化的。如果使用高斯分布来模拟，都会有较好的效果。

2.8.4 自然中的高斯分布

为什么许多自然中的分布都遵循高斯分布（或者说是钟形曲线）呢？比如人类智商或者树的高度？中心极限定理给了我们暗示。当有许多均匀的（或者甚至是非均匀的）随机因素作用于一个给定的属性时，这个属性的分布将变得更为正态（而不是保持均匀）。虽然这只是自然界中大多数系统的一个粗略简化，但这确实阐明了为什么这么多属性和系统大致展现出高斯分布。

举例来说，如果一个 IQ 测试的分数受遗传基因、饮食、教育、生活经历和环境的影响，这些因素每一个都将计入单个的 IQ 分数。如果所有的这些因素都是均匀分布且权重相同，中心极限定理指出其结果将是一个正态分布。当然，不太可能每个因素都是均匀的，而可能是其他一些随机因素的和，而影响它的这些随机因素又可能是受其他更多的随机因素所影响的。因此，许多影响 IQ 的随机因素比如饮食和教育可能已经遵循一个正态分布了。最后，你可以将自然界中的许多属性大致归结为许多小的、相对独立的因素叠加作用于一个给定的属性。

2.8.5 结 论

通过中心极限定理为游戏生成高斯随机性是相当简单和高效的。但是，许多游戏程序员甚至都不知道这类随机数生成器，因此，最大的挑战在于简单地将方法描述出来，并让开发人员了解到这个工具的存在。

许多倾向于具有正态分布的物理系统和特性都可以使用高斯随机性来建模。通过在极坐

标中组合均匀随机性和高斯随机性，像为弹道添加真实的变动这种应用可以轻松实现。

2.8.6　参考文献

[Box58] Box, G.E.P. and Muller, Mervin E. "A Note on the Generation of Random Normal Deviates," *The Annals of Mathematical Statistics* (1958), Vol. 29, No. 2, pp. 610–611.

[Knop69] Knop, R. "Remark on Algorithm 334 [g5]: Normal Random Deviates," *Commun. ACM*, 12(5), 1969.

[Marsaglia00] Marsaglia, George, and Tsang, Wai Wan. "The Ziggurat Method for Generating Random Variables," *Journal of Statistical Software*, Vol. 5, 2000.

[Marsaglia03] Marsaglia, George. "Xorshift RNGs," *Journal of Statistical Software*, Vol. 8, 2003.

[Press97] Press, W.H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. *Numerical Recipes in C*, 2nd edition, Cambridge University Press, 1997.

[Thomas07] Thomas, David B., Leong, Philip G.W., Luk, Wayne, and Villasenor, John D. "Gaussian Random Number Generators," *ACM Computing Surveys 39,* 2007.

第 3 章

人工智能

简　介

Brian Schwab

每年都会有新的游戏推出，游戏程序员们迫不及待地想要看到他们可以做些什么来使游戏实现更多的效果。是否可以做到照片级的真实感渲染？所有的物理交互都将用逼真的物理建模实现吗？游戏音效是否让你感觉自己离游戏中的情节只有六码近，并且心跳随着音乐加速？最后，也是对于《游戏编程精粹 7》中的本章最为重要的，游戏中的角色能有人类大脑的哪怕一半那么聪明吗？

人工智能是游戏最难处理的部分之一。的确，艺术和音乐是非常主观的，但是对人类来说，智能完全是另一个层面上因人而异的东西。让我们看一个本能反应的例子。你正沿着街道行走，一只棕色的成年雄狮从你前方的那棵树后面缓缓走出。那做什么才算是智能的事情？大部分人可能会说出下面这些：

（1）跑；

（2）等待并看看将要发生什么；

（3）慢慢地向后走，但不做出突然的行动。

但是，任何这样的问题也可能产生很多不寻常的反应：

（1）找些东西来防卫自己；

（2）招一辆出租车赶紧离开这里；

（3）把你硕大的皮质钱包扔向那只狮子，希望狮子会停下来吃它；

（4）用最大的声音叫喊，把它吓跑；

（5）用胡椒水喷它。

哪一种选择才算正确呢？答案可能很简单，就是它们全部，但是在实际生活中，大概 70%的人在这种情况下都会像车头灯照射下的小鹿一样迈不出一步。我们之所以天生地就这么站着一动不动，主要是因为在进化过程中，我们学到了大多数猎食者有着基于动作的视线（意味着相对于颜色或者形状等其他视觉刺激来说，他们对动作更加敏感）。如果非洲平原上的狮子和鬣狗在它们的进化史中得到了基于颜色的视线反应，那人类今天也许会像变色龙一样变色，也就是说，如果你是一个纹身艺术家，你就会成为一个悲剧。

显然，我们不能在游戏中模拟那种停顿，如果你拿着枪走过街角，很有可能所有人都被吓住不动了，但在游戏中，玩家可能会觉得游戏出了什么问题。然而，如果所有人都跑开，游戏将会很快变得单调乏味（虽然一些游戏仍然在使用这种方法）。如果每个人都准备好了他们自己的武器，玩家将会得到一个相当震撼的场景。然而，这些方式的组合可以起到很好的效果，你甚至可以调整组合的方式，来得到你所想让玩家克服的挑战级别。

问题在于，你不能总是把你游戏中的智能行为建立在现实之上，也不能把它们建立在我们中任何一个人认为是正确的行为之上。因为人们对什么是智能反应的观点各不相同。只有持续对新技术的研究，你才可以指望从创造中表达出一点所谓的“智能”。

本章将介绍人工智能的发展程度。我们不再像开发社区那样只关心琐碎的人工智能实现方法，而是深入研究更加真实的感知模拟，用新的编程技术来简化人工智能系统，给智能角色赋予真正的人类特征，并在一个统计趋势层面上分析人工智能。

John Harger 和 Nathan Fabian 编写的文章讲述了如何使用一种叫做行为克隆的监督学习技术，它可以被用来捕捉人类的行为。Steve Rabin 和 Michael Delp 详细描述了一种统一的感知模型，你可以用一种美妙且系统的方法，有效率地对大部分现实进行建模。Iskander Umarov 和 Anatoli Beliaev 解释了如何使用泛型编程来创建并管理庞大且复杂的人工智能系统，其代码十分简练、快速且健壮。Michael F.Lynch 带来一篇详细的文章，介绍了在人工智能代理中对态度进行模拟。G. Michael Youngblood 和 Priyesh N.Dixit 描述了先进的玩家日志分析，可以用来发现一些平常只通过粗略观察很难预见的游戏玩法和玩家交互的有趣模式。Michael Dawe 向读者展现了如何使用计划合并来增强你的计划系统，增加系统的反应能力，而不会造成重新计划的开销。最后，因为也许你从来不会厌倦听到有关改善寻路算法的用法并理解其方法，所以 Robert Kirk DeLisle 介绍了一种叫做 fringe 搜索的对 A*算法的深入见解。

3.1 用行为克隆创建有趣的代理

John Harger
Nathan Fabian

人类的对手十分有趣。网络游戏的流行可能和你在那里所能找到的对手有关（或者大部分是因为你可以击败一些新手），但抛开它流行的原因，如果能把一些生命赋予离线的单机游戏对手（或者同盟）也不会有什么坏处。本精粹解释了如何使用一种叫做行为克隆[Sammut92]的机器学习技术来学习人类玩游戏的策略和行为方式，并把它们赋予游戏中的代理。

行为克隆从本质上说是一种监督学习技术。在监督学习中，基本的理念是：人类训练者为特殊的对象提供一系列基于这些对象属性的标识以及一个算法，来学习如何识别、预测这些标识。当该算法在一个新的对象上看到相似的特征时，就应该给予正确的标识。在行为克隆中，训练者对刺激做出反应。反应行为成为“刺激”这个对象的关联标识。然后算法学会在出现同样刺激的情况下重复做出同样的行为。

这种概念可以很好地扩展到游戏中，因为在游戏中很容易找到一个人对刺激的反应，这里的刺激也就是游戏状态（在游戏中，几乎所有的交互都在游戏背景中发生，并且每个游戏分段都完全和上一个相似。但是还有一点重要的是，有时游戏可以包含一些像家族成员或者敌对关系等额外的信息，这是代理所不能模拟的）。

因为在监督学习和行为克隆之间有着太多的相似，这种技术可以使用任何现成的监督学习算法实现。本精粹使用的是决策树算法，因为其输出比神经网络的权重更容易修改。

换句话说，你不需要成为一名机器学习专家，就可以用这种技术来开发更多的东西，甚至游戏玩法的训练者为了效率都不需要知道机器学习正在进行。本精粹将向你展示当借用人类特征来创建有趣并可玩的游戏代理时，这种行之有效的决策树学习算法表现得多么出色。

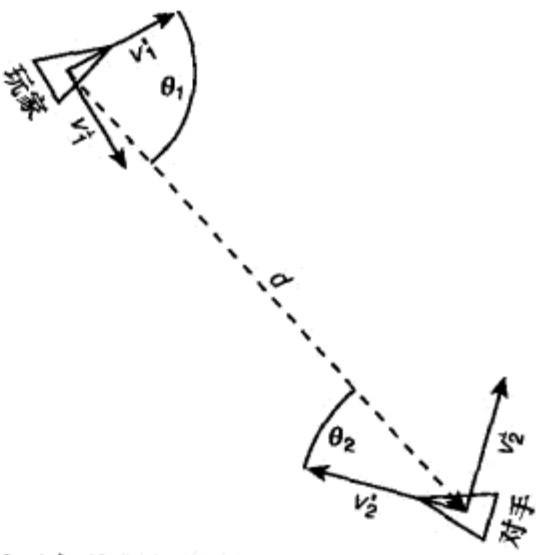
3.1.1 实例：The Demo Game

这篇文章中的范例参考了类似经典电脑游戏 Space War 的游戏设计。两条飞船在无限的二维空间中准备好开始战斗，游戏的目标就是摧毁另一条飞船。这个简单的设计为训练一个代理提供了简单易懂的游戏环境。

如何设置特征空间输出

在基于实例的机器学习中，特征空间是最重要的，因为你要使用它来做行为克隆。实际上，特征空间是用来记录游戏状态实例的一套属性或者特征（见图 3.1.1 和表 3.1.1）。你必须小心地考虑这些特征的设计，使它能够训练出有趣的代理，或许能够完全地训练出一个代理。

这些特征必须提供足够的信息来做出决策。记住，你的目标是训练一个代理，使它的行为像一个特殊的人类。试着思考你会怎么处理情况，你也许会考虑与对手飞船的距离，基于你朝向他的飞船的方位，甚至基于他朝向你的飞船的方位。不要害怕一直在添加特性，如果他们的对手从雷达上消失，一些玩家可能就会表现得不同，而这些区别将会提供比距离更多的信息。也许用详细的特征来训练一个代理会花费学习算法 30 分钟之久，但它将会提供一个更智能的代理。



θ_1—从玩家到对手的角度
θ_2—从对手到玩家的角度
d—两艘飞船的距离
v_1—玩家飞船的方向向量
$v_1^\perp$—与玩家方向成直角的向量
v_2—对手飞船的方向向量
$v_2^\perp$—与对手方向成直角的向量

图 3.1.1 特征空间的例子

然而，你应该注意，不要在信息中使用一些绝对值，比如位置或者方向。如果代理被训练成基于绝对方向来加速，那么得到的行为也许会与期望的相反。比如你的飞船在点（2, 1），而敌人在点（0, 0），你可能会向右转。但如果代理按照位置点来学习，当它的对手在（0, 0）时，它就会总向右转，即使训练者已经朝左转。如果特征是相对的，比如向左 15°（±2.5°），当对手正好在左边时，代理就会向右转，而不会去考虑绝对位置。这样想，当你在加入一场斗殴时，你会考虑你的绝对方向和经纬度么？如果你想让你的代理表现得更真实，同样也不要那样做。

另外，对于不规则行为来说，绝对数值会导致一个过大的而难以处理的搜索空间。如果位置的范围是无限的，学习算法也许永远不能正确地对行为进行归类，这样，你会得到一堆杂乱的数据，而不是一棵紧凑且规划得很好的树。

表 3.1.1 计算在图 3.1.1 中的特征的公式

名称	公式	描述
Distance	$\lvert\vec{d}\rvert$	两艘飞船之间的距离
DirectionTo	$\theta_1 = \tan^{-1}\dfrac{\vec{d}\cdot\vec{v}_1^{\perp}}{\vec{d}\cdot\vec{v}_1^{\parallel}}$	从飞船 1 的朝向到飞船 2 的角度（-180° ～180° ）
DirectionFrom	$\theta_2 = \tan^{-1}\dfrac{\vec{d}\cdot\vec{v}_2^{\perp}}{\vec{d}\cdot\vec{v}_2^{\parallel}}$	从飞船 2 的朝向到飞船 1 的角度（-180° ～180° ）
Vnorm	$\lvert\vec{d}\rvert_t - \lvert d\rvert_{t-1}$	飞船距离的改变
DdirectionTo	$\theta_{1t} - \theta_{1(t-1)}$	DirectionTo 的变化
DdirectionFrom	$\theta_{2t} - \theta_{2(t-1)}$	DirectionFrom 的变化

训练一个代理

现在到了有意思的部分了：训练代理。这几乎可以通过任何可用的机器学习技术来完成。当开发自己的游戏时，可以随意地试验已有的实现或者编写自己的算法。如果对机器学习有兴趣，可以看看[Witten99]。

我们选择创建自己的简单的决策树实现。当然，它不是最佳的，因为它不能表现线性回归，并且只能使用预估值。因为预先定义的线性范围必须要映射到整数上，所生成的决策树就可能不像别的方法那样与数据配合得那么紧密。结果就是，如果 0.5～1.0 的距离是一个预定义的区间，一个训练者准备在 0.54 反应而另一个在 0.98，那么它们都会在距离 1.0 处发生反应，从而给两个代理相似的感觉。

当运行光盘中的 AIShooter 时，游戏每一秒会记录 100 次状态。这些状态被填入到之前定义的特征空间中：两艘飞船之间的距离、它们之间的相对方向等。另外，玩家的控制状态也会被记录，比如向左右转向、向前向后加速，并且考虑到人类的反应时间而对这些控制补偿 150ms。这就是构建决策树所需要的刺激或反应的信息。

使用采样来构建树

当你已经记录了游戏的一小部分间隔时（见表 3.1.2），可以用那些信息来构建决策树。不同的控制组被不同的树分离。前进、倒退和空操作会是一棵树可能的决策，向左、向右和空操作会是另一棵树可能的决策。树中的节点会测试一个特征所承担的全部数值。对于每个特征，承担的数值都会有一个子节点和一条在树中向下的路径。每一个子节点都从它的父节点继承所有和那条特征路径的数值相关的数据。子节点会递归地决定这些数据是否是纯净（pure）的（全部相同的）。

表 3.1.2　　记录游戏状态数据的采样

距离	来的方向	单击	转弯
“2 to ∞”	1	100	右
“0 to 1”	1	100	左
“0 to 1”	1	100	左
“1 to 2”	1	100	空操作
“0 to 1”	3	100	空操作
“2 to ∞”	2	100	左
“1 to 2”	2	100	空操作
“2 to ∞”	3	100	右
“0 to 1”	3	100	右
“2 to ∞”	2	100	左
“0 to 1”	3	100	右

现在有足够的数据可以用来做出决策并为某个特征做搜索。如果没有，叶子节点会决策哪个标记（那就是前进）是主要数据，并把它设置为返回值。清单 3.1.1 展示了节点学习的递归算法。

清单 3.1.1 递归的节点学习算法

```
TreeNode* DataSet::learnNode (const string& targetName,
      const string& columnName, const col_t& column,
      const col_set_t& workingSet, unsigned int threshold)
{
   col_t newTarget = getColumn (targetName, workingSet);
   int majority = for_each (newTarget.begin (), newTarget.end (),
       Majority ());
   float purity = (float) count (column.begin (), column.end (),
       majority) / (float) column.size ();
   if (column.size () <= threshold || purity >= 0.99f) {
      return new TreeNode (majority);
   }
   int max = *max_element (column.begin (), column.end ());
   TreeNode *node = new TreeNode (columnName, 0, max);
   for (int i = 0; i <= max; i ++) {
      col_set_t newWorkingSet = getAllWhere (columnName, i, workingSet);
      newWorkingSet.erase (columnName);
      if (newWorkingSet.empty ()) {
         continue;
      }
      pair<string, col_t> best = getBest (targetName, newWorkingSet);
      node->addChild (i, learnNode (targetName, best.first,
         best.second, newWorkingSet, threshold));
     }
   node->fillIn();
   return node;
}
```

为了决定为节点使用哪一个特征，你必须找到一个可以得到最一致数据的特征。因为决策树会一直寻找一个对给定状态来说有意义的标记，你会想要找到描述那一个标记的一系列状态，这是决策树通过对那些符合标记的状态的特征进行编码所做的核心工作。理想状况下，当树处在叶子节点时，就像之前所说的，对于每一个状态只有一个标记。

举例来说，假设你要从表 3.1.2 中训练一棵决策树来决定是否转向，而你的特征空间定义了 3 个整数的距离：0to1、1to2、2to∞。让我们从信息获取方面来看看距离特征。首先，来看公式 3.1.1，用一系列的数据来计算信息。

$$Info(V) = \frac{\left(-\sum_{v \in V} v \log_2 v\right) + n \log_2 n}{n}, n = \sum_{v \in V} v \qquad (3.1.1)$$

无需在信息理论的意义上获取更多细节，这里的目的是跟踪数据的差异程度。信息越少越好，因为那意味着标记更一致。例子中有 3 个空操作、4 个向左的操作和 4 个向右的操作，所以得到信息 $Info([3,4,4]) = (-3\log_2 3 - 4\log_2 4 - 4\log_2 4 + 11\log_2 11)/11 = 1.5726$。

如果你只去看距离为“1 to 2”的那些标记，就只得到两个都为空操作的实例，那得到的信息为 $Info([2]) = (-2\log_2 2 + 2\log_2 2)/2 = 0$。也就是说，在这个集合中得不到信息是因为它们都是一样的。

“0to1”的信息为 Info([1,2,2]) = 1.521 9。“2 to∞穷”的信息为 Info([2,2]) = 1。注意，这些信息都很接近原来的信息，因为数值的分布十分相似。

现在你发现获取值为 1.572 6−3/11*0−4/11*1.521 9−4/11*1=0.655 5。因为没有发生任何变化，所以很容易地发现生命值的获取值为 0。使用方向的获取值为 0.481 6。当你使用距离时，在那些例子中获得一致的结果确实对获取值有帮助。

$$Gain(O;F) = Info(O) - Info(O|F) \tag{3.1.2}$$

在 3 个新节点的每个节点中，你传递与那个距离数值有关的数据的子集并只使用它来递归。没有理由在第二次传递中使用距离（不会再改进信息），所以方向变成了显而易见的选择，因为生命值始终没有提供有关标记的信息。见图 3.1.2 中的决策树，再次注意到，对距离“1to2”来说，因为结果已经是纯净的，所以只需要返回标记，而没有理由去继续递归。

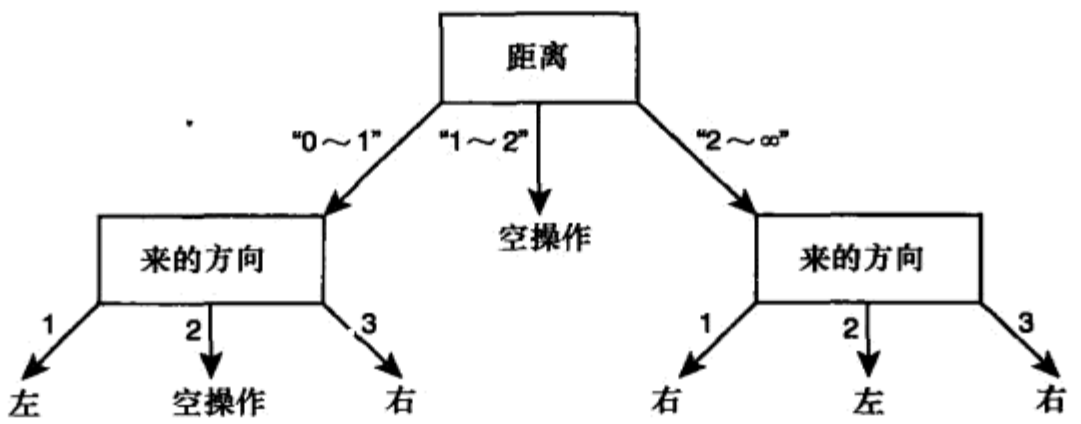


图 3.1.2　决策树的例子

你通过寻找集合中的大多数数值来选择或控制标记。注意有一点很重要，为了使大多数数据在统计中有效，你必须保证有足够的样本来让数据有意义。在清单 3.1.1 中的 LearnNode 函数，你注意到会有一个列检查来保证至少有若干的行，这就是机器学习中的“没有免费的午餐”规则。总会有一个空闲的参数并且经常依赖于数据。我们使用 250，是因为我们设想在游戏中 2.5s 足够抵消任何意外的被认为是噪声的按键事件。另外，我们对游戏状态中的控制补偿 150ms 来模拟人类的反应时间。

从决策树中建立 AI 脚本

当决策树建立好以后，需要把它们转换为游戏中可用的形式。虽然可以用树解释器，一般来说是足够快速和有效率的。在这个例子中，我们还是把它翻译成脚本的形式，使之可读并能够被手动编辑。所以你真正必须要做的是：使用条件命题来将决策树转换为你所选择的脚本语言。比如，在图 3.1.2 中的转向树被写成 Lua 脚本将会如清单 3.1.2 所示（注意，为了说明方便，我们在一些地方使用 strings，而真正的脚本会使用 numbers）。

清单 3.1.2　转向树的 Lua 脚本

```
function turn
   --根节点
   if GameState.Distance == "0 to 1" then --左分支
      if GameState.DirectionFrom == 1 then
         Ship.turn ("LEFT")
      elseif GameState.DirectionFrom == 2 then
```

```
            Ship.turn ("NONE")
        elseif GameState.DirectionFrom == 3 then
            Ship.turn ("RIGHT")
        else
            Ship.turn ("NONE")
        end
    elseif GameState.Distance == "1 to 2" then --中间分支
        Ship.turn ("NONE")
    elseif GameState.Distance == "2 to inf" then --右分支
        if GameState.DirectionFrom == 1 then
            Ship.turn ("RIGHT")
        elseif GameState.DirectionFrom == 2 then
            Ship.turn ("LEFT")
        elseif GameState.DirectionFrom == 3 then
            Ship.turn ("RIGHT")
        else
            Ship.turn ("NONE")
        end
        else
        Ship.turn ("NONE")
    end
end
```

它的意义就在这里，当代理正在运行时，你在设置游戏状态之后执行这段脚本函数，决策树就会做它自己的工作，代理就会以近似于训练它的人类的方式来对状态做出反应。

3.1.2 结论

这篇文章阐述了一种十分简明并且方便的方式，来让代理在一个简单的游戏中学习人类的行为。使用这种技术最好的地方可能会是创建那些支持作用的角色，比如警卫，他们通常只会有有限的行为，但是会从引入的多样性中受益，特别是在他们对玩家的反应方式方面。

我们没有涉及的一些细节包括扩展特征空间到更多的对手或者是障碍。你可以想象，如果给每一个敌人或障碍添加距离和方向，开销将会真正地飞速增长。如果要考虑添加同盟到敌人的距离和方向，将会变得更糟糕，这将会是一个完全连通的图。解决方法是给这些物体分组，并且把它们看做是一个拥有单一距离和方向的整体。

这些代理对时间的消逝缺乏足够的理解。它们不直接感知时间，而是感知对飞船的伤害，而伤害是随着时间降低的。但是，如果飞船被修理而恢复到一个水准时，他们不会有对一开始受到伤害的记忆。你可以考虑将时间作为一个特征而添加进来，但是添加绝对时间会像添加绝对位置或方向一样出现问题。然而，当你玩光盘中的 demo 时会发现，要获得很好的行为，并没有必要去拥有那样的记忆。

“天下没有免费的午餐”，你不能用这种算法创造一个游戏结局中的超级恶棍，但你可以让他的手下有更多样化的行为。我们已经建立了一个论坛，上面有一些创建更复杂行为和处理更复杂游戏的问题和方法。那里还有一些链接，深入探讨了为什么这种算法可以起作用以

及它的理论背景。我们十分乐意你能加入进来分享你的经验！

3.1.3　参考文献

[Sammut92] Sammut, C., Hurst, S., Kedizer, D., and Michie, D. "Learning to Fly," *Proceedings of the Ninth International Conference on Machine Learning*(ICML–1992), Aberdeen: Morgan Kaufmann, pp. 385–393.

[Witten99] Witten, Ian H., and Frank, Eibe. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations,* Morgan Kaufmann, 1999.

3.2 设计一种真实且统一的代理感知模型

Steve Rabin, Nintendo of America Inc.
Steve.rabin@gmail.com
Michael Delp, WXP Inc.
michaeljdelp@gmail.com

随着虚拟现实的迅速发展，玩家们期望代理可以更逼真、更精细地感知游戏世界。然而，游戏中的代理感知模型一直就是十分简单的，通常使用视距、视锥和视线检查的组合[Rabin05]。听觉模型的实现一样也很简单，通常通过测试一些截断距离来验证声音是否被听到[Tozour02]。虽然这些基本的代理感知模型十分高效并且易于编写，但它们对游戏玩家来说太过透明并且太过肤浅。举例来说，当代理使用离散的距离来检查视线时，在超出一个特定的距离之后将会有绝对的盲区。玩家熟悉这个漏洞后将会用它来操纵敌人的 AI。这个很常见，玩家通过重复缓慢向前再跑开来从敌人群体中引诱单独的敌人。

当开发者发现当前的代理感知模型过于简单时，很多更聪明的改进基本模型的方法开始出现。这篇文章提出了很多这样的改进，因为所有的感觉应该合作地形成一个代理对世界的感知，所以最后它们被合并成一种统一的感知模型。最后的模型可能会作为 AI 的核心部分，可以在任何游戏类型中使用。

3.2.1 基本视觉模型

在这篇文章开始关注特殊的视觉改进前，本小节会扼要地重述一下现在大多数游戏中使用的核心视觉模型。3 种核心视觉计算分别是视距、视锥和视线。注意，为了效率，这 3 种检查通常会按这个顺序来计算。这是因为范围测试十分廉价，而视线测试花费会比较昂贵。图 3.2.1 阐明了这些测试。

视距检查的计算是非常简单的距离测试。然而，测试距离的平方而不是真正的距离会更有效率，因为它避免了平方根操作。举例来说，如果一个代理最远可以看到 10m，它的坐标位置是(0,0,0)，而玩家在点(5,8,0)处，那么你就可以计算向量的点乘来和视距的平方做比较。向量的点乘结果是 $5^2+8^2+0^2=89$，和视距的平方 $10^2=100$ 做比较，因为 89 小于 100，你会发现代理可以看到玩家。之所以可以应用距离平方优化，是因为你只关心相对的距离，而不是确切的距离。

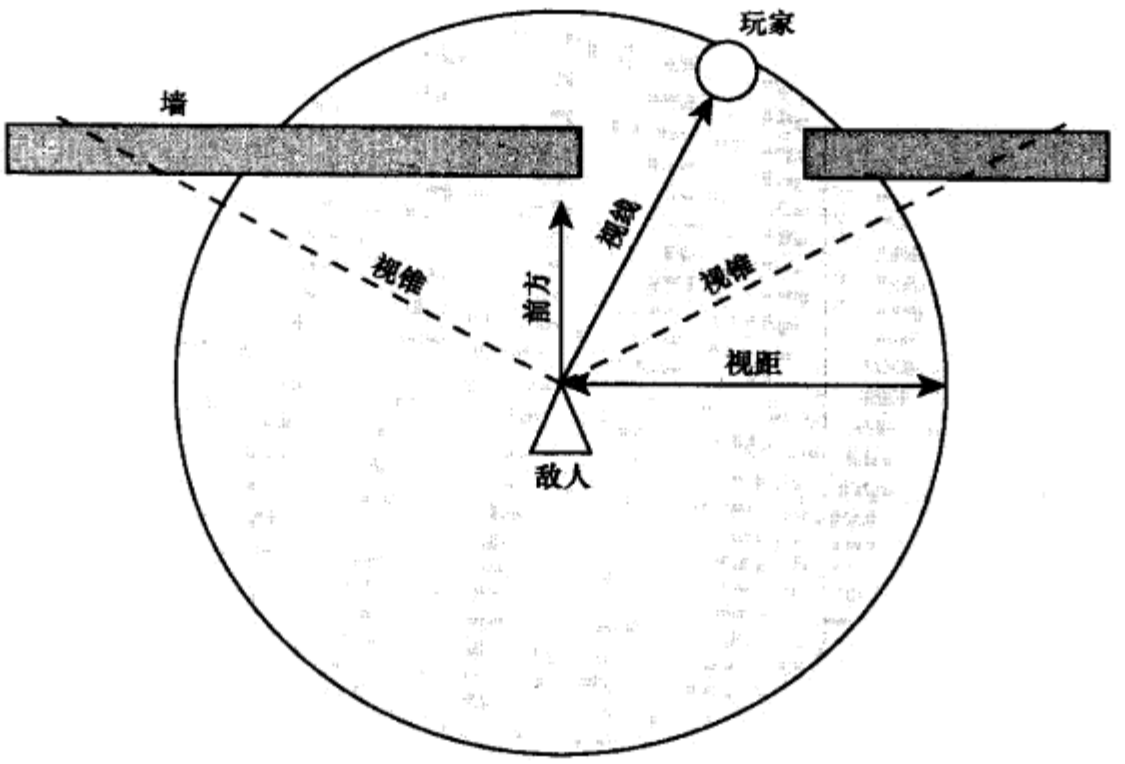


图 3.2.1　视距测试、视锥测试和视线测试的例子。这里因为玩家通过了所有的测试，所以敌人看到玩家了

第二个通常的步骤是做视锥测试。先将代理的向前向量归一化，同样将从代理到玩家的向量归一化（参考图 3.2.1 中的两个向量），然后对它们进行点乘操作。如果结果大于 0，玩家就在代理的 180°角的视锥中。如果结果大于 0.5，玩家就在代理的 120°角的视锥中（cos60° =0.5）。作为优化，如果只需要测试 180°角的视锥，就没有必要归一化向量了（可能会排除掉两个平方根计算）。

最终的测试，也就是视线测试，执行起来是最耗时的。测试会从代理的眼睛的水平高度向玩家的位置射出一条射线。如果在它碰到玩家之前碰到了任何几何体，代理就无法看到玩家。这种测试可以通过使用关卡几何体的包围盒来进行优化，而不是对单独的多边形进行测试。为了测试世界中的物体，物体最低级别的 LOD 和包围盒一样，也是有用的。

这 3 种测试为视线模型奠定了基础，但稍候你将看到我们可以在这之上应用很多改进。

3.2.2　基本听觉模型

模拟代理听觉的游戏，比如重视秘密行动的游戏，会通过让物体发出可以穿越特定距离的单发声音事件来模拟听觉。举例来说，玩家的每一个脚步声都会向离玩家一定距离内的代理发送一个声音事件。声音事件穿越的距离依赖于脚步声的大小，而这通常又会关联到玩家的速度。踮着脚走的玩家会发出很微弱的、只能传播很短距离的声音，而跑动中的玩家会产生很强的、可以传播很远距离的声音。

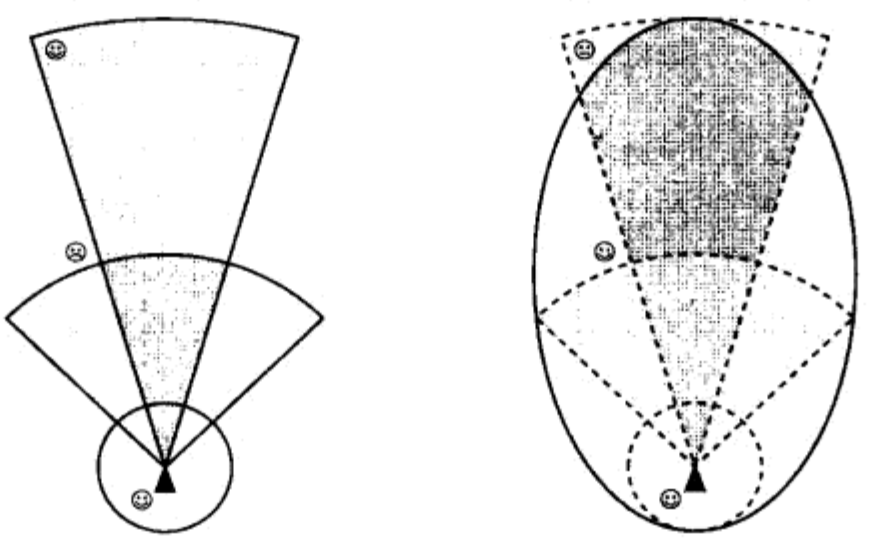
对很多游戏来说，这种听觉模型已经足够了。但它也有和视觉模型一样的问题，就是绝对和主观的截断距离。一厘米的距离差异就可以让人完全听到或者完全听不到，看起来确实十分奇怪并且不自然。同样，你可以猜到，有很多的改进可以被应用到这个基本的感知模型中。

3.2.3 用椭圆扩充视觉模型工具箱

之前讨论的简单视觉测试并不能很好地为人类视觉建模，特别是视锥有很多缺点。

- 代理不可能看到它正右方的物体。
- 视觉敏锐度在视野的中心最高并随着距离衰减。视锥模型过分地估计了远处的视觉区域，而对近处的视觉区域估计不足。
- 为了避免视觉远处过大的区域，设计者会让视距不真实地变短。

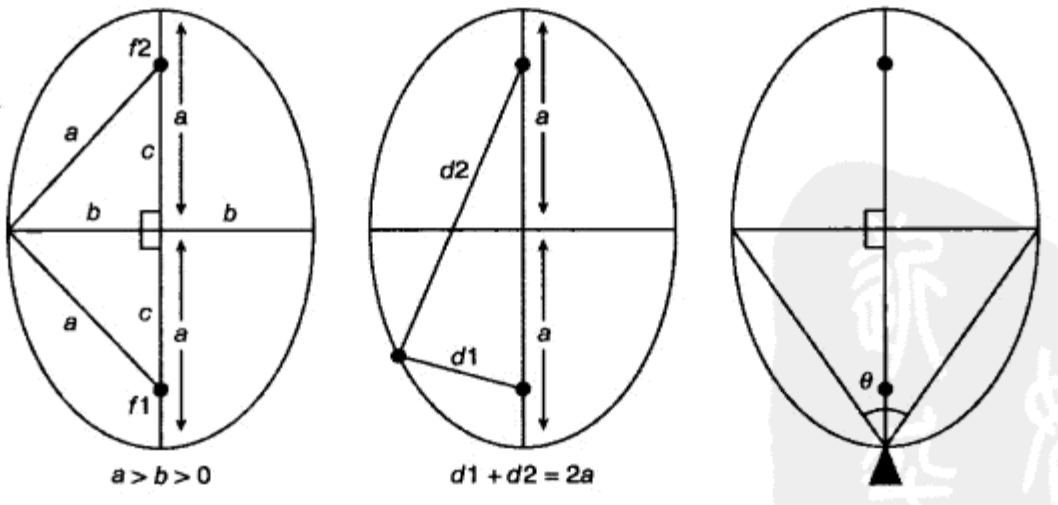
设计者为了应对这种问题而提出的一种方法是用多个视锥来模拟人类视觉[Leonard03]。但是，多个视锥会在代理的视野上留下漏洞。图 3.2.2（a）展示了使用两个视锥和一个圆来模拟视野。较窄的锥体模拟焦点中心，并可以扩展到远距离；较宽的锥体为近距离提供了更宽的视觉范围；圆周会捕捉任何靠近代理的物体甚至是在它身后（因为人类对正好在他背后的人有感知）。注意，这种视觉模型在两个锥体的交叉部分之外有很大的缺口。



（a） （b）

图 3.2.2 （a）表示使用两个视锥和一个圆的视觉模型，注意视觉系统中的漏洞；
（b）表示使用一个椭圆覆盖以前的模型。椭圆优美地包裹了不同的视角，从而给出更加准确的视觉模型

一种可以解决这些问题的简单方案是使用椭圆来模拟视野。如图 3.2.3（c）所示，一个椭圆可以优美地模拟视觉敏锐度随着距离的衰减而不留下任何视觉缺口。椭圆会从代理身后的几英尺开始，来模拟人类感知在身后的人的第六感，而且也囊括了接近代理的物体。



（a） （b） （c）

图 3.2.3 （a）展示了椭圆的重要参数；（b）展示了如何计算椭圆的示例点；
（c）展示了一个代理的例子视野，其中[θ]是视角的一半，a 是最大视距的一半

椭圆的实现

为了用椭圆来模拟视觉，理解它的参数是很重要的。如图 3.2.3（a）所示，长轴长为 $2a$，短轴长为 $2b$，两个焦点（$f1$ 和 $f2$）离椭圆中心的距离是 c 和 $-c$，而 $c^2=a^2-b^2$。图 3.2.3（b）表明了椭圆的一个重要特性。椭圆上的任意一点到两个焦点的距离之和必等于 $2a$。为了检查某个点是否在椭圆内，你必须先找到焦点的位置。

为了模拟人类视觉，你要将椭圆的一端放置在代理的眼睛位置，设计者接着可以指定视锥的角度。视角、代理的眼睛以及椭圆中心点可以确定一个三角形，如图 3.2.3（c）所示。设计者同样可以指定一个最大视距，视距的一半就是代理到椭圆中心的长度，所以给定[θ]是视角的一半，而 a 是视距的一半，你必须先找到 c、θ和 a 之间的公式。

$$\tan\theta = \frac{b}{a} \tag{3.2.1}$$

$$a\tan\theta = b \tag{3.2.2}$$

$$c^2 = a^2 - b^2 \tag{3.2.3}$$

将公式 3.2.2 代入到公式 3.2.3 中，得到以下公式：

$$c^2 = a^2 - (a\tan\theta)^2 \tag{3.2.4}$$

$$c^2 = a^2(1-\tan^2\theta) \tag{3.2.5}$$

$$c = a\sqrt{1-\tan^2\theta} \tag{3.2.6}$$

公式 3.2.6 可以在初始化时就进行预计算。下面来找焦点的位置。代理的眼睛在 vPos，看的方向为 vDir，公式如下：

$$F1, F2 = \text{vPos} + \text{vDir}(a \pm c) \tag{3.2.7}$$

如果你想让椭圆在人物后面 fBehindDist 处开始（这样，人物可以感知在他身边的人），最终的公式如下：

$$F1, F2 = \text{vPos} + \text{vDir}(a - \text{fBehindDist} \pm c) \tag{3.2.8}$$

当然，为了节省多余的减法操作，fBehindDist 可以在初始化时就从 a 中减掉。

为了检测一个物体是否进入代理的视野内，只需要计算物体到两个焦点的距离并相加，并检查其结果是否小于最大观察距离（$2a$）。所以对于每个物体，每个代理只需要做两次距离检查。记住，你不能在这个公式中使用距离的平方，因为你需要把两个距离加起来。这些公式可以应用在 3D 或者 2D 的椭圆上。如果高度对你的游戏世界很重要，就使用 3D 的。

使用椭圆来模拟视觉易于计算，而且不会比视锥方案昂贵很多。这是提供更准确的人类感知模型的第一个组成部分。

3.2.4　用确定性模拟人类视觉

如前面所述，离散视觉测试方法的副作用是物体会完全可见或完全不可见。当你考虑到真正人类视觉的细微差别时，离散测试的缺点就十分明显了，比如周边视觉，物体有时只会被部分地识别。为了更准确地理解这个问题，让我们快速回顾一下真实人类视觉的原理。

从视网膜到脑神经，人类视觉已经被深入地研究过了。但在这里，我们只抽出对人工视觉系统模拟有用的方法和属性。人有两只眼睛，总共有 200°的可视范围，其中有 120° 的视觉重

叠（双目并用）[Wandell95]。眼睛聚焦光线到眼睛后方，用杆体细胞和视锥细胞来检测光线和颜色。视觉敏锐度在固影（fixation）的中心处最强，并随着到中心距离的变大而减弱。视锥细胞检测颜色并将其收集后送到视网膜的中心处，在那里，对光有 100 倍敏感度的杆体细胞主要负责产生夜视和周边视觉，结果就是周边视觉对颜色有很少的反应，但是对运动极其敏感。

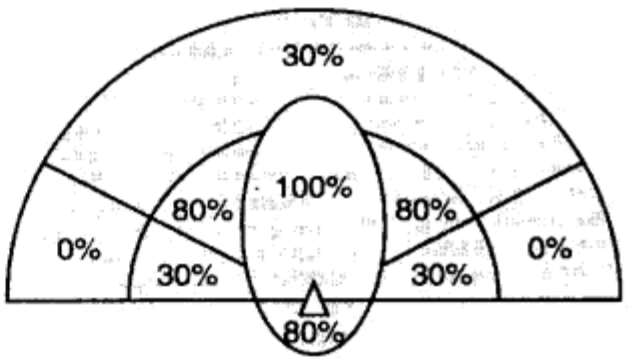
使用更多该模型包含的科学知识，你可以开始做一些观察。首先就是视觉敏锐度和颜色检测在视觉中心是最高的，并且在周边迅速下降。第二个结论就是当周边视觉弱的时候，运动检测变得更敏锐。

使用工具箱中的离散测试可以创建一个视觉模型，以给视觉范围中的物体记分。在图 3.2.4 中，百分比表示特定物体被辨识的确定性。视觉中心的物体被完全地辨识，在近周边、中间周边，远周边以及身后的（第六感）区域有更低的确定性。

这个模型的一个重要特性是移动的物体得到 50%的加分，作为对运动的特殊感知。对游戏来说，行走和跑动可能会触发 50%的加分，但潜行或爬行却不会。

如果伪装或隐藏（可能在阴影中）在游戏中起到了很重要的作用，就需要结合参考对比度和暴露出的轮廓来减弱辨识度。比如，当玩家在黑暗角落里蹲着，他们的轮廓就会更小而难以看清楚，结果就需要 100%地减弱他们的辨识度。就算是代理直接看着玩家，也许代理只会盯着看几秒然后继续前进，因为它不能 100%地辨识物体。如果使用这种细微差别，建议再添加一个更小的椭圆，在其中辨识度会无条件地为 100%，而无视轮廓和对比度。

确定性的百分比可以用游戏设计中任何合理的方式来解释。我们的方法是代理会在不同的阀值范围中表现出不同的行为。比如，在 100%的确定性下，物体被完全辨识，代理会射击物体。在 80%或更高的确定性下，代理会转头并开始处理物体辨识。在 50%或更高的确定性下，代理也许会转头。任何在 50%以下的确定性都不足以刺激代理采取任何行为。



规则 1：如果物体显著地移动，+50%

规则 2：如果暴露的轮廓减少并且/或者对比度很弱，则适当地减少确定性

图 3.2.4 视觉中的确定性被表示为离散测试的集合。在这个模型中，100%表示物体被完全辨识，80%及以上表示高度怀疑，50%及以下表示轻度怀疑。为了模拟对运动的特殊感知，运动的物体获得 50%的额外加分。伪装、隐藏或蹲下（减少的轮廓）的物体会减少 50%的确定性

图 3.2.4 模型的弊端是它仍然存在强制离散区域。用户可以改良这个模型，以使它支持随角度和距离的逐渐衰减。图 3.2.5 展示了一个视觉模型，内圈随着角度衰减，外圈使用角度衰减和距离衰减的组合。椭圆测试保持了离散，在前进方向上具有 100%的确定性，而身后只有 80%的确定性。

图 3.2.4 和图 3.2.5 的模型只是例子，在游戏设计中应该视需求来调整。它们只是基于特殊的人类视觉的近似。很明显，这些模型有很大的自由度，并且只是近似科学。比如，为了简化，这些模型使用了 180°的视觉而不是 200°。但是这些模型相对于经典的游戏视觉模型，在敏锐和敏感度上有了很大的进步。

这个视觉模型的另一个重要特性是考虑了代理的精神警觉度。当代理高度警觉时，确定性百分比会提高，而且视觉区域也会扩大，当代理分心或困乏时，确定性百分比会下降，而视觉区域也会减小。

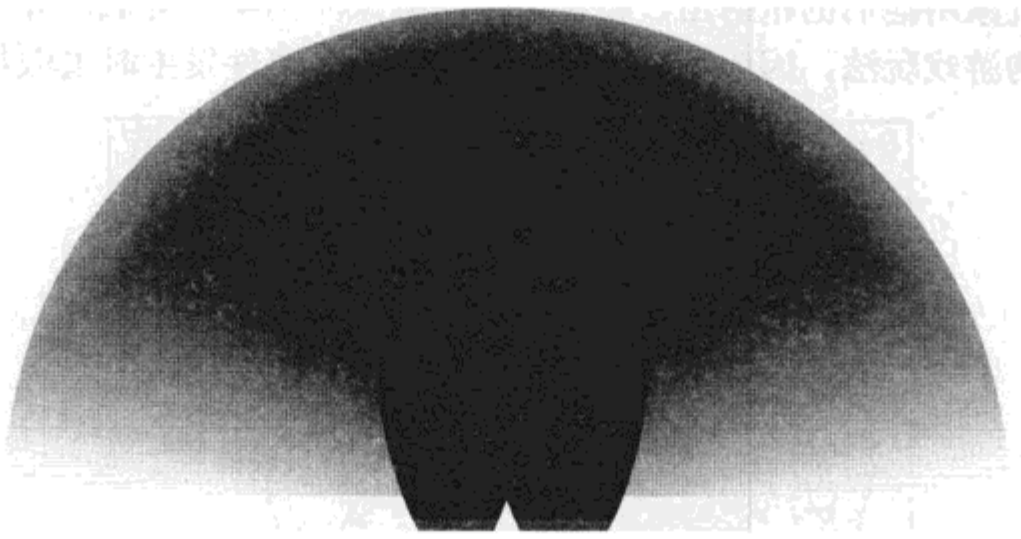


图 3.2.5 带有梯度区域确定性的视觉模型。内圈随着角度衰减，外圈随着角度和距离衰减。椭圆测试仍然是离散的。注意黑色是 100%的确定性，白色是 0%的确定性

3.2.5 用确定性模拟人类听觉

就像视觉模型中所展示的那样，用百分比来表示感知鉴别是一个很有效率的方法，可以将敏锐度引入视觉模型中。类似地，同样值得去建立一个听觉模型来计算确定性的百分比。在开始创建听觉模型之前，让我们看看和声音及听觉有关的一些问题。

声音最重要的属性是强度会随着距离指数级地衰减。虽然声音很容易通过空气传播，但是只有低音调的声音才可以穿墙。这让对话和很多高音调的声音很难从隔壁房间中听到。最后，声音可以从墙面反射出来并在房间内回响，让声音可以绕开大部分的障碍。

当建立听觉模型时，一个简单的半径检测可以测试代理是否听到特定的声音，这个检测可以使用很多种方法来补充。首先，声音的音量大小影响它传播的距离，从能够清楚地识别衰减到不确定。第二，墙可以导致一些程度上识别的不确定性，随墙的厚度等而定。第三，因为声音可以从表面弹回，如果音源和听者之间的视线被挡住了，可以尝试通过计算来看看是否能得到一条畅通的路线。如果可以，这条路径的距离可以被用来决定衰减程度。另外，还有一种较少计算量的替代方法，使用区域来判断声音是否可被听到，在特定区域内或临近区域内发出的声音总是可以被听到，不管音源和听者之间有没有墙。在一些情况下，分层寻路中使用的粗糙的搜索空间也可以被用来决定声音区域。

图 3.2.6 证明了声音伴随着区域衰减。基于声音的强度，声音在某个半径内具有 100%的辨识度，大于那个半径，确定性就会在一定距离后衰减为 0。但是，只有当听者位于与音源相同或者临近的区域才能听到声音。

如果区域方法需要太多对游戏世界的预计算或者说其不适合随机地图，那么我们可以开发寻路引擎来决定声音是否可以在视线被阻挡的情况下从音源传播到听者。虽然这个方法需要更大的计算量，但可以准确地知道声音是否可以无阻碍地传播到听者，以及得到声音传播的近似距离。

就像墙可以挡住视线一样，其他声音可以淹没特定的声音而让它很难被听到。为了模拟这种效果，你需要考虑所有的声音，包括环境音效，并且确定是否存在一个更大的声音会喧闹到遮住其他所有的声音。比如，当玩家在跑时，如果有火车经过，就可能听不到他们的脚

步声了。而如果玩家用他们的枪射击，枪声可能会被火车的噪声降低，并且更难被听到。这种模拟创造了新的游戏玩法，因为玩家被鼓励在其他噪声事件发生时去做带噪声的行为。

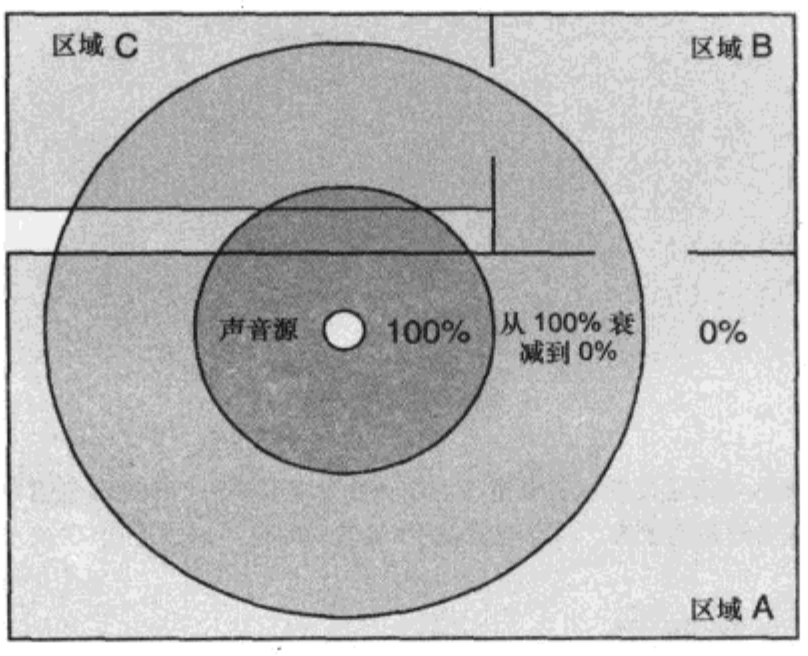


图 3.2.6 听觉模型证明了声音强度伴随着区域衰减。只有当音源在相同或相邻区域时，一个代理才可以听到声音。在这个例子中，即使半径检查允许，声音也不会传播到区域 C 中

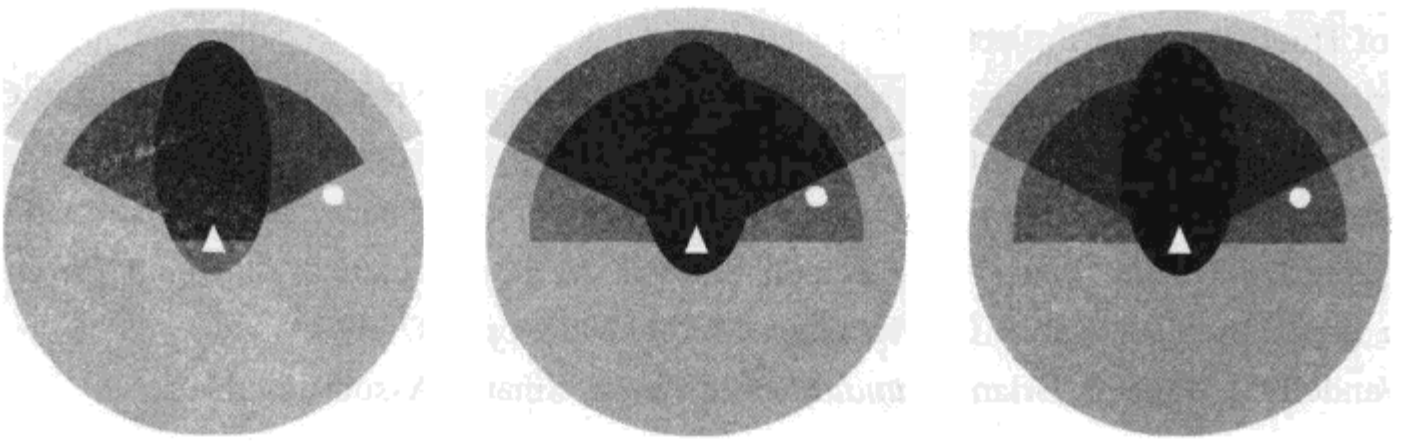
类似于在视觉模型中加入第六感，听觉模型同样可以包含其他感觉，如味觉。比如，一具腐烂的尸体发出味道，传播一段距离并且衰减，就像声音一样。另外，更强的味道会盖过弱一些的味道，所以也要考虑模拟这个特性。味觉可能不会在每个游戏中都十分有趣，但当视觉不足以辨别一个物体时，它可以用来增加额外的信息，并创造更多的游戏玩法。

3.2.6 统一的感知模型

我们已经创建了视觉、听觉、味觉以及第六感的感知模型，最后的任务是把它们合并到一个统一的感知模型里。这样做的动机是：所有的感觉应该一起将它们的环境告知代理，把它们的线索合并到一个完整的描述中来表示当前的情况，可以让代理最好地感知。

因为这个例子已经使用了百分比表示的确定性，自然的扩展就是把它们用某种方法合并起来。有 3 种选择，第 1 种是在视觉、听觉和味觉中挑选最大的确定性，如图 3.2.7（a）所示。比如，如果视觉区域有 30%的确定性，听觉有 50%的确定性，区域就会有 max(30%,50%)=50%的确定性。第 2 种选择是把所有感知的确定性相加，如图 3.2.7（b）所示。比如，如果视觉区域有 30%的确定性，听觉为 50%，区域就有 30%+50%=80%的确定性。第三种选择是取得视觉模型确定性，并且为听觉/味觉添加剩下空间的一半，如图 3.2.7（c）所示。比如，如果一个特定的视觉区域有 30%的确定性，在那个区域听到声音会添加(100−30)/2=35%，最后结果为 65%的确定性。最后的方法避免了任何区域有大于 100%的确定性。

为了理解统一感知模型的成果，考虑图 3.2.7 中的白圈为玩家，如果玩家十分安静并且没有行为，则代理完全察觉不到他，因为他只有 30%的确定性。如果玩家发出了很大的噪声，依照不同的方法他会被辨识为 50%、80%或 65%，这也许会让代理做出转头的反应。如果物体跑动并且做出很响的射击声，依照不同的方法，他会被辨识为 80%、100%和 90%。在这种情况下，代理也许会迅速地转动他们的头和身体，并且当他们面对玩家时就会射击。



(a)
取感知的最大确定性

(b)
把视觉和听觉的确定性相加

(c)
有比例地添加听觉确定性

图 3.2.7 3 个合并视觉、听觉或味觉的统一感知模型的例子。听觉有最大 50%的确定性，而且不带衰减。(a) 从每个区域中挑选最大确定性的感觉。(b) 把视觉和听觉加在一起。(c) 取得视觉确定性，然后为听觉添加剩下空间的一半（为了避免确定性大于 100%）。在这个例子里，白圈为发出了很大声音的玩家，依照不同的方法，它们会分别被辨识为 50%、80%和 65%

3.2.7 为统一感知模型添加记忆

为了达到更好的仿真，代理必须拥有短暂的记忆来改进他们的感知模型。为了让代理不要忘记他们最近辨识的物体，这是必要的。比如，如果玩家在代理的中间外围视觉范围内迅速移动，玩家会被 100%的辨识。如果玩家超出代理而移动到远端外围视觉范围内并停下，对于玩家 100%的辨识记忆需要被保留一段时间（就算玩家现在只能被 30%辨识）。这是有意义的，因为代理完全辨识过玩家，而且仍然有视觉接触，有理由去假设它是同一个物体并且是完全被辨识的。

为了实现这种记忆，每一个进入感知模型的物体都需要被记录。物体应该有一些特有的标记数字，可以与不同级别的辨识联系起来。一个时间戳和物体上次可知的位置也应该被记录，这些信息应该保存在代理中。一般的规则是只允许增长确定性的级别，因为所有线索都只会更增进对物体的了解。当物体在一段时间内没有被感知，这个结构就会被从记忆中清除。

3.2.8 结论

统一感知模型把视觉、听觉、味觉，甚至第六感结合到一起，给游戏代理连贯并详细的对游戏世界的认识。很多十分吸引人的特性被引入这个模型，比如运动检测、藏匿和警觉，这将会带来十分丰富有趣的游戏玩法。当玩家更好地理解了底层的感知模型时，他们就可以设计创新的方法来操纵和欺骗代理，这会大大提高游戏体验的质量。

就像之前所说的，统一的感知模型是为了给游戏代理带来更多的精细以及仿真。然而，它是一个十分灵活的模型。区域和百分比表示只是简单的建议，对于特定的游戏，它们不可避免地需要被改进。考虑你可以使用的每一个工具，可以创建你自己的感知模型来配合并增强特定的游戏设计。

3.2.9 参考文献

[Leonard03] Leonard, Tom. "GDC 2003: Building an AI Sensory System: Examining the

Design of Thief: The Dark Project."

[Orkin05] Orkin, Jeff. "Agent Architecture Consideration for Real-Time Planning in Games," AIIDE Proceedings, Artificial Intelligence for Interactive Digital Entertainment Conference, 2005.

[Rabin05] Rabin, Steve. *Introduction to Game Development,* Charles River Media, 2005.

[Tozour02] Tozour, Paul. "First-Person Shooter AI Architecture," *AI Game Programming Wisdom,* edited by Steve Rabin, Charles River Media, 2002, pp. 387–396.

[Wandell95] Wandell, Brian. *Foundations of Vision,* Sinauer Associates, 1995.

3.3　管理 AI 算法复杂度：泛型编程方法

Iskander Umarov
Anatoli Beliaev

在过去 7 年中，TruSoft 公司致力于行为捕捉人工智能技术的研究和开发。这些技术允许创建一种新类型的 AI 游戏代理，进而以人类的方式来学习和调整。AI 游戏代理学习人类玩家的游戏风格，并通过调整这些策略来达到他们的目的。

系统允许游戏设计者直接训练行为捕捉 AI 代理，只需要坐在游戏机或 PC 面前，扮演要训练的代理的角色进行游戏即可。代理可以直接从人类的控制中学习战术和策略，而不需要编程。最终用户可以用同样的系统来训练游戏角色，把传统的机器人开发提升到一个全新的高度。通过简单地进行游戏，行为捕捉系统允许用户创建风格截然不同的由 AI 控制的代理。

3.3.1　介绍

在进行我们的行为捕捉 AI 技术（Artificial Contender）的工作中，我们遇到了一个和其他许多 AI 系统一样有趣的挑战。有些时候 AI 决策相关算法的复杂度超出了控制。刚开始时它们是一段很简单的代码，结束时则变得非常混乱（充满了手动循环和分支）。我们需要一种不会引入重大抽象惩罚的管理复杂度的方法。

Artificial Contender 技术是基于实例的学习系统，它收集学习到的行为的实例，然后在决策过程中使用它们。学习时收集的数据可能不能直接用于现在的状况。学习到的实例需要重新评估。可能的行为会被过滤或修改、一般化或特殊化，优先级可能会被调整。一个行为应该是从一组行为中挑选出来的，如果这个行为还不够好，应该分析另一组行为。你也许需要对下一组使用不同的算法，或者不同的过滤标准。但是，如果下一组不存在更好的行为，你也许需要重新考虑上一组的行为，比对每组数据的一致性，等等。这样复杂的算法是十分典型的。

你准备如何解决这个问题呢？让我们先考虑最直接的实现。不要过度设计，不要使用不成熟的优化。当不得不处理一组行为时，只需要创建一个描述行为的对象容器。当不得不遍历行为时，就实现一个循环。当不得不遍历一个行为子集时，就实现一个嵌套循环。当不得不过滤行为时，就检查循环中的必要条件。当用这种方式实现时，我们会遇到下列问题。

- 算法中的每一个部分都增加了实现的复杂度。代码变得很难领会。看上去最简单和最直接的方法导致了非常复杂的代码。
- 以这种形式来维护代码变得十分困难。用户很难去理解和修改，调试的过程会很有挑战性。
- 优化这种算法也会变得越来越难。用户很难去找到性能的瓶颈，很难去修改代码并保证它还正确，很难去创建为不同环境和条件而优化的定制版代码。
- 代码被很紧密地耦合在一起，使它们很难被重用。
- 当做出改变时，引入 BUG 的风险很高，单元测试不能消除风险，因为确定整个算法的期望结果经常会变很难。

怎么管理这种复杂度呢？最明显的答案是使用分解。很多方法可用来分解。我们想要实现方法容易理解、改变和重用。我们想要能够迅速建立这些算法，并做出快速和安全的改变。换句话说，我们不能牺牲这种技术的性能特性。当你把系统分解为组件后，你必须处理组件间的通信问题（双向地发送数据、转换数据等）。Artificial Contender 处理很多数据，并在处理每个单独的数据项时引入一点额外的花费，就会导致处理时间和资源消耗增长让人难以接受。

3.3.2 行为选择工作流程

“管道过滤器”设计模式

我们已经发现工作流程是最合适的用来表达这一类算法的比喻，它是基于著名的“管道过滤器”设计模式[Buschmann96]。考虑一下为什么这种设计模式会很合适（注意，我们使用“单元”而不是原本管道过滤器设计模式中的“过滤器”，是为了避免引起混淆——过滤器在我们的工作流程中是一种特殊类型的单元）。

- 我们想要从分开的单元中构筑工作流程。
- 每一个单元都应该正好负责算法的一部分。
- 每一个单元都应该有定义明确的输入和输出。
- 工作流程的结构应该是齐次化的，所以我们可以以不同的方式连接单元，除非实现部分的类型不允许连接单元。
- 我们想要能够创建非线性的工作流程配置，包括分支、合并和循环。
- 我们想要能通过最少的工作来改变工作流程配置。

“管道过滤器”模式的好处众人皆知[Buschmann96]：作业线配置的灵活性、组件的重用性、可能的并行处理等。让我们来看这个模式如何能够在这个实例中帮上忙，先不管实现的问题，但是记住这里的优先级：管理复杂度而不牺牲性能。

下列益处可以帮助我们管理复杂度。

- 关系的分离。处理算法被分为一个单独变换的序列，每一个变换都是不同且独立的任务。
- 模块化。每一个处理的任务都使用分离的单元封装，可以单独编码。
- 减少耦合度。单元只会在明确定义的通道中通信。通常单元不会共享状态，并且不

知道周围单元的实现，只要周围的单元遵守通常的需求标准。

- 可测试性。每一个单元都可以单独地被测试。为一个分离的简单的任务指定要求的结果比整个算法要容易很多。如果改变输入和检测输出很容易，那么每一个单元可以被看做是一个黑盒。同样，测试任何单元组合的协作结果也是可能的。
- 配置的灵活性。从相同集合的单元中建立不同的配置是可能的。同样，将更简单的单元组合为一个整体来当作更复杂的单元也是可能的。
- 特殊化的灵活性。单元可以有可选择的替换实现，这些实现是为不同的环境而定制的。
- 重用性。低耦合度可以让你把单元当作可被简单重用的独立模块。避免额外的依赖让单元的适应性更强。

下列益处可以帮助改进性能。

- 并行性。进行递增处理的单元不需要等待周围单元完成他们的计算；他们可以继续并行地工作。这会让工作流程显著变快，因为它利用了多处理器架构。
- 特殊化的灵活性。你可以创建另外的单元实现，特别是为了提高性能的实现。
- 性能分析。把处理过程分成不同的组件，可以让分析代码和找到性能的瓶颈更加容易。
- 部分结果。这部分对于 Artificial Contender 决策算法来说十分重要，并且会在接下来的部分中讨论。

部分结果

Artificial Contender 决策算法在描述潜在行为的数据序列上操作。最底层的单元产生行为数据序列，通常是提炼在知识库中的数据，或者基于统计，或者启发规则。查询一些来源对于处理时间来说是很昂贵的。如果现在的游戏形势众所周知，那么只有最“廉价”的资源才是必要的。只有当对于现在游戏形势没有准确匹配的情况下，才会有必要去查询更精密和昂贵的资源。

多数情况下是不需要完整的行为数据序列的。部分计算的序列也许足够做出最终决策，这样就可以避免昂贵的计算。如果一个行为足够好，就可以接受它进而停止计算了。如果事先计算好所有数据，就很有可能不得不扔掉大部分计算结果，而且这是你所不能承担的。然而，这样来实现工作流程是可能的：用高级别的单元来控制执行流程。由高级别的单元来决定怎么、什么时候、多长时间他们想要继续从低级别单元中拿到结果。他们可以在任何时间打断查询低级别单元，或者根本就不会开始查询一些低级别的单元。如果可能，低级别的单元在被问询前不应该被预计算。

另一个需要考虑的问题是：在实时环境下工作，有时做出一个不完美但可以接受的决策也许要比花费更多的时间来计算一个完美的结果更好一些。有可能这样来安排工作流程中的单元：首先计算并延缓“可接受但是不完美的”行为，然后在还来得及执行行为时继续计算，并在没有找到更好的行为的情况下接受其中一个延缓的行为。

“管道过滤器”的缺点

“管道过滤器”模式也不是完美无瑕的，让我们看一下它的缺点，并看看能做些什么把

它们降到最低。

- 共享状态信息。如果单元需要共享状态信息，它可能是不高效和不灵活的。看上去对于这个应用，这不是什么严重的问题，因为大多数单元不需要直接共享任何状态信息。在一些例外的情况下，你可以放开管道过滤器的限制，允许单元以特别高效的方式来进行通信，避开正式的单元输入和输出。但是那些例外应该十分罕见。
- 通信和数据传输的开销。单元必须交换数据，这可能会引起一些开销。可能需要一些不直接对实现算法做出贡献的数据处理。如果单元是可以互换的，它们必须商定共有的通信协议，有些时候只是一个字符流大小。序列化和解析的过程可能会因太过昂贵而导致这个模式不可用。我们准备解决这个问题，并且最小化或者彻底去掉这个开销。
- 并行处理让人失望。因为不同的原因，并行处理的性能可能让人失望[Buschmann96]。

① 可能是因为通信开销，但是我们已经承诺会最小化它们。

② 也可能是因为架构依赖的原因，比如，上下文切换和同步开销。我们现在先不考虑这些问题，因为它们看上去在所有的并行处理方案中都很常见。

③ 也可能是因为处理的性质或者不良代码，比如，单元在给出任何输出数据前就消耗掉了所有的输入数据。这个单元可能变成整个工作流程的瓶颈。为了避免这个问题，无论何时只要有可能，你都应该让处理变为递增的。

- 复杂的流程控制逻辑。工作流程的处理性质大多数是按顺序的，所以实现分支、循环和其他复杂的结构就变得很难。但是，在这个应用中，我们几乎可以从次序化处理方面重新思考和定义算法。这些新的定义会对自身十分有益。当一个看上去需要复杂控制执行流程的复杂任务被改变为更简单步骤的序列时，无疑会提高实现的内部质量。在少数情况下，当你不能做到时，可以使用传统管道过滤器模式之外的特殊结构。
- 错误处理。总的来说，在管道过滤器模式中，错误处理可能会十分复杂，但是 Artificial Contender 决策算法不需要任何错误后的恢复。所以，整个错误处理问题对于这个应用来说不是十分重要。
- 复杂度、增加的可维护性尝试。这个模式采用了它自己的复杂度和可维护性尝试。因为将一个单独的实现分解为多个组件增加了组件和依赖性的数量。这个问题并不是只针对于这个模式的，而是任何分解的结果。我们的解决方案是尽可能地保持简单和轻量级。

让我们更仔细地看一下 Artificial Contender 决策工作流程和它们特殊的需求。

工作流程图表

任何工作流程都有一个视觉表现，比如一个单元图表。我们开发了一个特殊的图形语言，可以让你用十分紧凑和生动的方式表达不同的工作流程配置。使用不同的形状和连接线，你可以阐述处理步骤的次序、数据流和互相依赖关系。

图 3.3.1 展示了一个平均复杂度的工作流程的例子。这个工作流程的英文描述会是笨重且让人迷惑的。但是，对于熟悉这些图表的开发者来说，能十分容易地理解发生了什么。

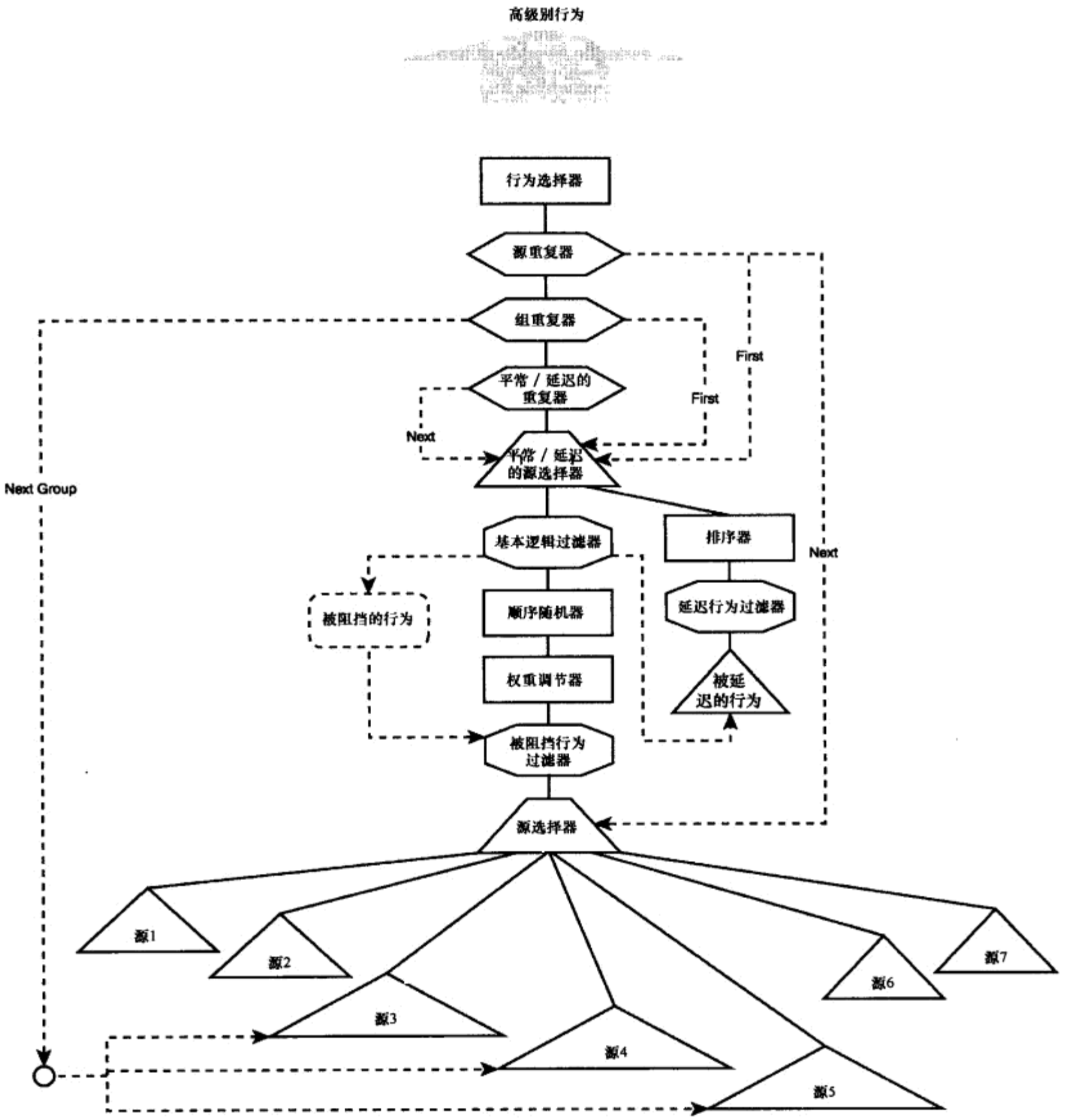


图 3.3.1　Artificial Contender 决策工作流程示例

这些图是紧凑和可读的。它们同样可以让修改工作流程变得十分容易。你可以交换单元，重新安排和重新连接，添加和删除。比如，你也许想要在过滤前或过滤后修改行为。看一下图表，你就会知道现在的工作流程做了什么。如果想要改变次序，只需要重新连接单元。把这个和第一个算法的简单实现进行对比。改变行为过滤和修改需要多长时间？怎么能够保证改变一定正确？工作流程图让这些问题的答案显而易见。

执行流

一个工作流程图只展示了一张工作流程的静态图，它用陈述的方式表现工作流程配置，但是它不能阐述实际的执行次序。然而，当图表的读者知道基本规则时它是足够的。省略数据和执行流的细节可以让图表紧凑并让人印象深刻。但是这里发生了什么呢？

单元和一些表达单独或一套行为知识的不同对象序列一起工作，这些对象被称为行为信息。单元消耗、处理并且发出行为信息对象的序列。行为信息对象从低级别单元移动到高级别单元。在移动的过程中，它们可以变换为其他的行为信息对象，可以被过滤掉，也可以被分离成几组不同的行为信息对象，还可以与其他对象一起被合并等。

单元是如何并以什么顺序来处理行为信息对象的呢？为了回答这个问题，考虑图 3.3.2 中的一个十分简单的工作流程。

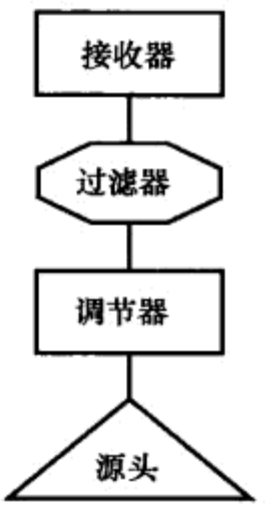


图 3.3.2 非常简单的工作流程

工作流程应该是这样工作的：

- 源头生成行为信息对象（比如，基于 Artificial Contender 知识库）；
- 调节器改变行为信息对象，把它们调整到现在的游戏形势；
- 过滤器检测行为信息对象是否足够好，来选择是否过滤它们；
- 接收器接收过滤器发出的第一个行为信息。

你可以让它就在那个序列中工作，实现“推动”模型（从源中开始处理，把源中产生的数据传到调节器等）。这就是多个“管道过滤器”实现如何工作的。然而，这不是你在 Artificial Contender 系统中所需要的。对于性能来说，查询一些知识源可能过于昂贵。

首先，你可能没有必要去查询它们全部。其次，你可能不需要从它们每个中获得整个行为信息序列。这个例子中的工作流程可能会接收源产生的第一个行为信息，其前提是它可以从过滤器中通过；那种情况下没有必要去生成更多的行为信息。但是源应该不会察觉到这个事实，所以你不能让源来决定什么时候生成整个行为信息序列。更高级的单元必须可以决定查询哪个低级别的单元以及决定什么时候停止。虽然理论上可能使用“推动”模型来实现，但是“拉动”模型看上去更加自然。

- 接收器从过滤器中要求一个行为信息。
- 过滤器从调节器中要求行为信息并一个一个地检测它们，寻找应该被返回到接收器的行为信息。
- 调节器一个一个地查询源对行为信息的检索，修改它们，并且同样一个一个地返回到过滤器。
- 源响应调节器的要求，一个一个地发出行为信息对象。

- 当最顶级的单元（接收器）停止要求更多的行为信息时，就让工作流程停止工作。

图 3.3.3 展示了“拉动”工作流程的序列图。

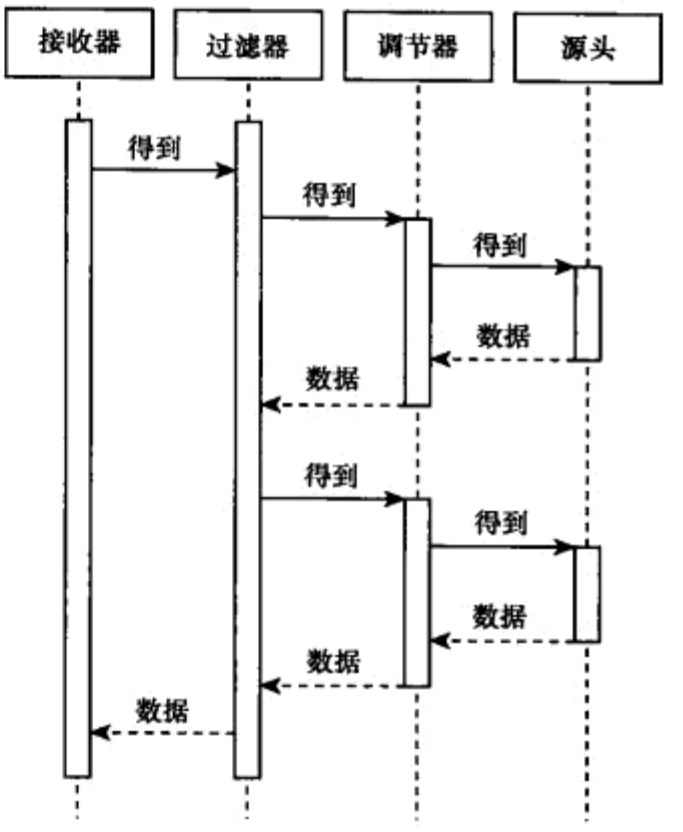


图 3.3.3　“拉动”工作流程

典型单元

为了 Artificial Contender 决策算法，你通常需要一些单元的分类。

- 源会基于决策工作流程相关的外部数据生成行为信息：从 Artificial Contender 知识库中、从启发算法、从统计表等。
- 过滤器决定消耗的行为信息是否满足了明确的条件，并且相应地输出或者忽略这些行为信息。通常，过滤器会确定潜在的行为对于现在的游戏状况是合适的。
- 调节器改变消耗的行为信息对象并且输出改变过的对象。比如，它们可以依照现在的游戏状况来调整从知识库中检索到的行为，或者可以调整行为的优先级。
- 排序器消耗行为信息对象的序列，并且以新的顺序输出相同的对象。比如，它们可以以优先级、估计效果、种类等方式对行为排序。排序标准可以是十分灵活的，并且不用指定一个确定的顺序。它们可以执行加权随机重新排序，从而引入更多的多样性到 Artificial Contender 代理的行为中。
- 分裂器把正到达的行为信息对象流程分裂到多个流程中，并且重新定向这些流程到多个输出中。分裂器用来为特殊处理过程分离一些行为信息对象。
- 合并器有多个输入，并且把所有输入中的行为信息对象流程重新定向到单独的输出中。它们通常用来顺序地查询一系列的行为信息源，并把结果放入一个集合中。
- 选择器有多个输入并将行为信息对象流程从一个输入重新定向到单个输出中。通常，选择器有一个特殊的控制通道，可以用来切换活跃输出。它们同样经常被用于顺序地查询一系列行为信息源。但是，与合并器不同，它们允许把每个源中的行为信息当作不同的组。
- 重复器消耗并输出所有可用的行为信息对象，然后执行一个特定的行为，然后再次消耗并输出所有可用的行为等。与选择器合作，重复器可以实现对来自不同源的行为进行查询和处理的循环。

Artificial Contender SDK 中包含了常用单元的泛型实现。但是，这不是单元类型的全部列表。开发更加定制化和特定的单元是可能的。把这些单元合并进不同的配置，可以创建高通用性和高灵活性的工作流程。

约束

单元可以用很多种方式来连接，然而，系统自由是有限制的。因为单元的本质差异性太大，所以某些合并是说不通的。比如，如果想要一个单元执行行为的加权随机选择，必须保

证输入的行为信息对象有关联的权重。想要可以表达这些约束并把它们和单元联系起来，然后可以可视化这些约束并且自动检测它们，以保证工作流程的正确性。

3.3.3 实现

泛型编程和 C++

我们选择 C++作为 Artificial Contender 的主要实现语言。这种语言提供工具来处理抽象，并且仍然允许对低级别实现细节的控制，来达到更高的性能。使用 C++，我们可以利用“管道过滤器”模式的益处，并同时克服潜在的性能问题。

泛型编程帮助我们同时达到这些目标。在不用牺牲效率的前提下，用十分泛型和抽象的形式来实现代码是泛型编程的一个关键思想[JLMS98]。

为了让工作流程足够灵活，在设计工作流程单元时，我们应用泛型编程规则。单元实现应该对于周围环境做出最小的假设。越小依赖于实现细节，比如明确的数据类型，实现就越有适应性。责任定义明确且互不相关的单元应该只需要了解那些与责任直接有关的细节，并且是以相当泛型的方式。当设计一个单元时，主要的规则是不用过度详述。如果单元的本质功能不依赖于一个特定的数据类型，就不要在实现中提到这种数据类型。如果它依赖于一种数据类型，但是仍然可以与不同的类型工作，那就把这种数据类型作为一个参数。具体细节的缺失让实现看起来有一点模糊，但是事实上正好相反，实现变得简洁和精确了。

多态工作流程单元

最常见的工作流程单元需求是它们应该处理和输出数据。如果可能，单元也可以消耗其他单元生成的数据。同时，必须可以通过不同的方式来连接单元。

依赖倒置规则[Martin02]规定单元不应该直接互相依赖。相反，单元之间应该依赖于输入和输出的抽象需求。在这种情况下，只要所有的单元都满足需求，那么改变一个单元不会要求改变其他的单元。需求有多么抽象？更多的抽象会给予更多的灵活性，但是会更难保证单元的开发者有足够的信息来真正地满足在具体实现中的需求。考虑到需求和工具的限制，你必须平衡这些问题。

虽然单元的本质可能绝对地不同，但是它们仍然有一个共有的性质：输入和输出数据项目。如果你在所有单元中相似地实现这个性质，那么它让你可以从相同单元中建立不同的配置。你可以继续连接和重新连接单元，而不用考虑不同的输入/输出接口。

因为我们在实现“推动”模型，单元应该只提供获得数据的普遍方法。如果单元知道其他单元如何输出数据，它们会自动输入数据。我们会使用一些形式的多态来让这个过程看起来统一。

怎么才能让单元具有多态性？有面向对象背景的开发者也许已经有了一个答案：基于虚函数或其他形式的动态绑定来统一接口。

需求输入的单元可以保持对一个实现了基本接口的对象的引用。单元不需要知道其他单元的明确类型，它们可以只依赖接口，看下面的代码。

```
class BaseBlock {
```

```
public:
    virtual OutputData getData() = 0;
};
class Modifier : public BaseBlock {
public:
    Modifier(BaseBlock& input) : input_(input) { }
    virtual OutputData getData() { return modify(input_.getData()); }
private:
    BaseBlock& input_;
};
```

看上去足够灵活吧？但它不会应用在这个实现中，因为这个系统的性能不够高。

基于虚函数的多态会引入显著的性能问题。如果你遵照单一职责原则[Martin02]，并且让单元具有细粒度，你最后会得到很多只有简单或无意义实现的函数。

有时候你可以忽略间接函数调用的开销，但是你肯定不能忽视间接调用造成了编译器优化瓶颈这样一个事实。通常，编译器可以优化一系列静态函数调用。如果函数定义可见，可以写为内联函数，消除传递的参数和返回值的开销，并且产生十分紧凑和有效率的执行代码。但是，如果编译器不知道将会调用哪个函数实现，那么内联函数就不再是一个选择，也就无法避免所有的这些开销了。

在前一个代码示例中，你会怎么定义 OutputData 类型？在这里，你有一个相似的挑战。因为单元应该处理其他单元要求的数据对象，输出数据对象就应该是固定的或者多态的。然而，在这种情况下，问题看上去甚至更严重。每一个单元对输入数据对象都期望不同的属性，几乎不会有普遍的需求。大多数的需求是特定于单元的，并在其他单元的上下文中没有任何意义。如果你固定数据类型，类型就必须实现所有的可想象的函数和数据成员占位符。只有其中一些会被真正地使用，而且（甚至更恐怖的是）还有一些是在被正确初始化前禁止使用的。相反，你可以试着去提取接口，覆盖到任何可能，然后建立继承等级并重载虚函数，提供对特定子类型来说是不“合法”的方法的空白（stub）实现。但是，它让我们很容易就会违背 Liskov 替换规则，并且我们从拒绝继承的代码风格中得不到好处。而且，就算你准备这样实现，你也会遇到以前提过的性能问题。

为什么不使用其他形式的多态呢？你可以使用无界的动态多态，类似于动态类型化语言，比如 Smalltalk 和 Ruby。这些方法对 C++开发者来说需要额外的工作，但肯定是可能的。你甚至可以引入反射和运行时元编程能力，动态地分析单元需求，并在运行时构建合适的对象。

主要的障碍是一样的：性能。虽然它们不可思议地灵活，但是所有类型的动态多态，无论是有界还是无界的[Czanecki00]，都不能解决这个问题，主要是因为性能开销。单元粒度越细，性能问题就越明显。

同样，它可能会让工作流程过于灵活。你将不能够再依赖静态类型检查，而且必须执行工作流程来检测明显的约束干扰。这会让设计工作流程变得过于复杂，而阻挠了方案的目的。反射和运行时元编程甚至会让性能更加糟糕。

所有类型的动态多态提供了在运行时替换不同类型对象的能力，但是会让你为这种能力付出性能代价。你不想为不会使用的灵活性付出代价，并且通常不需要在运行时改变工作流

程配置。在设计工作流程的时候，你需要足够多的灵活性，但当代码被编译后，你就不需要改变它了。

静态多态可以帮助你把尽可能多的工作从运行时转移到编译时，这就是我们选择基于C++模板的泛型编程方法的原因。在这个方案中，你仍然可以从细粒度的单元中建造工作流程，但不需要为它们付出处理时间或者内存。同样，编译时类型的安全性是完整无缺的。让我们用这个概念来再次实现调节器单元。

```
template <typename Input>
class Modifier {
public:
   Modifier(Input& input) : input_(input) { }
   OutputData getData() { return modify(input_.getData()); }
private:
   Input& input_;
};
```

没有继承，没有虚函数，但仍然是多态的。“实现 BaseBlock 接口”的需求被模糊代替，却是更加灵活的：“实现 getData 函数，返回一个表现得像 OutputData 的对象。”

实现静态多态，你仍然可以在更合适的情况下使用动态多态（比如，有必要在运行时改变工作流程配置的时候）。你不需要改变单元实现，只需要一个简单的适配器，而基于虚函数或者信息调度，这个适配器可以将静态的多态接口转换到相似的动态多态接口。看下一个例子。

```
template <typename AdaptedBlock>
class Adapter : public BaseBlock {
public:
   Adapter(AdaptedBlock& adapted) : adapted_(adapted) { }
   virtual OutputData getData() { return adapted_.getData(); }
private:
   AdaptedBlock& adapted_;
};
Modifier modifier;
Adapter<Modifier> adaptedModifier(modifier);
```

同样，可以实现相似的适配器来支持不受限制的动态多态。显然地，当使用这些适配器时，性能问题又回来了，但现在你有一个选择，你可以只在必要时才使用它。

行为信息流程

输出数据看起来会像什么？你需要从行为信息对象中获得一定程度的多态行为。但是，性能考虑会指引你避免有关的开销。行为信息实现的细节会在后面讨论。现在假设你已经定义了足够泛型的行为信息类型来满足工作流程中每一个单元的需求。

大多数时间你会与行为信息对象序列一起工作。你如何存储这些序列？你如何在单元之间传递它们？你会在查询单元前提前分配内存么？被调用的单元应该自己分配内存么？获得整个序列可能昂贵，如果你可以选择第一个行为就结束，你会那么做么？

幸运的是，大多数时间你不是真的需要整个序列，至少不是同时。在大多数情况下，

你可以一个一个地处理行为信息对象，分开考虑这些对象并做出合适的决策。你可以传递函数而不是请求数据，以此来取代决定怎么去存储和传递你可能不需要的数据。我们应用“说，不问”的方法。你告诉单元对数据去做什么和什么时候停止，而不是要求所有的数据。

可以用如下规则替换单元接口需求。

- 单元应该实现拥有一个固定名字的函数（称为 forEach）。
- forEach 函数应该接受一个函数作为参数。
- forEach 函数应该应用接收到的函数到每个被单元发出的行为信息实例。

这暗示了传递到 forEach 函数的函数应该可以接受行为信息对象作为参数。注意，我们不在需求中指定任何类型，所以单元可以用不同的方式自由实现 forEach 函数和回调函数。

单元实现实例

下面的代码段展示了源单元（不需要输入的单元）看上去会是什么样的。

```
class Source {
public:
   template <typename F>
   void forEach(F f) const {
      ...
      f( generateNext() );
      ...
   }
};
```

你想要 forEach 函数可以接受任何可调用的实体，包括函数和函数对象（仿函数）；这就是将 F 作为一个模板参数的原因。

注意，forEach 函数的调用者不用担心新行为信息对象的内存分配问题。源单元管理这个存储，并且可以在接下来的每个行为信息对象中重用它。通常，单元不用保存上一个行为信息对象，除非这是单元性质的需求（比如，对单元排序通常必须在发出第一个输出行为信息之前收集所有的输入行为信息）。这会帮助我们把数据传输的开销降到最低。

如果所有的单元满足这些需求，那么需要输入的单元可以期望其他单元实现一个相似的 forEach 函数。下面的代码展示一个典型的调节器的 forEach 函数实现。

```
template <typename F>
void forEach(F f) const {
   input_.forEach(ApplyToModified<F>(f));
}
```

Input_.forEach 调用保证行为信息对象是从输入单元中检索的。ApplyToModified 仿函数的目的是修改从 Input_中收到的每个行为信息对象，并且应用原本的 F 函数到修改后的行为信息中，如下：

```
template <typename F>
class ApplyToModified {
```

```
public:
    ApplyToModified(F f) : f_(f) { }
    void operator()(const ActionInfo& ai) const { f_(modify(ai)); }
private:
    F f_;
};
```

下面的代码展示一个典型的过滤器实现。

```
template <typename F>
bool forEach(F f) const {
    return _input.each(ApplyIfAcceptable<F>(f));
}
template <typename F>
class ApplyIfAcceptable {
public:
    explicit ApplyIfAcceptable (F f) : f_(f) { }

    void operator()(const ActionInfo& ai) const {
        if (isAcceptable(ai)) f_(ai);
    }
private:
    F f_;
};
```

注意，这里没有数据的实体复制。输入单元生成的行为信息对象会被适当地检查，原本的函数会被应用。当然，你对原本的函数一无所知，并且它也许会因为自身原因去复制或者转换数据。但由于过滤器单元的目的不会有复制或者转换，也就是说性能开销被彻底地消除了。

部分结果

当一个可接受的行为信息被找到，或者没有时间去完成整个决策过程时，最重要的需求之一就是可以打断工作流程。这很容易实现：只是让传递到 forEach 函数的函数返回一个布尔值，表明是否允许单元继续发出行为信息对象。每一次函数被调用，forEach 实现都应该检查结果并在必要时尽快退出。这会很有效率地停止整个工作流程。

图 3.3.4 中的序列图阐明了执行流程。

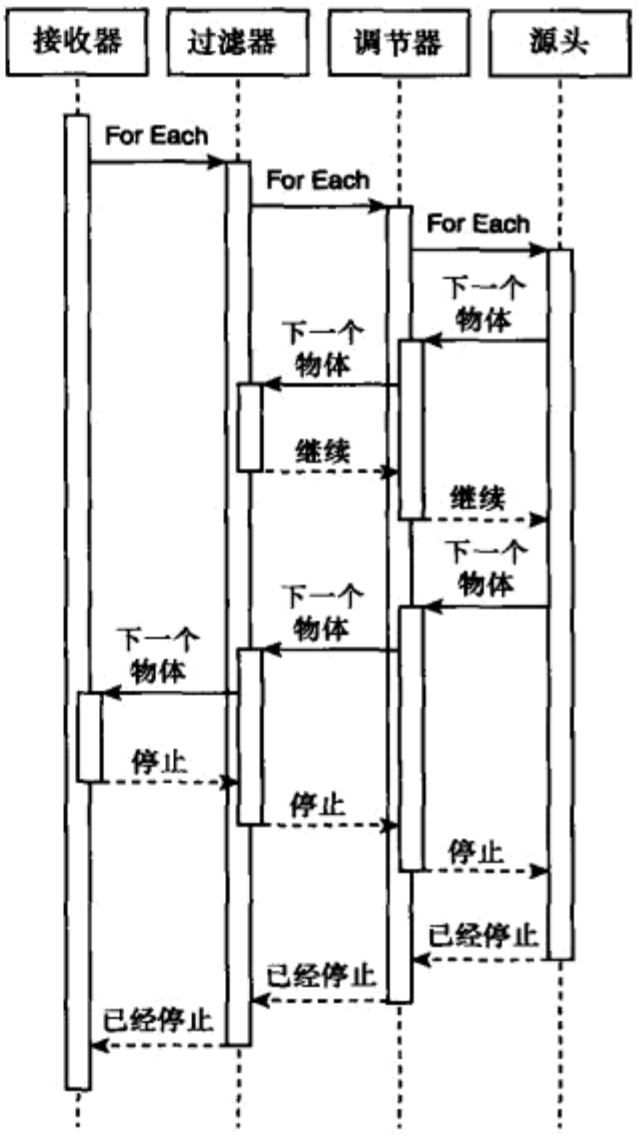


图 3.3.4 带有回调函数的“拉动”工作流程

函数指针与仿函数

你可以传递函数指针到 forEach 函数中。然而，

不同于通过函数指针的调用，仿函数（函数对象）的调用可以被内联（inlined），有效率地消除了大多数或者全部函数调用的开销[Meyers01]。此外，空的或者琐碎的执行过程可以被一起优化。这是用来获得抽象的好处而不是抽象的惩罚的一个很重要的方法。想象一下前一个工作流程的实现，所有被调用的代码都“适当地”被实现，甚至对单元来说，它们在工作流程中彼此都离得很远，以至于没有更多的必要去真正地调用函数和传递参数。当恰当地使用内联函数时，可执行结果中的整个算法可以被合并进一个只执行必要数据处理的高优化的代码块，而不用转移和转换数据。与此同时，开发者仍然处理这个算法的十分高级、抽象和分解的表示。

不过这个方法有个附加说明，基于你的编译器不同，内联的结果也许会不同。

- 首先，编译器也许不会完全使用内联的机会，而仍然使用真正的函数调用。为了达到期望的结果，你可能需要实验、调整和改变你的代码以及编译器设置。
- 其次，当大型函数被复制时，失控的内联可能会导致代码膨胀。在这种情况下，你可以使用函数指针。

行为信息类型

我们一直提到行为信息类型，但我们仍然没有展示它的定义，原因在于共用的行为信息类型是不存在的。每一个单元都对进来的行为信息流程有它自己的需求，每一个单元发出的行为信息对象都拥有特有属性。把所有的属性合并到一个行为信息类型中是低效且不安全的。你需要能够在最低级别的单元（源）中定义极简的行为信息类型，并且可以在编译时添加或者去除属性来提升工作流程。怎么才能做到这些？

- 只有源才会定义具体的行为信息类型。这些类型可以不同。每一种源只包含与它相关的成员。
- 所有其他的单元让行为信息成为一个模板参数。然后，通过使用继承或者聚合来修改行为信息属性，这些单元从它们的输入行为信息类型得出输出行为信息类型，结果是每一个单元的输出行为信息类型依赖于工作流程配置。
- 所有的单元使用直接关联单元职责的行为信息属性，而不使用任何其他的属性。这让单元具有很高的可适配性：它们接受不同的输入行为信息类型，但是任何未知的属性被传播到输出，并且可以被更高级别的单元所使用。

其他的单元实现

另一个泛型编程的关键思想是你可以为相同的泛型算法提供另外的实现，即为了尽可能地高效，在特定的情况下使用特殊化。这个方法对于我们的应用非常有帮助。比如，它允许我们拥有相同单元的不同版本，它们针对不同的平台做了优化，这些版本可以在编译时被手动或自动选择。同样，基于策略的设计[Alexandrescu01]帮助部分化地定制泛型单元实现，而不用重新实现整个单元并且复制代码。

构建工作流程

怎么让开发后的单元互相连接并让工作流程运行呢？在C++中，创建和连接单元看上去就像下面的代码一样简单。

```
template <typename Input>
Filter<Input> makeFilter(const Input& input) {
    return Filter<Input>(input);
}
...
makeFilter( makeModifier( makeSource() ) ).forEach(AcceptFirst());
```

但是，你不用总是手动地来做这些，因为所有单元遵循一样的规则，代码有十分规则的结构，这样就使得从脚本甚至是可视化的表示中自动生成代码成为可能，并且这个过程也是相对容易的。

限制

感谢静态多态，你仍然可以利用 C++的编译时类型检查。比如，如果过滤器尝试使用从调节器中接受到的行为信息对象的成员，而这些成员不存在或者有不同的类型，这段代码就不能被编译。在这种情况下，过滤器不能从调节器的输出中直接消耗数据。

但是有些时候还不够，编译错误信息可能是难以理解或使人误解的，特别是过度使用模板的代码，这会让我们理解哪个需求坏掉变得困难。为了简化诊断，我们使用 C++概念检查[Stroustrup03]。同样，我们也可以使用“红代码、绿代码”方法[Meyers07]。

这些限制同样可以在工作流程表中显现出来。

- 一系列的标记被绑定到单元输入。每一个标记表示进来的行为信息流程的一个需求。
- 另一系列的标记被绑定到单元输出。每一个标记表示被单元添加或者被单元删除的一个属性。
- 当单元要被连接时，首先应该分析标记。高级别单元的需求应该被低级别单元的输出所满足。如果它们之间不矛盾，连接就被建立了。然后，低级别单元输出的标记可以被传播到高级别单元的输出。

这个可视化表示可以让建立工作流程的过程十分直观和直接。

3.3.4 结论

虽然泛型编程不是一个新的想法，它仍然没那么容易去掌握并恰当地应用。此外，准备好与 C++编译器和其他的开发工具做斗争。即使当它们遵守现代的 C++标准，C++模板被支持的效率仍然留下很多要去改进的地方。幸运的是，编译器和工具正在改善，而且即将到来的新的 C++标准会使泛型编程和模板元编程更容易。

这个结果值得你去这样做，特别是当你需要编写出抽象和可重用的代码时，但是有着十分严格的性能需求。这篇文章描述的方法让 Artificial Contender 可以同时具有十分灵活、紧凑和快速的性质。

3.3.5 参考文献

[Alexandrescu01] Alexandrescu, A. *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley Professional, 2001.

[Buschmann96] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. Pattern-Oriented Software Architecture Volume 1: A System of Patterns, Wiley and Sons Ltd., 1996.

[Czarnecki00] Czarnecki, K., and Eisenecker, U.W. Generative Programming: Methods, Tools, and Applications, Addison-Wesley Professional, 2000.

[JLMS98] Jazayeri, M., Loos, R., Musser, D., and Stepanov, A. Report of the Dagstuhl Seminar 98171 "Generic Programming," Schloss Dagstuhl, April 27–30, 1998.

[Meyers01] Meyers, S. Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library, Addison-Wesley Professional, 2001.

[Martin02] Martin, R.C. Agile Software Development. Principles, Patterns, and Practices, Prentice Hall, 2002.

[Meyers07] Meyers, S. "Red Code, Green Code: Generalizing Const."

[Stroustrup03] Stroustrup, B. "Concept Checking—A More Abstract Complement to Type Checking," Technical Report N1510, ISO/IEC SC22/JTC1/WG21, September 2003.

3.4 有关态度的一切：为意见、声望和 NPC 个性构建单元

Michael F.Lynch, Ph.D.,
Rensselaer Polytechnic Institute, Troy, NY.
lynchm2@rpi.edu

有关态度的概念，也就是对一些态度对象的积极或者消极的评价，在心理学上有着很悠久的历史。一些游戏在意见和声望系统中使用这个概念，但是态度的概念更为全面。态度系统可以被用来以意见和声望系统之外的方式丰富 NPC 的行为，比如，作为决策树或者行为树的输入，作为社会网络建模的部分，丰富 NPC 的个性。

态度系统对于那些 NPC 需要对其他 NPC 或玩家展示可信的社会行为的游戏来说是很适合的。包括的游戏类型有上帝模拟游戏、RPG、约会游戏、政治派系游戏或者宫廷阴谋、间谍行为和一些类型的 RTS。对于一些特定形式的严肃游戏来说，态度系统也是有吸引力的。比如，那些需要模拟政治过程、媒体或者宣传效果，或者市场战役等的游戏。

本精粹展示了基本的态度理论，并建议一些可以被用来连接其他部分 AI 的轻量级实现。

3.4.1 简介

随着游戏主机和个人计算机变得更加强力，开发具有表现像真正人类（或者可能是异形）的游戏的前景变得更吸引人，并且更加被玩家所需求。感情和态度是让我们成为人类的一大部分因素，如果要为我们的游戏角色开发类似人类的行为，开发者需要解决人类行为这些复杂的概念。

本精粹展示了叫做态度的心理学结构，这个结构可以成为你工具的一部分来构造更有趣和更像人的 NPC。在表达这些想法时，注意我会对这些概念在社会科学中的处理方式进行相当大的修改，具体就是将一些更微细的方面过度简化为我认为会对游戏开发有用的几点。总之，为了让这些概念变得实用，我们必须用代码来实现它们。

幸运的是，游戏角色是真正人类的夸张模仿：更简单、更极端，甚至比真人更过火。这应该会简化困难。

所以你将要做的不是完全正确的科学。可以把它叫做认知工程学或甚至心理学破解。一个描述它的好短语是“关键技术实践”，在 1997 年被 Agre 提出并被 Michael Mateas 引用[Mateas02]。

我自己对关键技术实践的描述是：它是一种基于科学理论的工程实践风格，然而按照它自己的轨道发展并且获得自己的成功。不过如果有新的发现，它会重新检查它的前提和技术。游戏 AI 无疑是符合要求的。

3.4.2 态度

什么是态度？态度就像社会心理学中定义的那样，有一个通俗的意思，就像“对什么东西有积极的态度，”但不是“有一个坏的态度”或“抓住一种态度”的感觉。在社会科学中对态度的学习有着很悠久的历史，并且在社会科学中有关于它的大量跨越很多学科的文献。

学术研究者们在他们的定义中力求简洁，但对于态度来说没什么不同。几年来有很多不同的定义，但是 Alice Eagly 和 Shelly Chaiken[Eagly93]的定义更好些，态度是“一种通过评价特定实体的喜好或厌恶程度来表达的心理学倾向”。这个定义更好、更全面地覆盖了这个主题，感兴趣的读者可以去看看他们权威的文字[Eagly93]。

对于态度结构最重要的是评价度量。还原到最基本的核心，它意味着掌握着态度的一个人做出他喜欢或不喜欢态度对象的程度判断，也就是说，这个人判断态度目标有多么地吸引人或不吸引人。

态度可以被应用到任何可以评价的事物上，其目标通常被称为态度对象。态度对象可以是具体的，比如人、物理对象、地点；或者是抽象的，比如自由、平等、国家主义、公正的概念。对表示道德度量（像自由或平等）的抽象实体的态度通常被称为价值。

态度对象可以是单独的项目或者一整个种类或类别的相关项目。人类使用倾向来正确或者错误地对类型对象产生态度，进而减少认知负担和做出流程化决策。不好的一面就是可能会产生不公平的偏见和模式化的见解，好的一面就是它可以简化生活。如果一个人对于汰渍洗涤剂的态度是它很好闻、能去掉污渍、清洁得很好，那么他可以抛开一长串的原因和其他洗涤剂，做出简单的行为“就买汰渍”。人类是“认知的吝啬鬼”；我们试着去用最少的思考来完成工作。

重要的是，态度也可以有关事件，有关其他态度的态度，甚至对某人对态度反应的态度，等等。当处理人类态度时，事情可以迅速变得复杂。

态度也是其他形式的人类认知的基础。对信仰的一种看法是：它是关于一个论点的一种态度。一个人可以相信“地球是平的”（论点可以是真的或者是假的），或是“Bobby 骗了 Danielle”（同样可以是真的或者是假的）。

态度证明了什么是所谓的意向喜好。意向喜好（态度的基础）和暂时喜好不同，后者是对一些事物的迅速感情反应。态度是对态度对象可估价的信念；它们通过感动、行为和认知来形成。感动是感情反应的技术称谓。一些态度因为与态度对象有情感上的相关经验而形成。行为也就是动作；在一个好的餐厅愉快就餐的经历可以带来用餐的正面态度。认知也就是想法。思考事物或问题可以形成态度；思考自由在人类事态中的影响，会带来对自由正面的态度，相应地，也会对某些政治和哲学带来正面的态度，因为它们会带来更好的自由。

态度在与世界交互的过程中积累。经历、使用或者思考态度对象时，态度就会形成。迅速的喜好或厌恶的反应会附加到以前的经验，来形成态度或让态度更坚决。

意向喜好和暂时喜好的区别也许看上去微不足道并且不重要，但是，后面你会看到态度

或多或少地会更持久，虽然它们可以被暂时经验所修改，但它们比那些经验更持久。

严格地说，这些讨论并不是完全正确的。我们不能直接读出人们的态度；相反，我们基于我们的观察臆测人们的态度。态度对象（可观察的）有多外露会带来内部态度（不可观察）的激活，然后（虽然不是不可避免的）又带来一些可感知的态度（可观察的）的外部表达。态度的表达可以有很多方式，比如脸部表情、姿势以及其他不是口头上的交流，语言和实际的行为。

复杂的态度对象，如人、历史事件和组织，可以用态度对象的很多有意义的属性来评估。你可以同时爱她的酒窝，恨她的厨艺，适度地厌恶她的政治观点，欣赏她对电影的品位，痛恨她的大笑。这个概念会在后面我们说到爱/恨时说起。

同样，因为态度表达的是某个人对某样东西的评价，是完全主观的，所以一个态度实际上不是正确或错误的立场，即使它被强烈地感觉到了。

态度经常（虽然不是总是）带着感情的“指示”。一些态度是完全理智的，但其他的是剧烈感情经验的结果，而且这些会成为最持久的态度。在这种情况下，情绪上的（感情的）反应（顺着吸引/不吸引的轴）就不是认知的——我们不思考我们的反应，而是迅速和本能地用经验来判断。这是因为这样的反应涉及大脑中更多的处理感情的区域。感情同样连接着记忆；高度情绪化的事件会被很牢地记住。这就是创伤后应激障碍发生的原因。同样，这也是为什么我们这么容易记住在 911、挑战者事故或是肯尼迪被刺杀时我们在做什么的原因。

3.4.3 态度里有什么

幸运的是，通过从严密的社会科学中取得一些小心的自由，一个人可以制造一个可被 AI 游戏程序员使用的、合理的轻量级态度模型。事实上，有些很像态度模型的东西已经被一些游戏使用，也许最值得一提的是 XBOX 上的游戏 Fable[Russel06]。Greg Alt 和 Kristen King 提到了早些时候在 Ultima Online 系列中使用过的方法[Alt02]。

在态度系统中，不需要在每一帧都执行更新数值的计算。事实上，对于态度系统，有件好的事情是：态度只有当游戏中发生了可以影响态度的事件时才需要刷新。Russell[Russell06]把它们叫做选择事件。更普遍地，它们在被称为戏剧跳动的游戏故事点发生。这些发生的频繁度有多大？在 Facade 中，Michael Mateas[Mateas02]估计大约每分钟发生一次。所以态度系统不会给 CPU 开支带来压力，除非你正在处理特别多的带着大量态度的 NPC。如果这样，更新可以分布在多个帧中，不会带来太多的问题或者导致它们无法同步。另外，我们可能需要更多地关注内存的使用。[Alt02]详细地讨论了这个问题。

让我们假设每一个可以持有态度的代理实际上会持有游戏所要求的包含有很多态度对象的态度集合。很可能的是，主要的态度对象会是人类玩家或者玩家人物（PC），它们形成了意见系统的基础——游戏中的 NPC 是怎么用一些策划认为重要的度量来考虑玩家的。

这里的代理可能是游戏中所有重要的 NPC，但也同时可能只是代理的集合，比如整个村庄甚至游戏世界。表达其他形式的组织的代理一样可以有态度。这就是 Fable 使用的方法，和 PC 有关的态度信息是全局存储的（叫英雄状态），然后是在 Albion 游戏世界中每 10 个村庄的村庄级别上存储，然后是每个重要的单独 NPC[Russel06]。这种方法带来了很多重要的实现益处，在本文中都有涉及。

价（Valence）

一个单独的态度应该包含什么数据？显然，第一个应该是喜好/厌恶的可估价度量，通常这个值被称为价。作为开始，每一个态度都至少需要带着一个整数或者浮点数值来表示这个度量。很有可能它会是个双极值，来表达完全的“喜好/厌恶、爱/恨、满足/不满足、同意/不同意”的范围，以及一个评价可能被表达的其他方式。在一些情况下，一个单极的数值可能更加适合，你会在后面看到。

价可以存储为一个单精度浮点数，特别是如果游戏需要可以随时间变换态度的能力（说服）。但是对于许多应用来说，一个整数就可以很完美了。如果预计需要为很多 NPC 存储很多的态度，可以用 4 个比特来存储价，产生−7～+7 的范围（第 16 个未使用的值−8（1000b）可以保留作为哨符值）。

然而，价要怎么来定范围可没有那么明确。最简单的方法是使用− 1.0～+ 1.0（并且在计算中归一化整数值，似乎它们是跨越这个范围的），并且进一步假设喜好/厌恶在那个范围内是线性的。事实上很多系统都暗中这么假设，但这是真的么？首先，从研究中并不清楚喜好/厌恶的反应可以跨越多少级的量值。如果“轻微的”厌恶是 0.1，那么是否意味着“盲目火热的憎恶”就是 1.0，或者是−10 或−100？下面有两个可能的选择。

- 一个方法是仍然把评估值范围设置为−1.0～+1.0，但是把 S 型曲线（sigmoid）作为反应曲线。Chris Crawford 建议在故事讲述系统[Crawford04]中使用这个方法。
- 另一个可能是让评估值在很小的范围内，比如{1.0…5.0}来表达对数形式中的 5 个级的量值，就像里氏或者分贝。“轻微的厌恶”是−1.0，强烈的厌恶是−2.0，中等憎恨是−3.0，盲目火热的憎恨是−5.0。这里的描述性词语只是简单地给出每个级别表现的大约概念。它们并不打算表达英语描述的线性（这又是另一个未解决的问题）。

有些情况下单极值表示也许会更合适，但是在大多数游戏设计情况下这不会发生。比如，在理性情绪心理学中，爱的反义不是恨，而是冷漠[Ellis75]。在这种观点下，恨不能作为爱的反义，因为它也需要一个强烈的与态度对象的纠缠，而且并不是 180° 的转变。恨的反义也是冷漠。

这里要谨慎。如果你开发的游戏的本质需要对这种细微情感进行建模，那么是否需要花大量的时间来开发一个这样的模型就取决于任务了。

效力（Potency）

第二个可以在态度中存储的是效力，它用来度量态度是多么强烈。效力和价不同，它可以是非常强烈的政治中庸，有一个近似于 0.0 的价，但却感觉十分强烈。这种情况的发生是因为一个态度表示的是一生中与态度对象接触的积累，但却表达为一个瞬间的值。当与态度对象的接触不断积累，假设每一次接触不会离现在的态度太远的情况下，态度会变得更加难以改变。一个人第一次吃芽甘蓝（一种蔬菜）也许会有瞬间喜好+0.2。最后，随着吃芽甘蓝的积累，意向喜好（态度）会变回为+0.16。它会一直保持在那儿，除非那个人经历到某些特定的对芽甘蓝的超群或是可怕的（甚至创伤的）经历，而强制导致一个戏剧性的重新评估。

省略效力的度量是有可能的，但是如果你这么做，你会需要一些其他的技术来减弱那些巨大并不可信的态度的转变，否则会发生在部分 NPC 上。

这导致了另一个意见。在现实生活中，一个人有时会从根本上重新考虑他的私人方面的

位置或身份，在这过程中会经受相当大的感情动荡。比如，一个人知道她的兄弟被发现是很多极度可耻的谋杀案的罪犯，突然，她的生活中对她兄弟积累的感觉和态度开始崩塌，她也许会穿过著名的否认/愤怒/罪恶/屈从/和解的序列，而经历或者跳过这一切的速率是以前所无法想象的。

如果你的游戏有一些严重的背叛或不忠的情节，任何实际的态度系统也许都会在这些情况下发生故障。这里，最好的劝告也许是避开。态度类需要实现一个方法，可以强制地重置价和效力到任何想要的数值。然后，当戏剧性的时刻发生时，游戏脚本系统只需要做两件事情——使用那个方法来强制地重置所有被影响的态度，态度持有人需要被调整为更适合的态度数值，然后播放精心构思的动画来表现 NPC 正在经历很大的情感动荡。人类极端的情感忧伤时期是非常复杂的，对于现在来说，尝试在游戏 AI 中来模拟它们是不现实的。

持续时间（Duration）

随着时间的逝去，人们会忘记一些事情，并且极端的态度和不好的记忆会随着时间变得缓和。在很多情况下，我们也许想要允许态度变弱或者消失。相比保持对正式理论的信念，我们更关心的是获得真实的 NPC 行为。与大多数这种问题相同，真实人类的状况比我们希望这些模型所能做到的更加复杂。

持续时间可以用很多方式来表达，因为衰减会或多或少地遵循对数形式，所以半衰期（half-life）的方法是一种表达的方式。如果游戏只跨越很短的时间，那么态度和记忆会保持新鲜，持续时间会一起被忽略。但是，如果游戏时间会跨越很多年，就可能会需要一些衰退函数。如果游戏打算跨越 8 个游戏年并且如果记忆每两个游戏年衰退 50%，那么在游戏结束时，在游戏开始时的态度会衰退到它们本来强度的 1/16。当然，其他用来衰减态度强度或价的方法也是可能的。

另外一件要考虑的事情是 NPC 的个人因素。你也许想要它们中的一些带有怨恨，对于它们来说，应该设置很长的半衰期（half-life），以至于它们的负面态度只有很少的（甚至没有）衰退。[Russell06]描述了在 Fable 中这个问题是如何被解决的。

模型

从这些要素中，态度构建可以被实现为如下面这样一个轻量级的类。

```
class CAttitude
{
private:
    Entity* target;      // 指向一个游戏实体的态度对象
    Valence valence;     // 需要 Valence 的类型定义
    Potency potency;     // 需要 Potency 的类型定义
int months;           // 作为半衰期的游戏时间（按月计算）
public:
    CAttitude (Entity*);
    Decay;               // 计算另一个月的衰减
    Float Product;       // 价 * 效力
    // 等
}
```

3.4.4　复杂的态度对象

态度类中经常需求的一个有用的方法是：当处理复杂态度对象（比如 PC）时，可以从它们具有的多个属性上获得一个全面的或总的评估。

首先，对于线性表示来说，似乎一个归一化的 SRSS（Square Root of Sum of Square，平方和开方）就可以工作得很好。游戏引擎的 SDK 甚至可能已经提供了方法来从基于属性的评价值的向量中计算这个数值。

但是，这不是理论所需要的。每个态度的价乘以每个态度的效力，然后乘积相加，再做归一化，这个数值会更好：

$$A_{\text{tot}} = \frac{\sum_i A_i \text{valence} * A_i \text{strength}}{\sum_i A_i \text{strength}} \tag{3.4.1}$$

如果你正在使用一些其他的表示，比如 Sigmoid 非线性函数或者对数曲线，产生这个总值自然需要更多的运算开销。

一个简单的例子

考虑在一个游戏中，玩家与一个小的但是残暴的部族战斗了一段时间。敌人 NPC 是很好的战士并且足够聪明，知道什么时候撤退或消失，而择日再战。也就是说，在这种游戏类型中，一个典型兽人宝藏中的一个典型兽人的存活时间大约是 11s，而这些个别的 NPC 的持续时间则要长得多。

图 3.4.1 展示了一个十分简单的 FSM（有限状态机）[Schwab04]。当然，如果那些战士真的要像描述中的那么好，这个 FSM 也许不是一种合适的选择。一个更新、更方便的算法也许会是行为树（实际上是一个 DAG）[Isla05]。但是为了说明观点，我会使用 FSM。

如果没有使用意见或者态度系统，你可以设置一个像这张图里的 FSM，选择和调整不同的 A（接近距离）、R（撤退距离）和 H（生命值）的数值，来为战士生成有细微不同的行为。你仍然可以使用这个 FSM，但现在我们要添加态度系统。

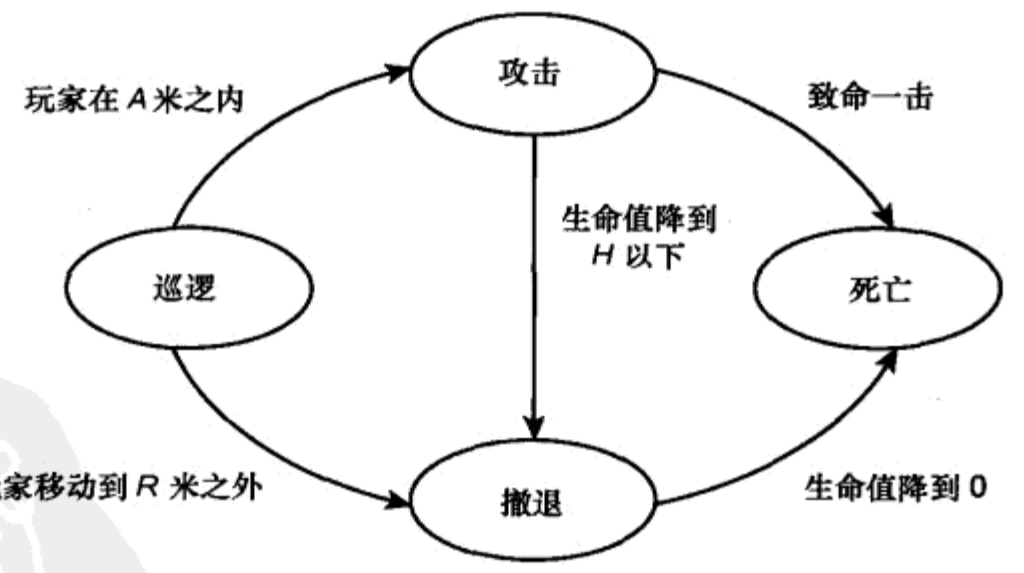


图 3.4.1　一个战士的简单 FSM：A 是接近的距离，R 是撤退的距离，H 是生命值

在这个例子中，每一个战士都痛恨玩家（在某种程度上），并且惧怕玩家（在某种程度上），因为态度系统会给予每个战士对玩家的痛恨和惧怕的数值来作为态度。如果玩家杀掉其他战士的一个兄弟，那么活下来的战士会加强他对玩家痛恨的态度，并且可能也会改变惧怕的数值（也许依赖于玩家在杀掉他兄弟的战斗中表现得是英勇还是胆小）。从那开始，活下来的战士可以交流他加强的憎恨和恐惧，首先是告诉他的兄弟（亲近的人），然后，就像池塘里的涟漪，传播给更多部族里的成员，但其强度降低了。我们会简单地讨论如何实现这个部分。

你现在有一个基础，可以通过向上或向下修改 *A*、*R* 和 *H* 的初始设定中的态度数值来动态改变 FSM 的行为。惧怕的战士会在更高的 *H* 数值下就撤退。充满仇恨的战士在停止攻击之前会需要更远的 *R* 距离，并且也需要更远的 *A* 距离，理论上这些充满仇恨的的战士会更像是监视环境，等待玩家返回。

即使是这个意见系统的简单应用也可以丰富其他的简单模型，但是你不应该满足于此。图 3.4.2 展示了一个扩展的 FSM，其中添加了额外的节点来模拟成为难以和解的敌人的战士。现在如果仇恨的级别 *D* 上升得足够高，并且恐惧的级别降得足够低，到达了突变点，这时这个战士的 FSM 就转换到一个新的模式，在这种情况下，战士会追捕、战斗，并且在与玩家的战斗中只会短暂地撤退，直到两者中的一个被杀掉。你可以继续添加 NPC 的信任度，因为对一个战士来说，到达与玩家进行私人战斗的地点无疑是可信的。注意，你现在也开始侵入了游戏设计领域。一旦一个战士转换到了难以和解的状态，游戏玩法本身就会改变，这种决策也许会在脚本层被更好地处理，而不是在 NPC 的 FSM 中用代码实现。

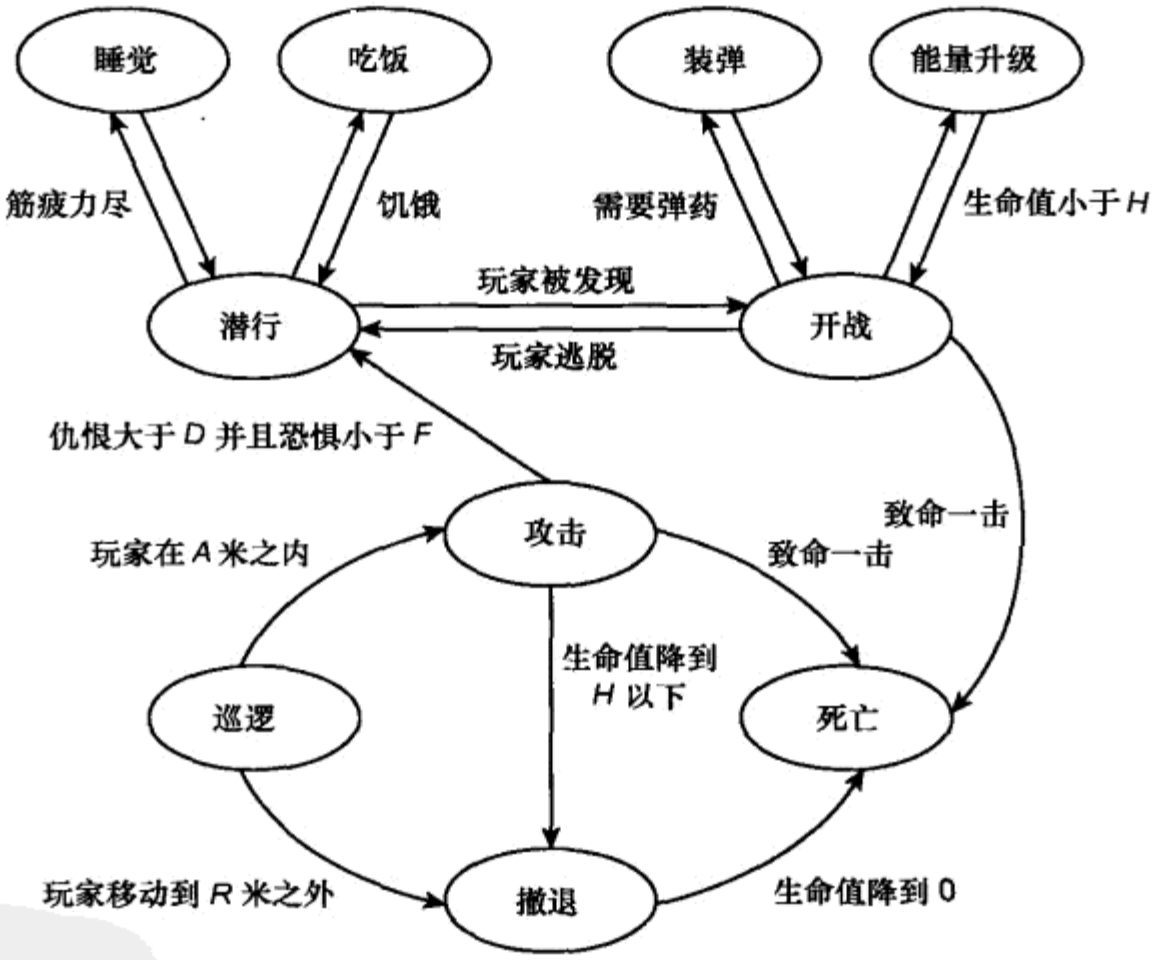


图 3.4.2 一个扩展的 FSM，态度在节点中影响转换：*D* 是仇恨的级别，*F* 是恐惧的级别

所以，你也许会想：“你正在创造一个梦魇，不只是为 *Q/A*，也为脚本和关卡设计师，他们都要处理这些添加的复杂度！”这确实是真的，但更多的像人类的 NPC 会被探寻，所以很难改善这种情况。很清晰的是，对于设计、测试和调整这些 NPC 的更好策略会是挑战的一部分。

3.4.5　态度和行为

这是另一个拥有广泛理论（和与其相随的论战）的领域。当然，一个人类对于她所遇见的每一件东西都拥有许多种态度，并且只有在一些例子中是特定的态度（在给定的时间和地点），接下来是一个可观察的行为。但是，在游戏环境中存储那些造成玩家观察不到的相应行为的态度是浪费的。

你只想在有用的地方使用态度的数据结构，这意味着再次避开了许多理论来获得让工作完成的最小配置。

所以，首先让我们引入两种新的方法到我们的模型中。首先提出的概念是对行为的态度。是的，这是可能的。所有 NPC 可以做出的行为都可以关联一个态度，这个态度表示了 NPC 感觉这个行为是否可取的趋向和程度。在一个模拟社会的游戏中可能会出现谋杀，但是一个特定的人物可能会认为它是完全反面的而不予考虑，或者只有当挑衅到了极端的阈值时才会犯下谋杀。

下一个概念是行为意向（Behavioral Intention， BI）。BI 在态度和行为之间。在这个模型中，一个行为发生前必须要有一个 BI 连接到它。这个可以被证明是一个受欢迎的简化，但是它把自己借给行为树或其他 NPC 行为模型来使用。让我们看看它实际是怎么工作的。

假设一个 NPC 已经有足够正面的态度来帮助玩家。这个 NPC 有一个复杂的脚本树，包含很多可能的帮助行为（比如，给予金钱、分享秘密或者安排住宿）。但是，这些行为只有在条件正确时才会发生。给予金钱和分享秘密只有在玩家接近 NPC 时才会发生，而安排住宿可以预先完成并且还可以在远处（玩家可以在客栈中出现，以为不得不乞讨住宿的地方，却被告知一个房间和饭菜已经被准备好了）。给予金钱或者安排住宿只有在 NPC 有足够的金钱来给予这些礼物时才会发生。分享秘密只有在 NPC 拥有有价值的秘密来分享时才会发生。

BI 可以被用来事先准备好那些帮助行为，接下来依赖于测试条件。所以，如果 NPC 对 PC 已经有足够正面的态度，NPC 可以形成一个 BI 来帮助 PC。BI 接下来测试在行为树中的规则栈中的 IF 部分，来决定是否也满足其他条件。如果满足，行为可以真正地发生。

3.4.6　说服和影响

这又是一个复杂且难懂的理论部分，你同样需要减少其中的理论。需要减少多少决定于你在建造什么类型的游戏。需要模拟政治斗争、广告或宣传战的严肃游戏，会比不需要这些的游戏更有必要地需要这个模型。

在理论中，一个人或者 *A* 组说服 *B* 组来采取或变换位置（也就是采取或者变换态度）的效率，依赖于大量的布置在偶然事件中的因素。说服者 *A* 通常被称为发送者，说服信息的目标被称为接收者。发送者至少需要对接收者来说是可信的并被接收者所喜欢和熟悉。接收者必须要注意信息，可以处理信息，并且完全知道信息是关于什么的。信息本身可以是原因、情感，或者两者都是。如果表现为原因，它必须在接收者听起来是合乎逻辑的。同样，如果说服信息提倡的态度对于接收者现在的态度来说太过极端，而超出接收者可以接受的立场，那么说服信息就很容易失败。

在游戏中，你可以改掉很多这样的例子。首先，让我们假设在游戏中的说服性的沟通都是重要的，接收者总是会发现它们很重要，注意它们并且可以理解它们。这会迅速去掉 4 个变量。发送者被分成更少数目的组，可以瓦解可信度、喜好和相似度。接下来有必要维护一个矩阵来记录 *A* 组有多么信任 *B* 组的信息。

注意，这个矩阵表示为 $N \times N$ 的态度，因此我们也可以看到：随着游戏进行过程中组之间的交互，矩阵的数值一直在改变。如果这对于游戏来说过于复杂，并且这样的态度将不会被改变，那么你应该用静态数值来组成矩阵，并且在游戏中不用管它。

这个矩阵也可以通过为特定的个体（同样的组成员）添加条目来补充。这允许 *A* 组的成员大部分不相信 *B* 组的信息，而允许 *A* 组的成员不情愿地接受人物 *P* 的信息（除非 *P* 正好是 *B* 组的成员）。[Alt02]提供了关于这个概念的详细实现。

最后让我们假设你不会担心信息是怎么构成的，只是简单地认为它带有内容和说服强度。继续我们的例子，存活下来的战士强烈地痛恨 PC，这可以通过一段游戏过场动画表现出来，他会向他的伙伴战士们进行高谈阔论，来说明为什么杀掉 PC 有最高的优先级。在游戏内部，信息被简单地传达，并且伙伴战士的态度被合适地调整。

3.4.7 态度的社会交换

以前提过，被战争结果影响的战士可以与他喜好的组群交流他对玩家的增强的痛恨和恐惧。此外，如果他和他的兄弟很亲近，并且和他喜好的组群也很亲近，那么那个组群的成员大概也会喜欢死去的兄弟，并且会赞同存活的战士的信息。这就进入了平衡理论[Wikipedia07]。平衡理论是社会网络建模中很强大的一部分，也是在这篇文章范围之外的另一个人类行为方面。使用上一个部分的矩阵方法（也许需要一些增强），就可以解决大部分这种问题。

在很多游戏中，只有这是重要的：不同的 NPC 都只对玩家有态度，而彼此之间不持有态度。如果在游戏设计中没有理由来保持 NPC 之间的态度信息，那么不理会它们会大大简化设计。事实上，[Russell06]直言 Fable 只存储了对 PC 的意见，并且没有对 NPC 进行复杂的控制。在 Fable 中使用的意见系统本质上是基于态度结构的，只是没有使用那个名字。

然而，如果游戏玩法包括联盟、不忠和背叛，那么一些游戏设计会从这样一个系统中获益。这么做，增加的复杂度会让你头大。

这样的一种复杂度是完全算法开销。当然，对于可以有态度的 *N* 个角色来说，有 $N \times (N-1)$ 种配对，产生 $O(N^2)$的复杂度。你不用把这个结果除以 2，因为假设 *A* 对 *B* 的态度与 *B* 对 *A* 的态度完全对称是不安全的。一个策略是通过分配更少的关键 NPC 到更小数目的组中并跟踪它们，以此减少 *N* 的大小。

3.4.8 另一个例子

考虑在一个假定的动作冒险类游戏中应用态度系统，而这个游戏是一个开放的世界。在这个世界中居住着敌对的犯罪组织和肮脏但多样化的 NPC，游戏的目的是与 NPC 竞争，游戏方式是做肮脏的勾当，加强流氓们制定的规矩，扩张地盘、赚钱、转换阵营，甚至有时当

对你有利时背叛或杀掉 NPC。

作为 PC，你的行为会让 NPC 尊敬或厌恶，他们对你形成意见，从而让你爬到顶峰。Fable 中使用了 5 种度量：道德、名望、恐惧、愉快和吸引。在这个例子中，我们使用这样一些度量：有能力的、受尊敬的、无情的、有魅力的和忠诚的。

从理想上来说，选择的属性应该尽可能地正交，每个属性都静态独立于其他属性。这 5 个属性看上去是满足要求的。也许包括受尊敬的属性看上去有些奇怪，但是可以把它看成是指示无辜的居民（尤其是谁的母亲）不会被伤害的行为代码，这些人包括任何在路上从你身边走过的人，但不包括店主、赌徒、酒鬼和嫖客。

假设游戏设置了几个不同的竞争犯罪组织，每一个都有自己达到统治的风格。第一个为喜欢厚颜无耻、公开暴力的小头目所领导（他也许在早期就死了）；第二个喜欢向警察和法官行贿；第三个喜欢把他的组织伪装成合法的公司。每一个头目都喜欢按照是否符合他的组织风格来评价玩家或任何 NPC 的不同模式的态度。

当玩家逐步建立起他的地位时，不同的头目会想要评价这些可感知属性的不同组合。一个喜欢保持秘密暴力的头目，作为最后的对策，也许会避开对他的风格来说太鲁莽的玩家。一个搞砸了太多工作的玩家会发现他的感知能力已经恶化，那些会有利于他的职业的重要任务会被停掉，这导致这个玩家不得不做更多的低级别任务来获得组织上面的那些老大对他的好印象。

只需要少数的属性/态度的度量来让这种游戏玩法变得可能。在 Fable 中，意见系统和英雄状态只使用了上面提到的 5 种度量。即使你只考虑每个度量的高低程度（前提是极端的人物更有趣），这也会带来 32 种关于可能的玩家如何被感知和玩家声望如何被建立的组合。

这些度量的其他组合可以是游戏策划构建剩余 NPC 的个性的一部分。毕竟，即使是一个愚蠢但无情同时高度忠诚的 NPC，对于犯罪组织来说也会是有用的。

3.4.9　注意事项和结论

在这时，你也许会想，为什么不把像这样需要 NPC 的游戏做成多人游戏，来让真正的人类玩家处理所有的复杂性呢？在某种程度上，多人游戏是可行的，并且在很多游戏中使用，但是这个方法的问题在于玩家创造内容的相似性——大多数（也许 95%）质量太差而派不上太大的用场，更不用提趣味性了。不是所有的人都是完美地讲故事的人、角色扮演者或是即兴演员，再者，它包含了很多工作。看上去这会是游戏策划和其他有才华的人的命运，来创造玩家需要的丰富的游戏世界和可信的人物。

本精粹介绍了一些基本的关于态度心理学的概念。态度只是那些最复杂生物（人类）心理的一个方面。但是，这是一个很有用的开始并且触手可及。幸运的是，一个单独态度的表示可以是十分轻量级的，但是在一个有任意复杂度的游戏中，它们也可以变得很庞大。

随着越来越多的 CPU 和 RAM 的开支可以被用来分配给游戏 AI，游戏开发者会更需要从心理学、社会心理学和认知科学中获得大量的人类行为知识。好的消息是游戏开发者需要建造夸张的模仿而不是真人，挑战是采纳我们对真实人类所了解的信息并把它重构到现实中。

3.4.10 参考文献

[Agre97] Agre, Phil. *Computation and Human Experience*, Cambridge University Press, 1997.

[Alt02] Alt, Greg and King, Kristen. "A Dynamic Reputation System Based on Event Knowledge." *AI Game Programming Wisdom*, Charles River Media, 2002: pp. 425–435.

[Crawford04] Crawford, Chris. *Chris Crawford on Interactive Storytelling*, New Riders Books, 2004.

[Eagly93] Eagly, Alice and Chaiken, Shelly. *The Psychology of Attitudes*, Harcourt Brace Jovanovich, 1993: pp. 1.

[Ellis75] Ellis, Albert and Harper, Robert A. *A Guide to Rational Living*, Wilshire Book Company, 1975.

[Isla05] Isla, Damien. "Handling Complexity in the Halo 2 AI," GDC 2005 Proceedings.

[Mateas02] Mateas, Michael. *Interactive Drama, Art and Artificial Intelligence*, (Ph. D. Dissertation), Carnegie Mellon University, School of Computer Science, 2002, report CMU-CS-02-206.

[Russell06] Russell, Adam. "Opinion Systems," *AI Game Programming Wisdom 3*, Charles River Media, 2006: pp. 531–554.

[Schwab04] Schwab, Brian. *AI Game Engine Programming*, Charles River Media, 2004.

[Wikipedia07] "Balance Theory."

3.5　用玩家追踪和交互玩家图来理解游戏 AI

G. Michael Youngblood, UNC Charlotte
youngbld@uncc.edu
Priyesh N.Dixit, UNC Charlotte
pndixit@uncc.edu

随着游戏 AI 的发展，我们可以为游戏人物和游戏反应赋予可信的、有挑战性的、甚至有洞察力的行为。创建这样的游戏人物和游戏反应的能力得到了提升。但是，描述是什么让这些对象看上去真正有智能的基本能力却落后了。本精粹提供了一些关于特定可视化和基于图的 AI 分析理念，用一些工具和技术来达到更好地理解游戏中人工智能和人类智能的目的。本精粹展示了通过可视化数据挖掘，基于图的交互表示和聚类工具的使用，以玩家为中心记录的游戏数据如何被用来更好地理解自然和人工的玩家行为。

3.5.1　简介

游戏 AI 开发者关注创建单个或多个实体，或者一个系统，让参与的人类玩家可以感觉到一些形式的挑战。满足人类玩家的认知需求可以让游戏变得有趣和有挑战性，最后也会让游戏卖得更好。有趣的问题是：并不是每一个人类玩家的感知都一样，甚至他们的玩法都各不相同。对于一个玩家来说智能就是对另一个人来说沉默。这让游戏 AI 的开发十分困难，并且带来关于在游戏中是注重人与人之间还是人与 AI 之间的竞争的权衡的讨论。幸运的是，对于单个玩家来说，更好游戏体验的强烈需求导致了对更好 AI 的强烈需求（或者说玩家不断提出新的需求，使得游戏有必要采用人类智能级别的 AI）。目前，已经有很多关于人类智能级别 AI 的讨论[Heinze02、Laird01、Laird02]，但是仍然有很多关于如何决定你是否已经实现它的问题。

一个游戏 AI 图灵测试[Russell03]是不是一种决定一个实体是人类或是机器控制的有效方法呢？虽然它可能会被证明是一个有趣的尝试，但无疑会像其他主观方法一样陷入很多讨论和猜想。我们需要的是一个以人类在相同或相似任务或剧情的表现为基准来对游戏 AI 做出客观评价的方法。这种方法需要数据的支持。

现在计算机游戏的趋势是省去细节记录来为其他材料空出系统资源。

然而，这种数据可以通过提供对人类和机器玩家行为的见解来加强交互游戏的体验。

3.5.2 信息的价值

游戏中的记录通常被连接到保存功能，因为这些子系统通常会跟踪玩家的进程。然而很多游戏不记录玩家的信息；这种趋势变得越来越坏，你可以注意到很多游戏中使用checkpoint系统来存储，而不是在任何时候都可以存储游戏。这意味着游戏只有在玩家到达关卡中的特定地点才可以存储。使用这种方法，开发者不需要每时每刻地跟踪玩家的状态，只有在他们到达特定里程碑时才会跟踪。Eilers [Eilers05]提出，在玩家刚开始玩这个关卡的那几次，这个方法是没问题的；但是，一旦玩家已经记住了这个关卡之后，它就会变成一种障碍。它会为玩家制造麻烦，使玩家不得不在已经征服的区域重复劳碌一番后才能到达新的挑战。它也阻碍了真正记录的能力，因为现在在游戏中没有一个进程来监视哪里是很好的记录点。

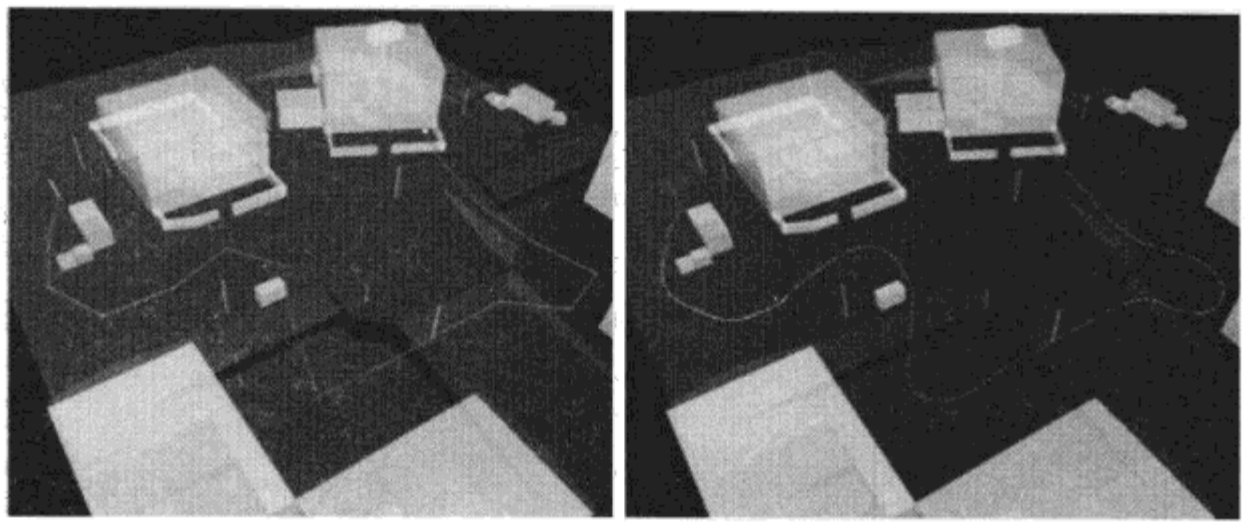
在玩家测试过程中，记录是十分有用的，因为它可以让这个过程变得更加有效率。开发者团队可以用记录视频的方式来捕捉测试者的行为，但是观看所有的视频通常会比较费时，分析最终会十分主观，视频消耗很大的磁盘空间，而客观、自动的视频处理是很困难的过程。直接观看视频会在观察者的意见中产生偏见。视频或屏幕捕捉的另一个问题是：只从玩家角度获取游戏的完整描述会很困难。观察玩家交互的整个路径或想要的部分，或在交互分析工具中的特定观点，都会对理解玩家的行为十分有用（无论它是人类还是机器）。

一系列好的记录数据和一些分析工具（包括可视化工具）同样可以帮助找到突发行为或者没有被开发者预料到的行为。这些不一定是错误，而是有趣的“事故”，使得游戏区别于非交互的娱乐形式[Consalvo06]。

排除或者过度简化数据记录最常见的理由是它会减慢游戏速度。比如，在“帝国时代II：国王时代”的开发中，开发者为了试玩测试版，需要使用比玩发布版本更快的机器，原因就是记录玩家测试数据会减慢速度[Marselas00]。不是所有的情况都是这样的，因为对于数据记录来说，开销最昂贵的是文件IO，但这个可以在硬件空闲、过场动画或被设计的间歇中执行，或者使用多线程而不是在游戏中不断地写文件。对于记录来说，也会有需求的精确度问题。我们可以从1Hz或者更慢的记录中获得足够的信息，所以10Hz或更高也不总是必要的。

在过去的几年里，我们已经对我们自己创建的一些游戏试验台进行了AI研究，一些是从头开始的，另一些是在商业游戏上修改的。我们经常使用城市战斗试验台（UCT）。UCT是对Id Software开发的流行的Quake 3的一个完全的转换mod。Quake迷的用户数让它十分容易就找到了研究的参与者（特别是在Quakecon上，总是玩得很愉快），并且我们已经捕捉了成百上千的玩家在我们的游戏脚本中交互。

我们经常以10Hz来捕捉记录，但是你可以在图3.5.1中看到，1Hz捕捉到的信息也同样可以提供有用的信息。在人类计算机接口（HCI）分析工作中，10Hz的采样被广泛地使用，因为它对于人类使用一个接口设备来说是最快的正常反应时间。



（a）　　（b）

图 3.5.1　从 UCT 中获得的玩家踪迹数据，在 1Hz（a）和 10Hz（b）下展示了玩家的运动。细线展现了，被关注的实体在环境中随时间的空间运动

PlayerViz 是我们的分析工具，它的交互输出展示在图 3.5.1 中，在每个时间步，它需要包含下列信息的数据文件：

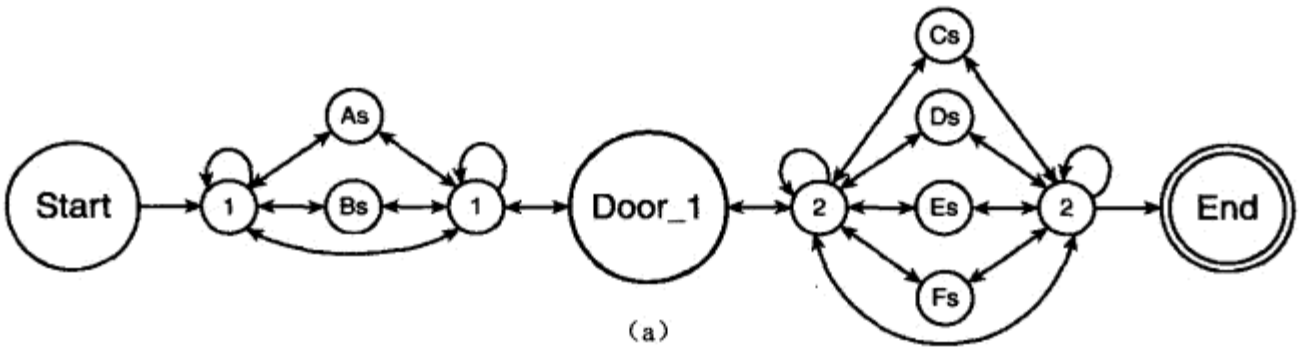
- 位置（x、y 和 z）；
- 方向（yaw、patch 和 roll）；
- 速度；
- 消逝的时间；
- 时间分数；
- 生命值；
- 发射的子弹数；
- 是否有旗被夺。

PlayerViz 工具包含在这本书的 CD-ROM 中。

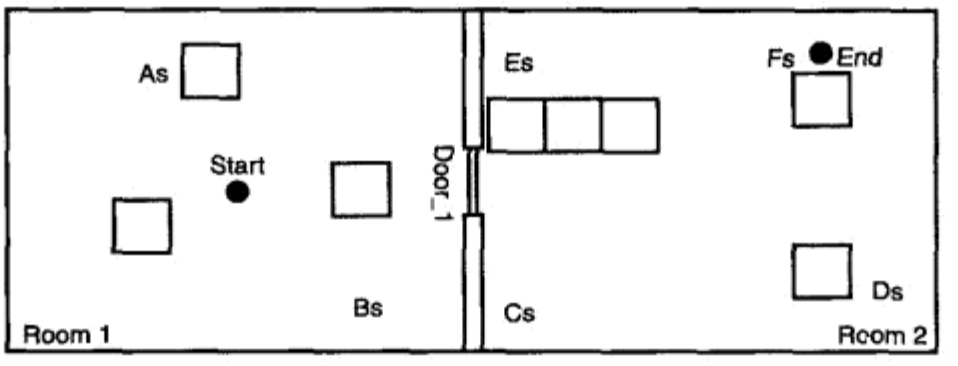
时间步的频率不是固定的，可以依据需求细节来调节。理想状态下，一个记录系统可以动态地调节记录频率，来按照可用的系统硬件能力最小化对游戏的影响。但是，即使只是每秒记录一次，对玩家行为做出分析也是足够的。记录的实现和记录下的情况会根据游戏而不同，但应该考虑对于知识探索和理解游戏中实体的智能行为来说，什么信息是容易获得并且可用的。

捕获交互是理解所有理性的 AI 在游戏中智能行为的关键。所以，你记录玩家与环境的交互功能点之间的交互[Youngblood02]。交互功能点是游戏中实体可以表现行为的元素（比如，开、关、推、跳过、代替、射击和套索）。这些元素也许会占有正空间区域，并且表达一个真正的世界或幻想对象（比如，窗户、门、树、公共马车、箱子和魔法药剂）；也许会是负空间区域，可以包含其他的游戏元素甚至是玩家（比如，一个庭院、在屋内、或在车上）；还可以是游戏中的其他 AI 代理（比如，一个相对的势力、一匹你可以骑的马或者你可以乘坐的龙）。一个玩家以任何形式交互的任何东西都可以被考虑为一个交互功能点。

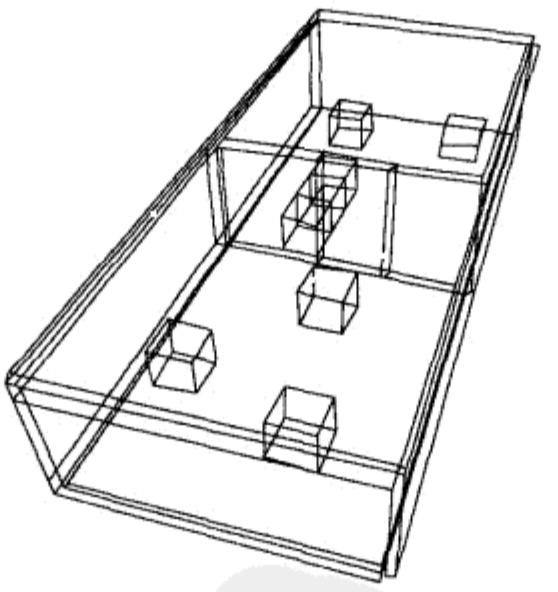
在真正的世界中，交互功能点的数量是无限的，但是在游戏世界中它们是有限的，并且由策划和游戏引擎的能力决定。所有在游戏和游戏情节中的交互功能点可以用交互可能性的图来描述，就像在图 3.5.2（a）中展示的简单 FPS 环境的实例一样。点表现的是交互功能点，边表示一个交互可以紧接着当前的交互发生 —— 一个可能的扩展是列举每个交互功能点的可能交互类型。在分析玩家踪迹时，这个图可以用来从记录数据中查找无用的交互或者设计问题。



（a）



（b）



（c）

图 3.5.2 游戏中的交互功能点可以被用来生成交互可能性图，如（a）图所示。
这表现了 FPS 游戏场景中的交互可能性，其中（b）图为场景顶视图，（c）图为 3D 透视图

从游戏设计中为验证生成信息，从玩家测试和代理交互测试中捕捉信息，并创建有深度的知识作为分析中的新信息，这些强调了信息对于游戏策划和游戏中想得到的交互多么有价值，

而这些信息主要由人类和 AI 驱动的行为组成。了解游戏中行为的一个十分重要的因素在于能够可视化这些行为，如图 3.5.3 所示。

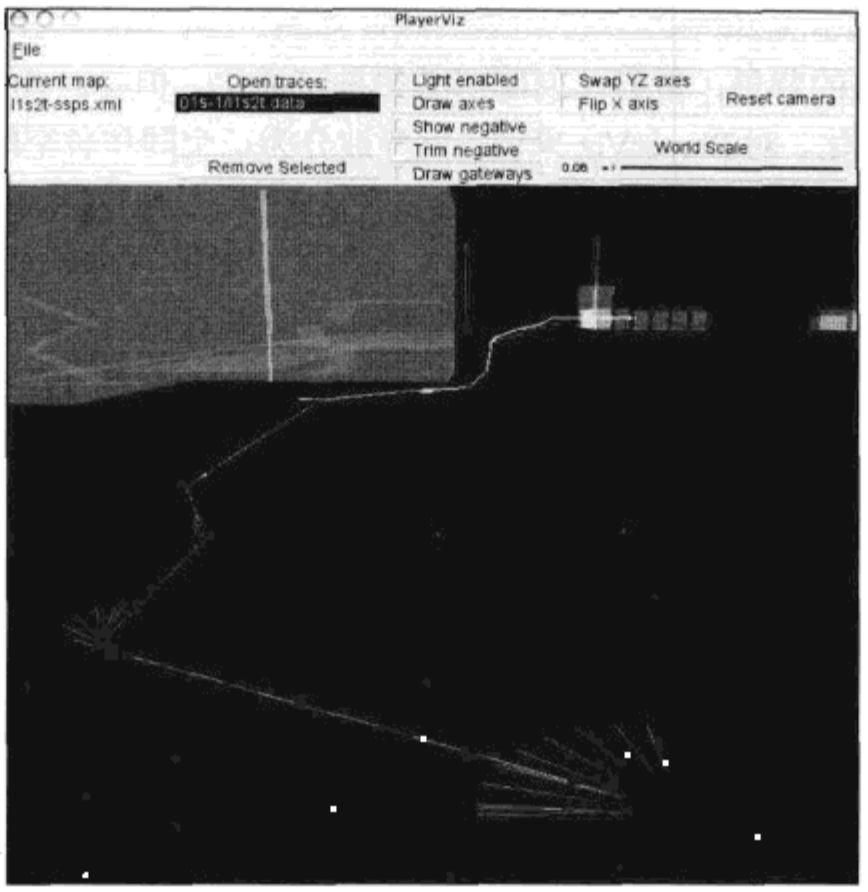


图 3.5.3　为可视化单人或多人玩家踪迹的 PlayerViz 工具的屏幕截图。
它展现了随着时间推移玩家的路径，其中球体表现位置，从球体伸出的线段表示方向

一张图胜过千言万语

从游戏记录中收集的玩家行为信息可以很轻易地变得巨大。有很多变量被同时追踪，包括时间、位置、方向和交互。如果玩家踪迹（player trace）是用可视化的形式来表达的，数据会更容易理解一些。一个玩家追踪用可视化的方法来表现玩家从头到尾的行为。PlayerViz 工具被设计来允许玩家踪迹数据的交互可视化表现，这些数据可能来自一个或多个玩家。如图 3.5.3 所示，一个玩家跟踪用一系列的球体和连接线来表示。球体的颜色以彩虹色表循环，随着时间过去从蓝变绿再变红，当移动慢些，颜色就会变得更快，反之亦然。球体的位置表示玩家的位置，玩家在那时的方向被描述成从球体出来的带方向的线段。一个线框形式的白色球体表示玩家找到目标，并且它通常处在踪迹的最后位置。玩家身边的一个红色的线框球体表示失去了生命值，线段末端的一个红色的实体球体意味着玩家开火。一个玩家踪迹提供了物理交互的很好的概述，但也许它不会总是应用到所有游戏中——一些游戏没有一个清晰的空间部分，比如，解谜游戏。

PlayerViz 同样可以被用来立刻检查多个玩家踪迹，可以提供一些有趣的结果。比如，如果玩家在一起玩，他们的集体动态是可以被研究的。你同样可以计算玩家踪迹的平均值来获得综合表现的概述。如果玩家是敌人，你可以检查他们采取的战术以及他们对其他人行为的反应。比如，如果两个玩家同时追捕同一个目标，观察每一个玩家选择的路径以及路径如何交叉就变得很有趣。玩家踪迹同样可以被用来表现 AI 代理的行为并且与人类行为做比对。

所有人类和所有 AI 代理的组合踪迹可以用来对比，以给出对这个游戏来说 AI 有多么像人类的大体概述。

世界数据的简化

为了提供玩家踪迹的背景，世界几何图形必须被可视化。但是，几何图形不需要表现很多细节并且可以被大大简化。对 PlayerViz 来说，世界被分为正空间和负空间两个区域。3D 建模者手动地指定正空间，它表现一个对于一个物体来说是外部世界几何形状的近似值（比如，建筑的包围盒）。这些区域都不精确，但是你只需要一个几何体的粗糙估计来提供玩家踪迹和世界的空间相关性。负空间区域可以是手动指定的，或者通过像细胞分解之类的方法自动计算得出。

为了简化负空间几何，你必须把世界分解为一系列凸的区域。有很多技术可以用来把世界几何体分解为区域。我们使用的技术叫做关键顶点细胞分解[Youngblood06]。这个方法需要通过在正空间物体（比如角落）上连接关键顶点来创建多面体，避免相交的线段，并且在保证多面体保持凸面的前提下联合相邻的区域来扩大它的尺寸。

因为负空间通常包含大部分的空区域，所以这些区域有很多无关的多边形。但是，如果你只渲染与地面粗略平行的多边形，你通常会获得有用的几何体。你同样可以在几何体的空间区域中辨识入口。这些入口被区域间共面的边界所指出。但是，为了入口可以很容易被发现，这些边界必须完全共面。这些区域和它们的连接入口同样可以被用来帮助 AI 代理进行路径计划和其他的的空间任务。AI 代理可以通过记录它访问过的区域来学习它周围的世界。观察这样一个 AI 代理的玩家踪迹在机器学习研究中十分有用。

为你的思想服务的一个像素

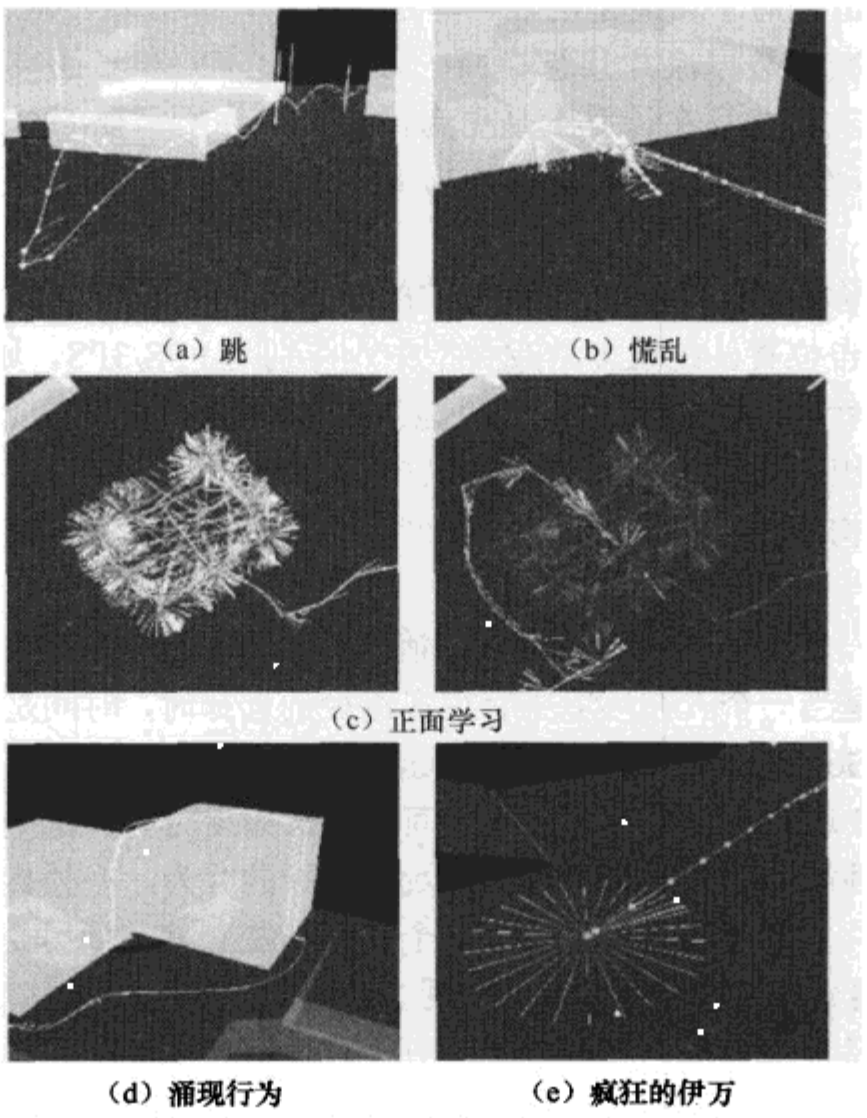
即使有了玩家踪迹可视化的帮助，考虑到如果有上百个玩家，要检查所有的玩家踪迹仍然是一个困难的任务。一个可以让它变简单的方法是使用可视化数据挖掘[Keim02]。PlayerViz 可以被用来生成一系列的网页，展示从不同角度的每一个玩家踪迹的缩略图。这允许用户迅速在资料中查找和发现有趣的踪迹，并且可以被读进 PlayerViz 来做进一步的检查。一旦想要的或异常的现象被可视化地辨识出来，用户可以实现一些方法自动寻找这些现象的发生，通常通过创建特定的功能寻找算法来实现。可视化数据挖掘主要是一种辅助程序来指引为特定的游戏创建更加特定的工具，但是它非常强大，可以帮助定义你应该寻找什么，而这个对于决定优先级通常是很困难的。

在我们自己的工作中，我们使用图 3.5.4 中的这些玩家踪迹图可以找到一些有趣的现象，而它们只通过看数字是基本不可能发现的。比如，图 3.5.4（a）展示了一个跳跃的人。这是一个喜欢连续跳跃的人，即使当他或她走在平面上。另一个例子，图 3.5.4（b）中展示了一个慌乱的人。一个慌乱的人在玩家踪迹中的现象是玩家看上去失去了对鼠标的控制。可能是因为对鼠标瞄准没有经验或者使用有问题的鼠标造成的。如果没有可视化数据挖掘，那么这些现象是很难被跟踪到的。

图 3.5.4（c）展示了同一个玩家的两个玩家踪迹。左边的图是源，或者第一次尝试，右边的图是目标，第二次尝试。用户可以很清楚地看到玩家从他上一次的尝试中学到了怎么走出封闭的区域。相反，也有可能玩家第一次找到了目标，但是第二次尝试却忘记了。因此，你可以使用这个技术来观察正面和负面的学习。

你同样可以用这个技术找到突发行为，就像在图 3.5.4（d）中展示的玩家踪迹。目的是

让玩家爬墙并且找到目标，但是玩家没有这么干，而是从房子的一边爬上来，并从房顶上跳了下来。如果没有视觉参考，这种行为同样很难被跟踪。最后一个例子是疯狂的伊万，一个玩家旋转 360°来检查他的周围，如图 3.5.4（e）所示。



（a）跳　　（b）慌乱

（c）正面学习

（d）涌现行为　　（e）疯狂的伊万

图 3.5.4　在玩家踪迹的可视化数据挖掘中发现的有趣现象的列表：（a）跳；（b）慌乱；（c）正面学习；（d）涌现行为；（e）疯狂的伊万

使用可视化的数据，你同样可以分析最常被玩家访问的区域（或最少被访问的），这允许游戏策划来决定他们的关卡设计是否符合玩家的经验。比如，如果游戏策划在地图的一个区域放置了一个线索，而只有很少的玩家去过，那么策划就会知道大多数玩家都不愿意去寻找它。

在计划中，要添加很多功能到 PlayerViz 工具中，比如，使用一些玩家的踪迹来计算平均玩家踪迹。平均玩家踪迹可以被用来迅速和简单地查看大多数玩家采取的大体路径。另一个功能可能是使用一些玩家踪迹来生成密度图，以高亮显示那些被访问得最多的区域。这个功能会告诉游戏策划以后的玩家会更喜欢去探索哪些区域，所以他们可以相应地放置交互功能点。

3.5.3　交互玩家图

理解空间轨道并且观察简单的时空交互提供了对所观察的理性代理的智能理解，但是交互可视化工具仍然需要分析师的大量手动工作。可以用来与其他玩家比较的交互表现在这里是有用的。游戏环境中的交互轨迹是个体在游戏中采取战略的典型表现。捕捉和比对战略的能力是十分重要的。交互功能点有限集合的建立，伴随合适游戏记录的设计交互可能性图的

引入和加强，它们一起就可以为每个游戏中的玩家生成交互玩家图。

一个交互玩家图（IPG）要么在游戏中，要么在游戏运行或完成的玩家记录文件的游戏后被建立。游戏环境需要捕捉想要的交互来建立图。图的顶点表示一个交互事件（比如，按下按钮、捡起血瓶、击晕对手、捡起钥匙、穿过一道门，或者站在地图的新区域）。IPC 可以十分的细节化，但是减少数据流并且忽略那些从运动、方向和其他难解的状态变化中产生的较小干扰，没有那么重要并且在计算上是更为可行的。你所需要的是关于一些玩家活动的更高抽象，那些玩家活动可以给你合适的解决方案，来理解同时减少那些会让分析停顿以及模糊意向的无用信息。

标准的流程是：你在细胞分解方法所确定的定量凸面区域中跟踪空间运动，当进入那些被标记的区域时就报告位置。在很多游戏类型中，特别是 FPS/3PS，如果跟踪的分辨率过高，正常运动的空间交互就可以在 IPG 中占主要地位。其他的交互一般会在它们发生时被记录。所以你从一个交互的组合中构建一个 IPG，其中组合是由在环境区域的粗略空间运动和玩家与特定物体的交互所构成的。但是，IPG 不一定要包含空间交互，因此在分析那些没有清晰空间部分的游戏中也十分有用。

图的表现形式的一个问题是在很多游戏中确定的计时问题，并且在一个情景下的表现中是真正的不同点，特别是当一个人在环境中有很多路径的选择时，时间是让不同玩家完成同一个任务的真正不同点。Gonzalez[Gonzalez99]和 Knauf 等[Knauf01]同样也指出了时间在人类模型验证中的重要性。IPG 通过测量交互功能点之间的边来捕捉时间，这些边的长度表示玩家在交互之间的时间。IPG 抽象一个玩家在游戏或游戏情景中的行为，在捕捉它们交互方法的同时，去掉在环境中的细节化的状态改变干扰，从一个交互功能点移动到另一个，并且捕捉它们行为的时间。通过使用与情景相关的空间分解图，图 3.5.5 展示了一个玩家踪迹被转换为 IPG —— 注意唯一表现出的交互是地图遍历；其他的交互只不过会用增加的交互顶点来扩充图。

还要注意的是，对于这里描述的基本 IPG 形式有很多扩展或变化。比如，如果使交互的类型和交互功能点发生联系，使每一个功能点表现为一个单独的顶点。代理会同样标记内部的状态为转换或交互。

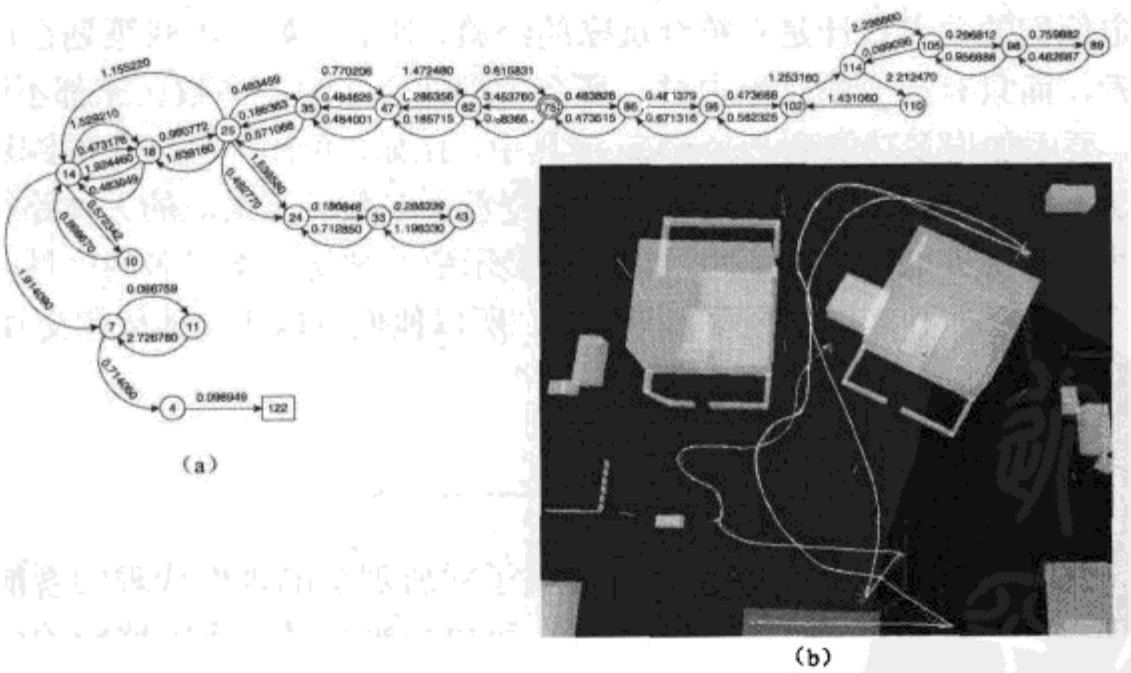
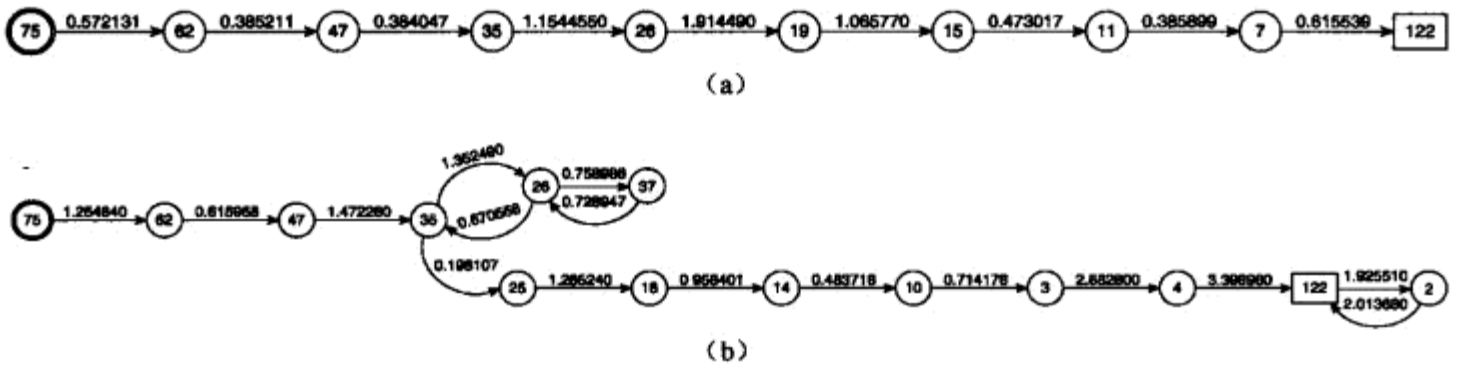


图 3.5.5 从玩家的记录数据中创建交互玩家图（a），并使用 PlayerViz 来显示玩家的踪迹（b）。顶点表示交互功能点（本例子中是到新空间地图区域的入口），同时边代表了在交互之间的时间权重

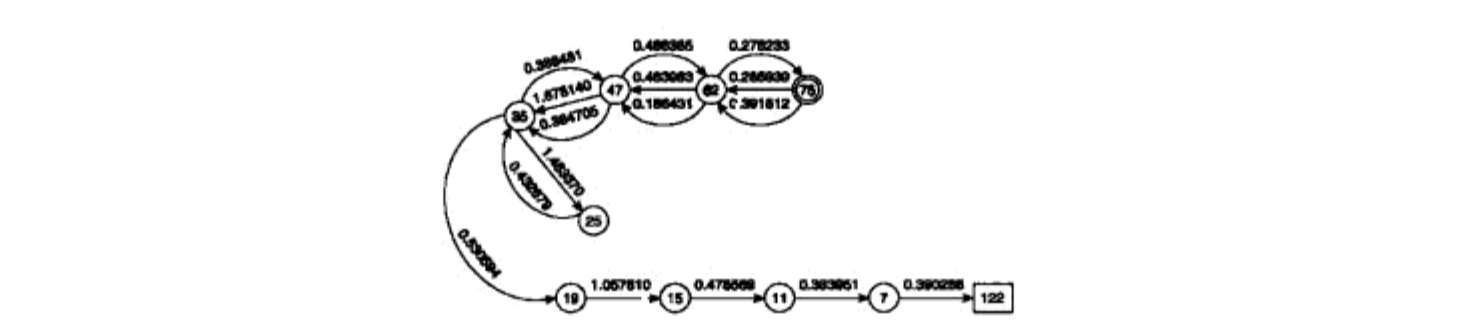
对 IPG 分类

图的形式的用处是它可以被用来与其他图进行比较。为了比较图，需要图的相似度的度量。我们建议使用图的修改距离（Graph Edit Distance），它被定义为从一个图到另一个图的最小数量的改变。一个改变被定义为诸如插入边、删除边、插入点、删除点、为边增加 0.1s 或为边减少 0.1s。每一个改变都带有一个相等的度量，可以通过改变它来减少偏差。因为时间是差别的主要因素，所以这个方案偏重于时间，在很多你评估的游戏中，时间的表现是玩家之间的主要差异。你可以用同样的方法但不带时间的度量来评估图，在时间度量中相应地也不带增减。

对比的一个目的是可以用玩家的 IPG 对它们进行分类，IPG 会本质地展现玩家在游戏或游戏情景中的战略选择和表现。图 3.5.6 阐述了即使在相同的游戏情景中，玩家也会有很多不同的交互轨迹。如果你对玩家的表现进行分类，你应该以玩家的相对技术等级来对它们分类[Youngblood02]。如果以玩家的技术等级分类，这个技术可以用来作为玩家分级。从 AI 的角度来说，更有趣的是，如果你用人类数据评价机器驱动的代理数据，并且让代理与人类玩家一起分类，那么你可以断言代理的玩法符合那个类别的人类玩法，或者那个代理的行为是符合人类习惯的。

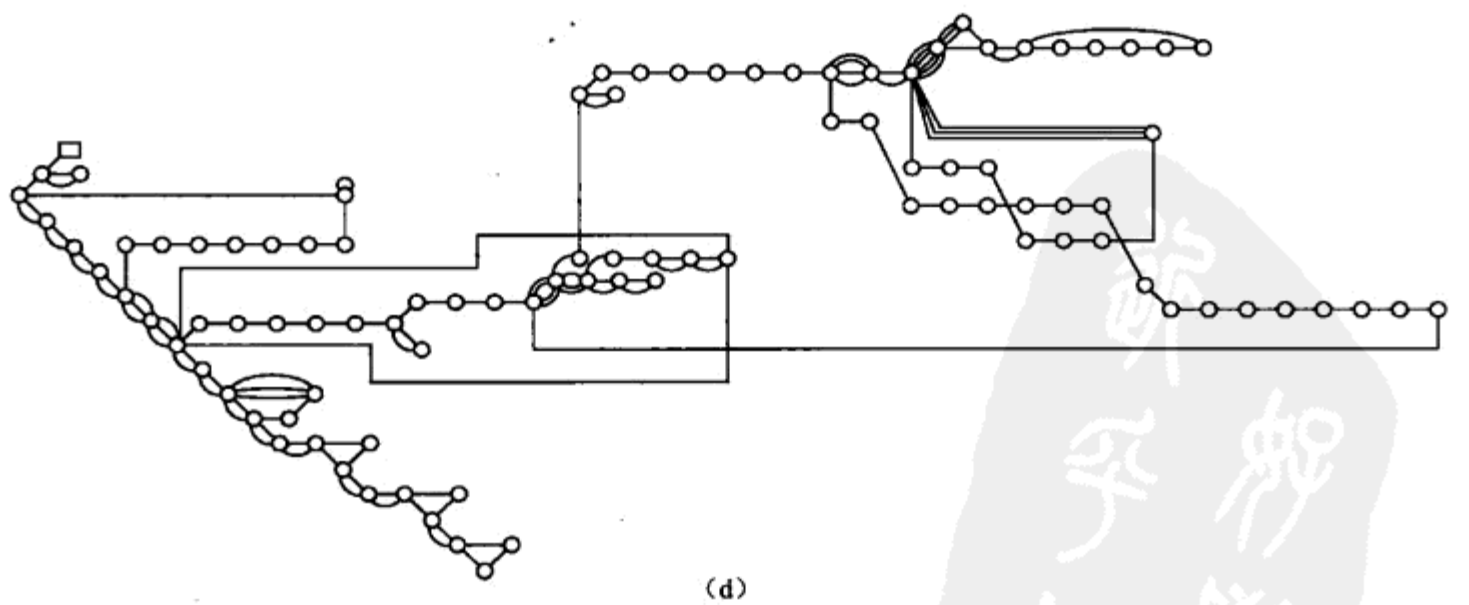


（a）



（b）

（c）



（d）

图 3.5.6　交互玩家图应该使用不同的方法（策略），这 4 个例子从相同的游戏场景和约束中获得，与图 3.5.5 的例子相似

我们标准地使用 K-medoids 分类[Friedman99]。K-medoids 是一个遍历算法，它基于一个给定的数据点集合以及中心由初始代表性物体近似而来的 k 个分类（我们在初始化中随机地给出我们的数据集成员），所有的成员被初始化为离它们最近的 k 分类描述（或 medoid）。然后，基于一个非分类描述是否改进了分类的总距离，每一步分类描述都被重新计算，然后成员被基于它们对新中心的距离重新分类。这个过程会继续下去，直到在两个连续的迭代之间没有区别（或没有改进）。应用一个分类标准函数来评价 k 分类的质量。由于 IPG 的高维数并且产生已有的成员，我们会在我们的分类中遍历所有可能的初始种子数值。由于数据的抽象维度，为了保证最好的分类发现与分类质量标准函数一致，我们同时也建议 k 使用从 2～n-2 中的值，其中 n 是 IPG 被比对的个数。

在分类 IPG 中，分类标准函数利用距离测量 $d(x_{ij}, x_{pq})$，它计算了在分类 i 中的第 j 个成员和分类 p 中的第 q 个成员之间的距离。距离 d 表现的是在两个成员间的图的修改距离。第 i 个分类的类内距离如下所示，其中 n_i 是在第 i 个分类中成员的数量。

$$\overline{d}^{i}_{INTRA} = \frac{\sum_{j=1}^{ni}\sum_{q=j+1}^{ni} d(x_{ij}, x_{pq})}{\frac{n_i(n_i-1)}{2}}$$

获得一个想要的数据分类同时也是一个优化分类标准函数的结果[Zhao02]。在我们分类 IPG 的经验中，我们发现下面的分类标准函数工作得最好[Youngblood02、Youngblood03]。

最小化，如下所示：

$$\sum_{i=1}^{k} \frac{\overline{d}^{i}_{INTRA}}{n_i}$$

在 IPG 上使用 K-medoids 应该产生一些基于所观察游戏中相似的战略和表现的分类。对人类和代理分类可以帮助决定代理是否表现得与已知的人类玩家或甚至其他的代理玩家很相像。分类可以被用来决定玩家技能等级和离下一个技能等级的距离。在游戏中被动态地评估的当前人类玩家的分类可以用来决定游戏 AI 的行为（基于观察到的玩家技能或战略）。虽然 K-medoids 可能对于游戏的实时处理来说不合适，但从游戏测试中创建玩家类型档案，并且使用比如 K 最近邻居（kNN）的简单快速方法来匹配和适当地响应，会十分有效率。

在图中挖掘

除了分类技术外，还有其他的方法可以用来从基于图的数据（比如，IPG 所表现的）中发现知识。基于图的数据挖掘领域提供像 Larry Holder 的 SUBDUE 之类的工具。SUBDUE 已经被用来为 UCT 数据寻找在 IPG 中的常见模式[Cook07]。使用压缩技术和最小描述长度规则，SUBDUE 可以找到 IPG 中的常见子结构。这些可以表现玩家的常见战略，而不用管它们实际的分类相似度。为预期行为而使用适当反应的游戏 AI，可以利用这些常见的子战略。

3.5.4　行为的更深理解

数据记录和玩家踪迹也可以用在其他目的上。玩家的观察方向可以和他们的位置一样有用。通过使用随着时间的玩家观察方向，你可以得出公认最大或最小观察的表面。我们已经开发了一个叫做 HIIVVE 的工具（Highly Interactive Information Value Visualization and Evaluation，高度交互信息数值可视化和评价），就是为这个目的所设计的[Dixit07]。在图 3.5.7 中展示的这个工具，使用玩家踪迹数据来计算交叉点，并为在世界几何体中的每个表面寻找信息数值。表面的信息数值表现玩家会看到在表面放置的信息的可能性。这个数据可以用来做出关于放置资源或交互功能点的设计决策。

PlayerViz 可以被用来跟踪被捕捉到的几乎任何类型的信息。比如，可以更好地理解 AI 代理的有用信息，在它们的智能机制中可以反映代理的内在状态变化（FSM 和 FuSM 代理）或归类于关卡触发的潜在的调试问题。代理行为决策也可以被明确地表达。

我们在夏洛特 UNC 的小组继续研究更好地理解游戏中的人类和代理智能。通过 CGUL 工具包（一个常见的用于游戏理解和学习的工具包），我们提供了一整套用来改善游戏 AI 的分析工具和方法。

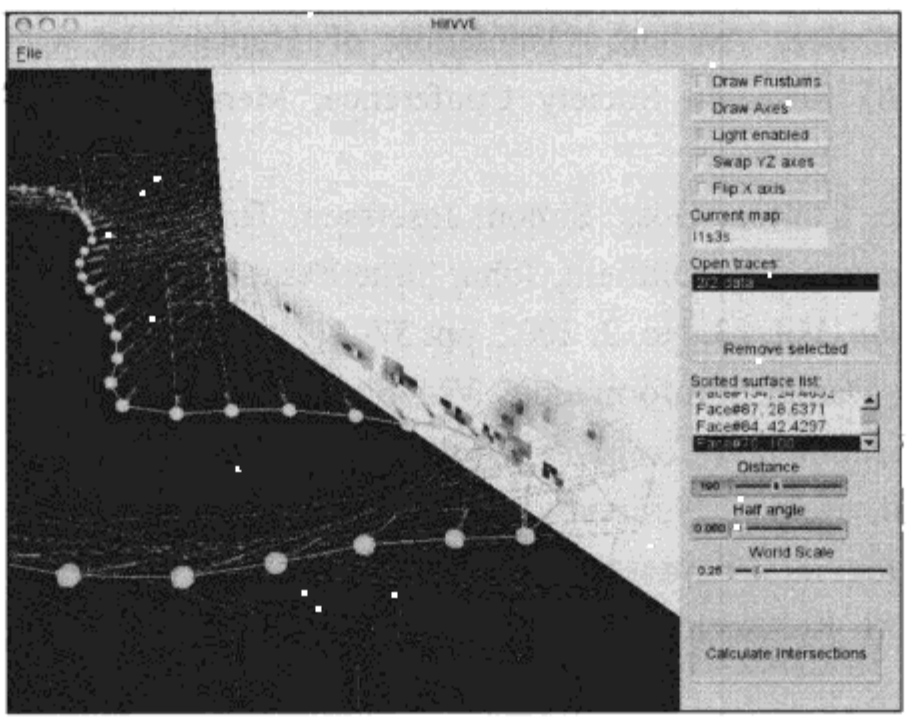


图 3.5.7　高度交互信息数值可视化和评价（HIIVVE）工具帮助决定游戏表面的信息数值。这是有用的游戏中的记录和从玩家测试数据分析中获得的信息的另一个示例

3.5.5　结论

有一些很有说服力并且能吸引人的原因使得在游戏中引入记录的能力。从记录人类玩家和 AI 在游戏环境中的交互中收集到的数据可以用像 PlayerViz 这样的工具执行可视化数据挖掘。为了游戏特定的分析和更好地理解游戏中的智能行为，PlayerViz 可以被用做自展（bootstrapping）过程来监督工具全部过程的开发。然而，玩家踪迹只完成了一半的工作。通过跟踪和建造从观察的玩家/代理交互的轨迹中生成的交互玩家图，并使用分类和知识发现技术来分析交互玩家图，开发者可以取得玩家表现和分类的新见解。接下来，这个信息可以被

用来开发更好的游戏 AI。

3.5.6 参考文献

[Consalvo06] Consalvo, Mia and Dutton, Nathan. "Game Analysis: Developing a Methodological Toolkit for the Qualitative Study Of Games," *The Interactive Journal of Computer Game Research,* Vol. 6, No. 1, December 2006.

[Cook07] Cook, Diane J., Holder, Lawrence B., and Youngblood, G. Michael. "Graph-Based Analysis of Human Transfer Learning Using a Game Testbed," *IEEE Transactions on Knowledge and Data Engineering,* 2007.

[Dixit07] Dixit, Priyesh and Youngblood, G. Michael. "Optimal Information Placement in 3D Interactive Environments," *Sandbox Symposium,* 2007.

[Eilers05] Eilers, Michael M. "Soapbox: Difficulty and the Interstitial Gamer."

[Friedman99] Friedman, Menahem and Kandel, Abraham. *Introduction to Pattern Recognition: Statistical, Structural, Neural, and Fuzzy Logic Approaches,* Imperial College Press, London, 1999.

[Gonzalez99] Gonzalez, Avelino. "Validation of Human Behavioral Models," Twelfth International Florida AI Research Society Conference, Menlo Park, AAAI Press, 1999, pp. 489–493.

[Heinze02] Heinze, Clinton, Goss, Simon, Josefsson, Torgny, Bennett, Kerry, Waugh, Sam, Lloyd, Ian, Murray, Graeme, and Oldfield, John. "Interchanging Agents and Humans in Military Simulation," *AI Magazine,* Vol. 23, No. 2, 2002, pp. 37–47.

[Keim02] Keim, Daniel. "Information Visualization and Visual Data Mining," *IEEE Transactions on Visualization and Computer Graphics,* Vol. 8, No. 1, (March 2002).

[Knauf 01] Knauf, Rainer, Philippow, Ilka, Gonzalez, Avelino, and Jantke, Klaus. "The Character of Human Behavioral Representation and Its Impact on the Validation Issue," Fourteenth International Florida AI Research Society Conference, Menlo Park, AAAI Press, 2001, pp. 635–639.

[Laird01] Laird, John E. and van Lent, Michael. "Human-Level AI's Killer Application: Interactive Computer Games," *AI Magazine,* Vol. 22, No. 2 (2001), pp. 15–25.

[Laird02] Laird, John. "Research in Human-Level AI Using Computer Games," Communications of the ACM, Vol. 45, No. 1, 2002, pp. 32–35.

[Marselas00] Marselas, Herb. "Profiling, Data Analysis, Scalability, and Magic Numbers, Part 1: Meeting the Minimum Requirements for Age of Empires II: The Age of Kings."

[Russell03] Russell, Stuart and Norvig, Peter. *Artificial Intelligence: A Modern Approach,* Prentice Hall, 2003.

[Youngblood02] Youngblood, G. Michael. "Agent-Based Simulated Cognitive Intelligence in a Real-Time First-Person Entertainment-Based Artificial Environment," Master's thesis, The University of Texas at Arlington, 2002.

[Youngblood03] Youngblood, G. Michael and Holder, Lawrence B. "Evaluating Human-Consistent Behavior in a Real-Time First-Person Entertainment-Based Artificial Environment," Proceedings of the Sixteenth International FLAIRS Conference, 2003, pp. 32–36.

[Youngblood06] Youngblood, G. Michael, Nolen, Billy, Ross, Michael, and Holder, Lawrence. "Building Test Beds for AI with the Q3 Mod Base," *Artificial Intelligence in Interactive Digital Entertainment* (AIIDE), June 2006.

[Zhao02] Zhao, Ying and Karypis, George. "Evaluation of Hierarchical Clustering Algorithms for Document Datasets," Proceedings of the 11th Conference of Information and Knowledge Management (CIKM), 2002, pp. 515–524.

3.6 面向目标的计划合并

Michael Dawe

使用面向目标的行为计划系统来创建和管理自动化代理的行为是一种强大的技术，并且迅速地在游戏开发者中得到认同。在游戏开发中，计划系统相对来说是新技术，但是学术界使用计划来解决问题已经有 50 年了。所以，找到一个可供游戏开发者使用来改进他们的计划系统的研究基础并不让人意外。

计划者可以得到改进的一种方法是通过使用计划合并，这种技术在学术界已经以很多方式使用，但是还没有应用到游戏中。使用计划合并可以允许自动化代理的行为范围变得更宽，甚至让它们同时尝试追求多个目标。本精粹将研究在实时游戏的背景下实现计划合并系统的一种方法，并讨论使用这个系统的含义。

3.6.1 回顾面向目标的计划系统

面向目标的行为计划系统是决策算法，它被设计用来让程序员摆脱对特定的代理行为的选择，而将这些选择置入代理自身的感觉—思考—行为循环。使用这个系统的最大好处是减少设计人工代理个人行为的复杂度，同时在代理的总行为中保持很高水准的真实感。

面向目标计划让特定的代理通过追求特定的目标来决定他们自身的行为。一个代理的目标可能包括破坏一个目标或者获得一个物品。在使用代理来记录世界状态的系统中，目标被表现为期望的世界状态。在传统的计划系统中，代理被限制只能在给定的地点及时地挑选一个最重要的目标。一旦这个目标被选中，一个代理可以将原子行为串联成一个序列来创建一个计划，有时也被叫做运算符。

比如，如果你的代理决定摧毁目标，它选择来完成这个目标的行为可能是攻击。行为有前提条件，这个前提条件描述了在行为执行之前必须是世界上的为真的条件；行为也有效果，效果描述了当行为完成时存在于世界上的必要条件。在攻击行为的例子中，一个前提条件可能是代理的武器已经上膛，一个效果将会是目标的毁灭。

使用效果和前提作为指导，任何启发式的搜索都可以通过列出一个代理可用来达到期望目标的行为序列来创造计划。Jeff Orkin 描述了如何为计划目的而使用 A*算法[Orkin04]。完成后的计划就是代理用来实现目标的一系列行为。

在讨论计划合并之前，还需要一些最终的术语。完全有序计划：在计划中每个行为的顺序完全地被指定，比如，一个特定的行为第一个发生，另一个第二个发生，等等。部分有序计划：可能会指定单独的行为顺序，但是尽可能不理会所有行为的准确顺序。也就是说，一个部分有序计划不会指定行为的顺序，除非一个行为满足了另一个行为的前提条件。完全有序计划可以通过给予计划中行为的特定顺序而从部分有序计划中生成。

图 3.6.1 展示了制作三明治的部分有序计划和完全有序计划的例子。在部分有序计划的版本中，注意独立的行为（获得肉、奶酪和面包）相互之间没有顺序关系。但是行为可以有相对的顺序；所有的组成部分必须在制作三明治前获得。通常来说，给予行为的唯一顺序是行为的前提条件所要求的。另外，完全有序计划加强了所有行为的特定顺序，而不管单独的行为是否满足其他行为的前提条件。虽然可以为三明治以任何顺序获得肉、奶酪和面包，但是一个完全有序计划指定了执行这些行为的顺序。显而易见，任何部分有序计划都可以被表现为一个完全有序计划。

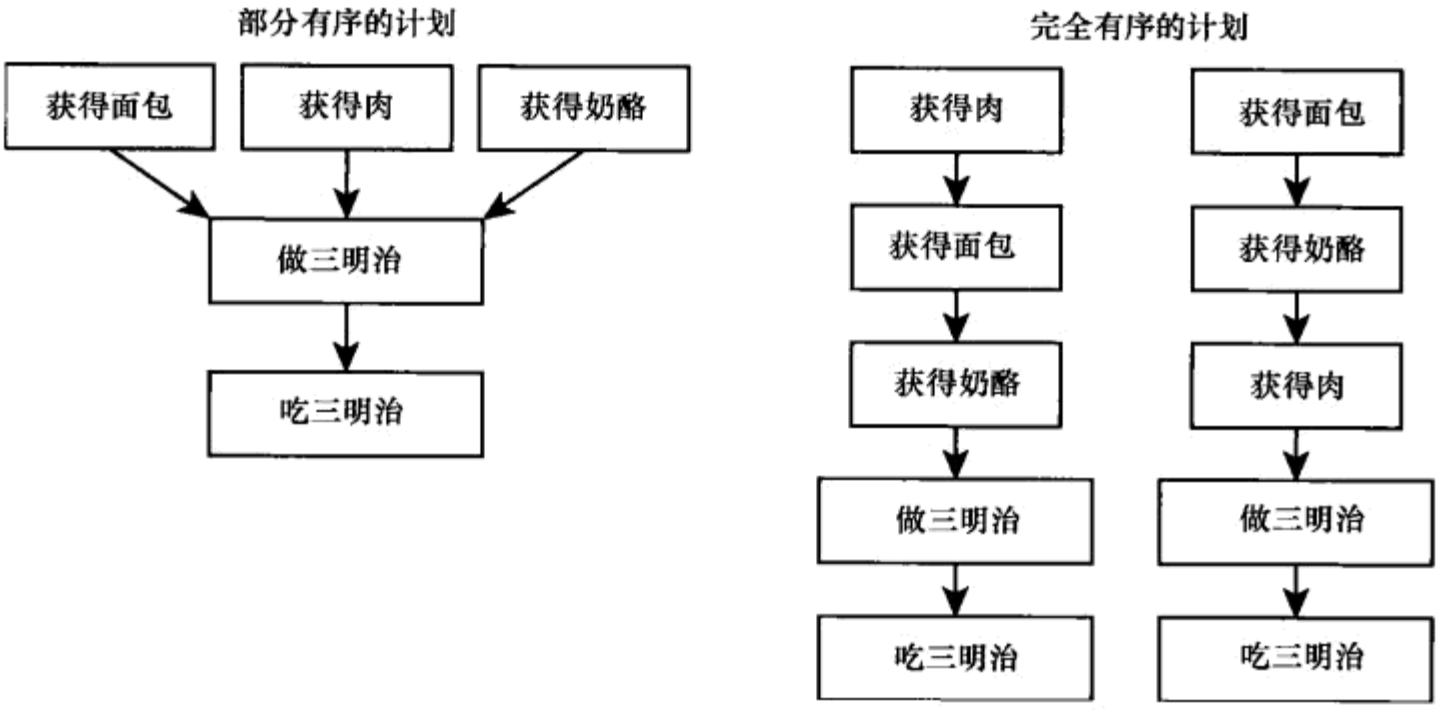


图 3.6.1　部分和完全有序计划。并没有给出部分有序计划的所有完全有序实例

虽然绝大多数基于学术的计划算法产生部分有序计划，但是这种类型的计划者还没有在游戏中找到广泛应用。有一些原因解释了为什么完全有序计划能更直接地应用到 NPC 上。

- 首先，给予一个部分有序的计划，为了执行计划的行为，一个代理在某时不得不明确地或含糊地定义一个完全有序计划。也就是说，代理仍然需要在任何数量的未排序的行为中选择一个行为来第一个执行。在这个基础上，有些原因解释了为什么在其他行为前执行一个行为可能是有利的，但是代理选择第一个行为的原因可以很容易地被抽象到计划者自身。
- 游戏传统地处理完全有序计划的第二个主要原因是用 A*来创造计划的方便性。因为 A*是一个众所周知并且多用途的算法，对于面向目标的计划系统是一个好的选择，而且 A*本质上产生完全有序计划。

[Orkin04a]、[O4kin04b]和[Orkin06]覆盖了许多用于游戏中实现 A*计划系统的实践细节。

3.6.2　用于面向目标计划的计划合并

计划合并适用于采取一些独立生成的计划并从中生成一个单独计划的过程，通常伴随着

减少计划的总体开销的目的。减少了开销的计划往往也有着产生更合理行为的益处。为了证明计划合并的力量，让我们在进入算法细节前看一个例子。

假设一个代理有一个任务，从世界中收集所有道具并把它们带回到家中。如果一个代理一次只能携带一个道具，很显然，最好的选择就是走向一个道具，收集它，然后回家。但是，如果一个代理可以携带多个道具，显然存在很多情况，代理可以通过同时收集多个道具来减少它旅行的总路程。

利用一个计划系统，你可以有很多方式来完成这个行为。假设你收集道具并返回家的目标叫做道具返回目标。你可以写一个收集道具行为来完成这个目标。一个执行收集道具行为的代理会寻找最近的道具，尽可能多地收集，然后带着它们返回家。虽然这会是一种解决方案，但很明显，收集道具行为会非常复杂。它需要包括寻路和在道具间移动，拾起道具，寻路和回家，以及到达之后放下道具的代码。为了使在一个行为中增加的功能起作用，会挫败拥有一个灵活的计划系统的目的。

写一些更小、可重用的原子行为是更容易的，比如为了寻路的 GoTo，从世界中收集道具的 GetItem 和把道具放在家中的 ReturnItem。这些多样行为允许计划者以正确的顺序将它们串联起来，以完成复杂的工作，并进一步允许你在很多类型的 NPC 中重用行为。但是没有一个行为可以与代理交流它每次应该试着收集多个道具。相反，你可以通过计划合并来完成期望的行为。

普遍会选取两个有重叠行为的计划，把它们合并为一个比单独执行每个原本计划开销要低的单一计划。在这个例子中，代理可以计划独自收集每个道具，产生两个不相关但十分相似的计划，如图 3.6.2（a）所示。合并这样的两个计划的可能结果是尽可能合并更多的行为，产生单独的计划，如图 3.6.2（b）所示。当代理执行这个计划时，它会在回家前收集两个道具。

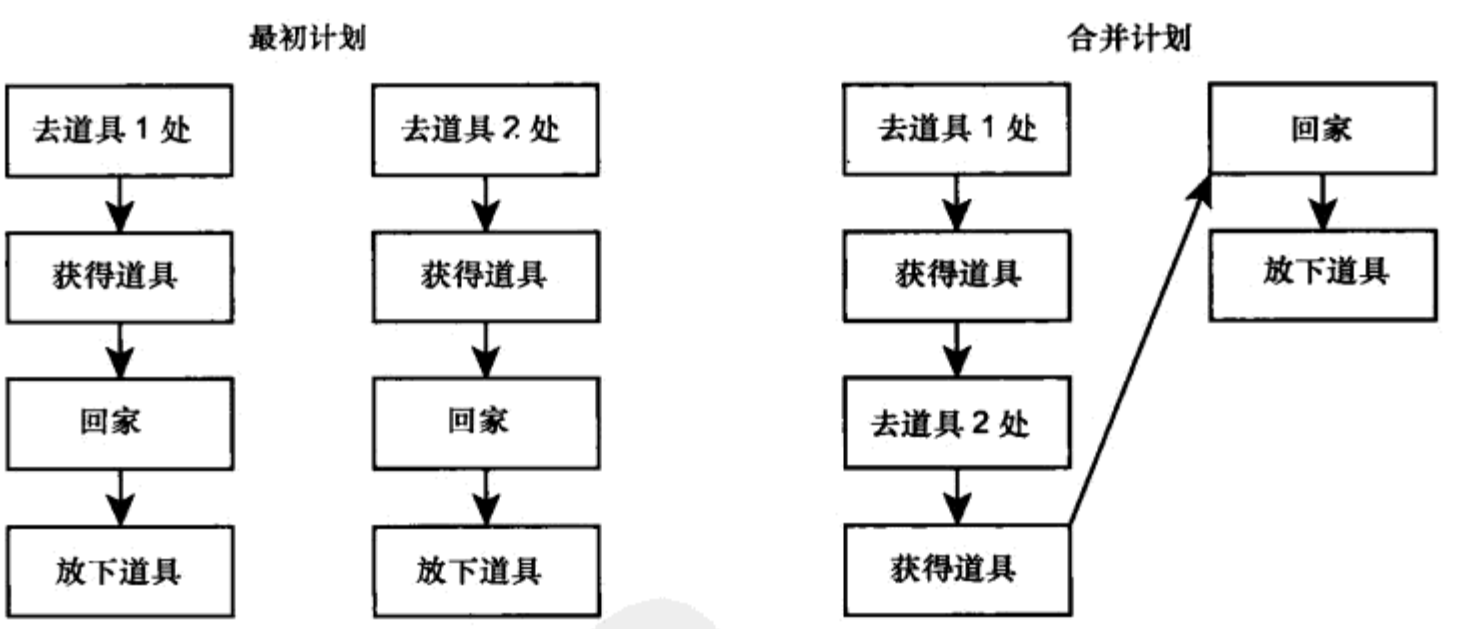


图 3.6.2 两个完全有序计划和它们可能合并的结果

实现一个计划合并算法

学术上，计划合并的主要目的是优化计划。[Foulser92]指出了优化一个计划的两个主要组成部分：找到可以被合并的行为；如果存在多于一种合并操作的方法，则算出最优的方法来合并行为。分开处理这些问题会更容易，所以我们在本精粹中采用这种方法。

寻找可合并行为的第一个挑战在于准确地发现什么类型的行为可以被合并。但是简单来说，如果有另一个行为可以用下面的结果来代替被合并的行为，任何行为都可以被合并。

- 如果行为有相同的有用的效果。
- 如果用来代替的行为的开销小于被合并行为的开销总和。

如果效果直接为计划中的另一个行为建立了前提条件或者是目标本身的前提，那它们就是有用的。比如，假设一个代理计划用开火和上膛行为来摧毁目标。上膛行为有很多效果，首先，它让武器有子弹，其次，它减少了代理的弹药量。第一个效果是有用的，因为它完成了计划中另一个行为的前提条件。第二个效果没有用，因为它与计划的执行无关。

如果没有行为本身的信息，搜索可合并行为计划的开销就会很大，所以最好特别地寻找那些已知可以合并的行为。在一个已实现的系统中，这意味着要么寻找自己可合并的特定行为，要么寻找已知的可以合并的行为组合。在早些的资源收集例子中，你知道代理会有多种计划，每一个都有 ReturnItems 行为。这就是一个要寻找的完美的候选行为，因为你知道可以合并两个 ReturnItems 行为。在这个特定的例子中，你甚至可以从计划的末尾开始寻找，因为很可能是每一个计划中的最后行为可以被合并。GoTo(Base)也可以与自身合并，因为它显然可以完成同样的效果。

第二个挑战是一旦发现了可能的合并后，要创造一个最优的计划。[Foulser92]处理了创建最优计划的困难，要注意创建一个最优化的计划会迅速变得开销巨大，并且可能对于游戏来说过火了。对于收集资源的 NPC 来说，你已经通过允许代理一次收集多个资源来改进行为。与其花时间来担心计划是否最优，不如只是把剩下的两个计划合并到一起，如图 3.6.2 所示。

然而，为了让代理看上去更加智能，你可以加入评价以及对特殊目的的检查来帮助为余下的行为排序。比如，你知道你有两对 GoTo（道具），且 GetItem 的行为要在合并的行为前被代替，所以你可以写一个评价来保证代理先去最近的道具处。评价是为了在计划合并中加强所期望的行为而制定的一般规则。

最简单的是，接下来合并计划算法接受由通用 A*计划系统生成的两个计划。比如，代理可以把它最重要的两个目标发给计划者，然后把那两个独立的计划发送给计划合并者。对于第一个计划的每一个行为，算法检查它是否可以被第二个计划的一个行为合并，如果可以执行合并，那两个行为被放到一个单独的计划中，从两个计划中放置合并行为前的行为要小心，对于在合并行为后的行为也是一样。如果需要对未合并行为的顺序进行更精确的控制，可以加入评价来决定最好的顺序，并且根据需要重新安排行为的顺序。对于更广范围的可能合并来说，一个完整的计划合并算法应该在每一个计划中检查每一个可能的行为组的综合效果，寻找一连串的行为可以被一个单独的、更廉价的行为所代替的情况。这样一个算法对可合并计划产生非常可观的改进，但是运行起来开销同样巨大。

超越单代理合并

虽然为一个单独的代理合并两个计划确实可以改进行为，但计划合并同样在分组行为区域中可以提供出色的益处。比如，一个利用计划合并的代理可以合并一个单独的目标（拾起武器或增加生命值）和分组目标（提供火力掩护）。在这些情况下，利用计划合并可以允许一个代理在分组顺序下维持它自己的目标和个性，甚至允许代理同时完成很多目标的情况。

提高行为搜索的战略

搜索带有相似效果的两个或多个计划行为是昂贵的，特别是如果你考虑用不同的综合效果代

替行为组。如果游戏是快节奏的（比如，很多 FPS），代理的第一个和第二个目标会变换得更快，甚至可以为它的第二个目标设计计划。确实，如果你不能快速执行合并，计划合并是没有用的。

一个可能的减少搜索行为时间的战略是只在计划存在特定行为时才搜索可合并的行为，这可以在计划决策过程中做出决定。对于非常长的计划来说，计划结构本身会带有直接与可能合并的行为的联系，不仅要指示算法立刻进入正确的位置，并且也会通知它是否值得搜索一个合并。在特定类型的代理中，在每一个计划中只寻找特定行为来合并也是值得的。

相似地，你也许只在计划目标相容的情况下才尝试合并。相反地，如果两个计划之内的目标不相容，也就没有意义去费力地尝试合并。确实，如果目标不相容，即使为第二个目标做计划也是浪费时间。这个决定可能最好由程序员做出。也许对你来说很显然，攻击和撤退目标永远不会产生可合并的计划，但是算法会在报告没有可合并行为存在之前搜索每一个计划的每一个行为。

3.6.3 结论

计划合并提供方法来改进代理在单独或组行为时的可感知智能。虽然可能是一个开销十分昂贵的过程，但是只要仔细地考虑，可以通过花费一小部分额外的时间检查生成的计划来完成它。

应该注意，这只是执行计划合并的一种方法。[Thangarajah02]和[Thangarajah03]介绍了不同的执行计划合并的系统和方法，也许更适合于比这里描述的代理行为更长的代理。比如，在[Thangarajah03]中描述的计划合并算法特别适合于策略游戏 AI 对手，可以用多种不同的方法来完成目标，并且可能延迟行为来利用明确的合并机会。

计划是一个多功能的 AI 系统，有很多的机会来进行扩展和改进。即使如果在给定情况下计划合并没有用，但是它所提出的想法可以应用到其他 AI 系统中，或者甚至是其他的像分级任务网络（HTNs）那样的计划系统中。这种技术提供的行为的改进使得代理有更好的智能，同时让玩家可以有更好的游戏体验。

3.6.4 参考文献

[Foulser92] Fousler, David, Li, Ming, and Yang, Qiang. “Theory and Algorithms for Plan Merging.” *Artificial Intelligence*, 57(2–3): pp. 143–181, 1992.

[Orkin04a] Orkin, Jeff. “Applying Goal-Oriented Action Planning to Games,” *AI Game Programming Wisdom 2*, Charles River Media, 2004.

[Orkin04b] Orkin, Jeff. “Symbolic Representation of Game World State: Toward Real-Time Planning in Games.” *AAAI Challenges in Game AI Workshop Technical Report*, 2004.

[Orkin06] Orkin, Jeff. “Three States and a Plan: The A.I. of F.E.A.R.,” Proceedings from Game Developers Conference, 2006.

[Thangarajah02] Thangarajah, John, Winikoff, Michael, Padgham, Lin, and Fischer, Klaus. “Avoiding Resource Conflicts in Intelligent Agents,” Proceedings of the 15th European Conference on Artificial Intelligence 2002 (ECAI 2002).

[Thangarajah03] Thangarajah, John, Padgham, Lin, and Winikoff, Michael. “Detecting & Exploiting Positive Goal Interaction in Intelligent Agents,” *AAMAS* ’03, July 14–18, 2003.

3.7　超越 A*：IDA*和边缘搜索

Robert Kirk DeLisle

图搜索技术在游戏编程中无处不在。无论是什么游戏类型，图搜索方法不可避免地成为游戏 AI 的基础。现在领先的 3D FPS 类型的游戏非常依赖于寻路方法，可以让 AI 玩家在环境中为了自我防御或者进攻行为而进行移动。这同样可以扩展到以迷宫或地形遍历为主体游戏玩法的 2D（或者 2.5D）游戏中。此外，像跳棋、象棋、黑白棋甚至井字游戏这样的游戏都有一些程度的游戏树或状态图的评价，以开发出令人信服和有竞争力的人工智能。

在寻路领域，问题标准地呈现在树的形式上，开始点会被考虑为树根（见图 3.7.1）。

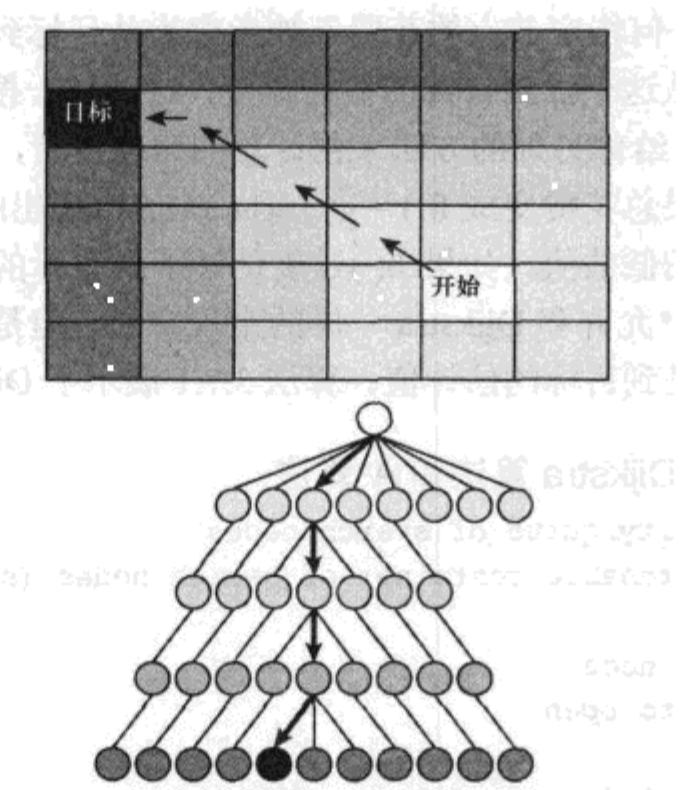


图 3.7.1　一个搜索网格和它相关的搜索树，对应的网格和树节点被相似地上色。网格和搜索树中都展示了带箭头的路径。导致对相同节点的多余访问的路径已被移除

根节点可以被扩展为一些子节点，表现在搜索中所有下一个可能的步骤。标准的 2D 寻路过程可以很清楚地表达为用子节点表示每个可移动的方向。比如，如果 4 个主要的方向是允许的，根节点扩展为 4 个子节点，每一个代表一个方向：北、南、东和西。对角线运动增加为 8 个子节点，表示多出的 4 个方向：西北、东北、西南和东南。随着在搜索目标中扩展路径，子节点可以被进一步扩展。这种类型的问题规划也可以被应用到为乱序魔方搜索最短的可能解决方案之类的问题。最初乱序状态的魔方是

根节点，每一个可能的魔方面的旋转都对应于根节点的子节点。通过这种方式规划问题，比如，在有开始和目标状态的图中进行图遍历，就打开了通往一系列算法的大门。

3.7.1 A*和 Dijkstra

在游戏 AI 中，A*已经成为最常见的寻路搜索算法。A*的历史从广度优先搜索开始，在广度优先中，根节点被扩展，评估它的所有子节点，然后再移动到下一级树深度。如果在当前深度下没有找到目标，下一级的子节点就会被扩展和评估。Dijkstra 修改了这个算法，通过添加一个 open list 和 closed list 来提供两个重要的功能。

- 每一个节点记录到这点的路径的花费，openlist 中的花费可以通过最佳优先的策略被排序。当从一个节点转移到另一个节点的花费不同时，这种方法特别有用，就像进行从沼泽还是从干燥地面通过的选择一样。到目前，允许最佳路径的扩展可以偏离搜索花费比较大的路径。
- 同时，open list 和 closed list 表现为以往被评价过的节点的目录，因此可以阻止已经访问过的节点的重新扩展。这些可以相当地改进广度优先搜索；但是通过合并启发式搜索战略可以有更好的改进。

到这一步，任何特定节点的花费都被考虑为从目标到这个点的花费，通常会被参考为 g( )。如果你包含一个从这个点到目标的估计花费，这种统一搜索会被显著改进。这个启发计算通常会被参考为 h( )，给你另外的方法来估计总的路径花费，并且再次显著地使搜索偏向目标。任何特定节点的结果总花费变成 f( ) = g( ) + h( )，值得提出的是 h( )应该总是可采纳的，或者是一个从节点到目标的低估值。如果 h( )过度估计了到目标的花费，搜索可能多产的路径会被延迟或者完全丢失。A*允许和 Dijkstra 一样的常规算法，但是现在任何特定搜索节点的花费包含了启发花费，也就是到目标的估计值。算法 3.7.1 展示了 Dijkstra 算法和 A*算法的比较。

算法 3.7.1 Dijkstra 算法和 A*算法

```
open - priority queue of search nodes
closed - searchable container of search nodes (such as an associative array)

root = start node
push root onto open

while goal not found and open not empty
    sort open by f() of each search node
    remove top of open and set to current node
    if current node = goal
        stop
    else
        push current node onto closed

    for each child of current node
        if child present in closed
            continue
        else
```

```
set child's f() = g() + h()(for Dijkstra's, h() = 0)
push child onto open
```

不需要惊讶 A*的最基本的弱点在于 open list 和 closed list 的维护。open list 必须被保持为已排序的，列表顶部是最低花费的节点。open list 和 closed list 会被不断轮询来决定一个节点是否已经被评价，这会带来很高的计算负担。虽然很多 A*算法的优化方法已经被开发出来，但是如果搜索空间非常大，与 open list 和 closted list 相关的总花费还是会带来性能的损失，甚至会导致功能的完全缺失。虽然简单的 2D 寻路问题不会严重得遭受这些与不同方法对应的缺点，但是在复杂 3D 环境中的寻路会很容易妨碍 A*算法的能力。

其他的问题，比如对魔方搜索最短的解决方案，搜索空间太大，以至于 A*算法在几分钟内迅速地超出了计算机内存的上限。比如 3×3×3 的魔方，对于任何特定的搜索节点都有 18 个子节点。即使你限制下一次移动不会包括某些特定的操作（比如，你不应该允许同一边旋转两次），你也只能把下一级的子节点的数目减到大致 13 左右。在 8 个回合后，这会带来超过 10 亿种可能的状态。在这种复杂的搜索树中，必须用其他的方法来简单地启用一种解决方案的辨识。

3.7.2 迭代延伸 A*(IDA*)

A*的一种扩展是迭代延伸 A*(IDA*)，如算法 3.7.2 所示。在它最基本的形式中，这个算法去掉了 open list 和 closed list。这确实增加了状态重复评价的风险，但是这可以通过适当构建节点扩展的方式来适应（特定的顺序、防止回溯等）。

算法 3.7.2 迭代延伸 A*(IDA*)

```
root = start node
threshold = root's g()

perform a depth-first search starting at root

if goal not found,
    set threshold = minimum g() found that is higher than current
    threshold
    repeat depth-first search starting at root

depth-first search(node):
    if node = goal
        return goal found

    if node's f() > threshold
        return goal not found
    else
        for each child of node, while goal not found, depth-first
        search(child)
```

它也可以是自适应的，因为节点早些扩展会比晚扩展得到更低的 g( )值，并且不管何时 h( )的值应该总是一样的。在 IDA*中，为 f( )建立了一个花费阈值，定义了节点可以被评估的最大

允许花费。所有的节点都在这个阈值下被扩展，如果目标节点没有被找到，阈值会增加。因为你没有维护历史，你必须从起始点重新初始化搜索，并用给予的阈值扩展所有允许的节点。也许看上去重复所有以前的非目标节点评价是违反直觉的，但是扩展和评价节点的花费远远低于维护 open list 和 closed list。另外，在边缘的节点，那些以前不会被探索到的在搜索边缘的节点，在数量上会总比在阈值下的已扩展的节点数目要多。这个事实就意味着我们有效地把以前节点的重复探索的花费减少到比扩展新边缘节点要小的花费，最终的结果是以搜索需要的时间花费为代价，而获得内存上的最小开销。

3.7.3 边缘搜索算法

在 A*和 IDA*之间有一种叫做边缘搜索的算法（算法 3.7.3），节点会像 IDA*一样给予一个花费阈值来扩展，但是这里的边缘节点不会丢失。相反，边缘节点在 now 和 later 列表中被维护。

算法 3.7.3 边缘搜索

```
now - linked list of search nodes, list order determines order of evaluation
later - linked list of search nodes
root = start node
threshold = root's g()
push root into now

while now not empty
   for each node in now
      if node = goal
         stop

      if node's f() > threshold
         push node onto end of later
      else
         insert children of node into now behind node

      remove node from now and discard

   push later onto now, clear later
   set threshold = minimum g() found that is higher than current threshold
```

在 now 列表中顶部的节点被评估，如果它的 f( )值比阈值大，它会被移动到 later 列表中。如果 f( )的数值比阈值低，节点的子节点被扩展，并且现在的节点被丢弃。新扩展的子节点被添加到 now 列表的顶部，并准备好下一次评估。

这个过程以比较弱的排序来维持列表，并且以像 IDA*的深度优先的方式来有效地扩展节点。如果一次完成遍历（一次迭代）now 列表后还没有找到目标，阈值就会像在 IDA*中一样增加。later 列表会转变成 now 列表，搜索从 now 列表的顶部重新开始。虽然边缘搜索过程确实需要 now 和 later 列表的维护，却没有排序的开销。此外，这个额外的内存花费比 A*算法要低，因为不需要存储所有以前评价过的节点。边缘搜索和 IDA*一样，不会在迭代间的重复搜索中损失太多速度。

在 Bjornsson、Enaenberger、Holte 和 Schaeffer [Bjornsson05]的研究中，这些算法与 Baldur's Gate II 中提取的游戏地图来进行比较，结果发现边缘搜索的搜索时间减少到 A*的 25%～40%和 IDA*的 10%。即使 IDA*被优化为适应对搜索树中节点的重复访问，这个搜索时间的改进也会被保持。总的来说，速度的提升归功于不需要维护有序的 open list。因为在 now/later 列表中需要维护一些程度的搜索历史，所以边缘搜索的性能花费显然提高了内存的使用。用这种方式，边缘搜索看上去是 A*和 IDA*算法的有用的中间体。

3.7.4　结论

我们不缺少算法来进行图搜索和寻路。虽然 A*算法展示了最广泛的应用，但对于为单独情况而进行的 A*算法的特别化和优化极大地扩展了这一系统。算法选择的驱动力过去是、将来也会是内存和时间限制，并且几乎在每个实例中都必须牺牲一个而换取另一个。IDA*和边缘搜索表现了对 A*系列算法有用的修改，并且可以最终优于传统寻路方法。

3.7.5　参考文献

[Bjornsson05] Bjornsson, Yngvi, Enzenberger, Markus, Holte, Robert, and Schaeffer, Jonathan. "Fringe Search: Beating A* at Pathfinding on Game Maps," IEEE Symposium on Computational Intelligence and Games (2005), pp. 125–132.

第 4 章

音　频

简　介

Alexander Brandon

游戏音频编程正在变得比以往更复杂。随着游戏音频越来越接近电影这样的后期制作产品，大量的相关因素就进入了游戏。应该将和摄像机关联的点音源放在什么位置呢？要循环播放吗？这些声音会不会有延迟？或者是有先后顺序的关联吗？它们是怎么被触发的？如何对它们进行分类和混合？本章的作者们已经提供了一些让人印象非常深刻的、新鲜的技巧来应付这些新事务，通过这些手段可以让你的游戏用高音频质量来保证竞争力。

来自索尼计算机娱乐公司欧洲分部的 Jason Page 为我们展现了革命性的单元处理技术，这一技术运用于 Playstation 3 以及 Jason 和他的团队开发的多流体工具中。Robert Sparks 给多组混合且有多层次的声音播放提供了一个充满活力且优雅的解决方案。特别是在不同平台上使用不同硬件配置来播放时，这个特别的法宝就像是天赐之物一样。在 Halo 3 中已经使用了音效的专业工作室标准（音波）。看看 Mark France 的文章就可以得到更多的消息，来让你知道这种实时效果的功能实现。Ken Noland 还提供了更深层的技巧来优化这种效果。最后，Stephan Schütze 费尽心思地为大家提供了一套避免音频重复播放的方法，他的点子对于目前的游戏开发来讲简直是太先进了。

所有这些作者都是在业界广受尊敬的专家，他们的点子有可能让你的下一个游戏获得最佳音效奖。希望你喜欢这些精粹。

4.1 基于可编程图形硬件的音频信号处理

Mark France
mark@raccoongames.com

现代化的实时声音处理有时候会产生高强度的计算，因为许多算法经常会被同时处理。可编程的数字信号处理器对开发者来说还算正常，但对普通消费者来说就实在太贵了。再者，新式的声卡还只是一个固定功能的实现，它发展的步伐较慢，而且经常会限制音频程序员施展拳脚。本精粹提供的技术能够让你把原本放在 CPU 上处理的例行程序转嫁到 GPU 上，从而在 GPU 相对巨大的 SIMD（单指令多数据）并行流处理能力上获益（见图 4.1.1）。灵活性提高了，就可以允许你建立自定义的、高质量的回声模型。这个回声模型可以根据场景的几何数据实时计算出来，比依靠在上个年代的硬件中预设参数要好得多。

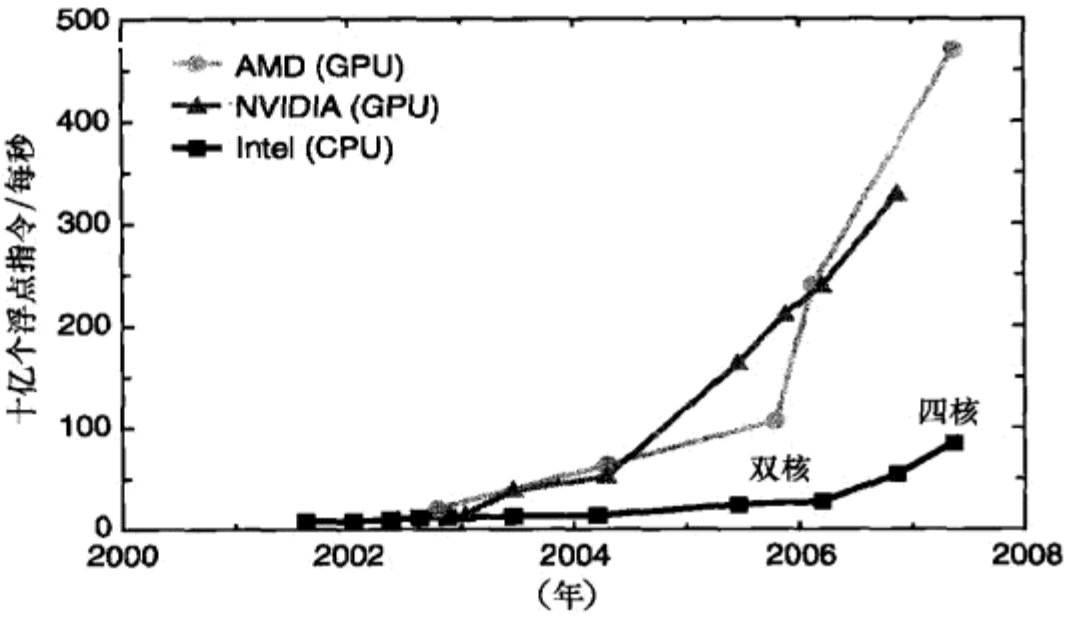


图 4.1.1 GPU 和 CPU 计算能力的比较[Owens07]

4.1.1 GPGPU 编程概述

GPU 向可编程管线的转移及其不断提高的可编程能力，允许它被用做一个强大的通用协处理器。如图 4.1.2 所示的管线能够被一般的应用程序编程所用，而并非只适用于特定的图形程序，这个就被称为 GPGPU（通用图形处理器）编程。这种方法已经被成功地运用到从人工神经网络[Rolfes04]到布料物理模拟[Zeller05]的应用程序上了。

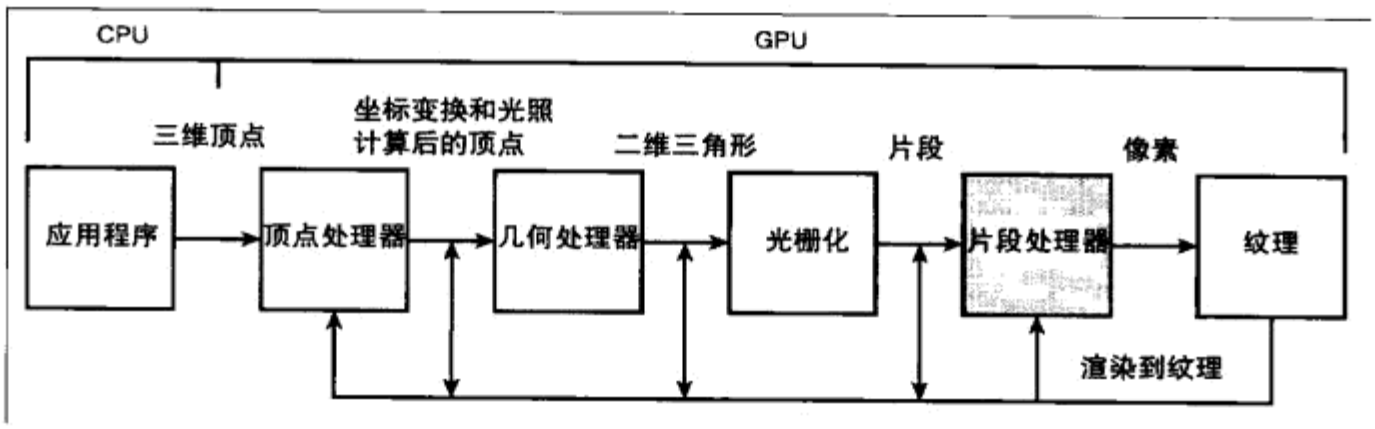


图 4.1.2　目前的图形管线

对于 GPGPU 而言，片段着色器会更有用，这是因为片段管线比顶点管线多，还有就是片段处理器处于管线末端，它允许直接输出。着色程序可以用汇编语言或者高级着色语言来写，例如 Cg、HLSL 和 GLSL。基于这个目的，我比较倾向用 GPU 的 Brook 语言来写。该语言是为处理流而特别设计的，且通过生成具有 C++运行时的 Cg 代码，而直接在 GPU 上运行。关于 GPGPU 编程的更多信息可以在[GPGPU07]中找到。

GPU 音频优化

诸如多处理单元或者乘法累加指令等 GPU 特性，和那些专业的音频数字信号处理器硬件[Gallo04]非常类似，因此无所不在的 GPU 就可以成为一个有效的数字信号处理器的替代品。

GPU 操作的是包含 4 个浮点数的向量，其通常重现为 RGBA 分量。因此，音频采样数据通常被保存在某个分量中，并且在提交给 GPU 处理之前，会将一维采样数组映射到二维矩形纹理中。

使用 GPU 来处理音频会明显地提升性能吗？Whalen[Whalen05]通过以下方式来回答这个问题：分别在图形硬件和 CPU 上使用着色语言来处理一组数字信号效果，其目的就是要指出哪个更快。实验发现：在 GPU 上处理合声算法和压缩算法时，执行的时间明显减少了。类似于滤波器和延迟效果的其他算法会有一点点慢。在处理与 GPU 的流处理模型（即许多处理器并行处理同样的代码）相适应的任务时，GPU 才会有优势。因此，并不是所有的音频编程技术都可以通过在 GPU 上处理而得到优化。

4.1.2　音频效果

这个部分关注的是描述合声以及压缩音频处理效果的算法。合声效果是给音频信号加上了一个很短的延迟以及轻度的音调调高，目的就是增加一种听得见的“层次”效果。合声效果可以帮游戏建立一种超现实主义的“梦幻”效果。该效果的处理需要两个纹理查找过程，它们中间的插值如下：

```
lookahead(coord, index)
{
    coord.x = coord.x + index * step;
    if(coord.x > 1.0)
    {
        rowsUp = floor(coord.x / rowSize);
```

```
        coord.x = coord.x - rowsUp * (1 + step);
        coord.y = coord.y + rowsUp * step;
    }
    return coord;
}
chorus(coord, texture)
{
    s1 = lookUp(texture, coord);
      s2 = lookUp(texture, lookahead(coord, 20 * sin(coord.x)));
      return interpolate(s1, s2, 0.5);
}
```

和数据压缩无关的音频压缩效果降低了音频信号的动态范围，并对平衡游戏的整个音频混合是有帮助的。该效果需要一次纹理查找过程，计算压缩的算法执行如下：

```
compress(coord, texture)
{
    s1 = lookUp(texture, coord);
    return pow(abs(s1), 1 - level / 10);
}
```

许多其他音频效果，例如，延迟和标准化处理，可以用类似的方法来优化。

4.1.3 室内效果

可以使用 GPU 来得到更好效果的另外一种音频处理技术就是室内效果，它已经通过[Jedrzejewski06]验证。

从环境几何体来实时计算共鸣、遮挡物和闭塞需要很大的计算量。射线跟踪方法是实现该效果的一种方法，而且该方法又很适合在 GPU 中执行。室内效果的射线跟踪和图形学中的射线跟踪是不一样的，因为这里面被计算的场景并不需要精确的视觉表示，并且通常使用更小的渲染目标。射线被跟踪的路线只是从声音源到听众的位置而已。

预计算

场景几何体包含用来描述墙的多边形；其他的大到足够影响声音环境的游戏物体其实可以被近似地看成盒子。在这个算法的预计算阶段，几何体可以被分割成二叉树（BSP）。在这个过程中，实心的突出区域作为二叉树的叶子。计算好的二叉树用来建立一个入口图（portal graph），这个入口图反映了叶子间的通路。如果一个入口和多边形处于某个叶子的同一平面上，那么叶子必须再被分割一次。如果需要再将叶子分割一次，就需要计算新的入口和通路了。入口和平面的信息（包括标识此信息是入口还是平面，及其吸收值）被保存在分离的一维纹理中。叶子信息包括了平面纹理的索引以及它包含的平面数量。每当场景几何体发生了变化，都会执行这个步骤。

实时渲染

片段着色器是这样执行的：首先计算每条射线与当前叶子的交点，然后传播到新的叶子

处，接着被反射的射线与监听对象相交。监听对象的位置可以被近似地看成一个包围球面。如果用监听对象来代表游戏玩家，那么我们会经常地用到玩家角色的包围体积。着色程序的伪代码如下：

```
LeafPlaneIntersect(Ray)
{
    为当前射线获取平面索引
for (i=1; i<=6; i++)
    {
        对于当前叶子计算和平面的交点
        为最邻近的相交射线存储数据
}
}
PropagateRay(Ray)
{
    检查 currentLeaf 是否包含更多的平面

    if(currentLeaf == listener.leaf)
        计算射线和 boundingsphere 的交点
    if(intersection with plane)
        计算反射射线及其吸收值
    If(intersection with portal)
        为射线设置新的叶子
}
```

然后就可以创建环境混响模型了。该过程包括渲染目标纹理和最后射线数据的获取。这里使用了 3 个渲染目标纹理：一个是保存状态信息，另一个是射线源，还有一个是射线方向。

4.1.4　结论

并不是所有的音频算法都能够从 GPU 的并行计算那里得到好处的。然而，在图形硬件上执行某些音频效果算法和听觉射线跟踪任务，确实能够达到加速的效果。除了在本精粹中介绍的音频技术以外，GPU 也在某些地方显示出了超越 CPU 执行力的优势，例如，在音频处理中无处不在的 FFT（快速傅里叶变换）。当基于 PCI-Express 的显卡变得越来越普遍时，把大量的数据从显卡内存传输到系统就不再是一个明显的瓶颈了。这些技术展示了 GPU 可以用来优化许多音频算法，甚至用来替代音频数字信号处理硬件也是可行的。

4.1.5　参考文献

[Buck04] Buck, Ian, et al. “GPGPU: General Purpose Computation on Graphics Hardware,” SIGGRAPH, 2004.

[Gallo04] Gallo, Emmanuel, and Tsingos, Nicolas. “Efficient 3D Audio Processing with the GPU,” Proceedings of the ACM Workshop on General Purpose Computing on Graphics Processors, ACM, 2004.

[GPGPU07] "General Purpose Computing Using Graphics Hardware."

[Jedrzejewski06] Jedrzejewski, Marcin, and Krzysztof, Marasek. "Computation of Room Acoustics Using Programmable Video Hardware," *Computer Vision and Graphics*, Springer Netherlands, 2006.

[Owens07] Owens, John D., Luebke, David, Govindaraju, Naga, Harris, Mark, Krüger, Jens, Lefohn, Aaron E., and Purcell, Tim. "A Survey of General-Purpose Computation on Graphics Hardware," Computer Graphics Forum, 26(1), pp. 80–113, March 2007.

[Rolfes04] Rolfes, Thomas. "Artificial Neural Networks on Programmable Graphics Hardware," *Game Programming Gems 4*, Charles River Media, 2004.

[Whalen05] Whalen, Sean. "Audio and the Graphics Processing Unit."

[Zeller05] Zeller, Cyril. "Cloth Simulation on the GPU," SIGGRAPH, NVIDIA Corporation, 2005.

4.2　多流——编写次世代音频引擎的艺术

Jason Page，索尼计算机娱乐公司欧洲分部
Jason_Page@scee.net

在过去的 3 年里，索尼计算机娱乐公司欧洲分部的音频小组一直在编写一个“次世代”音频引擎，这个引擎是 Playstation 3 官方软件开发包的一部分。本精粹的目标是：从索尼计算机娱乐公司欧洲分部的引擎项目角度出发，告诉用户如何设计和建立自己的音频引擎。然后，这些信息可能对建立自己的音频引擎有帮助，或者当用户意识到达到期望的效果所需要的工作量后，或许会选择使用某个软件开发包或者其他中间件来替代。

因为所有 PS3 的注册开发者都能够随时查看多流（MultiStream）函数的调用文档，所以我不打算对它进行详细的讲解，但是我希望能够让你注意到那些我的团队之前所必须解决的问题。本精粹的最后将会指出一些基于对次世代音频的期待而需要解决的问题。

在项目刚开始的时候，我们对于可用的硬件一无所知；我们不知道需要多少内存，也不知道效率应该达到什么程度。

我们决定不考虑硬件数字信号处理所能提供的任何帮助，而用软件来完成所有的功能。后来，我们发现这真的是一个正确的选择，因为 PS3 先前没有音频硬件模块。我也发现，根据已知的硬件和游戏需求来设计音频引擎，其结果是非常平庸的。如果你知道一个游戏需要一个用来处理闭塞（occlusion）和阻扰（obstruction）效果的低通滤波器，而不支持高通、通带（band-pass）、凹槽（notch）等其他很有用的滤波器类型，那么你会失去多少创意的可能呢？往大处想意味着你能删减一些长期都不会被使用的东西，同时也意味着你能得到和别人不一样的东西。

多流中的“流”由以下部分组成：用来播放的音频数据（最多 8 个声道）、播放频率、音量参数/环绕声位置、振幅包络线、数字信号处理效果和输出路线位置。

4.2.1　一切将如何开始

除了建立一个音频引擎的技术方面外，我们还必须决定引入一些其他的特性来使“次世代”的多流名副其实。下面的章节将解释更多我们想回答的问题。我想再一次申明，建立自己的音频引擎会是一个吸引人的点子，但是完成这样的工程可能要用好几年才行，而且一开始就需要你对许多事情制定详细的计划。让一个团队用 3 年时间开发一个音频引擎也不便宜。

下面将介绍我们在编码以前的设计阶段。

4.2.2 理解“次世代”音频

虽然建立一个音频引擎看上去很简单，但是要使游戏的音乐比原来更好听并不像它看起来那么直接。对于游戏开发者来说，播放 CD 质量的音频的能力已经是十多年以前的技术了。在数百个音频通道中增加高质量的回声（尽管它可能不符合 Yamaha 或者 Lexicon 公司建立的专业插件标准）在 20 世纪 90 年代已经进入了我们的生活。是不是在这个高度上就可以建立“次世代”的感觉了呢？别忘了，这个不是我们所能决定的，而要看张三李四花了上百元在某个游戏以后能不能留下深刻的印象。

如果你正在考虑编写一个自己的音频引擎，先问问自己打算让这个引擎具有哪些功能。对于多流，其中的一个目的就是要让游戏音频比当代的好听，而且在某种意义上要和当代的引擎有所不同。在与前一时代音频引擎相同的音频能力的基础上采用更多的声道，虽然这样可能通过建立更丰富的环境效果来让游戏声音更好听，但是好像并不能说这样的做法就很出众，可以称为“次世代”。当然，你可以用非实时的方法来做你想做的事情，但次世代的真正威力就是全部采用实时处理。诸如声音合成器和卷绕回声这样强大的声音效果都从来没有被实时地处理过，主要是因为这些效果都需要频域处理。这个在以前的游戏中曾经被认为是不可能实时处理的，但是我们的确需要这样的效果。这也很快就变得很明显，次世代音频需要我们从从来没有接触过的领域中获得专门的知识，以实现这样的功能。

希望实现的功能列表

从我对 PS3 音频的经验来看，一种对每个人都比较好的方法就是制作一个希望实现的功能列表，列出他们所需要的音频处理种类。在开始编写这个列表时，好像觉得任意一种类型的音频处理都不仅仅是可以实现的，而且都可以被实时地实现。给你一个例子，多流可以实时地处理超过 50 个单声道卷绕回声，然而这样做也意味着没有用来处理任何音频通道的空间了。但是，不可否认的是：对于以前的游戏来说是不可能的一个单声道卷绕回声，现在是可以实现的。PS3 上的音频编程让许多新的方法成为了可能，那些技术原先被认为只是在专业的音乐包中才会有。

有多少音频通道?

为了达到期望的效果，我们假定任何一个声音引擎必须能够处理足够多的声道数据。是的，更多的通道能够帮助产生更好的“音效”，但是莫扎特曾经说过：“音符间隙中的安静和音符本身一样重要。”即便如此，今天（还有明天）的游戏需要更多的音频通道来处理声音，纵然是将以前的声音重做一遍。这样，你就会很自然地想到一个汽车引擎的声音需要 25～30 个音频通道。

- 汽车引擎工作循环×8(对于乘以 8 中的每一个，我们可以说是 1 000 转速的工作范围)。
- 汽车排气管循环×8（与汽车引擎同时录制）。
- 车轮打滑的声音（4 个打滑的声音，一个轮子分配一个）。
- 路面轰隆的声音（4 个声音，每个轮子一个）。

- 齿轮切换的噪声。

由于在标准竞赛游戏中单个汽车需要用到如此之多的声道，以前只有玩家自己的车才会用到这么细致的模型。考虑到硬件的承载能力、CPU 和/或者内存的承载能力，其他所有的人工智能车所使用的声道都远远小于这个量。

对于今天来说，这个就不是什么问题了。多流最多有 512 个通道。在一个赛车游戏中，这么多的通道可以控制 20 辆车，所有的赛车都和玩家赛车有同样的音频能力。当然，你的使用量可以超过 512 个声音（依赖于你为哪个平台做开发），但是你最好定一个界限。对于我们的项目而言，保持这样一个限制意味着我们有更多的资源和时间来做数字信号处理、巴斯路由（buss routing）、重采样和振幅封闭。

最后，在看这篇文字时你必须注意到，在汽车引擎的这个例子中，引擎的交叉混合循环方法或许也已经成为了过时的东西。在汽车引擎的例子中使用“颗粒合成技术”（granular synthesis techniques）来播放小区域（或者“小颗粒”的声音），以此建立一个比单独循环播放更好的真实感的引擎声音。再一次地，直到现在，这个技术还不是真正可行的。如果不能使用诸如 MP3 或 ATRAC3 等文件格式，在理论上进行了尝试和检验的技术仍然会使用大量的内存。

采样格式

播放一个音频文件必须考虑到采样数据的格式和通道的数量。注意，立体声文件并不是必须使用单声道文件两倍的处理时间或者内存空间，这是取决于文件格式的。举例来说，MP3 关联立体声模式把一些左声道和右声道播放频率一样的音频数据作为单声道来录制，没必要把这些相同的数据保存两次。但事实上，的确有许多书和网站解释了你为什么需要将它们保存两次，但要回答这个问题还要写上好多页才行。

对于游戏音频来说，其中一个最重要的事情就是可行性。采样是否能够被准确地播放是你需要达到的目标。对于 MP3 格式来说，它能相对简单地播放音频，采样还原的失真不会超过±1000。因此，你可以多做一些工作（使用内存和 CPU）来使 MP3 完全准确地还原采样，或者像我说的那样，达到“游戏兼容”。

你也不得不考虑一下你的音频引擎能够接受哪些文件格式，音频引擎的文件格式如表 4.2.1 所示。

表 4.2.1　　音频引擎的文件格式

格　式	赞同和反对的理由	注 意 事 项
Float32 PCM	赞同： 不需要解码 最佳质量音频 采样边界的循环播放很方便 反对： 内存消耗大	处理速度更快意味着 CPU 可以更多地为其他任务工作
16-bit PCM	赞同： 很好的“CD 质量”音频 采样边界的循环播放很方便 比 Float32 使用更少的内存 反对： 内存用量还是很大 好像游戏不会有那么多内存空间可以存放这种格式的采样信息	在游戏中比 Float32 格式更有用，而且在许多情况下，玩家是听不出什么区别的

续表

格　式	赞同和反对的理由	注 意 事 项
ADPCM	赞同： 过得去的质量 比 PCM 使用更少的内存 解码速度非常快 反对： 解码器很可能只对单声道输入文件起作用 解码需要更多的 CPU 开支 采样边界的循环播放可能做不到	在许多情况下，这个编码仍旧是游戏音频格式的一种标准。ADPCM 要求压缩和快速解码。采样准确性和循环播放时的查找可能做不到；它也对输入的信息不需要太多的把握来表示采样的边界
MP3	赞同： 质量好 优秀的压缩 许多解码器能够掌握多声道数据 反对： 高 CPU 开支 在采样中找到准确的播放位置并不容易	把所有声音同时载入内存的最好方式，但是你必须考虑到处理时必须解码这种格式

你还需要注意的是 MP3 的编码对每个音频通道也需要额外的数据缓存，解码后的数据和其他信息都保存在这个缓存中。考虑到多流能够同时播放 512 个 MP3，即便每个音频频道只需要 2KB 的缓存，编码仍旧会占用 1MB 的空间。虽然这个看起来是明显的，但是在游戏开发流程中还是一块需要和游戏制作人及策划事先说明清楚的地方。

还有，如在表 4.2.1 中，虽然 Float32 输入会得到最佳的质量，但会引起的另外一个问题就是 DMA 带宽问题（在系统周围传输数据的方法）。在这个问题上，使用 16 位的数据只会占用一半的带宽，同时还是能够提供 CD 质量的音频效果的。

循环标志位也是需要考虑的一个内容点。循环标志位可以被保存在文件的头结构中（类似于.WAV），在采样数据中（例如，SCE 的 VAG ADPCM 结构），或者根本不保存（例如，从.WAV 文件转换过来的 MP3 文件，循环信息已经被丢失）。控制音频的循环根本不像它看上去那么简单。如果采样文件是在内存中驻留的，你只需要播放就可以了，内存中的地址标明了循环的位置。如果你用的是流音频内容，那就需要小心了，只有当循环点被载入内存，才能去循环播放。

是流还是非流

大多数音频系统需要被告知数据可能是流状态的。这里，你的音频引擎必须迎合某种缓存技术，数据会被复制到内存中的一个区域进行播放（这种数据通常是从磁盘中载入的，但也有可能是通过音频引擎外部的解码器从.MP3 文件里面解码出来的 PCM 数据）。

从我的经验来说，我不会建议你的音频引擎内部有控制数据载入的函数。如果音频引擎需要更多的数据来制作流缓存，那么就要这么做了（在多流里面，这是通过回调函数来控制的）。如果你开始在音频引擎中控制数据的载入，后面等待着你的将是世界级的痛苦。这里说一下原因：

- 你的音频引擎载入数据的时候，需要和其他游戏数据载入同步执行；
- 你需要控制所有数据损坏的情况（在数据载入时磁盘移走或者磁盘是坏的）。

本质上，你的音频引擎因此就变得非常难以操控，以至于无法优化和维护。另外，你还是要注意到音频流必须比其他数据的载入具有更高的优先等级。为什么呢？简单来说，如果

音频数据流没有及时在播放时送达，你就不得不要么重复播放上一段缓存（听起来像是一台坏了的 CD 机在播放），要么什么都不播放。在质量保证阶段，这两种可选方案都会使游戏称不上“次世代”的。

甚至如果你的音频系统并不直接控制数据流从磁盘上载入的过程，负责输入/输出系统的程序员也必须了解下面根据重要性排列出的优先等级：

- 必须首先更新一些当前播放的流；
- 然后可以更新一些新提出更新需求的流；
- 最后为游戏载入数据。

使用这种方法，如果一个播放器一直需要更多的音频成为流，也就是说，在游戏中的每一帧，任何一个目前正在播放的音乐将仍旧会被正确地播放下去，而不会被忽略或者跳过。在许多情况下，流在类似于体育节目实况报道的这类领域中被使用到。这种方法也经常出现在屏幕上目前动作的前后连接内容里面。正如我所关心的，这也就是播放这样的音频比载入游戏数据更重要的原因了。没有比实况解说员在错误的时间说了错误的话更糟糕的事情了。

没有哪门科学是去研究流缓存大小的。流缓存的大小会依赖于你的数据格式和采样频率，也取决于你载入数据的频繁程度。从硬盘上读取数据流绝不会像在 DVD 上读取数据流那样困难，当然，从 DVD 头机制物理地移动到正确的位置所要的时间以及载入数据的时间也会是另外的因素。在许多情况下，把同样的文件复制在多处是一种加快 DVD 载入数据的技术方法，就是在软件中保存当前的头位置，然后流引擎（注意，流引擎和音频引擎是两个分开的引擎）将会选择一个最近的文件来读取。

这个方法也能够帮助音频数据制作成流时做优先等级的排序。如果多个音频声道需要更多的数据，你必须选择那个数据最优先被载入。当然，也不存在研究它的具体科学。如果多个流需要更多的数据，那么你需要保证它们能够尽快得到那些数据。如果你不能及时载入数据，简单的做法就是增加流缓存大小或者减少音频的采样率。采样率减半的效果和加倍流缓存大小的效果是一样的。例如，用 24kHz 来播放 48 000 采样率的音频文件，会比用 48kHz 来播放 48 000 采样率的音频文件时间长 2 倍。

对于判断音频播放时的那些爆破音是不是因为系统处理的数据已经用完了所造成的，减少流播放频率的方法也是很有帮助的。这样的修改通常对于任何音频调用都能够很方便地用上；而与之相对的增加流缓存大小的方法就会受到非音频游戏需求的限制。

对于有循环播放标志的流音频（要看是什么采样数据格式），你只有到达循环标志的时候才会知道需要循环播放，那样太晚了。对于多流，我们决定在音频引擎中忽略所有的循环播放标志。因此，这个取决于用户是解码.WAV 文件的头结构，还是对需求数据制作准确的流，这样做不仅让用户能够进行控制，而且还能够关照到先前提及的所有问题。

所以，如果我们需要循环播放文件内的某一段位置，可以先检查一下这段内容是不是已经在内存中了。如果不是，我们需要先把这段内容载入内存，然后就可以按照意愿继续播放了。

音量参数

设定一个音频通道的音量等级以及设置它的频率是两个非常基础的音频数字信号处理效果。

对于多流来说，你还需要决定许多事情：PS3 可以输出 7.1 声道的音频，你需要允许任

何音频通道都可以传递到任何数量的喇叭上。例如，一个单声道音频信号可能要在左前方和右后方的喇叭中发出，这就意味着任意一个声道都需要 8 个音量参数。此外，当多流能够播放携带了 8 个音频声道信息的数据时，每个流就会有 64 个有效的音量参数信息。最后，多流可以处理高达 512 个声道，并且这些声道的音量参数可以是 Float32 类型的。这也就意味着单单是音量参数信息就需要 512×64×4 个字节。

如果使用 16 位音量参数的话，你可以减少内存使用量，或者有可能让减少的幅度更大。想象一下，MIDI 的音量参数范围是 0～127，给你的是 128 个可能的设置，你为什么需要浮点数作为参数而得到上百万个可能的设置呢？首先你要问的是，在你设置 MIDI 音量参数时，硬件（或者软件插件）是不是直接使用它们的呢？这个数值有可能被缩放到浮点数系统中，在这里，音量被转化成想要的音量等级。其次，为了使用方便，通常情况下有一个浮点数的系统，可以让音频引擎的其他部分运行得更快。一开始，音量等级和多种格式之间的转换就不会那么多了。

播放频率

在“音量参数”这部分开始的时候我们提过，音量和频率是两个最基础的音频数字信号处理效果。

对于一个基于纯软件的系统，在设置一个采样文件的播放频率时都需要谨慎行事。任何重新采样的做法都会占用 CPU 的时间，浪费时间来做这么基础的事情并不明智。慎重考虑的不仅有重新采样算法，还有用高频率来播放音频，这会用掉更多的处理时间。因此，一个能处理 4 000 个音频通道的系统可能最高只能按 48kHz 的速度来播放。

再者，如果需要用 48kHz 的频率来播放一个一秒的音频数据，那需要处理 48 000 个采样。如果要用 96kHz 来播放，就需要处理 96 000 个采样。“处理”在这里可能意味着 MP3 文件的解码工作。所以你也看到了，如果是 MP3 文件，用 96kHz 来播放 48kHz 的文件，你需要做的解码量是原来的 2 倍。

有没有更明智的方式来使用这个处理时间呢？如果需要播放一段音乐，这段音乐被播放时要求比原先的音调高八度，那就可以把这段音乐重新采样成 24kHz，因为调高了一个八度以后就变成了 48kHz。你只要简单地从文件采样率入手，就可以减少一半的处理时间。

频域处理

对于任何频域处理来说，都是在说 FFT/iFFT 常规。理解 FFT 的详细细节并不像它看上去那么重要。的确，那里包含了许多数学内容，但一旦写好了，就不需要再去关心了。相应地，使用它就是另外一回事了。

频域处理的一个主要问题就是选择正确的窗口大小。如果你的效果需要一个高频的分辨率，你就需要一个大的窗口。但是大的窗口在时间上反应就缓慢一些，就不能快速地改变参数。大的窗口还会有很大的延迟，需要更多的内存空间和处理时间。虽然一个较小的窗口不会遇到这些问题，但是它会导致频域的分辨率很低，进而就很难达到执行频域效果的目的。

如何给出正确的答案？主要是通过调整窗口大小，让程序充分地利用可用资源，进而找到合适的值，最后还要仔细倾听处理后的声音是否正确。窗口的大小还要根据效果的不同来调整。你是准备根据信号路径的不同而调整窗口的大小，还是做一个固定大小的窗口“一劳永逸”呢？

FFT 的基础

在使用 FFT 时要考虑好几件事情。首先，输入采样的数量必须是输出采样数量的 2 倍。所以，举例来说，在使用 FFT 时，你需要输入 1 024 个采样，才能够得到 512 个输出。其他的常规也有这样的连带关系，就好比振幅封闭，你需要处理所有的 1 024 个采样，然后反过来用 512 个采样播放一遍封闭参数，只有这样，下次振幅封闭才会被处理（见图 4.2.1）。你用的值要对哦！

图 4.2.1　简单的振幅封闭（淡入/淡出）超过 2 048 个采样

使用 FFT（以及窗口）的一个非常大的好处就是你会发现这样做能够移除许多爆破音，这些爆破音经常出现在较大幅度的音量调整时或者播放到采样不符合的边界处。

正如在图 4.2.2 中所看到的，在每一步中，对于数据包的前半部分，封闭必须被重新计算一次，甚至尽管它刚刚才计算过前一个包的后半部分。

注意，有些系统在这一步之前还会有一个步骤，就是把前一步的计算反过来再做一次。振幅封闭的淡入部分只处理前面的 512 个采样。图 4.2.3 描述了该内容。

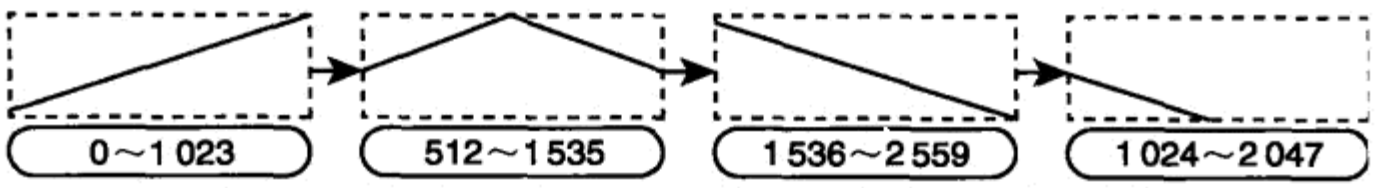


图 4.2.2　如果使用窗口技术来处理，处理 1 024 个采样包时起码要有 4 波数据

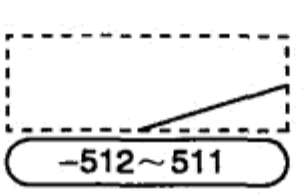


图 4.2.3　有些系统使用只处理最前面的 512 个采样的渐变技术

这一步在最后输出时增加了许多延迟，而这个延迟在实时的应用程序中太明显了。

延迟

很明显，实时的应用需要比较低的延迟。对于多流来说，我们决定让它在每次常规更新被处理的时候，每个通道都能产生 512 个采样（这个技术被称为粒度）。当使用 FFT 的时候，因为窗口化函数的需要（参见“FFT 的基础”部分），输出 512 个采样需要输入 1 024 个采样才能完成正常的工作。

使用 512 个采样粒度，就能够为 FFT 函数提供足够的数据来满足两个目的：

- 满足绝大多数游戏需求的低延迟；
- 512 的频带（当多流使用 1 024 个采样作为输入数据时）为许多基于 FFT 的数字信号处理效果提供了足够的余地，但同时也保持了很高的质量。如果你降低到 256 的频带（512 采样作为输入数据），那就会发现音频质量对于任何应用来说都太差了。

处理 512 个采样意味着更新例程每秒将会被调用 93.75 次（每 10.66ms 调用一次）：

48 000 个采样 = 1s 的播放

48 000 / 512 = 93.75Hz（每秒需要更新音频的次数）

1 000 / 93.75 = 引擎每 10.66ms 被调用一次

通常来说，这样做就可以给音频提供足够快的速度来和每秒最多 60 次的图像更新保持同步了。虽然输出 512 个采样可能看上去并不难（记住，一秒需要播放 48 000 个采样），处理类似 MIDI 编曲会需要比这个更快的速度。在许多情况下，这些编曲的频率会高到 240Hz，甚至到 384Hz（在 2～4ms）。因此，问题是：如果 MIDI 编曲需要一个乐器开始播放，那么这个乐器可能要到下一个音频更新才会被实际播放出来。许多人不会注意到这点，但是那些听觉很灵敏的人（例如，那些用你的音频引擎来调试他们制作的音乐的音频工程师）会发现这一点。如果你从事的工作要求一个比 512 个采样更低的延迟，FFT 处理可能不适合你。

对于多流的处理来说，我们既采用了频率域模式，也采用了时间域模式。所以如果你不需要频率域的效果，那么就可以完全用时间域来操作，这就意味着窗口大小不再是一个问题，我们可以采用可选的 128 或者 256 的粒度设置。

包平滑

就像在“延迟”部分中讨论的那样，粒度就是在每次音频更新时产生的采样数量。在最简单的层面上，每次更新都会采用用户对每个音频通道设置的参数，例如，播放的频率和音量。这里需要考虑的一个问题是：如果每个包都使用被设置好的音量，突然大幅度调整音量就会造成锯齿瑕疵。另外一个瑕疵就是爆破音，例如，在播放汽车引擎多音频通道交叉渐变时（这些渐变的条件依赖于机器工作的情况），就会很容易地注意到这一点。

对于时间域处理来说，这里就需要有一个过滤处理，让音量变化变得很平滑；而对于频率域处理来说，你会发现对于 FFT 所需要的窗口（例如，海明（hamming）或海宁（hanning）窗口）会帮你把所有的难题解决掉。

在“频域处理”部分的开头，我们讨论到，当处理频率数据的时候，我们会用到窗口技术。转变成频率域并做窗口化的理由就是：当你处理数据的时候，你只要关注某一段数据就可以了。因此，分析一下就可以得出结论：如果不把前面被处理的数据和后面跟随的数据一起考虑进去，每段数据之间就会出现断层现象（对于你我都熟悉的爆破音）。海宁或者海明窗口类型在本质上就是一种修改每个数据包的算法。每个数据包会被处理，然后和前一个包混合，产生一个和希望得到的数据相似的输出。这是一个非常简单的段落，通常在其他的书上也能找到，但这里还是为你提供了足够的信息，希望可以让你用 Google 来查找它们。

窗口化也可能产生大量的错误，这些错误通常会造成输出爆破音，例如，循环播放的采样，它的开头和结束部分的采样数目不匹配。这种地方总是需要小心一点的。虽然用窗口技术可以让任何采样播放起来都很好听，但如果因为某种原因，你需要把你的音频引擎改换成“时间域”模式，而在改换的那段数据处，你又没有很好的窗口化类型或者滤波器来处理音频输出，那么你还是会听到这些爆破音的。源数据默认循环播放起来就是最好的。

4.2.3 环绕声音

考虑如何控制环绕声音。有两条路可以选择：

- 用户提供声音源和听众位置的 *X*、*Y* 和 *Z* 轴坐标；
- 用户提供相对于听众位置的声音源的角度和距离。

多流使用 *X*、*Y*、*Z* 的方法，并采用 OpenAL 1.1 的算法，尽管对于这样的一个例程来说，用 *X*、*Y*、*Z* 系统来建立一个环绕声音平面坐标（角度）和那个位置的全局音量（距离）无论如何也是可以的。

在环绕声音中处理多通道音频需要谨慎一些。

如果要在环绕声音中获得坐标，多流会把多通道音频包裹成单点音源。另一个控制多通道音频的方法就是把每个通道都作为单声道来播放（例如，通道 0=左前声道，通道 1=右前声道，两个声道加起来为一个立体声通道），然后为每个喇叭设置环绕声音的 *X*、*Y*、*Z* 坐标位置。图 4.2.4 显示了 6 个声道的音频。

这种方法可以用到赛车游戏中，它把摄像机的位置从玩家的背后移动到玩家所在的车子里面，这也就意味着所有的音频播放都必须正确工作，比如，废气排放管的声音是从玩家的背后播放的（见图 4.2.5）。

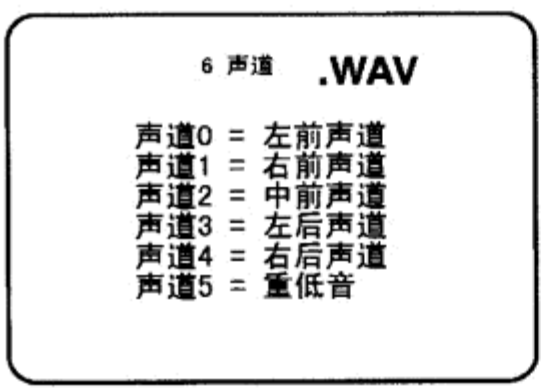


图 4.2.4　音频的 6 声道。把.WAV 分离成不同的声道，可以在游戏世界中定位每个喇叭的位置，来达到重现所希望达到的效果的目的

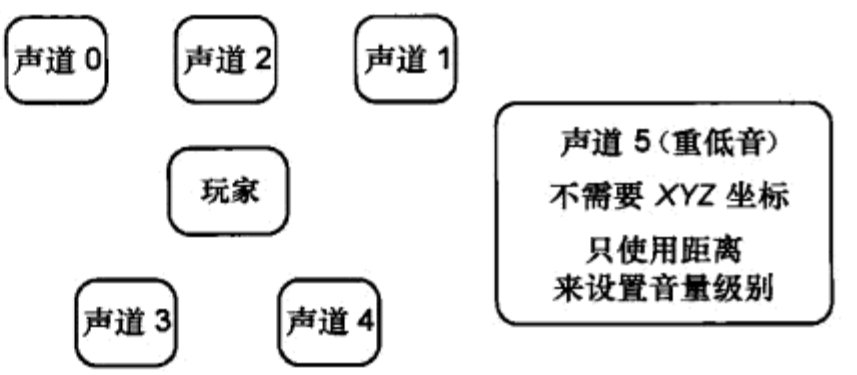


图 4.2.5　相对于玩家位置的声道定位

对于这种类型的游戏来说，通常的做法就是把非玩家的音频作为单声道（点源）来处理，如果需要，玩家特定的音频可以作为多声道来处理。因为玩家总是在摄像机的前面，可以安全地假设：它们的音频不需要环绕声处理，并且只要把它们的音频作为立体声来播放，效果就会很好。这样做不仅可以允许有更高质量的采样，同时，因为在游戏世界中有更少的环绕声物体，还能节省处理的开支。

同步声道

音频编程中经常出现的一个问题就是同步多个声道。同步多声道可以使它们在同一时间开始、结束和调整音调。如果不采取措施，你有可能听到不同步或者合声效果。

从图 4.2.6 中可以看到产生这个问题的原因。

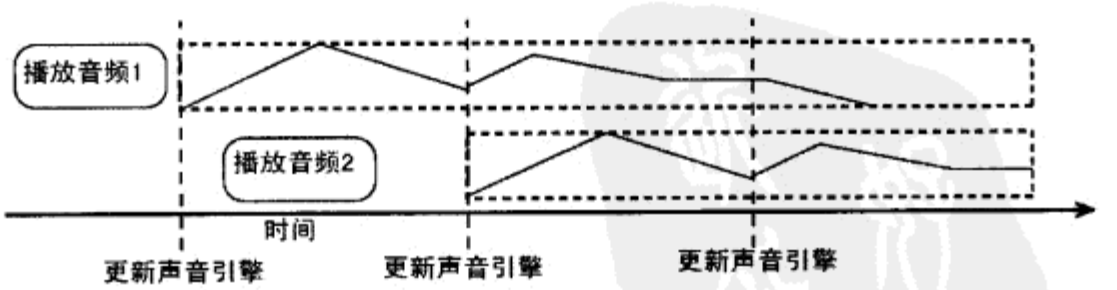


图 4.2.6　如果音频引擎在播放音频的命令之间进行更新，声道可能就会产生不同步现象

在图 4.2.6 中，你可以看到两个音频声道在播放声音，但因为音频引擎更新常规在初始化两个音频声道之间进行更新，“音频 1”的输出比“音频 2”的输出超前一个数据包。从多流的角度来说，就是“音频 1”比“音频 2”提前了 512 个采样。在暂停和继续播放声道以及调整多声道音调的时候，不同步的情况也会发生。如果两种情况都发生，只会离同步的目标越来越远。

这个问题的解决方法是：你要确认任何会产生不同步的函数（播放和音调改变的函数）不会被音频更新例程分割。最简单的做法就是用两个函数来处理：

```
void Sync_On(void)
void Sync_Off(void)
```

在这两个函数中间调用的任何播放或者音调函数都会被“记忆”下来，然后它们会在下一个音频更新，也就是在 Sync_Off 函数以后调用的音频更新常规中处理，如图 4.2.7 所示。

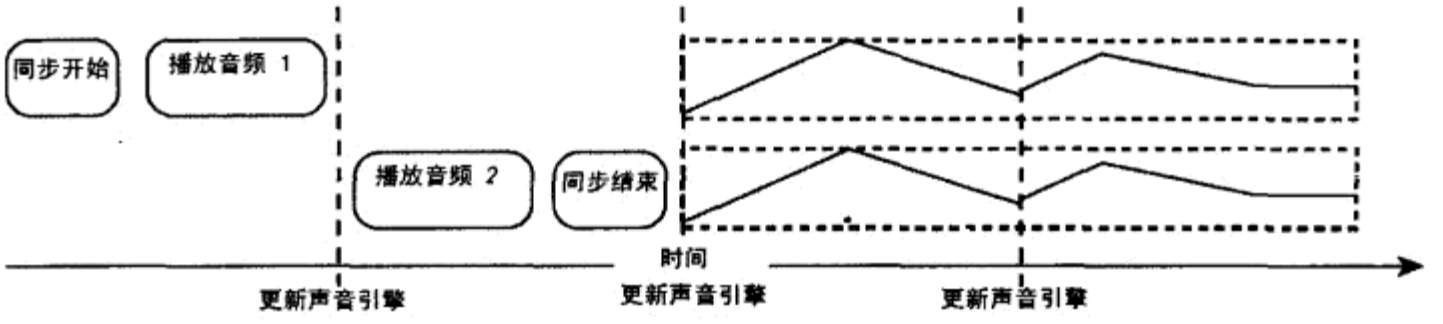


图 4.2.7 在同一音频引擎更新中，排成队的播放音频的命令同步地开始执行

就像在图 4.2.7 中看到的那样，“音频 1”现在要等到“Sync Off”函数处理结束，也就是说两个通道现在都按照希望同步播放了。

数字信号处理效果

数字信号处理效果把“次世代”分割成了现时代和上时代两个标题。以我在 PS3 上的经验来看，有效的处理能力让实时音频处理成为了可能，而在质量上使用最少数量的 CPU 通常只能在专业效果的部件上体验到。

本精粹的目的不是讨论每个数字信号处理的效果。目前已经有许多书籍涉及了滤波器的设计、FFT 和许多其他的效果。因此，就让你们自己去研究这些领域吧。

毫无疑问，只是用低通滤波器来制作闭塞和阻碍效果，并不能让你的游戏成为声音上的次世代，但是你也只是需要稍微再努力一下来给大家留下好印象。想想那些能用到普通音乐和声音效果制作上的数字信号处理效果，然后再想想哪些效果可以用到你的游戏中。例如，可以想想让每个关卡中的每个房间都有它自己独特的回声效果。因为“万事皆有可能”，最好的方法就是开始时让你的程序员和音频工程师交流一下，让音频工程师告诉程序员他们想在实时中听到的效果，以及为什么要这样的效果。

当然，实时地处理数字信号效果也就意味着预先处理音频采样的工作会更少一些。考虑到一个游戏会包含几万个采样信息，允许开发者按照自己的意愿来修改参数并在游戏中实时处理，总比回头让音频工程师去修改一大堆的采样信息来符合某个效果要好得多。想象一下，

你决定让一个人说话的声音通过无线电头带耳机传出的效果会更好一些。不需要浪费时间做数据预处理的能力，不仅可以加快开发速度，而且在调试时你可以得到更多的创作灵感。

4.2.4　路由引导

一个音频声道可以被混合到的总线数量是不可以被低估的。对于多流，我们目前有一条主总线和 31 条次总线。很明显，以后这些数值是会增加的。以前这组声音源是用来做音量范围调整的。例如，所有声音特效会引导到一条总线上，所有的音乐被引导到另外一条总线上，所有的实况声效则在第三条总线上。然后这 3 个音频参数就可以通过游戏的“选项”菜单来修改，通过调整这些参数来调整这 3 条总线上音量的大小。

今天，当我们用到那么多音频声道来创建汽车引擎那样的物体时，总线就不仅仅使用在调整音量这样不起眼的小事上了。通过在总线上增加数字信号处理效果，可以使用比较少的 CPU 强度，而且更简单地一次性对所有各个部件设置这种类型的效果（见图 4.2.8）。想象一款汽车游戏，当你看到一辆车绕到一个物体的背后时，你可以通过总线进行一次性处理，而不是用低通滤波器对 30 个或者更多的音频声道进行处理。

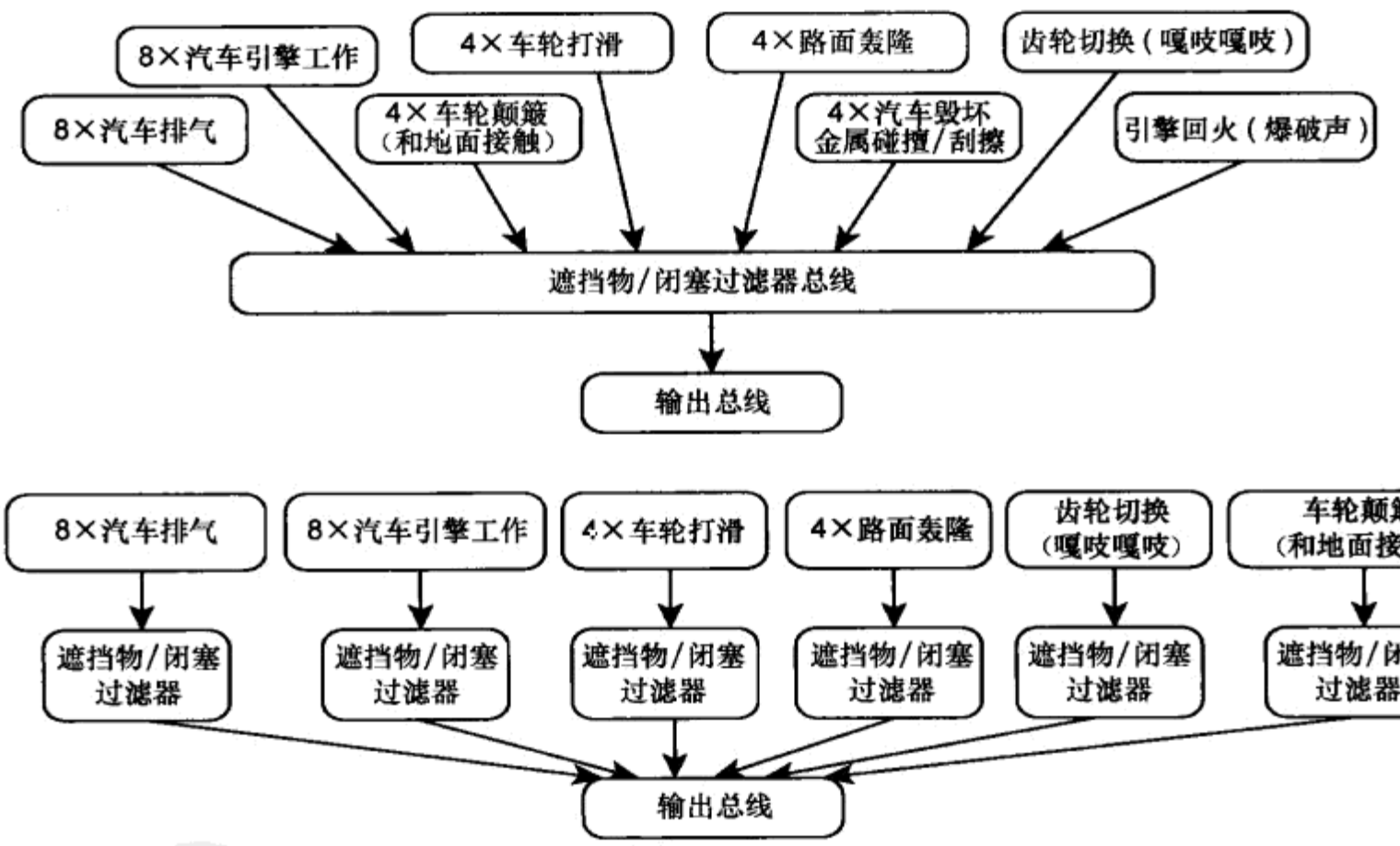


图 4.2.8　把数字信号处理效果都放在总线处理，这样可以减少每个声道处理的数量

4.2.5　结论

建立一个好的主调配器目前还被认为是一种“魔术”。事实上，在这件事情上，你看上去总不可能做好。不像电影、做游戏有太多的不可预测性。你永远不可能确定摄像机会指向哪里，或者玩家会处在什么位置。试着播放正确的声音并不容易。

以前我们曾使用过闪避技术来提供一点点清晰度。大多数体育游戏在实况报道的声音出现时会自动降低其他声音的音量。早期，我们使用一种“如果我正在播放实况报道的话，就把其它所有的音量都降低 *X*%”的方法来控制。在现实世界中，闪避器（或者侧链压缩机）就会被用到。该技术分析音频输入的信号（在这个例子中，实况解说员的声音就是被分析的信号），然后相应地降低其他输入信号（所有其他音频）。

该技术在次世代游戏中很容易被引用，而且远远比以前的方法更接近现实。以前的方法并不检查正在播放什么样的解说，它就这么简单地处理了。如果在解说音频采样中有一个比较长的静音，其他音频的音量还是会被降低。使用闪避器数字信号处理效果就不会引起这样的问题。

优先级系统也能够用来帮助你听到对于场景来说更重要的音频。哪种声音对场景来说更重要，这取决于游戏引擎的选择。例如，想象一下，在一个游戏中有 10 个敌人都站在离玩家相同距离的地方向玩家射击，你就必须选择哪个声音更重要了。可能你需要根据敌人面对的方向或者敌人使用的武器来排个序（可能激光步枪要比手枪更具有威力）。

优先等级的数量也是一个因素（高优先级会覆盖低优先级的声音）。我以前曾经写过一些系统，这些系统给予使用者 256 个声音特效的优先等级。虽然这感觉上是个好主意，但是实际上使用 122 的优先级和 121 的优先级并没有很明显的不同。如果采用更小范围的系统，例如 0～7，就管用多了。

把两种技术（闪避器和优先级系统）混合起来，就能让你自动产生一个主调配器。这里我们会使用到一定数量的总线，每个优先等级分配一条总线。除了那条有最高优先等级的总线外，其他的每条总线都分配有一个闪避器，然后每条总线会和其他总线有一个配合关系。0 总线会闪避 1～6 总线，总线 1 会闪避 2～6 总线，总线 2 会闪避 3～6 总线，以此类推。接着，确认你的音频都引导到了正确的总线上。如果没有什么其他的问题，你的音频音量就应该被控制得很正确了。这样的做法让使用者参与的操作减到了最少，相关的操作就是选择音频，然后引导到相应的总线上，这和你选择声音的优先级一样。

在多流情况下，这种路由引导和数字信号处理设置是有效的，尽管我承认还是有许多其他的问题值得考虑。其他总线可能含有回声效果，你要知道如何设置这 6 个优先级的总线和其他总线之间的路由引导关系。从后期制作的那些值来看，即便是这样，我也相信这其中的一部分能够把游戏的感觉做得像电影一样好，而且作为次世代的标题来说，目前只有这一种方法能做到这个程度。

4.3 听仔细了，你应该不会再有机会听到这个了

把游戏里音频环境中的重复去掉，并讨论声音设计的一个新方向

Stephan Schütze
超群脱俗

在桌子上丢一枚硬币，然后倾听它产生的声音。再丢第二次、第三次，或者上百次地重复，你会发现它的声音基本不可能重复。声音的产生是基于无数的原因的，可变性极强，这和科学测量的声音是无关的（例如正弦波）。大多数游戏中的声音就不一样，它们基本上都是静态的，或者变化受到了限制。有时候，声音的多变性是值得去做的。因为从大体上讲，一种声音效果没有变化或者以很小的变化进行重复，会降低游戏环境的真实性，而且更重要的是，玩家会有挫折感，甚至还会厌烦。

在某些时候，这种在游戏里建立实时变化的声音效果的技术是可行的。这些技术不仅解决了简单的声音重复问题，而且还能在游戏中产生比一些游戏机上提供的有限资源更复杂的音频素材。在最新一代游戏机里添加了这种优势以后，音频策划就可以建立很丰富的音频环境了，这样的环境可以拥有能反馈并且真实互动的声音和音乐。最终，开发者的设计就能达到像几年以前图形在互动娱乐界达到的那种不可思议的等级了。

本精粹将讨论建立这些更复杂的可变声音环境背后的方法论，当然也将从音频素材制作者的角度描述我们的想法。我也会向音频制作者介绍一些有效的工具。本精粹的目的不仅要告诉大家目前有效的技术，而且也会告诉不是音频策划的读者如何来测试自己的作品。我也希望能够告诉那些潜在的制作者们目前有哪些非常好的音频环境。

4.3.1 如何做到？采用不同的理念！

第一步就是把在电影和电视里面传统的静态线性音频挪走。游戏是不会在线性状态下工作的，要成为好榜样的需求驱使着游戏产业经常努力去达到电影产业的质量和产品标准。在游戏技术发展的初期，这曾经是一个很有用的基准。但是当游戏和一个大预算的电影或者电视的差距越来越小时，我们就越是应该想办法超过它们。目前已经很明显了，在不久的将来的发行娱乐业中，游戏会超过电影和电视。正因为是这样，产品质量的基准就应该超越线性媒体。在互动娱乐产品中，音频领域应该成为领头羊，超越电影标准。选择静态的、事先制作好的声音，然后在游戏中的某个地方被触发播放，虽然说这样做已经相当不错了，但这样会完全扼杀策划和开发者创作的可能。

这个技术的基本原则就是从它们相对独立的音频模块中建立复杂的声音。虽然这样一段接一段连续播放声音的效率有些低下，但是用这样的技术来创建多变的声音，并把这些声音整合成一个音频环境，通常会占用更少的内存和更少的资源，而且这比事先已经实现好的声音效果要有趣得多。这种技术也给声音策划提供了可添加到游戏中去的大量可能的选择。这样你就可以使用更少的内存，而在游戏中没有重复地播放更多的声音了。开始学习这个处理方法时，对于声音策划来说会有一个比较陡峭的学习曲线，可能需要更长的时间来跟上。但是当你学会了以后，你就可以为以后的项目提供源源不断的原料，而根本不必冒险去重复使用同样的音效库了。

4.3.2 前进，砰！

我们可以从录制一段声音到引擎中，并把这段声音作为建立游戏中所有其他声音的核心模块开始考虑问题，这样会很有帮助。其实这也没有什么大的区别，出去录制几段音频素材，准备好声音源，把它们混合到一起，产生一个我们想要的声音效果。这里不同的地方就是我们在游戏中把所需要的声音给混合出来。这个方法需要你对预制作声音效果库比较熟悉，但这个好处值得你去努力。

爆炸在许多游戏中都有大量的需求。我更喜欢把它们看成爆破音。我用了“爆破音”这个词，是因为它比爆炸涵盖了更多的内容。爆破音出现在许多射击游戏中，作为手榴弹、导弹、火箭引爆或者任何可以爆炸的物体的声音，也出现在许多平台游戏里来描述对手被击垮，玩家拿到了什么东西，或者如传送、隐身、得到无敌等特殊的效果。真实的爆炸效果或者上面所讲的这些效果和“爆破音”在结构上是相同的，因为它们包含了许多相同的元素，或者它们包含的所有元素都是一样的。

你首先要理解真实的爆炸通常是某种化学反应的能量释放，这一点很重要。真实的能量释放会产生“砰”的一声巨响，这声巨响会附带着合声或者回声，然后渐渐消退。在好莱坞大片中听到的爆炸声是经过特殊处理的，那是一种爆炸刚开始时能量对周围事物产生影响的结果。所以玻璃被震碎、树木被扭断、金属被弯曲等效果不会出现在爆炸刚开始的那段时间，它们是爆炸能量以一种摧毁的方式对树木、金属和玻璃等物体产生的一系列效果。然后这些被摧毁的东西也会用差不多的方式进行连锁反应。所以正常的情况是剧烈爆炸的声音很快就消退了，紧接着的是那些被爆炸波及到的东西，例如，一个玻璃盘子被震碎了，数以千计的玻璃碎片到处飞溅和散落，整合在一起就是一个壮观的爆炸效果。

在制作声音的时候，我们经常会碰到录制的声音听起来很模糊，没有什么生机。比如说录制的一段真枪开火的声音播放出来的效果会让你非常不满意。许多设计者会把好几段录制的声音整合在一起来做出一段新的声音。有些时候，录制的音轨是用来在某个频率范围上增加重音，在最后做出来的音频素材上增加深度。低频可以给一段声音增加沉重的影响力，高频声音则听上去更响亮。对最后制作出来的声音做效果调整还能让你想要的那个声音突出于其他的声音。对最后制作成的音频环境做平衡调整，需要慎重考虑用到的那些频率的混合以及声音的振幅等级。如果某个特殊频率范围的声音播放得太多，很容易让玩家变得疲倦，并且还会厌烦。

为了更好地理解如何制作声音，首先要做的是分解它。

- 开始时剧烈爆炸的声音/能量的涌起。一个很短促的、剧烈的爆破音，声音没有降低或者消失。想象一下鼓掌或者枪击的声音。

- 声音降低，然后消退。想象一下在大教堂里鼓掌的回声或者开枪的回声。事实上，这是爆炸声开始的那段时间里面的一部分，但是在分解爆炸声时，把它看做不同的元素还是很有帮助的。
- 受到影响的元素。所有受到初始能量影响的东西开始发出声音。
- 所有受到影响的元素的声音开始减弱。
- 第二阶段主体的效果。元素返回到未受影响的状态。想象一下较大碎片坠落的过程。
- 第二阶段非主体的效果。和前面的情况差不多，不过是较小碎片下落的情况，如灰尘、泥土等其他东西。

这个例子把一个传统的爆炸效果分解成了它基本的声音元素。有时候包含更多的音频素材可以显著提高最后的效果。你可以在游戏中按部就班地来完成那些爆破音。例如，在给小孩子玩的平台游戏中，你可以按这个方法来制作简单道具的音效。

- 开始时剧烈爆炸的声音/能量的涌起。一个很短促的、剧烈的爆炸，声音没有降低或者消失。例如，闹钟的响铃声，或者满是铃铛的树被撞击后发出的声音。
- 声音降低，然后消退。就好比钟声响起，然后逐渐消退的效果。
- 被影响到的元素经常产生好多种声音。就好像和谐的钟声，并和周围产生共鸣，但是没有那么剧烈的能量，音色也是随机的。
- 每个被影响到的元素声音的降低。就好像这里的钟声已经停止，其他地方的钟声还在响。
- 第二阶段主体效果。钟声开始敲打的泛音，或者合声交杂在和谐的钟声里越来越小。
- 第二阶段非主体效果。和谐的钟声开始渐渐消退，声音慢慢地停息。

在分解了爆炸、泡泡破裂或者钟声的形成以后，就可以利用那些单独的声音元素重新组建起一些声音了。当你了解了各种原始声音元素的听觉效果以后，就可以使用很有限的元素巧妙地编排出令人信服的爆破音。那么，让我们来实际操作一下吧。前面我把爆破音全部分解了，你已经清楚了那些组成爆破音的成分。让我们使用下面的元素。

- Big_Bang01–03：一个短小而尖锐的金属碰撞声。
- Stone_Fall01–02：一些石头被冲击后的能量所影响。
- Debris01–02：小型物体被影响以后恢复平静。

这些声音都用标准的 PCM 的.WAV 文件格式保存在光盘中。光盘中还包含了使用这 7 个基础声音制作出来的、已经被用在游戏中的 7 段样例。

7 个波形文件总共 629KB 大小，在游戏中组合成了 7 段不同的声音，这 7 段不同的声音大小总共 1.38MB。这些组合出来的声音文件都是从微软的 XACT（跨平台音频创建工具）里直接录制和输出的。Ingame_sound01 和 Ingame_sound02 的 3 个变形是用来表现变化情况的例子，这些变化本质上是没有限制的。IngameSound03 是用来描述从基础资源中也可以创建出完全不同的结果。

我限定自己只用 1 小时来收集这些基础声音，接着配置和创建 XACT 工程，最后试听和录制新的声音。我是故意这么做的，目的是想找找有哪些合适的工具是可用的。我可没说这些新制作出来的声音可以得什么奖，但是这些例子可以告诉你如何用一些简单的定义来快速创建实时的、无穷变化的真实感效果。我故意没告诉大家这些是什么声音，因为我不想在大

家没有播放这些文件以前就混淆视听。

4.3.3 旧的不去新的不来

图 4.3.1 描述了各种各样的文件结构，它们可能是用传统的线性音频编辑程序创建出来的爆破音效。那些轨道允许声音从不同的角度触发，水平轴是用来按照时间完成声音之间的相对定位的。这些声音自己能够任意组合，并产生期望的最终效果。

这是用来编辑音频和视频的传统方法：许多声音被组合在一起，渲染成期望的结果，并保存成某种规定的格式。策划曾经很喜欢这种方式。

图 4.3.2 描述了和图 4.3.1 一样的声音事件的安排设计、临时位置和重叠。但是在图 4.3.2 中，事件的安排只是一种重现，重现你想在实时状态下如何用游戏引擎将声音进行组合；在这里面是没有渲染过程的。这些声音事件也不局限于一个单独的声音文件。每个声音事件括号里面的数字描述了一个声音池的文件数量，这些数字是随机的，目的就是产生一个达到期望值的声音输出。对于每个声音事件的声音文件数量，其上限只被物理内存的大小所限制。

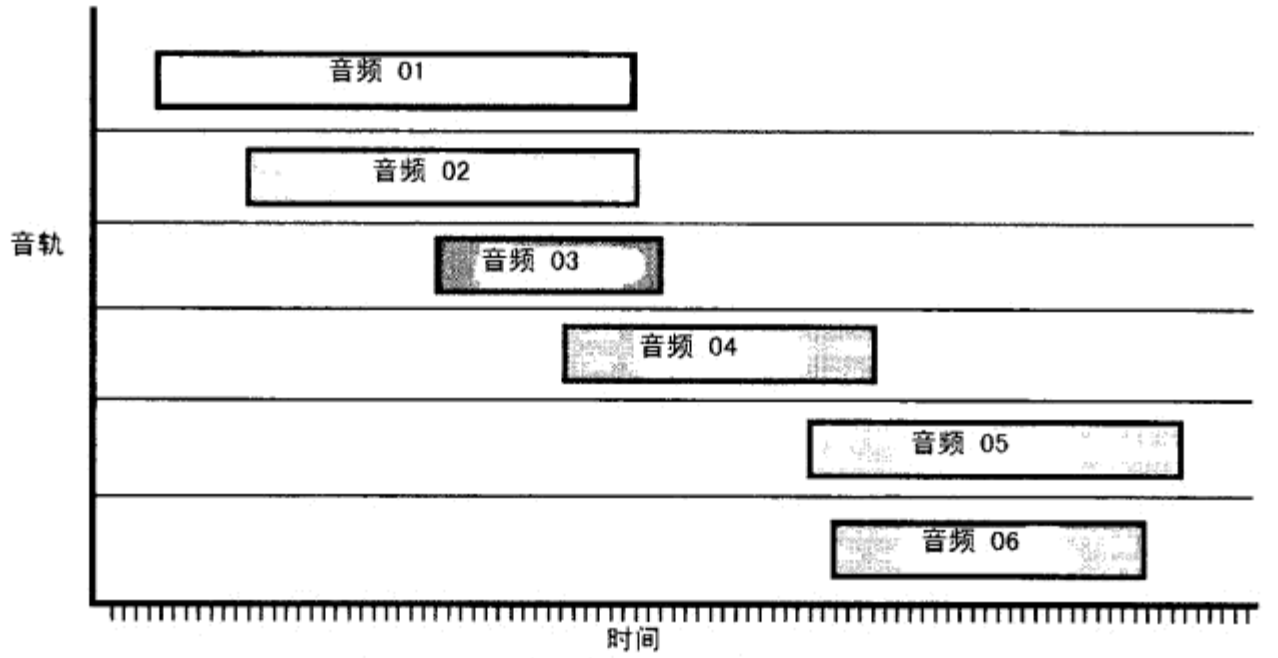


图 4.3.1 标准编辑软件显示线性编辑过程

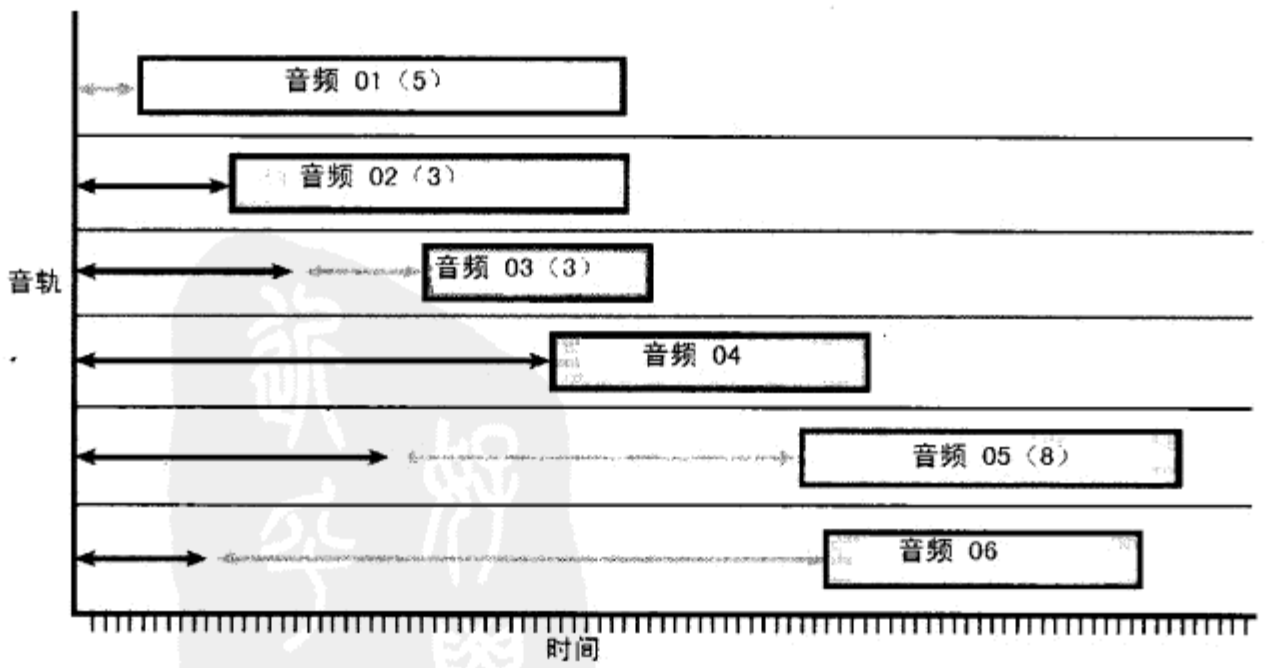


图 4.3.2 声音工具布局

该方法的另一个不同之处就是可以及时随机改变声音的方位。黑色箭头描述了时间的偏移值。每次播放声音时，音轨会在它即将播放以前算好时间偏移值。灰色箭头描述的是一个可变的时间偏移。在图中的情况下，这个声音被播放以前的时间是随机的，但是不会超过最大值。例如，声音 01 会在每次播放前稍微等待一下。作为比较，声音 02 等待的总时间是声音 01 播放所需要时间的 2 倍。声音 03 等待的时间中还包括更多的随机等待时间。这就意味着声音 03 有时候会在声音 02 开始播放后马上也开始播放，但有时候也会在声音 02 播放了一半的时候才开始播放。

用来创建声音的工具主要有振幅、音调和时间操作。把这 3 种因素有效地结合好，就可以制作出一个和原先完全不同的声音出来。目前所有的声音策划都用这些工具来创建他们想要的声音，并把这些做好的声音导出成合适的文件格式。这种方式替代了原先制作声音的工具。这种操作方式在游戏中是实时进行的。用这种方法来操作不会存在一成不变的渲染。一个声音是按照需求并使用提供的参数和源声音制作出来的，用完以后就可以将生成的声音丢掉。每次我们需要什么声音时，这个过程就会被重复，可变的参数就会被应用到过程中，然后一个不同的声音就被制作出来了。

4.3.4　称手利器

图 4.3.3 显示了 FMOD 的声音设计界面。你可以发现这个界面和前面的两张图有很多相同的地方。声音事件被水平整理到两个音轨层里。FMOD 在设计工具里采用声音事件而不是真实的波形文件，其好处就是允许编辑者通过一个声音事件操作多个声音文件，就像图 4.3.2 中显示的那样。在图 4.3.3 中，声音事件重叠的目的是希望达到一种声音之间交叉消退的效果。

在 FMOD 中，一个显著的特性就是水平轴不再被限制于只描述时间了。这是和传统编辑声音方法的另一个很大的区别，而且这个区别所带来的效果是非常显著的。在图 4.3.3 中，在水平轴上进行移动就好像在描述引擎的转速一样，但是你也可以简单地认为这是在描述高度、速度或者游戏里面人物血槽的数值。根据参数的定义和影响，声音就随之变化。这些系统的强大之处就是可以让创作人员随心所欲地设计参数和编辑音频，这样就会显著减少编码人员的工作量，因为编码人员可以根据提供的很少的几个标志来连接这些音乐文件。在汽车的例子中，一旦加入了声音，编码要做的事情就是把游戏中引擎的转速数据和 FMOD 中的转速标签对应起来。

在图 4.3.4 中，微软的 XACT 音频工具和 FMOD 声音设计工具的界面有许多不同的地方，但是许多属性和强大的地方都差不多。XACT 使用的是设计人员自定义的波形文件库和声音库。声音库中保存的基本都是那些制作好的声音，每个声音事件可以像在 FMOD 中一样把许多声音文件组合起来。在创建声音的时候，随机音调和音量的参数可以在多个层面上进行修改和调整。这样，设计人员就可以随机调配声音事件中的声音文件，然后调整音调或者根据需要改变最后的事件。XACT 的工作方式和图 4.3.2 中的例子所显示的是一样的，不同之处在于：XACT 用的不是传统的线性编辑窗口。

在某些情况下，这种做法是好的，因为它会强迫用户用不同的方法来创建音频素材。虽然 FMOD 支持几乎目前所有的平台，但是 XACT 只能运用在微软的平台以及 PC 下。当然，

在支持的平台下，XACT 的界面还是很方便、友好的。

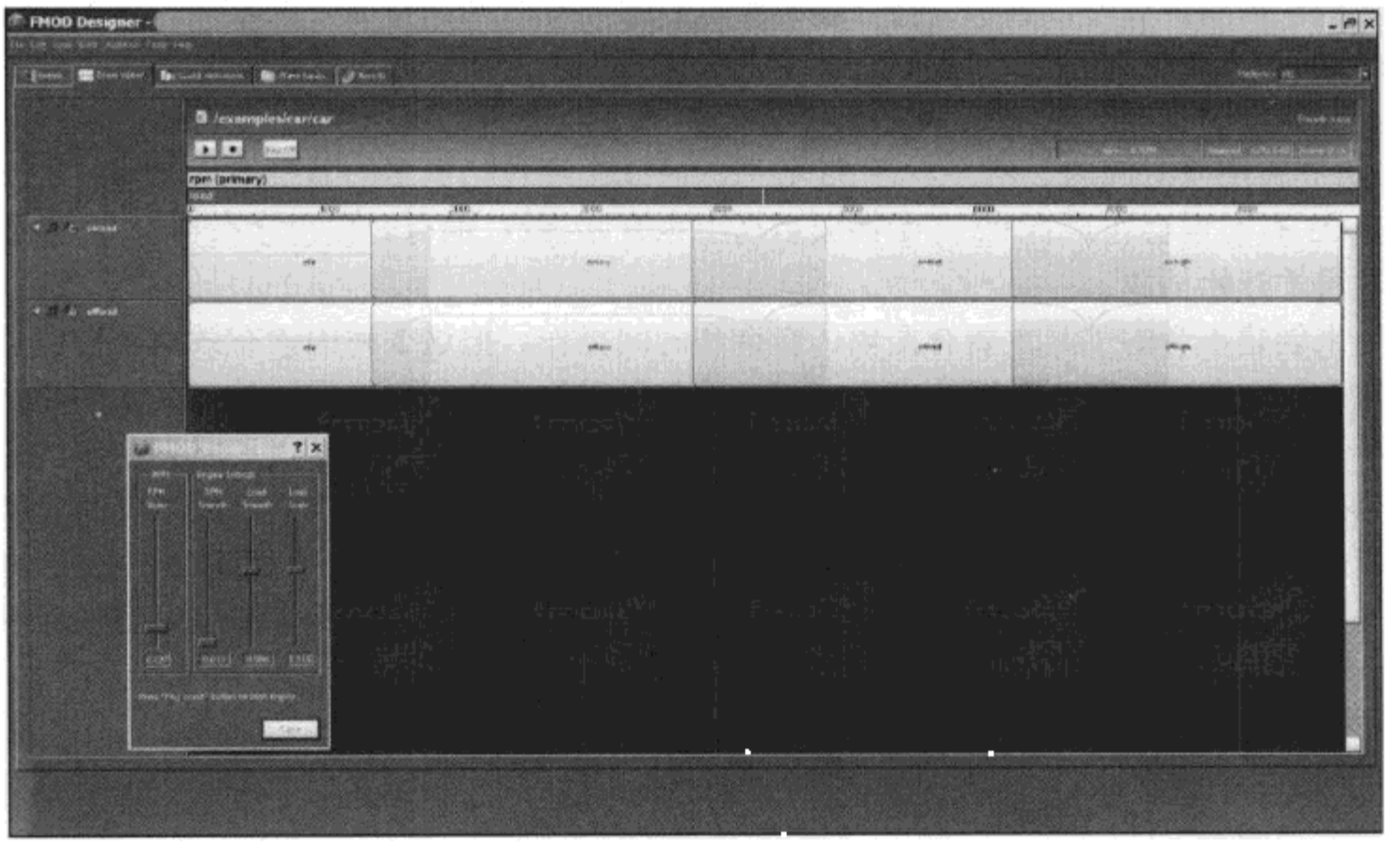


图 4.3.3 FMOD 的声音设计界面

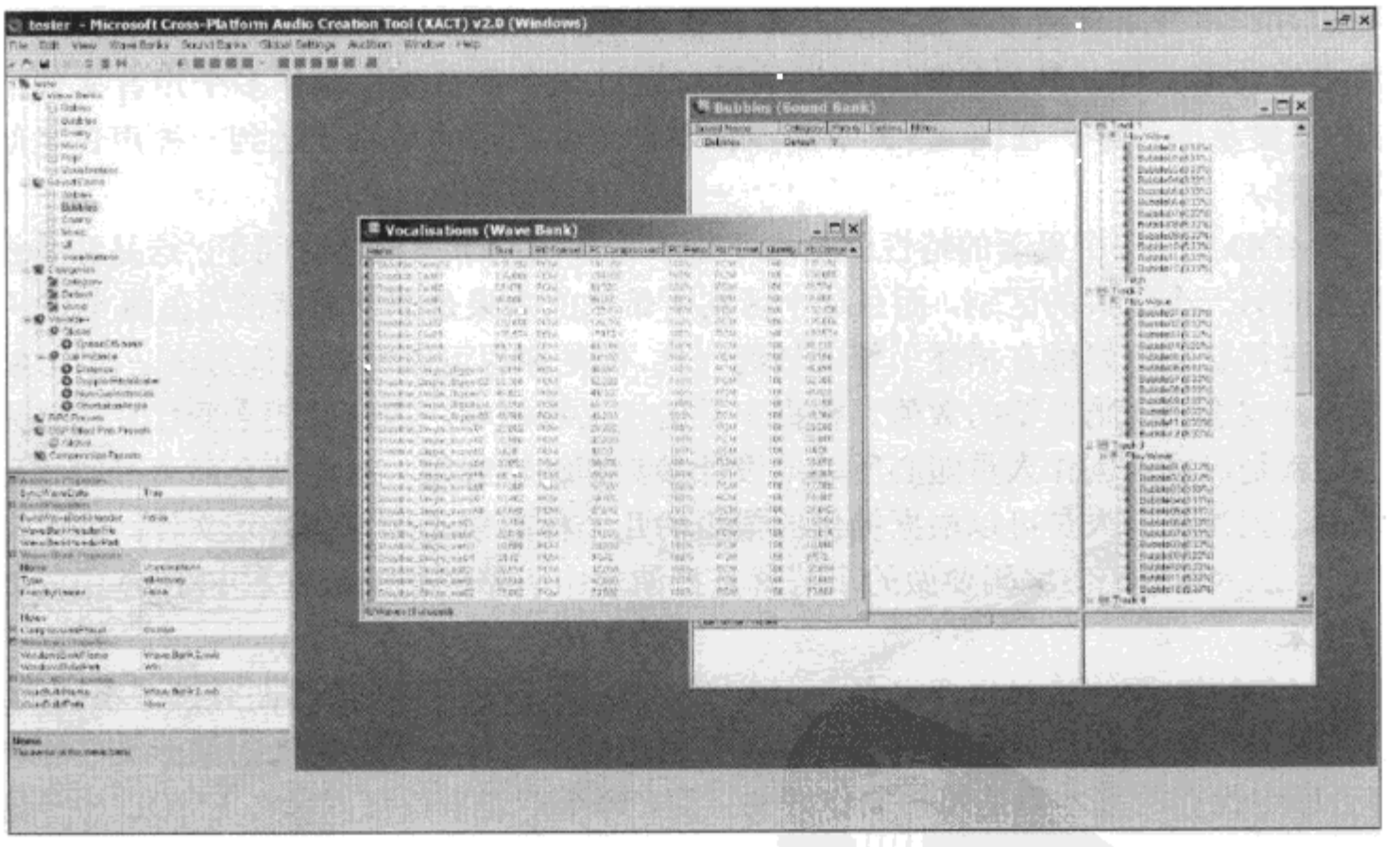


图 4.3.4 微软的 XACT 音频工具

4.3.5 细节管理

除了最简单的声音以外（比如正弦波），所有的声音都是由更小更简单的声音组合而成的。通过把声音分割成更小的模块，可以增加这些小模块对整个声音大环境的用处。例如，汽车车

门“喀喇/砰”的关门声音就是一个典型的例子，仅仅这点素材就可以为形成其他的声音提供足够的帮助，我们甚至只需要在音调的调整上下点工夫。作为最后被分解出来，并且要制作成其他声音的元素（喀喇和砰），不仅让你拥有了两个可以重新组合成其他声音的素材，而且你还能在这个原始声音的基础上对音调和两个声音之间的播放时间做一些细微的调节。

这是一个相对基本的例子，当然，没有重复的汽车关门声音不会让你得到任何奖项，但是，在你考虑如何设计声音并得到更强的现实感时，这个绝对会有帮助。试着去多开关几次汽车的门，仔细倾听一下每次声音的不同。把两个声音分开也没什么花费，也不会显著地增加内存消耗，把拆分好的声音再组合成新的文件也是同样的长度。

但是这种方法会明显增加你需要处理的文件数量，因为该方法的结果就是增加资源。如果不算其他，你的头文件或者存放资源列表的地方会显著地增加。这些改变在第三代游戏机上体现得很小，基本可以被忽略。但是好处就是可以有更多动态的音频环境供应用，在这样的情况下，多产生一些文件显然已经不是需要争论的地方了。总之，这就是我们的工作。

4.3.6　为什么我们要再做一次

这个系统的基本目标就是把每个声音在游戏中渲染一遍，在渲染中避免简单的重复，并建立一个动态有效的音频环境。这个系统实施这些过程的时间可能会更长一些，尤其是设计人员是否能够掌握这套系统，把音频环境的复杂度解析出来。很明显，在一个很细微的地方花费很多时间是不值得的，但是设计人员可以在创建声音时自由选择细节程度。

相对于普通重复脚步的声音，随机化简单的脚步需要更多的时间。因为这样做需要先通过分离脚步的压力和地心引力让脚底产生的咯吱声，然后再根据地表的不同替换地心引力的声音。如果加上音调和音量的随机化，产生的声音听上去可能会更真实一些。不过别以为所有的玩家都会注意到这点，这也是好的声音设计经常让人叹息的地方。

设计人员通过选择音轨资料，然后设置那些在实时播放时起到控制作用的参数来创建声音。但由于声音多变的自然属性，设计人员在制作完以后的试听也是非常重要的，这样做可以避免出现偏离预期的结果。在更多的声音加入到音频环境的情况下，它们之间需要取得一个平衡，那么经常的调试将会成为一种必要的手段。这种方法最好的手段就是在每段声音加入到游戏中的时候，都要做到能够被制作工具中给出的参数所调整。

这意味着音频环境的大量调整也就是在调整一个数据文件，这样做就不需要重新编译整个游戏引擎。作为结果，声音部门的工作就不再像以前那样需要代码组的大量协助了，音频的平衡和变换变得更简单和常见。这种方法对于在线的内容也很有帮助，因为游戏更新所需要的新声音就不再需要全部大量地下载了。设计人员可以使用玩家们已经安装了的资源来创建新的声音，所做的只是制作一些新的定义而已。一个 MMORPG 的更新包中可能包含上百个新的声音，我们可以只是做一个几百 KB 大小的新定义包，重用原先的资源就可以了。

4.3.7　更进一步

本精粹关注于大多数基本的、用来制作和操作声音的工具，例如，时间、音调、振幅以

及它们一些最基本的用法。目前可用的软件工具都会提供更多、更高级的功能，例如，滤波器、音效和执行工具等。更重要的是，这些工具可以允许你建立无法想象的音频环境。一系列的音乐主题曲或者甚至是单个的音符事件，能够根据预先定义的参数实时组合并在游戏中播放，来表现出游戏对玩家行动的反应或者游戏内部的交互。如果想用一个升音和降音模式来表现多菲鸭在楼梯上跑上跑下的声音，你可以做到。如果玩家在楼梯的中间停下或者从中间跳上和跳下，你的方法也不会出现什么问题。音乐可以适当地做出反应。

虽然这种方法还存在一定的局限性，但是它能给自由和创意带来的大多数好处就是你的创造力没有被局限住。一个游戏音乐方面的评分可能因为玩家感官方面的体验而逐步上升，或者无论在游戏中的哪个地方，游戏对玩家的行为产生了互动，而方法是使用了一个独特的音频来表现，这些都会让这个游戏因为音频而更受大众关注。那就把你的想法加进来，让它成为现实吧。

虽然我推荐这种方法来实时渲染和创建这些声音，但是这并不能减少或者取代音乐设计人员的工作。事实上这只会让他们的工作变得更艰巨，因为这种方法需要音乐设计人员不仅仅像传统的工作那样只是简单地使用一些音乐库。这种方法很快就会暴露设计人员是否技能不足或者缺乏想象力。相反地，一个伟大的设计人员可以用这个系统创建出游戏业界中最好的音乐。这种方法至少在你创建第一个可以利用的项目时会花费比较长的时间和精力，但是一旦开发人员度过了开始的那段学习曲线，这种方法就会变得非常柔性。这种方法对最后一分钟的音乐片段做修改和替换的手段要比传统游戏音乐设计方法简单得多。

4.3.8 结论

在过去的 5 年里，游戏产品标准在迅猛提高。作为开发的工作室来说，最好能够弄明白好的工具以及产品流程的重要性，应该持续保持质量上的提升。以前，游戏音乐经常被忽视，或者得到的重视程度远远不够。新的中间件软件和产品工具的开发（如 XACT），允许音频内容制作者们采用全新的方法来做内容设计和创作。一旦设计人员放弃了传统的音乐创作方法，这些新的方法就可以允许他们创作出无法想象的多变作品。这种以前从来没有的创作音频环境的能力不仅适用于天才音频制作小组，而且可以给玩家提供娱乐的内容，这种娱乐并不是其他媒体可以提供的。

4.4　实时音频效果的运用

Ken Noland

本精粹的目的是从一个高层次的视角指出一些音频处理的基本原理，其中包括这些年来我为视频游戏制作音频引擎时所得到的经验和技巧。一些经验和技巧理解起来很直接，而其他一些则需要时间来思考和琢磨以后才能够理解。

如果在网络上搜索，你很快就能找到如何有效地建立一个图形渲染流水线或人工智能的框架。但是，当你要搜索如何建立你自己的声音系统时，你得到的会是一大堆文档，在我看来，就是介绍 API 使用的方法和一些很普通的理论性内容，这些东西绝对不会把那些关键的例子和你讲清楚，它们只是倾向于说明音频编程。最近这种情况正在改变，重心越来越多地从音频编程转移到从数字信号分析的角度来说明问题。

本精粹也不会着重说明 API 的用法，虽然我会在一些很有意思的特性中介绍一些 API，但是本精粹的重心还是放在创建一个音频系统所需要的一般规则上。在阅读本精粹以前要提醒大家一下：在本精粹中，我会讲到一些比较高深的内容，这些内容需要大家阅读其他的资料来加深理解。

在开始之前，先向大家介绍一个非常基本的概念。声音是通过采样的不同来识别的，记住这一点。如果你看到一个波形，你就会知道它几乎全部是由振荡的数值组成的，并且这些数值一直在改变。这些改变标识了伴随着时间的频率。如果信号是平的，那就说明没有频率了。如果一个信号的改变速度很快，那说明频率很高。

这是你必须知道的一个很重要的概念。记住，这些值是一直在振荡的，如果你把一个信号值从很高的地方立即调整到某处，那说明你需要用零来清除缓存，并把一个正在走向峰值的信号值从中间截断了，而这样做的结果就是让这样的信号听起来像是一个爆破音。下一部分将会向大家介绍一种更准确的方法来清除声音缓存。

目前有两种最主要的 API——DirectSound（Win32 平台和 Vista 操作系统）和 OpenAL（在绝大多数平台上都能用，包括 Win32、Vista、Linux 和大多数的游戏机）。两种 API 都不错，并且支持大量的各种各样的格式和效果。我也说不准究竟哪个更好，这基本上取决于你到底工作在什么平台上。

不过这两种 API 也有各自的优缺点。因为 DirectSound 和 OpenAL 支持的驱动程序不同，我建议在这两个 API 的基础上写一个音频系统来抽象出一个层面给最终用户使用，这样最终用户也就不用考虑不同的 API 对不同的驱动程序的识别了。同时，我也建议你的音频系统要针对这两种 API

提供一个软件处理的选项，用这种方法就可以处理那些因为驱动程序而产生的问题了。

OpenAL 和 DirectSound 有两个区别很大的设计方法论，就好比 OpenGL 和 DirectX 在图形设计方面的不同。如果曾经使用过 OpenGL，那么用起 OpenAL 来就很自然。如果你基本用的是 DirectX，那么 DirectSound 对你来说就会很顺手。

4.4.1 声音系统的概览

当面对一个高级层面的声音系统时，你必须了解 4 个概念——主缓存、听众、声音和任何一种运用于声音或者听众的音效。

主缓存

主缓存是在播放以前 PCM 采样最后停留的地方。在大多数的声音系统中，你是不会直接去填充主缓存的，不过从听众的角度来说，你就会碰到主缓存这个概念。对于主缓存，你需要关注的唯一一件事情就是它在每一帧上的耗费。

听众

听众是在 3D 空间中的一个特殊对象，它获取所有传过来的声音，并且应用任意的特殊变换和特效，例如摇摆（panning）和衰减（falloff），还有类似于多普勒频移和相对于头的移动延时（Head Relative Transfer Delay）的高级滤波器。

你应该总是这么认为，提供给你的任何 API 都只有一个听众。通常来说，这个不会成为什么问题，但是，如果我们制作的游戏具有多个视口或者监视器，那么只提供一个听众就会给我们带来一个小小的挑战。不过要解决这个问题也很简单，就是把所有的声音都传到听众那里，然后把一些内容（如速度）记录在声音属性中，这样就可以正确地计算多普勒频移效果了。但是当不同的听众在这些声音上应用不同的效果时，事情就会变得有些复杂，稍候我将探讨这些效果。

声音源

声音源就是标准的单声道信号，这些信号从世界的各个角落传来。声音源都有类似位置、衰减和速度等标准属性，听众和效果会使用这些属性去处理声音。

在任何声音系统中，你应该区别对待声音源和实际的声音数据。声音源包含声音数据的一些参考值和这个声音源的位置、方向，还有在实际声音数据中的当前播放位置。实际的声音数据只包含一个装有 PCM 数据的容器和其他一些和音频编辑器相关的属性，比如衰减参考值、实例的最大数量和这些实例上所采用的普通效果。

声音效果

声音源中还包含效果的内容，其中一部分效果是从声音数据中继承的，而另外一部分则是由它在这个世界中所在的位置提供的。你可以把这两种效果叠加起来以满足需求，这不失为一种好方法。

把这些概念结合起来，你就能够了解到，主缓存需要从听众那里获取数据，然后听众就会做出举动来决定播放哪种声音，并向声音源请求采样数据。声音源接收到听众的请求后，先会

释放音效栈，然后把正确的数据发送给听众。接着，听众就会在收到的声音效果上运行一个数字信号峰值限制器，把自己的效果堆栈也释放掉，最后就把内容重现到最后的缓存中。

在整个声音系统例子中，有一件我们需要注意的事情，即系统使用了一个模型-视图-控制器（Model-View-Controller）架构。数据是在声音数据（模型）里包裹好的，并且收到声音源（控制器）的申请，然后在数据上应用各种声音效果（更多的控制器），接着当听众提出需求后，就会将数据输出到主缓存中（视图）。

4.4.2　声音缓存

在几乎所有的机器上，你都会受到硬件所能够播放的音频数量的限制，甚至在软件处理模式下，你也应该将可用声音缓存的数目限制在你的性能目标的范围之内。在我写这篇文章的时候，一线用户一般采用的声卡可以最多同时播放 128 个声音。把这个记在心里，后面会用得到。

不过如果使用 Vista，那就不要考虑这个了，因为 Vista 会强制你在 DirectSound 下使用软件处理模式。如果想在 Vista 操作系统中使用硬件处理模式，唯一的选择就是 OpenAL。

在大多数情况下，你会希望能够载入足够的声音采样到声音缓存中去，这样你就可以连续播放声音了，虽然说这样做会引起帧速率的大幅度下降。我的标准做法就是建立一个足够容纳频率在 44 100kHz 的一秒钟声音数据的内存空间作为缓存。

这里我想起一件事情，就是以开始、停止的形式播放声音缓存比一直播放下去的过程要消耗更多的时间和资源（在超出播放的时间范围时，请确认清空声音缓存，否则就会听到不想要的声音）。但是这样做是有原因的，比如，在刚才讲到的例子里，当你拥有 128 个声音缓存，你可以先开始播放其中的 8 个。当 8 个声音缓存被声音数据占据以后，就要去取更多的缓存。一旦这个声音缓存被播放完了，并且有一段时间没有被访问到，就应该停止这个声音缓存。这样的平衡做法并不是必需的，但是我曾经发现，当我要正确地一个一个地播放一些短促的声音时，这样的做法会很有帮助。

一旦得到一个播放声音的请求后，就要在目前开始写的内存位置上去申请尽量多的声音缓存，同时你也要记住，你可能已经开始播放这段缓存了。你可以通过记录先前播放的游标位置来提升目前的播放质量。这里有件事情值得注意，有时候驱动程序可能会提供给你一个错误的位置信息。有时候提供给你的位置只是相差了几个采样而已，但是其他时候可能会是比较大的误差。为了补偿这个问题，在你申请缓存来播放声音时，只申请缓存一半大小的空间。也就是说，我的缓存是一秒钟的，实际上这一秒钟的缓存只包含半秒的数据。

所以说，让我们假设你得到了一个 44 100 采样大小的缓存，写的位置是在 44 000 的地方，而你播放的位置恰好超过了最后一帧 150 个采样。用刚才提出的办法，你可以先获取 22 050 大小的采样（一半缓存大小）。现在你从声音源那里得到了那些采样，你需要在目前写的位置填充 100 个采样，写完这 100 个采样以后，位置正好是缓存结束的尾部。剩下的 21 950 个采样就从缓存 0 的开始位置写下去，这种做法被简单地称为环绕缓存。

在你下一帧更新时，你要做的事情就是在内存中最后写的位置那里填充缓存，填充的数量就是已经播放的采样数量。在上一个例子中，你要做的就是在 21 950 这个位置继续写 150 个采样。

作为一个安全的预防措施，你也需要把先前播放的采样清空。当这么做的时候，可能需要清理到播放游标当前位置的后面 3～4 帧，因为我们刚才说过，位置有可能存在偏移。另外一个安全的预防措施就是在最后写的位置设置一个回调函数，这个回调函数会把前面的内容清空。当游标播放到最后的位置时，就触发了这个回调函数，然后回调函数会先把整个缓存里的声音渐渐淡出直至消失。因为声音播放经常会是在另外一个线程中进行的，所以在所有情况下它都应该可以工作。采用这样的方法以后，你就不会让那些声音在主更新线程锁定的事件中重复播放了。

4.4.3 分级缓存

采用先前提到的声卡作为例子，我打算这么认为，你在任何时间可以播放的声音总共可以有 128 个。问题是在你的 3D 游戏世界中会有成千上万的声音从不同的方向传过来。在这里就需要实现一个叫做分级缓存的特殊缓存类型了。

分级缓存是一个很简单的概念。你用你的算法生成等级，然后可以根据等级来申请缓存。如果所有的缓存都满了，并且你给的等级也超过了另外一个已经播放的声音缓存的等级，那么最低等级的缓存上的声音就会被清理掉，新的声音就会复制上去并进行播放。

等级计算的方法有许多种，其中最普通的方法就是确定衰落（attenuation）值（距离、衰减和音量），然后乘以一个由音频设计人员提供的数值。在大多数情况下，这样的做法是行得通的，但不是所有的情况。最好的方法就是把所有的声音属性都考虑进去，比如，距离、衰减、效果和其他与声音有关的内容，通过这样的方法，你就能得到关于声音等级的一个非常清楚的概念。对重复的回声效果产生的声音定义一个高优先级，而对更显著的声音对应一个较低的优先级是行不通的。

同样值得提起的事情就是，音频设计人员喜欢指定一个特别的声音或者声音种类能有多少个实例被播放。例如，如果你待在一个房间里，而房间里满是机枪扫射的声音，那么最好只播放 10 个左右这种机枪的声音。当你建立自己的声音等级算法时，最好把这个例子记在心里。以前我做过一件事情，就是允许声音数据根据自己独特的前后关系来确定自己的等级，这里我用到了一个抽象的简单函数，只是把传给声音的那些参数做一些处理，例如，相对于监听者和一般世界数据的声音位置。

对缓存等级的划分的确存在着一些难题，尤其是在音频处理上面。最主要的难题就是你不可能在停止一个声音后，马上接着去播放另外一个声音。记得前面我提到的情况吗，这样做会让采样听起来有所不同。如果你突然停止播放一个声音，你会听到一些爆破音。如果你想这么做，比较好的方法就是把前一段声音逐渐淡化出去，然后接着播放另外一段声音。

如果前一段声音还附带音效，问题就会变得更复杂。因为要考虑到驱动程序对声音缓存上采取的音效控制方法，你不能简单地采取线性变化，必须等到前一段声音结束了淡出效果，然后才能关闭音效。在这种情况下，当前一个声音的音量降到零时，你才能开始把新的声音复制到缓存上，或者再附加上音效。这里有件事情你还需要注意，就是你必须等到播放游标到达了音效变换的位置以后才能够做音效的转换，注意是播放游标，而不是写入游标。

记得前面你复制半秒的声音采样到一秒的缓存上的事情吗？你想做的事情就是能够让上一段声音可以立即过渡到本段声音。记住，当你把数据传送到声音缓存上以后，接下来的事情

都交给驱动程序去执行了，我可不想打赌那段声音在被执行以后是不是还会在那里。为了确保这段声音还是在缓存里，你能够做的就是一直把这些采样复制到那段缓存上，只有这样做了，你才能够回头再去播放那段声音，让它在目前写的位置上播放，并且做出淡出的效果。

淡化采样的数量是不定的，但是我通常会采用大约 5ms 的长度，或者粗略地认为是 44 100kHz 频率下的 220 个采样。你可以把它作为一个声音数据的属性，这样设计人员就可以进行调整了。

4.4.4　效果和滤波器

效果对象应该是通过与你的声音系统 API 对应的抽象工厂函数建立起来的，这样就使得硬件处理成为可能，然后这些效果可以附加在声音源或者听众上，当得到需求时就可以被释放了。

效果和滤波器是两回事。一个效果可以包含多个滤波器，或者简单地生成声音数据，或者可能包含了一个对硬件加速特性的包装器。在任何情况下，效果是一种中间设施，这个设施存在于声音源和被效果调用的滤波器之间。当你设计滤波器时一定要记住，把滤波器设计得越通用越好。在设计中要把任何执行时需要注意到的细节都隐藏在效果对象里，这样做可以让你在抽象新的效果时更快一些。讲得简单一点就是：效果应该针对特例去实现，而过滤器需要尽可能地考虑通用性。

需要注意的两种类型的滤波器如下。

- 无限脉冲响应（IIR）滤波器，这个滤波器在声音采样上是递归调用的。
- 有限脉冲响应（FIR）滤波器，这个滤波器只是在不考虑优先输出的情况下用某种方法来处理声音采样的变形。

在设计效果的时候，你需要区别对待这两种类型的滤波器。

在滤波器里，有个概念叫干/湿混合。湿采样是先前变过形的，干采样是从音轨上获得的还没有经过任何变形的采样。你需要定义一个分辨属性来区别这两种采样，让你的效果可以采取各种干湿的混合比例。

为了把事情弄得更复杂一些，有很多方法都可以用来变换采样数据。一种最常用的方法就是快速傅里叶变换（FFT）。这种计算方法虽然很有用，也很容易被运用到各种效果上去，但很耗费时间和计算量，不过有许多人已经在提高速度上做了许多研究。采取这种方法时，一定要慎之又慎，并且对任何你能够从它那里得到的数据都要做缓存。这样做意味着你可以把效果对象中的声音变换到频域，并且在变换后的频域中运行所有的滤波器（在此之前请确认滤波器能够使用频域数据）。当所有的滤波器被处理以后，再由效果本身将声音数据变换回采样空间。当然，你要确认效果确实调用了这个变换过程。

有限脉冲响应滤波器是最容易应付的，因为你所要做的事情就是把数据（干混合）发给它，它就可以把结果分离开来。无限脉冲响应滤波器稍微复杂一些，因为它依赖于前一次生成的结果（湿混合）。最简单的方法就是在一个效果里采用不同的缓存设置来记录滤波器输出的内容（湿混合缓存）。当效果被创建的时候，这段缓存的大小是要被特别指定的，这样做可以设置延迟线。在一些效果中，延迟线可以用效果的输入来设置，例如反馈延迟等，这种方法也可以被转换成缓存的大小。你也可以不这样做，而显式地设置缓存的大小，但这样会让你的频域窗口被限制得死死的。

无限脉冲响应和有限脉冲响应滤波器也可以用有向图来安排，允许滤波器之间以环路的

方式相互引用，一直运行到延迟线指定的程度。这种滤波器设计风格是在《游戏编程精粹 5》中的“基于反馈延迟网络的快速环境回声”[Schüler05]里提到的风格。使用这些类型的滤波器是很快捷的，因为你可以很快地设计新的效果，也可以很快地模拟周围世界听得见的那些特殊的效果。

下面讲讲信号计时——就是那些通过延长声音播放时间产生的效果，和原来的声音组合在一起，以产生一种效果，如合唱。我已经向大家解释过一个系统，在这个系统中，你需要采样数据，然后采样通过解析效果堆栈，得到最后的数据。通过这个过程，一段声音就被处理好了，当然那时候还没算上通过听众的那些采样。这里有个容易误解的地方需要说明一下，在一个循环声音源上通过一个无限脉冲响应滤波器简单地等待返回无采样的请求，将会使这个声音无法循环播放下去。因此，你不得不放置一些分隔开的标志位来告诉声音源是不是要循环，这样，当运用在这段声音上的效果播放到这个位置时，才会知道要从头开始循环播放该效果。

4.4.5 压缩和流

目前有许多音频压缩格式可用，每种格式都有自己的特点，以满足某种特别的需要。一些格式，例如 ADPCM，关注的是效率和快速解码；其他格式，例如，MP3 和 OGG Vorbis，关注的是高压缩比率，在压缩比率很高的同时还能保持原来的音质。这里对这 3 种压缩方法进行一个快速的比较。

- ADPCM 是这 3 种格式中最简单的格式。这种格式采用一种简单的预算法来产生音频数据块的中间变数。这些中间变数用 4 位比特数值来保存，这种方法就可以让解压缩算法非常简单。这个过程只需要两个表的查找和解码 4 比特的中间变数，另外还有两个相乘操作和一个相加操作，这只需要占用极少的 CPU 资源，而带来压缩上最高的回报。但是，这种格式的压缩比率不会超过 4:1，和其他两种格式相比较而言，在低采样的数字信号恢复方面并不出色。
- MP3 是一种常见的格式，众所周知，它在各个平台上都能够被使用。MP3 使用的是帧，和 ADPCM 里使用块很相像。这些帧中包含了声音上补偿的信息，这些补偿是声音信号在频域里变形所带来的，它们被分解量化成查找表[MP307a]、[MP307b]。MP3 允许多种编码方法，例如，可变比特率和 ID3 标签。
- OGG Vorbis 采用的是经过修改的离散正弦余弦变换，用来从信号空间到频域的转换，其实这个和 MP3 差不多，然后做取整操作。后面它就量化扩散编码，将得到的变值存储到查找表中 [OGG07]。这种编码允许可变比特率的有损压缩，并对于快速解压缩经过特别的调试，不过它的效率还是没有 ADPCM 那样高。

你可能在疑惑，我为什么如此详细地介绍这 3 种压缩方法呢？市面上有许多的库可以帮你来做这些格式的转换（[MP307a]、[MP307b]、[OGG07]），还有考虑到效率相关的数据，真没必要把这些格式说得那么详细。但是我还是想说，这里面还是有些东西需要记住的。OGG 和 MP3 在频域中保存它们的信息，这意味着我前面介绍的昂贵的 FFT 已经出现了。

这也就是说，使用市面上已经有的库来处理这些格式，可以解压缩频率数据，并且使用

这些信息来运行自己的频域效果，然后把这些信息转换到信号空间来做最后的重现。

把每种格式都介绍得很详细的另外一个原因就是：你会发现每种压缩方式中都提到“帧”或者“块”。使用这些信息，你可以建立一个单独的用来缓存 PCM 解码采样或频率解码数据的分级缓存机制。当你正在从硬盘上读取流的时候，就意味着你可以用已经解码好的方式来缓存一定量的帧和块，而不用把整个解码的文件都保存好。对于音乐来说，这是非常重要的。在读取音乐文件的时候，你总是想读得越多越好，然后缓存那些解码数据，但是你不可能单单为了你的背景音乐而花费 300MB 的内存空间吧。通过基于一帧接一帧的解码方式，你可以随心所欲地限制内存空间的分配；而且通过利用分级缓存机制（如果不谈那些淡出的采样），你就可以得到一个流式地处理磁盘文件的有效机制。

4.4.6　结论

从散乱的线条开始整理，直到创建出一整套音频系统，初看的确会让人感觉到这是一项难以完成的工作，会让人觉得底气不足。但如果你是个程序员，利用这里提到的方法和概念，你应该可以迅速地启动起来。除了这里讲的内容以外，还有很多其他的内容需要你去学习，还有许多资源需要你去了解，这样你才能够更好地开始音频方面的工作（参考文献中给出了我先前提到的一些资料）。另外我建议，经常去访问一些相关的论坛和新闻组，那里会有目前最好的信息。

音频编程是一个挑战，但是从挑战中你也会赢得荣誉。等你开发好自己的声音系统后，就可以根据你的游戏需求和性能要求，利用新的知识来增加新的实现，进而满足策划的要求，最后扩展游戏的效果，让你制作的游戏达到一个完全不同的可玩性很强的境界。然后，一旦这种截然不同的特性让玩家发现了，玩家就能更沉浸在你的游戏里了。所以在我看来，音频就是游戏编程里一个很重要的领域。

4.4.7　参考文献

[dsnd07] Microsoft “DirectSound.”

[MP307a] Underbit Technologies. “MAD: MPEG Audio Decoder.”

[MP307b] Mike Cheng. “The LAME Project.”

[OGG07] Xiph.Org “Vorbis audio compression.”

[openal07] Creative. “OpenAL: A Free (LGPL-ed) and Open Source, Cross-Platform Audio Library Used for 3D and 2D Sound.”

[Schüler05] Schüler, Christian. “Fast Environmental Reverb Based on Feedback Delay Networks,” Game Programming Gems 5, Charles River Media, 2005.

4.5 上下文驱动，层叠混合

Robert Sparks
sparks.robert@gmail.com

在我们的游戏界里，声音的技术质量正在接近电影界。次世代的游戏机就摆在面前。许多游戏已经支持杜比音效和 DTS 音效。这两个标准都使用很高的采样率。实际上，这两个标准里的声音数量是无限制的；在此基础上，这两个标准还采用了无损压缩算法。所以说，电影对于游戏来说，目前还保持的强大优势就是它对最终产品的控制是全面的。

想想声音混合的过程。一部电影在做混音的时候可以对每个声音效果做完全的控制。每个场景都可以基于某种目的来进行混合，并且传递给观众一种富有感情色彩的体验。游戏中能够控制的东西就没有那么多，比如最常见的，我们不能在场景更换时做那么多改变。我们依靠的是位置和环境的模拟。

本精粹将向大家介绍的混音系统可以给开发人员带来更多的游戏整体声音的控制能力。类似这样的系统曾经被使用于《疤面煞星：掌握世界》（Scarface: The World Is Yours），还在游戏《天行者》中做过 3 周的最后混音阶段支持。

4.5.1 概述

这个混音系统的原则是游戏参数能够被实时调整。其自身主要关注于将调整体验感的过程组织成一个有效的、基于电影混音的工作流。

这个系统把声音的参数（例如，音量、音调和滤波器设置）描述成了混音板上一行行的推杆和旋钮，每行控制了整组相关的声音（例如音乐、对话或者脚步声）。

这个系统还把游戏里的动作按照逻辑分割成了各个场景。和可以按照时间顺序来定义的电影场景不同的是，游戏场景必须按照玩家的动作来定义。

把每个逻辑场景和一系列的混音参数联系起来，可以精确地控制整体的声音（见图 4.5.1）。这样做还有一个好处是能够让每个场景在一系列混合的过程中进行实时的独立混音。

因为采用了一个场景接一个场景的方法，这个系统所采用的就是上下文驱动方式。后面你会看到各个场景之间会互相覆盖和修改，这样也造就了这个系统有层次的概念。

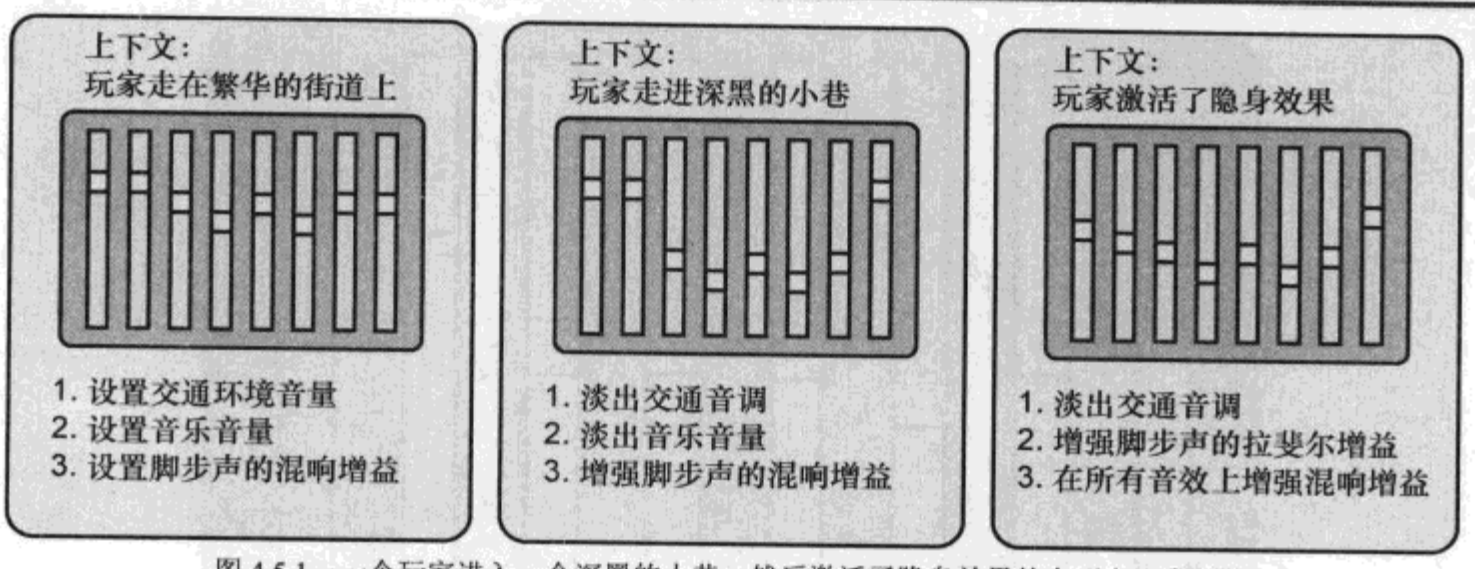


图 4.5.1　一个玩家进入一个深黑的小巷，然后激活了隐身效果的上下文混音例子

4.5.2　实现

下面是对混音系统的逻辑和其中主要类的高层次描述。光盘中有更详细的C++实现。

混音系统

这个混音系统为游戏中的其他系统提供了一个中央混音界面，它管理着部件的生命周期，同时还进行一些计算。

混合分类

混音系统将相关的声音组合到混合分类中。这个系统是针对这些分类进行工作的，而不是某个单独的声音。

这些分类可能是背景音乐、环境音乐、爆炸、玻璃破碎的声音、脚步声或者鸟鸣声。游戏里播放的声音都对应着一个混合分类。

中央混音

混音系统将所有声音的混合（或者调整）参数集中成一个单独的逻辑对象，这就是中央混音。这些参数可以是音量、音调、拉斐尔增益（LFE gain）、辅助效果增益或者位置模拟的相关参数等。中央混音为每个混合分类提供了一系列参数。

从概念上来讲，中央混音就像一个混音板，它把游戏里所有的声音都串联了起来。当声音在游戏中播放时，它们会根据中央混音里对混合分类给予的参数进行相应的调整（见图4.5.2）。

混音快照

声音编辑器基于混音快照上的参数值，在中央混音上进行操作。中央混音的状态是由这些快照来计算得出的（见图 4.5.3）。混音快照就像混音板上的预设或者中央混音上的动态摇杆控制器。

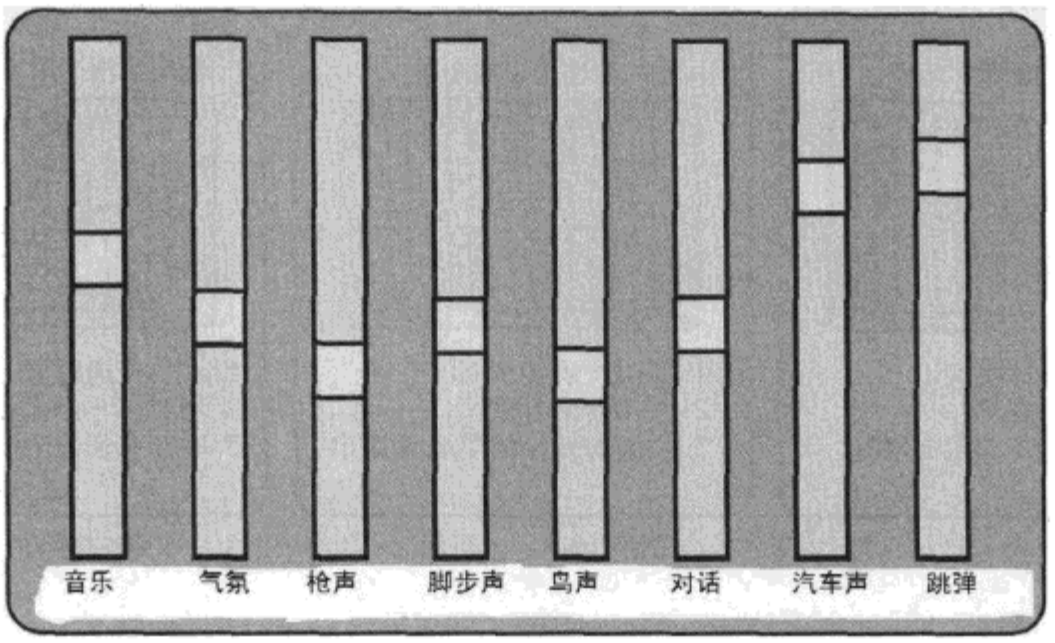


图 4.5.2 中央混音像一个混音板一样为游戏工作，并控制多组声音的播放

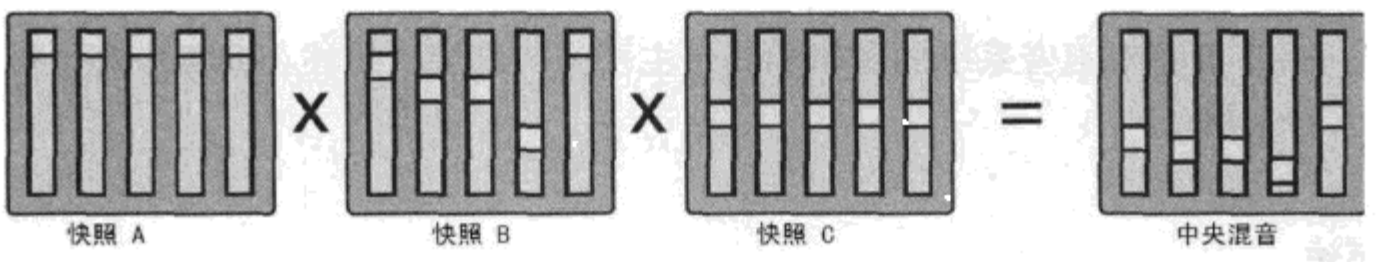


图 4.5.3 混音系统使用声音编辑器提供的混音快照来计算中央混音上的值

声音编辑器为每一个逻辑场景定义一个混音快照。当场景开始的时候，一个混音事件就会随之被触发，把一个相关的混音快照加到中央混音上进行计算。当场景结束的时候，另一个混音事件就会被触发，同时把快照删除。当然，快照会提供淡入/淡出的持续时间，让这些快照在被加入和删除时显得比较柔和。

混音快照给声音编辑器提供了对每个场景里声音的绝对控制。控制粒度取决于场景的数量和混合分类的数量。

场景可能是很普通的，并且贯穿整个游戏。场景在不同的世界位置也可能会有不同。另外，场景也可能是重叠的，还会被其他场景影响。表 4.5.1 给出了一些混音快照和场景的例子。

表 4.5.1 混音快照的例子

快照名称	场景描述	声音描述
on_foot_night	当玩家在晚上走路的时候激发	脚步声 晚上环境的声音 晚上回声的设置和降音的设置
on_foot_day	当玩家在白天走路的时候激发	白天的脚步声和环境声 白天的回声设置和降音设置
in_car	当玩家开车的时候激发	玩家汽车的声音 降低了的环境声 在车里的回声设置 堵车等级上升
interior	当玩家进入一个建筑的时候激发。有可能和 on_foot_day 或者 on_foot_night 一起调用	降低了的户外声音

续表

快照名称	场景描述	声音描述
dialogue_duck	当玩家说话的时候激发。有可能和其他任何一种快照一起调用	着重在对话的清晰度上 降低背景音乐和其他干涉到的声音
invincible	当玩家进入一种特殊的隐身模式时触发。 有可能和其他任何一种快照一起调用	音调降低，提供特殊音效 提高次低音的音量
nis_2	在游戏中播放电影时触发。无互动序列（NIS）被称为 nis_2	所有游戏内的声音都被去掉，除非是 NIS 需要的
pre_mix	一直激发	所有声音里的全局调整

混音层

混音层将被激活的混合快照有效地组织起来。声音编辑器将混合快照分配到各个混音层里。这里有 3 个混音层，每个都代表一种特殊的行为。

- 预混合层包含了一个永远不会改变的快照。这个层允许声音的属性在所有的上下文里被全局改变。
- 基础层永远只包含一个快照。如果有新的快照被激活，那就会覆盖前面一个快照。
- 修改层在某一时刻包含任意数量的快照，进而允许场景层叠。这些快照会影响到其他快照，其中比较标准的会是降低某个特殊声音的音量，应用特定的滤波器或者音调效果。例如，一个修改的快照会迅速降低对话时的背景音乐，或者在关键游戏情节中应用特殊的滤波器。

图 4.5.4 描述了这 3 个混音层。

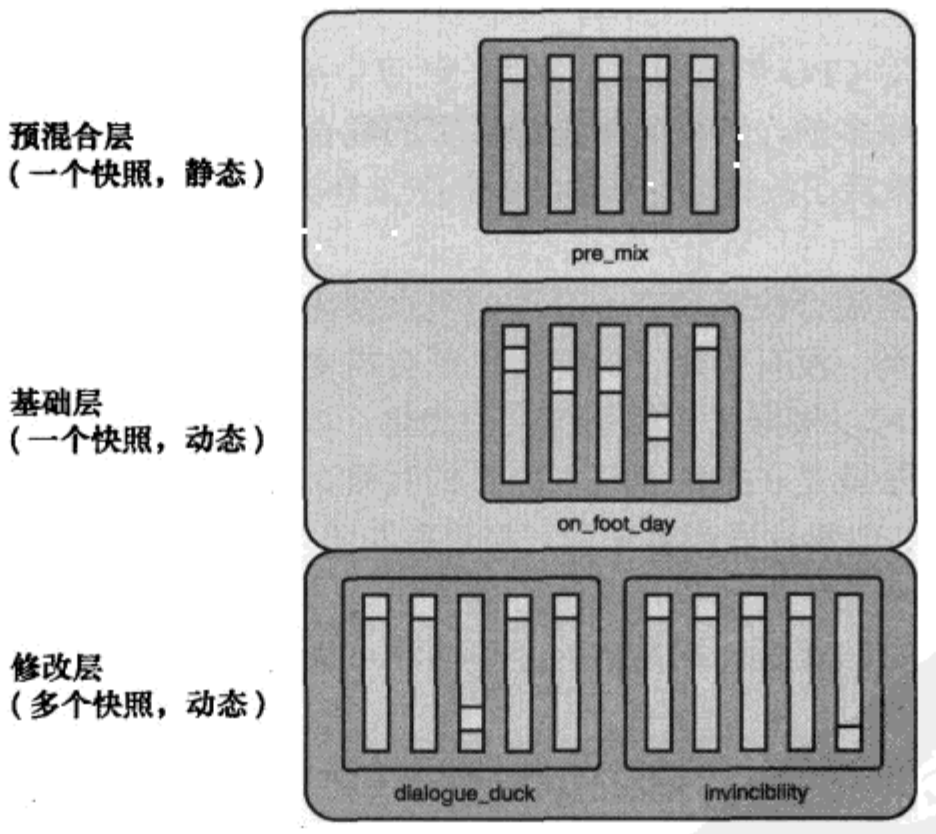


图 4.5.4　这 3 个混音层被用来组织激活的混合快照

4.5.3 扩展实时调整的概念

为了得到一个真正有效的工作流程，一个远程调整程序是很有必要的。只有通过实时调整，音频策划们才能够在听到声音时调整音量等级和一些其他的设置，进而完成对一些问题的修正。

调整程序可以通过一些简单的数字数组来表现调整参数，也能通过图形的方法来表示，如混音板。同时能够显示激活了的混合快照和中央混音的状态会很有用处。这样就能单个选择，并调整那些针对每个场景的混合快照了。

最后产生的工作流程是很精密的。标准地说，这个过程包括了远程传送到本地，或者需要混音的任务，在远程调整程序里选择满意的混合快照，然后在播放时和场景融合。

对于《疤面煞星：掌握世界》来说，我们的团队在调整程序和实际的混音板之间实现了一个 MIDI 的界面。这个界面让非游戏音频界的人员也能够很容易地为我们的项目工作（见图 4.5.5）。

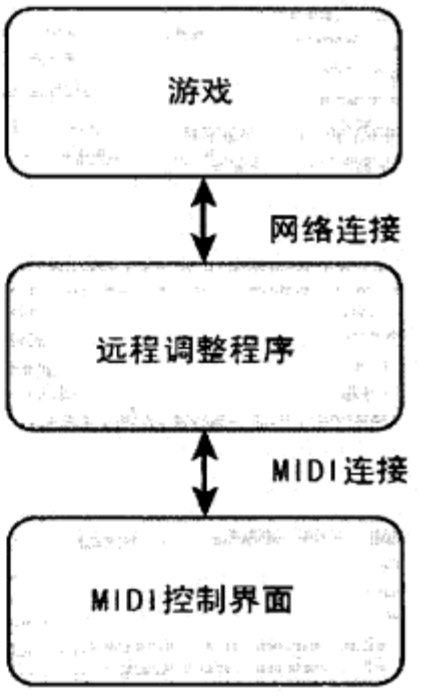


图 4.5.5 通过 MIDI 控制界面来实时混音

4.5.4 效率

我们的混音系统对 CPU 的资源要求很低。计算中央混音也就是它大多数的能量耗费，中央混音的工作可是有很多的，其中包括把激活了的混合快照的参数都绑定起来。标准地说就是，里面会有 4 个被激活了的混合快照和 20 个声音分类，每个分类有 4 个参数。参数混合经常用到的是加法和乘法。

内存的需求是根据混合快照和声音分类的多少得出来的。大型游戏可能需要几百个混合快照和几十个声音分类。没有优化过的混合快照会需要 512 个字节，那么 200 个快照就需要 100KB 的内存。大家可以根据工作的系统来做优化，以减少内存的使用。

最有效的优化方法就是在把快照载入内存的时候，每次只是载入那些需要的快照（例如，对于一个任务、区域、电影片段或者角色，将快照和美术资源绑定在一起）。这会需要流水线的工作，并且还需要和其他内容载入系统协调工作。

还有一个优化的方法就是，把参数的类型从浮点类型转换成短整型，这样做可以把混合快照的大小减半。

把这些优化方法整合起来，一个 512 大小的混合快照可以变成 256 字节；花费本来 10 个快照需要占用的内存空间，现在就可以满足 200 个快照需要的空间了。因此，本来 100KB 的内存占用就减少到了 2.5KB。

4.5.5　例子程序

本书光盘中包含了本文的一个示例程序。这个程序是一个小游戏，在里面有个小的混音场景。单击按钮可以触发声音效果和混音事件。使用混音板和相关的控制器可以选择和调整混合快照，以及体验上下文驱动、层叠混音。

4.5.6　结论

本精粹讨论了一个很强大的混音方法，并验证了它的可操作性以及使用效率。

当我们讨论到需要提交高质量的声音时，工作流程就成为了必不可少的课题。定义完善、直观、易操作的流程才能够让创意和精雕细琢成为可能。我们可以从电影业界借鉴到已经建立好的有效的过程。就看我们的技术人员在游戏界里如何关注这些音频制作的流程，然后把它们建立起来。

第 5 章

图形学

简　介

Timothy E. Roden，Angelo State University
troden@angelo.edu

在三维计算机游戏的初期，开发商普遍关注的是保持低多边形数和降低场景的复杂程度。在渲染和动画方面，有固定功能管线的图形引擎只允许有限的创作空间。现在，事情有了惊人的变化。这版《游戏编程精粹》的“图形”部分提出了各种各样的文章，涉及了许多不同的主题，如内容制作、动画和渲染。

英特尔公司的 Jeremy Hayes 推广了 Jason Shankel 的工作，阐述了一个用粒子沉积（Particle Deposition）的过程地形生成的高级方法。新技术描述了火山的放置，山脉、沙丘和悬垂地形。这些新方法增加了更多的控制，使关卡设计师可以更好地定义重要地形特征的放置。由于制作有趣的、有用的地形不仅仅和几何形状相关，因此，另一个精粹探索了地形的纹理映射。Antonio Seoane、Javier Taibo、Luis Hernández 和 Alberto Jaspe 提出了一个映射超大纹理到户外地形的方法。Ben Garney 提供了此想法的一种实现方法，及在 SM 1.0 级图形卡上使用该技术的一些提示。

“图形”部分特别提供了关于渲染的出色精粹。10Tacle 工作室的 Joris Mans 和 Dmitry Andreev 描述了一种先进的贴花系统（Decal System），能在贴花下面适当地融合凹凸和漫反射贴图，从而消除贴花常有的“在上面”的外观。Tony Barrera、Anders Hast 和 Ewert Bengtsson 的精粹陈述了一个对粗糙材料实现漫反射光照的实时渲染系统。Chris Lomont 全面地概述了一个高效的细分表面。在卡通风格的植物和毛皮效果的渲染中，Joshua Doss 演示了如何使用嫁接贴图（Graftal Imposters）。

在这一版精粹系列中，我们比以往更着重描述了动画部分的技术。索尼娱乐美国分部的 Bill Budge 解释了在骨骼动画序列中处理累积误差的技术。Vitor Fernando Pamplona、Manuel M. Oliveria 和 Luciana Porcher Nedel 描述了一种动画浮雕替代（relief impostors）的方法。最后，我也贡献了一个精粹，那就是使用一个免费语音字典，过程地生成关于人类模型的口型同步数据。

5.1 先进的粒子沉积

Jeremy Hayes，英特尔公司
armyofzin@gmail.com

粒子沉积是一个过程地形生成技术。迄今为止，它的应用局限在创建火山山脉的拓扑里。然而，粒子沉积的美在于它的多功能性。本精粹将介绍关于粒子沉积的一些允许创建新型地形拓扑和改进火山山脉的优势。这些粒子沉积的优势提供了关卡设计师预览及完善地形特征的位置和大小的功能，从而使他们可以从艺术的角度出发来进行改善和控制。

5.1.1 为什么使用粒子

地球的表层被称为大陆地壳。因为它的密度低于地幔层，所以大陆地壳漂浮在地幔的上面。在很长一段时间内，大陆地壳表现得像一个延性固体（如热蜡）[Grotzinger07]。地球的拓扑是由地表上方和下方的力共同创造的。大陆地壳被由地球内部的地热力驱动的板块构造破裂、波动和扭曲。在表面上方，地球的气候也在影响它的拓扑。久而久之，风、水和冰的侵蚀可以造成巨大的改变。

粒子可以用来很自然地模拟由板块构造和侵蚀造成的地形变形。粒子可用于模拟物质的流动，也可以连接起来形成固体。换句话说，粒子提供了一种简单且灵活的、生成虚拟地形拓扑的机制。

5.1.2 粒子沉积

Shankel 提出了最初的粒子沉积算法来生成类似于火山山脉的地形[Shankel00]。粒子沉积用一个随机步行器（walker）遍历高度场。随机步行器在每个访问位置至少释放一个粒子。在粒子落到高度场后，它必须检查邻近位置的高度。如果发现较低的邻近位置，粒子就移到那里。粒子重复此过程，直到它再也不能移动到较低的邻近位置。图 5.1.1 显示了一个粒子在一维高度场下降。当已释放预定数目的粒子或当用户对结果满意时，该算法即可停止。图 5.1.2 展示了一个用粒子沉积创建的地形。

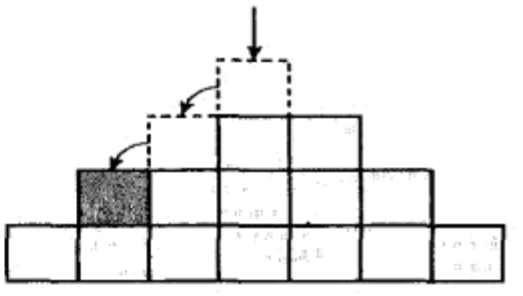


图 5.1.1 沉积一个粒子

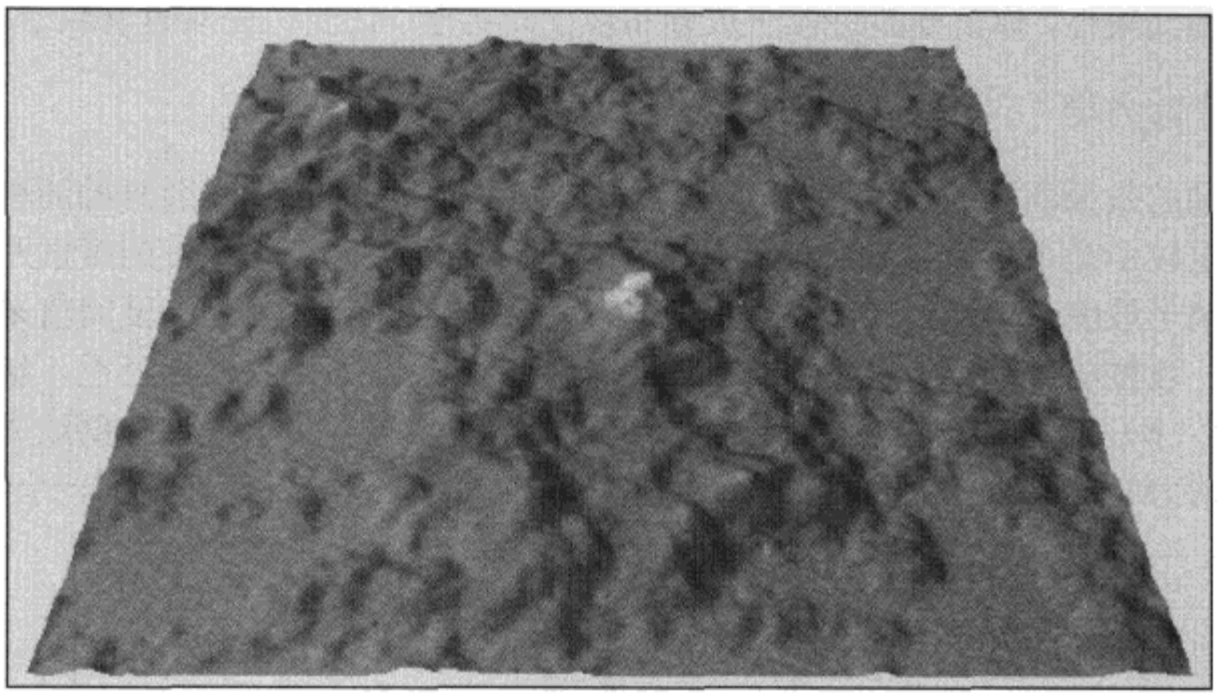


图 5.1.2　一个用原始的粒子沉积算法生成的地形的截图

5.1.3　改进粒子沉积

虽然粒子沉积确实创建了有趣的火山山脉拓扑，但是，用户很容易看出它有一些局限性。注意，粒子所形成的斜坡地形几乎总是 45°，这是在地形上使用试探法来放置粒子的结果。因为如果有较低的邻近位置，粒子就不允许堆起来，所以斜率将永远不可能大于 45°。有时粒子会短时间形成小于 45°的斜坡，这通常是粒子在山谷处累积或接近现有高峰时发生。不幸的是，这些缓坡带从不超过少许几个位置。开发人员希望能够创建更多有趣的地形斜坡，如图 5.1.3 所示，由不同的角度组成，跨越距离或大或小。

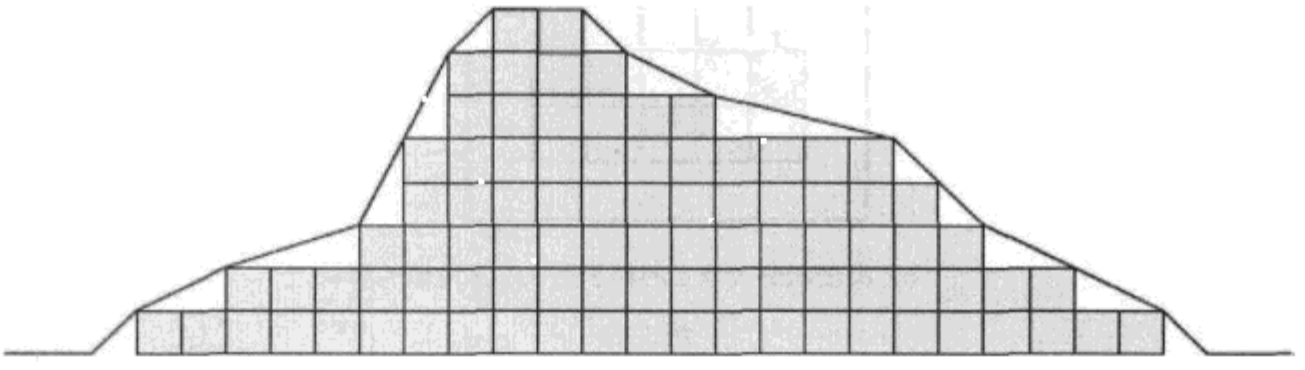


图 5.1.3　一个理想的由各种角度组成的地形的例子

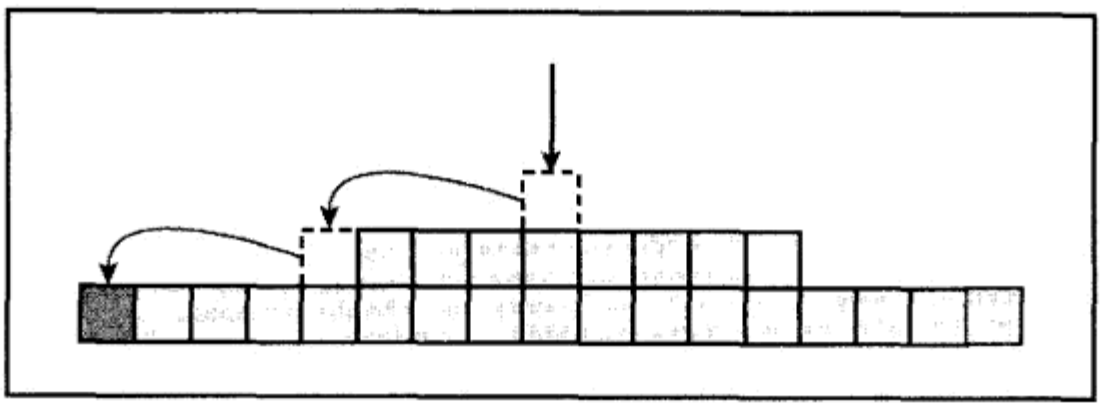
粒子沉积的另一个局限是对主要地形特征的放置没有控制，如火山的高峰，同时也难以控制要创建多少高峰。创建的地形几乎完全是随机的。如果一个关卡设计师想在指定的位置创建特定数量的火山，这将是一个很大的局限性。让关卡设计师对主要地形特征（如大小、一般形状和位置）有更多的控制将是一件很美好的事情。也许，粒子沉积的最大局限是它只适合创建火山山脉的地形。如果要创建其他类型的地形，所有这些限制都可以通过简单地修改粒子沉积来克服。

注意，粒子沉积可以分为两个主要步骤：第一步，定义最初在哪里释放粒子；第二步，定义释放后的粒子在哪里沉积。让我们称第一步为粒子放置，第二步为粒子动力学。为了克

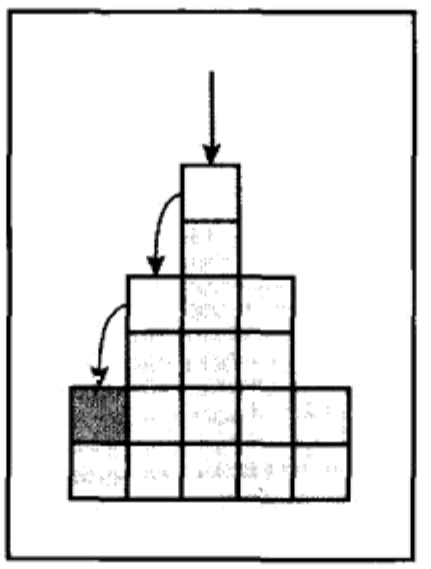
服粒子沉积的局限性，你需要改进粒子放置和粒子动力学。让我们先从查看粒子动力学开始。

改进粒子动力学

模拟侵蚀的效果需要粒子动力学。当粒子被放到高度场后，它就会开始随机地寻找邻近位置，来确定粒子是否可以移动到较低的位置。地形的斜坡被粒子许可的搜索距离隐性地定义。单调的地形坡度可以通过改变粒子的搜索半径及其放在斜坡上的高度阈值来打破。如果搜索半径大，坡度将缓和些，如图 5.1.4（a）所示。相反，如果搜索半径小，坡度将会陡峭些。图 5.1.4（b）说明粒子能堆积形成一个非常陡峭的斜坡。为了使这一斜坡成为可能，需要改变粒子动力学，以至于粒子不会移动到相邻的位置，直到高度差异达到一个特定的阈值。



（a）大搜索半径的粒子形成一个缓坡



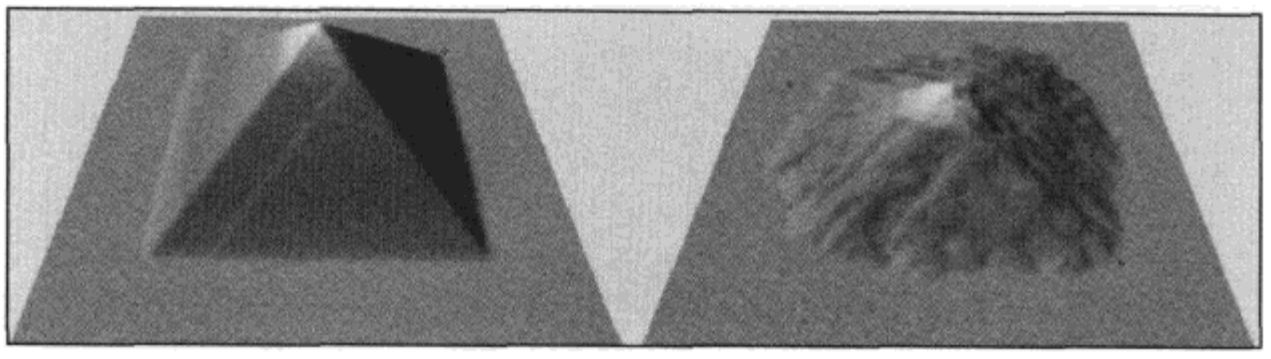
（b）高度阈值大于 1 的粒子形成非常陡峭的坡

图 5.1.4

每个粒子的搜索半径和高度阈值可以随机选择，但这只会导致地形坡度的微小变化，而噪声函数可获得更好的结果。噪声将允许缓坡和陡坡之间的平稳过渡。有几个众所周知的噪声函数。为简单起见，本精粹中的结果利用值噪声（Value Noise）得到。对噪声函数的完整讨论可参阅[Ebert03]。图 5.1.5 显示了使用固定的和用噪声函数定义的搜索半径之间的差异。在图 5.1.5 中，为强调坡度变化的特点，所有的粒子都被释放在相同的位置。与此类似，高度阈值的改变可创建更多极端坡度的地形。下面的伪代码代表了本文使用的粒子动力学。

```
for（每个释放的粒子）
    用二维（或三维）噪声函数确定搜索半径
    while（有较低的位置（在搜索半径内））：
```

移动至最接近的较低位置
增加最后位置的高度场值



5.1.5　左边的地形图是采用恒定搜索半径 1 创建的；右边的地形是在 1～4 改变搜索半径创建的

改进粒子放置

粒子放置试探法定义了粒子最初在高度场的哪个位置释放，这是粒子沉积的一个非常重要的步骤。如果粒子放置是随机的，地形特征也会表现得随机。不同的粒子放置试探法会产生不同类型的地形。接下来的 3 个部分将研究不同的粒子放置试探法，每一个都被设计用来创建一个特定类型的地形。

火山

在考虑一个对火山合适的粒子放置试探法之前，知道真正的火山如何形成很有帮助。火山是由从中央口喷发出的层层火山灰和熔岩形成的。经过许多年，火山灰和熔岩层累积成一个圆锥的形状。圆锥的确切形状是由从火山喷出的岩浆类型决定的。不同的岩浆类型产生不同类型的喷发和地貌。一些火山也有边喷口和裂隙，创建出更多的非对称形状。火山可以有缓坡或陡坡，它们可以有对称形状或不对称形状。像所有地貌一样，火山的形状也由表面的侵蚀所定义。

一个可能的火山粒子放置试探法是在一个地点释放大量的粒子，直至满足停止的条件。尽管这样做可能已经足够了，但由此产生的形状会相当对称，而且有些单调。如图 5.1.6 所示的是一个更有趣的启发式粒子放置试探法，它松散地模拟了熔岩流，这些熔岩流从火山的中央喷口径向漂移。这个粒子放置试探法的伪代码如下：

```
选择一个中央喷口的位置
选择熔岩流的数量
对每个流，选择一个随机长度和方向
while（停止标准还没有得到满足）：
    for（每个流）：
        从中央喷口开始
        while（流的终端还没有达到）：
            释放一个粒子并计算粒子动力学
            沿流的方向移动（+/- 小随机角）
```

用此试探法，火山的形状被流的数量、每个流的长度（对每个流，这不一定是相同的）和释放粒子的总数所决定。停止条件可以是：当已释放某一特定数目的粒子，或当火山高峰已经到达某个高度。如果实现的粒子沉积可以让用户实时预览地形的生成，那么当用户对成果满意时，他们就可以停止算法。

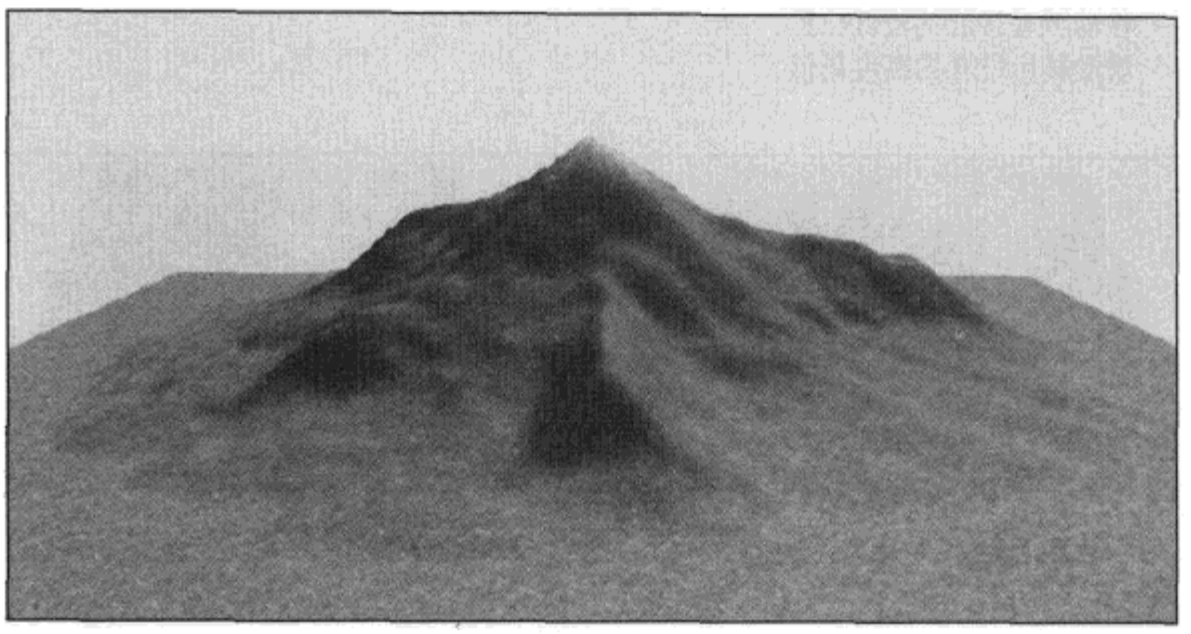


图 5.1.6 使用先进的粒子沉积创建的火山。在颜色插图区查看本精粹的彩色照片

如果想要在火山的高峰有破火山口，可以使用[Shankel00]中的反演算法来颠倒火山高峰。首先，任意选择破火山口的高度，在整个破火山口面的中央喷口倒置高度，然后检查周边位置，如果它们在破火山口面的上面，则颠倒它们，再检查它们的邻近位置。重复此过程，直到没有更多的邻居可被倒置。

注意，现在火山的形状可以更容易地被关卡设计师所定义。一个关卡设计师可以选择在哪里摆放中央喷口来控制火山的位置。此外，如图 5.1.7 所示，熔岩流的路径可以预计算，并覆盖在高度场上，这可以让设计师不用释放一个粒子就预览大小和一般形状。

山

用一个聪明的粒子放置试探法，粒子沉积也可以创建具有真实感的山，山峰之间的山脊形成独特的树形结构。显而易见，这将被称为山的山脊结构。这是多年侵蚀的结果，且周边河网的树形结构和山的山脊结构也有关。因为山脊结构提供了释放粒子的理想地点，所以它很重要，同样，生成山的放置试探法也很重要。

现在，你需要一个过程地生成具有真实感的山脊结构。幸运的是，已经有了一个合适的算法。扩散限制凝聚（Diffusion Limited Aggregation，DLA）是一种形成叫布朗（Brownian）树的枝晶结构的物理过程。图 5.1.8 显示了一个在二维点阵中用 DLA 创建的布朗树。质量上，这个看起来类似于山脊结构。在二维点阵中，创建布朗树的伪代码如下：

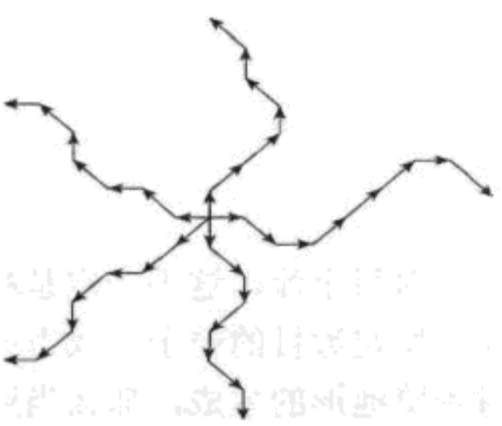


图 5.1.7 模拟熔岩流所采取的一个路径的例子

图 5.1.8 用 DLA 创建的一个布朗树例子

```
选择一个或多个种子位置
while（停止条件还没有得到满足）：
    在随机位置放置一个随机步行器
    移动随机步行器，直到它靠近一个种子（即触到）
    在随机步行器的位置放置一个新的种子
```

布朗树最明显的停止标准是：已经释放了想要数量的粒子，或当布朗树覆盖了想要的面积或体积。布朗树被生成后，定义生成山的粒子放置试探法就直截了当了。首先，在高度场上覆盖布朗树，然后遍历整个高度场，在每一个布朗树覆盖的位置释放一个粒子。你需要遍历高度场好几次，直到地形达到理想的尺寸。图 5.1.9 显示了使用此粒子放置试探法与前面讨论的粒子动力学的结果。

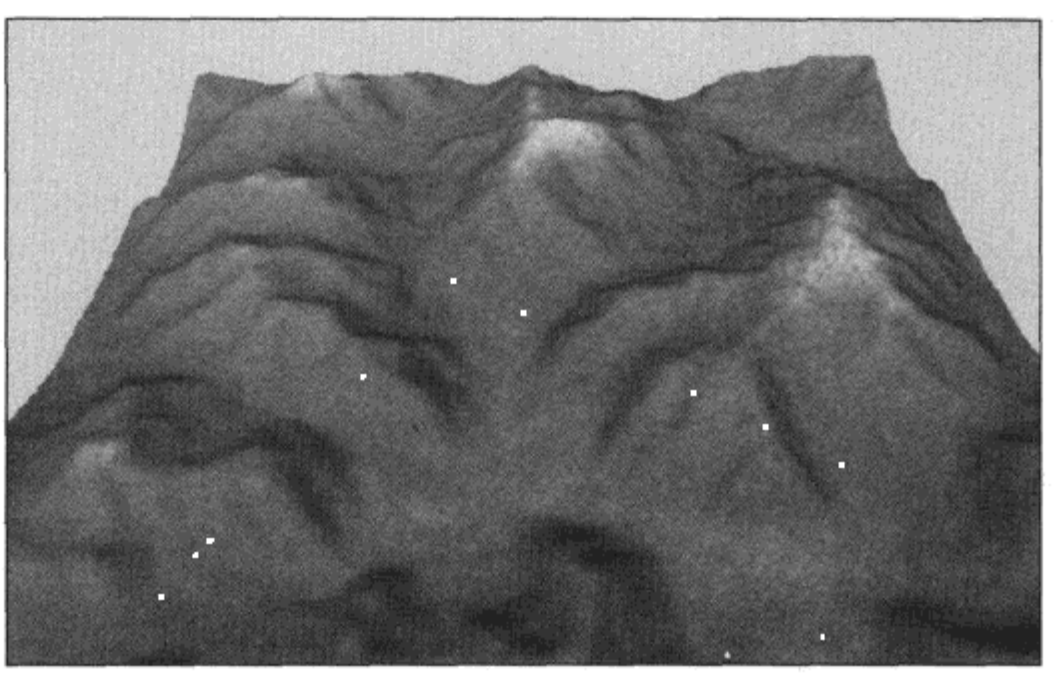


图 5.1.9　使用先进的粒子沉积创建的山（在彩色插页区有彩色版）

注意，布朗树提供了一个很好的用来预览山脊形状的方式，如火山熔岩流，且无需释放一个粒子。一个关卡设计师可以用布朗树来很容易地决定山的位置和大小。布朗树的一般形状也可以通过在期望成长方向的位置启动随机步行器来控制。

如果你熟悉 L－系统（见[Prusinkiewicz96]），你可能在想是不是可以用 L－系统来生成一个适合粒子放置试探法的树形结构。答案是肯定的。然而，L－系统需要一个语法来定义树结构。本文喜欢布朗树的简单性，但这里不得不提一下 L－系统的潜能。

沙丘

沙丘是一个非常有趣的、通常与沙漠相联系的地貌，但也可以形成水下（underwater）地貌。沙漠中的沙丘是风形成的，以至于它们不断移动和改变形状。事实上，据记载，沙丘每年移动多达 20m。沙丘的类型有好几种，但本精粹把重点放在一种常见的名为横贯沙丘的沙丘上。横贯沙丘形成一个垂直于主风向的脊背。如图 5.1.10所示，当风卷起并将粒子迎风抛起，在背风坡释放粒子时，一个沙丘就形成了。这些动作可以方便地利用粒子沉积来模拟。

一个比较容易想到的解决办法就是在高度场里随机挑选粒子，并沿着风向偏离一个小的随机距离。然而，这不一定行得通。少掉的关键是粒子更有可能释放在背风坡而不是迎风坡，因为风的“影子”在背风坡。要模拟该效果，你可以给每个粒子横移的距离指定一个成本。

迎风坡向上的横移费用应低于背风坡下移的费用。下面的伪代码实现了一个合适的费用函数，图 5.1.11 说明了此结果。

```
while（停止标准还没有得到满足）：
    选择一个随机的位置，并删除一个粒子
    位移 = 小随机数
    当位移 >= 0：
        沿着风向，将粒子移动一个位置
        if（粒子上移）
            位移 -= 1
        else
            位移 -= 2
    释放粒子，并计算粒子动力学
```

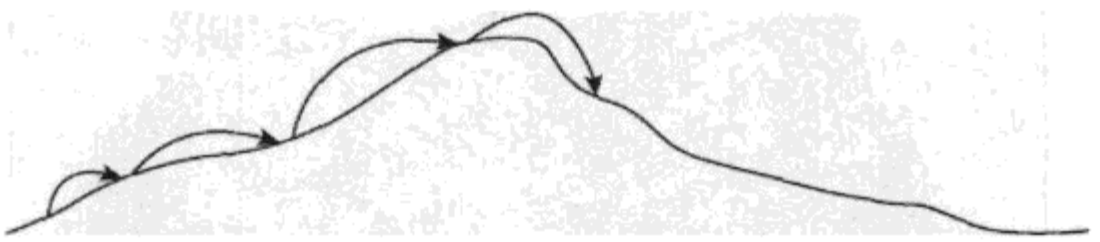


图 5.1.10 粒子在迎风坡上移及在背风坡沉积，从而形成沙丘

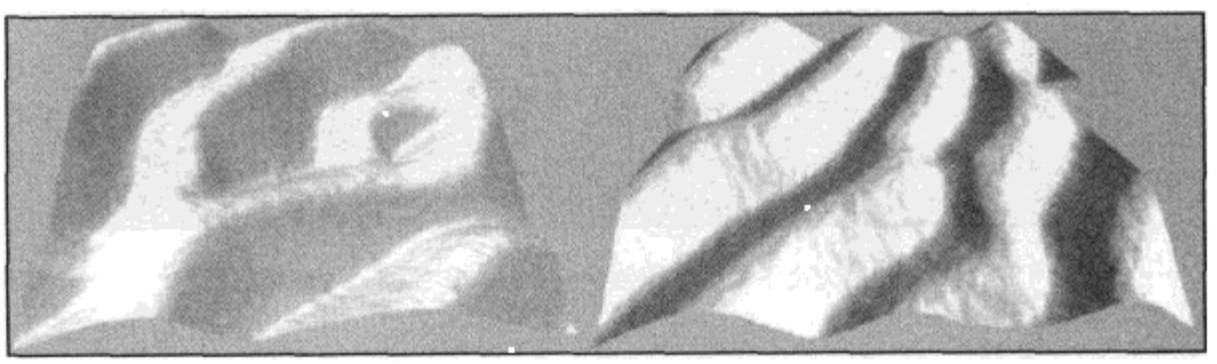


图 5.1.11 采用先进的粒子沉积创建的沙丘

这个例子中，迎风坡上移的费用只是水平移动的距离（即没有垂直费用），而背风坡下移的费用是横向和纵向的移动距离，这是一个非常简单而有效的成本函数。不同的成本函数将产生不同形状和动力学的沙丘。因此，大家应该多做试验。

悬垂（overhang）地形

如图 5.1.12 所示，悬垂地形是一个凸出于其他地形的地势。对粒子动力学稍加修改，粒子沉积就可以创建这类地形。对释放在地形上的每个粒子指定其黏度属性，并查看它朝地形降落时的粒子路径。如果在降落到地形之前，一个粒子触到另一个相邻位置上的粒子，那么下降粒子的黏性将决定粒子是停止还是继续下降。如图 5.1.13所示，当非常黏的粒子触及陡坡的表面时，它们将积累形成一个悬垂。区域的黏性可由用户确定或用噪声函数定义。下面的伪代码提供了更多细节。

```
任意选择一个阈值，S
for（每个下降的粒子）：
```

```
确定粒子的黏性度，Sp（三维噪声）
检查粒子朝高度场下降时的路径
if（粒子触及邻近粒子）
    确定相邻粒子的黏性，Sa（三维噪声）
    if（Sp >= S）和（Sa >= S）
        让粒子停留在这一位置
else
    使用在粒子动力学一节讨论的试探法
```

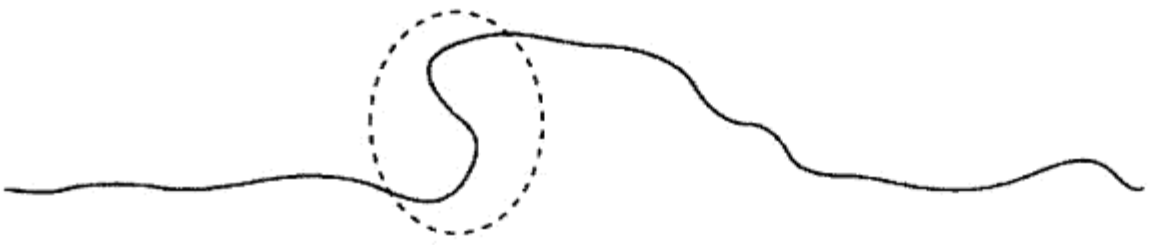


图 5.1.12　悬垂地形

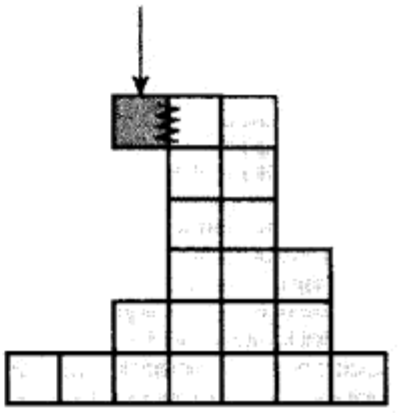


图 5.1.13　当黏性粒子落到表面上时，它们附着在陡坡上

注意，由于高度场是一个对每个位置（即二维标量场）都指定高度的两维点阵，传统的高度场不能用来定义悬垂地形。在三维点格中（即三维标量场），体素（voxel）可以代表体积。因为它们可以利用移动立方体（marching cube）/四面体算法来实现多边形化，所以它们用于悬垂地形的建模是很理想的。体素表示法将增加粒子沉积的空间和时间复杂性，而混合表示法只是在需要的地方才使用体素，这样可以降低一些成本。

5.1.4　结论

粒子沉积是一种用于创建不同类型真实感地形的强大工具。本文中的地形不是粒子沉积可创建地形的全部列表。峡谷、弹坑、岩洞、高原、梯田和各种岩石突起也仅仅是用粒子沉积可能实现的几个其他例子。

5.1.5　参考文献

[Ebert03]Ebert, David S., Musgrave, F. Kenton, Peachey, Darwyn, Perlin, Ken, and Worley, Steven. Texturing & Modeling: A Procedural Approach, Morgan Kaufmann Publishers, 2003.

[Grotzinger07]Grotzinger, John, et al. Understanding Earth, W. H. Freeman and Company, 2007.

[Prusinkiewicz96]Prusinkiewicz, Przemyslaw, and Lindenmayer, Aristid. The Algorithmic Beauty of Plants, Springer Verlag, 1996.

[Shankel00]Shankel, Jason. "Fractal Terrain Generation—Particle Deposition," Game Programming Gems, pp. 508–511. Charles River Media, 2000.

5.2 减少骨骼动画中的累积误差

Bill Budge，索尼娱乐美国分部
bill_budge@playstation.sony.com

本精粹将介绍一个在播放骨骼动画中减少累积误差数量的简单技巧。它在离线动画数据处理时作为正常动画工具链的一部分被应用，并不影响动画数据的大小，无须对运行时的动画重放引擎做任何改变。

5.2.1 游戏动画系统的快速巡视

在先驱 3D 游戏 Quake 中，人物是通过对每个姿势存储一个单独的网格，并在每一帧渲染一个不同的网格来制作动画的[Eldawy06]。这种动画比较简单，在理论上能够达到最高的质量，但很难修改和融合（blend），同时也需要大量的内存来存储网格。因为这些原因，现在骨骼动画是 3D 游戏的标准。

骨骼动画是通过在单个角色的网格顶点上绑定一系列坐标变换（“骨骼”的骨头），再通过摆动骨头制作出动画来变形网格的。角色网格的顶点定位于一个单一的坐标空间，还有些把骨头与网格对齐的变换。这些变换组成所谓的骨架的“默认”或“重设”姿势。为了变形网格到一个新的姿势，首先使用重设变换的逆运算把顶点从它们原来的空间带到“骨头”空间，然后用新的骨头变换把顶点移动到它们的最终位置。每个顶点的方程如下：

$$V' = M_{\text{pose}}(M_{\text{rest.}}^{-1}V) \qquad (5.2.1)$$

骨骼动画的效果不错。因为大多数游戏角色和物体不是无定形的滴状物，所以能被一系列刚体密切地近似。目前，因为骨头的数量远比需做动画的网格顶点数要少，这就大大减少了数据量。又因为修改和融合骨头变换比对网格做同样的操作容易得多，这也大大增加了灵活性。

进一步观察得出了另外一个技巧，这又将动画的数据减半。游戏角色和物体不只是一系列刚体的随机集合，它们是互相连接的（至少在粉碎它们之前）。在这些关节处，每个骨头和其他的骨头相连接，这样你就可以把这些变换组织成一个层次，使得除了根变换以外的所有变换都相对于它们的父变换。由于关联骨头不会相对平移，因此所有的子平移都是恒向量，可从动画中去掉，而只存储在骨架中。事实上，它们已经在重设变换里。因此，动画只需要有一个根的平移轨迹和对每一个骨头的旋转轨迹就可以了。

父相对变换的重放是直接了当的。首先，你从根的平移和旋转轨迹构造根的变换。然后，对根的每个孩子，你通过连接孩子旋转和父变换来构造孩子变换。对孩子的孩子，重复此过程，直到你已经重构骨骼层次中每个骨头的变换为止。

对于骨骼动画更深入的说明，及在关节周围改进网格变形的蒙皮技术的介绍，参见[Lander98]。

5.2.2　累积误差

不幸的是，有层次结构的骨骼动画虽然可以减少数据，但这是要付出代价的，这不是意味着要做许多额外的工作（按所描述的宽度优先顺序重建，对每个骨头，你只需要一个额外的矩阵级联）。真正的问题是方程 5.2.1 实际已成为：

$$V'=M_{\text{root}}\cdots M_{\text{parent}}M_{\text{child}}(M_{\text{rest}}{}^{-1}V) \tag{5.2.2}$$

利用方程 5.2.2 的变换重建比方程 5.2.1 更不可靠。这有两个原因。首先，较高层次的变换的任何误差将影响在它下面的变换。例如，根变换的一个误差将影响到所有其他变换。其次，每一步的误差会自然地倾向于积累，进而建立一个更大的误差。对本精粹而言，我们假定每个变换的误差是一个随机变量（否则将能够补偿它）。因此，级联旋转变换引起的第二个效应就像是加上一些随机变量。

这两个效应意味着最大的误差将在离根最远的骨头上，这些通常是角色的手和脚。这样的瑕疵（artifacts）在许多游戏中都可以看到。一个经典的例子是：站立着的空置动画的脚看起来在地上滑来滑去。更糟糕的情况是：一个角色两手夹持大棒子和剑。被抓的物体是一个叶骨头，以其中的一只手为父。在不同变换链终端的另一只手看起来像游泳般到处移动，并穿过那个物体。

对于父相对的动画，你应该主要关心旋转误差。这个误差从何而来？一小部分是由于使用了有舍入和精度误差的浮点算法。然而，到目前为止，最重要的误差来源是大部分游戏用来进一步缩小动画数据大小的有损压缩方法。

有许多方法可用来压缩旋转数据，有些是无损压缩。例如，像膝盖和胳膊肘关节都只有一个自由度。对每个姿势都存储全部的旋转，如四元数，是一种浪费。相反，可以存储旋转轴，动画数据就被减少成一系列的角度了。

有损压缩算法可以削减更多的存储成本。一个最简单的技术是关键帧削减，它查看旋转值并试图消除那些可以从邻近的值来插值得到而不超过一定误差阈值的值。关键帧削减的问题是很难知道哪些值要保留，哪些值该扔掉。一个更好的方法是采用曲线拟合算法把旋转值转变成一个多维的，并近似原始数据到某些误差之内的样条曲线[Muratori03]。样条曲线是一个对现实世界旋转数据的良好拟合。它们非常紧凑，并且很容易在运行时评估。小波压缩是另一种流行的技术[Beaudoin07]。

即使我们找到了一个高效率的表示方法，如果数字在一个已知的小范围内，那么存储大量的浮点数据是低效的。如果使用四元数，所有数字都在[-1，1]的范围，那么，每个数字的指数的 8 位都被浪费了。你可以将数字压缩成 12 位或 16 位定点形式。对四元数压缩的完整精粹，可参见[Zarb-Adami02]。

压缩算法常常被扩展到因累积误差所引起的瑕疵变得很明显的地步。在这种情况下，在每个旋转处，你永远都有重大误差。

图 5.2.1 分别在原始的姿势和一些可能的重建姿势处，显示了一个简单双骨头的二维层次，其中，对每个变换都给予同样的随机误差。父变换的误差范围是用灰色区域表示的。3 个子变换连同它们的误差范围也被显示了出来，其中一个和实际的子变换对齐，另外两个的父变换有很大的误差。在从父亲到孩子的变换过程中，注意传递到骨头末端的误差是如何增加的。

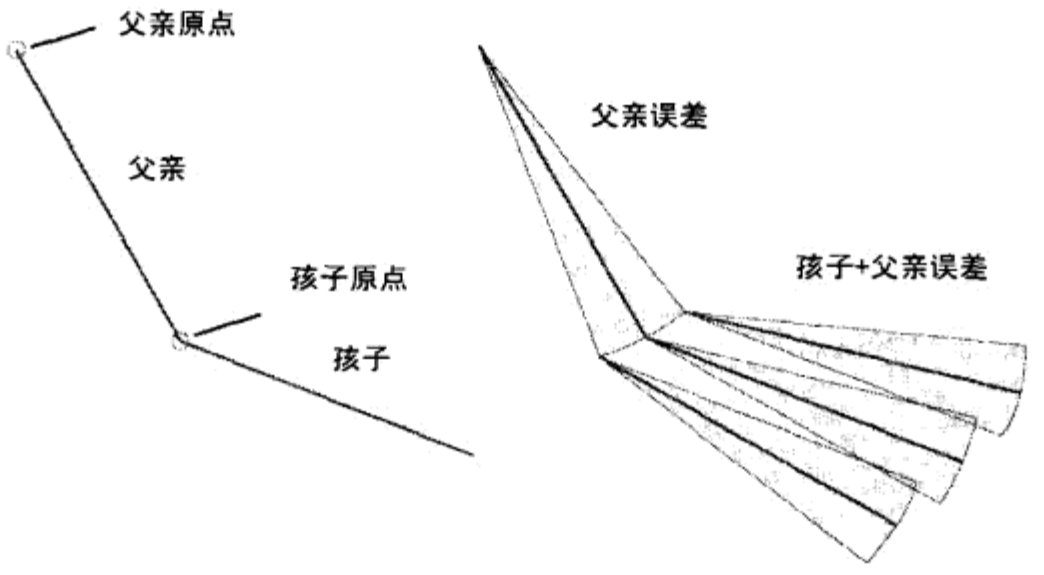


图 5.2.1　从父亲到孩子，累积误差增加了

消除累积旋转误差

处理动画的传统算法，首先从核准（authored）表示中提取所有的变换数据到一个共同的坐标空间。然后，所有子变换通过与父变换的逆相连接变成父相对。最后，相对变换被压缩，并依照运行时的播放引擎进行格式化。让我们称之为天真的算法，因为它假定压缩和运行重建没有加入误差。

本精粹的想法是要考虑重建误差，并用重建算法来获得有变换误差的父变换，使子变换对那个变换形成父相对，而不是针对原始的变换。

这就导致了下面的过程（我们称之为算法 1）：

（1）压缩并格式化根变换的数据。

（2）对步骤（1）的结果运行解压缩算法，并用结果取代原来的变换。

（3）对根的每个孩子，使它们的变换数据相对于解压缩的根变换数据。压缩和格式化父相对的旋转数据。

（4）对步骤（3）中的每个孩子运行解压缩算法来获得最终的变换数据，并用其结果取代原来的孩子数据。

（5）继续沿层次往下走，直到所有的骨头都被处理过。

这完全消除了旋转误差的积累，因为对每个孩子变换，第（3）步减去了父变换的旋转误差。然而，父旋转误差不仅仅是转动孩子，它还平移孩子（除非孩子的原点和父亲的原点重合）。这意味着第（3）步产生的父相对转换通常会有一个和存储在骨架中的常量不同的平移。这种平移误差不能被任何孩子旋转消除。虽然你可以通过增加一个新的平移来校正它，

但这将失去把转换变成父相对的全部目的，因为第（1）步就是要消除这些平移。所以，移动误差仍然在积累。但是，因为旋转误差较少，所以总误差小于天真算法。

图 5.2.2 显示了天真算法和算法 1 的结果。注意，所有的孩子骨头是怎样有跟真实姿势相同的方位的（尽管仍然存在局部误差），以及它们是如何抵消因父骨头的旋转误差造成的移动误差的。

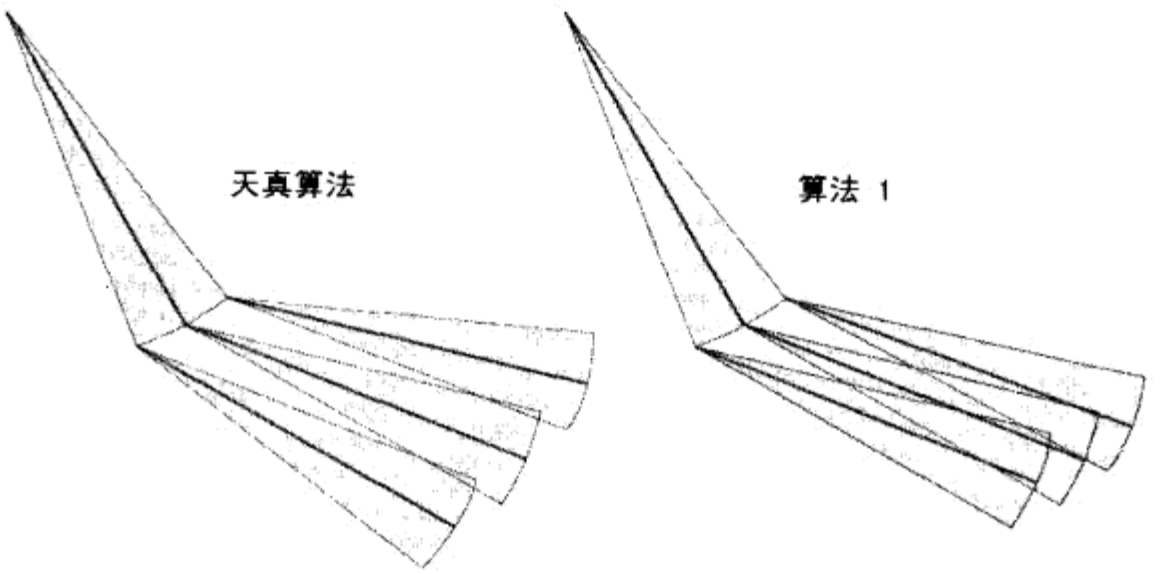


图 5.2.2　去除累积旋转误差

减少这种移动误差是可能的，但这样做必须旋转孩子骨头而离开它的真正方位。要计算此旋转，要先在骨头上选择一个想把移动误差最小化的定点，我们把它称为一个核心点。一个核心点可以是孩子骨头的原点，或者是标识骨头“终端”的一些任意点。然后旋转由算法 1 重建的骨头来移动核心点到其真实位置的最近点。图 5.2.3 显示了那个几何结构。

用以下的方程来计算旋转：

$$\text{Axis} = O'S' \times O'S \tag{5.2.3}$$

$$\text{angle} = \cos^{-1}(O'S' \cdot O'S) \tag{5.2.4}$$

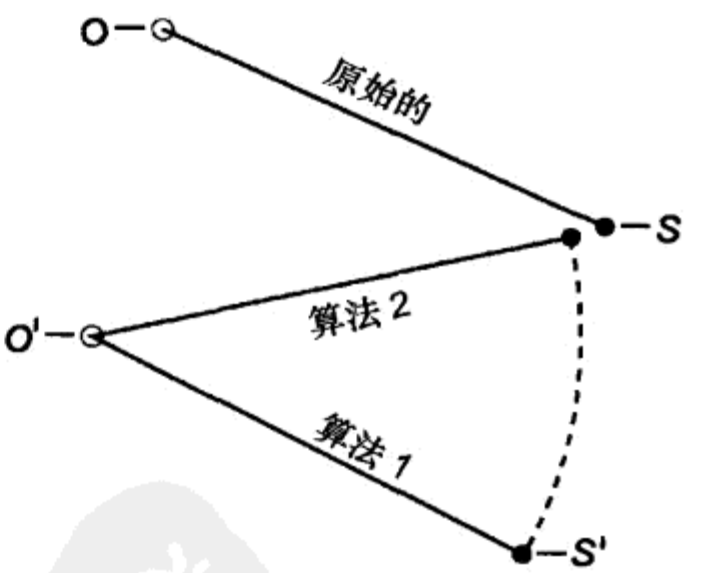


图 5.2.3　在核心点处，减少平移误差

修改算法 1 的第（3）步，得到算法 2：

（1）压缩并格式化根变换的数据。

（2）对步骤（1）的结果运行解压缩算法，并用结果取代原来的变换。

（3）对根的每个孩子：

① 使它的变换数据相对于解压缩的根变换数据；

② 将此和解压缩的父变换连接得到重建的变换，而且没有误差；

③ 计算把重建变换中的核心点，带到离它的实际位置最近的那个旋转，并将其加入变换；

④ 压缩并格式化父相对的旋转数据。

（4）对第（3）步的每个孩子，运行解压缩算法来获得它的最终变换数据，并用结果取代原来的孩子数据。

（5）继续沿层次往下走，直到所有的骨头都被处理过。

图 5.2.4 显示了天真算法和算法 2 的结果。注意孩子骨头现在如何有微小的旋转误差（然而，它们并不积累，因为每一步都将校正父亲的误差），以及平移误差怎样被减少了。

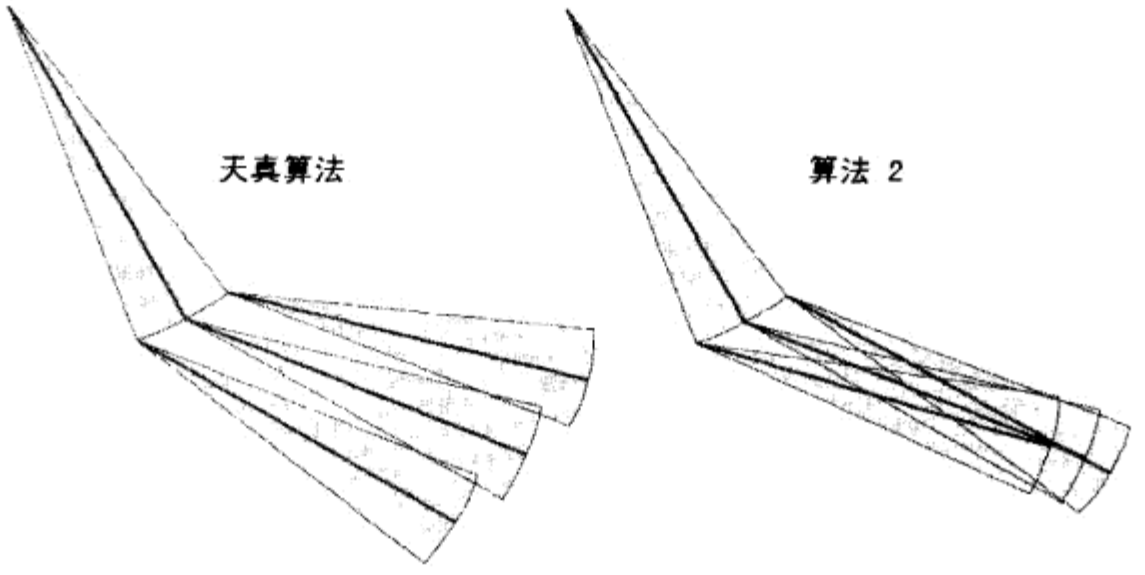


图 5.2.4 减少累积平移误差

算法 2 并没有完全消除平移误差。其中的一个解决方法是在叶骨头上加上平移轨迹来抵消任何可恶的瑕疵。另一种方法是采用逆运动学（Inverse Kinematics，IK）系统，以确保骨头停留在它们应该在的地方。即使一个游戏采用了 IK 系统，这些减少误差的技术仍然非常有用。因为它们提高了姿势重建的质量，使其更接近艺术家的原始版本。

5.2.3 结论

你已经看到骨骼动画系统是怎样受累于累积误差的，以及动画数据的常规处理可导致播放时的明显瑕疵。但是，通过对处理算法的简单修改，你可以消除累积旋转误差及减少平移误差。

因为只有动画预处理被改变，所以它对游戏运行时没有效率及内存方面的影响。最后，移动轨迹可加在核心的骨头上，以消除任何剩余瑕疵。

5.2.4 参考文献

[Beaudoin07]Beaudoin, Philippe. “Adapting Wavelet Compression to Human Motion Capture Clips.”

[Eldawy06]Eldawy, Mohamed. “Trends of Character Animation in Games.”

[Lander98]Lander, Jeff. “Skin Them Bones: Game Programming for the Web Generation,” Game Developer Magazine, May 1998, pp. 11–16.

[Muratori03]Muratori, Casey. “Discontinuous Curve Report.”

[Zarb-Adami02]Zarb-Adami, Mark. “Quaternion Compression. ”Game Programming Gems 3, Chartes River Media Press,2002.

5.3　粗糙材料漫反射光着色的另一个模型

Tony Barrera，Barrera Kristiansen AB
tony.barrera@spray.se
Anders Hast，创造媒体实验室，University of Gävle
aht@hig.se
Ewert Bengtsson，图像分析中心，Uppsala University
ewert@cb.uu.se

本精粹将展示有可能用一个相当简单的着色模型来改善粗糙材料的着色，并讨论对粗糙材料很明显的平坦效果以及用来产生后向散射效果（backscattering effect）的可能方法。

5.3.1　简介

通常，Lambert 模型（余弦法）[Foley97]被用来计算漫反射光照（diffuse lighting），特别是当速度至关重要的时候。这种模式适用于 Gouraud [Gouraud71]和 Phong[Phong75]着色。然而，众所周知的是，这个模型会生成类似于塑料的材料，理由是：该模型假定物体本身是完美的，即表面在所有的方向上同等地散射光。但是，现实生活中没有这样的理想材料。

现有的文献介绍了一些可用于金属的模型[Blinn77、Cook82]。这些模型假定表面是由小的 V 型凹槽组成。Oren 和 Nayar 提出了一个适合粗糙表面的光照模型[Oren94、Oren95a、Oren95b]，该模型可用于像黏土和沙子的粗糙表面。不过即使是这种模式的简化形式，其计算也相当昂贵。尽管如此，利用此模式的优点是它对粗糙表面产生比较准确的漫反射光照。它们展示了除了边缘的光照强度突然下降以外，圆柱形的黏土花瓶在整个被光照的表面几乎同样明亮。

Lambert 模型的着色会逐渐消退，然而在现实生活中，这是很少见的。这种效果显示在图 5.3.1 和图 5.3.2 中。注意，对图 5.3.1 中 Lambert 着色的茶壶，其强度没有缩减。因此，它看起来比图 5.3.2 中用 Oren-Nayar 模型着色的茶壶更亮。然而，对 Oren-Nayar 模型，在整个表面上，光的强度显然几乎同等明亮。

图 5.3.1 用 Lambert 着色的茶壶

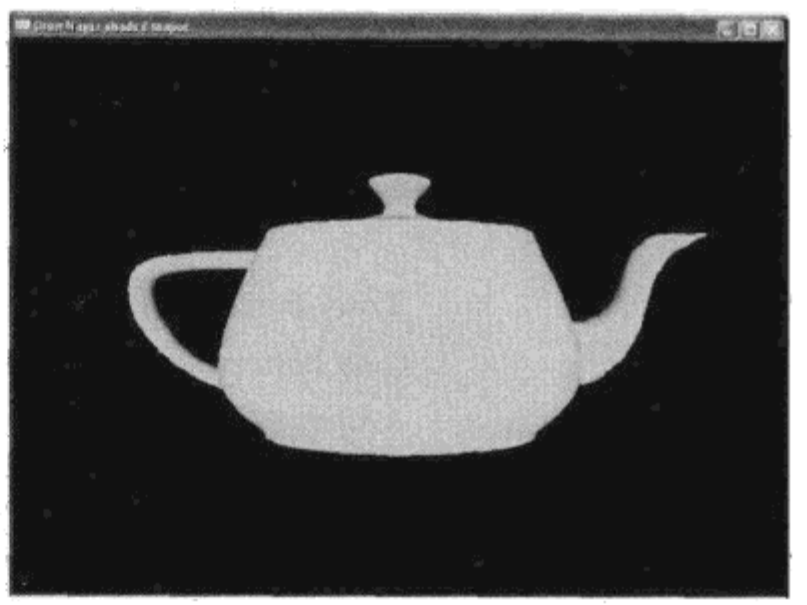


图 5.3.2 用 Oren-Nayar 着色的茶壶

5.3.2 平坦效果

Lambert 和 Oren-Nayar 模型之间的一个主要不同点是 Oren-Nayar 模型产生的漫反射光照在整个表面几乎是同样明亮的。这种平坦效果可以通过迫使漫反射光线近似达到最大强度来建模，但除了在光照强度应该很快降到零的边缘区域。因此，在很大一部分区间内，着色曲线在水平方向上是平坦的。下面的函数可用于此目的。

$$I_d = k(1 - \frac{1}{1 + \rho\cos\theta}) \tag{5.3.1}$$

这里，$\cos\theta = n \cdot l$ 是 Lambert 法则，ρ是表面粗糙度，来决定那个函数有多平（或接近 1），k 是一个常量。

$$k = \frac{1+\rho}{\rho} \tag{5.3.2}$$

常数 k 确保当 $\cos\theta=1$ 时，$Id=1$。注意，k 可预先计算，也可以包含表面颜色。

粗糙度ρ不是用其描述物理行为的方式（Oren 和 Nayar 描述的凹槽分布）推出来的。相反，它可以用来调整曲线的斜率，从而模拟不同的粗糙度。图 5.3.3 显示了 Lambert（$\cos\theta$）（最陡的曲线）和 ρ={0.75，1.5，3.0 和 6.0}的新模型的比较。所用的 ρ值越大，在区间上，曲线就越接近于 1。

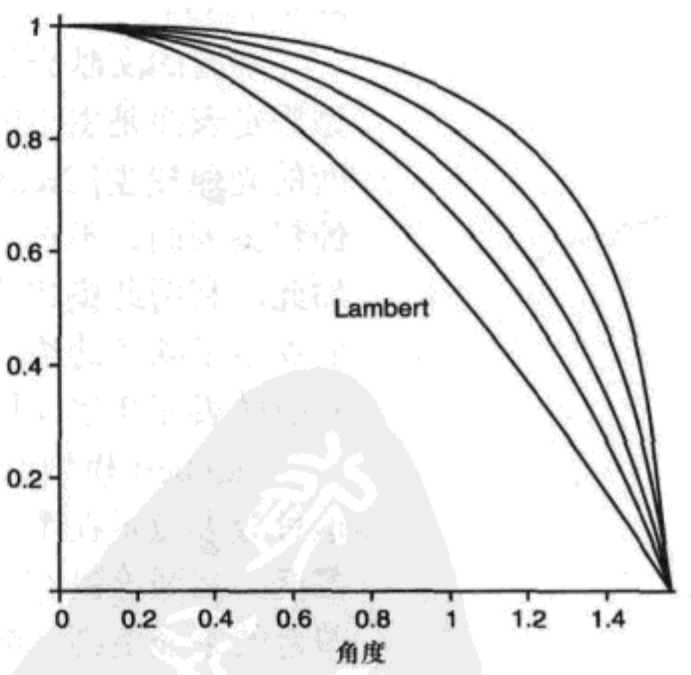


图 5.3.3 对于ρ={0.75，1.5，3.0 和 6.0}法线和光源向量间不同夹角下的强度

图 5.3.4～图 5.3.7 显示了利用此方法对一个茶壶着色的效果。注意当ρ增加时表面是如何显得更平坦的。

用 GLSL 写的着色器代码如下：

```
uniform float shininess;
varying vec3 normal, color, pos;
```

```
void main()
{
    vec3 l = normalize(gl_LightSource[0].position.xyz - pos);
    vec3 n = normalize(normal);

    float nl = max(0.0, (dot(n,l)));

    // 平坦化
    float rho = 6.0;
    float k = (1.0+rho)/rho;
    float diff = k*(1.0-1.0/(1.0+rho*nl));

    gl_FragColor = vec4(color*diff, 1);
}
```

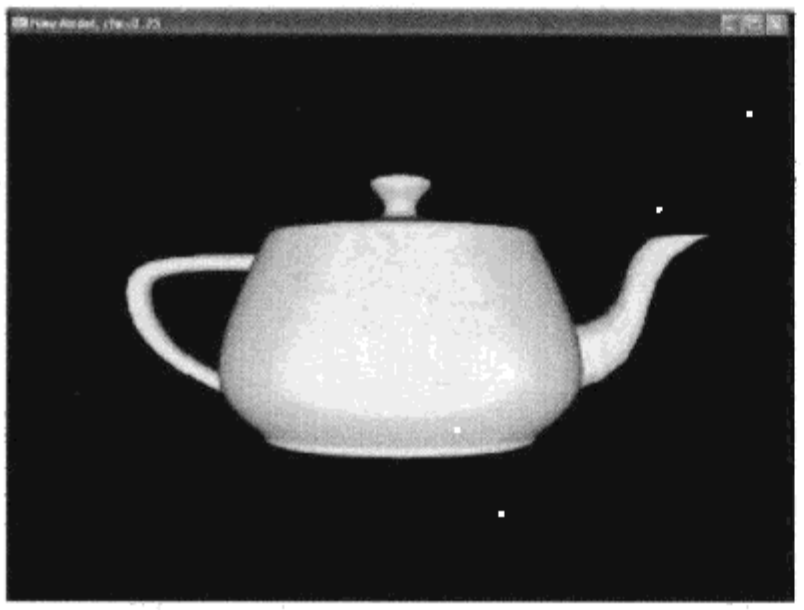


图 5.3.4　ρ是决定函数该有多平坦（或接近 1）的表面粗糙度。这里的ρ为 0.75

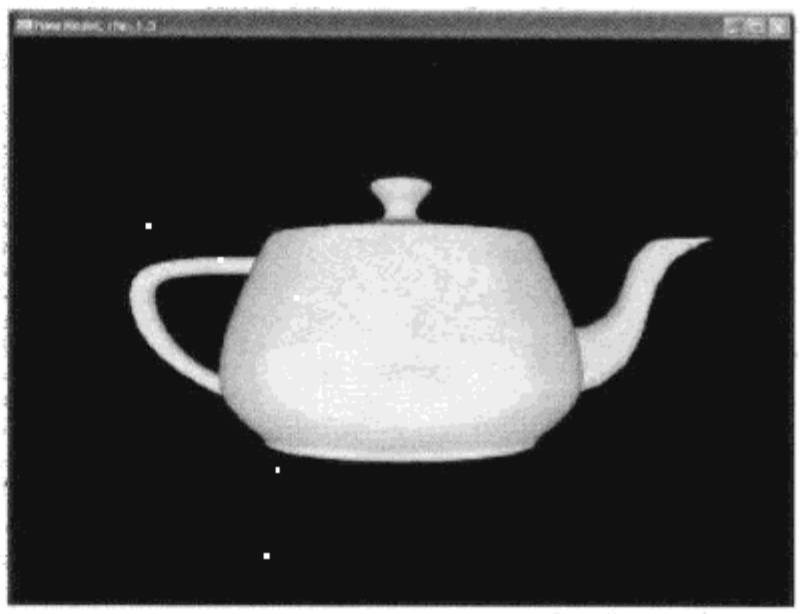


图 5.3.5　这里的表面粗糙度ρ为 1.5

图 5.3.6　这里的ρ为 3.0。注意当ρ增加时表面如何看起来较平坦

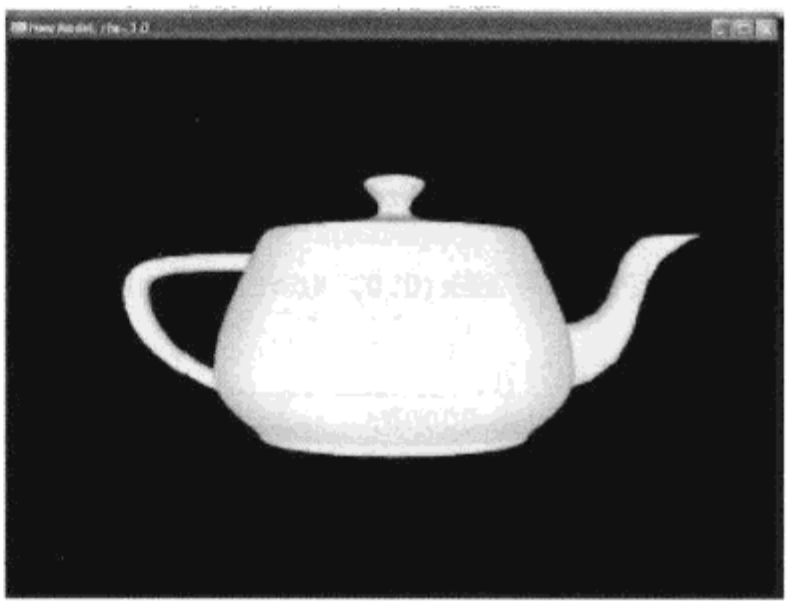


图 5.3.7　这里的ρ为 6.0。所有表面中最平坦的一个

5.3.3 后向散射

后向散射效果在许多材料中都可见，它是我们在黑暗中使用手电筒能很好地看见东西的一个主要原因。然而，这是一个相当微妙的效果，在现实生活中很难被注意到，所以我们应该很小心地使用它。因为只有当光源和观察者在同一方向时它才可见，所以可以用 $i \cdot v$ 来建模。下面的方程被用做和漫反射光照相乘的一个衰减因子。

$$F_{bs} = \frac{f(i \bullet v) + b}{1 + b} \tag{5.3.3}$$

对 f，我们使用了幂函数，但也可以使用 Schlick 模型[Schlick94]。此函数决定了表面效果是如何按照和镜面光类似的方式分布的。

常量 b 将确定后向散射效果对漫反射光有多大的影响。一个大的 b 将产生小的影响，反之亦然。在图 5.3.8 中，一个小的 b 只用于展示效果。

在图 5.3.9 中可以清楚地看到，当观察者在和光源不同的方向看物体时，后向散射效果就消失了。

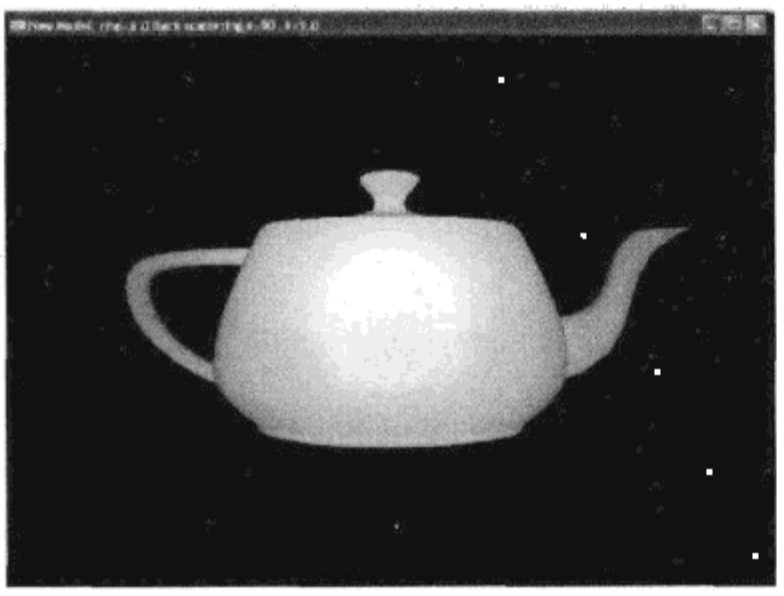


图 5.3.8 因为光源和观察者在同一方向，茶壶的中心如何看起来更亮

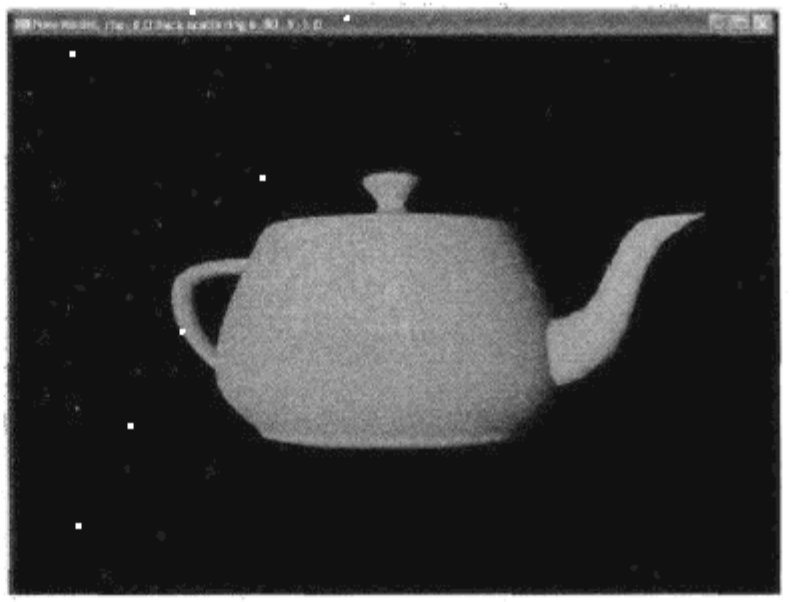


图 5.3.9 光源不再和观察者在同一方向，后向散射就不见了

计算后向散射所需的额外代码如下：

```
vec3 v = normalize(-pos);

float lv = max(0.0, (dot(l,v)));

// 后向散射
float b = 1.00000;
float bs = (pow(lv, 80.0)+b)/(1.0+b);

gl_FragColor = vec4(color*diff*bs, 1);
```

另一种可能性是把此效果作为它自己的一项加到 Phong-Blinn 模型中。图 5.3.10 用了以下公式。

$$I_{bs} = K_{ks} f(i \bullet v) \tag{5.3.4}$$

常量 K_{ks} 决定多少效果是可见的，再一次，函数 f 决定了表面效果将如何分布。值得一提的是，在图 5.3.10 中，后向散射的强度和表面的颜色相乘。

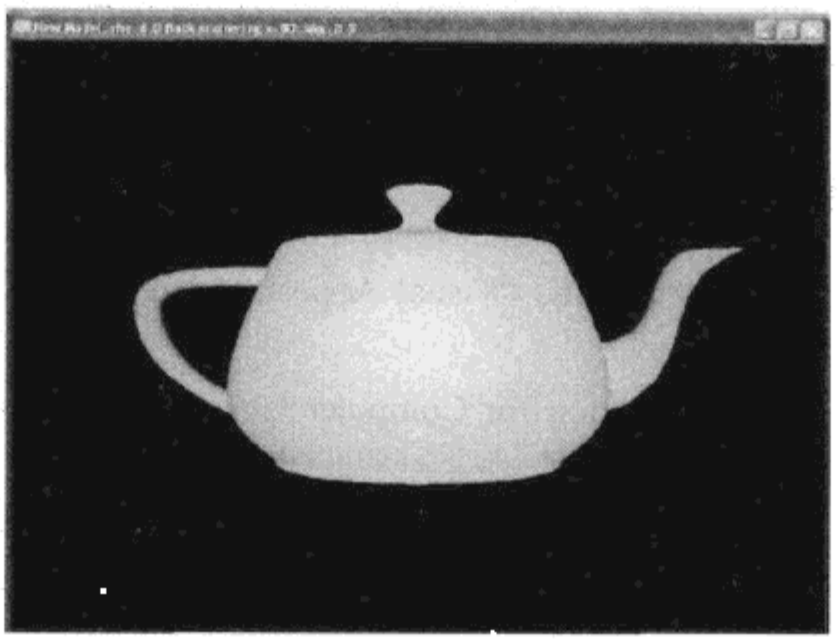


图 5.3.10　因为光源和观察者在同一方向，茶壶的中心看起来更亮了

更改代码如下：

```
float kks = 0.3;
float bs = kks * pow(lv, 80.0);
gl_FragColor = vec4(color * (diff+bs), 1);
```

5.3.4　结论

Oren-Nayar 模型是相当复杂的，而本文提出的模型很简单，且易于使用。然而，它产生的效果同样可以模仿粗糙材料的典型行为。你看到了去计算后向散射效果的两种可能方法，并且很难分辨出哪种方法更好。虽然你可以使用大的幂函数值来使两者的区别在图像中可见，但很显然，当一个物体被交互旋转时，较低的值提供了一个更令人满意的结果。

5.3.5　参考文献

[Barrera05]Barrera, T., Hast, A., and Bengtsson, E. “An Alternative Model for Real-Time Rendering of Diffuse Light for Rough Materials,” SCCG’05 Proceedings II, pp. 27–28, 2005.

[Blinn77]Blinn. J.F. “Models of Light Reflection for Computer Synthesized Pictures,” Proceedings of the 4th Annual Conference on Computer Graphics and Interactive Techniques, 1977, pp. 192–198.

[Cook82]Cook, R.L., and Torrance, K.E. “A Reflectance Model for Computer Graphics,” ACM Transactions on Graphics (TOG), 1, 1, 1982, pp. 7–24.

[Foley97]Foley, J.D. van Dam, A., Feiner, S.K., and Hughes, J.F. Computer Graphics: Principles and Practice, Second Edition in C, 1997, pp. 723–724.

[Gouraud71]Gouraud H. "Continuous Shading of Curved Surfaces," IEEE Transactions on Computers, Vol. c–20, No. 6, June, 1971.

[Oren94]Oren, M., and Nayar, S.K. "Generalization of Lambert's Reflectance Model," Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques, 1994, pp. 239–246.

[Oren95a]Oren, M., and Nayar, S.K. "Generalization of the Lambertian Model and Implications for Machine Vision," International Journal of Computer Vision, 1995, pp. 227–251.

[Oren95b]Oren, M., and Nayar, S.K. "Visual Appearance of Matte Surfaces," Science, 267, 5201, 1995, pp. 1153–1156.

[Phong75]Phong, B.T. "Illumination for Computer Generated Pictures," Communications of the ACM, Vol. 18, No. 6, June, 1975.

[Schlick94]Schlick, C. "A Fast Alternative to Phong's Specular Model," Graphics Gems 4, 1994, pp. 385-387.

5.4　高效的细分表面

Chris Lomont
chris@lomont.org

细分表面是一种用较粗多边形网格来代表光滑曲面的方法。通常，它用于存储和生成由低细节网格加上各种标量贴图而来的高细节几何体（经常是动态地）。由于它的易用性、支持多分辨率编辑、能造模任意拓扑结构和数值鲁棒性，所以在建模和动画工具中已经很流行。本精粹融合众家之长，提出了 Loop 细分的扩展，并增加了有用的实现细节，其结果是一套对几何形状、纹理及其他属性的完整的细分规则集。这些表面适合于游戏中的地形、人物和几何体。

本精粹还大体回顾了快速细分和渲染的方法。在学习所提供的资料后，你就可以在工具或在游戏引擎自身的生产环境中实现细分了。

5.4.1　细分方案的介绍

有许多类型具有不同属性的细分方案(scheme)，其中的一些属性如下。

- 网格类型——通常，网格由三角形或四边形组成。
- 光滑度——这是极限表面的连续性，通常用 C^1、C^2…，或 G^1、G^2…表示。
- 插值[Zorin96]还是逼近——插值方案通过原始数据点，而逼近方法不会。
- 支持尺寸——这是影响到一个给定表面点最终位置的周边几何的数量。
- 分裂——有些方案通过更多的表面来取代表面，其余的通过新的顶点集去取代顶点来实现。更有一些用一个“新的”网格取代前面的整个网格。

表 5.4.1 列出了一些常见方案以及关于它们的一些数据。

虽然本精粹主要集中在生成几何和渲染问题上，但是细分表面还有许多其他用途，包括：

- 渐进式网格；
- 网格压缩；
- 多分辨率网格编辑（[Zorin97]）；
- 表面和曲线拟合（[Lee98]和[Levin99]）；
- 点集到网格的生成。

表 5.4.1 细分方案

方　法	网　格	光 滑 度*	分　裂	方　案
Catmull-Clark	四边形	C^2	面	近似
Doo-Sabin	任何	C^1	顶点	近似
Loop	三角形	C^2	面	近似
Butterfly	三角形	C^1	面	插值
Kobbelt	四边形	C^1	面	插值
Reif-Peters	任何	C^1	新的	近似
Sqrt(3)(Kobbelt)	三角形	C^2	面	近似
Midedge	四边形	C^1	顶点	近似
Biquartic	四边形	C^2	顶点	近似

*在特殊点，光滑度一般少掉 1 次的连续性

虽然在业界还没有一个标准类型，但大多数 3D 动画和渲染包都支持细分表面作为图元。Catmull-Clark 和 Loop 细分是最常使用的，因为它们是最简单的（有争议性的），文档齐全，并非常适合实时渲染。

一个相关的议题是 PN 三角形[Vlachos00]，它提供了一种在渲染时用光滑图元取代三角形的方法。其基本思想是二次插值表面法线，类似于 Phong 着色，并利用其结果 3 次插值新的几何体。[Zorin00]很好地概括了细分。

细分方案的应用

对于互动游戏的生产工具链，使用细分表面的一种方法是利用高分辨率几何和纹理创造艺术品（几何可用任何支持细分方法的工具造模）。然后，美术资产作为高密度的多边形模型和相关数据被导出。接着，一些工具用相关的偏离细分贴图（displaced subdivision map）、纹理及动画数据将资产减少为低多边形数的网格。在运行时，基于速度、到相机的距离和硬件支持等，GPU 的细分内核（kernel）把资产动态地转换回所需要的多边形数，这允许不同的细分表面用在资产创建和资产渲染阶段，这样是有好处的。

一个这样做的工具是 ZBrush。为了在多分辨率层次中添加几何，它允许你用细分表面来编辑网格，然后把由此而来的高细节几何转化为低细节网格和位移贴图（displacement map）。

细分类别的选择

本精粹涵盖了 Loop 细分的实现[Loop87]。部分原因是：因为它是基于三角形的，这或许使它更容易在 GPU 上执行，大多数艺术家和工具都已经支持三角形网格，同时它又被很好地研究，并且可以产生很好看的表面。另一种较常见的选择是 Catmull-Clark 细分[Catmull78]，但它是基于四边形的，看起来并不适合游戏。顺便提一句，Pixar 使用 Catmull-Clark 细分来做角色的动画。本文的许多想法适用于基于四边形的细分以及其他方案。

5.4.2 Loop 细分的特征和选项

对一个封闭的三角形网格作用原始 Loop 细分算法，一个迭代会生成另一个有更多面的封闭三角形网格。重复使用迭代就会生成一个光滑的极限表面。需要扩展原始方法来造型更

多的功能。一个全功能的细分工具包括以下内容。

- 边界——允许非封闭网格。
- 折皱（Creases）——允许锐边（sharp edges）和表面折痕（ridge）。添加边界会生成折皱（从技术上讲，折皱应具有[Biermann06]中的技术，以防止微小的角落（corner）误差。但是，[Zorin00]认为这些误差在视觉上是轻微的。相对于原本提出及打算的一个适合实时渲染的技术，修正需要更多的计算）。
- 角落——可用于做尖状物（pointed items）。
- 半锐度——修改边界（boundary）、折皱和角落的基本规则，以得到不同的锐度。
- 颜色和纹理——渲染和游戏所需要的细分过程的易扩展性。
- 确切位置——经过一些细分，如果细分进行到了极限，顶点就能够被推到它们的最终位置。这种计算不是很昂贵。
- 确切法线——为着色计算精确的法线不是很昂贵，并且比面法线的平均还便宜。
- 位移映射——添加几何到被细分的表面，是一个非常想要的好功能，但是本精粹没有实现。不过读者可以参考[Lee00]和[Bunnell05]。
- 在任意点评估——允许在表面的任意位置计算极限表面[Stam99]，这对射线追踪和非常详细的碰撞检测相当有用，但是游戏渲染不太可能需要。
- 指定的法线——允许在极限表面的给定顶点上要求得到指定的法线[Biermann06]，并且它对造型有用。不过它的执行比本精粹中提出的要昂贵。由于篇幅的关系，我们省略这种方法的介绍。
- 多分辨率编辑支持——通过存储细分网格的所有层次，用户可以在细分的任一层次工作，使许多编辑功能更容易[Zorin97]。
- 碰撞检测——游戏动力学需要它；一种方法是在[DeRose98]中。
- 自适应细分（adaptive subdivision）——基于某些指标，将网格的某些部分细分成不同的数额，并且修补在此过程中形成的任何洞。自适应细分对维持低多边形数有帮助，同时还给出漂亮的曲线、轮廓和细节层次（level of detail）。通常，在网格的哪里细分是根据曲率决定的。

通过用参数标记顶点、面和边来控制细分算法，从而将特征添加到网格中。需要时，标记组合限制可在软件中执行，以防止退化的情况。

几何创作

为了实现边界、折皱、角落和半光滑特征，每个顶点和边用一个 $0 \leqslant w < \infty$ 的浮点权重来标记。0 权重表示标准的 Loop 细分，∞权重为无限尖折皱或边界。因为权重实际上是一个被影响到的细分层次的计数器，所以不需要在数据结构中编码无限远，任何大于进行了细分的最高层次的数字就足够了。例如，32 767 就足够了，因为任何网格都不太可能细分这么多次。

Loop 细分接受一个网格，通过把每个旧三角面分为 4 个新的面来创建一个新的网格，如图 5.4.5 所示。这需要两个步骤完成：第一步在每个现有的边插入一个新的顶点；第二步修改旧的顶点（不包括在边上插入的新顶点）。

这些几何规则大多数来自[Hoppe94a]和[Hoppe94b]，其中有些想法是综合[DeRose98]及

[Schweitzer96]得到的。

边

第一步用附近顶点的加权和在每边插入一个新的顶点，边的权重及其每个终端顶点的类型被用来对边进行分类。顶点类型在下一部分列出。

每个（非边界）边有两个相邻的三角形；新顶点的位置是 $\frac{3}{8}(v_0+v_1)$，其中 v_0 和 v_1 是待分裂边的顶点，另两个顶点是两个相邻的三角形的剩余顶点。图 5.4.1 对此进行了描述，其中，圆指三角形之间边上的新顶点。权重可写为 $(\frac{3}{8},\frac{3}{8},\frac{1}{8},\frac{1}{8})$，其中，位置 j 对应于顶点 j（0 一索引）。

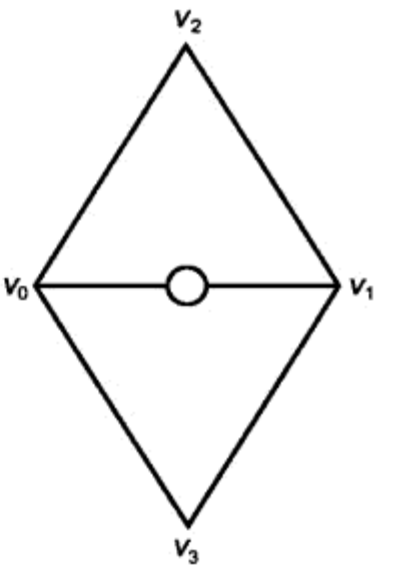


图 5.4.1 边遮罩

用来创建一个新边的权重取决于边权重和定义边的两个顶点（v_0 和 v_1）的类型。给定边上的两个顶点，表 5.4.2 显示了用哪种类型的权重来创建新的边顶点。权重如下。

- 1 型权重：$(\frac{3}{8},\frac{3}{8},\frac{1}{8},\frac{1}{8})$。
- 2 型权重：$(\frac{1}{2},\frac{1}{2},0,0)$。
- 3 型权重：$(\frac{3}{8},\frac{5}{8},0,0)$，这里，$\frac{3}{8}$ 的权重对应于角落边。

如果一个边的权重 $w=0$，那么它是光滑的。如果它的权重 $w\geqslant 0$，那么它是尖的。如果边的权重 $0<w<1$，那么新顶点在 $w=0$ 和 $w=1$ 的两种情况之间线性插补，维持终端顶点类型不变。

表 5.4.2 边遮罩的选择

	尖	正规折皱	非正规折皱	角落
尖	1	1	1	1
正规折皱	1	2	3	3
非正规折皱	1	3	2	2
角落	1	3	2	2

当一个边分裂后，每一个新边得到权重 $\tilde{w}=\max\{w-1,0\}$。因为先在一些层上作用折皱规则，然后作用光滑规则，而且在每一步都有可能插值，所以这对锐度有更精细的控制。一种可以有更多控制的选择是用一个权重标记边的每一端，每边给两个权重去插补新的边，然后再到处做出相应的改变。

注意，所有情况下的总权重数额为 1（顶点遮罩也是如此）。

顶点

顶点类型取决于顶点权重和入射边的类型。一个光滑顶点有 0 个入射尖边及权重 0；一

个尖顶点有 1 个入射尖边及权重 0；一个折皱顶点有 2 个入射尖边及权重 0；一个角落顶点有两个或两个以上入射尖边及权重 $w \geqslant 1$。如果一个内部折皱顶点有 6 个邻居，并且在折皱的每一端刚好有两个非尖边，那它是正规的。如果一个边界折皱顶点有 4 个邻居，那么它也是正规的。否则，折皱顶点和边界顶点是非正规的。如果一条边的权重 $0 < w < 1$，就顶点分类而言，这就足够称它为光滑了。

Loop 细分的第二步用原始顶点及其所有相邻顶点的加权和去修改所有原始顶点（而不是在步骤 1 插入每边的顶点）。

权重取决于周边顶点的数量 n。图 5.4.2 描述了光滑的顶点和尖顶点。b 的值通常是 $b_n = \frac{1}{64}\left(40 - \left\{3 + 2\cos\frac{2\pi}{n}\right\}^2\right)$，尽管文献中有其他的值（例如，[Warren95]建议对 $n > 1$，有 $b(n) = 3/(8n)$，同时 $b(3) = 3/16$，但是，这对于几个价态有无限曲率）。旧的顶点给予权重 $1 - b(n)$，每个旧邻居（而不是在步骤 1 中创建的顶点）给予权重 $b(n)/n$ 来确定新顶点的位置，也就是所有这些顶点的加权总和：

$$v_{\text{new}} = \left(1 - b(n)\right) \cdot v_{\text{old}} + \frac{b(n)}{n}\sum_{j=1}^{n} v_j \text{。}$$

对于角顶点，顶点的位置不移动，所以 $v_{\text{new}} = v_{\text{old}}$。

为于折皱顶点，新的顶点是原始顶点的 $\frac{3}{4}$ 加上折皱上两个邻居的各 $\frac{1}{8}$。

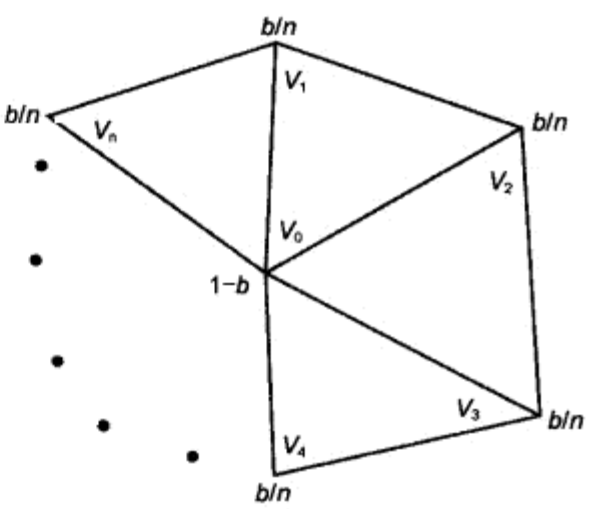


图 5.4.2　顶点遮罩

如果一个顶点有权重 $0 < w < 1$，新的顶点就在 w=0 和 $w = 1$ 的两种情况之间线性插值。一个新的顶点也有一个新的权重 $\tilde{w} = \max\{w - 1, 0\}$。

最后一种情况是，当一个顶点有权重 $0 < w_v < 1$，而且一些邻边有权重 $0 < w_e < 1$，从而促成许多可能的插值组合。在这个情况下，分别用权重 0 和权重 1 评估，在 w_v 插补，而不是在 4 种权重情形（0, 0），（0, 1），（1, 0）和（1, 1）间双线性插补。

另一种选择是要求整数权重，完全避免插值情形，但失去对半尖折皱的控制。

位移映射的表面是按照位移贴图，根据需要移动顶点来实现的。通过利用[Biermann06]来实现指定的法线来修改顶点，这也把折皱规则分为凸凹两种情况，避免了退化的情形。

极限位置

顶点可以投影到那些位置，即表面被细分无穷多次后它们将占据的位置。这通常在被作用几个细分层次后做。这一步是可选的，而且往往不修改表面。

极限位置 v^{∞} 是从当前顶点 v_0 和 n 个邻居顶点 v_j 的加权和计算得来。角顶点保持固定，即 $v^{\infty} = v_0$。光滑顶点用 $v_{\infty} = \frac{3}{8b(n)(n+1)}v_0 + \frac{1}{n+1}\sum_{j=1}^{n} v_j$ 来投影。一个正规的折皱使用权重

$(\frac{1}{6},\frac{2}{3},\frac{1}{6})$，这里，$v_0$对应于$\frac{2}{3}$。两个折皱邻居得到权重$\frac{1}{6}$，其他邻居获得权重 0 。同样，非正规折皱使用权重$(\frac{1}{5},\frac{3}{5},\frac{1}{5})$。

顶点及折皱法线

每个顶点的真实法线可以计算出来，但这应在对每个最终顶点计算极限位置之后。令人惊讶的是，这比通过平均每个相邻面法线来计算近似法线要快（分离到折皱的每一边）。

计算两个真实的切线，并且做一个叉积来计算在每个顶点的真实法线。

对一个光滑或尖顶点，两个切线是$t_1=\sum_{j=1}^{n}v_j\cos(\frac{2\pi\times j)}{n})$和$t_2=\sum_{j=1}^{n}v_j\cos(\frac{2\pi\times(j+1)}{n})$。

折皱顶点和边界顶点需要做更多的工作。对角落，没有为每顶点定义法线，但这对每个面来说是必需的。切线需要对折皱的每一边计算。沿着折皱（或边界），一个切线等于：-1×一个折皱邻居+1×另外一个折皱邻居。第二个切线的计算相对复杂，它的计算随后给出。对每个顶点，计算权重w_j，其中$j=0$是要计算法线的那个顶点。然后，其他索引从一个折皱到另外一个被编号为$j=0$，2，…，n。权重取决于顶点数。对每一种情形，如下：

$$(w_0, w_1, w_2)=(-2, 1, 1)$$

$$(w_0, w_1, w_2, w_3)=(-1, 0, 1, 0)$$

$$(w_0, w_1, w_2, w_3, w_4)=(-2, -1, 2, 2, -1)$$

对于$n\geqslant 5$，$w_0=0, w_1=w_n=\sin(z), w_i=(2\cos(z)-2)(\sin(i-1)z), z=\frac{\pi}{n-1}$。

这将从一个被标记的立方体创建超过 4 个的细分层次，其中一个面丢失并被标示为边界，如图 5.4.3 所示。注意，一些角落有不同的半锐利度。

功能实现

除了几何之外，一个完整的解决方案需要颜色、纹理和其他每顶点或每面的信息。

当添加新的顶点时，如颜色和纹理坐标值之类的面参数可用同样的细分方法来插值。一个简单的方法是：当旧的顶点被修改后，根据距离插值赋予新面新值。除非在特殊的地方，如沿着纹理坐标所形成的缝的边，许多特征可以每顶点细分。[DeRose98]有一些 Catmull-Clark 表面的细节介绍（但也适用于 Loop 面）。基本上，每顶点参数像顶点坐标一样被插值，因此，增加（u，v）纹理坐标就像把顶点作为（x，y，z，u，v）坐标一样简单。每顶点纹理不便于简单处理接缝。在这种情况下，每面纹理坐标较为有用。然而，所有在细分面上的内部点就成了每顶点参数。

由于篇幅缘故，本文没有覆盖的功能是适应性细化（tessellation）（只有需要曲率时，网格的一部分才被细分，如为了轮廓和裁剪等，并且确保不加入裂缝）和偏离细分面（通过使用“纹理”贴图去偏离计算出的顶点来增加几何体）。适应性细化在[Bunnell05]中涉及，同时，偏离细分面在[Bunnell05]和[Lee00]中涉及。

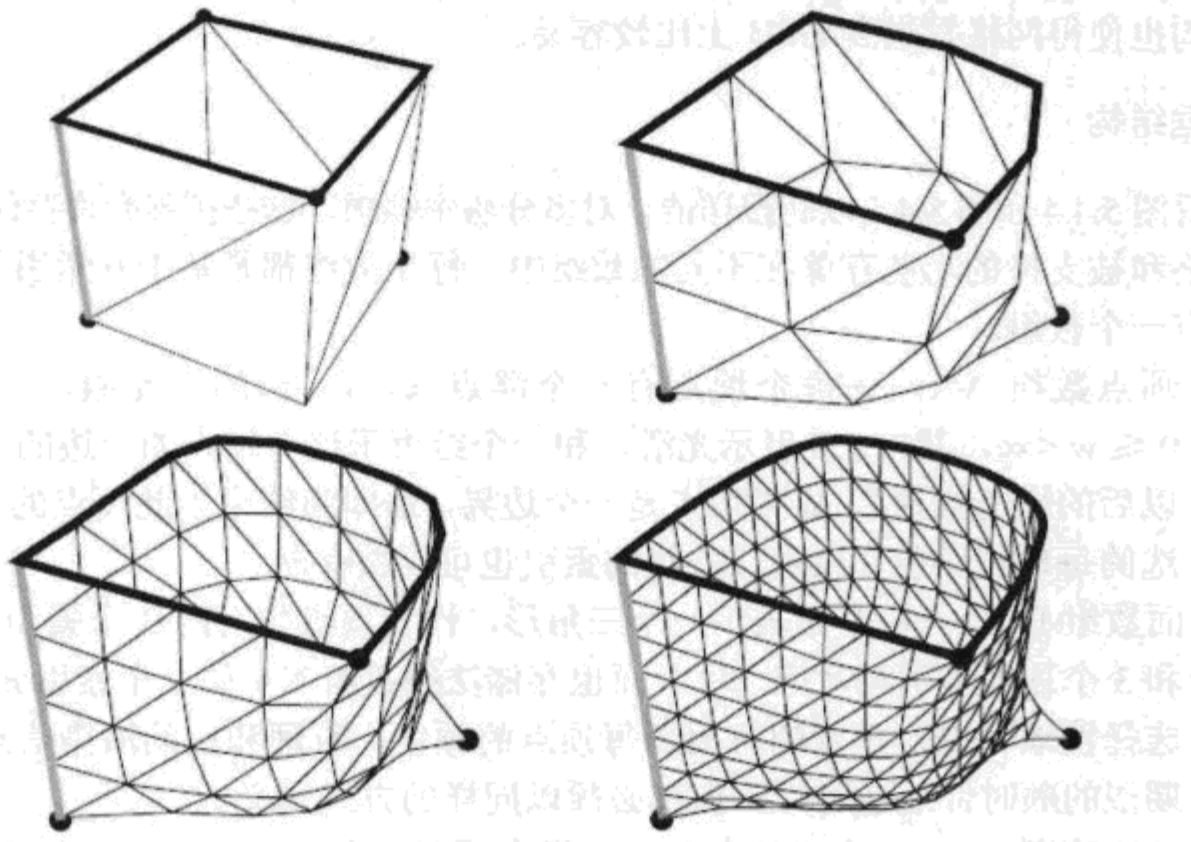


图 5.4.3　例子几何

碰撞检测

如果没有实现预定法线，那么曲面有凸包属性，也就是说，表面坐落在包围网格的凸包中。这可用于粗碰撞检测。更精确的（也是更昂贵的）细分表面之间的碰撞在[Wu04]中谈及，即使用一种新型的“区间三角形”紧紧包住极限表面。[Severn06]高效地计算了在任何分辨率下的两个细分表面的相交。这里将不会进一步谈及碰撞检测。

下一节，一个可包容在三角形网格上 Loop 细分的数据结构被提出。随后的一个算法将执行一个层次的细分，并返回一个新的网格。这个结构支持本精粹描述的大多数功能，并可扩展到许多其他功能。

5.4.3　细分数据结构

在文献中，有许多数据结构的方法被用来存储和操纵，包括半边、翼边、混合和格子在内的细分表面[Müller00]。对 Loop 细分，数据结构应允许轻易地寻找邻居顶点和入射边，并在每一层细分都保留这种能力。

设计数据结构时要考虑许多因素。把一个网格转换成 Loop 细分的网格是主要目标，从而得到某些特定的结构。但其他时候，最终目的是 GPU 渲染。在这些情况下，应该对数据结构进行优化。这里提出的方法是有些混合的，从而促成了一个可轻易移植到 GPU 上的数据结构。后面的部分会讨论移植到 GPU 上的性能问题。

下面的数据结构很容易从文件或其他地方读/写，快速的内部使用，而且不使用指针。避免指针会使移植到 GPU 或其他不像 C/C++般对指针友好的语言更容易。因为它不是显式地存储连接信息，而是从索引位置推导出来的，所以使内存占用小于前面提到的方案（有益于大

型网格工具)。因为所有项被安排成顶点数组，法线数组等，并且用索引来渲染多边形，所以这种结构也使得网格发送到 GPU 上比较容易。

数据结构

查看图 5.4.4 和图 5.4.5 以获得详情。对多分辨率编辑，这些扩展允许存储细分的所有层次。

网格和被支持的功能存储在不同的数组中。每个数组都是基于 0 索引的。对下面的每一项，都有一个数组。

- 顶点数组 VA——每个顶点有一个浮点 x，y，z 的三元组，一个浮点锐度权重 $0 \leqslant w < \infty$，其中，0 表示光滑，和一个结束于这个顶点的半边的半边索引 v_h（为了以后的快速查找)。如果顶点是一个边界，v_h 即为结束于此顶点的边界半边索引。可选的每顶点颜色、纹理或法线的索引也可以被存储。
- 面数组 FA——每个面表示一个三角形，作为顶点数组的 3 个索引 v_0、v_1、v_2 存储。和 3 个顶点索引相对应，每个面也存储法线数组 NA 的 3 个索引 n_0、n_1、n_2。每面可选择性地存储，想要的每面或每顶点的颜色、纹理和别的渲染信息。面的朝向可为期望的顺时针或逆时针，但都必须以同样的方式定义。
- 半边数组 HA——在半边数组中，每个面有 3 个（半）边，它们以相同的顺序存储。所以一个有索引 f 和（有序的）顶点索引 $\{v_0, v_1, v_2\}$ 的面，有索引为 $3f$、$3f+1$ 和 $3f+2$ 的有序半边，分别表示从 v_0 到 v_1、v_1 到 v_2 和 v_2 到 v_0 的半边。注意，半边是有向边，一对（pair）边的两个半边有相反的方向。一个半边条目（entry）有两个值：一个表示对的半边索引的整数（如果是边界的话，此值为-1）和一个表明折皱值浮点尖锐度权重 $0 \leqslant w < \infty$，0 表示光滑，较大的值表示锐利。每个匹配的半边对必须具有相同的折皱值，以避免模棱两可。注意，一个半边索引确定了相应的面索引，而这又决定了有向半边的开始和结束顶点。
- 法线数组 NA——法线可以用有不同权衡的许多方法包含在方案中。为了干净利落地处理折皱、边界和半尖锐特征，你需要一个每顶点每面的法线，但是许多顶点（例如，光滑和正规的）只需要一个法线。一个法线数组就能处理这个；其每个元素都有一个单位向量和权重 $0 \leqslant w < \infty$。权重说明了一个顶点法线趋向于指定的法线有多快，0 意味着没有指定的法线。法线根据索引引用，从而避免了多余的存储。

除了存储每个数组的大小外，还存储边数 E（这里，一个匹配的半边对或边界边构成一个边)。如果网格没有边界（E = 半边数 /2 = 面数 ×3/2)，那么这不是过于昂贵的计算，并且可以通过扫描半边数组及让 E =（半边数组的大小 + 在半边数组中边界边的数）量/2 来计算。

考虑到速度，顶点类型的信息（光滑、折皱等）也可存储在顶点标记中。其他条目（item）也可进行标记，但需要修改这里描述的算法，以保持细分之间的不变性。

我们可以丢弃不需要的功能，如每面的 3 条法线、指定的法线或半尖折皱，但这会失去精细的粒度控制。

一个构件良好的网格需要一些规则。如果一个半边不配对（它是在一个边界上)，那么它有对索引 -1，而且必须有无限折皱权重，否则边将缩小。同一边的每个半边必须有相同的

权重，否则那些边将会被不同地细分，从而造成裂缝。

文件格式

基于描述的数据结构，这里定义了一个作为对流行的基于文字的 Wavefront 的 OBJ 文件格式的扩展。每一个条目是一行文字，以指定线类型的记号开始，其次是空格分离的字段。不同的数据块被存储来加快载入，以致所有的项（如成对边）不需要在每次被加载时都重新计算。以文件读/写为顺序的格式被列在表 5.4.3 中。

表 5.4.3　　文件格式条目

条　　目	说　　明
# SubdivisionSurfL 0.1	指定一个非标准的 OBJ 文件，有版本
si v f e n	可选的给定顶点数、面数和法线的细分信息。允许数组的预分配
v x y z	每顶点一条浮点位置
f v1 v2 v3	每面一条的基于 1 的顶点索引，有向
hd j wt	按半边顺序 $v_0 \to v_1$、$v_1 \to v_2$、$v_2 \to v_0$ 的半边数据，按面描述的顺序每半边一条。每条是基于 1 的整数边对索引 j（对边界，1）和一个浮点权重 wt
fc r1 g1 b1 a1 r2 g2 b2 a2 r3 g3 b3 a3	可选面颜色 RGBA，每面一个，在区间[0, 1]的浮点。不允许有每顶点颜色 *fc*
vc r g b a	可选的每顶点颜色数据 RGBA，在区间[0, 1]的浮点。不允许有每面颜色 *fc*
ft u1 v1 u2 v2 u3 v3 texname	可选的面纹理坐标（u, v），在区间[0, 1]的浮点。texname 取决于应用本身
fn nx ny nz w	可选的法线数据，带指定法线的权重。 0 是默认权重
fni n1 n2 n3	可选的在法线数组中的面法线索引，每顶点一个法线。需要 *fn* 项
vs wt	可选的顶点锐度，在[0, ∞]之间， 0 为默认光滑；每个顶点一项

5.4.4　细分算法的细节

这是 Loop 细分算法的一个概述。让 V = 旧顶点数，F = 旧面数，$H = 3F$ = 半边数，$\#E$ = 边数 =（H + 边界边数）/2。

一个层次的细分包括 6 个步骤：

（1）计算新的边顶点；

（2）更新原始顶点；

（3）分裂面；

（4）创建新的半边信息；

（5）更新其他特征；

（6）用新的数组取代数据结构中的数组，并丢弃、存储，或按预期那样释放旧的。

这些步骤将在下列各部分中详细说明。

计算新的边顶点

按照下列步骤可计算新的边顶点。

（1）因为每一个现有顶点将很快被修改（原始的需要保留，直到所有的工作完成），且因为新的顶点将按每边相加，所以需要为大小为旧顶点数 + 旧边数的所有新顶点分配一个数组 *NV*。当创建新的边顶点，在数组中的前面 V 个位置被跳过，以至于原始的顶点在修改后

能放回相同的位置。

（2）分配一个整数数组 EM（边图），大小等于半边数，来存储映射半边到新顶点索引的索引。初始化所有的为−1 来指明半边还没有被映射。

（3）对每个半边 h，如果 $EM[h]=-1$，用边分裂规则在边上插入一个顶点。把新的顶点存储在 NV 中的，超过原始 V 个位置的一个未用槽中，并且在 $EM[h]$中存储生成的 NV 索引。如果 $h_2=E[h]$不是−1，那么 h 有一个成对的半边 h_2，因此，也在 $EM[h_2]$中存储 NV 索引。

更新原始顶点

现在，你必须把每个原始顶点移动到一个新的位置，以与之前一样的顺序和位置将新的顶点放置在新的顶点数组 NV，使面分裂容易些。这是利用之前的顶点修改规则来做到的。在更新时，顶点权重减少 1，钳制（clamp）在 0，这样，新顶点的权重就是 0。每个顶点存储一个结束于此顶点的半边索引 v_h，这可用来快速步行邻近的顶点，并确定边类型，如图 5.4.4 所示。给定一个结束于该顶点的半边索引 h，相连的邻居顶点是 VA[FA[Floor[h/3]].vertexIndex [h mod 3]]。给定 e_A，下一个感兴趣的半边可用 $e_B=EA[e_A]$和 $e_c=3Floor\left[\frac{e_B}{3}\right]+((e_B+2)\bmod 3)$ 来找到。有了这个信息，就可以迅速查询边及邻居顶点。

需要用一个入射折皱来标识边界顶点的原因是：这样遍历只需要沿一个方向，从而使代码更简单。

更新所有信息之后，让所有顶点（新的和旧的）都有半边索引−1，这意味着没有入射匹配的半边，这些将会在面分裂时填充。

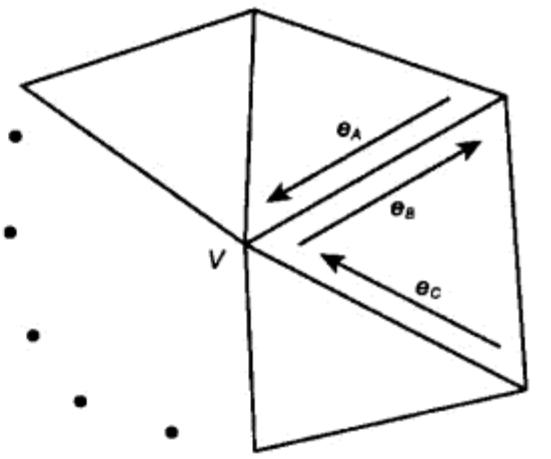


图 5.4.4 顶点邻居

分裂面

每个旧面都将成为 4 个新的面，分裂如图 5.4.5 所示。图 5.4.5 显示了原始的有边和面朝向的三角形，以及这如何映射到新的边和面朝向和新面被存储的秩序（0，1，2，3）。

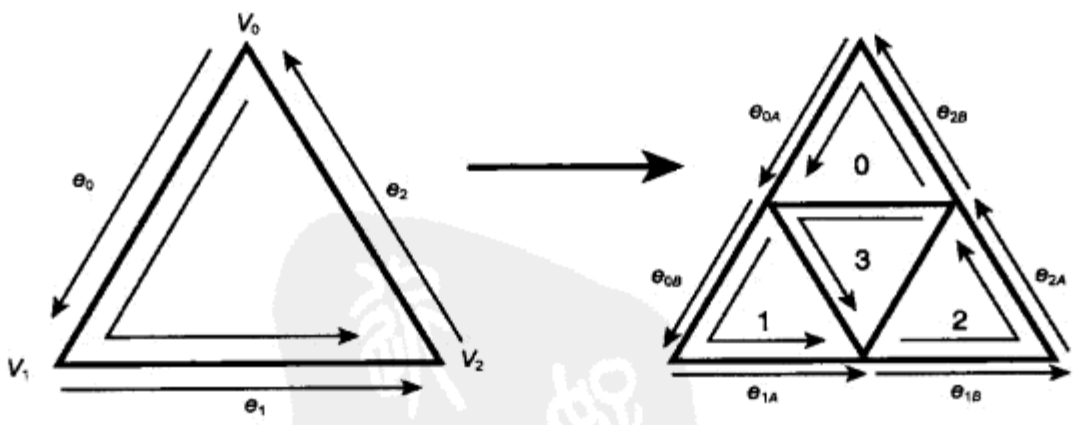


图 5.4.5 面分裂

（1）对大小为 $4F$ 的新面分配一个数组 NF。

（2）对于有顶点索引 v_0、v_1、v_2 的每个面，用 $j_0=NV[3f+V]$、$j_1=NV[3f+1+V]$和 $j_2=NV[3f+2+V]$

查找 3 个边顶点的索引。

（3）按图 5.4.5 的顺序分裂面。对 NFm，在位置 4*f*、4*f*+1、4*f*+2 和 4*f*+3 上，添加面{j_2，v_0，j_0}，{j_0，v_1，j_1}，{j_1，v_2，j_2}，{j_2，j_0，j_1}。这个顺序很重要。在概念上，每个父半边 e_k 被分裂成 e_{kA} 和 e_{kB} 两个子半边。

（4）在面分裂时，用一个此顶点的入射半边的索引来标记每个顶点（其中还有一个从以前步骤得来的−1 标记），其中，入射边界优先。

创建新的半边信息

这一步可以与面分裂那一步合并，但为了清晰起见，我们把它分开，这时需要一个新的而且已经正确匹配和加权的半边链表。创建一个大小为 $12 \times F$ 的新的半边数组 *NE*（每个新面 3 个，每个旧面变成 4 个新面，即产生 12 个）。

定义一个函数 nIndex(j,type)来计算新半边对的索引，这里，j 是旧半边的索引，类型是 0 = A 或 1 = B，标明新半边的哪个部分正在被匹配。此函数如下：

```
function nIndex(j, type)
    /* 索引偏移的数据表-匹配新半边 */
    offsets[]= {3, 1, 6, 4, 0, 7}
    /* 原始半边对索引 */
    op = EV[ei]
    if(op == -1)
        return -1 /* 边界边 */
    /* 配对运算(pair op)的面的分裂边的新位置 */
    bp = 12*Floor[op/3]
    /* 返回匹配的新索引 */
    return bp + offsets[2*(op mod 3)+ type]
```

数组{3，1，6，4，0，7}来自和邻居半边匹配的半边，它依赖于在所示的数组中插入新条目。对于每个原始面索引 *f*，做如下步骤。

（1）让 *b*=12**f* 为一套新半边的基半边索引，存储在数组 *NE* 从 *b*～*b*+11 的 12 个索引中。

（2）以顺序{e_{2B}，e_{0A}，*b*+9，e_{0B}，e_1A，*b*+10，e_{1B}，e_{2A}，*b*+11，*b*+2，*b*+5，*b*+8}在 *b*，*b*+1，…，*b*+11，存储 12 个新半边对索引，这里，e_iT 是 nIndex(e_i,type)，其中 e_i 是边索引。对 $T = A$ 有 type = 0，对 $T = B$ 有 type = 1。它们按每面 3 个分组，顺序是图 5.4.5 中面被创建的顺序。

（3）在前面的 12 条中更新半边权重。子半边的权重为父半边的权重减 1，钳制在 0。没有父亲的新半边权重设为 0。

更新其他特征

在创建边顶点和用简单插值给顶点重定位时，每顶点颜色和每顶点纹理坐标可被更新。每面每顶点颜色和纹理坐标可在以前的步骤中插值完成，或在最后一步完成。

通过用位移贴图修改当前顶点位置，偏离细分面的修改也可以在这里实现。

如果曲面即将被渲染，那么临时法线可以用法线平均技术或用确切值来计算。精确法线对于有顶点在极限点的曲面更合适。一般直到渲染时才需要计算法线。

最后一步

最后一步是用新的数据取代数据结构中的数组，并丢弃、存储或按计划释放。

最后，如果网格不会被进一步细分，顶点可被推至极限位置，并且，真正的法线可以被计算。在物体将要被绘制的一个渲染引擎中，这是一个适当的步骤。

5.4.5 性能问题

本节将概述该算法的硬件渲染技术。

性能改善

该算法本身存在很多可以改善其性能的地方，尤其是如果并非需要所有的特征时。如果你需要的是一个简单、光滑和封闭的网格，那么可以去除所有的特殊情况，使细分非常快。

考虑如下这些实现技巧。

- 对基于 $b(n)$的权重、切线权重、法线权重、极限位置权重和其他任何项，使用表。一个给定网格有最高价态的顶点，并且所有新顶点的价态都会比那小，这就让使用表成为可能。
- 让半边数组每面 4 项而不是 3 项间隔，让更多的除以 3 和余 3 运算用移位操作（Shift）取代。这是传统的空间换时间的权衡。
- 大多数内部顶点将有价态 6，并且是光滑的，因此让这些代码快些，而对其他情况特殊处理。大多数边界顶点将是正规的，并且有价态 4。大多数边将有权重 0，并且跟价态 6 的光滑顶点连接。
- 标记边和顶点，看它们是否光滑，或是否需要特殊情形的代码，以允许更快的决策，而不是通过步行邻居来确定顶点和边类型。一旦顶点或边类型被决定下来，很容易标记子项。
- 预计算一个层次的划分来隔离特殊情况的顶点。因为有较少的情形，在运行时可用一个简单版本的算法。这个是小的速度提高，被用于某些硬件的实现。
- 像一个半边结构的一个基于指针的数据结构可以加快细分，但使用更多内存（可能不太连续），并且使读/写更难。直到你做一些测试，才会知道这是否会更快。
- 在可能的情况下，把每顶点每面参数变成每顶点。例如，因为邻居面需要沿折皱的不同法线，所以折皱要求每顶点每面法线，但是一旦面已细分，内部光滑的新顶点就可以（而且应该）使用每顶点法线。
- 如果想要，你可以在每一步骤做多个细分，只存储生成的三角形，而不是更新所有的连接信息。这在 GPU 部分中详解。
- 实现适应性细分。使得在一些步骤后有更少数量的三角形，这将很大程度地提高速度（但将打破作用在所述数据结构上的简单算法）。

GPU 细分和渲染

当我在回顾文献并意识到现代 GPU 细分渲染技术过期得有多快时，我就放弃了我原来想回顾这些技术的计划。相反，这里的重点是把细分方案的统一规则放在一个地方，这将有

助于未来的硬件和软件的实现，使这个精粹能在一个较长的时期内有用。

对于 GPU 渲染，下面是一个对几篇论文的回顾（按时间顺序）。大部分文章可在 Internet 上找到。论文大致同程度地覆盖了 Catmull-Clark、Loop 和一些普遍的方法。

- [Pulli96]提出了一种有效的 Loop 渲染方法，它的工作原理是在预计算阶段将三角形配对，有效地传递平方值和一个 1- 邻居给渲染函数，然后将两个三角形渲染到一个任意的细分深度。
- [Bischoff00]提出了一种对内存非常有效并且快速的 Loop 渲染方案，主要的理念是使用多变量前向求差（forward differencing）来生成几个深细分层次的三角形，而不需要生成中间层。渲染是一个面片接着一个面片完成的。
- [Müller00]提出了一个对[Pulli96]的扩展，并详细说明了一个三角形解析算法和一个滑动窗口的方法，还详细介绍了适应性细分及防止裂缝。
- [Leeson02]覆盖了几个细分方法，并概述了一些渲染技巧，如层次性背面剔除。
- [Bolz02]实现了 Catmull-Clark 细分，用一个静态数组存储结果。该方法适合于单指令多数据（SIMD）的执行。
- [Bolz03]在 GPU 上实现了 Catmull-Clark 细分，特别考虑到裂缝避免和由浮点误差造成的像素丢失。
- [Boubekeur05]提出了一种适合渲染许多种细分表面的通用方法，其主要思想是在 GPU 上实现一个“完善模式”。传递给 GPU 的每个三角形或其他图元用此模式完善。
- [Bunnell05]和[Shiue05]都在 GPU 上实现了 Catmull-Clark 细分，有足够的细节。

快速细分表面渲染

一种适合 GPU 的快速细分程序基于以下的观察。对于每个三角形，在细分后，新的项目（顶点、边、面、颜色等）是三角形的 1- 邻居的一个线性组合。第二个细分项目是第一个细分项目的线性组合，因此也是原始邻居的线性组合。这个特征在前面的文献中已经被不同地应用到，下面将用一个简单的情形来解释。

一个面片是一个三角形 T 和其周围的三角形（那些影响三角形 T 的子三角形）。如图 5.4.6 所示的面片说明——在左侧，T 被着色及包括一个 1- 邻居；右侧显示 T 被细分一次后和新的 1-邻居（不含所有画的边线）。

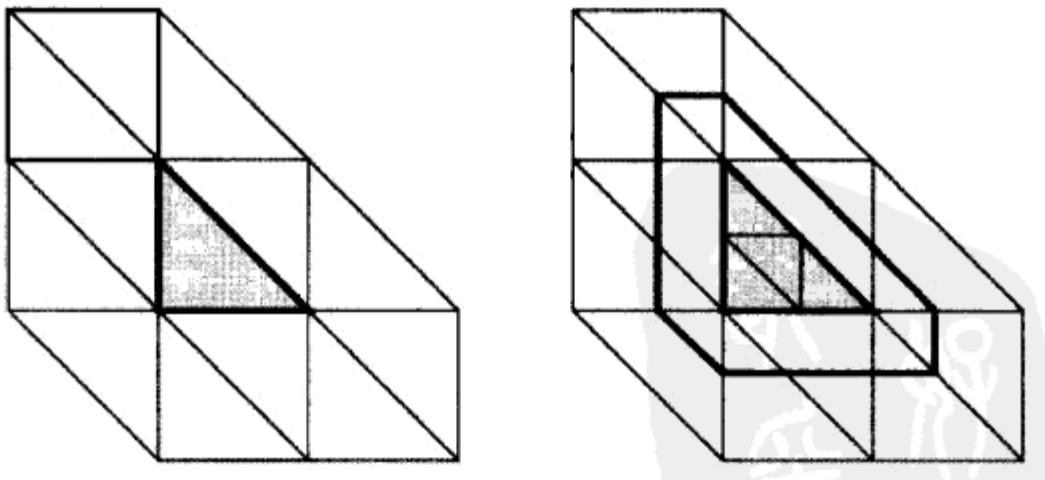


图 5.4.6　细分一个面片

暂时假设没有折皱或边界（以后可被添加上），所有在 T 以下的细分层可以用现有的顶

点线性组合生成。因此对每个想要得到的细分层次，一个遮罩可以通过周边顶点计算出来，而这些顶点输出在 T 以下的所有三角形，而无须计算中间层。连接信息也不必计算或存储——只需要面的顶点，它们自然地成为一个网格阵列，并适合 GPU 渲染。

网格预计算收集每个面片所需的数据，按每个三角形存储。在渲染时，一个细分层次被选中，并且将每个面片传递到 GPU 内核。然后，GPU 内核在一个通道中，用低分辨率三角形创建细分三角形，并渲染生成的三角形。为了用上本精粹的所有特征，应该实现不同的内核。另外，预处理可以简化事例数量，促成更少的 GPU 内核变化。

最后一点，这种方法可能会导致像素丢失或裂缝，因为邻居三角形可能使用不同顺序的浮点运算来评估。对 Catmull-Clark 表面，[Bolz03]解决了这个问题。

5.4.6 结论

本文讲述了如何实现有额外特征的 Loop 细分表面的细节，并且提供了一个关于细分表面文献的起点。几何特性（如折皱、边界、半尖锐的物体和法线）以及曲面的标识（像颜色和纹理）都得到了覆盖。今后的方向是在算法中增加偏离细分表面和适应性细分。

5.4.7 参考文献

[Biermann06]Biermann, Henning, Levin, Adi, and Zorin, Denis. "Piecewise Smooth Subdivision Surfaces with Normal Control," Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, pp. 113–120, 2006.

[Bischoff00]Bischoff, Stephan, Kobbelt, Leif, and Seidel, Hans-Peter. "Towards Hardware Implementation of Loop Subdivision," Eurographics SIGGRAPH Graphics Hardware Workshop, 2000 Proceedings.

[Bolz02]Bolz, Jeffery, and Schröder, Peter. "Rapid Evaluation of Catmull-Clark Subdivision Surfaces," Web3d 2002 Symposium.

[Bolz03]Bolz, Jeffery, and Schröder, Peter. "Evaluation of Subdivision Surfaces on Programmable Graphics hardware."

[Boubekeur05]Boubekeur, Tamy, and Schlick, Christophe. "Generic Mesh Refinement on GPU," ACM SIGGRAPH/Eurographics Graphics Hardware, 2005.

[Bunnell05]Bunnell, Michael. "Adaptive Tesselation of Subdivision Surfaces with Displacement Mapping," GPU Gems 2, 2005, pp. 109–122.

[Catmull78]Catmull, E., and Clark, J. "Recursively Generated B-Spline Surfaces on Arbitrary Topological Meshes," Computer Aided Design 10, 6(1978), pp. 350–355.

[DeRose98]DeRose, Tony, Kass, Michael, and Truong, Tien. "Subdivision Surfaces in Character Animation," International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1998, pp. 85–94.

[Hoppe94a]Hoppe, Huges. "Surface Reconstruction from Unorganized Points," PhD Thesis, University of Washington, 1994.

[Hoppe94b]Hoppe, Huges, DeRose, Tony, DuChamp, Tom, et. al. "Piecewise Smooth Surface Reconstruction," Computer Graphics, SIGGRAPH 94 Proceedings, 1994, pp. 295–302.

[Lee98]Lee, Aaron, Sweldens, Win, et. al. "MAPS: Multiresolution Adaptive Parameterization of Surfaces," Proceedings of SIGGRAPH 1998.

[Lee00]Lee, Aaron, Moreton, Henry, and Hoppe, Huges. "Displaced Subdivision Surfaces," SIGGRAPH 2000, pp. 95–94.

[Leeson02]Leeson, William. "Subdivision Surfaces for Character Animation," *Game Programming Gems 3*, 2003, pp. 372–383.

[Levin99]Levin, Adi. "Interpolating Nets of Curves by Smooth Subdivision Surfaces," Proceedings of SIGGRAPH 99, Computer Graphics Proceedings, Annual Conference Series, 1999.

[Loop87]Loop, Charles. "Smooth Subdivision Surfaces Based on Triangles," Master's Thesis, University of Utah, Dept. of Mathematics, 1987.

[Müller00]Müeller, Kerstin, and Havemann, Sven. "Subdivision Surface Tesselation on the Fly Using a Versatile Mesh Data Structure," *Comput. Graph. Forum*, Vol. 19, No. 3, 2000.

[Pulli96]Pulli, Kari, and Segal, Mark. "Fast Rendering of Subdivision Surfaces," Proceedings of 7th Eurographics Workshop on Rendering, pp. 61–70, 282, Porto, Portugal, June 1996.

[Schweitzer96]Schweitzer, J.E. "Analysis and Application of Subdivision Surfaces," PhD Thesis, University of Washington, Seattle, 1996.

[Severn06]Severn, Aaron, and Samavati, Faramarz. "Fast Intersections for Subdivision Surfaces," In International Conference on Computational Science and its Applications, 2006.

[Shiue05]Shiue, L.J., Jones, Ian, and Peters, Jörg. "A Realtime GPU Subdivision Kernel," ACM SIGGRAPH Computer Graphics Proceedings, 2005.

[Stam99]Stam, Jos. "Evaluation of Loop Subdivision Surfaces, " SIGGRAPH 99 Course Notes, 1999.

[Vlachos00]Vlachos, Alex, Peters, Jörg, Boyd, Chas, and Mitchell, Jason. "Curved PN Triangles," ID3G 2001.

[Warren95]Warren, J. "Subdivision Methods for Geometric Design," Unpublished manuscript, November 1995.

[Wu04]Wu, Xiaobin, and Jörg Peters, "Interference Detection for Subdivision Surfaces," *EUROGRAPHICS*, 2004. Vol. 23, 3.

[Zorin96]Zorin, Denis, Schröder, Peter, and Sweldens, Wim. "Interpolating Subdivision for Meshes with Arbitrary Topology," Proceedings of SIGGRAPH 1996, ACM SIGGRAPH, 1996, pp. 189–192.

[Zorin97]Zorin, Denis, Peter Schröder, and Wim Sweldens. "Interactive Multi-Resolution Mesh Editing," CS-TR-97-06, Department of Computer Science, Caltech, January 1997.

[Zorin00]Zorin, Denis and Schröder, Peter. "Subdivision for Modeling and Animation," Technical Report, ACM SIGGRAPH Course Notes 2000.

5.5 用径向基函数纹理来替代动画浮雕

Vitor Fernando Pamplona，Instituto de Informática: UFRGS
vfpamplona@inf.ufrgs.br

Manuel M. Oliveira，Instituto de Informática: UFRGS
oliveira@inf.ufrgs.br

Luciana Porcher Nedel，Instituto de Informática: UFRGS
nedel@inf.ufrgs.br

为了达到实时性能，游戏通常使用场景元素的简化表示。例如，用法线贴图扩展的简单多边形模型和精心设计的纹理被用来制作令人难忘的情景[IdSoftware]；同时，公告板（billboard）和替代体（imposter）被用来代替远处的物体。最近，浮雕纹理[Oliveira00]（即基于每纹素的，含有深度及法线数据的纹理）已经被用来创建详细 3D 物体的替代体。每一个浮雕纹理使用一个四边形，并且保留自遮挡、自身阴影、运动视差和物体轮廓[Policarpo06]。

浮雕渲染利用深度和表面的法线信息去着色每个片段（fragment），从而模拟几何表面细节的外观。这是在 2D 纹理空间，完全在 GPU 上进行光线－高度－场的相交来获得[Policarpo05]。浮雕细节到一个多边形模型的映射是用传统的方法来完成的，即给模型的每个顶点都指定一对纹理坐标。浮雕替代（relief impostor）是通过映射浮雕纹理到四边形上而得到的，这些纹理含有多层次的，基于每纹素的深度、法线和颜色数据[Policarpo06]。

5.5.1 简介

通常，纹理可以用来表示静态和动画的物体，传统上，基于纹理的动画用像图像扭曲（image warping）或一套静态纹理周期性地映射到一些多边形上的技术。虽然传统的图像扭曲技术局限于一些平面变形，但是，第二种办法需要和动画序列帧数一样多的纹理，这常常需要大量的艺术工作。

本精粹描述了一项替代动画浮雕的新技术，它是基于单个多层的，利用径向基函数（Radial Basis Function，RBF）的浮雕纹理。此技术保留了浮雕替代的特性，可以让观察者在动画时看到遮挡及视差的变化。图 5.5.1 显示了从狗浮雕替代创建的、3 帧狗行走的动画序列，注意腿的位置变化。

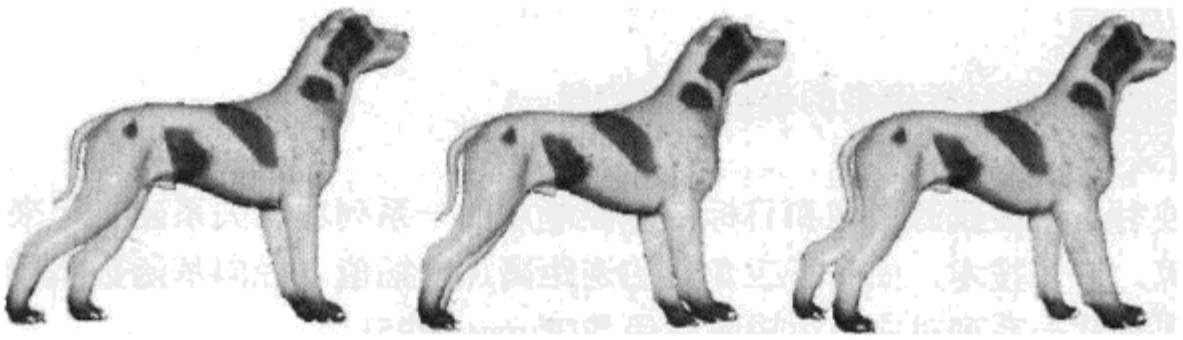


图 5.5.1　通过扭曲一个浮雕替代创建而成的 3 帧狗行走动画，注意腿的位置变化

为了制作这些动画，在预处理步骤中，用户在浮雕替代的纹理上指定一系列的控制点。在 2D 空间，移动这些控制点会扭曲纹理，从而把被表示的物体带到新的姿势。在动画过程中，这样的姿势是来插补的主要姿势。注意，这些姿势只是由那些控制点及一个单一纹理来隐性地表示，如图 5.5.2 所示。

作为预处理的一部分，为了得到动画中期望的帧数，这个算法也插补这些控制点的位置，对于其中的每一帧，需要求解一个线性系统来获得了一系列的 RBF 系数。这些控制点及其相应的 RBF 系数，定义了一系列产生实际动画的变形函数。由于效率的原因，这些控制点和系数被存储在一个纹理中（通常是 16×16 或 32×32 纹素）。在运行时，这些数据用来在 GPU 上重建动画。

提出的技术可用来动画，基本上任何类型的，基于纹理的表示，如浮雕纹理[Oliveira00]、用法线映射的广告牌和位移贴图[Cook84]。注意，用可以在 GPU 上评估的和任何描述了所期望转换的其他方法来取代 RBFs 也是可行的。对进行重复运动的、有生命的和移动的物体，提出的技术可生成实时的、有真实感的动画。

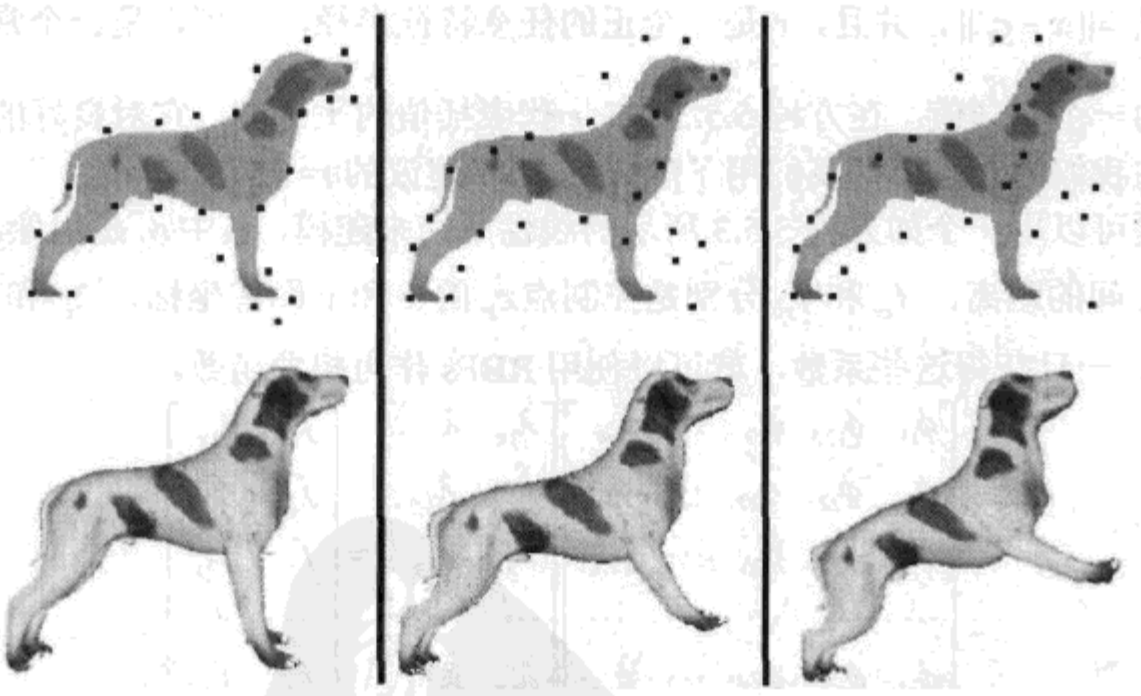


图 5.5.2　放在浮雕替代的纹理上（上面一行）的控制点（暗点）扭曲了纹理，并改变被渲染的狗的位姿（下排）。所有姿势被一个单一纹理及一系列控制点隐性地定义

5.5.2　图像扭曲

为了计算序列中的每一帧，基于扭曲的纹理动画在整个源图像上评估一个函数。给定一

个源图像，通过计算每个源像素的新坐标，一个扭曲函数就产生一个输出图像。因此，图像扭曲包括两个步骤。

- 关联源像素和目标像素的坐标映射阶段。
- 重新采样阶段。

通常，映射是用涉及源图像和目标图像控制点的一系列对应关系建立起来的一个全局分析函数来计算。许多技术，如基于三角形的逆距离加权插值、径向基函数和局部有限的径向基函数，可用来从一系列对应点生成映射函数[Ruprecht95]。

5.5.3 径向基函数

径向基函数是一种用于产生多元逼近的数学方法，并且是其中一个最流行的、用来插值分散数据的选择[Buhmann03]。在计算机图形学领域，RBFs 已被用于从点云的表面重建[Carr01]、图像扭曲[Ruprecht95]和动画[Noh00]。方程 5.5.1 定义了一种 RBF。

$$f(x)=\sum_{i=1}^{N}\lambda_i\phi(\|x-c_i\|) \tag{5.5.1}$$

这里，N 是中心点的数量，ϕ是一个基函数，λ_i 是 RBF 表示法的第 i 个系数，C_i 是第 i 个中心点，x 是将要被函数评估的一个点。对于图像扭曲，c_i 代表控制点的像素坐标，x 代表图像中任何像素的像素坐标。在这种情况下，一个不错的ϕ是最先由 Hardy[Hardy71]提出的多二次（multiquadrics）[Hardy71]。

$$\phi(d_i)=\sqrt{r^2+d_i^2} \tag{5.5.2}$$

其中，$d_i=\|x-c_i\|$，并且，r 是一个正的任意特征半径，它可以是一个常量，或是对每个控制顶点的一个不同值。在方程 5.5.2 中，r 代表插值的平滑度，它对良好的图像扭曲结果至关重要。在我们的实验中，我们用了[Ruprecht95]建议的 r=0.5。

扭曲问题可以用一个如方程 5.5.3 所示的线性系统来建模，其中 ϕ_{ij} 是用像素坐标表示的，控制点 c_i 与 c_j 间的距离。f_{kx} 和 f_{xy} 分别是控制点 c_k 的 x 和 y 图像坐标。λ_{kx} 和 λ_{ky} 是需要求解的 RBF 系数。一旦获得这些系数，就可以使用 RBFs 作为扭曲函数。

$$\begin{bmatrix}\phi_{11}&\phi_{12}&\phi_{13}&\cdots&\phi_{1m}\\ \phi_{21}&\phi_{22}&\phi_{23}&\cdots&\phi_{2m}\\ \phi_{31}&\phi_{32}&\phi_{33}&\cdots&\phi_{3m}\\ \cdots&\cdots&\cdots&\cdots&\cdots\\ \phi_{n1}&\phi_{n2}&\phi_{n3}&\cdots&\phi_{nm}\end{bmatrix}\begin{bmatrix}\lambda_{1x}&\lambda_{1y}\\ \lambda_{2x}&\lambda_{2y}\\ \lambda_{3x}&\lambda_{3y}\\ \cdots&\cdots\\ \lambda_{nx}&\lambda_{ny}\end{bmatrix}=\begin{bmatrix}f_{1x}&f_{1y}\\ f_{2x}&f_{2y}\\ f_{3x}&f_{3y}\\ \cdots&\cdots\\ f_{nx}&f_{ny}\end{bmatrix} \tag{5.5.3}$$

5.5.4 插值扭曲函数

给定用户分别在时间 t 及（$t+k$）指定的，两个关键姿势的两套控制点 S_t 和 S_{t+k}，中间

姿势的 RBF 系数可以通过插值在 S_t 和 S_{t+k} 相对应的控制点对的坐标和解方程 5.5.3 得到插值的 λ_s 来得到。为了得到一些光滑的插值，我们用了一个 3 次 Hermite 样条，其中，切线的端点用向量 $0.5(S_t + S_{t+k})$ 指定，如图 5.5.3 所示。当使用法线贴图时，同样的扭曲办法也应该用于法线贴图。因此，对每一帧，两个纹理都必须使用相同的 RBF 来评估。

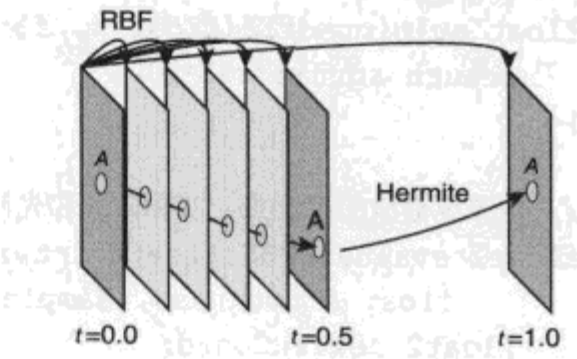


图 5.5.3　在时间 0.0 和 0.5 之间的浅灰色帧使用了 3 次 Hermite 样条去插值控制点

5.5.5　使用着色器评估扭曲函数

现代的 GPU 可以执行叫做着色器的程序。因为扭曲函数需要对每个纹素都执行，显然，RBF 应该在片段着色器上被评估。为此，需要倒置（invert）扭曲函数，因为，给定片段 f，你必须能够获取，在扭曲转换下的、被映射到 f 的纹理坐标。幸运的是，用方程 5.5.3 来倒置扭曲函数只需要两个步骤。

- 用当前（期望的）姿势的控制点的坐标计算 ϕ_{ij}。
- 把未修改的（在移动前）控制点的 x 和 y 坐标作为 f_{kx} 和 f_{xy}。

如图 5.5.2 的示例，用于渲染底部中间图像的 RBF 系数计算如下：ϕ_{ij} 是所示顶部中间图像中，控制点 c_i 与 c_j 之间的距离，其中，f_{kx} 和 f_{xy} 是所示左上角第 k 个控制点的坐标。注意，图像扭曲操作需要的第二步，即重新取样，在纹理过滤硬件上是免费的。

如前所述，你在着色器运行时读取的一个纹理中存储 RBF 数据（控制点坐标和λ值）。这个纹理的第 j 行代表了动画时的第 j 帧。第 i 个纹素的 RGBA 通道分别存储 c_i 的（x，y）坐标和它的 λ_{ix}、λ_{iy} 系数（见图 5.5.4）。因为λ的值有可能不在[0, 1]范围内，所以我们使用了一个 32 位浮点的（float32）、非单位化（non-normalized）的纹理。

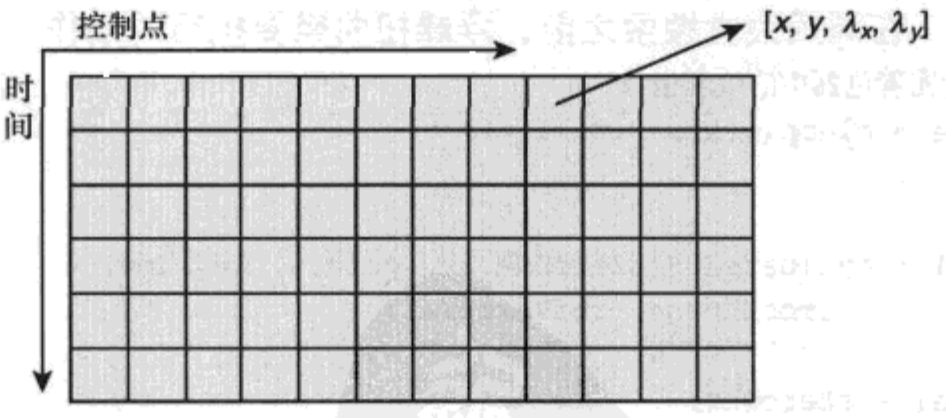


图 5.5.4　动画数据存储在一个纹理中。纹理的每一行代表动画的一帧。沿着一行，第 i 个纹素存储控制点 c_i 的（x，y）坐标和它的 λ_{ix}、λ_{iy} 系数

评估基于 RBF 的扭曲函数的着色器代码如下，它映射当前片段的纹理坐标（在渲染纹理映射的四边形后获得）到原始纹理的纹理坐标。

程序清单 5.5.1　评估基于 RBF 的扭曲函数

```
// 计算 Phi 的函数
```

```
float multiquadric(float r, float h){
    return sqrt(r*r+h*h);
}

// 用预定数量的控制点、实际时间和光滑度来评估 RBF,以得到 texCoord
float2 evaluateRBF(float2 i_texCoord, float points, float keyTime,
        float smoothness, samplerRECT rbfTexture){
    float2 newTexCoord;
    newTexCoord.xy = float2(0.0, 0.0);
    for(int i=0; i<points; i++){
        float2 access = float2(i, keyTime);
        float4 rbf = texRECT(rbfTexture, access);
        float distance = sqrt((pow(rbf.x - i_texCoord.x, 2))
                        +(pow(rbf.y - i_texCoord.y, 2)));
        float temp = multiquadric(distance, smoothness);
        newTexCoord.xy += temp * rbf.zw;
    }

    return newTexCoord;
}
```

5.5.6 动画浮雕贴图

通过在浮雕映射的像素着色器中增加一些额外的代码，就可以生成基于 RBF 的浮雕贴图的动画[Policarpo05]。就在调用线性搜索之前，你应该将原始纹理坐标钳制（clamp）在[0，1]的范围内。如果整个浮雕贴图只覆盖多边形的一部分，这是必需的。在这种情况下，一些片段的纹理坐标将超出的 RBF 评估所需的[0，1]范围。因为在纹理不覆盖的区域之外没有深度或法线信息，所以这样的钳制不影响动画。然后，在调用线性搜索之前，需要把程序清单 5.5.2 的代码加到一个浮雕贴图的着色器中。

程序清单 5.5.2　在调用线性搜索之前，浮雕扭曲需要执行的动作

```
// s 是用在浮雕绘图着色器中的纹理坐标
float2 sZeroOne = clamp(s.xy, 0.0, 1.0);

// 评估 RBFs
float2 sRBFEval = evaluateRBF(sZeroOne.xy, points, keyTime,
                    smoothness, rbfTexture);
// 补偿钳制
s.xy += sRBFEval - sZeroOne;

... // 调用线性搜索
```

5.5.7 动画浮雕替代

浮雕替代使用多层浮雕表示来渲染[Policarpo06]。图 5.5.5（右）显示了一个用 4 层浮雕

纹理建模的替代狗，其深度值显示在左侧。对于浮雕替代的情况，前面描述的扭曲策略将导致所有的层应都被作用同样的扭曲函数，也因此产生同样的运动。因此，虽然一个单一的扭曲函数可用于动画跑动的狗，但是，它不会产生一个有说服力的狗的运动。例如，在这种情况下，两个前腿总是一起，而不是朝着相反的方向移动。

因此，对多层浮雕表示，你可能想单独动画每一层。图 5.5.1 描述了一个步行狗的运动。然而，在这例子中，动画是使用一个单一扭曲函数创建的，并利用双脚及四足动物步行运动的对称性，即在时刻 t 的每个帧 f，前两层使用时间 t 渲染，而最后两层用时间（$1-t$）渲染。因此，虽然右前（后）腿向前移动，但是，左前（后）腿往后移。如清单 5.5.2 所示的代码片段所示，在评估函数 evaluateRBF 中，t 被用作参数 keyTime。

在这种情况下，程序清单 5.5.3 中的线性和二分查找调用接收两个新的参数：sFront 和 sBack。这些参数分别代表了正面和背面层的扭曲纹理坐标。这些坐标用来从不同的层次采样深度和法线贴图。清单 5.5.4 阐述了法线贴图的 x 元素的实例，这里，读回的值被组合成一个 RGBA 变量（normal_x）。法线贴图的 x、y 元素分别存储在不同的纹理，normal_map_x 和 normal_map_y 中。需要时，z 元素可以从另外两个元素计算而来[Policarpo06]。

图 5.5.5　一个用 4 层浮雕纹理建模的狗的替代体。渐进层的深度值分别存储在 R、G、B 和 A 通道（左）。被渲染的狗的一个视图显示在右侧。彩色板 8 是这个图像的彩色版

程序清单 5.5.3　使用单一扭曲函数来产生如图 5.5.1 所示的步行运动

```
float2 sZeroOne = clamp(s.xy, 0.0, 1.0);

sFront = evaluateRBF(sZeroOne, points, keyTime,
                     smoothness, rbfTexture);
sFront = s.xy +(sFront - sZeroOne);

int keyTimeBack =(int)(keyTime+maxKeyTime/2)%(int)maxKeyTime;
sBack = evaluateRBF(sZeroOne.xy, points, keyTimeBack,
                    smoothness, rbfTexture);
sBack = s.xy +(sBack - sZeroOne);

... // 用 sBack 和 sFront 调用线性搜索
```

程序清单 5.5.4　使用纹理坐标 sFront 和 sBack，在两个位置上采样多层法线的 x 元素

```
float4 normal_x;
normal_x.xy=tex2D(normal_map_x, sFront.xy).xy;
normal_x.zw=tex2D(normal_map_x, sBack.xy).zw;
```

对法线的 y 元素，可进行类似的操作。下面的代码片段使用 sFront 和 sBack 去采样彩色纹理。

程序清单 5.5.5 用纹理坐标 sFront 和 sBack 采样彩色纹理，查看视线相对于几层的相对位置

```
// 在相交点读取颜色
float4 c;
float4 cFront = tex2D(texture, sFront.xy);
float4 cBack = tex2D(texture, sBack.xy);

float4 z=abs(s.z-q);    // q 是连接起来的四深度值
float zt=z.x;
c = cFront;            // 击中第一层
if(z.y<zt)c=cFront;     // 击中第二层
if(z.z<zt)c=cBack;      // 击中第三层
if(z.w<zt)c=cBack;      // 击中第四层
```

5.5.8 结果

我们已经用 C++和 Cg 实现了所述算法，使用它们去动画了几个纹理和浮雕替代。在我们的所有实验中，纹理有 400×400 个纹素。在一个有 2.0GB 内存、768MB 的 NVIDIA GeForce 8800 GTX 的 2.21GHz PC 上，当分别用法线贴图、浮雕贴图和浮雕替代的纹理来渲染动画时，我们的执行达到 3 000fps、710fps 和 500fps。

图 5.5.6 描述了定义图 5.5.1 所示的步行狗动画的控制点（小暗点）。用户定义了一组，放于狗图片（左）上面的一系列控制点。然后，其中的一些点被交互地移动，从而定义了图 5.5.6 所示的中间及右边的构型。当用户移动一个控制点时，底层的纹理就被自动扭曲，进而提供即时的视觉反馈，并使用户可以相应地规划和定义动画（见图 5.5.2）。

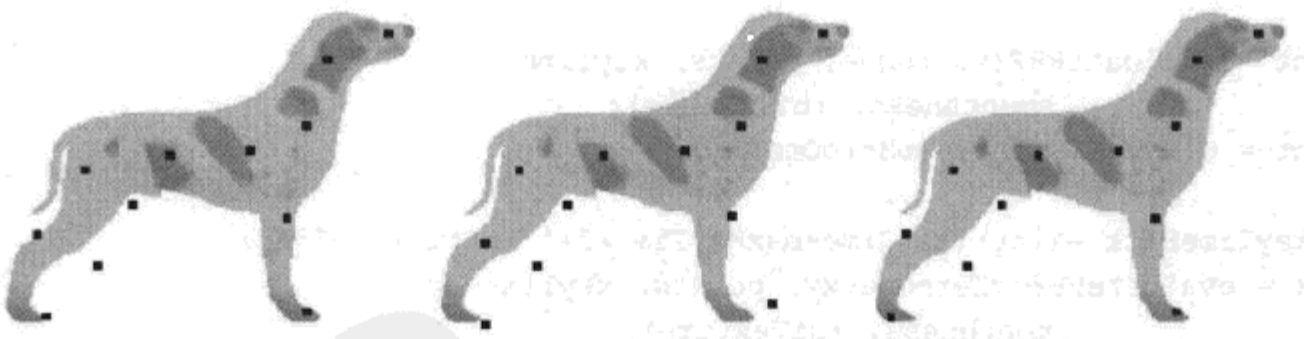


图 5.5.6 定义基于 RBF 的扭曲函数的控制点（暗点），相应地，这些函数创建了图 5.5.1 中步行狗的动画。除了这 12 个控制点外，4 个额外的控制点被放置在纹理的角落来锚固它

图 5.5.7 显示了马动画的几个帧。左边是原始的浮雕替代，右边显示了从同一个视角点观察的，用基于 RBF 的扭曲函数获得的不同姿势。CD-ROM 上的附带视频显示了这些动画。

图 5.5.7　马的动画。左边图像显示了原始浮雕替代的一个视图，右边的 3 个图像显示了从同一视点看到的动画的帧，注意马身体和尾巴的变化。一共用了 27 个控制点来制作动画，包括在纹理角落的 4 个锚固点

5.5.9　结论

本精粹提出了一个用基于 RBF 的扭曲函数来实时地动画浮雕替代的技术。这种方法创建了进行重复运动的、有生命的、移动物体的真实感动画。鉴于其普遍性，它可以用来做基本上任何类型动画的纹理表示。在预处理阶段，用户指定一套控制点，也就是 RBF 表示的中心点。通过在 2D 到处移动这些控制点，用户可获得生成的动画的即时反馈。一旦指定关键变形，系统插值中间帧的控制点，并求解方程 5.5.3 定义的线性系统，来找到一套 RBF 系数（λ_s）。使用控制点的二维坐标，这些系数保存到一个纹理中。然后，在运行时，存储的信息被着色器读取，通过纹理重采样来进行实际的动画。

对浮雕纹理的不同层，我们的技术可用来定义单独的扭曲函数。因此，它支持使用简单接口、复杂动画的定义，从而减少了大量的、通常与纹理动画相关的时间及艺术工作。和任何其他方法一样，这种方法有一些局限性：大变形容易让纹理畸变（distort）太多，导致不好的结果。另外，当从入射余角处看用来渲染替代的多边形时，清单 5.5.2 和清单 5.5.3 中的钳制函数的使用可能会造成一些小缺陷。在这样的视角情形下，以一些性能损失为代价，这些瑕疵可以通过在线性和二进制搜索的每一步调用 evaluateRBF 函数来避免。

所附光盘包含了动画法线贴图和多层浮雕贴图的视频与演示，包括源代码及着色器。

5.5.10　鸣谢

我们要感谢 NVIDIA 捐赠了用于这项工作的 GeForce 8800 GTX 显卡。

5.5.11　参考文献

[Buhmann03]Buhmann, Martin. *Radial Basis Functions*, Cambridge University Press, 2003.

[Carr01]Carr, Jonathan et al. "Reconstruction and Representation of 3D Objects with Radial Basis Functions," Proceedings of SIGGRAPH 2001, ACM Press, New York, NY, pp. 67–76.

[Cook84]Cook, Robert L. "Shade Trees," *In Computer Graphics*(SIGGRAPH 84）18(3), pp. 223–231, 1984.

[Donnelly05]Donnelly, William. "Per-Pixel Displacement Mapping with Distance Functions," *GPU Gems 2*, 2005.

[Hardy71]Hardy, Roland L. "Multiquadric Equations of Topography and Other Irregular Surfaces," *J. Geophys. Res.*, 1971, Vol. 73, pp. 1905–1915.

[IdSoftware]Id Software. DOOM 3.

[Litwinowicz94]Litwinowicz, Peter, and Williams, Lance. "Animating Images with Drawings," Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques SIGGRAPH, 1994, ACM Press, New York, NY, pp. 409–412.

[Noh00]Noh, Jun-yong, et al. "Animated Deformations with Radial Basis Functions," Proceedings of the ACM Symposium on Virtual Reality Software and Technology, Seoul, Korea, October 22–25, 2000, VRST '00. ACM Press, New York, NY, pp. 166–174.

[Oliveira00]Oliveira, Manuel M., Bishop, Gary, and McAllister, David. "Relief Texture Mapping," Proceedings of SIGGRAPH 2000, New Orleans, LA, July 23–28, 2000, pp. 359–368.

[Policarpo05]Policarpo, Fabio, Oliveira, Manuel M., and Comba, João. "Real-Time Relief Mapping on Arbitrary Polygonal Surfaces," ACM SIGGRAPH, 2005, Symposium on Interactive 3D Graphics and Games, Washington, DC, April 3–6, 2005, pp. 155–162.

[Policarpo06]Policarpo, Fabio, and Oliveira, Manuel M. "Relief Mapping of Non-Height-Field Surface Details," ACM SIGGRAPH 2006 Symposium on Interactive 3D Graphics and Games, Redwood City, CA, March 14–17, 2006, pp. 55–62.

[Policarpo06b]Policarpo, Fabio, and Oliveira, Manuel M. "Rendering Surface Details in Games with Relief Mapping Using a Minimally Invasive Approach," In Wolfgang Engel(Ed.), SHADER X4: *Lighting & Rendering.* Charles River Media, Inc., Hingham, Massachusetts, 2006, pp. 109–119.

[Ruprecht95]Ruprecht, Detlef, and Müller, Heinrich. "Image Warping with Scattered Data Interpolation," IEEE Computer Graphics and Applications, Vol. 15, No. 2, 1995, pp. 37–43.

5.6　SM 1.1 和更高版本上的裁剪贴图

Ben Garney
GarageGames

裁剪贴图是一个快速强大的地形纹理技术。本精粹将简要地介绍裁剪贴图的理论，并讨论它们在 SM 2.0（Shader Model）硬件上的实现，最后还将探讨一些高级议题，如在固定功能、SM 1.x 和 SM 3.0+硬件上的支持以及图像数据的不同来源。

5.6.1　裁剪贴图的基本概念

当渲染到大小为 1 024×768、每像素 32 比特（bit-per-pixel，bpp）的显示器上时，提供一个完全详细的、独特的视图只需要 786 432 个颜色信息，这刚好是 3MB 的数据。如果你想每秒渲染 60 帧（60Hz），只需要每秒传递 180MB 的数据到显示器，而这是在大多数游戏平台的能力范围之内的。

假如你想在屏幕上画一个模型，不管它的纹理可能包含多少细节，你能显示的纹素（texel）也不可能比屏幕所具有的像素（pixel）多。对于一个小物件的情况，如角色或加强体（power-up），你可以舍弃纹理较详细的 mip 层次（见[Forsyth07]）。然而，对于在大多数时间内，相机只是对着网格的一小部分的环境，舍弃 mip 层次是不切实际的。如果模型的任何一部分需要高细节，就需要全部或大部分的 mip 层次。在地形或其他环境的某些部分，你将几乎总是需要高细节。

能够做些什么来有效地处理地形纹理呢？只装载 mipmap 的一部分！在理想的情况下，你只需要加载当前帧所需要的纹素到 GPU——这意味着，你可以在只有 3MB 的显存（VRAM）中装载一个任意细节的地形。不幸的是，GPU 生产商还没有制造支持这种运算的硬件。

裁剪贴图是 mipmap 的一个通用化，它允许你只装载每个 mip 层的一个子区间。如果一个被采样的纹素还没有装载，那么就使用已装载的低分辨率数据。这意味着你可以上载一个相对较小的数据集而得到高效率的渲染，并且，如果视点的变化比你的纹理页面调度（texture paging）还快，它可以优雅地降低质量。虽然当前图形硬件还没有直接地支持裁剪贴图，但是你可以利用着色器来有效地模仿它们。

5.6.2 裁剪贴图的实现

基于 mipmapping 的概念，SGI 公司开发了裁剪贴图（见图 5.6.1），目的是虚拟化一个大的纹理[Tanner96]。

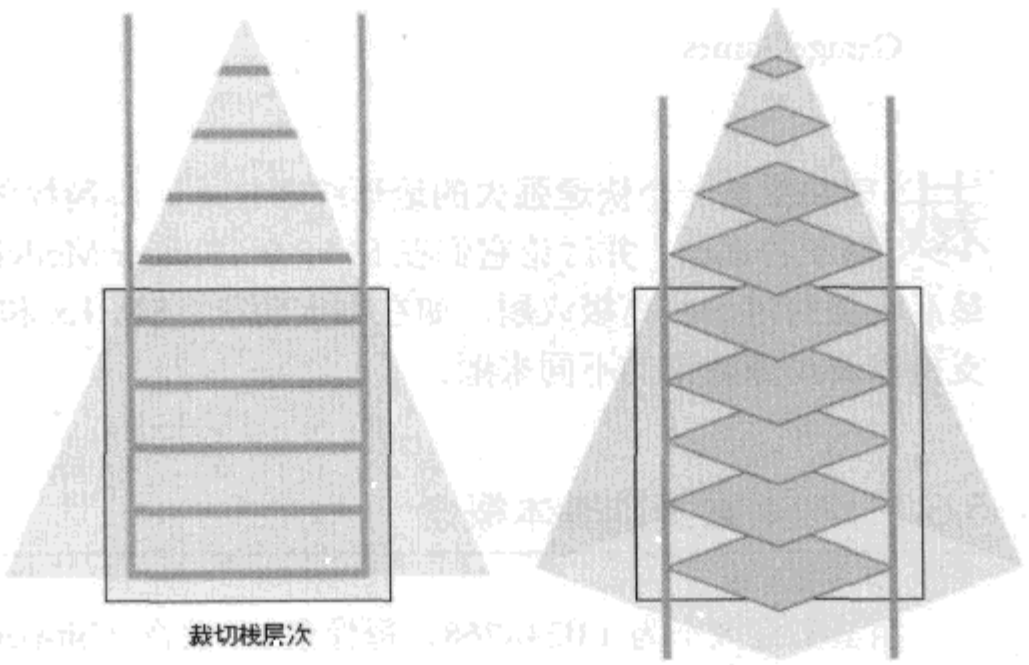


图 5.6.1 被裁切的 mipmap 堆栈的图像

记住，mipmap 是用来减少锯齿和本地化内存的存取。从概念上说，当三角形的一个特定像素被渲染时，像素的边界（bounds）就被投影到纹理空间，根据它的大小选用一个 mipmap，然后从其中的一个点采样。这样做的结果是：当一个三角形变得较遥远时，它就从一个低细节的 mip 层次采样，此时内存的存取要比不这样做时相对集中。

裁剪贴图采取有同样的基本思想，但是，mip 金字塔更详细的层次被裁切，来限制它们的内存使用量。这意味着，如果一个通常要 15 个 mip 层次、消费 500MB 内存的、32KB 的像素纹理被放入一个最大层尺寸为 512 像素的裁剪贴图，它只需要 6 个 512 像素的纹理内存，加上一个 512 像素的“上限”（cap）及一个完整的 mip 链，它的内存占用只有 7.3MB。

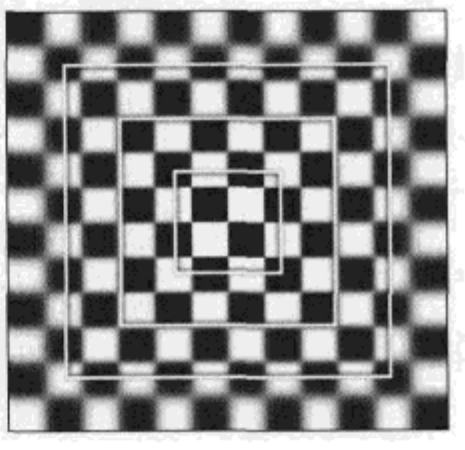


图 5.6.2 每个细节层次包含受正在更新的裁剪贴图影响的那些区域

在 SGI 的 InfiniteReality2 硬件平台上，当存取 mip 层时，硬件检查内存中缓存的裁剪贴图区域是否覆盖它希望读取的区间（见图 5.6.2）。如果是，它就正常采样。如果不是，它就移到下一个不太详细的 mip 层次再试，结果是：对于一个区域，如果不存在详细的数据，那么就使用低细节的数据。

在 CPU 上，通过清除旧数据和从磁盘数据库上载新的数据，SGI 的 Performer 场景图（scene-graph）被用来负责调整裁剪贴图的每一层数据。

裁剪贴图的优点

和讨论过的其他策略相比，裁剪贴图带来以下几个主要好处。

- 在细节层次（LOD）之间，裁剪贴图总是光滑过渡，而不需要用特定的 LOD 来渲染几何。在最坏的情况下，图像也只是有点模糊而已。
- 裁剪贴图有一个固定的内存消耗，在渲染或更新时没有动态的 GPU 资源分配。因为大多数的 GPU 驱动程序不能很好地处理频繁的分配及释放，所以这一点很重要。
- 只要把细节焦点放在相机的位置，裁剪贴图就和视点无关。一个在所有的方向都平滑减少的数据可被利用，这意味着原地打转不会影响性能。
- 在任何有可编程着色器的硬件上，裁剪贴图的实现直接了当，即使在固定功能的硬件上也是有可能仿效裁剪贴图的。实现更新区域的选择有点棘手，但整个系统用起来很直接，没有复杂的缓存逻辑。
- 在图像质量和分配给裁剪贴图的资源之间，裁剪贴图有明确界定的关系，所以很容易基于用户的偏好调整视觉体验。
- 裁剪贴图可以从很多来源接收数据。唯一的文件数据，CPU 或 GPU 合成的数据都可支持。
- 最后，因为更新的节奏是可变的，所以裁剪贴图具有良好的更新特性。所需的最少更新次数通常是相当小和有限的 —— 对一个裁剪贴图层次，只需要上载新的像素，这往往只有几千个纹素。最坏的情况是上载一个全部的裁剪贴图，那也只有十几兆左右。

裁剪贴图的缺点

裁剪贴图的主要缺点是它不能处理变化的细节层次。在纹理空间中，细节只是从焦点线性减少，这使它们不适合处理复杂的室内环境。在室内环境中，纹理空间的多个区域可能需要高细节（例如，地板和墙壁可能使用不同的 UV 区域）。如果你要求中端的 SM2 或更高，有一些好的选择值得看一下，像[Lefebvre04]。

对于裁剪贴图，完全没有优化的着色器也很昂贵，而且，至少需要 SM2。然而，你稍后会读到，只要对几何稍加处理，就可以被大大地优化。在 SM3 或更高版本中，也有可能利用梯度算子来写一个效率更高的裁剪贴图着色器。

裁剪贴图的细节

下面的部分将解释和说明有关裁剪贴图的细节，包括裁剪栈（clipstack）的大小、焦点和更新裁剪贴图的方法。

裁剪栈的大小

在裁剪贴图栈中，纹理大小是使用裁剪贴图时的主要变量，它可以被简单地控制——它应该是最接近显示分辨率 2 的幂函数。为了得到更高质量的结果，就往上偏离；为了得到较低的质量，就往下偏离。这样的原因可以追溯到一个最佳渲染器所需纹素数据的原始的讨论。就纹理而言，要求最高的可能情形是视图直接对着裁剪表面，尽可能靠近但不放大原始纹理。在这种情况下，需要一个和屏幕同等大小的纹理来得到一个全部细节的感觉。

焦点

在用裁剪贴图时，选择焦点是另外一个悬而未决的问题。焦点是最高细节时裁剪贴图的 UV

空间的位置。有许多可能的试探法，但是，能提供最一致结果的就是从相机的位置、朝被裁剪的几何向下投影，并用这些坐标作为新的焦点。

更新方法

有 3 种广泛的途径可把数据变成这里讨论的裁剪贴图的格式。它们的做法大致相同：用裁剪贴图代码来确定更新区域，并提供数据来填补这些区域。

首先，你可以把磁盘上的文件数据飞快地传到更新区域。在这种情况下，一旦数据在系统内存中，你只需要直接将它们上载到 GPU 中。第二，你可以在 CPU 上合成数据再上载。我发现这个没有下一个方法好，但是如果你有一个现成的快速合成程序，它也许是有用的。最后，你可以在 GPU 上通过用渲染到纹理来合成。这要求一开始就把裁剪栈分配成渲染目标，否则操作就和其他方式相同了。

实现裁剪贴图

以下各部分覆盖了有关裁剪贴图的实现细节。

SM 2.0 路径

为简单起见，本精粹只深入讨论 SM 2.0 的裁剪贴图路径。一旦完成了 2.0 路径，大多数的工作就完成了，获得 SM 1.x 和 SM 3.0+路径变得直截了当。

这是光盘上演示程序的、裁剪贴图效果的像素着色器代码。

```
PS_OUTPUT Output;

// 因为基层次的后面什么都没有，它总是可以被采样，从而节约了些数学运算
float3 colAccum = tex2D(clipSamplers[0], In.TextureUV[0]);

// 抓住其余的，根据到每一层中心的距离衰减
for(int i=1; i<CLIP_LAYER_COUNT; i++)
{
    float fade = smoothstep(0.4, 0.5, distance(In.TextureUV[i],
                        g_clipLayerAndCenter[i].xy));
    float4 curColor = tex2D(clipSamplers[i], In.TextureUV[i]);
    colAccum = lerp(curColor, colAccum, fade);
}

// 存储积累的结果并返回
Output.RGBColor = float4(colAccum, 1);
```

在 SM 2.0 路径，你对每个像素进行所有的裁剪贴图层次计算。在每一个像素，你必须确定 UV 坐标，并利用通过统一的着色器常数来传递的信息，通过缩放及偏离原来的 UVs 来对每个裁剪栈条目生成纹理坐标。基于在纹理空间它到焦点的距离，你还可以对每个裁剪栈条目生成一个“衰减”值。然后，通过使用衰减值作为系数，按细节逐渐增加的顺序，你线性插值每个裁剪栈条目的颜色。

环形更新和矩形裁剪器

裁剪方案的一个主要优化是将纹理看成一个环形缓冲，这意味着堆栈的移动被做得尽可能有效（你只要上载新的数据）。然而，确定需要更新的区域有点棘手。

考虑裁剪栈的一个层次。在任何时候，有一个矩形的数据被包含在裁剪栈的纹理中，我们把这个矩形叫做 currentRect。在虚拟纹理的某一个 mip 层次，这是目前已加载的全部数据的一个子集。当移动焦点时，这个矩形到处移动，以保证它还是以焦点为中心。

自上而下，假设你是在裁剪贴图的第三个层次，这意味着裁剪栈纹理的大小只是完整源层次的 1/16。图 5.6.3 说明了这样的情形。网格说明了纹理如何被映射到几何上。你 4 倍缩放单位大小的 UV 坐标，所以纹理在每个方向都被重复 4 次。然而，currentRect 没有和这一网格对齐，它在中间的某个地方。通过按右边所示的模式上载纹理数据，你最终会在几何上得到想要的每一个数据。

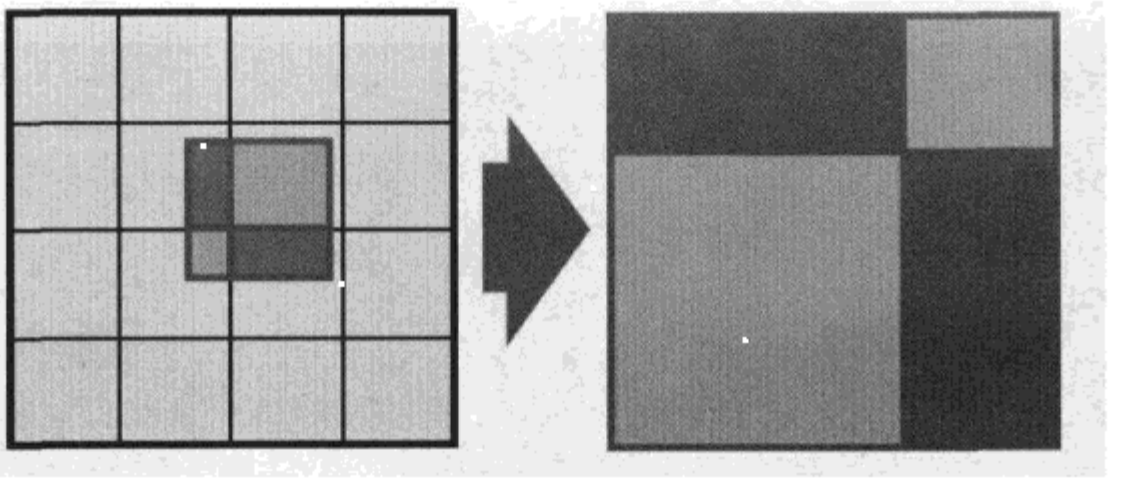


图 5:6.3　裁剪栈纹理的大小只是完整的源层次的 1/16

然后，你根据网格裁剪 currentRect。网格的大小和裁剪栈纹理相等。你总是得到不同的块数（在通常情况下为 4。但是，如果你用不同的方式与网格对齐，它可以更少）。对于所发生的事情和为了被正确地显示，为什么上载的数据必须映射到纹理上，这给了你一个基本思想。ClipMap::fillWith TextureData 实现了这个想法，用来再填充整个裁剪栈。

那更新呢？当移动长方形时，你常会得到看起来像图 5.6.4 的情形。

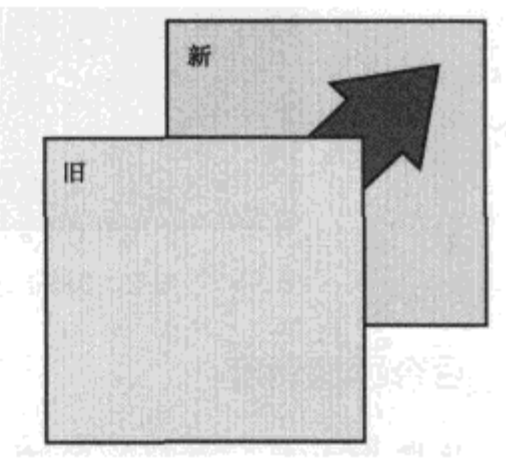


图 5.6.4　在一次更新中，倒 L 形区域是需要上载的

那个倒 L 形区域就是你需要上载的。因此，当你要更新 currentRect 包含的东西时，应先确定这个更新区域，然后，就像以上你对 currentRect 所做的，裁剪和盘绕（wrap）它，从而最终在纹理上得到一个要上载数据的矩形区域。这会导致高效的裁剪贴图更新（焦点移动一个像素意味着只有一个像素宽的矩形要上载）。对于这样的实现，参见 ClipMap::recent。

基本的 CPU 合成

一个简单的棋盘格合成器是一个有益的调试帮助，一个像素的棋盘格使它更容易发现任

何采样或其他问题。应用一个梯度更新使得用户很容易地发现不正确的更新(色彩不会匹配)。

对于这样的例子，见 ClipMap::uploadToTexture。那个#if 块可以切换成 1，来打开一个简单的、为每次裁剪贴图上载选择一个随机颜色的 CPU 合成器。这对于观察更新如何发生和它们覆盖哪些区域非常有用。

基本的 CPU 上载

默认情况下，示例应用程序从系统内存存储的大图像加载数据到裁剪贴图（这可以避免分页调度的复杂性）。这是一个直截了当的位块传送（bitblt）操作。

高级裁剪贴图

下面的部分将涉及一些关于裁剪贴图的高级议题，包括添加后台页面调度、有更好性能的预算更新、优化填充率及其他。如图 5.6.5 所示是光盘上提供的一个裁剪贴图应用中的图像。

图 5.6.5 光盘上提供的一个裁剪贴图应用中的图像。到 9 号彩色板看这幅图像的彩色版本

后台页面调度

从概念上看，这是简单的（尽管实施一个有效的页面调度需要一些工作）。把源图像和它的 mip 层次分成许多块，在系统存储器保留一个裁剪栈重叠的所有块的缓存，加上一个边界数据，以便可快速实现焦点变化后的调整。

确保从你的块到裁剪栈纹理有一个快速拷贝。如果没有可用的更新裁剪栈层次的数据，那么确保数据已经被要求，并推迟更新到稍后的时间。我发现，自下而上的工作给出不错的结果 —— 用户总是有高细节，而中间层次的数据有时需要一段时间才能出现。

预算性更新

和其他方法相比，这是裁剪贴图的一个重大胜利。大多数的表面缓存办法需要做一个固定数量的工作 —— 如合成 128×128 像素的块。当纹素密度增加时，工作量也增加了，直到你看着一个至少要做 512 像素或 1 024 像素的块，这很糟糕。

相反，当焦点变动时，裁剪贴图一般需要频繁的小更新。对一个 512 像素的裁剪贴图，典型的更新只有沿裁剪栈层的横向或纵向边的几层，这可能只是几千个像素的上载。

因此，你可以设置一个纹素上载的预算，在每个级别被更新后，检查是否已经超支。如果是，则停止更新，等待下一帧的更新去处理它。这对于相机移动的情形也有帮助 —— 当细节只有在一帧可见时，你不需要浪费太多的时间。同理，也有可能基于在下一帧前的有效时间来预算。

只要经常更新，每一层至少一次直到停止，所有必需的数据最终将进入裁剪贴图，并且可避免很多只有在一两帧才可见的工作。

优化填充率/低端支持

通过调整你的几何成有已知纹理坐标界限的组块（chunk），在运行时，也可能有效地确定需要用哪些裁剪栈层次来纹理那一组块，从而允许你减少必须绑定到那个裁剪贴图的着色器的纹理数量。

这也开始允许 SM 1.0 的支持，因为你可以只需要 4 个或更少的活跃纹理，这在 SM 1.0 的 4 个纹理采样器的限制之下。到那时，把层次的衰减计算移到顶点着色器（并确保一定的最小顶点密度），你就可以在 SM 1.x 像素着色器上使用裁剪贴图逻辑。

即使在高端卡上，使用与 SM 1.x 兼容的路径也是一个好主意，因为它远远快于固有的（native）SM 2.0 着色器，特别是在那些报告高性能但做起来不快的卡上，像 X300，这可能是一个巨大的胜利。

通过扩展这一想法，这也可以针对快闪卡（Fast Flash，FF）。你可以要么绑定包含组块纹理记录的最小裁剪栈层次，要么使用寄存器组合器（register combiner）和模拟着色器来近似两个或 3 个层次之间的转变。

利用高端的优势

在有像素梯度运算器的着色器模型上，你可以自己做 mipmap 的计算，并查找感兴趣像素的准确裁剪贴图层次。虽然着色器的评估可能很昂贵，但这大大削减了填充率。

在高端的环境中，去考虑对不同的属性维持几个裁剪贴图也是可行的。例如，一个给法线贴图，另一个给漫反射，第三个给镜面光。对于如法线贴图的“本地化”的属性，它们在远处往往平均到零，只维持两个或 3 个最详细的裁剪贴图层次也许是有利的。

5.6.3　如果你想节约些时间……

如果你只是想找个现成的实现，我的雇主，GarageGames 所销售的 Torque Game Engine Advanced（TGEA）包含了 Atlas 地形系统。TGEA 带有完整的源代码和自由的合约条款，而且

Atlas 有完整的、支持 SM1 和更高的页面调度及优化的裁剪贴图实现。对 XNA 平台，TorqueX 的 3D 地形系统还包括一个用 C#写的兼容实施。看一下它们，这可能会为你节省大量的时间和金钱。

L3DT、3d Studio Max 以及 Panda DirectX 导出器被用来创建光盘上演示的美术资产。

5.6.4 参考文献

[Forsyth07]Forsyth, Tom. "TomF's Tech Blog—Knowing Which Mipmap Levels Are Needed."

[Lefebvre04]Lefebvre, Sylvain, Darbon, Jerome, and Neyret, Fabrice. "Unified Texture Management for Arbitrary Meshes," 2004.

[Tanner96]Tanner, Migdal, Jones. "The Clipmap: A Virtual Mipmap."

5.7　一个先进的贴花系统

Joris Mans
joris.mans@10tacle.be

Dmitry Andreev
dmitry.andreev@10tacle.be

最近，大部分游戏都以某种方式使用贴花。例如，在环境中显示子弹痕迹，或在重复的几何形状中加些变化。通常，这是通过在现有的几何上渲染一个透明的多边形来实现的。但是，这种技术有一些缺点，特别是如果你想在贴花上使用凹凸贴图时。当在现有的凹凸映射的几何上渲染凹凸映射的贴花时，光照是不正确的，因为贴花下方的像素应该用贴花凹凸贴图和几何凹凸贴图的组合来光照。本精粹将阐述如何渲染贴花，取代几何的凹凸和漫反射映射（这可以扩展到你所使用的任何类型的纹理映射），从而产生正确的光照效果和更高的图像质量。

5.7.1　要求

本精粹的实现可以在任何支持渲染对象和着色器逻辑的平台上做到，也就是能够从至少两个不同的贴图（如果你只想用它来作为漫反射纹理）或 4 个（如果你想增加凹凸贴图支持）获得的值来采样及插值。最佳的图像质量可以用与屏幕的分辨率相同的渲染目标来得到，但较小的渲染目标也可以，只是图像质量会下降。提供的演示可在任何支持像素着色器 2.0 版本的、有与 DirectX 9 兼容的显卡的 PC 上运行。

5.7.2　正常的贴花方法

在使用贴花系统的传统引擎中，你首先在帧缓冲区渲染所有的几何，然后，通常使用某种形式的融合，你在那上面渲染包含贴花的多边形。

5.7.3　先进的贴花方法

在这个例子中，你将略有不同地做事情。首先，你需要在运行时创建必要的工具来实现贴花（decal）渲染。在这种情况下，我们将建立两个全屏渲染目标，第一个是被称为 DiffuseRenderTarget 的 32 位 RGBA 格式；第二个

也将是 32 位 RGBA 格式，它被称为 BumpRenderTarget。对于这第二个，如果你的凹凸贴图有较高的精度存储，可以使用一个每分量为 16 位的缓冲区或任何其他形式的渲染目标。在演示中，我们使用了 DXT5 压缩凹凸贴图，所以，32 位 RGBA 就够了。

渲染场景可以分成两个部分：首先生成贴花缓冲，然后渲染作用了贴花的场景。为了生成贴花缓冲，执行下列步骤。

- 在主要的 z 缓冲中，渲染几何的所有深度值（不包括贴花）。所用深度比较函数和你用来渲染正常场景的一样。
- 选择 DiffuseRenderTarget 作为当前渲染目标，同时仍然使用主要 z 缓冲。用黑色作为清除颜色来清除渲染目标。
- 渲染所有的贴花几何到那个渲染目标，使用和前面的深度通道同样的深度比较函数，但不要用在正常贴花情况下用的全部着色器来渲染。渲染使用一种特殊的着色器，它只输出漫反射纹理的颜色，与不透明纹理（或漫反射 Alpha，取决于艺术管道）前乘。在渲染目标的 Alpha 分量中，输出用来缩放漫反射值的不透明度值。
- 在 BumpRenderTarget，你做些类似的事情。这次用在世界空间的凹凸映射纹理作为输出，渲染贴花几何。如果你的目标硬件支持多种渲染目标的渲染，这一步可与前面几步合并，采样结果如图 5.7.1 和图 5.7.2 所示。

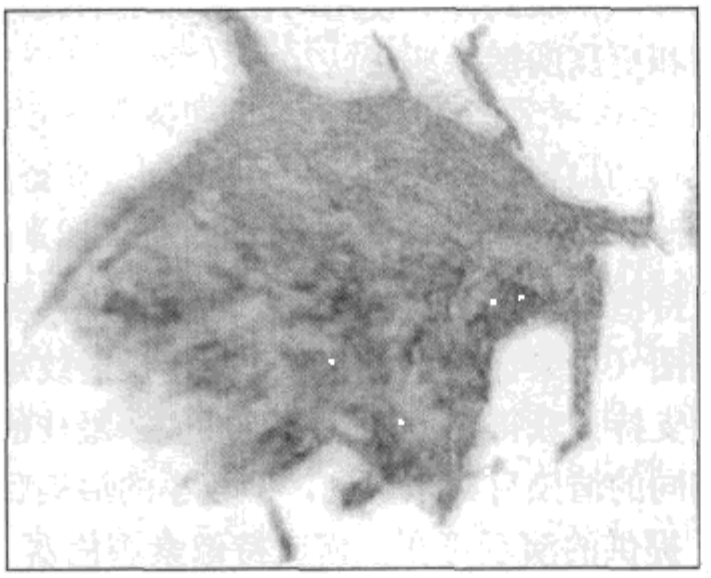


图 5.7.1 贴花系统所使用的 DiffuseRenderTarget

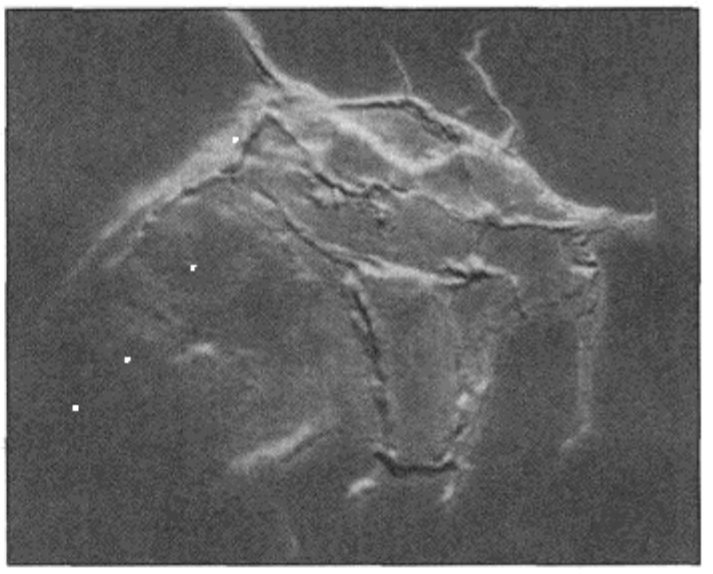


图 5.7.2 贴花系统所使用的 BumpRenderTarget

最后，渲染场景。在用于几何的着色器中，你必须做的事情是改变进行漫反射纹理查找和凹凸纹理查找的代码。事实上，你必须组合作用于几何纹理的漫反射值和在 DiffuseRenderTarget / BumpRenderTarget 纹理找到的值，过程如下。

- 得到你正在渲染的像素的屏幕空间位置，它将被用作纹理坐标来读取渲染目标中的值。
- 从贴花漫反射贴图中，用纹理坐标来读漫反射 RGB 值，称此值为 *drt*。
- 融合从渲染目标中读取的 Alpha 值和从用于几何的漫反射纹理中读取的 RGB 值。这给你以下公式，其中，*dt* 是物体的漫反射纹理，*drt* 是包含贴花漫反射值的渲染目标纹理，*d* 是生成的漫反射值。

$$d_{rgb} = dt_{rgb} \times (1 - drt_a) + drt_{rgb} \tag{5.7.1}$$

- 读取贴花凹凸贴图中所存的值，并将其存入变量 *brt*。
- 按照下列公式，把它和物体的凹凸贴图结合。这里，*bt* 是物体的凹凸纹理，*wsb* 是在世界空间的凹凸向量，*drt* 是包含贴花漫反射值的渲染对象纹理，*b* 是生成的凹凸值。

$$wsb = TransformToWorldSpace\left(DecodeBump(bt_{rgba})\right)$$
$$b_{xyz} = wsb \times (1 - drt_a) + DecodeBump\left(brt_{rgba}\right) \times drt_a$$
$$b = normalize(b) \qquad (5.7.2)$$

取决于你存储凹凸贴图的方式，DecodeBump 是把 RGBA 纹素转换成一个凹凸向量的函数。当然，在数学上，像这样插值凹凸向量并不真正的正确，但是，在这种情况下的视觉效果不错，所以你不必寻找更先进的解决方案。

图 5.7.3 和图 5.7.4 显示了传统方法和本精粹中阐述的方法的一个对比。

图 5.7.3　使用传统技术的贴花

图 5.7.4　使用这个技术的贴花

DecodeBump

正如前一段提到的一样，本示例使用了一个叫做 DecodeBump 的函数来解码凹凸贴图。有几种可以存储凹凸贴图的方法，具体选用其中哪个取决于硬件支持、质量和速度。详细地解释不同的方法超出了本精粹的范围，但它确实包括了一些说明如何做到这一点的实例。最简单的解决方案是使用一个每个分量为 8 位的 RGB 渲染目标，用颜色值存储凹凸贴图、缩放及偏移，以适应在 0～255 范围内的像素颜色。

编码一个凹凸向量成这个格式将看起来像：

$$color.rgb = (bumpvector.xyz + 1) \times 127.5 \qquad (5.7.3)$$

相应的 DecodeBump 函数将类似于：

$$bumpvector.xyz = color.rgb \times 2 - 1 \qquad (5.7.4)$$

想一想，在编码时，虽然你写的字节值范围为 0～255，但在像素着色器，所有的值都是单位化的浮点数，其中 0 映射到 0、255 映射到 1.0。这种存储方式的缺点是凹凸贴图是非压缩的，而使用 DXT1 压缩会产生相当不良的视觉效果。

另一种有时被使用的办法是在 DXT5 压缩表面存储凹凸值，利用绿色分量存储凹凸向量的 *x* 值，使用 Alpha 分量来存储 *y* 值。当读取凹凸贴图时，你就可以用 *x* 和 *y* 来重建 *z* 值。

凹凸向量的编码为：

$$\begin{aligned} color.r &= 0 \\ color.g &= (bumpvector.x + 1) \times 127.5 \\ color.b &= 0 \\ color.a &= (bumpvector.y + 1) \times 127.5 \end{aligned} \tag{5.7.5}$$

DecodeBump 函数重建第三个分量：

$$\begin{aligned} bumpvector.x &= color.b \times 2 + 1 \\ bumpvector.y &= color.a \times 2 + 1 \\ bumpvector.z &= \sqrt{(bumpvector.x)^2 + (bumpvector.y)^2} \end{aligned} \tag{5.7.6}$$

利用 DXT5 压缩纹理的绿色分量包含 6 字节精度，而 Alpha 分量被分开压缩的事实，这允许你用压缩格式存储凹凸贴图，同时仍然保持一个良好的质量水准。当然，权衡是有重新生成凹凸向量的更多计算 —— 这样的计算不是所有平台都具备或可能过于昂贵。存储需求和像素着色速度之间有一个折衷，不过，在有一些平台上，压缩的数据让你有更少的缓存命中失败，实际上是比直接读解压值更快，即使要计算 *z* 分量。

5.7.4 这个先进贴花系统的优势

和通常的凹凸贴图相比，这个系统的主要优点是提高了图像的质量，它还允许贴花以不同的方式来被使用。例如，想象使用只包含凹凸贴图的贴花来影响重复墙体的外观，如通过增加裂缝、噪声或其他的变化。不仅仅可以在运行时使用贴花来添加子弹和爆炸痕迹，在创建场景时，你也可以在关卡编辑器中使用它们。

在支持标准贴花的引擎中，创造同样的多样性需要一个完全不同的做法。因为你不能用正常的贴花取代凹凸贴图，所以，对于场景中需要变化的每一部分，你将不得不实际创建凹凸贴图，用变化的凹凸贴图取代原来的。这不仅意味着在内存中，你将有更多的凹凸贴图，而且它还需要艺术家做更多的工作来创建，及在几何上放置这些凹凸贴图。

正如图 5.7.5 所示，改变贴花的不透明度可以模拟随时间变化的几何的磨损和撕扯。由于在这里有连贯的光照，所以在侵蚀凹凸贴图中，你可以通过改变贴花的不透明度来完美地融合。在使用大量实例的场景中，它也可以用来创建不同的变化。你可以到处实例化同一几何，如果可能，也可以利用硬件实例支持。但由于贴花系统，你可以在每个实例的表面上添加变化。在内存中，不是每个实例都需要有单独的内存占用。因为在贴花系统中使用了深度缓冲，所以它支持非平面贴花。唯一的限制是，贴花必须尽可能密切地贴着它们底层的几何。除此之外，贴花的拓扑没有任何限制。一个贴花在非平面几何上的例子如图 5.7.6 所示。

它也很容易实现，将其集成到现有的技术中通常并不需要在渲染管线中做重大变化。如果 *z* 的预通道(pre-pass)已经可用，可以在预通道之后添加渲染到两个贴花渲染目标中，并且，为了从贴花缓冲中读取，在渲染场景时，你只需要改变漫反射和凹凸纹理采样的着色器代码（对此精粹所示的许多图像的彩色版本，参见彩色插图区）。

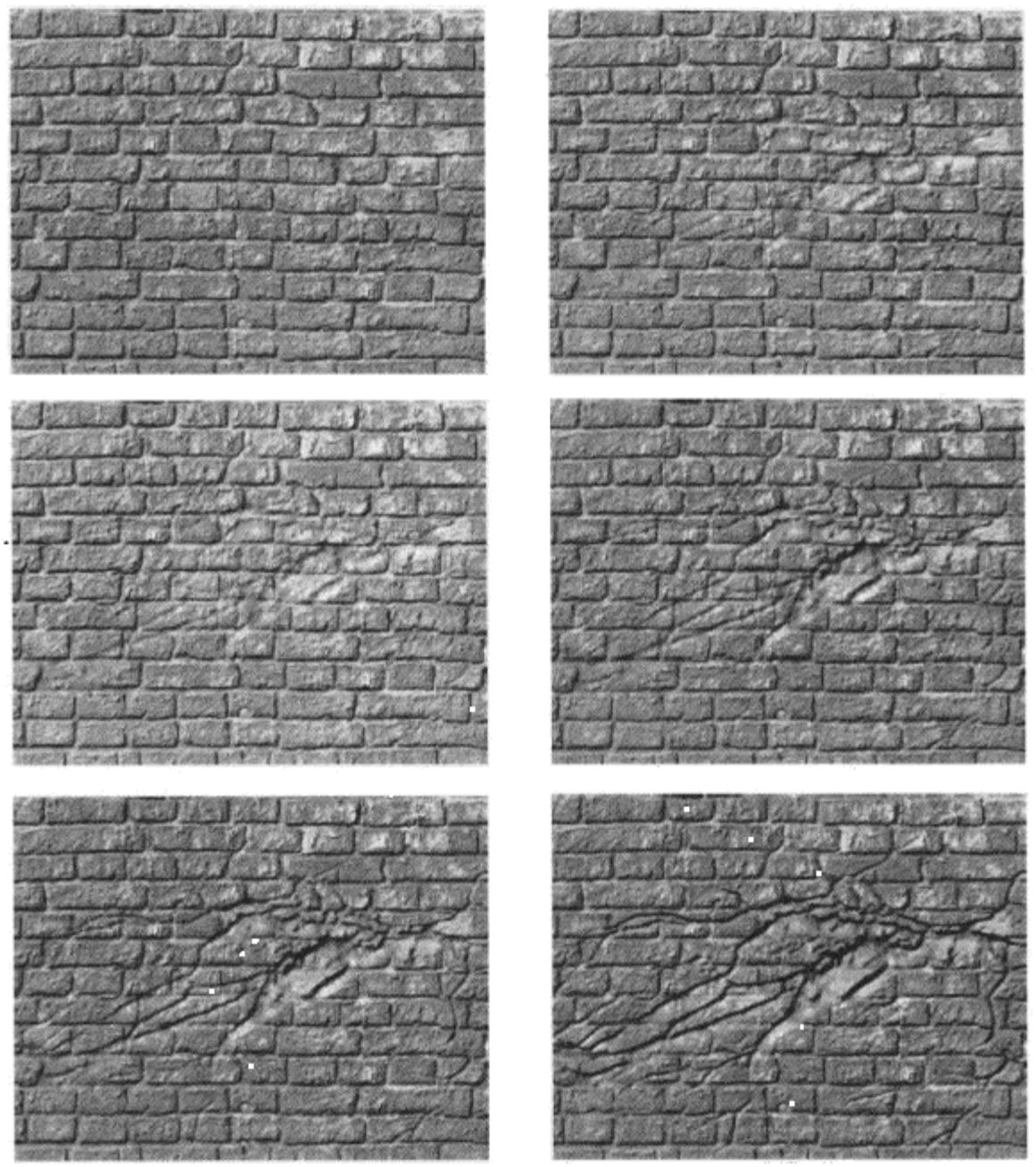


图 5.7.5　贴花用做随着时间变化的侵蚀。从左至右、自上而下，不透明度的值分别为 0%、20%、40%、60%、80%和 100%

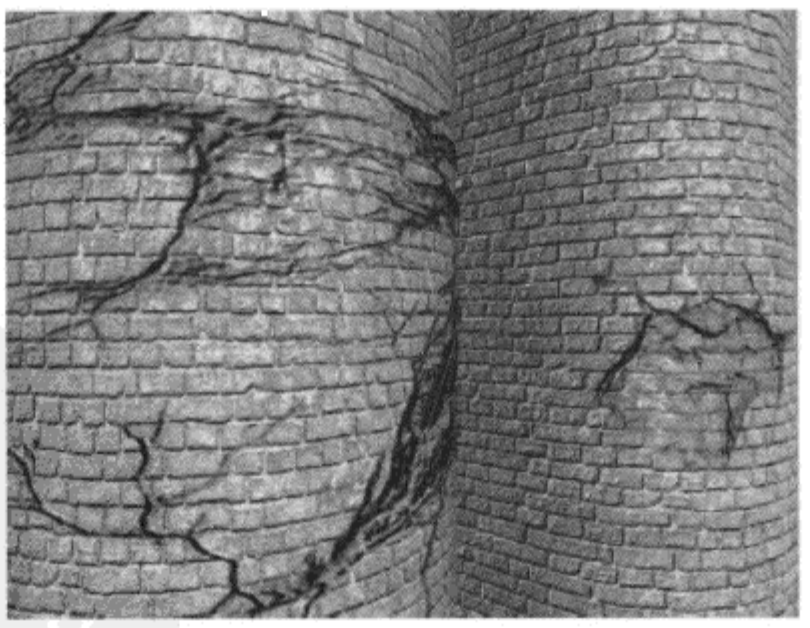


图 5.7.6　非平面几何上的贴花

5.7.5 性能和实验结果

本小节展示了我们对这里所描述技术的实现结果、一些性能试验和潜在的问题。本书光盘上的演示只是先进贴花技术在我们的游戏引擎中的一个简单化实现，它显示了技术的主要部分，给出了清楚的性能趋势。所有的测试都是在一个有 GeForce 6800 和 GeForce 7800 GT 的 3.0GHz 的 P4（奔腾 4）上进行的。

我们用演示的 4 个渲染预设来显示性能的不同方面。

- 原始的 —— 一个在几乎所有的现代引擎中都已经存在的标准光照模式，没有使用贴花。在这种情况下，它是基于两个每像素计算的光源，利用切线空间的法线贴图、漫反射贴图和镜面贴图。
- 正常的贴花 —— 使用原始的着色器，渲染在几何上的正常贴花。一个贴花包括一个漫反射纹理和凹凸纹理。
- 先进的（原始的） —— 渲染贴花到贴花缓冲，但使用原始的着色器来渲染场景中的物体，所以没有贴花显示。这可让你看到填补两个贴花缓冲的开销。
- 先进的 —— 渲染贴花到贴花缓冲，并且将它们应用于场景中的物体。

首要的问题是，正常贴花与先进贴花的性能区别是什么？图 5.7.7 和图 5.7.8 显示了每帧每秒的性能。有两个 512×512×32（漫反射和镜面）和一个 1 024×1 024×32（法线贴图）的纹理分配给每个物体，也有相同大小、相同数量的纹理分配给贴花。

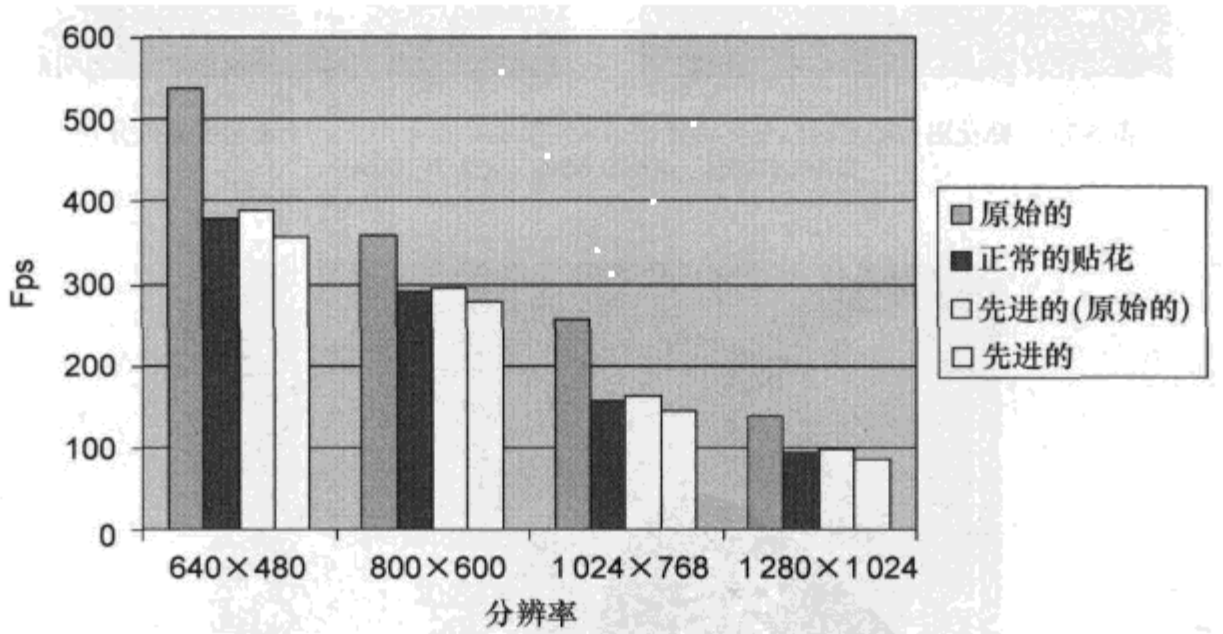


图 5.7.7 全场景测试

这两项测试中，我们使用了非压缩的纹理，法线贴图的 8×各向异性过滤。在图 5.7.7 中的“全景测试”，摄像机被这样放置，以至于覆盖远近场景物体的几乎所有贴花都可见。然而，如图 5.7.8 所示的近拍测试是在相机只指向一个覆盖全屏的单个物体时完成的，以至于所有其他的都被 *z* 剔除掉（cull）。

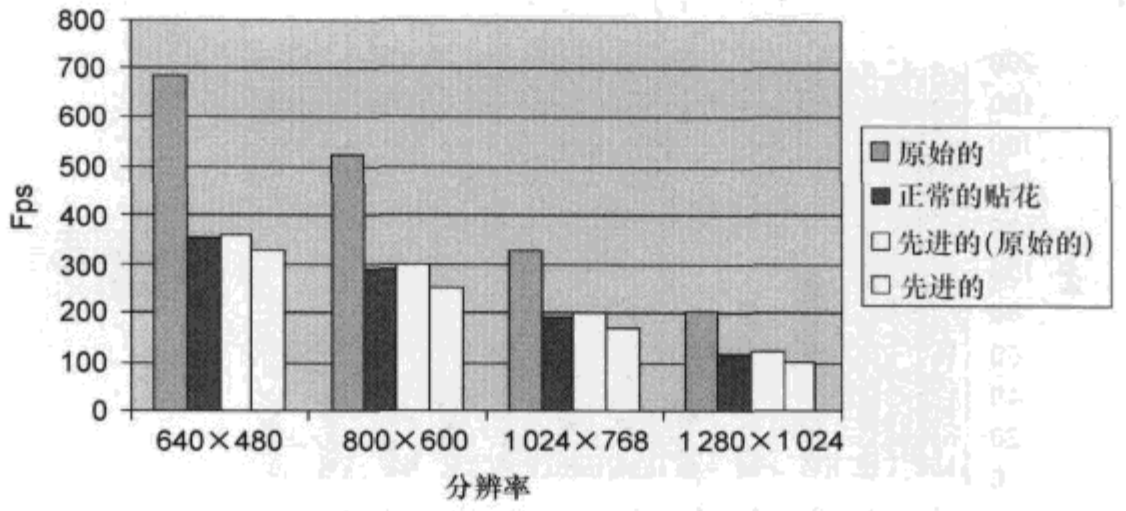


图 5.7.8　近拍测试

不管我们在什么样的分辨率下渲染，是渲染整个场景还是近拍，“先进”技术比“先进（原始）”技术大概慢 11%。同样的试验也用压缩纹理做过，它们有更高的帧速率。并且，它们在“先进”技术比“先进（原始）”间给出了 23%的差异。因此，那些少数额外的，在主要着色器上的纹理获取和混合指令花费了我们 11%～23%的速度。

但是，那些区别如何取决于贴花的复杂程度呢？为了回答这个问题，我们做了两个表明依赖性的试验。在场景中，我们多次渲染一个全屏幕贴花，看看它如何影响性能。因为在贴花缓冲内的查询开销独立于使用的贴花数目，所以先前做的测试已经表明了这些查询开销的潜在性能影响。第二个试验表明了实际的渲染贴花它们自己的成本。

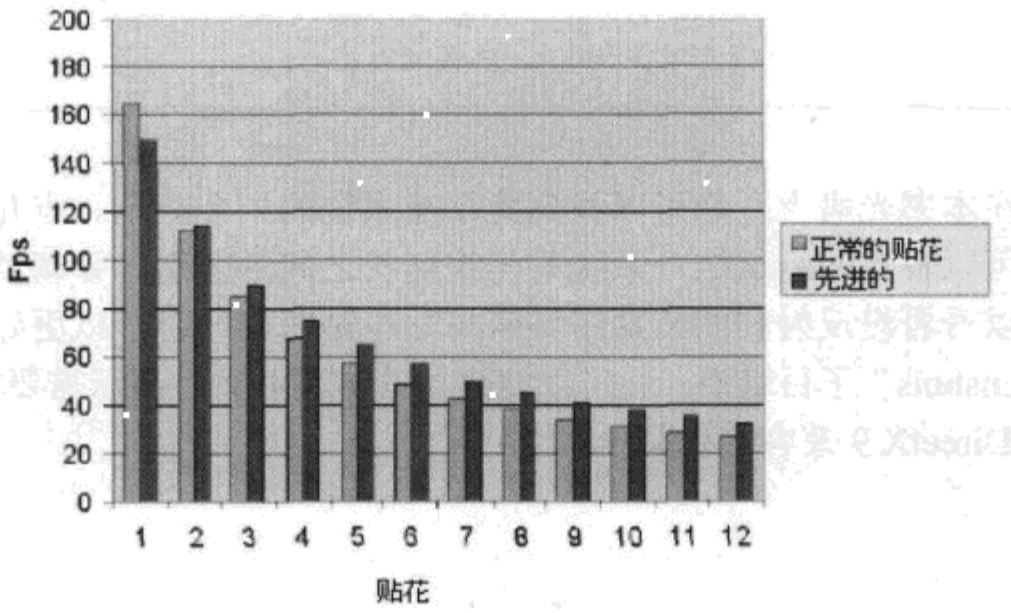


图 5.7.9　非压缩纹理

正如图 5.7.9 和图 5.7.10 所示，当只用一个贴花时，标准贴花技术较快，但在渲染多个互相重叠的贴花时，先进的技术实际上渲染更快。这是因为当渲染多个互相重叠的标准贴花时，进行光照和凹凸贴图的复杂着色器对每个正在渲染的贴花都执行一次，而对先进的贴花，即使多个贴花重叠，复杂着色器也只运行一次。

如果你的渲染管线受限于内存或 API 调用，那么，所有贴花缓冲（纹理）可以一次使用多个渲染目标来填充。在此演示中，它只能说明它不会造成任何额外的性能问题。但是，使用它会减少纹理状态的变化和 API 调用，只是因为在这种情况下你只需要渲染贴花一次。

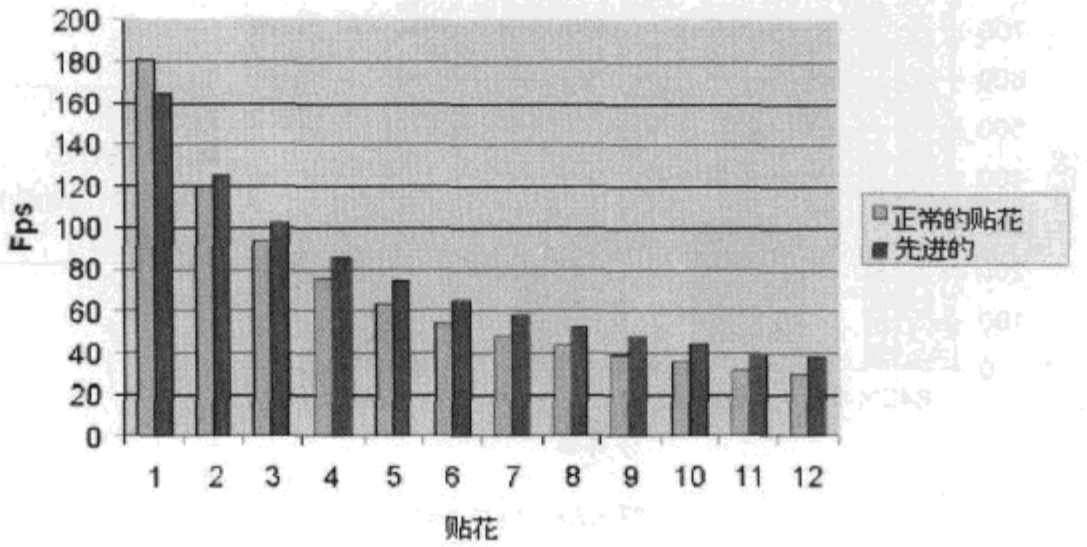


图 5.7.10 DXT1/5 压缩的纹理

在我们的测试场景中，虽然我们可以得出一个有约 12%的性能开销的结论，但是，我们不应该忘记，这些都是测试案例，一个渲染引擎要进行渲染贴花物体之外的更多渲染。当考虑到渲染时其他事情的开销（例如，全屏效果、粒子系统和阴影映射等），总的性能打击百分比将会小些。通过使用这些贴花，其他要考虑的事情是：你可以用少量的不同纹理建立一个场景，从而促使可用内存的增加，较少的状态变化和更大的批量。这实际上可能提高渲染性能。因此，使用这些贴花的成本将最终小于 12%，同时获得可用内存。当有很多重叠的贴花时，你甚至可以看到性能上的优势。

5.7.6 演示

在本书光盘上，你可以找到此贴花系统的一个演示。有几个可调参数，这样你就可以看到和测试这一系统与传统贴花之间的区别，并调整某些渲染设置。你也可以看看漫反射和凹凸贴图如何与贴花贴图相结合，以更好地了解该算法。在“screenshots”子目录中，还可以找到一些屏幕截图。演示需要支持着色器模型 2.0 和与 DirectX 9 兼容的视频卡。

5.7.7 结论

本精粹覆盖了一种渲染贴花的方法。与传统的方法相比，它有更多的优势，它会产生更好的图像质量和被贴花覆盖部分的一致光照，而且它允许使用以前不可能的贴花。例如，可使用只包含凹凸贴图而没有漫反射纹理的贴花。这里提出的方法可以很容易地在现有的技术中集成，而不需要生产或渲染管线的巨大变化，且性能成本相对较小。此外，把此技术略加推广，就可以使用贴花来取代用于场景几何的任何纹理。另一种可能性是：在编辑器中创建几何时，在场景中增加贴花，从而在通用的贴片纹理上增加变化，而无需使用有更多细节的纹理。

5.7.8　参考文献

[Jing06] Jing, YingHui, et. al. "A Post-Processing Decal Texture Mapping Algorithm on Graphics Hardware," Proceedings of the 2006 ACM International Conference on Virtual Reality Continuum and Its Applications, pp. 99–104.

[Lengyel01] Lengyel, Eric. "Applying Decals to Arbitrary Surfaces," *Game Programming Gems 2*, 2001, Charles River Media, pp. 411–415.

5.8 室外地形渲染的大纹理映射

Antonio Seoane, Javier Taibo, Luis Hernández, and Alberto Jaspe
VideaLAB, University of La Coruña

(antonio.seoane@videalab.udc.es), (jtaibo@udc.es),
(lhernandez@udc.es), and (jaspe@videalab.udc.es)

在许多游戏（尤其是飞行模拟器）中，给非常详细的大型地形区域添加纹理是必需的。幸运的是，硬件支持高达 8 192 平方纹素的大纹理。传统技术是基于大纹理的贴片或细节纹理的融合。这些技术的问题是，在一种情况下，几何必须分成段，以至于几何的边界刚好匹配纹理贴片的边界；在其他情况下，外观是重复的、不自然的、不真实的。本精粹将解释一个基于裁剪贴图的、允许使用大纹理的方法。这个技术可以被任何几何算法所使用，而不需要分割纹理成与几何边界适应的贴片。此外，它允许动态的几何变形。

5.8.1 简介

对于网络游戏，庞大的纹理可以存储在游戏服务器上，这样它们就可以被实时下载。这将允许超过用户计算机存储容量的材质，以及允许轻松地更新游戏服务器来添加更多的细节、新功能等。此外，容许巨大的纹理是一件很自然的事情，它也使艺术家的工作更轻松。他们可以使用一个大的画布，用尽可能详细的信息来画，并消除由于重复贴片纹理造成的瑕疵。

为了成功地处理大于系统内存和显存的纹理，需要一些特定的技术。Virtual Terrain Project 的网站上有大量的关于在地形上映射大纹理的论文汇编[VTerrain07]。绝大多数现有解决方案的一个缺点是：纹理和几何数据库之间的强耦合性。这就需要细分纹理，使其适应于几何形状，反之亦然。

裁剪贴图是其中一个最好的用来管理和系统内存不适配的大型纹理的办法[Tanner98]。这一技术解耦了纹理与几何的处理，从而允许这两个数据库之间的独立性。在 Silicon Graphics 系统中第一次实现了这一技术，它使用了昂贵的、特定的硬件[Montrym97]。

裁剪贴图的主要想法是：通过在显存只保留金字塔的一个子集来处理大型 mipmap 金字塔（其中，大型意味着比纹理尺寸的限制和/或可用的显存大）。每个层次被驻留的部分被一个用户指定的叫做裁剪尺寸的参数所限制。

尺寸小于或等于裁剪尺寸的层次总是留在显存，较大的层次根据此限制来裁剪。不完整层次的驻留部分集中在一个叫做细节中心或裁剪中心的点周围。当相机移动时，裁剪中心就被动态地更新，每层在显存缓存的区域也因此被更新。这样，总是有最可能好的质量来将几何映射到正在被可视化的区域，它们可以是低分辨率的较大区域，或者是有微小细节的小区域。

裁剪贴图的主要优势是：一个巨大的纹理可以用内存的一个有限部分来处理。例如，一个裁剪尺寸为 1 024 的 65 536×65 536 纹素的裁剪纹理只需要 29.34MB 大小的显存。如果使用 32 位深的存储设备，它就要 21.3GB 的存储空间。根据分配给它的显存数量，该系统可以被调整来使用任何裁剪尺寸。

随后，本精粹将介绍一个允许使用当前的 PC 和游戏机硬件来处理众多纹理的技术。这项技术用贴片来存储图像，这些贴片不被直接当作纹理使用。类似于裁剪贴图的想法，这些贴片在缓存兴趣区域的纹理栈组合。虽然它是受裁剪贴图启发，但是其结构、显存管理和纹理的应用方式有着重大的差异。这个技术可以在任何图形卡上实现，而不要求特殊的硬件 —— 执行这一技术只要求 OpenGL 或 Direct3D 的固定功能管线。

本精粹描述的技术已成功地应用于好几个项目，如在约 60 000km^2 的地区使用 0.25 米/纹素的纹理细节[Santi07]。它也被成功地用于不同的基于网格的和不规则三角网的（TINs）的几何算法。

这一技术的主要优点如下。

- 它可以使用固定功能管线的 API 实现，如 OpenGL 或 Direct3D。
- 它保持几何与纹理数据库的独立性。
- 纹理坐标可以在 GPU 上自动计算，无需向图形系统的传送过程。这允许几何的实时修改，同时保持正确的纹理映射。
- 纹理锯齿可使用三线性和各向异性过滤的硬件能力来避免。
- 它允许高分辨率纹理的可视化，且有可能包括更高分辨率的区域。
- 它允许使用多个独立的大型纹理，这些纹理可以合并，用来在地形上同时显示不同的信息类型。

5.8.2　结构

提出的技术可管理几乎无限的、被称为虚拟纹理的纹理。它是使用金字塔形的 mipmap 办法来储存的。此金字塔的最高细节层次最多由 $2^{l-1}\times 2^{l-1}$ 个纹素组成（如正方形纹理），其中，l 是金字塔中层次的数量。如图 5.8.1 所示，各层从 0～2^i 编号，层 i 有最大的边长。

虚拟纹理被完整地存储在永久性存储器上，比如本地磁盘或者是通过远程访问的服务器。这些虚拟纹理是以边长为纹素二次方的正方形贴片的形式存储在永久性存储器上的。其中的一个例外是金字塔中那些纹理尺寸小于贴片尺寸的层次。贴片用向量来寻址（列、行和层次）。

虚拟纹理的预过滤 mipmap 层以所需存储空间的三分之一增加，但这是用一个有效的方式来映射纹理而防止锯齿瑕疵所需要的。

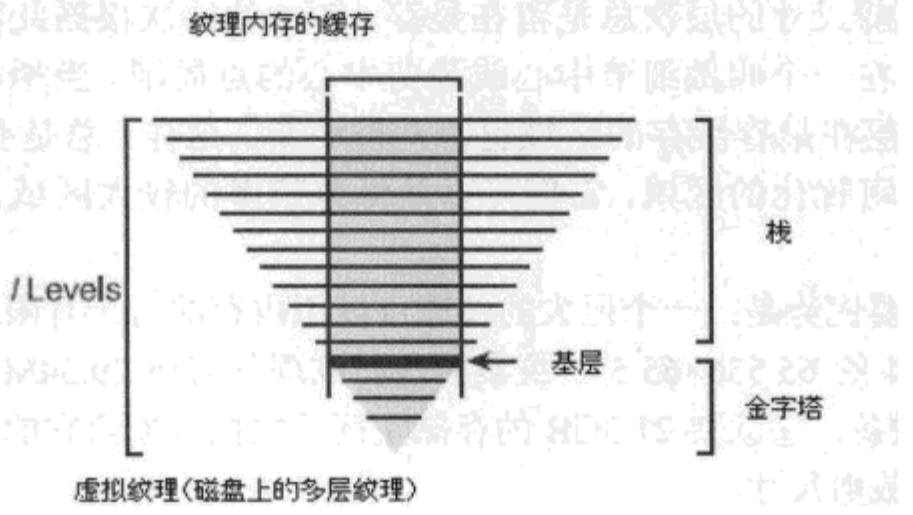


图 5.8.1 虚拟纹理

纹理缓存

根据裁剪贴图的概念，一个完整金字塔的子集被缓存在纹理内存，用来对可视化区域应用适当的细节层次。这个虚拟纹理通过一个二级缓存系统来管理。二级缓存位于主存，并使用一批缓冲器来存储最近最少使用的贴片。

贴片按需求异步加载。请求按层次排优先级，粗层次获得较高的优先级。通过这种方式，大的面积会被尽快地覆盖，当更高层次的贴片到位时，兴趣中心周围的细节就被逐步完善。贴片尺寸是一个关键的参数，因为它会影响从持久存储器到主存的传送率。

第一级缓存驻留在纹理内存，它是虚拟纹理层次的一个子集。从最顶点到基层次，虚拟金字塔完全被存储在纹理内存。这套层次被称为金字塔，它将会作为一个一般的 mipmap 化的纹理来管理。基层的尺寸叫做裁剪尺寸。基层(l_b)用 $l_b = \log2(c)$从裁剪尺寸(c)算出。

从基层次往上，只有完整层次的一个子区域才被存储。不完整的层次被称为堆栈。堆栈层以纹素为单位，大小都一样，在一半的地形区域上，这些层次是逐渐由粗到精的细节层次。组成栈的层次是对应的虚拟纹理层次的不完全子集（$c \times c$ 纹素大小）。这些层次以细节中心的兴趣点为中心。图 5.8.1 显示了这些概念。

在纹理内存，如图 5.8.2 所示，你将使用（$l - l_b + 1$）个独立纹理。第一个纹理（t_0）对应于金字塔最精细的层次，随后的纹理 t_i 缓存对应的虚拟层次 l_b+i。

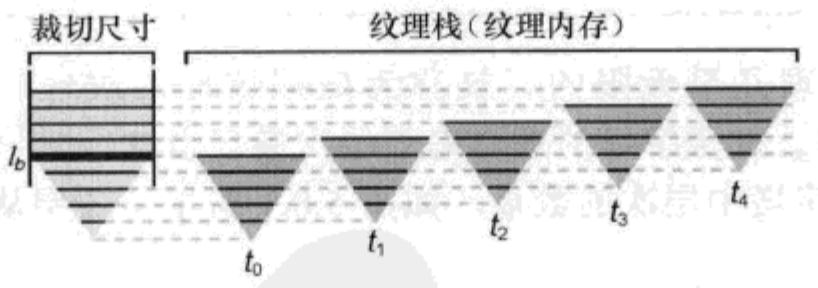


图 5.8.2 纹理堆栈

三线性过滤

为了允许图形系统使用三线性过滤而避免锯齿，每个纹理的 mipmap 层次是必需的。设 t_{ij} 是纹理 i 的第 j 个 mipmap 层次，它缓存虚拟纹理的第 $l_b + i - j$ 个层次。

正如图 5.8.3 所示，没有必要去拥有对应于堆栈纹理的所有 mipmap 层次。在更新缓存时，这可以节省宝贵的带宽。我们的经验证明，在堆栈纹理中，使用 1 024×1 024 纹素的裁剪尺寸，大约四五个 mipmap 层次足以取得良好的质量，而没有明显的瑕疵。

图 5.8.4 展示了一个被堆栈的不同层次所覆盖的地形区域。你可以看到，图中的地形使用的层次问题由颜色纹理表示（这个图像的彩色版本参见彩图 11）。

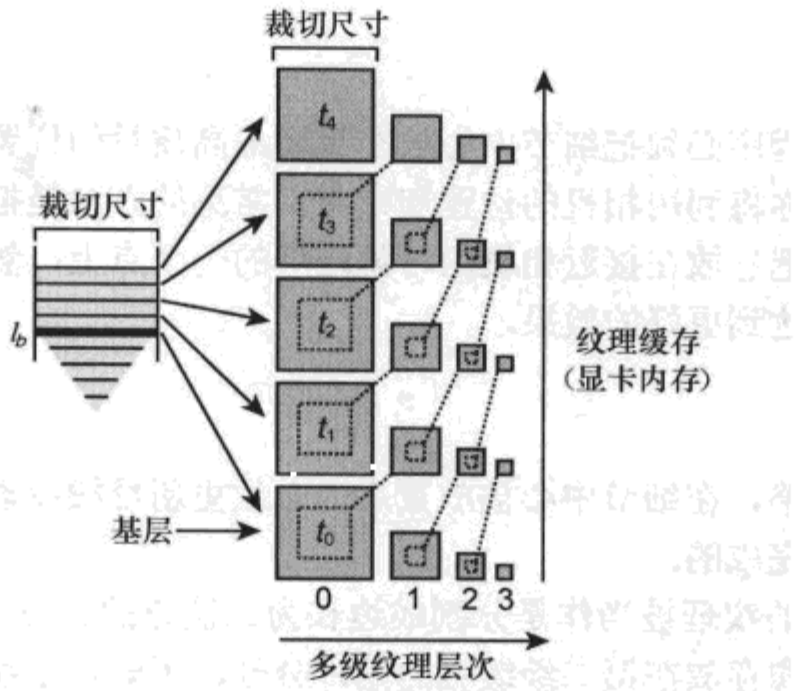


图 5.8.3　有对应 mipmap 层次的纹理堆栈

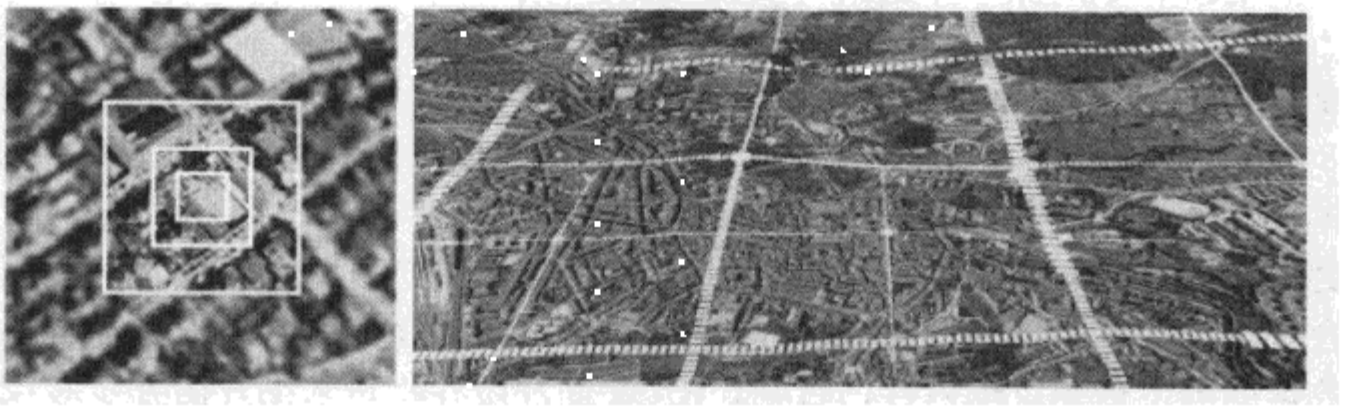


图 5.8.4　细节环和一个应用于地形的虚拟纹理的例子，利用颜色代码来显示细节层次

纹理内存的使用

对一个裁剪尺寸为 c 的层次 l，堆栈纹理有 m 个 mipmap 层次，b 字节纹素深度的虚拟纹理，缓存的纹理内存使用可如下计算。

$$texture\_memory = \left((l - l_b - 1) \cdot \sum_{i=0}^{m-1} \left(\frac{c}{2^i} \right)^2 + \sum_{i=0}^{l_b} 2^{2i} \right) \cdot b \qquad (5.8.1)$$

金字塔的较高层可以是不完整的，允许在不变的整体图像细节上对特殊兴趣区域增加额外的细节。这在像飞行模拟器的游戏中是很普遍的，所有的地形都有一个中等细节的卫星纹

理，在飞机可能低飞或靠近的地方（如机场），细节会增加。

5.8.3 更新缓存的内容

存储在纹理缓存的数据对应于围绕细节中心的、被虚拟纹理所覆盖的地形的一个区域。当细节中心移动后，必须更新缓存的内容。

细节中心的计算

在每一帧，应用程序必须把细节中心放在想要最高质量的位置。有好几种策略都可以被使用。通常情况下，你将利用相机的位置和方向。常见的办法是把细节中心放在相机在地面的垂直投影上。通过把它放在接近相机的可见地形的一个点上，然后计算它在视线方向和地形的交叉点，就可以达到更好的效果。

纹理栈更新

不管使用何种策略，在细节中心被放置后，必须更新堆栈纹理层次。每个层次的更新是按照从粗到细的顺序完成的。

对应于这些层次的纹理被当作是分割成边长为二次方的正方形区块（block）。这些区块被称为子贴片，用来和存放在第二级缓存的贴片分开。子贴片是纹理更新的最小单元，它们的大小必须是裁剪尺寸（c）和贴片尺寸（t）的约数，其中：

$$s=2^i, t=2^j, c=2^k, \text{且} i<=j,\ i<k \tag{5.8.2}$$

当细节中心移动后，一些子贴片将会变成无效，因而必须被更新，而另一些将保留有用的数据。对于每一个纹理，用一个子贴片状态矩阵来表明纹理中的每一个子贴片的有效性。这些矩阵将随着细节中心的位置改变而被更新。

在更新状态矩阵之后，每个纹理按细节从粗到精被处理。对每一个无效的子贴片，你计算包含子贴片数据的贴片地址。这个贴片是从第二级缓存请求的。如果它是驻留的，子贴片的数据就被上传到纹理内存，否则，RAM 贴片缓存将请求贴片的异步装载，在随后的几帧，数据就可被使用了。对不完整的级别，不在持久存储器的无效子贴片将从不被更新。

在相应的真实纹理上，缓存在每个堆栈层次的虚拟纹理窗口被环形地访问，这允许每个层次的部分更新，从而大大提高了效率。如图 5.8.6 所示，当细节中心被更新后，窗口位置也发生变化。使用环绕式（wraparound）寻址，只有不在以前窗口位置的新的子贴片需要被装载，而重叠区域在原地保持不变。

纹理子贴片的更新意味着更新纹理的每一个 mipmap 层次的相关区域。在 mipmap 层次的更新过程中，考虑对应于 mipmap 层次 m 的子贴片的大小为 $s/2^m$。如图 5.8.3 所示，因为 mipmap 层次的数据是重复的，所以层次 t_{ij} 的更新可以用粗纹理来完成，其中 $j>0$。

负荷控制

对于像游戏的实时图形应用，为了维持帧速率，纹理的上传时间至关重要。渲染时间加上

纹理更新时间不得超过帧时间。出于这个原因，子贴片的更新被限制在每个帧的持续时间内，这也意味着，对细节中心的快速运动，在单一帧内达到最精致的细节将是不可能的。通常，这不是一个问题，因为快速运动通常不需要让观众欣赏图形的细节，模糊的外观通常是可以接受的。

当决定子贴片大小时，需要同时兼顾负荷控制和传输速率。子贴片越小，测量的更新时间越准确。尽管子贴片的更新时间主要依赖于硬件，但是较小的尺寸通常会导致效率低下。我们的经验表明，128×128 的子贴片尺寸给出最佳的性能。

同心圆环更新

由于前面提到的更新时间的限制，堆栈中的纹理并不总是完全更新，这需要决定纹理在什么时候已经有足够的可被使用的数据，最简单的办法是在堆栈中只使用完全更新的纹理。这里的问题是，每次细节中心移动超过一个子贴片的距离时，它将变成无效的，直到再被完全更新。如果允许使用只有部分区域被装载的纹理，就可以在一定程度上减轻这个问题。

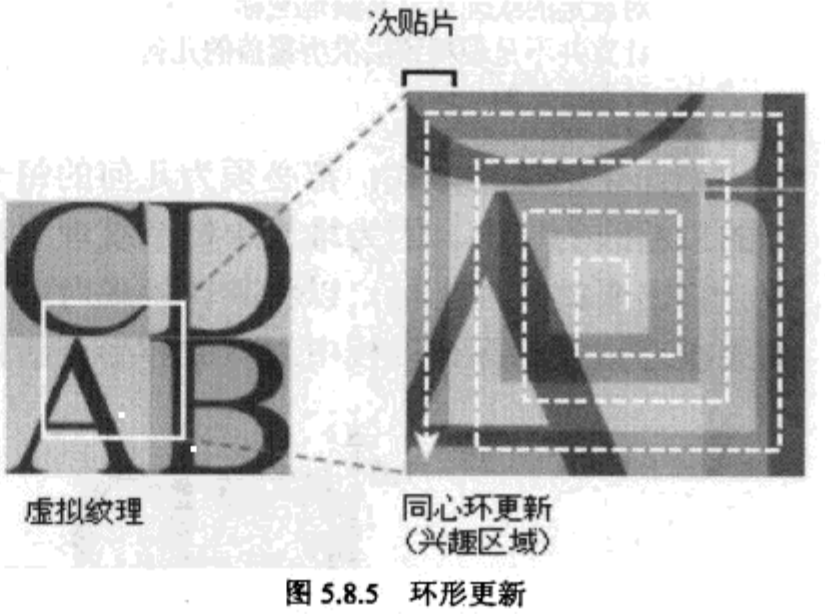


图 5.8.5　环形更新

从最内到最外层，你以同心圆环方式更新每个纹理的子贴片，以至于覆盖面积随着子贴片环的更新而增大（见图 5.8.5）。这样，当纹理开始有一些有效的子贴片时就可被利用了。从中心开始，最高兴趣区域较早可用，此外，中心子贴片的生命周期也相对较长。

伪代码

以下伪代码归纳了虚拟纹理的更新。

```
计算细节中心的位置
对于堆栈的每个纹理层次
    更新子贴片的有效性矩阵

按细节从粗到精，对堆栈的每个纹理层次
    对于层次的每个子贴片（从最内到最外层），并且没有超过更新的时间限制
        如果子贴片状态无效
            计算磁盘上贴片的地址
            请求贴片到 RAM 贴片的缓存
            如果贴片是缓存的
                更新所有 mipmap 层次的子贴片
            将子贴片状态设置为有效
```

5.8.4　渲染问题

可以通过结合这里所描述的技术来调整几何管理算法。映射一个虚拟纹理到一个几何模型有两种可能的方式。考虑到几何模型是被分成面片的，第一种方式是使用覆盖每个几何面

片的最精致的可用纹理。在这种情况下，你将遵循下列步骤。

```
对于每一个几何面片
    使用覆盖面片的最精致的纹理层次
    对选定的纹理计算其纹理坐标
    绘制面片
```

第二种方式是选择每个纹理层次，计算其覆盖范围，并绘制选定的但不是最精致的层次所覆盖的几何。在这种情况下，你将按照以下步骤进行。

```
对于每个纹理层次
    选择和使用纹理
    对选定的纹理计算其纹理坐标
    计算并不是最精致层次所覆盖的几何
    绘制计算的几何集
```

无论使用哪种方式，都必须为几何的每个顶点计算纹理坐标，而这些坐标对应着虚拟纹理的最精致的层次。因为堆栈中每个纹理层次覆盖其较粗层次的一半虚拟空间，所以需要缩放计算出来的纹理坐标，以将其转换成对应的虚拟纹理层坐标。层次 i 的缩放系数是 2^{l-i-1}。如果纹理重复，那么堆栈中纹理的环形更新保证映射是正确的（见图 5.8.6）。

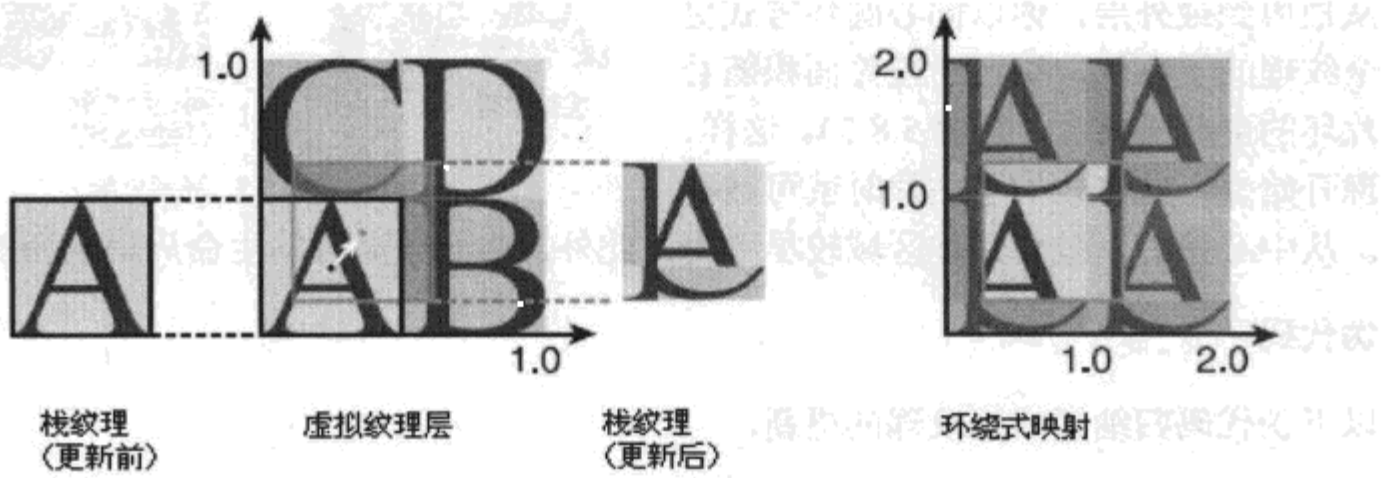


图 5.8.6 环形更新和映射的例子

刚刚描述的纹理坐标的计算可用几种方法完成。对静态几何，纹理坐标可以预计算，并静态存储在纹理的坐标数组。这样一来，所有的计算都可以用纹理矩阵缩放和纹理重复模式完成，因此根本不需要着色器，只需要有 OpenGL 和 DirectX 所具有的标准固定功能图形管道。

对于动态几何，每次顶点被修改时，必须重新计算纹理坐标。在这两种情况下，尤其是动态几何，在顶点着色器中自动计算纹理坐标是非常有帮助的。这样，你避免了它们在 CPU 上的计算、从主存到显存的传送，以及在显存中纹理坐标数组的存储。以下伪代码显示了如何计算纹理坐标。

```
L:左纹理极限；R:右纹理极限
T:顶端纹理极限；B:底端纹理极限
(x,y,z)：顶点的位置，i：所选的虚拟纹理层次
scale = 2^(l-i-1)
u = scale * (x-L)/(R-L)
v = scale * (y-B)/(T-B)
```

如果纹理数据库和几何数据库有不同的坐标系，那么纹理坐标的计算可包括额外的转换。只使用一个 GPU 纹理阶段来映射虚拟纹理，就可以让你轻易地把虚拟纹理与其他虚拟或一般纹理结合起来，每一个绑定到一个纹理阶段。

5.8.5　结果

所提出的技术已用一个私有的地形导航系统进行了测试，所用的数据集包含一个空中地形照片的虚拟纹理，覆盖约 250km×200km 的面积[Santi07]。图形的分辨率是 0.5m/纹素，此堆栈有 19 个层次。

图 5.8.7 显示了一个在低端计算机上执行的压力测试的结果，测试用一个程序设置的在地形上方的 3 000km/h 的飞行，并使用一个大的裁剪尺寸（2 048 平方纹素）和一个仅有 1ms 的更新时间限制。

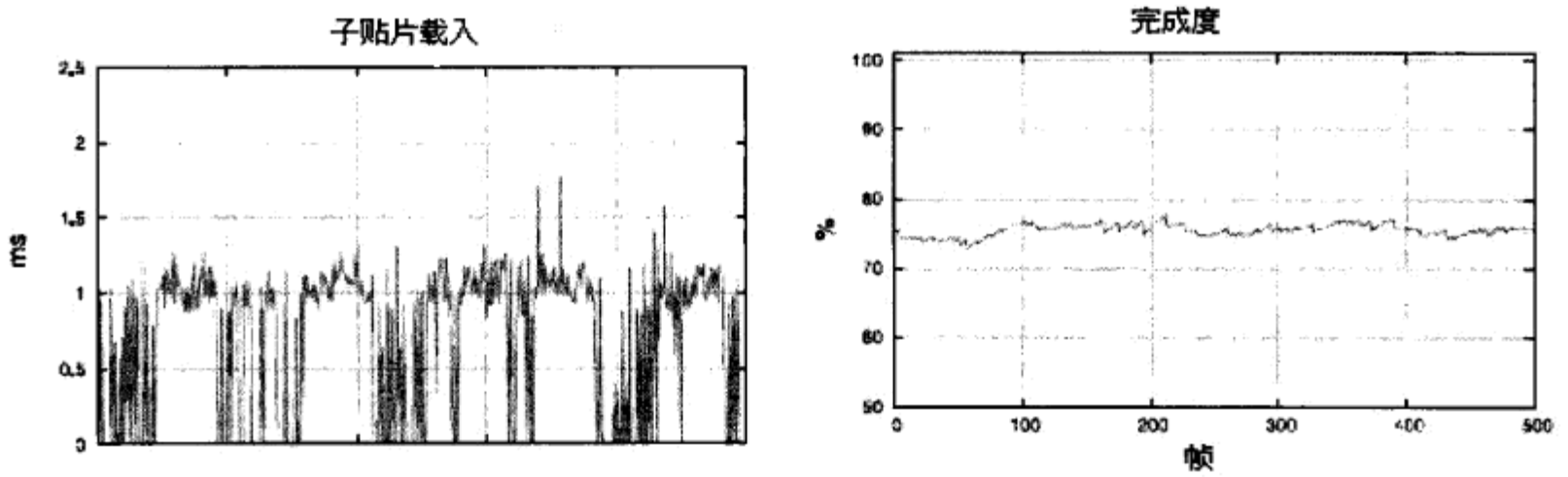


图 5.8.7　测试结果

图 5.8.7 显示了每 6s 间隔的图示，使用的配置如表 5.8.1 所示。第一幅图显示了系统上载纹理子贴片到显存（VRAM）所用的时间。该系统试图将更新时间限制为 1ms，以便留出时间来处理应用程序的其他部分。

第二幅图显示了纹理栈的完整度，它可以用来衡量系统所显示纹理的质量。不管测试的压力条件，它有 75%左右的完整度，这意味着 1m/纹素的纹理细节。

表 5.8.1　测试系统配置

图 形 硬 件	AGP 8X NVIDIA GeForce 7 800 GS
裁剪尺寸	2 048 纹素
贴片尺寸	512 纹素
子贴片尺寸	128 纹素
更新时间限制	1ms
虚拟纹理尺寸	约 1m×1m
虚拟纹理颜色深度	24 位（RGB888）
过滤	三线性过滤
各向异性过滤	4 倍
飞行速度	3 000km/h

表 5.8.2 统计

	最小	最大	平均	标准偏差
每帧的子贴片数	0	4	2.15	1.55
子贴片负荷（毫秒）	0.19	1.09	0.33	0.09
完整度（%）	72.83	77.69	75.53	0.86

表 5.8.2 显示了一些有趣的统计信息，如平均每帧有两个子贴片上传到显存，或平均每帧用 0.33ms 来更新。在一些更有利的情形下，测试结果能平均达到 95%以上的质量（0.5m/纹素），比如在 1ms 的更新时间限制下，将裁剪尺寸降低为 1 024 纹素，这些都证明了该技术的有效性。

5.8.6 结论

本精粹所描述的技术使有效地管理在硬件能力限制之外的大型纹理成为可能。因为可配置的负荷控制，它们可用于多种实时应用。这项技术使用不被纹理直接使用的贴片来存储图像。基于裁剪贴图的想法，这些贴片在缓存兴趣区域的纹理栈被组合。你不必像许多地形可视化技术一样，细分几何成面片来匹配纹理贴片的边界。不管使用何种几何算法，总会有纹理来映射每个面片。

这项技术的限制更多是关于面片大小，而非几何结构、细分或细化。本纹理技术已成功地使用基于网格或 TINs 的不同几何算法实现。虽然在近距离观察大型几何面片时精度会略有下降，但这些通常都是可以避免的。

5.8.7 参考文献

[Montrym97] Montrym, J.S., Baum, D.R., Dignam, D.L. and Migdal, C.J. "InfiniteReality: A Real-Time Graphics System," SIGGRAPH 97, Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, pp.293–302. ACM Press/Addison-Wesley Publishing Co, 1997.

[Santi07] The SANTI Project Web Page.

[Tanner98] Tanner, C.C., Migdal, C.J., and Jones, M.T. "The Clipmap: A Virtual Mipmap," In Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques, pp. 151–158. ACM Press, 1998.

[VTerrain07] The Virtual Terrain Project Website.

[Williams83] Williams, L. "Pyramidal Parametrics," SIGGRAPH 83, Proceedings of the 10th Annual Conference on Computer Graphics and Interactive Techniques, pp. 1–11. ACM Press, 1983.

5.9　基于艺术品的嫁接贴图渲染

Joshua A. Doss，先进的视觉计算，英特尔公司
joshua.a.doss@intel.com

在计算机图形学中，嫁接体（graftals）是用来描述植物形态和形成的一种正式语法。作为分形体（fractal）的一个近亲，嫁接体允许树叶的紧凑表示[Smith84]。在交互式帧频[Kowalski98]中，嫁接体也被用于实现非真实感的卡通渲染。利用现代 GPU 的几何着色器，嫁接贴图被作为一种实时渲染方法，用来绘制卡通风格的植物和毛皮。我们想要的特殊风格是受启发于 Dr. Seuss 的儿童书籍插图[Seuss71]。

一个艺术家将提供嫁接贴图的素描，一套把树叶放到场景的材质以及对最终的外观和感觉的许多控制。替代者被沿着物体的轮廓边缘放置。对一个全彩的例子，见颜色板 12。

5.9.1　资产

除了几何之外，创建嫁接贴图还需要一系列的美术资产，如一个纹理集、控制纹理以及向量场纹理。纹理集包含 3 种类型的嫁接贴图和各种类型的几个变化。控制纹理提供了在网格上的某些位置应该放置哪种嫁接贴图之类的信息。向量场提供了一个方向和关于景观网格的着色以及嫁接贴图信息的彩色纹理。

纹理集

纹理集包含嫁接贴图本身，它是通过在不同的行指定一些不同类型的嫁接贴图而创建的。每个嫁接贴图之外的 RGB 和 Alpha 值全为 0。当靠近嫁接贴图的软边时，Alpha 值应平滑地从 0 增加到在笔划中间的 1，给用户一个平稳的过渡并减少任何锯齿效果。如图 5.9.1 所示，红、绿和蓝通道应保持为 0，直到 Alpha 通道达到饱和。

当达到轮廓（outline）的中间时，它光滑地融合成红色，而 Alpha 通道全饱和。嫁接贴图的内部将和底层几何的颜色相融合，而外部将和场景的其余部分相融合。

控制纹理

控制纹理让设计师可以指定一个嫁接贴图该放在哪里，以及绘制什么类型的嫁接贴图。Alpha 通道是用来指明不可以绘制嫁接贴图的地方。如果

想用纹理集中第一行的嫁接贴图，设计师应该使用红色；第二行用绿色；第三行也就是最后一行用蓝色。虽然用 3 种颜色通道不是最佳的编码方法，但它简化了美术资产的创建过程。

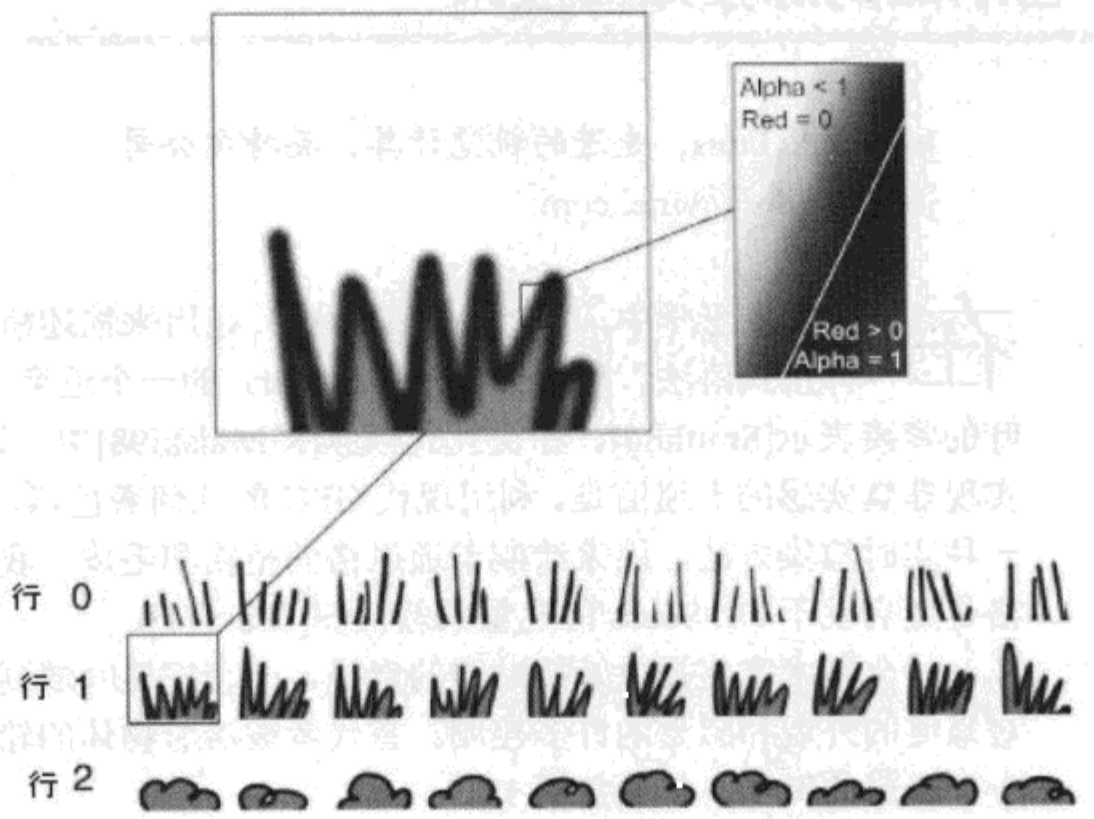


图 5.9.1 纹理集使用红色作为嫁接贴图内部的颜色键，并用 Alpha 通道光滑地融合嫁接贴图和其余的场景

向量场

向量场通常用于表明一个方向，或嫁接贴图的流动。有这样一个例子，在角色上使用这一技术创建毛皮（或头发），你可以使用法线作为挤压方向，然而，这限制了终端用户对场景最后外观的控制数量。头发、植物和树木等并不总是从它们的伸出表面以直角向外增长的。为了解决这个问题，你可以创建提供方向的矢量场（见图 5.9.2）。

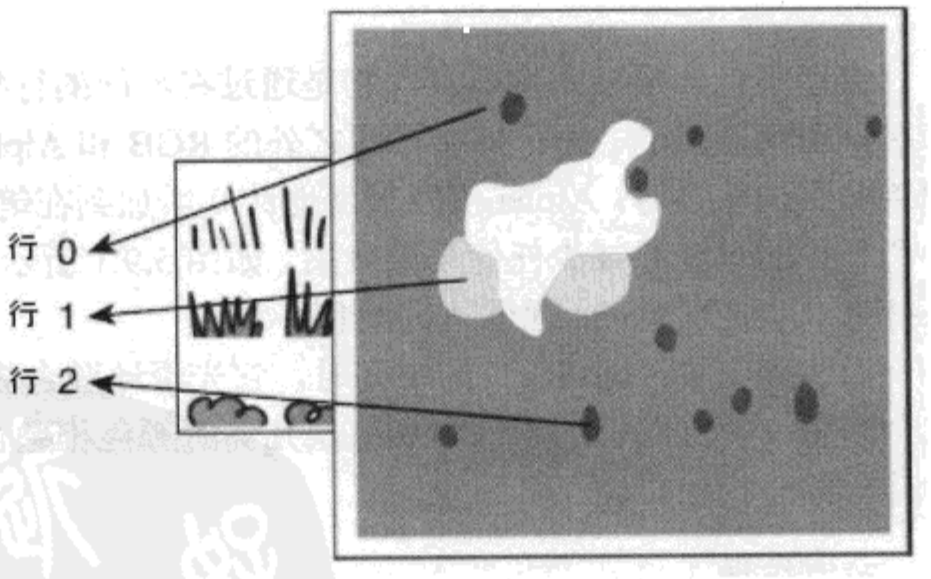


图 5.9.2 表明覆盖和行选择的控制纹理

你利用现有的数字内容创建应用程序和插件去创建矢量场。在创建一个理想分辨率的网

格后，设计师可以把它存起来，大大降低网格的细化。因为原始法线将在另一个步骤中被使用，所以需要保留它们，并和网格一起传递。下一步，设计师调整法线来显示嫁接贴图的延伸方向。这可以基于每面或通过同时选择几个法线完成。一旦操作完毕，这些新的值可以用一个创建矢量场的法线贴图插件来保存，结果见图 5.9.3。

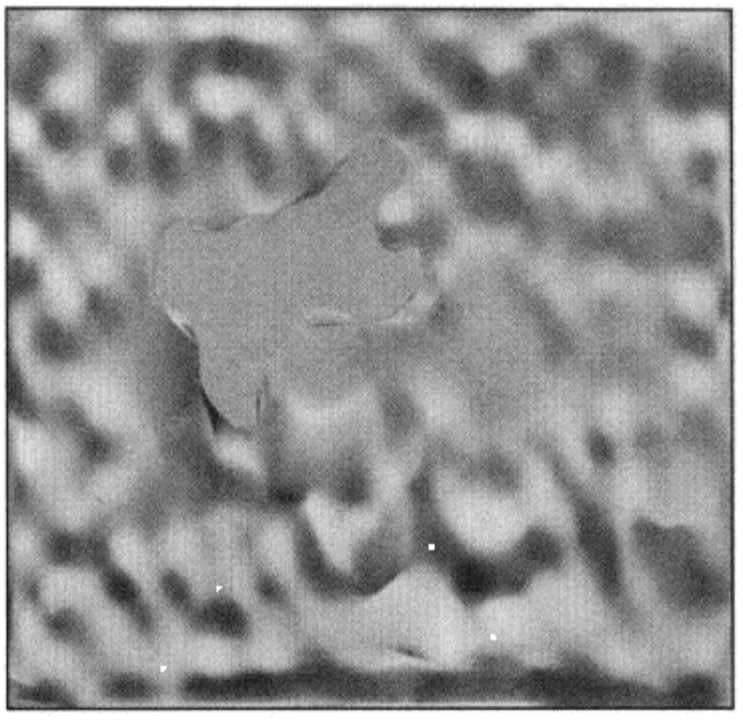


图 5.9.3　表明嫁接贴图方向的矢量场纹理

颜色纹理和网格

颜色纹理用来指定网格的颜色以及嫁接贴图内部的着色。图 5.9.4 是场景的颜色纹理。这项技术沿着边缘来创建嫁接贴图。较大的边长差会在嫁接贴图的宽度上导致明显的视觉缺陷。因此，网格应包含大小较均匀的三角形。在需要非常小的三角形的高细节区域，或许最好使用控制纹理而省略嫁接贴图时创建。

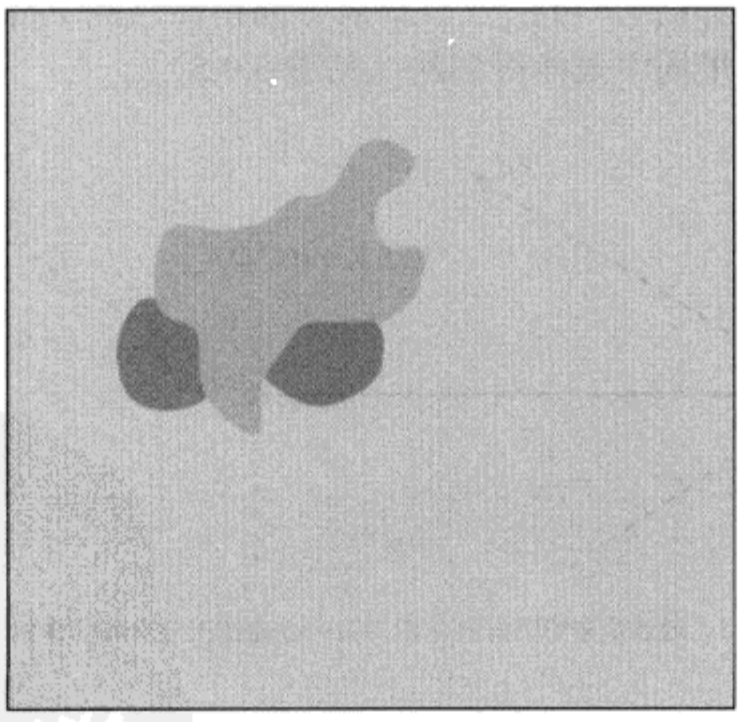


图 5.9.4　着色基础网格和嫁接贴图的颜色纹理

5.9.2 运行时

现在，你可以在算法的运行元素中使用在上一步中创建的美术资产，这个嫁接贴图的实现需要使用可编程图形语言的几何着色器。首先，绘制原始网格及应用颜色纹理。接下来，用几何着色器来确定在哪里放置以及放置哪种类型的嫁接贴图。最后，使用像素着色器来确定嫁接贴图的最终颜色，并将它们与场景的其他部分相融合。

前面所创建的控制纹理用来表示可以在哪里创建以及在某一特定区域绘制什么类型的嫁接贴图。你需要测试三角形来确定图元是否应该有嫁接贴图，如果有，则赋予一种类型。

```
texCoordCentroid = ( vertex1uv + vertex2uv + vertex3uv ) / 3;
controlSample = controlTexture.sample( sampler, texCoordCentroid );
if( controlSample.a == 1 )
    if( controlSample.r == 1)
        glyphType = 0;
    elseif( controlSample.g == 1)
        glyphType = 1;
    else
        glyphType = 2;
```

Direct3D 10 提供的统一指令集允许从几何着色器采样纹理。因为可以访问多个顶点，所以需要选择纹理采样的一个点。前面的伪代码表明，你可以使用正在处理的三角形的中心点来采样控制纹理。

现在，你已经知道三角形是否应该有嫁接贴图，需要测试它的每个边，看它们是否是轮廓边。一个轮廓边缘就是一个前向和后向三角形所共有的边。为了测试一个边是否是轮廓边，需要对两个面计算面法线 N_1、N_2 与视线方向 V 的点积，看它们的符号是否不同[Lake00]。

$$(N_1 \cdot V) * (N_2 \cdot V) \leqslant 0 \tag{5.9.1}$$

要在轮廓边缘创建新的几何，可在一个 D 的方向挤出顶点 V_0 和 V_1。方向 D 是通过用边的中点 M 的纹理坐标从向量场中采样得到的（见图 5.9.5）。

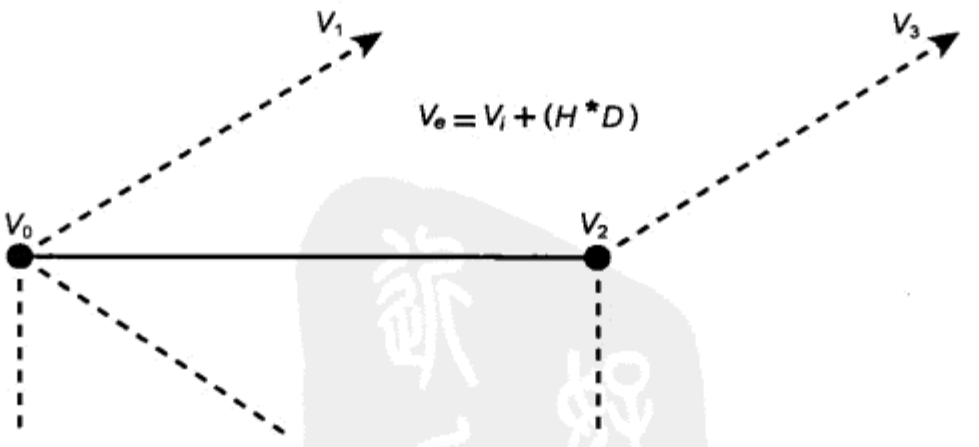


图 5.9.5 通过沿着想要嫁接贴图的边的方向挤压顶点而创建新顶点

```
// 重建原始顶点 V0
Position = Input.Position;
```

```
Output.Position = mul(Position, ObjectToProjection);
Output.GraftalImposterColor = V0Color = ColorTexture.sample (sampler, Input.V0.Texcoord);
...
AppendVertex();

// 在适当的方向创建一个新的顶点
Position = Input.Position * Direction + GraftalHeight;
Output.Position = mul(Position, ObjectToProjection);
Output.GraftalImposterColor = V0Color;
...
AppendVertex();

// 重建原始顶点 V1
Position = Input.Position;
Output.Position = mul(Position, ObjectToProjection);
Output.GraftalImposterColor = V1Color = ColorTexture.sample (sampler, Input.V1.Texcoord);
...
AppendVertex();

// 创建最后的新顶点，完成四边形
Position = Input.Position * Direction + GraftalHeight;
Output.Position = mul(Position, ObjectToProjection);
Output.GraftalImposterColor = V1Color;
...
AppendVertex();
```

这个伪代码表明了如何创建嫁接贴图表面以及采样颜色纹理来着色嫁接贴图。因为当颜色传递到像素着色器时，它将被插值，所以，对于每一个传入顶点选择一次颜色，允许你有一个跨越颜色边界的嫁接贴图。

接下来，你给新创建的几何指定纹理坐标，通过索引纹理集来在新创建的表面放置嫁接贴图。用嫁接贴图类型 G 来索引纹理的正确行，用一个伪随机值，如噪声纹理的采样值来确定列 C。N_C 是纹理集包含的变化的数量或列（见方程 5.9.2～5.9.5）。

$$uv_{v_0} = \frac{C}{N_C}, \frac{G}{3} + \frac{1}{3} \tag{5.9.2}$$

$$uv_{v_{0New}} = \frac{C}{N_C}, \frac{G}{3} \tag{5.9.3}$$

$$uv_{v_1} = \frac{C}{N_C} + \frac{1}{N_C}, \frac{G}{3} + \frac{1}{3} \tag{5.9.4}$$

$$uv_{v_{1New}} = \frac{C}{N_C} + \frac{1}{N_C}, \frac{G}{3} \tag{5.9.5}$$

最后一步是用在几何着色器中计算的纹理坐标来采样纹理集。你想有一个从嫁接贴图的黑色轮廓到被顶点着色器传入的内部颜色的平滑过渡。为了达到这一点，结果的红色通道被遮盖掉，样本用由几何着色器传递过来的颜色线性插值。红色通道用做插值的混合系数。

```
// 给嫁接贴图着色
AtlasColor = AtlasTexture.Sample(Sampler, GraftalImposterTextureCoords);
```

```
RedZero = float3(0,AtlasColor.gb);
GraftalImposterColor = lerp(RedZero.rgb, IncomingColor.rgb, AtlasColor.rrr);
GraftalImposterColor.a = AtlasColor.a;
```

横跨采样的两个顶点，传入的颜色在几何着色器中被插值。在红色通道被遮盖掉时，通过做一个线性插值，就可以融合嫁接贴图的软边。保留 Alpha 值允许你融合光滑边的外部和底层的景观颜色。对一个用嫁接贴图渲染的全彩色例子，参见颜色板 12。

5.9.3 感谢

作者要感谢 Jeffery A. Williams、Rahul Sathe、David Bookout、Nico Galoppo、Adam Lake 和英特尔的先进视觉计算小组，感谢他们的援助、支持和贡献。

5.9.4 结论和未来的工作

本精粹表明了如何创建一个风格类似于 Dr. Seuss 的儿童读物的场景。这个以 GPU 为中心的技术利用了几何着色器的新技术能力，给游戏逻辑和其他任务留下更多的 CPU 周期。目前，我们仅在轮廓边应用嫁接贴图。在今后的工作中，我们要自动生成用于挤压方向的矢量场，而不是要求一个艺术家去编码整个几何。

在生产环境中，实现该技术也许需要一些额外的功能。一个可能的方案是，通过在嫁接贴图中缩放或“逐渐衰减”，调整嫁接贴图的引进和消除来提供帧间的连贯性。另一个要考虑的重要因素是 z-fighting 的处理。

5.9.5 参考文献

[Doss07] Doss, Joshua A. “Inking the Cube: Edge Detection with Direct3D 10,” *Game Developer Magazine*, June/July 2007, pp. 13–18.

[Kowalski98] Kowalski, Michael A., et. al. “Art-Based Rendering of Fur, Grass, and Trees,” Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 1998, pp. 433–438.

[Lake00] Lake, Adam, et. al. “Stylized Rendering Techniques for Real-Time 3D Animation and Rendering,” Proceedings of the 1st International Symposium on Non-photorealistic Animation and Rendering, NPAR, 2000, pp. 13–20.

[Rost06] Rost, Randi J. *OpenGL Shading Language*, Addison Wesley, 2006.

[Seuss71] Seuss, Dr. *The Lorax*, Random House, Inc., 1971.

[Smith84] Smith, Alvy Ray. “Plants, Fractals, and Formal Languages,” *Computer Graphics*, Vol. 18, No. 3, SIGGRAPH 1984, pp. 1–10.

5.10　廉价的对话：动态实时口型同步（Lipsync）

Timothy E. Roden, Angelo State University
troden@angelo.edu

对于游戏中的 3D 人物，游戏开发商越来越多地使用口型同步。现在的问题是：设置口型同步并让它运行起来可能是费时的，而且比较昂贵。一个自定义的解决方案可能需要程序员的宝贵时间，而更权宜的方法涉及购买中间件，可能有其他的缺点。也许是作为概念验证演示的一部分，特别是对于想尝试口型同步的开发商来说，更快、成本更低的解决方案比较理想。幸运的是，你可以便宜地在一个游戏中加入口型同步，而且是在一个最短的时间内。至少，其结果对概念验证是足够了，对一个包装游戏也可能足够了。本精粹将说明一种既快又简单的口型同步方法。

5.10.1　需求

为了使用此方法，一些一般性的要求必须得到满足。首先需要一个 3D 角色。因为在例子中需要做嘴唇动画，所以至少需要一对嘴唇，最好是整个头部。本精粹的例子使用了一个用 Singular Inversion 的 FaceGen®软件生成的头部模型。图 5.10.1 显示了我们使用的头部模型。不包括头发，该模型由 7 341 个顶点和 1 2960 个三角形组成。

头部模型需要有一些可以动态调整的、控制口部位置的控制参数。一套形态目标就很够了。如果不知道形态目标如何运作，Lever 提供了一个很好的解释[Lever02]。一件好的事情是：FaceGen®模型配备了大量的面部表情和嘴唇位置的形态目标，它们对应于语音的各种基本单位。如图 5.10.2 所示，这里用的头部有视素的 16 个形态目标，它们有诸如语音“aah”和“ee”的视觉表达。Watt 和 Policarpo 将视素描述为视觉语音的基本单位，视觉语音用极端的嘴唇形状来描述，这对应于基本的听觉语音单位[Watt03]。一套视素构成了一个最少量的独特集，它代表了一个语言的声音。

也许，你可以想象出比图 5.10.2 所示的 16 个更多的嘴唇位置。然而，在这里，这一套最低限度的视素实际上已经相当不错了，它们将允许你生成令人非常信服的口型同步动画。

图 5.10.1 一个用 Singular Inversion 的 FaceGen®软件生成的头部模型

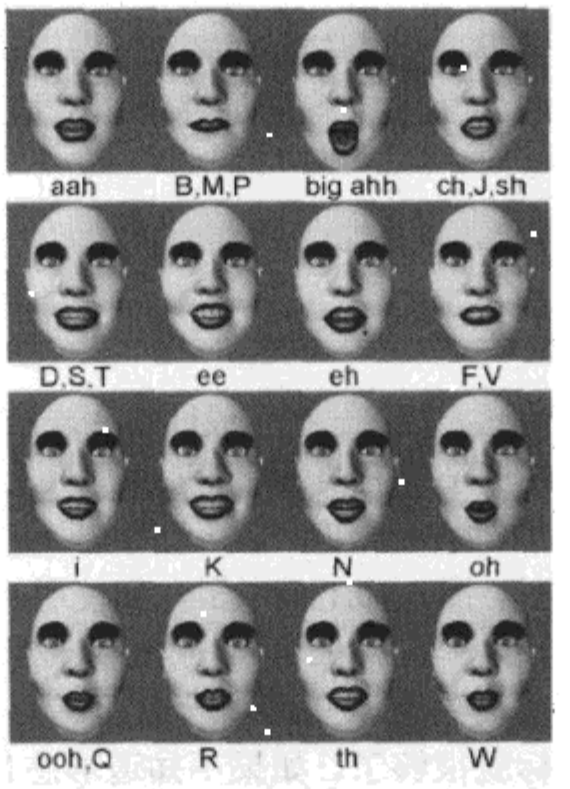


图 5.10.2 本精粹所用的 16 个视素，每个在其极端（1.0）形态显示。见彩色插图板 13

在程序中，你需要能使用范围在 0.0～1.0 的浮点值来独立地调整 16 个视素中的每一个。值 0.0 有效地关掉视素，而值 1.0 意味着视素处于它的最大强度。图 5.10.3 说明了值 0.0 如何没有改变口的位置，而更高的值使得口腔变形成想要的形状。这个例子允许任何一套组合。举例来说，你可以通过应用 1.0 的“aah”视素和 0.5 的“ee”视素的组合来改变嘴的形状。事实上，这个功能至关重要，它使你能够生成真实感的动态口型同步。

对于音频，你可以使用预先录制的或在运行时音频生成的语音，如文本到语音引擎的输出。你还需要正在被讲出的文本。使用一个文本到语音的引擎是很不错的，因为这样可以通过文本字符串来动态地生成音频，同时得到文字和音频。

图 5.10.3　在不同值的“aah”视素（左起为 0、0.33、0.66 和 1.0）

5.10.2　一般过程

对于每个口型同步的音频样本，一般运行程序如下：

（1）将文本的每个字翻译成与其对应的音素集；

（2）将每一个音素翻译成与其对应的视素；

（3）基于视素集生成动画数据；

（4）开始播放音频；

（5）在播放音频的同时，利用动画数据来驱动 3D 模型。

本书附带光盘中包含了一个为静态库写的 C++源代码，实现了这一过程的（1）～（3）步。

词到音素的映射

音素不同于视素，它是语音如何被以语言的方式所听到的不同基本单元。基于组成词独特的声音，不同的词可分成对应的音素。在另一方面，视素是语音的基本可视单位。音素和视素之间有很密切的对应关系。通常，一个语言的音素比视素多，这是因为几种不同的声音可以用同样的嘴唇位置来表达。例如，在听觉上，“s”和“z”是不同的声音，但嘴唇的位置可以是相似的。

将词翻译成音素没有比用 Carnegie Mellon Pronouncing Dictionary 更方便了[CMU07]。CMU 字典可在线使用，而且可以被毫无限制地用于任何研究或商业目的。它是一个含有超过 118 000 个英文单词及其相应语音翻译的文本文件。例如，词“hello”转化为 4 个音素：HH、AH、L 和 OW。CMU 字典共有 39 个不同的音素。表 5.10.1 列出了每个音素和在字典中找到的使用此音素的一个例子。因为在文本文件中字典已经是按字母顺序排列的，所以将字典读到一个数组，执行二分检索去查找单词和检索与其对应的音素，是一件相当简单的编程任务。

表 5.10.1　　39 个 CMU 音素

音　素	例　子	（例子的）转化
AA	Odd	AA D
AE	At	AE T
AH	Hut	HH AH T

续表

音　素	例　子	（例子的）转化
AO	Ought	AO T
AW	Cow	K AW
AY	Hide	HH AY D
B	Be	B IY
CH	Cheese	CH IY Z
D	Dee	D IY
DH	Thee	DH IY
EH	Ed	EH D
ER	Hurt	HH ER T
EY	Ate	EY T
F	Fee	F IY
G	Green	G R IY N
HH	He	HH IY
IH	It	IH T
IY	Eat	IY T
JH	Gee	JH IY
K	Key	K IY
L	Lee	L IY
M	Me	M IY
N	Knee	N IY
NG	Ping	P IH NG
OW	Oat	OW T
OY	Toy	T OY
P	Pee	P IY
R	Read	R IY D
S	Sea	S IY
SH	She	SH IY
T	Tea	T IY
TH	Theta	TH EY T AH
UH	Hood	HH UH D
UW	Two	T UW
V	Vee	V IY
W	We	W IY
Y	Yield	Y IY L D
Z	Zee	Z IY
ZH	Seizure	S IY ZH ER

音素到视素的映射

音素到视素的转化是一个基于表的直接查找，这个表需要用户事先创建好。字典包含了 39 个单独的音素，而这里所用的 3D 模型有 16 个视素。这样就需要一点创造力来为每一个音素确定正确的视素了。也许，做此任务最简单的方法就是在镜子的前面，利用表 5.10.1 朗读每个示例词，留意当你试探词中特别音素时嘴唇处于什么位置。用图 5.10.2 中最接近的视素来匹配你嘴唇的位置。鉴于本精粹的目的，我们将使用映射表 5.10.2。

表 5.10.2　　音素到视素的映射

音　素	视　素	音　素	视　素	音　素	视　素
AA	Big aah	F	F,V	P	B,M,P
AE	Aah	G	Ch,J,sh	R	R
AH	Aah	HH	Eh	S	D,S,T
AO	Big aah	IH	I	SH	Ch,J,sh
AW	Big aah	IY	Ee	T	D,S,T
AY	Aah	JH	Ch,J,sh	TH	Th
B	B,M,P	K	Ch,J,sh	UH	Oh
CH	Ch,J,sh	L	Th	UV	Ooh,Q
D	D,S,T	M	B,M,P	V	F,V
DH	Th	N	N	W	W
EH	Eh	NG	D,S,T	Y	Ee
ER	R	OW	Oh	Z	W
EY	Eh	OY	Ooh,Q	ZH	Ch,J,sh

实时口型同步

在运行时，对每个你想要口型同步的音频样本，你必须将文本字符串转化成音素，然后转化到视素。利用本书附带光盘所提供的代码，这个过程包含两个函数的调用，然后调用第三个函数，把视素转化成在播放音频时用来制作 3D 模型动画的口型同步动画数据。首先，让我们研究如何生成口型同步数据。

可以使用几种具有不同复杂度的方法，更复杂的方法可能提供更准确的数据。然而，鉴于本精粹的目的，让我们使用一个简单的办法。对于其简单性，此方法也给出相当不寻常的结果。这个方法是用那些视素来划分所讲语音音频的持续时间，并在播放音频时给每个视素分配一个活跃时间段。

例如，图 5.10.4 对单词“hello”进行了图解。“hello”是由 4 个视素组成的。在时刻 0，你开始从 0 到 1 变形视素“eh”，然后回到 0。在“eh”变成不活跃之前，你必须开始变形视素“ahh”，依次类推。它的想法是相邻的两个视素之间稍微重叠，这就形成了更多自然的口型同步。

通过改变重叠的程度，你可以实现一些有趣的效果。例如，至少在视觉上，长的重叠时间往往使发言者表现得含糊其词；短重叠时间产生非常清楚的视素（当人愤怒时可以看到这个效果）；太短或太长的重叠都会产生看起来不自然的效果。

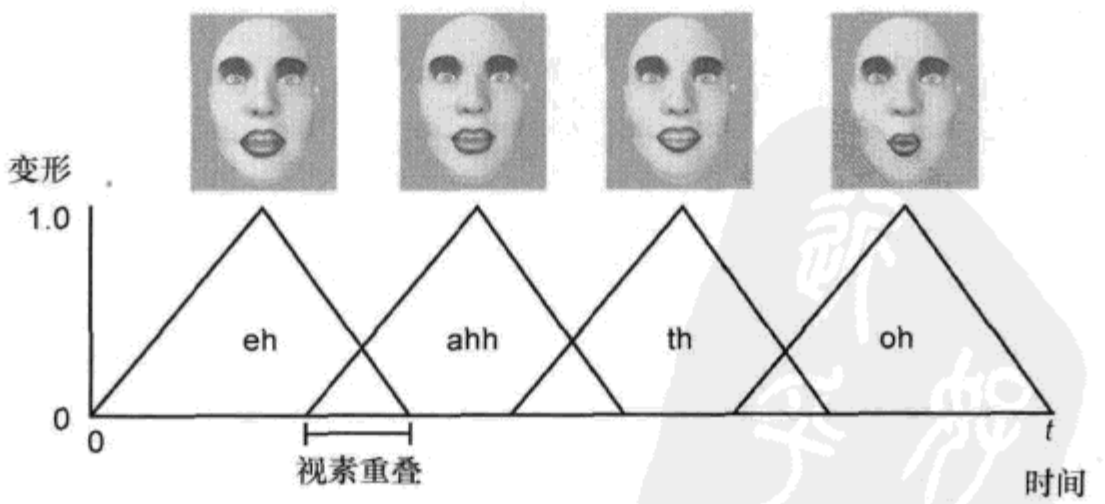


图 5.10.4　单词“hello”和在时间上与其对应的视素动画

此外，还需要解决字词定时的一些细节问题，使用任何办法都是可以的。对于多个句子的音频，文本应包含终止符来表明句子结束的时刻。然后，每个终止可以被分配一个时间段，以至于一个句子的最后视素不会溢出到下一个句子开始处的第一视素。句子末尾的停顿时间可能需要用反复试验法来找到。附带光盘中的代码使用了 500 ms。如果使用文本语音转换引擎，一个诀窍是保存包含多个句子的、几个文本语音转换的音频文件，然后在 WAV 文件编辑器中加以检查。通过观察音频数据，很容易看到句子末尾停顿的长短。

提出的解决办法存在一些明显的缺点。也许最大的问题是关于实际的声音文件。不像文本语音转换引擎通常有一个恒定的速率，人们常常以不同的速率来讲同一句话。因为这里所描述的简单口型同步算法依赖于一个恒定的语音速率，所以这可能会有一些问题。不过，这种方法的优点是可实时地生成口型同步数据，这在你想快速启动和运行口型同步的快速原型环境中是非常有用的。

5.10.3 结论

对一个有经验的程序员来说，使用本精粹提出的方法去创建一个动态的口型同步动画实时系统很可能只是几天的工作。该方法可以被进一步改善来得到更好的结果。一种想法是考虑协同发音效果，也就是对一个特定的声音考虑什么声音在它之前和什么声音在它之后，而改变其音频。[Watt03]给出了关于实现它的想法。

5.10.4 参考文献

[CMU07] The Carnegie Mellon Pronouncing Dictionary.

[Lever02] Lever, Nik. *Real-Time 3D Character Animation with Visual C++*, Focal Press, 2002.

[Watt03] Watt, Alan, and Policarpo, Fabio. *3D Games: Animation and Advanced Real-Time Rendering*, Vol. 2, Addison-Wesley, 2003.

第 6 章

网络和多人游戏

简　介

Diana Stelmack

现在，使用多人模式玩法的游戏类型数量正在飞速地增长。互联网正前所未有地普及到各种平台，家用机也在加入到这个潮流中。家用机开发商正在为他们的用户提供互联网服务，以吸引用户与朋友们一起在线玩游戏。所有的这些网络都意味着需要更多的网络编程，而且所有这些游戏类型以及网络服务的增长都意味着有更多的游戏程序员将面对网络接口。这意味着什么呢？这意味着网络系统，这个游戏的组成部分，将需要更清楚定义的接口，以便非网络程序员来使用，也需要工具来帮助找到关键时刻的 BUG，需要适当的方法来处理越来越多的安全问题，以及更多其他技术。本章包含了一些精粹，可以帮助你解决其中的一个或多个问题。

第一节，由 Hyun-jik Baeb 带来的精粹，描述了一种被称为高层抽象（High Level Abstraction，HLA）这项技术介绍了一种可供开发的工具，以使得非网络程序员与网络引擎打交道更多容易。无论你是想要使网络引擎更为简单易用的网络程序员，还是想要一个更为简单的网络引擎的非网络程序员，都应该看看这篇文章。

众所周知，一个机器上只要有运行程序，就会有一些人想要黑掉它。保障网络安全是一项永无止境的工作，这就意味着网络程序员需要考虑一项保障玩家信息安全的策略。“信息高速公路”上异常繁忙，而用户们并不知道在他们的 PC 和他们所连接的主机之间到底有多少个站。第二节，由 Jon Watte 所撰写的精粹，将探讨海量的安全方法，并展示一种成熟的可解决当今大多数安全需要的方案。

假如有一个需要大量模拟的游戏，这些模拟要减缓帧率来做一些很炫的图形效果。现在，为了更多的游戏乐趣，那些会影响模拟的数据加入了网络延时，因此也会延迟场景的渲染。而同时，网络中的多个人类玩家正在向对方射击，而其中一些玩家想要得到杀戮的奖励。所有的这些情况都将产生大量的网络流量，这些网络包需要在合理的时间内以一个合理的精度从点 *A* 传送到点 *B*。加入一个断点会破坏这个测试现场，除非你拥有一个智能的包嗅探器。第三节，由 David Koenig 撰写的精粹，将探究如何创建一个为游戏定制的包嗅探器，以使得定位网络问题更多容易。当你在调试一个网络中的游戏问题时，理解数据就赢了一半的战役。

6.1　游戏世界同步的高层抽象

Hyun-jik Baeb

网络中游戏主机的一个重要角色就是要与其他主机交流，以保持游戏世界同步，包括要保证世界中所有主机都运行在同样的状态下。要保证游戏世界中多个主机间的同步，游戏程序员需要编写以下代码：

- 收集本地主机的变动；
- 将这些变动打包为一个或多个消息；
- 将这些消息发给远端主机；
- 将这些变动应用到远端主机的游戏世界状态。

可使用一些简便的技术为这些任务编写代码，比如远端程序调用（Remote Prodedure Call,RPC）系统[HyunJik04]。RPC 发送及接收信息的代价很小，只需为每个信息类型编写一行代码。但是，你还是要手动编写管理游戏世界状态的程序，收集同步信息，发送然后处理它。这项工作将飞速膨胀，如果你的游戏策划开发了几百种不同的战役单元，而你的程序结构不能轻易得概括它们。

相对于手动编写代码，元编程[Wikipedia07]的强大能力大大增加了产出率。当然，RPC 也是一种元编码技术。本节将介绍另外一种元编程技术，使用高层次抽象（High Level Abstraction，HLA）来同步游戏世界。RPC 将主机间交换消息的操作抽象为较低代码层级的几行源代码，而 HLA 在较高的层级进行抽象，这个层级交互的消息是用来同步游戏世界状态的。这也就是为什么这项技术被称为高层次抽象。

原始内存同步技术也可以进行游戏世界同步，但它在以下方面有缺陷。

- 实际运行的多人游戏都需隐藏延时技术，比如航位推算法[Aronson97]。同步原始内存没有办法实现这个。
- 原始内存同步需要游戏世界数据都存储在一个内存块中。在使用了自动内存管理和垃圾收集器的情况下，这是很难做到的。
- 在实际的多人游戏世界中，并不是最新数据的每一个字节都需要精确同步。比如说，一个远离视口的单位并不需要全精度的同步。

在 HLA 世界中，游戏世界同步可通过为每一个游戏对象声明对象类型和同步行为来实现，而不需要直接编写发送或接收消息的代码。而实际的代码是由本节提供的源代码生成器自动生成的。

本节将讨论一个 HLA 用例，并解释游戏世界同步的整体系统，之后将创建一个 HLA 系统。

6.1.1 HLA 用法

HLA 的目标就是提供一种可行的方法来抽象游戏世界的同步。它是由对象类型定义、它们的同步行为和一个确定每个对象可见性的设备组成的。

同步对象的定义是以一个你定义的语法存储在一个源文件中，你可以称之为 SWD（Synchronized World Define，同步世界定义）文件。它将被编译成一些源文件，然后与你的工程文件一起构建。

确定同步范围的设备实际上将是一个函数，你可以使用不同的方法拓展它。

6.1.2 游戏世界同步剖析

因为这个例子包含了如何编写你自己的 HLA 基础，所以当你要把 HLA 技术应用到自己的游戏工程中，是不会有任何限制的。本小节假设网络为客户端/服务器布局。该布局解释如下。

- 游戏主机由一台服务器和其他一些客户端组成。服务器拥有所有的游戏世界对象并控制它们。
- 当客户端上游戏世界发生改变时，一条或多条消息从客户端发送给服务器，然后它们被应用到服务器的游戏世界上并广播到其他客户端更新。
- 当服务器上游戏世界发生变化时，消息从服务器发给客户端。客户端收到消息后基于这些变化更新本地的游戏世界。

图 6.1.1 解释了这个协作过程。

你可以将游戏世界状态的变化分类。以下是发送消息的条件：

- 一个游戏对象的属性改变；
- 一个对象被创建；
- 一个对象被销毁；
- 一个对象出现或消失（将在之后讨论）；
- 每一个时间间隔。

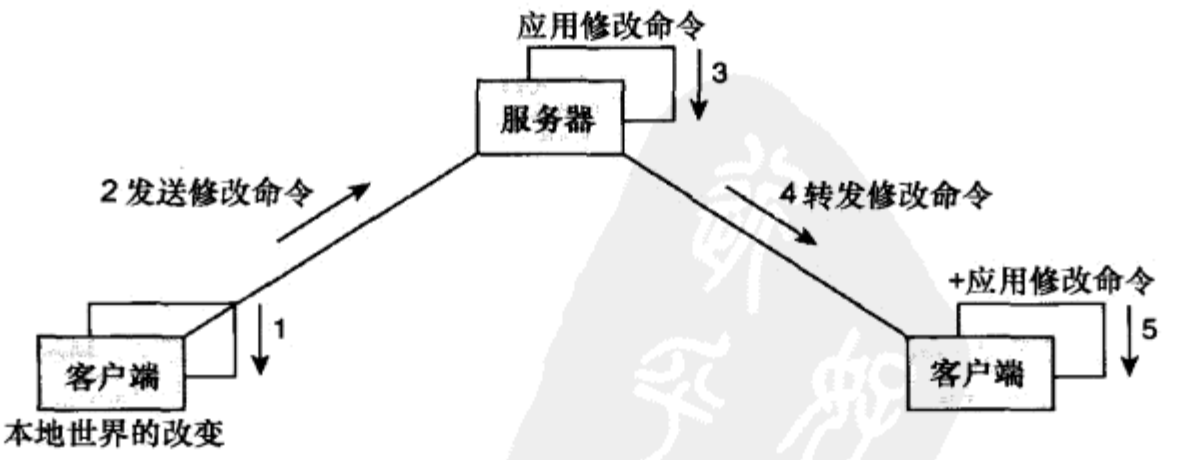


图 6.1.1 HLA 协作图

当数据改变非常频繁，而不是每一个改动都必须发送，条件“每一个时间间隔”是必需的。一个很好的例子就是人物的位置。这类改动可以通过一个不可靠的消息协议来广播，比如用户数据报协议（User Datagram Protocol，UDP）。

实际上在许多游戏产品中，由于网络带宽的限制，不是每一个游戏对象都同步到每一个远端主机。这对大型多人游戏尤其重要。在大型多人游戏中，每一个客户端只保持游戏世界状态的一小部分，而服务器保持所有的游戏世界状态（如果服务器是一个分布式系统，甚至任何单个服务器端的游戏世界也是不完整的）。每一个主机覆盖的同步范围是由每一个游戏对象独特的规则所确定的。

图 6.1.2 展示了一个同步范围的例子。这个例子使用一个以每个观察者为中心指定半径的圆来裁剪需同步的对象。一个圆表示一个主机的视口，而每个星星表示一个需要同步的对象。当在视口外的一个对象进入视口或者一个视口接近它并探索它时，这个视口的主机将收到“一个新对象出现了”的消息，然后这个主机在它的游戏世界状态中创建一个对象。相反的，当移动了视口或对象而导致一个对象离开这个视口时，“消失”消息将发送到适当的主机上。

会破坏游戏世界的修改必须被禁止。比如，谁也不想自己心爱的装备被敌方删除。你可以将修改许可分级，如表 6.1.1 所示。

表 6.1.1　　世界状态修改许可

	允许服务器改动	允许本地主机改动	允许远端主机改动
Server-only	是	否	否
Server-and-local-only	是	是	否
Everyone	是	是	否

当主机收到改动消息时，并不是完全按照改动消息中的数据来更新游戏世界的，而是以插值的方式来更新。较受青睐的技术是航位推算法[Aronson97]。

现在你已经理清了游戏世界同步的逻辑顺序，你可以依照这个逻辑来实现你的 HLA 基础。这是个同步世界状态的例子，而你可能希望确定你的游戏工程所需要的同步系统，从而设计自己的 HLA 基础。

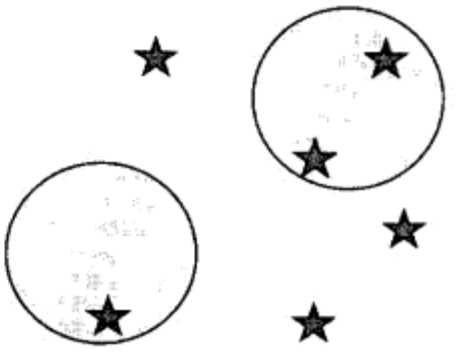


图 6.1.2　视口和对象

6.1.3　HLA 组件

HLA 系统包括一个 SWD 编译器和一个 HLA 运行器以及 SWD 文件。SWD 文件的语法取决于同步对象定义的重要属性。本小节所讨论的 SWD 文件包括以下属性：

- 对象类型，AKA 类；
- 这些类拥有成员变量；
- 这些变量拥有同步行为。

现在你可以以一个简化的 BNF 形式来定义 SWD 语法的主要部分，如程序清单 6.1.1 所示（注意：省略了符号和关键字）。

程序清单 6.1.1 一个 SWD 文件的伪语法

```
compilation_unit := (first_id,class*)
class := (name,member*)
member := (behavior,type,name)
behavior := (behavior_selection,additional_attribute)
```

语法定义 compilation_unit 是分析的入口。

程序清单 6.1.2 是一个遵循程序清单 6.1.1 语法的一个 SWD 文件例子。关键字 conditional、periodic 等等会在稍后解释。

程序清单 6.1.2 一个 SWD 文件例子

```
world MedivalWorld
{
    synch_class Knight
    {
        conditional float Life;
      periodic(interval=0.2,duration=1) int MotionState;
      periodic(interval=0.2,duration=1) float rotationY;
      dead_reckon Vector3 Position,Velocity;
      conditional int Type;
      static ItemList Inventory;
   }
   synch_class Mountain
   {
      conditional int Type;
      // No mountain moves, of course.
      conditional Vector3 Position;
   }
}
```

由 SWD 编译器生成的代码负责以下任务：

- 管理同步对象并收集对象上的改动（创建、销毁或者成员变量改动）；
- 将这些改动转化为消息，并将消息发送给网络层；
- 从网络层接收到消息并处理。

在理想状况下，应该由 SWD 编译器生成的代码处理所有世界同步工作。但是，在实际编程中，这样做是很没有效率的，尤其当只需要对 HLA 源代码做很小的改动时。所以，让我们将 HLA 基础大部分抽出为一个通用库。

现在你可能已经可以想象如何让 HLA 系统整合进入游戏程序架构中了，如图 6.1.3 所示。

编译 SWD 文件推荐的方式是将其放入自定义构建配置中，详见[HyunJik04]。

同步对象

让我们将同步对象简称为 SychEntity，以避免与“对象”这个名字混淆。一个 SynchEntity 就是 SWD 文件中定义的一个类。

实际上，一个 SynchEntity 就是一个普通的类，但是它有更多的属性和行为，这在接下来将进行解释。

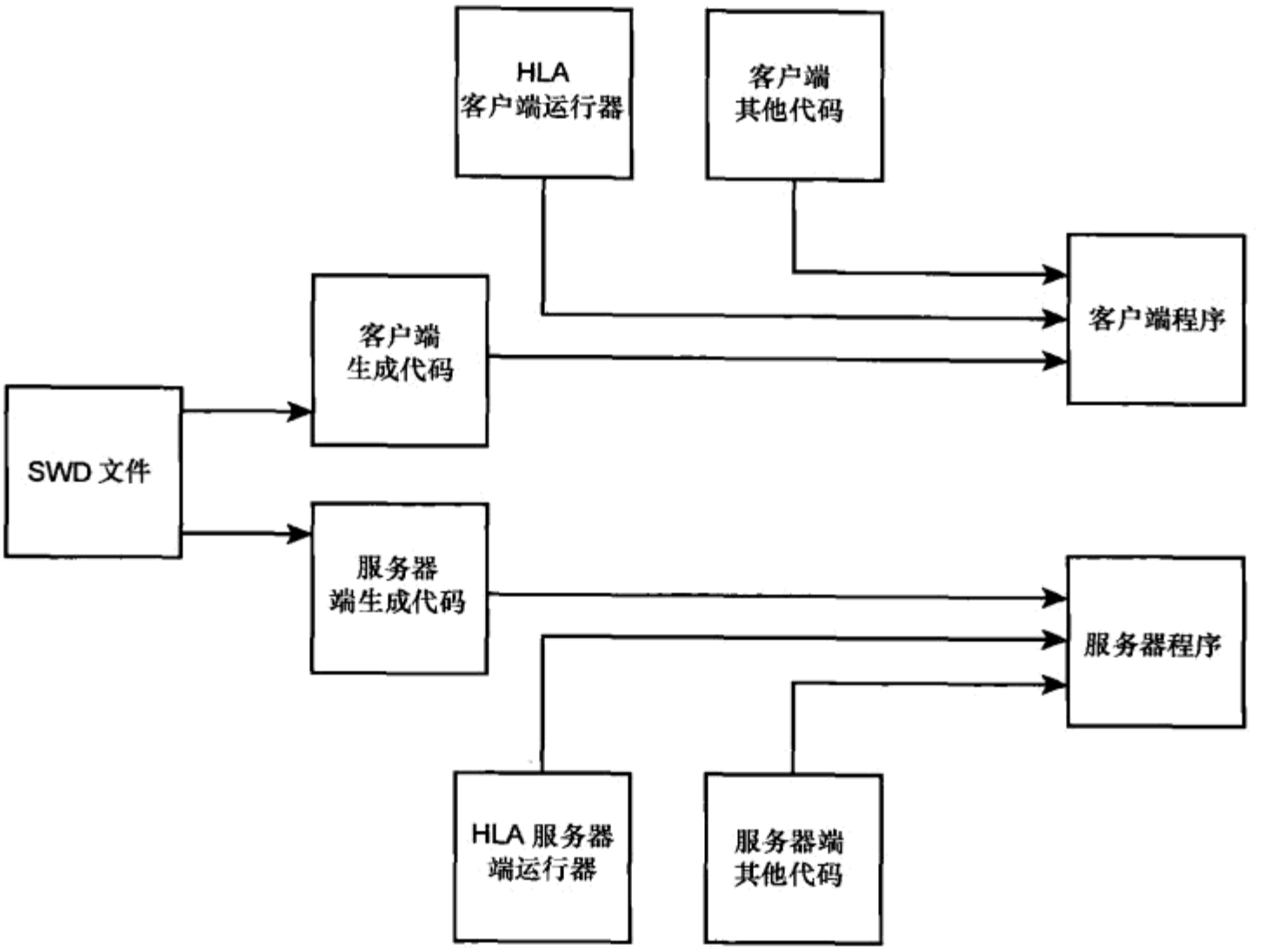


图 6.1.3　程序架构内的 HLA 活动

一个 SynchEntity 是 original（原有的）还是 replica（复制的），取决于那台主机拥有它的所有权（和修改它的完全许可）。拥有所有权的主机是这个 SynchEntity 的 subject（领主），所以 SynchEntity 会有一个 subject 属性。

在多个网络主机中，被标识的每个对象都必须是唯一的，所以每个 SynchEntity 实例有一个唯一的标识值，由服务器配置。

一个 SynchEntity 的每个成员变量实际上都是一个 property（属性）成员，这个 property 成员包括一对 set/get 函数以及将这两个函数绑定到一个实际成员变量的别名定义。许多当代的编译器都支持属性特性，比如 Visual C++中的__declspec（property）关键字。如果你的编译器不支持属性特性，你也可以通过使用一个类型转换操作符和一个赋值操作符来实现这个特性。程序清单 6.1.3 和程序清单 6.1.4 展示了这两种情况。

程序清单 6.1.3　使用__property 关键字

```
class MyClass
{
public:
__declspec(property(get=getX,put=setX)) int X;
void setX(int);
int getX();
};
```

程序清单 6.1.4 使用一个类型转换操作符和一个赋值操作符

```
class XType
{
public:
    XType();
    XType(int value); // takes a value into self
    XType& operator=(int value); // takes a value into self
    operator int(); // outputs the internal value
};
class MyClass
{
public:
    XType X;
};
```

每个成员变量的同步行为都在 SWD 文件中定义，然后由 SWD 编译器通过定义的这些行为来生成恰当的代码。一些代码可能会监视变量的改动。你可以通过提交类似程序清单 6.1.5 的代码，以获得更高的效率，因为当没有任何改动时，它可以帮助程序快速跳过比较。

程序清单 6.1.5 当赋值时标记一个变量为改动过的

```
void SetXXX(int newVal)
{
    m_maybeChanged=true;
    m_value=newVal;
}
```

一个 SynchEntity 成员变量的同步行为是以下 4 种之一：static（静态的）、conditional（有条件的）、periodic（周期的）或者 dead reckoning（需要推算的）。

static 行为表示这个变量永远不需要同步。如果没有 static 行为，那你应该定义一个类继承 SynchEntity，以添加不需要同步的成员变量。

conditional 行为表示这个值改动时做同步，这是最常用的行为。但是，如果这个值被频繁改动，可能会造成网络堵塞。

periodic 行为解决了 conditional 行为可能造成的问题，它表示这个值会以特定的间隔发送。这个行为需要发送间隔值和持续时长。如果一个 periodic 行为的成员变量改动了，它将以发送间隔值的间隔发送，一直到发送持续时长耗尽。举例来说，假设你为一个 Periodic 行为变量设置了发送间隔值为 0.2s，持续时长为 1s，那么这个值将每隔 0.2s 向远端主机发送，一共发送 5 次。periodic 经常与不可靠消息协议一起使用，比如 UDP。

程序清单 6.1.6 是一个 SWD 文件中使用 conditional 行为成员变量的例子，而程序清单 6.1.7 展示了它编译后的代码。

程序清单 6.1.6 SWD 文件使用 conditional 行为成员变量

```
synch_class Knight
{
    conditional int life;
    <...and more...>
}
```

程序清单 6.1.7　为 conditional 行为成员变量生成的代码

```
class Knight
{
private:
    int m_private_life;
    bool m_private_life_changed;
    inline void set_life(int value)
    {
        if(value!=m_private_life)
        {
            // A variable whose *_changed
            // is true will be broadcasted soon.
            m_private_life_changed=true;
            m_private_life=value;
        }
    }
    inline int get_life(int value)
   {
       return m_private_life;
   }
public:
    __declspec(property(get=get_life,put=set_life)) int life;
    <...and more...>
};
```

dead reckoning 行为允许你隐藏由于网络延时造成的振荡值。一个简单的航位推算模型需要以下 3 个变量：发送者的实际值、接收端的预测者和一个中间插值。所以 SWD 编译器应该为每个 dead reckoning 行为变量生成这 3 个值。

用于表示该值是否改变过的标记（程序清单 6.1.7 中的 m_private_life_changed）之后用来收集游戏世界中的改动信息。一个简单的模型是遍历每一个 SynchEntity，并通过查验该标记来收集改动过的 SynchEntity。因为 HLA 运行器本身并不知道这个标记是什么，所以遍历程序也应该由 SWD 编译器生成。程序清单 6.1.8 展示了一个处理程序清单 6.1.6 中变量的例子。

程序清单 6.1.8　为标记改动并收集以输出消息对象生成的代码

```
class MedivalWorld_Runtime
{
public:
    void GatherTheChangeToMessage(SynchEntity* entity,
        CMessage &outputMessage)
    {
        // the identifier SynchEntity_Knight is
        // generated enumeration value from the SWD compiler.
        if(entity->GetType()==SynchEntity_Knight)
        {
            Knight* typedEntity=(Knight*)entity;
            if(typedEntity->m_private_life_changed)
            {
```

```
                outputMessage.Write(typedEntity->m_private_life);
                typedEntity->m_private_life_changed=false;
            }
            <...and more...>
        }
    }
};
```

还有一个需要研究的部分就是从其他 HLA 运行器接收消息，并应用本地游戏世界的过程，这项任务被称为 deserialization（反序列化），在[HyunJik04]中提到。

HLA 运行器间通信

在世界同步中，主要的同步情况分为 SynchEntity 创建、销毁、出现、消失和值改动。每一种情况对应一个消息序列。

激活所有的 SynchEntity 都是在服务器确定需要创建（也就是，先必须在服务器端创建该对象）之后，它的创建事件才广播到客户端。然后接收到的客户端就会创建这个新的 SynchEntity 的一个副本。创建这个 SynchEntity 需要的参数包括它的 ID 和它的初始成员变量值。这些值被序列化到一个消息中，然后发送到需要的客户端。

有些 SynchEntity 在表现上不重要但特别影响效率（比如机枪的子弹），甚至可以直接在客户端创建在服务器许可之前。在这种情况下，客户端先创建这个 SynchEntity，然后再通知服务器，剩下的步骤就与之前所述一样。在客户端创建的 SynchEntity 的 ID 在一个由服务器在客户端加入游戏时就预先配置好的范围内。[Yongha06] 展示了更多细节。

SynchEntity 的销毁与创建类似，除了该类型消息外，只包含了要销毁 SynchEntity 的 ID。当一个 SynchEntity 在服务器被销毁，服务器将这个事件发给能看见该对象的客户端，而客户端也将该对象在本地的副本销毁。需要一个额外的消息序列是用于客户端销毁不重要的 SynchEntity，然后通知服务器。

SynchEntity 所有的改动都被收集起来并发送到拥有副本的客户端。消息包含了 SynchEntity 的 ID 以及一个附带变量 ID 的改动值列表。然后，每一个收到消息的客户端将这些改动应用到本地的副本。

考虑另一种情况：先由客户端决定修改某个 SynchEntity 并通知服务器，但是只有当这个 SynchEntity 无关紧要以致客户端拥有修改的许可或者这个 SynchEntity 的领主（subject）是该客户端。

每一个 SynchEntity 的可见性时时刻刻都可能改变，因为它的位置或者观察者的位置都在改变。如果一个 SynchEntity 进入了一个视口，在服务器向拥有该视口的客户端发送包含 SynchEntity 的 ID 以及它的序列化值的出现消息之后，客户端就会创建这个 SynchEntity 的副本。相反地，当客户端收到包含 SynchEntity ID 的消失消息时，客户端将删除相应的副本。

6.1.4 在 HLA 运行器中的视口

HLA 运行器的视口维持一个当前状态（位置等）以及一个网络主机标识符，用于发送或接收同步消息。通常一个视口会包含一个摄像机位置（或者多个，取决于玩家使用什么雷达）

以及一个主机标识符值。SynchEntity 和视口的基类 SynchViewport 都是抽象类。

一个对象-视口可见性检测的简单实现就是调用一个使用两个参数的函数，这两个参数是：一个 SynchEntity 和一个 SynchViewport。这个函数一般会被调用 $N \times M$ 次，其中 N 为所有 SynchEntity 实例个数，而 M 是所有 SynchViewport 实例个数。你可能希望实现这个函数来满足游戏的需要。举例来说，你的方法可能是基于地理范围、每个场景图节点的父子关系或者 BSP/PVS 的入口分块。这个函数的原型如程序清单 6.1.9 所示。

程序清单 6.1.9　一个对象–视口可见性检测函数

```
bool IsOneEntityVisibleToOneViewport(SynchViewport *viewport,
                                     SynchEntity* SynchEntity);
```

这里讨论的客户端/服务器布局只允许服务器使用这个功能，所以这个函数只存在于服务器端。

HLA 时间处理器 3

在世界状态改变时，你可能需要处理一些事情。有些例子是一个 SynchEntity 的出现和消失事件。比如，要加载人物的 JIT（just-in-time）资源文件，这些事件就很有用。

因为你是使用自己的 HLA 系统，你可以毫无限制地添加这些事件处理接口。你只需将这些时间处理器原型和调用代码插入 HLA 编译器或者 HLA 运行器的代码中即可。

创建 HLA 运行器

这个 HLA 运行器适合我们之前研究的结构。记住这个 HLA 运行器设计，如图 6.1.4 所示。

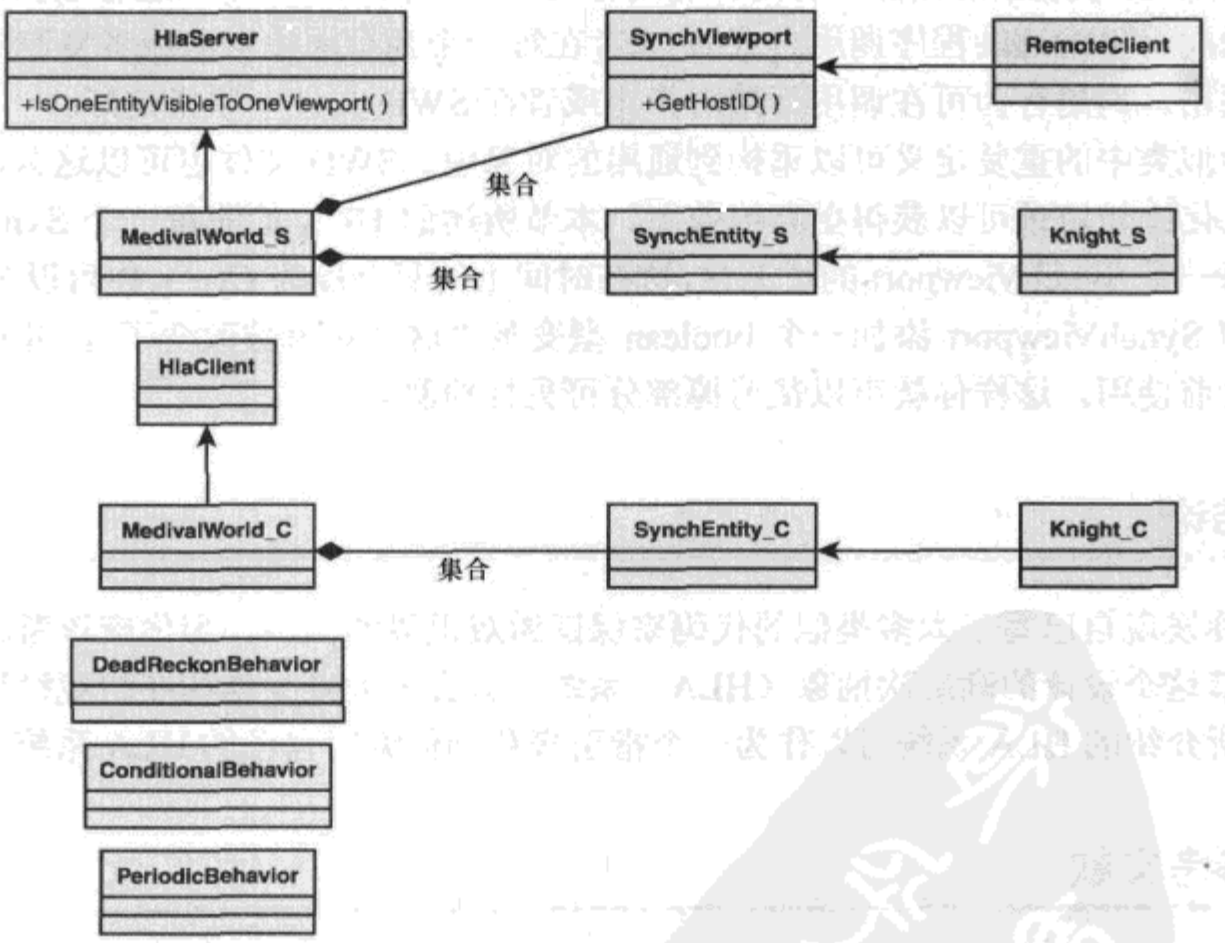


图 6.1.4　HLA 系统主要类的 UML 类图

HlaServer 具有以下特性。

- HLaServer 包含所有 SynchEntity_S 和 SynchViewport 派生对象的实例以及一个对象-视口可见性检测器（注意后缀_c 和_s 代表客户端和服务器端）。
- HlaServer 监测每一个 SynchEntity_S 和 SynchViewport 实例的状态。如果发现改动，HlaServer 将这些改动序列化为一些消息，并发送到远端主机。
- HlaServer 与一个网络引擎交互，以发送和接收与世界同步相关的消息。

HlaClient 维护从服务器复制而来的 SynchEntity_C 实例。像服务器一样，它也必须与网络引擎交互并保证所有 SynchEntity_C 的状态与服务器同步。

Knight_S 和 Knight_C 是 SWD 文件中一个类 Knight 的生成类。

类 Knight_S 和 Knight_C 的每个成员的类型是 DeadReckonBehavior、ConditionalBehavior、和 PriodicBehavior 类其中之一。这些类帮助 HlaServer 和 HlaClient 来决定是否这些变量需要广播。如果使用 ConditionalBehavior 类，程序清单 6.1.7 中的代码可以更加简明。

6.1.5 进一步讨论

为了阅读和讨论方便，本节所述的 HLA 系统实现只是一个简单的网络模型。用户可以基于本节提供的 HLA 系统，拓展以下的特性。

- 除了 conditional、periodic、static 和 dead reckoning 行为外，还有更多的同步行为类型，比如，基于时间戳的同步。
- 我们之前讨论的 SynchEntity 都没有成员函数。可以通过发送事件消息给远端主机来为 HLA 系统添加函数。有两种调用行为——只在原始对象上运行成员函数（以一种面向对象的远端程序调用方式），或者在每一个拥有原始对象或者复制对象的主机上调用。调用行为可在调用开始时指定或者在 SWD 文件中预指定好。
- 相似类中的重复定义可以重构到通用的对象中。SWD 文件也可以这么做。
- 优化比较瓶颈可以获得更高的效率。本节所述的 HLA 检测每一个 SynchEntity 对于每一个 SynchViewport 的可见性，这在时间上的复杂度为 $O(n^2)$。你可以为 SynchEntity 和 SynchViewport 添加一个 boolean 型变量“这个对象被改变了”，并在真正的比较之前使用。这样你就可以裁剪掉部分可见性检测。

6.1.6 结论

如果你发现自己写了太多类似的代码来保证游戏世界的同步，那你应该考虑一下实现一个基于本节这个设计的高层次抽象（HLA）系统，这会大大减少你花在游戏世界同步上的精力。本节所介绍的 HLA 系统可以作为一个指引来帮助你编写自己的 HLA 系统。

6.1.7 参考文献

[Aronson97] Aronson. “Dead Reckoning: Latency Hiding for Networked Games,”

available online at http://www.gamasutra.com/features/19970919/aronson_01.htm.

[HyunJik04] Bae, Hyun-jik. "Fast and Efficient Implementation of a Remote Procedure Call System," *Game Programming Gems 5,* edited by Kim Pallister, Charles River Media, 2005, pp. 627-642.

[Wikipedia07] "Meta-Programming," available online at http://en.wikipedia.org/wiki/Meta_programming.

[Yongha06] Kim, Yongha. "Generating Globally Unique Identifiers for Game Objects," *Game Programming Gems 6,* edited by Michael Dickheiser, Charles River Media, 2006, pp. 623-628.

6.2 网络游戏的身份验证

Jon Watte
gpg-7@mindcontrol.org

游戏中，不可信赖的客户端连接到一个或多个可信赖的服务器上，这种情况下的鉴定是特殊而有趣的。本节将展示一些设计一个在线游戏鉴定系统时需要考虑的方案选择，并推荐一个特殊的设计集合，包含一些内聚的设计选择。

6.2.1 介 绍

鉴定是一个确定某个人是他所表明的身份的过程，结果是一个给定主机集合上的确定的交互过程。对于计算机游戏，鉴定有以下两种情况。

- 登录游戏——确定客户端提供的凭信（一个用户名和密码）与合法玩家数据库中的信息匹配。
- 游戏会话——确定一个网络包是由它所声明的登录玩家发送的。

注意，鉴定只包含了确定谁发送了一个特定的包的功能，与加密并没有太大的关系。而加密是对非计划内的接收者隐藏消息的功能。一个例外是，有一类加密系统（公共/私有密钥系统）同时提供了这两个功能。不幸的是，这类加密系统的计算通常都是相当耗时的，因此对于一些实时的在线服务，比如游戏，并不适用。

6.2.2 游戏登录安全

为保障游戏登录安全，你需要考虑以下一些问题。

- 不安全的密码——玩家的密码可能是一个常用的词语（比如“secret”）、玩家的名字或者甚至是空格。你的密码设置机制应该要检测到弱密码，并要求玩家输入更安全的密码。
- 不安全的密码存储——你的服务器安全吗？有人可能会闯入服务器。如果他们能够以明文读到玩家的密码，那问题就大了。你的系统管理员可以信赖吗？如果你需要解雇他们或者其中一个怎么办？
- 被嗅取密码——如果你不使用安全套接协议层或者一些类似的高度加密协议，某些人就有可能使用一个包嗅取器来获得以明文发送的密码，然后模仿该用户。虽然这类攻击比较少见，但确实发生过，作为部分数据中心妥协的一部分。

- 键盘嗅取器——一些恶意软件和木马程序会将自身安装到用户的机器上，并记录用户所有的键盘输入。如果熟悉游戏的问询方式，就可以很快从记录中推出登录名和密码。
- 无知用户——在许多在线游戏中，总有用户想在与其他玩家的交互中直接得到用户名和密码。一旦获得这些敏感信息，这个账户里所有有价值的虚拟物品都会被掠夺，而密码会被修改为一些任意值，原用户将无法使用。
- 重复登录——系统不应该允许同一个用户在同一时间多次登录。否则，一个玩家付费之后会将账号与他的朋友们分享。虽然这只占流失收费的一小部分，但更大的问题是，当你因为某些人没有遵守规则而需要禁掉这个账号时，那种情况绝对是客服的梦魇。

本节的主要目的是帮助你校正客户端/服务器设计中的验证。

一项你必须做出的决定是你是否允许从数据库中恢复密码。你有以下一些选择。

- 可恢复密码——如果密码是可恢复的，你可以使用挑战哈希验证体系，这个将在之后详述。但是，当数据库中的密码是可恢复的，密码也更不安全，因为一个不可信赖的操作员或者系统入侵者都可能得到密码的列表。

不要将密码以明文存储在数据库中。起码将密码与你内置在代码中的一些关键字混在一起，以保证普通的检查员不会“偶然”看到密码。但密码可以修改始终是危险的，所以时刻警惕会改变你的数据的人为因素。

- 单向哈希密码——每当设置一个密码，就算一个该密码的单向哈希（比如一个 SHA256 校验和），并存储这个校验和。当玩家下次登录时，他们给你密码，而你计算出一个 SHA256 的校验和并与你存储的值比较。如果匹配，你就认为提供的密码正确。

这么做最主要的好处是保证了系统端的安全；读一个 SHA256 校验和不会让任何人知道实际的密码是什么。找到另一个生成同样校验和的密码在计算上是非常困难的。但是，当你要这么做的时候，你必须使用秘密交换验证（之后将详述），它比挑战哈希验证更易受攻击。

- 公共密钥基础——如果你拥有一个公共/私有密钥加密系统，比如 RSA 或者 SSL，你可以在游戏服务器上公开公共密钥，甚至可以将之写入游戏客户端运行文件中。然后用户的密码通过加密后再连接上传送，不用担心窃听。这项技术另外的好处是只有真正的服务器可以解密这个消息，所以客户端就可以合理地认为它在与真正的服务器对话，而不是个冒充者。这个方案的弱点是实现相当复杂（最好的选择可能是使用一个开源的加密系统库，比如 OpenSLL）。

挑战哈希验证

在挑战哈希验证中，服务器向客户端发布一些随机数，称为 challenge 或者 nonce。客户端计算出这些随机数和输入密码的一个哈希，并将这个值发送回服务器，然后服务器会计算出之前 challenge 值和存储密码（纯文本）的一个哈希，并与客户端提交的做比较。如果两个哈希匹配，说明密码正确。

这个系统有以下 3 个主要特性。

- 密码没有直接被传送——因此黑客通过嗅探常规的登录包并不能直接得到密码。
- challenge 值每次登录请求都是不一样的——因此，就算你探测到连接，你记住这个哈希值并在之后登录时提供同样的哈希值也不能蒙混过关，因为由服务器为一次特定登录随机生成的 challenge 值每次都是不一样的。

- 服务器端保持明文密码——如果服务器端是可修改的，这会是一个安全问题，但是这个明文密码是被交互的两端秘密共享的，因此可以用来加密所有该特定客户端发送和接收的数据。注意，要使用一个密钥，不可以被轻易破解的加密算法——XOR或者 ROT-13 是不合适的。

一个需要注意的常识是可以使用一个明文密码的哈希作为交互的密钥，但不要把同一个哈希用于验证，否则使用一个“不可嗅探”的共享密钥的优势就没了。

秘密交换验证

在秘密交互验证中，服务存储一个密码的哈希。客户端提交一个纯文本的密码，而服务器计算出这个纯文本密码的哈希，并与存储的值比较。如果匹配，说明密码正确。

这个系统具有以下一个优点和两个缺点。

- 服务器不存储明文的密码——如果有人闯入服务器并窃走密码文件，那也不要紧，因为你不可能通过哈希值（密码强度）就得到密码。在老的 UNIX 机器上，加密的强度并不是很高，所以你还是要保证你的/etc/shadow 文件安全，但是使用一个 256 位的 SHA 哈希，你的密码将是非常安全的。如果你不信任你的备份操作员，或者你的服务器被黑了，这将是最主要的好处。
- 密码在每次登录请求时都被传送——如果有人嗅探连接，那他就可以获得密码。因此你需要使用一些加密保证登录请求安全——但是使用什么密钥来保证安全并不确定。最安全的方法是使用一个 Diffie-Hellma 密钥交换。虽然正确实现它的代码相当晦涩，但它可以为两个终端提供一个安全加密过的通道，而不需要事先交互密钥。如果你想要阻止一个老练的攻击者，将自己插入网络中间，你必须引入一个额外的基于公共密钥的加密验证系统，这项工作的工作量是相当大的。
- 服务器有明文密码——因为客户端发送的是明文密码，服务器起码暂时能够读到明文的密码，并可以使用密码作为登录之后交互加密的密钥。不幸的是，这也意味着如果有人冒充服务器，或者能够读到你服务器进程的内存，那他们就能够得到纯文本的密码，尽管密码存储文件本身是安全的。

公共密钥基础

如果你拥有一个公共/私有密钥加密系统，比如 RSA 或者 SSL，你可以在游戏服务器上发布你的公共密钥，甚至将其写入游戏客户端执行文件中。然后用户的账号信息在连接上传送，不用担心在线上被窃听。另外一个选择是在用户设置账户时生成一个私有密钥，在服务器端存储匹配的公共密钥，并在本地使用用户的密码加密私有密钥（被称为密码短语，pass phrase）。

这样，一个系统具有以下特性。

- 服务器永远不知道密码短语——因此，恶意的雇员和服务器系统的入侵者都不可能轻易地通过包嗅探或者跳过登录来得到用户账号信息。一个坚定的攻击者可以通过反汇编服务器的二进制文件得到这些信息，但这时，你的整个游戏都被破解了，你可能要担心更大的问题了。
- 用户有很好的保障不会受到冒充服务器者的欺骗——只要服务器私有密钥不被破解，没有人可以假装服务器并提取用户信息。

- 用户账号信息不可迁移——如果你的用户要在多个地点登录游戏，他需要制作账号私有密钥的副本才可以通过登录验证。这是玩家一般难以接受的，而且极可能让客服支持头痛。

6.2.3　保障游戏时安全

玩家登录进来之后，并不意味着你的麻烦就结束了。你经常需要把一个玩家从一台服务器转到另外一台，或者允许玩家从服务器断开连接（可能是崩溃了），然后重新连接，并回到玩家离开的位置。显然你不可能直接相信客户端所声明的 ID，因为一个玩家突然冒充其他玩家的事情时有发生。相反，你必须使用以下 3 种技术之一：IP 地址身份验证、验证记号身份验证和密码身份验证。

IP 地址身份验证

在这种方法中，服务器查看接收包的源 IP 地址和端口号，并且内部维护一个玩家对应的地址/端口表。这是安全的，只要你确定玩家总是从同一个端口发送消息，并且互联网不会接收带有欺骗地址的包——或者，如果一个包是欺骗包，往返确认会在真正的客户端发生。

这种往返确认可以以特定怀疑命令的直接回包形式出现，或者间接以使用一个从随机初始位置开始的循环序列索引来实现。

但可悲的是，如果你使用 TCP 连接，或者你需要服务器间的握手连接关闭，客户端地址中的端口部分将不可能始终一样。TCP 为每台机器上的每个连接分配一个新的端口号，甚至当你切换目标机器时 UDP 也会有端口重分配的问题，如果目标机器是在一个不友好的 NAT 网关之后（虽然大多数家庭 NAT 路由器不会有这个限制）。

验证记号身份验证

当玩家登录进来，服务器决定一个连接有效的时长——比如，一个小时，然后服务器计算出一些数据段的哈希：客户端 ID、登录会话过期时间和一个只有服务器知道的秘密数字。然后服务器向客户端发送一个记号，这个记号包含客户端 ID、过期时间和这 3 部分数据的哈希。

当客户端发送数据时，它在数据之前加上这个记号。服务器从中取出 ID、时间和哈希部分，并使用它内部的秘密数重新计算哈希。如果两个哈希匹配，服务器就知道这个包确实是由之前验证过的玩家所发送的，或者这个 ID 和会话时长已经是服务器停用了的。

如果客户端崩溃然后重新连接，它可以从硬盘上读 cookie 并重新应用，只要这个会话还有效，就不需要新的验证。如果游戏会话超出一个小时，当前与玩家对话的服务器会通过重新生成一个新的记号将 cookie 每次延长半小时。这样，一个客户端崩溃然后在半小时内重新连接就可以继续游戏而不用重新登录。

密码身份验证

如果你使用一个服务器和客户端的共享秘密，比如一个纯文本的密码，你可以使用这个秘密作为一个密钥，或者更佳的方案是使用一个密码和一些随机数的哈希。注意这些随机数不要选择与初始连接验证中使用的相同。每个由客户端发送的包都包括明文的客户端 ID，接

着是由共享密钥加密过的包数据，然后是一个数据（未加密）的校验和。

当服务器收到一个包时，先在内部表中查询用户的密码，解密这个消息，并检查校验和。如果校验和不匹配，那么数据就不是使用正确的密码加密的，因此这个包也不是由正确的客户端所发送的。

经验告诉我们应该有一个序号作为加密数据的一部分，这样连续的相同包加密之后也是不一样的，而截获和重发一个包将不大可能真正被接受。

游戏会话的其他考虑

本节第二部分所提到的其他问题在这里也要说一下，虽然不可能像本节的主题一样详细说明细节。

- 不安全的密码——当玩家生成或改变密码时，你应该检验一下这个密码是否包含至少 6 个字符（最高 24 个）。此外，检验这个密码是否包含 3 个字符组中至少一个字符——字母、数字和非字母数字字符。
- 不安全的密码存储——为保护服务器秘密免于被恶意的内部操作员所窃取，采用最好的 IT 实践。不要授权任何人访问所有的服务器。使用一些硬编码在可执行文件中的关键字，以混杂格式存储纯文本密码数据。用一个独立于游戏主服务器的服务器存储一些特别敏感的信息，比如信用卡信息或者用户住址，使用一个额外的防火墙来隔断游戏和账单信息。
- 键盘嗅探器——如果你担心键盘嗅探器，那就让玩家使用一个屏幕上的键盘来输入密码。也要注意一些恶意软件可以从标准的文字编辑控件中读出所有信息，所以你可能需要使用一个自定义的 GUI 控件来读取密码。
- 无知用户——为玩家行为和安全创建一个广泛的规则集，并在用户注册时要求接受。但是注意将一些特别重要的信息浓缩之后为摘要，如“永远不要提供你的密码，即使有人称它是我们公司的人”。将这些摘要加入每一个加载页面，可能的话循环播放，以强调这些消息。
- 重复登录——当一个会话授权或者 cookie 产生，废弃之前所有的授权或 cookie。这意味着一个账号的第二次登录将踢掉之前的登录用户。但是，如果一个用户断开连接后重新登录，将不会受到影响，因为老的会话授权已经不再使用了。

6.2.4 结论

如果你阅读了本文，这是一个很好的信号——起码你重视安全并想做好！当交互两端都知道密钥时（比如当你使用秘密交互验证和密码身份验证时），可使用的一个好的加密算法是 Tiny 加密算法。这个算法易于实现，并且密码强度很高。真正注重安全细节的人推荐只使用标准化的协议，比如 AES，因为它们经历了更多的研究和发布，因此任意的缺陷都会被很快发现并公布，警示你到了更好加密系统的时候了。

SHA256 是一个常用的标准化哈希函数，到现在为止还没有发现旧 MD5 哈希算法中的那些缺陷。也有一些可用的替代算法，比如 Tiger。

在一个协作服务器群（比如 MMORPG 或者虚拟世界的服务器）之上，身份验证的实现

应该类似如下所示。

- 在设置时，所有在群中的服务器共享一个大随机数，被称为群密码。
- 客户端使用未加密的 TCP 或 UDP 来连接服务器登录。
- 服务期向客户端发布一个 challenge 值，包含一个 256 位的随机数（nonce）。
- 客户端计算出一个该值和密码的哈希，并发送给服务器。
- 服务器使用 challenge 值和存储的密码计算出哈希值，并与客户端发送的校验是否匹配，然后向客户端发布一个包含用户 ID、授权过期时间，以及这两者与群密码的哈希值的验证授权。
- 登录服务器也生成一个供该客户端在这个会话中使用的随机密钥，并发送给客户端。它也把这个密钥以及授权过期时间记录下来。这个密钥在发送给客户端之前使用一个由用户密码和已知 salt（比如字符串“abcd”）哈希生成的密钥加密。
- 服务器连接到游戏服务器群中任意一个。
- 客户端通过发送之前获得验证授权来开始与群中的新服务器连接。
- 新的服务器校验授权是否过期，以及哈希是否正确。使用授权中的用户 ID，新服务器从登录服务器上得到加密密钥。
- 这时新的服务器和客户端也协商出将来交互用到的序号。
- 一旦验证通过，新的服务器和客户端使用会话密钥加密过的数据，这里加密的数据包括数据规矩（作为校验和）和一个序号生成的哈希。这些包每一个都需要包含客户端 ID 和授权标志符（一个小整数）作为包头，而不是整个验证授权。
- 客户端当前连接到的服务器周期性地检查会话验证授权是否将要过期；如果是，它将联系登录服务器请求一个新的授权并发送给客户端。

这个体系将保护你的系统免于在开放的互联网上被嗅取密码，并应对某些人使用嗅取包做重放攻击。对于中间人攻击，被破解的会话是不安全的，但是中间人不可能得到身份验证信息以在之后重新验证通过。要保证连接中间没有人干预，你需要加入公共/私有密钥加密和验证。

同样，需要注意的是，没有相应的技术来保护客户端机器接收到的数据不被用户查看——用户控制运行客户端的机器，所以他总是可以查看到内存中的数据。这意味着你的游戏设计应该是能够防止作弊的，或者你必须能够提供使玩家不作弊的激励。验证和加密只是让你的秘密信息不被第三方得到，而不是参与进来的两方（客户端和服务器）。

6.2.5　参考文献

AES, the Advanced Encryption Standard algorithm. The successor to the Data Encryption Standard, and the current U.S. Federal Information Processing Standard encryption algorithm, available online at http://csrc.nist.gov/CryptoToolkit/aes/aesfact.html.

OpenSSL. A high-quality Secure Sockets Layer library, using an Apache-style Open Source license, available online at http://www.openssl.org/.

[Schneier95] Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source*

Code in C, 2nd edition, Wiley, 1995.

SHA, Secure Hash Algorithm. The successor to MD5, and the current U.S. Federal Information Processing Standard hash/digest function, available online at http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf.

Tiger. A fast hashing (digest) function, with no known patent encumbrance, available online at http://www.cs.technion.ac.il/~biham/Reports/Tiger/tiger/tiger.html and http://en.wikipedia.org/wiki/Tiger_(hash).

X-Tea and Corrected Block TEA. A fast, high-strength, simple-to-implement symmetric encryption algorithm, with no known patent encumbrance, available online at http://en.wikipedia.org/wiki/XXTEA.

6.3　使用智能包嗅探器来调试游戏网络

David L. Koenig, The Whole Experience, Inc.
yarnhammer@hotmail.com

通常，游戏中的大多数网络包对于包间的长延时是非常敏感的。当你想要实时调试网络代码时，这将是个问题。在这种情况下，标准的做法是使用一个包嗅探器收集游戏中的网络包，并在之后分析。包嗅探器可轻松访问包来源和包目的地，以及其他网络协议栈信息，它不能提供的信息是你的游戏协议特性。

6.3.1　智能包嗅探器概念

智能包嗅探器，或者更好的称呼是游戏消息嗅探器，它的概念是你不仅需要查看线上传送的原始二进制数据或者 TCP/IP 协议数据。还需要查看更多的以人类可理解的形式存在的细节信息。基本来说，这是一个了解你游戏内部协议特性的包嗅探器。

这个智能包嗅探器是在制作 PlayStation 2 上的游戏 Greg Hastings' Tournament Paintball Max'd(GHTP)时开发的。这个游戏在 2006 年下半年发布。一个已经不在该公司工作的工程师编写了基线游戏消息系统和网络代码。由于网络代码和消息系统没有提供相应的文档，在这之后开展工作是相当困难的。第一件事就是要阅读代码，这样可以让你了解整个底层系统的架构，了解到底什么游戏消息在线上发送以及什么时候开始难以把握。这也就是一个智能包嗅探器能派上用场的地方。

6.3.2　一个例子

在 GHTP 项目中，这个嗅探器提示我们：我们的服务器向客户端发送了大量的 100- and sub-100-byte 玩家位置包。有 42 个字节是以太网络帧、IP 以及 UDP 头信息，我们的包头开销大概占 40%。我们可以通过合并数据包来改善效率并大大减少总花销。使用一个标准的包嗅探器，我们也许可以通过检查一系列包的二进制数据得到相同的结论，但使用我们的嗅探器，你只需一眼就可以精确地了解到发送的网络消息，以及它们所消耗的带宽。毋庸置疑，这个工具使我们节省了大量的时间。

6.3.3 传统调试技术的缺陷

当你调试网络代码时，你可能不想完全放弃标准的调试函数。但是，你应该了解这些技术可能会引入的缺陷。你想要避免的是引起在终端用户中并不存在的 BUG。这通常会因改变了代码路径或者改变了代码时钟而引起。以下是一些可引发这两类问题的例子。

断点

这通常是开发者测试代码正确性所设的第一条防线，它可以让你检查一个操作是否按照预定的路径进行，可以让你检查重要变量值以及寄存器值。这些信息对于开发者都是至关重要的。当它使用于网络代码所引入的问题是你只在模拟的一端停止了，在另一端，游戏连线的另一个主机还在运行。在某个时刻，第二台主机会认为连接已经丢失并停止。这对你的工作有没有影响，就要看你在调试什么类型的问题。如果你只是要确定某个部分的代码是否执行到了，加断点是一个快递有效的方法。但是，如果你想要知道有多少个心跳包发送到服务器或者哪些消息发送最为频繁，那断点就会失效。

跟踪点

跟踪点的概念是在 Visual Studio 8.0（也被称为 Visual Studio .Net 2005）中引入的，这项技术允许你在代码中放置跟踪点，当该点命中时，游戏不需要停止。你可以做各类事情，如可以让进程停止，也可以运行脚本，在调试窗口打印信息，或者打印出调用栈。虽然在调试选项上相当进步了，但它还是会改变你的代码时钟。

输出调试信息

通常通过调用 printf、OutputDebugString 或者同类函数来实现。这在获取一些诸如每秒使用带宽或者丢包百分比等信息时非常有用。但问题是：每当你为一个操作添加额外的代码时，就改变了代码的时钟。额外的代码导致额外的处理器指令，这会导致原始代码中的 BUG 消失，或者可能引入其他的缺陷。当使用输出调试信息方法来调试时间敏感的代码时，记住一定要小心。

6.3.4 实现

一个智能包嗅探器的基本实现是很简单的，接下来的部分将概要介绍创建这个嗅探器的基本步骤。

暴露网络结构

一个智能包嗅探器的基本原理就要求你暴露出你的游戏内部协议信息，这可以通过简单地让游戏和嗅探器包含同样的头文件来实现。一个建议是你可以设置一个共享文件夹，用于放置所有游戏和嗅探器共用的代码，以保证这两个程序所使用的协议版本是同步的。

包截获

你需要一种技术从网络上截获数据。有几种选择。你可以自己编写一些代码来截获包。

另一种替代方法（也是推荐的方案）是使用一个第三方库，比如 Pcap[Pcap]。这是我们用来实现嗅探器的方法。使用 Pcap 的好处是它的代码已经被使用和测试了好多年，而且是开源的，且易于使用。

包解码


ON THE CD

一旦你截获了一组包，接下来需要做的是将之翻译为游戏消息，这时候就需要分析代码了。基本上这部分就是你的协议解码器，当然，这也是智能嗅探器与普通包嗅探程序的主要不同之处。在 CD-ROM 上的例子中，我们使用了一个插件方法。我们这个简单例子的协议是使用一个动态加载的 dll 来解码的，这种方法允许你支持各种可能的协议，使你的嗅探器核心代码不需要考虑协议特性。

显示

有很多方法可用来显示你的数据。一些工具如 tcpdump[Tcpd]使用一个命令行界面。而 GHTP 嗅探器使用了一个 MFC 用户界面[Mfc]。列表控件是一种基本控件，但可以自动伸展，以容纳我们的数据。它让你可以设置一个简单的多行多列视图。当然，还有很多的方法来创建用户交互界面。你可以自行选择最适合你的方法，见图 6.3.1。

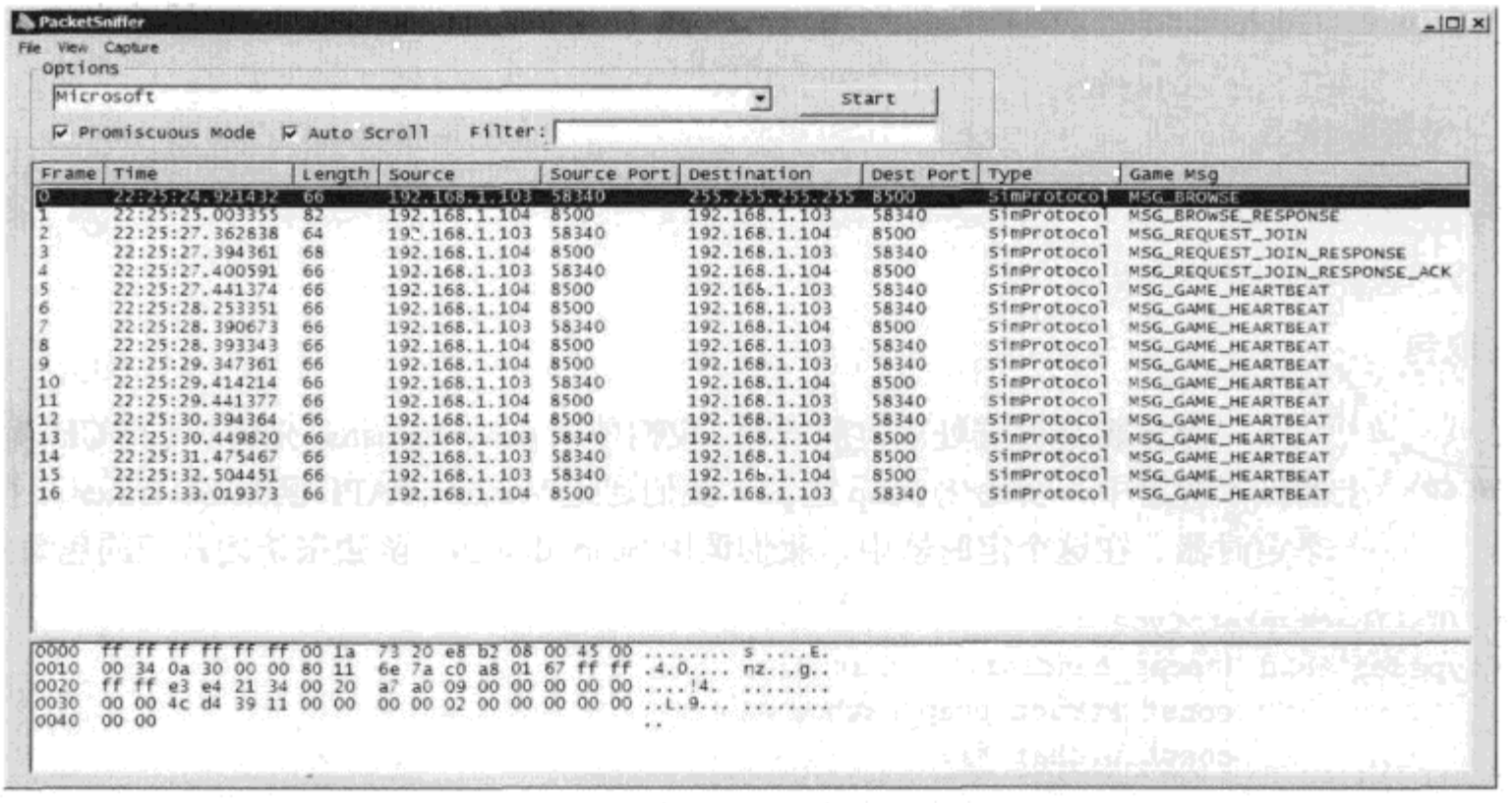


图 6.3.1　智能包嗅探器的用户界面视图

6.3.5　使用 WinPcap 库


ON THE CD

开源的 Wireshark 包嗅探器以及其他许多网络工具都使用了 Pcap 库。WinPcap 是这个库的 Windows 版本，它可以帮助开发者轻松地截获网络上发送的包。开发者开始时只需要使用这个库的一个子集即可。查阅 CD-ROM 上可运行的例子程序，它涵盖了所有本节所提到的内容。为节省篇幅以及你的精力，这里只列出函数原型。想获得更多这些函数的深入信息，可阅读 Pcap 文档。

枚举设备

要开始进行包截获，需要定义想要监听哪一个本地网络设备。首先，需要知道在你的系统中哪些设备是可用的。Pcap 提供了以下两个函数来获得设备信息以及清理查询所使用的内存。

```
int pcap_findalldevs(pcap_if_t **, char *);
void pcap_freealldevs(pcap_if_t *);
```

初始化 Pcap

在开始截获包之前，必须初始化 Pcap，这将设置你要使用哪一个网络设备来截获包。你也可以设置截获操作过滤器，比如，你也许想过滤掉 UDP 包。我们的包嗅探器认为需要这么做。

```
//获取 Pcap 设备的句柄
pcap_t*pcap_open_live(const char *, int, int, int, char *);
//检测设备的媒介（例如以太网）
int pcap_datalink(pcap_t *);
//编译从文档中获得的包捕获过滤器
// Pcap 的文档描述这部分关于过滤器的说明
int pcap_compile(pcap_t *,
      struct bpf_program *,
      char *,
      int, bpf_u_int32);
//设置包过滤器
int pcap_setfilter(pcap_t *, struct bpf_program *);
//设置 Pcap 设备
void pcap_close(pcap_t *);
```

取包

下一个步骤是设置处理包的管线，我们使用 pcap_dispatch 来设置。在 CD-ROM 上的例子程序中，初始化 Pcap 之后，我们通过 Windows API 函数 SetTimer 来设置一个定时器。在这个定时器中，我们调用 pcap_dispatch 函数来访问截获的包数据。

```
//Callback prototype
typedef void (*pcap_handler)(u_char *,
             const struct pcap_pkthdr *,
             const u_char *);
             int pcap_dispatch(pcap_t *, int, pcap_handler, u_char *);
```

在这个包处理器回调函数中，你就可以访问包中的数据了，这时你应该使用你的协议解码器分析原始的网络数据，并返回一些有用的信息。

6.3.6 降低安全风险

一个能解码网络数据并显示网络代码的内部信息的工具当然对工程师调试协议很有帮助。但与此同时，对于那些想破解你协议的人，这也是一个再好不过的工具，所以这个工具也很危险。因此你需要考虑下如何限制这个工具的用途。

限制使用

确保只有需要直接使用该工具的人可以得到它。作为一个网络工程师，你最不想看的事就是：在游戏发布的第一天，你的协议就被破解了，而且是通过你开发的调试工具。

加密

如果你的协议包含一些级别的加密，你可能有一条内在的未使用防线。在调试时，允许禁用加密将是个好主意。你有一些方法来实现这个功能，可以连接到一个网络库的一个不加密版本，也可以创建一个单独的构建目标（fix later），在这个目标中设置一个预编译宏定义来禁用加密。

也许你的包嗅探器只能处理未加密的网络包，这会降低这个工具的风险。即使它被暴露给公众，用户也不能直接使用它来分析你的网络数据。当然，这个解决方案不是完美的。有些人可以分析你的嗅探器的汇编代码，并以反向工程学来推导你的内部网络代码结构。有这个工具肯定会使黑客更易于找到你二进制文件中相应的结构。

6.3.7　一个替代方案

如果你不想从头开始写这样一个工具，还有其他方案。Wireshark 包嗅探器有一个插件架构允许你定义自己的协议特性。比如，你可以为你的协议写一个包剖析器，这需要做的只是将你的协议内部信息暴露给 Wireshark。这为一个已经很强大的工具又增添了可观的可拓展性。Wireshark 已经打包了一些第三方的包剖析器。比如，在主配置中就包含了一个 Quake 3 协议剖析器。作为一个网络程序员，你应该拥有一个功能齐全的包嗅探器。

6.3.8　例子程序

CD-ROM 上的例子包含了一个简单的命令行客户端和服务器模拟程序集，以及一个简单的智能包嗅探器。所有程序都有全部的源代码。工程文件需要 Visual Studio 2005 和 WinPcap 开发库。

6.3.9　结论

有些时候，正确的事情会使你事半功倍。我们只花费了半天来编写嗅探器，而它给了我们很大的帮助，大大减少了开发人员用于调试协议的时间。尽可能在工程的早期就考虑为你的游戏收集哪些信息，以更好地调试，尽可能早得放置“钩子”，你会看到成效的。

6.3.10　参考文献

[Mfc] Microsoft Foundation Classes documentation. Available online at http://msdn2.microsoft.com/en-us/library/d06h2x6e(VS.80).aspx.

[Pcap] WinPcap Website. Available online at http://www.winpcap.org.

[Tcpd] tcpdump Website. Avaliable online at http://www.tcpdump.org.

[Wireshark] Wireshark Website. Available online at http://www.wireshark.org.

第 7 章

脚本和数据驱动系统

介　绍

Scott Jacobs
Tom Forsyth

效率最大化是我们在游戏开发中不停追求的目标。从传统意义上来说，追求效率主要关注游戏运行时的速度，因此编译型语言目前还是游戏编程的基石。但是，开发者们常常发觉，他们也需要在其他方面找到提升效率的方法。于是，首先出现在他们脑海里的是实现功能的速度和迭代的频率。脚本和数据驱动的效率能够在这里最好地被体现出来。本章将介绍 5 个脚本和数据驱动的精粹内容，每个精粹内容在光盘里都提供相关的代码。

首先，Julien Hamaide 为我们提供了一种方法，可以自动绑定 C++类到流行的游戏脚本语言 Lua 上。他的实现方法着重于效率，有效的内存利用以及线程安全。下一个讲解 C++类接口的是 Joris Mans，他的精粹讲解了如何将 C++类的实例序列化到 PostgreSQL 之类的关系型数据库中，以及如何从数据库中取出。用这种方法保存类里的数据，为数据操作、共享、计算方阵和平衡开拓了一条崭新的途径。

Martin Linklater 分享了一种他称之为数据端口（dataports）的设计理念，这种设计理念提供了一个常规的通信 API，用来操作其他代码里的数据。这种通用的接口可以在各个模块之间减少耦合几率，并允许更具柔性的界面。

“通过数据驱动的方法来管理 shader”一文由 Curtiss Murphy 提供。这个构架由 XML 文件来配置，在开始实现时做一些额外的工作以后，它能够令人信服地对 shader 迭代和开发进行管理，并且只需要很少甚至不用图形程序员的参与。

最后，Zou Guangxian 提出了一种想法，通过直接操作 Python 的抽象语法树（AST）来创建字符串表。该精粹介绍了如何在 Python 里继承，并且通过继承来挂钩在 Python 的分析和编译阶段，目的就是解析出有用的代码结构，或者动态影响和自定义编译结果。这里的内容一定会让你产生兴趣，并激发灵感。

7.1 Lua 自动绑定系统

Julien Hamaide
julien.hamaide@gmail.com

因为游戏内容增长的速度远远超过了以前，所以程序员已经不能够完全控制所有代码的行为了。他们需要其他游戏开发者的帮助。脚本语言在游戏里已经被用了几十年了，但是现在的游戏机更能够抓住它们的优点来大大提高玩家的体验。本精粹着重说明 Lua 语言的绑定实现。这门技术能够让程序员将他们的 C++类暴露给 Lua，但不需要了解这个系统。这里介绍的工具不仅能用在 C++语言里，对其他语言也有效。这种设计的核心思想是由易用性、效率、内存占用和多线程的设计目标来驱动。

7.1.1 介绍

在本精粹里介绍的绑定允许在 Lua 脚本里建立、访问和使用 C++对象。举例来说，在一个 singleton 的类 WORLD 里保存了一个 ENTITY 类型的实例列表，下面的脚本就能够用来设置玩家的生命值。

```
local entity = WORLD:GetEntity( "player" )
entity:SetHealth( 50 )
```

在这个例子里用到的绑定是由如下的声明定义的。

```
// in .h and class definition
SCRIPTABLE_DefineClass( WORLD )

// in .cpp
SCRIPTABLE_Class( WORLD )
{
   SCRIPTABLE_ResultMethod1( GetEntity, ENTITY, std::string )
}
SCRIPTABLE_End()
```

绑定一个类就这么简单。不需要做其他任何步骤，允许程序员把 C++类和自己的函数暴露给 Lua，就这么简单。

7.1.2 特 性

这个系统的设计目标如下：

- 低内存消耗；
- 高效率绑定；

- 支持 C++的继承；
- 使用方便；
- 在脚本与脚本之间保证线程使用的安全。

7.1.3　函数的绑定

Lua 需要一个特别的接口来绑定函数。被绑定的函数必须是下面代码里定义的类型。Lua 的绑定是基于堆栈的，lua_State 包含了所有要传到那个函数里去的参数。这些参数必须通过使用堆栈索引号来由 lua_to*来收集。在这个例子里，这个函数接收的第一个参数是字符串，第二个参数是数字，返回值也是数字。关于 C 函数的绑定，你可以在 Lua 手册[Ierusalimschy06]里找到更多的相关信息。C 函数的绑定是 Lua 绑定到 C/C++的唯一一个方法，也是这个系统的基础。

```
int binding_method( lua_State * state )
{
      const char * some_string;
      double some_number, another_number;
      some_string = lua_tostring( state, 1 );
      some_number = lua_tonumber( state, 2 );
      // 在这里可以设置返回值，或者做点你需要做的事情
      // 另外一个数字
      lua_pushnumber( state, another_number );
      return 1; //假如说我们返回 1
}
```

7.1.4　在 Lua 里的面向对象

Lua 是使用广泛的一种编程语言，功能也很强大。本精粹介绍了如何把 Lua 变成一个面向对象的语言。为了帮助大家用好这个功能，Lua 的作者们定义了一系列的工具来给大家提供语法帮助（syntactic sugar）。下面列出了我们在本系统里使用的一个。在这段代码里，the_object 是一个初始的变量，这段代码模拟了 this_call 的函数返回。

```
the_object:Test( 5 ) == the_object["Test"]( the_object, 5 )
```

面向对象的方法可以用这种语法来实现。对象被当作关系数组，用函数名来进行索引，返回的就是你需要调用的函数。Lua 里有一个机制，允许通过使用元表（metatables）来让任何类型的变量对一个数组进行交互（这是 Lua 5.1 的特性。Lua 5.0 中只有表和用户数据对象才有元表）。元表是 Lua 里的表，这个表被分配给一个对象，这个对象包含一些特殊的字段：__index、__newindex 等[Ierusalimschy06b]。在那些特殊字段里设置的函数会根据情况来被调用。当一个对象被访问时，采取的如果是数组形式的访问方式，__index 就被调用了。下面的代码说明了如何将一个元表设置到一个对象上。

```
metatable = {}
metatable._ _index = function( table, key ) return key end
```

```
setmetatable( object, metatable )
test_return = object[ "Test" ]          -- 在元表里面调用_index 函数
```

Lua 的内部类型有数字（double 或者 float）、string、table、nil、function（Lua 或者 C）、thread 和（轻）用户数据。 我们采用最后一个类型在 Lua 里保存对象。轻用户数据和用户数据稍微有点不一样。第二种完全是 Lua 对象，可以拥有一个元表。

7.1.5 在 Lua 里绑定 C++对象

绑定需要几个机制：在 Lua 里重新描述 C++对象、绑定函数的保存，最后是每个 C++对象绑定数据的注册。在本精粹里，我们先把整体的技术介绍给大家，稍后我们会讲解一些特别的例子。

绑定数据结构

在已经存在的实现中[Celes05]，绑定是直接保存在 Lua 面的，而相关的绑定数据就保存在每个脚本里。但是如果系统必须支持的脚本数量很大，那么绑定数据就会不必要地被重复保存。为了避免这样的情况发生，我们决定把绑定数据保存在 C++中一个叫做 SCRIPTABLE_BINDING_DATA 的类里。每个绑定的类都会被分配到一个索引值。SCRIPTABLE_BINDING_DATA 里包含一个记录类名和类的索引号的映射，它被保存在 ClassIndexTable 里。然后每个类也都有个映射，记录每个函数名和对应的绑定函数。MethodTable 是这种映射的数组，可以根据 ClassIndexTable 里的值进行索引。因为 delete 操作符是没有名字的，所以它的绑定被存放在了单独的数组里，这个数组叫做 DeleteTable。最后，ParentTable 保存了每个类的父类的索引。如果某个类是没有父类的，那么 ParentTable 的入口就被设置成了-1。

在本书附带光盘中，你可以找到一系列的辅助函数，这些辅助函数能够让你访问这些映射。你可以在 scriptable_binding_data.h 文件里找到它们。

```
class SCRIPTABLE_BINDING_DATA
{
    typedef int (* BINDING_FUNCTION) (lua_State *);
    std::map<std:string, int>
        ClassIndexTable;
    std::vector<std::map<std::string, BINDING_FUNCTION>*>
        MethodTable;
    std::vector<BINDING_FUNCTION>
        DeleteTable;
    std::vector<int>
        ParentTable;
};
```

指向这个绑定数据的指针和这个类的索引被存放在 lua_State 里。这个数据的空间是由 luaconf.h 里的 LUAI_EXTRASPACE 常量分配的，多出来的内存在 lua_State 创建以前就被分配好了。

作为 Lua 对象的 C++对象

被绑定的 C++对象需要把它的类索引存储在 ClassIndexTable 里，这个索引就是查找它的类的结果。这个类的索引在绑定数据里是被用来查找绑定函数的。就像我们前面提过的那样，在 Lua 里把 C++对象重现成用户数据，这个用户数据包含了指向绑定对象的指针和它的类的索引。SCRIPTABLE_BINDING_HELPER 结构可以帮我们解读代码。

在每个绑定类里会声明一个叫做 CLASS_SCRIPT_TYPE 的类。这个类在绑定过程的几个部分里会被使用到，后面会单独解释。目前这里的关注点是它存储类的索引，让 C++对象在 Lua 里存储的对象更简单。_VALUE_::CLASS_SCRIPT_TYPE::GetClassIndex()覆盖了所有类的索引。

下面的函数是在 C++代码里被用到的，它的作用就是把 Lua 里调用的参数给读取出来，然后把结果返回给 Lua。

```
template< typename _VALUE_>
void SCRIPTABLE_PushValue(
    lua_State * state, _VALUE_ & object, _VALUE_ * dummy )
{
    SCRIPTABLE_BINDING_HELPER
        *helper;
    helper
       = lua_createuserdata( state, sizeof(SCRIPTABLE_BINDING_
                                            HELPER ) );
    helper->Object = & object;
    helper->ClassIndex = _VALUE_::CLASS_SCRIPT_TYPE::GetClassIndex();
}

template<typename _VALUE_>
_VALUE_& SCRIPTABLE_GetValue( lua_State * state, int index, _VALUE_
                              *dummy )
{
    SCRIPTABLE_BINDING_HELPER
        *helper;
    helper = lua_touserdata( state, index );
    return *(helper->Object);
}
```

作为默认的设置，SCRIPTABLE_GetValue 返回指向对象的引用，但是这些函数也可以被特殊化，从而支持特定的类型，例如通过值传递来支持字符串。我们这里对每个可以转换成 Lua 基本类型的 C++类型定义了一个版本，Lua 的基本类型包括字符串、整型和浮点数。

```
std::string SCRIPTABLE_GetValue(
   lua_State * state, int index, std::string * dummy )
{
   return lua_tostring( state, index );
}
```

上面的函数签名里采用了一个小技巧，它把 dummy 指针作为第三个参数给传了进来。

这个参数是用来选择正确的重载函数的。如果使用了模板特殊化（template specialization），那么返回值总是会指向对象的引用。通过使用 dummy 这个指针，返回值根据对象返回的类型，就可以改变成那些类似于字符串和浮点数的基本类型了。

上面的代码还没有完成，用户数据还需要得到一个元表，这个元表为__index（访问数组）和__gc（垃圾收集器）定义了一个方法。当所有的绑定数据被保存在了 SCRIPTABLE_BINDING_DATA 以后，每个脚本只需要这个元表的一个实例来操作就可以了，并且这个元表的实例可以分配到所有的 C++对象上。

绑定函数的建立

绑定函数覆盖了从 Lua 堆栈里得来的参数，并且实际上是由它们来执行调用的。通过 SCRIPTABLE_GetValue 和 SCRIPTABLE_SetValue，绑定函数的创建变得很简单。this 指针总是作为第一个参数被传进来。函数参数从 2 开始索引。

```
int ENTITY_AddHealth( lua_state * state )
{
   ENTITY &entity = SCRIPTABLE_GetValue( state, 1, ( ENTITY*) 0 );
   float health_add = SCRIPTABLE_GetValue( state, 2, (float*) 0 );
   float new_health = entity.AddHealth ( health_add );
   SCRIPTABLE_PushValue( state, new_health, (float*) 0 );

   return 1;  // One return value
}
```

虽然很容易建立像这样的绑定代码，但这个任务是重复的，容易产生错误，所以我们采用了一个基于宏的系统来创建这个函数。下面是这类宏的一个例子。

```
#define SCRIPTABLE_ResultMethod1( _RETURN_, _METHOD_, _
                                  PARAMETER_0_ ) \
int _RETURN_##_METHOD_##_PARAMETER_0_( lua_State * state ) \
{ \
   CLASS \
        &self = SCRIPTABLE_GetValue( state, 1, (CLASS *) 0 ); \
   SCRIPTABLE_PushValue( \
      state, \
      self._METHOD_( SCRIPTABLE_GetValue( state,
                     2,(_PARAMETER_0_*)0 )), \
      (_RETURN_*) 0 \
      ); \
   return 1; \
}
```

这个宏只是用来创建绑定函数的，所以也要被注册到系统里去。我们可以采用一点点 C++的技巧，把这两件事情一起做了。通过使用带有静态方法的内部类的函数，就可以同时创建和注册这个方法，请看下面的代码。

```
void register_ENTITY( BINDING_DATA & binding_data )
{
    class float_AddHealth
```

```
    {
    public:

        static int Call( lua_State * state )
        {
            // ... ENTITY_AddHealth() code as above...
            return 1;
        }
    }
    binding_data.Register(
       "ENTITY", "AddHealth", &float_AddHealth::Call );
}
```

这个模式可以制作成一套通用的宏，请看下面的代码。

```
#define SCRIPTABLE_Class( _CLASS_ ) \
void register_##_CLASS_ ( BINDING_DATA & binding_data )\
{

#define SCRIPTABLE_End( _CLASS_ ) \
}

#define SCRIPTABLE_ResultMethod1( _RETURN_, _METHOD_,
                                  PARAMETER_0_ ) \
class _RETURN_##_METHOD_##_PARAMETER_0_ \
{ \
  public: \
  static int Call( lua State * state ) \
  { \
      ... SCRIPTABLE_ResultMethod1 () code as above...
      return 1; \
  } \
} \
binding_data.Register( class_name, #_METHOD_, \
  &#_RETURN__#_METHOD__#_PARAMETER_0_::Call );
```

这些宏的使用方法为：

```
SCIPTABLE_Class(ENTITY)
{
    SCRIPTABLE_ResultMethod1(float,AddHealth,float)
}
SCRIPTABLE_End(ENTITY)

//...and then at start of day, register the class...
register_ENTITY(binding_data);
```

属性（例如类的数据成员）也是由系统来掌管的。标准的绑定允许通过 object.attribute 来访问属性。要掌管这种访问，__index 和__newindex 中继承方法就必须被重写。为了避免这样做，当一个属性和 SCRIPTABLE_Attribute 绑定的时候，在 Lua 里就需要建立 Set 和 Get 函数。本书附带光盘里的例子包含了这个宏的定义，以及使用这个宏来操作 vector_3 类的例子。

现在大家可以从我列出来的内容里知道，对于每个类的绑定建立和注册都是由一些宏来控制的。除此之外，还需要一个简单的方法来调用这些函数。

类型的自动注册

为了尽可能保证系统对用户的透明度，我们决定把注册函数封装进一个类中。每个绑定的对象都要声明一个从 SCRIPTABLE_TYPE 衍生出来的内部类。这个内部的类有一个静态的成员，类的构造函数会把它加进一个全局的表格里去。这个表格在程序启动时就会被载入并且运行，它就会去调用每个注册函数。

```
class SCRIPTABLE_TYPE
{
   public:
       SCRIPTABLE_TYPE()
       {
           SCRIPTABLE_TYPE_TABLE::Add( this );
       }
       virtual void Register( BINDING_DATA & binding ) = 0;
}
```

然后再把下面的代码插入绑定的类中。

```
class CLASS_SCRIPT_TYPE;
friend class CLASS_SCRIPT_TYPE;
class CLASS_SCRIPT_TYPE :
    public SCRIPTABLE_TYPE
{
    public :

    typedef GAME_ENTITY CLASS;
    CLASS_SCRIPT_TYPE( void );
    static const char * GetClassName() { return "GAME_ENTITY"; }
    static int & GetClassIndex()
    {
        static int index = -1;
        return index;
    }
    static int Delete( lua_State * lua_state );
    virtual const char * GetName() const { return GetClassName(); }
    virtual int & GetIndex(){ return GetClassIndex(); }
    virtual void Register( SCRIPTABLE_BINDING_DATA & binding );
};
```

内部类包含 Register 函数、Delete 函数、绑定类的名字和它的索引。内部类的定义被隐藏在 SCRIPTABLE_DefineClass 宏中。这个宏的细节内容可以在本书附带光盘的演示程序里找到。

```
class GAME_ENTITY
{
```

```
public:
SCRIPTABLE_DefineClass( GAME_ENTITY )

};
```

在整个系统都到位的情况下，你可以在这张表里调用所有的注册函数，就像下面的代码所显示的。

```
void SCRIPTABLE_TYPE_TABLE::Register( BINDING_DATA & binding_data )
{
    int type_index;

    for( type_index = 0, type_index < TypeTable.size(); ++type_index )
    {
        TypeTable[ type_index ]->Register( binding_data );
    }
}
```

7.1.6　扩展绑定系统

本小节将解释如何扩展这个绑定系统。

引用计数和原始对象

Lua 里对象的实例其实包含了对象自己的指针。当一个对象被删除以后，如果 Lua 还是包含这个对象的变量，问题就会产生。为了避免这个问题，我们用以下两种方法来控制对象。

- 如果一个对象的引用被计数了，那么 Lua 就要增加引用的计数。甚至如果这个 C++ 的对象没有在 C++程序这里被引用，也不能在 Lua 没有释放以前删除。当 Lua 的变量被回收的时候，关联到__gc 元方法上的 Delete 函数就会被调用，然后这个方法会把用计数减小 1。
- 如果这个对象是不统计的，例如容器，那么就要建立它的一个副本。在这种情况下，Delete 函数就直接把这个对象删除。

对于每个类来说，都要选择这两种方法中的一种来管理自己的实例。我们做了两个定义，SCRIPTABLE_Class 给引用计数的对象，SCRIPTABLE_UncountedClass 给不引用计数的对象。这些宏必须出现在类的定义里。光盘里的演示程序也表明了这两种方法的工作方式。

继承

继承是通过在 ParentTable 里存储父类的索引来的。如果在目前的绑定类里找不到某个方法，程序就会去搜索它父类的绑定。函数 BINDING_DATA:: GetFunction 就是用来做这样的搜索的，在本书附带光盘里能够找到这个函数。通过宏来创建的注册函数包含了一个调用，这个调用可以把父类设置给目前你在使用的类上面。SCRIPTABLE_Class 用于基础类，否则用 SCRIPTABLE_Inherited 类。

支持继承也就意味着被继承的类可以作为一个参数被放到一个函数里。SCRIPTABLE_PushValue 宏里有类的索引的硬编码。为了避免这种问题，我们可以把绑定函数放在类的一个虚函数里。下面我们看一个例子，SCRIPTABLE_PushValue 函数的模板版本在下面的代码里，它调用 LuaPushValue 的派生版本。这个函数藏在了 SCRIPTABLE_DefineClass（虚函数）和 SCRIPTABLE_DefineRawClass（非虚函数）里。

```
template< typename _VALUE_>
void SCRIPTABLE_PushValue(
    lua_State * state, _VALUE_ & object, _VALUE_ * dummy )
{
    object.LuaPushValue( state );
}
```

单件、静态函数和属性

单件、静态函数和属性有一个共同点：没有对象是与函数调用相关联的。为了尽可能地和 C++语法保持一致，我们在函数名前加上类名来调用这样的函数，例如 WORLD:GetEntity()。这个调用触发了一个在全局表里查找变量 WORLD 的举动（在 Lua 里，所有全局变量的名字都保存在一个叫做_G 的表格里）。还有一个简单方法，是在 Lua 里为每一个 C++的类都建立一个变量，但是这样做就会占用比较多的内存。因为即便你的脚本不会用到类函数或者类里没有静态函数，还是会有一个这个类的变量被建立，并塞到这个全局的表格里去的。

当然，我们也可以在全局表格里加入__index 机制。当一段脚本访问的全局变量不在全局表里时，SCRIPTABLE_LUA_REGISTERER::GlobalIndexEventHandleris 就被调用了。如果这个访问的变量名和类的名字匹配，那么一个新的对象和它对应的空指针就会被建立并插入到全局表里，采用的名字就是这个类的名字。如果不匹配，就像 Lua 平时做的那样，句柄就会返回一个空，告诉大家在表格里没有找到。使用这种方法，只有那些被用到的类的变量才会建立，一旦这个事件处理函数被调用了，这个变量就会在全局范围里有效。这个机制对于其他变量的访问是透明的。

模板类

绑定也能够控制模板类。模板不需要是个绑定类，但在这个情况下，模板类需要使用 SCRIPTABLE_DeclareScriptableTypeName 在 Lua 里定义它自己的名字。所有内部的类型已经被声明好了。在 Lua 里类的名字就是用模板类的名字加上参数名做后缀。因此 RANGE_OF_<VECTOR_3>在 Lua 里变成了 RANGE_OF_VECTOR_3，还是十分一致的。如果这样的变换对你来说不是很舒服，你可以很容易地修改这个系统。

在宏方面，其实和非模板类是一样的，只是加上了一个“Template”，例如，SCRIPTABLE_Class 变成了 SCRIPTABLE_TemplateClass。这样的绑定需要一个 cpp 文件来包含所有的定义。绑定必须用 SCRIPTABLE_InstantiateTemplateClass 明确的实例化。例子里有一个空的模板类，这个例子告诉大家这个绑定是怎么做的。这个绑定对于单个参数的模板类有效，但如果想要支持多参数，也是很容易扩展的。

枚举的支持

我们的系统里也支持枚举类型。SCRIPTABLE_PushValue 和 GetValue 被重新定义，并且通过使用 SCRIPTABLE_CastValue(ENUM_TYPE, int)来把它们作为整型处理。为了让用户能够在代码里使用枚举的名字，在代码里的预处理器也被制作好了，通过替换所有匹配的枚举值来实现。在预处理器里有一个宏系统来创建和注册文字，就像#define 那样。细节就不在本精粹中一一介绍了。

绑定重载的函数

C++支持函数重载。在不同的形式下，SetValue 可以用在 bool、int 和 real 等类型。Lua 并不支持重载。这里我们有两个解决方法可以让它支持：第一个是用 BINDING_DATA::GetFunctioncan 来做参数匹配的比较工作，但是这样做的代价太大；另外一个方法就是在 Lua 里重定义函数的名字。SCRIPTABLE_Renamed*宏允许在 Lua 里修改函数名字，所以 SetValue 可以被重新命名为 SetBoolValue、SetRealValue 或者其他名字。

调试用的辅助功能

因为绑定函数是由宏的实例来创建的，所以很容易对所有的绑定函数增加调试函数和断言（assert）。调试系统通过预处理器定义_LUA_DEBUG_来实现，它会检查参数的计数和类的类型。如果在检测中发现了错误，就会触发 Lua 的错误。这个调试系统是设计并被用在 C++release 版本下的，这样做可以使调试的速度大大提高。使用 Lua 来报错的一个好处就是游戏不会崩溃，就只是退出调用而已。这样的方式适用于你使用的任何 Lua 调试器。绑定错误也能像 Lua 错误一样进行处理。调试系统可以在发行版本的编译中被完全屏蔽掉，从而提高编译的整体速度。

7.1.7　结　论

要绑定一个类，必须把 SCRIPTABLE_DefineClass(MY_CLASS)这个宏放在类的定义里。你可以设置 4 个选项。

```
SCRIPTABLE_DefineClass : LuaPushValue is virtual
SCRIPTABLE_DefineRawClass : LuaPushValue not virtual
SCRIPTABLE_DefineTemplateClass : template class, LuaPushValue is
                                 virtual
SCRIPTABLE_DefineRawTemplateClass : template class, LuaPushValue
                                    not virtual
```

模板参数是作为宏里的第二个参数的。在 cpp 文件里，类的绑定执行必须像下面那样被设置好。

```
SCRIPTABLE_Class( MY_CLASS )
{
   SCRIPTABLE_VoidMethod( SetValue, float )
```

```
}
SCRIPTABLE_End()
```

选项是这样的：SCRIPTABLE_(Uncounted)(Inherited)(Template)Class。如果这个对象没有被计入引用的计数，那么 Uncounted 宏版本就应该被用到。如果这个类是从其他类继承过来的，请使用 Inherited。示例代码基本覆盖了所有对象类型的定义。

7.1.8 后续工作

本小节将介绍这个系统的一些可扩展的地方。

在 Lua 里重载 C++函数

存在于 Lua 里的 C++对象可以作为 Lua 对象来使用。重载这些 C++对象是一个很有趣的扩展。例如，这样做就可以勾住（hoo）函数的调用。为了侦听所有对 GetHealth 的调用和打印目前的健康值，GetHealth 方法可以被重载，如下所示。

```
Object.GetHealth =
   function( self )
       local health = ENTITY.GetHealth( self ) -- Do the call to C/C++
       print( "Health of object " .. self .. " is " .. health )
       return health;
   end
```

当元表里的一个值通过数组访问被设置以后，__newindex 就会被调用。这个索引会把 Lua 的函数存进表里，用来代替 C 的函数。__index 函数也被相应地改变来影响一个新的行为：在数组访问里，索引函数先搜寻 Lua 函数表格，然后搜索 C++绑定数据。虽然这个方法看上去能行，但是它也会产生一个问题——继承对象必须在 Lua 里用某种方法来保存。如果它被垃圾收集器收集了，它的重载就会消失。在示例里演示的程序并不支持重载，但是支持对象的持久化。当每次产生一个对 SCRIPTABLE_PushValue 的调用时，在表_object 里的 Lua 对象就会被查询起来。如果这个对象存在，那么它就会被继续使用；如果不存在，Lua 版本的对象就会被建立。示例里有这样的代码。

沙盒和类型过滤

目前描述的这个系统允许对象绑定到 Lua 里去，但是你或许也想用沙盒来模拟一些脚本，以限制它们的访问。底层脚本可以用来做配置文件，配置诸如文件 I/O 或者图形配置的属性，而用户级的脚本则只能访问一系列制定的类。访问级别的定义可以在宏里定义好，并且保存在 SCRIPT_TYPE 里。注册函数能够通过访问等级来访问。在下面的代码里，WORLD 类就是被声明成用户级的。SCRIPT_MANAGER 包含了 BINDING_DATA。几个管理器通过不同的访问等级被建立起来。当某个管理器被初始化时，它就会通过自己的绑定数据和访问等级去调用 SCRIPT_TYPE_TABLE::Register。绑定数据里只包含它能够访问的那些类。脚本就通过这个管理器来创建和访问，并访问它的绑定数据。

```
SCRIPTABLE_Class( WORLD, ACCESS_LEVEL_User )
```

```
void SCRIPT_TYPE_TABLE::Register(
   BINDING_DATA & binding_data, const ACCESS_LEVEL access_level )
{
   int type_index;

   for( type_index = 0, type_index < TypeTable.size(); ++type_index )
   {
       if( TypeTable[ type_index ]->GetAccessLevel() >= access_level )
           TypeTable[ type_index ]->Register( binding_data );
   }
}
```

优化代码生成的大小

这里描述的技术可以为所有绑定的函数建立一个绑定函数，这个方法可以产生许多的代码，一次又一次地执行相同的职责。一个解决方法就是建立函数群的描述，而不是绑定的函数群。绑定数据是一个存储了 FUNCTION_DESCRIPTION 的类。

```
struct FUNCTION_DESCRIPTION
{
   const char * FunctionName;
   const void * FunctionPointer
   int ArgumentCount;
   ARGUMENT_DESCRIPTION * ArgumentDescription;
   RETURN_DESCRIPTION * ReturnValueDescription;
}
```

宏系统填充了一个数组，这个数组里都是这样的描述：

```
#define SCRIPTABLE_VoidMethod1( _METHOD_, _PARAMETER_0_ ) \
{ #_METHOD_, &CLASS::_METHOD_, 1,
   { GetArgumentDescription<_PARAMETER_0_>() }, 0 },
```

现在我们只需要一个绑定函数就可以做到了，伪代码如下：

```
for each argument_index  < description.ArgumentCount
   description.ArgumentDescription[
     argument_index ].PushArgumentOnCStack();

call( decription.FunctionPointer );

if( description.ReturnValueDescription )
    description.ReturnValueDescription->StoreResultValueInLua();
```

这个代码必须有一部分用汇编来写，因为它访问了 C 的堆栈。为了节省内存，这个系统会有些慢，但如果你在一个内存有限的系统中工作，就会变得有用。

7.1.9 例 子

本书附带光盘里提供的代码包含了所有功能的演示，而且也应该很容易就能够移植到其他代码里去。例子已经尽力覆盖了这里介绍的所有功能属性。如果你启动例子程序，除了几行文字外，你应该看不到什么东西。一步一步看里面的代码才是理解这套系统最好的方法。一旦这些细节都清楚了，你就可以自己动手扩展宏和代码，并进一步了解细节的工作情况。

7.1.10 结 论

本精粹介绍了一个自动绑定系统。用户只需要设置一些绑定类的声明，再次编译，类就能够通过脚本来访问了。你并不需要知道这个系统或者 Lua 绑定是怎么回事。这个系统对于 CPU 和内存的占用也比较少，而且还提供调试帮助，例如对于零散编译的参数的检查可以被忽略等。本精粹也介绍了一些 C++的技巧，例如自动类型注册和利用假指针进行函数选择。掌握了这些工具，任何人都应该可以把绑定的方法加入到它们的引擎和脚本语言里。

7.1.11 参考文献

[Celes05] Celes, W., de Figueiredo, L.H., and Ierusalimschy, R. "Binding C/C++ Objects to Lua," *Game Programming Gems 6*, edited by Michael Dickheiser, Charles River Media, 2005, pp. 341-356.

[Ierusalimschy06] Ierusalimschy, R., de Figueiredo, L.H., and Celes, W. "Lua 5.1 Reference Manual," available online at Lua.org, 2006.

[Ierusalimschy06b] Ierusalimschy, R. "Programming in Lua (second edition)."

7.2　用内省（introspection）方式把 C++对象序列化到数据库中

Joris Mans
joris.mans@10tacle.be

因为需要内容创建工具管理的素材的数量一直在增长，所以管理这些素材变得越来越难，尤其是当大量的东西不能简单地、一次性地加载到内存中去的时候，难度可想而知。因为用户想方便地找到自己需要的东西，所以在使用的时候总是想浏览所有的内容，关键词搜索、分类和结构图就成了用户们浏览素材的方法。另一个问题就是：有许多内容制作者希望能够使用相同的共享素材，而素材的制作者们则相应地想把做好的东西尽快暴露给内容制作者们。

所以我想到了一个方法来解决这个问题，那就是采用数据库后端。本精粹将介绍一个可以把 C++对象存储进 SQL 数据库的系统，取出的时候可以使用过滤器将它们分类。我采用的例子中使用了 PostgreSQL 8.2 和 Microsoft Visual Studio 2005。

7.2.1　元数据（Metadata）

在把数据序列化存进数据库以前，你需要在代码中加入一些内省的工具。如何高效、完善地执行元数据系统（这里后面我们就会说到）超越了本精粹的内容范围，所以我将只会介绍数据库序列化所需要用到的那些内容。

类的元数据被保存在一个类的实例中，这个类叫做元类型（MetaType）。为了使这个系统能够工作起来，你需要从这些类中获取下列信息：

- 类的名称；
- 父类的名称；
- 属性表，对每一个可序列化的属性包含一个 MetaAttribute 的实例；
- 对象实例的大小，单位是字节。

这个类可以允许你随心所欲地处理任意类型的对象，而不用使用多态或者 RTTI。

属性

对于类中每一个你需要序列化的属性，你要把它的信息加载到元数据中去。为了做到这一点，你还要建立一个叫做 MetaAttribute 的类，这个类包含了下列信息。

- 属性名称；
- 属性的元类型；
- 该属性在对象中的偏移，单位是字节；
- 这个属性是不是指针；
- 这个属性是不是数组。

每个 MetaAttribute 实例都将会被加入到属性表中去，和元类型的实例相对应。在本书附带光盘中可以找到这些类的代码。

7.2.2 数组

本精粹使用一个特别的数组类，这是一个从 ArrayBase 继承下来的模板类。基础类包含了一个接口，这个接口允许你查询数组内元素的个数，也可以设置个数并得到数组内被使用到的那些类的元类型。它也允许你得到数组里数据的指针。通过这样的方法，你可以在不知道数组中对象类型的情况下对它进行操作，这就是你操作数据库系统中的对象时需要的方法。

7.2.3 序列化成文本

在实现这个数据库系统时，你还需要做一件事情。你需要把一些简单的类型序列化成文本格式（text format）。例子中采用了一些 C 函数的重载来实现这个功能。

```
void WriteObjectToText(
    const void * object,
    const MetaType & meta_type,
    std::string & output_text
    );

void ReadObjectFromText(
    void * object,
    const MetaType & meta_type,
    const std::string & input_text
    );
```

这些函数支持读写标量对象和字符串，例如：

```
int a = 5;
std::string output_text;
WriteObjectToText( &a, META_TYPE_GetStaticMetaType( a ),
                   output_text);
```

这会使 output_text 包含字符串“5”。

7.2.4 数据库系统

在开始序列化数据，并把它们保存进数据库之前，你要考虑到 C++的对象也需要序列化。

那么这些对象都包含什么呢？

一个 C++的类由下面的组合构成：

- 标量成员（例如，int、float、char 等）；
- 如果有，一个或者多个父类；
- 指针；
- 其他 C++类的实例。

在要介绍的情况中，我要小小地修改一下。为了达到我们的目的，一个 C++的类由下面的内容组成：

- 标量成员；
- 字符串；
- 如果有，一个父类；
- 指针；
- 其他 C++类的实例；
- 指针的数组，其他 C++类或者标量的实例，这里的数组是使用我们的数组类型。

在本精粹中介绍的系统可以扩展来支持更多 C++类的属性（多重继承、其他集合类型和其他等），但是扩展的程度太广了，所以我做了一下限制，只是介绍上面的内容。

表

因为你要把对象存储到关系数据库中，所以你需要定义一些表来存储这些对象。每个表相关于一个类，主键就是_Identifier，整型自增长（主键的名字可以自己定义，只要不和对象的名字以及对象中属性的名字冲突就可以了）。

使用前面定义的属性类型，你就会知道如何将每个类型保存在数据库的字段中了。对于每个属性来说，属性的名称和字段的名称是相关联的。

标量成员

每个标量成员被保存在标量类型的字段里，数据库的标量字段类型有整型（integer）和浮点（real）。

字符串

每个字符串都被保存在 varchar 字段中。

父类

在数据库中，你可以使用任何表作为另外一个表的字段类型。这里的例子中建立了一个字段叫 _Parent，这个字段是表示父类的表类型（你可以使用这个字段的名称作为那个表的名称，只要不冲突就可以）。请看下面的类。

```
class Base
{
   int a;
};
class Subclass : public Base
{
   int b;
};
```

建立一个叫 Base 的表：

```
CREATE TABLE "Base"
(
   "_Identifier" serial,
   "a" integer
) ;
```

再建立一个叫 Subclass 的表：

```
CREATE TABLE "Subclass"
(
   "_Identifier" serial,
   "_Parent" "Base",
   "b" integer
);
```

指针

这里有几个方法可以将指向一个对象的指针保存在数据库中。首先你可能需要考虑到，这个指针会是一个外键，这个外键指向相关表，这个表也就是指针指向的那个对象。例如：

```
class Base
{
   int a;
};
class StoreInDatabase
{
   Base * basePointer;
};
```

你可以像这样建立一个叫 Base 的表：

```
CREATE TABLE "Base"
(
    "_Identifier" serial,
    "a" integer
);
```

然后还有一个表叫 StoreInDatabase：

```
CREATE TABLE "StoreInDatabase"
(
    "_Identifier" serial,
    "basePointer" integer,
);
```

现在来建立一些实例：

```
Base * base_object = new Base;
StoreInDatabase * store_object = new StoreInDatabase;

base_object->a = 10;
store_object->basePointer = base_object;
```

你可以像这样把它们保存到数据库里：

表 Base:

```
_Identifier     a
1                10
```

表 StoreInDatabase：

```
_Identifier     basePointer
1               1
```

从数据库里获取对象看上去很直接。你从表 StoreInDatabase 里读取内容，使用在字段 basePointer 中找到的值，从 Base 表中取得相应的行，从中找到指向的对象。

看看下面的例子：

```
class Base
{
   int a;
};

class Subclass : public Base
{
   int b;
};

class StoreInDatabase
{
   Base * basePointer;
};
Subclass * subclass_object = new Subclass;
StoreInDatabase * store_object = new StoreInDatabase;

subclass_object->a = 10;
subclass_object->b = 20;
store_object->basePointer = subclass_object;
```

如果你也想在这个例子上采用同样的系统，那就会很麻烦，保存的数据会出现下面的结果：

表 Subclass:

```
_Identifier   b      _Parent
1             20     {10}
```

表 StoreInDatabase:

```
_Identifier     basePointer
1               1
```

当你要从数据库中获取对象时会遇到这样的问题，就是从 basePointer 里获取值的时候，还没有办法知道你保存的是 Subclass 还是 Base。你根本不知道在 basePointer 里存储的键相对应的是哪个表。

当然，我们也有好几个方法来解决这个问题。本精粹使用的是其中的一种，也希望大家有时间去尝试一下其他的方法。让我们试试保存下面格式的一个字符串，而不是用一个整型来保存主键索引指向的对象：

```
"( primaryKeyValue, tableName )"
```

对上面的例子采用这个方法以后，就得到了如下的结果：

```
CREATE TABLE "Base"
(
   "_Identifier" serial,
   "a" integer
);

CREATE TABLE "Subclass"
(
   "_Identifier" serial,
   "_Parent" "Base",
   "a" integer
);

CREATE TABLE "StoreInDatabase"
(
   "_Identifier" serial,
   "basePointer" varchar,
);
```

保存同样的对象会在这些表里产生这样的值：

表 Subclass：

```
_Identifier     b      _Parent
1               20     {10}
```

表 StoreInDatabase：

```
_Identifier     basePointer
1               "(1,Subclass)"
```

当读取字段值时，你可以采用字符串操作的方法来得到主键部分，还有存储在 basePointer 的字段值的表的名称部分，然后就可以使用对应的表来获取指向对象的内容。

另一个方法是保存一张表，这张表里包含所有在数据库里的类的名称，用这个方法来取代一对一对地保存（primaryKeyValue, tableName）指针，字段会有（primaryKeyValue, classNamePrimary KeyValue），classNamePrimaryKeyValue 包含了主键，主键索引类的名称。特别是在大的数据库里，这种方法就会更有效率一些，因为和数据库相对而言，从数据库里获取的数据数量更小，而且还因为类的名称在不同的表里没有重复出现。

其他 C++类的实例

你可以像保存父类一样保存其他的类，通过增加一个属性字段，这个字段和相应类的表有关联。

例如：

```
class Base
{
    int a;
};

class StoreInDatabase
{
    Base baseInstance;
};
```

```
CREATE TABLE "Base"
(
    "_Identifier" serial,
    "a" integer
);

CREATE TABLE "StoreInDatabase"
(
    "_Identifier" serial,
    "baseInstance" "Base"
);
```

除非其中的对象有指向其他对象的指针成员存在，否则这种方法是有效的。比如：

```
class Base
{
    int a;
};

class StoreInDatabase
{
   Base baseInstance;
};
```

```
class AnotherToStoreInDatabase
{
    Base * basePointer;
    int b;
};
StoreInDatabase * store_1 = new StoreInDatabase;
AnotherToStoreInDatabase * store_2 = new AnotherToStoreInDatabase;

store_1->baseInstance.a = 10;
store_2->basePointer = &store_1->baseInstance;
store_2->b = 20;
```

在 store_2->basePointer 中，你是没有办法保存指针值的，因为它引用的另一个对象的一部分不是一个在数据库里完整的一行。对于这个问题，我们可以像保存指针一样地保存 C++ 对象的实例。你把类相关的实例存放到表里去，在包含这个实例的对象的表里建立一个 varchar 字段，然后把包含主键以及表名的字符串写到字段里去。这里有一个例子，使用的是先前介绍的例子里的对象。

```
CREATE TABLE "Base"
(
    "_Identifier" serial,
    "a" integer
);

CREATE TABLE "StoreInDatabase"
(
   "_Identifier" serial,
   "baseInstance" varchar
);

CREATE TABLE "AnotherToStoreInDatabase"
(
   "_Identifier" serial,
   "basePointer" varchar,
   "b" integer
);
```

对象保存的结果是这样的：

表 Base:

```
_Identifier     a
1               10
```

表 StoreInDatabase：

```
_Identifier     baseInstance
1               "(1,Base)"
```

表 AnotherToStoreInDatabase：

```
_Identifier     basePointer     b
```

```
1                "(1,Base)"      20
```

数组

因为数组是数据库支持的一种数据类型，所以你可以像前面一样使用同样的方法，但是是用数组的方式来保存在字段中。如果是一个标量，那就是简单的一串标量；如果是指针，那就是一连串的 varchar，诸如此类。

建立表格

通过对每个类的实例使用 MetaType，你可以在数据库中生成表。使用下面的伪代码，你可以生成一个包含 SQL 语句的字符串，用这个字符串就可以建立表了。

```
procedure AddTypeToDatabase( meta_type )
begin
  if meta_type.HasParent()
      if NOT(TypeExistsInDatabase( meta_type.parentClassName )
          AddTypeToDatabase( GetMetaType( meta_type.parentClassName ) )
      endif
   endif

  sql_statement =  "CREATE TABLE" + meta_type.className
  sql_statement +=  "_Identifier serial,"

  foreach attribute in meta_type.attributeTable
      sql_statement += GenerateCreateAttributeStatement( attribute )
  endfor

  if meta_type.HasParent()
      sql_statement += "_Parent " + meta_type.parentClassName
  endif

  ExecuteSqlStatement( sql_statement )
end
```

在这段伪代码中，GenerateCreateAttributeStatement 生成了建立和属性类型相对应的字段的 SQL 语句。对于一个 myText 的字符串属性，这段伪代码会生成包含“myText varchar”的几行语句。实际的代码会在账号中加入一些内容，例如分割字段声明的逗号以及生成正确的引用，在 SQL 语句中不会造成大小写问题。

在执行 SQL 语句时需要注意的一件事情就是，你要确认先建立基类，然后再建立基类下的类，否则数据库会报错，说_Parent 使用了一个未知的类型来声明，所以在过程开始的时候要做类型检查，使用 TypeExistsInDatabase。

相关代码在本书附带光盘的例子中可以找到，具体的函数是：

```
bool DatabaseManager::CreateTable( const MetaType & meta_type )
```

保存一个对象

保存一个对象是通过几个步骤来做到的。首先，得到对应类型的表中一个新的主键值。然后，把对象插入到表里去，这个对象除了自己的主键以外，其他的字段都是空的。最后，要更新表里的这个对象，把它的属性都填入进去。为什么这么麻烦？为什么不直接把对象存进表里，然后让数据库自动生成这个对象的主键值？原因很简单，当你去搜索并得到这个对象时就知道了。

记住，在数据库表里的每个对象的实例都是通过唯一的主键来识别的。从C++的角度来说，每个对象的实例都有它自己唯一的一个内存地址。想象一下你正在跑你的程序，而且有一个对象是在你的数据库的表里的，你向数据库请求这个对象，程序就去执行一个查询，然后你就得到了结果。通过这个结果，你在内存里建立了C++的对象实例，然后返回一个指针。到目前为止一切都还好。

然后你又要去执行这个相同的查询。如果这一幕再次出现，那你就会遇到麻烦了，因为系统会在内存里再次建立一个相同的对象，而这个对象在数据库里只存在一个。实际上，你是想让系统返回一个先前已经建立好的对象的指针的。

解决这个问题的一个方法就是在数据库管理器内建一个实例的表，这个缓存保存了主键、表名和对象实例之间的关系。再想想在这样的条件下，第一次你查询的时候会发生什么事情，系统会建立对象的实例，赋值，然后把指向这个对象的指针、表名和主键都存进这个缓存里。下次你再找这个对象的时候，数据库系统就会通过主键在这个缓存里找，而不是去新建。

但是要保存这个对象，我们应该怎么做呢？让我们想象一下下面这种情况。程序在数据库里插入一个对象，然后后面的某段时刻再去查找。原来那个插入数据库的对象其实现在还在内存里，于是数据库系统理应把指向原先那个对象的指针返回给程序，而不是建立一个新的实例。所以在插入的那个时刻，对象也应该放到缓存里去。因为你需要主键值来保存这个对象，所以在查找时也会用到。除了主键，其他没有任何更好的方法来查找获取对象了。甚至你用SQL语言里的SELECT和WHERE来把所有的属性值作为查找条件来搜索这个对象也不会很有效，因为需要匹配的东西太多了。

所以你需要准确无误地得到主键值。但是为什么先插入一个空的对象，然后再更新里面的值？这完全是为了让代码看上去更简洁。因为SQL语句里对于插入和更新的语法是不一样的，如果你对同一个对象进行插入然后再更新，需要的代码量会少一些。

把对象保存到数据库里去的伪代码如下：

```
procedure StoreObjectInDatabase(object, meta_type )
begin
   BeginTransaction();
   InsertObjectAndAttributePrimaryKeys( object, meta_type );
   UpdateObject( object, meta_type );
   EndTransaction();
end
```

一些需要重点注意的地方是对BeginTransaction和EndTransaction的调用。因为你执行的是多个连续的SQL语句，所以始终保持数据库状态的一致性是很重要的，使用事务处理允许

你回滚数据库，如果在 BeginTransaction 和 EndTransaction 之间的有什么东西让程序崩溃，就可以进行回滚来恢复数据。想象一下，如果你的程序恰恰在你插入一个只有主键的空记录时崩溃了，下次你运行程序到这个记录，得到的数据将是不完整的内容。用事务处理来保护这些数据可以保证直到 EndTransaction 被调用才会有数据真正写入数据库中。

让我们更进一步来看看保存对象的这两个函数。第一个函数是 InsertObjectAndAttributePrimaryKeys。在这个函数里，你需要执行的是下面几步。

- 如果对象已经在实例表里存在了，或者它是数据库内建类型（例如整型和字符串等），那么就返回。
- 重复这个对象的所有属性以及上级类，对它们每一个都执行一次 InsertObjectAndAttributePrimaryKeys。
- 检查一下和这个对象相对应的表是不是已经存在于数据库中，如果没有，则建立一个。
- 最后，所有这个对象的属性和上级类都被递归地处理，然后你就为这个对象产生一个主键，把这个主键以及这个对象保存到数据库中。

第二个函数是 UpdateObject。该函数有以下几步。

- 如果这个对象是数据库内建类型，那么返回。
- 重复这个对象的所有属性和它所有的上级类，然后对它的属性和上级类都执行一次 UpdateObject。
- 建立和执行 SQL 语句，更新对象的内容。

更新对象的内容

生成一个 SQL 语句的 UPDATE 由 4 部分组成。首先是要更新的表的名称，其次是要更新的字段名称，再次是那些字段的更新值，最后是一些条件来决定哪些行需要被更新。

选择要更新的表的名称是很简单的；表的名称和保存在对象的元类型中的类名称是一样的。语句中的条件部分也很直接，就是主键。主键可以从实例表中找到。我们要关注的就是字段名称和值。让我们从字段名称开始着手研究。

C++类中的数据是由一些属性以及类的上级类组成的。当你要建立字段名称的列表时，你可以从类中的属性着手。这个很简单，因为字段的名称和属性的名称是一样的。对于父类来说，你就直接把父类中的内容保存在一个叫做_Parent 的字段中。在那个字段中的属性可以像在 C++类中访问成员一样地来访问，通过把这个类写在_Parent.attributeName 表格里。如果父类中还有更上一级的父类中继承的属性存在，你也可以用同样的方法来访问_Parent._Parent.attributeName。然后，如果你要为上级类用 SQL 语句来建立一个字段名称的列表，那就要遍历迭代每一个上级类的属性，把属性的名称都写出来，在一个或者多个_Parent 中把名称的前缀加好。对于每上进一级，你就要在前面多加一个_Parent.。

例如：

```
class Base
{
   int a;
};
```

```
class Subclass : public Base
{
   int b;
};

Class SubSubclass : public Subclass
{
   int c;
};
```

产生类 SubSubclass 的属性的名称，结果如下：

```
c
_Parent.b
_Parent._Parent.a
```

生成每个字段的值当然应该是按照字段的顺序来的，否则数据就会无法对齐。根据每个属性的类型不同，你有不同的方法重复每个属性的操作。下面介绍 3 种情况。

数据库内建类型

内建类型有标量和字符串。用函数 WriteObjectToText 可以把属性转换成字符串类型，字符串类型可以在 SQL 语句中使用。

对象或者指向对象的指针

在这种情况下，你拿到的是对象在内存里的地址（或者指针指向的对象），你再要得到元类型，然后从实例表里得到主键。记住，实例表是一定包含那些值的，因为你在生成 UPDATE 语句以前已经建立了它们。有了这个主键，就可以建立包含主键-类名对的字符串了，就像在“指针”部分中介绍的那样。

数组

当你遇到数组时，你就用 SQL 中相应的数组来重现就可以了。格式就是“{item1, item2, ..., itemN}”。后面当你获取对象的时候，你需要知道的就是数组里元素的个数。其实你只要简单地将个数保存在数组的第一个单位里，然后你要取得数组中保存的对象的元类型，对每个元素进行重复操作，再对数组中的元素做指针再执行。

如果数组包含指向别的对象指针，可以对这些指针重复执行下面的代码。

```
item_address = *( array_data + sizeof( void * ) * item_index )
```

array_data 就是数组缓存的开始，item_index 就是数组中元素的索引。

如果数组中包含对象的实例，那么这段代码就会变成这样：

```
item_address = array_data + item_meta_type.GetByteCount() * item_index
```

在这个情况下，item_meta_type 就是数组中元素的元类型（在标准模板数组类中，它就

会是模板参数类型的元类型)。通过使用这样的字符串，你就可以编译 SQL 的 UPDATE 语句并且修改数据库中对象的内容了。

取回对象

当插入记录的时候，取回对象分两步走：首先，你要执行一个 SELECT 语句，把主键找回来；其次，对于每一个主键，你都要去实例表中查一查有没有对象已经存在了。如果有，那就把指针返回。如果不存在，那么就要再建立一个 SELECT 语句，把字段中的值都取回来。这个查询的结果被用来把你想得到的对象的属性填好，伪代码如下：

```
procedure GetObjectsFromDatabase( object_table, meta_type, predicate )
begin
   key_table = GetAllPrimaryKeysCorrespondingToPredicate( predicate )
   foreach key in key_table do
      if HasObjectInInstanceTable( key, meta_type )
         object_table.Add( GetObjectFromInstanceTable( key,
                           meta_type ) )
      else
         object_table.Add( CreateNewObject( key, meta_type ) )
      endif
   endfor
end
```

predicate 包含了查询的过滤器，meta_type 是你要取回的对象的类的元类型，object_table 就是含有查找结果的表，这个表就是一系列指向实例的指针。要建立新的对象，就要使用这个对象的元类型来建立一个新的实例。MetaType 类有一个方法调用了 CreateNewInstance 来完成这个任务。

下面就要生成 SELECT 语句了。在 UPDATE 语句中，你生成了一列类的属性和类的上级类的字段名称，后面要用这个来指定字段中的值以及填充的顺序。当你重复将结果加入到查询里时，你需要考虑到属性的不同类型。

数据库内建类型

ReadObjectFromText 用来把对应值的字符串版本转换成该属性的字符串或标量值。

指向对象的指针

当你在字符串里得到主键加类的名称时，首先要去实例表中检查一下这个主键和类是不是已经存在。如果有了，那么就把它的地址放到指针中去；如果没有，就建立一个新的实例，并执行代码到数据库中，把这个对象的内容找出来，存放到新建立的实例中去。

对象

在前一个例子中，你得到了主键和类的名称，这一对关系存放在字符串中。因为这是一个属性，所以你在实例表中是没有它的地址的。你需要把地址加入到实例表中，同时把它的主键和元类型一起存放进去。去执行一个 SELECT 语句，得到它的属性值，然后把内容填充进去。

数组

数组是通过字符串格式 format "{ item_count, item1, item2, ..., itemN}"来表达的。通过字符串操作，你就能够得到 item_count，然后建立一个指针指向这个数组，再强制转换这个指针指向的类型为 BaseArray，然后设置它的 item_count。下一步， 遍历数组中的每一个元素。使用你在类 BaseArray 中得到的每个元素的元类型，你就可以把数组中的元素进行分类（内建类型、指针或者对象）。对于数组中的元素，你要从数据库中得到它的地址和字符串的描述，然后填充这些元素的值。

7.2.5 例 子

我这里介绍给大家的例子不可能把实际发生的问题完全解决清楚，它最主要的意义还是在概念的验证上，而且和本精粹无关的代码也已经被精简到了最少。如果要用这个例子，你需要先安装好 PostgreSQLserver。光盘中有个 PostgreSQL，如果你没有 PostgreSQLserver，就可以使用那个来代替。安装时请使用默认的选项，当要建立数据库的时候，超级用户的名称可使用“postgres”，密码用“gem”。把这些都设置好了以后，例子就可以正常运行了。

直接运行程序看不到太多的东西，不过如果你看不到错误，就说明例子运行正常，并且功能都正确。要想知道数据库系统中的运作情况，你可以调试和跟踪代码。你可以从主函数单步调试入手，了解到对象如何建立、插入记录，然后从数据库中查询到的。在例子中有一些测试，用于对象从数据库中获取到以后验证数据的正确性。

使用 pgAdmin，你就可以看到表的建立以及对象在数据库中保存的情况。

7.2.6 问题和将来的改进

世上无完物，这里描述的系统也离完美很遥远，我们还可以加入许多东西来提高这个系统。

在 C++对象中，你可以做的一些事情在当前这个系统里面还不支持。例如，把一个指针保存成字符串形式，或者指向整型的指针，又或者其他数据库内建的类型。这些问题都可以解决，但是都超出了本精粹的范围。

目前，这个系统支持将指向对象的指针作为类的属性保存起来（例子里面有）；但是，如果这个指针指向的是某个属性就不行了。这个问题可以通过分两步走的方法来解决，第一步就是所有的属性地址都要在处理以前保存到表中。那时内存管理还没有被触及到。想象一下，你有一个指针指向一个对象，这个对象是保存在数据库的实例表中的，但是你的程序早就把这个对象删除了，那你不就有一个空悬的指针了吗？所以还是有必要使用 smart point 的，这也是解决上面问题的第二步。

使用数据库的优势就是你可以用判断语句来执行查询。在这个例子中，数据库管理器支持简单的过滤器，过滤器可以在你要取回的类的属性上直接产生效果（例如，“a=5”）。这个系统还是有许多地方可以提高的。你可以加入更多复杂的过滤器，或者支持更多的过滤方法

来让这个系统更容易被读懂，这些都是在 SQL 语句内部转换的。

当你的数据库是中央数据库，有许多用户在上面的时候，一个重大的挑战就是让所有的用户在本地查看时都要尽快和数据库中的内容保持同步。你可以在系统中加入一个提醒机制，提醒数据库管理时把用户 X 的修改及时反应到用户 Y 的界面上。

7.2.7　结　论

本精粹描述了使用一个元数据系统把 C++的对象保存到数据库中的方法。这个机制是不需要介入的，也就是说在存储时你不需要对你的类和对象做任何修改。这种方法有好几个优点：你可以把许多小的对象保存到数据库里，并且通过查询可以快速取回它们，而不是在硬盘上的许多文件里去寻找它们。因为数据库可以有许多用户同时访问，所以可能同一时间有许多用户在访问同样的数据，或者增加新的对象，又或者修改对象。数据库拥有支持并行访问所需要的一切机制，这些功能在共享文件或者其他工具里很难实现，特别是应对那种大量的、很小的数据时。当然，这个系统也允许 C++程序员在数据库里保存和获取对象，这个操作也不需要对 SQL 了解什么，因为所有的操作细节都被隐含在数据库中。

7.2.8　参考文献

[Abrahams06] Abrahams, D., and Gurtovoy, A. *C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond*, Addison-Wesley Profes- sional, 2004.

[Postgresql06] The PostgreSQL Global Development Group. “PostgreSQL 8.2.4 Documentation.”

7.3 数据端口

Martin Linklater
mslinklater@mac.com

随着游戏项目越来越大，游戏编程变得越来越复杂，也越来越难于管理。因为游戏越来越大，所以程序员就要处理日益增长的复杂度。有两种方法可以管理这种复杂度——你要么工作得更艰苦，时间更长；要么创建一个更好的系统来管理这个复杂度问题。如果我是你，我会选择第二种方法。复杂性一方面体现在通过代码来管理数据如何在各种系统传递中。代码模块通过把数据传来传去进行通信，还有就是把其中的数据部分暴露出来给别的模块。这种数据需要通过一种格式来保存，这种和格式相关的代码模块都被别的模块和程序员清晰理解。这就需要各个模块之间分享知识，这也增加了运行时和编译时的互相依赖，而更多的依赖就代表更多的复杂结构和更长的编译时间。数据端口的作用就是用来管理这种复杂性，并减少编译时的依赖性，让运行时的表现更柔灵活，增加数据驱动性。

控制代码模块之间的通信有两种基础的方法——用代码来写，在源代码中用指针来明确指定；或者用数据驱动系统，通过载入和分析外部的关系定义文件来定义数据之间的关系。直接编写代码的方式有两个主要的劣势——一个就是你必须重新编译和链接你的代码，因为你要修改数据的链接，其次在执行的时候必须大小写准确无误，所以在运行时修改和扩展这个系统或者以后的发布中都会引发问题。数据端口就是一个可以帮助你的系统，你可以通过这个系统在你的程序中建立更多动态的、数据驱动的流。

7.3.1 概述

从概念上来说，数据端口是很简单的。一个数据端口其实就是一段数据，这段数据有一个全局唯一的标记。这个数据可以是一个结构、一个类，或者一个简单的 C++数据类型。一旦建立了它们，数据端口就会把它们的标记登记在管理类中。在代码中用到的时候，你就可以通过建立数据端口的指针，并且要求管理类将其绑定到那段你要的数据上，然后通过这个数据端口的指针来访问数据。这个实现中有许多微妙的技巧，不过你需要想象的最基本模式就是数据结构和指针的关系。

数据端口

数据端口其实就是一个模板的包装，在程序员眼里就是一些数据而已。数据端口只有两个简单的基本方法，如下：

```
void Register( std::string ID );
```

数据端口一旦建立就需要注册自己。注册以后，数据就成为公共的了，而且对所有代码的查询都有效。

```
void DeRegister( void );
```

调用 DeRegister 来把数据端口从公共视野中移去。

数据端口指针

数据端口指针就是传统意义上的指针，不同之处在于把指针绑定到数据是由数据端口管理器来做的，而不是由源代码或者编译链接来绑定的。数据端口指针的两种方法如下：

```
Dataport<T>* Attach( std::string );
```

这个调用要求数据端口管理器把你需求的数据端口附加到数据端口指针上去，然后把指向数据端口的指针返回给用户。所以 C++代码看上去就会类似于：

```
pDataport = pDataportMgr->Attach( "Dataport ID" );
```

把数据端口的地址从指针上去除，需要调用如下代码：

```
Detach();
```

返回值告诉你是否成功。把数据端口地址附加到指针上以后，就可以通过访问 data 成员来访问数据了。

```
pDataport->data.<member variable>
```

数据端口管理器

数据端口管理器是隐藏在数据端口系统内部的核心组成。在数据端口管理器核心部分，它是一个仓库，是一个包含所有已注册数据端口的链表的存储系统。数据端口管理器还对数据端口指针进行计数。数据端口管理器值得你好好思考一下，因为它内部的算法是否适合你的应用程序会直接影响到效率。

如果要迅速创建和删除数据端口并且很少绑定，那么就需要有一个创建和删除效率很好的实现方法，但并不一定在查找上拥有很好的效率。从另外一方面来讲，如果数据端口的创建和删除不那么频繁，而绑定指针是很频繁的，那就需要一个快速查找的算法。因为这个例子是用 C++语言来做特别示例的，所以它是架构在 STL 库上，并且使用 STL 中的<list>来作为内部存储机制的。我推荐使用这种方法来解决问题，除非你最后发现效率实在不行。STL 库的编写是以运行时的效率为首要目标的，如果不用别人已经写完、并且已经被测试过且被

证明很稳固的代码，那这就太不可思议了。

数据端口管理器是一个单件类，也就是说在运行的时候程序中总是只有这一个实例存在。这是因为数据端口在全局有一个唯一的标记，虽然我们可能并行运行多个数据端口，但是这样做一点好处也没有。

7.3.2 类型安全

我第一次制作数据端口系统时没有放入任何形式的类型检查。不过也没花多久，我就重构了代码，把类型检查加入进来，因为你完全可能把指针绑错类型。一旦出现类型错误，你是很难检查出来的。所以检查类型绝对是个好事情。包含模板函数 GetID<T>的 C++代码，你给它一个类，它就会返回一个唯一的 32 位数字来标识它。其实这个 32 位数字是一个指向静态类函数的指针。ID 在数据端口管理器中的作用就是分辨不同的类型，以避免冲突。

7.3.3 引用计数

无论什么时候，你只要和指针以及数据打交道，就可能会遇到某个指针本来是有效的，但是突然之间不知道什么原因就无效了的风险。这种情况很难跟踪，因为那些崩溃可能发生在数据无效很长一段时间以后。所以从理想状态来讲，在还有指针指向这堆数据时，你应该没有办法使这些数据无效。

数据端口采用的方法就是引用计数来避免这种情况的发生。虽然从功能上讲不是严格必须的，但是引用计数在调试时还是作用很大的。

数据端口模板定义了一个成员 m_refCount，这个成员被所有的数据端口所继承。

- 当一个数据端口被注册时，这个引用计数就被设置成 0。
- 当数据端口指针指向时，引用加 1。
- 当数据端口指针移开时，引用减 1。
- 当数据端口要从注册表中移开时，首先检查引用计数，如果不是 0，那么就不能移开；并且返回一个错误（kErrorNonZeroRefCount）。

数据端口引用计数在调试中作用非常大，但是在运行中的确占用了一些性能消耗。你也可以在最后发布的时候把引用计数删除。不过只要你在调试系统中保留引用计数，就会得到很大的帮助。

7.3.4 实践例子

我从 2000 年就开始使用数据端口了，从那时起我就觉得数据端口是我程序工具包中一个很有用的工具。下面我就介绍几个我以前用到过数据端口的例子。

摄像机系统

在游戏里的摄像机系统中加入一层抽象层是很有用的。我把摄像机系统抽象成三脚架和

摄像机。三脚架就是摄像机摆放的位置，你在场景里可以自由摆放。玩家控制对象可以有多个三脚架，而像调试时用的自由镜头也有个三脚架。所以这个三脚架就用数据端口来制作。实际控制渲染的摄像机有一个三脚架的数据端口指针作为成员变量。把摄像机移动到一个新的位置，摄像机三脚架数据端口指针只需要很简单地赋值给指针即可。

这个系统一旦建立好，团队里的人就能够很容易地建立新的三脚架，并且在运行的时候把摄像机架到这个三脚架上去。因为所有这些能够由我们可以读懂的文件来定义，所以代码就增加了非常大的柔性。一旦三脚架和摄像机系统被建立好，新建一些摄像机视角就不需要什么维护和额外的工作了。

舰船控制调试值

极速像限（Quantum Redshift）和反重力赛车（Wipeout Pure）中舰船的操控数据都保存在可读性很好的 XML 文件格式里。那些文件的文件名包括和操控数据相关的团队的名称。在程序启动时，这些文件就被读入，并且由文件名来定义这些数据端口的名称。然后，当船在游戏中建立的时候，它们就很容易地和操控数据绑定在一起。新团队被加入的时候，不需要增加多余的绑定数据来关联操控数据。这种简单的数据驱动方式对于程序员和策划来说都是一个好的方法。

广播位置信息

编写游戏逻辑的时候，疯子才会把数据分布在各个不同的类中。钻进游戏的类中，又要保持面向对象的封装，又要维护对象的访问权限，这本身就是一项棘手的工程任务。游戏对象的数据的普通访问可以被数据端口所封装，并且可以用一种很友好的方式暴露给游戏中的其他部分。相对于把你的类都记忆下来，并且一层层跟踪你的类的继承结构，最后找到数据，还不如简单地记下你要的数据的类型和 ID，然后交给数据端口管理器来帮你找到。好比游戏对象的位置信息就是可以放在数据端口里，然后让其他的系统可以更容易地来访问。

在以前，我还设置了游戏 HUD（抬头显示界面）的数据端口，所以这些 HUD 可以很容易地动态地被游戏中的各种不同对象来控制。只要你把数据端口的结构定义下来，并且采用一个容易辨识的 ID 系统，就可以很方便地控制你的数据了。

7.3.5　问　题

数据端口肯定不是那种可以让你的代码很容易地使用、并且给世界带来和平的银弹。很可惜，它们也有自己的弱点。

如果你在数据端口管理器中采用某种格式的哈希表，很可能就会在哈希表中产生冲突。你可以通过把数据端口的类型加载到你的存储库中来降低这种冲突，但是这样做就很可能引起程序的崩溃。在例子程序中，我根本就没用到哈希，但为了效率，你有可能会在最终发行版本中引入哈希。

数据端口调试起来也更麻烦。由于它们的自然性质，数据端口在数据绑定上采用了动态和自由的概念，这种情况下你会觉得调试很难。你可以通过引入更多的调试以及记录系统来

缓和这个问题，但要记住，这个系统调试起来很麻烦。

大量地使用数据端口系统会引入这个系统本质上的性能消耗。数据端口并不是一定要让你把你所有的指针都换成它的，你需要针对 CPU 的开支来找到一个灵活的平衡点。

我个人的建议就是通过数据端口来做一些普通的游戏元素访问，这些元素可能是要被全局访问的，而系统中的各个模块都需要方便快速地找到这些元素。不过你可以发现，数据端口并没有对你的帧数有任何影响。事实上，我从来没有看到数据端口的操作在最后的游戏代码中对效率产生过任何影响。

7.3.6 结 论

目前的这些内容还没有在数据端口系统中包括任何方式的访问控制——所有的数据端口都能够读写，并且在全局访问。喜欢用常量类型指针的人可能不喜欢这种自由度，还喜欢在数据端口的 API 中加入常量定义。到目前为止，我也没有这种需要，但如果有人能够加上一层定义来支持常态的 API，我也会很感激的。

7.4　支持你本地的美术家：为你的引擎增加 shader

Curtiss Murphy; Alion Science
and Technology
cmmurphy@alionscience.com

随着最近硬件的提高，作为一款画面有竞争力的游戏，shader 是必不可少的。目前市面上有许多书，当然也包括这本，充斥着几千种 shader 技术和最佳实践。你的团队也因此充满激情，拼命地催促你把这些耀眼的效果加入到游戏中。那我们现在应该怎么做呢？如何把 shader 加入到你的引擎中去呢？作为一个开发者来说，把 shader 直接编码到你的引擎是一种最容易的方式。不幸的是，这种方法会产生许多问题，比如代码、美术资源和 shader 参数的紧密耦合，以及最终的硬编码系统没有弹性和无法扩展等。那么，你打算怎么办？本精粹能够提供这方面的帮助。

下面是一个可以帮助你将 shader 加入到你引擎的数据驱动的设计。这个设计提供了一种技术，能把大多数 shader 的参数从你的角色逻辑中独立出来。它也提供对于简单参数的支持，例如浮点和整型。当然，对于更复杂的参数，例如贴图和智能类型（automatic oscillating values）等。本精粹的实现可以让美术家和策划方便地在 XML 中定义 shader 的参数，这其中只需要程序员很少的参与。这里还会介绍一个例子，这个例子显示了一个飞艇在三维空间里盘旋，并且有一个粉红的高光点的动画出现在飞艇的上面。这个例子不用写一行代码就可以整合进引擎中。本精粹全程介绍了如何制作一个基础的系统，在这个基础系统上，你可以搭建自己更复杂的系统。

7.4.1　shader 专用名词

对于 shader 开发的新手来说，有一些因素会困绕它们一段时间。首先就是 shader 的定义。原本，shader 的名称来源于它对像素的着色。现在有两种 shader——顶点 shader 能够操作顶点，几何体 shader 能够操作整个模型。

一个更好的术语是处理器。毕竟，shader 可以处理各种各样的数据，这些数据早就超过了着色的范围。但是，shader 这个词会在本精粹中作为标准来引用。

另一个容易混淆的地方就是 fragment shader 和 pixel shader。它们听上去应该是不一样的，但实际上是一样的。片段（fragment）这个术语意味

着 shader 只计算一个“有可能”的像素。也就是说，作为图形流水线一部分被计算的这个像素，可能不会成为最后的帧缓存的一部分。所以，举例来说，“逐像素光照”其实是“逐片段光照”。在理想情况下，所有的片段可以成为真实的像素——不存在过多绘制——那样你只需要一个术语就够了。但是，在实际操作中，对于 GPU 来说，计算多个片段比计算每个单独的像素来说会更有效率。

本精粹描述了一些通用的概念，这个概念可以在 OpenGL 和 DirectX 中使用。因为它们都是通过状态来驱动的，这样的机制和概念可以很方便地转移。状态涉及全部渲染流程，包括绑定 shader、渲染状态、绑定贴图等。网格（Mesh）表示任何对象，包括几何体、节点或者几何面片等可以在一个渲染状态绘制出来的东西。为了简单化，本精粹采用了 OpenGL 和 OpenGL Shading Language（GLSL）。例子程序采用的是 OpenSceneGraph，使用的开源引擎是 Delta3D。

7.4.2 程序、参数和管理器，哦我的老天！

本解决方案的主要类是 ShaderProgram、ShaderParameter 和 ShaderManager。本节介绍这 3 个类的存在意义。

ShaderProgram 类

ShaderProgram 类是本解决方案的核心，它相当于 OpenGL 中的 Program 或者 DirectX 中的 Effects。这个 Program 就是大多数人提到 shader 时所指的东西。Program 是编译好的顶点和片段着色器的可执行程序。ShaderProgram 类是建立在 OpenGL 程序之上的。Program 有顶点 shader、片段 shader 或者两者都有。但是 ShaderProgram 最主要的工作就是保存 shader 参数的列表。

ShaderParameter 类

ShaderParameter 类掌管一个简单的统一变量。每个参数值通过它的状态直接绑定在一个网格上，这也是一个程序和 shader 沟通的最主要的方法。参数可能是一个高光点的基色、一个光泽的贴图、一个模糊的权重、一个点的位置、粒子密度、Alpha 强度，或者是任何你传送给 shader 的值。这些参数都是一致的，因为它们对于每个顶点和像素都做同样的动作。大多数的参数都是浮点、整型或者三元向量，还有作为影响 shader 输出的二维贴图。这个结构当然也支持复杂的参数，例如基于时间变化的参数（在后面的章节介绍）。

ShaderManager 类

ShaderManager 类就是把整个结构控制在一起的类，它负责将 shader 的原型从 XML 定义文件中读取，分配或者收回状态参数，跟踪分配的 shader，还有管理编译好的程序的缓存。这个管理器就是你用来寻找 shader 程序并把它分配到游戏引擎中的网格上去的工具。这个类是基于单件设计模式的[Gamma95]。

为了把大家都放到一起，管理器管理程序，而程序管理参数。这 3 个类呈现为如图 7.4.1 所示的结构。

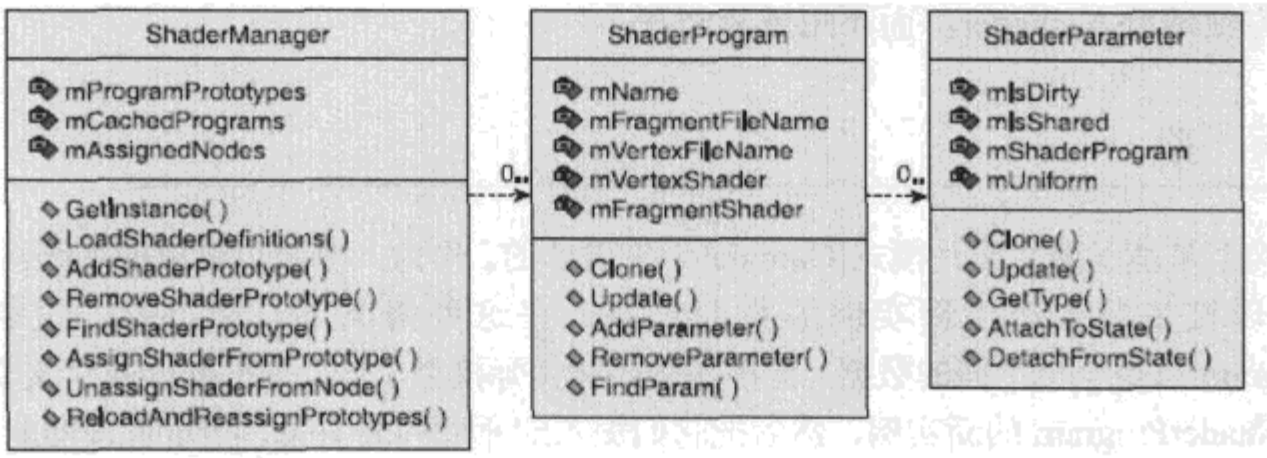


图 7.4.1　ShaderManager 类图

7.4.3　灵活性是关键

为什么总是要遇到这个麻烦？总之，如果你已经开始为你的角色写代码了，为什么不直接把 shader 的代码加上去呢？答案就是灵活性。现在的游戏都很大，个个都好像复杂的软件巨兽[Blow04]。制作这些游戏都需要一个巨大的团队，包括美术家、策划和程序员。事实上，美术增长的速度要快于程序。在这样的环境下，保证流水线尽量平滑就变得很重要了。美术家需要能够在没有程序的参与下测试模型和 shader 效果。shader 开发者需要能够在不重新编译或者修改代码的情况下修改 shader 的参数，甚至增加新的 shader。这个设计能够提供一个灵活的系统来满足这些需求，把美术和程序之间依赖性的关系给抹掉。

这个设计价格可以帮助美术家在实际引擎中实时看见自己的修改。在大多数工作室里，美术流水线上有许多工具是不在实际引擎中工作的。我们经常在一个工具中建立好模型，然后在另一个工具中画贴图，然后再在第三个工具中制作 shader。这就意味着美术在制作美术资源时，各个层面都是分割的，而不像玩家那样看到的是个整体。有时候会有某个工具可以提供 shader 的预览，而且效果和真实引擎中的效果差不多。但是，不考虑这个工具能够好到多少，问题是这个工具就不是在真实引擎中用真实的角色、灯光、武器、阴影和摄像机来工作的，它能做到的只是越来越接近而已。事实上就没有任何东西可以替代最终的混合所有效果的成像。达到最终真实度的目标激发了在这个体系里采用数据驱动的想法。因为 shader 很容易被定义和嵌入，所以美术家可以在引擎中立即测试自己的资源，这样他们可以基本不需要程序的帮助就能做到了。

测试模型

这个解决方案的另一个着重点就是在运行时动态加载 shader 的能力。shader 开发其实是一种艺术，也就是说，我们很可能尝试了上百次后才能够得到自己想要的，或许这个光照太强了、雾散太快了，或者光照的高亮实在太强了。幸运的是，这个结构的数据驱动特性使得它在任何时候都能很容易地重载 shader。ShaderManager 知道什么时候 shader 被创建，然后把它加入到活跃的程序列表中，然后访问所有的参数。ShaderManager 有它需要的所有东西来载入在 XML 中定义的 shader，然后系统化地替代目前程序中已经有的 shader，并且更新值。这个方式是由 ShaderManager 的 ReloadAndReassignShaderDefinitions()函数提供的。只要简单地按一下键盘上的按键，美术家就能够把新的 shader 载入，并且看到调整后的效果。不管你用的是什么 shader 系统，程序员应该尽力将测试系统做到引擎中来，然后还能够在运行时让

美术家很方便地载入 shader，而不用重启程序。

7.4.4 原 型

这个构架是基于原型设计模式[Gamma95]来设计的。作为一个可以刷新的系统，一个原型就是一个可以被复制去产生新实例的原型对象。在这种情况下，从 XML 文件中载入的 ShaderProgram 只是一个中间的数据，而不是运用到实际模型上的。管理器读取定义文件的时候，就会建立 ShaderProgram 的新实例，然后把它们加入到原型列表中去。接着管理器就会为每个实例读入 shader 的参数。一旦做好了，管理器就会重新编译 shader，把它加入到程序的缓存中去。

原型应该在载入地图或者程序开始的时候载入内存，这样在游戏中给新的对象加上 shader 也就不会给 CPU 增加什么困难。这个设计已经被进一步优化，以至于任何唯一的 shader 混合（顶点加片段）都只会被编译一次。为了实现这点，管理器在程序中会进行查找，找到那些会在原型之间能被共享使用的程序，并将它们保存在 mCachedPrograms 中。从本质上来讲，如果两个原型使用的是同一个顶点和片段的源代码，不同的只是参数的值，那么它们使用的都是同一段被缓存的程序。例如，基于车辆类型不同的高光贴图的不同，就是用的这个算法来实现的。

大家会注意到在 ShaderManager 里大多数的函数都会有“prototype”在名称里。这是因为管理器只会有两次是工作在一个实例上，而不是原型——第一次是把原型 shader 分配到一个网格上去，另外一次就是把 shader 从网格上剥离下来（即删除）。在第一种情况下（例如，AssignShaderFromPrototype()），管理器会把原型克隆，建立一个新的唯一 shader 实例。为了支持克隆这个功能，ShaderProgram 和 ShaderParameter 都会提供一个 Clone()函数。这个程序其实只是一个很轻量级的类而已，所以当它去执行克隆的时候，它也就是简单地得到几个引用，设置一些字符串而已，然后它就克隆所有的参数，并且把它们加入到参数图中去。对于每个新的参数，程序会调用 AttachToState()。这个函数的作用就是把参数绑定到网格上。最后，管理器把新的程序实例加入到叫做 mAssignedNodes 的列表中去。

第二种情况就更简单了。UnassignShaderFromNode()函数通过调用 DetachFromState()保证每个参数从网格上去除，然后把 shader 实例从活跃的列表上移除。因为在执行的时候我们使用的是智能指针，所以所有的对象都可以正确地被清除。整个分配和去除 shader 的过程都是在同一个程序实例里发生的，这个程序的参数也在指定的网格的状态里。程序是预编译的，并且是作为一个原型来被使用的，但还只是一个实例。

状态集和场景图

如果引擎能够支持场景图和状态集，这个结构还有一些额外的功能。场景图就是一个用结构化的方法来保存场景里的网格的做法。状态集就是一种机制，这种机制允许每个网格有自己的状态值，还能在绘制时允许这些数值进行更换。当拥有了这两种功能以后，你就可以得到一个结构化的网格系统，这个系统能够管理它们自己的状态。这个结构模型通常可以允许状态值的传递，从父结构传到子结构。也就是说，每个子节点的网格不仅可以设置自己的状态，而且还能从父辈网格那里继承到其他的值。所有这些值，包括网格节点自己的和从父亲节点继承得到的，组成网格的有效状态。

在绘制阶段，shader 程序是由编译了的 shader 代码以及相关的参数组成的。两者之间的

区别和操作系统使用的代码段以及数据段之间的差别很像。标准来讲，在一个基于状态的场景图系统中，shader 程序和参数是作为独立的状态变量来跟踪的。用另一句话来说，你可以将一个 shader 程序指定到网格上去，不论你是否指定参数，反之亦然。

通过本精粹，你可以得到一个更好的分级系统来允许通用的、高级别的 shader 程序来把网格从顶端分解到底端，并得到任何一个没有自己 shader 的子节点。这种方法可以用来定义不同的默认设置，以配置诸如光照模型等效果。你也可以在不知情的情况下定义“全局”shader 效果，这样做也不必知道哪个程序会最终用到这个效果。

例如，你可以为 HDR 的光照编辑器设置一个全局的值、自定义的雾的变量、定义开始和结束的值来让使用者微调，或者水的模糊效果和鱼眼效果。因为这些值是通过场景图来自顶向下分析的，所以它们会成为每个网格状态的值的一部分。这样做也让一些普通的全局渲染效果的参数调整变得更简单，管理 shader 参数也方便许多。你会注意到通过目前的设计，使用在 ShaderParameter 里的 mIsShared 标志，你可以很方便地实现一些小的功能（在后面进行讨论）。

7.4.5　shader 参数

在简单浏览这个构架以后，我们可以进入下一个环节，仔细观察那些 shader 的参数。不过在此之前，你需要先了解一下 shader 获取数据的方法。通常来说，传送到顶点 shader 或者片段 shader 的数据有 3 种。在 OpenGL 中，这些参数都有一致性、属性和变化的情况存在。

- 一致性参数（uniform parameters），是在整个几何体上保持不变的参数。这些参数在同一帧中基本不会改变，不过通常来说，它们从头到尾不改变的情况也很多。
- 属性（attributes），和统一参数一样并不经常改变值，但不同的是它们属于某一个顶点。也就是说，每个顶点的 shader 属性可以不尽相同。顶点属性常常被用来传递顶点法线、顶点颜色和切线向量。
- 多变参数（varying parameters），在顶点 shader 里被计算出来，然后传递给片段 shader。GPU 通过 3 个顶点传过来的数值来插值，并计算出这 3 个顶点组成的三角面的表面 shader 值。

一致性

ShaderParameter 类应该支持这些参数其中的哪几个类型呢？因为多变的参数是完全在顶点和片段 shader 里定义并由 GPU 来产生的，很明显多变的参数是不在其中的。那还剩下两种类型：一致性的参数和属性。让我们先来看看属性。通过定义，一个属性参数必须为每个顶点来设置。因为每个顶点都需要有自己的属性值，而且美术也会使用 3D 建模工具来设置顶点的一些属性，例如顶点法线和顶点色。另一方面，一些属性也可能是由引擎计算得来的，例如正切和反正切向量。在这两种情况下，属性参数都是美术流水线必须的元素，组里所有的人都必须认同这些参数，并把它们加入到美术工具和引擎里去。所以，属性参数也不是。也就是说，现在只有一致性参数留下了。

不过幸运的是，你也希望能够控制这些一致性参数。这些参数通常会用来设置灯光、雾的情况、裁剪区域等。还有，这些一致性参数也是传递用户定义数据给 shader 的最好方法。对于传递高光贴图、细节贴图和凹凸贴图来说，一致性参数是一条最好的路径。它们是控制

视觉通道的深度等基于时间的效果的最好控制方法。所以，我们的目标就是定义一些一致性的值，让美术家能够很好地进行控制。

类型相配吗?

shader 语言支持许多类型的参数，包括整型、浮点型、贴图和不同大小的向量。这也就是说你需要支持多种参数类型，以后你也会需要提供一个机制来支持更复杂的类型。总之，我们的目标就是把对程序员参与的需求减到最低，所以说设计的时候就把支持的参数考虑好是一个不错的方法。很明显，ShaderParameter 不可能一次性就包含所有这些属性，所以它必须是一个基础类。每个特指的参数类型就是它下面的子类。图 7.4.2 中显示了类的继承关系。

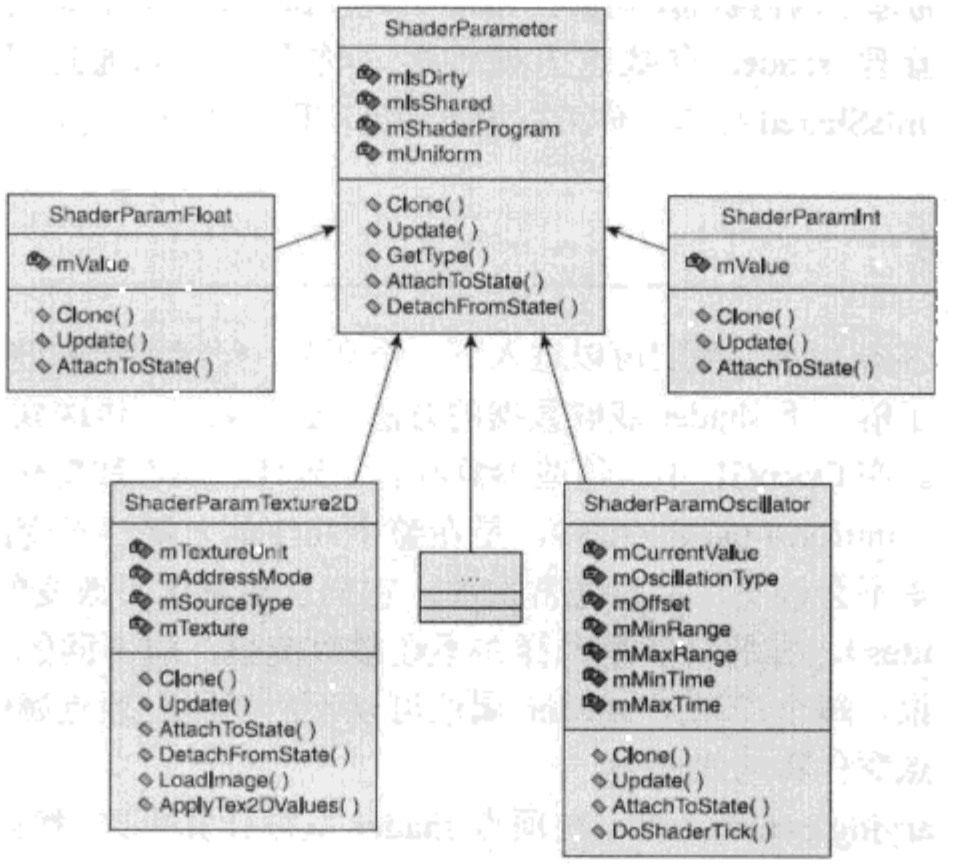


图 7.4.2 ShaderParameter 类图

基础类提供基础数据成员，例如实际的一致性变量、父类程序，还有就是目前这个 shader 是否是正在被使用或者已经定义了。基础类暴露了每个参数类型都需要的行为，例如克隆自己的能力和把自己附加到网格上的能力。每个子类控制把自己绑到状态上，以及管理自己的类型。

图 7.4.2 显示了一些简单的类型，例如 ShaderParamFloat 和 ShaderParamInt。图 7.4.2 也显示了一些较大型的数据，例如 ShaderParamTexture2D，这个数据是从文件或者缓存里把一张图片载入进来，然后绑定到贴图数据类型上去的，以及 ShaderParamOscillator 这个比较复杂的数据类型。这个不定的参数在某些时候会从最小和最大值里循环取一个值出来作为当前值。默认的情况就是从 0～1 中取值，然后再回到 0，这个循环是 2s 一次。美术家可以调整最大最小值的范围、时间频率、取值的间隔以及取值的自由度。你当然可以很简单地把这个系统扩展一下，加入自定义类型。

克隆

为了支持原型设计模板，每个参数类型都需要能够克隆自己。如整型和浮点型，就是简单地建立一个新的实体，然后把值赋上去就可以了。

比较大型的数据需要特别小心，尤其是在做共享引用和缓存机制时。我做的 ShaderParam-Texture2D 一个早期版本里，就在管理实例引用时出问题了，重复的贴图数据一下子占用了 500MB 的空间，把整个系统都给毁了。

克隆的过程产生一个很有意思的标志位，这个标志位控制共享参数的行为。就在参数实例被创建以前，Clone()检查了 mIsShared 成员，为的就是知道这个参数是不是在克隆以前已经被共享。一个被共享了的参数实际上就像一个全局的值一样为所有的 shader 原型进行工作。所以如果 mIsShared 的值是 false，函数就会像你所期望的那样被克隆，并且赋值。但如果是 true，那么原型参数实例就会直接返回。返回的值就会被加载到 shader 的程序中。结果就是这两个程序都会有一个完全一样的参数。当然，这也会出现奇怪的结果，克隆过程也有这样的功能，多个对象对一个参数进行响应。例如在这种情况下，多个对象能够在同一种方法下随机取值，并允许所有的机车实例使用同一个细节贴图。

7.4.6　例子——飞艇目标

为了检验我们的结构是如何工作的，让我们用飞艇来做一个简单的实验。在这个例子里，美术家希望飞艇能够在空中盘旋，并在三维空间里的各个方向都有些轻微的抖动，还希望给飞艇加上一个动态的旋转效果，好让它更凸显出来。为了达到这个目的，美术家需要建立飞艇的模型，并且定义如下 shader。

飞艇顶点 shader

顶点 shader 极其简单。为了达到盘旋的效果，我们需要建立 ShaderParamOscillator 的 3 个实例，然后它就可以处理 3 个轴向的值和移动了，每个点都按照正弦曲线运动，如下所示：

```
uniform float MoveXDilation;
uniform float MoveYDilation;
uniform float MoveZDilation;

// 顶点-为了“盘旋”和“高光”制作的简单的 shader
// 为了简单点，光照被去除了
void main()
{
  gl_TexCoord[0] = gl_TextureMatrix[0] * gl_MultiTexCoord0;

  gl_Vertex.x += 1.5 * sin(3.14159 * MoveXDilation);
  gl_Vertex.y += 1.5 * sin(3.14159 * MoveYDilation);
  gl_Vertex.z += 2.0 * sin(3.14159 * MoveZDilation);

  gl_Position = ftransform();
}
```

然后我们来看看片段 shader。

飞艇的片段 shader

片段 shader 稍微有一点复杂，让我们先看看代码：

```
uniform sampler2D DiffuseTexture;
uniform sampler2D HighlightTexture;
uniform float TimeDilation;

// 片段-提供了从细节贴图混合的来的高光
void main()
{
  float whackyOffset = sqrt(abs(TimeDilation - 0.5) + 1.0);
  float x = gl_TexCoord[0].x;
  float y = gl_TexCoord[0].y;

  //查找颜色
  vec2 lookup1 = vec2(x + TimeDilation, y + TimeDilation+.25);
  vec2 lookup2 = vec2(x - whackyOffset, y + whackyOffset);
  vec2 lookup3 = vec2(x - (TimeDilation*2.0), y + TimeDilation);
  vec4 color1 = texture2D(HighlightTexture, lookup1);
  vec4 color2 = texture2D(HighlightTexture, lookup2);
  vec4 color3 = texture2D(HighlightTexture, lookup3);

  // 现在将颜色混合，形成高光
  vec4 highlightColor;
  highlightColor.a = 1.0;
  highlightColor.r = color1.r*0.6 + color2.r*0.3 + color3.r*0.3;
  highlightColor.g = color1.g*0.2 + color2.g*0.7 + color3.g*0.2;
  highlightColor.b = color1.b*0.2 + color2.b*0.2 + color3.b*0.8;

  // 最后，把原来的颜色和高光的颜色混合起来
  vec4 diffuseColor = texture2D(diffuseTexture, gl_TexCoord[0].st);
  gl_FragColor = (0.2 * diffuseColor) + (0.8 * highlightColor);
}
```

这个处理器获得一个浮点的参数和两个贴图的参数（ShaderParamOscillator 和 ShaderParam Texture2D）。为了建立旋转的高亮效果，处理器在细节贴图里面做了 3 个独立的查找，每个查找都使用变量 TimeDilation 的一个计算结果。然后处理器使用查找到的值来计算一个最终的高亮颜色，并用那个颜色和原始漫反射贴图做混合。为了把新的 shader 加入到引擎中去，美术家还需要把下面的 XML 定义加入进去。

```
<shader name="Green">
  <source type="Vertex">Shaders/green_vert.glsl</source>
  <source type="Fragment">Shaders/green_frag.glsl</source>
  <parameter name="diffuseTexture">
    <texture2D textureUnit="0">
       <source type="Auto"/>
    </texture2D>
  </parameter>
  <parameter name="TimeDilation">
    <oscillator cycletimemin="2.0" cycletimemax="4.0"/>
  </parameter>
  <parameter name="MoveXDilation">
    <oscillator cycletimemin="5.0" cycletimemax="8.0"/>
  </parameter>
```

```
    <parameter name="MoveYDilation">
      <oscillator cycletimemin="5.0" cycletimemax="8.0"/>
    </parameter>
    <parameter name="MoveZDilation">
      <oscillator cycletimemin="5.0" cycletimemax="8.0"/>
    </parameter>
    <parameter name="HighlightTexture">
      <texture2D textureUnit="1">
         <source type="Image">Textures/green_detail.png</source>
         <wrap axis="S" mode="Repeat"/>
         <wrap axis="T" mode="Repeat"/>
      </texture2D>
    </parameter>
  </shader>
```

入口定义了叫做 Green 的 shader 程序，指定了顶点和片段 shader 的文件，这个定义还指定了两张贴图和 4 个动态浮点变量，在 0～1 动态取值。注意，动态浮点变量使用合理的默认值，所以美术只需要设置动态取值的时间。

整个效果没有添加一行程序代码。美术家建立了顶点和片段 shader，然后把它们加入到 XML 定义文件里去。美术家还能够在游戏里看到效果，并能够重复调整那些数字，一直达到自己的目标，在此期间不需要重启程序。程序代码在光盘中提供，完整的例子和源代码也可以在 Delta3D 中的某个部分见到（参见“结论”部分）。调色板 14 显示了几个极具戏剧性的不同结果的颜色变化，这也不需要重启程序。

7.4.7　高级技术

前面的部分介绍了一个基础的结构，这个结构可以直接加入到引擎里去。另外，我还想介绍几个比较高级的技术，它们也能够加入到这个系统里去，相信对你的开发会有帮助。

shader 组

shader 组允许几个互相关联的 shader 相互结合成一个块，并放到一个组中。这种做法允许对多个分离的 shader 进行定义，例如伤害模式、毁坏模式和正常模式。换句话来说，你可以为有目标或者没有目标的 shader 定义一个组，或者为系统里所有角色种族的白天和晚上的 shader 定义一个 shader 组。

为了实现这个方法，我们要在 ShaderManager 和 ShaderProgram 之间增加一个叫做 ShaderGroup 的类。这样做会在两方面修改原先的设计。第一个，管理器现在管理的是组，而不是某个单个的程序原型。第二，在查找单个 shader 前，你需要先找到它所在的组。注意，每个组都会把其中一个 shader 作为自己默认的效果；你总得在组中用一个 shader 吧。在完整的例子中，飞艇有两组 shader——一组是绿色高亮的，一组是普通的没有被选中的效果。下面的代码片段显示了拥有群的概念的 XML 定义文件的情况。

```
<shaderlist>

  <shadergroup name="Target Shaders">
```

```
    <shader name="Normal" default="yes">
      ...
    </shader>

    <shader name="Green" default="no">
      ...
    </shader>
  </shadergroup>

  <shadergroup name="Tank Shader">
    <shader name="Normal" default="yes">
      ...
    </shader>
  </shadergroup>

</shaderlist>
```

把 shader 绑定到角色和属性上

另一个高级特性可以用来调控角色和它的属性。这个特性可以允许美术家或者关卡设计师设置某个对象的 shader 组属性，从而控制这个角色的 shader。就像 XML 定义允许美术家用很简单的方法来设置 shader 一样，角色属性系统允许美术家很方便地给角色定义 shader 组，方法就是制作一个 API，对角色和角色的属性操作都很方便。为了添加乐趣，美术家使用这个属性来为地形增加新的 shader。想要完全了解角色和角色的属性，参考[Campbell06]。

下面的代码片段列出了所有用来改变应用于一个角色的 shader 必需的代码。当我们需要设置 shader 的时候，这个函数被 string 属性自动调用。shader 属性就是一个字符串，你可以在地图或者收到的角色网络信息里很方便地进行定义。为了把 shader 引用到网格上，程序员会调用管理器上这 3 个很重要的函数： FindShaderGroupPrototype()、GetDefaultShader()和 AssignShaderFromPrototype()。

注意，为了简单化，所有的检查都被去掉了，而且如果不需要支持 shader 组，这个函数调用会减少到两个。

```
// 把角色的属性设置到 shader 的群里
void GameActor::SetShaderGroup(const std::string &groupName)
{
  ShaderManager &sm = ShaderManager::GetInstance();

  // 确保旧的 shader 被清除了。有备无患
  sm.UnassignShaderFromNode(*GetOSGNode());

  // 得到 shader 群和默认的 shader
  const ShaderGroup *group = sm.FindShaderGroupPrototype(groupName);
  const Shader *defaultShader = shaderGroup->GetDefaultShader();

  // 按照 shader 原型来制作一个新的克隆的实例
  // 然后设置到网格的状态里去
  sm.AssignShaderFromPrototype(*defaultShader, *GetOSGNode());
}
```

7.4.8　后续工作

虽然这个版本的设计是完全可以工作的，但我们还是有很多地方可以提高。下面列出了一些你可能需要为自己的引擎考虑的特性，也可能会最终被加到我们用来演示的引擎 Delta3D（参看“结论”部分）上。

- 几何 shader（Geometry shaders）——本精粹没有支持几何 shader。但是，基于 shader 4.0 的特性，这个可以很容易添加上去。
- XML 编辑器——增加一个编辑器可以让美术家很方便地编辑他们的 XML 定义文件。这个有点像《游戏编程精粹 6》[Campbell06]中介绍的关卡编辑器。
- 生成的 shader 源代码——一些引擎支持在运行时生成 shader 源代码。这种设计可能会引起争论，大家会为到底是在运行时生成 shader 代码好还是从磁盘上直接读取好而争论不休。不过在运行时优化了的代码对美术家测试新单元总会有好处。
- 角色属性参数——当角色的属性（例如生命值或者速度）改变时，增加一个新的数据类型来自动更新网格的状态。
- 增强的缓存——为顶点和片段 shader 维护不同的缓存。

7.4.9　结　论

本精粹介绍了一个已经可以使用的 shader 架构，这个架构可以直接整合到你的引擎中去。它的数据驱动特征让在引擎外部对 shader 系统进行操作成为可能，也让美术家在游戏引擎里实时预览他们的美术工件成为可能。本精粹讨论了 3 个主要的类，ShaderManager、ShaderProgram 和 ShaderParameter，也介绍了如何使用原型设计模板来提供一个有很高效率的、灵活性很高的系统。本精粹解释了为什么一致性变量很重要，也介绍了如何支持多种不同的数据类型。最后，本精粹阐明了一个真实的例子，这个例子允许美术家旋转和为飞艇制作动画，这个过程中不需要任何程序开发人员的参与。

你可以到本书附带光盘里查找这个概念更多的例子。还有一个完整的、完全可以工作的例子在开源 Delta3D 项目中也可以看到。

7.4.10　参考文献

[Blow04] Blow, Jonathan. “Game Development: Harder Than You Think.”

[Campbell06] Campbell, Matt and Murphy, Curtiss. “Exposing Actor Properties Using Nonintrusive Proxies,” *Game Programming Gems 6*, edited by Michael Dickheiser, Charles River Media, 2006, pp. 383–392.

[Gamma95] Gamma, Erich, Helm, Richard, Johnson, Ralph, and Vlissides, John. *Design Patterns—Elements of Reusable Object Oriented Software*, Addison-Wesley, 1995, pp. 117–126.

7.5 与蟒共舞 用好 AST

邹光先

在任何 MMORPG 游戏中，NPC 与玩家之间都会有很多文本对话信息。加密这些信息会消耗 CPU，而传输它们则会占用带宽。

Python 是一门动态语言，在 MMORPG 开发中用得很多。通过 Python 的 compiler 包（package），开发者就能操控抽象语法树（Abstract Syntax Tree，AST），并在运行时生成并控制 Python 字节码。

本精粹将介绍如何通过操控 AST，从而实现用数字 ID 代替字符串，以节省传输带宽和运行时的开销。在 CD 中有基于本文思路的工具源码。

7.5.1 简 介

在计算机科学中，AST 表示抽象语法树（Abstract Syntax Tree），它在语法分析（parsing）阶段产生，并作为字节码生成器的输入，其内部节点表示操作符（operator），如加法（addition）或连接（concatenation），叶节点表示操作数。通过顺序访问 AST 树种的节点，就可以生成基于 AST 的字节码。

AST 中的每个节点都有特定的含义和信息，例如，是变量还是常量，如果是常量，他的值是什么？利用这些信息或改变这些信息，程序员就能控制字节码的生成，从而改变源代码行为。

7.5.2 背 景

在游戏中处理文本的标准方式是用转换表。每个字符串都对应一个 ID，当需要显示时，从表中通过 ID 查找出字符串。用 ID 而不是直接用字符串的好处是可以节省带宽和内存，同时无需更改代码，就能很方便地变化语言。

然而，在游戏开发过程中，要保持 ID 与字符串的对应关系既耗时也耗精力。此时如果能直接用字符串会更容易，效率也更高。但这就意味着需要以后替换成通过 ID 查找的方式，并在那时去跟进对应关系，这样就需要修改代码。显然，这很费时，也容易出错。

如果能有一种方法在现有的代码中找出所有的字符串，把它们放到数据库中，并跟进对应关系就很有用了。另外，只要能“理解”处理字符串的代码，也就很容易编辑代码并修改字符串。与把代码改成每次通过表间

接查询相比，直接修改字符串更简单。除了在玩家修改语言或者服务器更新时需要重新修改代码外，其他情况下就没有额外的间接查询成本，减少了代码的复杂度。

7.5.3　方　案

Python 代码中的字符串放在一对单引号（'和'）或者一对双引号（"和"）中。例如：

```
companyName = 'NetEase.Co'
projectName = "Tang Dynasty"
address = """
GuangZhou,
China
"""
```

一般来说，要从上例中取出字符串并不容易，得写个分析器来做，挺麻烦的。

此外，还有些情况下，直接用查询函数替换字符串还会出现问题，例如在错误的范围内，或字符串是编译期宏的一部分。

幸运的是，Python 已经有很好的机制使此事变得简单。在编译器包中，有以下 5 个函数。

```
compile( source, filename, mode, flags=None, dont_inherit=None )
compileFile( source )
parse( buf )
parseFile( path )
walk( ast, visitor[, verbose] )
```

compile 和 compileFile 都用于编译源代码，compileFile 还会产生 PYC 文件，而 compile 则返回目标代码（code object）。

parse/parseFile 则返回源代码对应的 AST。

walk 函数遍历 AST，对每一个节点调用 visitor 对象的相应函数。例如，当遇到 Const 节点时，就调用 visitConst 函数。通常对于节点类型是“Type”，如果 visitType 存在，则 visitType 函数就会被调用，否则就调用 ASTVisitor.default。这样，只要提供一个 visitor 对象给 walk 函数，就能取到 AST 中节点的信息。

为解决我们的问题，需要提供一个实现了 visitCallFunc 函数的 visitor 对象。本方案分两步：第一步，找出代码中的所有字符串常量，并为每一个字符串赋给一个 ID，把 ID 与字符串映射关系保存到一个叫“stringres.txt”的文件中；第二步，执行 walk 函数，用 ID 代替字符串，并产生 PYC 文件。

为便于理解 AST，本文也给了一个名为“astshow.py”的工具，用它可以生成 AST 的格式化结果。这里我讲讲当源码中的函数被调用时我们能得到什么信息。

例如，有一个文件叫“sample.py”，其内容为：

```
import game
game.msg2player( "hello" )
```

当 walk 函数遍历上述代码的 AST 时，传给 visitCallFunc 的节点将会是：

```
CallFunc(Getattr(Name('game'), 'msg2player'), [Const('hello')], None, None)
```

当子节点是"GetAttr(Name('game')), 'msg2player')"时，参数就将是"[Const['hello']]"。函数的名称则可以通过下面这个函数得到。

```
def getFunctionName( node ):
  if isinstance(node, compiler.ast.Name):
     return node.name
  elif isinstance(node, compiler.ast.Getattr):
     return getFunctionName(node.getChildNodes()[0])
                              + '.' + node.attrname
  else:
     return ''
```

在第一步中，visitCallFunc 的实现为：

```
import os
from compiler import ast, pycodegen
import utils

class Visitor:
  def visitCallFunc(self, node):
     func_name = utils.getFullName( node.node )
     if func_name in utils.helper.functions :
         for arg in node.args :
             if isinstance( arg, ast.Const ) :
                 if isinstance( arg.value, basestring ):
                     utils.helper.append( arg.value )

  # the rest is copied from pycodegen
  # and simply continues to walk the AST.
  pos = 0
  kw = 0
  self.visit(node.node)
  for arg in node.args:
      self.visit(arg)
      if isinstance(arg, ast.Keyword):
          kw = kw + 1
      else:
          pos = pos + 1
  if node.star_args is not None:
      self.visit(node.star_args)
  if node.dstar_args is not None:
      self.visit(node.dstar_args)
```

此函数中，检查参数，并将字符串常量参数通过' utils.helper.append '函数加到 ID 与字符串映射表中。

在第二步中，用 compile/compileFile 生成字节码，compile/compileFile 与 CodeGenerator 是强耦合，因此我们没法通过一个继承自 CodeGenerator 的类来操控结果，只能通过给 CodeGenerator

一个定制的函数来达到目的。visitCallFunc 的实现代码如下所示，当 arg.value 是字符串常量时，被替换成 ID。

```
import os
from compiler import ast, pycodegen
import utils

def visitCallFunc(self, node):
  func_name = utils.getFullName( node.node )

  if func_name in utils.helper.functions :
     for arg in node.args :
         if isinstance( arg, ast.Const ) :
             if isinstance( arg.value, basestring ):
                arg.value = utils.helper.get( arg.value )

  # the rest is copied from pycodegen
  # and simply continues to walk the AST.
  pos = 0
  kw = 0
  self.set_lineno(node)
  self.visit(node.node)
  for arg in node.args:
     self.visit(arg)
     if isinstance(arg, ast.Keyword):
         kw = kw + 1
     else:
         pos = pos + 1
  if node.star_args is not None:
     self.visit(node.star_args)
  if node.dstar_args is not None:
     self.visit(node.dstar_args)
  have_star = node.star_args is not None
  have_dstar = node.dstar_args is not None
  opcode = pycodegen.callfunc_opcode_info[have_star, have_dstar]
  self.emit(opcode, kw << 8 | pos)
```

更多细节可参考本书附带光盘中的完整代码。

7.5.4　结　论

本文利用 Python 的编译包便能访问并修改 AST，基于此，就能给出一个优雅的方案来创建字符串映射表而无需人工帮助。此外，脚本程序员也不用担心以后如何转换，此过程是透明的，并能被无缝集成到项目中。

7.5.5　参考文献

[Python] Python language Website.

关于本书附带光盘

关于《游戏编程精粹 7》的附带光盘

本书附带的光盘包括源代码、可执行示例、库文件、图片以及文章，这些都是用以演示或者补充说明本书中的精华文献。要对本书进行完全掌握，需要读者对光盘认真学习。我们尽最大努力保证本书所提供的源代码没有错误，而且可以通过编译，可执行文件能不出错地运行，并且所有图片和文字都能随意浏览。

内容

为了便于查找，本书附带光盘中的所有材料都放在书中对应章节和精华文献的文件夹中。为了使读者使用方便，我们提供了一个 Windows 系统的自动运行程序来帮你定位每一个文件夹，但你并不是一定要通过这个自动运行程序才能浏览此光盘的内容。每一个文件夹里的源代码都被验证通过，可以在 Microsoft Visual Studio C++ 7.0 里编译，同时我们还提供 Visual Studio 2005 格式的解决方案文件和项目文件。在很多例子里，我们还提供了预编译二进制文件。如果情况允许，我们也提供了额外的一些库文件，但是有几个库文件必须读者自己去获取。

Windows 系统需求

此光盘要求 Windows 2000、XP 或者 Vista 操作系统。补充的文章需要一个能够显示 Microsoft Word 或者 PDF 文件的文档查看器。使用到或者演示图形技术的示例程序需要支持 DirectX 9 的 3D 显卡。