游戏编程精粹 5

GAME PROGRAMMING Gems 5

[美] Kim Pallister 编
孟宪武 等 译

人民邮电出版社
POSTS & TELECOM PRESS

内容提要

本书是著名技术丛书“游戏编程精粹”系列书的第 5 卷，由全球 60 多位优秀游戏开发精英撰写的文章汇集而成。书中每篇文章都针对游戏编辑中的某一特定问题给出解决方案，并提供实用算法和源代码。全书由 7 章组成，包括通用编程、数学、人工智能、物理、图形图像、音频以及网络和多人游戏，覆盖了当前游戏开发中的所有关键技术领域。本书附光盘一张，提供书中所有的源程序和演示程序。

本书适合游戏开发专业人员阅读。专家级开发人员可以立刻应用书中介绍的技巧，而初中级程序员通过阅读本书将增强其技能和知识。本书是游戏程序员必备的参考资料。

关于随书光盘

随书光盘中包含了与本书文章有关的全部源程序代码和演示程序。我们竭尽所能，尽量确保这些源代码没有 bug，而且是可以编译的。可以浏览 *http://www.gameprogramminggems.com/* 这个网站，从中获得勘误表和相关的更新信息。

光盘内容

code 目录：随书光盘中的源代码和演示程序，按照章节名称、文章标题和作者名字，以子目录的形式编排整理。

书中相关的源代码程序清单也收录在光盘中。根据每个作者的喜好，不是所有的文章都提供了完整的 Demo 程序。Windows 演示程序都是用 Microsoft Visual C++ 6.0（工程文件为*.dsw）或者 Microsoft Visual C++ 7.0（工程文件为*.sln）编译的。

GLUT 目录：在这个目录下，会找到 GLUT v3.7.6 distribution for Windows。和 Windows 有关的信息，请访问 Nate Robins 的网页 *http://www.xmission.com/~nate/glut.html*。

J2SE 目录：对于那些用 Java 语言编写的示范代码，我们在此提供了 Sun 公司的 Java 2 Platform Standard Edition（J2SE），1.4.2 版本。其中包括 Windows 版本和 Linux 版本，都是自解压打包文件。

DirectX：如果使用的是 Windows 系统，非常有可能会用到 DirectX API。为了方便起见，我们在这里提供了 DirectX SDK 9.0。

系统需求

Windows 系统

建议使用 Intel Pentium 系列、AMD Athlon 或者更新的 CPU 产品。应使用 Windows XP（64MB RAM），或者 Windows 2000（128MB RAM），或者更高版本的操作系统。有些示范程序的运行要求有 3D 图形卡。还需要 DirectX 9 和 GLUT 3.7，或者更高版本的系统。

Linux 系统

建议使用 Intel Pentium 系列、AMD Athlon 或者更新的 CPU 产品。需要 Linux kernel 2.4x 或者更新的版本。推荐使用至少 64 MB RAM。有些示范程序的运行要求有 3D 图形卡。还需要 Xfree86 4.0、GLUT 3.7、OpenGL 驱动、glibc 2.1，或者相应更新的版本。可以使用 Mesa 代替 3D 硬件支持。

PDG

序

Mark DeLoura
madsax@satori.org

欢迎来到《游戏编程精粹 5》的精彩世界。这一系列书旨在：竭尽所能地搜集更多业界专家的智慧结晶，将它们集结成书，帮助读者迎接游戏编程的各种挑战。Kim Pallister 和他的编辑同仁们为此做了杰出的工作，他们发掘了这些宝贵的精粹文章，并精心打磨，让读者可以充分享受阅读的乐趣。衷心希望读者可以在本书中找到“宝藏”，并对下一个游戏编程项目有所帮助。

从本系列书的第 1 卷发行至今，游戏引擎的编写工作并没有变得更加简单。而且，游戏和游戏开发团队的规模也没有任何萎缩的迹象。如今，人们常常会听说某个游戏项目的开发用了 3~5 年的时间，或某个项目的开发团队多达 200 人。游戏的开发周期会这样不断地增长下去吗？这些团队的规模有可能变得更大吗？大家当然不希望如此。但这种趋势似乎非常明显。不论如何，《游戏编程精粹》系列书可看成是很多“正义力量”的一员，这些“正义力量”站在等式的另一边。我们会尽可能多地搜集整理业界专家的智慧，包括完整的代码，并把它们奉献给读者，从而使大家至少有机会加速游戏引擎的开发进程。

游戏引擎的开发

好在还有很多乐意帮助大家进行游戏引擎开发的中间件厂商。但是，和业界其他领域一样，这些公司最近也已经开始进行整合。有些公司合并为一，为业界提供更为完整的游戏引擎产品包。还有一些公司被开发商或发行公司收购，这样这些开发商和发行公司就可以自己关起门来享用这些先进的技术，同时也减小了大家可以选择的产品范围。虽然市场上还是有一些产品包是可以掏钱来购买的，但不管怎么说，使用一款属于竞争对手的游戏引擎至少不是最佳选择，对此一定要谨慎考虑。

但是，由于开发一款游戏引擎所需的费用不断增加，所以对于每一个新的游戏项目，是否真的有必要从头开始去开发相应的游戏引擎呢？这样做是否有意义呢？如果放在过去，答案是肯定的。但是现在，由于游戏引擎的开发已经变成了一项如此复杂和昂贵的工作，以至于很多工作室都在积极地开发能够增长基础代码生命周期的策略。现在，很多开发商至少会

在某个平台的生命周期之内通过不断地对所创建的 sku 的游戏引擎进行步进式的优化来延长该引擎的生命周期。一些发行公司还开始高瞻远瞩，考虑核心技术的开发。还有一些公司甚至鼓励与之合作的工作室一起协作来开发通用的技术和工具集。

所以，至少从技术的角度来讲，我们还是可以提供一些策略来降低业界的开发费用，帮助大家发展过渡到一个新的、更为复杂的平台上。

勇往直前

对于下一代游戏机和 PC 产品，让很多人感到害怕的是大量需要创作的可以充分发挥这些平台优势的美术内容。据某游戏工作室的估算，创建下一代的游戏角色模型要比以往多花费大概 6 倍的时间，这主要是由于对高模网格和多纹理图层的需求。当庞大的美术团队已经占用大量的资金并拉长项目周期的时候，大家应该如何应对呢？

显然，我们并不准备在这里解决这些问题，但是随着 PC 和游戏机平台上的游戏开发工作的复杂度的不断增加，随着这些游戏产品开发费用的不断增长以及开发团队规模的不断扩张，该产业所受的影响也呈现出多种形式。其中一个最明显的影响就是游戏设计上的风险更小了。一款不成功的、开发费用昂贵的游戏可以把一些小型发行公司搞跨，受这种影响却会有更多缺乏创新和创意的模仿作品涌现出来。如果说我们确实看到了某些创新，那也是“局部”上的创新，即只是在游戏类型上的简单的发展，而不是根本上的全新的游戏概念。当然，从经济效益上讲，这样做绝对是有意义的。但是，越把游戏当成一个产业，就越容易面临令玩家厌烦的风险。还有很多其他的娱乐方式在和游戏争夺玩家的眼球和钱包。

工作室的另一个选择就是在小型平台上开发更多大胆的游戏，其好处是开发费用低。随着更多掌上游戏设备、可玩游戏的手机和 PDA 产品的出现，移动游戏平台的时代已经到来，游戏公司可以越来越容易地在一个风险更低的框架下去尝试他们在游戏性方面的创新。而且，在那些成功的游戏产品之中，一些革命性的创新一定会找到自己通往“大成本、大制作”平台的成功之路。

游戏开发人员的教育培训

令该产业获得成长的最有希望的迹象就是，过去几年中社会对那些开设游戏开发课程的学校的需求高涨。根据国际游戏开发者联合会（International Game Developers Association，*http://www.igda.org*）的数据，目前有超过 280 所学校提供与游戏相关的课程。这种在正式的游戏开发教育方面的增长所产生的结果，就是为业界带来了更多的、富有新鲜点子的年轻一代的开发人员。这些热情奔放的、受过专业教育的学生新人，不断地涌入业界。只要愿意倾听，他们就一定会带来很多新颖的、有创意的想法。当然，也必须承认，能够让一个发行商塌实地坐下来聆听，然后说服他把几百万美元都押宝在一个由刚从学校毕业的新雇员提出的未经验证的游戏概念上，这的确是一件很具有挑战性的事情。

结果就是我们最近已经开始看到的，出自那些小型、年轻的开发团队的独立游戏产品、游戏模式和休闲游戏的产品数量有了快速的增长。这些小型团队有能力去尝试很多有冒险性的想法。而且，无论最后成败与否，整个过程对所有参与的人员都是一个非常好的从业经历，

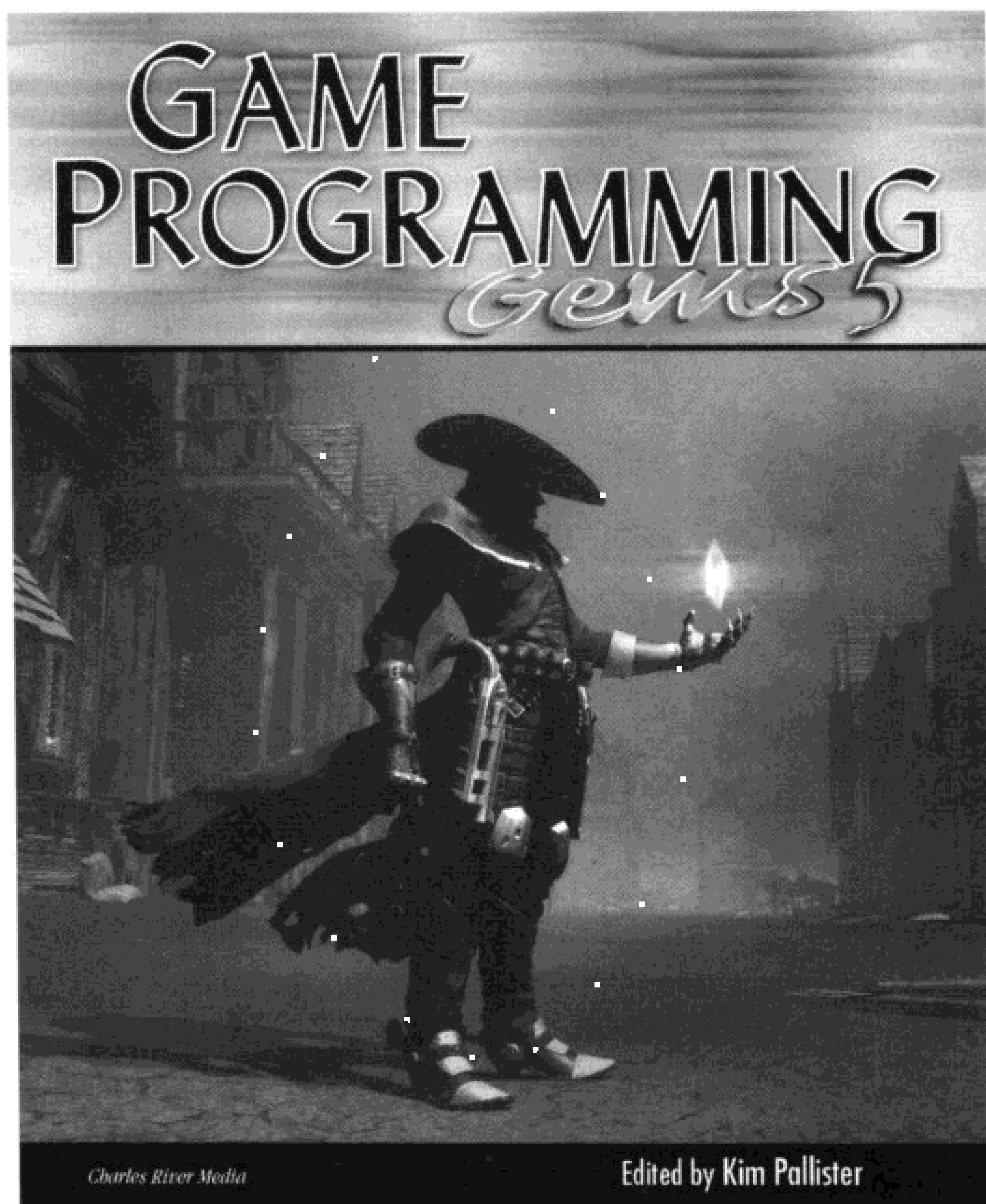
也是个人简历上非常优秀的素材。对那些有前瞻力的发行商而言，这些成功的项目则是一次能引起关注的机会。

社会对正规的游戏开发教育的需求的不断增长，也成为使独立游戏开发领域创新之火熊熊燃烧的助燃剂。很多发行商总是在那里简单地改造游戏概念，小心翼翼地推出自己的产品，但是如果有足够的支持，革命性的概念创新将会产生于那些独立游戏开发社团之中。

对专业的开发人员来说，这就意味着跳出现在的框框，和学生及发烧友们共同分享自己的专业经验更加重要了。而且和以往相比，现在有更多的机会让你可以这样做。通过和一个独立开发商的小团队一起攻关一个颇具风险的游戏策划，也许能丰富自己的游戏开发项目，找到很多令人兴奋的东西。

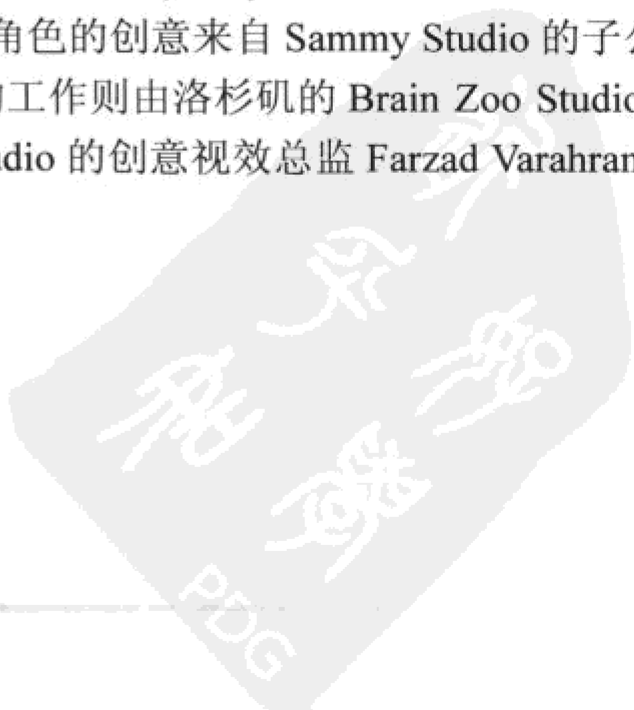
关于封面图片

© Sammy Studios



半吸血鬼枪手 Jericho Cross，在旧日的西部与不死的吸血鬼魔王展开对抗。Sammy Studio 公司的这款游戏《黑暗标靶》（*Darkwatch*）于 2005 年推出，面向的平台是 Sony Playstation® 2 和微软的 Xbox™。

其中，场景与角色的创意来自 Sammy Studio 的子公司 Carlsbad Studio，渲染和光照处理的工作则由洛杉矶的 Brain Zoo Studios 完成。本图的美术指导是 Sammy Studio 的创意视效总监 Farzad Varahramyan，背景美工是制作助理 Dan Kit。



前　言

Intel 公司，Kim Pallister
Kim.Pallister@intel.com

为什么我会是本书的编者呢？

这是一本关于游戏开发的书，而我却不是一个专业的游戏开发人员。在过去的十几年里，我一直处于这个产业的“边缘”。最近的七年，我在 Intel 公司；之前，我则在一家图形硬件供应商就职。在这个有着迎合年轻市场的传统，又吸引了近似人群的产业之中，我置身于长辫子年轻人和满头白发的怪脾气老头这两个角色之间。

怪脾气老头那一辈的人一定会记得，在这个产业最初的几十年间，我们看到的是从单人手工作坊式的开发向团队协作时代的过渡。在团队协作时期，人们彼此依赖他人的技能和努力来实现自己的创想。在过去的十年间，或者在此之前的几年中，我们已经过渡到了一个小组之间互相依赖的时代。程序员、美工和设计师共同组成开发团队。一个公司里的若干团队会互相借助他人的工具和资产，而公司和公司之间也会广泛地互相依赖。

公司间的关系并非这一行的新生事物，开发商和发行商之间的关系已经存在了很长一段时间。但是，过去的十年间，我们又看到了新的产业结构、依赖关系和沟通渠道。现在，开发商会从其他公司那里获得技术授权，并从中间件、硬件和平台供应商们那里获得如何充分利用新技术的知识。反过来，这些供应商也会从开发商那里搜集信息，来指引自己的技术开发。一些实体机构，像 CMP 游戏集团（《游戏开发者》杂志、游戏开发者大会）和国际游戏开发者联合会，促进了关于游戏策划、商业实践和发行质量的对话。越来越多的学术机构，以及一些出版社，如 Charles River Media（即《精粹》系列书的出版机构），正致力于整体从业人员的教育和信息传递。仔细审视这些关系链，我们看到的是一个生命体的循环系统——一个所有成功的产业所固有的产业链系统。

但是慢慢地，这个产业正从一个吝啬合作的状态，逐渐转变为一个开放、有序的简易化状态。我们与中间件和硬件厂商分享源代码，资助学术圈的人们展开相关的研究工作，并同时加强对话和出版。这个循环系统中的每一个成员，都在游戏这个媒介的成功以及该媒介创造者的成功中，拥有自己的既得利益。该循环系统也包括我在内，这就是我编辑本书的原因。

由此而往何处?

不远的将来已经足够清晰了。这个市场会继续增长，因为新的玩家不断涌现，已有的玩家变老了并继续玩着游戏，而且游戏本身的产品质量的提高反过来也会吸引原来并不热衷于游戏的用户。

我们这个产业所面临的其中一个最大的困难和最重要的拐点，是向多线程平台和多线程软件开发的转移。在撰写本序之际，台式 PC 产品已经在其处理器中展示了多线程技术，而真正的多核处理器也呼之欲出。在线游戏服务器端部分的开发已经采用了多处理器技术和分布式系统架构，次世代的游戏机产品也宣称会在短期内将多处理器平台带到消费者家中的客厅里。

向真正的多线程游戏引擎和平台过渡，看上去是个非常艰难的挑战。我们需要新的工具和新的技术，如果希望在过渡的同时还能够带来游戏的发展进化，那么我们甚至需要新的编程语言。我们还需要专注于可靠的软件工程——原理和实践。投资高达 7 位或 8 位数（或者 9 位数，只要我们敢说的话）且团队规模超大，这样的游戏项目，在纪律和严密性上都需要达到一个新的高度。

在这个过渡时期，知识的共享（个人与个人之间、公司与公司之间，在所有的层面上的共享）是非常关键的。希望在这个历程中，《游戏编程精粹》系列书能够扮演一个小小的角色。

路之远景

越过不久的将来往前看，视线变得模糊，但却令人兴奋。撰写本序之际，大型多人在线游戏正在亚洲（特别是韩国和中国）爆炸式地流行起来。未来的 4 年里，将出现几亿的家庭宽带用户。可以玩游戏的手机和移动设备会继续激增并彼此互连。电视和其他消费电子设备也会提供更丰富的交互特性。只要有交互性，那就有游戏可言！

业界的很多人曾经快速地推断出游戏与好莱坞电影之间的比较关系。通常这些比较都集中在各自产业的收益上，或者是制作过程的相似度上。其实，还有一个类似的可比产品。

在 1939 年的世界博览会上，人们见到了一个新的可以保证娱乐大众的技术——电视。但在当时，评论家们却满腹狐疑。《纽约时报》的一篇文章声称：“电视的问题在于人们必须端坐下来，并让自己的眼睛紧盯着屏幕。一般的美国家庭是没有时间来做这件事的。”这种言论现在听来是很可笑的，但在当时，即使是这个技术的支持者也想象不到这个新的娱乐媒介会对人类生活产生如此大的影响。

在几年前的一次游戏开发者大会（GDC）上，Chris Hecker 就曾经预言：游戏将是“21 世纪定义的娱乐媒介”。我再也找不到比这更好的表达方式了。我们才刚刚开始抓住交互式娱乐媒介的某些可能性。今天，我们尚可以把互动娱乐称为“游戏”，但是我们正在扩展这个词汇的定义和外延，我们仍要继续努力，去描绘其所有的含义。

现在就轮到大家来扮演某个角色，去定义这个媒介了。但愿我们能给大家提供一些工具，帮助人们完成这个任务。让我们好好利用这些工具，为这个媒介的未来尽一份力吧！

致谢

撰写本序的时候，我一直质疑着自己作为本书编者的身份。在本序收尾之际，我要说：如果没有许多人的直接或间接的贡献，我根本无法完成这项工作。我首先要感谢的就是该系列书的编辑 Mark DeLoura 和出版人 Jenifer Niles。非常感谢他们二人给了我这个机会，并用他们在《游戏编程精粹 1~4》的制作过程中学到的“书刊编撰精粹”指导我完成了整个编书历程。

虽然我为本书付出了大量的时间和精力，但是，如果没有那么多提供精彩文章的作者和 7 位编辑人员，我是不可能完成这项工作的。

通用编程：ATI 研究院，William E. Damon Ⅲ

数学：Terathon 娱乐公司，Eric Lengyel

人工智能：Northwestern 大学，Robin Hunicke

物理：Red Storm 娱乐公司，Michael Dickheiser

图形图像：ATI 研究院，Jason Mitchell

网络和多人游戏：Gigante 工作室，Shekhar Dhupelia

音频音效：Sony 计算机娱乐公司，Mark DeLoura

我们非常荣幸，能够在一个大家都鼓励并实践知识共享的行业里工作。

同样的感谢还要给 Pete Baker 和我在 Intel 公司的其他同事，是他们让我能够做得更好。当我固执地认为这一编撰工作不会影响到我的日常工作的时候，他们总是通情达理，让我放手去干。

最要感谢的，是我的妻子——Alisa，是她帮助我在整个编撰过程中保持始终如一的热情。同年，我们的双胞胎宝宝来到了这个世上。对她、对我，这都意味着有更多的工作要做。最后，我要感谢我的父亲和母亲，是他们培养我对任何事情都充满着旺盛的好奇心（也因为好奇而购买了 Commodore 64，现在回想起来，这真是一个相当不错的投资）。

作者简介

Neeharika Adabala

nadabala@cs.ucf.edu

Neeharika Adabala 在气态容积的建模和渲染工作中引入了粒子贴图的概念，并由此获得了博士头衔。她曾经在 Philips Research、MIRALab、日内瓦大学、Media Convergence Lab 和佛罗里达中央大学任职。她在动力仿真和真实渲染领域都发表过研究报告。她感兴趣的研究包括实时渲染、物理动力学、照明模型、渲染中的知觉问题、科学可视化和敏捷图形。

Barnabás Aszódi

ab011@freemail.hu

Barnabás Aszódi 是布达佩斯技术和经济大学的在读博士。他所关注的领域集中在计算机图形图像（例如：实时真实阴影计算）、动画制作和游戏。他曾给 WSCG'04、CESCG'02 和其他一些出版物投过稿件。他非常欢迎大家通过上面的邮件地址给他一些反馈或交换彼此的想法。

Tony Barrera

tony.barrera@spray.se

Tony Barrera 被公认为自学成材的数学天才。他已经发表了 25 篇论文，涉及多个不同的主题，其中有 18 篇是关于计算机图形、数字分析和数学领域的科技论文。Tony 现在正与 Ewert Bengtsson 和 Anders Hast 一起搞研究。

Ewert Bengtsson

ewert@cb.uu.se

1988 年，Ewert Bengtsson 成为了瑞典乌普萨拉大学计算机化影像分析的教授。他主要研究影像分析在生物医学中的应用开发的方法和工具。这包括可视化和计算机图形两个方面，因为三维生物医学影像的可视化与此有关。他已经发表了大约 120 篇国际研究论文。1974 年，他获得了瑞典乌普萨拉大学的博士学位。他还是 IEEE、SPIE 和 IAPR 的成员。

James Boer

author@boarslair.com

1997 年以来，James Boer 一直都是游戏开发行业的积极参与者。当年，他帮助开发了令人称奇的热门游戏《猎鹿人》（*Deer Hunter*）。他还是一个多产的作家，曾经参与过 7 本游戏编程书籍的写作，其中包括他自己独立完成的《游戏音效编程》（*Game Audio Programming*）一书，并在行业杂志上发表过若干篇文章。James 目前是 Amaze 娱乐公司的程序员，正在帮助开发 Elemental 引擎的新技术，该引擎是 Amaze 娱乐公司内部开发的跨平台游戏机引擎。

John Bolton

johnjbolton@yahoo.com

John Bolton 是美国旧金山 Page 44 Studios 的软件工程师。他目前正在紧张地制作 Sony 公司面向 PS2、PSP 和 PS3 的曲棍球系列游戏，这些游戏即将明年推出。从 1992 年起，他就开始了专业的游戏编程工作，并曾经在几款游戏中担任过主程序员，其中包括《我没有嘴，但我必须尖叫》（*I Have No Mouth and I Must Scream*）、《权利和魔力英雄》（*Heroes of Might and Magic*）、《高热棒球》（*High Heat Baseball*）。

Markus Breyer

thebreyers@comcast.net

Markus Breyer 拥有工程计算机科学硕士学位，并且在游戏行业有 8 年的从业经历。他曾经参与开发了多款游戏，包括《星战赏金猎人》（*Star Wars Bounty Hunter*）、《角斗士》（*Gladius*）和《回火》（*Return Fire*）。Markus 目前正着手于面向次世代平台的 Factor 5 开发技术。

Martin Brownlow

mbrownlow@shiny.com

Martin 10 岁的时候就开始在朋友的 ZX81 机器上编写程序了。完成学业之后，Martin 在 Virtuality 有限公司（英国）开始了自己的职业生涯，当时他负责编写 VR 街机游戏。3 年之后，他搬到美国，就职于 Shiny 娱乐公司。他参与了 Shiny 公司若干款游戏的开发，包括 *MDK* 和 *Sacrifice*。目前，他正在潜心开发《骇客帝国》（*Matrix*）视频游戏。

Warrick Buchanan

warrick@chimeric.co.uk

Warrick Buchanan 是 Chimeric 有限公司的开发主管，负责 Maxinima 和 ScreenSaverMax 系列产品。在多家视频游戏开发公司任职的几年间，他还为 Imagination 科技有限公司的显卡

驱动程序的开发出了一份力。他的玩乐范围颇广，从尖端的显卡到蹦床，无所不包。

Jamie Cheng

jcheng@relic.com

Jamie Cheng 是 Relic 娱乐公司的 AI 程序员。他最近开发完成了游戏《*Warhammer 40000: Dawn of War*》的 AI 部件。他还是 Relic 公司和阿尔伯达大学 GAMES 小组的核心联络人，双方目前正通力合作，积极推动游戏 AI 的商业应用。工作之余，James 喜欢和其他人一起开发反标准的游戏。James 获得了西蒙 Fraser 大学的计算机科学学士学位。

Octavian Marius Chincisan

mariuss@rogers.com

1987 年，Octavian 从 Cluj-Romania 技术大学毕业，并获得了电子工程硕士学位。1 年之后，他获得了大学毕业之后的又一个应用电子学的文凭，并完成了一个项目，即建造一台基于 Z80 的、与 Sinclair Spectrum 兼容的个人电脑。从 1988 年到 1994 年，他一直是一家金融机构的 C++程序员。1994 年，他来到加拿大，分别在几家公司担任 C++资深软件程序员。2000 年，他迎来另一个新的挑战：游戏编程。他在这个领域自学而成，他用自己的知识、热情和辛苦的工作换来的成果就是创作了 Getic 3D 编辑器和 Getic SDK，这两个产品目前都在开发之中。现在，Octavian 在 Zalsoft 公司担任软件架构师。

Ignacio Incera Cruz

ignacio@incera.net

Ignacio 是一名软件工程师，目前正在参与西班牙马德里欧洲防御区域计划（European Defense Area Projects）。明确来讲，他的工作内容是实时仿真、3D 地形、地理信息系统（GIS），以及任务规划和报告系统。他拥有 Deusto 大学的计算机科学学士学位和虚拟现实专业的硕士学位。他现在还在马德里技术大学攻读计算机科学和人工智能的博士学位。他的研究集中在机器人、分子计算和人工头脑（Artificial minds）方面。

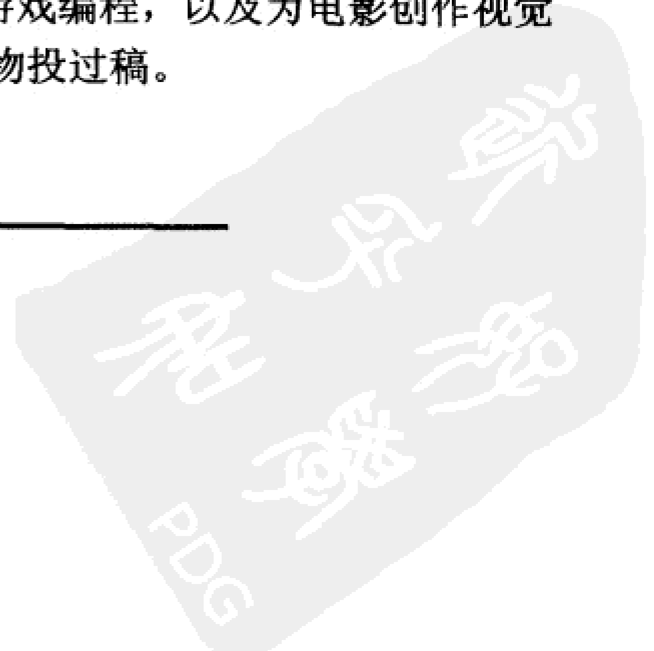
Szabolcs Czuczor

cs007@ural2.hszk.bme.hu，czsz@freemail.hu

Szabolcs Czuczor 是匈牙利 BUTE 大学控制工程和信息技术学部计算机图形小组的在读博士。他的研究兴趣包括多媒体、视频和影像处理、Web 和游戏编程，以及为电影创作视觉和声音特效。他曾向 WSCG'04、CESCG'02 和其他一些出版物投过稿。

William E. Damon Ⅲ

wdamon@ati.com



William 是 ATI 研究公司的工程师。在出版本书时，他已有很好的专业背景，其中包括游戏行业 5 年的技术经验。在这 5 年里，他主要关注的是软件核心技术和平台性能。William 拥有美国弗吉尼亚工学院和州立大学的计算机科学荣誉学士学位。

Mark DeLoura

madsax@satori.org

Mark DeLoura 是《游戏编程精粹》系列书的策划编辑。作为 Sony Computer Entertainment 美国公司的开发者关系经理，他有机会和世界各地的游戏开发人员共享那些技术和非技术的信息。Mark 非常执著于自己的理念，即创造一个共享且愉快的体验，教育并鼓励人们互相交流。他曾是美国《游戏开发者》（*Game Developer*）杂志的前主编，并曾担任过任天堂美国分公司开发者支持小组的首席软件工程师，期间他一直倡导着“愉快体验”的理念。

Chuck DeSylva

Chuck.V.DeSylva@intel.com

Chuck DeSylva 是美国加利福尼亚州福尔松市 Intel 公司软件解决方案小组的资深软件应用工程师。他目前正领导一个工程师团队，研究行业中主流软件游戏和媒介游戏的优化问题。在钻研软件性能之余，他喜欢弹奏吉他和贝司，玩第一人称射击游戏，以及周游世界。

Shekhar Dhupelia

sdhupelia@gmail.com

Shekhar Dhupelia 刚进入游戏行业，就在圣地亚哥 Sony（SCEA）公司的 SCE-RT 小组工作了两年，期间他负责开发 PS2 的在线软件和后端服务器架构，为 *SOCOM: US Navy Seals*、*Frequency*、*Twisted Metal Black Online*、*NFL Gameday* 和其他许多 PS2 游戏提供不可或缺的平台支持。后来，他转向了微软公司 *NBA Inside Drive 2004* Xbox Live 的开发。而在此之前，他在 Midway 游戏公司工作了一段时间，开发面向 PS2 和 Xbox 的游戏 *NBA Ballers*。Shekhar 以前曾为《游戏编程精粹 4》写过文章，还曾参加过 Charles River Media 的《游戏商业的秘密（修订版）》一书的写作。他在游戏开发者大会（GDC）和 Penny Arcade Expo（PAX）大展上发表过演讲，相关主题都围绕着游戏策划。他现在正在为 Studio Gigante 和 THQ 公司的游戏 *WWE Wrestlemania XXI* 开发 Xbox Live 玩法。

Mike Dickheiser

mike.dickheiser@redstorm.com

Mike 是一个有 9 年游戏行业经验的老手，现在是 Red Storm 娱乐公司的软件工程师。在其职业生涯当中，他曾开发过飞行模拟器、动态流体建模、碰撞系统和汽车物理模型。Mike 当前的工作重点是为 *Ghost Recon* 产品线开发汽车的 AI 控制系统。在教授直升飞机如何穷追

不舍并摧毁坦克、步兵团，以及指导游戏策划者之余，Mike 喜欢在家和相濡以沫的妻子 Jaye 玩计算机游戏、做体育运动、弹钢琴和休息放松。

Jean-François Dubé

jfdube@ubisoft.qc.ca

Jean-François（在 Gamedev.net 上又名 deks）已经在游戏行业摸爬滚打了近 8 年的时间。他目前在育碧公司（UbiSoft）的蒙特利尔工作室工作，担任一款近期上市的次世代游戏机游戏的首席技术程序员。Jean-François 曾经是 Xbox 游戏《彩虹 6 号 3》（*Rainbow Six 3*）的首席程序员，并且装配过其他几款游戏，像在 PS2 上运行的 *Batman Vengeance* 和在 PC 上运行的游戏 *Speed Busters*。

Patrick "Gizz" Duquette

gizmo@gizz-moo.com

Patrick，又名 Gizmo，或简称"Gizz"，就像百事可乐一样：少生气并且一直认为 15 摄氏度是最适宜工作的室内温度，这比他的同事们所害怕的温度要低了很多。他还在钻研他的无缝服务器（Seamless Server）和很多工具软件，于此同时，他渴望着某天能抽出时间来完成他的分布式射线追踪器。

Dominic Filion

dfiliong@videotron.ca

Dominic 是 Artificial Mind & Movement 的资深 3D 引擎开发人员，负责 3D 特效和物理仿真的研究。此前，他在 DC Studios 担任技术主管，带领技术团队开发了该公司内部的跨平台 3D 引擎。他之前曾经参与开发了 4 款商业 3D 引擎产品，并担任了其中 2 款引擎的主架构师。他还曾经在 Microids 公司和 Key Studios 公司工作过。大家可随时和他探讨书中的文章，或者交个朋友，随便聊聊。他的个人网站是 *http://www.bingecoder.com*。

Mario Grimani

mgrimani@san.rr.com

Mario Grimani 早在 20 年前就投身游戏行业，可算是元老级人物。自 1987 年发行了自己的第一款游戏之后，他倾注了所有的精力，去开发 Amiga 平台。但是，由于 Amiga 平台过早地死亡，他离开了游戏行业，并不打算再回头。20 世纪 90 年代中期，他又重出江湖，分别在几家知名的公司任职，包括 Ion Storm、Ensemble Studios、Verant Interactive 和 Sony 在线娱乐公司。在 Ensemble Studios 工作期间，他的角色是 AI 专家，负责提高人机对战的竞争性。他为《帝国时代 2：王者时代》（*Age of Empires II: The Age of Kings*）和《帝国时代 2：征服者》（*Age of Empires II: The Conquerors*）这两款游戏开发了一个脚本系统和一个人机 AI 系统。

在《神话时代》(*Age of Mythology*)这款游戏的开发初期，Mario 担任 AI 主管，负责 AI 架构。在加入 Verant Interactive 公司（后来变成了 Sony 在线娱乐公司）之后，他担任了 Sovereign 这款游戏的主程序员，并在后来加入了 *EverQuest II*的开发团队。现在，Mario 是 Xtreme Strategy 游戏公司的合伙人，他正在此利用自己丰富的技术经验，为市场带来下一代的游戏产品和游戏技术。

Julien Hamaide

julien_hamaide@hotmail.com

8 岁的时候，Julien 就开始在他的 Commodore64 机器上编写文本游戏了。同年，他编写了自己的第一个汇编程序。岁月流逝，但热情不减。他坚持自学，阅读他父母可以买来的所有书籍。2003 年，21 岁的他从比利时的 Faculté Poly-technique de Mons 学校毕业，成为一名多媒体电子工程师。他现在在 TCTS/Multitel 公司负责语音和图像处理工作。为了能够进入游戏行业，他孜孜不倦，努力工作。Open-eXtnd 是他最新的项目（XTND 的一个免费实现），该技术将应用在 AI 中。

Sami Hamlaoui

disk_disaster@hotmail.com

备受 AI 困扰的 Sami，在意识到自己已经写完一篇有关音频的文章时，变得更加糊涂了。在文章没有跑题之前，他把大部分的时间花在了那 500 个虚拟角色身上，试图（在把帧速率保持在每秒几帧，而不是每帧几秒的情况下）让它们看上去很敏捷。

Matthew Harmon

matt@matthewharmon.com

念大学的时候，Matthew Harmon 就已经在开发游戏了。他在攻读电影理论和评论专业的学位的同时，就参与了 subLogic 公司的“微软飞行模拟器（Microsoft Flight Simulator）”的开发工作。此后，他还在 Mission Studios 公司和 Velocity 开发公司担任过主程序员和开发主管。最近，他与朋友合伙成立了 eV Interactive 公司，继续开发游戏产品，并在军事训练和模拟仿真领域应用游戏开发技术。闲暇时间里，Matt 会绕着房子，与儿子 Alex 和 Greg 一起追逐嬉戏。

Anders Hast

aht@hig.se

1996 年，Anders Hast 就已经成为计算机科学专业的讲师，并在 2004 年成为 Gävle 大学的副教授。同年，他获得了瑞典乌普萨拉大学的博士学位。他和 Barrera 和 Bengtsson 一起研究出了计算机图形的基础算法，并探索了新的方法来解决这些算法背后的数学问题。他们还

为此发表了约 20 篇研究论文和书刊文章。

Daniel F. Higgins

dan@stainlesssteelstudios.com

Dan Higgins 是 Stainless Steel Studios（简称 SSS）公司引以为豪的一名成员，这是一家位于剑桥的、由 Rick Goodman 领导的开发历史题材即时战略（RTS）游戏的公司。在此之前，Dan 的背景并不在游戏行业，而是为 History Channel（历史频道）、A&E（Arts & Entertainment）和 Biography Channel（传记频道）编写高性能的搜索引擎。作为 SSSI 公司一位充满激情的程序员，Dan 的工作领域非常广泛，包括军事 AI、寻径、地形分析、优化、动物 AI、阵型和物理。Dan 的老家在马里兰，他和他的家人“意外地”爱上了波士顿，并且感觉就像是土生土长的新英格兰居民。他毕业于马里兰 Frostburg 州立大学，学习的是计算机科学专业。人们在交谈中描述 Dan 的时候，常常会以惊人的频率冒出“freak（思想怪诞）”和“spaz（怪人）”这样的字眼。

Charles E. Hughes

ceh@cs.ucf.edu

Charles E. Hughes 是佛罗里达中央大学计算机科学学院的教授和毕业生协调人。他还是该校电影和数字媒体学院的合聘教授，并担任着 Media Convergence 实验室的首席科学家，该实验室位于佛罗里达中央大学（UCF）的仿真训练学院，是个跨学科的合作项目。他独立或与他人合作完成了 100 多篇学术文章、7 本图书部分章节和 6 本书的写作。他当前的研究兴趣是混合现实（mixed reality）和分布式计算的模型。

Robin Hunicke

hunicke@cs.northwestern.edu

Robin Hunicke 正在西北大学（Northwestern University）攻读 AI 和游戏专业的博士学位。业余时间里，她致力于消除学术研究与行业应用之间的鸿沟——一边为 IGDA 的教育委员会工作，一边在 GDC 的游戏策划和调整研讨会（Game Design and Tuning Workshop）中授课，并积极参与着 Experimental Gameplay Workshop 和 Indie Game Jam 之类的行业活动。她的爱好是 M.U.L.E。

Hyun-jik Bae

imays@hitel.net

Hyun-jik Bae 是 MowelSoft 公司的技术主管，该公司目前正在开发 *Blitz 1941*™，一款以第二次世界大战为背景的 MMO 游戏。从 1997 年开始，他已经开发了几款 3D MMO 游戏。他的第一个游戏作品是用 MSX BASIC 编写的 *SpeedGame*（not™），这个游戏需要玩家快速

PDG

地敲击空格键，也因此这个程序就成了他小学校园里计算机崩溃的元凶。

Scott Jacobs

scott@escherichia.net

Scott Jacobs 原本是要成为一名微生物学家的。他放弃昂贵的大学学业，在游戏行业里工作，为的是能够支付房租。声称要成为一名网络程序员的他，经常喜欢在其他所有领域工作，如脚本引擎、物理和图形图像，以借此汲取营养。他曾在 Interactive Magic 公司、Sinister 游戏公司（育碧子公司）和 Red Storm 娱乐公司（育碧子公司）工作过。目前，他正在开发游戏 SimWars，他渴望它们能够让 *Super Puzzle America's Army II: Turbo* 正式退场。

Wendy Jones

gamegirl@fasterkittycodecode.com

Wendy Jones 是一名游戏开发人员兼行业福音传道者。她担任着多个角色，从行业记者到游戏程序员到作家。她是 IGDA 南佛罗里达分部的活跃分子，是当地分部的董事成员。她目前正忙着为手持设备开发自由软件，并撰写一些与游戏开发有关的文章和书籍。

Eric Lengyel

lengyel@terathon.com

Eric Lengyel 是 Naughty Dog 公司高级技术小组的资深程序员。他是畅销书《3D 游戏编程和计算机图形中的数学》（*Mathematics for 3D Game Programming and Computer Graphics*）的作者，并为很多行业媒体撰写过大量的文章，从 gamasutra.com 到《游戏编程精粹》系列书。Eric 致力于 3D 图形的研究已有 10 多年之久，在此期间，他曾担任过 *Quest for Glory 5* 这款游戏的主程序员以及 C4 引擎的首席架构师。

Chris Lomont

Clomont@math.purdue.edu

Chris Lomont 从小学 5 年级的时候就开始编程，那时候他就学会了在 TI-55 可编程计算器上编写简单的游戏。经历过各样各式的计算机之后，他在大学里开始了 PC 编程，并在那里获得了物理、数学和计算机科学三学位，且编写了一个作为高级项目的国际象棋程序。他曾在 Black Pearl 公司（现已倒闭）做过短期的视频游戏编程工作，并用所得收入还清了学校贷款。然后他去了 Purdue，并在 2003 年获得了数学博士学位。尽管他已经非常专业地开发了许多类型的应用程序，包括视频游戏、金融模型、机器人软件、解析器和编译器、图像处理工具和密码软件等，但作为一个毕业生，他还为很多公司提供过咨询服务，大部分都是与图形开发有关的。目前，他在 cybernet.com 做量子计算的研究。他的爱好有钢琴、国际象棋、

体育、编程、迷题设计，以及尝试写书。

Michael Mandel

mmandel@gmail.com

Michael Mandel 刚刚毕业于卡耐基·梅隆大学，并在此获得了计算机科学的本科和研究生学位。他的研究生毕业课题是利用仿真和数据驱动的方法实现角色动画。他曾发表过很多有关智能代理开发的学术文章。在完成毕业课题的同时，他一直都是卡耐基·梅隆大学的访问学者。他曾在 LucasArts 和微软公司任职，参与过多款 Xbox 和 PC 游戏的开发。他现在是苹果计算机公司的工程师。

Adam Martin

adam@grexengine.com

Adam Martin 是 Grex 游戏公司的 CEO，这是一家 MMOG 中间件公司。虽然 Adam 目前专注于公司的发展战略及业务开发，但 Grex 公司的大部分产品都是基于 Adam 自己的专利技术。他在英国剑桥大学获得了计算机科学专业的学位，并担任过两款已发行游戏的开发者和制作人。他在 MDC 大会上做过演讲，并经常现身于 MUD-DEV。2004 年，他成立了 Java Games Factory，以推动专业质量 Java 游戏的开发，并为众多的 Java 游戏工作室提供帮助。

Maciej Matyka

maq@panoramix.ift.uni.wroc.pl

Maciej Matyka 生于波兰 Wroctaw 市。他在 Wroctaw 大学（物理理论部和天文系）学习计算物理学，并凭借杰出的学术表现获得了奖学金。8 年来，Maciej 一直是 Amiga 和 PC DEMO SCENE 领域里非常活跃的程序员。他的研究兴趣主要在基于物理的建模领域。Maciej 是物理仿真软件的创作者，其中包括获奖产品 *Fluid*（1999 年第二届全系物理软件竞赛第二名）和 *Waves*（2000 年第三届竞赛第一名）。他还是波兰国内的几家专业刊物的作者，其文章大部分都与物理仿真有关。在 Wroctaw 大学的时候，他给高中生做过演讲，并在全系的研讨会上发言，介绍自己的物理软件。Maciej 是《物理学中的计算机仿真》(*Computer Simulation in Physics*) 一书的作者，该书于 2001 年由 Helion 出版社出版。在 2003 年由 Charles River Media 公司出版的作者为 Jeff Lander 的《图形编程方法》(*Graphics Programming Methods*) 一书中，还收录了 Maciej 的一篇关于软体动力学的文章。

Patrick Meehan

pmeehan@tenaciousgames.com

1996 年从 DigiPen 毕业之后，Patrick Meehan 在任天堂技术开发公司开始了自己的职业生涯。他在游戏行业的最初经历是在 Interactive Imagination 和 Amaze Entertainment 公司做开

发人员。他的行业经验包括游戏开发的多个方面，并参与过多种平台的引擎产品开发。在出版本书之际，他是西雅图一家公司——Tenacious 游戏公司的开发人员，追求着吉普赛式的生活。

Nathan Mefford

nmefford@yahoo.com

Nanthan Mefford 是 Firaxis 公司的软件工程师，他专注的领域是软件架构、优化和 3D 图形技术。他非常高兴能有机会为《游戏开发精粹》系列书提供稿件，该丛书也使他受益颇丰。他欢迎读者的来信和反馈。

Jason L. Mitchell

jasonlmitchell@comcast.net

Jason 是 ATI Research 公司 3D 应用研究小组的组长，他在这里开发着新颖的 3D 图形技术并编写相关文档。他曾经为《游戏编程精粹》和《ShaderX》系列书、《游戏开发者》杂志、Gamasutra.com 以及学术出版物撰写过有关图形图像处理的文章。他定期出席世界各地的图形图像和游戏开发大会。在 http://www.pixelmaven.com/jason 这个网址可以查阅 Jason 的出版作品和过去的演讲内容。

Ian Parberry

ian@cs.unt.edu

Ian Parberry 是北得克萨斯大学计算机科学与工程系的教授，是该校游戏编程教育的早期倡导者。他创作出版了 6 本书，其中 3 本是关于游戏编程方面的。他还在各种学术会议和期刊上发表了大量的论文，内容涉及广泛的计算机主题，从计算机科学理论到计算机游戏。

Kim Pallister

Kim.Pallister@Intel.com

Kim Pallister 是 Intel 公司软件解决方案小组的工程经理兼技术传道者。他专注的领域是图形图像与游戏技术。Kim 为《游戏编程精粹 2》和《游戏编程精粹 3》，以及其他一些出版物提供过稿件。他非常欢迎大家给上面的地址发邮件，来提供反馈或交换看法。

Borut Pfeifer

borut_p@yahoo.com

1998 年，Borut Pfeifer 从美国乔治亚技术学院（Georgia Tech）毕业，获得了计算机科学学士学位。在担任过各种软件开发的职位之后，2001 年，他和朋友合伙创办了自己的游戏工

作室——White Knuckle Games。他目前在 Radical 娱乐公司工作，是 *Scarface* 这款游戏的程序员，并写过很多游戏开发方面的文章。

Frank Puig Placeres

fpuig2003@yahoo.com

Frank Puig 是古巴信息科学大学虚拟现实小组的主管。他设计实现了 CAOSS 引擎和 CAOSS 工作室。CAOSS 引擎已经成功地应用到了几款游戏中，如 *Herlec* 和 *Knowledge Land*。他还设计开发了几个游戏开发工具，用以改进和简化运动捕捉数据、高级纹理贴图和动画合成等事务的处理。

Steve Rabin

steve_rabin@hotmail.com

Steve Rabin 投身游戏业已有十余载，目前在任天堂美国分公司工作。他曾为 3 款已经发行的游戏编写了 AI 系统，并为《游戏编程精粹 1～4》投过稿件。他是《游戏编程精粹 2》"人工智能（AI）"部分的编辑，并且还是《游戏开发概论》（*Introduction to Game Development*）和《AI 游戏编程箴言》（*AI Game Programming Wisdom*）1 和 2 的创办者兼主编。他曾在游戏开发者大会（GDC）上发言，并且是华盛顿大学分校游戏开发认证计划（*Game Development Certificate Program*）的讲师。Steve 拥有华盛顿大学计算机工程学位，目前正在攻读计算机科学的硕士学位。

Rishi Ramraj

thereisnocowlevel@hotmail.com

Rishi Ramraj 是滑铁卢大学系统设计工程专业的学生，刚刚完成了第一学年的学业。他高中时就利用相当多的时间学习了 C++编程，校园里时常能看到他骚扰老师的身影。Rishi 将自己的职业经历更多地归功于 Jeff Molofee 和 nehe.gamedev.net 社区。在其第一份工作期间，他为 Bedrock Research Corp.™ 设计开发了一个 3D 向量分析套件——*VectorChrome*™。他目前在 Alias®公司任职，这是他的第二份工作。如果不在下棋、设计代码或编程，那么他一定是在学习。本书收录了他放弃了睡眠而编写的一个程序。

Michael Ramsey

miker@masterempire.com

Micheal Ramsey 是 2015 公司的主程序员，他刚在此完成了针对 Xbox 和 PC 平台的游戏 *Men of Valor*。他目前正在开发用于次世代游戏机的技术。20 世纪 90 年代早期，Mike 曾是 VR1 公司 *Lost Continents* 游戏项目的 3D 客户端首席工程师，并为 *Mike Piazza's Strikezone*、*Master of the Empire* 和许多 RPG 游戏编写了 3D 引擎。他拥有 MSCD 的计算机科学学士学位。

Mike 为《游戏编程精粹》和《*AI Wisdom*》系列丛书都投过稿件。闲暇时，Mike 喜欢宠溺他的宝贝女儿 Gwynn。

Aurelio Reis

AurelioReis@gmail.com

Aurelio 现在是 Raven 软件公司的一名玩法和技术程序员，曾参与了 *Jedi Academy* 和 *Quake 4* 等游戏的开发。业余时间，他喜欢研究新的图形图像算法，搞搞自己的游戏引擎，和琢磨有趣的令人兴奋的新游戏玩法。

Bjarne Rene

bjarne.rene@circle-studio.com

1995 年，Bjarne 在 Bullfrog 公司找到一份工作，并由此进入了游戏行业。他曾废寝忘食地编写了游戏《基因战争》(*Gene Wars*) 中计算机对手的程序，此后，又投入到了游戏《主题公园》(*Theme Park Wrold*) 中的游戏逻辑和 A.I.（人工智能）系统的编程。然后，他在老家挪威的卑尔根大学用两年时间完成了计算机科学专业的学位。在这之后，他回到英国，重新投身游戏行业。他目前在 Circle Studio 工作，负责对象管理系统和 A.I.系统。

Graham Rhodes

grhodes@nc.rr.com

Graham Rhodes 是美国北卡罗莱纳州 Raleigh 市 Applied Research Associates 公司东南分公司（Southeast Division）的首席科学家。他在交互式实时 3D 图形、游戏和物理仿真等软件开发领域拥有近 20 年的经验。Graham 曾经担任过多个游戏项目的软件开发主管，这些游戏包括在 Commodore VIC-20 和 Atari 400 家用电脑上为青少年开发的街机风格的游戏；为 *World Book Multimedia Encyclopedia* 开发的一系列受赞助的教育类小游戏；为贸易行业安全训练开发的第一/第三人称动作游戏（采用了最新技术的 3D 游戏引擎）。他目前正在参与开发为军队及本国防御仿真及训练提供物理解决方案的软件。Graham 以前曾为《游戏编程精粹 2》编写过一章内容，并且是《游戏编程精粹 4》“物理”部分的编辑。他是游戏开发互联网门户网站 gamedev.net 的数学和物理频道的主持人，出席过每年一度的游戏开发者大会（GDC），并定期参加着 GDC 和一年一度的 ACM/SIGGRAPH 大会，同时还是 ACM/SIGGRAPH 和国际游戏开发者联合会（IGDA）的会员。

Timothy E. Roden

roden@cs.unt.edu

Timothy 是北得克萨斯大学计算机科学与工程系的讲师和博士应考人。在北得克萨斯大学里，Timothy 教授的是编程和计算机图形课程。在进入北得克萨斯大学之前，他曾是仿真

行业中的图形软件开发人员，在 Evans & Sutherland 工作过 6 年。他主要的研究兴趣是 3D 图形应用程序的过程化内容生成。

Thorsten Scheuermann

thorsten@ati.com

Thorsten 是 ATI 公司 3D 应用研究小组的软件工程师，负责图形图像 demo 和新渲染技术的开发，该小组是 ATI 公司 Demo 团队的一部分。在 ATI 公司工作之前，他是教堂山（Chapel Hill）北卡莱罗纳大学有效虚拟环境（Effective Virtual Environments）研究小组的成员，这让他有机会玩弄各种昂贵的 VR 玩具和创作开发 sickness-inducing immersive 游戏。Thorsten 获得了北卡莱罗纳大学计算机科学专业的硕士学位，之前他在德国的卡尔斯鲁厄大学学习。

Shawn Shoemaker

shansolox@yahoo.com

Shawn Shoemaker 有着 7 年游戏行业的从业经验，现在是 Stainless Steel Studios 的首席程序员助理。他负责过游戏《地球帝国》（*Empire Earth*）和《帝国：现代曙光》（*Empires: Dawn of the Modern World*）的物理、AI、战斗、图形特效和随机地图等系统，并由此在业界获得了良好的声望。现在，Shawn 正努力开发着 Stainless 公司的下一个游戏项目，该项目目前还没有对外公开任何信息。Shawn 拥有弗吉尼亚技术大学计算机工程专业的学士学位和硕士学位。在同意牺牲自己的夏季假期、临危受命之前，Shawn 从事的是 Intel 公司的硬件和 CAVE 虚拟现实环境方面的工作。虽然这份差事非常不错，但 Shawn 很快就表明，他从来不想靠此谋生。他一直想成为一名伐木工人。

Dr. Finnegan Southey

fdjsouthey@uwaterloo.ca

Finnegan Southey 博士是 Alberta 大学 GAMES 小组和 Alberta 机器学习灵创中心（Ingenuity Centre for Machine Learning）的成员。他在滑铁卢大学获得了人工智能的博士学位。他的研究集中在商业游戏上，包括计划、机器学习和游戏可玩性分析的数据采样方法。除了在 Relic 娱乐公司的工作成绩之外，他的研究成果已经被行业伙伴 Electronic Arts 公司应用到了 FIFA 2005 这款游戏的开发当中。

Shea Street

shea.street@tantrumgames.com

在因特网还未发展到今天这个样子之前，Shea Street 就为早期的拨号服务供应商开发过在线游戏，并由此开始了他的多人游戏编程生涯。他已经有超过 15 年的游戏编程经验，且完

PDG

全是自学成材，却拥有 Full Sail 公司游戏设计与开发方面的计算机科学学位。这些年来，他帮助开发了无数的游戏产品，同时还为很多公司提供私人咨询服务，以满足他们项目的需要。Shea 现在是 Tantrum 游戏公司的联合创办人兼主程序员。Shea 非常感谢 Tantrum 公司全体同仁的辛勤劳动和无私奉献，以及在写作本书时他们所给予的理解和耐心。他还要感谢所有论坛中参与他讨论和演讲的每个人，以及那些和他一起熬夜喝咖啡的人。最后，Shea 要感谢所有一直以来都帮助和支持他的朋友及家人，特别是他的父亲。如果不是父亲给他买了第一台计算机，让他对编程产生了兴趣，他也不可能有今天的成就。

Gábor Szijártó

szijarto.gabor@freemail.hu

Gábor 是匈牙利布达佩斯科技大学的在读博士。他 10 岁就开始编写程序了，他的硕士毕业设计课题是 3D 面部建模。他的特长是着色器编程和 3D 图形算法。

Andy Thomason

athomason@acm.org

最近，Andy Thomason 和游戏 *Lara Croft* 的策划者 Toby Gard 一起完成了游戏 *Galleon* 的开发。从 20 世纪 70 年代开始，利用自制的 Z80 硬件，他就一直与 Psynosis 和 Rage 一起工作。现在，他是一个自由技术研究员兼英国布里斯托尔港口的造船主。

Matthew Titelbaum

mtitelbaum@lith.com

Matthew Titelbaum 是 Monolith Production 公司的资深软件工程师，现在正为 MMORPG 游戏《骇客帝国在线》(*The Matrix Online*) 开发 AI (人工智能) 系统。他在 Crystal Dynamic、DreamWorks Interactive 和 SCEA 公司开发过一些游戏机游戏，并由此开始了自己在游戏行业的职业生涯。此后，他进入 Sony 在线娱乐公司，开始涉足大型多人游戏的开发。在 Sony 在线娱乐公司任职期间，他参与开发了 *Sovereign* (一个实验性质的 MMO 即时战略游戏) 和几个《无尽的任务》(*EverQuest*) 游戏的扩展包。最让他感到骄傲的，是他在《无尽的任务》这款游戏的寻径系统的大修和优化工作中所做的贡献。Matthew 拥有卡耐基·梅隆大学计算机科学专业的学士学位。

Marco Tombesi

baggior@libero.it

Marco Tombesi 是一位意大利计算机工程师，2002 年毕业于罗马的 La Sapienza 大学，获得了一级学位。他做了多年的自由游戏开发者，现在被公认为 C++和 OpenGL 大师。他在《游戏编程精粹 2》中发表了一篇成功的文章——《3ds max 皮肤导出器和动画工具包》。目前他

正在开发一款足球游戏。

Christopher Tremblay

chris@Barney.zapto.org

Christopher Tremblay 住在旧金山海湾地区。他拥有软件工程学士学位，并且很快就要拿到数学学士学位。他的业内工作涉及各种主题，包括游戏 AI、核心网络、软件渲染算法，以及 3D 几何算法的底层和优化。他现在就职于 Motorola 公司，正在为下一代手机产品开发下一代的 2D/3D 图形光栅化引擎。业余时间里，他喜欢玩滚轴溜冰、滑雪，甚至是一边编程序一边享受显示屏的辐射。

Bretton Wade

brettonw@microsoft.com，Bretton_Wade@acm.org

Bretton Wade 是个在游戏和图形图像行业中有着 10 年经验的老手，目前是 Xbox 系统软件团队的一名主管，同时也是华盛顿大学分校饱受称赞的游戏开发课程的讲师。他以前担当过的游戏开发角色包括：一个与暴雪娱乐公司签约的独立游戏开发工作室的项目主管，微软公司的飞行模拟器（*Flight Simulator*）开发主管，以及多款由微软游戏工作室（Microsoft Game Studios）发行的游戏的个人贡献者。在 SGI 公司工作时，Bretton 负责虚拟现实（VR）创作工具的开发。他是微软研究院的研究工程师，研究方向是高级渲染技术。他毕业于弗吉尼亚技术大学（Virginia Tech），拥有科内尔大学（Cornell University）计算机图形编程专业的硕士学位。

Niniane Wang

niniane@gmail.com

Niniane Wang 已经在微软游戏公司工作了 5 年，最近刚刚升为《飞行模拟器 2004》的开发主管。她在气象图形领域的工作成果已经发表在了 SIGGRAPH、GDC、《游戏开发者》杂志和一些学术期刊上。她拥有加利福尼亚理工学院的计算机科学学士学位，以及华盛顿大学的计算机科学硕士学位。她现在是 Google 公司的软件工程师。

Jon Watte

hplus-gpg5@mindcontrol.org

10 岁的时候，Jon 就开始创作软件了，并且一发不可收拾。他协助发布了几个大型产品，包括 CodeWarrior 开发工具、BeOS 多媒体操作系统和 There 虚拟世界。Jon 现在在 Forterra System 公司工作，负责大型分布式仿真平台的研究开发。

翻译和审校员

孟宪武 mxwbrio@hotmail.com

本书主审并翻译了通用编程、人工智能、物理及图形等部分34篇文章。毕业于北京航空航天大学计算机系，目前供职于班布技术有限公司，之前曾在《中国计算机报》和《数字娱乐开发》杂志社工作。

潘李亮 xheartblue@163.com

翻译2.1、2.2、2.5、7.1共4篇文章。毕业于西北工业大学软件学院，目前在Corel-InterVideo公司从事多媒体软件开发工作。之前曾在游戏公司从事3D图形引擎和休闲游戏开发。

毛慧明 mhm15191@msn.com

翻译2.3、2.6、7.2、7.3共3篇文章。毕业于西北工业大学土木建筑工程系，目前在Corel-InterVideo公司从事多媒体软件开发工作。之前曾在游戏公司从事3D图形引擎和休闲游戏开发。

李巍 noslopforever@hotmail.com

翻译文章2.4。毕业于中国农业大学计算机系，目前在中娱在线从事图形引擎和工具的开发工作。之前曾参与3D图形引擎和MMORPG开发。

龚敏敏 gongminmin@yeah.net

负责翻译5.2至6.7共16篇文章。中国科学技术大学软件工程硕士，开源游戏引擎KlayGE的开发者。目前在微软亚洲研究院网络与图形组担任研究和开发工作。

孔玲君 adapop51@hotmail.com

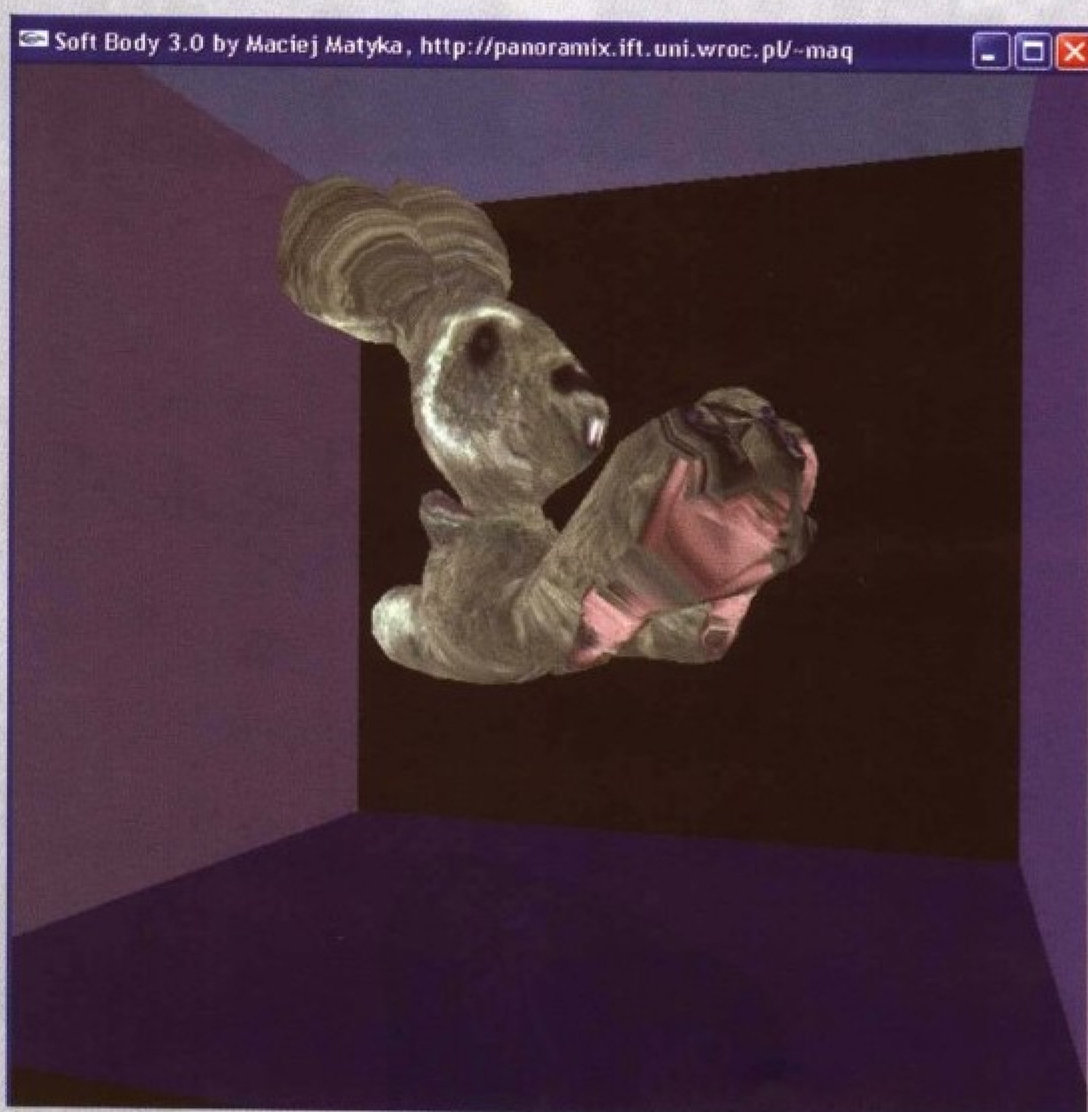
负责审校《游戏编程精粹5》5.2至6.7共16篇文章。

徐丹 flymemory@sina.com

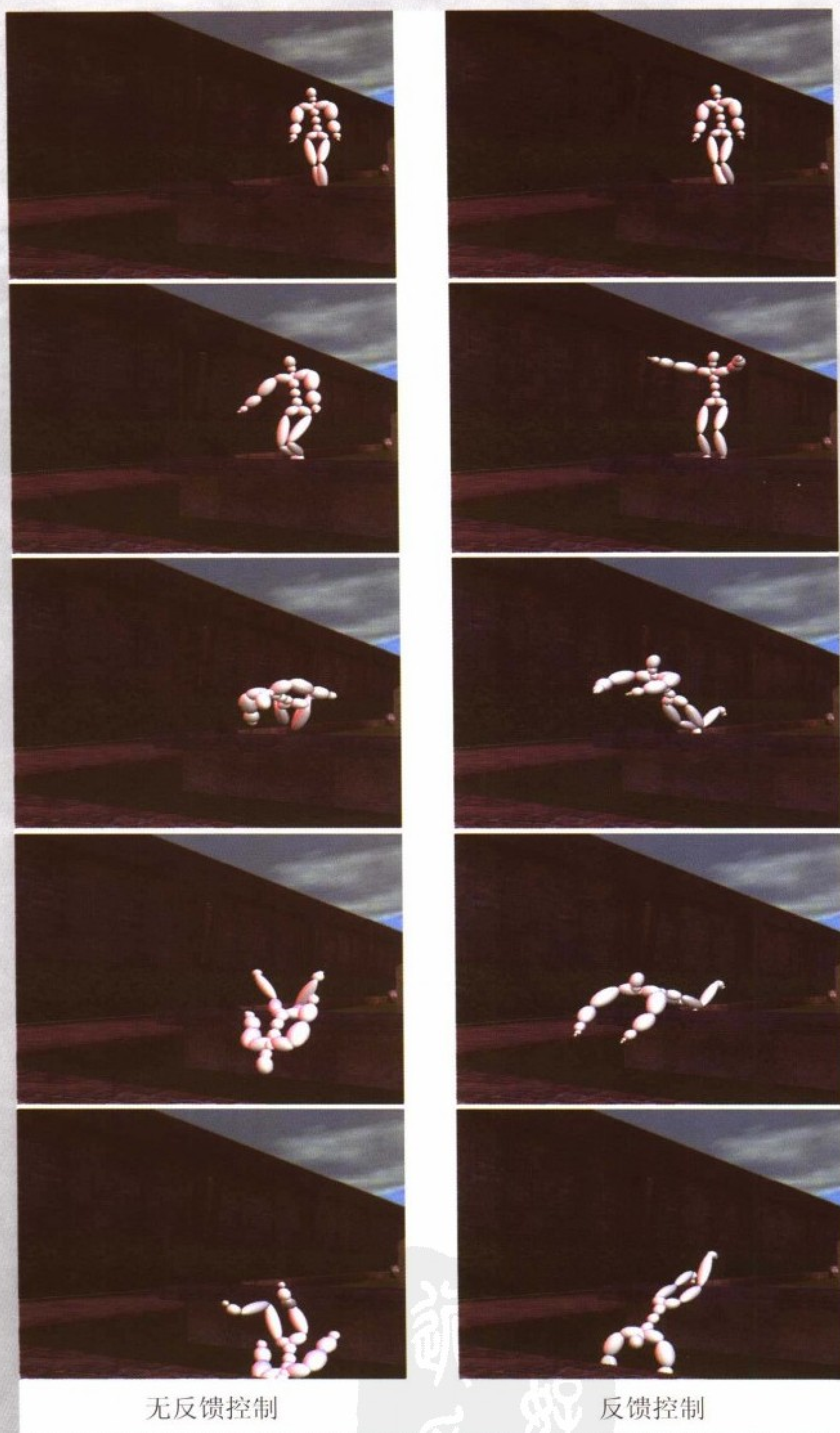
翻译6.8、7.3、7.4、7.5共4篇文章。目前就职于北京百竹数码。长期从事游戏引擎的研究与开发工作。编著有《PC游戏编程—窥门篇》以及《PC游戏编辑—基础篇》两本游戏编程图书。曾参与开发《1937特种兵》、《傲视三国2》、《荣耀》等游戏。目前负责开发一款大型网络游戏。

彩插1A　布料动画（见4.3节）

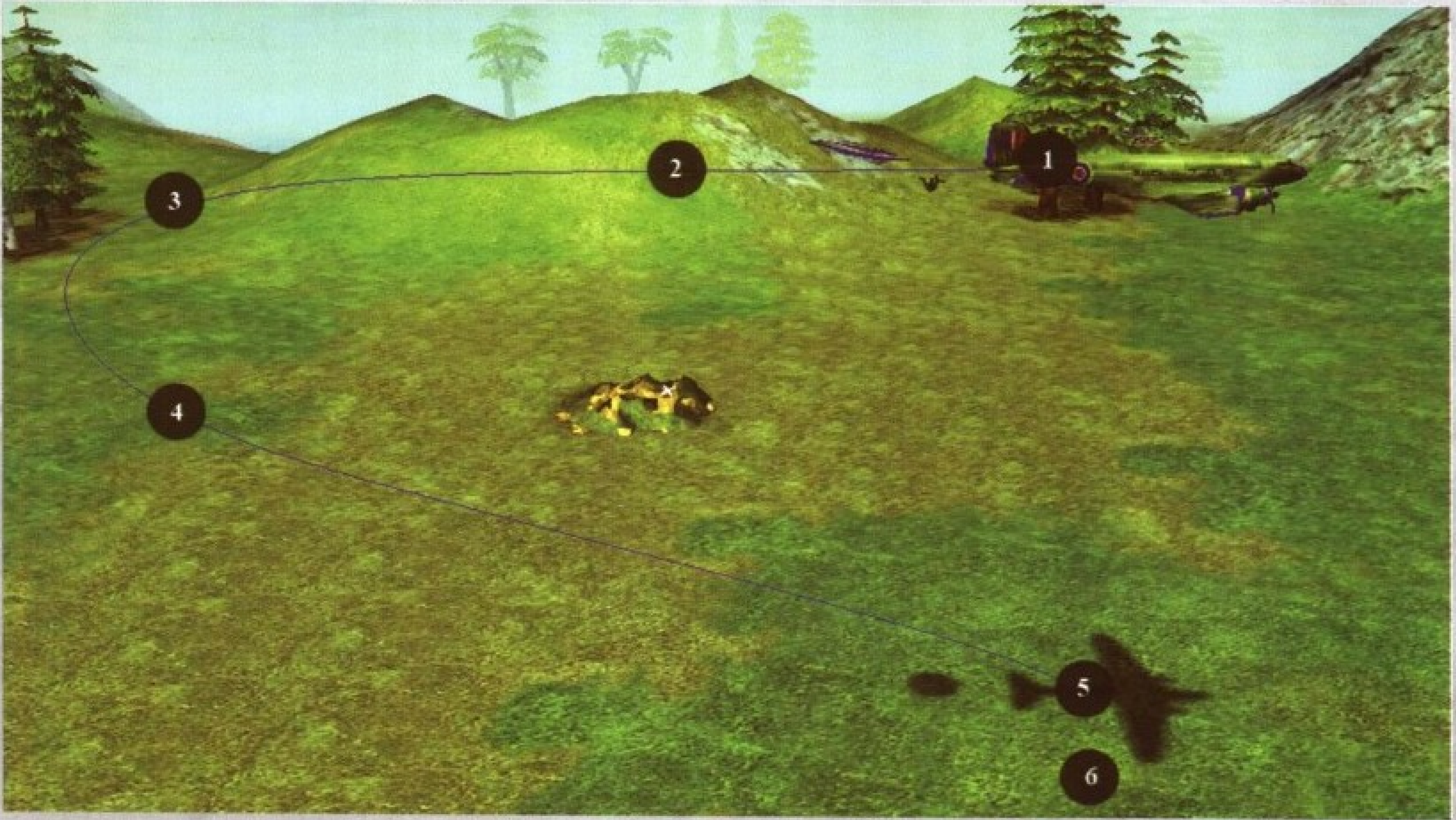
彩插1B　覆盖在球体上的布料动画（见4.3节）



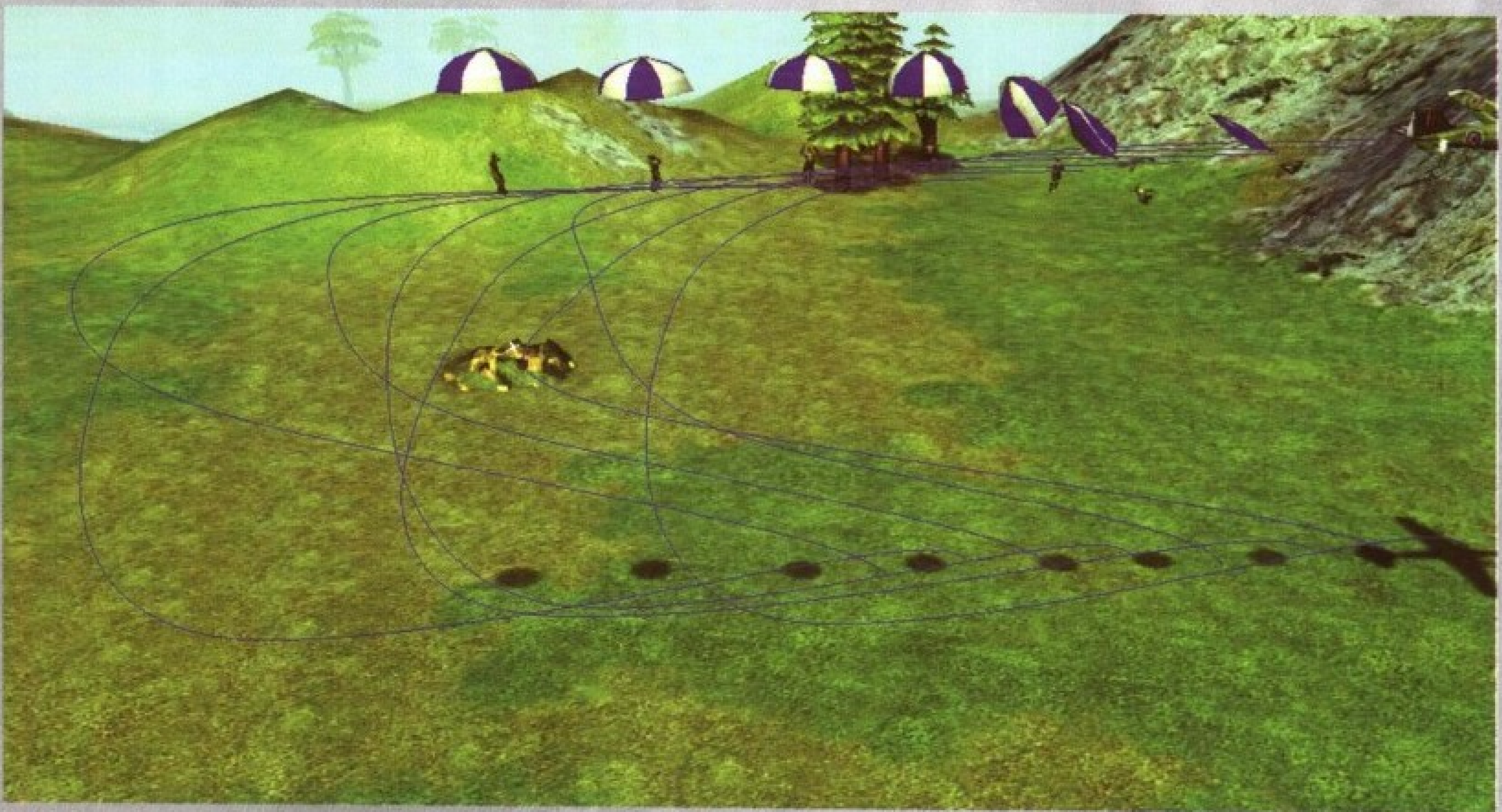
彩插2　实用柔体动画中可变形的兔子模型（见4.4节）



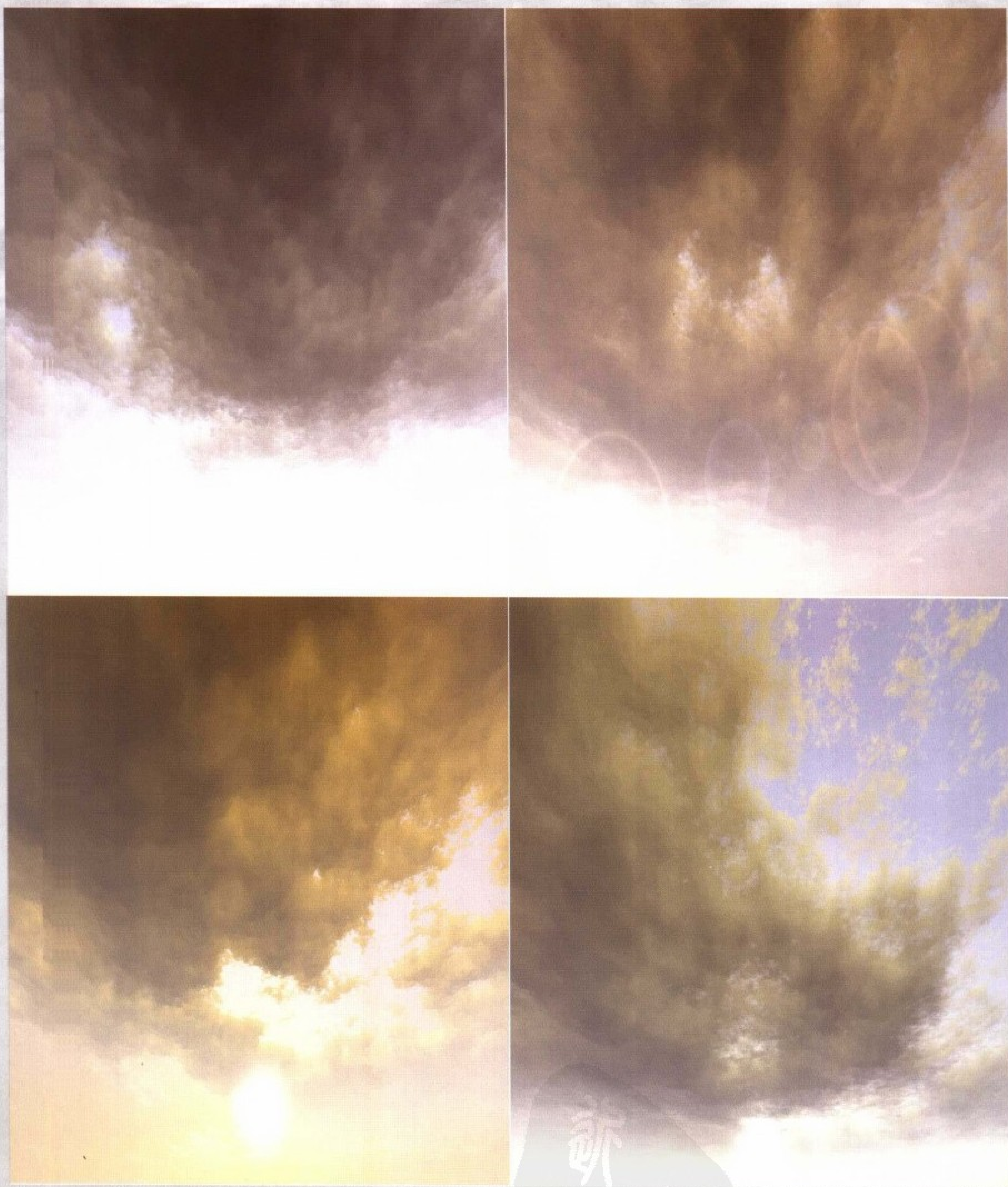
彩插3 由Ragdoll物理系统控制的两个角色以两种方式的摔跤动作。左图为没有反馈控制的情况，右图为有反馈控制的情况（见4.5节）



彩插 4A　预定式物理系统的应用（见4.7节）



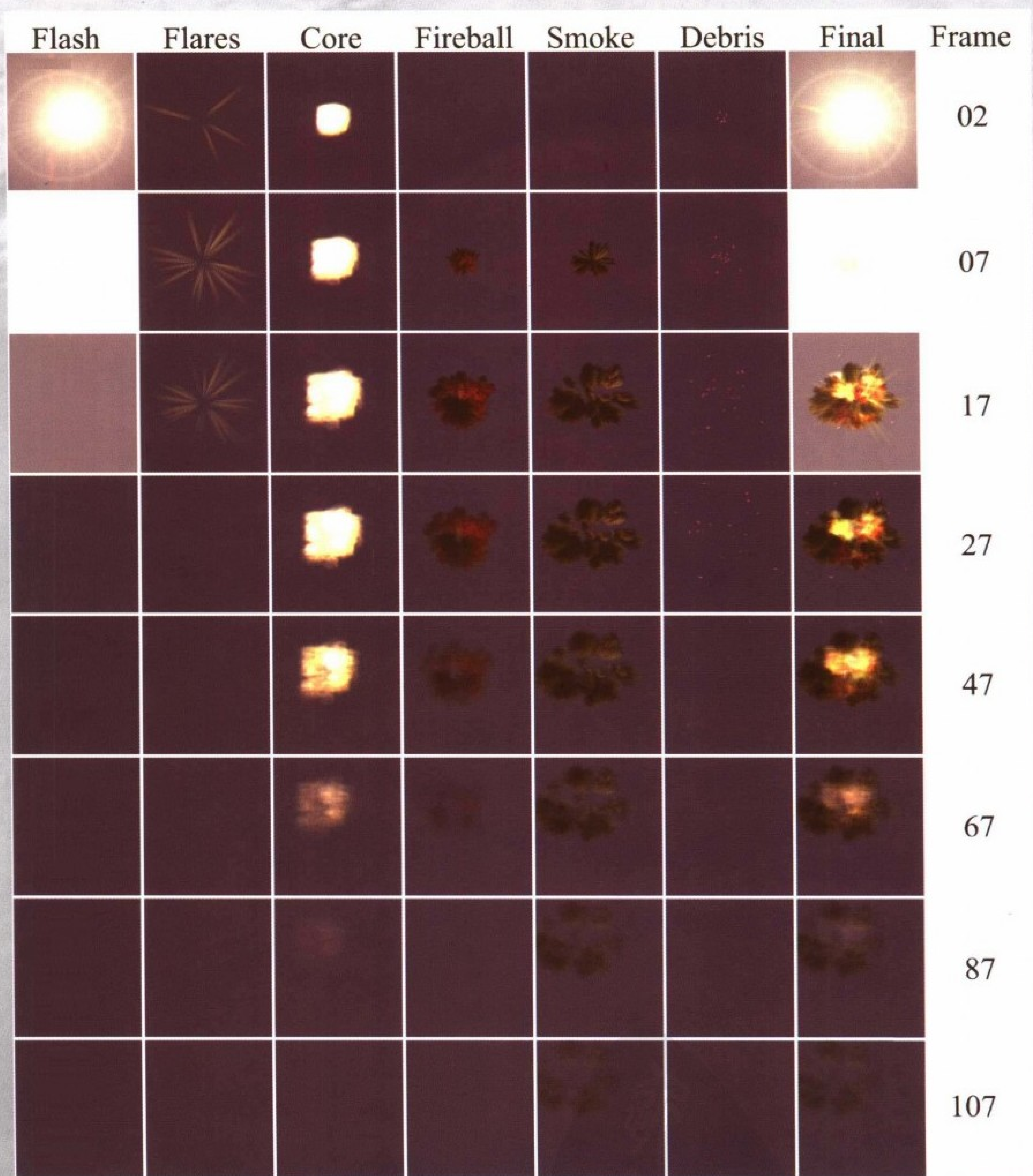
彩插 4B　预定式物理系统的应用（见4.7节）



彩插5 过程化云彩生成技术。这4幅图是在使用ps.3.0的GeForce6800系统上以640×480的分辨率，每秒60帧的速度渲染出来的。左上图是给高频倍频纹理更高的权重。右上图使用了典型的设置和镜头眩光。左下图中，我们选择了一个清晰度的值，让云彩更浓厚。右下图显示的是一些典型设置（见5.1节）

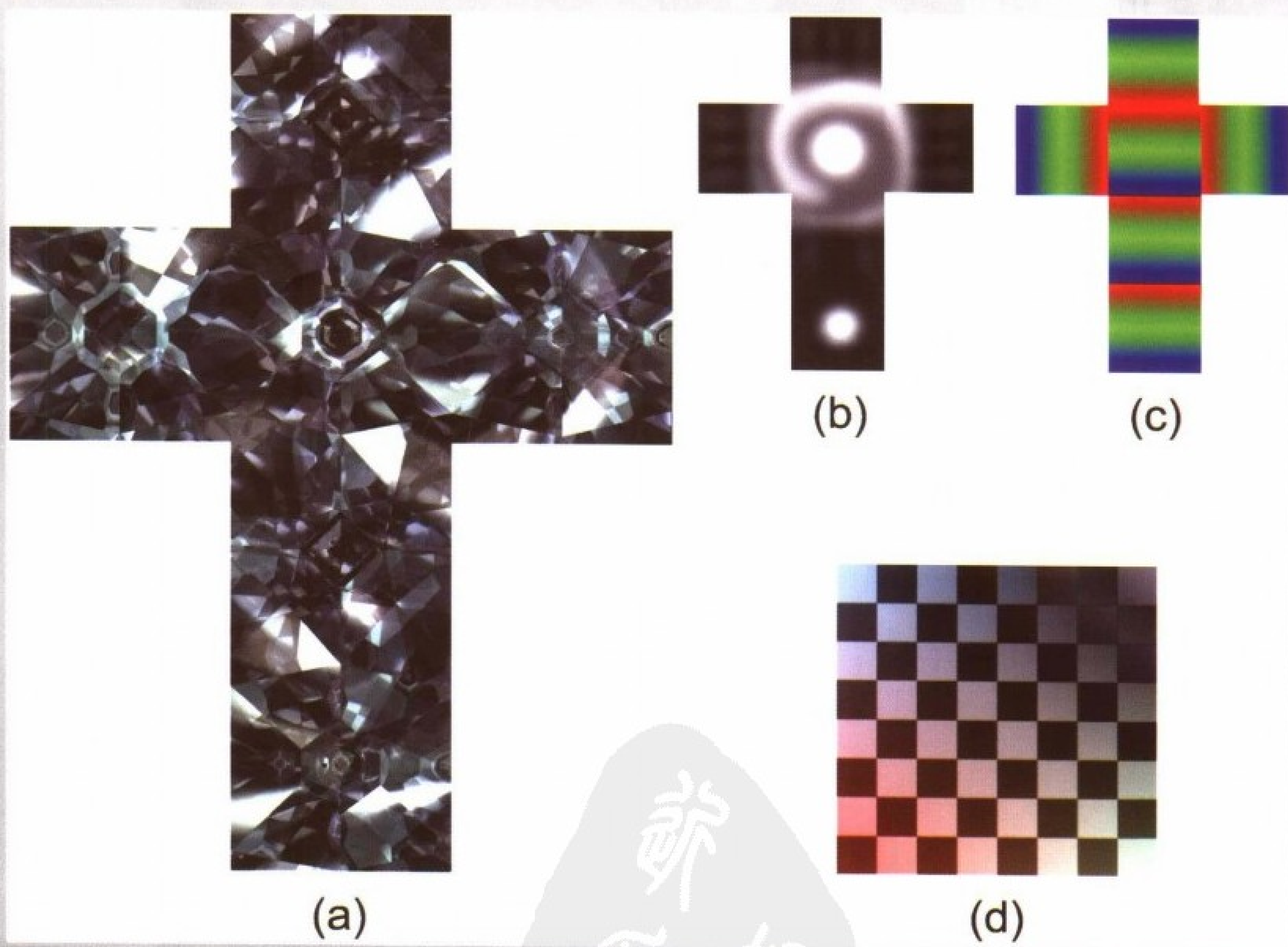
彩插6 画面中雪花高速运动（见5.5节）

彩插7 无栅格火焰技术（见5.5节）

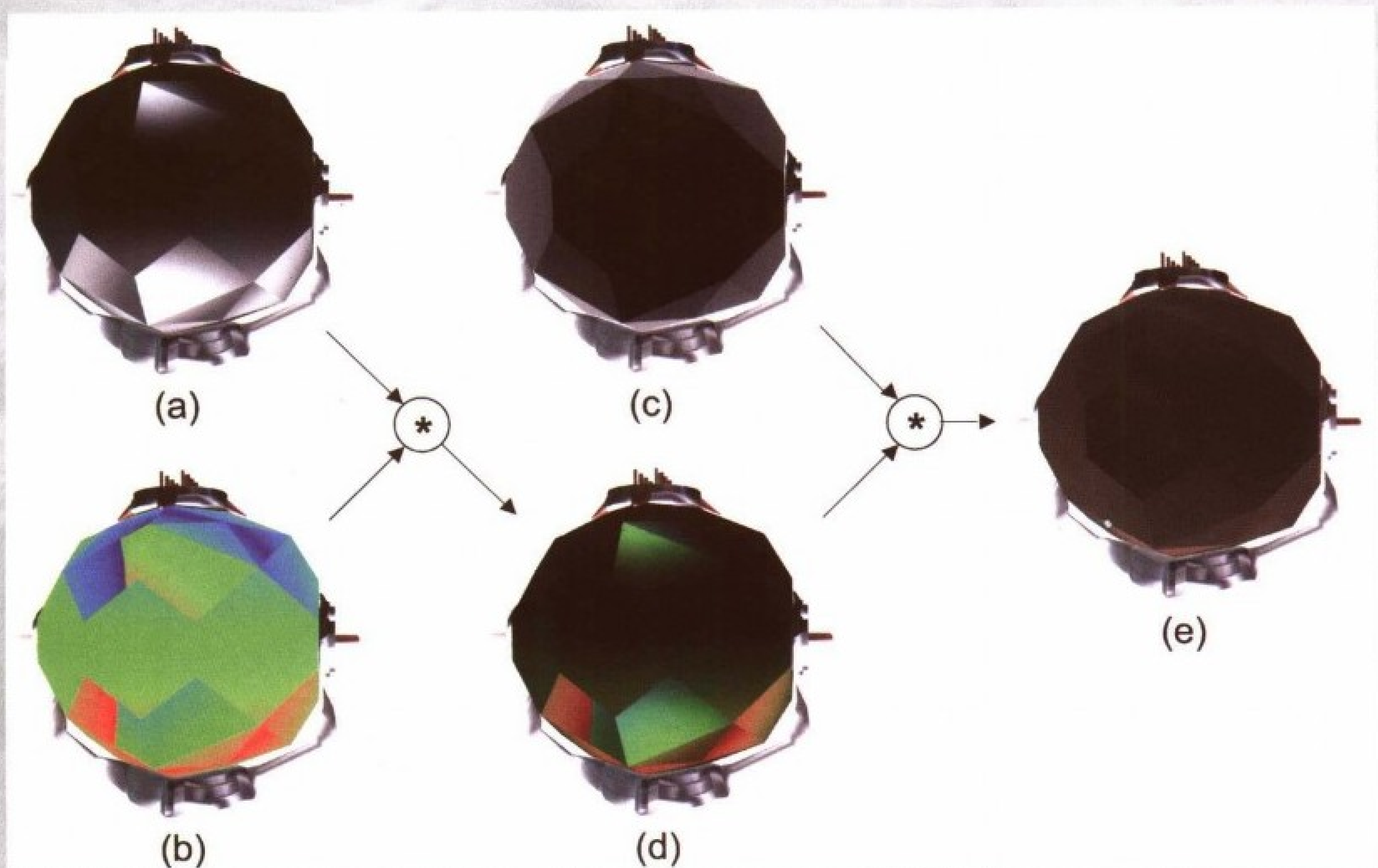


彩插8A　文章5.6所述技术中所使用的爆炸组件

彩插 8B 文章 5.6 中最后的爆炸特效



彩插 9A 文章 5.7 所述中宝石渲染技术所使用的组件



彩插 9B　文章5.7所述中宝石渲染技术所使用的组件

彩插 9C　ATI公司演示程序"Ruby: The Double Cross"的画面，参见文章5.7中的宝石渲染技术

目　　录

第1章　通用编程

引言 ········· 2

William E.Damon III

1.1　面向编辑器的上下文相关 HUD ········· 3

Grex 游戏公司，Adm Martin

1.1.1　问题 ········· 3

1.1.2　解决方案 ········· 6

1.1.3　实现 ········· 7

1.1.4　用户控制 ········· 10

1.1.5　总结 ········· 11

1.1.6　参考文献 ········· 11

1.2　在游戏中解析文本数据 ········· 12

Aurelio Reis

1.2.1　开始之前 ········· 12

1.2.2　token 到底是什么 ········· 12

1.2.3　编写词法分析器 ········· 13

1.2.4　工作原理 ········· 14

1.2.5　制定自己的格式 ········· 15

1.2.6　解析 token 列表 ········· 17

1.2.7　总结 ········· 18

1.2.8　参考文献 ········· 18

1.3　基于组件的对象管理 ········· 19

Circle Studio 公司，Bjarne Rene

1.3.1　除旧迎新 ········· 19

1.3.2　组件 ········· 20

1.3.3　系统的创建 ········· 23

1.3.4　总结 ········· 29

1.4　用模板实现一个可在 C++中使用的反射系统 ········· 30

Artificial Mind & Movement 公司，Dominic Fillion

1.4.1　需求 ········· 31

1.4.2　第 1 部分：运行时类型信息 ········· 31

1.4.3　在 RTTI 的实现中使用模板 ········· 33

1.4.4　关于 RTTI 的其他修改建议 ········· 35

1.4.5　第 2 部分：属性对象 ········· 36

1.4.6　属性的存储 ········· 38

1.4.7　属性类型 ········· 38

1.4.8 属性注册钩子（Hook）函数 …… 39
1.4.9 属性的注册 …… 40
1.4.10 脚本应用 …… 41
1.4.11 Tweaker 应用 …… 42
1.4.12 其他应用 …… 42
1.4.13 总结 …… 42
1.4.14 参考文献 …… 43
1.5 可加速 BSP 算法的球体树 …… 44
Artificial Mind & Movement 公司，Dominic Filion
1.5.1 BSP 算法 …… 44
1.5.2 创建 BSP 树 …… 45
1.5.3 优化最初步骤 …… 46
1.5.4 总结 …… 51
1.5.5 参考文献 …… 51
1.6 改进后的视锥剔除算法 …… 52
Frank Puig Placeres
1.6.1 视锥剔除 …… 52
1.6.2 传统的六面法 …… 53
1.6.3 雷达法 …… 54
1.6.4 这个点在视锥内部吗？ …… 54
1.6.5 球体在哪里？ …… 56
1.6.6 其他应用 …… 57
1.6.7 进一步的改造 …… 58
1.6.8 总结 …… 60
1.6.9 参考文献 …… 60
1.7 通用的分页管理系统 …… 61
Ignacio Incera Cruz
1.7.1 老式的分页解决方案：一查到底 …… 61
1.7.2 GP 分页解决方案：只检查需要的 …… 62
1.7.3 索引是关键 …… 62
1.7.4 GPtile：空间中的块 …… 65
1.7.5 The world：搜索空间 …… 67
1.7.6 窗口：在 GPworld 中航行 …… 69
1.7.7 多窗口，多用户 …… 70
1.7.8 优化：多线程分页 …… 71
1.7.9 总结 …… 71
1.7.10 参考文献 …… 71
1.8 基于栈的大规模状态机 …… 72
James Boer

1.8.1 传统状态机编码及相关问题 ······ 72
1.8.2 用 C++方法解决游戏状态难题 ······ 74
1.8.3 状态接口类 ······ 75
1.8.4 状态的堆叠管理：为什么三维比二维好用 ······ 75
1.8.5 状态对象管理系统 ······ 76
1.8.6 总结 ······ 78
1.8.7 参考文献 ······ 79

1.9 使用 BSP 树构造 CSG 几何体 ······ 80

Octavian Marius Chincisan

1.9.1 CSG 的布尔运算 ······ 80
1.9.2 为什么要使用 BSP 树 ······ 84
1.9.3 BSP 树的实现 ······ 85
1.9.4 组合装配 ······ 86
1.9.5 总结 ······ 88
1.9.6 参考文献 ······ 89

1.10 在游戏中集成 Lua ······ 90

eV Interactive 公司，Matthew Harmon

1.10.1 Lua 的概况 ······ 90
1.10.2 Lua 与 C 语言的接口 ······ 92
1.10.3 在游戏中嵌入 Lua ······ 94
1.10.4 实时性方面的考虑 ······ 97
1.10.5 脚本管理框架 ······ 99
1.10.6 总结 ······ 102
1.10.7 参考文献 ······ 102

1.11 用基于 policy 的设计改进 Freelist ······ 103

Nathan Mefford

1.11.1 Freelist 概述 ······ 103
1.11.2 Policy:雷霆救兵 ······ 104
1.11.3 分解 Freelist ······ 106
1.11.4 实现 Freelist：这是它吗？ ······ 107
1.11.5 选择最佳的 policy ······ 109
1.11.6 可能性 ······ 111
1.11.7 总结 ······ 113
1.11.8 参考文献 ······ 113

1.12 实时远程调试信息日志生成器 ······ 114

Microids Canada 公司，Patrick Duquette

1.12.1 对标准化的调试日志的需求 ······ 114
1.12.2 数据表示：你可看到我所看到的 ······ 115
1.12.3 本文提议的解决方案 ······ 115

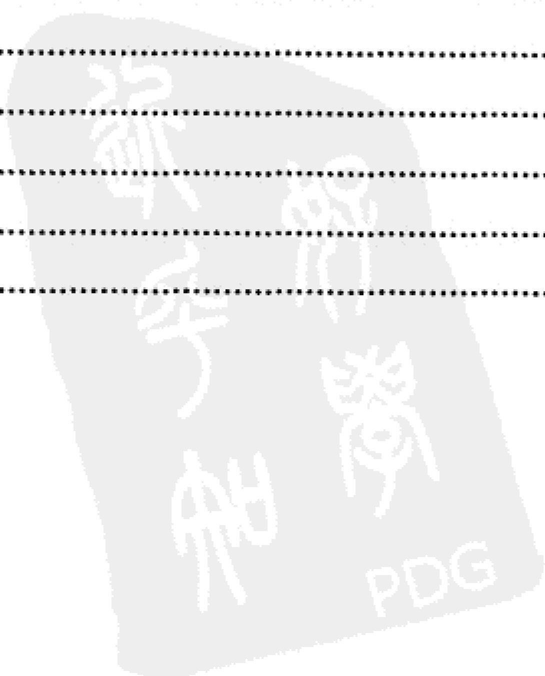
1.12.4 游戏日志模块 …… 117
1.12.5 可能的改进和扩展 …… 118
1.12.6 总结 …… 118
1.12.7 参考文献 …… 118
1.13 透明的类的保存和加载技巧 …… 119
Patrick Meehan
1.13.1 小窍门 …… 119
1.13.2 FreezeMgr 的实现 …… 120
1.13.3 其他几个特性 …… 124
1.13.4 如何使用范例 …… 125
1.13.5 总结 …… 126
1.13.6 参考文献 …… 126
1.14 高效且忽略缓存的 ABT 树实现方法 …… 128
瑞士联邦理工学院（Swiss Federal Institute of Technology，简称 EPFL），虚拟现实实验室（virtual Reality Lab，简称 VRLab），Sébastien Schertenleib
1.14.1 计算机内存结构 …… 128
1.14.2 ABT 树 …… 129
1.14.3 确认阶段 …… 134
1.14.4 总结 …… 134
1.14.5 参考文献 …… 134
1.15 状态机的可视化设计 …… 136
Scott Jacobs
1.15.1 为什么需要代码生成 …… 136
1.15.2 让“可视”成为可能 …… 137
1.15.3 状态的管理 …… 138
1.15.4 系统组装 …… 138
1.15.5 总结 …… 141
1.15.6 参考文献 …… 141
1.16 泛型组件库 …… 142
Warrick Buchanan
1.16.1 类型识别系统 …… 142
1.16.2 工厂 …… 143
1.16.3 工厂单例与子工厂 …… 145
1.16.4 DLL 工厂 …… 145
1.16.5 组件 …… 146
1.16.6 组件接口 …… 147
1.16.7 接口版本管理 …… 147
1.16.8 定义组件及其接口 …… 149
1.16.9 组件的使用 …… 150

PDG

1.16.10 配置组件库 …… 151
1.16.11 总结 …… 151
1.16.12 参考文献 …… 151
1.17 选择自己的路线——菜单系统 …… 152
Wendy Jones
1.17.1 为什么需要菜单系统 …… 152
1.17.2 菜单系统的对象 …… 153
1.17.3 总结 …… 158
1.17.4 参考文献 …… 159

第2章 数 学

引言 …… 162
Naughty Dog 公司，Eric Lengyel
2.1 在计算机图形学中使用几何代数 …… 163
Chris Lomont
2.1.1 引言 …… 163
2.1.2 几何代数 …… 164
2.1.3 线性代数 …… 169
2.1.4 词典 …… 171
2.1.5 实例 …… 172
2.1.6 总结以及将来的方向 …… 176
2.1.7 参考文献 …… 177
2.2 最小加速度 Hermite 曲线 …… 178
Tony Barrera, Barrera Kristiansen AB；Gävle 大学创意媒体实验室 Anders Hast；乌普萨拉大学图像分析中心, Ewert Bengtsson
2.2.1 连接具有 C^1 连续的最小弯曲曲线 …… 180
2.2.2 封闭的最小弯曲的曲线 …… 181
2.2.3 总结 …… 182
2.2.4 参考文献 …… 182
2.3 动画中基于样条的时间控制 …… 183
Red Storm Entertainment 公司，James M. Van Verth
2.3.1 开始 …… 183
2.3.2 一般的距离-时间函数 …… 184
2.3.3 根据样条构造距离-时间函数 …… 185
2.3.4 接口选择 …… 191
2.3.5 总结 …… 191
2.3.6 参考文献 …… 191
2.4 快速四元数近似插值 …… 193
Andy Thomason



2.4.1 使用四元数来表示旋转 ······ 193
2.4.2 四元数旋转插值 ······ 195
2.4.3 近似算法 ······ 196
2.4.4 算法之间的比较 ······ 206
2.4.5 Squad 相关的计算 ······ 207
2.4.6 延伸阅读 ······ 208
2.4.7 总结 ······ 208
2.4.8 参考文献 ······ 208

2.5 极小极大数值近似 ······ 209

Christopher Tremblay

2.5.1 众所周知的优化 ······ 209
2.5.2 什么是理想的近似 ······ 210
2.5.3 极小极大近似的介绍 ······ 211
2.5.4 误差分析 ······ 214
2.5.5 进一步改进近似 ······ 215
2.5.6 参考文献 ······ 216

2.6 应用于镜面和入口的斜视锥 ······ 217

Eric Lengyel

2.6.1 平面的表示 ······ 217
2.6.2 投影矩阵 ······ 218
2.6.3 裁剪面的修改 ······ 219
2.6.4 OpenGL 实现 ······ 221
2.6.5 Direct3D 实现 ······ 224
2.6.6 致谢 ······ 225
2.6.7 参考文献 ······ 225

第 3 章 人工智能

引言 ······ 228

美国西北大学，Robin Hunicke

3.1 利用导航网格实现自动掩体寻找 ······ 230

Radical Entertainment 公司，Borut Pfeifer

3.1.1 导航网格 ······ 230
3.1.2 开放目标寻路 ······ 231
3.1.3 搜索掩体位置 ······ 232
3.1.4 在掩体间行进 ······ 233
3.1.5 团队掩护行为 ······ 234
3.1.6 其他功能 ······ 235
3.1.7 总结 ······ 235
3.1.8 参考文献 ······ 236

3.2 使用人工势场实现快速目标评级 ······ 237
Factor 5 公司，Markus Breyer
3.2.1 基本思想 ······ 237
3.2.2 公式 ······ 238
3.2.3 势值函数的评估 ······ 240
3.2.4 可视化 ······ 240
3.2.5 方向场的应用 ······ 241
3.2.6 多维扩展 ······ 242
3.2.7 总结 ······ 243
3.3 利用 Lanchester 损耗模型来预测战斗结果 ······ 244
Page 44 Studios 有限责任公司，John Bolton
3.3.1 概述 ······ 244
3.3.2 场景 1：全体混战 ······ 245
3.3.3 场景 2：狭窄的石阶 ······ 247
3.3.4 场景 3：炮战 ······ 248
3.3.5 场景 4：关底 Boss ······ 250
3.3.6 关于战斗力的再讨论 ······ 251
3.3.7 局限性 ······ 252
3.3.8 总结 ······ 252
3.3.9 参考文献 ······ 252
3.4 为游戏 AI 实现一个实用的智能规划系统 ······ 254
Relic Entertainment 公司，Jamie Cheng;加拿大阿尔伯塔大学计算机科学系，Finnegan Southey
3.4.1 规划系统的框架 ······ 255
3.4.2 规划域 ······ 255
3.4.3 一个多主体规划器的例子 ······ 258
3.4.4 规划的搜索 ······ 262
3.4.5 几个应用问题 ······ 263
3.4.6 优化 ······ 265
3.4.7 总结 ······ 266
3.4.8 参考文献 ······ 266
3.5 针对多线程架构的决策树查询算法优化 ······ 268
Intel 公司，Chuck DeSylva
3.5.1 概述 ······ 268
3.5.2 注意事项 ······ 269
3.5.3 优化 ······ 270
3.5.4 总结 ······ 273
3.5.5 参考文献 ······ 273
3.6 利用并行虚拟机实现 AI 系统的并行开发 ······ 275

2015 公司，Michael Ramsey
3.6.1 功能强大，但不白给 ……275
3.6.2 核心术语及概念 ……276
3.6.3 任务的创建 ……277
3.6.4 任务管理 ……279
3.6.5 PVM 的实现 ……282
3.6.6 实际应用：即时战略游戏 ……283
3.6.7 强化游戏性 ……284
3.6.8 总结 ……285
3.6.9 参考文献 ……286
3.7 超越 A*算法 ……287
Xtrem Strategy 游戏公司，Mario Grimani；Monolith Productions 公司，Matthew Titelbaum
3.7.1 问题的定义 ……287
3.7.2 算法 ……288
3.7.3 算法的改进 ……290
3.7.4 实现的细节 ……292
3.7.5 应用实例 ……292
3.7.6 性能方面的考虑 ……297
3.7.7 几个前沿问题 ……298
3.7.8 总结 ……299
3.7.9 参考文献 ……299
3.8 实现最小重新规划开销的先进寻路算法：动态 A*（D*）算法 ……301
Marco Tombesi
3.8.1 D*算法 ……302
3.8.2 D*算法的实现细节 ……302
3.8.3 实例 ……303
3.8.4 在游戏中又如何呢？ ……305
3.8.5 总结 ……305
3.8.6 参考文献 ……305

第 4 章 物理学

引言 ……308
Red Storm 娱乐公司，Mike Dickheiser
4.1 游戏物理中空气动力学的近似计算 ……310
美国应用研究联营公司（Applied Research Associates Inc.），Graham Rhodes
4.1.1 背景知识 ……310
4.1.2 钝体上的作用力 ……313
4.1.3 流线体上的作用力 ……315
4.1.4 应用实例 ……318

4.1.5 总结 ······ 319
4.1.6 参考文献 ······ 320
4.2 动态青草的模拟和其他自然环境特效 ······ 321
沃特卢大学，Rishi Ramraj
4.2.1 水面特效 ······ 321
4.2.2 青草的模拟 ······ 323
4.2.3 变化传播模型 ······ 324
4.2.4 树叶的模拟：模型的应用 ······ 325
4.2.5 总结 ······ 327
4.2.6 参考文献 ······ 327
4.3 使用质点-弹簧模型获得真实的布料动画 ······ 328
塞维利亚大学，Juan M. Cordero
4.3.1 布料的离散表示 ······ 328
4.3.2 作用力 ······ 330
4.3.3 动态系统方法 ······ 333
4.3.4 仿真模拟 ······ 334
4.3.5 结论 ······ 334
4.3.6 参考文献 ······ 335
4.4 适合游戏开发的实用柔体动画技术：受压柔体模型 ······ 336
波兰弗罗茨瓦夫大学，Maciej Matyka
4.4.1 简化的质点-弹簧模型 ······ 337
4.4.2 PSB 模型背后的物理学 ······ 338
4.4.3 PSB 模型的实现 ······ 339
4.4.4 典型的质点-弹簧模型 ······ 340
4.4.5 PSB 步骤 ······ 341
4.4.6 体积计算 ······ 341
4.4.7 采用预估修正法的 Heun 积分 ······ 342
4.4.8 时间步长的计算速度 ······ 342
4.4.9 几个仿真实例 ······ 343
4.4.10 进一步的发展 ······ 344
4.4.11 总结 ······ 344
4.4.12 源代码说明 ······ 345
4.4.13 致谢 ······ 345
4.4.14 参考文献 ······ 345
4.5 使用反馈控制系统让“布娃娃”活起来 ······ 347
苹果公司，Michael Mandel
4.5.1 现有的研究成果 ······ 347
4.5.2 仿真过程的控制 ······ 348
4.5.3 行为动作的创建 ······ 350

4.5.4 总结 ······ 351
4.5.5 致谢 ······ 351
4.5.6 参考文献 ······ 351

4.6 预定式物理系统的设计 ······ 353

Daniel F.Higgins

4.6.1 什么是预定式物理系统 ······ 353
4.6.2 预定式物理引擎 ······ 356
4.6.3 打磨上光 ······ 360
4.6.4 龙卷风：一个好的开始 ······ 363
4.6.5 总结 ······ 365
4.6.6 参考文献 ······ 366

4.7 预定式物理系统：相关技术及应用 ······ 367

Shawn Shoemaker

4.7.1 为什么要使用预定式物理系统 ······ 367
4.7.2 预定式物理系统 ······ 368
4.7.3 应用 1：RTS 游戏中建筑物的毁坏 ······ 369
4.7.4 应用 2：跳跃 ······ 369
4.7.5 应用 3：爆炸特效中的物体运动 ······ 370
4.7.6 应用 4：浮力 ······ 371
4.7.7 应用 5：伞兵 ······ 371
4.7.8 总结 ······ 373
4.7.9 参考文献 ······ 373

4.8 三维汽车模拟器中真实的摄像机运动 ······ 374

匈牙利布达佩斯技术经济大学，控制工程与信息技术系，图形图像小组，

Barnabás Aszódi，Szabolcs Czuczor

4.8.1 我们需要什么？物理法则 ······ 374
4.8.2 我们得到的是什么？偶尔不够真实的运动 ······ 375
4.8.3 考察摄像机的控制 ······ 376
4.8.4 终极决策：实现人类的行为 ······ 377
4.8.5 编程中涉及的问题 ······ 379
4.8.6 总结 ······ 382
4.8.7 关于演示程序 ······ 382

第 5 章 图形图像

引言 ······ 384

ATI 公司，Jason L. Mitchell

5.1 在现代 GPU 上渲染逼真的云彩 ······ 386

育碧公司，Jean-François Dubé

5.1.1 制造噪音 ······ 386

PDG

5.1.2 云彩的密度 ······ 388
5.1.3 云彩的光照处理 ······ 389
5.1.4 优化 ······ 390
5.1.5 总结 ······ 390
5.1.6 参考文献 ······ 391

5.2 下雪吧，下雪吧，下雪吧（下雨吧） ······ 392

微软公司(现就职于 Google 公司)，Niniane Wang; 微软公司, Bretton Wade

5.2.1 使用纹理为粒子束建模 ······ 393
5.2.2 渲染雪或雨的视差 ······ 393
5.2.3 用锥体模拟摄像机的移动 ······ 394
5.2.4 合并到一个矩阵中 ······ 394
5.2.5 增加美工控制 ······ 395
5.2.6 总结 ······ 395
5.2.7 参考文献 ······ 395

5.3 Widget：快速渲染和持久化小物体 ······ 396

Martin Brownlow

5.3.1 Widget 的网格 ······ 396
5.3.2 高效地绘制 widget ······ 397
5.3.3 裁剪 widget ······ 401
5.3.4 总结 ······ 403
5.3.5 参考文献 ······ 404

5.4 逼真的树木和森林的 2.5 维替用物 ······ 405

布达佩斯理工大学, Gábor Szijártó

5.4.1 引言 ······ 405
5.4.2 以前的基于图像的方法 ······ 406
5.4.3 改进以前的方法 ······ 407
5.4.4 算法 ······ 408
5.4.5 实现 ······ 409
5.4.6 总结 ······ 412
5.4.7 参考文献 ······ 413

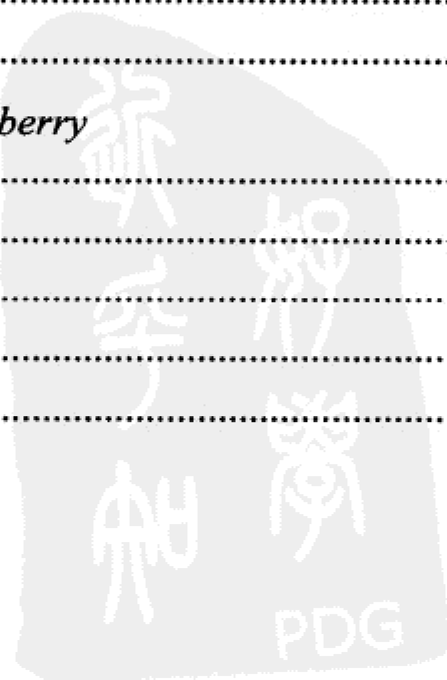
5.5 无栅格的可控火焰 ······ 414

佛罗里达中心大学，计算机科学学院，Neeharika Adabala；
佛罗里达中心大学，计算机科学学院和电影与数字媒体学院，Charles E. Hughes

5.5.1 建模火焰 ······ 415
5.5.2 实时渲染 ······ 418
5.5.3 实例和讨论 ······ 419
5.5.4 总结 ······ 420
5.5.5 参考文献 ······ 420

5.6 使用公告牌粒子构建强大的爆炸效果 ······ 422

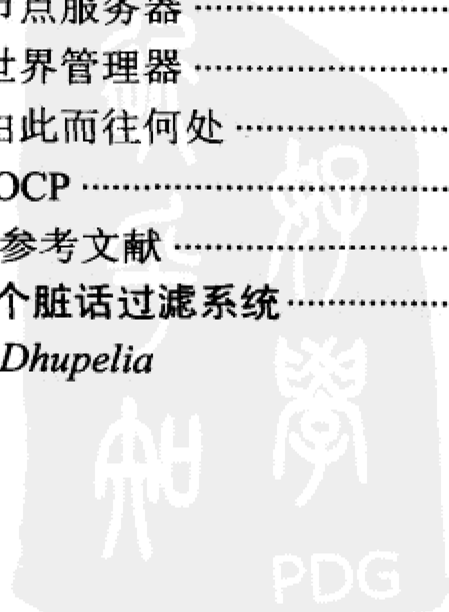
美国任天堂公司，Steve Rabin
5.6.1 最初的闪光 …… 422
5.6.2 放射的火苗 …… 423
5.6.3 白色的热核心 …… 424
5.6.4 强烈的火球 …… 424
5.6.5 散发的烟雾 …… 425
5.6.6 碎片 …… 426
5.6.7 效果表 …… 426
5.6.8 额外的感觉 …… 426
5.6.9 效率问题 …… 427
5.6.10 总结 …… 428
5.6.11 参考文献 …… 428
5.7 渲染宝石的简单方法 …… 429
ATI 研究院公司，Thorsten Scheuermann
5.7.1 技术概览 …… 429
5.7.2 法线和 cubemap 采样问题 …… 430
5.7.3 传递的光能 …… 430
5.7.4 反射 …… 432
5.7.5 光斑 …… 433
5.7.6 总结 …… 436
5.7.7 参考文献 …… 436
5.8 体积化的后期处理 …… 437
A2M 公司，Dominic Filion；摩托罗拉公司，Sylvain Boissé
5.8.1 体积化的后期处理 …… 437
5.8.2 深度知识 …… 438
5.8.3 使用 shader 作 *z* 比较 …… 438
5.8.4 像素完美的裁剪 …… 439
5.8.5 后期处理 …… 440
5.8.6 最后一遍 …… 440
5.8.7 多个体 …… 440
5.8.8 总结 …… 441
5.8.9 参考文献 …… 441
5.9 过程式关卡生成 …… 442
北得克萨斯大学，Timothy Roden 和 Ian Parberry
5.9.1 大致的方法 …… 442
5.9.2 关卡设计 …… 442
5.9.3 使用预制的几何体 …… 443
5.9.4 图的生成 …… 444
5.9.5 把预制件映射到图中 …… 446



5.9.6 可见性和碰撞检测 …… 448
5.9.7 增加关卡内容 …… 448
5.9.8 总结 …… 449
5.9.9 参考文献 …… 449
5.10 重组 shader …… 450
A2M 公司，Dominic Filion
5.10.1 组合效果 …… 450
5.10.2 处理组合爆炸 …… 451
5.10.3 通过 HLSL 生成 shader 变体 …… 452
5.10.4 整合重组的 shader …… 454
5.10.5 通过 shader 建立完整的流水线 …… 455
5.10.6 其他问题 …… 455
5.10.7 总结 …… 456
5.10.8 参考文献 …… 456

第 6 章 网络和多玩家

引言 …… 460
Shekhar Dhupelia
6.1 保持大型多人在线游戏大型、在线和永存 …… 461
Tantrum 游戏公司，Shea Street
6.1.1 快速浏览 …… 461
6.1.2 大型化 …… 462
6.1.3 保持在线 …… 464
6.1.4 保持永存 …… 465
6.1.5 总结 …… 466
6.1.6 参考文献 …… 466
6.2 实现一个无缝的世界服务器 …… 467
育碧公司，Patrick Duquette
6.2.1 一些定义 …… 467
6.2.2 实现 …… 468
6.2.3 远程控制器，或如何管理服务器的启动时期 …… 468
6.2.4 代理服务器 …… 468
6.2.5 登录服务器 …… 469
6.2.6 节点服务器 …… 469
6.2.7 世界管理器 …… 472
6.2.8 由此而往何处 …… 473
6.2.9 IOCP …… 473
6.2.10 参考文献 …… 473
6.3 设计一个脏话过滤系统 …… 475
Shekhar Dhupelia



6.3.1 语法与内容 …… 475
6.3.2 字典 …… 475
6.3.3 解析器 …… 476
6.3.4 过滤 …… 476
6.3.5 最好的过滤实践 …… 477
6.3.6 人工干涉 …… 478
6.3.7 总结 …… 478
6.3.8 参考文献 …… 478

6.4 远程过程调用系统的快速和高效实现 …… 480

Hyun-jik Bae

6.4.1 RPC：简介 …… 483
6.4.2 RPC：设计 …… 484
6.4.3 RPC：实现 …… 486
6.4.4 RPC：使用 …… 489
6.4.5 样例程序 …… 490
6.4.6 更多特性 …… 490
6.4.7 总结 …… 492
6.4.8 参考文献 …… 492

6.5 在对等通信中克服网络地址转换 …… 493

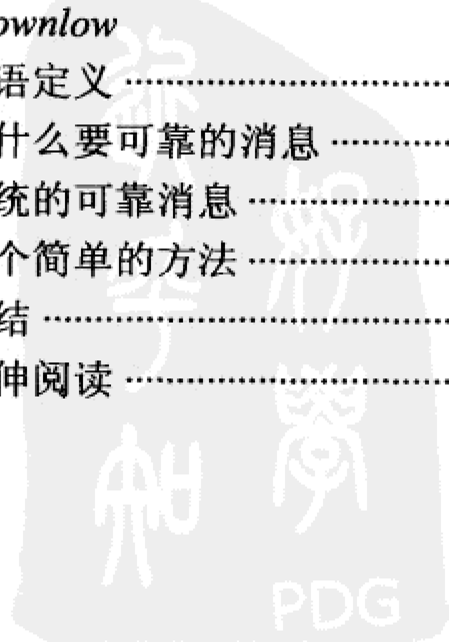
Jon Watte

6.5.1 读者 …… 493
6.5.2 IP 地址 …… 494
6.5.3 套接字使用 …… 494
6.5.4 路由器、点、协议 …… 495
6.5.5 UDP 包图 …… 496
6.5.6 什么是 NAT …… 496
6.5.7 NAT 是如何破坏客户/服务协议的 …… 499
6.5.8 NAT 是如何破坏对等协议的 …… 501
6.5.9 用于游戏的端到端解决方案 …… 503
6.5.10 总结 …… 509
6.5.11 参考文献 …… 509

6.6 一个可靠的消息协议 …… 510

Martin Brownlow

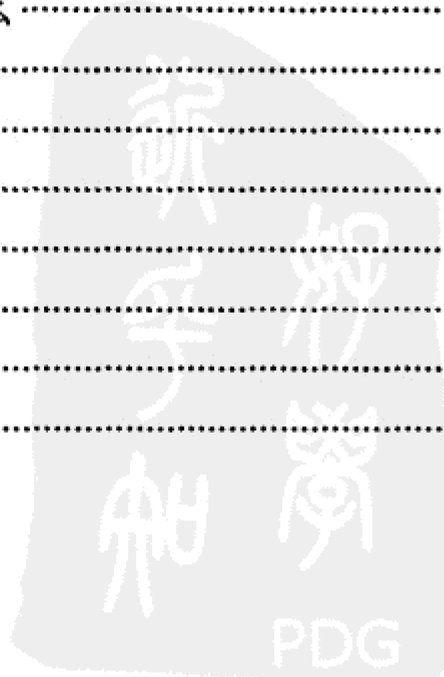
6.6.1 术语定义 …… 510
6.6.2 为什么要可靠的消息 …… 510
6.6.3 传统的可靠消息 …… 511
6.6.4 一个简单的方法 …… 511
6.6.5 总结 …… 514
6.6.6 延伸阅读 …… 515



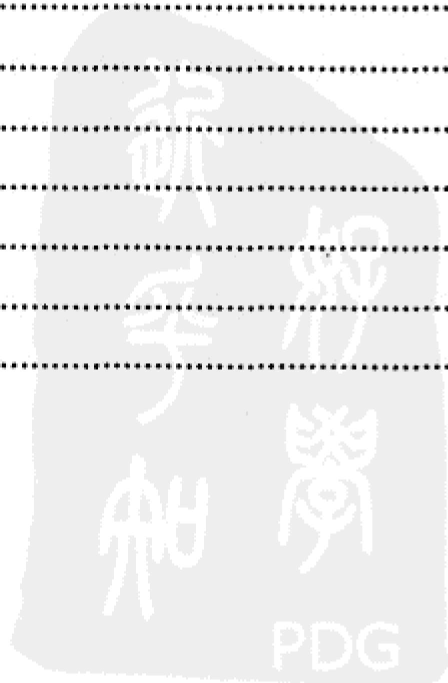
6.7 安全的随机数系统 ······ 516
Shekhar Dhupelia
6.7.1 随机数影响在线游戏 ······ 516
6.7.2 网络模型 ······ 517
6.7.3 随机数池 ······ 517
6.7.4 随机数发生器 ······ 518
6.7.5 重载标准的 rand()和 srand() ······ 520
6.7.6 下一步：记录和调试 ······ 520
6.7.7 下一步：即时重放 ······ 520
6.7.8 总结 ······ 521
6.7.9 参考文献 ······ 521
6.8 安全的设计 ······ 522
Adam Martin, Grex 游戏公司
6.8.1 安全问题真的如此重要么？ ······ 522
6.8.2 目标 ······ 523
6.8.3 术语 ······ 523
6.8.4 威胁模型：测试不安全性 ······ 524
6.8.5 安全策略：让威胁无效 ······ 527
6.8.6 同时修订两个文档 ······ 527
6.8.7 该技术的其他优势 ······ 528
6.8.8 延伸阅读 ······ 528
6.8.9 总结 ······ 529
6.8.10 参考文献 ······ 529

第7章 音 频

引言 ······ 532
Mark DeLoura
7.1 多线程音频编程技巧 ······ 533
James Boer
7.1.1 多线程编程简介 ······ 533
7.1.2 线程的术语和机制 ······ 534
7.1.3 鉴别适合多线程编程的音频任务 ······ 535
7.1.4 Intel 的超线程技术是什么 ······ 536
7.1.5 线程编程技巧和操作 ······ 536
7.1.6 多线程的示例程序 ······ 537
7.1.7 实时流数据机制 ······ 541
7.1.8 流和线程 ······ 541
7.1.9 总结 ······ 543
7.1.10 参考文献 ······ 543
7.2 基于组的声音管理 ······ 544


PDG

eV Interactive 公司，Matthew Harmon
7.2.1 API 包装预览 …… 544
7.2.2 能力 …… 545
7.2.3 定义组 …… 546
7.2.4 实现细节 …… 548
7.2.5 总结 …… 549
7.3 利用三维曲面作为音频发生器 …… 550
Sami hamlaoui
7.3.1 方法 …… 550
7.3.2 点发生器 …… 551
7.3.3 线发生器 …… 552
7.3.4 球体发生器 …… 553
7.3.5 方体发生器 …… 554
7.3.6 总结 …… 555
7.3.7 光盘上的内容 …… 555
7.3.8 参考文献 …… 555
7.4 基于回馈延迟网络（FDN）的快速环境反响 …… 556
Phenomic 游戏开发公司，Christian Schüler
7.4.1 怎样将 FDN 作为资源来利用 …… 556
7.4.2 什么是反响 …… 557
7.4.3 回馈延迟网络（Feedback Delay Network） …… 557
7.4.4 选择正确的回馈矩阵 …… 559
7.4.5 选择正确的延迟长度 …… 560
7.4.6 控制回响时间 …… 560
7.4.7 sweeping 和细部延迟问题 …… 561
7.4.8 总结和可能的改进 …… 563
7.4.9 感谢 …… 564
7.4.10 参考文献 …… 564
7.5 单演讲者语音识别简介 …… 566
Julien Hamaide
7.5.1 引言 …… 566
7.5.2 识别系统 …… 567
7.5.3 特征提取 …… 568
7.5.4 即时配对匹配 …… 570
7.5.5 训练 …… 571
7.5.6 局限性 …… 572
7.5.7 总结 …… 572
7.5.8 参考文献 …… 572



第 1 章

通用编程

PDG

引 言

William E.Damon Ⅲ
wdamon@ati.com

内容（背景美术、设置选项、位置、角色、剧情、背景故事，以及给游戏体验带来生气的打斗动作）决定着一款游戏是否会取得成功。对于游戏的制作人员来说，这就意味着他们要在一个相对固定的前提下去构建产品，并雇佣杰出的、受过专业训练的服务机构。大家首先想到的也许是信息技术部门，但这里所说的服务机构指的却是软件工程团队。

软件工程师可以解决问题。有些时候，游戏编程中遇到的一些问题需要我们花费大量的精力才能解决。但是，现在思路正在发生转变。游戏编程工作正在发展成为一个严格的工程实践，其目的就是为更广泛、更具可重用性的应用程序开发提供通用的解决方案。无论是采用或改编某个中间件技术，还是公司自己开发产品，这个解决方案都是适用的。这个发展的过程使得软件工程师们可以有更多的时间，专注于解决客户面临的实际问题。客户面临的问题是内容生成器（content creator），而不是一个陈旧方案的再改造。因此，内容生成器应该可以更便捷地在各种应用程序上运行，可以方便地调整和修改各种细节，以便在最终的产品中达到更高的质量。

本章所有的文章都贯穿着这条线索。无论讨论的是如何改进底层库文件以方便项目中其他工程师的使用，还是面向系统核心（游戏、编辑器或其他部分）的通用智能内存管理技巧、远程调试程序，以及/或者用来设计用户界面的可视化脚本环境，所有这些文章都是为了能够在产品制作线上提供可重用的功能组件。

本章中的文章涉及的主题很广，甚至包括面向一些常见重大问题的多种解决方案，目的是帮助大家为手头的游戏产品构建最好的工具集。作为一个无时无刻不在解决问题的工程师，如果能够充分理解、吸收这些文章的精髓，其中的内容总是会派上用场。

1.1 面向编辑器的上下文相关 HUD

Grex 游戏公司，Adm Martin
gpg@grexengine.com

大多数游戏产品都需要定制的图形图像工具，用来创建游戏内容（关卡布局和 AI 行为编辑等）。真正优秀的工具可以大大提高内容开发团队的效率，成倍地增加内容开发的数量。例如，一款好的关卡编辑工具可以非常容易地让关卡设计人员变得很“高产”，其效率等同于 3 个使用劣质工具的关卡设计人员。但是，优秀的工具一般都非常昂贵，我们难以判断，是自己开发的风险大，还是持续不断的维护费用更大？

本文采用非常简单的技巧，提供了一个抬头编辑（head-up editing）系统的实现方法。这个系统易于维护，可以大大降低公司自己开发编辑器的费用和难度。它可以在工具开发过程的早期阶段提高终端用户（游戏设计人员）的生产效率，降低项目风险。

1.1.1 问题

定制编辑器的工作涉及 4 个方面的问题，我们会针对每个问题进行详细的剖析，以便清楚地解释本文提供的方法是如何实施的。

第 1 个方面是开发内容编辑器所固有的复杂度问题。开发编辑器迟早都会变得异常复杂（有人称之为“复杂度膨胀（*bloated*）”）。如果开发工作非常简单，那么主流的编辑器产品就够用了，而且还可以规避开发的费用。

其他 3 个方面的问题分别与开发团队中的特定角色有关。例如，如果内容创作人员使用编辑器的方式不同，那么由此产生的问题只会惟一地对应于内容创作人员，对其他人则不存在类似的问题。

1. 编辑器与 GUI 复杂度

就拿 3D 建模程序来说，针对每一个正在编辑的对象，我们通常都会有 4 种不同的视角，每个视角都在自己专用的视区里显示。每个视区都必须是独立可控的。从最基本的摄像机视角控制开始，编辑器还要为每个视区分别提供下列控制功能。

PDG

显示	控制
显示观察点（look_at point）的位置	拉近或拉远，即改变视场（FOV，field of View）
在其他视区可见位置显示不同视区	二维平移摄像机
摄像机的位置	依 3 个坐标轴旋转
渲染“文档”（3D 模型）	三维变换
渲染游戏世界陆标（主要是笛卡儿坐标轴、正负坐标轴向等）	选择每一个可选项，对摄像机或观察点，或者对这两个项目同时进行操作
渲染每个主要道具的精确笛卡儿坐标，至少是摄像机和观察点	

这就是一个“编辑器”，一个除了移动摄像机（且只有最基本的操作，而没有其他有创造力的控制方式）、什么都干不了的编辑器！我们有 5 个独立的需要显示的东西，还需要提供 5 个独立的控制集。这 10 个项目都需要各自独特的算法，虽然这些算法本身是很容易实现的，但还是需要我们编写程序。我们已经有了足够多的独立算法（每个都需要单独维护、升级、替换以及/或者移植），但其维护起来也很不容易，而且还没有添加关键的特性，例如物体选择和基本的编辑功能。

一个典型的游戏内容编辑器也许要包含 50 个独特的显示算法、30 个（或者更多的）控制算法，以及不断延伸的拓展算法。这样的维护工作简直就像噩梦一般。而且，大部分的编辑器（无论是代码，还是结构）很快会变得非常笨拙而难以操控。这样，即使是添加非常简单的功能，也会带来可观的费用支出。如此一来，我们很难对这个工具做出合理的判断，特别是在终端用户（玩家）通常不会看到这个工具的时候。

2. 用户

内容编辑工具的最终用户基本上不会是程序人员。虽然，编程人员确实也开发并使用某些编辑器（当然包括 IDE 集成开发环境），但是，开发这类工具通常是为了让非技术人员可以更容易地去创作和定制游戏内容。对于大型游戏项目，主要的用户通常都是美工（创作二维或三维图形图像内容）、游戏作家（创作要玩家搜寻的物品、分剧情、游戏逻辑等），以及使用解释型/专用程序语言（例如开发 AI 脚本）的编程人员。

这些用户的主要需求包括：

- 强大的、功能丰富的图形用户界面（GUI）
- 可扩展，随时（从终端用户的反馈到工具开发人员决定添加新的特性这个时间段）可以添加新特性
- 可靠，大部分的编辑工作可以很快、很容易地让终端用户使用，而且工具必须是稳定、可靠的

第一个需求也许是非常明显的，但是常见的平台应用工具和应用程序接口 API 可以解决这个问题。但是，可扩展性这个需求对开发人员来说是最主要的挑战，因为工具本身就是一个定制的解决方案。而最具挑战性的需求，莫过于速度和易用性——不但要快，而且还要易于使用。开发商要罗列出一长串的功能特性（随着时间的推移，一定只增不减），同时还不能对每个命令的使用多点一下鼠标，这可如何是好呢？为了压平学习曲线，让最终用户更有效率，可以这样假设：只要给最终用户一些字元控制，让他们可以自己定制 GUI，这样就足够

了。也就是说，最终用户可以选择自己熟悉的编辑器主界面的布局方式。

不幸的是，编辑器的 bug 可以说是项目最致命的东西。例如，关卡编辑器的 bug 可能会无声无息地毁掉存档文件。这对游戏本身并没有什么长期的影响，因为这只会伤害某些正在开发的内容。因此，我们面临的诱惑是，在常规的项目里，给这些 bug 比较低的优先级（虽然，我们很明确地知道应该在内容编辑器的 bug 列表里给这些 bug 非常高的优先级）。但是，用户也会因此浪费很多精力，特别是因为很多对工作内容损毁无法进行回溯性的弥补（数据丢失了，就是丢失了，这是无法挽回的）。因此，内容创作人员必须假定这些 bug 肯定会在某个时间发作，并在每次编辑操作中事先预防，即使这个 bug 平均每 100 次编辑操作才会发生一次。不可靠的编辑器会很容易地毁掉大部分或所有努力的结果，而这些结果恰恰是它本应该提供给我们的（如果这个编辑器稳定可靠的话）。更糟糕的是，开发人员根本不重视这个问题，他们认为仅当 bug 出现时才会影响用户的效率。而事实上，这个“不常出现”的 bug 一直都在不停地影响着用户的效率。

3. 开发商

一般来说，游戏工具（例如编辑器）并不能直接提高游戏产品的评价分数，或者直接提高游戏的销量。

坦率地讲，一般情况下，编辑器本身是不能赚钱的。

很多游戏都是基于上述想法而开发的，并且为游戏产品相配套的内容创作工具，往往会在产品优先级列表上处于比较靠后的位置。另外，我们都知道，开发商很少为自己开发编辑器。之所以会有这些工具，都是为了让更多的人能够对游戏做点贡献（也就是说，所有非程序人员，或者只会用 AI 语言、但不懂 C++的人等）。这样的结果就是，开发商通常会开发出自己不想用的东西，还得费力去维护，这也无疑会占据他们本应花在核心任务上的时间。

新的特性会零零星星地添加进来，与此同时，开发人员可能正把大量的时间花在核心代码上。这就意味着，这个编辑器的源代码经常会变得多少有些“枯朽”（也就是说，每次要添加下一个新特性时，由于相隔的时间实在是太久了，连原创人员都忘记编辑器是如何工作的了）。

有些团队有专门的高水平人员负责工具的开发工作，这样会降低上述那些问题带来的不良影响。即使如此，这些人手头上也通常会有大量其他的工具，需要他们去支持、升级和维护，因此也多少会面临同样的问题。

所以，对于编辑器开发人员而言，一个工具软件（例如关卡编辑器）的主要需求是：

- 最小的时间投入，也就是用尽可能少的时间去开发编辑器。
- 可维护能力，程序代码必须非常易于理解，而不必花太长时间去领会。
- 可扩展性，必须可以快速地添加新特性，而不必重构原有代码。

4. 制作人和项目经理

对于一个游戏项目，制作人主要考虑关键路径、利益/支出比率和利益/风险比率等方面的问题。从利益/支出比率来看，定制编辑工具看上去确实不错，因为它们不但能够降低内容开发的费用，同时也有助于内容创作人员制作出更为复杂的内容，而这些复杂的内容是他们在拥有编辑工具之前无法管理。

PDG

但是，在全部的工具开发工作接近完成之前，这些定制的编辑工具通常没什么用处（还有比没有用处更糟糕的，比如，编辑器只是简单地写入某个文件格式，但是主要的游戏引擎却不支持这种文件格式了）。如果制作人知道，未来的某天，为了赶工，他们也许要取消或按比例缩减工具开发工作，那么，花时间在这样一个需要先期投入的工作上，会是多么巨大的风险。

更糟糕的是，从实际操作的层面上看，编辑器的开发工作通常是和主要团队启动某个游戏项目同步开始的。游戏制作人的项目总结报告通常会引用由内容创作人员造成的损失：数月地等待相关编辑器到位；或者是证实新版本的编辑器无法向后兼容之后，只好又使用早期的版本，结果只能删掉所有的东西，从头再来。通常的结论都是"下次要尽量多地使用中间件产品"。但现实情况是，如果只想出低价钱，那么买到的中间件产品只能提供最高层的通用功能。不管怎样，最大的价值就在于拥有一个为游戏项目全面定制的工具（地图格式、引擎架构和专用编码方案等），并由此达到生产效率最大化。

1.1.2 解决方案

在深入讨论了相关问题之后，该是拿出解决方案的时候了。我们会涉及相关的几个领域，第一个就是抬头显示系统（heads-up-display），或简称 HUD。

1. HUD

很多游戏开发商都很熟悉抬头显示系统（HUD）。人们最初发明 HUD，以及现在频繁地使用 HUD 都是为了实现快节奏的游戏。在这种游戏中，玩家不会冒险调出相关的菜单，将注意力脱离主要视区。但他们要求随时能够即时地获取某些关键的信息（例如：当前的健康值、剩余弹药等）。

对于游戏和编辑器，HUD 的主要好处是：

- 即时获取关键数据。从大型数据集中精选最关键的数据片段，并显示出来。
- 方便。将数据和主视区进行融合，从而让玩家不用脱离主视区就可以看到想看的数据。
- 清楚、可视。对屏幕区域进行合理布局，从而使被覆盖的区域显示的都是相对不太重要的信息（例如：标识目标物体的光标不再是覆盖于目标物体之上，而是显示成围绕在目标物体周围的圆弧或线框；在天空或背景的无关区域上设置盲区，而不是覆盖目标本身）。

这些好处对复杂的编辑器是很有帮助的。但是，这些好处并不能解决前面提到的那些问题。

2. 上下文相关的 HUD

我们扩展了单一整体式 HUD，将之变成了一个上下文相关的 HUD。由于有了大量内置的智能，一个上下文相关的 HUD 可以时实地对用户的行为作出反应。例如，一个第一人称射击（FPS）游戏的上下文相关 HUD 可以正常地显示出弹药、健康值和得分（有效射杀对手的次数）等信息。但是，如果玩家不小心中毒了，健康值就会不断地减少。这时候，系统就会放弃其他所有的信息，只是简单地用一个温度计来计数玩家的健康情况，因为这个时候其他信息都变得相对不太重要了。

在一个编辑器中，在某个给定的时间，鼠标光标的位置是玩家当前行为或思想的最好指示。因此，上下文相关的 HUD 通常会对鼠标光标的变化最为敏感。例如，标准的 Windows 应用程序中的工具提示弹出窗口虽然很简单，当光标离开某个按钮时，它就会马上消失，但这就是一个上下文相关的 HUD 例子。

Microsoft 是率先在应用程序中使用上下文相关 HUD 的公司。虽然他们也只是做到了一些皮毛而已，但在可用性方面却获得了很大的提高。例如，在 Word 中，如果鼠标光标移动到某个单元格的边缘线上，它就会变成调整大小的光标，而且单击鼠标键所对应的功能也会临时随之变化（只有光标的变化才是 HUD 的变化，后面有详细说明）。

3. 抬头编辑

在上面提到的 Word 程序的例子中，不仅光标发生了变化，鼠标的全部功能也随之发生了变化。上下文相关的 HUD 只会为显示的内容提供丰富的、智能的改变。抬头编辑能力结合上下文相关的 HUD 则可以自动地转换当前的工具或者编辑器的模式。

一个有抬头编辑能力的编辑器对用户来说是个不小的进步，原因如下：

- 自动化。消除了长期以来手工切换工具的要求（光标会根据上下文环境自动变化，同时切换到适当的工具上）。
- 简化用户界面。工具栏上的大部分小图标都可以去掉了，因为用户不再需要手动地选择它们。
- 学习曲线下降了，生产效率提高了。学习如何使用编辑器只需要很短的时间，因为系统会自动给用户呈现正确的工具，从而节省了在菜单里、工具栏上或其他地方寻找合适工具的时间。

但是我们可以做得更好：抬头编辑的实现只需要一个非常简单的架构，它可以让开发人员更容易地进行维护和扩展，从而基本上消除“代码枯朽”的问题。我们提出的架构要求所有要添加的新特性都作为一个完全封装的模块，目的是保护现有代码，以便将来可以轻松地更换某些部件。

1.1.3 实现

迄今为止，这个解决方案的实现可以分解成几个部分：投影胶片、渲染器和工具。我们还将讨论在几个组成部分之间共享代码的细节。

1. 投影胶片

对于用户，我们基于会议室或礼堂里常见的高射投影机（overhead projector，简称 OHP），提出了一个投影胶片的概念模型。每一个投影胶片只是一张透明的塑料纸。每个演讲人都可以独立地进行插入、移除或编辑等工作。每张投影胶片包含一个或几个图表或文本。分组工作是精心安排好的，目的是保持投影胶片的数量尽可能得小（以便于管理），同时还要允许有一个相当的间隔尺寸，以满足演讲者快速更换图表的需求。通常情况下，某个特定投影胶片的内容本身是没有什么意义的（例如一些无实质内容的箭头），但是与其他投影胶片放在一起，就产生了上下文关系，进而突然变得意味深长。

对于真实的投影胶片，我们很容易就能重新调整它们的顺序，旋转或调动某个单独的层，以及隐藏或显示某些部分等。这个独特的方法经证明是非常强大的，因为它给了用户（演讲人）很多方法去即时改变显示的内容：用户可以非常快速地添加或去掉某些内容，也可以很容易地简化或添加更多的细节。

我们的软件方法就是构建一个分层的系统。在这个系统中，每个层都类似于一张投影胶片。与投影胶片一样，每个层都是透明的，覆盖着主要的数据（视区）。也就是说，每个层都是一个 HUD。但是，与投影胶片不同的是，我们这个分层的 HUD 系统有着更为丰富的控制选项。这主要是因为我们通过计算机的 GUI 获得了更多的能力。另外，每个层都有一个或几个与之相关的控制模式（也称“工具”），例如“转换选区”和“旋转选区”。这样就把每一个层变成了一个带有相关编辑模式的、已经封装好了的上下文相关 HUD。

2. 渲染器

主编辑器视区看上去就像一个传统的编辑器。它显示的是当前正在编辑的文档，这个文档可能是文本文档、3D 模型，或者其他任意的渲染数据。文档是屏幕上显示的数据集，我们可以保存或打开文档。但文档并不是 GUI/编辑器的一部分。主视区不再显示其他内容，其他的所有事情都由 HUD 来处理。这也为用户提供了一个便捷的方式，让他们可以确切地看到保存的内容，而不会错把 HUD 提供的纹理着色当成模型和纹理的一部分，进行错误的保存。他们只须关闭所有的 HUD，然后就能得到一个“所见即所得”的结果。

如果文档本身太过复杂，你或许需要在核心渲染器中仅渲染部分文档，然后在 HUD 里渲染其他部分。例如，有些 3D 模型的文件格式会在一个单一文件里保存顶点数据、纹理数据、骨骼和动画等所有数据。你可以启动编辑器，让它仅仅在核心渲染器中读取顶点数据并进行渲染，从而让编辑器可以更快速地运行。然后，再为每个更先进的元素添加一个独立的 HUD/层，而且不需要改变现有的代码。如果纹理渲染器中存在某些 bug，那么用户总是可以简单地关掉纹理渲染层，并继续运行顶点渲染器，就好像使用的是原先版本的编辑器一样。

每个 HUD 都会在编辑器视区的顶部将自己透明地显示出来。和对待投影胶片一样，我们需要做一些仔细的安排，以便将想要的视觉特性划分到不同的 HUD 中（如对选中的部分进行高亮显示）。而且通常情况下，HUD 的数量越多越好。

用户有个简单的工具，可以选择他们编辑的层。动作向导程序会智能地选择一个层的队列，然后自动地按顺序一个层一个层地传递给用户，以便完成整个编辑任务。

听起来，这与现在流行的绘图程序中的图层很类似：每个图层都是一个独立的个体，用户的行为仅限于那些在某个时刻被选定的图层。这个工作界面对很多电脑画家都再熟悉不过了。其主要的区别在于，我们是使用投影胶片来完成 GUI，而绘图程序通常是使用图层来处理孤立的内容（仅在某一个图层上做修改）和内容的合成（例如，如果是在背景上进行混合操作，一旦完成，就很难还原。所以，通过图层进行临时的合成，事情就简单多了。这就像一个即时的预览，而不会永久性地将创作好的内容提交到当前的合成图中）。

3. 工具

每个工具会分配有一个自己专用的投影胶片，或者是享用一个属于某个显示元素（及其他工具）的投影胶片。如果不太确定，那么最安全的方法就是让每个投影胶片对应一个工具。

PDG

4. 共享代码和扩展 Context

如果把每个 HUD 的代码分解成一个或几个类，就会碰到代码复制的问题。这也是为了让 HUD 彼此尽可能地独立所产生的副作用。这是很自然的事情，但是代码复制会很轻易地破坏维护工作的简易性，所以，如果可能，我们肯定要尽量避免。

会被复制的代码，通常大部分都是常见的显示代码，而不是工具代码，例如某个点的 *x,y,z* 坐标。之所以要代码复制，是因为在特定的编辑器中，不同的工具往往要显示或者影响同样的信息，只是彼此的方式不同而已。这还算幸运，因为它意味着我们可以安全地把代码封装在一个单一的 HUD 渲染器中，从而让这个渲染器中的每一个工具对其只有非常有限的依赖关系（也就是说，这些工具之间不再有任何依赖关系）。

要想完全避免依赖关系，还有一个更好的、着眼长远的解决方法：把 *rendering X* 作为渲染器的主要特性（这里的“*X*”是一个变量。不同的工具，会以不同的方式在不同的时间来指定 *X* 的值）。为此，我们要对 *context*（上下文）的定义做些补充。context 原来只表示光标的位置，现在我们还要让它保存 *X* 的当前值。我们还需要提供至少一个 HUD 来查询 context 的内容，并进行渲染。这样做的一个好处是，可以有多个不同的 HUD 一次性地以不同的方式（不同的格式、不同的细节等）进行渲染，然后让用户来决定他们想看到的是什么（使用显示/隐藏 HUD 的控制功能）。

举个例子，如果有多个工具需要针对某个点或向量，弹出提示它们（*x,y,z*）坐标的信息，那么我们就可以给 context 补充一个字段 *currentpoint*。工具会在合适的时候填充这个字段，然后会有某个 HUD 把它渲染出来。

5. 源代码

不管使用的是何种视窗系统，源代码实现上都主要有两个元素：一个 HUD 管理器，以及每个单独的 HUD 个体（所有的 HUD 个体都是相同的类型）。

HUD 管理器是 MVC（Model、View、Controller）术语中的一个 Model 对象。它有一个数据结构，其中包含了指向所有当前 HUD 的引用，并且有由程序自动完成添加/移除 HUD 的方法。它还负责跟踪监视 GUI 的状态（例如，管理 HUD 堆栈，详见“用户控制”一节）。

所有的 HUD 都是一个模版/界面的实现，它会提供回调函数，用于触发系统去渲染配套的界面（例如，在后置缓冲中绘制 HUD），以及输入 GUI 事件（例如，单击鼠标）。

通过把某个特定 HUD 的所有逻辑都封装到一个类中，开发人员就可以很容易地在编辑器中添加/移除 HUD，或者改变现有的 HUD，而不必同时去编辑多个源文件。显然，对于一个特别复杂的工具，其自身就包含很多的类，但是激活该工具以及渲染 HUD 交互部分的逻辑，都保存在一个地方。

ON THE CD

本书配套光盘中的程序实现有些太过简单，但是大家应该可以很容易地按照自己的需要去扩展。HUD 管理器对应的文件是 net.tmachine.gpg.hud.HUDManager。HUD 的界面/ADT 对应的文件是 net.tmachine.pgp.hud.HUD。基础抽象类包含了一些通用方法，其对应的文件是 net.tmachine.gpg.hud.BaseHUD。net.tmachine.gpg.hud.HUDDemo 是一个演示程序（如果尝试运行 heads-up-editor.jar 文件，该演示程序就会自动运行）。

1.1.4 用户控制

用户对 HUD 编辑系统的交互作用包括下面 3 个方面：

- 交互性，抬头选择要使用的工具。
- 激活和关停某个 HUD（以及所带的工具）。
- 查看抬头编辑系统的内部状态。

1. 工具的选择

一个 HUD 就是一系列其他 HUD 和工具的代理。这个 HUD 有大量上下文相关的渲染功能，包括用于每个所代理 HUD 的激活符号。例如，如果它代理的是平移物体、旋转物体和伸展物体，它就会在应该激活的区域为每个物体显示相应的图标（例如，当光标位于物体的边界之内，平移图标就会被激活；当外围物体距离某个窗口角落在 10 个像素以内时，旋转图标就会被激活；而当外围物体距离某个角落超出了 10 个像素，但距离任意一个边缘又在 10 个像素以内，拉伸图标就会被激活）。

在激活区域（代表映射到该区域的其他 HUD 和工具）单击鼠标左键，就可以把当前的 HUD 和工具放入堆栈中。这样我们就可以随意地嵌套代理 HUD。这对非常复杂的编辑环境特别有好处，而且还可以允许我们从几个不同的 context 中申请同一个 HUD/工具，而不需要什么上下文感知或是使用几乎相同的代码重新实现每一个不同的上下文。

2. 激活 HUD

为了管理不同的 HUD，我们简单地从高端的绘图程序中拷贝了标准的图层调色板。这是一个非常成熟的 GUI 形式，很多用户对此都已经耳熟能详。这样的 GUI 形式将所有的图层按顺序列表，并且提供了相应的控制功能，可以显示/隐藏特定的图层，还可以对图层进行拖放操作以重新调整它们在列表中的顺序。

在 HUD 系统中，重新排序的工作是非常有用的，这样就可以控制哪些 HUD 可以在其他哪些 HUD 之上进行绘图操作。有了这个功能，用户通常就可以解决屏幕上的任何绘图冲突，保持底层代码的简单有效。否则，每个 HUD 之间就需要互操作，以避免由于重叠交错而造成的对屏幕空间使用的协调，以及对特定屏幕区域的绘图保留。

显示/隐藏功能的使用和其他绘图软件一样：隐藏一个 HUD 之后就会关闭其绘图代码。这样做或许是为了减少屏幕上的混乱情况，提高渲染性能，或许仅仅因为用户在当前的会话中不需要某个特定的 HUD。由于在整个编辑会话过程中，HUD 一直是永存不变的（这和绘图软件中的图层不一样，图层的生命周期短暂，且只和当前的文档关联），所以在一个相对复杂的抬头编辑器中，有这样一个非常有用的特性：用户可以快速保存/重载一个隐藏 HUD 的调色板。一个相对简洁的实现方法是，在隐藏 HUD 的预置参数对话框上创建一个小标签，当用户创建预置参数时为其提供一个文件名。这样，用户就可以在不同的编辑模式下快速地进行切换，高效率地完成目标操作，减少在对话框、菜单和工具栏之间的穿行。

3. 内部状态

这可以达到两个不太明显的目的。首先，由于系统的内部状态可能会随意地屏蔽原本可

用的工具，所以，让用户理解什么是中间选择状态是很重要的（举例来说，如果用户搞错了，选择了错误的工具，它们通常不会知道自己点选了哪个工具，而只是知道自己想选择什么工具），该功能类似于一个导航图。其次，这样做会让调试工作变得异常容易，因为用户可以很容易地“回复”到原先出问题的地方，而不需要开发人员去猜测或者查看日志文件。这是非常有帮助的，因为它可以让用户避开那些会发生致命 bug 的地方。如果用户知道“导航图”的某个地方会出事，他们就可以很容易地避开它。

1.1.5 总结

本文给出了一个简单的编辑器设计方法，它可以给用户提供更为强大的功能，同时也让开发人员变得更轻松。这是一个有效的闭环，可以为周遭人等减轻工作压力：开发人员可以不用再花时间和那些不能维护的代码纠缠不清；用户也可以快速地添加那些看似简单（而实际上颇为复杂）的功能特性。

从功能特性的角度来看，用 HUD 模式组织编辑器可以使之保持很好的灵活性：由于单个 HUD 的独立性，添加新特性不再困难。同时，不用太费力气，编辑器就能同时运行同一个控制的备用实现：即同时运行某个实现及其备用实现，且用户可以根据自己的需要实时地打开/关闭它们。有了这个特性，只需要很低的开发费用，我们就可以很容易地去实验开发很多非凡的功能，或尝试很多新的点子。

但是，对于抬头编辑（HUE），我们只是涉及其皮毛而已，并没有深入探讨。有很多方法可以将本文的思路进行拓展，从而获得 HUE 更多、更强大的支持。当然，这也需要相当多的编程工作，不可能轻易获得。本文涉及的核心支持，其开发成本和维护成本都比较低廉，但却提供了大多数好处。

ON THE CD

本书配套光盘中源代码的勘误表和升级信息会在[Martin04]中提供。现在能提供的源代码是比较少的，笔者以后会在网站上提供一个基于该代码的库（对商业应用和个人都免费），以及最新的版本和改进细节。

1.1.6 参考文献

[Eclipse04] Eclipse. Available online at *http://eclipse.org*. 2004.

[Martin04] Martin, Adam, et al. Java Games Factory, “Libraries and Code Snippets.” *http://grexengine.com/people/adam/gpg5/* and *http://grexengine.com/sections/externalgames/*.

[Popa99] Popa, Adrian. “Re: How Does a Heads-Up Display (HUD) Work on the Aircraft?” *http://www.madsci.org/posts/archives/jul99/931328936.Eg.r.html*.

1.2 在游戏中解析文本数据

Aurelio Reis
AurelioReis@gmail.com

很多游戏都需要读取并解释大量的文本数据。从脚本语言到着色器（shader），必须经常从一个标准格式中读取文本数据，并将之转换为程序可以直接使用的二进制数据结构。在本文中，我们会考查词法分析器（tokenizer）的创建和使用：该模块负责将文本数据转换为分立的单位，以便于解释并集成到正确的游戏信息中。

1.2.1 开始之前

明确说来，*token* 有时是指词位/词法（*lexeme*），而 *tokenizer* 指的是词法分析器（*lexical analyzer*，或称单词分析器）。本文将使用 *token* 和 *tokenizer* 这两个术语。因特网上有很多免费的词法分析器，读者应该好好看一下，特别是其中比较流行的 Lex（with Yacc）[Lex & Yacc]词法分析器。这是一个开放源代码的词法分析器，是一个非常有价值的学习工具。词法分析和解析理论是一个非常庞杂的领域，本文不会覆盖其方方面面。相反，本文的目的是向大家介绍并详细解释词法分析器在游戏制作过程中的用途，以及如何去实现一个词法分析器并在实际中加以应用。关于词法分析器和解析器的历史和理论，可以参见本文结尾部分的“参考文献”。

1.2.2 token 到底是什么

一个 token（记号）就是一组不可分割的字符，用来表示独立的基本符号。基本上，它就是一组构成特殊“单词”的字符。例如，程序清单 1.2.1 中的一小段 C++程序代码：

程序清单 1.2.1 一段简单的 C++代码

```
if ( i == 0 )
{
   return false;
}
```

如果把程序清单 1.2.1 分解成若干个 token，就能得到程序清单 1.2.2 中的列表。

程序清单 1.2.2 从程序清单 1.2.1 的代码中生成的已发现记号列表

```
if          <string>
(           <left parenthesis>
i           <string>
==          <string>
0           <number>
)           <right parenthesis>
{           <left brace>
return           <string>
false            <string>
;           <semi-colon>
}           <right brace>
```

正如读者所看到的，token 被分成几个特殊的类型。词法分析器只是盲目地将那个片段作为 ASCII 代码数据读取进来，然后再执行分类过程。后面会详细讲述如何处理这些 token，但是现在只要清楚是我们来定义这些基本的类型，然后形成一个基于数据序列（文本）的被发现记号列表就可以了。

1.2.3 编写词法分析器

词法分析器的工作原理实质就是一个有限状态机（*finite state machine*，简称 FSM）。它通常的功能就是，根据当前的全局状态执行某个特定的动作或事件。为了与游戏世界并行，在虚拟的游戏世界中，当前会使用有限状态机来指引 AI 代理程序的工作。例如，它当前的状态可能是“漫游”，在这种情况下，它可能会随机地遍历整个节点网格（或相似的导航方案）。

一个输入事件，例如看到一个敌人，会触发状态的改变，即从当前状态变为“搜索并摧毁”的状态。在这个状态下，AI 代理程序会不断地搜索敌人并发起攻击，直到干掉对方或对方自己逃脱。此时，系统的状态会再次发生变化。在这个例子中，我们把虚拟的视觉和听觉输入作为输入事件。但是，在词法分析器中，输入事件采用的是特殊字符的形式。让我们看一个最简单的情况：程序注释。在 C++语言中，程序注释有两种形式：单行注释，在两个左斜线（`//`）后的文字被程序编译器视为注释，直到该行的结束；多行注释，在`/*`和`*/`之间的文字被编译器视为注释。词法分析器会遍历数据集中的每一个字符，直到它找到了一个左斜线（`/`）。然后，词法分析器会继续分析下一个字符，以判断其是否确实是一个程序注释。如果确实是程序注释，就会把状态切换到“在注释语句中”。之后，词法分析器会继续前行，跳过所有新字符，直到本行结束或者多行注释语句结束。

如果愿意，可以把词法分析器做得非常复杂。如果需要处理的只是简单的文本数据，则可以完全放弃记号（token）类型，需要定义的只是一些分隔符。分隔符会确定如何将数据集切分成 token，其作用与标点符号非常相似。当然，在书面英语中，最常使用的分隔符是空格。如果打算使用空格作为分析本句词法的分隔符，那么每个单词都会变成一个新的 token。但是，要注意还会使用逗号和句号。如果要跳过这些字符，同时还要使用它们来定义 token 的边界，就需要把它们也指定成分隔符。最常用的分隔符应该是空格（）、逗号（`,`）、回车符（`\n`）、换行符（`\r`）和制表符（`\t`）。也许还会用到分号（`;`）。但是，在某些情况下，最好还是把

它保留为一个特殊的 token 类型。

如果需要更为可靠的 token 类型检查，就需要用更为具体的东西来表示它们，而不是使用简单的字符串 token。这也许是更为可取的方法，因为这样做可以进行更直接的 token 解析工作。使用 C++方法，可以从一个基础 token 那里继承派生出一个新的 token 类型。一旦创建了一个新的 token，它就和那个与其特性最为接近的 token 类型相匹配。还可以定一些规则以允许使用字符串组。例如，一个字符串 token 可能会以引号开头，并包含多个字符，其中一些字符可能会成为分隔符，然后 token 以另一个引号结束（例如，"This is a single token"）。表达式(1+2*3)也可以用这个方式来定义，但是很多人会选择在解析阶段创建 token 之后才来定义。

还是来举个例子，回过头再看看程序清单 1.2.1 和程序清单 1.2.2。只要看看 token 的类型，就可以了解 token 位置的基本结构。正如下面要看到的，在解析这样一段代码时，判断数据类型要比检查每个 token 的对应文本简单得多。例如，可以定义这样一条基本规则：当碰到一个 if 语句字串 token 时，要求有一个左括号、一个由一些数值表示的等式和一个右括号，后面跟着其他的 token。不必检查 token 的显式名字，只需要根据类型就可以寻找相应的 token。虽然只有一点点方便之处，但这对于代码的可读性和使用确实颇有帮助。另外，这样也可以保证在解析数据时使用的是正确的语法。如果有适当的校错功能，词法分析和解析数据的工作对终端用户来说会变得容易得多。这也是为什么校错功能会成为一个优秀解析器的最重要部分的原因。

1.2.4 工作原理

本书的配套光盘中有一个简单的词法分析器，及相应的测试样本。对字符串进行词法分析时，词法分析器要遵循几个简单的步骤。首先，系统会将包含文本数据的文件读取到缓冲区中，或者映像到一个视区中（如果使用的是文件映像技术），然后针对这个缓冲区，调用词法分析函数。词法分析函数会遍历每个字符，并根据当前的状态（初始状态是 TKS_INWHITE，即读取空格）将字符进行分类整理。在整个过程中，会有很多次的状态变化。当碰到一个有效的字符时，状态就会变为 TKS_INTEXT，意思是正在读取一个真正意义上的字符。当碰到一个分隔符时，就会创建一个 token，赋予相应的类型，并将其添加到 token 列表中。另一方面，如果碰到引号，状态就会马上变为 TKS_INQUOTES，表示当前正在读取一个文本组（text group）。一旦碰到另一个引号，文本组就会结束，并最终将之确定为一个字符串 token。同样的过程也用来处理单行程序注释和多行程序注释。

这里要研究的主要函数是 TokenizeString()，它将所有字符顺序迭代，并根据字符的状况来改变状态。token 的分类工作是由函数 ClassifyToken()（见程序清单 1.2.3）来完成的。该函数会初步判断出某个字符是否与浮点变量、特殊字符（来自特殊字符表）或字符串变量的特性相匹配。该函数首推采用递归算法，由于这里只是起一个演示说明的作用，所以跳过了算法。正如读者看到的，该函数可以判断某个字符串是否可以转换成数字；如果可以，又会判断这个数是否为负数 isdigit()函数用来判断某个字符串是否是数字。但是比较起来，扫描字符串，看其中是否有小数点（小数点意味着这个字符串是个小数）的做法似乎更简单。函数 IsSpecialCharacter()只检查字符串的第一个字符，以判断它是否是

一个单字符的特殊 token，如左花括号、左方括号或引号。例如，要是把分号也作为一个 token 类型，该函数就会对其进行检查。如果做好了确立 token 的准备，函数 AllocToken()就会创建一个已分类的 token。如果词法分析过早地结束了，就会调用函数 FinalizeToken()，来确保后面没有剩下未处理的 token。在得到一个 token 列表后，就可以开始进行解析工作了。但是，需要解析的究竟是什么内容呢？首先，必须创建几种文本格式。让我们快速浏览一下程序清单 1.2.3 中所谓的 BNF 范式（*BN Form*）。

程序清单 1.2.3　函数 ClassifyToken()的实现（简化起见，去掉了程序注释）

```
TOKENTYPE CTokenizer::ClassifyToken( const char *strText, unsigned long ulFlags )
{
TOKENTYPE TokenType;
if ( !( ulFlags & TKFLAG_STRINGSONLY ) && ( isdigit( strText[ 0 ] ) ||
( strlen( strText ) >= 2 && ( strText[ 0 ] == '-' && isdigit( strText[ 1 ] ) ) ) ) )
    {
return TKT_FLOAT;
    }
else if ( TokenType = IsSpecialCharacter( strText[ 0 ] ) )
    {
return TokenType;
    }
else
    {
return TKT_STRING;
    }
}
```

1.2.5　制定自己的格式

走到这步，就需要找个例子来进行词法分析了。在附书光盘中所带的测试样本和源代码中，可以找到一个非常简单的例子。它基本上已经搭建好了一个格式的基础框架，我们称之为角色文件。不过，我们将使用.txt 作为文件后缀，这样，不用注册新的文件类型，就可以打开它。角色文件用来解释游戏角色的个性特征，包括玩家角色和非玩家角色。例如，在一个角色扮演类游戏中，必须首先创建自己的角色，或者从事先创建好的角色模版中选择自己的角色。角色生成之后，需要进行保存。而角色文件就是我们用来保存角色的文件（对于有角色模版的情况，角色文件就是用来加载角色的文件）。文件的格式其实很简单（也许是太简单了），但是为了便于解释，我们首先要理解巴科斯-诺尔范式（*Backus-Naur Form*，简称 BNF 范式，也有人称之为 Backus-Naur Notation）的概念。简单地讲，BNF 范式就是如何应用语言语法的解释。一个与日常书写直接相关的非常好的例子，可以在[CS310]以及对[Estier]中使用的符号的解释中找到。BNF 范式有固定的文法规则，它定义了字符串可以如何一起使用。笔者最喜欢的一个例子是 BNF 范式的日期，如程序清单 1.2.4 所示。

程序清单 1.2.4　用于日历日期的 BNF 范式

```
<date>        ::=  <month> '/' <day> '/' <year>
<month>       ::=      0 .. 12
```

```
<day>        ::=    1 .. 31
<year>       ::= 1900 .. 3000
```

读者也许凭直觉就可以知道，BNF 范式定义了一些符号，并告诉我们某个符号可以赋予什么类型的值。通过使用事先定义好的语法，并忽略某些月份不够 31 天的事实（以及闰年），就可以很容易地构造出一个日期，例如 12/25/04，然后使用这些定义好的规则就可以使之生效。把这些规则翻译成日常语言，BNF 范式表达的意义就是："一个日期就是月份后面跟一个斜线，然后跟日子，再一个斜线，最后是年份。月份是 0～12 中的某个值；日子是 1～31 中的某个值；有效的年份是 1900～3000"。

由此而举一反三通常是非常容易的。描述自己的"语言"，然后把它翻译成 BNF 范式。如果想获得更详细的解释，参见[Garshol03]。最后要注意的一件事情是，很多人并不遵循严格的 BNF 解释。例如，假设你要用"+"分隔那些符号来表示"外加"的意思，BNF 范式可以是有效的（虽然在接连使用多个字符串符号时，如稍后的例子所示，类似这样的一些用法也许是有效的）。找一个最适合自己的系统，但是也不要太偏门，否则别的系统就无法解释你的指令了。还要避免歧义（同一个规则如果在语法中有多种可能的解释，系统将无法判断哪个才是正确之选）。BNF 范式的作用更多地是作为一个指南，来帮助人们思考得出合理化的关键设计决策。作为一个有趣的练习，不妨看看是否可以为程序清单 1.2.1 的那段 C++程序写出相应的 BNF 范式。

了解了什么是 BNF 范式之后，就该看一下角色文件实例——Character.txt 了。对这个角色文件格式的语法规则的定义，如程序清单 1.2.5 所示。

程序清单 1.2.5　角色文件格式的语法规则

```
<CharacterFile>   ::= 'Character' + '{' <TextBody> '}'
<TextBody>        ::= <CharacterStat> <Value>
<CharacterStat>   ::= 'Name' | 'Strength' | 'Dexterity | Constitution | Intelligence |
Wisdom | Charisma | HitPoints
<Value>           ::= <Float> | <Bool> | <String>
<Float>           ::= 0 .. 9+ + . + 0 .. 9+
<Bool>            ::= TRUE | FALSE
<String>          ::= a .. z+ | A .. Z+
```

读者也许已经猜到了，这里有意地加入了一些多义性的规则。对初学者来说，CharacterStat 现在可以是 3 种类型：float（浮点型）、bool（布尔型）或 string（字符串型）。利用这个既定的语法，从技术上讲，角色的名字可以被赋予 TRUE 或 3.14 这样的值。这很明显是错误的。补救的方法见程序清单 1.2.6。

程序清单 1.2.6　消除多义性之后的角色文件格式的语法规则

```
<CharacterFile>   ::= Character + { <TextBody> }
<TextBody>        ::= <CharacterStat>
<CharacterStat>   ::= (<FloatStat><Float>)  | (<BooltStat><Bool>) | (<StringStat>
<String>)
<FloatStat>       ::= Strength | Dexterity | Constitution | Intelligence | Wisdom |
"Charisma | HitPoints
<BooltStat>       ::= IsPlayer
<StringStat>      ::= Name
```

```
<Float>         ::= 0 .. 9+ + . + 0 .. 9+
<Bool>          ::= TRUE | FALSE
<String>        ::= a .. z+ | A .. Z+
```

正像读者所看到的，通过明确规定 FloatStat 只能是浮点型，BoolStat 只能是布尔值，且 StringStat 只能是字符串，多义性问题就完全消除了。通过在数字和字母的定义后面加上“+”，BNF 范式就有了一些自由度。这是因为，如果每次编写新的 BNF 范式都要重新精确定义一个数字，那就太笨了。放置“+”主要就是为了说明，在这个地方指定的数值，其位数是没有限制的。这样，如果角色文件交叉引用这个 BNF 范式，我们就会发现它可以满足所有需求。事实上，它可以帮助你在创建 BNF 范式之前，就把所有想要的格式书写出来。正如前面提到的，有些格式初看上去没什么问题，使用起来才会发现其中有灾难性的漏洞（大家可别笑，这种事情确实发生过），而 BNF 范式就是一个非常优秀的堵住漏洞的方法。

1.2.6 解析 token 列表

定义好格式，编写了一个带合法输入的文件，并把文本转换成分立的 token 之后，就该解释这些 token，把它们转换为有效的二进制数据结构了。这个处理过程通常被称为“解析（*parsing*）”，因为它会对数据集进行独立的分析和处理。在这个例子中，就是把数据处理成计算机可以理解的形式。解析数据是一个非常棘手、非常深入的课题，弄不好就会出错。有些人喜欢采用递归解析法（也就是递归下降解析法，Recursive Descent Parsing[RecursiveDescent]），在不同的层次上对数据进行解析，然后返回一个总体的结果。在这个例子中，我们会根据语法规则，简单地线性处理每一个 token。

解析数据最重要的一点，就是要记住校错工作是很关键的。其重要程度怎么强调都不过分。如果不能确保数据解析的准确性，灾难就不可避免。可以选择使用异常处理程序，来应对格式不符的情况。如果不符规则，可以简单地抛出异常。如果引用这里的代码，就会看到首先检测的是 Character 和相应的声明部分，以确保正在一个新角色的入口处。在此之后，就进入一个循环。如果为 CharacterStats 赋予了一个值，这个循环就终止。如果处理完所有的 token，或者出错了（例如，给 FloatStat 指定了 BoolStat 的值），这个循环就会终止，并结束整个处理过程，同时退出程序并报错。

虽然配套光盘上的范例程序并不怎么太令人兴奋，但是它却可以很容易地扩展。对初学者而言，可以想象利用完全表达式检测功能和角色的移动/姿态控制命令为角色增加脚本行为。除了基本的角色加载功能之外，我们还对这个代码进行了扩展，使之具有了可以读入武器的功能。其他的一些道具也是很容易就可以添加进来的，例如盔甲和盾牌。随之附带的还有一个简洁的小技巧，即将 CharacterStat 直接与其类对象的数据成员进行匹配，其方法是使用 Character 类对象（CCharacter）的字节地址偏移量。使用这个技巧，解析器可以很容易地通过 CCustomField 数组来检测新的类成员（详细内容，参见代码）。对于网络变量同步或者保存/加载功能，这个技巧用起来也很方便。还有很多方法可以扩展这个范例程序，希望读者在不断的尝试中找到快乐。

1.2.7 总结

本文深入研究了实现通用词法分析器的基本方法。用这个词法分析器可以将文本数据分解成有效的 token，进而再将其解析成为有效的游戏数据。利用这些知识，我们创建了一个非常简单的文本格式，用来保存游戏中虚拟角色的相关信息。光盘中的代码有非常完整的程序注释，这样，读者就可以在调试程序中一步一步地研究，从而更深入地了解这段代码的运作流程。接下来，大家就可以在自己的游戏里应用这些知识了。

1.2.8 参考文献

[Aho, Sethi, Ullman85] *Compilers: Principles, Techniques and Tools.*

[CS310] *http://marvin.ibest.uidaho.edu/~heckendo/CS310/grammar.html.*

[Eli] *http://eli-project.sourceforge.net/elionline4.4/lex_toc.html.*

[Estier] *http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html.*

[Garshol03] *http://www.garshol.priv.no/download/text/bnf.html.*

[Lex&Yacc] *http://dinosaur.compilertools.net/.*

[Punctuation] *http://www.wordiq.com/definition/Space_(punctuation).*

[RecursiveDescent] *http://encyclopedia.thefreedictionary.com/Recursive%20descent%20parser.*

1.3　基于组件的对象管理

Circle Studio 公司，Bjarne Rene
bjarne.rene@circle-studio.com

传统的对象管理模型通常是依靠继承层次，在不同类型的对象之间共享功能。随着游戏产品复杂度的不断增加，这种方法会导致游戏策划的调整工作难以进行。而且，如果有几个分支都需要这个对象功能，就必须提高该功能在继承层次中的层级。

解决该问题的一个好办法是通过组件创建对象，每个组件只负责一个特定任务的数据和行为。基于这个原理的对象管理模型，在创建新对象和修改现有行为等方面，都提供了更大的灵活性。

本文的第一部分要首先介绍传统的对象管理系统与基于组件的管理系统之间的区别，以及使用后者所带来的好处。本文的第二部分会专门讲解如何从头开始创建一个基于组件的系统。最后，我们会对可用于实际项目的程序实现做一个基本的总结。

1.3.1　除旧迎新

在传统的对象管理系统中，所有的对象都是从同一个抽象基类中派生而来的。为了叙述的方便，我们只考虑这样的一个系统：所有的子类都是从 Cobject 这个类派生出来的。接下来就要决定从 Cobject 这个类中要派生出哪些子类。大多数情况下，从 Cobject 直接派生出来的子类也是抽象类。其中的一些子类带有某些功能，而另一些子类则不带什么功能。在继承树中，这些直接派生出来的抽象子类代表了这两种子类的切分。这种切分最典型的例子是可渲染/不可渲染，以及可动画/不可动画。图 1.3.1 展示的是一个简单的继承树。

乍一看，这个树图挺不错。这样的一个系统，其诱人之处就在于它在纸面上看上去很美，至少短时间看是这样的。对于游戏产品的制作，众人皆知，随着开发进程的推进，设计需求也在变化。这个模型在需求变化的应对方面无法令人满意。让我们举几个实际的例子来说明这个问题。

如果设计需求变化了，要求武器也要有动画效果，那该怎么办呢？如图 1.3.1 所示，Cweapon 和 CanimatingObject 分别处于同一棵树的两个不同分支。解决该问题的一个方法，是让 CWeapon 直接由 CAnimatingObject 继承而来。但是，且慢！武器是一个可以搜集的对象，所以，我们还必须让 CcollectableObject 也从 CanimatingObject 继承而来。结果如图 1.3.2 所示。

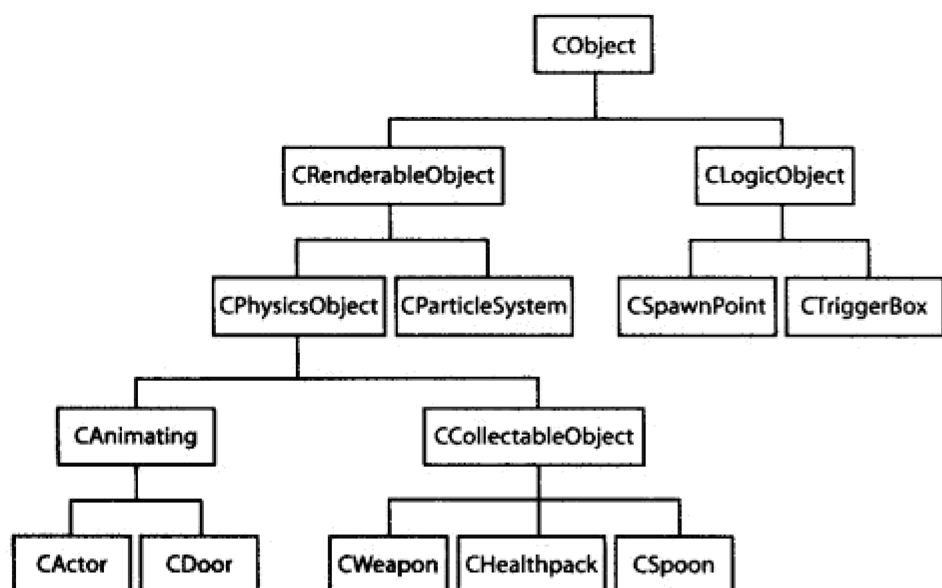


图 1.3.1 传统的继承树

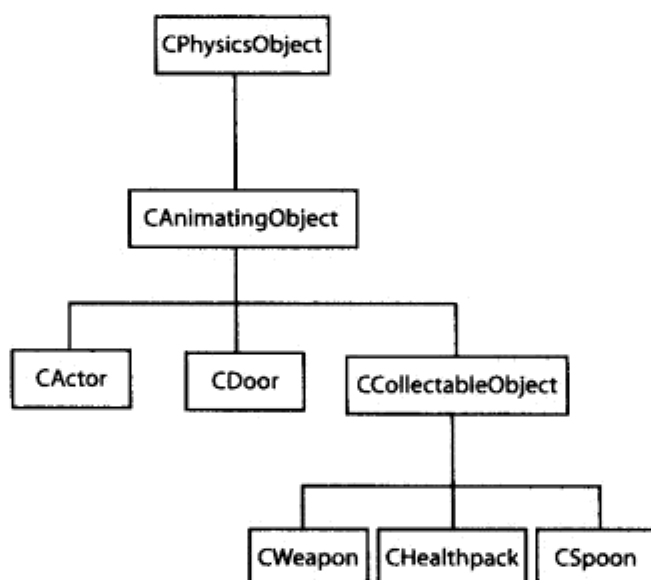


图 1.3.2 去掉一些子类，以便让 Cweapon 对象可以获得动画功能

这样虽然解决了问题（至少目前是这样），但同时又引发了新的问题。现在，由于从 CcollectableObject 派生出来的一个子类需要有动画功能，所以所有的可搜集对象都变成了有动画功能的对象。这样，好多对象都得背着一堆自己用不着的行李（功能）。这也意味着，为了成为合法的对象，其他所有可搜集的对象不得不去实现 CanimatingObject 中的纯虚函数（pure virtual method），即使这些函数对它们根本没什么实际的意义。

现在，考虑一下如果想让对象具备攻击破坏力，又会出现什么情况。应该只需要几个方法就可以实现这一点。于是，我们决定将这些方法添加到现有的一个类中。这种设计会调用从 CActor 类派生出来的对象，来实施攻击任务（这里并没有把这些内容表示出来，但通常至少要包括 CPlayer 和 CEnemy/CAICharacter 对象）。于是，我们继续走下去，把这些方法添加到 CActor 中。看上去，这是个明智的选择。

随着时间的推移，我们决定要让“门”也具有攻击破坏力，同时还可以被破坏掉。为了能够实现这个需求，我们要把实施攻击任务的方法上移到 CAnimatingObject 中，因为它才是 CActor 和 CDoor 的第一个通用祖先类。听上去，这确实是个比较合理的决策，也让我们得到了想要的结果。如果一个没有动画的对象也需要攻击力，那么另外一个合理的决策也许是将实施攻击任务的方法上移到 CRenderableObject 中。把很多这样“合理”的决策放在一起，其结果就是继承树变得越来越头重脚轻。某个类的向上移动也会带动很多方法的上移。这不是由类的功能来决定的，而是由类的位置来决定的。这样的类会丧失它们的内聚性，因为它们试图为所有的对象完成所有的事情。

1.3.2 组件

我们已经看到，在一个大型的继承树中，如果过分依赖其所有的对象，就会碰到很多问题。我们真正希望得到的是这样一个系统：即可以将现有的功能组合到新的对象中，并将新的功能添加到现有的对象中，而不用每次去重构大量的代码、调整继承树的结构。

1. 一个简单的对象

一个不错的解决办法是从组件创建一个对象。组件是一个包含所有数据成员和方法的类。这些数据成员和方法用来完成某个特定的任务。通过把几个组件合成起来，就可以创建一个新的对象。图 1.3.3 显示的是一个 Spoon 对象以及组成该对象的几个组件。

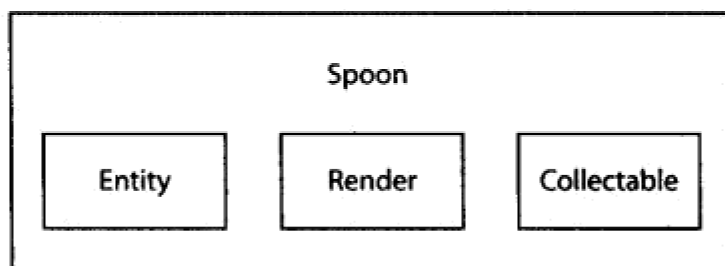


图 1.3.3　由组件创建的一个简单的对象

Entity 组件让我们可以把对象放置到游戏世界中。Render 组件让我们可以为对象指定一个模型，并根据组件的设置对其进行渲染。Collectable 组件让我们可以拾取这个对象，并将其保存在物品库中。

2. 接口

所有的组件都是从一个基础组件接口派生而来的。我们可以假想称之为 IComponent。该接口包含若干个所有组件都必须实现的方法。如果能够通过指向该基础类的指针来控制访问所有的组件，对象管理器（将在下一节中讨论）会非常受益。

只需要关心每个组件所显示的公共接口，这才是我们愿意做的。为了能够行得通，每个组件都需要从一个有功能约定的接口派生而来。用 Render 这个组件举例说明如下。

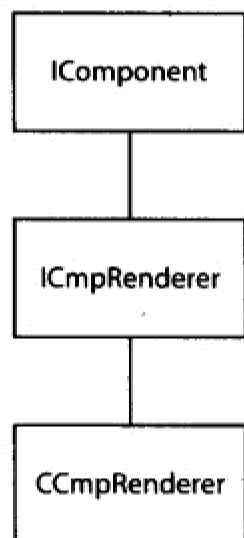


图 1.3.4　CCmpRender 的继承树

图 1.3.4 显示的是一棵继承树，它表现了 CCmpRenderer 的继承关系。位于顶层的是 IComponent；然后得到的是 ICmpRenderer；最后得到的是 CCmpRenderer，它负责实现其上层接口约定的功能。

这看上去和之前在传统对象管理系统中所做的工作惊人得类似。所有的子类都是从一个基础类派生而来，所有的地方都使用了继承关系，那么其中的区别是什么呢？其不同之处就在于，这个继承链上的类都执行着紧密相关的任务集。换句话说，这些类有着非常高的内聚性。

3. 跟踪管理

我们需要以某种方式合理地组织这些组件。为此，我们想到了对象管理器。对象管理器实际上是数据库的一个接口，其目的是防止暴露太多其余的代码。数据库不一定要是（也不太可能是）Oracle 或 SQL 类型的。它只需要跟踪所有的组件，并尽可能高效地在对象管理器中显示访问方法即可。对象管理器让我们可以创建、查询和销毁某个组件。我们将仔细地研究对象管理器，并在“实现”一节中提供一个可行的数据库实现。

4. There is no spoon[1]

回头看一下 spoon 对象的例子（参见图 1.3.4）。我们已经有了所有的组件，但对对象而

[1] 译者注：直译为没有勺子，电影《骇客帝国》中 Neo 的经典台词。

言，到底有什么变化呢？没有，什么都没有！为了舒适起见，也许可以保留 CObject 类，但我们不打算这么做。没有什么东西要求我们保留 CObject 类。而且，如果把它保留下来，那我们该如何决策哪些内容可以放在 CObject 类中，哪些内容必须放在组件里呢？把所有内容都放到组件中，也是设计系统时的一种选择。那么对于对象而言，剩下的就只有一个对象 ID 了。这个 ID 用什么数据类型都可以，只要它惟一地表示一个对象，而且便于传递。某种基于 int 类型的指针是可以用作对象 ID。当然了，谈论“对象”仍然是方便的，因为这是它们对外的形象。从这里开始，我们谈论的“对象”实际上指的是组合形成一个对象的组件集合。

5. 沟通

组成 spoon 对象的所有组件都会执行它们各自定义好的任务。但是，如果由此认为它们彼此之间不需要通信就可以完成这些任务，那就有点儿太幼稚了。Render 组件需要向 Entity 组件询问被渲染对象在游戏世界中的位置。当物体被拾取以后，Collectable 组件则必须告诉 Render 组件关闭渲染功能。

组件之间的通信有两种方式。如果知道想到询问或告之的特定接口，可以从对象管理器那里得到一个指向该接口的指针，然后通过这个指针来调用该组件中的方法。对于上面提到的 Render 组件向 Entity 组件询问被渲染对象在游戏世界中的位置的情况，这个方法非常好用。游戏中的任何可以访问对象管理器的代码都可以查询属于某个特定对象的接口。所需要的只是对象的 ID。

组件之间的第 2 种通信方式是以消息的形式进行的。在有些情况下，某个组件可能想说些什么，但又不是很清楚该说给谁听，这时第 2 种通信方式就派上用场了。我们可以把消息发送给一个或所有对象。比如，Collectable 组件需要通知某个对象：你已经被捡起来了，不用再显示了。Collectable 组件可以向对象中的所有组件发送这个消息。在初始化的时候，每一个组件类型都会告诉对象管理器应该接收什么样的消息。

6. 扩展对象

如果想要把勺子弄弯[2]，该怎么做呢？在当前的状况下，spoon 对象还是非常刚硬的，无法轻易弯折。解决的办法也非常简单：给 spoon 对象添加一个动画组件。

到现在为止，我们还没有谈及这些对象是如何定义的。在传统的对象管理系统中，这项工作非常简单：通过对象在继承树中的位置，以及对象对外显示的方法来定义对象。我们还可以为每个对象类型创建一个类，并使用聚合技术，从而使每一个对象类都是由一些原有的对象创建而来。新创建的对象与那些原有的对象之间是有关系的，但并不只是原有对象的一个关系。这样，我们就离目标更近了。但问题是，对象仍然是在代码里创建的，这意味着每当改变某个对象的结构时，都需要重新编译。

创建对象的过程完全由数据来驱动，才是我们真正想要的。如此这般，游戏策划者就可以玩转现有的对象，而且即使是要创建新的对象类型，也不需要程序员的插手。这个系统已经拥有了完成该梦想所需的所有东西。现在，我们只需要派生出相应的文件格式来说明组件

[2] 译者注：像《骇客帝国》电影里那样。

和对象，并为游戏策划人员提供所需的工具就可以了。

1.3.3 系统的创建

有了一个合理的想法之后，就可以继续前进，去创建这个系统了。对于这个基于组件的对象管理系统，其所有的代码都在本书附带的光盘中。其内还有几个不同的组件和信息，帮助读者从头开始，理解并使用这个系统。这并不是一个完整的系统。考虑到每个游戏产品都不尽相同，我们确实也无法确定相应的系统该是什么样子。但是，我们可以非常简单地将这个系统进行扩展，使它的应用范围更为广泛。

1. 组件接口

我们要看的第一个类是组件接口。其他所有的组件接口和组件都是从这个类派生出来的。它的程序代码如下：

```
Class IComponent
{
public:
    IComponent();
    virtual ~IComponent() = 0;
    virtual bool   Init(CParameterNode &) = 0;
    virtual void   Deinit(void) = 0;
    virtual EMessageResult HandleMessage(const CMessage &);
    virtual EComponentTypeId GetComponentTypeId(void) = 0;

    CObjectId      GetObjectId(void) const;
    ICmpEntity     *GetEntity() const;
private:
    void           SetObjectId(CObjectId oId);
    CObjectId      mObjectId;

    friend CObjectManager;
};
```

我们看到了具体的程序代码，但这个类到底有什么用呢？回过头来看看其中的每一个声明。首先是构造函数（constructor）和析构函数（destructor）。虚拟的 destuctor 用来说明这个类是一个派生类，而不是实例类，除此之外，它们没有其他的作用。下面的代码就更有趣了。每个组件在初始化和非初始化时要分别调用下面两个声明：

```
virtual bool Init(CParameterNode &) = 0;
```

和

```
virtual void Deinit(void) = 0;
```

Init()函数的调用参数是 CParameterNode 对象的一个引用。CParameterNode 是组件

数据树中的一个节点。我们可以把它看成是 XML datafile 中的一个节点，我们可以从该节点及其子节点中读取数据，而不必像通常情况下那样，读取 XML 数据就必须进行词法解析。每个组件的 `Init()` 函数会向该参数节点申请其需要的数据，然后对该组件进行适当的初始化工作。在使用组件以后，`Deinit()` 函数负责清理工作，以确保释放内存和该组件所有的句柄。

下一个声明：

```
virtual EMessageResult HandleMessage(const CMessage &);
```

并不是进行纯虚拟声明。如果不需要，组件类也不必一定要对其进行覆写（override）。这个声明的返回值是一个枚举类型（enumerated type）的 `EMessageResult`，如下：

```
enum EMessageResult
{
    MR_FALSE,
    MR_TRUE,
    MR_IGNORED,
    MR_ERROR
};
```

其中 `MR_FALSE`、`MR_TRUE` 和 `MR_ERROR` 表示消息已经被处理过了。`MR_IGNORED` 则表示没有尝试去处理消息。`IComponet` 中的 `HandleMessage()` 函数不需要做其他什么工作，只是简单地返回 `MR_IGNORED`。

下一个声明：

```
virtual EComponentTypeId GetComponentTypeId(void) = 0;
```

返回的是一个枚举类型的 `EComponentTypeId` 值。枚举变量中的入口与可实例化的组件类是一一对应的关系。

接下来是：

```
CObjectId    GetObjectId(void) const;
CObjectId    mObjectId;
```

`GetObjectId()` 是一个非虚拟的成员函数。我们在此添加了一个成员变量 `mObjectId`。理想状态下，我们并不需要这个成员变量，只要 `GetObjectId()` 是一个纯虚拟的函数，就可以让 `IComponent` 类成为一个合适的接口。但在实际应用中，让 `mObjectId` 成为 `IComponent` 类的一个成员变量，还是很有意义的。这样对象管理器就可以通过下面 Private 中的两个声明来访问并修改 `IComponent`：

```
void          SetObjectId(CObjectId oId);
friend CObjectManager;
```

如果不这样做，那么每个组件类都不得不包含相同的代码，以便设置对象 ID，我们也会因此碰到方方面面的问题。

最后，来看一下这个公共函数：

```
ICmpEntity     *GetEntity() const;
```

这里又是这样一个情况，实用比好看更重要。我们之前就决定，系统中所有的对象都要包含一个组件。该组件负责实现 ICmpEntity 接口。所以，把 GetEntity() 函数安排在这里，以便从同一个对象中的其他组件来查看该组件。ICmpEntity 类的详细内容已在本书附带的光盘中提供。

2. 对象管理器

该系统的核心部分就是对象管理器。CObjectManager 类有点过于庞大，无法在这里一个一个函数地讲解，所以我们会重点讲那些核心的功能。读者可以参见随书光盘中的 CObjectManager 声明和实现的详细代码。

该对象管理器最重要的功能就是数据库。它是作为一个独立的 struct 来实现的，以便让实现的细节隐藏于用户。但我们还是要在这里窥视其一二。代码如下：

```
struct SObjectManagerDB
{
    // 静态成员类型数据
    SComponentTypeInfo
       mComponentTypeInfo[NUM_COMPONENT_TYPE_IDS];
    std::set<EComponentTypeId>
       mInterfaceTypeToComponentTypes[NUM_INTERFACE_TYPE_IDS];
    // 动态成员数据
    std::map<CObjectId, IComponent*>
       mComponentTypeToComponentMap[NUM_COMPONENT_TYPE_IDS];
    // 消息数据
    std::set<EComponentTypeId>
       mMessageTypeToComponentTypes[NUM_MESSAGE_TYPE_IDS];
};
```

SObjectManagerDB 的所有数据成员都是由一些更为复杂的数据类型组成的数组。可以把它们看成是二维数组，其中的一维在编译时就知道了，因为我们知道组件、接口和消息类型的数量。

在系统初始化的时候，这个 struct 的前两个数据成员就设置好了，并且在游戏运行的时候也不会变化。每一个组件类型会调用 CObjectManager::RegisterComponentType() 来设置这两个数组中的数据。SComponentTypeInfo 包含了创建组件所需的数据。代码如下：

```
struct ComponentTypeInfo
{
    ComponentCreationMethod        mCreationMethod;
    ComponentDestructionMethod     mDestructionMethod;
    CHash                          mTypeHash;
};
```

构造函数和析构函数这两个成员的类型是函数指针 **typedef**:

```
typedef IComponent* (*ComponentCreationMethod)(void);
typedef bool (*ComponentDestructionMethod)(IComponent *);
```

我们将创建带有这两个声明的函数，作为每个组件类的静态成员函数。它们分别负责创建和销毁组件。对象管理程序只负责处理指向组件的指针，组件内存的管理工作则作为练习题留给读者。对于这里实现的系统，标准的“新建”和“删除”功能还是有的。基于组件名称的哈希（hash）字串，可以使用变量 mTypeHash 用来查找组件的类型 ID。

对于每一个接口类型，我们都维护着一系列的组件类型信息，用以实现这个接口。这些信息保存在数组变量 mInterfaceTypeToComponentTypes[] 之中。对象管理器的 QueryInterface() 函数会用到这个数组。为简化起见，每个对象中的每个接口只允许有一个实现。这样，如果对象中没有组件来实现相应的接口，QueryInterface() 函数就返回 NULL；反之，它会返回一个组件指针，该组件是对象中惟一一个用来实现接口的组件。

ON THE CD

在游戏运行过程中，会更多地用到 mComponentTypeToComponentMap 这个数组。该数组中的每一个元素都是一个映射，即将 IComponent 的指针映射到其所属对象的 object ID。创建一个组件时，它的 object ID 和地址（以一个 IComponent 指针的形式）会被添加到这个映射中，其数组位置索引由相应组件的组件类型来决定。QueryInterface() 函数最后必须查看 mComponentTypeToComponentMap，看它是否可以通过 mInterfaceTypeToComponentTypes 这个数组的接口与一个组件指针相匹配。该函数的详细代码可从随书光盘中获得。

SObjectManagerDB 的最后一个成员是 mMessageTypeToComponentTypes。它负责跟踪哪个组件类型订阅了何种消息类型。消息的订阅是在系统初始化的时候进行的，但是在游戏运行过程中，没有终止订阅或者退订的机制。

3. 组件实例

如果游戏涉及角色之间的攻击与被攻击，那就需要一个 health 组件。其需求非常简单：

- 监视对象当前的健康状况。
- 允许查看当前的健康状况。
- 在收到 MT_TAKE_HIT 消息后，更新当前的健康值。
- 当健康值为零时，发送 MT_HEALTH_DEPLETED 消息。

4. 接口类

首先创建接口 ICmpHealth，代码如下：

```
class ICmpHealth : public IComponent
{
public:
    virtual int32 GetHealth()=0;
protected:
    static void RegisterInterface(EComponentTypeId);
};
```

为什么要首先创建这个接口，其中的函数声明 GetHealth() 就是原因所在。下面一个函数的实现代码是这样的：

```
void ICmpHealth::RegisterInterface(EComponentTypeId compId)
{
    GetObjectManager().RegisterInterfaceWithComponent(
                                         IID_HEALTH,
                                         compId);
}
```

我们需要从实现这个接口的组件中的 Init() 函数来调用上面这个函数。RegisterInterfaceWithComponent() 函数的调用是为了告诉对象管理器：compID 类型的组件会实现 IID_HEALTH。GetObjectManager() 是一个全局函数。

5. 组件类

调用组件类 CCmpHealth，它的代码如下：

```
class CCmpHealth : public ICmpHealth
{
public:
    // 静态函数
    static void          RegisterComponentType(void);
    static IComponent*   CreateMe();
    static bool          DestroyMe(IComponent *);

    // Icomponent 的虚函数
    virtual bool              Init(CParameterNode &);
    virtual void              Deinit(void);
    virtual EMessageResult    HandleMessage(const CMessage &);
    virtual EcomponentTypeId  GetComponentTypeId(void)
                                  { return CID_HEALTH; }

    // IcmpHealth 的函数
    virtual int32 GetHealth()     { return mHealth; }
private:
    int mHealth;
};
```

首先，看一下几个静态函数的实现：

```
void CCmpHealth::RegisterComponentType()
{
    ICmpHealth::RegisterInterface(CID_HEALTH);
    GetObjectManager().RegisterComponentType(
                                        CID_HEALTH,
                                        CCmpHealth::CreateMe,
                                        CCmpHealth::DestroyMe,
                                        CHash("Health"));
    GetObjectManager().SubscribeToMessageType(
                                        CID_HEALTH,
                                        MT_TAKE_DAMAGE);
}
```

这可能是 3 个静态函数中最有趣的一个。一开始，它就自己注册成 ICmpHealth 接口的一个实现者（implementer）。然后，它会调用对象管理器中的 RegisterComponentType()，告诉对象管理器有两个新的函数 CreateMe() 和 DestroyMe()，以及它将使用的名称（以一个哈希值的形式）。最后，它会把这个组件注册成 MT_TAKE_DAMAGE 消息的接收者。

剩下的两个函数用来负责创建和销毁组件。我们可以简单地使用标准的 new 和 delete 来完成这些工作。

```
IComponent *CCmpHealth::CreateMe()
{
    return new CCmpHealth;
}

bool CCmpHealth::DestroyMe(IComponent* pComponent)
{
    delete pComponent;
    return true;
}
```

在这个地方，应该多花些工夫设计一个度身定制的内存管理系统。这样做是有好处的，因为会有大量的组件创建和销毁工作。不同的组件类型甚至可以使用不同的内存分配方案，只要保证创建和销毁函数匹配每个类型即可。

现在来看一下 IComponent 里面定义的几个函数。下面这个函数：

```
bool CCmpHealth::Init(CParameterNode &compNode)
{
    mHealth = compNode.GetInt("Health");
    if (CParameterNode::GetLastResult() != EPR_OK)
        return false;

    return true;
}
```

会读取组件所需的数据。compNode 是组件级的参数节点。对函数 GetInt（"Health"）的调用可以得到名为“health”的子节点的整数值。为简化起见，假设 compNode 下面只有一个节点子层。如果数据读取失败，CParameterNode 类就会标识出错。调用 GetLastResult()可以检测是否出错。

接下来这个函数：

```
void CCmpHealth::Deinit(void)
{
}
```

其实什么都不干。惟一的数据成员是一个 int 值，所以没有太多需要清除的东西。如果组件占用了某些内存，就得在这里将其释放。

再来看一下消息处理函数：

```
EMessageResult CCmpHealth::HandleMessage(
                    const CMessage &Message)
```

```
{
    int newHealth;
    switch(Message.mType)
    {
    case MT_TAKE_DAMAGE:
        newHealth = mHealth -
                    reinterpret_cast<int>(Message.mpData);
        if(newHealth <= 0 && mHealth > 0)
        {
            GetObjectManager().PostMessage(
                GetObjectId(), MT_HEALTH_DEPLETED);
        }
        mHealth = newHealth;
        return MR_TRUE;
    }

    return MR_ERROR;
}
```

收到 MT_TAKE_DAMAGE 消息后，我们就知道它的数据字段是 int 类型的，表示的是受伤害程度。如果健康值由正数变为负数，说明对象的健康值用尽了，我们就需要把这个消息通知给其他组成这个对象的组件。为此，就要调用对象管理器中的函数 PostMessage()。当然了，我们必须制定接收这个消息的对象。为此，就需要调用从 IComponent 那里继承的函数 GetObjectId()。如果想让所有的对象都能收到某个消息，可以调用函数 BroadcastMessage()。然后，再设定新的健康值，并返回 MR_TRUE，表示一切正常。如果不能识别接收到的消息，这就算是一个程序错误。对于那些已经订阅了，但却决定不予理会的消息，我们可以使用 MR_IGNORED。

最后两个函数比较直观易懂，我们已经通过声明把它们定义为内联函数。

1.3.4 总结

我们已经看到，静态继承层次已无法应对当代游戏产品的挑战。本文向大家展示了一个一个有力的替代方法，即基于组件的对象管理系统。这个系统有足够的灵活性，可以应对需求的不断变化。游戏策划师们由此也可以自己创建并修改对象，而不需要程序员的插手。这样就可以更快速地测试游戏设计上的变更，并且可以有更多的时间去反复修改游戏设计方案，由此就可以开发出更优秀的游戏产品。

以前，一旦游戏的设计方案发生变化，程序员们就不得不把精力花在重构系统的工作上。有了这个系统，他们只需要实现游戏设计方案中需要的功能，就可以继续推动游戏的开发了。

1.4 用模板实现一个可在 C++中使用的反射系统

Artificial Mind & Movement 公司，Dominic Fillion

dfilion@hotmail.com

从原始的 C 语言谦虚地演变出来以后，C++编程语言已经有了相当大的发展。它已经从原先那个为了提供“可读汇编代码”的程序语言（C 语言），发展成为一个拥有大量强力工具，支持结构化编程、过程化编程、面向对象编程和泛型编程的编程语言。选择上述哪种模式，主要看哪种模式最适合自己（或者最适合自己的问题域）。C++语言最新的变种是 C#语言。C#语言增加了很多新的功能，包括垃圾收集、即时编译和其他很多功能。但最值得一提的是反射。

反射机制可以让程序检查（有时也可以修改）其自身在运行时的高级结构。在程序进行编译的过程中，对于那些很容易就能在程序外部观察到的结构，它们的汇编代码通常会被抽取出来。在反射编程语言中，这些结构化的信息被保留为元数据。很多著名的公司，如微软公司，从 COM 到.NET，投入了大量的精力去开发 reflection 机制的基础应用架构。

现在，所有关于程序代码这个特性的内容，我们都称之为“*自我感知*”。这个特性已经让《星际迷航》（Star Trek）的粉丝们爽歪了。但是，这个特性是如何在真实世界里帮助我们的呢？说得更具体点，如何利用这个特性，才能让自己做出来的四条腿的长毛僵尸巨人比别人做出来的更好呢？

Reflection 有很多用途。一个最常见的用途就是将外部解释型的脚本代码与实际的 C++引擎代码进行绑定。例如，假设有一个类 `CbigAssWhooping-Weapon`。把这个类对脚本语言开放，脚本引擎就可以很容易地、自然而然地看出这个类的类型和结构信息，如属性和函数，并自动地路由脚本，去调用 `CbigAssWhoopingWeapon` 的函数和存取器。

其他的应用实例还有：自动地将游戏中的对象进行序列化操作（加载/保存），将其转换成 XML 数据；或者提供类对象的基本的网络持久性。本文首先讨论实现 reflection 的细节问题，然后再分析更多 reflection 的应用实例。

因为本文会充分利用模板的高级特性，所以其前提是假设读者对模板的知识非常熟悉。关于如何使用模板的更多细节内容，会随着文章内容的发展给出相应的解释。

1.4.1 需求

我们的目标是创建一个界面友好的、可在 C++中使用的 reflection（反射）系统。这个系统的需求如下：

高效：现在，“游戏开发”一词通常是指游戏机游戏的开发。我们希望这个 reflection 系统轻便小巧，这样它就可以在内存容量有限的游戏机平台上运转了。

跨平台：同样，这也是游戏机游戏开发领域的一个问题。所以，我们不能使用编译器相关的代码，例如解码 PDB 文件。除此以外，没人会热衷于把调试信息打包捆绑到最后发行的游戏产品中。

透明：不能为了使用 reflection 的特性，而要求程序员去改变他们常用的编码方式。应该让 reflection 系统自己嵌入到用户的程序代码中，而不是反过来让用户的程序代码来适应 reflection 系统。

不影响编译工作：这个系统不能反过来影响编译工作的性能，或者让编译过程更加复杂。这样就不用专门去设计一个 C++解析器，让它在编译的时候读取 C++代码，查看类型信息了，这样的过程通常都会很慢，而且极容易出错。

灵活性：程序人员应该可以精确地控制自己的类，决定对外公开哪些接口。通常情况下，大家只希望对外公开某个类中有限的几个功能，而不是整个类。

健壮可靠：该系统应该是类型安全的，能够捕获任何常见的错误（例如，在脚本中试图将某个特定的类型的值设置为错误类型的属性）。所以，这可能是第 1 个暗示：该系统将会使用 template（模板）。

这个 reflection 系统的功能可以拆分成 3 个主要的部分：

注册功能：让程序人员告诉我们，哪些数据成员是他想对外公开的。

内省功能：可以让程序人员检查某个类所支持的属性和函数的名字，以及支持的类型。

操作功能：通过一个外部接口（脚本、GUI 或文件等），可以从外部调用程序员的代码。

reflection 操作通常要对外公开某些代码元素，如数据成员或类的函数。由于本文要谈及很多基础的内容，所以我们会抓住一个重点：实现一个对外公开数据成员存取器的 reflection。读者可以很容易地进行扩展，使之支持类函数的公开。而对于功能函数的实现，则把它作为练习题留给读者。

属性（properties）是这个 reflection 系统的核心元素。为了定义的方便，一个 property 只是一个简单的数据属性，对它的访问由读写存取器来控制。在大部分时间里，存取器只是简单地返回或者设置一个 protected 或 private 变量；而在其他时间里，存取器会计算出一个结果，或者改变一个内部状态。下面就从底层开始，自下而上围绕属性来创建一个 reflection 系统，并随着需求的增加，添加更复杂的功能。

1.4.2 第 1 部分：运行时类型信息

即使是一个最基础的 reflection 系统，如果没有运行时类型信息（*runtime type information*，简称 RTTI）的概念，或者说不能在运行时动态地判断对象的类型，那么该系统是无法运转的。

为此，可以使用 C++自带的 RTTI 系统。但是，如果使用我们自己的 RTTI 系统，就可以简化 property 系统的实现工作，同时也有助于提高系统的可扩展性，并确保它是最优化的系统。虽然现在看上去，好像是在切入正题之前绕了一个大圈子，但是实现这样一个定制的 RTTI 系统，以后使用起来就方便多了。在文章的第 2 部分，我们会根据已经了解的知识来实现 reflection（反射）支持。RTTI 系统也会用到 template（模板），所以在开始实现 reflection（反射）系统之前，需要先准备一些有关模板的工作。

为了能够在运行时判断出某个对象的类型，这个对象必须在类的每个层次上实现一个虚函数（virtual function）覆写。该函数会返回对象的类型信息。类型信息的结构如下：

- 类的名称
- 惟一的标识这个类的 ID（class ID）
- 一个指向祖先类 RTTI 信息的指针
- 一个调用工厂函数的回调函数

类的名称是保存在一个字符串中的、未被修饰过的类的名字。Class ID 是一个由用户提供的 32 位数字，具有惟一性。在比较对象的类型，以及将对象的类型信息进行序列化以用于文件或网络协议方面，class ID 可以有效地提高这些工作的效率。

指向祖先类 RTTI 信息的指针是自描述型的（self explanatory）。

回调函数会指向一个工厂函数，后者会创建一个属于该类型的新的对象实例。如果在只知道某个类的 class ID 的情况下，要想在运行时创建一个未知类型的对象实例，那么这个回调函数就非常有用。

RTTI 结构信息应该形如：

```
typedef DWORD          ClassID;
typedef CBaseObject*   (*ClassFactoryFunc)( ClassID );

const DWORD CLASSNAME_LENGTH = 32;

class CRTTI
{
public:
    CRTTI( ClassID CLID, const char* szClassName,
           CRTTI* pBaseClassRTTI, ClassFactoryFunc pFactory );

private:
    ClassID          m_CLID;
    char             m_szClassName[CLASSNAME_LENGTH];
    CRTTI*           m_pBaseRTTI;
    ClassFactoryFunc m_pObjectFactory;
};
```

正常情况下，公共数据存取器（public data accessor）（Get, Set）也会添加到这个类中，但是为了让大家看得更清楚，这里并没有把它们加进来。系统中的每一个类都包含一个自身 RTTI 结构的静态实例，并且还要实现一个虚函数以返回一个指向其静态 RTTI 结构的指针。这里要注意的是，每个类的 RTTI structure（结构）只保存一次，而不是每个实例保存一次。

1.4.3　在 RTTI 的实现中使用模板

我们可以使用宏来实现 RTTI 系统，但更巧妙的方法是使用模板来实现 RTTI 系统。

如果使用宏，就要在每个应用程序类的 RTTI 代码中粘贴来粘贴去。取而代之的是，我们可以让所有的应用程序类都从一个模板化的 CSupportsRTTI 类派生而来，并由此实现 RTTI 功能。

如果所有的引擎类都是从 CSupportsRTTI 继承得到的，那它们是如何最终从真正的祖先类派生出来的呢？是否需要根据每一个可能的祖先类，去编写 CSupportsRTTI 类的变种呢？答案是不需要，因为这样来定义模板是完全合法的：

```
template < class BaseClass >
class CSupportsRTTI: public BaseClass
{
};
```

注意：对于模板化的类，其祖先类实际上是它的一个模板参数。这样，当某个应用程序类从 CSupportsRTTI 派生出来时，我们还可以这样指定 CSupportsRTTI 的祖先类：

```
class CMyClass : public CSupportsRTTI<CMyBaseClass>
{
};
```

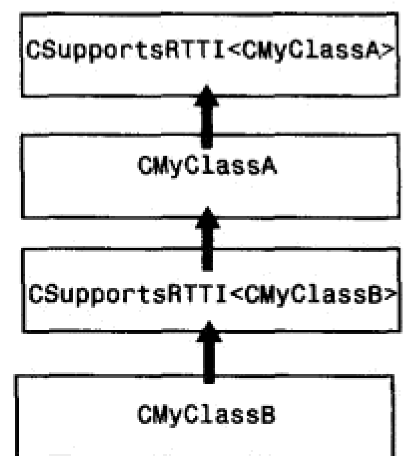


图 1.4.1　RTTI 的层级结构

由于所有的应用程序类都是从 CSupportsRTTI 派生得到的，所以我们可以很方便地把 RTTI 功能的实现安排在每个类及其祖先类之间的位置上。我们要做的就是把 RTTI 代码注射到每个类及其祖先类之间的位置上，相关的层次结构如图 1.4.1 所示。

再来看一下 RTTI 功能的其他代码。

```
#include <typeinfo.h>

template <class T, class BaseClass, ClassID CLID>
class CSupportsRTTI: public BaseClass
{
public :
    const static ClassID ClassCLID = CLID;

    CSupportsRTTI();

    static T*         Create();
    static void       RegisterReflection();
    static inline     CRTTI* GetClassRTTI() { return &ms_RTTI; }
    virtual           CRTTI* GetRTTI()      { return &ms_RTTI; }

protected :
    static CRTTI ms_RTTI;
};
```

```
template <class T, class BaseClass, ClassID CLID>
CRTTI CSupportsRTTI<T, BaseClass, CLID>::ms_RTTI
    ( CLID, typeid(T).name(), BaseClass::GetClassRTTI(),
    (ClassFactoryFunc)T::Create, NULL );

template <class T, class BaseClass, ClassID CLID>
inline CSupportsRTTI<T, BaseClass, CLID>::CSupportsRTTI()
{
}

template <class T, class BaseClass, ClassID CLID>
T* CSupportsRTTI<T, BaseClass, CLID>::Create()
{
    return new T();
}

template <class T, class BaseClass, ClassID CLID>
void CSupportsRTTI<T, BaseClass, CLID>::RegisterReflection()
{
}
```

RTTI 的信息结构是作为模板中的一个静态成员置入的。每个模板实例(在给出的应用程序中，每个对象类型只有惟一的模板实例）都会产生属于自己这个静态成员的实例，而这恰恰就是我们想要的。GetRTTI()是一个虚函数，它会根据多态对象的类型，返回正确的 RTTI 信息。

GetClassRTTI()是一个函数，用来查询某个指定类的 RTTI 信息，如 CClassType::GetClassRTTI()。这里要注意的是，静态成员函数可以将另外一个静态成员函数隐藏在祖先类中，这也是我们所需要的。CClassType::GetClassRTTI() 隐藏的是 CBaseClassType::GetClassRTTI()。

Create()函数是类工厂函数。它只是简单地为实例化的模板类型分配一个新的实例。T::Create()会解析为 CSupportsRTTI<T、BaseClass、CLID>::Create()，这是 T 的基础类。

应该注意那几行用来声明静态 RTTI structure 的代码。在这里，RTTI structure 是静态构造的，所有相关的 RTTI 构造器参数都会传递到 RTTI structure 中：

- CLID 是惟一的、32 位的类标识符号。这是一个模板参数。
- Typeid(T).name()是类的名称字串。这里好像是在欺骗大家：这样还是利用 C++的运行时类型信息系统在此基础上构建自己的系统啊？这和前面所说的编写自己的 RTTI 系统可是背道而驰的啊。其实不是这样的。这里的 typeid (T) 会在编译的时候被编译器解析，所以我们并非真正在使用 C++的动态 RTTI 结构。实际上，大家会发现，即使关闭 C++的 RTTI 系统，这个代码依然可以工作如常。在实际运行中，这个代码片段只会简单地让编译器为对象 T 创建并填写一个 type_info structure，然后返回相关的字串，并不需要进行运行时的多态检验。
- BaseClass::GetClassRTTI()是基础类的 RTTI structure。BaseClass 是一个模板参数。
- CSupportsRTTI<T、BaseClass、CLID>::Create()提供了指向工厂类函数的

指针，并自动由模板实例化。当只知道 class ID 或类的名称字串时，创建类的实例还是非常有用处的。

在程序启动时，当静态变量初始化完毕，我们就可以把所有的 RTTI structure 添加到一个全局的 RTTI 管理器中。但是必须小心，尽量避免静态变量构造器之间的依赖关系，因为编译器无法保证以指定的顺序来初始化静态变量，而让一个静态变量 *A* 在其构造器中引用另一个静态变量 *B* 会造成程序的混乱。幸运的是，这些静态变量构造器的参数并不需要这样的依赖关系，这样初始化的顺序就与 RTTI 系统无关了。

现在，创建一个支持该定制 RTTI 系统的应用程序类就非常方便了：

```
class CMyObject : public CSupportsRTTI< CMyObject, CBaseObject, 0x2e160f7a>
                // 0x2e160f7a 是一个用户提供的随机惟一 ID
{
};
```

这样，我们的 RTTI 系统就完成了。

为了能够在运行时根据 class ID 创建类的实例，我们只需要在 RTTI structure 的构造器中添加几行代码，就可以把所有的 RTTI structure 注册到一个全局管理的列表或映射表中。RTTI 管理器可以检索这个映射表，并通过一个 class ID 找到相关的 RTTI structure，然后调用相关的工厂函数。

1.4.4 关于 RTTI 的其他修改建议

对于前面描述的 RTTI 系统，还可以在以下方面进行修改和调整：

- class ID 并不是必须有的。在应用程序中，RTTI structure 是每个对象类型特有的静态实例。这样，就可以使用指向 RTTI structure 自己的指针作为分类的 ID，进行比较和赋值操作。只有需要把 class ID 保存到文件中或者在网络上传递时，才需要 class ID。
- 如果只是需要在运行时查询多态类的类型，前面的类工厂函数就是可选的。
- 如果使用 `typeid(T).name()` 语法来检索类的名称会造成编译器出错，那么这个语法也不是必需要用的。但是，将 RTTI 绑定到应用程序类的语法肯定是不太好的。我们可以把类的名称字串作为一个模板参数来传递。但是，不允许把未命名（*unnamed*）的对象作为模板的参数，所以下列语句是不允许的：

```
template <const char* szString> class CMyTemplate
{
};

class CMyClass : public CMyTemplate<"Hello">
```

因为其中的字符串“`Hello`”是个未命名的字符串变量。必须将这个字符串指定为一个常量，就像这样：

```
char szHello[] = "Hello";
class CMyClass : public CMyTemplate<szHello>
```

这样做就没问题了，但多少有点麻烦。

在创建了自己的 RTTI 系统之后，就可以在反射系统中享受它带来的好处了。将 reflection 添加到软件中意味着要扩展 RTTI 系统，使它包含其所支持属性的元数据，以及相关的函数。还要扩展 RTTI structure，使之包含一个属性对象列表。下面就来看看这些属性对象都是什么。

1.4.5 第 2 部分：属性对象

属性对象就是一个类型明确的、已命名的对象，它是内部数据表示的一个出入口。属性是一个非常抽象的概念，涉及很多定义和概念。这些定义和概念会逐渐地介绍给大家。我们要创建的属性对象是下面这 3 个属性类分层体系的一种：

抽象属性层：这个基础属性类是一个类型不明，且与任何对象类型都不相关的类。如果需要查询某个属性，但只知道该属性的名称，而不关心它的类型和其他细节，那就可以使用这个基类。

类型明确的属性层：基于这个基础属性类创建的属性对象，是属性对象一个模板化、类型明确的版本。如果要查询某些指定类型的属性，就可以使用这个属性对象。

类成员属性层：属性对象最下面的一层是另一个模板化的类。在这里，属性实际上已经绑定为某个特定类型的成员。在后面会解释它的必要性，以及它是如何从整体上来适应整个系统的。

首先来看一下抽象属性层：

```
enum ePropertyType
{
    eptBOOL,
    eptDWORD,
    eptINT,
    eptFLOAT,
    eptSTRING,
    eptCOLOR,
    eptENUM,
    eptPTR,
    eptMAX_PROPERTY_TYPES
};

class CAbstractProperty
{
public:
    virtual ePropertyType GetType() const = 0;

protected:
    const char*          m_szName;
};
```

抽象属性层只包含属性的名称和一个抽象虚拟函数，后者用来描述属性的类型。本文后面的部分会介绍如何描述属性的类型。这个抽象属性层是非常轻量级的。

```
template <class T> class CTypedProperty: public CAbstractProperty
{
```

```
public:
    virtual ePropertyType GetType() const;

    virtual T    GetValue( CBaseObject* pObject ) = 0;
    virtual void SetValue( CBaseObject* pObject, T Value ) = 0;
};
```

类型正确性的检查工作是在属性对象的下一层中实现的。这是一个模板化的类，它的数据类型是这个属性作为一个模板参数所具备的类型。这个层包含抽象的模板化的函数，用来获得或者设置属性的值。这个层同样也是非常抽象的，在功能实现上没什么作为。同样，在后面的文字中可以看到这个类是如何实现 GetType() 函数的。

这里要注意的是，属性希望以 GetValue() 和 SetValue() 函数的参数的形式，来传递它的拥有者（即从中获得这个属性的对象）。属性类不一定必须是某个特定的类的实例。它会把对一个特定数据成员的存取绑定到一个对象类型上。一个类的所有实例会共享同一个属性对象。为此，程序中所有的类必须最终从一个公共类（如 CBaseObject）派生得到。CBaseObject 不能为空，而且为了一致性，还需要一个公共祖先类。

```
template <class OwnerType, class T> class CProperty : public CTypedProperty<T>
{
public:
    typedef T    (OwnerType::*GetterType)();
    typedef void (OwnerType::*SetterType)( T Value );

    virtual T    GetValue( CBaseObject* pObject );
    virtual void SetValue( CBaseObject* pObject, T Value );

protected:
    GetterType   m_Getter;
    SetterType   m_Setter;
};
```

在类成员属性层上，我们最后就得到了足够的信息（类型和类的类型），可以实现一个属性类。

属性类管理着自己的数据成员，并通过标准的 getter（取值）和 setter（赋值）存取器来访问数据成员。属性类可以有一个直接指向其管理的数据成员的指针。但是，这样就要对外公开这个属性类私有的细节信息。使用标准的存取器可以保持自己的数据是私有的，这也更符合面向对象程序开发的规则。存取器应该采用标准化的格式，即 T Get() 用于取值操作，Void Set (T Value) 用于赋值操作。

记住这些原则，模板化的属性类可以为它的那些存取器回调函数进行类型定义（typedef，见前面的代码片段）。属性类会用这些函数来存取其相关的数据。指向函数指针的指针也保存在属性对象中。

通过属性来进行取值和赋值操作需要调用相关的回调函数。

```
template <class T>
T CProperty <OwnerType, T>::GetValue( CBaseObject* pOwner )
{
    return pOwner->*m_Getter();
```

```
}

template <class T>
void CProperty<T>::SetValue(CBaseObject * pOwner, T Value )
{
    if ( !m_Setter)
    {
        assert( false ) // 只读属性是不能写的
        return;
    }
    pOwner->*m_Setter( Value );
}
```

可以看到，在这里使用了 C++的“->*”（指向成员）操作符。这是必需的，因为我们在调用一个类对象成员的函数。

属性结构现在万事俱备，可以打包数据类型并可通过标准的取值函数和赋值函数来访问它了。属性结构可以嵌入到对象自身中，也可在对象中使用。

1.4.6 属性的存储

属性与某个特定的对象类型相关联。我们可以很自然地扩展这些类的 RTTI 信息，使之包含一个自身属性的列表。我们还需要一个全局的属性管理器，来管理所有属性的分配和删除。

本文的实现方法，是把所有属性放到一个标准的 C++模板库（Standard Template Library，简称 STL）列表中，其存储管理模式如图 1.4.2 所示。

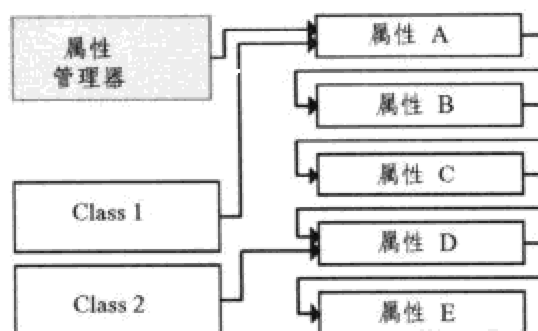


图 1.4.2 属性的存储管理

这个全局属性系统管理器会管理一个包含所有属性对象的全局属性列表。属性会注册并顺序地添加到这个列表中，也就是说，属于同一个类型的所有属性在列表中是连续存储的。每个类的 RTTI 结构中也包含一个列表，保存着其相关的属性对象。

1.4.7 属性类型

属性是一个定义了类型的构造。它必须允许系统查询它的类型，并与其他的属性类型进行比较。一个简单的实现方法是，声明一个枚举类型，来指定所有计划支持的属性类型：整型、浮点型、字节、字、双字、字符串、颜色、枚举、矢量和对象指针等。让 reflection 的使用者把某个属性的类型指定为注册函数的参数是个足够简单的做法。不过，还有一种更好的做法，即允许使用者把属性的类型指定为一个模板参数。这样，就不用使用如下的语句：

```
RegisterProperty("MyProperty", eptINT );
RegisterProperty("MyProperty2", eptPTR );
```

取而代之的是使用这样的语句，它更为自然，也不太容易出错：

```
RegisterProperty<int>( "MyProperty" );
RegisterProperty<CMyClass*>( "MyProperty2" );
```

为此，模板类型无论如何必须通过编译程序，与相应的类型枚举值关联起来。为了完成这项工作，可以定义一个其中包含一个静态数据成员的 CPropertyType 模板类。在每次进行模板说明时，模板中的静态数据成员会被实例化一次。这样，CMyTemplate<CmyObject>::ms_MyStaticDataMemeber 会引用一个不同的静态数据成员，而不是 CMyTemplate<Cmy-OtherObject>::ms_MyStaticDataMember。利用这个事实可以让编译程序自动地为每个惟一的类型生成一个静态数据成员。这个惟一的类型是我们计划在属性对象中使用的类型（一定要记住，是为每个惟一的类型，而不是为每个类型引用生成静态数据成员）。用这种方法，就可以将惟一的静态数据成员与属性对象所使用的代码中引用的所有类型关联起来。

通过使用模板的特化，可以将类型与正确的枚举类型关联起来。下面就是结果代码：

```
template <class T> class CPropertyType
{
public:
    static ePropertyType ms_TypeID;
};

template<class T> ePropertyType CPropertyType<T>::ms_TypeID = eptUNKNOWN;
template<> ePropertyType CPropertyType<bool>::ms_TypeID  = eptBOOL;
template<> ePropertyType CPropertyType<DWORD>::ms_TypeID = eptDWORD;
template<> ePropertyType CPropertyType<int>::ms_TypeID   = eptINT;
template<> ePropertyType CPropertyType<float>::ms_TypeID = eptFLOAT;
template<> ePropertyType CPropertyType<char*>::ms_TypeID = eptSTRING;
```

ON THE CD

模板的特化要针对所有已知的类型进行声明。对于没有特化的类型，会将它们赋值为枚举成员 eUNKNOWN，这和默认的模板实现代码中的方法是一样的。指针属性是比较特殊的，对于它的处理，是通过一个独立的属性注册调用来实现的（请参见随书光盘中的程序代码）。

通过模板特化，可以将属性代码与惟一的、可重复的数字关联起来。一旦通过模板参数为这个数字指定了属性类型，上述的关联就可以成立。

1.4.8　属性注册钩子（Hook）函数

每个对象都会有一个对所包含属性的描述，这个描述包括属性的名称、类型和存取信息。我们需要的是一个在开始的时候描述每个类的属性，然后将这些属性存储到一个列表中的机制。

由于我们希望让 reflection 的使用者能够控制哪个数据成员可以变成有 reflection 特性，所以这些属性就必须由程序员手工地注册到使用这些属性的类中。这些属性必须在程序运行

时尽早地进行注册，以便可以尽早地使用它们。

我们在前面的文章中已经看到了 RTTI 系统是如何通过使用静态变量的实例化来初始化 RTTI 信息的。既然可以在应用程序启动的初期对静态变量进行初始化操作，那么对于类属性的初始化，这也是一个理想的时间窗口。我们需要的只是一个简单的钩子（*hook*）函数，它让每个惟一的对象类型可以注册那些与自己相关的属性。

可以在 CSupportsRTTI 模板化类中添加一个对静态函数 RegisterReflection() 的调用。我们已经对 RTTI 系统进行了修改，这样 T::RegisterReflection()就可以作为一个指向 RTTI 类的构造器的函数指针来进行传递。构造了 RTTI 类之后，它就可以调用这个函数指针，提供前面提到的钩子函数，并为类设计器提供一个机会，为某个对象类型注册相关的属性。

CSupportsRTTI<t>::RegisterReflection()的默认实现是空的。类设计器可以重写这个函数，前提是要在一个 CSupportsRTTI 的派生类中实现 RegisterReflection()。在一个派生的基础类中，一个静态函数会隐藏另外一个同名的静态函数。所以，如果 CMyClass::RegisterReflection()存在的话，CSupportsRTTI<CMyClass>中的下列代码就会传递 CMyClass::RegisterReflection()；如果 CMyClass::RegisterReflection()不存在，编译程序就会把它解析成 CSupportsRTTI<CMyClass>::RegisterReflection()：

```
    template <class T, class BaseClass, ClassID CLID> CRTTI CRTTIClass<T, BaseClass,
CLID>::ms_RTTI
        ( CLID, typeid(T).name(), BaseClass::GetClassRTTI(),
        (ClassFactoryFunc)T::Create,
        (RegisterReflectionFunc)T::RegisterReflection() );
```

如果 CMyClass::RegisterReflection()没有被定义，系统就会调用 CSupportsRTTI<CMyClass>::RegisterReflection()函数。这是一个空实现，正好适合那些没有 reflection 数据成员的 RTTI 类。

1.4.9 属性的注册

前面的文章已经为程序员定义了一个方法，让他们可以在程序运行时，来运行那些与每个对象类型相关联的代码。属性描述器负责对属性进行初始化，然后将其与类进行绑定。下面，就来看看属性描述器是如何声明的。

reflection 的使用者需要说明，在 RegisterReflection()的调用中有哪些属性。我们需要说明每个已注册属性的：

- 名称（Name）
- 类型（Type）
- 取值存取器回调函数（Getter accessor callback）
- 赋值存取器回调函数（Setter accessor callback）

在调用钩子函数之前，RTTI 系统还会设置一个惟一的静态变量，来跟踪当前正在为其注册属性的类。

然后，这个反射系统的使用者要在钩子函数中编写相应的代码，来调用静态函数 RegisterProperty()，传递前面提到的枚举类型的参数：

```
void CMonster::RegisterReflection()
{
    RegisterProperty<int>( "Health", GetHealth, SetHealth );
    RegisterProperty<char*>( "Name", GetName, SetName );
}
```

属性的这些参数会被添加到一个新的属性对象中，并链接到 RTTI 结构和属性的全局列表中。

在这个操作之后，每个类的 RTTI 结构中就包含了一个属于自己的属性对象列表。在这个 reflection 系统的编码过程中，应异常小心，以确保该系统能够尽可能地对其使用者透明。现在，该看看这个系统的实际应用情况了。

1.4.10 脚本应用

最后，该看看这个倾注了所有努力开发出来的抽象系统，是如何在实际的游戏制作中加以应用的。应用程序多种多样，功能也非常强大。属性系统为程序员提供了一个非常省力的方法，来对外公开特定的数据成员。程序员不必改变其定义的类的内部细节，只需要为每一个对象类型写几个短小的钩子函数，告诉系统某个特定的类有哪些可用的属性（就可以了）。

在游戏产品的开发中，reflection 最常见的用法是作为脚本的胶连接口。脚本通常需要调用核心引擎来影响游戏机制。这个 reflection 系统为此类跨域的通信提供了一个理想的传输通道。一个只有一行代码的脚本可能是这样的：

```
Player.Health = Player.Health - Monster.AttackDamage;
```

全局脚本调用可以打包到独立的类中，然后再注册成属性。代码可能是这样的：

```
void CPlayer::RegisterReflection()
{
    // 假设 CPlayer 包含存取器函数 GetHealth()和 SetHealth()，用于访问属性 health（健康值）的
数据
    RegisterProperty<int>( "Health", GetHealth, SetHealth );
};

void CMonster::RegisterReflection()
{
    // 假设 CMonster 包含存取器函数 GetaAttackDamage()和 SetAttackDamage()，用于访问属性
health（健康值）的数据
    RegisterProperty<int>( "AttackDamage", GetAttackDamage, SetAttackDamage );
}

void CGlobalScript::RegisterReflection()
{
    // 假设 CGlobalScript 包含存取器函数 GetPlayer()和 GetMonster()，用于访问脚本子对象。
    // 只读属性
```

```
    RegisterProperty<CPlayer*>( "Player", GetPlayer(), NULL );
    // 只读属性
    RegisterProperty<CMonster*>( "Monster", GetMonster(), NULL );
}
```

一个脚本引擎可以使用 reflection 查出全局脚本类对外公开了哪些对象。这样，它可能会发现有一个名为 Player 的属性，访问这个属性会返回一个 CPlayer 对象。反过来，系统可以查询这个对象，看它是否支持一个名为 Health 的属性。在这个时候，脚本引擎就可以使用 reflection，直接检索到对象 Player 的 health 值。同样，脚本还可以查询 monster 的 attack damage（攻击力）值，并计算结果。这个反射系统可以自动对 player 对象调用 SetHealth()来设置新的值。因为每个属性都包含了自己的类型 ID（type ID），所以脚本引擎还可以使用该 reflection 系统，辅助进行脚本的类型检查工作。

1.4.11 Tweaker 应用

对编辑器应用程序来说，对象公开的属性信息对自动地公开可调整的数据是非常有用的。随书光盘中提供了这样一个应用实例。

在这个情景下，程序员写了某个类，假设叫 CplayerStats，这个类公开了某些属性。然后，编辑器程序可以使用这个属性信息，为这个对象创建一个通用的属性页 GUI 接口，它会为每一个属性类型显示一个 GUI 控件。

这样的应用可以很大地提高游戏的数据驱动特性。一个典型的应用案例可能是 AI 程序员创建了某个游戏类，来控制某些 AI 功能。经过几次实验之后，这个 AI 程序员也许会意识到，AI 系统变量的某些方面有些太过随意和武断了，最好能够让游戏策划人员来做些调整。AI 程序员可以把他的一些存取器作为属性进行公开，这样就可以很容易地将他自己代码的某些方面变成可调整的。“tweaker（调整器）”程序或者编辑器，可以扫描这些属性，并为游戏策划人员提供一个动态的接口。这样，数据驱动的整个策划流程就会变得非常流畅，而且紧密集成。

1.4.12 其他应用

关于属性的应用，这里还有其他几个例子。

隐式序列化（Implicit Serialization）：可以使用属性以通用、开放的格式，自动地加载或保存游戏的某些特定对象。XML 本身就是一个非常适合用来保存属性数据的文件格式。

简单网络持久性：可以使用属性发现哪些数据类型需要在网络上进行同步。通过这样的一个系统，就可以实现游戏对象的简单网络持久性。

生成日志文件：对属性的访问可以保存成日志文件，以方便脚本的调试工作。

1.4.13 总结

本文详细地讲解了一个通用系统的实现。这个系统允许程序代码在运行时公开其部分结

构（更确切的提法是数据成员）。由于使用了 template（模板），这个系统非常棒，不但是跨平台的、类型安全的，而且对使用者也是透明的。

对于这个系统实际可能的应用，本文只介绍了一些皮毛而已。另外，虽然我们没有在这里讨论，但 reflection 其实还可以包含类函数公开的函数。而且，如果把模板类型参数用于类型正确性检查，并将之绑定到代码中，template 也可以同样用于类函数。

1.4.14 参考文献

[BIL00] Bilas, Scott. “A Generic Function-Binding Interface.” In *Game Programming Gems*, 56–67. Charles River Media.

[CAF01] Cafrelli, Charles. “A Property Class for Generic C++ Member Access.” In *Game Programming Gems 2*, 46–50. Charles River Media.

[JEN01] Jensen, Lasse Staff. “A Generic Tweaker.” In *Game Programming Gems 2*, 118–126. Charles River Media.

[OLS00] Olsen, John. “Stats: Real-Time Statistics and In-Game Debugging.” In *Game Programming Gems*, 115–119. Charles River Media.

[POU02] Pouratian, Allen. “Platform-Independent, Function-Binding Code Generator.” In *Game Programming Gems 3*, 38–42. Charles River Media.

[STR97] Stroustrup, Bjarne. “Templates.” In *The C++ Programming Language, Third Edition*, 327–354. Addison Wesley, 2000.

[WAK01] Wakeling, Scott. “Dynamic Type Information.” In *Game Programming Gems 2*, 38–45. Charles River Media.

1.5 可加速 BSP 算法的球体树

Artificial Mind & Movement 公司，Dominic Filion
dfilion@hotmail.com

多年以来，空间二叉树（Binary Space Partitioning tree，简称 BSP 树），一直都是 3D 编程人员的必需品。虽然现在 BSP 算法已经不像以前那么流行了，但是它仍然广泛应用在 3D 引擎开发中的很多关键领域，如可视性预处理、碰撞检测和多边形排序。很少有算法能像 BSP 算法这样，可以成功地解决众多领域的诸多问题。

随着硬件处理能力和人们对游戏预期的不断发展，以及业界对增加多边形数量的需求的与日俱增，BSP 算法也显露出了一个弱点：冗长的预处理时间。构建一个 BSP 树，其时间复杂度为 $O(n\log^2(n))$，而对于下一代的视频图形卡，现在最多每帧可处理百万个多边形，所以采用一个时间复杂度为 $O(n\log^2(n))$的预处理算法来处理每一个多边形，可不是什么好的选择。即使对于较小的数据集，BSP 算法也是一个速度上的阻碍，它使得某个引擎的工具集无法从"20 秒内预览关卡"的状态，加速到"即时预览关卡"的状态，而后者在速度上的显著改进，可以让我们的关卡设计师以最快的速度进行设计。

本文提出了一个算法，可以有效地将 BSP 算法的时间复杂度从 $O(n\log^2(n))$，降低到 $O(n\log(n))$。该技术采用了一个略显粗糙、但却更快速的分区空间（球体树）法，来优化 BSP 算法的预处理步骤。

1.5.1 BSP 算法

从文章的完整性上考虑，在这里要简要地回顾一下 BSP 算法。不过，对于文章的大部分内容，其前提还是读者对这两种算法都比较熟悉。关于 BSP 算法的更详细内容，请参考[Ranta03]。

一个 BSP 树就是一个三维空间的分割，这个三维空间由无限个平面分割成递归的子空间。我们可以用这些子空间来发现某个空间中各个多边形之间的关系。通俗的说法就是，多边形 *A* 位于多边形 *B* 的前面还是后面？

在 BSP 树的构造算法中，业界还引入了多边形汤（polygon soup）的概念。在 BSP 树的构造过程中，我们会选择其中的一个多边形作为分割多边形。首先要计算分割多边形的平面，然后将其他多边形归为两类：位于分割多边形平面前面的和位于分割多边形平面后面的。如果有某个多边形和分割平面交叉，通常的做法就是将这个多边形沿分割平面切分成 2 个多

边形，然后再将这 2 个新的多边形按上述方法进行归类。同时，我们又得到了 2 个新的独立的空间，同样也要进行递归分割，其方法是在这 2 个独立的空间中分别选择一个分割多边形，并在计算后，将这 2 个空间中的多边形进行上述归类。这个算法就这样一直递归下去，直到所有的多边形都被用作分割多边形。这时候，原来的三维空间就被分割成了多个递归的二进制空间，这些二进制空间就会形成一个闭凸区域。

1.5.2 创建 BSP 树

为了全面理解如何对 BSP 树的构造过程进行优化，以便降低时间复杂度，回顾一下 BSP 树的构造过程是非常有用的。

一个好的 BSP 树必须在下面这两方面得到很高的评分：

最小分割：一个严格的 BSP 树绝对不能有任何多边形与相邻 BSP 叶子节点重叠。也就是说，BSP 树中的所有多边形都必须进行分割，这样每个多边形就可以归类为一个，且只能是一个叶子节点。多边形的分割工作在 BSP 树的时间复杂度和对内存容量的需求方面，增加了 BSP 算法的开销。一个好的 BSP 树必须要避免这些额外的开销，方法就是选择合适的分割平面，使多边形分割的次数最小化。

平衡："平衡"意味着每个 BSP 节点的 2 个子节点（前部子节点和后部子节点）上的多边形数量大致相等。良好的平衡性可以确保以平均相同的步数来遍历 BSP 树。不平衡的 BSP 树是不可靠的，因为有些较长的分支会比其他较短分支长几十或几百倍，从而使得访问时间极不可靠。

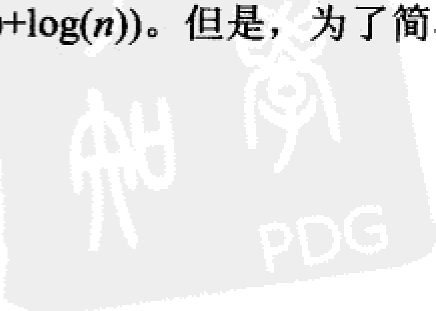
根据应用情况的不同，还有其他很多评判 BSP 的标准。有的应用也可能不会考虑上面的这两个评判标准，但是这对算法优化的核心都没有什么影响。

从本文的角度讲，我们认可上面的这两个评判标准，我们需要的是一个严格的 BSP 树。也就是说，每一个多边形都仅归类为惟一的 BSP 叶子节点。我们会假设使用多边形平面作为 BSP 的分割平面，而且当 BSP 中的每个多边形都被用作分割平面时，我们就认为这个 BSP 树完成了。作为一个回顾，下面是 BSP 树的构造步骤。

首先，BSP 会递归地将空间分割成二进制空间，分割的次数是 n。这里的 n 是多边形池中多边形的数量，也是复杂度 $n\log^2(n)$中的第 1 个 n，由此就形成了外部循环。外部循环总是需要将所有的多边形都放到 BSP 树中，所以我们不能在这个循环之外进行优化。

每一次的分割操作都必须遍历当前 BSP 分支上的所有多边形，找出最佳的候选分割平面，以便将分割次数最小化，为 BSP 树提供最好的平衡性。如果扫描到了多边形 A，那就必须将之与当前 BSP 空间中的其他所有多边形进行比较，以便判断出以多边形 A 的平面作为分割平面的话，会产生多少次分割。由于进行的是一分为二的递归分割，所以这 2 个嵌套的内部循环的每一次迭代需要遍历的多边形的个数，是上一次迭代操作涉及多边形个数的一半（理想情况下），这就是复杂度公式 $n\log^2(n)$中的 2 次方的来历。而我们要做的，就是把 BSP 算法的复杂度从 $n\log^2(n)$优化为 $n\log(n)$，即减掉 $\log(n)$。

最后，一旦选中了最佳的候选分割平面，BSP 空间中的所有多边形都会相应地归类为 BSP 前部子节点和后部子节点，并进行必要的裁剪。从技术上讲，这样的算法复杂度应该是 $n(\log^2(n)+\log(n))$。但是，为了简单起见，我们前面提到的算法复杂度估算中，并没有第 2 项



的 log(*n*)（被加数），因为根据标准的算法分析规则，与线性项（一次项）相比，二次函数主导着复杂度的最终结果。但即便如此，我们还会将这个一次项 log(*n*)（被加数）从算法中优化出去。

1.5.3 优化最初步骤

下面就把 2 个 log(*n*)循环优化出去。在这个循环中，系统会对某个特定多边形的平面进行检测，以验证如果采用这个平面来分割 BSP，会产生多少次分割以及由此产生的 BSP 树的平衡性如何？

我们想到的第 1 个办法，可能是使用古老的边界球体（bounding sphere）检测法来代替实际物体（对于本文，就是指多边形）检测，以获得这个平面相对于多边形的大致位置。在大多数情况下，多边形的边界球体严格地依赖于这个平面的前后 2 个面中的一个，这样就不必再去逐个检测每个多边形的所有顶点了。

把这个办法进一步地深化才是成功的开始：使用球体树进行分割，并赢得优化工作的胜利。球体树就是一个分层的、由球体组成的树，球体树节点中的每个球体包围了其下面所有节点上的其他球体。这样，使用这个方法，一个单一的边界球体检测操作就可以同时判断出几十或几百个多边形的位置。

1. 创建球体树

这里实际上要做的工作是采用一个粗放、松散的空间分割算法（球体树），来优化另一个细致、精确的空间分割算法（BSP 树算法本身）。球体树是一个理想的工具，因为它们可以很快地创建出来，在时间复杂度上比 BSP 树快了很多。我们不想用所使用树去优化主算法，因为在创建树所花费的时间实际上比我们通过优化算法省出的时间还要多。

那么，怎样来创建这个球体树呢？我们需要一个快速、变通的方法来分割多边形池，而且在这里，并不必追求最佳的球体树。下面是一个简单的方法，可以快速构造出一个好的球体树。

首先要计算出一个包含多边形池中所有多边形的边界球体。这就是球体树根节点的边界球体。这个球体不用是最优球体，只需要简单地计算所有多边形顶点的边界盒子，然后把这个盒子的中心点作为边界球体的球心，并对边界球体的半径进行相应的调整就可以了。

接下来，要在与边界球球心重合的 x 轴上计算出一个轴平面。这样，就可以把所有的多边形分成两组：一组位于该平面的前面，另外一组位于该平面的后面。不用分割得特别细致，如果某个多边形横跨该平面，只需要简单地计算出位于该平面两侧的顶点个数就可以了。哪一侧的顶点个数多，就把这个多边形归类为哪一侧的多边形。如果两侧的顶点数量一样，就默认这个多边形位于该平面的前面。在这个时候，最重要的事情不是精确度，而是需要有一个可重复的启发式方法，明确地将多边形归类为 A 组（位于轴平面的前面）或 B 组（位于轴平面的后面）。

一旦所有的多边形都归类完毕，就要计算每个分组的边界球体，在球体树上创建 2 个子节点，并将它们与父节点相连，然后给节点分配分好类的多边形。继续这个递归的过程，这次该使用 y 轴上的轴平面了（下次使用 z 轴上的轴平面，再下次又回到 x 轴，依次循环）。在 3 个轴向上的这种循环操作，可以保证在所有维度上保持一个大致平均的分布。一直这样递

归地创建球体树，直到叶子球体和最小的多边形的边界球体的大小一样，或者叶子节点上只有一个多边形，这样就算创建完成了。

还有一件重要的事情是，要保存每个球体树节点及其所有子节点所包含的多边形的数目。这个球体树中根节点所包含的多边形数量等于场景中所有多边形数量的总和，而叶子节点只计算那些与自己相连的多边形的数量。球体树分支则要计算在这个分支上的所有叶子节点所包含的多边形数量的总和。

该过程会生成一个球体树，对那些在球体的边界中彼此邻接成簇的多边形进行松散的分组。图 1.5.1 是一个球体树结构的简单示意。

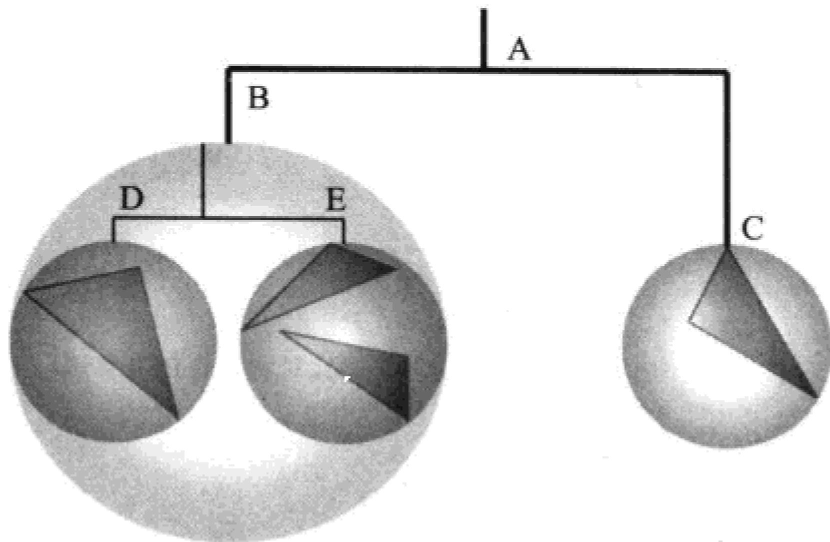


图 1.5.1　球体树的简单示意

2. 优化 BSP

现在就可以使用这个球体树来加速 BSP 树的创建过程了。

我们要对每一个多边形进行候选分割平面的检测。为了判断某个多边形是否是一个好的候选分割平面，应测试它所带来的多边形分割次数和对 BSP 树平衡性的影响，并将它与 BSP 树这个分支中的其他所有多边形进行比较。通过使用球体树，这个比较工作就可以在球体之间进行，而不用在每个多边形之间进行。

球体树之间的对比测试工作是从根节点开始的。根节点包含着所有的多边形。对于每一个球体树节点，应将它的边界球体与候选分割平面进行对比测试。如果边界球体完全位于该分割平面的前面或后面，那就可以知道，这个球体树分支上的所有多边形都不需要进行分割。至于 BSP 树的平衡性，也可以通过查找该分支中的多边形数量来计算。

如果球体与分割平面有交叉，该球体树节点中的多边形可能会、也可能不会和这个分割平面交叉。这种情况下，就必须递归地测试球体树的子节点，直至达到完全位于该分割平面的前面或后面的球体子节点。

如果无法找到这样的子节点，也就是说，已经到了最后一个叶子球体，但该球体仍然与分割平面交叉，那么这个时候，就只好把叶子节点球体中所有的多边形逐个与分割平面进行比对。这实际上是这个方法中惟一需要进行单个多边形比对的情况。

大多数情况下，在这个球体树中，只要经过很少的几个节点就可以将整个场景空间中的所有多边形归类完毕：位于分割平面前面的、位于分割平面后面的以及和分割平面交叉的。

这时就可以把 BSP 算法复杂度中的那个 log(*n*)去掉了。图 1.5.2 显示的是 BSP 树平面与球体树节点之间的位置关系。执行 BSP 比对测试的程序代码请参见程序清单 1.5.1。

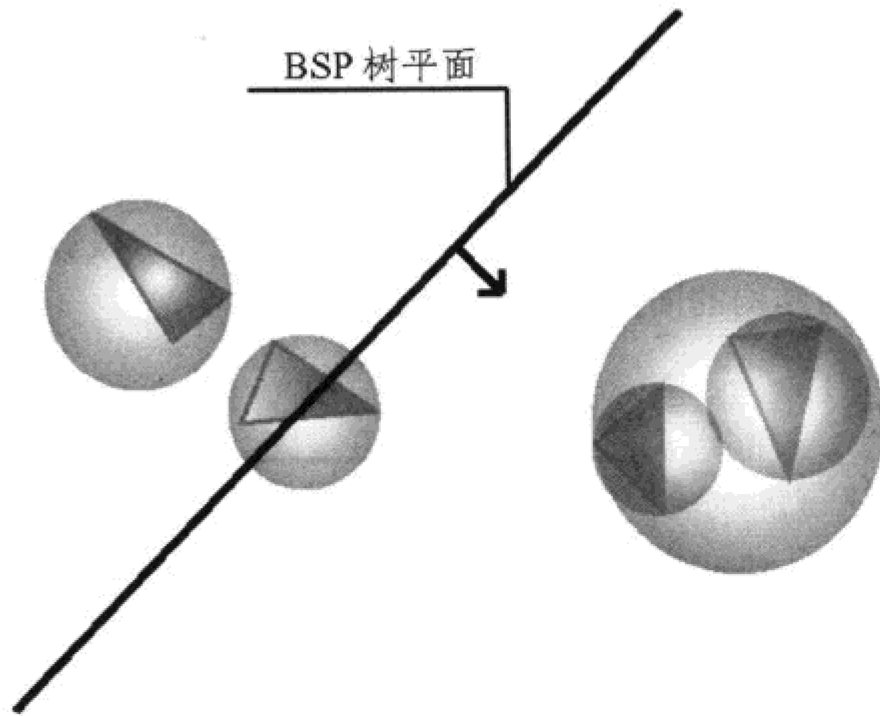


图 1.5.2 使用球体树进行分割平面比对测试的 BSP 算法

程序清单 1.5.1 测试候选分割平面

```
void CBSPTreeBuilder::TestSplitCandidate(
     CSphereTreeNode* pSphereTreeNode, bool& bTerminateEarly )
{
    // 用候选分割平面测试球体树节点
    float fDistance = m_pCurCandidate->m_Plane.GetDistance(
       pSphereTreeNode->m_Bounds.m_vPosition );
    if ( fDistance < -pSphereTreeNode->m_Bounds.m_fRadius )
    {
        // 球体树节点完全位于平面的后面
        m_dwBackCount += pSphereTreeNode->m_dwPolyCount;
    }
    else if ( fDistance > pSphereTreeNode->m_Bounds.m_fRadius )
    {
        // 球体树节点完全位于平面的前面
        m_dwFrontCount += pSphereTreeNode->m_dwPolyCount;
    }
    else
    {
        // 球体树节点可能与分割平面交叉
        if ( pSphereTreeNode->m_pPolygons )
        {
            // 这是一个球体树叶子节点，所以要单独测试每一个多边形
            CPolygon* pCurPoly =
                (CPolygon*)pSphereTreeNode->m_pPolygons;
            while ( pCurPoly )
            {
                // 检测当前候选分割平面的分割次数
                if ( pCurPoly != m_pCurCandidate )
```

```
                {
                    switch ( pCurPoly->GetSide(
                            m_pCurCandidate->m_Plane ) )
                    {
                    case CPolygon::epsFRONT :
                        m_dwFrontCount++;
                        break;
                    case CPolygon::epsBACK :
                        m_dwBackCount++;
                        break;
                    case CPolygon::epsBOTH :
                        // 啊喔，这个多边形需要分割
                        m_dwSplits++;
                        if ( m_dwSplits > m_dwBestSplit )
                        {
                            bTerminateEarly = true;
                            break;
                            //分割次数太多了，这个候选分割平面不能用，尽早放弃
                        }
                        break;
                    }
                }
                pCurPoly = pCurPoly->m_pNext;
            }
        }
        else
        {
            // 这是球体树节点，所以要测试其 2 个子节点
            if ( pSphereTreeNode->m_pChildren[0] &&
                pSphereTreeNode->m_pChildren[0]->m_dwPolyCount > 0 )
            {
                TestSplitCandidate( pSphereTreeNode->m_pChildren[0],
                                    bTerminateEarly );
            }
            if ( !bTerminateEarly && pSphereTreeNode->m_pChildren[1] &&
                pSphereTreeNode->m_pChildren[1]->m_dwPolyCount > 0 )
            {
                TestSplitCandidate( pSphereTreeNode->m_pChildren[1],
                                    bTerminateEarly );
            }
        }
    }
}
```

3. 砍树

不过，这个算法要想正常工作，还欠缺一个重要的部分。由于 BSP 树是递归分割的，所有多边形都会归类到 2 个独立的半开空间。当在 BSP 层级中的下一个层中选择下一个候选分割平面时，这个候选分割平面必须只能和它所属的那半个空间中的多边形进行比对，绝对不可以和整个多边形池中所有的多边形进行比较。

球体树包含了所有的多边形。针对每一次 BSP 的分割操作都使用同一个球体树，不但无法生成正确或最佳的分割平面，而且还会错误地报告相应那半个空间中 BSP 树的平衡信息。为了解决这个问题，必须把球体树分割成 2 个独立的部分，因为我们已经把空间分成连续的多个平面。

球体树有一个 `Split()` 函数，它的功能是把球体树上所有位于分割平面后面的球体树节点拿掉，并把它们放到一个独立的后部球体树中，这个 back sphere tree 是该函数的返回值。

分割算法会将每一个球体节点与分割平面进行比较，这和算法中对候选分割平面的测试比对工作是类似的。如果球体节点完全位于分割平面的前面，则不用做任何操作。这说明该球体节点是前部球体树，也就是当前球体树的一部分。如果球体节点完全位于分割平面的后面，那就必须将这个节点重新链接到后部球体树。很明显，重新链接球体节点同时也会重新链接该节点所有的子节点，这就会在后部球体树上添加一个全新的分支。如果球体与分割平面交叉，在实施进一步的处理之前，首先要判断该球体节点是否是一个叶子节点。

如果是一个叶子节点，那就要将多边形逐个与分割平面进行比对。每个多边形个体都会重新链接到前部球体树节点和后部球体树节点上。如果多边形与分割平面交叉，那就要裁剪这个多边形，且被裁剪下来的两个子多边形会各自链接到相应的球体节点上。要注意的是，多边形的裁剪实际上是球体树分割过程的一部分，而不是在 BSP 树的创建过程中完成的。对于 BSP 创建程序而言，它分割的是球体树，而不是多边形。

如果不是叶子节点，那就要进行递归分割。对节点的分割会导致对其子节点的递归分割。调用 `Split()` 函数时，每个球体树节点的子节点会作为一个参数。为这个节点的父节点创建一个后部节点版本，然后子节点 *A* 会为自己创建一个后部节点版本，并链接到父节点的后部节点版本上。而子节点 *B* 也同样会为自己创建一个后部节点版本，并链接到父节点的后部节点版本上，如图 1.5.3 所示。

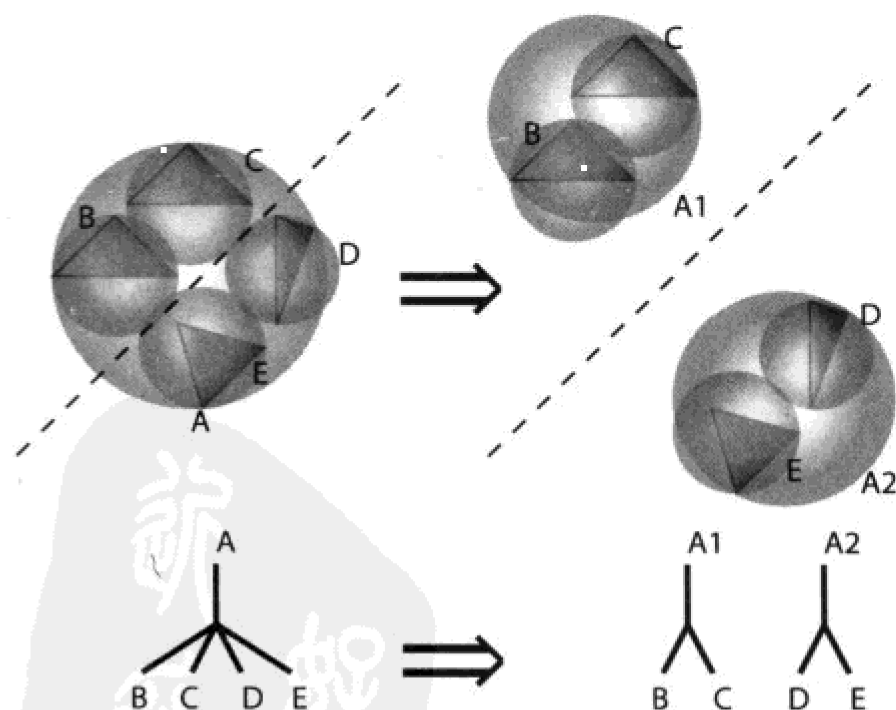


图 1.5.3 递归分割情况的示意图

还需要小心对待的是，别忘了更新前部球体树和后部球体树各自的多边形数量，特别是

在一个球体树分割操作之后。

4. BSP 极限：O(*n*)及其他

前面介绍的这个技术，可以生成一个优化的、功能良好的 BSP 树。虽然牺牲了某些运行时的最优性，却可以更多地提高算法的速度。还有一个可探讨的方法，从统计学角度上讲，只需要检测所有候选多边形的一个子集，比如 5%～10%的候选者，就可以得到一个相当好的 BSP 分割平面。这样生成的 BSP 树不一定就是高效率的，但是其生成速度大约是 $O(n\log(n)*0.1)$，与理论极限<Eqn015.eps>极为接近。虽然不是最优的方法，但是与随机选择分割平面的方法相比，只评估候选分割平面的一个子集的效率要高得多。对于预览级应用，或者确实不需要完全优化的 BSP 树，一个次优的 BSP 树就足够好了。

1.5.4　总结

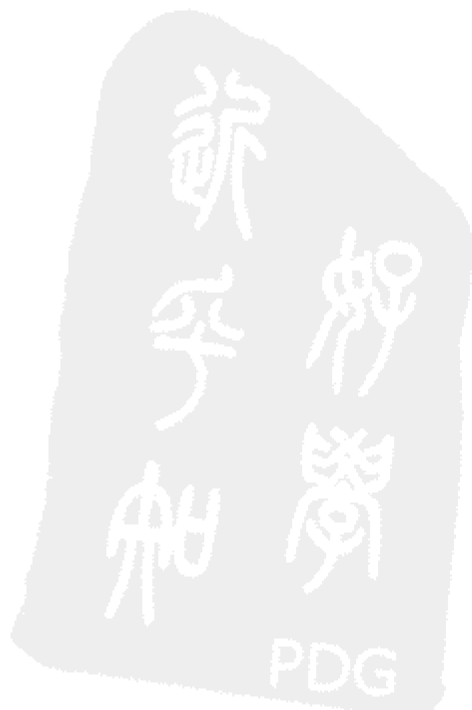
采用前面描述的算法，可以把 BSP 树的创建变成一个近乎交互式的过程。对于最后时刻的关卡调整工作，每个微小的变化都得用 10～20 分钟才能完成；而采用这个算法，关卡的调整工作只要几秒钟就能完成。当工期紧张时，这种速度上的提高是非常有价值的。

1.5.5　参考文献

[FAQ99] "BSP Tree FAQ." Available online at *http://www.gamedev.net/reference/articles/article657.asp*. August 22, 1999.

[Ranta03] "BSP Trees: Theory and Implementation." Available online at *http://www.devmaster.net/articles/bsp-trees/*. November 9, 2003.

[RatCliff01] RatCliff, John W. "Sphere Trees for Fast Visibility Culling, Ray Tracing, and Range Searching." *Game Programming Gems 2*, 384–387. Charles River Media.



1.6 改进后的视锥剔除算法

Frank Puig Placeres
fpuig2003@yahoo.com

一个典型的游戏场景通常要包含很多的对象，如果不能适当地管理好这些对象，渲染性能就会受到影响。一些库（如 OpenGL 和 DirectX）提供了一些简单的几何体管理功能，方法是对部分位于屏幕之外的多边形进行裁剪，以及提前抛弃掉那些完全位于视区之外的多边形，对它们不做任何进一步的处理。但是，在执行裁剪操作时，一个多边形的所有顶点都必须经过库管道，转换为屏幕坐标，然后再与活动视区进行比较检测。当多边形的数量不多时，这个过程是行之有效的。但是，当要渲染很多的对象，且每个对象又包含成百上千的多边形时，传送和处理所有这些顶点的系统开销就会非常高，很快就会造成系统超载。我们必须实现一个更高层的、以对象为单位的剔除算法，来有效地管理一个典型场景的渲染工作。视锥剔除法是一个著名的场景管理技术，本文对该技术进行了很聪明的改造。

1.6.1 视锥剔除

视锥剔除的工作原理是首先定义一个可视体,从一个给定的视点出发，要把当前可见的所有空间都包含在这个可视体中。如图 1.6.1 所示，这个可视体的构建是从一个金字塔开始的。金字塔的塔尖就是眼睛的位置(视点)，4 个三角面的底边与屏幕的 4 个边缘对齐。然后，用 2 个平行的平面（一个近点剪切平面和一个远点剪切平面）对这个金字塔进行平切。比屏幕更接近视点的区域（也就是屏幕到视点之间的区域），以及超出预定距离之外的区域，都会被截掉。剩下的这个几何体在数学中称为棱锥台或平截头体(*frustun*)。那些完全位于该锥台之外的对象，摄像机是无法看到的，我们会以对象为单位，一个一个地丢弃它们。

创建了视锥之后，就可以将场景中的对象逐个进行比对，看它是否位于视锥之内，从而确定其是否可见。场景中的各个对象可能会完全被一个边界几何体所包围，如球体、立方体、圆柱体或圆锥体。这些几何体是用来与视锥进行比对测试的，而不是构成场景的有效实体，这样就避免了复制渲染管道中已有内容的额外开销。与基于多边形的剔除算法相比，视锥剔除算法可以提供更高层次的系统效能。如果这个简单的边界几何体完全位于视锥之外，那么该图元之中的一个或几个对象都会被系统剔除。这里

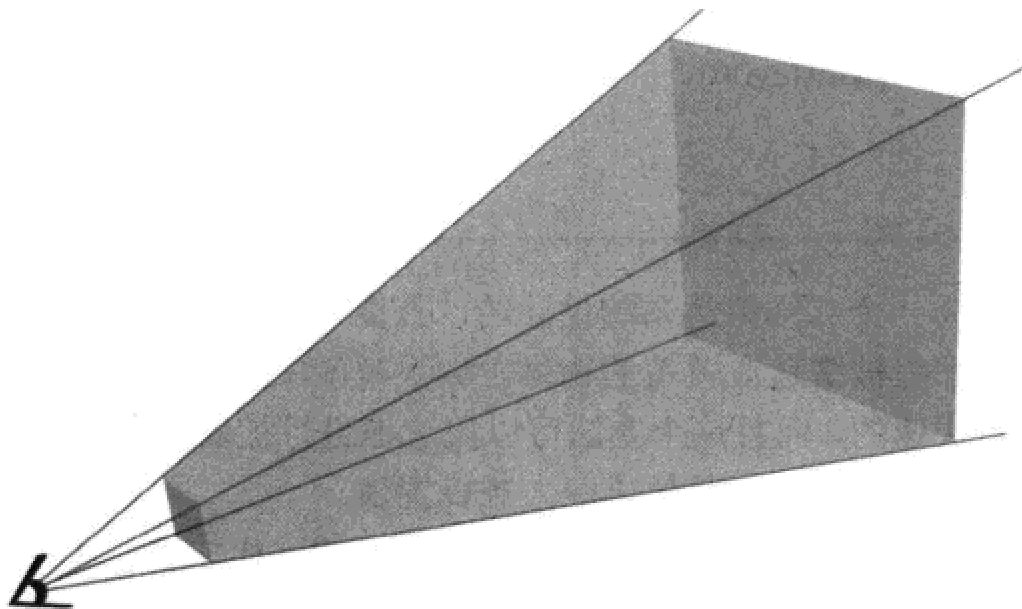


图 1.6.1　游戏世界中包围可视区域的几何对象称为视锥

的关键是，要使用那些可以快速与视锥进行比对测试的几何体，所以最常使用的图元是球体和沿坐标轴的包围盒（axis-aligned bounding box，简称 AABB）。

1.6.2　传统的六面法

视锥本身最常见的表示方法就是 6 个平面，这 6 个平面是从一个矩阵抽取出来的，该矩阵是由一系列的模型视图矩阵和投影矩阵组成，如图 1.6.2 所示。每个平面会将场景空间分割成 2 个部分，6 个平面横纵交叉所围成的几何体，就定义了摄像机的可视区域。

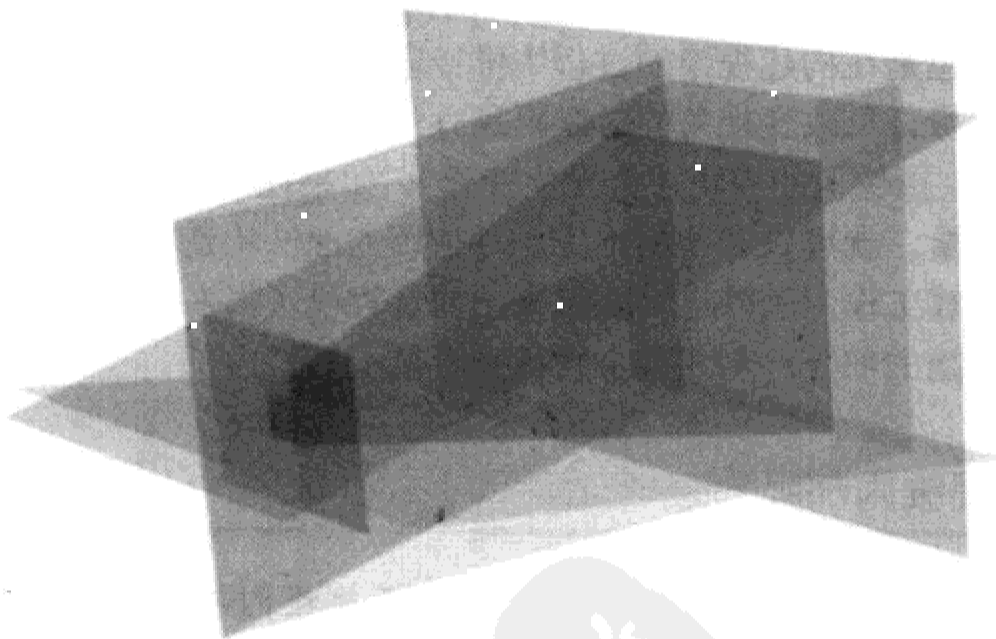


图 1.6.2　定义了一个视锥的 6 个平面

采用六面法有几个不利之处，第 1 个就是系统处理开销。为了判断某个点是否位于视锥之内，我们要对每一个平面求解一个方程式。只有当所有 6 个平面的求解值都为正数，才能确定这个点确实位于视锥之内。此方法的第 2 个问题是，提取视锥的位置和坐标不是一件容易的事情，这使得某些操作（如更新操作）实施起来很困难。这导致每次更新摄像机信息之后，几乎所有的算法实现都会重新创建一个新的视锥。如此一来，在使用 6 个平面重新创建

视锥的过程中，就会带来一些额外的系统开销，其表现形式就是一些不必要的浮点操作（包括浮点除法和求平方根）[Morley00]。

1.6.3 雷达法

这个算法最初是为一个二维雷达系统设计的，其惟一的目的是通过忽略那些未被雷达波覆盖的物体，来显示那些被雷达波覆盖的物体。如果视区中确实有一些边界几何体，那么这个方法被证明是非常快速的，而且这个方法可以快速地进行扩展，以适用于三维系统。现在，视锥剔除算法中也使用了该算法，其效果令人难以置信。

为了能够用一个容易理解的方式向大家解释雷达法的工作原理，还是先回过头来介绍一下雷达法的二维算法，然后再把它扩展到三维世界。

对于一个二维系统，视锥就变成了一个三角形。因为视锥是一个对称的实体，所以这个三角形就是我们所说的等腰三角形，如图1.6.3所示。从顶点到底边的垂直线段，在数学上称为三角形的高。这个线段的方便之处在于，由于视锥是对称的，所以这条垂线将等腰三角形平均分割为2个相等的三角形。同时，这条线段正好与摄像机的前视向量是平行的。

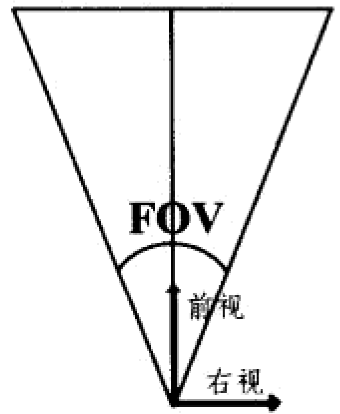


图 1.6.3 用等边三角形表示的二维平截体，三角形的高和向前的方向重合

使用雷达法可以把视锥定义得和摄像机一样直观，这样视锥的创建工作就很容易了。这个二维的雷达只需要知道视角（*field-of-view angle*，简称FOV）、前视向量和右视向量。特别方便的是，这3个参数和摄像机使用的3个参数一模一样，所以我们可以直接从摄像机中提取这些数据。但是，雷达法的实现并不直接使用FOV角度，而是使用这个角度的正切值，这也是在构建视锥的过程中惟一需要计算的东西。即使如此，也可以先进行缓冲保存，然后在摄像机改变视角的时候再进行计算（在大部分游戏中，只有执行变焦操作，改变视图的时候，摄像机的视角才会发生变化）。如果视角保持不变，直接从摄像机中拷贝前视向量和右视向量，就可以创建视锥了。与使用六面法构造视锥的过程相比，雷达法不需要去做那么多的计算工作，所以它实现起来比前者轻松了很多。

现在就进入最有趣的部分：判断某个物体是否位于视锥内部。

1.6.4 这个点在视锥内部吗?

首先要看一下，为了判断某个给定的点是否位于视锥内部，我们都需要些什么，见图1.6.4。为了判断点P是否位于视锥内部，雷达法将向量OP投影到前视向量和右视向量上。如果前视投影向量F在far和near值之间，且右视投影向量R在*rLimit*和–*rLimit*之间，就可以断定点P位于视锥之内，否则就意味着点P完全位于视锥之外。

正如前面所介绍的，首先必须要做的事情是计算出投影向量F和R。将一个向量投影到另一个向量，必须进行点乘积运算，得到一个标量，然后分别用向量F和R乘以这个标量，来获得相应轴向上的投影，如图1.6.5所示。这里有2个向量$\mathbf{V}_1$和$\mathbf{V}_2$，它们的点乘积返回一个标量s。

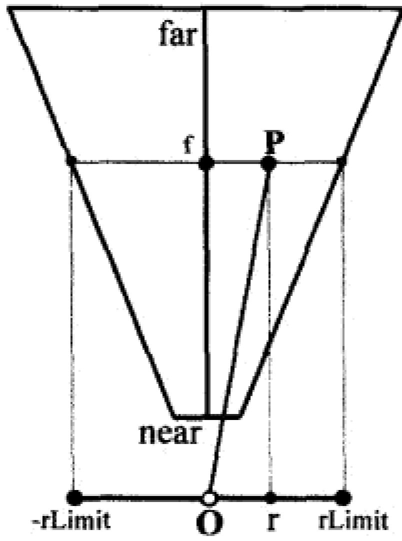


图 1.6.4　判断一个点（P）是否位于视锥内部

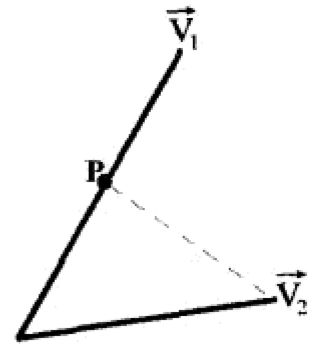


图 1.6.5　一个向量到另一个向量的投影

$$s = \mathbf{V}_1 dot \mathbf{V}_2 = \mathbf{V}_1 \bullet \mathbf{V}_2 = \mathbf{V}_2.x \cdot \mathbf{V}_1.x + \mathbf{V}_2.y \cdot \mathbf{V}_1.y$$

向量 $\mathbf{V}_1$ 在向量 $\mathbf{V}_2$ 上的投影是 $\mathbf{V}_2 * s$，条件是 $\mathbf{V}_2$ 的长度为 1。

使用前面的那个等式，并把前视向量和右视向量看作是单位向量，投影向量 **F** 和 **R** 就可以这样计算出来：

$$f = \mathbf{Forward} \bullet \mathbf{OP} \quad 和 \quad r = \mathbf{Right} \bullet \mathbf{OP}$$

如果条件（$near \leqslant f \leqslant far$）不成立（not true），那么这个点就位于视锥之外。下面是第 1 部分的代码：

```
bool cFrustum::IsPointIn( const cVector2f& Point )
{
    cVector2f OP = Point - EyePosition;
    float f = OP * ForwardVector;           // OP 和 ForwardVector 的点乘积
    if (f < NearZ || FarZ < f) return false;
    float f = OP * RightVector;             // OP 和 RightVector 的点乘积
    float rLimit = fFactor * f;
    if (r < -rLimit || rLimit < r) return false;

    // 到这里，才知道这个点确实位于视锥内部
    return true;
}
```

rLimit 的计算要更为复杂一些，涉及 FOV 角的正切函数。FOV 的正切值被称为 *rFactor*，我们在前面讨论过，它可以这样计算得出（见图 1.6.6）：

$$rFactor = \tan\left(\frac{\text{FOV}}{2}\right) = \frac{\text{opposite side}}{\text{adjacent side}} = \frac{rLimit}{f}$$

$$rFactor = \frac{rLimit}{f}$$

$$rLimit = rFactor \cdot f$$

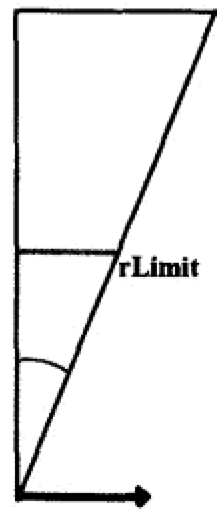


图 1.6.6　计算 *rFactor* 所需要的参量

如果条件（$-rLimit \leqslant r \leqslant rlimit$）不成立，那么该点就位于视锥之外。否则，就可判定该点位于视锥之内。

以上就是判断一个点是否位于二维视锥内部所必须做的全部工作。

为了将上述算法转换为三维视锥算法，只需要简单地引入另外一个系数——*uFactor*，该系数由 ViewAspect 与 rFactor 相乘得到。使用 *ViewAspect* 来定义视图矩阵，视锥就可以和视区吻合。另外，还要考虑一个新的向量：上视向量（*Up* vector）。与前视向量和右视向量一样，*Up* 向量也可以从摄像机中直接提取。给定一个点 *P*、*Up* 向量和 *uFactor*，就可以计算出投影向量 *U* 和 *uLimit* 的值。如果条件（$-uLimit \leqslant u \leqslant ulimit$）不成立，那么点 *P* 就位于视锥之外。

结合前面那个二维视锥的例子，就可以判断出一个点相对于三维视锥的位置。与前面的六面法相比，这个方法只有不到一半的数学计算工作。考虑到每一帧都要针对视锥进行大量的位置判断工作，雷达法确实节省了大量的系统开销。

判断点位置的数学计算是比较简单的，但是我们很少有必要去判断一个单独的点是否位于视锥内部。大多数情况下，我们需要判断的对象是更为复杂的几何体。好在，今天的游戏中，几乎所有边界实体的位置判断都可以沿用点对视锥的算法。因此，为了理解该算法在复杂几何体上的应用，我们必须首先理解点对锥的算法。所以，如果读者现在还有不清楚的地方，最好再回过头去，仔细看看这节的内容。

1.6.5 球体在哪里?

在今天的游戏中，比较常用、且比较快速的边界实体是球体。球体（*sphere*）由中心点（球心）和半径定义。球体的创建很容易，其速度与点对锥算法几乎一样快，因为它是点对锥算法的一个变种。但是，判断一个球体相对于视锥的位置，不再是简单的内部和外部的问题，因为如果边界球体不是完全位于视锥内部，或者虽然完全位于视锥外部，但却和视锥平面相交，上述算法也会返回检测结果，这就使得我们可以做一些聪明的优化，后面会对此进行讲解。不过，这篇文章只会涉及边界球体可见（*Visible*，即完全位于视锥内部）或完全位于视锥之外的情况。更先进的方法请参考[Puig03]。

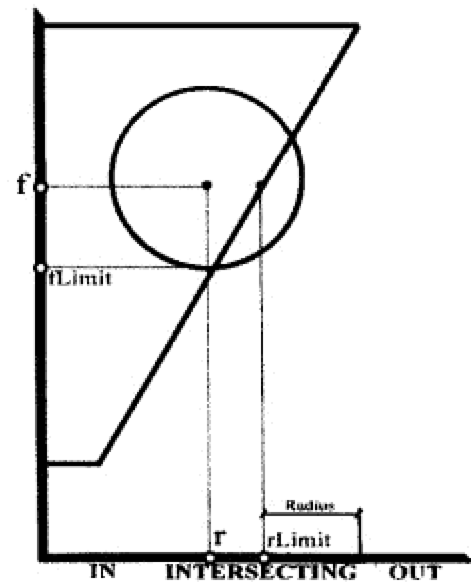


图 1.6.7 如果投影向量位于 Out 区域，那么就认为球体完全位于视锥外部（Completely Out）；否则，球体就是可见的（Visible），因为它位于 In 区域或者 Intersecting 区域

考虑到球体的定义：球心和半径，我们可以把前面对一个点的位置检测看成是对一个半径为 0 的球体的检测。现在要做的就是将上述算法进行扩展，使它可以检测一个半径不为 0 的球体。见图 1.6.7。

该算法首先会检测球体是否位于视锥内部，并根据情况返回 true 或 false。这部分代码与前面对点的检测算法很类似，只是这次使用的限定值是球体的半径。

```
char cFrustum::ClassifySphere(const cVector3f& Center, const float Radius)
{
	cVector3f OP = Center - EyePosition;
	float f = OP * ForwardVector;            // OP 与 ForwardVector 的点乘积
	if  (f < NearZ-Radius || FarZ+Radius < f) return false;

// 未经优化，但更容易理解
```

```
        float r = OP * RightVector;
        float rLimit = rFactor * f;
        if (r < -rLimit-Radius || rLimit+Radius < r) return false;

    // 优化的代码（拿掉了一个减数）
        float u = OP * UpVector;
        float uLimit = uFactor * f + Radius;
        if (u < -uLimit || uLimit < u) return false;

      return ture;
    }
```

读者可以看到，这里把 *r* 轴上的测试进行了彻底的扩展。这个代码还可以进行一些优化，实际上，随书光盘中的程序就重写了这部分代码。请注意，第 2 个 if 语句中的条件（*–rLimit–Radius*）和–（*–rLimit+Radius*）是一样的。有了这个重新分解的算式，在计算 *rLimit* 的值时，该加法操作只需要执行一次，然后使用其结果的正值或负值就可以了。

如果条件检查失败，那么系统就会报告：边界球体完全位于视锥之外。第一眼看上去，会觉得那段根据视锥检索球体位置的代码有些乱，但是一旦仔细地走下去，就会发现这其实是非常基本的代码，更不用说它比六面法快得太多了。

边界球体虽然检测起来很快，但是结果通常会非常不精确，在对象周围还有大量的空白空间无法确定。为此，视锥算法还必须提供一些必要的功能，来判断复杂几何体的位置，如箱体（盒子）和圆柱体等。但由于本文只是雷达法的一个基本介绍，所以就不再详述这些方法了。尽管如此，随书光盘中仍然提供了这些程序的源代码，包括针对沿坐标轴的包围盒和定向包围盒进行检测的函数。我们使用雷达法对这 2 个图元的检测代码进行了比较大的优化。在有些情况下，当对象完全位于视锥外部时，六面法算法就会失败，即仍然报告该物体是可见的。对于这些情况，雷达算法却依然非常可靠。

1.6.6 其他应用

在六面法中，找到视锥的 8 个顶点是一项艰巨的任务，因为要计算 8 次 3 个平面的交叉，这实在是一个“昂贵”的操作。但是，对雷达法而言，这项工作确非常简单。看一下图 1.6.8。

将图 1.6.8 中标记的两个向量相加就可以计算出点 *A* 的位置。而计算向量 *nVector* 只需要用“near”值乘以单位前视向量即可。找到向量 **R** 需要一些技巧，它是右视向量和一个 factor 因子的乘积。factor 因子可用下列方法估算出来：

$$rFactor = \tan\left(\frac{\text{FOV}}{2}\right) = \frac{\text{opposite side}}{\text{adjacent side}} = \frac{factor}{near}$$

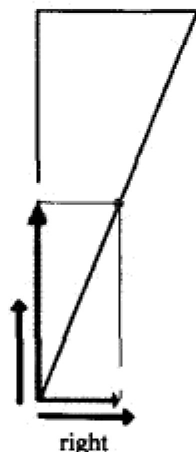


图 1.6.8 向量 NVector 和 Rvector 相加，可计算出点 *A* 的位置

$$rFactor = \frac{factor}{near}$$

$$factor = rFactor \cdot near$$

因此，向量 **R** 被定义为：*rVector* = **Right**·*rFactor*·*near*。*rVector* 加上 *near* 向量就得到点 *A* 的位置，且所有 8 个顶点的位置都可以使用同样的方法得到。读者可以在随书光盘提供的源代码中找到一个完整的程序实现。

知道了 8 个顶点的位置，就可以很方便地做很多事情，如画出摄像机的视锥，将场景中的可视区域用 AABB（沿坐标轴的包围盒）或边界球体围起来，等等。

1.6.7 进一步的改造

之前的这个算法本身已经非常快了，但是还有一些技巧可以帮助我们提高效率和速度。下面就按照效果的明显程序来看看这些技巧，位置越靠前的，其效果越明显。

在文章前面的内容中，读者已经学会了用雷达法来判断对象在视锥中的位置，该函数的返回值只有两个：*completely outside*（完全在视锥之外）或 *visible*（可见）。可以把这个方法进行扩展，来判断几何体是完全在视锥内部、完全在视锥外部，还是与视锥平面相交。这个扩展算法实现起来也不困难，而且因为能够采用下面的这些优化技巧，所以在速度上也不会有大的损失。

1. 层次化场景管理

想象一个有很多实体的大型场景。一种做法是对每个对象进行检测，并且只渲染那些可见的对象；另一种做法是把所有对象都发送给图形硬件，并让图形硬件针对每一个顶点去判断，然后抛弃那些不可见的多边形。很明显，第 1 种做法要快得多。虽然如此，但是如果这样做的话，就要检查场景中的每一个对象，所以这也不是最优的方法。通过使用场景层次，用一连串的边界几何体将场景中的对象围起来，并进行分组，就可以快速地确定哪些分组的对象位于视锥之外并抛弃它们，从而大大减少系统的检测次数。例如，如果一个边界几何体完全位于视锥之外，那么它所包围的所有对象（检测时使用的是凸面边界几何体，组成该凸面几何体的所有边界几何体之内的全部对象都应包含在内）也一定是完全位于视锥之外。报告为完全位于视锥内部的分支（子几何体），也同样适用上述原则。根据定义，边界几何体的所有分支都完全位于其内部，所以只会对这些分支进行遍历，但不会做再次检测。

有一些空间分割算法允许用视锥对场景进行分层遍历，然后直接剔除不可见的分支。其中最简单、应用最广泛的有八叉树、BSP 树、KD 树（k-dimensional tree 的缩写）和 ABT。这些算法都是把游戏场景组织成一个分层的树结构，但是各个算法之间还是有一些微小的差别和特性，这也使它们各自适用于不同的场景。选择使用哪种算法，取决于具体要实现的应用程序。

2. 平面遮蔽

将场景层次化可以让我们以对象组为单位来剔除对象，从而不必对每个单个的对象进行

视锥检测。但是，如果确定某个父对象有可能是可见的（也就是说，该对象并不完全位于视锥之外），那么其所有子对象都必须经过视锥检测。这样的算法没什么问题，速度也相当快，但是在某些情况下，还可以对它进行一些优化。

当检测某个对象相对于视锥的位置时——为简单起见，假设要检测的是一个球体。那就要做 3 种检测：远近检测、右检测和上检测。如果右检测的结果是对象完全位于视区之内，那就不必再去检测它的子对象了，因为检测结果肯定是这些子对象也完全位于视区之内（参见图 1.6.9）。只有当检测结果为交叉时，才必须再去检测它的子对象。因此，在进行视锥检测时，有些操作是可以避免的，因此也就能获得一些额外的性能提升。

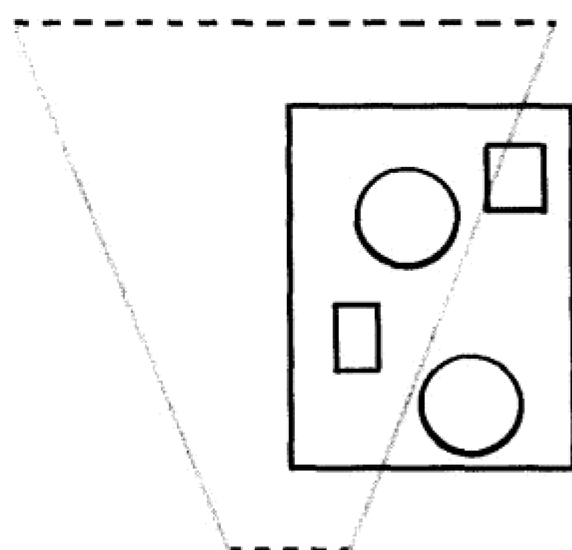


图 1.6.9 视锥中的虚线表示的是远近测试，该测试的返回值是对象及其所有子对象都为 Completely_In

为了实现平面遮蔽的算法，可以在需要进行上述 3 种测试的边界对象中保存一个字节的信息。这样，如果二进制代码为“101”，就会进行第 1 种和第 3 种测试，但不会执行第 2 种测试。这样速度就快多了，因为我们只需要清除或检查这些字节的信息，而不需要其他更复杂的操作。

3. 平面一致性

在大部分应用程序中，特别是在游戏中，摄像机的移动都比较平稳，其产生的视锥变化也比较小。这样，对于上述 3 种测试，如果有一种测试失败，那么在下一帧中，该测试很有可能也会失败。因此，在下一帧中，视锥必须首先从之前失败的那个检测开始检查。这样，在视锥本身变化不大的情况下，很有可能只需经过一次测试，就可以剔除对象了（参见图 1.6.10）。

有很多方法可以实现平面一致性算法。最简单的方法是在边界对象中保存一个字节的信息，该边界对象就是在上一帧中，有某种测试环节失败的对象。当视锥检测对象时，会从该字节信息所指向的那个检测开始，如果第 1 种检测通过了，就会照常继续其余的 2 种检测。

4. 检测对象是否位于视锥的 AABB 之内

检测某个对象是否与一个沿坐标轴的包围盒（AABB）交叉时，几乎总是可以把检测的次数降低到 6 次，即每个轴向上检测 2 次。由于不需要执行特别复杂的操作，所以与通常的视锥检测相比，这个方法快了很多。在本文“其他应用”一节中，介绍了一种确定视锥的 AABB 的方法。该方法是很成熟的，而且最重要的是，如果视锥发生变化，每帧也只需要调用一次。

如果判定某个对象确实位于视锥的 AABB 之外，那么它也一定是位于视锥的外部（参见图 1.6.11）。检测一个对象是否位于 AABB 之外的工作是很快的。我们只需要进行简单的比较，就可以剔除一些对象。但是，如果这个对象并非完全位于 AABB 之外，这时候就要再进行通常的视锥检测了。

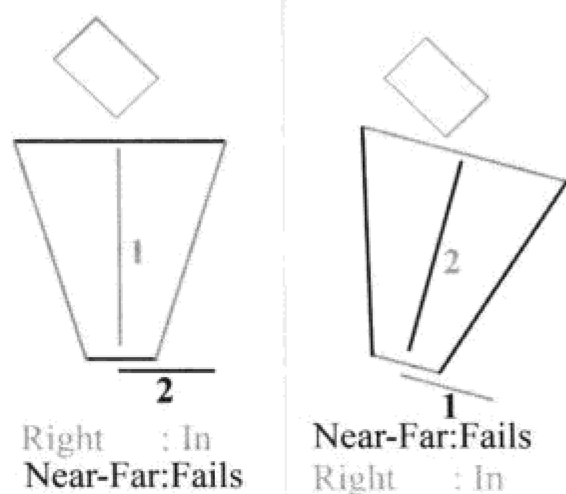


图 1.6.10 根据上一帧的信息，重新安排测试的顺序，这样可能只需一次测试就可以剔除对象

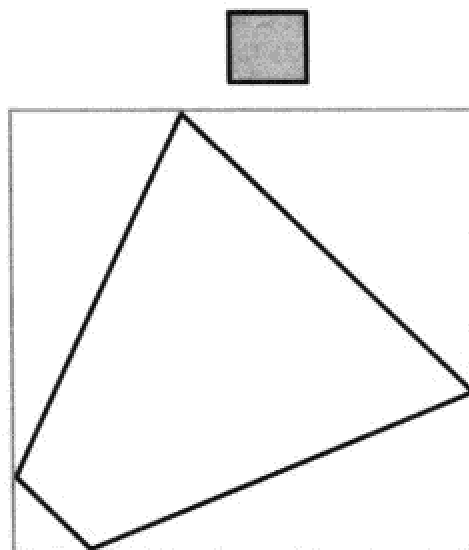


图 1.6.11 如果判定某个对象完全位于视锥的 AABB 之外，那么不需要进一步的检测，就可以判定该对象也完全位于视锥之外

1.6.8 总结

视锥剔除法是最常用的剔除技术之一，而且它不需要太多的处理开销，就可以提供以对象为单位的剔除算法。现在的大多数应用程序和游戏采用六面法来完成剔除工作，就可以得到非常好的效果。但是正如本文所提到的，雷达法是另外一个可以选择的快速算法，该算法对内存的需求比较小，检测速度却非常快，而且也更为直观易懂。最重要的是，用它来创建视锥几乎不需要什么系统开销。

1.6.9 参考文献

[Jelinek&Sykora01] Jelínek, Josef and Daniel Sykora. “Efficient Frustum Culling.” Available online at *http://www.cg.tuwien.ac.at/studentwork/CESCG/CESCG-2002/DSykoraJJelinek/*.

[Morley00] Morley, Mark. “Frustum Culling in OpenGL.” Available online at *http://www.markmorley.com/opengl/frustumculling.html*. December, 2000.

[Puig04] Puig, Frank. “A Fast Frustum Class.” Available online at *http://fpuig.cjb.net/*.

1.7　通用的分页管理系统

Ignacio Incera Cruz
ignacio@incera.net

我们经常会遇到这样的情况：在某个给定的时间，为了加载新的信息，我们需要从系统中卸载掉那些不必要的信息。因为要随时跟踪所有的信息，以便确定哪些信息是需要马上卸载的，哪些信息是需要加载的，所以这个简单的任务却背付着复杂的管理包袱。如果有多个用户需要在同一时间访问同一条信息，那么情况又会变得更加复杂。我们把这个针对当前加载数据的管理过程通常被称为“分页”管理。

从传统上讲，要解决这类问题，就要设计一个复杂的系统，而这些系统并不总是像我们希望的那样富有效率。由于这些系统是针对特殊情形下的分页问题而专门设计的，所以通常限制颇多。本文介绍的是一个更为通用、更为简单，却更为高效的分页问题的解决方案，我们称之为“*generic pager*”（简称 GP）。通过实现通用性、易用性和高效性，该 GP 系统让系统设计人员可以完全忘却创建这样一个定制的系统所面临的挑战，使他们可以把更多宝贵的时间用在游戏其他关键部分的设计和实现上。

为了消除读者心头的疑虑，笔者会通过两个大的项目（来自两个完全不同的领域），来强调该 GP 系统的灵活性和高效率。

- GP 曾被成功地集成到一个自发的机器人系统中，其中机器人的资源是有限的，并且需要在给定的时间对所需的信息进行高效的管理。
- GP 曾被应用于导航系统和可视化 3D 地形系统中。

1.7.1　老式的分页解决方案：一查到底

流传最广的（但不一定是最好的）解决方案包括以下步骤：

1. 定义要加载或卸载的信息。
2. 定义每个信息块的大小。
3. 将搜索空间分割成小的信息块。
4. 定义必要的数据结构，来管理所有的信息块。
5. 每次需要的时候，检查要加载或卸载的信息，查遍搜索空间中的所有信息块。

读者或许已经注意到了，这个解决方案非常没有效率。它把太多的时间花在对搜索空间的分割上。并且为了维护如此复杂的一个数据结构，每个信息块中又有太多的信息，这个解决方案因此也耗费了太多的资源。检

查所有信息块的工作也是代价高昂的，且这个方法只能加载或卸载那些在设计时定义的信息类型。如果需要处理其他类型的信息，可能就要重新设计整个分页系统！最后一点，一旦搜索的空间变大了，所有上述问题都会呈指数级的数量增长。既然如此，那就往前走，花点时间在一开始把它做好吧。

1.7.2 GP 分页解决方案：只检查需要的

通过简单、直观的接口，GP 非常轻松地就解决了前面提到的所有问题。GP 系统的主要功能罗列如下：

- 对设计人员和用户几乎是完全透明的
- 与搜索空间的大小无关（或者说，独立于搜索空间的大小）
- 没有预处理、空间分割，也没有复杂的数据结构
- 内存里只放置必要的数据
- 多用户透明（对设计人员和用户都是如此）
- 与信息类型无关

在下面的内容中，会循序渐进地讨论 GP 系统的设计及其实现的一些细节。有必要在这里说明一下，该实现不是惟一的。读者可以使用大部分的高级编程语言和范式，按自己的要求来实现 GP 系统。

1.7.3 索引是关键

在 GP 系统以及其他传统的分页系统中，必须定义每个要加载或卸载的信息块的大小。还要定义一个机制，为每个信息块定义惟一的标识符，用来定位各个信息块在搜索空间中的位置。

我们首先设计一个名叫 `Gpindex` 的类，让它来管理上述这些信息。为此，就要同时实现 3 个函数：

- 定义分页信息块的大小
- 定位信息块在搜索空间中的位置
- 惟一地标识每一个信息块

`GPindex` 类包含了两个属性：`position`（位置）和 `size`（大小）。每个属性包含 n 个元素，其中 n 是信息块的维数。通常情况下，信息块是二维的，如内存、图片或者数字地形模型。在这些情况下，`GPindex` 中保存的位置信息包含 x 和 y（二维坐标值），而大小信息则包括 `height`（高）和 `width`（宽）。不过，该系统并非只限于一维和二维的信息块，我们可以根据信息块的维数，随意地向每个属性里添加相应数量的元素。

属性 `position` 会定义、定位并标识每一个信息块。由于每个信息块都占据着一部分的搜索空间，所以它的位置也是惟一的。如此一来，我们就可以方便地用信息块的位置来定义其方位和惟一的标识。

`GPindex` 还提供了另外一个非常实用的函数，它使我们不需要定义任何数据结构，也不

需要预先进行空间分割，更不用在内存里加载信息，就可以访问到所有的信息块。该函数实现了 3 个方法：

```
GetIndex (position p);
    // 获取位置为 p（position p）的 GPindex
GetNext(int n)
// 获取任何维度中向后的第 n 个 GPindex
GetPrevious(int n)
// 获取任何维度中向前的第 n 个 GPindex
```

下面是一个例子：

```
GPindex GetIndex (int x, int y);
    // 返回位置是 x, y 的 GPindex
GPindex GetNextX(int n);
    // 返回 X 方向上向后的第 n 个 GPindex
GPindex GetPreviousX(int n);
    // 返回 X 方向上向前的第 n 个 GPindex
GPindex GetNextY(int n);
    // 返回 Y 方向上向后的第 n 个 GPindex
GPindex GetPreviousY(int n);
    //返回 Y 方向上向前的第 n 个 GPindex
```

如图 1.7.1 所示，其中有一个用 position(100, 100)和 size(50, 50)定义的索引。

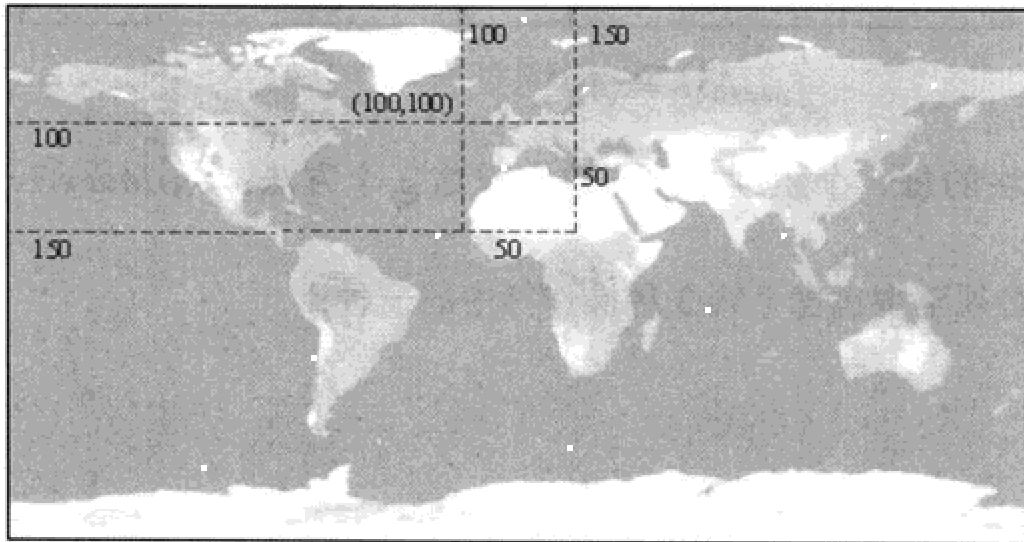


图 1.7.1　位置为 position(100, 100)，大小为 size(50, 50)的 GPindex。感谢 Earth Observatory（地球观测站）：Blue Marble Web 网站（eobglossary.gsfc.nasa.gov）为本文提供图片中所使用的世界地图

这真可谓是一专多能啊，因为一旦定义好了索引，就可以认为搜索空间也被分割好了，虽然我们并没有真正进行如图 1.7.2 所示的分割（从而节省了时间和内存）。

现在可以随便移动，通过在 50×50（size）的块上移动，就可以创建一个，如图 1.7.3 所示的索引。

ON THE CD

一个非常有趣的特性是，position 和 size 可以引用任何类型的数据。随书光盘中的示例代码是使用整数类型来实现的，但是读者可以使用任何一种数据类型或者数据结构，只要考虑好下列两个标准：

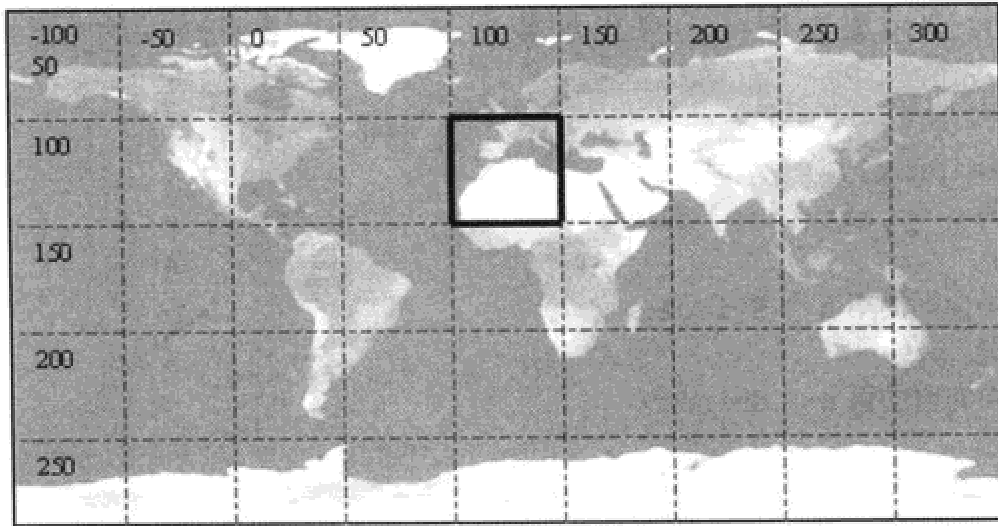


图 1.7.2 隐式分割搜索空间

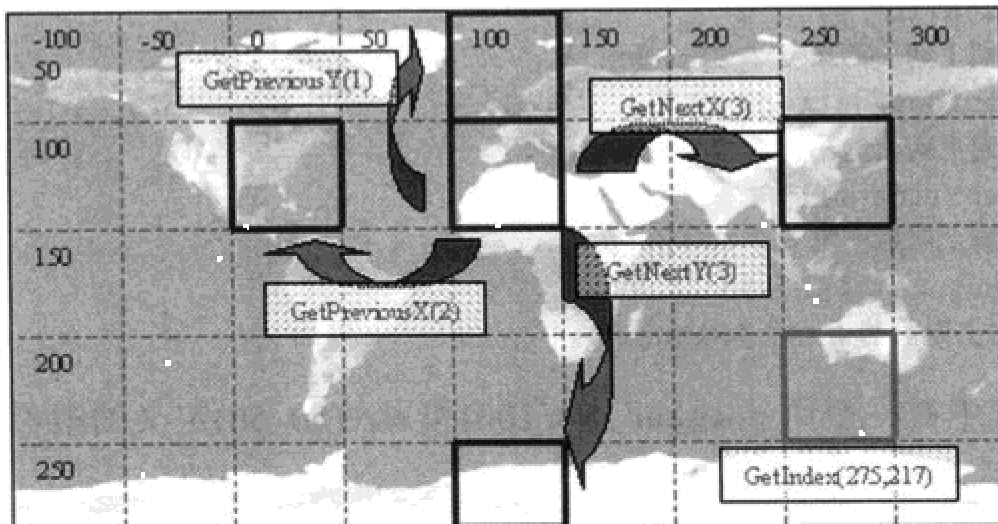


图 1.7.3 在搜索空间里漂移航行

- 针对信息块的每一个维度，相应地实现这 3 个方法：*GetIndex()*、*GetNext()*和 *Get-Previous()*。
- 数据类型必须至少包含下列 3 种操作符中的一种：

```
操作符 <
操作符 = =
操作符 < =
```

尽管如此，在大多数情况下，整数类型还是最常用的（或者至少是一种可以很容易地转换为整数的数据类型），所以随书光盘中的程序实现还是可以满足大部分应用的。

如果某个索引超出了搜索空间的范围，或者有潜在为空的信息块，这些都不是什么问题。解决的办法是检查边界，如果必须加载某个空的信息块，那就释放空间，下一节会对此进行详细解释（参见图 1.7.4）。

回顾前面所讲的内容，有了索引的定义，马上就可以开始分页了：

```
GPindex
(
    Position,  //x, y, …,n 个维度
    Size       //width（宽）, height（高）, …,n 个维度
)
```

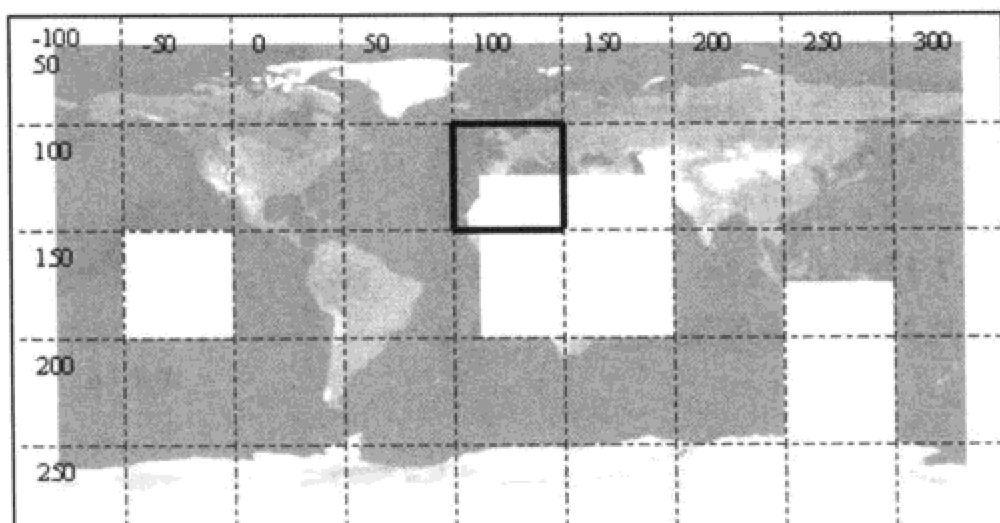


图 1.7.4 越界和释放空间

1.7.4 GPtile：空间中的块

知道了如何分割搜索空间，以及定位并标识一个特定的信息块之后，接下来就要定义信息块本身了。GP 系统中有一个简单的类——`Gptile`，这个类主要包含了 `GPindex` 来定义自己的位置和大小。这就意味着，*tile* 其实就是搜索空间中的块，并且是由索引来确定的。在后面的地形分页系统中，每个 Gptile 都对应一块地形，它的位置和大小是由 tile 的索引来定义的。

一旦通过指定索引定义了 GPtile，剩下的工作就是加载或卸载数据了。

```
virtual void Load()
virtual void Unload()
```

GP 系统的用户只需要设计一个满足下列需求的类：

1. 从 `GPtile` 派生而来
2. 实现 `Load()` 和 `Unload()` 这两个方法

在实现这些方法时，用户可以利用 `GPtile` 索引中的信息，以及从 `GPtile` 派生出来的那个类，完成信息的加载或卸载。

继续前面的那个例子，position(100, 100)和 size(50, 50)定义了一个索引。现在就得到了这个信息块在搜索空间中的位置和大小，但是我们根本不需要知道这个块是什么，或者它包含什么类型的信息。在这个例子中，要管理的信息是地形块。这时候，就要定义一个从 `GPtile` 派生而来的类。参见图 1.7.5。

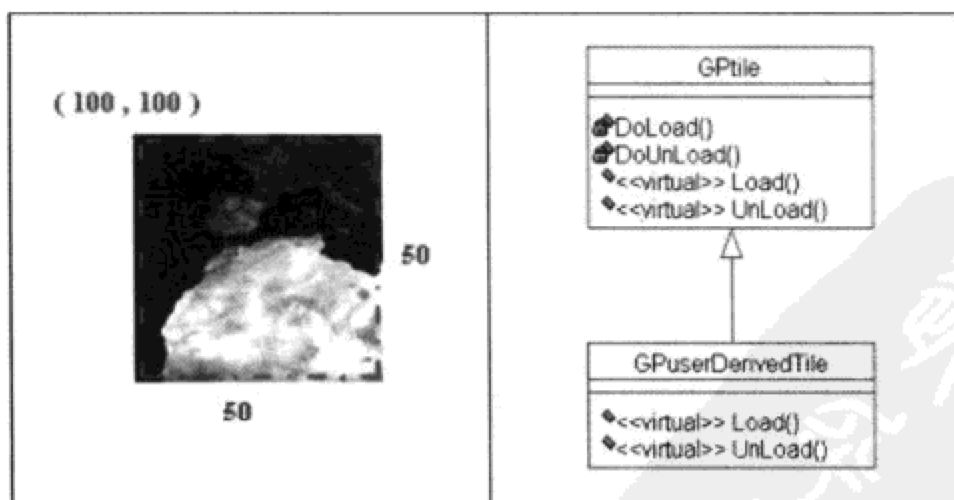


图 1.7.5 `GPtile` 的定义

在 Load 方法中，我们将位于 position(100, 100)、size(50, 50)中的信息加载进来，如图 1.7.6 所示。

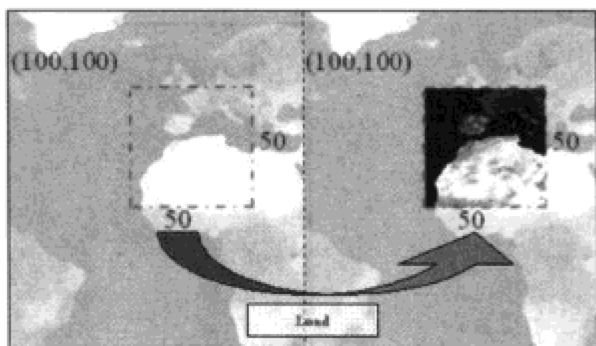


图 1.7.6 GPtile 的加载

知道 GPindex 可以引用任何数据类型是非常重要的，这意味着 GPtile 可以加载或卸载任何种类的信息（参见图 1.7.7）。GPtile 包含的信息可以是一个 50×50 的图片片段、高度贴图、数字地形模型的一部分，等等。

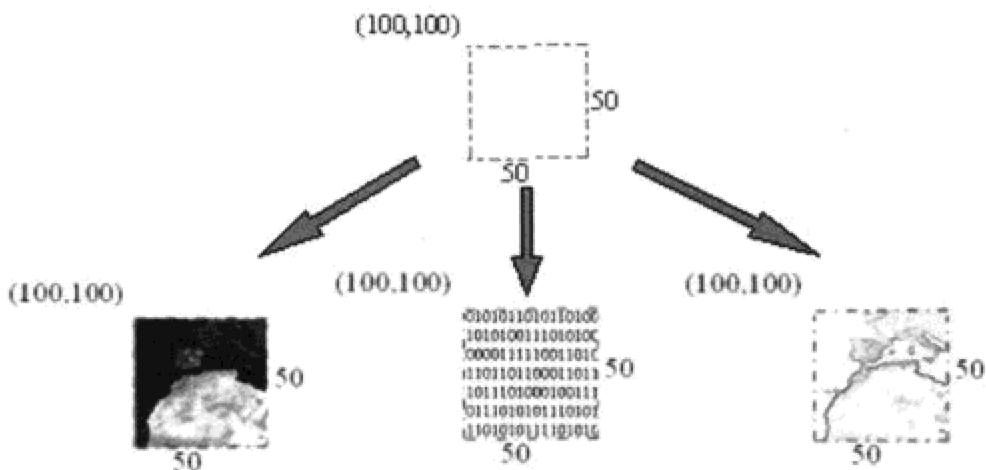


图 1.7.7 同一个 GPindex 中不同类型的信息

每个 GPtile 都含有一个非常有用的属性——State（状态）。每个 GPtile 都有 4 个不同的状态：

LOAD：GPtile 已经加载完毕。

LOADING：GPtile 已经开始加载，但还没有完全加载完毕。

UNLOADED：GPtile 已经卸载完毕。

UNLOADING：GPtile 已经开始卸载，但还没有完全卸载完毕。

GPtile 类以透明于用户的方式来管理这些状态（参见图 1.7.8），用户只需要实现 Load() 和 Unload() 这两个方法。

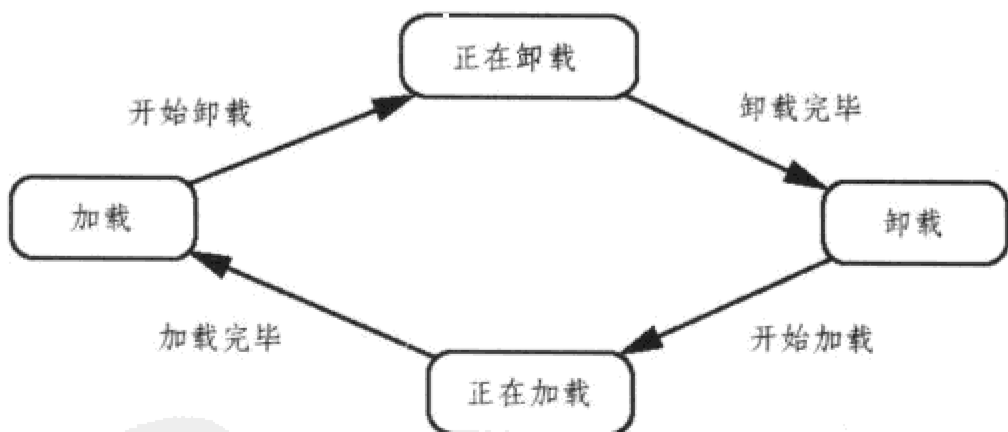
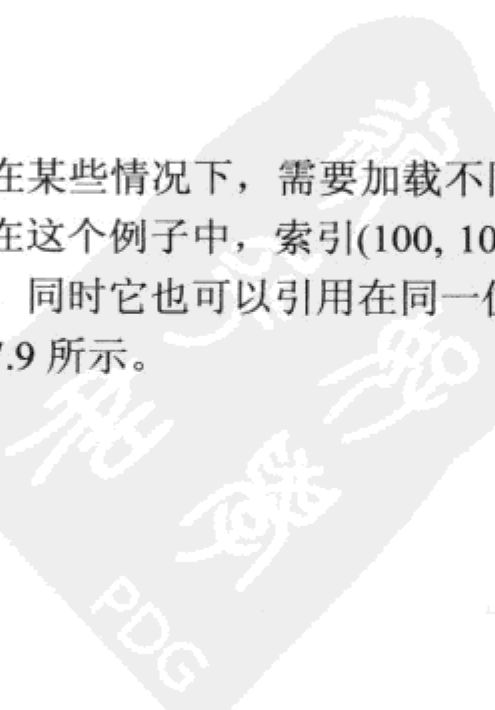


图 1.7.8 状态图表

在某些情况下，需要加载不同类型的信息和原始信息，但在概念上应该是在同一个 tile 中。在这个例子中，索引(100, 100)(50, 50)可以引用一小块位于(100, 100)，大小是(50, 50)的地形；同时它也可以引用在同一位置、同样大小的图片片段，作为那一小块地形的纹理，如图 1.7.9 所示。



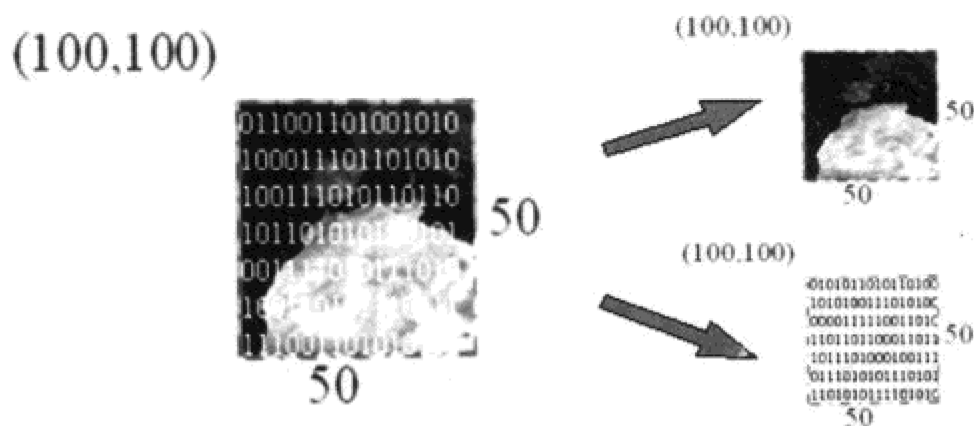


图 1.7.9 不同的信息同时存在于同一个 GPtile 中

为了便于管理，GP 提供了两种选择：

全局概念法：Load() 和 Unload() 方法会分别加载和卸载所有必需的信息。GPtile 的概念包括所有的信息，这些信息由索引来定位，与类型无关。在本例中，加载和卸载了两种类型的信息：一小块地形和一小块纹理，如图 1.7.9 所示。

专用概念法：同时使用 GP 的几个不同的实例，每个实例负责分页管理一种类型的信息，但要同时更新导航系统（窗口详见后文）。

回顾一下本节的内容，现在只需要两步就可以实现 GP 分页系统了：

1. 定义 GPindex
2. 实现 GPtile 的两个方法：Load 和 Unload。

1.7.5 The world：搜索空间

在讨论过索引（GPindex）和 GPtile 之后，需要涉及另外一个非常重要的概念。这个概念在前面的文章中多次提到过，但并没有详细的解释：搜索空间。在 GP 系统中，搜索空间就是一个虚拟的世界，分页信息块（GPtile）就存在于这个世界中，并由索引来定位。

GP 系统通过一个 GPworld 类对该世界进行建模。这个类包含了所有已经创建的，正在使用的 GPtile。当系统需要一个 GPtile 时，GPworld 就会创建一个 GPtile。当不再需要这个 GPtile 时，就会卸载它的信息，且 GPworld 会自动将它销毁。这一过程优化了内存的使用，内存中只严格地保留那些必需的数据。

当 GP 系统需要一个 GPtile 时，它就会通过方法 GetTile() 向 GPworld 发出请求，将搜寻到的 GPtile 的 GPindex 作为参数传递过去。这个时候，GPworld 就会核实自己是否有这个 GPtile。如果有，就将它返回；如果没有，就用 BuildTile 方法来创建。如果 GP 系统要加载或卸载一个 GPtile，它就会通过 GPworld 的方法 LoadTile() 和 UnloadTile() 来完成这项工作。GP 系统的用户只需要设计一个从 GPworld 派生的类，然后再实现虚拟方法 BuildTile() 即可。

在 BuildTile() 方法中，惟一要做的就是创建一个类对象，这个类就是从 GPtile 派生得到的。参数的使用实际上并不是必需的，GP 系统会自动地为这个新创建的 GPtile 赋予相应的 GPindex。用户只需要考虑这个派生类的构造，这其实是件非常容易的事情。可以像下面这段代码一样简单地实现，也可以按照集成了 GP 的那个系统的要求去实现。

```
Tile* BuildTile(const Index& index)
```

```
{
return (new GPuserDerivedTile())
}
```

还有一个方法也可以由 GPworld 的派生类来实现。

```
virtual bool IsValidIndex(const GPRindex& index);
```

这个方法默认的返回值为 true，它的目的是告诉用户，某个特定的 GPindex 在这个 GPworld 中是否合法。GP 系统会自动使用这个方法。当 GP 要为某个给定的 GPindex 创建 GPtile 时，它就会自动检查该 GPindex 的合法性。如果这个索引是合法的，GPworld 就会通过它的方法 BuildTile() 来创建 GPtile（前面已经讲过），如图 1.7.10 所示。如果这个索引不合法，GPworld 什么都不会创建，然后 GetTile() 方法就会返回 false。这项在 GPworld 中进行的 GPindex 合法性的验证以及查错的工作，都是为了避免由此带来的错误加载或其他可能引发的问题而进行的。

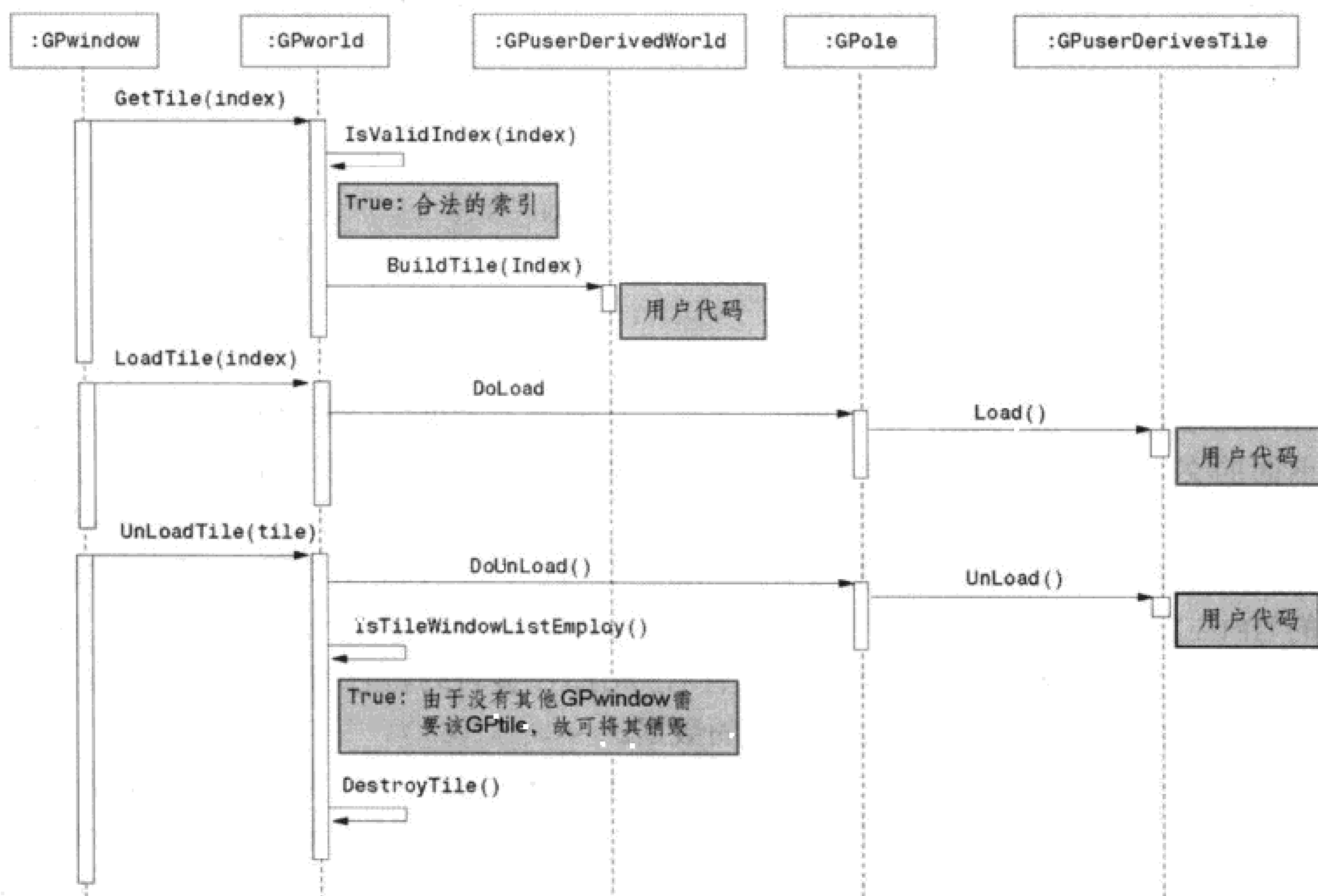


图 1.7.10 时序图

一个搜索空间中可以有很多包含所有必需信息的 GPtile，虽然这些信息是不同类型的（全局概念法）。或者，另一方面，GP 中又可以有很多不同的 Gpworld，每个 GPworld 都只针对一种数据类型，也就是说每个 GPworld 就是 GP 的一个实例（专用概念法）。

只需要 3 个简单的步骤，GP 系统就可以形成了：

1. 定义一个 GPindex。
2. 实现 GPtile 的 Load() 和 Unload() 方法。
3. 实现 GPworld 的 BuildTile() 方法。

1.7.6　窗口：在 GPworld 中航行

一旦拥有了自己的世界（GPworld），只要通过其位置（GPindex）就可以得到它的任意一个信息块（GPtile），我们只需要知道自己想加载或卸载哪些信息块就可以了。为此，GP 系统提供了 GPwindow。有了 GPwindow，不需要知道 GPtile 中包含的是什么信息，我们就可以使用 GPtile 的 GPindex 在 GPworld 中自由航行（参见图 1.7.11）。

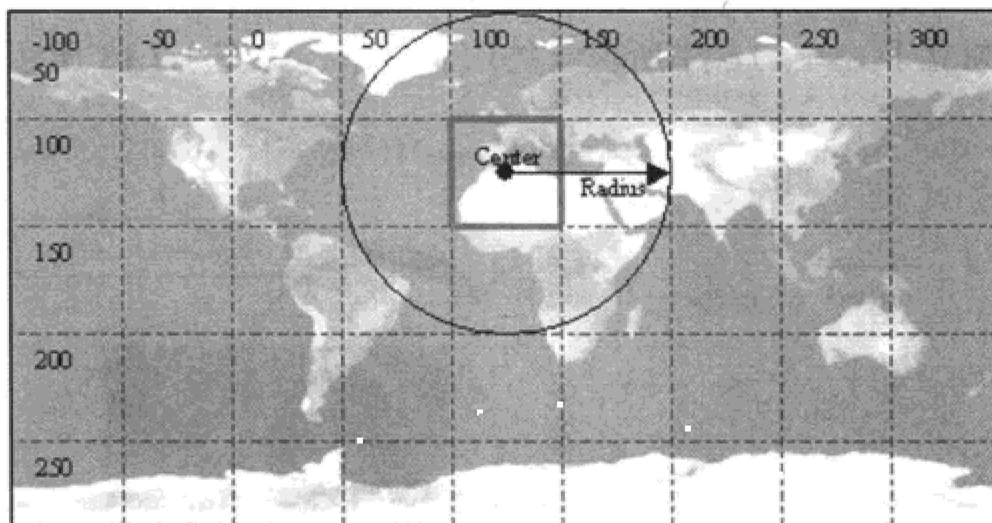


图 1.7.11　GPwindow

GPwindow 有 3 个主要的属性：

CenterIndex：GPtile 的 GPindex 位于 GPwindow 的中央

Radius：表示在中央 GPtile 的周围有多少个 GPtile

GPtilesList：包含着 GPwindow 中所有的 GPtile。

GPwindow 的初始化也非常简单：

1. 为它提供一个 GPworld 的引用；
2. 给 GPwindow 一个 GPindex，用来定位 Gpwindow；
3. 给它一个 Radius（半径）。

此后，GP 系统会自动填写 GPtilesList，将给定的 GPindex 作为中央点，并使用这个给定 GPindex 的两个方法：GetNextIndex() 和 GetPreviousIndex()，告诉 GPworld 加载整个列表，如图 1.7.12 所示。

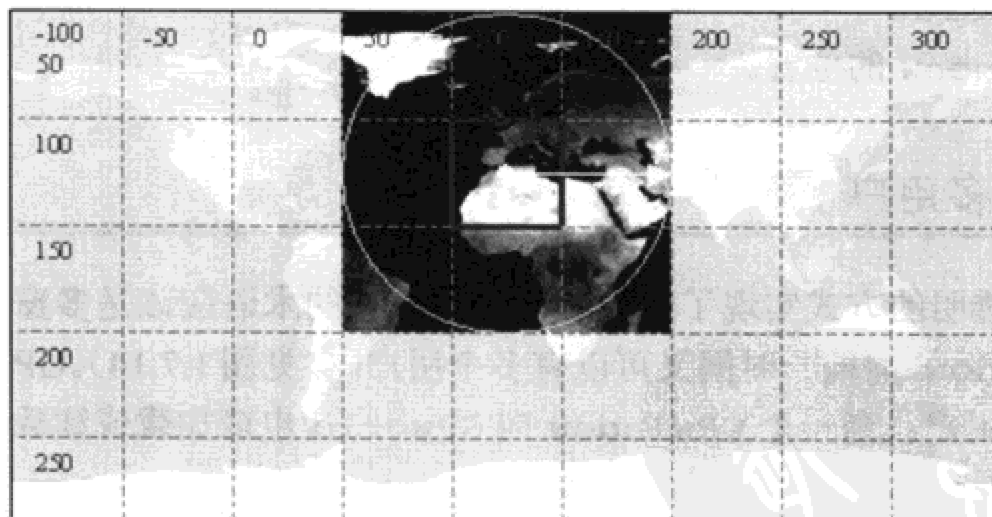


图 1.7.12　加载完毕的 GPwindow

为了能够进行导航，我们只要通过 GPwindow 的更新方法 Update()，将新的位置信息告诉 GPwindow 即可。每次调用这个方法时，它就会检查新的位置是否在中央 GPtile 的内部，也就是说这个新位置是否包含在中央 GPindex 之中。如果新位置在中央 GPtile 之内，GPwindow 就不会更新自己；否则，就会把那个包含了新位置的 GPindex 作为中央点，然后 GPwindow 会根据新的中央点和半径来更新自己。

在更新过程中，系统会修改 GPtilesList 列表，把位于 GPwindow 中的新的 GPtile 添加到列表中，并把不在 GPwindow 中的 GPtile 去掉。GPwindow 会通知 GPworld 去加载新的 GPtile，并卸载那些从列表中拿掉的 GPtile。这个过程是自动进行的，对用户是透明的，而且完全独立于 GPtile 中所包含的信息类型，如图 1.7.13 所示。

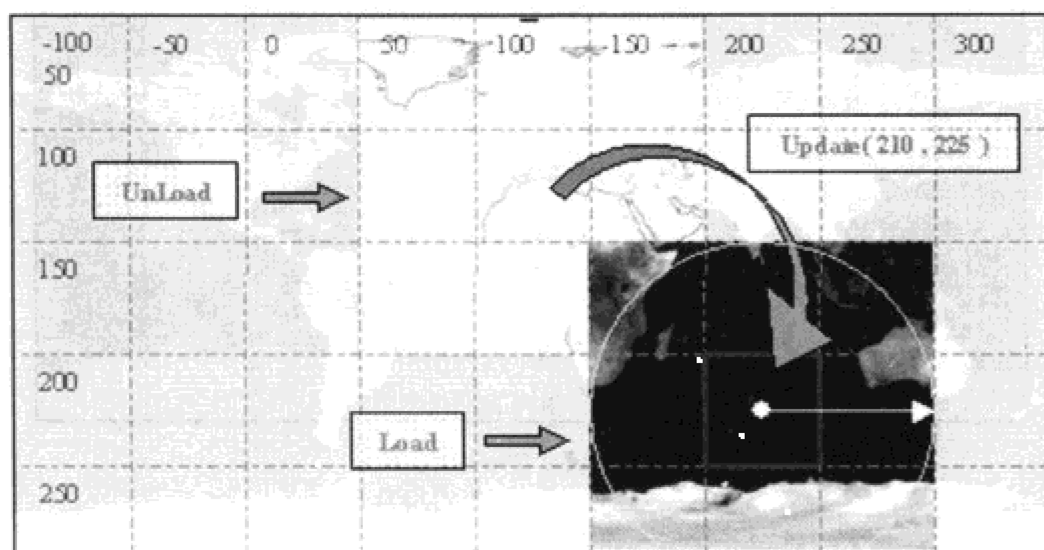


图 1.7.13 GPwindow 的更新过程

如果半径变了，更新过程也是同样的。但是请注意，在这种情况下，中央点是保持不变的。

最后，要准备使用这个 GP 系统了。下面是实现这个 GP 系统的所有步骤：

1. 定义一个 GPindex。
2. 实现 GPtile 的两个方法：Load()和 Unload()。
3. 实现 GPworld 的方法 BuildTile。
4. 给出下面这 3 个值，来初始化 GPwindow：
   - GPworld
   - 一个 Center（中央点）
   - 一个 Radius（半径）
5. 更新 GPwindow 的位置。

1.7.7 多窗口，多用户

GP 用自然、透明的方式实现了多用户特性。准确的术语应该是多窗口，因为一个用户可以有多个 GPwindow，而同一时间又可以有多个用户（参见图 1.7.14）。GP 把每个 GPwindow 看成是一个不同的用户。当一个 GPwindow 向 GPworld 申请加载或卸载一个 GPtile 时，系统就会在这里进行一些简单的检测：

GPtile::Load()：如果 GPtile 已经加载完毕，GPworld 就什么都不做。

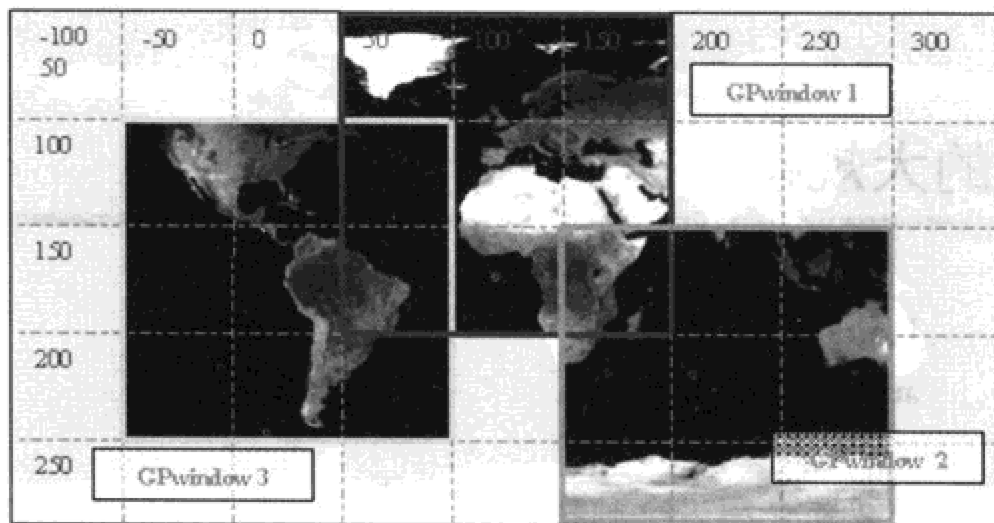


图 1.7.14　多窗口，多用户

GPtile::Unload()：只有当没有 GPwindow 需要某个 GPtile 时，系统才会去卸载这个 GPtile。卸载之后，GPworld 会销毁这个 GPtile。

整个过程对用户是透明的，用户只需在必要的时候，小心地更新本地拥有的一个或多个 GPwindow。

1.7.8　优化：多线程分页

对 GP 系统的一个优化是避免 GPworld 马上执行对 GPtile 的 Load() 方法和 Unload() 方法的调用。GPworld 会在独立的线程中执行这些任务，以避免加载或卸载过程与系统其他部分的互相干扰，进而提高性能。

为此，GPworld 有一个 GPtile 的列表。当某个 GPwindow 向 GPworld 申请加载（或者卸载）一个 GPtile 时，GPworld 就会把这个 GPtile 添加到列表中，并将之标识为“可加载”（或者“可卸载”）。在加载或卸载线程的主循环中，这个列表中的每个元素（GPtile）都会被移出，并进行相应的处理。

1.7.9　总结

本文描述了一个完整的、多用户的、通用的分页系统（GP）。该系统可以用简单、高效的方式，来管理任何类型的信息，优化系统资源的使用。GP 系统的作用域是全局性的，所以任何一个需要进行信息的加载和卸载管理的系统都可以把它集成进去。

该 GP 系统的设计遵循了[Lakos96]和[Alexandrescu01]的一些设计建议，而 GP 系统的实现则采用了[Meyers96]的一些建议。

1.7.10　参考文献

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

[Lakos96] Lakos, John. *Large-Scale C++ Software Design*. Addison Wesley, 1996.

[Meyers96] Meyers, Scott. *More Effective C++*. Addison Wesley, 1996.

1.8 基于栈的大规模状态机

James Boer
author@boarslair.com

本文要介绍一个独特的合成方法，将传统的状态对象与基于栈的管理和队列系统结合起来。这种合成的系统被称为基于栈的状态机。这个系统的机制要明显优于传统的状态机，特别是在应对处理那些习惯上很混乱的逻辑流时，例如嵌套很深的用户界面屏幕菜单。基于栈的状态机还可以轻松地处理那些与状态相关的棘手问题，如怎样去实现一个全局状态（如游戏暂停的状态），而同时不用到处编写特殊情况代码，或者在某个状态中内置固有的信息，告诉它在终止执行后应该返回到哪里。

本文还演示了一个先进的状态管理器系统，该系统中添加了一些特殊的功能，如队列式的状态命令、延迟状态切换和集中化的状态计时功能等。正如大家所期望的，状态还是用层次化的C++对象来表示。这里设计的状态类，可以自动处理一些细节信息（如在进入或退出某个状态时的前一个状态和后一个状态的信息），以及堆叠事件（在其他状态之上的状态进栈和出栈）。该系统提供了一种有效的方法，来管理大规模的状态系统，如程序中的全局应用程序状态（或游戏状态）的概念。

1.8.1 传统状态机编码及相关问题

游戏最基本的任务之一就是提出一个内部“状态”，来表示那些当前显示给玩家的视觉和逻辑元素。一般来说，游戏分成两个不同区间的状态：前端用户界面状态和游戏内部状态。例如，在前端显示的每个用户界面都应该看成是惟一的、独特的状态，因为每个屏幕显示都有其独特的功能，而且在向其他状态切换时，也有其独特的方式和方向。对话框以及其他用户必须与之交互的主要屏幕元素，也可以看成是独特的状态，虽然这些元素有其各自的特殊之处，后面会对此进行讲解。

同样地，在实际的游戏运行中，一个游戏可以有一个或任意多个独特的全局状态。例如，一款传统的第一人称射击游戏（FPS）可以让玩家与计算机对战。这时候，游戏就进入另外一个完全不同的状态：渲染路径改变了，而且游戏的界面也不一样了。从本质上讲，这就好像玩家在玩的是一个完全不同的 mini 游戏，而不是最初的那套控制机制。图 1.8.1 用一个标准的状态机图表表示了上述这些典型的状态。

PDG

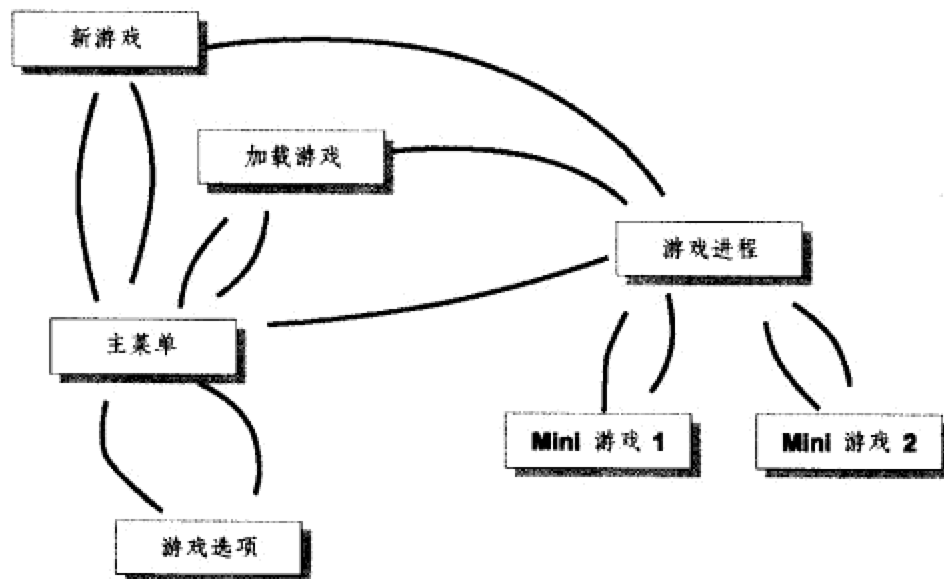


图 1.8.1 一个典型的游戏状态图表

用 C 或 C++代码创建一个简单的状态机是相当容易的。所需要的就是一个枚举型变量和一个开关（switch）语句声明，如程序清单 1.8.1 所示。

程序清单 1.8.1 一个简单的状态机

```
enum GameState
{
    STATE_OPENING_TITLE,
    STATE_MAIN_MENU,
    STATE_RUN_GAME
};

GameState m_State;

//…

switch(m_State)
{
    STATE_OPENING_TITLE:
    //打开游戏主题代码
    break;

    STATE_MAIN_MENU:
    //打开主菜单代码
    break;

    STATE_RUN_GAME:
    //运行主游戏代码
    break;
};
```

不幸的是，现实世界绝非如此简单。马上就会有一些棘手的问题。例如，在切换状态时，如果需要得到第一次更新的通知，这会发生什么事情呢？一个快速而简单的解决办法是：添加一个计数器，记录更新次数，并在状态发生变化时重新归位。那么离开一个状态时又会怎样呢？可以为进入状态、离开状态或更新状态这些情况单独添加开关语句。如果想在以后某个

PDG

特定的时间切换到某个状态，又该如何实现呢？同样，也可以引入一个带有延时量的计时器。如果想对一系列连续的状态变化进行队列式管理，又该如何呢？到这里，那个原本简单的 C 风格的状态机已经变得庞大而混乱，有诸多的变量、开关语句和函数纠缠在一起。虽然在这个时候，C 风格的状态机可能是非常有用的，但是它的可读性、可用性和维护工作，都变成了很严重的问题。更糟糕的是，这些状态各自特定的数据该保存在哪里呢？因为有的只是函数，所以在某个特定状态的生命期内，创建和使用该状态特有的数据就会使情况变得更加混乱。就算没有上述这些问题，该状态机也只适用于单一系列的状态，而且除了剪切和粘贴，这个代码是不可重用的。

那么，该如何处理一些更基本的问题呢？又该如何准确地表示诸如“游戏暂停”之类的状态呢？该状态的行为不同于其他的状态，因为它要能够返回游戏暂停之前所退出的那个状态。同样地，很多菜单系统都采用层次化风格的操作模式，将菜单一层一层地添加到菜单系统中。使用传统的状态机，在不同的用户界面屏幕之间切换（如某个确认对话框弹出位于整个用户界面屏幕的最上层）当然是可以的，但这绝对不是最优的方案。也许用户正在编写下列形式的代码：如果一个对象的类型是 T1，那就执行 A 计划；如果对象的类型是 T2，那就执行 B 计划。如果是这样，那么著名的 C++权威和 C++技术作家 Scott Meyers 先生就会给出他的观点：就此打住，因为这根本不是 C++的风格[Meyers98]。

本文的目的并不是为了颂扬 C++的优点，而是希望向大家说明，为什么面向对象的解决方案通常会大大地优于其他同功能的解决方案。因此，本文会向大家展示利用面向对象的解决方案来解决游戏状态的难题，是如何简化编程工作的难度，并让用户的代码更加简洁易懂的。

1.8.2 用 C++方法解决游戏状态难题

虽然大部分程序员都非常习惯于根据物理实体来创建类的概念（也就是说一个 Weapon 或 Player 类），但是 C++的类也可以非常有效地用来建模更抽象的概念，如状态[Gamma94]。如果把特定函数的含义标准化，可以得到一个直接的方法，来用类给一个状态建模。程序清单 1.8.2 展示了它的大致模样。

程序清单 1.8.2 用类给一个状态建模

```
class SomeState
{
public:
    void OnEnter();
    void Update();
    void OnExit();
};
```

其中，每个函数代表着处理这个状态时的一个特定事件。当游戏要求激活这个状态时，系统就会调用该类中的函数 OnEnter()。这样，该状态就有机会进行初始化、分配内存，或者去激活该状态正常运转所必需的其他物件。在游戏的每个更新点上（更新或渲染循环），系统就会调用这个类的函数 Update()，来处理任何需要处理的事件。根据不同游戏引擎的特殊设计，或许需要一个单独的函数调用来完成渲染工作，也可能根本没有这样的需求。最后，

当系统退出这个状态时，就意味着有另外一个状态要被激活，系统会调用 OnExit() 函数，让该状态完成必要的善后清理工作。

这个类为某个游戏状态所需要的初始化、更新和清理功能的实现，提供了一个理想的场所。而且，几乎同样重要的是，我们现在有了一个逻辑地点来保存上述功能必然要用到的永久数据。举个例子，如果这个状态表示的是一个用户界面的屏幕显示，那么这个类就可以包含屏幕上必须显示和处理的各种各样的用户界面元素。

1.8.3　状态接口类

为了确保每个状态都能遵循这个接口，并且能够通过一个共同的系统来操作不同的状态，可以使用一个基础状态，把它作为标准的接口，由它来派生出所有其他的状态。除此之外，我们假设还要往这个系统添加一些额外的功能：通过这些函数，将状态的名字传递给对应的状态对象。程序清单 1.8.3 显示的就是这个状态接口类的定义。

程序清单 1.8.3　状态接口类

```
class IBaseState
{
public:
    virtual ~IBaseState() {}

    virtual void OnEnter(const char* szPrevious) = 0;
    virtual void Update() = 0;
    virtual void OnExit(const char* szNext) = 0;

    virtual void OnSuspend(const char* szNext) = 0;
    virtual void OnResume(const char* szPrevious) = 0;
};
```

读者可以看到，当系统进行状态切换时，任何一个状态对象都可以很容易地访问到前一个或后一个状态。当要处理那些状态切换专用代码时，这个特性是非常有用的。读者也许已经注意到了这两个函数：OnSuspend() 和 OnResume()，它们是做什么用的呢？我们会在下一节里解释它们的重要性。实际上，它们的部分功能设计是为了降低状态切换专用代码的必要性，这种必要性会使状态机的设计和维护工作复杂化。

1.8.4　状态的堆叠管理：为什么三维比二维好用

对于一个典型的游戏，它可能会有很多不同的游戏模式，这是很常见的事情，而这些不同的游戏模式就是用不同的状态对象来表示的。例如，根据玩家的具体行为，一个典型的角色扮演游戏可能会有十几个不同的游戏状态。玩家是否在屋子外面闲逛，还是在城镇里？他是在商店里买东西，还是在战斗？或者他正在玩里面的 mini 游戏？在游戏进程中的任意一点上，我们都希望玩家可以随时调出同一个选项菜单，然后再返回到他调出菜单之前的游戏状态。让游戏在后台继续渲染也是可取的，但要是在游戏的暂停状态中。在这种情况下，完全

退出一个游戏状态，然后再返回这个状态，就会出问题了。

同样的问题也经常发生在用户界面的屏幕显示中。一般情况下，当某个对话框在屏幕的最上层弹出时，在它下面的主屏幕仍然要继续渲染和显示。这种情况是无法用传统的状态机来解决的，因为我们并不是为了要进入一个新状态，而完全退出当前的状态。我们真正想要的结果，是将某个状态挂起（或暂停），同时优先执行另外一个状态的功能，然后再返回到原来被挂起的状态，继续执行（或取消暂停）它的行为。

用状态栈的概念就能轻松地解决这类问题：在同时运行的多个状态之上压入或弹出一个或几个状态。这也就是前面提到的两个函数 OnSuspend() 和 OnResume() 的功能所在。当一个状态压入到另外一个状态（状态 A）之上时，系统就会调用原先那个状态（状态 A）的 OnSuspend() 函数。但是，仍然会在每一帧中调用 Update() 函数。图 1.8.2 展示了如何通过一个状态栈的机制，让状态机以三维而不是二维的形式来运转。

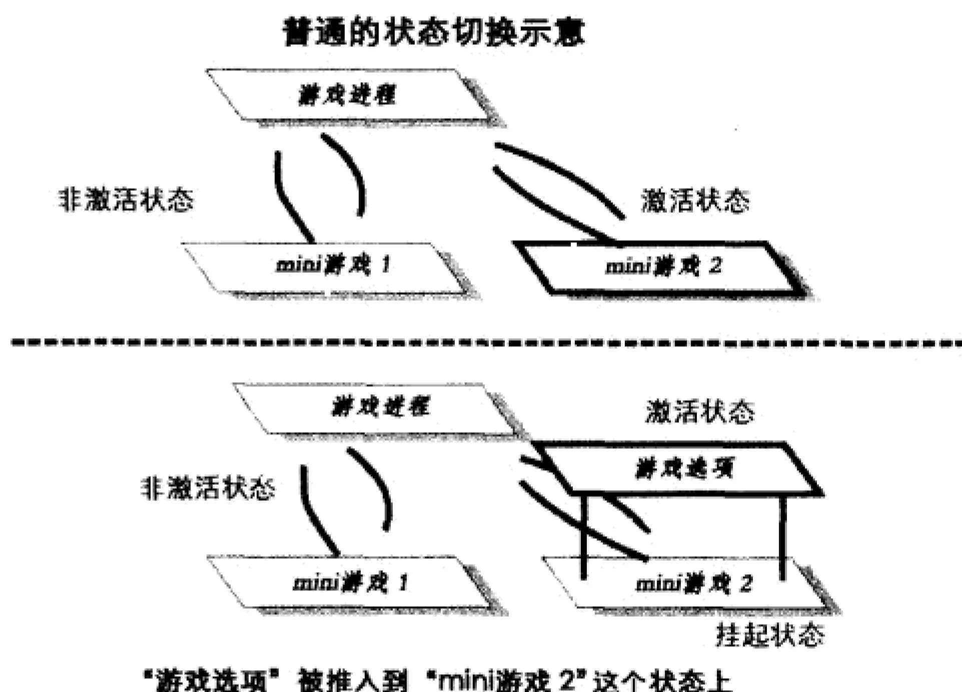


图 1.8.2 在一个典型的游戏状态图表中添加第 3 个维度

这样一来，系统就给了状态两个选择。第 1 个选择，是当另外一个状态被推入到该状态之上时，该状态可以让自己暂停。这是一个非常简单的事情，只要在函数 OnSuspend() 被调用时设置一个 m_bPaused 标记，并在函数 OnResume() 结束时再将其清零即可。如果有另外一个状态正在该状态之上运行，在该状态的更新循环中做一个简单的检测就可以阻止该状态代码的执行。另外一个选择是，该状态对象可以继续在后台更新，让两个状态高效地并行执行。大家甚至可以把这个功能内置到基类中，以便对所有的类执行一致的操作。为了简单起见，本文的例子并没有把这些功能添加到基类的接口中。

1.8.5 状态对象管理系统

读者也许注意到了，我们一直强调的都是“游戏状态”，而不是发生在游戏中的其他类型的状态，如一个 AI 实体中的各种状态，或者用户接口程序中的状态。这是因为，面向对象的状态机更适合那些较为复杂的状态系统，诸如那些表示整个游戏状态的状态。这有两个理由。首先，每个状态必须由一个完整的类来表示。

而通常由于多态性的原因，这个类都是从一个基类派生出来的（或者是用模板来实现的）。对单独一个状态而言，这种工作量已经算是不小的投资了，只有这个状态本身算得上复杂时，这项投资才有价值。第 2 个原因是，这些状态必须全部由一个外部系统来管理，以便可以高效率地使用。下面就来看一下，这样的一个状态管理系统都应该包含哪些功能。程序清单 1.8.4 显示的是 StateManager 类的接口，读者可以在随书光盘中找到这部分代码。

程序清单 1.8.4 StateManager 类的接口

```
class StateManager
{
public:
    StateManager();
    ~StateManager();

    void Init();
    void Term();

    // 注册一个状态对象，并将它与一个字符串标识符关联起来
    bool RegisterState(const char* szStateName,
        IBaseState* pState);

    // 检测当前这个状态是否会在下一个更新循环中发生变化
    bool IsStateChangePending() const;

    // 返回当前的状态
    const char* GetState() const;

    // 根据字符串 ID，来获得状态对象
    IBaseState* GetStateClass(const char* szState);

    // 获取当前状态栈中最上面的一个状态
    IBaseState* GetCurrentStateClass();

    // 返回状态栈的大小
    int GetStateStackSize() const;

    // 如果 bFlush = true，则会覆写所有也许正处于悬而未决的状态。
    // 否则，就把状态命令进行排队，然后根据调用的顺序来执行这些状态命令。

    // 在下一个更新循环中，改变当前的状态
    void ChangeState(const char* szState,
        float fDelay = 0.0f, bool bFlush = false);

    // 在下一个更新循环中，在当前状态之上压入另外一个新状态
    void PushState(const char* szState,
        float fDelay = 0.0f, bool bFlush = false);

    // 弹出一个或几个当前的状态，恢复在它们下面被保存的状态。
```

```
    // 不能弹出最后一个状态
    void PopState(int iStatesToPop = 1,
        float fDelay = 0.0f, bool bFlush = false);

    // 弹出所有的状态，最后一个状态除外。
    void PopAllStates(float fDelay = 0.0f,
        bool bFlush = false);
    //----------------------

    // 更新状态机的内部机制。
    // 只有主更新循环才需要调用一次这个函数，其他循环不能调用该函数。
    void Update(float dt);
};
```

ON THE CD

为了节省版面，这里并没有向大家展示这个类的内部运做细节（私有数据或函数的具体代码），但是读者可以浏览随书光盘中的源程序代码，它们保存在StateManger.h 和 StateManager.cpp 这 2 个文件中。

正如该接口类所展示的那样，状态管理器会使用简单的字符串标识符让状态关联起来。之所以选择字符串标识符，有两个原因。首先，为了调试工作的方便，在调试信息中用字符串来表示状态是非常直观、方便的。第 2 个原因是，从使用效率来看，我们没有理由去选用枚举类型的标识符。我们假设状态的切换相对来说不会太频繁，特别是在使用这个系统来专门跟踪游戏状态的时候。另外，用字符串来表示状态还可以非常容易地与脚本系统进行整合。

该状态管理器可以自动、轻松地处理各种问题，如在队列中的状态和（或）被延迟的状态，并且还可以让各个状态在另外一个状态之上彼此压入或弹出。这就为处理所有基于状态的切换和状况提供了一个可靠、一致的机制。如果游戏或引擎还需要一些其他的附加功能，其添加工作也是非常容易的。而且，如果把该系统作为一个库（或引掣）组件，而不仅仅是一个游戏组件，那么其他项目也会从中获益。

将统一的游戏状态切换和定义机制标准化所带来好处之一，就是各种不同的库元素可以表示完整的游戏状态，而不仅仅是简单的功能组件。例如，有一个库组件代表的是一个屏幕软键盘（这是很多游戏机游戏必需的一个组件），那么它不但要包含这个软键盘的所有功能，还要包含完整的状态处理代码，这些代码是初始化和处理在这种复杂的屏幕显示中发生的所有事件所必需的。随着游戏产品变得越来越复杂，为了缩短开发时间，避免为每一个新的游戏重新开发同样的技术，开发人员就需要开动脑筋，找出能够有效地反复重用更大型、更复杂的组件的方法。

1.8.6 总结

用对象来处理状态，这个简单的概念没什么新鲜的。即使是整合了状态栈技术，并通过一个集中式的状态管理系统加以实现，也没什么可炫耀的。可惜的是，在游戏编程领域，为了支持所谓的“进化式”设计，人们常常远离这些最基本的系统。这就是说，在编码开始之前，根本没人会去考虑这些系统。说得严重点，在游戏开发过程中，这会导致噩梦般复杂的、纠缠不清的编码工作。

事实上，与如何在项目中管理大规模的状态相比，状态管理器内部的工作细节并不重要。无论决定采用什么样的方法和代码来管理这类与状态相关的问题，只要用户的系统有能力处理各种基本的问题，可以有效地管理那些大规模、复杂的软件系统（如游戏）中的各种状态，从长远来看，就可以很容易地节约开发时间，避免生成那些充满了 Bug、难以维护的程序代码。

1.8.7 参考文献

[Gamma94] Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*, 305–313. Addison Wesley, 1994.

[Meyers98] Meyers, Scott. *Effective C++*, Second Edition, 176–177. Addison Wesley, 1998.

1.9 使用 BSP 树构造 CSG 几何体

Octavian Marius Chincisan
mariuss@rogers.com

构造性实体几何法（Constructive Solid Geometry，简称 CSG）是一种使用最基本的图形（如立方体、球体、圆柱体和圆锥体）来构造较为复杂的几何体的方法，这些复杂几何体被普遍地应用于各种应用程序的游戏引擎中。这些基本的几何图形通常被称为图元。CSG 中有几个用来将图元组合到一起的操作，它们是并集、交集和差集。

在本文中，读者会发现，CSG 法和空间二叉树法（Binary Space Partitioning tree，简称 BSP 树）实际上都相当简单，只不过是依据平面来划分和分割多边形而已（以及一点细致的常规内务操作）。除此之外，本文还会带领大家一步一步地认识图元之间的布尔运算（如前面列出的那几种运算），并在最后将这些概念提升到应用的高度，去构造几个复杂的几何体。

1.9.1 CSG 的布尔运算

虽然经典的 CSG 法不是本文的主题，但是为了文章的完整性，还是有必要简单介绍一下 CSG 法的几个基本布尔运算。

在 CSG 法中，两个实体之间的基本运算如图 1.9.1 所示，其中(b)是*并集运算*，(c)是*交集运算*，(d)和(e)是*差集运算*。为了更好地说明这些算法，本例中采用的实体都是简单的二维片段，其中每个线段都表示一个面（face）。所有实体的面都是朝外的。因此，图中的 X 和 Y 就被认为是实体（*solid*）。外空间则被认为是空的。

1. 并集运算

两个或更多实体之间的*并集运算*就是去掉实体空间中重合的面，将两个或两个以上的实体合并成一个。在图 1.9.2 中，就要去掉 C'''C'、C'E 和 EC''这 3 个面。在本文后面的内容中，所有的线段（*segment*）都称为“*面*”（*face*），且所有由线段围成的平面空间都被称为“*平面*”（*plane*）。为了执行并集操作，实体 X 的所有面都要被实体 Y 裁剪，而实体 Y 的所有面也要被实体 X 裁剪。裁剪剩下的几何体就是并集运算最后的结果实体，如图 1.9.3 所示。

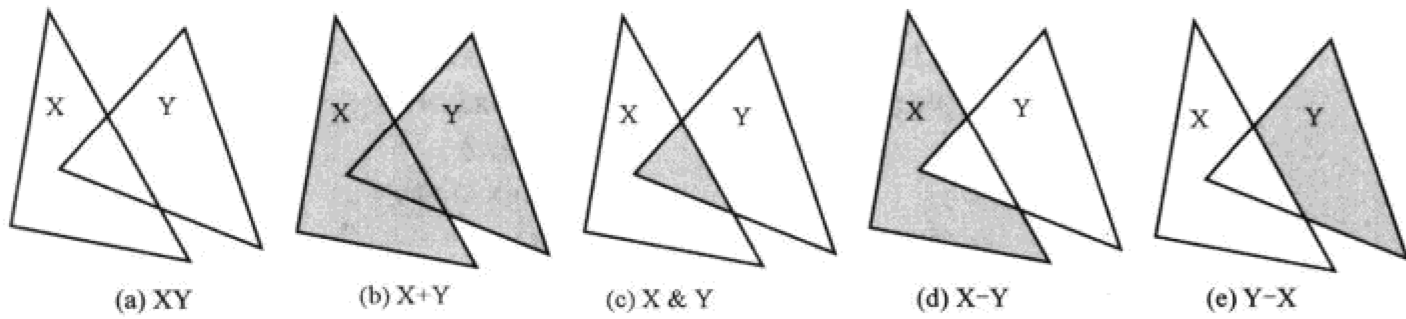


图 1.9.1　实体 X 和实体 Y 之间的布尔运算

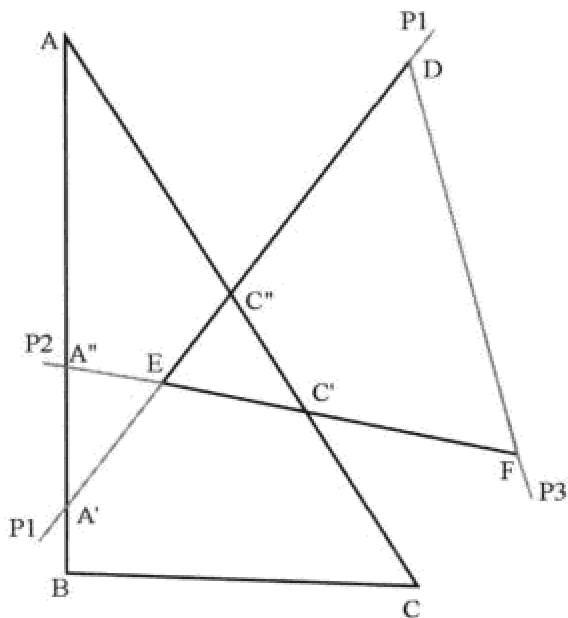


图 1.9.2　用平面 DEF 裁剪平面 ABC

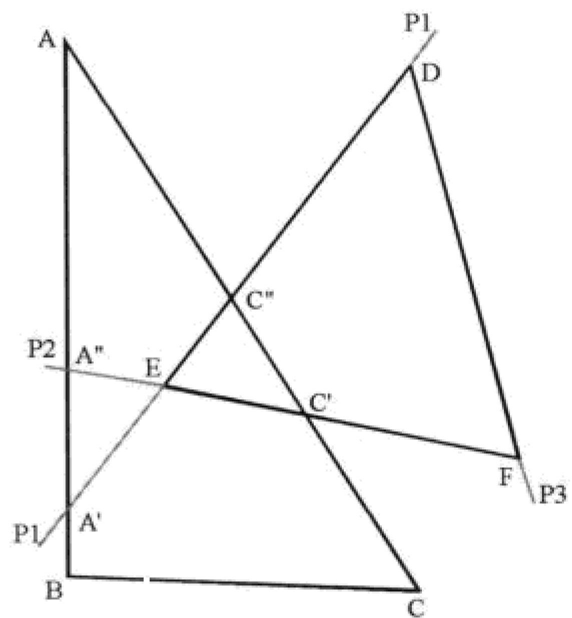


图 1.9.3　实体 X 被实体 Y 裁剪

在这里将实体 X 的面（AB、BC 和 CA）逐个用实体 Y 的 3 个平面（P1、P2 和 P3）进行裁剪（参见图 1.9.2)。“裁剪”意味着要根据实体 Y 的面所在的平面，将实体 X 的所有面进行分类。下面这个函数实现了裁剪过程中使用的“根据平面切割实体面”的功能。如果某个面完全位于平面的后面，这个函数就会返回–1；如果那个面完全位于平面的前面，函数就返回 1；如果平面与那个面交叉（也就是平面将那个面切割开来），函数就返回 0。变量 `frontFace` 和 `backFace` 表示的是函数返回的两个新面。之后这个裁剪过程还要将两个新面中的其中一个抛弃掉。

```
int Face::Split(Plane& plane, Face& frontFace,
                Face& backFace)
{
    Vertex vertex1    = m_points.first();
    Vertex vertex2    = m_points.back();
    float  fB;
    float  fA                 = plane.DistTo(vertex2._xyz);

    for_each(vertex in m_poins)
    {
            vertex1 = *vertex;
            fB      = plane.DistTo(vertex1._xyz);
            if(fB > EPSILON)
            {
                    if(fA < -EPSILON)
```

```
                {
                        float   t = -fA /(fB - fA);
                        Vertex midvertex = vertex1 +
                                           (vertex2-
                                            vertex1)*t;
                        frontFace << midvertex;
                        backFace << midvertex;
                }
                frontFace<<vertex1;
        }
        else if(fB < -EPSILON)
        {
                if(fA > EPSILON)
                {
                        float t = -fA /(fB - fA);
                        Vertex midvertex = vertex1 +
                                           (vertex2-
                                            vertex1)*t;
                        frontFace<<midvertex;
                        backFace <<midvertex;
                }
                backFace <<vertex1;
        }
        else
        {
                frontFace << vertex1;
                backFace << vertex1;
        }
        vertex2 = vertex1;
        fA   = fB;
    }
    if(m_points.size()==frontFace.size())
          return 1;
    else if m_points.size()==backFace.size()
          return -1;
    return 0;
}
```

在裁剪过程中，X 实体的某些面会被 Y 实体的平面分割成两个新的面。如果某个面完全位于 P1、P2 和 P3 中任意一个平面的后面，那就会对这个面继续进行裁剪，直到它被所有这 3 个平面（P1、P2 和 P3）都裁剪过为止。如果那个面继续存在，最后的结果是仍然完全位于 Y 的所有平面的后面，那就丢弃这个面。其他面或者新产生出来的面，就会添加最后的结果实体中。

首先是从面 AB、BC 和 CA 开始，依次用平面 P1、P2 和 P3 进行裁剪。AB 被平面 P1 分割成 AA"和 A"B。AA"位于 P1 的前面，因此就不需要再用 P2 和 P3 来分割了，在用 P2 分割 A'之前，AA"就会被添加到最终结果实体中。AB 被添加最后实体中。A"B 位于 P1 的后面，并由 P2 和 P3 进行分割。被 P2 裁剪之后，A"B 位于 P2 的前面（两端都位于平面 P2 的前面），因此，可以把 A"B 添加到最终结果中（而不再需要用 P3 去分割了）。

接下来可以看到，BC 位于 P1 的后面（两端都位于 P1 的后面），因此我们就继续用 P2 来分割 BC。而 BC 位于 P2 的前面，于是我们就把它添加到最终的结果实体中（参见图 1.9.3），然后再处理其他的面。

CA 被 P1 分割，产生了 AC'和 C'C。AC'位于 P1 的前面，所以被添加到最终的结果中（参见图 1.9.3）。C'C 则继续由 P2 来分割。正如我们在图 1.9.2 中看到的，C'C 被 P2 分割成 2 个面：C'C"和 C"C。C"C 位于 P2 的前面，可以直接把它添加到最终结果中（参见图 1.9.3），而 C'C"则还需要被实体 Y 的最后一个平面 P3 来分割，C'C"完全位于 P3 的后面，于是我们只好丢弃 C'C"，不把它添加到最后的实体中。

到此为止，我们已经解决了一半的问题。下一步是要用 X 的平面来裁剪 Y 的实体面（参见图 1.9.4）。用实体 X 的 3 个原始面（AB、BC 和 CA）以及它们所在的平面，来裁剪实体 Y 的 3 个面（DE、DF 和 FD）。沿用与上面一样的算法，任何位于实体 X 的 3 个平面（P1、P2 和 P3）后面的面都会被裁剪掉（见图 1.9.5）。如果某个面被 P1、P2 和 P3 都裁剪过，这个面也不会添加到最终结果中（见图 1.9.4）。

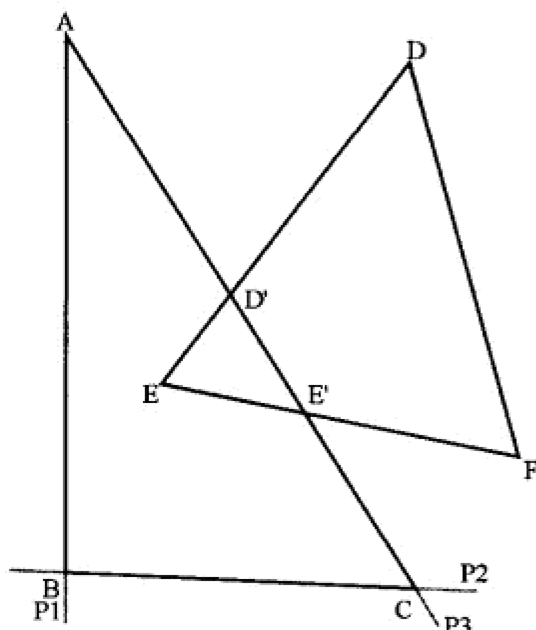


图 1.9.4 用 P1、P2 和 P3 裁剪 DEF

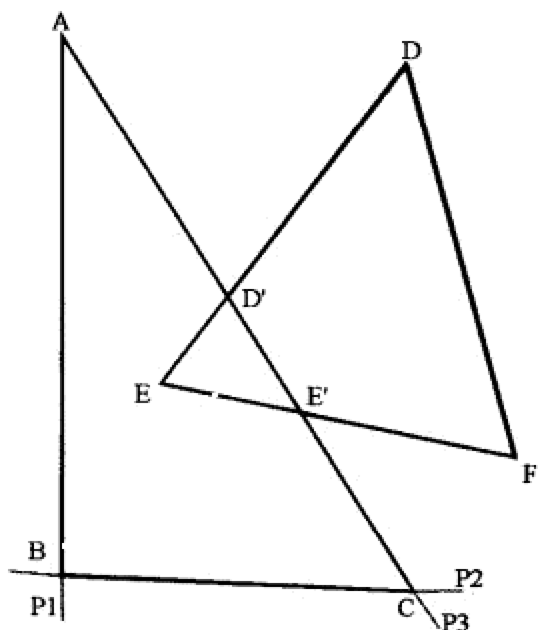


图 1.9.5 用实体 X 裁剪实体 Y

我们从面 DE 开始。DE 完全位于 P1 的后面，并被 P2 裁剪。DE 被 P2 分割成 2 个新的面。其中，DD'位于 P2 的前面，而 D'E 则位于 P2 的后面，所以 DD'被添加到最终结果中，而 D'E 会继续进行裁剪。D'E 完全位于 P3 的后面，所以不会添加到最终结果中。用同样的逻辑对待 EF，EE'被抛弃，而 E'F 则被添加到最终结果中。

所有添加到最终结果实体中的面就实现了平面 ABC 和 DEF 之间的并集运算，如图 1.9.6 所示。不要担心最终的结果实体被切割成几个片片。AB 实际上就是 AA'加上 A'B（参见图 1.9.7）。在以后分析 CSG 的 BSP 算法时，再来处理这些由 CSG 布尔运算带来的不必要的分割。

不必惊讶，到目前为止，我们已经覆盖了经典的 CSG 法 90%的内容。而交集运算和差集运算只不过是并集运算的变种而已。

2. 交集运算

交集运算（图 1.9.1(c)）的实现过程是这样的：首先将 2 个实体都反转过来，对它们进行并集运算，然后将它们再反转回来，就可以得到最终的结果。实体的反转就是将实体的所有

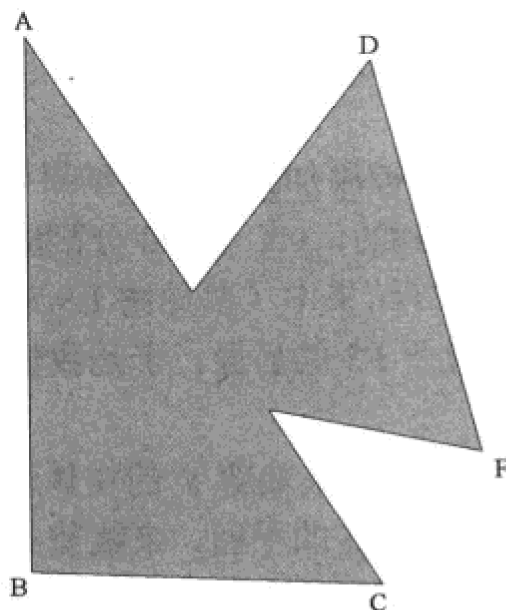


图 1.9.6 并集运算的最终结果

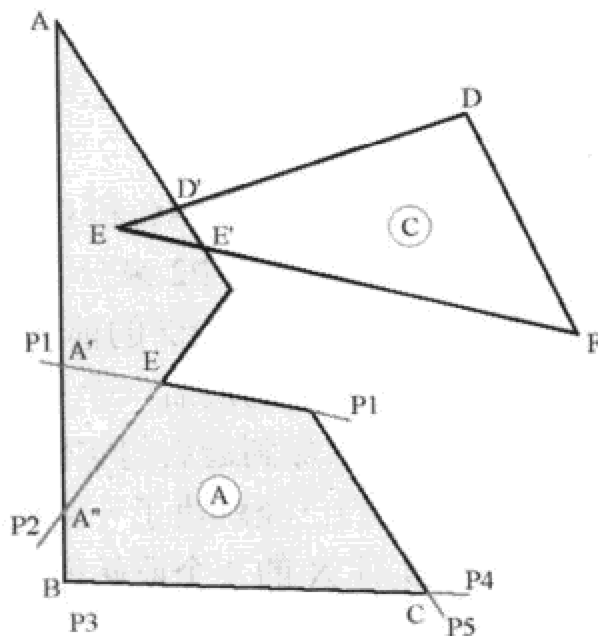


图 1.9.7 多实体的 CSG 法

面所在的平面（平面法线和平面本身）翻转过来。通过这个操作，实体内部变为空，而它们的边界外则变为实体。并集操作会把位于实体 AB、BC 和 AC'中的部分（参见图 1.9.2）以及实体 DD'、EE'和 DF 中的部分消掉（参见图 1.9.4）。

3. 差集运算

差集运算（图 1.9.1(d)和图 1.9.1(e)）的实现过程是这样的：反转实体 A，并与实体 B 进行并集运算，然后再进行反转，这样就从实体 A 中减去了实体 B。另一方面，如果要从实体 B 中减去实体 A，就先把实体 B 反转过来，与实体 A 进行并集运算，然后再反转回来，得到最后的结果。下面这个表格显示的是各种 CSG 的布尔运算及其对应的运算符：

布尔运算	运算符
Union	+
Intersection	&
Subtraction	−
Reversed solid	!

使用这些运算符就可以把交集和差集运算定义成下列形式：

$$A \& B = !(!A + !B)$$

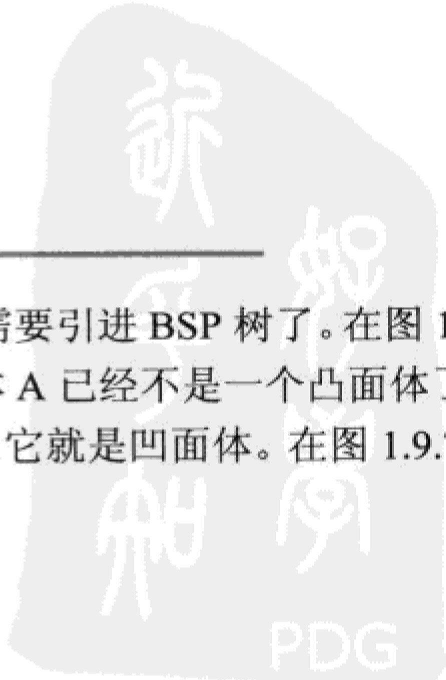
$$A - B = !(!A + B)$$

和

$$B - A = !(A + !B)$$

1.9.2 为什么要使用 BSP 树

当参与 CSG 布尔运算的实体不再是凸面体的时候，就需要引进 BSP 树了。在图 1.9.7 中，我们可能要从实体 A 中减去实体 C。从图中可以看到，实体 A 已经不是一个凸面体了。一个实体如果至少有一个面所在的平面可以将自身分割开，那么它就是凹面体。在图 1.9.7 中，平



面 P1 和 P2 可以将实体 A 分割开来，所以实体 A 就是一个凹面体。这时候，如果进行实体 A 和实体 C 的并集运算，CSG 法就会出问题了。我们假设实体 A 的所有面都已经裁剪完毕，现在开始处理实体 B 的面。用平面 P2、P3、P4 和 P5 来分割 EF。我们马上就能判断出来，EE'会要被裁掉，因为它完全位于实体 A 中。但是，EE'也在 P3 的后面，所以要做进一步的裁剪。最后，EE'位于 P4 和 P2 的后面，但却位于 P1 的前面。由于至少是位于 P2、P3、P4 和 P5 中某一个平面的前面，所以 EE'会添加到最终的结果中。但是，如果 EE'成为最终结果的一部分，那就错了。因为它是在实体 A 内部的啊。这该怎么办呢？

解决的办法是将凹面体分割成若干个较小的凸面区域（凸面体），然后在这些凸面区域之间执行 CSG 的布尔运算。众所周知，BSP 叶子树可以解决这个问题。BSP 叶子树，或称 *Beam Tree*，可以帮助我们把任何实体分割成较小的凸面区域。

1.9.3 BSP 树的实现

优秀的 BSP 算法所需要的是一个功能出色的多边形分割函数，以及良好的程序内部管理。对于程序的内部管理，我们引入了一个节点结构，它包含一个分割平面以及前节点和后节点的 2 个引用。另外，我们还引入了一个叶子结构，它包含的是 BSP 的多边形。见图 1.9.8。

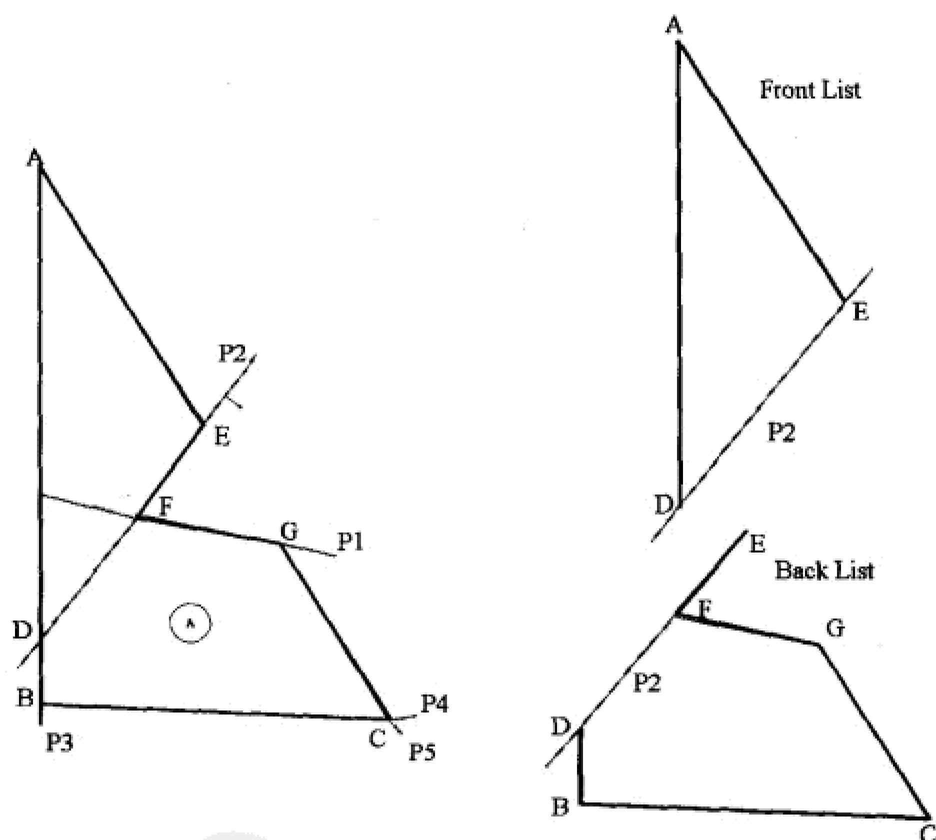


图 1.9.8 用平面 P2 分割 A

随书光盘中提供了完全的带叶二叉树的实现。下面就总结一下实体 A 的二叉树构造过程，如图 1.9.8 所示。

1. 随机选择一个平面，作为二叉树的根节点（也就是初始分割平面）。这里假定选择的平面是 P2。

2. 给其所在平面被选为初始分割平面的面做一个标识。

3．用这个分割平面分割实体。

4．所有位于该分割平面前面的面（包括分割产生的新面）都添加到 *front* 列表中。如果某个面与分割平面共面，那么根据分割平面的方向，可以对这个面进行如下划分：如果这个面的法线方向与分割平面相同，那就把这个面添加到 *front* 列表中；否则就添加到 *back* 列表中。

5．所有位于分割平面后面的面（包括分割产生的新面）都被添加到 *back* 列表中。

6．把 *front* 列表和 *back* 列表分别看成是根节点（初始分割平面）的前节点和后节点。

7．对 *back* 列表和 *front* 列表从第 1 步开始重复上述过程。但要注意，应选择没有被标识过的面来确定下一个分割平面：

- 对于 *front* 列表，当所有多边形都被用做分割平面，或者前多边形的大小为 1 时，就停止这个递归循环过程，并创建一个终端叶子节点，把 *front* 列表中的所有多边形添加到这个叶子节点中。
- 对于 *back* 列表，一直执行这个递归过程，直到列表中没有多边形为止。这个时候，再创建一个终端叶子节点，并把它标识为实体。

用这种算法建立的 BSP 的内部节点只包含分割平面以及前方与后方节点的指针。图 1.9.9 所示为 S01:dA 的最终 BSF 树。

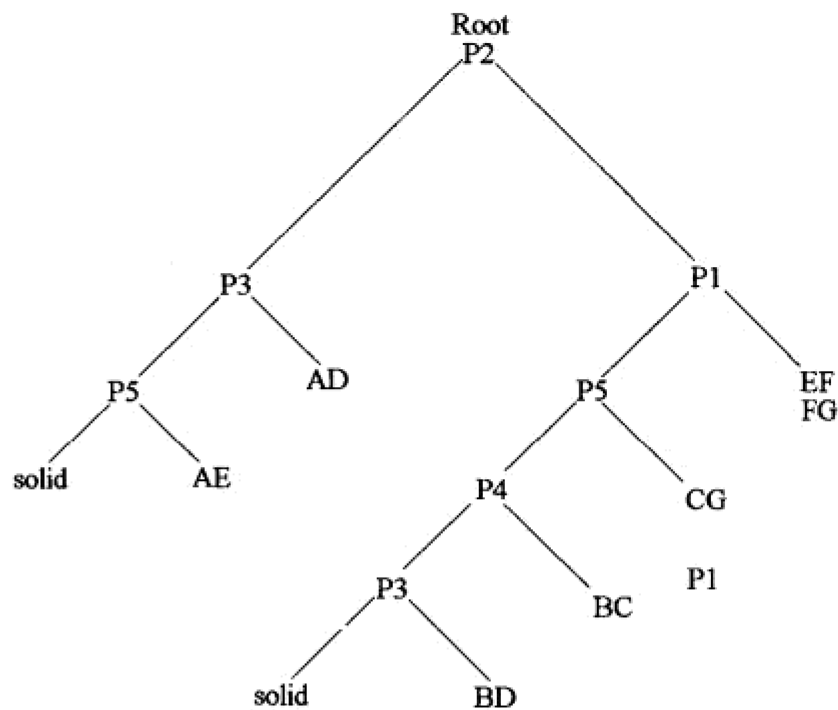


图 1.9.9 由实体 A 生成的最终的 BSP 树

1.9.4 组合装配

最后，我们就得到了完成 CSG 算法所必需的元素。通过使用 BSP 树，我们就不再需要用实体多边形来分割其他的实体多边形了。取而代之的则是，用其他实体的 BSP 树来分割多边形，反之亦然。实体 X 和实体 Y 之间的 CSG 处理管道是这样的：

1．创建实体 X 的 BSP 树。

2．用实体 X 的 BSP 树分割实体 Y，并保留合适的多边形。

3．创建实体 Y 的 BSP 树。

4．用实体 Y 的 BSP 树分割实体 X，并保留合适的多边形。

用 BSP 树裁剪实体的过程就是：用所有的区分节点（BSP 树的分割平面）划分所有的实体多边形。这个分割过程从根节点开始，持续到叶子节点为止。下面这段代码向我们示范了该递归过程：

```
void Bsp::Recurs_Clip(int node,
                      list<Polygon>& polys2Clip,
                      list<Polygon>& finalList)
{
    BspNode  node = GetNode(node);
    if(node->IsLeaf())
    {
        if(node->IsSolid())
        {
            if(m_csgUnion)
            {
                return;
            }
            finalList << polys2Clip;
        }
        else
        {
            if(!m_csgUnion)
            {
                return;
            }
            finalList << polys2Clip;
        }
        return;
    }

    for_each(pSpPoly in polys2Clip)
    {
        Where_Is rp1 = pSpPoly.Classify(
        node->GetPlane());
        switch (rp1)
        {
        case ON_PLANE:
            if(SameFacing(pSpPoly,node.GetPlane()))
            {
                 backPolys.push_back(pSpPoly);
            }
            else
            {
                if(m_csgUnion == FALSE)
                    frontPolys.push_back(pSpPoly);
                else
                    backPolys.push_back(pSpPoly);
            }
            break;
        case ON_FRONT:
```

```
                frontPolys.push_back(pSpPoly);
                break;
            case ON_BACK:
                backPolys.push_back(pSpPoly);
                break;
            case ON_SPLIT:
                {
                    Polygon fp, bp;
                    pSpPoly.Split(node->GetPlane(),fp,bp);
                    frontPolys.push_back(fp);
                    backPolys.push_back(bp);
                }
                break;
            }
        }
        if(backPolys.size())
            Recurs_Clip(node->BackNodeIndex(),
                        backPolys, finalList);
        if(frontPolys.size())
            Recurs_Clip(node->FrontNodeIndex(),
                        frontPolys, finalList);
    }
```

在根节点处，所有的多边形被相应地划分到 *front* 列表和 *back* 列表中。如果用具体例子来说明的话，就是从 BSP 树的根节点开始（参见图 1.9.9），来分割多边形 DE、EF 和 FD（参见图 1.9.7）。所有组成实体 C 的多边形都被根节点裁剪过。最后，有些多边形被分割了，有些则没有被分割。所有位于分割平面前面的多边形都被添加到 *front* 列表中，而所有位于分割平面后面的多边形则被添加到 *back* 列表中。对于标准的算法，所有与分割平面共面的多边形都会被添加到 *back* 列表中。根据当前使用的 CSG 布尔运算的不同，上述处理过程也会有所不同。读者可以通过随书光盘中的程序代码，来详细了解这个算法。接下来，P3 和 P1 会分别裁剪 *front* 列表和 *back* 列表。这个过程会一直重复下去，直到剩下的多边形到达叶子节点为止。当某个列表到达叶子节点时，就可以决定哪些多边形需要被裁掉，哪些要保留下来组成最后的 CSG 实体。回到图 1.9.1，对于其中列举的那些 CSG 布尔运算，如果要使用这个算法，其相关的应用条件如下：

CSG 的布尔运算	**描述**
并集	抛弃所有以实体（solid）叶子节点结尾的列表
差集	应用公式!(!*A*+*B*)或公式!(!*BA*+*AB*)，或者对裁剪算法做些微小的改动，使之不需要进行太多次的前后翻转操作（可参考示例代码）。
交集	采用公式!(!*A*+!*B*)，或者简单地抛弃所有以空叶子节点结束的列表。

1.9.5 总结

本文介绍的这个 CSG 算法在同一时刻只能处理 2 个实体，但是经过递归式的改造，就可以在多个实体之间执行复杂的 CSG 布尔运算。随书光盘中的示范代码实现可以很好地说明这

一点。通过减少一些不必要的分割操作，就可以提高该算法的性能，也就是说，如果某个多边形与其他实体有接触，那就不用为它创建 BSP 树了。这需要为每个面维护一个包围盒，并在 CSG 构造工作之前，检测各个面与其他实体的包围盒之间的位置关系。如果该多边形与其他实体的包围盒没有任何接触，那就直接把这个多边形复制到最终结果中。减少分割次数的另一个方法是，记录好每个多边形被分割的次数，并在其分割产生的每个片段中保存一个对该多边形的引用。最后，如果引用同一个多边形的所有分割片段都存活下来（即没有被算法抛弃），那就用那个原始的多边形来替换这些分割片段。现在，就去看看示范程序，并享受 CSG 的美好吧！

1.9.6　参考文献

[Rottensteiner] Rottensteiner, Franz. “Constructive Solid Geometry.” Available online at *http://www.ipf.tuwien.ac.at/fr/buildings/diss/node38.html*.

1.10 在游戏中集成 Lua

eV Interactive 公司，Matthew Harmon
matt@ev-interactive.com

在过去的十年中，计算机游戏和游戏机游戏的开发逐渐减少了硬编码的使用，取而代之的是越来越多地使用数据驱动的设计方法。这样做的好处不胜枚举，从更从容地进行开发，到一个完整的亚文化体系，这种体系让我们可以开发出用户群更广的游戏产品。实际上，当站在技术的立场上审视这些产品的时候就会发现，现在很多游戏都更像是一个虚拟机或操作系统，而不是一个单独的、有特殊用途的应用程序。

数据驱动式设计最强有力的推动元素，无疑是嵌入式语言的应用。简单的数据驱动式设计专注于游戏硬数据的外部化，这些硬数据包括地图数据、实体参数、if/then 触发器和游戏引擎的其他参量。通过使用嵌入式语言或者“脚本”语言，游戏还可以将逻辑外部化。这为我们打开了一个最终用户可以自己定制游戏的全新世界，同时也彻底改变了游戏和游戏系统的开发模式。

但不幸的是，从头开始开发一个嵌入式语言可不是一件容易的事情。即使市面上有各种各样工具可以辅助我们做这项工作，但是要想最后得到一个完整的、健壮可靠的、灵活的系统，还是要耗费大量的工时。幸运的是，现在已经有几个现成的解决方案，可以很快地集成到游戏项目中。诸如 Python、Ruby 和 Lua 之类的嵌入式语言，都被证明是可靠的，并被业界广泛地采用。另外还有其他一些解决方案，其中一些还是专门针对游戏度身定制的，但是那些解决方案并没有这些通用编程语言那么流行。

本文是为了向大家展示将一个现有的嵌入式语言集成到游戏系统中是多么快速和简单。对于嵌入式语言的用例，本文选择了一个迅速被游戏开发人群所拥护的语言包——Lua。

1.10.1 Lua 的概况

Lua 是一个简单、但健壮可靠的嵌入式语言，它所拥有的很多特性深深地吸引了众多的游戏开发人员。Lua 已经成功地应用在很多专业的游戏中，还有更多的游戏正在集成使用 Lua。不妨浏览一下 Lua “uses” 网站（*www.lua.org/uses.html*），你马上就能了解到该语言是如何突袭到游戏行业中的。游戏开发人员很可能会把 21 世纪的第 1 个十年命名为“Lua 的十年”。

下面的这些特性使得 Lua 特别适合游戏开发：

- Lua 是很小巧的。一个完整的 Win32 库只有八百 KB 的代码。实际上，随书光盘中的 Demo 程序，其可执行文件只有 145 KB！
- Lua 可以运行文本脚本或者预先编译的字节码文件。开发人员可以公开某些脚本，隐藏其他的脚本，甚至可以省略 Lua 的词法分析器、语法分析器和代码生成器。通过这样的方法，有人曾经把 Lua 程序压缩到不足 25 KB！
- Lua 是一个完整的、功能强大的编程语言。
- Lua 特别适合那些“半程序员”。Lua 的数据类型定义是动态的，并且含有自动内存管理和垃圾收集机制。
- Lua 被证明是可靠的，有一个非常活跃的用户群体。
- 而且，最重要的是，Lua 易于嵌入，也很容易与 C 语言接口。

1. 一个小程序

Lua 程序的结构非常直接明了。虽然全面地描述 Lua 语言远远超出了本文的范畴，但可以肯定的是，大部分的 C 语言程序员都会快速地掌握 Lua 的基本内容。下面一小段 Lua 代码可以帮助我们熟悉 Lua 程序。

```
—定义一个函数 （"—"表示程序注释）
function EruptSequence()
    Camera.SetMode(CAMMODE_ORBIT);
    Camera.SetOrbitRate(PIOVER2);
    ent = Entity.Find("Volcano");
    Camera.SetTargetEntity(ent);
    end

—每 20 分种运行一次 EruptSequence 函数
while (1) do
    Script.WaitSeconds(20 * 60);
    EruptSequence();
    end
```

这段代码向我们展示了一些简单的 Lua 语法。这个代码片段实际上是从游戏引擎中调用了若干函数，这些函数是用 Lua 注册在游戏引擎中的。

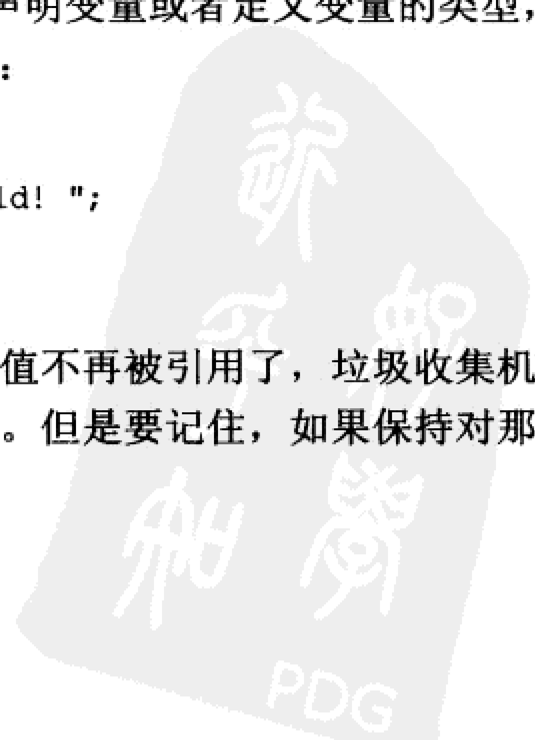
2. 动态数据类型定义

Lua 的数据类型定义是动态的。不需要声明变量或者定义变量的类型，只要拿来用就好。例如，在 Lua 中，这样的语句绝对是合法的：

```
b = 24.5;
b = "Wait! Now b is a string.  How wild! ";
```

3. 自动内存管理

Lua 会自动地对内存进行管理。当某个值不再被引用了，垃圾收集机制就可以把它处理掉。这样，就不需要专门去释放任何内存了。但是要记住，如果保持对那些不再需要的对象的引用，就会防止这些对象被清理掉。



4. Lua 状态

有几个 Lua 的术语可以实现快速定义，其中第 1 个就是 Lua *state*（状态）。从本质上讲，Lua state 是 Lua 解释器的一个单独的操作环境，包括堆栈、执行状态和全局变量等等。当把 Lua 嵌入到游戏中时，Lua state 就是一个接口对象。大多数程序架构仅需要一个 Lua state。

5. Lua 段

一个 *chunk*（段）就是一系列的声明，由 Lua 来翻译并有选择地执行。可以把 chunk 看成是加载到 Lua 环境中可供使用的程序或者程序片段。chunk 通常来自脚本文件，也可以来自另外一个 chunk 的静态文本字串，甚至可以直接来自命令行控制台。在后一种情况下，chunk（段）可能是一个由用户输入的单独的声明。

1.10.2 Lua 与 C 语言的接口

从一个游戏程序员的角度来看，Lua 最吸引人的地方之一就是它与 C 语言之间非常简单的接口。在真正开始介绍如何在 C 语言中嵌入 Lua，并为 Lua 提供脚本之前，先来看看如何才能让 Lua 的解释器可以使用 C 函数。

这两个语言之间的通信是通过一个堆栈来提供的。这个堆栈服务于两个语言，既像一个绝缘层，又像一个翻译机制。由于 Lua 的数据类型定义是动态的，所以一个单独的堆栈入口就可以表示任何类型的 Lua 数据，因此系统提供了一些翻译例程来把这些数据转换成 C 语言可以使用的数据格式。

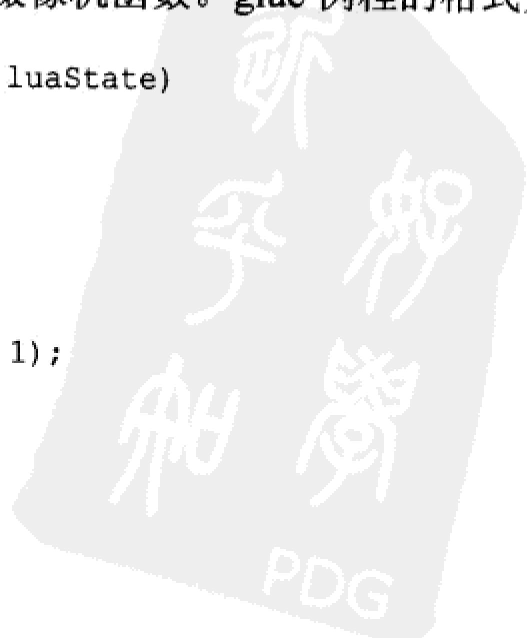
1. 在 Lua 中调用 C 语言代码

很多程序员要完成的第 1 件事情，是把他们游戏引擎中的某些功能对 Lua 公开，并对这些功能实现基于脚本的控制。只需要用 Lua 来注册实现一个“glue（接口粘合）”例程，就可以完成上述工作。假设需要一个 Lua 脚本，来实现对游戏中摄像机位置的控制。这个 Lua 脚本应该是这样的：

```
--使摄像机对准火山
Camera.SetTargetPos(100.0, 40.0, 230.0);
```

首先要编写一个 C 语言的“glue”例程，从 Lua 脚本中获取参数，转换为 C 语言中的数据类型，然后再用这些参数来调用游戏引擎中的摄像机函数。glue 例程的格式大致如下：

```
static int LuaSetTargetPos(lua_State* luaState)
{
    float  x;
    float  y;
    float  z;

    x = (float)lua_tonumber(luaState, 1);
```



```
        y = (float)lua_tonumber(luaState, 2);
        z = (float)lua_tonumber(luaState, 3);

        CameraSetTargetPos(x, y, z);
        return(0);
    }
```

当 Lua 调用一个 C 的函数时，它会把在脚本文件里找到的参数顺序地推入到通信栈中。然后，Lua 再调用为这个 C 函数注册的 glue 例程。调用这个 glue 例程时，要传递给该例程一个指向 lua_State 的指针，脚本就是在 lua_State 中运行的。我们要知道，Lua state 只是一个对象，它代表着 Lua 解释器的整体状态，包括所有的数据、通信栈和任何用 Lua 注册的函数。

然后，这个例子中的 glue 例程会尝试从通信栈中取出 3 个值，也就是摄像机的坐标（*x*, *y*, *z*）。调用 lua_toxxx()，从堆栈中读取数据。lua_toxxx() 会将堆栈中给定位移上的数据按照要求转换为 C 语言的数据类型。要注意的是，可以任意指定一个堆栈地址，而且第 1 个参数是从索引 1 开始，而不是从 0 开始。接下来，调用引擎中的摄像机定位函数，来完成对摄像机的操作。最后，这个例程会返回 0，表示并没有把任何数据放回到通信栈中作为返回 Lua 脚本的返回值。

由于 Lua 的类型定义是动态的，所以必须要知道所期待的参数是什么类型的，并进行适当的转换。在前面的例子中，需要用 3 个浮点值来表示摄像机的位置。Lua 把所有的数字值都设定为 double（双精度浮点）型，所以虽然 lua_tonumber() 会把动态的 Lua 值转换为 double（双精度浮点），但我们仍然要把它再强制转换成 float（浮点型）。Lua 提供了很多工具来操作通信栈，包括可以检查一个堆栈的入口是否是给定的类型。这样，当发现不正确的参数或者参数的个数不够时，glue 例程就可以给出详细的错误信息。

2. 将值返回到 Lua

现在，假设需要让 Lua 脚本来查询玩家当前的得分。在 Lua 脚本中，我们或许想做这样一些事情：

```
--当玩家得分为 9 时，终止这一关的游戏
if Game.GetPlayerScore() > 9 then
    TriggerEndOfLevel();
    end;
```

为了向 Lua 脚本返回玩家的得分，需要创建一个 glue 例程。这个 glue 例程会把一个数字值（玩家的得分）推到通信栈中，就像这样：

```
static int LuaGetPlayerScore(lua_State* luaState)
{
    lua_pushnumber(luaState, game->playerScore);
    return(1);
}
```

在这种情况下，glue 例程并没有从通信栈中取走任何参数，而是把一个值放到了堆栈中，然后返回 1 并通知 Lua。实际上，Lua 的函数可以有多个返回值，而且一个 C 语言的 glue 例程可以向通信栈返回多个值，这是完全合法的。

3. 注册 glue 例程

现在我们已经知道了该如何创建 glue 例程，但为了让脚本可以使用它们，我们还需要对它们进行注册。最简单的函数注册方法是使用 Lua 的库管理调用，这些调用可以简化相关函数的注册过程。

首先创建一个以 null 结尾的二维数组，数组的值由函数指针和函数的字符名称组成。Lua 脚本可以用这些字符名称来识别函数。

```
//摄像机库（camera library）的 glue 例程和函数名称
static const luaL_reg cameraLib[] =
{
    {"SetMode",           LuaSetMode       },
    {"GetMode",           LuaGetMode       },
    {"SetTargetPos",      LuaSetTargetPos  },
    {"SetTargetEnt",      LuaSetTargetEnt  },
    {"SetOrbitRadius",    LuaSetOrbitRadius},
    {"SetOrbitRate",      LuaSetOrbitRate  },
    {"SetFov",            LuaSetFov        },
    {"SetWobble",         LuaSetWobble     },
    {"SetZoomPos",        LuaSetZoomPos    },
    {"SetPivot",          LuaSetPivot      },
    {"EnableShake",       LuaEnableShake   },
    {NULL, NULL }
};
```

接下来，调用一个很便利的 Lua 效用函数，来注册上述数组中的每一个函数。

```
luaL_openlib(luaState, "Camera", cameraLib, 0);
```

第 1 个参数是当前正在使用的 Lua state。第 2 个参数是为这个库起的名称。第 3 个参数是由函数指针和函数名称组成的数组。在这个调用之后，Lua 脚本就可以使用这些函数了，且这些函数的名字会以库的名称作为前缀。这样就可以在 Lua 脚本中调用 C 语言的 glue 例程 `LuaSetOrbitRate()`，调用格式是 `Camera.SetOrbitRate()`。

这就是所有要做的。只用了很少的代码，一个游戏引擎的功能就可以很快速、很容易地对 Lua 公开了。还有很多比较复杂的方法可以把 Lua 和 C 链接起来，但由于本文只是介绍性质的，所以那些复杂的方法已经超出了本文的范畴。另外，Lua 也不是只能与 C 集成。有些工具（如 Luabind）还可以帮助我们建立 Lua 和 C++类之间的联系。

1.10.3 在游戏中嵌入 Lua

理解了如何在游戏引擎和 Lua 之间创建应用接口之后，下面就来了解一下在游戏中嵌入 Lua 的方法。

1. 创建和销毁 state

为了能够在游戏中使用 Lua，我们必须首先创建一个 Lua state，然后再加载并执行 Lua

代码段。下面的语句创建了一个 Lua state：

```
lua_State*    luaState = lua_open();
```

当要关闭游戏时，我们就关闭这个 state：

```
lua_close(luaState)
```

上述语句会释放解释器所使用的内存，并对加载到这个 state 中的代码进行垃圾收集工作。

2. 加载并执行代码

一旦 Lua 初始化完毕，且我们也按照前面的方法注册了新的函数，那就要开始加载并执行 Lua 代码了。Lua 提供了相应的支持，可以从任意的数据源向解释器提供数据。对于这个例子，我们会使用一些 Lua 的帮助器例程，从文件和字符串中获得输入数据。下列语句可以执行一个脚本：

```
lua_dofile(luaState, "LuaScriptFile.lua");
```

如果存在某些执行点，这样的语句会让系统对脚本进行编译和执行。如果某个文件中什么都没有，只有函数定义，那就会定义所有这些函数，并把它们添加到 Lua state 中，但并不会执行任何实际的代码。

同样地，一个文本字符串中的 Lua 代码可以这样被加载和执行：

```
lua_dostring(luaState, "a=14; b=7;");
```

现在读者应该能猜测到，用 Lua 创建一个交互式的命令行控制台只需要几行简单的代码。

Lua 脚本加载工作的一个非常有趣的特性是，这些例程既可以接受文本脚本，也可以接受预翻译的二进制 Lua 代码。Lua 的发行版带有一个预译器 luac，它可以把脚本编译成二进制格式。完成这项工作之后，应用程序可以选择省略掉 Lua 的词法分析器和语法分析器组件，让代码开销更小。这个特性让开发人员也多了一个选择：他们可以把某些脚本保留为最初的文本格式以便进行修改，并对另外一些脚本进行编译，以防止别人偷窥。

我们也可以只加载代码和翻译代码，而不去执行代码，实现方式如下：

```
luaL_loadfile(luaState, "LuaScriptFile.lua");
```

3. 在 C 中调用 Lua 函数

正如前面提到的，我们可以把脚本加载到 Lua 中，但先不去执行，而是等以后需要的时候再执行实际的代码，这个功能是非常有用的。关于这一功能，有个特别好的例子：游戏实体的游戏逻辑脚本。在启动的时候，每个实体类可以加载一个 Lua 脚本，这个脚本定义了该实体在游戏中的行为。可能会有几个独立的 Lua 例程，分别负责设置、移动和渲染的工作。一个简单实体的 Lua 代码可能是这样的：

```
—虚拟“步行者”实体的函数

function WalkerSetup(walker)
```

```
    -- 步行者只存在了一分钟
    timer = 60.0;
    end;

function WalkerMove(walker, elapsedSec)
    timer = timer - elapsedSec;
    if (timer > 0.0) then
        Walker.DoAI(walker, elapsedSec);
        Walker.UpdateControls(walker);
        Walker.DoPhysics(walker, elapsedSec);
        Walker.UpdateAccelAndPos(walker, elapsedSec);
        end;
    else
        Walker.Destroy(walker);
        end;
    end;

function WalkerRender(walker)
    visible = Walker.GetVisibility(walker)
    if (visible == 1) then
        matrix = Walker.GetMatrix(walker);
        Game.SetModelMatrix(matrix);
        Game.RenderModel(Walker.GetModel(walker));
        end;
    end;
```

当这个实体类被初始化后，它就可以使用 `luaL_loadfile()` 来读取并编译脚本。然后，在游戏循环中，这个实体的 C 代码可以根据需要调用 Lua 脚本例程。要想在 C 代码中调用指定的 Lua 函数，首先要将这个 Lua 函数及其所有的参数放到通信栈中，然后再调用 `lua_call()`。一个调用 `WalkerMove()` 例程的示范代码如下：

```
—调用这个实体的移动（Move）脚本。
lua_getglobal(luaState, "WalkerMove");
lua_pushlightuserdata(luaState, this);
lua_pushnumber(luaState, elapsedSec);
lua_call(luaState, 2, 0);
```

第 1 个调用会将一个全局变量推入到通信栈中，该变量的名字是“WalkerMove”。这正好就是初始化 walker 类之后加载的脚本中的那个函数的名字。

接下来的调用会将 C++实体的 this 指针放进通信栈中，供 Lua 使用。事实上，Lua 不会直接使用这个值，因为它根本无法理解这个 C++对象。相反，Lua 只是把这个值回传给 C 的 glue 函数，这样这些函数就知道该使用哪个对象指针了。为了把 this 指针推进通信栈，需要调用 `lua_pushlightuserdata()`。“Light user data”是一个特殊的 Lua 数据类型，当 C 代码需要传递给 Lua 某个它不能理解的数据时，就可以使用这个数据类型。大多数情况下，就像上面的这个例子，会用 light user data 类型来保存指向 C/C++对象和结构的指针。

然后，只需要简单地调用 `lua_pushnumber()`，将从上一帧到现在过去的时间秒数推

入通信栈中即可。函数和所有的参数都已经正确地进入通信栈之后，就可以通过调用下列函数来执行相应的 Lua 代码了：

```
lua_call(luaState, 2, 0);
```

其中，第 2 个参量(2)告诉 Lua，有多少个参数被推入通信栈中；而第 3 个参量(0)告诉 Lua，我们想得到多少个返回值。Lua 接下来就会执行整个函数，执行完毕返回的时候，通信栈中就有了我们想要的返回值。

现在，可以在 Lua 中调用 C 函数并在 C 中调用 Lua 函数了。我们正一步一步地接近目标：把 Lua 集成到游戏中，不是吗？

1.10.4　实时性方面的考虑

注意，当使用 `lua_dofile()` 加载并执行 Lua chunk（段），或者使用 `lua_call()` 调用 Lua 脚本中的函数时，只有当被请求的 Lua 脚本处理完毕，这些调用才会返回。对于某些应用和架构而言，这样是没有问题的，但在很多游戏中，这样的使用方法是远远不够的，我们要能够根据时间来调度事件。

1. 一个基于时间的例子

举个例子，比方说在火山爆发时，要用 Lua 脚本来控制摄像机的移动。对于每一个动画帧，可以在 Lua 脚本中调用一个 `MoveCamera()` 函数。这个 Lua 函数会在内部更新计时器，如果计时器触发了新的摄像机状态，那就会调用不同的子函数。但是，这样做确实有些乱，而且我们更喜欢一个简单的线性脚本，大致如下：

```
—触发火山爆发的脚本
—在整个游戏过程中，每 20 分钟触发一次

function EruptSequence()
    —开始轻微地抖动摄像机，持续时间为 10 秒
    Camera.SetShakeMag(0.002);
    Script.WaitSeconds(10);

    —挂起玩家控制
    Game.SuspendPlayer();
    —让摄像机缓慢地围绕火山移动
    entTarget = Entity.FindByName("Volcano");
    Camera.SetMode(CMX_ORBIT);
    Camera.SetTargetEnt(entTarget);
    Camera.SetOrbitRadius(500.0);
    Camera.SetOrbitRate(0.003);
    —并且让摄像机开始震动
    Camera.SetShakeMag(0.01);
    Script.WaitSeconds(5);

    —OK，5 秒钟之后，逐渐减轻震动
```

```
        Camera.SetBankShakeMag(0.002);
        —然后把摄像机回复到玩家视角
        entPlayer = Entity.FindByName("Player");
        Camera.SetTargetEnt(entPlayer);
        Camera.SetMode(CMX_FIRSTPERSON);
        Game.ResumePlayer();
        Script.WaitSeconds(10);

        —10 秒钟之后，完全停止摄像机震动
        Camera.SetShakeMag(0.0);
        end

—这是脚本中第 1 个可执行代码
——个隐含的 main()，如果想每 20 分钟运行一次火山爆发的代码
while (1) do
        Script.WaitSeconds(20 * 60);
        EruptSequence();
        end
```

这段代码首先定义了一个函数，用于处理摄像机的系列动作。然后，该脚本每 20 分钟就重复调用一次这个函数。我们用一个新注册的例程来调度各种事件，该例程名为 `Script.WaitSeconds()`，它的作用是简单地按照给定的时间进行延时操作。

从理论上讲，这个脚本是没有问题的，但是如果脚本代码必须执行完毕，程序才能返回，那么如何才能按照自己的预期来运行真正的游戏循环呢？答案就在于 `Script.WaitSecond()` 函数的实现以及 Lua 的另一个特性——协程。

2. 协程支持

简单地讲，Lua 的协程支持特性可以让 Lua 脚本在中途随时终止脚本的执行，并把控制权交回给调用它的 C 程序。这是协同多任务的一种形式，在这里决定挂起执行并把控制权交回给调用它的程序的是代码自己，而不是操作系统。这被称为“屈从让权”，其使用方法是：在 Lua 脚本中使用 `yield`，或者在 C 语言 API（应用程序接口）中使用 `lua_yield()`。

正如读者想到的，我们的 C 函数实现 `Script.WaitSeconds()` 包含了一个对 `lua_yield()` 的调用，它会把控制权交还给游戏，这样在脚本挂起的同时，我们就可以渲染游戏帧了。如果脚本管理系统觉得已经过去了足够的时间，就可以调用 `lua_resume()` 来继续处理剩下的脚本代码。对游戏开发人员而言，协程支持是 Lua 语言包最吸引人、最重要的特性之一。

3. 多脚本支持

虽然只使用一个 Lua 脚本也可能会获得很好的结果，但是如果能同时运行很多的脚本，然后按照需求唤醒相应的脚本来完成所需的工作，或者直接从 C 代码中调用相应的脚本，似乎更为实用。通过一个线程管理系统，Lua 可以支持多个脚本的并发执行。

不幸的是，Lua 对“thread”一词的使用让初学者颇感迷惑。这个特定的词本身的含义是：抢占式多任务处理，但在 Lua 中却完全不是这么回事儿。一个 Lua 线程可以看成是主 `lua_State` 的一个子 state，它只运行自己的脚本。我们可以这样创建一个新的 state 或线程：

```
lua_State*   newState = lua_newthread(mainLuaState);
```

每个新创建的 state 可以共享最初这个 lua_State 所有的全局函数和变量，但却有自己的堆栈和执行状态，而且每个新的 state 可以独立地向调用它的程序交还控制权。这样，系统就可以管理很多的脚本，每个脚本都可能会因为各自不同的原因，向调用程序交还控制权。

于是，我们就得到了一个通用的架构，可以将 Lua 集成到游戏中。那就是开发一个脚本管理器，为系统正在运行的每一个脚本创建新的子 state。脚本管理器负责跟踪每个脚本被挂起（交出了控制权）的原因，并按照需要使交还过程继续下去。

1.10.5　脚本管理框架

使用前面已经了解的 state 和协程，我们就可以构造出一个基本的脚本管理系统来处理多个 Lua 脚本的创建和运行工作。该管理器最重要的工作是把跟踪和唤醒那些被挂起（已交出控制权）的脚本的工作封装起来。我们可以让脚本在给定持续时间内的几帧中交出控制权，或者在到达给定时间前交出控制权。我们还可以很容易地添加其他附加条件。

ON THE CD

这个范例框架程序是通过两个类来实现的：LUAMANAGER 和 LUASCRIPT。为了简短起见，这里就不再详细介绍了，随书光盘中提供了一个完整的功能框架。

1. 管理类

管理器通过调用 lua_open() 来初始化 Lua，创建一个 lua_State。它通过提供 CreatScript() 函数作为创建脚本对象的惟一方法，来维护一个正在运行的 LUASCRIPT 对象的链接列表。管理器还包含一个 Update() 函数。每完成一次游戏循环，就会调用一次 Update() 函数，并向下调用每个脚本对象各自的 Update() 函数。我们就是在这里来检查那些被挂起（已交出控制权）的脚本，并唤醒那些需要继续执行的脚本。

管理类对象 LUAMANAGER 的一个简单定义框架如下：

```
class LUAMANAGER
{
public:
                LUAMANAGER       (void);
              ~ LUAMANAGER       (void);
    LUASCRIPT*  CreateScript     (void);
    void        DestroyScript    (LUASCRIPT*  s);
    void        Update           (float elapsedSec);

private:

    lua_State*  masterState;
    LUASCRIPT*  head;
};
```

管理器还会在 Lua 解释器中注册一个常用脚本管理 glue 例程库，其中包括那些可以让某个脚

本根据时间和帧数来交还控制权的 Lua 例程。在 Lua 中，这些例程大致是这样的：

```
Script.WaitSec(seconds);
Script.WaitFrame(frames);
Script.WaitTime(timestamp);
```

后面的章节会详细讲解这些例程。

2. 脚本对象

脚本对象代表一个从（由管理器创建的）masterState 派生得到的 lua_State 子对象。每个脚本对象可以把一个 Lua 程序作为（Lua 的）线程来运行，并根据需要随时交出控制权或继续执行脚本。脚本对象还维护着一些额外的数据，让我们可以知道某个脚本被挂起的原因，以及何时再唤醒这个脚本。下面的代码就是一个 LUASCRIPT 对象的简单框架：

```
typedef enum
{
    YM_NONE,        // 尚未交出控制权
    YM_FRAME,       // 等待 x 个帧的时间
    YM_TIME,        // 等待 x 秒
} YIELDMODE;

class LUASCRIPT
{
public:
    void          RunFile   (char*   fileName);
    int           RunString (char*   buffer);
    LUASCRIPT*    Update    (UDWORD  elapsedSec);

private:
    lua_State*    childState;
    LUAMANAGER*   manager;
    LUASCRIPT*    next;
    YIELDMODE     yieldMode;
    int           waitFrame;
    float         waitTime;

                  LUASCRIPT  (void);
                  ~LUASCRIPT (void);
};
```

LUASCRIPT 脚本类为我们提供了执行 Lua 代码的机制。RunFile() 和 RunString() 为 Lua 解释器提供给定的源代码。脚本代码将一直运行下去，直到结束，或者交出控制权。

正如我们所期望的那样，LUASCRIPT 脚本类创建并维护着一个指向它所管理的新的 lua_State 对象的指针。但是，正如我们马上就会看到的，让 Lua 知道哪个 C 对象拥有自己独特的 lua_State，也是非常重要的。为此，我们会在 Lua 的一个数据结构——table 中保存一些数据。对 table 的详细解释并非本文的内容，但是你可以把它们想成是可以用任何值来索引的数组。

为了将 LUASCRIPT 对象的地址与那个新创建的 lua_State 对象关联起来，我们在 masterState 的全局 table 中增加了一个入口。使用 lua_State 的指针作为索引，因为我们知道它是独一无二的。然后，Lua_State 的地址会被传递给 glue 例程，通过使用这个地址可以检索 LUASCRIPT 对象的指针。创建了子 state 后，就可以完成这些工作：

```
//从主 state 创建一个新的 state（线程）
childState = lua_newthread(mgr->masterState);

//使用这个新 state 的指针作为键值，把该脚本对象的指针保存在全局 table 中
lua_pushlightuserdata(mgr->masterState, childState);
lua_pushlightuserdata(mgr->masterState, this );
lua_settable(mgr->masterState, LUA_GLOBALSINDEX );
```

看上去很复杂，但这与在一个对话框的 GWL_USERDATA 中保存一个窗口处理对象的地址非常类似。如此这般，当调用某个 glue 例程时，它就可以判断出来发出该调用请求的 LUASCRIPT 对象。

3. yield 例程

脚本管理系统的一个关键，就是让 Lua 线程可以因为各种不同的原因而交出控制权，并在需要的时候又得到控制权并继续执行。为了实现这个功能，我们编写了几个新的函数，并用 Lua 注册它们。例如，我们肯定喜欢下面这个调用，因为它可以让某个脚本在给定的时间段内保持挂起状态：

```
Script.WaitSeconds(seconds);
```

下面是实现上述调用的 glue 例程：

```
static int LuaWaitSeconds(lua_State* l)
{
    LUASCRIPT*  s;

    //得到一个和该脚本相关联的 C++对象的指针
    lua_pushlightuserdata(l, l);
    lua_gettable(l, LUA_GLOBALSINDEX);
    s = (LUASCRIPT*)lua_touserdata(l, -1);

    //保存被挂起的时间和等待状态（wait state）
    s->waitTime       = lua_tonumber(1);
    s->state          = YM_TIME;

    //让 Lua 返回，并挂起这个线程
    return(lua_yield(l, 0));
}
```

当在 Lua 脚本中调用这个 glue 线程时，我们并不知道是哪个 C++的 LUASCRIPT 对象在管理它，所以我们必须首先从之前保存该对象的 Lua 全局 table 中检索到这个对象的指针。然后，从通信栈中读取一个秒数（即脚本被挂起所持续的时间），并将之保存起来。我们还要告诉该

对象正在执行的这个等待的类型。最后，调用 lua_yield()，将脚本挂起并返回到 C 程序中。

现在，我们要做的事情只不过是在脚本的 Update() 例程中检查计时器。如果已经过去了足够的时间，就调用 lua_resume()，将脚本唤醒，让它继续执行。

ON THE CD

通过遵循这个模型，我们还可以根据时间、帧数，甚至是在旗子被举起时，唤醒被挂起的脚本。通过聪明的管理，一个脚本执行的动作甚至可以触发其他脚本的唤醒操作。读者可以在随书光盘中找到这个简单的脚本管理器的示范代码。

1.10.6 总结

本文告诉大家，不需要费什么力气，开发人员就可以快速地将 Lua 语言嵌入到他们的游戏中。如果再配合一些有效的管理，我们的游戏就可以很容易地利用到 Lua 语言的众多有用特性，包括运行多个脚本，挂起脚本的执行，以及在需要的时候使它继续执行。

关于如何编写一个脚本并使之快速地运行，还有很多 Lua 的详细信息，本文并没有谈及。作为认真考虑将 Lua 语言嵌入到游戏中的开发人员，其实应该好好地研究这些细节。特别是 Lua table 的概念，以及对通信栈的深入理解，这些都是非常有价值的，能帮助大家对 Lua 有更全面的了解。不妨浏览一下 *www.lua.org* 这个网站，这是 Lua 使用者的最好起点。

到目前为止，Lua 已经成功地应用到了很多的游戏中，我们也鼓励读者去尝试使用这个语言。如果你正考虑在游戏中嵌入某个语言，但还没有真正开始这项工作，那就花几个小时的时间，把 Lua 集成进去吧。这几个小时也许是你花在整个项目中的最有价值的时间了。你甚至会发现自己开始用一个全新的视角来看待游戏代码，并且反复不断地在那里琢磨：“如何才能让 Lua 更多地控制我的游戏呢？”

1.10.7 参考文献

[Ierusalimschy03] Roberto Ierusalimschy. *Programming in Lua*. Published by Lua.org, December 2003. *www.lua.org luabind.sourceforge.net*

1.11　用基于 policy 的设计改进 Freelist

Nathan Mefford
nmefford@yahoo.com

人们希望游戏一天比一天变得更动态化，而我们的内存管理策略也必须要跟得上这种需求。但不幸的是，动态内存分配有自己的不足，这限制了我们将内存处理成一个真正动态资源的能力。《游戏编程精粹 4》中曾经介绍了 *freelist*（空闲块列表）的概念[Glinker04]，这是一个专用的内存分配器，它通过把自己限定为只分配单一类型的对象，从而解决了动态内存分配的问题。

实际上，freelist 确实是一个提高游戏中内存分配性能的强有力的工具，它几乎适用于所有的游戏项目。然而，即使是对于这样一个基本概念，它的设计与实现也是问题重重，很多问题都还没有明确的答案。是否应该允许这个列表扩容？大块内存应该如何分配？如何在内部跟踪大块内存和小的空闲块？每次内存分配都要重新构造并销毁对象，还是只需构造并销毁一次？完全未被使用的内存块是否应该返回给内存管理器？为了满足不同的需求，有些项目最终提出了多个 freelist 的实现方案。还有一些 freelist 的实现方案，为了追求灵活性，把自己的接口搞得万分复杂，让用户难以理解和使用。

基于 policy 的设计模式可以让库的使用者解决这些设计问题，创建出灵活的、可重用性高的类对象，且不需要牺牲速度，也不用增加接口的复杂度。本文将向大家展示一个基于 policy 的 freelist（空闲块列表），这个 freelist 可以很容易地用不同的行为进行配置，以满足各种不同的需求和分配模式。除此之外，我们对[Glinker04]中的 Freelist 实现进行了改进，开发了一个缺省的参数化表示，其方法是紧密模仿操作符 new 和 delete 的行为，使应用程序不必再去初始化隐式类，并将每次内存分配工作的系统开销降至零。

1.11.1　Freelist 概述

有关游戏中的动态内存管理的问题，本文的先行篇[Glinker04]中有详细的讨论。除了处理速度慢之外，通过一个多用途的内存管理器来分配和释放内存，不仅会造成较差的访问局部性，更糟糕的是，还会产生内存碎片。另外，大部分的内存管理器还会在每次分配过程中增加一些不可见的簿记内存开销。这个开销听上去不会有多大，但是即使每次内存分配工作

只有 16 B 的开销，不多的几次分配之后，就会很快浪费掉 1 MB 或更多的内存。

为了解决这些问题，就提出了以 freelist 为形式的解决方案：一个适合运行时内存分配的类，它将自身功能限定为只能分配和释放单一类型、固定大小的对象。大家也许会问，一个只能分配单一类型的对象的内存分配器到底有什么用处？但事实证明，freelist 分配法可能在一个典型游戏中的很多情况下都是适合的。由一些固有的动态特效（如粒子特效和半透明的贴花特效）所分配的对象，是 freelist 分配的最佳候选者。freelist 可以提供一个简单而自然的机制，用于对象的再循环利用，像爬行世界中的汽车和步行者，在这种地方为这些对象的所有实例都分配内存是不太现实的。常见数据结构（如链表和树）中的节点也可以有效地利用 freelist 内存分配策略。实际上，现在流行的标准模板库（Standard Template Library，简称 STL）的一个实现——*STLPort*[STLPort04]，其缺省的内存分配器就是一个定制的 freelist 内存分配器。很多现代的设计模式都会产生数量庞大的类，以便它们可以协同工作，完成一个复杂的任务。例如，策略、状态和装饰者模式的应用[GoF95]，有时会产生很多小型对象。在这种情况下，这些小型对象的内存分配工作最好由 freelist 来管理。简而言之，任何需要在运行时频繁分配和释放内存的对象类型，以及那些内存可以多次循环使用的对象，都可以从一个专门化的 freelist 中受益。

虽然《游戏编程精粹 4》中的那篇文章提供了一个 freelist 的程序实现，但是由于它一系列严格的硬编码设计决策，使得最后的库相对来说不是那么灵活。例如，它的容量是不可变的，并且使用了一个附加的列表来跟踪空闲内存块。另外一个例子是，为了避免在每次分配时都要调用构造函数和析构函数，算法会将已满额分配内存的对象保存在列表中。这个决策可能会提高程序性能，但代价是增加了有失直观的内存分配行为，它使得应用程序必须在每次分配内存之前都要进行初始化并在最后进行清理工作。对很多情况而言，这些决策是完全适合的，之前那篇文章展示的 freelist 也可以照常使用。但不幸的是，在这个特殊的 freelist 实现中，那些折中性的决策最终限制了它的适应性，使之无法满足各种不同的需求和内存分配模式。在有些情况下，可能会需要一个容量可大可小的 freelist；有些情况则可以从模拟 new 和 delete 操作符的行为中受益；而另外一些情况下，我们可能无法负担由跟踪空闲内存块的附加列表所带来的额外内存开销。

与其把这些特殊的折中方案强加给用户，不如去开发一个单一的 freelist，让用户可以根据自己的情况，选择适合自己的折中方案。当然了，这种灵活性和可重用性的获得不能以牺牲性能、易用性和安全性为代价。

1.11.2 Policy：雷霆救兵

C++中有一个设计机制，使安全、高效和高可定制化的行为成为可能，这就是 policy。在 C++的设计环境中，*policy* 到底是什么呢？从最基本的层面上讲，*policy* 只是简单地定义了一个接口，这个接口可能包含成员函数、成员变量和类型定义。任何实现这个接口的类都被称为 *policy class*（policy 类）。一个给定的 policy 可以有无限数量的 policy 类的实现。policy 和 policy 类本身是没有用处的。只有设计了其他的类，来使用一个给定的 policy 时，它们才有用武之地。使用 policy 和 policy class 的类被称为宿主或者宿主类。

在清楚了这些基本的定义之后，我们就该问了，policy、policy 类以及宿主类是如何协

调使用的呢？它们能给我们带来什么好处呢？游戏程序员向来就是注重实效的人，因此文章写到这里，笔者认为与抽象的定义和理论相比，几行程序代码能帮助我们更好地理解 policy。

```
template <class APolicy>
class AHost : public APolicy
{
    ...
    void DoSomething()
    {
        ...
        APolicy::Foo();
        ...
    }
};
...
//下面是客户端程序代码
AHost< MyPolicy > hostInstance;
hostInstance.DoSomething();
```

第一眼看上去，这好像也没什么特殊的，但如果仔细地看，就发现这几行代码实际上包含着惊人的力量、灵活性和完善性。在这个简单的例子中，AHost 是宿主类，它是围绕着一个名为 APolicy 的 policy 而设计的。当 AHost 被实例化的时候，我们需要提供一个 policy class，它不但要实现 APolicy 接口，还要作为一个模板参数，让宿主类可以访问 policy 的具体实现。当调用 AHost::DoSomething()时，AHost 就会调用 APolicy::Foo()，将自己的某些实现委托给 policy class。这样，AHost 的行为就可以让用户来配置，而不是由 AHost 的作者通过硬编码来控制。这种可以配置宿主类行为的能力是基于 policy 的类设计方法的核心所在。

好戏还在后面。请注意，在前面的例子中，宿主类是从 policy class 公共派生得到的，这就很方便地完成了宿主类和某个特定 policy 类的绑定任务，以及那些由 policy 类定义的结构的聚合工作。由于我们选择将 policy 类作为宿主类的一个公共基类，所以 policy 类就可以用它自己的公共函数来扩展宿主类的接口。不需要修改宿主类或者因为某些个别实现才使用的函数和变量将基本接口复杂化，复杂的 policy 类就可以公开一个根据自己的功能和特性度身定制的强化接口。如果用户以后切换到一个接口更小的 policy，那么编译程序就会捕获这个新的 policy 类不支持的所有调用，有效地强制执行设计约束。

这种 policy 的实现方法可以平衡不完整的实例化。在 C++中，如果一个模板函数从头至尾都未被调用过，那么它不会被实例化，而且编译程序甚至不会理会它，可能只有语法检查例外。在我们的例子中，这就意味着那些没有定义或声明 Foo()函数的 policy 也可以用来配置 AHost，这也许是因为 Foo()函数对于某些具体的 policy 类根本就没有什么意义。如果在后面又调用了 AHost::DoSomething()，而 Foo()函数又未被实现，那么编译程序马上就会报告这个错误，并严格地、自动地强制执行 policy 的设计和 policy 类的限制。有了这个能力，宿主类可以充分利用一个潜在丰富的 policy 接口，而同时还可以使用那些真正最小化的 policy 类（虽然功能上有所减少）。结合 policy 类可以公开附加功能的能力，我们就拥有了一个真正强力的机制来定制宿主类的功能。

利用模板将 policy 类和宿主类绑定起来，这样做还有其他一些明显的好处。最大的好处就是，由于绑定操作是静态完成的，所以编译程序可以生成非常优化的代码，且与手工进行的优化不相上下。而且，与传统的由虚函数组成的接口不同，policy 的接口定义更为松散。policy 类只需要在语法上遵从这个接口就可以，不需要去覆写一个准确的虚函数签名。在前面的例子中，用来实现 `APolicy` 的 policy 类可以随意地将 `Foo()` 函数定义为静态函数、虚函数，或者两种都不是。最后一点，这个方法可以很容易地扩展为多个 policy，只需要添加一些额外的模板参数，并从每个新增的 policy 类中派生出宿主类即可。实际上，只有多个 policy 才能真正发挥基于 policy 的设计模式的能量，因为仅以线性数量的代码扩张就可以实现爆炸数量的行为。

现在我们已经理解了 policy 的运行机制，可是该如何使用它们，来解决真正的设计难题呢？最好的开始方式，是对编制类所涉及的高层次的设计决策进行标识。任何可以用多种方式实现的东西，或者那些涉及折中决策的内容，都应该从类中抽取出来，委托给一个 policy。最极端的做法，是将一个宿主类所有的设计决策全部委托给 policy。在这种情况下，宿主类就变成了一个简单的“壳”，它惟一的用途就是装配、组合多个 policy class，来执行必要的任务。

当把一个类分解成若干个 policy 的时候，必须要努力得到彼此正交的 policy。正交的 policy 就是彼此独立、可以安全地修改的 policy。非正交的 policy 会让宿主类和 policy 类都变得很复杂，这样生成的类缺少类型安全性且难以使用。一种发现非正交的 policy 分解的简单方法是注意下面两种情况：当 2 个 policy 需要彼此通信时；或者更糟糕的是，合并若干个 policy 却生成一个非法的宿主类的时候。

可惜的是，本节的内容实际上只能涉及到基于 policy 的设计的皮毛。有关使用 poliy 来增强 C++类的理论和实践，鼓励大家去阅读[Alexandrescu01]和[Vandervoorde03]这两篇文章。

1.11.3 分解 Freelist

这样看来，为了实现我们的目标，开发一个可重用的、但不会影响易用性或高性能的 freelist，policy 是一个非常有希望的候选者。不过，在开始创建一个基于 policy 的 freelist 之前，首先要快速、粗略地了解一下 freelist 是如何运转的。我们感兴趣的 freelist 最开始都是先分配一个相对很大的内存块，以便满足大量的小块内存分配需求。然后，这个巨大的内存块会被分割成很多小的内存块，并放到高速缓存中，便于以后检索使用。当系统收到一个对象分配请求时，就会从可用内存块列表中取出一个内存块，并将之初始化为给定的类型来使用，然后再返回应用程序。如果列表中没有可用的空闲内存块，系统可以随意地分配一大块新的内存，并将之重新注入空闲内存块列表。最后，当应用程序为 freelist 返回一个对象时，这个对象会被转换为小的内存块，并放回到空闲块列表中，以便为将来的内存分配工作提供快速的再循环使用。了解这些流程之后，就可以开始确定哪些行为应该被分割到 policy 中。

第 1 步最好将这个 freelist 的扩容行为分离出来。有些 freelist 只是简单地预先分配一大块内存，而并不允许以后有扩容的行为。在其他一些情况下，每当 freelist 为空时，系统就会要求分配固定数量的空闲内存块。扩容 policy 让用户可以配置这个扩容行为，让他们可以相对精细地控制 freelist 分配空闲块的数量和时间。扩容 policy 还为我们提供了一个方便的钩子函数，当列表中没有空闲块时，用户就可以借此来定制相关的行为。

用来分配和释放大块连续内存（以后会被分割成独立的小块内存）的方法，最好避免使用硬编码，这是非常有意义的。有些 freelist 的实现会调用 malloc() 函数来从堆（heap）中获得一大块未初始化的内存；而另一些 freelist 会通过操作符 new() 来分配一个经过完全初始化的对象块。把这些行为都委托给一个 policy，还可以使 freelist 利用定制的内存管理器，而不必去修改实际的 freelist 类。

还有一个需要委托给 policy 的职责，是将空闲内存块转换为指定类型的对象的方式，或者反过来把特定类型的对象转换为空闲内存块的方式。如果每次分配对象或释放内存时，freelist 都完全地构造和销毁这些对象，这时候使用 freelist 就是最自然的了。但是，在某些情况下，这样的做法要付出很高的代价。定制这个行为，就可以让用户在不同级别的性能和安全性之间做出自己的选择。更常见的是，这个 policy 提供了一个很好的场所，让我们可以在应用程序使用某个对象之前和之后，完成初始化和清除这个对象的工作。

最后，我们需要一个负责保存当前可用的内存块列表的 policy。跟踪空闲内存块的方法有很多，每种方法都有各自不同的功能集、性能和内存折中方案。这个 policy 与其他 policy 略有不同，因为它要负责定义 freelist 的结构及其行为。这种将结构参数化的能力是 policy 最强大的特性之一，这也是简单的虚函数所无法做到的。

现在我们好像已经确定了这个 freelist 要使用的 4 个 policy：扩容、分配、创建和存储。但不幸的是，经过仔细的审查，我们发现分配、创建和存储这 3 个 policy 之间并没有彼此正交。某些存储空闲内存块的方法可能会影响合法创建和分配内存块的方式。有些用来存储对象的 policy 要求为某些类型的对象提供一个特定的创建 policy。这 3 个策略也正暗藏其他一些令人讨厌的依赖关系。由于基于 policy 的设计模式非常倚重于找到一系列正交的 policy，所以在继续推进之前，我们必须解决好这个问题。

最简单的解决方案是将上述 3 个 policy 的职责功能合并到一个 policy 中。从这里开始往后，我们把这个新的 policy 称为分配 policy。做出这种选择会导致我们要在其他方面做些让步。一方面，这个新的 policy 会有一个比较复杂的接口，这使得新 policy 类的编制工作更加棘手。另一方面，这个新的 policy 使得更强大、更复杂的 policy 类的实现成为可能，因为这个新的 policy 可以控制 freelist 更多的行为。将这 3 个 policy 合并起来可以实现以前无法实现的功能，这也表明这些 policy 可能永远无法正交。最后，除了把它们合并起来，避免与非正交 policy 相关的缺陷，我们也没有什么好的选择。

所以，接下来我们会设计一个有两个 policy 的 freelist。一个是扩容 policy，负责决策要分配多少个内存块，以及当列表为空时，该采取什么措施。另外一个是分配 policy，负责定义如何分配大块内存，如何将它们划分成小的内存块，以及如何将内存块转换为对象或将对象转换为内存块。

1.11.4 实现 Freelist：这是它吗？

在明确了 policy 及其职责之后，freelist 类的实际编码工作变得令人惊讶的简单明了。运用前面学到的知识，freelist 类的声明就是它自己的编码。

```
template< typename T, class GrowthPolicy,
    class AllocationPolicy >
```

```
class FreeList : public GrowthPolicy,
    public AllocationPolicy
{
...
};
```

让我们由简到繁，先来看看类构造函数。这个构造函数要向扩容 policy 请求一个需要预先分配的内存块的数量，然后让分配 policy 有机会去准备这么多的内存块。为了看得更清楚，我们省略了模板参数，其代码如下：

```
FreeList::FreeList()
{
    unsigned int numToPrealloc =
        GrowthPolicy::GetNumberToPreallocate();
    if (numToPrealloc > 0)
        AllocationPolicy::Grow(numToPrealloc);
}
```

在最基本的层面上，freelist 惟一的职责就是执行快速的对象（类型为 T）分配和释放，所以我们的 freelist 只需要两个公共成员函数：Allocate()和 Free()。下面是 Freelist 实现这两个函数的代码：

```
T* FreeList::Allocate()
{
    void* pBlock = AllocationPolicy::Pop();
    if( !pBlock )
    {
        unsigned int numAlloced =
            AllocationPolicy::GetNumAllocated();
        unsigned int growSize =
            GrowthPolicy::GetNumberToGrow(numAlloced);

        if( growSize > 0 )
        {
            AllocationPolicy::Grow(growSize);
            pBlock = AllocationPolicy::Pop();
        }
    }

     if( pBlock )
        return AllocationPolicy::Create( pBlock );
    else
        return 0;
}

FreeList::Free( T* pObject )
{
    if( !pObject )
        return;
    AllocationPolicy::Destroy( pObject );
```

```
        AllocationPolicy::Push( pObject );
    }
```

Freelist::Allocate()会从分配 policy 那里申请一个内存块。如果这个申请无法满足，它就会询问扩容 policy，以决定需要储备多少个新的内存块（如果有的话），然后通知分配 policy，在它自己的列表中添加这个数量的空闲内存块。如果仍然无法处理分配工作，那就返回 NULL，否则分配 policy 就有机会去确定将要返回的内存块是否是一个经过正确的初始化、类型为 T 的对象。Freelist::Free()只是简单地要求分配 policy 将对象转换为一个内存块，然后将这个内存块返回给分配 policy，以便将来快速地循环利用。

请注意，在所有这些函数中，Freelist 宿主类本身没有多少实际的工作，它更多地是作为一个框架，来协调其 policy 的行为。这就是一个典型的基于 policy 的设计。

前面已经讲过，这里只有两个公共成员函数，对所有的意图和目的而言，这基本上是真实的。不过，这里还使用了几个模板成员函数，它们使用的是几个任意类型的参量，并会将这些参量传递给 AllocationPolicy::Create()。正如我们后面会看到的，这些函数的使用是为了使 freelist 更自然，使用起来更安全。这里还有一个模板构造函数，它会将惟一的 template 参量传递给 GrowthPolicy 的构造函数。使用这个函数，可以在运行时很方便地配置扩容 policy。多亏了不完全实例化这个功能，编译程序只会在它们被使用时才会去编译这些函数，这使得 policy 类和用户可以放心地忽略这些函数，直到真正用到它们。

这就是定义我们的 freelist 以及它所依赖的 policy 接口所需要的全部代码。一旦明确了 policy，freelist 的实现就变得非常直接和机械。

1.11.5 选择最佳的 policy

定义好 freelist 并固化好 policy 接口后，剩下的工作就是要提供几个具体的 policy 类。

首先看一下相对简单的扩容 policy。它的接口只包含两个函数：GetNumberToPreallocate()和 GetNumberToGrow()。下面就是一个实现该 policy 的简单、但却高效的类

```
struct ConstantGrowth
{
    ConstantGrowth( int pre = 16, int grow = 16 )
        : preAllocate( pre ), numToGrow( grow )
    {}
protected:
    int GetNumberToPreallocate() const
    {
        return preAllocate;
    }
    int GetNumberToAllocate( int unused ) const
    {
        return numToGrow;
    }
private:
    int preAllocate, numToGrow;
};
```

这是一个 Freelist 可以马上使用的类。创建新的 policy 类，为宿主类提供全新的、定制的功能就是这么容易。不需要什么神秘的编程窍门，也不需要什么特殊的 C++技艺。

分配 policy 的接口类稍微有点复杂。它要包含如下函数：把可容纳特定类型对象的内存块推入列表或从列表中取出的函数，将内存块转换为特定类型的对象以及将特定类型的对象转换为内存块的函数，以及一个对列表进行扩容的函数。这样一个实现此接口的 policy 类是 PlacementNewEmbeddedLink。我们先来看看这个类的声明及其数据成员，然后再了解一下它的工作原理。

```
template< typename T > class PlacementNewEmbeddedLink
{
public:
    ...
private:
    struct FreeBlock
    {
        FreeBlock* pNext;
    };
    FreeBlock* pFreeBlocks;
    std::vector< void* > chunks;
};
```

这个特殊的分配 policy 会分配若干块连续的内存，并将它们划分成小的内存块，其大小正好可以容纳一个类型为 T 的对象。每个内存块最前面的 4 个字节用来保存指向下一个可用内存块的指针，且每个内存块都会被推入到一个已链接列表的头部。Grow() 函数负责实现这个功能。

```
void PlacementNewEmbeddedLink::Grow( int numBlocks )
{
    void* pChunk = malloc( numBlocks * sizeof(T) );
    chunks.push_back( pChunk );
    for( int ix = 0; ix < numBlocks; ++ix )
        Push( (char*)pChunk + ix * sizeof(T) );
}
```

Push() 和 Pop() 这两个函数负责维护这些简单的链接，并负责将内存块添加到该列表的头部，或者从列表的头部取出内存块。它们的具体实现和大家想像的一样容易。

```
void PlacementNewEmbeddedLink::Push( void* pBlock )
{
    FreeBlock* pNewHead = (FreeBlock*)pBlock;
    pNewHead->pNext = pFreeBlocks;
    pFreeBlocks = pNewHead;
}

void* PlacementNewEmbeddedLink::Pop()
{
    if( !pFreeBlocks ) return 0;
    void* pNewBlock = pFreeBlocks;
    pFreeBlocks = pFreeBlocks->pNext;
```

```
        return pNewBlock;
    }
```

在每个内存块的头部保存一个指向下一个可用内存块的指针，这样做的主要好处是省去每个内存块的额外开销。与很多通用内存管理器涉及的开销相比，此举省下来的开销总计可达几百 KB。其明显的缺陷就是，原先保存在内存块头部中的数据会被这个指针覆盖掉，该数据有可能会是一个虚函数表。这个 policy 类的另外两个函数 Create() 和 Destroy() 完善地解决了这个难题，并且可以确保将原始内存转换成一个完全成熟的对象。

```
static T* PlacementNewEmbeddedLink::Create( void* pBlock )
{
    return new( pBlock ) T;
}

static void PlacementNewEmbeddedLink::Destroy( T* pObject )
{
    pObject->~T();
}
```

Create() 函数使用了 placement new（指定位置创建）操作符，它会向编译程序发出指令，在指定的内存地址创建一个构造完整的类型为 T 的对象。由于该函数并没有对内存的内容做任何假设，这就隐含地解决了可用空闲块指针会覆盖潜在的重要数据的问题。

更为重要的是，这使得那些配备了这个 policy 的 freelist 更为安全且更易使用。由于 placement new 会调用该对象的构造函数，所以这个 policy 的 Create 函数会很自然地模仿系统调用操作符 new() 来创建对象时的对象构造方式。作为一个额外的收获，定义 Create() 函数的模板化重载函数（这些重载函数可以将参数传递给类构造函数）也变得相对不那么重要了。当然了，与构造函数的初始化工作对应的是析构函数的清理工作，这个是由 Destroy() 函数来实现的。

检查用这个 policy 配置的 Freelist，就会发现，我们已经胜利地实现了所有的目标。分配和释放对象当然是很快的，因为它只包含几个指针操作及对构造函数或析构函数的调用。而且，每次内存分配的额外开销是 0。但最重要的是，使用构造函数和析构函数自动地完成对象的初始化和清理工作，使我们可以用 freelist 安全、自然地替代对操作符 new() 和操作符 delete() 的调用。

1.11.6 可能性

如果这就是故事的结尾，那么之前我们把 freelist 分解成几个 policy 的辛苦就算白费了。如果前面提到的 policy 总是能够解决所有的问题，那可就太棒了，但可惜它们还做不到。为了说明基于 policy 的设计模式把 freelist 类变得有多么万能，下面提供了 4 种独立的分配 policy，它们有各自不同的行为和相关的折中方案。

1. PlacementNewEmbeddedLink

这就是前面讲的分配 policy。它集众多优点于一身：高性能、每个内存块的额外开销为零、类型安全、易于使用，这使得它成为一个万能的选择。正是由于有如此多的积极因素，

它才成为配置 Freelist 时默认使用的分配 policy。但是，它也有一些局限性。其中之一，就是配备了这个 policy 的 freelist 从来不会把内存返回给全局堆（Global heap），也不会以任何方式来共享它们的空闲内存块。根据所采用的分配模式，大量的内存会以空闲内存块的形式一直闲置在 freelist 中，而无法用在其他地方。而且，在一些性能极其关键的领域，PlacementNewEmbeddedLink 也许并不适合那些带有大开销的构造函数和析构函数的类。但是，由于其易用性，它还是成为了缺省的分配 policy。

2. ConstructOnceStack

这个分配 policy 可以和[Glinker04]中实现的那个 freelist 的设计和行为很好地进行匹配。本文一开始讲到的关于那个 freelist 实现的优点和不足都可以落实在 ConstructOnceStack 的编码中。我们的 freelist 可以轻松、完美地效仿这样一个不同的 freelist 实现，这正是 policy 的能力的具体体现。这个 policy 甚至还公开了先前那篇文章中 freelist 实现所提供的一个额外函数——FreeAll()，它使得整个“模仿秀”更加完整。

3. CompactableChunkPolicy

有一个不太寻常的分配 policy 类：CompactableChunkPolicy。这个 policy 的行为和 PlacementNewEmbeddedLink 很类似，但是稍有变化。在这个 policy 中，每个大块内存都维护着一个计数器，记录它有多少个空闲块被使用了。如果 2 个大块内存都没有空闲块被应用程序使用，这个 policy 就会将两个块中更大的一个返回到堆中。这个 policy 的问题是，它有一个开销比较大的释放操作。如果对象的分配和释放的爆发性出现相对不是很频繁，或者很多 freelist 会被实例化，但在任何给定的时间都有很多 freelist 为空，此类情况下，这个 policy 就能发挥自己的长处了。

4. SharedChunkPolicy

最后也是最不寻常的一个分配 policy，是随书光盘中所带的 SharedChunk-Policy。在内部实现中，这个 policy 类的所有实例都共享一组静态的 freelist。这就意味着配置了这个 policy 且管理着相近大小的对象的两个独立的 freelist 会共享内存块。如果一个应用程序使用的是大量很少会接近峰值使用率的 freelist，那就可以大大减少 freelist 中的闲置内存数量。这个 policy 的主要问题是，从一个 freelist 分配出来的几个对象无法确保在内存中的位置是临近的，这就降低了访问局部性。另外，根据这个 policy 的配置方法，每个对象上还会有少量的潜在开销。一些通用的内存管理器实际上就是用这种方式在内部操作的，且 STLPort 中默认的分配器差不多也是这样运做的。

随书光盘中的所有分配 policy 都公开了两个额外的成员函数：GetNumBlocksInUse()和 GetPeakBlocksInUse()。在开发过程中，可以使用这两个函数来调整扩容 policy，获得最佳的内存使用。光盘上还提供了几个扩容 policy，其中一个可以扩容 1 倍，有一个可以线性地扩容至固定的最大容量，还有一个可以确保每个临近的内存块刚好适合一个内存分页。

当然了，如果这些 policy 都无法满足你特定的要求，改变它们的行为和另外编写一个 policy 类都一样简单，而且这么做根本不需要改动 `FreeList` 类的任何代码。这就是基于 policy 的设计的真正美妙之处。

1.11.7 总结

本文的目标是开发一个快速的、易于使用的，足够灵活，能在尽可能多的情况下使用，且不需要做出什么折中的 freelist 类。为了实现这些目标，我们开发了一个带有简单、可靠的接口，但具有开放式行为的 freelist。最后，我们并没有满足于一个单一的 freelist，而是提供了一个可配置程序高的 freelist 框架。我们提供了 4 个（可不是 1 个）不同的实现，每个实现都提供了不同的折中方案。即使这 4 个实现你都不喜欢，也没关系，因为你不必改变本文提供的代码就可以很容易地提供一个符合自己特殊要求的实现，更重要的是，连接口和内部设计都不需要改变。

基于 policy 的设计是我们达成目标的关键。即使根本不会用到这个 freelist 类，但是通过阅读本文，希望大家可以看到 policy 所具有的能力。如果以后要开发灵活的、可配置的类，甚至是完整的类库，这个技术都会是无上之选。policy 是一个工具，每一位想获得可靠的、可重用的代码的开发人员，都应该把它加到自己的工具箱中。

1.11.8 参考文献

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

[GoF95] Gamma, Erich, et al. *Design Patterns*. Addison Wesley, 1995.

[Glinker04] Glinker, Paul. "Fight Memory Fragmentation with Templated Freelists." In *Game Programming Gems 4*. Charles River Media, 2004.

[STLPort04] STLPort Web page. Available online at *www.stlport.org*. September 20, 2004.

[Vandevoorde03] Vandevoorde, David, and Nicolai M. Josuttis. *C++ Templates: The Complete Guide*. Addison Wesley, 2003.

1.12 实时远程调试信息日志生成器

Microids Canada 公司，Patrick Duquette
gizmo@gizz-moo.com

在过去的几年中，我们已经看到，业界对在屏幕上显示的、游戏内部的调试面板的兴趣日益浓厚。虽然它们确实是很棒的，能正常地完成工作，但是它们也有着把游戏屏幕搞得乱七八糟的奇怪能力。当我们想看到当前帧率（每秒的帧数，FPS）计数器以外的更多信息时，下场总是丧失好大一块屏幕显示空间。在这一点上，游戏机游戏最为糟糕，因为它们的屏幕分辨率首先不是非常的高；而为了能够看清楚，我们不得不使用大号字体。所以，一个 640×480 的屏幕上很难在显示太多信息的同时还能看到这些文字背后的游戏画面。

当然了，你可能会说，我们还有颇受信任的 `OutputDebugString()` 函数啊。但是，由于它单一的输出显示窗格和滚动列表式的显示风格，用它来显示频繁变化的值的实时日志信息，简直就是噩梦一场。假设我们在一个项目中，使用了输出显示窗口来定期地显示调试信息，信息确实是显示在那里了，但你别指望轻松地找到想看的信息。这根本无法满足我们对一个多产的调试会话过程的需要。

本文将跳过日志文件的话题，因为我们要把重点放在一个调试信息的实时监控解决方案上。

1.12.1 对标准化的调试日志的需求

由于现在的游戏项目通常都需要 20～30 个全职的程序员，所以用一种标准的方式来管理和操控游戏内部的调试信息是非常重要的。如果数据的显示格式很不方便阅读，那么这些信息几乎没什么用处。同样的道理，不实用的数据日志生成过程也是没什么用处的。如果在生成调试数据日志之前必须要费尽周折，那么人们是不太可能使用它的。

调试数据的显示应该简明扼要，更重要的是，它应该以一种可计量的方式来显示。让那些无关的数字在我们眼前不停地滚动，不但会让我们头痛不已，而且如果不是该日志功能的开发者，我们可能根本看不到什么调试日志，因为那些数字对我们来说没什么意义。

我们应该把调试数据组织整理到相关的会话或页面中。如果像在使用 `OutputDebugString` 时的输出显示窗口中那样，把很多无关数据放在一个单一的页面中，我们就不得不在大量无关的数据行中搜索有关的信息。

这样不但费时，而且还可能让我们漏掉真正要找的信息。

而且，调试信息分页不应该采用硬编码来实现。虽然采用硬编码来实现调试分页信息确实很诱人，但我们还是应该抑制这种行为。在设计调试日志生成器时，我们根本无法预见每个调试数据的类型，因此我们应该为显示页面的动态生成做好准备。当然了，有些显示页面肯定会在所有的项目中出现，但是通过让最终用户指定每个页面的属性，我们就可以在实用的基础上提供一个方便的解决方案。

1.12.2　数据表示：你可看到我所看到的

数据表示是最重要的事情之一。确保最终用户（这些最终用户可能是程序员、美工或者技术人员）能够按照我们的意图去理解结果数据，这是至关重要的。如果我们需要得到某个变量的值在一段时间内的精确日志，滚动式的日志生成器也是一种很好的方案，但是如果只需要看到变量在一段时间内的情况，那么直观的图表可能是更好的方案。对于那些只需要显示当前值的变量，与其显示最近的 500 个值，不如就用一行信息来显示它的值，并根据需求对其进行修改。

完美的情况下，调试程序的运行速度不应该因为日志生成过程而降低。当然，这是不可能的（因为我们确实要搜集并发送很多的调试信息），但是我们应该尽量降低 CPU 和内存的消耗。可能的话，应该使用异步函数来发送数据。如果调试日志生成器没有运行，我们也不应该执行调试信息的搜集工作。这样，在需要的时候，我们还可以暂停日志生成器的运行。

被调试的游戏可能会涉及各种不同的运行平台，我们的日志生成解决方案必须对所有这些平台提供同样的特性支持。用一种标准的方式来生成调试信息日志，这样不管运行游戏的具体平台是什么，调试信息的最终用户都不用去熟悉各种不同的界面，从而减轻了学习压力。

1.12.3　本文提议的解决方案

这里带给大家的解决方案非常简单，基本上就是一个客户端/服务器端结构。其中，调试程序是客户端，而跨平台的游戏模块就是服务器端。也可以反过来，把被调试的游戏模块作为客户端，但是这就迫使我们不得不告诉这个游戏模块，什么时候该连接到什么地方。把游戏模块作为服务器时，我们可以从任何一个机站，在游戏运行中的任意时刻与之连接。

那么数据是如何表示的呢？我们已经知道，调试数据必须进行分类处理，以便帮助我们尽快找到想要的信息。但是，在那些分类中，各个数据段是如何显示的呢？根据不同的数据类型和所需要的可视化效果，有这样几种可能：

滚动显示：标准的数据显示方法，数据一个接一个地追加进去；

只显示当前值：仅仅显示最新的值；

图表：把某个变量在一段时间内的数值变化快速地、可视化地显示出来。

1. 滚动显示

通过一个标准的 `ListBox`（列表框）显示所有的数据，每个数据项对应一个显示行。新

的数据添加在列表的尾部。不需要对显示的数据项进行排序，因为这会降低数据插入过程的速度。

为了提高数据插入过程的速度，我们会通过 InitStorageListBox 函数预先分配一定数量的列表行。我们需要显示一个上下文菜单，来帮助管理 ListBox 的内容：

清除列表（Clear list）：清除 ListBox 中的所有项目；

拷贝（Copy）：将选定的数据行的内容复制到剪贴板中；

拷贝所有（Copy all）：将整个 ListBox 的内容复制到剪贴板中；

保存（Save）：将列表的内容保存到一个文件中。

2. 当前值显示

这种显示方法是最简单的，它是通过一个只读的 EditBox（编辑框）来实现的。使用 EditBox 而不是一个静态的控件，这样就可以使用标准的 EditBox 上下文菜单，对选中的文本进行复制和粘贴操作。

3. 图表显示

图表可能是一种让你把游戏中发生的事情尽收眼底的最快速、最直观的数据显示方法。当接收到绘图数据时，首先要做的是查出那些越界的数据。有两个可能的解决方案可以处理这件事情：让用户自己修改数据范围；或者当有数据越界时，适当地延长绘图数据的收集时间，调整收集到的数据，让 Graph 函数自己进行修正。同时使用这两种策略也是非常方便的，因为用手工方式重新设置 Graph 的范围，可以让用户看到以前他认为合适的数据。

这个日志生成器还可以集成很多不同类型的 Graph，但是要提前开发一个系统，用于扩展图表库。图 1.12.1 中显示的就是一个图表的例子。随书光盘中提供的样本代码只实现了一种类型的 Graph，但是在作者的网站上[Gizz04]，可以找到这个代码的改进版本。

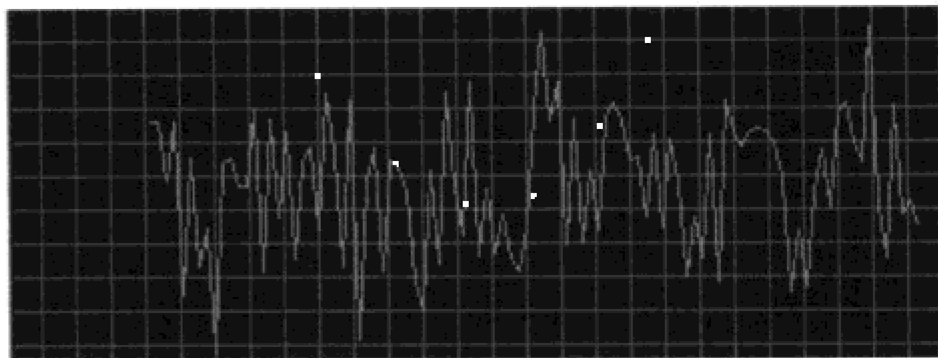


图 1.12.1 调试日志生成器中的一个图表

分类并整理好调试数据后，下面的重点就是如何让调试日志生成器在同一个页面上显示不同类型的数据。在同一个分类中，对于不同类型的调试信息，其最佳的数据显示方法也是不一样的。

4. 组装

在一个基于对话框的窗口中，TabControl 负责管理信息的显示。系统会根据被调试的游戏发回的信息，动态地创建 TabPages。

图 1.12.2 显示的是一个 TabControl 和 TabPages 的例子。

MyListPage　MyValuePage

Name	Value
1st variable	123
2nd variable	weds
3rd variable	125.12

图 1.12.2　TabControl 和 TabPage 示例

在启动调试日志生成器之后，系统就会尝试与要调试的游戏进行连接。一旦连接成功，被调试的游戏就会把想要在生成器中注册的 TabPages 的名字，发送给调试日志生成器。这些名字都有各自相应的索引，在以后的通信中只会发送这些索引。然后，被调试的游戏会开始发送数据包，内容包括：想要生成日志的 TabPages 的名字列表，以及应该使用的显示类型等。如果需要的话，还会包括一些特定的数据（如 Graph 显示要用的数据范围）。

一旦连接成功并初始化完毕，调试日志生成器就会开始接收数据。每个数据包都带有 TabPage 索引、ValueName 索引和实际的数据。经过一个快速的查找，我们可以准确地知道在某个特定的页面中，哪些部分需要更新。数据显示类则负责按照预计的方式来显示数据。

1.12.4　游戏日志模块

LogModule 是在一个单例对象中实现的。这让我们可以安全地访问一个全局可用对象。创建好 LogModule 后，它就对平台相关网络库进行初始化。

TabPages 和日志变量的注册是通过一个静态函数来完成的，该函数是在 LogModule 的初始化阶段中被调用的。我们用枚举类型作为 TabPages 和日志变量的索引。变量的名字要在调试日志生成器中使用，所以我们要用一个宏来完成变量名字中枚举成员的转换。

在示范代码中，当 LogModule 被认成是一个服务器时，它每次只能接受一个连接。我们并不打算同时注册多个日志生成器，来获取 LogModule 的调试信息。直到当前的连接关闭，LogModule 才会开始接受其他进来的连接。

由于我们的目标是设计一个 CPU 使用率很低的日志系统，所以通信部分是通过异步网络函数来处理的。这部分代码和网络初始化部分的代码就是与平台相关的两段代码。

日志的生成

log 函数是一个可变参数的函数，它为使用调试日志生成系统的程序员提供了一个易于使用的平台。调试信息不需要格式化，就可以直接传递给 log 函数。这样的话，就像以前使用 OutputDebugString 函数时那样，我们仍然可以向日志生成器发送字符串，只是比以前要更加灵活了。

LogModule 全部的调试日志代码，都会用一个预处理命令的条件编译指令#ifdef 包装起来，其目的是为了在不想要调试日志信息的时候，可以很容易地关闭调试信息。对 log

函数的调用是通过一个宏定义来完成的。这样，游戏代码中就不需要再使用#ifdef 来包装了，因为当日志系统没有包含（#include）在某个游戏版本中时，这些语句就没有任何意义了。

1.12.5 可能的改进和扩展

对于这个调试信息日志生成器，我们还可以对它进行如下的改进或扩展：

- 滚动显示方式应该可以限定它显示的行数。设置一个全局变量，如果用户需要，可以在每个列表框中覆写这个变量，这也许是个正确的做法。
- 对于一个给定的调试信息，让用户可以切换它的显示类型。
- 让用户可以修改 Graph 的有效数据范围。
- 当发现数据越界时，让 Graph 函数可以在一定的时间之内自行修正它的范围。
- 以页为单位，或者在每次收到了少量信息的时候，自动形成日志文件。每页的日志文件应该是不同的文件。
- 允许多个调试日志生成器同时注册到 LogModule。

1.12.6 总结

本文向大家展示了一个简单、可扩展、但却非常高效的方法，来显示调试信息，同时还不会把游戏显示屏幕搞乱。虽然在一个项目的工具优先级列表中，调试日志生成器的位置并不靠前，但是在工期临近，大家不分昼夜赶工的时候，一个优秀的调试信息日志生成器或查看器，可以帮助我们迅速地发现各种游戏特性中隐藏的 bug，救大家于水火。

1.12.7 参考文献

[Gizz04] 笔者的网站, *http://www.gizz-moo.com*.

1.13 透明的类的保存和加载技巧

Patrick Meehan
gems@tenaciousgames.com

为了将加载时间最小化，一个众所周知的方法是将结构化的数据保存在相邻的内存块中，以便在加载时可以即时使用[Olsen00]。该方法有一个局限性，就是相关的指针（包括虚拟函数表指针）无法保留，这使得某些复杂的数据结构或者包含虚函数的类难以保存和恢复。

本文的第 1 部分介绍了一个简单的方法来保留用户定义的指针，另外还介绍了一个安全恢复虚函数表指针的小窍门。这就意味着，我们可以随意地使用虚函数、类和指针来描述游戏数据，且不需要冗长地实现每个类的方法就可以把它们序列化或者恢复它们。

本文的第 2 部分设计、实现了一个 API 的范例。该实现提供了透明类的保存和加载功能，而且对最终用户的限制非常少。本文详细地讨论了该实现的细节内容，希望读者可以对它进行改进和扩展。

1.13.1 小窍门

这个小窍门归结起来，就是直接使用指针运算和 placement new 操作符。

1. 保存和恢复用户定义的指针

如果不想在运行时处理相对偏移量，就可以在加载的时候对它们进行解析，其方法是创建一个指针的二次映射（或重映射，remap）表，并把这个表保存在文件的尾部。这个表中的每一项都包含着需要重新映射的指针的偏移量，以及该指针重新映射后的偏移量。由于这两个偏移量都是相对于文件的尾部，所以只要再加上文件的目的地地址，就可以解析这些指针。

2. 保存和恢复虚函数表指针

当在类中声明一个虚函数时，编译程序就会在为这个类分配的内存中添加一个虚函数表指针。当该类构造完成时，相应的虚函数表的地址就会保存到这个指针中。但不幸的是，这个虚函数表指针只在其创建会话中才是合法的。如果把一个带有虚函数表指针的类保存在文件中，那么在下一个会话中重新加载该文件的时候，那个指针可能已经变成了垃圾。

所以，简而言之，这个问题就是：当该类在前一个会话中被构造出来

的时候，我们怎么做才能安全地恢复这个类的虚函数表指针呢？

问题的答案就在于 **placement new** 操作符。placement new 操作符会使用用户指定的内存来“适当地”构造一个对象。与 new 操作符不同，placement new 操作符并不分配任何内存。

```
char rawMem [ sizeof ( Foo )]; //容纳一个 Foo 的裸内存。
Foo* foo = new ( rawMem ) Foo (); //构造一个适当的 Foo。
```

如果想调用该类的析构函数，那就不得不进行手工操作：

```
foo->Foo::~Foo(); //显式调用析构函数。
```

调用 placement new 操作符，并在指定的内存（这部分内存已经被一个同类型的对象所占用）中构造一个对象，就可以确保该对象的虚函数表指针是合法的。如果对构造函数进行空调用，我们就可以恢复在前一个会话中保存的对象。让构造函数为一个空函数，就可以避免在加载时进行的重新初始化工作。另外一个替代方案，是为 placement new 恢复操作创建一个独立专用的构造函数。

1.13.2 FreezeMgr 的实现

本节，我们从背景知识中跳出来，详细地了解一下随书光盘中提供的那个实现的设计过程。假设你是一名特别乐于为团队中的其他成员提供解决方案的系统工程师或引擎程序员。本文所指的用户是会使用（但可能不会改变）这个实现的游戏程序员或工具开发人员。

希望大家可以仔细阅读这些讲解，并对这个例子进行试验，然后编写自己的实现或修改这个实现以满足自己的实际需求。本文讨论的例子是一个实现的有限版本，为了专注起见，它只结合了部分特性，而跳过了其他一些特性。

1. 使用模型

我们想让最终用户可以按照自己喜欢的顺序来创建游戏数据。这些数据包括类、结构和数组（复合类型和简单类型）。我们还希望用户的指针能在加载时自动恢复。

我们需要一种跟踪用户创建的数据，并能提供一个接口来保存和恢复数据的方法。为此，我们会使用一个单例类，其名称为 FreezeMgr（冷冻管理器）。打个比方就是，不管用户创建的是什么数据，其在自然状态下都是冷冻的，它可以随时解冻并在以后再次使用，这一过程无须额外的准备工作。

因此，FreezeMgr 的关键操作就是冷冻和解冻——Freeze() 和 Thaw()。由于 FreezeMgr 必须跟踪用户创建的数据，所以它的接口也要包括分配内存和释放内存的方法。

由于使用了单例设计模式，所以这个例子的单例接口会采用最基本的方式来实现。参考 [Bilas00]，可以找到更为复杂的实现方法。

从最终用户的角度来看，这个过程应该是：

- 使用 FreezeMgr 创建游戏数据。

- 用任何合适的方式初始化这个数据。
- 命令 FreezeMgr 把所有相关的数据打包到一个文件中，并进行保存，以便优化重载操作。

2. 跟踪用户分配的内存

每当用户要求 FreezeMgr 去创建什么东西的时候，FreezeMgr 就会分配一个内存块，并为这个内存块保存一个内部记录。这个记录包括了该内存块的大小和地址。我们提供用下列方法来分配内存：

```
template < typename TYPE >
TYPE* FreezeMgr::AllocTyped ( u32 elements );
```

其中，AllocTyped() 是一个模板方法，它会使用 malloc() 和 placement new 操作符创建一个用户指定的类型（或类型数组），并调用相应的构造函数。我们使用了 malloc() 而不是 new()，因为我们想要精确地控制构造这个类的时间，并且要独立于其内存分配的时间。

用来跟踪内存分配的记录被称为 MemBlock。FreezeMgr 会把 MemBlocks 保存在一个地址映射中。MemBlock 包含一个检测给定的地址是否属于自己的范围的方法。该方法是：

```
bool MemBlock::ContainsAddress ( void* addr ) const;
```

反过来，FreezeMgr 也包含一个方法。对于给定的任意一个地址，该方法会返回包含该地址的 MemBlock，应该存在这样一个 MemBlock：

```
MemBlock* FreezeMgr::FindContainingMemBlock ( void* addr ) const;
```

3. 跟踪用户声明的指针

在一个名为 FreezePtr 的模板类的帮助下，我们就可以跟踪用户声明的指针。它的模板参数是一个指针类型，惟一的数据成员就是这个指针自己。

在构造过程中，FreezePtr 会尝试向 FreezeMgr 注册自己。如果 FreezePtr 属于某个 MemBlock 的范围，系统就在那个 MemBlock 中保存它的一个记录，否则系统就会忽略掉这个 FreezePtr。

判断一个 FreezePtr 是否位于某个 MemBlock 的范围之内，只需要调用 FreezeMgr::Find-ContainingMemBlock()。

4. 创建一个文件

在这一阶段，我们可以实现两个目标。

- 将用户数据保存到相邻的内存块中，以便飞速加载。
- 在加载时读取那些内存块中的内容，并修正所有的指针。这样，用户就不用去摆弄那些相对偏移量了。

由此可见，只要不包含虚函数，类和结构就可以“开箱即用”（下一节会讨论虚函数表）。

在为用户打包一个文件之前，我们必须考虑清楚，一旦重新载入内存，用户要如何访问这些数据。也许，用户想通过某些自己定义的类或结构来访问数据。也许，用户会把文件看

成是好像有一些根类型，他们可以通过这些根类型来访问数据。例如，如果该文件表示的是一个游戏关卡，它的类型可能是用户定义的类 GameLevel，这个类包含一些指针（当然应该是 FreezePtr），指向的可能是出怪点、碰撞几何体、节点图表，等等。

让用户指定一个根类型的决定其实暗示了打包文件应该使用的方法。用户会把一个根类型的类作为参数，来调用 FreezeMgr::Freeze()。FreezeMgr 会找到包含该类的 MemBlock，并把它打包到用户文件中。然后，FreezeMgr 会通过遍历每个 MemBlock 所知的 FreezePtrs，递归地对其他和根类型相关的 MemBlocks 重复这个过程。

在遍历由 FreezePtr 连接构成的 MemBlock 网络时，要做下面几件事情：

- 把遇到的每一个 MemBlock 添加到内存块列表中。Freeze() 中会包含这个列表。
- 计算该内存块在相邻的存档内存空间中的位置。
- 计算每一个 FreezePtr 的值，作为相邻内存空间中的偏移量。
- 把每个 FreezePtr 添加到指针列表中。Freeze() 中会包含这个列表。

在这个过程中，我们假设每个 FreezePtr 都指向一个由 MemBlock 管理的合法内存，或者指向 NULL。换句话说，如果指针不为 NULL，那么每次调用 FreezeMgr::FindContainingMemBlock() 都应该返回一个合法的 MemBlock。

一旦创建了 MemBlock 的列表，就遍历这个列表，并把所有的内存块打包到文件中。然后，就可以写出指针表。这个表中的每一项都包含这指针的偏移量，以及指针内容的偏移量。

正如我们想要的那样，解冻文件的过程简单而又快捷。

加载一个文件时，在把它读进来之前，首先要分配一大块足以容纳那些相邻内存块的内存。然后，通过扫描指针的二次映射表并执行指针运算，就完成了对指针的修正。

请注意，指针对它们二次映射的顺序并不敏感，所以在输出结果之前，可以对二次映射表进行排序，以便最大化地提高性能。

5. 恢复虚函数表指针

正如前面讨论过的，我们可以使用 placement new 操作符来恢复虚函数表指针。因此，FreezeMgr 也是一个可以查看某个类的实例并使用 placement new 操作符将其恢复的工厂类（factory class）。也可以使用离散工厂类，为了讨论的方便，我们要将这个功能打包到 FreezeMgr 中。

为了让这个实现可以正常工作，所有包含虚函数的类必须继承一个用来识别它的抽象基类。该类只有一个数据成员，一个 32 位的类型识别码。我们将调用基类 Freezable，尽管“Freezable”这个名字多少有些用词不当，因为这个类只能用来描述那些可以被冷冻的东西的一个子集。

在分配内存时，Freezable 可以作为一个独立完整的类，也可以嵌入到其他的类中。和 FreezePtr 一样，Freezable 也是由包含它们的 MemBlock 来跟踪管理的。

为了在加载时构建 Freezable，FreezeMgr 维护着一个抽象 creator 类的内表。每个类型的 Freezable 都必须有它自己的具体 creator 类。我们把这个抽象 creator 类命名为 AbstractCreator。我们提供了一个名为 ConcreteCreator<>的模板类，这样用户就不必再为每个 Freezable 都编写一个 creator 了。注册一个用户定义的类所使用的方法同样也是一个模板：

```
template <typename TYPE>
void FreezeMgr::RegisterFreezableType(void);
```

这个函数声明了一个静态的ConcreteCreator<TYPE>类，并把它分配给AbstractCreator表中下一个可用表项。随后，这个表项的索引就变成了上面那个类的 32 位识别码。我们可以使用一个映射，把用户定义的识别码转换为相应的表项，但还有更高效的方法，那就是在加载时刻检查每个 Freezable 的类型的时候，直接索引到 AbstractCreator 表中。作为折中，如果 FreezeMgr 创建的有些数据在每个会话中都维持在合法状态，那么它们的类型注册顺序就必须像添加新类型那样来对待。

为了确保能够把正确的 32 位类型识别码映射到一个给定的 Freezable 实例上，我们要求用户在 Freezable 中重载一个纯虚方法：

```
char const* Freezable::GetClassName(void) const;
```

为了算出给定类的识别码，FreezeMgr 维护着一个从类的名字到识别码的映射表。通过调用 GetClassName()，并使用这个名字查看是否给该类指定了一个识别码，任何一个 Freezable 类都可以确定自己的识别码。

除此以外，业界还有其他很多可靠、有效的方式来处理运行时的类型信息[Wakeling01]。大家应该研究一下这些方法，然后用它们来替代这里使用的方法。

由于在解冻的时候，系统会调用 Freezable 的构造函数，所以用户一定要小心，千万不能重新初始化它的成员。例如，如果在构造的时候，某个 Freezable 把它的成员初始化为 NULL，那么在解冻过程中，这些成员也会被重新初始化为 NULL。

通过提供一个替代的构造函数，只在解冻过程中由 placement new 操作符来调用，就可以解决重新初始化的问题。但新的问题又出现了，我们无法将这个方法拓展到 Freezable 的所有成员中，除非用户为每个成员都实现一个替代的构造函数，并在整个构造函数链中建立对这个函数的显式调用。我们有可能会因此搞出一大堆空的构造函数，这可就麻烦了。一个更为实际的方法，是在一开始就向用户公开这个问题（这样，用户就会意识到这个问题），并要求用户去调用 FreeMgr::IsThawing()，以检测是否存在重新初始化的问题。

我们回顾一下系统对最终用户的一些限制：

- 用户必须从 Freezable 派生得到所有包含虚函数的类。
- 用户必须重载 GetClassName() 函数。
- 在使用 FreezeMgr 之前，用户必须声明所有的 Freezable 类型。
- 用户绝对不能改动 Freezable 类型的声明顺序。
- 用户在 Freezable 的构造函数或其成员类的构造函数中应该什么都不做；或者应该检测是否有一个 thaw（解冻）操作正在执行中，如果没有，才能执行初始化操作。

若要管理并冷冻用户的类，应该：

- 在创建 Freezable 的时候就开始跟踪它们。
- 先对所有的 MemBlock 和 FreezePtr 进行重新映射，然后再重新映射 Freezable。
- 写出 FreezePtr 的二次映射表。
- 写出 Freezable 的表。这个表只包含每个 Freezable 的相对偏移量。

若要从文件中解冻，应该：

- 读入那些相邻的数据块。
- 修正所有的指针。
- 找到内存中的每一个 Freezable，然后判断得出它们的 32 位类型识别码。
- 用类型识别码作为索引，来检索 AbstractCreator 表。
- 将 Freezable 传递给 AbstractCreator::Construct()。

1.13.3 其他几个特性

对于它的黄金时间，FreezeMgr 还没有完全准备好，它还要再添加一些特性，才能使用。

1. 序列化数据

在有些情况下，我们希望把部分数据当作序列化的流来处理。为了支持这个特性，我们在 Freezable 中增加了 2 个虚方法：

```
virtual void Freezable::SerializeIn ( FILE* file );
virtual void Freezable::SerializeOut ( FILE* file );
```

通过重载这两个方法，用户就可以在 Freezable 类中添加序列化的数据。正如大家所期待的，在 freeze（冷冻）操作中会调用 SerializeOut()，在 thaw(解冻)操作中会调用 SerializeIn()。在这两种情况下，用户都可以直接控制文件。FreezeMgr 会记住 SerializeOut() 函数写入了多少个字节的内容（如果确实写入了），因此 SerializIn() 也必须读入同样字节数的内容。

2. 以编辑模式加载

到目前为止，我们一直都这样假设：用户是在一个会话中创建和修改游戏数据，然后以只读模式加载这些数据，在另外一个会话中使用。

如果要实现一个编辑或读写模式的用户选项，就需要在加载的时候重新创建 FreezeMgr 的内表，包括内存块表和指针重映射表。一旦实现了这个特性，用户就可以随意地增加或修改数据，然后再把它重新冷冻。

现在，很多游戏都可以让玩家随时保存他们的游戏进程，FreezeMgr 的这个扩展与实现这个特性的方案正好呼应。

3. 文件引用

可以让用户选择以下列方式来初始化 FreezePtr，这样它们就会在加载时分解到文件中：

```
mFoo.PointAtFile ("bar.bin" ); // mFoo 是一个 FreezePtr
```

当加载一个含有 mFoo 的文件时，资源管理程序会自动地加载“bar.bin”文件，并让 mFoo 指向“bar.bin”文件。要想将这个特性在一个独立的示范程序中实现，就需要关联一个资源管理程序[Boer00]，或者实现一个回调接口，这两种方法它们都超出了本文的范畴。但不管怎么说，这都是一个很有用的特性，大家应该考虑实现这个特性。

1.13.4 如何使用范例

大家可以原封不动地使用本书提供的范例，但是应该尝试对它进行修改，以便满足自己的需求。该范例是作者为本文专门编写的一个工具的独立版本，这里照原样提供给大家。我们不对商业应用提供担保，也不打算支持商业应用，所以建议大家好好熟悉这个系统，了解它的不足，然后创造出属于自己的版本。FreezeMgr 可能会成为项目中的一个关键部分，所以请接受这个放弃声明，并仔细对待自己的实现版本，给予它足够的和应得的重视。

由于各种各样的设计原因，本范例中并不包含对解冻（thawed）类调用析构函数的机制。如果使用序列化数据这个特性，那样做就会成为一个问题，因为你可能要对内存进行分配。因此，大家必须考虑清楚，该如何处理释放由序列化类所分配的内存的工作。

1. FreezePtr 模板类

在构造的时候，FreezePtr 会自动向 FreezeMgr 注册自己。所以，直接把它包含在你的数据中，并像常规指针那样使用它就可以了。在一个结构或类中，进行如下声明：

```
FreezePtr<Foo> mFooPtr; // Foo 类型的一个指针
```

只能将 FreezePtr 指向那些用 FreezeMgr 分配的内存，否则当你冷冻数据时，FreezeMgr 就会 assert()。

2. 可冷冻类

如果某个类包含虚函数，它就必须继承抽象基类 Freezable。它的接口是：

virtual char const* GetClassName(void)：一个纯虚函数。必须重载这个函数，返回一个常量来标识该类。

virtual void SerializeIn(FILE* file)：在 thaw() 的结尾处调用。在调用之前，指针和虚函数表已经恢复了。

virtual void SerializeOut(FILE* file)：在 freeze() 的结尾处调用。重载这个方法就可以按照想要的格式输出数据。

因为必须在加载的时候调用类的构造函数来恢复虚函数表指针，所以除非调用 FreezeMgr::IsThawing() 以判断是否有一个 thaw() 在执行过程中，否则一定要确保在构造函数中没有做任何事情（空的构造函数）。

在冷冻或解冻之前，还需要调用 FreezeMgr::RegisterType()，将类的名字映射到一个 32 位的识别码上。

3. FreezeMgr 单例

FreezeMgr 单例控制着文件的创建、加载和保存。它还提供了内存分配和类创建方法，来创建可以被冷冻的数据。FreezeMgr 不会接受那些不是它创建的数据，如果试图冷冻这样的数据，FreezeMgr 就会 assert()。如果将 FreezePtr 指向未管理的数据，FreezeMgr 同样会 assert()。

FreezeMgr 的接口如下：

template<typename TYPE>TYPE* AllocTyped(int total=1)：分配并返回一个类，或者返回一个类型是 TYPE 的数组。如果要生成一个数组，只要将数组的大小传递进来即可。

void Flush(void)：释放由 FreezeMgr 分配的内存，并清除内表。

void Free(void)：删除由 FreezeMgr 分配的内存，通过 Thaw()加载的文件也包含在内。

void Freeze(char const* filename, void* addr：从一个给定的地址开始，遍历要冷冻的数据的图表，然后再写出一个文件。为它提供的地址必须是一个由 AllocTyped<>()返回的地址。

static FreezeMgr* Get(void)：返回 FreezeMgr 单体。

bool IsThawing(void)：如果有一个解冻操作在执行过程中，则返回 true。这与 Freezable 的构造函数有关，目的是避免重新初始化。

template<typename TYPE>void RegisterType(u32 typeID)：向 FreezeMgr 注册一个从 Freezable 派生的类。

template<typename TYPE>TYPE* ThawTyped(char const* filename, bool edit=false)：加载一个由 Freeze()生成的文件，并恢复它以供使用。

如果要创建从 Freezable 派生的类，那么在初始化代码中，必须在调用 FreezeMgr 之前，向 FreezeMgr 注册这些类。请记住，FreezeMgr 是根据注册的顺序为注册的类指派类型识别码的，所以顺序必须保持一致。如果觉得这个做法太过限制，也可以牺牲一些性能，轻松地改进这个做法。

Freeze()和 Thaw()是允许 void（空）指针的。这确实有些危险，但是却省掉了对 Freezable 的使用要求。希望用户不会滥用这个特性。如果你不像我这么乐观，这个地方同样可以很容易地进行改动。

1.13.5 总结

这个特殊的工具会带来大量的无趣代码，这些代码总是会为系统带来很多幕后的簿记开销。虽说如此，但我们也为一个一切齐全、即可使用的 API 打下了良好的基础。我们可以使用这些 API 来管理那些在游戏文件的书写和维护工作中产生的混乱和麻烦。

我们极力推荐这个范例程序，希望大家能够埋头研究，并对它进行扩展，将众多优秀作者的思想融汇其中。正是这些作者辛勤的劳动构成了《游戏编程精粹》系列书。

1.13.6 参考文献

[Bilas00] Bilas, Scott, “An Automatic Singleton Utility.” *Game Programming Gems*. Charles River Media, Inc., 2000.

[Boer00] Boer, James. “Resource and Memory Management.” *Game Programming Gems*. Charles River Media, Inc., 2000.

[Brownlow02] Brownlow, Martin. "Save Me Now!" *Game Programming Gems 3*. Charles River Media, Inc., 2002.

[Olsen00] Olsen, John. "Fast Data Load Trick." *Game Programming Gems*. Charles River Media, Inc., 2000.

[Wakelin01] Wakeling, Scott. "Dynamic Type Information." *Game Programming Gems 2*. Charles River Media, Inc., 2001.

1.14 高效且忽略缓存的 ABT 树实现方法

瑞士联邦理工学院（Swiss Federal Institute of Technology，简称 EPFL），虚拟现实实验室（virtual Reality Lab，简称 VRLab），Sébastien Schertenleib
Sebastien.Schertenleib@epfl.ch

今天，计算机体系结构所描述的多级存储系统正不断变得越来越慢、越来越大。这个多级存储系统包括寄存器、一级缓存、二级缓存、主存和硬盘。它们的访问时间从寄存器的一个存储周期，增长到了缓存、主存和硬盘各自的 10、100 和 100 000 个存储周期（见图 1.14.1）。从当前和未来 CPU 及内存的发展情况来看[Moore65]，[Hennessy03]，那些不能充分利用这个多级存储结构的算法，会有更多的缓存缺失，从而造成更严重的后果。为了解决这个问题，[Frigo99]提出了忽略缓存的算法。该算法的中心思想是在不需要知道存储块大小的情况下优化 I/O 模型的方案。他们在文章中描述到，通过使用忽略缓存的优化算法，可以解决一些基本的问题[Frigo99]。在第 1 次尝试中，他们使用了矩阵和快速傅里叶变换（Fast Fourier Transformation，简称 FFT）。后来，[Bender00]给出了另外一些建议，即使用 B 树（平衡树，B-tree）和二叉查找树表示。忽略缓存的算法与缓存感知算法不同，因为前者适合所有的内存结构。

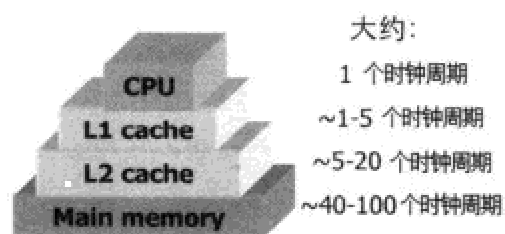


图 1.14.1 多级存储系统

1.14.1 计算机内存结构

大多数算法都会忽略内存结构。那些由小型数据结构驱动的算法，如二叉树，在访问它们的数据时会遭受严重的性能下降。每一个内存层都是以相似的方式工作，并由缓存块组成。当前的内存结构使用的缓存行的大小大约是 32～64 个字节。通过访问同一个缓存行中的数据，可以获得明显的性能提升[Patterson97]，[Hennessy03]。缓存行的生命周期取决于硬件相关的试探法[Smith87]。缓存的管理要选择合适的替换策略和关联策略。有些时候，缓存缺失是不可避免的[Hill02]，例如在下列情况下：

强制缺失：一种无法避免的缓存缺失，当某些数据第 1 次被访问时就会发生这种情况。

容量缺失：数据在前面的步骤中容纳于缓存中，但是由于缓存的更新

策略，数据从这一级的缓存中移出。

冲突缺失：由于不同的数据被映射到相同的缓存行而造成的缓存垃圾。

1．数据结构和缓存一致性

空间的排列对缓存的使用影响颇深。特别要注意的是，密集使用指针的数据结构并不是好的选择。可能需要一起访问的指针或数据，最好也保存在一起，这样可以优化内存的访问[Ericson03], [Ding04]。在某些情况下，它们也许无法遵从面向对象编程的方法论。

2．van Emde Boas 布局

van Emde Boas 布局[van Emde Boas77]是一个在内存中编排出平衡树的标准方法（参见图 1.14.2）。在忽略缓存的模型中，使用它可以非常高效地遍历从根节点到叶子节点的每一条路径，其时间复杂度为 $O(4*\log_B(n))$次内存传输，其中 $B=NbElement/CacheLine$。

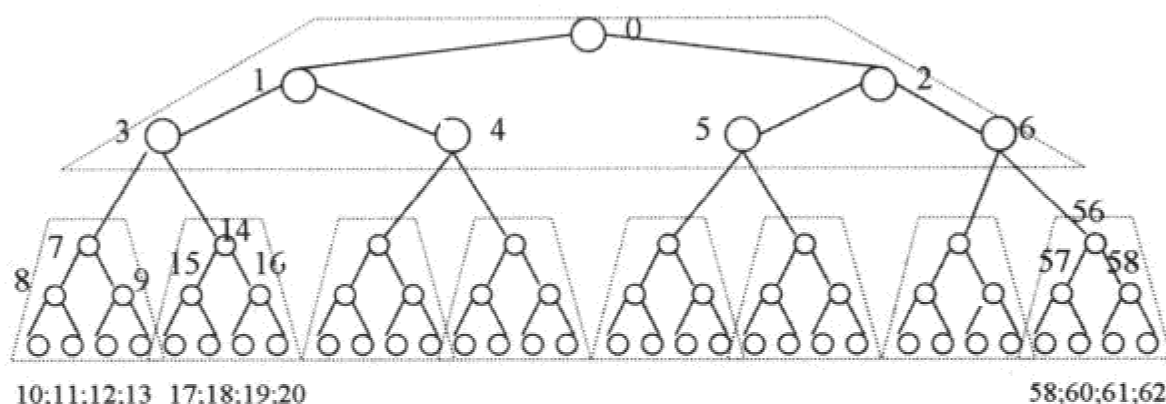


图 1.14.2　van Emde Boas 树的表示法。在这个例子中，每个子树由 7 个节点组成

1.14.2　ABT 树

ABT 树与 KD 树非常类似[Szécsi03]。在每个构建步骤中，沿坐标轴的分割平面会分割出两个子节点。与 KD 树有一点不同的是，ABT 树的算法可以将分割产生的子节点的 AABB 盒最小化，并将所有几何体独占式地存储在叶子节点中。这样，每个节点就变成了空间中一个完全封闭的区域。在这个空间中，我们会遍历这些内部节点，将场景中的不可见部分剔除。

1．ABT 树的创建

ABT 树的创建工作是从一个根节点开始的，这个根节点包含一个对整个场景 AABB 盒的引用。递归创建法会用一个轴向上的分割平面，将局部当前节点的 AABB 盒分割成两个部分（2 个面），然后根据它们的中间层，把这 2 个面分别分配给两个子节点（参见图 1.14.3）。

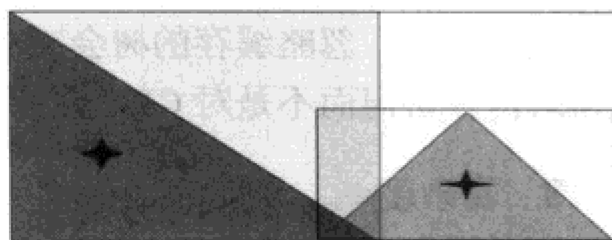


图 1.14.3　沿坐标轴的包围盒的重新调整

一旦所有面都分配完毕，每个子节点就会重新计算它们各自的、含有这个层中独一无二的面的 AABB 盒。结束条件取决于特定的 3D 场景和设想的硬件。分割策略会尽量将下列属性最小化：

- 空间局部化，参见公式 1.14.1：

$$f_1(n) = Min(Area(boxLeft) + Area(boxRight)) \quad (1.14.1)$$

- 树的平衡性，参见公式 1.14.2 和 1.14.3：

$$f_2(n) = Min\left[\Delta\left(Area(boxLeft) - Area(boxRight)\right)\right] \quad (1.14.2)$$

$$f_3(n) = Min\left[\Delta\left(\sum(ExtendedArea) - \sum\left(Area\right)\right)\right] \le \varepsilon \quad (1.14.3)$$

- 在性能出现显著降低之前，Epsilon 应该维持在 5%～10%以下。在大多数情况下，将 AABB 场景复杂度扩展 5%，就可以包含 90%的面，参见公式 1.14.4 和 1.14.5：

$$f_4(n) = Min\left[\Delta\left(\sum faces(boxLeft) - \sum faces(boxRight)\right)\right] \quad (1.14.4)$$

$$f_5(n) = Min\left[\Delta\left(\int ressources(boxLeft)dt - \int ressources(boxRight)dt\right)\right] \quad (1.14.5)$$

最后的公式就变成了公式 1.14.6：

$$f(n) = w_1 * f_1(n) + w_2 * f_2(n) + w_3 * f_3(n) + w_4 * f_4(n) + w_5 * f_5(n) \quad (1.14.6)$$

根据游戏引擎的瓶颈和场景的组织（场景图、特效等），公式中各项的权重会有所变化。在预处理阶段和运行时，采用的方法也是不同的。

在运行时，在不同的渲染阶段中（剔除、阴影和碰撞检测等），每个视点通常都会执行一个完全或部分的遍历。好在，ABT 树的遍历计算起来确实非常简单和快速。从上一个局部根节点开始，递归函数会检测两个子节点是否位于视锥内部，如果确实位于视锥内部，那就继续执行树的遍历。当遍历到一个叶子节点时，它包含的所有几何体都可以传送到渲染管道中的下一个阶段。由于每个面都是独一无二的，所以不需要进行额外的测试（如碰撞检测）。我们还可以为叶子节点和局部素材维护一个很小的顶点缓冲器。为了减少所使用的顶点缓冲器的数量，我们可以从邻居子节点的位置中获益。因此，几个叶子节点可以共享一个单独的顶点缓冲器，只要不超出顶点缓冲器的容量限制即可（通常是 65K，16 位索引）。这样一来，当几个邻居节点需要一起处理时，就可以获得更高效的分支处理，提高渲染性能[Wloka03]。

通过简单地记录每个 AABB 盒在它们的子树中的移动情况，就可以动态地调整 ABT 树。动态树的一个缺点就是它们会随着时间而退化。

2. 效率

所有二叉树，特别是 BSP 树和 KD 树，都会受到树的深度的困扰。即使 ABT 树的处理能力非常优秀，树的深度也依然是个重要的问题。假设树的实现是使用指针，而不是隐式指针，那么我们可以认为每当树要沿用一个指针时，就会发生 CPU 的缓存缺失。相对于树的深度，缓存缺失的数量会相应地增加到一个极限值。这时候，缓存缺失变得比交叉检测的代价还要大。但是，忽略缓存的树会减少与缓存行大小有关的缓存缺失。这样，树的遍历工作就变得对内存访问而不是对 CPU 性能不太敏感了。

3. 复杂度

和所有的二叉树一样，ABT 树也需要一个 $O(\log n)$的搜索时间[Sedgewick90]。通过使用 van Boas 忽略缓存的布局表示法，搜索时间可以估算为 $O(4*\log_B(n))$。其中，$B = CoupleNode \div CacheLines$[Brodal03]（参见图 1.14.4）。

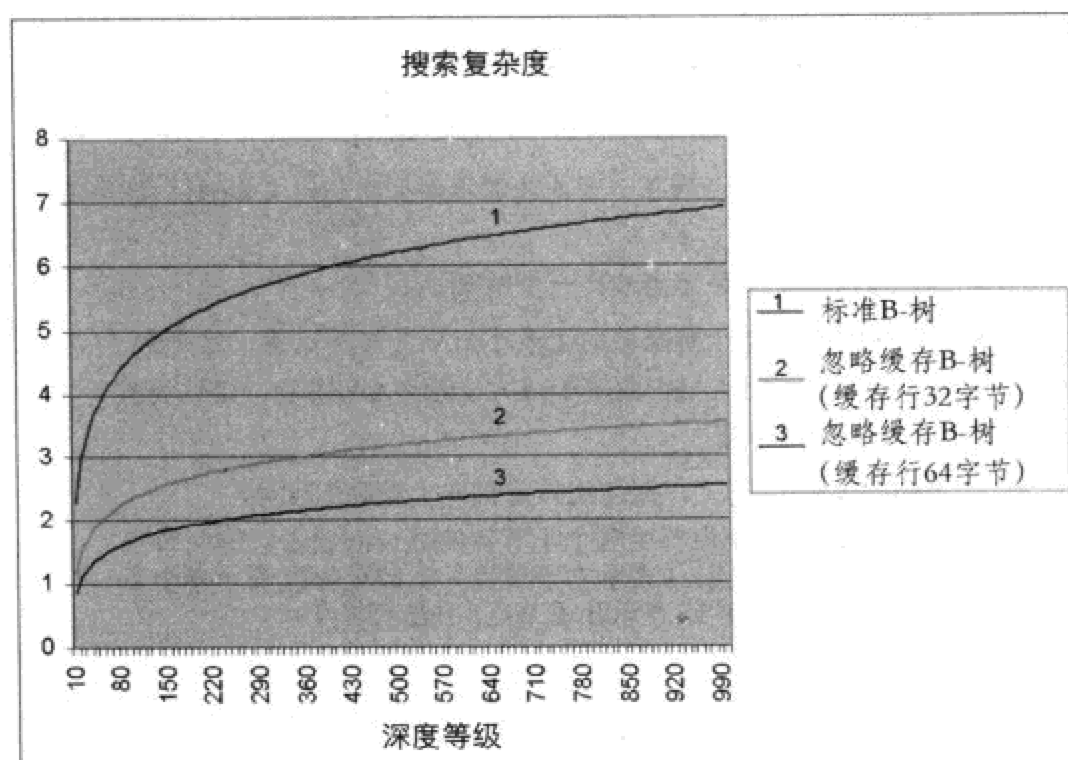


图 1.14.4　搜索时间排序

每个节点需要存储各自的局部 AABB 盒和一个指向其两个子节点的指针（或索引）。如果使用一个不成熟的实现，内存的消耗相对地就变得更为重要了：

- AABB 由 6 个浮点值来描述：6×4 个字节
- 指向两个子节点的指针：2×4 个字节（32 位 CPU），或者 2×8 个字节（64 位 CPU）

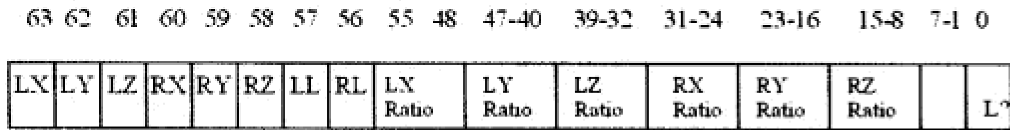
这样算来，每个节点的内存总需求是 32 个字节（对 32 位 CPU 而言），或者 40 个字节（对 64 位 CPU 而言）。

但是，[Gomez01]告诉我们只需要保存每个子节点 AABB 盒的相对范围（extent），这样就可以把每个范围（extent）压缩成一个 8 位的整数值。这个谨慎的估算方法只有 1/255，也就是约 0.4%的相对误差，完全可以被平均 5%～10%的 AABB 重叠率覆盖掉。

4. 采用冗余技术

[Gomez01]描述了一些可以减少内存开销的实用方法。在每次分割时，定义子节点 AABB 盒的 12 个范围（extent）中的 6 个直接来自父节点，因为父节点包含的所有面都会被分割、传递给叶子节点。我们不是单独为每个子节点保存几个字节的信息，而是把两个子节点保存为一个整体。这样，我们就不用局部地保存每个子节点 AABB 盒的绝对数据，取代它的是每对子节点都会保存它们相对于父节点 AABB 盒的一个比例。

因此，我们需要有 6 个字节来表示子节点的相对比例。通过不同特征指定的另外一个字节会使用两个子节点的相对位置，并再次使用父节点的相关数据。在运行时的遍历过程中，递归方法会趁机重新计算子节点局部 AABB 盒的位置。最后，因为在此阶段最后那个字节还有两位没有用到，所以我们就用这两位来说明左/右子节点是右节点，还是叶子节点（参见图 1.14.5）？为了遵守忽略缓存算法的数据结构定义，我们把这些数据都保存为 8 个字节的一个值，为树本身留下了 7 个未使用的位（bit）。例如，我们可以使用这 7 个位（bit）来标识是否加载了下列子树（subtree），这对流（streaming）应用非常有价值。

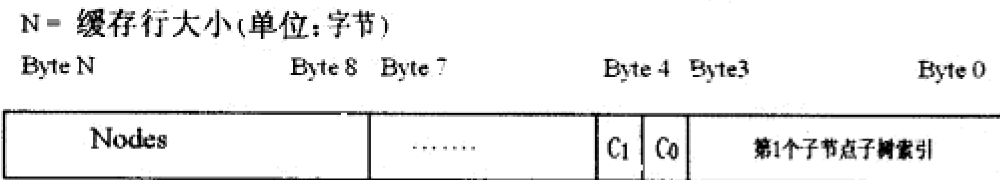


计算机的缓存行架构总是2的幂，因此我们把数据结构定义为64位或8个字节

- L? 指出它是一个节点或是一个叶子
- Ratio Lx,y,z: 基于特定轴向上最小部分的相对比例，这是为了强制压缩后的AABB盒等于或大于绝对AABB盒。该比例被分割为255等份
- Ratio Rx,y,z: 同上，但使用特定轴向上较高的部分
- 特征位：63至58位标明哪个子节点的extent属于它的父节点（1表示左子节点，0表示右子节点），57至56位表示哪个子节点是叶子节点（1表示叶子节点，0表示节点）

图 1.14.5 ABT 树的节点数据表示（一对子节点）

最后，因为子树总是 2 的幂减去 1，且缓存行的大小也总是 2 的幂，所以我们有 8 个字节可以用来把这棵子树链接到下一棵子树。由于打算用隐式指针来表示这种分级，所以我们用 4 个字节来保存第 1 个可用子节点的索引。另外 4 个字节中，有些位用来说明哪个端节点（end node）连接在子节点的子树上。根据不同缓存行的大小，有些位可能仍然未被使用（参见图 1.14.6）。



- Bytes 0～3: 第1个子节点子树索引，所有子树都是直接邻居
- Bytes 4～7: Cx?: 特征位，说明端节点i是否连接到一棵子树上
- Byte 8～N: 8个字节为一组，表示一对子节点或一个叶子节点

图 1.14.6 缓存行的组织结构

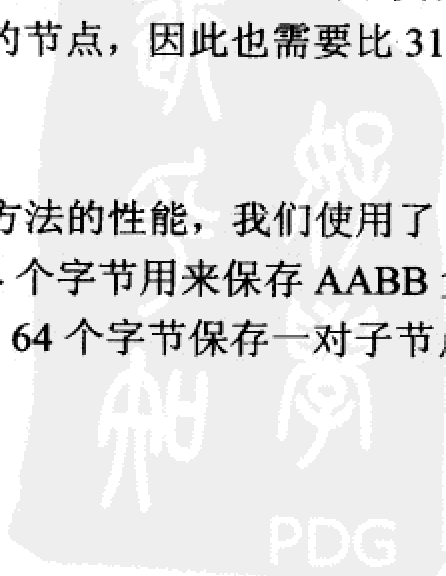
当上一棵子树的所有子树都成为直接邻居时，这个创建过程就完成了（参见图 1.14.7）。

现在该考虑叶子节点了，它也是由 8 个字节组成。一个位用来表示它的条件。同样，它可能也取决于系统使用的节点数量。在 32 位系统中，一般 31 个位就足够了，留出 4 个字节可以用来保存其他的附加信息（主要用于资源的动态管理，参见图 1.14.8）。不过，64 位程序可能需要使用更多的节点，因此也需要比 31 位更多的内存空间。

5. 性能

为了分析这种方法的性能，我们使用了 3 个不同的实现（参见图 1.14.9）：

直觉算法：6×4 个字节用来保存 AABB 盒，8 个字节用来保存指向子节点的指针（32 位操作系统），或者用 64 个字节保存一对子节点。



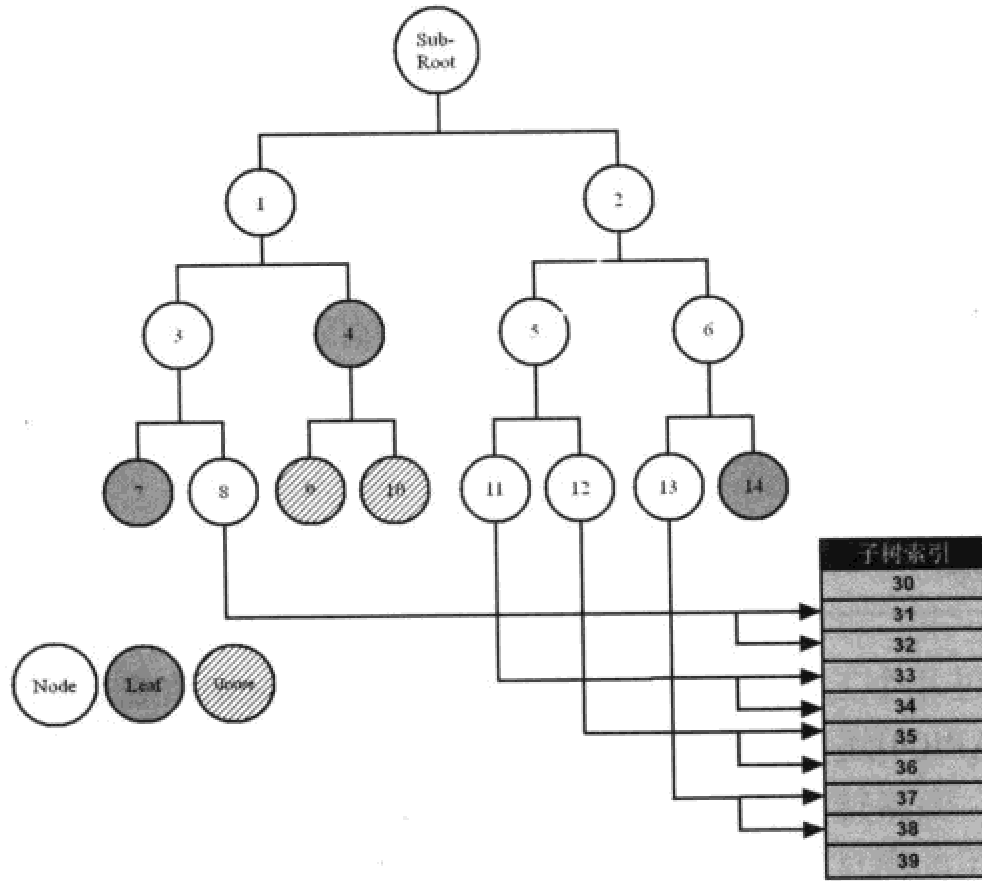
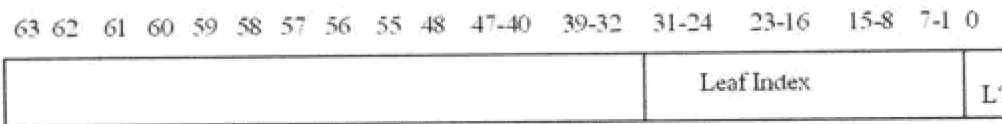
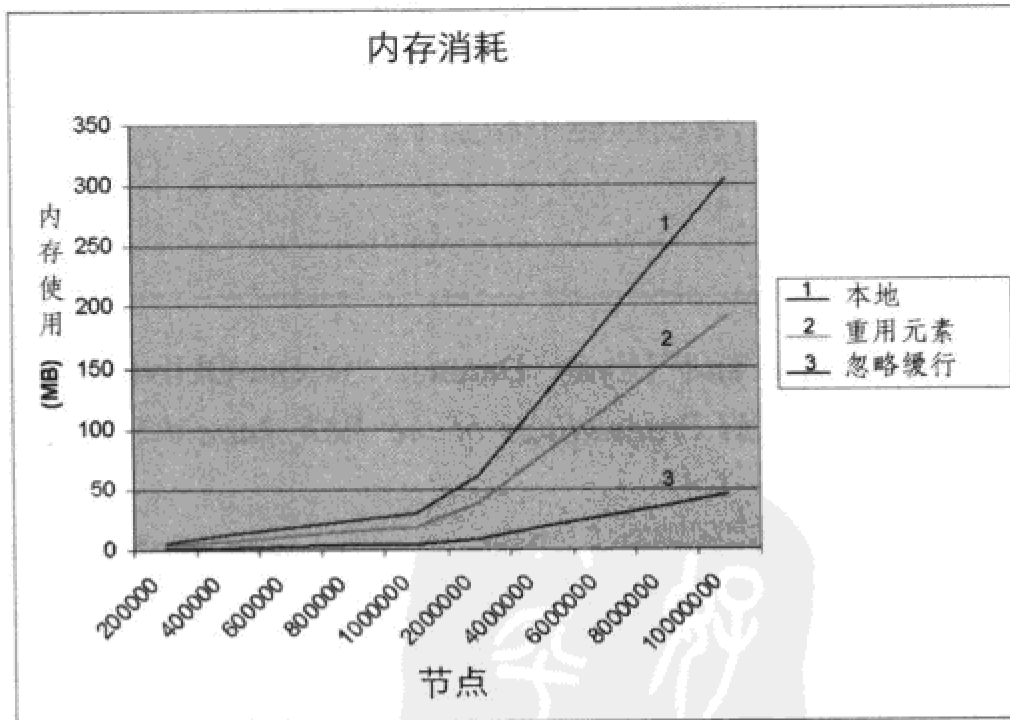


图 1.14.7　子树的组织结构，包括到次级子树的链接



- L? 指出它是一个节点还是叶子
- 在这个配置下，我们使用31个比特来表示叶子节点，最后4个字节没有用于树本身，但是可以在其中保存一些附加信息（例如在流应用中）

图 1.14.8　ABT 树的叶子节点的数据表示



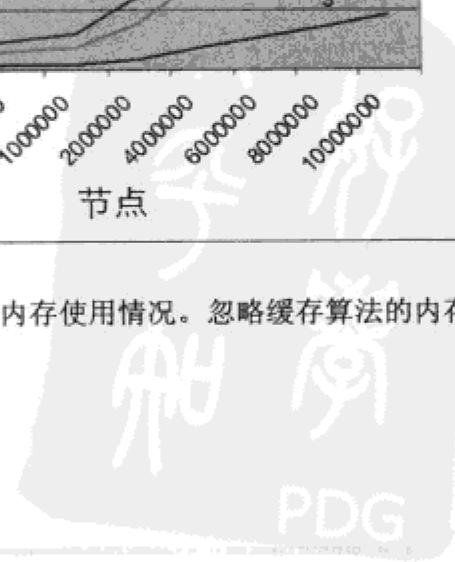


图 1.14.9　3 个不同实现方法的内存使用情况。忽略缓存算法的内存消耗取决于树的平衡性

采用冗余技术：8 个字节用来保存 AABB 盒，8 个字节用来保存指向子节点的指针，或者 48 个字节用来保存一对子节点。

忽略缓存算法（缓存行大小为 64 个字节）：每一对节点需要 8 个字节，外加 8/7 个字节用来保存隐式指针，平均下来是 9.14 个字节。全局内存需求取决于树的平衡性（参见图 1.14.7）。

考虑到要使用 8 位整数值来表示子节点 AABB 盒的范围，我们要估计转换这些值所需要的系统开销。我们对一个有 16 K 个叶子的随机分布系统进行了实验，结果显示其开销大概是 7%，通过采用更优的缓存友好的设计方案，这个开销完全可以得到弥补。

1.14.3 确认阶段

确认工作是通过几个工具来完成的。我们使用了[Vtune04]来观察内存访问，并发现非关联的瓶颈。使用[PAPI04]还可以监视硬件计数器。它们可以跟踪缓存缺失、TLB（Transition Look-Aside Buffer，旁路转换缓存）缺失，及其他类似事件。但是，详细而精确的测试还需要我们创建一个专用的测试工具。通过对几个不同算法实现的测试（从基于 RAM 的算法实现，到忽略缓存的算法实现），我们搜集了一些统计数据，它们可以帮助我们更好地理解忽略缓存的实现，并对它进行有效的改进。

1.14.4 总结

虽然本文专注于讨论剔除算法，但 ABT 树在其他很多领域也是非常有用的，包括 AI 和 3D 音效管理。这让我们可以使用多个视图来共享游戏世界的表示[Bar-Zeev03]。

当前的硬件技术发展，要求我们去研究通过不同的内存层访问数据所需要的系统开销。CPU 的速度会继续大幅提升，但是内存的发展却跟不上这个脚步。计算机的性能将会变得越来越依赖于内存访问，而不是单纯的 CPU 原始性能。

实时仿真需要大量的数据集，而内存瓶颈会成为它的限制，CPU 的发展所带来的好的影响也会因此而弱化。最后要说的是，忽略缓存的算法具有很好的适应性，它为缓存感知算法提供了一个很好的替代方案，且二者的性能不相上下。

1.14.5 参考文献

[Agarwal03] Agarwal, Arge and Bryan Danner. "Cache-Oblivious Data Structures for Orthogonal Ranger Searching." *ACM Proceedings of the 19th Annual Symposium on Conceptual Geometry*. 2003.

[Bar-Zeev03] Bar-Zeev. "Scenegraphs: Past, Present and Future."Available online at *http://www.realityprime. com/scenegraph.php*. 2003.

[Bender00] Bender, Demaine and Farach-Colton. "Cache-oblivious B-trees." In *Proc. 41st Ann. Symp. on Foundations of Computer Science*, 399–409. IEEE Computer Society Press, 2000.

[Brodal03] Brodal. "Cache Oblivious Searching and Sorting." Seminar, IT University of

Copenhagen. Copenhagen ,Denmark. 2003.

[Ding04] Ding,C. "Data Layout Optimizations, Computer Organization".Lecture, Rochester, NY, 2004.

[Ericson03] Ericson, Christer. "Memory Optimization." CDC 2003, Santa Monica: Sony Computer Entertainment, 2003.

[Frigo99] Frigo, Leiserson and Ramachandran Prokop. "Cache-Oblivious Algorithms." In *Proc. 40th Annual Symposium on Foundations of Computer Science* (FOCS), 285–297. 1999.

[Gomez01] Gomez, Loura. *Compressed Axis-Aligned Bounding Box Trees*. Charles River Media, 2001.

[Hennessy03] Hennessy, Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., 2003.

[Hill02] Hill, Lipasti. *Cache Performance*. University of Wisconsin-Madison, 2002.

[Moore65] Moore. "Cramming more components onto integrated circuits." In *Electronic Magazine* 38, 114–117. 1965.

[PAPI04] PAPI. "Performance application programming interface." Available online at *http://icl.cs.utk.edu/projects/papi*. 2004.

[Patterson97] Patterson, Hennessy. *Computer Organization and Design Second Edition : The Hardware/Software Interface*. Morgan Kaufmann, 1997.

[Sedgewick90] Sedgewick. *Algorithms in C*. Addison Wesley, 1990.

[Smith87] Smith. "Line (block) size choice for CPU cache memories." In *IEEE Transactions on Computers*, 1987.

[Szécsi03] Szécsi. "An Effective Implementation of the K-D Tree." In *Graphics Programming Methods*, Charles River Media, 2003.

[van Emde Boas77] van Emde Boas. "Preseıving order in a forest in less than logarithmic time and linear space." *Inf. Process. Lett.*, 6:80–82. 1977.

[VTune04] *VTune*. Intel Corp, 2004.

[Wloka03] Wloka. "Batch, Batch, Batch: What Does It Really Mean?" *GDC*, 2003.

1.15 状态机的可视化设计

Scott Jacobs
scott@escherichia.net

并非所有要在游戏中运行的代码都需要程序员去编写。一款游戏要完成很多必需完成的行为，对于其中的一些行为，如果能用一种特殊的描述来表达它们，然后在开发的过程中将其转换为数据和代码，这样会更好。这个过程就是大家熟知的“代码生成”。代码生成非常适合于那些可以用状态机来实现的系统。有关代码生成的更完整讨论，可参考[Herrington03]。代码生成要求在生成过程开始之前，完整、清晰地表达出目标系统的需求。用如此详细的信息来描述一个状态机，这就是本文的主题，这个描述过程可以用流程图可视化地完成。

1.15.1 为什么需要代码生成

在开发过程中融入一定程度的自动代码生成有颇多好处。例如，只要编写（和调试）一次将状态机描述转换为数据的代码，且代码能够运行，那么在整个项目中，你就可以不断地重用这个转换器。而且，状态机本身也可以不断地使用这些生成的数据和代码。只需一次编码、调试和测试，就可以反复重用这些组件来强化用户界面屏幕、复杂的粒子特效、游戏中的过场动画，甚至是游戏逻辑。将这种级别的运行时数据和代码生成机制融入到系统中，并在系统运行中执行这个机制，可以为内容创作人员提供强大的灵活性和自由度。标准的且非常常见的做法是让设计人员用文本来描述系统需求，然后再把它作为规范交由程序员去实现。与这种做法相比，根据专用的可视化状态机描述生成状态机，会带来非常多的好处。

对于这样的一个系统，其最大的优势也许就是那个最明显的特性。状态机的描述是用可视化方式表达的，且最终的结果是根据这个可视化的描述直接生成的，这就为我们提供了设计与功能实现的即时同步性。设计上的修改会直接地、即时地传递到系统中，导致状态机的运行时操作的改变。另外一个先进之处就是，专用的数据创建和可视化工具软件可以使状态机的设计工作变轻松。

对于数据的创建和编辑，第 1 个选择显然就是流程图的绘制工具。与文本式的系统操作描述不同，设计人员和程序员在这里可以查看描述整个系统的流程图，并直观地从中找出逻辑错误或设计问题，从而避免将编码时间花费在一个有问题的系统的实现上。在可视化工具的约束下进行设计，

还可以加强系统的一致性和可行性。举个例子，对于玩家是否可以在跳起的同时开枪射击，游戏策划人员可以在策划阶段就做出决定：链接或者不链接玩家的射击状态和跳起状态，这样程序员就不用回头去查看策划文档了。我们还可以快速地重新配置游戏的逻辑流，且不需要进行任何手工代码修改，就可以应用多种不同的方法来处理游戏的运行动作。

1.15.2 让“可视”成为可能

为了让可视化的状态机设计成为可能，我们需要提前定义一些游戏引擎的参数和方法，并把它们提交给状态机的设计人员。这些参数和方法就是那些也需要在游戏引擎中对状态机引擎公开的内容。它们可以很简单，如用来描述玩家当前输入状态的全局布尔变量；也可以非常复杂，如用来公开某些计时器、事件生成、游戏对象管理等等的脚本系统。

对于一些微小的、容易测试的、可能无关的变量、方法和模块，可以用状态机把它们链接在一起，实现我们预期的游戏内部活动。举个例子，比如需要一个简单的状态机，用来描述游戏中某种枪支的行为。程序员可以实现一些必要的离散行为，把这种枪支集成到游戏中，如提供一些钩子函数来播放音效、生成粒子特效和破坏游戏中的物体。然后，就可以用可视化的设计工具设计出状态机，并反复实验各种相关的游戏性动作，本文稍后会对此进行讨论。

使用常见的工具软件

有很多不同的工具软件都可以用来绘制状态机流程图。也许对于你来说，定制一个可以满足自己特殊应用需求的工具，才是最佳的解决方案。但是，如果是第 1 次尝试使用可视化状态机的设计和代码生成，最好还是使用现有的流程图编辑软件，比如 UMLPad[Bignami04]。这个有 GPL 授权许可（General Public License，通用性公开许可证，简称 GPL）的软件可以在很多平台上运行，而且作为一个简单的状态机编辑器，UMLPad 用起来很顺手。使用这个工具可以轻松地创作出一些基本的流程图，而且它生成的基于文本的文件格式，非常容易理解和进行语法分析，这样就可以很直接地将流程图定义转换为可以在游戏引擎中运行的状态机。

读取那些经过可视化设计得到的状态机描述的状态机引擎，可以用各种编程语言来实现，但本文使用的状态机引擎是用 Lua[Lua04]编写的。在其发展的第 10 个年头中，Lua 在游戏开发领域获得了前所未有的广泛应用[Burns04]，用它来开发状态机引擎仿佛也是很自然的选择。

通过 Lua 把游戏引擎的变量和方法公开给状态机设计者的过程是非常直接的，有很多现成的可用资源描述了具体的实现方法。在一些工具的帮助下，这个过程常常可以更为高效，比如 tolua++[Manzur04]，它可以自动生成有关代码，向 Lua 公开 C/C++的类型。至于如何使用 Lua，应该阅读那个简短、但却无所不包的 Lua 使用手册[Ierusalimschy03a]，以及 Roberto Ierusalimschy 编著的图书《*Porgramming In Lua*》，二者都提供了书面版本和电子版本[Ierusalimschy03b]。

Lua 是一种动态脚本语言，函数在其中是第一类的对象。这给了我们很大的灵活性，因为从流程图到状态机的转换过程可以产生一个由数据和机器生成的代码组成的混合体。每个状态可以包含几个方法，以判断该状态机是否应该前进到一个新的状态。使用 Lua 的一个名

为“*meta table*（元表）”的特性，可以编写出简单、通用的状态机管理代码，来统一管理状态机在游戏中的所有实例。由于所有的状态机都共享同一个 meta table，所以它们对外的接口也是相同的，这就使得游戏引擎对不同状态机的使用能够保持一致性。每个状态机的独特信息都完整地包含在（用可视化方式设计出的）数据中。

1.15.3 状态的管理

本文使用的状态机引擎虽然简明，但功能却很强大。这里讲到的数据结构、方法和状态之间的关系，都是由基于 UMLPad 文件的转换脚本自动生成的。知道了这些，就会觉得这些工作更加简单了。每个状态都是作为 Lua 元表来实现的，状态机引擎会跟踪这些表，看哪个表是当前的状态。每个表可以包含任意数量的标准方法，状态机引擎会在适当的时间查找并运行这些方法。

在这些方法中，对于某个给定的状态，那些没有必要存在的方法就是进入或退出一个状态时需要运行的方法。当状态机引擎更新时，当前的状态也有机会来更新自己。每个状态的表中都包含一个链接列表，这些链接指向的是从当前状态可能会进入的其他状态。每个链接可以包含一个前进条件方法，运行这个方法可以判断状态机引擎是否应该使用该链接，前进到另一个新的状态上。如果没有这个条件判断方法，状态机引擎就会一直沿用该路径，将该路径所指向的状态设为当前状态。

在每个状态机的更新循环中，系统会评估当前状态所有路径链接的前进条件。如果没有任何一个前进条件可以满足，状态机就会维持在当前状态上。在可视状态机设计工具中，每个前进条件都与一个给定的路径链接相关联。前进条件可以很简单，比如一个单独的布尔变量（输入按钮 X 当前是否被按下？）；也可以是一个更为复杂的函数（计时器 X 是否过期？装备库中剩余的弹药是否足以允许重新装弹？）。所有的状态都有其各自的特定状态数据表。进入、退出和更新方法，以及每个路径链接的前进条件方法，都可以使用这些数据表。这个特定状态数据区还可以方便地放置计时器或计数器。

通过向状态机工厂申请一个已命名状态机的实例，就可以创建状态机。状态机工厂会加载那些由可视的状态机数据文件生成的文件，并安装状态机引擎元表，然后向调用者返回对象。调用者一旦获得了这个元表，就可以开始更新状态机了。状态机工厂的使用者可以是另外一个 Lua 脚本、游戏引擎的 C/C++代码，或者是其他 Lua 接口使用的代码。状态机启动之后，该状态机的使用者应该以适当的周期频率来运行 `Update()`。如果状态机完成了，`Update()`就会返回一个 Lua 值 nil，它在条件声明中求值为 false。

1.15.4 系统组装

作为这项技术的一个具体应用案例，让我们设计一个状态机来操作一把简单的手枪。这把手枪应该可以发射子弹，打光子弹，用装备库中的弹药重新装弹；如果没有剩余弹药了，还可以空枪射击。为了看到它的可视化设计，可以运行 UMLPad，打开随书光盘中的文件 Gun1.uss。图 1.15.1 显示的是该状态机设计的一个快照。

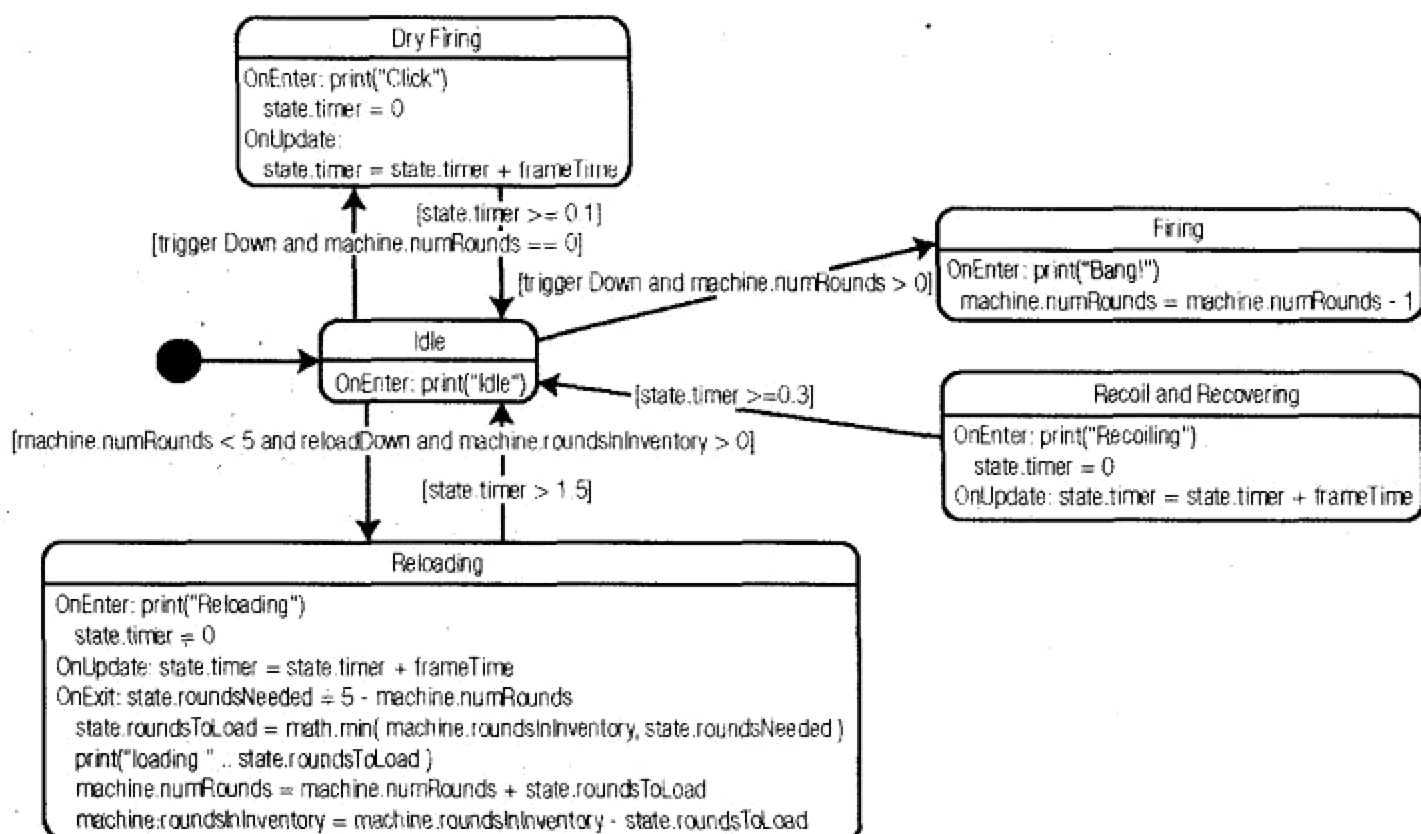


图 1.15.1　一个简单的状态机，描述了一把手枪可以执行的几个动作

这个状态机有一个起始点（图中的大黑圆点），却没有终止点。一开始，状态机会立即进入 Idle（空闲）状态。由于我们对一个通用状态机编辑器进行了功能再造，所以在为每个状态填写数据时，必须非常小心地遵循一些既有的约定。在 Activites（行为）和 Description（描述）字段，代码生成脚本会搜索一些特殊的字符串：`OnEnter:`、`OnExit:`和 `OnUpdate:`。每个字符串后面的所有文本就成了状态机引擎在适当的时间要运行的方法（method）。我们要为这些方法提供一个单独的参数：state（状态），这个参数就是一个特定状态数据表，可以用来存储数据。如果你想在自己的制作管道中集成这个好东西，可能需要对 UMLPad 做一些修改，为每个方法增加几个独立的字段。Idle 状态中只定义了一个方法：OnEnter。每当状态机引擎将状态机导入 Idle 状态时，系统就会运行 OnEnter 方法。在这个例子中，OnEnter 方法在屏幕上显示为“Idle”字样。

通过观察，我们可以看到，状态机离开 Idle 状态有 3 种方式。每个路径链接上都详细地写出了前进到下一个状态需要满足的条件。在每个状态机的更新循环中，系统会对这些条件依次进行判断，当某个条件可以满足时，状态机引擎就会将路径链接指向的状态设置为当前状态，并首先运行上一个当前状态的 OnExit 方法（如果有的话），然后再运行新的当前状态的 OnEnter 方法（如果有的话）。

在这个例子中，我们假设由于玩家按下了射击键，游戏引擎作为回应将 triggerDown 设为 true，Fire（射击）状态就变成了当前状态。在实际的实现中，Firing（正在射击）状态的 OnEnter 方法会调用相应的其他方法来播放射击音效，并在游戏世界中生成一个子弹的实例。在下一个更新循环中，状态机发现只有一个路径链接在 Firing 状态之外，而且没有什么条件限制，所以状态机就会马上采纳这个路径链接，前进到下一个状态。

在 Recoil and Recovering（反冲和复位）状态中，也只有一个连出的路径链接（回到 Idle 状态），但是我们不能马上使用这个路径链接，必须要等到计数器达到某个限定值。在每个更新循环中，状态机引擎都会运行 Recoil and Recovering 状态的 OnUpdate 方法，给计数器加上

一个增量。frameTime 是游戏引擎设置的一个变量。其他几个状态的运做情况与此类似，不再赘述。

数据驱动式设计的美妙之处

运行 Gun1.uss 上的转换脚本 ussToState.lua，产生的结果文件是 Gun.lua。StateMachine-Factory.lua 中的状态机工厂可以使用这个结果文件，为发出请求的代码提供手枪状态机。脚本 testGun.lua 做的就是这件事情，即向 StateMachineFactory 请求一个名为 Gun 的状态机，然后模拟一个游戏引擎的角色为状态机提供一些变量，如 frameTime、triggerDown 和 reloadDown。脚本文件 testGun.lua 会让这把手枪在 Idle、Fire、Recover 和 Reload 这几个状态之间依次运行，直到打光所有的弹药，然后进入空闲状态和空枪射击状态的死循环。

现在就可以看到该技术的美妙之处：重新配置现有的状态机，使之符合新的设计需求。策划部门可能会想对一种新武器做些玩法实验。该武器在弹夹打空和扳机松开时，会自动重新装弹。为了帮助实现这一点，可以增加一个从 Idle 状态到 Reload 状态的路径链接，其条件是：not triggerDown and numRounds == 0。现在，这个状态机就可以实现我们想要的行为了。图 1.15.2 中显示的是 Gun2.uss，请留意从 Idle 状态到 Reload 状态的新路径链接。

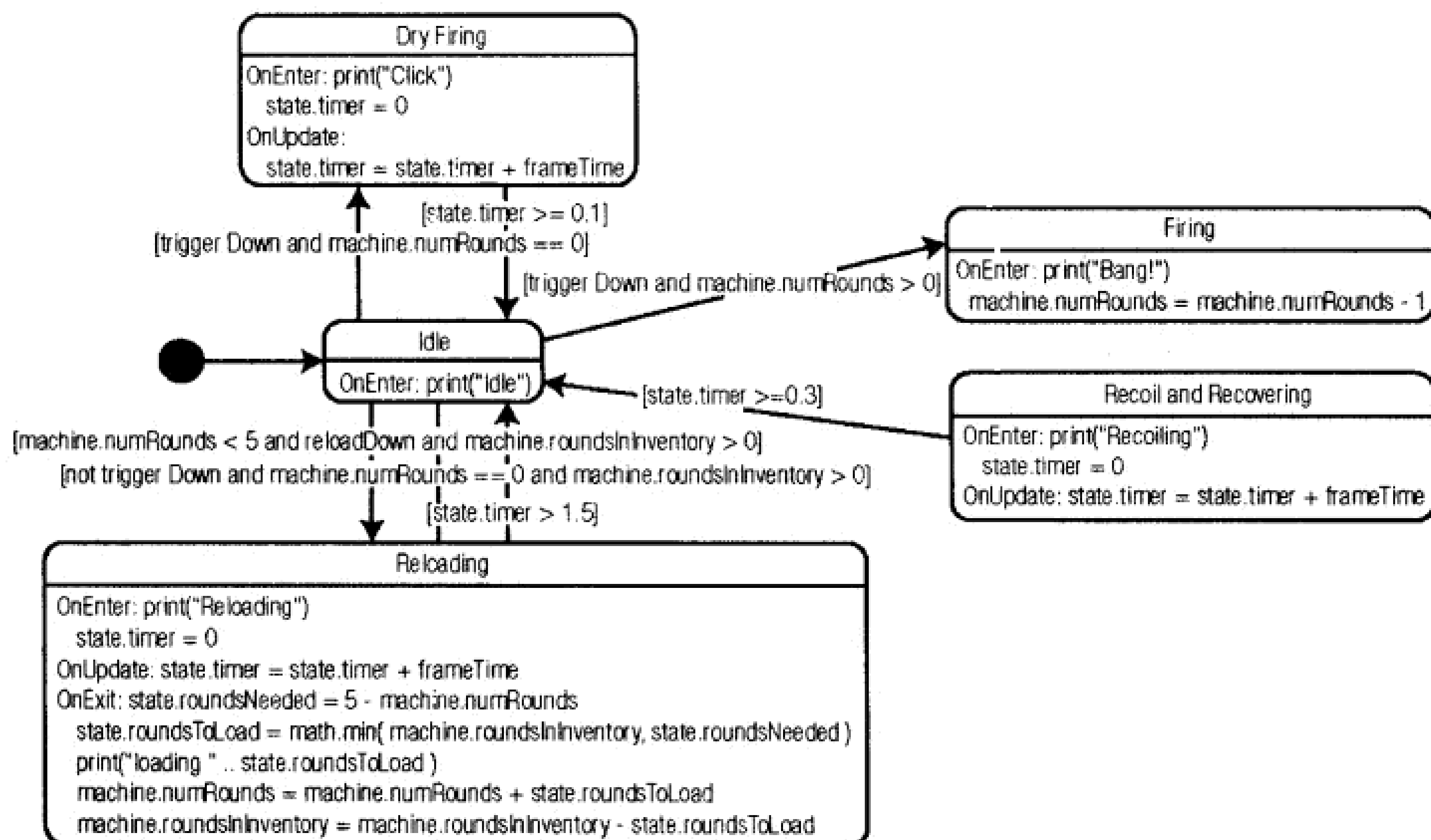


图 1.15.2　从 Idle 状态到 Reload 状态增加了一个路径链接，当手枪打光弹夹后，就要采纳这个路径链接

可以作个对比：分别运行从 Gun1.uss 生成的 Gun.lua，和从 Gun2.uss 生成的 testGun.lua，可以将 reloadDown 强制设置为 false，来看看新增行为的效果。即使 reloadDown 总不为 true，当子弹打光时，枪支仍然会自动装弹。现在，再假设策划部门又来了新的需求：枪支需要有 4%的卡弹的机会，且需要 3 秒钟的时间才能清除。在每次的 OnUpdate 循环中，改变 Idle 状态，转而去计算卡弹出现的几率。当扳机扣下时，就会连续地射出子弹，如果卡弹的几率小于 0.04，状态机就会把 Jammed（卡弹了）状态设置为当前状态（新状态）。Jammed 状态的计时器必须增加到 3 秒，之后 Idle 才能再次变为当前状态。瞧，只是用鼠标单击了几次，并用键盘输入了很少的东西，我们就已经创建了两个新的游戏对象行为。

1.15.5 总结

希望这种用文本编辑器之外的工具来创建游戏中的动作和行为的潜力，已经激起了大家的兴趣。本文谈及了一些工具和程序语言，这为大家开始尝试使用状态机代码生成打下了良好的基础，但更为重要的是让我们受到鼓舞的这个概念：系统的核心结构，如状态机，只需要提取、编写和调试一次，然后通过工具提供的数据和代码，就可以实现。这一观念带来的诸多好处有：系统的自我文档编制、快速的行为修改、系统约束的强制执行，而且系统设计人员不需要为此去了解专门的编程语言知识。

1.15.6 参考文献

[Bignami04] Bignami, Luigi. UMLPad. Available online at *http://web.tiscali.it/ggbhome/umlpad/umlpad.htm*.

[Burns04] Burns, Jon and David Eichorn. "GDC 2004—Lua in the Gaming Industry Roundtable Report." Available online at *http://lua-users.org/lists/lua-l/2004-04/msg00164.html*.

[Herrington03] Herrington, Jack. *Code Generation in Action*. Manning Publications, 2003.

[Ierusalimschy03a] Ierusalimschy, Roberto., L. H. de Figueiredo, and W. Celes. "Lua 5.0 Reference Manual." Technical Report MCC-14/03, PUC-Rio, 2003. Available online at *http://www.lua.org/manual/5.0/*.

[Ierusalimschy03b] Ierusalimschy, Roberto. *Programming in Lua*. Lua.org, 2003. Also available online at *http://www.inf.puc-rio.br/~roberto/book/*.

[Lua04] Lua. Available online at *http://www.lua.org/*.

[Manzur04] Manzur, Ariel. tolua++. Available online at *http://www.codenix.com/~tolua/*.

1.16 泛型组件库

Warrick Buchanan
warrick@argonaut.com

随着计算机游戏的体积和复杂度的不断增加，对那些可以使用更加结构化的设计方法的技术的采纳就变得越来越关键。把一个软件系统看成是使用一系列可重用的组件，这个概念并不新鲜，但对游戏行业而言，这一实践仍然没有得到广泛的应用。

本文提出了一个可配置的C++模板库，它的设计是为了减轻可重用软件组件的开发过程，并提供一些有用的特性支持，如工厂、接口、接口版本控制、类型识别策略和引用计数。

首先要介绍几个基本概念，它们是该组件库赖以生存的基础，而且这些概念有各自的用途可以分开独立使用。我们将通过学习这些内容为组件系统建立基础并验证其使用。

1.16.1 类型识别系统

该库依赖于一个在运行时进行类型识别的方法。由于大家对如何实现类型识别系统有着很多不同的偏好，所以我们会把它描述为组件库的一个可配置的参数。它是一个由用户定义的简单的类，但必须至少以下列格式来实现：

```
struct TypeID
{
    template<class Type> TypeID(const Type* type);
    TypeID();

    int operator==(const TypeID& typeID) const;
    bool operator<(const TypeID& typeID) const;

    static TypeID FromName(const char* const name);
};
```

这使得我们可以从类的指针来创建相关的 TypeID 对象，或者通过静态成员函数的方式，使用类的字符串名称来创建 TypeID 对象。它还使我们可以对两个 TypeID 对象进行等同性测试。当 STL 容器类使用这个类时，它还可以提供一个非常有用的排序方法。

类型识别参数是作为一个模板参数供库里的所有类使用的，所以这些类并不受特定类型识别系统的约束。随书光盘中提供了一个实现范例，请

参见头文件 RTTITypeID.hpp，这个例子使用了微软公司 Visual C++ 7 中的 C++ type_info 支持。

1.16.2 工厂

一种以将客户端代码与实际的组件细节隔离的方式创建组件的方法，是采用 Factory（工厂）设计模式[Gamma95]。如果临时在对象的更为通用的名称下查看组件，我们就可以设计一个 Factory 类，它的用处更广泛。

我们希望 Factory 对象要满足的第 1 个需求是：给定一个要创建的对象的类型，它就能创建出该对象的一个实例：

```
template<class TypeID, class Base>
struct Factory
{
    Base* Create(const TypeID& typeID);
};
```

从这段伪代码中看到的第 1 个问题是：我们必须要决定由工厂对象的 creation 方法返回的对象类型。对于这个实现，所有的对象都是从一个公共基类派生而来的，而这就是 creation 方法返回的类型，因为这样做就可以让这个要创建的对象的具体类更隐蔽地与客户端代码隔离开来。

第 2 个问题是，如何通知工厂我们希望创建的是哪个对象？我们只要将识别我们希望创建的对象的类型识别对象的一个实例传递给 creation 方法就可以了。

因为想把这个库设计得尽可能地通用，所以我们还会通过模板将一个类型识别参数传递给工厂类。前面讨论过这个参数，且 creation 方法返回的就是那个基类。这样就消除了对该组件库其他部分的不必要的依赖。

“喜欢什么随便选”，以这样的心态设计组件库是很关键的，这可以让用户在最终的产品中获得更多。一个是生命周期长且高产的泛型库，一个是框架不太灵活、需要不断重构的库，二者之间的区别就在于此。

现在可以添加几个 creation 方法的重载版本，以便使用出色的 C++句法，让我们可以更容易地使用工厂类：

```
template<class TypeID, class Base>
struct Factory
{
    Base* Create(const char* const name);
    Base* Create(const TypeID& typeID);
    template<class Type> Base* Create();
};
```

这样，就可以使用下列创建模式：

模式 1：

```
Factory factory;
TypeID typeID;
```

```
Base* base = factory.Create(typeID);
```

模式 2：

```
class MyObject;
// ...
Factory factory;
Base* base = factory.Create<MyObject>();
```

模式 3：

```
Factory factory;
Base* base = factory.Create("MyObjectName");
```

模式 4：

```
class MyObject;
// ...
MyObject* myObject = 0;
Factory factory;
Base* base = factory.Create(myObject);
```

前面 3 个创建模式应该是最有用处的。当编译的时候不知道要创建的对象时，模式 1 和模式 3 就显得尤为重要，这是数据驱动式应用程序不可或缺的功能。

看到这里，工厂类又如何知道该选择哪种模式来创建我们要求的对象呢？答案就是：工厂类必须要知道该怎么做。如果对于我们要求的对象，工厂类根本不知道该如何创建，它们的 creation 方法就会返回 NULL。我们将工厂类的定义进行了如下的扩展：

```
template<class TypeID, class Base>
struct Factory
{
    Base* Create(const char* const name);
    Base* Create(const TypeID& typeID);
    template<class Type> Base* Create();
    template<class Type> void RemoveSupport();
    template<class Type> void Support();
};
```

现在，有两个新的方法可以动态地向工厂类添加新的支持功能，使它支持我们希望的对象类型（在这里请大家注意，我们假设的前提是所有的类都有一个缺省的构造函数)。如果给定一个对象的实现，该对象是从相应的基类派生得到的，要想创建这个对象，我们就会调用 MyObject。可以这样在工厂类中添加对这个对象的支持：

```
class MyObject : public Base
{
    MyObject();
//...
};

Factory factory;
factory.Support<MyObject>();
```

现在，我们的工厂对象就可以支持这个类了，也就能够创建它的实例了。动态地添加或删除对某个特定对象类型的支持，这样的功能是非常有用的，但在其他工厂系统的设计中却不是很常见。

至于工厂对象是如何轻松地做到这一点的，这里就不详细叙述了，但是随书光盘中提供了相应的源代码。简而言之就是，工厂类会为它所支持的对象类型保留一个模板化构造函数对象的映射表。其中每个构造函数对象都知道该如何创建特定的对象类型。

1.16.3　工厂单例与子工厂

在工厂类中使用单例设计模式是非常有益的，原因有很多，最重要的原因就是：简单。我们只需要简单地将下述静态方法添加到工厂类中，就可以提供单例模式的应用：

```
struct Factory
{
    //...

    static Factory& Singleton()
    {
        static Factory factory;
        return factory;
    }
};
```

由于单例设计模式的使用，我们只需要将工厂对象引用为 `Factory::Singleton()`，就可以在任何地方访问工厂对象的惟一实例。这样的用法不但方便，而且更为安全。此外，我们可以增加对子工厂的支持，这意味着我们可以将所有的工厂类以层级的形式连接起来，以更简单的方式影响对象的创建。如果某个工厂类不知道该如何创建我们所要求的对象，我们就递归地向下遍历它的子工厂，直至找到一个知道该如何创建该对象的子工厂。为了提供这种功能支持，我们将工厂类进行了如下扩展：

```
struct Factory
{
    void RemoveChild(Factory* factory);
    void AddChild(Factory* factory);
};
```

通过这两个新的方法，工厂类现在可以维护一个列表，列表中保存的就是指向其子类对象（构造函数）的指针。在这个实现中，被销毁的子工厂类是不能被另外一个工厂对象引用的。当然了，可以对这个系统进行扩展，以对付这种情况。其方法是在工厂类中增加引用计数的功能（或其他类似的安全过滤机制）。

1.16.4　DLL 工厂

对于微软公司的 Windows 操作系统而言，我们可以使用动态链接库（Dynamic Link

Library，简称 DLL）进一步增强工厂类的功能，使它可以支持动态可加载工厂的概念。为此，我们首先要创建一个类——DLLFactory，它是从 Factory 派生而来的。我们将 DLL 的加载工作打包到这个新的类中，通过它来公开工厂类，并将加载的 DLL 与公开的工厂类进行绑定。DLL 只需要导出一个函数，返回一个指向需要公开的工厂对象的指针。请大家注意，通过子工厂的使用，一个 DLL 可以公开多个工厂对象。

与这部分内容相关的源代码，请参考随书光盘中的文件 DLLFactory.hpp。同样，我们也提供了一个它的应用实例。要注意的是，如果工厂类中的 DLL 被卸载了，而用它创建的对象可能还在被使用，这时候就会导致访问冲突。所以，在卸载 DLL 的时候，通过 DLL 工厂类创建的对象也必须被销毁。因此，从基类派生出来的那些所有可被工厂类创建的对象，都必须提供一个自毁函数。一个典型的实现如下所示：

```
struct Base
{
    void DeleteThis()
    {
        delete this;
    }
};
```

我们还可以实现更可靠、更复杂的功能来处理这种情况。可以要求工厂对象去跟踪它所创建的对象，让它们在这些对象的生命周期中扮演更主动、更活跃的角色。但是，这些内容已经不属于本文的讨论范畴了。

1.16.5 组件

现在，用来实现我们该组件库的基本元素已经到位了。接下来，就可以将精力集中在组件库的具体实现细节上了。从概念上讲，一个组件应该是一个代码单位，它的实现细节是透明于客户端代码的。我们应该通过接口的使用，将组件与客户端代码之间的通信抽取出来。在此有两个关键的概念：组件及组件接口。

我们必须用一些特定的方式来管理组件的生命周期。我们的工厂类提供了创建机制，但我们可以使用一个引用计数系统来跟踪管理组件，并提供这些组件被析构的详细信息。提供引用计数功能的类大致如下：

```
struct ReferenceCount
{
    //...

    void Reference();
    void Release();
};
```

至于 ReferenceCount 的完整实现代码，可以参考随书光盘中提供的文件 ReferenceCount.hpp。它所提供的一些方法可以分别自动增加或删减对象被引用的次数。当对象的引用计数为 0 时，对象就会自动将自己销毁。组件库里提供了一个 Component 基类，我们可以从这个

基类和 ReferenceCount 类派生出一个特定的 Component 类实现（这部分代码请参考文件 ExampleConfiguration.hpp）。除此之外，还可以在 Component 类中定义一个 Clone() 方法，形式如下：

```
virtual Component* Clone() { return 0; }
```

这就为我们提供了一个透明的方式，让客户端代码可以复制该组件。对于要实现的组件，此做法是可选的。即使如此，还是无法从组件库里获得更多的支持，因为我们只能创建或销毁某些对象而已。我们需要在组件上增加对接口的支持，真正把它们用起来。

1.16.6　组件接口

为组件添加接口的标准方法就是使用 QueryInterface() 方法。我们用想要获得的那个接口的一个标识符来调用这个方法，然后就能返回所需接口的一个指针，或者是 NULL：

```
struct Component
{
    template<class Type> Type* QueryForInterface();
    //...
};
```

可以使用类型识别系统来明确要找的接口，但是应该把相关事宜安排好，以便在需要的时候可以使用另外一个类。我们的组件类实际上有两个模板参数：一个是类型识别系统使用的参数，另外一个是接口识别系统使用的参数。

```
template<class TypeID, class InterfaceID >
struct Component
{
     //...
};
```

参数 InterfaceID 类与前面定义的类型识别类非常类似，其形式如下：

```
struct InterfaceID
{
    template<class Type> InterfaceID(const Type* type);
    InterfaceID();

    int operator==(const InterfaceID& interfaceID) const;
    bool operator<(InterfaceID const& interfaceID) const;
};
```

实际上，由于二者非常相似，所以如果愿意，也可以使用类型识别系统代替它。既然如此，为什么还要区分接口识别系统和类型识别系统呢？原因在于，这样做可以支持另外一个特性——接口版本管理。

1.16.7　接口版本管理

在该组件库中，接口版本管理功能是作为一个可选项来提供的，它的设计是为了解决在

代码开发过程中发生的接口调整问题。理想情况下，接口只需要定义一次，以后就不会有什么改变了。如果希望扩展或改变接口，我们会保留旧的接口，然后再增加一个新的接口来实现新的功能。

如果我们的讨论是定位于公众使用的官方代码发行版本，上述做法绝对是没什么问题的。但是，对于内部开发过程而言，接口调整的频率非常之高。如此频繁地增加新的接口确实不太实际，这种做法可以带来灾难性的后果：对于多个分别独立编译的代码单位，之前大家认为定义的接口都是一样的，但实际上它们却相差十万八千里。如果这些代码单位试图通过这些接口进行通信，其结果就是致命的。

为了避免这种窘境，最简单的解决方案就是为每个接口指定一个版本号。当要查询某个组件的接口时，接口识别系统就会读入这个版本号。当两个版本号吻合时，查询工作才算成功。当程序员修改了某个接口时，这个接口的版本号就会增加一个量。这比重新增加一个新的接口要容易得多。当然，也可以建立一个机制，自动地增加接口的版本号，从而在接口代码进行登记或有变动时，防止接口定义不兼容的问题。

为了实现给接口添加版本号的功能，我们可以从下面这个模板类派生出一个 Version 类，它的参数就是接口的版本号：

```
template<unsigned Number>
struct Version
{
    #ifdef _DEBUG
        enum { VersionNumber = -(int)Number };
    #else
        enum { VersionNumber = (int)Number };
    #endif
};
```

带有版本号的接口大致如下：

```
struct MyInterface : Version<2>
{
    //...
};
```

如果代码目前是在调试状态下进行编译，我们可以把所有接口的版本号设置成负值。这样做是非常有好处的，因为在某些情况下，我们不想让别人使用正在调试的组件，也不想让它们与那些正式版本的接口混为一谈（并把它们的使用标识为程序错误）。如果可以随意地混合使用调试版本和正式版本的组件，系统只会使用接口版本号的绝对值来进行接口识别的比较工作。这个功能也是可选的，可以通过一个编译时的设置项来控制。

更灵活的做法是为接口指定版本号，这也是可选的。如果没有指定版本号，缺省的版本号就是0或者其他的一个什么值。这就叫做“松散版本管理”。随书光盘中的文件 InterfaceID.hpp 里包含了3个接口识别类，让我们自己选择要采用哪种管理模式：严格版本管理、松散版本管理，还是不进行版本管理。举个例子，如果选择的是严格版本管理，就可以这样声明接口识别类：

```
typedef  InterfaceID_WithStrictVersion <TypeID> MyInterfaceID;
```

需要注意的是，这个类仍然需要一个基础的类型识别系统才能运作。随书光盘中提供的文件 RTTITypeID.hpp 就是一个多用途的类型识别系统。`InterfaceID` 这个类使用了 C++的一个编程技巧来实现版本管理功能，大家可以在提供的源代码中很明显地看到这一点。

1.16.8 定义组件及其接口

所有这些背景知识和准备代码仍然没有告诉我们如何才能实现一个组件和它的接口。为了更好地说明这个实现过程，并将组件和接口绑定在一起，下面来看一个直接的例子。

```
//定义一个接口
struct Movable
{
    virtual void GetPosition(float& x, float& y) = 0;
    virtual void SetPosition(float x, float y) = 0;
};


//组件的实现
struct Player : Component, Movable
{
    float x, y;

    Player ()
    {
        ExposeInterface<Movable>(this);
    }

    void GetPosition(float& x, float& y)
    {
        x = this->x;
        y = this->y;
    }

    void SetPosition(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
};
```

这里的关键是那个模板化的 `ExposeInterface()` 方法(它是从 Component 类继承而来的)，该方法带有一个指针，指向要公开的接口。基类 `Component` 则保留着一个从接口标识符到实际的接口指针的映射表。当要申请一个接口时，系统就会使用这些接口指针进行检索。

该系统还有一个很棒的特性，即接口的实现不一定非要通过继承来完成，我们还可以使用对象组合的方法。在很多情况下，后者更为可取。例如，对于前面实现的那个组件，也可以选择这样的实现方法：

```
//定义接口
struct Movable
{
    virtual void GetPosition(float& x, float& y) = 0;
    virtual void SetPosition(float x, float y) = 0;
};


//定义一个接口实现
struct MovableHelper : Movable
{
    float x, y;

    void GetPosition(float& x, float& y)
    {
        x = this->x;
        y = this->y;
    }

    void SetPosition(float x, float y)
    {
        this->x = x;
        this->y = y;
    }
};


//组件的实现
struct Player : Component
{
    MovableHelper movableHelper;

    Player ()
    {
        ExposeInterface<Movable>(&movableHelper);
    }
};
```

1.16.9 组件的使用

实际上，一个工厂对象就可以创建出前面定义的组件，只需要一行代码，就可以将它公开给工厂类单例：

```
Factory::Singleton().Support<Player>();
```

然后，使用该组件时，通常可以这样编码：

```
Component* component = Factory::Singleton().Create<Player>();

if(component)
```

```
{
        Movable* movable = component->QueryForInterface<Movable>();

        if(movable)
            movable->SetPosition(3.2f, 4.0f);

        component->Release();
}
```

这里要注意的是，与其他组件系统不同，如微软公司的COM（组件对象模型，Component Object Model），我们并没有通过组件的接口来实现对组件生命周期的管理。由于这样做不会因为类的继承将接口束缚到组件上，所以分离式的做法可以为组件的实现提供更大的灵活性。

1.16.10　配置组件库

一般来讲，在进行某些特定的配置时，本文这个组件库的使用者会使用 `typedef` 来定义新的类型，这主要是因为模板库类的句法相对比较复杂。即使是在本文提供的示范代码中，为了清晰起见，我们也省略了某些模板参数。在随书光盘提供的源代码中，头文件 ExampleConfiguration.hpp 给出了一个 `typedef` 的用例。

1.16.11　总结

本文向大家展示了一个可以很容易地向可重用的软件组件添加结构化代码的泛型组件库。由这个组件库可以很容易地形成一个现成的插件基础框架，供其他很多应用程序使用。与其他同类系统的设计相比，该组件库提供的句法更为简练。

感谢 Achim Stremplat，他为我们提供了接口管理的思想。也非常感谢 Alan McDonald，他给我提供了很多很好的建议。

1.16.12　参考文献

[Gamma95] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley Professional, 1995.

1.17 选择自己的路线——菜单系统

Wendy Jones
wendy2032@yahoo.com

如何实现一个游戏的菜单系统？很多时候，大家会挺到项目的尾声阶段才随意地回答这个问题，这种做法通常会导致菜单系统的实现非常草率。开发人员通常都会对那些更具挑战性或者更有紧迫感的问题，感到莫名的兴奋。但是，就游戏的全面理解和感知以及游戏的品质而言，菜单系统是至关重要的。玩家坐下来玩游戏，他第一眼看到的就是游戏的菜单。如果最初的游戏体验是面对一个很难操控的用户界面，那么玩家对游戏的感知，连同那些有趣的游戏元素，都会在瞬间烟消云散。

本文的焦点就是告诉大家如何去设计、构建一个灵活的、可扩展的通用菜单系统。

1.17.1 为什么需要菜单系统

很多情况下，菜单的添加工作都非常匆忙，不太会去考虑良好的设计或者代码重用性。当下一个项目临近结束时，我们只好从头开始，匆忙地设计游戏菜单。游戏菜单的开发制作确实是一个非常痛苦的工作。但是，赛车手需要选择自己的赛车，阻击手需要选择自己合手的武器，所以游戏菜单是非常必要的。对于一个菜单系统，我们也需要从良好的设计入手，让自己的生活更从容些。

设计一个可以随着不同的项目规模灵活扩展且可以重用的菜单系统，并不是件非常困难的事情。游戏的菜单基本上只需要执行以下几个基本任务：

- 显示一系列的可选项目
- 让用户使用既定输入方法，轻松地选择各种选项
- 根据用户的选择，依次显示下一个菜单

如果只考虑这些任务，我们很容易就知道该如何把它们转换成菜单系统的几个部件。那就让我们花些时间，把这些任务分解成抽象组件，再了解一下它们各自的用途和功能。

首先是菜单组件。可以把菜单组件看成是一个单屏的选项。例如，游戏里第 1 个菜单通常会向玩家显示一些高层次的选项，如开始、设置或玩家个数。这一系列选项就构成了游戏中一个单一的菜单组件。游戏中每个后续的菜单又可以组成另外的菜单组件。

PDG

下一个组件是用户控件。我们很熟悉视窗环境中的那些典型的常用控件，如按钮、列表框、滑块等等。这些控件为用户与系统之间的交互提供了简单、便捷的方式。每个菜单组件都聚集了一系列的控件，来搜集来自用户的输入信息。

最后一个组件是菜单管理器（*menu manager*）。菜单管理器是菜单系统最主要的控制者，它会在必要的时候创建菜单，搜集来自菜单的信息，跟踪用户遍历菜单系统的路线。菜单管理器的简易、雅致，不但可以让用户界面的设计人员创建复杂的菜单路径，同时还可以将开发时间最小化，让程序员只需进行一次性的时间投资。菜单管理器会跟踪用户在菜单系统中的点选路线，加载并显示必需的菜单组件。

1.17.2 菜单系统的对象

我们已经大致了解了 3 个主要组件及其相关的任务，现在该考虑实现的细节问题了，例如我们需要哪些类（class）呢？

菜单系统包含 3 个主要的类，每个类都基于一个菜单任务：menuScreen 类、menuControl 类和 menuManager 类。

1. menuScreen 类

menuScreen 类是所有要显示菜单的基础。这个抽象类包含一组纯虚函数并提供了一个共同的接口，所有菜单都必须遵循这个接口。通过强制菜单遵循这个接口，菜单系统就不用去了解每个菜单的工作细节，我们也就可以快速地创建新的菜单，并且可以非常容易地将之插入到菜单系统中。为了充分利用这个特性，系统中所有的菜单都必须从 menuScreen 这个类继承而来。

menuScreen 类包含 3 个虚函数：init()、update() 和 render()。每个具体菜单组件类都必须实现这些函数。

init() 函数提供了一个单独的空间，用于加载某个菜单的图片，并创建所需的控件。我们应该在菜单显示之前调用 init() 函数。

update() 函数负责处理用户输入，并更新所有屏显项目的状态。在菜单显示出来之前，每一帧都需要调用一次 update() 函数。最后，render() 函数会执行实际的绘制菜单任务，在屏幕上把菜单显示出来。在 render() 函数里，我们可以修改菜单显示的顺序。

menuScreen 类也包含 loadBackground() 函数，以及一个指向背景图片的指针。因为所有菜单通常都包含一个背景图片，为了访问的方便，我们会把这个函数放置到 menuScreen 类中。保存在父类中的背景图片，在调用 render() 函数时才会显示出来。程序清单 1.17.1 显示的是 menuScreen 这个类的描述。

程序清单 1.17.1 menuScreen 类的描述

```
class menuScreen
{
public:
    //menuReturn 类的程序代码
    // update 函数会返回下列代码中的一个代码，以通知 menuManager 它的状态。
```

```
    // NONE - 没有动作，继续显示当前的菜单
    // NEXT - 应该结束当前的菜单，并转到下一个菜单
    // PREV - 结束当前的菜单，显示上一个菜单
    // POPUP - 菜单请求显示一个弹出式菜单
    // END  - 这是最后一个菜单，菜单结束
    static enum menuReturn { NONE=0, NEXT, PREV, POPUP, END };

    menuScreen(void);
    virtual ~menuScreen(void);

    // 加载这个菜单所需的全部资源
    virtual bool init(void) = 0;

    // 调用每个帧来更新菜单
    virtual int update(BYTE keys[]) = 0;

    // 调用每个帧来画出当前的菜单
    virtual void render(void) = 0;

    // 返回一个字符串来表示下一个菜单的名字
    std::string& getNextMenu(void);

protected:
    // 加载背景图片
    bool loadBackground(std::string imageName);

    // 所有菜单都有一个相关联的背景图片
    resourceImage *bkgrdImg;

    // 下一个菜单的名字
    std::string nextMenu;
};
```

2. 添加用户控件

采集用户输入所需要的用户控件是由 menuControl 类来创建的。MenuControl 是作为一个抽象类来实现的。具体菜单类中使用的所有控件都要从这个类中继承它们的基本功能。

menuControl 类包含大部分类型的用户控件的常见信息，如控件的位置、控件的类型和当前的状态。MenuControl 类还提供了一个 render() 函数，再一次使得系统可以一视同仁地对待所有控件。另外，menuControl 类还包含了可以改变可视状态的函数，让我们可以全面控制某个菜单对象是否要被显示出来。程序清单 1.17.2 包含了 menuControl 类的描述。

程序清单 1.17.2　menuControl 类的描述

```
class menuControl
{
public:
    static enum controlType {
```

```
        NONE   = -1,
        BUTTON = 0,
        STATIC,
        SLIDER,
        LIST
    };

    static enum controlState { ACTIVE=0, DISABLED };
    static enum controlView { VISIBLE=0, HIDDEN };

    menuControl(void);
    virtual ~menuControl(void);

    virtual void render(void) = 0;

    void setControlXY(int X, int Y) { locX = X; locY = Y; }
    int getType(void) { return type; }

    void activateControl(void) { state = controlState::ACTIVE; }
    void disableControl(void)  { state = controlState::DISABLED; }
    bool getControlState(void) { return state; }

    void showControl(void)    { view = controlState::VISIBLE; }
    void hideControl(void)    { view = controlState::HIDDEN; }
    bool getControlView(void) { return view; }

protected:
    //控件属性
    int type;              // 控件的类型
    int locX;              // X坐标位置
    int locY;              // Y坐标位置

    bool state;            // 控件是否激活
    bool view;             // 控件是否可见
};
```

3. 全面控制：menuManager 类

正如前面的描述所暗示的，menuManager 类是整个菜单系统中干粗活的真正劳动者。由于是以单件模式来实现的，menuManager 可以确保只创建一个实例，作为惟一的控制中心来创建和显示菜单，并跟踪用户在菜单系统中的行进路线。menuManager 包含 2 个主要的部分，一个是管理器本身，一个是菜单工厂。

4. 管理器

根据具体的游戏产品，菜单的数量有可能很快变得非常庞大、难以控制。菜单管理器要阻止这样的事情发生。为了完成这个任务，管理器会保存一个维护已访问菜单的列表。这个列表里的最后一个菜单会被看做是当前的菜单。在更新或显示一个菜单时，管理器只需要简单地访问列表中的最后一个对象，并传递相应的命令。

菜单列表，也称为菜单访问踪迹表，是菜单管理器跟踪的用户所访问过的菜单序列。有时，用户可经由多个不同的路径到达同一个菜单，这样就很难保证让用户返回到正确的上一个菜单屏幕。菜单访问踪迹表可以解决这个难题。如果用户想从菜单上原路返回，管理器只需要把菜单访问踪迹表倒转过来就可以了。

到目前为止，我们只讨论了菜单管理器是如何使用菜单访问踪迹表中的菜单的。在下面的文章中，我们看一下如何创建菜单。

程序清单 1.17.3 显示的是 menuManager 类。

程序清单 1.17.3 menuManager 类

```
class menuManager
{
public:
    // 单体模式确保菜单系统只有一个实例
    static menuManager& getInstance()
    {
        if (pInstance == NULL) pInstance = new menuManager();
            return *pInstance;
    };

    // 初始化菜单系统
    bool init(void);

    // 关闭并释放菜单系统
    void shutdown(void);

    // 向菜单下传更新消息
    int update(BYTE keys[]);

    // 画出菜单
    void render(void);

private:
    static menuManager *pInstance;

    menuManager(void);
    ~menuManager(void);

    menuScreen *popupMenu;

    // 用该标志来跟踪一个弹出窗口，判断它是否是处于激活状态
    bool popupActive;

    // 菜单访问踪迹表会跟踪用户访问过的所有菜单。这样，系统就有能力回溯用户
    // 访问过的菜单。
    std::vector<menuScreen*> menuTrail;

    // 菜单工厂的实现
    #define REGISTERMENU(a) registerMenu(#a, &CreateObject<a>);
```

```
    std::map<std::string, menuCreateFunc> menuList;

    // 向菜单工厂注册一个菜单的函数
    void registerMenu(std::string menuName,
                      menuCreateFunc menuFunc);

    // 创建一个新菜单
    menuScreen* createMenu(std::string menuName);

    // 从系统中删除一个菜单
    void destroyMenu(menuScreen* menu);
};
```

5. 菜单工厂

菜单工厂的工作内容是按需创建菜单。当需要一个新菜单时，菜单工厂就会收到一个创建菜单的请求。大多数情况下，在一个单一的会话中，我们只需要显示部分的菜单，而不必显示全部的菜单。举个例子，如果某人参与到游戏中来，他们很可能不会先看选项或积分榜菜单。菜单工厂会控制这些不用显示的菜单，不将它们实例化，以节约时间和内存。

我们用一系列的函数指针和一个标准模板库（Standard Template Library，简称 STL）映射表来实现菜单工厂。该映射表保存着一个指向菜单构造函数的指针。每一个菜单都有其对应的构造函数。这样，在收到请求之后，菜单管理器就会把菜单的名字（字符串）作为查找键值，来访问这些构造函数。在一个新的菜单被创建出来之前，菜单管理器会向菜单工厂发送这个新菜单的名字。菜单工厂会在 STL 映射表中查找这个名字字串，并以此来创建新的菜单。然后，菜单工厂会把这个新菜单（系统要求创建的菜单）的指针返回给菜单管理器，以便日后使用。图 1.17.1 展示了这个过程。

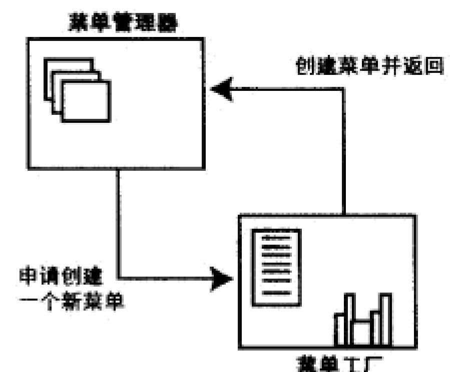


图 1.17.1 菜单工厂创建菜单的方式

在大多数菜单系统中，菜单的创建是通过一个开关语句来控制的。每个菜单的构造函数都隐藏在一个 label 后面，开关语句会使用这个 label 来判定需要创建哪个菜单。对于那些只有一个新菜单的系统，这样做行了。但是随着菜单数量的增加，这个方法很快就会变得难以维护。通过使用菜单工厂，我们可以把添加新菜单所必需的代码量压缩到最小。

程序清单 1.17.4 显示的是工厂实现。

程序清单 1.17.4 菜单工厂的实现

```
typedef menuScreen *(*menuCreateFunc)();
typedef std::map<std::string, menuCreateFunc>::iterator Iterator;

template<typename ClassType>
menuScreen *CreateObject()
{
    return new ClassType;
}
```

```
void menuManager::registerMenu(std::string menuName,
                              menuCreateFunc menuFunc)
{
    menuList[menuName] = menuFunc;
}

menuScreen* menuManager::createMenu(std::string menuName)
{
    //在注册菜单列表中找到被请求的菜单
    Iterator iter = menuList.find(menuName);

    //如果菜单不在列表中，就返回NULL
    if (iter == menuList.end())
        return NULL;

    //second是相关的值，first是键值
    //这会生成构造函数
    return ((*iter).second)();
}

void menuManager::destroyMenu(menuScreen* menu)
{
    if (menu)
    {
        delete menu;
        menu = NULL;
    }
}
```

6. 可扩展性

本文涉及的组成该系统的所有类，都是采用跨平台的设计，只需微小的改动就可以达到跨平台的兼容特性。由于每个游戏的需求不同，这里展示的菜单系统只是一个非常好的基础系统，大家可以在此基础上进行扩展。这里提供几个对系统功能进行扩展的建议。

脚本支持：可以用外部文件描述每一个菜单，然后在运行时把它们加载进来。控件的位置以及它们的行为也可以用脚本文件描述。这样就可以用一个完全动态的方法创建菜单。

新增用户控件：新控件的添加非常容易，可以把它们集成到系统中，以便让菜单能够以任何必要的方式来运转。

多个输入设备：现在该系统只能支持键盘的输入，但是简单地做一些修改，就可以让它提供对游戏杆和鼠标的支持。

1.17.3 总结

在随书光盘中可以找到该菜单系统的示范代码。为了简单起见，我们提供的程序代码采用了 Microsfot Windows GDI 来完成图形图像操作。但是我们可以很容易地改造这个系统，以

使其支持 OpenGL、DirectX 或其他图形图像 API。正如大家所看到的，创建一个强大的菜单系统并不是件非常复杂的事情。只要遵循一个简单、但可以灵活扩展的设计方案，这个系统就可以很好地适应需求的变化，且不需要太多或者根本就不需要什么额外的付出。

1.17.4　参考文献

[Gamma95] Gamma, Erich, et al. *Design Patterns*. Addison Wesley, 1995.

第 2 章

数学

PDG

引 言

Naughty Dog 公司 Eric Lengyel
lengyel@terathon.com

很多现代的视频游戏，其背景故事都发生在某种模拟的虚拟环境中。这些游戏试图让环境像一个真实的世界那样呈现和运行，至少也要在一个可接受的帧率上，达到计算机硬件所允许的真实度。要显示一个逼真的 3D 环境，模拟遵守物理法则的动态系统，程序员首先需要理解为现实世界中各种事物建模所使用的数学原理。

从总体上讲，数学是一系列无休止的概括和归纳——我们已经知道了很多，但还有很多等待着我们去发现。在某个层面上以各自独立的方式来应用的两个看似不同的数学概念，可以在某个更高的思维层面上统一起来，将其想成或看做是一个广义概念的两种情况。本章的第 1 篇文章是由 Chris Lomont 撰写的，讨论了数学的一个分支——几何代数，并介绍了最常用的多重矢量运算，如旋转和交叉乘积。

在游戏的开发过程中，某些数据点通常需要平滑地进行插值。这些数据点的例子有：摄像机位置数据、动画的关键帧、低分辨率网格的顶点。执行平滑的插值操作的一个常用工具是样条（spline），通常使用的是三次样条插值。本章接下来的两篇文章都和样条有关。首先是 Tony Barrera、Anders Hast 和 Ewert Bengtsson 的文章，他们提出一个可以在保持平滑插值的同时，最小化 Hermite 曲线的曲率的技术。第 2 篇文章是由 James Van Verth 撰写的，探讨了样条在动画控制中的应用。

虽然游戏引擎的目标是竭尽所能模拟出逼真的环境，但逃不掉的结果是：在很多不同的层面上都只能做到“近似”。在计算领域，精度和速度是两股对立的力量，需要以最适合应用的方式进行平衡。下面的两篇文章提供了几个近似计算法，目的是在获得最佳速度的同时尽可能少地牺牲精确性。首先是 Andy Thomason 描述了一个四元插值的近似算法，接下来 Christopher Tremblay 讨论了一个极大极小逼近算法。

最后是以 Eric Lengyel 的文章来结束本书“数学”部分的内容的。Eric Lengyel 介绍了一个修改投影矩阵的技巧。透视投影矩阵背后的数学和齐次裁剪空间（homogeneous clip space）的属性，有些时候是无法靠直觉来感知的，但是对它们本质的深层次理解可以给我们带来很多有用的调整。最后一篇文章中介绍的技术是让视锥（View Frustum，沿用《游戏编程精粹 4》的译法）产生变形，以使近处的平面可以被任意一个裁剪平面替换，此外文中还介绍了如何最小化景深缓存精度上所受的影响。

2.1 在计算机图形学中使用几何代数

Chris Lomont
Clomont@math.purdue.edu

几何代数（Geometric Algebra，简称 GA）是表示很多几何问题的一种简明方法，它对计算机图形学很有用。它允许我们以更加一致的方式进行计算，简化公式的推导和算法的设计。

2.1.1 引言

几何代数提供了一种单一的语言，该语言统一了计算机几何学的很多领域，并被应用到物理、计算机视觉、微分几何和其他领域中。在图形学中，几何代数的作用就是以一种非常简练的方式来处理很多看上去没有联系的东西，如线性代数、在任意维空间中的旋转（包括四元数）、子空间中的相交、普吕克（Plücker）空间、点积和叉积及至微积分学和曲面的表现。例如，几何代数可以用一个方程来表示任意两个子空间的交集：直线和直线、直线和平面、平面和平面，等等。大多数的求交方程都需要一些特殊的条件，且不能扩展到更高维中。这仅仅是几何代数简化几何问题的概念和思考方式的部分应用。在继续本节之前，有些读者可能希望先阅读本文末尾的例子，以更好地体会几何代数所带来的便利。

回想一下，我们掌握线性代数花了多长时间，然后是四元数（诚然，它依然是很神秘的），所以不要认为经过十几分钟的阅读就能对几何代数有直觉上的认识并有效地运用它。虽然几何代数开始的时候看起来和四元数一样神秘，但它是几何计算的一个有力工具，前期花时间掌握它，在以后的几何计算过程中就可以获得很大的回报。下面。我们就通过一个例子来了解几何代数的一些本质特征。

启发性范例

为了演示几何代数，我们先从一个用 x-y 平面表示的 2D 向量空间开始看一个简单的例子。如果仅仅把点（x_1, y_1）和（x_2, y_2）当做“点”来对待的话，并不能给计算带来更大的便利，除非把它们当成向量（每个点定义了一个从原点指向该点的箭头）来思考。我们可以用一种自然的方式对向量做加法操作，这为平面提供了更高水平上的结构，并为解决问题提供了额外的工具。接下来，我们可以给点（或者向量）定义一个乘法操作，让它的行为能和加法很好地协调起来。定义的方式有很多，我们选择把每个

点看成一个复数，换句话说，就是把(x_2, y_2)看成$x_1+i\,y_1$。定义$i^2=-1$，就可以通过$(x_1+iy_1)(x_2+iy_2)=(x_1x_2-y_1y_2)+i(x_2y_1+x_1y_2)$进行乘法运算，其结果是另外一个点或者向量。这种处理点和向量的方法非常强大，因为这种乘法是可逆的，即我们可以除以一个点（即一个向量或者复数）。在这种处理方式的基础上，我们可以做大量的计算。乘法的这种定义非常直观（经过一定的学习以后），因为点的乘法和加法与实数的乘法和加法非常类似。

注意，从(x_1, y_1)到$x + iy$的映射会把基$\{e_1=(1,0)\}$，$\{e_2=(1,0)\}$映射到复数 1 和 i 上。所以我们可以认为构造了一个以 1（实部）和 i（虚部）为基的空间，也就是说 x-y 平面的另外一个基是$\{1,i\}$。

这种表示方法的一个非常有用的例子就是：我们可以通过乘法来旋转一个点（或者整个图形）。点（x_1, y_1）乘以点 $\exp(i\theta) = \cos\theta + i\sin\theta = (\cos\theta, \sin\theta)$等于把点（$x_1$, y_1）旋转了一个角度θ。向量乘法的几何意义是：乘以一个点意味着旋转后再拉伸，图 2.1.1 推出了方程 2.1.1。

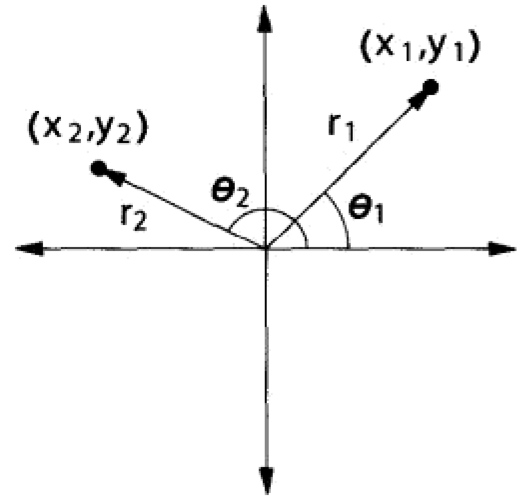


图 2.1.1 复平面上定义的 2D 实空间中的向量乘法

$$(x_1,y_1)(x_2,y_2)=\left(r_1e^{i\theta_1}\right)\left(r_2e^{i\theta_2}\right)$$
$$=r_1r_2e^{i(\theta_1+\theta_2)} \quad (2.1.1)$$

该方程表示把点（x_1, y_1）旋转一个角度，角度的大小是 x 轴到向量（x_2, y_2）之间的夹角，然后再拉伸向量（x_2, y_2）的长度 r_2。从 2D 空间中得到这个公式的，这意味着我们可以把 2D 空间中的任意两个元素相乘，得到 2D 空间中的另外一个元素。

现在，我们把这个想法扩充到任意维上。

2.1.2 几何代数

我们要修正一些符号来让表达更加准确：用希腊字母α，β，γ和δ表示实数；用粗体的小写字母 **a**，**b**，**c** 表示标准向量；后面用粗体的大写字母 **A**，**B**，**M**，**R** 表示多向量（*multivector*），这是几何代数中的一般元素，如同向量是向量空间中的一般元素那样。

1. 代数

首先给定一个 n 维向量空间 V^n，然后在它上面构造一个新的向量空间 CV（C 代表 Clifford，也就是 Clifford Algebra），即 V 的几何代数空间。我们通常认为 V 是一些向量的集合，表示一维的有向数量（magnitude）。几何代数将推广这个概念，来处理所有的二维有向面积和三维有向体积，等等。下面就从外积（*outer product*）开始定义 CV 中的对象。

2. 外积

对于向量空间 V 中的两个向量 **a** 和 **b**，我们可以定义一个二重向量（*bivector*），写成 $\mathbf{a}\wedge\mathbf{b}$，表示由 **a** 和 **b** 两个向量张成的平面、平面的方向和由 a 和 b 定义的平行四边形的面积的大小。大致的概念请参见图 2.1.2（c）。二重向量也叫 2-blade 或者 2-vector，运算符$\wedge$称为外积（*outer-product*，也称为楔积 wedge-product 或 exterior product），它对标量α，β，γ以及向量 **a**，

b，**c** 满足方程 2.1.2 中的属性：

$$
\begin{aligned}
&\lambda \wedge \mathbf{a} = \mathbf{a} \wedge \lambda = \lambda \mathbf{a} && \text{向量和标量满足交换律} \\
&\mathbf{a} \wedge \mathbf{b} = -\mathbf{b} \wedge \mathbf{a} && \text{向量和向量满足反交换律} \\
&(\lambda \mathbf{a}) \wedge \mathbf{b} = \lambda (\mathbf{a} \wedge \mathbf{b}) && \text{和标量满足结合律} \\
&\lambda (\mathbf{a} \wedge \mathbf{b}) = (\mathbf{a} \wedge \mathbf{b}) \lambda && \text{和标量满足交换律} \\
&(\alpha \mathbf{a} + \beta \mathbf{b}) \wedge \mathbf{c} = \alpha \mathbf{a} \wedge \mathbf{c} + \beta \mathbf{b} \wedge \mathbf{c} && \text{双线性} \\
&\mathbf{a} \wedge (\alpha \mathbf{b} + \beta \mathbf{c}) = \alpha \mathbf{a} \wedge \mathbf{b} + \beta \mathbf{b} \wedge \mathbf{c} && \text{对每个因子满足线性}
\end{aligned}
\tag{2.1.2}
$$

扩展外积，以相同的形式计算 3 个或者更多向量的乘积。我们发现一个重要的性质，即对任一向量 **a**，可从反交换律得到 $\mathbf{a}\wedge\mathbf{a} = -\mathbf{a}\wedge\mathbf{a}$。因为只有 0 取负后才和本身相等，所以必然有 $\mathbf{a}\wedge\mathbf{a}=0$，结果为一个实数。

外积的直观几何表示见图 2.1.2。向量 **a** 是一个有大小和方向的一维对象。把这个概念扩展到标量上，将 **a** 看成是一个 0 维的大小为 α 的点。二重向量是一个有向面积（二维），它具有大小和方向。三重向量则为一个有向三维体积。这就是为什么外积是反交换律的原因。交换参与运算的向量的顺序，将会改变运算结果的方向。独特的只是子空间、大小和方向，而不是平行四边形的形状。例如，下面的双重向量是相等的，但是定义它们的边却是不相同的：

$$(2\mathbf{a})\wedge(3\mathbf{b})=(6\mathbf{a})\wedge\mathbf{b}=(3\mathbf{a})\wedge(2\mathbf{b})=6(\mathbf{a}\wedge\mathbf{b})$$

现在用一组正交基 $\{\mathbf{e}_1, \mathbf{e}_2, \cdots, \mathbf{e}_n\}$ 来简化表示。

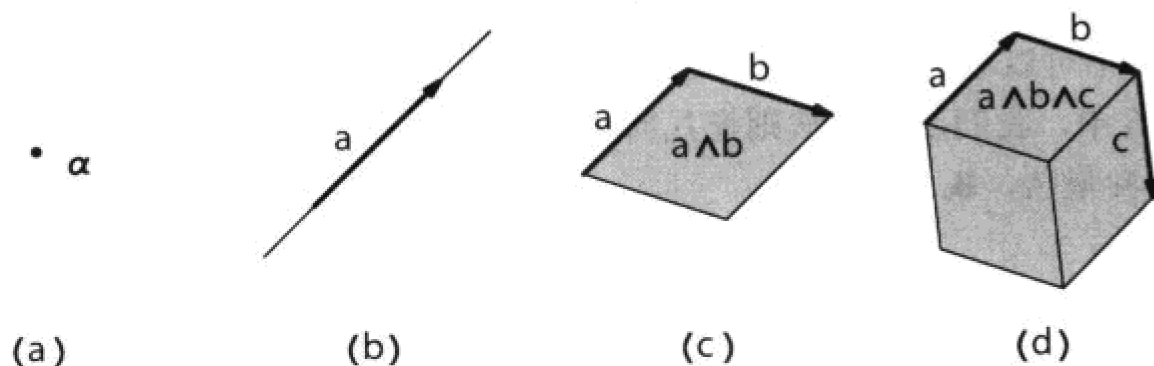


图 2.1.2　认识外积的一种方法是使用长度、面积和体积元素

定义与向量空间 V 关联的几何代数向量空间 CV，该空间的基包含了向量空间 V 的所有基的外积，以及标量 1。例如，V 是一个三维的向量空间，则 CV 的基如方程 2.1.3 所示。我们不可能有其他基了，因为如果一个基出现两次，那么外积的结果肯定是 0。根据只要交换临近的 $\mathbf{e}_i$ 就会导致结果取负这一事实 $\mathbf{e}_i \wedge \mathbf{e}_j = -\mathbf{e}_j \wedge \mathbf{e}_i$，把下标按照递增顺序进行排列：

$$\left\{ \underbrace{1}_{\text{标量}}, \underbrace{\mathbf{e}_1, \mathbf{e}_2, \mathbf{e}_3}_{\text{向量空间}}, \underbrace{\mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_1 \wedge \mathbf{e}_2, \mathbf{e}_1 \wedge \mathbf{e}_2}_{\text{双向量空间}}, \underbrace{\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3}_{\text{三向量空间}} \right\} \tag{2.1.3}$$

所以说向量空间 CV 是由方程 2.1.3 中基元素的所有线性组合构成的，如果 V 的维数是 n，那么 CV 就是一个 2^n 的空间（验证此结论是一个很好的练习）。CV 中的向量被称为多重向量（*mulitvector*），以区别于 V 中的普通向量。例如，CV 中的一个多向量可以是 $\mathbf{A}=3+2\mathbf{e}_1+4\mathbf{e}_1 \wedge \mathbf{e}_2 \wedge \mathbf{e}_3$。$CV$ 中的任何两个多重向量都可以用前面提到的法则进行外积运算。下面就定义一些其他的术语。*k-blade*[1] 是 k 个向量的外积，它不一定必须是 $\mathbf{e}_i$ 的积。例如，$(\mathbf{e}_1+2\mathbf{e}_2)\wedge(3\mathbf{e}_1-\mathbf{e}_2)$ 是一个 2-blade。*k-vector* 是 *k*-blade 的和，所以 *k*-blade 肯定是 *k*-vector，反之则不一定。k 称为 blade 的阶（*step* 或者 *grade*）。多重向

[1] 在其他的数学领域中，它也被称为 k-形式，CV 是外代数。我们使用 GA 的术语是因为更适合用来表达其几何意义。

量是几何代数中的基本元素，它是 *k*-blade 的和，有区别的只可能是 *k* 的值。因此，二重向量 $\mathbf{a}\wedge\mathbf{b}$ 是一个 2-blade，阶为 2。需要注意的是，向量（*V* 中的向量）也是一个多重向量。

举个例子，如果 $\mathbf{a}=\alpha_1\mathbf{e}_1+\alpha_2\mathbf{e}_2$ 且 $\mathbf{b}=\beta_1\mathbf{e}_1+\beta_2\mathbf{e}_2$，我们就可以按公式 2.1.4 所示的计算：

$$\begin{aligned}\mathbf{a}\wedge\mathbf{b}&=(\alpha_1\mathbf{e}_1+\alpha_2\mathbf{e}_2)\wedge(\beta_1\mathbf{e}_1+\beta_2\mathbf{e}_2)\\&=\alpha_1\beta_1\mathbf{e}_1\wedge\mathbf{e}_1+\alpha_1\beta_2\mathbf{e}_1\wedge\mathbf{e}_2+\alpha_2\beta_1\mathbf{e}_2\wedge\mathbf{e}_1+\alpha_2\beta_2\mathbf{e}_2\wedge\mathbf{e}_2\\&=0+\alpha_1\beta_2\mathbf{e}_1\wedge\mathbf{e}_2-\alpha_2\beta_1\mathbf{e}_1\wedge\mathbf{e}_2+0\\&=(\alpha_1\beta_2-\alpha_2\beta_1)\mathbf{e}_1\wedge\mathbf{e}_2\end{aligned}\tag{2.1.4}$$

注意，标量部分是向量 **a** 和向量 **b** 围成的平行四边形的面积。

对一个 *n* 维空间来说，这个空间所有正交基的外积 $\mathbf{I}_n=\mathbf{e}_1\wedge\mathbf{e}_2\wedge\cdots\wedge\mathbf{e}_n$ 为伪标量（*pseudoscalar*），并被经常简写成 **I**。

3. 几何积

几何积在几何代数中是最重要的积，所以我们不用任何符号来表示这种运算，仅仅是把多重向量写到一起。首先，用向量空间 *V* 中的点积（也叫内积）和外积来定义两个向量 **a** 和向量 **b** 的几何积，如方程 2.1.5 所示：

$$\mathbf{ab}=\mathbf{a}\cdot\mathbf{b}+\mathbf{a}\wedge\mathbf{b}\tag{2.1.5}$$

其很吸引人的一个地方就是它是可逆的，所以如果 **a** 不为 0 的话，可以把 **ab** 除以 **a**，得到 **b** 的值；同样，除以 **b** 可以得到 **a** 的值。注意，**a** 的逆是 $\mathbf{a}^{-1}=\mathbf{a}/(\mathbf{a}\cdot\mathbf{a})$，且满足 $\mathbf{aa}^{-1}=\mathbf{a}^{-1}\mathbf{a}=1$。现在我们有了一个可以取消的积运算，这使得它与复数的乘积运算类似。

在方程 2.1.6 中，我们用下列法则把几何积扩展到了 *CV* 中的任意元素上：标量α，β、向量 **a**，**b** 和多重向量 **A**，**B**，**C** 等。

$$\begin{array}{ll}\alpha\beta\text{ and }\alpha\mathbf{b} & \text{通常的用法}\\ \alpha\mathbf{A}=\mathbf{A}\alpha & \text{标量计算}\\ \mathbf{ab}=\mathbf{a}\cdot\mathbf{b}+\mathbf{a}\wedge\mathbf{b} & \text{向量和向量}\\ \mathbf{A}(\mathbf{BC})=(\mathbf{AB})\mathbf{C} & \text{结合律}\\ \mathbf{A}(\mathbf{B}+\mathbf{C})=\mathbf{AB}+\mathbf{AC} & \text{左分配律}\\ (\mathbf{A}+\mathbf{B})\mathbf{C}=\mathbf{AC}+\mathbf{BC} & \text{右分配律}\end{array}\tag{2.1.6}$$

为了展示如何使用这些法则来进行计算，下面给出针对正交的简化计算方程——还有一些一般化的方程，但它们要复杂得多。任何一个向量都可以写成正交（*orthonormal*）基$\{\mathbf{e}_1, \mathbf{e}_2,\cdots,\mathbf{e}_n\}$的线性组合。我们可以把任意一个多重向量用 *CV* 的基来展开。根据向量的定义，可以很容易地得到方程 2.1.7：

$$\mathbf{e}_i\mathbf{e}_j=\begin{cases}\mathbf{e}_i\wedge\mathbf{e}_j=-\mathbf{e}_j\mathbf{e}_i, & \text{如果 } i\neq j;\\ 1, & \text{如果} i=j.\end{cases}\tag{2.1.7}$$

把 *k* 个基的几何积简写为 $\mathbf{e}_{i_1i_2\cdots i_k}=\mathbf{e}_{i_1}\mathbf{e}_{i_2},\cdots,\mathbf{e}_{i_k}$。对 *k* 个不同的基向量，我们不加证明就可以得到 $\mathbf{e}_{i_1i_2\cdots i_k}=\mathbf{e}_{i_1}\mathbf{e}_{i_2},\cdots,\mathbf{e}_{i_k}=\mathbf{e}_{i_1}\wedge\mathbf{e}_{i_2}\wedge,\cdots,\wedge\mathbf{e}_{i_k}$。所以，计算任意两个多重向量的几何积时，我们按照如下的步骤来处理：

（1）把多重向量展开成基向量的外积。

（2）如前所示，把外积写成几何积的形式。

（3）用几何积的结合律，把展开后的多重向量相乘，注意保持顺序。

（4）对每个乘积，交换临近的下标以得到想要的下标顺序。每一次交换不同的下标都会使符号产生变化，如果相邻的两个下标相同，则可以把这两个项用 1 代替。

（5）最后，写成每一项都是外积的简化形式（这一步需要惟一的下标形式，所以必须要在前面的步骤中交换下标，并用 1 来代替相应的项）。

这种计算必须要很好地掌握，因为它很基本。例如，考虑方程 2.1.8：

$$\begin{aligned}(\mathbf{e}_3+\mathbf{e}_2\wedge\mathbf{e}_3)(\mathbf{e}_1+\mathbf{e}_1\wedge\mathbf{e}_3)&=(\mathbf{e}_3+\mathbf{e}_{23})(\mathbf{e}_1+\mathbf{e}_{13})\\&=\mathbf{e}_{31}+\mathbf{e}_{313}+\mathbf{e}_{231}+\mathbf{e}_{2313}\\&=-\mathbf{e}_{13}-\mathbf{e}_{133}+\mathbf{e}_{123}+\mathbf{e}_{1233}\\&=-\mathbf{e}_1\mathbf{e}_3-\mathbf{e}_1\cdot 1+\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3+\mathbf{e}_1\mathbf{e}_2\cdot 1\\&=-\mathbf{e}_1\wedge\mathbf{e}_3-\mathbf{e}_1+\mathbf{e}_1\wedge\mathbf{e}_2\wedge\mathbf{e}_3+\mathbf{e}_1\wedge\mathbf{e}_2\end{aligned}\tag{2.1.8}$$

有了这些法则，就可以构造出一个 CV^3 空间中 8 个基的乘法表，这是一个很好的练习。现在可以用手工方式计算出整个 CV 代数系统中任意两个元素的外积和几何积。注意，一般情况下几何积都包含很多项：一个 m-blade 和 k-blade 的几何积可能会包含阶为 $|m-k|$ 到 $m+k$ 的所有项。

4．收缩积

我们要用的最后一个乘积是收缩积（contraction product）。它以一种非常适合计算机图形学的方式把两个向量的内积（也称为点积）的概念扩展到了两个多重向量的内积上。这有时候称为左投影（*left projection*），还有一种右投影（right projection），它是双重左投影，但是本文不做介绍。这种乘积用于把空间投影到另一个空间上，和用内积把一个向量投影到另一个向量上非常相似。收缩积用符号⌋表示，并用和外积一样的形式根据阶对多重向量进行展开。它读为“a 收缩到 b”中或者“a 左收缩到 b”。

我们需要一些符号：对一个 k-blade $\mathbf{A}=\mathbf{a}_1\wedge\mathbf{a}_2\wedge\cdots\wedge\mathbf{a}_k$ 中，我们把其颠倒项记为 $\tilde{\mathbf{A}}=\mathbf{a}_k\wedge\mathbf{a}_{k-1}\wedge\cdots\wedge\mathbf{a}_1=(-1)^{k(k-1)/2}$。对一个多重向量 $\mathbf{A}$，我们把它的阶为 r 的部分记为 $\langle\mathbf{A}\rangle_r$；在不引起混淆的情况下，有时候也记为 $\mathbf{A}_r$。

多重向量收缩积的一般定义如下：

$$\mathbf{AB}=\sum_{r,s}\langle\mathbf{A}_r\mathbf{B}_s\rangle_{s-r}$$

这表示什么意思呢？为了计算一个 r-blade $\mathbf{A}$ 和一个 s-blade $\mathbf{B}$ 的收缩积，我们要先计算出几何积，然后取它的阶为 $s-r$ 的部分，它有可能为 0。当 $r>s$ 的时候，收缩积为 0，因为没有阶数小于 0 的元素。为了计算收缩积，需要把多重向量按 blade 展开并独立地计算每一个组合。在几何上，两个 blade 的收缩积 $\mathbf{A}⌋\mathbf{B}$ 会得到包含在 $\mathbf{B}$ 里的一个 blade，它和 $\mathbf{A}$ 垂直。图 2.1.3 演示的是向量 $\mathbf{a}$ 收缩到平面 $\mathbf{B}$ 中的一个例子。

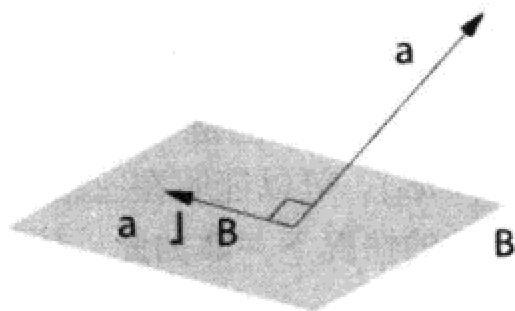


图 2.1.3　收缩积把一个空间投影到另一个空间的垂直分量上

对向量来说，收缩积需要被简化成点积，并且对所有分量都保持线性关系。从这些需求可以得出方程 2.1.9 中的关系：

$$
\begin{array}{lll}
\alpha \rfloor \beta = \alpha\beta & \text{标量} & \\
\mathbf{a} \rfloor \beta = 0 & \text{向量和标量} & \\
\alpha \rfloor \mathbf{b} = \alpha\mathbf{b} & \text{标量和向量} & \\
\mathbf{a} \rfloor \mathbf{b} = \mathbf{a}\cdot\mathbf{b} & \text{向量的点乘} & (2.1.9) \\
\mathbf{a} \rfloor (\mathbf{b}\wedge\mathbf{C}) = (\mathbf{a}\rfloor\mathbf{b})\wedge\mathbf{C} - \mathbf{b}\wedge(\mathbf{a}\rfloor\mathbf{C}) & \text{展开法则} & \\
(\mathbf{A}\wedge\mathbf{B})\rfloor\mathbf{C} = \mathbf{A}\rfloor(\mathbf{B}\rfloor\mathbf{C}) & \text{分配法则} & \\
\mathbf{A}\wedge(\mathbf{B}\rfloor\mathbf{C}) = (\mathbf{A}\rfloor\mathbf{B})\rfloor\mathbf{C} & &
\end{array}
$$

注意，收缩积并不满足结合律$(\mathbf{A}\rfloor\mathbf{B})\rfloor\mathbf{C}\neq\mathbf{A}\rfloor(\mathbf{B}\rfloor\mathbf{C})$。因此，必须使用圆括号来避免歧义。为了能简化符号以避免到处使用圆括号，可以使用方程 2.1.10 中的运算优先级约定：

$$
\begin{aligned}
(\mathbf{A}\wedge\mathbf{B})\mathbf{C} &= \mathbf{A}\wedge\mathbf{B}\,\mathbf{C} &&\neq \mathbf{A}\wedge(\mathbf{B}\,\mathbf{C}) \\
(\mathbf{A}\rfloor\mathbf{B})\mathbf{C} &= \mathbf{A}\rfloor\mathbf{B}\,\mathbf{C} &&\neq \mathbf{A}\rfloor(\mathbf{B}\,\mathbf{C}) \qquad (2.1.10)\\
\mathbf{A}\rfloor(\mathbf{B}\wedge\mathbf{C}) &= \mathbf{A}\rfloor\mathbf{B}\wedge\mathbf{C} &&\neq (\mathbf{A}\rfloor\mathbf{B})\wedge\mathbf{C}
\end{aligned}
$$

这就是说，当有多种类型的乘法混在一起，可能会引起混淆的时候，外积将最先计算，然后是收缩积，最后是几何积。

在继续讲解之前，先来看一个计算的例子。这个例子涵盖了到目前为止我们已经介绍过的大部分内容。给定多重向量 $\mathbf{A} = 1 + 2\mathbf{e}_2 - 3\mathbf{e}_1\wedge\mathbf{e}_3$ 和 $\mathbf{B} = 2\mathbf{e}_2 + 7\mathbf{e}_1$，其外积、几何积和收缩积的计算见方程 2.1.11：

$$
\begin{aligned}
\mathbf{A}\wedge\mathbf{B} &= 2\mathbf{e}_2 + 7\mathbf{e}_1\wedge\mathbf{e}_2\wedge\mathbf{e}_3 + 4\cdot 0 + 14\cdot 0 - 6\mathbf{e}_1\wedge\mathbf{e}_3\wedge\mathbf{e}_2 - 21\cdot 0 \\
&= 2\mathbf{e}_2 + 13\mathbf{e}_1\wedge\mathbf{e}_2\wedge\mathbf{e}_3 \\
\mathbf{AB} &= 2\mathbf{e}_2 + 7\mathbf{e}_{123} + 4\mathbf{e}_{22} + 14\mathbf{e}_{2123} - 6\mathbf{e}_{132} - 21\mathbf{e}_{13123} \\
&= 2\mathbf{e}_2 + 7\mathbf{e}_{123} + 4 - 14\mathbf{e}_{1223} + 6\mathbf{e}_{123} - 21\mathbf{e}_{11233} \\
&= 2\mathbf{e}_2 + 7\mathbf{e}_{123} + 4 - 14\mathbf{e}_{13} + 6\mathbf{e}_{123} - 21\mathbf{e}_2 \\
&= 4 - 19\mathbf{e}_2 - 14\mathbf{e}_1\wedge\mathbf{e}_3 + 13\mathbf{e}_1\wedge\mathbf{e}_2\wedge\mathbf{e}_3 \qquad (2.1.11)\\
\mathbf{A}\rfloor\mathbf{B} &= 2\rfloor\mathbf{e}_2 + 7\rfloor\mathbf{e}_{123} + 4\mathbf{e}_2\rfloor\mathbf{e}_2 + 14\mathbf{e}_2\rfloor\mathbf{e}_{123} - 6\cdot 0 - 21\mathbf{e}_{13}\rfloor\mathbf{e}_{123} \\
&= 2\langle\mathbf{e}_2\rangle_{1-0} + 7\langle\mathbf{e}_{123}\rangle_{3-0} + 4\langle\mathbf{e}_2\mathbf{e}_2\rangle_{1-1} + 14\langle\mathbf{e}_2\mathbf{e}_{123}\rangle_{3-1} - 21\langle\mathbf{e}_{13}\mathbf{e}_{123}\rangle_{3-2} \\
&= 2\mathbf{e}_2 + 7\mathbf{e}_{123} + 4 - 14\mathbf{e}_{13} - 21\mathbf{e}_2 \\
&= 4 - 19\mathbf{e}_2 - 14\mathbf{e}_1\wedge\mathbf{e}_3 + 7\mathbf{e}_1\wedge\mathbf{e}_2\wedge\mathbf{e}_3
\end{aligned}
$$

5. 逆收缩积

收缩积使我们可以轻松地计算很多多重向量的逆。我们称一个多重向量为 *versor*，如果它可以写成向量外积的形式[2]。所以，*k*-blade 是一个 versor，versor 也可以写成一个 *k*-blade。有些 *k*-vector 不是显式的 versor，有些 *k*-vector 则不是 versor。例如，对于任意的 *k*，$\mathbf{e}_{13}-\mathbf{e}_{12}+\mathbf{e}_{23}$ 不是一个 *k*-blade（但是一个 *2-vector*），但它是一个 versor，因为它可以写成 $\mathbf{e}_{13}-\mathbf{e}_{12}+\mathbf{e}_{23}=(\mathbf{e}_1+\mathbf{e}_2)\wedge(\mathbf{e}_1+\mathbf{e}_3)$。可以证明在三维或者更低维的情况下，所有的 *k*-vector 实际都是 versor，虽然在更高维的情况下并非如此。对于非零的 versor A_r，可以写成 *r*-vector 的外积形式，它的（左和右）逆为 $\mathbf{A}_r^{-1} = \tilde{\mathbf{A}}/(\mathbf{A}\rfloor\tilde{\mathbf{A}})$，从它能推导出一个特例：如同前面所看到的，向量 **a** 的逆为

[2] 注意 versor 通常可以写成正交基的外积并乘以一个“体积”标量的形式，这些正交基形成相同的子空间，这在计算中很有用。

$\mathbf{a}^{-1}=\tilde{\mathbf{a}}/(\mathbf{a}\rfloor\tilde{\mathbf{a}})=\mathbf{a}/\mathbf{a}\cdot\mathbf{a}$。非常有趣的是，我们可以证明 $1+\mathbf{e}_1$ 在任意维空间的情况下都没有逆（不管是左逆还是右逆）。然后求出 $1+\mathbf{e}_1\wedge\mathbf{e}_2$ 的逆。最后，可以看到 $1+\mathbf{e}_1+\mathbf{e}_1\wedge\mathbf{e}_2$ 和 $1-\mathbf{e}_1-\mathbf{e}_1\wedge\mathbf{e}_2$ 都是可逆的，但它们都不是 versor。必须小心处理逆，因为在有些情况下，左逆和右逆是不同的。对于 versor 来说，求逆是安全的，因为方程生成的逆既可用做左逆也可用做右逆。

2.1.3 线性代数

定义了几何代数并知道如何进行一些积运算之后，我们要重温一些线性代数概念，这些概念已经在文中提到过。

1. 复数

任何一个平面上都存在复数。给定一个平面，选取两个正交的单位向量 **u** 和 **v**，并把它的二重向量标记为 $\mathbf{i}=\mathbf{n}\wedge\mathbf{v}$，然后把该二重向量看作虚部，则该平面中的任意一个矢量 $\mathbf{a}=\alpha\mathbf{u}+\beta\mathbf{v}$ 都可以像引言中介绍的那个例子一样进行旋转。乘以一个多重向量 $\mathbf{R}_n=\cos\theta+\mathbf{i}\sin\theta$，得到方程式2.1.12：

$$\mathbf{R}_\theta\mathbf{a}=(\alpha\cos\theta+\beta\sin\theta)\mathbf{u}+(\beta\cos\theta-\alpha\sin\theta)\mathbf{v}=\mathbf{a}\mathbf{R}_{-\theta}. \tag{2.1.12}$$

注意，旋转的方向依赖于左乘还是右乘，为了让它看起来更像一般的旋转，可以写成 $\mathbf{a}\to\mathbf{R}_{\theta/2}\mathbf{a}\mathbf{R}_{-\theta/2}$，角度 $\theta/2$ 使这一变换看起来像四元数（quaternion）旋转，后面将对此进行讲解。

2. 四元数

在三维空间中定义如下的二重向量：

$$\mathbf{i}=\mathbf{e}_3\wedge\mathbf{e}_2$$

然后应用四元数的关系：$\mathbf{i}^2=\mathbf{j}^2=\mathbf{k}^2=-1$，$\mathbf{ij}=\mathbf{k}$（以及循环组合），其中的乘积为几何积。选出来的二重向量 **i** 用来表示绕 x 轴的旋转，等等，这样就在四元数和几何代数的一般概念之间建立了联系。用几何代数的观点来看，形如 $\alpha+\beta\mathbf{i}+\gamma\mathbf{j}+\delta\mathbf{k}$ 的多重向量就是四元数，但是它把原来只能用向量（如同我们平常使用的四元数）进行的旋转扩展到了可以用多重向量进行旋转。对于非零四元数，它的逆就是它的几何逆。

用四元数对一个平面进行旋转，需要用四元数对每个定义平面的参数进行处理。几何代数的方法则是用四元数直接旋转平面（二重向量），从概念上讲要更加完善。

现在换一个角度，我们不再按常规的定律将四元数看成是四维球中的一个单位向量，而是把它看成图2.1.4（轴一转角视图）中的一个（定义了旋转轴的）平面和旋转角度。这样，复数就是二维空间中的旋转，而四元数则是三维空间中的旋转，几何代数把他们统一了起来。任意维空间中的旋转都符合相同的法则，复数和四元数只是两个特例而已（注意他们都采用二重向量来表示旋转）。

3. 反射和旋转

一个多重向量 **k** 关于一个 *k*-blade **A** 的反射（均通过原点）是通过变换 $\mathbf{X}\to-(-1)^k\mathbf{A}\mathbf{X}\mathbf{A}^{-1}$ 计算的。如图2.1.5所示，它演示了如何用变换 $\mathbf{x}\to\mathbf{a}\mathbf{x}\mathbf{a}^{-1}$ 把一个向量 **x** 通过另外一个向量 **a** 做反射。这是一个非常有用的方程，因为它适用于任意维空间，而且它是与维数无关的旋转的基础。

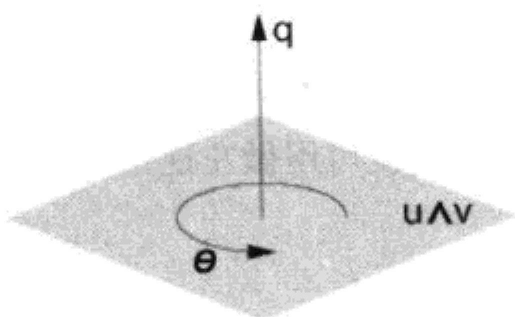


图 2.1.4 四元数可以看成是一个有大小的有向面积

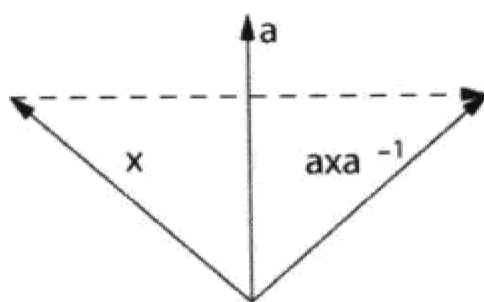


图 2.1.5 反射有非常简单的 GA 记法

将一个向量对两个夹角为$\theta/2$做两次反射，可以把编码到多重向量中的对象旋转角度θ。对象 **R**=**ab** 被称为转子（*rotor*），它能在任意维空间上，用变换 $\mathbf{X}\rightarrow\mathbf{RXR}^{-1}$ 对任意多重向量 **X** 做旋转操作。它比用矩阵表示的旋转更加有效，并且在数值上更加稳定。它还能旋转多重向量，而不仅仅是向量（参见后续范例）。正如我们所预见的，在 3D 空间中，一个 rotor 展开成基的形式后就是一个四元数。

有很多等效的方法可以用来构造 rotor。把 **a** 和 **b** 取成单位向量是非常有用的，因为这样一来，$\mathbf{R}^{-1}=\mathbf{ba}$ 就很容易计算。在3D中，绕轴 **c** 旋转θ角度的rotor可以这样构造：$\mathbf{R}=\cos(\theta/2)-\sin(\theta/2)\mathbf{Ic}$。有了这些不同且简洁的构造旋转的公式，GA 提供了一种更为简单的计算方法。在任意维中，构造一个 rotor 就是计算一个几何积。甚至还可以对 rotor 进行运算，得到速度、加速度、插值、最小路径、摄影机方向等，但是介绍这些需要更多的篇幅。和所有数学问题一样，我们很难在一篇文章里将其阐述完整。rotor 的另外一个优点是：不同于标量－角度或旋转矩阵的方式，它能保持旋转的方向，并允许对旋转进行插值和分割。在三维空间中，rotor 的插值通常使用 SLERP 方式，如方程 2.1.13 所示：

$$\mathbf{R}(\lambda)=\frac{\sin\left((1-\lambda)\theta\right)\mathbf{R}_1+\sin(\lambda\theta)\mathbf{R}_2}{\sin\theta} \tag{2.1.13}$$

对应两个旋转的中点可以简化为：

$$\mathbf{R}(1/2)=\frac{\sin(\theta/2)}{\sin\theta}(\mathbf{R}_1+\mathbf{R}_2)$$

4. 和线性代数的关系

给定一个线性变换 $f:V^n\rightarrow V^n$（例如，一个矩阵乘法），f很自然地以$f(\alpha)=\alpha$作用在 *k*-blade 上。对标量增加这个法则$f(\alpha)=\alpha$，并扩展这个线性关系，得到定义于几何代数中的所有多重向量上的变换 $f:CV\rightarrow CV$。这个扩展能很好地和几何符号配合起来。例如，线性变换会把一条直线变成另外一条直线、把一个平面变换成另外一个平面，等等。为了说明变换把直线变成另外一条直线，可以把f应用在一般直线方程$(\mathbf{x}-\mathbf{a})\wedge\mathbf{u}=0$的两边，得到方程 2.1.14：

$$\begin{aligned}
f\left((\mathbf{x}-\mathbf{a})\wedge\mathbf{u}\right)&=f(0)\\
f(\mathbf{x}-\mathbf{a})\wedge f(\mathbf{u})&=0\\
\left(f(\mathbf{x})-f(\mathbf{a})\right)\wedge f(\mathbf{u})&=0\\
(\mathbf{x}'-\mathbf{a}')\wedge\mathbf{u}'&=0
\end{aligned} \tag{2.1.14}$$

PDG

2.1.4 词典

在演示几何代数是如何简化方程的求解之前，先来看一些普通的几何关系，以及这些关系是如何转化为几何代数的。本文没有足够的篇幅来推导和解释每一个关系，只是简单地列出事实以及可以应用在它们之上的一些简化的计算法则。

1．几何积是最基本的乘积，其他乘积都可以表示成几何积的形式[3]。例如方程 2.1.15：

$$\begin{aligned}\mathbf{a}\wedge\mathbf{b}&=\frac{1}{2}(\mathbf{ab}-\mathbf{ba})\\ \mathbf{a}\cdot\mathbf{b}&=\frac{1}{2}(\mathbf{ab}+\mathbf{ba})\\ \mathbf{a}\times\mathbf{b}&=(\mathbf{a}\wedge\mathbf{b})\rfloor\tilde{\mathbf{I}}_3=-\mathbf{I}_3(\mathbf{a}\wedge\mathbf{b})=-(\mathbf{a}\wedge\mathbf{b})\mathbf{I}_3\\ \mathbf{a}\rfloor\mathbf{b}&=\langle\mathbf{ab}\rangle_0\end{aligned}\tag{2.1.15}$$

2．3D 空间中的叉积为 $\mathbf{a}\times\mathbf{b}=(\mathbf{a}\wedge\mathbf{b})\rfloor\tilde{\mathbf{I}}_3=-\mathbf{I}_3(\mathbf{a}\wedge\mathbf{b})=-(\mathbf{a}\wedge\mathbf{b})\mathbf{I}_3$。通过给 $\mathbf{a}\wedge\mathbf{b}$ 张成的空间选一组正交基，并将其展开成 3D 的基的形式，然后计算出每一个表达式，就可以证明这一点。

3．两个向量当且仅当点积为 0 时是垂直的。这可以扩展到应用于 *k*-blade 的收缩积上。

4．线性相关：当且仅当向量是线性相关的时候，其外积为 0。所以当且仅当两个向量的外积为 0 的时候，它们才是平行的。这可以简化平面和直线的方程。

5．过点 **a**，方向为 **u** 的直线方程为 $(\mathbf{x}-\mathbf{a})\wedge\mathbf{u}=0$。这适用于任意维空间，所以它比维数相关的形式更加简练。

6．类似的，过点 **a**，和二重向量 $\mathbf{u}\wedge\mathbf{v}$ 平行的平面方程为 $(\mathbf{x}-\mathbf{a})\wedge\mathbf{u}\wedge\mathbf{v}=0$。再强调一遍，它也与维数无关。

7．blade **A** 到 blade **B** 上的正交投影由 $P_{\mathbf{B}}(\mathbf{A})=(\mathbf{A}\rfloor\mathbf{B})\mathbf{B}^{-1}$ 给出。这个公式非常完美，因为它适用于任意维的任意两个空间：点到线、线到平面、平面到平面，等等。

8．对投影进行扩展后，表达式 $\mathbf{A}-P_{\mathbf{B}}(\mathbf{A})$ 一定是表示 **A** 对 **B** 的垂直分量。

9．伪标量（pesudoscalar）Z 对于奇数维空间中的所有分量都满足交换律。

10．给定一个向量 **n**，在三维空间中垂直于 **n** 的平面为 $-\mathbf{nI}$。给定一个由 $\mathbf{u}\wedge\mathbf{v}$ 定义的平面，法向量为 $-(\mathbf{u}\wedge\mathbf{v})\mathbf{I}$。可以由图 2.1.6 来表示。注意，平面是有方向的，所以我们包含了负号来表示平面是“双向”的。

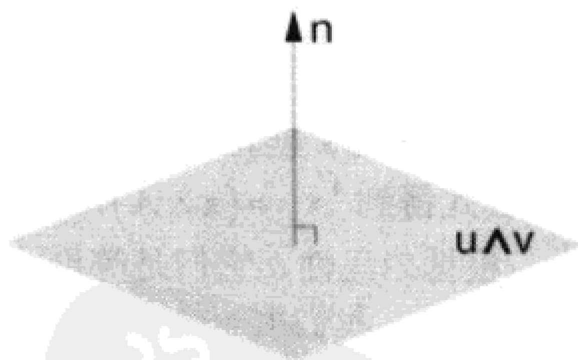


图 2.1.6 在 3D 空间中，垂直于平面的向量是“双向”的

11．二元性：多重向量 **A** 的双重由 $\mathbf{A}^*=\mathbf{A}\rfloor\tilde{\mathbf{I}}$ 定义。它对交换“生成”关系和“垂直”关系非常有用。它是对前面讲述过的事实的概括。在 3D 空间中，这是根据法向量来交换二重向量，所以可以用它快速得到平面的法向量。平面和法向量的 3D 演示如图 2.1.6 所示。

[3] 这是按照公理的方式来定义几何积，可以从它推导出其他的概念，所以没有循环定义，这样的讲解方式更易于理解。

12．多重向量的 *norm* 是向量长度的推广，写成$|\mathbf{A}| = \sqrt{\mathbf{A} \rfloor \tilde{\mathbf{A}}}$。它会返回二重向量的面积、向量的长度，等等。

13．给定两个 versor **A** 和 **B**，它们各自定义了 V 的一个子空间，其相交的部分称为交（*meet*）并记为$\mathbf{M} = \mathbf{A} \cap \mathbf{B}$。它可以通过$\mathbf{A} \cap \mathbf{B} = \mathbf{A}^* \rfloor \mathbf{B} = \left(\mathbf{A} \rfloor \tilde{\mathbf{I}}\right) \rfloor \mathbf{B}$计算。和它对应的概念是并（*join*），它是 V 中包含这两个子空间的最小子空间，写成$\mathbf{J} = \mathbf{A} \cup \mathbf{B}$。例如，3D 空间中两个不平行平面的交是这两个平面的交线，如图 2.1.7 所示。两个 versor 的交可以看成是定义它们的向量的最大公约数，而并则可以看成是最小公倍数。通过方程 2.1.16 中所示的关系，知道其中一个，就可以很容易地求出另外一个。

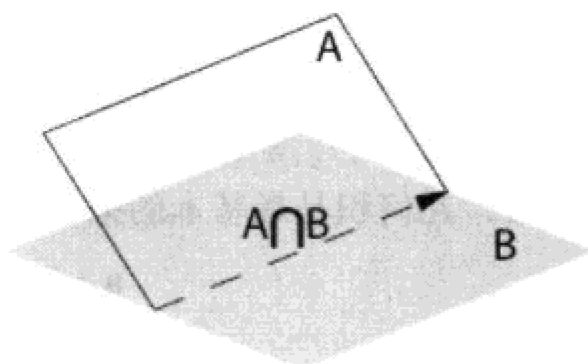


图 2.1.7 两个子空间的交

$$\mathbf{J} = \mathbf{A} \wedge \left(\mathbf{M}^{-1} \rfloor \mathbf{B}\right)$$
$$\mathbf{M} = \left(\mathbf{B} \rfloor \mathbf{J}^{-1}\right) \rfloor \mathbf{A} \qquad (2.1.16)$$

2.1.5 实例

本节将提供几个计算实例来演示如何使用几何代数。本文的出发点就是演示几何代数对手工演算和代码复杂度的简化。所以在学习这些例子的时候，读者应该考虑两种几何引擎的代码复杂度：一个是基本线性代数例程，一个是支持几何积的例程。在每种情况下，手工推导均非常简单，并且在几何代数引擎中只涉及很少的几行代码。但是，这些例子的大部分在使用线性代数的时候，却会涉及大量的代码。

1．给定两个向量 **x** 和 **a**，假定需要求出 **x** 垂直于 **a** 的分量 $\mathbf{x}_\perp$，如图 2.1.8 所示。垂直的条件为$\mathbf{x}_\perp \rfloor \mathbf{a} = 0$，它的大小还需要满足另外一个条件：由 **a** 和 **x** 张成的面积与 **a** 和 $\mathbf{x}_\perp$张成的面积是相同的（也就是说，平行四边形的面积是底乘高）。所以，$\mathbf{x}_\perp \wedge \mathbf{a} = \mathbf{x} \wedge \mathbf{a}$。把这两个方程加起来，得到几何积$\mathbf{x}_\perp \wedge \mathbf{a} + \mathbf{x}_\perp \rfloor \mathbf{a} = \mathbf{x} \wedge \mathbf{a} + 0 = \mathbf{x}_\perp \mathbf{a}$，把两边除以 **a** 得到$\mathbf{x}_\perp = (\mathbf{x} \wedge \mathbf{a})\mathbf{a}^{-1}$。为了看清它是如何对应标准的线性代数的，可以选择一个诸如$\mathbf{a} = \alpha \mathbf{e}_1$和$\mathbf{x} = \beta \mathbf{e}_1 + \delta \mathbf{e}_2$的基代入并展开，得到$\mathbf{x}_\perp = \left((\beta \mathbf{e}_1 + \delta \mathbf{e}_2) \wedge \alpha \mathbf{e}_1\right)(\mathbf{e}_1 / \alpha) = \delta \mathbf{e}_2$，正如我们所期望的。现在让我们举一反三一下，如果想得到 **x** 垂直于一个由二重向量 **A** 给出的平面的分量，该怎么办呢？可以用相同的数学公式得到[4] $\mathbf{x}_\perp = (\mathbf{x} \wedge \mathbf{A})\mathbf{A}^{-1}$。非常完美！（或者，我们只是把 **x** 投影到平面的法向量$-\mathbf{A}I_3$ 上，根据二元性，它们是等价的。）

2．接下来，我们求一个点对平面的反射。在图 2.1.9 中，B 是平面的一个侧面，**a** 是需要对该平面求反射的点。通过向量的思路，我们得到一个非常巧妙的方程来计算反射。从前面的内容可以知道，把 **a** 对 **n** 做反射就是 $\mathbf{nam}^{-1}$，且从图 2.1.8 可知点对平面的反射是$-\mathbf{nam}^{-1}$。为了避免因为相信完美的图片而得到错误的结果，要注意这个方程对平面的“其他”法线仍

[4] 事实上，我们需要知道对于任意向量 **x** 和二重向量 **A**，当且仅当 $\mathbf{X} \perp \mathbf{A}$ 的时候，才有 $\mathbf{xA} = \mathbf{x} \rfloor \mathbf{A} + \mathbf{x} \wedge \mathbf{A}$ 和 $\mathbf{x} \rfloor \mathbf{A} = 0$。这两个结论都很容易证明，向量的情况下对这非常有启示。

然适用。计算一个具体的例子是有指导意义的。所以给定点和平面，只需一行代码就可以进行计算，而且这个方程在任何维空间都是适用的。

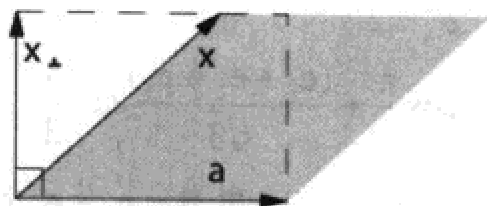


图 2.1.8　几何代数能用简单的符号来分离对象的平行和垂直分量

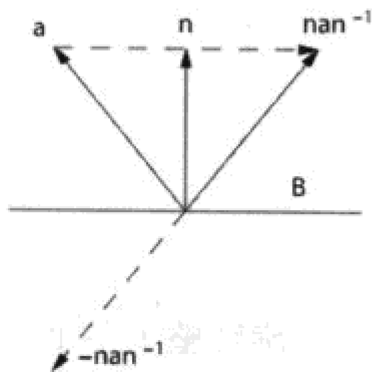


图 2.1.9　对向量求反射以后再取负，得到的结果就是对平面求反射

3．通过计算两个通过原点的非平行平面的交线来演示如何计算两个 blade 的交，如图 2.1.7 所示。我们选择那些可以很容易观察的平面，并得到了期待的答案，如方程 2.1.17 所示。

$$\begin{aligned}(\mathbf{e}_1\wedge\mathbf{e}_3)\cap(\mathbf{e}_2\wedge\mathbf{e}_3)&=(\mathbf{e}_1\wedge\mathbf{e}_3)^*\rfloor(\mathbf{e}_2\wedge\mathbf{e}_3)\\&=\left((\mathbf{e}_1\wedge\mathbf{e}_3)\rfloor\tilde{\mathbf{I}}\right)\rfloor(\mathbf{e}_2\wedge\mathbf{e}_3)\\&=\langle\mathbf{e}_1\mathbf{e}_3\mathbf{e}_3\mathbf{e}_2\mathbf{e}_1\rangle_{3-2}\rfloor(\mathbf{e}_2\wedge\mathbf{e}_3)\\&=\langle-\mathbf{e}_2\mathbf{e}_2\mathbf{e}_3\rangle_{2-1}\\&=-\mathbf{e}_3\end{aligned}\tag{2.1.17}$$

4．通过一个具体的公式得到最终的旋转，来证明欧拉（Euler）定理：两个旋转的乘积是另外一个乘积。我们从一个标量β_0加上一个旋转轴$\mathbf{q}=\beta_1\mathbf{e}_1+\beta_2\mathbf{e}_2+\beta_3\mathbf{e}_3$的观点来考察一个四元数，写成$\beta_0+\mathbf{q}$。假设任何一个旋转都可以写成四元数的形式[5]。回忆一下伪标量$\mathbf{I}=\mathbf{e}_1\wedge\mathbf{e}_2\wedge\mathbf{e}_3$并使用前面的定义，我们可以用几何代数的变换$\beta_0+\mathbf{q}\to\beta_0-\mathbf{I}_3\mathbf{q}$重写四元数的表示形式。现在把两个四元数相乘，从方程 2.1.18 中就可以很容易地读出定义最终四元数的标量和向量。这就证明了欧拉定理，并得到标量和向量的具体值。用通常的四元数方法来处理这一关系会非常麻烦的，但使用几何代数的方法就比较直接。

$$\begin{aligned}(p_0+\mathbf{p})(q_0+\mathbf{q})&\to(p_0-\mathbf{I}_3\mathbf{p})(q_0-\mathbf{I}_3\mathbf{q})\\&=p_0q_0-p_0\mathbf{I}_3\mathbf{q}-q_0\mathbf{I}_3\mathbf{p}+\mathbf{I}_3\mathbf{p}\mathbf{I}_3\mathbf{q}\\&=p_0q_0-\mathbf{p}\mathbf{q}-p_0\mathbf{I}_3\mathbf{q}-q_0\mathbf{I}_3\mathbf{p}\\&=p_0q_0-\mathbf{p}\cdot\mathbf{q}-\mathbf{p}\wedge\mathbf{q}-p_0\mathbf{I}_3\mathbf{q}-q_0\mathbf{I}_3\mathbf{p}\\&=p_0q_0-\mathbf{p}\cdot\mathbf{q}-\mathbf{I}_3(p_0\mathbf{q}+q_0\mathbf{p}+\mathbf{q}\times\mathbf{p})\\&\to(p_0q_0-\mathbf{p}\cdot\mathbf{q})+(p_0\mathbf{q}+q_0\mathbf{p}+\mathbf{q}\times\mathbf{p})\end{aligned}\tag{2.1.18}$$

5．一个具体的旋转例子是有序的：沿 $\mathbf{e}_1$ 旋转$\pi/2$ 后再绕 $\mathbf{e}_2$ 旋转$\pi/2$，得到的结果为绕向量$(\mathbf{e}_1+\mathbf{e}_2+\mathbf{e}_3)/\sqrt{3}$旋转 $2\pi/3$。该计算使用的是前面一个例子中的四元数表示方法。记住要把旋转角度除以 2，证明过程如方程 2.1.19。

[5] 假设每个旋转都可以用一个四元数来表示。笔者已有一个证明，但无法写在本页空白处。

PDG

$$\left(\cos\frac{\pi}{4}+\mathbf{e}_1\sin\frac{\pi}{4}\right)\left(\cos\frac{\pi}{4}+\mathbf{e}_2\sin\frac{\pi}{4}\right)\rightarrow\frac{1}{\sqrt{2}}(1+\mathbf{e}_{32})\frac{1}{\sqrt{2}}(1+\mathbf{e}_{13})$$
$$=\frac{1}{2}(1+\mathbf{e}_{13}+\mathbf{e}_{32}+\mathbf{e}_{21}) \qquad (2.1.19)$$
$$=\frac{1}{2}-\frac{\sqrt{3}}{2}\mathbf{I}\frac{(\mathbf{e}_1+\mathbf{e}_2+\mathbf{e}_3)}{\sqrt{3}}$$
$$\rightarrow\cos\frac{\pi}{3}+\frac{(\mathbf{e}_1+\mathbf{e}_2+\mathbf{e}_3)}{\sqrt{3}}\sin\frac{\pi}{3}$$

6．举一个一次旋转整个多重向量的例子：把一个用二重向量 $\mathbf{B}=\mathbf{e}_1\wedge\mathbf{e}_2$ 定义的平面绕向量 $\mathbf{a}=\mathbf{e}_1+\mathbf{e}_2$ 旋转 60 度。不需要先把平面分解成向量。[6] 先构造一个单位化的 rotor $\mathbf{R}=\cos(\pi/6)-(\mathbf{Ia}/\sqrt{2})\sin(\pi/6)$，然后写出旋转平面的方程，如方程 2.1.20 所示：

$$\mathbf{RBR}^{-1}=\left(\cos\frac{\pi}{6}-\frac{\mathbf{e}_{23}-\mathbf{e}_{13}}{\sqrt{2}}\sin\frac{\pi}{6}\right)(\mathbf{e}_1\wedge\mathbf{e}_2)\left(\cos\frac{\pi}{6}+\frac{\mathbf{e}_{23}-\mathbf{e}_{13}}{\sqrt{2}}\sin\frac{\pi}{6}\right) \qquad (2.1.20)$$
$$=\frac{1}{2}\mathbf{e}_{12}+\frac{\sqrt{6}}{4}(\mathbf{e}_{13}+\mathbf{e}_{23})$$

7．给定 **a**，**b** 和 **c** 三个共面的向量，能计算出矢量 **x**，**x** 满足“**a** 比 **b** 等于 **c** 比 **x**”，如图 2.1.10 所示。如果是实数，能通过 **x:c=b:a** 解出 **x**。几何代数提供相同的形式，因为通过除以向量并乘上 **c** 后可以得到 $\mathbf{x}=\mathbf{ba}^{-1}\mathbf{c}$。

8．同样，在平面内可以求出方程 $(\mathbf{x}-\mathbf{a})\wedge\mathbf{u}=0$ 表示的直线到原点的距离，如图 2.1.11 所示。向量 **d** 垂直于直线并经过原点，其长度就是所求的距离。应用实例 2.1.1 中使用的法线条件和面积条件，得到 $\mathbf{d}=(\mathbf{a}\wedge\mathbf{u})\mathbf{u}^{-1}$，它给出了距离 $|\mathbf{d}|$。那么它对于平面到原点的距离也适用吗？

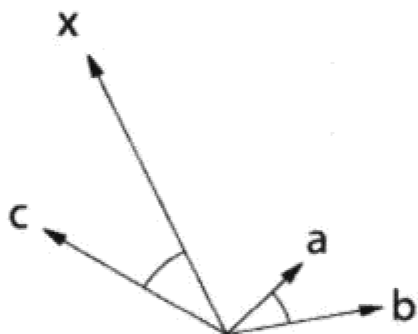


图 2.1.10 比例通过向量的大小和方向来展现

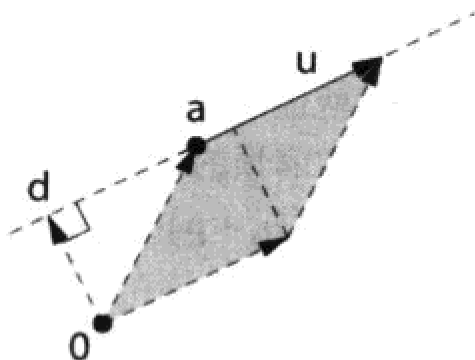


图 2.1.11 计算一个点到一条直线的距离是非常简单的

9．为了展示一个投影方程的例子，我们把向量 $\mathbf{a}=\alpha_1\mathbf{e}_1+\alpha_2\mathbf{e}_2+\alpha_3\mathbf{e}_3$ 投影到平面 $\mathbf{B}=\mathbf{e}_1\wedge\mathbf{e}_2$ 上，选择的平面能更容易地验证结果。因为平面的朝向很完美，我们可以很容易地知道答案应该是 $\alpha_1\mathbf{e}_1+\alpha_2\mathbf{e}_2$。方程 2.1.21 的计算验证了这一点。

[6] 这样可避免将旋转作用在平面上和不得不重新计算法向量的问题。注意，旋转后平面的法向量并不等于旋转平面的法向量。通常平面用一个点和一个法向量的方式来定义，旋转一个平面需要两个垂直于法向量的线性元素向量，旋转这两个向量，计算出一个新的法向量和一个新的点。

PDG

$$
\begin{aligned}
P_{\mathbf{B}}(\mathbf{a}) &= (\mathbf{a} \rfloor \mathbf{B})\mathbf{B}^{-1} \\
&= \langle(\alpha_1\mathbf{e}_1+\alpha_2\mathbf{e}_2+\alpha_3\mathbf{e}_3)(\mathbf{e}_{12})\rangle_{2-1}(\mathbf{e}_{21}) \\
&= (\alpha_1\mathbf{e}_2-\alpha_2\mathbf{e}_1)(\mathbf{e}_{21}) \\
&= \alpha_1\mathbf{e}_1+\alpha_2\mathbf{e}_2
\end{aligned} \tag{2.1.21}
$$

10．有一个通用的方程可以求解平面中两条直线的交（即克莱姆法则 Cramer's Rule，只不过是用直觉和几何的方法求解）。如图 2.1.12 所示，第一眼见到这个图的时候可能会觉得有点难。

可以把两条给定的直线 $(\mathbf{x}-\mathbf{a})\wedge\mathbf{u}=0$ 和 $(\mathbf{x}-\mathbf{b})\wedge\mathbf{v}=0$ 写成展开的形式 $\mathbf{x}\wedge\mathbf{u}=\mathbf{a}\wedge\mathbf{u}$ 和 $\mathbf{x}\wedge\mathbf{v}=\mathbf{b}\wedge\mathbf{v}$，设 $\mathbf{U}=\mathbf{a}\wedge\mathbf{u}$ 和 $\mathbf{V}=\mathbf{b}\wedge\mathbf{v}$。很明显，它们的交应该是 $\mathbf{u}$ 和 $\mathbf{v}$ 的线性组合。为了找到线性组合的系数，我们来观察图片。$\mathbf{u}$ 的系数应该是 $\mathbf{V}$ 表示的面积除以 $\mathbf{u}\wedge\mathbf{v}$ 表示的面积，而 $\mathbf{v}$ 的系数应该是 $\mathbf{U}$ 除以 $\mathbf{u}\wedge\mathbf{v}$。但是，因为必须考虑二重向量的方向，所以需要把 $\mathbf{u}\wedge\mathbf{v}$ 在 $\mathbf{v}$ 方向上取反。因此最后的答案应该是：

$$\mathbf{x}=\frac{\mathbf{V}}{\mathbf{u}\wedge\mathbf{v}}\mathbf{u}+\frac{\mathbf{U}}{\mathbf{v}\wedge\mathbf{u}}\mathbf{v}$$

如果对几何代数非常熟悉，会很容易画出图片并很快地写出上式。有了几何代数引擎，就不需要复杂的线求交例程——只需要几何积就足够了。

11．家庭作业：由 $(x_i,y_i), i=1,2,3$ 给定一个平面上的三角形，它的面积是由方程 2.1.22 给出的行列式的绝对值。如果只用解析几何的知识，证明起来会非常费事，但使用几何代数的方法却非常简短（大约只需要半页纸）。提示：使用外积会非常容易。使用几何代数可以方便地扩展到高维：

$$Area=\frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix} \tag{2.1.22}$$

12．最后一个例子，求解从一个方向变换到另外一个方向的旋转，如图 2.1.13 所示：

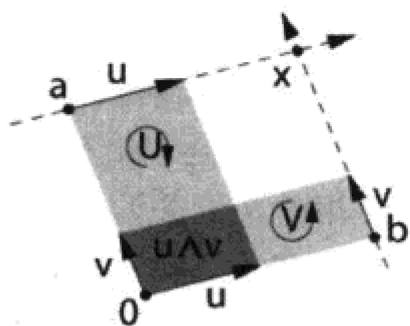


图 2.1.12　这是克莱姆法则的图形化表现

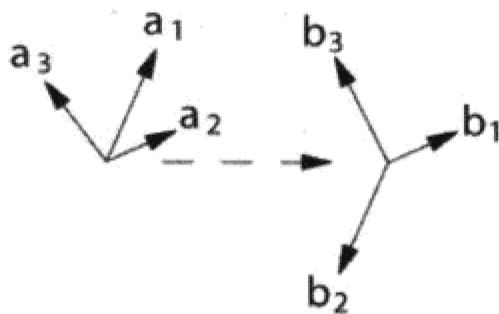


图 2.1.13　计算从一个朝向到另外一个朝向的变换是很简单的

例如，让一个由 AI 控制的太空船和一个由玩家控制的太空船对齐。如果把朝向作为四元数来保存的话，就仅仅需要做求逆和乘法操作。但如果只是用三个线性无关的向量来定义朝向，求解这个旋转就会有很大的工作量。这三个向量不需要是正交的，只要是线性无关的就行。假设有一个旋转（rotor）R，$\mathbf{b}_i=\mathbf{R}\mathbf{a}_i\mathbf{R}^{-1}$，其中 $i=1,2,3$。需要求解这三个方程来得到 R。设 $\mathbf{R}=\alpha-\mathbf{B}$，B 为一个二重向量（所以 $\mathbf{R}^{-1}=\alpha+\mathbf{B}$）。那些喜欢冒险的读者可能会在继续阅读之前尝试用线性代数的知识来求解。建立一个逆坐标系（它在某种程度上起到了正交坐标系的

作用），将其写成上标的形式：

$$\mathbf{a}^1 = \frac{\mathbf{a}_2 \wedge \mathbf{a}_3 \mathbf{I}}{\mathbf{a}_1 \wedge \mathbf{a}_2 \wedge \mathbf{a}_3 \mathbf{I}}$$

它是一个向量（这很容易验证），用相似的方法定义出 $\mathbf{a}^2$ 和 $\mathbf{a}^3$。注意 $\mathbf{a}^i \cdot \mathbf{a}_j = \delta_j^i$[7]。然后可以（仔细地）计算方程 2.1.23 中的表达式，使用它可以分离出 $\mathbf{R}$。接下来，解出 $\mathbf{R}$ 关于 $\mathbf{a}^i$ 和 $\mathbf{b}_i$ 的表达式。为了消去未知数 α，需要将其单位化：设 $\mathbf{T} = 1 + \sum_i \mathbf{b}_i \mathbf{a}^i$，然后通过 $\mathbf{R} = \mathbf{T}/|\mathbf{T}|$ 得到最终的 rotor。推导过程有很多地方需要检验，但是最终的结果很容易计算，而且在几何代数引擎中只需要很少的代码。如果仅仅用线性代数的知识写这个程序，将需要很大的工作量。关于这个例子，最后需要注意的是对最终的旋转的符号进行检查，如方程 2.1.23 所示，要当心 180° 的旋转。

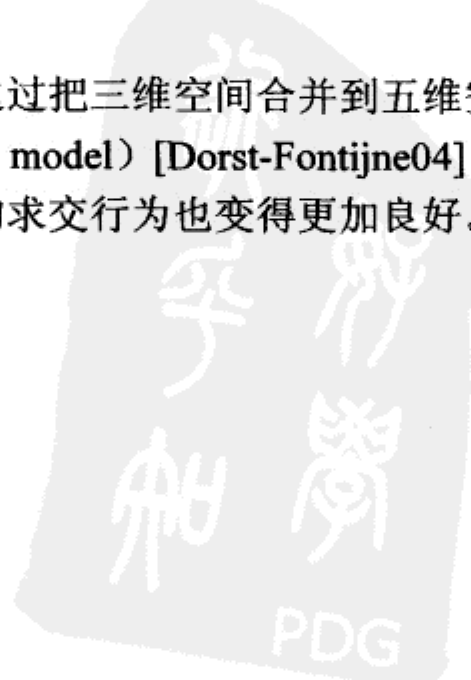
$$\begin{aligned}
\sum_i \mathbf{b}_i \mathbf{a}^i &= \sum_i \mathbf{R}\mathbf{a}_i \mathbf{R}^{-1} \mathbf{a}^i \\
&= \sum_i \mathbf{R}\mathbf{a}_i (\alpha + \mathbf{B}) \mathbf{a}^i \\
&= \mathbf{R}(3\alpha + \mathbf{B}) \\
&= -\mathbf{R}(\alpha - \mathbf{B} - 4\alpha) \\
&= -\mathbf{R}(\mathbf{R}^{-1} - 4\alpha) \\
&= -1 + 4\alpha \mathbf{R}
\end{aligned} \tag{2.1.23}$$

2.1.6 总结以及将来的方向

上面已经讲述了几何代数的定义以及手工计算的法则，它们很容易就能转到一个用计算机进行计算的库中。还讲述了一些基本的几何概念，如投影和旋转，并给出了很多例子来演示如何用几何代数进行几何计算。显然，几何代数把许多概念统一到一个单一的框架下，并提供了比线性代数更加简练和有效的框架。那么，下一步该做什么呢？

其实，几何代数已经在数学、计算机科学和物理学上有很多的应用。[Hestence86]中有关于经典几何代数的完整描述。几何代数看起来非常适合物理学。可以对几何代数对象进行微积分运算，可以进行最大值和最小值的求解，可以用微分方程的形式对运动、交互和物理属性进行描述。几何代数结合了四元数、复数、投影、求交、线性相关（无关），等等。它还包含了普吕克空间并把下列所有几何结合到一个单一的框架里：欧几里得、仿射、投影、球面、逆、双曲和正形投影。这样的统一使得从一个系统转移到另外一个系统变得更加容易，并且提供了对每一个领域的函数更加便捷的访问。

奇特的是，几何代数很适合表示三维几何，并通过把三维空间合并到五维空间以进行交互，人们称之为“双奇次模型”（double homogeneous model）[Dorst-Fontijne04]。它的强大之处在于给球面更好的定义，并且不同的几何体之间的求交行为也变得更加良好。但是这些内容已经超出了本文的范畴。



[7] 译者注：δ_i^j 为科洛内克记号，i=j 的时候为 1，其他时候均为 0。

读者也许还想问一个重要的问题：几何代数能在未来的游戏引擎中替代线性代数吗？事实上，几何代数在目前的高速动作游戏中并没有足够的性能优势。线性代数和计算机图形学用了 30 多年才达到目前的性能水平，而几何代数才刚刚开始在图形学中应用。现在的硬件使用的是线性代数，所以图形引擎需要线性代数。但是，用几何代数创建算法所使用的代码更简练，时间也更少，所以它适合用于工具、测试、原型和其他很多领域。同样，细分曲面过去在实时游戏领域中的效率非常低，但现在却变得非常普遍。在未来的几年内，几何代数可能会成为专业游戏开发领域不可缺少的工具。[8] 在[Gaigen04]中可以找到一些实验性的成果，它们在 C/C++中使用线性代数和几何代数实现了一个光线追踪器，并做了很多比较。

论文[Dorst-Fontijnc04]、[Dorst-Mann02a]、[Dorst-Mann02b]和[Suter03]为学习更多信息提供了很好的起点。[Hestenes98]讨论了几何代数的微积分并包含了参考文献。以作者 Doran，Dorst 和 Hestenes 的名字在网络上进行搜索也能得到很多参考资料。最后需要注意的非常重要的一点，是要意识到不同的作者可能会使用不同的符号。

2.1.7　参考文献

[Dorst-Fontijne04] Dorst, Leo and Daniel Fontijne. "An Algebraic Foundation for Object-Oriented Euclidean Geometry." In preparation; available online at *http://www.science.uva.nl/ga/publications/itm.pdf*.

[Dorst-Mann02a] Dorst, Leo and Stephen Mann. "Geometric Algebra: a computational framework for geometrical applications (part 1: algebra)." *IEEE Computer Graphics and Applications*, May/June 2002. Available online at *http://www.science.uva.nl/~leo/clifford/dorst-mann-I.pdf*.

[Dorst-Mann02b] Dorst, Leo and Stephen Mann. "Geometric Algebra: a computational framework for geometrical applications (part 2: applications)." *IEEE Computer Graphics and Applications*, July/August 2002. Available online at *http://www.science.uva.nl/~leo/clifford/dorst-mann-II.pdf*.

[Gaigen04] A C++ Library to generate geometric algebras. Available online at

http://www.science.uva.nl/ga/gaigen/index.html.2004.

[Hestenes86] Hestenes, David. *New Foundations for Classical Mechanics*. Dordrecht: Kluwer Academic Publishing, 1986.

[Hestenes98] Hestenes, David. *New Foundations for Mathematical Physics*. Two chapters are available online at *http://modelingnts.la.asu.edu/html/NFMP.html*.

[Suter03] Suter, Jaap. "Geometric Algebra Primer." Available online at *http://www.jaapsuter.com/*.

[8] 研究表明，几何代数的消耗只比线性代数稍微多一点点（小于 2 倍），因为游戏通常只花费非常少的时间来进行数学计算，所节省的开发时间和代码让几何代数在很多场合中成为非常吸引人的选择。

2.2 最小加速度 Hermite 曲线

Tony Barrera, Barrera Kristiansen AB
tony.barrera@spray.se

Anders Hast, Gävle 大学创意媒体实验室
aht@hig.se

Ewert Bengtsson, 乌普萨拉大学图像分析中心
ewert@cb.uu.se

这篇文章描述了如何使用 Hermite 样条来获得加速度最小的曲线[Hearn04]。加速度在弯曲的地方比较高，所以这种曲线是弯曲最小的曲线。在对具有这种性质的曲面进行细分的时候，这种类型的曲线非常有用，它们要求曲面尽可能的光滑。类似的针对 Bezier(贝塞尔)曲线以及细分的方法可以在[Overveld97]中找到。它在摄影机的移动中也非常有用[VlachosOl]，因为它允许在曲线上设置摄影机的位置和方向。除此以外，本文还演示了如何把多条曲线段链接起来达到 C^1 连续。

三次 Hermite 曲线由 4 个约束条件来定义：两个端点 $\mathbf{P}_1$ 和 $\mathbf{P}_2$，以及在端点处的切线 $\mathbf{t}_1$ 和 $\mathbf{t}_2$。这篇文章的理念就是让曲线有最小加速度，这可以通过修改切线的长度来实现。这样，端点以及切线的方向都是相同的，但是切线的长度被设置为最优的值，以使曲线有最小的弯曲。为此可以引入变量。图 2.2.1 展示了 3 条不同的 Hermite 曲线，它们各自有不同的切线长度。切线最长的那条虚线表示的曲线在中部能看到明显的弯曲，切线最短的那条由虚线表示的曲线在中部非常平坦，但是在端点附近却有非常明显的弯曲。但是，实线表示的曲线具有最小弯曲的属性。注意图中所有切线的长度都已经缩短到原来的 25%，这样才不至于使曲线相对切线来说显得太小。

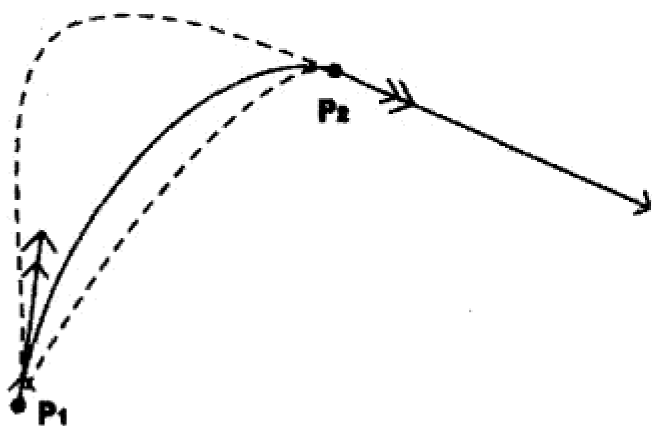
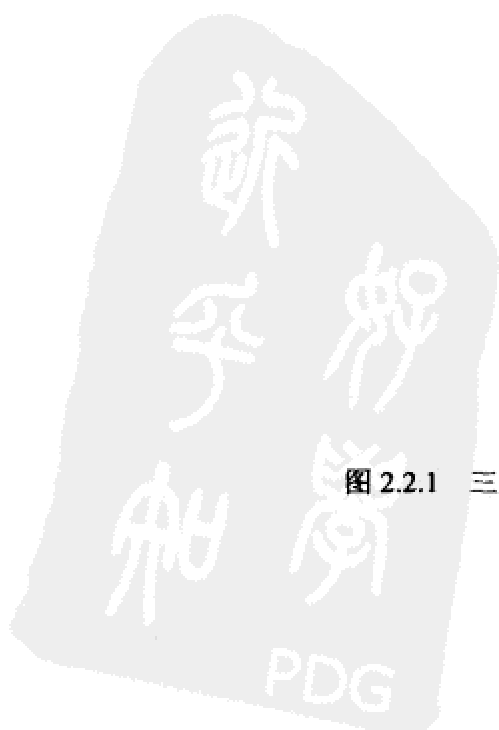


图 2.2.1 三个不同的弯曲对应不同的切线长度。那条实心的曲线拥有最小的弯曲属性


PDG

一般，三次 Hermite 样条的定义如下

$$\mathbf{h}(u)=\mathbf{a}u^3+\mathbf{b}u^2+\mathbf{c}u+\mathbf{d} \tag{2.2.1}$$

Hermite 样条的系数[Hearn04]由如下公式给出

$$\begin{bmatrix}\mathbf{a}\\ \mathbf{b}\\ \mathbf{c}\\ \mathbf{d}\end{bmatrix}=\begin{bmatrix}2 & -2 & 1 & 1\\ -3 & 3 & -2 & -1\\ 0 & 0 & 1 & 0\\ 1 & 0 & 0 & 0\end{bmatrix}\begin{bmatrix}\mathbf{p}_1\\ \mathbf{p}_2\\ \alpha_1\mathbf{t}_1\\ \alpha_2\mathbf{t}_2\end{bmatrix} \tag{2.2.2}$$

注意变量α_1和α_2在后面用于设置优化切线的长度。把 $\mathbf{p}_1$ 和 $\mathbf{p}_2$之间的向量记为 $\mathbf{p}_{12}$，所以方程 2.2.1 中的系数可以由方程 2.2.3 给出。

$$\begin{aligned}\mathbf{a}&=\alpha_1\mathbf{t}_1+\alpha_2\mathbf{t}_2-2\mathbf{p}_{12}\\ \mathbf{b}&=3\mathbf{p}_{12}-2\alpha_1\mathbf{t}_1-\alpha_2\mathbf{t}_2\\ \mathbf{c}&=\alpha_1\mathbf{t}_1\\ \mathbf{d}&=\mathbf{p}_1\end{aligned} \tag{2.2.3}$$

通常使两个函数之间的差最小的方法是最小平方近似[Burdcn89]。使用相同的思想，但这里是要使加速度最小。对每一个α_i，最小加速度可通过解以下方程获得：

$$\frac{\partial}{\partial\alpha_i}\int_0^1\left\|\mathbf{h}''(u)\right\|^2du=0 \tag{2.2.4}$$

这个方程可以这样解释。曲线 h 的加速度（即 h 的二阶导数）求平方的目的是为了避免负数，然后通过积分把所有的值累加起来。累积的和依赖于不同的α_i。为了得到最佳的系数，要对结果进行微分，然后让它等于0。这样，就找到函数的最小值，以得到整个区域中的最小加速度。必须进行的计算在方程2.2.5、2.2.6和2.2.7中列出。

$$\mathbf{h}''(u)=6\mathbf{a}u+2\mathbf{b} \tag{2.2.5}$$

$$\left\|\mathbf{h}''(u)\right\|^2=36a^2u^2+24\mathbf{a}\cdot\mathbf{b}^2u+4b^2 \tag{2.2.6}$$

$$\int_0^1\left\|\mathbf{h}''(u)\right\|^2du=12a^2+12\mathbf{a}\cdot\mathbf{b}+4b^2 \tag{2.2.7}$$

使用符号$a^2=\mathbf{a}\cdot\mathbf{a}$是为了使方程更加易读。将方程 2.2.3 求得的值代入方程 2.2.7，然后对不同的α_i求微分，得到方程 2.2.8。

$$\begin{aligned}\frac{\partial}{\partial\alpha_1}\int_0^1\left\|\mathbf{h}''(u)\right\|^2du&=8\alpha_1t_1^2+4\alpha_2\mathbf{t}_1\cdot\mathbf{t}_2-12\mathbf{p}_{12}\cdot\mathbf{t}_1\\ \frac{\partial}{\partial\alpha_2}\int_0^1\left\|\mathbf{h}''(u)\right\|^2du&=8\alpha_2t_2^2+4\alpha_1\mathbf{t}_1\cdot\mathbf{t}_2-12\mathbf{p}_{12}\cdot\mathbf{t}_2\end{aligned} \tag{2.2.8}$$

让两个等式都等于0，就得到了如下的方程组：

$$\begin{aligned}8\alpha_1t_1^2+4\alpha_2\mathbf{t}_1\cdot\mathbf{t}_2&=12\mathbf{p}_{12}\cdot\mathbf{t}_1\\ 4\alpha_1\mathbf{t}_1\cdot\mathbf{t}_2+8\alpha_2t_2^2&=12\mathbf{p}_{12}\cdot\mathbf{t}_2\end{aligned} \tag{2.2.9}$$

或者除以 4 后用矩阵形式来表示：

$$\begin{bmatrix}2t_1^2 & \mathbf{t}_1\cdot\mathbf{t}_2\\ \mathbf{t}_1\cdot\mathbf{t}_2 & 2t_2^2\end{bmatrix}\begin{bmatrix}\alpha_1\\ \alpha_2\end{bmatrix}=\begin{bmatrix}3\mathbf{p}_{12}\cdot\mathbf{t}_1\\ 3\mathbf{p}_{12}\cdot\mathbf{t}_2\end{bmatrix} \tag{2.2.10}$$

把两边都乘上第一个矩阵的逆得到：

$$\begin{bmatrix}\alpha_1\\\alpha_2\end{bmatrix}=\begin{bmatrix}2t_1^2 & \mathbf{t}_1\cdot\mathbf{t}_2\\\mathbf{t}_1\cdot\mathbf{t}_2 & 2t_2^2\end{bmatrix}^{-1}\begin{bmatrix}3\mathbf{p}_{12}\cdot\mathbf{t}_1\\3\mathbf{p}_{12}\cdot\mathbf{t}_2\end{bmatrix} \tag{2.2.11}$$

解即为方程 2.2.12：

$$\begin{aligned}\alpha_1&=\frac{3\left(2t_2^2\left(\mathbf{p}_{12}\cdot\mathbf{t}_1\right)-\left(\mathbf{t}_1\cdot\mathbf{t}_2\right)\left(\mathbf{p}_{12}\cdot\mathbf{t}_2\right)\right)}{4t_1^2t_2^2-\left(\mathbf{t}_1\cdot\mathbf{t}_2\right)^2}\\\alpha_2&=\frac{3\left(2t_1^2\left(\mathbf{p}_{12}\cdot\mathbf{t}_2\right)-\left(\mathbf{t}_1\cdot\mathbf{t}_2\right)\left(\mathbf{p}_{12}\cdot\mathbf{t}_1\right)\right)}{4t_1^2t_2^2-\left(\mathbf{t}_1\cdot\mathbf{t}_2\right)^2}\end{aligned} \tag{2.2.12}$$

这两个“(”的值就是用于图 2.2.1 中所示的实线曲线。现在已经有了把多条曲线连接在一起的数学工具了。事实上，可以通过把第一个曲线段的终点作为下一条曲线段的起点的方式来定义多条曲线段。此外，如果第一条曲线的终点处的切线方向和下一条曲线的第一个端点处的切线方向也相同，可以得到 G^1 连续[Foley97]。如果让切线也有相同的长度，就得到了 C^1 连续。在下面的章节中，将以稍微不同的方式进行介绍。

2.2.1 连接具有 C^1 连续的最小弯曲曲线

如果多条 Hermite 曲线以 C^1 连续的方式连接起来，如图 2.2.2 所示，就必须求解由方程 2.2.13 中的积分得到的方程系统：

$$\frac{\partial}{\partial\alpha_i}\int_0^1\left\|\mathbf{h}_1''(u)\right\|^2+\left\|\mathbf{h}_2''(u)\right\|^2+\ldots+\left\|\mathbf{h}_{k+1}''(u)\right\|^2du=0 \tag{2.2.13}$$

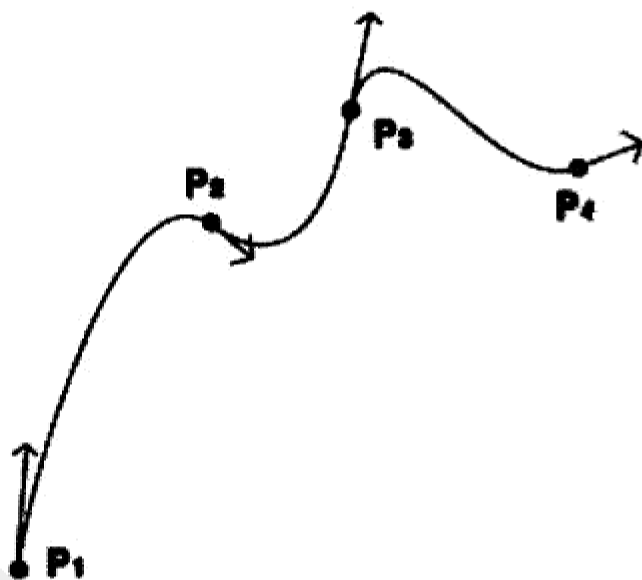


图 2.2.2 三条连接在一起的最小弯曲的 Hermite 曲线

这里并没有列出所有计算，因为它和前面解释的内容非常类似。但是如果有 k 条曲线，则有 k+1 个 α，那么方程系统有 k+1 个未知数。用 $\mathbf{p}_{i,j+1}$ 来表示 $\mathbf{p}_i$ 和 $\mathbf{p}_{i+1}$ 之间的向量，这样就得到

$$\frac{\partial}{\partial\alpha_1}\int_0^1\left\|\mathbf{h}_1''(u)\right\|^2du=8\alpha_1t_1^2+4\alpha_2\mathbf{t}_1\cdot\mathbf{t}_2-12\mathbf{p}_{12}\cdot\mathbf{t}_1 \tag{2.2.14}$$

$$\frac{\partial}{\partial \alpha_2}\int_0^1 \left\|\mathbf{h}_2''(u)\right\|^2 du = 4\alpha_1\mathbf{t}_1\cdot\mathbf{t}_2 + 16\alpha_2 t_2^2 + 4\alpha_3\mathbf{t}_2\cdot\mathbf{t}_3 - 12\mathbf{p}_{12}\cdot\mathbf{t}_2 - 12\mathbf{p}_{23}\cdot\mathbf{t}_2 \tag{2.2.15}$$

$$\frac{\partial}{\partial \alpha_3}\int_0^1 \left\|\mathbf{h}_3''(u)\right\|^2 du = 4\alpha_2\mathbf{t}_2\cdot\mathbf{t}_3 + 16\alpha_3 t_3^3 + 4\alpha_4\mathbf{t}_3\cdot\mathbf{t}_4 - 12\mathbf{p}_{23}\cdot\mathbf{t}_3 - 12\mathbf{p}_{34}\cdot\mathbf{t}_3 \tag{2.2.16}$$

最后得到方程 2.2.17：

$$\frac{\partial}{\partial \alpha_{k+1}}\int_0^1 \left\|\mathbf{h}_{k+1}''(u)\right\|^2 du = 4\alpha_k\mathbf{t}_k\cdot\mathbf{t}_{k+1} + 8\alpha_{k+1} t_{k+1}^2 - 12\mathbf{p}_{k,k+1}\cdot\mathbf{t}_{k+1} \tag{2.2.17}$$

同样，把方程 2.2.14 至方程 2.2.17 除以 4 后，再将它们写成矩阵形式，就和处理等式 2.2.8 得到等式 2.2.10 一样。注意，如果在已经把切线单位化，那么 $t_i^2=1$ 。这会使矩阵更加简单，但是这里并没有这么做。需要求解的系统如方程 2.2.18：

$$\begin{bmatrix} 2t_1^2 & \mathbf{t}_1\cdot\mathbf{t}_2 & 0 & 0 & \cdots & 0 & 0 \\ \mathbf{t}_1\cdot\mathbf{t}_2 & 4t_2^2 & \mathbf{t}_2\cdot\mathbf{t}_3 & 0 & \cdots & 0 & 0 \\ 0 & \mathbf{t}_2\cdot\mathbf{t}_3 & 4t_3^2 & \mathbf{t}_3\cdot\mathbf{t}_4 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \mathbf{t}_k\cdot\mathbf{t}_{k+1} & 2t_{k+1}^2 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_{k+1} \end{bmatrix} = \begin{bmatrix} 3\mathbf{t}_1\cdot\mathbf{p}_{12} \\ 3\mathbf{t}_2\cdot(\mathbf{p}_{12}+\mathbf{p}_{23}) \\ 3\mathbf{t}_3\cdot(\mathbf{p}_{23}+\mathbf{p}_{34}) \\ \vdots \\ 3\mathbf{t}_{k+1}\cdot\mathbf{p}_{k,k+1} \end{bmatrix} \tag{2.2.18}$$

注意非零的项只在主对角线以及紧挨着的上方和下方出现。包含这种矩阵的系统被称为三对角线系统（tridiagonal），它可以通过专门的算法来非常有效地求解[LengyeKM]。

2.2.2 封闭的最小弯曲的曲线

如果封闭循环的曲线需要如图 2.2.3 所示，那么可以简化系统，因为第 1 个点和切线与最后一个点和切线是相同的。这里没有列出所有计算步骤，结果如下：

$$\begin{bmatrix} 4t_1^2 & \mathbf{t}_1\cdot\mathbf{t}_2 & 0 & 0 & \cdots & 0 & \mathbf{t}_k\cdot\mathbf{t}_1 \\ \mathbf{t}_1\cdot\mathbf{t}_2 & 4t_2^2 & \mathbf{t}_2\cdot\mathbf{t}_3 & 0 & \cdots & 0 & 0 \\ 0 & \mathbf{t}_2\cdot\mathbf{t}_3 & 4t_3^2 & \mathbf{t}_3\cdot\mathbf{t}_4 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ \mathbf{t}_k\cdot\mathbf{t}_1 & 0 & 0 & 0 & \cdots & \mathbf{t}_{k-1}\cdot\mathbf{t}_k & 4t_k^2 \end{bmatrix} \begin{bmatrix} \alpha_1 \\ \alpha_2 \\ \alpha_3 \\ \vdots \\ \alpha_k \end{bmatrix} = \begin{bmatrix} 3\mathbf{t}_1\cdot(\mathbf{p}_{k,1}+\mathbf{p}_{12}) \\ 3\mathbf{t}_2\cdot(\mathbf{p}_{12}+\mathbf{p}_{23}) \\ 3\mathbf{t}_3\cdot(\mathbf{p}_{23}+\mathbf{p}_{34}) \\ \vdots \\ 3\mathbf{t}_k\cdot(\mathbf{p}_{k-1,k}+\mathbf{p}_{k,1}) \end{bmatrix} \tag{2.2.19}$$

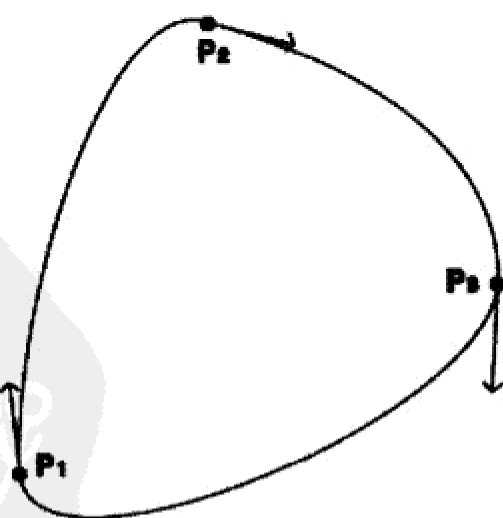


图 2.2.3 由三条最小弯曲的曲线连接在一起的封闭循环曲线

PDG

这个系统给出了 k 个未知量，而不是非闭合曲线的 $k+1$ 个未知数。非零项只出现在左下角和右上角，这样的系统叫做循环三对角线系统。它可以通过应用 Sherman-Morrison 公式，转换成标准的三对角线系统来求解，详情见[Press92]。

2.2.3 总结

本文描述了如何修改三次 Hermite 样条以得到最小加速度曲线（有时也叫最小弯曲曲线）。通过计算曲线加速度的最小平方值并把切线长度设置为最优值，就可以完成此任务。本文还演示了将数条这样的曲线连接起来以达到 C^1 连续所必要的计算，这会得到一个需要求解的简单方程系统。除此之外，还可以用这些曲线构造一个封闭的循环。最小弯曲的曲线可用于曲面的细分，封闭的曲线可以用于摄影机的移动。

2.2.4 参考文献

[Burden89] R. L. Burden and J.D. Faires. *Numerical Analysis*, 439, 440. Boston: PWS-KENT Publishing Company, 1989.

[Foley97] Foley, J. D., et al. *Computer Graphics: Principles and Practice*, 2nd ed, 480. Addison Wesley, 1997.

[Hearn04] D. Hearn and M.P Baker. *Computer Graphics with OpenGL*, 426–429. Pearson Education Inc., 2004.

[Lengyel04] E. Lengyel. *Mathematics for 3D Game Programming and Computer Graphics*, 2nd ed., 433–436. Charles River Media, 2004.

[Overveld97] C.W.A.M. van Overveld and B.Wyvill. “An algorithm for polygon subdivision based on vertex normals,” *Computer Graphics International*, 3–12. June 23–27, 1997.

[Press92] W. H. Press, et al. *Numerical Recipes in C*, 74–75. Cambridge University Press, 1992.

[Vlachos01] A. Vlachos and J. Isidoro. “Smooth C^2 Quaternion-based Flythrough Paths.” In *Game Programming Gems 2* (Mark Deloura, ed.), 220–227. Charles River Media, 2001.

2.3　动画中基于样条的时间控制

Red Storm Entertainment 公司，James M. Van Verth
jimvv@redstorm.com

让物体沿着一个由参数曲线 $\mathbf{Q}(u)$确定的路径移动，是动画系统中的一个标准问题。对大部分参数曲线来说,使用标准的参数化方法无法获得恒定的速度，因此通过距离 d 重新参数化曲线得到 $\mathbf{Q}(d)$就十分有必要。为了获得常数速度,函数 $d(t)$必须是线性的，除非使用其他函数把时间映射到距离上去。

早先的论文（[Olsen00]，[Krome04]）已经讨论过使用 erase-in/ erase-out 函数处理沿着曲线的速度。本文描述该如何使用分段样条来构造通用的距离-时间函数。可以使用各种不同的参数来约束这个函数。举个例子，有一个距离的关键点，要求特殊的时间里必须对应特殊的点。这个关键点里还需要一个速度项，表示在特殊时间点的特殊速度。低特性参数会在每个关键点上设置，像快进/快出（快速靠近/快速离开）、慢进慢出（靠近/离开的时候速度为 0）和平滑（尽可能地平稳移动）。我们的目标是为动画师使用内建的动画工具提供一个具有弹性的系统，特别是为游戏中的影片做的摄影机动画。

为了构建距离-时间函数，我们要使用分段 Hermite 样条。假设的前提是读者已经了解这种样条，有关它们的基本信息，可参考[Burden93]和[Rogers90]。

2.3.1　开始

构建这个特殊函数的目的，是为了在空间中沿着一条确定路径移动时可以控制速度。这条路径通常由一条参数曲线产生,但鉴于我们的目的，我们并不关心它是线性函数、贝塞尔曲线,还是 B 样条。我们只关心 $\mathbf{Q}(u)$的一般功能：输入 u 的值,得到 3D 立体空间的点 $\mathbf{Q}$。当 u 增加时，输出将是空间内的一条轨迹。

在加入我们自己的时间控制之前，必须把需要沿着曲线长度以恒定速度移动的物体使用惟一的参数 u 来表示。但是，对于三次曲线来说（通常情况下），沿着曲线的距离和参数 u 并不是线性关系。在某些地方，为获得步长Δu，我们沿着曲线的移动距离可能很短；在另外一些地方，则可能比较长。这是因为为了提供我们需要的曲率，各处的一阶导数就必须是变化的，因此沿着曲线方向的速度也必须是变化的。

显然，如果要维持恒速，这就不是我们想要的。在曲线的一些区域，我们的移动要比另外一些地方快。解决的办法是依靠距离重新参数化曲线。我们并非通过使用参数 u 来确定曲线上的一个点，而是通过从曲线开始的位置沿着曲线方向增加的距离参数 d 来得到曲线上的点。当以一个恒定速率增加 d 的时候，曲线上的移动也有一个恒定的速率。在大部分情形下，用解析法为一个三次曲线重新参数化并不实用。处理这个问题需要用到数值方法，通常会涉及求根和生成查找表，详细信息请参见[Eberly01]、[Parent02]或者[VanVerth04]。鉴于本文的目的，假设我们的曲线已经有了这样的参数。

2.3.2 一般的距离-时间函数

比起依赖距离沿着曲线移动，我们通常更应该根据时间的增加进行移动，即确定时间 t 时我们在曲线上的位置。因此，需要用一些方法把时间转换成距离，并把它作为重新参量化曲线的输入。通过一个距离-时间函数 $d(t)$可以描述它，这个距离函数的自变量是时间。在时间 t 时，曲线上一个相应的点可以通过计算 $Q(d(t))$求得。例如，以恒速行进的线性函数，是一个开始于(0,0)、结束于时间和位置最大值处的函数。这通常会被归一化，以使时间和距离的最大值都是 1（图 2.3.1），以便它能够在多条曲线上应用。为了调整输入 t，使之适用于归一函数，可以使用公式 2.3.1：

$$\bar{t} = \frac{t - t_s}{t_e - t_s} \tag{2.3.1}$$

t_s 和 t_e 分别是曲线开始和结束的时间。为了调整输出，要将曲线的全长乘上结果 $d(t)$，然后将其插入重构参数曲线以获得最终位置。出于简化的目的，我们假定默认情况下都会进行这种校正，且所有时间和距离的值都在 0～1 之间。

没有理由限制距离-时间函数必须是线性函数。看看另外一个例子：ease-in/ease-out 函数（图 2.3.2）。计算这一类型的函数有不少方法（[Parent02]，[Olsen00]），它们都有相同的基本形态。用这种方法作为距离-时间函数能得到如下结果：在曲线开始的时候速度为 0，在曲线的中部达到最大速度，然后慢下来在末端回到 0。这使得曲线上的移动看起来非常自然。和在开始的时候突然给它一个速度，然后沿着曲线维持这个速度，最后在末尾突然停止相比，这种做法更像是移动一个物体所需要的加速和减速的过程。

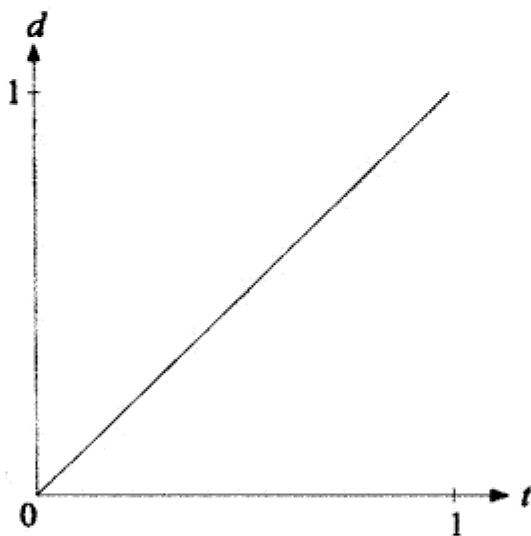


图 2.3.1 线性的距离-时间函数

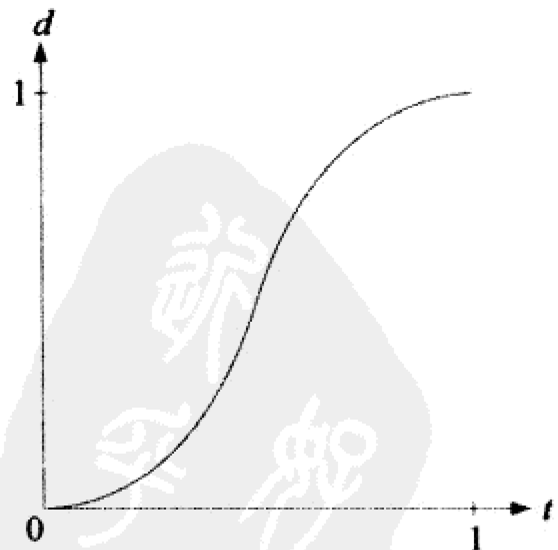


图 2.3.2 Ease in/Ease Out 的距离-时间函数

我们并不准备到此为止。我们可以使用 t 在范围[0，1]内的任何函数，也就是说它没有处于[0，1]以外的范围（即时间和距离都被限制在正常的范围之间）。在基本的约束之外，该怎样安排控制沿曲线移动方式的函数呢？图 2.3.3 显示了一个斜率总是非负的距离-时间函数。这种情况下，当时间 t 增加时，我们不会沿着曲线向后移动。如果这个函数在任意点都有一个负斜率，那么在那个片段中我们将沿着曲线向后移动。图 2.3.4 显示了这样一条曲线，灰色区域表示负斜率的段。注意，在这个函数中，开始的时候有延迟，然后提前到达并等待。使用这种技术使我们在控制路径动画中的速度和到达时间方面拥有了很大的弹性。

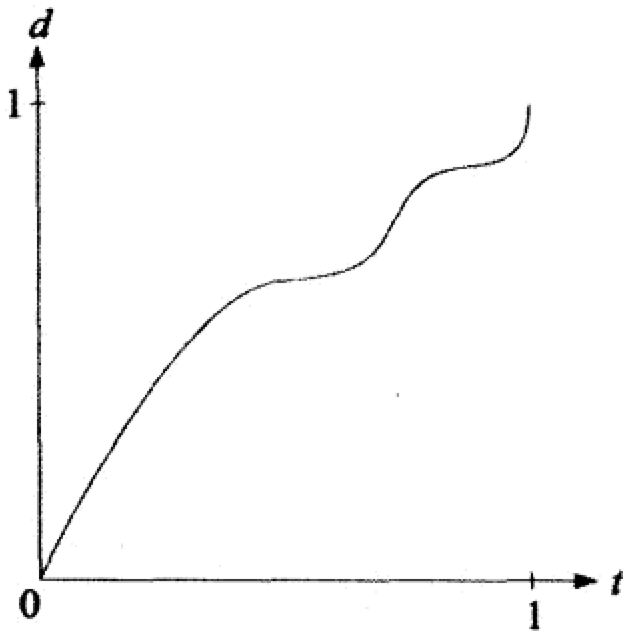


图 2.3.3 非负斜率的距离-时间函数

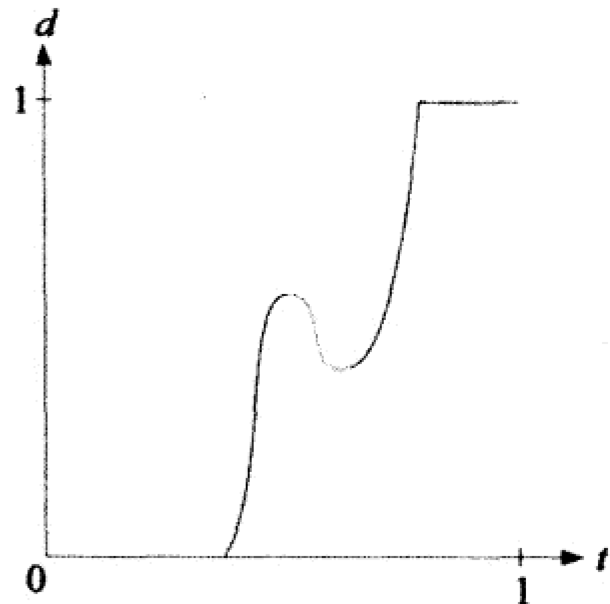


图 2.3.4 距离-时间函数，灰色区域表示负斜率的段，并演示了延迟离开和提前到达

2.3.3 根据样条构造距离-时间函数

一般情况下，我们可以使用符合前面提到的要求的任何函数，但是为了更容易构造，使用分段 Hermite 样条曲线会具有一些优势。它们的计算非常简单，并且端点的位置和速度可以用非常简单的方法来控制。不过，这里使用它们的方法会和通常介绍的方法有些不同。因为仅仅是通过距离而不是通过空间的点进行插值，所以这些位置是实数而不是多维向量。同样，每个控制点的正切都变成了曲线在该点的斜率，而速率则变成了速度。

位置在 p_k 和 p_{k+1} 之间，斜率为 p'_k 和 p'_{k+1} 的 Hermite 曲线的标准定义如公式 2.3.2 所示：

$$H(t)=\left(2t^3-3t^2+1\right)p_k+\left(-2t^3+3t^2\right)p_{k+1}+\left(t^3-2t^2+t\right)p'_{k,0}+\left(t^3-t^2\right)p'_{k+1,1} \quad (2.3.2)$$

由于参数 t 在[0,1]之间，这样的曲线也叫做常态 Hermite 曲线。在这个例子中，有 p_0 到 p_n 多个样本位置。每个，位置 p_k 都有两个斜率：向外的斜率 $p_{k,0}$ 和向内的斜率 $p_{k,1}$。如果想获得一条光滑的曲线，这些斜率就是相匹配的，如果想获得一条“扭转”的曲线，这些斜率就是不匹配的。为了产生一条从 p_0 到 p_n 的连续函数，可以通过在后续的两个位置之间进行插值来创建子曲线。子曲线 H_0 被插值到 p_0 和 p_1 之间，H_1 被插值到 p_1 和 p_2 之间，等等。这就叫做分段 Hermite 曲线。图 2.3.6 至 2.3.9 显示了一些这样的曲线。

在这个例子中，位置是用沿着路径的距离的值表示的。每个距离值 p_k 都对应一个时间 t_k，就是到达这段距离的时间。每一对值叫做一个距离键。t_k 的值是逐渐增加的，因此 $t_k \leqslant t_{k+1}$。

这意味着每个子曲线在它自己的局部空间内的间隔不一定是[0,1]，而是都有不同的间隔$[t_k,t_{k+1}]$。显然，我们不能使用标准的定义。解决办法是用更加复杂更加具有弹性的方法来表示 Hermite 曲线。

这分为两个部分。首先，必须把输入的时间转换为可以使用的标准规格。给定一个时间 t，首先获得一条 $t_k \leqslant t \leqslant t_{k+1}$ 的子曲线。然后使用公式 2.3.1，其中 $t_s=t_k$ 且 $t_e=t_{k+1}$。和以前一样，这将把 t 映射到[0,1]，也就是 Hermite 曲线使用标准规则的范围内。然后必须修正每个采样位置的斜率。原始的斜率假定 t 和 d 用相同单位数量移动。然而，我们通过 $1/(t_{k+1} - t_k)$缩放了 t 值，所以要通过$(t_{k+1} - t_k)$ 来缩放斜率，得到正确的 t。因此，对于给定的 p'_k，$\overline{p}'_k = p'_k(t_{k+1} - t_k)$。这两项调整相结合就可以将时间输入应用到 Hermite 曲线的标准等式中去。

这样，计算距离-时间函数就是一件很简单的事情：通过在 $t_k \leqslant t \leqslant t_{k+1}$ 搜索时间键，获得正确的子曲线。通过它计算$\overline{t}$，然后把结果用于校正过子曲线斜率的标准 Hermite 形式中。其结果是距离值 d。但是，第 1 个位置上的 Hermite 样条该如何创建呢？

1. 引入和引出速度

一种产生基于样条的距离-时间函数的方法是在每一键指定一个引入（Incoming）和引出（Outgoing）的速率，这就像使用分段 Hermite 曲线的端点处斜率一样。在设置速度的时候，有几种可能。首先，用户可以定义它们。通常情况下，用户希望在游戏世界内指定速度，因此必须把它们在单位化的距离-时间函数里转换成等价的速率，方法是乘以想得到的空间曲线的总的时间，并除以曲线的总长度。像以前一样，这些速率必须乘以子曲线的时间间隔，来为非单位化的 Hermite 曲线做校正。值得关注的一点是，当用户用一个很大的数量级设置一个速率的时候，曲线会在我们想要的范围[0,1]之外循环（图 2.3.5）。但是，如果显示一个图形并提供足够的错误反馈，这种做法也很有效。

对于另一种方法，可以通过一些标准的方法确定缺省值，并且还可以使用一些组合方法创建函数，且使用者不必直接设定斜率。标准的动画用语是慢入（slow-in）、慢出（slow-out）、快进（fast-in）和快出（fast-out）。“进（-in）”和“出（-out）”分别是指关键点（key）的到达和离开。慢就意味着这个关键点的速率是 0。快很少有很好的定义，但基本意味着物体会尽可能快地离开一个关键点。假定那些关键点随距离渐增，且用户希望避免物体沿着曲线倒退，我们定义“快”以使沿着最终曲线的最小速率是 0。如果关键点随距离而渐减，做法刚好相反，以使最大的速率是 0。从一个关键点慢出到在另一个关键点慢入，可以得到一个类似于 ease-in/ease-out 的曲线。快出到快进可得到相反的曲线：快速的出发，接着减慢到 0，接着速率再次回升。图 2.3.6 显示的是一个从快出到快进，接着从慢出到慢进的函数。

设置慢入和慢出很简单：只需要把在那个关键点出入的速度设成 0 即可。如果想得到接近于 ease-in/ease-out 的曲线，可以在最初的关键点中间的平均点增加一个附加的关键点，并给这个关键点设置快进/快出的速率。

通过使用两个固定值可以得到快进和快出的速率。首先，曲线中点的速率是 0。为此，可以复制一个标准的 Hermite 曲线在中点的导数，并把它设置成等于 0 来实现这一点，如等式 2.3.3 中所示：

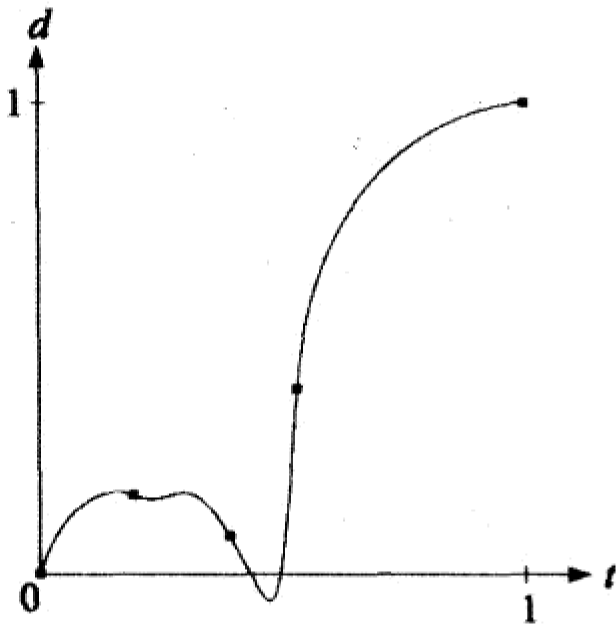


图 2.3.5 用户定义的过大的速率导致无效的距离-时间函数

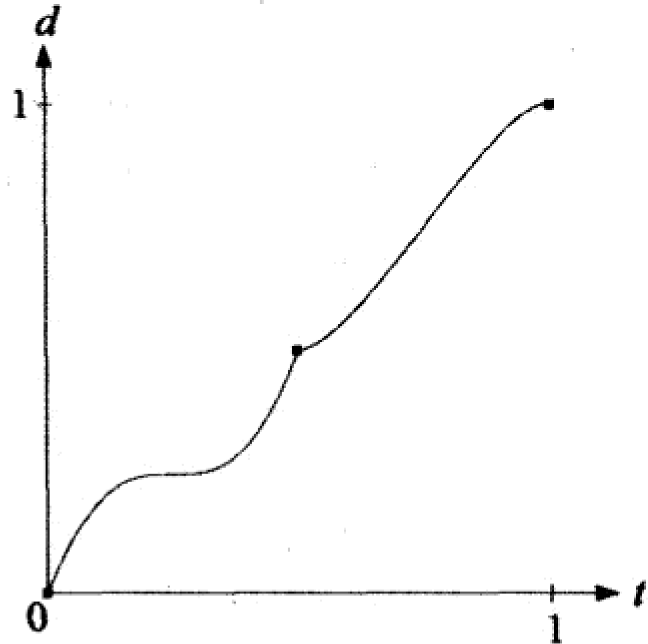


图 2.3.6 紧接在慢出/慢入后面的快出/快入

$$\begin{aligned}0&=H'(1/2)\\&=\left(6(1/2)^2-6(1/2)\right)p_k+\left(-6(1/2)^2+6(1/2)\right)p_{k+1}\\&\quad+\left(3(1/2)^2-4(1/2)+1\right)p'_{k,0}+\left(3(1/2)^2-2(1/2)\right)p'_{k+1,1}\\&=6p_{k+1}-6p_k-p'_{k,0}-p'_{k+1,1}\end{aligned} \tag{2.3.3}$$

其次，子曲线起始的速率需要和终点的速率相等。可以把等式 2.3.3 改成等式 2.3.4：

$$\begin{aligned}0&=6\left(p_{k+1}-p_k\right)-2p'_{k,0}\\p'_{k,0}&=3\left(p_{k+1}-p_k\right)\end{aligned} \tag{2.3.4}$$

这个速度将用于关键点 k 的快出和关键点 k+1 的快进。可以在单位化的 Hermite 等式中用这个速度，因此不必像在其他情况下那样用（$t_{k+1}-t_k$）校正它。但是，如果校正所有的速率，就能用 $3(p_{k+1}-p_k)/(t_{k+1}-t_k)$代替快进和快出速度。

还存在其他的标准参数。线性意味着距离-时间曲线是一条从一个关键点到下一个关键点的直线。开始关键点的引出速率和终止关键点的引入速率，被设置成由两点确定的直线的斜率。步幅是指距离-时间曲线在一个关键点停留，直到时间间隔消逝，然后立即跳到下一个关键点。这并不像实现一个单独的样条那么方便，因为曲线里有一个中断。解决方法是在和第 1 个关键点相同的距离上创建一个新关键点，但是在时间上要在第 2 个关键点之前，接着创建两个线性步幅：从第 1 个关键点到隐藏关键点，然后从新关键点到第 2 个关键点。剩下的就是在该点分割样条曲线，并开始一条新的样条曲线。图 2.3.7 显示了被一个步幅分割的两个线性段。

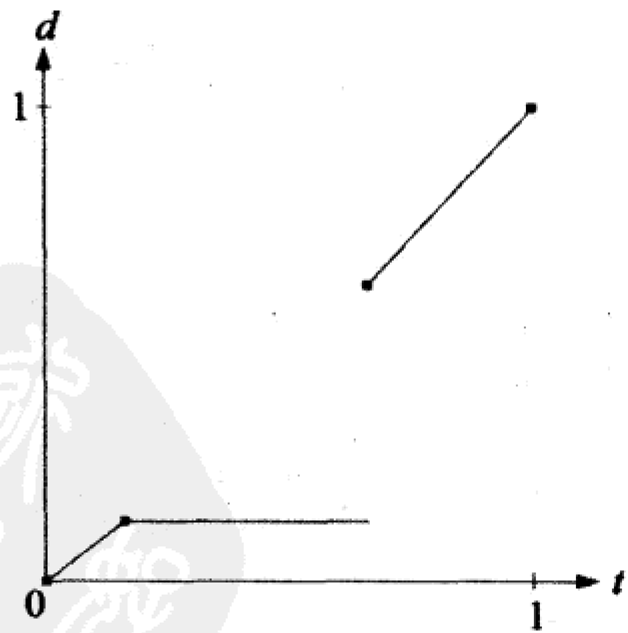


图 2.3.7 显示了线性段和一个步幅键的距离-时间曲线

图 2.3.6 和图 2.3.7 显示出在一个给定的关键点处，引入的速率和引出的速率并不一定要匹配。在不追求物理上的真实时候，让物体慢慢地到达一个点，然后马上以很高的速度跑掉，这在许多时候是很有用处的。举另外一个例子，动画师们可以用快进/慢出给事件创建一个快速反应时间和一个缓慢的恢复。同样，没有必要在一个给定的子曲线上匹配进出速率。例如，可以在一个关键点的开始有慢出，下一个关键点的结束有快进。最终的结果会使一个物体在从第 1 个关键点以很高的速度到达第 2 个关键点的时候狂跳。具体情况完全要根据动画师的需求。

2. 自动曲线生成

设置速度方向的一个替代方法是设置一个距离关键点的序列，就像前面描述的那样，接着自动产生在这些关键点之间插值的子曲线。有了定义每个子曲线端点的“位置”，我们仍然需要可以在分段样条曲线中使用的每个关键点的速率。我们通常想得到一段光滑的曲线，所以在这种情况下，会假定引入的速率和引出的速率是相同的。可用的方法有很多，但是能获得最光滑的结果的方法是使用一个普通的分段 Hermite 样条。这涉及设置一系列线性方程组，保证曲线内部点处的 C^2 连续，并保证端点处的 2 阶导数为 0。更多细节请参看[Burden93]和[Rogers90]里找到。对于一个非单位化的 Hcrmite 样条，其看起来像等式 2.3.5：

$$
\begin{bmatrix}
2 & 1 & & & \\
\Delta t_0 & 2(\Delta t_0+\Delta t_1) & \Delta t_1 & & \\
 & & \ddots & & \\
 & & \Delta t_{n-2} & 2(\Delta t_{n-2}+\Delta t_{n-1}) & \Delta t_{n-1} \\
 & & & 1 & 2
\end{bmatrix}
\begin{bmatrix} p_0' \\ p_1' \\ \vdots \\ p_{n-1}' \\ p_n' \end{bmatrix}
=
\begin{bmatrix}
\dfrac{3}{\Delta t_0}(p_1-p_0) \\
\dfrac{3}{\Delta t_0\Delta t_1}\left[\Delta t_0^2(p_2-p_1)+\Delta t_1^2(p_1-p_0)\right] \\
\vdots \\
\dfrac{3}{\Delta t_{n-2}\Delta t_{n-1}}\left[\Delta t_{n-2}^2(p_n-p_{n-1})+\Delta t_{n-1}^2(p_{n-1}-p_{n-2})\right] \\
\dfrac{3}{\Delta t_{n-1}}(p_n-p_{n-1})
\end{bmatrix}
\tag{2.3.5}
$$

其中 $\Delta t_k=(t_{k-1}-t_k)$。解 $p_0',\ldots,p_n'$ 的线性方程组可以得到每个关键点的斜率以及建立 Hermite 样条所需要的信息。因此左边的矩阵是稀疏的斜对角矩阵，所以可以在线性时间内求解。[Bur-dcn93]中有更多的相关细节。注意这个解决方案没有为非单位化的时间间隔校正斜率，因此仍然需要用适当的Δt_k去乘以它们。在图 2.3.8 中可以看到一个这样的距离-时间曲线。

3. 光滑组合和速度规范

一般来说，使用者通常需要创建一条在某个关键点处具有固定的终止和起始速率的曲

线，其余的自动生成区域也一样。为此，我们需要加入平滑切入点和切出点。一个平滑切入切出关键点的序列表示了用户希望自动生成的曲线区域。如果考察一个平滑关键点，我们会从跟踪一个曲线的光滑区域开始，并逐渐细化线性系统的参量，直至到达一个并不光滑的关键点。然后，将这些参数代入对角矩阵，求解生成这段曲线区域的斜率。

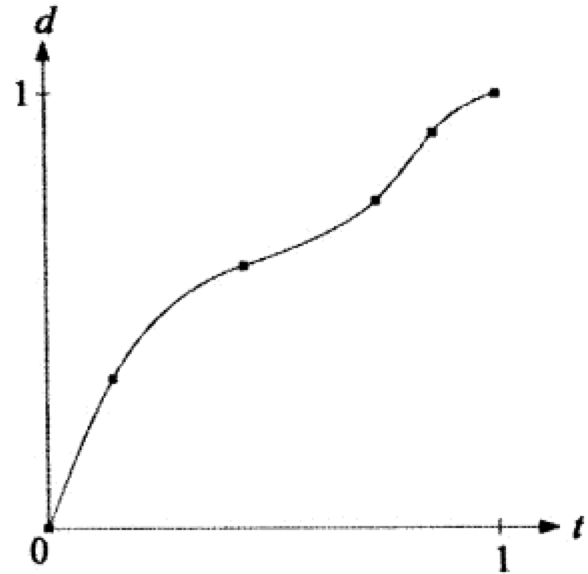


图 2.3.8 通过距离键用普通曲线创建的距离-时间函数

由于系统是弹性的，在光滑区域的末端点最终可以获得指定的速度。例如，假设切入（smooth-in）关键点后跟着一个慢出（slow-out）关键点。这条曲线开始的速率是 0，并和其后的距离关键点平滑混合。最初的端点裁减条件是已知的。为它创立矩阵只是对自然样条设置的修正。在对角线中用 1 代替第 1 个矩阵的行，并把右边向量中对应的项用指定速率代替。如果是在一个不平滑关键点处结束，我们也要对最后一个矩阵行做同样的处理。所以，从这个例子来看，线性系统和方程式 2.3.6 相像。

$$\begin{bmatrix} 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} p'_0 \\ p'_1 \end{bmatrix} = \begin{bmatrix} 0 \\ \dfrac{3}{t_1 - t_0}(p_1 - p_0) \end{bmatrix} \tag{2.3.6}$$

下列伪代码显示了对平滑部分的跟踪是如何处理的。我们遍历每一个关键点，并设置给定的速率或为平滑区域设置参量。inSmooth 的布尔类型变量是用来表明我们当前是否在追踪一个平滑部分。

```
inSmooth = false
for each key do
    if current in-speed is not smooth
        if inSmooth
            finish clamped spline
            inSmooth = false
        else
            set given speed
     else
        if !inSmooth
            start clamped spline
            inSmooth = true
        if not at end and out key is smooth
            add to middle of smooth spline
        else
            finish natural spline
            inSmooth = false

     if current out-speed is not smooth
        set given speed
     else if !inSmooth
```

```
start natural spline
inSmooth = true
```

为了清晰起见，一些细节已经被忽略。例如，这里不考虑第 1 个关键点的切入速率或最后的关键点的切出速率，因为它们是无效的。完整的细节可以在示例代码中找到。

4. 例子

作为一个例子，我们假定用户已经用如下的速率参量设置了 3 个距离-时间对：

Time（时间）	Distance（距离）	In-Speed（切入速率）	Out-Speed（切出速率）
0.0	0.0	—	Linear
0.45	0.60	Fast	Smooth
1.0	1.0	Slow	—

第 1 个关键点上的切入速率是线性的，所以它的数值是(0.6～0.0)/(0.45-0.0)，不过这里通过乘以（0.45-0.0）对此进行了纠正，所以最后的存储速率是 0.6。第 2 关键点的切入速率是快出（fast-out），所以它的值是 3(0.6 - 0.0)。第 2 个关键点上的切出速率是平滑的，所以要开始构造一个线性系统。在这个简单的例子中，可以在下一关键点处立即停止构造因为它慢下来并且不平滑。这个曲线区域的线性系统和公式 2.3.7 类似。

$$\begin{bmatrix} 2 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} p'_{2,0} \\ p'_{3,1} \end{bmatrix} = \begin{bmatrix} 3\dfrac{1.0-0.6}{1.0-0.45} \\ 0.0 \end{bmatrix} \tag{2.3.7}$$

解此方程得到中间值 $p_{2,0}'=1.09091$ 和 $p_{3,1}'=0.0$ 。再用(1.0-0.45) = 0.55 校正了以上两个数据后得到了最终数值 $p_{2,0}'=0.6$ 和 $p_{3,1}'=0.0$ 。这样 Hermite 曲线的参量为：

Time（时间）	Distance（距离）	In-Speed（切入速率）	Out-Speed（切出速率）
0.0	0.0	—	0.6
0.45	0.60	1.8	0.6
1.0	1.0	0.0	—

最终的曲线可以在图 2.3.9 中看到。

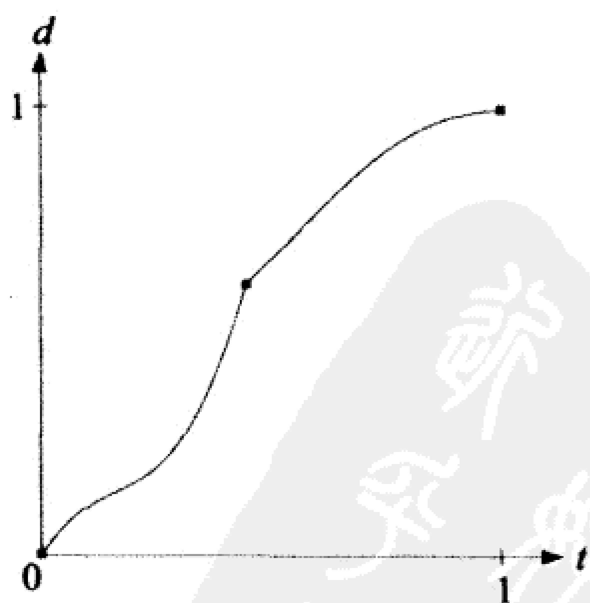


图 2.3.9 根据例子中的键值创建的距离-时间函数

2.3.4 接口选择

虽然早期的一些资料为该技术提供了数学基础，但并不理想，除非它是可控的。其中一个可能性是为空间曲线提供一个接口，用来在每个插值控制点设置到达的时间，然后使用沿着曲线到这些点的距离为距离-时间函数创建关键点。平滑的参量和（或）引入（引出）速率也可以在空间控制点上设置。如此一来，能够展示当前的距离-时间图，并用距离关键点来测点就变得很有用。这些点可以被单击并在时间轴上从左移到右侧。因为被固定在空间曲线的位置上，所以它们的距离值是不可以改变的。如果函数不在 0 和 1 之间，那就应该报告有错误发生。

另外，空间曲线和距离-时间函数可以单独设置，所以函数有一个完整独立的关键点集合。但是，在这个例子中，比较明智的做法是把图标放在距离-时间图上以显示空间控制点的位置，从而使用户对那些点上的到达时间有所认知。到达时间数据还可以复制回空间曲线的显示图。

还有一种折中的方法，即从空间控制点派生出起始距离关键点，并添加一些和空间没有关系而只是用于控制距离-时间函数的附加点。

2.3.5 总结

本文展示了一个使用分段 Hermite 样条为动画创建距离-时间函数的方法。Hermite 样条允许很多的用户输入，尤其是曲线上的切线，从而为管理速度控制提供了一种直观的方法。样条的自动创建，如自然样条，以及速度控制的默认设置对用户快速创建距离-时间函数也是非常有用的。还可以把这些想法扩展到其他类型的样条上，比如分段的贝塞尔曲线或者 B 样条，只要能满足距离-时间函数的基本需求即可。

这项技术也可以用在其他应用上。例如，用于四元数插值的 slerp 函数，将 0 和 1 之间的 t 值映射为两个插值结果，这两个插值结果也在 0 和 1 之间。这两个插值结果用来混合两个四元数。整个函数通常需要 3 个正弦和一个浮点除法。每个插值函数可以用分段的 Hermite 曲线来近似。其结果虽然没有 slerp 准确，但是会更快，而且效果比直接线性插值要好。

2.3.6 参考文献

[Burden93] Burden, Richard L. and J. Douglas Faires. *Numerical Analysis*. PWS Publishing Company, 1993.

[Eberly01] Eberly, David. "Moving at Constant Speed." Available online at *http://www.magic-software.com*. January 2001.

[Krome04] Lowe, Thomas. "Critically Damped Ease-In/Ease-Out Smoothing." In *Game Programming Gems 4*, 95–101. Charles River, 2004.

[Olsen00] Olsen, John. "Interpolation Methods." In *Game Programming Gems*, 141–149. Charles River, 2000.

[Parent02] Parent, Rick. *Computer Animation: Algorithms and Techniques*. Morgan Kaufmann Publishers, 2002.

[Rogers90] Rogers, David F. and J. Alan Adams. *Mathematical Elements for Computer Graphics*. McGraw-Hill, 1993.

[VanVerth04] Van Verth, James M. and Lars M. Bishop. *Essential Mathematics for Games and Interactive Applications*. Morgan Kaufmann Publishers, 2004.

2.4 快速四元数近似插值

Andy Thomason
athomason@acm.org

由于在处理物体旋转方面的简单和高效，四元数在游戏开发中有着广泛的应用。四元数仅采用一个旋转矩阵所需的 4/9 的数据量便可以平滑地进行插值和旋转，而且它在处理蒙皮和层次动画（hierarchical animation）方面具有其他很多优点。特别是，四元数能用来描述角色仅能绕特定点旋转的关节点。

游戏越来越复杂，我们必须使用一些与超级计算机所使用的线性代数算法类似的更高级的方法。Structures Of Arrays（SOA）方法可以通过将大量相似的操作组织到一起进行排序来改善计算效率和减少内存访问，以及通过处理单指令多数据（Single Instruction,Multiple Data，简称 SIMD）指令来使用所有可能的数学逻辑单元（Arithmetic and Logic Units，简称 ALU）。

使用批处理线性代数方法的一个不足是：与四元数插值相关联的三角函数无法被使用。因此，通过使用近似方法，虽然牺牲了一些精确性，但我们不仅可以得到速度的提升，而且使用加减乘除和平方根便可以完成计算。

在游戏《Galleon》中，我们处理了很多角色的近距离格斗，每个角色都有布料系统、实时的脚步声定位，以及 AI。如果在同样的硬件上使用成千上万个这样的角色，载入蒙皮和动画将占用相当大的花销，因为在做关键帧扩充、动作混合和碰撞时所用的四元数插值的花销将非常大。

通过 Vertex Shader 增加的复杂性，现在可以使用四元数进行蒙皮（见[Hejl04]）。我们可以从动画数据中使用近似方法生成一批四元数，并将它们发送向 Vertex Shader。

这些一度只适用于专用的向量处理单元（vector units）的技术渐渐在游戏机（Console）以及计算机类平台上变得很普遍。本文的例子是用 C++ 提供的，但可以很方便地用汇编指令重写它们。

2.4.1 使用四元数来表示旋转

回忆一下，四元数 $\mathbf{q}$ 是一个原型如下的由四个分量组成的数：

$$\mathbf{q}=\langle w,x,y,z\rangle = w+xi+yj+zk \qquad (2.4.1)$$

它由一个标量 w 和三个矢量 x、y 和 z 组成。四元数乘法被定义为使用通常的分配律，其“虚部”i、j 和 k 满足下面的等式：

$$\begin{aligned} i^2 = j^2 = k^2 &= -1 \\ ij = -ji &= k \\ jk = -kj &= i \\ ki = -ik &= j \end{aligned} \tag{2.4.2}$$

通过这些法则，四元数 **a** 和 **b** 的乘法可以展开为：

$$\begin{aligned} \mathbf{ab} = &\left(a_wb_w - a_xb_x - a_yb_y - a_zb_z\right) \\ &+3\left(a_wb_x + a_xb_w + a_yb_z - a_zb_y\right)i \\ &+3\left(a_wb_y - a_xb_z + a_yb_w + a_zb_x\right)j \\ &+3\left(a_wb_z + a_xb_y - a_yb_x + a_zb_w\right)k \end{aligned} \tag{2.4.3}$$

任何非零四元数 $\mathbf{q} = w + xi + yj + zk$ 都有其逆四元数 q^{-1}：

$$\mathbf{q}^{-1} = \frac{\overline{\mathbf{q}}}{q^2} \tag{2.4.4}$$

其中 $\overline{\mathbf{q}} = w - xi - yj - zk$ 叫做 **q** 的共轭四元数。对于单位四元数 $q^2 = 1$，其共轭四元数和逆四元数具有同样的取值。

四元数通过单位长度的轴 $\mathbf{A} = \left\langle A_x, A_y, A_z \right\rangle$ 和角度 θ 来表示旋转，它经常被写做如下的形式：

$$\mathbf{q}_{\text{rotation}} = \cos\frac{\theta}{2} + \mathbf{A}\sin\frac{\theta}{2} \tag{2.4.5}$$

用四元数对矢量 **v** 进行旋转时，可以把向量看做是一个标量为 0 的四元数，并用如下的积进行计算：

$$Rotate\left(\mathbf{v},\mathbf{q}\right) = \mathbf{qv\overline{q}} \tag{2.4.6}$$

这可以推导出我们熟悉的四元数-矩阵转换公式：

$$\mathbf{qv\overline{q}} = \begin{bmatrix} 1-2q_y^2-2q_z^2 & 2q_xq_y-2q_wq_z & 2q_xq_z+2q_wq_y \\ 2q_xq_y+2q_wq_z & 1-2q_x^2-2q_z^2 & 2q_yq_z-2q_wq_x \\ 2q_xq_z-2q_wq_y & 2q_yq_z+2q_wq_x & 1-2q_x^2-2q_y^2 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{2.4.7}$$

注意，如果不做如下假设：

$$q_w^2 + q_x^2 + q_y^2 + q_z^2 = 1 \tag{2.4.8}$$

就可以给出一个替代公式：

$$\mathbf{qv\overline{q}} = \begin{bmatrix} q_w^2+q_x^2-q_y^2-q_z^2 & 2q_xq_y-2q_wq_z & 2q_xq_z+2q_wq_y \\ 2q_xq_y+2q_wq_z & q_w^2+q_y^2-q_x^2-q_z^2 & 2q_yq_z-2q_wq_x \\ 2q_xq_z-2q_wq_y & 2q_yq_z+2q_wq_x & q_w^2+q_z^2-q_x^2-q_y^2 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ v_z \end{bmatrix} \tag{2.4.9}$$

且可以像下面那样在四元数中合并一个缩放系数 s：

$$Rotate\left(\mathbf{v},\pm\sqrt{s}\mathbf{q}\right) = \mathbf{qv\overline{q}}s \tag{2.4.10}$$

这个公式当且仅当使用共轭四元数，而不是逆四元数的时候才起作用。很明显，缩放系数不能为负。同时，注意对四元数取负不会对旋转结果产生影响，也就是说：

$$Rotate\left(\mathbf{v},-\mathbf{q}\right) = Rotate\left(\mathbf{v},\mathbf{q}\right) \tag{2.4.11}$$

在大部分情况下，下面章节所讨论的技术都会假定缩放系数是单位化的，因此请特别注意非单位化的四元数的用法。不过，当在 Vertex Shader 中使用四元数时，额外的比例系数将会非常有用。

2.4.2 四元数旋转插值

在电脑游戏中，动画是由关键帧组成的，通常旋转表示角色关节的角度。为了在不同的关键帧之间平滑插值，并避免使用宝贵的存储器来储存每一帧的信息，这里使用了一个被称为球面线性插值（spherical linear itterpolation，简称 slerp）的方法。slerp 方法的目的是平滑地在两个四元数 **a** 和 **b** 间进行插值，它会在保证矢量为单位长度的前提下，让矢量在单位时间内扫过常数角度。

为什么不能直接使用线性插值呢？答案就是我们需要保留插值四元数的单位长度，以避免引入缩放系数。即使重新单位化线性插值的结果，旋转动画的角速度仍然不会是常数，而是不稳定的角速度，从而导致不平稳的移动。

在现实生活中，一个 slerp 的例子是客机沿地球表面所做的圆弧运动。飞机与地心的距离是一个常数，而且以常数速度绕着最短的弧线进行运动。这样，如果需要从阿姆斯特丹飞往柏林，就满足函数：

$$Slerp(Amsterdam, Berlin) \tag{2.4.12}$$

怎么完成这一点呢？

在图 2.4.1 中，矢量 **a** 表示阿姆斯特丹，矢量 **b** 表示柏林。假设在一个小时的旅途中，喷气机的位置 **p** 沿着弧线运动了弧度 θ。那么在时间 t 处，飞机移动了 θt 弧度。

这个位置可以由两个矢量 **a** 和 **b** 的线性混合来描述：

$$\mathbf{p}(\theta,t)=\alpha(\theta,t)\mathbf{a}+\beta(\theta,t)\mathbf{b} \tag{2.4.13}$$

通过图中划出的三角形可以算出 **a** 和 **b**。对 $op=1$，$oc=\alpha$，$cp=\beta$ 的三角形 **opc**，使用正弦公式，可得到下面的等式：

$$\frac{1}{2}\alpha\beta\sin\theta=\frac{1}{2}\alpha\sin(t\theta)=\frac{1}{2}\beta\sin((1-t)\theta) \tag{2.4.14}$$

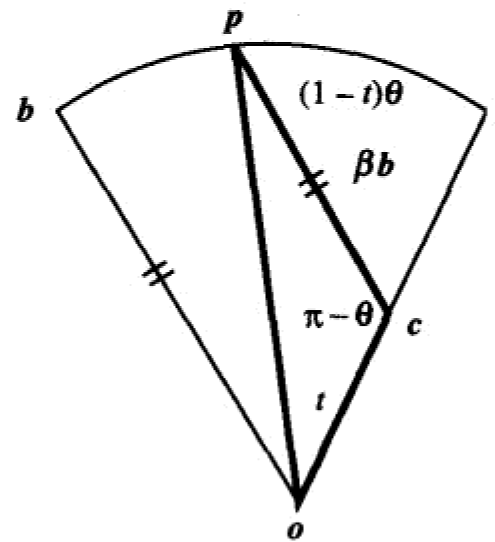


图 2.4.1 slerp 函数的演示

这个等式可以推导出一个非常知名的公式，通过它可以得到 α 和 β：

$$\mathbf{p}(\theta,t)=Slerp(\mathbf{a},\mathbf{b},t)=\frac{\sin(1-t)\theta}{\sin\theta}\mathbf{a}+\frac{\sin t\theta}{\sin\theta}\mathbf{b},\quad \theta=\arccos(\mathbf{a}\cdot\mathbf{b}) \tag{2.4.15}$$

这样，如果 x=**a** • **b**，那么：

$$\alpha(x,t)=\frac{\sin((1-t)\arccos x)}{\sqrt{1-x^2}},\quad \beta(x,t)=\frac{\sin(t\arccos x)}{\sqrt{1-x^2}} \tag{2.4.16}$$

因为：

$$\sin(\arccos x)=\sqrt{1-x^2} \tag{2.4.17}$$

图 2.4.2 给出了 $\beta(x,t)$的图。注意当 x=1 的时候，图变得平坦，而当 x= −1 的时候，图弯曲得

非常厉害。这是近似算法中潜在问题的原因所在。

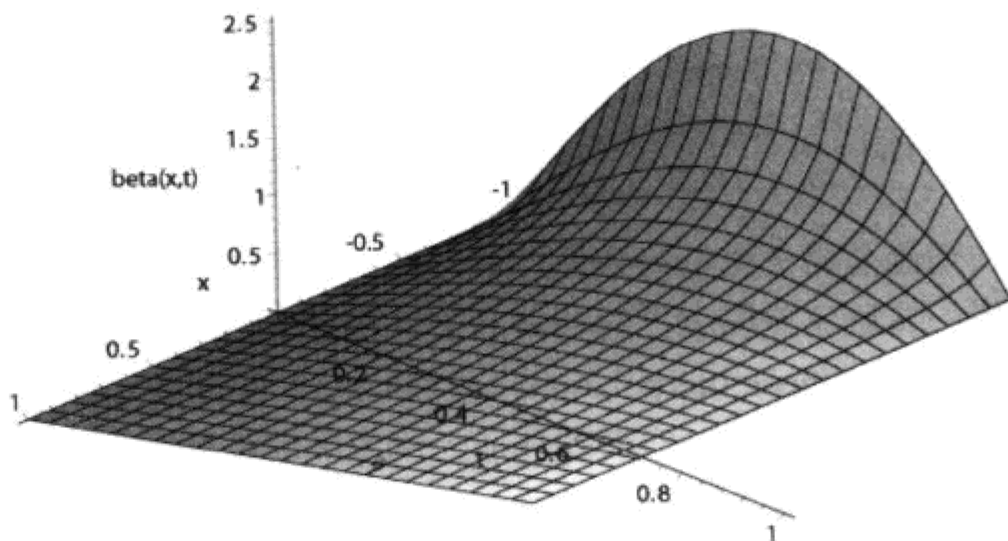


图 2.4.2 函数$\beta(x,t)$的 3D 绘图

2.4.3 近似算法

现在可以讨论近似算法了。为了说明可以使用的方法，我们将使用数学软件 Maple 的处理例子。

我们讨论过一些近似算法，并总结了它们的好处和缺陷。关于数值近似算法的一个金科玉律是：没有哪个算法是最好的，必须根据当前的环境来决定使用哪一个近似算法。

在面对电脑游戏这个环境的时候，我们需要问自己几个问题：

- 能节省多少 CPU 周期？
- 需要多大的精度？
- 可否尽量少用 `sqrt(x)` 和 `exp(x)` 之类的函数？
- 是否可以使用 SIMD 指令？如 VU 宏模式（VU macro mode），成对的浮点数（paired float），SSE 和 3DNow？
- 是否准备一次混合两个以上的四元数？
- 两帧之间角度的最大值是多少？

这里会提供一些方法，大家可根据实际需要来决定使用哪一种方法。

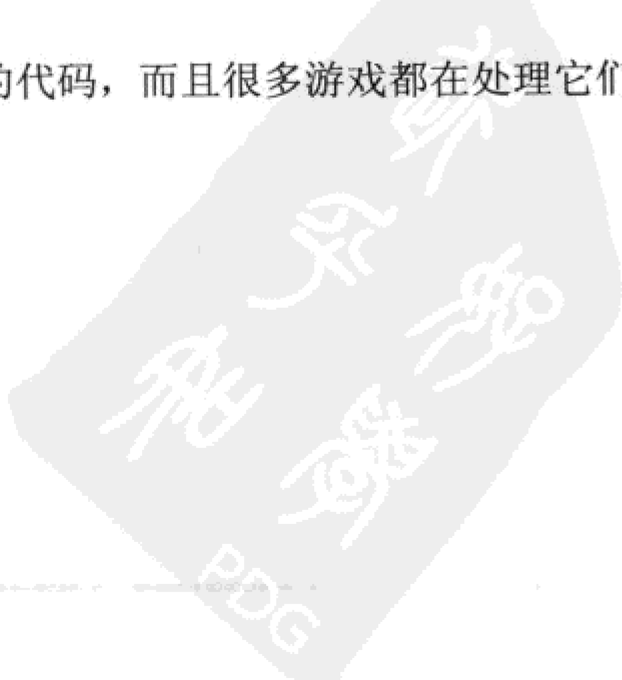
1. 直接方法

首先从直接近似的方法开始。考查 beta 函数的公式，并对它的各部分进行近似。如果需要非常高的精度，这是最好的方法，但是其开销相应也比较大，尤其是面对一大堆 slerp 的时候。

几乎所有的游戏引擎都包括下面的代码，而且很多游戏都在处理它们的时候花费了大量的精力。

程序清单 2.4.1 Slerp 参考类

```
class SlerpReference
{
```



```
    public:
        SlerpReference( const Quat &a, const Quat &b ) : mA( a ), mB( b )
        {
            float adotb = a.X * b.X + a.Y * b.Y + a.Z * b.Z + a.W * b.W;
            adotb = Min( adotb, 0.99999f );
            mTheta = acosf( adotb );
            mRecipSqrt = RecipSqrt( 1 - adotb * adotb );
        }

        Quat Interpolate( float t ) const
        {
            float alpha = sinf( ( 1 - t ) * mTheta ) * mRecipSqrt;
            float beta = sinf( t * mTheta ) * mRecipSqrt;
            return Quat( alpha * mA.X + beta * mB.X, alpha * mA.Y + beta * mB.Y
                         , alpha * mA.Z + beta * mB.Z, alpha * mA.W + beta * mB.W );
        }
    private:
        float mTheta;
        float mRecipSqrt;
        const Quat &mA;
        const Quat &mB;
    };
```

这里使用了三角函数来创建 slerp 参考类。该类有一个构造函数和一个用来计算各个经过插值的四元数的函数。我们努力避免了使用会引起过长的流水线延迟的分支，其方法是使用 Min 函数以避免溢出。注意，尽管当 adotb=1 的时候，结果可能是可靠的，但当四元数互相远离的时候，结果将无可预知。

要得出上面算法的近似算法，需要首先得到 sin(x) 和 arccos(x) 的近似算法。

有了传统近似多项式工具(traditional polynomial approximation tools)，对 sin(x) 部分的近似将比较简单。在 Maple 中，有一个名为 numapprox 的工具包，其内含有求近似多项式的工具，它们可以在特定的值域上将任意函数转换成多项式。

泰勒级数可以将某函数 $f(x)$展开为一个关于 x 的多项式，使之在某个点非常精确。Maple 有一个泰勒级数命令，可以轻松得到下面的结果：

$$taylorseries = x - \frac{x^3}{3!} + \frac{x^5}{5!} + O\left(x^7\right) \quad (2.4.18)$$

可惜，泰勒级数近似并不是非常有用，因为其仅在一点处是精确的，但是它易于计算，并有益于提供一个函数的常规形式。多项式可在 n 个或者更多不同的地方实现精确，其中 n 是多项式的次数，或者最大指数。这可以有效减少误差。

在 numapprox 工具包中，Maple 有一个 minimax 指令，它会选择最好的地方，使多项式的“最大误差”最少，以尽量满足多项式精确度的需要，这也是其得名的原因。

$$minimaxSeries = \text{minimax}(\sin(x), x=-\pi...\pi, 3) = \left(0.824535+\left(-0.08692x\right)x\right)x \quad (2.4.19)$$

在图 2.4.3 中，有泰勒级数、*minimax* 级数和 sin(x) 三条曲线。当$|x|>1$ 的时候，泰勒级数趋近于无穷大。图 2.4.4 中显示的是一种误差的情况，泰勒级数在原点很精确，而 minimax 多项式仅在五个互相分离的区域精确，随着与精确点的距离的加大，误差也越来越明显。

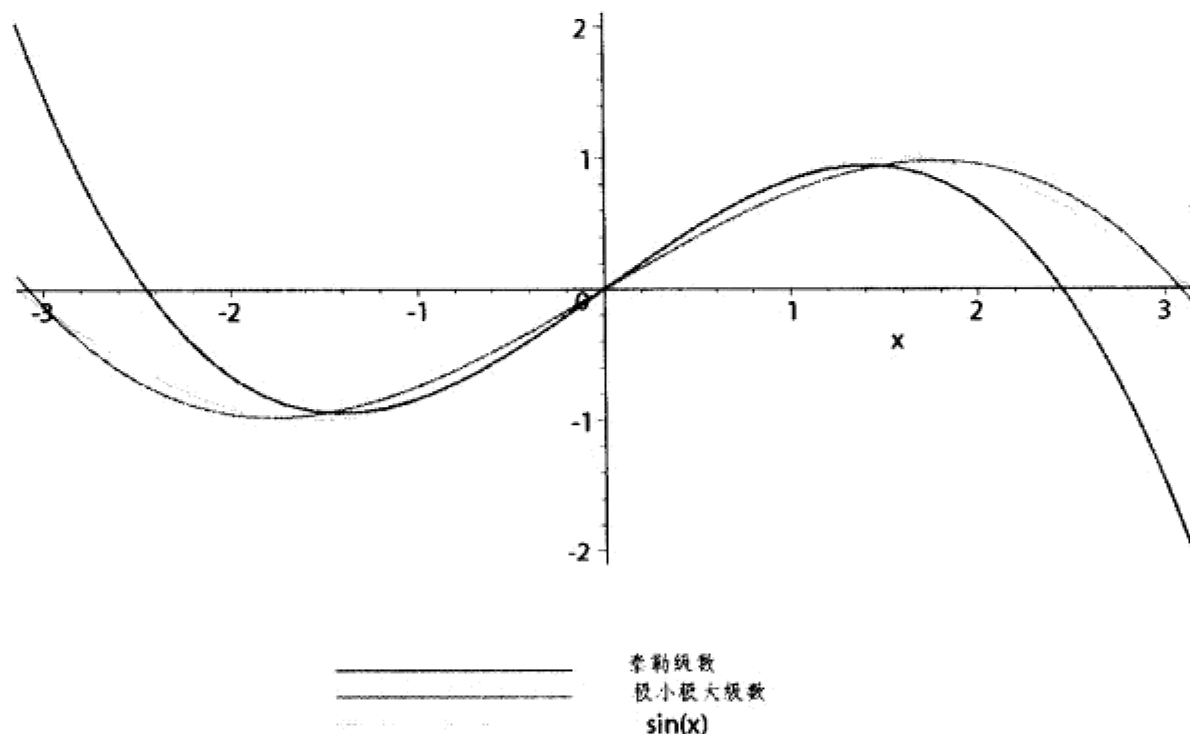


图 2.4.3 泰勒级数、极小极大级数和正弦函数的差别图示

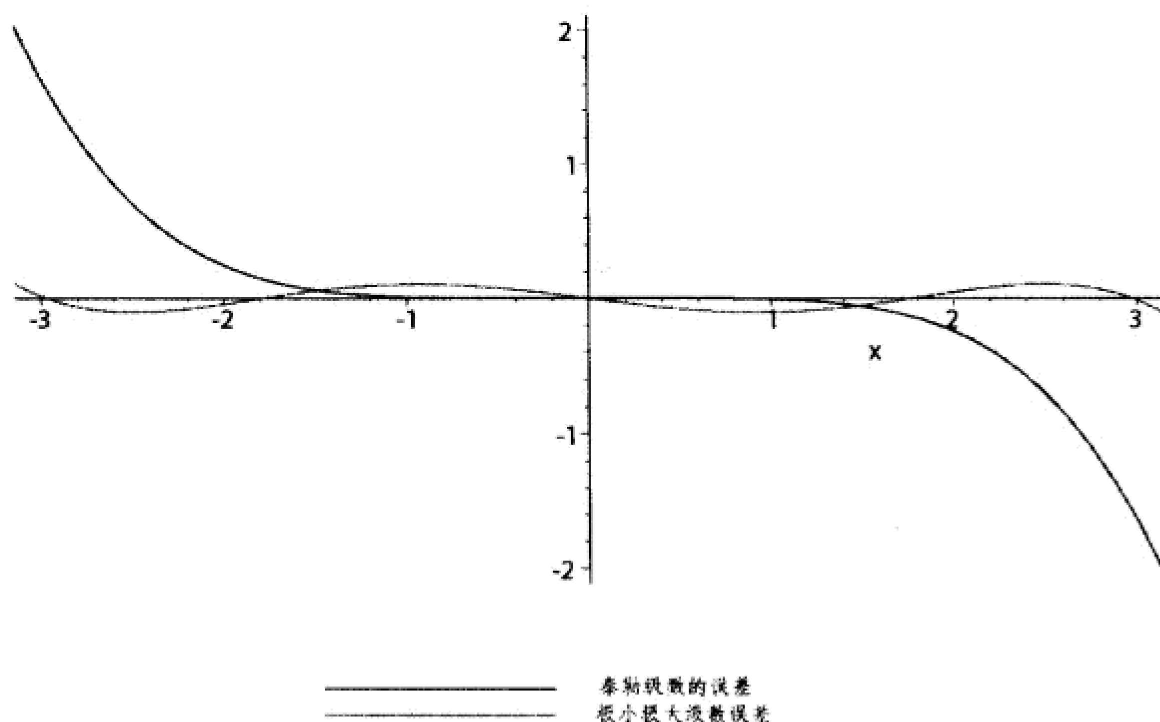


图 2.4.4 泰勒级数和极小极大级数的误差图示

这就是 C++内置函数的实现。下面是对 cos(x)在值域 $x \in [-\pi/2, \pi/2]$上的近似，这个近似可用于对 sin(x)在值域 $x \in [0, \pi]$上进行近似。这种情况下，cos(x)比 sin(x)更容易近似，因为它仅使用了多项式的 x^2 项：

$$\cos(x) = \sin(x + \pi/2) = 1 + \left(-0.4999991 + \left(0.416636 + \left(-0.0138537 + 0.000231540x^2\right)x^2\right)x^2\right)x^2 \quad (2.4.20)$$

图 2.4.5 显示的是上述近似的误差图：

但 `arccos(x)` 的算法就有些不同了。图 2.4.6 给出了八个项的误差情况，情形非常可怕！其原因必须要从 `arccos(x)` 函数自身说起，如图 2.4.7 所示。在 $x = 1$ 和 $x = -1$ 处有两个奇点，分别形如 $\sqrt{1-x}$ 和 $\sqrt{1+x}$。多项式很难近似这样的函数。在 Maple 中，可以使用“series”命令发现类似的函数。

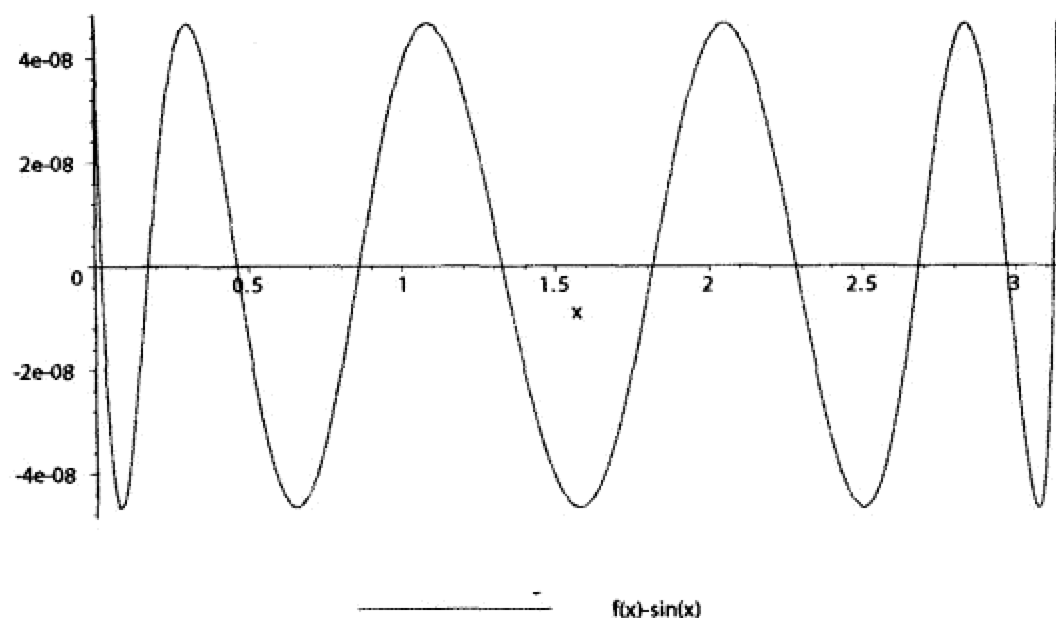


图 2.4.5　展示了 cos(x)平移到 slerp 函数的定义域后的误差

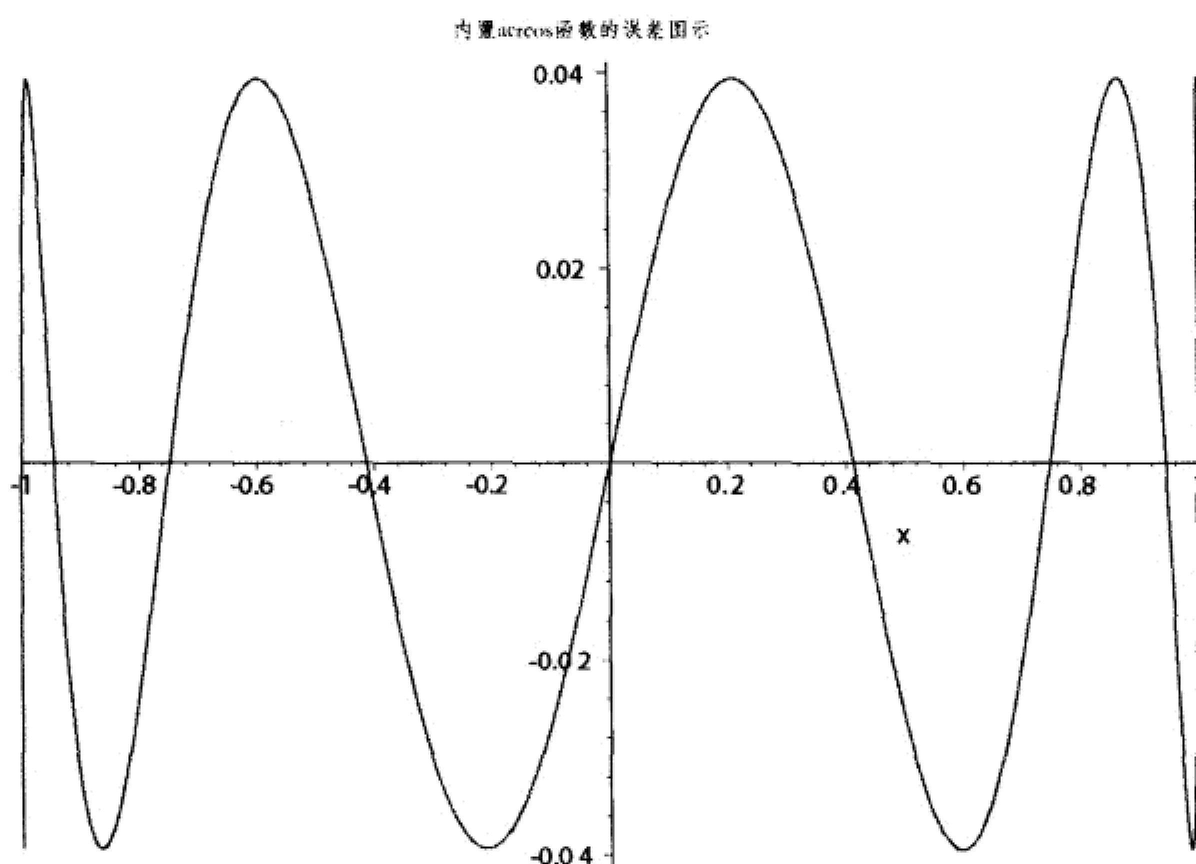


图 2.4.6　Maple 对 arccos 函数进行极小极大运算的结果

在非线性项中进行近似会更好一些：

$$\begin{aligned}&\arccos(x) \approx \\ &\sqrt{2.218480716 - 2.441884385x + 0.2234036692x^2} - \\ &\sqrt{2.218480716 + 2.441884385x + 0.2234036692x^2} + \\ &\pi/2 + 0.6391287330x\end{aligned} \tag{2.4.21}$$

图 2.4.8 展现出这种近似的误差曲线。注意这些函数仅能在需要的定义域内工作，如果超出了这个范围，就必须分开对待。随书光盘中包含了一些 Maple 的例子。

平方根的倒数在大多数现代处理器上都可以被除法单元支持。但考虑到反应时间，最好先计算它们，并在之后的计算中直接使用结果。16 位 SIMD 指令集也经常支持近似的平方根的倒数。

旋转中的比例误差可以通过具有相同次序的 alpha 和 beta 函数的误差来计算。为了完成

这样的计算，应该假设函数有一个误差 e，并使用 Maple 的“series”命令。减少级数项，直到序 $O(e^k)$成为 e 中的最后一个保留项。

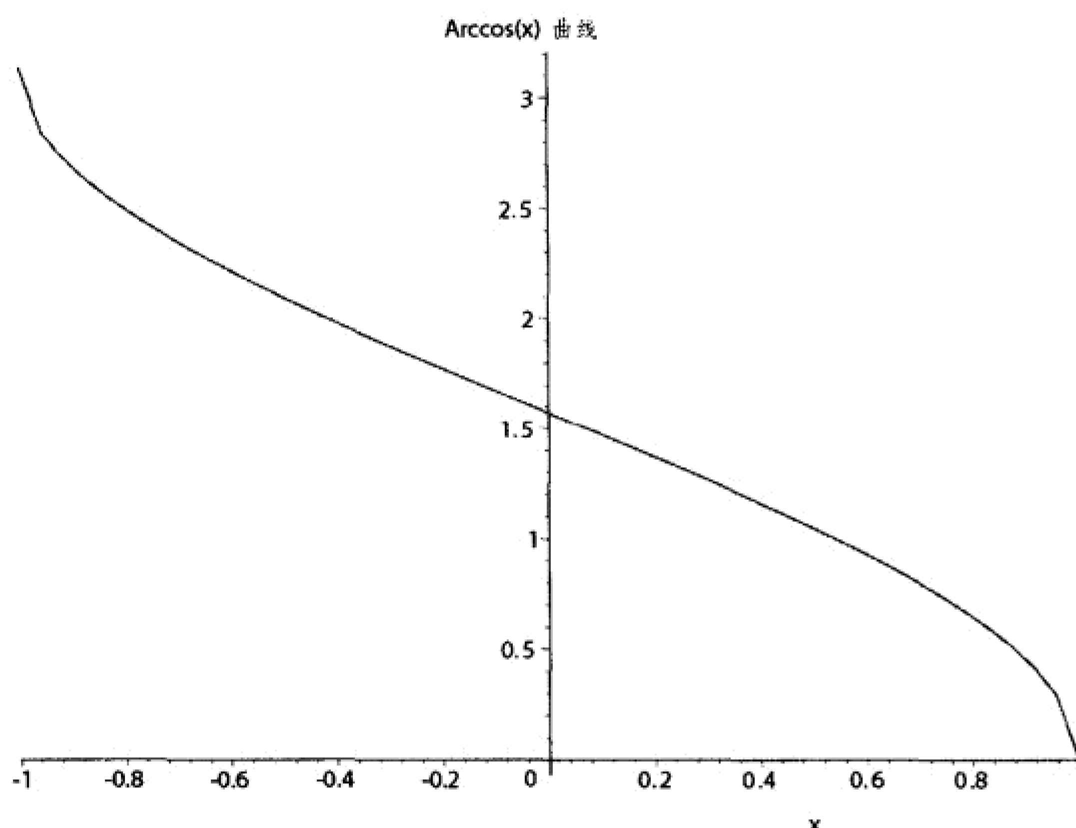


图 2.4.7 arccos(x)在 $x=\pm1$ 处显示出异常

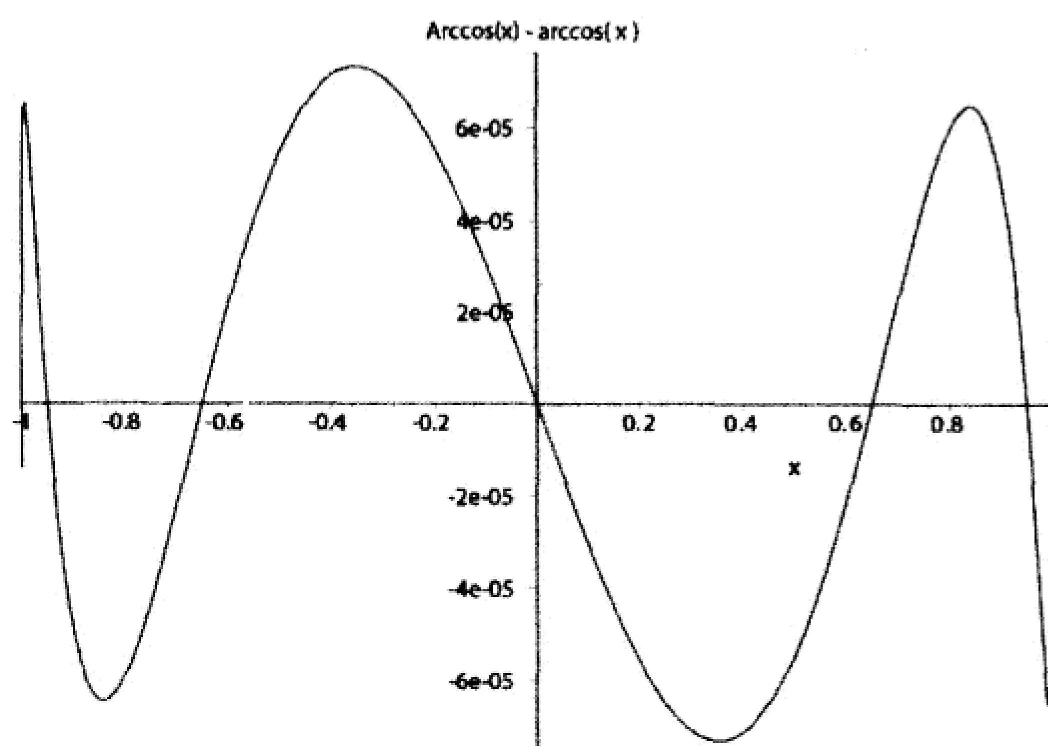


图 2.4.8 arccos(x)的更好的近似

图 2.4.9 是 beta 函数四元数在一些角度取值下的误差级数图。

$$\frac{Sin(t\,Arccos(x))}{\sqrt{1-x^2}}-\frac{\sin(t\arccos x)}{\sqrt{1-x^2}} \tag{2.4.22}$$

其中 *Sin(x)*和 *Arccos(x)*是近似函数。

如程序清单 2.4.2 所示，使用 SIMD 架构能同时进行两个正弦函数的计算。

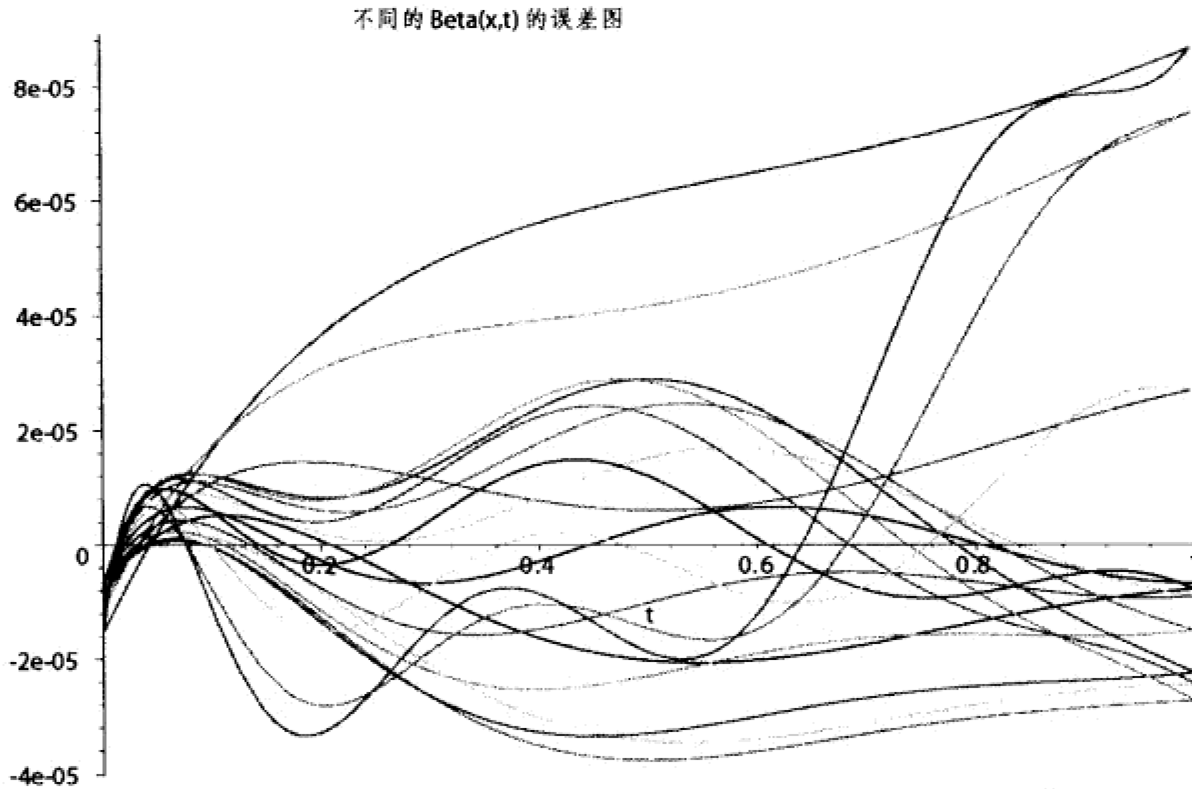


图 2.4.9 函数$\beta(x,t)$在不同的 x 处的误差图

程序清单 2.4.2 直接 Slerp 方法的类

```
class SlerpDirect
{
public:
    SlerpDirect( const Quat &a, const Quat &b ) : mA( a ), mB( b )
    {
        float adotb = a.X * b.X + a.Y * b.Y + a.Z * b.Z + a.W * b.W;
        adotb = Min( adotb, 0.99995f );
        float even = 2.218480716f + 0.2234036692f * adotb * adotb;
        float odd = 2.441884385f * adotb;
        mTheta = Sqrt( even - odd ) - Sqrt( even + odd ) + 1.570796327f + 0.6391287330f * adotb;
        mRecipSqrt = RecipSqrt( 1 - adotb * adotb );
    }

    Quat Interpolate( float t ) const
    {
        float A = ( 1 - t ) * mTheta - 1.570796327f; A = A * A;
        float B = t * mTheta - 1.570796327f; B = B * B;
        float sinA = .9999999535f+(-.4999990537f+(.4166358517e-1f
            +(-.1385370794e-2f+.2315401401e-4f*A)*A)*A)*A;
        float sinB = .9999999535f+(-.4999990537f+(.4166358517e-1f
            +(-.1385370794e-2f+.2315401401e-4f*B)*B)*B)*B;
        float alpha = sinA * mRecipSqrt;
        float beta = sinB * mRecipSqrt;
        return Quat( alpha * mA.X + beta * mB.X, alpha * mA.Y + beta * mB.Y, alpha
* mA.Z + beta * mB.Z, alpha * mA.W + beta * mB.W );
    }
private:
    float mTheta;
    float mRecipSqrt;
    const Quat &mA;
```

```
        const Quat &mB;
    };
```

这个类展示了对传统 slerp 算法的构成函数进行近似后得到的结果。

2. 矩阵近似

因为$\beta(x,t)$只是一个关于两个变量的简单函数，所以在理论上我们可以把整个结果表现为关于 x 和 t 的二维多项式。这就是

$$\beta(x,t) = \mathbf{XMT} \tag{2.4.23}$$

其中 **M** 是一个 $N_x \times N_t$ 矩阵，且

$$\mathbf{X} = \begin{pmatrix} 1 & x & \cdots & x^{N_x-1} \end{pmatrix}$$
$$\mathbf{T} = \begin{pmatrix} t & t^3 & \cdots & t^{N_t-1} \end{pmatrix} \tag{2.4.24}$$

在分析 arccos(x)的过程中采用相同的处理方法，也就是说使用 Maple 的 series 命令，这就变成了一个对整个多项式进行近似的问题，因为

$$\lim_{x\to-1} \beta(x,t) = \infty \tag{2.4.25}$$

需要利用函数的这个特点，以使我们能用更少的条件对其进行近似。

如果计算

$$g(x,t) = \beta(x,t)(1+x) \tag{2.4.26}$$

会得到一个在 $x\in[-1,1]$内表现更好的函数，如图 2.4.10 所示。然后在两边同时乘以 $1/(1+x)$，就可以得到$\beta(x,t)$。

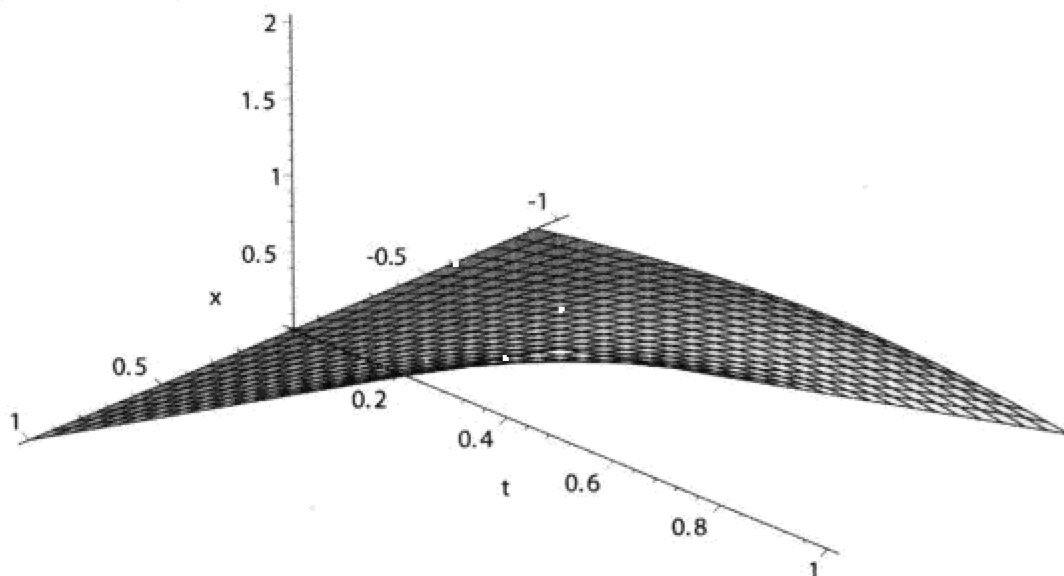


图 2.4.10 $g(x,t)$，对$\beta(x,t)$近似的一个更好选择

可惜的是，Maple 没有 minimax 命令的 2D 版本，此外统一的误差分布只有在特定的条件下才有用。

可以建立一个 2D 版的契比雪夫（Chebyshev）逼近算法，它构造了一个具有以下形式的函数：

$$\sum_{j=0}^{N_t}\sum_{i=0}^{N_x} c_{i,j} T(i,x) T(j,t) \tag{2.4.27}$$

其中 $T(i,x)$和 $T(j,t)$是第一类契比雪夫多项式，以下列形式给出：

$$\begin{aligned} T(0,x) &= 1 \\ T(1,x) &= x \\ T(2,x) &= 2x^2 - 1 \\ T(3,x) &= 4x^3 - 3x \end{aligned} \tag{2.4.28}$$

建立了一个上述形式的函数后，通过提取项 x 和 t 就能简单地将它转化为一个规则多项式。随书附送的光盘中已经包含了一个具有上述功能的 maple 程序。

将上式代入 $g(x,t)$，得到：

$$\mathbf{M} = \begin{pmatrix} 1.570994357 & -0.6461396421 & 0.07949824672 & -0.004354110679 \\ 0.5642929825 & 0.5945659091 & -0.1730440015 & 0.01418982936 \\ -0.1783657609 & 0.08610292588 & 0.1079287872 & -0.01567243477 \\ 0.04319948653 & -0.03465102568 & -0.01439451411 & 0.005849053560 \end{pmatrix} \tag{2.4.29}$$

这可以在 $0<\mathbf{a}\cdot\mathbf{b}<1$ 中得到最好的精度。显然，还有更多的项可以使用，每一行添加两条 SIMD 指令，就可以使 x 的展开式更精确。

该算法的一个非常有用的功能，如果计算出向量

$$\frac{1}{1+x}\mathbf{XM} \tag{2.4.30}$$

乘数 $\alpha(t)$ 和 $\beta(t)$ 的值就能通过简单的代数计算得到。

程序清单 2.4.3 给出了一种实现方法。

程序清单 2.4.3 矩阵 Slerp 方法的类

```
class SlerpMatrix
{
public:
    SlerpMatrix( const Quat &a, const Quat &b ) : mA( a ), mB( b )
    {
        float adotb = a.X * b.X + a.Y * b.Y + a.Z * b.Z + a.W * b.W;
        mRecipOnePlusAdotB = Recip( 1 + adotb );
        mC1 = 1.570994357f+(.5642929859f+( -.1783657717f
            +.4319949352e-1f*adotb)*adotb)*adotb;
        mC3 = -.6461396382f+(.5945657936f+(.8610323953e-1f
            -.3465122928e-1f*adotb)*adotb)*adotb;
        mC5 = .7949823521e-1f+( -.1730436931f+(.1079279599f
            -.1439397801e-1f*adotb)*adotb)*adotb;
        mC7 = -.4354102836e-2f+(.1418962736e-1f+( -.1567189691e-1f
            +.5848706227e-2f*adotb)*adotb)*adotb;
    }

    Quat Interpolate( float t ) const
    {
        float T = 1 - t, t2 = t * t, T2 = T * T;
        float alpha = (mC1+(mC3+(mC5+mC7*T2)*T2)*T2)*T * mRecipOnePlusAdotB;
```

```
        float beta = (mC1+(mC3+(mC5+mC7*t2)*t2)*t2)*t * mRecipOnePlusAdotB;
        return Quat( alpha * mA.X + beta * mB.X, alpha * mA.Y + beta * mB.Y,
            alpha * mA.Z + beta * mB.Z, alpha * mA.W + beta * mB.W );
    }
private:
    float mRecipOnePlusAdotB;
    float mC1, mC3, mC5, mC7;
    const Quat &mA;
    const Quat &mB;
};
```

这里预先计算了多项式系数，在后面的计算中，仅需要计算一系列的插值即可。实际应用中，可能会使用 SIMD 指令重写代码。

3. 重新标准化

一个快速的近似算法（其精确性并不重要）是对四元数进行简单的线性插值，然后将结果重新规格化。这样产生的结果对通常的四位到八位计算系统，其精度已经足够。如果四元数间的角度非常小，结果将非常精确。

这个方法经常在 vertex shader 中使用，和传统的矩阵混合相比，它能更高效率地进行多个四元数间的混合。

但是，除了初始化时间，插值法要比之前展示的矩阵 slerp 算法慢，而且精度低。

程序清单 2.4.4 给出了一种实现方法。

程序清单 2.4.4 简单的插值和重新标准化

```
class SlerpSimpleRenormal
{
public:
    SlerpSimpleRenormal( const Quat &a, const Quat &b ) : mA( a ), mB( b )
    {
    };

    Quat Interpolate( float t ) const
    {
        float alpha = 1 - t;
        float beta = t;
        Quat result( alpha * mA.X + beta * mB.X, alpha * mA.Y + beta * mB.Y,
                     alpha * mA.Z + beta * mB.Z, alpha * mA.W + beta * mB.W );
        float recip = RecipSqrt( result.X * result.X + result.Y * result.Y
            + result.Z * result.Z + result.W * result.W );
        return Quat( result.X * recip, result.Y * recip,
                     result.Z * recip, result.W * recip );
    }
private:
    const Quat &mA;
    const Quat &mB;
};
```

简单线性算法可通过近似

$$\beta(x,t)=\frac{\sin(t\arccos x)}{\sin(t\arccos x)+\sin((1-t)\arccos x)},\quad \alpha(x,t)=1-\beta(x,t) \tag{2.4.31}$$

然后进行重新标准化，可以改进简单线性算法。或者，也可以在线性算法和重新标准化操作之前使用角平分算法。

重新标准化也能显著增加直接法和矩阵算法的精度，但是需要耗费额外的时间。

程序清单 2.4.5 给出了改进后的重新标准化算法的一种实现方法。

程序清单 2.4.5　改进后的重新标准化方法（相对来说更麻烦）

```
class SlerpRenormal
{
public:
    SlerpRenormal( const Quat &a, const Quat &b ) : mA( a ), mB( b )
    {
        float adotb = a.X * b.X + a.Y * b.Y + a.Z * b.Z + a.W * b.W;
        adotb = Min( adotb, 0.995f );
        float even = 2.218480716f + .2234036692f * adotb * adotb;
        float odd = 2.441884385f * adotb;
        mTheta = Sqrt( even - odd ) - Sqrt( even + odd )
            + 1.570796327f + .6391287330f * adotb;
    }

    Quat Interpolate( float t ) const
    {
        float T = 1 - t, t2 = t * t, T2 = T * T;
        float A = ( 1 - t ) * mTheta;
        float B = t * mTheta;
        float sinA = -.67044e-5f + ( 1.000271283f + ( -.17990919e-2f
            + ( -.1621365372f + ( -.556099983e-2f + ( .1198086481e-1f
            - .1271209213e-2f * A ) * A ) * A ) * A ) * A ) * A;
        float sinB = -.67044e-5f + ( 1.000271283f + ( -.17990919e-2f
            + ( -.1621365372f + ( -.556099983e-2f + ( .1198086481e-1f
            - .1271209213e-2f * B ) * B ) * B ) * B ) * B ) * B;
        float recipAB = Recip( sinA + sinB );
        float alpha = sinA * recipAB;
        float beta = sinB * recipAB;

        // renormalise
        Quat result( alpha * mA.X + beta * mB.X, alpha * mA.Y + beta * mB.Y,
            alpha * mA.Z + beta * mB.Z, alpha * mA.W + beta * mB.W );
        float recip = RecipSqrt( result.X * result.X + result.Y * result.Y
            + result.Z * result.Z + result.W * result.W );
        return Quat( result.X * recip, result.Y * recip,
            result.Z * recip, result.W * recip );
    }
private:
    float mTheta;
    const Quat &mA;
    const Quat &mB;
};
```

4．角平分方法

通过下式可以平分两个四元数之间的夹角。

$$\beta\left(x,\tfrac{1}{2}\right)=\alpha\left(x,\tfrac{1}{2}\right)=\frac{\sin\left(\tfrac{1}{2}\arccos(x)\right)}{\sqrt{1-x^2}}=\frac{1}{\sqrt{2+2x}} \tag{2.4.32}$$

它能精确地得到平分角、四分之一角、八分之一角、十六分之一角，等等，并能自由组合任意数量的这些角度。随书附送的光盘中已经包含了一个 Maple worksheet 文件，它通过平方根项来计算 β 函数的平分算法。

$$\begin{aligned}
&\beta(x,0)=0,\\
&\beta(x,1/4)=\frac{1+\sqrt{2+2x}}{\sqrt{2+\sqrt{2+2x}}\sqrt{2+2x}},\\
&\beta(x,1/2)=\frac{1}{\sqrt{2+2x}},\\
&\beta(x,3/4)=\frac{1}{\sqrt{2+\sqrt{2+2x}}\sqrt{2+2x}},\\
&\beta(x,1)=1
\end{aligned} \tag{2.4.33}$$

2.4.4 算法之间的比较

我们测试了上面几个类的精度及计算速度。测试数据是三组分别以大、中、小三种旋转角度进行旋转的四元数。

我们期望小角度的旋转最为精确，因为四元数之间的距离最小。虽然大角度旋转在动画数据中使用最少，但是算法同样必须能正确处理。表 2.4.1 列举了在不同输入数据下各个算法的精确度。

表 2.4.1 相同 bit 下的近似精度

数据集	最大角度		中等角度		小角度	
	最坏	平均	最坏	平均	最坏	平均
SlerpDirect	11	13	11	11	11	11
SlerpRenormal	16	18	17	19	19	19
SlerpMatrix	13	16	14	15	15	15
SlerpSimpleRenormal	4	8	11	15	19	19

表 2.4.2 列出了各个算法在 slerp 50 000 个四元数时花费的时间。测试由一次初始化和十次使用大范围随机数据的插值运算组成。

表 2.4.2 slerp 的时间

SlerpReference	22 906
SlerpDirect	13 077
SlerpRenormal	24 115
SlerpMatrix	7 829
SlerpSimpleRenormal	12 553

这表明近似函数比非近似函数要快得多，虽然得到了速度上的提升，但是我们可能需要使用汇编指令来编码。

矩阵算法具有明显的优势，甚至超过了简单线性算法和重新标准化算法，这可能是因为 sqrt 在 fpu 上的执行速度比较慢。

这在 PS2 的 VU0 协处理器上性能尤其出色，因为可使用它的宏模式对一组蒙皮四元数进行批处理。

2.4.5 Squad 相关的计算

我们也研究了加速 squad（spherical quadrangle，经常被错认为是贝塞尔 slerp）近似算法的方法。

通过使用近似 arccos 函数、近似 sin 函数和其他类似的近似函数，可以简化使用 log 和 e 指数的四元数的传统算法。

ON THE CD

程序清单 2.4.6 实现了一个能简单化用 SIMD 指令编码的包含了良性分量的函数。程序清单 2.4.6 中的逼近函数 ArccosFast 和 SinFast 已经包含在随书附送的光盘中。

程序清单 2.4.6 良好的 squad 导数生成器

```
// Squad 导数计算，手工优化
// 这样非常快速并能结出出色的结果 (16bit)
// 试着删掉在? 操作符中隐含的分支
Quat DerivativeCompact( const Quat &a, const Quat &b, const Quat &c )

{
    Quat bconj = Conj( b );
    Quat arel = Mul( a, bconj );
    Quat crel = Mul( c, bconj );
    float aScale = arel.W > 0.9999f ? -0.25f :
    -0.25f*ArccosFast( arel.W ) * RecipSqrt( 1 - arel.W * arel.W );
    float cScale = crel.W > 0.9999f ? -0.25f :
    -0.25f*ArccosFast( crel.W ) * RecipSqrt( 1 - crel.W * crel.W );
    float logx = aScale * arel.X + cScale * crel.X;
    float logy = aScale * arel.Y + cScale * crel.Y;
    float logz = aScale * arel.Z + cScale * crel.Z;
    float length = Sqrt ( logx * logx + logy * logy + logz * logz );
    float sinLength = SinFast( length );
    float cosLength = Sqrt( 1 - sinLength * sinLength );
    float xyzScale = length < 1e-5f ? 1 : sinLength / length;
    return Mul( b, Quat(xyzScale * logx, xyzScale * logy, xyzScale * logz, cosLength
) );
}

//  这个函数可以如下方式使用

Quat d1 = DerivativeCompact( q0, q1, q2 );
Quat d2 = DerivativeCompact( q3, q2, q1 );
```

```
Quat q12 = SlerpMatrix( q1, q2 ).Interpolate( fract );
Quat d12 = SlerpMatrix( d1, d2 ).Interpolate( fract );
Quat squad = SlerpMatrix( q12, d12 ).Interpolate( 2 * fract * ( 1 - fract ) );

// 其中 q0,...,q3 是关键点序列，fract 是小数部分的因子
```

2.4.6 延伸阅读

如果对四元数 slerp 操作的背景感兴趣，可以阅读 Ken Shoemake 的 SIGGRAPH 论文[Shoe85]。这篇论文不仅介绍了相关概念，而且介绍了四元数旋转。

$\beta(x,t)$几乎等同于第二类契比雪夫多项式，但与 2D 近似算法中使用的第一类契比雪夫多项式不一样。

$$\beta(x,t)=U_t(x) \tag{2.4.34}$$

这些是斯塔姆-刘维尔（Sturm-Louiville）微分等式：

$$\left(1-x^2\right)\frac{d^2}{dx^2}\beta(x,t)-3x\frac{d}{dx}\beta(x,t)+\left(t^2-1\right)\beta(x,t)=0 \tag{2.4.35}$$

通常，在函数是一个多项式的情况下，契比雪夫多项式的 t 是一个整数。这样，当参数 t 是一个实数的时候，将得到一个非多项式的结果。

阅读[Rich02]的第 11 章，可以获取进一步的信息。本书也有对 Maple、幂级数和近似方法的讲解。

在写本文的时候，我们使用了前向差分法来解这个关于 $\alpha(t)$和 $\beta(t)$的等式。尽管对于计算函数特定点的值来说，这不是一个好的方法,，但它对于计算一连串四元数插值是比较有帮助的。在实际工作中，也可使用多角度公式来列举连续 slerp 的结果。

2.4.7 总结

文中列举了一些在四元数插值过程中使用的近似方法，这些方法在基于角色的游戏（character-based games）中有广泛应用。这里提出了一种简单的基于矩阵的方法，它只需要一些乘法和加法就可以完成精确的四元数旋转插值。

本文还讨论了仅使用平方根的角平分（Angle subdivision）方法，在特殊的插值角度下，它非常简单。

最后还讨论了 squad 插值的方法，并提供了一个简单的函数，来生成在一系列帧中进行平滑插值所必需的附加四元数。

2.4.8 参考文献

[Hejl04] Hejl, Jim. “Hardware Skinning with Quaternions”*Games Programming Gems 4*, 5.12. Charles River Media,2004.

[Rich02] Richards, D. *Advanced Mathematical Methods with Maple*.Cambridge University Press,2002.

[Shoe85] Shoemake, Ken. “Animating Rotations with Quaternion Curves,” ACM SIGGRAPH, 1985.

2.5　极小极大数值近似

Christopher Tremblay
ti_chris@yahoo.com

游戏编程里有一个经常被忽视但却总会出现的领域，就是数值近似。从游戏的观点来看，近似（approximation，也称为逼近，本文统一使用“近似”这个概念）是非常重要的，因为整个游戏本身就是对需要表现的东西的近似。角色是用多边形来近似，物理是用给定的模型来近似。更加数学化，但是依然非常有趣的是，典型的近似就是把一个复杂的函数用更加简单、快速的函数来代替。例如，涉及正弦和余弦函数的计算是出了名的慢。事实上，我们可以提供一个更快、但不一定那么精确的函数来代替浮点处理单元（foating-point unit，简称 FPU）提供的版本，并从中获益。这样的函数通常用来定义世界中特定物体的运动或者位置，不完全精确通常不会明显消弱最终输出。在这样的情况下，找到近似原始函数的一个快速版本是非常有用的。

2.5.1　众所周知的优化

在学习微积分的时候，一个对普通函数的典型近似技术就是泰勒级数（Taylor series）。泰勒级数通过用给定次数的多项式对曲线进行近似。选择多项式函数是为了更加快速地计算并易于使用。多项式能快速地进行计算，是因其涉及的操作能够相对较快地在今天的 CPU 上完成（例如，和正弦/余弦或指数函数的计算进行比较的时候）。而且，它能写成 Horner 形式，这使得它们仅仅需要计算加法和乘法。多项式的 Horner 形式就是把 x 作为因子提到多项式的外面来。例如多项式：

$$ax + bx^2 + cx^3 + dx^4 + e \tag{2.5.1}$$

能很容易地改写成 Horner 形式：

$$e + x\left(a + x\left(b + x\left(c + xd\right)\right)\right) \tag{2.5.2}$$

泰勒级数对曲线的近似是通过拷贝曲线上给定点的属性。具体来说，就是拷贝曲线的位置、倾斜度、加速度等，或者更准确地说，这个多项式的导数（指的是高阶导数，最高阶到一个给定值）和曲线上给定点的导数相同。

PDG

2.5.2 什么是理想的近似

可惜的是，泰勒近似有很多问题。近似值的收敛非常得慢（因为很多项都需要像样的近似）。因此，曲线在给定点的误差将变得非常大。如果用泰勒级数近似计算出正弦值的误差是0.5，那么这个近似显然不是一个好的近似，因为曲线的值域是[-1 , 1]。注意图 2.5.1 和图 2.5.2 中正弦函数和它的四阶麦克劳林（MacLaurin）级数之间的不同（麦克劳林级数就是泰勒级数在 x=0 处的展开）。在这些图中，可以看到近似值在区间的末端效果不好，但是在起始处却非常好。

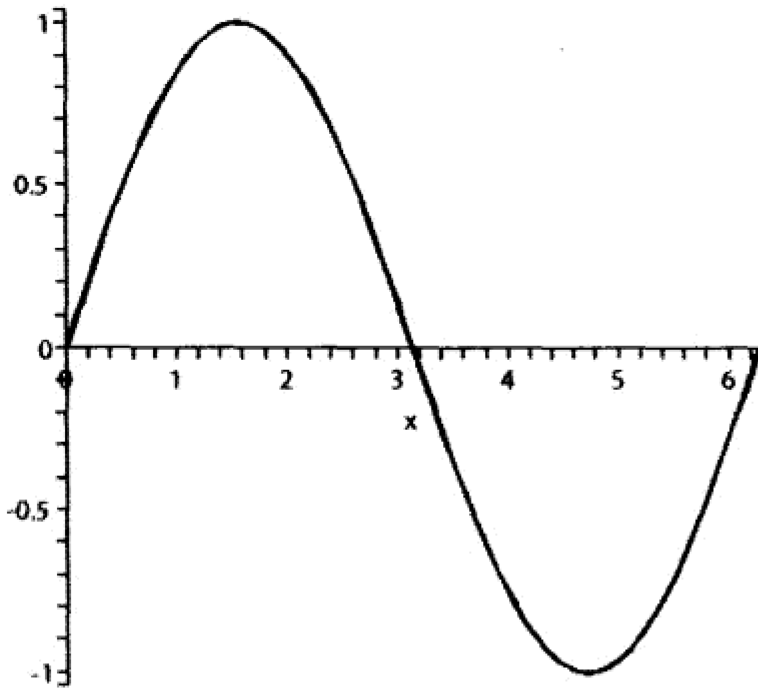


图 2.5.1 [0,2π]区间上的正弦函数

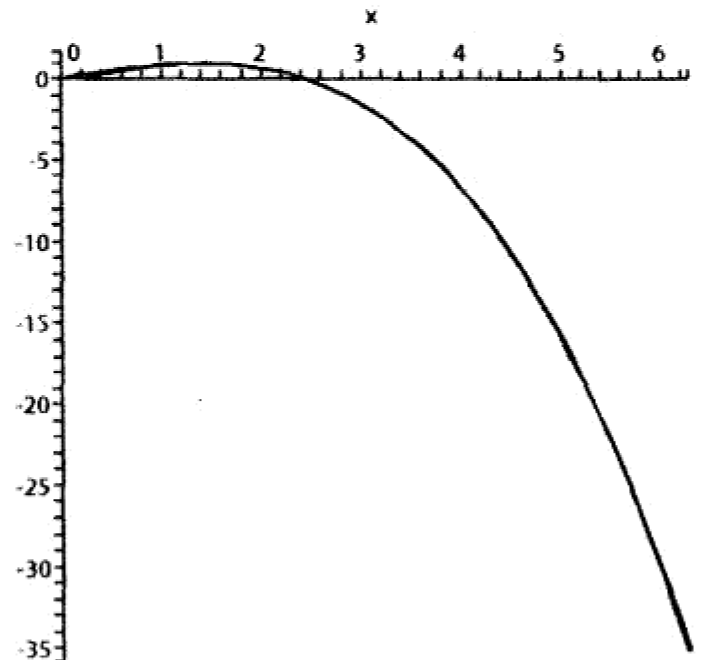


图 2.5.2 区间[0,2π]上的正弦函数的四阶麦克劳林级数近似

曲线在 $x = 0$ 处有极好的近似，但是在 $x = \pi$处的情况却非常糟糕。一个对曲线更好的逼近是选择 $x=\pi/2$，因为[0, π]区间上的平均误差会更小。因此在这一点上显然需要一个定义。究竟是什么使一个近似成为好的近似呢？其中一条就是，它必须和原函数非常相似，而且为了满足我们的目的，它还必须足够快。众所周知，当泰勒级数的多项式阶数足够高的时候，它将有非常好的近似程度。换句话说，当曲线上的更多阶的导数匹配的时候，对曲线的近似程度会更好，但是这就要求进行更多的计算。当近似计算比实际函数更慢的时候，就有问题了，也就是说这个近似并不能被认为是“好”的近似。

现在的问题是，什么是“非常相似”的确切定义和含义。它是一个非常含糊不清的定义，需要在数学术语上给出更加确定的定义。首先要定义一个需要对函数进行近似的区间。如果限制近似的区间，将能得到更好的近似。根据这个想法，就可以说函数的最佳近似就是在区间[a,b]上最大误差最小的那个函数。此外，如果在多项式次数相同的情况下，一个函数的精度比另外一个函数高，那么可以说前者的近似程度更好。例如，正弦函数的四阶麦克劳林公式如下：

$$\sin x \approx x - \frac{x^3}{6} \qquad (2.5.3)$$

但以 $x = \pi$为中心的泰勒级数如下：

$$\sin x \approx \pi - \frac{\pi^3}{6} + \left(\frac{\pi^2}{2} - 1\right)x - \frac{\pi}{2}x^2 + \frac{1}{6}x^3$$
$$\approx \pi - \frac{\pi^3}{6} + x\left(\left(\frac{\pi^2}{2} - 1\right) - x\left(\frac{\pi}{2} + \frac{1}{6}x\right)\right) \quad (2.5.4)$$

后面的 Horner 形式给出的项比前面更多，所以虽然第 2 种情况下的误差比第 1 种情况要小一些，但是仍然有可比性。第一种近似比较好的原因在于它的计算复杂度更少。如果给第 1 种近似加上更多的项，比如和第 2 种近似的项一样多，那么第 1 种近似将能提供比第 2 种近似更小的误差，所以会认为第 1 种近似比后一种更好。

$$\sin x \approx x\left(1 - \frac{x^2}{6}\right) \quad (2.5.5)$$

再推广一下，对任意次多项式，x^2 都可以作为因子被分解出来，对于给定的精度，它能明显降低计算的复杂性，因为正弦函数的麦克劳林级数中所有的偶数次项的系数全为 0。因此，重要的不只是看两个函数有多接近，而且还要看计算的复杂性。这就使我们可以更容易地得出结论：对函数的最佳近似就是在给定的次数（多项式次数）下能在区间[a,b]上让误差的最大值最小。

2.5.3 极小极大近似的介绍

定义了“最佳近似”的属性之后，就可以定义极小极大近似了。极小极大近似就是对给定多项式次数下的最佳近似。这和计算复杂性（也就是说，需要进行多少计算工作）没有联系，所以这和最佳近似的定义有些不同。对于我们的目的，用一句话来说就是：极小极大近似就是一个 n 次近似多项式对给定函数有最小的最大逼近误差。在构造过程中，极小极大近似对于给定的多项式次数来说是惟一的，它在对方程进行重新排列后也是惟一的。因为它和计算复杂性没有联系，所以应该注意到其他一些函数在某些情况下可能得到“更好的近似”。换句话说，对于相同的多项式次数，和最佳近似相比，极小极大近似仍然是比较好或者是等价的。但是因为计算最佳近似所需的 CPU 的工作量使它比极小极大近似来的更快，所以极小极大近似并不是最佳近似。例如，当总共只有 13 比特的精度的时候，对于正弦函数来说这是正确的。在这种情况下，泰勒级数就是最佳近似，因为 Horner 形式允许把方程改写成更加有效的方式。

为了让符号更加简洁，可以把一个 n 次多项式 $f(x) = c_0 + c_1x + c_2x^2 + \ldots c_nx^n$ 改写为点积形式。

$$f(x) = \begin{bmatrix} c_0 & c_1 & c_2 & \cdots & c_n \end{bmatrix} \begin{bmatrix} 1 \\ x \\ x^2 \\ \vdots \\ x^n \end{bmatrix} \quad (2.5.6)$$

在这里定义行向量为 $\mathbf{c}$，列向量为 $\mathbf{x}$。我们的主要兴趣在于找到向量 $\mathbf{c}$ 中的系数 c_i，极小极大多项式的关键是 Chebyshev Equioscillation 理论。设 f 为定义在区间$[a,b]\in\mathbf{R}$ 上的连续函数。

当且仅当存在 $n+2$ 个点 $a \leqslant x_0 \leqslant x_1 \leqslant \ldots \leqslant x_{n+1} \leqslant b$ 满足 $f(x_j)-p(x_j)=-(-1)^j E$ 时，n 次多项式 $p(x)$是极小极大多项式，其中 $j=0,1,\ldots,n+1$，而 $E=\pm(f-p)$。总的来说，这表明误差函数事实上有（$n+2$）个极值点。作为定理的一个结论，在近似多项式的次数为 n 的时候，两个函数总共有（$n+1$）个点是相同的。当然，例外的情况是误差 E 为 0，此时多项式和原始曲线在定义区间内能很完美地重合在一起。

当给定的极值比前面的理论提到的要多时，可以很容易地推导出近似曲线中第 1 个和最后一个点是误差最大的点。而且，带符号的误差必须改变符号。给定了这么多信息，就可以用很简单的方法来解决这个很简单的问题。例如，需要用一条直线来近似抛物线。如果能简单地使用前面提到的数学定理和辅助定理的话，就可以解决这个问题。我们需要找到直线 $f(t)=a+bt$ 的系数向量 $\mathbf{c}=[a, b]$，它满足抛物线 $g(t)=3t^2$ 在区间[0,1]上和直线的误差 E 在三个点处的绝对值相等（其中两个点是 $t=0$ 和 $t=1$）。这可以用下面的方程来归纳：

$$\begin{gathered} f(0)-g(0)=E \\ f(x)-g(x)=-E \\ f(1)-g(1)=E \\ \frac{d}{dx}\left[f(x)-g(x)\right]=0 \end{gathered} \tag{2.5.7}$$

代入之后就可以知道，事实上我们在处理一个有 4 个未知数和四个方程的系统：

$$\begin{gathered} a+b\cdot 0-3\cdot 0^2=E \\ a+bx-3x^2=-E \\ a+b\cdot 1-3\cdot 1^2=E \\ b-6x=0 \end{gathered} \tag{2.5.8}$$

这个方程组可以用我们喜欢的任何方法很容易地求解。对于这个特定问题，它的解是 $(a,b)=(-\frac{3}{8},3)$。可以用同样的方式处理任意的参考函数 $g(t)$，并可以看到进一步计算极小极大近似是多么的容易。最大误差通常由原始函数和近似函数在应用近似的区间的端点处的差值给出。可以用和这里提到的相同的逻辑表达式得到三阶极小极大近似的值。当需要计算的极小极大近似的多项式次数大于三次的时候，将需要特殊的技巧。我们会碰上未知数比方程的个数多的情况，并因而无法求解方程系统。在这种情况下，需要求助于更加高深的数学技巧。

任意阶极小极大近似的求解

获得任意极小极大近似的系数绝非易事。对于求解并没有速度上的严格要求，因为它并不需要在实时的情况下进行，Remez 算法可以帮助我们手工解决这个问题，具体步骤如下：

1. 对系数向量 **c** 做一个初始值猜测。
2. 如果误差 E 满足要求，则停止计算，否则对给定 **c** 找到 $h(x)=g(x)-f(x)$的最大向量 **x**。
3. 得到线性系统 $g(x)-f(x)=(-1)^i E$ 系数向量 **c** 的值（忘记前面得到的 **c** 的值），给定最大向量 **x**，回到第 2 步重新计算。

简言之，就是选择一个系数的初始值并求出最大值，用它来进行迭代运算以提高解的精度（用 **c** 求出 **x**，再用 **x** 求出 **c**，等等）。作为一个求解的例子，我们用该算法来求解前面的例子。

假设给出的初始猜测是 **c**=(4,1)。现在需要找到最大值（即导数的根）。有许多方法能找到多项式的根，具体方法请见参考文献（参看[Math]）。对于这个特殊的问题，其数学是非常简单和直接的。

$$\begin{aligned} f(x)-g(x) &= a+bx-3x^2 \\ \frac{d}{dx}\left[f(x)-g(x)\right] &= b-6x=0 \\ x &= \frac{b}{6} \\ &= \frac{1}{6} \end{aligned} \tag{2.5.9}$$

我们找到一个一般的方程，它告诉我们在 $x=b/6$ 处总有最大值。显然，为了符合预期的构造，在 $x=0$ 和 $x=1$ 处也有最大值，因为它们是极小极大近似的边界。

对 x 有了初步的估计之后（在这个例子里，事实上只有一个未知的最大值），可以通过步骤 2 求出系数：

$$a+bx\pm E=3x^2, x=\left\{0,\frac{1}{6},1\right\}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & \frac{1}{6} & -1 \\ 1 & 1 & 1 \end{bmatrix}\begin{bmatrix} a \\ b \\ E \end{bmatrix}=\begin{bmatrix} 0 \\ \frac{1}{13} \\ 3 \end{bmatrix} \tag{2.5.10}$$

解这个线性方程组，得到解为 $(a,b,E)=\left(-\frac{11}{52},3,\frac{11}{52}\right)$，它与前面计算得到的极小极大值已经相差不远了。进行第 2 次迭代后，得到 $x=3/6=1/2$，把方程改写成如下形式：

$$a+bx\pm E=3x^2, x=\left\{0,\frac{1}{6},1\right\}$$

$$\begin{bmatrix} 1 & 0 & 1 \\ 1 & \frac{1}{2} & -1 \\ 1 & 1 & 1 \end{bmatrix}\begin{bmatrix} a \\ b \\ E \end{bmatrix}=\begin{bmatrix} 0 \\ \frac{3}{4} \\ 3 \end{bmatrix} \tag{2.5.11}$$

解这个线性系统，得到了已知的确切解：$\mathbf{c}=\left(-\frac{3}{8},3\right)$。显然，并不是所有的近似都是这么容易计算的。有时候，求根的过程是乏味的，并且需要一个迭代过程，例如牛顿-拉斐森（Newton- Raphson）法和 Haley 法迭代，而且有时候线性系统的求解也是非常难以处理的。非常幸运的是，计算函数近似的时候只需要计算一次系数向量，因此并不需要很高的效率。图 2.5.3 展示了近似和真实函数之间的误差，可以清晰地看到它在极大值和零误差方面服从前面提到的理论。

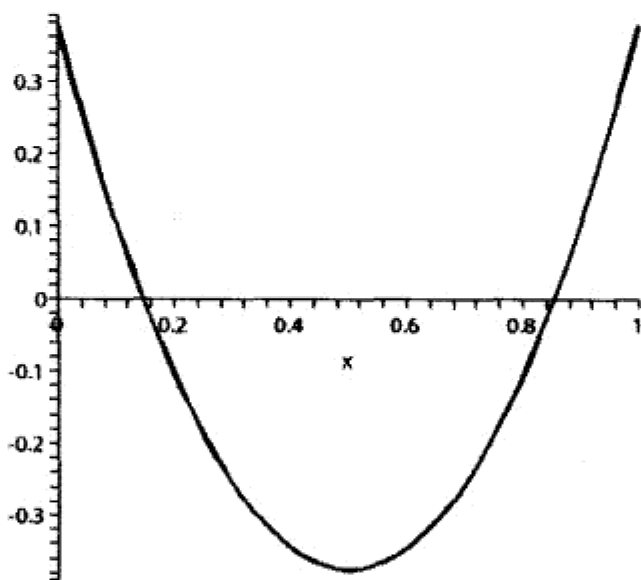


图 2.5.3 一阶极小极大近似 $f(x)$和 $g(x)$之间的误差图

2.5.4 误差分析

近似计算有一个非常特殊的事项，即误差分析是至关重要的一项工作。极小极大近似没有直接定义最佳近似，它仅仅定义了在给定的多项式次数下最大误差最小的近似。它并不保证在给定比特精度下是最快的方法。为了证明这一点，假设要在区间[0,π/2]上对正弦函数做近似。我们可以很容易地通过三角恒等式来计算函数剩下的值。正弦函数的四阶极小极大近似以系数向量形式给出。

$$\mathbf{c}=(0.000107652, 0.9964223759, 0.0190787764, -0.2026644465, 0.02841900366) \quad (2.5.12)$$

用 Horner 形式表示的时候，这个方程一共有四次乘法。换句话说，以 Horner 形式写成的正弦函数的泰勒级数只有四次乘法。

$$\sin(x) \approx x\left(1+x^2\left(-\frac{1}{3!}+x^2\left(\frac{1}{5!}-\frac{x^2}{7!}\right)\right)\right) \quad (2.5.13)$$

如果把 x^2 作为事先计算好的一个值，如图 2.5.4 和 2.5.5 所示，那么这个近似的误差函数就要比相同复杂度的极小极大近似稍好一些。所有这些都意味着极小极大近似是相当不错的近似，但是有时候其他的近似可能会更好一些，这要取决于具体的比特数（bit depth）。所以在计算的时候必须非常小心，以确保在给定的复杂性前提下极小极大近似为最佳近似。

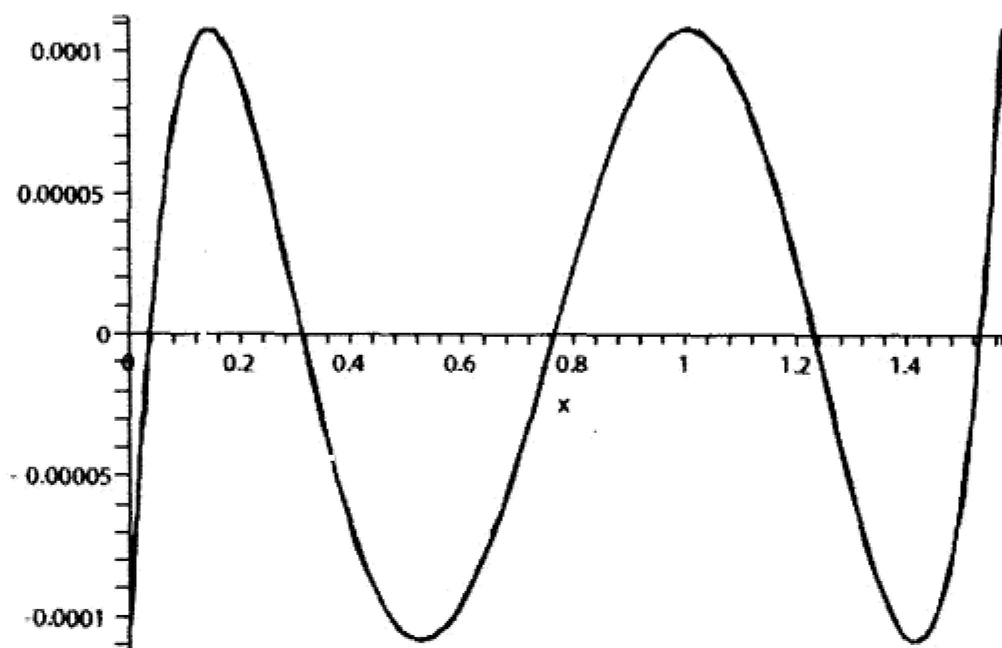


图 2.5.4 正弦函数的四阶极小极大近似

如果比较图 2.5.4 中的极小极大近似和图 2.5.5 中相同次数的泰勒级数，可以很明显地看出极小极大近似是优胜者。一个的最大误差为 0.0001，而另外一个则为 0.00016。最重要的是，当对精度要求不高的时候，极小极大近似实现的函数要比已有的 FPU 实现的函数更快。例如，用前面提到的方法对正弦函数做近似，在提供可信赖的近似情况下，其速度将是 FPU 提供的版本的两倍，且精度损失并不是很高，如表 2.5.1 和表 2.5.2 所示。如果用 SIMD（Single Instruction，Multiple Data）处理器来优化它，效果会更好。通过这种方法，可以同时计算 4 个正弦函数值，能把单个正弦函数的速度提高 7 倍，或者说在保证 23 bit 尾数精度的情况下至少将速度提高 6 倍。

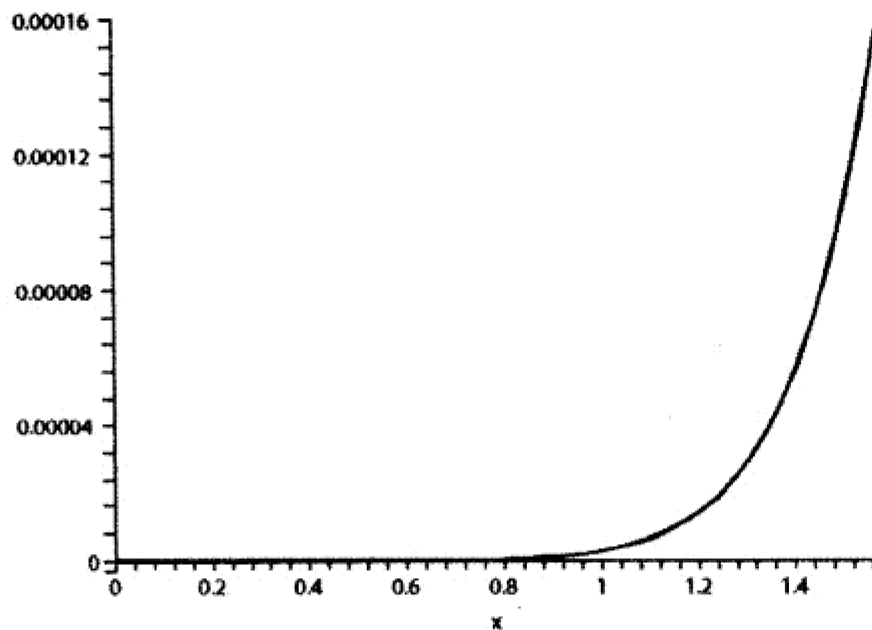


图 2.5.5　正弦函数的七阶泰勒级数近似

表 2.5.1　　正弦函数的泰勒级数的误差

度　数	Sin x 的泰勒等式	在 $\theta=[0,\pi/2]$ 时的最大误差
0	0	1
1	x	0.5707
2	x	0.5707
3	$x-x^3/6$	0.7516×10^{-1}
4	$x-x^3/6$	0.7516×10^{-1}
5	$x-x^3/6+x^5/120$	-0.4524×10^{-2}

表 2.5.2　　正弦函数极小极大近似的误差

度　数	Sia x 的极小极大近似等式	在 $\theta=[0,\pi/2]$ 时的最大误差
0	0.5	0.5
1	$0.1051+0.6366\cdot x$	0.1051
2	$-0.1385\cdot10^{-1}+1.1748\cdot x-0.3314\cdot x^2$	0.1385×10^{-1}
3	$-0.1365\times10^{-2}+1.0252\cdot x-0.7068\times10^{-1}\cdot x^2-0.1125\cdot x^3$	0.1365×10^{-2}
4	$-0.1076\times10^{-3}+0.9964\cdot x+0.19078\times10^{-1}\cdot x^2-0.2026\cdot x^3+0.2841\times10^{-1}\cdot x^4$	0.1076×10^{-3}
5	$-0.7064\times10^{-5}+0.9996\cdot x+0.2193\times10^{-2}\cdot x^2-0.1722\cdot x^3$ $+0.6097\times10^{-2}\cdot x^4+0.5721\times10^{-2}\cdot x^5$	0.706482×10^{-5}

从这一点上看，计算一个函数的近似非常简单。在实际应用中，可以对每一个 FPU 实现的函数进行近似，通过一些简单的恒等式几乎可以把所有函数的速度至少提高至两倍。

2.5.5　进一步改进近似

一共有 3 种方法可以减少近似误差。最明显的一种方法是提高多项式的次数，从而提高计算的复杂度。这不是一种理想的方法，因为它会提高计算的复杂度。另外一个技术是可以考虑把一个函数转化成其他函数的组合形式。举个例子，如果要计算 $\sin x\cdot\cos x$，就应该考虑计算两个函数（一个是正弦函数，一个是余弦函数）的近似是否比计算整个函数的近似来

得更快。在这个特定的例子下，它并不是更快。虽然当与计算两个函数的近似相比的时候，计算整个函数的近似会更快，但是对待它需要慎重考虑。对很复杂的函数来说，它就不是一个很好的办法，尤其是对有理函数来说更是如此。

最后一个提高近似的方法是简单地减少近似的区间。在第 1 个例子里，只需要在区间[0,π/2]上对正弦函数进行近似，三角恒等式可以用来计算这个区间以外的值。有些时候，最好计算函数的片断近似以降低复杂度。举个例子，如果再次计算正弦函数的近似，可以计算 18 个线性（一阶）均匀分布的近似值。如果函数是均匀分布的，我们就能非常容易地知道该使用哪个近似值——它是整个曲线的一个分割，我们可以把系数保存在一个表里来计算值。在嵌入式设备里，经常会使用正弦余弦表，它们表示了函数给定角度的值。如果使用这个技巧，为每个计算增加额外的加法和乘法，我们就可以通过更少的计算来达到更高的精度。这种做法是非常值得的，而且很容易计算。

最后能给出的建议是尽可能地使用几何关系和恒等式。FPU 的所有超越函数都可以使用极小极大近似来加速。为了减少近似的区间，惟一的问题就是要知道函数中存在什么样的几何关系。举个例子，对于函数 $\tan x$，初一看，这是一个非常难近似的函数，因为它有渐近线属性。幸运的是，如果考察如下恒等式：

$$\tan x = \frac{1}{\tan\left(\frac{\pi}{2} - x\right)} \tag{2.5.14}$$

就可以通过建立 $\tan x$ 函数在 0 到π/4 角度上的近似来显著减少精度问题。另外一个例子是考虑 tan 函数的逆：arctan 函数。这个函数也非常难，因为它的渐近线是与 x 轴平行的，所以它没有一个有限的定义域。幸运的是，通过一个奇妙的等式，可以把近似的定义域缩小到 0 与 1 之间：

$$\arctan x = \frac{\pi|x|}{2x} - \arctan\left(\frac{1}{x}\right) \tag{2.5.15}$$

只要找到的等式并不比用 FPU 计算函数本身更复杂，那就没有问题。对前面提到的两个例子，除法以及需要的其他额外工作的开销并没有产生什么的影响，因为这两个函数的等价 FPU 函数比正弦/余弦函数还要慢，因此计算这样的函数需要更多额外的复杂的逻辑。

2.5.6 参考文献

[Math] available on-line at *http://mathworld.wolfram.com/Root-FindingAlgorithm.html.*

2.6 应用于镜面和入口的斜视锥

Eric Lengyel
lengyel@terathon.com

人们已经开发了一些技术，用于渲染那些包含了具有递归本性的元素的 3D 图像。其中一些例子有：反射周围环境的镜子、能看到场景中远处的入口（portal，也翻译为“门”，本文统一用“入口”这个词）和应用了透明折射的水面。在这每一种情况下，都需要在虚拟摄影机的透视空间里渲染场景的一部分，而虚拟摄影机的位置和方向则是通过正在使用中的真实摄影机的位置来计算。例如，在镜子中看见的图像可以通过使用一个由真实摄影机在镜子中的镜像来进行渲染。

一旦这种图像是通过一个虚构的摄影机来渲染，当从真实摄影机的角度进行渲染的时候，往往会把它当做一个平面物体来处理。被选来描绘图像的平面是一个简单的平面，它会很自然地把该图像从环境里分离出来，就像一面镜子、一个入口或者水面。在通过虚拟摄影机渲染的过程中，可能会出现某些几何体比用来描绘图像的镜子和入口等的表面所在的平面更靠近摄影机的情况，如果这样的几何体被渲染出来，那么最终图像里将产生意外的走样。

解决这个问题的最简单办法，是使用一个用户定义的裁剪面裁掉在那个表面的所有图元。不幸的是，一些老的 GPU 不支持用户定义的裁剪面，如果使用的话，就必须采用基于软件的顶点处理的方法。其他现代的 GPU 支持一般的用户裁剪操作，但是启用这个功能需要修改顶点程序——这不是很方便，因为它需要为每个顶点程序维护两个不同版本。

本文提供了一个替代方案：使用被渲染的场景里已经存在的裁剪面。正常情况下，每个图元都要用视锥的六个面来裁剪，包括四个侧面、一个近平面和一个远平面。增加第 7 个裁减面，几乎总是能得到与近平面相冗余的结果，因为我们是对一个穿过视锥的平面做了另外一次裁减。我们的策略是把近面移动到那个表面所在的平面上，并用这种方法修改投影矩阵。因此仍然是用视锥的六个面进行裁剪，所以这样的修改能带来期望的结果，且不会有任何性能上的损失。而且，这个技术可以应用到任何投影矩阵上，包括常规的透视投影和正交投影，就像模板体积阴影算法里使用无限投影矩阵那样。

2.6.1 平面的表示

在考察投影矩阵和决定如何定义视锥的六个面之前，先快速地回顾一下 3D 图形中平面的工作方式。平面 **C** 在数学上用一个四维向量表示如下：

$$\mathbf{C}=\left\langle N_x,N_y,N_z,-\mathbf{N}\cdot\mathbf{Q}\right\rangle, \tag{2.6.1}$$

其中 **N** 是这个平面的法向量，**Q** 是这个面上的任意一个点。当且仅当点乘 **C** • **P** 为 0 的时候，齐次坐标点 $\mathbf{C}=\left\langle N_x,N_y,N_z,-\mathbf{N}\cdot\mathbf{Q}\right\rangle$ 在这个面上。对于在这个面正面的点，点乘为正；对于在背面的点，点乘则为负。

平面 **C** 被任何非零的标量缩放后仍然是同一个平面。同样，一个齐次坐标点 **P** 被一个非零的标量缩放后仍然是同一个点。如果平面 **C** 的法向量 **N** 是单位长度，并且 **P** 点的 *w* 坐标是 1，那么点乘 **C** • **P** 就是点 **P** 到面 **C** 的垂直带符号的距离。

平面是一个协变向量，因此必须从一个坐标系变换到另外一个坐标系，其方法是使用变换普通坐标点的矩阵的转置的逆来进行变换。当用投影矩阵变换平面的时候，这一点尤其重要，因为它不是正交变换。给定一个摄影机空间中的点 **P** 和一个摄影机空间中的面 **C**，用投影矩阵 **M** 产生一个裁剪面的点 **P'** 和一个裁剪面 **C'** 遵循下列等式：

$$\begin{aligned}\mathbf{P}'&=\mathbf{MP}\\ \mathbf{C}'&=\left(\mathbf{M}^{-1}\right)^{\mathrm{T}}\mathbf{C}\end{aligned} \tag{2.6.2}$$

转换这些等式得到下列公式，它会从裁剪空间变换到摄影机空间：

$$\begin{aligned}\mathbf{P}&=\mathbf{M}^{-1}\mathbf{P}'\\ \mathbf{C}&=\mathbf{M}^{\mathrm{T}}\mathbf{C}'\end{aligned} \tag{2.6.3}$$

2.6.2 投影矩阵

现在花一点时间来回顾一下投影矩阵的功能以及它和视锥裁剪面的关系。我们不考虑投影矩阵的任何特殊形式，只需要它是可逆的。这就允许我们的结果可以应用于任意的投影矩阵，包括有可能已经从标准形式修改过的矩阵。

回想一下，在 OpenGL 的摄影机空间（也叫做“眼空间”）里，摄影机的位置在原点并指向 *z* 轴负方向，就像图 2.6.1 里所展示的一样。作为完整的右手坐标系，*x* 轴向右，*y* 轴向上。（在 Direct3D 里，*z* 轴是颠倒的，且摄影机空间是左手坐标系。）顶点通常会被模型视图矩阵从它们特定的空间里变换到摄影机空间里。在本文里，我们不关心模型视图矩阵，而是假设顶点位置是在摄影机空间里直接指定的。

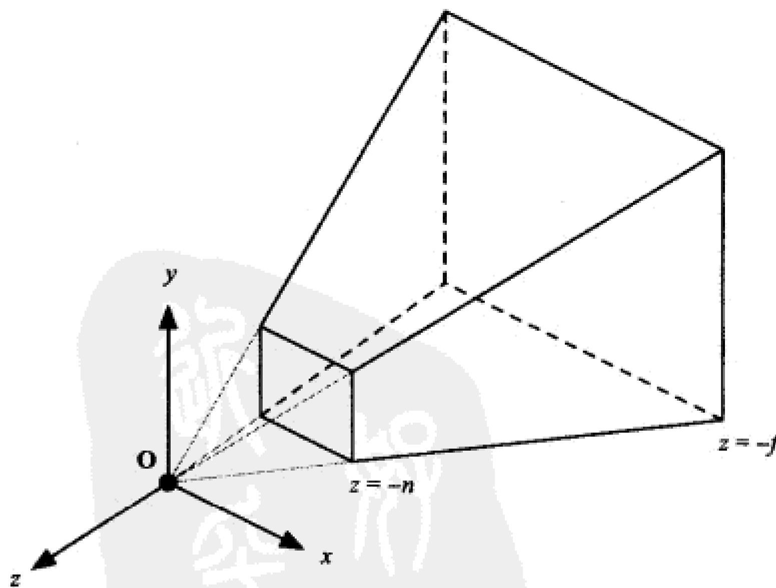


图 2.6.1 OpenGL 摄影机空间和标准视锥。近平面和远平面垂直于 *z* 轴，分别位于距离摄影机 n 和 f 处

标准的视锥是一个六个面的被切去顶端的棱锥，它围绕着摄影机的可见空间体。如图 2.6.1 所示，它被视口的四条边所在的四个面包围着，一个近平面在 $z=-n$ 处，一个远平面在 $z=-f$ 处。近平面和远平面通常垂直于摄影机的观察方向，但是对投影矩阵的修改将会移动这两个面并改变视锥的基本形状。

投影矩阵会把顶点从摄影机空间变换到齐次坐标裁剪空间。在 OpenGL 的齐次坐标裁剪空间里，如果满足以下条件，四维点(x, y, z, w)就位于摄影机空间的视锥投影内：

$$
\begin{aligned}
-w &\le x \le w \\
-w &\le y \le w \\
-w &\le z \le w
\end{aligned} \tag{2.6.4}
$$

用 w 坐标执行透视除法后会把点变换到单位化的设备坐标里，在那里，视锥里每个点的坐标都在[−1, 1]区间内。我们的目标是要改变投影矩阵，从而使给定的任意平面上的点在单位化的设备坐标里都有一个值为−1 的 z 坐标。

图 2.6.2 展示了四维齐次坐标裁剪空间的三维片断的 x 和 z 分量。在这个片断里，所有点的 w 坐标都是 1，且用等式 2.6.4 描述的视锥的投影以形成立方体的六个面为边界。每个面的 w 坐标都是 1，严格地说，每个 x，y 和 z 坐标都是±1，并且其余的分量都是 0，如表 2.6.1 所示。给定一个任意的投影矩阵 **M**，等式 2.6.3 可以用来将这些面映射到摄影机空间中。这就产生了表 2.6.1 里的简单公式，在那里每个摄影机空间的平面被表示成一个投影矩阵的两行的和或者差。

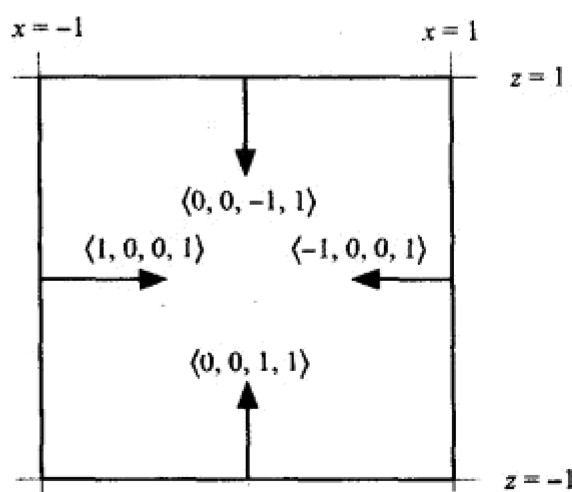


图 2.6.2 OpenGL 齐次裁剪空间在 $w=1$ 处的三维片断，以及这个空间里形成立方体的 6 个裁剪平面中的 4 个平面

表 2.6.1 OpenGL 裁剪空间和摄影机空间的视锥面。矩阵 M 表示把顶点从摄影机空间变换到投影空间的投影矩阵。下标表示是矩阵 M 的第 i 行

锥体平面	裁剪空间坐标	摄影机空间坐标
Near（近平面）	<0, 0, 1, 1>	$M_4 + M_3$
Far（远平面）	<0,0, −1,1>	$M_4 - M_3$
Left（左平面）	<1, 0,0,1>	$M_4 + M_1$
Right（右平面）	<−1,0,0,1>	$M_4 - M_1$
Bottom（底平面）	<0,1,0,1>	$M_4 + M_2$
Top（顶平面）	<0, −1,0,1>	$M_4 - M_2$

2.6.3 裁剪面的修改

假设 $\mathbf{C} = \langle C_x, C_y, C_z, C_w \rangle$ 是图 2.6.3 显示的平面，它在摄影机空间里有特定的坐标，我们将用它来裁剪几何体。摄影机应该在那个面的负方向上，因此能确定 $C_w<0$。平面 **C** 将会代替视锥的普通近平面。如表 2.6.1 所示，摄影机空间的近平面由投影矩阵 **M** 的最后两行给出，所以一定要满足：

$$\mathbf{C} = \mathbf{M}_4 + \mathbf{M}_3 \tag{2.6.5}$$

我们不能改变投影矩阵的第 4 行，因为投影矩阵用它把 z 坐标的负向移动到 w 坐标里，而且这对于纹理坐标之类的顶点属性的正确的透视插值是很有必要的。因此我们别无选择，只能用下式来替换投影矩阵的第 3 行

$$\mathbf{M}_3' = \mathbf{C} - \mathbf{M}_4 \qquad (2.6.6)$$

经过等式 2.6.6 里的替换后，视锥的远平面 **F** 变成：

$$\begin{aligned}\mathbf{F} &= \mathbf{M}_4 - \mathbf{M}_3' \\ &= 2\mathbf{M}_4 - \mathbf{C}\end{aligned} \qquad (2.6.7)$$

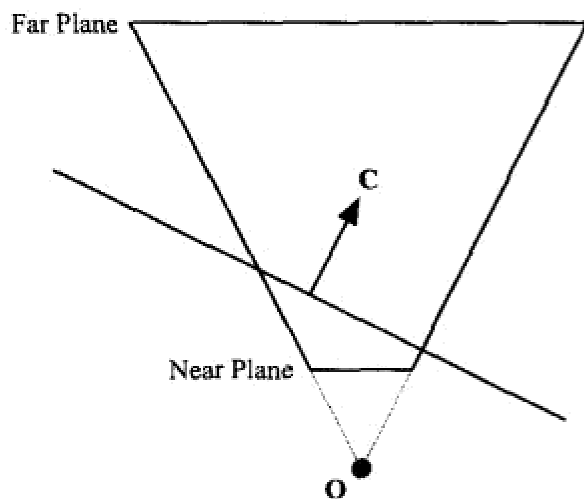


图 2.6.3 用任意平面 C 代替视锥的近平面

这个事实呈现了透视投影的一个重要问题。一个透视投影矩阵的第四行一定为 $\mathbf{M}_4$=(0,0,–1,0)，这样裁剪空间的 w 坐标才会接受摄影机空间的 z 坐标的负值。由此可以推导出，如果 C_x 或者 C_y 非零的话，近平面和远平面将不再平行。非常不直观并且会导致视锥变成一个很不合理的形状。注意，对于任意点 $\mathbf{P}=\langle x,y,0,w\rangle$，如果满足 $\mathbf{C}\cdot\mathbf{P}=0$，则意味着 $\mathbf{F}\cdot\mathbf{P}=0$，由此能够推断出远平面和近平面在 x-y 平面处相交，如图 2.6.4（a）所示。

由于点的最大投影深度接近远平面，所以投影深度不再表示沿着 z 轴的距离，但是该位置相应的值在新的近平面和远平面之间。这就对沿着不同方向的视锥的深度缓冲有很大的影响。幸运的是，有一个措施可以使这个影响最小，就是使远平面和近平面的夹角尽可能的小。平面 **C** 拥有一个隐含的缩放因子，这里没有用任何方式去限制它。改变平面 **C** 的缩放因子会使远平面 **F** 的方向跟着改变，因此需要在不用裁剪原始视锥的任何部分的情况下计算适当的缩放，以使 **C** 和 **F** 之间的夹角最小，如图 2.6.4（b）所示。

使 $\mathbf{C}' = \left(\mathbf{M}^{-1}\right)^{\mathrm{T}}\mathbf{C}$ 成为新的近平面在裁剪空间里的投影（使用原始的投影矩阵 **M**）。视锥相对于平面 **C'**的角 **Q'**由下式给出

$$\mathbf{Q}' = \left\langle \operatorname{sgn}\left(C_x'\right), \operatorname{sgn}\left(C_y'\right), 1, 1\right\rangle \qquad (2.6.8)$$

其中 sgn 函数会返回其自变量的符号。

$$\operatorname{sgn}(k) = \begin{cases} +1, & \text{如果 } k>0; \\ 0, & \text{如果} k=0; \\ -1, & \text{如果} k<0。\end{cases} \qquad (2.6.9)$$

（对于大多数透视投影，假设 C_x' 和 C_y' 的符号与 C_x 和 C_y 相同不会有什么问题，因为这样可以避免把 **C** 投影到裁剪空间里）一旦确定了 **Q'**的值，就可以通过计算 $\mathbf{Q} = \mathbf{M}^{-1}\mathbf{Q}'$ 得到它的摄影机空间的副本 **Q**。作为一个标准视锥，在两个边平面和远平面相交的地方，点 **Q** 在平面 **C** 的反面。

为了强制远平面包含点 **Q**，需要使 $\mathbf{F}\cdot\mathbf{Q}=0$。等式 2.6.7 里惟一可以改变的部分是平面 **C** 的缩放，因此我们引入一个因数 a，如下所示

$$\mathbf{F} = 2\mathbf{M}_4 - a\mathbf{C} \qquad (2.6.10)$$

解以 a 为变量的方程 $\mathbf{F}\cdot\mathbf{Q}=0$ 得到：

$$a = \frac{2\mathbf{M}_4 \cdot \mathbf{Q}}{\mathbf{C} \cdot \mathbf{Q}} \qquad (2.6.11)$$

用 $a\mathbf{C}$ 替换掉等式 2.6.6 里的 $\mathbf{C}$，得到：

$$\mathbf{M}_3' = \frac{2\mathbf{M}_4 \cdot \mathbf{Q}}{\mathbf{C} \cdot \mathbf{Q}}\mathbf{C} - \mathbf{M}_4 \tag{2.6.12}$$

并且这会得到如图 2.6.4（b）中所示的远平面的最佳方向。应该注意的是，在 $\mathbf{M}$ 是一个强制远平面平行于视锥中相交的两个边平面一条边的无限投影矩阵（例如，把常规的远平面放在无限远处）的情况下，这种技术也能正确地工作。

像前面提到的那样，调整视锥去实现任意平面的裁剪会影响深度缓存的精度，因为深度的值域在摄影机空间里不能用来沿着不同的方向。可以看到，沿着摄影机空间里的 $\mathbf{V}$ 方向，单位化的设备的 z 坐标所能达到的最大值由下式给出：

$$-\frac{a(\mathbf{C} \cdot \mathbf{V}) + V_z}{V_z} \tag{2.6.13}$$

通常情况下，深度缓存的精度会随着裁剪面 $\mathbf{C}$ 的法线方向和 z 轴的夹角的增加而减少，随着摄影机离裁减面的距离的增加而增加。更多关于深度精度的问题，可以在[Leng04]里找到。

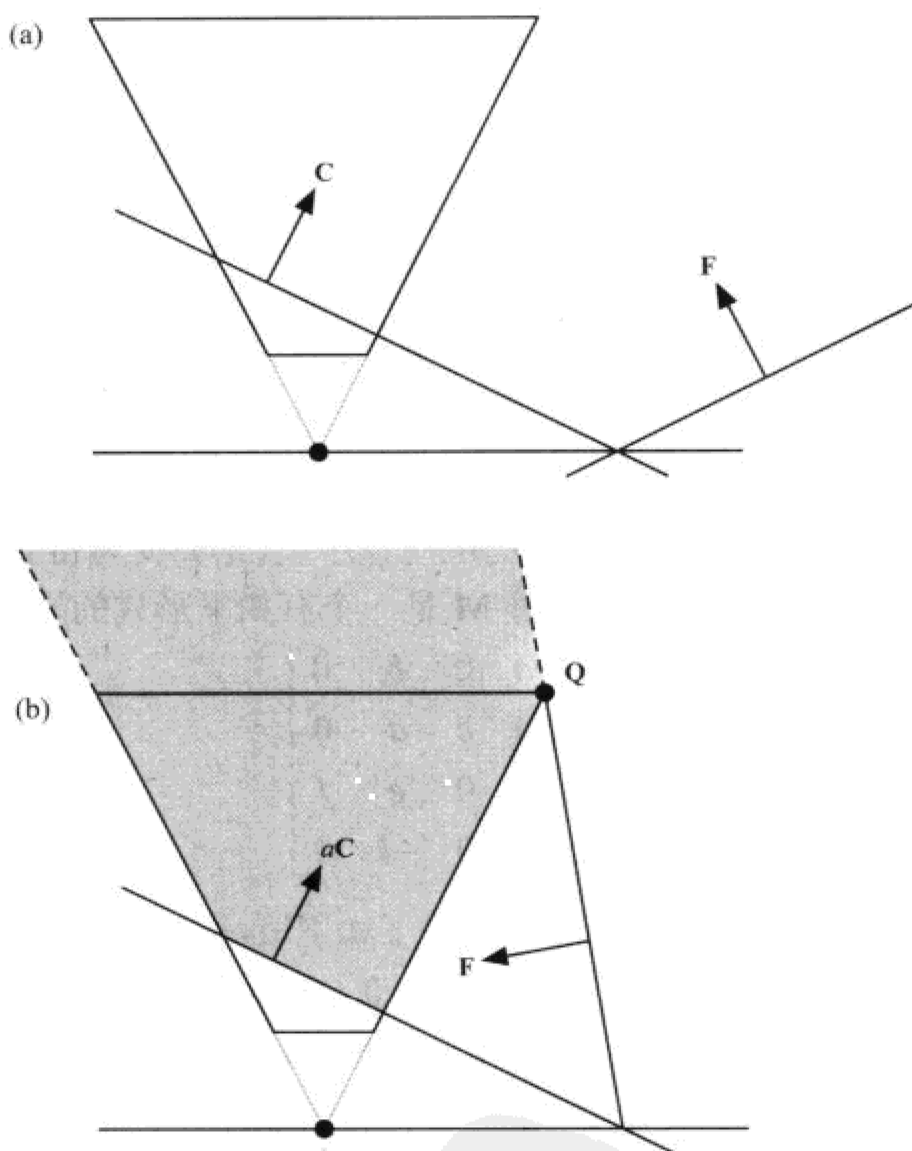


图 2.6.4 （a）用方程 2.6.7 修正过的远平面 **F** 和近平面 **C** 相交于 x-y 平面。（b）近平面 **C** 通过方程 2.6.11 给出的 a 值来缩放，这调整了远平面，使近平面和远平面的夹角尽可能小，不至于裁减到原始视锥的任何一部分。阴影部分表示没有被修改后的视锥裁减掉的部分

2.6.4 OpenGL 实现

标准的 OpenGL 透视投影矩阵 **M** 如下：

PDG

$$\mathbf{M}=\begin{bmatrix}\frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0\\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0\\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n}\\ 0 & 0 & -1 & 0\end{bmatrix} \tag{2.6.14}$$

其中 n 是到近平面的距离，f 是到远平面的距离，l ,r , b 和 t 表示依靠视锥的四个边平面确定的近平面外部长方体的左边、右边、底边和顶边。因为 $\mathbf{M}_4$=(0, 0, −1, 0)，等式 2.6.12 可以简化为：

$$\mathbf{M}_3'=\frac{-2Q_z}{\mathbf{C}\cdot\mathbf{Q}}\mathbf{C}+\langle 0,0,1,0\rangle \tag{2.6.15}$$

点 **Q** 由下面的等式给出：

$$\mathbf{Q}=\mathbf{M}^{-1}\left\langle \operatorname{sgn}(C_x),\operatorname{sgn}(C_y),1,1\right\rangle \tag{2.6.16}$$

应用 **M** 的逆，可以得到

$$\mathbf{Q}=\begin{bmatrix}\operatorname{sgn}(C_x)\frac{r-l}{2n}+\frac{r+l}{2n}\\ \operatorname{sgn}(C_y)\frac{t-b}{2n}+\frac{t+b}{2n}\\ -1\\ 1/f\end{bmatrix} \tag{2.6.17}$$

程序清单 2.6.1 演示了在一个基于 OpenGL 的应用程序里，是如何为得到一个典型的投影矩阵来实现投影矩阵的修改的。它假设投影矩阵 **M** 是一个有如下形式的透视投影：

$$\mathbf{M}=\begin{bmatrix}a & 0 & b & 0\\ 0 & c & d & 0\\ 0 & 0 & e & f\\ 0 & 0 & -1 & 0\end{bmatrix} \tag{2.6.18}$$

其中 $a>0$，$c>0$，并且 $f\neq 0$。这段代码直接应用了如下所示的矩阵 **M** 的逆。

$$\mathbf{M}^{-1}=\begin{bmatrix}1/a & 0 & 0 & b/a\\ 0 & 1/c & 0 & d/c\\ 0 & 0 & 0 & -1\\ 0 & 0 & 1/f & e/f\end{bmatrix} \tag{2.6.19}$$

程序清单 2.6.1　对等式 2.6.18 所表示的 OpenGL 投影矩阵的投影矩阵修改的实现如方程 2.6.18 所示（传递给 `ModifyProjectionMatrix` 函数的参数 `clipPlane` 表示了要进行裁减的摄影机空间。）

```
inline float sgn(float a)
{
```

```
    if (a > 0.0F) return (1.0F);
    if (a < 0.0F) return (-1.0F);
    return (0.0F);
}

struct Vector4D
{
    float    x, y, z, w;

    Vector4D() {}

    Vector4D(float a, float b, float c, float d)
    {
        x = a; y = b; z = c; w = d;
    }

    // 标量积
    Vector4D operator *(float s) const
    {
        return (Vector4D(x * s, y * s, z * s, w * s));
    }

    // 点积
    float operator *(const Vector4D& v) const
    {
        return (x * v.x + y * v.y + z * v.z + w * v.w);
    }
};

void ModifyProjectionMatrix(const Vector4D& clipPlane)
{
    float        matrix[16];
    Vector4D     q;

    // 从 OpenG 得到当前投影矩阵
    glGetFloatv(GL_PROJECTION_MATRIX, matrix);

    // 通过乘以投影矩阵的逆矩阵，把裁剪空间的角点变换到裁剪面的反面，即摄影机空间
    q.x = (sgn(clipPlane.x) + matrix[8]) / matrix[0];
    q.y = (sgn(clipPlane.y) + matrix[9]) / matrix[5];
    q.z = -1.0F;
    q.w = (1.0F + matrix[10]) / matrix[14];

    // 计算缩放平面的向量
    Vector4D c = clipPlane * (-2.0F / (clipPlane * q));

    // 替换投影矩阵的第 3 行
    matrix[2] = c.x;
    matrix[6] = c.y;
    matrix[10] = c.z + 1.0F;
```

```
        matrix[14] = c.w;

        // 加载回 OpenGL 中
        glMatrixMode(GL_PROJECTION);
        glLoadMatrix(matrix);
    }
```

2.6.5 Direct3D 实现

在 Direct3D 环境里，摄影机空间是左手系的，并且近平面相当于一个裁减空间的 z 坐标为 0 的点的集合。因此表 2.6.1 里的近裁减面的值应该针对 Direct3D 应用程序改为(0,0,1,0)。因此，近平面在摄影机空间的值简单地由 $\mathbf{M}_3$ 给出。这意味着投影矩阵的第 3 行的项是近平面在摄影机空间中的坐标。远平面 **F** 仍然由 $\mathbf{M}_4 - \mathbf{M}_3$ 给出，这样在用任意的平面 **C** 替换掉投影矩阵的第 3 行之后得到等式：

$$\mathbf{F} = \mathbf{M}_4 - a\mathbf{C} \tag{2.6.20}$$

解出缩放因子 a，使远平面包含由等式 2.6.8 的逆投影所给的点 **Q**，这样可以得到

$$\mathbf{M}_3' = \frac{\mathbf{M}_4 \cdot \mathbf{Q}}{\mathbf{C} \cdot \mathbf{Q}}\mathbf{C} \tag{2.6.21}$$

下面给出标准的 Direct3D 的透视投影矩阵 **M**

$$\mathbf{M} = \begin{bmatrix} \dfrac{2n}{r-l} & 0 & \dfrac{r+l}{r-l} & 0 \\ 0 & \dfrac{2n}{t-b} & \dfrac{t+b}{t-b} & 0 \\ 0 & 0 & \dfrac{f}{f-n} & -\dfrac{fn}{f-n} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.6.22}$$

这里每个值都和等式 2.6.14 里的 OpenGL 透视投影矩阵里所表示的意义一样。既然这样，等式 2.6.21 可以简化为：

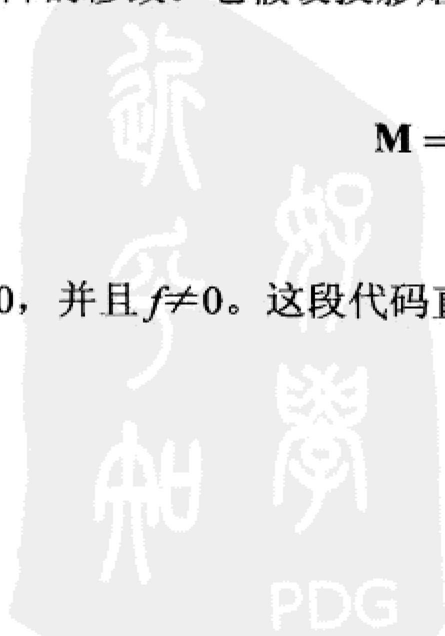
$$\mathbf{M}_3' = \frac{Q_z}{\mathbf{C} \cdot \mathbf{Q}}\mathbf{C} \tag{2.6.23}$$

等式 2.6.16 给出点 **Q** 在 Direct3D 里以及在 OpenGL 里的值。应用等式 2.6.22 里所给的投影矩阵 **M** 的逆，得到和等式 2.6.17 里相同的 **Q** 点坐标，只不过 z 坐标是负的。

程序清单 2.6.2 示范了在一个基于 Direct3D 应用程序里，如何为得到一个典型的投影矩阵来实现投影矩阵的修改。它假设投影矩阵 **M** 是一个有如下形式的透视投影：

$$\mathbf{M} = \begin{bmatrix} a & 0 & b & 0 \\ 0 & c & d & 0 \\ 0 & 0 & e & f \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.6.24}$$

其中 $a > 0$，$c > 0$，并且 $f \neq 0$。这段代码直接应用了如下式所示的 **M** 的逆矩阵。



$$\mathbf{M}^{-1}=\begin{bmatrix} 1/a & 0 & 0 & -b/a \\ 0 & 1/c & 0 & -d/c \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1/f & -e/f \end{bmatrix} \tag{2.6.25}$$

程序清单 2.6.2 对等式 2.6.24 所表示的 Direct3D 投影矩阵的投影矩阵修改的实现如方程 2.6.24 所示（传递给 ModifyProjectionMatrix 函数的参数 clipPlane 表示了要进行裁减的摄影机空间,这段代码里使用的 Vector4D 类在程序清单 2.6.1 里）。

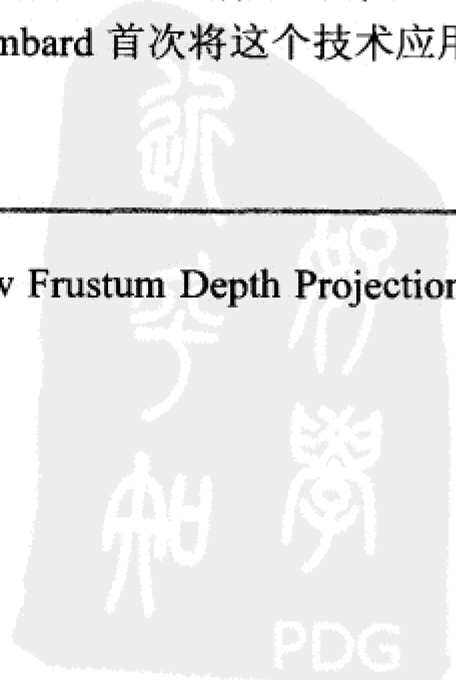
```
void ModifyProjectionMatrix(const Vector4D& clipPlane)
{
    D3DXMatrix    matrix;
    Vector4D      q;

    // 从 Direct3D 中获取当前的投影矩阵
    D3DDevice.GetTransform(D3DTS_PROJECTION, &matrix);

    // 通过乘以投影矩阵的逆矩阵，把相对于裁减面的裁减空间中的角点变换回摄影机空间
    q.x = (sgn(clipPlane.x) - matrix._31) / matrix._11;
    q.y = (sgn(clipPlane.y) - matrix._32) / matrix._22;
    q.z = 1.0F;
    q.w = (1.0F - matrix._33) / matrix._43;

    // 计算缩放后的平面向量
    Vector4D c = clipPlane * (1.0F / (clipPlane * q));

    // 替换投影矩阵的第 3 行
    matrix._13 = c.x;
    matrix._23 = c.y;
    matrix._33 = c.z;
    matrix._43 = c.w;

    // 将其加载入 Direct3D 中
    D3DDevice.SetTransform(D3DTS_PROJECTION, &matrix);
}
```

2.6.6 致谢

感谢 Nvidia 的 Cass Everitt 对该主题所做的一些有趣的讨论，是他提出了移动近平面到任意方位的原始构想。还要感谢 Yann Lombard 首次将这个技术应用到 Direct3D 环境中。

2.6.7 参考文献

[Leng04] Lengyel, Eric. "Oblique View Frustum Depth Projection and Clipping." In *Journal of Game Development*, Vol. 1, No 2, 2005.



PDG

第 3 章

人工智能

PDG

引　言

美国西北大学，Robin Hunicke
hunicke@cs.northwestern.edu

放眼四顾，无论是在 GDC（游戏开发者大会）和 E3 大展上，还是在游戏传媒和大众传媒领域，人们都在谈论“次世代”游戏。学者和玩家们的观点一致：游戏 AI 的革新有能力带领我们超越那些令人绚目的特效画面和熟悉的游戏类型，到达一个令人激动、从未体验过的新领域。如果做得足够好，AI 可以帮助我们使游戏产品的内容更加丰富多彩，改进开发流程，降低游戏的制作成本，同时巩固并拓展游戏产品的受众人群。

说起来容易做起来难！大家都很清楚，这个“简单”或“知名”的技术在几年前才刚刚得到人们的重视，它不会带我们直接跳到下一个“关卡”。但是，当我们展望未来，一切又变得模糊不清。当我们准备迎接新硬件（带有全新的、前所未见的性能）的时候，开发商们不禁要问，从逻辑上讲，游戏 AI 的下一步该走向哪里？我们是否有足够的时间和资源来迈出这一步？

有一些开发商已经开始了相关的实验：游戏的叙事、角色和情感、程序化内容、永续性、声望以及游戏的结局。在这些团队中，游戏 AI 和游戏策划的任务彼此交织，他们之间要互相沟通设计策略，通报游戏结构的变更。虽然新技术不断地融入进来，但是在游戏的平衡性、玩家操控、反馈和“好玩性”等方面，也必须符合现有的这些标准。工作的内容不同了，工作的难度也增加了。

实验和创新是工作的重点，但是可靠的工程却是必须的。我们不断地研究着游戏 AI 新的表现形式，研究它在游戏开发中扮演的新角色。我们的这些努力，在系统的清晰性和最优化方面起到了关键的推动作用。程序人员辛苦工作，在所有可能的地方去平衡那些快速、可预见的算法，同时还要保持高效、可调试的代码。策划工具必须对程序员和非程序员提供同样的一致性和透明性。为了实现它所有的好处，并行开发也严重影响着这些利害关系。

围绕着当代游戏 AI，不断涌现出大量的新问题。问题与思考，就是本章所涉及的内容。这个领域的很多文章都是探索性质的，还有一些文章则综合性地介绍了一些更为广阔、复杂的学科领域，多少突破了现有“精粹”文章的传统形式。为此，我们还是收录了一些比较通俗的文章，对一些常见的话题进行探讨，包括：搜索和寻径、目标锁定、策略、格斗分析。即使如此，我们也希望大家能在这里找到让你们惊喜的内容。

展望未来，很难说哪些 AI 的梦想能够真正变为现实（根据以往的记录，到目前为止，现实多少让人有些沮丧）。不妨再看得更远些，如果要实现最简单的目标，那么接下来又会面临一大堆什么样的问题呢？

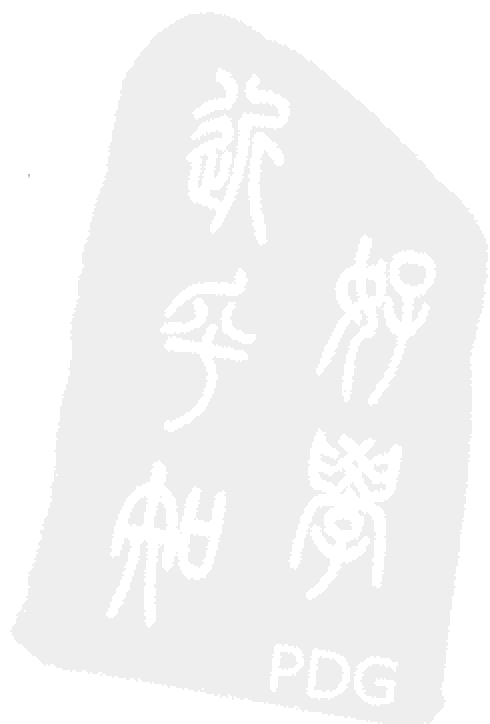
随着游戏角色不断增加的“自治力”，从游戏策划的角度来看，它们会变得越来越难以“控制”吗？程序化内容会让游戏看上去感觉像是从一个模子里出来的吗？会因此影响玩家的游戏体验吗？游戏的永续性会导致复杂的、但不可预知的（和不想要的）结局吗？会因此让那些非常投入的玩家们伤心吗？这样的结局模式，与那种预先设计好各种情况变化的结局有着天壤之别。未来的 AI，能让我们在已保存的游戏中看到某些类似全球变暖的情况吗？

随着游戏 AI 和游戏策划二者之间的关系越来越密不可分，该如何将 AI 系统从游戏相关的问题和实现中抽取出来呢？是否有正确的语言，用来讨论、设计适用范围更广，可重复使用的解决方案呢？就其本身而言，我们是否能够精确定义游戏中的 AI 组件呢？几年之内，这一章的内容里是否能有一个或三个新的标题呢？

当然，除非你和我们剩下的这些人一样：作者、开发商、研究人员和玩家，大家一起努力工作（和努力地玩），让明天的游戏做得更好。对我们而言，这些问题并不会让我们心灰意冷，反倒让我们兴奋不已。因为和其他所有事情一样，挑战本身也是乐趣所在。

希望大家把自己看成是我们这个阵营的一分子。如果不是这样想的，那么请考虑一下，加入我们这个阵营吧。接下来，就从本章的内容开始，再去阅读其他书籍，进而再将所学的知识应用到实践中。希望大家能不断地学习进取，不断地与大家展开讨论、交流。我们的工作远没有结束，新面孔总是能找到自己的空间。

祝大家阅读愉快！



3.1 利用导航网格实现自动掩体寻找

Radical Entertainment 公司，Borut Pfeifer
borut_p@yahoo.com

在很多与射击有关的游戏中，NPC（非玩家角色）的掩体寻找都是通过放置隐身（不可见的）的实体来实现的。这个不可见的实体所表示的区域可以用来作为掩体，躲避射击。无论是独立于导航图，还是放到导航图中，关卡策划必须手工定位这些放置地点。在大型的游戏关卡中，或者在那些开放的、可自由探索的游戏世界中，这件事情很快会变成一个很耗时的工作。

本文介绍了如何将导航网格与碰撞信息结合起来，让 NPC 可以自动地找到掩体的位置。我们将导航网格进行了扩展，为它增加了一些附加信息。这些附加信息是用来描述邻居网格的导航性的。然后，可以使用标准的、运行时搜索算法，为 NPC 找到一个合法的掩体位置，从而为开发人员节省大量的时间和精力。

3.1.1 导航网格

Greg Snook 在他的文章《使用导航网格简化 3D 移动和寻径》一文中，详细地描述了如何在游戏中使用网格（[Snook00]）。还有一些文章（[Tozour02]、[White02]）则讨论了一些特殊的技术，简化了导航网格，加速了在导航网格上的搜索速度。本文所使用的网格类型是一个基于三角形的网格，和游戏世界里碰撞检测使用的网格一模一样。

特别要指出的是，对于导航网格中每个可以走到的三角形，在碰撞网格中都有一个直接与它相对应的三角形（这个三角形本身的数据是可以共享的）。但是，碰撞网格中会有很多无法通过的三角形，这些三角形在导航网格中没有表示。

1. 创建导航网格

网格中的每一个三角形都最多有 3 个邻居三角形，但其中总会有个别的邻居是无法通过的。把所有的三角形都存储起来，以便其他程序可以使用（如印花贴图）。每个三角形要为它的所有邻居三角形设置一个位域（bitfield），表示其可能的一些遍历选择（对于本文，这些选择就是可通过、直立掩体、蹲伏掩体和不可通过）。

PDG

在前期处理阶段，需要确定每个三角形的特征位。对每个三角形的 3 个边，以适当的高度，分别进行光线投射测试（参见图 3.1.1）。一旦确定了每个边上可用的掩体，或者没有可用掩体，就可以把这些信息保存起来。每个三角形的这些信息不足一个字节（因为每个边的信息只有 2 个比特）。

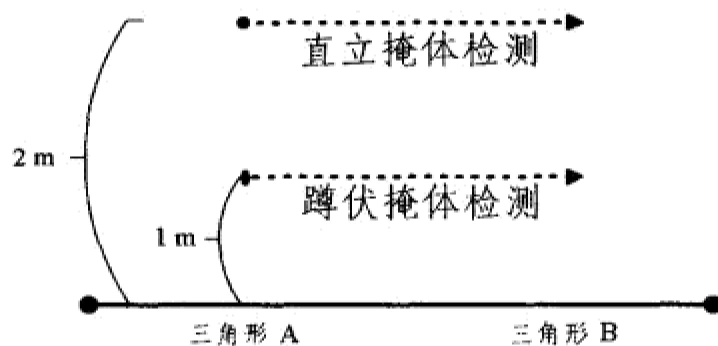


图 3.1.1　在前期处理阶段，在每个边的前视图方向上进行光线投射测试

对于这两种简单掩体的不同之处（直立掩体和蹲伏掩体），可以假设 NPC 是真人大小的。如果想在 NPC 搜索掩体的过程中，可以使用更多不同高度的掩体，可以简单地增加掩体高度值的个数，设置更多不同的掩体高度（例如，每米高度设置一个特征位），或者把三角形的高度值保存在导航图中（其他潜在的应用也可以使用这些数据，比如判断 NPC 是否可以爬上或跳上这个相邻的三角形）。

2. 测试三角形以寻找掩体

为了判断某个给定的三角形是否可以作为一个隐蔽点，必须做下列两件事情：

1. 进行路径检测，确定从这个给定三角形的中心到攻击目标点之间是否存在一条直线路径。也就是说，希望能找到一个隐蔽点，不被那个攻击目标发现或射到。这个工作可以使用简化的二维线段相交测试，详见 Snook 的文章[Snooke02]。

2. 如果这个三角形没有一条通向攻击目标的直线路径，那么三条边中肯定有一条是位于这个三角形的中心和攻击目标之间。这时候，就检测这条边的特征位。如果在前期处理中，这个连接被标识为蹲伏掩体或直立掩体，那么这个三角形就可以被接受为相对于那个攻击目标的隐蔽点。

3. 局限性

对于这种类型的导航图及相关的掩体检测，确实要面临两个限制。首先，由于导航地图是经过预处理的，所以算法只能找到静态掩体（对于额外添加的隐蔽点，可以重复使用一个包含足够多动态对象的列表）。

另外一个限制是，对瞄准线进行的二维数学运算，使我们受限于只能使用墙体式掩体，而地面地形则不能作为掩体。即使 NPC 和它的攻击目标之间有一座小山，也不能作为掩体。在很多环境下，如城市或室内，这种局限性倒也不算什么。

3.1.2　开放目标寻路

为了搜索一个隐蔽点，寻路系统需要支持开放目标搜索的概念。在一个典型的寻路需求中，搜索工作从一个起点开始，到一个指定的终点结束。因为准确地知道满足我们目标的元素，所以这种寻路方法称为“固定目标寻路”。而开放目标搜索只是简单地去寻找满足既定条件的元素，但不知道满足条件的元素具体在什么位置，是什么样的东西。

为此，需要将搜索工作参数化成两大类：

估价函数（Heuristic）：A*算法使用这个函数来计算每个节点的权值。

目标检测（goal test）：明确一个给定的节点是否满足搜索的目标参数。

寻路函数大致如下：

```
template<class PathSearch>
PathResult FindPath(NodeID startNode,
            Path* pInputPath,
            PathSearch searchParam);
```

作为参数化数据的两种方式，也可以使用一个基类，把估价函数和目标测试函数作为基类的虚拟函数；或者使用函数指针。但是在理论上，使用模板（template）可以改进游戏机的代码局部性（locality）（因为它不用对所有节点都调用各自的虚拟函数）。这些游戏机的指令缓存都比较小，比如 PS2。

在有些游戏的实现中，人们把这些开放目标搜索（例如找到一个掩体的位置）算法在寻路系统之外加以实现。但是，如果使用前面介绍的那个方法（method），可以充分利用寻路引擎中所有现成的搜索代码，以及其他特性（如时间分片）。

3.1.3 搜索掩体位置

由于定义掩体的概念是相对的，而且也经常变化，因此要想检测一个掩体，不得不提供几个输入参数。主要的两个参数就是 AI 主体的位置（搜索的起始点）和攻击目标位置（相对于这个点的位置，才能确保确实身处掩体之中）。

除此之外，还需要一个给定的方向（*given direction*）和允许的偏离角度（*acceptable angle*）。这样就可以对这个方向上的节点计算权值。这个方向其实就是 NPC 已经准备要移动的方向。同时还要给出一个“最大距离”（*maximum distance*），把搜索结果限定在几个合理的掩体位置上。不然的话，NPC 只会考虑那些在战团中心附近的区域。

可以将模板参数 PathSearch 类传递给前面那个 FindPath 函数。这个类大致定义如下：

```
class PathSearchForCover
{
    PathSearchForCover(Vector start,
                       Vector coverFrom,
                                                Vector inDirection,
                       float angle,
                       float maxDist);

    bool TestGoal(NodeID);

    float GuessRemaining(NodeID);
    ...
}
```

目标检测函数的动作就是前面定义的掩体检测。如果某个节点具备了一个掩体的条件，那么，这个搜索就会找到一个可接受的终点（但是，这并不能保证找到最佳的掩体。但在实际应用

中，这一点无关紧要)。估价函数的功能就像一个标准的 *A* 点到 *B* 点的路径估价（或者一个类似的估价方法：曼哈顿距离 Manhattan distance)，因为它会返回一个从起点到最终目标的直线距离。

在计算节点权值的方法（method）中，还有两个附加规则：

- 位于攻击目标后面的节点（这样，就得从起点位置出发，经过攻击目标，才能进入掩体)，可以把它的权值设得很高（10 倍)。这可以防止 NPC 做出“不理智”的行为：从玩家面前飞奔过去，跑到玩家身后的掩体里。
- 如果节点位于给定方向上，并且在给定的角度范围之内，那么它的权值就只有正常水平的三分之一。这样可以确保搜索的方向与 NPC 想要前进的方向一致。

一旦搜索工作结束，而且找到了一个隐蔽点，我们的工作就接近（但不是彻底）完成了。由于在掩体检测中使用的都是每个三角形的中心点，所以还需要计算出这个三角形中最满意的隐蔽位置。

如果知道当前在请求路径的 AI 主体的宽度，就以为我们提供掩体的三角形的边为基准，沿着这条边，离攻击目标最近的那个顶点的方向就是 AI 主体下一步的行进方向。将隐蔽点做一个垂直方向的位移，位移的距离就是 AI 主体的宽度，如图 3.1.2 所示。

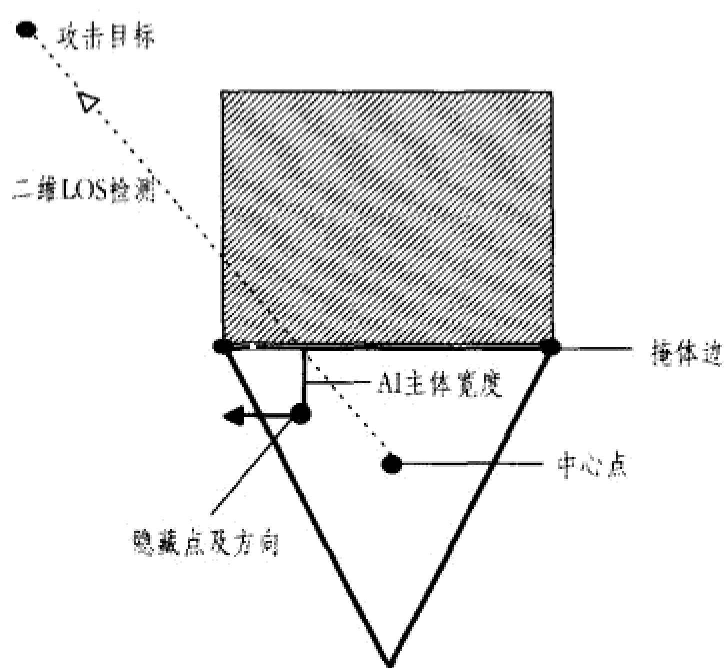


图 3.1.2 在找到的三角形中确定最后的隐蔽点

现在，以掩体墙为准，以 AI 主体的宽度为偏移量，可以得到一个新的隐蔽点。而且，还可以知道 AI 主体准备跳出掩体时的行进方向。根据掩体边线的特征位（是直立掩体，还是蹲伏掩体)，可以将隐蔽点调整到一个正确的高度，告诉 AI 主体是蹲下，还是站着。

掩体检测偶尔会出现一些错报的结果，因为它可能会把一个相邻的墙体标识为一个掩体。为了避免这个错误，可以检测与当前掩体的边线相连的三角形，看它们是否与当前这个掩体的边线共用一个顶点。如果共用，就说明这个边是沿着墙的。错报的情况还与攻击目标的位置有关。如果有两个三角形，它们的边组成一个墙体。那么当攻击目标的位置在这两个三角形中心点的对面时，肯定会产生一个错报。

在实际应用中，由于这个问题很明显是对玩家透明的（当 NPC 躲在一个稍大的墙体后面时，它们会认为自己只要稍微移动一下就能看到玩家)，因此在 PS2 上，为每个节点花费额外的内存检索开销来解决这个问题似乎不太值得。而且，即使有这样的错报现象，掩体检测也不会出什么问题（最多是找到一个掩体的位置，但实际上对攻击目标来说却是完全暴露的，可以一枪将 NPC 击中)。

3.1.4 在掩体间行进

既然可以找到隐蔽点，为了设法一直躲在掩体后面，就也可以在两个隐蔽点之间生成行

进路径。这只是意味着我们是一个节点一个节点地进行正常的搜索。但当碰到一个可作为掩体的节点时，就会把这个节点的权值放得非常低（这样，搜索过程会优先考虑这个节点，而不是那些离得更近的节点）。

然后，在路径平滑处理中，要确保那些掩体的位置，不会在瞄准线（LOS）测试中被排除在路径之外，如图 3.1.3 所示。

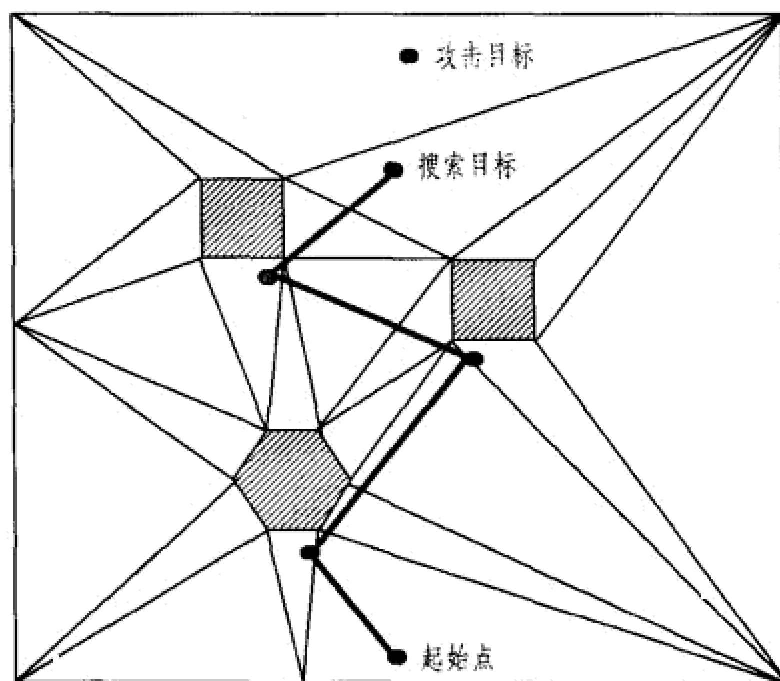


图 3.1.3 平滑路径，其中隐蔽点被保留下来作为路点（waypoint）

3.1.5 团队掩护行为

有时可能会希望若干个 NPC 作为一个团队，配合使用掩体位置，共同对付玩家。举个例子，NPC 队友之间提供火力掩护，或者绕到对面的隐蔽点，来包抄攻击玩家。为了实现这个功能，需要对搜索算法进行微小的改动。我们不再是寻找一个特定的目标，而是寻找一系列搜索目标的位置。虽然本文的搜索算法并非是针对这种情况设计的，但是，可以很容易地在开放目标搜索中融入这个功能。

由于需要对某个给定的点，找到它周围所有的隐蔽点，因此搜索算法本身就变成了深度优先搜索。虽然这样就不能用估价函数来简化搜索过程，但是可以使用同样的架构去找到想要的结果。

和前面一样，估价函数的工作还是根据一个给定的点来检测找到相关的掩体。但现在，它还要保存合法的隐蔽点。搜索函数负责检测附近的节点，估价函数负责记录可用的节点。只有当估价函数已经记录下既定数量的隐蔽点时，目标（goal）函数才会返回 true。

一个团队的 NPC 可以共享使用这些隐蔽点，向玩家逼近，并且可以为队友提供火力掩护。一旦找到了附近的那些隐蔽点，一个 NPC 可以向玩家射击，同时其他 NPC 就会分散到各个不同的掩体中。然后，负责掩护的 NPC 就可以跑向掩体了（假设它还活着），同时其他的 NPC 可以跳出来，为它提供火力掩护。

如果某个 NPC 正在搜索刚刚跟丢的一个玩家，搜索算法还可以提供一个可能的藏身地点

的列表。如果某个三角形的一个边挡住了 NPC 的瞄准线（LOS），那么它就可能是一个藏身地点。可以把附近所有这样的三角形都找出来。然后，NPC 就可以巡视它周围的环境，排查所有的障碍物，来搜寻那个跟丢的玩家。

3.1.6 其他功能

我们或许希望能够再添加一些其他的功能，本节将介绍几个这样的功能。

1. 保存隐蔽点

如果寻路系统考虑了简单的路径节点保存（防止其他路径搜索时再重复考虑这些节点），那么 NPC 也可以保存它们的隐蔽点。这样，其他的 NPC 就不会在它们的路径搜索中考虑这些隐蔽点了。这样可以防止很多 NPC 都集中在同一个隐蔽点。

2. 更多的数据

将碰撞信息嵌入到导航网格中已经变得没那么重要了。虽然掩体是自动侦测到的，但是游戏策划仍然可以对特定区域进行设置，使之可以或者不可以用于掩体搜索。游戏策划只要使用 3D 工具，把这些区域的值手工画在导航网格上，就可以实现这个功能。我们还可以自动检查其他方面的寻路事宜，例如从悬崖往下跳，其方法就是检测到悬崖的边缘，这些边缘就是有一定距离间隔、高度相同的一些三角形。

3. 其他搜索

我们可以很容易地将这个系统进行扩展，增加其他一些搜索项目。对于那些要逃离玩家的 NPC，可以简单地强制它们向与玩家相反的方向逃跑，以制造出 NPC 在慌乱之中一头撞到墙上的赝像（artifact）。利用开放目标式搜索，找到一个在玩家特定半径范围之外的点，可以为 NPC 提供一个确信无疑的安全路径，使之不会受到玩家的伤害（虽然在寻路过程结束之前，它就得按照给定的方向开始移动了）。另外，如果想让 NPC 在逃跑时不被干掉，可以把这个路径搜索的权值放低一些，故意地让 NPC 跑向掩体的位置。

3.1.7 总结

如果一款游戏的大量情节都是在狭小、紧凑的场所里展开枪战，那么我们仍然需要使用手工掩体放置系统，让策划人员可以控制掩体的位置。大型游戏则可以考虑使用战术战斗（tactical combat），不需要在制作过程中做上述这些工作。本文描述的技术是相当简单的，但是结果却非常有意义，同样也可以帮助游戏策划省掉上述的工作流。另外，这样的一些方法可以让行为从系统中体现出来，因为 NPC 在各种情况下都可以找到可用的掩体。

在此要感谢 Tinman、Stan Jang、Marcin Chady、Ben Geisler 和 Adrian Johnston，感谢他们的参与和所做的工作。

PDG

3.1.8 参考文献

[Snook00] Snook, Greg. “Simplified 3D Movement and Pathfinding Using Navigation Meshes.” In *Game Programming Gems*. Charles River Media, 2000.

[Tozour02] Tozour, Paul. “Building a Near Optimal Navigation Mesh.” In *AI Game Programming Wisdom*. Charles River Media, 2002.

[White02] White, Stephen and Christopher Christensen. “A Fast Approach to Navigation Meshes.” In *Game Programming Gems 3*. Charles River Media, 2002.

3.2 使用人工势场实现快速目标评级

Factor 5 公司，Markus Breyer
me@markusbreyer.com

玩家可以使用“autotarget”功能，进行自动目标选定。但是 AI 主体经常需要自己在 3D 场景中从一大堆人为安置的游戏对象中进行目标的选择。是去攻击敌人，还是去拾取某些道具？是选择一个 AI 主体，开始交谈；还是去寻找一个最理想的地方，准备接管它？再或者是去挑选最佳的着陆点？在游戏中，目标的选择无处不在。

在大部分的目标选择算法中，道具的评级是根据距离和角度来进行的：优先拾取近处的物品，而不是远处的物品；正前方的物品，其优先级高于侧面或后面的物品。这是因为，观察者必须花时间转身，然后才能与身后的物体发生交互，这就使得该目标有点儿不太可取。但是，如果身后的那个目标距离更近，它就比前方的目标更为可取。很明显，距离和角度是两个指标，必须把它们组合成一个单一的衡量标准，以便做出合理的决策。本文提供的公式使用了简单的、计算量不大的有理函数(rational function)，来计算一个三维的伪势场，用平滑和非常自然的方式将距离与角度这类平衡问题具体化，最后生成一个数量值。使用这个数量值可以对目标进行优先级排序。

3.2.1 基本思想

我们的目的是为人造场景中的所有实体取得一个量化指标，用这个量化指标对它们进行优先度评级。我们要找到一个可以产生这个数量值的数学函数，处理目标的位置与方向问题，以及观察者的位置与方向问题，并返回一个优先度评级（该算法实现会将比较低的值解释为较高的优先级，但这只是一种实现的选择而已）。

解决该问题有两个方法。一般情况下，点（point）都是通过距离（或距离的平方）来评级的。当把距离解释为一个“势”时，就可以画出一个二维等高线图（即一个等值线图，将“势”相同的点用线连起来），如图 3.2.1 所示。

如果只是基于距离来进行评级，那么在观察者后面的目标，以及在观察者前面的目标，都会同等对待。如果比较倾向于观察者前面的目标，那么我们的选择可以参考目标与观察者的视轴（或者说是局部 z 轴）之间的夹角。基于角度的势场等高线图请参见图 3.2.2。

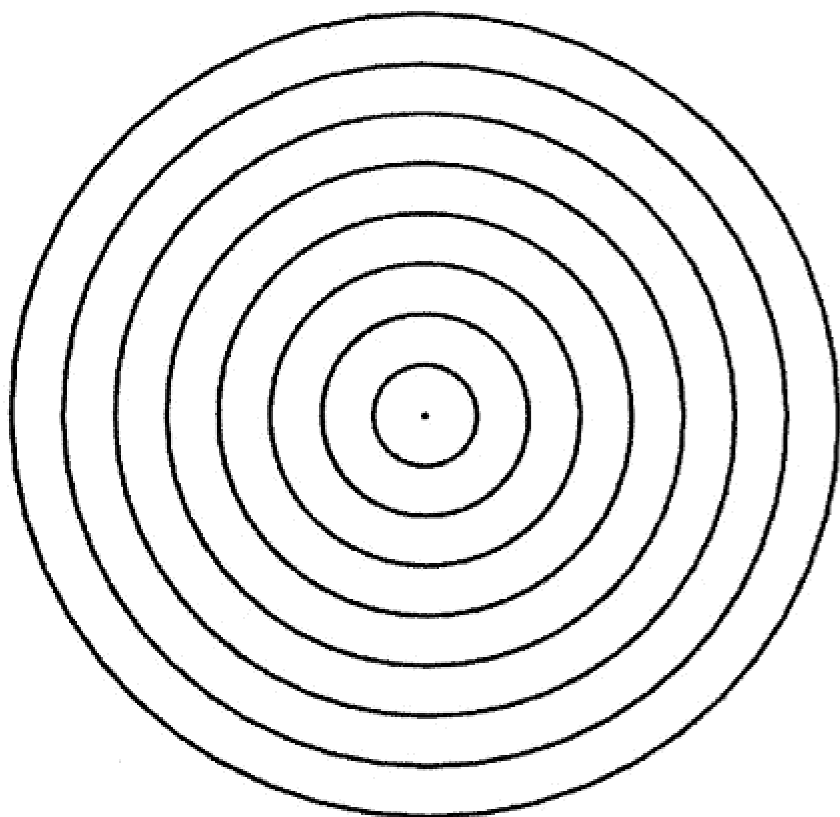


图 3.2.1 仅基于距离的一个势场。有相等势值的点以观察者为中心，形成同心圆（或者在三维环境中，就是形成若干个同心球体）

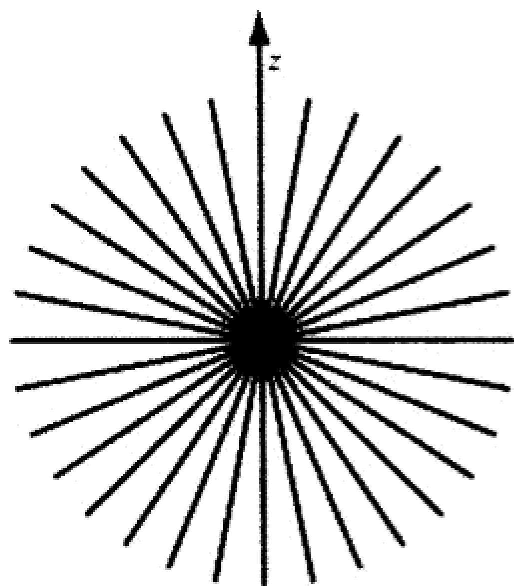


图 3.2.2 仅基于角度的一个势场。势值相等的点形成一对源自观察者的 V 形射线（对于三维的情况，形成的就是圆锥体）

如果综合考虑距离和角度来选择目标，应该同时比较距离和角度，并把它们的势场混合起来，获得一个惟一的结果。还有一种方法，我们也可以从另一方面解决这个问题：设想在二维空间中的若干条等值线将所有优先级相同的点连接起来，如图 3.2.3 所示。

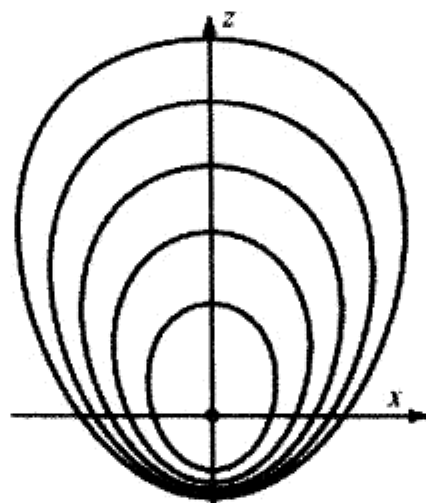


图 3.2.3 基于距离和角度的势场

通过这种方法，一个在远处（但却在正前方）的目标与那些位于侧面和后面（但却很近）的目标，有着同样的优先等级。这就是说，近处的目标比远处的目标优先级高，正前方的目标比侧面的目标优先级高。在下面的内容中，将向大家介绍一个方法，推导出一个可以产生上述势场的数学函数。

3.2.2 公式

一个满足这些需求的数学函数，在观察者的位置上应该有一个全域极限值（global extremum）。而整个域中，这个函数应该是单调的（monotonous）、横向对称的（laterally symmetric）。对于一条势场曲线上的若干个选定的点，我们要设定几个常量值。如果选择正确的函数，其他的点就会依序排列入位。

从头开始，先来分别看看这 3 个坐标方向。在 z 轴方向上，在观察者身后，要让这个函数的曲线比较陡。在观察者的位置上达到最小值，然后再缓缓上升。在 x 轴和 y 轴方向上，这个函数应该是对称的，并在 z 轴上（$x=y=0$）达到最小值。对单独每个坐标轴而言，满足这些要求的函数是：

$$v_z = \frac{az^2 + bz + c}{pz + q};\ v_x = dx^2 + e;\ v_y = fy^2 + g \quad (3.2.1)$$

将这些项相加、消去、合并，并重新命名各个系数，就得到下面这个公式：

$$\begin{aligned} v(x,y,z) = v_x + v_y + v_z &= \frac{z^2 + (ax^2 + by^2 + e)z + cx^2 + dy^2 + f}{pz + q} \\ &= \frac{z^2 + ax^2 z + by^2 z + cx^2 + dy^2 + ez + f}{pz + q} \end{aligned} \quad (3.2.2)$$

现在，需要找到8个边界条件，这样就可以解出上面公式中的8个系数。

假设在观察者的位置上，势值（v）的值最小，为0（也就是说，v 的值越小，这个目标的优先级越高，越适合拾取）。然后，令 v=1 的等值线（三维空间中就是等值面）通过空间中的某些特定的点[1]。

v=1 的等值线围起来的等值面泡状体的范围可以用5个直观的参数来定义，如图3.2.4所示。

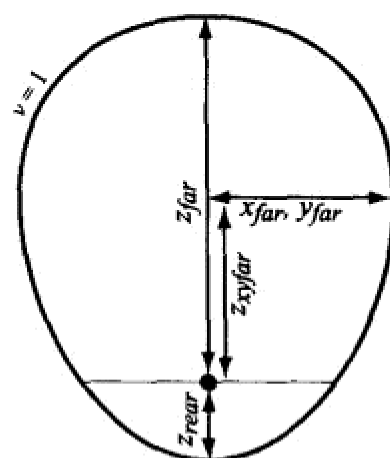


图3.2.4 等值面泡状体由5个参数来定义：z_{far}、x_{far}、y_{far}、zxy_{far} 和 z_{rear}

改变这些参数（z_{far}、x_{far}、y_{far}、zxy_{far} 和 z_{rear}）可以让我们按照自己的需求来改变泡状体的形状。可以故意让它水平和垂直方面的范围有所区别，这样就可以定义一个选择器（selector），使它要么对水平方向比较敏感，要么对垂直方向比较敏感。经验显示，游戏策划通常可以随时定义这些参数，得到令人满意的结果。这个结果顶多只需要做一些微小的改动，或者根本不需要后期的修改。

最大横向范围是 $z = zxy_{far}$，结合前面的要求：点（0,0,0）处的势值最小，等于0，这样就可以确定下列8个边界条件：

$$v(0,0,z_{far}) = 1 \qquad v(0,0,0) = 0$$

$$v(0,0,-z_{rear}) = 1 \qquad \frac{\partial v}{\partial z}(0,0,0) = 0$$

$$v(x_{far},0,z_{xyfar}) = 1 \qquad \frac{\partial v}{\partial z}(x_{far},0,z_{xyfar}) = 0$$

$$v(0,y_{far},z_{xyfar}) = 1 \qquad \frac{\partial v}{\partial z}(0,y_{far},z_{xyfar}) = 0$$

求解8个系数，得到：

$$p = z_{far} - z_{rear} \qquad a = \frac{p - 2z_{xyfar}}{x_{far}^2} \qquad c = \frac{q + z_{xyfar}^2}{x_{far}^2} \qquad e=0$$

[1] 这里人为地将 v 的值设为1，因为我们需要某个非零值的等值线勾勒出势场的形状。

$$q = z_{far} z_{rear} \qquad b = \frac{p - 2z_{xyfar}}{y_{far}^2} \qquad d = \frac{q + z_{xyfar}^2}{y_{far}^2} \qquad f=0$$

现在，就可以用 z_{far}、x_{far}、y_{far}、zxy_{far} 和 z_{rear} 这 5 个参数来定义 v 了。由于系数 e 和 f 都等于零，被去掉了，势值函数就变成这个样子：

$$v = \frac{z^2 + (az + c)x^2 + (bz + d)y^2}{pz + q} \tag{3.2.3}$$

3.2.3 势值函数的评估

在评估这个势值函数时，一定要注意三件事情，确保这个函数能够产生我们想要的结果：

1. 分母一定不能为零，或者是负数。否则，不但会让势值变得无穷大，还会让势场变成非连续的、非单调的。

2. （$az+c$）和（$bz+d$）这两项不能为负数。如果它们变成负数，最终得出的势值就会落在某个给定的 z 值的后面。这样会产生不一致的拾取行为。也就是说虽然当前目标的 v 值很低（比较高的优先级），但是相应的拾取行为却比较差（也就是说，拾取这个目标不是个好选择）。

3. 边界等值线 $v=1$，必须放在选择范围的最外边。也就是说，这是目标选择器最大的选择范围。只有小于或等于 1 的值（$v \leq 1$）才可以用来进行正确的评级。如果某个值大于 1（$v > 1$），就可以确定这个目标已经超出了选择范围。但是，大于 1 的值不能用来进行评级。因为当 v 的值大于 1 后，势值函数已经开始丧失原先定义的特性了。

考虑到这些因素，评估工作应该这样进行：

$$R = \max(az + c, 0); \tag{3.2.4}$$

$$S = \max(bz + d, 0); \tag{3.2.5}$$

$$D = \max(pz + q, eps); \tag{3.2.6}$$

$$v = \frac{z^2 + Rx^2 + Sy^2}{D} \tag{3.2.7}$$

在这里，eps 是某个非常小的正数，例如 0.001。这是一个计算量非常小的公式，可以轻松地在包含大量可选目标的每一帧中执行。

3.2.4 可视化

为了便于开发和调试，通过一些图形化的调试标志（盒体、球体等），游戏引擎通常都能够可视化其内部运作。同样地，你会发现通过一个或更多的等值面（特别是 $v=1$ 的那个等值面）可视化生成的势场，是非常有用的。

即使这个势场是通过一个隐式公式来定义的，也可以派生出一个显式公式，来描述 $v = v_c$ 的等值面。这个等值面，由一系列垂直于 z 轴的椭圆组成，其参数化表示是：视轴方向上的距离 z，与观察者的视轴之间的角度φ。使用上面定义的常量 R、S 和 D，每个值为 z 的椭圆的主轴半径可以这样计算出来：

PDG

$$R_x = \sqrt{\frac{Dv_c - z^2}{R}} \text{ 和 } R_y = \sqrt{\frac{Dv_c - z^2}{S}} \tag{3.2.8}$$

然后渲染显示点云（point cloud）（$z = z_{min}...z_{max}$，$\varphi = -\pi..\pi$）：

$$P = \begin{pmatrix} R_x \sin\varphi \\ R_y \cos\varphi \\ z \end{pmatrix} \tag{3.2.9}$$

这里的 z_{min} 和 z_{max} 可以通过求解方程式 $Dv_c - z^2 = 0$ 得到。

在 $z_{far} > z_{rear}$ 的情况下，求解 z_{min} 和 z_{max} 的最佳方法是定义 $S = v_c p$ 和 $T = v_c q$，然后再通过下列公式来计算 z_{min} 和 z_{max}：

$$z_{max} = \frac{1}{2}\left[S + \sqrt{S^2 + 4T}\right]$$
$$z_{min} = -T / z_{max} \tag{3.2.10}$$

这里要注意，当 $v_c = 1$ 时，z_{min} 和 z_{max} 其实就是$-z_{rear}$ 和 z_{far}。所以，如果不需要画出 v_c 其他数值的表面，那也就不需要上面的公式了。

如果愿意，得到的点云可以很容易地镶嵌成小方格。请一定记住，v_c 的值必须介于 0 和 1 之间，这样才能得到有用的结果（但是，也可能有其他的应用会使用超出这个范围的 v_c 值）。图 3.2.5 是一个可视的势场。

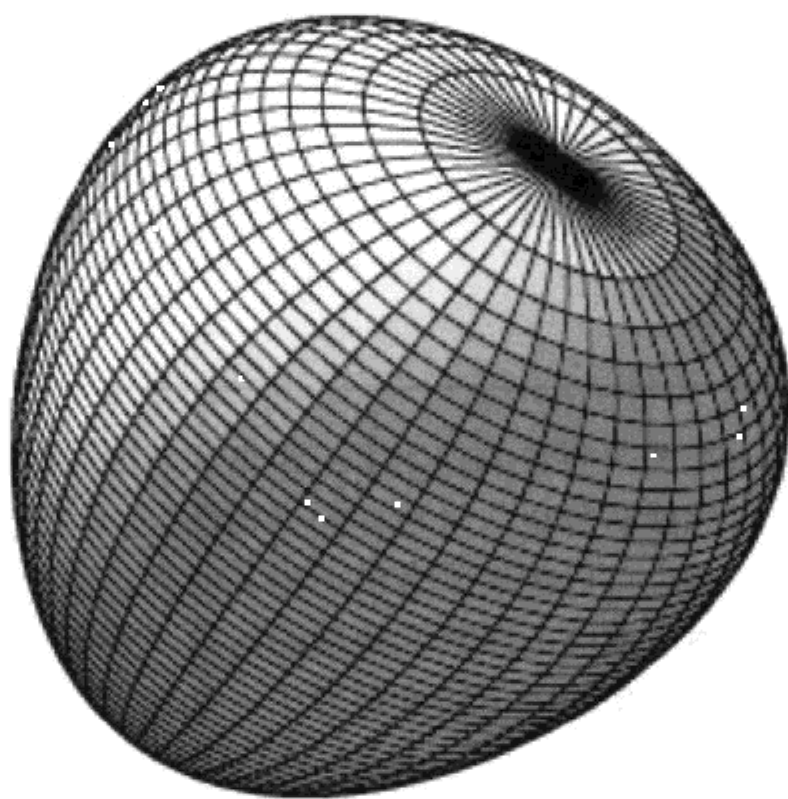


图 3.2.5 一个可视的三维势场

3.2.5 方向场的应用

这个势值函数还可以有更多其他的应用，例如：破坏场、推力场和吸力场。这里需要定义一个场强：$H = \max(1 - v, 0)$。然后，可以把 H 或者 H^2 作为一个火焰喷射器的破坏场，或者作为推力场或吸力场的强度值。对于方向，可以使用归一化梯度向量（如果需要，也可以取

反）。为了计算这个梯度向量，要定义两个新的常量：

$$\begin{aligned} M &= aq - cp \\ N &= bq - dp \end{aligned} \tag{3.2.11}$$

然后，利用这两个常量，就可以计算出 v 的梯度：

$$\nabla v = \begin{pmatrix} \partial v / \partial x \\ \partial v / \partial y \\ \partial v / \partial z \end{pmatrix} = \begin{pmatrix} 2xR / D \\ 2yS / D \\ [Mx^2 + Ny^2 + (pz + 2q)z]/D^2 \end{pmatrix} \tag{3.2.12}$$

这样的一个吸力场（在二维空间中）大致如图 3.2.6 所示：

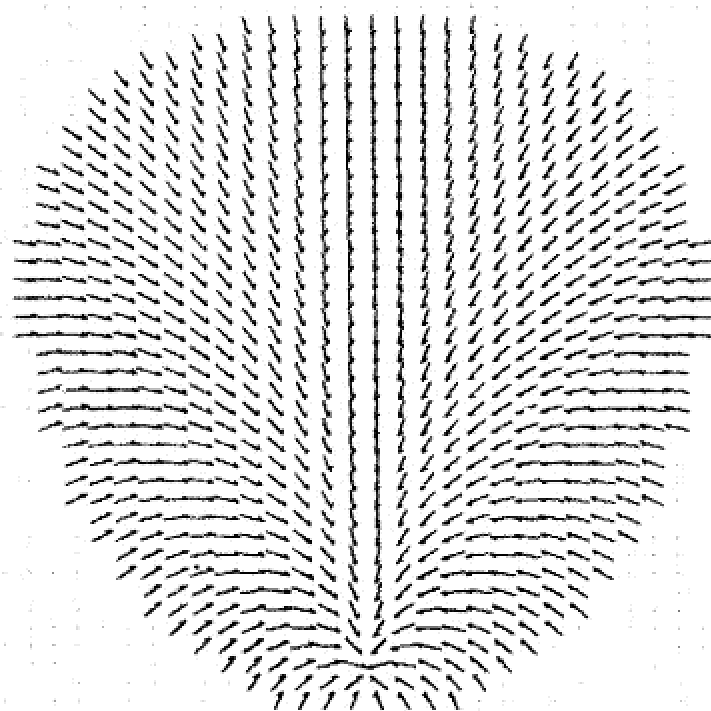


图 3.2.6 一个二维“吸力”的势场

3.2.6 多维扩展

本文所讨论的应用集中在两个参数的使用上：距离和角度。要想把它扩展成更多维的应用，也不是很困难，如“目标威胁等级”、“目标的易损性”，或者其他很多类似的应用实例。对于带有等值面的势场，当然也可以扩展到多维应用。但是，本文推导出的这个有理函数是非常特殊的，只针对三维观察者面临的距离和角度的问题，并不一定可以扩展成一个通用的势场表示方法。但是，读者可以使用类似的推理方法，推导出其他专门针对不同边界条件的势场函数。

举个例子，对于一个需要在价值（value）与威胁（threat）之间进行平衡的情况，可以找到一个貌似抛物线的势场。方法就是，即使是一个威胁等级很低的目标，也要让它具有一个最小值的价值，使它值得被攻击。而价值应该与威胁成正比。而对于那些威胁等级很高的目标，它的价值参数也要快速地增加。这样做的结果大致就是：v = 1/(a*value − b*threat^2)。同样地，这里的分母也必须是某个很小的正数值。然后，这样的几个势场函数可以进行合并，例如简单地把它们相加，结果就是一个抛物线模样的图形。这也同样可以用一个简单的势场函数来表示。而且，这个方法也许还有更广阔的应用前景。大家来实验吧！

3.2.7 总结

本文介绍了一个简单的、计算量很小的方法，实现了一个人工势场。利用这个势场，我们从距离和角度两个方面，自然、平滑地解决了目标对象评级的问题。这里介绍了一个简单的方法，可以把这样一个势场的等值面渲染显示出来。还介绍了另外一个方法，来计算某个场的梯度，作为吸力、推力和破坏力的方向矢量。对于本文中所有的公式，随书光盘中提供了它们的 C++实现代码。

3.3 利用 Lanchester 损耗模型来预测战斗结果

Page 44 Studios 有限责任公司，John Bolton
johnjbolton@yahoo.com

早在第一次世界大战之前，F.W.Lanchester（兰彻斯特）就正式提出了一套基本的数学模型，以部队作战单位的数量，以及这些单位被摧毁的速率，来描述一场战斗。这些模型就是著名的 Lanchester 损耗模型（Lanchester Attrition Model），它已经成为对战争进行数学分析的理论基石。本文向大家介绍了如何利用这些模型快速、高效地预测游戏中的战斗结果，其内容包括：Lanchester 损耗模型的基本原理回顾，介绍几个用来评估作战系统和战斗场景的模型。

3.3.1 概述

通过几个微分方程的使用，Lanchester 损耗模型描述了在一场战斗中，作战单位损耗的速度。求解这个微分方程系统，就可以回答下列一些问题：

- 谁赢了？
- 战斗持续了多长时间？
- 某时某刻作战的某方所拥有的作战单位是多少？
- 战斗结束时，获胜一方剩下的作战单位是多少？

这些微分方程取决于战斗的类型、作战条件，以及作战系统。对于所展示的这些模型，当某方所有的作战单位都被摧毁时，战斗就宣告结束。一般来说，我们假定战斗是持续不间断的，同时发生的，而且双方所有的作战单位都是一模一样的。

接下来的内容是几个在游戏中经常发生的典型的场景（或者说是作战系统）。对于每一个场景，都给出了相应的评估模型及解决方案。在所有这些场景中，都只有两方军队：蓝军和红军。在战斗中，双方军队中作战单位的数量由变量 B 和 R 表示。这两个变量的初始值，也就是双方军队中作战单位的初始数量，由常量 B_0 和 R_0 表示。双方军队中所有的作战单位都有一个战斗力（*combat rating*），用常量 c_B 和 c_R 表示。“战斗力”通常表示的是一个作战单位摧毁敌方作战单位的速率。这个常量可以包括一些因子，例如天气和地形、防御能力和攻击能力。在下面的场景讨论中，将会详细解释战斗力这个概念。

PDG

3.3.2 场景 1：全体混战

在这个场景中，由 1 000 个魔兽组成的蓝军与由 200 个人组成的红军在广阔的平原上展开交战。人类的军队装备精良，训练有素，每个人每分钟可以干掉 10 个魔兽。而魔兽则动作迟钝，蠢笨透顶，每个魔兽每分钟只能杀死一个人。这场战斗的结果会如何呢？在这个场景中，所有的作战单位都不停地作战。作战单位被摧毁的速率，就是一方军队中作战单位的数量，乘以对方军队中作战单位摧毁敌方作战单位的速率。下列公式系统描述了这个场景：

$$\begin{aligned} dB &= -c_R R dt \\ dR &= -c_B B dt \end{aligned} \qquad (3.3.1)$$

在这个例子中，B_0 等于 1 000，R_0 等于 200，c_B 等于 1，而 c_R 等于 10。

1. 谁会赢呢

为了能够判断出哪方会胜出，我们将上面的公式合并如下：

$$\frac{dB}{dR} = \frac{-c_R R dt}{-c_B B dt} = \frac{c_R R dt}{c_B B dt} \qquad (3.3.2)$$

这是一个可分离一阶微分方程式。根据给定的初始条件：当 $R = R_0$ 时，$B = B_0$，求解如下：

$$\begin{aligned} c_B\left(B^2 - B_0^2\right) &= c_R\left(R^2 - R_0^2\right) \\ B^2 &= \frac{c_R}{c_B}\left(R^2 - R_0^2\right) + B_0^2 \end{aligned} \qquad (3.3.3)$$

根据定义，当 R=0 时，如果 B>0，则蓝军获胜，那就可以得到下列结果（请注意，如果 B>0，那么 B_2 也大于 0）：

$$\begin{aligned} 0 &< \frac{c_R}{c_B}\left(0 - R_0^2\right) + B_0^2 \\ \therefore \frac{c_R}{c_B} &< \frac{B_0^2}{R_0^2} \end{aligned} \qquad (3.3.4)$$

在这个场景中，人类战败了，如下所示：

$$\frac{c_R}{c_B} = \frac{10}{1} = 10, \quad \frac{B_0^2}{R_0^2} = \frac{1000^2}{200^2} = 25, \quad 10 < 25 \qquad (3.3.5)$$

这是一个很重要的结果。它显示出，对于开阔地的混战，虽然战斗的结果依赖于单个作战单位的战斗力（combat rating），但同时也取决于每支军队中作战单位的初始数量的平方。也就是说，蓝军只需要 $\sqrt{2}$ 倍数量的作战单位，就可以击败单位作战能力是它 2 倍的红军。这个模型就是 Lanchester 的平方法则（*Square Law*）。

2. 在时间 t 的时候，还剩下多少个作战单位

在某个特定的时间，剩余作战单位的数量可以用下列公式计算出来。

$$\begin{aligned} B &= B_0\cosh(\sqrt{c_R c_B}\,t) - R_0\sqrt{\frac{c_R}{c_B}}\sinh(\sqrt{c_R c_B}\,t) \\ R &= R_0\cosh(\sqrt{c_R c_B}\,t) - B_0\sqrt{\frac{c_B}{c_R}}\sinh(\sqrt{c_R c_B}\,t) \end{aligned} \quad (3.3.6)$$

尽管人类的军队有超强的能力，但在作战单位的数量上远远小于敌方，所以人员损失很快。仅仅过了 3 秒钟（0.05 分钟），人类几乎损失了四分之一的军队，而魔兽则还有很多。

$$\begin{aligned} B &= 1000\cdot\cosh(\sqrt{10\cdot 1}\cdot .05) - 200\cdot\sqrt{\frac{10}{1}}\cdot\sinh(\sqrt{10\cdot 1}\cdot .05) &= 912 \\ R &= 200\cdot\cosh(\sqrt{10\cdot 1}\cdot .05) - 1000\cdot\sqrt{\frac{1}{10}}\cdot\sinh(\sqrt{10\cdot 1}\cdot .05) &= 152 \end{aligned} \quad (3.3.7)$$

3. 一方失败后，获胜方还剩下多少作战单位

战斗结束后，获胜方剩下的作战单位数量可以从下列公式计算得出。计算结果只对获胜一方有效：

$$\begin{aligned} B_{R=0} &= \sqrt{B_0^2 - \frac{c_R}{c_B}R_0^2} \\ R_{B=0} &= \sqrt{R_0^2 - \frac{c_B}{c_R}B_0^2} \end{aligned} \quad (3.3.8)$$

当战斗结束时，蓝军还剩下 775 个魔兽，可以去挑战别的军队了。

$$B_{R=0} = \sqrt{1000^2 - \frac{10}{1}\times 200^2} = 775 \quad (3.3.9)$$

4. 战斗持续了多长时间

战败一方的所有作战单位被全部摧毁所用的时间，可以使用下列公式中的一个来计算。计算的结果只对获胜一方有效。还有很重要的一点要注意，如果双方军队各项指标完全一样，那么战斗将一直进行下去。

$$\begin{aligned} t_{R=0} &= \frac{\tanh^{-1}\left(\dfrac{R_0}{B_0}\sqrt{\dfrac{c_R}{c_B}}\right)}{\sqrt{c_R c_B}} \\ t_{B=0} &= \frac{\tanh^{-1}\left(\dfrac{B_0}{R_0}\sqrt{\dfrac{c_B}{c_R}}\right)}{\sqrt{c_B c_R}} \end{aligned} \quad (3.3.10)$$

在这个场景中，在 3 秒钟之内，如果有近四分之一的红军被歼灭，而蓝军损失很小，那么可以预计，歼灭余下的红军所用的时间也会很快。战斗实际持续的时间是 0.236 分或 14 秒。

$$t_{R=0} = \frac{\tanh^{-1}\left(\dfrac{200}{1000}\cdot\sqrt{\dfrac{10}{1}}\right)}{\sqrt{10\cdot 1}} = .236 \quad (3.3.11)$$

3.3.3　场景 2：狭窄的石阶

在这个场景中，在一个悬崖峭壁的石阶上（而不是开阔的平原上），人类与魔兽遭遇了。尽管双方的兵力规模都不小（200 个人对抗 1 000 个魔兽），但是由于空间狭小，在同一时刻，只允许 2 个作战单位交战。这个场景与前面那个场景的区别就是，只有固定数量的作战单位（而不是双方所有的作战单位）在战斗。所以，作战单位损失的速率是与正在战斗的作战单位成比例的，而不是与作战单位的总和成比例。下面的公式系统描述了这个场景。这个模型假设双方当前参加战斗的人员数量相等。但是，也可以很容易地对这些公式做一些改造，用来描述当前双方参战人员不等的情况：

$$\begin{aligned} dB &= -c_R n dt \\ dR &= -c_B n dt \end{aligned} \qquad (3.3.12)$$

同样，在这里，B_0 等于 1 000，R_0 等于 200，而 c_B 等于 1，c_R 等于 10。在这个场景中，n 等于 2。

1．谁会赢呢

当 $R=0$ 时，如果 $B>0$，则蓝军获胜，也就是满足下列条件：

$$\frac{c_R}{c_B} < \frac{B_0}{R_0} \qquad (3.3.13)$$

在这个场景中，人类扭转了局面，蓝军失败了：

$$\frac{c_R}{c_B} = \frac{10}{1}, \quad \frac{B_0}{R_0} = \frac{1000}{200} = 5, \quad 10 > 5 \qquad (3.3.14)$$

这也是一个很重要的结果。它说明，与前面的场景相比，在这个场景中，拥有超强能力作战单位的军队更有机会胜出。这就是 Lanchester 的线性法则（*Linear Law*）。

2．在时间 t 的时候，还剩下多少个作战单位

在某个特定的时间，剩余作战单位的数量可以用下列公式计算出来。

$$\begin{aligned} B &= B_0 - c_R n t \\ R &= R_0 - c_B n t \end{aligned} \qquad (3.3.15)$$

在战斗开始 30 分钟以后，双方军队都损失了很多兵力，但是人类还是占优（人类剩下 70% 的兵力，而魔兽只剩下 40%）：

$$\begin{aligned} B &= 1000 - 10 \cdot 2 \cdot 30 = 400 \\ R &= 200 - 1 \cdot 2 \cdot 30 = 140 \end{aligned} \qquad (3.3.16)$$

3．一方失败后，获胜方还剩下多少作战单位

战斗结束后，获胜方剩下的作战单位数量可以从下列公式计算得出。计算结果只对获胜一方有效：

$$\begin{aligned} B_{R=0} &= B_0 - \frac{c_R}{c_B} R_0 \\ R_{B=0} &= R_0 - \frac{c_B}{c_R} B_0 \end{aligned} \qquad (3.3.17)$$

PDG

在这个场景中，虽然红军也有一半的兵力损失，但还是赢得了这场战斗：

$$R_{B=0} = 200-\frac{1}{10}\times1000 = 100 \quad (3.3.18)$$

4. 战斗持续了多长时间

战败一方的所有作战单位被全部摧毁所用的时间，可以使用下列公式中的一个来计算。计算的结果只对获胜一方有效。

$$\begin{aligned} t_{R=0} &= \frac{R_0}{nc_B} \\ t_{B=0} &= \frac{B_0}{nc_R} \end{aligned} \quad (3.3.19)$$

在这个例子中，红军只用了 50 分钟就取得了最后的胜利：

$$t_{B=0} = \frac{1000}{2\times10} = 50 \quad (3.3.20)$$

3.3.4 场景 3：炮战

假设有一款类似《Battleship》的游戏，对战双方都不知道对方作战单位的具体位置。每支军队有 100 个作战单位，放在 100×100 的格子中。而单个作战单位可以在格子中随时随意地移动。一个作战单位可以发射一发炮弹，打到敌方的格子中，击中其中的某个点。如果该点上正好有一个敌方的作战单位，那么它就会被摧毁。一个作战单位射击的速率由一个外部因子决定。所有作战单位的射击速率都是一样的。对于这个场景，蓝军每秒可以射击 2 次，红军每秒可射击 1 次。这个场景的战斗结果取决于概率。因此，无法计算出准确的结果，但是可以预测一些我们想要的结果。

在这个场景中，一方兵力损失的速率与敌方兵力的总数和敌方的作战能力，以及自身作战单位的分布密度成比例。如果 A_x 是军队 X 占据的区域，它的作战单位分布密度为 ρ_x，那么 ρ_x 可以用下列公式表示：

$$\rho_B = \frac{B}{A_B},\quad \rho_R = \frac{R}{A_R} \quad (3.3.21)$$

在这个例子中，A_B 和 A_R 都是 10 000：

$$\rho_B = \frac{B}{10000},\quad \rho_R = \frac{R}{10000} \quad (3.3.22)$$

下列公式系统描述了这个场景：

$$\begin{aligned} dB &= -c_R R \rho_B \, dt \\ dR &= -c_B B \rho_R \, dt \end{aligned} \quad (3.3.23)$$

在这个场景中，B_0 等于 100，R_0 等于 100，c_B 等于 2，c_R 等于 1。

假设一支军队占据的区域大小是个常数，将区域（A_x）与这个常数项合并，就可以简化上述公式。这里引入了一个调整过的战斗力常量：

$$\chi_R = \frac{c_R}{A_B}, \quad \chi_B = \frac{c_B}{A_R} \tag{3.3.24}$$

在这个例子中：

$$\chi_R = \frac{1}{10000} = 0.0001, \quad \chi_B = \frac{2}{10000} = 0.0002 \tag{3.3.25}$$

简化后的模型如下：

$$\begin{aligned} dB &= -\chi_R RB dt \\ dR &= -\chi_B BR dt \end{aligned} \tag{3.3.26}$$

1. 谁会赢呢

当 $R=0$ 时，如果 $B>0$，则蓝军获胜，也就是当下列条件成立时：

$$\frac{\chi_R}{\chi_B} < \frac{B_0}{R_0} \tag{3.3.27}$$

在这个场景中，蓝军的射击速率是红军的 2 倍，所以蓝军轻易获胜：

$$\frac{\chi_R}{\chi_B} = \frac{0.0001}{0.0002} = 0.5, \quad \frac{B_0}{R_0} = \frac{100}{100} = 1, \quad 0.5 < 1 \tag{3.3.28}$$

2. 在时间 t 的时候，还剩下多少个作战单位

在某个特定的时间，剩余作战单位的数量可以用下列公式计算出来：

$$\begin{aligned} B &= B_0 \frac{\chi_B B_0 - \chi_R R_0}{\chi_B B_0 - \chi_R R_0 \exp\left(\left(\chi_R R_0 - \chi_B B_0\right)t\right)} \\ R &= R_0 \frac{\chi_R R_0 - \chi_B B_0}{\chi_R R_0 - \chi_B B_0 \exp\left(\left(\chi_B B_0 - \chi_R R_0\right)t\right)} \end{aligned} \tag{3.3.29}$$

在这个场景中，开战 100 秒后，双方剩余的作战单位可以计算出来。局势似乎对红军很不利。

$$\begin{aligned} \chi_B B_0 &= 0.0002 \cdot 100 = 0.02 \\ \chi_R R_0 &= 0.0001 \cdot 100 = 0.01 \\ B &= 100 \cdot \frac{0.02 - 0.01}{0.02 - 0.01 \cdot \exp\left(\left(0.01 - 0.02\right) \cdot 100\right)} = 61 \\ R &= 100 \cdot \frac{0.01 - 0.02}{0.01 - 0.02 \cdot \exp\left(\left(0.02 - 0.01\right) \cdot 100\right)} = 23 \end{aligned} \tag{3.3.30}$$

3. 一方失败后，获胜方还剩下多少作战单位

正如要在后面向大家解释的，这种场景下的战斗总是会一直持续下去。但是，随着时间趋向于无穷大，每支军队剩余的作战单位也趋近于某个值。如果两支军队各项指标完全相同，那么两支军队剩下的作战单位也就趋近于 0。

PDG

$$\begin{aligned} B_{R=0} &= B_0 - \frac{\chi_R}{\chi_B} R_0, \quad \frac{\chi_R}{\chi_B} \leqslant \frac{B_0}{R_0} \\ R_{B=0} &= R_0 - \frac{\chi_B}{\chi_R} B_0, \quad \frac{\chi_R}{\chi_B} \geqslant \frac{B_0}{R_0} \end{aligned} \tag{3.3.31}$$

在这个场景中，红军战败了，它剩下的作战单位也趋近于 0。而蓝军最后剩下的数量接近 50。

$$B_{R=0} = 100 - \frac{0.0001}{0.0002} \times 100 = 50 \tag{3.3.32}$$

4. 战斗持续了多长时间

这个场景中的战斗会一直持续下去。这是因为，随着某支军队剩余作战单位的数量逐渐变小，那它们被击中的概率也会变小。但是，随着时间趋近无穷大，一个军队，或者双方军队中剩余单位的数量总是会趋近 0。可以对这个场景进行一些修改，以便限制战斗的持续时间。当某一方的剩余兵力小于某个数值后，或者小于初始总量的某个百分比，就宣布对方获胜。

3.3.5 场景 4：关底 Boss

假设在一款 RPG 游戏中，一个健康值为 5 000 点的 Boss 与一个由 3 个成员组成的团队打了起来。玩家团队总的健康值为 1 000。在这个系统中，Boss 每个回合造成的伤害是一个固定的量——90 点。而每个玩家可以造成的伤害与其健康值成比例，即每一回合每一点健康值造成的伤害为 1 点。

在这个场景中，蓝军要挑战的是一个单一的红军 Boss。当 Boss 的健康值为 0 时，Boss 就被干掉了。当蓝军团队总的健康值为 0 时，蓝军团队才算被摧毁。这个场景是前面场景 1 和场景 2 的组合。只不过在这个例子中，用剩余健康值的点数取代了剩余作战单位的数量。这也是一个非常好的例子，它展示了当作战双方的规则有所区别时，该如何建立一个作战系统的数学模型。下列公式系统描述了这个场景。

$$\begin{aligned} dB &= -c_R\,dt \\ dR &= -c_B B\,dt \end{aligned} \tag{3.3.33}$$

在这个场景中，B_0 为 1 000，R_0 为 5 000，c_B 为 1，c_R 为 90。

1. 谁会赢呢

当 $R=0$ 时，如果 $B>0$，则蓝方获胜，也就是当下列条件成立时：

$$R_0 < \frac{1}{2}\frac{c_B}{c_R} B_0^2 \tag{3.3.34}$$

在这个场景中，玩家团队打败了 Boss，但其实战况非常接近。

$$5000 < \frac{1}{2} \times \frac{1}{90} \times 1000^2, \quad 5000 < 5556 \tag{3.3.35}$$

2. 在时间 t 的时候，还剩下多少个作战单位

在某个特定的时间，剩余作战单位的数量（剩余的健康值）可以用下列公式计算出来：

$$
\begin{aligned}
B &= B_0 - c_R t \\
R &= R_0 - \left(B_0 c_B t - \frac{1}{2} c_B c_R t^2 \right)
\end{aligned}
\tag{3.3.36}
$$

在这个场景中，5 个回合过后，玩家团队的健康值（以及它的攻击力）几乎下降了一半。但是 Boss 的健康值只下降了 20%。战况还是很接近的。因为玩家团队的攻击力下降了很多，而 Boss 的攻击力却没有太大变化。

$$
\begin{aligned}
B &= 1000 - 90 \times 5 = 550 \\
R &= 5000 - \left(1000 \times 1 \times 5 - \frac{1}{2} \times 1 \times 90 \times 5^2 \right) = 1125
\end{aligned}
\tag{3.3.37}
$$

3. 一方失败后，获胜方还剩下多少健康值

当战斗结束时，剩余的健康值可以用下列公式计算得到。计算结果也只对获胜一方才有效。

$$
\begin{aligned}
B_{R=0} &= \sqrt{B_0^{\ 2} - 2 \frac{c_R}{c_B} R_0} \\
R_{B=0} &= R_0 - \frac{1}{2} \frac{c_B}{c_R} B_0^{\ 2}
\end{aligned}
\tag{3.3.38}
$$

在这个场景中，Boss 被干掉了。但是，玩家团队的总的健康值也只剩下 316 点（初始值为 1 000）。玩家团队的情况如此糟糕，最好不要再继续战斗了，进行恢复：

$$
B_{R=0} = \sqrt{1000^2 - 2 \times \frac{90}{1} \times 5000} = 316 \tag{3.3.39}
$$

4. 战斗持续了多长时间

干掉失败一方所用的时间，可以使用下列公式中的一个来计算。计算的结果只对获胜一方有效。

$$
\begin{aligned}
t_{R=0} &= \frac{B_0 - \sqrt{B_0^{\ 2} - 2 \frac{c_R}{c_B} R_0}}{c_R} \\
t_{B=0} &= \frac{B_0}{c_R}
\end{aligned}
\tag{3.3.40}
$$

在这个场景中，如果要让 Boss 最后取胜，需要大概 11 个回合（每一回合打掉 90 点，总共需要打掉 1 000 点）。但是，玩家团队只需要 7.6 个回合就可以干掉 Boss：

$$
t_{R=0} = \frac{1000 - \sqrt{1000^2 - 2 \times \frac{90}{1} \times 5000}}{90} = 7.6 \tag{3.3.41}
$$

3.3.6 关于战斗力的再讨论

“战斗力”通常是指一个作战单位摧毁敌方作战单位的速率。它的值依赖于具体的作战

系统，还可以包含几个影响因子。例如，某个作战单位的攻击力的值是 a_X,这是一个时间单位中给敌方造成的伤害点数。而它本身也有一个“健康值”或“生命值” h_X，当它受到这个数量的伤害时（健康值为 0 时），它就被摧毁了。这样，蓝军中一个作战单位摧毁红军作战单位的速率就是：a_B/h_R，所以 $c_B = a_B/h_R$，（而 $c_R = a_R/h_B$）。

战斗力也可以融入一个特殊的防御能力，将受到的伤害降低一个固定的百分比。身披厚重盔甲的作战单位，或者机动性很强的作战单位，由于其防御能力的不同，会相应地影响敌方军队的战斗力。

天气和地形也可以影响战斗力。某种天气条件会降低一方军队的效率，但对另一方却没什么影响。在某些类型的地形中作战，某些类型的作战单位会更有效率。只要这些因子对整个军队都是固定不变的，那么它们都可以合并到一个参数中——战斗力。

3.3.7 局限性

这些模型假设的前提是：战斗是连续的，同时发生的。但比较常见的是，游戏中的作战实现都是有时间间隔的，即在特定时间内进行攻击，而不是连续攻击，或者是每个军队轮流攻击。还有些战斗的实现不会把军队分成若干个作战单位，而是所有兵力作为一个整体来作战，直到最后被消灭。对于这些情况，量化过程会导致这些模型和实际的结果之间存在一定的差距。所以，这些模型只能用来近似估算或者预测战斗的结果。但是，如果作战单位的实际数量远远大于它们被摧毁的速率，这个近似值会相当接近实际的结果。

即使这些损耗模型确实不适合用来判断游戏中的战斗结果，但是它们仍然是非常有用的，可以帮助我们设计一个作战系统，并平衡游戏中的作战单位。这里介绍的模型还可以用来判断那些在玩家视野之外的战斗。或者，为了节省时间，推动游戏世界的发展等，也可以用这些模型快速地判断战斗结果。

3.3.8 总结

本文简单介绍了 Lanchester 损耗模型，以及它们在游戏中战斗结果预测上的应用。对于计算机战争游戏和即时策略游戏中常见的几个战斗场景，这里向大家示范了如何应用这些模型，预测战斗结果。当然，也可以对这些模型进行修改和扩展，使它们适用于本文未曾罗列的其他战斗场景和作战系统。

3.3.9 参考文献

[Lanchester16] Lanchester, F.W. “Aircraft in Warfare: The Dawn of the Fourth Arm.” In *Engineering 98*, 422–423; reprinted in *World of Mathematics*, ed. J.Newman, vol. IV, pp.2138–2148. Simon and Schuster, 1956.

[Saperstein01] Saperstein, A “Simple Models Suggest Answers to Complex Questions: Do We Need Satellite Surveillance for Land Warfare? A Lancastrian View.” *Peace Economics, Peace Science and Public Policy*, Volume 8 no. 1: 21–29. Available online at *http://www.crp.cornell.*

edu/peps/journal.htm. August 2004.

[Darilek01] Darilek, R., et al. "Measures of Effectiveness for the Information-Age Army." Ch. 4 Available online at *http://www.rand.org/publications/MR/MR1155/MR1155.ch4.pdf*. August 2004.

[Veale04] Veale, T. "Strategy and Tactics in Military War Games: The Role and Potential of Artificial Intelligence." Lecture III. Available online at *http://www.compapp.dcu.ie/~tonyv/gamesAI/war3.rtf*. August 2004.

3.4 为游戏 AI 实现一个实用的智能规划系统

Relic Entertainment 公司，Jamie Cheng
jcheng@relic.com

加拿大阿尔伯塔大学计算机科学系，Finnegan Southey
fdjsouthey@uwaterloo.ca

随着游戏世界复杂度的不断增加和人们对组队制游戏性的关注度的不断提高，游戏 AI 的发展也得到了极大的推动，完全超越了有限状态自动机（FSM）的限制。游戏策划和引擎上的改动，对有限状态自动机方法是非常致命的。有限状态自动机方法通常都难以调试，而且扩展性极差。要想对最初的设计目标进行扩展，可谓难之又难。设计多个有限状态自动机，让它们聪明地交互作用，协同工作，是非常需要技巧的。而且，游戏中未曾预见的状况马上会让整个 AI 系统陷入混乱。

智能规划（*Planning*）可以很好地解决这些问题。它把推理过程提炼到一个新的层次，并用简洁的语言来表达这些推理过程。这个语言也描述了相关世界中的动作及其影响。通过使用规划（Planning）系统，大部分的推理过程都可以由规划引擎来自动处理。而游戏策划上的改动也可以很快地在规划域中有所反映。由于规划器（planner）可以显式地推理出我们想要得到的最终目标和子目标，所以它能够有效地分配任务，让多个 AI 主体可以互相配合行动。

本文介绍了在这些概念及相关工具上的研究工作，告诉大家如何实现一个实用的规划系统。我们还探讨了如何将规划系统与其他类型的控制系统混合使用（如寻路系统、有限状态自动机和脚本），游戏引擎与规划引擎之间的通信，以及系统优化等其他一些重要的问题。

为了便于讲解，这里引入了一个类似《模拟人生》的游戏。但是在这个游戏中，玩家需要的是为他们模拟的角色指定具体的目标，而不是具体的动作。不过，智能规划方法可以同样适用于各种类型的游戏。我们曾经提出了一些实质性的想法，讨论了它在其他各类游戏中的应用，这些游戏包括：RPG 游戏（角色扮演类游戏）、组队制作战的 FPS 游戏（第一人称射击游戏）、RTS 游戏（即时策略游戏）的高级策略，以及潜入、盗窃/间谍、侦探类游戏等。而且，我们可以预见，在这些游戏类型以及在这些类型之外的游戏中，还会涌现出很多新类型的游戏。

3.4.1　规划系统的框架

从概念上讲，一个规划系统的框架会与一系列游戏世界的状态（*state*）一起工作。这些状态包含着游戏世界在某个时间点上的重要信息。在开始进行系统规划之前，必须提供一个起始状态，用它来描述要推理的游戏世界。例如：

- Bob 在厨房里。
- 储物柜的钥匙放在厨房里。
- 储物柜在小房间里。
- 银盘子在储物柜里。
- 储物柜是锁着的。

通过“动作”，一个状态可以转换为其他的状态。所以，抽象描述中有一部分内容会详细介绍动作是如何改变状态的。在这里，一个“移动（move）”的动作让 Bob 可以在厨房和小房间之间移动。同样地，“拿（take）”的动作可以让 Bob 在自己所在的房间里拾取对象。如果 Bob 对一把钥匙使用了“拿（take）”的动作，那么新状态就是：

- Bob 在厨房里。
- Bob 手里拿着钥匙。
- 储物柜在小房间里。
- 银盘子在储物柜里。
- 储物柜是锁着的。

使用这个模型，游戏策划就可以指定目标条件，如 Bob 拿着盘子。我们关心所有“Bob 拿着盘子”的状态，并把这些状态称为“目标状态（*goal state*）”。还可以有多个可能的目标状态，因为这里指定的惟一一个条件就是：Bob 拿着盘子，而游戏世界的其他部分则可以处于任意一个状态。

规划就是搜索一个动作序列，通过这个动作序列将起始状态转换为目标状态。在搜索过程中，规划器会显式地（或隐式地）构造很多可能的世界状态，逐步向目标靠近。这些状态并不会直接地映射到游戏引擎，而是根据起始状态提供给规划器。如果规划器成功了，它就会提出一个规划：到达目标状态所需要的一系列动作。系统然后就会一步一步地执行这个动作序列，改变实际的游戏世界，如图 3.4.1 所示。

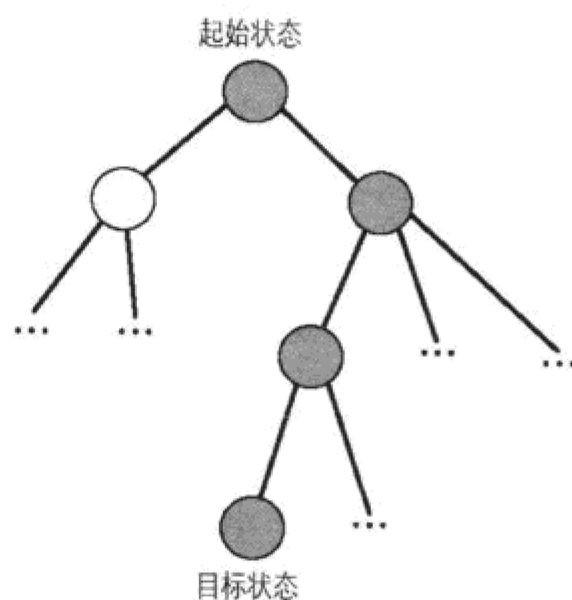


图 3.4.1　到达目标状态的一个规划

3.4.2　规划域

我们把对游戏世界的抽象提取称为游戏世界的“规划域”。经典的规划系统会把世界看成是一个逻辑构造，其中有某些关于这个世界的“事实（*fact*）”会与每个状态相关联。使用命题逻辑在系统中引入规划，是很常见的。我们需要使用一系列的命题，这些命题要么是真

（true），要么是假（false）。具体到程序来说，就是一系列的布尔变量。一个命题要包含一个声明（例如：Bob 拿着钥匙），而这个声明要么是真（true），要么是假（false）。我们把命题看做是一个布尔变量，名为 BOB_CARRIES_KEY。

通过一个布尔函数将对象作为变量，谓词逻辑（*Predicate Logic*）也可以表示同样的命题声明。这样，这个例子就变成了：carrying（Bob，key)。这就提供了一种更为简洁的规划语言。很多当代的规划器都使用谓词逻辑。最著名的规划器 STRIPS[Fikes71]就是以谓词逻辑为基础的。STRIPS 规划语言及其很多变种和扩展集至今仍在使用。现在广为业界接受的规划语言是 PDDL（Planning Domain Definition Language，规划域定义语言），当代大部分的调查规划器使用的就是这种语言[McDermott98]。PDDL 使用了类似 LISP 语言的句法，这在游戏行业中并没有被广泛地使用。因此，我们开发了自己的句法和语法分析器，大体上可以更好地适合游戏行业。对于那些编程经验有限的游戏策划而言，我们开发的这个语言的可读性更强。

1. 对象

谓词逻辑为一系列的对象给出了为真（true）的声明。“对象”指的是组成游戏世界的实体。在前面的例子中，包括的对象有：Bob、钥匙（key)、银盘（silver plate)、小屋（den)、厨房（kicthen）等。这些“对象”与面向对象编程（OO）中的对象不一样，它们没有自己的方法(method)，也没有动、静态之分。但是，将规划对象(planning object)与游戏引擎程序中的 OO 对象以一对一的方式关联起来，是非常自然的事情，甚至是令人向往的事情。这样一来，规划器最后规划出来的动作序列就可以直接翻译成游戏对象的操作。

但是，这些对象确实有类型（type）之分，与面向对象框架中的类型差不多。这个机制提供了类型检查，所以在开发阶段就可以发现一些错误。因为允许多类型继承，所以多个类型就形成了一个“格子（*lattice*)”。我们识别一个特殊的类型“Object（对象)”，通常被称为“通用类型（*universal type*)”，其他所有类型都继承自这个类型。这就把我们带到了第一种声明，为规划器定义对象类型。

```
type Locatable;
type Location;
type Creature isa Locatable;
type Person isa Creature;
type Item isa Locatable;
type Key isa Item;
type ValuablePlate isa Item, Decoration;  //多类型继承
```

一旦定义了类型，就可以定义对象了。

```
object alice isa Person;
object bob isa Person;
object cabinetkey isa Key;
object silverplate isa ValuablePlate;
object kitchen isa Location;
object den isa Location
```

2. 谓词

谓词帮助我们描述游戏世界中对象的某些事实。它将对象元组（tuple）映射为真（true）或假（false）。举个例子，有一个谓词 carrying（持有），它有两个参数。第一参数是 creature（生物），第二个参数是 Item（物品）。现在可以这样来表达我们的思想：carrying(bob, cabinetkey)，或者 carrying(bob,silverplate)。

用谓词表示的事实就组成了规划器中的声明（statement）。一个状态（state）则是关于游戏世界所有事实的一个合集。一般情况下，我们只保存关于这个世界的真实（ture）情况，也就是那些被评估为真（true）的谓词。而其他的谓词表达式都被假设为是假（false）。这就是著名的封闭世界假说（*closed world assumption*）。它的出发点是：在封闭世界中，假的谓词表示要多于真的谓词表示，所以保存真的谓词表示就更高效。

谓词的使用也有几个不同的方法。在某些情况下，我们会断言（*assert*）某些谓词，使谓词表达式在当前状态下为真（true）。这里就断言 carrying(bob,cabinetkey)为真，并把这个事实添加到状态中。另外，也可以将这个谓词当成一个 *query*（询问）：“Is Bob carrying the cabinet key（Bob 是否持有储物柜的钥匙）？”如果在当前状态下，这是一个事实，那么这个谓词就返回 true。

在规划语言中，谓词是有参数的，而且必须进行显式声明。

```
predicate carrying(Creature, Item);
predicate inroom(Locatable, Location);
```

3. 变量

谓词记录着对象之间的关系。我们可以询问是否存在某个关系。但是，我们要更进一步，实现同时查询若干个关系的功能。这是通过变量来实现的。例如，如果想知道 Bob 都拿了些什么东西，就可以这样询问：carrying(bob,? x)。其中的“? x”是一个标记，表示那个“x”是一个变量。

把这个过程当作模式匹配来考虑，似乎更容易理解。Query 会检查状态中 carrying 谓词的所有事实。如果某个谓词的第一参数是 Bob，就把这个谓词的第二个参数的值依次地赋给 x。如果 query 找到了一个或多个合适的匹配，它就会为这些匹配分别返回 true，然后再相应地绑定一个或多个变量。

4. 逻辑连接符

使用逻辑与（and）和逻辑非（not），可以创造出更有趣的 query（询问）。例如，可以检测 Bob 和钥匙是否在同一个房间里：

```
inroom(bob, ?x) and inroom(cabinetkey, ?x)
```

其中，第一个 inroom()询问会根据“bob”找到一个匹配的谓词，然后将? x 绑定到 kitchen。第二个询问就变成了 inroom (key1, kitchen)。

还有一件很重要的事情，就是要考虑用一个特殊种类的谓词来表示“不同于（*inequality*）”。仅当两个参数是不同的对象时，这个谓词才返回 true，例如：?x != ?y。

5. 操作符

可以使用操作符（*operator*），从一个状态转换到另一个状态。操作符的形式很简单，它带有一个 *precondition*（前提条件）和多个 effects（影响）。在执行某个动作之前，precondition（前提条件）中的询问（query）语句必须为真（true）；*effects*（影响）表示应该对游戏世界做出一系列相应的改变。它们都可以使用简单的逻辑表达式（由谓词、逻辑与和逻辑非组成）来声明。

```
operator take(Creature ?c, Item ?i)
    precondition: inroom (?c, ?r) and inroom(?i, ?r)
    effect: not inroom (?i, ?r) and carrying(?c, ?r);
```

操作符（operator）有两个参数（请注意它们的类型）。precondition（前提条件）的声明是：creature（生物）必须与 item（物品）在同一个地点。effect（影响）语句则会创建一个新状态，它与起始状态相同，然后将"creature is carrying the item（生物持有这个物品）"的事实添加的这个状态中。

注意：变量 ?r 并不是操作符的一个参数。这是因为它对这个动作并不是必需的，它只是一些信息，告诉我们该如何执行它。可以把它看成是操作符的一个局部变量。当游戏引擎需要真正执行这个动作时，它只需要知道 creature（生物）和 item（物品）就可以了。

有了操作符，就可以检查某个状态，通过测试每个动作的 precondition（前提条件），就可以自动地判断出哪些动作是可以执行的。每当有 precondition（前提条件）可以匹配，就执行一次操作符的动作。就实现规划的搜索而言，现在已经万事俱备，只差一个搜索算法了。

3.4.3 一个多主体规划器的例子

综合上面介绍的基本内容，现在可以看一个简单的例子，它可以清楚地展现规划器的强大功能。在这个例子中，首先要创建一个状态，包含一个可以很容易解决的问题。然后，将这个问题进行扩展，把它变成一个要求两个 AI 主体共同来求解的问题。

```
type Locatable;
type Location;
type Creature isa Locatable;
type Person isa Creature;
type Dog isa Creature;
type Container isa Locatable;
type Item isa Locatable;
type Key isa Item;
type Plate isa Item;
type ValuablePlate isa Plate, Decoration;           //多类型继承
type Food isa Item;
```

完成类型定义后，可以定义对象了。

```
object alice isa Person;
object bob isa Person;
object cabinetkey isa Key;
object silverplate isa ValuablePlate;
object kitchen isa Location;
object den isa Location;
object cabinet isa Container;
object nelly isa Dog;
object lasagna isa Food;
object fridge isa Container;

predicate inroom(Locatable ?a, Location ?b);
// 对象在 location 中
predicate carrying(Creature ?c, Item ?i);
// Creature（生物）持有 item（物品）
predicate locked(Container ?c);
// Container（容器）是锁着的
predicate unlocks(Key ?k, Container ?c);
// Key（钥匙）打开了 Container（容器）
predicate hungry(Creature ?c);
// Creature（生物）饿了
predicate prepared(Food ?f);
// Food（食物）准备好了
predicate inside(Item ?i, Container ?c);
// item（物品）在 container（容器）中

// 拾取房间中的某个 item（物品）
operator take(Person ?p, Item ?i)
    precondition: inroom(?p, ?r) and inroom (?i, ?r)
    effect: not inroom (?i, ?r) and carrying(?p, ?r);

// 移动到另一个房间
operator goto(Person ?p, Room ?r)
    precondition: inroom(?p, ?o) and ?o != ?r
    effect: inroom (?p, ?r) and not inroom(?p, ?o);

// 放下持有的某个东西
operator drop(Person ?p, Item ?i)
    precondition: inroom(?p, ?r) and carrying(?p, ?i)
    effect: inroom (?i, ?r) and not carrying(?p, ?i);

// 从 container（容器）中取出某个东西
operator remove(Person ?c, Item ?i)
    precondition: inside(?i, ?v) and inroom(?c, ?r) and inroom(?v, ?r) and unlocked(?v)
    effect: carrying(?c, ?i) and not inside(?i, ?v);

// 用钥匙打开 container（容器）
operator unlock(Person ?c, Container ?v)
    precondition: locked(?v) and unlocks(?k, ?v) and carrying(?c, ?k)
    effect: not locked(?v);
```

PDG

```
// 准备一些食物
operator prepare(Person ?c, Food ?f)
    precondition: carrying(?c, ?f) and not prepared(?f)
    effect: prepared(?f);

// 吃掉plate（盘子）里准备好的食物
operator eat(Person ?c, Food ?f, Plate ?p)
    precondition: carrying(?c, ?f) and carrying(?c, ?p) and hungry(?c) and prepared(?f)
    effect: not hungry(?c) and not carrying(?c, ?f);
```

下面来定义起始状态和目标状态。这个简单的事情就是：Bob 需要这个 plate（盘子），这个盘子在 den（小屋）里，而 Bob 在 kitchen（厨房）里：

```
//定义起始状态
inroom(bob, kitchen)
inroom(silverplate, den)


//定义目标状态
goal carrying(bob, silverplate)
```

给定这两个状态后，规划器返回如下结果：

```
goto(bob, den)
take(bob, silverplate)
```

接下来看看扩展这个问题是多么容易：定义一个新的起始状态，让盘子被锁在小屋里的储屋柜中，而 Alice 拿着钥匙。这里还是要达到同样的目标：

```
//定义起始状态
inroom(alice, den)
inroom(bob, kitchen)
carrying(alice, cabinetkey)
inroom(cabinet, den)
unlocks(cabinetkey, cabinet)
inside(silverplate, cabinet)
locked(cabinet)
```

一个可能的规划是：

```
unlock(alice, cabinet)
remove(alice, silverplate)
goto(alice, kitchen)
drop(alice, silverplate)
take(bob, silverplate)
```

现在考虑一个新的情况：Alice 有点饿了，要喂她吃点东西：

```
//定义起始状态
inroom(alice, den)
inroom(bob, kitchen)
```

```
carrying(alice, cabinetkey)
inroom(cabinet, den)
unlocks(cabinetkey, cabinet)
locked(cabinet)
hungry(alice)
inside(silverplate, cabinet)
inside(lasagna, fridge)

//定义目标状态
not hungry(alice)
```

达到这个目标状态的规划是：

```
unlock(alice, cabinet)
remove(alice, silverplate)
goto(alice, kitchen)
remove(bob, lasagna)
prepare(bob, lasagna)
drop(bob, lasagna)
eat(alice, lasgna, silverplate)
```

这个简单的例子有着无限的分支。突然之间，游戏策划就可以很容易地创建一个场景，要求 AI 主体来配合求解一个问题！还有更强的功能：随着玩家改变游戏世界，AI 主体可以自动地重新规划，并执行相应的动作。在游戏进行的过程中，系统可以断言（assert）新的谓词，或者增加新的操作符，让 creature（生物）可以做更多的事情。其他的可选项包括：游戏策划编写简单的脚本，为规划器指定需要执行的目标，或者玩家可以要求某个 AI 主体来帮忙。无限可能尽在其中！

要想了解这些可能性，最好的办法就是去思考。对于前面给定的规划域，希望大家花些时间，考虑一下如何快速地实现下列功能：

- 增加一个可以让人与人之间直接传递物品的方法。
- 描述这个房子的布局，增加一道门。
- 增加一个小偷，他只能利用未上锁的门，只盗窃贵重物品。
- 增加一个操作符（operator），当房间里没人时，让小狗可以自己吃东西。
- 让 Bob 变得矮一些，使他无法拿到储物柜里的盘子，因而需要 Alice 的帮忙。
- 如果 Bob 在房间里，就禁止 Alice 打开储物柜的锁。
- 需要两个人来移动电冰箱。
- 食物在吃之前需要加热。
- 制造一些混乱，作为烹饪动作的结果（比如：脏盘子），因而需要进行清理工作。
- 只允许受过训练的人来做饭。
- 增加驾车出去采购要准备的食物的行为（需要有驾驶执照的人）。
- 要求人与人之间彼此互相喜欢，或者在正确的情绪下进行合作。
- 给 Alice 一把剑，给 Bob 几件小盔甲，给小狗 3 个头，制作一个 RPG 小游戏。
- 给 Alice 夜视的功能，给 Bob 全套卫兵的装备，让小狗有嗅觉功能，制作一个潜入盗窃类游戏。

在这个家务模拟游戏中，虽然使用的是由 AI 主体自动进行的规划，但仍然有很多诱人

的游戏性：在当前游戏中设定的这些动作的基础上，为 AI 主体指定一系列的目标；为游戏世界增加新的对象来帮助它们；为 AI 主体增加新的属性，对它们进行训练，使它们获得新的能力；或者让周围的一个 AI 主体来指导某个 AI 主体的动作。

3.4.4 规划的搜索

但是，得到所有这些好处并非没有代价。在开发成本和机器资源方面，规划系统的实现都存在相关的系统开销。开发人员必须指定规划域，其本质就是对游戏世界的二次描述。同时还要确保这个域中的所有动作不会产生荒谬的状态（例如，玩家同时出现在两个地方）。必须像正规程序那样设计和调试规划域。和编译程序一样，规划器只能根据输入的信息来运行。如果有垃圾信息输入进去，那么出来的东西也只能是垃圾。如果只用比较简单或比较低层次的系统机制，或许可以省掉这些开发成本，但是就得不到更有智能的角色行为。

规划工作最实际的开销是在搜索上。根据所使用的特定方法，规划搜索可能会消耗掉大量的内存。正是因为这个原因，规划搜索的实现有很多不同的方法。

第一个差别是正向规划和逆向规划。在正向规划中，是从起始状态开始，依次考察每个状态，直到找到一个目标状态。在逆向规划中，则是从目标状态开始，考察所有这些动作，找到那些可以达到这个目标的动作。这些动作的 precondition（前提条件）又形成一个新的目标，然后继续逆向搜索。逆向规划法有它的优势，它的分支系数比较小，因为它根本不会考虑那些不能满足目标条件的动作。可以想象，对于正向规划法，它则要尝试所有可能的动作。另一方面，正向规划法是更为自然的方向，它只考虑那些从起始状态可以达到的状态；而且即使是一个不完整的正向状态序列也是有意义的，也是可以执行的。

搜索策略之间的另外一个主要差别是搜索状态空间（*state space*），还是搜索规划空间（*plan space*）。在状态空间中，搜索的是状态，最后再构造出一个可以达到目标状态的状态序列。这些状态所必需的动作会组成一个列表，从这个列表中可以提取出需要的规划。而在规划空间中，直接考虑的就是动作序列或动作集，对它们逐个进行检测，看是否可以达到目标。随着搜索工作的进行，需要不断地提炼、扩展采集到的动作。

大部分当代的规划器都属于下列三大类规划器中的一种：启发式搜索规划器（*heuristic search planner*）、图规划系统规划器（*planning graph planner*）和可满足性规划器（*satisfiability planner*）。下面会简要介绍以上三种规划器系统，但重点还是放在启发式搜索规划器上。因为，对游戏来说，它是最佳选择。理由有很多，这也是要探讨的内容。

1. 启发式搜索规划系统

启发式搜索规划系统采用的是状态空间法，可以进行正向搜索，也可以进行逆向搜索。搜索过程与寻路算法非常类似。可以直接应用大家熟知的一些算法，如 A*算法，并且可以重用现有的 A*算法代码。和寻路算法一样，也可以很容易地计算出某个动作的开销。这样，就可以计算出一个“好的”规划，而不仅仅是一个可行的规划。在这些开销之内，并利用 A*算法所使用的启发式，对游戏的特定知识进行编码是相当容易的。

另外一个关键的优势是，即使正向规划失败（没有满足条件的状态序列，或者是因为限

定了规划器运行的时间)，没有找到目标状态，还是可以利用当前所找到的最佳路径，这样游戏角色至少可以做一些看上去还比较合理的事情。而且，在随后的二次规划中，这些不完整的动作序列也许可以帮助我们找到一个完整的规划。

最后一点，启发式搜索规划系统还可以提供最优异的性能，特别是对那些非最优规划器来说（最优规划器能够找到最小规划长度的规划，或者是开销最小的规划；而非最优规划器只会去搜索“足够好的”规划。对游戏来说，也许并不需要最优的方案）。非最优规划器的例子就是 FF[Hoffmann01]，它是最近一届规划器大赛的优胜者。

2. 图规划系统和可满足性规划系统

图规划（graph plan）系统本质上是一个平面空间中的逆向规划器。从高层次角度来看，图规划系统的基本思路是构造一个规划图（*planning graph*）。从起始状态开始，所有可能的动作及其影响都包含在这个数据结构中，并假设这些动作不会彼此冲突。在构造规划图时，互斥动作的信息都会保留下来。规划图完成之后，从目标状态开始，执行一个逆向搜索，在规划图中找到合法的动作集[Weld99]。

图规划系统会生成半序规划（*partially ordered* plan），并不指定准确的动作序列。因此，有些动作的执行顺序是随意的；或者在多个 AI 主体共同执行动作的情况下，某些动作可以同时执行。如果要控制多个 AI 主体，这个特性尤其诱人。但是，逆向规划也意味着，如果没有足够的时间去构造一个完整的规划，那么也就没有不完整的规划可以执行。虽然这个方法有可能适合游戏开发，但却没有启发式搜索法那样强的适应性。值得一提的是，有些时候，人们会使用规划图为其他种类的规划器来计算启发式（例如，前面提到的 FF 规划器）。

可满足性（SAT）规划器将规划问题编码转换为 SAT（可满足性）问题。SAT（可满足性）问题是经典的 NP（Non-deterministic Polynomial，即多项式复杂程度的非确定性问题）完全问题，有很多现成的、高性能的求解器（solver）可以使用。SAT 规划器之间最大的区别在于它们对规划问题进行编码的方式。一旦完成编码，它们就可以调用最新的、最优秀的 SAT 求解器来生成最后的规划。这就意味着，这些规划器可以使用现成的组件，年复一年地不断地改进。但是，由于 CNF 范式实例已经变得越来越庞大，让人难以理解；而 SAT 求解器本质上就是一个黑盒子，很难用它为问题去编码一些额外的知识，所以 SAT 规划器变得越来越晦涩难懂，可编程性也越来越差。对于此方法，这里不再进行更深入的探讨。不过，由于 SAT 求解器的飞速发展，该方法仍然值得我们去考虑。

3.4.5 几个应用问题

下面，讨论几个规划器的应用问题。

1. 局限性

规划器是一个功能强大的工具，应该把它与已有的工具组合起来，让工作更有效率。通过使用规划器，原先嵌套在有限状态自动机（FSM）及其规则中的很多复杂问题，都可以在规划域中得到更好的表达，给我们留下的是这些低级构造的简化版本。但是，规划器也有一

些局限性：

1．像规划器这样的符号式方法，不能适用于连续性问题（例如，不能用它求解开车或瞄准目标的问题）。应该用现有的其他机制来处理这些问题，用规划器来指挥相应的动作。最好把规划器看成是一个协调机制，协调简单、低级的动作，让它们以可用的最佳方式来实现。

2．除了有限的应用范围之外，经典的规划器都有一些很硬性的假设。它假设了一个世界，一个可以按照我们的描述来运转的世界，而且假设所有的动作都会成功。但是很明显，因为随机性和偶然性的存在，也因为我们抽象化了游戏世界，最重要的，是因为人类的玩家，所以现实的情况根本不是假设的那样。这就是为什么必须要定期地重新进行规划，来反映这个变化的世界和先前那些规划的失灵。

3．经典的规划器都是非对抗性的，它不会考虑有人来阻挠的情况。但是，只有很少的几个游戏 AI 系统着手在解决这个问题。它们不只是使用了基本的启发式和规则，还使用了一些基本策略。这些基本策略大部分可以归结为目标选择：用一个高层的策略模块选择目标，规划器给出实现这个目标的路径，然后给有限状态自动机发出指令，执行相应的低层次的动作。

4．这种符号式的方法也不太擅长进行数值的判断，比如两个 creature（生物）的相对力量强弱。在这种情况下，应该使用现有的启发式函数和规划器中的一个谓词。规划器调用这个启发式函数，并采用一个阈值谓词（例如：`canKill(Creature,Creature)`），就可以检测出其中一方的力量是对方的两倍。

2. 与游戏引擎的集成

除了上面这些问题，如何将规划器整合到游戏引擎中，也是一个很重要的问题。这个问题至少部分地取决于规划器代码的可重用程度。但是，这里还是会谈论几个关键点。

首先考虑的是规划器的起始状态，这是游戏引擎需要获得的状态。其他在规划过程中生成的后续状态必须是这个起始状态的复制品。我们不希望在规划的过程中改变游戏引擎中的变量。为了获得起始状态的事实（fact），有下列几个策略。

在“pull（拉式）”策略中，规划器从游戏引擎中获得信息，来构造状态。在开始规划之前，规划器会检查所有的谓词，并从游戏引擎中获得它们的值，保存在起始状态中。但是这种重复的拷贝操作也不是没有开销的，而且大量的无用状态信息也可能被保存起来。

解决这个问题的一个方案是“按需读取（*read-on-demand*）”。在这个方法中，仅在评估谓词时，系统才去读取相应的信息。另外一种方式是使用“push（推式）”模型。在这个模型中，只有当状态信息变化了，游戏引擎才会把变化的信息发送给规划器。如果需要频繁地进行规划，这是个不错的可用策略。而对于非频繁的规划操作，则会执行很多不必要的更新操作。

利用游戏引擎中的数据结构和函数，将规划对象与动作关联起来，这个工作也是必需的。这个映射是相当直接的，但是也应该在策划阶段仔细地考虑清楚。另外，不同的模块应该可以和规划器交互。我们开发了一个定制的规划语言，来描述规划域（类型、对象、谓词和操作符）。但是，这里使用了一个流行的脚本语言——Lua，用它来驱动起始状态和目标，将对象和动作绑定到游戏引擎中[Lua]。这个规划器也可以直接通过 C++语言来控制。游戏策划者

们可以使用他们自己喜欢的脚本语言来精化、调整关卡；并在同一个脚本中，让 AI 主体去解决一些有趣的问题，而不需要加载另外一种不同语法的脚本系统。

请注意，我们完全有可能不用编写一个定制的语法分析器，而通过 Lua 脚本（或其他你喜欢的脚本语言）来驱动整个规划器。本文之所以使用了一个定制的语法分析器，是因为它对读者更有指导意义，也更容易帮助大家理解一个完整的规划器开发过程。但是，它确实不是必需的。

3.4.6 优化

智能规划在理论上是很牢固的，而且创建高难度的规划问题也很容易。但是，游戏展示了一个需要更多控制的实例。在机器人学和工业领域，世界是不可控的。在游戏中，游戏世界是经过精心策划和设计的，有着完美的可观测性。我们开发可靠的游戏，而 QA 人员期待的游戏品质也可以让我们很容易地生成规划域。大部分的游戏都经过仔细的策划，这样玩家就不会被搞晕。游戏的每一个状态都应该有一个明确的结果（即使这个结果是死亡）。没有中间过渡状态对规划系统是有好处的。

很多游戏在它们的动作中都包含了大量的单调性。这就意味着，一旦某个动作发生了，它对游戏世界的影响就一直存在。这样的例子包括：一旦捡起来就再也不会被放下的对象，被锁上的门再也不会被再次锁上，被杀死后不会再次出现的怪物。回想一下曾经玩过的游戏，你会逐渐看到大量的单调性。这是因为，为了试图维持动作的简单性，良好的游戏策划通常会导致单调性。如果没有什么理由要让那些锁着的门再次被锁上，或者被按下的开关又被按下一次，单调的动作可以简化控制，简化玩家的选择，并简化调试和试玩工作。

单调性也带来简单的规划域。而且，很多规划算法都假定规划域像启发式一样，是单调的。这样的算法在广泛单调的游戏世界里是非常高效的。但是，非单调性也将会发展起来，逐步成为有趣的游戏性中一个很自然的组成部分。对于非单调性，当代的规划算法也可以执行得很好。对于例外的情况，如果有一些高难度的非单调性子任务确实是一个问题，那么可以用一个定制的解决方案，一个专门的算法来解决这个问题。该过程可以看成是规划器的一个动作。将比较难的问题作为一个单独、特殊的动作，可以有效地消除规划域中的难点部分。

例如，可以要求某个 AI 主体在一个由很多扇门组成的复杂迷宫中寻找出路，还要求所有的开关必须按下两次，才能逃生。如果规划器在求解这个问题时速度非常慢，那么可以设计一个专门的算法或者脚本化的解决方案，与一个 `navigateMaze`（走迷宫）的动作关联起来。在规划层，这个动作只是简单地要求 AI 主体站在迷宫的起点位置上；这个动作的影响就是把 AI 主体放到迷宫的出口处。具体该怎么走，由那个专门的算法来处理。类似这样的问题还有拼图游戏。这类问题引发的疑惑是，游戏策划是否应该把拼图工作交给 AI 系统来求解。由于拼图游戏通常的目的是为了挑战玩家，所以要么让 AI 系统绕过它，要么就强制玩家去求解。

还有很多其他可以优化的地方。比如，我们有个想法，动作的顺序对这个想法非常重要，那么我们可以设置一个顺序，让规划器以这个顺序去查找动作。如果某些状态含有一

些烦人的属性，我们也可以自动丢弃这些状态，以便精简搜索过程。最后要告诉大家，还有大量的研究文献和很多优秀的教程材料，它们都提出了很多很好的优化思路，可以给大家更多启发。

3.4.7 总结

规划算法为游戏中的智能行为增加了新的维度，并降低了现有游戏 AI 系统实现的复杂度。规划系统与游戏可谓是珠联璧合。最新的基于 A*算法的策略不仅为广大的游戏开发人员所熟悉，也是规划搜索算法中的尖端技术。让游戏采用规划系统的时机已经成熟。

例如，Monolith Productions 公司的 Jeff Orkin，已经在他们即将推出的 FPS（第一人称射击）游戏 *F.E.A.R* 中集成了一个规划系统。它采用了一个命题逻辑规划器。虽然只是使用很少量的事实（大概 20 个左右）为一个 AI 主体提供规划，但是它却在快节奏、实时的 FPS 游戏环境中，提供了动态的、前瞻性的人工智能[Orkin04]。如果再增加一些复杂的启发式，并进行一些优化，它完全可以应用于多个 AI 主体和大型场景。我们自己的这个规划器则更为普通，从开始到实现只用了大约一个月的时间。

对于即时性稍差一点的游戏，比如 RPG 游戏，我们认为更大的游戏环境也是可以用规划域来建模的。对于战术战略类游戏和潜入盗窃类游戏，它们也可以从这个技术中受益。规划系统可以帮助它们减少 NPC 角色所表现出来的那些多少有点短视的行为。无论是盟友，或是对手，都可以为玩家提供一个更丰富、更生动的游戏体验；下一代崭新的游戏性展现在游戏策划者的面前，等待他们去把玩。我们不只认为这是可能的，而且确信这是必然会发生的。

3.4.8 参考文献

[Bonet01] Bonet, Blai and Hector Geffner. "Planning as Heuristic Search." In *Artificial Intelligence-Special Issue on Heuristic Search*, Vol. 129, No. 1–2, 2001. Available online at *http://www.tecn.upf.es/~hgeffner/index.html*.

[Fikes71] Fikes, R. and N. Nilsson. "STRIPS: A new approach to the application of theorem proving to problem solving." In *Artificial Intelligence*, Vol. 2, No. 3–4, 1971: pp. 189–208.

[Ghallab04] Ghallab, Malik, Dana Nau, and Traverso Paolo. *Automated Planning-Theory and Practice*. Morgan Kaufmann, 2004.

[Hoffmann01] Hoffmann, J. and B. Nebel. "The FF Planning System: Fast Plan Generation Through Heuristic Search." In *Journal of Artificial Intelligence Research*, Vol. 14, 2001: pp. 253–302. Available online at *http://www.informatik.uni-freiburg.de/~hoffmann/ff.html*.

[Lua] "Lua The Programming Language." *http://www.lua.org*.

[McDermott98] McDermott, Drew, et al. "Planning Domain Definition Language." 1998. Available online at *http://www.informatik.uni-freiburg.de/~hoffman/ipc-4/pddl.html*.

[Orkin04] Orkin, Jeff. "Symbolic Representation of Game World State: Toward Real-Time

Planning in Games." In *AAAI Workshop Technical Report WS-04-04: Challenges in Game Artificial Intelligence* (July, 2004): pp. 26–30.

[Weld99] Weld, Daniel S. "Recent Advances in AI Planning." In *AI Magazine* (1999): pp. 93-123. Available online at *http://www.cs.washington.edu/homes/weld/papers/pi2.pdf*.

3.5 针对多线程架构的决策树查询算法优化

Intel 公司，Chuck DeSylva
Chuck.v.desylva@intel.com

使用决策树（Decision Tree）可以很容易地实现小巧、快速的游戏 AI 系统。这样的游戏 AI 系统更易于实现和维护。虽然从长远角度看，神经网络似乎效果会更好（一旦它经过训练之后），但是对于一个期望值（expectation）的有限集，就像一个典型的游戏中所定义的那样，决策树仍然是个理想的选择。

本文向大家介绍一个很简单的方法，用它可以改进一个基本决策树算法的实现。这是一个相当直接的方法，实现起来也很容易。更重要的是，它可以带来更快速的 AI 响应。这个方法尤其适用于那些带有庞大决策集的 AI 系统。在这些 AI 系统中，对于一个给定的 AI 响应集（例如：攻击、移动等），游戏中大量的 AI 单位都使用一棵或多棵一样的决策树，而这些决策树都是从一个执行（execution）的单一主线程中产生出来的。

3.5.1 概述

从本质上讲，决策树就是对一系列问题所提供的数据进行评估，并给出一个建议模型，来解释这些数据，以此做出准确的预测。根据游戏场景中的数据，游戏中的 AI 角色会提出一系列的问题，这些问题就构成了一棵决策树。对于一个给定的数据集，可以提出大量不同种类的问题，所以一个典型的游戏 AI 系统会包含（也应该包含）大量的决策树（例如：寻路、自发反应、Q&A，以及某种程度的 AI 系统物理方面的工作，如回避及响应)。对于那些更为复杂的系统，例如 agent 系统（智能主体)，决策树就变得更为重要。

决策树可以操作有限数据集，或无限数据集。对于游戏这种应用程序，有限数据集是最为重要的。与其他算法相比，决策树有两个最显而易见的价值：不但提供了简单、离散的回归结果的解释（explanation)，而且对决策依据（decision rationale）的“解释”也非常容易。目前，在决策树 AI 系统应用领域，使用最广泛的一个决策树算法是 Quinlan 提出的 ID3 算法。关于这个算法及其工作原理，互联网上有很多现成可用的参考资料。

决策树本质上就是计算机科学中的二叉树结构，从根节点（表示某个决策范畴的开始）开始，经过一系列的决策状态，最后达到一个结论状态。在本文所使用的例子中，所有的状态都是两路的，如图 3.5.1 所示。决策树提供了一个技术，让我们可以根据一个特定问题的定义做出最后的结论。

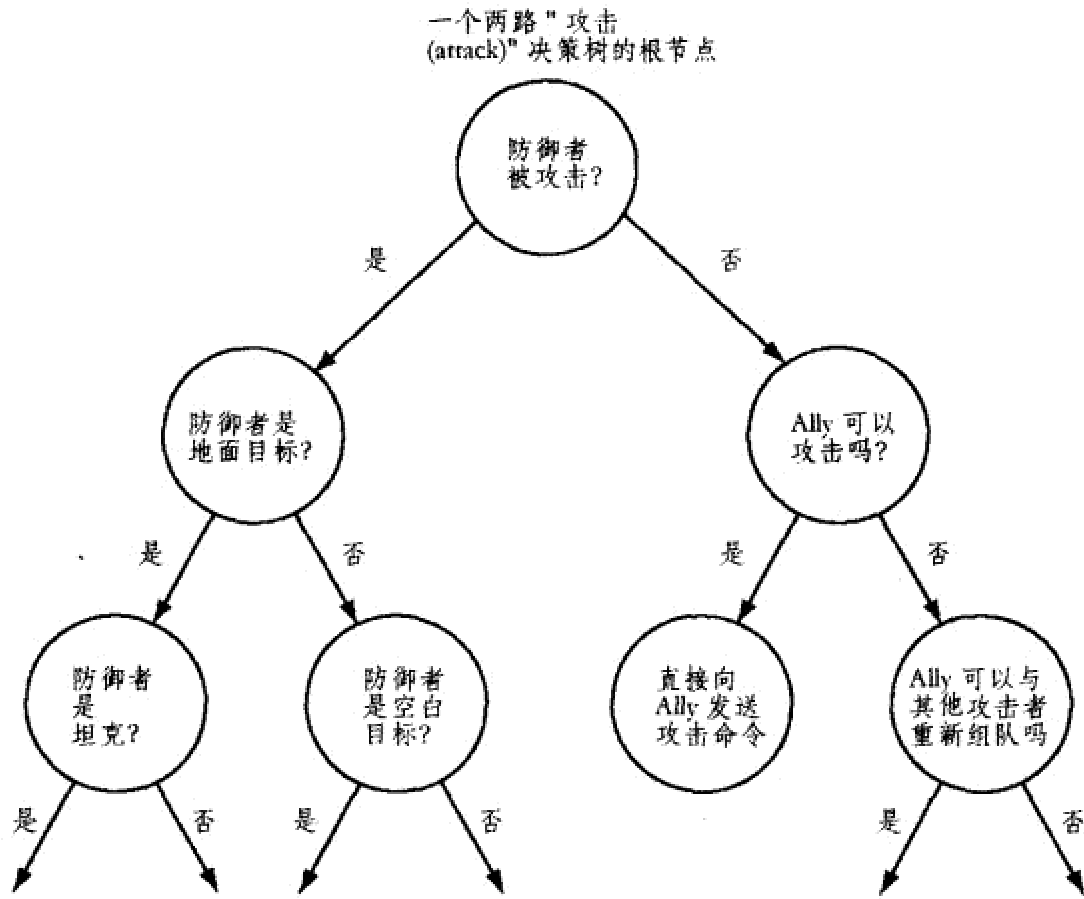


图 3.5.1　一个基于 2 路的"攻击"决策树

由于决策树是一个很好的方法，可以将训练过的问题集转换为树的节点，并利用这些节点得出最终的答案（对问题的回答），所以它们不但对分类系统特别有用处，而且对计算机游戏 AI 系统也大有帮助。决策树的架构通常需要大量的输入信息，这些信息由讨论中的设计方案给出，而不同的设计方案会带来不同的决策树架构。但是，由于现代游戏中的问题大多比较简单，利用与决策树类似的多线程化算法创建独立于游戏（与游戏无关的）的 AI 角色，这种可能性已经离我们越来越近了。从现在开始，我们将会把话题限定在一个简单的优化方法上。利用这个优化方法，对决策树算法的一个实现进行优化，可以将这个实现扩展为其他与之类似的决策树的实现（也就是 n 路树，每个节点有多个答案的决策树）。

3.5.2　注意事项

有些决策树涉及的决策可能有多种答案（除了 yes 和 no 之外的答案）。归纳学习法（*inductive learning*）就是这种类型的 AI。在归纳学习法中，系统的目标是找到一些规则或函数，让我们可以根据在决策树上任何一点获得的决策案例（decision case）得出有用的结论。对于任何一个特定的决策集，各种输入信息都是通过熵函数进行分类的。熵函数通常都是基于对数函数（logx）的。在有关这个主题的大量资源中，都有关于这些函数的详细信息，而且它们对决策树算法的性能没有太大的影响（现代很多的编译程序都可以使用特定的编译选项，自动地优化数学函数[1]），因此，这里就不去讨论算法内部运作的细微差别了，而是把重点放在如何利用优化工作获得最大的性能回报。

最初，这个决策树的实现是通过在命令行输入命题来求解答案的。为了能够有效地对它

[1] 例如，使用某些编译选项（即 Netburst 架构的/QaxW 选项），Intel 公司的 C/C++编译器（v.8.0）可以将数学函数矢量化。

进行剖析，我们对它进行了改造，让它从一个内存映射文件获取输入信息（以便模拟来自游戏的实时输入信息）。另外，可以将树的深度设计得足够深（在一个强健的 AI 系统中，决策树的深度通常都非常的深），使得我们对这个算法的剖析更接近于一个真实世界的实现。

3.5.3 优化

大多数游戏开发者都希望他们的 AI 系统所占用的时间不会超过其（帧）计算周期时钟 20%左右的时间。与一个强健游戏体验所具有的其他苛刻要求相比，这是一个相当宽容的要求。如果有一棵完全平衡的决策树（一棵 yes 和 no 分布均等，且没有分支的决策树，如图 3.5.2 所示），在一个多线程感知（或者说支持多线程）的系统中[2]，使用优化技术，可以再从 AI 系统那里拿回 10%的时间。

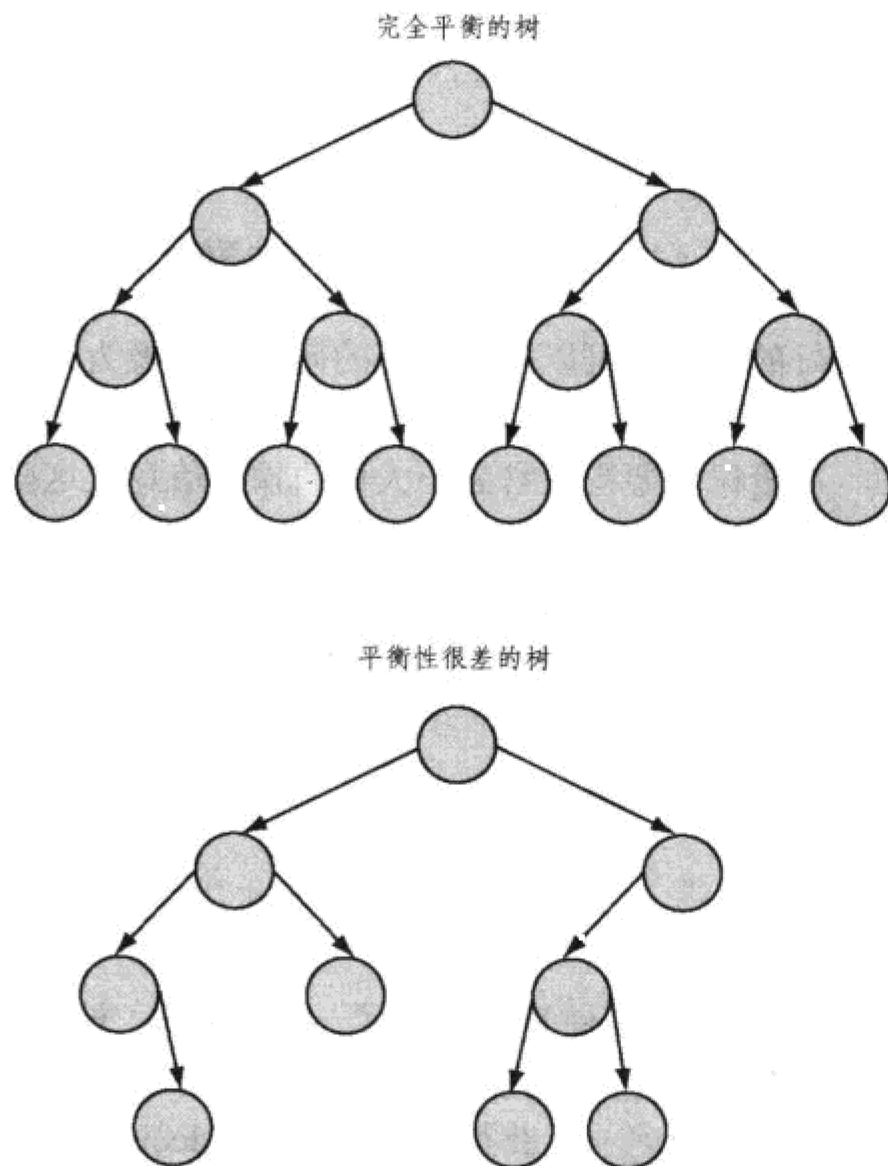


图 3.5.2 完全平衡的树和平衡性很差的树，二者的深度都是(δ)4

本文向大家详细介绍的这个决策树算法，是根据 Generation 5 网站上提供的资料来实现的。这个网站上详细地介绍了一个简单、但却强健，且易于修改的决策树实现。

对这个例子的代码修改是通过一个支持 OpenMP（参见 OpenMP 规范说明 2.0）指令的编译程序[3]来完成的。由于这个算法的主要（运行）时间都花在答案搜索上，所以我们选择这

[2] 诸如所有的多处理器系统。
[3] Intel 公司的 C/C++ 8.0 编译器。在撰写本文时，微软公司的 Visual Studio .Net 2005 也打算支持 OpenMP。

个算法的搜索部分，对它进行优化。

由于这个算法本身是递归的，所以需要一些额外的例程。算法的开始是调用成员函数 DecisionTree::Query()（参见程序清单 3.5.1)。然后，这个成员函数再调用另一个成员函数 DecisionTree::QueryBinaryTree(*dt node)。随着搜索工作的进行，在这个函数和 DecisionTree::AskQuestion(*dt node)之间一系列的递归调用{ δ：这里的 δ是树的深度}让双方彼此调用对方，直到找到答案节点，或者没找到合适的答案。

随着台式机 CPU 工业逐步接近于基于 80×86 的 32 位计算和 64 位计算共存的状态，应用软件的性能面临又一次的考验。业界的期望是，带有这些技术的基于 80×86 的处理器[4]会要求应用软件进行适当的扩展，以便维持一个与硬件的能力水平相匹配的性能。由于双核和多核技术的发展将会重新定义当今应用软件的可伸缩性（scalability)，将应用程序适当地线程化将是保证应用程序性能的关键所在。

程序清单 3.5.1 未修改前的决策树搜索代码

```
void DecisionTree::Query(){
        QueryBinaryTree(m_pRootNode);
}

void DecisionTree::QueryBiaryTree(TreeNode*currentNode){
      // 错误检查，否则就缺省地在 currentNode（当前节点）上提出问题
      AskQuestion(CurrentNode);
          // 否则就缺省地在 currentNode（当前节点）上提出问题
ode
}

void DecisionTree::AskQuestion(TreeNode*node){
      // 错误检查
      if(answer=="yes")
             QueryBinaryTree(node->m-_pYesBranch);
      else if(answer=="no")
             QueryBinaryTree(node->m-_pNoBranch);
      else {    //输入错误
      }         AskQuestion(node);
}
```

为了与这个发展趋势保持一致，针对该算法提出的优化方法的目标系统就是双 CPU（逻辑或物理上的）系统。如果要扩展成 4 路、8 路和 16 路的实现，就意味着要做进一步修改，以便优化方法也可以适用于这些实现。我们的修改方法是进行分支化（branching）处理，在第 N 次（从根节点开始）递归迭代中调用第一个线程化的例程。也就是说，线程化操作的起点是在 δ=2N 的树深上。实现这个功能的最好方法是将线程化操作的起点定位在某个深度 δ 上。条件是，在深度 δ 上的节点数量与 CPU 的数量相等。[5]

虽然编译程序会根据 CPU 的数量来决定生成的线程数量，但是它并不负责确定在什么地方来产生这些线程。新的算法（参见程序清单 3.5.2）利用改进的数据分解，一旦在 query()

[4] 在撰写本文之际，业界预计这些产品会在 2005 年中开始发售。

[5] 在 Microsoft Win32 API 中，为了得到 CPU 的数量信息，可以调用 Platform SDK 的函数 GetSystemInfo(...)。

中到达了适当的深度，就开始执行对线程的调用。

程序清单 3.5.2　一个 2 路线程化的决策树实现[6]

```
#include <omp.h>
...
void DecisionTree::Query(){
      QueryBinaryTreeFirstTime(m_pNode);
}
void DecisionTree::QueryBinaryTreeFirstTime(TreeNode*currentNode){
      // 错误检查，否则就缺省地在 currentNode（当前节点）上提出问题
      // 否则就缺省地在 currentNode（当前节点）上提出问题
      AskFirstQuestion(CurrentNode);
}
void DecisionTree::QueryBinaryTree(TreeNode*currentNode){
      // 错误检查，否则就缺省地在 currentNode（当前节点）上提出问题
      // 否则就缺省地在 currentNode（当前节点）上提出问题
      AskNextQuestion(CurrentNode);
}
void DecisionTree::AskFristQuestion(TreeNode*node){
#pragma omp parallel sections·
{
      if(answer=="yes")
#pragma omp sections
             QueryBinaryTree(node->m-_pYesBranch);
      else if(answer=="no")
#pragma omp sections
             QueryBinaryTree(node->m-_pNoBranch);
}
      else // 输入错误
              AskNextQuestion(node);
}

void DecisionTree::AskNextQuestion(TreeNode*node){
      // 错误检查
      if (answer=="yes")
             QueryBinaryTree(node->m-_pYesBranch);
      else if(answer=="no")
             QueryBinaryTree(node->m-_pNoBranch);
      else // 输入错误
              AskNextQuestion(node);
```

为了考察前面夸下的海口：节省 10%的 AI 计算周期，现在对新算法进行有效负载测试。使用的是一个平均帧速为 30 FPS 的渲染负载样本。简单计算一下，这相当于每帧大约 33 毫秒（这里忽略了另外 0.3333 毫秒的帧处理时间）。1 秒的 10%是 100 毫秒，所以这就应该是在 30 FPS 的情况下，每秒钟分配给 AI 周期的时间，相当于每秒大约 1/3 帧。这绝对不是太多的时间。所以，这个测试在 30 FPS 的帧速下，每 33 帧就产生一个 AI 周期。这可以保持稳定的线程同步。线程同步由 OpenMP 的内部线程池（thread pooling）机制来初始化。

[6] 用 Intel Complier 的/Qopenmp 选项进行编译的。

下一步要考虑一个与测试负载相关的决策集。在一般的游戏中，每个 AI 单位中同时会有很多处于激活状态的决策树。假设每个 AI 单位可以做出 10~200 个决策（节点），这当然完全是假设了，因为一个强健的 AI 系统可以有更多的选择（特别是采用 N 路决策时）。这个数量范围的几何平均数大约是 45。最坏的情况是，每个场景中有 300 个 AI 单位（也许是一个大型的即时战略游戏），这样，每个场景总计就有 13 500 个节点（或者说是决策）需要进行决策——这个数量实在是太大了，无法在 1 秒种内进行解析。测试平台是一个主频为 3.2 GHz 的 Pentium 5 系统，打开了它的超线程功能（参见图 3.5.3）。

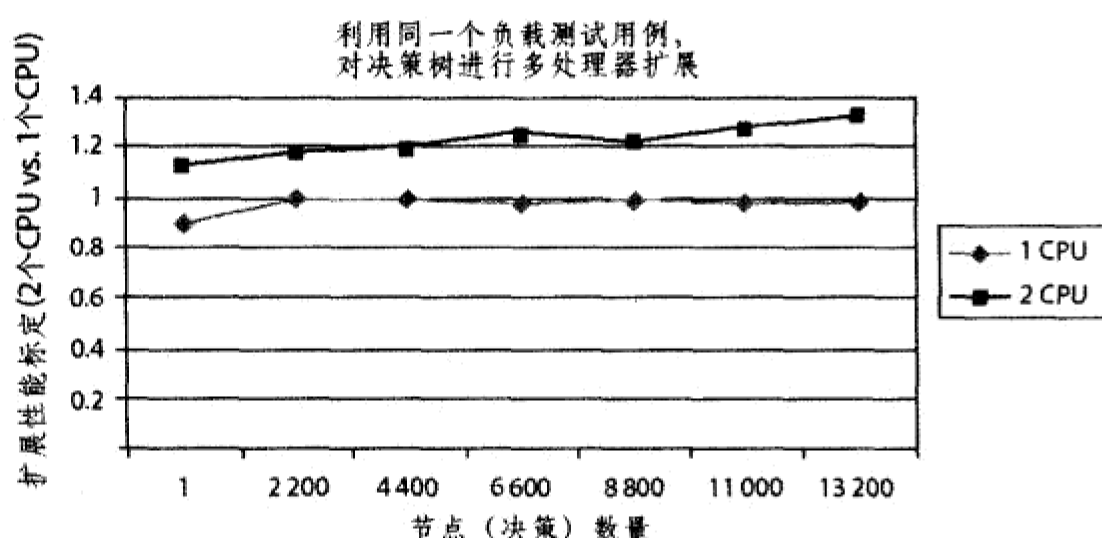


图 3.5.3 决策树负载测试的结果

请大家注意程序清单 3.5.2 中的 OpenMP 指令（directive）。在这个例子中，操作系统会为代码中每个 section（节）调度两个线程。要实现这样的功能，需要在程序代码中包含 OpenMP 头文件（omp.h），并在编译的时候使用/Qopenmp 编译选项。编译预处理指令“#progma omp parallel sections”则会通知编译程序，下列复合语句是可并行处理的代码。而且可以看到，每个工作单位（work unit，在这里就是指函数的调用）都是用“omp section”指令来标识的。有关这个指令和其他 omp 指令的详细内容，请大家参考 OpenMP 规范说明。

3.5.4 总结

本文介绍的方法向大家展示了如何修改一个现有的决策树搜索算法。对于决策树模型的应用，可以查阅列出的参考文献，以获得更多详细的信息。这里使用的 OpenMP 编码技术不但适用于数据集极其庞大的情况，也适用于那些决策树尽可能均匀平衡的场合。至于可以获得的算法性能的提升，不但取决于决策树的平衡情况，也取决于 CPU 的缓存资源，以及特定场景下搜索算法被调用的频率。正如这里向大家所展示的，为了能够在 SMP（对称多处理技术）系统中获得真正的实惠，通常要对算法的架构进行微小的重构，以便进行负载平衡。另外要注意的是，鱼与熊掌，不可得兼。伴随着代码复杂度的增加，算法对内存的需求也大大增加。

3.5.5 参考文献

Decision Trees. Available online at *http://www2.cs.uregina.ca/~hamilton/courses/831/ notes*

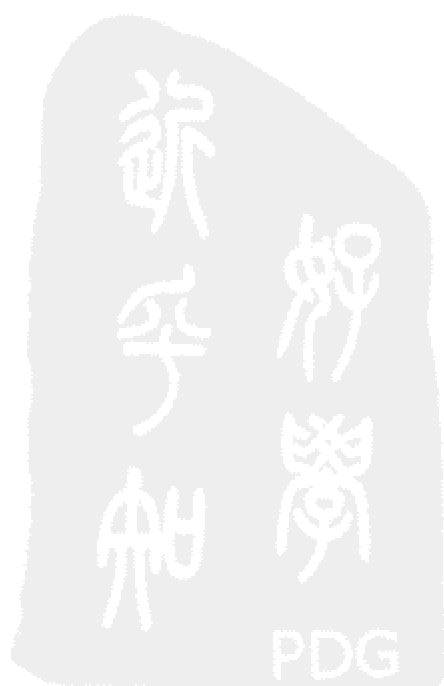
/ml/dTrees.

Generation 5 Implementation. Available online at *http://www.generation5.org/content/2004/bdt-implementation.asp*.

Intel®C/C++Compiler Users Guide and Reference. Available online at *http://www .intel .com/software/products/compilers/cwin/*.

Microsoft Visual Studio 2005 OpenMP Support. Available online at *http://msdn.microsoft.com/msdnmag/issues/04/05/visualc2005/default.aspx*.

OpenMP Specification 2.0. Available online at *http://www.openmp.org/specs/*.



3.6 利用并行虚拟机实现 AI 系统的并行开发

2015 公司，Michael Ramsey
miker@masterempire.com

对于游戏中的并发处理系统或并行处理系统，芯片技术发展的最新趋势为这些应用提供了基础架构支持。作为一个软件系统，并行虚拟机（Parallel Virtual Machine，简称 PVM）为 AI 系统的开发人员提供很好的支持，让他们能够编写出可在多处理器平台和多种操作系统中运行的 AI 系统。本文将向大家介绍 PVM 开发过程中一些相关方面的问题，内容重点是即时战略游戏 AI 系统的开发，目标架构则是 Windows 系统或 Linux 系统。

虽然本文的目的，是为了告诉大家如何使用 PVM 进行 AI 系统的开发，但是其中的很多概念和框架都可以扩展到其他领域，使之可以从并行化技术中受益。这些概念和框架还可以应用在那些采用多线程并发技术的平台上，如 Intel 公司的超线程技术，或未来几代的游戏机平台。

3.6.1 功能强大，但不白给

一个并发式的 AI 系统通常会被打上平行系统或分布式系统的标记。并行虚拟机（Parallel Virtual Machine，简称 PVM）则是二者兼有。通过 PVM，一款游戏既可以使用一个机器中的多个处理器，也可以使用一个局域网中的多台机器，从而强化玩家的游戏体验。

PVM 开发中一个关键的诉求，是广泛地使用消息传递模型，以便有效地利用分布式系统环境。但是，为了使用这个消息传递机制而对现有的系统进行翻新改造，并不会带来更好的游戏性，也不会产生更智能的 AI 系统。为了能从 PVM 中受益，必须从一个通用框架入手，去考虑开发一个并发式的 AI 系统。这个前期工作，可以让那些被下传的 AI 任务与游戏程序的核心部分进行通信，并一起工作。只要有了一个最基本的框架，AI 系统开发人员就可以让他的智能主体们不断地进行评估，采用最佳路径；或者让一个独立的 AI 任务不停地评估战术方法，并采取最佳策略，逐步逼近那些到处躲藏的玩家。把所有的 AI 询问都构造成 AI 任务，并用其他的处理器来执行，系统就可以在后台不间断地执行这些 AI 任务。没有哪个 AI 开发人员会愿意将他的 AI 系统完全托付给一个依赖于帧的“夹子”。

3.6.2 核心术语及概念

在讨论 PVM 给游戏带来的好处之前，首先需要定义一些专门的用语。这里不会带领大家去回顾范型并行编程技术，重点是介绍几个必需的组成部分，来高效率地开发一个 PVM 框架，如图 3.6.1 所示。

关于 PVM 更深入详细的讨论，请参阅[Geist1994]。

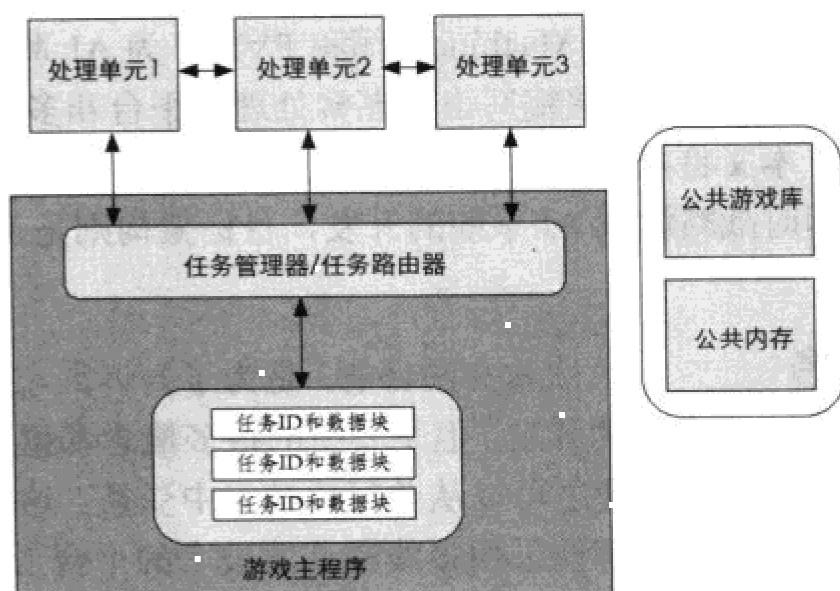


图 3.6.1　一个基于 PVM 的 AI 系统及其组成部分

所有的任务都发生在处理单元（简称 PE）中。主板上的处理器，或者局域网中的计算机资源，都属于处理单元。可以把游戏主程序理解为“主人”，而把所有可用的处理单元理解为它的“奴隶”。这样，游戏主程序和附属的处理单元之间的关系就清晰可见了。处理单元并不主持工作，它们只在那里为游戏处理一些任务。

由一个庞大的问题可以细分出很多的“任务”，例如路径生成、确定安置游戏单位的空间位置、造墙、经济分析，及其他与游戏相关的行为。

任务分解

并行 AI 系统开发的一个基本部分就是任务分解的概念：将一个任务分解成若干个子任务，并分别交由不同的处理单元来处理。这个架构可以将一些基本任务进行分解，并可以很容易地由多个处理器来处理。

任务的分解通常有两大类：功能分解和数据分解。数据分解（*data decomposition*）是任务分解的一个算法方法，也是这两大类分解方法中最为复杂的一种。功能分解（*functional decomposition*）则是一个比较简单的方法，因为一个被封装的功能可以传递给另外一个处理器来执行。虽然这个方法需要转变处理 AI 系统的设计与实现的方式，但是 PVM 系统仍然采用这个方法来进行游戏的处理。

在功能分解方法中，每个任务会在另外一个处理器上触发一个特定的操作。举个例

子，如果要执行一个后台任务，分析潜在敌人的移动方向，那就可以创建一个任务，利用一些预先安排的参数，来启动一个游戏单位（unit）的分析程序。这些参数会传递一些信息，如优先目标评级、先前的外交约束、游戏单位对军队编制的偏好等。在设计这个分析任务时，开发人员必须深思熟虑，想清楚这个任务的各个方面该如何在并发系统中执行。

3.6.3 任务的创建

为了向大家说明如何将一个大问题分解成若干个小问题，首先来看看即时战略游戏 AI 系统中常见的一个问题（参见图 3.6.2）：从更具认知性的长期行为中，区分出毁灭性的短期决策。

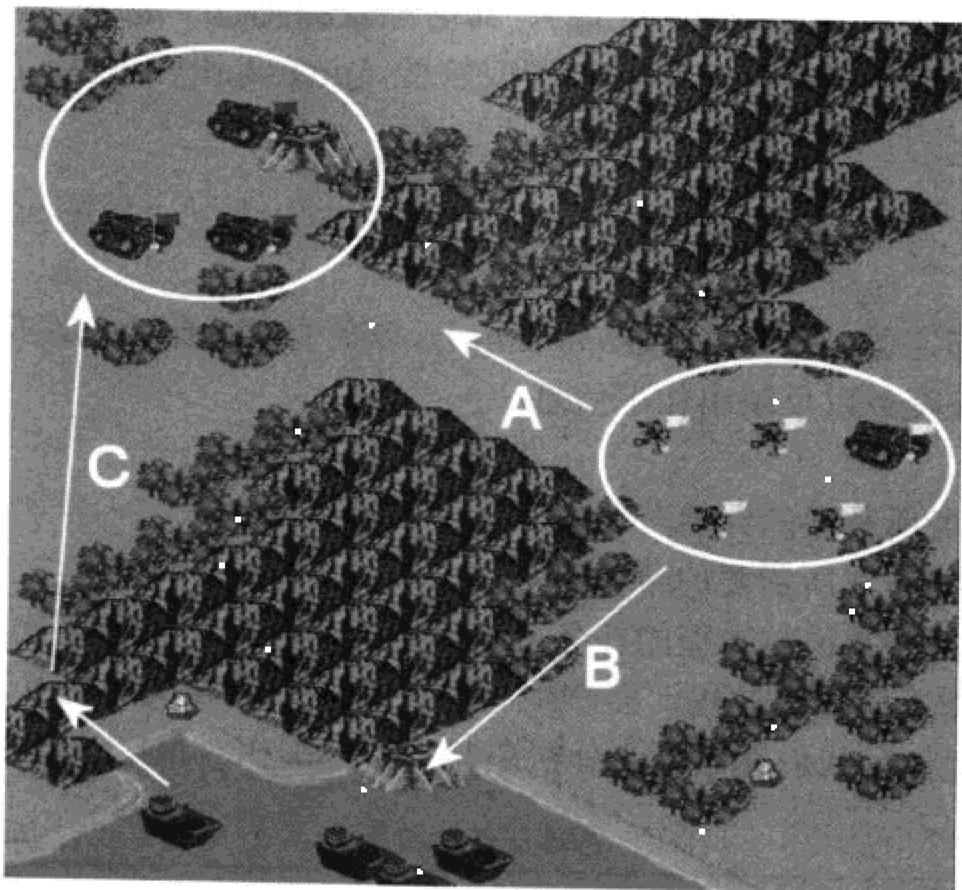


图 3.6.2 一个即时战略游戏中的场景实例

在图 3.6.2 中可以看到，一个 AI 主体发现了一座敌人的城市，周围布满坦克。在这个例子中，有一个玩家在山的西坡聚集着军队，而 AI 系统则相应地在山的东麓驻扎着几个特遣队。AI 系统试图侵占这个交战国的城市，那它应该选择最直接的路线（A），还是另外一条路线（B 到 C）呢？

理想情况下，AI 系统可以做出一些带有预见性的反应。在策略层面上，假设 AI 系统已经知道了它需要拿下这座城市。为了达到这个目的，AI 系统需要评估可用的路线，然后形成一个计划。

这里有三种可能的计划：

计划 A： 让军队从山隘中通过。

计划 B： 让军队先开进到港口城市，然后再将它们运到南翼。

计划 C： 让部分军队从山隘中通过，吸引敌人注意力。而主力部队则执行计划 B。

PDG

这三个计划都有一些类似的组成成分。让我们看一下计划 C 的组成部分，它包含了计划 A 和计划 B 的组成部分：

更新影响地图（influence map）：在这个特殊的例子中，AI 系统需要知道它的兵力是否可以在被攻击之前调动通过这个据点。这需要一个最新的影响地图。

为作战单位生成路径：对于走进山隘的作战单位，以及那些通过 B 到 C 的路线行进的作战单位，需要为它们提供移动路径。

提供运输支持：要么就在城市里建造交通工具，要么就调度交通工具，辅助兵力调动。

提供攻击编队：一旦到达山脉的西麓，就开始指挥这些作战单位。

评估外交约束：确定在攻击西坡城市之后，会产生什么类型的政治余波。

只是为了确定 AI 系统应该如何展开攻击，就已经生成了几个潜在的任务。除了这些新产生的任务，还有高级的、幕后的战略规划和正在处理的其他任务。有些任务可以快速、容易地计算出来（通常都在本地处理器上完成）；而有些任务则过于庞杂，需要分派给其他的处理单元。正如将要看到的那样，任务的大小或粒度（granularity）规定了每个任务被调度的方式。

粒度

每个任务都被指定了一个粒度（*granularity*），以便有效地确定它在任务管理器中的调度排位。一个任务的粒度可以反映出如下信息：

- 任务的大小
- 完成这个任务所需要的计算开销
- 达成一个解决方案所需要的带宽

PVM 通常更适合处理那些比较大型的任务。与任务的实际处理过程相比，任务的创建过程，也就是向任务池添加新任务的过程，会产生相当可观的系统开销。所以，为了能够向 PVM 提供更适合处理的任务，我们通常只会分配那些工作范围相对很广的任务，如表 3.6.1 所示。

表 3.6.1　　各种任务的粒度

任务示例	粒度
影响地图分析	大
生成游戏单位的 LOS（视线范围）	中
造墙	中
作战小分队（多个游戏单位）的寻路	中
周围单元的分析	小
一个游戏单位的寻路	小
战术影响地图分析	小
组合军队编队	小

虽然也可以手工协调多个任务的执行，但我们还是根据开发过程中对每个任务的观察，基于每个任务的处理时间，自动地为它们指定一个粒度。每个任务的执行时间是参照一个主计时表，为这个“类型（type）”的任务指定一个粒度。然后，这个任务的引用就会保存在任务管理器的任务粒度表中。当新的任务被执行时，这个粒度表就会被更新。在运行时分析每个任务执行时间的长度，就可以把相关的任务归为一组，或把它们链接起来。

PDG

3.6.4 任务管理

一旦创建成功，就把新的任务放到任务池（*task pool*）数据结构中，然后再分配给任务管理系统和任务路由系统。根据正在执行的任务的不同状态，从任务池将任务分配到相应数量的处理单位，这个时间可能会非常长。所以，在设计任务时一定要考虑到这一点，为任务指定合适的粒度，并创建相应的分配算法。

在处理单元之间进行任务分配取决于两个因素：

- 可用处理器单元的数量
- 适合进行分布式处理的任务类型

继续前面的例子，看一下攻击任务的分配到底受哪些因素的影响。在一个只有两个处理器的系统中，一个处理器应该作为辅助处理器处理那些高度或中度复杂的任务；而另外一个处理器则负责处理比较小的任务，或者那些需要立即执行的任务。这个处理器同时也要负责游戏主程序的执行。

更新整个影响地图，或者为一个作战小分队生成最佳路径（包括对潜在的军队编制问题的处理），这些任务应该分配到辅助处理器中去执行。但是对于一些关键任务，例如为一个小分队的作战单位提供的寻路支持，为什么还要下传（offload）到附属处理器上执行呢？难道不应该要求 AI 系统立即生成最佳路径吗？当然需要了，但是，对于一个完整的任务行为规划，这只是一大堆需要处理的任务中的一个任务而已。

并不是所有的 AI 任务都需要进行分布式处理。有些任务实例确实需要在本地处理单元上立即执行，这是由 PVM 架构负责的。例如，如果游戏运行在一个双 CPU 系统上，PVM 会按照惯例为这个任务指定一个比较高的优先级，这样任务管理器就马上进行调度，安排这个任务在本地执行。再举个例子，如果有一个掉队的士兵受到敌方的攻击，而他自己又无法还击，不能保护自己，那他只有尽快逃离。快速地找到一个可以躲避的地方，这个任务的优先级就会高于该士兵的潜在长期目标。这种“非战即逃”的反应纯粹是一种反应性 AI，此类任务应该马上被调度执行。一旦危险已经过去了，AI 单位才能继续他的长期目标，这个长期目标更适合让辅助处理器来处理。

1. 依赖性

任务的依赖性是任务机制一个潜在的副产品。它是指某个任务的执行依赖于前一个已执行任务的输出结果。还是使用前面那个即时战略游戏的例子，AI 主体不但需要了解到达敌方城市的路径，还需要了解这个城市周围潜在敌人的移动情况。这里就需要一个任务，来分析敌方军队的部署情况。于是，我们就使用最新的影响地图（influence map）来评估、分析敌方的实力以及它们的历史移动记录，并制定出 AI 系统相关的响应。对付这种依赖关系链的最简单方法（实际上就是最简单、合理的调度方案）就是忽略它。在这种情形下，AI 系统的精确程度取决于影响地图和分析算法中更新任务的（强制）排序。

强制调度虽然简单，但是效率低，而且容易出错。对于性能要求很高的应用程序，比如游戏，我们必须考虑任务与相应的调度算法之间各种可能的关系。是否应该采用预留策略（Reservation Policy），提前几个帧的时间来调度某个任务？这样可以确保那些重要的

任务能够得到及时、完整的处理，并预留一些空闲处理单元来处理那些不太重要，但却非常庞大的任务。如果采用预留策略，是否应该中断任务的执行？后面的文章中将会讨论“任务中断”。

2. 分组

一个简单的方法是将类似的任务组成一个执行包。这些任务都比较小，合并到一起正好适合在一个处理单元上执行。这种分组方法可以避免任务初始化的无关开销，减少带宽的浪费，而且通常可以更有效地使用处理器资源。另外，当几个类似的任务在同一个处理单元上执行的时候，处理这些任务所需要的数据对处理单元而言仍然是局部性的。数据的局部性（data locality）可以节省内存读取和通信的开销。图 3.6.3 展示了一个例子，类似的任务被合成一组。

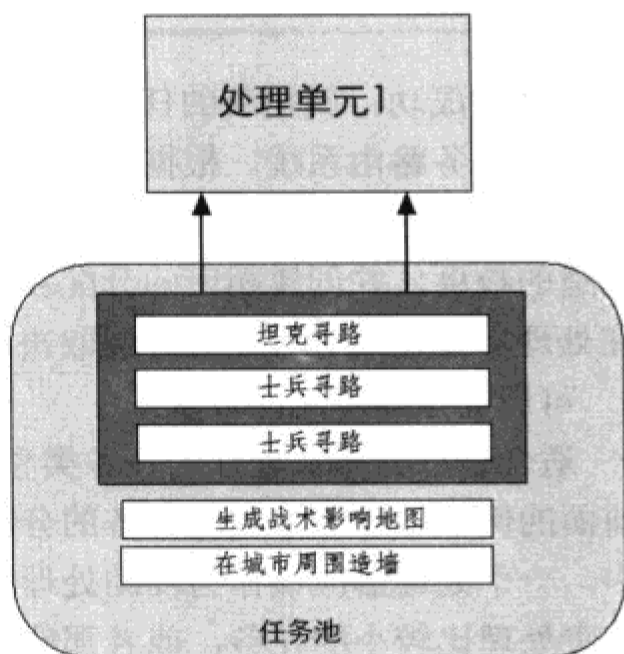


图 3.6.3 在进行分布式处理之前，将类似的任务集结成组。在这个图中，三个类似的寻路命令归并为一组，以便提高效率

任务的动态分组需要定义一个管理类（management class）。它负责对任务进行评估并按类型进行分组。这个类处于任务池和通用的任务管理器之间，其用途就是作为一个选择机制，抽取合适的任务，提交给任务管理器。

为了方便任务的灵活执行，可以调整某个操作的粒度，将它分散给可用的处理单元。虽然这么做是有好处的，但是有些任务（比如，重新生成一个影响地图、为一个 AI 单位的部署计算空间位置，以及潜在的地形分析等）在进行重构时，会导致进程间通信量的增加。对于那些非常倚重数据的任务，它们的数据都是预先计算好，并保存在共享内存中的。这类任务通常都会增加进程间的通信量。

内存共享是进程间通信最常见的机制，可以让一系列的处理器单元访问同一个内存池。当某个处理器单元改变了共享内存池，内存池就会更新所有的处理器单元。在 Windows 系统和 Linux 平台上，必须使用信号量（semaphore）来提供访问共享内存的同步机制。

3. 链接

作为任务分组的一个辅助变量，链接是非常有用的。当大型任务要依赖小型任务提供初始化输入时，链接就显得尤为重要。在 task ID 的数据结构中，有一个预留的 unsigned integer 字段，用来链接其他的任务包。任务调度程序会留意这个链接。当一个辅助包被路由到一个处理单元时，调度程序就会选择这个链接所指的任务包，准备投入执行。必须小心使用链接的任务包，因为它们对同样可以导致线程泄漏的竞争条件（race condition）和死锁问题特别敏感。

虽然任务调度程序实现起来非常简单、直接，但是对于单个任务的选择机制，不同的游戏有各自不同的实现方法。最关键的一个组成部分就是数据处理。如果处理单元处于“饥饿”状态（等待数据），那么在诸多AI例程中，那些负责向任务池提供任务的例程就要考虑这个情况了。一旦发现处理单元出现“饥饿”状态，就一定要重新修订AI例程和调度算法，这样可以更早地预见潜在的任务处理阻塞的现象。

4. 缓存和处理器关联性

在将任务下传给处理单元的时候，有一点很重要，它是必须考虑的一个因素——缓存和处理器关联性（affinity）。这个特性可以把类似的任务放到曾经执行过相同或类似任务的处理器上执行。为了适应这个特性，任务管理器应该包含一个基本的历史跟踪系统。该系统包含一个差异任务列表（其中的各个任务都各不相同），以及最近负责执行这些任务的相关处理单元。

5. 中断

一旦某个任务被路由到一个处理单元去执行，任务的执行就不能被中断，这是很关键的一点。要让任意一个被下传到处理单元的任务彻底执行完毕，然后再把其他的任务下传给这个处理单元。这样做是有好处的。如果因为一个优先级更高的任务进入到任务管理器中而中断正在执行的任务，那就必须终止进程的执行，保存整个处理单元的状态，这个机制实在是太过笨重了。如果发现某个任务的执行确实需要被中断，那很有可能是因为这个任务太大了，需要将它分解成若干个较小的任务。

在即时战略游戏中，一个典型的例子就是影响地图的生成。如果只是一个单独的AI单位去攻击敌方的坦克群，就没必要生成整个影响地图。由于大部分的影响地图尺寸都比较大，所有生成工作很可能会成为正常游戏性的瓶颈。取而代之的是，可以使用一个战术影响地图（*tactical influence map*）。战术影响地图只是一个较小的影响地图，它只关注一个特定的目标区域。生成大型影响地图的正常操作仍然会执行，只不过生成地图的比例更小了，也更合适了而已。不必去限制任务的范围，只需要为它设计一个更小的数据集，这样就可以降低任务被中断的可能性。

6. 负载平衡

在创建需要执行的任务时，必须考虑某个处理单元能够处理的工作量的实际大小。如果分配的工作量太小，那么处理单元就会处于“数据饥饿”状态，即实际处理任务的时间没多少，更多的时间都浪费在等待和通信工作上。而如果任务太大，要占用太长的执行时间，就会导致数据依赖问题。更糟糕的是，这样就无法及时将关键的结果传递给游戏，如此一来，决策过程必然会产生一个断层。

要想充分利用PVM架构的优势，必须让所有的线程都保持在忙的状态，这是很关键的。任务完成的顺序并不重要，但是有些情况一定要避免，如某个处理器单元成为另一个处理器单元的直接输入者。数据依赖会导致线程阻塞（stall），如果不理会这个问题，任其扩散，最终会把这个并行AI系统变成一个顺序处理的AI系统。

前面那个即时战略游戏的例子，其中涉及的任务都可能被设计成顺序依赖的任务。正如

本文"依赖性"一节所提到的，现在有一系列的任务来构成 AI 系统的一个规划。这些任务包括：更新影响地图、评估 AI 单位的位置、分析 AI 单位的实力和跟踪历史移动记录。当然了，对于一个真实的即时战略游戏，还可以创建出无限多的任务。假设把上面四个任务组成一个更大的任务，然后提交执行。可以看到，更新影响地图的任务必须首先完成，然后 AI 单位评估的任务才能开始执行。这样就会产生一个很讨厌的副作用：必须等另外一个任务最终完成了，某个处理单元才能开始执行自己的任务。这种延迟会给任务管理器中其他排队等待的任务带来某些不良影响。队列中的任务就在那里等着，而前面的任务则阻塞了整个处理管线。

7. 线程池

绝大多数的任务都是由任务管理器来管理的，并且要创建线程，以便在各个处理单元上执行相应的任务。线程池可以在运行时创建一组线程，任务管理器可以访问这些线程，并提供给处理单元来执行。这种做法转移了连续线程及创建堆栈工作的系统开销，把这些开销变成了系统启动时的一个一次性操作。

一个普通的并行处理过程面临的问题包括：死锁问题、同步问题、资源颠簸（resource thrashing）和线程泄漏。而线程池也同样面临着这些问题[Gerber04]。

对于线程池的使用，一个重要的方面就是我们可以完全相信线程池的大小是正好合适的。如果仔细地调整每个游戏任务可用的线程数量，就可以避免系统资源的浪费；而且，通过对过量的线程请求进行排队管理，还可以减少或者彻底消除线程泄漏。

3.6.5 PVM 的实现

设计并实现一个并行友好（parallel-friendly）的游戏是要费些功夫的。我们必须重新评估基本假设（通常还包括之前在 AI 系统实现方面所积累的经验）。

将设计方案模块化，可以更容易地从顺序处理过渡到并行处理。创建若干个公共游戏库，让所有的组件（处理单元，或是游戏的主程序）可以访问这些库。我们的目标是构造一个模块框架，让所有的处理单元（包括游戏的主程序）可以共享公共代码，而不需要针对每一个处理单元去编写特殊的实现。

Wrapper（包装器）的使用也可以提高实现的模块性（modularity），使代码远离那些 PVM 专用的调用。虽然 PVM 支持字符串（string）、整型（integer）等类型的基本打包（pack）和解包（unpack）机制，我们仍然不得不编写几个自己的打包机制。这些游戏专用的数据包装例程可以提高模块性，可以创建由 wrapper 自动打包或解包的数据类，不需要对 PVM 系统有任何直接的依赖。如果游戏是在一个单处理器的系统中运行，它们还可以帮助把同样的游戏进程平稳地运行下去，不会有任何的中断。

这个设计范例包括一个基类（用于处理一些基本行为）和几个并行实现。这样的设计使得游戏的目标系统既可以是单处理器系统，也可以是支持 PVM 的系统，如图 3.6.4 所示。

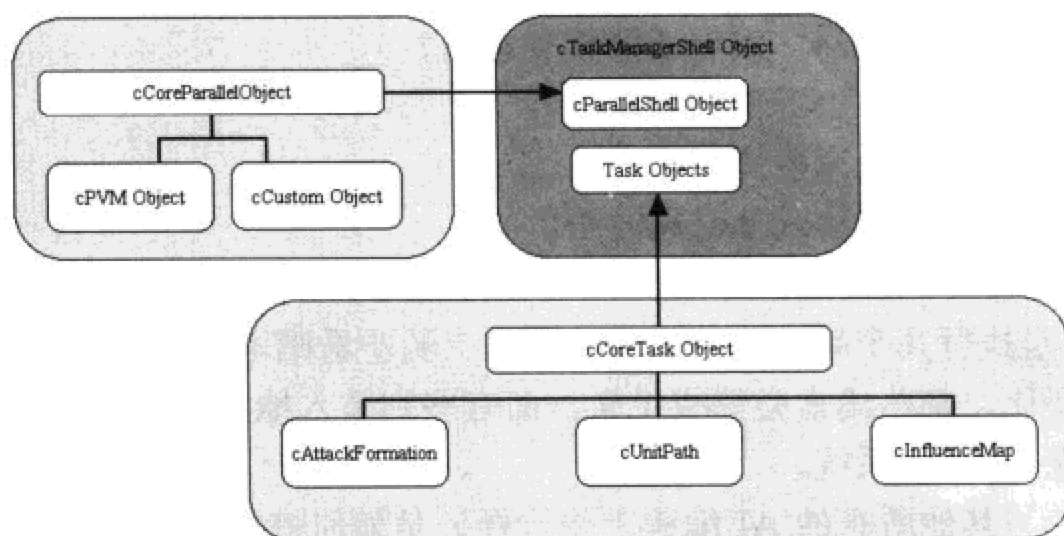


图 3.6.4　一个并行 AI 系统范例的类图。对象管理器外壳（cTaskManagerShell）驻留在游戏 AI 的工作空间中。对象管理外壳包含着并行外壳对象（cParallelShell）及核心任务对象（cCoreTask）。这两个类都是通过 factory（工厂）方法创建的

PVM 有一个简单的资源管理器，可以处理多个任务的布局。应该把这个简单的资源管理器只当做一个简单的任务路由机制来使用，向可用的处理器分配任务。任务管理器/路由器的实现中应该包含这样一个复杂的、游戏专用的资源管理器。这样就可以清楚地知道哪些处理单元最近执行了相同或类似的任务。于是，对先前缓冲数据的评估就派上用场了。数据的评估通常都存储在共享内存中，以期提高处理速度。

PVM 使用用户数据包协议（User Datagram Protocol，简称 UDP）来处理进程间的通信。如果要自己定制的话，可以考虑增加一个容错系统。即使与一个正在处理任务的处理单元失去通信联系，这个增加的容错系统也可以让游戏继续执行下去。有了这样一个基本的系统，任务管理器/任务路由器就可以核实某个任务是否被过早地终止了，或者一直处于终止状态。当出现这种情况时，系统就会终止当前的任务，重发这个任务。验证某个任务当前状态的系统开销是很小的，远远小于准备一个新任务的系统开销。

3.6.6 实际应用：即时战略游戏

既然已经拥有了一个 PVM 系统需要的所有组件，如果把原先那个普通生产型的 AI 系统转变成一个全新的 AI 系统，又会发生什么变化呢？这个全新的 AI 系统看上去好像什么事情都能想在前头，其行为反应更接近现实世界，而且还能预测玩家的行动。那就看一个典型的即时战略游戏的例子吧。

有两个交战国在一座孤岛上对峙。这个孤岛只是游戏世界中大量陆地中的一小块。在一个典型的顺序处理的 AI 系统中，level 启发式将会顺序做出判断，是否应该发起攻击（以及是否可以在攻击中存活），某些游戏单位是否需要支持，以及这些支持又该来自于哪里？

在一个基于 PVM 的实现中，AI 系统可以在后台进行局势分析，预先计算出可能发生的各种游戏局面，当需要做出相应的动作时，就从中进行选择。由于有了正确的预见性动作，AI 可以对当前的局势进行推演，对玩家未来的战略和战术做出更为有效的反应。

具体地说，AI 系统可以提出以下问题：

- 是否输得起这座孤岛？

- 如果马上丢掉这个孤岛，后果会是什么？
- 如果现在丢掉这个孤岛，以后还能收复它吗？
- 如果想收复该岛，该去哪里招募军队？
- 军队登陆岛屿的速度有多快？
- 在同盟国中会引起政治或外交风波吗？
- 会引起什么样的风波呢？

这个 AI 系统并不是执行几个顺序化的评估，而是“骑驴看唱本”，可以同时追踪多条推理线索。随着局势的变化，某些线索会修成正果，而有些则坠入地狱。但总体上看，这个 AI 系统更像一个准备充分的 AI 系统。

真正的工作（与其他所有的 AI 编程工作一样）是如何将相对模糊的策略询问（query）翻译成具体的计算动作。像“是否输得起这座孤岛？”这个问题，实际上就是评估 AI 当前的军队和作战单位的实力、所把持的土地，以及外交立场，并将这些与对手的各项等级排名进行比较。

这些比较是简单的、自发的，但如果这样的对比数量堆积起来就会非常强大。伴随着十来个其他的指导分析任务，它们创建的 AI 可以不断地评估最重要的场景：*what-if*（假设分析）场景。

3.6.7 强化游戏性

虽然有了一个能在后台执行 what-if 场景分析的 AI，但这不一定就意味着 AI 系统会给玩家带来难度更高的游戏体验。不过，一个正确开发的基于 PVM 的 AI 系统确实是有这种能力的。这样的 AI 系统还有另外一个优势：可以开发出更为复杂的模拟器，来与玩家交互。再来看看典型的即时战略游戏的例子。

1. 外交

在一个典型的即时战略游戏中，当玩家与 AI 主体展开政治谈判时，一般的 AI 系统都是根据事先安排好的响应来给出答复。这些响应通常都是基于有限的启发式数据，再根据敌国的统计数据（军力、敌军的作战单位与 AI 的城市或基地接近的程度等）生成出来的。

如果 AI 可以评估出玩家潜在的动机，那么它所创建出来的政治环境，就能够更好地对玩家的行为做出灵活的响应，也更能表现出接近真实世界的政治环境。采用本文介绍的技术，AI 系统的开发人员可以编写出一系列的评估任务，并根据当前发生的事件来执行相应的评估任务。

假设玩家与一个 AI 对手展开一场外交谈判。玩家并没有什么作战单位部署在 AI 的基地附近，但是玩家已经暗中逐渐地、不断地从多个国家向 AI 的基地调动军队。AI 在后台执行着一个任务，来评估敌方军队的行踪。通过跟踪记录敌方军队的移动情况，AI 就可以确定玩家正在从地图中多个不同的位置集结军队，有入侵 AI 基地的潜在可能。

这种类型的评估逻辑在外交谈判中一定会起到重要的作用。于是，AI 就可以提议，让玩家削减某个国家的兵力。因为这个国家对 AI 的一个战略基地威胁最大。该任务可能只是所有外交事宜中很小的一部分，也只是消息灵通、有着精确响应的谈判逻辑的一小部分。

通过后台任务的执行，基于 PVM 的 AI 还可以评估玩家的意图，为达成一揽子的长期外

交协议做出贡献。就好像真实世界中的谈判那样，AI 会评估另外一方在过去的一些行为，这些行为对当前局势又有着怎样的影响，以及当前的这些决策会如何作用于未来的行为。

2. 生产

对于一款标准的即时战略游戏的开发，其另外一个相当明确的特色，是让 AI 系统有能力管理游戏单位的生产、收获食物、研究与发展，以及其他经济活动。所有这些功能的实现都作为理想的后台任务，在其他处理单元上执行。而中央控制机制则驻留在游戏的主程序中。

在一个典型的即时战略游戏中，核心的游戏单位担当着收获某种特定资源的角色。然后，这些资源被运送到不同的工厂，生产出不同的游戏单位或补给。可以编写一个 PVM 任务，对这些工厂进行资源流量分析。这个资源流量分析的工作内容主要是确保每个工厂都有适当的资源，可以用来生产或制造某个特定的游戏单位或其他资源。每个工厂还可以有一个直接或间接的消耗者，完全依赖于该工厂生产的产品。

假设有个面包师，他需要的是面粉，以便生产出烘烤食品。他赖以生存的是一家面粉厂。PVM 资源流量分析任务会确保不断地给这个面包师供应面粉，以便让他可以为这个小镇提供面包。该任务会评估这个面包师过去、现在和将来的需求和产出。如果该任务发现生产系统中某个生产环节有所欠缺，它就会通知任务管理器/任务路由器，让它们启动一个任务来处理工作进度。这个新的任务会制造出更多的收获单位，帮助面粉厂提高产量。

此系统不但可以用在游戏 AI 上，还可以帮助玩家。很多即时战略游戏里都有一些大臣或总督，他们负责管理和控制城市或基地。驱动 AI 系统的这些逻辑同样可以用来运营玩家的城镇，而且只需要很少的 CPU 开销。现在很多 AI 系统都可以实现这些任务，但是如果能让这些任务在后台运行，就可以为其他新的任务释放出更多的传统资源，为玩家提供更具沉浸感、更完整的游戏体验。

3. 其他领域

在传统的第一人称射击游戏中，玩家通常都希望 NPC 可以尽快找到逼近玩家的路径，与玩家展开对战。为了给 NPC 增加一些虚幻的真实感，或许可以在他们逼近玩家的路上执行一些用脚本文件定制的行为。

通过并行 AI 系统，可以在后台不间断地处理这些事情：

- NPC 相对于玩家的威胁等级
- 玩家对场景中其他 NPC 的威胁程度
- 哪个 NPC 可能会需要支援
- 可能的侧翼包抄移动
- 恰当地使用掩体，或使用干扰术，分散敌人的注意力

NPC 可以做出的决策是无穷无尽的。将如此大量的潜在行为进行并行处理，重复决策的可能性就不存在了，这样就可以为玩家提供真正独特的、意外突发式的、自由发展的游戏体验。

3.6.8 总结

创建一个智能的、快速反应的、有先见之明的 AI 系统，是一个颇具挑战性的任务。在

PDG

并行 AI 系统开发的介绍文字中，我们向大家讲述了实现一个基于 PVM 的 AI 系统所必需的基本原则，并向大家展示了这个 AI 框架是如何进行任务的分派，如何提供并发的、完整的后台评估机制的。利用这个基础框架，AI 开发人员可以为对手和 AI 主体增加更多的深度，提高 AI 系统的战略机会，也可以为使用者提供功能更强大的管理工具。

人们通常会认为，AI 开发人员专注的是 AI 系统“人工”的一面，试图利用烟雾、镜子和门面装饰，营造出虚幻的智能行为。我们坚信 PVM 绝对不只是装饰品，它所提供的平行处理技术将会帮助开发人员实现 AI 系统“智能”的一面。虽然 AI 系统的并行实现要求我们远离标准的技术和工具，但是随着桌面和游戏机多处理器系统的逐渐普及，所有这些努力都是合理的，也是值得的。

PVM 应用正当时！

3.6.9 参考文献

[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.

[Ramsey03] Ramsey, Michael. “Setting up PVM for an AI System.” Available online at *www.masterempire.com/OpenKimono.html*. November 10, 2003.

[Ramsey04] Ramsey, Michael. “PVM Gotchas!” Available online at *www.masterempire.com/OpenKimono.html*. March 15, 2004.

[Tozour01] Tozour, P. “Influence Mapping.” In *Game Programming Gems 2*. Charles River Media, 2001.

[Woodcock02] Woodcock, S. “Recognizing Strategic Dispositions: Engaging the Enemy.” In *AI Game Programming Wisdom*. Charles River Media, 2002.

3.7 超越 A*算法

Xtrem Strategy 游戏公司，Mario Grimani
mgrimani@xtremestrategy.com

Monolith Productions 公司，Matthew Titelbaum
matt@lith.com

大部分游戏产品都使用了某种形式的人工智能（AI），而寻路算法通常都会是这些 AI 模块的一个组成部分。A*算法是一个非常著名的算法，可以用来解决通用目的的寻路问题[Stout00]。A*算法是一种贪婪算法[Cormen01]，综合考虑了两种移动开销。一个是从起点开始到当前位置的移动开销，这也是 Dijkstra 算法所考虑的；另外一个是从当前位置到目标地点剩余的移动开销。后者是通过启发式方法预估出来的。这是目前寻找最小开销路径的最快算法，能够很好地解决基本的寻路问题。

但 A*算法也有很多局限性。其中之一，就是随着搜索空间的增大，算法的性能会骤然下降，而且对 CPU 和内存资源的需求也会急剧上升。这使得它无法解决那些搜索空间较大的问题。更糟糕的是，来自高级 AI 模块的查询会产生数量极大的 A*算法调用，由此也带来了对 CPU 资源的更多需求。由于这些问题的存在，有些游戏只好放弃使用那些最佳的解决方案；而有些游戏则只能放弃使用那些查询密集型的高级 AI 模块。

虽然有一些方案可以解决大型搜索空间[Botea04]和其他搜索空间的问题[Stout96]，但是由于在寻路算法与 AI 其他部分的整合工作上没有太多的建树，它们也无法有效地减少查询开销。如果能够减少开销，就可以增加查询的数量，为高级 AI 模块生成更多的知识。这些新增加的知识可以帮助各种类型的游戏，改善其决策系统中战术和战略的决策水平。

3.7.1 问题的定义

首先，要正式地定义问题域（Problem Domain）。这里将问题域定义为一个连通图，其中的节点表示位置，而承受开销的边则表示在这些位置（节点）之间移动的开销。

在该问题域中执行的 A*算法，大致需要下列几个参数：起始节点、目标节点、启发式估算函数，以及主体移动的条件。起始节点和目标节点是要尝试用最小开销路径连接的 2 个节点。对连接图的遍历过程从起始节点

开始，到达目标节点就算完成。正如已经提到的，启发式估算函数是A*算法的一大特色，它表示的是从当前位置到达目标节点剩余开销的估算值。

最后一个参数——主体移动的条件，并不是每个A*分析的必需部分，但大部分的A*算法实现几乎都包括了这个部分。它代表着主体在节点之间移动的能力，反过来也表示着节点阻挡主体移动的能力。在连接图的遍历过程中，需要考虑这个能力，因为遍历逻辑需要忽略那些移动受阻的节点。

为了叙述的方便，这里将“查询（query）”定义为询问（inquiry）或请求（request）一个由高级AI模块初始化的信息。针对这个“查询”的回答则由底层代码自动生成。不要把这个定义与常见的“查询”定义混淆起来。通常，对“查询”的定义是指对一个数据库或搜索引擎发出的正式请求。当然，对于请求的这些信息，也可以提前把它们计算出来，然后保存在数据库或缓存中，但是这些概念要说明的是如何按需生成相应的信息。这些内容已经超出了本文的讨论范围。

高级AI模块通常运行于多个问题域之上，并针对多个问题域发出查询请求。本文只关心那些与前面定义的连通图有关的查询。而且，我们还将这些查询限定为与寻路有关的一类查询。这类查询中比较典型的一个查询被称为底层代码（underlying code）。它向寻路算法发出连续的调用，同时也会改变一个或几个参数。每次迭代都会创建一个独一无二的参数集，然后将这个参数集传递给A*算法，由它产生一个最优的结果。在搜集到所有的结果，并从中选出一个最优结果之后，底层代码就会将产生这个结果的参数集提取出来，并返回最优结果和相应的数据集。

这样的查询有一个很好的例子：一个建筑物有若干个出口，然后发出一个请求，在这些出口中进行选择。查询的结果是找到一条通往某个出口的最小开销路径。在这种情况下，底层代码会为每一个出口位置调用一次A*算法，最后返回的是产生最小开销路径的出口位置，以及最小开销路径。正如前面所讨论的，连续多次地调用A*算法很快就会造成性能的下降，我们需要更高效的解决方案来解决这个问题。

3.7.2 算法

知道了问题所在及问题域，现在要开始寻找解决方案。计划以A*算法作为出发点，衍生出一个新的算法，一个更适合前面描述的高级AI查询的算法。作为第一步，为给大家提供一个A*算法的参考，这里用伪代码的形式向大家展示A*算法：

```
List OpenList, ClosedList

AStarPathfinder(Node              StartNode,
                Node              TargetNode,
                MovementCriteria  MovementCriteria,
                Path              PathFound)
{
    Node StartNode, BestNode, SuccessorNode
    Cost NewCost

    reset OpenList and ClosedList
```

```
    if (StartNode fails MovementCriteria) return as failure
    set StartNode cost to 0
    set StartNode estimate to heuristic estimate of remaining cost
      to TargetNode
    set StartNode value to sum of cost and estimate
    set StartNode parent to NoParent
    add StartNode to OpenList

    while OpenList is not empty
    {
        remove best node BestNode from OpenList
        if BestNode is TargetNode
        {
            construct path and save it in PathFound
            return as success
        }
        for each successor SuccessorNode of BestNode
        {
            if (SuccessorNode fails MovementCriteria) continue
            set NewCost to sum of BestNode cost and cost of
              moving from BestNode to SuccessorNode
            if ((SuccessorNode is in OpenList or ClosedList) and
                (NewCost is not less than SuccessorNode cost))
                continue
            set SuccessorNode cost to NewCost
            set SuccessorNode estimate to heuristic estimate of
              remaining cost to TargetNode
            set SuccessorNode value to sum of cost and estimate
            set SuccessorNode parent to BestNode
            if SuccessorNode is in ClosedList
                remove SuccessorNode from ClosedList
            if SuccessorNode is not in OpenList
                add SuccessorNode to OpenList
        }
        add BestNode to ClosedList
    }
    return as failure
}
```

这个伪代码只使用了少数几个数据类型。List 是一个泛型数据类型，表示节点的集合。它的实现方法有很多，我们会在后面的文章中看到。Node 是一个数据结构，包含着每个独立节点的所有相关信息。节点数据结构包括的信息有：到达这个节点所需要的开销、到达目标节点所需开销的启发式估算值、上面这个开销加上启发式估算值得到的一个启发式和值，以及一个指向父节点的链接。MovementCriteria 是一个聚合（aggregate）数据结构，表示一系列的变量，定义了一个节点在连通图中的移动能力。Cost 是从起始节点到当前节点的开销。在一个典型的实现中，通常用一个浮点数来表示 cost（开销）。Path 这个数据类型中包含着路径数据。

现在可以开始着手改进这个算法的性能。要想找到解决方案，一个显而易见的方法是将所有要使用的参数实例都捆绑在一起，在一次调用中将它们传递给寻路算法。利用这个额外信息，算法应该可以更高效地执行这些组合式的请求。

通过查看 A*算法的伪代码实现会发现，如果可以减少每个路径方案求解过程中所需要的遍历次数，就能够节省大量的开销。遍历循环中使用的参数也是遍历逻辑的一部分。这个遍历逻辑决定了遍历工作将如何展开。在循环执行的过程中，是无法改变这些参数的，除非中断这个算法。为这些参数传递多个值不会带来什么好处，因为同一时刻只能使用一个值。实际上，在所有传递给这个算法的参数中，只有 start node（起始节点）没有被遍历逻辑使用，这使得它成为该优化工作惟一的一名候选者。

3.7.3 算法的改进

那么该如何利用上面的这些发现修改 A*算法，让它可以接受多个起始节点，并从中进行选择，作为单一搜索空间遍历的一部分呢？事实证明，这个问题的解决方案是相当简单的。从最初那个 A*算法的伪代码中可以看到，算法的逻辑中没有任何设置可以防止我们使用多个起始节点。除了可以传递单一的起始节点，我们也可以将所有的起始节点一次性地传递进来，然后再对它们逐个进行处理。这个处理过程和单一起始节点的处理过程是一样的。这就意味着，在初始化步骤中，每个起始节点都变成了一个开放节点，被添加到开放列表中（open list）。开放列表提供了当前开放节点的排序，可以帮助我们选择一个起始节点。它会返回一个最佳起始节点，这个起始节点也是最终路径的一部分。

那么很自然地，我们现在也想知道，在结果路径的另一端，是否也存在这种类似的情况呢？是否有办法可以在多个结束位置中进行选择呢？如果仔细地看一下最初那个算法的伪代码，就会发现目标节点其实有着双重的目的。其一就是为开销估算函数执行的计算工作提供信息，指引搜索空间的遍历方向，最终到达目标节点。另一个目的是作为遍历工作的停止点。那么，是否可以对目标节点的功能进行分割，使它不再作为遍历工作的停止点呢？

事实证明这也是有可能的。为了提高算法效率，可以有多个停止节点。这些停止节点设置在通往目标节点的路上。停止节点与目标节点非常类似，但它们只能在被移出开放列表之后才能使用，而在此之前是不能使用的。考虑了上述所有这些改变之后，现在就可以提出一个衍生的寻路算法，依旧采用伪代码的形式展示给大家：

```
List OpenList, ClosedList

BeyondAStarPathfinder(List               StartList,
                      List               StopList,
                      Node               TargetNode,
                      MovementCriteria MovementCriteria,
                      Path               PathFound)
{
    Node StartNode, BestNode, SuccessorNode
    Cost NewCost

    reset OpenList and ClosedList
```

```
    for each StartNode in StartList
    {
        if (StartNode fails MovementCriteria) continue
        set StartNode cost to 0
        set StartNode estimate to heuristic estimate of remaining
          cost to TargetNode
        set StartNode value to sum of cost and estimate
        set StartNode parent to NoParent
        add StartNode to OpenList
    }
    while OpenList is not empty
    {
        remove best node BestNode from OpenList
        if BestNode is in StopList
        {
            construct path and save it in PathFound
            return as success
        }
        for each successor SuccessorNode of BestNode
        {
            if (SuccessorNode fails MovementCriteria) continue
            set NewCost to sum of BestNode cost and cost of
              moving from BestNode to SuccessorNode
            if ((SuccessorNode is in OpenList or ClosedList) and
                (NewCost is not less than SuccessorNode cost))
                continue
            set SuccessorNode cost to NewCost
            set SuccessorNode estimate to heuristic estimate of
              remaining cost to TargetNode
            set SuccessorNode value to sum of cost and estimate
            set SuccessorNode parent to BestNode
            if SuccessorNode is in ClosedList
                remove SuccessorNode from ClosedList
            if SuccessorNode is not in OpenList
                add SuccessorNode to OpenList
        }
        add BestNode to ClosedList
    }
    return as failure
}
```

与 A*算法相比，这个衍生出来的算法有几个主要的区别：它的参数列表可以接受多个起始节点和多个停止节点；多个起始节点都变成了开放节点，而任何一个停止节点都可以停止算法的遍历。

如何才能知道这个衍生出来的算法保证可以生成最小开销路径呢？虽然这个问题的证明已经超出了本文的范畴，但在这里还是会给出关于这个主题的几个方向性的指导。首先，对于那些确保能够找到最佳路径的算法，可以把它们定义为“可接纳”的算法。其次，需要考查这个 A*算法的可接纳性[Nilsson98]，并以此作为出发点。论证工作的关键是引入一个额

外的节点，该节点与所有起始节点相连。我们把这个新节点看做是一个单一的起始节点。该节点从根本上将这个衍生算法变成一个标准的 A*算法。

3.7.4 实现的细节

该衍生算法的一个典型的实现与典型的 A*算法实现并无太大的区别。由于 A*算法实现的细微差别是众所周知的[Pinter01]，所以我们的重点就放在衍生算法中那些专有特性的实现上。

这个衍生算法可以传递多个起始节点和停止节点。正如大家所想到的，将多个起始节点和停止节点作为参数来传递，会增加有关的数据量，同时也会增加额外的性能开销。为了减少这个开销，可以将多个节点数据存储在数组中，然后通过引用来传递这些数组。很多实际的算法实现都对付过类似的问题。还有一些做法是将节点数据独立地存储起来，然后使用数组来保存节点数据的引用、句柄或对象 ID，以此来访问实际的节点数据。

这个衍生算法还使用了多个停止节点，以便终止算法的执行。给这些停止节点打上标签（tag）是比较可取的方法。这些标签应该保存在某种节点数据结构中。同时，该节点其他的运行时数据（开销值和一个指向父节点的指针）也保存在这个节点数据结构中。给这些节点打上标签，就可以直接检查节点标签，而不用再将每个遍历到的节点与所有的停止节点进行比较。假设需要遍历 m 个节点，寻找 n 个停止节点中的一个。对于没有使用停止节点标签的实现，需要进行 $m \times n$ 次比较。而使用节点标签的算法可以将比较次数降低到 m，每个遍历到的节点只需要比较 1 次。

3.7.5 应用实例

在实际应用中，这个衍生算法可以提供一些先进的功能。下面来考虑一个典型的即时战略游戏（RTS）。游戏中有一些游戏对象（比如军营），其中驻扎着一些作战单位。指定一个集结地点，这些驻扎的作战单位就应该从游戏对象中走出来，移动到这个集结地点。为了叙述的方便，我们把这些游戏对象称为“容器类游戏对象”（*containing game object*）。在实际的游戏产品中，容器类游戏对象通常都被实现成建筑物，或者是各种运输工具。

1. 使用多个起始点的例子

与这个操作相关的最常见的问题，是如何在容器类游戏对象周围的出口位置中选择最合适的出口，来放置那些要出营的作战单位。最简单也是最快速的方法，是预先定义好出口地点顺序，然后从中选择第一个畅通的出口地点。由于这个方法并没有考虑集结地点的位置，于是在向集结地点进发之前，那些要出营的作战单位通常会被迫在容器类对象里绕来绕去。

为了避免这个问题，应该选择合适的出口，出口的位置与集结地点有着最短的直线距离。虽然这明显是一个更好的方法，但也有它自身的问题。其中之一就是，它没有考虑到在集结地点和出口地点之间有障碍物的情况。因此，这个方法不能保证所选择的出口就是最优的出口地点，也不能保证从这里到集结地点有一条最小开销路径。还有更糟糕的情况，即障碍物将选择的出口地点与集结地点完全隔断，困住了那些要出营的作战单位，也彻底削弱了这些

作战单位对玩家的效用。

要想解决这个障碍物的问题，有一个方法就是计算集结地点到每一个出口地点的路径，并从中选择开销最小的路径。但不幸的是，这个算法的性能开销实在太高，除非只有少数几个出口地点。

要想避免这个问题，不需要连续地调用 A*算法，只需要调用一次衍生算法。可能的出口地点会提供多个起始节点，而集结地点则可以作为目标节点，同时也可以作为一个停止节点。通过使用这些参数，衍生算法会找到一个最优的出口地点，而且只需要一次搜索空间的遍历，就可以得到通向集结地点的最小开销路径。

图 3.7.1 就是一个例子。其中的“建筑物”可以看做是一个容器类对象，它的四个方向上有着十几个出口。在这个建筑物与集结地点之间有一个类似墙体的结构，这就是一个障碍物。而最优的出口地点就是一个起始点，从这里到集结地点存在一个最小开销路径。在这幅图中，建筑物右下角有一个出口地点，它与集结地点（目标节点）有着最短的直线距离，但它却不是一个最优的出口地点。

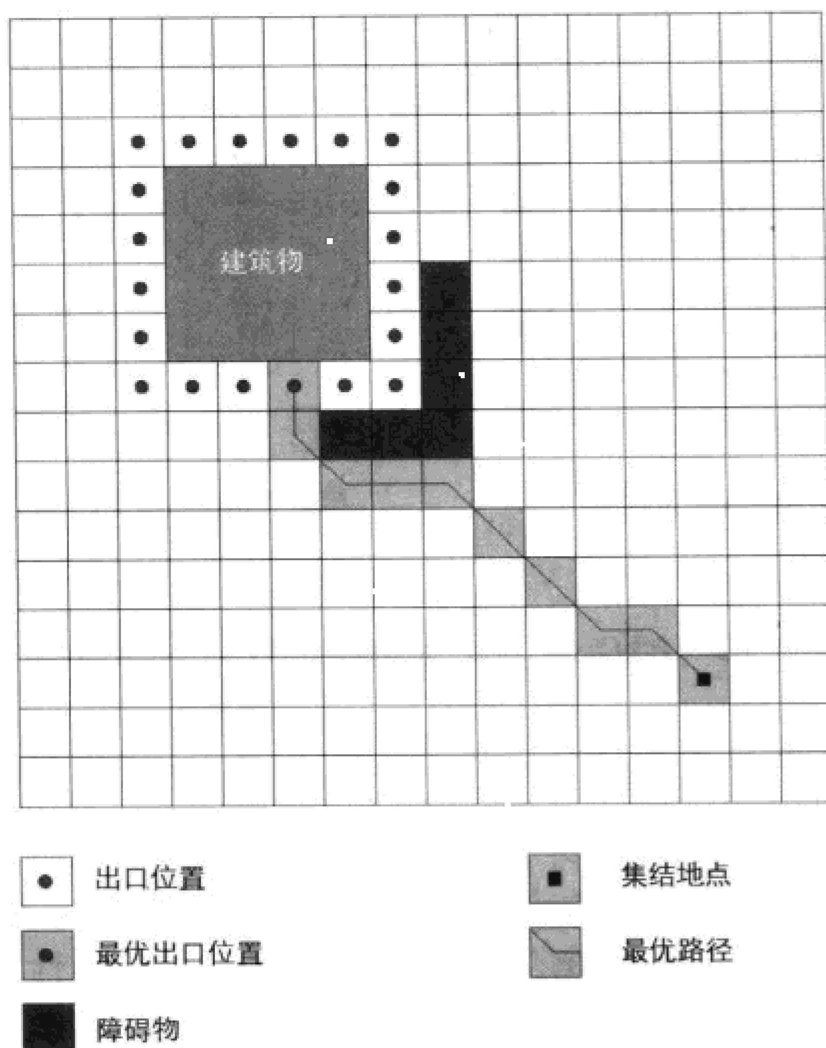


图 3.7.1　一个实用案例：使用多个起始节点，找到一个最优出口位置

还有很多的实际应用都使用了多个起始节点。再举个例子，假如有若干个建筑物，然后给定一个目标位置，比如一个战况激烈的区域。那么，该如何从这些建筑物中选择合适的一个，向目标区域派遣需要的作战单位。对于最后被选定的建筑物，从它那里派遣的作战单位可以最快速地达到战场。我们只需要调用一次衍生算法，就可以很快地得到有关的答案。全部建筑物的所有出口位置都可以作为起始节点，目标区域可以作为目标节点。有了这些参数，衍生算法绝对可以找到一个通向目标节点的最小开销路径，同时也可以提供一个最优出口位

置，以及和这个出口相关的建筑物。

在图 3.7.2 中，有候选的三个建筑物 A、B、C，都可以向目标地点派遣作战单位。建筑物 B 正好有一个最优出口，也就是最小开销路径的起点。其中有一点大家要特别注意，建筑物 A 有几个出口位置距离目标位置更近。但是因为建筑物的右侧有一个墙体样的障碍物，所以这些出口都不是最优的。

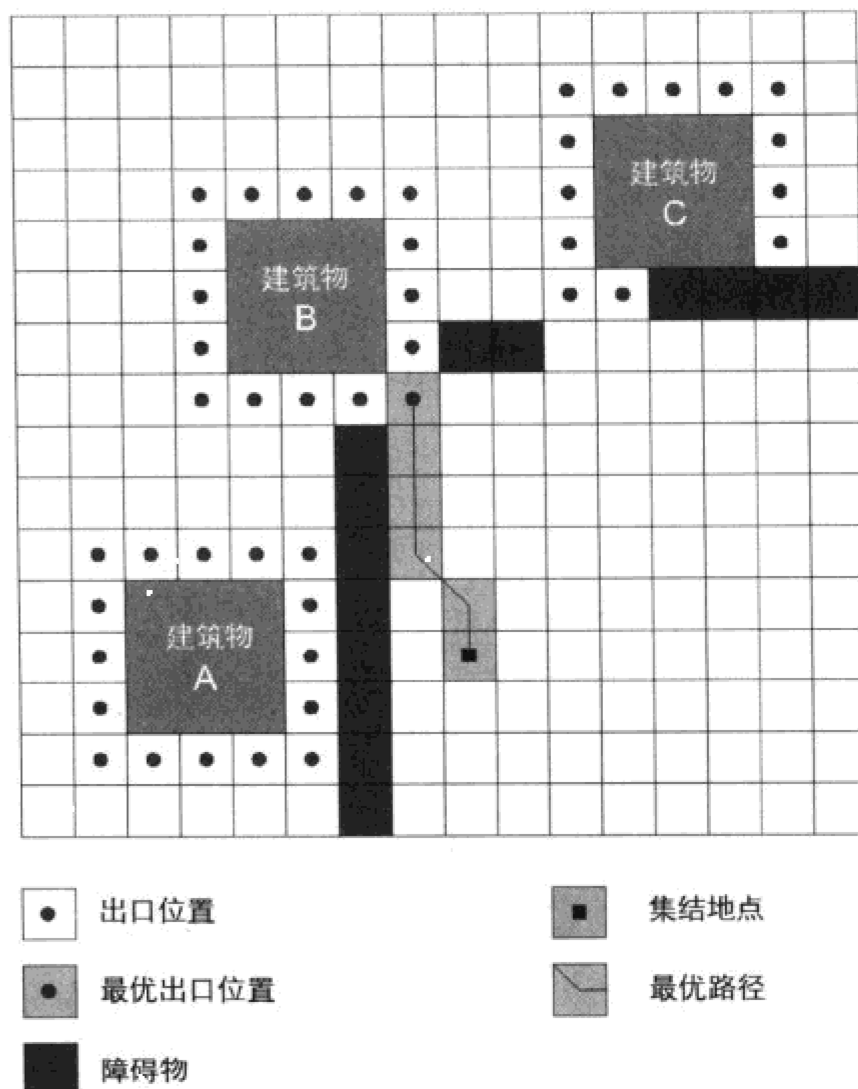


图 3.7.2　使用多个起始节点，找到一个最优建筑物，以便派遣作战单位

2. 使用多个停止节点的例子

现在换个角度，假设命令一个作战单位去驻防指定的容器类游戏对象。我们的任务是在容器类游戏对象四周的那些入口位置中，选择一个最优入口，让作战单位进入这个游戏对象。选择离作战单位的直线距离最小的入口，是一种可能性。但是这个方法也同样面临着障碍物问题的烦恼。这和上面那些多个入口的例子是同一个问题。另外一种可能的方法，是在寻路遍历过程中把容器类游戏对象看成是一个障碍物。当作战单位“碰撞”到这个对象时，就让它进入这个对象。此方法对前面的那些例子不太适用。但不幸的是，这个方法也只适用于那些凸面形状的容器类游戏对象。在其他情况下，可能无法找到一个最优方案。

更好、更通用的解决方案，是考察作战单位与每个入口位置之间的路径。就像前面处理那些出营的作战单位一样，我们也不需要连续地调用 A*算法，而只需要调用一次衍生算法。可以把作战单位的位置看成是起始节点，而所有可能的入口位置都可以作为停止节点。为了确保算法最后收敛在容器类游戏对象上，我们需要在容器类游戏对象内部确定一个目标节点，

最好是在接近游戏对象中央的位置上。将这些参数传递给衍生算法，就可以得到一条通向最优入口位置的最小开销路径。

在图 3.7.3 中，容器类游戏对象就是那个建筑物，它大概有十几个入口位置。图中还标明了被派去驻防的作战单位的初始位置，以及从这个位置到最优入口位置的一条最小开销路径。与图 3.7.1 中的例子类似，在这个图里，注意位于建筑物左上角的那个入口位置。虽然它到作战单位的直线距离最短，但是因为中间隔了一个障碍物，所以它也不是最优入口。

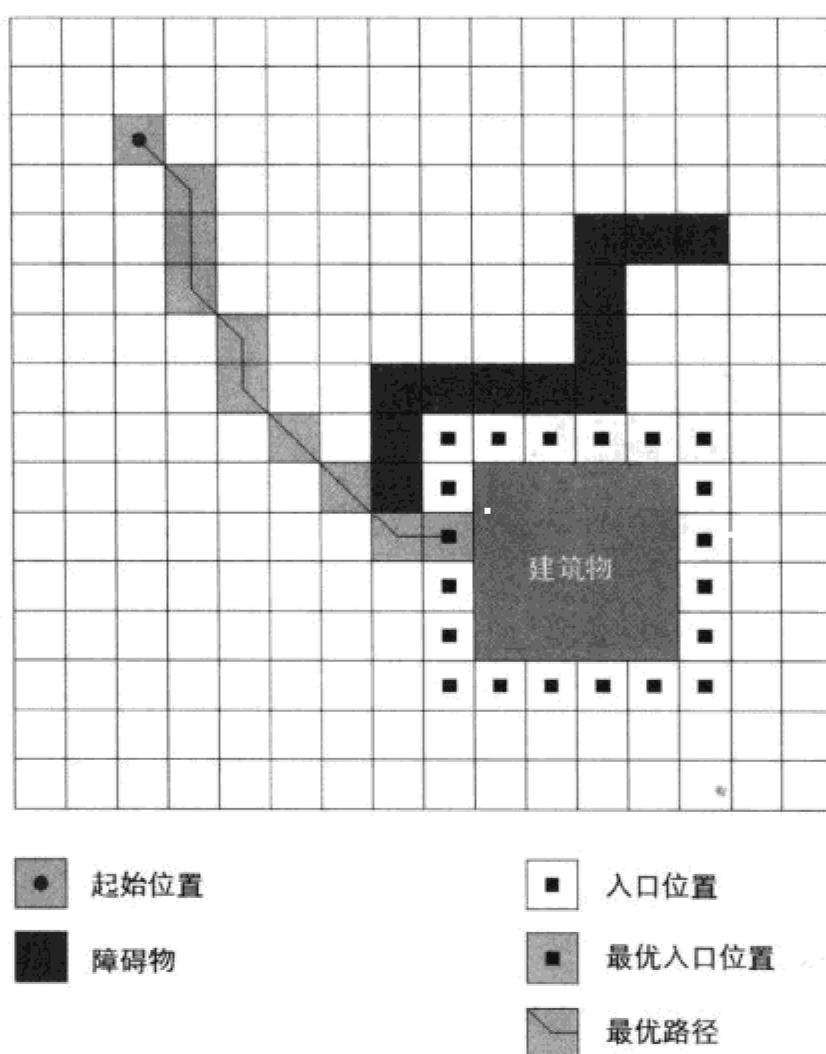


图 3.7.3　使用多个停止节点，找到一个最优入口位置

3. 同时使用多个起始节点和多个停止节点的例子

前面几个例子，要么是使用了多个起始节点，要么是使用了多个停止节点。现在，来看看多个起始节点和多个停止节点是如何一起工作的。考虑这样一种情况：一个作战单位正在一个容器类游戏对象中驻守。现在命令它走出游戏对象，马上进入另外一个容器类游戏对象。我们的任务是在两个容器类对象之间找到一条最小开销路径，同时还要考虑潜在的障碍物。

根据从前面几个例子学到的经验，很明显，第一个游戏对象的出口位置提供了多个起始节点；而第二个游戏对象四周的入口位置提供了多个停止节点。另外，还需要在第二个游戏对象内部接近中心的位置上安排一个目标节点。利用这些参数，衍生算法就可以得出作战单位应该从第一个对象的哪个出口出来，应该从第二个对象的哪个入口进入到第二个对象中，以及这个过程中最优的移动路径。该算法可以保证找到的路径是一个最小开销路径。

这并不是什么特殊的情形，在很多游戏中都可以找到类似的情况。再举个例子，有一个负责生成建筑物的游戏单位，它可以自动地调动一些作战单位去驻守某些防御工事，比如碉堡或塔楼。为了设置实现这个功能，玩家可以在这个生成建筑的游戏单位和防御工事之间设置一个集结点。

图 3.7.4 展示了一个有两个建筑物的例子——建筑物 A 和建筑物 B，以及它们之间的一条最小开销路径。和前面的例子一样，它们之间的直线路径上也有一个类似墙体的障碍物，表明最短直线距离的方法并不好用。

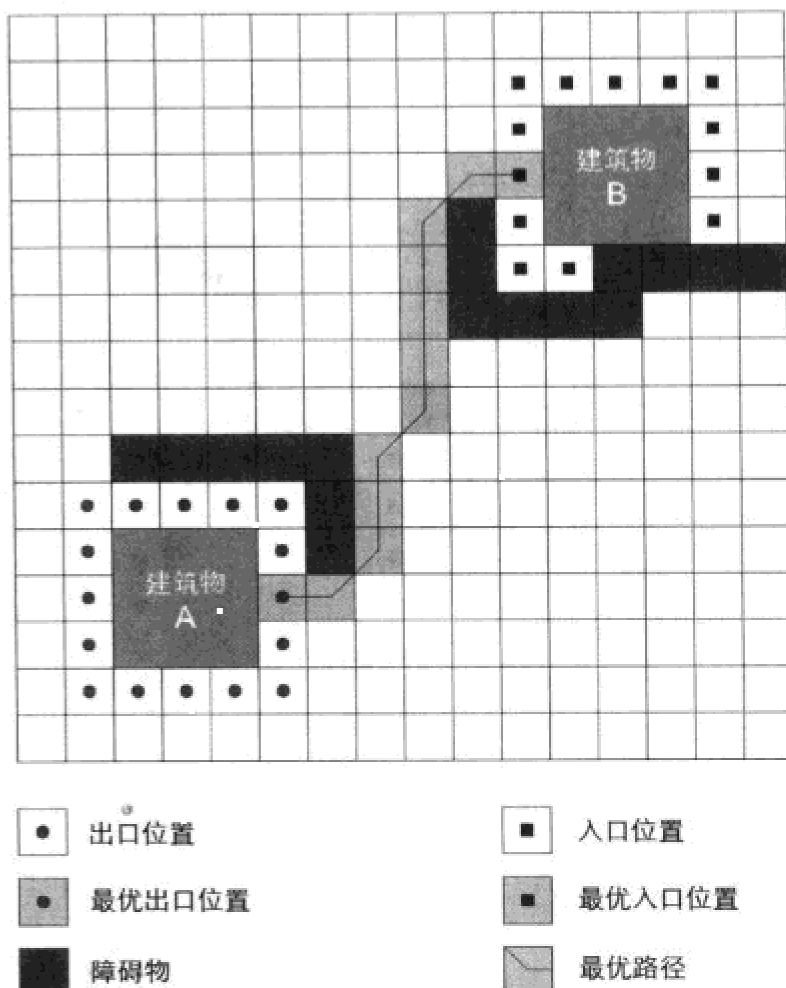


图 3.7.4 使用多个起始节点和多个停止节点的例子，目标是找到两个建筑物之间的一条最小开销路径

对于一个由路点图表示的搜索空间[Tozour03]，目前介绍的这些技巧同样非常有用。在这样的一个系统中，位于二维或三维连续空间中的点，被称为路点（waypoint）。路点定义了图形中的节点。由于游戏世界的本质使然，以及那些用来表示这个世界的图形数据，路点的布置通常都相当稀少。

在这样的搜索空间中进行寻路，主要的问题是寻路（pathing）游戏对象当前的位置。由于这个位置可能会在图中很远的地方，因此如何选择合适的起始节点和停止节点，就成了一个有挑战性的工作。假设有某个节点，它到寻路对象的直线距离最短。如果把它作为起始节点，就会产生不正常的结果。对象首先要移动到起始节点，这就表示实际的路径并不是最优路径。这使得对象在开始向目标位置移动之前，其表现好像是一个要离开目标位置的对象。图 3.7.5（a）就是这样的一个例子，可以看看这种奇怪的移动是怎么发生的。

解决这个问题的一个方案，是在距离寻路对象最近的那条连通图的边上，搜索距离寻路对象最近的一个点。但是，这个搜索工作带来的性能开销实在是太大了。而且，在结果路径两端上都存在开销问题。使用衍生算法，给定多个起始节点和多个停止节点，并同时提交若

干个不错的候选节点作为起始节点或停止节点，就可以很漂亮地解决这个问题。算法可以接受这些候选节点，然后在寻路的过程中，它会自动判断出哪个节点应该是实际的起始节点，哪个应该是停止节点。图 3.7.5（b）中的情况和图 3.7.5（a）一样，但是这次使用了衍生算法，于是就看不到之前那种笨拙的移动了。

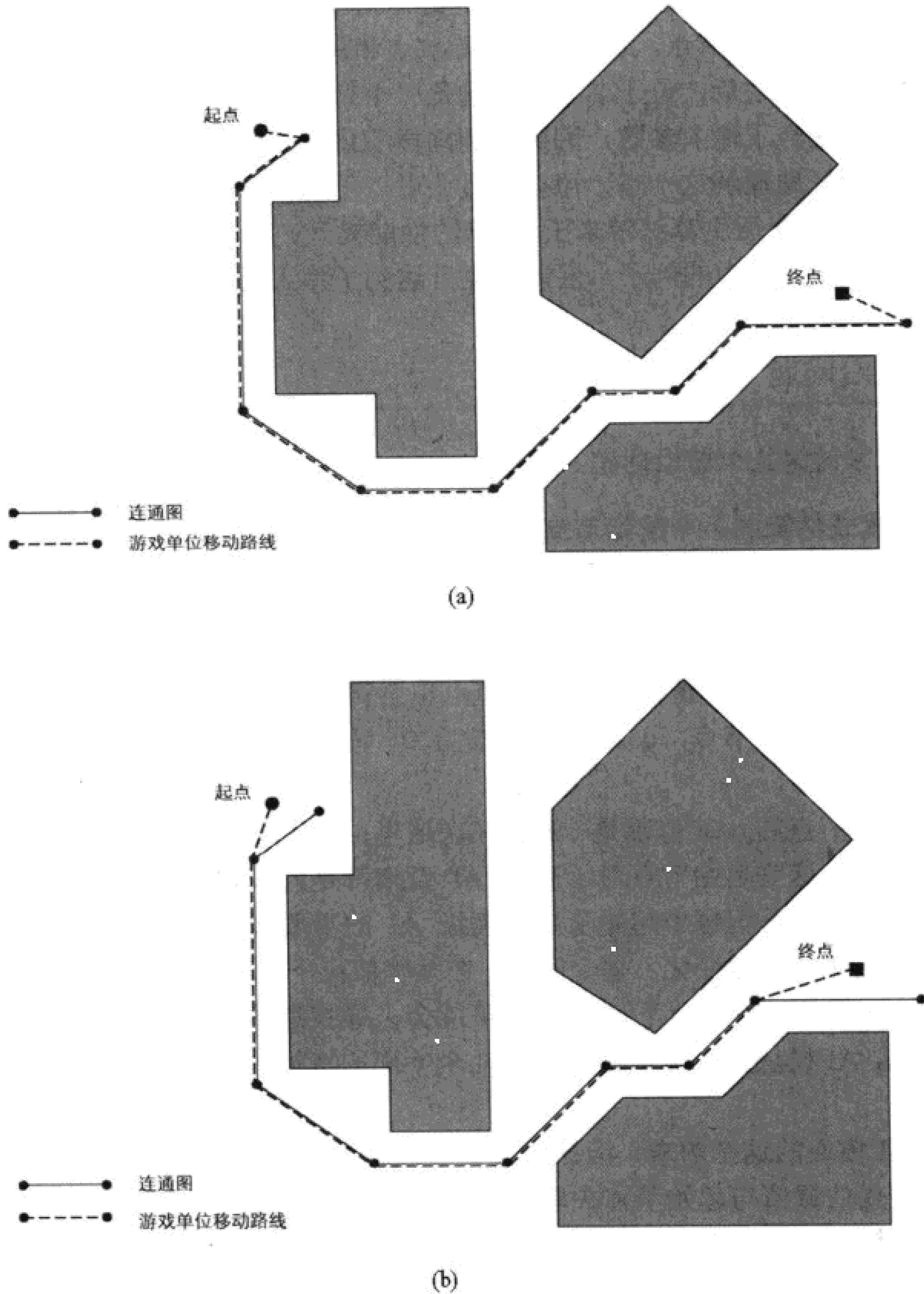


图 3.7.5 （a）游戏单位使用路点图的例子 （b）使用衍生算法改进游戏单位选择起始节点和停止节点的例子

3.7.6 性能方面的考虑

为了进行公平、全面的对比，我们用衍生算法一次迭代的性能与 A*算法多次迭代的性能进行比较。这个功能是颇具代表性的，由高级 AI 查询组成。为了查询的方便，这个衍生的

A*遍历取代了对原先 A*算法的多次调用，而且在功能上也相当于原先 A*算法的多次迭代。但是与原先的 A*算法不同的是，由于同时使用了多个输入参数，而且起始节点和停止节点之间的节点都只需要遍历一次，因此整体的性能提升可以达到一个新的数量级，甚至更高。

假设起始节点的数量是 l，停止节点的数量是 m，二者之间需要被遍历的节点数量是 n，现在就可以看看这个衍生算法所带来的不同之处。对于使用多次连续迭代的 A*算法，每个 AI 查询需要检查 $l\times m\times n$ 个节点，才能找到起始节点和停止节点之间的最佳路径。每次更换新的起始节点和停止节点之后，位于二者之间的每一个节点都需要被检查 $l\times m$ 次。而如果使用衍生算法，由于它聚合了输入参数，因此找到同样的最佳路径只需要检查 n 个独立的节点。

为了让大家更好地理解这一点，可以回过头看一下前面那些例子的性能提升情况。对于图 3.7.1 中的例子，这个衍生算法带来了 20 倍的性能提升；对于图 3.7.2 中的例子，性能提升了 45 倍；而对于图 3.7.4 中的例子，性能的提升达到了惊人的 208 倍。

3.7.7 几个前沿问题

下面将和大家讨论几个前沿问题。

1. 带有启发式估算的多个起始节点

从前面这个算法的伪代码实现中可以看到，作为初始化步骤的一部分，所有的起始节点都变成了开放节点，并添加到开放列表中。由于开放列表中所有的节点都必须有一个相关的开销值（cost）和启发式估算值，所以起始节点也会得到这些为它们指定的值。对于所有新添加到开放列表中的开放节点，cost 的值被设为 0，而启发式估算值被设置为到达目标的估算开销。

既然所有起始节点的 cost 值都是一样的（在这里，所有 cost 的值都是 0），那也就是做出了这样的假设：所有这些起始节点对于高级 AI 查询都是同等重要的。但在实际的游戏中，这个假设基本不成立。不但每个起始节点对高级 AI 的重要程度不同，而且它们的重要程度会随着不同的情况而不断地变化。举个例子，在某些情况下，AI 模块会去寻找那些不太容易被攻击的节点；在另外一些情况下，AI 模块可能会去寻找那些离资源比较近的节点。而对于更为复杂的情况，AI 模块也可能会综合考虑几个不同的特征，寻找那些符合这个组合特征的节点。

为了迎合 AI 模块的这个需求，应该让起始节点可以有不同的初始开销（cost）值。这个特性可以将启发式估算值与起始节点关联起来，并根据重要程度对它们进行排序、评级。对所有的启发式估算值也是如此，这些值需要小心地选择，并考虑算法在执行过程中所使用的其他开销值。否则，初始的开销（cost）值会让算法的启发式变得无以为用。为了实现这个特性，可以将初始开销（cost）值与那些传递给算法的起始节点关联起来，并在初始化的时候使用这些值。

2. 返回多个解决方案

截止到目前，我们一直都在假设使用该算法的 AI 查询只需要得到一个停止节点的一个解决方案。这也就是为什么在达到第一停止节点之后，算法就终止了。而在实际应用中，有

些 AI 查询可能需要接收所有停止节点的结果，然后再将这些结果与其他数据组合起来，最后才确定要选择哪个停止节点。

为了迎合这种新的需求，我们对当前的算法进行了修改。这样，当它达到第一个停止节点时，算法就会将得到的路径保存到第一个停止节点，然后再继续查找其他的停止节点。当它达到下一个停止节点时，也同样会把得到的路径保存在新到达的停止节点中。算法就一直这样继续下去，直到所有的停止节点都得到了自己的路径，或者所有的开放节点都用完了。用完了所有的开放节点，也就表示有一个或几个停止节点无法达到。这个修改过的算法返回的是一个路径的数组，每个可达到的节点都有一个对应的结果路径。

新的算法有一个潜在的缺陷，需要在这里指出来。这个算法会引发很严重的性能下降。与原始的 A*算法非常类似，这个衍生算法在修改之后也有一个最坏的情况。由于算法必须不断地遍历搜索空间，直到找到一个合理的路径，因此如果找不到停止节点，这样的设计会使算法遍历整个搜索空间。修改后的算法要试图到达多个停止节点，这使得最坏情况发生的可能性大大增加。这个危险是真实存在的，所以在决定是否使用可以返回多个解决方案的版本时，一定要考虑到这个危险因素。

3.7.8 总结

A*算法是 AI 模块实现中很重要的一个组成部分，它是基本寻路需求的一个可靠的解决方案。但不幸的是，来自高级 AI 模块的查询会产生数量庞大的 A*算法调用，为程序性能带来很大的负面影响。解决该问题的一个方法是设计一个寻路算法，只通过一次性能高效（performance-efficient）的调用，就可以完成多个 A*算法调用的工作。本文向大家介绍了一个这样的算法。

这里介绍的算法是 A*算法的一个衍生算法，可以接受多个起始节点和停止节点。由于充分利用了这些附加信息带来的好处，因此只需要一次搜索空间遍历，这个算法就可以为 AI 查询找到答案。这种高效率反映在性能的改进上，即将性能提高了两个数量级之多。该算法有很多实际的应用，文章中也讨论了其中的几个案例。这些素材为大家提供了一个很好的参考，从这里开始，就可以编写更深层次的 A*衍生算法，并将它应用在高级 AI 模块中了。

3.7.9 参考文献

[Botea04] Botea, A., M. Müller, and J. Schaeffer. “Near Optimal Hierarchical Path-Finding.” In the *Journal of Game Development*. March, 2004.

[Cormen01] Cormen, Thomas H., et al. *Introduction to Algorithms, Second Edition, 370–404*. MIT Press, 2001.

[Nilsson98] Nilsson, Nils J. *Artificial Intelligence: A New Synthesis, 145–150*. Morgan Kaufmann Publishers, Inc., 1998.

[Pinter01] Pinter, Marco. “Toward More Realistic Pathfinding.” In *Gamasutra*. Available online at *www.gamasutra.com/features/20010314/pinter_01.htm*. March 14, 2001.

[Stout00] Stout, Bryan. “The Basics of A* for Path Planning.” In *Game Programming Gems*,

254–263. Charles River Media, 2000.

[Stout96] Stout, Bryan. "Smart Moves: Intelligent Pathfinding." In *Game Developer Magazine*. October 1996. Available online at *www.gamasutra.com/features/19970801/pathfinding.htm*.

[Tozour03] Tozour, Paul. "Search Space Representation." In *AI Game Programming Wisdom 2*, 85–102. Charles River Media, 2003.

3.8　实现最小重新规划开销的先进寻路算法：动态 A*（D*）算法

Marco Tombesi
baggior@libero.it

寻路问题是游戏开发领域最著名的理论问题。业界有大量的文献资料涵盖了这个问题的方方面面。这是因为，寻路问题并不仅仅表现在计算机科学领域，在机器人学、数据挖掘和自动化技术等领域，寻路问题同样也广泛存在。

在游戏中，地图接近于现实。在大多数情况下，地图是指用来表示游戏环境的图形。我们一般使用规则的网格图，以特定的比例或分辨率来表示地图，其中的每个节点对应地图中的一个点（参见图 3.8.1）。

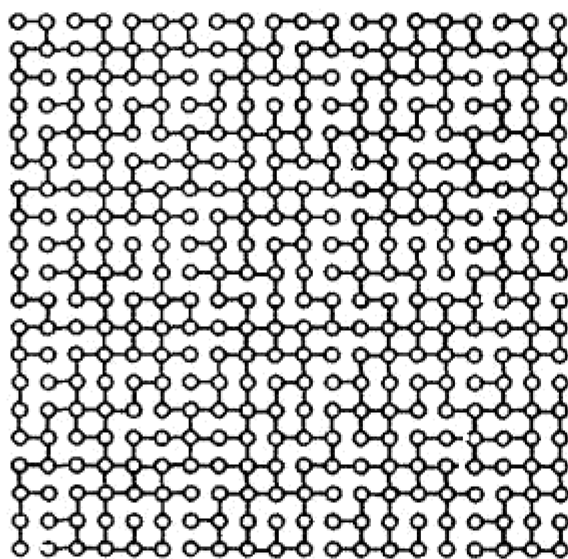


图 3.8.1　用一个规则的网格图来表示地图，分辨率为 1 英尺

在这个地图中，如果给定一个起始点和一个目标点，如何才能找到二者之间的最短路径呢？解决这个问题的最常用算法是 A*算法。这个算法的性能远好于 Dijkstra 算法，或其他的单源最短路径（Single-Source Shortest Path，简称 SSSP）算法[Stout96]。虽然 A*算法在静态环境中效果不错，但是对于动态地图，它就显得效率很低：任何一个地图的改动都必须重新进行规划。

对于一条已经计算好的路径，对地图的改动可能只会影响其中很小的一部分，且这种可能性非常之高，所以重新计算整条路径绝对是比较浪费的。A*算法的动态版本（也就是众所周知的 D*算法）可以帮助我们解决这个问题。

3.8.1 D*算法

正如前面所说的，我们需要用一个图形来表示真正的地图。如果地图中有两个点是相连的，那就在图形中用一条边来表示这两个节点。一般来讲（但也不是必须的），每一条边都有一个关联的开销，表示通过这个节点所要付出的代价。

在图 3.8.2 中，可以这样假设，如果对于一条特定的边，其相关的开销＞n（地图中节点的数量），那么在表示地图的图形中，那个圆弧边就不用再画出来了。

本文假设如果从 A 点能够达到 B 点，就把这种情况表示为：Cost [Edge(A,B)]≤n，其中 n 是地图中节点的数量。如果两个点不是相连通的，就把这种情况表示为 Cost [Edge(A,B)]＞n。

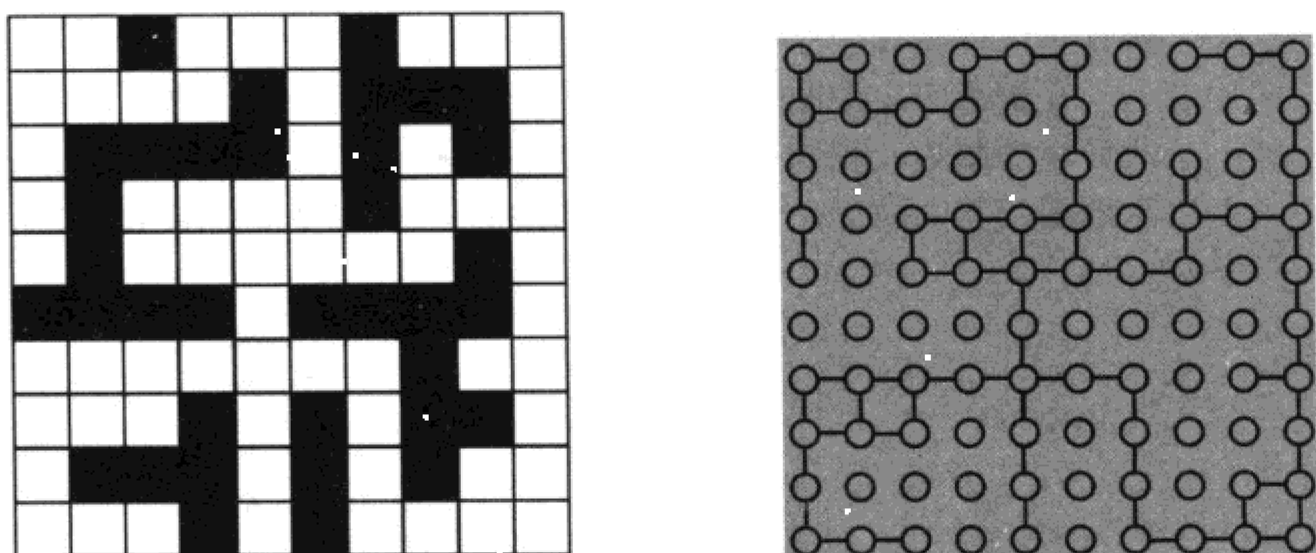


图 3.8.2 一个带有障碍物的地图及其表示图

D*算法将一个地图的表示图作为输入。在遍历过程中，这个算法会首先检查地图中是否在某些点上发生了变化。如果在表示图中确实发现了变化，D*算法就会从这个发生变化的节点开始，只去修改已计算路径中那些受影响的区域。

从本质上讲，这个算法的中心思想就是只在一个非常小的区域中进行重新规划。实时测试[Stentz94]清楚地表明，D*算法的优越性远远高于 A*算法，性能的提升可以达到地图中节点数量的指数级增长。

3.8.2 D*算法的实现细节

假设环境是一个边为 M 的正方形，表示其周遭环境的图形就作为 D*算法的输入。（这里不失普遍性地假设节点之间只有水平和垂直方向的移动，没有对角线方向的移动。）

D*算法维护着一个开放状态列表，用于处理节点的状态，并将计算工作扩展到当前检测节点所影响的邻居节点之上。在开始的时候，所有的节点都被标识为 NEW（新节点）。当这些节点被添加到开放列表中时，它们就变成了 OPEN（开放）状态。对它们的计算完成之后，这些节点又被标识为 CLOSED（关闭）状态。

D*算法为每一个节点的这些标签维护着一个显式的标签列表，我们把这个列表看成是 *Tag(x)*。

*Backpointer(x)*表示的是到达目标需要行进的方向。从每个节点 x 开始，如果沿着 *Backpointer(x)*行进，就可以达到目标，而且走过的路径就是最短路径（实际上，对于相同的开销，会存在多条不同的最短路径）。路径的开销就是路径上所有遍历到的边的总和。

我们将 $H(x)$定义为从节点 x 到目标的估算距离。在重新规划之后，$H(x)$就是节点 x 到目标的最短距离。

关键函数 $K(x)$被定义为下面两种情况中的最小者：

- 在发生变化之前，$K(x)$就是 $H(x)$
- 将节点 x 放入开放列表后，$K(x)$是所有 $H(x)$的最小者

在将节点分成两大类的过程中，这是一个很重要的线程。根据 $K(x)$的值和 $H(x)$的值，可以考虑下列两大类的节点：

Raise $K(x)<H(x)$：当图中的 cost（开销）增加时，才会用到这类节点。必须向所有受影响的节点传达这个信息。

Lower $K(x)= H(x)$：当图中的 cost（开销）减少时，才会用到这类节点。必须向所有受影响的节点传达这个信息。

我们马上就会看到，D*算法对待这两类节点的方式不尽相同。

3.8.3　实例

讲解了 D*算法实际应用方面的关键问题之后，下面最好看一个真正的例子。随书光盘中提供了一个简单的算法实现。大家可以仔细看一下，以便详细地了解这个算法的工作流程。此外，随书光盘中还提供了一个 win32 演示程序。

为了简单起见，假设有一个 5×5 大小的地图，其中还有几个障碍物（参见图 3.8.3）。每个白色的小方块表示可以通过的节点（FREE），而黑色的小方块则表示障碍物（OBSTACLE）。每个小方块可能会变成这两种节点中的任意一种。这种非零即一的选择方式可以更好地向大家阐明算法的各个步骤。我们从左上角开始，目标则是右下角。

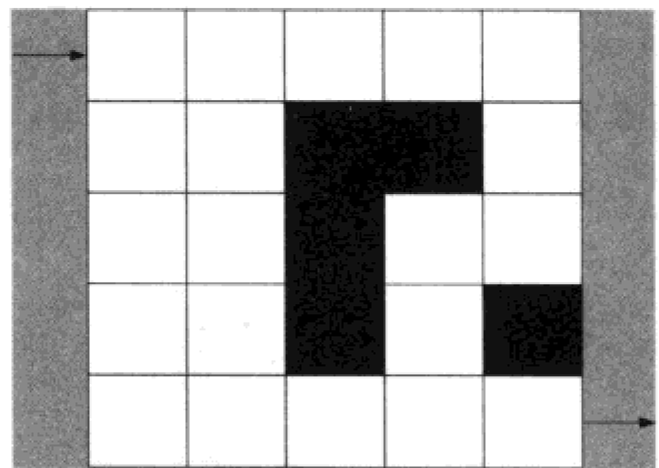


图 3.8.3　一个 5×5 大小、带有障碍物的地图

算法不断重复地调用函数 `ProcessState()`，执行一个初始化寻路。在图 3.8.4 中，箭头显示出了每个节点的 backpointer（回溯指针）。大家应该记得，backpointer（回溯指针）会告诉我们应该走哪个方向，以便最小化路径的整体开销。

图 3.8.5 中显示的是，如果从左上角（如果使用 C 语言的符号，把地图当成一个矩阵来读取的话，这个位置就是[0][0]）出发应该走的方向。

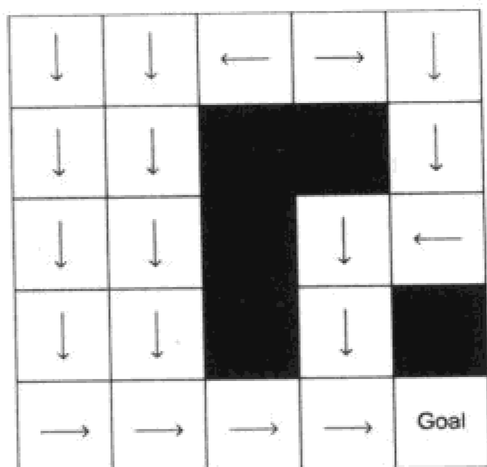


图 3.8.4 初始状态下 backpointer（回溯指针）的配置情况

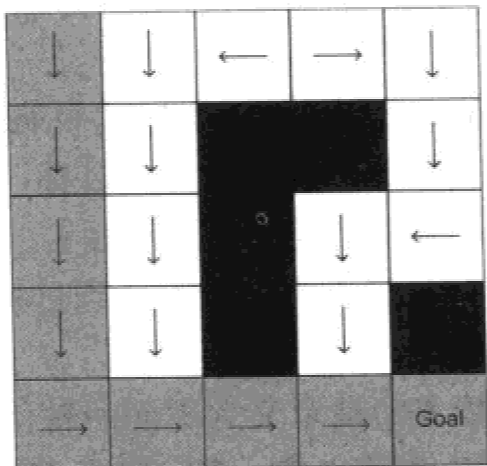


图 3.8.5 从起点到目标点的路径

现在，AI 主体开始沿着上述的路径前进。假设它走到位置[4][1]上的时候，[4][2]突然变成了一个障碍物。那么 D*算法是怎么处理这个突发情况的呢？

当算法监测到环境中的某个变化时，它就会调用函数 `ModifyCost(x,y,value)`，将圆弧的开销从 *x* 变为 *y*；而且，如果节点 *x* 是一个关闭节点（close node），就还要把节点 *x* 添加到开放列表中。接下来，当开放列表中有某些节点到目标的距离比当前节点到目标的距离还要短时，算法就会调用函数 `ProcessState()`。用这种方法，D*算法就只会去修改那些受影响节点的 *backpointer*（回溯指针），将已经计算出来的路径结果在任何可能的地方保存起来。

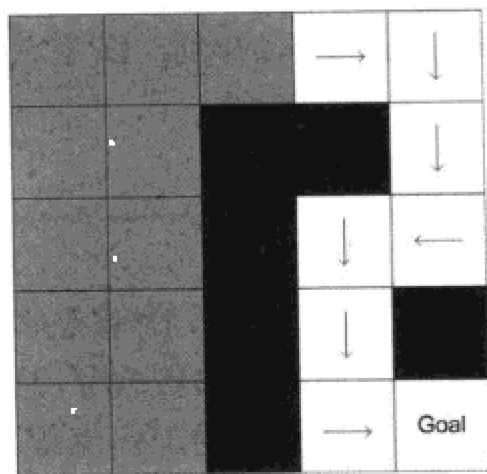


图 3.8.6 受变化影响的节点被标识为深灰色

在图 3.8.6 中，如果算法监测到了环境中的某个变化，对于需要重新计算的节点，就会将它们标识为深灰色。看了图 3.8.6 之后，就可以确信 D*算法只分析最小数量的所需节点，来正确地计算出新的最短路径。这已经很明晰地展示了 D*算法的力量。

在图 3.8.7 中可以看到，变化发生之后 *backpointer*（回溯指针）新的配置情况。从位置[4][1]（用深灰色标识）开始，图 3.8.8 中展示了一条新计算出来的路径。

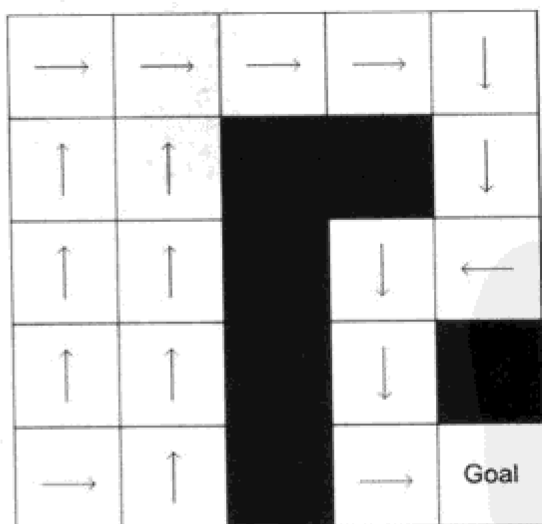


图 3.8.7 位置[4][2]上发生变化以后，新的 backpointer（回溯指针）配置情况

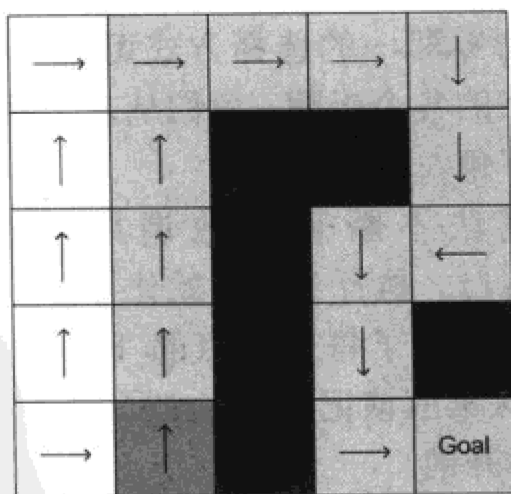


图 3.8.8 从位置[4][1]开始，计算得到一条新的路径

3.8.4 在游戏中又如何呢?

实际上，无论以什么样的形式，现在还没有太多游戏真正在使用 D*算法来执行任何种类的寻路算法。这主要是由于游戏仿真工作内在的本性使然。在这个领域，人们将虚幻的 AI 置于真实的 AI 之上。在一个清楚明白的（预先设计好的）环境中，找到一种伪自然的路径，让游戏单位在环境中的两个点之间移动，是非常简单的事情。而让游戏单位在移动的过程中发现最佳路径，则要复杂得多。实际使用的搜索地图，其空间大小也是有限的，而且地图本身也不是非常详尽，更不用说 A*算法的效率了。很明显，在现在和过去的游戏产品中，由于分配给 AI 系统的 CPU 资源非常有限，才导致了上面的 AI 局限性。

如果程序员想给游戏的主体增加一些学习能力，那么 D*算法是非常理想的选择。在这种情形下，游戏主体对它周围的环境没有任何预先的了解，所有的信息都是在它的生命周期内通过自己的感官来获得的。

D*算法控制的主体可以在更大型的地图上移动，因为在执行路径搜索操作时，D*算法只需要扫描整个游戏环境的某个局部区域；而且当环境中有混乱发生时（发现了某个新的障碍物，或者某个障碍物被移走了），重新规划的工作也只会影响一小部分的搜索区域。这样，分配给游戏主体用来进行路径规划的 CPU 资源就足够用了。

现在，很多游戏开发人员都认为，下一代的游戏会把更多的时间花费在 AI 的执行上。由于人们的关注点将转移到游戏主体行为的真实性上，所以游戏开发世界中会有越来越多类似这样的复杂算法。

3.8.5 总结

本文分析了 D*算法的基本实现。正如在[Stentz94]中所看到的，随着游戏环境的扩张，与使用 A*算法相比，使用 D*算法的好处会越来越大。这主要是因为，随着地图中节点数量的增加，从起点进行重新规划的开销也变得越来越大，而 D*算法可以尽量避免过多的重新计算工作。

也可以使用 Focused D*算法，在环境发生变化时，它可以有效地降低需要检查的节点数量。其基本思路是使用启发式来驱动研究那些有希望的节点。对这个内容感兴趣的读者可以查阅[Stentz95]。

最后要说明的是，随书光盘中的演示程序使用了 Leonardo 库[Leonardo03]，有兴趣的读者可以根据参考文献中罗列的信息查阅有关内容。

3.8.6 参考文献

[Leonardo03] Leonardo Computing Environment. Available online at *www.leonardo-vm.org*.

[Stentz94] Stentz, T. "Original D*." In ICRA 94. Available online at *www.frc.ri.cmu.edu/~axs/doc/icra94.pdf*.

[Stentz95] Stentz, T. "Focused D*." In IJCAI 95. Available online at *www.frc.ri.*

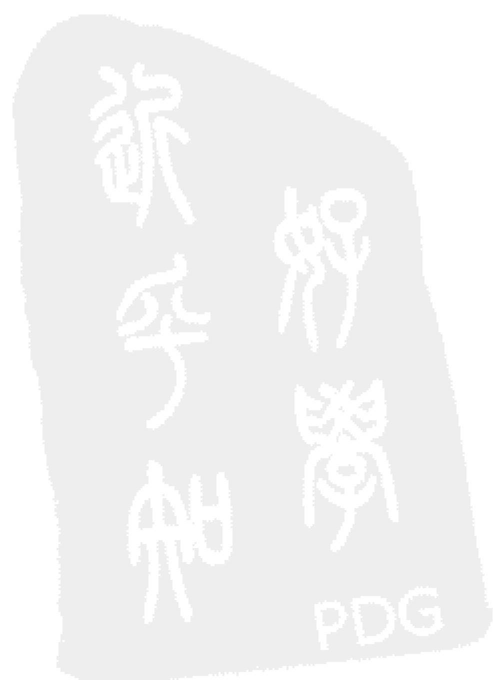
cmu.edu/~axs/doc/ijcai95.pdf.

[Stentz96] Stentz, T. "Map strategies for using D*." In AAAI 96 workshop. Available online at *www.frc.ri.cmu.edu/~axs/doc/aaai96.pdf.*

[Stentz01] Stentz, T. "Constrained D*." In AAAI 02. Available online at *www.frc.ri.cmu.edu/~axs/doc/aaai02.pdf.*

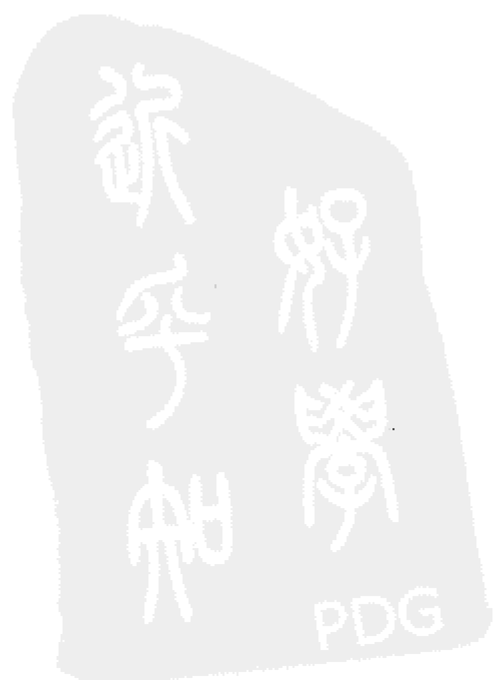
[Stentz98] Stentz, T. "Framed Quad-trees with D*." In ICRA 98. Available online at *www.frc.ri.cmu.edu/~axs/doc/icra98.2.pdf.*

[Stout96] Stout, B. "Smart Moves: Intelligent Path finding." In *Game Developer Magazine*. October, 1996.



第 4 章

物理学


PDG

引　言

Red Storm 娱乐公司，Mike Dickheiser
mike.dickheiser@redstorm.com

下次户外走动时，不妨驻足停留片刻，放眼环顾四周。我们看到的这个世界，是一个充满了运动的世界。无论是剧烈宏大的运动，亦或是那些微妙精细的运动，从人类精巧制造出来的伟大的机械化产品，到大自然中正在下坠的落叶的舞蹈，运动无处不在。每个运动都触及我们的感官，不断地补充着我们对世界的理解，并帮助我们定义对真实的期待。

很自然地，我们也把这个真实的世界作为理解、实验，以及享受其他虚幻世界的一个基础。毕竟，这个世界才是我们知道的世界，我们看到的、感觉到的以及想象到的，都过滤自对这个熟悉的世界的理解。如果幻想的世界不能迎合这个标准世界（遵循它的运转方式），我们就会对幻想的世界感到厌倦、困惑或者失望。作为虚幻世界的创造者，我们当然不能这么做。于是，我们不断地探索，寻找与标准世界更接近的虚幻效果，这样我们才可以沉浸在让我们感到惊讶、兴奋，并深深吸引着我们的新世界中。

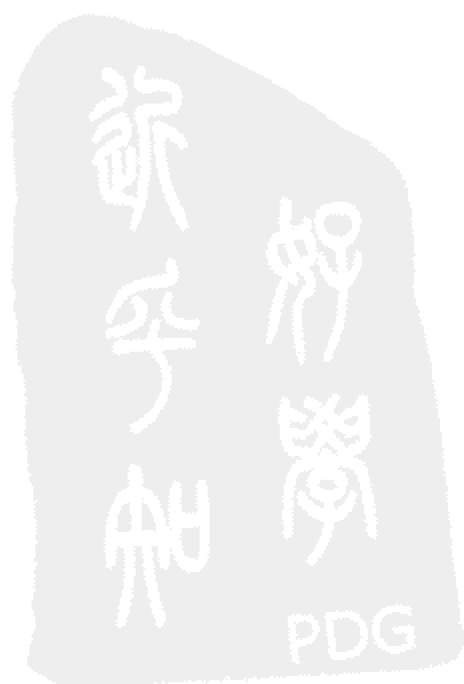
让我们把注意力拉回到计算机游戏上，看看它已经发展到了什么程度。作为游戏开发人员，我们创造的虚幻世界中的真实程度在今年又向前迈进了一大步。游戏中的物理仿真达到了前所未见的逼真程度，可以与我们在现实世界中看到的情形相媲美。很明显，在过去的几年中，我们的技术手段取得了长足的进步。接下来的部分，用几篇精粹文章来展示我们取得的这些进展，并就如何达到更高的层次提出一些想法。

Graham Rhcdes 作为急先锋，全速启动地向我们全面、直观地介绍了空气动力学的方方面面，以及空气动力学在游戏中的各种应用。从空中飞翔，到微风吹拂，接下来是 Rishi Ramraj 的文章。他在文章中讨论了动态草木的仿真模拟，以及其他一些自然现象的仿真模拟，包括水面和树叶运动的仿真。然后，现实主义工具箱又增加了两个新“工具”：Juan Cordero 的布料动画，以及紧随其后的、Maciej Matyka 提出的柔体动画领域的创新技术。

接下来的文章提醒我们，计算机游戏开发的艺术要一分为二地对待：一方面，我们要努力奋斗，将我们对现实世界仿真的真实程度最大化；另一方面，我们要刻苦钻研，提高创建虚幻世界的技巧。Michael Mandel 为“布娃娃”仿真增加了反馈控制系统，巧妙地操纵了游戏角色的“连线木偶”。接下来，我们又来到物理现实主义的另一端，给大家带来两篇有关预定式物理系统（prescripted physics）的文章。Dan Higgins 的文章介绍了预定式物理系统的基础框架，而 Shawn Shoemaker 则向大家介绍了预定式物理系统的广泛应用。

最后一篇文章将我们带回到本章开始的话题：仿真美好的现实世界。这次的观点提供者是 Barnabás Aszódi 和 Szabólcs Czuczor，他们介绍的一些方法可以让我们真实地控制三维赛车仿真游戏中的摄像机运动。

本章的精粹文章涉及多个主题，全面展示了我们在计算机游戏物理领域所取得的惊人进展。与此同时，他们也向我们暗示了很多令人兴奋的、尚未实现的可能性，用那些我们仍然无法触及的能力来揶揄我们，让我们更加沉浸其中，无法自拔。每当可以更接近地审视这个现实世界，我们就会注意到更多的真实细节。这些细节挑战着我们模仿工作的成果，将我们带入一个更高的发展层次。真诚希望本章的这些文章可以成为我们通向下一个更高发展层次的垫脚石，可以鼓舞那些新加入这个学科的人们，让他们继续我们的事业。



4.1 游戏物理中空气动力学的近似计算

美国应用研究联营公司(Applied Research Associates Inc.), Graham Rhodes
grhodes@nc.rr.com

在现实生活中，空气动力特性可以让重于空气的飞行器（重飞行器）自由飞翔，让美妙的曲线球成为可能，让异国情调沙滩上的棕榈树在漂亮姑娘的头顶婀娜婆娑。长期以来，空气动力学一直在游戏中，特别是在飞行模拟类游戏中，扮演着重要的角色。另外，在有些情况下，为了提高特效（例如粒子系统）的真实感，人们也在使用空气动力学的原理。很多游戏开发人员都使用一种专用的方法来处理空气动力学的问题，这种方法更多地依赖于数字实验，而不是可靠的空气动力学原理。本文会向大家提供一些简单的、低 CPU/GPU 开销的、可用于粗略估算的公式。这些公式是从可靠的工程原理推演出来的，并严格地通过风洞实验进行控制。游戏编程人员可以使用这些公式，支持多种游戏类型中各种各样的空气动力学特效。虽然在这里使用了“空气动力学”这个字眼，但实际上，只要速度足够快，这些公式同样可以适用于那些在流体（比如水）中运动的物体。

本文包括两大部分的内容。第 1 部分主要描述了如何将不同的空气动力学本原体作为游戏几何体的代理（proxy），引出简单的公式，用于计算作用于这些本原上的空气作用力。对于那些在空气中或其他流体中运动的物体，只要马赫数不低于 0.75，就可以对它们使用这些公式。由于版面所限，这部分内容中理论方面的说明比较少，我们只提供了空气动力学中关键的几个原理。本文的第 2 部分主要介绍了空气动力学估算公式的应用，即如何在动作类游戏中获得很酷的特效。最后还介绍了相关的几个源代码实现。

4.1.1 背景知识

在下面的内容中，我们将向大家介绍一些有关空气动力学的背景知识。

1. 空气动力载荷与刚体动力学

这里罗列的公式可以帮助大家近似地计算那些作用于刚体的空气动力载荷（aerodynamic load）——力和力矩（也称为转矩、扭矩）。用这些公式计算出来的空气动力载荷正好可以给现有的刚体物体系统使用！

在飞机飞行动力学的工程学界中，存在着 6 个标准的空气动力载荷量：三个力（升力、气动阻力和侧向力）及三个力矩（俯仰力矩、偏航力矩和滚动力矩）。但是，本文处理几何体的方式使得我们可以把这些情况进行简

化，只对升力、气动阻力和一个力矩进行建模。三个力和三个力矩的完整表示只是一种普遍性的概括。图 4.1.1 说明了这三种载荷量的方位。

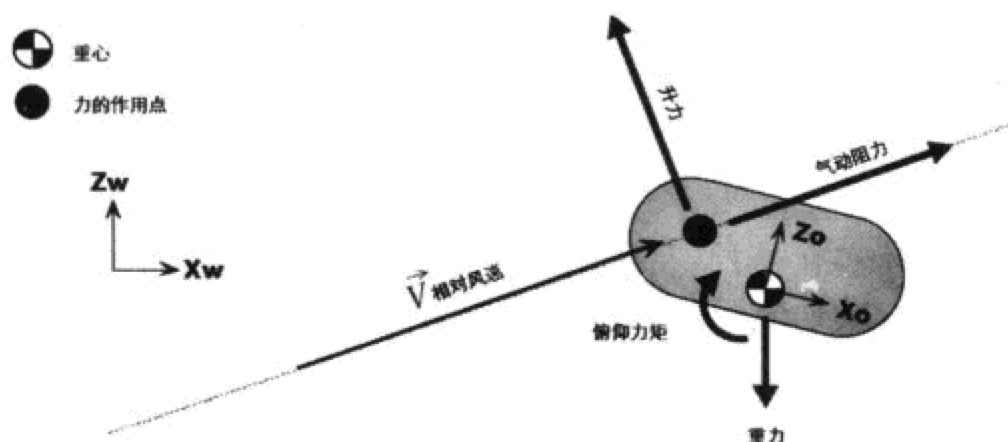


图 4.1.1　升力、气动阻力和俯仰力矩的作用点及方向。气动阻力的作用方向就是相对风（relative wind）的方向，而升力则是垂直于相对风。这两个力的作用点通常都不是物体的重心。俯仰力矩围绕一个坐标轴向发生作用。这个坐标轴穿过重心，且与升力/阻力平面垂直正交

在这里，坐标轴系统 X_w 和 Z_w 表示的是世界空间坐标系统，而坐标轴系统 X_0 和 Z_0 表示的则是物体的局部坐标系统。对于三维物体，当然各自还有一个 Y_w 和 Y_0 的方向。更多的时候，我们可以随意指定这两个坐标系统的方位。物体的方位并不重要，只要我们知道就可以了。要想计算这些力，相对风的方位才是非常重要的，我们将在后面对它进行定义。

我们在所谓的风轴（*wind axes*）坐标系统中定义的这些载荷的方位可能会让人感到吃惊。有人也许一直以为，升力的作用方向是垂直向上的（与重力方向相反），阻力的方向是水平的，而且它们都经过物体的重心，因为关于空气动力学的很多过分简单的介绍都是这样来描述这些作用力的。有人可能也从未听说过俯仰力矩。但请放心，这个风轴坐标表示法是对简单的空气动力学理论最真实、也是最基本的表示。

关于力的计算，我们只需要依赖于物体相对于风的方位，而不是相对于世界的方位。作用于机翼的升力通常都会有一个垂直方向的分力（垂直分量）来抵消物体的重量，使物体可以在水平方向运动，或者带有一个不变的垂直速度。实际上，升力可以作用于任何方向，甚至是水平方向！正是基于对这一事实的认知，才使得我们可以合理地忽略侧向力。举个例子，作为一种特殊情况，侧向力通常只是一个作用在水平方向上的升力。

2. 无量纲形式

把所有这些空气动力学载荷以无量纲系数的形式表示出来，通常是可行的。对于一个给定的无量纲系数的值，公式 4.1.1～公式 4.1.3 就是用来计算这些载荷的公式。

$$D = \text{Drag} = \frac{\rho V^2 S_{ref} C_D}{2} \tag{4.1.1}$$

$$L = \text{Lift} = \frac{\rho V^2 S_{ref} C_L}{2} \tag{4.1.2}$$

$$M = \text{Pitching Moment} = \frac{\rho V^2 S_{ref} l_{ref} C_M}{2} \tag{4.1.3}$$

其中，C_D 是阻力系数（或称为风阻系数），C_L 是升力系数，而 C_M 是俯仰力矩系数。可变的 S_{ref} 是一个参考面积常数，通常是几何体的投影面积，例如一个截面面积，或者一个自上而下的投影面

积。变量l_{ref}是一个参考长度，通常被取为物体的物理尺寸（长、宽、高等），例如机翼的弦长，或者一个球状物体的直径。变量ρ是流体密度，对于游戏应用程序，这个变量通常被取为常量。

最后一个，变量V是流体的速度，是相对于物体（*relative to the object*）的速度。也就是说，它是流体通过物体的速度，在世界空间中测量得到。在世界空间中，给定一个物体上受力点的速度$\vec{V}_{location-of-force}$，以及世界空间中的一个风速$\vec{V}_{wind}$，利用公式 4.1.4 就可以很容易地得到相对于物体的相对风速$\vec{V}_{relative_wind}$。这个情形的几何图示请参考图 4.1.2。在公式 4.1.1 至公式 4.1.3 中，V的量值其实就是$\vec{V}_{relative_wind}$的大小。在计算$\vec{V}_{location-of-force}$的时候，一定要考虑由于物体转动而造成的平移速度，因为这会影响相关的力。具体来讲，就是令$\vec{V}_{location-of-force} = \vec{V}_{point-of-rotation} + \left(\vec{\omega} \times \left(\vec{r}_{location-of-force} - \vec{r}_{point-of-rotation}\right)\right)$。其中$\vec{r}$是一个点的位置，在世界空间中测得；$\vec{\omega}$是围绕着一个轴的转动速度，这个轴穿过点$\vec{r}_{point-of-rotation}$。$\vec{\omega}$以每秒转过的弧度数来计量，参见公式 4.1.4。

$$\vec{V}_{relative_wind} = \vec{V}_{wind} - \vec{V}_{location-of-force} \tag{4.1.4}$$

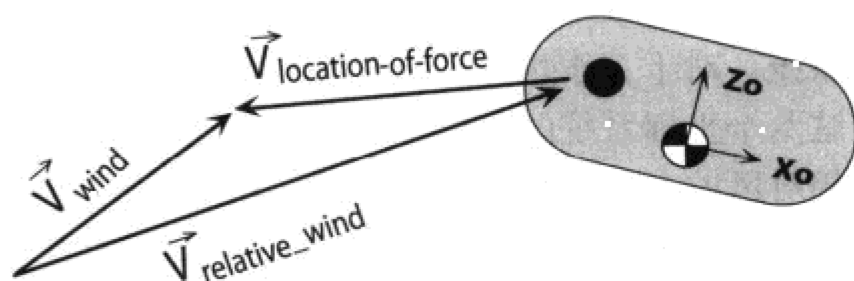


图 4.1.2 相对风的几何示意图

3. 流体属性与标准大气

这里罗列的公式取决于不同流体的物理属性。这个流体可能是空气、水或其他的什么。在海平面，空气有一个平均密度ρ，约为 1.225 kg/m^3。这时候，空气的动态黏性μ，约为 1.789×10^{-5}N·s/m^2。在 20 摄氏度时，纯净水的平均密度为 1000 kg/m^3，动态黏性约为 1.0×10^{-3} N·s/m^2。

对空气而言，在任何给定的时间片段中，它的属性会随着海拔、气候等条件的变化而变化。国际上有一个被称为“标准大气”（各国有各自版本的标准）的模型，表示了一定范围的海拔高度内平均的空气属性。如果在互联网上搜索这个词汇，可以找到许许多多的链接，其中也包括各种空气属性的列表，以及查询这些列表需要的软件。

4. 空气动力学几何体

虽然，在对纳维叶-斯托克斯（*Navier-Stokes*）方程所描述的各种流体进行实时仿真的工作上[Stam03，Lander02]，我们已经取得了一些进展，但是，如果是在游戏中进行烟气、云彩和水流的仿真，当目标只是为了找到作用于一个刚性物体上的合力及力矩时，使用这些方法就显得有点小题大做。对游戏而言，在很多情况下，一个非常近似的解决方案就足够真实了。为此，本文剩下的内容将讨论一些空气动力学几何体的空气动力载荷的计算方法。在这些几何体都有着简单的形状。对于这些简单的形状，工程学界早在很久以前就开发出了闭形的、简单的代数方程。在这些研究成果中，有些还可追溯到几个世纪之前。为了在游戏中应用这些方程，只需

要选择最为合适的空气动力学几何体，然后针对该几何体来应用这些方程。通常情况下，所选择的几何体是与游戏对象的形状最为接近的一种。对于一些更复杂的形状，也可以把它们看成是由若干个空气动力学几何体组成的（例如，一个球体加上一个简单的翼，链到一起成为一个刚性凝固体），来近似地计算出它们的载荷。当然了，这个方法通常只能产生一个非常近似的一阶近似估算或零阶近似估算，忽略了物体之间的干涉效应及其他一些因素。但幸运的是，这些近似估算的计算开销确实非常低，而且在效果上通常也是绝对有说服力的！

4.1.2　钝体上的作用力

我们将钝体（*bluff body*）定义为不是细长形或流线形，也没有翅膀可以产生升力的物体。为了描述的方便，本文考虑的钝体是指那些类似球体或者圆柱体的物体，它们的中心轴垂直于相对风。这类物体包括不规则的泡状体、立方体和管状体等。我们认为作用于这些钝体上的力，其作用点位于物体的质心（centroid）。

1. 阻力（空气动力几何体：球体）

对于钝体，C_D 主要是一个无量纲参数的数学函数。这个无量纲参数被称为雷诺数（*Reynold's Number*），由公式 4.1.5 定义。雷诺数表示流体中的惯性作用力与黏性作用力的比值。

$$\text{Reynold's Number} = R_e = \frac{\rho V l_{ref}}{\mu} \quad (4.1.5)$$

其中有 3 个变量是我们认识的。第 4 个参数 μ 表示的是流体的动态黏性（单位是 ft/长度的平方）。图 4.1.3 表示的是在雷诺数的一个庞大数值范围内，一个球体的 C_D 变化情况。

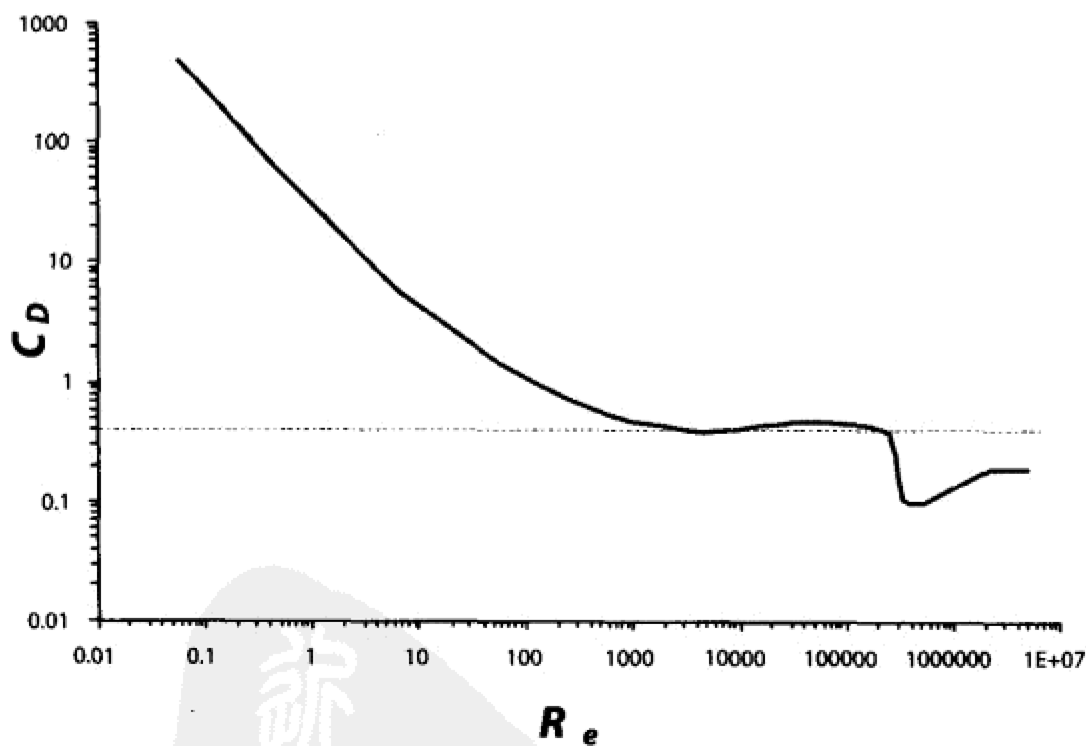


图 4.1.3　在一个雷诺数的范围内，一个球状钝体的 C_D 值的变化情况

为了计算作用于类似球体的钝体上的阻力，首先我们将 l_{ref} 设置为直径，然后计算它的 R_e；接下来，从图 4.1.3 中选择一个 C_D 的值；最后，将 C_D 带入公式 4.1.1，将其中的 S_{ref} 设置为物体有代表性的那个面的前投影面积的大小。

为了方便起见，White[White74]给出了一个简化的公式（参见公式 4.1.6）。当 R_e 的值小于 200 000 时，用这个公式来估算球体上的阻力是相当精确的。这个公式是通过实验数据的曲线拟和得到的。

$$C_{D,sphere} = \frac{24}{R_e} + \frac{6}{1+\sqrt{R_e}} + 0.4 \tag{4.1.6}$$

如果 R_e 的值大致介于 2 000 和 200 000 之间，那么我们将 C_D 取为 0.4 即可。对于更大的雷诺数，可以使用 0.1～0.4 之间最合适的一个值。

在有些情况下，球体并不是表示某个给定形状的最佳选择。虽然对于其他一些不同形状的物体，并没有相应的 C_D 和 R_e 的对照表，但针对这些物体，还是有一些资源可以为我们提供一个合理的 C_D 测试值。特别值得一提的是，Virginia Tech 公司的 M.S. Cramer 教授提供的一个基于 *JavaScript* 的 *Bluff Body Drag Calculator*（钝体阻力计算器）[Cramer98]。这是一个优秀的工具，也是我们极力向大家推荐的一个资源。

2. 我敢打赌，你一定认为阻力会随着速度呈线性变化，对吧

正如前面所描述的，从定义上讲，阻力和其他的空气动力，与速度的平方和一个无量纲系数的乘积成正比。无论是什么样的流体，以什么样的速度流动，也不管是作用于什么样的物体，这种比例关系总是正确的。这是一个定义。但是，正如你从本文中了解到的，在某些情况下，阻力也可以随着速度呈线性变化。这里怎么会出现两种可能呢？事实就在于 C_D 的变化性。事实证明，当 R_e 的值近似地小于 1 000 时，C_D 的变化是与 R_e 成反比的。其结果就是，在公式 4.1.1 的分母上增加了一个额外的 V，使得阻力成为了 V 的一个线性函数。但是，公式 4.1.1 所定义的这种关系也是完全正确的。

3. 自转钝体上的升力（空气动力几何体：圆柱体）

升力是垂直于相对风的一个力。物体表面的流加速度和流减速度造成的流体压力差，就产生了升力。虽然，关于升力的产生，其更详尽的物理学上的解释已经超出了本文的范畴，但是我们却可以利用理论空气动力学早先的研究成果得到一个方便的公式，用来估算钝体的升力。业界对纳维叶-斯托克斯（*Navier-Stokes*）方程有很多经典的简化模型，其中之一就是所谓的线性势能流（*linearized potential flow*）模型。利用这个模型，一个完整的流场（flow field）可以表示成若干个基本流场的叠加（或总和）。流体中一个普通物体周围的流可以近似为一个背景流（background flow）。在背景流中，我们增加了一个所谓的环流（*circulation* flow），将流体的速度进行了平衡（而且不产生升力）。环流表示的是流体在物体的一个侧面的加速度和在另外一侧的减速度。要想真正理解环流的概念，最简单的方法是把它想成一个向心流（concentric flow），一个涡流，叠加在背景流之上。涡流不断地增加物体某一侧的流体速度，产生更低的压力；同时不断地降低物体另外一侧的流体速度，产生更高的压力。物体两侧的这个压力差就导致了升力的产生。

早在 20 世纪 90 年代，就有两位科学家（Kutta 和 Joukowski）分别独立地测定出来，如果相对风是非零的，而且还存在一个环流，那么就存在一个升力，并可以用一个简单的公式进行量化。公式 4.1.7 就是库塔-儒可夫斯基定理（*Kutta-Joukowski Theorem*）的一个归纳，它用环流定义了升力。

PDG

$$\vec{L}_{per_unit_length} = \rho \vec{V}_{relative_wind} \times \vec{\Gamma} \tag{4.1.7}$$

其中，$\vec{\Gamma}$ 是一个向量，表示每个单位长度的环流，有一个确定的方向；而合力 $\vec{L}_{per_unit_length}$ 则是每个单位长度上的升力。这里的“长度”是沿着环流方向上物体的长度。在现实世界中，环流是随着物体的长度变化的，所以公式 4.1.7 必须对物体的长度求积分，才能得到全部的升力。

物体曲率的几何不对称性（例如，机翼上部的曲率要大于其下部的曲率），就会造成环流的产生，旋转运动——自转——同样也可以产生环流。棒球运动中的曲线球，主要就是因为自转的棒球周围的环流所引发的升力所造成的。在棒球的某一侧，沿着相对风的方向上（低压力），表面摩擦加速了空气分子的运动；而在棒球的另外一侧，在与相对风相反的方向上（高压力），表面摩擦降低了空气分子的运动的速度，这样就产生了环流。自转棒球上的环流，其方向就是棒球自转的方向，如图 4.1.4 所示。

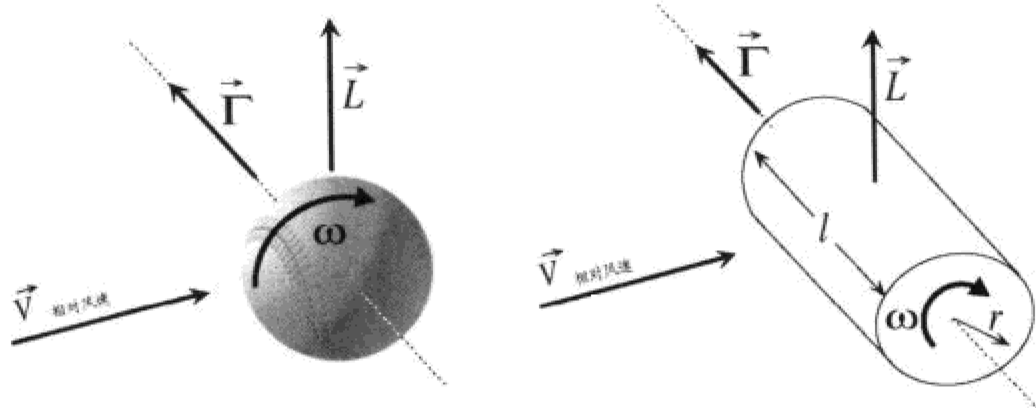


图 4.1.4 由于表面摩擦，在自转的棒球和圆柱体上产生的环流

在现实世界中，一个普通物体周围的流体流动是非常复杂的，其环流的情况也是非平凡的。为了大大地简化在游戏中的应用，我们可以把物体简单地看做是一个圆柱体。空气动力几何体就是一个边界圆柱体，位于自转轴上。在这种情况下，可以近似地得到一个圆柱体上每个单位长度的环流。公式 4.1.7 的结果乘以圆柱体的长度，就可以近似地得到总的升力。对于一个半径为 r，自转的速度为每秒钟 ω 个弧度（正的或负的）的圆柱体每个单位长度上的环流可以用公式 4.8.1 计算出来。

$$\Gamma_{cylinder} = 2\pi\omega r^2 \tag{4.1.8}$$

这样的话，对于被这个边界圆柱体（长度为 l）包围着的钝体，其总的升力就可以用公式 4.1.9 计算出来。其中，$\vec{e}_{spin}$ 是一个单位向量，表示圆柱体的自转轴。而 $\omega\vec{e}_{spin}$ 就是在这个弧度上每秒的旋转速度。其中的附加因子 0.785 是为了近似地模拟在有限长度的圆柱体上发生的三维损耗。圆柱体的长度越长，这个公式的结果就越精确。

$$\vec{L} = 0.785 l \rho \vec{V}_{relative_wind} \times (2\pi\omega r^2 \vec{e}_{spin}) \tag{4.1.9}$$

如果把与棒球自转相关的力看做是马格努斯效应（*Magnus Effect*，也称为 *Robin's Effect*，罗宾效应）造成的，也是非常正确的。早在 Kutta 和 Joukowski 提出数学公式之前，业界就已经有了这两个提法。可是在现实世界中，除了环流，还有很多其他的因素在起作用。

4.1.3 流线体上的作用力

在下面的内容中，我们看看作用于流线体上的几个作用力。

1. 翼状体上的升力和阻力（空气动力几何体：四角平板/翼状体）

有些物体的形状像飞机的机翼，特别擅长制造环流，也因此可以特别容易地制造出升力。为了叙述方便，我们把那些基本扁平，与相对风之间的角度在 10～15 度之间的物体，看成是一个翼状体，也就是一个高效的升力体（lifting body）。工程学界又衍生出了薄翼理论（*Thin Wing Theory*），推演出了很方便的公式，用来计算薄翼的 C_D 和 C_L。除了真实的飞行模拟外，这些公式非常适合游戏应用程序。图 4.1.5 中说明了计算这些公式所需要的几何参数。

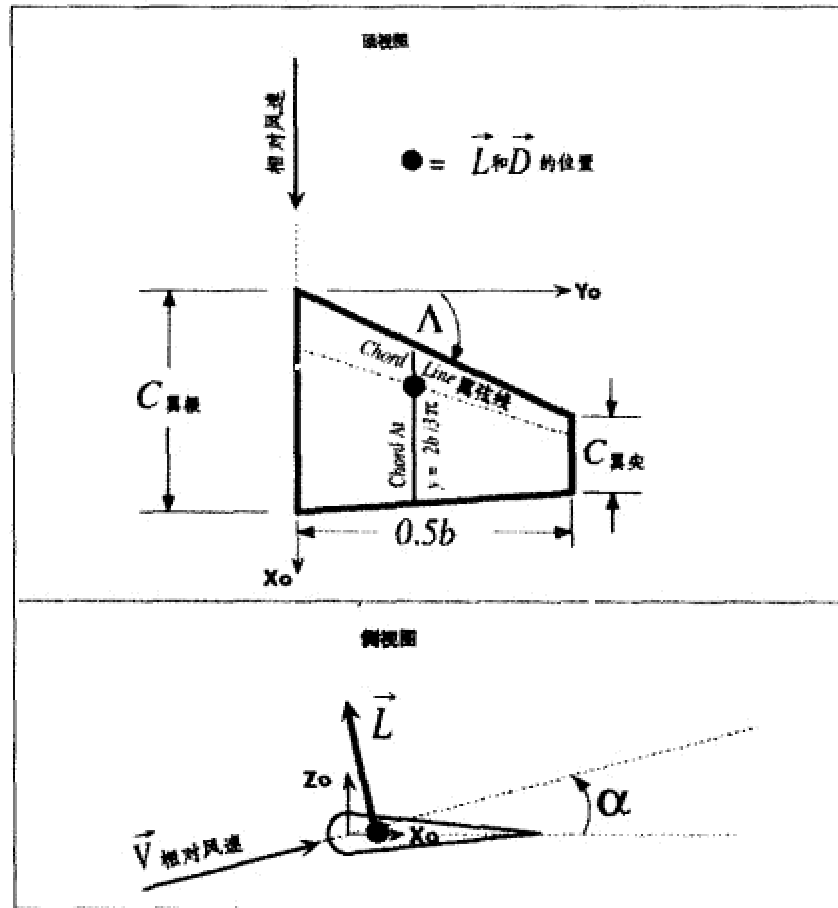


图 4.1.5 翼状体的几何图示。物体所在坐标系统的 *XY* 平面位于表示这个物体的中心平面上

其中，A 是机翼前缘掠角，C 的值分别是翼根和翼梢的弦长，0.5b 是半翼展长度，α 是相对于 $\vec{V}_{relative_wind}$ 的机翼攻角。应该感谢 Hale[Hale84]，因为他才有了一个简单的公式，可以近似地计算这类攻角较小的（小于 10～15 度）翼状体的升力，参见公式 4.1.10。

$$C_L = C_{L,0} + \frac{\pi A \alpha}{1 + \sqrt{1 + \left(\frac{A}{2\cos\Lambda}\right)^2}} \tag{4.1.10}$$

要注意的是，公式中的α的单位是弧度。其中，$C_{L,0}$ 是攻角为零时产生的升力。当机翼上表面和下表面的曲率不一样时，该值就是个非零值。对于游戏中的近似解决方案，我们可以假设它为零。A 是展弦比，等于翼展除以平均弦长。如果α大于 10～15 度，那么 C_L 的值就会随着α进行非线性变化。对于α的值大于 15 度的情况，读者可以参考 Bertin 和 Smith[Bertin79]，或者 Raymer[Raymer92]的文章，了解如何获得 C_L 更真实的行为变化。

作用于翼状体上的阻力包含两个主要的分力：一个是废阻力（parasite drag），这个力与钝体受到的阻力是一样的；另外一个是涡流诱导的阻力（induced drag），这是升力产生时的

一个副作用。根据薄翼理论，我们可以通过一个抛物线型的阻力极线（*drag polar*），近似地计算出翼状体的阻力，参见公式 4.1.11。

$$C_D = C_{Do} + \frac{1}{\pi Ae} C_L^2 \qquad (4.1.11)$$

其中，C_{Do} 是废阻力（parasite drag）分力，剩下的那个数据项就是诱导阻力（induced drag）分力。通常来讲，由于机翼是流线型的，因此 C_{Do} 非常小。一个比较合理的值是 0.045，当然，大家可以根据自己的需要来调整。A 是展弦比，e 就是所谓的“奥斯瓦尔德翼展效率系数(*Oswald span efficiency factor*)”。除了飞行模拟类游戏外，其他的游戏将 e 取为 0.8 就可以。

由于这些力对力矩载荷是有影响的，所以一定要注意升力和阻力的作用点，这一点非常重要。为了确定升力和阻力的中心点，首先要找到 $y = 2b/3\pi$ 处的弦长。这个位置承担着半椭圆的展向升力分布。升力和阻力近似地作用于翼弦线和四分之一弦线（*quarter-chord line*）的交叉点上。这个交叉的地方就是沿着整个翼展，在机翼前缘后面 1/4 弦长处。还有一点至关重要，要把力的作用点上的平移速度考虑进去，这个平移速度是由于物体的转动速度而产生的。该速度分量对基于物理的俯仰阻尼（Pitch damping）是有影响的，如果没有它，仿真就会表现出不稳定性。

我们还要做出两个重要的观察。第一，公式 4.1.10 假设的前提是机翼对称于 XZ 平面，也就是说，有两个对半的机翼组成的全翼。如果不考虑这个事实，假设我们真的只有半个机翼，例如，导弹弹体上某一侧的尾鳍（通常是一个有两直角的四边形），这时候只要把公式 4.1.2 中的 S_{ref} 选定为模型中的机翼部分的面积，公式 4.1.10 仍然可以很好地工作。第二，如果确实有一个全翼对称于 XZ 平面，那就需要包含左翼和右翼这两个翼上的升力和阻力，将合力加倍。大家要注意，左右两个翼的升力合力与阻力合力，其作用点的 X 坐标与左右两个翼的 X 坐标一样，但 Y 的坐标值为 0。也就是说，升力和阻力的作用点位于两个对称翼之间的中心线上。

Raymer[Raymer92]提供了一个更容易理解的介绍，解释了翼状体上的升力和阻力，并提供了更多的、依旧很简单的公式。强力推荐这个资源，很多工程大学的图书馆里应该可以找到这个文献。

2. 俯仰力矩

在现实世界中，升力和阻力并不是在物体的重心上产生的。它们是物体表面上的压力和摩擦力分布的综合结果。这两个力的实际质心从来不会与物体的重心相重合。为了便于计算刚体运动方程的时间积分，我们才会在刚体仿真中使用物体的重心。如果将升力和阻力应用在前面给定的作用点上，就可以得到机翼产生的俯仰力矩的一个合理的近似值。

3. 一般力矩

仔细思考一下，就会意识到，任何一个空气动力（升力、阻力，以及升力的一个变种——侧向力）都可以产生一个围绕物体重心的力矩，而这些力矩都是在仿真中必须要考虑的。如果某个物体有一个钝体部分产生了阻力，而从这个物体的重心到钝体部分的中心的向量不与相对风平行，那么这个阻力就会在重心上产生一个力矩。任何方向上的升力通常都会产生最大的力矩。最后的结论是，如果可以确定相关的力都作用在正确的位置上，那么对空气动力力矩的近

PDG

似计算，其结果也会很自然。

4. 细长体上的作用力（空气动力几何体：细长旋转椭圆或导弹）

有些物体，它们既不是钝体，也不像机翼或者平板。这些物体就包括所谓的细长体。这些物体的形状类似于胶囊体或导弹，具有很高的长度-直径比（length-to-diameter ratio，或称 *slenderness* ratio，长细比）。它们在空气中飞行时，其长度轴近似地与相对风成一条直线。在现实世界中，这些物体可以产生升力、阻力，以及俯仰力矩。但不幸的是，由于篇幅的限制，我们在这里无法深入地探讨细长体的空气动力学。但是，Karamcheti[Karamcheti80]在他的书中给出了比较全面的理论介绍。必要的时候，可以使用 Karamcheti 公式的下面这个变种，近似地计算俯仰力矩。当α近似地小于 10 度时，计算的结果是相当逼真的。公式 4.1.12 定义了俯仰力矩系数的大小。

$$C_{M,slender_body} = |2\alpha| \text{（}\alpha\text{的单位是弧度）} \tag{4.1.12}$$

其中，α是细长体的轴与相对风之间的夹角。在这种情况下，公式 4.1.3 中的$S_{ref}l_{ref}$就取值为细长体的体积。俯仰力矩作用于物体的重心，而这个力矩的方向则由$\vec{V}_{relative_wind} \times \vec{A}_{slender_body}$的向量积给出。其中，$\vec{A}_{slender_body}$是细长体的最前端到最后端的一个轴，该轴通过物体的重心。请注意，这个力矩是不稳定的。也就是说，如果细长体变得有些偏离相对风了，这个俯仰力矩就会使细长体更加偏离相对风。为了使细长体可以稳定地运动，我们在重心的后面增加了尾鳍。尾鳍其实就是尾翼，会抵消俯仰力矩，使物体保持稳定。

为了图方便，我们假设细长体不产生升力，所以可以选择C_L=0。对于C_D，可以尝试 0.1 或更小的值，而S_{ref}的值则可以设定为垂直于物体轴的最大横截面积。

4.1.4 应用实例

为了说明如何应用目前所讲的这些原理，我们看三个不同的应用实例：一个风动的粒子风暴，一个曲线球的仿真，以及一个简单的飞机仿真。这三个例子的实现代码都提供在随书光盘中。

1. 风动的粒子风暴

这个例子使用了球形钝体的阻力公式来仿真暴风（一个简单的龙卷风）中的粒子。在这个例子中，我们使用了一个所谓的涡流势场和一个强度对暴风建模。强度会随着海拔高度的二次方而变化。暴风的中心，也就是涡流的基点，为（0,0,0）。世界坐标 Z 轴的方向表示的是海拔高度。我们假设暴风有一个强度，在海平面S_0到 500 米的S_{500}之间随着海拔高度的二次方变化，参见公式 4.1.13。

$$S(z) = S_0 + (S_{500} - S_0)\frac{z^2}{500^2} \tag{4.1.13}$$

根据这个公式，对于局部风速$\vec{V}_{wind}$，在世界坐标中任意一个点上由暴风造成的风速由公式 4.1.14 给出。

$$\vec{V}_{wind,x}(x,y,z) = \frac{yS(z)}{2\pi r};\ \vec{V}_{wind,y}(x,y,z) = \frac{-xS(z)}{2\pi r};\ \vec{V}_{wind,z} = 0 \tag{4.1.14}$$

其中，r 是该点到涡流核心轴的垂直距离。在这种情况下，由于涡流位于起始位置，因此

$r=\sqrt{x^2+y^2}$。在 r 的值特别小的时候，我们一定要十分小心。Bertin[Bertin79]提供了一个更容易理解的涡流势场的介绍。

对于每个粒子上阻力的计算，首先要计算在粒子当前的位置上，由风暴造成的局部风速 $\vec{V}_{wind}$；然后计算相对风，最后再计算 C_D 和实际的阻力。这个阻力与物体的重力相加，获得作用于粒子上的合力。这个合力可以应用于一个简单的粒子物理仿真程序。

2. 曲线球的仿真

这个例子使用了计算球状钝体阻力的公式，以及计算自转钝体升力的公式，来仿真棒球比赛中的曲线球。在这个例子中，假设风速为零，这样的话，相对风就是棒球当前速度的反方向。这个例子在前面的公式 4.1.4 中就有过说明，不同的是自转轴是垂直的，这样就产生了一个水平方向的升力。可以把投球速度设定在 70～90 英里每小时范围之内，自己可以随时调整（由 a 键和 s 键控制）；自转速率的调整范围为每分钟−100 转到每分钟 100 转（由+键和−键控制）。投出的球总是从水平开始，沿着−X 轴（飞向本垒板）飞行，而自转轴就是 Z 轴。重力则沿着−Z 轴的方向（垂直向下）作用于球体。按下 g 键就可以开始仿真，按下 p 键可以暂停，按下 r 键则会重新开始。

3. 一架简单的纵向稳定飞机

这个例子向我们演示了一个简单的飞机上的升力和阻力的计算（参见图 4.1.6）。这个飞机是鸭式风格的，就是说，它的主翼是在水平尾翼（horizontal stabilizer）的后面，而重心就位于这两个翼之间。当两个机翼各自升力的中心恰当地位于相对于重心的位置上时，由此产生的力矩是可以稳定机身的。这个例子就是向大家展示该力矩的稳定特性。为了简单起见，这个例子忽略了众所周知的“下洗”现象（因机翼所产生的下降气流）。这个现象会使得前翼的存在造成后翼攻角的减小。在计算两个机翼的相对风时，这个例子会将飞机的转动也考虑在内，同时还考虑了基于物理的俯仰阻尼。后者在飞机的攻角比较小时，可以使飞机保持动态的平稳。飞机产生的俯仰运动是很逼真的，这些俯仰运动是根据升力和阻力自然产生的。

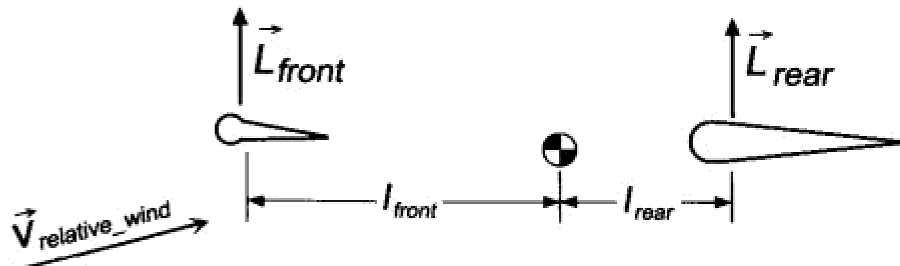


图 4.1.6　一个简单的鸭式纵向稳定飞机的空气动力配置

在这个例子中，可以调整前翼相对于后翼的方位角（用+键和−键来控制）。在调整这些参数的时候，要注意飞机方位在发生变化，但一段时间之后，就会达到转动平衡。这就要归功于飞机两个机翼上俯仰力矩的平衡特性。这是一架纵向稳定的飞机。

4.1.5　总结

本文介绍了一系列空气动力学的基本概念，并提供了一些简单的公式。读者可以在自己

的游戏中，利用这些公式来计算有趣的空气动力学特效。根据游戏平台的不同，实现空气动力学特效的选择方法也不同。如果要使用定制的、免费的，或者授权使用的物理引擎，那就需要实现一些回调函数。给定当前的状态和风速，用这些回调函数可以计算一个刚体中心处的空气动力载荷。一旦计算出了相对风坐标系统中的载荷，就可以把它们映射到世界空间中，最后应用到刚体上。物理引擎只是简单地把这些载荷集成到它的数值时间积分中，再不会帮我们做什么其他的工作。随着可编程图形硬件的出现，我们可以在 GPU 上执行一些有限的物理计算。这里提供的公式非常简单，可以在使用当前 GPU 或未来 GPU 产品的顶点着色器中实现。在粒子系统中增加空气动力学，是一个非常引人注目的方法。这些简单的公式的计算开销非常小，甚至可以在手持游戏平台上进行有限的使用。这里的一个挑战是，如何对公式进行优化，让它们在浮点计算功能有限的 CPU 上跑得更快；或者利用定点数学进行优化。

虽然这些公式偶尔会基于一些比较苛刻的假设前提，并忽略了很多高阶的物理效应，比如物体之间的干涉效应、地面效应等，但是我们还是可以合理地使用这些公式，提供一些开销非常低的幻景，让游戏世界看上去更加真实。单有空气动力学是无法做成一款游戏的。但是，如果能把空气动力学与更为传统的视觉特效、物理特效和动画特效结合使用，它就会为更丰富的游戏体验做出自己的贡献。最有趣的效果都是通过不断的实验和试玩来获得的。因此，可以用任何意想不到的方式应用本文所介绍的这些技术。利用空气动力学，让下一个游戏世界变得生机勃勃吧！

4.1.6 参考文献

[Bertin79] Bertin, John J., and Michael L. Smith. *Aerodynamics for Engineers*. Prentice Hall, 1979.

[Cramer98] Cramer, M.S. Bluff Body Drag Calculator. Available online at *http://www.fluidmech.net/jscalc/cdcal26.htm*. 1998.

[Hale84] Hale, Francis J. *Introduction to Aircraft Performance, Selection, and Design*. John Wiley & Sons, 1984.

[Karamcheti80] Karamcheti, Krishnamurty. *Principles of Ideal-Fluid Aerodynamics*. Robert E. Krieger Publishing Company, 1980.

[Lander02] Lander, Jeff. "Taming a Wild River." Presented at the Game Developers Conference 2002. Available at *http://www.darwin3d.com/confpage.htm*.

[Raymer92] Raymer, Daniel P. *Aircraft Design: A Conceptual Approach*. AIAA Education Series. American Institute of Aeronautics and Astronautics, Inc., 1992.

[Stam03] Stam, Jos. "Real-Time Fluid Dynamics for Games." Presented at the Game Developers Conference 2003. Available online at *http://www.dgp.toronto.edu/people/ stam/reality /Research/pdf/GDC03.pdf*.

[White74] White, Frank M. *Viscous Fluid Flow*. McGraw-Hill, 1974. Available online at *http://hyperphysics.phy-astr.gsu.edu/hbase/fluids/kutta.html*.

4.2 动态青草的模拟和其他自然环境特效

沃特卢大学，Rishi Ramraj
thereisnocowlevel@hotmail.com

优秀游戏（如《半条命 2》（*Half Life 2*））的开发工作表明，游戏环境的物理表现在身临其境的玩家体验中扮演着非常重要的角色。但同时，仿真模拟复杂的自然环境会涉及大量的额外计算开销。

通过对现有的水面模拟算法进行重新的审视，可以减少额外的计算工作量。这种方法可以把内存的需求降低到 50%，同时不会降低（如果不需要提高）模拟效果的质量。这个方法还可以扩展到其他自然环境特效的模拟，例如风吹草低和树叶婆娑的自然现象。

本文有 3 个目的。首先，展示一种算法，可以优化特定环境下水面仿真的内存需求。其次，用最后的结果模型提供一个稳健的、易于实现的算法，用于模拟动态草木。最后一点，这些方法是普遍适用的，相同或不同物体群组之间的变化所产生的相互影响，也可以用这些方法来模拟类似的特效。

4.2.1 水面特效

本文提供的用于模拟自然环境特效的方法，是从《游戏编程精粹 4》中的一个算法（iWave 算法）演化而来的[GPC04]。这个算法的简化版本可以在[Willemse00]中找到。这两种算法在本质上是一样的，但为了简化起见，我们后面再深入讨论。

1. 算法

iWave 算法通过一系列的网格节点近似模拟了一个水体平面。模拟网格上的每个点，其垂直方向的移动可以模拟水面波纹的效果，如图 4.2.1 所示。

为了得到这个系统的动画效果，需要使用两套网格，分别代表一个时间段之前和之后的两个状态。一个网格保存着系统当前的高度值，然后利用这些高度值计算出下一时间段的高度值，并将之保存在另外一个网格中。使用两个网格可以确保对$(i-1,j)$点的计算不会影响到(i,j)点。

为了表现出波浪在这个水体表面运动的效果，我们将每一个网格点(i,j)孤立地来看。每个点的高度值是这样计算出来的：将该点周围所有点的高度值相加，除以 2，再减去该点当前的高度值：

PDG

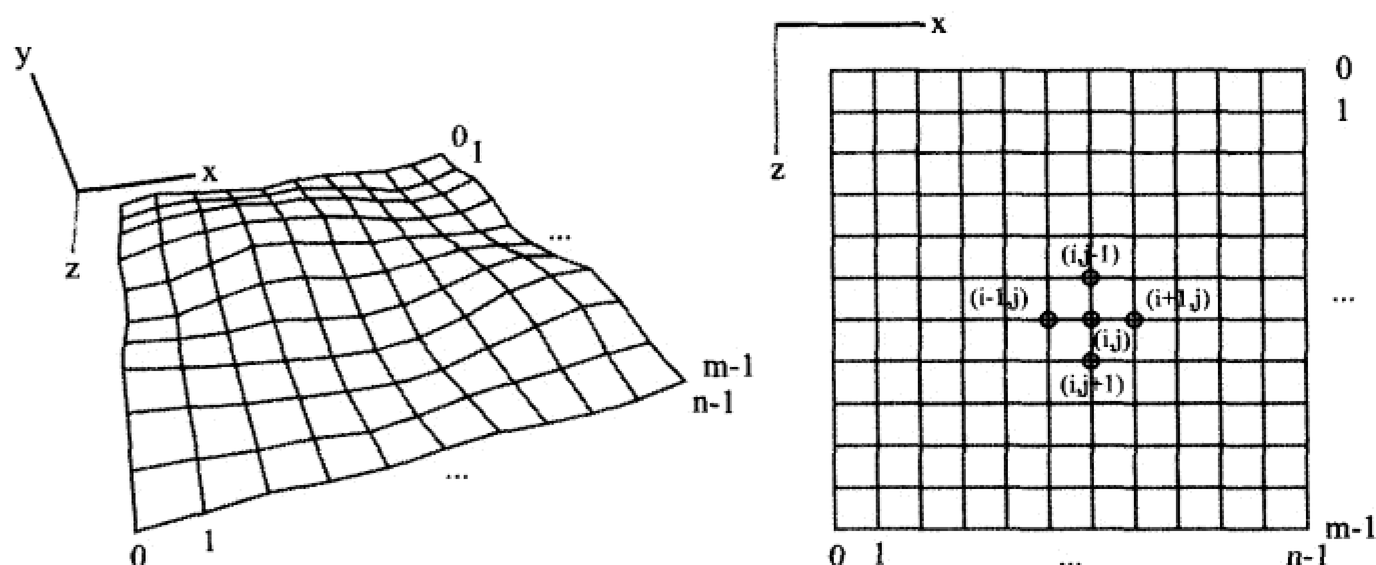


图 4.2.1　*iWave* 算法近似模拟了一个水体平面。三维（左图）和二维（右图）的描述如图所示

```
res_h[i,j] = (src_h[i-1,j] +
              src_h[i+1,j] +
              src_h[i,j-1] +
           src_h[i,j+1] ) / 2 - src_h[i,j];
```

最后得到的就是波浪在水体表面运动的效果。波浪的振幅保持恒定，所以必须将能量从系统中去除。解决的方法是每次系统更新时，简单地从当前的高度值中减去一小部分的能量：

```
res_h[i,j] -= res_h[i,j] * damp_factor;
```

为了在水体表面生成波纹的效果，我们可以简单地将表面点的高度值设置为一个非零的值。得到的效果是产生的变化会传遍整个系统，影响所选点周围的所有点。而波浪则会在这个选定的点之外产生细小的波纹。

至于为什么要使用这些公式，如何使用这些公式，以及为什么这些公式会产生这样的效果，我们可以参考[Willemse00]，[GPG04]中则有更详细的讨论（包括如何在一个变化的时间段里生成动画）。该方法把水体看成是一个守恒的整体，水体表面在某个位置上升，就会在另外一个位置下降。如果有障碍体造成波浪的反弹，[PGP04]还讨论了如何对障碍体进行数学建模。

这些细节，已经超出较讨论的范围。下面我们看一下算法分析。

2. 算法分析

直观地讲，我们把这个系统看做是一个使用平面网格的近似体，而且系统本身也是这样进行渲染的。另一方面，如果把这个系统看成一个网络，我们还可以做一些有趣的观察。

这个网络是由几个节点组成的。每个节点都是一样的，最多有 4 个，最少有 2 个相连的邻近节点。每个节点都保存着高度值，它们在网络中的逻辑位置用来确定它们在渲染器中的物理位置。

网络中的链接可以告诉系统彼此连接着的节点之间发生的变化。位于一个链接两端的两个节点可以相互影响。因此，在某个时间段，如果改变了系统中某个节点的高度，那就会影响其邻近的邻居节点，并在下一个时间段受到邻居节点的反作用。一段时间之后，其结果就是一个节点上发生的变化所造成的影响在整个系统中的传播。

先前定义的函数是每个链接的一部分，它定义了某个节点的属性是如何影响其相邻节点的属性的，以及变化在网络中传播的特征。这个函数是整个模拟操作不可或缺的一部分。例如，如果

将节点的高度取成平均值，那么整个模拟效果就变成了一滩污渍；再比如，如果没有以前提到的 damp_factor 这个因子，这个模拟操作就会变得非常离散，非常零星，而波浪也永远不会消失。

3. 优化

[GPG04]里面提到，iWave 算法对生成大规模的环境波浪（如开放的海洋波浪）并不是非常有效。这里的优化算法提供了一个简单的环境波浪的近似模拟，同时降低了对内存的需求，使对海洋的模拟成为可行。它还在变化传播模型中添加了一个有趣的急转。

优化的工作非常简单：只使用一个平面网格，而不是两个。这个方法保留了两次计算之间时间段的概念，同时减半了系统对内存的需求。当更新过程在这一个网格上推进的时候，某个点(i,j)的计算是那些已更新点（如$(i-1,j)$）和当前点（如$(i+1,j)$）的混合值。波浪不可能从某个点向所有方向传播，相反，它们只会向需要更新的方向传播。这样，水面模拟的结果就是好像有风吹过一样，这和大洋中的大规模环境波浪非常类似，如图 4.2.2 所示。

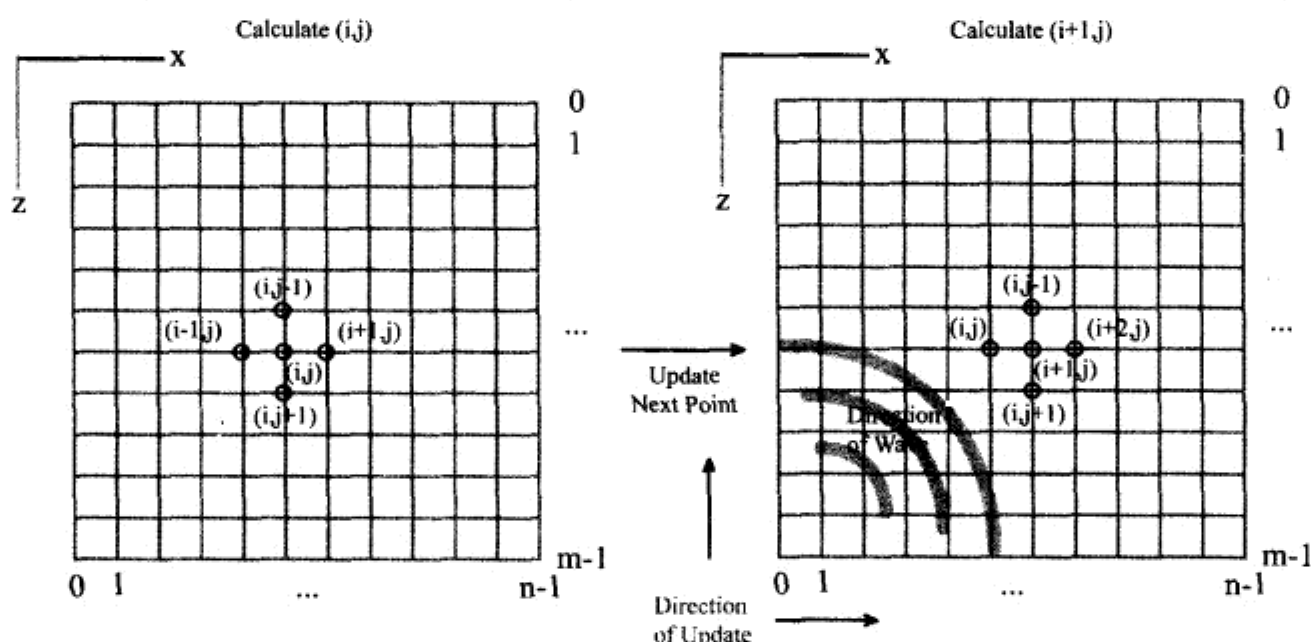


图 4.2.2　*iWave* 算法的优化。当更新操作在网格中推进时，我们要计算点(i,j)的值，然后再用这个值来计算点（i+1, j）的值。

为什么会这样呢？我们当然要抛弃原先算法中使用的“双缓冲”方法。原始算法之所以要使用“双缓冲”，是为了确保某个时间段的数据不会被下一个时间段的计算冲掉。但是，是否存在这种问题，取决于模拟的目标。如果我们的目标是要做到非常精确，需要某种特殊的模拟仿真，那么放弃第 2 个缓冲就不是一个好的选择。相反，如果我们的目标只要求“看上去很美”，那么放弃第 2 个网格，不仅可以降低内存的需求，而且得到的效果看上去和 iWave 算法的效果一样好。

其实，很多情况下，精确的时间步长是必要的。但是，在大部分自然环境特效中，却不是这么回事。因为，这些特效只是起一些装饰的作用。本文后面要讲的一些特效也不会为了适应精确的时间步长，而使用第 2 个缓冲。接下来的一个例子——青草的仿真模拟，将开始涉及网络分析方面的问题。

4.2.2　青草的模拟

青草的模拟与动态水面的模拟只有一些微小的不同。对于一大片草地，当“能量”作用于草秆上，这个草秆也会依次作用于其周围的草秆。需要说明的是，“能量”一词是比较通俗的用法。在这里，“能量”指的是影响一个物体，或者由物体所支配的可测量的值。本文后面所有的“能量”

一词的用法均是如此。如果施以足够的能量，如一阵狂风，那就会有一个能量波浪在草地上流过。

算法

如果用前面提到的网络方法分析水面特效，就可以筛选出可保存的元素。与水面模拟一样，青草的模拟也使用同样的平面网格来表示这些元素。每个节点代表一株青草，周围最多有4株相邻的其他青草。在这两个系统中，能量流动的方式是类似的，因此用在两个链接之间的函数并不需要做什么改动。两个系统之间惟一的区别是对节点属性的说明。在水面模拟中，每个节点表示的是高度值。而在青草的模拟中，情况要复杂得多。当能量作用于一株青草，青草的秆和叶子会在能量流动的方向上产生“旋转”（如被风吹弯了）。这个过程需要两样东西：能量作用的方向，以及能量的数量表达（能量标量），用来决定所用影响的程度。

对于每个节点，都可以使用这个值来表示能量。也就是说，这个值原来是表示波浪的高度，而现在就变成了青草旋转的角度。我们可以定义一个向量，代表能量的作用方向。这样可以控制风吹过来的方向，这个向量的缩放比例由能量标量来决定。然后这个向量沿着 x 和 z 轴分解成两个分量，每个轴向上的旋转与相应的分量是成比例的。将每个轴向的分量转换为旋转的角度有很多不同的方法，最简单的方法就是将这些分量按比例进行缩放，如图4.2.3所示。

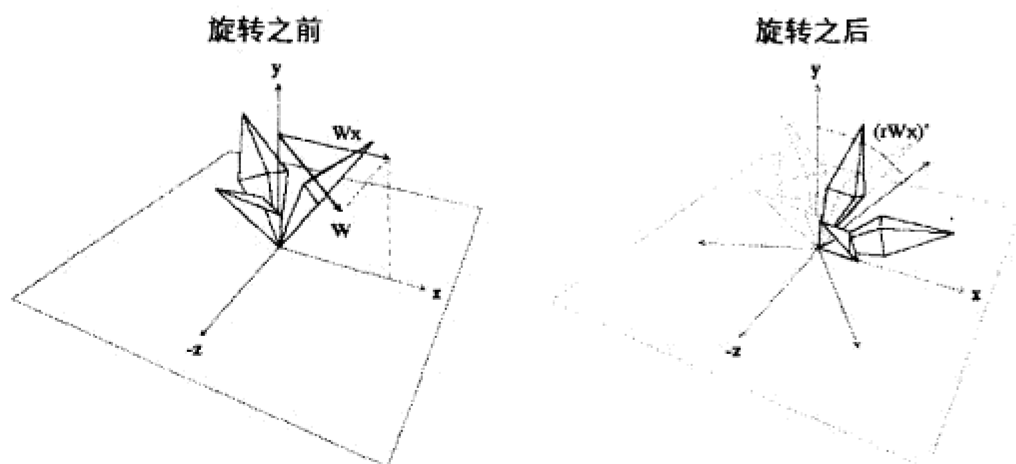


图4.2.3 青草的旋转或“倾斜”。向量 Wx 由向量 W 计算得出，然后经过 r 缩放得到（rWx）度的旋转角度。Z 轴上进行了同样的旋转操作

重复这个过程就可以模拟出一大片草地，同样也可以制作出“风吹草低”的幻景。到此为止，我们轻而易举地将水面模拟转换成了青草模拟。接下来，要看看变化传播模型的一些规则。

4.2.3 变化传播模型

在前面的讨论中，我们曾经重点探讨了一个变化传播模型必需的几个元素。该模型是由一个网络组成，网络中的每一个节点彼此相连。每个节点都用来保存某个数值。由于这些节点彼此连接，如果它们彼此作用，产生的变化是由几个函数来控制的。下面就分别考察组成模型的每个元素，以便搞清楚哪些是关键要素。

1. 网络

到目前为止，我们说的网络都是网格矩阵。在上面的两个例子中，网络的逻辑排列（如一个网格）直接映射到物理表达。但是，这些例子只是一个抽象概念的简单实现。我们可以

归纳出概念，而网络则负责定义两个节点之间的互连。网络并不受限于结构或者任何针对现实环境的物理映射（在下一个特效模拟中会有说明）。网络自身的结构也会随着时间而变化。

2. 节点/释义

在前面的例子中，节点都只由一个值组成。该值是个浮点标量，映射到某个物理元素上。每个网络都是由相同的节点集合而成。

一般情况下，我们可以把一个节点看成是可以变化的东西。它可以表示任何数量，从一个单一浮点变量，到一个任意类（arbitrary class）的链表。节点并非一定要映射到某个物理量上。它们可以简单地表示某些抽象的东西，比如一个神经网络的高度[Buckland01]。另外，节点也不必一定是它们邻居节点的复制品，只要它们满足链接和函数的需求就可以。后面会对此进行讨论。

3. 函数

这些函数用标量值来产生新的标量值。例如，iWave 算法利用网格的高度来生成新的高度。但是一般来说，在这个模型中，我们可以把函数看成一个关系（Relation），它将一个节点的输出结果映射到另外一个节点的输入上（或者映射回原始节点）。为了解决前面提到的不同节点的问题，我们可以写一个函数，将一个节点的输出转换为另一个节点的输入。而且，这些函数也不一定必须是双向的。

函数的关联函数和派生函数或多或少会受到使用它们的应用程序的影响。例如，iWave 算法的一个函数就是从描述流体力学的函数 *Linearized Bernoulli Equations*[GPG04]中派生得来的。虽然我们不能完全躲开最终应用程序的影响，但应该说明的是，函数不是必须用数学的方法来证明的。函数是随机的输入与相应输出结果之间的一个关系。近似值和任意的关联都是可以接受的，只要它们能够产生我们想要的效果。

4. 链接

前面的几个例子并没有涉及链接的概念。通常而言，链接是两个节点之间的函数集。在变化发生时，链接就像是两个节点之间的一座桥梁。一个链接可以使用多个函数。反过来，在模拟工作进行中，函数也可以在节点之间进行交换。如果网络的结构发生了变化，那就需要一个标准的输入/输出的数据类型。这个转换可以由插入一个链接中间的函数来完成。

4.2.4 树叶的模拟：模型的应用

在下面的练习中，我们将使用变化传播模型来模拟“树叶婆娑”的景象。在这个特效中，临风面上的树叶似乎应该反应得更为猛烈。随着能量流过这棵树，离能量源近的树叶要遮挡住那些离能量源更远的树叶。为了重现这个特效，我们先看看变化传播模型中的每个组成部分，并针对它们引发的问题拿出合理的解决方案。

1. 网络

网络负责定义两个节点之间的互连。在现实环境中，两个树叶之间的相互作用是相当小的。树叶彼此之间也没有什么有规则的联系。但是，通过风的吹动，树叶彼此之间还是互相

遮挡，这就意味着它们产生了变化。在这种情况下，网络的结构并不随着时间而变化。

理想状态下，每个叶子都会影响到其他所有的叶子。由于风可能来自任何一个方向，因此每个叶子都可能会影响到其他的叶子。而相关的函数就应该遵循叶子周围气流的轨迹。很明显，这样做会累死人的。通过观察，我们可以简化模拟工作。这个模拟工作的依据是能量会从网络中一个或几个点散发开来。我们可以这样认为，一个节点连接着与其最近的三个相邻节点，而两个节点之间传送的能量与它们之间的距离是成比例的。这个值是人为指定的，而实际的使用证明，在精确的全网格化模拟和单链接网格之间，它是一个相当好的折中方案。

2. 节点/释义

节点是一个可以变化的东西。在对树叶的模拟中，我们要保存某种标量，表示树叶在风向上被吹动的程度。每个树叶都必须清楚自己当前的方位以及风的全局方位。这样，树叶才能被吹向正确的方向。当然，我们也可以设计这样的一个系统：让树叶被吹向随机的方向。在这个例子里，我们将通过系统中能量流的路线来推断风的方位。

对能量标量的释义给我们提出了几个有趣的问题。我们有两个向量：一个定义了树叶的方位；另一个则定义了风的方位。能量标量显示了树叶对风向量的依赖程度。如果没有能量，树叶就会回复到自己原来的位置；如果有很多能量，树叶就会在风吹的方向上剧烈运动。

有很多数学方法可以解决这个问题。我们也许可以使用四元数和球状线性差补法[Wattenberg97]。但是，采用这个方法，如果试图为树叶的行为引入随机性，或者按照希望的方向去渲染树叶，都会碰到一些难题。我们还可以采用球状坐标算法[Bobick03]。每个树叶的初始方位以及风的方位由一个方位角和一个倾斜角指定。能量标量用来确定我们该在树叶当前的方位上沿着风的方向增加多少角度。我们还会加入很小的随机角度，来制造一些噪声。图 4.2.4 展示了计算树叶新方位的算法。

3. 函数

当风吹过一棵树时，距离风源最近的树叶受到的影响最大。风停下来后，能量对树叶的作用逐渐消失。受影响最大的树叶，其能量也是最后才消失。模糊（Blur）算法可以最好地重现这种类型的行为，本文的水面模拟部分已对此做过讨论。在典型的模糊算法中，点(i, j)的值是其周围所有点和其自身值总和的平均值。

虽然这个函数仿效出了我们追求的行为，但是一段时间之后，由于更多的能量不断地添加进来，系统的能量会一直增加下去。拿一个使用该函数的格子作为例子。如果在点(i, j)上使用一个能量的增加量，能量的传播经过时间 t 稳定下来，那么格子里所有的点都会占有部分的初始能量，进入这个系统的初始能量最后会被格子里的这些点平均分配。

我们需要的是一个能量最后消失为零的系统：回复平静，树静而风止。我们可以让能量的交换与节点之间的距离成比例，这样两个节点之间只有部分的能量在传递。随着节点之间距离的增加，能量的传递也在减少，然后我们将结果进行求和。一段时间之后，节点之间的距离会不断地减少系统中的能量，直到最后系统恢复平静。

4. 链接

在这个特定的模拟中，链接扮演的角色是非常小的。当要考虑动态结构的网络时，使用

链接的分析才会变得更有意义。

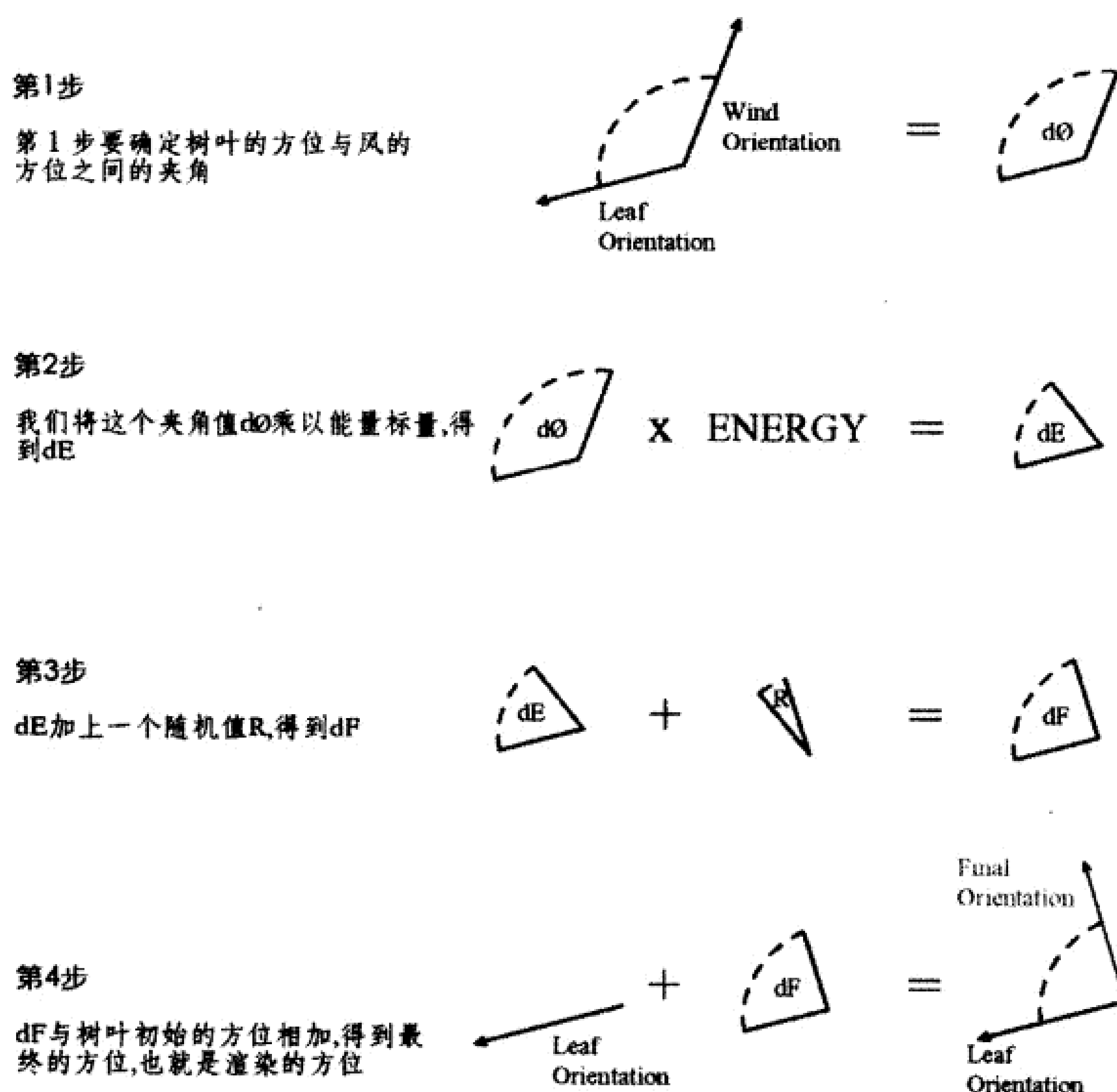


图 4.2.4　计算树叶新方位的算法

4.2.5　总结

本文考虑了三种模拟特效：水面、青草和树叶。每个模拟都使用了同样的基本思想：变化传播模型。我们的目的是让读者自己思考这些例子，并结合使用变化传播模型，创作出新的特效。

4.2.6　参考文献

[Bobick03] Bobick, Nick. "Rotating Objects Using Quaternions." Available online at *http://www.gamasutra.com/features/19980703/quaternions_01.htm*. June 3, 2003.

[Buckland01] Buckland, Mat. "Neural Networks in Plain English." Available online at *http://www.ai-junkie.com/ann/evolved/nnt1.html*. June 12, 2001.

[GPG04] Tessendorf, Jerry. *Game Programming Gems 4*. Charles River Media, 2004.

[Wattenberg97] Wattenberg, Frank. "Spherical Coordinates." Available online at *http://www.math.montana. edu/frankw/ccp/multiworld/multipleIVP/spherical/body.htm*. May 21, 1997.

[Willemse00] Willemse, Roy. "The Water Effect Explained." Available online at *http://www.gamedev.net/reference/articles/article915.asp*. February 15, 2000.

4.3 使用质点-弹簧模型获得真实的布料动画

塞维利亚大学，Juan M. Cordero
cordero@lsi.us.es

虽然布料是很普通的物品，但是管理布料力学的物理法则却非常复杂。模拟布料动画的算法在不断地改进，但都集中在两个主要的方面：模型的效率和探索更真实的效果。

从 20 世纪 80 年代中期开始，业界就提出了几个分别基于几何学[Weil86]、物理学[Provot95，House00]和混合特性[Tsopelas91]的布料模拟模型。到目前为止，只有那些基于物理学的模型可以得到比较真实的效果。但是，由于计算的复杂度，以及难以为特定的布料特性生成界面友好的函数，物理模型仍然是非常耗时的过程。

最近几年，业界将 Kawabata 评估系统（Kawabata Evaluation System，简称 KES）引入到了布料仿真模型中，以根据经验来获得某种类型的布料所固有的参数。但是，Kawabata 评估系统是一个代价高昂的技术。而且相对来说，在大多数情况下，采用这个方法获得的结果很少应用在仿真模型中。

本文详细介绍了一个新的基于质点-弹簧模型的仿真算法。该算法可以在很低的计算成本下，获得高质量的仿真效果。下一节描述了一个基于质点-弹簧系统的布料计算模型。在此之后，会介绍布料力学中几个相关的作用力。然后，会概要性地介绍布料力学系统的几个等式，并提供一个解决的方法。最后会做一个总结，明确未来的研究方向。

4.3.1 布料的离散表示

一块长方形的布料可以由一个 $n \times m$ 的质点粒子网格表示。组成网格的质点粒子之间由弹簧连接，如图 4.3.1 所示。

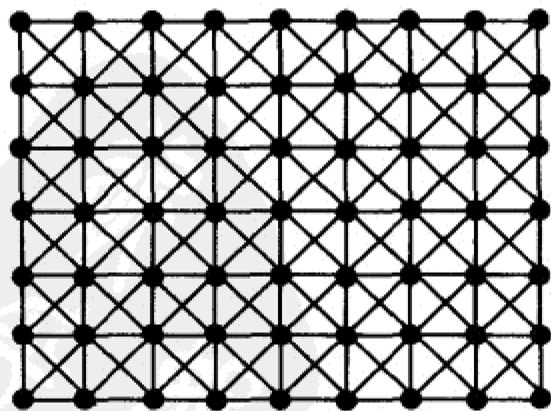


图 4.3.1 使用网格的布料离散表示

每个粒子的质量代表的是这个粒子周围材料的质量。连接粒子的弹簧负责产生粒子之间的作用力，这个作用力使粒子之间保持一定的距离。网格是有方向的，这样，对于其中的每一个粒子，都可以定义一个垂直于表面的法线向量。描述网格方向的法线向量有如下的表达式，见公式 4.3.1

$$\vec{n}_{ij}=\frac{\vec{N}_1+\vec{N}_2+\vec{N}_3+\vec{N}_4}{\|\ \vec{N}_1+\vec{N}_2+\vec{N}_3+\vec{N}_4\ \|} \quad (4.3.1)$$

其中，$\vec{N}_k$ 是定义粒子四周那些网格的每一个三角形的法线向量，如图 4.3.2 所示。

正如要向大家展示的那样，知道每个粒子的表面法线是非常重要的，这样才能判断某个作用力是来自网格平面外部（外力），还是内部（内力）。对于网格中的每一个粒子，都有 3 种相邻的邻居粒子：拉伸邻居粒子（结构粒子）、剪切邻居粒子和弯曲邻居粒子。一旦这些邻居粒子都定义好了，就可以描述网格的内力了。从一个位置在(i, j)的网格粒子入手，对于有 4 个邻居粒子的情况，可以用方程式 4.3.2 描述拉伸邻居粒子。

$$V_T=\{P_{i+1,j},P_{i-1,j},P_{i,j+1},P_{i,j-1}\} \quad (4.3.2)$$

如果粒子位于网格的边缘，那它就只有 3 种邻居粒子。如果位于网格的角上，那么该粒子就只有 2 个邻居粒子（见图 4.3.3）。

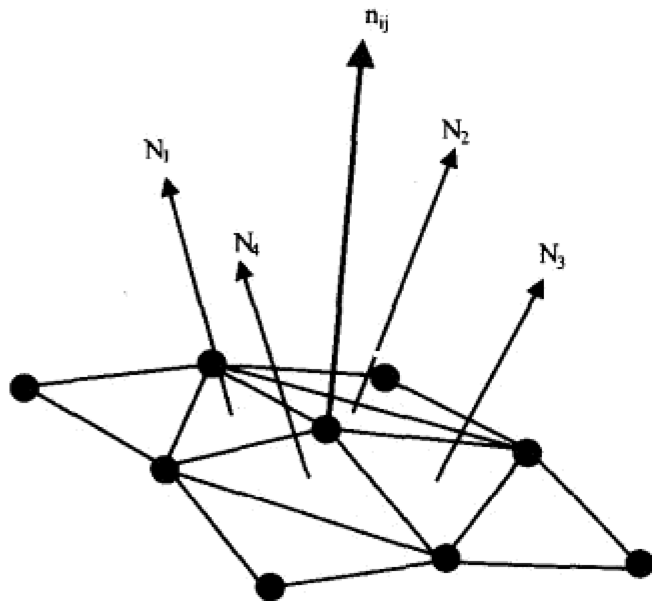


图 4.3.2　在粒子位置上的网格的法线向量是该粒子周围所有三角形法线向量的组合

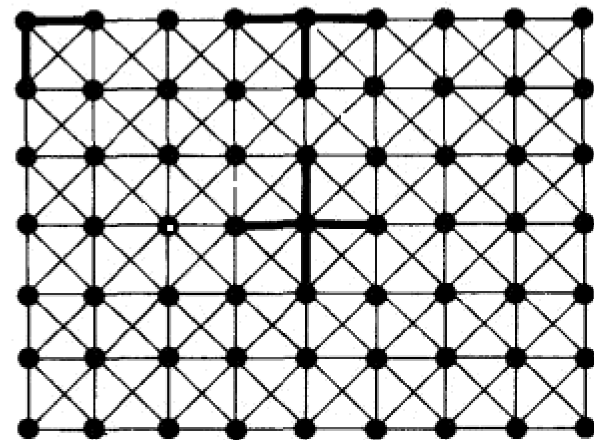


图 4.3.3　拉伸邻居粒子的示意图

根据网格的主要方向，拉伸邻居粒子集 V_T 可以分成 2 个子集：

$$V_{T^{wp}}=\{P_{i+1,j},P_{i-1,j}\}$$

$$V_{T^{wf}}=\{P_{i,j+1},P_{i,j-1}\}$$

其中，V_T^{wp} 是经向上的拉伸邻居粒子，而 V_T^{wf} 是纬向上的拉伸邻居粒子。

剪切邻居粒子可以用公式 4.3.3 表示（见图 4.3.4）。

$$V_S=\{P_{i+1,j+1},P_{i-1,j+1},P_{i+1,j-1},P_{i-1,j-1}\} \quad (4.3.3)$$

注意，根据粒子在网格中的具体位置，它可能有 1 个、2 个或 3 个邻居粒子。

我们会在后面文章中向读者说明，剪切邻居粒子不需要进行切分。

弯曲邻居粒子集可以用下面的公式表示：

$$V_B=\{P_{i+2,j},P_{i-2,j},P_{i,j+2},P_{i,j-2}\} \quad (4.3.4)$$

和拉伸邻居粒子一样，根据粒子在网格中的位置，弯曲邻居粒子可能会包含 4 个、3 个或 2

个邻居粒子（见图 4.3.5）。

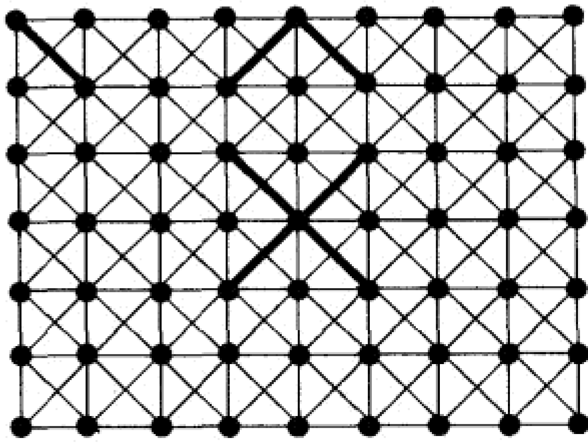


图 4.3.4 剪切邻居粒子的示意图

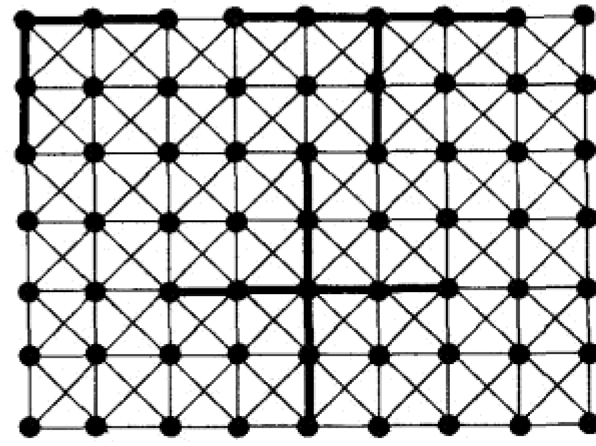


图 4.3.5 弯曲邻居粒子的示意图

弯曲邻居粒子集也可以分成 2 个子集：

$$V_{B^{wp}} = \{P_{i+2,j}, P_{i-2,j}\}$$
$$V_{B^{wf}} = \{P_{i,j+2}, P_{i,j-2}\}$$

上面这 2 个公式分别代表了网格经向和纬向上的粒子。

为了表述上的方便，我们用 R 来指定公式 4.3.2、公式 4.3.3 和公式 4.3.4，以及各自子集的下标索引：

$$R_T = \{(i+1,j),(i-1,j),(i,j+1),(i,j-1)\}$$
$$R_{T^{wp}} = \{(i+1,j),(i-1,j)\}$$
$$R_{T^{wf}} = \{(i,j+1),(i,j-1)\}$$
$$R_S = \{(i+1,j+1),(i-1,j+1),(i+1,j-1),(i-1,j-1)\}$$
$$R_B = \{(i+2,j),(i-2,j),(i,j+2),(i,j-2)\}$$
$$R_{B^{wp}} = \{(i+2,j),(i-2,j)\}$$
$$R_{B^{wf}} = \{(i,j+2),(i,j-2)\}$$

4.3.2 作用力

一旦得到了布料的网格，并且定义了粒子及其邻居粒子集的方位，我们就可以表达与每个粒子相关的作用力。我们要确定 2 组不同的作用力：内力和外力。

1. 内力

内力与布料的准弹性行为特性有着密切的关系。这里总共有 3 种类型的作用力：拉伸力、剪切力和弯曲力。其中，拉伸力和剪切力是平面内力（In-plane force），而弯曲力是面外力（out-of-plane force）。这就是说，拉伸力和剪切力的作用方向是沿着网格定义的方向，而弯曲力则是垂直于网格表面的。因此，为了定义这些作用力，必须首先知道网格中每一个粒子的方位。

在这 3 种作用力中，拉伸力占据主导地位。由于拉伸力与日常生活中布料的伸缩特性颇

为相似，所以在仿真的时候，我们不会因为布料的类型不同而对这个力区别对待。因此，很多作者都不会考虑拉伸力的影响，相反，它们会对网格强加一个基本的假设：拉伸邻居粒子之间的距离是恒定不变的[Witkin90]。而这篇论文则要考虑拉伸力及其影响。对于一个网格粒子 P_{ij} 的拉伸力，虎克定律（*Hook's Law*）给出了下列公式：

$$F_{ij^T} = -\sum_{(k,l)\in R_{T^{wp}}} K_{T^{wp}}(\xi)\vec{1}_{ijkl} - \sum_{(k,l)\in R_{T^{wf}}} K_{T^{wf}}(\xi)\vec{1}_{ijkl} \quad (4.3.5)$$

其中：

$$\vec{1}_{ijkl} = \vec{P}_{ij} P_{kl}$$

$$\xi = \frac{\left\|\vec{1}_{ijkl}\right\| - 1_{ijkl^0}}{1_{ijkl^0}}$$

在这里，1_{ijkl^0} 是网格在静止状态下，粒子 P_{ij^0} 和 P_{kl^0} 之间的弹簧长度。

也就是：

$$1_{ijkl^0} = \left\|P_{ij^0} P_{kl^0}\right\|$$

其中，$K_T^{wp}(\xi)$和 $K_T^{wf}(\xi)$取决于拉伸率ξ。$K_T^{wp}(\xi)$和 $K_T^{wf}(\xi)$一起时候使用就可以模拟布料的准弹性行为特性。

另外一个平面内力是剪切力。这个作用力表示的是布料横向的变形。作用于粒子 P_{ij} 上的剪切力可以用公式 4.3.6 表示：

$$F_{ij^S} = -\sum_{(k,l)\in R_S} K_S(\xi)\varphi_{ijkl}\vec{1}_{ijkl} \quad (4.3.6)$$

其中：

$$\vec{1}_{ijkl} = \vec{P}_{ij} P_{kl}$$

$$\xi = \frac{\left\|\vec{1}_{ijkl}\right\| - 1_{ijkl^0}}{1_{ijkl^0}}$$

$$1_{ijkl^0} = \left\|P_{ij^0} P_{kl^0}\right\|$$

$$\varphi_{ijkl} = 1 - \left|\frac{\vec{1}_{ijkl}}{\left\|\vec{1}_{ijkl}\right\|}\right| \vec{n}_{ij}$$

n_{ij} 是粒子 P_{ij} 的法线向量，这和前面看到的一样。

对于参数 φ_{ijkl}，可以有如下的解释：剪切力是一个平面内的力，当 φ_{ijkl} 等于 1 时，它会达到最大值。当粒子 P_{ij} 的法线与向量 $\vec{l}_{ijkl}$ 互相垂直时，就会发生上述情况。如果上述两个向量不是正交向量（二者互不垂直），那么随着作用力逐渐从剪切力变成弯曲力，φ_{ijkl} 的值也会随之减小，如图 4.3.6 所示。

该网格没有绝对坐标系统，所以我们无法表示那些基于布料的经线和纬线的剪切力。因此，当发生横向变形时，我们就认为这两个方向上受力的影响是一样的。所以，我们可以不再使用 K_S^{wp} 和 K_S^{wf}，而是使用它们的一个组合 K_S。

弯曲力是惟一一个平面外力，可以用如下公式表示：

$$F_{ij^B} = -\sum_{(k,l)\in R_{B^{wp}}} K_{B^{wp}}(\xi)\psi_{ijkl} - \vec{1}_{ijkl} - \sum_{(k,l)\in R_{B^{wf}}} K_{B^{wf}}(\xi)\psi_{ijkl} - \vec{1}_{ijkl} \qquad (4.3.7)$$

其中：

$$\vec{1}_{ijkl} = \vec{P}_{ij} P_{kl}$$

$$\xi = \frac{\left\|\vec{1}_{ijkl}\right\| - 1_{ijkl^0}}{1_{ijkl^0}}$$

$$1_{ijkl^0} = \left\|P_{ij^0} P_{kl^0}\right\|$$

$$\varphi_{ijkl} = \left|\frac{\vec{1}_{ijkl}}{\left\|\vec{1}_{ijkl}\right\|}\right| \vec{n}_{ij}$$

与参数 φ_{ijkl} 一样，参数 φ_{ijkl} 代表平面外部的作用力。当 $\varphi_{ijkl}=0$ 时，法线 $\vec{n}_{ij}$ 和向量 $\vec{l}_{ijkl}$ 互相垂直，弯曲力就可以作用于平面上。但是，如果 $\varphi_{ijkl}=1$，法线 $\vec{n}_{ij}$ 和向量 $\vec{l}_{ijkl}$ 是平行的，弯曲力就作用于平面之外，如图 4.3.7 所示。

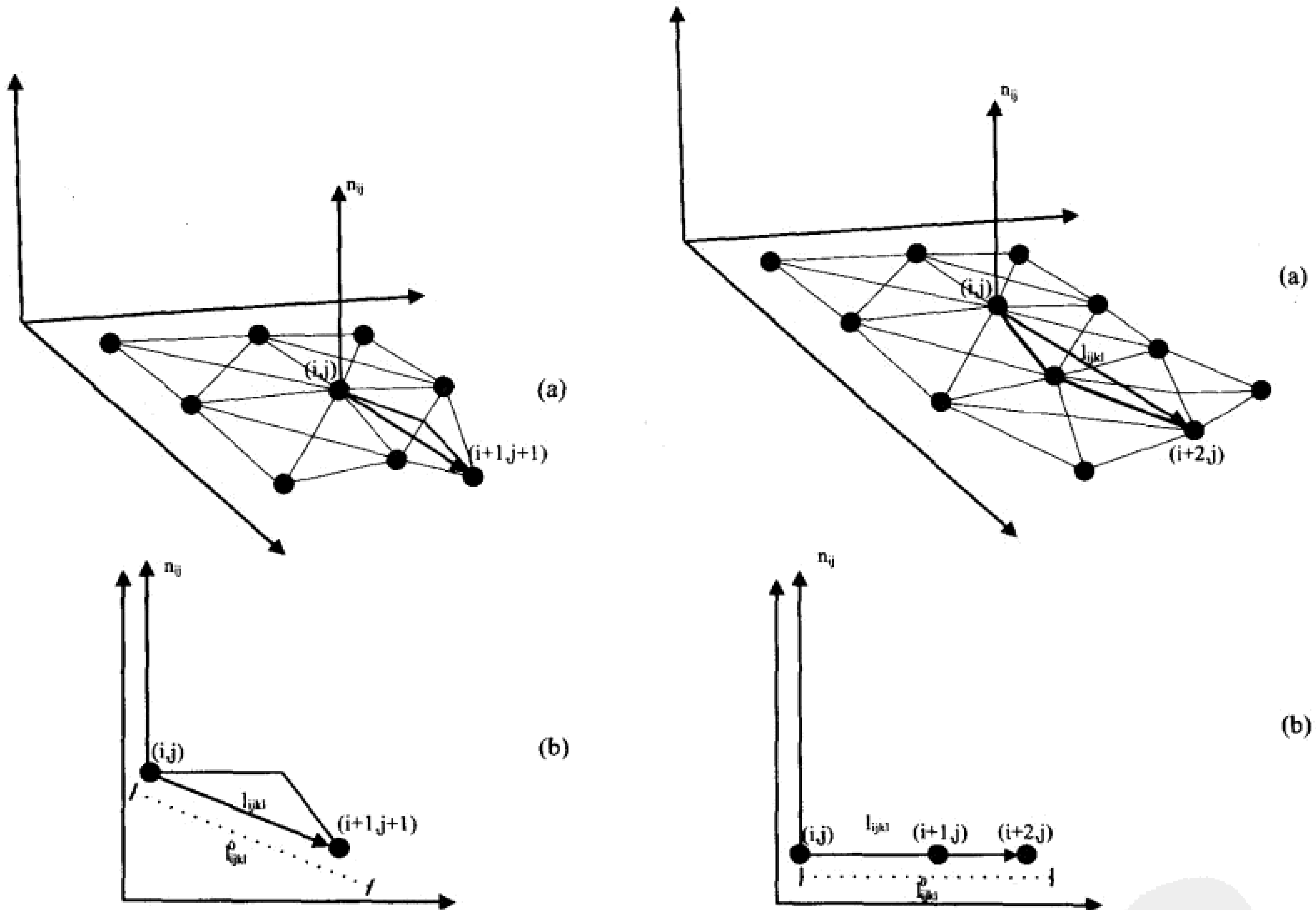


图 4.3.6 一个剪切弹簧的经度已经发生了变化（a）。弹簧的拉伸方向垂直于网格平面，所以这里没有剪切力

图 4.3.7 一个剪切弹簧的纬度发生了变化（a）。弹簧的拉伸方向垂直于网格平面，所以这里没有变曲力

2. 拉伸函数

前面介绍过的函数 $K(\xi)$，可以根据拉伸率来决定力的变化。如果把这些函数考虑成线性的，那么弹簧的行为也应该是线性的。这就意味着，拉伸力、横向变形或者布料的弯曲，

都符合虎克定律。但是，布料的准弹性行为却是非线性的。当拉伸率超过某个特定的限度，那相关运动的阻力会出现指数级的增加（参见 Kawabata 评估系统的有关图示）。函数 $K(\xi)$ 将布料的弹性行为引入了作用力的表达式。因此，为了在动画中获得一致的结果，$K(\xi)$必须至少满足两个条件：

$$K(0)=0$$

$K(\xi)$是单调非递减函数

3. 外力

不是由布料的准弹性行为产生的力，被称为外力。这些力包括：重力、空气阻力等。在我们提出的模型中，每个粒子的质量与它所控制区域的质量是相对应的。这样，重力就会作用于所有的粒子。对于重力，我们使用常用的公式：

$$F_{ij}^{G} = m_{ij}\vec{g}$$

其中，m_{ij} 是粒子 P_{ij} 的质量，$\vec{g}$ 是重力加速度。空气对布料的影响取决于空气的速度和方向，以及布料本身的速度。空气阻力大致估算如下：

$$F_{ij}^{A} = C_a\left[\vec{n}_{ij}\cdot\left(\vec{u}_a - \vec{v}_{ij}\right)\right]\cdot\vec{n}_{ij} \quad (4.3.8)$$

其中，C_a 是空气粘性常量，$\vec{u}_a$ 是一个向量，定义了空气的速度和方向。$\vec{v}_{ij}$ 是粒子 P_{ij} 的速度，$\vec{n}_{ij}$ 是粒子 P_{ij} 的法线向量。当空气速度向量 $\vec{u}_{ij}$ 等于 0 时，空气保持稳定，之前的公式可以看成是空气阻力。还有很多其他作用于布料的力，我们把它们都归类为外力。例如，翻转布料时产生的力；揪着布料的一个角，把布料悬挂起来产生的力。

4.3.3 动态系统方法

得到了作用于网格粒子的力之后，就可以用公式动态地表达这个系统。为了实现这个目标，粒子必须遵守牛顿第二定律，见公式 4.3.9。

$$\sum F_{ij} = m_{ij}a_{ij} \quad (4.3.9)$$

其中，F_{ij} 表示的是作用于每个粒子 P_{ij} 上的力，m_{ij} 是粒子的质量，$a_{ij} = \ddot{P}_{ij}$ 是加速度。根据公式 4.3.9，对于网格的每一个粒子，我们可以得到如下的微分方程式：

$$\begin{cases} \ddot{P}_{ijx} = \dfrac{1}{m_{ij}}F_{ijx} \\ \ddot{P}_{ijy} = \dfrac{1}{m_{ij}}F_{ijy} & \begin{array}{l} i=1,\ldots,n \\ j=1,\ldots,m \end{array} \\ \ddot{P}_{ijz} = \dfrac{1}{m_{ij}}F_{ijz} \end{cases} \quad (4.3.10)$$

用来求解微分系统的计算方法同样可以用来解决布料的动态问题[Volino00]。对于这个特殊的情况，由于使用了拉伸函数而不是拉伸常量，我们推荐大家使用时间步长的方法来求解前面的方程式。大多数情况下，作用力不会增加太多，但是为了抵抗剧烈的拉伸作用，这些力就会变强。使用 *Runge-Kutta-Fehlberg* 方法的第四、第五规则，可以获得很好的仿真效果。

4.3.4 仿真模拟

介绍了模型和描述网格动态行为的公式之后，就可以大致描述出布料仿真动画的方法（见图 4.3.8）。

首先，把一个布料实体建模成一个关联的网格。我们需要选择一个相匹配的网格，网格中的粒子数量不多也不少，以便让网格的表面尽量平滑。

其次，计算网格表面上每一个粒子的法线向量。然后，就可以获得作用于每个粒子的内力和外力，并进而得出前面所说的微分方程式。

最后，采用递增的时间间隔来求解这个系统，从而获得网格的一个新位置。重复这些步骤，就得到了布料仿真的实时动画。

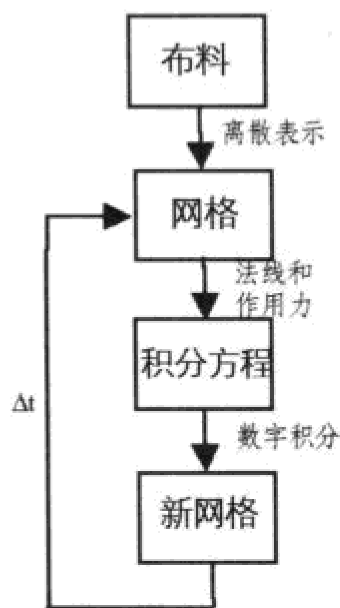


图 4.3.8 本文提出的仿真方法的流程图解

4.3.5 结论

本文提出了一个真实布料仿真动画的模型，这个模型的优势在于其易于实现。网格表示法是基于一个大家比较熟悉的质点-弹簧系统。这个方法提供了一个适合计算机游戏的高效的计算系统。

我们把布料的行为看成是准弹性的，而不是完美弹性的。这样，就提高了仿真效果的真实感，避免了一些不想要的效果，例如超弹性效果[Cordero01]。

另外，对平面内力和平面外力的区别对待，可以产生更精确的仿真效果，避免了在只有剪切力存在的情况下出现弯曲力。

图 4.3.9 和图 4.3.10（见彩色插图 1A 和 1B）是采用这个模型产生的动画效果截图。

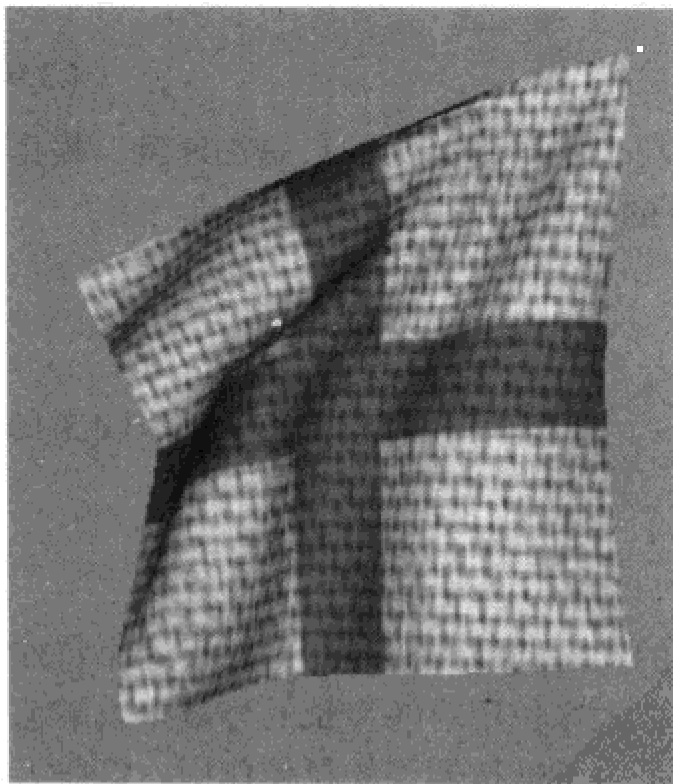


图 4.3.9 飘扬的旗帜

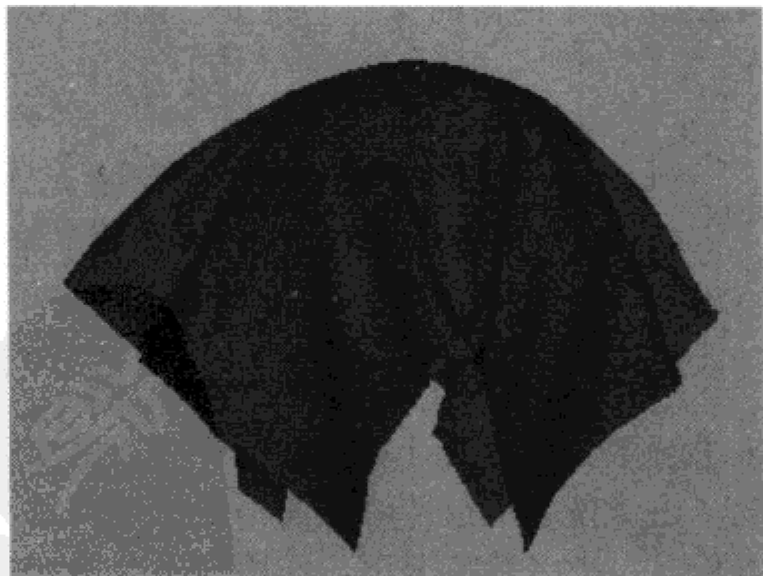


图 4.3.10 覆盖在球体上的布料

对于这个模型，还有进一步的工作。我们应该努力去发现 $K(\xi)$的更多功能，获得更真实的布料仿真。也就是说，要找到最佳的 $K(\xi)$函数及其参数的值，将布料的真实运动与布料仿真之间的差别最小化。另外，这个方法使用的是一个矩形布料的网格，未来模型的发展方向应该是可以独立于布料本身的形式，表示任何的布料。

4.3.6 参考文献

[Cordero01] Cordero, J.M., J. Matellanes,, and J. Cortés. *Corrección del Efecto Superelástico en Dinámica de Telas*. XI Congreso Español de Informática Gráfica (CEIG'2001), Girona, pp:141–147. 2001.

[House00] House, D.H. and D. E. Breen. *Cloth Modeling and Animation*. A K Peters, Ltd, 2000.

[Kawabata75] Kawabata, S. *The Standardization and Analysis of Hand Evaluation*. Committe of the Textile Machinery Society of Japan, 1975.

[Provot95] Provot, X. *Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior*. Graphics Interface '95 Proceedings, 147–154. Quebec, Canada, 1995.

[Tsopelas91] Tsopelas, N. *Animating the Crupling Behavior of Garments*,11–24. Proc. 2nd Eurographics Workshop on Animation and Simulation. Blackwell, UK, 1991.

[Volino00] Volino, P., N. Magnenat-Thalmann. *Virtual Clothing. Theory and Practice*. Springer-Verlag, 2000.

[Weil86] Weil, J. *The Synthesis of Cloth Objects*. Computer Graphics (Proc. Siggraph), 20:49–54, 1986.

[Witkin90] Witkin, A., M. Gleicher, and W. Welch. *Interactive Dynamic*. Computer Graphics (Proc. Symposium on 3D Interactive Graphics), 24(2):11–21. 1990.

PDG

4.4 适合游戏开发的实用柔体动画技术：受压柔体模型

波兰弗罗茨瓦夫大学，Maciej Matyka
maq@panoramix.ift.uni.wroc.pl

在计算机图形图像研究领域中，我们已经获得了很多柔体仿真的方法。通常来讲，这些方法可以分为两大主要模型：基于几何的几何学模型和基于物理的物理学模型。由于缺少足够的控制，快速、简单的几何学模型（即通常所说的“自由变形技术”，*Free Form Deformation*，简称 FFD[Sedenberg86]）在游戏开发中不太行得通。为了获得更真实的计算机动画，业界又提出了其他几种方法，特别是基于物理学的方法。

基于物理学的弹性理论应用为柔体运动难题提供了一个复杂的解决方案。这个解决方案在数据计算上是非常繁杂难懂的。通过使用有限体积法（Finite Volume Method）[Irving04]，我们可以非常精确地对粘弹性体（viscoelastic body）进行仿真。但不幸的是，使用这些工程学方法，我们得到的只是非实时的动画，因此无法满足游戏应用的需求。即使是最经典的计算流体力学（Computational Fluid Dynamics，简称 CFD）也在这里找到了自己的位置。在[Nixon02]一文中，作者构建了一个由网格包围的可压缩模型，并在其中引入了另外一些附加的作用力。由于该模型与受压柔体模型非常类似，我们会在后面对它进行更深入的讨论。请大家注意，本文并不考虑 CFD 方法。由于求解纳维叶-斯托克斯（Navier-Stokes）方程的计算工作非常复杂，因此 CFD 方法（计算流体力学）并不是一个实时的解决方案。

我们并不打算深入讨论非实时的柔体模型。之所以提到这些模型，是因为有一点很重要，就是大家要知道，在现在的 CG 研究领域中，确实有很多更为精确的、具备正确物理特性的模型，这些模型可以作为未来游戏开发应用的基础。

对于实时柔体仿真应用，业界已经提出了几个不同的方法。但没有一个看上去特别适合游戏开发。由于实现起来很简单，计算速度也非常快，很多方法都引用了简单的质点-弹簧（SM）模型（参见[Lander03]）。但是，质点-弹簧模型通常无法生成比较好的真实效果。为了提高逼真程度而增加弹簧的数量，并无法自动生成一个理想的解决方案。这个方法有它自身的问题。如果大量地使用弹簧，通常会造成仿真的“硬性”问题。还有就是，如果使用质点-弹簧模型，动画师就不得不面对很多物理上的约束，包括仿真模型本身变化的不确定性（可以想象一个有 1 500 个弹簧的物体，如果

改变其中一个弹簧的属性，动画师根本不可能预测其可能引发的影响）。

4.4.1 简化的质点–弹簧模型

由于质点–弹簧模型在可变形体仿真方面存在的问题，业界提出了一些改造的方法，以避免增加额外的弹簧连接。

考虑图 4.4.1（a）中的几何物体。在一个典型的质点-弹簧仿真中，它的构造几乎没什么用处。如果在一个有重力表现的仿真环境中，这个物体马上就会分崩离析。我们需要某种内力系统，使物体保持稳定。最简单、最直观的方法，是在物体各个顶点之间增加支撑弹簧。

用这个方法，可以构造出一个新的模型，如图 4.4.1（b）所示。这样的一个模型应该可以作为可变形体，在一个典型的 SM 仿真中合理地良好运转。要注意点和点之间的连接。这就意味着，如果有 N 个点，就需要进行 N^2 次弹簧计算。很多物体都有好几千个顶点，所以这样的计算工作实在是太繁重了。而实际上，我们可能也快把实时 SM 仿真一起给毁了。随着弹簧数量的增加，同时也增加了刚性维护工作的难度，和无用的内部运动的管理难度。有两个方法可以使“硬性”问题有所缓解。一种方法是采用计算运动方程所使用的隐式积分（implicit integration）[Barraf98]；另一种方法是在显式积分（explicit integration）方法中，使用反动力学约束作为修正措施[Lander03，Provot95]。这两个方法对问题的解决都有一定程度的帮助，但它们也带来了处理速度上的额外开销。

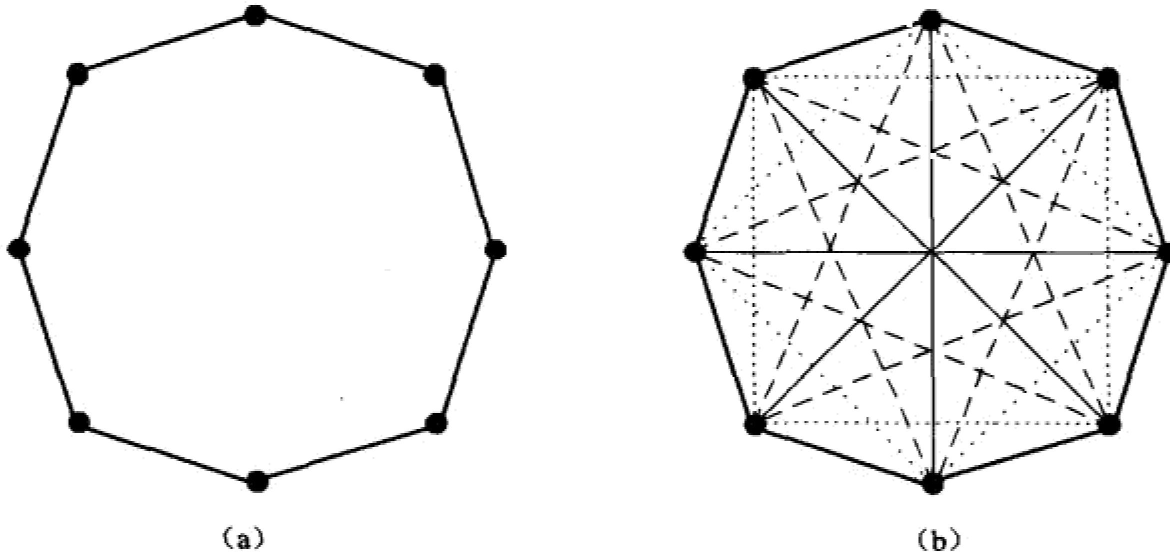


图 4.4.1 典型的质点-弹簧模型，图（a）是一个简单的壳体，没有内部支撑；图（b）是一个有内部连接支撑的壳体

在计算机图形图像研究领域，我们已经引入了简化了的质点-弹簧构造模型（参见[Katyka03，Nixon02，Meseure00]），成功地避免了这些问题。Nixon 以及其他几个人[Nixon02]创建了一个三维网格，其中填满了可压缩的流体，然后他们在这个物体中求解纳维叶-斯托克斯（Navier-Stokes）方程。对于基于物理学的模型，这个方法看上去似乎是一个精确的、非常有趣的新想法。但不幸的是，这个解决方案完全不适合高效的实时仿真。其主要原因就是该方案在求解纳维叶-斯托克斯方程时复杂度比较高，而且还需要构造一个随时间变化的网格。

在医学影像和虚拟现实领域，人们同样也在研究实时可变形体，并得到一些有趣的结果。Meseure 等人[Meseure00]断言，仅由弹簧组成的模型不适合外科医学应用，因为顶点数太多的物体，其几何构造比较复杂。他们也构造了一个质点-弹簧网格，与 Nixon 等人采用的方法

类似，但是并没有填充流体。相反，他们在物体中填满了“虚拟刚性元素”。之所以把刚性元素称为“虚拟的”，是因为它不与仿真环境发生交互。

Meseure 等人引入的虚拟刚性元素的想法看上去对游戏开发颇有好处。但是，它的一大弱点就是在简单模型之上又增加了新的复杂度。例如，这个模型的物理学背景多少有点深奥、难懂。当然了，在以结果为导向的游戏开发工作中，我们感兴趣的只是如何获得一个不错的可变形体的仿真。但是，当我们要处理基于物理学的动画模型时，从道理上讲，还是应该了解一下其背后的物理学知识。

4.4.2 PSB 模型背后的物理学

在受压柔体（Pressurized Soft Body，简称 PSB）模型中，我们考虑的是一个由网格构成的几何物体。这个网格是由节点（质点）和弹簧连接（虎克线性弹簧）组成的，如图 4.4.1 所示。我们假设物体的形状是闭合的，也就是说物体上不存在孔洞（不连续性）。由于和类似布料的物体的质点-弹簧模型的构造方法类似[Barraf98，Provot95]，我们通常把它看做是一个类似于用布料缝制成的物体。

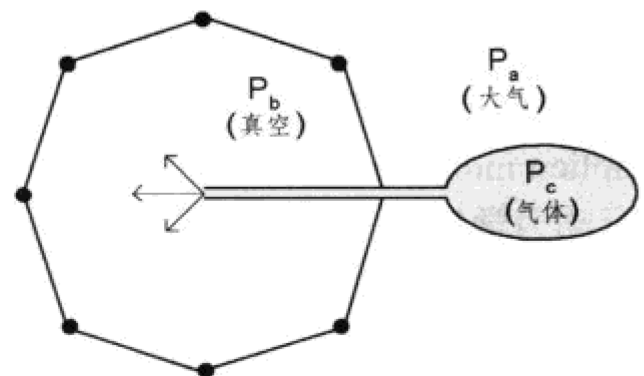


图 4.4.2 初始气压 $P_b=0$，一根管子插入到物体中。管子另一端连接着储气罐，罐中的气压是 P_c。仿真环境中的大气压力是 P_a

为了获得这个“用布料缝制的”物体的可变形体物理行为，我们引入了 PSB 模型。其基本思想如图 4.4.2 所示。大家会注意到，其中有一个小管插入到物体中。

这个管子连接到一个储气罐上。在储气罐中，存在着压力 $P_c>0$ 的气体。由于物体内部压力（P_b=0）与储气罐中的压力（$P_c>0$）之间的压力差，气体会从储气罐中流入物体内部，直到 $P_b=P_c$。然后，我们将管子拿走。这样的话，仿真模型就和图 4.4.1 中的模型有了一些微小的差别。如果模型内部的气体压力大于大气压（$P_b>P_a$），它就会发生变形（膨胀）。

理想气体的近似计算

由于仿真物体的宏观大小以及气体中粒子的微观大小，我们可以忽略气体粒子之间的相互作用，对非粒子碰撞的理想气体进行一个近似的计算[Callen85]。这就是说，可以使用大家非常熟悉的理想气体定律（*ideal gas law*），参见公式 4.4.1。

$$PV = nRT \tag{4.4.1}$$

其中：

- $P[N/m^2]$是气体的压力；
- $V[m^3]$气体的体积；
- n 是气体的摩尔数；
- R 是气体常数；
- T 是气体的温度。

在这个表达式中，假设 n、R 和 T 是常量，不会在仿真过程中有什么变化。同时还假设，我们知道该如何计算物体的体积。这样一来，就得到一个简单的表达式，用来计算物体内部的

气压。这个气压只会随着物体体积的变化而变化，参见公式 4.4.2。

$$P = \frac{nRT}{V} \tag{4.4.2}$$

在公式 4.4.2 等号的右边，有三个常量（分别是 n、R、T），以及一个变量 V。物体体积的计算方法会在后面讨论。

根据理想大气定律可以直接计算出一个气压，利用这个气压值就可以计算作用在网格节点的力。这是一个很直接、也相当简单的过程，但却可以帮助我们了解其背后的物理学和数学。

由于压力 $P[N/m^2]$的物理量纲，我们只能通过公式 4.4.2 中计算出来的压力值来测定作用在节点上的力。一般说来，压力必然伴有一个力的量纲，作用于一个单一场（unitary field）。

我们可以考虑该三维物体的一个三角面，如图 4.4.3 所示。这样，对于作用在网格节点上的力，就可以写出一个简单的表达式。

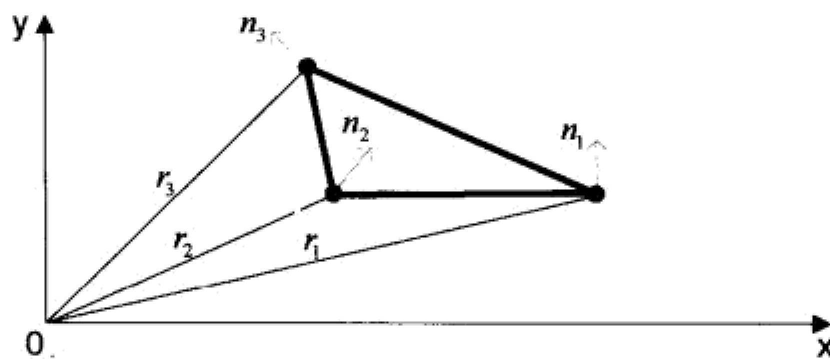


图 4.4.3 一个由三个网格节点和三个弹簧连接组成的三角面。其中，浅灰色的向量表示的是质点的法线向量

利用图 4.4.3 中给出的符号，可以把作用在三角面上的力用公式 4.4.3 表示出来：

$$\vec{F}^P = P \cdot \vec{n}_i \cdot A \tag{4.4.3}$$

其中：

- P 是利用公式 4.4.2 计算出来的压力值；
- A 是该面的一个场（field）；
- $\vec{n}_i$ 是第 i 个节点的法线向量。

在这里，有两件事情需要说明一下。第一，我们必须确定该如何计算节点的法线向量。这个工作很简单，只需要把该节点所在的所有平面的法线向量相加即可。

为了计算这个面的场，我们只需要使用向量的叉乘“×”运算。利用图 4.4.3 中提供的符号，该图中三角面的场可以用公式 4.4.4 表示。

$$A = \left|(\vec{r}_1 - \vec{r}_2) \times (\vec{r}_1 - \vec{r}_3)\right| \tag{4.4.4}$$

公式 4.4.4 表示的是向量的长度，将位于三角面的两个边上的向量进行叉乘“×”运算，就可以得到这个长度。

4.4.3 PSB 模型的实现

[Matyka03]一文中引入了“基于压力的模型（*pressure based mode*）”，也提出了实现这个模型的基本原理。我们会在后面的内容中，向大家简单介绍这个算法，全面描述该仿真程序的工作流程，以此来帮助大家了解 PSB 模型。更详细的信息，建议读者朋友们参考[Matyka03]。

PDG

但是我们相信，通过下面的描述，大家绝对可以从零开始，自己动手编写程序。

首先总结一下，看看我们进展到什么程度了。我们定义了一个可变形体的物理模型，其中充满了理想气体。前面几节内容中简单地描述了一下相关的数学模型。但是，我们并没有解释该如何使用这些公式，以及这些公式在整个规划中适用于什么地方。

为了能够更好地理解这个模型，读者朋友们需要对质点-弹簧仿真的思想体系有所了解。我们觉得不需要在这里对它进行详细的介绍，因为有大量的研究文献可以供大家参考（参见[Lander03，Provot95，Barraf98]）。但是，我们会在后面向大家概要性地介绍一下在柔体应用中使用的一些技术。

首先从一个封闭的三维网格开始。这个网格由若干个三角面组成，每个网格节点上都有一个质点，如图 4.4.1 所示。这个网格中的每一个边都表示一个弹簧连接。至于如何在计算机内存中保存这样一个物体，我们把问题留给读者朋友们。为了简单起见，我们使用的是 STL（标准模板库）向量。

4.4.4 典型的质点-弹簧模型

对于一个典型的质点-弹簧引擎，它的运作过程（procedure）是相当简单易懂的。需要被仿真的物体就表示为这个过程的输入。在完成一个时间段的处理之后，被更新的（变化了的）物体就作为过程的输出。然后，新版本的物体又作为输入提供给这个过程，如此循环下去。

质点-弹簧系统的高级算法可以用以下三个步骤表示：

1．计算所有质点上的作用力
2．求解碰撞检测和碰撞响应的运动方程的积分
3．将计算结果可视化

在这些步骤中，第一步是需要我们仔细审视一下的。可以将这一步的工作分解成下列步骤：

① 外力的计算。首先在所有的质点（由网格节点表示）上迭代，进行内力的计算，并把它们保存在质点力累加器中。由于假设的是系统中只有一个外力（重力），那么第 i 个点上所有的力可以表示为公式 4.4.5。

其中 m_i 是这个点的质量，而 $\vec{g}$ 是重力向量。

$$\vec{F}_i^T = m_i \cdot \vec{g} \tag{4.4.5}$$

② 计算节点间力。对所有的弹簧连接进行循环操作，并计算网格上节点之间的作用力。这个作用力是一个带有阻尼的虎克力，可以写成力矢量的形式，见公式 4.4.6。

$$\vec{F}_{12}^H = -k_s \cdot \left(\left|\vec{r}_1 - \vec{r}_2\right| - d\right) \cdot \hat{n}_{12} + k_d \cdot \left[\left(\vec{v}_1 - \vec{v}_2\right) \cdot \hat{n}_{12}\right] \cdot \hat{n}_{12} \tag{4.4.6}$$

其中的法线向量可以用公式 4.4.7 表示。

$$\hat{n}_{12} = \frac{\vec{r}_1 - \vec{r}_2}{\left|\vec{r}_1 - \vec{r}_2\right|} \tag{4.4.7}$$

其中，k_s 是弹簧的弹性系数，k_d 是弹簧的阻尼系数。我们会在“几个仿真实例”一节中给出这些系数的一些典型值。请大家注意，用公式 4.4.6 计算出来的向量是两个相互作用的点上正负两个向量累加得到的。

③ PSB 模型步骤将在下一节讨论。

④ 数值积分。数值积分的步骤应该在这里实施。请大家注意，即使是一阶欧拉积分器

也可以很好地适合一些有限的物理特性。关于数值积分，请大家参考[Ancona02]。

对于组成可变形体的几何模型中的所有点，它们的运动情况可以用欧拉积分器给出的一个表达式来描述。假设已经累加得到了这些质点上的所有作用力，通过对所有质点的循环处理，可以写出牛顿第二定律的一个离散的形式，参见公式 4.4.8。

$$\begin{cases} \vec{v}_i^{n+1} = \vec{v}_i^n + \dfrac{\vec{F}_i^T}{m_i} \cdot \Delta t \\ \vec{r}_i^{n+1} = \vec{r}_i^n + \vec{v}_i^{n+1} \cdot \Delta t \end{cases} \tag{4.4.8}$$

其中，$n/n+1$ 枚举了时间步长（n 表示的意思是“在前一个时间步长”）。采用这个简单的积分器，就可以用 PSB 模型来仿真可变形体了。后面会向大家展示，采用预估修正法的 Heun 积分器在可变形体仿真中的应用。

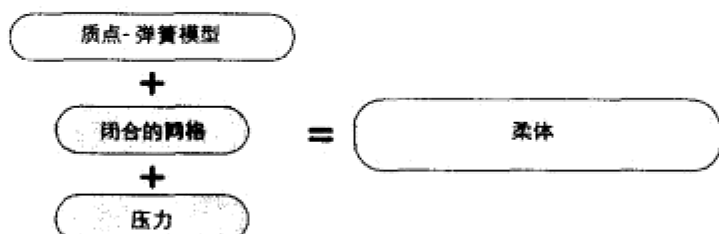


图 4.4.4　PSB 模型概览，在布料动力学上又增加了两个改进的地方，产生了实时可变形体

正如在前面的算法中所看到的，我们提出的可变形体的仿真方法，完全是基于质点-弹簧模型的仿真（参见图 4.4.4）。这里要强调的是，该模型可以看成是通用质点-弹簧模型的一个增强版本，因此它适用于任何一个用来处理简单 SM 仿真的物理系统。

4.4.5　PSB 步骤

在前面一节中我们注意到，在计算作用力的过程中，需要调用一个 PSB 步骤过程（procedure）。而且，调用这个过程的同时，还要调用其他的作用力计算函数，这是因为 PSB 过程同样也要计算填充到物体中的气体的作用力。

前面介绍了如何利用公式 4.4.2 计算气体的压力。假设参考大气压为零，就可以重用公式 4.4.3 中的压力值。我们只要对物体所有的面进行迭代，确定每个面上的压力值，并把这个压力值分布到每个面所属的节点上就可以了。

4.4.6　体积计算

为了计算这些作用力，首先需要确定被仿真物体的体积。考虑到[Matyka03]，我们可以使用最简单的方法，为这个物体构造边界盒子，然后计算这些盒子的整体体积。

但事实证明，这样的一个近似方法给仿真工作带来了几个新的问题。意想不到的振荡，物体变得无穷大（压力也会增大），这些都是不成熟的体积近似计算方法带来的错误。[Matyka03]一文指出，我们需要寻找的是一个优秀的近似算法，能对物体的体积做出比较确的估算。研究发现，对于没有孔洞的封闭形状，或者是一个完整的面（例如，气球状的物体），利用高斯定理（Gauss’s theorem），就可以很容易、很快速地得到一个精确的物体体积。公式 4.4.9 的数学推导过程就此跳过，有兴趣的读者可以参考[Feynman01]一文，文中对高斯定理解释得很清楚。对于那些想要推导体积积分方程的朋友们，可以假设物体被放置在一个向量场中，形式是 $f(x, y, z)=x$。然后，就可以计算出该向量场的散度（divergence），结果等于 1。如果把这些计算结果代入特定向量场的高斯积分定理中，就可以得到一个物体体积的简单表

达式，写成公式 4.4.9。

$$V=\sum_{i=1}^{NUMF}\frac{1}{3}\cdot\left(\vec{r}_{i1}.x+\vec{r}_{i2}.x+\vec{r}_{i3}.x\right)\cdot\vec{n}_i.x\cdot A_i \tag{4.4.9}$$

其中，r_i是节点的位置，n 是三角形的法线，A_i是第 i 个三角形的场，而 *NUMF* 是组成物体的面数。

4.4.7 采用预估修正法的 Heun 积分

欧拉积分是求解运动方程积分最简单的方法，通常要求比较小的时间步长。同时，这个方法还会带来“硬性”问题，这是低阶常微分方程（ODE）积分方法的一个众所周知的缺点。为了让解决方案更精确、更稳定，我们决定使用更为复杂的积分器。可选的方案可能会是来自龙格-库塔（Runge-Kutta）家族（二阶中点法也是一个好的选择，参见[Matyka03，Ancona02]）的一个显式方法，或者是无条件稳定的隐式法中的某个方法，例如向后欧拉积分（Backwards Euler）。

我们决定在此采用一种介于隐式法和显式法之间的解决方案。一个半隐式的预估修正 Heun 积分器会为我们提供二阶精度，而且在某种程度上，它仍然和显式积分法一样简单。我们不打算在这里就算法的推导进行深入的剖析。感兴趣的读者可以阅读任何一本关于数值计算的书，比如[Ancona02]。

半隐式的 Heun 积分器包含两个主要步骤，对于一个典型的问题，可以用公式 4.4.10 来描述。

$$\frac{dy}{dt}=f\left(y^n,t^n\right) \tag{4.4.10}$$

这个积分的第一步（我们称之为“预估”）与欧拉积分一模一样，用来计算在下一个时间步长中 y 的值，参见公式 4.4.11。

$$\hat{y}^{n+1}=y^n+\Delta t\cdot f\left(y,t\right) \tag{4.4.11}$$

一般情况下，过程函数会在这里结束，而下一个时间步长则准备启动。但是，在 Heun 积分器中，还有一些额外的值要在下一步中（我们称之为“修正”）用到，也就是公式 4.4.12 中的第二个项。

$$y^{n+1}=y^n+\frac{\Delta t}{2}\left(f\left(y^n,t^n\right)+f\left(\hat{y}^{n+1},t^{n+1}\right)\right) \tag{4.4.12}$$

一个完整的半隐式积分法可以写成两个公式：公式 4.4.11 和公式 4.4.12。大家要注意两个 y 的区别，它们分别是从两个独立的步骤中（公式 4.4.11 和公式 4.4.12）得到的。

4.4.8 时间步长的计算速度

为了让读者对该方法的速度有个大致的概念，我们进行了一个基准测试。这个基准测试与[Meseure00]中采用的方法类似。我们仿真了一个圆环管落地的情况，但不进行额外的碰撞检测（我们只考虑了可变形体与地面的碰撞）。图 4.4.5 是 PSB 模型和[Meseure00]方法的对比情况。我们的仿真是在 AMD Athlon 1.4 GHz 处理器系统上运行的。需要指出的是，由于[Meseure00]中的测试工作是在 R10000 194 MHz 处理器系统上运行的，所以图 4.4.5 中的对比只是一种定性的比较。

从图 4.4.5 中可以看出，通过使用 PSB 模型，对于一个节点数高达 4500 的物体，我们仍然可以获得实时性能（每个时间步长不超过 25 ms）的仿真。

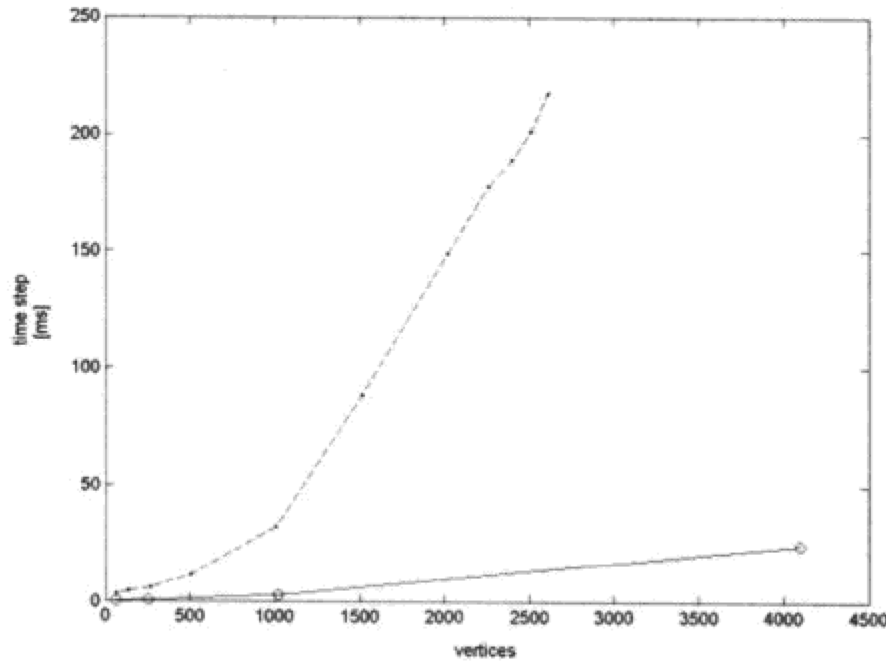


图 4.4.5 一个圆环管落地动作的仿真时间计算。PSB 模型（由小圆圈和直线来表示）和[Meseure00]方法（由黑点虚线来表示）的对比

4.4.9 几个仿真实例

图 4.4.6（a）显示的是一个没有内部压力的兔子模型的初始状态。在图 4.4.6（b）中，还是同样的兔子模型，但其中的内部压力 $P>0$。图 4.4.6（c）显示的是同一只兔子，用户在其运动过程中将其捕获。最后，图 4.4.6（d）中的兔子模型，有一个网格节点是固定不动的，但仍然有重力作用。这些图片都是从 Soft Body 3.0 软件中获取的。被仿真的兔子模型有 690 个顶点，1 376 个面。仿真过程中使用的物理约束是：$k_s = 350\,000$，$k_d = 10$，$P = 53\,000$，而每个节点的质量为 $m = 1.0$。这个仿真程序的运行速度是 50 帧/秒，运行平台是一个带有 Radeon 9200 图形卡的 Athlon 1.4 MHz 系统。

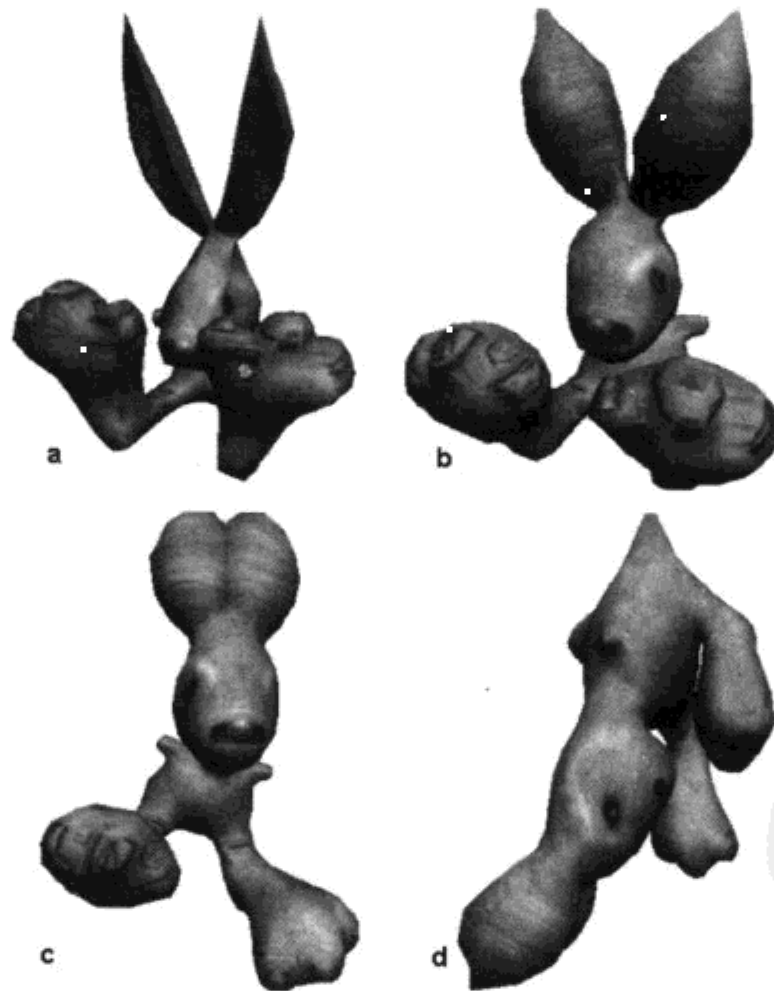


图 4.4.6 一个可变形兔子模型的仿真截图

PDG

图 4.4.7 向大家展示了又一个采用 PSB 方法的实时动画的例子。这次使用了一个圆球，顶点数为 642，面数是 1 280。圆球下落并与地面发生碰撞。在这个例子中，只进行了网格节点和地面（$y = 0$）之间的碰撞检测。使用的物理参数如下：$k_s = 121\ 100$，$k_d = 110$，$P = 6111\ 20$，而节点的质量 m=1.0。这个仿真过程的运行速度是 50 帧/秒，运行平台是一个带有 Radeon9200 图形卡的 Athlon 1.4 GHz 系统。

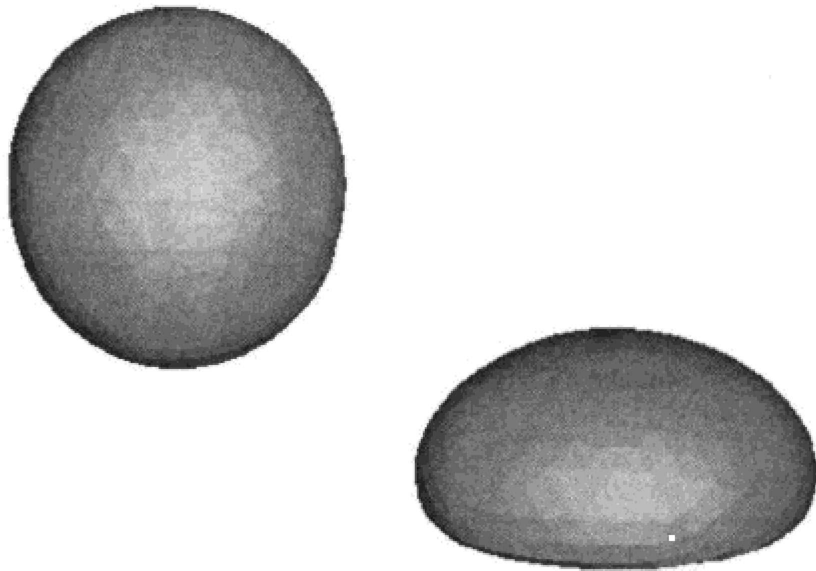


图 4.4.7 一个可变形的弹跳球的仿真截图

4.4.10 进一步的发展

可变形体的 PSB 模型还处于发展的初级阶段。目前，已经有一些项目采用了这个模型 [OpenCAL，Jello，MotionPlan]，而且业界也一直在尝试用各种不同的方法来开发这个模型。对于如何改进 PSB 模型，我们已经看到了很多不同的想法，大致罗列如下：

- 隐式积分的实现方法可以采用“柔体仿真中的大步长”一文中的实现方法[Barraf98]。
- 采用反动力学约束，包括对行为参数的实验。[Provot95]曾经使用过反动力学约束。
- 将物理模型与物体的图形表示独立开来，这样就可以更专注于简单模型的仿真，并以此作为复杂形状的仿真基础（参见[Meseure00]）。
- 实现物体和物体之间的碰撞检测及响应。为了本文叙述的方便，我们省略了这个“细节”（参见[Matyka03]）。
- 在物体网格中测试并使用非线性弹簧。
- 在 GPU 上进行碰撞的计算，实现一个实时的 PSB 仿真。

这些想法为 PSB 模型未来的发展提供了一些可能的思路。另外，值得一提的是，布料仿真动画领域的重要研究成果都可以作为我们的背景材料，并可以根据被仿真 PSB 物体的具体几何形状，进行更深入的开发。

4.4.11 总结

本文向大家展示了一个可变形体的实时仿真系统。这个模型之所以值得我们进行更深入的开发，主要就在于它是基于著名的质点-弹簧模型，可以很容易地进行升级和扩展，以便处

理各种柔体的仿真。同时，我们还将一个具体的仿真实例与之前的系统实现进行了对比，从中可以看出这个方法的性能所在。

为了简化模型，除了需要计算质点-弹簧模型中的有些作用力，还需要计算一个额外的作用力，这样就牺牲了一些计算时间。在该系统的开发工作中，我们把主要的精力放到了合适的物体构造和仿真参数的选择上。

真诚地希望这个模型可以满足游戏开发人群的需求。关于该方法的更新信息、源代码的最新版本和应用程序，会在作者[Matyka05]的网站上及时地向大家提供。

4.4.12　源代码说明

本文提供的源代码包含了部分 Soft Body 3.0 程序，开发中使用了 MS Visual C++的编译器。关于这些源代码的更新信息，大家可以在作者的网站上获得（参见[Matyka05]）。这些代码为我们提供了一个实现带有用户交互的三维可变形体的实时仿真系统的解决方案。

4.4.13　致谢

在此，要特别感谢 Jos Stam，他为我们提供了利用高斯定理（Gauss's theorem）计算物体体积的宝贵想法。还要感谢 Mariusz Jarosz 为本文的图片提供了三维渲染支持。

4.4.14　参考文献

[Ancona02] Ancona, M. G. *Computational Methods for Applied Science and Engineering: An Interactive Approach*. Rinton Press, 2002.

[Barraf98] Baraff, David and Andrew P. Witkin. "Large Steps in Cloth Simulation." Proceedings of SIGGRAPH 98, pp. 43–54, 1998.

[Callen85] Callen, H.B. *Thermodynamics and an Introduction to Thermostatistics*. New York: John Wiley & Sons, 1985.

[Feynman01] Feynman, Richard P. "The Feynman Lectures on Physics, Vol. 2.1." PWN, Warszawa, 2001.

"Geometric Models." Computer Graphics (Proceedings of SIGGRAPH 86), 20(4), pp. 151–160, 1986.

[Irving04] Irving, G., J. Teran, and R. Fedkiw. "Invertible Finite Elements for Robust Simulation of Large Deformation." ACM SIGGRAPH/Eurographics Symposium on Computer Animation (SCA), 2004.

[Jello] Mecklenburg, P. and C. Miller. *Jello Simulation*. Available online at *http://vorlon.cwru.edu/~prm8/eecs466/overview.html*.

[Lander03] Matyka, M. "Inverse Dynamic Displacement Constraints in Real-Time Cloth and Soft-Body Models." In *Graphics Programming Methods* (edited by Jeff Lander). Charles River, 2003.

[Matyka03] Matyka, M. and M. Ollila. "A pressure model for soft body simulation." Proc. of Sigrad. UMEA, November 2003.

[Matyka05] Matyka. M. *http//panoramix. ift. uni. wroc. pl/~maq/eng/*. Author's home page.

[Meseure00] Meseure, P. and C. Chaillou. "A deformable body model for surgical simulation." J. Visual. Comput. Animat., pp. 197–208, 2000.

[MotionPlan] Gayle, R.*Motion Planning for Physically-based Deformable Objects*. Available online at *http://www.cs.unc.edu/~rgayle/Courses/Comp259/MPDO/mpdo.html*.

[Nixon02] Nixon, D. and R. Lobb. "A fluid-based soft-object model." *Comp. Graph. and App., IEEE*, Vol. 22 Iss. 4, pp. 68–75, 2002.

[OpenCAL] Dierckx, J. *Open CAL Project*. Available online at *http://sourceforge.net/ projects/ opencal/*.

[Provot95] Provot, Xavier. "Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior." *Graphics Interface* '95, pp. 147–154, 1995.

[Sedenberg86] Sederberg, Thomas W. and Scott R. Parry. "Free-Form Deformation of Solid Geometric Models," *Computer Graphies* (Proceedings of SIGGRAPH 86) 20(4), pp. 151-160, 1986.

4.5 使用反馈控制系统让“布娃娃”活起来

苹果公司，Michael Mandel
mmandel@gmail.com

有些时候，在与动态环境进行交互时，游戏中的角色看上去不是那么的真实，这是因为游戏角色的动作行为都是预先设定好的，要么是动画师通过关键帧来设定，要么就是通过动作捕捉系统预先采集生成的。为了对付这个难题，业界普遍使用的是“布娃娃物理系统”（*Ragdoll physics*）。其基本思想就是当角色与角色之间发生碰撞，或者角色与环境之间发生碰撞时，对角色身体的物理反应进行建模。其最大的好处就是可以应用到游戏中，这样当玩家把敌人扔出去，与场景中其他的物体发生碰撞时，游戏角色相应的反应就会非常逼真。但可惜的是，由于缺乏必要的控制系统，无法生成更真实的行为，尽管有很多用处，但这个方法目前也只能应用在那些表现游戏角色倒地死亡的动画上。如果开发人员可以控制布娃娃的肌肉，它们就可以指挥自己的双臂，在倒地之前伸出去保护自己的身体，就像活人倒地之前的反应一样。

本文要讨论的，就是如何在仿真过程中控制游戏角色的身体，同时还要保持原先在身体和环境之间使用物理定律获得的真实表现。我们当然不能指望通过仿真来替换现有的动画数据，因为很多的人类行为都需要复杂的、协调的运动。但是，在很多情况下，通过使用简单的反馈控制器来生成肌肉力矩，很自然地指挥被仿真角色的四肢，确实可以得到更好的动画效果。在这方面，Color Plate 3 为我们提供了一个很好的例子。它向我们展示了控制器是如何改进普通布娃娃仿真系统，使角色的跌倒动作更具有和真人同样的行为特色。本文介绍的工具为我们提供了一个通用的方法，来驱动一个典型的布娃娃仿真，进而为仿真角色生成带有保护性的跌倒动作、平衡反应动作，甚至是跳跃动作或者滑铲动作。大家尽可以随意发挥创造力，只要可以对某个特定动作行为的底层控制定律进行建模。

4.5.1 现有的研究成果

有些研究人员已经使用物理控制器为各种人类行为生成了仿真动作。业界也展示了一些手动调整的反馈控制器，用来仿真体育运动中的一些行为，包括跑步、跳跃和骑自行车[Hodgins95]。Faloutsos 将组合式控制器（*composable controuers*）的思想进一步发扬广大，他把很多人类行为（例如跌倒、站立、平衡）组合起来，生成了一个虚拟的特技人[Faloutsos01]。

PDG

如果可以交互式地控制一个按自然法则仿真的角色，这会为我们打开一扇通往新型游戏性机制的大门。Laszlo 等人一直在研究如何通过直观的用户界面，让玩家直接控制仿真角色，使其做出奔跑、攀爬和一些体操动作[Laszlo00]。

用现有的运动数据作为接口操纵那些被物理法则控制的动作行为，这个能力也是非常重要的。业界已经开发出了几种技术，使仿真角色可以对受到的外力作用做出相应的物理反应，并通过轨迹跟踪技术，平滑地返回到现有的动画中[Zordan02]。另外还有一些解决方案，使用了[Mandel04]方法或[Shapiro03]方法，以环境相关的方式，让仿真系统和动画数据自动竞争，以取得对角色的控制。生物力学的研究成果是另外一个起点，可以帮助我们深入地理解人体是如何执行一个特定的动作的。还有一些可以进行刚体仿真的仿真器，比如 Open Dynamics Engine[Smith04]，它们都是免费的，读者可以利用它们进行控制器的开发实验。本文介绍的这些方法则是一个底层控制机制，它对上面提到的那些研究成果是至关重要的。

4.5.2 仿真过程的控制

我们通常把用“布娃娃”法控制的角色表示成一个有关节的人形物体。这个人形物体是由一系列的刚性部件及连接这些部件的关节构成的。刚体仿真引擎用一组基本几何图元来表示身体的每一个部位，每一个部位都是有质量的，也有一定的惯性属性。我们选择合适的关节将这些部位连接起来，并用相应的物理法则约束条件将每个关节的运动控制在实际人体关节运动的范围之内。控制布娃娃仿真的基本组成功能就是关节的目标姿态，以及计算关节力矩的方法。关节力矩会驱动一定的动作，达到想要的目标姿态。对于关节力矩的计算，本文将涉及常用的比例微分（Proportional Derivative，简称 PD）控制器。那我们该如何指定目标姿态呢？有一个选择，就是使用少量的预先设计好的姿态，用时间进行分隔或者根据事件进行转换。这种方法被称为“姿态控制器”，这些目标姿态会将仿真程序导向某个行为中的关键元素，如跳水动作中转体和抱膝的位置[Wooten96]。连续控制器（Continuous Controller）可以根据系统当前的状态（四肢的位置和速度），自动地生成目标姿态。这种与仿真系统反馈之间的紧密耦合，使得连续控制器比姿态控制器更为动态化，但是也可能会使其更难以进行人为地指定。在仿真过程中，一个跌倒动作的连续控制器应该注意观察肩部和臀部的速度的发展变化，并不断地调整上肢的目标位置，以尝试阻止跌倒动作。

1. 这不就是关键帧吗?

通过目标姿态的导向序列来控制仿真程序，这听上去有点像关键帧方法。但二者之间确实存在一些重要的区别。首先，作为输入信息提供给控制器的只是我们预期的关节角度，并不是实际的关节角度。控制器会计算肌肉力矩，驱动关节向期望值（关节角度）靠近。但是，四肢的动作仍然要表现出所有作用力的作用效果，包括那些来自于仿真环境的作用力。由于环境中的作用力，关节可能永远也无法达到我们的预期值。举个例子，如果身体的一只胳膊拎着一个重物，那么关节控制器就无法产生足够的力，让关节运动到预期的位置。第二，角色的全局位置和方向并不是直接指定的，而是从角色与环境的自然交互作用中推演而来的。例如，角色总是会因为重力的作用而下落。最后一点，预期的姿态并不像关键帧方法那样是预先确定好的。在使用连续控制器时，系统可以自适应地选择相应的姿态，以此作为系统状态的一个功能。

2. PD 控制器中力矩的计算

PD 控制器是一个非常有价值的工具，它提供了一个底层控制系统，驱动着仿真角色的预期动作。关于这些控制系统的详细介绍，可参考 *Dorf*[Dorf98]。控制器的输入参数是：预期的关节角度 q_{des}、系统的状态 $x = [q\ \dot{q}]$，以及各种传感器数据（比如两只手的接触状态）。而控制器的返回数据就是作用于每个关节的肌肉力矩 τ_i，以便驱动关节到达一个预期值。这就是一个闭环系统最典型的配置（之所以是闭环的，是因为它要求得到系统状态的反馈）。图 4.5.1 是这个基本的控制器反馈循环的结构示意图。

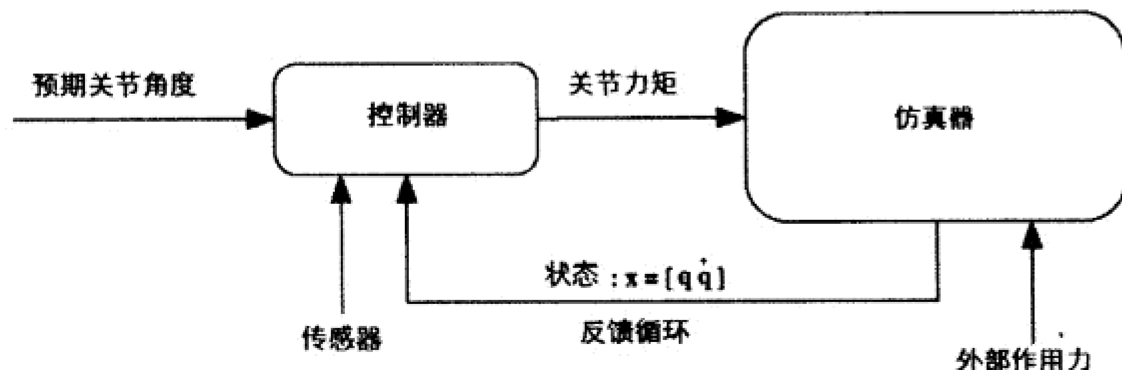


图 4.5.1 一个基本的闭环反馈系统的结构示意图

对于每一个关节，PD 控制器利用下列公式计算所需的力矩：

$$\tau_i = k_p(\theta_{des} - \theta) - k_d\dot{\theta}$$

其中，k_p 和 k_d 分别是比例增益和微分增益。θ 和 θ_{des} 分别是当前的关节角度和预期的关节角度，而 $\dot{\theta}$ 则是关节当前的速度（在其依附的身体上产生角速度）。要想维持一个稳定的仿真，最好将关节力矩的值限定在一个合理的最大值之内。下列程序代码片段实现了 PD 控制器要做的计算工作：

```
void ApplyPDControlTorques(Vec3 *Kp, Vec3 *Kd, Vec3 *des, int numJoints)
{
     for(int i = 0 ; i < numJoints ; i++)
     {
          Vec3 torque;
          Vec3 vel = GetJointVelocity(i);
          Vec3 cur = GetCurrentAngleForJoint(i);
          torque[0] = Kp[i][0]*(des[0] - cur[0]) - Kd[i][0]*vel[0];
          torque[1] = Kp[i][1]*(des[1] - cur[1]) - Kd[i][1]*vel[1];
          torque[2] = Kp[i][2]*(des[2] - cur[2]) - Kd[i][2]*vel[2];

          if(torque.length() > MAX_TORQUE)
               torque = MAX_TORQUE*torque.normalize();

          ApplyTorqueAtJoint(i, torque);
     }
}
```

3. 控制器增益的调整

PD 控制器的比例增益和微分增益参数 k_p 和 k_d 可以控制结果响应曲线，这使得它更像是

一个弹簧和阻尼器。协调好这些参数是很关键的，否则就无法得到很自然的仿真动作。比例（强度）增益控制着弹簧的强度，而微分（阻尼器）增益则可以调整关节到达预期值的平滑程度。对于弱阻尼（under-damp），我们会得到一个振荡响应，关节同时也会超过预期值；如果是强阻尼（over-damp），那么达到预期值的过程就会非常缓慢。在二者之间的某个位置上，可以得到一个临界阻尼，使系统达到完美的平衡状态：关节可以很快地达到预期值，但不会超过预期值。传统的做法是通过手动的方法调整这两个增益的值。这个过程多少有点太耗费时间。如果真打算采用手动调整的方法，一个好的经验是在开始的时候将比例增益和微分增益之间的比例设为 10:1。

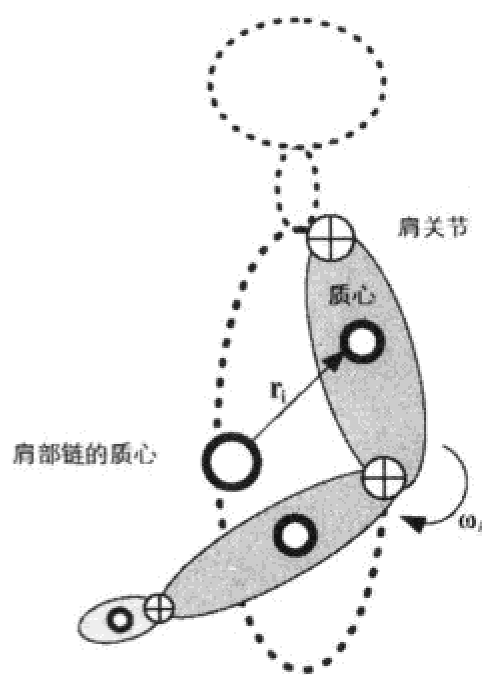


图 4.5.2 一个身体部位链的例子，它影响着肩关节的运动。我们可以计算出这些身体部位的惯性力的相对力矩，这样就可以非常高效地调整控制器的两个增益值

有一个技术可以大大减少需要手工调整的参数个数：根据每个关节所影响的身体部位链的惯性力的有效力矩[Zordan02]，将计算得到的力矩按比例进行缩放。例如，上臂、前臂和手部的惯性力的相对力矩会影响肩关节，如图 4.5.2 所示。使用这个技术，对于整个身体而言，需要调整的参数就可以减少到一个强度和阻尼参数。这是因为，最后的增益数值是由每个关节所影响的身体部位链来调整的。搜集每个关节所影响的身体部位，根据下列公式计算惯性力的相对力矩：

$$H = \sum_i (m_i r_i \times v_i + I_{CM_i} \omega_i)$$

其中，m_i 是某个身体部位的质量，r_i 是身体部位相对于链的质心的相对质心（center of mass，简称 CM）。v_i 是身体部位相对于链的质心的相对速度。I_{CMi} 是身体部位相对其质心的惯性张量，而 ω_i 则是身体部位的有角速度（参见[Kwon98]）。除了这个技术，读者也可以尝试使用一些优化技术，找到最适合某个给定行为的增益值，如模拟退火算法（Simulated Annealing）和遗传算法（Genetic Algorithms），参见[Sims94]。

4.5.3 行为动作的创建

有了一个底层的控制机制，可以控制被仿真角色的关节之后，现在就可以开始创建某些特定的行为动作了。有限状态机是一个常用的方法，它可以管理运动控制状态之间的转换。状态的转换通常都是基于时间或是基于事件的。针对不同状态的目标，可以为它们定制出新的控制器增益值，或者是预期的角度值。如果可以根据这些值进行状态的转换，可能会更容易。举个例子，一个跌倒动作控制器可能会包含几个不同的状态，将两只胳膊甩向跌倒的方向。这个过程要求被仿真的身体保持一定的刚度，同时要求双臂做出快速的反应。当双臂接触到地面时，系统就要切换到一个新的控制器状态，消化这个跌倒动作带来的冲击，降低臀部和上身的速度。图 4.5.3 展示的跌倒动作控制器，仿真的是一个后跌动作。

只考察仿真体的原始状态并不能为控制器状态提供最有用的信息。实现一些传感器是非常有帮助的，可以为控制器提供必要的信息，如支撑多边形、质心、身体部位之间的接

触，以及角色面对的方向。利用这些信息，再加上个人的直觉或者少许的生物力学知识，就可以将一个行为分解成最基本的控制状态。我们需要提前做好规划，确保控制器可以针对游戏过程中可能发生的各种不同的输入，做出稳定的响应。另外，一定要确保控制器的目标状态是可管理的。简单的控制器只是为了让角色最后仆倒在地上，而要创建一个可以生成稳定运行步态的控制器则是件相对困难的事情。创建仿真的行为固然是很困难的，但是作为游戏开发人员，我们的创造力、灵活性，会极大地影响最后的结果。

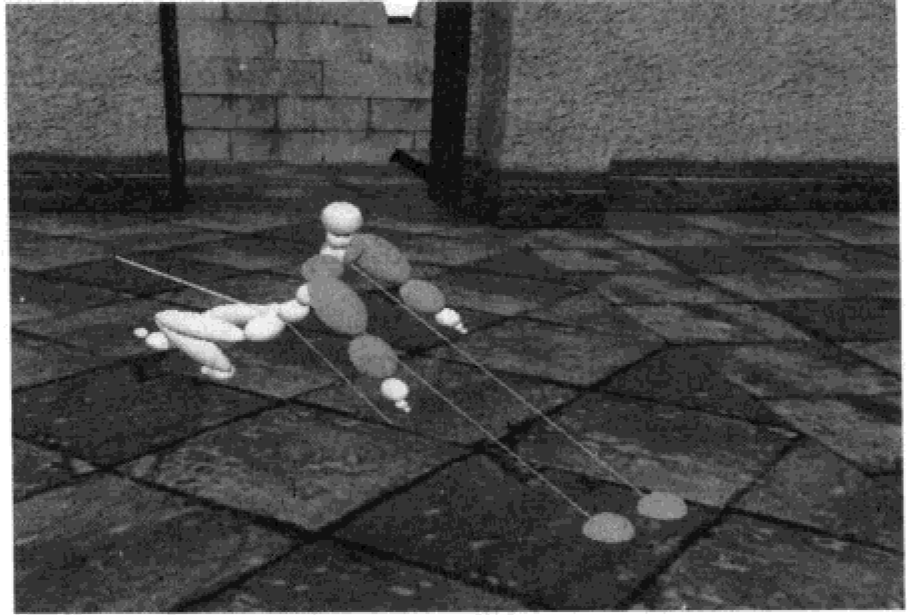


图 4.5.3　一个跌倒动作控制器的例子。地面上的两个圆球点表示的是预计中的肩部着地位置。它们是作为参考点来计算双臂的预期位置。向前的速度和角色面对的方向用向量来表示，以便根据跌倒的方向确定当前的状态。©2004 Iikka Keranen & Rich Carlson 版权所有，已取得刊登授权

4.5.4　总结

本文介绍的技术可以为现有的“布娃娃”角色增加更多有趣的行为。PD 控制器可以驱动角色的四肢，使它在整个倒地过程中的动作更符合物理定律。虽然关节预期角度值的计算，以及相应参数的手动调整是需要一些技巧的，但这里介绍的方法确实可以减少需要手动调整的参数个数。使用这个技术能有多少创造性的成果，完全取决于使用者的创造力。但是，随着我们不断地发现更好的方法，对人类的行为动作进行建模，无限可能尽在其中。

4.5.5　致谢

感谢 David Cherry、Iikka Keranen 和 Rich Carlson，他们为本文的某些插图提供了几何场景。

4.5.6　参考文献

[Dorf89] Dorf, R. C. *Modern Control Systems*. Addison-Wesley, 1989.

[Faloutsos01] Faloutsos, Petros, et al. “The Virtual Stuntman: Dynamic Characters with a Repertoire of Autonomous Motor Skills.” In *Computer Graphics*, Vol. 25, no. 6: pp. 933–953.

[Hodgins95] Hodgins, Jessica, et al. “Animating Human Athletics.” In *Computer Graphics*, Vol 29, Annual Conference Series: pp. 71–78.

[Kwon98] Kwon, Young-Hoo. “Mechanical Basis of Motion Analysis.” SIGGRAPH 2000, *Computer Graphics Proceedings*: pp. 201–208.

Available online at *http://kwon3d.com/theories.html*.

[Mandel04] Michael, Mandel. “Versatile and Interactive Virtual Humans: Hybrid use of

Kinematic and Dynamic Motion Synthesis." Master's thesis, Carnegie Mellon University, available online at *http://www.city-net.com/~amandel/portfolio/masters.html*.

[Shapiro03] Shapiro, Ari, et al. "Hybrid Control for Interactive Character Animation." In *Pacific Graphics* 2003: pp. 455–461.

[Sims94] Sims, Karl. "Evolving Virtual Creatures." SIGGRAPH 1994, Computer Graphics Proceedings: pp. 15–22.

[Smith04] Smith, Russell. "The Open Dynamics Engine." Available online at *http://ode.org/*.

[Wooten96] Wooten, Wayne, et al. "Animation of Human Diving." *Computer Graphics Forum*, Vol 15, no. 1: pp. 3–14.

[Zordan02] Zordan, Victor, et al. "Motion capture-driven simulations that hit and react." In *ACM SIGGRAPH Symposium on Computer Animation*: pp. 89–96.

4.6 预定式物理系统的设计

Daniel F.Higgins
webmaster@programming.org

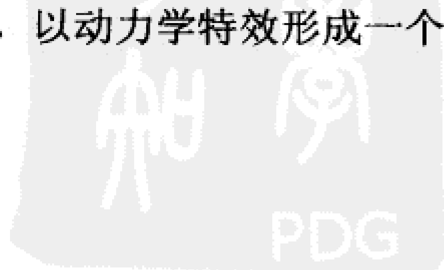
毋庸置疑，游戏开发是一项高强度、高耐力的运动。这项运动的领先者，那些处于领导地位的开发商们，拥有的是一批经验丰富、非常老练的团队，他们统治着不同的游戏流派。这一切应当归功于他们令人吃惊的专注程度、不断的创新以及万丈的激情。开发商们就好像是在玩一场高赌注的扑克牌游戏，围坐在无限压注的牌桌前，不断地将彩池金额抬高到天量。我们紧握着手中的纸牌，指望着能够依靠新游戏的技术、“趣味因子”，以及粉丝们的热爱，来赢得这场赌局。因此，如果有竞争者翻开他的那手赢牌，我们只有傻坐在那里，嘴巴张得大大的，看着自己的筹码眨眼之间烟消云散。这样的情形司空见惯，也没什么可令人惊讶的。

为了加强手里的这把牌，我们不得不四处寻找灵感。我们可以从《魔戒三部曲：双塔奇兵》的海尔姆斯深谷（Helms Deep）战役中找到灵感；也可以从滑板乐园中蠢笨的滑板技巧中找到灵感；还可以通过观察边境牧羊犬（Border Collie）跃起来叼住满是黏液的网球来找到灵感。

本文向大家介绍的引擎就是一个灵感的产物。这个灵感对我们是一个挑战：如何设计一个物理系统，使它产生的物理特效可以让人误以为它是一个大制作物理引擎。同时，这个引擎的开发周期必须要短，都并可以避免大量的开发支出。另外，无论是新手，还是老练的程序员，这个系统对他们都是一样可用的。而且，这个系统还要维系足够的灵活性，可以集成到一个独立于游戏的物理工具中。最后要说的是，这个系统已经呱呱落地了，它的名字就是“预定式物理（*Prescripted Physics*）”系统。

4.6.1 什么是预定式物理系统

平日里，当我们与游戏开发人员讨论物理学时，通常都会包含诸如速度、摩擦力、刚体、重力和加速度这类的字眼。大家或许还听过“布娃娃”这个词。以前，游戏物理学的功能就像是一个运载工具，沿着一条路径，枯燥地将物体从 A 点移动到 B 点。怎么说它都更像是一个反应迟钝的旋转木马，让我们感到万分痛苦。在今天这个充满竞争的游戏开发竞技场上，“儿童游艺设施”显然已经不行了。现在的引擎不但要让物体沿着路径移动，还要可以把它们猛抛出去，使之破墙而过，并让那垛砖墙逐渐变得松垮，直到坍塌下来，且砖块之间互相碰撞，以动力学特效形成一个碎石堆。竞



争对手都装备有大制作的物理引擎，与他们在游戏开发竞技场上展开白刃战，听起来确实有点吓人。那么，如何才能加入这场混战，与装备精良的对手展开对抗呢？其实也简单，我们可以制作一个“仿制品”，用它仿制出复杂物理系统中的物理特效。说白了，就是当要移动物体时，我们让它们沿着一个预先设计好的轨迹来移动，就像游乐场里的过山车那样。

即使我们使用的是一个“真”的物理系统，预定式物理系统也是很有用的，它可以驱动仿真程序中那些更能烘托氛围的元素行为。例如，场景中的汽车，虽然它的运动轨迹不是预先指定的，但是当它爆炸时，爆炸所产生的残骸碎片的特效（弧线飞行和四处崩溅）就是预先编码指定的。

1. 预定式物理系统是如何工作的

预定式物理系统的工作方式，是用一条曲线路径将一系列三维的点连接起来。要想把这个过程形象化，最简单的方法就是把这些点看做是一条晾衣绳上夹衣服的夹子。现在我们想像地面上有一条晾衣绳，上面以固定的间隔夹着十几个衣服夹子。如果这条晾衣绳不太紧，是比较松弛的，那么这些衣服夹子之间的线就是一条曲线。在预定式物理系统中，我们用路径点（waypoint）来表示这些衣服夹子，用样条线（spline）来表示晾衣绳。

创建预定式物理事件的第一步是要求我们有一定的观察艺术技巧。制作人员必须首先观察那些想要仿制的事件，可以在电脑屏幕上看，也可以在现实生活中观察，或者凭借自己的记忆。接下来我们对这个事件进行分解，分解成几个物体运动的关键帧，比如在这里调头转弯、在那里旋转、启动爆炸或者弹跳动作结束。一定确定了这些关键帧，接下来程序人员就要进行“反向工程”，以确定这个事件是如何发生的。该过程就是从时间的角度对游戏物理进行编码，而不是使用传统的速度累加的方式。简而言之，对于每一个关键帧（或者说是路径点），程序人员要记录物体在这个点上的方向和三维空间的位置，以及从事件开始已经过去的时间。然后，将这个路径点列表传递给物理引擎。物理引擎负责确保物体在给定的时间出现在给定的路径点上，物体也会正确地在这些路径点之间移动，以正确的方向达到正确的位置。最后，物理引擎再处理所有指定的其他属性（比如物体最后的朝向，以及相应的动画）。

为了说明这个工作流程，可以假设一个物理事件。在这个事件中，要把小鸡 Edgar 发射到空中。我们首先要明确想要的最终物理表现效果，然后再沿着飞行路线设置路径点。

对于小鸡 Edgar，我们能够做出的最简单的物理事件就是让它沿着一个固定的路径移动，如图 4.6.1 所示。这个事件可能是世界上最无趣的物理事件了，而且任何人看到这个事件之后都会不知所谓，疑惑万分，哪个游戏会容忍如此“假”的特效呢？不用担心，马上就可以解决这个问题，修改该物理事件中最不自然的地方：Edgar 自始至终一直保持着古怪的姿势。虽然保持良好的身体姿态无疑是一个值得称赞的品格，但是因为我们想要 Edgar 看上去更加自然，所以需要在它飞行的时候让它有点前倾（参见图 4.6.2）。

这可以算是个改进吧，但仍然不是那么令人信服，还是有点儿假。让我们尝试一下，还 Edgar 在飞到第一路径点时转入一个表情惊慌的动画。有了这个动画，物理事件就和以前大不相同了。但是，我们可不能就此打住。这场战役的胜利或失败，完全取决于细节！搞室内装饰的人都知道，如果没有足够的枕头和一个可以躺的沙发，装饰工作就不算完。同样地，我们也需要一些额外的东西，给 Edgar 赋予生命。这些额外的东西可能会是一些图像特效，比如在 Edgar 从正常的动画状态切换到一个受到惊吓的动画时，我们可以随机地让一些羽毛

PDG

迸溅出来（参见图 4.6.3）。最后，在进行状态转换时，还要大声地播放小鸡的尖叫声。这样做效果就应该很不错了。虽然还可以把这个事件装饰得更好，但让我们暂时先进行到这里吧。

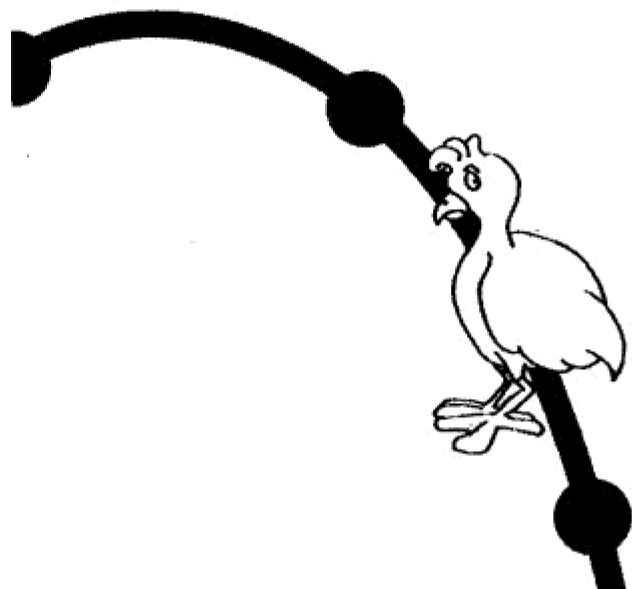


图 4.6.1　小鸡 Edgar 以固定的姿势做无聊的飞行

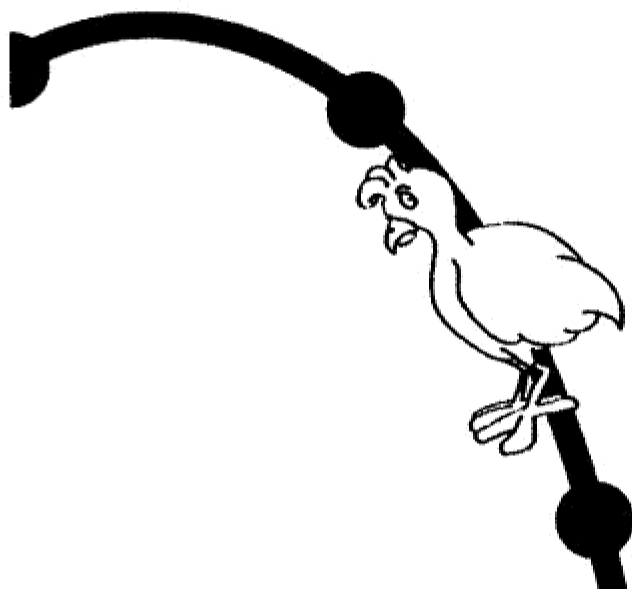


图 4.6.2　小鸡 Edgar 在飞行途中身体略有前倾

多亏了小鸡 Edgar，我们才可以了解到，一个沿着固定路径点移动的物体也可以得到与传统物理引擎一样的终极效果。要知道，虽然预定式物理系统是把物体的移动精简成简单的关键帧，以及连接这些关键帧的路径，以此来仿真那些大型的复杂的“真”物理引擎，但最重要的是添加的细节内容，是它们让整个事件活灵活现，同时也掩盖了基本技术的简单性。

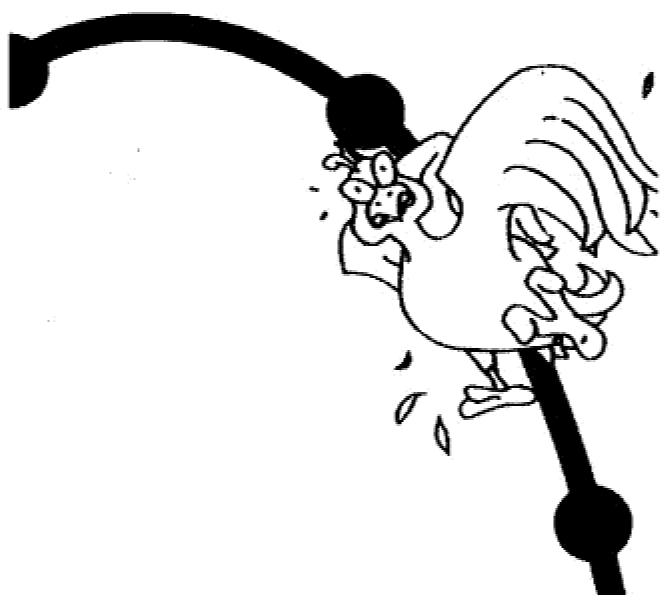


图 4.6.3　动画及其他特效可以帮助我们让 Edgar 飞行特效看上去像个大制作的物理特效

2. 预定式物理系统的优点和不足

和生活中所有的事物一样，预定式物理系统也有其自身的优点和不足。这些优点和不足主要是因为该系统并不是实时地计算物理数据。相反，它是在事件开始的时候就完成了物理数据的计算，这才是关键所在，只有理解了这个差别，我们才能真正认识到预定式物理系统的优势和弱点。

预定式物理系统的优势体现在以下几个方面：

易于使用：那些更先进的物理系统要涉及很多复杂的数学计算，而有些开发人员可能并不精通这些数学计算工作。预定式物理系统就是为方便这些朋友的使用而设计的。

开发周期短：创建一个预定式物理引擎需要的时间很短。一个周末的时间就可以把原型设计好，不到一周的时间就可以开发出可供游戏使用的版本。在要创建新的定制的物理事件时，开发周期短的好处就体现出来了。我们根本不用花几天的时间去实现一个新物理事件所涉及的复杂的数学计算。大部分情况下，只需要用几个小时就足够了，而且不需要太多的程序编码工作。

工具：可以把预定式物理系统制作成一个工具。这样，美工和游戏策划人员就可以使用这个工具，为游戏世界中的物体生成“物理文件”。这样，一旦创建好预定式物理引擎，可以供游戏使用了，程序人员就可以彻底抽身出来，去做别的事情。

PDG

性能：在使用物理系统时，大家最常面对的一个障碍就是对性能的担心。由于预定式物理系统中的物理事件都是根据定义预先计算好的，因此在 CPU 上的实时性能开销就非常之小。

重放：在整个游戏世界中，预定式物理引擎非常适合对物体的移动过程进行“倒带重放”，就像体育比赛转播中的“即时回放”一样。

预定式物理系统的不足则体现在下列几个方面：

反应：如果想让预定式物理系统与游戏世界中的其他物体进行实时交互，比如一个正在飞行的棒球被一只球棒击中了，那么我们就要编写额外的代码，以便可以支持这种反应。在预定式物理系统中，由于物体行进的路线在物理事件开始的时候就已经计算完成了，因此要想支持这个反应功能，预定式物理引擎必须支持碰撞检测功能，并且有能力将该事件“变态”为一个反应事件。这个功能的编码工作需要花费的时间不亚于开发一个完整的预定式物理引擎所用的时间。

独特性与生疏感：如果美工和游戏策划人员没有合适的工具可用，就必须仰仗程序人员来制作物理事件。程序人员首先需要理解这一点：在使用预定式物理系统时，他们想问题的方式应该是不同于以前的。我们并不需要一个数学天才对这些物理事件进行编码，但确实需要有关人员有足够的观察技巧，并且有能力将观察到的东西转换成可以实施的几个步骤。面对一个如此异类的物理系统，在开始的时候，很多有数学天赋的程序员也会疲于应付，或者干脆拒绝使用这个系统。这也没什么可奇怪的。一旦他们把自己的微积分书本抛在一边，像看电影那样去考虑游戏中的物理事件，鼠标键总是会被按下，他们一定会喜欢这个系统的。

复合物理：对于复合物理表现，例如前面提到的反应动作，我们就需要付出额外的工作才能实现。在通常情况下如果一个物体是在预定式物理系统的控制之下，游戏世界中的其他物理特性就不应该再对它产生什么影响了，除非可以编写出特殊的代码来支持相应的物理作用。

4.6.2 预定式物理引擎

最基础的预定式物理引擎有两个主要部分：三维路径点的移动以及物体方向的变化。无论是物体的移动，还是物体的方向，我们使用的都是一个百分比数值。这个百分比数值的计算是以时间作为插值计算的基础。对于位置，我们将时间转换成一个百分数，表示物体在样条曲线（从点 A 到点 B 之间的样条）上的位置。对于方向，同样是使用百分数来确定物体的偏航、俯仰和滚转的角度。例如，一个路径点从世界时间 100 毫秒开始，一直走到世界时间 250 毫秒截止，就会得到一个 150 毫秒的路径长度。那么当世界时间为 75 毫秒时，就走完了 50%的路径长度，也就是（175–100）/150。使用这些百分数，通过样条和四元数，就可以确定物体的位置和方向。

1. 三维点的移动

不妨把物理事件看成是在玩连点（connect-to-dots）游戏。在这个版本的游戏中，那些点是已经按顺序编好了数字号码的，而且我们已经知道要勾勒出的形状是一个人的面部轮廓。由于大部分的脸部轮廓都不是方形的，所以当在那些点之间勾勒轮廓时，要在其中增加一些艺术格调，使两个点之间的连线变成美妙的曲线，而不是生硬的直线。其结果就是漂亮的曲线让脸部轮廓看上去更自然，比简单地用尺子画直线更让人满意。

就像是在玩连点游戏一样，点和点（或者说是路径点）之间勾画连线的方式决定了用户

将要看到怎样的整体画面。大多数的预定式物理引擎可以很容易地支持多连点移动（multiple connect-the-dot）算法，包括最基本的线性方法，以及几个样条算法的实现。本文将集中介绍如何使用 Catmull-Rom 样条[Dunlop00]，在点和点之间移动物体。

要想了解这个新的物理引擎的“本末倒置”，速度（velocity）是个很好的例子。速度是基于我们摆放路径点的位置，以及允许物体在两个路径点之间移动的时间而确定的。在使用该物理系统时，这是程序员最常面对的一种逆向思维障碍。给定一个物体的加速度和速度，然后把它派发到游戏世界中，我们对这种方式已经太习以为常了，以至于当我们意识到，一个在时间量 T 内移动了 X 距离的物体也可以很好地适用于原先的速度公式时，反而会大感困惑。

程序清单 4.6.1 是一个基本的函数，用于生成当前的位置。它使用的参数包括：百分数（inPercent）、百分数的平方（inTSquared）、百分数的立方（inTCubed）、给定位置的三维坐标（inOurPoint），以及该位置之后 n–2、n–1 和之前 n+1 间隔位置上路径点的三维坐标（分别是 inBack2、inBack1 和 inNext）。

程序清单 4.6.1 给定一个位置的 *x*、*y*、*z* 坐标，Catmull-Rom 样条法的计算函数

```
inline float GetSpline(float inOurPoint, float inPercent,
                       float inTSquared, float inTCubed,
                       float inBack2, float inBack1, float inNext)
{
    return (0.5f * (2.0f * inBack1 +
                ((-inBack2 + inOurPoint) * inPercent) +
                 ((2.0f * inBack2 - 5.0f * inBack1 + 4.0f *
                  inOurPoint - inNext) * inTSquared) +
                ((-inBack2 + 3.0f * inBack1 - 3.0f * inOurPoint)
                + inNext)* inTCubed)));
}
```

程序清单 4.6.2 则是一个简化了的函数。给定一个百分数，函数就会按照这个比例将物体向下一个路径点的方向上推进。该函数还调用了 GetSpline 方法，利用当前路径点的世界坐标位置来确定物体当前的三维位置。举个例子，如果 inPecent 的值为 0.5，那么物体当前的位置就应该是当前路径点和上一个路径点之间连线上的中间。

程序清单 4.6.2 沿着样条路径向前推进一个点

```
void AdvanceSpline(float inPercent, GE3DPoint& ioPoint, const
GE3DPoint& inBack2, const GE3DPoint& inBack1,
const GE3DPoint& inNext)
{
inPercent = min(1.0F, inPercent);
float theTSquared = inPercent * inPercent;
float theTCubed = theTSquared * inPercent;

/* X 方向的样条 */
ioPoint.mX = ::GetSpline(mPoint.mX,
                  inPercent, theTSquared, theTCubed,
                  inBack2.mX, inBack1.mX, inNext.mX);

/* Y 方向的样条 */
```

```
    ioPoint.mY = ::GetSpline(mPoint.mY,
                     inPercent, theTSquared, theTCubed,
                     inBack2.mY, inBack1.mY, inNext.mY);

    /* Z 方向的样条 */
    ioPoint.mZ = ::GetSpline(mPoint.mZ,
                     inPercent, theTSquared, theTCubed,
                 inBack2.mZ, inBack1.mZ, inNext.mZ);
}
```

程序清单 4.6.3 是另外一个 AdvanceSpline 函数，它使用了大部分预先计算好的样条。如果希望样条的计算速度更快，可以将大部分的数学计算提前进行，并将结果保存在一个查询表中。（要注意的是，在本文后面几节内容中会看到，我们是可以在运行时影响一个物体的物理路径的。如果是那样的情况，还是应该采用普通的、速度稍慢的样条计算方法，因为每次物理路径发生变化时，都必须重新计算样条）。

程序清单 4.6.3 使用数据缓存方法的样条计算

```
/* 使用一个预先计算好的数组，实现样条的快速计算 */
inline float GetFastSpline(float* inArray, float inSquared,
          float inCubed, float inPercent)
{
return (0.5f * (inArray[0] + (inArray[1] * inPercent) +
         (inArray[2] * theTSquared) + (inArray[3] * theTCubed)));
}

/* 简化后的 AdvanceSpline 方法 */
void AdvanceSpline(float inPercent, GE3DPoint& ioPoint)
{
ioPoint.mX = GetFastSpline(mOptimizedSplines,
                                 theSqd, theCube, thePercent);
ioPoint.mY = GetFastSpline(mOptimizedSplines + 4,
                              theSqd, theCube, thePercent);
ioPoint.mZ = GetFastSpline(mOptimizedSplines + 8,
                              theSqd, theCube, thePercent);
}
```

2. 朝向和四元数

“朝向”描述的是一个物体的偏航、俯仰和滚转的角度。简单地讲，朝向并不是一个物体的位置，而是在当前位置上物体所面对的方向。我们经常说某个物体是大头朝下或者歪歪扭扭，指的就是物体的朝向。说实话，我们真的很幸运，用四元数来处理物体的朝向是最适合不过的了。四元数不但在物体朝向的平滑调整上有着出色的表现，而且还特别适合于前面在三维移动和样条计算中使用的时间-百分数（time-percent）的计算范例。如果那个范例还不够好，那么使用四元数来推进物体的朝向也实在是太简单了。对于物体朝向的处理，四元数法的魔力作用，在于它使用了球状线性插值（spherical linear interpolation，也被称为“slerping”），可以沿着一个曲线平滑地缩放物体的朝向值（参见[Svarovsky00]）。

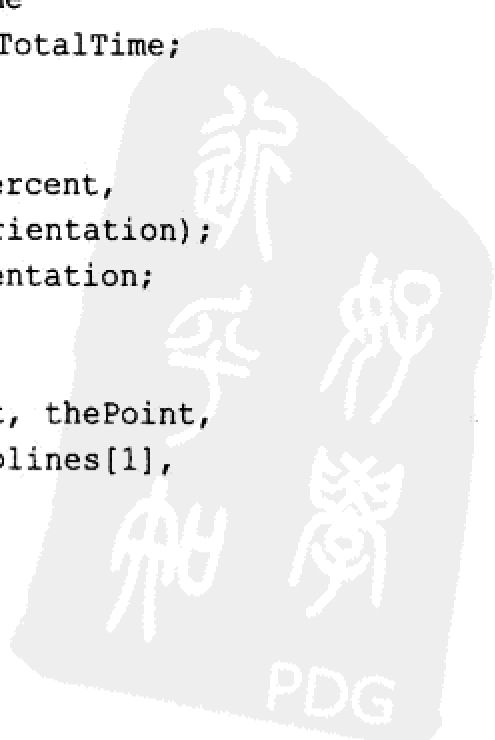
3. 发动引擎

现在是时候将所有的部件组装起来，制造出一个轰鸣的引擎了。这个引擎会让游戏迷们兴奋不已。这个基本引擎的核心部分很简单。它的工作就是处理所有的物理节点，使用样条来推进物体的位置，然后利用四元数推进物体的朝向。最后，系统将正确的朝向应用在物体上，并为物体设置正确的位置。

通过阅读程序清单 4.6.4，可以清楚地看到预定式物理引擎更新工作的代码大致是什么样子的。更新工作的第一个操作是让节点出栈，直到我们位于正确的时间范围之内。例如，世界时间是 200，节点 3 从 195 行进到 220，我们会将节点 1 和节点 2 弹出栈，确保在节点 3 上是准备好的，可以进行处理。一旦得到了当前节点，我们就会推进物体的朝向（使用时间和四元数），然后再推进物体的三维位置（使用时间和样条）。最后，在结束更新工作之前，要确保物体的旋转是围绕着物体上某个正确的点（底部、头部或者中部，等等）进行的。这个引擎实现起来非常简单、快速，也异常的容易。

程序清单 4.6.4　预定式物理引擎的更新方法（Update Method）

```
float          thePercent;
bool           theResult = true;
bool           thePointMatches;

/* 确保下一个点是正确的点（时间正确的点） */
AdjustToCorrectPointUsingTime(inWorldTime);

/* 堆栈中还有物理节点吗？ */
if(mPhysicsNodes.empty() == false)
{
GE3DPoint thePoint;

        /* 得到我们的点 */
        inCurrentLocation.GetPosition(tnePoint);

        /* 获得第一个数据项 */
        FPNode& theNode = mPhysicsNodes.front();

        /* 计算百分数（计算的结果限定在 0~1.0F 这个范围之内，主要是为了 AdjustToCorrectPoint
方法使用的方便） */
        thePercent = ((float)(inWorldTime -
theNode.mStartTime)) * theNode.mInverseTotalTime;

        /* 推进物体的朝向 */
        theNode.AdvanceOrientation(thePercent,
                                   mQuatOrientation);
        outResultOrientation = mQuatOrientation;

        /* 推进物体的位置 */
        theNode.AdvanceSpline(thePercent, thePoint,
        theNode.mSplines[0], theNode.mSplines[1],
```



```
    theNode.mSplines[2]);

            /* 在正确的地点进行旋转（头部/中部/等等） */
            ApplyRotation(theNode, outTransform, thePoint);
    }
    else
    {
            theResult = false;
    }
```

4.6.3 打磨上光

几乎与游戏行业中的所有事情一样，细节上的精雕细琢会让我们脱颖而出，否则只能是抱残守缺，停滞不前。下面这些项目需要我们进行细化，它们对引擎物理表现的优劣程度有着重大的影响：加速度斜坡（acceleration ramp）、动画和路径点特效。还有其他一些项目，包括：将样条上的路径点映射到物体或地形上，围绕物体上一个指定的点进行旋转，物理事件重放，或者触发相应的反应事件。

1. 加速度斜坡和事件倒放

几乎在每个回合中，预定式物理系统似乎都在与传统的物理观念进行着斗争。如果不使用加速度，我们怎么可能在一系列的空间点之间使物体自然地运动呢？幸运的是，四元数需要一个0～100%之间的百分数，以便正确地确定物体的朝向。正是这一点才使得定制加速度斜坡对我们如此重要。

普通的物理引擎通过作用力将物体从A点移动到B点。这些作用力会产生加速度，而加速度反过来也会随着时间的发展，不断地改变物体的速度和加速度，直到达到一个预期的速度，或者物体移动到一个预期的位置。在预定式物理引擎中，我们使用加速度斜坡来作用于百分数（0～1）。它反过来也可以告诉我们，物体在路径上已经走了多长的距离。

现在想象一下，假如我们要将小鸡Edgar沿着直线从A点移动到B点。如果不使用加速度斜坡，Edgar就会以一个固定不变的速度移动。但是，如果采用一个正弦（sine）斜坡或余弦（cosine）斜坡来影响当前的百分数，Edgar就可以快速地移动，只在接近目的地时才逐渐慢下来；或者是开始的时候移动得很慢，只在接近目的地时再开始加速。加速度斜坡产生的最终结果可能是：给定一个百分数25%，对它进行修正，并返回一个值为12%的百分数。然后，我们可以执行样条计算，确定物体的朝向。但此时，我们考虑的是12%这个百分数，而不是25%这个百分数。我们还可以使用一个斜坡修正器（ramp modifier），作为斜坡处理函数（ProcessRamp）的一个可变参数（mOrientationRampModifier）。假设要使用一个余弦（cosine）斜坡，而且可能只需要用到一半的余弦曲线，我们就可以通过使用这个浮点类型的斜坡修正器（ramp modifier）参数，来控制斜坡处理函数（ProcessRamp），控制它如何改变输入的百分数（inPercent）。很自然地，这些斜坡处理函数（ProcessRamp）也应该集成到AdvanceSpline方法和AdvanceOrientation方法中，请参见程序清单4.6.5。但不要忘了，对于物体朝向和移动的处理，应该使用不同的斜坡处理函数（ProcessRamp），因为二者的处理过程是独立进行的。

程序清单 4.6.5 带有一个斜坡处理函数（ProcessRamp）的 AdvanceOrientation 方法

```
void AdvanceOrientation(float inPercent, Quaternion& outQ)
{
/* 使用斜坡处理函数（ProcessRamp）改变输入的百分数 inPercent */
inPercent = ProcessRamp(inPercent, mOrientationRamp,
                        mOrientationRampModifier);

/* 来吧，我的宝贝（slerp to my-lou my darling）*/
      mStartQuat.Slerp(mQuaternion, inPercent, outQ);
}
```

“事件倒放”是一个很简单的特性，其操作方式与斜坡处理函数非常类似。如果在每个事件节点完全执行之后，将这些事件节点保存起来，我们就可以翻转百分数（比如，25%变成 75%），或者让时间值反绕。我们可以看到小鸡 Edgar 的移动和定向过程的倒放，看着它从目标路径点沿原路返回到起始节点。

2. 将路径点映射到地形和物体之上

使用预定式物理系统的一个最大的担心，就是虽然在事件开始的时候为某个物体规划好了路径，但是突然在某个时刻，游戏世界中发生了意外事件，影响了物体的路径，最终导致整个物理事件的崩溃。举个例子，假设有个物体从一幢特别高的楼上掉下来，我们确定最终的着地点是一个漂亮的长满青草的山顶。但是，在它下落到一半的时候，突然发生的一个爆炸将整座山全部摧毁，使之变成了一堆废土，而且物体的高度也比我们预定的着地点低了很多。希望尚存！我们可以在物体下落的过程中，实时地改变路径。但如果要这样做，就必须调整事件中相关的（或者全部的）路径点时间和样条，CPU 的效率也会因此大大降低。所以，总得来讲，我们并不喜欢在预定式物理事件中突然蹦出个什么“惊喜”。

有很多工具可以帮助我们摆脱这种窘境，其中之一可以将路径点映射到地形或者其他物体之上。在创建阶段，当把一个路径点映射到地形上时，我们就可以肯定无论系统中发生了什么意外事件，这个路径点一定会坚决地落实到地面上。同样地，当我们把一个路径点映射到某个物体上时，这就意味着无论物体走到那里，这个路径点都会一直锁定在物体之上。如果真的决定采用映射的方法，一定要记住当路径点移动时，必须对它们的时间进行缩放，以便让它们的感知速度（perceived velocity）保持不变。这就需要调整当前在移动的路径点的结束时间，可能还要调整前方路径上所有节点的开始时间和结束时间。还有一项工作很重要，就是要调整当前这个路径点（以及前方路径上相邻的 2 个路径点）的样条位置，这是因为样条的计算需要用到反方向上路径点（n–2）的位置信息。

3. 旋转

对于在引擎中实施的旋转功能，它们通常都是围绕着物体的三维位置进行的。如果物体所在位置正好就是我们想要旋转的位置，那可就太合适了！否则，如果有个物体，其被定义的中心并不是几何中心，那就得进行一些坐标变换，以确保物体能够围绕着正确的区域进行旋转。举个例子，在一款即时战略游戏中，我们可以得到一个物体站立位置的三维空间点，但是我们可能想让物体围绕其中心位置进行旋转，甚至有可能想让物体不对称地进行旋转，

PDG

即从物体的最下部为中心开始旋转，然后在物理事件的整个过程中，把旋转的中心位置不断上移，直到最后以物体的最上部作为旋转的中心。程序清单 4.6.6 是一个未经优化的函数，说明了如何使用坐标变换来进行物体的旋转：

程序清单 4.6.6　使用当前的四元数让物体围绕着物体上指定的点进行旋转

```
GETransformation      theRotateCenter;
GETransformation      theRotateCenterInverse;
GETransformation      theTranslation;
GETransformation      theRotation;
GETransformation      theResult1;
GETransformation      theResult2;
GE3DPoint             thePoint;

/* 开始进行变换操作 */
this->mQuatOrientation.ConvertToTransform(theRotation);

/* 设置平移操作 */
theTranslation.SetPosition(inPoint);

/* 设置旋转中心（调整 z 坐标，而不是 x 和 y 的坐标） */
theRotateCenter.SetZ(mHeight * ioNode.mRotationModifier);

/* 求逆（对这个值与主变换进行保存） */
theRotateCenter.MakeInverse(theRotateCenterInverse);

/* 对旋转中心执行世界坐标的平移操作 */
theTranslation.Apply(theRotateCenter, theResult1);

/* 对世界坐标/旋转后的坐标使用这个结果 */
theResult1.Apply(theRotation, theResult2);

/* 从局部空间转换为世界空间 */
theResult2.Apply(theRotateCenterInverse, ioFinalTransform);
```

4. 反应

与传统的物理系统相比，“反应”是预定式物理系统最薄弱的部分。由于一个事件的所有内容都是提前规划好的，所以这种引擎对意外事件不能做出适当的反应，除非进行额外的编码，让引擎处理这些意外事件。一般来讲，所谓的“反应”主要是针对某些事件，创建相应的“反应事件”。这些事件可能是物体与墙发生碰撞，或者与其他的物体发生碰撞。当碰撞发生时，某个特殊反应事件应该检查当前的状况，创建相应的反应物理事件。反应事件应该沿着自己的路径执行，而不是沿着原先那个物理事件的路径。与前面讲到的将路径点映射到地形和物体的情况类似，反应事件也会实时地影响路径，但它还会造成更剧烈的变化。

假设我们使用预定式物理系统投出一个棒球。在棒球飞向捕手的过程中，突然出现一只球棒击中了这个棒球。这样，物体（棒球）的路径就发生了剧烈的变化，从而引发了一个反应。系统对反应的处理过程包括创建一个新的物理事件。在创建新的物理事件时，要考虑诸多因素：速度、当前的位置、球棒挥舞的力量等。然后，将这个事件提交给物理引擎。

物理引擎会中断当前的事件，并执行新的事件。

优化提示：碰撞检测工作的系统开销很高，所以如果知道某个路径点上不会有碰撞发生，跳过碰撞检测是很安全的。我们可以针对每个单独的路径点，打开或关闭碰撞检测功能。

5. 独立于游戏的工具

虽然对程序员而言，预定式物理事件的创建工作是非常容易的，但有时候还是需要根据美工和游戏策划人员的需求进行必要的改动，这是不可避免的。理想状态下，一旦预定式物理系统显露出了对游戏引擎的价值，我们就应该编写一个独立于游戏的工具，让美工和策划人员使用这个工具生成对物理事件的描述。工具的设计非常复杂，有太多的创建工作都需要检查相应的游戏世界，毕竟我们是在这个世界中创建物理事件的。有一个好主意，我们可以先制作出几个简单的物理事件，用它们验证预定式物理系统是否可以在游戏引擎中正常工作，然后再着手开发辅助的工具。否则的话，从创建一个预定式物理引擎到引擎可以正常运转，很容易就会拖长到几周的时间，而程序员也只能为用户界面的细节问题费尽脑筋，如美工和策划人员该如何“仿真重力”，或者“在一个圆环上增加路径点”。

4.6.4 龙卷风：一个好的开始

预定式物理引擎组装完成之后，我们终于可以享受一下编写物理事件的快乐感觉了。作为一个简单的例子，假设我们要编写一个龙卷风的事件。我们可以从物理事件基类中派生出一个类，重载 Create 方法，然后就可以通过一个简单的函数得到自己的龙卷风了。程序清单 4.6.7 是一个让龙卷风把物体卷到半空中的非常简单的版本。

程序清单 4.6.7 一个简单的龙卷风物理事件

```
/* 利用速度和距离，获得相应的时间 */
unsigned long theTimePerIteration =
::FloatToUnsignedLong((kPI * (2.0F * inRadius)) / inVelocity) * 1000;

/* 计算所需要的迭代*/
long theTornadoWaypointsPerPI = 35;
float theFloatIters = ((float)inTime /    (float)theTimePerIteration);
float theInterval = (theTimePerIteration / theTornadoWaypointsPerPI);
float theIncPerLoop = (1.0F / theTornadoWaypointsPerPI);

/* 设置起始点 */
thePNode.SetStartingQuat(inCurrentQuat);
theRotationTransform.SetPosition(inCenter);
theNewPoint = inStartPoint;

/* 播放动画 */
thePNode.SetAnimation(inAnimation);

/* 设置当前的点 */
```

PDG

```
thePNode.SetPoint(theNewPoint);

/* 启动一个新的点 */
long theNextYPRCounter = 0;
long theZCounter = 0;

/* 我们将使用的旋转数量 */
float theRotationAmount = -((kTwoPI / 35.0F) * 2.0F);
theZCounter = (theTornadoWaypointsPerPI / 4);

/* 对所有路径点的循环操作 */
for(theFLoop = 0.0F;
    theFLoop < theFloatIters;
    theFLoop += theIncPerLoop,
    theZCounter-,theNextYPRCounter-)
{
/* 将最新的 z 值保存起来，供以后使用 */
      theLastZ = theNewPoint.GetZ();

      /* 是否重设偏航、俯仰和滚转的步进增量？ */
      if(theNextYPRCounter == 0)
      {
             /* 重设下一个步进增量*/
             theNextYPRCounter = 4;

             /* 取反，而不是除 3 次 */
             theInverse = 1.0F / (float)theNextYPRCounter;

             /* 计算变化的范围 */
             theYaw = GetRandom(-TwoPI, TwoPI) * theInverse;
             thePitch = GetRandom(-TwoPI, TwoPI) * theInverse;
             theRoll = GetRandom(-TwoPI, TwoPI) * theInverse;
      }

      /* 朝向的初始化 */
      thePNode.SetOriYPR(theOri.mYaw + theYaw,
          theOri.mPitch + thePitch,
          theOri.mRoll + theRoll);

      /* 航向/目的地点 */
      theRotationTransform.RotateByYaw(theRotationAmount);
      theRotationYaw = theRotationTransform.GetYaw();

      /* 小提示：改变半径和时间，就可以生成一个漏斗 */
      /* 计算相对于半径的局部点间隔距离，然后再变换为世界坐标 */
   radius and transform into world space */
      theLocal.SetY(inRadius);
      theRotationTransform.Apply(theLocal, theWorld);

      /* 旋转世界坐标 */
```

```
        theFloatPoint = theWorld;
        theFloatPoint.RotatePointAroundZero(theRotationYaw);
        theFloatPoint.SetZ(theWorld.GetZ());
        theNewPoint = inCenter;
        theNewPoint += theFloatPoint;

        /* 是否获取一个新的 z 值? */
        if(theZCounter == 0)
        {
                /* 重设 */
                theZCounter = (theTornadoWaypointsPerPI / 4);

        /* 计算 z 的改变量(0.0F = TERRAZN Z) */
        theZChange = GetRandom(0.5F, 5.0F) + 0.0F;
        theZChange = (theZChange - theLastZ ) /
                        (float)theZCounter;
        }

                /* 设置位置，并将 z 值与当前的 z 值相加 */
                thePNode.SetPoint(theNewPoint.GetX(),
            theNewPoint.GetY(),
                        theLastZ + theZChange);

                /* 需要的时间 */
                theCurrentTime = thePNode.SetTime(theCurrentTime,
            theCurrentTime + theInterval);

                /* 添加到路径点列表中，以备后用 */
                theNodes.push_back(thePNode);
    }

        /* 将这些路径点添加到物理事件中，OK，全都搞定了！ */
        AddPoints(theNodes);
```

4.6.5　总结

本文中需要我们铭记在心的关键论点有：

游戏物理是很重要的：今天的市场需要的是可以匹敌电影和特效软件的游戏物理。不要害怕迎接挑战！不要走别人走过的路，解决方案总是会有的，只是需要我们去寻找。

预定式物理系统有它的优势：可以用它处理一些特殊的物理事件，如投掷物体、反冲物理，以及其他动画片段。

预定式物理系统有它的不足：在一款游戏中，如果传统的线性物理占据着主导地位，比如即时战略游戏中有一个走动的士兵，这时候就不要使用预定式物理系统了。

快速编码：我们可以非常快速地创建一个预定式物理引擎，而且对物理事件的编码工作速度还会更快。不要害怕，用省下来的时间去实验新的加速度斜坡、样条，或尝试细化其他需要打磨的地方。

不要急于开发物理工具：不要急于创建物理工具。首先要完成引擎的创建和事件的编码

工作，然后再开始创建物理工具。不管怎么说，把物理引擎搭起来，让它可以运行，这才是首要完成的工作。

对样条进行优化：如果不打算对物理点应用实时的作用力（比如被龙卷风卷起来的物体），那就可以把样条保存在一个数组中，减少数学计算的工作量。

美妙之极的四元数：如果之前没有用过，那就为应用程序编写一个四元数类，对物体的朝向进行插值处理，以获得平滑的旋转。

调整物体的旋转：可以使用一个浮点类型的修正器改变物体旋转的方式。它的值介于0.0F（物体的底部）和1.0F（物体的顶部）之间。否则，物体所有的旋转都是围绕着实际的三维坐标位置进行的，这个位置并不确定，可能会是游戏世界中的任何一个地方。

雕琢：本文只是详细地介绍了一个最基本的预定式物理引擎，还有很多方法可以对它进行改进。一定要记得在其中增加斜坡、重放、动画、音效、图像特效，以及所能想到的、可以让游戏熠熠生辉的其他特性。

在今天这个充满竞争的游戏行业中，游戏物理和图形图像同等的重要。图形图像勾画出了玩家看到的游戏世界。但是，如果不能让这个世界值得一看，再好的图形图像也是枉然。游戏开发人员都希望在游戏中增加有趣的物理特性，而大制作的传统物理引擎不但需要高额资金投入，而且需要很长的开发时间。对他们而言，预定式物理系统绝对是一个优秀的解决方案，它的实现需要的时间非常之短。

每年一届，我们大家坐在游戏开发商的“世界扑克系列锦标赛”的牌桌前，小心地看护着自己手中的牌，既紧张又充满自信。下一手牌该怎么出呢？你会打得很紧凑，主动进攻吗？你会固守现有的知识吗？你有创新精神吗，会去重新构筑手中的牌吗？无论选择何种风格的玩法，一定要记得，为了大家，我们要不断地抬升赌注金额。归根结底，这才是我们不断突破、创新，让客户志达意满的动力所在。

感谢 Richard Woolford 授权使用小鸡 Edgar 的形象。（在撰写本文的过程中，没有小鸡受到过伤害。）

4.6.6 参考文献

[Bourg01] Bourg, David, M. “Physics for Game Developers.” 2001.

[Dunlop00] Dunlop, Robert. “Introduction to Catmull-Rom Splines.”
Available online at *http://www.mvps.org/directx/articles/catmull*. 2000.

[Svarovsky00] Svarovsky, Jan. “Quaternions for Game Programming.” In *Game Programming Gems*, Charles River Media, 2000.

4.7 预定式物理系统：相关技术及应用

Shawn Shoemaker
shansolox@yahoo.com

游戏中的真实物理仿真为我们提供了充满活力的表现效果，但这样的物理仿真工作需要高昂的系统处理开销，特别是对一些较为普通的系统平台或者 Web 应用程序，这样的开销是无法承受的。另外，“真实”的物理仿真为游戏单位生成的位置和朝向都是没什么限制的，因此会引发一些意外的（无法预料的）情况。很多游戏都要求自己的游戏单位可以获得特定的位置和朝向信息，以此作为一个物理运动的结果。而且，还有一些游戏需要同时进行成百上千个游戏单位的仿真。预定式物理系统（Prescripted Physics）可以生成真实物理的“仿真”特效。本文向大家介绍了预定式物理系统的实现及其优点，并向大家展示了如何快速、低投入地获得好莱坞式的游戏特效，同时还不违背游戏世界的既定规则。目前，这些好莱坞式的游戏特效都是通过昂贵、复杂的物理引擎获得的。

在刚体物理学和通用力学领域，业界已经发表了很多文章，但其中涉及预定式物理系统的文章屈指可数。本文则以一款使用了预定式物理系统的实际游戏为例（不锈钢游戏工作室开发的游戏 *Empire: Dawn of the Mordern World*），详细地介绍了这款即时战略游戏中空投伞兵部队中涉及的游戏物理系统。本文的读者应该对基本的游戏物理、四元数和样条等概念有一些初步的了解，并且熟悉预定式物理系统的基本运行机制。这些内容在 Dan Higgins 的文章[Higgins05]中有详细介绍。

4.7.1 为什么要使用预定式物理系统

为什么不在游戏中使用真正的物理仿真呢？真正的物理仿真常常无法让策划人员准确地指定游戏中应该发生什么样的事情。这是因为，在很多情况下，真正应该发生的事情并不遵守那些物理定律。大部分游戏都喜欢好莱坞式物理特效中的那些爆炸性的、亦幻亦真的夸张视觉效果。一般来讲，真正的物理仿真很难得到这样的效果。举个例子，假设步兵团附近发生了爆炸。真正的物理仿真会把所有的游戏单位从爆炸地点飞抛出去，同时让某些游戏单位带点儿旋转效果。与此相反，好莱坞式的特效则会让所有的步兵单位飞上天，经过几次空翻，最后重重地坠落到地面上。这是一个非常戏剧化、愉悦观众视觉的特效。

也许有人会说，只要掌握了扎实的、有效的物理学知识，充分利用那

些最根本的物理学公式，程序员就可以改变环境参数，从而获得预期的效果。虽然这种想法是正确的，但预定式物理系统可以让程序员，更重要的是，也可以让游戏策划人员（他们可能真的不太精通物理仿真）全面地控制最终的物理效果。而且，由于这个系统通常都是为某个特定结果专门调整好的，所以对全局物理参数的改变会不可预知地把物理仿真导向不同的区域中。如果每个特定的结果都需要我们去调整物理仿真参数，那么程序员就会被困在这个工作上，脱身不得。既然如此，我们还有什么好的理由去使用一个真正的物理仿真系统呢？相反，如果使用预定式物理系统，我们每次都可以准确地得到各种情形下的预期效果。

预定式物理系统不但可以帮助我们获得预期的视觉效果，还可以让我们知道仿真过程最后的结束状态应该是什么样子。在真正的物理仿真中，我们只能知道起始状态和输入信息，但最终的结果通常是无法预先知道的，各种复杂的情况都有可能发生，但这个问题根本不会出现在预定式物理系统中，因为程序员或者策划人员首先确定的就是最终的结果，然后才开始规划相应的物理进程。对于那些有成千上万个游戏单位的游戏而言，预定式物理系统就显得更为灵活，而其他计算开销庞大的系统却没有足够的灵活性，游戏单位只能局限于它们自己的寻路过程中。

预定式物理系统的最后一个卖点，也是游戏制作人最欣赏的一个好处：开发周期短。仅在两个星期之内，就可以开发出一个可运行的简单的预定式物理系统。而一个可靠的刚体仿真系统则需要较长的开发周期，也许是几个月的时间。刚体仿真系统必须要面对一些躲不掉的挑战，如积分器（integrator）的不稳定性、摩擦力、静止约束（resting constraint）和渗透问题，但预定式物理系统则完全规避了这些问题。之所以要使用这样的一个系统，最根本的一点就是，预定式物理系统的实现、调试和维护工作的开销都非常小。

4.7.2 预定式物理系统

预定式物理系统的详细设计过程不在本文的范畴之内，但预定式物理系统的基本机制却是值得在这里介绍一下的。概括地讲，Catmull-Rom 样条和四元数的组合是预定式物理系统主要的组成部分。游戏单位沿着样条线段，以特定的速率移动，并在移动的过程中进行相应的旋转，播放相应的动画。

样条上的每个控制点（*control point*）都带有各种参数，我们可以设置这些参数，以便控制游戏单位的运动。首先，每个控制点都有一个总的时间量，这是游戏单位通过特定样条线段所用的时间。其次，每个控制点都有一个时间函数，用来控制游戏单位在样条线段上运动的“加速度”。该时间函数可以是一个简单的线性函数，也可以基于某些高阶的时间函数。总的时间与这个时间函数表示着样条线段的速度。

控制点还以四元数的形式指定了一个目标旋转量。这样，游戏单位在样条线段上移动的过程中，游戏单位可以转动到一个明确的方向。样条上任意一个给定点上的方向数据是通过两个控制点之间的插值运算得到的。最后，每个控制点还指定了一个特定的动画，游戏单位在样条上移动的过程中需要播放这个动画。我们可以实时地设置这些参数，也可以在设计阶段预先确定这些参数，最重要的是要选择好这些参数的值。

选择“好的”参数值

对于一个可用的预定式物理系统，它的物理特效从表面上看应该是真实可信的。例如，

游戏单位下落的过程应该看上去比较真实，而且碰到障碍物后应该有个反弹的过程。为了让人们相信这个系统的表现效果，它必须具备真实物理仿真中的某些属性，这一点很重要。那接下来的问题就是在预定式物理世界中，我们该如何对真实物理仿真系统的特效进行部分建模呢？

动力学为真实仿真提供了所有必需的公式。一个典型的做法，是从这些描述真实世界的公式中挑选出必需的公式来为特定的物理效果建模，如重力、浮力、阻力和摩擦力。然后，组合这些公式，并为物体的移动选择一些时间点，插入到相应的参数中。接着，在每个时间点上求解方程。将一系列的时间输入和相应的方程求解结果组合在一起，最后就会形成一个数据表。我们可以直接将这个数据表提供给预定式物理仿真系统使用。请大家注意，我们提高时间点的抽样密度，那么结果与真实物理仿真的近似程度就越高。

另外还有一种方法，也可以获得预定式物理系统的输入值。“脱机”运行一个真正的仿真系统，并随着时间的推进，将一些位置数据记录下来。然后运行一个“伪”仿真系统，来匹配那些从“真”仿真系统中获取的数据，这样就可以产生一个在视觉和感觉上都比较自然的物体运动。

一旦开发了预定式物理系统，并确定了其参数值的选择方法，我们就可以利用这个系统很容易地处理各种情况。下面的内容是预定式物理系统应用的几个实例。

4.7.3　应用 1：RTS 游戏中建筑物的毁坏

在三维即时战略游戏（RTS）中，建筑物就像是从地下长出来似的，而当建筑物被毁掉时，它们就好像又回落到地面以下。这个简单的行为可以很容易地用预定式物理系统来建模。我们需要一些较短的样条线段。样条的一端是建筑物的中心点，另外一端则是地形下面的一个点。二者之间的距离要足够长，可以将建筑物隐藏在地形下面。每个控制点的四元数都包含着一个很小的俯仰量（pitch）或转动量（roll），来仿真建筑物的震动效果。随着建筑物不断地下沉，控制点的速度也应该随之增加，这样看上去就好像建筑物在加速下沉。图 4.7.1 显示的是一个建筑物的下沉，它使用了 5 个控制点。在这个例子中，5 个控制点就足够了，可以充分表现出速度的变化和建筑物倒掉过程中的转动特效。

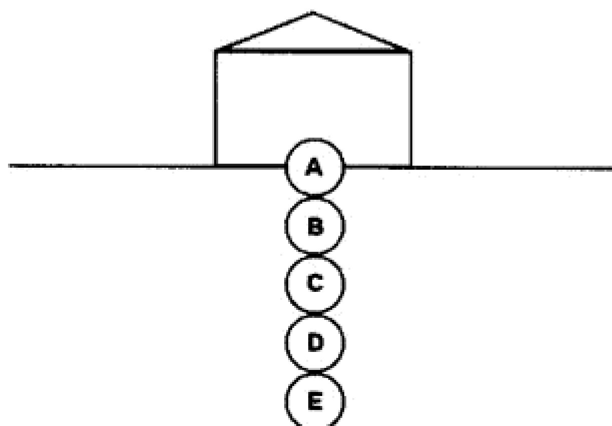


图 4.7.1　建筑物毁掉过程中使用的样条控制点

4.7.4　应用 2：跳跃

预定式物理系统的另外一个最直接的应用就是“跳跃”。游戏单位从某个出发点起跑，跑了一小段距离后，低头弯腰，然后跃起。经过很短的时间，游戏单位逐渐下落，最后落回到地面上。如图 4.7.2 所示，从控制点 A 到控制点 B，游戏单位应该播放一个跑步的动画，并不断地加速。在控制点 B 上，游戏单位应该播放一个“跳向空中”的动画，然后以很高的速度移动到控制点 C。在向控制点 D 移动的过程中，游戏单位的速度应该逐渐慢下来。虽然

在这个系统中，我们无法做到只调整垂直速度，而不修改水平速度，但是此仿真不需要这样的精度就可以看上去很真实。在控制点 D 附近，游戏单位的速度应该有一个微小的停顿，由于重力的作用，游戏单位向上的速度被抵消了。从控制点 D 开始，游戏单位加速通过 E 点，并很快回落到控制点 F。在控制点 F 处，游戏单位会播放一个着陆的动画，同时伴随着落地点上的地形视觉特效。我们在控制点 G 结束整个物理特效，给游戏单位留出一些空间，以便稳定自己的身形。

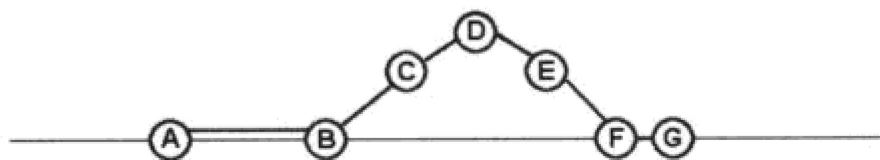


图 4.7.2 一个跳跃动作的样条控制点

4.7.5 应用 3：爆炸特效中的物体运动

预定式物理系统最得意的应用就是爆炸特效。如图 4.7.3 所示，图中的游戏单位都沿着类似抛物线的样条移动。但是，与真正的物理仿真不同，游戏单位在空中飞行时还可以不断地翻滚。所有的游戏单位都会从爆炸中心飞离出去。根据自己与爆炸中心的距离，以及一些随机化的因子，游戏单位可以从一些预先确定好的样条中选择适合自己的样条。每个样条上都有各自不同的控制点和随机旋转参数，控制着高度和旋转动作，这样每个游戏单位的飞行效果都不尽相同。

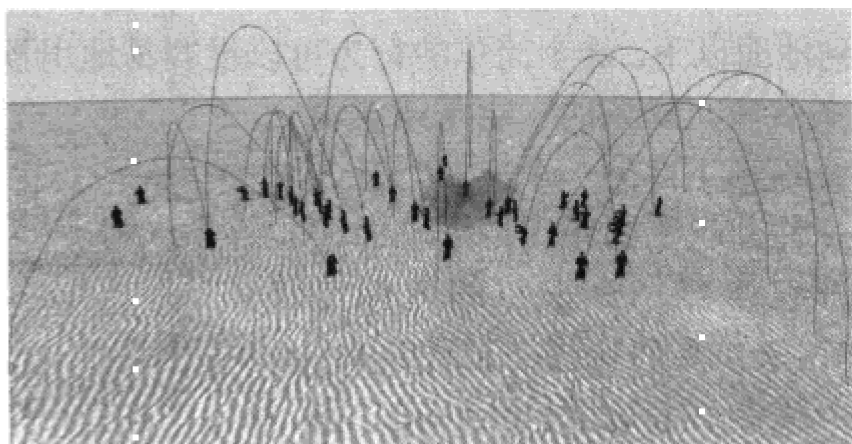


图 4.7.3 在爆炸特效中，游戏单位沿着一个抛物线路径进行移动。©2004. Stainless Steel Studio 授权刊用

飞行过程中的旋转动作应该均匀分布在几个控制点上，这样就不会出现某个样条线段上旋转动作过多的现象。将旋转动作进行分解，使得游戏单位可以执行一个比较完整的翻滚动作。在整个样条路径上，每个游戏单位的旋转方向应该是固定的。虽然在空中翻滚是一个好莱坞式的特效，但如果在飞行中途改变旋转的方向，对玩家会是一个很明显的刺激。在接近样条尾部的时候，无论是什么样的旋转动作，都必须将游戏单位的姿态调整好，使之以背部或腹部着陆，这是着陆动画所要求的。

我们还需要增加一些动画，来表现游戏单位是怎么飞上天的。游戏单位可以俯卧飞行，也可以仰面飞行。一旦身在空中，游戏单位就会播放一个半空中“失控”的动画。对于着陆动作，我们会相应地使用“腹部着陆”或“背部着陆”动画。如果着陆动画能够表现出游戏单位在着陆时的反弹和缓冲影响，这个物理特效就更令人信服了。最后，当游戏单位坠落到

地面上时，还要增加一个相应的地形碰撞视觉特效，例如扬起的尘土或者溅起的其他残骸碎片等。

4.7.6　应用 4：浮力

浮力的特效是很容易仿真的。让玩家信服浮力特效的关键是什么呢？游戏单位必须跌落到水中，下落的速度应该在水下有明显的降低。在水中下沉一段时间后，游戏单位就不再下沉，转而开始上浮。上浮的速度应该是越来越快，直到它浮出水面。或许游戏单位会再次沉入水平面，然后再浮起来，直到最后在水面上找到一个静止的位置。这就是一个真实可信的浮力特效所必需的过程。

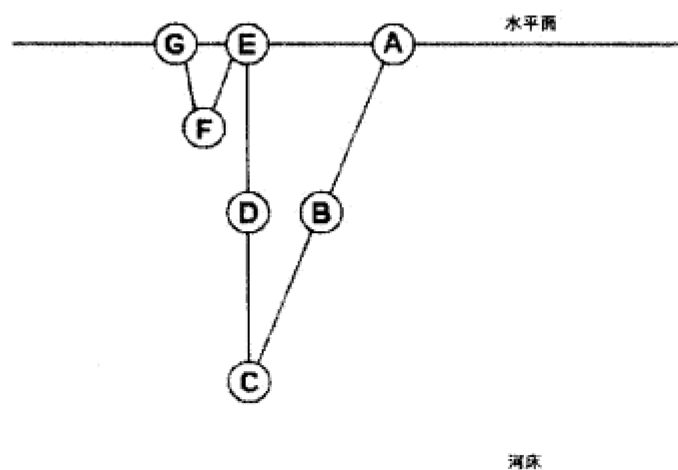


图 4.7.4　一个预先规划好的浮力特效所使用的样条控制点

图 4.7.4 向我们展示了浮力特效的一些细节信息。样条控制点 A 和 B 之间的速度应该有非常明显的降低。这个减速度应该持续到控制点 C。在控制点 C 上，游戏单位的浮力抵消了它的重力。然后，游戏单位稍作停顿，就开始向控制点 D 移动，速度不断增加，一直移动到控制点 E。在控制点 E 稍作停顿，游戏单位又移动到控制点 F。控制点 F 和 G 重复了重力和浮力相互作用的过程，表现了二次入水的过程，但是每个控制点的持续时间大大缩短。

为了增加真实感，同样需要增加一些动画。我们应该在控制点 G 和 E 上播放一个“踩水”动画，在控制点 C、D 和 F 上播放一个游泳的动画，在控制点 A 和 B 上调用落水动画。每次进出水平面的时候应该伴随着一个视觉特效，表现出水花四溅的效果。为了增加物理效果的多样性，游戏单位的密度和质量应该影响控制点的下落深度和运动速度。最后，如果游戏单位有足够快的速度，或者河水足够浅，游戏单位就可以用脚蹬一下河床，这样控制点 D 和 E 的速度就应该有所增加。

4.7.7　应用 5：伞兵

在游戏“*Empires: Dawn of the Modern World*”中，C-47 飞机会空投一定数量的伞兵。空投下来的伞兵应该沿着一条看似真实的路径降落到地面上。另外，伞兵应该降落在玩家指定的着陆点附近。伞兵不能降落在非法的地图位置上，否则会让玩家感到很沮丧。

我们为每个伞兵创建了 6 个样条控制点，将伞兵移动到预先计算好的着陆地点。每个伞兵样条控制点的位置会有微小的随机变化，以便伞兵的降落路径各不相同。一些并发的样条会让伞兵向不同的方向降落，进一步分散它们的降落地点。

图 4.7.5（另见彩插 4A）中展示了一个伞兵的 6 个样条控制点。控制点 1 是伞兵开始降落的位置，位于 C-47 飞机模型的一个标记点上。正如前面提到的，控制点 2、3 和控制点 4 会根据每个伞兵做一些微小的随机变化，这样同时降落的多个伞兵会各自有不同的飞行路线。在图 4.7.6（另见彩插 4B）中，可以很明显地看到这种多样性。这幅图提供了一个较好的场

景观察角度，还配备了几个完整的伞兵。控制点 5 和控制点 6 在样条的尾部形成一个弯钩形状。这个弯钩可以确保伞兵与预计着陆点上地形之间的交互作用。注意，控制点 6 实际上是在地形下面的。

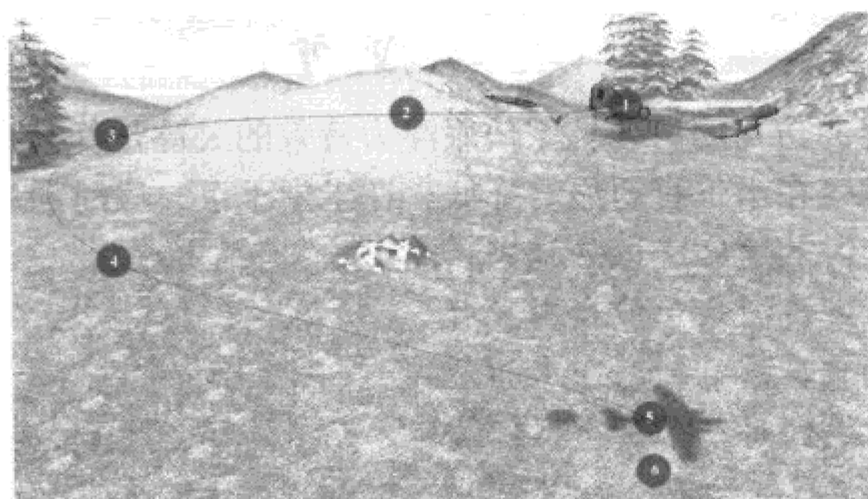


图 4.7.5 一个伞兵的 6 个控制点。©2004 Stainless Steel Studios 授权刊登

图 4.7.6 不同伞兵的不同样条路线。©2004. Stainless Steel Studios 授权刊登

在每个物理仿真片段上，结合已经过去的时间量，以及伞兵的加速度，我们可以计算出伞兵已经移动过的距离。每个样条线段有它自己的长度，这个长度在仿真开始之前就已经计算好了。通过这种方式，伞兵就可以确定自己当前是处于哪个样条线段上。一旦知道了它当前所在的样条线段，根据伞兵目前已经移动过的距离，以及当前所在线段的总长度，我们就可以确定该伞兵在这个样条线段上的新位置。

为了让伞兵的运动看上去更为可信，我们在样条路径上结合了一些特定的动画。伞兵离开 C-47 的位置上（控制点 1）增加了一个“快速出舱”的动画，且降落的过程中大量地使用了“漂浮不定”的动画。对于着陆动作，则使用了两个不同的动画，一个是伞兵抬起双腿准备着陆的动画，另外一个动画表现的是利用双腿来消化着陆时产生的震动冲击。预先规划好的降落运动与这些动画的结合，使得伞兵的整个降落过程更为真实可信。举个例子，“抬腿”动画与一个近乎平滑的地形结合，就会产生一个着陆时的滑行动作。这和我们在现实世界中看到的情况一模一样。

为了仿真风向的作用，我们还为每个伞兵增加了随机偏转的动作。当伞兵接近地面时，偏转速度会逐渐慢下来。另外，随着伞兵离地面越来越近，他们的身形尺寸也会逐渐地变大。这主要是因为，C-47 飞机尺寸的缩放比例与周围建筑物和伞兵的尺寸匹配得非常精确。因此，

开始的时候伞兵的尺寸都比较小，然后在降落的过程中逐渐变大。

4.7.8 总结

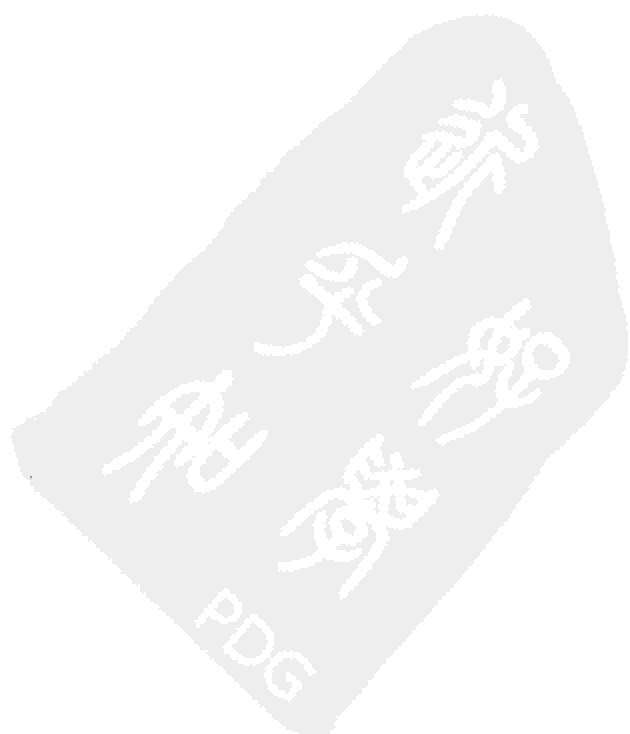
本文介绍了预定式物理系统在即时战略游戏和其他游戏中的一些应用。有几个地方需要再次强调一下。预定式物理系统可以为我们带来很多好处，包括实现周期短、好莱坞式的特效表现，以及与策划人员的想象完全一致的最终结果。我们可以利用真实的仿真为预定式物理系统生成初始启动数据。另外，样条和四元数是这个系统中驱动物体运动的关键元素。但是，仅有物体的运动是不够的，我们必须在预定的物体运动中结合必要的动画。

当然了，预定式物理系统也有一些明显的不足。由于一切都是预先规划好的，所以这个系统没有即时反应的能力。如果没有考虑到可能的碰撞问题，这个系统中的物体就不能彼此碰撞。另外，在预定式物理系统中，针对不同的应用环境，我们需要编程或创建出不同的运动路径。例如，一个物体从 40 层高的楼上降落下来，我们需要为它创建一个运动路径；而对于物体从楼梯上滚下来的动作，我们必须为它重新创建一个不同的运动路径。

尽管存在上述的问题，但是在即时战略游戏和其他类型的游戏中，预定式物理系统依然大有用武之地。在这些类型的游戏中，我们需要成百上千的游戏单位，以制造出好莱坞式的物理特效。在其他一些游戏领域中，有些游戏单位的运动无法使用简单的动力学公式来描述，这时候就可以使用预定式物理系统。获得所有这些好处只需要很少的时间投入，开发人员因此可以有更多的时间去创建更多有趣的游戏行为。

4.7.9 参考文献

[Higgins05] Higgins, Dan. "Designing a Prescripted Physics System." In *Game Programming Gems 5*. Charles River Media, 2005.



4.8 三维汽车模拟器中真实的摄像机运动

匈牙利布达佩斯技术经济大学，控制工程与信息技术系，图形图像小组，Barnabás Aszódi，Szabolcs Czuczor

ab011@hszk.bme.hu，cs007@hszk.bme.hu

在汽车模拟器或驾驶游戏中，整个游戏过程中通常会使用若干不同的摄像机机位，其中大部分时候是将虚拟摄像机锁定到移动的汽车上（如驾驶舱内部的摄像机，车篷顶部的摄像机，等等）。对于驾驶舱内部的摄像机，为了增加真实感，汽车每个微小的运动都会给摄像机造成同样的运动，这不但是我们可以接受的，而且正是我们所期待的。对于其他几种情况，例如位于汽车跑道一侧的虚拟摄像机，如果汽车的每个微小运动也使其发生震动，这反而会让我们颇感烦恼。另外，如果发生了碰撞，导致汽车很快停了下来，这个时候，如果摄像机也很快地停下来，就会显得非常不和谐。本文提出了一个摄像机模型，可以提高虚拟摄像机运动的真实程度。这个虚拟摄像机是可以有重量和惯性的，但它的行为基本上是比较自然的。因此，汽车由于在虚拟世界中是运动着的，所以它不可能总是位于屏幕的中央，时而会超出屏幕范围，时而会落在屏幕后面，以此来提高画面表现的真实程度。

4.8.1 我们需要什么？物理法则

在现实世界中，每个物体都遵守着连续性（continuity）法则。这就意味着，一个真实的物体不可能通过空间中突然的跳跃来改变自己的位置。如果没有这个法则，物体的速度就需要无穷大，而这是不可能的。因此，当我们把物体的位置定义为一个时间的函数时，如果这个函数是一条简单的直线，或者是带有很多曲线和角度的路径，那么它应该是连续的。参见公式 4.8.1。

$$\boldsymbol{r}=\boldsymbol{r}(t) \tag{4.8.1}$$

在这里，r 是一个物体的当前位置，用一个三维向量来表示，其值取决于时间。正如牛顿第二定律定义的那样，一个物体可以像改变其位置一样，连续地改变自己的速度，参见公式 4.8.2。

$$\vec{F}=m\cdot\vec{a}\Rightarrow\vec{a}=\frac{\vec{F}}{m} \tag{4.8.2}$$

F（作用力）和 a（加速度）都是向量，而 m（质量或重量）则是一个标量。当加速度无穷大时，物体的速度就会有一个跳跃值。这个表达式告诉我们，只有当作用力无穷大或者物体的质量为零时，上述情况才会发生。当然了，这在现实生活中都是不可能发生的。所以，当要控制一个虚拟场景中的虚

拟摄像机时，我们必须小心谨慎地对待这两个主要的定律，否则摄像机的运动就会非常不真实。

真实的摄像机是有质量的，因此也就具有惯性。为了简化摄像机运动中的计算工作，我们用两个点来表示虚拟摄像机：摄像机的取景孔（以下简称“摄眼”）和目标点（*object*）。摄像机从摄眼开始，朝目标点看过去，这意味着目标点将一直位于屏幕的中央。基于这两个点的相对位置，我们就可以定义摄像机的方向。由于摄眼和目标点都是有质量的，而且因为加速度的大小是有限的，所以真实物体的速度变化一定是连续的。我们也不能非连续地改变作用力，所以表示作用力的函数也只能是连续的。让我们反过来看看这个问题。

利用图 4.8.1 回顾一下，看看我们曾经提过的不同度数的连续性到底是什么意思。在讨论公式 4.8.1 的连续性的时候，这个概念非常重要，它以时间定义了物体的位置。图 4.8.1 中最左侧的一列，显示的是不遵守物理定律的不连续性运动。如果函数中没有突然的跳变，如图 4.8.1 中第二列所示，我们就称之为 C^0 连续曲线。为了计算物体在其运动路径上的速度，我们要取得这个时间函数的导函数。如果导函数也是连续的，如图 4.8.1 中第三列所示，我们就称之为 C^1 连续曲线。最后，考察路径的二阶导数，它表示的是路径上运动着的物体的加速度，如果它也是连续的，那我们就称之为 C^2 连续曲线，如图 4.8.1 中最右一列所示。为了让虚拟摄像机的运动比较逼真，它运动的路径必须至少是 C^0 连续的。但是，如果路径也可以满足 C^2 连续路径的需求，效果就会更加真实。

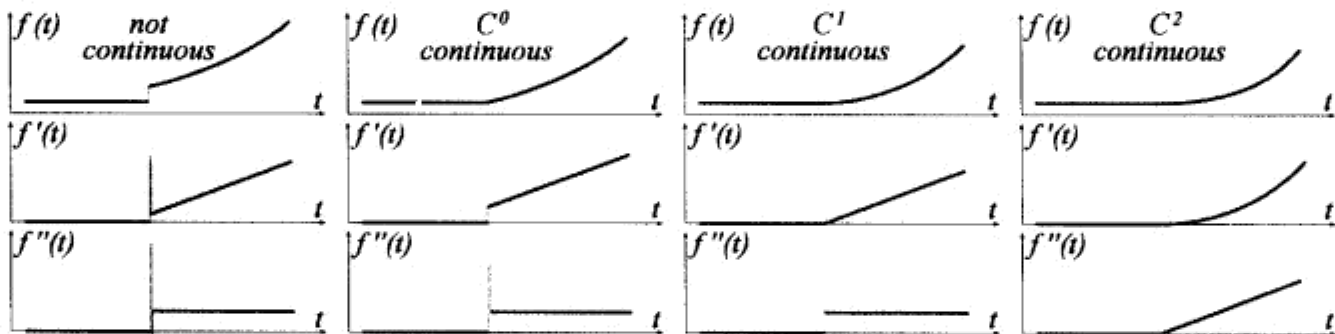


图 4.8.1　带有不连续性的曲线，C^0 连续性、C^1 连续性和 C^2 连续性。最上面一行是初始时间函数。第二行和第三行是这些时间函数的一阶导函数和二阶导函数

4.8.2　我们得到的是什么？偶尔不够真实的运动

今天，市场上存在着数量巨大的 3D 游戏。其中一些游戏将玩家带入到现实的世界，而有些游戏则将玩家置身于虚幻之地。在一些游戏和模拟器中，我们需要遵守交通规则；但在另外一些游戏中，我们则可以横冲直撞，不计后果。但是，不管是哪种情况，我们通常都希望虚拟汽车的运动要比较真实。

即使是那些提供驾驶舱视角的赛车游戏，它们也经常会提供一个赛事重放模式。在这个模式中，我们可以从一个外部视角回顾整个赛程的细节信息。除了内部视角，我们还可以通过固定在赛道旁边的摄像机观看赛事。假设你是一位摄像师，你的工作就是用手中的摄像机跟踪拍摄一辆赛车。摄像机被固定在一个三脚架上，你可以围绕垂直轴（*Y*）和水平轴（*X*）转动摄像机。在这种情况下，可以说摄眼的位置是固定的，而目标点则是运动着的。我们可以把这两个点定义为两个三维向量。同时，还需要第三个向量，我们称之为 ***up***（向上）向量。这个向量定义了摄像机缺省的垂直轴向，如图 4.8.2 所示。在缺省情况下，可以让摄像机作俯仰运动和偏转运动。如果还想让摄像机翻滚，就需要改变假设的这个 up 向量，并使用其他的

公式，例如欧拉角（Euler angle），如图 4.8.3 所示。

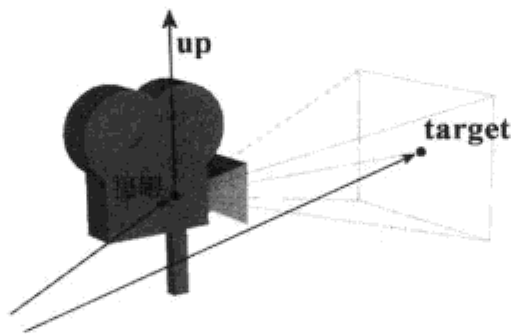


图 4.8.2 虚拟摄像机及其重要的几个向量。摄眼（eye）定义了视点，目标点定义了方向（目标点将位于显示屏幕的中央位置），而 up 向量定义了摄像机的垂直轴向

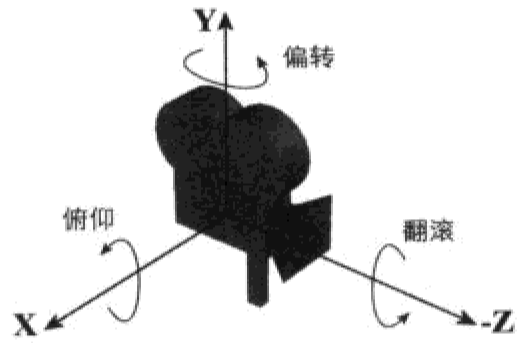


图 4.8.3 物体可以分别围绕这 3 个轴向进行旋转：俯仰旋转（X 轴）、偏转旋转（Y 轴）以及翻滚（Z 轴）。欧拉角定义了这些旋转

现在，你负责摄像的虚拟赛车比赛已经开始了，赛车正向摄像机视点的方向开过来，如图 4.8.4 所示。跟拍的赛车已经出现在地平线上，你需要将它作为摄像机的目标点，一直跟踪它。这辆赛车的速度已经非常快了，但是车手仍然在不断加速。这时候，你已经知道车手肯定会加速，因为你负责的这段赛道是一个很长的直道，所以你就在这儿等着他过来。终于，赛车马上就要开过来了。但是，一个巨大的石头怪兽突然出现在赛车前方！这个怪兽太过蠢笨，无法闪避，而车手也没有足够的时间去踩刹车来避免撞上怪兽。咣，最终还是撞上了！！！怪兽似乎没什么感觉，但是赛车却被撞成碎片，散落在赛道上。由于摄像机的质量和意外的事件，摄像机还在转动着。等你意识到有事情发生了，才将摄像机转回到事故地点。

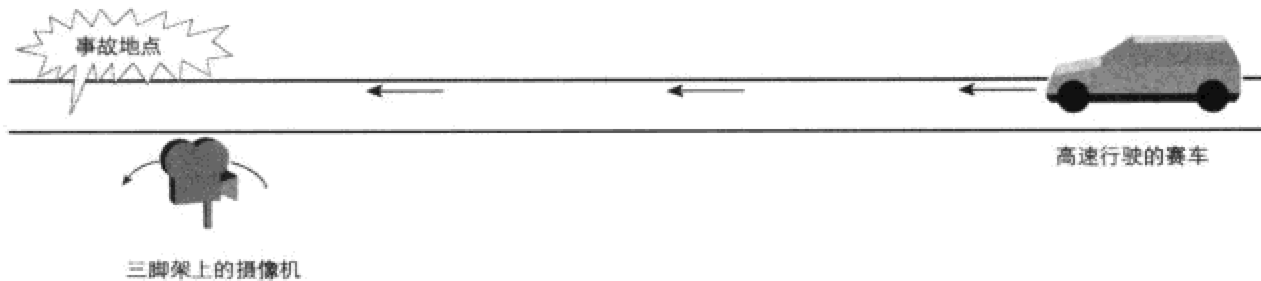


图 4.8.4 赛道旁边的一台摄像机正试图跟拍一辆赛车，但赛车却突然停了下来

4.8.3 考察摄像机的控制

正如在前面讨论的，我们希望将摄像机的运动路径定义为一个 C^2 曲线。在一个赛车游戏中，无论玩家怎样驾驶赛车，赛道旁边的摄像机总是能跟拍到玩家的赛车。大家都知道，计算机游戏是针对一系列离散的时间点来计算游戏物理和图形图像的。因此，必须把运动着的物体和摄像机的路径当作采样曲线来处理。但是，我们该如何根据离散采样来定义一条 C^2 连续曲线呢？

1. 参数化的曲线

根据给定物体的当前位置的变化来定义参数化的曲线，是否可行呢？这个解决方案会带来以下几个问题。正如前面提到的，被摄像机跟踪的物体（例如赛车）可能会出现不切实际的运动，但我们还得使用它的采样路径来创建摄像机的 C^2 连续路径。一个大问题就是突然出现的意外运动会产生非连续的跳跃值，使我们无法得到一个 C^2 曲线。而且，如果不能未卜先

知，我们怎么可能根据玩家实时控制的物体运动轨迹的采样点来生成 C^2 曲线呢？在曲线上增加新的点，曲线的整个形状就会发生改变。我们或许可以将摄像机的路径分解成若干个更小的曲线，但在曲线上增加新的点的问题依然存在。我们只好抛弃这个数学方法，从另外的角度考虑该问题。

2. 观察人类的行为

回想一下前面那个故事中的摄像机的运动情况。当然了，摄像机是有质量和惯性的，但是虚构的摄像机的行为的最重要来源是人为控制。实际上，从这里开始，我们就把真实的摄像机运动称之为“人为控制的摄像机”。如果摄像机比较大比较沉，那么根据牛顿第二定律，它的运动应该会越加平稳。也就是说，即使用来加速摄像机的作用力比较有限，到达给定的速度也只是一个时间问题。所以，移动或旋转一个笨重的摄像机和移动一个手持摄像机一样快，只是运动的细节要少一些而已。物体运动的“平稳性”意味着我们可以跟拍缓慢的物体运动，也可以跟拍快速的物体运动（偶尔会有些延迟），但是我们不能跟拍突然的变向。“频率”是物体运动中的一个典型参数，可以用曲线来描述。这样一来，我们就可以用一个频率极限来表示运动的平稳度。这个极限值可以通过某种低通滤波方法获得。

所谓“人类的行为”，不仅意味着要让摄像机平稳地运动，还要考虑摄像师的思维模式。首先，他会在取景器中发现某个运动中的物体，然后开始跟踪拍摄。然后，他逐渐掌握该物体的运动速度，但仍然继续跟拍。如果物体开始加速，摄像师也会注意到这一点，并试图将物体继续保持在屏幕中央。如果物体出现突然性的运动，摄像机肯定会拍过。这种情况可以用两个属性来解释：一是“思维模式”，因为摄像师想要做得更机灵些，但意外事件发生后，他还是有一些延迟；二是“平稳”，因为摄像师的身体和摄像机本身都是有质量的，而且摄像师的肌肉能使的力气也是有限的。

实际上，如果我们创建一个平滑的路径，使之具有人类的预测能力，可以预测路径未来的形态，我们就可以近似得到一条足够完整的 C^2 连续曲线。

4.8.4 终极决策：实现人类的行为

我们已经看到数学方法是非常正确，也是非常科学的。但是，数学方法不但需要花很多时间去解释、理解，而且实现起来也颇费工夫。除此之外，我们一直认为，用人类行为的理论来表示这个有待求解的问题是一种更好的解决方案。而且，它实现起来也不是很困难。所以，现在就让我们找出合适的算法，实现这两个主要的人类行为：平稳和思维。

1. 利用惯性使曲线更平滑

正如前面所提到的，惯性的主要作用就是使物体运动的路径更加平滑。那如何将惯性应用到离散的采样点上呢？如何使它们更加平滑，或者如何对它们使用低通滤波呢？如果对数字信号处理技术非常熟悉的话，就会明白，由于 Shannon 的采样理论也称为 Nyquist criterion，奈奎斯特准则)，一个信号波形存储的信号分量最多只能达到采样频率的一半。例如，每秒采样 120 次，我们最多只能存储频率为 60 Hz 的声音信号。

我们保持初始的采样频率不变，取得波形信号中两个相邻采样点的平均值，如图 4.8.5

所示。这个过程类似于对波形进行降频采样，每秒采样 60 次，最大频率被限定为 30 Hz。如果我们计算的是相邻的 3 个采样点的平均值，那么最大频率就变成了 20 Hz，依次类推。

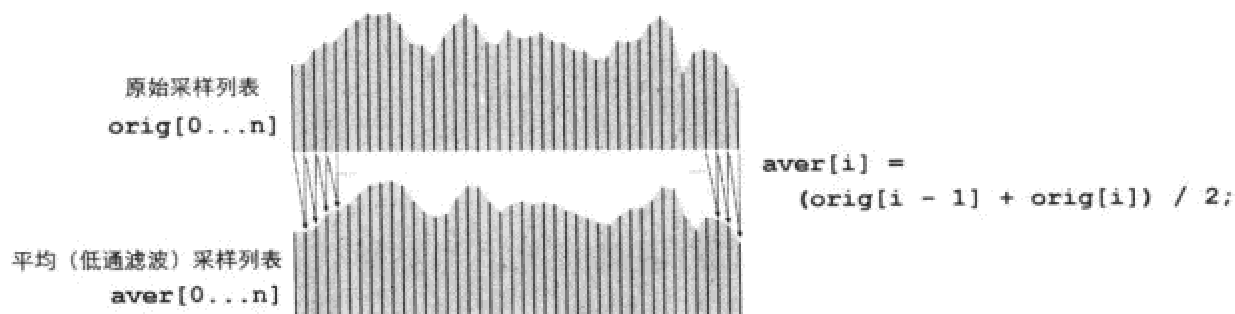


图 4.8.5 取得采样列表中相邻 2 个采样点的平均值

当然了，在摄像机的控制中，我们要逐帧处理一个接一个的三维位置向量。前面那个例子中的采样频率对应的就是物体运动控制中的帧率。首先，我们需要一个列表来保存目标物体的位置向量。另外还需要一个列表，以便计算输入位置列表的低通滤波值。这个平滑过程是通向人为控制摄像机的第一步。

2. 思维：采用外插法处理位置采样值

还记得摄像师的行为吗？你试图预测目标物体的运动方向，以及物体运动的速度。在内插算法的处理步骤中，我们得到了 2 个或更多的关键采样点，并尝试找到它们之间的位置上应该都是些什么值，如图 4.8.6（a）所示。在外插算法中，我们同样有 2 个或更多的采样点，但现在需要知道的是下一个采样点，如图 4.8.6（b）所示。参考已知的位置采样值之间的“区别（A）”，我们就可以预测物体下一个新的位置。而根据这些“区别（A）”之间的“区别（B）”，我们就可以预测出物体移动的速度。通过这个加速度，对之前预测得到的位置信息进行修正，就可以得到一个更新的、更合理的位置信息。另外，如果再去分析“区别（B）”之间的区别，我们还可以预测物体运动的加速度。根据这个加速度，我们反过来可以修正之前预测得到的物体的速度；再利用这个修正的速度，又可以重新修正之前预测的物体的位置信息。但是，如果只有少量的采样点，预测得到的结果就会不太精确。为了解决这个问题，可以考虑进行更多的采样。我们还可以给采样值赋予一定的权重，来定义它们的重要程度。例如，新近的采样值应该比较早采样值的权重要高一些。

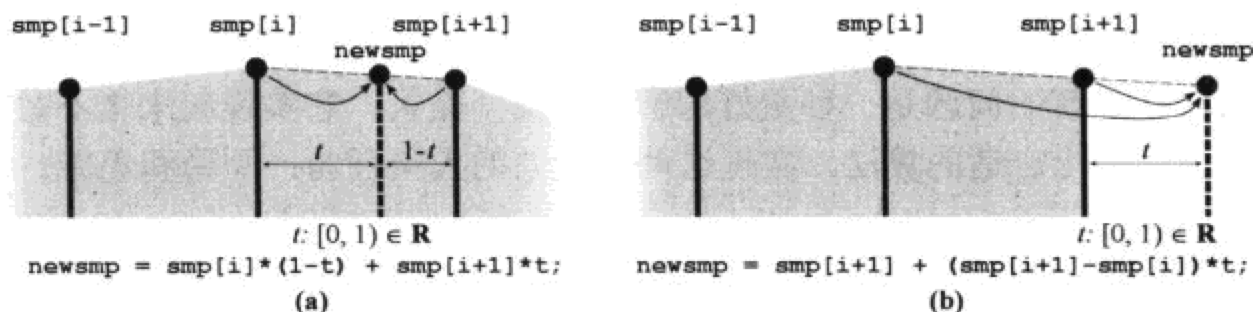


图 4.8.6 内插算法（a）和外插算法（b）的采样点

通过选择外插法的采样长度，我们可以定义出虚拟摄像师对物体运动的“警觉性”。前面提到至少需要 2 个采样点，才能利用外插法计算出目标物体的下一个位置。在下一个运行时刻（比如下一帧），马上就会用到这个新预测出的物体位置。在这种情况下，我们就可以说虚拟摄像师对物体的运动非常警觉，动作也很麻利，因为被跟拍物体运动中每一个微小的突

变都在下一帧中有所反应。如果帧率太高，例如每秒 120 帧，对人类而言，这个速度就有点过快了。如果真是这样，那就意味着摄像师要在 8.3 毫秒内觉察到物体运动的变化。而一般来讲，人类可以在 0.7 秒内觉察到突然的变化，这就会造成反应动作的延迟。为了进行时间上的补偿，对被跟拍物体当前位置的预测要提前 0.7 秒做出。为此，保存外插位置信息（extrapolated position）的采样缓冲区就要从最近的一个 0.7 秒开始保存采样值。在固定帧率下，这个问题很容易解决。但是，计算机游戏的帧率总是取决于被渲染场景的复杂度。因此，我们必须使用一个可变长度的采样缓冲区，来保存外插位置信息。

4.8.5　编程中涉及的问题

本文的虚拟摄像机模型是用 C++语言实现的，并使用了 Win32 环境下的 OpenGL 和 GLUT 图形支持。本节将向大家介绍这个虚拟摄像机模型，以及其他辅助数据结构。这个模型及这些数据结构都可以很容易地应用在 OpenGL 中。

1. OpenGL 中的摄像机

首先看一下如何在 OpenGL 中使用 GLUT 定义透视图（perspective view）。我们使用了两个函数来定义视图。第一个函数是

```
gluPerspective(fov, width/height, nearDist, farDist);
```

其中，`fov`（field of view，视场）是摄像机的垂直视角，单位为“度”。如果使用长焦镜头的话，这个值会比较小；如果使用广角镜头的话，这个值会比较大。如果还想仿真摄像机的变焦，就必须时不时地调整这个变量。`width/height`（宽/高）的值定义了显示区域的纵横比，这和避免显示器屏幕失真应该是一个道理。最后两个参数 `nearDist` 和 `farDist`，定义了摄眼（camera's eye）到近端裁剪平面和远端裁剪平面的距离。这两个裁剪平面之间的场景则是我们要拍摄的场景。

定义摄像机的基本函数是：

```
gluLookAt(eye.X(), eye.Y(), eye.Z(),
          target.X(), target.Y(), target.Z(),
          up.X(), up.Y(), up.Z());
```

通过这个函数调用，我们可以定义摄眼（camera's eye）的当前位置、目标点（target），以及摄像机的 up 向量。在计算机游戏里，场景总是在变化之中。如果摄像机也发生了变化，我们就要在每一帧中重新计算上述两个函数中的所有变量。

2. 可变的视场和自动变焦

我们的摄像机模型具备了自动变焦的功能。它可以根据被跟拍物体的大小，以及该物体与摄像机之间的距离，自动地在每一个时间步长中计算 `fov`：

```
fov = 2.0 *
      atan((targetSize / 2.0) / (target - eye).Length()) *
      180.0 / M_PI;
```

在这个程序片段中可以注意到，我们使用的是摄眼到目标点（target）的距离，而不是摄眼到物体本身的距离。为什么这么做呢，其主要的依据是摄像机的目标点总是代表着被跟拍物体的直接（*direct*）或间接（*indirect*）位置。在经过低通滤波和外插操作后，目标点就会偏离物体的当前位置，这就是我们上面说的“间接位置”。

3. 人类行为的实现

接下来要讨论的，是如何在这个摄像机模型中实现人为摄像机运动模型。Camera 类有各种不同的数据成员和成员函数。这里将介绍其中最为重要的几个。

我们在头文件 camera.h 中定义了 `Camera` 类。在头文件 camera.h 开始的部分中有一些全局常量，其中的一个常量定义了数组的最大长度，在摄像机目标点的外插操作中会用到数组。该常量的名字是 `MAXEXTRAPOLLENGTH`，我们使用的值是 1000。它的值应该大于计算机中场景渲染的最大帧率，这是因为用这个长度定义的列表保存的是 1 秒钟内从每一帧中搜集的向量。

Camera 类中另外一个非常重要的数据成员是 `Vector3f` 类型的变量，它定义了摄像机本身(`eye`、`target` 及 `up` 向量)。`Vector3f` 类型的数组(`objPosList`、`objPosDifList`、`objPosDifDifListlpfObjPosList`）和浮点类型的数组（`objFrameWeightList` 和 `sumWeight`），都是在外插算法中要用到的。第一个浮点类型的数组中保存的是权值（weight)，定义了数组 `objPosDifList` 和 `objPosDifDifList` 中每个数据相应的重要程度；第二个浮点数组保存的是预先计算好的、每个数据对应的权值的总和，会在计算加权平均值的时候使用。

在这些数组中，数据的保存是依照时间顺序进行的。索引 0 所指向的数据就是当前这个帧的数据，而索引 *n* 所指的数据就是 *n* 帧之前的数据。

下面马上就可以看到这个虚拟摄像机所具备的人类行为的精华部分。函数 `placeObjectHereToFollow()` 取得被跟拍物体的当前位置（`objPos`），并用一个布尔变量（`lookAtItExactly`）告诉目标点应该直接跟踪物体，还是进行人为控制的跟拍。

```
void placeObjectHereToFollow(Vector3f objPos,
                              bool lookAtItExactly = false) {
    if (actualFPS == 0.0) return; }
```

这个函数在开始的时候有一个控制语句。如果当前的帧率为 0，那就会跳过这个函数的执行，以免出错。一般在应用程序刚启动时的一小段时间之内，会出现帧率为 0 的情况。

接下来，要定义那些和时间有关的参数，这些参数都是通过当前的帧率（`actualFPS`）计算而来的。浮点变量 `lengthForExtrpl` 和 `lengthForLPF`，以及它们各自的整型（integer）版本变量（分别被称为 `iLengthForExtrpl` 和 `iLengthForLPF`）表示的是外插算法或低通滤波中使用的采样值的数量。换句话说，它们表示的是外插算法的时间常量（在这个例子中，它的值是 0.7 秒)，以及低通滤波的时间常量（0.4 秒，也就是低通滤波的极限频率 2.5 Hz）。

```
float lengthForExtrpl =
      min(actualFPS, (float)MAXEXTRAPOLLENGTH) * 0.7;
float lengthForLPF =
      min(actualFPS, (float)MAXEXTRAPOLLENGTH) * 0.4;
```

在下面的代码片段中，我们要计算下一个低通滤波的采样值。这里要用到的数据是那些保存在 objPosList 数组中被跟拍物体的原始采样值。

```
Vector3f lpfObjectLastPos = objPos;
for (int i = 0; i < (iLengthForLPF - 1); i++)
    lpfObjectLastPos += objPosList[i];
if (iLengthForLPF != 0)
    lpfObjectLastPos *= (1 / (float)iLengthForLPF);
else lpfObjectLastPos = objPos;
```

无论发生什么情况，我们必须逐帧地进行 4 个数组的移位操作，执行范围是这 4 个数组的整体长度（MAXEXTRAPOLLENGTH）。

```
for (int i = MAXEXTRAPOLLENGTH - 1; i > 0; i-)
{
    objPosList[i] = objPosList[i - 1];
    lpfObjPosList[i] = lpfObjPosList[i - 1];
    objPosDifList[i] = objPosDifList[i - 1];
    objPosDifDifList[i] = objPosDifDifList[i - 1];
}
```

移位操作完成之后，我们要在它们各自的 0 位上插入一个新的值。要注意的是，对于保存速度数据的列表，它的新值就是 LPF（低通滤波）采样值之间的差（也就是我们前面提到的采样值之间的“区别”）。

```
objPosList[0] = objPos;
lpfObjPosList[0] = lpfObjectLastPos;
objPosDifList[0] = lpfObjPosList[0] - lpfObjPosList[1];
objPosDifDifList[0] = objPosDifList[0] - objPosDifList[1];
```

下面，要计算速度数据和加速度数据的加权平均值。

```
Vector3f averageTargetStep;
Vector3f averagePosDifDif;
for (int i = 0; i < iLengthForExtrpl - 1; i++)
{
    averageTargetStep +=
        (objPosDifList[i] * objFrameWeightList[i]);
    averagePosDifDif +=
        (objPosDifDifList[i] *
        objFrameWeightList[iLengthForExtrpl - i - 1]);
}
averageTargetStep *= (1 / sumWeight[iLengthForExtrpl - 1]);
averagePosDifDif *= (1 / sumWeight[iLengthForExtrpl - 1];
```

如果想让摄像机直接跟拍物体，在这个函数中就不用做别的工作了，只要将目标点（target）“锁定”到被跟拍物体上即可（target = objPos）。否则，我们就将低通滤波处理过的位置列表（数组 lpfObjPosList）中的[iLengthForExtrpl]位置上的值（外插算法经过 0.7 秒的操作），加上平均速度（averageTargetStep）与外插算法的时间常量（lengthForExtrpl）和低通滤波时间常量的一半（lengthForLPF/2）的乘积，作为最后的修正，并在此基础上再加

上平均加速度（averagePosDifDif）的值，最后的总和就作为 target 的值。

```
    if (lookAtItExactly) { target = objPos; return; }

    target = lpfObjPosList[iLengthForExtrpl] +
                averageTargetStep *
            (lengthForExtrpl + lengthForLPF / 2.0) +
            averagePosDifDif;
}
```

或许有人会问，上面的计算中，为什么如此准确地用到了 LPF（低通滤波）时间常量的一半呢？答案很简单：在生成低通滤波位置的数据时，我们用的是两个相邻采样点的平均值，参见图 4.8.5。让我们针对一个直线运动的、移动速度恒定的物体，检验一下用虚拟摄像机的目标点（target）如何跟拍这个物体。首先，我们取得 iLengthForLPF 个相邻的位置采样值，计算它们的平均值。这样，就得到了这些采样值的算术平均值。对于这样的物体运动，向量的算术平均值就是这些采样向量定义的区间的中心点。从时间上讲，该中心点也是该区间在时间上的中心点。在这样的物体运动中，会产生数组 lpfObjPosList 中的数据。如果取得数组 lpfObjPosList 的第一个值（lpfObjPosList[0]），在到达这个值所指的位置时，已经过去的时间就是（lengthForLPF/2)。因此，延迟时间就是（lengthForLPF/2)。如果取得数组 lpfObjPosList 中 iLengthForExtrpl 位置上的数据（lpfObjPosList[iLengthForExtrpl]），总的延迟时间就是（lengthForExtrpl+lengthForLPF/2）。这个时间量再乘以平均速度（averageTargetStep），就得到了被跟拍物体的当前位置。当然了，这个物体是直线运动、速度恒定的。这就是为什么该算法可以跟踪直线运动区间中的物体。对于其他类型的物体运动，摄像机跟拍的行为则会非常的人性化。

4.8.6 总结

本文描述了一个可以为虚拟摄像机增加人性化控制的算法。这个算法实现起来非常简单易懂。它模拟了摄像师克服惯性的作用，一直跟拍虚拟赛车的运动轨迹的行为。这个算法的性能与帧率或者赛车的速度无关。该算法的主要优点是它所需要的处理时间非常少，所以它不会明显地降低程序性能。

4.8.7 关于演示程序

ON THE CD

我们用一个简单的演示程序实现了这个算法，在随书光盘中可以找到这个演示程序。演示程序中有一个三维场景中的汽车，三脚架上有一台摄像机在对它进行跟拍。使用键盘上的方向键可以控制汽车的移动。如果让一个虚拟摄像师操控摄像机，它就会一直跟拍这个汽车。Q 键可以将虚拟摄像机切换到人为控制模式。

第 5 章

图形图像

引 言

ATI 公司，Jason L. Mitchell
JasonM@ati.com

GPU 的处理能力正在飞速发展，其发展速率远远超过了支撑它的其他配套系统的能力，在内容创作方面不断增加的费用投入就更不用说了。未来几年，通过实例化和程序主义，大家会看到数据扩张的趋势。因此，在《游戏编程精粹》丛书的第五卷中，我们选编的图形图像文章，其内容主要集中在数据扩张和自然环境这两个方面。

在第一篇文章《在现代 GPU 上渲染逼真的云彩》中，来自 UBISOFT（育碧公司）的 Jean François Dubé 利用先进的像素着色器和预先计算好的噪音纹理（noise texture），生成了带有动态光照的程序化云彩。在文章《下雪吧，下雪吧，下雪吧（下雨吧）》中，Niniane Wang 和 Bretton Wade 描述了他们开发的雨雪渲染技术。这一技术是他们为游戏《微软飞行模拟器 2004：飞行的世纪》（*Microsoft Flight Simulator 2004：A Century of Flight*）开发的。这项技术不但高效，而且是可控的，可以让游戏美工人员调整模型，以满足特定游戏场景的需求。

接着，我们从天上来到地面，把注意力转移到高效、逼真的植物渲染上。在文章《widgets：快速渲染和持久化小物体》中，Martin Brownlow 讲解了利用当前的图形硬件产品，高效率地渲染出大量实体化的物体作为地被植物的技术。这篇文章充分利用了 GPU 和 CPU 技术来维系最佳的渲染性能。在文章《逼真的树木和森林的 2.5 维替用物》中，Gábor Szijártó 和大家探讨了一个使用动态景深精灵渲染出逼真的林冠的技术。

一个带有用程序实现的天空和植被的交互式虚拟世界也许是美丽的，但游戏不可能没有任何破坏性场面。为此，本章的最后提供了几篇颇有“破坏力”的文章。在《无栅格的可控火焰》一文中，Neeharika Adabala 和 Charles E. Hughes 为我们介绍了一个可以模拟和渲染出非常逼真的火焰的技术。有些时候，人们可能想得到某些更富戏剧性、更夸张的好莱坞式的特效。在《使用公告牌粒子构建强大的爆炸效果》一文中，任天堂公司的 Steve Rabin 提出的技术可以制造出强大的、夸张的爆炸特效。这样的特效不但更加逼真、可控，而且渲染效率非常高。

在《游戏编程精粹》系列书中，我们第一次收录了一篇有关宝石渲染的文章！ATI Research 公司的 Thorsten Scheuermann 的文章《渲染宝石的简单方法》，向我们展示了一个渲染宝石的高效技术。这是 ATI Research 公司的演示程序“*Ruby：The DoubleCross*”使用的一项技术。接下来，Dominic

Filion 和 Sylvain Boissé在他们的文章“体积化的后期处理”中，展示了一个可以将逼真的光折射特效和热气特效集成到 3D 场景中的新技术。

正如开始所说的，随着 GPU 对大数据集处理能力的不断提高，内容创作的成本投入也会不断增加。在《过程式关卡生成》一文中，Timothy Roden 和 Ian Parberry 提出了一个程序化生成游戏世界的框架，它极大地减轻了创建游戏关卡分级工作的沉闷无聊。为了能够高效率地渲染日益复杂的游戏世界，我们必须生成大量的着色器。在本章的最后一篇文章《重组 schader》中，Dominic Filion 为图形图像程序员们展现了一个特别的技术，该技术可以高效地管理数量庞大的着色器。

一种理想的技术可以快速、容易地应用到人们手头上的游戏编程项目中，而不需要做大量的重写工作。因此，我们选择的内容都是为了满足此需求，并重点突出数据扩张和自然环境这两个日益重要的主题。

5.1 在现代 GPU 上渲染逼真的云彩

育碧公司，Jean-François Dubé
jfdube@ubisoft.qc.ca

随着新一代渲染硬件的发展，游戏正变得越来越逼真。逐像素光照和阴影特效，以及体积式光照符合潮流一些光照和大气特性，现在都变成了可能。尽管已经取得了这么多进步，但如果仔细观察任何一个现代的游戏产品，就会注意到这些游戏有一个共同点：静态的（而且有时是非常难看的）贴图天空。有一些较先进的解决方法是添加一个移动的云层，这虽然小有帮助，但还是不像我们想象得那么逼真。在这篇文章中，我们向大家介绍一个新的方法，即如何利用 Shader Model 3.0 在 GPU 上渲染出有光照效果的、逼真的动态云彩。

在本文中，我们会了解如何在 GPU 上程序化地生成一个动画噪音纹理，其效果很像真实的云彩。然后，我们把这个噪音纹理贴图应用到摄像机上方的一个类似于天空盒的平面上。最后，我们使用 PS 3.0 渲染器，利用循环让光线穿透云层，获得逼真的光照特效。渲染效果如图 5.1.1 所示。

图 5.1.1 逼真的实时程序化云彩

5.1.1 制造噪音

云彩渲染背后的基本思想，即是不时地生成噪音动画，让云彩形态的改变看上去似是而非。因为所有的计算都是在 GPU 上实现的，所以现有的 3D Perlin Noise 算法实现无法很好地适应我们的任务。取而代之的，是以不同的倍频和权重，合成多个预先计算好的噪音纹理。每个倍频的噪音都会向最后的纹理添加一些细节。第一个倍频噪音给出粗略的形态，随着我

们使用更多的倍频噪音，更多的细节被添加进来。在实际应用中我们发现，使用 8 个倍频的噪音能够在性能和视觉质量上获得很好的平衡。

1．倍频纹理合成

每个噪音倍频是由一个平滑的 128×128 的噪音纹理来表示的。这个噪音是用随机数生成的。平滑滤波采用的是简单的邻域过滤方法。由于噪音有正负之分，所以根据使用的纹理格式，必须做一些放大和偏置的工作，将这些数据打包到纹理中。而且，当在像素着色器中对它进行抽样时，同样也需要做这些工作。在像素着色器中，所有的 8 个倍频会以不同的放大率和权重进行合成，以达到预期的效果。也就是说，对于第一个倍频不会做任何操作，以便给出最后复合的噪音纹理的粗略模样。随着我们继续添加更高频率的倍频，它们重复的次数也越来越频繁，但权重在不断地变小。这样就会为噪音纹理添加越来越多的精美细节，如图 5.1.2 所示。我们可以控制覆盖率和权重因子，以便调整云彩的外观。如果给那些更高频率的倍频纹理更大的权重，只能形成比较小的云彩合成体，但是其形状与较低频率的倍频纹理定义的形状基本一致。对于倍频放大系数，比较好的初始值是 1、2、4、8 等，依次类推；而对于倍频权重，比较好的初始值是 1、1/2、1/4、1/8 等，依次类推。

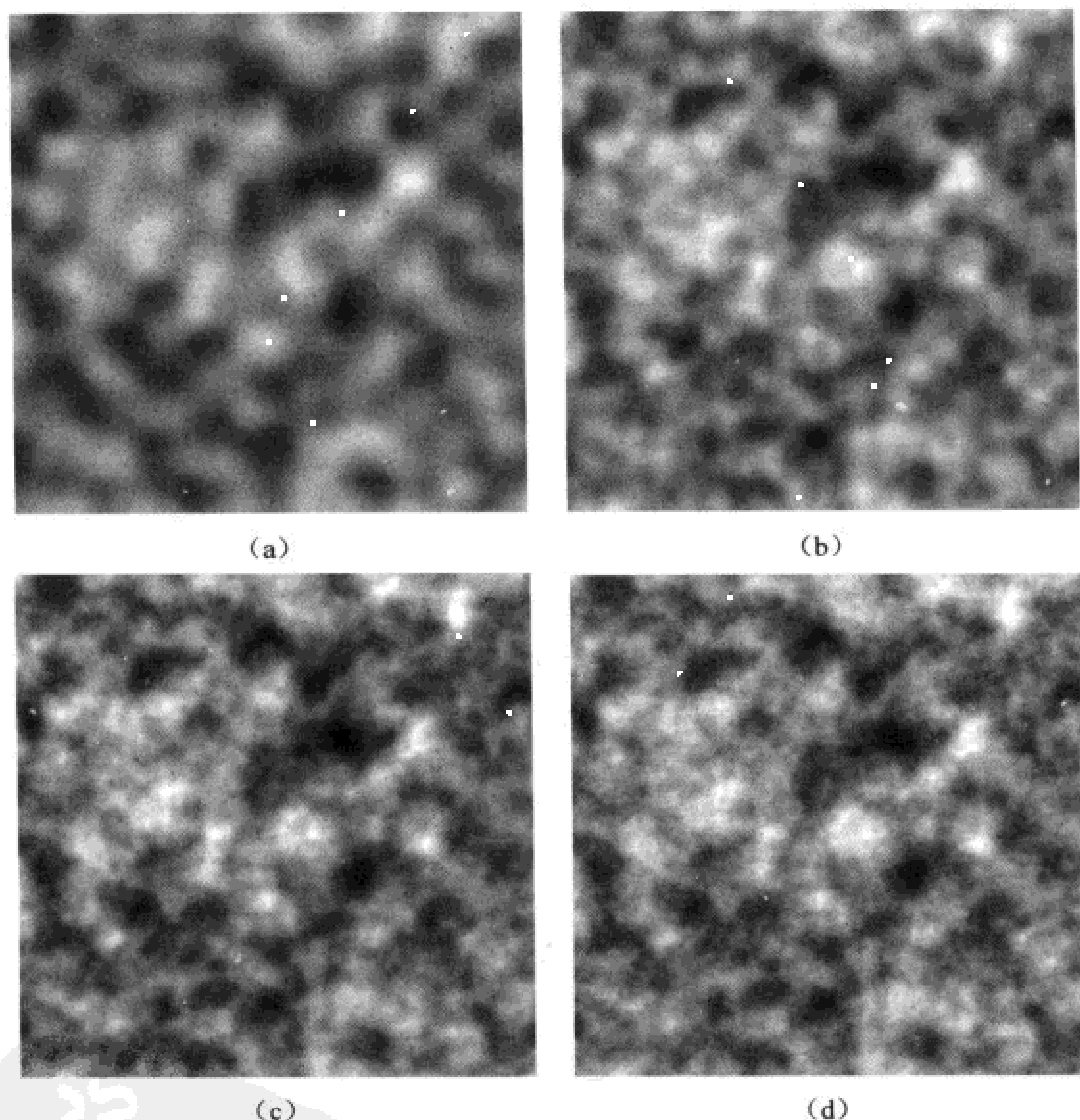


图 5.1.2　(a) 第 1 个倍频纹理给出了云彩的大致形态。(b) 在此基础上，第 2 个倍频纹理添加了一些细节。(c) 第 3 个倍频纹理添加了更精美的细节。(d) 第 4 个倍频纹理及其他更高频率的倍频纹理添加了越来越多但越来越微妙的细节

2．噪音纹理动画

噪音纹理动画就是简单地随着时间的推进，为噪音纹理添加动画效果，这在[Elias]和[Pallister01]

中有详细介绍。低频纹理的动画相对于高频纹理的动画要慢一些。这样，云彩的基本形状就会慢慢地有所变化，而较精致的细节会变化得快一些。对于每一个倍频，要保留 2 个噪音纹理，然后以不同的速率缓慢地在二者之间进行插值。在后面讨论优化的内容时，还会回到这个步骤。

5.1.2 云彩的密度

到目前为止，我们已经得到了组成模拟云彩的最基本构建块：倍频可控的动态噪音纹理。一个典型的倍频噪音纹理图片如图 5.1.3（a）所示。现在必须执行另外一些操作，以便让噪音纹理看上去更像云彩。例如，我们需要控制天空中云彩的百分比，也就是天空中的云量。为此，我们要从噪音纹理中减去一个值（我们称之为“云量”），并将计算结果进行 clamp（夹紧）处理（处理超出范围的结果：如果结果>1，则将结果置为 1；如果结果<0，就将结果置为 0），这样就可以去除一定数量的噪音，如图 5.1.3（b）所示。云量值的有效范围是[0..1]，代表着我们想要看到的蓝天的数量（0 代表满天乌云，1 则代表万里无云）。到了这一步，我们仍然没有完全得到想要的云彩。所以，我们要对当前的值进行取幂操作，以获得想要的蓬松的云彩效果，如图 5.1.3（c）所示。

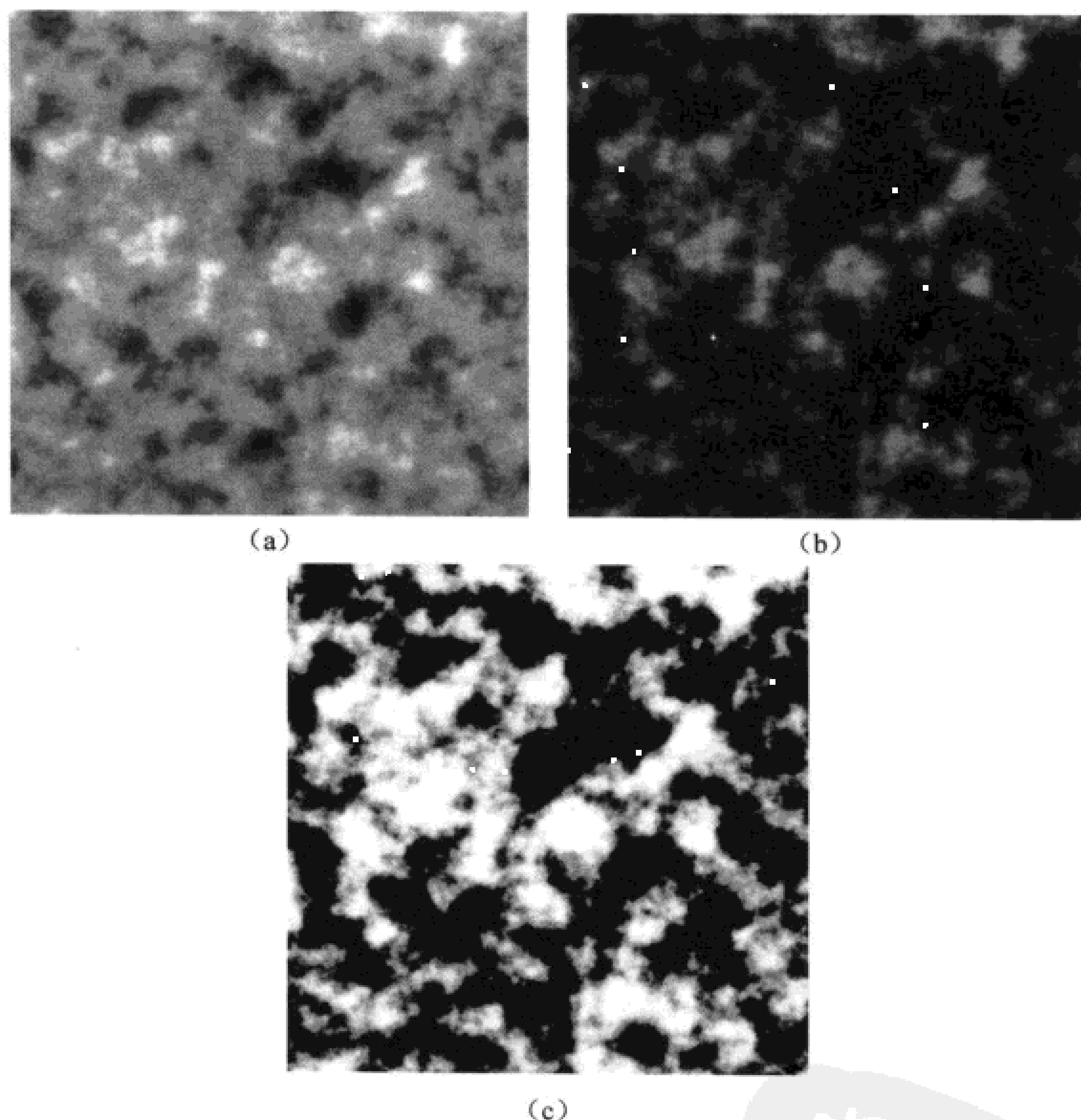


图 5.1.3 （a）8 倍频的噪音纹理。（b）同一个噪音纹理在减去云量值，并对结果进行 clamp 处理之后的效果。（c）同一个噪音纹理进行取幂操作之后的效果。现在看上去有点像云彩了

下列 HLSL 代码执行的是前面描述的计算工作（这个着色器的其他代码会在后面给出）。

```
01. float cloud_cover = 0.45f;
02. float clouds_sharpness = 0.94f;
```

PDG

```
03. float3 tex = tex2D(Octave0, uv * OctavesScales0.x) * OctavesWeights0.x;
04. tex += tex2D(Octave1, uv * OctavesScales0.y) * OctavesWeights0.y;
05. tex += tex2D(Octave2, uv * OctavesScales0.z) * OctavesWeights0.z;
06. tex += tex2D(Octave3, uv * OctavesScales0.w) * OctavesWeights0.w;
07. tex += tex2D(Octave4, uv * OctavesScales1.x) * OctavesWeights1.x;
08. tex += tex2D(Octave5, uv * OctavesScales1.y) * OctavesWeights1.y;
09. tex += tex2D(Octave6, uv * OctavesScales1.z) * OctavesWeights1.z;
10. tex += tex2D(Octave7, uv * OctavesScales1.w) * OctavesWeights1.w;
11. tex = max(tex * 0.5f + 0.5f - cloud_cover, 0.0f);
12. tex = 1.0f - pow(clouds_sharpness, tex * 255.0f);
```

第 3 行到第 10 行执行的是 8 个倍频噪音纹理的合成。uv 的值包含着云彩平面的纹理坐标。在采样之前，这个纹理坐标要进行放大，从噪音纹理中提取的值也要进行加权处理。这个着色器会假设噪音纹理是保存在一个有正负之分的纹理格式中，因此 tex 的值也是有正负之分的。我们在第 11 行中将噪音纹理的范围从[-1..1]转换为[0..1]，然后再减去云量（cloud_cover）的值，并对结果进行 clamp 处理。第 12 行的代码执行的是取幂操作。

至此，我们使用相当传统的方法计算出了似是而非的实时云彩密度。但是，为了能够将得到的云彩动画特效完整地集成到游戏世界，我们还需要对它们进行逼真的光照处理。

5.1.3 云彩的光照处理

云彩的实时光照处理是非常复杂的，光线散射特效的表现也很难实时地进行评估。即使是最复杂的解决办法也仍然会留下很多遗憾（[Harris01]，[Harris02]）。因此，我们决定忽略物理校正计算，并尽量逼近想要的效果。

1. 光的散射

在现实生活中，云彩接收来自太阳的光线。我们在地球上看到的是穿透云层的光线。由于光线要穿过云体，一旦碰到水珠粒子，光线就会向各个方向散射。其结果就是造成云体底部有暗灰色的部分。理想情况下，对于我们在云彩平面上渲染的每一个像素，都要追踪一条从太阳射向该像素的光线，并累积计算光线在云彩中行进路线的距离，以便用积分近似法来近似模拟光线散射的特效。这个积分近似法的实时运算是很昂贵的，所以我们提出了近似算法，这样就可以将其应用在游戏中。

2. 云彩密度域追踪

为了追踪一条从太阳射向云彩底部给定像素的光线，我们使用了前面合成的噪音纹理，把它作为一个和高度域非常类似的密度域。为了减少锯齿，我们只使用前面的 4 个倍频噪音纹理来执行云彩的光照操作。在每一帧的开始部分，我们把云彩密度前面的 4 个倍频纹理渲染成一个 512×512 的渲染目标体。在最终的着色器中，这个渲染目标被用作密度域。下面的 HLSL 代码执行的是路径追踪。这段代码紧接着前面的代码，其中 tex.r 是当前的云彩密度。

```
01. float Density = 0.0f;
02. float3 EndTracePos = float3(uv, -tex.r);
03. float3 TraceDir = EndTracePos - SunPos;
```

```
04. TraceDir = normalize(TraceDir);
05. float3 CurTracePos = SunPos + TraceDir * 1.25f;
06. tex = 1.0f - pow(clouds_sharpness, tex * 255.0f);
07. TraceDir *= 2.0f;
08. for(int i=0; i<64; i++)
09. {
10.    CurTracePos += TraceDir;
11.    float4 tex2 = tex2D(DensityFieldTexture, CurTracePos.xy) * 255.0f;
12.    Density += 0.1f * step(CurTracePos.z*2, tex2.r*2);
13. }
14. float Light = 1.0f / exp(Scattering * 0.4f);
15. return float4(Light, Light, Light, tex.r);
```

请注意，在第二行代码中，我们使用-tex.r 作为太阳到云彩的光线的最终位置。这是因为，在现实生活中，云彩并不是映在平面上的，而是三维的，所以在追踪循环里，云彩的高度域会作用于两个方向（云彩平面上方和下方相等的距离）。还要注意，取幂操作是在计算光线信息之后才执行的。这是因为取幂操作只是一个小技巧，是为了让云彩看上去更加逼真。第 7 行到第 13 行是着色器的核心部分：太阳到云彩密度域的追踪循环。正如我们所看到的，这个追踪循环采用的增量是 64，增量单位是 2，以此对追踪过程中经过的云体密度进行累加。第 12 行代码进行的是检测操作，确定当前的光线位置是否在云体内部。从本质上讲，我们是要检查当前追踪位置的 *Z* 轴位置是否在云体内部。第 14 行粗略地近似计算出光线的散射。最后在第 15 行中，返回经过光照处理的像素。随书光盘中提供了这个着色器的完整代码，以及一个演示程序。对于那些没有支持 Shader Model 3.0 显卡的朋友，我们还提供了一个演示视频。

3. 性能

由于使用了追踪循环，这个技术对填充率有着非常高的依赖性。我们的开发工作是在 GeForce 6 图形卡上完成的。这个天空动画在 640×480 分辨率下，可以达到每秒 60 帧。随着分辨率的提高，性能的下降很明显。如果把追踪循环替换成直接光照，这个技术还可以在支持 ps 2.0 的硬件上运行。

5.1.4 优化

该算法的某些步骤可以针对每一帧提前计算出来，特别是噪音纹理的动画。在每一个像素上为每一个倍频使用 2 个纹理进行插值，与其如此，不如使用渲染目标体，把它们提前计算出来，这也是有可能做到的。在两个纹理之间进行插值操作是很慢的，我们可以把渲染目标体作为云彩着色器的输入纹理。另外，由于是在每一帧中渲染密度纹理中的前 4 个倍频纹理，因此我们就不需要在最终的云彩着色器中再次计算它们了。只需要从密度纹理中把它们提取出来，然后用最后 4 个倍频纹理添加最终的细节。

5.1.5 总结

模拟逼真的云彩是非常困难的。但是，这里要告诉大家，利用现代的图形硬件还是可以

PDG

获得非常好的实时效果的。我们预测，随着次世代游戏机产品的推出，我们有足够的渲染能力以高清电视的分辨率实现那些先进的算法（例如本文介绍的算法）。

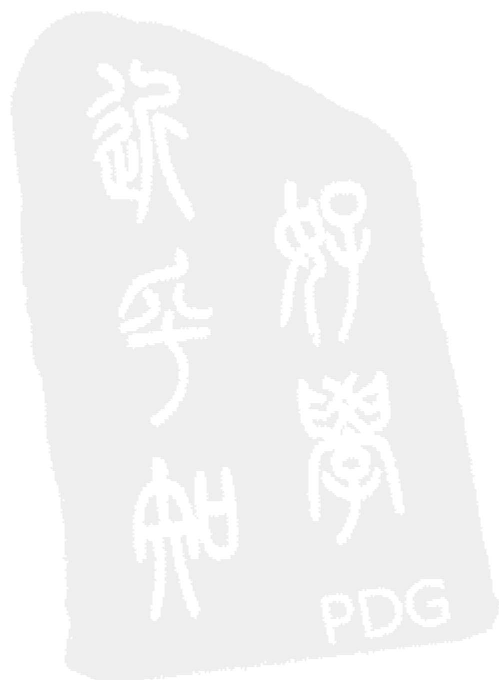
5.1.6 参考文献

[Elias] *http://freespace.virgin.net/hugo.elias/models/m_clouds.htm*.

[Harris01] Harris, Mark J. and Anselmo Lastra. "Real-Time Cloud Rendering." *Computer Graphics Forum* (*Eurographics 2001 Proceedings*), 20(3):76–84. September 2001.

[Harris02] Harris, Mark J. "Real-Time Cloud Rendering for Games." *Proceedings of Game Developers Conference*. March 2002.

[Pallister01] Pallister, Kim. "Generating Procedural Clouds Using 3D Hardware." In *Game Programming Gems 2* (edited by Mark Deloura). Charles River Media, 2001.



5.2 下雪吧，下雪吧，下雪吧（下雨吧）

微软公司（现就职于 Google 公司），Niniane Wang
niniane@gmail.com

微软公司，Bretton Wade
brettonw@microsoft.com

无论开发的是驾驶游戏、第一视角射击游戏，还是大型 RPG，逼真的雨和雪都可以增加室外场景的真实性。透过雨帘或浓密的雪观察车辆、荷枪实弹的士兵和长着两个头的怪物，会对它们有更好的感觉。

很多游戏都用粒子系统对降水进行建模[Reeves83]，即通过模拟每一滴雨或雪来产生真实的运动。例如，每片雪花可以根据风的方向，沿着它们自己的轨迹运动。但是，这种做法通常很昂贵，而且开销会随着降水密度的增加而增加，因为那需要模拟更多的粒子。

本文展示了一种用少量的性能开销渲染降水的技术，其方法是把纹理映射到一个双锥体上，然后通过硬件纹理变换来平移和拉伸雪片或雨滴。这种方法在 *Microsoft Flight Simulator 2004: A Century of Flight* 中得到了实现，如图 5.2.1 和图 5.2.2 所示，而且它在运动因素和雨滴外观上比一般的粒子系统有更好的艺术控制能力。

图 5.2.1 Seattle 的雪景——罕见的景色

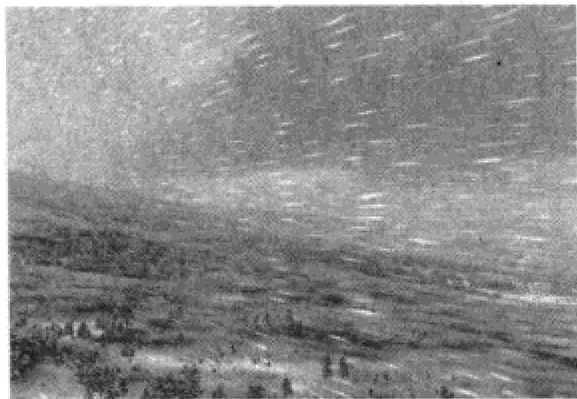
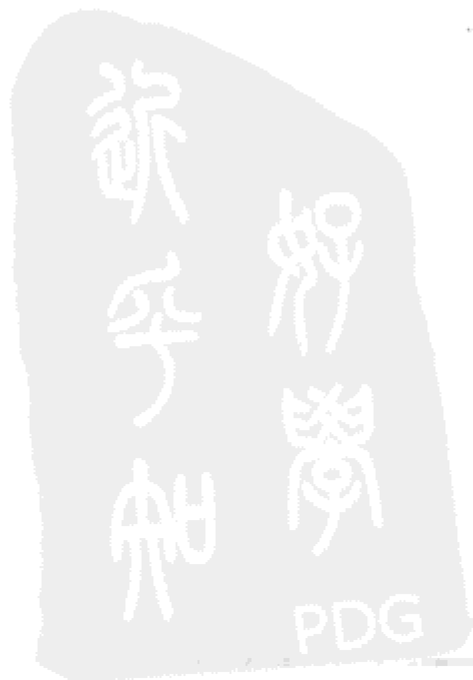


图 5.2.2 高速运动的雪片的截图


PDG

5.2.1　使用纹理为粒子束建模

我们的基本思想是使用一张纹理，以较低的性能开销来模拟降水。纹理由美工建立的一束雨滴组成，图 5.2.3 中显示了一个例子。

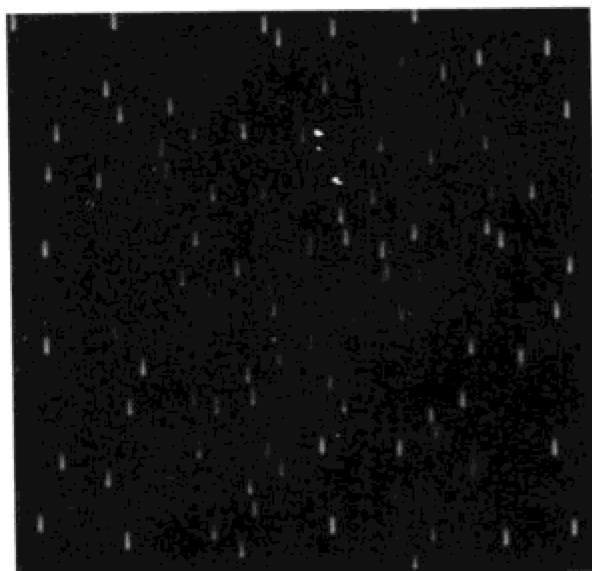


图 5.2.3　雨滴纹理

我们需要对纹理应用平移操作，使降水看起来会随着时间降落。我们从一个直白的方法开始——根据游戏的时间决定出所需的降水速度，并让平移纹理的速度和这个速度成正比。这个简单的方法会产生两个视觉上的走样。首先，当平移距离远大于单个雨滴大小的时候，眼睛就不会把一帧中的雨滴和下一帧中平移后的同一个雨滴联系起来，因为空间分隔较大，且缺乏运动模糊。因此，画面看起来就像是没有联系的雨滴在随机地掠过摄像机，而不是一个雨滴做连续运动。如果纹理坐标的平移更大一些——以至于一帧到下一帧之间几乎间隔了整个纹理的长度，雨滴看起来就像是在做反向运动，就好像老西部片中常见的车轮效果那样。为了修正这一点，每一帧平移纹理的大小要固定为一个雨滴的宽度。这才能确保眼睛会在帧之间连接每个雨滴的运动。

注意，平移距离是常量，与降水速度或帧速率没有关系。因为我们想让雨水看起来是以恒定的速率下降，不受帧速率的影响，所以我们必须考虑变化的帧时间——否则，雨水在帧速率快的时候会下降得快，在帧速率慢的时候会降得慢。为了防止这一点，我们根据 D_{precip} 在每一帧滚动纹理，D_{precip} 用方程 5.2.1 计算：

$$D_{precip} = S_{streak} * \frac{t_{delta}}{t_{const}} \qquad (5.2.1)$$

其中 t_{const} 是固定的帧时间，比如 1/30 秒。

为了模拟运动模糊，我们使用了前面的伸展因数。这个因数与移动速度成正比，所以当摄像机快速移动的时候，条痕会更长。

根据 Vec_{precip}、S_f（一个预先计算的缩放因数，用来缩小连续的纹理，让它们看起来越来越远）、C_{pixel}（一个转换因数，用于指定相当于纹理中每个像素的世界空间）和 S_{streak}（一个由美工控制的缩放因数，用于给出不同的条痕长度以区分雨的不同类型和强度，例如小雨的条痕比大雨长），按照方程 5.2.2 的描述计算伸展因数。

$$E = \frac{S_{streak}}{S_f * \left(C_{pixel} * S_f * Vec_{precip} + S_{streak}\right)} \qquad (5.2.2)$$

5.2.2　渲染雪或雨的视差

除了提供粒子移过玩家的感觉，我们还要提供深度和视差的感觉。远处的雨滴和雪片应该看起来比近处的更小、更慢[Langer03]。为了模拟这一点，我们把纹理滚动技术应用到多纹理上。我们的系统使用了四张纹理，一张张地逐步缩小纹理坐标，来产生更小的降水和更慢

PDG

的滚动。我们会使用 Direct3D 固定的多纹理把这四张纹理混合起来。

缩小的量取决于降水的类型和强度，是由美工控制的。

虽然这产生了视差的效果，但是 Z 信息是不正确的。纹理被设计为看起来离摄像机很远，但实际上它们被渲染在一个很近的平面上。不过，当水滴足够小的时候，这个走样是不易察觉的。

5.2.3 用锥体模拟摄像机的移动

当摄像机向前移动的时候，雨或雪应该倾斜，以使它看起来像射向和越过摄像机。类似地，当摄像机侧向移动的时候，降水条痕在移动中应该有一个侧移成分。为了模拟这些效果，我们把这四张降水纹理映射到一个双锥体上，如图 5.2.4 所示。我们把纹理设计成可以拼接，这样当它们在锥体表面上重复的时候就看不出缝隙。

为了降低锥体尖端奇异点的可见性，我们要加强锥体在这些地方的透明程度，并让它们朝着锥体相接的地方逐渐地减弱。我们还会选择一个适当的锥体高度来调整雨滴开始下落的角度。如果锥体太短，那么当摄像机固定的时候，因为圆锥陡峭的边缘，雨会以一个不真实的倾斜角度向下落，而不是垂直下落。另外，圆锥也不能太高。这才能确保当摄像机移动且圆锥旋转的时候，雨滴会按照需要从屏幕中间朝边缘处下落。

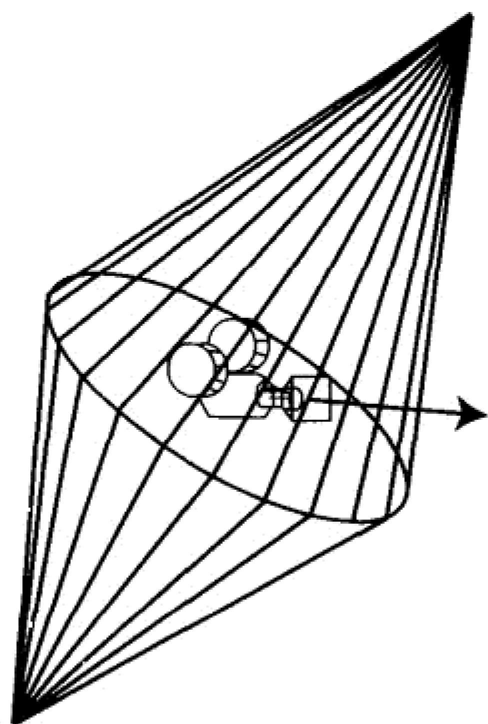


图 5.2.4 映射纹理的双锥体

随着摄像机的移动，我们会旋转双锥体，使降水看起来像从移动的方向落向摄像机。其实现方法是通过把锥体的世界矩阵设置为沿着降水移动向量 Vec_{precip}，Vec_{precip} 可用方程 5.2.3 计算。

$$Vec_{precip} = \left(C_f * Vel_{camera} + Vel_{gravity}\right) * t_{delta} \tag{5.2.3}$$

其中，Vel_{camera} 是摄像机的速度，C_f 是一个由美工控制的衰减常量，用来限制圆锥的倾斜；$Vel_{gravity}$ 是因为重力造成的降水速度；而 t_{delta} 是和上一帧的时间间隔。

5.2.4 合并到一个矩阵中

D_{precip} 和延长因数 E 可以合并到方程 5.2.4 中演示的纹理变换矩阵中。D_{precip} 是以像素空间表示的，并通过 E 转换为世界空间，它也会在连续的纹理中减慢滚动。

$$\mathbf{M} = \begin{bmatrix} S_f & 0 & 0 & 0 \\ 0 & \dfrac{S_{streak}}{S_f * \left(C_{pixel} * S_f * Vec_{precip} + S_{streak}\right)} & 0 & 0 \\ 0 & S_{streak} * \dfrac{t_{delta}}{t_{const}} & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.2.4}$$

我们也可以通过改变矩阵上的缩放和偏移来模拟不同的降水强度。例如，强的降水用较大的雨滴（较少的缩小）和较快的移动（较大的平移）来建模。设计出有这么多可调变量的系统，它的一大好处就是可以由美工进行控制。

5.2.5 增加美工控制

我们设计的系统使得美工可以在最终样式和感觉上做精细的控制。首先，美工要建立纹理，所以他们会控制雨滴和雪片的分布。我们对小雨和大雨使用不同的一套纹理，而美工可以针对每个强度调整雨滴密度。美工也可以把薄雾和烟尘加入到纹理中，那会使场景中的物体变得朦胧。在纹理的一些部分，雾可能比较浓，以增加变化性。

通过让美工在一些点上调整衰减和缩放因数，我们还允许他们控制下雨/下雪方程。我们写了一个工具，面向游戏中的渲染引擎，以便使用滑动条来调整可视化参数。

美工可以调整 S_f 来改变雨滴大小和滚动速度。他们也可以调整 S_{streak}，条痕的长度（例如，使小雨的条痕比大雨长）和 C_f 来限制锥体的倾斜。

对于雨或雪的每种强度，美工会实验多组参数设置，直到找到正确的条痕长度、雨滴大小、速度和圆锥衰减组合。即时的视觉回馈和美工控制的参数是建立高质量最终结果的关键。

5.2.6 总结

我们的系统在雨和大雪方面看起来非常真实。在现实生活中，小雪在下落时会翻滚和摇摆，这在我们当前的系统中还不能很好地模拟。随着时间在纹理坐标附近进行偏移，或许能近似得到这种效果，这是未来工作的一个领域。

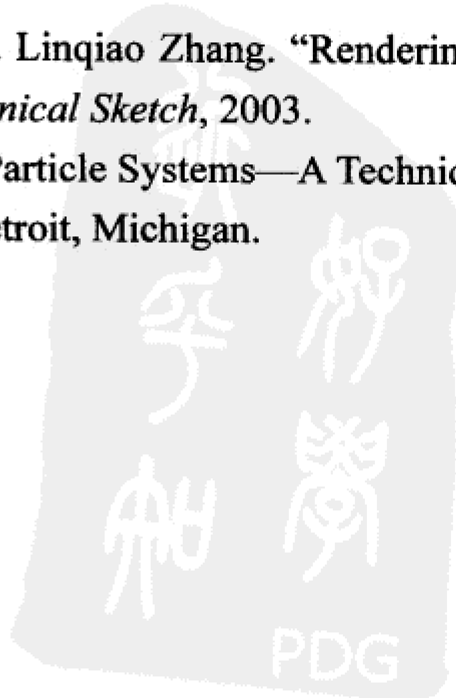
本系统也更适合于帧速率相对稳定的游戏。当游戏帧速率变化非常大的时候，降水纹理的滚动也会变化，其相对于帧的其余部分会更容易被注意到。实际上，伸展会加强变化，可能导致其效果看起来像反向移动。

该技术对游戏整体帧速率的影响可忽略不记，且对于大的降水和小的降水，其性能开销是一样的，这一点不像粒子系统。*Microsoft Flight Simulator 2004: A Century of Flight* 使用了该技术，并在消费级 PC 上保持了 15 到 60 fps 的帧速率。

5.2.7 参考文献

[Langer03] Langer, Michael and Linqiao Zhang. "Rendering Falling Snow Using an Inverse Fourier Transform." *SIGGRAPH Technical Sketch*, 2003.

[Reeves83] Reeve, William T. "Particle Systems—A Technique for Modeling a Class of Fuzzy Objects." *SIGGRAPH 1983, ACM.* Detroit, Michigan.



5.3 Widget：快速渲染和持久化小物体

Martin Brownlow
martinbrownlow@msn.com

渲染室外环境的游戏面临着一个共同的问题：为了显得真实，必须绘制大量的小物体。本文的关注对象就是这些小物体，包括树木、灌木、大石头、草茎和其他地面物体。这些物体使得地面纹理具有复杂的层次，形成了更真实的、外观立体化的地面。

用来做小物体的物体（我们应该称之为 *widget*），通常是不动的（它们不能参与交互），它们的存在只是为了增加地面的深度感。虽然这些 widget 可以随着活动场景的新区域随机地产生，但是机敏的玩家在返回前一个位置的时候可能会注意到一些细微的区别。另外，美工通常要控制所有影响关卡外观的东西，所以最好定义好哪种类型的小物体会出现在哪块区域，以及它们出现的频率。在极端的情况下，美工甚至可能想更进一步，单独放置一些 widget 来达到特殊的外观。

另一个需要考虑的是可破坏性：虽然 widget 本来是不动的，但是我们经常希望场景事件会影响它们。例如，游戏中巨大的爆炸会通过更改原先的地形纹理来表现地面所受的破坏，那么如果小草丛在爆炸中毫发无伤，那就非常荒谬。有些方法能够临时或永久地从场景中移除独立的 widget 或 widget 组，这有助于增加沉浸感。

对于高效渲染小物体，我们需要注意两个问题：首先我们能够使快速生成的 widget 落在视觉平截体中，然后高效得出模式的结果列表。

5.3.1 Widget 的网格

虽然我们的目标是渲染高度复杂的、覆盖着小物体的场景，但是组成 widget 的独立网格完全不需要很复杂。实际上，用作 widget 的最有效的模型通常只使用少数几个双面多边形和单张的纹理。这种简单性使我们可以渲染比其他方法多得多的实例，从而增加场景的总体复杂度。图 5.3.1 演示了

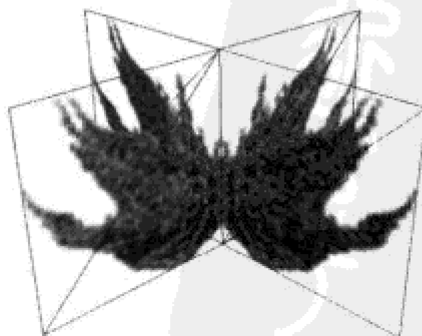


图 5.3.1 一个简单的地面覆盖网格

PDG

一个可以用作 widget 的简单的网格，它由 8 个双面三角形和一个纹理组成。

不幸的是，虽然网格本身非常简单，但是绘制它们却不是件简单的事了。今天的图形硬件可以渲染巨量的三角形，但是只有当渲染的单位是由上千个顶点组成的块时才能得到最好的性能表现。渲染上千份不同的八个三角形网格并不能最好地利用图形硬件。

5.3.2　高效地绘制 widget

把“该如何生成一组要绘制的 widget”的问题放在一边，先看看应该如何高效地绘制大量的小模型。我们已经确定了 widget 网格是由少量三角形和单个简单的材质组成。接下来可以进一步认为每个 widget 网格应该只由一个长度不确定的三角条带组成。

为了从图形硬件上获得最好的吞吐量，我们必须尽可能少地提交三角形组，且每一组应该做得尽量大。然后，我们的目的就是：找到可以把大量小模型分在一起的方法，这些小模型的位置在一帧一帧之间可能会独立地改变（随着视点的改变，需要画不同组的 widget)。这显然排除了让一个大的顶点缓冲区包含预变换的 widget 的方法，因为随着视点在世界中的移动，我们需要不断地编辑这个缓冲区。

1. Widget 组

虽然用预变换的 widget 填充顶点缓冲区是毫无疑问的，但是我们是否可以找到一个方法来使图形硬件用正确的矩阵变换每个 widget 呢？实际上，我们可以做到。这和蒙皮的问题相似，每个顶点在变换的时候会索引到一个矩阵调色板中，然后把多份相同的 widget 加入缓冲区。每个独立的 widget 对它包含的每个顶点必须有相同的索引值，但连续的 widget 可以而且应该有不同的值。使用这样的索引系统，只要顶点常量还有矩阵的空间，我们就可以把尽量多的 widget 放入单个顶点缓冲区中。这种方法由 Gosselin 等人引入，用来渲染大群的角色，但因为是骨骼动画，所以一次 API 调用中只能渲染 4 个角色[Gosselin04]。对小物体来说，我们在一次 API 调用中可以渲染更多的 widget。下面的结构演示了描述 widget 的一个可能的顶点格式。

```
typedef struct
{
    float   position[3];
    float   uv[2];
    u32     mtxIndex;

} WIDGETVTX;
```

现在我们建立了一个顶点缓冲区，它包含多份相同的 widget，每一个都有自己的纹理索引，我们必须指出如何在单个组中绘制它们。由于每个 widget 都是由单个三角条带组成，因此现在只需要把这些连续 widget 的条带缝合在一起就行了。这可以通过使用退化三角形来完成，退化三角形有两个或多个顶点是相同的。要把两个三角条带连接在一起，我们需要以两个索引的形式增加 4 个退化三角形。例如，要连接两个条带（1, 2, 3, 4）和（8, 9, 10, 11），我们可以让第一个条带的最后一个顶点和第二个条带的第一个顶点重合，形成条带（1, 2, 3, 4, 4, 8, 8, 9, 10, 11）。在这个例子中，4 个退化三角形是（3, 4, 4），（4, 4, 8），（4, 8, 8）和（8, 8, 9）。退化三角形都不产生像素，所以最终结果看起来像画了两个不连接的三角条带，但是只用了

单次绘制调用。

然而，用这种方式连接条带要求第一个条带的索引是双数。如果第一个条带的索引长度不是双数，那么第二个条带的缠绕顺序就是错误的，因为条带中的三角形的缠绕顺序与其他三角形都是相反的。为此，如果第一个条带的长度是单数个索引，那么我们必须让它成为双数，其方法是重复最后一个顶点。例如，把两个条带（1, 2, 3, 4, 5）和（8, 9, 10, 11）连接成一个有 12 个顶点的条带，结果就是（1, 2, 3, 4, 5, 5, 5, 8, 8, 9, 10, 11）。

因为顶点缓冲区包含多个 widget 的实例，所以我们也必须建立一个包含单个条带的索引缓冲区，来绘制所有这些 widget。当建立这个缓冲区的时候，必须记得每个连续 widget 的基顶点必须增加一个 widget 的顶点数目，以寻址到正确的顶点集。下面的代码将产生 nw 个 widget 的单个条带每个 widget 都由 nv 个顶点和 ni 个索引组成，并返回条带中生成的索引的数目。

```
u32 CreateWidgetIndices(
                u16 *pOutput,    // 输出缓冲区
                u32 nw,          // widget 的数量
                u32 nv,          // #每个 widget 的顶点数量
                u32 ni,          // #每个 widget 的索引数量
                u16 *pIndices ) // 指向 1 个 Widget 索引的指针
{
    u32 i,basev,j;
    u16 *pout;

    // 基顶点 = 0
    basev = 0;
    pout = pOutput;

    // 对于每个 widget
    for( i=1;i<=nw;i++ )
    {
        // 复制 widget 的索引，由基顶点来偏移
        for(j=0;j<ni;j++ )
        {
            pout[j] = pIndices[j] + basev;
        }
        pout += ni;

        // 如果 Widget 长度是奇数
        if( ni&1 )
        {   // 重复最后一个索引
            pout[0] = pout[-1];
            pout++;
        }

        // 如果现在不是最后一个 widget，就加一个退化三角形
        if( i!=nw )
        {
            // 建立退化三角形：
            // 重复最后一个索引
            pout[0] = pout[-1];
```

```
            // 增加基顶点
            basev += nv;

            // 重复下一个 Widget 的第一个顶点
            pout[1] = pIndices[0] + basev;
            pout += 2;
        }
    }
    // 返回索引数目
    return pout - pOutput;
}
```

2. 绘制 Widget 组

一旦我们有了顶点和索引缓冲区，就可以在单次绘制调用中绘制多个 widget。为了实现这一点，我们首先需要生成一组必须绘制的 widget 和它们对应的变换矩阵。然后，我们将把第一批 widget 的变换矩阵发送给对应的 vertex shader 常量寄存器，并绘制图元。如果 widget 的数目多于可以在单个组中发送的数目，那么我们只要发送多个包含尽可能多的 widget 的组就可以了。

只要画的是完整的 widget 组，那就相对简单，但当我们不能完全填满一个组的时候会发生什么呢？在这样的情况下，我们只要从条带中减少要渲染的索引数目，以去掉一些 widget 就可以了。如果一个 widget 是 n 个索引长，且我们要把该 widget 画 m 份，则那就需要画$(((n + 1) \& (-1)) * m + ((m - 1) * 2))$个索引。也就是说，我们需要绘制单个 widget 所需要的索引数目，向上取整到偶数，乘上要画的 widget 数目，再加上每个要画的 widget 之间的两个额外索引。

3. 压缩 Widget 变换

正如我们在前面看见的，如果 vertex shader 常量寄存器中还有用于变换矩阵的空间，我们就可以在单个组中画尽量多的 widget。一般来说，一个矩阵变换会占用三个顶点常量寄存器，但是我们可以在 widget 上使用一些限制，以将其减少到两个常量寄存器，增加可以绘制的 widget 数目。我们知道 widget 是假设放在地面上的，因此我们可以限制每个 widget 的自由度。明确地说，我们将把 widget 限制到五个自由度、一个 3D 位置、一个绕着垂直轴的旋转和一个缩放系数。用于这种变换的矩阵如下所示。

$$\begin{pmatrix} \sin(\alpha)\cdot scale & 0 & \cos(\alpha)\cdot scale & x \\ 0 & scale & 0 & y \\ \cos(\alpha)\cdot scale & & -\sin(\alpha)\cdot scale & z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

如我们所见，这个矩阵包含非常少的独立值。如果顶点处理器有任意元素 swizzle 的能力，我们就可以简单地把需要的元素压缩到下面的两个常量寄存器中。

$$\begin{pmatrix} x & y & z & scale \end{pmatrix}$$
$$\begin{pmatrix} \sin(\alpha) & \cos(\alpha) & -\sin(\alpha) & 0 \end{pmatrix}$$

如同我们将在下一节的 HLSL vertex shader 代码中看见的，从这些常量重建出变换矩阵是很

容易的事情。和使用 4×3 的矩阵来表现每个 widget 的更通用的变换相比，通过把用于每个 widget 的变换矩阵压缩成这样的形式，我们可以在一个组中多画 50%的 widget。

4. 实际使用中的 Widget

ON THE CD

看了绘制 widget 的理论之后，让我们看一个现实的例子。配套光盘中包含 FoliageDemo 的目录（/chapter5-Graphics/5.03-widgets-Brownlow）是一个简单的小物体 demo 的源代码。文件 widgetmesh.h 和 widgetmesh.cpp 定义了一个类——`CWidgetMesh`，它处理了优化渲染给定 widget 的大量实例涉及的所有步骤。

FoliageDemo 程序要求 PC 图形硬件可以执行的 vertex 和 pixel shader 为版本 1.1 或者更高。它还必须可以同时显示四张纹理。如果图形硬件没有这样的能力，那么 FoliageDemo 程序将不会执行。

成员函数 `Create` 负责从一个常规网格建立一个 widget 网格。它的参数是一个顶点数组的指针、一个索引数组的指针、数组中顶点和索引的数目，以及纹理的名字。这个函数会建立一个顶点缓冲区和一个索引缓冲区，它们的空间足够容纳 `WIDGET_MAXINSTANCES` 份（定义在“widgetmesh.h”中）输入数据，然后按本章前面描述的方法填充缓冲区。

为了绘制一系列的实例，我们必须首先调用 `Begin` 成员函数。这个函数会把渲染流水线设置为渲染 widget。它会设置和网格相关的顶点和索引缓冲区，以及 vertex 和 pixel shader、顶点格式和纹理。调用完 `Begin` 之后，我们就可以迭代想要绘制的实例，对每一个实例调用 `AddInstance`。每一次执行 `AddInstance` 的时候，它都会把实例的一个记录保存在静态数组中。一旦这个数组满了，`FlushInstances` 将会自动被调用。

`FlushInstances` 成员函数负责绘制一批 widget。它首先把 `AddInstance` 建立的一组实例发送到适当的 vertex shader 寄存器。之后，调用一个 `DrawIndexedPrimitive` 命令，使用适当数量的索引和顶点，就和从数组中的实例计算而来的一样。最后，它会清空数组中的实例并返回。`FoliageDemo` 类中最后一个相关的函数是 `End` 函数。这个函数只是调用 `FlashInstances` 来确保所有添加的实例都已经绘制了。

用来绘制 widget 的 vertex shader 程序在文件 widget.hlsl 中。这是一个非常基本的 vertex shader；惟一需要注意的地方在于它是用一个压缩的实例矩阵变换每个顶点。这段 shader 使用 swizzle 来重新建立未压缩旋转矩阵的第一和第三行（第二行只包含一个元素 *scale*，所以实现起来没什么），并用它们来旋转输入点，然后加上实例位置。这由下列代码片断表现。

```
float4  pos;

pos.x   = dot(vtxin.pos,mtxInstances[vtxin.index+1].xwy);
pos.y   = vtxin.pos.y;
pos.z   = dot(vtxin.pos,mtxInstances[vtxin.index+1].ywz);
pos.w   = 1;
pos.xyz  = pos.xyz*mtxInstances[vtxin.index].w +
           mtxInstances[vtxin.index].xyz;
```

一旦计算出旋转和缩放位置之后，这个点就变换到了夹持空间并和平常一样被传给 pixel shader。

5.3.3　裁剪 widget

现在我们可以高效地绘制大量实例了。首先必须看看如何根据一个总是改变的视点生成相应的实例，主要有两种方法可以实现这一点。第一个方法是在到达场景的一个新区域的时候，半随机（但是可以确定）地生成实例，不再看得见的实例就会被删除。第二种方法，也是本章将要集中讨论的方法，是预计算场景中所有 widget 实例的位置并建立一种高效的表示法，用它来快速生成 widget 的可见集合。

虽然会消耗更多的内存，但是该方法的主要优点是这些 widget 的位置变成了常量——玩家可以离开一片区域，后来返回的时候会发现小物体的位置和原先一样。此外，使用此方法，只要在程序员那边做一点点努力就可以在必要的时候删除 widget。这使得玩家可以用爆炸物或其他任何武器在场景的小物体上留下记号。

1. BSP 树

我们将用来存储场景中 widget 实例位置的方法是修改后的 BSP 树。使用 BSP 树，而不是四叉树或八叉树，使得场景可以是非规则的形状，而不用花费额外内存来存储空的节点。对于不熟悉 BSP 树的人来说，它们由层次化的面组成。每个平面会把场景切成两片，场景表面上的每个物体都在每个平面的一边或另一边。图 5.3.2 演示了 BSP 树的前两个平面的例子。第一个平面把场景分成两半，第二个平面把这两半中的一个再分成两半。以这种方式，场景中的物体就被划分成了一棵二叉树。

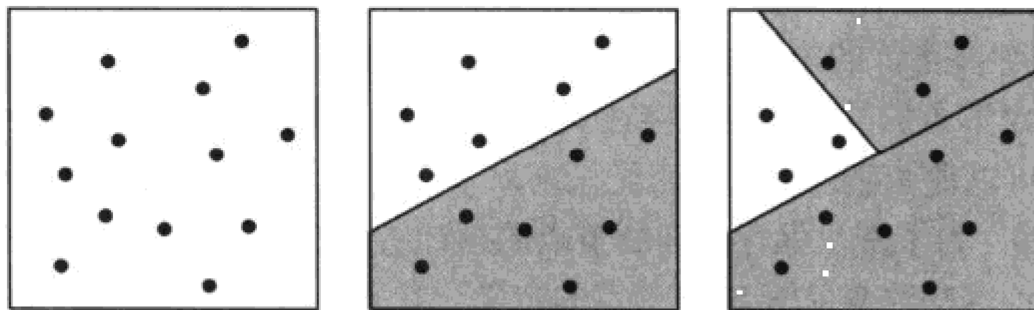


图 5.3.2　BSP 树的构造

一个典型的 BSP 节点可以用下列结构体表示：

```
typedef struct
{
    float   plane[4];      // 裁剪平面
    BSPNODE *front;        // 平面前方的节点
    BSPNODE *back;         // 平面后方的节点
    BSPLEAF *coplanar;     // 同平面的元素

} BSPNODE;
```

这个结构体有 28 个字节长（这里假设目标机器的指针是 32 位的长度），有一些过多。另外，因为该结构体包含三个可能需要解引用的指针，所以对于使用这种 BSP 节点表示法的代码，其数据访问模式不能保证是最好的，这会破坏数据 cache。最后，该结构体的长度不仅过长，

PDG

而且不是 cache 行长度的倍数（或约数）。这意味着 CPU 每一次把这个结构体放入数据 cache 中，它都必须填充至少一个（且有可能是两个）cache 行，具体情况取决于这个节点的内存地址。图 5.3.3 演示了 32 字节 cache 行的机器上的两种可能。

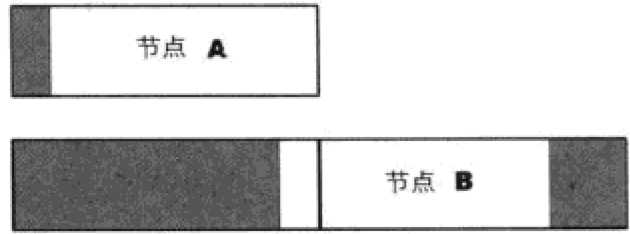


图 5.3.3 两个未对齐的 BSP 节点的 cache 行的使用

在图 5.3.3 中的每个大块中，白色区域表示 BSP 节点占用的内存，灰色区域表示读入内容未知的 cache 的内存。根据图 5.3.3 的演示，可以很容易地看到当随机访问 BSP 节点的时候，有很多未使用的数据可能会被放入数据 cache。即使节点在内存中是连续存储的，但是我们仍要随机地访问它们（从内存控制器的角度看，实际的访问模式是由视点位置和 BSP 树的布局决定的）。这意味着放入数据 cache 的无用数据的数量和有用数据一样多，其造成的性能损失相当大。显然，我们得以一定的方式修改算法和数据结构，以避免这种糟糕的 cache 行为。

2. 高效使用内存的 BSP 树

提升 CPU cache 使用率的两种明显方法是减少 BSP 节点结构体的大小，以及使用某种类型的可预测内存访问模式。检查一开始定义的节点结构体，可以看出它分成两个大致相等的部分。第一个部分包含分割平面，第二个部分包含子节点的位置。

通过确保选择的每个平面都是轴对齐于三个坐标轴之一，就可以极大地减少存储每个节点的分割平面所需的内存。这样，我们就可以将平面所需的 16 个字节减少到 4 个字节（保存平面到原点的距离）和 2 个位（用于表示用了哪个轴）。这留下 30 多位来存储树的指针，为了占用总共微不足道的 8 个字节。下面看看是如何实现的。

为了节省用于存储指针的内存消耗，我们还需要利用一些内存访问模式来进行树查询。如果我们总是确保给定节点的前子节点保存在父节点的后面，那么我们就可以完全不需要前面节点指针。另外，我们可以使用额外的两位来记录前后节点是否是叶节点，只包含数据（在这里，是 widget 实例）。最后，为了减少保存指向后节点的指针所需的内存，我们可以把它保存成距离当前节点的绝对位移。这些改变使得 BSP 节点的定义一共只占 8 个字节，如下面的结构体所示。

```
typedef struct
{
    u32     axis            : 2;
    u32     numFrontLeaves  : 4;
    u32     numBackLeaves   : 4;
    u32     backNodeOffset  : 22;
    float   distance;

} WIDGETNODE;
```

如果可以保证搜索 BSP 树的代码都是先访问节点的前子节点，然后才访问后子节点，那么我们就可以最大化地使用数据 cache。这是因为前子节点的数据非常可能位于数据 cache 中，因为它靠近于它的父节点。

3. 建立 BSP 树

建立 BSP 树的工作是非常微不足道的，特别是如果把每一个 widget 实例看作一个点来

处理的话。在建立了场景中所有 widget 实例的数组之后，我们会为它们计算一个轴对齐的包围盒。在每个节点上，我们会找到这个盒子的最长轴。这个轴会变成分割平面。在把节点按照它们的位置和轴之间的距离排序之后，我们就把该平面放在两个中间 widget 之间的地方。也就是说，如果场景中有 63 个 widget，分割平面将被放在第 31 和第 32 个 widget 之间。

继续用这种方式建立节点，直到每个节点中只剩下少量的 widget 实例，然后这个节点就被称为叶节点。这个实例中的叶节点被看做一个 WIDGETLEAF 结构体的数组。数组中剩下的 widget 数量放在数组的每个元素中。这里的 WIDGETLEAF 结构体也只占用 16 个字节的内存，保持了高效和对 cache 的友好。

```
typedef struct
{
    float   position[3];   // widget 的位置
    s8      sinAngle;      // 朝向的 sin (*127)
    s8      cosAngle;      // 朝向的 cos (*127)
    u8      scale;         // widget 的缩放比 (*32)
    u8      pad;

} WIDGETLEAF;
```

光盘中包含的 FoliageDemo 应用程序包括文件 widgetbsp.cpp 和 widgetbsp.h，它们描述了 CWidgetBSP 类。这个类负责建立和管理一个 widget BSP 树。CreateTree 成员函数的参数是一个 WIDGETLEAF 结构体的数组，会根据它们建立一个 BSP 树，正如本节所描述的那样。一旦树被建立出来了，我们就可以高效地搜索，以找到必须绘制的 widget 实例的列表。

4. 搜索 BSP 树

现在我们有了一个紧凑的、内存访问高效的 BSP 树定义，下面看看如何计算应该渲染的 widget 集合。为此，我们必须从 BSP 树中得到视锥中所有和视点在一定距离之内的 widget。我们可以用视口的大小、摄像机矩阵和视野来构造一个轴对齐的盒子，以包含有效的区域，如图 5.3.4 所示。

有了这个盒子，搜索 BSP 树就变得微不足道了。在每个节点，我们可以把与它相关的距离和观察盒中对应轴预计算的最小、最大值做比较，以确定要访问哪个或哪些子节点。我们必须记住，为了维护 cache 的一致性，如果要访问一个节点的前子节点，就应该总是在后子节点之前访问。

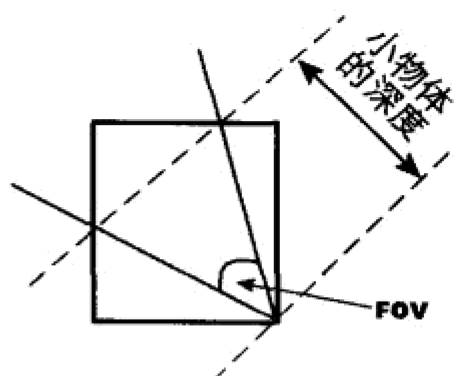


图 5.3.4 一个轴对齐的观察盒

传递了观察参数和最大绘制距离之后，CWidgetBSP 类的 Draw 成员函数将构造一个观察盒(view box)，然后对 widget BSP 树执行一个基于视点的搜索。对于在观察盒中找到的每个 widget 实例，会调用适当的 CWidgetMesh 类的 AddInstance 成员函数。

5.3.4 总结

在本章可以看到，通过减少绘制一组模型所需要的 API 调用次数，我们可以极大地减少

处理和渲染这些模型时的 CPU 和 GPU 开销。这可以形成一个高效的方式来绘制大量小模型，且该方法兼容于老的 vertex shader 1.0 的图形硬件。

我们也看到了如何才能构造一个内存高效和对 cache 友好的 BSP 树。这些优化允许我们建立和操控 BSP 树，以包含足够的小物体来覆盖相当大的场景。BSP 树的内存布局和访问模式甚至允许在内存速度相对较慢的系统上达到高效，比如 Sony Playstation 2 控制台。通过选择使用任何有效的 cache 预取指令，这个 BSP 树的效率可以进一步提高。

组合这两种技术使得我们可以在场景中放入大量小物体模型，而且可以高效地在场景中的任何地方进行绘制。

一种可能的增强方法是建立一个算法来决定小物体在一个小区域中的位置，然后在这些区域接近视点的时候建立 BSP 树。这样的算法允许我们在获得 BSP 树的内存 cache 效率的同时保持真实内存的使用率最小。这样就可以在内存消耗不变的情况下放置更多的小物体，因为在任何时候，只要存在当前视点下可能被渲染的 BSP 树就可以了。

5.3.5 参考文献

[Gosselin04] Gosselin, David, Pedro V. Sander, and Jason L. Mitchell, “Drawing a Crowd: Instancing in Current Hardware.” In *ShaderX*^{3} (edited by Wolfgang Engel). Charles River Media, 2004.

5.4 逼真的树木和森林的 2.5 维替用物

布达佩斯理工大学，Gábor Szijártó
szijarto.gabor@freemail.hu

开发真实和高性能室外环境可视化的最大挑战之一就是渲染植物。要建出令人信服的树和灌木的模型，需要非常多的三角形，这些三角形会超过当前渲染硬件的极限。过去，人们采用了许多方法来解决这个问题，其中的大部分都是多解析度模型和细节等级算法的变体。

本文开发了一种基于 2.5 维替代物（impostor）的方法[Szijarto03]，用于高解析度的树木渲染，该方法使用的是常规树的结构。另外，它可以利用最近的视频卡提供的可编程渲染流水线。这个算法在大部分细节等级（levels of detail）中使用视点相关的 2.5 维替代物来可视化令人信服的树。因为使用了替代物，所以其性能极大地依赖于视频卡的填充率。

5.4.1 引言

我们常常需要指定特殊的模拟比例，以便植物渲染算法可以提供需要的真实度。大部分应用程序可以指定成下列种类中的一个或多个。

昆虫比例：模拟的等级是固定的，希望真实地描述出独立的树枝和叶子。（角色可以爬树。）

人类比例：从触手可及到几十米远的距离范围内，场景必须看起来真实。最好能一致，但不是必需的。（角色可以撞到树，甚至冲过灌木，但不会关注特别的细节。）

车辆比例：在这个等级，植物和背景差不了多少。独立的树木几乎从来没有被关注过，也不需要一致性。观察距离可能超过几百米远。（角色穿过环境的时候通常会比地面高一些，而且比跑步的速度快。）

本研究的重点在于人类和车辆比例的模拟算法，它们可以应用于低高度飞行模拟（直升飞机和滑翔机），地面车辆模拟和第一人称射击。

植物可视化似乎是个棘手的问题。有两种常用的方法：基于物理和基于图像的方法。顾名思义，前者的技术是使用几何体来表现植物。对于一棵树，建立一个令人信服的模型要使用大概十万个多边形，必须应用一些形式的细节等级（LoD）渲染技术把一帧的多边形数量减少至合理的水平（[Remolar03]，[Puppo97]）。看起来不错的结果只能通过复杂的算法或极大的内存开销来达到。此时此刻，基于几何体的方法不适合实时应用程序。基于图形的方法在一致性和物理精度方面体现出了平衡，有利于更真实的视觉效果。

5.4.2 以前的基于图像的方法

在所有基于图像的方法中，最简单的是精灵渲染，如图 5.4.1 和 5.4.2 所示。该技术就像使用一个纸板图案，在上面画一个类似树的图像，而且总是面向摄像机。虽然其视觉效果远远无法令人满意，但是人们经常用这项技术描述小的植物。

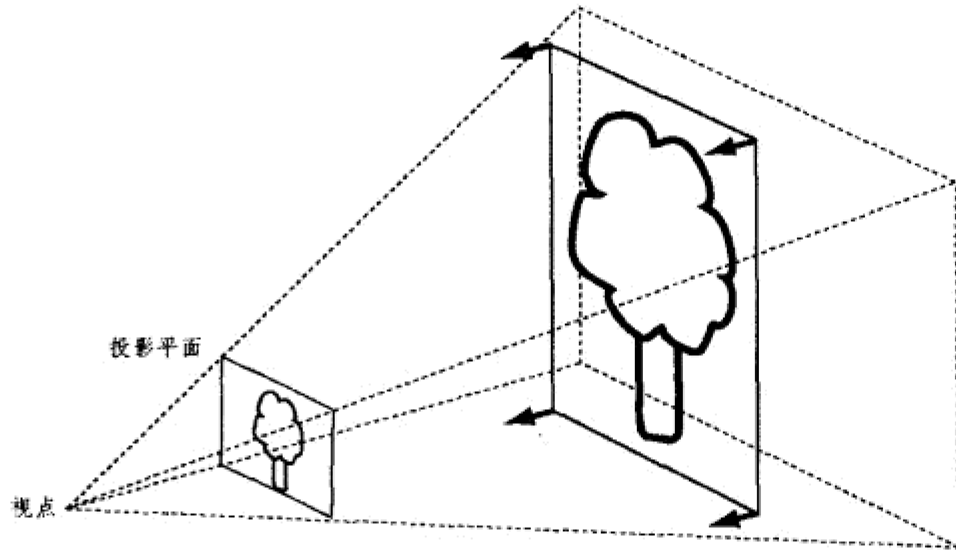


图 5.4.1 精灵渲染。贴了纹理的多边形总是面向摄像机

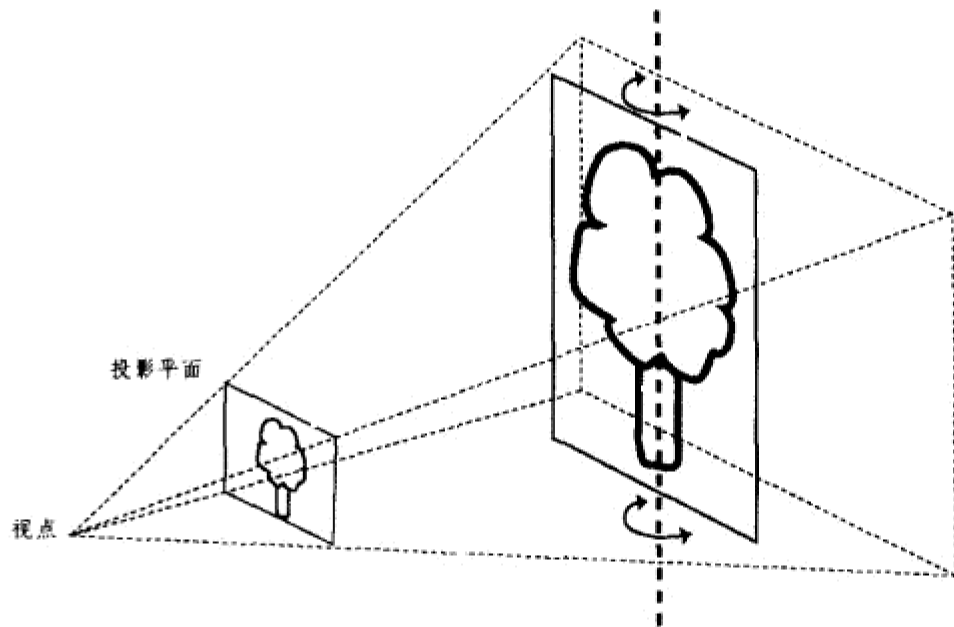


图 5.4.2 有任意位置和朝向的公告牌（billboard）

两种引入了某些形式的视点依赖的改进方法，是视点相关的精灵集和更复杂的图案。视点相关精灵的方法只需要预生成一个视点的有限集合（通常是 4 或 8），然后在运行期呈现出和视线方向轴最接近的一个。当轴发生改变并选择了另一个视点的时候，会看到弹出（popping）的走样。复杂图案的方法是使用纹理透明度和混合把多个视点同时渲染到正确对齐的表面上，如图 5.4.3 所示。在车辆比例的模拟中，这两种方法都会产生可接受程度令人惊讶的结果，但是当需要更近的视点时，它们就不能提供足够的质量了。引入树的不同形状和大小而不增加内存负担是很难的。光照也会受到这些方法的影响。

一种真正在商业娱乐软件中实现的最先进的方法，是应用了一些 LoD 的基本任意贴纹理的树模型，如图 5.4.4 所示。虽然这个主意直截了当，但只是到了这几年，硬件才变得强大到

可以处理这些任务。虽然因为使用了简单的几何体模型，结果的视觉效果是令人满意的，但是很近的视点通常看起来会充满走样，而且新模型或组合模型的部分经常会引入许多变化。在场景中增加树的数量会快速地降低性能。最近的人类比例模拟器是依赖于这项技术，并依赖于原始的硬件能力来处理它。

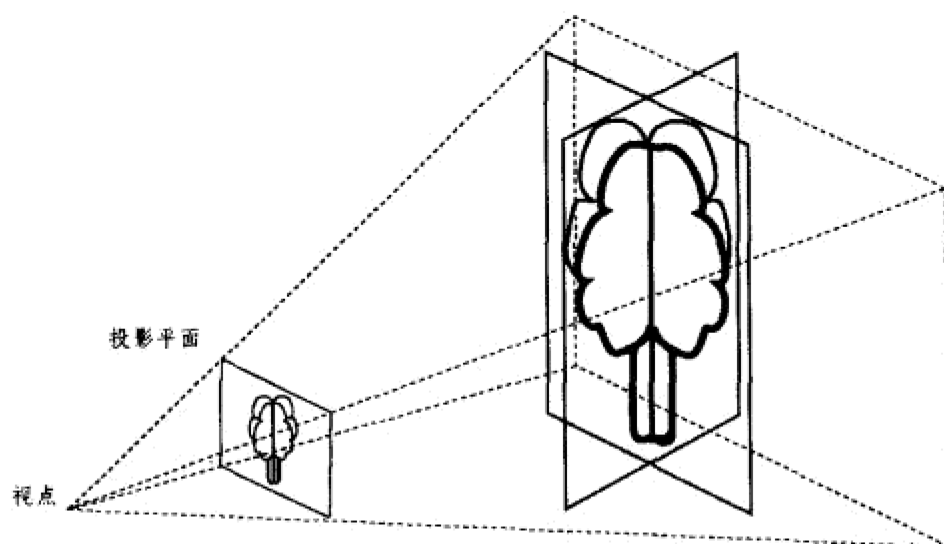


图 5.4.3　使用两个面的复杂图案

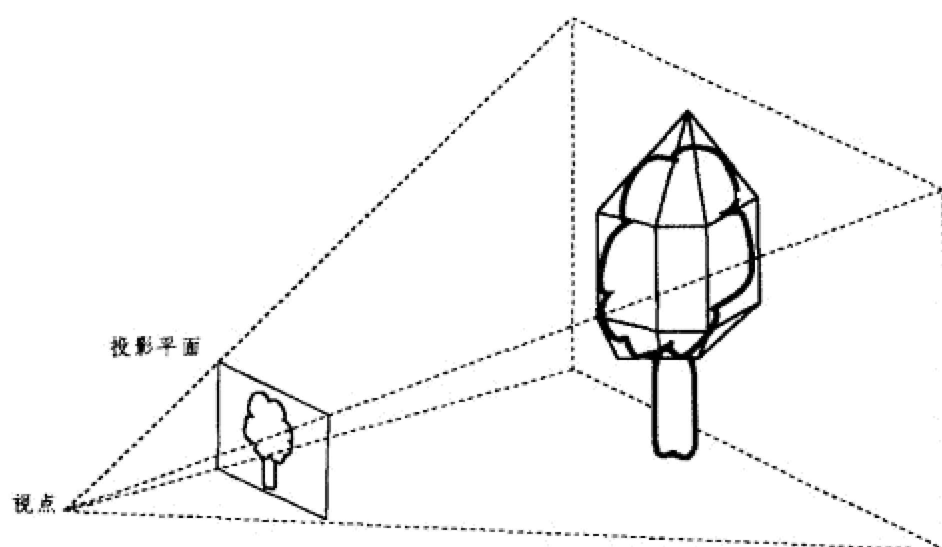


图 5.4.4　任意贴纹理的树模型

5.4.3　改进以前的方法

到目前为止，介绍的技术都在利用一个事实：渲染一个树的记录图像比真正处理描述树模型的几何体信息要快得多。这主要有两个原因：

- 树上的叶子通常会映射到非常少的像素。在今天的硬件上，渲染一个像素要比变换一个三角形快得多。
- 看不清的叶片数量非常大。因此，大量数目的变换都是没有意义的操作。

基于图像的方法有两个主要的缺点：固定的透视，以及移动和旋转的时候不会改变。幸运的是，在树的渲染中，一个固定的透视并没有太大问题。树冠是非常不规则的结构，和规则形状相比，人眼对不规则形状的透视形变要不敏感得多。其他问题更为麻烦。叶片在摄像机移动的时候是静止的，但在观察者相对于树移动的时候，我们是希望一些叶子出现，而另一些叶子变得不可见。必须用某种方法解决这个问题，以便把渲染质量提高到可以接受的层次。对人眼来说，弹出的走样和运动中的静态纹理一样麻烦。因此，我们可以得出结论：为

了达到令人信服的视觉效果，必须把叶片的几何体处理到某种程度。

下面描述的算法会把树冠渲染成一些小叶片云（leaf clouds）的集合。一个叶片云由建模整棵树的所有叶片中的一小部分组成。渲染一个叶片云所需的几何体是很少的，可以在逐帧的基础上处理。这里的想法是处理一个叶片云，把它渲染到纹理，并多次应用那张纹理来渲染树冠，因此引入了前面的基于图像的方法所没有的运动视差。不过，因为叶片云纹理的重叠，走样仍然会出现，除非能正确地处理深度信息。在利用深度一致的替用物渲染形成的图像中，一个叶片云可以正确地重叠于其他叶片云、树枝几何体、其他树或场景中的其他任何物体。

5.4.4 算法

这里建议的技术是对传统的替用物渲染的改进。替用物渲染有两个阶段。第一个阶段是与视点相关的渲染到纹理的操作（绘制替用物），它的结果会在第二个阶段中使用，作为精灵或公告牌纹理（参见图 5.4.5 和 5.4.6）。

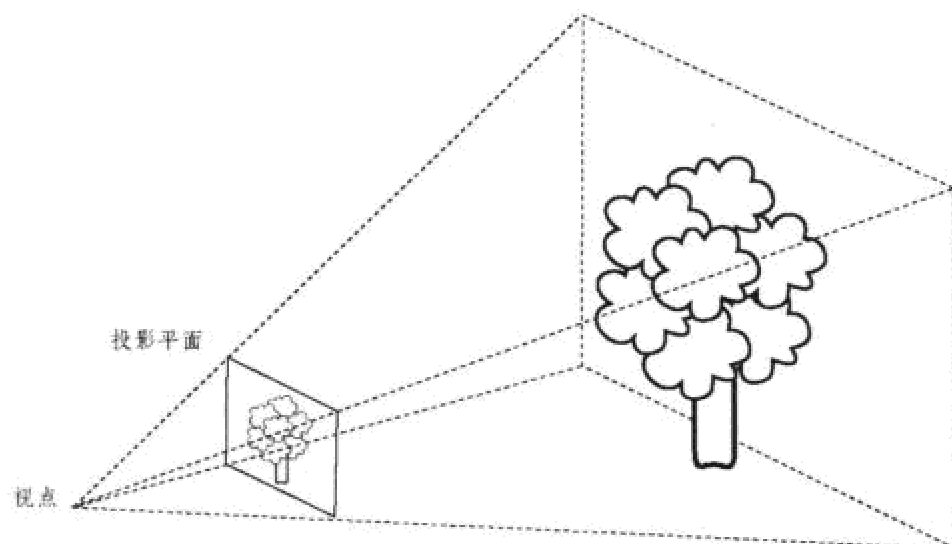


图 5.4.5 用多个精灵来渲染树

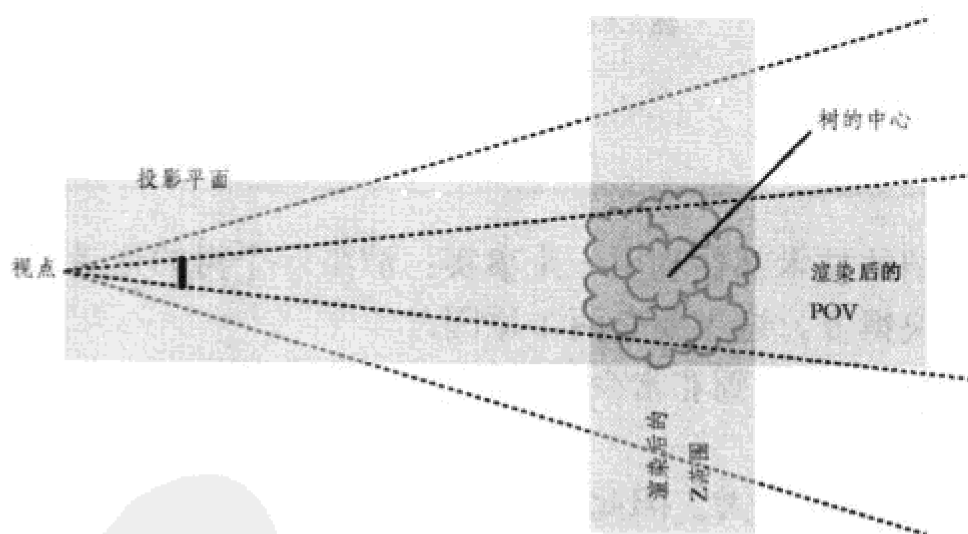


图 5.4.6 用一组树叶的替代纹理进行渲染，假设组的中心和树重合

这里的想法是把树冠表示成多个精灵，如图 5.4.5 所示。精灵在空间中相对位置的扰动可以引入足够的变化。为精灵渲染两个或三个不同的纹理可以引入更丰富的多样性。独立的精灵也可以用不同的颜色混合，从而再次引入了变化，甚至光照。使用 10 到 100 个精灵描述的树冠在静止图像中看起来非常逼真，如图 5.4.8 所示，即使是同样的纹理在所有的精灵上重复。

然而，当视点运动的时候，精灵的视点无关性就使得前面的方法在实时渲染中完全没用了。因为精灵总是面对着摄像机，所以如果它们的外观是不变的，那就会颠覆移动视差，产生非常不真实的效果。通过使用每一帧都更新的替代物而不是静态的精灵，因此而引入的视点相关性会完全去掉这个问题。

在替代物渲染阶段，深度和颜色信息都保存在目标纹理中。其结果是一个 2.5 维的替代物。*z*-near 和 *z*-far 平面被调整到接近一个用来渲染叶片组的合理的包围盒，如图 5.4.6 所示。在渲染的最后阶段，在深度测试执行之前，保存的深度值会适当地缩放和裁剪到最终的深度缓冲区中，这会给贴纹理的精灵产生一个立体化的感觉，这些精灵可以以空间一致的方式重叠。

在许多精灵上重复相同图像所造成的走样外观几乎完全去除了，因为立体化的重叠会削弱这种排列，使人不可能在任何点看到单个替用物。

5.4.5 实现

这里建议的技术会产生非常逼真的效果，并可以高效地用最新的视频卡 GPU 实现，如图 5.4.7 所示。图 5.4.8 演示了使用替用物的树冠渲染。

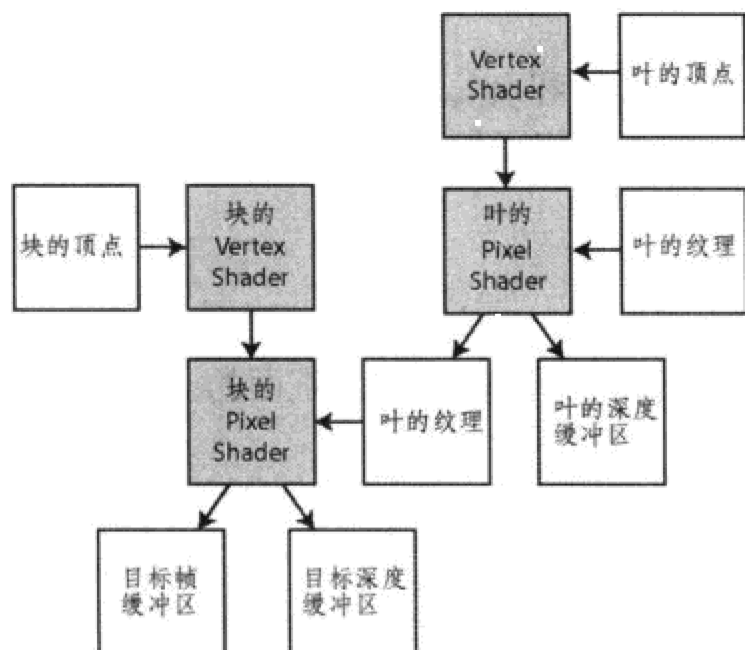


图 5.4.7 用两个 vertex shader 和两个 pixel shader 来实现的块状图

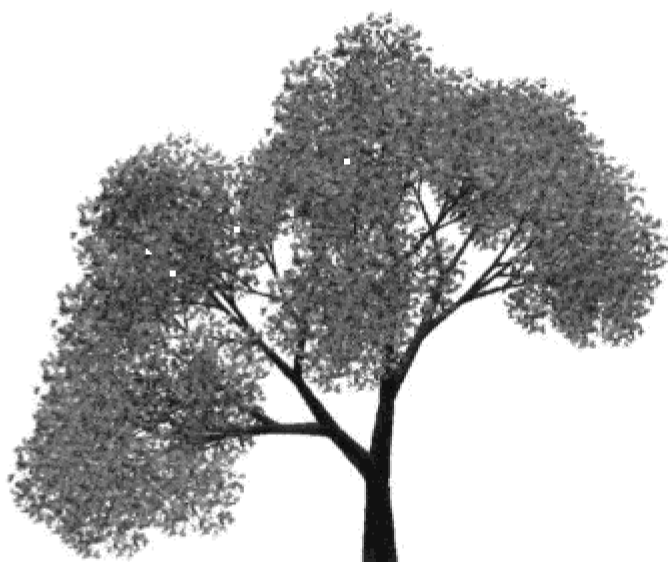


图 5.4.8 使用替用物渲染的树冠

该实现假设 vertex 和 pixel shader 的版本是 2.0，而且只使用标准的技术。代码是用 C++ 和 DirectX 9 写的[Microsoft04]。shader 程序用 Cg 写成[NVIDIA02]。样例程序演示了一个等价于 160 万个多边形的树。

叶片以最终图像的解析度渲染到替用物纹理中，以避免缩放的走样和性能开销。每一帧都分配替用物纹理会有大的性能开销，因此推荐应用程序为所有情况预先分配足够大的替用物纹理。

1. 深度精度

出于内部优化的考虑，当前的图形卡不提供深度缓冲区的高效访问。因此，第一遍的像素程序会把深度信息存储在 alpha 通道中，以允许下一阶段访问它。因为 alpha 通道只有 8 位的精度，所以应该仔细地设置投射矩阵以减少信息的丢失。

前裁剪平面被移到叶片云之前，而远裁剪平面被移到它的后面。另外，窗口中心被移到植物的中心。在 Direct3D 渲染流水线中，对应的投射矩阵是[Szirmay95]。

$$\mathbf{M}'_{\mathrm{Pr}} = \begin{bmatrix} w' & 0 & 0 & 0 \\ 0 & h' & 0 & 0 \\ x_{offset} & y_{offset} & \dfrac{z'_f}{z'_f - z'_n} & 1 \\ 0 & 0 & \dfrac{-z'_n \cdot z'_f}{z'_f - z'_n} & 0 \end{bmatrix}$$

$$w' = w \cdot a_{impostor}$$

$$h' = w' \cdot a_{impostorXY}$$

$$z'_n = z_{center} - r_{impostor}$$

$$z'_f = z_{center} + r_{impostor}$$

其中，w 是窗口宽度的一半（它可以从水平视域计算而来：$w = 1/\tan(fov/2)$，$a_{impostor}$ 是帧缓冲区和替用物纹理中像素列数目的比例，z_{center} 是眼睛位置和植物中心沿着轴 z 的距离，而 $r_{impostor}$ 是叶片云的半径。

要把叶片云移动到替用物纹理的中心，就要确定 x_{offset} 和 y_{offset}。这些值是通过把植物中心用标准的模型、观察和投射进行变换计算出来的，它们确定了屏幕坐标系中的植物中心。计算出的 x_{offset} 和 y_{offset} 值是前两个屏幕坐标乘上−1。

2. 渲染替用物

正如前面提到的，这个渲染算法有两遍。第一遍计算替代物，并使用下面的 vertex shader。

```
vertout main(appin IN,
             uniform float4x4  mModelViewProj,
             uniform float4    invTrSunDir,
             uniform float4    colorParam)
{
    vertout OUT;

    float4 pos     = IN.Pos.xyzz;
    pos.w          = 1;

    pos            = mul( pos, mModelViewProj);
    pos            /= pos.w;

    OUT.Pos        = pos;
    OUT.Col.a      = pos.z;

    float light    = max( 0, dot(IN.Normal.xyz, invTrSunDir.xyz));
    OUT.Col.rgb    = IN.Col * (light * colorParam.x + colorParam.y);

    return OUT;
}
```

叶片云将用这个 vertex shader 渲染到替用物中。每个顶点有一个位置、法线和颜色。vertex shader 也有两个额外的参数，用于定义阳光的方向，以及它的方向光和环境光强度。用散射照明模型来计算反射的强度。

vertex shader 的输出是变换后的位置和光照后的颜色值。颜色值包含像素颜色和 alpha 通道中的深度。

渲染替用物的 pixel shader 程序非常简单。因为在这种情况下，alpha 通道包含深度值，所以在执行这段代码之前我们必须禁止 alpha 混合。

```
void main( vertout     IN,
           out float4  color   : COLOR)
{
    color.rgba    = IN.Col.rgba;
}
```

替用物没必要 mipmap，因为替用物的纹素（texel）有相同的朝向，而且其大小基本相似于像素。由投射校正造成的缩放效果并不明显。

3. 使用替用物

第二遍是把树冠当作一系列的替用物渲染到最终图像中，这个过程更为有趣。vertex shader 用来计算替用物纹理的投射大小和位置。应用程序会把这些值载入 vertex shader 常量寄存器，就好像替用物位于植物的中心。顶点程序会根据叶片云在图像中的真实位置做出相应的矫正，并把相应的深度值传给 pixel shader。pixel shader 负责渲染替用物纹理，包括正确的深度信息。

把每个替用物绘制到屏幕需要两个三角形。首先，vertex shader 会计算替用物中心。替用物纹理包含了从替用物前裁剪平面到叶片的深度。因此，我们要把叶片云的 z 距离与存储在替用物中的 z 坐标相加。另外，替用物也会根据投射形变进行缩放。

```
vertout main(appin IN,
             uniform float4x4   ModelView,
             uniform float4x4   Proj,
             uniform float4     constans,
             uniform float      impostorSize,
             uniform float      centerBlockSize,
             uniform float      impostorZSize,
             )
{
    vertout OUT;

    float4 pos     = IN.Pos.xyzz;
    pos.w          = 1;

    // 从观察点出发,计算替代物的中心
    Pos            = mul( pos, ModelView);
    float4 scl     = float4( 0, impostorSize, pos.z, 1);

    // 投影变换
    pos            = mul( pos, Proj);
```

```
    pos            /= pos.w;
    pos.z          -= impostorZSize;

    // 计算替代物的大小
    scl            = mul( scl, Proj);
    scl            /= scl.w;

    // 计算缩放因子
    float scale    = scl.x / centerBlockSize;

    // 计算替代物的角
    float2 uvp     = (IN.uv - 0.5) * constans.xy * scale;
    pos            = pos + float4( uvp.x, uvp.y, 0, 0);

    // 计算替代物的纹理坐标系数
    float2 uv      = IN.uv * constans.z + constans.w;

    OUT.Pos        = pos;
    OUT.Col.rgb    = IN.Col;
    OUT.Col.w      = 0;
    OUT.uv         = float4( uv.x, uv.y, pos.z, 1);

    return OUT;
}
```

像素程序必须确定替用物纹素的真实深度，并忽略任何不可见的纹素。如果纹素缩放后的 z 值大于 1，它就不可见。当前的 pixel shader 不支持像素流水线的中断，因此必须将这些不可见纹素的最终深度设置成足够大的值，让 z 缓冲区硬件忽略他们。pixel shader 带有常量 `depthScale`，它会缩放替用物的深度值。最后，缩放后的深度值会加到在替用物计算中用到的前裁剪平面的深度上。

```
void main( vertout IN,
           uniform sampler2D      impostorTexture,
           uniform float          depthScale,
           out float4             color          : COLOR,
           out float              depth          : DEPTH)
{
    float4 texCol= tex2D(impostorTexture, IN.uv.xy);
    if (texCol.a >= 1.f) {
        depth     = 1;
    } else {
        depth          = IN.uv.z + texCol.a * depthScale;
    }
    color.xyz      = texCol.xyz * IN.Col.xyz;
    color.a        = 1;
}
```

5.4.6 总结

在本文中，深度替用物被用来渲染植物。当物体移动的时候，这项技术的能力变得非常

显而易见，因为没有弹出的走样，没有明显的平面精灵旋转，等等。我们可以得到高等级的细节，因为对于一个平常树林的树冠，可感知的叶片数目可以很容易地超过一百万。

这里展示的算法可以用于车辆和人类比例的模拟，以便实时地渲染逼真的树木和树林。这项技术充分利用了当前图形加速卡上的可编程渲染流水线。

5.4.7　参考文献

[Microsoft04] Microsoft Corporation. *DirectX 9.0*. Available online at *http://www.msdn.com/directx*.

[NVIDIA02] NVIDIA Corporation. *C_g Language Toolkit*. Available online at *http://www.nvidia.com*.

[Puppo97] Puppo, E. and R. Scopigno. *Simplification, LOD, and Multiresolution— Principles and Applications*. Eurographics'97. Tutorial Notes, 1997.

[Remolar03] Remolar, I., M. Chover, J. Ribelles, and Ó. Belmonte. *View-Dependent Multiresolution Model For Foliage*, 370–378. WSCG 2003, 2003.

[Szijarto03] Szijártó, G. and K. József. *High Resolution Folaige Rendering for Real-time Applications*. Budmerice, Slovak Republic: SCCG, 2003.

[Szijarto032] Szijártó, G. and K. József. *High Resolution Foliage Rendering for Real-time Applications*. Budapest, Hungary: GrafGeo, 2003.

[Szijarto04] Szijártó, G. and K. József. *Real-time Hardware Accelerated Rendering of Forests at Human Scale*. Plzen, Czech Republic: WSCG 2004.

[Szirmay95] Szirmay-Kalos, L. *Theory of Three-Dimensional Computer Graphics*. Budapest: Akadémia Kiadó, 1995. Available online at *http://www.iit.bme.hu/~szirmay/book.html*.

5.5 无栅格的可控火焰

佛罗里达中心大学，计算机科学学院，Neeharika Adabala
nadabala@cs.ucf.edu

佛罗里达中心大学，计算机科学学院和电影与数字媒体学院，Charles E. Hughes
ceh@cs.ucf.edu

游戏场景中经常会有火焰：着火的物体/车辆/建筑，燃烧的火把、壁炉，等等。火焰是炙热的燃烧所产生的现象，有着紊乱的动作。计算机图形学中使用的火焰模拟技术通常利用了在栅格（grid）上解动态流方程组的方法。这些方法经常需要很大的计算量，不能在实时的情况下使用[Nguyen01]。其他一些方法[Wei02]可以在实时情况下工作，但它们是基于在三维栅格上的计算，这在栅格大小、解析度和位置的选择方面引入了明显的设计问题。例如，当风吹向火的时候会发生什么？栅格应该定义成容纳全部可能会包含火的区域，或者设计成跟随火焰移动吗？在后一种情况中，就需要知道火焰会到达的可能区域。同时，基于栅格的计算通常不能保证稳定性，它与所选栅格的解析度相关，而且应用这些方法很复杂。因此，基于栅格的火焰模拟需要在每个用到火焰的场景中有技巧地选择栅格大小、位置和解析度，那将是一个乏味的任务。

本章将展现一个用于给火焰建模的无栅格技术，它是基于一个随机的拉格朗日过程[Pope00]。在这个方法中，用于动态模拟的方程定义了每个粒子的轨迹。其结果就是，它们可以直接在连续的时间中求出每个粒子的位置。这种方法的随机本性使得计算相对稳定。

游戏场景中出现的大部分火焰都是散射的火焰，换句话说，火焰中的氧化剂和燃料没有预先混合过（不像燃料和氧化剂已经预先混合过的火炉所产生的稳定火焰）。燃烧的燃料或物体会蒸发，并在燃烧前接触到氧化剂。因为这个过程并不是均匀地出现，所以火焰会闪烁，并表现出“跳动”的行为特征。大部分现有方法并不允许我们捕捉到火焰的这个特有属性。本文用一个简化的方法来为闪烁的火焰建模，以增强真实感。Lamorlette 和 Forster [Lamorlette02]的工作做出了火焰中间歇的火苗区域，但是他们的模型被设计为用于一个离线产生的环境，而不是实时应用程序。在本章中，火焰的闪烁是通过建立一个火焰“总体熄灭（global extinction）”的简化模型来表现的。总体熄灭就是火焰的燃烧非常慢，以至于看不到火苗的时候。我们的技术也可以建立“火苗刷（flame brushes）”的模型，也就是出现在高速化学反应的地方的很明亮的火苗区域。

在游戏中给火焰建模的另一个问题是需要用参数来控制火焰的外观。本章中呈现的这种方法可以控制闪烁的速率，火苗高度和火焰中火苗区域的数量。除了模拟之外，还描述了一种使用可编程图形硬件的实时火焰渲染技术。

我们将在下一节介绍火焰模型，该模型有两个主要的方面：动力学的随机拉格朗日模型和伴随燃烧产生的化学模型。接下来详细描述这个模型使用的可编程图形硬件的渲染。然后用一些带有例子的讨论，演示该技术的能力。最后一节将对本章的内容进行总结。

5.5.1 建模火焰

在模拟火焰的时候，建模的关键方面是紊乱的动态和伴随的化学反应。

1. 动力学模型

热量在火焰中的传播可以用不可压缩的湍流来进行建模。定义这个湍流的方程是质量守恒方程，如方程 5.5.1 所示；

$$\nabla \cdot \mathbf{u} = 0 \qquad (5.5.1)$$

以及 Navier-Stokes 方程，如方程 5.5.2 所示。

$$\frac{D\mathbf{u}}{Dt} = -\frac{1}{\rho}\nabla p + \upsilon\nabla^2\mathbf{u} + \mathbf{F} \qquad (5.5.2)$$

其中，$\mathbf{D}\mathbf{u}/Dt$ 是实质导数 $\partial/\partial t + u\partial/\partial x + v\partial/\partial y + w\partial/\partial z = \partial/\partial t + \mathbf{u}\cdot\nabla$，$\mathbf{u}$ 是速度向量(u, v, w)，p 是压力，∇^2 是 Laplacian 算子，ρ 是密度，υ 是动粘滞率的系数，而 $\mathbf{F}$ 表示外力和质量力。

这些方程可以通过 *Eulerian* 方法求解，那是用于在栅格的固定点定义流的速度场的解法器，或者通过拉格朗日方法求解，那是用于一组在流中变化的粒子轨迹的解法器。在紊乱流的情况中，流的混沌特性使得定义栅格大小、形状、位置和解析度的问题很麻烦。同时基于栅格的技术需要非常了解流的期望行为。例如，栅格应该在外部风场的方向上移动一些，以保持在栅格点上的解适当。因为这些问题，我们选择了无栅格的拉格朗日方法。

当计算应用到实时模拟的时候，它们必须稳定。紊乱流是混沌的，而且对初始条件的小变化非常敏感。因此，不能保证计算的稳定性。然而，流对初始条件小变化的敏感性使它适合于随机的模型。这里使用了随机拉格朗日方法保证计算的无栅格特性。在这种方法中，流体通过一组粒子来建模，它们的统计特性和基于流体方程变化的粒子相同，这些方法在数值上比方程的确定性解更加稳定。

粒子的紊乱行为通过使用随机拉格朗日方法来模拟。方程 5.5.3 到 5.5.5 定义了在模拟中第 i 个粒子的变化[Pope00]。

$$d\mathbf{X}^{(i)} = \mathbf{U}^{(i)}dt \qquad (5.5.3)$$

$$d\mathbf{U}^{(i)} = \frac{3}{4}C_0\langle\omega\rangle\mathbf{U}^{(i)}(t) - \langle\mathbf{U}\rangle dt + \sqrt{C_0 k\langle\omega\rangle}d\mathbf{W} \qquad (5.5.4)$$

$$d\omega^{(i)} = -(\omega^{(i)} - \langle\omega\rangle)C_3\langle\omega\rangle + \sqrt{(2\sigma^2\langle\omega\rangle\omega^{(i)}C_3\langle\omega\rangle)}dW^{*} \qquad (5.5.5)$$

方程 5.5.3 根据粒子的速度定义出它的位置。速度的计算是基于简化的 Langevin 模型，用于密度不变的静态各向同性的湍流。推导的细节超出了本章的范围，可以在 Pope 关于湍流的书中找到[Pope00]。〈〉中的项表现了括号内变量的局部平均值。在燃烧的粒子中，它们可以通过除以一

个栅格中的粒子所占用的区域来求出，并把在同一个栅格内的粒子看作第 *i* 个粒子。在我们的方法中，是使用一个 *kd* 树来存储粒子系统，并在第 *i* 个粒子的 *n* 个最近的邻域粒子上求这些局部平均值。我们使用的 *n* 的值一般落在 10 到 20 的区间内。这种把粒子存放在 *kd* 树的方法以前在[Adabala00]中介绍过，叫作粒子图方法。常量 C_0=2.1 是湍流模拟中使用的标准值，而 *k* 是紊乱运动能量。在本作品呈现的模拟中，*k* 的值是 1.5。**d** 和 **W** 表示各向同性的 Wiener 过程 $\mathbf{W}(t)$中的增量。它实现成由三个标准正态分布的样本组成的向量。下一个方程表示湍流频率的变化。常量 C_3 的值是 1，$\mathbf{W}^*$表示标量 Wiener 过程 $\mathbf{W}^*(t)$的增量，它独立于前一个方程中的 Wiener 过程。

前面的方程使我们可以建模火焰的紊乱运动，下面描述模型中火焰的化学方面。

2. 化学变化模型

化学变化模型模拟了随着反应的进程，燃料成分的改变。总体熄灭的建模需要在反映的过程中鉴别出这个阶段——无法再支持一个可见的火苗。在这个阶段，会发生总体熄灭。总体熄灭后，燃料和氧化的散射在继续，再次遇到燃烧的条件时火苗会重新出现。整个过程出现在一转眼的工夫中。因此不存在火苗的时刻是感觉不到的，出现的只是一个火苗的闪烁。计算机图形中一般的火焰建模方法集中于塑造火苗中温度的变化，而不会建模这个现象。

在燃烧的研究中，有许多给火焰的化学方面建模的方法。其中很多结果是基于火焰的经验研究[Drysdale99]。燃烧的模型和实际现象之间还是有很大的沟壑，因为做了很多简化的假设。例如，每种燃料都有它们自己惟一的燃烧方式，这取决于它的化学组成、燃料的扩散能力、氧化物和产生物。Subramaniam 和 Pope [Subramaniam98]提出的 Euclidian 最小生成树（EMST）混合模式是建立湍流的燃烧中，燃料成分变化的通用模型。这种方法可以和[Subramaniam99]中其他的建模燃烧的方法相比较，该比较是通过在一个简单的周期热化学模型中模拟成分的变化来完成的[Lee95]。这个模型的基本概念是相关的化学成分参数和燃烧过程中的粒子。粒子的成分一开始由周期的热化学模型定义。后来粒子成分的变化通过一个在成分空间中的 EMST 来构造，并通过考虑粒子在树中的邻域节点来更新成分。这个更新成分的方法有助于维护化学成分的局部性随着燃烧而发展。在视觉上，这使得我们有可能模拟火苗刷。在这种方法中，总体熄灭是通过计算反应过程变量的期望值，并和一个阈值比较的方法来实现的。如果这个值小于阈值，就出现总体熄灭。

我们将阐明一个简化的模型，模仿 EMST 模型的主要方面，以用于实时应用程序。在简化的热化学 EMST 模型中，平衡状态的粒子初始成分呈现在图 5.5.1 的实线中。然后成分随着时间变化，到达图 5.5.1 的点线描述的分布上。它朝着点线变化的程度取决于燃烧的本性。当燃烧稳定后，混合氧化物和燃料的时间正比于燃烧的时间。然而，当反映不稳定的时候（当有总体熄灭时），散射的时间明显比化学反应的时间长得多。结果，粒子成分的变化在图 5.5.1 中是点线。成分的准确分布可能会剧烈地变化，具体情况取决于定义了 EMST 混合模型的几个参数的值[Subramaniam99]。EMST 模型中成分变化的本质可以总结成图 5.5.1 中从实线到点线的移动，同时保证了在成分空间中的邻域。

图 5.5.1 中的 *x* 轴是混合分率 $\xi(X, t)$，而 *y* 轴是反应过程变量 $Y(X, t)$。其中，*X* 是粒子的位置向量(x_1, x_2, x_3)。反应过程变量是产物的质量分率，这里的化学反应被看成是燃料 + 氧化剂↔产物[Subramaniam99]。

在简化模型中，我们没有改变混合分率的值。因此，我们把它表示成 $\xi(X)$，去除了对时间的依赖。粒子的 $\xi(X)$值被定义成使梯度 $\partial\xi/\partial x_1$ 是常量，和[Subramaniam99]中的情况一样。

在我们的方法中，定义了一个常量作为参数 $\eta=(0.0,\infty]$。这个参数用来控制火苗刷的数量。在 x_1 值改变了一个单位的空间区域内出现的火苗刷的数量等于 η 的值。因此当 $\eta=1.0$ 的时候，这个空间区域（x_1 值变了一个单位）只有一个火苗刷。在我们的模型中，$t=0$ 的时候，$Y(X,t)$ 的值定义在方程 5.5.6 中。

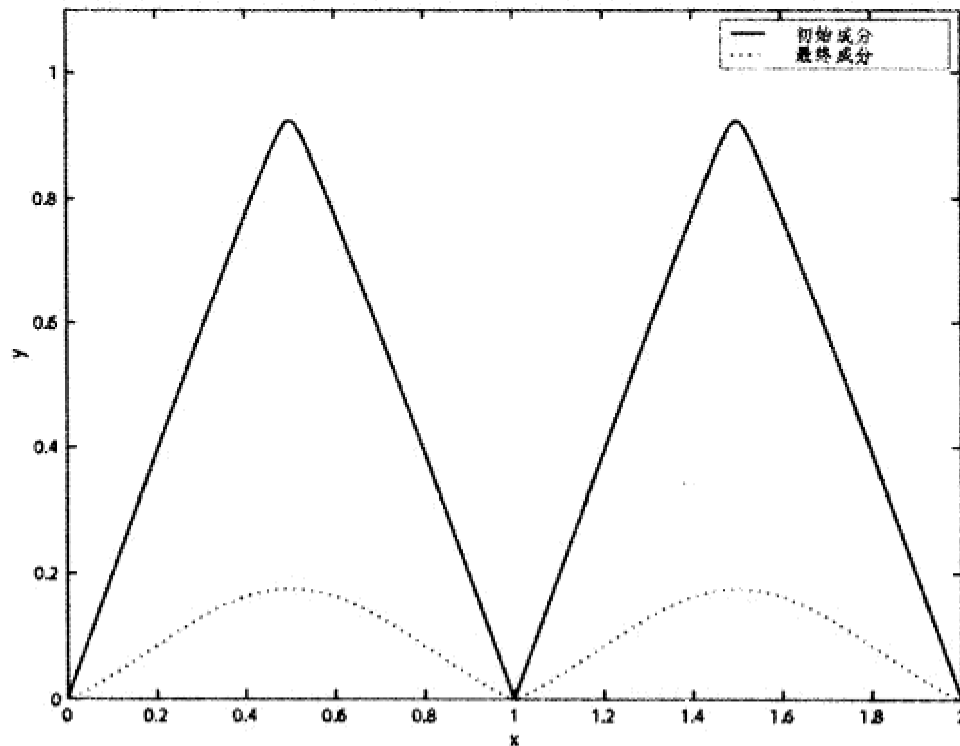


图 5.5.1　EMST 混合模型模拟中成分值的主曲线分布在最初和最终（在总体熄灭之前）的时间上。x 轴是混合分率 $\xi(X,t)$，而 y 轴是反应过程变量 $Y(X,t)$。

$$Y(\mathbf{X},t)=Y(\xi(\mathbf{X}))=\exp(-(\xi(\mathbf{X})-\lfloor\xi(\mathbf{X})\rfloor)-0.5)^2/\lambda \tag{5.5.6}$$

这是相邻重叠的高斯分布的表示。参数 $\lambda\in(0.0,\infty]$ 控制了两个相邻火苗刷之间的重叠。越小的值重叠越少，火苗刷分离得很好，而越大的值重叠就越多。图 5.5.2 给出了 $Y(X,t)$值的图，其中 $\eta=1.0$，$\lambda=0.8$。

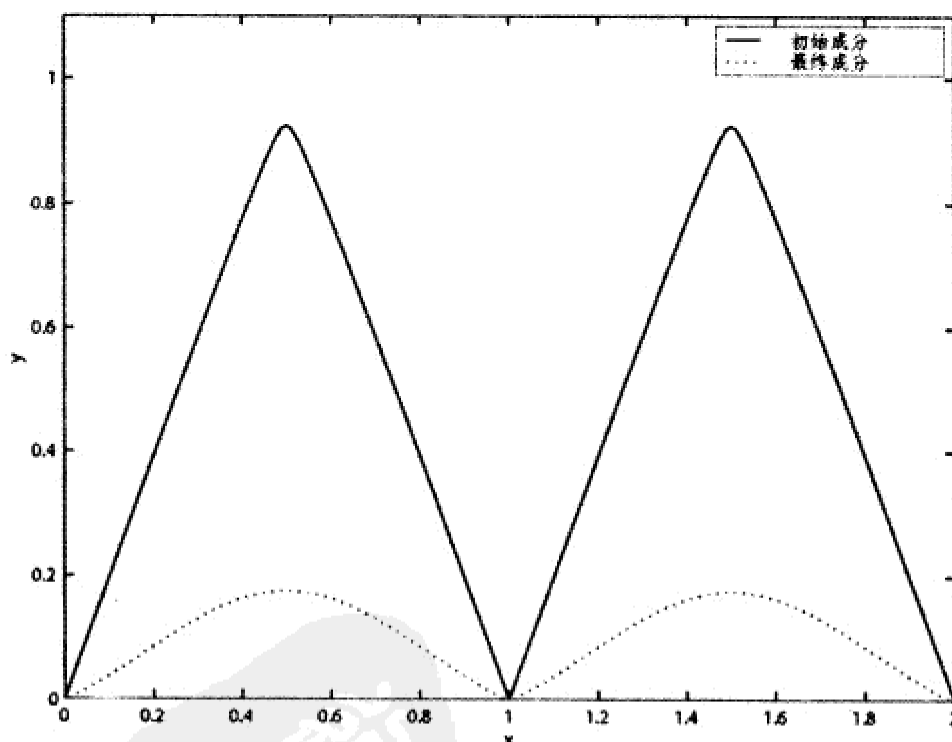


图 5.5.2　在我们模仿 EMST 混合模型的模型中，成分值的主曲线分布在最初和最终（在总体熄灭之前）的时间上。x 轴是混合分率 $\xi(X,t)$，而 y 轴是反应过程变量 $Y(X,t)$。

在我们的模型中，一开始是把粒子的成分按照方程 5.5.6 给出的高斯分布来散布，如图

5.5.2 所示；然后把成分变化到图 5.5.2 中的点线所表示的曲线上。之所以选择高斯分布作为初始分布，是因为在经验和数值上看，成分的分布会随着时间缓和至高斯。EMST 模型被表示成成分分布随着成分的连续更新缓和到高斯。在我们的简化方法中，是从模拟的第一步开始可视化火焰。没有使分布缓和到高斯的预处理模拟步骤。因此，成分的分布一开始就应该是高斯。这是通过把方程 5.5.6 用于初始化成分来保证的。

我们用方程 5.5.7 来改变成分 $Y(X, t)$。

$$Y(\mathbf{X},t) = (\chi + rd) * Y(\mathbf{X},t-1) \tag{5.5.7}$$

其中，χ 是反应过程变量下降的速率。$rd \in [0, 0.01 * \chi]$ 是对 χ 值的一个小的随机扰动。这个用来更新成分的简单方法类似于 EMST 混合模型的本质，相邻区域保持在成分空间中，并在外观相似的初始和最终曲线上有一个变化。χ 的值可以在区间 $(0, 1.0]$。我们发现 χ 的值在 [0.85, 1.0]区间中可以给出一个真实的外观。χ 的值趋向 1.0 会形成高的火苗，因为在可见区域的粒子反应过程在动态模拟中占了更多的时间。

在燃烧过程中，总体熄灭出现在总体反应过程不足以维持火苗的时候。在这个阶段，火苗的强度降低，并会在粒子的成分在热化学平衡（$Y(X, 0)$的值）时重新分布的下一个时间步中重新出现。在 EMST 模型中，总体熄灭的阶段是通过计算一个衰减序数来预测的，它与反应过程变量的期望值直接相关。衰减序数会和一个固定值做比较。如果序数较小，就会发生总体熄灭。在我们的简化方法中，会计算模拟中涉及的所有粒子的反应过程变量的平均值。如果小于一个阈值，就发生总体熄灭，然后我们用新的粒子和成分重新开始模拟，在总体熄灭后旧的粒子不再可见，如方程 5.5.8 所示。

$$\text{如果满足} \quad mean(Y(\boldsymbol{X},0)) < \theta \qquad \text{全部熄灭} \tag{5.5.8}$$

其中，θ 是可以调整以控制总体熄灭频率的阈值参数。改变阈值 θ 值的理由是不同的燃料会产生不同类型的火焰，而且根据燃料的不同，支持火苗的最小反应过程的值也不同。为了得到真实的外观，θ 的值应在区间[0.0, 0.4]之间选择。我们使用 $mean(Y(X, 0))$而不是反应过程值的和，来估计系统中的总体反应过程，所以阈值独立于模拟中涉及的粒子数量。

可以实时工作的一个闪烁火焰的模型就是利用本节描述的技术来实现的。下面，我们要描述一种用来渲染粒子系统的方法，它是根据前面呈现的模型演变而来的。

5.5.2 实时渲染

可编程图形卡用来渲染粒子系统，该系统是根据前一节呈现的模型演变而来的。特别的是，这种方法利用了渲染到 OpenGL p-buffer 的能力。

粒子系统是作为从粒子的当前位置延伸到它的前一位置的光条来渲染的。之所以采用这种方法，是因为当一个明亮发光的粒子高速移动的时候，我们由于视觉延迟会感觉到一个光条。成分参数用来定义线的纹理坐标。成分的当前值用作当前粒子位置的纹理坐标，前一个时间步的成分用作线的另一端。因为粒子成分和位置表现了在某个位置的一小块燃料的特性，所以这些线有一定厚度。这些线都会渲染到 p-buffer 中。在向上的方向上建立出模糊/光环，是为表现出燃烧生成的高温气体散发出的粒子光。为实现模糊效果，就用两个随机的纹理来获得纹理坐标的偏移。计算的结果会保存回曾经用做数据源的 p-buffer 中，以获得纹理坐标。这样就可以建立一个积累的模糊。

然后这张模糊的纹理就会用作纹理坐标，索引入一个表现了燃烧过程中的放射光变化的一维纹理中。

5.5.3　实例和讨论

这里的例子是用 C++和 OpenGL 实现的，运行在 Linux 操作系统的机器上。在 2.2 GHz Pentium4、768 MB 内存和 Geforce 5800 图形卡的机器，以及 1.46 GHz Athlon XP、512 MB 内存和 Geforce 5900 Ultra 图形卡的机器上，本算法都以每秒大约 60 帧的速度执行。所有图片和动画中使用的粒子数目都是 300。

图 5.5.3 演示了在两个总体熄灭阶段之间的一些火焰的图片。图片中的火焰以 $\eta = 2$ 生成，火焰的伸展是 x_1 方向上的两个单位。因此就有了四个火苗刷。λ 的值是 1，χ 是 0.97；阈值 θ 是 0.1。在区间 0.1 到 0.4 的 θ 值，外观看起来最好。

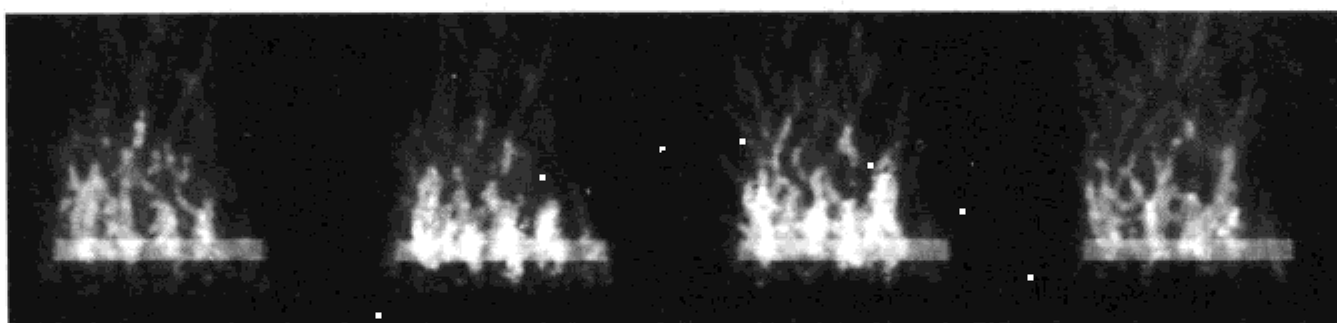


图 5.5.3　在两个总体熄灭阶段之间的火焰图片。在总体熄灭的两个时间实例之间还有几帧，这里没有连贯的帧

图 5.5.4 演示了用 $\eta = 1.0$ 和 $\eta = 2.0$ 生成的火焰。在两种情况下，λ 都选了 1.0。这形成了火苗刷之间的重叠，使火焰有更真实的外观。

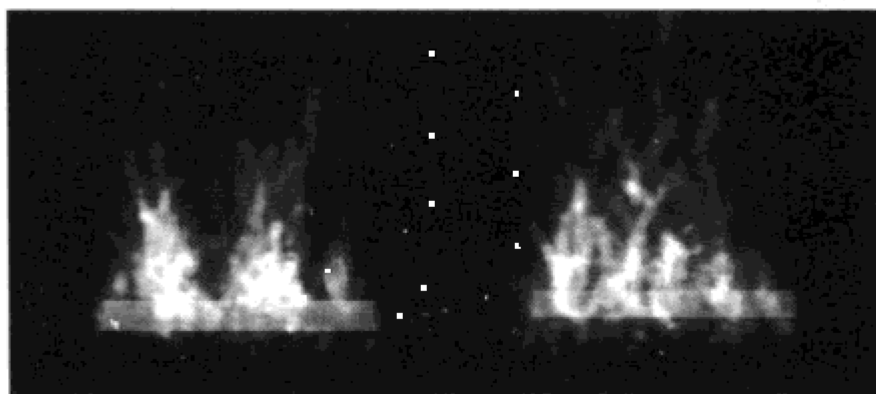


图 5.5.4　比较有不同数量火苗刷的火焰。左边的火焰有两个区域（$\eta = 1.0$），右边的有四个区域（$\eta = 2.0$）

图 5.5.5 演示了用我们的模型建立的不同高度的火焰。对于所有高的火苗，χ 的值都接近 1。高火苗经历的总体熄灭更少或根本没有。当 χ 的值比较小的时候，总体熄灭会频繁地出现。这和直觉一致——当火焰频繁地熄灭的时候，它必须从燃料源开始起火，在它再次熄灭之前不能传播到很高的地方。

在本实现中，粒子是通过在向上的方向上赋一个初始速度来引入模拟的，以表现出热浮力形成的初始向上速度。除了用参数 χ 控制之外，火苗的高度也可以通过在引入模拟的时候赋给的向上速度来控制。更高的初始向上速度会形成更高的火苗高度。这与一个事实一致——大火苗会出现在高速地注入了燃料或加入了氧化剂的时候。应该注意粒子会随着湍流变化（方程 5.5.3 到 5.5.5），不总是保证向上运动。当粒子明显向上移动的时候，会从系统中删除，且每次会删除

粒子后就会引入新的粒子。当粒子的成分减少到不再发出看得见的光时，这样的粒子就会从模拟中删除。在这些例子中，我们没有模拟烟。在火焰的顶端可以加入模拟烟的技术，比如[Lamorlette02]中描述的技术。在那种情况下，不再发出光的粒子可以引入烟的模拟系统。

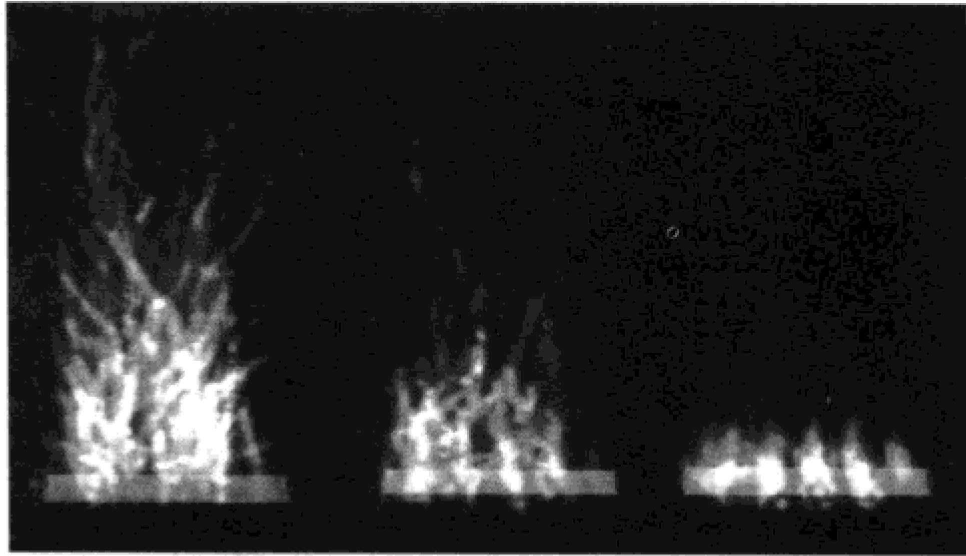


图 5.5.5 不同高度火焰之间的比较：左图以 x=0.99999 创建，中图以 x=0.97 创建，右图以 x=0.9 创建

这些图片是用固定的阈值 θ 创建的。这些例子表明我们可以用简单的直觉参数 η、χ 和 θ 来设计期望的视觉属性的火焰。

5.5.4 总结

本章呈现了一种在实时计算机图形应用程序中合成火焰的方法。本工作的特性包括：

- 一个无栅格的稳定数值模拟技术，用于基于随机 Lagrangian 方法解形式的湍流。
- 总体熄灭现象的模型，可以观察到火焰中的闪烁。火焰的这个属性在以前的计算机图形火焰模型中并没有出现过。
- 一个参数模型，闪烁速率、火苗高度或火苗刷的数量可以在模型中控制。
- 用于渲染火焰粒子系统的硬件加速技术。

这种动态技术是用粒子图的方式实现的，使得这种方法不需要栅格，因此克服了与栅格设计相关的问题，如栅格大小、栅格解析度和栅格在空间中位置的选择等等。这项技术的提议受到了基于物理和热化学模型的启示，但它是为计算机图形和游戏应用程序量身定做的，在这里，火焰的视觉控制要比物理正确性更加重要。

5.5.5 参考文献

[Adabala00] Adabala, N. and S. Manohar. "Modeling and rendering of gaseous phenomena using particle maps." *Journal of Visualization and ComputerAnimation*, 11:279–293, 2000.

[Bentley75] Bentley, J. L. "Multidimensional Binary Search Trees Used for Associative Searching." In *Communications of the ACM*, 18(9):509–517, 1975.

[Drysdale99] Drysdale, D. *An introduction to fire dynamics*. Chichester, New York: John Wiley and Sons, 1999.

[Lamorlette02] Lamorlette, A. and N. Foster. "Structural modeling of natural flames." *Proceedings of ACM SIGGRAPH 2002*, pages 729–735, July 2002.

[Lee95] Lee, Y. Y. and S. B. Pope. "Nonpremixed turbulent reacting flow near extinction." *Combustion and Flame*, 101:501–528, 1995.

[Nguyen01] Nguyen, D. O., R. Fedkiw, and H. W. Jensen. "Physically based modeling and animation of fire." *Proceedings of ACM SIGGRAPH 2002*, 21:721–728, July 2002.

[Pope00] Pope, S. B. *Turbulent Flows*. Cambridge: Cambridge University Press, 2000.

[Subramaniam98] Subramaniam, S. and S. B. Pope. "A mixing model for turbulent reactive flows based on Euclidean minimum spanning trees." *Combustion and Flame*, 115(4):487–514, 1998.

[Subramaniam99] Subramaniam, S. and S. B. Pope. "Comparison of mixing model performance for nonpremixed turbulent reactive flow." *Combustion and Flame*, 117(4):732–754, 1999.

[Wei02] Wei, X., W. Li, K. Mueller, and A. Kaufman. "Simulating fire with texture splats." *IEEE Visualization 2002*, pages 227–237, August 2002.

5.6 使用公告牌粒子构建强大的爆炸效果

美国任天堂公司，Steve Rabin
steve_rabin@hotmail.com, steve@aiwisdom.com

爆炸效果在游戏中很常见，但是它们常常缺乏冲击力或强烈的感觉。本文解释了如何用公告牌（billboard）粒子建立一个令人印象深刻的燃料爆炸效果，该方法可以用于任何 3D 硬件平台，包括可移动设备。因为计算机和视频游戏控制台目前还没有强大到可以模拟真正的爆炸，或者重放预先录制的由上千个粒子组成的爆炸，而游戏中的爆炸必须模仿真实的东西。所以，我们的目标并不是直接模拟一个爆炸，而是表达出发生了强大爆炸的感觉。这需要仔细地近似和模仿。游戏中的爆炸必须看起来像真的爆炸，而且要夸大某个方面以突出这个特效。

本文中的爆炸效果由七种不同的粒子效果组成：最初的闪光、放射的火苗、白色的热核心、强烈的火球、散发的烟雾和碎片。这个效果演示在彩图 8A 和 8B 中，以及光盘上的几段视频中。虽然爆炸并不像其他任何类型的模拟，但是模拟爆炸和烟雾的研究产生了许多技术[Fedkiw01，Feldman03，Stam03]。

5.6.1 最初的闪光

当爆炸发生的时候，最开始的一刹那，观察者会被光线致盲。虽然这个特效在大部分视频游戏的爆炸中都没有体现，但这是对爆炸的剧烈和强大的一个关键暗示。

模拟最初闪光的一种方法是建立一个半透明的粒子，发出亮黄/亮白的光，并在边缘的地方完全衰减。一种有效的方法是把一个 Photoshop 镜头闪光放在纹理的 alpha 通道，如图 5.6.1 所示。纹理的颜色成分应该是实心的黄色/白色。

在爆炸的时候，粒子应该从爆炸的中心产生，并快速均匀地膨胀，充满整个屏幕。一开始是完全的透明，并快速地变强，在大约 0.3 秒之后，它应该快速地减弱，直到变成完全透明（在那个时候，它就会被销毁）。这个序列演示在彩图 8A 最左边的一列。

这项技术对于发生在天空的爆炸很好用，因为快速膨胀的闪光粒子会与地面、建筑和其他物体相交，公告牌粒子和场景几何体会发生熟悉的 *z* 缓冲区相交。如果这个人造效果是不可接受的，另一个选择是在整个渲染的画面中用后期处理特效来建立闪光。

PDG

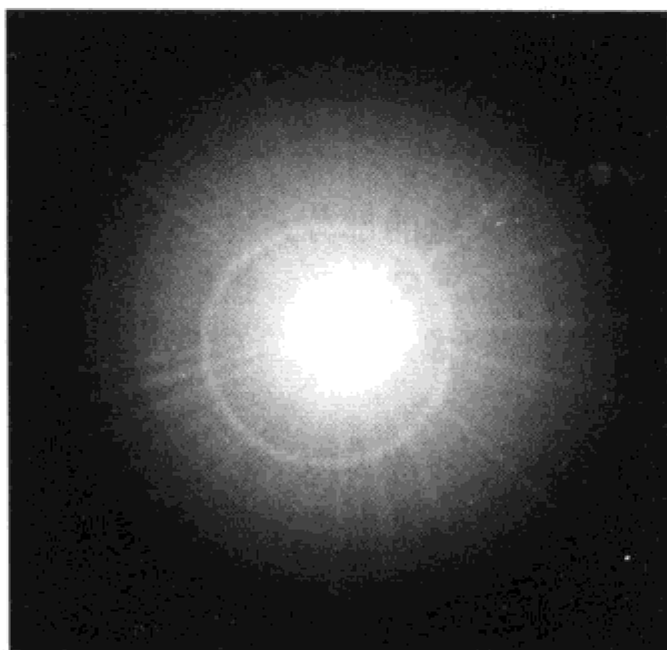


图 5.6.1　闪光纹理的 alpha 通道

5.6.2　放射的火苗

正如在图 5.6.1 中看见的，最初的闪光包含一些放射的火苗。然而，这项技术非常有效，以至于值得在爆炸中强调它。这个特效包括建立 10 到 30 个尖的火苗，随机地分布于爆炸的周围，如图 5.6.2 所示。

这些火苗并不会缩放，但是应该足够突出，以便很容易就能从最初的爆炸中分辨出它们，每一个的长度都是随机的。每条火苗都应该从爆炸开始后的 0 到 0.2 秒的时间随机出现。它们一开始应该完全可见，0.1 秒之后开始逐渐消失。注意如果火苗很长，或者比这消失得慢，那么爆炸就会更为卡通化。如果火苗很短而且快速消失，这个效果会在潜意识中非常有效地表达出爆炸的强度。这个序列可以在彩图 8A 的第二列看到。

每条放射的火苗公告牌都是一个窄的四边形，指向爆炸的方向，如图 5.6.3 所示。一个在 alpha 通道带有类似云彩斑点的实色纹理很适合于这些公告牌。一个好的颜色框架是火苗在中心处是浅灰色的，然后过渡到末梢的亮黄色。上色和梯度可以容易地通过一个单色的纹理和提供偏黄颜色的插值顶点颜色来实现。

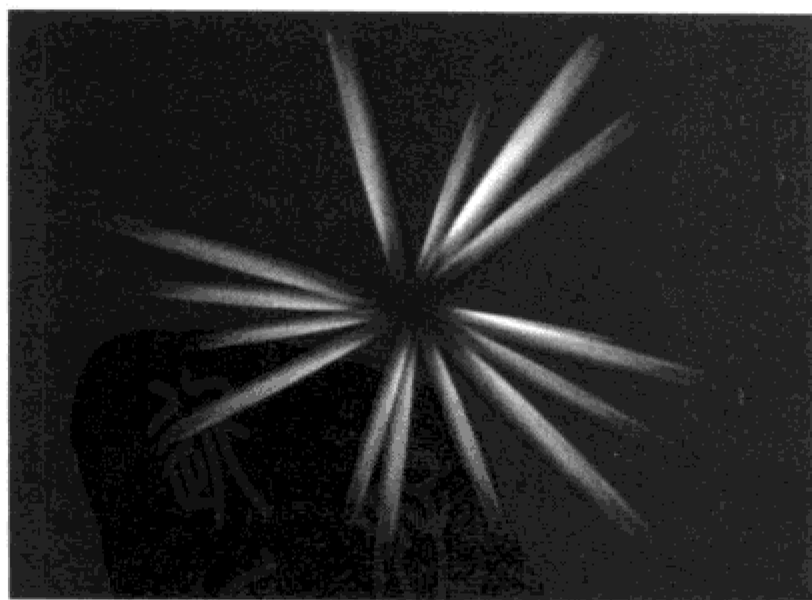


图 5.6.2　放射的火苗从爆炸中心随机地指向外部

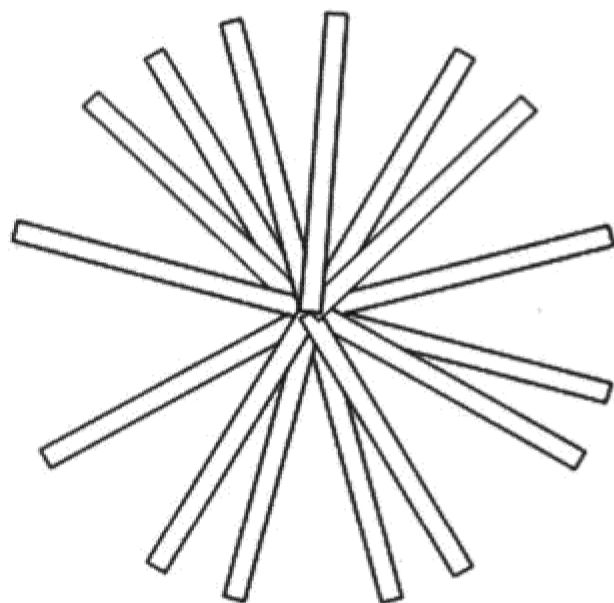


图 5.6.3　放射火苗的公告牌，由四边形组成

PDG

5.6.3 白色的热核心

白色的热核心是爆炸效果中非常关键的部分。它一开始是一个小的完全不透明的白环，边缘是黄色和橘红色，然后以非常高的速度扩散，扩散随着时间指数衰减，直到终止增长。在其最大的时候，白色的热核心是一团像云一样的白色、黄色和橘红色的物质，如彩图 8A 的第三列所示。

从建立的时候起，这个核心就应该保持大约一秒的完全可见，然后开始非常慢地减弱。这个效果是火球和烟雾效果的背景，后面马上会提到。随着烟雾和火球的扩散，这个白色的热核心会透过空的地方显示出来，为爆炸增添斑驳的外观。

渲染核心有两种方法。第一种方法对于 CPU 和 GPU 都是最廉价的，只需要一个粒子。第二种方法比较昂贵，需要大约 100 个粒子。正如我们将要看见的，游戏可以在廉价或昂贵的方法之间做出选择，以根据平台或场景的复杂度来调整性能。

廉价的方法是使用单张纹理，上面有乱七八糟的白色斑点，边缘附近环绕着非常薄的黄色和橘红色。这张纹理的 alpha 层应该有一个尖锐但平滑的边缘。粒子开始快速地膨胀（放大），然后指数减慢，直到停止增长。一秒之后它将慢慢消失。

昂贵的方法是用大约 100 个白色粒子，每个粒子的 alpha 通道都有类似云彩的斑点。通过把每个粒子的顶点都着色为黄/橘红色，然后用单独一遍混合来渲染它们，热的白色斑点就会呈现为重叠的粒子，过渡到黄/橘红色的边缘。一个有效的颜色框架是在右上顶点用明亮的黄/橘红色，右下顶点用中等的黄/橘红色，左上顶点用中灰色，左下顶点用暗灰色。这个颜色框架使得爆炸的不同卦限有不同的强度，所以核心看起来不那么均匀。

在昂贵的方法中，每个粒子在整个效果里面保持相同的大小。但是，粒子会随机地从中心高速地发射出来，并指数衰减，直到粒子不运动为止。一旦所有的粒子都停止运动，核心就到了它的全部大小。大约一秒之后，粒子开始慢慢消失。

在现实中，缓慢的爆炸因为空气中的热浮力上升很慢，然后因为风而偏移到一个给定的方向上。这些效果都可以实现——通过定义在水平方向上轻微吹拂的风力以及略微向上的分量来表现热浮力。风应该应用到所有热内核粒子上。这个小的特征有助于突出效果。

5.6.4 强烈的火球

火球是在前三种效果之后出现的第二级的效果。它在爆炸出现后大约 0.1 秒开始，并从爆炸的中心向外发射。火球刚开始应该完全透明，0.3 秒后慢慢消失（如彩页 8A 的第四列所示)。

和热核心一样，可以选择两种方式来渲染爆炸效果中的火球部分。第一种方法很廉价，只用到了一个粒子。第二种方法需要大约 50 个粒子，而且更昂贵。

廉价的方法使用的是一个乱七八糟斑点形状的纹理，类似于廉价的热核心步骤。纹理本身应该是一个斑驳的红火球，有着黑色的类似云朵的边缘，而 alpha 通道应该是一个类似云朵那样的致密和无定的形状。这个粒子基本上和热内核同时淡入淡出，膨胀的速度类似于指数衰减的速度。大约 0.3 秒后，这个粒子会慢慢消失。

昂贵的方法在爆炸的中心会产生大约 50 个粒子。每个粒子都是用类似于廉价效果中使

用的纹理，开始的时候非常小，然后慢慢膨胀到某个大小。每个粒子都以非常高的速度从中心发射到一个随机的方向。速度将指数衰减，直到粒子静止为止。在 0.3 秒之后，粒子应该缓慢消失。为了使火球更为斑驳，可以按照爆炸的卦限调暗粒子的顶点颜色。

昂贵方法的一个重要方面是在运动的方向上拉伸火球粒子。随着每个火球粒子从中心离开，它逐渐变成方形。这个效果对于建立从中心迸发出能量的感觉非常重要。图 5.6.4 演示了在粒子的生命期内，公告牌四边形是如何变换的。注意，随着粒子向外扩散和变慢，它们会变大、变方。

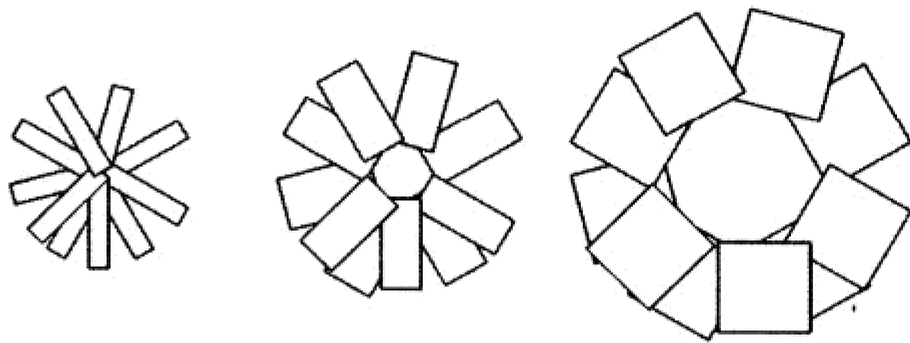


图 5.6.4　火球公告牌四边形在生命期的变换。注意一开始的时候四边形在爆炸方向上是窄的，然后慢慢变成正方形

5.6.5　散发的烟雾

散发的烟雾是另一个第二级的效果，出现在闪光、火苗和核心之后。它的出现时间基本上和火球一样，而且一般覆盖在前面所有效果的上面。从 0.1 秒开始，大约有 30 到 50 个烟雾粒子从爆炸中心发射出来（如彩图 8A 的第五列所示）。粒子的纹理应该是亮灰色的烟云，伴随着很多翻腾的细节。alpha 通道应该有类似云朵那样的致密和无定的形状。

烟雾效果对爆炸来说非常重要，因为它会散发到很远，而且逗留很久。没有廉价的方法来建立烟雾效果，因为它盖住了廉价的核心和火球效果，掩饰了它们的简单性。甚至对于昂贵的核心和火球，烟雾都需要很细致才能突出效果。很多游戏的爆炸都忽略了烟雾，但它是一个令人信服的效果。

随着烟雾从中心散发出来，其使用的基本技术和火球相同。每个粒子将在一个随机方向发射出来，速度非常高，并以指数衰减。每个粒子一开始应该在速度方向拉伸，然后随着时间变化成正方形，如图 5.6.4 所示。在 0.3 秒之后，烟雾开始非常慢地消失（比火球慢）。

在每个烟雾粒子的生命期内，它应该与核心和火球一样，受到风和热浮力的影响。然而，还有另一个效果可以增加良好的感觉。一般来说，烟雾有很多翻腾和有趣的扰动。这可以通过旋转烟雾粒子来描绘。其技巧是确定投射到屏幕空间的风向，然后旋转粒子来模拟轻微的旋转。在风向左边的所有烟雾粒子应该逆时针旋转，而在风向右边的所有烟雾粒子应该顺时针旋转，如图 5.6.5 所示。旋转应该非常慢，而且每个粒子应该有一个不同的随机指定的角速度。

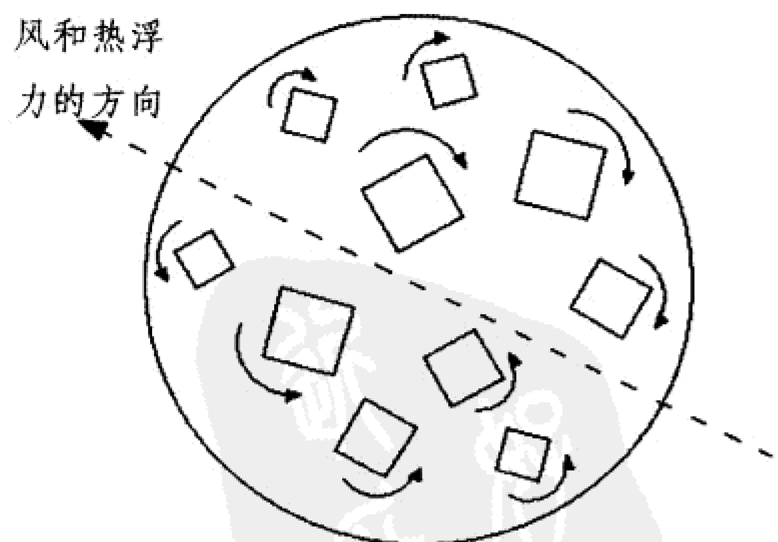


图 5.6.5　基于风和热浮力方向的伪旋转扰动

5.6.6 碎片

没有哪个爆炸会完全没有碎片飞出来。碎片可以是灰烬、泥土/金属的小黑块、小片着火的烟迹或整块燃烧的东西。碎片的初始速度应该非常高，而且不像其他气态效果那样衰减。取而代之的是，初始速度和重力会驱动它的动作。

暗的碎片非常有效，因为和它爆炸本身有很高的对比度。卡通化的外观可以通过让碎片块在爆炸发生后发出烟雾轨迹来做到。通常，这些烟雾轨迹发出的烟雾粒子从热白色开始，然后变成黄、橘红、亮灰，最后是暗灰。当然，这些烟雾轨迹甚至需要渲染更多的粒子，所以此效果需要明智地使用，以维持可以接受的性能。

5.6.7 效果表

前面的效果用到了很多精确的时间、速度和力。表 5.6.1 列出了每个效果和相关的数据，以便大家轻松地比较和重新建立爆炸效果。查阅彩图 8A 也有助于理解每个效果的精确时间。

表 5.6.1　每个效果使用的时间、速度和力的比较

效　果	粒　子	开始时间（秒）	出现	消　失	膨　胀	向外速度	其他力
最初的闪光	1	0.0	快	非常快	非常快	无	无
放射的火苗	10 到 30	0.0 到 0.2	立刻	快，0.1 秒之后	无	无	无
白色热内核（廉价）	1	0.0	立刻	非常慢，1 秒之后	快，指数衰减	无	风和热浮力
白色热内核（昂贵）	100	0.0	立刻	非常慢，1 秒之后	无	非常快，指数衰减	风和热浮力
强烈的火球（廉价）	1	0.1	立刻	慢，0.3 秒之后	快，指数衰减	无	风和热浮力
强烈的火球（昂贵）	50	0.1	立刻	慢，0.3 秒之后	很慢，直到变方形	非常快，指数衰减	风和热浮力
烟雾	30 到 50	0.1	立刻	非常慢，0.3 秒之后	很慢，直到变方形	非常快，指数衰减	风和热浮力
碎片	10 到 100	0.0 到 0.2	立刻	视具体情况而定	无	快，指数衰减	重力

5.6.8 额外的感觉

此前演示的公告牌效果会建立一个很好的爆炸，但是下面的额外感觉可以让爆炸更为生动。

1. 随机性

虽然本文呈现了大量的时间信息，包括开始时间、消失速率和速度，但是如果每个粒子是独特的，和其他粒子并不一致，那么整体效果就会更好。应该用随机性来调节所有这些时间、速率和速度，但是要把随机性保持在一个狭窄的区域内。因为大部分粒子需要类似的行为，所以随机性不该带来非常大的变化。

2. 屏幕震动

当出现了一个强烈的爆炸时，可以通过短暂地上下（不是左右）震动摄像机来增强感觉

强度。不是每个爆炸都会保证这样的效果，但近处或强烈的爆炸确实可以通过短暂的震动来增强。

5.6.9 效率问题

当很多爆炸效果同时出现的时候，会出现一些效率问题。接下来我们将解释如何处理三个这样的问题。

1. 控制粒子数量

在游戏中，一般很难控制一次发生多少个几乎同时的爆炸，因为无法预测玩家或 AI。因此，每个爆炸效果的开销应该相对小，因为积累的开销可能会快速地暴涨。不仅在每一帧必须更新每个粒子，而且所有爆炸发出的粒子都必须在 CPU 上排序，以便正确地在帧缓冲区混合。随着绘制的粒子越来越多，排序开销会指数级地增长。

本文描述的爆炸效果演示了如何做一个廉价的白色热内核和一个廉价的火球，目的是为了限制粒子数量。廉价的爆炸效果会使用大约 43 个粒子（不包括可选的碎片）。昂贵的爆炸效果会使用大约 230 个粒子（不包括可选的碎片），而且有很多过度绘制，这对 GPU 来说可能很昂贵。

选择使用廉价或昂贵的爆炸是限制粒子数目的方法之一，但是一个同样重要的技术是给予绘制的粒子数量一个强的限制，当快要到达限制的时候就重复使用最老的粒子。例如，1000 个粒子的限制可能会让第 5 个重复的爆炸开始重复使用第 1 个爆炸的粒子。因为第 1 个爆炸可能完成了它的生命期，并已经消失了，删除这些老的粒子一般不会被发现，因为新的粒子非常吸引视线。

2. 优化公告牌朝向

在游戏过程中，经常可以看到半打的同时爆炸，每个爆炸都由上千个粒子来表现。为了面向摄像机，每个粒子在每一帧必须重新计算它的朝向。这意味着一系列的计算，但它不是必要的，甚至是不可取的。

在一个爆炸中，如果每个粒子根据它的中心点来面向摄像机，那么粒子将会互相交叉，并造成很丑的走样。因此，一个爆炸中的粒子必须全都朝着相同的方向。解决方案是让每个产生的爆炸，都有一个中心点在生命期内随着爆炸移动（受到风和热浮力的影响）。这个朝向可以在每一帧对每个爆炸都计算一次。所有属于一个爆炸的粒子都是用同一个朝向，这就把朝向计算的数量从上千个减少到了一打。

3. 在帧速率的约束下给粒子排序

因为有上千个潜在半透明的粒子，所有粒子都必须按照与摄像机的距离排序，以便能正确地渲染。不幸的是，这个任务落在了 CPU 身上，所以效率是一个主要问题。

众所周知，快速排序算法对这种类型的排序是比较理想的，平均花费为 $O(n \log n)$。但是，每一帧中快速排序的花费的时间会野蛮地涨落，在最差的情况下甚至会升到 $O(n^2)$。这对于试图维持一个固定或基本固定帧速率的游戏来说是个大问题。

一个解决方案是使用可以在花费指定的时间量之后停止的排序算法，由此来保证它不会花费太长时间。显然，这会造成一些走样，因为在每一个渲染帧，不是所有的粒子都适当地排序了，但这是一个可以接受的让步。选择的算法必须有这样的递增特性：一旦排序中止的时候，列表必须比刚开始的时候更为有序。因为爆炸封闭在小的区域中，而且不会快速移动，这样的有序粒子列表将会在帧与帧之间相对不变。

一个可以解决这个问题的排序算法是著名的冒泡排序。大家都知道这个非常简单的排序算法性能很差，但是它有两个很好的属性。首先，它可以稳定地给一个列表排序，而且可以在任何时候中断，并保持列表的完整和部分有序。其次，如果列表有序，它可以很早退出，代价仅为 O(*n*)。因此，冒泡排序算法可以封顶，例如，从不占用多于 3%的帧时间。

5.6.10 总结

本文呈现的组合爆炸效果详细地讨论了一种类型的爆炸，它完全是用公告牌粒子来表现的。因为有很多类型的爆炸，所以必须仔细地调节和创造来为特定的游戏描绘我们感兴趣的爆炸，但愿本文呈现的技术可以派上用场。关键是要研究试图重新建立的爆炸类型。为此，可以查看来自 Internet、电影和军事文档的参考材料。

ON THE CD

很多游戏的爆炸专注于白色热内核或火球，但是可以通过考虑其他效果，比如闪光、烟雾和摄像机震动来增强爆炸的感觉。本文的关键革新之一，如图 5.6.4 所示，是用了火球和烟雾。让火和烟雾从中心爆炸出来能造成强大和猛烈的感觉。没有它，火球和烟雾就好像只是从中心快速地移开的静态喷烟。这个差别可以很容易地从光盘上的两个例子影片中看出来。文件 explosion1.mpg 使用了尖的火球和烟雾技术，而 explosion2.mpg 没有使用。光盘上的其他爆炸影片演示了不同的变化。

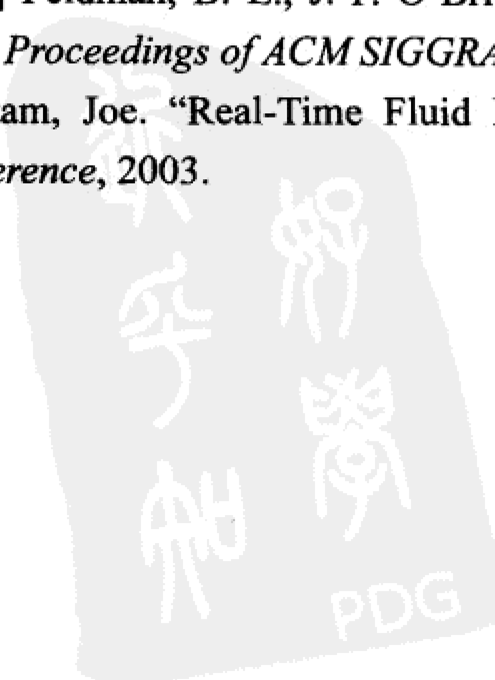
为游戏建立好的爆炸是在有限的硬件上设法获得最好效果的平衡艺术。我们必须尽可能地利用编程的巫术和艺术的创造性，直到可以模拟或重放有上万个粒子的爆炸为止。

5.6.11 参考文献

[Fedkiw01] Fedkiw, R., J. Stam, and H. W. Jensen. "Visual Simulation of Smoke." *The Proceedings of ACM SIGGRAPH*, 2001.

[Feldman03] Feldman, B. E., J. F. O'Brien, and O. Arikan. "Animating Suspended Particle Explosions." *The Proceedings of ACM SIGGRAPH*, 2003.

[Stam03] Stam, Joe. "Real-Time Fluid Dynamics for Games." *Proceedings of the Game Developers Conference*, 2003.


PDG

5.7 渲染宝石的简单方法

ATI 研究院公司，Thorsten Scheuermann
thorsten@ati.com

很多游戏需要玩家找到或赚取财宝来推进游戏环境。本文讨论了一项渲染宝石的技术，以便用赏心悦目的宝石来奖励成功的财宝猎人玩家。

宝石那漂亮而复杂的外观主要是由于其反射指数极高的透明材质。当光线穿越宝石的时候，就会出现彩色散射和全反射。

本文的宝石渲染技术用在了 ATI 的 Radeon X800 的 demo *Ruby: The Double Cross* 中（参见图 5.7.1）。

图 5.7.1 来自 ATI 的 demo Ruby: The Double Cross 的截图，演示了宝石渲染技术。
© ATI Technologies, Inc. 2004.

5.7.1 技术概览

该宝石渲染技术组合了用于光线穿过宝石的光照项、反射（使用立方环境映射）和镜面高光。对于光能传递项，我们会分别渲染宝石的背面和正面。反射和镜面高光只计算几何体的正面。

宝石的外观主要来自于穿过宝石和因为全反射而在内部反弹的光线，但要正确地模拟这些光线会很昂贵。[Guy04]中描述的宝石渲染技术以交互帧速率相当正确地模拟了宝石中的光能传递，但是它的性能对于当前硬件上的游戏情景来说是不可接受的。我们使用“折射”cubemap 来计算光能传递项的一个简单近似。多个来自这张 cubemap 的样本累加起来，就可以形成宝石中全反射和多次光反弹的外观。

PDG

最后，为了让宝石看起来很闪亮，我们会在它最亮的区域用屏幕对齐的公告牌渲染一些光斑。

5.7.2 法线和 cubemap 采样问题

切割的宝石有小平面和尖锐的边缘，这形成了着色的间断。但是，当使用没有共享顶点的宝石几何体，并把顶点法线设置成面法线的时候，反射和折射的视向量在每个面上不会改变太多。当用这些向量查找反射和折射 cubemap 的时候，就会只采样到 cubemap 中的小区域，并在宝石面中被放大（见图 5.7.2（a））。使用平滑的顶点法线会改进 cubemap 采样的覆盖率，但是沿着面边缘的着色间断消失了（见图 5.7.2（b））。作为折衷，我们的宝石几何体同时包含面和平滑的法线。对于反射和折射向量的计算，我们使用两个法线的平均，其结果是面中的法线插值改变速率比较合理。这就在改善 cubemap 的采样覆盖率的同时，仍然保持了边的不连续性（见图 5.7.2（c））。

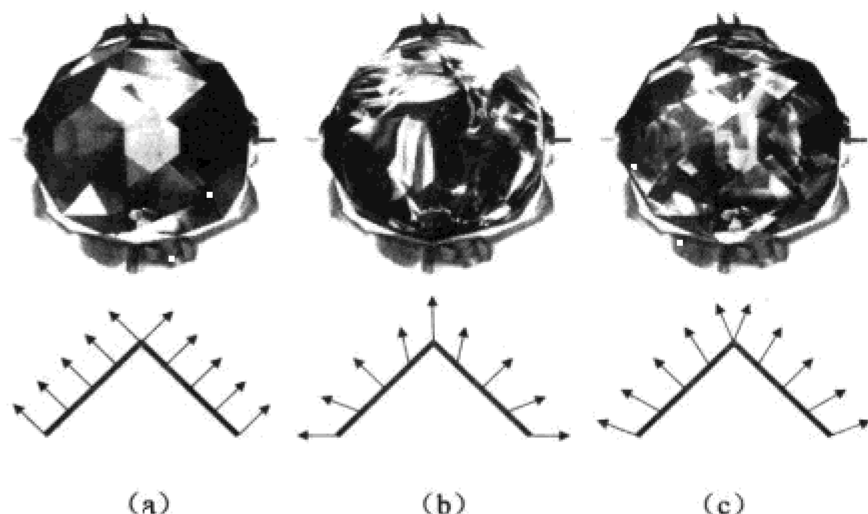


图 5.7.2 使用不同法线的视觉效果：（a）面法线，（b）平滑的法线，（c）平均的法线

5.7.3 传递的光能

为了计算光能传递项，我们使用的是一个非常简单的预计算近似形式：一个可以考虑全反射的离线渲染器通过光线跟踪生成一个从宝石内部向外看的 cubemap。图 5.7.3a 演示了用于本文截图的折射 cubemap。它是在 Maya 中用光线跟踪生成的，递归深度被设为四次反弹(bounce)。离线渲染器中的光照环境是用一张环境贴图来近似的。虽然宝石几何体很简单，但是 cubemap 捕捉到了很多视觉复杂性，因为光线在穿过宝石的时候路径很复杂。

宝石的 pixel shader 会从折射 cubemap 执行两次查找，并累加它们。宝石的背面和正面在不同的通道渲染，正面是使用相加的混合，所以最后一共有四次的折射 cubemap 样本累加到最终的图像上。图 5.7.4 演示了光能传递项是如何计算的。

对于每一个通道，用来查找折射 cubemap 的向量是用两个不同的折射率计算的向量。另外，第 2 个折射向量会融合一个逐面的随机方向的反射，以及一个额外的任意选择的混淆。随机反射方向通过查询一个包含随机值的 1D 纹理得到，纹理坐标在 vertex shader 中是根据模型空间中的面法线计算的。

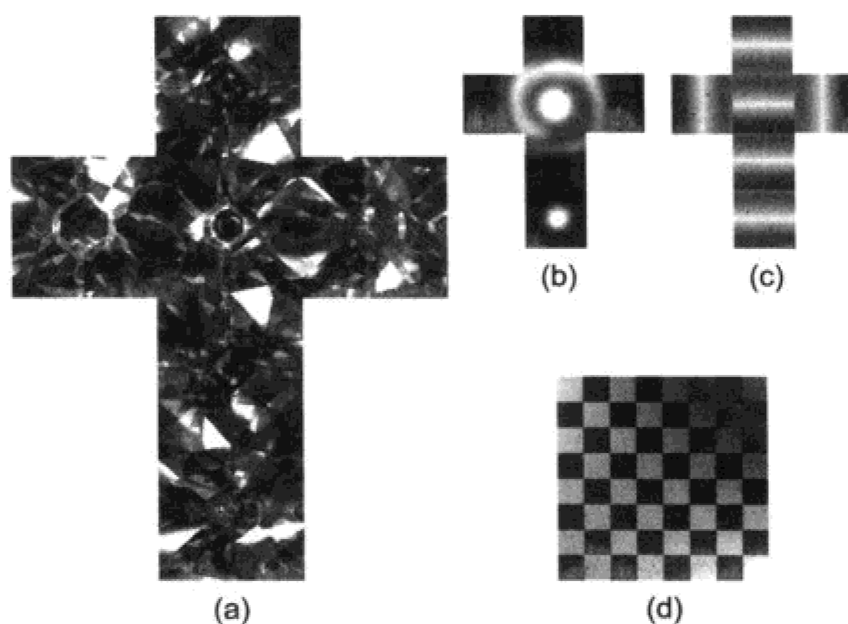


图 5.7.3　本例中使用的纹理：（a）折射 cubemap，（b）环境 cubemap，（c）彩虹 cubemap，（d）边缘图

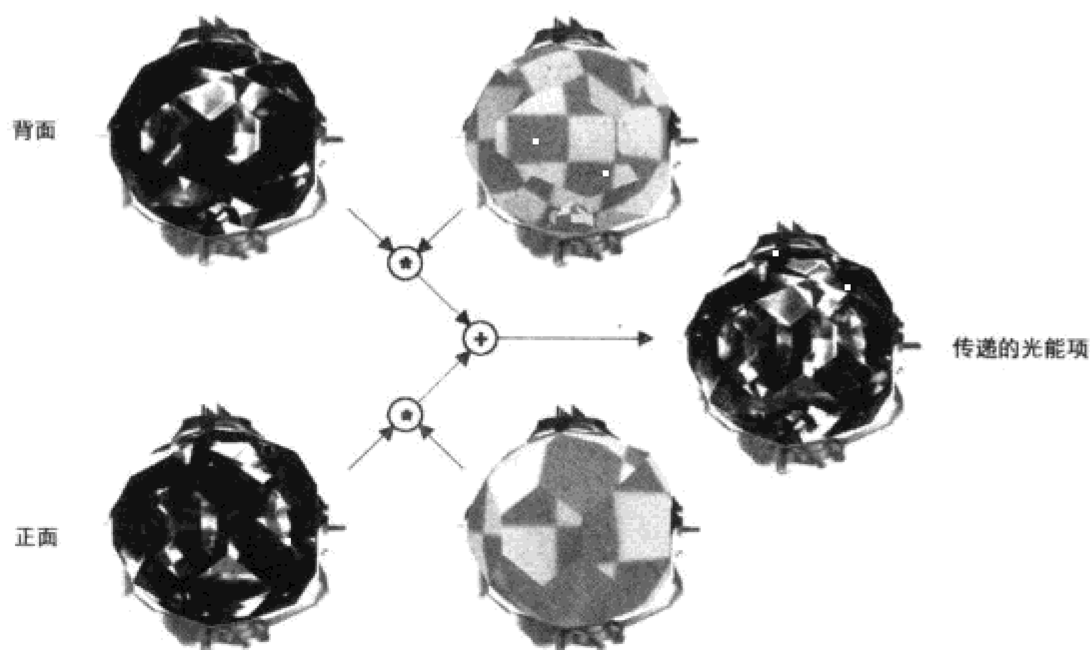


图 5.7.4　计算光能传递项的步骤分解

```
rndTexcoord = dot(N_face_model, float3(1, 1, 1));
```

半随机的反射和混淆使得两个向量会查询折射 cubemap 的不同区域，这就导致了传递项更复杂的外观。渲染两遍，而不是只在几何体的正面执行四次 cubemap 查询有一个优点，即背面和正面在计算中会用不同的法线，这就导致 cubemap 中有更多样的采样位置。

另一种增加视觉复杂性的方法是把一个有尖锐边缘的纹理（“边缘图”）映射到宝石的几何体上，并用传递项调整它（参见图 5.7.4）。在我们的例子中，使用的是一个简单的彩色棋盘模式（参见图 5.7.3d）。边缘图的颜色有助于得到彩色散射的外观（由于折射，光线撕裂成光谱）。这个效果的强度可以通过混合边缘图和白色的参数来控制。

这是用于光能传递项的 HLSL 函数。

```
sampler tRefraction; // 折射 cube map
sampler tEdge;       // 边图
sampler tRandom;     // 带有随机 RGB 值的 1D 纹理

float3 TransmissionTerm (float3 N_curved,    // 平均法线
```

```
                              float3 V,          // 视向量
                              float2 edgeUV,
                              float rndTexcoord,
                              float brightness,
                              float edgeStrength)
{
  // 计算折射向量
  float3 vTransmission1 = refract(V, N_curved, 2.4);
  float3 vTransmission2 = refract(V, N_curved, 1.8);

  // 用随机指向每个面的单位向量反射第 2 个向量
  // rndTexcoord 在 vertex shader 中计算;根据模型空间中的面法线
  float3 rnd = tex1D(tRandom, rndTexcoord);
  rnd = normalize(2.0 * rnd - 1.0);
  vTransmission2 = reflect(vTransmission2, rnd);

  // 查找折射 cubemap 并应用 gamma
  float3 cRefract = texCUBE(tRefraction, vTransmission1);

  // 再次查找，用额外的"随机性"来摇动这个向量
  // "randomness"
  cRefract += texCUBE(tRefraction, vTransmission2.yxz);

  // 把 gamma 曲线应用到 cubemap 上，得到亮的区域
  // （这可以放入 cubemap 中)
  cRefract = pow(cRefract, 4.0);

  // 边缘项
  float3 edge = tex2D(tEdge, edgeUV);
  edge = lerp(1.0, edge, edgeStrength);

  // 调制上边缘项，并缩放总体亮度
  return cRefract * edge * brightness;
}
```

5.7.4 反射

宝石 shader 的反射项组合了来自点光源的镜面高光和来自环境 cubemap 的反射，后者是通过一个 Fresnel 项来调制的。为了增加散射效果，我们使用一张每个面包含一个彩虹颜色梯度的 cubemap，并用环境贴图调制它（参见图 5.7.3）。这和 Fresnel 项一起，在宝石的边缘附近产生了变色现象（参见图 5.7.5e 的最终结果）。与来自边缘图的散射一样，这个效果的强度可以通过彩虹 cubemap 的亮度来控制。

这是用于计算反射项的 HLSL 代码。

```
sampler tEnvironment; // 环境 cubemap
sampler tRainbow;     // 彩虹 cubemap

float3 ReflectionTerm (float3 N_curved,       // 平均法线
                       float3 N_face,        // 平面法线
```

```
                        float3 V,              // 视向量
                        float3 L,              // 光向量
                        float3 lightColor,
                        float shininess,        // 镜面系数
                        float dispersionStrength)
{
  // 反射向量
  float3 R_face = reflect(V, N_face);
  float3 R_curved = reflect(V, N_curved);

  // 镜面高光
  float RdotL = clamp(dot(R_face, L));
  float3 specular = pow(RdotL, shininess) * lightColor;

  // Fresnel 项近似
  float fresnel = pow(1.0 - clamp(dot(N_face, V)), 2.0);

  // 查找环境图
  float3 cEnv = texCUBE(tEnvironment, R_curved);
  float3 cRainbow = texCUБE(tRainbow, R_curved);

  // 用 fresnel 项和散射来调制环境图
  cRainbow = lerp(1.0, cRainbow, dispersionStrength);
  cEnv = cEnv * cRainbow * fresnel;

  return saturate(specular) + cEnv;
}
```

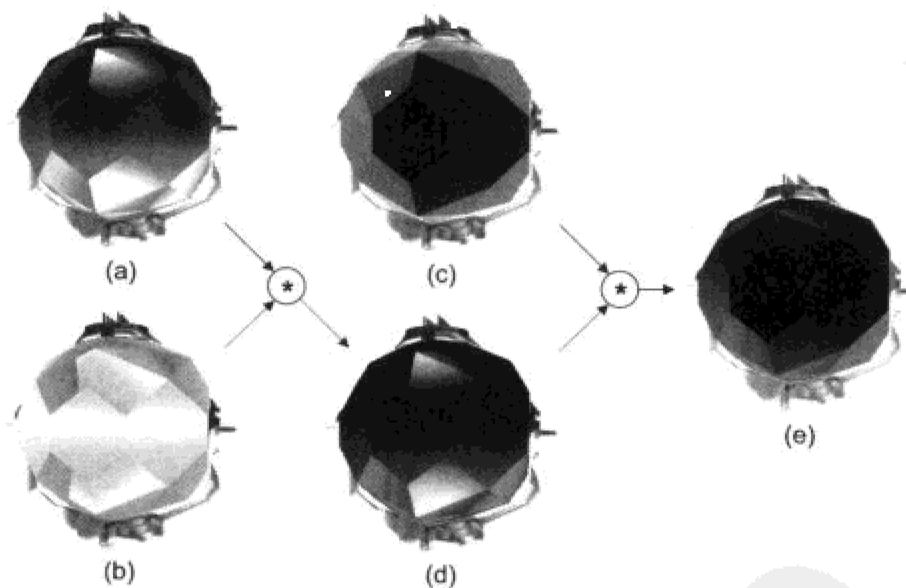
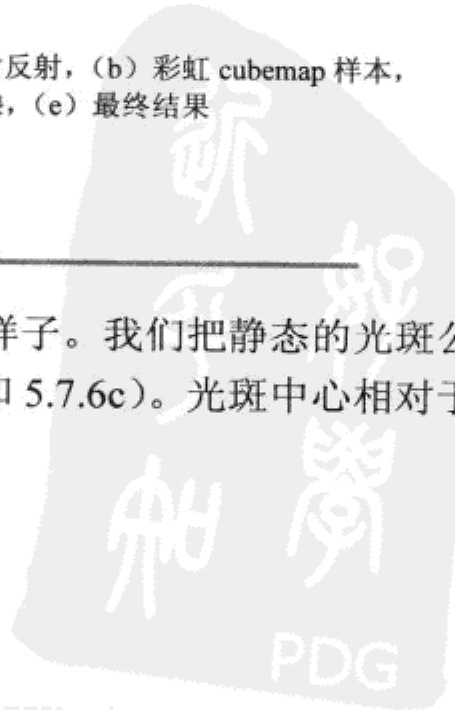


图 5.7.5　计算带散射的反射的分解步骤。(a) 环境映射反射，(b) 彩虹 cubemap 样本，(c) Fresnel 项，(d) (a) 和 (b) 相乘，(e) 最终结果

5.7.5　光斑

在宝石最亮区域渲染光斑可以得到一个更灿烂的样子。我们把静态的光斑公告牌几何体放在宝石的表面上，并遍及它的内部（参见图 5.7.6b 和 5.7.6c）。光斑中心相对于宝石保持固定，而光斑的几何体在 vertex shader 中执行屏幕对齐。


PDG

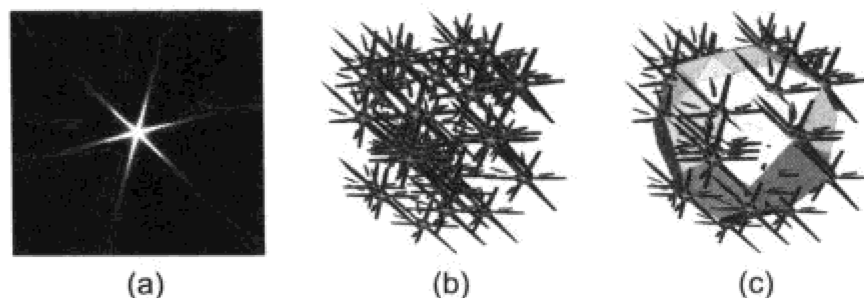


图 5.7.6 （a）光斑几何体，（b）和（c）光斑在宝石上的分布

光斑用相加混合的方式渲染到最终画面的上面，作为后期处理效果。在渲染光斑之前，帧缓存的内容必须拷贝到一张纹理中，以便可以在光斑的 pixel shader 中访问它们。光斑强度在 shader 中选择，取决于帧缓存中光斑中心位置的亮度。如果亮度小于一个阈值，就会丢弃这个光斑的所有片元。这使得光斑只出现在宝石中最亮的点上。此外，它们将随着视角和宝石位置的改变而快速地出现和关闭，这就掩盖了光斑相对于宝石的固定位置。另外，为了得到更闪亮的外观，光斑强度可以由一个噪声值调制，这取决于光斑的屏幕空间位置和当前视角。

渲染大量光斑，其中大部分会随时消失，这样可能很昂贵，因为存在高度的重复绘制。为了减少填充开销，最好让建立的光斑几何体紧密地包围光斑纹理的不透明区域，如图 5.7.6（a）所示，这比简单的公告牌方块的性能更好，因为简单的公告牌方块有大量的空白区域。

对于可以正确工作的公告牌 vertex shader，属于光斑的所有顶点必须有相同的顶点位置，位于光斑中心。光斑形状通过 2D 纹理坐标来确定，如 shader 代码所示。

下面是光斑的 vertex shader。

```
float4x4 mWorldViewProj;
float4x4 mWorld;
float4x4 mView;
float3 worldCamPos;
float flareRadius;

struct VsInput
{
  float4 pos      : POSITION0;
  float2 uv       : TEXCOORD0;
};

struct VsOutput
{
  float4 pos          : POSITION0;
  float2 uv           : TEXCOORD0;
  float2 noiseUV      : TEXCOORD1;
  float2 screenUV     : TEXCOORD2;
};

VsOutput main (VsInput i)
{
```

```
    VsOutput o;

    // 屏幕对齐的公告牌几何体和变换。注意一个光斑的所有顶点必须都设到光斑的中心位置。光斑的形状由纹理坐标来确定
    float2 pos2D = i.uv - 0.5;
    float4 pos = i.pos + (pos2D.x * mView[0] +
                          pos2D.y * mView[1]) * flareRadius;
    o.pos = mul(pos, mWorldViewProj);

    // 计算光斑中心的屏幕空间位置
    float4 flareCenterPos = mul (i.pos, mWorldViewProj);
    o.screenUV = flareCenterPos.xy/flareCenterPos.w;
    o.screenUV.y = -o.screenUV.y;
    o.screenUV = 0.5 * o.screenUV + 0.5;

    // 到光斑中心的视向量
    float3 V = normalize(worldCamPos - mul(i.pos, mWorld));

    // 传递纹理坐标
    o.uv = i.uv;

    // 根据位置和视向量计算一些"随机"纹理坐标，用来在 pixel shader 中查找噪声纹理
    o.noiseUV.x = fmod(abs(dot(pos.xyz, float3(1, 1, 1))), 2.0);
    o.noiseUV.y = fmod(abs(2.0 * dot(V, float3(1, 1, 1))), 2.0);

    return o;
}
```

Following is a flare pixel shader:

```
sampler tFlare;       // flare texture
sampler tNoise;       // 2D noise texture
sampler tScreen;      // back buffer contents

float flareIntensity;

struct PsInput
{
  float2 uv       : TEXCOORD0;
  float2 noiseUV  : TEXCOORD1;
  float2 screenUV : TEXCOORD2;
};

float4 main (PsInput i) : COLOR
{
  // 采样光斑纹理
  float fAlpha = tex2D(tFlare, i.uv);

  // 给光斑强度一个随机值
  float noise = tex2D(tNoise, i.noiseUV);
```

```
    noise = lerp(0.6, 1.0, noise);

    // 获得光斑中心的屏幕亮度
    float3 cScreen = tex2D(tScreen, i.screenUV);
    float lum = dot(cScreen, float3(0.3, 0.59, 0.11));

    // 如果亮度小于 0.8, 就丢弃片元
    clip(lum - 0.8);

    // 把可见光斑的亮度拉到[0, 1]区间中
    // 并应用亮度
    lum = smoothstep(0.8, 1.0, lum);
    lum *= lum;

    float4 o = 0;
    o.rgb = noise * lum * fAlpha * flareIntensity;
    return o;
}
```

5.7.6 总结

本文描述了一项渲染宝石的技术，在当前的图形硬件上执行得很好。这种方法在很大程度上忽略了物理真实性，取而代之的是注重有趣的外观，可以适用于典型的游戏环境。光能传递使用查找与计算的折射 cubemap 来近似。反射项是反射 cubemap 和镜面光照的组合。彩色散射的外观由简单的多个纹理混合来建立。宝石上的光斑在后渲染处理中（post processing pass）通过公告板（bill board）来渲染。

我们的渲染技术使用了可以应用到其他场景的技巧：平均面和平滑法线可以用于着色有尖锐边缘的几何体。光斑的后期处理技术可以泛化到覆盖整个图像，并用于特殊的效果。最后，包围粒子纹理非空区域的复杂粒子几何体——和用于光斑的一样——有助于改善受限于填充的粒子系统。

5.7.7 参考文献

[Guy04] Guy, Stephane and Cyril Soler. "Graphics Gems Revisited." *ACM Transactions on Graphics (Proceedings of the SIGGRAPH conference)*, 2004.

5.8 体积化的后期处理

A2M 公司，Dominic Filion
dfilion@hotmail.com
摩托罗拉公司，Sylvain Boissé
sylvainboisse@hotmail.com

随着可编程图形硬件的普及，我们可以看到越来越有趣的自定义后期处理效果。像素偏移、闪光和辉光这样的效果是今天的游戏中常见的后期处理特效。一般来说，后期处理效果本身是 2D 的。例如，像素偏移效果会变换像素以扭曲图像，模拟由热气造成的折射或其他特殊效果。这个过程一般应用于图像上的一个特定子区域，而且不考虑物体顺序和形状的问题。例如，图 5.8.1 中演示的热气摇曳（heat shimmer）的效果。火焰产生的热气会扭曲它后方的物体，并偏移像素来产生波纹的模式。如果做得不好，热气摇曳效果不仅会扭曲火焰后方的物体，而且会扭曲它前方的物体，那是不正确的。显然，后期处理效果需要一些形式的深度知识。本文将讨论如何用体积化（volumetric）的后期处理把深度信息整合到图像空间中去。

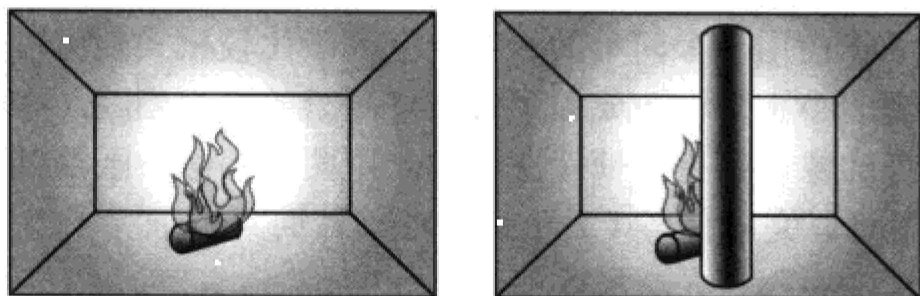


图 5.8.1 正确的后期处理。来自火焰的热气应该扭曲它后方的墙。但是，如果一个柱子那样的东西放在了物体前方，这个柱子就不该受到热气摇曳的影响

5.8.1 体积化的后期处理

为了应用上述的体积化后期处理，需要先定义后期处理体的概念。后期处理体（*post-process volume*）是一个 3D 物体，会影响它后方的所有场景像素。图 5.8.2 演示了一个后期处理体的例子。

和标准的后期处理不同，我们使用 3D 形状，而不是一个 2D 的屏幕空间矩形作为后期处理区域。后期处理体定义了一个区域——它不是直接渲染的。后期处理体可以是一个简单的立方体，如图 5.8.2 所示，也可以定义成一个更为任意的形状，如[Oat04]所讨论的。自然地，一些场景物体可能会穿透后期处理体，使得只有一部分物体会被扭曲。

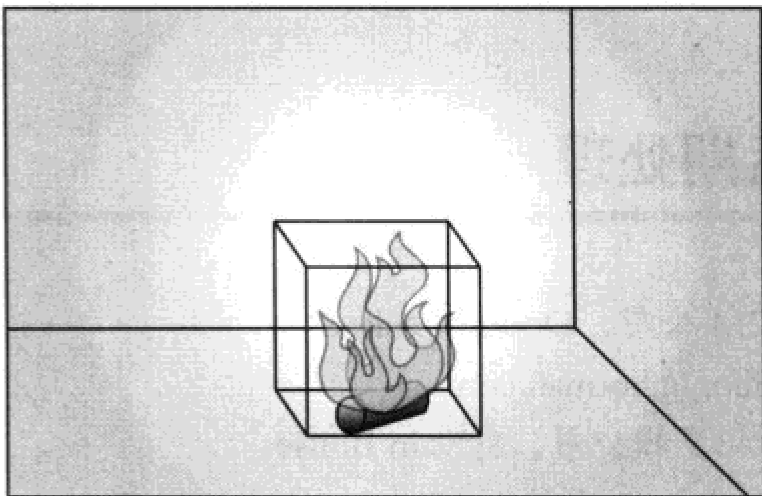


图 5.8.2 后期处理体

5.8.2 深度知识

在定义了后期处理体之后，我们就可以把场景分成两个集合：在体前面之后的区域和在它前方的区域。我们可以简单地排序场景中的对象，找到和后期处理体之间的空间关系。但是，这对于与体相交的物体没有用，而且对于任意场景普遍不够棒。

实际上，我们需要做如下工作。

- 只把后期处理体后方或包含在后期处理体内的物体渲染到离屏渲染目标上。
- 对渲染目标应用后期处理效果（也就是用扭曲来模拟由热气产生的折射）。
- 渲染在后期处理体前方的物体。

第一个问题是如何只渲染在图像空间中位于后期处理体后方的物体。我们需要把场景像素的 z 值和后期处理体像素的 z 值做比较，只允许通过 z 值大于后期处理体 z 值的那些像素。

这就暗示了使用一个简单的 z 缓冲区大于比较模式就可以奏效。不幸的是，把 z 缓冲区测试从它常规的小于等于改成大于会破坏场景的隐面删除。本质上，我们要让在后期处理体前面之后最近的像素存在于帧缓冲区中。用两个 z 缓冲区测试，每个比较值来自于不同的源帧缓冲区，就可以完成任务（大于后期处理体而且小于等于当前帧缓冲区的 z 值），但在当前硬件的深度缓冲中不存在这样的概念。

5.8.3 使用 shader 作 z 比较

因为图形卡本身不支持多个深度缓冲区测试，所以解决方案是通过使用 vertex 和 pixel shader 把这个功能添加到视频卡中。我们将用这种方式实现与后期处理体的比较。

自然地，我们先需要一个用于比较的值。后期处理体的 z 值必须计算并存储在图像缓冲区中，其大小和主帧缓冲区一样。我们不能直接使用视频卡 z 缓冲区的 z 值，因为这些值一般以特定硬件的格式编码或压缩。

取而代之的是，我们可以使用 vertex shader 来自己计算这些 z 值。这些值可以通过插值器传给 pixel shader，并存在一张纹理中。因为值比较中需要高精度，所以有必要使用浮点纹理。每个像素只需要存储单个浮点值，所以我们使用 Direct3D 中 `D3DFMT_R32F` 格式的纹理。因为不是所有的硬件都支持浮点纹理，所以标准的颜色纹理仍然可以用做后备。这会造成精

度限制，在屏幕上出现细微的走样，虽然这可以通过一些变通办法来减少，比如在计算深度值的时候减少远/近比。

把后期处理体的 z 值存放到一个单成分的浮点纹理使我们得到了需要的值，所以我们可以找到在后期处理体后方的物体。后面将把这张纹理作为后期处理体的 z 深度，成为体的深度纹理（*Volume's depth texture*）。我们只想让后期处理体之后和之内的物体受到后期处理的影响。

5.8.4　像素完美的裁剪

后期处理体的深度纹理现在可以和场景中的像素比较了。深度纹理首先是作为一个激活的纹理被选中的。和以前一样，一个 vertex shader 会计算场景中的 z 值并把它们传给 pixel shader。

我们还需要知道像素缓冲区中的像素对应于深度纹理中的哪个像素。为此，vertex shader 将执行一个到屏幕空间的透视变换。然后屏幕空间坐标将由 vertex shader 归一化到 0..1 的区间，使得屏幕坐标对应于深度纹理上的 UV 坐标。这些 UV 坐标会用插值器传给 pixel shader。然后 pixel shader 可以用这些 UV 坐标在深度纹理中查找后期处理体的 z 值，如图 5.8.3 所示。

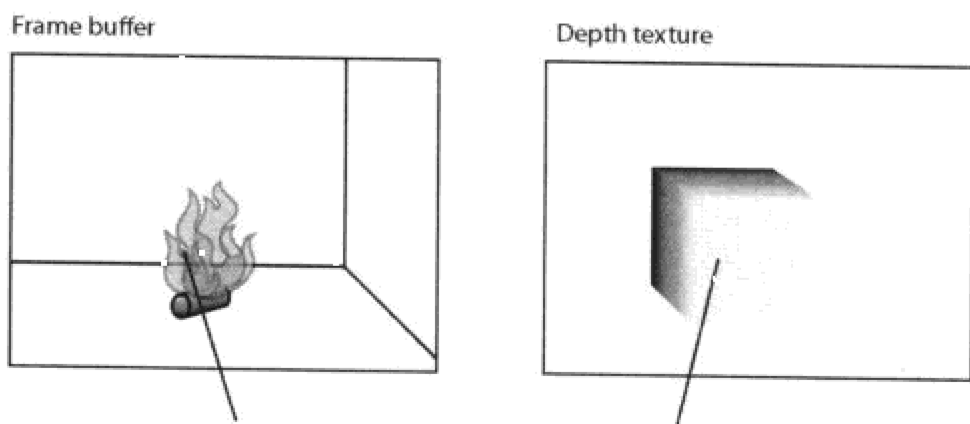


图 5.8.3　查找深度值

这两个 z 值在 pixel shader 中作比较，而 alpha 测试用来标记哪个像素要写到帧缓冲区中。如果屏幕像素的 z 值大于后期处理体的，结果像素的 alpha 就设为 1；否则，设为 0。把 alpha 测试比较函数设为大于，并把 alpha 参考值设为 0，就可以把在后期处理体前方的像素丢弃。我们也可以使用汇编的 `texkill`（或 HLSL `clip()` 函数）来执行这个有条件的像素丢弃。

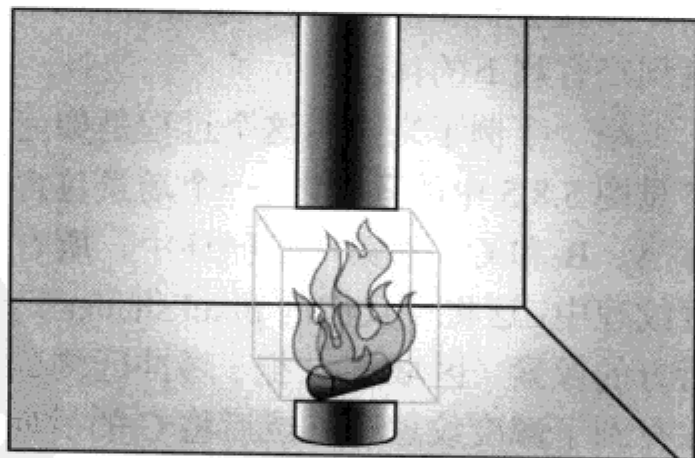


图 5.8.4　“裁剪”操作

渲染后，我们将得到一个在像素级被后期处理体网格“剪掉”的渲染场景。这个裁剪操作所用的后期处理体网格可以是任意复杂的，如图 5.8.4 所示。

5.8.5 后期处理

正如前面所提到的，有很多种后期处理变换可以应用到图像空间中，比如扭曲、模糊和颜色变换。在我们的例子中，是为一个热气摇曳效果建模，它可以通过执行一个正弦波扭曲加轻微的模糊来适当地完成。详细的细节请参考配套光盘上的样例代码。这个效果是通过应用适当的后期处理 shader（模糊、偏移、辉光等等）并重新渲染后期处理体来实现的。这会使得后期处理只应用在体中的像素上。

5.8.6 最后一遍

在这个阶段，我们将渲染后期处理体后方的东西，并把效果应用到后期处理区域中。我们现在必须渲染在后期处理体前方的没有扭曲的多边形。这完全是算法的第一遍的相反操作。然而，当场景多边形的 z 值小于后期处理体的时候，像素的 alpha 为 1，否则为 0。在这个时候，图像就是完整的了。

5.8.7 多个体

我们的技术可以用于多个后期处理体，只要那些体不相交，而且可以通过它们的 z 值很容易地进行排序。有了这些条件，我们的算法可以扩展成：

1. 清除帧缓冲区。

2. 把深度缓冲区清为 0。

3. 选择大于的 z 缓冲区比较测试，并把所有的后期处理体渲染到深度纹理中，在深度纹理中将 z 值存储成浮点数。

4. 把 z 缓冲区测试设为小于，而不是小于等于，以节省填充率（我们将多次渲染相同的多边形），场景用一个计算 z 值的 vertex shader 渲染，pixel shader 会过滤掉在深度纹理中 z 值前方的像素。

5. 把最远的后期处理体的后期处理效果应用到屏幕上。

6. 回到第 2 步，这次去掉离视点最远的后期处理体。循环步骤 2 到 5 多次，每一遍都去掉第二远的后期处理体，直到没有剩下的体。

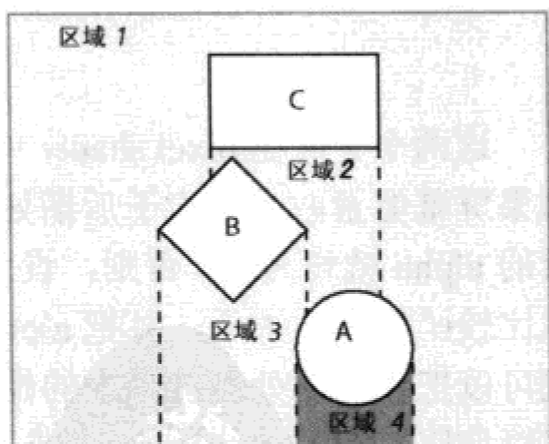


图 5.8.5 多后期处理体

举一个例子来说明这个过程是如何作用于多个体的，参见图 5.8.5 中的顶视图。一个场景包含不相交的后期处理体 A、B 和 C。在第一次循环中，所有三个体都渲染到深度纹理中，这将使我们的 pixel shader 只渲染在所有三个体后方的像素（区域 1）。把 z 缓冲区测试设置成大于，以保证来自这三个体的最远正面多边形保存到了深度纹理中。然后将 C 的后期处理效果应用到场景中。

在第二次循环中，清除深度纹理，后期处理体 A 和 B 会渲染到深度纹理中。我们重新渲染场景，允许将在 A 和 B 后方但在 C 前方的像素（区域 2）写到场景中。然后对 B 应用后期

处理（当前的后期处理体集合中最远的一个）。在第三次循环中，只把后期处理体 A 渲染到深度纹理，因此加上了在 A 后方但在 B 和 C 前方的像素（区域 3）。然后对 A 应用后期处理。最后，渲染在所有三个后期处理体前方的物体（区域 4）。这部分不会应用后期处理效果。

5.8.8　总结

本文呈现的技术允许任意的 3D 体受到后期处理的影响。这使得我们可以通过多种局部图像滤波器（filters）来实现抢眼的效果。

该算法依赖于一些最新的视频卡硬件所提供的特性，比如 shader 功能以及（最重要的）浮点纹理。当场景几何体渲染多遍的时候，会有一些开销，但是这可以通过聪明的裁减和好的可见性系统来减轻。

该技术也可以用于其他很多必须用体来达到像素完美的 3D 裁剪的地方。我们很容易就会想到用这项技术来做出“切掉一部分”的物体，或执行基于体素(voxel)的处理。

5.8.9　参考文献

[Isidoro02] Isidoro, John, Guennadi Riguer, and Chris Brennan. “Texture Perturbation Effects.” In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, 337–346. Available online at *http://www.ati.com/developer/shaderx/ShaderX_TexturePerturbationEffects.pdf*.

[Michell02] Mitchell, Jason L. “Image Processing with 1.4 Pixel Shaders in Direct3D.” In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, 258–269. Available online at *http://www.ati .com/developer/shaderx/ShaderX_ImageProcessing.pdf*.

[Oat04] Oat, Christopher and Natalya Tatarchuk. “Heat and Haze Post-Processing Effects.” In *Game Programming Gems 4*. Charles River Media, 2004.

5.9 过程式关卡生成

北得克萨斯大学，Timothy Roden 和 Ian Parberry
roden@cs.unt.edu
ian@cs.unt.edu

传统上，一个典型的 3D 游戏开发项目同时有两个过程。程序员负责设计、编码和测试游戏引擎，与此同时，美工会建立游戏的内容。对一些项目来说，这个范式不再必要，甚至不可行。硬件上的技术进步使得艺术产品的使用比以前更为详细。越来越多的存储器和可用的 RAM 进入了大型游戏世界中。一些游戏需要巨大数量的内容，比如在线多人游戏和以提供高级别重放效果为目标的游戏。另一个因素是越来越多可用的引擎和其他高质量的中间件，可以极大地减少引擎开发的时间。这意味着美工要花费更长时间去建立高清晰的艺术内容，而这些内容在开发周期中越早完成越好。这个问题的一个显而易见的解决方案是过程式（procedural）地建立艺术内容。本文呈现了过程式关卡生成器背后的想法和技术，并演示了如何用它来建立简单的 3D 地牢。

5.9.1 大致的方法

对于关卡生成器，我们有多个目标。首先，我们需要一组结合在一起并且使用入口（portal）技术进行渲染的 3D 室内几何体——也就是一个关卡。这个系统应该尽可能地通用，以便适用于特殊的需求。其次，我们要生成任意大小和复杂性的关卡，并使这个系统足够快，以使它可以在程序执行的过程中动态地生成关卡。最后，我们要确保这个关卡生成器尽可能少地依赖于人工建立的艺术制品。理想状况下，整个关卡都将由计算机合成。然而，实际上仍需要少量人工建立的艺术制品来达到质量更好的关卡。

建立一个关卡需要以下 5 个步骤。

1. 设计关卡。
2. 建立一组预制的 3D 几何体。
3. 过程式地生成 3D 图（graph）。
4. 过程式地把预制件映射到图的节点上。
5. 过程式地把内容（细节）添加到关卡上。

5.9.2 关卡设计

设计一个过程式生成的关卡非常类似于手工设计一个关卡，至少最初

的部分是相同的。要回答的问题是，这个关卡的主题是什么？关卡的大致大小和形状是什么？有多少房间？有多少走廊？因为关卡将过程式地生成，所以我们需要参数化地指定一些信息。为了本文的目标，我们要建立一个简单的由走廊、阶梯和房间组成的地牢。

我们将地牢放在一个 3D 栅格中，把世界空间划分成叫做“单元”的立方体区域，如图 5.9.1（a）所示。使用栅格会简化很多实现细节，包括几何体的拼接、把世界坐标系映射到局部坐标系、自动生成入口和几何体实例。每个栅格单元将是 90 英尺长，90 英尺宽和 50 英尺高，如图 5.9.1（b）所示。每个单元可以包含一个走廊或一个房间。每个单元中的几何体最多可以和 4 个相邻单元（东、南、西和北）的几何体连接。我们允许相同高度或者相同倾斜度的连接。例如，我们允许一个走廊连接到相邻的更高或更低的走廊，如图 5.9.4 所示。在这种情况下，我们会生成阶梯来连接两个单元。

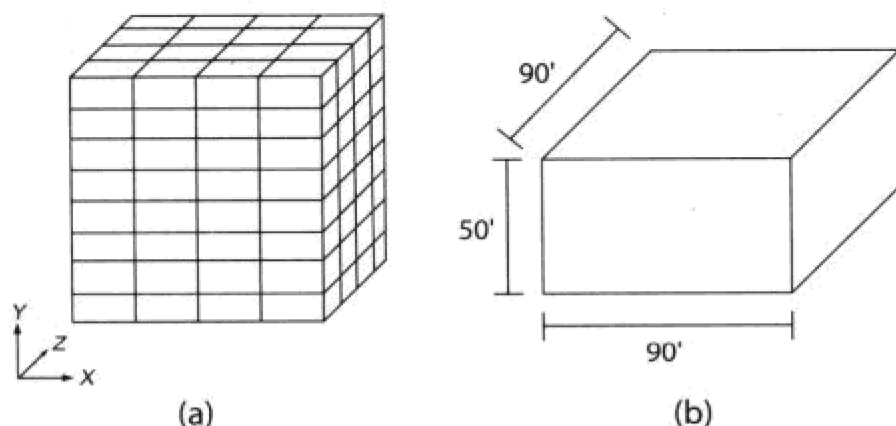


图 5.9.1　（a）世界空间用 3D 栅格分割成立方体单元。（b）一个 3D 栅格“单元”

走廊，不管是平的还是带阶梯的，都是 10 英尺宽。平的走廊有 10 英尺的天花板，而带楼梯的有 15 英尺。房间是 30 乘 30 英尺的方形，有 10 英尺的天花板，而且有 10 英尺的开放入口（没有门）。走廊之间的所有水平连接都有一定的角度。有了这些尺寸，我们就可以在单元内排布一个 4 方连接的走廊，如图 5.9.2（a）所示，或者一个 4 方连接的房间，如图 5.9.2（b）所示。

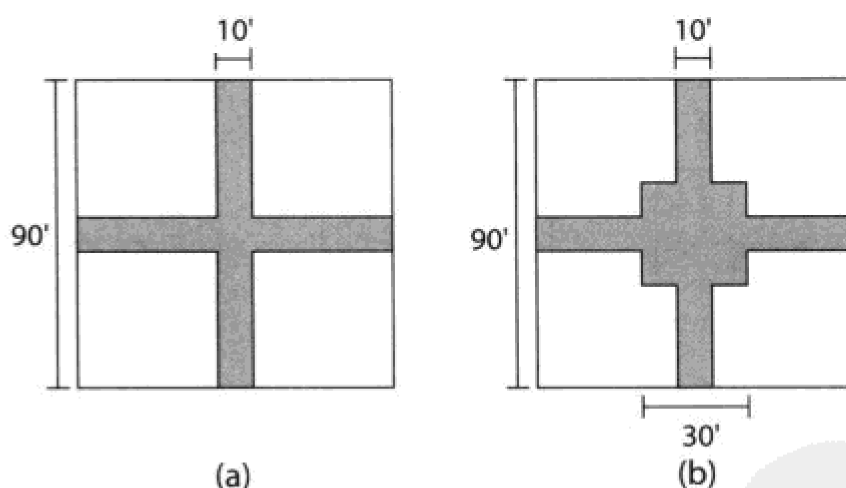


图 5.9.2　（a）包含一个 4 方连接走廊的顶视图。（b）包含一个 4 方连接的 30'×30'房间的顶视图

5.9.3　使用预制的几何体

和那些用预先定义的关卡来生成每个单元几何体的纯过程式方法不同，我们要加入人工元素。我们希望关卡看起来尽可能好，所以最好是加入美工建立的真实关卡几何体的入口。

我们用一个建模程序来建立一组预制的几何体块。使用预制件作为积木来构建更复杂的几何体是一个直截了当的方式，很多传统的以人为中心的关卡设计中已经使用了这种方法[Perry02]。

我们的目标是建立一个可用于每种可能的单元布局的预制件。它的问题是即使对于我们目前讨论的简单的地牢来说，也有大量可能的单元变化。考虑一个带走廊的单元。这个单元可以是 1 连接，2 连接，3 连接或 4 连接的。对于 1、2 或 3 连接的单元，有 4 个不同的方向。例如，一个 1 连接的单元可以连接到东、南、西和北的任何一个单元。为了不让事情更糟，连接可以发生在相同或不同的两个单元中。例如，一个 2 连接的单元，有 9 种不同的变体，当乘上四种可能的朝向时，就给了 36 种可能的布局。一个 4 连接的单元有 80 个可能的布局，等等。

为了让建立预制件的工作更为可控，并尽可能多地重用集合体，我们把每个单元水平地分成 3×3 的子单元，尺寸是每个 30×30×50 英尺。我们的目标是在运行期，把几个小的预制件拼接在一起，来组成单元的几何体。如你所见，我们只需要演示在图 5.9.3 中的九个小预制件就能完成这项工作。为了扩大这个子单元预制件的集合，我们可以选择建立一个或多个全单元（或子单元）“特殊”的预制件。如果我们的设计要求在地牢中包含那些不能简单地只使用九个小预制件构建出来的特殊房间或区域，这样做就会很有用。

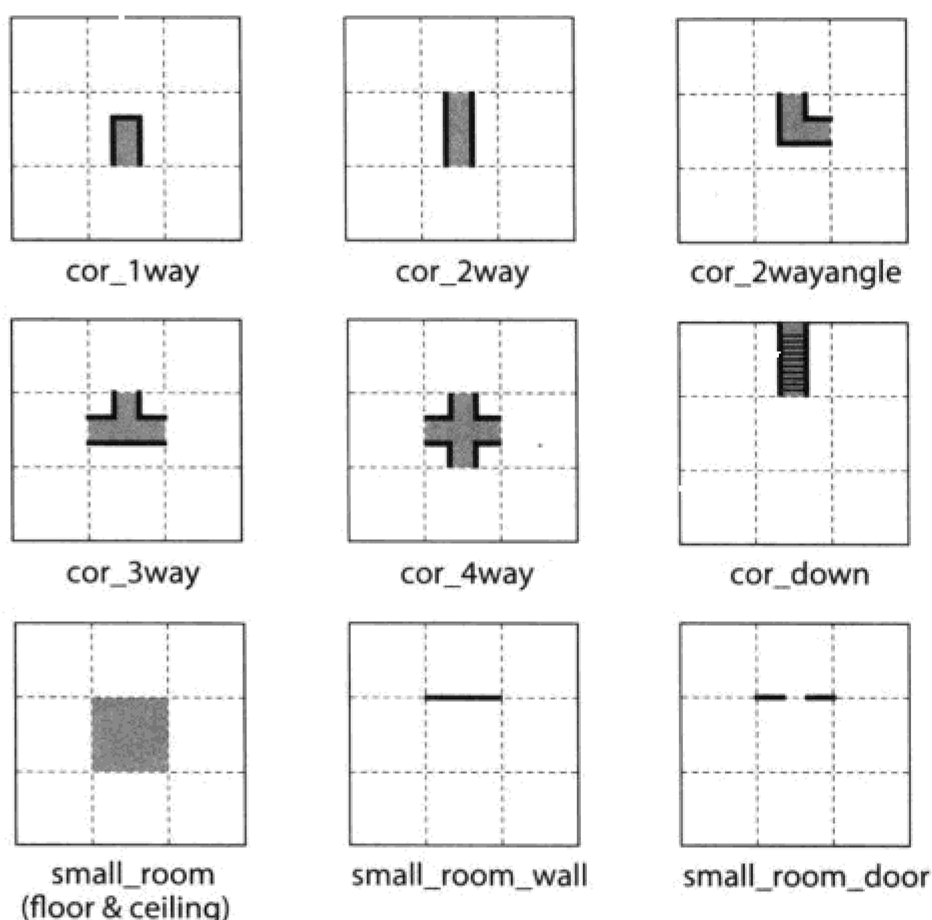


图 5.9.3 基本的地牢关卡所需的九个预制件的顶视图，每个都表示在 90'×90'的单元中，并细分成 3×3 的子单元，每个单元 30'×30'

5.9.4 图的生成

有了设计方案和一套可用的预制件，就可以准备实现关卡生成器的过程部分。我们要生成一个 3D 图数据结构作为过程式游戏关卡的高级表示。图中的每个节点将对应于世界空间

中的一个几何体单元。我们会把图的节点存储在数组中。图中的每个节点包含下面的数据。

```
struct GraphNode {
  int x, y, z;  // 在场景中的坐标，(0,0,0)为起始点
  int dir[4];   // 4 个相连的节点的索引，(0 表示没有,
                //   其他表示连接到 dir[x]-1 的节点)
};
```

生成一个随机图有无数种可能性。选择哪种方法应该受到设计方案的驱动。我们的设计可能在图生成器中包含约束。例如，我们可能让地牢中一些指定的房间只能以特定的顺序被访问。

为了本文的目标，我们要生成一个带有类似触角的拓扑，从图中的起始节点（地牢的入口）向外辐射。对需要的几何体有了一些知识，我们就可以在图生成器上放入一个约束——如果一个现有的节点已经连接到了下层节点，那么图中的一个新节点就不能直接建立在它的下面。其理由可以在图 5.9.4 中看到。在一个单元需要连接到下层单元的情况下，这个单元真实的几何体将落到它下面的单元中。因此我们需要它下面的单元仍保持为空，因为我们会假设这个几何体不但占用了本单元包含的空间，也占用了紧接在它下面的那个单元。

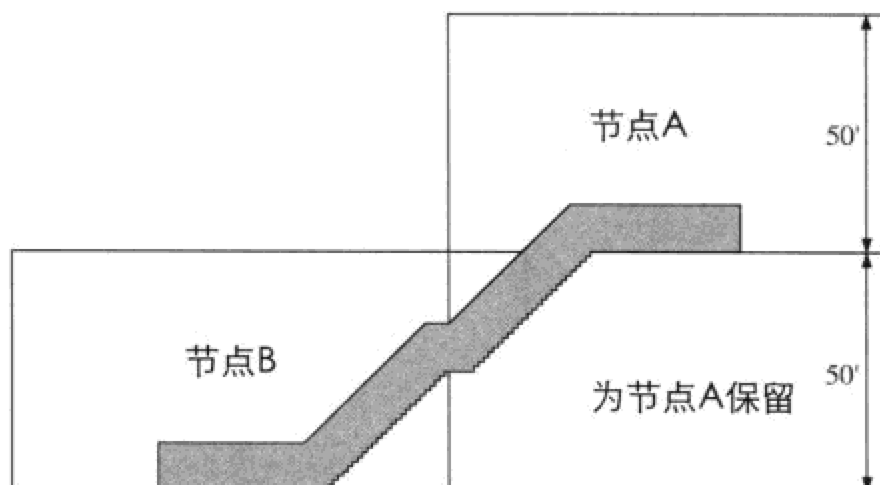


图 5.9.4 侧视图演示了两个走廊节点（单元）是如何在对角线用阶梯互相连接的。节点 A 下方的图节点是保留给节点 A 使用的

首先要指定图节点的最小和最大数量。通过一个参数，指定不同高度之间节点的连接百分比来控制图的高度。例如，使用 25%的值表示每个添加到图中的新节点将在 75%的时候连接到相同高度的节点，在 25%的时候连接到不同高度。

基本的算法工作方式如下。首先，生成一个起始节点，表示地牢的入口。因为这个节点最多有 4 个到其他节点的连接，所以我们把 4 个入口添加到“可用”列表中。然后，循环过每一个要生成的其他节点，在循环中完成如下工作。

1. 建立新节点。
2. 从可用列表中随机选取一个现存的节点作为挂接点。
3. 确认所有的约束都满足了。
4. 把新节点挂接到图中。
5. 从可用列表中删除一个入口。
6. 把新节点的三个入口添加到可用列表中。

如果发现挂接新节点会破坏任何约束，我们就选择其他的挂接点。对于非常复杂的约束，就

会有很多挂接点不是有效的。在这种情况下，如果已经生成了最小数量的节点，我们可以中止算法；或者从头重启算法。当实现一个新的图生成算法，或者修改一个已有的算法时，总是应该做自动测试，以确保算法可以很好地工作。

当从可用列表选择一个新节点的时候，我们要在挂接点优先选择最近建立的节点。这会使图向外生长，产生更有趣的图。选择可用列表的一个简单方法是生成一个在 1 和可用列表中条目数量之间的随机数。我们把这个值称为 r。接下来，生成第二个随机数，它在 r 和可用列表中条目数量之间，使用第二个数作为到可用列表的索引。

创建好图之后，我们把图存成二进制和文本文件格式。除了图的数据之外，这个结构还包含一些统计值，包括随机数发生器的种子值。这个种子值会非常有用。例如，给定一个完整生成的关卡，我们可以只使用这个种子值来再次生成它。一个文本文件的例子如下。

```
num_nodes: 99
max_level: 5
adjacency: 4
min_nodes: 50
max_nodes: 100
percent_vertical_connects: 0
percent_sloping_horizontal_connects: 25
random_type: 0
random_start_seed: 1076104227
random_end_seed: 1359372770

Node: 0
  location (x,y,z): 0,0,0
  connected to nodes: 1(N)
Node: 1
  location (x,y,z): 0,-1,1
  connected to nodes: 0(S) 2(E) 3(W)
...
```

配套光盘中包含了图生成器的源代码。

5.9.5 把预制件映射到图中

生成关卡的下一步是遍历图中的每个节点，把几何体映射上去。我们根据对应单元中需要的几何体类型来标记每个节点。有些节点变成房间，而有些节点标记变成走廊。在这里可以强加一些额外的约束，比如没有两个房间是相邻的。

把预制件映射到图中完全是机械式的。我们要初始化一个满单元模型的空列表。建立这个单元模型列表的时候，所有可能的单元几何体变化将被存储在关卡中。而出现相同类型（房间，走廊）、相同空间连接性的单元很可能有多个。在这种情况下，我们不想有重复的模型几何体。取而代之的是，我们允许图中的多个节点引用单元模型列表中的相同几何体，如图 5.9.5 所示。

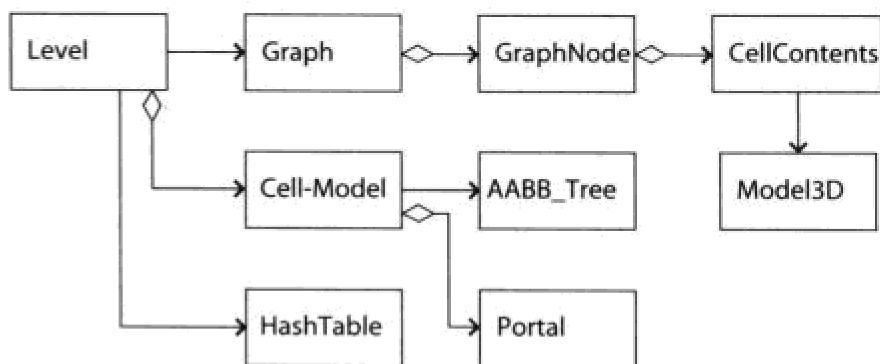


图 5.9.5 组成一个关卡的主要数据结构

使用子单元的预制件，我们可以通过提前生成所有可能的各种单元几何体的工作基本上从美工转移给了电脑。因为只需要存储关卡中用到的单元几何体，所以我们还可以减少关卡的运行期存储需求。

对于图中的每个节点，我们要检查它的连接性信息，并查看在单元模型列表中前面生成的几何体。如果找到了相同的几何体，那我们就引用这个几何体。否则，就为这个单元建立几何体，并把它放入单元模型列表。建立新单元几何体的过程如下：

1. 集合需要的子单元预制件。
2. 按照需要平移/旋转每个预制件。
3. 把所有的预制件拼接成单个模型（参见图 5.9.6）。
4. 把这个新模型添加到单元模型列表中。

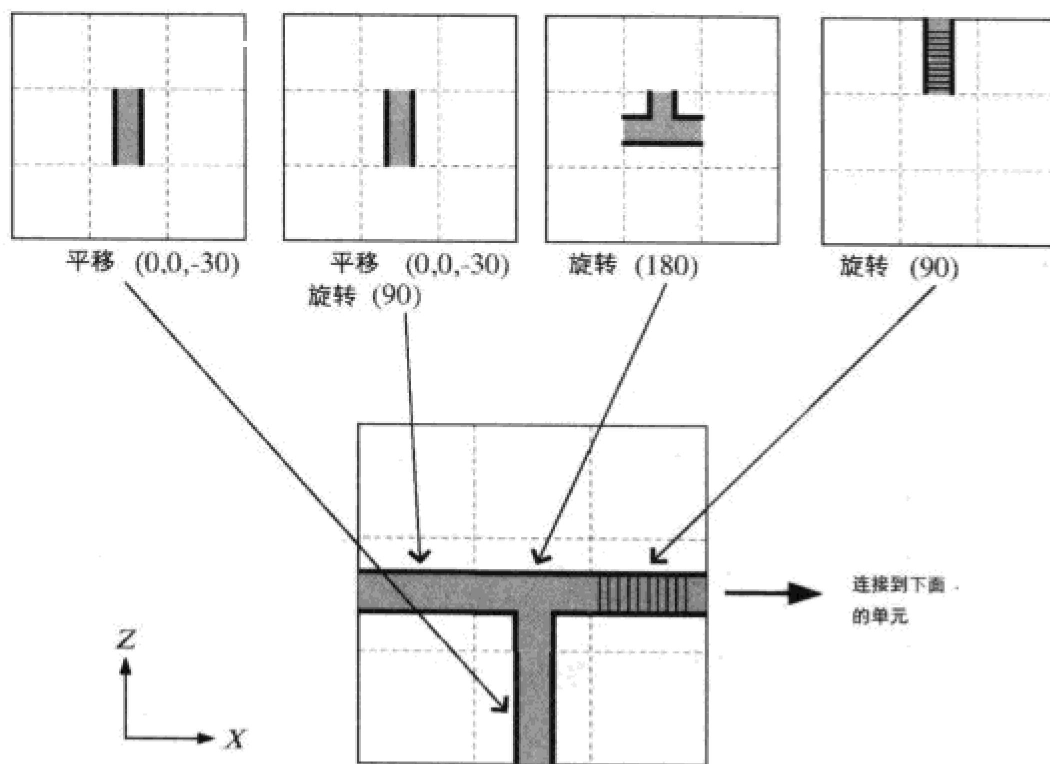


图 5.9.6 把 4 个子单元预制件拼接成单个模型

因为我们的子单元预制件在顶点数目方面很小，所以不推荐把它们作为独立的模型用分开的 API 绘制调用来渲染。取而代之的是，为了效率和简单性，我们把关卡中每个单元的几何体当做单个模型。为了把多个小模型组合成一个，我们必须改变方向并把每个单元的所有子单元预制件拼接成单个模型，如图 5.9.6 所示。这明显地减少了 API 开销，可以高效地进行渲染。

5.9.6 可见性和碰撞检测

一旦建立完所有的单元几何体，我们就可以动态地产生渲染和碰撞检测所需要的数据。对于单元模型列表中的每一个模型，我们都要建立相关的最多四个入口的集合，这取决于此单元的连接性。每个入口是一个 2D 矩形，位于连接一个单元和它的相邻单元的平面上，它会围住单元几何体的所有顶点。我们用这些入口来进行关卡的入口渲染[Luebke95]。标准的入口渲染方法使用的是通过入口连接的凸包单元，而我们与它不同，在每个单元中的几何体方面并没有限制。使用非凸单元的缺点之一是它可能在渲染的过程中造成过度绘制。但是，因为我们对单元几何体没有限制，而且可以很容易地自动建立入口，所以这一问题就不复存在。

为了使用碰撞检测，我们为单元模型列表中的每个模型建立一个轴对齐包围盒（AABB）的二叉树，其在方式上类似于[Schroeder01]。最后，因为在关卡中的一个单元（在世界坐标系中）和它对应的图节点没有直接的关系，所以我们可以建立一个散列表（hash table），把世界坐标映射到图的节点中。这个散列表的高效性非常重要，因为每次引用关卡中的数据，使用世界坐标，都会通过这个散列表来映射。对关卡可能的布局有大体的了解有助于让散列函数变得更快。

5.9.7 增加关卡内容

至此，我们已经有了一个基本的关卡，也许很大但是很空，如图 5.9.7 所示。使用一个基于规则的系统，就可以加入几种类型的内容，其中的规则来自于设计。图中的节点可以关联环境音频属性，以及其他基于每个节点几何体的大小、形状和内容的音效。真实几何体的添加分为两类。静态几何体包括把柱子或雕像之类的物体添加到关卡几何体中。这些是我们可能想要拼接到当前的单元几何体中的东西。如果是这样的话，就需要保证这个单元几何体没有被图中的多节点引用。如果有，那么在把任何新的几何体拼接到单元之前，我们就需要为这个单元几何体建立一个独立的版本。拼接新的几何特性也需要更新单元的 AABB 树，并可能要重新计算入口。非静态的几何体只要简单地加入到节点关卡中就可以了，不需要拼接。

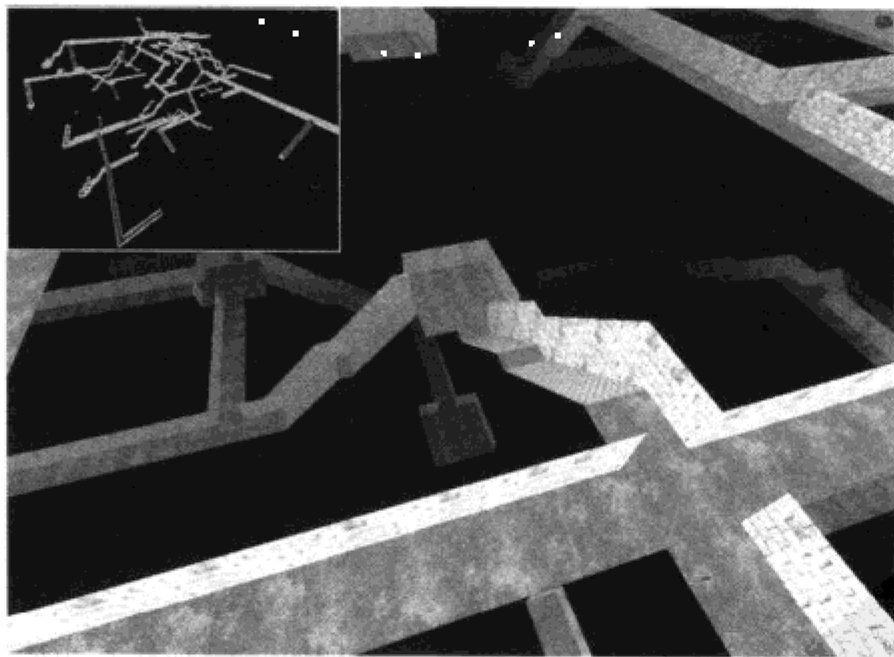


图 5.9.7 基本的地单走廊和房间，通过一个“触角”图生成（插入式）。地单的入口是显示在上图中的一个小走廊

5.9.8　总结

本文呈现了一个产生过程式游戏关卡的方法，用以解决越来越多的游戏类型所需要的大量内容，比如大型多人在线游戏。如果给定一个适当的规则集，我们的算法就可以把简单的数据结构扩大成复杂的几何游戏关卡[Roden04]。我们的方法是一开始使用适当的规则来产生一个 3D 图，然后将预制的 3D 几何关卡碎片映射到图的节点上。我们引用惟一的几何体，而不是拷贝，小心地把子单元预制件拼接成单个模型，以减少多余顶点的数量，并节省渲染时间。然后，为后面的入口算法计算可见性和碰撞检测信息。最后，添加静态几何体和环境音效之类的关卡内容。

这里呈现的方法应该被看成是读者建立自己的关卡生成器的基础。目前没有涉及的一个重要问题是光照。一个用于添加真实光照的过程不单取决于关卡的设计，而且取决于关卡生成器是如何使用的。如果关卡是作为预处理来生成的，那光照就可以用预处理的技术计算，如果是动态生成的关卡，那就可以使用多种不同的光照技术。

5.9.9　参考文献

[Luebke95] Luebke, David and Chris Georges. "Portals and Mirrors: Simple, Fast Evaluation of Potentially Visible Sets." In (Pat Hanrahan and Jim Winget, editors) *ACM Symposium on Interactive 3D Graphics* (April 1995): pp. 105–106.

[Perry02] Perry, Lee. "Modular Level and Component Design." In *Game Developer Magazine* (November 2002): pp. 30–35.

[Roden04] Roden, Timothy, and Ian Parberry. "From Artistry to Automation: A Structured Methodology for Procedural Content Creation." *3rd International Conference on Entertainment Computing* (September 2004).

[Schroeder01] Schroeder, Tim. "Collision Detection Using Ray Casting." In *Game Developer Magazine* (August 2001): pp. 50–56.

5.10 重组 shader

A2M 公司，Dominic Filion
dfilion@hotmail.com

引入了 shader 之后，图形处理器（GPU）近来变得非常接近于 CPU。GPU 从严格的图形处理设备变成了更通用的芯片，正如在越来越多的研究中可以看到的，shader 正用于物理、数学计算和光线跟踪。

使 shader 更为有用的 GPU 编译器的出现只是一个时间问题。GPU 确实和它们的 CPU 兄弟有着相似的进化过程（专用芯片、进化到通用目的、编译工具和优化器），但是步调快得多。

然而，当涉及灵活性和整合的容易程度时，GPU 却远远落在 CPU 之后。GPU 存在如下问题。

有限的程序长度：微程序的有限长度意味着整个程序的图形流水线不能放到单个程序或函数中，不像软件渲染器那样。

有限或不支持分支：缺乏在大部分编程模型中都有的分支指令，这限制了流水线本身的灵活性。

没有通用的不支持 Seatter-Guther（分散-收集）：GPU 没有一个通用的内存模型，因为它们的内存不能随机访问。

可能和固定流水线的交互：在 API 级别，shader 运行的路径和标准固定流水线完全不同。shader 中实现的效果必须自己实现蒙皮、光照和其他固定流水线已经提供的特性。没有办法高效地“调用”固定流水线的回路。

5.10.1 组合效果

实际上，图形管道的非常自然直接和范围有限的特性，通常还伴随着前述中的 shader 编程模型的限制。这直接导致应用程序把 shader 当作硬编码和预先定制过程来对待。而这是当初 shader 设计出来所要阻止的事情。

考虑类似计算蒙皮的矩阵调色板加一个普通的特效的 technique。特效的 shader 通常只影响光照、纹理坐标及除位置外的其他顶点分量。因此为J和蒙皮进行组合，vertex shader 必须针对有蒙皮和无蒙皮编写不同的版本。这种组合爆炸意味着即便是在一个普通的图形引擎里，也不能仅仅编写一个 shader 就应付所有的情况。实际上，即便 DirectX® 9 指定的固定函数流水线都不可能用单个 shader 来完整地支持[Sander03]。因此，为了最大限度地优化，每种可能的固定流水线设置组合都必须写一个独立的 vertex shader。

本文将探索处理这个多 shader 变体问题的方法，以及生成这些变体、存储它们和把一个灵活的 shader 流水线整合到 3D 引擎的高效技术。本文将向读者展示一个用于灵活 shader 整合的特殊方法，但是本文也可以为日后研究更新颖的 shader 整合方法提供起点。

5.10.2 处理组合爆炸

首先，让我们列出一些方法来处理 shader 变体的增长。

封闭法：这是解决该问题的最常见方法。大部分游戏会写 5 到 60+个高度特化的 shader，用来处理游戏中要实现的最常见的效果。图形流水线的特性集被清晰地定义之后，这个方法就是可行的，但是这限制了创造性，并使得引擎代码在游戏设计中不能灵活改变。在这个策略下，通过不允许在游戏中应用任意数量的效果类型，避开了 shader 变体的问题。

开放法。在运行期生成 shader 变体：最灵活的方式是在运行期建立 shader 变体。引擎渲染器必须分析当前的渲染器设置，确定需要的 vertex shader 代码片断，并把它们组合成单个 shader。有很多方法可以实现这一点。

- Nvidia 的 NVlink
- DirectX 片断连接器
- HLSL

生成 shader 变体的方法可以分为加法或减法。加法是用添加、拷贝或把片断连接成代码来完成的，而减法是采用一个大的通用 shader，并把它提炼成一个只有一部分功能的特殊变体。这里，我们简要地概括了这些方法，以便理解和指出将提出的新方法的优点。

1. NVIDIA 的 NvLink

NVIDIA 提供的 NvLink 工具是一个老的增加方法，用来把汇编语言 vertex shader 片断缝合在一起。`#beginfragment` 和`#endfragment` 关键字被放在代码中，用来划分出片断。INVLink 接口（从 NVIDIA 提供的库中获得）可以在运行期把多个片断连接在一起。我们不会更多地讨论这个技术，因为它只能用于汇编 shader，严重地限制了它的使用范围，它只有历史价值。

2. D3DX 片断连接器

D3DX 片断连接器由 D3DX 库中一组文档很少的函数组成。关键的入口是 `D3DXAssembleFragments()`，`LinkShader()`和 `LinkVertexShader()`。这些函数可以用来增加连接用 DirectX 汇编语言编写或从 HLSL 编译而来的 shader 片断。

为了使用这个连接器，要给前缀是 `r_`的语义（semantic）指定一些函数参数。随后这些参数就像 shader 片断之间的胶水一样，成为了它们之间通信的渠道。顶点片断也必须声明一个特殊的 `vertexfragment` 关键字，如下面的例子所示。

```
void Transform(
    float4 vPos : POSITION,
    float4 vNormal : NORMAL,
    float3 vPositionResult : r_TransformedPosition,
```

```
    float4 vNormalResult : r_TransformedNormal )
{
    vPositionResult = mul( vPos, mWorldView );
    vNormalResult = mul( vNormal, (float3x3)mWorldView );
}

Vertexfragment Transform = compile_fragment vs_1_1 Transform();
```

D3DXAssembleFragments()可以从一个文件中载入所有的 shader 片断，LinkShader()可以用来组合它们，构建出 shader [Boyd]。

使用片断连接器是一个轻量的过程，可以在运行期完成。这个连接器可以解决符号表，优化寄存器使用，并删除无用代码。

5.10.3 通过 HLSL 生成 shader 变体

因为 shader 可以用高级语言编写，所以自然能用 shader 编译器把 shader 代码字符串连接在一起，以建立更为复杂的 shader。通过高级 shader 语言生成 shader 变体可以使用增加或减少法来完成。

5.10.3.1 增加法

通过增加法，shader 代码可以从小的 shader 片断构建出来。它的优点是写小的 shader 片断时不需要知道整体 shader 流水线。以这种最简单的形式，我们只需要把 HLSL 字符串连在一起就可以了。然而，为了更高效，需要定义一些严格的规则，以便 shader 片断相互组合。

我们可以把 shader 流水线看做一组组合在一起的原子性组件块，而不像硬件组件块。这些块有用于通信的输入和输出端口。把片断添加到 HLSL 代码很简单，因为它们可以写成独立的函数。然后在运行期间使用 shader 组合器粘合代码，并在需要的地方调用函数。

5.10.3.2 减少法

减少法是使用一个完全不同的方法来生成 shader 变体。首先要编写一个大的通用 shader，描述游戏的整个流水线，包括蒙皮、特效、uv 动画等等。这个通用 shader 的特殊版本通过使用常量和/或 define 来生成，如下面的例子所示。

```
struct Input
{
    float4    Position     : POSITION;
    float4    Normal       : NORMAL;
};

struct Output
{
    float4    Position     : POSITION;
    float4    Color0     : COLOR0;
};

// 参数
```

```
float4    Diffuse[8];
float4    LightDir[8];

void Main( in Input In, out Output Out )
{
    // 插入点

    Out.Position = mul( view_proj_matrix, In.Position );

    float3 NormView = mul( (float3x3)view_matrix, In.Normal.xyz );

    // 光照
    Out.Color0 = AmbientCol;

    for ( float i = 0; i < LightCount; i++ )
    {
        // 方向光
        {
          Out.Color0.xyz +=   Diffuse[i] * dot( NormView, -
                               LightDir[i].xyz );
        }
    }
}
```

前面的例子演示了一个简单的光照流水线的 HLSL 代码，它使用的是可变数量的点光源。这个例子不能真的用 vs_1_编译目标进行编译，因为不支持循环和条件。

我们可以在插入点写上如下代码。

```
LightCount = 3;
```

这样就生成了代码的一个特殊变体，并使用通用的 vertex shader 把它特化成使用三个点光源的 shader。shader 编译器将分析这段代码，并发现可以展开三次的循环。还应该注意，如果把 LightCount 设为 0，shader 编译器将会优化掉整个光照循环和 NormView 的计算。

这就允许我们为游戏写一个统一的 shader。在通用 shader 中，我们可以忽略 shader 微程序在长度和寄存器方面的限制，因为特化的变体会去掉任何不使用的元素。

增加和减少法都有它们的优点。增加法更接近于重组 shader 的真正概念，它把 shader 片断粘合在一起，并允许每个 shader 片断独立地开发，也更灵活。减少法用起来更简单，而且通过让程序员预先汇编所有的片断简化了调试。

一些代码可能需要通过在 HLSL 代码中使用#define 和#ifdef 来做到“可切换”。不需要的顶点结构体成员（额外的 uv 坐标等）可以通过使用#ifdef 语句来裁剪掉。shader 组合器可以加入必要的#define 来根据需要增加必要的顶点成分。

5.10.3.3 混合

一个好的折中办法是把这两种方法混合起来。减少法用来做流水线中通用、标准的部分——蒙皮、光照、uv 动画，等等。同时，写一个大的通用 shader 来支持所有主要的流水线特性，如图 5.10.1 所示。在通用 shader 的代码中定义多个挂接点，让 shader 代码可以注入对

流水线插件的函数调用。这些流水线插件是特化的、可选的效果，比如卡通 shader 和阴影体。这样的流水线插件可以美工工具包的形式暴露出来，最后由了解技术的美工像程序员一样出色地编写。

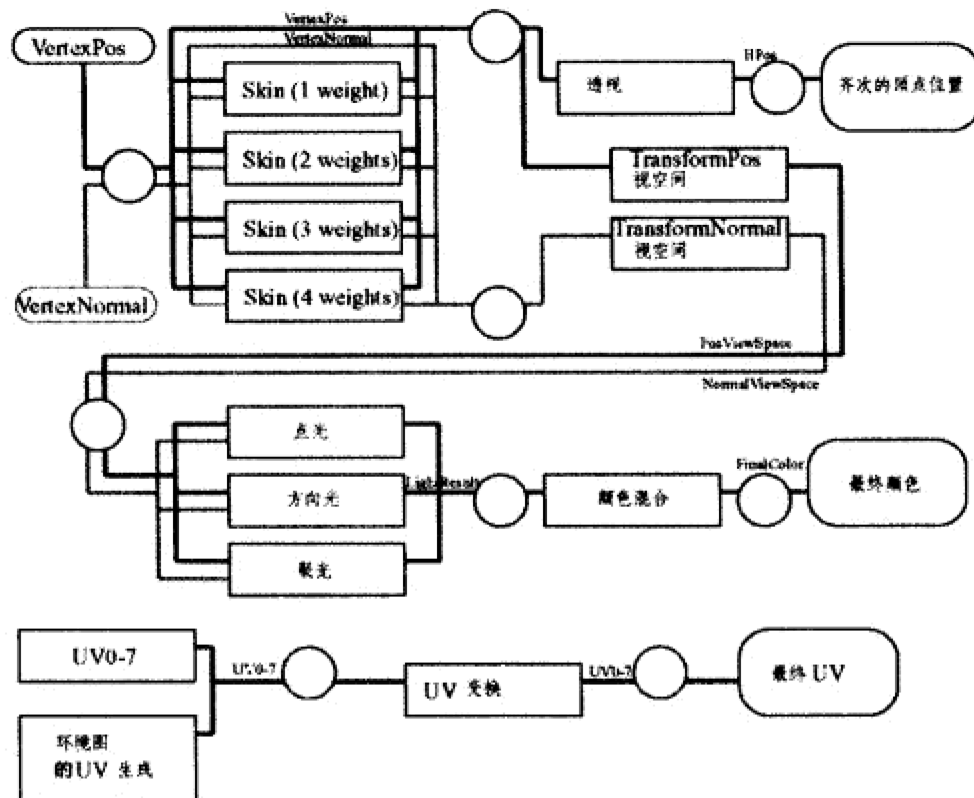


图 5.10.1 使用混合方法的流水线例子

对于可以以通用方式工作的技术，必须清楚地定义所有组件之间的通信渠道（函数参数和变量）并有标准的名字。它们都在图中表示通信渠道的线的上方标出来了。

在这个例子中，光照子组件需要视空间中的顶点位置。它可以自己计算，但为了避免对每个光源重新计算，这个任务就委任给了执行变换的 `TransformPosViewSpace` 组件。然后光照子组件可以通过 `PosViewSpace` 通信渠道从 `TransformPosViewSpace` 中获得视空间位置。必须仔细地计划流水线的组件，并产生最大的重用量，就像这个例子一样。

图中的圆圈表示挂接点，“外部”shader 流水线插件可以挂接到这里，在特定的注入点高效地传入通用 shader 代码。

5.10.4 整合重组的 shader

描述了让 shader 可以在运行期组合的方法之后，我们就可以检验如何高效地把这个系统绑到 3D 引擎中去。

大部分图形 API，比如 Direct3D 和 OpenGL，大体上都是状态机。可以设置多个渲染状态，并通过特殊的渲染状态把一组多边形发送给硬件。我们的 shader 系统渲染接口将用相同的机制。状态将通过一个结构设置到图形子系统中，并存储在 shader 系统中，直到 shader 准备好渲染下一组多边形为止。

通过记录所有的活动状态，shader 系统可以获得所有相关的状态，并把它们编码成 64 或 128 位的位域。例如，最低 2 个位可能用来给需要使用的蒙皮权重的数量编码，而接下来的 3 个位可能包含光源的数量，等等。这个 shader 键只描述需要激活的效果，不包含特

殊的参数，比如光源的位置、方向、颜色等，因为这些不会影响哪个 shader 片断应该连接进去。

shader 系统可以在 STL Map 中查出这些键。如果找到了这个键，就表示最近用到了这个效果的特定组合，因此预编译的 shader 可以立刻从 map 中获取。否则，必须生成一个新的 shader 变体。构建这个 shader 的方法是获取通用的 shader 代码，找到适当的插入点，使用减少 shader 组合来特化代码：把光照的数量、蒙皮权重数量等设置为常量。这个 shader 键可以用来推所有的代码特化。

要通过增加法来加入片断，shader 组合器必须找到哪个 shader 函数将要嫁接到哪个挂接点上。每个 shader 片断对每个兼容的挂接点都有一个标准的调用形式字符串，它将粘贴到主 shader 体的特定挂接点上。

这个 shader 被编译之后，我们就可以获取 shader 编译器，以找出这个 shader 需要哪些 shader 参数。这些参数可以连续上载到 GPU，从活动的渲染状态中获取它们。

生成的 shader 被添加到 map 中，并关联上它的键，由此给了我们一个 cache，存放最近使用的 shader。在生成 shader 的过程中，会计算它们编译后的二进制大小。如果一个新编译的 shader 使得 cache 超过了一个阈值，最先使用的 shader 将被销毁。

此时，shader 已经被编译并激活，而且参数已经设置好了。渲染器可以直接用它来渲染多边形组了。

5.10.5 通过 shader 建立完整的流水线

这里描述的方法让我们可以做一个灵活、统一的流水线，所有的 shader 效果可以组合在一起。用这种方法让 3D 引擎 100%的渲染都使用 shader 变体是完全有可能的。目前，这并不是在所有卡上都有理想的表现，因为老的视频卡仍然对固定函数变换使用特化的硬件路径，这可能使得标准的固定流水线比 shader 流水线更快。但是，诸如 ATI RADEON™ 9600 及以上的视频卡在内部会对所有的渲染都使用 shader，不管是否使用了固定函数流水线 API。

光盘中包含一个典型的固定函数风格的流水线例子，它完全是用重组 shader 建立的。这个样例流水线支持如下特征。

- 一到 4 个权重的蒙皮
- 任意数量的光源
- 带有镜面成分的点光、聚光灯和全方向光
- 环境映射
- UV 变换
- 多彩色模式：白色、材质颜色、顶点颜色、动态光照、顶点颜色 + 动态光照、顶点颜色*动态光照

5.10.6 其他问题

这一节涉及读者应该注意的其他一些问题。

1. Shader 版本 2.0 及以上

在只打算支持 shader 2.0 及更高版本的系统上，可能会觉得重组 shader 系统是没有必要的。shader 2.0 及更高版本支持更长的 shader，更多的分支、条件和循环。

虽然迁移到 shader 模型的更高版本意味着一定量的性能活动空间，但是短的 shader 仍然会比长的 shader 快，这个事实并没有改变。通过优化出主要的常用 shader 变体，重组 shader 仍然可以用于 2.0 版本以上的 shader。

2. 优化组合

shader 系统使所有的效果可以以各种方式进行组合，但必须小心以防生成过量的 shader 变体，因为当处理上千个 shader 变体的时候，任何机器都会崩溃。编译 shader 也会花费时间，在导出的时候，为静态物体预生成 shader 变体的任何步骤都有助于减少载入时间。某些相似的 shader 变体也可以重组成稍微不那么优化的 shader 变体，并依此表示某一类的 shader。

在建立游戏的特殊效果方面，重组 shader 系统给予了美工完全的控制，但是他们也必须明白不应该“每个多边形有一个效果”。切换 shader 通常会花费一些时间，因此使用相同 shader 键的多边形应该合在一起。

5.10.7 总结

让一个引擎基于 shader，却不至于落入只支持预定义效果以及引起设置上的不灵活，并非一件容易的事。然而，通过允许那些辛苦得到的效果以新颖和富有创造性的方法进行集成，一个构建良好的重组 shader 系统最终会带来好处。美工不需要知道 shader 流水线的完整知识就可以加入小的、特化的效果，甚至可以从头用积木建立出可以重组的新效果。不需要有程序员建立特化的冰雪 shader，美工可以用流水线中更为元素化的积木搭建出更复杂的 shader，由此开始制作他们自己的效果。通过重组 shader，美工真正完全控制了图形流水线。

5.10.8 参考文献

[Bean04] Bean, Scott. *ShaderWorksXT*. Available online at *http://www.shaderworks. com/shaderworks/shaderworks-main.html*.2004

[Boyd02] “Direct3D Tools.” Available online at *http://www.microsoft.com/korea/events/directx/ ppt/Direct3DTools.ppt*.2002

[Frick02] Frick, Ingo. “Visualization with the Krass Game Engine.” In *Direct3D ShaderX: Vertex and Pixel Shader Tips and Tricks*, 453–462.

[Lake02] Lake, Adam. “A programmable vertex shader compiler.” In *Game Programming Gems 3*, 404–412. Charles River Media.

[O'Rorke04] O'Rorke, John. “Integrating Shaders into Applications.” In *GPU Gems*, 601–616. Charles River Media.

[Pharr04] Pharr, Matt. "An Introduction to Shader Interfaces." In *GPU Gems,* 537–550. Charles River Media.

[Sander03] Sander, Pedro. "A Fixed Function Shader in HLSL." ATI Whitepaper, October 2003. Available online at *http://www2.ati.com/misc/samples/dx9/FixedFuncShader.pdf.*

第 章

6

网络和多玩家

引　言

Shekhar Dhupelia
sdhupelia@gmail.com

支持网络的在线游戏数量正在增长。其中有些提供了独立的“离线”和“在线”模式，而越来越多的游戏完全是在线的。虽然硬件和底层技术在过去的几年内保持稳定，但是各种平台上和各个游戏中的服务质量正在逐年变得越来越好。证据就是，有越来越多的人在玩大型多人在线游戏（MMO），而且 Sony 和 Microsoft 各自的在线游戏服务也日渐流行。

随着越来越多的游戏在一开始设计的时候就利用了这个玩家社团，越来越多的团队都雇了专职的网络工程师和在线游戏设计师来专门研究游戏性。很多问题已经被解决了，而且在一段时间之前就被完善地包装起来，我们每天都有很多新课程可以学习。

本章及各种类型的在线游戏，会给大家带来有用的信息。从 MMO 开始，Shea Street 解析了如何使用一个分布式服务的方法来更好地处理服务端。接着是 Patrick “Gizz” Duquette 的文章，内容是关于实现无缝的世界服务器，其中没有今天的游戏中出现的区域到区域的转换。下一篇是概要介绍多种过滤脏话的方法，适用于所有类型的在线游戏。然后 Hyun-Jik Bae 详细地解释了如何在用户/服务器架构游戏中的网络层使用 RPC 调用。

很多有宽带连接的人都在家安装了 NAT，Jon Watte 解释了如何最好地穿越这些限制。然后 Martin Bromlow 描述了如何为游戏通信设计一个可靠的消息层，并演示了在线游戏中比 C/C++更安全的随机数系统。最后，Adam Martin 回到了如何在安全方面设计游戏的话题，不管游戏的类型是什么，注重安全能提供更好的游戏体验。

不管是在一个大预算的游戏中实现在线功能的商业游戏开发人员，还是想在下一个伟大的 demo 中增加网络功能的独立开发人员，这些文章都能给人以鼓舞，并可在未来几年内作为参考。

6.1 保持大型多人在线游戏大型、在线和永存

Tantrum 游戏公司，Shea Street
shea.street@tantrumgames.com

在这个有快餐、快车，甚至更快的 Internet 的世界中，我们努力保持着东西被大量地生产，让它们总是可用，而且是高质量的。建立和维护一个大型多人游戏也是一样。玩家希望游戏也是如此：大规模、总是在线，而且从不会慢一拍。不幸的是，我们生活的世界总是缺少时间而且充满不可预知的事件。但是，在某个层面上，我们可以把这些东西提供给玩家。本文将讨论如何通过扩展到分布式服务的方法[Street05]来达到这些层面。

6.1.1 快速浏览

分布式服务方法是一种服务系统，它分布在多个服务器上，这些服务器连接成一个统一的网络（参见图 6.1.1）。

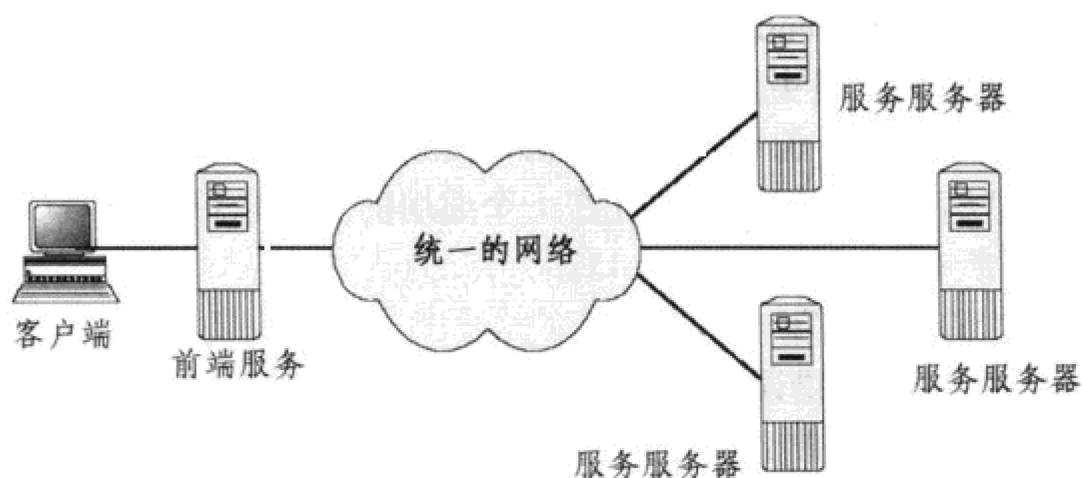


图 6.1.1 分布式服务系统图

这些服务向整个游戏世界提供任务特定的游戏功能。服务为游戏世界提供了一种方式，可以把负载分布到独立的任务基上。例如典型的游戏服务包括聊天服务、物品服务、位置服务、AI 服务和战斗服务（参见图 6.1.2）。通过使用分布式服务，游戏的整个模拟现在不再只运行于单台服务器上。

PDG

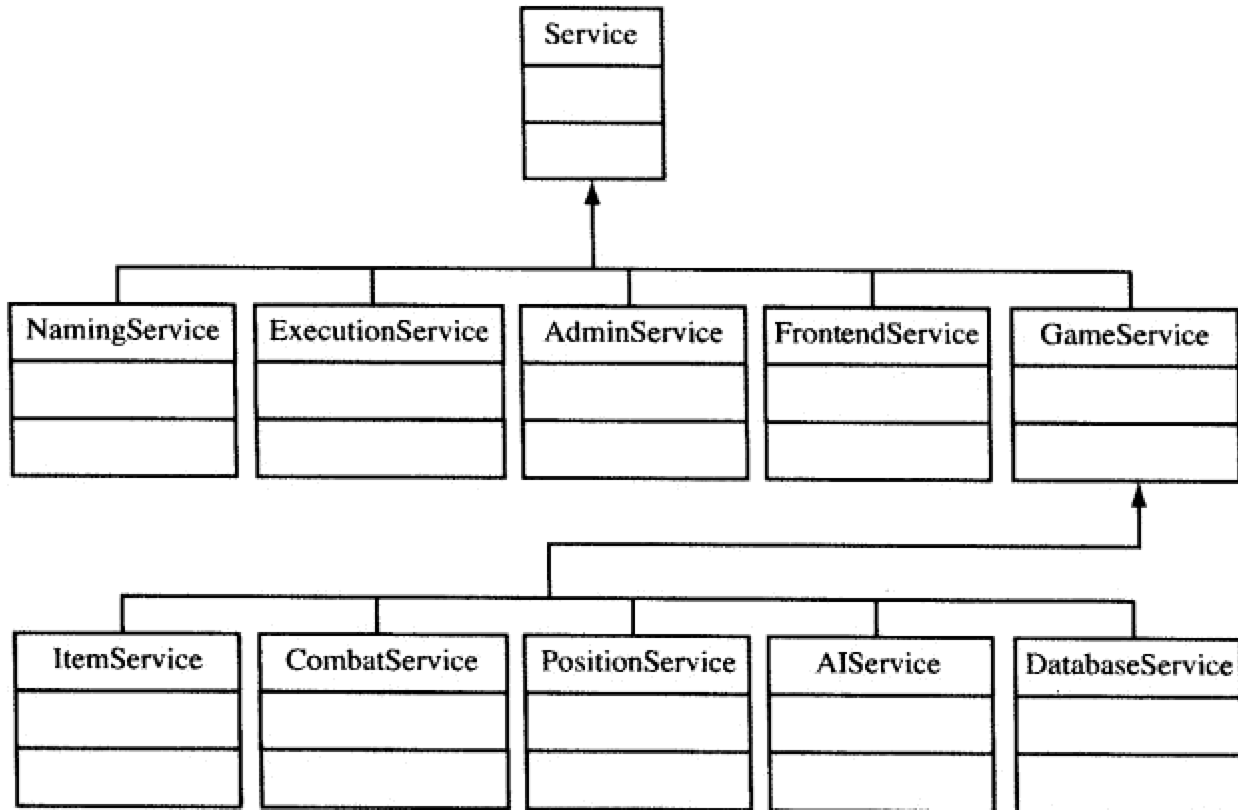


图 6.1.2 分布式服务的分解

6.1.2 大型化

随着在线游戏总体数量的增长，同时支持更多玩家的需求也在增长。有许多方法可以解决这个障碍。

1. 优化前端服务

前端服务（frontend service）在整个分布式服务系统中是最难的工作。乍一看，它似乎只是另一类的网关，但实际上它远远不止那样。理解前端服务的总体范围和优化这个过程的最佳方法是很重要的。但在优化它之前，需要正确地定义它的操作目标。这些目标和它们的实现都是游戏特定的，但仍是建立在一个通用的基本思想框架上。这些目标包括：

- 对所有玩家都是一个入口；
- 作为防守的第一线；
- 执行健全和错误检查；
- 管理和平衡带宽的使用；
- 确保对游戏信息的正确路由；
- 提供尽可能快的响应时间。

前端服务充当着介于客户端和游戏后端的中间人。它把所有后端服务的数据浓缩到一个过滤的可用流中，提供给所有连接的玩家，反之亦然。通过使用游戏特定的知识，前端服务可以提供更快的玩家响应时间，并给后端节省无用的流量。建立流程图并使用用例（use case）非常有助于定下目标，研究和开发这些可能的优化（参见图 6.1.3）。

一个可能的优化是外加一个空间局部的数据库。这个数据库可以是一棵球体树（sphere tree）[Ratcliff01]，它容纳了位置、状态数据和静态统计数据。因为这个数据库需要包括游戏中目前活动的所有玩家、物品和 NPC 的数据，所以这些信息越简单越好。当设计正确的时候，

这个空间数据库将占有非常少的资源，并允许前端服务做更多的优化工作。使用这个数据库和局部缓存信息的理由是允许快速输出。如果系统不需要越过前端服务以提供给客户端信息，那它就永远不会这么做。同时，当一个玩家执行了一个动作，连接到同一个前端服务的其他玩家就会快速地感觉到，它也会被传播到系统的其他地方。执行健全检查并让通信保持在“必须知道”的基础上可以防止后端被淹没和负荷过重。在所有服务的情况下这样做都是正确的；一旦这些方法都实现了，就有一些后续方法可以进一步保证质量和稳定性。

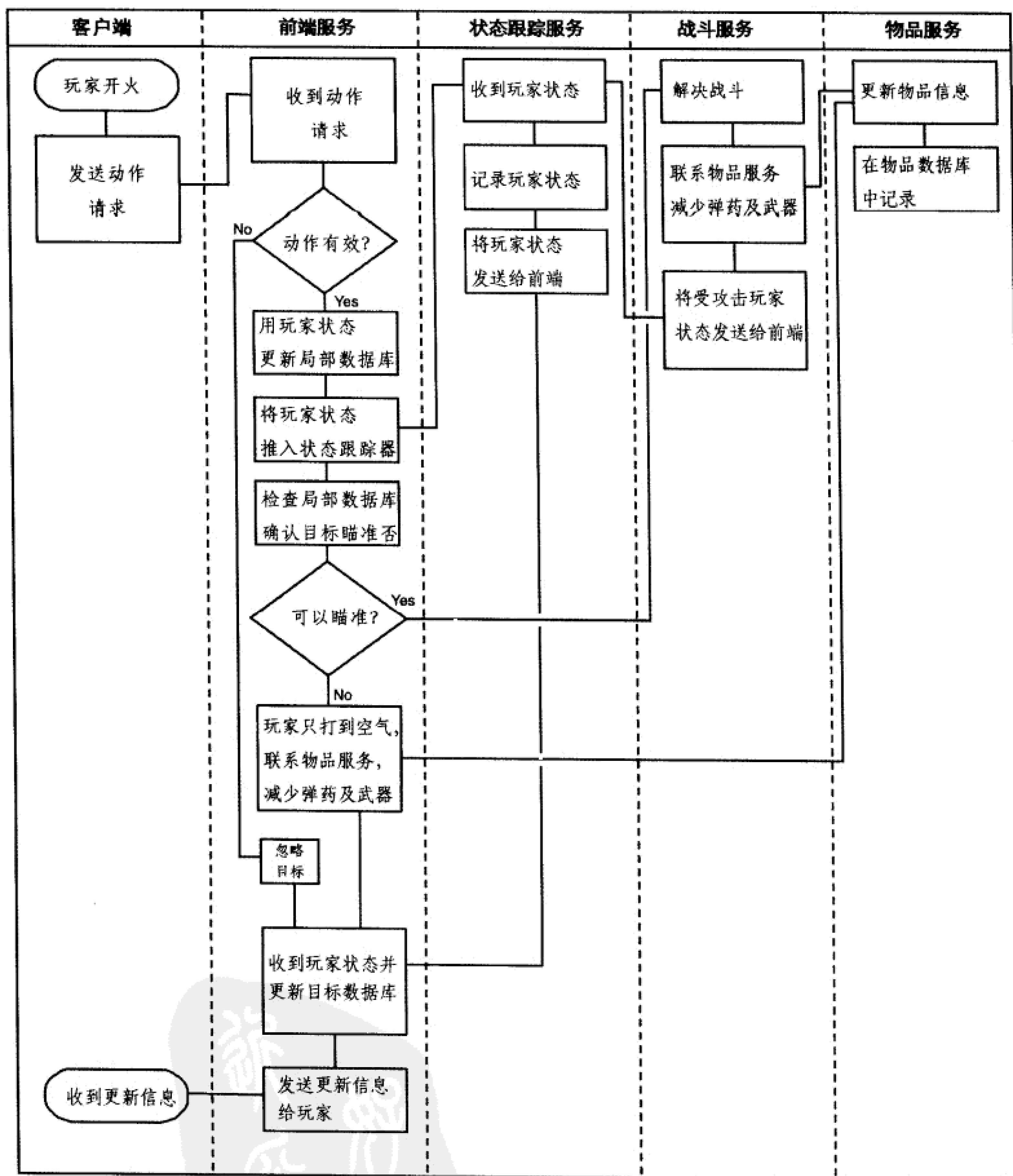


图 6.1.3 前端服务优化流程图

2. 增加和升级服务器

当提到增加或更新服务器的时候，我们主要关心处理前端服务运行的机器。分布式服务系统支持的玩家数量直接与这些前端服务的数量和能力成正比。这个系统被设计为只要增加更多的前端服务，就可以支持更多的玩家，而不用接触到后端。这并不意味着从来不用改变。随着游戏的成熟和改进，它可能需要更多的服务来完成它的目标，因此需要更多而且更强的服务器来达到这一点。

总的来说，这个过程很简单。一般来说，增加或升级一个硬件所要做的事情就是安装需要的服务，并设置管理服务，来指定何时这些服务应该如何在这些特定的机器上运行。分布式服务方法总有一个目标，就是可以使用消费级的普通硬件，但是所需服务器的绝对数量可能无法控制。在某种程度上，转用企业服务器解决方案或混合方法可能会更有效，这样可以节省空间。这个设计保证了游戏不会少于或多于它对某个功能的绝对需要。增加和升级服务器的整体概念大体上是显而易见的，但是在处理分布式服务系统的时候需要牢牢记住。

6.1.3 保持在线

没有办法保证无缺陷的代码。最重要的事情是避免一次崩溃就带倒了整个游戏。当一个系统崩溃的时候，它必须对玩家透明。这涉及监视崩溃，保持跟踪有缺陷的服务版本，并很快地恢复游戏运行。

1. 看门狗和影子服务

如果游戏的任何部分曾经崩溃过，那就需要知道它崩溃的时间，以及如何让它恢复并尽可能快地运行。最好的方式是使用可以处理大部分常见崩溃原因的自动进程。

看门狗服务（*watchdog service*）是一个程序，可以监视其他运行的服务是否有任何预期之外的终止情况。在一些时候，被监视的服务可以捕获中止信号，并执行一些形式的关闭过程。作为预防，看门狗会注意这些服务，所以即使有那个程序不能处理的突然终止，看门狗也会发现。当发生的时候，看门狗将启动等待的影子服务（*shadow service*）。

影子服务是它们对应的服务程序的拷贝。每次启动一个服务的时候，那个程序的影子都会作为服务的克隆来建立。虽然影子服务是其对应服务程序的一份拷贝，但是它会保持休眠，等到那个服务出现了重要的、突然的终止才启动。看门狗和影子服务可以组合进同一个程序中。

2. 热切换服务

在硬件失败的事件中，看门狗和影子服务就没办法了。那就可能需要有独立的硬件随时待命，在发现失败的时候尽快取代失去的服务。

管理服务将确定是否出现了这种情况，并负责告诉运行服务（运行在其他的服务器上）启动必要的服务。如果没有可用的服务器，管理服务将把一个负荷不重的服务器变成临时备用的服务。

PDG

3. 服务版本控制

如果一个重要的事件造成了服务的终止，那就需要用一种方法来标记它，并返回到稳定的版本。在服务异常终止的事件中，它的版本将被做上标记，出现一定量的崩溃之后，它将开始还原到前一个版本。一张服务版本表（*service version table*）将用来区分当前、稳定、不稳定和不可用的版本（参见表 6.1.1）。

表 6.1.1 服务版本表

服务名	版本	标记	服务名	版本	标记
聊天服务	1.32	当前，稳定	聊天服务	1.0	稳定，不可用
聊天服务	1.24	不稳定	物品服务	2.23	不稳定
聊天服务	1.11	稳定	物品服务	2.0	当前，稳定

当一个服务被还原到前一个版本的时候，影子服务会载入那个服务的前一个稳定版本，并负责控制。影子服务不会返回到前一个标记为不稳定的或者不可用的版本。取而代之的是，影子服务将保证总是返回到稳定的版本上。

4. 在线更新

有了上面提到的方法，现在就能明智地处理在线更新。把更新推到还原列表中，让服务器启动待命的新服务，把控制从旧服务切换到新服务上，然后关闭旧服务来清理资源。

6.1.4 保持永存

为了从崩溃的事件中恢复功能，数据必须持久地存储，不仅仅只是放在统计数据库中。为了从终止的地方继续，我们需要正确的游戏状态数据。目前，有些在线游戏每隔 20 分钟就把全部游戏状态写到磁盘上。如果出现了崩溃，那么保存的游戏状态会被重新载入。它的问题是，因为我们不知道什么时候会崩溃，所以状态会是 20 分钟之前的。

1. 备份游戏状态

这里的解决方案是共享内存（*shared memory*）。任何载入共享内存的东西都会保持，直到显式地销毁它，所以服务把它的游戏状态数据存放在这块内存中是安全的。在非硬件崩溃的事件中，影子服务只需要挂接到它的共享内存段，就能有前面所有的数据。另外，每隔几分钟就可以做一次共享内存的快照，并把它复制到另一端的内存中作为备份。通常，这个过程会很快，并占用很少的资源。然后在空闲的时候可以在后台把这些备份内存块备份到磁盘中。只需要把一个消息发送给所有服务就能在服务之间同步这些游戏状态快照。因为每个服务执行一个特殊的任务，而这些任务以很小的步子完成，所以它们可以处理当前的事件，队列化所有新到的事件，拷贝当前的共享内存，然后恢复正常操作。一旦发生了崩溃事件，一个新的活动服务可以开始把它自己的快照发送给其他服务器。如果这样做，就可以随时在网络上找到游戏状态备份，并在需要的时候轻松地获取它们。

2. 恢复保存的游戏状态

根据游戏的架构，在服务崩溃的时候有两种方法可以恢复一个保存的游戏状态。第一种方法适用于游戏的状态需要时间同步的情况，所有的服务都需要恢复。第二种方法用在服务结构足够独立，允许只恢复一个服务的情况。

当只有一个服务需要恢复的时候，系统会找到失败的服务器，定位一个剩余资源足以运行所需服务的服务器，并告诉它启动那个服务。然后带有死服务的最新快照的那个运行服务器会把它的快照推给新的服务器，系统会指挥那个服务载入它前面保存的快照。

然而，有的游戏架构需要在一个服务失败的时候恢复所有服务，此过程基本上是一个多米诺效应。和以前一样，系统会定位一个剩余的服务器，把死服务的保存游戏状态推给它，然后指挥这个服务器载入前面的死服务，并成为休眠状态。载入这个服务之后，系统会关闭每个独立的服务，让它们重启，并载入相同时间戳的快照。所有服务都重启之后，系统会恢复普通的游戏操作。游戏状态的恢复是基于快照保存和发送到其他服务器的频率。

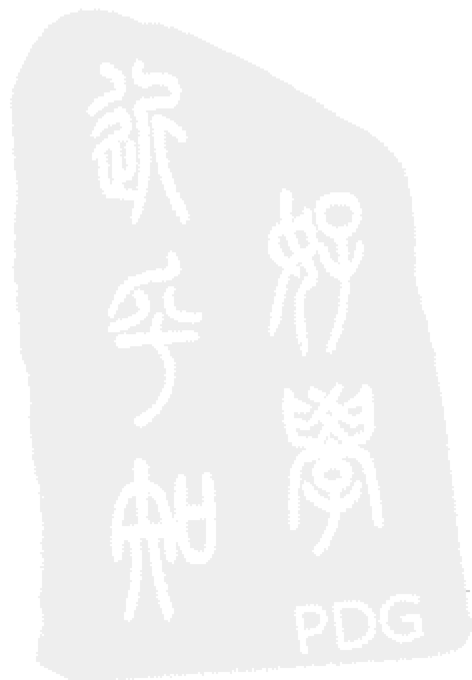
6.1.5 总结

在线游戏正在快速地成熟和发展玩家也是如此。他们的需要和期望将会越来越快地持续增长。由于有了一些进步，以前的问题将不再视为可以接受。分布式服务方法和本文讨论的这些改进只是一个用来帮助维持这个步调的工具。有了所有这些工具之后，现在我们就可以满足当前玩家的需要了。最后，大小、持续时间和连续性是赢得玩家的心的几个关键因素。

6.1.6 参考文献

[Ratcliff01] Ratcliff, John W. “Sphere Trees for Fast Visibility Culling, Ray Tracing, and Range Searching.” *Game Programming Gems 2*. Charles River Media, Inc., 2001.

[Street05] Street, Shea. “Massively Multiplayer Games Using a Distributed Services Approach.” In *Massively Multiplayer Game Development 2*. Charles River Media, Inc., 2005.



6.2 实现一个无缝的世界服务器

育碧公司，Patrick Duquette
gizmo@gizz-moo.com

在[Beardsley03]中，作者介绍了无缝世界服务器，以及它们的优点和缺点。他让我们思考了很多无缝世界固有的问题。本文讲述的是游戏服务器设计，一个欠未公开的话题。它将专注于一个无缝世界服务器的真实实现——由世界节点、一个代理和一个登录服务器组成。本文讲述了在这类服务器的设计中通常做出的决定，并以“由此而往何处”作为结语，为一些领域的未来探索提供思路。

6.2.1 一些定义

根据定义，一个无缝世界（*seamless world*）中的玩家可以自由尽兴地漫步、探险和游历。在“区域”之间没有物理屏障阻碍我们的旅行，而且在区域间转移的时候没有载入画面。在无缝的世界中，设计师不再需要在区域边界之间放迷宫，而且限制很少，他们的想象可以自由驰骋。

虽然无缝世界有利于任何游戏类型，因为它是服务器的核心，游戏特定的世界可以存在于它的上层，但是沉浸感的增加会根据类型和游戏性的设计而不同。RPG 游戏会受益于一个玩家可以游历的巨大的、连续的世界。而 FPS 游戏可以让世界延展到辽阔的地形中。去掉物理区域极大地推动了玩家在游戏中的沉浸感。

当然，自由的层面会带来开销。美工需要建立过渡区域来连接不同的气候，就像真实生活中那样。在传统的有区域世界中，可以让亚热带区域的旁边就是霜冻的山区，而不让玩家感到迷惑，因为你认为世界的两个部分有着明显的分隔，而且在改变区域或载入的时候有时间从游戏中“断线”。它可能会去载入新地图，你希望载入结束之后，看到的东西会有所不同。

从程序员的角度看，挑战几乎都在边界附近。当一个玩家从一个服务器移到另一个服务器或者在边界区域交换物品的时候，如果不正确地处理，那么由不同服务器管理的玩家可能会出现潜在的问题。另一个问题是随着我们在服务器之间的移动，可见物品的载入。如果不希望玩家注意的话，它就需要在后台完成。

因为不同服务器上的玩家、NPC 和物品之间可以有交互，我们就必须注意内部从服务器到服务器发送的消息数量。如果不注意设计服务器内部的通信细节，消息数量就会立刻爆炸式地增长。即使今天的网络硬件远能

够管理我们期望的 LAN 流量，我们仍然必须注意不仅要发送数据包，而且接收器也必须处理它们。服务器之间的数据交换越少意味着需要的处理越少。

6.2.2 实现

先简要介绍一下无缝世界中出现的几种不同类型的服务器：

远程控制器：登录协调器并负责发送 OK 给代理服务器，以开始接受客户连接。

代理服务器：外部世界和我们的服务器布局之间的桥。

登录服务器：验证用户。

节点服务器：管理世界的一个区段（segment）。

世界管理器：把世界区段分布到节点服务器中。

从一开始，最少有三个服务器：代理服务器、登录服务器和节点服务器。因为节点服务器的数量可以变化，而且我们不想手工向 ini 文件中添加东西，所以将使用一个自动注册系统。不同的服务器在启动的时候将把它们自己注册到代理服务器（嗯，实际上是世界管理器，但那将在后面讨论）。这个策略让我们可以随意地改变节点服务器的数量，而不用改变 ini 文件的任何一行。

6.2.3 远程控制器，或如何管理服务器的启动时期

当然，这么做会出现另一个潜在的问题。什么时候开始接受接入的连接？天真的解决方案是在代理开始接受连接之前，在上面放一个等待延迟。虽然这可以奏效，但是它不够灵活，而且非常危险，因为你不知道是不是所有东西在上线之前都已经启动好了。

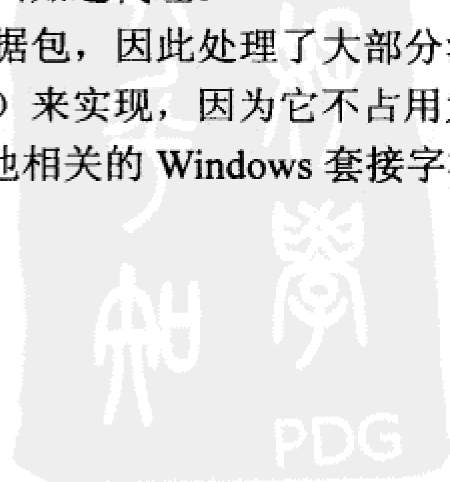
第二个选择是让代理等待一个特殊的数据包，包含一个“上线”的命令。这种方式可以精确地控制代理服务器什么时候上线。这个数据包只有在远程控制器的所有条件都达成了之后才会发送。这些条件包括：确定登录服务器启动，世界管理器把世界区段发给节点服务器，等等。

远程控制器不仅仅是一个实用的工具，而且是一个调度服务器启动时期的无价工具。另一个选择是手工控制代理服务器，那会导致人为错误和其他伴随的东西。

6.2.4 代理服务器

和许多在线游戏一样，你并不希望把服务器架构暴露给外部世界。达到这一点的一种方式是让所有的通信都通过一个代理服务器，让它转发给正确的接收器。这项技术的优点是只有一个对外的入口点，对客户端隐藏了真实的服务器分布。如果需要改变服务器分布、数量或内部协议，你可以那么做，客户端会一无所知，因为它们只知道代理。

服务器将处理所有客户端和服务器之间到来和发出的数据包，因此处理了大部分流量。时刻要记住的是，代理服务器可以使用 IO 完全端口（IOCP）来实现，因为它不占用太多资源，并可以服务很多并发的客户连接。对 IOCP 的介绍和其他相关的 Windows 套接字模型，请参考[Jones02]。



在实际安装中，可能希望有很多代理服务器来分担负载。客户端可以连接到一个主控的重定向器，它将根据当前负载和从客户端的延迟（ping 时间）告诉客户端应该用哪个代理服务器。这可以让代理服务器分布在世界中，而对客户端只保持着一个连接点。

当代理服务器收到一个来自未知客户端的连接时，就会把它转发给登录服务器来验证。一旦用户通过了验证，登录服务器就会通知客户出生点的代理服务器和处理出生位置的节点服务器。

客户连接在内部通过一个简单的数组进行维护，这个数组做了从 ClientID 到它应该连接到的节点服务器之间的转换。ClientID 是数组的索引，当代理服务器从登录服务器收到验证确认的时候，由它处理 ClientID。代理服务器在启动的时候会分配这个数组，知道它能支持的最大连接数。为了快速管理客户端的连接和断线，它还保存了一个无用连接的列表。为了最小化恶意用户在客户连接的时候做出欺骗的可能性，我们在它将要连接的节点服务器上保存连接细节。偶尔，要检查收到的数据包是否真的来自于合法的客户端。

6.2.5　登录服务器

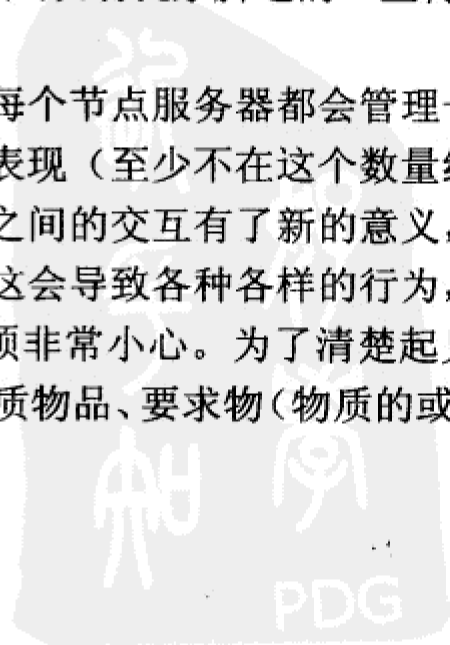
登录服务器处理的是客户的验证。登录服务器负责检查玩家的信息，得到出生点的坐标，找出正确的节点服务器，并把连接切换上去。这通过要求节点服务器处理的那部分世界的世界管理器来完成。登录服务器会处理数据库查找并通知世界管理器，因为我们希望代理服务器尽可能地轻量。

玩家的信息存放在标准的 SQL 数据库中。为了让实现更加简单，mySQL [mySQL]可能是一个好的选择，因为它不仅可以处理小的需要，而且是自由的。实际安装中可能需要参考其他数据库解决方案，包括[Oracle]和[SQLServer]。在产品的环境中，可能也要考虑在数据库前端有代理/队列服务器，以根据重要性队列化获取/更新。同时，如果可以的话，把数据库存放在内存中。当然，偶尔需要一次性保存物理内存，以在服务器崩溃的情况下有一个数据库的物理拷贝。但是，尽量保持最小化，而且如果可能的话，应在另一个服务器中来完成这件事（使用复制）。

6.2.6　节点服务器

节点服务器是世界的存放者，是世界中的客户表示和内部存放之间发生冲突的最终仲裁者。它们执行的是玩家和他周围的交互的健全检查，以使所有东西都保持整洁，而且还给游戏世界中的很多部分带来生命。例如，虽然它不能直接控制一阵风吹动玩家脚边的一些树叶，但是它将提醒玩家，现在正吹着某个方向的 5 km/h 的风。

因为在线游戏试图让玩家探索一个巨大的开放世界，所以每个节点服务器都会管理一个小区段。由于需要分割游戏世界，传统的划区域在线世界没有表现（至少不在这个数量级）的无缝世界实现就会有不可忽视的复杂开销。玩家和/或 NPC 之间的交互有了新的意义，因为当交互开始的时候，它们现在可以在两个不同的服务器上。这会导致各种各样的行为，当在设计让玩家获取/交换/给予“物品”的事务系统的时候，必须非常小心。为了清楚起见，任何可以在玩家或 NPC 之间交换的会影响游戏的东西，比如物质物品、要求物（物质的或“口



头的"），等等，将被放到"物品"类之中。

为了管理跨边界的交互，节点服务器不仅需要管理在它们区段中的世界，而且还需要知道外面的东西，我们把它叫做边界区域（*Border Zone*）。边界区域略微伸展过两个区段的边界。这个区域，虽然属于其他服务器，也需要让节点服务器告诉它的客户端已经接近该服务器已知区域的边界，即使它已经位于另一个节点服务器管理的部分世界。

为了让同一个对象同时出现在两个服务器上，我们必须引入代理对象的概念，它是一个服务端对象，表示在另一个节点服务器上的主控对象的幽灵。当两个在边界不同边的玩家之间开始一个事务的时候，服务器将在代理对象上执行事务，而不是执行真的位于另一个节点服务器上的副本。

边界区域的最后一个内容：如同[Beardsley03]和其他关于这方面的文章中提到的，边界区域的最小大小必须至少是玩家认知半径的大小，以避免两个在边界不同边的玩家可能看不到相同的东西，更糟糕的是一个可能看不见另一个，并因此打开了问题之门。同时，如果让边界区域略微大于玩家的认知半径，那在新对象作为代理复制到相邻的节点服务器的时候，将减少视觉弹出的可能性。

1. 性能考虑

服务器需要处理很多数据包，不管是外部世界的（通过代理服务器）还是内部的（服务器之间的通信）。它还需要管理一个大的物体基点和移动角色（玩家和 AI），需要更新游戏特定的东西，比如天气变化和 NPC 甚或其他玩家提出的要求/任务。如果在线世界不跨越多个太阳系，那么星体状的对象，比如星星和太阳/月亮应该有全局服务器管理，而不是节点服务器。

它向每个服务器增加了很多处理工作。为了减轻节点服务器的一些额外处理，就需要占用更多的服务器之间的带宽，让游戏 AI 实体在单独的专用服务器上管理，并被节点服务器看做普通的客户端。

世界被分成了很多区段，这会造成很多的边界和很多边界接合点。简而言之，一个简单的起点可能是一个条带的世界，每个节点服务器只有两个边界接合点，没有"T"或"交叉"接合点，如图 6.2.1 所示。

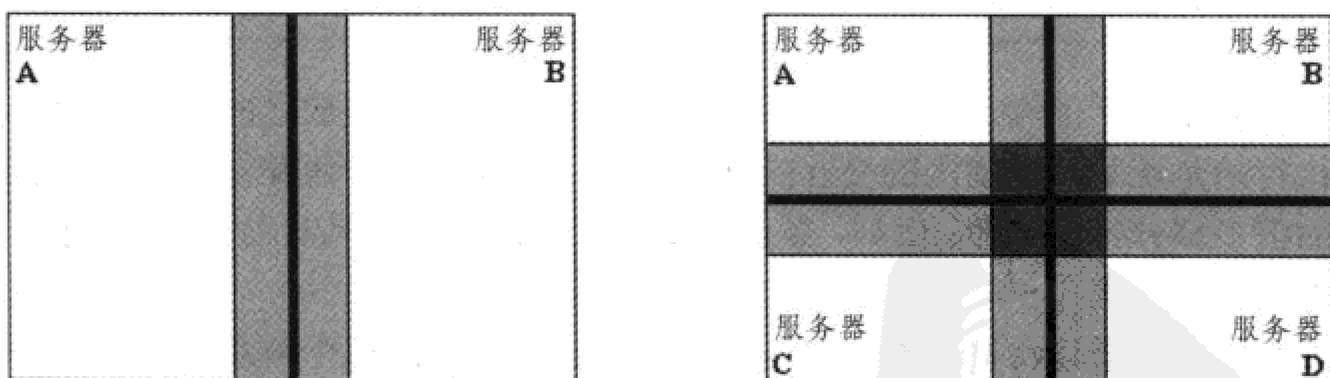


图 6.2.1 两个边界接合点和"交叉"接合点

2. 注册到代理服务器

当注册到代理服务器的时候，节点服务器仅告诉它"我已经在线，并准备好服务了"。

并没有确定世界的哪个部分由这个服务器管理。当然，一个 ini 文件可以解决这个问题，但那意味着每次都要对所有服务器手工管理这个文件。取而代之的是，那个服务器应该从另一个服务器——世界管理器——得到它们的“工作顺序”。在启动之后，注册到代理服务器之前，节点服务器和世界管理器协商得到所需的世界区段。每个节点服务器都有一份完整的世界。虽然当更新游戏内容的时候，这会造成一个负担，因为必须复制到所有的节点服务器上，但是它也使我们不必让所有的节点服务器在启动的过程中同时获取所有的世界数据。把所有东西都保存在局部还允许动态改变节点服务器边界的大小，这是“由此而往何处”一节要讨论的技术。

这个服务器的大量工作是验证玩家输入。比如物品交换/获得，玩家在世界中探索，或战斗中的攻击速率，节点服务器必须处理所有来自玩家的东西。这是旧的网络惯用法，最好“从不信任客户端”。

为此，服务器必须保持一个世界的模拟，并在客户端和服务端的表示发生不同的时候，将这个模拟作为参考。如果有东西不保存在节点服务器的主副本之中，那么必须通知客户端，而且必须做一些事情来校正这个情况。

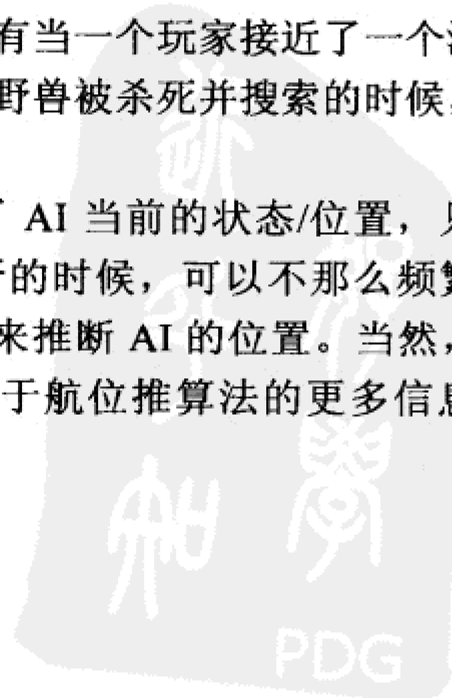
3. 处理和外部世界的交互

要知道这个服务器需要一个快速的方法来帮助到来的网络通信到达它的目标管理对象，我们必须保证它尽可能地高效。管理世界中的对象映射要有序，因为一个列表遍历可能会耗费太多时间。映射的缺点是插入可能成为连接处理的瓶颈之一，但因为在一个时刻不会有太多的对象建立/删除，所以我们不会感到任何减速。另一个可能是让世界中的每个对象都有一个惟一的 ID，并使用和代理服务器相同的技术。这甚至可能比映射搜索更快，且没有插入/删除的损失，但是内存消耗更大。（想象一个包含所有世界中对象的数组。）如果对象在跨越边界的时候发出了一个新的 ID，你只需要保存一个数组，其大小是在这个节点服务器上任何时刻会呈现的最大数量的对象。加上在紧急情况下增长这个数组的可能性，那就都完成了。

每个对象都表现在服务器上，网络消息可以索引到它们。当一个玩家和 NPC 交互的时候，玩家的动作会发送到服务器并验证。当一个玩家试图攻击一个怪物的时候，它会发送一个消息给节点服务器，告诉它“我瞄准了怪物 X”。节点服务器会验证玩家是否可以看见那个怪物，是否可以从他站立的地方瞄准它，以及是否允许他瞄准它。然后玩家会发送一个“我击打怪物 X”的消息。节点服务器会验证玩家是否近到能够打到怪物 X，并告诉怪物它是否被打到了，如果是，就告诉玩家他造成了多少伤害。如果这些步骤中有失败的，节点服务器需要记录这个过程，用于事后的剖析。它可能是延迟的问题或者是玩家试图欺骗节点服务器。

客户端应该总是收到必须知道的基本世界信息。只有当一个玩家接近了一个游戏对象的时候，它的相关信息才会发给玩家。例如，只有当一个野兽被杀死并搜索的时候，服务器才会发送掠夺的描述。

当 AI 频繁地在服务器上更新的时候，消息包含了 AI 当前的状态/位置，只需要立刻发送给客户端，每秒 10 次，在没有要求更频繁的更新的时候，可以不那么频繁。客户端将使用航位推算法（dead reckoning）根据上一次更新来推断 AI 的位置。当然，客户端在收到来自服务器的更新的时候，必须做一个校正。关于航位推算法的更多信息，请参考[Aronson97]。



4. 一个主控对象的X个代理对象

如何保持一个主控对象和未知数量的代理同步？因为对于任何给定的“主控对象”，可以有未知数量的代理对象，所以我们需要在内部状态改变的时候让所有人都更新它们。幸运的是，有一个设计模式就是为此设计的：观察者（Observer）模式（也称为发布-订阅模式）。这个模式的正式说明（和其他很多内容），可参考[Gamma95]。简单来说，这个模式定义了对象之间一对多的关系，用来同步它们之间的对象。其美妙的地方就在于我们不需要预先知道会有多少对象。

用这个模式让代理通知主控对象的变化，主控对象会把这些改变复制到注册的代理对象中。虽然没有真正相关于当前的实现，但是在以前的版本中，它是在边界接合点多于两个的节点服务器上完成的。主控对象最后会仲裁代理对象和主控对象之间有差异的情况。

5. 代理对象通知

因为代理对象可以随着主控对象移过世界而建立和删除，所以我们需要一种确定的方法来管理它们在节点服务器上的生命范围。代理是观察者模式的一部分，当主控对象移到边界区之外的时候，它会覆盖不需要对象的死亡，但当主控对象从管理的世界区段中心移到两个服务器之间的边界区域的时候，会发生什么呢？这个时候不会建立代理对象。主控对象，当越过边界区域阈值的时候，会提醒正确的相邻节点应该建立一个代理。代理一旦被建立，就会注册到主控对象上，然后主控对象将发送和更新对象的状态，并启动观察者关系。

6. 邻居

节点服务器可以从世界管理器得到它的邻居。这可以在启动的时候做，也可以在任何需要检查是否有动态边界的时候做（参见本章结尾的“由此而往何处”一节）。在这个简单的“条带世界”的例子中，只有两个邻居。我们的实现由此得到了极大的简化，同时，只需要有两个成员变量就能保存关于邻居的信息。如果没有设置信息，就请求它们所在的世界管理器，然后和它们建立一个通信通道，保持到服务器关闭为止。

6.2.7 世界管理器

世界管理服务器负责把世界分割成不同的管理区段，并把这些区段交给不同的节点服务器。当管理器分发出所有的世界区段之后，它会发一个消息给远程控制器，告诉它世界已经分布好了。这能确保在世界的区段还没有被管理，即不可访问之前，就接受玩家的连接。

在带状服务器的设计中，连接到世界管理器并申请一片世界的后续请求是被忽略的，因为没有提供备用的节点服务器；这一点将在“由此而往何处”一节中讨论。同时，世界区段的数量在启动的时候已经知道了，所以当节点服务器连接的时候，它们会立刻被分配一个世界区段。

世界管理器的另一个任务是回复来自登录服务器和节点服务器的请求，关于哪个节点服务器处理世界的哪个部分。这对一个玩家从一个会话切换到另一个会话的情况很重要，如果服务器关闭了，我们就不能保证那个节点服务器会处理那个玩家断线时的那部分世界。这可能比较奇怪和低效，但它让我们从手工指定每个节点服务器的世界边界中解脱出来，而且这

PDG

样就可以自动处理把新的节点服务器引入到我们的分布之中的情况，不管是因为扩充还是因为一个服务器死机。

6.2.8 由此而往何处

动态而不是静态，管理世界区段，是有待改进的一个领域。如果一个节点服务器过于拥挤，世界管理器会收到一个启动备用处理的请求。在需要的时候，通过改变边界大小或把当前世界区段切分成两个，把另一半交给备用节点服务器，可以减轻服务器的负担。节点服务器必须支持管理具体的直接传输，而不用通过代理对象的创建（一般与跨节点服务器边界的实体相关）。

让备用节点服务器获取垂死的节点服务器的负载也有助于从硬件错误中恢复，而不会中断服务。如果一个节点服务器监视着硬件消息，它就可以对此起作用。不必因为一个多余的电源关闭，就为了维护而关闭游戏服务，节点服务器可以告诉世界管理器它的状态。世界管理器会启动一个备用节点服务器，并把垂死的节点服务器所管理的世界区段赋给它。当所有管理实体的传输完成之后，垂死的节点服务器就可以安全地移除了。

因为实体传输的绝对大小，我们可能要渐进地去做。如果把实体负载从节点服务器 A 传输到节点服务器 B，在一个操作中完成可能很显眼。取而代之的是，通过移动节点服务器边界，并每次只迁移世界区段的一部分，虽然事实上在边界移动的时候大量代理对象将会建立和删除，但是这样的传输更为平滑。

有了动态边界，远程控制器也可以作为世界区段的界面，让管理员有能力控制它们确切的布局和大小，可以具体地管理世界的划分。没有了静态划分的限制，管理员就可以在计划游戏的哪个地方会出现大量玩家之前，就修改世界区段的大小。

不在节点服务器上直接计算 AI，它可以在专用的服务器上完成。而节点服务器会把 AI 看做一个普通的客户端，让它们通过专用的端口连接（这个端口用来做服务器之间的通信）使我们可能作进一步的优化，因为它们可以看做是“可信任”的客户端。一些与世界交互的双重检查或 AI 活动的验证可以忽略。有些人可能想知道 AI 服务器是否得通过其他端口来连接，而不是与外部世界通信的端口。（在一个实际的服务器中，这完全可以考虑使用另一个网卡）只有在连接的时候，当服务器要开始管理 AI 实体的时候，才会设置“可信任实体”标记。

有关后续实现内容可在[Gizz04]网站上找到。

6.2.9 IOCP

虽然本文讨论了 IOCP，但是请记住那是 Windows 特定的。IOCP 概念之间的不同之处和它的*nix 同类“dev/poll”，可参考 Ian Baril 的文章[Baril04]。

6.2.10 参考文献

[Aronson97] Aronson, Jesse. “Dead Reckoning: Latency Hiding for Networked Games.” Available online at *http://www.gamasutra.com/features/19970919/aronson_01.htm*. September 19,

1997.

[Beardsley03] Beardsley, Jason. "Seamless Servers: The Case For and Against." *Massively Multiplayer Game Development*. Charles River Media, 2003.

[Baril04] Baril, Ian. "I/O Multiplexing & Scalable Socket Servers." *Dr. Dobb's Journal* (February 2004): pp. 42–45.

[Gamma95] Gamma, Erich, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gizz04] *http://www.gizz-moo.com/*.

[Jones02] Jones, Anthony and Jim Ohlund. *Network Programming for Microsoft Windows*, Second Edition. Microsoft Press, 2002.

[mySQL] *http://dev.mysql.com/*.

[Oracle] *http://www.oracle.com/*.

[SQLServer] *http://www.microsoft.com/sql/*.

6.3 设计一个脏话过滤系统

Shekhar Dhupelia
sdhupelia@gmail.com

随着在线用户名和聊天室在最近的在线游戏中越来越普遍，出现了很多成人内容的问题（特别是当发布需要评级（E-rated）的游戏时！）。这个时候，每个人都承认不可能抓住所有可能使用的脏话，同时还保持高效和对用户友好。但是，我们有办法抓住大量比较显而易见的情况。

本文讨论了一个快速的单词搜索脏话过滤器的组件，包括如何在数据源中识别出“脏话”，如何搜索所有这些数据，以及多种用来应对脏话的选择。本文还讨论了在建立这样一个系统的时候，一些最佳的实践和额外的参考点。

6.3.1 语法与内容

要记住大部分游戏无法负担处理“内容”的问题，这一点很重要。内容代表着一句话或者一个完整想法的真实意思。虽然可以完全阻拦一些标准的句子，但是确定一个句子的意思，并拒绝违反潜在规则可能需要非常高级的人工智能；这是千真万确的，因为 AI 需要实时运行。

此外，对音频聊天的讨论远远超出了本文的范围。虽然在游戏中提供一个“家庭般友好”的环境是令人钦佩的，特别是一个适合于所有年龄层的游戏，但是监视音频聊天和对它做任何特别有用的事情远远超过了我们平均的游戏硬件范围。

这里的讨论是关于语法。语法涉及它们本身的单词；虽然“^#$% YOU!”仍然代表着一些意思，但主要目的还是阻止完整的咒骂语互相传递。

6.3.2 字典

一个字典，或数据字典，是已存脏话的查找表。这里的选择是无止境的；应用程序可能已经有一个适当的数据库系统或文件系统，可能使用一些类型的关系或记号查找。目标是有一个一致的、容易管理的数据源，可以在一个地方存储所有指明的“脏话”。

如果从头构建，一个好的出发点是 XML 文件。对于非从头开始的，XML 是信息鉴别、把标记赋给数据的一种形式，可以快速地搜索和引用。XML 的一个很好的参考可以在 O'Reilly Network 的 XML.com 找到[Oreilly01]。

6.3.3 解析器

把脏话存储在一个字典里只是一个开始。更消耗资源的部分是搜索这个数据字典，来匹配一个字符串。大部分 C/C++程序员和 Java 程序员都知道字符串的操作、解析会很慢，如果不仔细实现就会成为游戏的瓶颈。

而且，这个子系统可能被放入很多现有的游戏引擎中。解析数据源在游戏的其他很多领域也非常有用，而不仅限于这个用途。然而，当用 XML 建立一个新的引擎的时候，Xerces 是一个好的起点[Xerces01]。Xerces 库是一个开放源代码的 C++库，允许非常快地搜索 XML 文件源。

6.3.4 过滤

至此，游戏将有一个脏话和字符串的查找字典，以及一个解析器来积极地遍历和剔除任何违反字典的东西。下一步是过滤文字，阻止它或相应地改变它。但是，在这个游戏中使用的方法取决于本身特殊的设计，因为有多种不同的方式可以过滤脏话字符串。

1. 搜索和替换，预确定的字符串

第一种过滤方法是在字符串上做一个查找和替换的操作，使用预确定的字符串。这需要建立第二个单词或单词表作为适当的替代品。预确定的字符串例子如下：

```
char szReplacementText[] = "[censored]"
```

在这里，脏话会被替换成`[censored]`。为了在可能有很多脏话的地方增加变化，你可能要遍历一个 10 或 20 个变量的列表，比如`banned`、`blocked`和`not allowed`。

2. 搜索和替换，随机字符串

这个方法和前一个类似，只不过它使用随机的字符来提供变体。例如，英文的连环漫画经常用一些&^*$之类的字符来指代一句脏话。然而，并没有设置使用符号数量的标准，甚至没有要画出哪些特定符号的标准。要记住，通过选择随机数量的字母可以表示更多的变体，或者随机地从一组字母或替代单词中选择。这看起来可能像下面的过程。

（1）定义一个替代字母的列表。
（2）获得脏话字符串。
（3）选择一个随机数量的字符。
（4）一个一个字符地循环这个脏话字符串。
（5）从列表中选择随机的替代字符。
（6）返回过滤的字符串。

3. 单词剥离

单词剥离的用处在于它有助于从一块文字中删除一些意思。有时候，一个句子如`I want to %#@$@%`，可能仍然在会话或语境的内容中有一些意思。但是，单词剥离可能会把这个句

子改成 I want to，它只是完全去掉了脏话字符串，没有替代品。当然，这不能阻止用户简单地输入适当的替换字符的字符串。这是有必要进行人工干涉的地方，一个后面要讨论的主题。

4. 阻止和拒绝

这是非常暴力的方法，但是要尽可能地尝试去阻止脏话内容，而不只是脏话的语法。阻止和拒绝会检查脏话字符串，然后拒绝整个提交。比如在聊天室的构建中，输入句子 word word2 word3 violator 的用户不会发送出 word word2 word3 或 word word2 word3 %#$^。取而代之的是，他们将得到一个错误消息，提示有脏话，整个字符串都会被抛弃。

6.3.5 最好的过滤实践

虽然各种过滤方法可以高效地实现，不管是单独的或者合起来，但是也有许多关键的经验可以帮助这个过程更快而且更少地打扰最终用户，同时对游戏服务器保持高效。这些内容将在下面描述。

1. 使用离线和在线字典

虽然对一些嵌入式系统（比如缺乏大存储空间的游戏控制台）不太适合，但是一个好主意是使用离线字典，外加服务端的字典文件。这允许一些检查的执行做到快速和局部，不用花费额外带宽来验证每个字符串。在线字典可能作为第二种形式的保护。例如，一个静态的离线字典可以首先对每个字符串做一个快速检查，而动态的、可更新的在线字典可以在字符串传输的时候做另一遍检查。

2. 先离线过滤

如同前面提到的，不要把大量带宽不必要地浪费在任何不是核心游戏的部分中。先进行离线过滤可以节省很多时间和开销，而且确实让一些过程对用户更友好。其中的一个例子是当用户注册的时候，如果玩家输入了天神之类的词语或其他字符串，可以用离线字典做一个快速的检查。这至少能让用户完成输入过程，然后才在线处理。在线部分可能只要再次检查可疑的字符串，而不需要检查整张表。

虽然不可能适用于所有的系统，但一个动态更新的字典文件也是个强大的工具。随着玩家产生新的脏话、使用原先没注意到的脏话或可疑的行话，字典应该定期用这些增加的内容来更新。甚至在离线字典不能修改的情况下，在线字典也应该定期检查。

3. 字典更改控制

虽然已经确定了定期字典更新很重要，但是也要小心控制这些改变，并跟踪它们，类似于源代码改变或美工的更改。更新应该安排在有规律的时间间隔（有必要的话），跟踪和修订类似于源代码，它的完成应该对用户影响很小（如果不是实时，那就作为其他更新的一部分，比如漏洞修正或内容改变）。

4. 持续监控任何绕开的情况

当过滤是基于字典的时候，总会有单词可以骗过设立的系统，但仍然会传达出一些违法

PDG

或生气的语调。经常上网的用户习惯于用一些符号和数字写单词，来代替标准的字符。举例来说，假设字典中有单词“FOOL”。现在，如果用户输入的是“F00L”或“F@@L”，意思仍然很清楚，但是这些单词逃过了系统的注意。

如果不涉及对这些单词的修改，那整个脏话过滤系统就像是没有用的。不幸的是，没有自动的方法可以高效地从字典中处理所有这些可能的单词变化。在这里确实需要持续的、警惕的人工干涉。

6.3.6 人工干涉

在使用所有这些技术之后，仍然会怀疑这些策略不能完全提供“安全”、家庭般友好的环境。事实上，任何一个或多个组合在一起的、满足 CPU/内存限制的过滤方法都能很容易地被绕开。

不幸的是，能期望的最好情况是从技术的观点做出合理的努力，并希望它覆盖绝大部分潜在的违反者。但是这项技术在很多情况下会失败。如果游戏仍然承受了大量不顾这些保卫措施的谩骂，通常就没有别的办法了，只能增加一层人工干涉。有两种常用的方法可以帮助铲除大量情况最糟的用户：游戏中的管理和玩家反馈。

1. 游戏中的管理

游戏中的管理是指一个真正的监督员或一队监督员，他们积极地在在线游戏中巡逻，简要地察看任何种类的犯规。这些监督员可能被选列为管理员，或者“秘密行动”。虽然坏行为的结果可能包括不允许以后继续游戏，但是一个宽松的惩罚通常就能解决这个问题（比如，30 天不允许聊天）。

2. 玩家反馈

如果得到正确的处理，那么玩家反馈可以很强大。有了这项技术，其他的玩家就可以报告违规的行为。这就允许社区自己成为活跃的警察，并且和一队监督员相比，他们可能会铲除更多不好的行为。虽然这仍然需要有人去读反馈并相应地行动，同时处理更多的欺诈反馈，但这也允许把较为活跃的报告者变成社区的“高级”成员，并可以作为游戏最坚定的支持者。

6.3.7 总结

本文讨论了一些不同的方法来实现在线游戏中的脏话过滤。另外，本文也讲到了有些时候过滤可能会——而且必然会——失败，以及人工干涉成为的必要性。幸运的是，这里讨论的方法将严格地减弱恶意玩家之风，直到用户不能继续他的坏行为为止。通过组合这些方法，就可以保证用户是在安全的、受保护的游戏环境中。

6.3.8 参考文献

[Fepproject01] The Free Expression Policy Project. “Fact Sheet on Internet Filtering.”

Available online at *http://www.fepproject.org/factsheets/filtering.html*.

[IGN01] IGN Xbox. "Xbox Live Etiquette." Available online at *http://xbox.ign.com/articles/377/377569p1.html*.

[Oreilly01] O'Reilly Network. "XML.com: XML from the Inside Out." Available online at *http://www.xml.com/*.

[Xerces01] "Xerces C++ Parser." Available online at *http://xml.apache.org/xerces-c/*.

6.4 远程过程调用系统的快速和高效实现

Hyun-jik Bae
imays@hitel.net

有很多关于网络游戏开发的文章专注于航位推算法（dead reckoning）、分布式游戏服务器、节流（throttling）、负载平衡、巧妙的种子等方面。那些都是非常有用而且非常重要的想法，可以应用到很多项目中。所有这些技术都共享了一个常见的特性，它们都需要发送和接收至少一条消息；这段代码经常是冗余网络代码的最大源头之一。

很多网络游戏程序员都试图编写尽量少的代码，轻松统一地对付网络消息。发送和接收消息涉及很多 switch-case 语句和结构体定义。这样的冗余只会随着开发过程中消息的增加或改变而增长。

本文介绍了一个解决方案来减少这个负担，专注于代码级的观点。

首先，让我们看看大部分网络游戏的一个典型问题。程序清单 6.4.1 演示了一个例子，处理两个消息："move knight" 和 "attack enemy"。

程序清单 6.4.1　我们熟悉的网络代码

```
/////// 在两边
#define Message_Knight_Move_ID 12
#define Message_Knight_Attack_ID 13

struct Message
{
    int m_msgID;
};

struct Message_Knight_Move:public Message
{
    int m_id;
    float m_x,m_y,m_z;
};

struct Message_Knight_Attack:public Message
{
    int m_id;
    int m_target;
    int m_damage;
};

/////// 在发送方
```

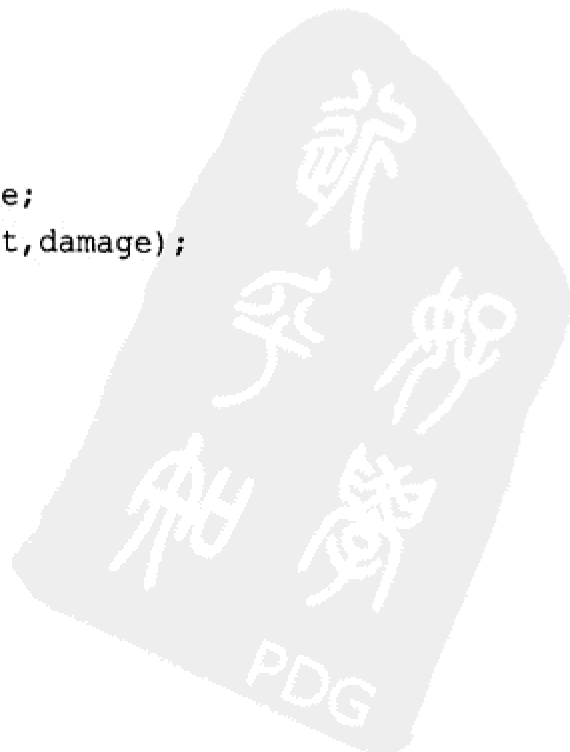
```
void Knight_Move(int id,float x,float y,float z)
{
    Message_Knight_Move msg;
    msg.m_msgID=Message_Knight_Move_ID;
    msg.m_id=id;
    msg.m_x=x;
    msg.m_y=y;
    msg.m_z=z;
}

void Knight_Attack(int id,int target,int damage)
{
    Message_Knight_Attack msg;
    msg.m_msgID=Message_Knight_Attack_ID;
    msg.m_id=id;
    msg.m_target=target;
    msg.m_damage=damage;
}

void DoReceivedMessage(Message* msg)
{
    switch(msg->m_msgID)
    {
    case Message_Knight_Move_ID:
        {
            Message_Knight_Move* msg2=
                (Message_Knight_Move*)msg;
        }
        Do_Knight_Move(
            msg2->m_id,
            msg2->m_x,
            msg2->m_y,
            msg2->m_z);
        break;
        // ... 其他消息类型的情况
        case Message_Knight_Attack_ID:
        {
            Message_Knight_Attack* msg2=
                (Message_Knight_Attack*)msg;
        }
        Do_Knight_Attack(
            msg2->m_id,
            msg2->m_target,
            msg2->m_damage);
        break;
        // ... 其他消息类型的情况
    }
}
```

假设有一种情况，我们必须增加新的消息类型。为了这个新消息，我们必须修改程序清单 6.4.1 的 4 个地方：一个新的消息 ID，一个新的消息结构体，一个新的函数用来发送消息，

以及一个新的 case 用来读取这个消息。这个麻烦的工作可以通过使用流化（streaming）的类来简化，比如 Carchive 或 std::istream，或者你自己写的类。有了这些类，程序清单 6.4.1 可以简化成 6.4.2 的样子。

程序清单 6.4.2　我们熟悉的网络代码的另一种风格

```
/////// 在两边
#define Message_Knight_Move_ID 12
#define Message_Knight_Attack_ID 13

/////// 在发送方

void Knight_Move(int id,float x,float y,float z)
{
    Message msg;
    msg<<(int)Message_Knight_Move_ID;
    msg<<id<<x<<y<<z;
}

void Knight_Attack(int id,int target,int damage)
{
    Message msg;
    msg<<(int)Message_Knight_Attack_ID;
    msg<<id<<target<<damage;
}

////// 在接收方
void DoReceivedMessage(Message* msg)
{
    int msgID;
    (*msg)>>msgID;
    switch(msgID)
    {
    case Message_Knight_Move_ID:
        {
            int id;
            float x,y,z;
            (*msg)>>id>>x>>y>>z;
            Do_Knight_Move(id,x,y,z);
        }
        break;
    case Message_Knight_Attack_ID:
        {
            int id,target,damage;
            (*msg)>>id>>target>>damage;
            Do_Knight_Attack(id,target,damage);
            break;
            // ... 其他消息类型的情况
        }
    }
}
```



不管哪种方式，都可以通过在发送方调用 Knight_Move()，在接收方调用 Do_Knight_Move() 来发送、接收和处理消息，还有 Knight_Attack() 和 Do_Knight_attack() 消息。

注意这些例子中的模式，它们在模式中非常冗余。它们现在可以抽象到只作为一行的声明，就像程序清单 6.4.3，目标是让真实的游戏代码就是与此类似。

程序清单 6.4.3　我们期望的最终风格

```
Knight_Move(int id,float x,float y,float z);
Knight_Attack(int id,int target,int damage);
Listing 6.4.3. The ultimate style we desire
```

我们可以把程序清单 6.4.1 和 6.4.2 称为手工的发送-接收-switch-case 代码，把程序清单 6.4.3 称为自动发送—接收-*switch-case* 代码。接下去，将讨论如何自动地生成这样的发送-接收-*switch-case* 代码，这是远程过程调用（*RPC*）的主要优点之一。

6.4.1　RPC：简介

RPC 是一个消息传递工具，它独立于内在的网络层，并允许分布式应用程序调用网络上不同机器所提供的服务。简而言之，RPC 抽象了一台机器上的一个程序请求调用另一台机器上的一个函数的技术。有关通用 RPC 使用的完整介绍，请参考操作系统的课本 [Sillberschatz02]。

有很多 RPC 的实现，有些由几个大的公司提供。最出名的是 MS-RPC，DEC-RPC，DCOM，CORBA 和 Java RMI。（DOM，CORBA 和 Java RMI 有面向对象的行为，但它们的概念也是基于 RPC。）本文这些实现叫做传统的 RPC 系统。传统的 RPC 系统是稳定的，而且支持很多特性，比如安全、认证和很多协议兼容性。然而，因为这些实现中有许多对游戏编程来说都有一些缺点，所以我们要写自己的 RPC 系统，并避免传统 RPC 的下列问题：

- 难以理解和使用
- 对游戏应用来说过于冗余和麻烦
- 不允许完全控制信息的格式
- 不允许完全控制信息的传输，比如带有节流功能
- 在实际应用中，传统的异步模型比同步模型复杂得多

本文将介绍一个实现用于游戏编程的 RPC 系统的方针，由一个 *RPC* 编译器和一个运行引擎组成。因为这个 RPC 系统完全在应用程序的控制下，所以它可以根据需要进行优化或流水化。

这里呈现的例子非常简单，而且对于快速开始来说已经足够快了。为了进一步简化，将有意忽略一些底层的东西，比如实际的套接字代码和这个 RPC 系统的错误恢复，因为增加越多的特性，这个例子就会变得越复杂。

我们把发出 RPC 调用的模块称为 *RPC* 客户端，将接收这些 RPC 的模块称为 *RPC* 服务端。注意 RPC 客户端和服务端与游戏的客户端和服务端不同。一般来说，游戏服务端和客户端在程序模块中都有两个异类的 RPC，用来调用客户到服务和服务到客户，而对等（peer-to-peer）的游戏有一个同类的 RPC，相互地调用和接收。这里包含的例子 *RPCDuel* 是

后面的一种情况。

RPC 有两种网络模型：异步和同步。同步 RPC 有输出参数。如果 RPC 客户端程序发送一个同步 RPC 到 RPC 服务端，PRC 客户端将等待，直到 RPC 服务端完成执行并接收到了返回值。另一方面，异步 RPC 在两个方面有所不同：

- 异步 RPC 不等待返回调用
- 异步 RPC 不能有输出参数
- 异步 RPC 允许不可靠的消息。因为一些消息可能在传输中丢失，不可靠消息的 RPC 也可能丢失。

游戏程序极少需要同步 RPC；大部分 RPC 服务端程序在设计上都没有等待 PRC 客户端执行的情况，因为这样可能造成瓶颈或死锁。取而代之的是，它们会定义两个消息，请求 RPC 服务端并从它回复。此外，RPC 客户端程序很少等待 RPC 服务端的执行。很多游戏必须等待来自 RPC 服务端的回复，显示一个等待动画并允许用户按取消按钮。另外，同步 RPC 很难实现，我们将在后面解释理由。

本文的样例代码包含一个编译器和解析器。关于语法定义、解析器和词法分析，有一些更详尽的阅读资料[Sebesta02]。

6.4.2 RPC：设计

这里简单地表示了一个通用 RPC 实现的常见序列设计。简单来说，它工作起来如图 6.4.1 所示。

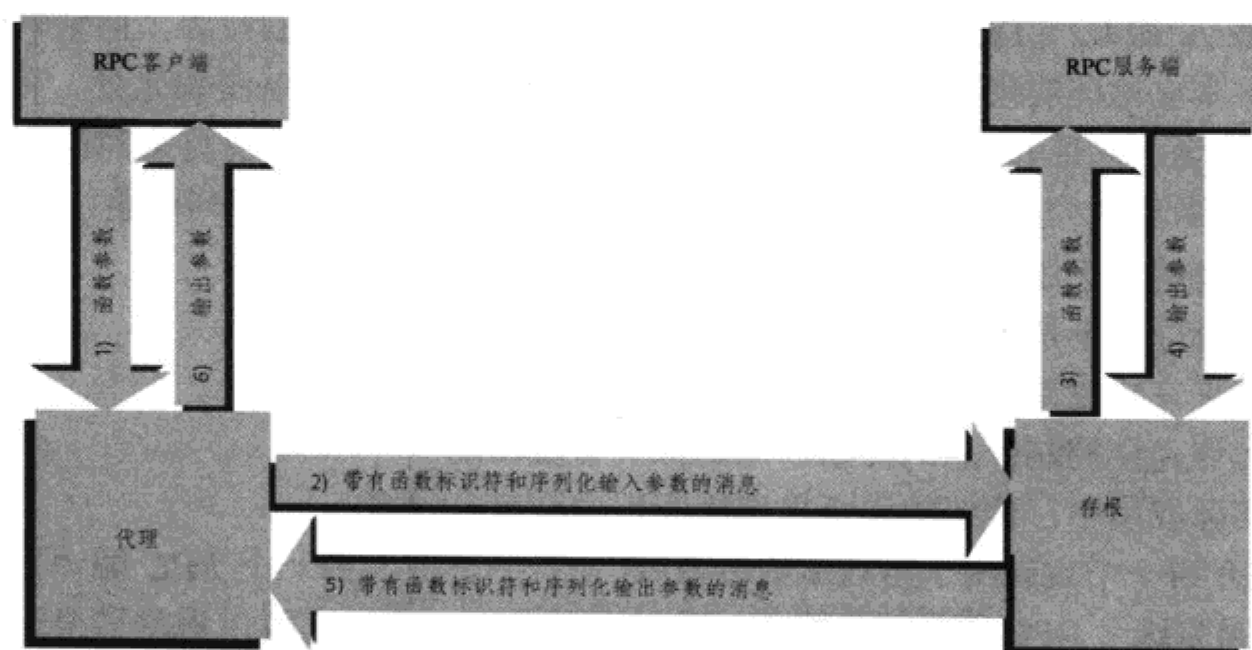


图 6.4.1 有回复的 RPC 序列图

因为我们只对异步模型感兴趣，所以不需要等待回复的阶段。整理之后，它工作起来如图 6.4.2 所示。

当一个 RPC 函数从 RPC 客户端调用的时候，当然不能直接在 RPC 服务端执行函数体。取而代之的是，它会汇集函数参数，并把这些参数增加到消息中，以及一个消息头，来鉴别需要的函数调用。执行这个阶段的代码叫做代理，因为它执行同样名字的另一个函数，它会把函数标识和参数序列化（集合（*marshal*））到一个数据流中，并发送一个消息，而不是调

用真实的函数。

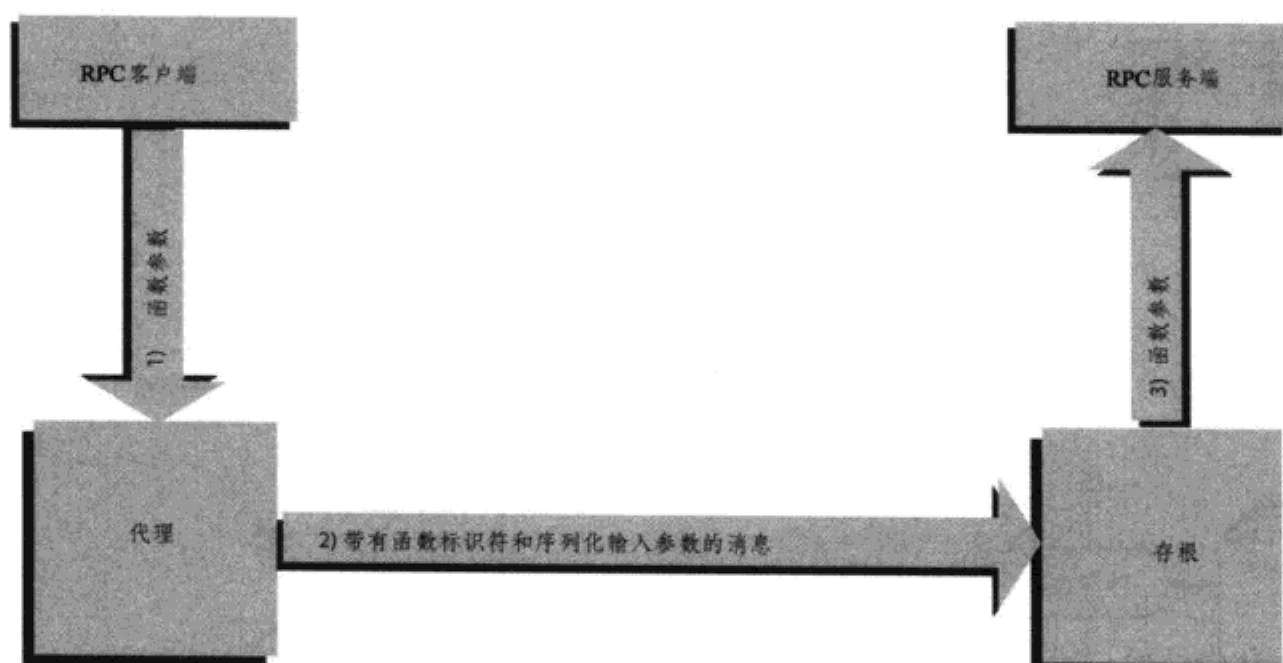


图 6.4.2　没有回复的 RPC 序列图

当 RPC 服务端收到一条消息的时候，它会把这条消息传给一个函数，那个函数会读取消息，确定哪个 RPC 体应该接管控制，把消息数据析取到参数中，然后调用一个 RPC 函数体。这叫做存根（stub），因为它的职能是选择一个或多个函数，并做出适当的调用。注意在这里只讨论异步 RPC，所以可以安全地忽略 RPC 的响应阶段。

因为一个 RPC 调用会被转换成一条消息并在另一端解析，所以每个 RPC 消息应该以特殊的方式形成。当在代理端调用一个 RPC 的时候，产生的消息必须包含所有关于函数调用需要的信息。这包括：

- 函数标识符
- 函数参数的序列

函数标识符是一个预定义的数字，它的值在声明另一个 RPC 函数的时候会增加。这对于检测消息序列化的位置是必要的。

函数参数的序列基本上是把每个参数一个接一个地连接成数据块，由代理建立。从现在开始忽略错误检测和恢复，只需要参数值本身，因为在看函数标识符的时候可以推出有多少个参数，以及它们是什么类型的。

有几种方式来序列化可变数据类型的参数。本文呈现的例子演示了一个消息类和几个序列化函数，它们的函数名相同，参数类型不同（函数重载），比如下面的例子：

```
void Message_Read(CMessage& m,double &val);
void Message_Read(CMessage& m,std::string &val);
void Message_Write(CMessage& m,const double &val);
void Message_Write(CMessage& m,const std::string &val);
```

图 6.4.3 演示了开发者如何在他们自己的网络应用程序中采用 RPC。

我们要用简单的单行声明来产生所有的网络代码。RPC 引入了接口定义语言（*Interface Definition Language*，简称 IDL）来进行类型定义。IDL 文件是我们写入 RPC 函数定义，以及这些单行声明的地方。记住 IDL 文件不能直接被应用程序源文件识别。取而代之的是，IDL 文件应该通过 IDL 编译器编译，它产生了代理和存根代码文件。代理代码然后连接到 RPC

函数远程调用的地方；同时，存根代码连接到它们被调用的地方。

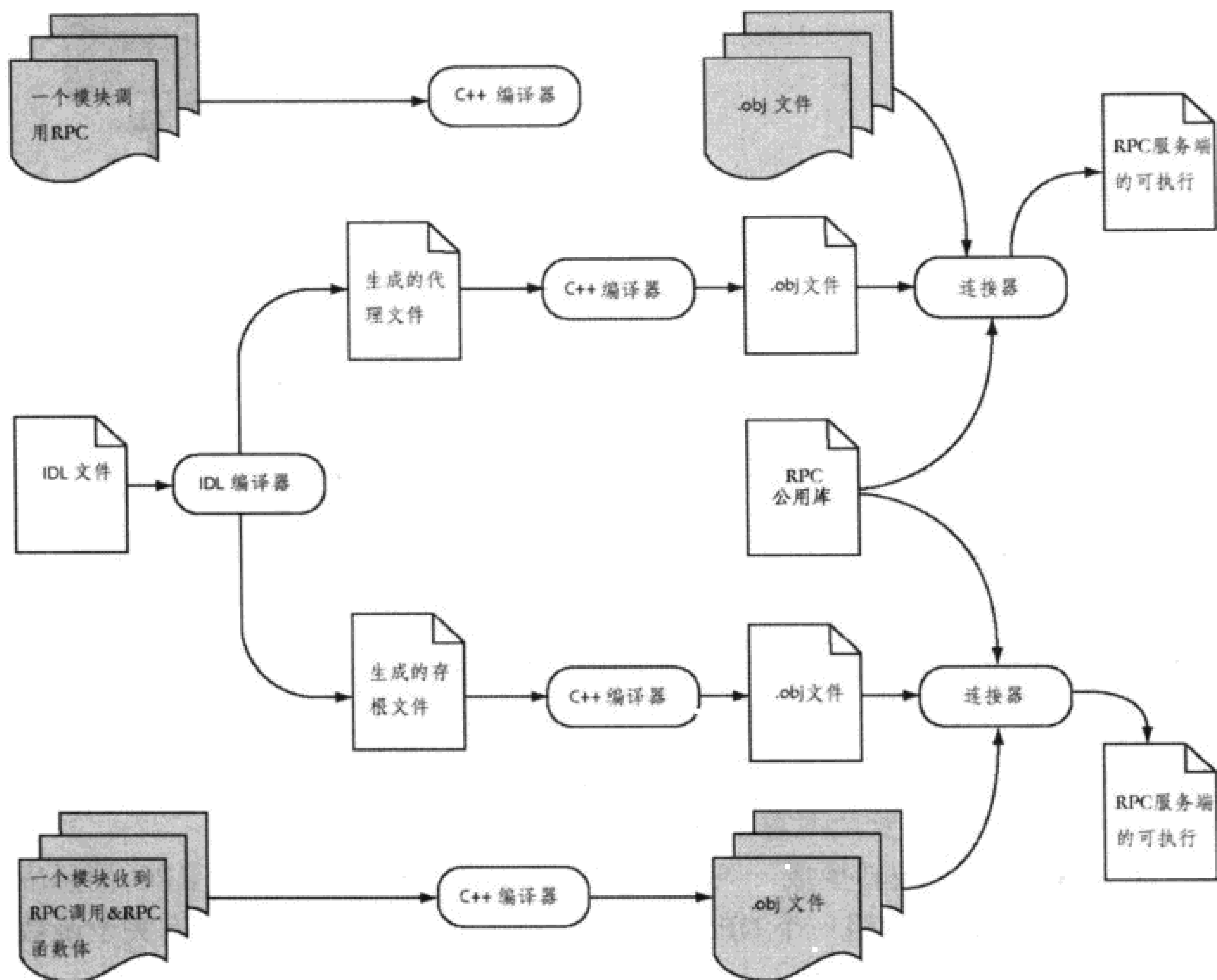


图 6.4.3 设计期序列图

因为至少存在一个用于存根和代理的帮助函数或类，可能要把它们移到一个库中，我们把它叫做公用运行库。

但是，不要把产生的代理和存根代码包含到源代码控制系统中。因为 IDL 编译器在 IDL 文件被修改的时候会覆盖代理和存根代码，而代理和存根代码还没有检出（check out）的时候是只读的，所以这可能会造成编译错误。

6.4.3 RPC：实现

下面一节是详细的 RPC 实现。

1. 代理和存根

一个 RPC 函数不能直接调用，因为它的真正函数体在另一个进程中。然而，应该存在一个有相同原型的本地函数。它把这个调用替换成建立一个消息并把它发送给远程进程，即真正的函数体存在的地方。这些远程调用的本地版本就是被我们称为代理代码（*proxy code*）的东西。

IDL 编译器负责产生代理代码。IDL 编译器使用一个指定的代码模式来产生代理代码。定义这个规范的一种简单方式是写一个代理代码的例子，从中推出代码模式，并在 IDL 编译

器中使用这个模型。

当收到一个消息的时候，它应该传给一个有大量 switch-case 的函数。首先，通过读取消息头中的函数标识符来识别是哪个请求。然后，到达一个 switch-case。每个 switch-case 会把消息解序列化（反集合（unmarshal））成每个参数值。接着，调用 RPC 函数体。存根代码由一个里面有很多 switch-case 的函数组成。函数体在存根代码之外。IDL 编译器也可以负责产生存根代码。像代理代码模式一样，我们会把存根代码模型提供给 IDL 编译器。

2. IDL 编译器

为了开始 IDL 编译器，必须做一个编译器，把 IDL 文件作为输入，然后产生存根和代理的输出代码。下面从解析器开始，它会分析这个 IDL 文件声明了哪些 RPC 函数。我们在这里的实现中选择了 ANTLR [Parr04]，它比 Lex 和 Yacc 更为广泛和容易使用。ANTLR 是一个用 Java 写的自顶向下分析器生成器，但是也可以用它生成 C++的分析器代码。

为了简化，我们将专注于 IDL 文件的基本语法。看传统 RPC 系统的 IDL 文件的时候，会发现它们有很多参数，比如网络协议、参数传递方向以及惟一的 ID。现在，看一个简单的例子：

```
Knight_Move(int id,float x,float y,float z);
Knight_Attack(int id,int target,int damage);
```

有了这个模型，程序清单 6.4.4 提供了一个 ANTLR 格式完整的语法例子。

程序清单 6.4.4　IDL 语法的一个片断

```
// 一个 RPC 函数定义
functionDefinition :
    IDENT // 函数名
    LPAREN
    ( parameterDefinition
        ( COMMA! param=parameterDefinition  )*
    )?
    RPAREN
    SEMI
    ;
// 一个 RPC 参数
parameterDefinition :
    IDENT // type
    IDENT // name
    ;
```

第一个函数标识数字可以在任何地方定义，甚至是在 IDL 文件之外，但是一般来说，把定义放在 IDL 文件中会减少疑惑。

编译器的后端应该根据来自前端的语义信息（由解析器发现的）生成这些文件。总体而言，一个 IDL 输出由几个源代码文件组成：

- 代理源代码和它的头文件
- 存根源代码和它的头文件

回到前面的程序清单 6.4.1 和 6.4.2，可以看到这些代码模式关联到程序清单 6.4.4，其中

FunctionName 表示一个 RPC 函数名，ParameterDeclarations 表示 C++声明中 RPC 函数所有的函数参数，Parameters 表示由逗号隔开的参数列表。参见程序清单 6.4.5。

程序清单 6.4.5　我们的代理和存根代码模式

```
// 公用
static const RPCHeader RPC_ID_FunctionName=(10+0);
FunctionIdentifierDefinitionsForOtherFunctions

// 代理
RPCResult FunctionName(RPCSendTo sendTo,RPCSendContext
                       sendContext,ParameterDeclarations)
{
    CMessage m;
    Message_Write(m,RPC_ID_FunctionName);
    Message_Write(m,FirstParameter);
    ...
    Message_Write(m,LastParameter);
    return RPC_Send(sendTo,sendContext,m);
}

ProxyDefinitionsForOtherFunctions

// 存根
RPCResult RPC_DoStub( RPCSendTo recvFrom,RPCSendContext
                      recvContext,CMessage& m)
{
    RPCHeader msgID;
    m.SetCursor(0);
    Message_Read(m,msgID);
    switch(msgID)
    {
    case RPC_ID_FunctionName:
        {
            FirstParameterDefinition;
            Message_Read(m,FirstParameter);
            ...
            LastParameterDefinition;
            Message_Read(m,LastParameter);
            FunctionName(
                recvFrom,
                recvContext,
                Parameters);
        }
        break;
    CasesForOtherFunctions
    }
}
```

注意 PRCHeader，RPCSendTo 和 PRCSendContext 只是本地类型定义。在这些例子中，它们定义成了 unsigned int。

为了简化，这里描述的后端代码使用 `printf()` 或 `cout`，以及多个 `for {}` 语句。然而，当代码模式变得复杂的时候，这会造成麻烦。所以采用一个基于文本模板的文本生成器会很有用，比如 eNITL [Breck99]。

随着把更多的功能增加到 RPC 系统中，维持和修改 IDL 编译器可能会越来越难。可以把它们的一部分加入通用库，库中包含了一些 `typedef`、帮助函数和一些类。本文的例子有一个非常小的通用库，叫做 `MyRPC`。`MyRPC` 有一个类 `CMessage` 和几个读取和写入函数，用于使用最频繁的基本数据类型，比如 `int`、`float` 和 `std::string`。

6.4.4 RPC：使用

要在应用程序中使用这个 RPC 实现，应该把 IDL 文件添加到项目中，并为这个 IDL 文件配置自定义建立选项。在 Visual Studio.NET 的例子中，可以按图 6.4.4 所示配置。

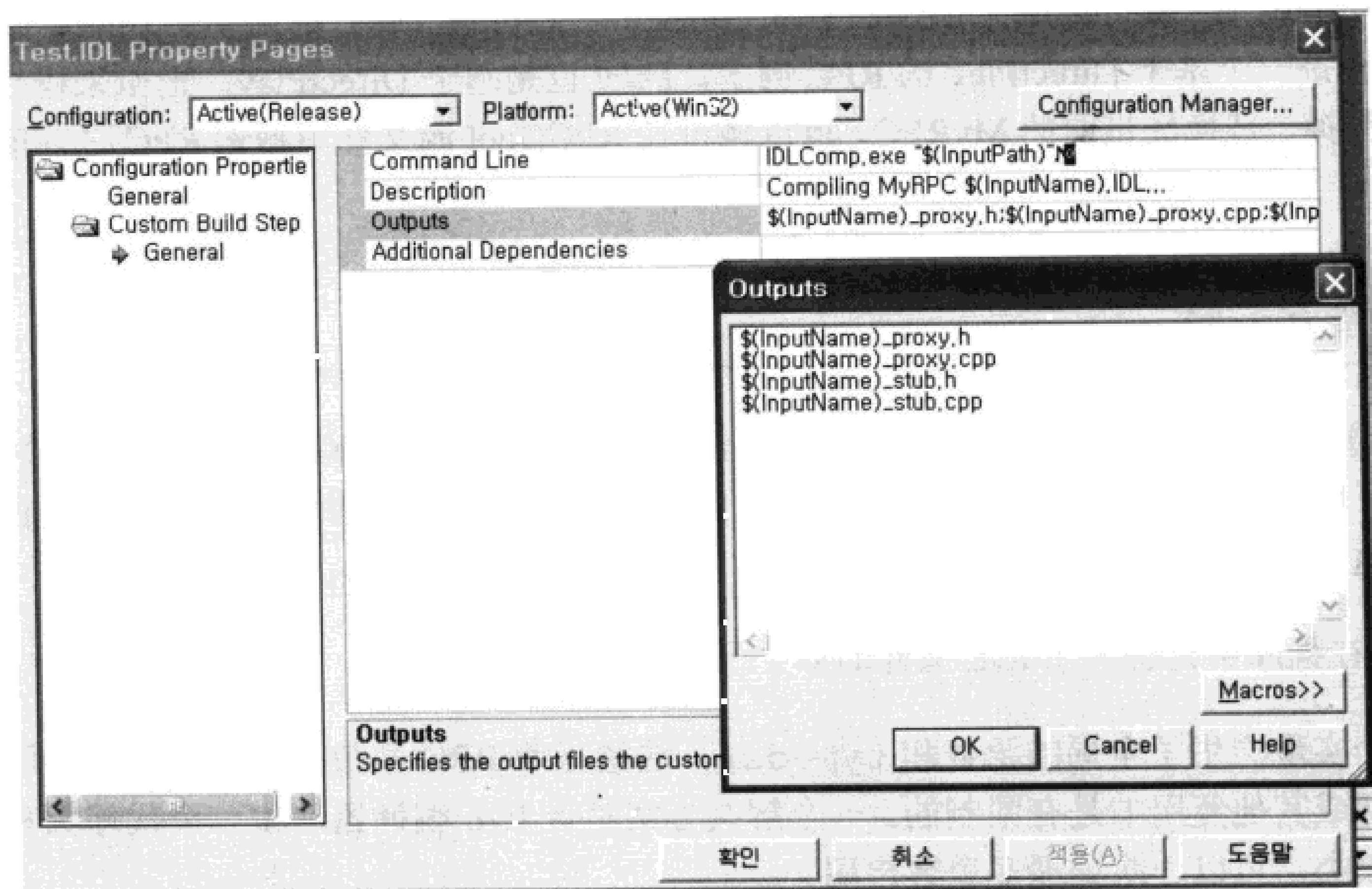


图 6.4.4　配置 Test.IDL

另外，编译的输出文件和通用库需要添加到我们的项目配置中。因为生成的代理和存根代码是一种类型的编译输出，所以推荐用 `#include` 语句把产生的代理和存根.h 和.cpp 文件间接地添加到源代码中。

`RPCHeader`、`RPCSendTo` 和 `RPCSendContext` 用来通过网络模块发送和接收任务。如果 RPC 函数定义的数目小于 255，一个减少流量的技巧是把 `RPCHeader` 改成 `unsigned char`。`RPCSendTo` 标识了接受这条消息的远程计算机。RPCSendContext 定义了其他参数，比如协议选择（TCP 和 UDP）、超时选项或用于多个异步响应的调用标识。可以把这些 `typedef` 改成你喜欢的。RPC 实现会自动把 `RPCSendTo` 和 `RPCSendContext` 添加到产生的 RPC 函数的声明中，分别作为第一和第二个参数。

然后需要向应用程序中添加一个网络模块，因为 RPC 系统本身没有这样的东西。当代理构造一条消息的时候，它会调用 `RPC_Send()` 来把它传给网络模块。同时，当网络模块收到一条消息的时候，它应该通过 `RPC_Stub()` 把消息传给存在于产生的存根文件中的那个存根。

6.4.5 样例程序

本文带有两个可执行的例子：*SimpleTest* 和 *RPCDuel*。SimpleTest 基本上是一个 Hello World 程序。它有一个 IDL 文件和一个 `main()` 函数，它会调用一些 RPC 代理函数，只是把产生的消息存储在本地内存中。然后假设它已经收到了代理发送的消息，就调用一个存根函数，解析消息并调用相应的 RPC 体。虽然微不足道，但是它应该作为一个入门的例子，来演示一个更复杂的 RPC 例子是如何工作的。

RPCDuel 是 Microsoft DirectX SDK 样例中一个叫做 *Duel* 的简单游戏的修改版本。RPCDuel 是一个基于 DirectPlay 的 RPC 例子，*Duel* 也是基于 DirectPlay。把原来程序的消息部分注释掉，替换成当前的 MyRPC。可以通过在 RPCDuel 源文件中搜索 RPC 之前的注释来检验 RPC 是如何工作的。

6.4.6 更多特性

有句谚语叫“千里之行，始于足下。”虽然这里的实现，作为足下的第一步，写起来很简单，而且只是开始演示 RPC 的实际使用，但是这个系统可以很容易地根据需要进行改进和扩展。这里是一些值得研究的改进方面。

1. 在程序中允许一个 IDL 文件的多个代理/存根实例

这个实现产生了全局的函数和 `typedef`。这意味着 RPC 函数的多个实例，在用于多人游戏会话或其他复用中是有限制的。一个解决方案是让 IDL 编译器产生一个代理类和一个抽象的存根类，可以一次处理多个客户端。

2. 错误检测和恢复

大部分在线游戏服务端或对等网的端点（peer）必须容忍一定的错误消息，这可能会造成各种异常，比如缓冲区溢出、无效的消息格式和网络错误。解决方案之一是在序列化和解序列化的函数中增加边界检测。当然，如果让 IDL 语法可以接受由游戏程序员指定的值域，就可以减少错误检测代码。

3. 更多的数据类型

在我们的例子中，只有一些基本数据类型，比如整型、浮点数和有序列化和解序列化函数的字符串。但是，我们可能要在 RPC 函数中支持更多的数据类型，甚至是数组和结构体。SimpleTest 例子中有一个可序列化的类 `Vector3D`。它有两个函数。

```
void Message_Write(CMessage& m,const Vector3D &val);
void Message_Read(CMessage& m,Vector3D &val);
```

可以根据需要继续定义更多的序列化和解序列化函数。然而，一些 C++模板函数可能对通用结构体更为方便，比如 array 和 list。

4. 消息加密

在很多在线游戏中，不是所有的消息都需要高度安全的加密，而且因为性能问题，只有一些类型的通信消息是安全的。

当把一个加密选项添加到 IDL 语法、IDL 输出代码模式和流化类的加密方法时，只需要一个这样的定义就可以使用加密：

```
[encrypted] RequestLogin(string id,string password);
```

5. 消息压缩

消息压缩通过增强流化类和 IDL 语法就可以简单实现。例如，我们可能要为每个参数增加一个“位的数量”选项，可以这样定义：

```
Knight_Attack(int type:4,int magicBuff:7,int critical:1);
```

变量 type 占有 4 位，magicBuff 占有 7 位，而 critical 占有 1 位。然后我们需要为了逐位的操作而修改流化类。

6. switch-case 优化

为了更高的性能，把 RPC 存根中的 switch-case 替换成一个二分搜索树也是一个好办法，这就在 n 个 RPC 函数的时候有了 $O(\log n)$的复杂度。

7. 调试和剖析工具

通过把一些探测代码添加到 RPC 代理和存根代码模板中，剖析就变得容易一些。例如，当要跟踪每个到达或发出的消息时，只要加上把函数名和参数值打印到控制台或调试输出的代码就可以了。当然，也可以查看表示传递了哪个函数和参数的跟踪日志记录。这是一个很好的方式，有助于检测来自坏的游戏客户端的错误接收或造成很长处理时间的 RPC 体函数。

8. 同步 RPC

大部分游戏几乎都不需要同步的 RPC；但是，如果程序需要，可以在运行期把这个功能加到 RPC 中。

把同步 RPC 添加到例子中需要一些工作。

- 需要定义更多的序列，等到 RPC 体返回并把返回值传给 RPC 函数调用者。
- 只有受保障的消息才被同步 RPC 接受。
- 需要添加一个参数方向的选项，分别是输入参数（*inparam*）和输出参数（*outparam*）。Inparam 发给 RPC 服务端，而 Outparam 在 RPC 体执行完之后从 RPC 服务端收到。为此，应该把参数方向语法添加到 IDL 编译器中，并在存根处增加序列化 outparam 的代码，在代理处

增加解序列化 inparam 的代码。这是一个同步 RPC 的例子。

```
UserLogin([in] string id,[in] string password,
        [out] int loginResult);
```

需要考虑特殊情况，比如超时和异常，来避免死锁。

需要管理函数调用序列号。当代理收到 RPC outparam 消息的时候，它可以确定哪个线程在等待这个调用序列号。为了只唤醒一个对应于那个序列号的线程，需要准备一个队列，包含等待 outparam 消息的 RPC 调用。

6.4.7 总结

写一个 RPC 实现通常要有一些和游戏程序员无关的预备知识。本文试图指导大家开始建立自己的 RPC 系统，鼓励大家探索和查看可以从 RPC 系统中得到的生产力、效率和性能。为了最好地理解本文，可能应该看一些参考书。

6.4.8 参考文献

[Breck99] eNITL. *The Network Improv Template Language*. Available online at *http://networkimprov.com/enitl/enitl.html*.

[Parr04] Parr, Terence. *ANTLR Parser Generator v2.7.4*. Available online at *http://www.antlr.org*.

[Sebesta02] Sebesta, Robert W. *Concepts of Programming Languages 5th Edition*, 109–123, 155–159. Addison Wesley, 2002.

[Silberschatz02] Silberschatz, Galvin, and Gagne, *Operating System Concepts 6th Edition*, 121–124. Wiley, 2002.

6.5 在对等通信中克服网络地址转换

Jon Watte
hplus-gpg5@mindcontrol.org

对等（peer-to-peer）计算革新了计算方式。从早在 20 世纪 70 年代后期研究用的 Internet，经过 90 年代前期的拨号 Internet，一直到现在大量的宽带网，每个人都使用计算机网络来娱乐和挣钱。在本文发表的时候，多于 50%的 Internet 用户都有了宽带。虽然这样的普及带来了很多问题：自我复制的蠕虫、死亡之 ping（ping of death）和 Windows 的安全问题，加上商家很严重的恐吓策略，使得大部分用户建立了防火墙来保护他们的计算机。连接共享防火墙是卖得很火的东西，在有多台计算机的家中共享一个宽带连接是最简单的路由方式；一个功能丰富，带有路由器、交换机和防火墙功能的无线访问节点，其价格下跌到了 100 美元以下。

在线游戏可能是计算机网络技术中最令人兴奋的应用之一了，可以让来自全世界的玩家互相竞争，不论是来自你自己的办公室、起居室还是咖啡店。这巧妙地改变了我们对“伙伴”一词和如何选择在线交互的看法。Sony Online Entertainment 和 NCSoft 之类的公司让他们的商业运营在大的服务器群集上，来自全世界的玩家都可以在那里见面和游戏。然而，这个大的服务器群集需要很多的租金、电力和网络费用，所以这样的服务通常会让最终用户交纳月租。因此，另一种流行的在线游戏形式是对等游戏（*peer-to-peer-gaming*），其中一个玩家的计算机也作为游戏的服务器，所有其他玩家都连接到这个玩家，这被称为建立游戏主机。

不幸的是，一些游戏在与其他人的连接模型上支持得不好。通常，防火墙和连接共享软件会妨碍游戏主机的建立，有时候甚至会妨碍加入游戏。虽然一些用户有办法在防火墙中配置 DMZ 或端口转发（port forwarding），但有很多人不知道该怎么做。甚至对于有技术的用户，请求 Starbucks®的员工让你访问他们的无线访问“热区”，也不会让建立 *Warcraft® III* 的游戏主机变得更容易。作为网络游戏的开发人员，你应该做得更好，本文将让你知道该怎么做。

6.5.1 读者

本章面对所有希望游戏在互联网中上尽可能地工作得好的游戏程序员。当程序员希望让游戏能支持点对点、玩家建立主机、扩展尽可能多的用户而不必告诉用户如何去配置端口转发以及设置 DMZ 和移除防火墙的

时候，这显得尤为重要。

假设大家熟悉一般的 Berkeley 套接字编程（Unix）或 Windows WinSock Version 2（Windows）。这里演示的技术在任何操作环境下都很好。MacOS X（及以上）工作起来很像 Unix，但是本文的代码还没有在 MacOS X 上验证过。本文在代码样例中使用的是 C++，但这项技术也可以用任何有能力在套接字层使用网络的语言来表达。

6.5.2 IP 地址

所有网络的核心中都有网络地址。Internet 大量地使用了称为 IPv4 的地址格式，它是 Internet Protocol Version 4 的缩写。前三个版本看起来已经很久远了，以至于几乎查不到了。还有一个就要到来的版本，叫做 IPv6，目前尚未被主流接受。本文中的“IP 地址”是“IPv4 地址”的同义词。

一个 IP 地址只是 4 个字节的序列。一般来说，它需要用点号分开来写，比如 192.168.1.2。一个 IP 地址表示一个特定的网卡或 modem 连接到 Internet 上一个特定的设备；这些独立的硬件设备被称为网络接口（*network interfaces*）。

连接到 Internet 的设备可以运行多个服务。例如，一个 Web 服务器通常会运行一个安全外壳守护进程（SSHD），允许管理员访问服务器的命令行界面来进行监视和维护。为了区别目标是 Web 服务器的网络包和目标是 SSHD 的网络包，这两种最常见的 Internet 协议加入了端口号的概念。每个服务会用自己的端口；对于网络服务器，未加密服务的端口是 80，而加密服务是 443。对于 SSHD，端口是 22。

对于游戏来说，开发者经常会选择一个允许范围中的任意端口。TCP 和 UDP 的端口地址是两个字节，因此范围是在 0～65535 之间，而传统规定的端口号要在 1024-49151 之间选择。这个传统似乎始于 Unix 上的一个实现选择，其中端口 1～1023 保留为机器上特殊的管理用户“root”，只有 root 才可以在这些端口上打开新的套接字监听流量。更多的信息，请参见[Man3rresvport]和[RFC793]。

因此 UDP 或 TCP 地址有 6 个字节：4 个范围在 0～255 的单字节数字，叫做 IP 地址，一个范围在 0～65535 的双字节数字，叫做端口。IP 协议组指定了所有大于一个字节的数字都以高字节在前（big-endian）传送，而且大部分套接字实现都支持名字叫做 `htonl()`、`htons()`、`ntohl()`和 `ntohs()`的函数或宏，以支持数字在本地机器表达和良好定义的网络表达之间转换。

6.5.3 套接字使用

在机器上运行的程序（在本章的剩余部分，我们把这个程序叫做“进程”，把机器叫做“节点”）经常有两个角色：服务端和客户端。一个服务进程将用`::socket()`建立一个套接字，并用`::bind()`绑定到一个本地端口，如果这个节点有多个网络接口，就需要绑定到多个本地地址之一。如果套接字是用于 TCP 连接的，服务器将调用`::listen()`并进入一个循环，在循环中调用`::accept()`来等待接受从客户端进入的连接。`::accept()`会对每个新客户端返回一个新的套接字，数据在这个套接字上用`::send()`和`::recv()`来交换。为了

让不同的连接分开，每个客户端的 TCP 连接会自动分配一个新的、未用的端口来用于特定的服务连接。

如果套接字使用的是 UDP 连接，进程将进入一个循环，::bind()之后直接在套接字上调用::recvfrom()。来自所有客户端的所有进入流量都到达同一个套接字。网络层完全不懂无连接，所以如果多个相关的数据包通过 UDP 交换，那某个更高层的软件（比如进程本身）必须小心调度。用 UDP 从服务端返回数据通常是通过::sendto()发送。

对于 TCP 服务的客户端，在调用::socket()之后，进程将调用::connect()来建立一个到想要的接收者的连接；一旦连接了，::send()和::recv()用来做数据交换。::connect()的参数是一个 TCP 地址，可以从一个文本的名字获取（比如“www.there.com:80”），使用如::gethostbyname()这样的名字服务函数。同时，UDP 的客户端只要调用::sendto()，传进 IP 地址（这里也可以用名字解析库来从文本形式得到），并用::recvfrom()接收数据回来。

在一个典型的进程里，同时会做很多事情，所以把“发送”和“接收”想像成序列事件通常是错误的。在典型的面向事件的网络计算机游戏中，物理模拟、图形绘制、音频混合和磁盘 I/O 经常是与网络同时发生的其他事件。管理这些重叠任务的方法通常会使用::select()来轮询网络，以确定能完成什么，或者产生多个线程来处理程序不同部分的操作。本文的支持代码使用了::select()（或在 Linux 下，类似::select()的函数是::poll()），因为它的同步开销较小，而且线程的 bug 比较少；这里呈现的技术在任何一种世界视图中都同样能用。

一个特定的进程可以同时成为服务端和客户端。对于 TCP，这通常意味着要建立至少两个套接字，一个监听和接受新到来的连接，另一个独立的套接字作为客户端连接到其他地方的服务端。对于 UDP 的进程，不需要多个套接字，因为可以在一个套接字上用::sendto()和::recvfrom()发送和接收任意数量的端点（peer）。实际上，当使用 UDP 的时候，“客户端”和“服务端”的概念只是在应用程序层可见，而不是网络层以下。

纯化论者会注意到 TCP 协议并没有指定“客户端”和“服务端”，但是在实际应用中，调用::accept()的进程是服务端，调用::connect()的进程是客户端。更多的细节请参见[Stevens94]。

6.5.4 路由器、点、协议

还需要一些预备知识来充实对等网络游戏的背景。

第一，Internet 不像电话那样工作——电缆连接到一边的听筒，从概念上讲，它是通过交换机连接到另一边听筒的电缆上，形成一个封闭的电流。取而代之的是，从 Internet 基本构造的观点来看，每个 IP 包独立于其他的包。当一个进程发送一个包的时候，本地节点会确定用哪个界面来发送这个包；然后这个包通过界面到达指定连接的另一端的路由器（这条路径的第一个路由器是本地节点的网关）。然后那个路由器确定把这个包转发到多个可能的界面中的哪一个，并传出去，等等。这个过程一直重复，直到这个包到达接受节点的界面，并被接收进程接收到了，或者传送失败。要牢记每个路由器会检查所有的到来包，并根据某些准则来指定应该转发到哪个界面，或是否只要忽略这个包（叫做“丢弃这个包”）。

第二，因为每个加入 Internet 的节点都有它自己的 IP 地址（而且，为了更多的服务，一个套接字监听一个特定的端口），没有办法只通过检查一个包的 IP 地址就知道这个包是发自客户端或者服务端。在 Internet 上，所有节点在逻辑上都是等价的；它们互为端点。客户端和服务端的概念是由 Internet 用户构造的，一般在每个节点上运行的进程软件中实现。

第三，Internet 上使用着大量的协议。最为常见的一些包括 802.3（用于常规的有线以太网），IP（包含一个 IP 地址的包），UDP（指定一个源和目的端口），和 DNS（用于获取名字服务，把文本节点名解析成 IP 地址）。一个给定的包通常会同时利用这些协议中的多个；一个 DNS 请求将以 UDP 包从 53 端口发给某个名字服务器 IP 地址，传输过以太网，通过第一条到达 DSL 或缆线 modem。每个协议都在底层协议的上面一层，所以 UDP 在 IP 上面一层，而 DNS 在 UDP 上面一层（在一些情况下）。

6.5.5 UDP 包图

一个 DNS 请求包的例子，正如前面解释的，在 IP 层如图 6.5.1 所示。在 IP 的下层，会有附加的框架，比如以太网、ATM 或 PPP，为了清晰起见，排除这一点，因为本文只处理 IP 协议层及以上看得见的现象。同时还要忽略一些细节，比如包分裂或 van Jacobsen 头压缩，它们在本文可见的层次上不会改变 IP 网络的基本操作。

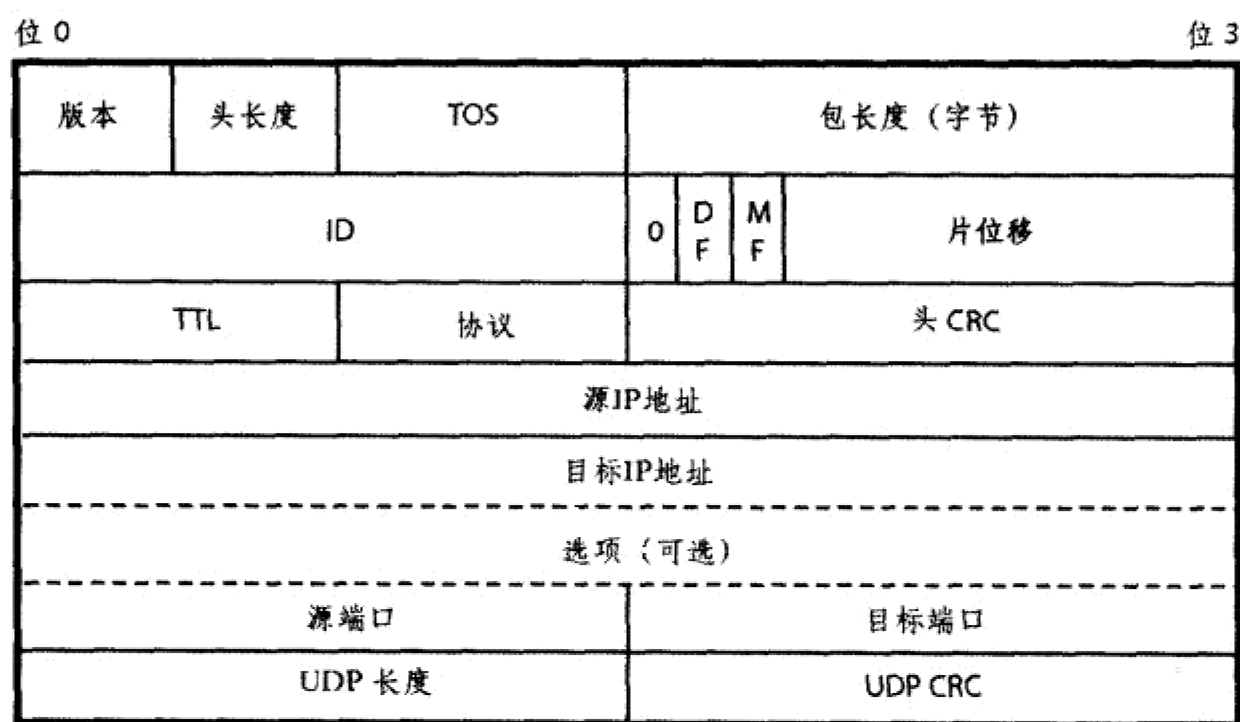


图 6.5.1 剖析一个 UDP 包，有 IP 头、UDP 头和负荷数据

有了所有这些知识之后，就可以写一个如同在 20 世纪 90 年代早期可以看见的在 Internet 上工作的网络游戏。然而，随着时间的流逝，引入的 NAT（网络地址转换）改变了所有这一切。

6.5.6 什么是 NAT

Internet 在 1992 年开始快速地增长，并一直保持增长。因为它是一个对等的网络，Internet 上的每个节点都需要有自己的惟一地址。这些地址是四字节的形式，所以世界上总共可用的

地址超过 40 亿。

1. 问题（IP 地址空间）

在某个时候，我们需要在世界中增加比四字节地址所支持的更多的节点（或至少是界面），更糟糕的是，在路由层面上使用地址的方式会造成相当量的浪费。另外，在世界的不同部分，分配非常不平均。据说 MIT 分配的内部使用的 Internet 地址比整个中国的都要多！不管是不是真的，Internet 已经面临了地址危机，其增长比分配地址能力的增长更快。

2. 另一个问题（安全）

连到 Internet 的节点对其他所有节点来说都是端点。这意味着可以在 Web 浏览器中输入任何的 URL，并把网络流量发给那个 URL 解析出来的节点。不幸的是，这也意味着 Internet 上的任何端点只要愿意就可以把任何网络包发送到你的节点。某些网络服务在设计上更注重这一问题。我们发现很多远程调用（安全漏洞）能通过网络软件（甚至类似于 Windows 这样的操作系统）的缺陷，来接管 Internet 上的任何节点而不需要经过密码认证。

和在必要的时候把机器从网络中拔出来相比，修正应用程序 bug 是避免 bug 造成的安全漏洞的一个更好方法。但是大部分用户没有办法自己修正。

3. 修正

针对 IP 地址空间短缺和阻止不必要的网络传输这两个问题，已形成一个流行的解决法案，那就是网络地址转换（*Network Address Translation*），或简称为 NAT。大部分 DSL 路由器、缆线 modem 和 Internet 连接共享设备都用 NAT 来完成它们的工作。

其思想很简单：大部分网络协议使用 UDP 或 TCP 进行底层传输。在这些协议中，端口号和 IP 地址指定了通信的端点。如果多个节点可以共享一个 IP 地址，但是使用不同的端口号，那么根据附加到地址上的端口，网络就可以以某种方法区分该把一个包发给哪个节点，然后你的全家、公司或其他子网，只要同时连接数少于 65 000，就只需要一个 IP 地址。

像 ARP 这样的底层网络协议让包在节点之间流动，如果多于一个节点共享了同样的 IP 地址，就不能很好地工作，但是有高层的实体完美地用这种方法来实现地址共享：网关路由器。如果所有的节点想要共享路由器一端的同一个地址（从 Internet 的观点上看），路由器是对 Internet 惟一可见的节点，然后所有东西就都好了，假设可以为内部节点找到一些地址，用于互相通信和与路由器通信。

幸运的是，IP 地址空间的一些地址范围保留为试验或私有使用。这些范围是 `10.x.x.x`（有 1.6 千万的内部节点空间），`172.16+x.y.z`（有一百万个内部节点空间），和 `192.168.x.y`（大约 65 000 个内部节点空间）。这些地址保证永远不在公共可见的 Internet 上使用，所以在 NAT 路由的网络内部使用这些地址并不会与任何 NAT 外部的东西相冲突。更多的信息，请参见[RFC1918]。

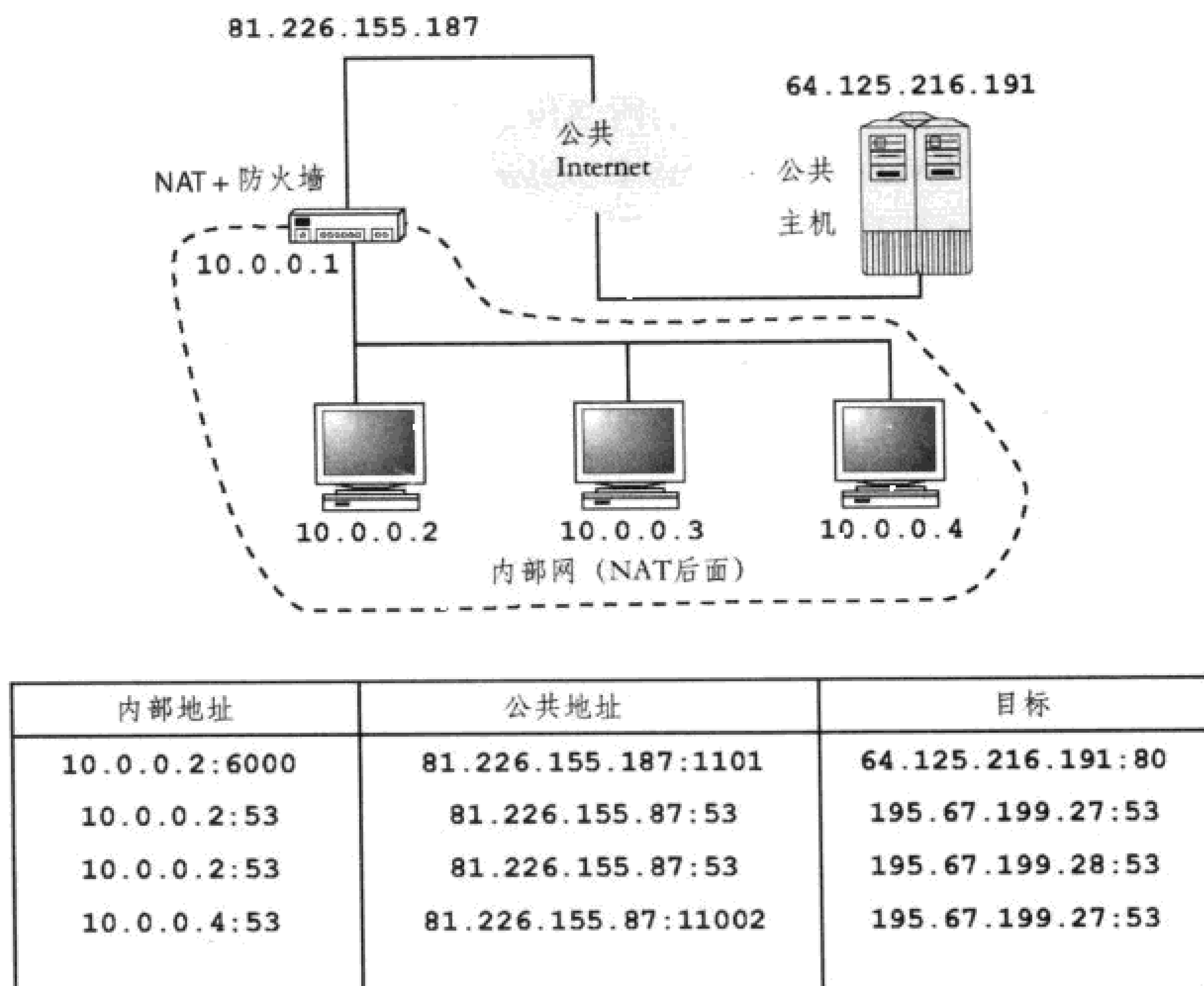
在我们的例子中，NAT 设备在内部的地址是 `10.0.0.1`，在外部的地址是 `81.226.155.187`（对当前使用那个地址的人表示歉意）。NAT 内部的节点是 `10.0.0.2` 和 `10.0.0.3`，我们想要连接的样例站点是 `64.125.216.191`（再次对当前使用该地址的人表

示歉意)。简而言之，使用了 NAT 盒子的用户的 ISP 分配到了 81.x 地址，而 64.x 地址由服务提供商分配给了目标服务器，被使用 DNS 的用户找到。

合起来，当 NAT 内部的节点启动的时候，它们被赋予私有地址范围之外的地址，而它们的网关被配置成 NAT 路由器。当它们要与外面的一些节点通信的时候（比如用于 Web 连接的 64.125.216.191:80），它们和平常一样形成网络包（比如，使用源地址 10.0.0.2:6000)，并把它们转发给 NAT 路由器。然后 NAT 路由器会注意源地址是来自它私有网络的内部，并替换成它自己的、公有的地址作为源地址，所以对于外面的设备来说，看起来就像这个包是来自 NAT 路由器的。

不幸的是，多个内部节点可能使用相同的端口号作为源端口。因此，NAT 路由器必须把源端口号替换成一个新的端口号，并把私有的源 IP 地址替换成它自己的 IP 地址。

最后，为了让返回的网络包返回到正确的节点，NAT 路由器要维护一张（源 IP，源端口，替换 IP，替换端口，目标 IP，目标端口）元组的表，用来重写返回的包。在我们的例子中，如图 6.5.2 所示，一个简单的 Web 请求的元组是表的第一行（10.0.0.2, 6000, 81.266.155.187, 11001, 64.125.216.191, 80)。一旦端口号 11001 在 NAT 上分配为节点 10.0.0.2 端口 6 000 以后，只要连接仍然开着，这个端口号就应该保持分配给那个地址/端口对，否则一个指向 81.266.155.187:11001 的返回包就不能返回正确的地方。



内部地址	公共地址	目标
10.0.0.2:6000	81.226.155.187:1101	64.125.216.191:80
10.0.0.2:53	81.226.155.87:53	195.67.199.27:53
10.0.0.2:53	81.226.155.87:53	195.67.199.28:53
10.0.0.4:53	81.226.155.87:11002	195.67.199.27:53

图 6.5.2 NAT Internet 连接的例子

第三方不能与内部节点通信，因为在开放的 Internet 上，内部节点的私有地址没有意义；来自外部的一个目标是 10.0.0.2 的包不能到达这个网关。同样，如果一个到达 NAT 路由器的包的源地址或端口所在的内部节点最近没有发送数据，在网关上就不会记录用于转换目标地址到内部地址的元组，网关只需要丢弃这个包就可以了。以这种方式，一个 NAT 路由器

就像正式的防火墙，并经常提供给一般的家庭用户所需要的所有保护，避免远程网络攻击。它仍不能保护依赖于用户行为的攻击，比如通过 E-mail 发送木马或诱使用户单击木马 Web 连接，但这是一个很好的第一道防线。

6.5.7 NAT 是如何破坏客户/服务协议的

NAT 路由器会把私有的地址/端口对替换成共有的源地址/端口对，记录下目标的地址/端口，并转发这个包。对于返回的包，它会执行相反的转换，内部网络中的节点可以无缝地和外部世界通信。

不幸的是，这不是对所有的协议都奏效，特别是对那些在 NAT 开始被广泛接受之前设计的协议，或者那些没有正确理解 NAT 的需要的协议。一个基本的例子是文件传输协议（FTP），它广泛地用于从 Internet 下载文件。当设计 FTP 的时候，它被设计为一个节点可以在两个其他节点上建立文件传输[RFC959]，[RFC1123]。控制节点 C 会连接到节点 A，告诉节点 A 需要一个文件，节点 A 开始监听一个端口，然后告诉节点 C 是哪个端口。节点 C 接着连接到节点 B，告诉节点 B 连接到节点 A 的那个端口，发送这个文件。这也允许控制连接保持开放，并当数据在数据连接上传输的时候发出更多的指令。

把文件从控制节点 C 传送到服务节点 B 的一般情况在 FTP 协议中是作为特殊情况处理的，其中节点 C 的端口和地址会转发给节点 B，节点 B 连回节点 C 以真正地传送数据。

问题在于，因为接收节点监听的是一个端口，而不是到外部服务器的活动连接，所以如果存在 NAT 路由器，那么节点 C 的本地端口和地址对要连回来的节点 B 来说就没有意义。NAT 路由器把向外请求的源地址进行转换用次连接回节点 C，但是它对 FTP 控制连接真实的数据流内部却一无所知，因此这个套接字等待连接的内部地址和端口都通过未修改的数据传了出去。

基于 UDP 的游戏也可能犯和 FTP 协议相同的错误，一个例子可以在程序清单 6.5.1 中找到。

程序清单 6.5.1 嵌入式地址代码的样例

```
1: Embedded Address Code Sample// 用于连接到服务器的包结构例子
struct CmdHello {
  char cmd;
  char len;
  unsigned short port;
  unsigned int addr;
};

// 获取套接字的局部地址；一般对公共 internet 是没用的
struct sockaddr_in addr;
socklen_t len;
len = sizeof(addr);
::getsockname( sock, (struct sockaddr *)&addr,
      &len );

// 把局部地址放入包中发给远程地址是不好的——你自己不要这么做!
```

```
Hello h;
h.cmd = CMD_HELLO;
h.len = sizeof(h);
h.port = addr.sin_port;  // don't do this!
h.addr = addr.sin_addr;  // don't do this!
::sendto( sock, &h, h.len, 0,
      serverAddr, sizeof(*serverAddr) );
```

1. 破解

因为 FTP 是一个非常常见的协议，使用的客户端非常多，必须要寻找一个修正方法。理想的情况下，那样的修正不需要修改所有现存的 FTP 客户端。

这个修正涉及让 NAT 路由器更为智能，让它观察到达某个端口的通信包数据。今天的大部分 NAT 路由器都知道 FTP 控制协议，可以截获包含地址和端口的控制信息，并重写协议的数据，以包含一个公共的地址和端口；路由器也将把适当的元组增加到活动的 NAT 会话表中，以允许来自服务节点的包返回。

虽然 FTP 协议非常的广泛，以至于它在多数 NAT 网关（但不是所有的）中享有了特殊的地位，但是新设计的协议不会指望能达到同样特殊的地位。

2. 修正

让一个网络协议对 NAT 安全的真正修正，是在构造出的协议中，IP 地址和端口号不需要通过数据流传送。确保协议能满足这一点的最简单方式是在客户端到服务端总是新建一个连接，使用 UDP 或 TCP，而且在客户端连接的时候，总是让服务端使用在服务端上可见的地址返回客户端。在服务端之间的返回通道不发生连接信息的转发。点地址的正确管理如程序清单 6.5.2 所示。

现在有些非常高级的数据中心设备能觉察出 NAT。反向 NAT 的一种用途是用来连接的负载平衡，使用安全集群来对付外部攻击和自由分配和管理 IP 地址空间。涉及反向 NAT 的通信系统的设计所面对的共享验证问题必须极其小心地处理，而且超出了本文的范围。

程序清单 6.5.2　正确的地址管理

```
// 为应答一个（正确的）CmdHello 包的建议包
struct CmdHelloAck {
  char cmd;
  char len;
};

// 服务端使用::recvfrom()来正确地得到连接点的地址
struct sockaddr_in addr;
socklen_t len = sizeof(addr);
union {
   CmdHello hello;
} command;
::recvfrom( sock, &command, sizeof(command), 0,
      (struct sockaddr *)&addr, &len );
// 服务端把客户端注册到收到的 hello 包上，然后使用::sendto()来应答收到的地址
```

```
if( command.hello.cmd == CMD_HELLO ) {
    // client_hello()是把新的连接客户端添加到某个内部列表的函数
    client_hello( command.hello, addr );
    CmdHelloAck ack;
    ack.cmd = CMD_HELLO_ACK;
    ack.len = sizeof(ack);
    ::sendto( sock, &ack, sizeof(ack), 0,
          (struct sockaddr *)addr, sizeof(addr) );
}
```

3. 其他问题

这个问题的一个变体是在游戏服务器群集使用一台服务器作为验证登录，然后把关于这个玩家的信息传给另一台服务器，它会把通信返回给客户端。即使客户端用的是公共的 NAT 地址，NAT 路由器还是不会看见来自新服务器的通信，所以返回的通信就被丢弃了。

让一个协议对 NAT 完全安全的规则是总在客户端和不同的服务器之间建立连接，且不把 IP 地址或端口号作为数据发送，而是依赖于`::accept()`，`::getpeername()`和`::recvfrom()`（但不是`::getsockname()`！）来得到另一端节点的地址。

为了支持为一个服务器群集验证用户名和密码的中心登录服务器的想法，可以使用一个加密的 cookie。在服务器之间共享一个公钥（通常是一个 128 位强随机数）。当玩家登录的时候，登录服务器会建立一个用户 ID、登录时间和共享密钥的散列（hash），并把用户 ID、时间和散列返回给用户。然后用户把这个信息提交给所有其他服务器；服务器只要验证用户 ID、时间和密钥（用户不知道）是否符合用户提供的散列，来知道登录服务器在指定的时间正确地验证了这个用户 ID 提供的用户名/密码。这个系统的强度基本取决于公钥的强度（它可以非常频繁地变化）以及散列算法的强度；在 Linux 上，使用来自/dev/random 的 16 字节对于公钥来说工作得很好，而 MD5 通常用做强的散列函数。

如果按照这个建议，客户/服务将是对 NAT 安全的。然而，当使用对等网络的时候该怎么办？

6.5.8 NAT 是如何破坏对等协议的

建立和操作大量的服务器群集是很昂贵的。对一个小游戏的开发人员，或者不想让玩家在初次付费后每个月花费 14.95 美元来访问的游戏开发人员来说，允许玩家建立他们自己的游戏服务器是一个吸引人的选择。这里有一个安全暗示，你不能真的信任任意的建立在某个黑客卧室中的服务器，但是对等网络游戏的流行从 id Software®的 *Doom*™ 到 Dice 的 *Battlefield 1942*™，验证了对等的游戏模型。

然而，前面讨论的客户/服务模型只工作于当服务端有一个端口和地址在 Internet 上公开可见，让玩家可以来连接的时候。这经常意味着在 NAT 路由器之后的用户不能建立游戏，只能加入，而且如果游戏不是设计成对 NAT 安全的，那 NAT 后面的用户甚至不能加入游戏！在现代网络中广泛流行着 NAT，所以这不是一个可维持的情况。图 6.5.3 中的图演示了没有一个节点可以真的与其他人通话（在端口 8960，大概用于某些游戏），因为 NAT 路由器在它们各自的会话状态表中没有适当的条目适用于第二个节点。这就是本文提供的技术可以拯救

你游戏的地方！

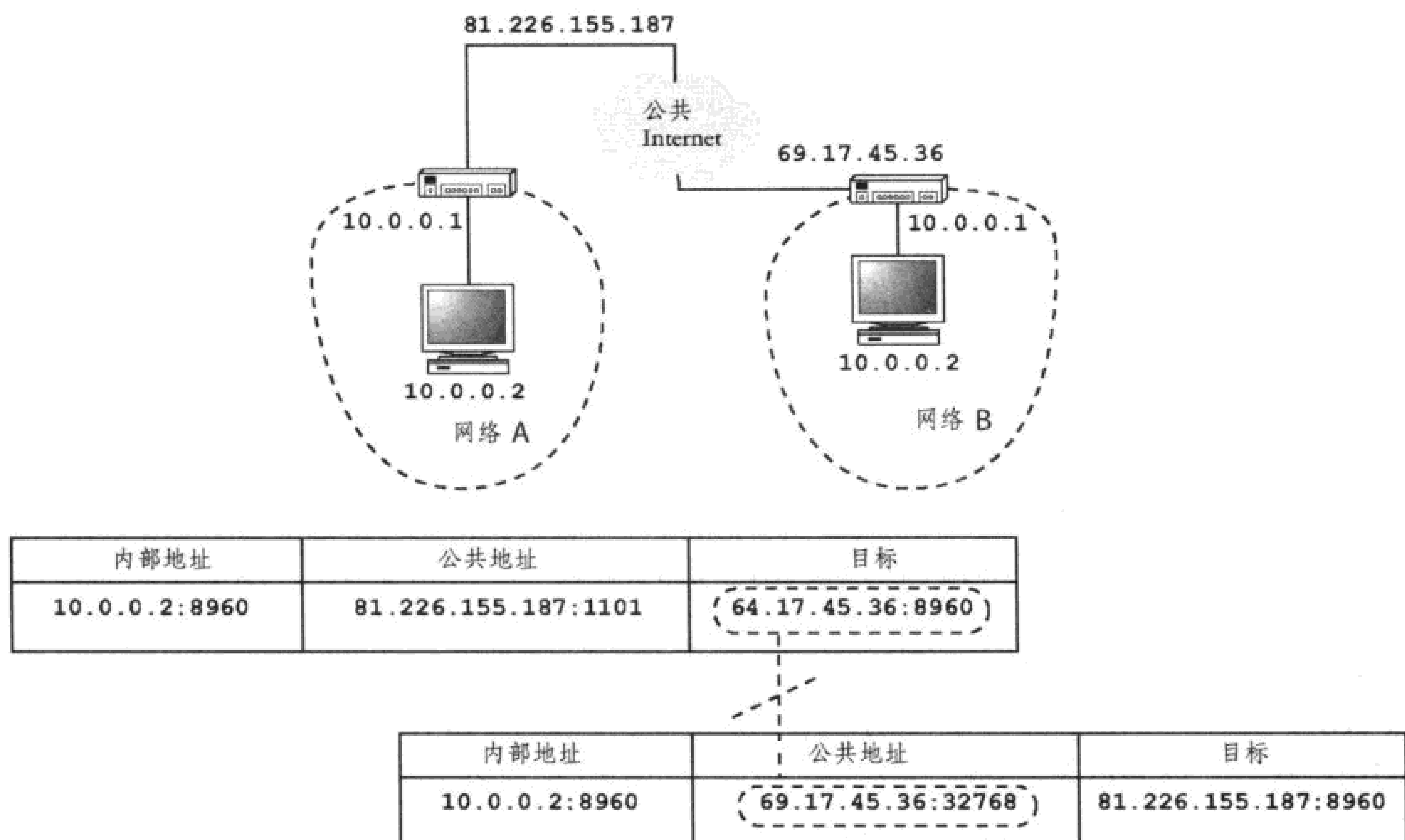


图 6.5.3　没有一个点可以把包发到对方

1. 破解

看看今天许多游戏的 README.txt，可能会发现其中提到了端口转发。端口转发意味着到 NAT 路由器那里，告诉它对到达一个特定端口的包采取特殊情况。在效果上，端口转发会把一个永久的元组（内部 IP，端口，网关 IP，端口，任何远程 IP，任何远程端口）添加到 NAT 路由器的会话状态表中。

效果是任何到达那个指明端口的包只重写为目标 IP 地址的一部分，并转发到作为端口转发目标的特定机器上。有一些线索的游戏将需要你只转发到一个端口，甚至让你配置在游戏中需要使用哪个端口。灵活性更低的游戏会需要一个特定的端口（所以在同一个 NAT 路由器上不能建立两个游戏主机）或需要让整个范围的端口都转发。在后面一种情况中，一个用户可能走到了在他们的 NAT 路由器上建立一个非军事区（DMZ）主机的极端，这就取消了所有处于 NAT 网关后面的安全优势，在效果上，移除了 NAT。

2. 修正

端口转发很麻烦，很多消费者在正确地设置方面有困难，而且在开发者和发布者之间经常有一个支持论战。幸运的是，存在着另一个解决方案，首次在[Kegel99]中由 Dan Kegel 描述，应在了 Activision®发布的 *Battle-Zone*™ 游戏中。

这个问题的核心是每个 NAT 路由器在它的会话状态表中都缺乏一个条目，用于内部节点想要通信的另一端。如果可以以某种方法欺骗 NAT 路由器来加入这样的条目，通信将会正确

地流动，而游戏就可以继续了！

记住当一个点通过 NAT 路由器发出数据包的时候，一个元组就添加到会话状态表中，包括内部和外部地址以及远程地址。作为第一个近似，如果假设 NAT 路由器将保留内部节点的源端口，假设端口没有被其他会话使用，就可以用它的外部 NAT 地址和这个已知的端口把包发送到远程节点。如果每个节点同时用同样的方式开始发送到那个点，那么两者就都在 NAT 路由器上得到了它们希望的端口，然后正确的事情就会发生。

3. TCP 中的问题

这个解决方案的草案可能适合于 UDP(假设解决了端口服用的需要)，但是它不适合于 TCP。理由是 TCP 连接是如何分配的：每个连接分配一个新的端口号来惟一地鉴别这个连接。大体上没有办法猜到下一次会分配到哪个端口，或者 NAT 路由器会如何映射那个端口号，所以除非能在一个时间尝试 65 536 个不同的端口号，否则不能真的让 TCP 连接的三次握手通过 NAT 工作。

Dan Kaminsky 建立了一个实验性的库[Kaminsky03]，用于让 TCP 连接穿透 NAT 路由器，但是这个库的基础看起来不是那么坚实，不足以用于产品代码。鼓励大家看看这个库，因为它是有教育意义的，但是成功率很低，不能把它当做一个可信任的产品。不幸的是，这大约和直接处理 TCP 和通过 NAT 的对等网络一样，所以我们将返回到 UDP，它的成功前景也非常好。

6.5.9 用于游戏的端到端解决方案

在前面草拟的解决方案中，“假设”略微多了一些。首先，端点如何知道同时开始与其他端点通信？其次，如果期望的端口已经在 NAT 路由器上使用了，而且路由器选择把内部端口映射到另一个外部端口，该怎么办？甚至为了实现方便或其他安全因素，NAT 路由器的实现可能总是重新映射端口。

1. 红利问题

从第一个问题开始：端点如何知道同时开始发送？甚至一个端点该如何知道其他可见(NAT)地址是什么？理想的时候，想要呈现给玩家的界面是一个已经建立的游戏主机列表，玩家可以选择建立一个新游戏或者加入一个已经建立好的游戏。这个游戏列表系统的良好实现包括 Blizzard® 的 Battle.net®和 Microsoft®的 Xbox Live™ 服务。Battle.net 上的 *Warcraft III*®用的是 TCP，而对等 NAT 的介入将不能通过 TCP 工作，因为每个连接都分配它自己的本地端口。然而，Xbox Live 使用 UDP，体验是游戏建主将通过大部分 NAT 来工作。所以该如何找到其他玩家一同游戏？

2. 匹配

浏览已建立的游戏不是在对等的情况下可以做到的，除非在本地 LAN 中使用广播。你必须停下来并在公共 Internet 上得到至少一个服务器，在那里建主的游戏可以注册，玩家就能搜索建主的游戏来加入。这个方法的优点是对这个服务器的流量和性能需求很低；每个建主的游戏只需要几百字节，每个玩家连接和获得所有可玩游戏的列表只需要一到两千个字节，甚至是对于有很多活动玩家的游戏也是这样。一晚上一万个玩家，其峰值有三个小时，分解成每秒少于一个玩家；有了这个建议的带宽使用量，甚至可以在一个拨号 modem 上建立它！

实际上，建议使用至少一个最小的专用建主设备，因为有效的正常运行时间和处理负载尖峰是很重要的，建主费用大约每月 100 美元，这是非常划算的。作为红利，你可能可以让公司的 Web 服务器运行在同一台机器上，假设负载很低。如果你的游戏非常成功，把这个解决方案放大是一个非常微不足道的操作。

在代码文件 sample.cpp 中，允许加入的游戏被保存在引入器服务器的全局集合 gHositing 中。在清单 6.5.3 中，加入了对 HOST 协议命令的响应，返回到假定的客户端并响应 LIST 协议命令的代码在清单 6.5.4 中（去掉了错误检测）。

程序清单 6.5.3 增加一个建主的游戏

```
enum {
  MAX_NAME_LEN = 32,
};

// 用于 Request::What::HOST
struct HostRequest {
  unsigned char what;
  char name[ MAX_NAME_LEN ];
};

struct Peer {
  std::string name;
  sockaddr_in addr;
  bool operator==( Peer const & o ) const {
     return name == o.name;
  }
  bool operator<( Peer const & o ) const {
    return name < o.name;
  }
};

std::set< Peer > gHosting;

// 当引入器得到一个主机包的时候，就处理它
void introducer_host( char * data, int len, sockaddr_in & sin ) {
  // 析取包，并确保它终止了
  HostRequest & hr = *(HostRequest *)data;
  hr.name[ MAX_NAME_LEN-1 ] = 0;
  // 记录有新人建立的事实（在现实中，我们也加入超时信息，并执行一些登录多/密码检查）
  Peer p;
  p.name = hr.name;
  p.addr = sin;
  gHosting.insert( p );
}
```

程序清单 6.5.4 返回建主的游戏

```
struct ListResponse {
  unsigned char what;
  char name[ MAX_NAME_LEN ];
```

```
  sockaddr_in addr;
};

// 当引入器得到列表包的时候，就处理它
void introducer_list( char * data, int len, sockaddr_in & sin ) {
  // 我只是送回所有可用的主机——没有限制，没有匹配，最重要的是，没有重发丢失的包
  ListResponse lr;
  lr.what = Request::LIST_RESPONSE;
  for( std::set< Peer >::iterator ptr =
       gHosting.begin();
       ptr != gHosting.end(); ++ptr ) {
    // 构造一个 ListResponse 包
    strcpy( lr.name, (*ptr).name.c_str() );
    // 我把地址作为数据，但这没问题，因为地址是公共可见的(::recvfrom())。
    lr.addr = (*ptr).addr;
    // 发送这个包，忽略错误（包可以在任何地方丢失，包括网络层——太糟了）
    ::sendto( gSocket, (char const *)&lr, sizeof(lr),
        0, (sockaddr *)&sin, sizeof(sin) );
    }
    // 发送最终的、空主机名来终止这个列表
    lr.name[0] = 0;
    ::sendto( gSocket, (char const *)&lr, sizeof(lr), 0,
        (sockaddr *)&sin, sizeof(sin) );
    }
```

3. 引入

一旦在公共 Internet 上有了一个游戏浏览服务器，客户端可以用绑定到一个已知端口的套接字上的 UDP 连接到这个服务器。当游戏浏览服务器调用`::recvfrom()`来接收网络包的时候，这个套接字的公开可见的、NAT 转换的地址将被它知道。这个浏览服务器要把这个 IP 地址和端口号发送给所有想要加入游戏主机的未来客户端。

另外，浏览服务器应该把想要加入游戏主机的客户端公开可见的 IP 地址和端口号发送给游戏主机。用这种方式，游戏服务器可以试着主动把包发送给加入的客户端。这样的效果就是把适当的元组添加到 NAT 路由器的会话状态表中，以允许从未来客户端发送的包能到达主机服务端。

作为公开可见 IP 地址和端口仓库的服务器，为了穿透 NAT 把这些提供给其他相关的端点，这通常被称为"引入器（introducer）"。

根据图 6.5.4，与我们"正确地猜测"公共端口号的情况不同的是，NAT 路由器用来表示内部节点的端口是任意的，因为它们对使用引入器的其他端点也可见。端口总是未使用的想法就不再必要了。

4. 实现细节

这里有一个让基于引入器的解决方案工作的基本假设：当一个特定的（源 IP，源端口）地址用于源地址的时候，NAT 路由器将把这个转换成一个特定的（NAT IP，NAT 端口），不管目标 IP 和端口是什么。虽然仍需要把包发送到引入器和要引入的端点，以在 NAT 路由器中建立出整个会话状态元组，如果 NAT 对不同的目标地址选择不同的端口，即使是相同的源

端口和 IP，那么从远程端点返回的流量就不能到达希望的端口，那这项技术就不能用了。

内部地址	公共地址	目标
10.0.0.2:8960	81.226.155.187:11002	64.125.216.191:8960

1

内部地址	公共地址	目标
10.0.0.2:8960	69.17.45.36:32769	64.125.216.191:8960

内部地址	公共地址	目标
10.0.0.2:8960 10.0.0.2.8960	81.226.155.187:11002 81.226.155.187:11002	64.125.216.191:8960 69.17.45.36:32769

2

内部地址	公共地址	目标
10.0.0.2:8960	69.17.45.36:32769	64.125.216.191:8960

内部地址	公共地址	目标
10.0.0.2:8960 10.0.0.2.8960	81.226.155.187:11002 81.226.155.187:11002	64.125.216.191:8960 69.17.45.36:32769

3

内部地址	公共地址	目标
10.0.0.2:8960 10.0.0.2:2960	69.17.45.36:32769 69.17.45.36:32769	64.125.216.191:8960 81.226.155.187:11002

第 1 步，每个端点都对引入器打开一个会话，它有公开的 IP 和端口地址。引入器会记录每个参与端点公开可见的 IP 和端口，并让它们对其他端点可见。

第 2 步，第一个端点收到第二个端点的公开地址，并发一个包给第二个，这就会在 NAT 网关上建立一个会话条目。在这个时候，从第二个端点到第一个端点的通信将会穿过 NAT，但第一个端点发送的包会在第二个网关上丢弃。

第 3 步，另外，第二个端点收到了第一个端点的公开地址，这样的包就建立了一个会话。因为第一个端点已经发送了建立会话的包，而且因为相同的本地端口用在了和引入器以及端点的通信上，所以现在包可以双向流动了。

图 6.5.4 使用了引入器的对等通信建立

幸运的是，有三个好理由让 NAT 路由器对相同的（源 IP，源端口）重用相同的 NAT 端口。

第一，在公共的 Internet 上，一个客户端进程在绑定到一个特定地址和端口的套接字上发送网络包，会希望那个包的发送者保持固定，不管目标是什么。毕竟，应用程序把套接字绑定到特定的端口，并使用这个套接字来把包发送到不同的远程节点。注意这只对 UDP 有效，而不是 TCP，它在设计上就在客户端为每个连接分配新端口。

第二，如果 NAT 路由器在它的内部网络中有多个节点，它会需要分配许多端口用于大量同时活动的会话。可用端口的数量不是无限的，所以 NAT 将节约端口空间，如果它为相同的

内部地址、端口号重用相同的外部端口，就可以服务更多并发的服务端。

第三，NAT 引入很快就会成为标准的技术。不支持这个技术的 NAT 网关将被用户认为是不完善的，退货率和维护开销也比能正确工作的设备更高。从 2001 年以来，这个行为同样也积极地鼓励了所有 NAT 构建者，通过 Internet 协会文档 RFC3022 [RFC3022]。

实际上，大部分网关都能与这项技术合作得很好，因为它们执行了正确的端口匹配。有少数报告不能用的；惟一一个确认不兼容于 RFC3022 的是基于 BSD 操作系统的，而在家庭环境中极其罕见。在 1999 年，一些 NAT 路由器不能正确地处理这种情况：从内部网络发送到外部地址，而那个外部地址代表了同一个内部网络的内部地址，也就是当处于同样的 NAT 路由器后面的两个端点加入了一个引入器的情况。这个 bug 已经被大部分路由器解决掉了，今天市场上的大部分设备应该能正确处理这个情况。虽然没有办法绕开 NAT 网关不重用端口的 bug，但是你可以绕开不允许内部主机通过公开可见的 NAT 地址通信的情况，通过开始通信的时候同时使用端点的内部和外部地址，响应真的得到回复的地址，如果两个地址都有回复，那就拿公开地址，因为公开地址保证是惟一的。

如果发现在 NAT 网关之后的客户端仍然不能用，你要么出手，告诉用户去修正网关，要么配置端口转发，要么可以决定消耗服务这些用户的带宽（因为它们相对小），并从可以连接得很好的引入器上反射他们的包。选择哪个选项几乎完全取决于想付出的低支持开销和低建主开销之间的权衡。

最后，WinSock 中有一个实现缺陷，这是大部分个人电脑用来连接到 Internet 的套接字库（是 Microsoft 集成到 Windows 中的）。如果你发送一个 UDP 数据包到没有监听的端口和接收到回复的 ICMP 消息“端口不可达”，WinSock 将卡住发送初始包的套接字，并在试图使用这个套接字的时候返回 `WSAECONNRESET`。这时候，必须关闭和重开这个套接字才能再次使用。这非常不方便，因为在建立对等 NAT 穿越的时候，在两个网关都建立了适当的会话状态记录之前，很可能收到至少一个端口不可达的消息。幸运的是，因为 UDP 是无连接的（与 TCP 不同），NAT 网关不会知道套接字已经关闭、重开和绑定到本地机器的相同端口，所有东西将会和平常一样处理。典型的 WinSock 代码因此将会像程序清单 6.5.5 那样处理这个问题（再次地，去掉了一些错误检测）。

程序清单 6.5.5　WinSock 上的变通方案

```
enum {
  GAME_PORT = 8960,
};
#define SOCKET_ERRNO WSAGetLastError()
inline bool SOCKET_WOULDBLOCK_ERROR( int e ) {
  return e == WSAEWOULDBLOCK;
}
inline bool SOCKET_NEED_REOPEN( int e ) {
  return e == WSAECONNRESET;
}
#define INIT_SOCKET_LIBRARY() \
   do { WSADATA wsaData; WSAStartup( \
         MAKEWORD(2,2), &wsaData ); } \
   while(0)
```

```
SOCKET gSocket;

// 分配一个全局的套接字，我们在所有角色中都用它
void allocate_socket() {
  if( gSocket != BAD_SOCKET_FD ) {
     ::closesocket( gSocket );
  }
  else {
    INIT_SOCKET_LIBRARY();
  }
  gSocket = ::socket( PF_INET, SOCK_DGRAM,
       IPPROTO_UDP );
```

// 在所有局部界面上绑定到我的端口。因为我要在一台机器上运行多个实例，我尝试一系列的端口。一旦我绑定到一个端口，如果我重分配了套接字，就要重用那个端口。所以用一个静态变量记住哪个端口用过了（这意味着用这段代码，我只可以为每个进程开一个套接字）

```
  static int portUsed = 0;
  for( int port = GAME_PORT; port < GAME_PORT+10;
        ++port ) {
    sockaddr_in addr;
    memset( &addr, 0, sizeof( addr ) );
    addr.sin_family = AF_INET;
    if( portUsed ) {
      //  如果设置了，就用旧端口
      port = portUsed;
    }
    addr.sin_port = htons( port );
    // 绑定套接字到指定端口
    int r = ::bind( gSocket, (sockaddr *)&addr,
        sizeof(addr) );
    if( r < 0 ) {
      if( portUsed ) {
          //  如果我不能重用旧端口，就跳出
          break;
      }
    }
    else {
      portUsed = port;
      break;
    }
  }
}
```

// WinSock 中有一个缺陷，我必须重开套接字，如果在套接字上出现了 CONNRESET。因为当它收到 ICMP 的端口不可达的时候，它会卡住套接字，这会在 NAT 引入商议的时候。

```
bool maybe_reallocate_socket( int r ) {
  if( r < 0 ) {
    if( SOCKET_NEED_REOPEN( SOCKET_ERRNO ) ) {
      fprintf( stderr,
    "Re-allocating socket because of WinSock.\n" );
```

```
        allocate_socket();
      }
      return true;
    }
    return false;
  }

      sockaddr_in sin;
      socklen_t slen = sizeof( sin );
      char data[ 512 ];
      // 等待到来的包
      int r = ::recvfrom( gSocket, data, 512, 0,
          (sockaddr *)&sin, &slen );
      if( maybe_reallocate_socket( r ) ) {
        continue;
  }
```

6.5.10 总结

使用本文介绍的技术，能够在各种网络配置下更进一步提高多人在线游戏的鲁棒性。结果，巫师和勇士就可以去执行拯救世界的任务，而不是满足于他们本地的 Starbucks。

6.5.11 参考文献

[Man3rresvport] Unix section 3 manual page for the rresvport() library call; for example found at *http://www.gsp.com/cgi-bin/man.cgi?section=3&topic=rresvport*.

[RFC793] USC ISI. Request For Comments document 793. TCP Protocol Specification. Available online at *http://www.faqs.org/rfcs/rfc793.html* (September 1981).

[Stevens94] Stevens, W. Richard. *TCP Illustrated*. Addison-Wesley Professional, 1994.

[RFC959] Postel, J. and J Reynolds. Request For Comments document 959. Available online at *http://www.faqs.org/rfcs/rfc959.html* (October 1985).

[RFC1123] Braden, R., editor. Internet Engineering Task Force. Request For Comments document 1123. Available online at *http://www.faqs.org/rfcs/rfc1123.html* (October 1989).

[Kegel99] Kegel, Dan. "NAT and Peer-to-peer Networking." Available online at *http://www.alumni.caltech.edu/~dank/peer-nat.html* (July 1999).

[RFC1918] Rekhter, Y., et al. Request For Comments document 1918. "Address Allocation for Private Internets." Available online at *http://www.faqs.org/rfcs/rfc1918.html* (February 1996).

[Kaminsky03] Kaminsky, Dan. Paketto Keiretsu 1.10: Advanced TCP/IP Toolkit. Available online at *http://www.doxpara.com/read.php/code/paketto.html* (December 2002).

[RFC3022] Srisuresh, P. and K. Egevang. Internet Society. Request For Comments document 3022. Available online at *http://www.faqs.org/rfcs/rfc3022.html* (January 2001).

6.6 一个可靠的消息协议

Martin Brownlow
martinbrownlow@msn.com

本文描述了一个简单的协议，用于实现可靠、有序的网络通信消息。这个协议独立于传输介质或网络模型（客户/服务或对等），所以可以实现在任何需要可靠网络的解决方案中。

6.6.1 术语定义

在开始看本文细节之前，应该首先花一些时间来定义本文的剩余部分使用的术语：

主机/客户端：为了本文的目的，将使用术语主机来表示消息的发送者，使用客户端表示接收端。

包：包是物理上传输过网络的信息。它由一条或多条消息组成。如果包中的任何一条消息被设置成可靠的，那么这个包就被认为是可靠的。

消息：一条消息是应用程序可以发送数据的最小形式。每条消息都可以设置成可靠或不可靠的。在每一帧，应用程序建立一系列的消息，然后由网络库打成一个包。

系统消息：一个由网络库发送的消息。这是独立于应用程序的，应用程序从不需要知道它们。系统消息和应用程序特定的消息以完全相同的方式打到包中。系统消息的一个好例子是包收到的应答。

心跳：心跳是一条系统消息，当没有其他消息处理的时候就频繁地发送。心跳的目的是向接收者表示发送者的应用程序仍然在正确地运行。

应答：应答是一条系统消息，声明了客户端已经从主机收到了一个指定的包。

消息处理器：一个应用程序的函数，网络库调用它来处理到来的消息。

6.6.2 为什么要可靠的消息

可靠消息对于任何网络库都是一个重要的部分。Internet 虽然是一个技术的集合，对于数据包来说却不是一个非常安全的地方。组成 Internet 的节点在任何时候都有可能收到不可预测数量的流量，而且没有义务保证所有的流量都传递了；如果一个节点被淹没了，为了保持运作，就会丢弃到来的包。另外，两台机器之间连续的包不保证以同样的过程传过中间的网

络。因为流量在 Internet 的节点之间变动，包被路由到不同的路径，以试图提供最好的路径。

Internet 的这些特性给了我们两个截然不同的问题：发送的任何数据包都可能无法到达它的目标，任何两个数据包到达目标的顺序可能和原先发送的相反。然而，应用程序发送的一些数据必须绝对到达目标；如果数据消失了，应用程序的行为就是不可预测的，甚至可能崩溃。因此，需要定义一种方式来确保一个给定的包会在合理的时间内到达目标。

6.6.3　传统的可靠消息

现在我们定义了将要使用的术语，并知道了为什么可靠的消息很重要，下面可以看看传统实现中的可靠消息模型是什么样的。这会让我们熟悉优点和弱点，以构建更好的模型。

在传统的实现中，当主机把一个可靠的包发给客户的时候，它会记录发送的时间，并把可靠的包放进列表中。这个可靠的包只有在收到来自用户的应答系统消息之后才从列表中删除。如果经过了一定的时间之后，这个可靠包的应答还没有收到，这个包就需要重发，并把计时器复位。

当一个客户端收到一个标记为可靠的包，它必须构建一个应答系统消息，传回主机。这个消息包含要应答的包的标识符。如果这个消息没有及时地发送，主机就会认为这个包没有收到，并重新传输它。

从这个描述，可以看到如果一个给定包的应答在包的定时器走完之前没有到达主机，主机将重发这个包，消耗一定的带宽。这有两个重要的结果。第一个是应答必须立刻发送，如果没有其他消息要发送的话，在那个时候就要为它们建立一个包。第二个是应答在发送后必须到达，以避免主机不必要地重发可疑的包。这意味着应答消息应该可靠地发送，但是这么做就得有“应答的应答”，而且在你知道之前，每个包都得设置为可靠的。

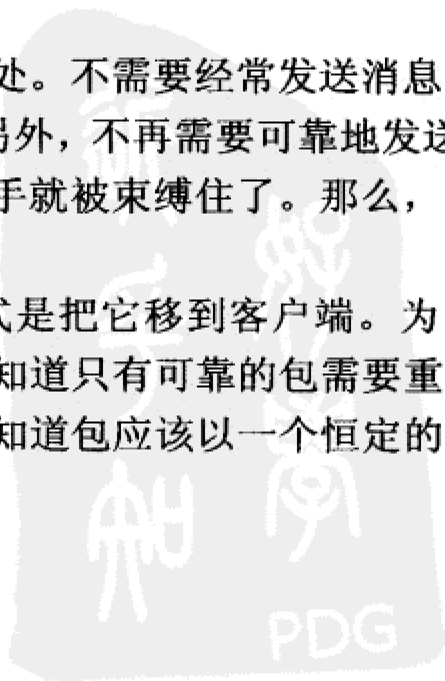
6.6.4　一个简单的方法

从前面对传统可靠消息实现的描述，可以看到一些地方需要改进。其中两个最重要的地方是减少应答发送的次数，以及去掉它们必须可靠发送的需要。

减少应答发送的数量可以通过在可靠消息中增加一个额外的限制来部分地实现。如果可以保证可靠的消息必须以它们发送的顺序来处理，然后收到一个给定包的应答自动地表示接收到了前面的每一个包。这是因为，为了维护顺序的限制，我们知道所有前面的包也必须处理过了。然而，因为需要快速地发送应答包，以避免主机重发它认为已经丢失的包，所以这种方法的实用性稍微降低了。

如果可以去掉快速发送应答的需要，就可以获得几个好处。不需要经常发送消息，而且当发送一个的时候，只要应答最近处理的可靠包就可以了。另外，不再需要可靠地发送应答。不幸的是，当由主机负责删除一个丢失的可靠包时，我们的手就被束缚住了。那么，这就是我们主要的攻击点。

从主机上去掉负责删除一个丢失的可靠包的最简单方式是把它移到客户端。为了这么做，客户端必须快速地检测一个可靠的包已经丢失了。我们知道只有可靠的包需要重发，而两个理想的可靠包将分解成一个或多个不可靠的包。我们也知道包应该以一个恒定的频率到



达——因为有心跳系统消息。

利用这点知识，可以组织一个用来快速检测丢失的可靠包的方法。最简单的方式是允许每个到达的包都包含足够的数据来推断一个丢失的可靠包。因为我们知道包以一个恒定的频率到达（因为心跳系统消息），就可以快速地做出这样的推断。例如，如果一秒钟之内到达10个包，那么在下一个包到达的时候，也就是十分之一秒之后，就可以推断出是否存在丢失的包。

1. 包标识符

然而，如何让一个到来的包判断出存在丢失的可靠包？每个可靠包都需要两部分数据：一个标志，用来说明这是可靠的，以及一个标识符，它的意思是可以应答。另一方面，一个不可靠包只需要一个标志来表明它是不可靠的；它不需要标识符。这意味着，是否应该赋一个标识符给不可靠的包，而且任意数量的不可靠包都可以有相同的标识符，并不会有什么副作用。通过可靠/不可靠标志和包标识符的组合，可以建立一个简单的包编号系统，允许我们推断存在丢失的包。

如果使用包标识符中最不重要的位来存储可靠/不可靠的标志，就可以看到，如果选择值为0的位来表示可靠的包，那么所有可靠的包都是双数的包标识符，而所有的不可靠包都是奇数的。然后，利用任意数量的不可靠包都可以共享同一个标识符的事实，我们可以提出以下两个规则：

- 当建立一个不可靠包的时候，包标识符应该把可靠/不可靠标志设为1。
- 当建立一个可靠包的时候，包标识符应该增加2，然后把可靠/不可靠标志设为0。

这个简单的规则就使得客户端可以检测一个可靠包是否丢失了，只要通过收到了任何类型的后来发送的包。为了达到这一点，客户端必须维护一个记录，表示下一个期望的可靠包的标识符的，我们把它称为 nextReliableID。

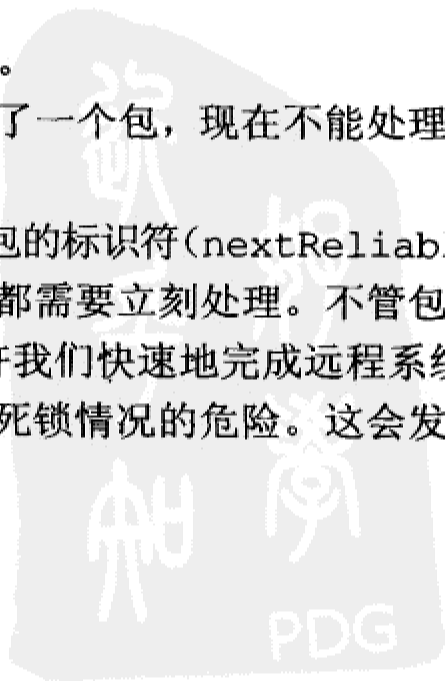
2. 到达包的队列

当收到进入的数据包时，它们会被放入一个队列，以包的标识符排序。网络库每次更新的时候，它都会处理这个队列，从最小标识符的包开始。来自队列的包按顺序处理，直到队列已经空了或者发现有一个不能处理为止，根据下面的准则：

- 如果包标识符是偶数（可靠的）而且等于 nextReliableID，就收到了下一个可靠的包。处理这个包，然后把 nextReliableID 增加2。
- 如果包标识符是奇数（不可靠的），而且等于 nextReliableID 减 1，就收到了一个需要处理的不可靠包。
- 如果包标识符小于 nextReliableID，丢弃这个包。
- 如果包标识符大于 nextReliableID，就知道丢失了一个包，现在不能处理这个新包或队列中所有后面的包。

在处理完到达包队列之后，就知道是否丢失了包，以及丢失包的标识符(nextReliableID)。

这些规则的一个明显例外是，到达包中的任何系统消息都需要立刻处理。不管包是否在队列中都会发生（注意每条系统消息只能处理一次）。这允许我们快速地完成远程系统请求，比如重发和应答。如果不处理这些系统级的消息，就可能有死锁情况的危险。这会发生在当



两台机器都检测到来自对方的包丢失了，它们向对方重发的请求都放在丢失包后面的队列中了，重发的请求就不会被处理，因此机器就都死锁了。

3. 重发计时器

如果一个包真的丢失了，就会检查一个叫做重发定时器的下降计时器是否已经激活，如果不是，就赋给一个小的值并启动。这个小值叫做顺序颠倒的延迟（*ont of order delay*）。这个小暂停的目的是让丢失包有一个小的到达时间，对付仍在传送但顺序颠倒了的情况。如果在任何时候，这个丢失包到达了，就停止这个重发计时器。

当重发定时器终止的时候，客户端会发送一个系统消息到主机，请求重发这个丢失的包（`nextReliableID`）。在这时候，重发定时器设置为一个大的值，叫做重请求延迟（*re-request delay*）并重新开始。每次重发定时器终止的时候，就继续请求这个丢失的包，直到丢失的包到了为止。

“自动重复重发请求”的使用意味着请求本身不需要作为可靠的消息发送。另外，请求一个丢失包之前的初始延迟使得任何顺序颠倒的包都能到达，因此减少了错误地发送重发请求的机会。顺序颠倒延迟的典型值是大约 1/10 秒，重请求延迟的典型值是大约 1.5 秒。

4. 应答

因为客户端现在负责检测丢失包，并请求重发包，就不再需要应答快速地发送。另外，顺序正确的包处理使得我们可以应答最近处理的可靠包。从那样的一个应答中，主机可以推测出这个应答包之前发送的每个可靠包都应答了。

这样的最终结果就是可以以低得多的速率发送少得多的应答包。另外，应答包不再需要是可靠的，因为如果一个丢失了，下一个到达的就会表示应答了所有的包，包括丢失的那个。因此应答可以以一个常速率来发送，不管互相之间有多少可靠的消息到达。这样的一个系统中，应答频率的一个典型值是每两秒一个。

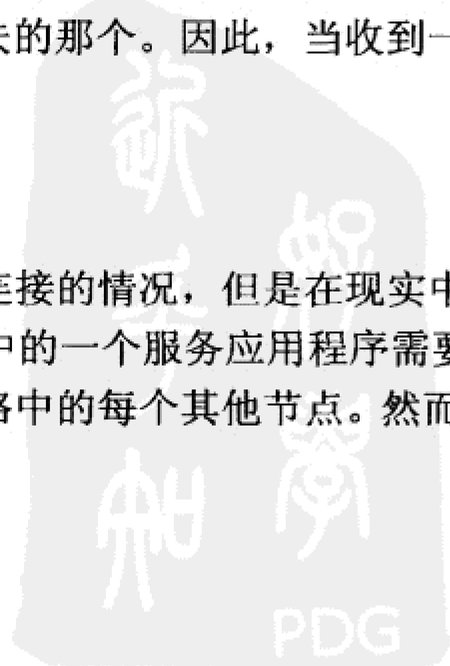
5. 可靠包队列

主机必须记录下发送的每个可靠包，直到它被客户端应答了之后。这通过可靠包队列来完成。当每个可靠包发送的时候，它就被放入队列。当一个应答从客户端到达的时候，队列中任何标识符小于或等于应答包的包都会被删除。

如果从客户端收到了一个重发的请求，首先要在队列中定位那个可靠包，然后重发给客户端。重发请求也是对前面所有包的一个隐式应答；为了检测这个丢失的包，客户端就必须处理前面所有的包，直到丢失的那个。因此，当收到一个重发请求的时候，被隐式应答的包就可以从队列中删除了。

6. 多重连接

到此为止，只看了单个连接的情况，但是在现实中，可能需要允许多重连接到单个机器上。例如，在客户/服务系统中的一个服务应用程序需要对每个客户有一个连接，对等系统中的每个节点都需要连接到网络中的每个其他节点。然而，客户/服务系统中的一个客户应用程



序可能只需要一个连接到服务器。

当需要多重连接的时候，每个连接都需要它自己的到达包队列和可靠包队列。另外，每个连接都需要分配它自己的包标识符，并保持跟踪下一个期望的到达可靠包，独立于其他连接。

7. 内存需求

在很多游戏中，特别是设计在控制台上运行的，内存非常紧俏，甚至在单玩家的游戏中也是。网络库可以使用的内存经常非常少。这对可靠消息来说很重要，它取决于跟踪未应答的可靠包，以使得它们可以重新发送。幸运的是，这个可靠消息协议的内存需求是适度的，而且如果内存非常紧俏，只要用少量的消息就能节省大量内存。

这个系统中的大部分内存都用于存储可靠包的元素和到达包队列。需要有足够的内存让可靠包队列存储几秒钟内发出的所有可靠包，长度要足够应答。需要记录可靠包的准确时间是不可预测的，而且完全没有限制，然而，可以安全地假设是大约二到三乘上应答频率的值。这就是说可以安全地去掉一条应答消息，没有队列溢出的危险。如果确实花光的发出可靠包队列的空间，那么游戏就可以不再传送可靠的包，因为没有地方存储它们，让我们可以在必要的时候重新传送。

同样，到达包队列需要有足够的元素来存放所有到达的包，它们可能在等待重发包的时候到达。然而，一旦到达包队列满了之后，丢弃不能立刻处理的到达包就是安全的了；它们以后总是可以重新请求的。

当使用多个连接的时候，内存需求量随着连接的数目线性地增长。然而，可以在所有连接之间共享这个队列使用的内存；除非是在极端的网络条件下，否则不会出现所有连接同时都需要大量队列空间的情况。

虽然在一些情况下，仍然会被认为占用了过多的内存。如果需要进一步减少内存耗用，可以选择减少存储在队列中包的数量和大小，虽然需要做的测量可能有些严峻。我们知道没有标记为可靠的包和消息的到达对应用程序的正确运行不是很重要。因此，可以选择不把这些包和消息存储到队列中。

工作的方式很简单；当把一个可靠包存储到发出包队列的时候，从中删除掉所有本身不可靠的包。这保证了只有这些包中最重要的部分才会被放到队列中，为了应付可能的重发，减少内存的使用。在到来包队列中，可以选择不保存不能立即处理的不可靠包，而且可以串联确实需要队列化的可靠包中的不可靠元素。

这样的包和消息串联可能是不受欢迎的，因为包丢失的时间对用户太明显了。这会出现在每次客户端都得等待丢失包重发的时候，多个包含了比如像可见敌人的位置更新这类东西的不可靠包将会丢失。可以稍微改进这个效果，通过选择在队列变满时候的实现。当队列为空的时候，所有消息将被存储，但如果队列超过了某个大小，网络库就应该检查和裁减掉队列中不可靠的包和消息。这减少了包丢失的现象，除非在严格的情况下。

6.6.5 总结

我们已经看到了可靠消息是网络库中的一个重要部分，以及传统实现可靠消息的方法有

些什么问题。我们知道了通过把检测丢失可靠包的任务移到客户端，就可以克服这些问题，并产生一个鲁棒的、可靠的消息系统。最后，我们检查了这个系统的内存需求，并学习了减少内存使用的方式，特别是在极端的包丢失的时候。

6.6.6　延伸阅读

有兴趣学习更多有关使用 TCP 和 UDP 协议进行网络编程的人，可以在下面的书中找到信息：

[Donahoo01] Donahoo, Michael J. and Kenneth L. Calvert. *TCP/IP Sockets in C: Practical Guide for Programmers*. Morgan Kaufmann Publishers, 2001.

[Napper97] Napper, Lewis. *Winsock 2.0*. John Wiley & Sons, Inc, 1997.

6.7 安全的随机数系统

Shekhar Dhupelia
sdhupelia@gmail.com

从一个高层次的观点来看，一般有两种方法来实现一个网络动作游戏。第一个是同步的方法，或者“步调一致”的方法，其中输入向后和向前传递，使游戏尽量地同步。第二种方法是异步的方法。其中每个端都尽量保持同步，但在同步之间，玩家暂时可以自由去做任何他们想做的事情，程序试着预测和延迟掩蔽来隐藏各个机器之间的不同。

实际上，同步模式对两台机器之间的任何不同极其敏感。而一些附件在机器与机器之间可能不同，所有机器上的任何会影响游戏的东西都必须保证完全一样。这种方法的一个很大的绊脚石是随机数。随机数可能用于游戏中的多种任务，从人工智能行为到选择音效。开发一个同步游戏的第一步中的一个内容应该是实现一个随机数系统，它在应用程序的控制下，而且在网络游戏中是安全的。

本文描述了一个安全随机数系统的架构，按本文的步骤，你可以在后续的调试中节省很多时间，了解如何用这类系统来取代现在的即时重放（Instant Replay）系统。

6.7.1 随机数影响在线游戏

当游戏代码调用标准 C 的 `rand()` 和 `srand()` 函数的时候，返回的值不是真的随机。相反，它们使用了一个伪随机数生成算法，一个良好定义的标准过程。

如果两个机器连接成在线游戏，而且网络游戏是两个应用程序启动时做的第一件事情，伪随机数返回就可能相同。然而，如果一个应用程序已经在前面，在离线会话中，使用相同的随机性玩了一阵子（在 AI、音频解说等），连接的机器将开始得到不同的结果。

一旦机器之间开始得到不同的随机数结果，然后就会发生“脱离同步”或失去同步。本来，一个随机选择人物应该向左或向右移动可能会造成不同的结果，游戏就不再处于一个相同的状态。结果会快速地分叉，快速地导致游戏的不可继续，特别是在确定性的（步调一致）游戏中，这些游戏主要由玩家输入来控制。

简单化这个问题方法的是随机数生成的算法本性。因为这些算法的大部分变体都依赖于让某些计算基于前面生成的数（“种子”），然后这个数在

机器之间传递。然后，一旦所有连接的机器使用同样的种子值开始游戏了，它们的随机数生成器将总是返回给应用程序相同的结果，使得它们虽然独立于其他的，但能保持相同的状态。

跟踪这个随机数发生器，以及直接控制这个种子值的关键是把随机数从标准的 C/C++库移到应用程序控制的代码中。后面讨论的一个类将完成这件事情。

6.7.2 网络模型

当把这个随机池类应用到网络游戏的时候，任意数量的机器都可以连接并使用这个系统。在 32 个玩家的游戏中，仍然需要确定什么时候是做出全部决定的开始点，以及每个池中应该用什么种子（这在后面会进一步讨论）。

为了简化起见，本文假设网络游戏中只有两台机器。然而，不管应用程序是在 2 或 20 台机器，只有 1 台机器可以指明为会话主机。这台会话主机的随机种子值将在会话开始的时候，在游戏中轮询和同步。必须决定选择一个游戏/安排逻辑的会话主机。

6.7.3 随机数池

虽然在连接的机器之间同步随机数可能解决很多失去同步的问题，这确实也会造成其他问题。实际上，一些游戏子系统可能真的对这个系统不利。

其中一个例子可能是音频解说系统。在体育游戏中，出现像进球这样的事件可能需要播放一段音频提示。另外，为了真实性，对于同一个事件，游戏都可能有一组的音频提示以供选择。选择提示有理由是一个随机事件。现在，如果音频剪辑常驻于内存中，就可以播放相同的解说，开始和结束都是同时的，而且继续同步地游戏。然而，如果音频剪辑必须从物理硬盘上实时地流式播放，那该怎么办？在这种情况下，一个系统的载入音频剪辑可能在其他系统之前完成，所以播放将会不同。虽然这也没问题，但在硬盘或 DVD 较慢的机器上，仍然有可能造成音频剪辑随着时间的“倒退”，最终导致游戏互相失去同步或不能再继续玩下去了。

但是进一步看看这里，程序员可能发现没有理由让音频解说保持同步。随着音频在游戏的过程中流式播放，不需要等待它们结束，两个连接的机器可以似真地播放完全不同的解说，而保持所有基本的游戏系统在同步的状态。

在这种情况下，AI 的随机性可能需要同步，但是音频的随机性要么不相关，要么有目的地不同步。为了完成这一点，不是开发单个随机种子和生成器，而是让应用程序可以从多个随机数的“池”中取数。

从随机数池中取数在概念上相似于在内存管理系统中从多个内存槽（memory bank）取内存。在分区的内存槽系统中，可能增加一些代码，用来调试或记录一个槽的使用并避开其余的。也可能有代码用来在运行期建立和销毁一些槽，而其他段则保持独立。虽然每个游戏可能有他们自己的需求，但这里有一组池的范例：

```
enum POOL_TYPE
{
```

```
    POOL_DEFAULT            = 0,
    POOL_ENVIRONMENT        = 1,
    POOL_AI                 = 2,
    POOL_COMMENTARY         = 3,
    POOL_MUSIC              = 4,
    POOL_MAX_TYPES
};
```

现在，为了简化，种子只不过是上一次产生的随机数。但是必须要有一个起点，而在标准C/C++中，这通过调用 srand() 来完成。一般来说，这个种子在内部存储为一个 4 字节的 int 或 uint 变量。但是因为有多个随机数的池，用于不同的游戏子系统，所以数据存储必须符合：

```
static unsigned int randPools[POOL_MAX_TYPES];
```

现在，如果应用程序总是需要获取一个池的种子，它只要根据池的枚举就可以得到：

```
unsigned int getPoolSeed(POOL_TYPE whichPool)
{
    return randPools[whichPool];
}
```

另外，当会话主机把这个随机种子广播到其他连接的机器时，它们可以依次用类似的方式重置本地的种子值：

```
void setPoolSeed(POOL_TYPE whichPool, unsigned int newSeed)
{
    // 确定种子不是 0
    if (newSeed != 0)
    {
            randPools[whichPool] = newSeed;
    }
}
```

这个时候，随机数发生器将根据这个新的随机值来执行，丢弃任何前面的种子。

6.7.4 随机数发生器

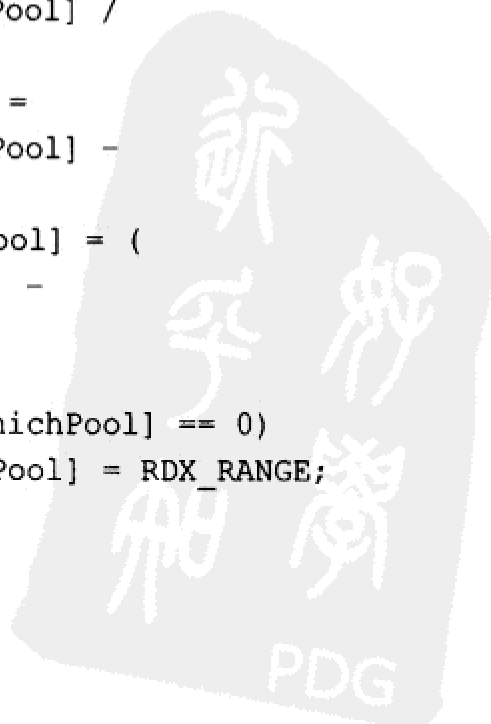
现在，在第一次使用随机数发生器之前，系统需要应用程序为池提供一个初始种子值。另外，系统不允许任何应用程序代码重新置入初始值。甚至，应用程序在完成初始种子后应该强制使用前面描述的 setPoolSeed() 函数。这里是一个例子，用于第一次使用池的置种和防止后面的置种：

```
static bool seeded = false;
void seedPools(unsigned int seedValue)
{
    // 确定种子不是 0
    if (seedValue == 0)
    {
            assert(0);
```

```
            return;
    }

    // 确定我们对每个池只置种一次
    if (seeded == true)
    {
            assert(0);
            return;
    }

    // 循环过随机数池
    for (unsigned short pool = 0;
            pool < POOL_MAX_TYPES;
            pool++)
    {
              // 初始化随机池
    randPools[pool] = seedValue;
    }

    // 设置我们的置种标志
    seeded = true;
}
```

最后，系统只剩下真的随机数生成器了。Internet 上可以找到很多地方，提供了不同算法的源代码。其中一些专注于速度，而一些更专注于随机性。另外，一些只提供 ANSI C rand() 函数本身的 C 代码实现。这里的选择取决于各个应用程序，但是对于这个例子，使用了 Randomal64 生成器[Thomas01]。记住，这可能对大部分应用程序都是很好的，如果不是，可以容易地替换掉它的内容。

```
static const unsigned long RDX_RANGE = 0x7FFFFFFF;
static const unsigned long RDX_CONST = 0x00000000000041A7;
static const unsigned long RDX_Q = RDX_RANGE / RDX_CONST;
static const unsigned long RDX_R = RDX_RANGE % RDX_CONST;

unsigned int getRandomNumber(POOL_TYPE whichPool)
{
    // 返回在 0 & range 值之间的随机 integer32
    unsigned long hi =
     randPools[whichPool] /
     RDX_Q;
    unsigned long lo =
     randPools[whichPool] -
     (hi * RDX_Q);
    randPools[whichPool] = (
     (RDX_CONST * lo) -
     (RDX_R * hi)
     );
     if (randPools[whichPool] == 0)
     randPools[whichPool] = RDX_RANGE;
```



```
        int rslt = int(randPools[whichPool]);
        if (rslt < 0) rslt = -rslt;
              return rslt;

        return 0;
}
```

注意这个生成器把每一轮的结果保存回池数组中，而这个值会在下一次中使用。

6.7.5 重载标准的 rand()和 srand()

把代码都集合起来，直到形成完整的随机数池系统，准备好在程序中使用。然而，当它放入程序之后，在线程序员仍可能发现一个新问题：很多程序员不知道新的随机池系统，或只是忘记或不想麻烦去用它。不幸的是，随机数可能用在游戏代码中的任何地方，网络程序员一般不想修改那么大的面积。游戏中某处的一次错误的 rand()调用都可能破坏在线游戏。

这个问题有一个简单的解决方案。C/C++允许用宏来定义重载标准函数调用；当在编译和连接期解析符号和#define 的时候，编译器将选择重载的函数调用，而不是标准的函数调用。

重载标准 C 的 rand()调用只需要：

```
#define rand() Pools::getRandomNumber(POOL_DEFAULT)
```

同时，一个追捕到工程师错误使用 rand()和 srand()调用的好方法是也重载所有到标准 srand()的调用：

```
#define srand(x) Pools::seedPools(x)
```

保证随机池的头文件在编译/连接层级中可以通过#include 得到，或列在了预编译头文件中，经常是传播这些宏的方法。

6.7.6 下一步：记录和调试

一旦随机池系统就位之后，它自己就变成了应对在线游戏问题的强大调试工具。在确定性游戏中尤为如此，那里的游戏状态在会话的过程中必须保持完全一致。

用来找到机器之间失去同步问题的一个好方法是比较每个系统中的随机池使用。实现 getRandomNumber()函数的调用记录很容易（取决于平台），比较来自每台机器的输出有助于隔离出一些“脱离同步”的程序代码问题。

6.7.7 下一步：即时重放

在只是提供额外的记录和调试之上，对于一些游戏，随机池类实际上可以构成即时重放功能的基础！不用每一帧都录制游戏的状态，只要记录调用 getRandomNumber()的结果和控制器/键盘/鼠标输入。这个数据应该以精确连续的顺序来存储。然后只要把相同的输入反

馈给工作中的不同子系统，这些值就可以用于基本的“重放”任何给定点的一个帧。虽然其他系统不知道发生了什么，但却基本上重复了相同的工作，这个数据集很小而且很经济。这甚至对很多离线游戏都有好处。

6.7.8　总结

本文讨论了一个随机数池系统的多种使用。其中主要的是网络游戏可以同步的起源；这保证了请求一个随机数在每台机器上都返回相同的结果。另外，应用程序控制的随机数使得程序员可以不断地重放各种子系统，不管是调试工具还是游戏中的真实功能。

6.7.9　参考文献

[Isensee01] Isensee, Pete. “Genuine Random Number Generation.” *Game Programming Gems 2*. Charles River Media, 2001.

[Thomas01] Thomas, Andy. “Randomal64 Pseudo Random Number Generator.” Available online at *http://www.generation5.org/content/2001/ga01.asp*.

6.8 安全的设计

Adam Martin, Grex 游戏公司
gpg@grexengine.com

一个系统的稳定性不会比它最薄弱环节的稳定性强多少，所以说单单在此系统的某几个方面花大量的时间和金钱通常来说是一种浪费。你必须对每一个单独的连接都进行检查，并对它们在安全方面有足够信心。到现在仍然有大部分的项目负责人甚至连他们有什么连接都不知道，也不理会如何评估连接的安全。

评价安全的能力对于任何多人游戏和在线游戏都是相当关键的，然而现在的软件工程师却喜欢在游戏的完成阶段和发布测试后的阶段来改进安全问题，而不是在设计阶段。我们需要在项目最开始的阶段就把安全性加入到游戏当中。

本篇文章将讲述如何将一个分成多个阶段的过程方便地加入到现有项目中。在理想情况下，你可以在项目的开始就根据这个流程来做，但是它对于已经开始的项目仍然十分有效。这篇文章以及例子特别关注服务器的安全，但是也同样适用于客户端的应用程序。

我们需要以一个不安全的系统为基础进行完善，它的定义通常是从目前的威胁模型（TM）发出一个或者更多的攻击但没有被目前的安全策略（SP）所处理。如果我们构建一个完整的 TM 并充分地遵守 SP，就能够通过简单的阅读这两个文档来计算出系统到底有多安全。这就是这篇文章的主要目标。

TM 和 SP 一起充分地验证了系统的脆弱性，它的解决方案和足够的信息可以用来快速地重建和重新评估最初的假设和结论。

6.8.1 安全问题真的如此重要么?

安全缺陷像软件 bug 一样，在修复它们的耗费往往是根据发现它的时间长短成指数地增长。但是，安全缺陷引起的问题甚至会比普通的 bug 更严重。例如，一个暴露所有客户的信用卡细节的缺陷，和在发布它之前就改正它相比，所造成的耗费是非常巨大的。

大多数的人通常把安全和加密，密码和认证策略联系起来，但是这些仅仅只是一些工具，而且它们本身并没有和安全结合起来。从全局的观念来讲，关于游戏安全性的实现现在主要有三种方法。

- 一直等到发布，再检查有什么缺陷，然后尽可能快地拼命发布补丁。

- 委派第三方，并且希望他们害怕承担责任而设计出一个安全的系统。
- 对偶然的方式配置一个已知的策略，希望你可以应付过去并让它“破解起来稍微难一点”。

在几乎所有的情况下，在开发过程中过晚修复安全问题将会非常昂贵和困难。一个公认的常识是安全并不能逆向地添加，因为它太昂贵了；它需要从一开始就设计进来。如果开发过程没有在安全问题上考虑得很周到，你就有可能面临制作没有尽头的安全补丁的风险，而没有一个补丁能够完全解决这些问题，当玩家伤心到在你的游戏里抓狂时，就会毁了这个游戏和玩家，而且将会直接导致很坏的评价，并且减少销量。

从另一个方面说，也许你的游戏是一个大型多人在线游戏（MMOG），而且就像游戏世界里伤心的玩家那样，你的用户也将会成群地离开。更糟的是，这些漏洞中的一个有可能暴露你的客户的信用卡细节，造成财务和名誉上的双重损失，那就完全是另外一个级别的事情了。如果游戏服务器的安全性不够，你拥有或者控制的任何数据、硬件、带宽，甚至你的公司 ID，都可能被轻易地毁坏或是偷走。

幸运的是，本篇文章要描述的过程简单而又经济，而且在一开始就会有很多优势。每一个游戏开发者——从专业的工作室到独立的爱好者——都能够简单并且有效地实施这个过程。

6.8.2 目标

就像[Schneier03]所指出的那样，如果你不知道由谁实施安全保证工作，并且不知道这个工作被完成多久，那么“安全”基本上就没有什么意义。因此，第一个目标是形式化到底什么是我们要表达的。这是一个分项目执行的过程，没有一个适用于所有游戏的答案。

假定我们知道要试图保护什么，我们也需要一个方法来衡量成果。现在的开发人员习惯于使用通用的制度去评估代码改进在一个项目的开发阶段：BUG 列表，单元测试，可玩的演示等等。我们同样需要一个类似的精确的衡量方法来确定“安全度”。

那就会有一个成本的问题：如果每一个游戏项目拥有无限的时间和一个无穷的预算，它就能够简单地通过小心的编码以及广泛、严格的测试来达到目标。不过，假设资源非常有限，我们就需要能够分级别地决定把资源分配在哪里，也需要能够制定一个分级的“安全度”的标准并让它们之间互相比较。因此，另外一个目标是兼容性。

以上的两个目标都同样要求结果可以重现，不仅仅是最开始的测量者，其他人也需要得到同样的结果。我们不可能让整个过程都是可重复的，保障安全的一方面（发现对系统的攻击）天生就是创造性做工作而不是有规可循的。如果这种不确定能够严格的用一些方法来限制，那么它将非常有用。

重新估计整个系统的每一个分支同样是很昂贵的，并且还需要一些方法来限制我们的对比。理想的解决方案将是一些标准的测量方式，所以我们只需要重新测试那些被改过的部分就可以了。

6.8.3 术语

然而，安全工业在规格化和标准化上都滞后于软件工程行业。不能更好地做好规格化，那就将延续那些自称专家的人去让你修补一些最有可能成为隐患的安全漏洞；在这种环境下，真的很难知道到底谁才是这方面的权威。工业标准化推广的足够广，大多数的术语才能够有一个严格

PDG

的定义。这篇文章会遵从大部分的主流术语，但还是有很多情况下没有一个明确的定义。

特别是“安全策略”，在很多资料中都被定义成自己的意思。这篇文章用到的将是大多数通用版本的一个扩展版本。一些资料很主观地蔑视那些基本定义，并宣称“策略并不是技术手册”。但 CIO 在运营一个大的企业的时候，他们会使用一种他们解释为很重要的文档，主要是作为给非技术人员每天的规章制度来贴在他们的工位上。很不幸，它经常会被认为是安全策略，导致引起很多误解，很大一部分员工认为他们已经把安全保卫工作做得很好了。其实这是另外一种关于安全保证的有力工具，但是它并不能提供我们所谓的安全策略的可重复性和完整性。

6.8.4 威胁模型：测试不安全性

[Berg02]提到第一个步是确定一个方法来量化和评价风险。我们从模拟内在的风险，也就是威胁模型（TM）开始。然后将使用它来产生安全策略（SP）,也就是解释如何处理这些威胁。

最基本的威胁模型就是一个简单的关于黑客可以攻击的系统规则的列表。它们之间的大多数都有很多共同点，但是都需要一个一个地写下来，所以每一个威胁都是独立的。这样就能保证以后的维护将会非常快捷：威胁模型不需要重写，我们就可以加入或删除其中的条目。

随意地进行系统安全建设是一个很不严密的工作：我们确实只想说，“确信没什么坏事可能发生，包括所有我们没有想到的而可能发生的坏事”，然后继续工作就可以了。如果真的如此天真地把这个概念应用到我们的需求文档和规格文档中的安全部分的话，那么需求文档将只会有一行，当然，规格文档也是同样的长度。然而我们就没有办法在如何达到可重复和可信的安全系统上表达我们的意图，如何去做并且我们已经做的。

威胁模型基本上等同于一个需求文档。就像需求文档描述游戏该做什么，威胁模型与之对应的是黑客将尝试对游戏做些什么。这个阶段是一次完整的创意锻炼，完全值得你叫上所有的团队成员来开展一次头脑风暴，想象一下任何有可能的攻击。没有任何从黑客观点上来说合理的攻击会被拒绝；不管这些攻击（威胁）是否对系统构成危险，但是他们都必须在这个阶段记录下来，以便在后面的阶段中把它们列入注意事项。

在这个阶段拒绝攻击会产生两个主要的问题。首先，今天能使一个攻击无效的设想可能明天就会改变；如果这个攻击根本没有在 TM 列表中，那么在进一步对游戏安全的估计中将可能不会注意到这个没激活威胁的危险。而如果它已经在接下来的阶段中涉及，那么我们将可以很容易地发现它的危险并给出适当的反应。

其次，如果你拒绝了攻击，那么事实上你并没有做攻击评估这件事。如果将来有其他人遇到一样的攻击的话，就必须要重新评估，即使有一些人现在能够记住它已经被评估而且被拒绝，因为他们也有可能会忘掉。

1. 威胁

每一个在威胁模型中的元素都是一个黑客有可能发起攻击的描述。每一个攻击的描述都要尽量的具体，而不要太抽象；比如说，它需要包含关于这个攻击到底是怎样的并且黑客将使用什么工具这样的具体的细节。它也非常有助于从黑客的观点记录他们攻击的动机，因此将来的读者就会很清楚每一个威胁是怎样添加到模型中，而且那样还起到了一个源代码文档的一样的角色。如果威胁模型过大的话，为了保证文档的可读性和简洁性，通常把这些信息

PDG

都放到附录中。

举例来说，一个对需要点卡的多人在线游戏的正确的攻击应该包括：

- 尝试偷盗其他玩家的角色；得到其他用户的密码并且用偷取的用户登录游戏；修改密码，信用卡（CC）细节，家庭住址，E-mail 地址，等等；长期的占用盗取的账号，但尽量让它看起来像是一个合法、正常的账户。
- 看看自己是不是足够幸运，在攻击怪物的时候会不会太困难。如果不够幸运的话，便使用作弊来避免死亡；或者试图在即将死亡的时候逃离战场。
- 如果并不能在游戏中逃脱，便使用中断游戏的方法来强制脱离战斗。尝试退出游戏。如果这个时候禁止从游戏中退出，那么就采用切断网络连接甚至重启计算机的方式。或者，在极端的情况下，尝试攻击服务器，让服务器崩溃，当服务器重启的时候，它会重置你的角色的位置，同时，你将不再处于危险状态。

这些都必须提炼出来；对于一个真正的多人在线游戏来说，你将面对比上述问题更细碎的问题！

以上每一个论述中，都包含了攻击者的意图。以下是三个很重要的理由。第一，要把可能遇到的攻击按照优先次序排列，这些信息能够帮助我们估计哪些攻击最有可能会发生。

第二，了解到黑客攻击的意图能够让你发现更多你一时想不到的攻击方式。当你召集所有的团队成员对可能发生的攻击方式进行头脑风暴时，由一个人来提出关于攻击动机的描述很有可能能够激发其他人关于类似攻击的更多的想法。像偷取账号这样的攻击最初是在组员门集中讨论“如果可能，你会在游戏中会做的事情”这样的问题时提出来的，同样像偷取其他人的装备也是这样讨论出来的。采用这样的方式，我们就能引出各种不同的可能的攻击方式。

第三，如果不理解最终的目标，那么对攻击的描述就会看起来根本不合理：你能够很容易创建自己的账号，为什么还要取偷取别人的账号并为之充值呢？偷取账号而不去修改信用卡信息的动机是完全不一样的，其目的是免费地进行游戏，一定不要把它和这种攻击的其他动机搞乱。错误的理解和记录这个差异，就有可能导致你是在假定信用卡细节是一直保持正确的情况下而实现你的“方案”。而在攻击真正到来时，同样会有很多不同的情况，这个假设是完全没有用的。甚至情况有可能更糟糕，以后的威胁模型的维护人员可能会觉得它是一个错误的威胁并删掉它！

2. 成本

看起来好像整理 TM 需要很多时间和努力。在实际操作中，它们往往能够很快地提出来，主要是因为每一个威胁基本上互相之间都没有联系，而且它们能够以清单的方式记录。通常情况下，大部分的 TM 都能够在几个小时内集中讨论出来。没有必要让整个团队开一个大型会议而仅仅去讨论生成 TM，只需要分成小组在休息的时候讨论，项目经理只需要查看并且汇总这些讨论结果。因为每一个元素都是与完全无关的，所以汇总工作通常都非常简单，而且不会产生任何冲突。

3. 结构化威胁模型

即使正在做一个简单的游戏，也能在很多单独的攻击方式中快速地生成一个 TM，而且

TM 的表示常常有必要利用结构化的表。简单的分类没有什么作用，主要是因为大多数攻击方式都很难严格地归到一个种类中。

很多人更多地使用攻击树（Attack Tree）[Schneier99]结构来构造他们的威胁模型。一棵攻击树通常都是以黑客特定的目标或者动机为起点，每一个树节点都是一些让他们能够达到最终目标的一些操作。你很快就能构建出一棵树，因为大部分的操作都能从多种途径达到目标，并且因此你需要把主要操作的多种子操作都列出来。

使用攻击树主要的优势有以下几点。

- 是一个记录攻击信息，剔除多种攻击方式的重复的公共部分的一个非常有效的途径。
- 在提供了良好结构的前提下，还保证了大部分攻击条目的无关性。
- 在如何组织攻击条目的问题上，得到的结果通常都不会有太多的争议。

并且，树型结构本身也同时记录各种攻击之间的联系的额外信息。它也使得威胁模型非常易读，因此你就能快速地找到特定组或者类别的攻击。它也同样提供了一种对细节抽象的模式，使用它你就有可能仅仅阅读每棵攻击树的最上面几层，而并不需要详细地遍历树的每一个节点就能得到一个攻击方式的总体看法。

其主要缺点在于它怂恿了限制性的思维方式，这将导致我们更有可能遗漏一些攻击方式。有一个通用的方法是把那些难于处理的清单做一次或者多次的迭代，然后把它们转化成攻击树的方式（比如说，你在不同组中同时把 TM 都做一次合并）。

还有一个重要的问题是如何用电子文档的方式来编辑这样一个庞大的树型结构。有少数团队使用了一些像 WORD 这样普通的工具来编辑树型结构。对于复杂的系统来说，结构化大型的威胁模型，使用这个方法带来的实惠远远大于使用它所带来的不便。如果文档无论从阅读还是修改方面都难到让人发疯的地步，人们就会放弃这个文档，使用和更新文档的频率就会降低。从根本上来说，这会给整个过程带来毁灭性地打击，所以针对所有团队维护一个简单易用的表是非常重要的。

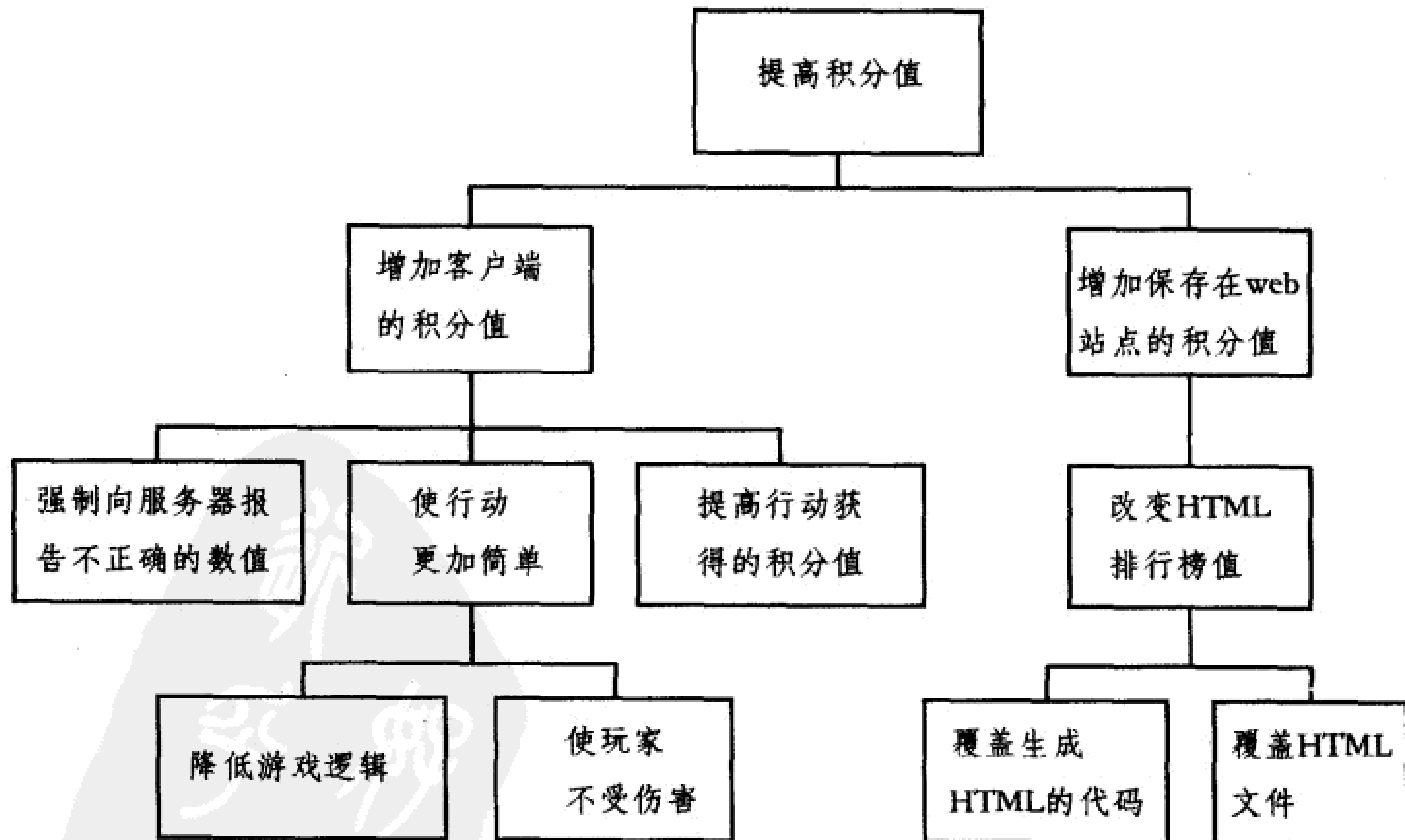


图 6.8.1 攻击树的一例子：1 个根节点，3 个子节点，1～5 个子小节点

一个优秀系统的整个威胁模型通常是由多个无关的攻击树构成。实际上也不需要严格地把所有的攻击都放到一棵攻击树中（这经常需要一个强制性的对攻击描述）。

6.8.5 安全策略：让威胁无效

安全策略是从威胁模型中继承过来的。它主要是描述如何消除每一个潜在的威胁，从而保证系统的安全。没有与之对应的威胁模型，那么安全策略基本上就是没有什么作用，从而变成了一个存在没有严谨定义问题的解决方案，不再知道该如何回答“我该保证谁的安全？”这个问题了。[Schneier03]

和威胁模型不一样，安全策略主要的目标在于指出什么是这个系统和它的管理员必须要做的。它描述了他们要做什么，怎么做，还有在什么地方适用，甚至他们为什么那样做。安全策略必须是精确的，一个不精确的安全策略会造成混淆，或者，在各种安全元素之间造成漏洞，以后将有可能被攻击者发现并且使用。

没有一个安全策略，没人能够在不定义他们该做什么的情况下去采取任何行动维护系统的安全。当发现攻击的时候，他们可能独立地推断、推论或者一时兴起地自己建立一个简单的安全模型来应付。安全策略应该被看做一个策略的指南并且为了发布要被十分详细地制定出来。当安全策略规定好以后，那么就是假设所有可能发生的问题都被估计到了，并且如果安全策略说要做什么，那么就肯定有一个切实的理由需要这么做。

重要的是，安全策略需要经常从概念设计开始重新编制，而且，在编制时要注意保持一致性。这就是它最核心的优势：可重复性，并且系统、科学地针对了随机和突发的攻击。应该将安全性作为一个程序开发纪律重视起来，而不是敷衍了事。

我们都知道它是可重复的，因为安全策略是有系统地从威胁模型中转化而来的。安全策略可以只在威胁模型改变的时候跟着改变。（实际上安全策略总是需要随着威胁模型改变而发生改变，至少也需要根据威胁模型的改变而重新评估。）

根据威胁模型生成安全策略的过程是非常简单的：确定好每一个威胁并确保当前的安全策略能够使之无效。如果没有，那就把它加到安全策略中，直到它能够被你的策略屏蔽。越往后，你就会经常发现有些威胁已经被现有安全策略中的一些项所屏蔽，根本不再需要额外的处理。

如果一个威胁被认为处理起来太复杂，或者未必需要担心，那么安全策略都需要精确的说明并给出理由。所有这些部分都是在确保安全策略系统，有逻辑地扩展威胁模型，并保留原有的可重复性和可读性。我们通过精确的处理威胁模型中的每一个元素，来生成一个单独的安全策略的命令文档，而不再需要参考威胁模型文档，这就使得工作人员可以直接使用安全策略文档来进行工作。如果在工作中只使用一个简单并且经过深思熟虑的文档的话，就能够提高基于安全策略的决策被所有的员工严格的执行的可能性。

6.8.6 同时修订两个文档

这一过程的最大价值就是快速实现了迭代生成威胁模型和安全策略的第一步。如果没有有规则地修订它们，你将理所当然地会错过一些优势。在这种情况下，同时的修订不会进行，

但它仍然值得按照正确的方法重新开始。

优先于其他工作，两个文档应该要按照规则重新编写。而根据任何得手的攻击再进行修订也是合乎情理的。在这两种情况下，除了不需要从概要设计重新开始以外，它们都应该执行一个重复的过程。然后安全策略也应该被测试，至少需要根据新的攻击，尽管最完美的是所有的安全策略都测试一遍。在相同的情况下，两个文档都应该同时更新，并且需要保持它们永远都是同步的（假定有一个文档版本系统）。

6.8.7 该技术的其他优势

接下来简单介绍该技术的其余优势。

1. 基于代码，容易上手

对于一个新程序员来说，这个技巧能够很快让他了解游戏是如何写成的，或者是游戏中的代码是如何相互适应的。这个技巧能够针对你的代码基础进行针对模块的分析，并且能够在用例级别上发挥作用，因为威胁模型本身就是以一系列用例为形式的。对于组外人员或者是新进成员来说，这个方法比任何文档都易于理解。

2. 能够从一个全新的角度分析现行程序设计

安全性审计能够让人们从一个全新的角度去分析系统或是他们所写的代码，这样做能够在发现系统的安全隐患之前，预先发现很多系统中和安全性无关的 bug。另外，这个技巧也可以称为“呼吸新鲜空气”，能够以一个从心理角度出发的不同于标准流程的方法来搜寻 bug。

3. 对于经验的积累有好处

为了项目开发的威胁模型和安全性政策也有可能为其他工作的起点。他们可以作为一个帮助小组成员在其他工作中引入头脑风暴的起点。通过这个技巧，团队能有机会建立一套随着时间更新的威胁模型的资料库，资料库中应该有关于每个威胁的细节描述，这样可以为其他团队成员在未来的工作中提供参考。这一个资料库应该和其他的软件资料库具有同等价值，需要分发给相关人员，并保持更新，并且有对应的评估机制等。

6.8.8 延伸阅读

Common Criteria(CC)-[Cox00,CCEVS04]相当于是这个技巧被行业标准化的一个结果。CC 标准又是来自于 TCSEC 和 ITSEC 编程标准(分别是美国和欧洲的标准的两个不同的名字，代表更高级标准）中的一些通用元素。CC 在架构上比我们讨论的技巧要大一些，它的目的是提供更高级的安全保证和更为精确的评估标准。作为一个实践的例子，可以参考一下 CC 对于 Windows 2000 的安全性评估[Microsoft02]。CC 有三个核心组成部分：保护概念图，评估目标和安全目标。我们这个技巧中所使用的威胁模型其实就是一个简化版的 CC 保护概念图，以及用一个增强了的安全政策来代替 CC 中的安全目标。

另外，Bruce Schneier[Schneier00,Schneier03]的研究成果也是在安全领域被很多前辈专家

PDG

所认同的。他的书不论是在理论上和实践上都很有深度。Ross Anderson[Anderson01]是另外一个备受尊敬的专家，他负责维护的网站[Anderson04]专门负责处理安全问题，从隐私保护到数据保护，甚至于密码学，和一些很深奥（或者说很有趣）的密码系统。

6.8.9 总结

现在你已经有了一个在开发初期就为游戏建立一个安全保护的方法，而且不会增加研发周期。另外，本技巧不需要专门的知识，你的每一个组员都可以对整个安全性进行建设，并且能够理解系统是为何以及如何进行安全保护的。

这个技巧没有涵盖任何关于安全性的执行问题，也就是所谓的对密码学等专业的理解。诚然，这些问题对系统的安全性来说仍然是至关重要的，但是通过应用这个技巧，那些密码学上的问题可以被看成是最后的设计阶段以及具体的执行问题，在你的主要设计阶段就可以完全不用考虑他们了。

例如，当一个安全政策需要用在某一个点上用到加密的字串的时候，你必须要决定应该使用哪种密码协议，不过这个决定其实并不好做。不过，你做这个决定所必需的一些信息早就包含在了威胁模型和安全政策里了，所以说在做这样一个决定其实是水到渠成的。正所谓“菜鸟拿起一本密码字典匆匆上路，但是专家直到最后一刻才开始动手” [Clayton04]。

我们在这个技巧中所使用的一些方法使你可以从一个很集中和很系统的角度来看待安全问题。你也可以从纪律并且科学的方式方法中获得好处——这种方法就是 Anderson 和其他人所说的“安全工程学”(在这里不得不提到安全工程学以及安全纪律对软件开发的好处)。尽管我们所讨论的技巧只是一个孤单的流程，但是通过不懈的努力，你就会使整个系统被保护起来，并且通过经验，你也能够知道安全的程度如何。没有所谓的敷衍，也不用插着手指去猜“可能是安全的。[画外音：我希望是]”。你还有了一个针对安全设计的规格说明书可以用来检查你的执行情况。

6.8.10 参考文献

[Anderson01] Anderson, Ross. *Security Engineering: A Guide to Building Dependable Distributed Systems*. Wiley, 2001.

[Anderson04] Anderson, Ross. Home page. Available online at *http://www.cl.cam.ac.uk/users/rja14/*, 2004.

[Berg02] Berg, Al, “6 Myths About Security Policies”, Available online at *http://infosecuritymag.techtarget.com/2002/ oct/securitypolicies.shtml*, October 2002.

[CCEVS04] “Common Criteria Evaluation and Validation Scheme”. Available online at *http://niap.nist.gov/cc-scheme/aboutus.html*, 2004.

[Clayton04] Clayton, Richard. Home page, Available online at *http://www.cl.cam.ac.uk/users/rnc1/*, 2004.

[Cox00] Cox, Peter. “Security Evaluation: The Common Criteria Certifications”. Available online at *http://www.itsecurity.com/papers/border.htm*, 2000.

[Microsoft02] Microsoft. "Windows 2000 Security Target", Available online at *http://download.microsoft.com/download/win2000srv/CCSecTar/2.0/NT5/EN-US/W2KCCST.pdf*, 2002.

[SANS89] SANS Institute. Available online at *http://www.sans.org*, since 1989.

[Schneier99] Schneier, Bruce. "Attack Trees; Modeling Security Threats". Available online at *http://www.schneier.com/paper-attacktrees-ddj-ft.html*, December 1999.

[Schneier00] Schneier, Bruce. *Secrets and Lies: Digital Security in a Networked World.* John Wiley and Sons, 2000.

[Schneier03] Schneier, Bruce. "A reality checklist for an effective security policy", Available online at *http://searchsecurity.techtarget.com/tip/0,289483,sid14_gci931792,00.html*, 2003.

第 章

7

音 频

引　言

Mark DeLoura
madsax@satori.org

音频已经存在很长时间了。我的一次商业音频编程经历是一个基于 PC 的 acrade 游戏，在那个游戏中，我们使用了 Gravis Ultrasound 声卡来对声音进行回放。我们对它感到非常得兴奋，因为 GUS 允许我们把声音信号采样下载到声卡后，以任何我们需要的声像、音量和音调进行播放。它同时也兼容 MIDI。它太酷了！我们在底层的 gfl 库上建立了一个声音管理层来自动实现在物理上对声音的处理。然后我们把音源绑定到场景中移动物体上，并相应地设置音量和声像来建立伪 3D 音效。我们弄乱采样率、比特深度、立体声和单声道采样，并尽我们最大的努力来优化在声卡上的内存映射，以此来满足 MIDI 设备的波形、引擎声音和环境声音。

我知道对于有些人来说，这使你们记起了那些玩弄 TRS-80 或者 Apple II 上的喇叭的日子，以及第一次使得机器发出一些类似音乐的声音的时候是多么的激动。幸运的是，对于现在的我们，艺术级的音频已经进步到让我们可以通过作曲家而不是编程来创建音乐了。我们可以很简单地把他们创建的东西从磁盘或者内存中进行流式播放。我们可以通过声音脚本工具创建豪华的实时声音环境，他们可以通过内置的 DSP 进行实时硬件效果处理并通过高保真的 5.1 声道的声音系统来进行回放。我们现在可以更加关注如何创建声音环境来增强游戏的体验而不是花时间简单地让机器发出哔哔剥剥的声音。

能出版五篇如此精彩的关于声音的文章是我们深刻的记忆。非常感谢 Sean Gugler 在开始的时候进行了本章的组织工作。在本章的开始，你将找到一篇讨论多线程和它在音频编程中的适应性的文章，它出自 James Boer。Matthew Harmon 将探讨如何编写声音 API，使得声音成组管理更加简单。然后 Sami Hamlaoui 会简要地描述一种简单的技术，它能让声音听起来是从一个 3D 表面发出来的而不是一个简单的点。Christian Schiller 深度挖掘了环境回声中的反馈延迟网络（Feedback Delay Networks）背后的数学原理。最后，随着语音识别在游戏中越来越突出，Julien Hamaide 处理了一种把输入语音和训练过的词典单词进行匹配的方法。

希望这些文章对大家能有帮助，而能让大家把其中某些技术应用到下一个游戏中！

7.1 多线程音频编程技巧

James Boer
author@boarslair.com

随着并行执行模式甚至是多处理器机器在桌面用户上越来越普遍，现代游戏硬件已经渐渐告别了单线程的执行模式。现代CPU，如Intel的Pentium 4 Xeon™（奔腾4至强）引入了超线程（Hyper Threading™）技术，它实际上是在一个芯片上模拟执行两个进程来获得内置并行能力的一种方法。另外，多处理器的机器在桌面上变得越来越普遍，用户也期望游戏能充分地使用他们花更多钱买来的硬件的所有长处。虽然本章的例子代码是针对Intel和AMD处理器，并且是基于Windows的，但是和多任务的概念相关的部分音频处理代码是可以应用到不同平台和操作系统的，包括应用到Macs、Linux和游戏机上。

我们将考察一个特定的平台，在这个平台上我们能使用 Windows 操作系统的多线程代码来体会多处理器或者超线程处理器的优点；主要的音频解码和回放的循环。通常的缺陷和多线程的问题都会被考虑到，并且还将讨论对多种常见的音频相关的算法的优化思路。无论如何，在本文出现的技术和技巧也能用到游戏开发的其他领域中。

7.1.1 多线程编程简介

简单地说，线程的编程涉及创建两个或者两个以上的可以同时执行的代码路径。这些线程都是一个单一进程（一个单一的执行程序）的部分，他们共享相同的地址空间。从外行的角度来看，这意味着从同一个进程派生出来的线程可以访问相同的数据。从本质上讲，每一个 C 或者 C++程序都可以看作是至少含有一个线程（主线程），它在程序启动的时候被调用。这个线程再负责为有限的或者特定的任务启动其他的子线程。图 7.1.1 演示了多处理器系统是如何工作的。

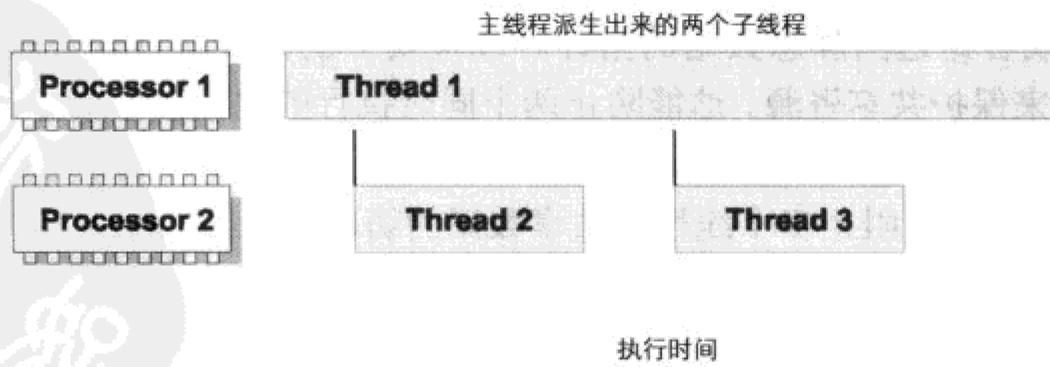


图 7.1.1 主线程在第二个处理器上派生出来的两个线程

很明显，只有一个处理器的机器实际上不能同时执行多段代码。相同的时间片轮转机制能允许多个程序或者进程在多任务操作系统上并行执行，也能允许一个进程里的多个线程同时执行。图 7.1.2 演示了在单处理器的机器上是如何模拟多任务的。

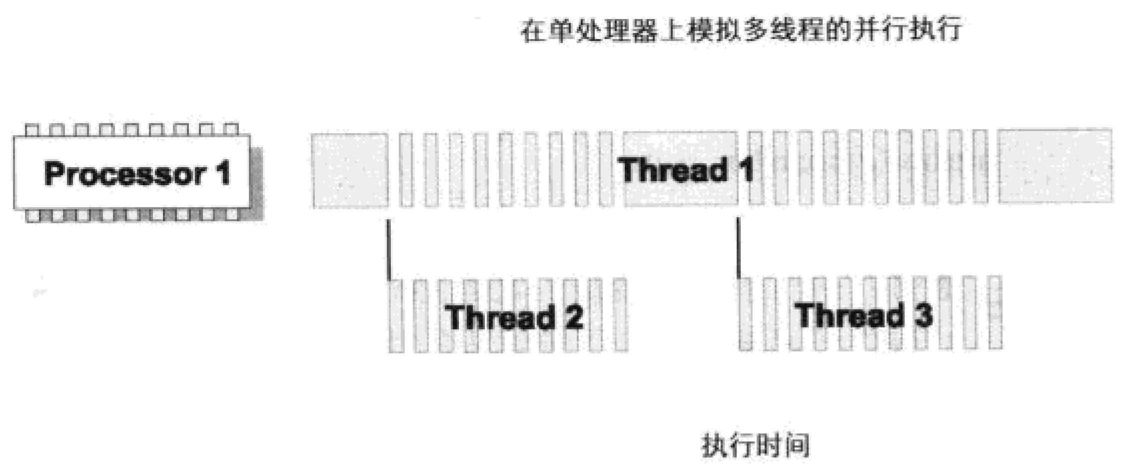


图 7.1.2 多线程在单个处理器上共享执行时间

幸运的是，对于一个操作系统来说，多线程的时间片轮转比多进程的效率要高。正因为如此，以下的做法是切实可行的：加入一定的多线程代码从多处理器系统（或者是能让多线程执行的更加有效率的单处理器）中能获益，而对那些只能通过模拟来获得这种机制的单处理器只会带来很少的开销。尽管如此，为所有的东西而不是最需要的任务都创建多线程是很愚蠢的行为。因为单处理器执行这些线程的开销以及管理和调试多线程代码的复杂性。

7.1.2 线程的术语和机制

多线程编程并不是仅仅涉及对传统线性编程以外的东西的思考。它还涉及一套完整的专用术语，其中很多将在这里讲到。

程序在一台机器上的运行实例被称为一个进程。进程的定义决定了进程的特性，它有自己的地址空间，能进行对磁盘以及其他硬件资源的访问保护。有些操作系统，比如 Windows、Mac OS X 或者 Linux 允许多个进程一起运行。其他的操作系统，比如游戏机上的操作系统，被优化成只能运行一个进程。每个进程（或者程序）会派生出一个或者多个当前运行线程。线程也许可以简单地认为是一个进程里的执行路径，它们都可以访问同样的地址空间（比如全局和静态数据）。

为了能安全地在多线程中同时使用函数（不管是自己的还是库函数），函数本身必须是可重入（reentrant）的和线程安全（thread-safe）的。可重入的函数没有为后续的调用保存静态数据或者是返回静态数据的指针。线程安全的函数通过锁（比如使用后面介绍的互斥量 mutex）来保护共享资源。这能防止两个同时执行的线程破坏对方的数据。对全局数据不加锁的使用会使函数变得线程不安全。

加锁意味着阻止多个线程对共享数据的访问，能防止数据被破坏。常见的锁称为互斥量（mutex），它是“相互排斥（mutually exclusive）”的简写。当一个线程拥有了锁并对受保护的代码进行操作的时候，其他线程必须等待或者是进行其他的工作。为了在等待的时候不占用 CPU 循环，线程可能被要求休眠一定的时间或者休眠到被特定消息和事件唤醒。

有很多和线程相关的特有的错误情况需要你去防止。当一个锁被激活用来和其他的线程进行交互，但是却从来没有被解锁的情况下，将会发生死锁。例如，两个线程在进行下一步处理前都在等待对方解锁。如果没有外部信号能对至少一个线程进行解锁的话，就将进入死锁状态。当两个或者两个以上的线程必须同时操作同一个数据区的时候，将会发生竞态条件（race condition），但是结果是依赖于这些线程的执行顺序。典型的，锁比如互斥量能用于避免竞态条件。

另外，所有的线程都被赋予一个执行优先级，它和进程的优先级类似。很明显，这能使线程的调度器更好地调度线程的优先级。当其他线程需要一个线程的结果，而它却没有完成任务的时候，我们称之为优先级失败。通常这是由于分配了不正确的优先级而导致的。饥饿失败也和优先级失败类似，它使线程不能在分配给它的时间里完成需要的任务[1]。

7.1.3 鉴别适合多线程编程的音频任务

通常情况下，能清楚地鉴别程序的哪个组件适合多线程编程是非常重要的。坦白地说，由于多个原因，传统的游戏编程倾向于避免使用多线程。首先大部分消费者的机器并没有多个处理器，因为消费者的操作系统并不支持。其次在单处理器的机器上使用多线程意味着需要更多的开销。我们并不认为在CPU周期上模拟多线程的功能是值得的。

尽管如此，在过去的几年里发生了一些事情。今天，操作系统（如 Windows XP）已经能支持多处理器，我们也认为游戏玩家有这样的系统。另外，有一件事也许更重要，在单处理器的机器上实现多线程特性需要的开销变小了，这是因为两个原因。首先，因为单个处理的平均速度在快速增加，即使切换线程的开销依旧和以前差不多，但是相对来说线程的开销比以前就小了。除此之外，现代处理器在设计上比以前的处理器能更加有效地运行多线程的程序。我们在下一节里将讨论 Intel 最新的硬件创新和这些芯片的分支。

音频编程有很多地方和多线程有明显的联系。任何一个音频系统都是很自然的呈现异步特性——也就是说连续的处理、混音，以及音频数据从原始的磁盘位置或者内存空间到目标硬件缓冲区的缓存过程——所有这一切都必须是实时的。我们通常希望音频流在主线程被其他任务打断以后还能继续，比如当需要加载新关卡的数据资源的时候。不像可视化的渲染，可以简单丢掉一些帧，没有有效的方法能掩盖一个音频数据流的饥饿——它会引起听得到的间断和暴音。

多线程编程一个明显的例子就是流式播放和解码高压缩比的音频数据,比如 MP3 或者 Vorbis 文件。不管发生了其他什么情况，这个任务需要周期性地访问磁盘资源，并且需要一定百分比的 CPU 时间来对磁盘中的音频数据解码，并把它填充到音频缓冲区中，所有这些都是需要实时的。

还有一些其他不是那么明显的线程在后台的使用方式。3D 音频数据，包括声音源（sound source）和听者（listener），在游戏世界中必须以异步方式和物体协同工作以确保正确的计算 3D 声音的输出。但是，如果每次听者或者是 3D 对象移动以后都需要计算音频输出的话，将非常耗时（可能会在每秒 60～90 次）。因此，你也许希望能降低 3D 音频数据计算的频繁程度。通过把这个计算放到一个独立的线程中，让它以比主更新低的频率周期性地唤醒这个进

[1] 译者注：处理器时间被更高优先级的线程占用。

程，这不仅仅节省了计算周期，把这些计算放到一个独立的处理器上，还允许超线程或者多处理器系统更加有效地工作。你还可以把数据传输任务放到一个独立的线程中，比如把声音数据从磁盘加载到正在播放的声音缓冲区中。

另外，你甚至希望能访问场景中的几何体并进行光线投射和寻路操作，以使音频系统能获益。如果这些看起来都不是音频相关的任务的话，你也许不了解很多现代音频的实现（如I3DL2 和 EAX）是需要多种任务的，如视线信息和其他空间信息来计算遮挡和衰减等声音属性，另外还有回声和回声的属性。这是一些很适合放到不同的线程中进行的任务。因为这些信息并不像可视的信息一样需要更新得那么快。因此，线程可以被调整为消耗较少的 CPU 周期，如果它和场景中的可视信息是异步更新的话。

在进入如何设置一个这样的多线程系统的详细描述之前，先让我们考查另外一个最近出现的技术，它使得线程即使在单处理器的机器上也变得更加重要。

7.1.4 Intel 的超线程技术是什么

Intel 最近在技术上的成就之一就是超线程（HyperThreading），它的设计使得一个处理器在操作系统（程序）上看上去是两个虚拟处理器。通过更加有效地使用在芯片上的多个执行单元（以前是仅仅用于无序执行），多线程实际上可以同时执行。和执行单个线程相比，它能获得很高的效率。

如同你预料的一样，它并没有两个物理上的处理器同时执行来得那么有效。这是因为两个线程在执行中总会有冲突，它们会同时请求处理器上的相同的资源。尽管如此，在最理想线程的情况下，两个线程都能在名义上获得 30%的性能提升，如果它们的负载是完全平衡的话。在最佳情况下，用户可能使用两个处理器，每一个都启用了超线程，那么能让系统拥有4 个虚拟处理器。

然而，超线程并不是万能的（magic bullet）。事实上，如果程序是单线程，或者相对于第二线程来说主线程的负载太重，它反而会降低整体性能。这个性能降低的原因是非常合乎逻辑的——处理器的资源被分给两个执行线程，而游戏只使用了其中一个线程，性能当然就会比把整个芯片资源都分配给一个主线程的时候来得糟糕。

从某种理解上来说，这把游戏程序员推到了一个难堪的位置——是该继续避免使用线程而允许在超线程的机器上有些轻微的损失呢，还是享受超线程的好处而接受在单处理器、单执行芯片上的性能损失呢？这个问题并没有确定的答案，但是游戏（和玩家）通常比其他应用更能提前对技术提出需求，玩家也会很快地接受一种新的技术，如果他能切实看到结果的话，许多硬件和软件开发都已经感觉到并行和多线程将是未来的主调。

7.1.5 线程编程技巧和操作

某些情况下，在游戏里创建一个后台线程的时候，并不需要让这个线程获得 100%的时间，因为工作线程通常比主线程要求更快地完成任务。MP3 或者 Vorbis 的解码，以及大多数音频相关任务都属于这种情况。

为了能以低的速率对数据进行解码，我们可以使用一个高优先级的线程，它处于周期工

作的状态，并休眠到定时器把它唤醒并再一次进行处理为止。这个机制保证了线程不会处于饥饿状态，同时也保证了工作线程占用的CPU时间是合理的。

设计线程系统最重要的步骤也许就是决定如何以及何时让工作线程去共享主线程处理的信息。实际上，这是两个线程惟一需要连接的地方，数据的传输必须是严格的安全和有效的。多线程不应该经常交叠，这一点非常重要，否则如果一个线程因为另外一个线程进入临界区而挂起，就会引起性能的损失。

7.1.6 多线程的示例程序

现在我们来检验一个小程序，它用来评估把解码任务放到一个线程里后能得到多少性能提升。另外它还演示了一个程序如何能允许两个代码路径：多线程或者单线程，这个简单的线程性能测量程序可以演示这两种技术在不同系统上进行的性能测试。我们将在后面展示测试的结果。

程序清单7.1.1展示了完整的测试程序。注意光盘里包含了这个程序。

程序清单7.1.1 多线程性能测试程序

```
#pragma pack(push,4)
__int64 g_total_val = 0;
int g_num_calculations = 0;
bool g_do_floating_point = false;
CRITICAL_SECTION g_val_update;
// 做一些计算，提高CPU的负载
void DoCalculations()
{
      int val = 0;
      if(g_do_floating_point)
      {
            for(int i = 0; i < g_num_calculations; i++)
            {
                  for(int j = 0; j < 1000000; j++)
                  {
                        val += int((float)i * (float)j *
                          ((float)j - (float)i - 0.25f) /  (val + i + j + 1));
                  }
            }
                  EnterCriticalSection(&g_val_update);
                  g_total_val += val;
                  LeaveCriticalSection(&g_val_update);
      }
      else
      {
                  for(int i = 0; i < g_num_calculations; i++)
                  {
                              for(int j = 0; j < 1000000; j++)
                              {
                                          val += i * j * (j - i) /
                                                          (val + i + j + 1);
```

```
                                }
                        }
            }
            EnterCriticalSection(&g_val_update);
            g_total_val += val;
            LeaveCriticalSection(&g_val_update);
        }
    }

    // 线程的启动函数
    void ThreadFunction(LPVOID lpv)
    {
        HANDLE hEvent = (HANDLE)lpv;
        DoCalculations();
        SetEvent(hEvent);
    }

    // 开始线程的时间测试
    int main()
    {
        char c;
        cout << "Do floating-point calculations (y/n)? ";
        cin >> c;
        if(c == 'y')
            g_do_floating_point = true;
        int threads = 0;
        cout << "How many total threads do you wish" <<
            " to create (including the main thread)? ";
        cin >> threads;

    // 如果有多个线程，就分配一个句柄数组
    HANDLE* pHandles = 0;
    if(threads > 1)
    {
        pHandles = new HANDLE[threads - 1];
    }

    cout << "How many millions of calculation loops" <<
            " should each thread perform? ";
    cin >> g_num_calculations;
    cout << "Now performing timing calculations...";

    InitializeCriticalSection(&g_val_update);

    // 记录起始时间
    unsigned int start_time = timeGetTime();

    // 为每个线程进行必要的预计算
    int i;
    if(threads > 1)
    {
```

```
        for(i = 0; i < threads - 1; i++)
        {
            pHandles[i] = CreateEvent(
                NULL, FALSE, FALSE, NULL);
            if(_beginthread(&ThreadFunction, 4096, pHandles[i]) == -1)
                return -1;
        }
        DoCalculations();
        // 在继续执行前等待另外的线程的结束
           WaitForMultipleObjects(threads - 1, pHandles, TRUE, INFINITE);
    }
    else
    {
        // 在单线程分支中做一些简单的计算
        DoCalculations();
    }
    // 得到结束的时间
    unsigned int end_time = timeGetTime();
    // 我们不再需要这个临界区了
    DeleteCriticalSection(&g_val_update);
    // 删除用于同步的句柄
    if(threads > 1)
    {
        for(i = 0; i < threads - 1; i++)
        {
            CloseHandle(pHandles[i]);
        }
        delete[] pHandles;
    }
    cout << " Finished!" << endl;
    cout << "Performed all calculations in "
        << end_time - start_time <<
        " milliseconds" << endl;
    getch();
    return 0;
}
#pragma pack(pop,4)
```

从本质上讲，这是一个基于线程的性能测试程序，它能计算出在任意数目线程的数学计算的开销。用户需要输入计算类型（浮点/整形），要运行多少线程，以及每个线程要有几百万的计算周期。

要模拟数据从线程中传输到普通缓冲池，就要计算被周期性加添到全局变量 g_total_val 的值。这是我们前面提到过的所有重要数据的传输。因为这是一个共享访问点，我们必须以保护的方式访问这些变量。别被这个变量迷惑了，它仅仅演示了如何从线程访问普通/全局变量。让我们简单地浏览一下代码，了解如何管理线程，接下来讨论如何将这些技术应用到音频系统。

第一个值得注意的函数是 DoCalculations()。它会进行一系列以毫秒为递增的循环计时测试计算。计算结果被保存到全局变量 g_total_val。因为这个全局变量被多个线程

同时访问，我们必须首先使用一个互斥量（mutex）锁住这个全局变量，并在访问结束后解锁。在写多线程代码时，要尽量减少锁的次数。举个例子，如果我们反复锁住和解锁一个全局变量，线程的效率将会降低，因为当其他线程在试图访问这个变量时会直接停止动作。

函数 ThreadFunction()对 DoCalculations()进行了封装调用。它将一个空指针作为参数。这是一个新线程的进入点。当这个函数结束，线程将会自动终止自己。

剩下的程序就比较直观了。程序向用户询问一些问题，例如：要进行哪种类型的计算，要创建多少个计算线程。在获取这些必要信息后，程序设定了一个计时器，然后启动一个或多个线程，每个线程在 DoCalculations()函数内处理一系列的计算。计算结果将被存储在全局变量 g_total_val 中。

这个例子演示了怎样将一个任务切分为一系列的多并发可执行任务。我们使用这个程序在不同的机器上做一些简单的计时测试，以此演示线程是怎样增加效率的。

在这个测试中，例子程序运行在 2 台测试机器上。测试机一：2 个 Xeon Pentium4 2.4 GHz；操作系统：Windows XP 专业版。测试还附带有开启和关闭超线程技术（HyperThreading）的展示。测试机二：Pentium4 1.5 GHz；操作系统：Windows 2000 专业版，没有打开超线程技术并且是单处理器。每种配置，我们都以整形、混合整形、浮点形计算方式运行。选择了一个测试周期（12 亿），以求一个有意义的时间长度。记住，总时间相应的比每种配置在每次测试中的相应执行时间要少。这里使用一个稍微原始的计时方法，所以执行时间会有 20 毫秒的出入，表 7.1.1 提供了这个测试结果。

表 7.1.1　　例子程序的测试结果

配　置	1 线程	2 线程	3 线程	4 线程	5 线程	6 线程
M1，HT，整型	28 078 ms	16 625 ms	12 172 ms	10 125 ms	10 438 ms	10 516 ms
M1，HT，浮点	68 719 ms	34 843 ms	23 500 ms	19 813 ms	20 672 ms	20 578 ms
M1，no HT，整型	28 063 ms	14 047 ms	14 062 ms	14 046 ms	14 141 ms	14 078 ms
M1，no HT，浮点	68 859 ms	34 422 ms	29 797 ms	27 484 ms	26 032 ms	25 125 ms
M2，整型	43 312 ms	43 332 ms	43 492 ms	43 593 ms	44 163 ms	43 442 ms
M2，浮点	147 232 ms	108 246 ms	95 247 ms	90 059 ms	85 943 ms	82 418 ms

注释：M1=机器 1，M2=机器 2，HT=超线程激活，no HT=超线程关闭

从表中可以看出在双处理器的机器上从单线程切换到双线程有多大的性能提升。这并不是不可能的，两个线程能并发地执行在 2 个实际的物理处理器上。实际的效率损失来源于同步执行和 2 线程间的数据采集。这就是从表中没有看到 100%的效率提高的原因。

测量超线程技术性能时（3 或 4 个线程运行在双处理器机器上，打开超线程），可以看出更加少的但是仍然有 30%的效率提升。这符合 Intel 的研究指出的当在理想情况下能期望的性能提升。有一个非常古怪的测试结果：在 2 台机器上的浮点测试中，多于 2 个线程的情况下均有性能提升，但是从逻辑上得出的结果刚好相反。这个测试的实际结果还不是很清楚，可以猜测，2 个操作系统和处理器的设计是为多线程优化的。特别是在多处理器上取得了更好的效果。而混合浮点、整型计算也仅仅达成了一半的目标。

最后，注意在测试结果中每个线程会产生有多少开销——在我们的测试里，结果是基本可以忽略的。只有在增加线程数，增加时间的技术上才能提高测量的准确性。在第二台机器上，因为是单个处理器，而且也没有超线程，操作系统只能通过模拟来得到多线程的能力。

我们可以从这些数据中得出什么呢？在现今运行 Windows 操作系统的 PC 上，在单处理

器上处理多任务是非常高效的，只要线程数不要变得很大。然而，现今的系统都被设计为处理并行任务，我们可以在性能增强的同时找到一组平衡处理器和线程数量的关系。如果游戏能按照这种方式设计，将会享受到并行执行的优势。

7.1.7 实时流数据机制

通常，我们需要一个线程系统来进行 MP3 解码或者其他类型的需要处理的音频数据。在系统间传输流数据有 2 种模式（和在线程间传递数据一样）。

第一种传输方式叫做“压”（Push）传输。这是因为：准备好传输一块数据时，这个系统将会通知目标系统，这时，源数据将会被压入到系统缓冲区中。网络间的音频流数据一般就是这样做的。音乐数据不能超过网络所能承当的数据传输能力，所以控制传输率就变得非常重要。如果音乐数据传输的比实际要快，那么多余部分将会被缓冲起来，待需要时继续传输。或者，可以使用一个节流阀装置确保有限量的数据在缓冲内。

另外一种方式叫做“拉”（Pull）传输。源系统在准备传输数据时将通知目标系统。那么目标系统将把数据从源数据拉过来。这种方法经常被用于实现一个 MP3 系统解码模型。如果想更多地了解这个模型和接口是如何传输对象的，微软网站上的 COM 方式的 IStream 接口有对这 2 种模型的解释。

7.1.8 流和线程

如果要写一个类似于 IStream 的流数据接口，那么你只需要关心这个音频系统里的线程是怎样实现的。我们提出两种可选方案——它们并没有绝对的正确或者错误，但各自都有优缺点。

现在最实质的问题是：怎样创建线程系统？现在假定使用“拉”传输模型。系统将周期性地需要一些小块音频数据。系统将在需要数据时向 MP3 解码对象请求数据。

第一种选择是创建一个基于 IStream 的解码对象，并视其为线程黑盒子，在外部有任何请求到来的时候，它允许其创建一个内部线程来响应抓取数据、解码并通过接口函数呈现出来。客户端从主线程发出数据需求。IStream 解码对象将始终从线程获取下一个请求，并用线程解码数据。当解码器对象内已经解码的数据低于一个指定的值时，线程重新开始工作，更多的数据被解码，直到重新达到某个极限。这时解码数据对主线程来说将一直有效，同时线程问题将整个隐藏在解码对象的接口背后。图 7.1.3 显示了这些接口间的关系。

这种方法最吸引人的地方就在于线程处理包含在解码对象里。这样可以保证 2 个线程间的高度不相关性——这总是件好事。对于那些寻求简单而鲁棒的方案的人来说，很难找到比这更好的方案。

任何简单的方案都是有一定的缺点。对初学者来说，意味着每个新的流将使用一个自己的线程。如果计划创建多线程解码对象，也许不确定是否该使用这种方法。我们做过一个测试，结果非常令人惊讶：运行多解码器时，如果每个流使用一个自己的线程，将明显比在一个线程排队处理要高效得多，特别是在单处理器之上。不幸的是，在一些特殊的情况下，这种机制并不起作用。例如：当一个声音缓冲启动时，缓冲总是以填充初始数据作为开始。这时，既要保持一些预缓冲数据作为解码数据使用，系统在播放声音缓冲前又要足够的智能来等待这些数据的解码。这样，无形中就为这个简单的解决方案增加了复杂度。

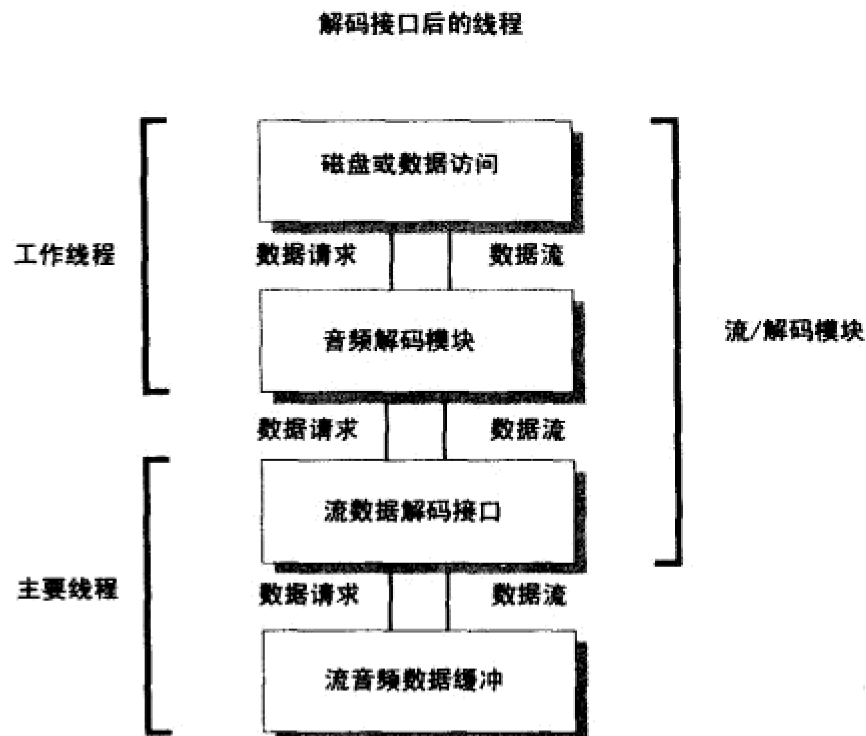


图 7.1.3 解码接口后的线程

另外一种方法是创建一个包含有更大些的子系统的线程。这个线程可以维护一系列的声音缓冲流和他们各自的解码对象。这个线程周期性地被激活，它向每个缓冲区写入需要的数据，然后一直休眠到被定时器唤醒，重新开始循环。

这个系统有一些确定的优点和缺点。优点是预填充缓冲区将留下一个很大的空间，因为这往往发生在使用同一个线程并往缓冲区周期性地添加新的数据时。结果就是初始化时的解码对于剩下的系统是不可见的。这个系统的一个问题是，那么多的数据被一个线程管理，在主线程和这个线程之间的接口将变得越来越复杂。因为比起简单的实时填充缓冲，线程直接访问声音缓冲时将被完全阻塞。这就意味着其他成员周期性地访问声音缓冲时将是非线程安全的。我们将耗费更多的精力来解决线程安全性和排错。图 7.1.4 显示了这个系统之间的关系。

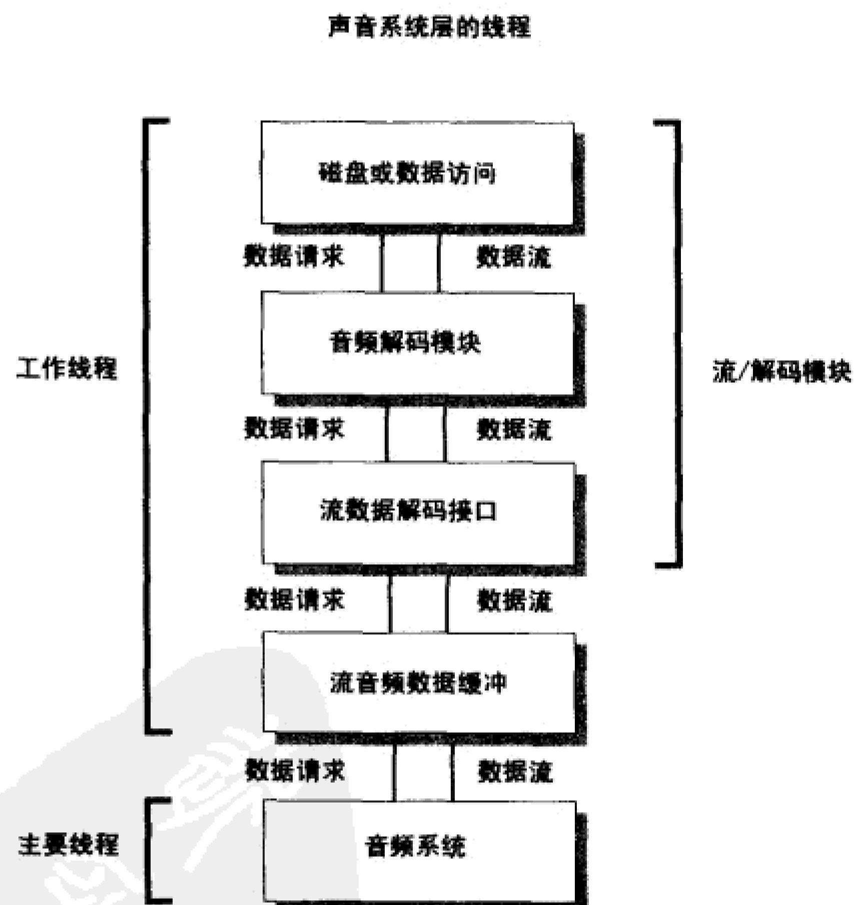


图 7.1.4 声音系统上的线程

7.1.9 总结

决定是否进行线程编程不是那么简单的。在设计的合理性和硬件的支持下，线程技术将使程序运行地更高效，同时也会难于调试，还会给你的编程词汇里加一个新的说法：新东西总是带来更好的效率并且花费更多的时间。

未来，游戏将会在这一代和下一代的 PC 和游戏机平台上利用新的多处理器技术或者超线程技术。那时，游戏将不会消耗很多的 CPU 周期，游戏程序员将更有可能推动新的硬件技术发展和发现更多的奇迹。

7.1.10 参考文献

[Boer02] Boer. *Game Audio Programming*. Charles River Media, Inc, 2002.

[Intel] Multithreaded and Hyperthreaded documentation. Available online at *www.intel.com*.

[MSDN] Multithreaded documentation. Available online at *msdn.microsoft.com*.

7.2 基于组的声音管理

eV Interactive 公司，Matthew Harmon
matt@ev-interactive.com

当前的声音程序接口提供大量独立的声音播放控件。音量、播放速率、位置，还有暂停/再继续状态是今天的 API 所提供的通用的特性。大部分 API 还提供一个主音量控制，它能改变所有在这个系统里播放的声音的音量。

这里有一些案例，当然，在这里程序员需要用组去管理相关的声音的播放。快速而且容易地改变整个组的声音的属性将会非常方便，而且预先把这个功能整合到一个游戏声音 API 里可以为后面的工程节省大量的时间和减少很多挫折。这这些好处可以用一些简单的案例表现出来。

- 一个团队正在开发一个包括室内和户外区域的冒险游戏。当玩家走近室内，所有从户外区域传来的环境声音——鸟的喳喳声、风吹过的声音、蟋蟀的唧唧声等，音量都应该减小。可以在这里使用遮挡技术，但是这经常有些要求过高，而且不是所有的平台都支持。取而代之，这个团队仅仅需要在玩家从一个区域转换到另外一个区域的时候动态地平衡室内与户外的声音的音量。
- 为一个空战游戏工作的程序员们最后考虑要实现一个用户参数选择菜单。屏幕的音频必须提供一个接口，它允许玩家独立地调整多种声音效果的音量：引擎和环境噪音、战场的警告声、无线电信息声和背景音乐的声音。程序员为每一个声音组增加一个音量衡量系数，接着必须追踪每一个声音触发的地方，并应用正确的衡量系数。糟糕的是，这仅仅是一个快速的设置，因为已经激活的声音不会被用户的声音控制滑动条向前或向后所影响。
- 一个创新的卡通风格的游戏给玩家提供了动态加速时间的能力。为了支持产品的这个古怪特性，这个团队想要加快声音特效的播放的速率，但是让音乐和界面声音仍然以正常的速度播放。

如果在项目的开始就把基于组的声音的管理直接封装到 API 里面，处理这些情况就变得极其简单了。

7.2.1 API 包装预览

大部分声音播放 API 在它们的核心上非常类似，并且本文假设低级别的声音 API 将会被封装到一个自定义的接口层里。为了做一个例子，让我们设计一个简单的 2D API 的封装，在这里声音被当成普通的句柄来管理。

```
handle = SndPlay(sampleBuffer);
SndSetRate(handle, newPlaybackRate);
SndSetVolume(handle, newVolume);
SndPause(handle);
SndResume(handle);
SndStop(handle);
```

为了使基于组的管理能运作起来，我们将增加一些简单的函数。首先，我们需要一个将声音和组联系起来的方法。

```
SndSetGroup(handle, group);
```

或者，我们通过在一个声音被触发的时候请求分配一个组来强制使用声音组。

```
handle = SndPlay(sampleBuffer, group);
```

接下来，我们增加一些非常简单的程序来手工操作整组声音的播放参数。

```
SndSetGroupRate(group, newPlaybackRate);
SndSetGroupVolume(group, newVolume);
SndPauseGroup(group);
SndResumeGroup(group);
SndStopGroup(group);
```

这个基本的伪代码能够被扩展成一个有 3D 界面的基于对象的 API。控制其他的声音参数可以很简单地添加。

7.2.2　能力

理论上，所有关于控制播放的参数都能在组的级别上进行管理。然而在实践中，仅仅有一些参数是有用的，就像早些时候的 API 概要里显示的那样。有必要考察一下这些属性，看看组的控制里应该包括哪些属性。

1．音量

不难看出立刻改变整个声音组的音量会非常方便。在每次播放声音的时候给音量参数的结果增加组级别的控制伴随着以下音量控制系数：

$$V_{actual} = V_{sample} * V_{master} * V_{distanceAttenuation} * V_{group}$$

V_{sample} 控制单独播放的声音的音量，V_{master} 是系统整个的主要音量，在 3D 声音方面 $V_{distanceAttenuation}$ 也会被应用，或者是被声音硬件本身使用。为了达到这个目的，我们加入 V_{group}，它能根据组来调整给定的回放音量。

2．声调

声调，或者说是播放速率，是另一个用来区分组控制器的有用的属性。控制一个采样的声调的系数如下：

$$P_{final} = P_{normal} * P_{sample} * P_{dopplerEffect} * P_{group}$$

在这里，P_{normal} 是声音数据的原始采样速率，就像音量一样，每个被播放的声音都有自己的速率修改器 P_{sample}。$P_{dopplerEffect}$ 是 3D 处理的结果，可以在软件或硬件上应用。最终，我们用 P_{group} 来修改组里所有声音的速率。

3. 暂停和恢复

暂停和恢复声音组包括一些逻辑上的判断。在大多数实现里，最明智的方法也许是拥有一个位于独立声音状态之上的、基于组级别的控制操作。换句话说，一个声音仅仅会在本地和组的控制器都处于“playing”状态的时候才会播放。同样地，组控制器的调用从来不会直接影响一个声音的本地控制器的状态；一个处于暂停模式的声音不会开始播放，即使它所属的组会接受一个播放的命令。为了激活一个声音，它的低级别的本地的控制器必须也设置成播放状态。

4. 停止

停止一个声音组非常的直接。然而实际上，一些声音系统在声音完成播放时会放出回收信号或者另外发出通知。当停止一个声音组的时候，一定要触发通知系统，让游戏正确地了解声音组的状态。

7.2.3 定义组

在先前的基本的 API 概要里，声音组被看成未定义的类型，就像我们简单地提到的组一样。当它变得完善的时候，这里有好几种定义组的方法，每一种都有它的优点和实现的困难。

1. 简单的组 ID

大部分给声音分类的基本方法是用简单的整数去标识一个组。仅仅用一个字节，我们就能管理 256 个不同的组，当然，用一个完整的双字我们能管理好几百万个组。当数百万个独特的组看起来有些过分的时候，那个冒险游戏的实例再次发生了。每个游戏里封闭的区域可以赋予一个不同的组 ID，分类管理系统还不如突然地变成一个十足的剔除系统。

从概念上，用简单的组来管理是很直观的。然而，由于每个组将需要保存一些数据，把分类映射到 ID 需要一种间接的和一些额外的管理。你当然不想去预先分配 232 个组的数据结构并直接索引它们。

2. 组的位域

另一个供选择的项是用一个整数来表示分类的位域。用一个双字的整数，我们有 32 个不同的声音分类。用位域增加弹性，使之能和简单的 OR-ing 分类标志一起一次控制好几种分类。这被证明是相当灵活的。事实上，通过简单地使用整套位域的方法，它甚至提供在所有声音之上的普通的“主”控制器。位域管理也是最容易实现的选项，就像一个有 32 个组的数据结构的固定数组可以预先分配并且直接索引一样。

当使用基于位域的分类时，声音播放采用这种形式：

```
typedef enum
{
SNDGRP_MENU   = 0x00000001,
SNDGRP_VOICE  = 0x00000002,
SNDGRP_EFFECT = 0x00000004,
SNDGRP_MASTER = 0xFFFFFFFF
} SNDGROUP;

// 引发一些声音
hSndBeep    = SndStart(sampleBeep);
hSndHello   = SndStart(sampleHello);
hSndGoodbye= SndStart(sampleGoodbye);

// 把它加入到组里
SndSetGroup(hSndBeep,    SNDGRP_MENU);
SndSetGroup(hSndHello,   SNDGRP_VOICE);
SndSetGroup(hSndGoodbye, SNDGRP_VOICE);

// 改变所有语音的声调(voice)
SndSetGroupRate(SNDGRP_VOICE, 1.3f);

// 改变游戏中的所有声响的音量(特效声和语音)
SndSetGroupVolume(SNDGRP_VOICE | SNDGRP_EFFECT,
0.75f);
```

分类位域选项天生地限制了声音分类的有效数量，在一些应用上，这将不是问题，而且简单的位域方法成为可选择的设计。然而，像先前的例子一样，当需要更多的分类的时候，这里会出现一些情况。即使在这种情况下，也可以使用一个宽位域的对象，比如std::bitset或者boost::dynamic_bitset。

3. 组对象

第三个选项是通过对象管理声音组。通过创建一个管理声音组数据的类，我们仅仅分配需要的一些组的对象。

通过对象管理声音组可能看起来像这样：

```
// 定义一些组
SNDGRP  sndGrpMenu;
SNDGRP  sndGrpVoice;
SNDGRP  sndGrpEffects;

// 引发一些声音，当触发声音时将其加入到组中
SndStart(sampleBeep,    &sndGrpMenu);
SndStart(sampleHello,   &sndGrpVoice);
SndStart(sampleGoodbye, &sndGrpVoice);

// 改变所有语音的声调
sndGrpVoice.SetRate(1.3f);
// 改变游戏中所有声响的音量(特效声和语音)
```

```
sndGrpVoice.SetVolume(0.75f);
sndGrpEffects.SetVolume(0.75f);
```

通过对象管理组需要一些特殊的处理。例如，如果一个组的对象在所有与之关联的声音停止播放之前被删除将会发生什么？虽然这种情况未必会发生，但它们至少应该被考虑到。

我们也可以这样构造系统，组对象担当一个声音播放控件的代理，并且所有的声音参数通过组对象自身去操作。然而这个设计可能被证明是过度复杂的，并且颠覆了大多数声音 API 的简单性。

7.2.4 实现细节

接下来讲述一些实现上的细节。

1. 跟踪播放声音

为了动态地改变声音组的属性，API 的接口需要明了当前播放（或暂停）的所有声音。在一些场合，一个现有的声音接口可能仅仅跟踪个别的声音（通过句柄或对象），并且不保存一个全局的活动声音的表。

为了能动态控制一个组的属性，系统需要在组的属性被改变的时候更新每个受影响的声音的属性。当一个像 SndSetGroupVolume()的调用被处理的时候，系统需要访问和更新组中的每一个当前活动的声音。

在像 OpenAL 和 DirectSound 一样，把播放中的声音当做对象（不必是 C++对象，可以是架构上其他的对象）来管理的 API 里，通常会保存一个包含所有活动的声音的表。在组发出命令时，这个表会遍历一次，并更新所有适当的声音。

在像 FMod 和 Miles 声音系统一样基于“跟踪（track）”的 API 里，很可能你封装的 API 已经跟踪到哪个声音在哪个通道里，因此组的管理会容易实现一点。

2. 简化的一次性声音

一次性声音是游戏中的很短的效果，并且从来不用去控制。枪声和脚步声是很好的例子。在一些情况下，封装 API 包括简单地调用以播放一次性的声音，而不必暴露动态的声音管理。例如一个射击，持续的时间很短，在播放它的时候不需要改变音量和声调。当简短的声音触发的时候，系统会计算出所有重要的参数，并在声音播放期间保持一个常量。一个一次性的声音可以通过下面这个调用来触发：

```
SndPlayOneShot(sample, volume, rate, pan, loopCount);
```

在实现声音组的管理的时候，你需要决定支持动态的、基于组的一次性的声音管理是否重要。回答很可能是 no，但是如果系统允许无序地播放一个很长的采样时，基于组的管理可能是需要的。

即使在触发一个声音之后不需要动态控制，一次性的声音也应该被分配给一个组。因而当声音初始化参数被计算出来时，组的因素要被考虑到。这仅仅涉及增加一个组的作业到声音的触发调用函数中：

```
SndPlayOneShot(sample, group, volume, rate, pan, loopCount);
```

3. 可移植性

不管创建了什么样的包装接口，它最好是能和所有你可能遇到的声音 API 类似。这将会帮助你构建一个有足够弹性的基于组的声音管理系统，它可以用于多种平台和多种底层的声音系统。像前面提醒的一样，不同的声音系统会采取不同的方法去管理个别正在播放的声音，理解这些差异是构造一个真正方便的 API 包装的关键。

7.2.5 结论

本文展示了用组来管理声音的概念，除了在单独的级别之外，它很有用而且容易实现。花几个小时的时间就能把这个特性加入到一个声音 API 的封装里，而受益却是很深远的。尤其是当一个基于组的声音管理的需求在开发的早期无法被确定的时候。在声音 API 里包含这些特性还保持了游戏代码的干净利落，并且允许很容易地包含一些能顺手增加的特性。

7.3 利用三维曲面作为音频发生器

Sami Hamlaoui
disk_disaster@hotmail.com

在今天的游戏中，不论是哪种声音资源，都会被表示为一个单独的点。例如：表示枪声是一组点，激光束是两组点，而表示下雨就是好多组点。尽管这个方法能够满足迄今为止的要求，但是随着每一代硬件设备的更新，游戏也在不断地追求着更高的细节表现。不过现在的一些技术，例如 EAX，可以从某些方面提高游戏声音的逼真程度，但是所有的声音还是来源于在一个 3D 空间中的某一个无穷小的点。我们再也不想让声音从一个点发出来了——我们需要声音能从一整个面上发出来。

这篇文章将展现如何基于面来创造声音，并且这种技巧基本上不需要花费过多的处理时间，而且不会降低硬件的效率。感兴趣吗？那就读下去吧。

7.3.1 方法

这个技巧的基础与单点定位音源的方式不同，你可以使用一个标准的几何体，例如一条线、一个盒子或者一个球。然后在每一帧计算几何体上离收听者最近的那一点，并将这一点作为音源的位置发送给 API。

这个方法看起来似乎平淡无奇，但是请注意到当听者围绕着几何体移动的时候，声音也会跟着他移动！！例如当你与几何体上的一条线段平行移动时，声音也会跟着你移动，直到你越过到这条线段的端点，声音就会自然而然地移动到你的身后了。当你围着一个球体运动时，声音可能感觉是一个点发生器，但是当你进入球里的时候，声音就会从各个方向传过来，给玩家以体积感，就像声音真的是在球的内部产生的一样。如果用一个盒子来表示建筑物外边的雨声，你只需要走到该建筑外的空地就会发现像要被雨声震聋了一样。

这个方法的另外一个好处就是，音频 API 和音效卡还会认为它们处理的是点音源，这就意味着在硬件方面不会影响硬件的效率。惟一一个需要是在软件方面实现一个最近点算法，这对于所有的发生器来说都太简单了，不会在使用的时候有任何表现上的难度。

在接下来的部分，我们会讨论四种不同的发生器：点发生器、线发生器、球体发生器和方体发生器。然后给出每一种发生器所需要的要素列表（源、方向、范围等），并一步一步地介绍最近点计算方程，以及在具体游戏中，发生器应该用到什么地方。

1. 关于体

就像我们先前提及的，一旦收听者在一个体发生器内（与点发生器或者线发生器不同，这个发生器有实际意义的“内部”，例如一个球或者一个盒子），声音将会从各个方向传过来，给玩家以体积感。这是该技术的特点，而且可以用来处理很多特殊音效。具体情况具体分析，这里就不一一列举这些特效了，以上给的那个下雨的例子只是冰山一角。

2. 关于数学

我们在讨论每一个发生器的要素的时候都会考虑到其最近点方程中变量的数学表现形式。不过，每一个方程中将涉及的变量不会太多，就像表 7.3.1 中的两个变量，他们是每次都会被用到的。

表 7.3.1

名　字	数学符号	类　型	名　字	数学符号	类　型
声音位置	**A**	向量	收听者	**X**	向量

定义如下。

声音位置：这是一个会被送到 API 的用于最近点算法的向量，用 **A** 表示。

收听者：一般来说会是摄像机的位置。也就是当前声音被听到的位置。用 **X** 表示（因为 **L** 被用来表示线性要素）。

另外还有两个会被用在方程中的临时变量，见表 7.3.2。

表 7.3.2

名　字	数学符号	类　型	名　字	数学符号	类　型
方向	**V**	向量	距离	*d*	标量

定义如下。

方向：从一个东西到另一个东西的方向。如无特殊说明，这个变量不严格遵循字面意义，因此会包括一些距离和方向的信息。

距离：从一个东西到另一个东西的距离。这是一个标量，一般会在对收听者做点投影的时候使用。

7.3.2　点发生器

表 7.3.3 是点发生器所涉及的要素。

表 7.3.3

名　字	数学符号	类　型
原始	$\mathbf{P}_{origin}$	向量

首先讨论的是点发生器，尽管重点讨论的是与其恰恰相反的体发生器，但是在实际实施中，点发生器可以满足大部分案例的要求。注意，当且仅当要为了实现声音效果而不得不建立其他发生器的时候，才应该放弃使用点发生器，除此之外，首先应选用点发生器。

点发生器就是将一个点作为音源。将这个点的值赋给 **A**，由于点除了位置之外再没有其他的值了，因此最近点就是它本身。但是为了文章的完整性，将最近点的方程（7.3.1）列在下边。

$$\mathbf{A}=\mathbf{P}_{origin} \quad (7.3.1)$$

将声音和点设在同一位置。

适用范围

刚刚也提到过，除非到了非用其他发生器不可的情况，都可以用点发生器来解决问题。它适用于说话、脚步声、小的爆炸、小物体击中目标、火箭追踪的声音、水滴掉落等情况。

7.3.3 线发生器

线发生器分为两种，无限线发生器以及线段发生器。一般常用的是线段发生器，除非是在线十分长或者无限延展的情况下。

1. 无限线发生器

表 7.3.4 是无限线发生器所涉及的要素。

表 7.3.4

名 字	数学符号	类 型	名 字	数学符号	类 型
线上的点	$\mathbf{L}_{point}$	向量	方向	$\mathbf{L}_{dir}$	向量

无限线就意味着这条线将会永远延展下去。为了表示它，只需要跟踪线上的一个点，以及线延展的方向（表 7.3.4）。为了计算声音位置，你需要将收听者的位置垂直投射在线上。不要怀疑这个简单方法的实用性，只要记住必须在整个计算的等式中使用线上的同一个点，不然声音位置就会建立在错误的位置上。

无限线发生器的最近点计算公式 7.3.2：

$$\mathbf{V}=\mathbf{X}-\mathbf{L}_{point}$$
$$d=\mathbf{V}\cdot\mathbf{L}_{dir}$$
$$\mathbf{A}=\mathbf{L}_{point}+\mathbf{L}_{dir}\,d \quad (7.3.2)$$

创建步骤为：

（1）在收听者和线上任何一点间建立向量；

（2）对该向量和线的方向做点乘。这会告诉我们建立新的点会离测试点沿着线上有多远；

（3）通过用距离调节方向，并将结果加上步骤 1 中使用的点，就能得出新声音的位置。

2. 线段发生器

表 7.3.5 是线段发生器涉及的要素。

表 7.3.5

名 字	数学符号	类 型	名 字	数学符号	类 型
原始	$\mathbf{L}_{origin}$	向量	长度	L_{length}	标量
方向	$\mathbf{L}_{dir}$	向量			

线段是无限线上的一部分。和无限线一样，线段计算也需要线上的一个点和方向，不过线段发生器的处理还需要知道线段的长度（表 7.3.5）。线上的点应该是线段起点，长度就是离这个点多远线段才结束。

计算声音位置的方法和等式 7.3.2 没什么差别，除了需要在 0 和线段长度之间做一个限定。如果不做的话，那么结果就完完全全和无限线发生器的计算一样了。

线段发生器的最近点计算公式 7.3.3：

$$\mathbf{V}=\mathbf{X}-\mathbf{L}_{origin}$$
$$d=\text{clamp}(\mathbf{V}\cdot\mathbf{L}_{dir},0,L_{length})$$
$$\mathbf{A}=\mathbf{L}_{origin}+\mathbf{L}_{dir}\,d \quad (7.3.3)$$

创建步骤为：

（1）在收听者和线段的起点间建立向量；

（2）对该向量和线的方向做点乘，并且将数值钳制在 0 和线段长度这一区间内。这会告诉我们建立新的点会离线段的起点沿着线上有多远；

（3）通过用距离调节方向，并将结果加上线段的起点，就能得出新声音的位置。

3. 适用范围

有可能会用到的情况：

坏掉的霓虹灯管：由于这个管子在不断地闪烁，你可能会在这上边做声音的文章。并配合动态的光影来渲染当时的气氛。

激光束：一束从超现实的来福枪里射出来的光束，很明显要使用线发生器来做。与声音在枪口发出不同，玩家能听到一种能量沿着光束脉动的声音。

7.3.4 球体发生器

表 7.3.6 是球体发生器所涉及的要素。

表 7.3.6

名　字	数学符号	类　型	名　字	数学符号	类　型
原始	S_{origin}	向量	半径	S_{radius}	标量

球体是我们讨论的第一个体发生器。跟线或者方体不同，球体一般只有一种表现形式，一个球心和一个半径（见表 7.3.6）。我们在方法部分介绍过，体发生器会在收听者走到体的内部时候发挥特殊的作用——声音从多方传来，有体积感。可以作为球体或者方体的适用范围的例子。

在一次小型爆炸中使用球体发生器并不会带来太多好处。实际上，很多球体发生器即使被点发生器所代替，玩家也不会注意到。球体发生器只适合于：虚拟声音是伴随着球在运动的，玩家可以走进或走出这个球。注意，方体发生器不适用这种情况，因为方体会改变声音运动的方式（因为声音延某一坐标轴运动，而不是在曲面运动），所以在使用方体发生器的情况下，任何其他的发生器都代替不了。

球体的最近点计算公式 7.3.4：

$$\mathbf{V}=\mathbf{X}-\mathbf{S}_{origin}$$

$$d = \min(|\mathbf{V}|, S_{radius})$$
$$\mathbf{A} = \mathbf{S}_{origin} + \hat{\mathbf{V}}d \qquad (7.3.4)$$

创建步骤：

（1）在收听者和球心间建立一个向量；

（2）对球的半径和该向量的绝对值取最小。这会说明球心和音源之间有多远，如果收听者在球内部，那么绝对值最小，不然，就是球半径最小；

（3）通过用距离调节法向量，并将结果加上球心的位置，就能得出新声音的位置。如果收听者在球外，那么这个位置会在球面上，不然，那个位置就是收听者的位置。

适用范围

一些可能用到的例子：

爆炸：尽管大部分的爆炸都可以用点发生器来做，但是如果爆炸是一个有半径的爆炸，那么球体发生器就是最佳选择。如果能把发生器的半径和视觉上的冲击波特效的半径进行绑定，那么效果会更好。

防护罩：当被一个防护罩包围的时候，你的周围会出现声音。如果你在防护罩外边，那么防护罩的表面会发出声音。防护罩越大，球体发生器的作用就越明显。如果玩家能够走进防护罩的话，声音将会是他得知他已经进入的另外一种方式。

7.3.5 方体发生器

表 7.3.7 是方体发生器所涉及的要素。

表 7.3.7

名　字	数学符号	类　型	名　字	数学符号	类　型
原始	$\mathbf{B}_{origin}$	向量	最大范围	$\mathbf{B}_{max}$	向量
最小范围	$\mathbf{B}_{min}$	向量			

按坐标轴排列的方体发生器可能是除点发生器外对游戏开发者来说最有用的了。因为大部分游戏里的物体已经被定义了 bounding box，所以从这个盒子上发出的声音基本上不用任何额外的计算，除了最近点！声音在盒子上有一个二维坐标系运动（xy，yz 和 xz）。这就意味着如果你和盒子的一个面平行走的话，除非你走到了另一个面，否则声音会始终保持和你一样远的距离。自然，作为一个体，当收听者一旦进入到里面，声音就会从各个方向传来，给人以体积感。

利用 bounding box 有两种方法：偏移和绝对。偏移的 bounding box 和盒子的原点做相关，绝对的 bounding box 则包含盒子所在的 3D 空间的真实的值。记住，与无限线和线段发生器的计算结果不同，偏移和绝对算出的结果是完全相同的——不过是处理数据的两种方法罢了。表 7.3.7 提供了两种盒子的方体要素，原点这一要素只被用在偏移计算中。

等式 7.3.5 和等式 7.3.6 介绍了两种最近点算法：

$$\mathbf{A} = \text{clamp}(\mathbf{X}, \mathbf{B}_{min}, \mathbf{B}_{max}) \qquad (7.3.5)$$
$$\mathbf{A} = \text{clamp}(\mathbf{X}, \mathbf{B}_{origin} + \mathbf{B}_{min}, \mathbf{B}_{origin} + \mathbf{B}_{max}) \qquad (7.3.6)$$

其作用如下：

（1）将收听者的位置钳制在 bounding box 的最大和最小值里（公式 7.3.5）；

（2）如果盒子被绑定在[0,0,0]这个原点周围，那么在进行钳制之前就要将盒子的原点引入进来（公式 7.3.6）。

适用范围

一些适用范围的例子：

环境特效：例如，你在一个长长的走廊的一端站着，在另一端打开了一扇通往外边的门，而外边在下着雨，那么你就会听到远处的雨声。当你向着门走的时候，雨声会变得越来越大，当你走出去的时候，雨声就会从各个方向传过来，振聋发聩。

方块迷题：每一次方块被推到了错误的地方，它都会闪烁着兰红交替的光并对玩家发出震耳欲聋的喊叫声。

7.3.6 总结

这些技巧，会帮助创造一种声音从整个表面传来的音效。但是要注意，这里只提到了几种形体，但并不意味着这就是我们所谈论的技巧所能应用到的全部情况，这些技巧还可以应用于其他情况，例如：多边形、平面、锥形等。如果是真的希望挑战一下的话，可以试试一些极其精细的关卡中的曲面，这些都是在游戏引擎中应该考虑到的。对于游戏来说，如果使用完全的物理 SDK，那么理论上你就能够为任何由简单形体派生出来的物体提供一个发生器。

如果你决定应用这个技巧，那就不要将发生器埋在冗杂的选项和菜单下面，要让策划人员能够很容易地找到它，不然的话就有违创建这些发生器的目的了（当然我们讨论过的方块迷题游戏除外，在那个例子里，把这些发生器藏得越深越好）。另外，在创造和放置发生器的时候要尽量发挥直觉的力量，尽力将这些发生器和游戏里物体的运动紧密联系在一起，或者利用一些脚本语言，这样他们才能够有进一步的发展和增强。

最后，如果对这些技巧有任何建议/问题/表扬或者问题，敬请 E-mail 作者。

7.3.7 光盘上的内容

在配套光盘中的音频文件夹下，可以找到一个小的示例程序，它演示了本文中提到的五种（两个方体看作一个）发生器。它使用 GLUT 作为框架，FMOD 作为音频处理[FMOD05]。FMOD 是一种最简单和最充分描述音频 API 特性的技术，而且它对于非商业性应用是免费的。可从查阅本文结尾处的参考文献获取详细信息。

最近点算法在 Primitives.cpp 中，发射管理器的代码在 Emitter.cpp 里。你可以试着自由地应用这些源代码，并且增加一些发射器类型去测试你的想法。

没有任何 licenses 附加在源码上，因此，一些你新编写的源码（和原始代码）可以自由使用在商业引擎上。

7.3.8 参考文献

[FMOD05] Firelight Technology. “The FMOD Sound System.” Available online at *http://www.fmod.org/*.

7.4 基于回馈延迟网络（FDN）的快速环境反响

Phenomic 游戏开发公司，Christian Schüler
cschueler@gmx.de

关于 DSP 为主题的邮件列表，每隔一段时间就会出现的一个同样的问题，“怎样计算一个反响的音效？”这些提问者在看了很多商业软件和硬件设备对反响音效的演示之后，都想知道有什么必不可缺的计算方法可以实现一个自己的声音信号的回响效果，就像在真实的房间里那样。

一般来说，对于该问题的最普遍答案就是“这个音效是一种古老的艺术”，因为理论上它在很早以前就可以实现了，现在人们所做的就是在那个理论的基础上修修补补而已。要不就是提示提问者对 Schroeder 早先提出的 comb filter 进行变形。但是尝试过想通过那些已经发布的资源来创造自己的反响特效的人很快就会失望地发现其实并不是十分有效。

这个技巧所针对的回馈延迟网络（FDN）是一个可以追溯到 20 世纪的古老方法（[Gerzon 1976]和[Stautner1982]中都曾经提到过它）。你可以在网上找到一个关于 FDN 的经典回顾[Smith Online]，尽管这个文章甚至没有一行代码。FDN 是所有延迟网络方法的（feedback network）的前辈，因为 FND 衍生了很多其他的算法，包括 Schroeder 的算法也可以作为 FDN 使用上的一个特例。另外，Jot（1993）还曾经用一个绑定了音调调节滤波器的 FDN 申请到了专利。

7.4.1 怎样将 FDN 作为资源来利用

首先，我们介绍的这个技巧将专注于介绍实现反响的算法，而不是如何利用一个特殊的音效 API。因此，即使那些电脑里没有预装反响解决方案的人也可以受益于这个技巧。而那些已经有了解决方案的人可以通过这个技巧来更深入地理解现有的解决方案，甚至重塑这个现有方案！

学习了这个技巧之后，你将可以很轻松地为 audio buffer 加上一个实用的函数。这个函数也可以作为一个插件用在任何与音效系统不发生冲突的地方，用来处理音频流。例如，对于 FMOD 这个多平台音效系统来说，就可以通过他的自定义 DSP 滤波器来把这个加进去。另外，DirectSound 也可以用到这个技巧，可以利用 `SetFX()` 函数为任意 `IDrectSoundBuffer8` 做一个 `Direct Media Object` 来实现这个技巧。

当然了，如果能有一套标准的音频渲染语言的话就再好不过了，因为它能够以一个标准的方式让硬件来执行用户的算法。但是正因为欠缺这样

一套机制，所以我们每次实现想要的音效的时候都不得不占用一部分 CPU 资源。好在音频数据对带宽的需求并不是太明显，另外没有专门音频卡的平台都不可避免地要用到 CPU，所以说实际上我们也没损失什么。

7.4.2　什么是反响

一个平滑的反响是回声被不断地多次叠加并被耳朵感知的过程。为了实现平滑的反响，必须要使声音的频谱尽量随机，而不应该是一个规律的图案。可以想象一下你在圣彼得大教堂内打一个响指，声音从你的指端发出来向四面扩散，很快他们就会碰到第一个障碍物，第一次回声反射回你的耳朵。然后一个又一个回声反射回来。再然后是第二次、第三次的非直接反射。每秒钟回声的平均个数按 t^2 增加，同时，每 Hz 的共鸣模式个数按 f^2 增加，图 7.4.1 表示一个理想矩形房间里的第一秒的冲激响应（impulse response）图。

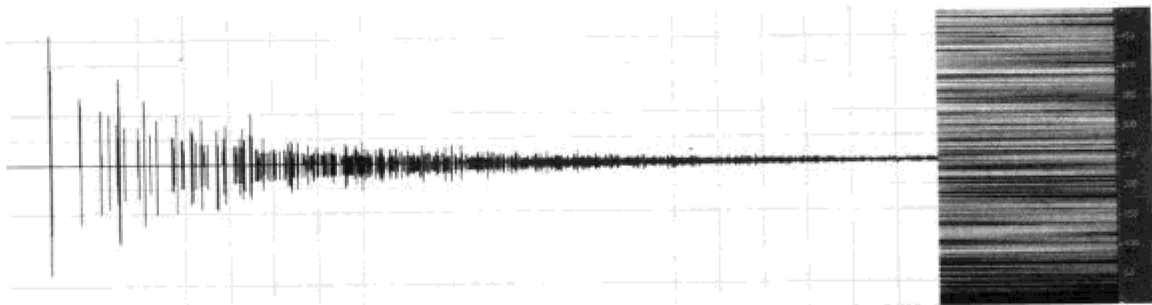


图 7.4.1　响应理想的矩形房间的冲激的时间频率图

如果想设计一个即时的反响模拟器来模拟这个过程的话，会遇到很多麻烦。首先作为一个即时的应用程序，你别无选择，只能建立这个过程的静态模型，因为如果要将这个过程整合到一个模拟的体中，也就是做一个 3D 模拟的话，即使最简单的屋子也会消耗极大以至于难以承受的资源，这种方法根本不能实现。FDN 是一系列一维向量的模拟，他能够模拟回声密度随着时间的增加，而不是随着频率增加的密度增加模型，后者只会给人制造出一种人造铃声的感觉。

7.4.3　回馈延迟网络（Feedback Delay Network）

图 7.4.2 介绍了 4 信道的 FDN 工作原理回路图。

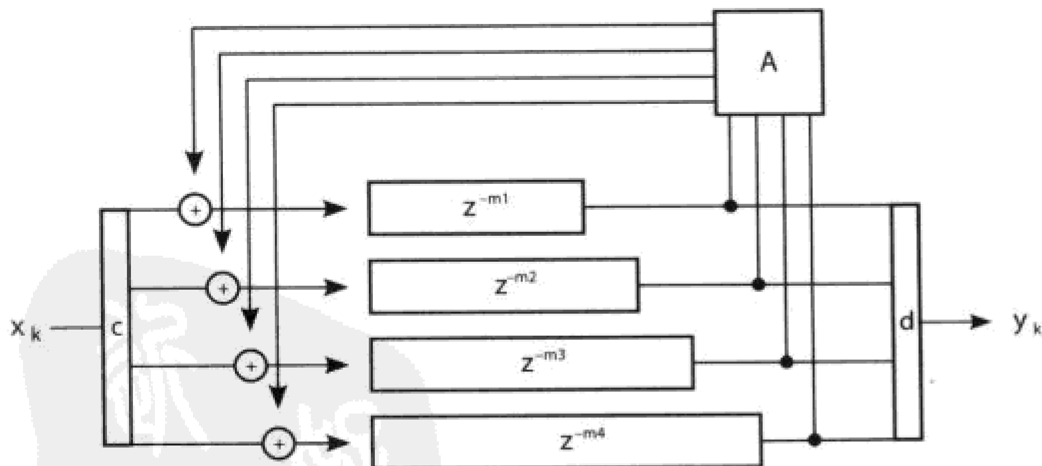
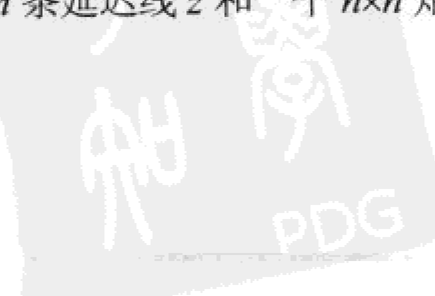


图 7.4.2　延时反应网络的电路图

FDN 的核心是由 n 条延迟线 z 和一个 $n \times n$ 矩阵组成的回馈循环。我们首先介绍一下 z 变换：


PDG

对于样本 m，z 的 $-m$ 次幂等于 m 的延迟，这是在数字音效领域经常会用到的。回馈矩阵中的元素 a_{ij} 控制着从线 i 中有多少信号能够被返回到到线 j 中。输入信号 x_k 在进入回馈循环之前转换成输入矩阵 c。同样，输出信号 y_k 是由一个经过各个信道加权求和得到的输出矩阵 d 表示。或者，可以通过收听者所在的 3D 空间的 4 个不相关信号（例如前、后、左、右）直接获取四条输出信道。

FDN 的一个主要特点就是信号在延迟线之间运行。从一个延迟线输出的回声会进入其余所有的延迟线，并在他们中再次做回响，然后所得的回声再进入所有其他的延迟线，如此循环。图 7.4.3 显示的是一个 4 信道 FDN 的第一秒的冲激响应（impulse response）图。从图中可以看到，图中所模拟的回声密度与前面矩形房间的回声密度十分相似。频谱图上显示了共鸣模型随着频率均一地展开，并没有出现高密度聚集的情况。这对于计算机运算的有效性是很有帮助的。

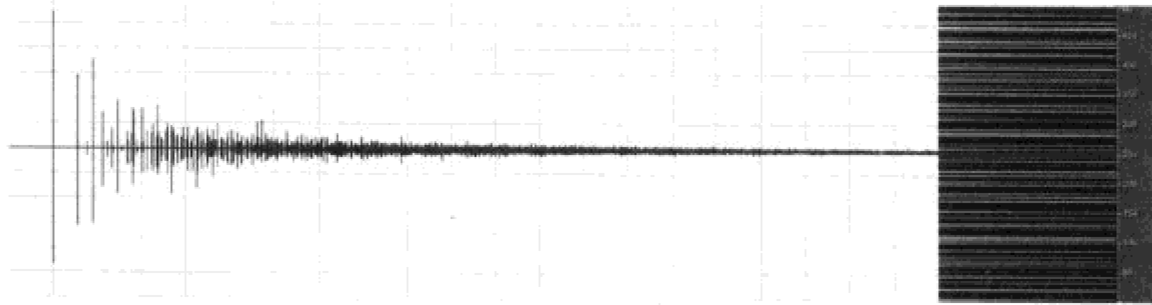


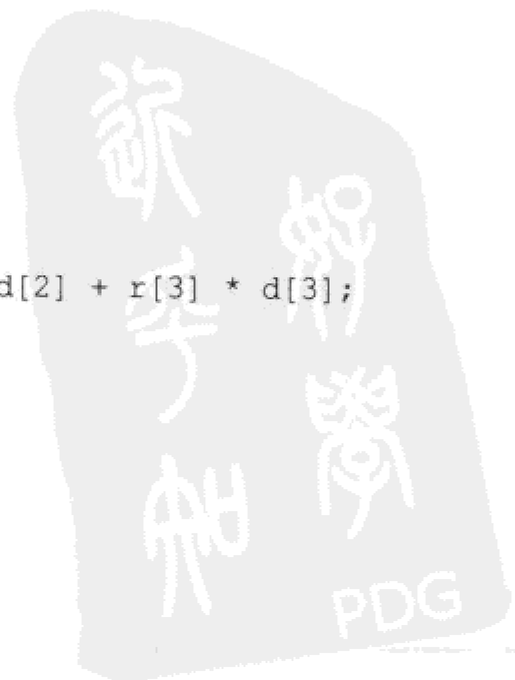
图 7.4.3 响应 FDN 冲激的时间频率图

现在我们来看看如何编码。首先，要建立一个 4 条延迟线的 FDN，可以通过创建一个简单的、先进先出的环缓冲区来实现。另外，我们假设所有的样本都是浮点数据。指令如下：

```
class FDN4{
    float *line[4];         // 4 条延迟线的内存
    unsigned size[4];       // 延迟线的大小(通常是 2 的次方)
    unsigned mask[4];       // 大小减 1，替换模块操作
    unsigned read[4];       // 读取 FIFO 指针
    unsigned write[4];      // 写 FIFO 指针
    float A[4][4];          // 反应矩阵
    float c[4];             // 输入矩阵
    float d[4];             // 输出矩阵

    /// ...
};
```

FDN 是依靠一个 `process ()` 函数而运作，进而实现了回路图中的运行机制。在这一函数中，基本上所有的指令都是对 4 元组的相同操作，所以我们用`//etc` 来做简化。

```
void FDN4::process( float *output, const float *input, unsigned n )
{
    for( unsigned k = 0; k < n; k++ )
    {
        // 步骤 1: 从延迟线读取信号
        float r[4];
        r[0] = line[0][ read[0] ];
        // 等等
        // 步骤 2: 接受输出矩阵并且输出
        output[k] = r[0] * d[0] + r[1] * d[1] + r[2] * d[2] + r[3] * d[3];

        // 步骤 3: 接受反应矩阵
```



```
            float w[4];
            w[0] = r[0] * A[0][0] + r[1] * A[1][0] + r[2] * A[2][0] + r[3] * A[3][0];
            // 等等

            // 步骤 4: 接受输入矩阵并且相加
            w[0] += input[k] * c[0];
            // 等等
            // 步骤 5: 随后插入

            // 步骤 6: 写信号到延迟线
            line[0][ write[0] ] = w[0];
            // 等等

            // 步骤 7: bump 指针
            read[0] = ( read[0] + 1 ) & mask[0];
            write[0] = ( write[0] + 1 ) & mask[0];
            // 等等
        }
    }
```

7.4.4　选择正确的回馈矩阵

这部分可能会涉及更多的数学知识。在[Smith Online]中，Smith 提到了回馈矩阵的特殊的类，单位矩阵($\mathbf{A}^{\mathrm{T}}=\mathbf{A}^{-1}$)，或者更有代表性的类——**无损的矩阵**。一个无损反馈矩阵在信号通过回馈回路时并不会改变信号的能量。而有着无损矩阵的 FDN 被称为无损原形（*lossless prototype*）[Smith 1996]或者参考滤波器（*reference filter*）[Jot1991]。无损矩阵的优点就是有些无损矩阵在进行矩阵向量乘法的时候可以将通常情况下的 $O(n^2)$次优化为 $O(n)$次。

一个 $n\times n$ 的回馈矩阵 $\mathbf{A}_{\mathrm{N}}$，当且仅当它有 N 个线性独立特征向量并且它的特征值是单元量值的时候，才是无损矩阵。所有的正交旋转矩阵，无论有没有反射，都可以看作是无损矩阵来使用。矩阵的密度，也就是矩阵中非零元素的多少，决定了矩阵中交叉耦合的多寡，进而影响回响建立的速度。

有一类很特殊的矩阵被称为 Household 反射，这类矩阵都是正交的并且包含的都是非零元素。假设 $\mathbf{U}_n$ 是一个 $n\times n$，且每个元素都设为 1 的矩阵，而 $\mathbf{I}_n$ 是一个单位矩阵，则：

$$\mathbf{A}_n=\mathbf{I}_n-\mathbf{U}_n$$

一个 Householder 反射对 4×4 矩阵特别合适，因为所有的元素都有着相同的量级。在矩阵向量相乘的时候可以通过一个 swizzled 向量相加和三个 swizzled 向量相减来实现：

$$A_4=\frac{1}{2}\begin{bmatrix}1 & -1 & -1 & -1\\ -1 & 1 & -1 & -1\\ -1 & -1 & 1 & -1\\ -1 & -1 & -1 & 1\end{bmatrix}$$

还可以尝试其他类型的矩阵。参见随书 CD-ROM 中附带的源代码，实现了许多可能反馈矩阵，它们都是归一化的。选择不同的反馈矩阵会得到完全不同的结果。最终，所有延时的长度之和决定了共鸣点的数量（系统的阶），而反馈矩阵则控制它们的轨迹。

7.4.5 选择正确的延迟长度

第一次回声的到达时间对于人耳估计周围环境（例如屋子的大小）是一个重要线索。这就意味着第一条延迟线的长度直接会影响模拟空间的尺寸。例如在 44.1 kHz 的取样频率下，对于每个样本声音会传播 3/4 厘米。一个 6×10 米的房间模型就可以将最短延迟线设定为 800 到 1 200 个样本长度。其他的延迟线可以根据系统的规则来进行筛选，因为我们急需每一个所能得到的共鸣模型。

接下来，要遵循一个很经典的需求——延迟长度应该是不对称的[Schroeder62]。另外一个最低需求就是延迟长度应该是互为素数的，同时一个好的不对称性应该可以通过图形表示出来。

图 7.4.4 显示了在一条一般时间线上四个不同的延迟的回声节拍。从图中可以看出，没有哪个延迟的整数倍和另外一个延迟的整数倍是重叠的。如果一旦碰巧发生了重叠，那么在对一个早发生和晚发生的重叠作取舍的时候，应该优先除去的是早发生的重叠。

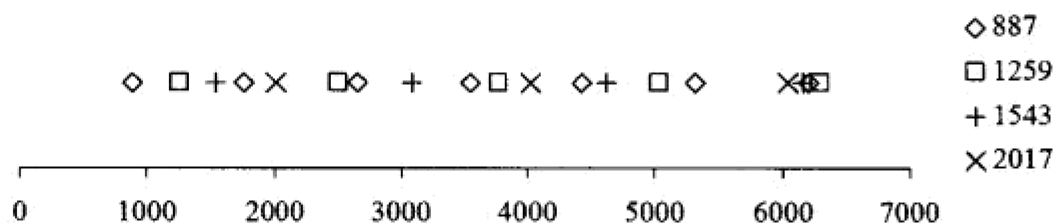


图 7.4.4 4 条延迟线的回声节拍

如果想要即时地判断空间的大小，那就需要为了这个需求建立一个算法来设定延迟的长度。一个可以接受的解决方法就是建立与已知的和需求空间的大小不对称的级数，并使其尽量接近于与其紧邻的素数。这里列了为 4 信道 FDN 建立的两套级数，两套级数的主要不同就是第三个延迟在第一个延迟第二次发生的之前或之后。

Series A: 1.0000, 1.5811, 2.2177, 2.7207.

Series B: 1.0000, 1.4194, 1.6223, 2.2401.

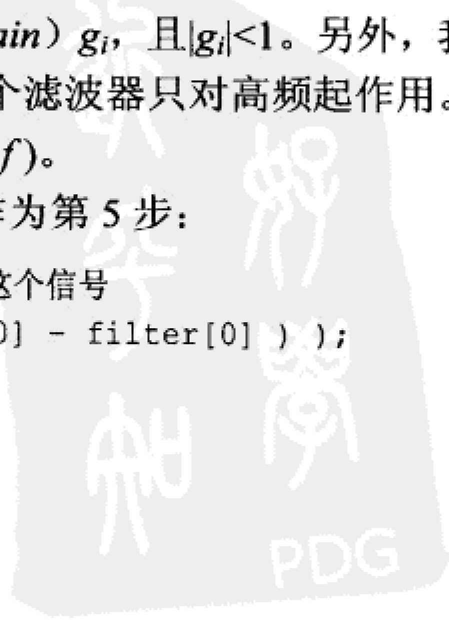
随书 CD-ROM 中提供了与图 7.4.2 类似的图表。

7.4.6 控制回响时间

到现在为止，我们的系统还只是一个一元的 FDN，不断累积的信号在回馈回路里永远循环，所以我们需要在其中加入能量损耗。由于大部分空间或多或少都会有高频阻尼，而我们希望允许低频维持的时间比高频更长，因此这个能量的损耗应该和频率联系起来。在这里，我们在每一个延迟线前插入一个标量增益（*scalar gain*）g_i，且$|g_i|<1$。另外，我们通过设定系数 α_i 来插入一个一阶滤波器（*first order filter*），这个滤波器只对高频起作用。标量增益和滤波器共同作用的产物就是一个全局频率相关增益 $g_i(f)$。

将以下代码插入到 FDN::process() 函数中作为第 5 步：

```
// 步骤 5: 在进入延迟线以前，以标量增益和滤波器处理这个信号
w[0] = g[0] * ( filter[0] += alpha[0] * ( w[0] - filter[0] ) );
```



```
// 等等
```

同时为 FDN 类增加下列新的命令：

```
float filter[4];   // 衰减滤波器的状态变量
float alpha[4];    // 滤波器的 alphas
float g[4];        // 标量形式的反馈增益
```

接下来的部分大概是整个技巧中涉及数学最多的部分，因为我们将为随机的回响长度推导 g_i 和 α_i。首先我们将控制参数设定为半衰期（*half life*），$\lambda(f)$这个参数的定义是当表示频率 f 的正弦曲线的振幅衰减为原来的一半时所花费的时间。我们定义 $\lambda_L=\lambda(0)$，为在频谱低端（f=0）的样本频率预期的半衰期；定义 $\lambda_H=\lambda(f_s)$，为在频谱高端（$f=f_s$）的半衰期（尼奎斯特极限，Nyquist limit）。由于信号不会持续削弱，而是按延迟长度 m_i 进行间隔削弱，所以 $g_i(f)$的对数一定要和这个时间间隔成常比。特定为，

$$g_i(f)=2$$

其中，m_i 是延迟线 i 的延迟长度，与 $\lambda(f)$单位相同。

到现在为止，一切进行得还比较顺利。早先提到的代码可能对于你会比较熟悉，类似于衰减平均，或者是别的名字的其他什么东西。因为它确实是对上次输出和当前输入进行混合的常用方法，并且满足这个循环关系：

$$y_k = \alpha x_k + (1-\alpha)\, y_{k-1}$$

其中，y_k 是滤波器的输出，x_k 是滤波器的输入，α 是滤波器的 alpha。通过利用 z 变换我们能够得到这个滤波器的系统转移函数 $H(z)$。z 变换通过乘以 z 来代替时间变换，所以我们得出 y_k 和 x_k:

$$y_k = \alpha x_k + (1-\alpha)\, y_k z^{-1}$$

$$H(z) = \frac{y_k}{x_k} = \frac{\alpha}{1-(1-\alpha)z^{-1}}$$

然后用 $z=\exp(j2\pi f/f_s)$来代表频率 f 的正弦信号，我们对任何参数 α 算出频率响应$|H(f)|$。在之前的叙述中，j 是单位虚数，这就是这里比较复杂的数学问题。因为我们必须考察两个特殊情况 f=0 和 $f=\frac{1}{2}f_s$，这两种情况分别对应 λ_L 和 λ_H，我们可以约分得出：

$$f=0 \rightarrow z=1 \rightarrow H(1)=1 \text{ 和 } f=\frac{1}{2}f_s \rightarrow z=-1 \rightarrow H(-1)=\frac{\alpha}{2-\alpha}$$

结果 $H(1)=1$ 说明滤波器并没有影响到低频，所以标量增益 g_i 可以直接从 λ_L 计算。对于频谱的高端，系数 α_i 就会起作用，使 $H(-1)$符合那些不是由 g_i 引起的衰变。

$$g_i = 2^{-m_i/\lambda_L},$$

$$\alpha_i=\frac{2\beta_i}{1+\beta_i}，且\ \beta_i=\frac{2^{-m_i/\lambda_H}}{g_i}$$

如果忘记了这些公式，那只需要通过计算 g_i 和 α_i，依然可以得到由 λ_L 和 λ_H 导致的能量损失。

7.4.7 sweeping 和细部延迟问题

我们希望 FDN 的延迟长度随着时间变化，这就是所谓的 sweeping。sweeping 的好处就是能够减少讨厌的固定波，或者通过一个静态的 FDN 来实现这一目的。sweeping 可以提高

输出的质量，或者减少必要的延迟线。通过按一个较低比率（0.5～2 Hz）来调节延迟长度的几个百分比可以达到某种特殊效果。如果在这个特效上做得太多的话，就会得到音质的变化。

当随着时间的变化平滑地改变延迟长度的时候，我们的延迟线的读取指针应该指向两个样本之间，这就是所谓的细部延迟（*fractional delay*）——一个类似于纹理滤波器的问题。而且音频是一系列标量，且标量对于混淆现象的干扰会比图形更敏感一些。所以说最近点采样并不是一个好的选择。另外，通过线性插值来实现的话就会导致模糊（blurring），通过图 7.4.5（中图）可以看出这会给我们带来多大的麻烦。从图上可以看出，这个 1 秒钟连续频谱的频率由于线性插值而损失了 8 kHz 以上的所有频率。

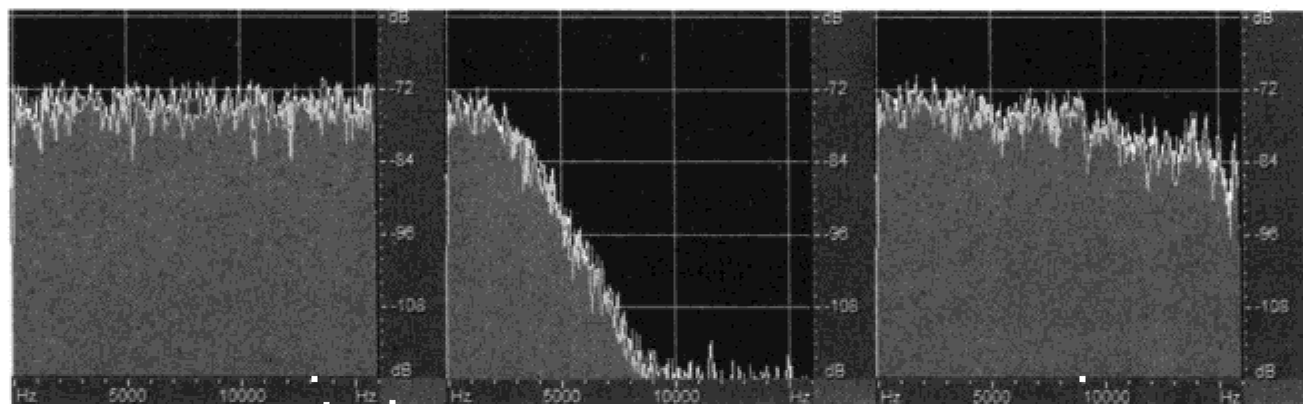


图 7.4.5 *t*=1 s 时，FDN 冲激响应的有代表性频率图。左图无 sweeping，中图为线性插值的 sweeping，右图为有补偿的 sweeping

现在已经有很多针对细部延迟的很复杂的算法。作为一个例子，Frenette 在他的 FDN 中使用了一个全通滤波器。可以研究一下这个全通滤波器是怎么解决线性插值问题的，然后调整我们的滤波器，以达到通过增加多余的高频来补偿由模糊导致的均值缺失。可以通过图 7.4.5（右图）看出它是如何发挥作用的。

首先，介绍一下 FDN 类的新命令，用来定位细部的读取指针。

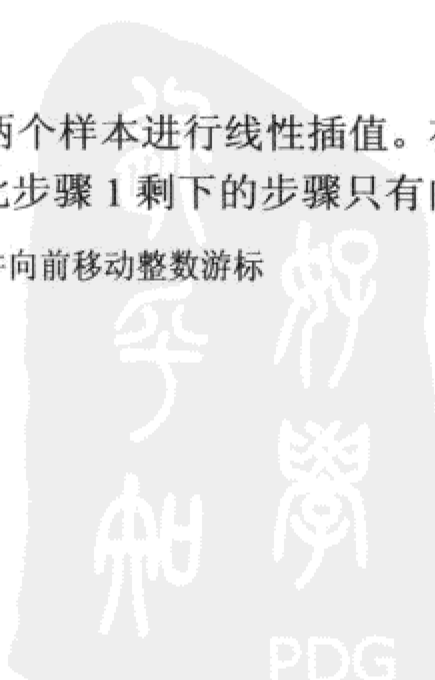
```
float p[4];        // 部分读游标的位置 (通常处于 0~1 之间)
float dp[4];       // sweeping 的速度
```

接下来，作为步骤 1 的替代。步骤 1a 将会整合 sweeping 的速率和时间，然后间断地放出整数的指针。

```
// 步骤 1a: 为读游标加入 sweeping 速度
p[0] += dp[0];
// 等等
read[0] = ( read[0] + int( floor( p[0] ) ) ) & mask[0];
// 等等
p[0] = p[0] - floor( p[0] );
// 等等
```

然后，在步骤 1b，对位于读取游标附近的两个样本进行线性插值。在线性插值中，我们还要改变了用来向前移动读取游标的代码（因此步骤 1 剩下的步骤只有向前移动整数游标）。

```
// 步骤 1b: 对读取游标附近的两个样本进行插值, 并向前移动整数游标
float r[4];
r[0] = line[0][ read[0] ];
// 等等
read[0] = ( read[0] + 1 ) & mask[0];
```



```
// 等等
r[0] += p[0] * ( line[0][ read[0] ] - r[0] );
// 等等
```

这里没有写入的一部分代码是关于缓慢调制 sweeping 速率的——一个简单正弦图或者其他的调制波形——这部分可以默认，就像所有的 16-64 样本一样。

现在，让我们来解决模糊问题。线性插值是一个有效的首位有穷冲激响应滤波器，可以被一个无穷冲激响应滤波器消除，就像我们已经在反馈回路里建立的那样。另外，我们需要知道如何调节滤波器的 alpha α_i 以补偿模糊效果。

线性插值的转换函数为：

$$y_k = (1-p)\,x_k + p\,x_{k+1}$$

$$H(z) = \frac{y_k}{x_k} = 1-p+p\,z$$

其中，p 是当前的读取指针的泛位置。如果 p=0，我们就对这些位置取整，并且不用考虑误差问题，如果 p=1/2，误差达到最大。现在假设 p 变化得非常快，以至于我们可以取得随机的 p 的净效应。在这种情况下，我们可以通过下面这个公式计算出绝对值$|H(z)|$的积分。

$$\int_0^1 |1-p+pz|\,dp = \int_0^1 \sqrt{\gamma p^2 - \gamma p + 1}\,dp\text{，其中 } \gamma = 2-2\cos(2\pi f/f_s)$$

在频谱的顶端，也就是f=f_s的位置，这个积分可以很简单地算出来，是 1/2。因此我们可以以 2 为因子试着调整滤波器的 alpha 值，得出平均模糊的补偿。

ON THE CD

不过不要忘了我们假设 p 是有效随机的，但如果不是的话，系统最终就会溢出。可以想象一下，一个泛读取指针一直被固定在 0 附近足够久的话；高频就会被放大至无穷。在实际操作中，补偿因子由于考虑到 sweeping 的速率 dp 而必须小于 2，这样的话完全补偿只有当 dp 足够高的情况下才会起作用。光盘中给出了一段程序来达到这个目标，通过一条平滑的饱和曲线，饱和度为 1.7，当 dp 低于 10^{-5} 时沉降为 dp^2。这段程序就是专门为了计算：

$$\beta_i' = \beta_i\left(1+\frac{0.7dp^2}{10^{-9}+dp^2}\right)$$

每次 dp 变化时，通过 β_i' 重新算出滤波器的 alpha β_i 的值。

7.4.8　总结和可能的改进

ON THE CD

至此，FDN 回响的一般概念就介绍完了，其中包括一些最重要的参数——回馈矩阵、延迟长度和回响次数。但是为了用好 FDN，还有很多底层的工作要做。光盘中有一段程序例子。这段程序能够实现我们前边提到的 4 信道 FDN 播放 stereo 波形文件，并且可以做不同的回响设定。另外为了使大家能够快速实践，光盘还提供了两个无回响波形样本，可以用来实验游戏音乐的混音。

在 3D 环境中，FDN 的 4 信道可以设为收听者空间的前、后、左、右。然后声音会根据它们传来的方向被分配给延迟线。这样做的目的是为了实现 *gradual decorrelation*，当早期的反射连接到音源位置时，晚期的回响会从各个方向共同发生。

多参数复合式空间需要使用复合式 FDN 来进行模拟，并且会加权计算模拟结果。调整单个 FDN 网络的参数没有什么现实的意义，但是有些时候会创造出一些有趣的特殊音效。

7.4.9 感谢

ON THE CD

特别感谢 Pex “Mahoney” Tufvesson 为我们提供了光盘中的所有 wave 文件，另外还要感谢 Peter Ohlmann 在 MFC 的一些细节问题上为我提供的帮助。

7.4.10 参考文献

[FMOD] “fmod” music and effects sound system. Available online at *http://www.fmod.org/*.

[Frenette00] Frenette, Jasmin. “Reducing Artificial Reverberation Algorithm Requirements using Time-Variant Feedback Delay Networks.” Masters Thesis. University of Miami, December 2000.

[Gerzon76] Gerzon M. A. “Unitary (energy preserving) multichannel networks with feedback.” In *Electronic Letters*, vol. 12, no. 11, pp. 278–279, 1976.

[Jot91] Jot, Jean Marc and Antoine Chaigne. “Digital delay networks for designing artificial reverberators.” In *Proc. 90th Conv. Audio Eng. Soc.*, Feb. 1991, preprint 3030.

[Jot93] Jot, Jean Marc and Antoine Chaigne. “Method and system for artificial spatialisation of digital audio signals.” United States Patent 5,491,754., Feb 1993.

[Mathworld] Erich Weisstein's world of mathematics. Available online at *http://www.mathworld.com*.

[MSDN] Microsoft Developer Network. Available online at *http://msdn.microsoft.com/default.aspx*.

[Moorer79] Moorer, J. A. “About this reverberation business.” In *Computer Music Journal*, vol. 3, no. 2, 1979.

[Schroeder62] Schroeder, J. M. “Natural sounding artificial reverberation.” J Audio Eng Soc, vol. 10, no. 3, 1962.

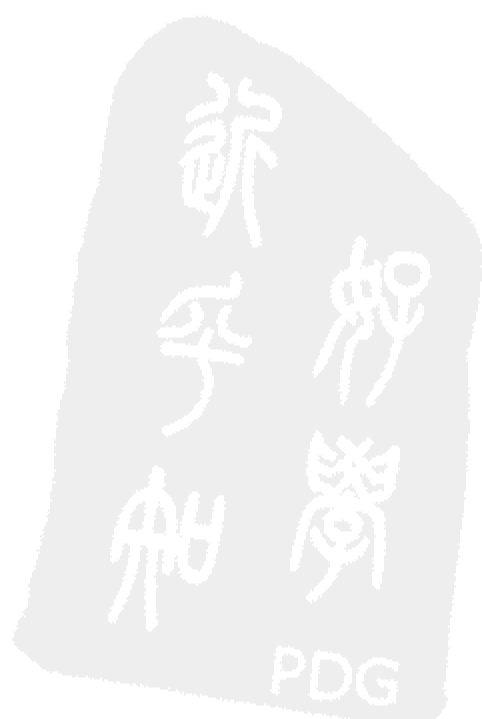
[Smith96] Smith, Julius O. and Davide Rochesso. “Circulant and Elliptic Feedback Delay Networks for Artificial Reverberation.” In *IEEE Transactions on Spech and Audio*, vol. 5, no. 1, pp. 51–60, Jan. 1996.

[SmithOnline] Smith, Julius O. “Physical Audio Signal Processing: Digital Waveguide Modeling of Musical Instruments and Audio Effects.” Available online at *http://www-ccrma.stanford.edu/~jos/waveguide/pasp.html*.

[Stautner82] Stautner, John and Miller Puckette. “Designing Multi-Channel Reverberators.” In

Computer Music Journal, vol. 6, no. 1, pp.52–65, Spring 1982.

[Välimäki00] Välimäki, V. and T. I. Laakso. "Principles of Fractional Delay Filters." *IEEE Int. Conf. on Acoustics, Speech and Signal Processing*, June 2000.



7.5 单演讲者语音识别简介

Julien Hamaide
julien_hamaide@hotmail.com

过去的若干年中，在 3D 和人工智能领域，人们投入了大量的努力，但是在人机交互领域，却很少有人问津。这就造成了现在大部分的游戏仍然在使用传统的人机交互手段：键盘、鼠标、摇杆或手柄。本文介绍的技巧会带领大家揭开一个全新的控制维度：声音。随着声音在一些游戏机系统上作为输入方式来使用，这个新式的人机互动正逐渐被玩家接受。另外，这种互动方式也可以作为新的游戏玩法的基础。

这篇文章会基于一些例子介绍单演讲者的语音识别技术。这个技术实施起来比较容易，流程简单，而且可以毫无困难地整合到现今的软件中。这个技巧并不是一个大而全的系统，而是针对问题提出的解决方案。随书附赠的光盘中会有一些演示软件，通过这些软件可以深入了解这个系统。

7.5.1 引言

语音是一个很复杂的信号。由人发出，并由非精确元素组成，这就导致很难对语音信号进行分析。这一部分会简单介绍一下语音信号的组成结构，并指出语音识别实现过程中的声学难点。

语音是由我们的大脑和身体制造的信号。大脑通过从耳朵接收到的反馈不断地更正和改编这个信号，我们才能顺利地说话。语音是空气循环通过喉咙里的声带时被制造出来的。当空气通过声带的时候，会引起声带振动，进而发出声音。

图 7.5.1 显示了语音产生的模型。

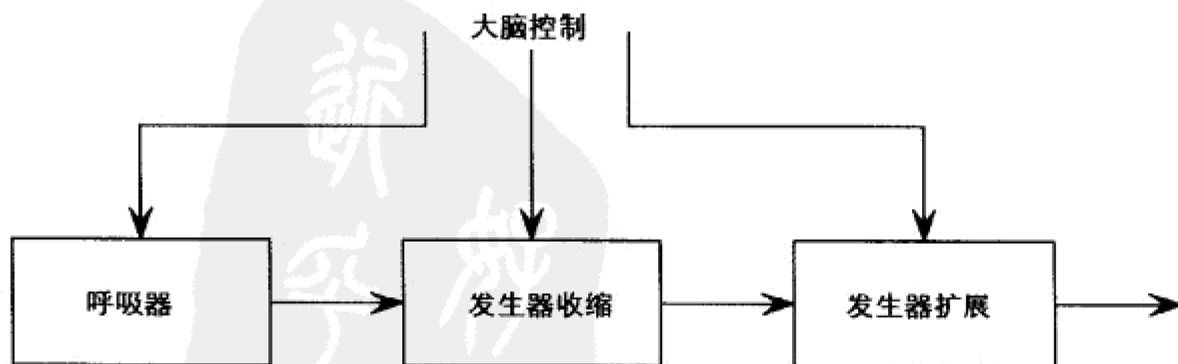


图 7.5.1 声音的产生模型

我们的语言会用到两种形态的声音：一种是嗓音，通过声带的振动发出（例如：元音）；还有一种是非嗓音，并不是由声带发出（例如：辅音中的 *t*，*v* 和 *s*）。图 7.5.2 显示了嗓音和非嗓音的频率图。音素 *u* 是一个嗓音，音素 *t* 是一个非嗓音。

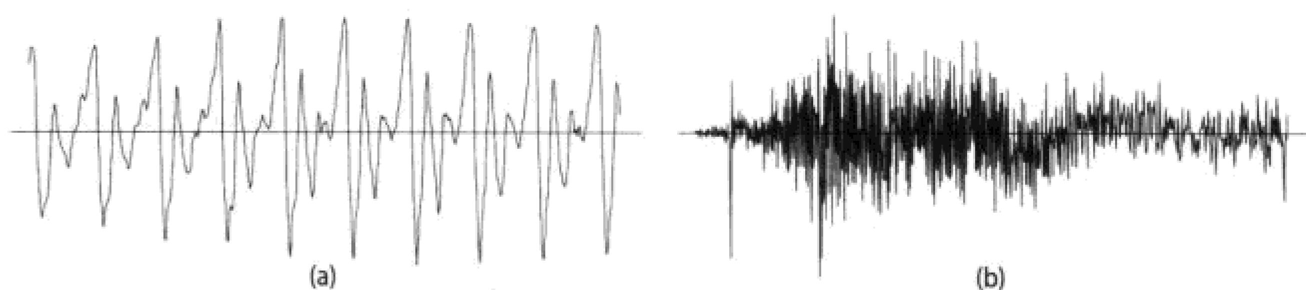


图 7.5.2　嗓音（a）与非嗓音（b）的比较

在处理语音的时候，一个主要问题就是信号产生过程中难以避免的极度不稳定性。那是因为实际上，语音的产生是大脑控制不同器官共同协作的过程：包括肺、声带、舌头等。由于我们的耳朵对于弱变调的感知能力较差，这些器官并不是被有意识地控制的，因此语音的产生就会不稳定。但是对于一个计算机系统来说，它需要尽可能独立地提取这些变调的特点。

对于语音信号来讲，一个重要的参数就是基本频率，也被称为音准。这个频率就是说话者声带振动的频率。对于一个成年男人来说，这个频率会在 70 到 250 Hz 之间；而对于女人和小孩，这个频率会在 200 到 600 Hz 之间。由于音准的变化幅度实在是太大，所以处理的时候一定要将其精确定位。

另外一个应该考虑到的重要因素就是可用带宽。一般来说，用于语音处理的带宽都会被固定在 3 400 Hz。根据 Shannon 法则，取样频率应该在 8 kHz 左右。大多数时间我们只能得到 44.1 kHz 的频率。为了速度或者简化的要求，8 820 Hz 的取样频率也会被用到，也就是初始频率的 1/5。其转换过程可以在光盘中找到。

7.5.2　识别系统

正如图 7.5.3 中显示的那样，整个识别链由两个完全不同的模块组成。第一部分将整个信号离散成一组离散的数值。然后这些离散的数值被输入到第二个模块中。这里会对这些数值与数据库里的参照系进行误差度量，然后选取误差最小的参照系，并提取其标识符。接下来，我们会对系统的执行方法与执行限制进行讨论，并提出这些限制的解决方法。

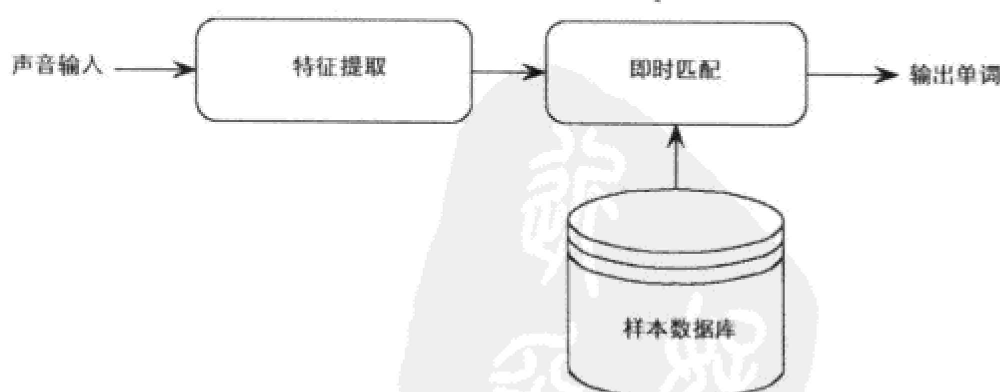


图 7.5.3　语音识别过程

这里讨论的方法必须在能够侦测到语音信号存在或者消失的情况下才能工作。为了节约

计算机的工作量，我们可以使用“按键聊天”系统。当玩家想要说话的时候，按这个键就可以了，这样能够简化语音识别系统。

7.5.3 特征提取

特征提取是语音识别过程中十分重要的一步。除了这篇本文讨论的，你还可以参考[Schalkwyk]的文章，那里有关于特征提取的全面介绍。实际上，特征提取就是将离散的声音信号片断转变成设定的数值的过程。这个片断的长度必须是固定的。而这个片断包含的信号也要是假性稳定的。一般有经验的人会每 10 ms 提取一段 20～30 ms 的片段。每 10 ms 产生一个特征向量。这样一个个单词就变成了一段段向量序列。在对任何一个片断进行处理之前，要应用一个预加重滤波器来对信号进行平坦处理（公式 7.5.1），再用一个 Hamming's window 来消除边界影响。

$$y(n)=x(n)-0.9*x(n-1) \quad (7.5.1)$$

图 7.5.4 显示了源片段(a)，Hamming's window(b)和经过处理的片段。我们接下来会介绍对这个 30 ms 信号的特征提取。

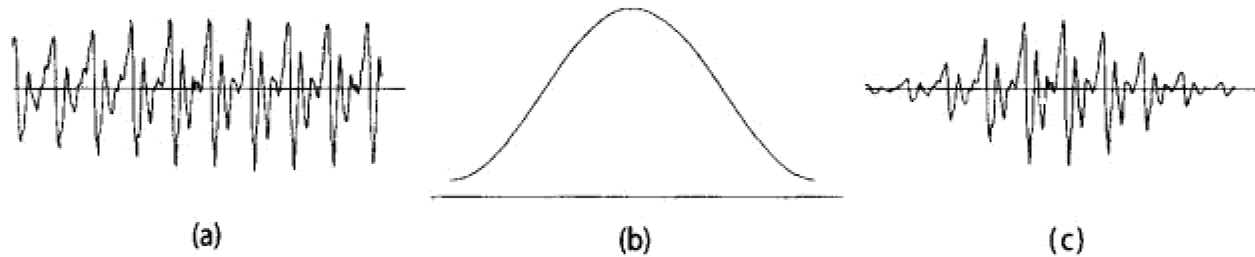


图 7.5.4 （a）信号片段，（b）海明窗（Hamming's window），（c）经海明窗处理后的信号片段

1. LPC

LPC（Linear Prediction Coding）分析，曾经在[Edwards 02]这篇文章中使用过，通过应用一个线性系统来创造语音模型。这个方法通过一个线性组合从最终样本 t 来推出最佳样本 $x(n)$，而这里 t 的作用就是达到期望的精确度。等式 7.5.2 为评估函数。这里的 a_i 是表示从线性预测中得到的系数。

$$x(n)=e(n)+\sum_{i=1}^{t}=a_i x(n-i)\text{, } e(n)\text{是剩余误差} \quad (7.5.2)$$

另外，方程 7.5.3 说明了线性滤波器的冲激响应就是我们所研究的信号的频谱包络线。

$$\tilde{x}(n)=\sum_{i=1}^{t}-a_i x(n-i) \quad (7.5.3)$$

图 7.5.5 将这一情况进行了图形化。这种表示方法会因为独立于信号的基本频率而具有一定的优势。这个序列的阶数代表了在进行线性预测的时候使用的样本数量。系数 a_i 等于序列的阶数。在识别过程中，会对阶数等于样本频率（kHz）+4 的样本进行分析。在我们这个例子里，阶数为 13 的样本会被使用，因为更高阶的参数并未携带足够的信息，将被忽略。

细节的计算过程可以从[Edwards 02]中找到，此处不再赘述了，因为代码也包含在光盘中。

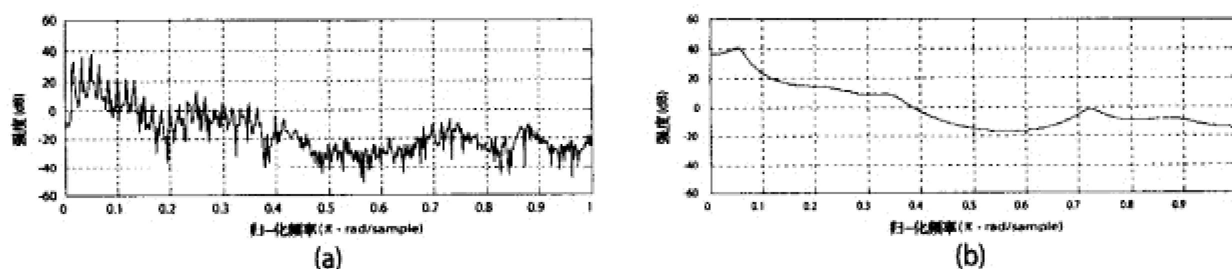


图 7.5.5　（a）信号的频谱，（b）LPC 滤波器的频率响应

不过这里的系数 a_i 是不稳定的，它会随着信号的细微变化而发生很大的变化。所以我们需要通过等式 7.5.4 把系数 a_i 转换成倒谱系数 c_i。其中，p 是 LPC 分析里的阶数。这个转换能够帮助我们得到适合于语音识别过程的更稳定的系数。

$$c(i)=a_i+\sum_{k=1}^{i-1}\left(\left(1-\frac{k}{i}\right)*a_k*c_{i-k}\right);\ 1<i<p \qquad (7.5.4)$$

通过利用这些系数，尽管可以得到满意的结果，但是并没有考虑到人耳朵的生理特点。所以再介绍另外一种方法，它可能会多消耗一点系统资源。这种方法被称为感知线性预测（Perceptual Linear Prediction），简称 PLP。

2. PLP

这种方法其实就是为了适应人的耳朵而改编的 LPC 方法。一般，健康人的耳朵呈现以下两个特点。

- 随着频率的升高，人耳处理声音频率的能力会下降。
- 人耳对中段频率更加敏感。

这部分技术需要花费较多的篇幅来解释各种细节。考虑到其数学上的复杂性，有兴趣的读者可以参考[Boite00]和[Costache02]。

与这部分技术相关的代码可以在随书附赠的光盘中找到。

3. 信号能量

最后一个涉及的参数就是信号的方差 σ_x^2，它代表信号片段的能量或强度。这个参数可以帮助我们区分出说话时的一些比较显著的部分：比如特殊的重音或者强有力的音素等。等式 7.5.5 提供了评估这种强度差别的公式。

$$\alpha_x^2=\frac{1}{N-1}\sum_i^N\left(x(i)-\bar{X}\right)^2$$
$$\bar{X}=\frac{1}{N}\sum_i^N\left(x(i)\right) \qquad (7.5.5)$$

尽管强度是一个很重要的系数，但是以当前的形式还无法被用在语音识别中。在实际应用中，如果玩家说话声音特别大，能量增加将超出参考值。因此，我们就必须使用能量的相对值（公式 7.5.6），而不是绝对值。

$$\Delta\sigma_x^2(n)=\frac{\sigma_x^2(n)-\sigma_x^2(n-1)}{\sigma_x^2(n)} \qquad (7.5.6)$$

4. 总结

为了在视频游戏中实现语音识别，可以使用两种方式：LPC 分析，这种方法在计算机运

算时是比较节约的；PLP 分析，这种方法也许会花费更多的计算机运算，但是会得到比较好的结果。他们都会创造出 13 个系数。在做选择的时候要顾及到现有的计算机的计算能力和期望达到的精确度。

无论选择哪种方法，其中的向量都会包含 14 个系数（13 个来自于 LPC 或者是 PLP，一个来自于强度），这些系数组成了信号片段的确认信息。它们会作为声音向量在下面的小节中讨论。

7.5.4 即时配对匹配

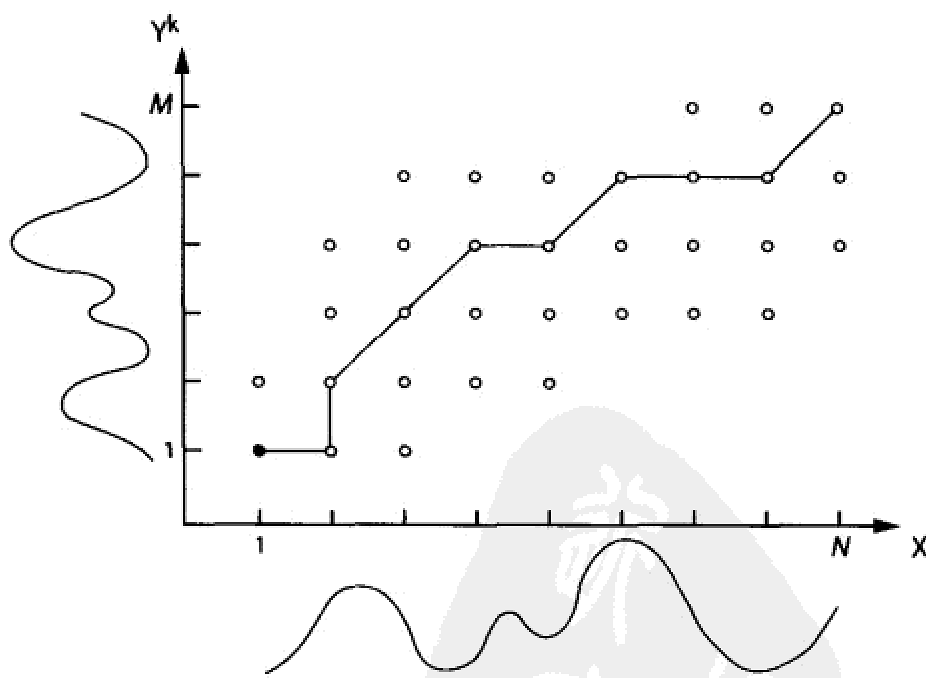
现在特征向量组已经准备好了，我们接下来就要利用它们从参照系中找到最合适的一个来代表这个单词。为了度量这两个单词发音之间的误差（被称为失真），我们要在两个声音向量间定义距离。为了简化，这里借用了 Euclidian 差距公式（公式 7.5.7）。其余的差距公式，例如 city block 差距公式，也可以使用。

$$d(x,y)=\|x-y\|^2=\sqrt{\sum_{n=1}^{N}(x_n-y_n)^2} \tag{7.5.7}$$

首先，我们要对 10 ms 长度的信号片段进行向量和向量的比较，并对每一组向量的误差进行求和。不过很难解决的就是一个单词不会总以一个我们期待的固定速度被说出来。或者说信号会主观地暂时性压缩和扩大。因此，需要一个技术来自适应声音向量发射频率。这个技术被称为动态时间变形（Dynamic Time Warping）。

DTW 的目标就是找到两个单词的声音向量分布的最佳轨迹，也就是说一个累计误差最少的轨迹。

被识别的单词（信号）被设为 X，而数据库中作为参照系的单词设为 Yk。图 7.5.6 表示了这个轨迹的由来，每一条坐标轴代表需要比较的那两个单词，而每一个点则代表一对声音向量。



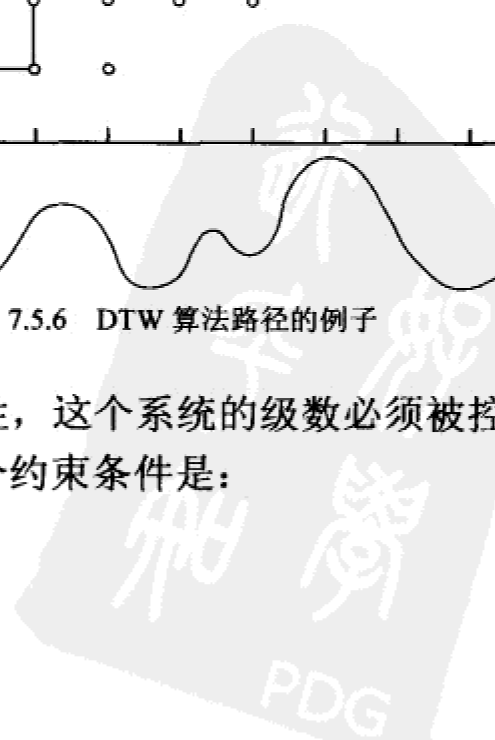


图 7.5.6 DTW 算法路径的例子

由于考虑到了语音产生的局限性，这个系统的级数必须被控制起来。以下约束条件是为了符合说话的本质而建立的。这三个约束条件是：

单调进化：声音向量代表的是一个暂时性的概念，所以说不可能回归。

边界条件：轨迹开始于（0,0），结束于（N,M），N 和 M 分别是参照系和被辨别的语音的频率大小。

级数约束：图 7.5.7 所示是一些约束的限制。只有如图中箭头显示的才可以替换。第一个限制会让系统全空。第二个限制在垂直和水平级数之前还需要一个对角的级数。

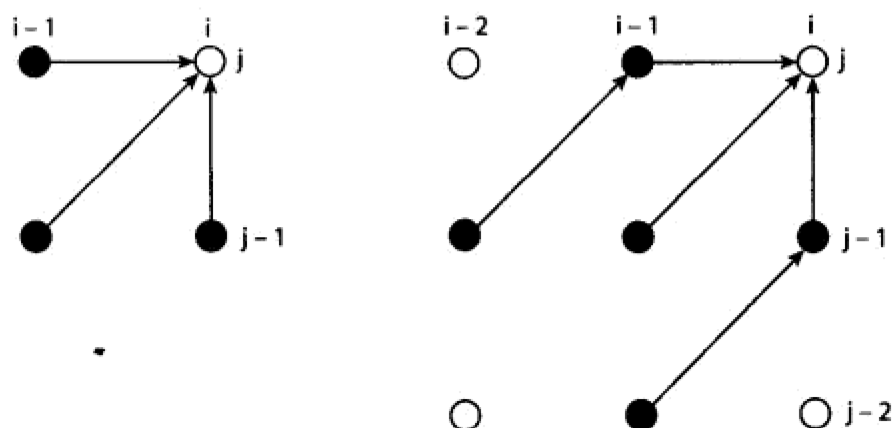


图 7.5.7　级数约束：黑点二起始点

路径算法

为了计算最小误差，我们将用到公式 7.5.8。（i,j）代表一对特征向量，i 表示要测试的向量的临时指数，j 表示参照向量的临时指数，$D(i,j)$表示离（i,j）这一点的最短距离。$p(i,j)$表示可能的源(i,j)（考虑到图 7.5.7 的级数约束）。

$$D(i,j) = d(i,j) + \min_{p(i,j)}\{D(p(i,j))\}$$
$$d(i,j) = \|X_i - Y_j^k\|^2 \qquad (7.5.8)$$

我们已经逐步地成功得出每一个(i,j)的误差积累。其实并不需要十分精确地知道具体的分布轨迹，只要知道最终的值就可以了。

当比较完参照系中每一个向量后，我们就可以挑出与被比较单词向量失真最小的作为我们的结果。另外要对所产生的最大失真进行修正，以避免说话者所说的语音不在参照数据库内。

ON THE CD

关于 DTW 的更多研究，可参考[Keogh03]，另外关于 DTW 实现的代码，可以参照随书的光盘。

7.5.5　训练

在开始进行语音识别之前，还要进行一系列数据库的工作。因此，说话人会被要求说一段话来确定单位时间内所说的音素的个数。然后音频向量的序列会从这段话中提取出来并储存，主要是为了：第一，我们不必再对数据库进行特征相量提取。第二，存储向量的空间要比存储声音信号样本的空间小很多。如果录音环境比较好，而且这段话的词汇不是很生僻，那么两到三次录音就足够了。但是如果处在一个嘈杂的环境中，或者词汇生僻的话，那么可能会需要每一个词要录 10 个例子。

7.5.6 局限性

这是一个比较简单的系统，而且如果使用条件比较合适的话（在一个安静的环境中，使用足够好质量的麦克风），那么结果将会接近 100%。尽管如此，它还是会有一些局限性。

首先，这个系统对环境噪音比较敏感。因此，使用的时候应该在一个安静的房间中。有些单进程的技巧可以用来消除这些环境噪音。比如背噪是来自游戏本身的话，那可以很简单地直接将它们从信号中减去。

其次，这个识别系统是和说话者紧紧联系的。还有些技术不用考虑说话者，但是那会需要更多的计算和消耗更多的系统资源。

最后，这个系统的复杂性依靠数据库中的单词量。随着参照系中用于和输入语音进行比较的单词量的增加，数据库必须随着增大。解决这个方式的惟一方法就是尽量从信号中提取音素进行比较，而不是比较单词。

另一个较大的局限性就是对比数据库的建立——使用者会被要求录入命令单词的例子。在大部分情况下，三次录音就足够了。

对于这些制约因素的解决方案就是使用更复杂的系统，基于概率模型、神经网络或者 Hidden Markov 模型。这些复杂系统不在本技巧讨论范围之内，如果读者感兴趣，可参考[Boite00]。

7.5.7 结论

本文介绍了语音识别系统的基本理论及系统。尽管我们讨论的这个系统比较初级，但是其中提到的特征提取是所有系统所共有的。我们所提出的这个系统简单、快捷、易于实现，完全可以应用于视频游戏中。因为在理想条件下，这个技巧是 100%有效的。尽管这个技巧有其局限性，但是它为初涉语音识别应用程序提供了一个简单的方法。当然，如果要把它整合到更为复杂的系统中，肯定需要第三方软件，因为语音识别的实现不是一个简单的问题。随着声音录入系统被应用在越来越多的主机上，我相信，语音识别将来一定会在游戏中扮演一个十分重要的角色。

7.5.8 参考文献

[Boite00] Boite R., H. Bourlard, T. Dutoit, J. Hancq, and H. Leich. “Traitement de la Parole.” *Presses Polytechniques et Universitaires Romandes*, 2000.

[Costache02] Costache, Gavat, Raileanu. “Voice Command System.” In *International Workshop Trends and Recent Achievements in Information Technology*. Cluj Napoca, Romania, May 2002.

[Edwards02] Edwards E. “Linear Predictive Coding for Voice Compression and Effects.” In *Game Programming Gems 3*, 613–621. Charles River Media, Inc., 2002.

[Keogh03] Keogh, E., and J. Pazzani. “Derivative Dynamic Time Warping.” In *Learned Representations in AI*. Division of Computer and Information Sciences, Rutgers University of New Jersey, 2003.

[Schalkwyk] Schalkwyk, J. “Feature Extraction.” Available online at *http://www. cslu.ogi.edu/toolkit/old/old/version2.0a/documentation/csluc/node5.html*. November 27, 1996.