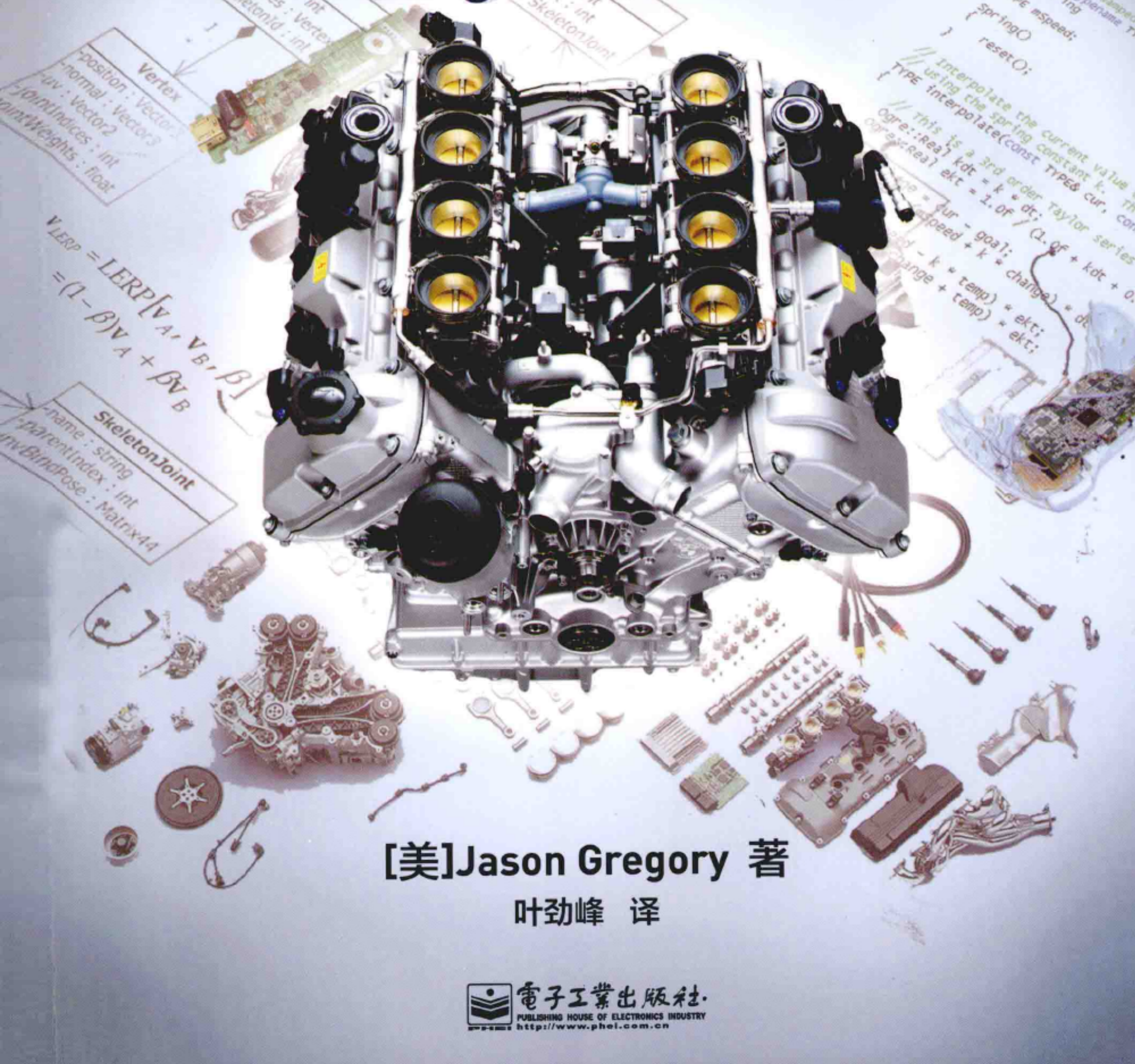
CRC Press
Taylor & Francis Group

Broadview®
www.broadview.com.cn

Jeff Lander，Matt Whiting 作序力荐

游戏引擎架构

Game Engine Architecture



[美]Jason Gregory 著

叶劲峰 译

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

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

本书同时涵盖游戏引擎软件开发的理论及实践，并对多方面的题目进行探讨。本书讨论到的概念及技巧实际应用于现实中的游戏工作室，如艺电及顽皮狗。虽然书中采用的例子通常依据一些专门的技术，但是讨论范围远超于某个引擎或API。文中的参考及引用也非常有用，可让读者继续深入游戏开发过程的任何特定方向。

本书为一个大学程度的游戏编程课程而编写，但也适合软件工程师、业余爱好者、自学游戏程序员，以及游戏产业的从业员。通过阅读本书，资历较浅的游戏工程师可以巩固他们所学的游戏技术及引擎架构的知识。专注某一领域的资深程序员也能从本书更为全面的介绍中获益。

内容包括：

- 游戏开发中的大规模C++软件架构
- 游戏编程所需的数学
- 供调试、源代码控制及性能剖析的游戏开发工具
- 引擎基础系统、渲染、碰撞、物理、角色动画、游戏世界对象模型等引擎子系统
- 多平台游戏引擎
- 多处理器环境下的游戏编程
- 工作管道及游戏资产数据库

作者简介

Jason Gregory在1994年开始任职专业软件工程师，自1999年3月开始在游戏产业中任职软件工程师。在圣迭哥Midway Home Entertainment公司开始游戏编程的他，为《疯狂飞行员（Freaky Flyers）》及《Crank the Weasel》开发PlayStation 2/Xbox上的动画系统。在2003年，他转到洛杉矶艺电，为《荣誉勋章：血战太平洋（Medal of Honor: Pacific Assault）》开发游戏引擎及游戏性技术，并在《荣誉勋章：空降神兵（Medal of Honor: Airborne）》中担任首席工程师。他现时是顽皮狗公司的通才程序员，为《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》及《神秘海域：纵横四海（Uncharted: Among Thieves）》开发引擎及游戏性软件。他也在南加州大学教授游戏技术的课程。

译者简介

叶劲峰（Milo Yip）从小自习编程，并爱好计算机图形学。上中学时兼职开发策略RPG《王子传奇》，该游戏在1995年于台湾发行。其后他获取了香港大学认知科学学士、香港中文大学系统工程及工程管理哲学硕士。毕业后在香港理工大学设计学院从事游戏引擎及相关技术的研发，职至项目主任。除发表学术文章外，也曾合著《DirectX 9游戏编程实务》。2008年往上海育碧担任引擎工程师开发《美食从天而降（Cloudy with a Chance of Meatballs）》Xbox360/PS3/Wii/PC，2009年起于麻辣马开发《爱丽丝：疯狂回归（Alice: Madness Returns）》Xbox360/PS3/PC，2011年加入腾讯互动娱乐引擎技术中心担任专家工程师，所研发的技术已用于《斗战神》、《天涯明月刀》、《众神争霸》等项目中。

策划编辑：张春雨
责任编辑：付　睿
封面设计：侯士卿

上架建议：游戏开发/游戏设计

ISBN 978-7-121-22288-7
9 787121 222887 >
定价：128.00元

游戏引擎架构

Game Engine Architecture

[美]Jason Gregory 著

叶劲峰 译

電子工業出版社

Publishing House of Electronics Industry

北京•BEIJING

内容简介

本书同时涵盖游戏引擎软件开发的理论及实践，并对多方面的题目进行探讨。本书讨论到的概念及技巧实际应用于现实中的游戏工作室，如艺电及顽皮狗。虽然书中采用的例子通常依据一些专门的技术，但是讨论范围远超于某个引擎或API。文中的参考及引用也非常有用，可让读者继续深入游戏开发过程的任何特定方向。

本书为一个大学程度的游戏编程课程而编写，但也适合软件工程师、业余爱好者、自学游戏程序员，以及游戏产业的从业人员。通过阅读本书，资历较浅的游戏工程师可以巩固他们所学的游戏技术及引擎架构的知识，专注某一领域的资深程序员也能从本书更为全面的介绍中获益。

Jason Gregory: Game Engine Architecture, First Edition. ISBN: 978-1-5688-1413-1

版权贸易合同登记号图字：01-2010-4326

图书在版编目（CIP）数据

游戏引擎架构/(美)格雷戈瑞(Gregory,J.)著；叶劲峰译.—北京：电子工业出版社，2014.2

书名原文: Game engine architecture

ISBN 978-7-121-22288-7

Ⅰ. ①游… Ⅱ. ①格…②叶…Ⅲ. ①三维动画软件—游戏程序—程序设计Ⅳ. ①TP391.41

中国版本图书馆CIP数据核字（2014）第002560号

策划编辑：张春雨

责任编辑：付　睿

印　　刷：北京市新伟印刷有限公司

装　　订：三河市皇庄路通装订厂

出版发行：电子工业出版社

北京市海淀区万寿路173信箱　　邮编：100036

开　　本：787×980　1/16　印张：49.75　字数：1093.4千字

印　　次：2014年3月第3次印刷

定　　价：128.00元

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

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

服务热线：（010）88258888。

推荐序1

最初拿到《Game Engine Architecture》一书的英文版，是编辑侠少邮寄给我的打印版。他建议我接下翻译此书的合同。当时我正在杭州带领一个团队开发3D游戏引擎，我和我的同事都对这本书的内容颇有兴趣，两大本打印的英文书立刻在同事间传开。可惜那段时间个人精力顾及不来，把近千页的英文读物精读而后翻译成中文对个人的业余时间是个极大的挑战，不能担此翻译任务颇为遗憾。

不久以后听说Milo Yip（叶劲峰）已开始着手翻译，甚为欣喜。翻译此巨著，他一定是比我更合适的人选。我和Milo虽未曾蒙面，但神交已久。在网络上读过一些他的成长经历，和我颇为相似，心有戚戚。他对游戏3D实时渲染技术研究精深为我所不及，我们曾通过Google Talk讨论过许多技术问题，他都有独到的见解。翻译工作开始后，Milo是香港人，英文技术术语在香港的中文译法和大陆的有许多不同。但此书由大陆出版社出版，考虑到面对的读者主要是大陆程序员，Milo希望能更符合大陆程序员的用词习惯，所以在翻译一开始就通过Google Docs创建了协作页面，邀请大家共同探讨书中技术名词的中译名。从中我们可以一窥他作为译者的慎重。

三年之后，有幸在出版之前就拿到了完整的译本。这是一本用LaTeX精心排版的800页的电子书，我只花了一周时间，几乎是一口气读完。流畅的阅读享受，绝对不仅仅是因为原著精彩的内容，精美的版面和翔实的译注也加了不少分。

在阅读本书的过程中，我不只一次地获得共鸣。例如在第5章的内存管理系统的介绍中，作者介绍的几种游戏特有的内存管理方法我都曾在项目中用过，而这是第一次有书籍专门将这些方法详尽记录；又如第11章动画系统的介绍，我们也同样在3D引擎开发过程中改进原有动画片段混合方法的经历。虽然书中介绍的每个技术点，都可能可以在某篇论文，某本其他的书的章节，某篇网络blog上见过，但之前却无一本书可以把这些东西放在一起相互参照。对于从事游戏引擎开发的程序员来说，了解各种引擎在处理每个具体问题时的方案是相当重要的。而每种方案又各有利弊，即使不做引擎开发工作而是在某一特定游戏引擎上做游戏开发，从中也可以理解引擎的局限性以及可能的改进方法。尤其是第14章介绍的对游戏性

相关系统的设计，各个开发人员几乎都是凭经验设计，很少见有书籍对这些做总结。对于基于渲染引擎做开发的游戏程序员，这是必须面对的工作，这一章会有很大的借鉴意义。

本书作者是业内资深的游戏引擎开发人，他所参于的《神秘海域》和《最后生还者》都是我的个人最爱。在玩游戏的过程中，作为游戏程序员的天性，自然会不断地猜想各个技术点是如何实现的，背后需要怎样的工具支持。能在书中一一得到印证是件特别开心的事情。作者反复强调代码实践的重要性，在书中遍布着C++代码。我不认为这些代码有直接取来使用的价值，但它们极大地帮助了读者理解书中的技术点。书中列出的顽皮狗工作室用lisp方言作为游戏配置脚本的范例也给我很大的启发，有了这些具体的代码示例以及作者本身的一线工程师背景，也让我确信书中那些关于主机游戏开发相关等，我所没有接触过的内容都也绝非泛泛而谈。

国内的游戏开发社区的壮大，主要是随最近十年的MMO风潮而生。而就在大型网络游戏在中国有些畸形发展，让这类游戏偏离电子游戏游戏性的趋势时，我们有幸迎来了为移动设备开发游戏的大潮。游戏开发的重心重新回到游戏性本身。我们更需要去借鉴单机游戏是如何为玩家带来更纯粹的游戏体验，我相信书中记录的各种技术点会变的更有帮助。

云风 @简悦云风

推荐序2

在我认识的许多游戏业开发同仁中，只有少数香港同胞，Milo Yip（叶劲峰）却正是这样一位给我印象非常深刻的优秀香港游戏开发者。我俩认识，是在Milo加入腾讯互动娱乐研发部引擎技术中心后，说来到现在也只是两年多时间。其间，他为人的谦逊务实，对待技术问题的严谨求真态度，对算法设计和性能优化的娴熟技术，都为人所称道。Milo一丝不苟的工作风格，甚至表现在对待技术文档排版这类事情上（Milo常执著地用LaTeX将技术文档排到完美），我想这一定是他在香港读大学、硕士及在香港理工大学的多媒体创新中心从事研究员，一贯沿袭至今的好作风。

我很高兴腾讯游戏有实力吸引到这样优秀的技术专家；即使在其已从上海迁回香港家中，依然选择到深圳腾讯互动娱乐总部工作。叶兄从此工作日每天早晚过关，来往香港和深圳两地，虽有舟车劳顿，但是兼顾了对家庭的照顾和在游戏引擎方面的专业研究，希望这样的状况是令他满意的。

认识叶兄当时，我便知道他在进行Jason Gregory所著《游戏引擎架构》一书的中译工作。因为自己从前也有业余翻译游戏开发有关书籍的经历，所以我能理解其中的辛苦和责任重大，对叶兄也更多一分钦佩。我以为，本书以及本书的中文读者最大的幸运便是，遇到叶兄这位对游戏有着如同对家对国般强烈责任感，犹如“游戏科学工作者”般的专业译者!

现在（2013年年末）无疑是游戏史上对独立游戏制作者最友好的年代。开发设备方便获得（相对过往仅由主机厂商授权才能获得专利开发设备，现在有一台智能手机和一台个人电脑就可以开发）、技术工具友好、调试过程简单方便，且互联网上有丰富的例程和开源代码参考，也有网上社区便于交流。很多爱好者能够很快地制作出可运行的游戏原型，其中一些也能发布到应用商店。

但是不全面掌握各方面知识，尤其是游戏引擎架构知识，往往只能停留在勉强修改、凑合重用别人提供的资源的应用程度上，难以做极限的性能改进，更妄谈革命式的架构创新。这样的程度是很难在成千上万的游戏中脱颖而出的。我们所认可的真正的游戏大作，必定是在某方面大幅超越用户期待的产品。为了打造这样的产品，游戏内容创作者（策划、美术

等）需要“戴着镣铐跳舞”（在当前的机能下争取更多的创作自由度），而引擎架构合理的游戏可以经得起——也值得进行——反复优化，最终可以提供更多的自由度，这是大作出现的技术前提。

书的作者、译者、出版社的编者，加上读者，大家是因书而结缘的有缘人。因叶兄这本《游戏引擎架构》译著而在线上线下相识的读者们，你们是不是因“了解游戏引擎架构，从而制作/优化好游戏”这样的理想而结了缘呢？

亲爱的读者，愿你的游戏有一天因谜题巧妙绝伦、趣味超凡、虚拟世界气势磅礴、视觉效果逼真精美等专业因素取得业界褒奖，并得到玩家真诚的赞美。希望届时曾读叶兄这本《游戏引擎架构》译作的你，也可以回馈社会，回馈游戏开发的学习社区，帮助新人。希望你也可以建立微信公众号、博客等，或翻译游戏开发书籍，造福外语不好的读者，所以如果你的外语（英语、日语、韩语之于游戏行业比较重要）水平仍需精进，现在也可以同步加油了！

沙鹰 yingsha@qq.com

译序

数千年以来，艺术家们通过文学、绘画、雕塑、建筑、音乐、舞蹈、戏剧等传统艺术形式充实人类的精神层面。自20世纪中叶，计算机的普及派生出另一种艺术形式——电子游戏。游戏结合了上述传统艺术以及近代科技派生的其他艺术（如摄影、电影、动画），并且完全脱离了艺术欣赏这种单向传递的方式——游戏必然是互动的，“玩家”并不是“读者”、“观众”或“听众”，而是进入游戏世界、感知并对世界做出反应的参与者。

基于游戏的互动本质，游戏的制作通常比其他大众艺术复杂。商业游戏的制作通常需要各种人才的参与，而他们则需要依赖各种工具及科技。游戏引擎便是专门为游戏而设计的工具及科技集成。之所以称为引擎，如同交通工具中的引擎，提供了最核心的技术部分。因为复杂，研发成本高，人们不希望制作每款游戏（或车款）时都重新设计引擎，重用性是游戏引擎的一个重要设计目标。

然而，各游戏本身的性质以及平台的差异，使研发完全通用的游戏引擎变得极困难，甚至不可能。市面上出售的游戏引擎，有一些虽然已经达到很高的技术水平，但在商业应用中，很多时候还是需要因应个别游戏项目对引擎改造、整合、扩展及优化。因此，即使能使用市面上最好的商用引擎或自研引擎，我们仍需要理解当中的架构、各种机制和技术，并且分析及解决在制作中遇到的问题。这些也是译者曾任于上海两家工作室时的主要工作范畴。

选择翻译此著作，主要原因是在阅读中得到共鸣，并且能知悉一些知名游戏作品实际上所采用的方案。有感坊间大部分游戏开发书籍并不是由业内人士执笔，内容只足够应付一些最简单的游戏开发，欠缺宏观比较各种方案，技术与当今实际情况也有很大差距。而一些Gems类丛书虽然偶有好文章，但受形式所限欠缺系统性、全面性。难得本书原作者身为世界一流游戏工作室的资深游戏开发者[1]，在繁重的游戏开发工作外，还在大学教授游戏开发课程以至编写本著作。此外，从与内地同事的交流中，了解到许多从业者不愿意阅读外文书籍。为了普及知识及反馈业界社会，希望能尽绵力。

[1]原作者是顽皮狗（Naughty Dog）《神秘海域（Uncharted）》系列的通才程序员、《最后生还者（The Last of Us）》的首席程序员，之前还曾在EA和Midway工作。

或许有些人以为本著作是针对单机 / 游戏机游戏的，并不适合国内以网游为主的环境。但译者认为这是一种误解，许多游戏本身所涉及的技术是具通用性的。例如游戏性相关的游戏性系统、场景管理、人工智能、物理模拟等部分，许多时候也会同时用于网游的前台和后台。现时，一些动作为主、非MMO的国内端游甚至会直接在后台运行传统意义上的游戏引擎。至于前台相关的技术，单机和端游的区别更少。此外，随着近年移动终端的兴起，其硬件性能已超越传统掌上游戏机，开发手游所需的技术与传统掌上游戏机并无太大差异。还可预料，现时单机 / 游戏机的一些较高级的架构及技术，将在不远的未来着陆移动终端平台。

译者认为，本书涵括游戏开发技术的方方面面，同时适合入门及经验丰富的游戏程序员。书名中的架构二字，并不单是给出一个系统结构图，而是描述每个子系统的需求、相关技术及与其他子系统的关系。对译者本人而言，本书的第11章（动画系统）及第14章（运行时游戏性基础系统）是本书特别精彩之处，含有许多少见于其他书籍的内容。而第10章（渲染引擎）由于是游戏引擎中的一个极大的部分，有限的篇幅可能未能覆盖广度及深度，推荐读者参考[1][2]，人工智能方面也需参考其他专著。

本译作采用LaTeX排版[3]，以Inkscape[4]编译矢量图片。为了令阅读更流畅，内文中的网址都统一改以脚注标示。另外，由于现时游戏开发相关的文献以英文为主，而且游戏开发涉及的知识面很广，本译作尽量以括号形式保留英文术语。为了方便读者查找内容，在附录中增设中英文双向索引（索引条目与原著的不同）。

本人在香港成长学习及工作，至2008年才赴内地游戏工作室工作，不黯内地的中文写作及用字习惯，翻译中曾遇到不少困难。有幸得到出版社人员以及良师益友的帮助，才能完成本译作。特别感谢周筠老师支持本作的提案，并耐心地给予协助及鼓励。编辑张春雨老师和卢鹆翔老师，以及好友余晟给予了大量翻译上的知识及指导。也感谢游戏业界专家云风、大宝和Dave给予了许多宝贵意见。此书的翻译及排版工作比预期更花时间，感谢妻子及儿女们的体谅。此次翻译工作历时三年半，因工作及家庭事宜导致严重延误，唯有在翻译及排版工作上更尽心尽力，希望求得等待此译作的读者们谅解。无论是批评或建议，诚希阁下通过电邮[5]、新浪微博[6]、豆瓣[7]等渠道不吝赐教。

叶劲峰（Milo Yip）

2013年10月

---

[2]中括号表示引用附录中的参考文献。一些参考条目加入了其中译本的信息。

[3]具体是使用CTEX套装（http://www.ctex.org/），它是在MiKTeX（http://www.miktex.org/）的基础上增加中文的支持。

[4]http://inkscape.org/

[5]miloyip@gmail.com

[6]http://weibo.com/miloyip/

[7]http://www.douban.com/people/miloyip/

奉献给

Trina、Evan及Quinn Gregory

纪念我们的英雄

Joyce Osterhus及Kenneth Gregory

序言

最早的电子游戏完全由硬件构成，但微处理器（microprocessor）的高速发展完全改变了游戏的面貌。现在的游戏是在多用途的PC和专门的电子游戏主机（video game console）上玩的，凭借软件带来绝妙的游戏体验。从最初的游戏诞生至今已有半个世纪，但很多人仍然认为游戏是一个未成熟的产业。即使游戏可能是个年轻的产业，若仔细观察，也会发现它正在高速发展。现时游戏已成为一个上百亿美元的产业，覆盖不同年龄、性别的广泛受众。

千变万化的游戏，可以分为从纸牌游戏到大型多人在线游戏（massively multiplayer online game，MMOG）等多个种类（category）和“类型（genre）”[8]，也可以运行在任何装有微芯片（microchip）的设备上。你现在可以在PC、手机及多种特别为游戏而设计的手持/电视游戏主机上玩游戏。家用电视游戏通常代表最尖端的游戏科技，又由于它们是周期性地推出新版本，因此有游戏机“世代”（generation）的说法。最新一代[9]的游戏机包括微软的Xbox 360和索尼的PlayStation 3，但一定不可忽视长盛不衰的PC，以及最近非常流行的任天堂Wii。

最近，剧增的下载式休闲游戏，使这个多样化的商业游戏世界变得更复杂。虽然如此，大型游戏仍然是一门大生意。今天的游戏平台非常复杂，有难以置信的运算能力，这使软件的复杂度得以进一步提升。所有这些先进的软件都需要由人创造出来，这导致团队人数增加，开发成本上涨。随着产业变得成熟，开发团队要寻求更好、更高效的方式去制作产品，可复用软件（reusable software）和中间件（middleware）便应运而生，以补偿软件复杂度的提升。

由于有这么多风格迥异的游戏及多种游戏平台，因此不可能存在单一理想的软件方案。然而，业界已经发展出一些模式，也有大量的潜在方案可供选择。现今的问题是如何找到一个合适的方案去迎合某个项目的需要。再进一步，开发团队必须考虑项目的方方面面，以及

---

[8]译注：Genre一词在文学中为体裁。电影和游戏里通常译作类型。不同的游戏类型可见1.2节。

[9]译注：按一般说法，2005年至今属于第7个游戏机世代。这3款游戏机的发行年份为Xbox 360（2005）、PlayStation 3（2006）、Wii（2006）。有关游戏机世代可参考http://en.wikipedia.org/wiki/List_of_video_game_consoles。

如何把各方面集成。对于一个崭新的游戏设计，鲜有可能找到一个完美搭配游戏设计各方面的软件包。

现时业界内的老手，入行时都是“开荒牛”。我们这代人很少是计算机科学专业出身（Matt的专业是航空工程、Jason的专业是系统设计工程），但现时很多学院已设有游戏开发的课程和学位。时至今日，为了获取有用的游戏开发信息，学生和开发者必须找到好的途径。对于高端的图形技术，从研究到实践都有大量高质量的信息。可是，这些信息经常不能直接应用到游戏的生产环境，或者没有一个生产级质量的实现。对于图形以外的游戏开发技术，市面上有一些所谓的入门书籍，没提及参考文献就描述很多内容细节，像自己发明的一样。这种做法根本没有用处，甚至经常带有不准确的内容。另一方面，市场上有一些高端的专门领域书籍，例如物理、碰撞、人工智能等。可是，这类书或者啰嗦到让你难以忍受，或者高深到让部分读者无法理解，又或者内容过于零散而难于融会贯通。有一些甚至会直接和某项技术挂钩，软硬件一旦改动，其内容就会迅速过时。

此外，互联网也是收集相关知识的绝佳工具。可是，除非你确实知道要找些什么，否则断链、不准确的资料、质量差的内容也会成为学习障碍。

好在，我们有Jason Gregory，他是一位拥有在顽皮狗（Naughty Dog）工作经验的业界老手，而顽皮狗是全球高度瞩目的游戏工作室之一。Jason在南加州大学教授游戏编程课程时，找不到概括游戏架构的教科书。值得庆幸的是，他承担了这个任务，填补了这个空白。

Jason把应用到实际发行游戏的生产级别知识，以及整个游戏开发的大局编集于本书。他凭经验，不仅融汇了游戏开发的概念和技巧，还用实际的代码示例及实现例子去说明怎样贯通知识来制作游戏。本书的引用及参考文献可以让读者更深入探索游戏开发过程的各方面。虽然例子经常是基于某些技术的，但是概念和技巧是用来实际创作游戏的，它们可以超越个别引擎或API的束缚。

本书是一本我们入行做游戏时想要的书。我们认为本书能让入门者增长知识，也能为有经验者开拓更大的视野。

Jeff Lander[10]

Matthew Whiting[11]

---

[10]译注：Jeff Lander现时为Darwin 3D公司的首席技术总监、Game Tech公司创始人，曾为艺电首席程序员、Luxoflux公司游戏性及动画技术程序员。

[11]译注：Matthew Whiting现时为Wholesale Algorithms公司程序员，曾为Luxoflux公司首席软件工程师、Insomniac Games公司程序员。

前言

欢迎来到《游戏引擎架构》世界。本书旨在全面探讨典型商业游戏引擎的主要组件。游戏编程是一个庞大的主题，有许多内容需要讨论。不过相信你会发现，我们讨论的深度将足以使你充分理解本书所涵盖的工程理论及常用实践的方方面面。话虽如此，令人着迷的漫长游戏编程之旅其实才刚刚启程。与此相关的每项技术都包含丰富内容，本书将为你打下基础，并引领你进入更广阔的学习空间。

本书焦点在于游戏引擎的技术及架构。我们会探讨商业游戏引擎中，各个子系统的相关理论，以及实现这些理论所需要的典型数据结构、算法和软件接口。游戏引擎与游戏的界限颇为模糊。我们将把注意力集中在引擎本身，包括多个低阶基础系统（low-level foundation system）、渲染引擎（rendering engine）、碰撞系统（collision system）、物理模拟（physics simulation）、人物动画（character animation），及一个我称为**游戏性基础层**（gameplay foundation layer）的深入讨论。此层包括游戏对象模型（game object model）、世界编辑器（world editor）、事件系统（event system）及脚本系统（scripting system）。我们也将会接触游戏性编程（gameplay programming）的多个方面，包括玩家机制（player mechanics）、摄像机（camera）及人工智能（artificial intelligence，AI）。然而，这类讨论会被限制在游戏性系统和引擎接口范围。

本书可以作为大学中等级游戏程序设计中两到三门课程的教材。当然，本书也适合软件工程师、业余爱好者、自学的游戏程序员，以及游戏行业从业人员。通过阅读本书，资历较浅的游戏程序员可以巩固他们所学的游戏数学、引擎架构及游戏科技方面的知识。专注某一领域的资深程序员也能从本书更为全面的介绍中获益。

为了更好地学习本书内容，你需要掌握基本的面向对象编程概念并至少拥有一些C++编程经验。尽管游戏行业已经开始尝试使用一些新的、令人兴奋的编程语言，然而工业级的3D游戏引擎仍然是用C或C++编写的，任何认真的游戏程序员都应该掌握C++。我们将在第3章重温一些面向对象编程的基本原则，毫无疑问，你还会从本书学到一些C++的小技巧，不过C++的基础最好还是通过阅读[39]、[31]及[32]来获得。如果你

对C++已经有点生疏，建议你在阅读本书的同时，最好能重温这几本或者类似书籍。如果你完全没有C++经验，在看本书之前，可以考虑先阅读[39]的前几章，或者尝试学习一些C++的在线教程。

学习编程技能最好的方法就是写代码。在阅读本书时，强烈建议你选择一些特别感兴趣的主题付诸实践。举例来说，如果你觉得人物动画很有趣，那么可以首先安装OGRE，并测试一下它的蒙皮动画示范。接着还可以尝试用OGRE实现本书谈及的一些动画混合技巧。下一步你可能会打算用游戏手柄控制人物在平面上行走。等你能玩转一些简单的东西了，就应该以此为基础，继续前进！之后可以转移到另一个游戏技术范畴，周而复始。这些项目是什么并不重要，重要的是你在**实践**游戏编程的艺术，而不是纸上谈兵。

游戏科技是一个活生生、会呼吸的家伙，永远不可能将之束缚于书本之上。因此，附加的资源、勘误、更新、示例代码、项目构思等已经发到本书的网站[12]。

[12] http://gameenginebook.com

致谢

书籍从来不会凭空出现，本书亦然。没有家人、朋友、行业同僚的帮助，本书就不可能出版。衷心感谢他们及所有曾协助我完成本书的人们。

当然，受写作项目影响最严重的莫过于我的家庭。所以我想首先向我的妻子Trina致以特别的感谢。在这个艰难时期，她成为家庭的力量支柱。当我闭关完成一章又一章时，她昼夜无间照顾我们的两个儿子Evan（5岁）和Quinn（3岁）。我要多谢她，因为她放弃了自己的计划去适应我的时间表，她完成自己及我份内的家务（通常多于我想承认的），她经常在最需要的时刻给我鼓励。我也希望向儿子们表示感谢，长子Evan耐心容忍我不陪他玩他最爱的电视游戏，幼子Quinn在我每天长时间工作回家后给我拥抱和无尽的笑容。

我还要向我的编辑Matt Whiting及Jeff Lander致以特别的感谢。他们适时的反馈深具洞察力和针对性，常常恰到好处。此外，他们丰富的行业经验坚定了我将本书写得尽可能精准及前沿的信心。能跟这么老练的专家合作，既是愉快的经历，也是我的荣幸。感谢Jeff替我和Alice Peters[13]穿针引线，使这一项目起初能得以开展。

许多顽皮狗同事也对本书做出了贡献。他们要么提供反馈意见，要么帮助我制定某些章节的结构和主题内容。多谢Marshall Robin和Carlos Gonzalez-Ochoa对渲染一章的指导及监督，同时感谢Pål-Kristian Engstad对该章的文字及内容提供的建议。还要多谢Christian Gyrling对本书几个章节的反馈，包括动画一章（动画是他的专长之一）。感谢顽皮狗工程团队同仁制作了这么优秀的游戏引擎系统，成为本书的亮点。特别多谢艺电的Keith Schaeffer提供12.1节中关于物理对游戏影响的原始材料。也要多谢艺电的Paul Keet和圣达戈Midway的Steve Ranck（他是“Hydro Thunder”项目的主工程师）多年来的教导。虽然他们没有直接参与本书，但他们以不同形式影响了几乎本书每一页的内容。

这本书的素材源自我在南加州大学的信息技术课程（Information Technology Programme）中所开发的课堂“ITP-485：游戏引擎编程（Programming Game Engines）”的笔记，这门课到现在已经讲授了差不多3年。感谢Anthony Borquez博士，正是当时身为资

[13]译注：Alice Peters是本书原出版社A K Peters创始人之一。

讯技术计划总监的他，聘请我开发ITP-485这门课程。我也要多谢现在的课程总监Ashish Soni继续支持及鼓励我去发展ITP-485。

还要感谢亲友们的不断鼓励，包括他们常常在我写作时照顾我的妻子和两个儿子。我要感谢我的小姨子Tracy Lee、小舅子Doug Provins、姻表兄Matt Glenn，以及我们的好朋友Kim Clark和Drew Clark、Sherilyn Kritzer和Jim Kritzer、Anne Scherer和Michael Scherer、Kim Warner和Mike Warner。在我少年时，父亲Kenneth Gregory写了一本关于股票投资的书，让我萌生写作的念头。对此及其他事情，我永远都要感激他。也要感谢母亲Erica Gregory，一部分是因为她坚持希望我着手这一项目，一部分是因为在我小时候，她花费了很多心血教导我写作的艺术，我的写作技巧（以至于工作态度……和我古怪的幽默感）完全得自于她!

最后，我谨多谢Alice Peters、Kevin Jackson-Mead及A K Peters所有员工为出版本书做出的巨大努力。我很高兴可以与Alice和Kevin工作，衷心感谢他们在这么紧迫的时间下拼命工作，并多谢他们给予我这新作者的无限耐心。

Jason Gregory

2009年4月

目录

第一部分

基础

第1章　导论

在1979年，笔者获得了人生中第一台游戏主机——美泰（Mattel）公司超酷的Intellivision[1]，当时“游戏引擎（game engine）”一词还没出现。那时候，多数成年人会认为电子游戏和街机游戏（arcade game）只是玩具而已，而游戏的软硬件都是为某游戏特制的。到2008年，游戏已经成为上百亿美元[2]的主流产业，无论是市场规模还是普及程度，游戏产业都不逊于好莱坞。这些现时无处不在的三维游戏世界，都由**游戏引擎**驱动。游戏引擎，例如id Software公司[3]的Quake及Doom、Epic Games公司的虚幻（Unreal）、Valve公司的Source等，都已变成具完整功能的可复用软件开发套件。厂商可以取得这些游戏引擎的授权，用这些引擎来制作几乎任何能想象到的游戏。

虽然各个游戏引擎的结构和实现细节千差万别，但无论是可公开授权使用的引擎，还是私有的内部引擎，都显现出一些粗粒度模式。几乎所有的游戏引擎都含有一组常见的核心组件，例如渲染引擎、碰撞及物理引擎、动画系统、音频系统、游戏世界对象模型、人工智能系统等。而这些组件内也开始显现一些半标准的设计方案。

坊间有许多讲述各游戏引擎子系统的书籍，比方说，在三维图形方面就有描述非常详尽的著作。另有一些，将多个游戏技术领域的窍门技巧，集结成书。但笔者尚未找到一本著作，能让读者全盘了解组成现代游戏引擎的各组件。本书的目标，就是引导读者走进这庞大、复杂的游戏引擎架构世界。

从本书中，读者能学习到以下内容。

- 如何架构工业级生产用游戏引擎。
- 现实中的游戏开发团队怎样组织及运作。

---

[1]译注：Intellivision是世界上第一款16位电子游戏机，比任天堂的8位红白机（Famicom）还早4年。

[2]译注：原文为multi-billion，即数十亿。但根据美国娱乐软件协会的资料，2008年美国的计算机及电视游戏软件的销售达117亿美元，因此做出修正。

[3]译注：游戏公司id Software名字中的id并非identity（身份）或identifier（标识符）的缩写，而是佛洛伊德提出的精神分析学说中，精神三大部分——本我（id）、自我（ego）与超我（superego）——之一。id的发音像kid除去了k，而非读I、D。

- 有哪些主要子系统及设计模式不断出现在几乎所有游戏引擎里。
- 每个主要子系统的典型需求。
- 有哪些子系统与游戏类型或具体游戏无关，有哪些子系统是为某游戏类型或具体游戏而设计的。
- 引擎和游戏的边界位于何处。

在书中我们会先学习一些流行游戏引擎的内部运作，例如雷神之锤及虚幻。也会讨论一些知名的中间件（middleware）包，例如Havok物理库、OGRE渲染引擎及Rad Game Tools公司的Granny三维动画几何管理工具箱。

在正式开始之前，我们会从游戏引擎的背景出发，回顾大规模软件工程中的一些技巧及工具，包括：

- 逻辑软件架构和物理软件架构的区别。
- 配置管理（configuration management）、版本控制（revision control）及生成系统（build system）。
- 最常用的C/C++开发环境——微软Visual Studio——的窍门及技巧。

本书假设读者对C++（多数游戏开发者所选择的编程语言）有充分理解，并明白一些基本的软件工程原理。同时，也假设读者懂得一些线性代数、三维矢量、矩阵、三角学（尽管我们会在第4章重温一些核心概念）。读者最好能了解一些实时及事件驱动编程的基本概念。但无须顾虑，本书会扼要重温这些内容，也会提供适当的参考资料供读者学习。

1.1 典型游戏团队的结构

在开始钻研游戏引擎之前，先简单介绍一下典型游戏团队的人员配置。游戏工作室（game studio）通常由5个基本专业领域的人员构成，包括工程师（engineer）、艺术家（artist）、游戏设计师（game designer）、制作人（producer）及其他管理/支持人员（市场策划、法律、信息科技/技术支持、行政等）。每个专业领域可细分为多个分支，以下逐一介绍。

1.1.1 工程师

工程师设计并实现软件，使游戏及工具得以运行。有时候，工程师分为两类：**运行时**程序员（runtime programmer）和**工具**程序员（tool programmer）。运行时程序员制作引

擎和游戏本身；工具程序员制作离线工具，供整个团队使用，以提高团队的工作效率。运行时/工具两方面的工程师都各有专长。有些工程师在职业生涯里专注单一的引擎系统，诸如渲染、人工智能、音效或碰撞/物理；有些工程师专注于游戏性（gameplay）[4]和脚本编程（scripting）；也有一些工程师喜欢系统层面的开发，而不太关心游戏实际上怎么玩[5]；还有些工程师是通才（generalist），博学多才，能应付开发中不同的问题。

资深工程师有时候会被赋予技术领导的角色。比如，首席工程师（lead engineer）通常仍会设计及编写代码，但同时协助管理团队的时间表，并决定项目的整体技术方向。从人力资源的角度来说，首席工程师有时候也会直接管理下属。

有些公司设有一位或多位技术总监（technical director，TD），负责从较高层面监督一个或多个项目，确保团队能注意到潜在的技术难点、业界走势、新技术等。某些工作室可能还有一个和工程相关的最高职位，这就是首席技术官（chief technical officer，CTO）。CTO类似整个工作室的技术总监，并履行公司的重要行政职务。

1.1.2 艺术家

游戏界有云：“内容为王（content is king）。”艺术家肩负制作游戏中所有视听内容的重任，而这些内容的品质能够决定游戏成败。下面来了解一下艺术家的不同分工。

- **概念艺术家**（concept artist）通过素描或绘画，让团队了解游戏的预设最终面貌。概念艺术家的工作始于游戏开发的概念阶段，一般会在项目的整个生命周期里继续担任美术指导。游戏成品的屏幕截图常会不可思议地贴近概念艺术图（concept art）。
- **三维建模师**（3D modeler）为游戏世界的所有事物制作三维几何模型。这类人员通常会再细分为两类：前景建模师（foreground modeler）及背景建模师（background modeler）[6]。前景建模师负责制作物体、角色[7]、载具、武器及其他游戏中的对象，而背景建模师则制作静态的背景几何模型（如地形、建筑物、桥梁等）。
- **纹理艺术家**（texture artist）制作称为纹理（texture）的二维影像。这些纹理用来贴附于三维模型之上，以增加模型的细节及真实感。
- **灯光师**（lighting artist）布置游戏世界的静态和动态光源，并通过颜色、亮度、光源方向等设定，加强每个场景的美感及情感。

---

[4]译注：Gameplay又译作游戏玩法、可玩性。

[5]译注：有些公司以游戏程序员（gameplay programmer，GPP）和引擎程序员（engine programmer）区分。本人曾获引擎工程师（engine engineer）的职衔，英文比较拗口。

[6]译注：又称为关卡建模师（level modeler）、环境建模师（environment modeler）或关卡美术设计师（level artist）。

[7]译注：由于人物建模和其他物体或场景的建模在技术及工作方式上有很大分别（例如前者需和动画师紧密合作），所以很多公司有独立的角色建模师（character modeler）或称为角色艺术家（character artist）。

- **动画师**（animator）为游戏中的角色及物体加入动作。如同动画电影制作，在游戏制作过程中，动画师充当演员。但是，游戏动画师必须具有一些独特的技巧[8]，以制作符合游戏引擎技术的动画。
- **动画捕捉演员**（motion capture actor）提供一些原始的动作数据。这些数据经由动画师整理后，置于游戏中。
- **音效设计师**（sound designer）与工程师紧密合作，制作并混合游戏中的音效及音乐。
- **配音演员**（voice actor）为游戏角色配音。
- **作曲家**（composer）为游戏创作音乐。

如同工程师，资深艺术家有时候会成为团队的领导。一些游戏有一位或多位**艺术总监**（art director），他们是资深的艺术家，负责把控整个游戏的艺术风格，并维持所有团队成员作品的一致性。

1.1.3 游戏设计师

游戏设计师（game designer）负责设计玩家体验的互动部分，这部分一般称为**游戏性**。不同种类的游戏设计师，从事不同细致程度的工作。有些（一般为资深的）游戏设计师在宏观层面上设定故事主线、整体的章节或关卡顺序、玩家的高层次目标。其他游戏设计师[9]则在虚拟游戏世界的个别关卡或地域上工作，例如设定哪些地点会出现敌人、放置武器及药物等补给品、设计谜题元素等。其他游戏设计师会在非常技术性的层面上和游戏性工程师（gameplay engineer）紧密合作。部分游戏设计师是工程师出身，他们希望能更主动地决定游戏的玩法。

有些游戏团队会聘请一位或多位**作家**（writer）。游戏作家们的工作范畴很宽，例如，与资深游戏设计师合作编制故事主线，以至于编写每句对话。

如同其他游戏专业领域，有些资深游戏设计师也会负责管理团队。很多游戏团队设有游戏总监（game director）一职，负责监督游戏设计的各个方面，帮助管理时间表，并保证每位游戏设计师的设计在整个游戏中具有一致性。资深的游戏设计师有时候会转行为制作人。

1.1.4 制作人

在不同的工作室里，制作人（producer）的角色不尽相同。有些游戏公司，制作人负责

[8]译注：游戏和动画电影的主要不同之处在于，游戏的角色能回应玩家的输入。这种互动性需要各个动画片段能互相结合，所以其制作过程也和动画电影有所不同。

[9]译注：一般称作关卡设计师（level designer）。

管理时间表，并同时承担人力资源经理的职责。有些游戏公司里，制作人主要做资深游戏设计师的工作。还有些游戏工作室，要求制作人作为开发团队和商业部门（财政、法律、市场策划等）之间的联系人。有些工作室甚至是完全没有制作人。例如在顽皮狗（Naughty Dog）公司，几乎所有员工，包括两位副总裁，都直接参与游戏制作，工作室的资深成员分担了团队管理工作及公司事务。

1.1.5 其他工作人员

游戏开发团队通常需要一支非常重要的支持团队，包括工作室的行政管理团队、市场策划团队（或一个与市场研究公司联系的团队）、行政人员及IT部门。IT部门负责为整个团队采购、安装及配置软硬件，并提供技术支持。

1.1.6 发行商及工作室

游戏的市场策划、制造及分销，通常由**发行商**（publisher）负责，而非开发游戏的工作室本身。发行商通常是大企业，例如艺电（Electronic Arts，EA）、THQ、维旺迪（Vivendi）、索尼（Sony）、任天堂（Nintendo）等。很多游戏工作室并不隶属于个别发行商，这些工作室把他们制作的游戏，卖给出最好条件的发行商。还有一些工作室让单一发行商独家代理他们的游戏，其形式可以是签署长期发行合同，或是成为发行商全资拥有的子公司。例如，THQ的游戏工作室都是独立运作的，但THQ拥有这些工作室，并对它们有最终的控制权。艺电更进一步，直接管理其下属工作室。另外，**第一方开发商**（first-party developer）是指游戏工作室直接隶属于游戏主机生产商（索尼、任天堂、微软）。例如，顽皮狗是索尼的第一方开发商。这些工作室独家为母公司的游戏硬件制作游戏。

1.2 游戏是什么

“游戏”是什么，每个人多半都有自己非常直观的理解。“游戏”一词泛指棋类游戏（board game），如象棋和《大富翁（Monopoly）》；纸牌游戏（card game），如梭哈（poker）和二十一点（blackjack）；赌场游戏（casino game），如轮盘（roulette）和老虎机（slot machine）；军事战争游戏（military war game）、计算机游戏、孩子们一起玩耍的多种游戏等。学术上还有个“博弈论（game theory）”，它是指在一个明确的游戏规则框架下，多个代理人（agent）选择战略及战术，以求自身利益的最大化。在游戏主机及计算机娱乐的语境中，“游戏”一词通常会使我们的脑海里浮现一个三维虚拟世界，玩家可以控制人

物、动物或载具。（老一辈的玩家可能会想起一些二维的经典游戏，如《乓（Pong）》、《吃豆人（Pac-Man）》、《大金刚（Donkey Kong）》等。）在《快乐之道：游戏设计的黄金法则（Theory of Fun for Game Design）》一书中，拉夫·科斯特（Raph Koster）把游戏定义为一个互动体验，为玩家提供一连串渐进式挑战，玩家最终能通过学习而精通该游戏[26]。科斯特的命题把学习及精通作为游戏的乐趣（fun）。这正如听一个笑话时，发现其中的奥妙，明白笑点的一瞬间该笑话变得有趣一样。

基于本书主旨，我们会集中讨论游戏的一个子集，子集里的游戏由二维或三维虚拟世界组成，并有少量的玩家（1～16个左右）。本书大部分的内容也可以应用到互联网上的Flash游戏、纯解谜游戏（如《俄罗斯方块（Tetris）》）或大型多人在线游戏（massively multiplayer online games，MMOG）。但我们主要集中讨论一些游戏引擎，这些游戏引擎可以用来开发第一人称射击、第三人称动作/平台游戏、赛车游戏、格斗游戏等。

1.2.1 电子游戏作为软实时模拟

大部分二维或三维的电子游戏，会被计算机科学家称为**软实时**（soft real-time）**互动**（interactive）**基于代理**（agent-based）**计算机模拟**（computer simulation）的例子。以下，我们把这个词组分拆讨论，以便理解。

在大部分电子游戏中，会用数学方式来为一些真实世界（或**想象**世界）的子集**建模**（model），从而使这些模型能在计算机中运行。明显地，我们不可能模拟世界上的所有细节，例如到达原子或夸克（quark）的程度，所以这些模型只是现实或想象世界的简化或近似版本。也因此，数学模型是现实或虚拟世界的**模拟**。近似化（approximation）和简化（simplification）是游戏开发者最有力的两个工具。若能巧妙地运用它们，就算是一个被大量简化的模型，也能非常接近现实，难辨真假，而能带来的乐趣也比现实更多。

基于代理模拟是指，模拟中多个独立的实体（称为代理）一起互动。此术语非常符合三维电子游戏的描述，游戏中的载具、人物角色、火球、豆子等都可视为代理。由于大部分游戏都有基于代理的本质，所以多数游戏采用面向对象（object-oriented）编程语言，或较宽松的基于对象（object-based）编程语言，也不足为奇了。

所有互动电子游戏都是**时间性模拟**（temporal simulation），即游戏世界是**动态的**（dynamic）——随着游戏事件和故事的展开，游戏世界状态随着时间改变。游戏也必须回应人类玩家的输入，这些输入是游戏本身不可预知的，因而也说明游戏是**互动时间性模拟**（interactive temporal simulation）。最后，多数游戏会描绘游戏的故事，并实时回应玩家输入，这使游戏成为**互动实时模拟**（interactive real-time simulation）。显著的反例是一些回合制游戏，如计算机化象棋及非实时策略游戏，尽管如此，这些游戏通常也会向用户提供某

种形式的实时图形用户界面（graphical user interface，GUI）。因此基于本书的目标，将假设所有电子游戏至少都会有**一些**实时限制。

时限（deadline）是所有实时模拟的核心概念。在电子游戏中，明显的例子是需要屏幕每秒最少更新24次，以制造运动的错觉。（大部分游戏会以每秒30或60帧的频率渲染画面，因为这是NTSC制式显示器刷新率[10]的倍数。）当然，电子游戏也有其他类型的期限。例如物理模拟可能需要每秒更新120次以保持稳定。一个游戏角色的人工智能系统可能每秒最少要“想一次”才能显得不呆。另外，也可能需要每1/60秒调用一次声音程序库，以确保音频缓冲有足够的声音数据，避免发出一些短暂失灵声音。

“软”实时系统是指一些系统，即使错过期限却不会造成灾难性后果。因此，所有游戏都是**软实时系统**（soft real-time system）——如果帧数不足，人类玩家在现实中不会因此而死亡。与此相比，**硬实时系统**（hard real-time system）错过期限可能会导致操作者损伤甚至死亡。直升机的航空电子系统和核能发电厂的控制棒（control rod）[11]系统便是硬实时系统的例子。

模拟虚拟世界许多时候要用到数学模型。数学模型可分为**解析式**（analytic）或**数值式**（numerical）。例如，一个刚体因地心引力而以恒定加速度落下，其分析式（闭合式/closed form）数学模型可写为：

$$y(t) = \frac{1}{2}gt^2 + v_0t + y_0 \tag{1.1}$$

分析式模型可为其自变量（independent variable）设任何值来求值。例如上面的方程中，给予初始条件y_0和v_0、常量g，就能设任何时间t来求$y(t)$的值。可是，大部分数学问题并没有闭合式解。在电子游戏中，用户输入是不能预知的，因此不应期望可以对整个游戏完全用分析式建模。

刚体受地心引力落下的数值式模型可写为：

$$y(t + \Delta t) = F(y(t), \dot{y}(t), \ddot{y}(t), \dots) \tag{1.2}$$

即是说，该刚体在$(t + \Delta t)$未来时间的高度，可以用目前的高度、高度的第一导数、高度的第二导数及目前时间t为参数的函数来表示。为实现数值式模拟，通常要不断重复计算，以决定每个离散时步（time step）的系统状态。游戏也是如此运作的。一个主“游戏循环（game loop）”不断执行，在循环的每次迭代中，多个游戏系统，例如人工智能、游戏逻辑、物理模拟等，就会有机会计算或更新其下一离散时步的状态。这些结果最后可渲染成图形显示、发出声效，或者输出至其他设备，例如游戏手柄的力反馈（force feedback）。

[10]译注：NTSC是美洲国家和日韩等国家的常用电视广播制式，其刷新率大约是59.94Hz。

[11]译注：控制棒由能吸收中子的材料制成，是用来控制核分裂速率的设备。

1.3 游戏引擎是什么

“游戏引擎（game engine）”这个术语在20世纪90年代中期形成，与第一人称射击游戏（first person shooter，FPS）如id Software公司非常受欢迎的游戏《毁灭战士（Doom）》有关。《毁灭战士》的软件架构相当清楚地划分成核心软件组件（如三维图形渲染系统、碰撞检测系统和音频系统等）、美术资产（art asset）、游戏世界、构成玩家游戏体验的游戏规则（rule of play）。这么划分是很有价值的，若另一个开发商取得这类游戏的授权，只要制作新的美术、关卡布局、武器、角色、载具、游戏规则等，对引擎软件做出很少的修改，就可以把游戏打造成新产品。这一划分也引发mod社区的兴起。mod是指，一些特殊游戏玩家组成的小组，或小规模的独立游戏工作室，利用原开发商提供的工具箱修改现有的游戏，从而创作出新的游戏。在20世纪90年代末，一些游戏，如《雷神之锤III竞技场（Quake III Arena）》和《虚幻（Unreal）》，在设计时就照顾到复用性和mod。使用脚本语言，譬如id公司的QuakeC，可以非常方便地定制引擎。而且，游戏工作室对外授权引擎，已成为第二个可行的收入来源。今天，游戏开发者可以取得一个游戏引擎的授权，复用其中大部分关键软件组件去制作游戏。虽然这个做法还要开发一些定制软件工程，但已经比工作室独力开发所有的核心软件组件经济得多。

通常，游戏和其引擎之间的分界线是很模糊的。一些引擎有相当清晰的划分，一些则没有尝试把二者分开。在一款游戏中，渲染代码可能特别“知悉”如何画一只妖兽（orc）；在另一款游戏中，渲染引擎可能只提供多用途的材质及着色功能，“妖兽”可能完全是用数据去定义的。没有工作室可以完美地划分游戏和引擎。这不难理解，因为随着游戏设计的逐渐成形，这两个组件的定义会经常转移。

数据驱动架构（data-driven architecture）或许可以用来分辨一个软件的哪些部分是引擎，哪些部分是游戏。若一个游戏包含硬编码逻辑或游戏规则，或使用特例代码去渲染特定种类的游戏对象，则复用该软件去制作新游戏就会变得困难甚至不可行。因此，这里说的“游戏引擎”是指可扩展的软件，而且不需要大量修改就能成为多款游戏软件的基础。

很明显这不是一个非黑即白的区别方法。我们可以根据每个引擎的可复用性，把引擎放置于一个连续谱之上。图1.1在这个连续谱上对几款知名的游戏/引擎进行定位。

有些人可能以为游戏引擎能变成一个通用软件（像Apple QuickTime或微软的媒体播放器），去运行几乎**任何**可以想象到的游戏内容。可是，这个设想至今尚未（或许永远不能）实现。大部分游戏引擎是针对特定游戏及特定硬件平台所精心制作及微调的。就算是一些最通用的游戏引擎，其实也只适合制作某类型游戏，例如第一人称射击或赛车游戏。我们完全可以说，游戏引擎或中间件组件越通用，在特定平台运行特定游戏的性能就越一般。

出现这种现象，是因为设计高效的软件总是需要取舍，而这些取舍是基于一些假设的，像是一个软件会如何使用及在哪个硬件上运行等。例如，一个渲染引擎为紧凑的室内环境而设计，一般就不能很好地渲染广大的室外场景。室内引擎可能使用BSP树或入口系统，不会渲染被墙或物体遮挡的几何图形。室外引擎则可能使用较不精确的（甚至不使用）遮挡剔除，但它大概会更充分地利用层次细节技巧，去保证较远的景物用较少的三角形来渲染，而距摄像机较近的几何物体则用高清晰的三角形网格。

随着计算机硬件速度的提高及专用显卡的应用，再加上更高效的渲染算法及数据结构，不同游戏类型的图形引擎差异已经缩小。例如，现在可以用第一人称射击引擎去做实时策略游戏。但是，通用性和最优性仍然需要取舍。按照游戏/硬件平台的特定需求及限制，经常可以通过微调引擎制作更精美的游戏。

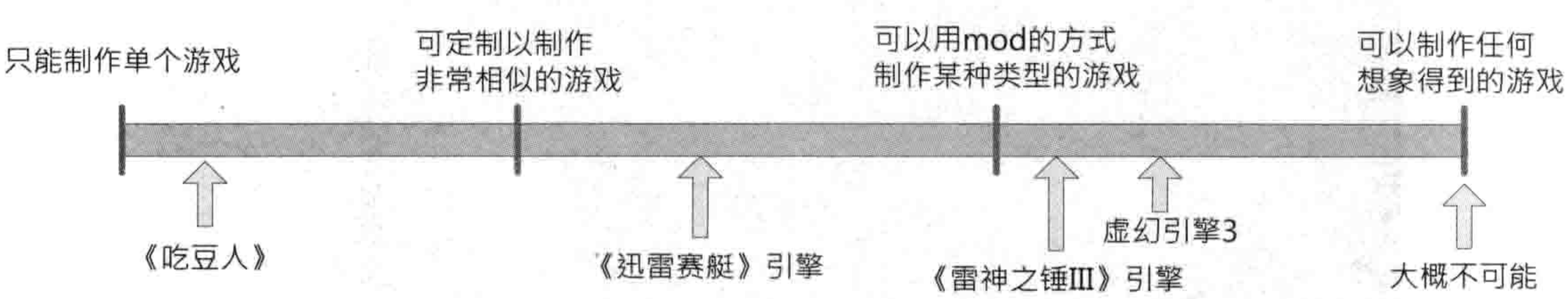


图 1.1: 游戏引擎复用性连续谱。

1.4 不同游戏类型中的引擎差异

通常在某种程度上游戏引擎是为某游戏类型（genre）而设的。为两人在拳击台上格斗而设计的游戏引擎，有别于大型多人在线游戏（MMOG）、第一人称射击（FPS）或实时策略游戏（RTS）引擎。可是，各种引擎也有很大的重叠部分，例如，无论是什么类型的三维游戏，都需要某形式的低阶用户输入（如从游戏手柄/键盘/鼠标）、某形式的三维网格渲染、某形式的平视显示器（heads-up display，HUD）[12]（包括渲染不同字体的文本）、强大的音频系统等。例如，虽然虚幻引擎是为第一人称射击而设计的，但它同样能制作其他类型的游戏，例如英佩游戏（Epic Games）工作室的畅销第三人称射击游戏《战争机器（Gears of War）》，上海麻辣马（Spicy Horse）工作室[13]的角色动作冒险游戏《格林童话惊魂记（American McGee's Grimm）》，以及韩国Acro Games公司的未来派赛车游戏《Speed Star》。

以下介绍几个常见的游戏类型，并探讨每个类型的技术需求。

[12]译注：平视显示器这个术语来自现代航空器，原意指一种不用低头看仪表就能把数据显示于机师面前的仪器。游戏中，HUD是指画面中游戏世界上浮动的用户界面，例如一直显示在画面上的玩家血条。

[13]译注：译者首次读到这里备感惊奇和荣幸，因为这是一家中国大陆的游戏工作室，而且译者初读本书时正在这家公司工作。

1.4.1 第一人称射击

第一人称射击（first person shooting，FPS）的典型例子是《雷神之锤（Quake）》、《虚幻竞技场（Unreal Tournament）》、《半条命（Half-Life）》、《反恐精英（Counter-Strike）》和《使命召唤（Call of Duty）》（图1.2）。历史上，这些游戏中的角色，以相对较慢的走路方式移动，并漫游于可能很大但主要为走廊的场景。可是，现代的FPS可以在不同的场景中进行，例如开阔的室外范围及狭小的室内范围。现代FPS的角色移动机制可包括行走、轨道载具、地面载具、气垫船（hovercraft）[14]、船只及飞机等。FPS的概略可参考维基百科[15]。

图 1.2: 《使命召唤2（Call of Duty 2）》（Xbox 360/PlayStation 3）。

FPS是开发技术难度极高的游戏类型之一。能与此相比的或许只有第三人称射击/动作/平台游戏，以及大型多人在线游戏。这是因为FPS要让玩家面对一个精细而超现实的世界时感到身历其境。也难怪游戏业界的巨大技术创新都来自这种游戏。

FPS游戏常会注重技术，例如：

- 高效地渲染大型三维虚拟世界。

[14]译注：hovercraft中文译名可能会引起误会，事实上气垫“船”也可以在陆地及冰上行走。

[15]http://en.wikipedia.org/wiki/First-person_shooter

- 快速反应的摄像机控制及瞄准机制。
- 玩家的虚拟手臂和武器的逼真动画。
- 各式各样的手持武器。
- 宽容的玩家角色运动及碰撞模型，通常使游戏有种“漂浮”的感觉。
- 非玩家角色（如玩家的敌人及同盟）有逼真的动画及智能。
- 小规模在线多人游戏的能力（通常支持多至同时64位玩家在线），及无处不在的死亡竞赛（death match）游戏模式。

FPS中使用的渲染技术几乎总是经过高度优化，并且按特定场景类型仔细调整过的。例如，室内“地下城爬行（dungeon crawl）[16]”游戏通常会利用二元空间分割树（binary space partitioning，BSP tree）或基于入口（portal）的渲染系统。室外FPS游戏使用其他种类的渲染优化，例如遮挡剔除（occlusion culling），或游戏在运行前预先把游戏世界分区化（sectorization），以自动或手动方式去设定每个分区是否能见到另一个分区。

当然，要让玩家在超现实游戏世界中有如亲临其境，除了经优化的高质量图形技术，还需要具备更多条件。在FPS中，角色动画、音效音乐、刚体物理、游戏内置电影及大量其他技术都必须是最前沿的。因此，这个游戏类型的技术需求是业界里最严格、也最全面的。

1.4.2 平台及其他第三人称游戏

“平台游戏（platformer）”是指基于人物角色的第三人称游戏（third person game），在这类游戏中，主要的游戏机制是在平台之间跳跃。经典的例子中，在二维时代有《太空惊魂记（Space Panic）》、《大金刚（Donkey Kong）》、《森林寻宝历险记（Pitfall!）》及《超级玛里奥（Super Mario Brothers ）》，三维时代有《超级玛里奥64（Super Mario 64）》、《古惑狼（Crash Bandicoot）》、《雷曼2（Rayman 2）》、《音速小子（Sonic the Hedgehog）》、《杰克与达斯特（Jak and Daxter）》系列（图1.3）、《瑞奇与叮当（Ratchet & Clank）》系列，以及较近期的《超级玛里奥银河（Super Mario Galaxy）》。对这个游戏类型的详细探讨，可参考维基百科[17]。

从技术上说，平台游戏通常可以和第三人称射击/动作/历险游戏类型一并考虑，例子有《Ghost Recon》、《战争机器（Gears of War）》（图1.4）、《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》。

---

[16]译注：地下城爬行原指幻想游戏中英雄在地下谜宫里对付怪兽和寻宝等的情景，是《龙与地下城》等游戏的常用术语。

[17]http://en.wikipedia.org/wiki/Platformer

图 1.3: 《杰克与达斯特（Jak and Daxter）》。

图 1.4: 《战争机器（Gears of War）》。

第三人称游戏和第一人称射击游戏有许多共通之处，但第三人称游戏比较看重主角的能力（ability）及运动模式（locomotion mode）[18]。除此以外，这个类型游戏的主角化身（avatar）需要高度逼真的全身动画，相比起来，典型的FPS里主角的“漂浮手臂”的动画要求是比较简单的。要注意，因为大部分FPS游戏都会有多人在线模式，所以除了第一人称的手臂外往往还需要渲染主角的全身动画。不过，在FPS游戏中，玩家化身的逼真程度一般并不及非玩家角色（NPC），更不能和第三人称游戏的玩家化身相比。

在平台游戏中，游戏主角通常是比较卡通而不是很真实或细腻的。但是，第三人称射击通常使用非常真实的人形玩家角色。这两种类型都需要非常丰富的行为和动画。

第三人称游戏特别注重的技术如下。

- 移动平台、梯子、绳子、棚架及其他有趣的运动模式。
- 用来解谜的环境元素。
- 第三人称的“跟踪摄像机”会一直注视玩家角色，也通常会让玩家用手柄右摇杆（在游戏主机上）或鼠标（在PC上）旋转摄像机（虽然在PC上有很多流行的第三人称射击游戏，但平台游戏类型几乎是游戏主机上独有的）。
- 复杂的摄像机碰撞系统，以保证视点不会穿过背景几何物体或动态的前景物体。

1.4.3 格斗游戏

格斗游戏（fighting game）通常是两个玩家控制角色在一个擂台上互相对打。典型的例子有《灵魂能力（Soul Calibur）》和《铁拳（Tekken）》（图1.5）。维基百科网页[19]介绍了这个游戏类型。

传统格斗类型游戏注重以下技术。

- 丰富的格斗动画。
- 准确的攻击判定。
- 能侦测复杂按钮及摇杆组合的玩家输入系统。
- 人群，或相对静态的背景。

由于这些游戏的三维世界比较小，而且摄像机一直位于动作的中心，以往这些游戏只有很少甚至不需要世界细分（world subdivision）或遮挡剔除。同样地，这些游戏不要求使用高阶的三维音频传播模型。

[18]译注：运动模式在这里是指动物的运动，例如行走、跳跃、游泳、飞行等。

[19]http://en.wikipedia.org/wiki/Fighting_game

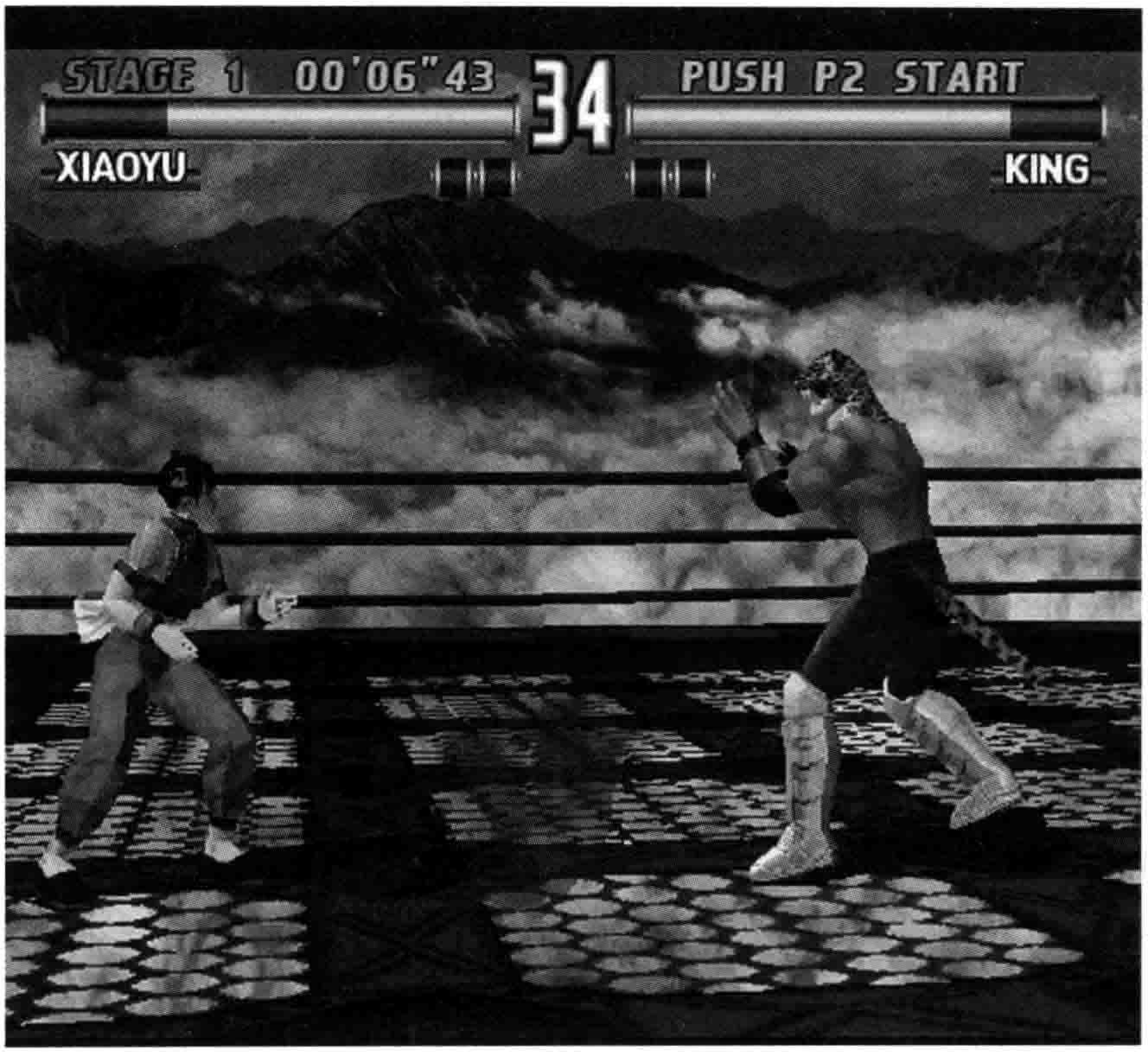


图 1.5: 《铁拳3（Tekken 3）》（PlayStation）。

最尖端的格斗游戏，如艺电的《拳击之夜3（Fight Night Round 3）》（图1.6），把技术提升到另一个层次，该作品有以下一些特点。

- 高清的角色图形，包括仿真的皮肤着色器（shader）。着色器模拟了次表面散射（subsurface scattering，SSS）及冒汗效果。
- 逼真的角色动画。
- 基于物理的布料及头发模拟。

值得留意的是，一些格斗游戏，如《天剑（Heavenly Sword）》，是在一个大型的环境下而不是受限的竞技场进行的。事实上，很多人认为这是另一个游戏类别，有时称其为**brawler**[20]。这种格斗游戏的技术需求比较近似于第三人称游戏及实时策略游戏。

[20]译注：有时候也称为beat 'em up（痛殴他们）。这类游戏中，玩家以单人或多人合作方式，在关卡中不断击倒敌人。以刀剑武器为主的同类游戏又称为hash and slash（切和斩）。

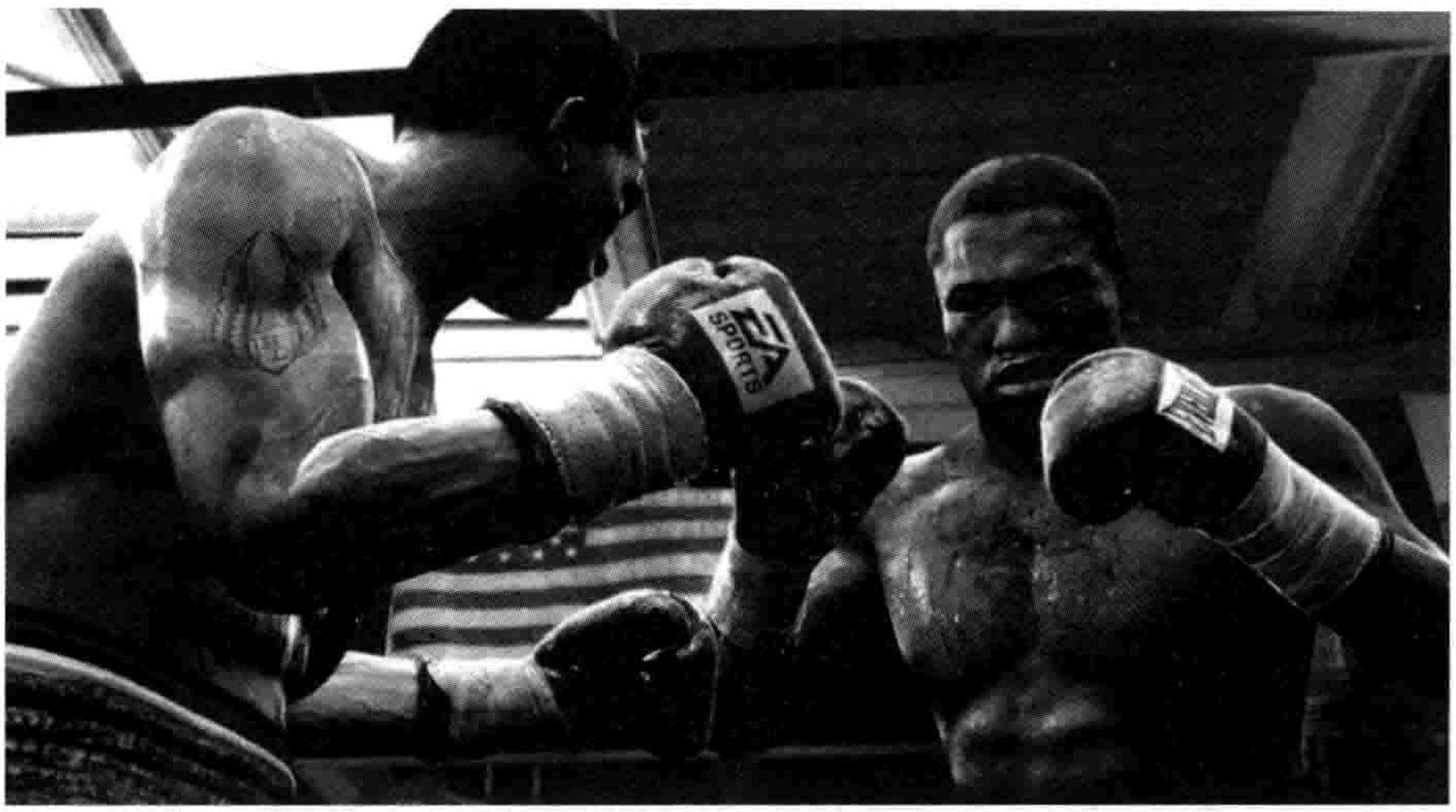


图 1.6: 《拳击之夜3（Fight Night Round 3）》（PlayStation 3）。

1.4.4 竞速游戏

竞速游戏（racing game）[21]包括所有以赛道上驾驶车辆或其他载具为主要任务的游戏。这个游戏类型有几个子类别。着重模拟的竞速游戏（“sims”）力求模仿真实的驾驶体验（如《跑车浪漫旅（Gran Turismo）》）。街机（arcade）竞速游戏偏好娱乐性多于真实感（如《洛杉矶赛车（San Francisco Rush）》、《极速狂飙（Cruis'n USA）》[22]、《迅雷赛艇（Hydro Thunder）》）。一个较近期的子类型是来自街头竞速（street racing）的亚文化，这类型游戏里采用可改装的汽车。卡丁赛车（kart racing）也是一个子类型，有时候会使用一些平台游戏或电视卡通角色为主角，驾驶怪诞的汽车（如《玛里奥赛车（Mario Kart）》、《杰克X（Jak X ）》、《捍卫战士（Freaky Flyers）》）。“竞速”游戏也不一定是和时间有关的比赛，例如，一些卡丁车游戏让玩家去射击对手、收集物品，或参与其他计时或不计时的任务。有关竞速游戏的讨论，可见维基百科[23]。

竞速游戏通常是非常线性的，这比较像旧式的FPS游戏。但移动速度一般比FPS游戏快许多。因此，这类游戏经常使用非常长的走廊式赛道和环形赛道，有时候加入一些可选分支或捷径。竞速游戏把图形的细节集中在载具（vehicle）、赛道及近景。但是，卡丁车游戏还需要投放足够的渲染及动画资源到驾驶角色上。图1.7是知名竞速游戏系列《跑车浪漫旅5（Gran Turismo 5）》最新版本的屏幕截图。

[21]译注：日式缩写为RAC，但此缩写在西方不流行。

[22]译注：原文Cruisin' USA并不正确。

[23]http://en.wikipedia.org/wiki/Racing_game

典型竞速游戏有以下技术特性。

- 使用多种“窍门”去渲染遥远的背景，例如使用二维纸板形式的树木、山岳和山脉。
- 赛道通常切开成较简单的二维区域，称为“分区（sector）”。这些数据结构用来实现渲染优化、可见性判断（visibility determination），帮助非玩家操控车辆的人工智能及路径搜寻，以及解决很多其他技术问题。
- 第三人称视角摄像机通常追随在车辆背后，第一人称摄像机有时候会置于架驶舱里。
- 如果赛道经过天桥底及其他狭窄空间，必须花精力防止摄像机和背景几何物体碰撞。

图 1.7: 《跑车浪漫旅5（Gran Turismo 5）》（PlayStation 3）。

1.4.5 实时策略游戏

现在的实时策略（real-time strategy，RTS）游戏类型可以认为是由《沙丘魔堡2（Dune II: The Building of a Dynasty）》（1992）奠定的。同类游戏包括《魔兽争霸（Warcraft）》、《命令与征服（Command & Conquer）》、《帝国时代（Age of Empires）》及《星际争霸（Starcraft）》。在这类游戏中，玩家在一个广阔的场地里，利用兵工厂策略地部署作战单位（battle units）去试图压倒对手。游戏世界通常会以斜面俯视（oblique top-down view）[24]的视角显示。关于这个游戏类型的讨论可参考维基百科[25]。

RTS通常不容许玩家改变视角以观看不同距离的景物。这个限制使开发者能在RTS渲染引擎上采用各种优化。

[24]译注：很多RTS游戏使用等角投影（isometric projection），即屏幕上3个轴的夹角均为（或接近）120°，如图1.8所示。

[25]http://en.wikipedia.org/wiki/Real-time_strategy

较老的同类游戏基于栅格（grid-based，或称为基于单元/cell-based）去构建游戏世界，并使用正射投影（orthographic projection）[26]，这两个技巧大大简化了渲染系统。例如，图1.8显示了经典RTS《帝国时代》的屏幕截图。

现在的RTS游戏也会使用透视投影及真三维世界，但这些游戏可能仍使用栅格排列系统，以保证作战单位和背景元素（如建筑物）能适当地对齐。例如，如图1.9所示的《命令与征服3》。

RTS游戏的惯用手法如下。

- 每个作战单元使用相对较低解析度的模型，使游戏能支持同时显示大量单元。
- 游戏的设计和进行多是在高度场地形（height field terrain）画面上展开的。
- 除了部署兵力，游戏通常准许玩家在地形上兴建新的建筑物。
- 用户互动方式通常为单击及以范围选取单元，再加上包含指令、装备、作战单元种类、建筑种类等的菜单及工具栏。

图 1.8: 《帝国时代（Age of Empires）》（PC）。

[26]译注：正射投影，即3个轴投影到屏幕时仍然是平行的。和透视投影（perspective projection）相反，正射投影不会有远小近大的效果。

图 1.9: 《命令与征服3（Command & Conquer 3）》（PC）。

1.4.6 大型多人在线游戏

大型多人在线游戏（massively multiplayer online game，MMOG）的典型例子有《无冬之夜（Neverwinter Nights）》[27]、《无尽的任务（EverQuest）》、《魔兽世界（World of Warcraft）》及《星球大战: 星系（Star Wars Galaxies）》。MMOG定义为能同时支持大量玩家（由数千至数十万），一般来说，这些玩家会在非常大的持久世界（persistent world）里进行游戏（持久世界是指其状态能持续一段很长的时间，比特定玩家每次玩的时间长很多）。除了同时在线人数和持久性外，MMOG的游戏体验和小型的多人游戏是相似的。MMOG的子类型有MMO角色扮演游戏（MMORPG）、MMO实时策略游戏（MMORTS）及MMO第一人称射击游戏（MMOFPS）。关于这些游戏类型，可参考维基百科[28]。图1.10是极度流行的MMORPG《魔兽世界》的截图。

MMOG的核心是一组非常强大的服务器。这些服务器维护游戏世界的权威状态，管理用户登入/登出，也会提供用户间文字对话或IP电话（voice over internet protocol，VoIP）

[27]译注：这里是指于1991—1997年在AOL营运的Neverwinter Nights MMORPG。国内玩家更熟悉的版本，是由Bioware公司开发的单机版，于2002发行。

[28]http://en.wikipedia.org/wiki/MMOG

等服务[29]。几乎所有MMOG都要求用户定期支付服务费用，也可能在游戏内或游戏外支持小额交易（micro-transaction）。这些都是开发商的主要收入来源，因此可能中央服务器最重要的角色是处理账单及小额交易。

因为MMOG的游戏场景规模和玩家数量都很大，MMOG里的图形逼真程度通常稍低于其他游戏。

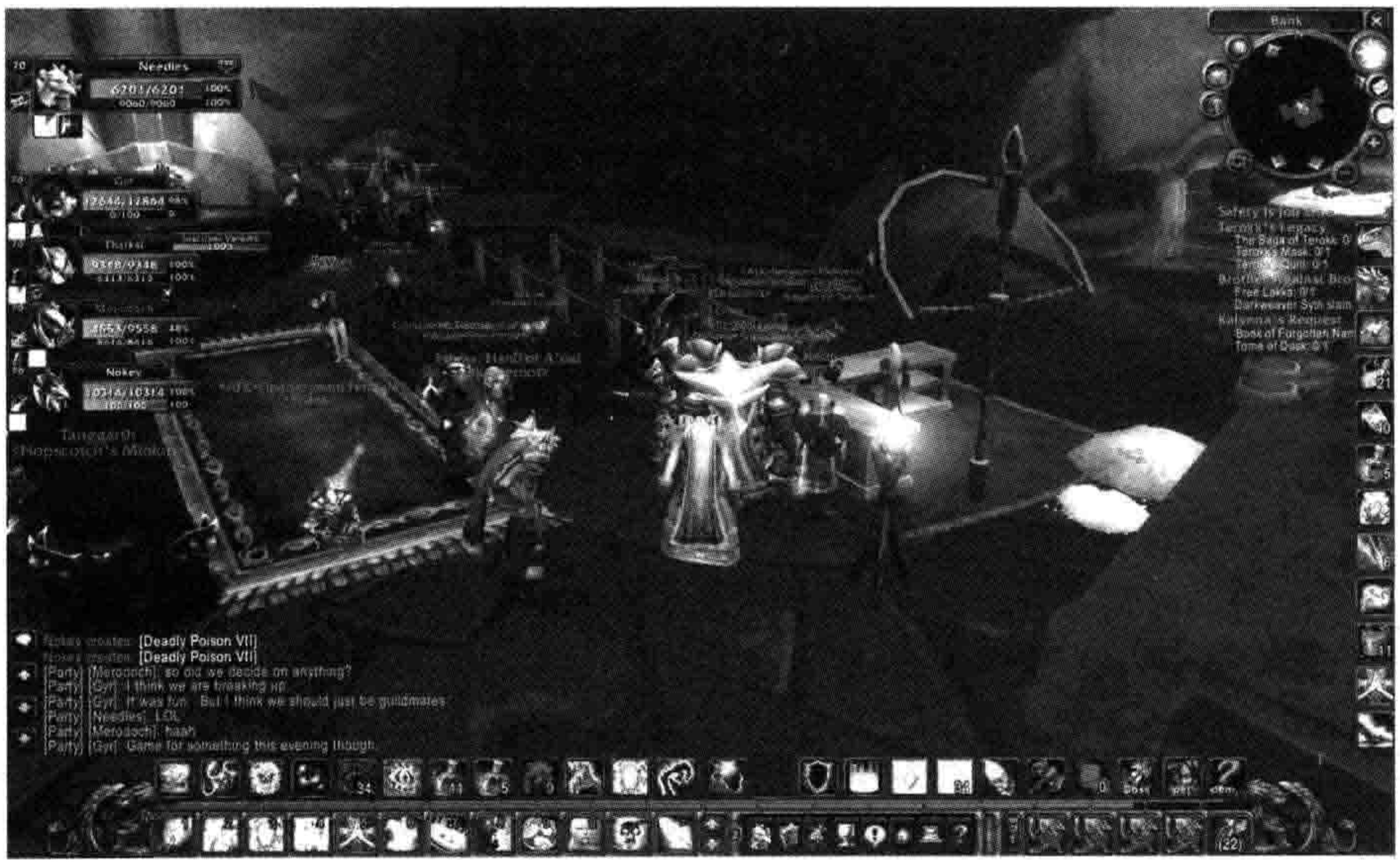


图 1.10: 《魔兽世界（World of Warcraft）》（PC）。

1.4.7 其他游戏类型

还有很多游戏类型，不在此详述，例如：

- 体育游戏（sports）[30]，各主要体育项目是其子类型（如橄榄球、篮球、足球、高尔夫球等）。
- 角色扮演游戏（role playing game，RPG）。
- 上帝模拟游戏（god game），如《上帝也疯狂（Populous）》[31]和《黑与白（Black & White）》。

---

[29]译注：此处谈及的服务器功能是MMOG相对其他小型线上游戏的特点。一些小型线上游戏不需要专用服务器（dedicated server），而是以一个玩家的客户端同时兼任服务器，或使用点对点（peer-to-peer）模式。但MMOG服务器的一个重要功能，是让所有玩家同步互动。

[30]译注：日式缩写为SPT，但此缩写在西方不流行。

[31]译注：原文Populus并不正确。

- 环境或社会模拟游戏（environmental/social simulation），如《模拟城市（SimCity）》和《模拟人生（The Sims）》。
- 解谜游戏（puzzle）如《俄罗斯方块（Tetris）》。
- 非电子游戏的移植，如象棋、围棋、卡牌游戏等。
- 基于网页的游戏，例如艺电公司Pogo网站提供的游戏。
- 其他游戏类型。

各游戏类型有其特殊的技术需求，因此传统上游戏引擎因游戏类型而有些差异。然而，不同游戏类型的技术需求也有很大的共通之处，尤其在单个硬件平台上，共通之处特别多。由于硬件性能的不断提升，因考虑优化而产生的游戏类型差异将会缩小。因此，现在把一个引擎技术应用于不同游戏类型，甚至不同硬件平台，变得越来越可行。

1.5 游戏引擎概观

1.5.1 雷神之锤引擎家族

一般认为，首个三维第一人称射击游戏（FPS）是《德军总部（Castle Wolfenstein 3D）》（1992年）。这款PC游戏由美国得克萨斯州的id Software公司制作，它引领游戏工业进入令人兴奋的新方向。id Software公司相继开发了《毁灭战士（Doom）》、《雷神之锤（Quake）》、《雷神之锤2（Quake II）》及《雷神之锤3（Quake III）》。这些引擎在架构上非常相似，所以本书统称为雷神之锤引擎家族。Quake的技术曾用来制作很多游戏，甚至用来制作其他引擎。例如，《荣誉勋章（Medal of Honor）》PC版本的引擎血统大约是：

- 《雷神之锤3》（id Software公司）。
- 《原罪（SiN）》[32]（Ritual公司）。
- 《重金属F.A.K.K.^{2}》（Ritual公司）。
- 《荣誉勋章：联合行动（Medal of Honor: Allied Assault）》（2015 & Dreamworks Interactive）。
- 《荣誉勋章：血战太平洋（Medal of Honor: Pacific Assault）》（洛杉矶艺电）。

其他许多基于雷神之锤技术的游戏，其引擎血统同样复杂，也历经了多个游戏及工作室。事实上，Valve公司的Source引擎（用来开发《半条命》）也能追溯到雷神之锤技术。

《雷神之锤》和《雷神之锤2》的源代码可免费获得，而原始雷神之锤引擎的架构相当

[32]译注：原文Sin的大小写不正确。

“优秀”并且“整洁”（虽然代码是有点过时，并且纯粹用C语言编写）。这些代码库都是非常好的例子，能说明工业级游戏引擎是怎样制作的。完整的《雷神之锤》和《雷神之锤2》源代码可在id Software的网站下载[33]。

若拥有《雷神之锤》或《雷神之锤2》游戏，就可以在Visual Studio中编译那些代码，并利用游戏盘上的真实游戏资产，在调试器里执行该游戏。这样做一遍是非常有启发性的。也可以先设置中断点执行游戏，之后单步执行代码，分析引擎如何运作。笔者强烈建议下载一两个这类引擎，并用上面所说的方式去分析源代码。

1.5.2 虚幻引擎

1998年，Epic Games公司通过传奇的游戏《虚幻（Unreal）》闯入FPS世界。自此，在FPS界里，虚幻成为雷神之锤的主要竞争对手。虚幻引擎2代（UE2）是《虚幻竞技场2004（Unreal Tournament 2004）》（UT2004）的基础，此引擎也曾用来制作无数的“mods”，其中包括大学项目及商业游戏。虚幻引擎3代（UE3）是其下一个进化阶段，号称拥有业界最好的工具和最丰富的引擎功能。例如，它有方便且强大的图形用户界面去制作着色器（shader），又有一个名为Kismet[34]的图形用户界面供编写游戏逻辑之用。近年很多游戏皆用UE3制作，包括Epic Games的当红之作《战争机器（Gears of War）》。

虚幻引擎以其全面的功能及内聚易用的工具见称。但虚幻引擎并非完美，大部分开发者会以不同方式优化它在具体硬件平台上的运行状况[35]。虚幻引擎是极为强大的原型制作（prototyping）工具和商业游戏平台，可用来制作几乎任何第一或第三人称的3D游戏（也可用来制作其他类型的游戏）。

虚幻开发者网络（Unreal Developer Network，UDN）提供不同版本虚幻引擎的丰富文档及其他信息[36]。UE2的部分文档可免费获得，使任何拥有UT2004游戏的玩家都可以制作mods。但是，UE2的其他文档及UE3的全部文档只供引擎授权者使用。可惜的是，UE3的授权费用极高，并非独立游戏开发者及大部分小工作室能承担的[37]。互联网上有许多关于虚幻的网站及维基，一个流行的网站是Beyond Unreal[38]。

---

[33] http://www.idsoftware.com/business/techdownloads

[34] 译注：Kismet在土耳其语和乌尔都语里，有命运或天命的意思。

[35] 译注：除了优化外，很多开发者也会修改或扩展引擎的功能以符合个别游戏的特殊需求。这也是译者目前的主要工作。

[36] http://udn.epicgames.com

[37] 译注：在2009年10月，Epic Games公司发布名为Unreal Development Kit（UDK）。这款软件可供业余爱好者免费制作基于虚幻引擎的非商业用游戏（不用购买游戏来做mod）。同时也提供较便宜的商用授权方式。UDK包含所有UE3的功能及工具，和UE3授权不同之处在于，UDK不提供源代码，开发者只能写UnrealScript代码（不容许用C++扩充），并暂时只提供Windows版本。详见http://www.udk.com/。

[38] http://www.beyondunreal.com

1.5.3 Source引擎

Valve公司使用自主开发的Source引擎制作了红极一时的《半条命2（Half Life 2）》、其续作《半条命2：第一/二章（HL2: Episode One/Two）》、《军团要塞2（Team Fortress 2）》、《传送门（Portal）》（这5个作品都包含在《橙盒（The Orange Box）》游戏套装里）。Source引擎的质量相当不错，其图形能力和工具套件可与虚幻引擎媲美[39]。

1.5.4 微软XNA Game Studio

微软XNA Game Studio是一个既易用又方便的游戏开发平台。该平台鼓励玩家去自创游戏，作品可于在线游戏社区分享，如同YouTube鼓励分享自制视频一样。

XNA基于微软的C#语言及公共语言运行库（Common Language Runtime，CLR）。XNA的主要开发环境为Visual Studio或其免费版本Visual Studio Express。Visual Studio能管理游戏项目里的一切资料，包括源代码和游戏美术资产（art asset）等。游戏开发者可以用XNA创作PC及微软Xbox 360的游戏。缴纳少许费用后，开发者就可以把游戏上传到Xbox Live网络，与朋友分享[40]。微软提供的这些工具，既优秀又免费，使一般人都能创作游戏。XNA显然有一个光辉迷人的未来。

1.5.5 其他商业引擎

此外坊间还有许多商业游戏引擎。尽管独立开发者的预算可能不足以购买引擎，但很多产品都有很好的在线文档或维基，这些都可作为游戏引擎及游戏编程的优质信息。例如，由Eric Lengyel于2001年创办的Terathon Software公司所开发的C4引擎[41]，其文档可于该公司网站上阅读，C4引擎的维基也提供了更多的详细资料[42]。

1.5.6 专有内部引擎

许多公司会开发并维护自己的游戏引擎。艺电的许多RTS游戏都基于由Westwood工作

[39]译注：译者认为，现时除了id Software公司的引擎和Source引擎，能与虚幻3竞争的可授权引擎是德国CryTek公司的CryEngine 3。CryEngine 3的室外场景管理、渲染能力和工具都非常强。CryTek公司把部分研究成果公开，可见http://www.crytek.com/cryengine/presentations。

[40]译注：微软自2008年年末，允许开发者用XNA制作Xbox Live Indie游戏，置于Xbox Live Marketplace售卖。可惜目前对开发者和顾客都有地域限制。详情可见XNA Creators Club Online网站http://creators.xna.com/。

[41]http://www.terathon.com

[42]http://www.terathon.com/wiki

室开发的SAGE引擎。顽皮狗（Naughty Dog）公司的《古惑狼（Crash Bandicoot）》、《杰克与达斯特（Jak and Daxter）》及最新的《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》系列，也都是在自己为PlayStation、PlayStation 2和PlayStation 3分别开发的专门引擎上开发而来。当然，大部分可商业授权的引擎，如雷神之锤、Source及虚幻引擎，一开始都是专有内部引擎。

1.5.7 开源引擎

开源三维游戏引擎是由业余及专业开发者制作，并在网上免费发布。“开源（open source）”通常意味着源代码可免费获得，并且其开发模式是全部公开的，即是说任何人都可以对代码做贡献。若有指明授权方式（licensing），通常都使用GNU通用公共许可证（GNU General Public License，GPL）或GNU宽通公共许可证（GNU Lesser General Public License，LGPL）[43]。GPL容许免费使用其代码，但其衍生作品也必须为GPL，即是作品的代码也要免费供他人使用；后者则容许在商业营利的产品中使用。此外还有其他免费或半免费的授权模式的开源项目。

互联网上有众多的开源引擎。有些质量相当不错，有些表现平平，有些糟糕透顶！游戏引擎的列表可在此网站[44]找到，登录后读者或许会感叹，原来现有游戏引擎如此之多。

OGRE是一个架构优良，又易学易用的三维渲染引擎。OGRE自夸拥有含高阶照明及阴影的全功能三维渲染系统、良好的骨骼角色动画系统、用作平视显示器（heads-up display，HUD）和图形用户界面（graphical user interface，GUI）的二维覆盖层（2D overlay）系统，以及用作全屏幕效果（如敷霜效果/bloom）的后期处理（post-processing）系统。OGRE的作者坦言OGRE并非一个完整的游戏引擎，但它提供差不多所有引擎都需要的许多基础组件。

以下列出其他一些知名的开源引擎。

- Panda3D是基于脚本的引擎。引擎的主要接口是用Python特制的脚本语言。其设计目标是方便快捷地制作三维游戏及虚拟环境。
- Yake是近期基于OGRE而开发的全功能引擎。
- Crystal Space是一个含扩充模组架构的游戏引擎。
- Torque[45]及Irrlicht都是知名且广泛使用的引擎。

---

[43]译注：原文Gnu Public License和Lesser Gnu Public License并不正确。

[44]译注：原文的超链接http://cg.cs.tu-berlin.de/~ki/engines.html已不能访问。读者可参考http://www.devmaster.net/engines/。

[45]译注：Torque并非一般意义上的开源项目。可能因其商业授权比较便宜并提供源代码，而产生这种错觉。

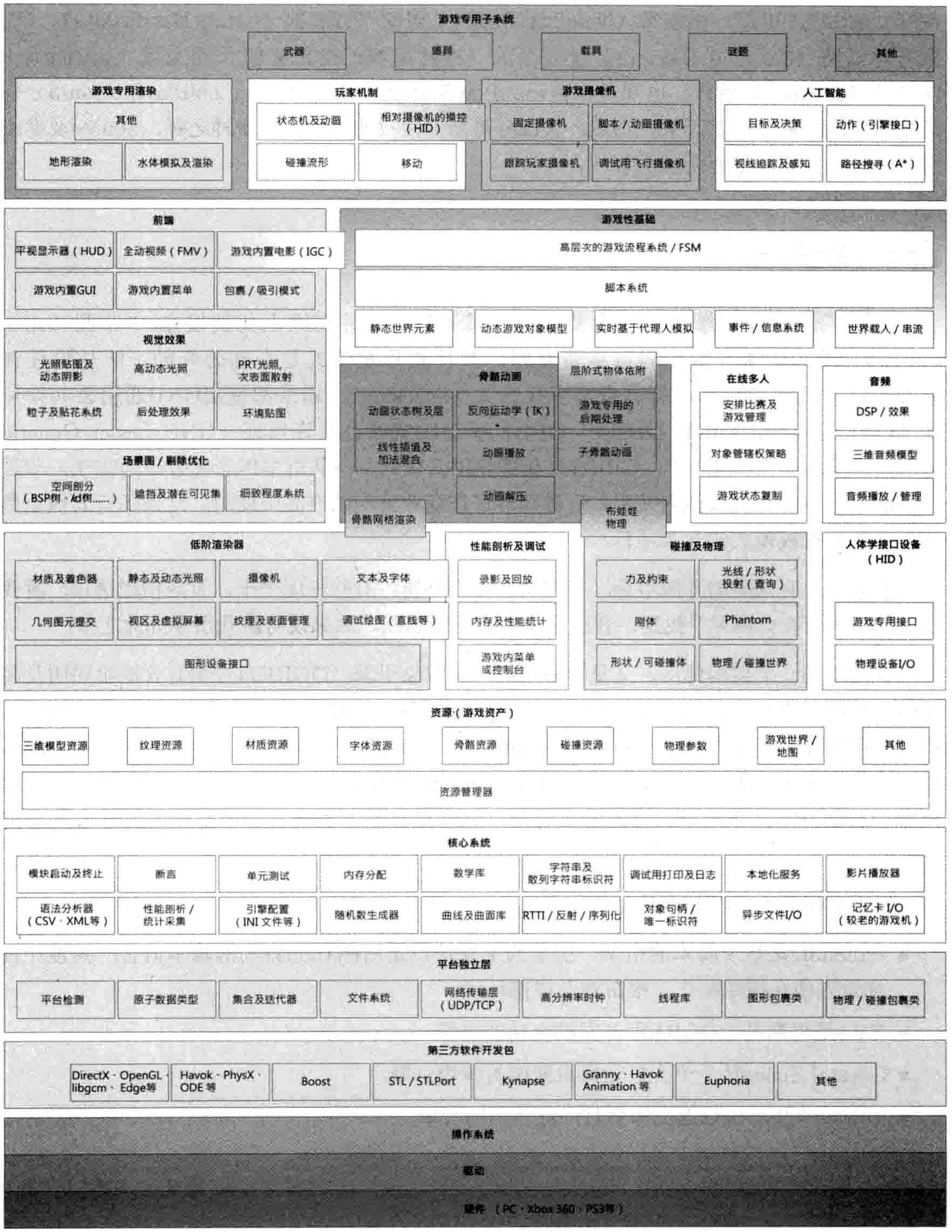


图 1.11: 运行时引擎架构。

1.6 运行时引擎架构

游戏引擎通常由工具套件和运行时组件两部分构成。本节先探讨运行时部分的架构，下节再阐述工具方面。

图1.11显示了一个典型三维游戏引擎的主要运行时组件。是的，此图很**庞大**！而且此图并未包含工具方面。由此可见，游戏引擎无疑是大型软件系统。

如同所有软件系统，游戏引擎也是以**软件层**（software layer）构建的。通常上层依赖下层，下层不依赖上层。当下层依赖上层时，称为循环依赖（circular dependency）。在任何软件系统中，**循环依赖**都要极力避免，不然会导致系统间复杂的耦合（coupling），也会使软件难以测试，并妨碍代码重用。对于大型软件系统，如游戏引擎，此问题尤其重要。

本节会逐一简介图1.11里的每个组件。本书余下部分会深入探讨这些组件，并学习如何整合这些组件至实际系统中。

1.6.1 目标硬件

图1.12显示了孤立的目标硬件层，它代表用来执行游戏的计算机系统或游戏主机。典型平台包括基于微软Windows或Linux的PC、苹果的iPhone及Machintosh、微软的Xbox/Xbox 360、索尼的PlayStation/PlayStation 2/PlayStation 3/PlayStation Portable（PSP）、任天堂的NDS/GameCube/Wii。本书大部分的内容是平台无关的，但也会提及PC和游戏主机之间的区别对设计的影响。

图 1.12: 硬件层。

1.6.2 设备驱动程序

如图1.13所示，设备驱动程序（device driver）是由操作系统或硬件厂商提供的最低阶软件组件。驱动程序负责管理硬件资源，也隔离了操作系统及上层引擎，使上层的软件无须理解不同硬件版本的通信细节差异。

图 1.13: 设备驱动层。

1.6.3 操作系统

在PC上，操作系统（operating system，OS）是一直运行的。操作系统协调一台计算机上多个程序的执行，其中一个程序可能是游戏。图1.14显示了操作系统层。操作系统如微软Windows，使用时间片（time-slice）方式，使多个执行中的程序能共享硬件，这称为抢占式多任务（preemptive multitasking）。这意味着PC游戏不能假设拥有硬件的所有控制权，PC游戏需要礼貌地配合其他系统中的程序。

图 1.14: 操作系统层。

在游戏主机上，操作系统通常只是个轻量级的库，链接到游戏的执行档里。在游戏主机上，游戏通常“拥有”整台机器。可是，自从Xbox 360和PlayStation 3的出现，这一说法变得不太准确。例如，这些新主机的操作系统会中断游戏的执行，接管某些系统资源以显示在线信息，或容许玩家暂停游戏以进入PS3的跨界导航菜单（Xross Media Bar，XMB）或Xbox 360的Dashboard。所以（不管是好是坏）游戏机和PC开发的分野正慢慢收窄。

1.6.4 第三方软件开发包和中间件

大部分游戏引擎都会借用许多第三方软件开发包（software development kit，SDK）及中间件（middleware），如图1.15所示。SDK提供基于函数或基于类的接口，一般称为应用程序接口（application programming interface，API）。以下会介绍几个例子。

图 1.15: 第三方软件开发包层。

1.6.4.1 数据结构及算法

如同任何软件系统，游戏也非常依赖数据结构（data structure）集合，以及操作这些数据的算法（algorithm）。例如，以下是一些提供这方面功能的第三方库。

- **STL**：C++标准模板库（standard template library，STL）提供很丰富的代码及算法去管理数据结构、字符串及基于流（stream）的输入、输出。
- **STLport**：这是一个可移植的、经优化的STL实现。

- **Boost**：Boost是非常强大的数据结构及算法库，采用STL的设计风格。（Boost的在线文档是一个学习计算机科学的好地方。）
- **Loki**：Loki[46]是强大的泛型编程（generic programming）模板库。它尤其擅长绞尽你的脑汁！

游戏开发者可分为两类：在他们的游戏引擎中使用STL模板库之类的，以及不使用的。一些开发者认为STL的内存分配模式（memory allocation pattern）不高效，也导致内存碎片问题（见5.2.1.4节），使STL不能在游戏中使用。一些开发者认为STL的强大和方便性超过它的问题，而且大部分问题实际上可以变通解决。笔者个人认为STL在PC上可以无碍使用，因为PC上有高级的虚拟内存（virtual memory）系统，谨慎地分配内存变得不那么紧要（虽然开发者仍要非常谨慎）。在游戏主机上，只有有限的（甚至没有）虚拟内存功能，而且缓存命中失败（cache miss）的代价极高，游戏开发者最好编写自定义的数据结构，保证是可预期和/或有限的内存分配模式。（在PC上做同样的事情肯定也错不了。）[47]

1.6.4.2 图形

大多数游戏渲染引擎都是建立在硬件接口库之上的，例如：

- **Glide**是三维图形SDK，专门为古老的Voodoo显卡而设。此SDK曾在硬件转换及照明（hardware transform and lighting，hardware T&L）的年代之前很流行。DirectX 7开始支持硬件T&L。[48]
- **OpenGL**是获广泛使用的跨平台三维图形SDK。
- **DirectX**是微软的三维图形SDK，也是OpenGL的主要竞争对手。
- **libgcm**是索尼提供给PlayStation 3 RSX图形硬件的低阶直接接口，在PlayStation 3上比OpenGL 更高效。
- **Edge**是由顽皮狗和索尼为PlayStation 3制作的强大高效渲染及动画引擎。

1.6.4.3 碰撞和物理

碰撞检测（collision detection）和刚体动力学（rigid body dynamics）（在游戏开发社区里简单称作“物理”）可由以下的知名SDK提供。

---

[46]译注：Loki是源于《C++设计新思维》的范例。另外，Loki（洛基）是北欧神话中的神祇。

[47]译注：有关STL标准在游戏应用的问题及解决方案，可参考艺电的EASTL论述，http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2271.html。

[48]译注：原文误写为DirectX 8。

- **Havok**是一个流行的工业级物理及碰撞引擎。
- **PhysX**是另一个流行的工业级物理及碰撞引擎，NVIDIA提供免费下载。[49]
- **Open Dynamics Engine（ODE）**是知名的开源物理及碰撞引擎包。[50]

1.6.4.4 角色动画

市面上有许多商用的角色动画包，例如：

- **Granny**：Rad Game Tools公司的流行Granny工具套件，包含健壮的三维模型导出器（exporter），支持主要的三维建模及动画软件如Maya、3ds Max等。Granny也包括负责读取及操作导出模型和动画数据的运行时库，以及强大的运行时动画系统。笔者认为，无论是商用或私有的API，Granny SDK拥有笔者见过设计得最好也最合逻辑的动画API，它在时间处理方面尤其优秀。
- **Havok Animation**：因为游戏角色变得越来越真实，物理和动画之间的分界线变得越来越模糊。制作知名Havok物理SDK的公司，决定制作一个附送的动画SDK，使融合物理和动画变得前所未有地容易。
- **Edge**：为PS3而设的Edge库是由顽皮狗的ICE团队、美国索尼计算机娱乐（SCE）的工具及技术组、欧洲的索尼高阶技术组联合制作。Edge包含强大及高效的动画引擎，以及为渲染而设的高效几何处理引擎。

1.6.4.5 人工智能

- **Kynapse**：直至不久前，每个游戏都是以自有方式处理人工智能（artificial intelligence，AI）。可是，Kynogon公司开发了一个名为Kynapse的中间件SDK[51]，提供低阶的AI构件，例如，路径搜寻（path finding）、静态和动态物体回避（avoidance）、空间内的脆弱点（vulnerabilities）辨认（例如，一扇开着的窗可能会有埋伏），以及相当好的AI和动画间接口。

---

[49]译注：NVIDIA在PhysX（曾被命名为NovodeX）加入了GPU加速以促进其显卡业务，只有PC版的PhysX是免费的。

[50]译注：现时另一个流行的开源物理引擎是**Bullet**（http://bulletphysics.org），它应用在多个商业游戏上，而且更有些平台专用优化。反而ODE已经多年没有更新了。

[51]译注：Kynogon公司已于2008年2月被Autodesk公司收购。另外，Kynapse此名字应是来自神经元的突触（synapse）。

1.6.4.6 生物力学角色模型

- **Endorphin和Euphoria：** 这两个动画套件，利用了真实人类运动的高阶生物力学模型（biomechanical model），去产生角色动作。[52]

如上面提及，物理和动画之间的分界线开始变得模糊。软件包，如Havok Animation，尝试用传统方式结合动画和物理，先由动画师利用Maya之类的工具制作基本动作，再于执行时利用物理去扩充那些动作。直至最近，Natural Motion公司制作了一个产品去尝试重新定义怎样在游戏或其他数字媒体中处理角色动作。

Natural Motion公司的第一个产品Endorphin，是一个Maya的插件，让动画师在角色上运行生物力学模拟，并产生如同手工制作的动画效果。生物力学模型同时考虑了角色的重心、体重分布，以及人类在地心引力及其他作用力影响下，会如何平衡及运动。

第二个产品Euphoria是实时版本的Endorphin，其目标是在执行时根据不能预知的外力，实时生成物理和生物力学上准确的角色动作。[53]

1.6.5 平台独立层

大多数游戏引擎需要运行于不同的平台上。像艺电、Activision Blizzard这样的公司，经常要游戏支持多个目标平台，从而覆盖最大的市场。通常，只有第一方工作室，例如索尼的顽皮狗和Insomniac工作室，可以无须为每个游戏同时支持两个或以上的目标平台。因此，大部分游戏引擎的架构都有一个平台独立层（platform independence layer），如图1.16所示。平台独立层在硬件、驱动程序、操作系统及其他第三方软件之上，以此把其余的引擎部分和大部分底层平台隔离。

图 1.16: 平台独立层。

平台独立层包装了常用的标准C语言库、操作系统调用及其他基础API，确保包装了的接口在所有硬件平台上均为一致。这是必须的，因为不同平台间有不少差异，即使所谓的“标准”库，如标准C语言库，也有平台差异。

[52]译注：这两个产品命名来自医学词汇。endorphin是内啡肽（一种脑内分泌的镇痛剂），euphoria是欣快症。

[53]译注：《侠盗猎车手4（Grand Theft Auto IV）》是其中一个利用了Euphoria的知名游戏。NPC会对其外在环境生成自然反应的动作。

1.6.6 核心系统

游戏引擎以及其他大规模复杂C++应用软件，都需要一些有用的实用软件（utility），本书把这类软件称为“核心系统（core system）”。图1.17显示了典型的核心系统层。以下是核心系统层的一些常见功能。

- **断言**（assertion）：断言是一种检查错误的代码。断言会插入代码中捕捉逻辑错误或找出与程序员原来假设不符的错误。在最后的生产版本中，一般会移除断言检查。
- **内存管理**：几乎每个游戏引擎都有一个或多个自定义内存分配系统，以保证高速的内存分配及释放，并控制内存碎片所造成的负面影响（见5.2.1.4节）。
- **数学库**：游戏本质上就是高度数学密集的。因此，每个游戏引擎都有一个或以上数学库，提供矢量（vector）、矩阵（matrix）、四元数（quaternion）旋转、三角学（trigonometry）、直线/光线/球体/平截头体（frustum）等的几何操作、样条线（spline）操作、数值积分（numerical integration）、解方程组，以及其他游戏程序员需要的功能。
- **自定义数据结构及算法**：除非引擎设计者想完全依靠第三方软件包，如STL，否则引擎通常要提供一组工具去管理基础数据结构（链表、动态数组、二叉树、散列表等），以及算法（搜寻、排序等）。这些数据结构及算法有时需要手工编码，以减少或完全消去动态内存分配，并保证在目标平台上的运行效率为最优。

各个核心引擎系统会于本书第二部分详述。

图 1.17: 核心系统层。

1.6.7 资源管理

每个游戏引擎都有某种形式的资源管理器，提供一个或一组统一接口，去访问任何类型的游戏资产及其他引擎输入数据。有些引擎使用高度集中及一致的方式（例如虚幻的包（package）、OGRE的`ResourceManager`类）。其他引擎使用专案（ad hoc）方法，比如让程序员直接读取文件，这些文件可能来自磁盘，也可能来自压缩文件（如雷神之锤引擎使用的PAK文件）。图1.18显示了典型的资源管理层。

图 1.18: 资源管理器。

1.6.8 渲染引擎

任何游戏引擎中，渲染引擎是最大及最复杂的组件之一。渲染器有很多不同的架构方式。虽然没有单一架构方式，但是大多数现在的渲染引擎都有些通用的基本设计哲学，这些哲学大部分是由底层三维图形硬件驱动形成的。

渲染引擎的设计通常采用分层架构（layered architecture），以下会使用这行之有效的方法。

1.6.8.1 低阶渲染器

如图1.19所示的**低阶渲染器**（low-level renderer）包含引擎中全部原始的渲染功能。这一层的设计着重于高速渲染丰富的几何图元（geometric primitive）集合，并不太考虑那些场景部分是否可见。这组件可以分拆为几个子组件，以下分别讨论。

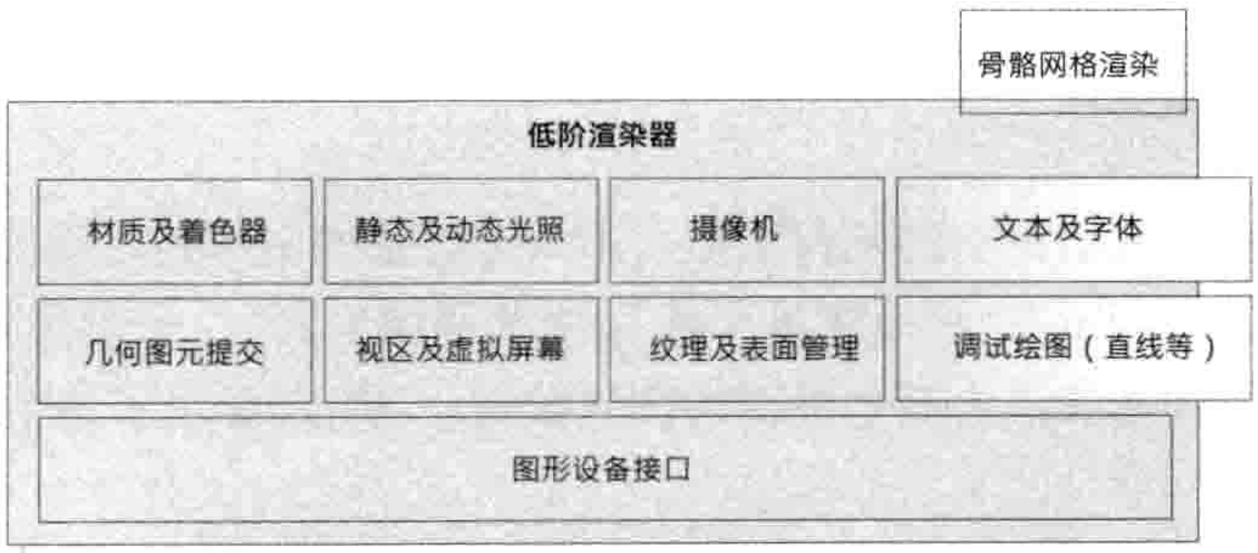


图 1.19: 低阶渲染引擎。

图形设备接口

使用图形SDK，如DirectX及OpenGL，都需要编写不少代码去枚举图形设备，初始化设备，建立渲染表面（如后台缓冲、模板 / stencil缓冲）等。这些工作通常由笔者称为**图形设备接口**（graphics device interface）的组件负责（然而各个引擎都有自己的术语）。

在PC游戏中，程序员须编写代码把渲染器整合到Windows消息循环中。通常要编写“消息泵（message pump）”去处理等待中的Windows消息，其余时间则尽快不断地执行渲染循环。这样做，会使游戏的键盘轮询和渲染器的屏幕更新挂钩。这种耦合令人不快，我们可以再进一步，使这种依赖最小化。以后会深入探讨这个课题。

其他渲染器组件

低阶渲染层的其他组件一起工作，目的是要收集须提交的**几何图元**（geometric primitive，又称为**渲染包**/render packet）。几何图元包括所有要绘画之物，如网格（mesh）、线表（line list）、点表（point list）、粒子（particle）、地形块（terrain patch）、字符串等。最后，把收集到的图元尽快渲染。

低阶渲染器通常提供视区（viewport）抽象，每个视区结合了摄像机至世界矩阵（camera-to-world matrix），三维投影参数如视野（field of view）、近远剪切平面（near/far clipping plane）的位置等。低阶渲染器也使用**材质系统**（material system）及**动态光照系统**（dynamic lighting system）去管理图形硬件的状态和游戏的着色器（shader）。每个已提交的图元都会关联到一个材质及被照射的n个动态光源。材质是描述当渲染图元时，该使用什么纹理（texture），设置什么设备状态，并选择哪一对顶点着色器（vertex shader）和像素着色器（pixel shader）。光源则决定如何应用动态光照计算于图元上。光照和着色是个复杂的课题，计算机图形学有很多优质书籍[14]、[42]、[1]深入探讨这个课题。

1.6.8.2 场景图/剔除优化

低阶渲染器绘画所有被提交的几何图形，不太考虑那些图形是否确实为可见（除了使用背面剔除（back-face culling）和摄像机平截头体的剪切平面）。一般需要较高层次的组件，才能基于某些可视性判别算法去限制提交的图元数量。图1.20显示了这个软件层。

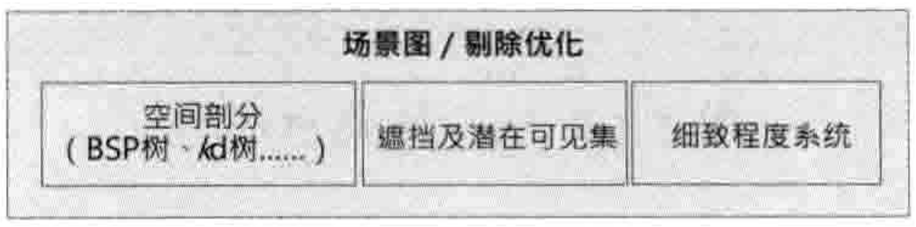


图 1.20: 典型的场景图/剔除优化层，负责剔除优化。

非常小的游戏世界可能只需要简单的**平截头体剔除**（frustum cull）算法（即去除摄像机不能“看到”的物体）。比较大的游戏世界则可能需要较高阶的**空间细分**（spatial subdivision）数据结构，这种数据结构能快速判别潜在可见集（potentially visible set，PVS），

令渲染更有效率。空间分割有多种形式，包括二元空间分割树（binary space partitioning，BSP tree）、四叉树（quadtree）、八叉树（octree）、kd树、包围球树（bounding sphere tree）等。空间分割有时候称为场景图（scene graph），尽管技术上场景图是另一种数据结构，并不归入空间分割。此渲染引擎软件层也可应用入口（portal）及遮挡剔除（occlusion culling）等方法。

理论上，低阶渲染器无须知道其上层使用哪种空间分割或场景图。因此，不同的游戏团队可以重用图元提交代码，并为个别游戏的需求精心制作潜在可见集判别系统。开源渲染引擎OGRE[54]正是运用这一原则的好例子。OGRE提供即插即用的场景图架构。游戏开发者可以选择其中一个已实现的场景图设计，或是自定义一个。

1.6.8.3 视觉效果

如图1.21所示，当代游戏引擎支持广泛的视觉效果，包括：

- 粒子系统（particle system），用作烟、火、水花等。
- 贴花系统（decal system），用作弹孔、脚印等。
- 光照贴图（light mapping）及环境贴图（environment mapping）。
- 动态阴影（dynamic shadow）。
- 全屏后期处理效果（full-screen post effect）[55]，在渲染三维场景至屏外缓冲（off-screen buffer）后使用。

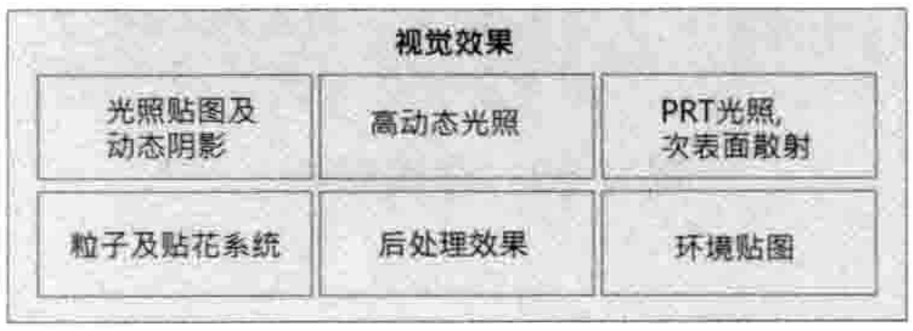


图 1.21: 视觉效果。

一些全屏幕后期处理效果如下。

- 高动态范围（high dynamic range，HDR）光照[56]及敷霜效果（bloom）。
- 全屏抗锯齿（full-screen anti-aliasing，FSAA）。

[54] http://www.ogre3d.org

[55] 译注：比较流行的写法是“(full-screen) post-processing effect”，所以这里采用“全屏后期处理效果”的译法。

[56] 译注：比较准确地说，HDR色调映射（tone mapping）是后制效果，而HDR光照是在后制之前进行的。

- 颜色校正（color correction）及颜色偏移（color-shift）效果，包括略过漂白（bleach bypass）、饱和度（saturation）、去饱和度（desaturation）等。

游戏引擎常有**效果系统**组件，专门负责管理粒子、贴花、其他视觉效果的渲染需要。粒子和贴花系统通常是渲染引擎的独立组件，并作为低阶渲染器的输入端。另一方面，渲染引擎通常在内部处理光照贴图、环境贴图、阴影。全屏后期处理效果可以在渲染器内实现，或在运行于渲染器输出缓冲的独立组件内实现。

1.6.8.4 前端

大多数游戏为了不同目的，都会使用一些二维图形去覆盖三维场景。这些目的包括：

- 游戏的**平视显示器**（heads-up display，HUD）。
- 游戏内置菜单、主控台、其他**开发工具**（可能不会随最终产品一起发行）。
- 游戏内置**图形用户界面**（graphical user interface，GUI）让玩家操作角色装备，配置战斗单元，或完成其他复杂的游戏任务。

图1.22显示了前端层。这类二维图形通常会用附有纹理的四边形（quad）（一对三角形）结合正射投影（orthographic projection）来渲染。另一个方法是用完全三维的四边形公告板（billboard）渲染，这些公告板能一直面向摄像机。

图 1.22: 前端图形。

这一层也包含了**全动视频**（full-motion video，FMV）系统，该系统负责播放之前录制的全屏幕电影（可以用游戏引擎录制，也可以用其他渲染软件录制）。

另一个相关的系统是**游戏内置电影**（in-game cinematics，IGC）系统，该组件可以在游戏本身以三维形式渲染电影情节。例如，玩家走在城市中，两个关键角色的对话可用IGC实现。IGC可能包括或不包括玩家角色。IGC可以故意暂停游戏，期间玩家不能控制角色；IGC也可悄悄地整合在游戏过程中，玩家甚至不会发觉有IGC在运行。

1.6.9 剖析和调试工具

游戏是实时系统，因此，游戏工程师经常要剖析游戏的性能，以便优化。此外，内存资源通常容易短缺，开发者也要大量使用内存分析工具（memory analysis tool）。图1.23显示了剖析和调试工具层。这层包括剖析工具和游戏内置调试功能。调试功能包括调试用绘图、游戏内置菜单、主控台，以及能够录制及回放游戏过程的功能，方便测试和调试。

图 1.23: 剖析和调试工具。

市场上有很多优良的通用软件剖析工具（profiling tool），例如：

- Intel公司的**VTune**。
- IBM公司的**Quantify**和**Purify**（Purify是**PurifyPlus**工具套件的一部分）。
- Compuware公司的**Bounds Checker**。

可是，多数游戏也加入自制的剖析及调试工具，常包括以下功能。

- 手工插入测量代码，为某些代码计时。
- 在游戏进行期间，于屏幕上显示性能统计数据。
- 把性能统计写入文字或Excel文件。
- 计算引擎及子系统所耗的内存，并显示在屏幕上。
- 在游戏过程中或结束时，把内存使用率、最高使用率、泄漏等统计输出。
- 容许在代码内布满调试用打印语句（print statement），可以开关不同的调试输出种类，并设置输出的冗长级别（verbosity level）。
- 游戏事件录制及回放的能力。这很难做得正确，倘若做对，便是追踪bug的非常宝贵的工具。

1.6.10 碰撞和物理

碰撞检测（collision detection）对每个游戏都很重要。没有碰撞检测，物体会互相穿透，并且无法在虚拟世界里合理地互动。一些游戏包含真实或半真实的动力学模拟（dynamics simulation）。这在游戏业界里称为“物理系统（physics system）”，但比较正确的术语是**刚体动力学模拟**（rigid body dynamics）；因为游戏中通常只考虑刚体的运动（motion），以及产生运动的力（force）和力矩（torque）[57]。研究运动的物理分支是运动学（kinematics），而研究力和力矩是动力学（dynamics）。图1.24显示了这一软件层。

图 1.24: 碰撞和物理子系统。

碰撞和物理系统一般是紧密联系的，因为当碰撞发生时，碰撞几乎总是由物理积分及约束满足（constraint satisfaction）逻辑来解决的。时至今日，很少有游戏公司会编写自己的碰撞及物理引擎。取而代之，引擎通常使用第三方的物理SDK，例如：

- **Havok**是今天的业界标准，功能丰富，在不同平台上也运行顺畅。
- **PhysX**是由NVIDIA公司提供的另一个优良的碰撞及动力学引擎，已整合到虚幻引擎3，也可以在PC游戏开发上免费使用。PhysX原来是为Ageia公司的物理加速硬件而开发的接口，但PhysX现在已属NVIDIA公司，并由NVIDIA公司负责发行。NVIDIA公司也改写PhsyX，使它能运行在该公司的最新GPU上。

互联网上也有开源的物理和碰撞引擎。最知名的是Open Dynamics Engine（ODE）[58]。

[57]译注：刚体在游戏中最常见，但也有些游戏使用软体动力学（soft body dynamics）、流体动力学（fluid dynamics）或其他物理分支，应用在游戏性或视觉效果上。

[58]http://www.ode.org

此外，I-Collide[59]、V-Collide[60]和RAPID[61]是流行的非商业碰撞检测系统[62]。这3个系统都是由北卡罗来纳大学（University of North Carolina，UNC）研发。

1.6.11 动画

含有机或半有机角色（人类、动物、卡通角色，甚至机器人）的游戏，就需要动画系统。游戏会用到5种基本动画。

- 精灵/纹理动画（sprite/texture animation）。
- 刚体层次结构动画（rigid body hierarchy animation）。
- 骨骼动画（skeletal animation）。
- 每顶点动画（per-vertex animation）。
- 变形目标动画（morph target animation）。

骨骼动画让动画师使用相对简单的骨头系统，去设定精细三维角色网格的姿势。当骨头移动，三维网格的顶点就相继移动。虽然有些引擎支持变形目标及顶点动画，但现今游戏中，骨骼动画仍然是最盛行的动画方式。因此，本书会集中讨论骨骼动画。图1.25显示了一个典型的骨骼动画系统。

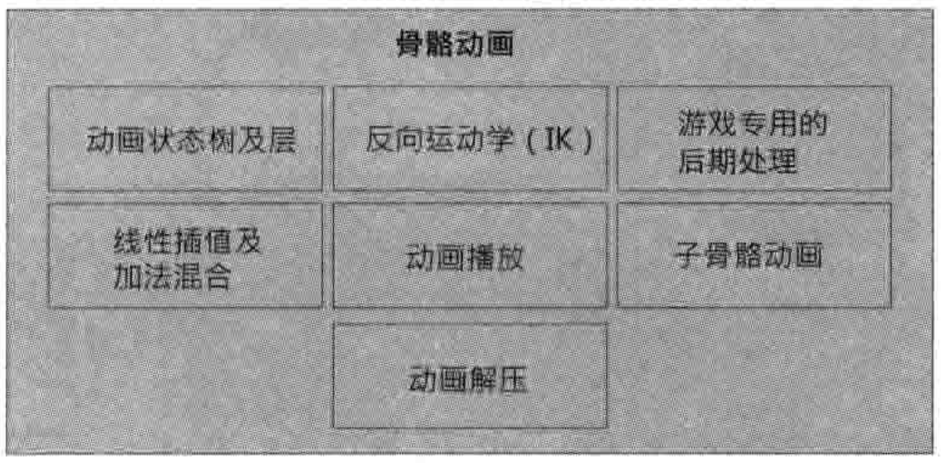


图 1.25: 骨骼动画子系统。

在图1.11中，骨骼网格渲染组件是连接渲染器和动画系统的桥梁。虽然这些组件能非常紧密地合作，但它们的接口还是有明确定义的。动画系统生成骨骼中所有骨头的姿势，这些姿势以矩阵调色板（matrix palette）形式传至渲染引擎。之后，渲染器利用矩阵表去转换顶点，每个顶点用一个或多个矩阵生成最终混合顶点位置。此过程称为**蒙皮**（skinning）。

[59] http://gamma.cs.unc.edu/I-COLLIDE

[60] http://gamma.cs.unc.edu/V-COLLIDE

[61] http://gamma.cs.unc.edu/OBB

[62] 译注：此处的几个系统都是学术性比较重的，具工业强度的开源实时碰撞侦测系统有Bullet（http://bulletphysics.org）。译者估计，Bullet是因为支持连续碰撞侦测（continuous collision detection）而得名（非连续的系统会容易让高速子弹穿过薄墙）。另外，Bullet使用了GJK算法，比ODE更容易支持不同的几何图形。

当使用**布娃娃**（ragdoll）时，动画和物理系统便产生紧密耦合。布娃娃是无力的（经常是死了的）角色，其运动完全由物理系统模拟。物理系统把布娃娃当作受约束的刚体系统，用模拟来决定身体每部分的位置及方向。动画系统计算渲染引擎所需的矩阵表，用来在屏幕上绘画角色。

1.6.12 人体学接口设备

游戏皆要处理玩家输入，而输入来自多个**人体学接口设备**（human interface device，HID），例如：

- 键盘和鼠标。
- 游戏手柄（joypad）。
- 其他专用游戏控制器，如方向盘、鱼竿、跳舞毯、Wii遥控器（WiiMote）[63]等。

该组件有时称作**玩家输入 / 输出**（player I/O）组件，因为除了输入功能，一些人体学接口设备也提供**输出**功能，如游戏手柄的力反馈/震动、Wii摇控器的音频输出等。图1.26显示了典型的人体学接口设备层。

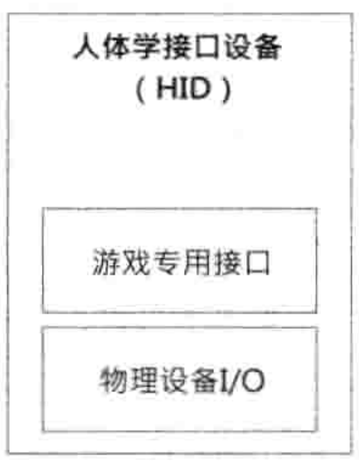


图 1.26: 玩家输入 / 输出系统，也称为人体学接口设备（HID）层。

在架构HID引擎时，通常让个别硬件平台游戏控制器的低阶细节与高阶游戏操作脱钩。HID引擎从硬件取得原始数据，为控制器的每个摇杆（stick）设置环绕中心点的死区（dead zone）[64]，去除按钮抖动（de-bounce），检测按下和释放按钮事件，演绎加速计（accelerometer）的输入并使该输入平滑（例如，来自PlayStation 3的Sixaxis控制器），以及其他处理等。HID引擎通常容许玩家调整输入配置，即自定义硬件控制到逻辑游戏功能的映射。HID引擎也可能包含一个系统，负责检测弦（chord）（即数个按钮一起按下）、序列

[63]译注：正式名称为Wii Remote。Wii摇控器也可接配多种延伸设备，例如双节棍（Nunchuk）、经典控制器（classic controller）、Wii MotionPlus等。

[64]译注：因摇杆是模拟（analog）输入，并不能精准表示中心点。当玩家放开摇杆让其复位时，取得的数值可能抖动。因此要设置一个范围，忽略范围内的输入，即死区。

（sequence）（即按钮在时限内顺序按下）、手势（gesture）（即按钮、摇杆、加速计等输入的序列）。

1.6.13 音频

游戏引擎的音频和图形同样重要。不幸的是，相对于渲染、物理、动画、人工智能及游戏性，音频通常容易被忽视。事实例证：程序员经常在编程时关闭音箱！（实际上，笔者认识一些程序员，他们连音箱或耳机都**没有**。）然而，没有出色的音频引擎，就没有完整的优秀游戏。音频软件层如图1.27所示。

图 1.27: 音频子系统。

音频引擎的功能差异很大。Quake和虚幻的音频引擎提供非常基本的功能，一些游戏团队会为这些引擎加入自定义功能，或用内部方案替换。微软为DirectX平台（PC和Xbox 360）提供一个优秀的音频工具包，名为XACT。艺电也开发了内部的音频引擎SoundR!OT。美国索尼计算机娱乐（SCEA）向其第一方游戏工作室，如顽皮狗，提供一个强大的三维音频引擎Scream，这个引擎已应用在多个PS3作品上，如顽皮狗的《神秘海域：德雷克船长的宝藏》等。然而，即使游戏团队用既有的音频引擎，开发每个游戏时仍然需要大量的定制软件开发、整合工作及注意细节，才可以制作出有高质量音频的最终产品。

1.6.14 在线多人/网络

许多游戏可供多位玩家游玩于同一虚拟世界里。多人游戏最少有4种基本形式

- **单屏多人**（single-screen multiplayer）：两个或以上的HID（游戏手柄、键盘、鼠标等）接到一台街机、PC、游戏主机。多位玩家角色同聚于一个虚拟世界，一个摄像机维持所有角色置于画面中。例子有《任天堂明星大乱斗（Super Smash Brothers）》、《乐高星球大战（Lego Star Wars）》、《圣铠传说（Gauntlet）》。
- **切割屏多人**（split-screen multiplayer）：多个角色同聚于一个虚拟世界，多个HID连

接到一台游戏机器，但每个角色有自己的摄像机。画面分割成多个区域，使每位玩家可以看到自己的角色。

- **网络多人**（networked multiplayer）：多台计算机或游戏主机用网络连接在一起，每个机器接待一位玩家。
- **大型多人在线游戏**（massively multiplayer online game，MMOG）：数百至数千位玩家能在一个巨大、持久（persistent）、在线游戏世界里玩。这些虚拟世界由强大的服务器组运行。

图1.28显示了多人网络层。

图 1.28: 在线多人子系统。

多人游戏和单人游戏有许多颇相似的地方。然而，支持多人游戏，会深切影响某几个游戏引擎组件的设计。游戏世界对象模型、人体学接口设备系统、玩家控制系统、动画系统等都会受影响。把一个现有的单人引擎改装为多人引擎，并非不可能，但会是个望而生畏的任务。然而，有些游戏团队仍可完成这个任务。但是如果可以，最好还是在项目之初就设计多人游戏的功能。

有趣的是，如果进行反向思维——改装多人游戏为单人游戏，问题就再简单不过了。事实上，许多游戏引擎把单人游戏模式当作多人游戏的特例，换言之，单人游戏模式是一个玩家参与的多人游戏。一个知名例子就是雷神之锤引擎的**客户端于服务器之上**（client-on-top-of-server）模式。运行单人游戏模式时，该可执行文件在单个PC上执行，但同时作为客户端和服务器。

1.6.15 游戏性基础系统

游戏性（gameplay）这一术语是指：游戏内进行的活动、支配游戏虚拟世界的规则（rule）、玩家角色的能力（也称为玩家机制/player mechanics）、其他角色和对象的能力、玩家的长短期目标（goal and objective）。游戏性通常用两种编程语言实现，除了用引擎其余部分使用的原生语言，也可用高阶脚本语言，又或者两者皆用。为了连接低阶的引擎子系统

和游戏性代码，多数游戏引擎会引入一个软件层，因无标准术语，笔者称之为**游戏性基础层**（gameplay foundation layer），如图1.29所示。该软件层提供一组功能，以方便实现其上的游戏专有逻辑。

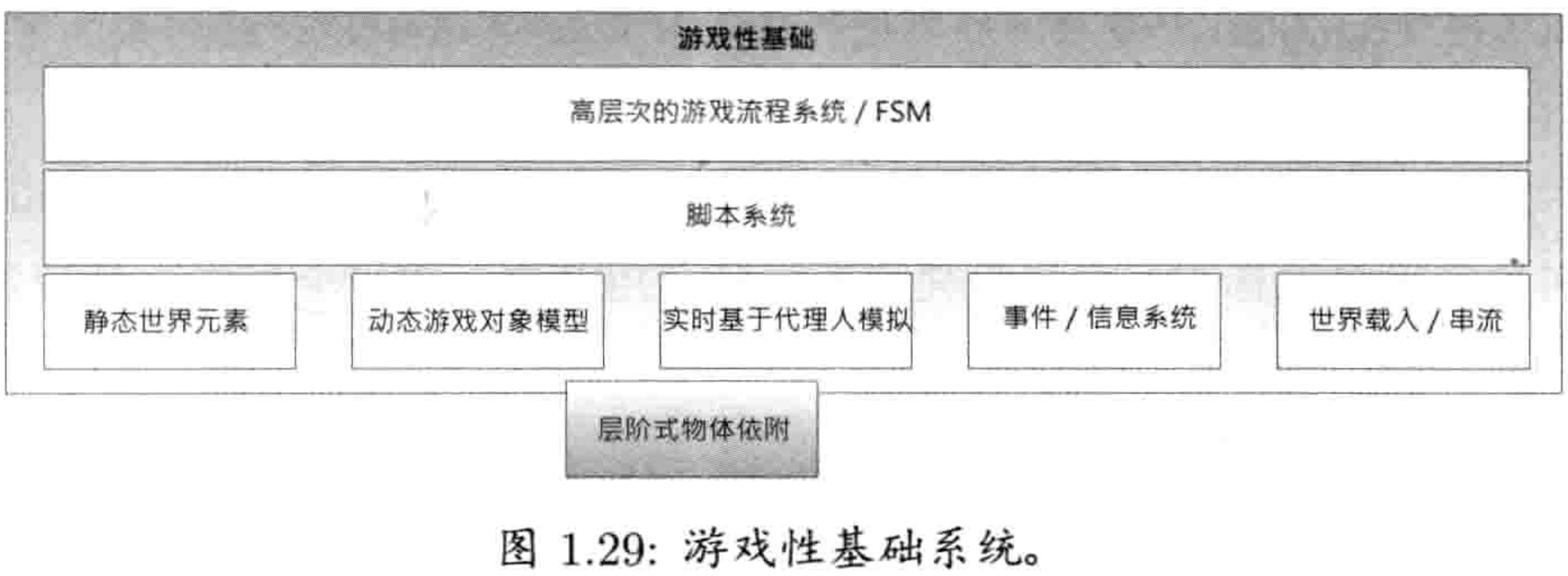


图 1.29: 游戏性基础系统。

1.6.15.1 游戏世界和游戏对象模型

游戏性基础层引入游戏世界的概念，游戏世界含动态及静态元素。游戏世界的内容通常用面向对象方式构建（多数使用面向对象语言，也有例外）。本书中，组成游戏的对象类型集合，称为**游戏对象模型**（game object model）。游戏对象模型为虚拟游戏世界里的各种对象集合提供实时模拟。

典型的游戏对象包括：

- 静态背景几何物体，如建筑、道路、地形（常为特例）等。
- 动态刚体，如石头、饮料罐、椅子等。
- 玩家角色（player character，PC）。
- 非玩家角色（non-player character，NPC）。
- 武器。
- 抛射物（projectile）。
- 载具（vehicle）。
- 光源（可在运行时用于动态场景，也可离线用于静态场景）。
- 摄像机。

游戏对象模型与**软件对象模型**（software object model）紧密结合，并且渗透于整个引擎中。软件对象模型是指，用于实现面向对象软件的一组语言特征、原则（policy）、惯例（convention）。在游戏引擎的语境中，软件对象模型要回答以下问题。

- 游戏引擎是否使用面向对象方式设计？
- 使用什么编程语言？C、C++、Java还是OCaml？
- 怎样组织静态类层阶？一个巨大的层阶，或很多低耦合组件？
- 使用模板（template）及基于原则设计（policy-based design），或传统的多态（polymorphism）？
- 如何参考对象？简明的旧式指针，智能指针（smart pointer）还是句柄（handle）？
- 如何独一无二地标识对象？只凭内存地址，用名字，或用全局统一标识符（global unique identifier，GUID）？
- 如何管理对象的生命周期？
- 如何随时间模拟对象的状态？

14.2节会深入探讨软件对象模型及游戏对象模型。

1.6.15.2 事件系统

游戏对象总要和其他对象通信。有多种方法可完成通信，例如，对象要发消息，可简单调用接收对象的成员函数。事件驱动架构（event-driven architecture），常用于典型图形用户界面，也常用于对象间通信。在事件驱动系统里，发送者建立一个称为**事件**（event）或**消息**（message）的小型数据结构，其中包含要发送的消息类型及参数数据。事件传递给接收对象时，调用接收对象的**事件处理函数**（event handler function）。事件也可储存于队列上，以推迟在未来处理。

1.6.15.3 脚本系统

很多游戏引擎使用脚本语言，使游戏独有游戏性的规则和内容，能更易、更快地开发。没有脚本语言，每次改动引擎中的游戏逻辑或数据结构，都须重新编译、链接方可执行程序。若集成了脚本语言至引擎，要更改游戏逻辑或数据，只需修改脚本代码并重新载入即可。有些引擎容许在游戏运行中重载脚本，其他引擎则需要终止后才能重编脚本。但两种形式所需的作业时间，总比重编、重链程序快得多。

1.6.15.4 人工智能基础

一般而言，人工智能（artificial intelligence，AI）一直是为个别游戏专门开发的软件，一般并不隶属于游戏引擎。但近期游戏开发商找到一些差不多每个AI系统都共有的模式，

使这些基础部分逐渐进入游戏引擎的范畴。

Kynogon公司开发了称为Kynapse的商用AI引擎，充当人工智能基础层。在这个软件层上，可以很容易地开发个别游戏的逻辑。Kynapse有强大的功能特征，包括：

- 用路径节点（path node）和漫游体积（roaming volume）组成的网络，定义AI角色可行走的地区和路径，防止与静态世界几何物体碰撞。
- 在漫游地区边界周围的简化碰撞信息。
- 每个区域的入口 / 出口知识，当中包含敌人能看见或埋伏玩家的地点。
- 基于著名的A*算法的路径搜寻（path finding）。
- 联系碰撞系统及世界模型，进行视线（line of sight）追踪及其他感知（perception）。
- 特制的世界模型，使AI系统知道感兴趣的实体（如朋友、敌人、障碍物）在哪里，并能防止和移动物体做动态回避（dynamic avoidance）。

Kynapse也提供AI决策层的架构，包含**脑**（brain）（每个角色各一）、**代理**（agent）（每个代理负责执行指定任务，例如，从某点走到某点、向某人开火、搜寻敌人等）、**动作**（action）（负责使角色进行基础运动，通常在角色骨骼上播放动画）。

1.6.16 个别游戏专用子系统

在游戏性基础软件层和其他低阶引擎组件之上，游戏程序员和设计师合作实现游戏本身的特性。游戏性系统通常变化很大，并且针对特定游戏来开发。如图1.30所示，这些系统包括但不局限于玩家机制、多种游戏内摄像机、控制非玩家角色的AI、武器系统、载具等。如果可以清楚地分开引擎和游戏，这条分界线会位于特定游戏专用子系统和游戏性基础软件层之间。实际上，这条分界线永远不会是完美的。一些游戏的特定知识，总是会向下渗透到游戏基础软件层中，更有甚者，会延伸至引擎核心。

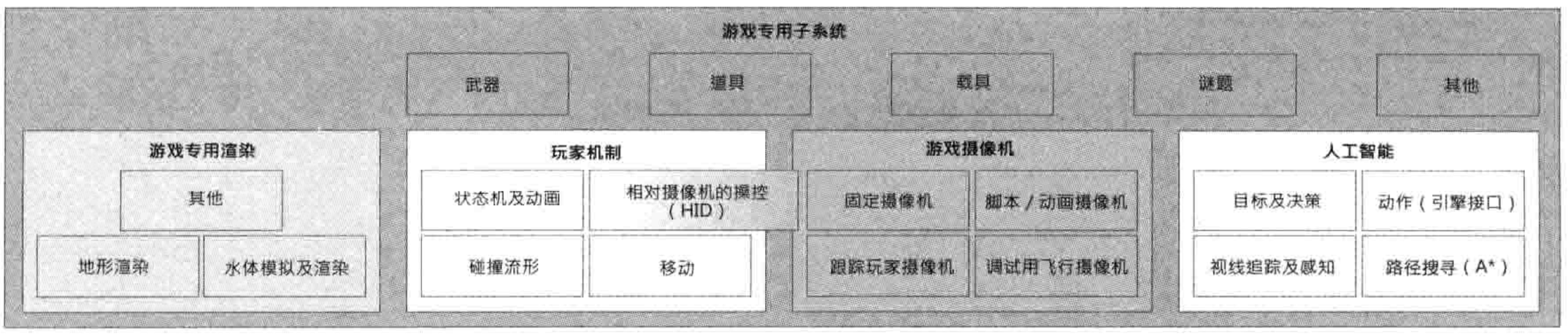


图 1.30: 个别游戏专用子系统。

1.7 工具及资产管道

游戏引擎都需要读取大量数据，数据形式包括游戏资产（game asset）、配置文件、脚本等。图1.31描画了现代游戏引擎中常见的游戏资产。图中，深灰色粗线箭头，指明数据如何从制作原始资产的工具一直流到游戏引擎本身；浅灰色细线箭头，表示各类资源会参考或应用到其他资源。

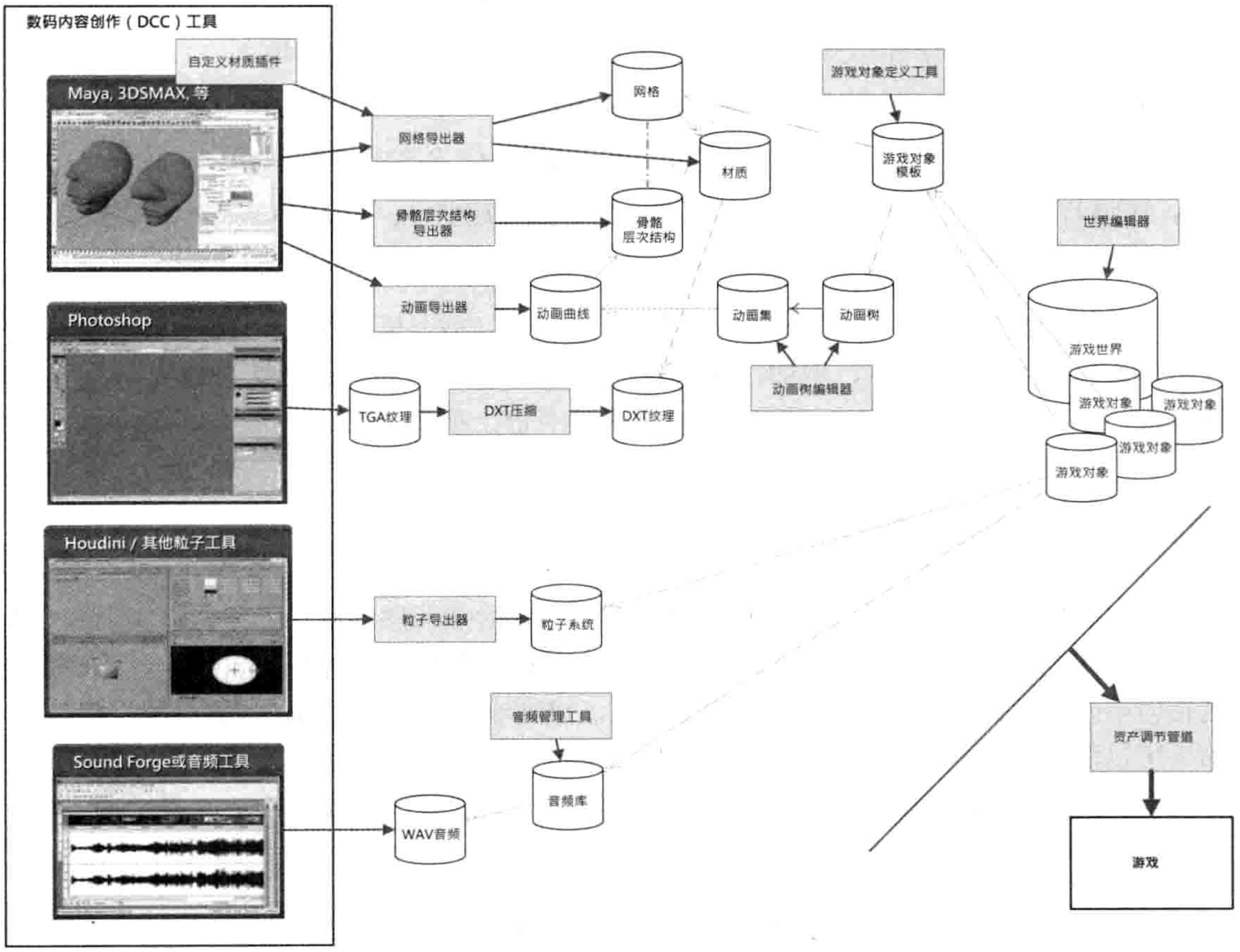


图 1.31: 工具及资产管道。

1.7.1 数字内容创作工具

游戏本质上是多媒体应用。游戏引擎的输入数据形式广泛，例如三维网格数据、纹理位图、动画数据、音频文件等。所有源数据皆由美术人员使用**数字内容创作**（digital content creation，DCC）应用软件制作。

DCC应用软件通常是为制作某一类数据而设，但也有些工具能生产多种数据。例如，Autodesk公司的Maya和3ds Max常用来制作三维网格及动画数据，Adobe公司的Photoshop和其家族成员用于创制及修改点阵图（纹理），SoundForge是制作音频片段的流行工具。然而，有些游戏数据并不能用现成的DCC应用软件去制作。例如，多数游戏引擎提供专门的编辑器，用来设计游戏世界。也有些引擎使用现成工具去编辑游戏世界。笔者见过有些游戏团队，使用3ds Max或Maya作为世界编辑工具，有些团队甚至会开发插件去辅助游戏开发者在这些软件上工作。若去问一些较有经验的游戏开发者，他们大多数可能会记得，曾使用简单的位图编辑器去制作地形高度图（height field），或直接把世界布局人手写到文本文件里。工具不需要精美，游戏团队能用任何到手的工具完成工作。话虽如此，游戏团队想要及时开发高完成度的产品，工具必须相对**易用**，并且绝对**可靠**。

1.7.2 资产调节管道

DCC应用软件所使用的数据格式，鲜有适合直接用于游戏中的，主因有二。

1. DCC软件在内存中的数据模型，通常比游戏所需的复杂得多。例如，Maya的场景节点，以有向非循环图（directed acyclic graph，DAG）储存，包含复杂的互相连接网络。Maya也储存了该文件的所有编辑历史记录。Maya场景中每个物体的位置、方向、比例，都以完整的三维变换表示，此变换又由平移（translation）、旋转（rotation）、缩放（scale）、切变（shear）所组成。游戏引擎通常只需这些信息的一小部分就能在游戏中渲染模型。
2. 在游戏中读取DCC软件格式的文件，其速度通常过慢。而有些格式更是不公开的专有格式。

因此，DCC软件制作的数据，通常要导出为容易读取的标准格式或自定义格式，以便在游戏中使用。

当数据自DCC软件导出后，有时必须再处理，才能放在游戏引擎里使用。若工作室要为游戏开发多个平台，这些中间文件须按平台做不同处理。例如，三维网格（3D mesh）数据可能导出为某中间文件格式，如XML或简单的二进制格式；之后，可能会合并相同材质的网格，或把太大的网格分割成引擎容许的大小；最后，为方便每个平台读取，用最适合的方式组织网格数据，并包装成内存影像。

从DCC到游戏引擎的管道，有时候称为**资产调节管道**（asset conditioning pipeline）。每个引擎都有某种形式的资产调节管道。

1.7.3 三维模型/网格数据

在游戏中可见的几何图形，通常由两种数据组成。

1.7.3.1 笔刷几何图形

笔刷几何图形（brush geometry）由凸包（convex hull）集合定义，每个凸包则由多个平面定义。笔刷通常直接在游戏世界编辑器中创建及修改。这种制作可渲染几何图形的方法比较“土”，但仍然在使用。

其优点为：

- 制作迅速简单。
- 便于游戏设计师用来建立粗略关卡，制作原型。
- 既可以用作碰撞体积（collision volume），又可用作可渲染几何图形。

其缺点为：

- 分辨率低，难以制作复杂图形。
- 不能支持有关节的（articulated）物体或运动的角色。

1.7.3.2 三维模型（网格）

对细致的场景元素而言，三维**模型**（3D model，也称为**网格**/mesh）优于笔刷几何图形。网格是复杂的图形，由三角形和顶点（vertex）组成。网格也可以由四边形和高次细分曲面（higher order subdivision surface）建立。但现时的图形硬件，几乎都是专门为渲染光栅化三角形而设计的，渲染前须把所有图形转换为三角形。每个网格通常使用一个或多个**材质**（material），以定义其视觉上的表面特性，如颜色、反射度（reflectivity）、凹凸程度（bumpiness）、漫反射纹理（diffuse texture）等。本书中，以“网格”一词代表可渲染的图形，并以“模型”一词代表一个组合对象，可能包含多个网格、动画数据和为游戏而设的其他元数据（metadata）。

网格通常在三维建模软件里制作，如3ds Max、Maya、SoftImage。有个比较新的工具ZBrush，可用直觉方式制作超高分辨率的模型，并向下转为有法线贴图（normal map）的低分辨率模型，以模拟高频率的细节。

我们必须编写导出器（exporter）才能从DCC工具（Max、Maya等）获取数据并储存为引擎可读的格式。DCC软件提供许多标准或半标准的导出格式，但通常都不完全适合游

戏使用（COLLADA可能是例外）。因此游戏团队经常要建立自定义格式及专门的导出器。

1.7.4 骨骼动画数据

骨骼网格（skeletal mesh）是一种特殊网格，为关节动画而绑定到骨骼层次结构（skeletal hierarchy）之上。骨骼网格在看不见的骨骼上形成皮肤，因此，骨骼网格有时候又称为**皮肤**（skin）。骨骼网格的每个顶点包含一组关节索引（joint index），表明顶点绑定到骨骼上的哪些关节。每个顶点也包含一组关节权重（joint weight），决定每个关节对该顶点的影响程度。

游戏引擎需要3种数据去渲染骨骼网格。

1. 网格本身。
2. 骨骼层次架构，包含关节名字、父子关系、当网格绑定到骨骼时的姿势（bind pose）。
3. 一个至多个动画片段（animation clip），指定关节如何随时间而动。

网格和骨骼通常由DCC软件导出成单个数据文件。可是，如果多个网格都绑定到同一个骨骼，那么骨骼最好导出成独立的文件。而动画通常是分别导出，特定时刻可只载入需要的动画到内存。然而，有些引擎支持导出动画库（animation bank）至单个文件，有些引擎更把网格、骨骼、动画全部放到一个庞大的文件里。

未优化的骨骼动画是由以每秒30帧的频率，对骨骼中每个关节（通常达100个或以上）采样（sample），记录成一串4×3矩阵。因此，动画数据生来就是内存密集的，通常会用高度压缩的格式储存。各引擎使用的压缩机制各有不同，有些是专有的。为游戏准备的动画数据，并无单一标准格式。

1.7.5 音频数据

音频片段（audio clip）通常由Sound Forge或其他音频制作工具导出，有不同的格式和采样率（sampling rate）。音频文件可为单声道（mono）、立体声（stereo）、5.1、7.1或其他多声道配置（multichannel configuration）。Wave文件（.wav）最普遍，但其他格式如PlayStation的自适应差分脉冲编码（ADPCM）文件（.vag及.xvag）也是常见的。音频文件通常组织成音频库（audio bank），以方便管理、容易载入及串流。

1.7.6 粒子系统数据

当今的游戏采用复杂的粒子效果（particle effect）。粒子效果由视觉特效的专门设计师制作。一些第三方工具，如Houdini，可制作电影级别的效果，可是，大部分游戏引擎不能渲染Houdini制作的所有效果。因此，多数游戏引擎有自制的粒子效果编辑工具，只提供引擎支持的效果。定制的编辑器，也可以让设计师看到与游戏一模一样的效果。

1.7.7 游戏世界数据及世界编辑器

游戏引擎的所有内容都集合在游戏世界。以笔者所知，并没有商用游戏世界编辑器（world editor）（即和Max或Maya软件等同的游戏世界版本）。然而，不少商用游戏引擎提供优良的世界编辑器。

- 不同版本的**Radiant**游戏编辑器[65]，应用在基于Quake技术的游戏引擎上。
- 《半条命2》的Source引擎提供名为**Hammer**的世界编辑器。
- **UnrealEd**是虚幻引擎的世界编辑器。这款强大的工具也同时作为资产管理工具，管理引擎支持的所有资产类型。

优良的游戏世界编辑器虽难以编写，但它却是优良游戏引擎的极重要部分。

1.7.8 一些构建工具的方法

可用不同方式去构建游戏引擎工具套装。一些工具可能是独立的软件，如图1.32所示。一些工具可能构建在运行时引擎使用的低阶软件层之上，如图1.33所示。一些工具可能嵌入游戏本身。例如，基于Quake和虚幻的游戏都提供游戏内部控制台，供开发者及“modder”在游戏运行期间，输入调试和配置命令。

一个有趣且独特的例子是虚幻引擎的世界编辑器UnrealEd，它同时也作为产管理工具，构建在游戏引擎之内。要执行编辑器，就要执行游戏并在命令行参数中加入“editor”。图1.34显示了这独特的架构风格。这种架构，容许工具存取引擎全部的数据结构，避开了一个常见问题——需要两份数据结构的表示方式，一份用于执行时引擎，一份用于工具。此外，在编辑器里执行游戏是极迅速的（因为事实上游戏已在运行中）。通常，要实现游戏内现场编辑（live in-game editing）功能，是非常棘手的事情。若编辑器是游戏的一部分，这一功能就比较容易开发。然而，游戏内编辑器的设计，也有其独特问题。例如，当引擎崩溃，工具也一样变得不稳定。因此，引擎和工具的紧密耦合，总体来看会降低开发效率。

[65]译注：id Software公司的版本是GtkRadiant，网址为http://www.qeradiant.com/。

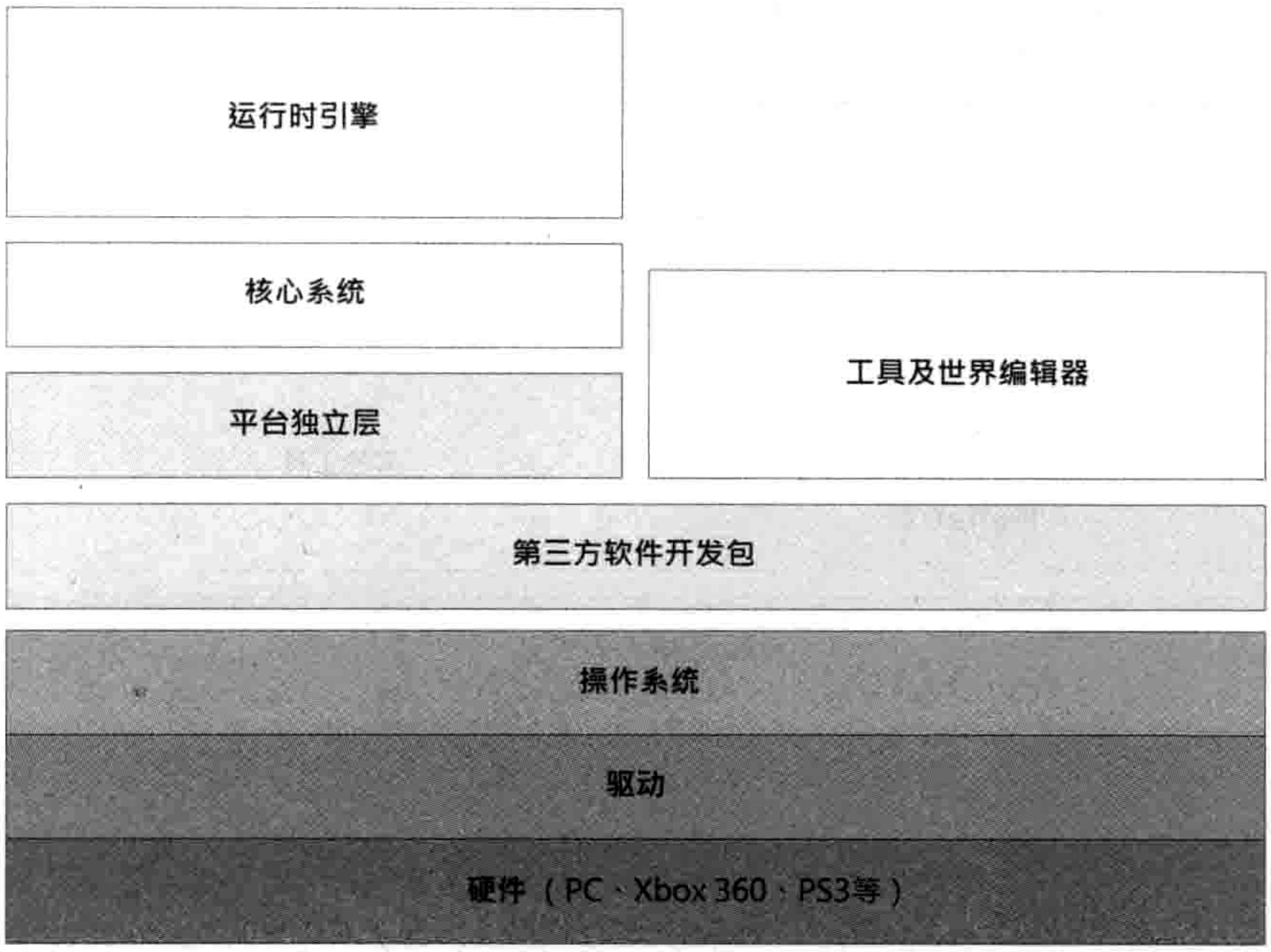


图 1.32: 独立工具架构。

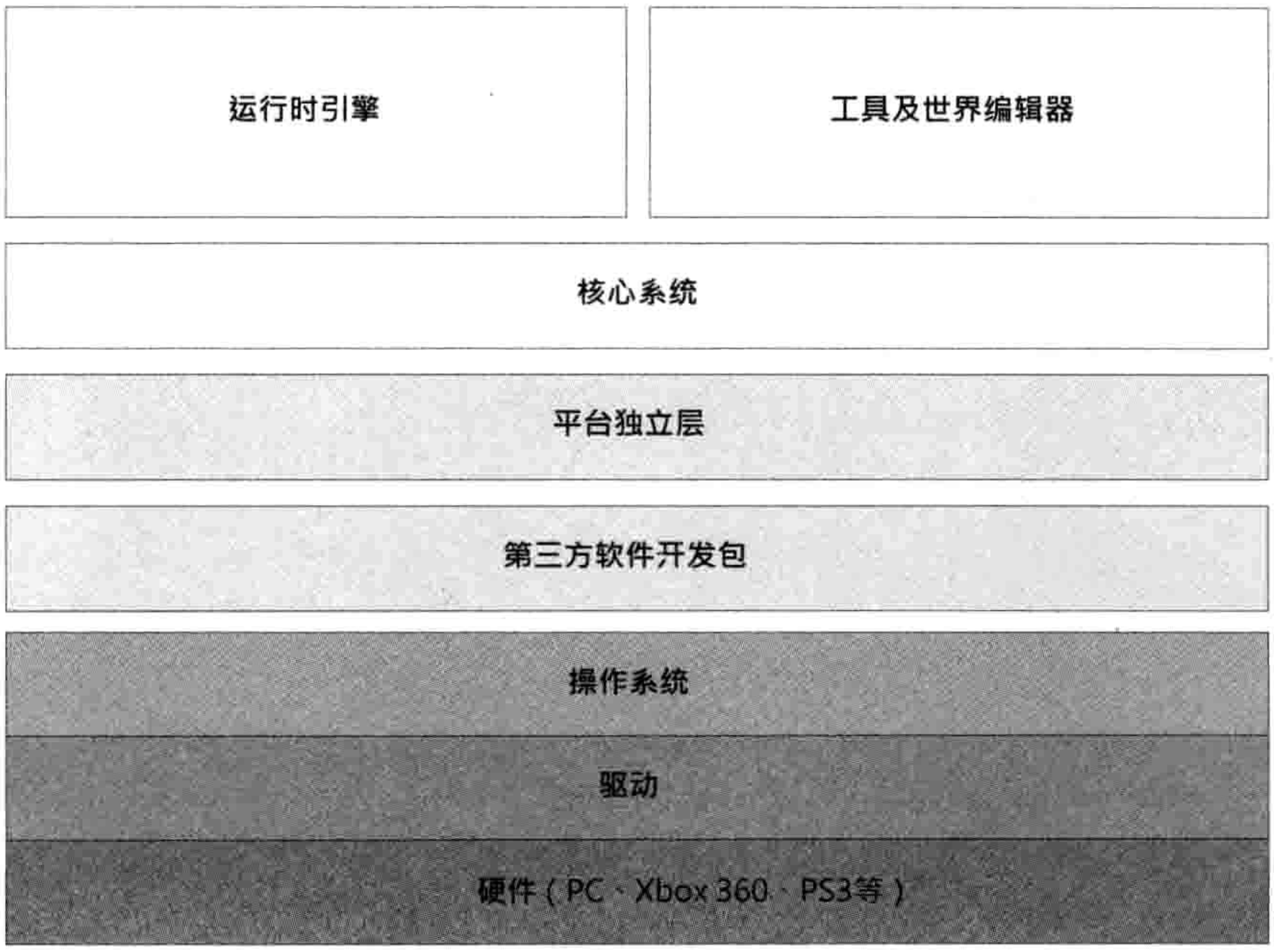


图 1.33: 工具与游戏皆构建在相同框架上。

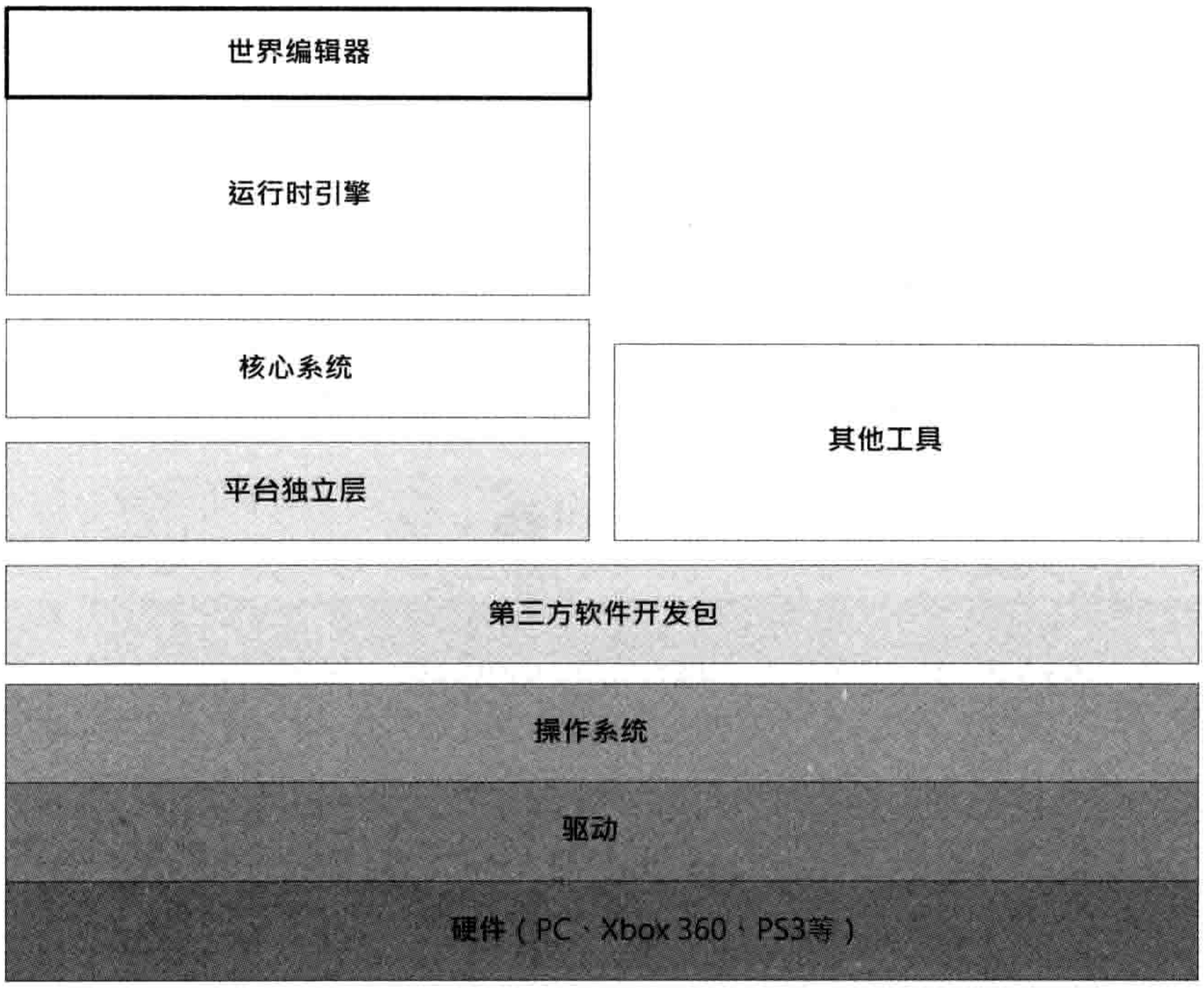


图 1.34: 虚幻引擎的工具架构。

第2章　专业工具

在带领读者踏入美妙的游戏引擎架构旅程之前，我们要充分装备一些基本工具及必需品。以下的两章会回顾软件工程的概念及实践，两者都是旅程所必备的。本章会探讨大部分专业游戏工程师所采用的工具。为了做进一步准备，第3章会回顾几个重要主题，覆盖面向对象、设计模式及大型C++编程等范畴。

在各类软件工程中，游戏开发是要求最高、覆盖领域最广之一员。所以请相信笔者，必须要装备十足，才能安全地渡过一些凶险难关。有些读者可能对这两章的内容非常熟悉，但笔者还是建议这些读者，不要完全略过这些内容。因为读者除了可以轻松地重温这些知识，或许还能学到一两招诀窍。

2.1　版本控制

版本控制系统（version control system）容许多位开发者在同一组文件上工作。由于版本控制系统记录了每个文件的历史，所以它可以追踪文件中的每个改动，需要时还可以把改动还原。版本控制系统容许多位用户同时修改文件，甚至修改同一个文件，并避免互相破坏成果。因为版本控制系统主要供程序员管理源代码，所以有时候又称为**源代码控制**（source control）。尽管版本控制系统也可以用来管理其他类型的文件，一般以文本为佳，下文将探讨其中的原因。许多游戏工作室使用单一版本控制系统，同时管理文本类型的源代码，以及以二进制文件为主的游戏资产，如纹理、三维网格、动画、音频文件等。

2.1.1　为何使用版本控制

多位工程师组成团队合作开发软件时，版本控制至关重要。版本控制系统有以下功能。

- 提供中央版本库（repository），工程师们可以分享其中的代码。

- 保留每个源文件的所有更改记录。
- 提供为某些版本加上标签的机制，供以后提取已加标签的版本。
- 容许代码从主生产线上建立分支（branch）。这一功能经常用来制作示范程序，或是为较旧的软件版本制作补丁（patch）。

源代码控制系统甚至在单人开发项目里也有所用。单人开发项目虽然用不上多人开发的功能，但是其版本控制功能，如维护历史修改记录、为版本添加标签、建立示范程序/补丁的分支、追踪缺陷等，仍然是非常宝贵的。

2.1.2 常见的版本控制系统

也许在读者的游戏工程师生涯里，会接触以下这些最常见的版本控制系统。

- **SCCS和RCS**：源代码控制系统（Source Code Control System，SCCS）和版本控制系统（Revision Control System，RCS）是两个最古老的版本控制系统。两者皆使用命令行界面，主要流行于UNIX上。
- **CVS**：并发版本管理系统（Concurrent Version System，CVS）是高强度、专业级、基于命令行接口的版本控制系统，原本建立在RCS之上（但CVS现在已成为独立工具）。CVS流行于UNIX上，但其他开发平台如微软Windows也能使用。CVS是开源的，并按GPL授权。CVSNT（也称为WinCVS）是一个原生的Windows实现，基于CVS并和CVS兼容。
- **Subversion**：Subversion（简称SVN）是一个开源版本控制系统，其目标是取代并改进CVS。因为Subversion开源且免费，是个人项目、学生项目和小工作室之选。
- **Git**：Git是开源版本控制系统，用于许多受人敬佩的项目，包括Linux内核。在Git开发模型里，程序员把文件的变更提交到一个分支上。之后，该程序员可以轻易把其修改合并到任何一个分支上，因为Git“知道”如何回溯文件的区别（diff），并把区别重新应用在新的基修定版（base revision），这个过程Git称为**衍合**（rebasing）。此开发模型使Git在处理多个代码分支时非常高效和快捷。有关Git的更多信息可参考官网[1]。
- **Perforce**：Perforce是专业级的源代码控制系统，同时支持基于文本和GUI的接口。Perforce成名之处在于其**变更列表**（changelist）的概念。变更列表，指被视为同一个逻辑单元而进行修改的源文件集合。变更列表会以原子方式（atomically）签入（check-in）版本库内，即是说，要么整个变更列表成功提交，要么没有东西提交进去。许多游戏公司使用Perforce，包括顽皮狗和艺电。

[1] http://git-scm.com

- **NxN Alienbrain**：Alienbrain是针对游戏产业而特别设计的强大版本控制系统，具有丰富功能。最著名的特点是支持包含文本及二进制游戏资产的海量数据库，并配合可定制的用户界面，以针对特定的专业，如美术设计师、制作人及程序员等。
- **ClearCase**：ClearCase是专业级的源代码控制系统，是为超大规模的软件项目而设。ClearCase功能强大，并且提供独特的用户接口，以扩展Windows资源管理器的功能。笔者未曾见过游戏业内使用ClearCase，可能是因为其价格较为昂贵。
- **微软Visual SourceSafe**：SourceSafe是轻量级的源代码控制软件包，已成功地应用于一些游戏项目上。

2.1.3 Subversion和TortoiseSVN概览

本书拣选Subversion做重点介绍，原因如下。首先，Subversion是免费的，免费总是好事。以笔者经验，Subversion可以工作得既好又可靠。设置Subversion的中央版本库颇为容易，而且，如果读者不想自己设置版本库，互联网上也有不少免费的版本库服务器可供使用。此外，网上也有许多优秀的Windows和Mac的Subversion客户端，例如Windows上有免费的TortoiseSVN。虽然Subversion并不是大规模商业项目的首选（对于这类项目，笔者的个人之选是Perforce），但是笔者认为Subversion非常适合小型个人和教育性项目。以下介绍如何在Windows 开发平台上设置及使用Subversion，同时复习一些核心概念，这些概念能应用到几乎任何版本控制系统上。

如同其他大部分版本控制系统，Subversion采用客户端/服务器架构（client-server architecture）。服务器负责管理中央版本库，版本库内储存受版本控制的目录层次结构。客户端能连接到服务器并发出操作请求，例如签出某目录树的最后版本、提交一个或多个文件的改动、为修定版加上标签、建立版本库分支等。在此不赘述如何设置服务器，相反，笔者假设读者已有服务器，将集中介绍客户端的设置及使用方法。有关如何设置服务器，读者可参考[37]的第6章。也许，读者可能永远不需要设置服务器，因为可以选择免费的Subversion服务器，例如Google提供的免费Subversion代码代管服务。[2]

2.1.4 在Google上设置代码版本库

开始使用Subversion的最简单方法就是登入`http://code.google.com/`，设置一个免费的Subversion版本库。若没有账号，可先创建一个。单击“Project Hosting（项目代管）”（图2.1），再单击“Create a new project（创建新项目）”，之后输入适当的项目名称，

[2]译注：Google Code只容许开源项目，所有代管文件都是公开的。所以对于非开源项目，读者可能要自己设置服务器。

如“**mygoogleusername**-code”。若读者愿意，还可以输入项目摘要、描述及标签，让全世界的用户能搜索到该项目。设置完毕，便可单击“Create Project（创建项目）”按钮。

图 2.1: Google Code主页中的Project Hosting链接。

创建项目之后，可在Google Code网站上进行管理。项目管理者能加入或移除用户、设置选项及执行进阶任务。但读者真正需要做的下一步是设置客户端及开始使用该版本库。

2.1.5 安装TortoiseSVN

TortoiseSVN是一个流行的Subversion前端程序。TortoiseSVN对Windows资源管理器进行扩展，加入方便的右键菜单，并能在文件图标上叠加信息，显示受版本控制的文件和文件夹状态。

要取得TortoiseSVN，登录`http://tortoisesvn.net/`，便可在“download”页面下载最新版本。双击已下载的.msi文件，并按向导指示进行安装[3]。

TortoiseSVN安装完成后，在Windows资源管理器右键单击任何一个文件夹，会显示TortoiseSVN的延伸菜单。要连接至现有的代码版本库（例如建立在Google Code上的版本库），可以在本地硬盘建立一个文件夹，之后在该文件夹上右键单击，选择“SVN Checkout...”，就会显示图2.2的对话框。在“URL of repository”处输入版本库的网址。若使用Google Code，网址为`https://`**`myprojectname`**`.googlecode.com/svn/trunk`，其中**myprojectname**是建立项目时的名称（如“**mygoogleusername**-code”）。

若忘了版本库的网址，可登录`http://code.google.com/`进入“Project Hosting”页面，单击右上角的“Sign in”链接进行登录，再单击右上角的“Settings”链接，然后在

[3]译注：TortoiseSVN官方网站提供简体中文的说明文档`http://tortoisesvn.net/docs/release/TortoiseSVN_zh_CN/index.html`。

“My Profile”选项卡里列出了所有项目。项目网址为`https://`**`myprojectname`**`.googlecode.com/svn/trunk`，当中**myprojectname**是“My Profile”选项卡里的项目名字。

在“Checkout”对话框中单击“OK”按钮之后，就会显示图2.3的对话框。用户名称（user name）是Google登录名。但password**并非**Google的登录密码，而是一个自动生成的密码。该密码可以在Google Code的“Settings”页面上取得（见上一段落）。勾选对话框的“Save authentication”（储存身份验证）复选框，以后就不用再次登录。在自己的计算机上工作时可勾选此复选框，在共享的计算机上则万万不可。

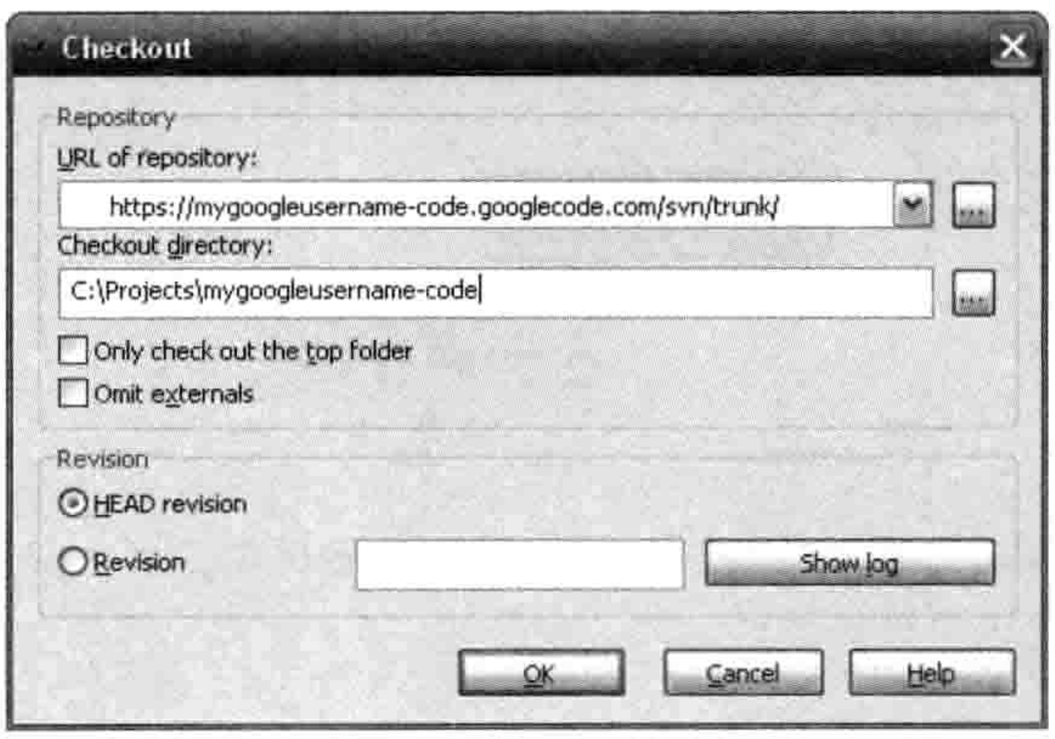


图 2.2: TortoiseSVN签出对话框。

图 2.3: TortoiseSVN用户身份验证对话框。

身份验证后，TortoiseSVN会下载（“签出”）整个版本库的内容至本地磁盘。若是刚设置的版本库，下载的内容就会是——什么都没有！刚才建立的本地文件夹仍然是空的。然而，该文件夹已经连接上Google Code（或其他Subversion服务器）。刷新Windows资源管理器窗口（按F5键），就会看到文件夹图标多了一个绿底白色小钩。此图标表示该文件夹已连接Subversion版本库，并且其内容是最新的[4]。

2.1.6 文件版本、更新和提交

如之前提及，所有版本控制系统，如Subversion等，其最关键功能之一就是可以在服务器里维护中央版本库，即所有源代码的“主（master）”版本，以容许多位程序员在同一软件代码库上工作。服务器维护每个文件的版本历史，如图2.4所示。此功能对于大规模的多程序员项目来说是至关重要的。例如，某人提交了含错的代码，导致“生成失败”，版本控制系统就可以轻易回溯，还原那些改动，并能从日志（log）里找到肇事者！此外，也可以取得任何时间的代码快照（snapshot），用来工作、示范或修正该软件的之前版本。

[4]译注：这个绿色小钩代表该文件夹（或文件）是在正常状态，不代表是最新的版本。

每位程序员可在其计算机上取得本地副本。以TortoiseSVN作为例子，如之前所述，可以“check-out（签出）”版本库取得初始的工作副本。用户应定期更新本地副本，以取得其他程序员的改动。也可右击文件夹，从弹出菜单中选取“SVN Update”以更新本地副本。

在本地副本上修改代码，并不会影响其他程序员（图2.5）。当准备好和其他人分享改动时，便可以**提交**（commit）改动至版本库（也称为**提交**/submit或**签入**/check-in）。可右击想要提交的文件夹，从弹出菜单选取“SVN Commit...”，如图2.6所示的对话框便会弹出，让用户确认改动。

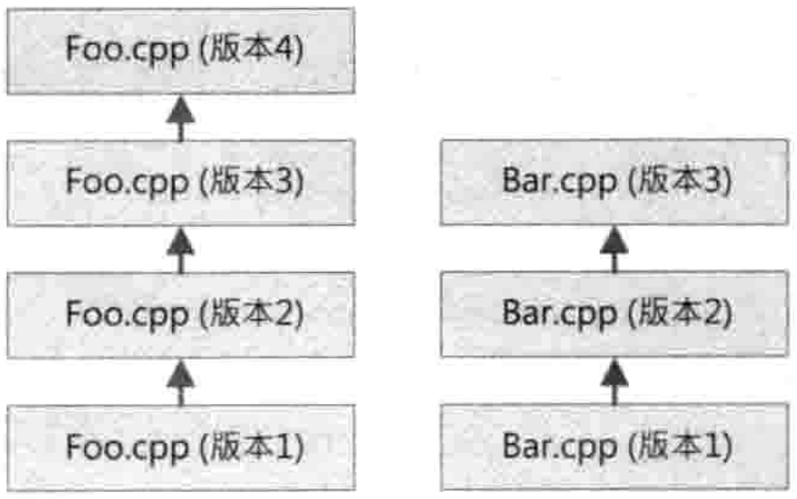


图 2.4: 文件的版本历史。

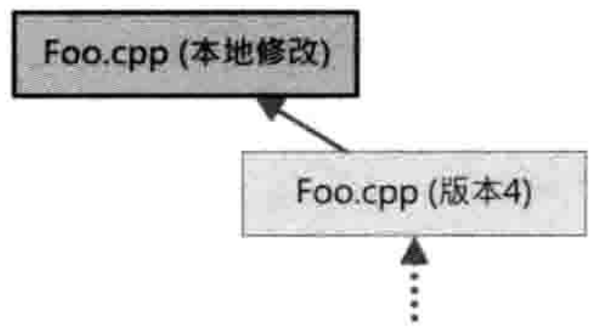


图 2.5: 在本地副本上修改代码。

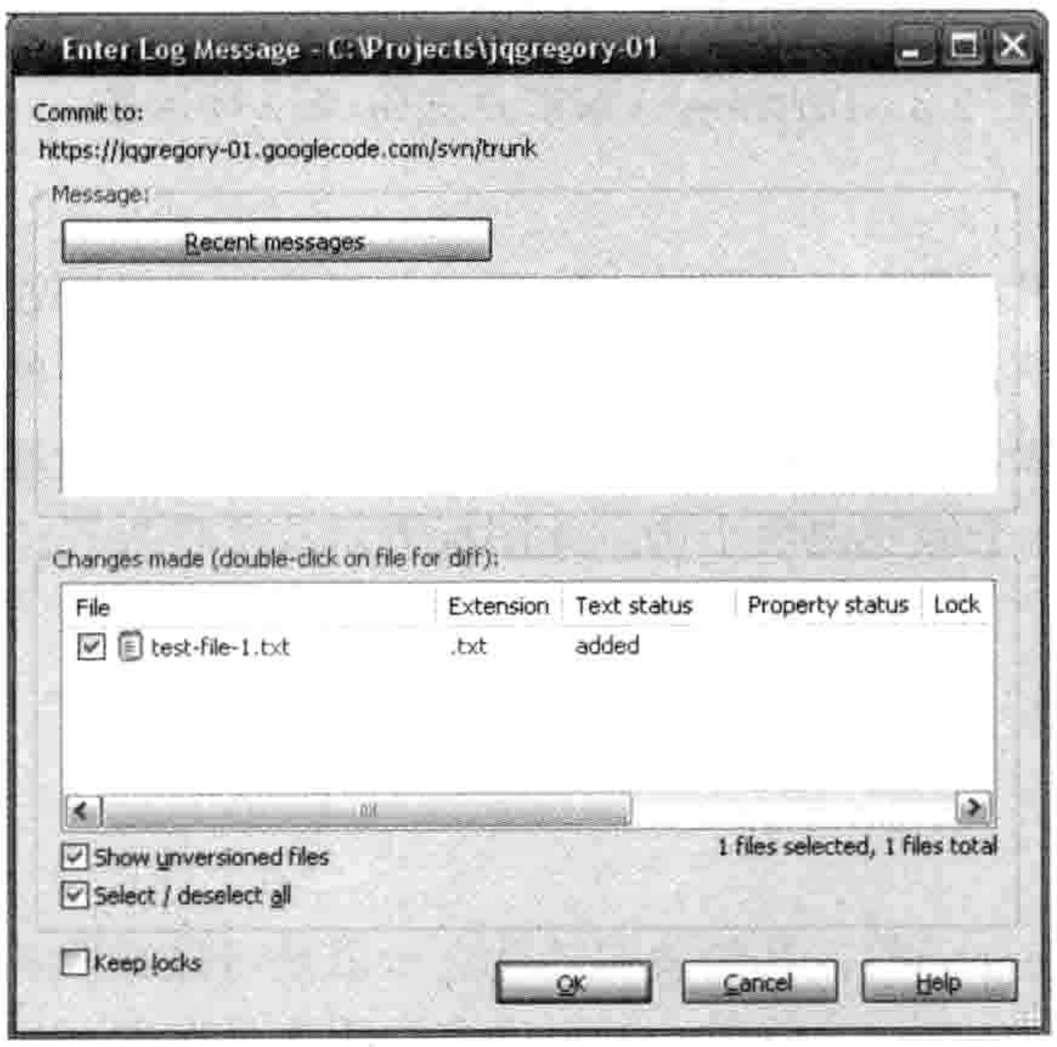


图 2.6: TortoiseSVN提交对话框。

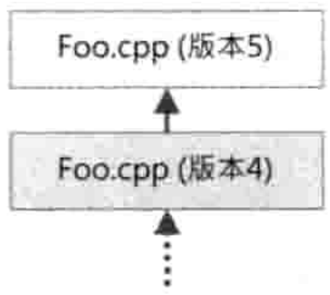


图 2.7: 提交本地改动至版本库。

提交操作时，Subversion会针对每个文件产生其本地版本和最新版本之间的**区别**（diff）。术语“区别”是指差异（difference），通常是由逐行比对文件的两个版本而来。在TortoiseSVN的提交对话框中（图2.6）双击一个文件，就可以看到该文件的本地版本和服

务器上最新版本的区别（即该用户做出的本地改动）。只有改动过的文件（即任何“有区别”的文件）才可提交。提交后，该用户的本地版本成为服务器的最新版本，服务器中也会增加一笔版本历史记录。提交时预设会忽略没改动的文件（即本地版本和版本库的最新版本一致）。图2.7显示一个提交操作的例子。

若用户在提交前创建了新的本地文件，这些文件会在提交对话框中列为“non-versioned（未受版本控制）”。只要勾选这些文件旁边的复选框，就可以把该文件添加至版本库。在本地删除的文件，会在列表中显示为“missing（缺少的）”。若勾选这些文件的复选框，这些文件就会从版本库中删除。此外，还可以在提交对话框中撰写注释，这些注释将会加到版本库的历史日志里，将来该用户及其他团队成员便可知道为何提交那些文件[5]。

2.1.7 多人签出、分支及合并

一些版本控制系统需要**独占签出**（exclusive check-out）。即是说，若用户意图修改某文件，须首先**签出**和**锁定**该文件。签出后的文件在本地磁盘变成可写入的，而那些文件不能被其他人签出。其他受版本控制的文件在本地磁盘是只读的。修改文件完成后，可以签入该文件。此操作会为文件解锁，并提交改动至版本库，使其他人能分享这些改动。独占锁定文件的机制，确保不会有两人同时修改同一个文件。

Subversion、CVS、Perforce及许多其他高质量版本控制系统都提供**多人签出**（multiple check-out）功能——一位用户在编辑某个文件的同时，其他人也可以修改该文件。哪位用户首先提交文件，该文件就能成为版本库内的最新版本。之后，其他人提交该文件时，便必须把自己的改动和之前已提交的文件合并。

由于同一个文件有多于一组改动（区别），版本控制系统必须**合并**（merge）这些改动，以产生该文件的最终版本。合并通常并非难事，实际上版本系统可自动解决很多合并冲突。例如，若一位程序员修改`f()`函数，而另一位程序员修改`g()`函数，那么两人在同一文件的修改行数范围便不一样。这种情况下，合并二人的改动并无冲突，可自动合并。然而，若二人同时修改函数`f()`，第2位提交者便必须进行**三路合并**（three-way merge）（图2.8）。

为了支持三路合并，版本控制系统需要足够聪明，知道用户的目前本地文件是哪个版本。当合并文件时，系统就能得知文件基于哪个版本（即共同祖先版本，图2.8中的第4个版本）。

Subversion允许多人签出，事实上Subversion根本不需要用户明确地签出文件。用户可简单地修改本地文件，任何时候所有本地文件都是可写入的。（顺带一提，笔者认为这

[5]译注：要尽量使每个提交都完整独立、意图清晰，例如新增了某个功能，或修正了某个bug。把提交的意图写进注释对版本管理很重要。良好的注释可以让团队成员更快速地追踪问题。

是Subversion不能延伸应用至大规模项目的原因之一。为找到被修改的文件，Subversion须搜索整个目录下的源文件，而这个过程可能很慢。版本控制系统如Perforce，则会明确地跟踪那些被修改的文件，管理大量代码时就更方便了。但对于小型项目，Subversion的方式没有什么问题。）

右击任何一个文件夹，并从弹出菜单中选择“SVN Commit...”，就可做提交操作。提交时可能会提示，需要与其他人的修改合并。但若更新本地副本之后，并无其他人提交修改，该用户提交时并不需要额外的操作。此功能很方便，但可能是危险的。建议每次提交时，都小心检查提交内容，保证不会提交了无意要修改的文件。当TortoiseSVN显示提交对话框时，可以双击每个文件去显示其区别，最后才单击“OK”按钮提交。

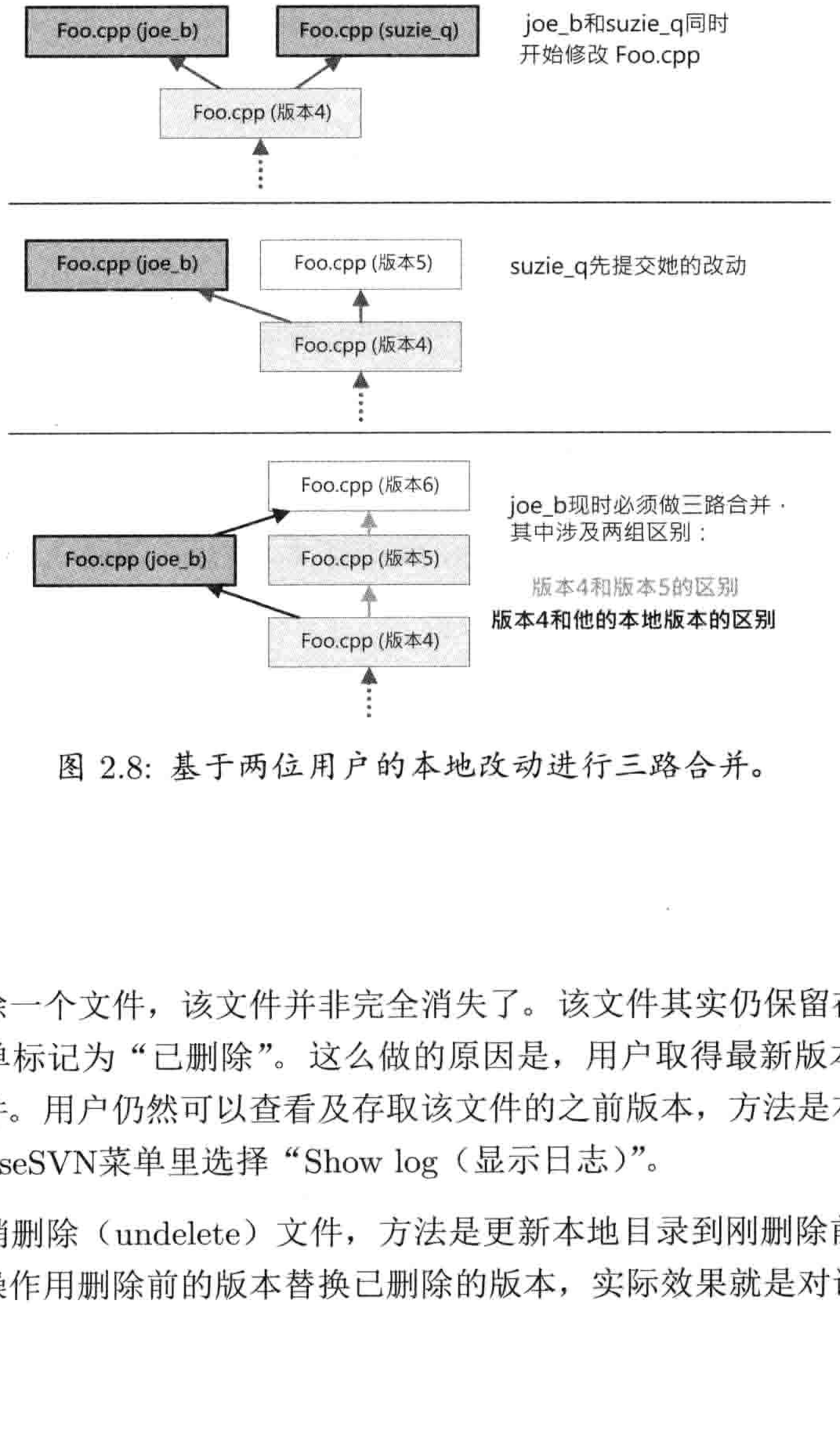


图 2.8: 基于两位用户的本地改动进行三路合并。

2.1.8 删除

从版本库删除一个文件，该文件并非完全消失了。该文件其实仍保留在版本库中，只是其最后版本被简单标记为“已删除”。这么做的原因是，用户取得最新版本时，本地副本便不会再包含该文件。用户仍然可以查看及存取该文件的之前版本，方法是右击该文件所在的目录，并在TortoiseSVN菜单里选择“Show log（显示日志）”。

用户可以撤销删除（undelete）文件，方法是更新本地目录到刚删除前的版本，并重新提交该文件。此操作用删除前的版本替换已删除的版本，实际效果就是对该文件撤销删除。

2.2 微软Visual Studio

编译式语言，如C++，需要使用**编译器**（compiler）及**链接器**（linker），把源代码转换成可执行程序。坊间有不少C++的编译器/链接器，而在微软Windows平台上，最常用的套装软件应该是微软Visual Studio。配备全副功能的Visual Studio专业版（professional）可在代理Windows软件的零售店购得。另外，Visual Studio速成版（Express），即Visual Studio的轻量级版本，可于网站[6]免费下载。微软开发者网络（Microsoft Developer's Network，MSDN）也提供了Visual Studio的在线文档[7]。

Visual Studio不只是编译器和链接器，更是一个**集成开发环境**（integrated development environment，IDE），包含为源代码而设的高质量全能型**文本编辑器**（text editor），以及强大的源代码层级（source-level）和机器层级（machine-level）**调试器**（debugger）。本书主要焦点是Windows平台，因此会较深入地探讨Visual Studio。但本节的大部分内容，也可应用于其他编译器、链接器、调试器。所以笔者建议，即使读者不打算使用Visual Studio，也应稍微了解一下本节有关一般编译器、链接器、调试器的使用秘诀。

2.2.1 源文件、头文件及翻译单元

用C++编写的程序由**源文件**（source file）所组成。常见的C++源文件扩展名为.c、.cc、.cxx和.cpp，这些文件包含程序的大量源代码。因为编译器每次只**翻译**一个C++源文件至机器码，所以在技术上，源文件称为**翻译单元**（translation unit）。

有一种特殊的源文件称为**头文件**（header file）。头文件通常用于在多个翻译单元之间分享信息，例如类型声明及函数原型。C++编译器并不知悉头文件，实际情况是，C++**预处理器**（preprocessor）预先把每个`#include`语句替换为相对应的头文件内容，然后再把翻译单元送交编译器。这是头文件和源文件之间一个细微但非常重要的区别。从程序员角度来看，头文件是独立的文件，但多亏有预处理器把头文件展开，编译器接收到的才都是翻译单元。

2.2.2 程序库、可执行文件及动态链接库

编译翻译单元后，输出的机器码会储存在对象文件（object file）（在Windows下采用.obj扩展名，在基于UNIX的系统里则是.o）中。对象文件中的机器码是：

---

[6] `http://www.microsoft.com/express/download/`

[7] `http://msdn.microsoft.com/en-us/library/52f3sw5c.aspx`

- **可重定位的**（relocatable）：未决定代码的内存地址。
- **未链接的**（unlinked）：未解决的外部函数参考，以及翻译单元外定义的全局数据。

对象文件可以集合成**程序库**（library）。程序库只是一个简单的存档（archive），像zip或tar文件一样，包含零到多个对象文件。程序库只是为方便而设，允许把大量的对象文件集合成单个易用的文件。

链接器把对象文件和程序库**链接**成可执行文件（executable）。可执行文件包含完全解析的机器码，操作系统可载入及执行这些机器码。链接器的工作包括：

- 计算全部机器码的最终相对地址，即当程序执行时机器码在内存中的分布。
- 确保正确地解析每个翻译单元（对象文件）的所有外部函数参考和全局数据。

谨记可执行文件里的机器码仍然是浮动的，文件中的所有指令和数据地址是**相对**于一个任意的基址，而非绝对地址。直至程序载入内存，在执行之前，程序的最终绝对基址才会决定下来。

动态链接库（dynamic linked library，DLL）是一种特殊的库，其行为像正常的静态链接库和可执行文件的混合体。DLL的行为像库，因为它包含函数，供其他多个不同的可执行文件调用。然而，DLL的行为也像可执行文件，因为操作系统能独立地载入DLL，而且DLL可包含启动及终止代码，其执行方式和C++可执行文件的`main()`函数相似。

使用了DLL的可执行文件含有**未完全链接**（partially linked）的机器代码。在最后的可执行文件中，已解析大多数函数及数据参考，但是存于DLL的函数和数据参考则维持未链接的状态。当运行可执行文件时，操作系统需要解析所有未链接的函数。在此过程中，操作系统会找出合适的DLL文件，若该DLL文件不在内存中则要载入，之后需要修正一些内存地址。载入动态链接库是操作系统非常重要的功能，因为这样就可以只更新个别DLL，而不需要更新使用到这些DLL的可执行文件。

2.2.3 项目及解决方案

理解了库、可执行文件和动态链接库的区别之后，便可学习如何创建它们。在Visual Studio里，**项目**（project）是**源文件**的集合。编译项目会产生库、可执行文件或DLL。项目储存为.vcproj扩展名的项目文件。在Visual Studio 2003（版本7）、Vistual Studio 2005（版本8）及Visual Studio 2008（版本9）中，.vcproj文件是XML格式[8] 的。XML格式方便人们阅读，甚至在有需要时可进行手工编辑。

[8]译注：Visual Studio 2010采用新的.vcxproj文件，格式虽是XML，但Schema和之前完全不同。

自从第7版本（Visual Studio 2003）开始，Visual Studio的各版本都采用**解决方案文件**（solution file）（扩展名为.sln的文件），负责容纳及管理项目的集合。解决方案是项目的集合，包含彼此有依赖性及没依赖性的项目，用以生成一个或多个库、可执行文件及DLL。在Visual Studio的图形用户界面中，如图2.9所示，**解决方案资源管理器**（Solution Explorer）通常显示在主视窗的左侧或右侧。

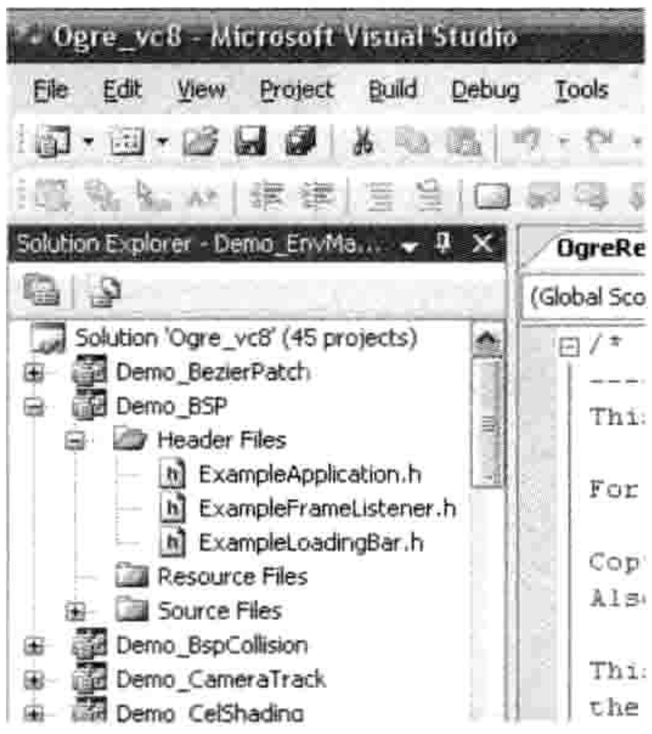


图 2.9: Visual Studio解决方案资源管理器。

解决方案资源管理器是一个树视图。方案本身置于根节点，而项目则是方案的直系子代。源文件和头文件被视为项目的子代。项目可包含任意数量的用户自定义文件夹，嵌套至任何深度。文件夹只作为组织用途，和文件在本机磁盘上的目录结构无关。然而，按照磁盘上的目录结构来设立项目文件夹是个常见惯例。

2.2.4 生成配置

C/C++的预处理器、编译器和链接器都提供大量选项，用来控制代码生成的方式。这些选项通常由执行编译器的命令行设定。例如，使用微软编译器，生成一个翻译单元的典型命令行如下。

```
C:\> cl /c foo.cpp /Fo foo.obj /Wall /Od /Zi
```

此命令行告诉编译器编译但不链接（/c）foo.cpp翻译单元，输出结果到foo.obj对象文件（/Fo foo.obj），打开所有警告（/Wall），关掉所有优化（/Od）并产生调试信息（/Zi）。

现代的编译器提供大量选项，每次生成时都重新指定这些选项，既不现实又易犯错，因此**生成配置**（build configuration）应运而生。生成配置是解决方案内个别项目的预处理器、编译器和链接器的选项集合。程序员可设置任意数量的生成配置，任意命名，并在每个生成

配置中设定不同的预处理器、编译器和链接器选项。默认把同一组选项应用到项目中的每个翻译单元，但也可以在个别翻译单元上做特殊设置，以替代项目的全局设置。（笔者建议，如非必要，避免使用此设置方式，因为很难去分辨哪些.cpp有自定义设置，哪些没有。）[9]

多数项目都有至少两个生成配置，通常名为“调试（Debug）”和“发布（Release）”。发布生成做最终软件出版之用，而调试生成则做开发用途。调试生成比发布生成运行得慢，但调试生成向程序员提供了宝贵的开发及调试信息。

2.2.4.1 常用生成选项

本节会列举一些游戏引擎项目生成设置中最常见的选项。

预处理器设置

C++预处理器处理#include文件的展开，以及处理#define宏（macro）的定义和替换。所有现代的C++预处理器皆有一个极强大的功能，就是可以通过命令行定义预处理宏（因而也能通过生成配置定义）。用这种方式定义宏，和在代码中编写#define指令等效。多数编译器提供此功能的命令行选项-D或/D，此选项可出现多次。

此功能让生成选项和代码沟通，而不需要修改代码本身。举一个常见例子，在调试生成中必然会定义_DEBUG符号，而在发布生成中会定义NDEBUG符号取代之。源代码可以检查这些符号，去“了解”目前的生成调试或发布模式。这称为**条件编译**（conditional compilation）。例如：

```
void f()
{
#ifdef _DEBUG
    printf("Calling function f())\n");
#endif
    // ……
}
```

编译器也可以基于其编译环境和目标平台的信息，自由地加入“魔法”预处理宏到代码中。例如，当编译一个C++文件时，大多数编译器会定义__cplusplus宏，从而能编写代码自动地适应C和C++编译。

[9]译注：有些情况下，可以在源文件或头文件中使用#pragma指令去设置一段代码的编译选项。例如，#pragma optimize("", off)可以关闭优化，方便调试某个翻译单元，甚至某个函数。

又例如，每个编辑器都会通过一个“魔法”宏，让代码识别编译器。当用微软的编译器编译代码时，编译器会定义_MSC_VER宏；当使用GNU编译器（gcc），则会定义__GNUC__宏，其他编译器也如是[10]。与此相似，执行代码的目标平台也是用宏来定义的。例如，生成32位Windows机器的执行代码时，就会定义_WIN32符号。可以利用这些关键功能去编写跨平台代码，因为这些宏使代码“了解”目前被哪个编译器编译，并需要编译至哪个目标平台。

编译器设置

控制编译器产生的对象文件是否包含**调试信息**（debugging information），是最常见的编译选项之一。调试器使用此信息去逐步执行代码、显示变量的值等。调试信息会增大磁盘上的可执行文件大小，也会方便黑客做反向工程。因此，最终发布的可执行文件必会剔除这些调试信息。然而，在开发期间，调试信息是无价之宝，应该经常藏于生成代码中。

另外，也可以控制编译器是否展开**内联函数**（inline function）。如关掉内联函数展开，每个内联函数在内存中只有一份，有唯一的内存地址。这样设置，使用调试器去追踪代码就容易得多，但其明显的代价是放弃了正常内联函数执行速度的提升。

内联函数展开是称为**优化**（optimization）的泛代码转换例子之一。可以使用编译器选项去控制编译器尝试优化代码的进取性（aggressiveness），以及使用哪些优化方法。优化可能会打乱代码里的语句次序，完全剔除一些变量，把变量移到不同地方，或在函数里将CPU寄存器（register）作为新用途重复使用。经优化的代码常会迷惑大多数调试器，令调试器以不同方式对用户“说谎”，并难以观察真实的执行情况。因此，在调试生成中，通常会关上优化选项。这样一来，每个变量、每行代码都会和原来编写的保持完全一致。但是，未经优化的代码，执行时较完全优化的代码慢许多。

链接器设置

链接器也提供多个选项，例如，控制输出文件的类型（如可执行文件、DLL），指定链接哪些外部库至可执行文件，以及指定搜索哪些程序库的路径。惯例之一，调试时，可执行文件链接调试用的库，发布版本则链接优化的库。

链接器选项也可控制堆栈大小、程序载入内存时的首选基址、代码在何平台上执行（以做平台相关的优化），以及许多其他细节选项，不于此展述。

---

[10]译注：_MSC_VER的值为微软编译器的版本号，如用9.0版本编译时，_MSC_VER的值为整数1500。__GNUC__和__GNUC_MINOR__则是gcc的主要及次要副版本号。

2.2.4.2 典型生成配置

通常，游戏项目不止有两种配置。以下是笔者在游戏开发中遇到的一些常见配置。

- **调试**（Debug）：调试生成版本是非常慢的程序版本。此版本关上各种优化，禁用所有函数内联，并且包含完整的调试信息。此生成版本是用来测试新代码，以及调试在开发过程中出现的几乎所有最不平凡（nontrivial）的问题。
- **发布**（Release）：发布生成版本是较快的程序版本，但仍然保留调试信息并开启断言（assertion）。（关于断言可见第3.3.3.3节。）游戏能表现接近最终产品的运行速度，并留有机会去调试问题。
- **制作**（Production）：制作配置，是为生成最终发行给消费者的游戏版本而设。此配置有时也称作“最终（Final）”或“光盘（Disk）”配置。制作配置与发布配置的差别，在于前者去除所有调试信息，通常关上所有断言，并完全启动优化。调试制作生成版本非常棘手，但制作生成版本是最快及最精干的生成类型。
- **工具**（Tools）：有些游戏工作室的工具和游戏本身会共用代码库。此方案中，加入“工具”配置很合理，用以为工具条件编译共用代码。工具配置一般会定义一个预处理宏（如`TOOLS_BUILD`），以告之代码当前是在生成工具用的版本。例如，某个工具可能需要一些C++类提供编辑用函数，而这些函数在游戏中并不需要。那么就可以用`#ifdef TOOLS_BUILD...#endif`指令包围这些函数。由于工具通常也要分调试和发布版本，所以开发者会建立两个工具生成，如命名为“ToolsDebug”及“ToolsRelease”。

混合生成版本

混合生成版本（hybrid build）是指其配置中，大部分翻译单元是发布模式，只有一小部分翻译单元为调试模式。使用这种配置，容易调试当前要监察的代码，而其余的代码能继续以全速运行。

基于文本的生成工具，如make，能很容易设置混合生成。用户能以翻译单元为单位把某些翻译单元设置为调试模式。大致做法是：定义一个make变量，如`$HYBRID_SOURCES`，列举所有要设置为调试模式的翻译单元（.cpp文件）；设置生成规则，编译所有翻译单元的调试及发布两个版本，并将每个对象文件（.obj/.o）按其版本分别输出到两个文件夹；设定最终的链接规则，链接`$HYBRID_SOURCES`列举的对象文件调试版本，以及其他对象文件的发布版本。若设置正确，make的依赖规则能处理余下工作[11]。

[11]译注：因为链接只依赖部分调试和发布版本的对象文件，make就能自动找到对应的翻译单元。并且，每个翻译单元只会编译一个版本（调试或发布）。

可惜，在Visual Studio中并不容易做到同样的事情。因为Visual Studio的生成配置是以项目为单位，而不是以翻译单元为单位的。问题的症结在于不能列举要生成调试模式的翻译单元。然而，若源代码已经组织成库，就**可以**在解决方案层面上，设立“混合”生成配置。此配置可挑选所需项目，并为每个项目（也因而为每个**库**）选择采用调试还是发布版本。虽然这不及按每个翻译单元设置那么有弹性，但若库的粒度足够细，仍是个不错的方法。

生成配置和可测试性

项目支持的生成配置越多，也就越难测试。虽然配置之间可能相差无几，但一些bug仍有可能只出现在某个配置中，而不出现在其他配置中。因此，每个配置都必须彻底测试。多数游戏工作室并不正式测试调试生成版本，因为调试配置主要用于开发新功能时，以及在其他配置遇到问题时做调试之用。然而，若测试人员花大部分时间测试发布配置，那么并不能在制作母片（gold master）[12]的前夜，直接制作出游戏的制作生成版本，并期望它的bug状况与发布生成版本一模一样。实践上，在alpha到beta阶段，测试团队应同样彻底地测试发布及制作生成版本，保证制作生成版本不会暗藏任何烦扰人的意外情况。保持最少数量的生成配置，最有利于测试。事实上，有些工作室为此而不加入制作配置，彻底地测试发布生成后，就直接发行发布生成。

2.2.4.3 项目配置教程

在解决方案资源管理器中右键单击一个项目，在快捷菜单中选“属性（Properties）”就会出现“项目属性页（Project Property Pages）”对话框。对话框左侧的树视图显示了各类设置，其中最常用的4个种类是常规（General）、调试（Debugging）、C/C++和链接器（Linker）。

配置下拉组合框

注意在对话框的左上角有标示为“配置（Configuration）”的下拉组合框。属性页内显示的所有属性，都会应用于个别的生成配置。如果在调试配置里设定了一个属性，不会意味着在发布配置也有同样的设定。

单击配置下拉组合框，可以在列表中选择一个或多个配置，并有“所有配置（All Configurations）”可供选择。这意味着不会在发布配置也有同样的设定。根据经验，最好选择“所有配置（All Configurations）”去编辑大部分的生成配置。这么做，不需要为每个配

[12]译注：gold master是指把软件拿去生产（如复制零售光盘）的版本。

置重复编辑，避免在某个配置中意外犯错。然而，一些在调试和发布配置中的设置需要有所区别。例如，内联函数和代码优化的设置，在调试和发布生成中就截然不同。

常规属性页

图2.10的常规（General）属性页中，最有用的属性如下。

- **输出目录**（Output Directory）：决定生成的最终产品（一个或多个文件）放于哪个目录之下。编译器/链接器最终输出可执行文件、库和DLL[13]。
- **中间目录**（Intermediate Directory）：定义中间文件在生成时输出的目录。中间文件主要为对象文件（.obj扩展名）。中间文件不需要包含在最终程序发行版里，而只是在生成可执行文件、库和DLL过程时所需。因此，中间文件和最终产品（.exe、.lib、.dll文件）放在不同的目录里是不错的主意。

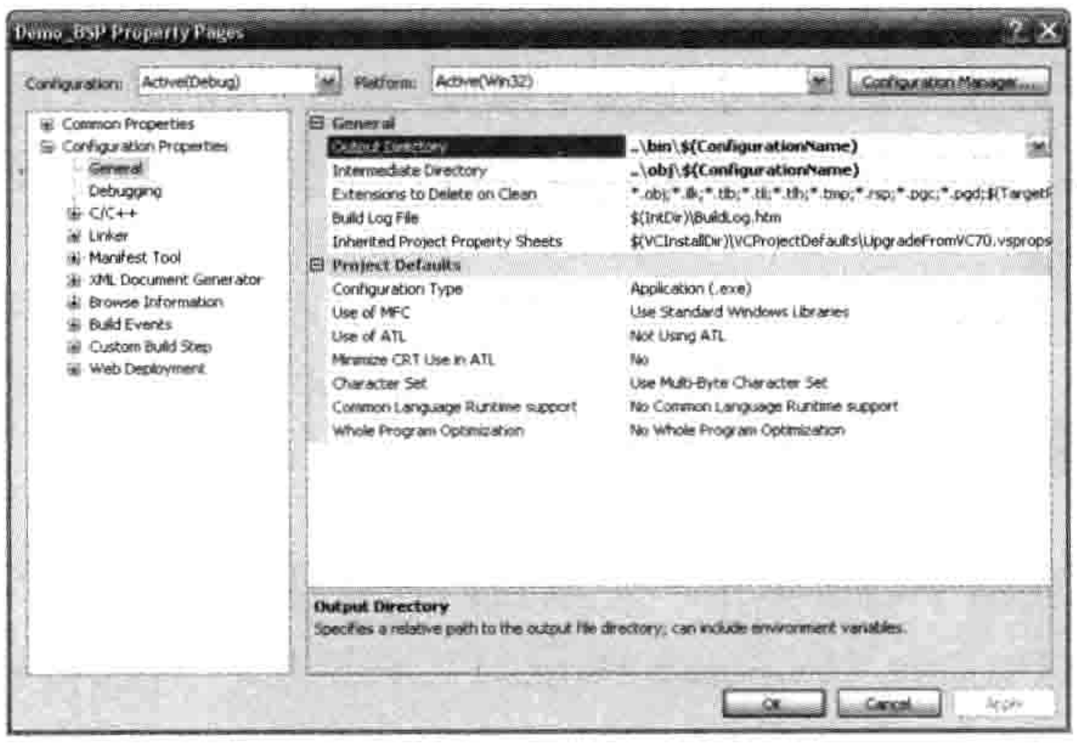


图 2.10: Visual Studio项目属性页之一，常规属性页。

注意Visual Studio提供宏（macro）功能，这些宏能用来在项目属性页中设置目录或其他属性。**宏**的本质是命名变量（named variable），变量的值是按项目配置而全局设置的。

只要把宏的名字用括号包围并前置美圆符（如`$(ConfigurationName)`）便可使用。一些常用的宏列举如下。

- `$(TargetFilename)`：项目生成的最终可执行文件、库或DLL的文件名。
- `$(TargetPath)`：最终可执行文件、库或DLL的绝对路径。
- `$(ConfigurationName)`：生成的配置名称，典型的值为“Debug”或“Release”。

[13]译注：这些文件都是由链接器（而非编译器）所产生的。

- `$(OutDir)`：在该对话框“输出目录（Output Directory）”设置的值。
- `$(IntDir)`：在该对话框“中间目录（Intermediate Directory）”设置的值。
- `$(VCInstallDir)`：安装Visual C++的目录。

相对手工硬性指定属性值，使用宏的好处是，修改宏的全局值会自动影响所有使用该宏的设置。而且，一些宏如`$(ConfigurationName)`能按生成配置自动改变其值，因而可以在所有配置中使用一样的设置。

要参见所有可用的宏，单击输出目录（Output Directory）或中间目录（Intermediate Directory）属性右方的下拉箭头，选择“Edit（编辑）...”，在弹出的对话框中单击“Macros（宏）”按钮。

调试属性页

在调试（Debugging）属性页中，可指定要调试的可执行文件的名称及所在位置。本页也可指定命令行参数，在运行时传递至程序。调试程序的方法会在稍后详细讨论。

C/C++属性页

C/C++属性页控制编译期的语言设置，影响代码如何从源文件编译至对象文件（.obj扩展名）。本页的设置**并不**影响对象文件最终链接至可执行文件或DLL。

在这里，鼓励读者探索C/C++属性页的各个分页，了解有什么设置可用。以下包括一些最常用的设置。

- **常规 / 附加的包含目录**（General/Additional Include Directories）：此属性列举当读取`#include`头文件时所搜寻的目录[14]。**重要提示**：最好使用**相对路径**及/或Visual Studio提供的宏去设置这些目录，如`$(OutDir)`、`$(IntDir)`。这么做，即使把生成目录移到磁盘上的其他位置或其他计算机，编译仍然可以照常运作[15]。
- **常规 / 调试信息格式**（General/Debug Information Format）：此属性控制是否产生信息格式。一般来说，调试及发布配置都包含调试信息，方便在开发游戏时追查问题。最终制作生成应该除去所有调试信息，防止程序被他人修改。

[14] 译注：多个目录可用空间分开，如目录中有空格，应使用双括号包围。

[15] 译注：这里指尽量不使用绝对路径。

- **预处理器 / 预处理器定义**（Preprocessor/Preprocessor Definitions）：此属性可方便地列出任意数量的C/C++预处理符号，当编译源文件时这些符号会被定义。之前的2.2.4.1节包含了对预处理定义符号的讨论。

链接器属性页

链接器（Linker）属性页控制对象文件如何链接成可执行文件或DLL。再次鼓励读者探索各个分页。一些常用的设置如下。

- **常规 / 输出文件**（General/Output File）：设置生成最终产品（一般为可执行文件或DLL）的文件名及所在目录。
- **常规/附加库目录**（General/Additional Library Directories）：如同C/C++属性页的附加包含目录属性，当链接时要读取库或对象文件，就会搜寻此属性列出的目录。
- **输入 / 附加依赖项**（Input/Additional Dependencies）：此属性列出需要和可执行文件或DLL链接的外部库。例如，若要生成使用OGRE的应用程序，就在此属性中加入OGRE的库。

此外，注意Visual Studio提供一些“魔法咒语”去指定需要链接那些库。例如，源代码中的特殊`#pragma`指令，可用来告诉链接器去自动链接某个库。因此，不能从“附加依赖项”看到所有实际上会链接的库（事实上，这是此属性称为**附加**依赖项的原因）。读者或许会注意到，例如，在DirectX应用程序中不需在附加依赖项手工列出所有的DirectX库，仍能正常链接，当中就是运用了`#pragma`指令。

2.2.4.4 创建新的.vcproj文件

创建项目需要正确地设置那么多预处理器、编译器、链接器选项，真是令人无比生畏。笔者经常使用以下两种方法创建Visual Studio项目。

使用向导

Visual Studio提供多元化的向导（wizard）创建不同种类的项目。若能找到合适的向导，此为最简单的创建项目方法。

复制现有项目

笔者创建一个新项目时，若发现已有一个相似并且能运作的项目，经常会先复制.vcproj文件，并在必要时修改该文件副本。此方法在Visual Studio 2005[16]中非常简单，只需复制磁盘上的.vcproj文件，并在解决方案资源管理器右击解决方案，然后在弹出菜单中选择“Add→Existing Project...（添加→现有的项目……）”即可加入复制出来的新项目。

在复制时须注意，老项目的名称储存在.vcproj文件里。因此，当第一次加载该新项目到Visual Studio 2005，该新项目仍保持原来的名称。为纠正此问题，可在解决方案资源管理器中选择该项目，再按F2键为新项目起一个合适的名字。

需要注意，项目创建的可执行文件、库或DLL的文件名是**明确地**在.vcproj里指定的。例如，可执行文件可指定为“`C:\MyGame\bin\MyGame.exe`”或“`$(OutDir)\MyGame.exe`”。这种情况下，需要打开.vcproj文件，并对可执行文件、库、DLL名称或/及其目录路径进行全局查找及替换。这并非难事。因为项目文件是XML格式的，所以把.vcproj文件改为.xml扩展名，就可用Visual Studio（或任何其他XML或文本编辑器）打开[17]。另一漂亮的解决方法是使用Visual Studio的宏系统，去指定项目中所有输出文件。例如，若指定输出可执行文件为“`$(OutDir)$(ProjectName).exe`”，那么项目的名称就会自动反映在输出可执行文件名里。

补充一点，其实大可使用文本编辑器去修改.vcproj文件。实际上，至少在笔者的经验里，这种方法是常用的。例如，假设要把放置图形的头文件的目录移到另一个路径去。与其手工逐个开启项目，打开项目属性页对话框，选择C/C++属性页，并最后修改包含目录，倒不如以XML文本方式编辑项目文件并进行查找及替换，既简单又不易出错。甚至可以在Visual Studio中对大量文件进行“多个文件替换”操作。

2.2.5 调试代码

任何程序员都须学习的最重要技巧之一就是如何高效地调试代码。本节提供一些有用的调试窍门和招式。当中有些能应用在任何调试器中，另一些是微软Visual Studio专用。然而，通常可在其他调试器找到对应Visual Studio的调试功能。因此，即使不使用Visual Studio去调试代码，本节也有帮助。

---

[16]译注：Visual Studio 2008也能使用此方法。

[17]译注：补充一点，如果项目副本和原来项目都置于同一解决方案，Visual Studio会做成冲突。解决方法是用文本方式打开副本，把`<VisualStudioProject>`里的`ProjectGUID`属性替换为另一个GUID。可在Visual Studio中选择“Tools→Create GUID（工具→创建GUID）”去产生新的GUID。

2.2.5.1 启动项目

Visual Studio解决方案可含有多个项目。当中一些项目会生成可执行文件，其余的生成库或DLL。在一个解决方案中，可能含有多个生成可执行文件的项目。然而，某一刻不能调试多于一个程序。因此，Visual Studio提供一个名为“Start-Up Project（启动项目）”的设定[18]。调试器把启动项目视为当前要调试的项目。

在解决方案资源管理器中，启动项目以粗体显示。按F5键会生成启动项目，并在调试器中运行生成的.exe文件（若启动项目生成可执行文件）。

2.2.5.2 断点

断点（breakpoint）是代码调试的基本所需。每个断点都可以使得调试器在程序某行停下来，以便视察程序的状况。

在Visual Studio中，可选择某行代码并按F9键切换断点。当程序运行到含断点的代码行时，调试器便会停止程序。此谓断点被命中（hit）。如图2.11所示，一支小箭头表示CPU的程序计数器（program counter）目前位于哪行代码。

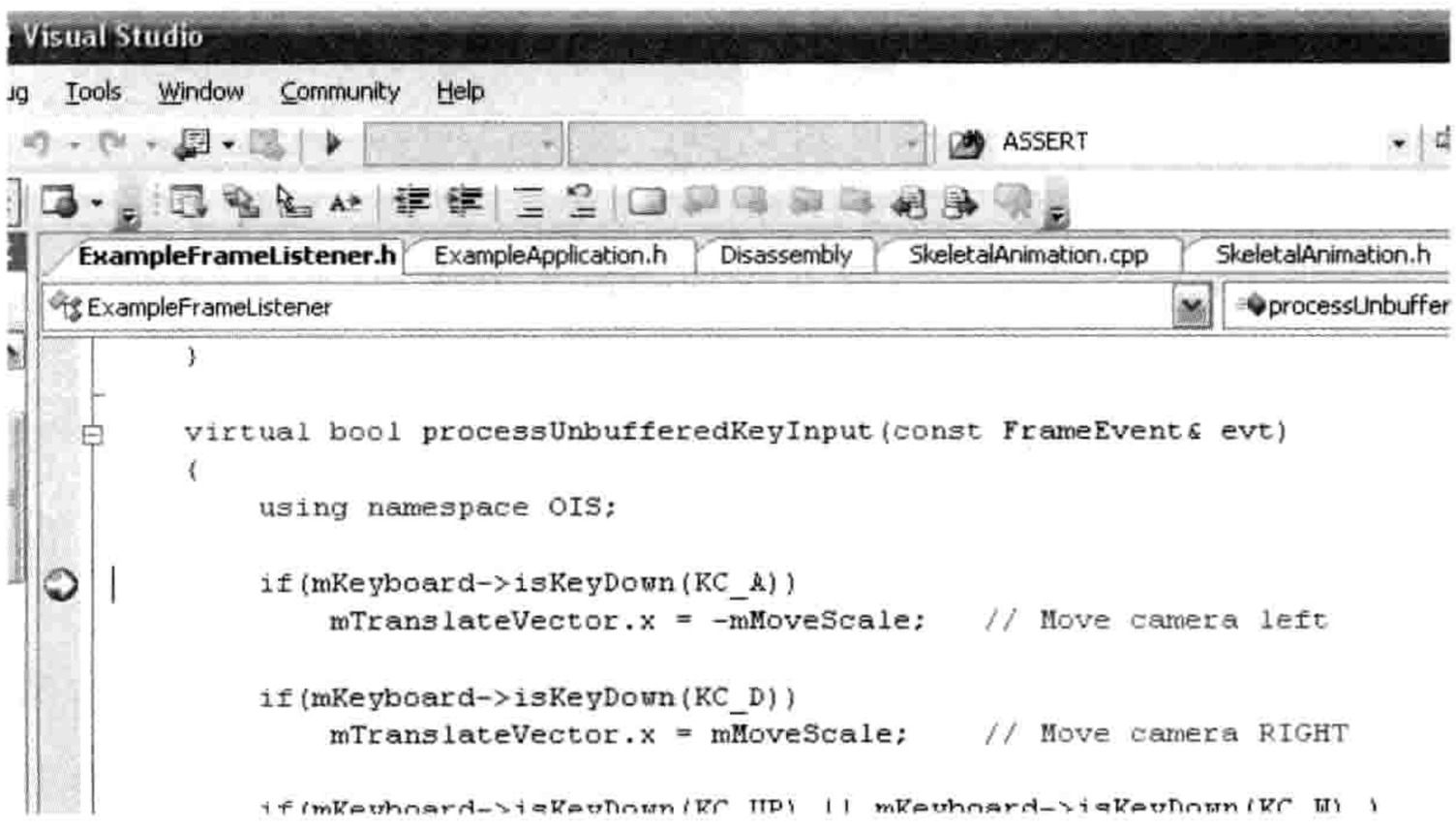


图 2.11: 在Visual Studio中设置断点。

[18]译注：原文没写明设定方法，在此补充。设定单个启动项目，可在解决方案资源管理器中，右击项目，在弹出菜单选择“设为启动项目（Set as Startup Project）”。此外，文中上一句并不准确，Visual Studio还提供调试多个项目，可在解决方案右击“Multiple Startup Projects（多启动项目）”。

2.2.5.3 单步执行代码

当断点被命中，可按F10键单步执行代码。黄色程序计数器会移动，显示程序执行的代码行。按F11键能逐语句**进入**（step into）函数调用（即下一行代码是被调用函数的首行），而按F10键则逐过程（step over）而**不进入**函数调用（即调试器以最全速调用函数，并在调用结束后再次停下来）。

2.2.5.4 调用堆栈

如图2.12所示的**调用堆栈**（call stack）窗口能显示任何时刻的函数调用堆栈。可于主菜单栏选择“Debug→Windows→Call Stack（调试→窗口→调用堆栈）”以显示调用堆栈窗口。

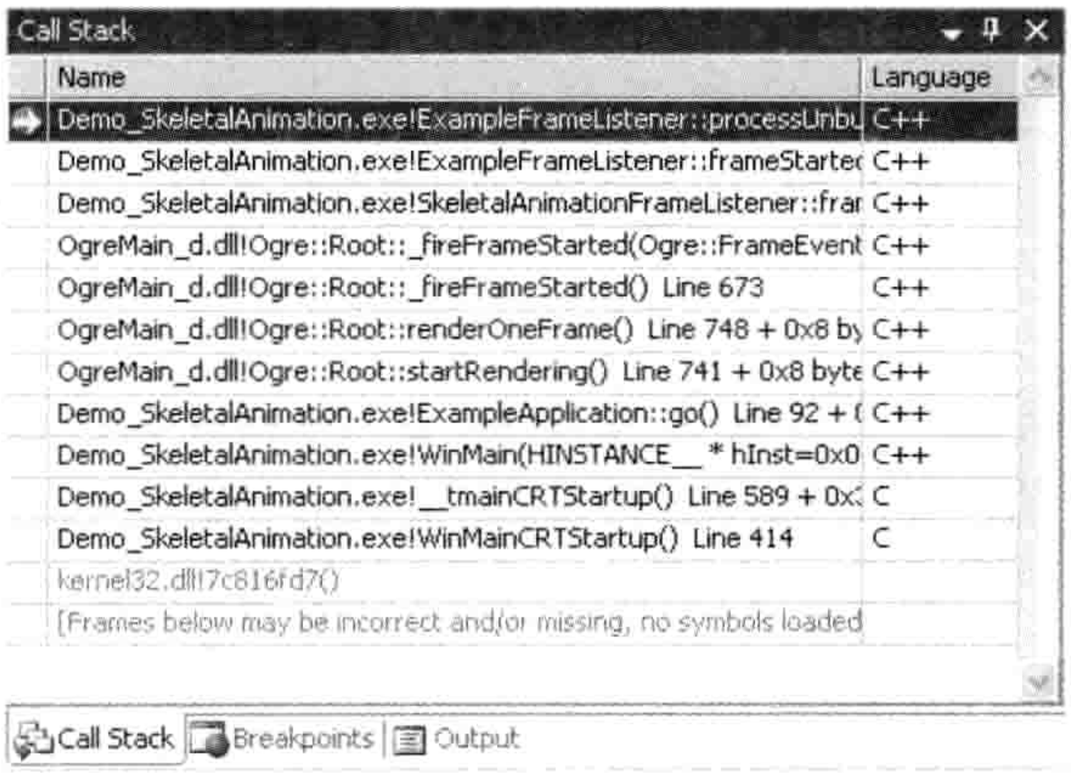


图 2.12: 调用堆栈窗口。

命中断点时（或手动把程序暂停），双击调用堆栈窗口的条目，就可在调用堆栈里上下移动。此操作非常有用，能检查从`main()`开始调用至目前代码行的一连串函数调用。例如，有时候在深入的嵌套函数调用中，此方法能往上追查父代函数，从而找出bug的源头。

2.2.5.5 监视窗口

当单步执行代码并在调用堆栈里上下移动时，程序员需要检查程序中变量的值。**监视窗口**（watch window）就是为此而设的。要打开监视窗口，可于主菜单栏中选择“Debug → Windows → Watch（调试→ 窗口→ 监视）”，最后选择监视1至监视4（Visual Studio容许同时开启4个监视窗口）。监视窗口开启以后，可在窗口键入变量的名字，或从源代码直接拖动表达式至窗口。

从图2.13可看到，简单数据类型的变量，其值会于变量名右方直接显示。复杂数据类型的变量，其值会以树视图显示，可展开节点往下观看到几乎所有嵌套结构。一个实例的基类总是显示为其派生类的首个子节点。因此，不但可以检查类的数据成员，而且也可以检查其基类（们）的数据成员。

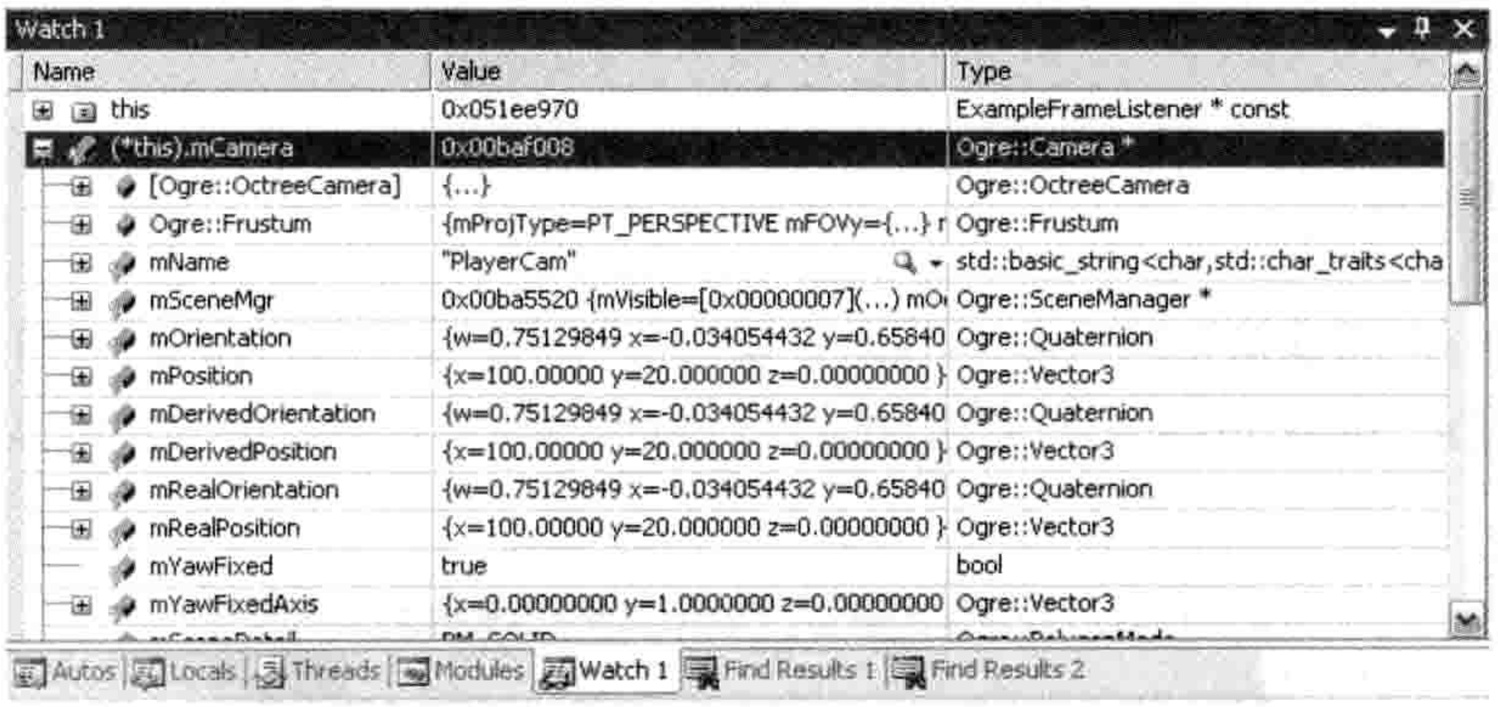


图 2.13: Visual Studio的监视窗口。

在监视窗口中可以输入几乎任何有效的C/C++**表达式**（expression），Visual Studio会对这些表达式取值，并尝试显示其结果。例如，输入“5+3”，Visual Studio会显示“8”。我们也可以使用C/C++的语法去转换变量的类型。例如，在监视窗口输入“`(float)myInteger Variable * 0.5f`”，会以浮点数显示`myIntegerVariable`的值除以2。[19]

监视窗口除了可显示变量的值，还可以**调用程序中的函数**。Visual Studio会自动重新为监视窗口内的表达式取值，因此，若在监视窗口内输入含函数的表达式，每次命中断点或单步执行代码都会调用那些函数。此技巧可帮助用户在调试器中，运用程序本身去诠释要检查的数据。例如，假设某游戏引擎提供一个名为`quatToAngleDeg()`的函数，可把四元数（quaternion）转换为旋转角度的度数。程序员便可以在监视窗口调用此函数，更容易在调试器中检查任何四元数的旋转角度。

此外，也可以使用几个后缀去改变Visual Studio显示数据的方式，如图2.14所示。

- “`,d`”后缀强制把值以十进制显示。
- “`,x`”后缀强制把值以十六进制显示。

[19]译注：此例子并不贴切，因为例子中即使不用显式转换（explicit cast），C/C++也会进行隐式转换（implicit cast）。译者举另一个例子，假设`a`和`b`都是整数类型变量，“`(float)a/b`”就可以显示两数之浮点数比。

- “,n”后缀（n为任意正整数）强制Visual Studio把该值视为一个有n个元素的数组。此后缀可以用来展开以指针参考的数组数据。

当在监视窗口中展开特大型的数据结构时，请务必小心，因为这么做有时候会使调试器速度变慢，甚至严重到不能使用的地步。

Watch 1

Name	Value	Type
mCamera->mSceneMgr->mLastFrameNumber,x	0x0000512c	unsigned long
mCamera->mSceneMgr->mLastFrameNumber,d	20780	unsigned long
mCamera->mCullFrustum->mFrustumPlanes,6	0x0000010c {normal={...	Ogre::Plane [6]
[0x0]	{normal={...} d=??? }	Ogre::Plane
[0x1]	{normal={...} d=??? }	Ogre::Plane
[0x2]	{normal={...} d=??? }	Ogre::Plane
[0x3]	{normal={...} d=??? }	Ogre::Plane
[0x4]	{normal={...} d=??? }	Ogre::Plane
[0x5]	{normal={...} d=??? }	Ogre::Plane

图 2.14: Visual Studio监视窗口中使用逗号后缀。

2.2.5.6 数据断点

常规的断点触发条件是CPU的程序计数器命中某个机器指令或代码行。然而，现在的调试器提供另一个极有用的功能，就是能够设立另一种断点，其触发条件是数据**写入**（即改变）某指定地址，所以这种断点称为**数据断点**（data breakpoint）。由于它是通过CPU的一个特殊功能而实现的，该功能可以在指定地址被写入时引发一个中断（interrupt），所以这种断点又称为硬件断点（hardware breakpoint）。

以下介绍数据断点的典型用法。例如，在追查一个bug时，发现某个对象的成员变量`m_angle`为零（`0.0f`），而此变量的值**应该**永不为零。程序员可能不知道哪个函数可能把零写入此变量，但是程序员知道该变量的地址（可从监视窗口键入“`&object.m_angle`”取得）。要找出肇事者，可在`object.m_angle`的地址中设立数据断点，之后让程序继续执行。当该变量的值被改动，调试器就会自动停下来。那时便可以检查调用堆栈，肇事的函数就能被捉个正着。

在Visual Studio设置数据断点，有以下几个步骤。

1. 在主菜单栏中选择“Debug→Windows→Breakpoints（调试→窗口→断点）”，打开“断点”窗口（图2.15）。
2. 在窗口的左上角按“New（新建）”下拉按钮。
3. 选择“新建数据断点（New Data Breakpoint）”。
4. 键入原始地址或结果为地址的表达式，例如“`&myVariable`”（图2.16）。

“Byte Count（字节计数）”字段一般填入4。因为32位奔腾CPU只能原生地检查4字节（32位）的值。如果设置其他数值，调试器需要做一些特殊处理，这有可能会令程序运行慢如蜗牛（假如能运行的话）。

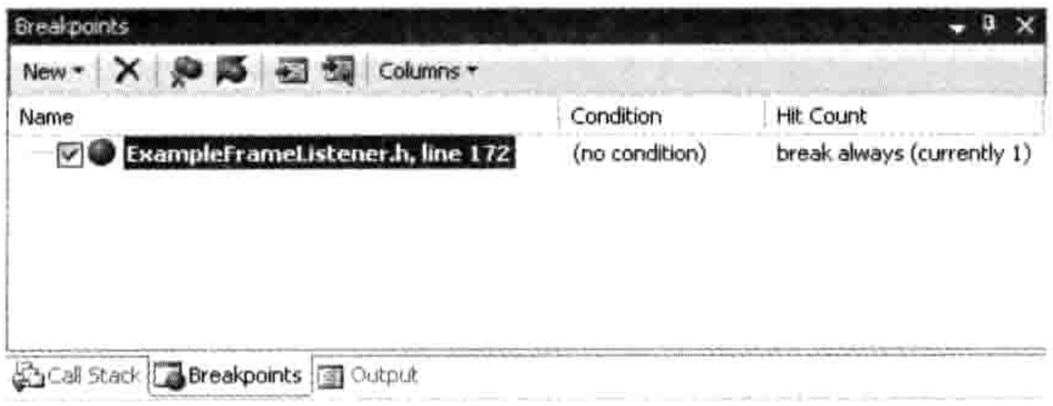


图 2.15: Visual Studio断点窗口。

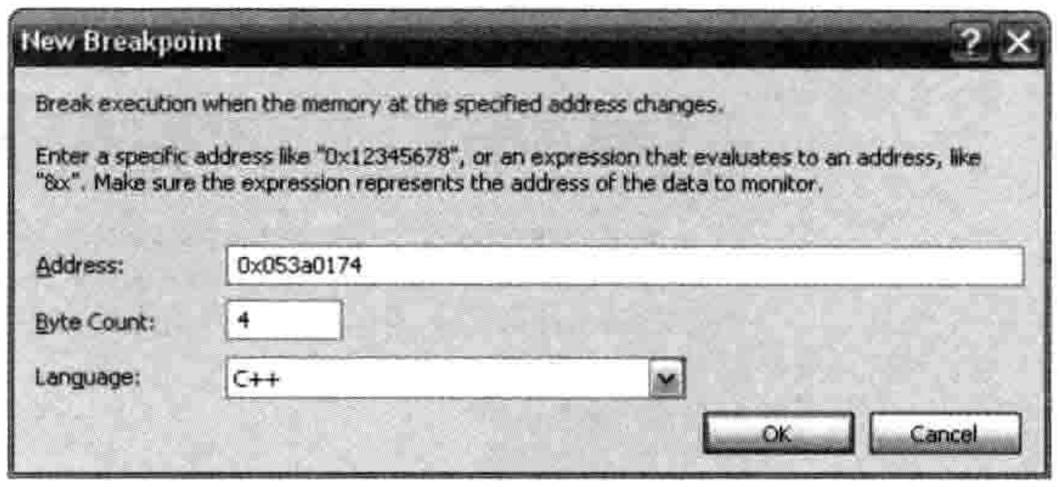


图 2.16: 设置数据断点。

2.2.5.7 条件断点

读者或许会发现，在断点窗口中，任何种类的断点（数据断点或常规的代码行断点）都可以设置条件（condition）和命中次数（hit count）。

条件断点（conditional breakpoint）促使调试器在每次断点命中时，对输入的C/C++表达式进行评估。若表达式为真（true），调试器会暂停程序，让用户检查程序运行情况。若表达式为假（false），调试器会忽略断点，继续运行程序。这个功能可以用来建立一种断点，当一个函数被某个实例调用时才触发。例如，假设一个游戏关卡中有20辆坦克，程序员希望第3辆坦克被调用时暂停程序。如果第3辆坦克的内存地址为`0x12345678`，便可以设置断点条件为“`(unsigned)this==0x12345678`”，去限制断点只命中第3辆坦克实体。

另外，设置**命中次数**之后，调试器每次命中时就会令该计数器减1，直至计数器变为零才触发暂停程序。这种设置对位于循环中的断点很有用。例如，要检查循环中第376个迭代的情况（例如数组第376个元素），不可能人工慢慢地去按F5键375次！但你**可以**设置命中次数，让Visual Studio自动完成任务。

有一点要小心，如果设定了条件断点，那么每次命中时调试器都要对表达式取值，因而会减慢调试器和游戏的运行速度。

2.2.5.8 调试已优化的生成

之前提及，使用发布版本调试问题是非常棘手的，主要是由于编译器优化代码的方式。理想情况下，程序员都宁愿选用调试生成去进行调试。然而，这经常不可行。有时候，一

个bug重现率很低，当bug出现时就要好好把握调试机会，甚至有时候某个bug只出现在别人机器的发布生成上。另外，一些bug只出现在发布生成，在调试生成中又会神奇地消失。这些**只出现在发布生成**的可怕bug，有时候是由未初始化变量造成的。因为在调试模式中变量和动态分配的内存通常会填入零，而在发布模式中这些内存没有初始化，内容为随机值。bug只出现在发布生成版本的其他常见原因，包括在发布生成中意外地略去一些代码（例如，一些重要代码被错误地放进断言语句），数据结构的大小或数据成员打包方式在调试和发布模式间有差异，内联或编译器引进的优化触发bug，（罕见情况下）编译器的优化器本身的bug，也会导致在发布生成中产生错误的代码。

显然，每位程序员都必须有调试发布生成的能力，即使这看上去并不是一件轻松的事情。减轻调试优化代码之痛，最佳办法是多练习，并且在有机会时扩展这方面的技能。以下是一些窍门。

- **学习在调试器中阅读及单步执行反汇编**（disassembly）：在发布生成中，调试器经常不能正确地显示目前正在执行的代码行。“感谢”指令乱序，在源代码检示模式中，经常会看见程序计数器不规律地在函数内游走。然而，若使用反汇编检示模式，则一切变得正常（即能逐条汇编语言指令执行）。每位C/C++程序员都应该稍稍了解目标平台的架构和汇编语言。如能这样，即使调试器受迷惑了，程序员也不会。
- **运用寄存器去推理变量的值或地址**：有时候，调试器不能在发布生成中正确显示变量的值或对象的内容。但是，如果程序计数器距离变量的初次使用不远，那么有很大机会该变量的值或地址仍然存于其中一个CPU寄存器里。若可以向前追踪反汇编，找到变量第一次载入寄存器的位置，就可以不断检查该寄存器以得知变量的值或地址。可使用寄存器窗口，或在监视窗口键入寄存器名字，检查寄存器内容。
- **使用地址去检查变量及对象内容**：知道变量或数据结构的地址，就可以在监视窗口中转换为适当的类别去检查其内容。例如，一个`Foo`类的实体位于`0x1378A0C0`，就可在监视窗口键入“`(Foo*)0x1378A0C0`”，那么调试器就会诠释该地址为指向一个`Foo`对象的指针。
- **利用静态和全局变量**：就算经过优化的生成版本，调试器通常也能够检查静态和全局变量。若不能推算出一个变量或对象的地址，则可以看看可能直接或间接地存有该地址的静态和全局变量。例如，若要找一个物理系统内部变量的地址，可能会发现它是存于`PhysicsObject`全局变量中的一个成员变量。
- **修改代码**：若想相对简单地重现一个只出现在发布生成版本中的bug，可考虑修改源代码以协助调试问题。增加打印语句去显示情况，引入全局变量使调试器里检查问题变量或对象更容易，加入代码以检测问题状态，加入代码去孤立一个类的某个实例。

2.3 剖析工具

游戏通常是高性能的实时系统。因此，游戏程序员经常要寻求加速代码的方法。有一个尽管不太科学但很有用的经验法则，称为**帕累托法则**（Pareto principle）[20]。此法则指出在很多情况下，一些事件的80%后果只取决于20%的原因，所以它又称为**80-20规则**（80-20 rule）。计算机科学常使用此法则的变种，称为**90-10规则**（90-10 rule），指任何程序的90%挂钟时间（wall clock time）[21]消耗在运行仅10%的代码上。换句话说，优化那10%的代码，带来的总体运行速度提升达完全优化的90%。

那么，如何得知需优化的**10%**代码**在哪里**？答案就是使用**剖析器**(profiler)[22]。剖析器能量度代码的执行时间，并能告之每个函数所花的时间。这些数据可引导程序员去优化占“狮子份额[23]”执行时间的函数。

一些剖析器也能告之每个函数的调用**次数**。这也是一个须了解的重要数据。某个函数耗用大量时间，原因有二：(a)运行函数本身需很长时间，(b)函数被频繁调用。例如，一个函数运行A*算法[24]搜寻游戏世界里的最优路径，每帧可能执行数次，而函数本身需要花显著的时间。另一方面，一个计算点积（dot product）的函数，运行可能只需几个时钟周期，但每帧此函数可能会调用数以十万次，从而拖慢游戏的帧率（frame rate）。

若使用适当的剖析器，则能进一步获取更多信息。一些剖析工具能报告调用图（call graph），可告之某函数被哪些函数调用（称为**父函数**/parent function），以及该函数调用了哪些函数（称为**子函数**/child function或**后代**/descendant）。甚至可以知道函数的运行时间花在后代的**百分比**，以及每个函数占整体运行时间的百分比。

剖析器大致可分为两类。

1. **统计式剖析器**（statistical profiler）：此类剖析器是不唐突的（unobtrusive），意指启动剖析器后，目标代码的执行速度差不多和没使用剖析器时相同。这些剖析器的原理是，周期性地为CPU的程序计数器寄存器采样，并以此获得正在执行的函数。由每个函数的采样数目，可计算出该函数占整体执行时间的近似百分比。对于运行于Pentium机器上的Windows平台，Intel的**VTune**软件是统计式剖析器中的不二之选，

---

[20] http://en.wikipedia.org/wiki/Pareto_principle

[21] 译注：挂钟时间指现实中的时间。

[22] 译注：profiler又译作探查器、(性能)分析器等。因profile有轮廓、外形之意，而“剖”指切开、分析，同时“剖面”一词也有分析物体轮廓之意，故选择剖析(profile)、剖析器(profiler)、剖析工具(profiling tool)的译法。

[23] 译注：《狮子的份额（Lion's share）》为伊索寓言：狮子和同伴共同捉到猎物，原本要分享成果，狮子却以强者身份宣告一切都归它所有。正文中指占大部分时间的函数。

[24] 译注：A*（读作A star）是一种常见搜寻算法，能在一个图（graph）的两点之间搜寻最优路径。A*的特点是利用启发函数（heuristic function）加速搜寻，同时能保证获得最优解。

现时也提供了Linux版本[25]。

2. **测控式剖析器**（instrumental profiler）：此类剖析器能提供最精确、最详尽的计时数据，但是却要以不能实时运行程序为代价——当启动剖析器后，目标程序慢如蜗牛。此类剖析器须预处理可执行文件，为其中每个函数安插特殊的初构代码（prologue code）和终解代码（epilogue code）。初构和终解代码会调用剖析器的库，调查程序的堆栈并记录所有细节，包括调用该函数的父函数、父函数调用子函数的次数。此类剖析器甚至可以设定监察每一行源代码，告之执行每行代码所花的时间。这些剖析结果极精准和详细，可是启动剖析器会令游戏慢得几乎无法玩。IBM的Rational Quantify软件[26]（Rational Purify Plus工具套装之一员）是个优秀的测控式剖析器。

微软也发行了混合这两种类型的剖析器，名为LOP，代表低开销剖析器（Low Overhead Profiler）。它使用统计方法，周期性地对处理器的状态采样，对程序速度的影响很少。但是，每个采样都会剖析调用堆栈，从而找出每个采样的一连串父函数。因此，LOP提供一般统计式剖析器无法提供的数据，例如被父函数调用的分布。

2.3.1 剖析器列表

坊间有许多优良的剖析器。维基提供一份相当详细的列表[27]。

2.4 内存泄漏和损坏检测

困扰C/C++程序员的另外两个问题是内存泄漏（memory leak）和内存损坏（memory corruption）。如果一块内存在分配后永不释放，就会产生内存泄漏。泄漏会浪费内存，最终造成致命性的内存不足（out of memory）。内存损坏则是指，程序不慎把数据写进内存的错误位置，覆盖了该位置原来的重要数据，也同时未能把数据写到**应该**写的位置。两个问题皆可毫不含糊地归咎于同一个语言特征——**指针**（pointer）。

指针是强大的工具，用得其所故然有益，但也很易化益为弊。若指针指向已释放的内存，或者指针被意外地赋值为非零整数或浮点数，那么这些指针就化为内存损坏的危险工具，因为数据最终会写入几乎任何地方。同样，若用指针来跟踪已分配的内存，也极容易在用完后忘记释放内存，导致内存泄漏。

避免因指针问题造成内存泄露的方法之一，是养成良好的编程习惯。这种习惯无疑可以

[25]译注：原文网址已失效，新网址为http://software.intel.com/en-us/intel-vtune-amplifier-xe。

[26]http://www.ibm.com/developerworks/rational/library/957.html

[27]http://en.wikipedia.org/wiki/List_of_performance_analysis_tool

确保编写理论上永不发生内存损坏和泄漏的可靠代码。当然，帮助检测内存泄漏和损坏的工具也必不可少。幸运的是，有许多这类现成工具。

笔者的首选是IBM Purify Plus工具套装中的Rational Purify[28]。Purify须在程序运行前安插测控代码，为所有指针解引用（dereference）及内存分配释放代码加入挂钩（hook）**钩子函数**（hook function）。在Purify下运行代码，能现场报告代码中的即时及潜在问题。程序结束后，Purify能产生详尽的内存泄漏报告。每个问题都直接链接至问题成因的源代码，使追踪和修正这些问题变得轻松。

另一流行工具是Compuware公司的Bounds Checker[29]，其用途和功能与Purify相似。

2.5 其他工具

游戏程序员的工具箱里还有许多其他工具。本文不会深入探讨这些工具，但以下的列表能让读者知道这些工具的存在，并可以按需学习。

- **区别工具**（difference/diff tool）：区别工具是用来比较一个文本文档的两个版本，找出版本之间的差异（关于区别工具的详细讨论可参见维基[30]）。区别工具通常以行为基准计算区别，但一些新的区别工具也能显示一行之中改动了的字符。多数版本控制系统都附带一个区别工具。有些程序员如对区别工具有偏好，可在版本控制系统里自行设置。流行的区别工具有ExamDiff[31]、AraxisMerge[32]、WinDiff（能在Windows的一些Option Pack及一些独立网站找到）、GNU区别工具包[33]。
- **三路合并工具**（three-way merge tool）：当两人修改同一个文件时，就会产生两组区别。能把两组区别合并成为含二人改动的最终文件的工具，称为三路合并工具。“三路”是指合并事实上使用了3个版本——原版本、用户A的版本、用户B的版本。（维基有关于二路和三路合并技术的讨论[34]）。很多合并工具附有连带的区别工具。流行的合并工具有AraxisMerge、WinMerge[35]。Perforce也附有杰出的三路合并工具[36]。
- **十六进制编辑器**（hex editor）：十六进制编辑器用于查看及修改二进制文件的内容。

---

[28] http://www-306.ibm.com/software/awdtools/purify

[29] http://www.compuware.com/products/devpartner/visualc.htm

[30] http://en.wikipedia.org/wiki/Diff

[31] http://www.prestosoft.com/edp_examdiff.asp

[32] http://www.araxis.com

[33] http://www.gnu.org/software/diffutils/diffutils.html

[34] http://en.wikipedia.org/wiki/3-way_merge#Three-way_merge

[35] http://winmerge.org

[36] http://www.perforce.com/perforce/products/merge.html

数据通常以十六进制整数显示，因而得名。大部分好用的十六进制编辑器，能把数据以整数（1～16字节）、浮点数（32位或64位）、文本（ASCII）方式显示。十六进制编辑器对追查二进制文件格式问题，或对未知的二进制格式文件做反向工程，特别有用。而这两种工作在游戏引擎开发圈里比较常见。坊间有数之不尽的十六进制编辑器，笔者使用Expert Commercial Software公司的HexEdit[37]，读者的选择也许不同。

毫无疑问，游戏引擎开发者需要更多的工具，使工作变得更轻松。希望本章介绍的主要工具足以满足读者日常工作之用。

[37] http://www.hexedit.com/

第3章　游戏软件工程基础

本章扼要地重温面向对象编程的基础概念，继而探索一些进阶课题。对于任何软件工程（尤其是游戏开发），这些课题皆是极有用的。比如第2章，笔者希望读者不会完全略过。在旅程开始前，配备充足的工具及储备，至关重要。

3.1　重温C++及最佳实践

3.1.1　扼要重温面向对象编程

本书大部分的内容，都会假设读者对面向对象设计原理有深刻的了解。若读者对这些内容感到有点生疏，本节可当作一次轻松快捷的复习。但是，若读者对本章内容不知所云，在继续阅读本书余下章节之前，笔者建议先选读一两本关于面向对象编程的书籍（如[5][1]）及C++的专门书籍（如[39]及[31]）。

3.1.1.1　类和对象

类（class）是属性（数据）和行为（代码）的集合，共同组成既有用又有意义的整体。类可视为**规格**（specification），这些规格描述类的个别**实例**（instance）——又称为**对象**（object）——的构造方法。例如，一只叫阿旺的狗是“dog”类的一个实例。因此，类和其实例之间存有一对多的关系。

[1]译注：关于面向对象编程，译者推荐《冒号课堂：编程范式与OOP思想》。

3.1.1.2 封装

封装（encapsulation）是指，对象向外只提供有限接口，隐藏对象的内部状态和实现细节。封装简化了类的使用方法，因为用户只需理解类的有限接口，而非类的内部实现细节，后者可能错综复杂。同时，程序员在编写类时，也可以通过封装使类的实体总是保持逻辑上的一致。

3.1.1.3 继承

继承（inheritance）能借着**延伸**现有的类去定义新的类。新类可修改或延伸现有类的数据、接口和行为。若一个名为Child的类延伸名为Parent的类，可以说Child**继承自**或**派生自**Parent。在此关系中，Parent称为**基类**（base class）或**超类**（superclass），而Child则称为**派生类**（derived class）或**子类**（subclass）。显然，继承会产生类的层次（树结构）关系。

继承使类之间产生“是一个（is-a）”关系。例如，圆形**是一个**图形种类。若要编写二维绘图应用软件，从基类Shape派生出Circle类，是很合乎情理的做法。

我们可采用统一建模语言（Unified Modeling Language，UML）定义的表示法，描述类的层次结构图。UML表示法里，长方形代表类，空心三角形箭头代表继承。继承箭头由子类指向父类。图3.1是一个UML**类图**[2]，表示一个简单的类层次结构。

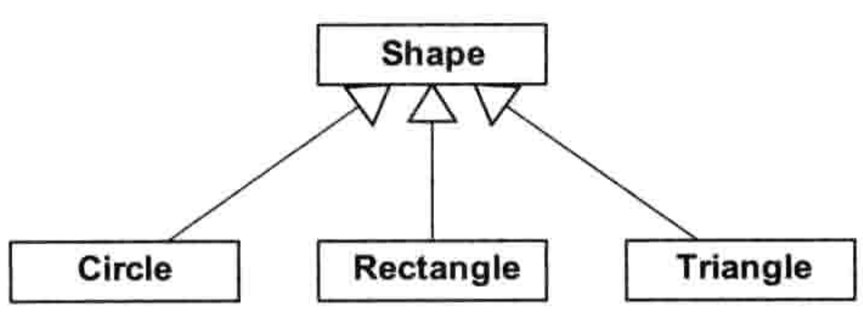


图 3.1: 一个简单的类层次结构的UML类图。

多重继承

一些编程语言支持**多重继承**（multiple inheritance，MI）。多重继承是指一个类有一个以上的父类。理论上，多重继承是颇优雅的，但在实际应用中，这种设计通常会产生很多混淆和技术困难[3]。这是由于多重继承把由类组成的简单的**树**（tree）变成可能很复杂的**图**（graph）。由类组成的图会形成很多问题，而这些问题不会在简单的树上发生，例如致命的

[2]译注：原文为UML static class diagram，但UML类图是用于表示静态结构的，一般不会加入“静态”二字，故略去。

[3]http://en.wikipedia.org/wiki/Multiple_inheritance

菱形继承问题（diamond problem）[4]。在菱形继承问题中，一个派生类最终包含了两份祖父类（见图3.2）。C++可以使用**虚继承**（virtual inheritance）去掉重复祖父类的数据。

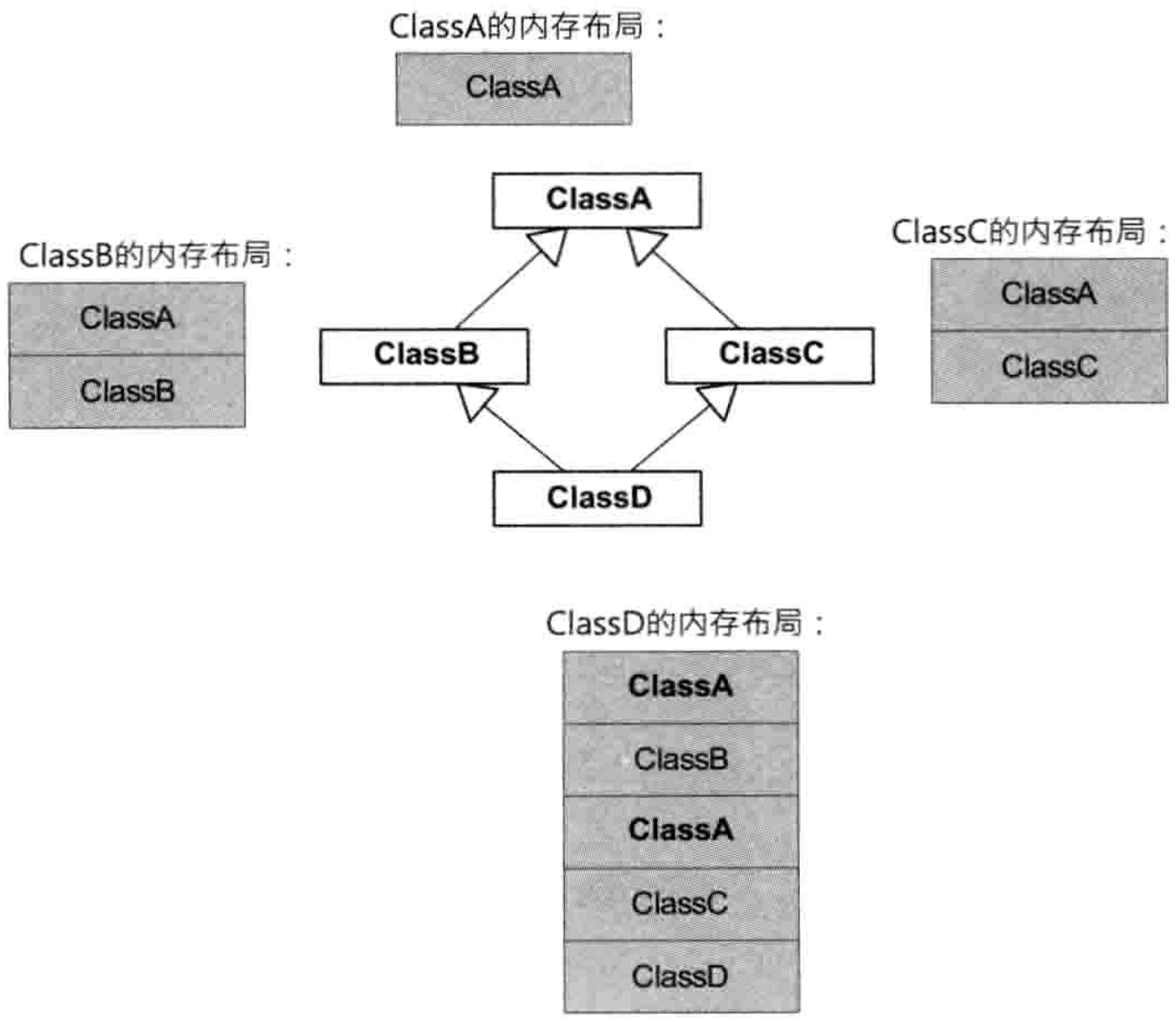


图 3.2: 多重继承中的“致命菱形”。

大多数C++软件开发者都会完全避免使用多重继承，或只容许有限制地使用。常见的惯例是，只容许从一个单继承层次结构中多重继承一些简单且无父的类。这些类有时称为**嵌入类**（mix-in class），因为它可在类树中任何位置加入新功能。图3.3是一个嵌入类例子。

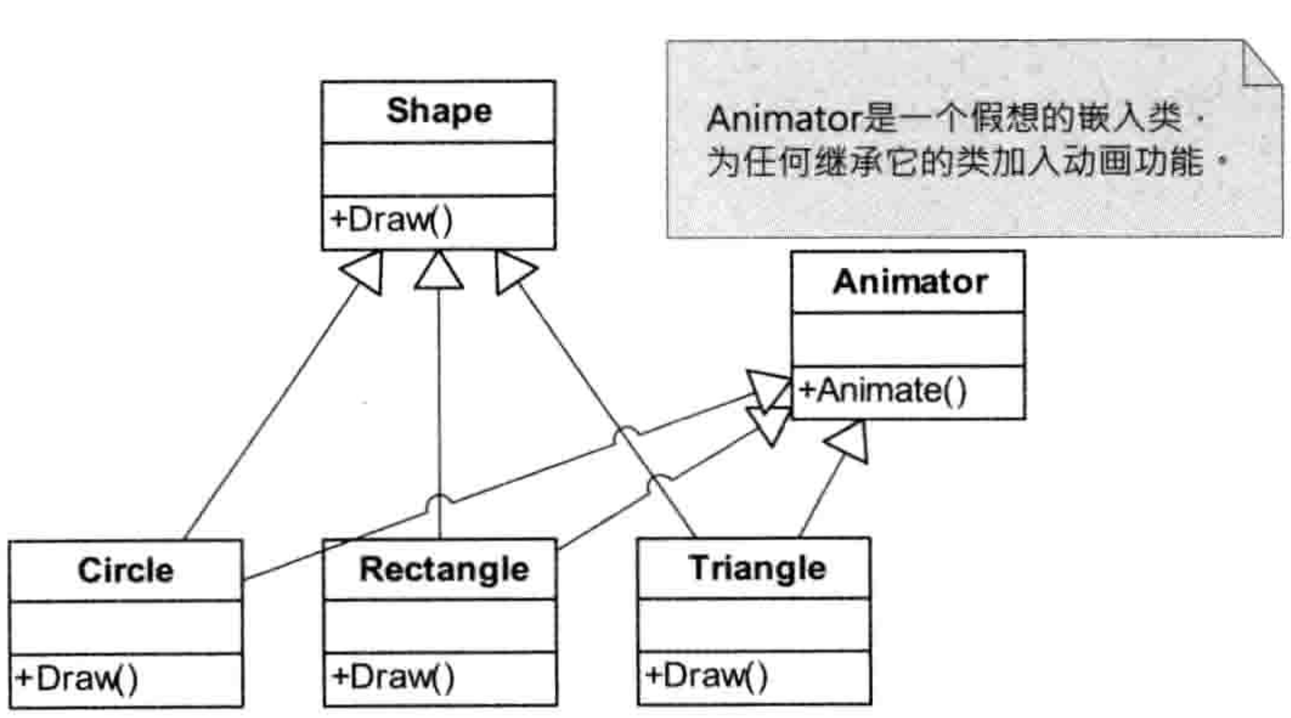


图 3.3: 一个嵌入类例子。

[4] http://en.wikipedia.org/wiki/Diamond_problem

3.1.1.4 多态

多态（polymorphism）是一种语言特征，容许采用单一**共同接口**操作一组不同类型的对象。共同接口能使异质的（heterogeneous）对象集合从使用接口的代码来看**显得**是同质的（homogeneous）。

例如，一个二维绘图程序要把一个形状列表绘于屏幕上，列表里有不同的形状。绘出这个异质形状集合的其中一个方法是，按不同形状类型，用switch语句区分并执行不同的绘画指令。

```
void drawShapes(std::list<Shape*> shapes)
{
    std::list<Shape*>::iterator shapeItr = shapes.begin();
    std::list<Shape*>::iterator shapeEnd = shapes.end();

    for (; shapeItr != shapeEnd; ++shapeItr)
    {
        switch ((*shapeItr)->mType)
        {
        case CIRCLE:
            // 绘画圆形
            break;

        case RECTANGLE:
            // 绘画矩形
            break;

        case TRIANGLE:
            // 绘画三角形
            break;
        // ......
        }
    }
}
```

这种方式的问题是，drawShapes()函数需要“知悉”所有可以绘画的形状类型。在简单例子里还好，但随着代码量的增加和代码复杂度的提高，在系统里新增形状类型变得越来越困难。每当加入一个新形状类型，必须搜索所有“知悉”各种形状类型的代码，如在例子里的switch语句中添加新的case去处理该新类型。

解决方法是把类型的内容从大部分代码中隔离出来。为了实施这个隔离，可以把每种要支持的形状定义为类。而所有这些类，都继承自一个共同的基类Shape。在基类Shape中可

以定义名为Draw()的**虚函数**（virtual function）（C++语言的主要多态机制），并在每个不同形状的类中，以不同方式实现这个函数。绘画时不需“知悉”给予的是何种形状，只需逐一简单地调用形状对象的Draw()函数便可。

```
struct Shape
{
    virtual void Draw() = 0; // 纯虚函数
};

struct Circle : public Shape
{
    virtual void Draw()
    {
        // 绘画圆形
    }
};

struct Rectangle : public Shape
{
    virtual void Draw()
    {
        // 绘画矩形
    }
};

struct Triangle : public Shape
{
    virtual void Draw()
    {
        // 绘画三角形
    }
};

void drawShapes(std::list<Shape*> shapes)
{
    std::list<Shape*>::iterator shapeItr = shapes.begin();
    std::list<Shape*>::iterator shapeEnd = shapes.end();

    for (; shapeItr != shapeEnd; ++shapeItr)
    {
        (*shapeItr)->Draw();
    }
}
```

3.1.1.5 合成及聚合

合成（composition）是指，使用**一组互动的**对象去完成高阶任务。合成在类之间建立“有一个（has-a）”和“用一个（uses-a）”的关系。（从技术上说，“有一个”的关系称为**合成**，“用一个”的关系称为**聚合** / aggregation）。例如，一艘太空船**有一台**引擎，引擎又**有一个**燃料缸。使用合成 / 聚合常常使各个类变得更简单、更专注。缺乏面向对象经验的程序员常会过分依赖继承，而忽视合成及聚合。

例如，如果要设计一个图形用户界面（graphical user interface，GUI）用作游戏的前端。假设定义了`Window`类表示任何长方形的GUI元素。另外也定义了`Rectangle`类封装数学上长方形的概念。缺乏经验的程序员可能会从`Rectangle`类派生`Window`类（使用`Window`“是一个”`Rectangle`关系）。但一个更具弹性及封装更好的方法是，`Window`类引用或包含一个`Rectangle`类（使用“用一个”或“有一个”的关系）。这样，每个类变得更简单、更专注，也更容易被测试、调试、重用。

3.1.1.6 设计模式

当同一类型的问题反复出现，而不同的程序员们却采用相似的方案去解决这些问题时，就可以说，该问题引发了一个**设计模式**（design pattern）。在面向对象编程中，已经有很多常见的设计模式获得识别及描述。当中最知名的，是“四人组（Gang of Four，GoF）”著作内的23个设计模式[17]。以下是几个常见的通用设计模式。

- **单例**（singleton）：此模式确保某个类只有一个实例（那个就是**单例实例** / singleton instance），并提供这个单例的全局存取方法。
- **迭代器**（iterator）：迭代器提供高效存取一个集合的方法，同时不需要暴露该集合之下的实现。
- **抽象工厂**（abstract factory）：抽象工厂提供一个接口，创造一组相关或互相依赖的类，而不需要指明那些类的具体类（concrete class）[5]。

游戏工业有自己一套设计模式，以对付渲染、碰撞、动画、音频等各领域的问题。从某种意义上来说，本书所有内容都是关于现在三维游戏引擎设计中流行的高阶设计模式。

---

[5]译注：具体类（concrete class）是指可以产生对象的类。相反，抽象类（abstract class）含未定义的虚函数而不能产生对象。

3.1.2 编码标准：为什么及需要多少

工程师之间讨论编码约定（coding convention）时，经常能引致热烈的“宗教”辩论。笔者不希望在此引发那种辩论，但我提议，应该最少要遵循以下这些最低限度编码标准（coding standard）。编码标准之所以存在，有两个主因。

1. 一些标准使代码更易读、更易理解、更易维护。
2. 另一些约定能预防程序员做蠢事，自找麻烦。例如，某编码标准可能会怂恿程序员只使用编程语言中更易测试、更不易出错的一小部分功能。由于C++语言充满滥用的可能性，所以这类编码标准对使用C++来说特别重要。

笔者认为，编码约定中最需要达到的事情如下。

- **接口为王：**保持接口（.h文件）整洁、简单、极小、易于理解，并有良好注释。
- **好名字促进理解及避免混淆：**持续使用能直接反映类、函数、变量用途的直观名字。花些时间去找合适的名字。如果有种命名方法，需要程序员查表才能理解代码的意义，就要避免使用这种命名方法。谨记，像C++这样的高级编程语言是为了供人阅读而设计的（若读者不同意，就问一问自己为何不直接用机器码来编写你的全部软件）。
- **不要给命名空间添乱：**使用C++命名空间或统一的名字前缀，以确保自己的符号（symbol）不会和其他库的符号冲突。（但慎防过度使用命名空间或嵌套过深。）为宏命名时要更小心，因为C++预处理器的宏只是文本替换，所以宏会跨越全部C/C++作用域及命名空间范围。
- **遵从最好的C++实践：**一些书籍提供了卓越的指导方针，使程序员避开麻烦，例如，Scott Meyers的《Effective C++》系列[31]、[32]、《Effective STL》[33]，以及John Lakos的《Large-Scale C++ Software Design》[27]。
- **始终如一：**笔者会用以下的规则。若从零开始写代码，可以自由地创造你的编码约定，然后坚持遵守约定。当编辑一些已有的代码时，无论那里有什么约定，都请尝试遵从。
- **显露错误：**Joel Spolsky写了一篇关于编码约定的出色文章《让错误代码显得错误（Making Wrong Code Look Wrong）》[6]。文中提出，所谓最“整洁”的代码，并不需要是表面看来简洁整齐的代码，而更重要的是，代码的编写方法能**容易显露**常见的编程错误。Joel的文章有趣且富有教育意义，笔者极力推荐此文。

[6] http://www.joelonsoftware.com/articles/Wrong.html

3.2 C/C++的数据、代码及内存

3.2.1 数值表达形式

在游戏引擎开发中（或在所有软件工程中），我们做的任何事情都是和数值密不可分的。每位软件工程师都应该了解，数值在计算机里是如何表达及储存的。本节将讨论这方面的基础知识，而这些知识是阅读本书余下章节所必需的。

3.2.1.1 数值底数

人们最自然地会使用**底数10**（base ten）的方式思考，这称之为**十进制**（decimal）记法。在这种记法里，使用10个不同的数字（digit）（0～9），每个数字由右到左代表下一个10的最高幂。例如，数值$7803 = (7 \times 10^3) + (8 \times 10^2) + (0 \times 10^1) + (3 \times 10^0)$。

在计算机科学里，数学上的量，如整数、实数，需要储存在计算机内存里。我们都知道，计算机以**二进制**（binary）格式储存数值，换句话说，即是只使用0和1两个数字。这又称为**底数2**（base two）记法，因为每个数字由右至左代表下一个2的最高幂。计算机科学家有时候使用“0b”前缀去表示二进制数值。例如，二进制数值0b1101等同于十进制13，因为$0b1101=(1 \times 2^3) + (1 \times 2^2) + (0 \times 2^1) + (1 \times 2^0)$。

在计算机领域里，另一个常见记法是**十六进制**（hexadecimal），或称为底数16。这种记法中，使用数字0～9和英文字母A～F，其中A～F分别代表10～15。C/C++编程语言使用“0x”作为十六进制的前缀。十六进制之所以流行，是由于计算机分组存储数据，每8位一组，又称为**字节**（byte），而一个十六进制数刚好是4位，所以**两个**十六进制数字恰好能代表一个字节。例如，0xFF=0b11111111=255是8位（1字节）能储存的最大数值。十六进制数的每个数字，由右至左代表下一个16的最高幂。因此，例如$0xB052=(11 \times 16^3) + (0 \times 16^2) + (5 \times 16^1) + (2 \times 16^0) = 45,138$。

3.2.1.2 有符号及无符号整数

在计算机科学中，我们同时使用有符号整数（signed integer）及无符号整数（unsigned integer）。其实，“无符号整数”有点用词不当。数学上，**自然数**（natural number）[7]的范围是由0（或1）至正无穷，而**整数**的范围则是负无穷至正无穷。虽然如此，本书还是采用计算机术语，统一用“有符号整数”和“无符号整数”。

[7]译注：原文此处还提及**完整数**（whole number），基本上和自然数同义，但因这一术语比较有歧义，所以忽略。

多数个人计算机和游戏主机能轻易处理32位或64位整数（虽然在游戏编程中也经常用到8位及16位的整数）。要表示一个32位无符号整数，只须把数值简单地编码为二进制记法（见上节）。32位无符号整数的可表示数值范围是从0x00000000（0）至0xFFFFFFFF（4294967295）。

要用32位表示**有符号**整数，便需要一个方法去分辨正值和负值。最简单的方法之一就是，把最高有效位（most significant bit，MSB）当作**符号位**（sign bit）——符号位为0代表数值为正，1代表数值为负。余下的31位则是数值的模（magnitude），实际上即是把模的范围缩小至一半（但这样能表示每个模的正负两个版本，包括正负零）。[8]

大多数微处理器采用更高效的技巧为负整数编码，此技巧称为**二补数**（two's complement）记法。对于数字零，二补数有唯一的表示方式，而简单使用符号位则会造成两个零的表示方式（正零及负零）。在32位二补数记法里，0xFFFFFFFF值代表−1，其他负值就从这个值倒数。任何最高有效位为1的值都代表负值。所以，从0x00000000（0）至0x7FFFFFFF（2147483647）的值代表正整数，从0x80000000（−2147483648）至0xFFFFFFFF（−1）代表负数。[9]

3.2.1.3 定点记法

除了用整数记法表示整数，若要表示分数和无理数，就需要有不同的格式来表达小数点的概念。

计算机科学家曾采用的方法之一是**定点**（fixed-point）记法。定点记法可随意选择整数部分及小数部分各用多少位去表示。如果从左到右（即从最高有效位至最低有效位）观察一个定点记法的值，整数部分的位表示2的递减幂（$\cdots$，16，8，4，2，1），而小数部分的位表示2的递减倒数幂（$\frac{1}{2}$，$\frac{1}{4}$，$\frac{1}{8}$，$\frac{1}{16}$，$\cdots$）。例如，要把−173.25用32位定点记法储存，32位中1位是符号位，16位为整数，15位为小数。首先，把符号部分、整数部分、小数部分分别转为二进制（负数 $=$ 0b1、$173 =$ 0b0000000010101101、$0.25 = 1/4 =$ 0b0100000000000000）。之后就可以把这些值打包成一个32位整数。如图3.4所示，最终结果为0x8056A000。

定点记法的缺点在于，它同时限制了可表示整数部分的范围及小数部分的精度。例如一个32位定点小数，当中有16位整数、15位小数、1位符号。此格式除去小数部分，其表示范围只是$\pm 65,535$，并不一定足够。要解决此问题，可使用**浮点**记法。

---

[8]译注：此处描述的方法称为原码法（sign-and-magnitude method）。

[9]译注：原文没提及二补数在哪方面比原码法高效。事实上，补码在加减法中，也可以当作无符号数值处理。详情可参考http://en.wikipedia.org/wiki/Two's_complement。

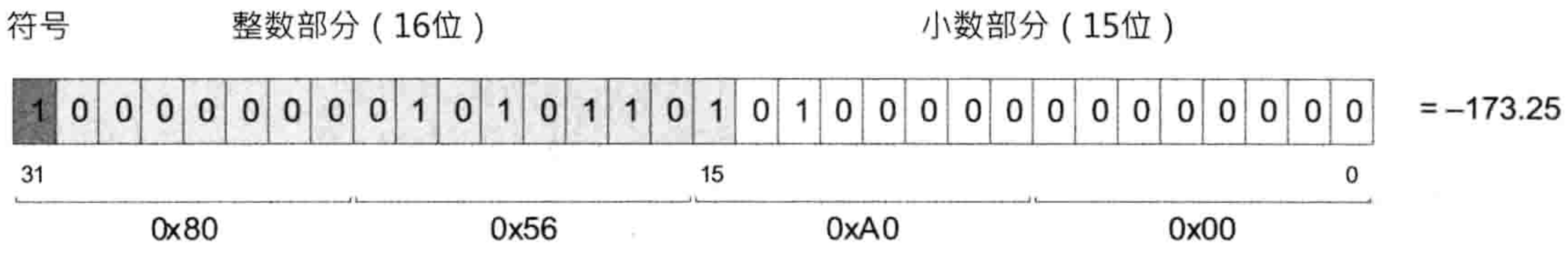


图 3.4: 16位整数、15位小数的定点记法。

3.2.1.4 浮点记法

在浮点（floating-point）记法里，小数点可以任意移动至不同位置，此位置仍是由指数（exponent）控制的。一个浮点数由3部分组成：**尾数**（mantissa）含有包括小数点前后的相关数字，**指数**（exponent）决定那串数字的小数点位于哪里，而**符号位**理所当然就是显示该值为正数或负数。虽然有不同的方式去安排这3部分在内存中的格式，但最流行的标准是IEEE-754。IEEE-754标准中定义的32位浮点数，其最高有效位是符号位，紧随的是8位指数和23位尾数。

若使用符号位s、尾数m、指数e去表达一个值v，则$v = s \times 2^{(e-127)} \times (1 + m)$。

符号位s的值为+1或−1。指数e在储存时加上偏移量127，使e能轻松表示负数。尾数的第1位是隐含的1，这个隐含位不储存于内存中，而之后的位则代表2的倒数幂。所以，若尾数位储存小数部分m，其实是表达$1 + m$。例如，图3.5的位模式代表了0.15625。当中$s = 0$（代表正数）、e=0b0111100=124、m=0b01000...$= 0 \times 2^{-1} + 1 \times 2^{-2} = \frac{1}{4}$，因此：

$$
\begin{aligned}
v &= s \times 2^{(e-127)} \times (1 + m) \\
&= (+1) \times 2^{(124-127)} \times (1 + \frac{1}{4}) \\
&= 2^{-3} \times \frac{5}{4} \\
&= \frac{1}{8} \times \frac{5}{4} \\
&= 0.125 \times 1.25 = 0.15625
\end{aligned}
$$

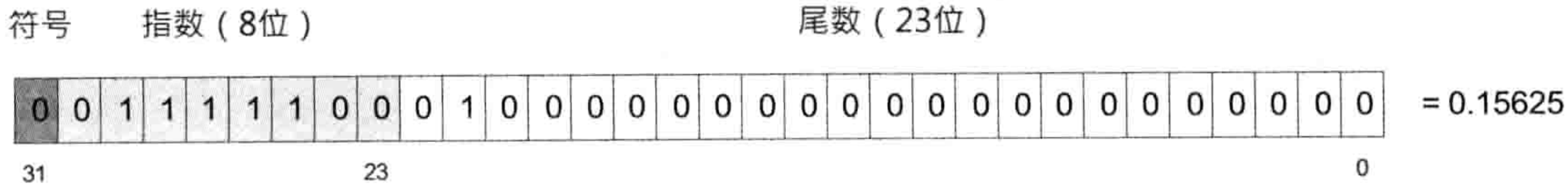


图 3.5: IEEE-754 32位浮点格式。

范围和精度的取舍

浮点数的**精度**增加，可表示**范围**则缩小，反之亦然。这是因为，若使用固定数目的位去表示浮点数，尾数和指数所占的位数此消彼长。尾数位越多，精度越高；指数位越多，可表示范围越大。物理中常使用**有效数字**（significant figure）去描述此概念[10]。

为了理解范围和精度的取舍，可分析一下IEEE 32位浮点数的最大值FLT_MAX$\approx 3.403\times 10^{38}$，其表示为0x7F7FFFFF。我们来解读此表示：

- 23位尾数的最大值为十六制0x00FFFFFF，即二进制中连续24个1。此24个1为23位的尾数和隐含的首个1。
- 在IEEE-754格式中，把指数设为255有特别意义，用来代表数值是NaN或无穷大，正常的指数并不能使用255。因此最大的8位指数实际是254，减去偏移量127之后，表示指数为127。

因此，FLT_MAX为$0\text{x}00\text{FFFFFF}\times 2^{127} = 0\text{xFFFFFF}0000000000000000000000000$。换句话说，二进制中24个1向左移了127个位，在尾数的最低有效位后剩下$127-23=104$个二进制0（或$104/4=26$个十六进制0）。这些尾随的0并不对应32位浮点数里任何实际位，只是通过指数无中生有得来。如果从FLT_MAX减去一个很小的数（所谓小，指任何小于26个位的十六进制数），其结果仍然为FLT_MAX，因该26个十六进制的位从未存在于浮点记法中。

当尾数远远地小于1（即非常接近0），浮点数也会带来相反的效果。这种情况下，指数为很大的负值，有效位就会移至小数点右方。我们用更大的尾数表示范围，去换取高精确度。总而言之，浮点数有不变数目的**有效数字**（更准确地说是**有效位**），而指数则用来把有效位推移到较高或较低的尾数范围。

另一个要注意的细微之处是，每种浮点记法的零和最小非零值都存在一个有限的间隙。IEEE-754 标准中的32位浮点数能表示的最小非零值为FLT_MIN$= 2^{-126}\approx 1.175\times 10^{-38}$，其二进制表示法为0x00800000（即指数为0x01，减去偏移量后代表-126，尾数除了顶头的隐含1外全为0）。无法表示比1.175×10^{-38}小的非零值，因为下一个能表示的最小值就是零。换言之，用浮点表示的实数轴是被**量化的**（quantized）。[11]

对于某浮点数表示方式，满足方程$1+\varepsilon\neq 1$的最小的浮点值ε称为**机器的epsilon**。例如，IEEE-754标准中32位浮点数的精度为23位，$\varepsilon=2^{-23}\approx 1.192\times 10^{-7}$。$\varepsilon$的最高有效位**刚好进入了**1.0的有效数字范围，因此1.0加上任何小于ε的值并无效果。换句话说，小于ε的任何值和1相加时，23位的尾数并不能包含加上去的位，23位之后的位都被“截除”了。

---

[10] http://en.wikipedia.org/wiki/Significant_figures

[11] 译注：浮点小数只能表示实数线上一些点。一个32位数据只能表示2^{32}个不同的值，不能表示连续区间内无穷多个值。

有限精度和机器epsilon的概念对游戏软件有实质影响。例如，假设我们用浮点数去表示游戏从开始至今经过了多少秒，并称之为游戏绝对时间，那么，游戏要运行多久，才会导致加上1/30秒后，游戏绝对时间却维持不变？答案是大约12.9日。多数游戏不需要运行这么久，所以使用32位浮点表示以秒计算的游戏绝对时间还是可行的。然而，此例清楚显示，我们必须了解浮点格式的限制，从而能预知潜在的问题，并按需采取措施加以防范。

IEEE浮点数的位操作技巧

[7]里的2.1章介绍了一些非常有用的IEEE浮点数的位操作技巧，能让某些浮点数运算快如闪电。

3.2.1.5 基本数据类型

读者都应该知道C/C++提供了多个基本数据类型（atomic data type）。C/C++标准提出这些数据类型的大小、有符号/无符号的指导方针，但是编译器能自由定义这些类型。每个编译器的定义稍有差异，目的是使目标硬件达到最高效能。

- `char`：`char`通常是8位的，足够储存一个ASCII或UTF-8字符（见5.4.4.1节）。有些编译器定义`char`为带符号的，但其他编译器则预设`char`为无符号的。
- `int`、`short`、`long`：`int`是有符号整数值，而其大小恰好是目标平台上最高效的运算单位。在奔腾（Pentium）系列计算机中，`int`通常定义为32位。`short`本意是比`int`小的类型，`short`在许多机器上为16位。而`long`则等于或大于`int`，`long`在一些平台上是32位，一些则是64位。
- `float`：在大部分现代编译器里，`float`是IEEE-754的32位浮点数。
- `double`：`double`是IEEE-754的双精度（即64位）浮点数。
- `bool`：`bool`保存真/假值。`bool`在不同编译器及硬件架构上会采用截然不同的大小。`bool`从不会实现为1位，有些编译器定义`bool`为8位，也有些定义为32位。

编译器专属特定大小类型

标准C/C++的基本数据类型特意设计为可移植的，因而不做明确规定。然而，很多软件工程范畴，包括游戏引擎编程，有时候必须知道某些变量的确切尺寸。Visual Studio的C/C++编译器定义了以下的扩展关键字去声明特定位数的变量：`__int8`、`__int16`、`__int32`、`__int64`。

SIMD类型

许多电脑和游戏主机的CPU皆有特殊的算术逻辑单元（arithmetic logic unit，ALU），称为**矢量处理器**（vector processor）或**矢量单元**（vector unit）。矢量处理器提供一种并行处理方式，名为**单指令多数据**（single instruction，multiple data，SIMD）。单个SIMD指令可以并行地对多个数据进行运算。数据由矢量处理器处理，需先把数据以两个或更多个数值打包，存进64位或128位CPU寄存器。在游戏编程中，最常用的SIMD寄存器格式，是把4个32位IEEE-754浮点数值打包，存进128位SIMD寄存器。此格式使一些计算，如矢量点积和矩阵乘数，比单指令单数据（single instruction single data，SISD）算术逻辑单元更加高效。[12]

每个微处理器的SIMD指令集名字各有不同，而且编译器对不同目标微处理器有特定的SIMD变量声明语法。例如，奔腾系列CPU的指令集称为**单指令多数据流扩展**（streaming SIMD extensions，SSE），而微软Visual Studio编译器则提供内建数据类型`__m128`，表示4个`float`的SIMD数值。在PS3和Xbox 360的PowerPC系列CPU中，其SIMD的指令集称为**Altivec**，而GNU C++编译器采用`vector float`语法去声明打包4个`float`的SIMD变量。在4.7节会更详尽地讨论SIMD编程。

可移植的特定大小类型

多数其他编译器也有其自定义的“特定大小”数据类型，语意接近，但语法稍有不同。因为这些编译器之间的差异，多数游戏引擎会定义自定的基本数据类型，以获得代码的可移植性。例如，在顽皮狗中，我们定义了以下的基本数据类型。

- `F32`为32位IEEE-754浮点数。
- `U8`、`I8`、`U16`、`I16`、`U32`、`I32`、`U64`、`I64`为无符号和带符号整数，依序代表8、16、32、64位的整数。
- `U32F`、`I32F`为“高速”无符号和带符号32位整数。这些数据类型用作保存32位值，但实质上在内存占据了64位的空间。目的是让PS3的中央处理器（基于PowerPC，名为PPU）能直接读 / 写这些变量到64位寄存器，比读 / 写32位变量的速度有显著提升。
- `VF32` 代表把4个float的SIMD值打包。

[12]译注：此外还有MISD和MIMD。这些都是费林对于高效能计算的分类方法。详情可参考http://en.wikipedia.org/wiki/Flynn's_taxonomy。

OGRE的基本数据类型

OGRE也定义了自己的基本数据类型。Ogre::uint8、uint16、uint32是基本的无符号特定大小整数类型。

Ogre::Real定义为实数浮点值。这个类型通常定义为32位（和float等价），但也可以通过把预处理器宏OGRE_DOUBLE_PRECISION设为1，那么就会在全局范围内重定义Ogre::Real为64位（即double）。仅当某个游戏对双精度运算有特别要求，才需要有改变Ogre::Real意义的能力，但实际上鲜有这种需求。图形处理器（GPU）总是使用32位或16位浮点运算[13]，CPU/FPU使用单精度浮点运算一般也较快，而SIMD矢量指令也是处理内含4个float的128位寄存器。因此，多数游戏倾向于坚持使用单精度浮点运算。

而Ogre::uchar、ushort、uint、ulong这几个类型仅仅是C/C++中unsigned char、unsigned short、unsigned int、unsigned long的缩写。因此，这些类型和它们相对的C/C++类型在功能上完全相同。

Ogre::Radian和Ogre::Degree特别有趣。这两个类型皆是简单Ogre::Real值的包装类。其主要功能是要描述硬编码的角度常数使用哪种角度单位，并在两种单位之间自动进行转换。此外，Ogre::Angle类型代表“预设”角度单位的角度。在OGRE应用刚启动时，程序员可选择预设的单位是弧度或度数。

比较意外的是，OGRE并没提供一些其他游戏引擎常见的基本数据类型。例如，OGRE没有定义带符号的8、16、64位整数类型。若读者要在OGRE上编写游戏引擎，某时刻可能需要自己定义这些类型。

3.2.1.6 多字节值及字节序

大于8位（1字节）的值称为**多字节量**（multi-byte quantity）。任何使用16位或以上的整数/浮点数值软件中，多字节量非常普遍。例如，整数值4660 = 0x1234可由两个字节0x12和0x34表示。0x12称为最高有效字节（most significant byte，MSB），0x34称为最低有效字节（least significant byte，LSB[14]）。32位的值中，例如0xABCD1234，最高有效字节是0xAB，最低有效字节是0x34。同样的概念也应用于64位整数及32/64位浮点数。

在内存中储存多字节整数有两种方式，不同的微处理器的选择有所不同（见图3.6）。

- **小端**（little-endian）：若微处理器储存多字节值的最低有效字节于较低的内存位置，

[13]译注：有些专门用作高性能计算的GPU是支持双精度浮点数运算的。不过直至2010年，一般游戏用GPU都不原生支持双精度浮点数运算。

[14]译注：因为MSB和LSB中的B可代表位或字节，以下遇到最高/低有效字节，均采用中文全写识别。

则该微处理器就是**小端**处理器。在小端的机器上，数字0xABCD1234在内存中储存为连续字节0x34、0x12、0xCD、0xAB。

- **大端**（big-endian）：若微处理器储存多字节值的最高有效字节于较低的内存位置，则该微处理器就是**大端**处理器。在大端的机器上，数字0xABCD1234在内存中储存为连续字节0xAB、0xCD、0x12、0x34。

```
U32 value  = 0xABCD1234;
U8* pBytes = (U8*)&value;
```

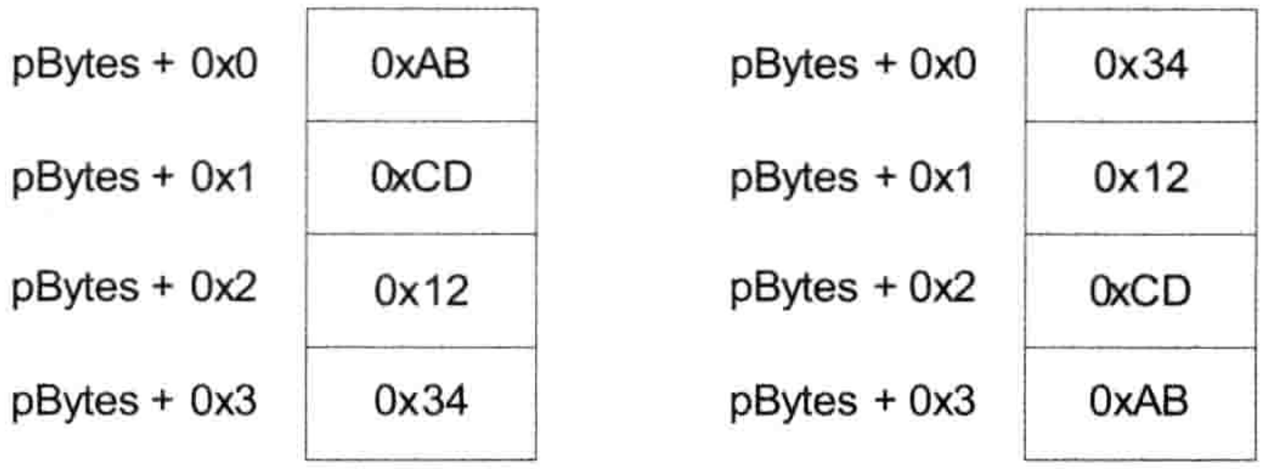


图 3.6: 数值0xABCD1234的大端和小端代表方式。

多数程序员不需要顾虑字节序（endianness[15]）。然而，字节序对游戏程序员来说可能是一根刺。因为游戏通常是在英特尔CPU（小端）的PC上**开发**，而游戏可能**执行**于游戏主机，如Wii、Xbox 360、PlayStation 3——这3台主机皆使用PowerPC处理器的变种（可设置使用任意字节序，但预设是大端）。如果产生一个数据文件，供英特尔处理器上运行的游戏引擎读取，之后再由PowerPC上运行的引擎读取，会发生什么事情呢？任何写到数据文件中的多字节值都是小端格式，但在PowerPC上运行的游戏引擎期望从数据文件中读取大字节格式的数据。结果是，若写的是0xABCD1234，就会读到0x3412CDAB，显然非我们所要。

此字节序问题最少有两个解决办法。

1. 所有数据以文字方式写入文件。多字节数值以一串十进制数字，每数字一个字节写入。此方法会浪费磁盘空间，但却可行。
2. 工具先转换数据字节序，然后再把转换后的数据写进二进制文件。也就是说，即使执行工具的机器字节序与目标机器相反，也可确保储存的数据为目标机器的字节序。[16]

[15]译注：字节序英文又称为byte order。

[16]译注：另一办法，是采用固定字节序的文件。程序读取文件时按需转换，例如，JPEG文件中的16位数字是以大端储存的。

整数字节序转换

整数字节序转换的概念并不复杂。首先把该值的最高有效字节和最低有效字节交换，再继续交换直到该值的中间点。例如，0xA7891023会变成0x231089A7。

唯一的难点在于要知道**哪些字节序**要转换。例如，把内存中的C struct或C++ class写入文件时，要正确地转换字节，便需要知道struct里每个数据成员的位置及大小，并基于每个成员的大小逐一进行适当转换。例如，以下的struct：

```
struct Example
{
    U32 m_a;
    U16 m_b;
    U32 m_c;
};
```

可用以下方式写入文件：

```
void writeExampleStruct(Example& ex, Stream& stream)
{
    stream.writeU32(swapU32(ex.m_a));
    stream.writeU16(swapU16(ex.m_b));
    stream.writeU32(swapU32(ex.m_c));
}
```

而转换函数可定义为：

```
inline U16 swapU16(U16 value)
{
    return ((value & 0x00FF) << 8)
         | ((value & 0xFF00) >> 8);
}

inline U32 swapU32(U32 value)
{
    return ((value & 0x000000FF) << 24)
         | ((value & 0x0000FF00) << 8)
         | ((value & 0x00FF0000) >> 8)
         | ((value & 0xFF000000) >> 24);
}
```

不可以简单地把Example对象转换为字节数组，并盲目地用单一通用函数去转换那些字节。必须知道哪些数据成员要转换及每个成员的大小，并逐个成员分别进行转换。

浮点字节序转换

让我们看看浮点字节序转换和整数字节序转换的差异。之前提及，IEEE-754标准中的浮点数有详细的内部结构，其中某些位作为尾数，某些位作为指数，并有一个符号位。虽然其结构比较复杂，但仍然可以把浮点数当作整数转换字节序，因为字节始终是字节。可以使用C++的`reinterpret_cast`操作去把浮点数诠释为整数，这又称为**类型双关**（type punning）。但是，当使用严格别名（strict aliasing）时，类型双关可能会导致优化bug。（有一篇文章对此问题做出了非常好的描述[17]。）取而代之，一个简便的方法是使用union：

```
union U32F32
{
    U32 m_asU32;
    F32 m_asF32;
};

inline F32 swapF32(F32 value)
{
    U32F32 u;
    u.m_asF32 = value;
    // 以整数方式转换字节序
    u.m_asU32 = swapU32(u.m_asU32);
    return u.m_asF32;
}
```

3.2.2 声明、定义及链接规范

3.2.2.1 再谈翻译单元

第2章曾经介绍，C/C++程序是由**翻译单元**组成的。编译器每次翻译一个.cpp文件，并输出个别的对象文件（.o或.obj）。编译器操作的最小翻译单位是.cpp文件，因此称为“翻译单元”。对象文件不仅含有.cpp文件内定义的所有函数编译后的机器码，也包含.cpp文件内定义的全局变量和静态变量。此外，对象文件也可能含有**未解决引用**（unresolved reference），这些未解决的引用是**其他**.cpp文件定义的函数和全局变量。

由于编译器每次操作只针对一个翻译单元，遇到未解决引用的外部变量和函数，只能“毫不怀疑地相信”该变量或函数真的存在，如图3.7所示。链接器的任务就是要把所有对象文件组合成为最终可执行映像（executable image）。在此过程中，链接器读取所有对象文

[17] http://cocoawithlove.com/2008/04/using-pointers-to-recast-in-c-is-bad.html

件，并尝试解决对象文件间的交叉引用。若链接成功，生成的可执行映像将包含所有函数、全局变量、静态变量[18]，并正确解决所有翻译单元间的交叉引用。图3.8显示了此结果。

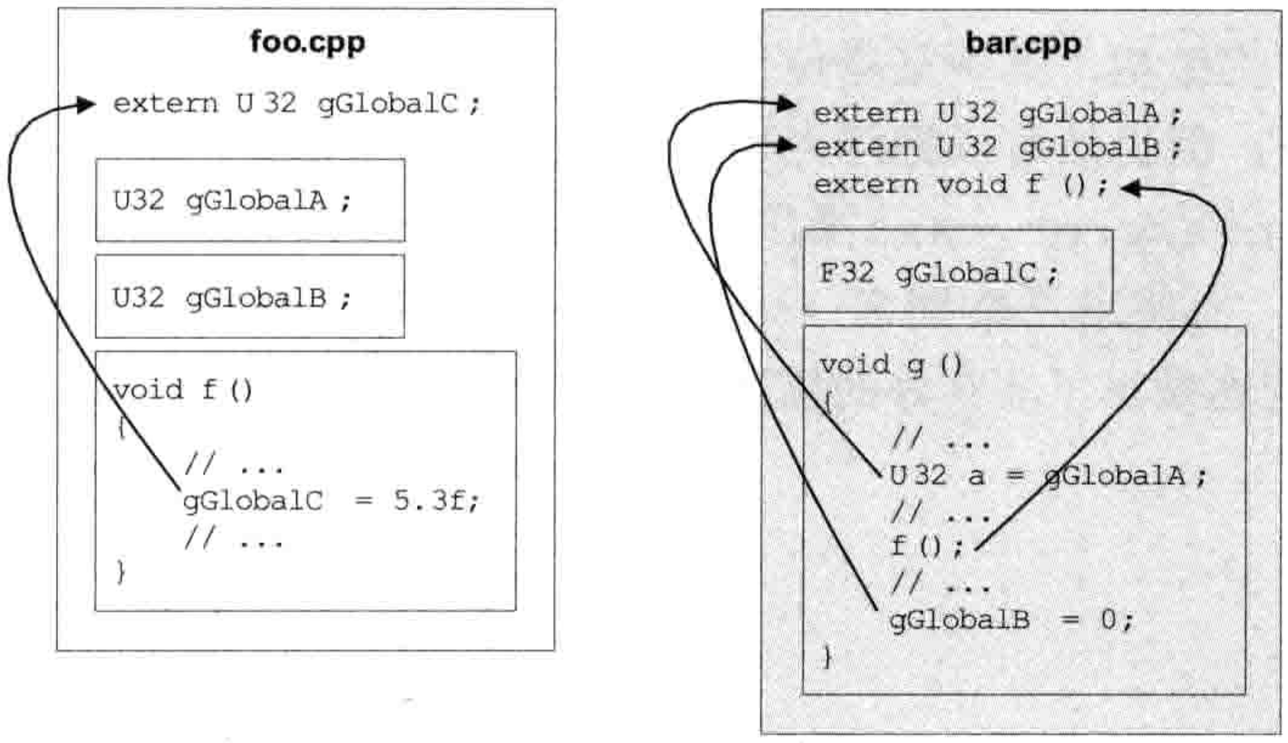


图 3.7: 无法解决的外部引用。

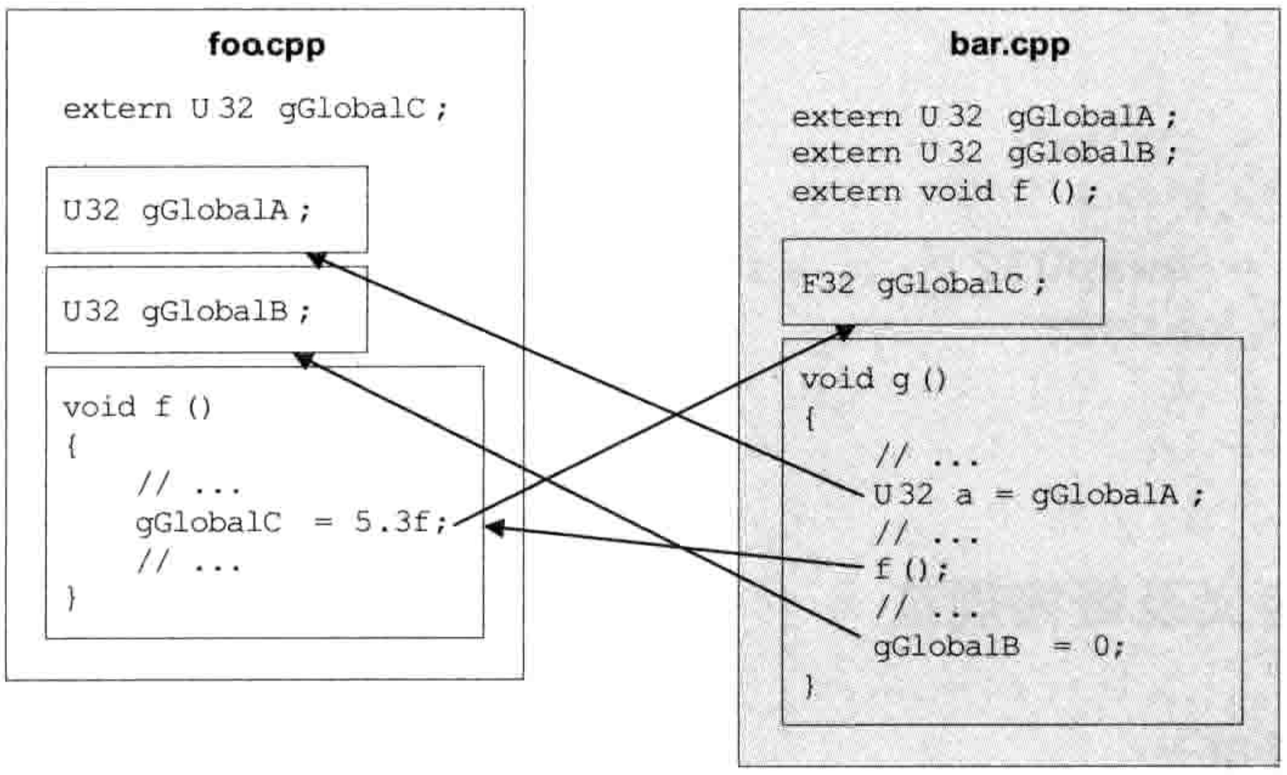


图 3.8: 成功链接后，被完全解决的外部引用。

链接器的主要功能是解决外部引用，因此也只能报告如下两种错误。

1. 若找不到extern引用的目标，链接器报告“无法解决的外部符号（unresolved external symbol）”错误。
2. 若找到两个或以上相同名字的实体（函数或变量），链接器报告“符号被多重定义（multiply defined symbol）”错误。

[18]译注：比较正确的说法是，包含所有“被引用的对象文件内”的函数、全局变量、静态变量。因为一般链接是以对象文件为单位的，故没有被引用的对象文件不会链接进可执行映像。然而，Visual C++提供函数级链接（function-level linking）功能，使用后链接的单位便由对象文件变成函数。

图3.9描绘了这两种情况。

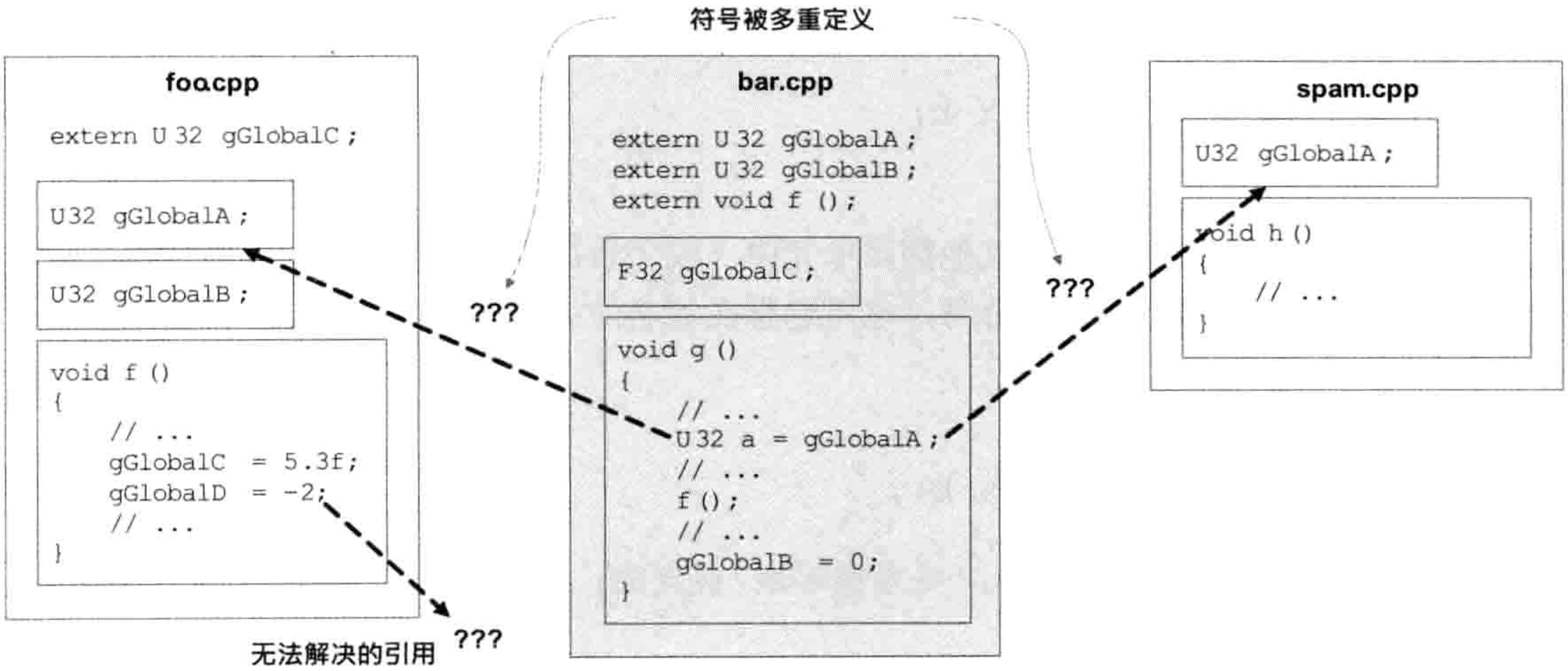


图 3.9: 两个最常见的链接错误。

3.2.2.2 声明与定义

在C和C++语言中，变量和函数必须先**声明**（declare）和**定义**（define），然后才可使用。了解C/C++的声明和定义之区别十分重要。

- **声明**（declaration）是数据对象或函数的描述。声明使编译器知道实体（数据对象或函数）的名字，以及其数据类型或函数签名（function signature，即函数的返回值类型、一个至多个参数类型）。
- **定义**（definition）则是程序中个别内存区域的描述。此内存区域可能用来放置变量、struct或class的实例，以及函数的机器码。

换言之，声明是**实体的引用**，而定义是**实体本身**。一个定义必然是一个声明，但相反则不然——C/C++中可以编写不属于定义的纯粹声明。

函数**定义**的写法是，用大括号包裹函数主体，并置于函数签名之后：

```
// foo.cpp
// 函数max()的定义
int max(int a, int b)
{
    return (a > b) ? a : b;
}
```

```
// 函数min()的定义
int min(int a, int b)
{
    return (a <= b) ? a : b;
}
```

函数可提供纯粹声明，使在其他翻译单元中（或之后在同一翻译单元中）能使用。纯粹声明的写法是，为函数签名加上分号，也可选择在签名前加上extern前缀：

```
// foo.h
// 一个函数声明
extern int max(int a, int b);

// 也是一个函数声明 ('extern'是非强制的/假定的)
int min(int a, int b)
```

定义变量和class/struct实例的写法是，先写出数据类型，再加上变量/实例的名字，最后可选择加入数组定义的方括号：

```
// foo.cpp
// 以下都是变量定义:
U32 gGlobalInteger = 5;
F32 gGlobalFloatArray[16];
MyClass gGlobalInstance;
```

在某翻译单元定义的全局变量中，可使用extern关键字做**声明**，以供其他翻译单元使用：

```
// foo.h
// 以下都是变量声明:
extern U32 gGlobalInteger;
extern F32 gGlobalFloatArray[16];
extern MyClass gGlobalInstance;
```

多重声明和定义

不奇怪，任何C/C++里的数据对象或函数都可以有多个同等的**声明**，但只能有一个**定义**。若有两个或更多的同等定义位于一个翻译单元，编译器会报告有多个同名实体的错误。但若有两个或更多同等定义存在于**不同的**翻译单元，编译器则发现不了错误，因为编译器每次是以翻译单位运作的。但是，此情况下，链接器就会在解析交叉引用时报告“符号被多重定义”错误。

头文件中的定义，以及内联

把**定义**置于头文件中，通常是危险的。原因很简单：若多个**.cpp**文件#include了含有**定义**的头文件，就肯定会产生“符号被多重定义”的链接错误。

对此规则，内联函数（inline function）是个例外，因为每个调用内联函数的地方都会复制该函数的机器码，并把机器码直接嵌入调用方的函数里。实际上，若内联函数要供多于一个翻译文件使用，则该内联函数**必须**置于头文件中。只是为**.h**文件内的函数声明加上inline关键字，并把函数主体置于**.cpp**文件内，是**不**够的。编译器必须“看见”函数主体才能把函数内联。例如：

```
// foo.h
// 此函数会正确地内联
inline int max(int a, int b)
{
    return (a > b) ? a : b;
}

// 此函数不能内联,因为编译器“看不见”函数主体
inline int min(int a, int b);

// foo.cpp
// 编译器实质上“看不见”min()的主体，所以min()只能在foo.cpp中内联
int min(int a, int b)
{
    return (a <= b) ? a : b;
}
```

其实，inline关键字只是给编译器的提示（hint）。编译器会为每个内联函数分析其内联的成本效益，即量度函数代码大小，对比内联该函数的潜在效率收益，决定是否对函数进行内联，编译器有最终决定权。某些编译器提供语法如__forceinline，让程序员绕过编译器的成本效益分析，直接进行内联。

3.2.2.3 链接规范

每个C/C++的定义都有名为**链接规范**（linkage）的属性。**外部链接**（external linkage）的定义可被定义处以外的翻译单元看见并引用。**内部链接**（internal linkage）的定义则只能被该定义所处的翻译单元看见，而不能被其他翻译单元引用。我们称此属性为**链接规范**，因为它决定链接器是否容许该实体做交叉引用。因此，在某种意义上，链接规范类似C++ public:/private:关键字在定义类时的作用，不过其作用对象不是类而是翻译单元。

所有定义预设为外部链接。使用static关键字可以把定义改为内部链接。要注意，两个或以上的**.cpp**文件可含有相同的static定义，因为链接器认为这些定义是**不同的实体**（如同给予不同的名字），故链接器不会产生“符号被多重定义”错误。以下是一些例子：

```
// foo.cpp
// 此变量可供其他.cpp文件使用(外部链接)
U32 gExternalVariable;

// 此变量仅供foo.cpp内使用(内部链接)
static U32 gInternalVariable;

// 此函数可供其他.cpp文件调用(外部链接)
void externalFunction()
{
    // ……
}

// 此函数仅供foo.cpp内调用(内部链接)
static void internalFunction()
{
    // ……
}

// bar.cpp
// 此声明给予foo.cpp变量存取权限
extern U32 gExternalVariable;

// 此gInternalVariable有别于foo.cpp定义的同名变量，不会产生错误
// 这如同我们把此变量命名为gInternalVariableForBarCpp，效果是一样的
static U32 gInternalVariable;

// 此函数有别于foo.cpp定义的版本，不会产生错误
// 这如同我们把此函数命名为internalFunctionForBarCpp，效果是一样的
static void internalFunction()
{
    // ……
}

// 错误：符号被多重定义！
void externalFunction()
{
    // ……
}
```

从技术上来说，**声明**不会有链接属性，因为声明不会在可执行映像中分配储存空间；因此，不存在链接器是否容许交叉引用那些储存空间的问题。声明仅作为引用，指向其他地方定义的实体。然而，有时候为了方便，可以说声明是内部链接的，因为声明只对它出现的翻译单元有效。若我们容许这样拓宽此术语的含义，那么声明**总是**内部链接的，无法在多个.cpp文件中交叉引用单个声明。（若我们把声明置于头文件，则多个.cpp文件能“看见”声明，但实际上每个翻译单元都独立**复制**了该声明，而声明的复制版本在每个翻译单元中都是内部链接的。）

此论述能使我们了解内联函数被定义为允许置于头文件中的真正原因：因为内联函数预设就是**内部链接**的，就好像它们被声明为static一样。若多个**.cpp**文件#include了包含内联函数的头文件，每个翻译单元就各有一份私有的内联函数主体复制版本，因而不会产生“符号被多重定义”错误。链接器把每个复制版本视为不同的实体。

3.2.3 C/C++内存布局

由C/C++编写的程序，会把其数据储存于内存的多个地方。要了解储存空间如何分配、多种C/C++变量类型如何运作，我们需要认识C/C++程序的内存布局（memory layout）。

3.2.3.1 可执行映像

当生成C/C++程序时，链接器创建可执行文件。像UNIX的操作系统，包括许多游戏主机，都使用一种流行的可执行文件格式，称为**可执行与可链接格式**（executable and linkable format，ELF）[19]。因此，在这些平台上的可执行文件使用**.elf**扩展名。Windows的可执行文件格式与ELF格式相似。在Windows上的可执行文件使用**.exe**扩展名。无论是何种格式，可执行文件总是包含程序的部分**映像**（image），程序执行时此部分映像会置于内存中。称为“部分”映像的原因是，由于程序除了把可执行映像置于内存中，一般也会分配额外内存。

可执行映像分为几个相连的块，这些块称为段（segment）或节（section）。每个操作系统的可执行文件布局方式都有些差异，同一个操作系统里的不同可执行文件也会有些微差异。但映像文件一般最少由以下4个段组成。

1. **代码段**（text/code segment）：此段包含程序中定义的全部函数的可执行机器码。
2. **数据段**（data segment）：此段包含全部**获初始化**的全局及静态变量。链接器为这些变量分配所需内存，其内存布局将会和程序执行时完全一样，并且链接器会填入适当的初始值。

[19]译注：原文“executable and linking format”是较老的叫法。

3. **BSS段**（BSS segment）：“BSS”是过时的名字，原意是“由符号开始的块（block started by symbol）”。BSS段包含程序中定义的所有**未初始化**全局变量和静态变量。C/C++明确地定义，任何未初始化的全局变量和静态变量皆为零。不过与其储存可能很大块的零值在BSS段，链接器只需简单地储存所需零值的字节**个数**，足以安置此段内未初始化的全局及静态变量便可。当操作系统载入程序时，便会保留BSS段所需的字节个数，并为该部分内存填入零，之后才调用程序进入点（如`main()`或`WinMain()`）。
4. **只读数据段**（read only data segment）：此段又称为**rodata**段，包含程序中定义的只读（常量）全局变量。例如，所有浮点常量（如`const float kPi = 3.141592f;`）及所有用`const`关键字声明的全局对象实例（如`const Foo gReadOnlyFoo;`）就是隶属此段。注意编译器通常把整数常量（如`const int kMaxMonsters = 255;`）视为**明示常量**（manifest constant），并且直接把明示常量插进机器码之中。明示常量直接占用了代码段的储存空间，而不储存于只读数据段。

全局变量——是指由所有函数及类声明外的**文件作用域**（file scope）定义的变量——按照是否被初始化，而决定储存于数据段或BSS段。下例中的全局变量之所以储存于数据段，是因为它已被初始化：

```
// foo.cpp
F32 gInitializedGlobal = -2.0f;
```

因程序员没初始化下列变量，按BSS段的规范，操作系统将为它分配空间并初始化为零：

```
// foo.cpp
F32 gUnInitializedGlobal;
```

我们已知道，可用`static`关键字来把全局变量或函数指明为**内部链接**，使其“不显露”于其他翻译单元。但除此以外，也可用`static`关键字来声明置于**函数内**的全局变量。函数静态变量的**词法作用域**（lexical scope）只在其定义的函数之内（即变量的名字只能在函数内“见到”）。变量会在第一次调用其函数时被初始化（而不像文件域静态变量，在`main()`调用前已被初始化）。但是，以可执行映像的内存布局来说，函数静态变量和文件域静态变量并无二致，都是根据是否被初始化而分别存于数据段或BSS段的。

```
void readHitchhikersGuide(U32 book)
{
    static U32 sBooksInTheTrilogy = 5; // 数据段
    static U32 sBooksRead;             // BSS段
    // ……
}
```

3.2.3.2 程序堆栈

当可执行程序被载入内存时，操作系统会保留一块称为**程序堆栈**（program stack）[20]的内存。当调用函数时，一块连续的内存就会压入栈，此内存块称为**堆栈帧**（stack frame）。若函数a()调用函数b()，函数b()的新堆栈帧就会被压入a()堆栈帧之上。当b()返回时，其堆栈帧就会弹出，并于调用b()之后的位置继续执行a()。

堆栈帧储存3类数据。

1. 堆栈帧储存调用函数的**返回地址**（return address）。当函数返回时，就可以凭这一数据继续执行调用方的函数。
2. 堆栈帧保存相关**CPU寄存器**的内容。借此过程，被调用方可以使用任何觉得合适的寄存器，而不必担心调用方所需的数据被覆盖。当函数返回时，各寄存器就会还原至调用方可继续执行的状态。若函数有返回值，该值会储存于指定的寄存器中，使调用方能取用，但其他寄存器会恢复原来的值。
3. 堆栈帧也包含函数里的所有**局部变量**（local variable），或称**自动变量**（automatic variable）。借此过程，每个函数调用都各自保持一组私有的局部变量集合，甚至函数对自己递归回调也是一样的。（实际上，一些局部变量会分配使用CPU寄存器，而不是存于堆栈帧。但是，这些变量在大部分情况下，运作方式如同使用堆栈帧。）例如：

```
void someFunction()
{
    U32 anInteger;
    // ......
}
```

堆栈帧的压入和弹出操作，一般会通过调整一个CPU寄存器的值实现，此寄存器称为堆栈指针（stack pointer）。图3.10显示了以下函数执行时的情形。

```
void c()
{
    U32 localC1;
    // ......
}

F32 b()
{
```

[20]译注：比较常见的名称为调用堆栈（call stack）。操作系统为每个线程分配独立的调用堆栈。当程序载入时就会为主线程分配一个调用堆栈。

```
    F32 localB1;
    I32 localB2;
    // ……
    c(); // 调用函数c()
    // ……
    return localB1;
}

void a()
{
    U32 aLocalsA1[5];
    // ……
    F32 localA2 = b(); // 调用函数b()
    // ……
}
```

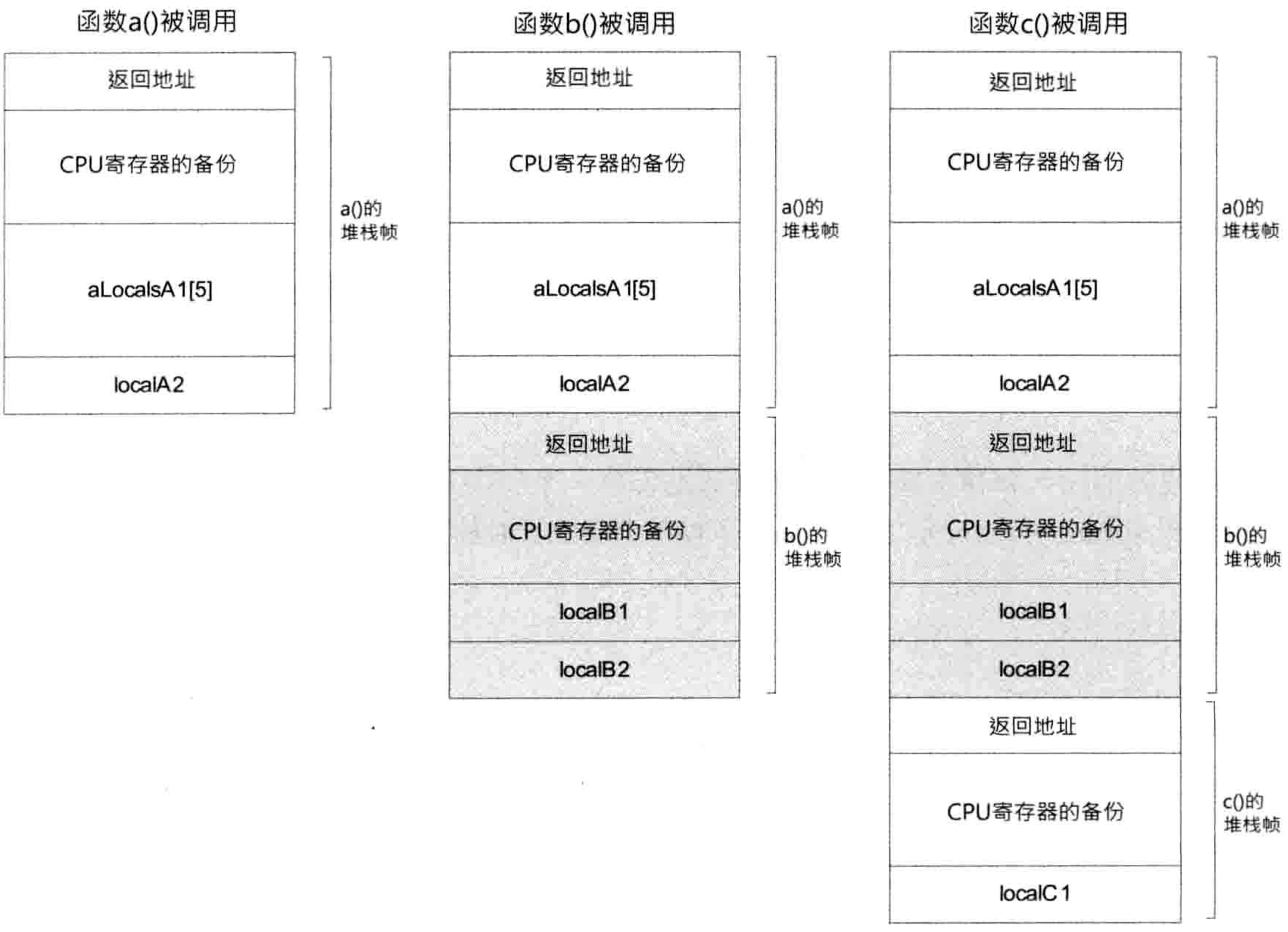


图 3.10: 堆栈帧。

当含自动变量的函数返回时，其堆栈帧就会被舍弃，该函数内的所有自动变量被视为不再存在。从技术上来说，这些变量所占的内存仍然在已被舍弃的堆栈帧中，当调用下一个函数时，这些变量所占的内存就很可能被覆盖。一个常见错误是返回局部变量的地址，例如：

```
U32* getMeaningOfLife()
{
    U32 anInteger = 42;
    return &anInteger;
}
```

只有程序立即使用返回的地址，并且在使用期间不调用其他函数，才**有可能**不会出事。多数情况下，这类代码会招来崩溃，而且这种类型的崩溃往往难以调试。

3.2.3.3 动态分配的堆

迄今为止，讨论了程序中的数据如何储存为全局、静态或局部变量。全局和静态变量分配于可执行映像里，而局部变量则分配于程序堆栈之中。这两种储存方式都是**静态地**定义的，这意味着，其所需的内存大小与布局在编译链接程序时便能知道。可是有时候，不能在编译期完全知悉程序的内存需求。程序经常需要**动态地**分配额外的内存。

为了提供动态分配功能，操作系统会维护一块内存，当运行程序调用`malloc()`时就会从中分配，稍后调用`free()`可把内存交还。此内存块称为**堆内存**（heap memory）或**自由存储**（free store）[21]。当动态分配内存时，我们有时候称分配得来的内存是**置于堆中**的。

C++中，全局`new`和`delete`操作符用来从自由存储分配和释放内存。然而要注意，个别的类可能重载了这两个操作符，用自定义方式分配内存，因此，不能简单假设`new`必然从自由存储中分配内存。

第6章会更深入探讨动态内存管理。此外，可参考维基百科[22]获得更多信息。

3.2.4 成员变量

C `struct`和C++ `class`都可用来把变量组成逻辑单元。谨记`class`或`struct`的**声明**并不占用内存。这些声明仅是数据布局的描述，如同一个模具用来制作`struct`或`class`的**实例**。例如：

[21] 译注：从技术上来说，堆是C语言和操作系统的术语，而自由存储是C++中通过`new`和`delete`动态分配和释放对象的抽象概念。基本上，所有C++编译器预设会使用堆去实现自由储存，但程序员可通过重载操作符，改用其他内存实现自由储存，例如全局变量做的对象池。

[22] `http://en.wikipedia.org/wiki/Dynamic_memory_allocation`

```
struct Foo // struct定义
{
    U32  mUnsignedValue;
    F32  mFloatValue;
    bool mBooleanValue;
}
```

当声明了一个struct或class，就能以和基本数据类型相同的任何方式进行分配（定义），例如：

- 作为自动变量，置于程序堆栈上：

```
void someFunction()
{
    Foo localFoo;
    // ……
}
```

- 作为全局、文件静态或函数静态变量：

```
Foo gFoo;
static Foo sFoo;

void someFunction()
{
    static Foo sLocalFoo;
    // ……
}
```

- 动态地从自由存储中分配。在此情况下，储存数据地址的指针或引用本身可以用自动、全局、静态，甚至动态方式分配。

```
Foo* gpFoo = NULL;  // 指向一个Foo的全局指针

void someFunction()
{
    // 从自由存储中分配一个Foo实例
    gpFoo = new Foo;

    // 分配另一个Foo实例，赋值至局部变量
    Foo *pAnotherFoo = new Foo;

    // 从自由存储中分配一个Foo指针
    Foo** ppFoo = new Foo*;
    (*ppFoo) = pAnotherFoo;
}
```

3.2.4.1 类的静态成员

如前所见，根据不同的上下文，static关键字有许多不同的含意。

- 当用于文件作用域时，static意味着“限制变量或函数的可见性（visibility），只有本.cpp文件才能使用该变量或函数”。
- 当用于函数作用域时，static意味着“变量为全局，非自动，只在本函数内可见”。
- 当用于struct或class声明时，static意味着“该变量非一般成员变量，而是类似于全局变量”。

注意当static用于class声明时，并不控制该变量的**可见性**（文件作用域才会），反而，其用途是区分正常的每个实例（per-instance）变量，以及行为像全局变量的每个类（per-class）变量。类静态变量的**可见性**是通过声明中的public:、private:、protected:关键字决定的。类静态变量自动包含于其被定义的class或struct命名空间（namespace）里。若在class或struct以外使用这些变量，必须加入class或struct的名字以消除歧义（例如Foo::sVarName）。

如同加上extern的一般全局变量，类声明内的类静态变量并不占内存。必须于一个**.cpp**文件内定义类静态变量以分配内存。例如：

```
// foo.h
class Foo
{
public:
    static F32 sClassStatic; // 不分配内存
};

// foo.cpp
F32 Foo::sClassStatic = -1.0f; // 定义内存及初始化
```

3.2.5 对象的内存布局

用图形来显示class或struct的内存布局十分有用，直接了当，只需画个长方形代表class或struct，以横线把数据成员分隔开即可。例如，以下的struct Foo就可以画成图3.11。

各数据成员的大小很重要，应于图上标明。对于每个数据成员，可用宽度显示其占据的位数大小，例如，32位整数的宽度应该是8位整数的4倍（见图3.12）。

```
struct Foo
{
    U32 mUnsignedValue;
    F32 mFloatValue;
    I32 mSignedValue;
};
```

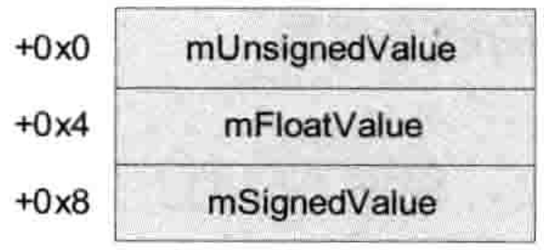


图 3.11: 简单struct的内存布局。

```
struct Bar
{
    U32  mUnsignedValue;
    F32  mFloatValue;
    bool mBooleanValue; // 假设8位
};
```

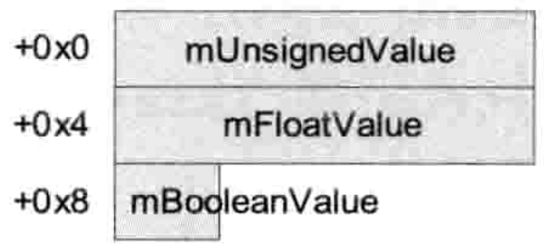


图 3.12: 内存布局用宽度显示成员大小。

3.2.5.1 对齐和包裹

当我们加倍留意struct和class在内存中的布局时，便会想弄清楚，若大小不一的数据成员交错放置，布局有何改变。例如：

```
struct InefficientPacking
{
    U32   mU1; // 32位
    F32   mF2; // 32位
    U8    mB3; // 8位
    I32   mI4; // 32位
    bool  mB5; // 8位
    char* mP6; // 32位
};
```

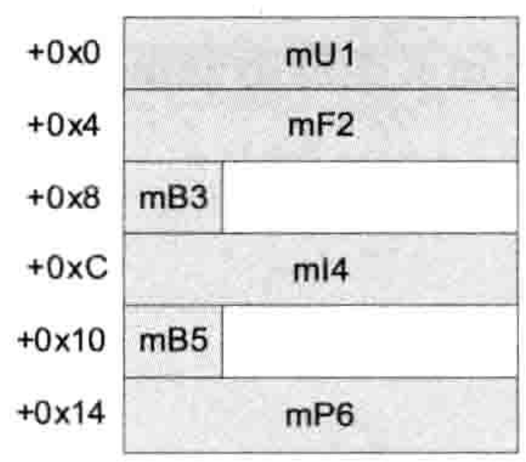


图 3.13: 混合数据成员大小导致低效的struct包裹。

读者可能会想象，编译器简单地把数据成员尽可能紧凑地包裹在一起。然而这毕竟不是常态。相反，编译器通常会在布局中留下空隙，如图3.13所示。（一些编译器可通过#pragma pack或命令行设置不留空隙，但预设行为会像如图3.13所示的那样在数据成员之间留空）。

为何编译器要留下这些空隙呢？原因在于，事实上每种数据类型有其天然的**对齐**（alignment）方式，供CPU高效地从内存读 / 写。数据对象的**对齐**是指，其**内存地址**是否为对齐字节**大小**的倍数（通常是2的幂）。

- 1字节对齐的对象，可置于任何地址。
- 2字节对齐的对象，只可置于偶数地址（即地址最低有效半字节（least significant nibble）为0x0、0x2、0x4、0x8、0xA、0xC或0xE）。

- 4字节对齐的对象，只可置于为4倍数的地址（即地址最低有效半字节为0x0、0x4、0x8或0xC）。
- 16字节对齐的对象，只可置于为16倍数的地址（即地址最低有效半字节为0x0）。

对齐是重要的，因为现在许多处理器实际上只能正常地读 / 写已对齐的数据块。例如，程序要求从0x6A341174地址读取32位（4字节）整数，内存控制器（memory controller）便可愉快地载入数据，因为该地址是4位字节对齐的（此例的最低有效半字节为0x4）。可是，若要从0x6A341173载入32位整数，内存控制器就需要读入**两个**4字节块：一块位于0x6A341170，另一块位于0x6A341174。之后，还需要通过掩码（mask）和移位（shift）操作取得32位整数的两部分，再用逻辑OR操作把两部分合并，把结果写入CPU的目标寄存器。图3.14显示了这个过程。

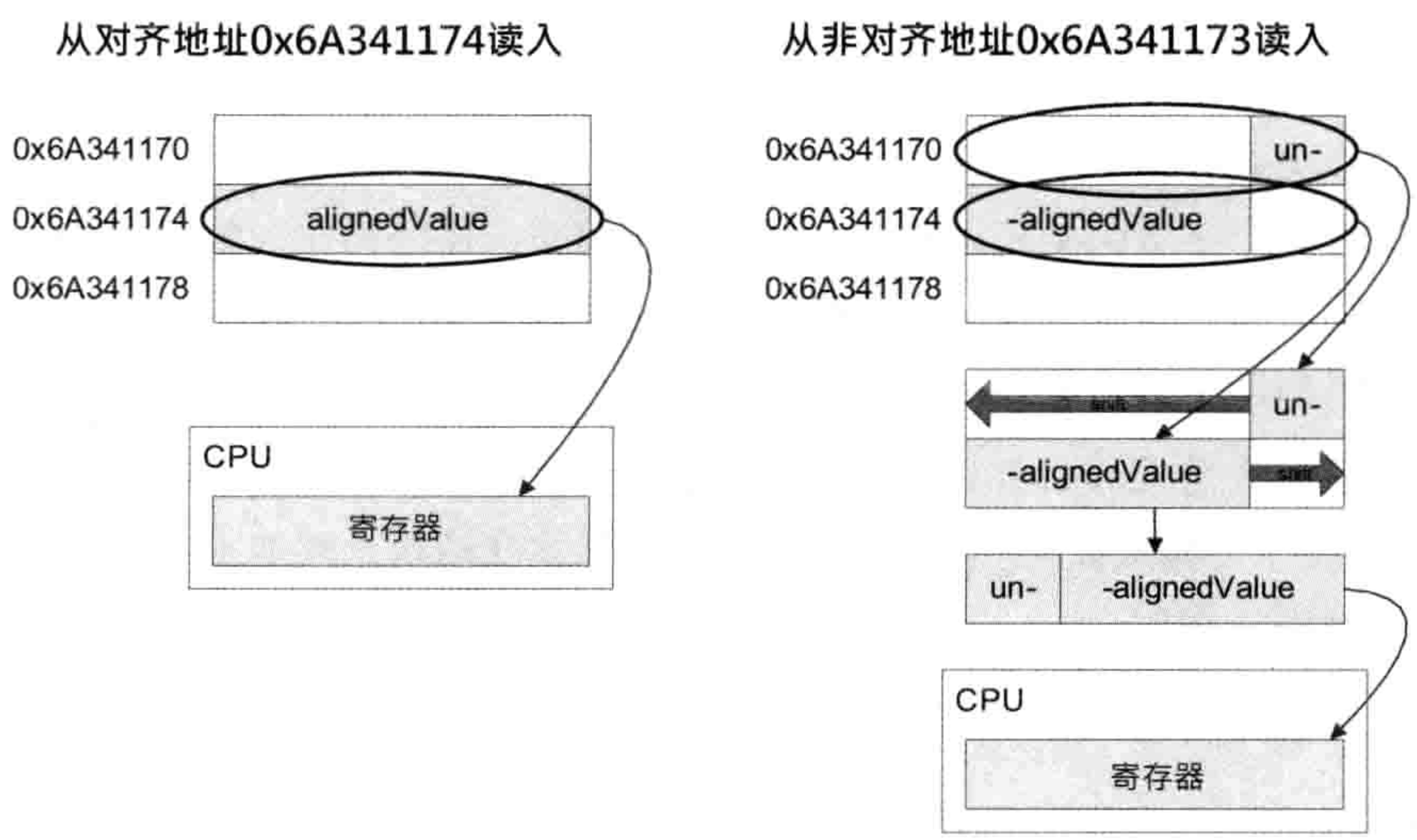


图 3.14: 对齐和非对齐地读取32位整数。

一些微处理器甚至不做这些处理。若读 / 写非对齐数据，读出来的或写进去的可能只是随机数。对于另一些微处理器，程序甚至会崩溃！（PS2就是这类“零容忍”的著名例子。）

各种数据类型有不同的对齐需求。作为一个良好经验法则，数据类型应该需要其字节大小的对齐。例如，32位值通常需要4字节对齐，16位值通常需要2字节对齐，8位值通常可存于任何地址（1字节对齐）。在支持SIMD矢量数学的CPU中，每个SIMD寄存器含32个4字节浮点数，共128位（16字节）。读者可能猜到了，包含4个浮点数的SIMD矢量通常需要16字节对齐。[23]

[23]译注：译者也为这个经验法则举出一个反例，在32位Linux下的8字节double，预设是4字节对齐的。

现在，我们可以重新考虑图3.13中struct InefficientPacking布局里的空隙。在class或struct中，当把较小的数据类型（如8位的bool）放置于较大类型（如32位的float）之间，编译器便会加入填充（padding）（空隙），以保证所有成员都是正常地对齐的。当声明数据结构时，认真对待对齐和包裹是个好习惯。如以下的代码及图3.15所示，只需简单地重新排列上述例子中的成员，就能省去一些浪费了的填充空间。

```
struct MoreEfficientPacking
{
    U32    mU1; // 32位(4字节对齐)
    F32    mF2; // 32位(4字节对齐)
    I32    mI4; // 32位(4字节对齐)
    char*  mP6; // 32位(4字节对齐)
    U8     mB3; //  8位(1字节对齐)
    bool   mB5; //  8位(1字节对齐)
};
```

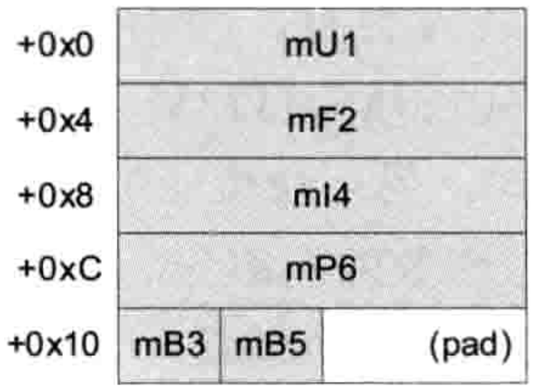


图 3.15: 小成员组合在一起，包裹更高效。

读者可能注意到在图3.15中，整个结构的大小是20字节，而非我们预期的18字节。这是由于在末端加进两个字节的填充。编译器加上这种填充，使结构作为**数组**类型时，仍能维持正确的对齐。换言之，若定义此结构的数组，并且其首个元素是对齐的，那么结构末端的填充保证了**所有之后的数组元素**皆是正确对齐的。

整个结构的对齐需求等同于其成员中的最大对齐需求。在以上例子中，最大的成员对齐是4字节，因此整个结构需要4字节对齐。笔者通常喜欢在结构末端加上明确的填充，使浪费了的空间更为清晰，例如：

```
struct BestPacking
{
    U32    mU1;       // 32位 (4字节对齐)
    F32    mF2;       // 32位 (4字节对齐)
    I32    mI4;       // 32位 (4字节对齐)
    char*  mP6;       // 32位 (4字节对齐)
    U8     mB3;       //  8位 (1字节对齐)
    bool   mB5;       //  8位 (1字节对齐)
    U8     _pad[2]; // 明确的填充
};
```

3.2.5.2 C++中类的布局

在内存布局上，C++的类有别于C的结构之处有二——**继承**与**虚函数**。[24]

[24]译注：在C++中，struct和class的区别只在于预设的成员可见性，struct的预设成员可见性为public，而class则为private。C++的struct一样可以有继承和虚函数。

当B类继承自A类，内存里B类的数据成员会紧接A类数据成员之后，如图3.16所示。每个新的派生类都会简单地把其数据成员附加到末端，即使类之间可能因对齐而加入填充。（多重继承比较混乱，例如会在派生类的内存布局中包含同一基类的多个版本。在此不再详述其细节，因为游戏程序员通常宁愿完全避免使用多重继承。）

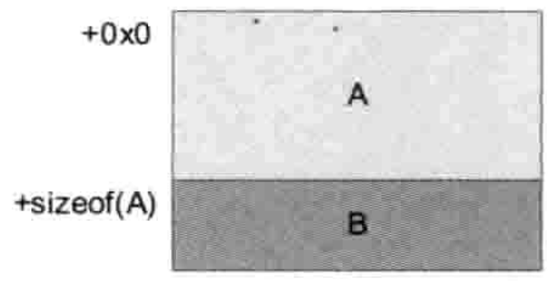


图 3.16: 继承对类布局的影响。

当类含有或继承了一个或多个**虚函数**（virtual function），那么就会在类的布局里添加4字节（或目标硬件中指针需占的字节数目），通常会加在类的布局最前端。此4字节称为**虚表指针**（virtual table pointer或vpointer），因为此4字节代表一个指针，指向名为**虚函数表**（virtual function table，vtable）的数据结构。在每个类的虚函数表里，包含该类声明或继承而来的所有虚函数指针。每个（含虚函数的）具体类都具有一个虚函数表，并且这些类的实例都会有虚表指针指向该虚函数表。

虚函数表是多态的核心，因为它促使我们在编写代码时无须考虑代码是和哪个具体类进行沟通的。回到那无处不在的例子，Shape为基类，Circle、Rectangle及Triangle为派生类。假设Shape定义了名为Draw()的虚函数，而所有派生类都重载了此函数，提供了个别的实现，包括Circle::Draw()、Rectangle::Draw()及Triangle::Draw()。任何继承自Shape的类，其虚函数表都有Draw()函数的条目，但条目会指向具体类的函数实现。Circle类的虚函数表包含指向Circle::Draw()的指针，Rectangle类的虚函数表则包含指向Rectangle::Draw()的指针，而Triangle类的虚函数表则包含指向Triangle::Draw()的指针。假设有一个指向Shape的指针（Shape *pShape），要调用其虚函数Draw()，代码可先对其虚表指针解引用，取得虚函数表，再从表中找Draw()的条目，即可调用。若pShape指向一个Circle类的实例，结果就是调用Circle ::Draw()，以此类推。

以下的代码片段可说明这些概念。注意基类Shape声明虚函数SetId()和Draw()，后者是**纯虚函数**（pure virtual function）。（纯虚函数指Shape不提供预设的Draw()实现，派生类若要能产生实例，必须重载此函数。）Circle类派生自Shape类，加入了一些数据及函数成员去管理其圆心及半径，并重载了Draw()函数，图3.17描画了Circle类的布局。Triangle类也派生自Shape类，加入了Vector3对象数组以储存3个顶点，并且提供函数去存取各顶点。如同意料之中，Triangle类也重载了Draw()函数，并因示范原因也重载了SetId()。图3.18显示了Triangle类的内存布局。

```
class Shape
{
public:
    virtual void SetId(int id) { m_id = id; }
    int          GetId() const { return m_id; }
    virtual void Draw() = 0; // 纯虚，无实现

private:
    int m_id;
};

class Circle : public Shape
{
public:
    void           SetCenter(const Vector3& c) { m_center = c; }
    const Vector3& GetCenter() const { return m_center; }

    void           SetRadius(float r) { m_radius = r; }
    float          GetRadius() const { return m_radius; }

    virtual void Draw()
    {
        // 绘画圆形的代码
    }

private:
    Vector3 m_center;
    float   m_radius;
};

class Triangle : public Shape
{
public:
    void           SetVertex(int i, const Vector3& v);
    const Vector3& GetVertex(int i) const { return m_vtx[i]; }

    virtual void Draw()
    {
        // 绘画三角形的代码
    }

    virtual void SetId(int id)
    {
        Shape::SetId(id);
        // 专为三角形而设的额外工作
```

```
    }

private:
    Vector3 m_vtx[3];
};

void main(int,char**)
{
    Shape* pShape1 = new Circle;
    Shape* pShape2 = new Triangle;

    pShape1->Draw();
    pShape2->Draw()
}
```

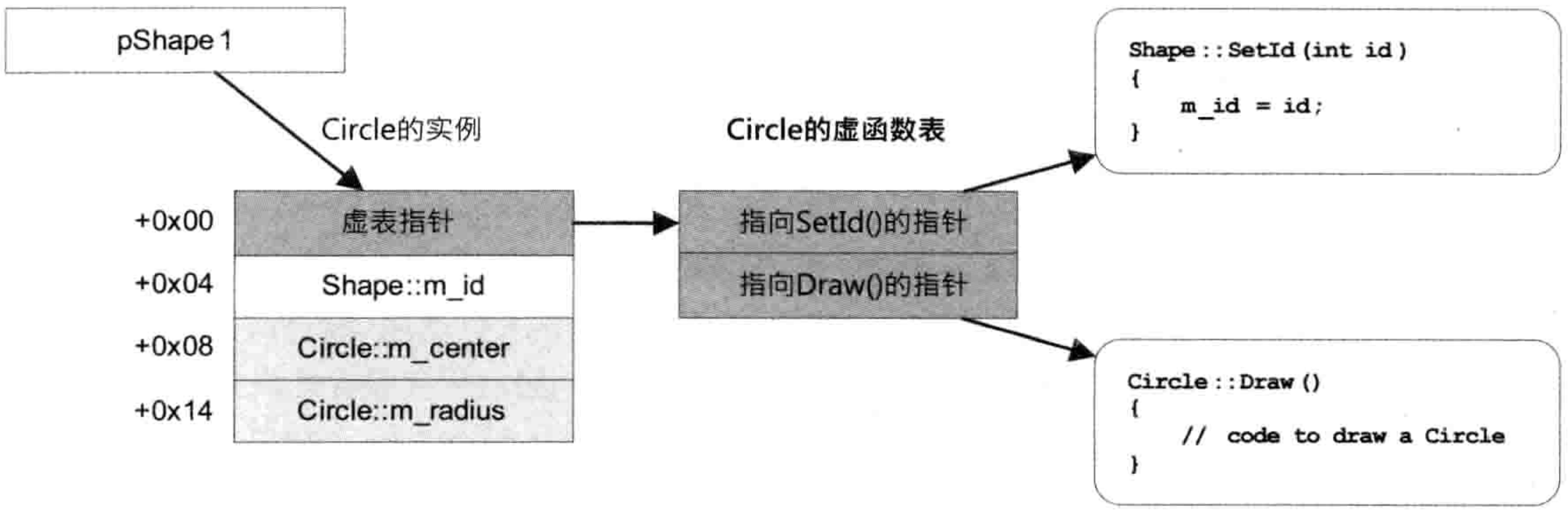


图 3.17: pShape1指向一个Circle类的实例。

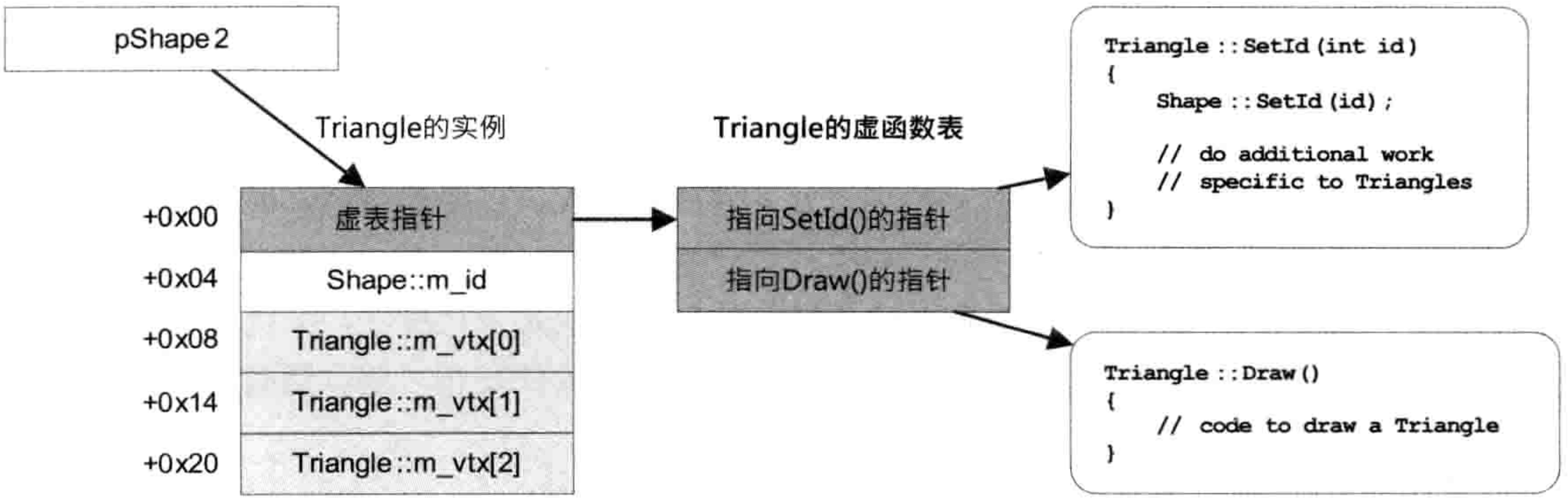


图 3.18: pShape2指向一个Triangle类的实例。

3.3 捕捉及处理错误

在游戏引擎中有多种方式捕捉及处理错误状况。游戏程序员必须理解这些机制各自的优劣之处以及适用时机。

3.3.1 错误类型

所有软件项目皆有两类基本错误状况：**用户错误**（user error）和**程序员错误**（programmer error）。用户错误，指用户做了些不正确的事情而引发的错误，例如键入无效的输入、尝试开启不存在的文件等。而程序员错误是由代码本身的**bug**所导致的结果。虽然程序员错误可能会因用户的某些操作而触发，但是程序员错误的本质是——若程序员不犯错，该问题是可以避免的，并且用户会合理地预期程序员**应该**妥善地处理该情况。

当然，基于不同的语境，“用户”的定义有所变化。在游戏项目的语境中，用户错误通常可分成两类：玩游戏者所引起的错误、制作游戏者在开发时所引起的错误。追踪受特定错误影响的用户种类，并恰当地处理错误，是游戏开发的重要一环。

实际上，还有第三类用户，就是团队里的其他程序员。（若开发对象是一套游戏中间件，如Havok、OpenGL，则此时的第三类用户就会扩展至全世界所有使用该软件的程序员。）就第三类用户来说，**用户错误**和**程序员错误**的分界变得模糊。想象程序员A君编写了函数`f()`，程序员B君尝试调用`f()`。若B君调用时采用了无效的参数（如空指针、超出范围的索引），则A君可视之为用户错误，但B君却可能视之为程序员错误。（当然，可以争辩说，A君应该预见会出现无效参数，并能优雅地处理这些参数，因此问题归咎于A君，属程序员错误。）此处的重点是，根据情况，用户和程序员之间的界限可能会转移，鲜有非黑即白的区分。

3.3.2 错误处理

处理这两类型错误的需求有重大差异。处理**用户错误**应该越妥善越好，并向用户显示有用信息，然后容许用户继续工作——若处于游戏状态下则继续玩。另一方面，程序员的错误**不**应采用“通知并继续”方针去处理。通常最好的处理方式是中止程序，并提供低阶调试信息，促使程序员能快速鉴定及修正问题。理想情况下，在软件发布之前，**所有**程序员错误都会被捕获及修正。

3.3.2.1 处理玩家错误

当“用户”为游戏玩家，显然要以游戏性来处理错误。例如，若玩家尝试为武器再装弹，而子弹用光了，则可使用声音提示及动画向玩家表明问题，而不应强制退出游戏。

3.3.2.2 处理开发者错误

当“用户”为游戏开发者，例如美术人员、动画师或游戏设计师等，错误可能来自某些无效资产。例如，某动画可能关联到一个错误的骨骼，某纹理的大小可能是无效的，或某音频文件可能使用不支持的采样率。对于这类**开发者错误**，有两种截然不同的看法。

一方面，避免坏游戏资产持续存在过久似乎很重要。游戏通常包含数以千计的资产，若某问题资产“不知所踪”，则其有可能一直存在，甚至有藏于游戏发布版本的风险。若从极端角度去看，处理坏游戏资产最好的办法，就是当遇到任何一个游戏资产有问题时，便不容许游戏执行。造成无效资产的始作俑者，便会有强大的动机去立即移除或修正该资产。

另一方面，游戏开发是混乱的迭代过程，实际上鲜有从一开始就产生“完美”资产的情况。按此思路，游戏引擎应该要尽可能健壮，能处理几乎任何想象得到的问题种类，即使遇到完全错误的资产，仍然能够继续工作。但此方式依然不是最理想的，因为游戏引擎会因大量错误捕捉及错误处理代码而变得臃肿，而且当开发步伐平稳下来及发布游戏时，这些代码就会变得冗余。发布含“坏”资产的产品的概率变得太大。

据笔者经验，最佳方式是寻找这两个极端之间的平衡点。当发生开发者错误，笔者希望**让错误变得明显**，并使团队可于问题存在的情况下继续工作。若只因为某开发者尝试加入一个无效资产，就要团队中所有其他成员暂停工作，这样代价实在太昂贵了。游戏工作室向员工支付不菲的工资，当多个团队成员要暂停工作时，成本就会成倍增加。当然，我们应该在实际可行时才使用这种处理错误方式，不至于花费过多的工程时间，或使代码变得臃肿。

比方说，我们假设某个网格载入失败。依笔者的见解，最好是在游戏世界中所有放置该网格的地方画红色的大长方体，也许最好再在每个长方体上附上文字，内容是“网格**某某**载入失败”。这样比输出信息到错误日志**更胜一筹**，因为那些信息很容易被忽略。而且，这样也比令游戏崩溃**好得多**，因为崩溃会使所有人在该网格被修正前都不能工作。当然，对于极糟糕的问题，还是可以仅弹出错误信息并使程序崩溃的。所有问题皆无银弹[25]，读者须按个别情况，判断采用哪种错误处理方法，以改善开发者的使用体验。

---

[25]译注：银弹（silver bullet）是欧洲传说里对付吸血鬼、人狼等妖魔的终极武器。《没有银弹（no silver bullet）》是IBM大型计算机之父佛瑞德·布鲁克斯（Fred Brooks）于1987年所发表的一篇关于软件工程的经典论文。详见http://en.wikipedia.org/wiki/No_Silver_Bullet。

3.3.2.3 处理程序员错误

检测及处理程序员错误（也即是bug），最佳方法一般是在源代码中嵌入错误检测代码，并且当检测到错误时终止程序。此机制名为**断言系统**（assertion system），将于3.3.3.3节仔细讨论。当然，如同之前所述，某位程序员的使用错误是另一位程序员的bug，因此，断言并非处理所有程序员错误的正确方式。明智地选用断言或更妥善的错误处理技术，仍是每位程序员须不断学习的技能。

3.3.3 实现错误检测及处理

我们刚讨论过一些处理错误的理论，现在返回程序员身份，专注关于错误检测及处理的各种代码实现方式。

3.3.3.1 错误返回码

常见错误处理方法之一，是当函数检测到错误时，从该函数返回某种错误码。错误返回码（error return code）可以用布尔值表示函数执行的成败；也可用“不可能”的值去表示，“不可能”的值指正常返回结果范围以外的值。例如，某函数正常执行时会返回正整数或正浮点数，当发生错误时可返回负值以做标示。比返回布尔值或“不可能”值更佳的方法是，设计函数返回一个枚举值（enumerated value）以表明函数执行的成败。此方式能清楚地区分错误和函数输出，并可显示失败的确切原因，例如：

```
enum Error { kSuccess, kAssetNotFound, kInvalidRange, ...};
```

函数调用方应检查错误返回码，并做出相应行动。调用方可以即时处理错误，也可对问题做应急处理，完成本身的运作，再把错误码转交上一层的调用方。

3.3.3.2 异常

错误返回码是传达及回应错误状况的简单可靠方式。然而，错误返回码也有短处。其中最大的问题大概是，检测到错误的函数与可处理错误的函数完全无关。在最坏情况下，在调用堆栈里的第40个函数时，可能侦测到一个错误，而此错误必须由顶层游戏循环或`main()`函数处理。此情况下，调用堆栈里的40个函数都需要编写代码，逐一传送恰当的错误码至上一层，直至能处理错误的顶层函数。

其中一个解决办法是抛出异常（exception），**异常**[26]是C++的强大功能。它让检测到错误的函数，无须知道能处理该错误的函数，就可以把错误信息传到其余代码处。抛出（throw）异常时，程序员可选择把相关错误信息储存于某对象，其被称为**异常对象**（exception object）。之后，程序会自动进行堆栈辗转开解（stack unwinding），从堆栈中寻找位于try区块里的函数调用。找到try区块后，便会对异常对象的类与try区块下的逐个catch区块参数进行比较，若找到匹配的，就会执行该catch区块的代码。在堆栈辗转开解的过程中，会自动调用所有自动变量的析构函数（destructor）。

能把错误检测和错误处理这样清晰地分离，无疑是个诱人的方式，在某些软件项目中也是极佳的选择。然而，异常会为程序添加许多额外开销。每个调用帧会变大，用来承载堆栈辗转开解时所需的额外信息。并且，堆栈的辗转开解通常很慢，相比简单地返回函数，前者用时大约多一两倍。另外，就算程序中仅有一个函数（或程序链接的一个程序库）使用了异常，**整个程序**都必须使用。编译器不能预知抛出异常时调用堆栈中会有哪些函数。

因此，在游戏引擎中，有颇充分的理由去完全**关掉**异常处理。顽皮狗以及笔者在艺电、Midway工作时的大部分项目都采用此方式。因为游戏主机的内存和效能都是有限的，游戏主机上的引擎大概永远不会使用异常。然而，为基于个人计算机而开发的游戏引擎，可以安然使用异常。网上有不少讨论这方面的文章[27, 28, 29]。

3.3.3.3 断言

断言（assertion）是一行检查表达式的代码。当表达式求值为真，一切如常。但若表达式求值为假，则暂停程序，打印消息，并在可行的情况下启动调试器。在Steve Maguire的必看名著《Writing Solid Code》[30][30]里能看到有关断言的深入讨论。

断言检查程序员的假设，就像是为bug而设的**地雷**。编写新代码时，断言可检查代码，以保证代码的功能正常。断言也能保证在一段较长的时间内，即使周遭的代码经常改动、演变，原来的假设继续维持成立。例如，某程序员改动一些本来能工作的代码，但意外地违返了原来的假设，那么该程序员就是踩到“地雷”了。断言会立即通知程序员问题所在，让程序员在最小影响的情况下纠正问题。没有断言的话，很多bug喜欢“藏匿”起来，最后问题暴露时，追查那些bug就变得又困难又耗时了。而通过在代码中嵌入断言，bug就会在其诞生之时对外宣布，那时最容易修正问题，因为程序员对代码的改动记忆犹新。

---

[26]译注：原文用“structured exception handling（SEH）”并不正确。因为SEH是微软视窗的专有技术而非C++功能。

[27]http://www.joelonsoftware.com/items/2003/10/13.html

[28]http://www.nedbatchelder.com/text/exceptions-vs-status.html

[29]http://www.joelonsoftware.com/items/2003/10/15.html

[30]译注：此著作年代久远，建议读者可参考同为微软作者的《代码大全（Code Complete）》第2版8.2节。

断言是以宏来实现的，所以如有需要，只需改写该宏，就能在代码中去除所有断言。在开发期间，断言引起的效能开销，通常可以忍受。游戏发行时，可以去除断言，取回那一点关键效能。

断言实现

断言通常实现为一个宏，宏会在if/else子句里对表达式求值，若断言失败（表达式求值为false），就会调用一个函数，并且使用一些汇编代码去暂停程序，如果程序于调试器里运行就会让调试器中断下来。以下是一个典型断言实现。

```
#if ASSERTIONS_ENABLED
    // 定义一个内联汇编让调试器暂停程序
    // 不同CPU的做法有所区别
    #define debugBreak() asm { int 3; }

    // 对运算式求值，若为伪值则报告断言失败
    #define ASSERT(expr) \
        if (expr) {} \
        else \
        { \
            reportAssertionFailure(#expr, __FILE__, __LINE__); \
            debugBreak(); \
        }
#else
    #define ASSERT(expr) // 不求值
#endif
```

让我们解构这个定义，分析其如何运作。

- 外层的#if/#else/#endif用于剥除代码里的断言。若ASSERTIONS_ENABLED是非零值，ASSERT()宏就被定义为真正的版本，程序中所有断言检查就会生效。若断言被关上，ASSERT(expr)不做任何事情，所有使用该宏的地方，实际上会被预处理器删除。
- debugBreak()宏定义为一些汇编语言指令，这些指令会暂停程序，并把控制权转交调试器（若在调试器执行该程序）。这些指令因CPU而异，但通常是单个汇编指令。
- ASSERT()宏本身是用完整的if/else语句定义的（而非单独的if）。这个方式使ASSERT()宏可以用在任何上下文中，就算放进没有花括号的if/else语句中，也不成问题。

  以下的例子，显示使用单独的if来定义ASSERT()所产生的问题。

```
#define ASSERT(expr) if (!(expr)) debugBreak()

void f()
{
    if (a < 5)
        ASSERT(a >= 0);
    else
        doSomething(a);
}
```

把宏展开后，会变成以下的错误代码。

```
void f()
{
    if (a < 5)
        if (!(a >= 0)) debugBreak();
    else // 哎呦！ 对应了错误的if()！
        doSomething(a);
}
```

- ASSERT()的else子句做了两件事情。第一，是向程序员显示一些消息，说明哪里出错；第二，就是在调试器里中断程序。注意第一个消息显示参数是#expr。符号（#）是预处理器操作符，使expr运算式转为一个字符串，这样就可以把该运算式作为消息的一部分打印出来了。
- 注意__FILE__和__LINE__。这两个编译器定义的宏，会神奇地表示出该宏的源文件名[31]和行号。把这两个宏传给消息显示函数，就可以打印问题的确切位置。

笔者十分推荐在代码中使用断言。然而，也必须注意断言的效能开销。可以考虑定义两种断言。惯常的ASSERT()宏在**所有**生成中保留，所以即使不在调试生成下也可以发现错误[32]。第二种断言宏，可能命名为SLOW_ASSERT()，只在调试生成中生效。显然SLOW_ASSERT()的效用较低，因为它们被排除于测试员每天玩的游戏版本之外。但至少这些断言在程序员调试中有效。

正确地使用断言极为重要。断言只应用于捕捉程序本身的bug，永远**不要**用来捕捉用户错误。另外，当断言失败，应该总是终止整个游戏程序。允许测试员、美术人员、设计师或其他非工程师去跳过断言，通常这是个坏点子。（这有点像“狼来了”的故事：若断言可以跳过，就会变得毫不重要，最终变成无效。）换言之，断言应该只用来捕捉严重错误。若可以在断言后继续正常运行程序，那么最好采用其他方式向用户报错，例如，在屏幕上显示消息，或是丑陋的亮橙色三维图形。关于正确使用断言的讨论，可参考此文[33]。

---

[31]译注：原文为.cpp文件，实际上也可以是头文件。

[32]译注：通常最终的交付版本会去掉断言。

[33]http://www.wholesalealgorithms.com/blog9

第4章　游戏所需的三维数学

游戏是在计算机上实时模拟虚拟世界的数学模型。因此，数学渗透游戏产业的各个环节。游戏程序员会用到几乎所有数学分支，如三角学、代数、统计学、微积分。然而，游戏程序员最常使用到的是三维矢量和矩阵（即三维**线性代数**/linear alegbra）。

仅此门数学分支已是既广且深了，我们不能期望在此章里涵盖非常深入的内容。相反，笔者会试述典型游戏程序员所需的数学工具，也会提供一些技巧和诀窍，帮助读者理解一些较易混淆的概念和规则。欲要深入了解游戏所需的三维数学，笔者强列推荐Eric Lengyel的著作[28]。

4.1　在二维中解决三维问题

本章中的许多数学运算同样适用于二维和三维。这对读者而言是个好消息，因为它说明解决有的三维矢量问题时，**有时候**能用二维方式去思考和绘图（这较三维方式容易得多）。可惜，二维和三维的解决方法并非在所有情况下都是等效的。一些运算，例如叉积（cross product），仅在三维中有定义，而一些问题也须同时考虑3个维度才有意义。尽管如此，用简化的二维方式去思考问题，通常倒也无伤大雅。理解二维解后，可以试着把问题扩展至三维。有时候，我们会欣然发现二维解在三维中也有效。在另一些情况下，我们可以找到某个坐标系，而该坐标系上的问题实际是二维的。如果某个问题对于二维和三维并无差别，本书会使用二维图解说明。

4.2　点和矢量

大部分现代3D游戏都是由虚拟世界里的三维物体组成的。游戏引擎须记录这些物体的位置（position）、定向（orientation）和比例（scale），不断改变这些属性以产生动画，并

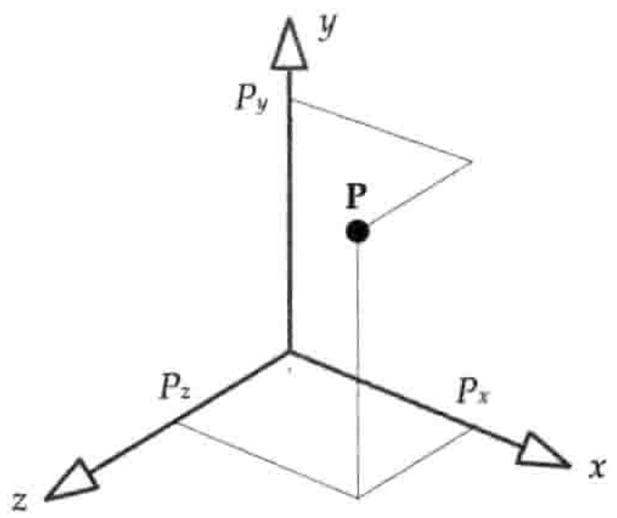
把这些属性变换（transform）至屏幕空间，使物体能渲染在屏幕上。在游戏中，三维物体几乎都是由三角形组成的，其中的三角形顶点（vertex）则以点（point）表示。所以在学习怎样表示游戏的整个对象之前，我们先来了解一下点，以及与之关系密切的矢量。

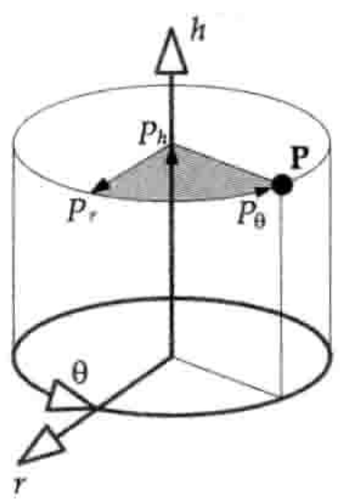
4.2.1 点和笛卡儿坐标

严格地说，**点**是n维空间里的一个位置（在游戏中，n通常等于2或3）。笛卡儿坐标系（Cartesian coordinate system）是游戏程序员最常用的坐标系，它使用2个或3个互相垂直的轴，来描述二维或三维空间的位置。因此，可以由2个或3个实数表示一个点**P**，即(P_x, P_y)或(P_x, P_y, P_z)，如图4.1所示。

当然，笛卡儿坐标系并非唯一选择。其他一些常用的坐标系如下。

- **圆柱坐标系**（cylindrical coordinate system）：此坐标系由3部分组成，分别是垂直“高度”轴h、从垂直轴发射出来的辐射轴r、和yaw角度θ。在圆柱坐标系中，以3个数字(P_h, P_r, P_θ)表示一个点**P**，如图4.2所示。
- **球坐标系**（spherical coordinate system）：此坐标系也是由3部分组成的，分别是俯仰角（pitch）phi（ϕ）、偏航角（yaw）theta（θ）和半径长度r。因此，以3个数字(P_r, P_ϕ, P_θ)去表示点，如图4.3所示。





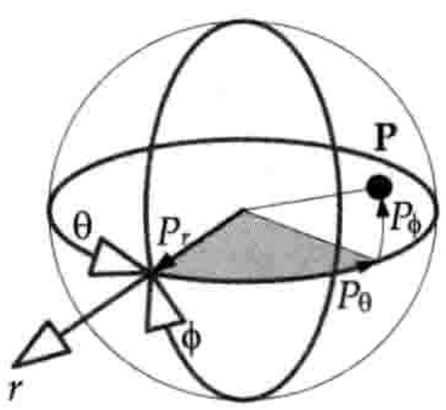


图 4.1: 以笛卡儿坐标表示的点。图 4.2: 以圆柱坐标表示的点。图 4.3: 以球坐标表示的点。

虽然笛卡儿坐标系是游戏编程中最广泛使用的坐标系，但必须谨记，应为当前问题选择最合适的坐标系。例如，在Midway公司的《Crank the Weasel》中，游戏主角Crank须在一个装饰派艺术风格的城市里四处跑，捡拾赃物。笔者希望那些物品像漩涡一样绕着Crank的身体旋转，越来越接近Crank并最终消失。笔者使用了圆柱坐标系表示物品的位置。要制作漩涡动画，只需简单地在θ上加入恒定角速率，在辐射轴r上加入少许向内的恒定线性速率，并在垂直轴h上加入少许向上的恒定线性速率，使物品缓缓向上升至Crank裤袋的水平位置。此简单动画非常好看，而且使用圆柱坐标系模拟比使用笛卡儿坐标系简单得多。

4.2.2 左手坐标系与右手坐标系的比较

在三维笛卡儿坐标中，要安排3个互相垂直的轴，我们有两个选择：右手（right-handed，RH）和左手（left-handed，LH）。要掌握右手坐标系3个轴的方向，可把右手握拳，伸出拇指指向x轴、食指指向y轴、中指指向z轴。左手坐标系则使用左手。

左右手坐标系的区别在于3个轴其中一个轴的方向不同。例如，若y轴指向上，x轴指向右，在右手坐标系中z轴会指向自己（走出页面），而左手坐标系中z轴则是远离自己（走进页面）。图4.4显示了左右手笛卡儿坐标系。

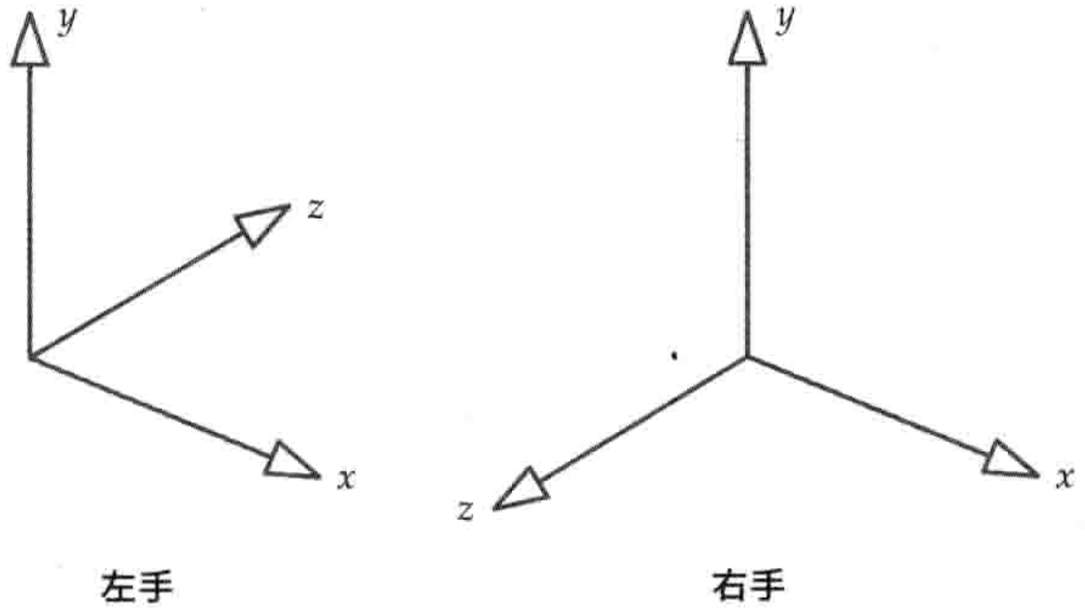


图 4.4: 左手与右手笛卡儿坐标系。

左右手坐标系相互转换十分容易。只需把其中一个轴反转，并保留另外两个轴不变即可。非常重要的是，数学法则在左手和右手坐标系里并不会改变，改变的只是我们如何把这些数字在脑海里**诠释**为三维空间。左手和右手约定只应用在可视化过程中，并不影响底层里的数学。（事实上，利手/handedness对物理模拟中的叉积有影响，但我们在大部分游戏编程中可忽略这些细微地方。详情可参阅维基百科[1]。）

如何从数值表示映射到视觉表示，完全由数学家和程序员决定。我们可选择以y轴向上、z轴向前、x轴向左（右手坐标系）或向右（左手坐标系）。或是，我们可选择z轴向上，又或是x轴向上或向下。至关重要的是确定映射方式后，便可以贯彻使用。

话虽如此，但在一些应用中，使用某些约定比其他约定更好。例如，三维图形程序员一般以左手坐标系工作，并以y轴向上、x轴向右、z轴向观察者离去（即虚拟摄像机指着的方向）。当三维图形以此坐标系渲染至二维屏幕时，z轴坐标增加意味着场景的**深度**增加（即与虚拟摄像机的距离增加）。在随后的章节里，会介绍此特性如何应用到z缓冲方案中，以解决深度遮挡。

[1] http://en.wikipedia.org/wiki/Pseudovector

4.2.3 矢量

矢量（vector）指n维空间中包含**模**（magnitude）和**方向**的量。矢量可绘画成**有向线段**，线段自一点（**尾**）延伸至另一点（**头**）。矢量和**标量**（scalar）（即普通的实数数值）比较，标量有模但没有方向。标量通常印刷为斜体（如v），而矢量则用粗体（如$\mathbf{v}$）。

三维矢量可以用3个标量(x, y, z)表示，如同点一样。点和矢量的区别实际上是很细微的。严格地说，矢量是**相对于**某已知点的偏移。一个矢量可移至三维空间中的任何位置，只要该矢量的方向和大小保持不变，无论在哪个位置，皆为同一个矢量。

矢量也可以用来表示点，只要把其尾固定在坐标系的原点（origin）。这些矢量有时候称为**位置矢量**（position vector）或**矢径**（radius vector）。对我们来说，可以把3个标量视为点或矢量，只要记住，**位置矢量**的尾固定于已选坐标系的原点便可。这意味着，数学上点和矢量使用时有微妙区别。或可以说，点是**绝对的**，而矢量是**相对的**。

大部分游戏程序员使用“矢量”一词来同时表示点（位置矢量）及矢量（线性代数中严谨意义上的矢量，即纯方向性的矢量）。多数三维数学程序库也使用“矢量”一词同时代表点及矢量。当两者不能混为一谈时，本书就采用“方向矢量”来区分。谨记，在心中必须清晰区分点和矢量（即使数学程序库不做区分）。在4.3.6.1节将提及，当把点和矢量转换成齐次坐标，与4×4矩阵一起操作时，点和矢量须以不同方式工作，所以混淆两者会导致出错。

4.2.3.1 笛卡儿基矢量

为方便起见，通常会按笛卡儿坐标的3个主轴去定义3个**正交单位矢量**（orthogonal unit vector）（即矢量间互相垂直，且每个矢量的长度等于1）。沿x轴的单位矢量一般记作$\mathbf{i}$，沿y轴的为$\mathbf{j}$，沿z轴的为$\mathbf{k}$。矢量$\mathbf{i}, \mathbf{j}, \mathbf{k}$有时候称为笛卡儿**基矢量**（basis vector）。

任何点或矢量都可以用3个标量（实数）与3个基矢量乘积之和来表示标量。例如$(5, 3, -2) = 5\mathbf{i} + 3\mathbf{j} - 2\mathbf{k}$。

4.2.4 矢量运算

多数标量运算也可应用至矢量，而矢量也有些特有的运算。

4.2.4.1 矢量和标量的乘法

矢量$\mathbf{a}$和标量s相乘，等于$\mathbf{a}$中的每个分量和s相乘：

$$s\mathbf{a} = (sa_x, sa_y, sa_z)$$

矢量和标量相乘，其效果为保留矢量的方向，同时缩放矢量的模，如图4.5所示。乘以−1则是把矢量的方向反转（头尾互换）。

每个轴上的缩放因子（scale factor）也可以不相等。此称为**非统一缩放**（nonuniform scaling），可表示为矢量和缩放矢量的**分量积**（component-wise product）。本书以⊗符号表示分量乘法。严格地说，这种两矢量间的特殊乘法称为**阿达马积**（Hadamard product）。这种乘法在游戏业界并不多见——实际上，非统一缩放是游戏中使用这种乘法的常见例子之一[2]：

$$\mathbf{s} \otimes \mathbf{a} = (s_x a_x, s_y a_y, s_z a_z) \tag{4.1}$$

我们在4.3.7.3节中将会见到，缩放矢量**s**其实只是3×3对角缩放矩阵**S**的紧凑表示方式。所以方程(4.1)的另一种写法为：

$$\mathbf{aS} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & s_z \end{bmatrix} = \begin{bmatrix} s_x a_x & s_y a_y & s_z a_z \end{bmatrix}$$

4.2.4.2 加法和减法

两个矢量**a**、**b**相加后会成为一个新矢量，该矢量的每个分量为**a**和**b**中每个对应**分量**之和。此运算可以用图形表示，把**a**的头连接至**b**的尾，那么它们之和就是一个由**a**的尾延伸至**b**的头的矢量：

$$\mathbf{a} + \mathbf{b} = [(a_x + b_x), (a_y + b_y), (a_z + b_z)]$$

矢量减法$\mathbf{a} - \mathbf{b}$其实等同**a**和$-\mathbf{b}$（即**b**以−1缩放，也就是反转**b**的方向）之和。这也对应一个矢量，其分量是**a**和**b**中每个相对分量之差：

$$\mathbf{a} - \mathbf{b} = [(a_x - b_x), (a_y - b_y), (a_z - b_z)]$$

矢量加法和减法如图4.6所示。

---

[2]译注：另一个常见的分量积在游戏中应用的例子是，在图形学中把两个RGB颜色（RGB三维空间的矢量）相乘。例如，光照射到物体表面，该材质会吸收光在R、G、B通道中能量的百分比。光的总反射量可以使用两者RGB颜色的分量积计算。

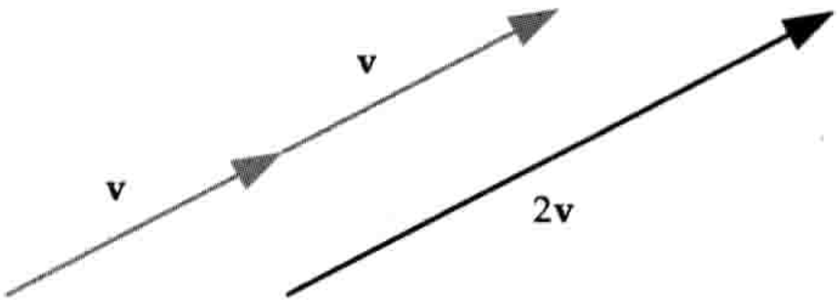


图 4.5: 一个矢量乘以标量2。

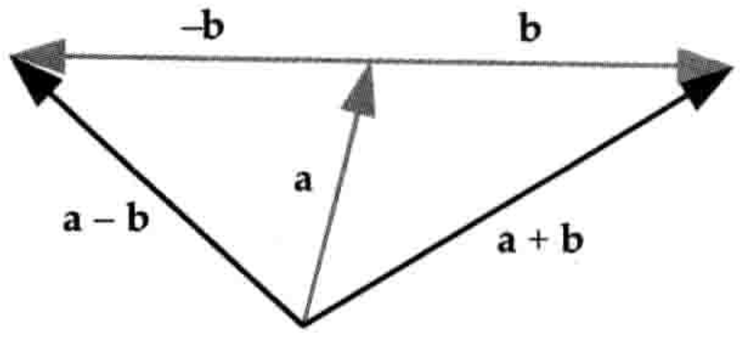


图 4.6: 矢量的加法和减法。

点和方向的加减

方向矢量可互相加减。然而严格地说，点和点不能互相加减，只可以把点和矢量相加，其结果为另一个点。类似地，点和点相减的结果是一个方向矢量。这些运算可以总结如下。

- 方向 + 方向 = 方向
- 方向 – 方向 = 方向
- 点 + 方向 = 点
- 点 – 点 = 方向
- 点 + 点 = **无意义**（不要这么做!）[3]

4.2.4.3 模

矢量的模（magnitude）是一个标量，代表矢量在二维或三维空间中的长度。它的写法是在粗体矢量两旁加上垂直线。可以利用勾股定理（Pythagorean theorem）去计算矢量的模，如图4.7所示。

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

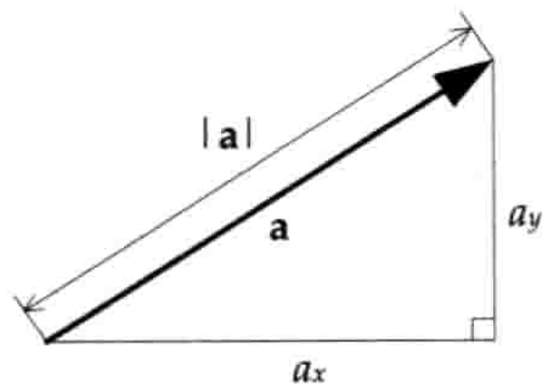


图 4.7: 矢量的模（二维）。

[3]译注：有一种针对点集的运算，称为闵可夫斯基和（Minkowski sum），它的定义中会对两个点（位置矢量）相加。此运算可用于一些碰撞检测算法（如12.3.5.5节谈及的GJK算法），也可用于组合各形状的采样，例如，线段上的点采样加上球体内的点采样会变成胶囊体内的点采样。关于闵可夫斯基和可参考维基百科条目http://en.wikipedia.org/wiki/Minkowski_sum。

4.2.4.4 矢量运算的实际应用

我们已经可以运用上述的矢量运算，来解决实际游戏开发中的问题。当尝试解决问题时，可对已知量使用加、减、缩放、模等运算产生新的数据。例如，假设某人工智能角色的现时位置为$\mathbf{P}_1$，其速度是矢量$\mathbf{v}$，则可以找到下一帧位置$\mathbf{P}_2$，方法是把$\mathbf{v}$以Δt缩放，再加上$\mathbf{P}_1$。如图4.8所示，结果等式为$\mathbf{P}_2 = \mathbf{P}_1 + (\Delta t)\mathbf{v}$。（此称为**显式欧拉法**/explicit Euler method，读者应该知道只有速度恒定时才有效。）

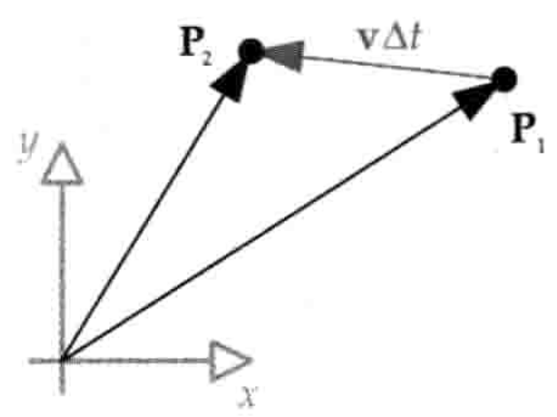


图 4.8: 简单的矢量加法，可用来从角色本帧的位置求出次帧的位置。

另一个例子，假设有两个球体，求两者是否相交。给定两球体的中心点为$\mathbf{C}_1$和$\mathbf{C}_2$，便可用减法计算两者之间的方向矢量，$\mathbf{d} = \mathbf{C}_2 - \mathbf{C}_1$。而此矢量的模$d = |\mathbf{d}|$表示两球体中心点的距离。若此距离小于两球体半径之和，则表示两球体相交；否则不相交，如图4.9所示。

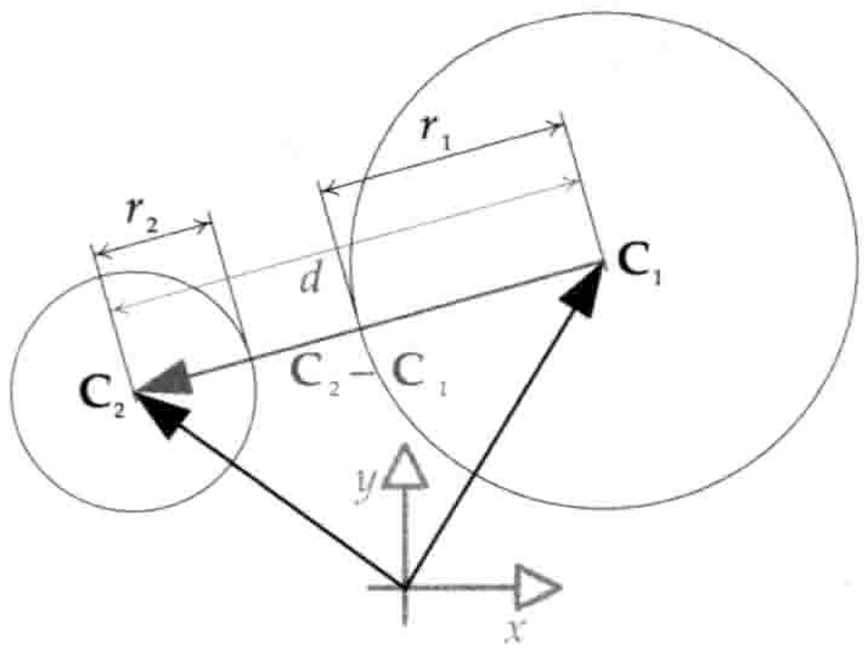


图 4.9: 球体对球体的相交测试只涉及矢量减法、矢量模和浮点比较运算。

然而，计算平方根在多数计算机上都是费时的运算，所以如果可以，游戏程序员应尽量改用**模的平方**：

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

使用模的平方有时候是有效的，例如，在比较两矢量的**相对长度**（“矢量$\mathbf{a}$是否比矢量$\mathbf{b}$长？”），或是比较一矢量的模和其他标量（的平方）时。同样以两球体的相交测试为例，只应计算$d^2 = |\mathbf{d}|^2$ 并比较此值与两球体半径之和的平方$(r_1 + r_2)^2$，运算速度才会最快。当

编写高效能软件时，不要计算非必需的平方根!

4.2.4.5 归一化和单位矢量

单位矢量（unit vector）即是模（长度）为1的矢量。单位矢量在三维数学和游戏编程中十分有用，稍后将解释其原因。

给定任何矢量$\mathbf{v}$的长度$v = |\mathbf{v}|$，可以把该矢量转换为单位矢量$\mathbf{u}$，使其保持$\mathbf{v}$的方向不变，长度变为单位长度。方法很简单，用$\mathbf{v}$乘以其模的倒数（reciprocal）。此过程又称为**归一化**（normalization）：

$$\mathbf{u} = \frac{\mathbf{v}}{|\mathbf{v}|} = \frac{1}{v}\mathbf{v}$$

4.2.4.6 法矢量

某表面（surface）的**法矢量**（normal vector）是指矢量**垂直**于该表面。法矢量在游戏和计算机图形学中非常有用。例如，一个**平面**（plane）可用一点和一个法矢量来定义[4]。在三维图形中，经常大量使用法矢量计算光线和材质表面之间的夹角。

法矢量一般为单位矢量，但此非必要条件。切记不要混淆归一化和法矢量两个术语[5]。归一化后的矢量是任何拥有单位长度的矢量；而法矢量是指垂直于材质表面的矢量，其模是否为单位长度并不重要。

4.2.4.7 点积和投影

矢量间可以相乘，但和标量不同，矢量有多种乘法。在游戏编程中，最常用的两种为：

- **点积**（dot product），又称为标量积（scalar product）或内积（inner product）。
- **叉积**（cross product），又称为矢量积（vector product）或外积（outer product）。

两矢量的点积结果是一个标量，此标量定义为两矢量中每对分量乘积之和：

$$\mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z = d \quad \text{(一个标量)}$$

---

[4]译注：事实上，定义一个三维平面最少需要4个实数，例如，使用$Ax + By + Cz + D = 0$方程定义一个平面，$\mathbf{n} = [A, B, C]$为平面的法矢量，若法矢量是单位矢量，则D为平面至原点的垂直距离（可为正或负）。此等式又可以写成$\mathbf{n} \cdot \mathbf{x} + D = 0$。4.6.3节会再介绍。

[5]译注：原文是指英文normalization和normal vector容易混淆。

点积也可以写成两矢量的模相乘后，再乘以两矢量间夹角的余弦：

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos\theta$$

点积符合**交换律**（commutative）（即两矢量的先后次序可互换），以及在加法上符合**分配律**（distributive）：

$$\mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a}$$

$$\mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c}$$

点积可结合标量乘法：

$$(s\mathbf{a}) \cdot \mathbf{b} = \mathbf{a} \cdot s\mathbf{b} = s(\mathbf{a} \cdot \mathbf{b})$$

矢量投影

若$\mathbf{u}$为单位矢量（$|\mathbf{u}| = 1$），则点积（$\mathbf{a} \cdot \mathbf{u}$）表示在由$\mathbf{u}$方向定义的无限长度直线上，$\mathbf{a}$的**投影**（projection）长度，如图4.10所示。此投影概念同样能应用至二维和三维，对解决各种各样的三维问题非常有用。

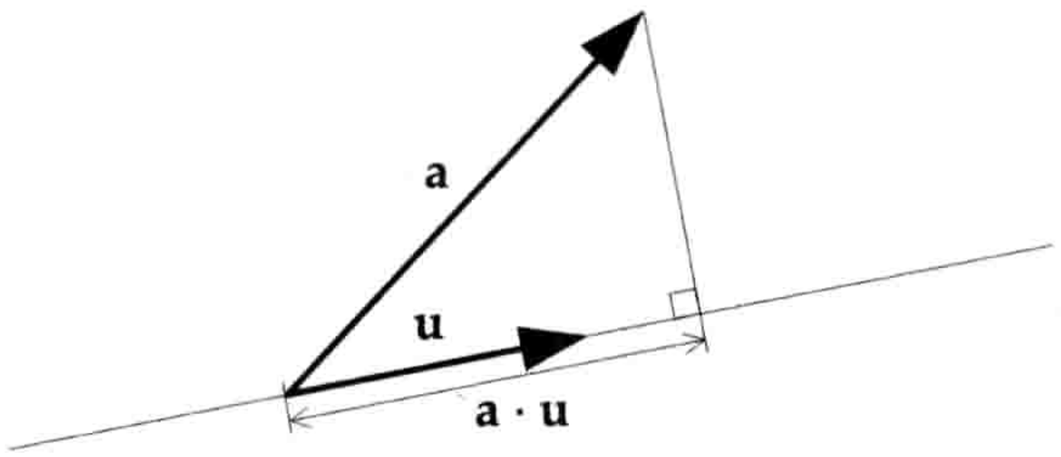


图 4.10: 使用点积计算矢量投影。

模作为点积

模的平方可以用矢量和自身的点积计算，而要计算矢量的模则可以将点积开平方：

$$|\mathbf{a}|^2 = \mathbf{a} \cdot \mathbf{a}$$

$$|\mathbf{a}| = \sqrt{\mathbf{a} \cdot \mathbf{a}}$$

这样可行是因为0°的余弦为1，点积算式便只余下$|\mathbf{a}||\mathbf{a}| = |\mathbf{a}|^2$。

点积判定（dot product test）

点积非常适合用来判断两矢量是否互共线（collinear）或垂直，或测试两矢量是否大致在相同或相反方向。对于任意两矢量**a**和**b**，游戏程序员经常使用以下判定，如图4.11所示。

- **共线：** $(\mathbf{a}\cdot\mathbf{b}) = |\mathbf{a}||\mathbf{b}| = ab$（即夹角精确地为0°。若**a**和**b**都是**单位矢量**且共线，则点积为+1）。
- **共线但相反方向：** $(\mathbf{a}\cdot\mathbf{b}) = -ab$（即夹角精确地为180°。若**a**和**b**都是单位矢量且共线，则点积为−1）。
- **垂直：** $(\mathbf{a}\cdot\mathbf{b}) = 0$（即夹角为90°）。
- **相同方向：** $(\mathbf{a}\cdot\mathbf{b}) > 0$（即夹角少于90°）。
- **相反方向：** $(\mathbf{a}\cdot\mathbf{b}) < 0$（即夹角多于90°）。

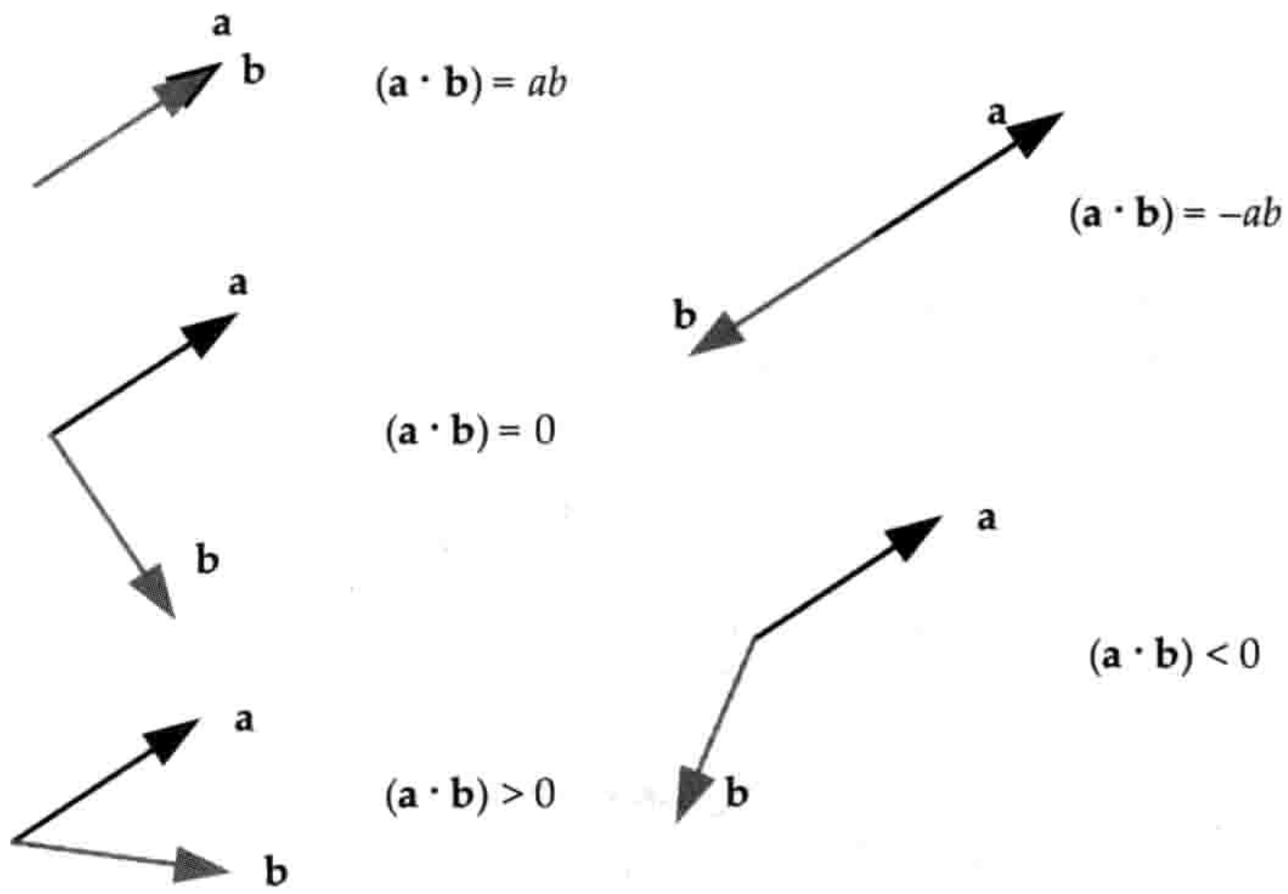


图 4.11: 一些常见的点积判定。

其他点积的应用

点积可应用在游戏编程中许多不同的问题上。例如，要得悉某个敌人是在玩家的前面还是后面，先用减法找出由玩家位置**P**至该敌人位置**E**的矢量（$\mathbf{v} = \mathbf{E} - \mathbf{P}$）。再假设玩家**面向**的方向为矢量**f**。（在4.3.10.3节会讲述，**f**可以从玩家的模型至世界矩阵中取得。）那么点积$d = \mathbf{v}\cdot\mathbf{f}$可以用来测试敌人在玩家前面还是后面，前面则点积为正，后面则点积为负。

点积也可以用来计算任意一点在某平面上方或下方的高度（例如，编写一个月球着陆游戏时可能有用）。我们可用两个矢量来定义一平面：平面上任意一点**Q**，以及与平面垂直

的单位矢量**n**（法矢量）。要得出**P**在该平面上的高度h，可先计算平面上任意点（**Q**就可以）至**P**的矢量，例如$\mathbf{v} = \mathbf{P} - \mathbf{Q}$。**v**和单位矢量**n**的点积，就是**v**在**n**方向直线上的投影，而这就是我们要找的高度。因此，$h = \mathbf{v} \cdot \mathbf{n} = (\mathbf{P} - \mathbf{Q}) \cdot \mathbf{n}$，如图4.12所示。[6]

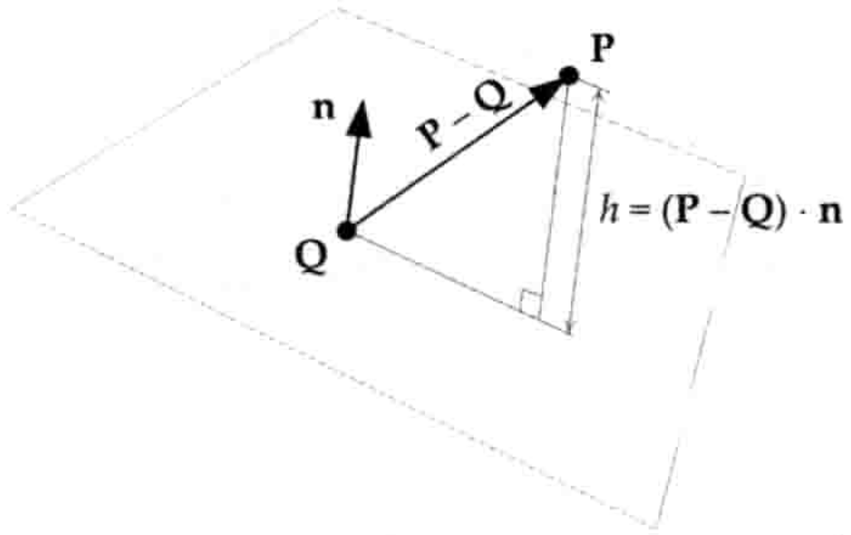


图 4.12: 点积可用于求一点高于或低于平面的高度。

4.2.4.8 叉积

两个矢量的**叉积**（也称为**外积**/outer product或**矢量积**/vector product）会产生另一个**矢量**，该矢量**垂直**于原来的两个相乘矢量，如图4.13所示。叉积运算只定义于三维空间。

$$\begin{aligned}\mathbf{a} \times \mathbf{b} &= [(a_y b_z - a_z b_y), (a_z b_x - a_x b_z), (a_x b_y - a_y b_x)] \\ &= (a_y b_z - a_z b_y)\mathbf{i} + (a_z b_x - a_x b_z)\mathbf{j} + (a_x b_y - a_y b_x)\mathbf{k}\end{aligned}$$

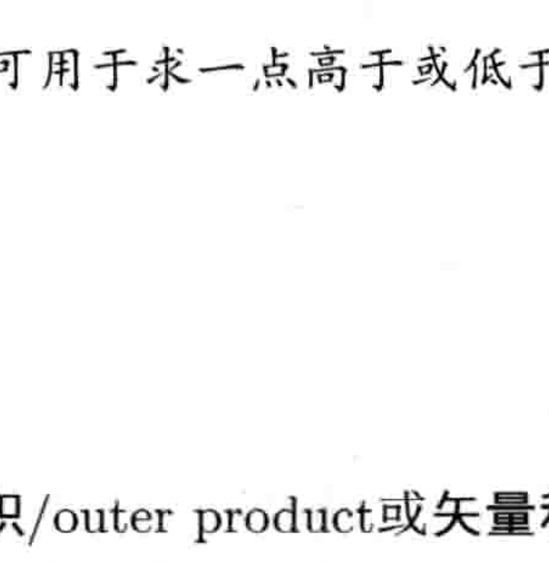


图 4.13: 矢量**a**和**b**的叉积（右手坐标系）。

[6]译注：若平面采用$\mathbf{n} \cdot \mathbf{x} + D = 0$形式来定义，则$h = \mathbf{n} \cdot \mathbf{P} + D$。

叉积的模

叉积的模等于两矢量各自的模的积再乘以两矢量夹角的正弦。（叉积的模和点积的模相似，只是用正弦取代余弦。）

$$|\mathbf{a} \times \mathbf{b}| = |\mathbf{a}||\mathbf{b}| \sin\theta$$

若**a**和**b**为平行四边形的两条边，其面积为两矢量叉积的模$|\mathbf{a} \times \mathbf{b}|$，如图4.14所示。由于三角形是平行四边形的一半，所以由$\mathbf{V}_1$、$\mathbf{V}_2$、$\mathbf{V}_3$顶点组成的三角形，其面积是任意两边的矢量叉积的一半[7]：

$$A_{\text{triangle}} = \tfrac{1}{2} |(\mathbf{V}_2 - \mathbf{V}_1) \times (\mathbf{V}_3 - \mathbf{V}_1)|$$

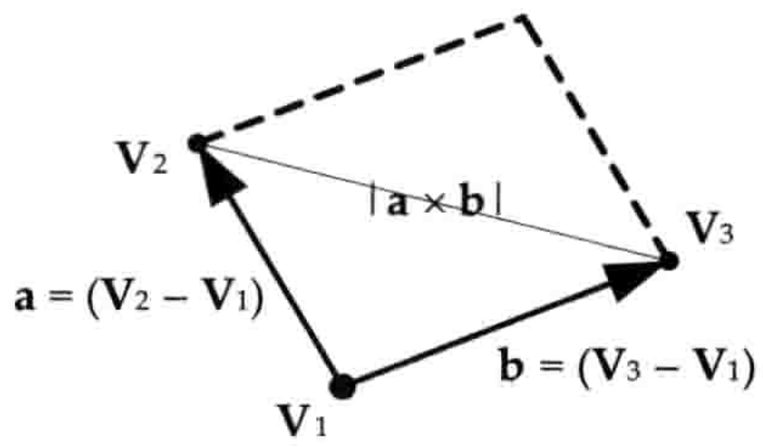


图 4.14: 以叉积的模来表示平行四边形的面积。

叉积的方向

当使用右手坐标系时，可以使用**右手法则**（right-hand rule）来表示叉积的方向。伸开右手掌，使除拇指以外的4只手指指向**a**矢量的方向，再把4只手指向内屈曲指向**b**矢量的方向，那么拇指的方向便是叉积（$\mathbf{a} \times \mathbf{b}$）的方向。

注意若使用左手坐标系，则叉积是用**左手法则**（left-hand rule）来定义的。这意味着，叉积的方向按选用的坐标系的改变而改变。开始可能会感到奇怪，但要记住，利手和数学计算并无关系，利手只影响如何使数字在三维空间中**视觉化**而已。当从右手坐标系转换为左手坐标系时，所有点和矢量的数字表达保持不变，只是当视觉化时一个轴变为相反方向。一切视觉化后会以反转轴形成镜像。因此，若一个叉积和该轴对齐（如z轴），视觉化后也会反转。若要它不反转，反而要修改数学定义去符合视觉化的结果。笔者不会为此而浪费时间，只要记住，当**视觉化**一个叉积时，右手坐标系使用右手法则，左手坐标系使用左手法则。

[7]译注：这里所说的平行四边形和三角形都是位于三维空间中的平面，形成边的矢量是三维的，否则叉积没有定义。

叉积的特性

叉积**不**符合**交换律**（即先后次序有影响）：

$$\mathbf{a} \times \mathbf{b} \neq \mathbf{b} \times \mathbf{a}$$

然而，叉积符合**反交换律**：

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

叉积在加法上符合分配律：

$$\mathbf{a} \times (\mathbf{b} + \mathbf{c}) = (\mathbf{a} \times \mathbf{b}) + (\mathbf{a} \times \mathbf{c})$$

叉积和标量乘法可如下结合：

$$(s\mathbf{a}) \times \mathbf{b} = \mathbf{a} \times (s\mathbf{b}) = s(\mathbf{a} \times \mathbf{b})$$

笛卡儿基矢量之间有以下叉积关系：

$$\mathbf{i} \times \mathbf{j} = -(\mathbf{j} \times \mathbf{i}) = \mathbf{k}$$
$$\mathbf{j} \times \mathbf{k} = -(\mathbf{k} \times \mathbf{j}) = \mathbf{i}$$
$$\mathbf{k} \times \mathbf{i} = -(\mathbf{i} \times \mathbf{k}) = \mathbf{j}$$

这3个叉积定义了绕笛卡儿轴的**正旋**（positive rotation）方向。正旋自x到y（绕z轴）、自y到z（绕x轴）、自z到x（绕y轴）。注意绕y轴旋转时，是按“反向”字母顺序自z到x（而非x到z）的。在下文中可以看到，这可以用来解释为何绕y轴的旋转矩阵，相对绕x、z轴的旋转矩阵而言，是**倒转**（inverted）的。

叉积的实际应用

叉积在游戏中有许多应用。最常见的是，用叉积来求垂直于两个矢量的矢量。我们将会在4.3.10.2节里看到，若有物体的本地的局部单位基矢量($\mathbf{i}_{\text{local}}, \mathbf{j}_{\text{local}}, \mathbf{k}_{\text{local}}$)，则可轻易建立一矩阵去表示该物体的定向。假设我们只知道物体的$\mathbf{k}_{\text{local}}$矢量，即物体面向的方向。若物体没有绕$\mathbf{k}_{\text{local}}$方向旋转，就可以用$\mathbf{k}_{\text{local}}$和世界空间向上矢量$\mathbf{j}_{\text{world}}$（即$[0,1,0]$）的叉积，去计算$\mathbf{i}_{\text{local}}$。方法是$\mathbf{i}_{\text{local}} = \text{normalize}(\mathbf{j}_{\text{world}} \times \mathbf{k}_{\text{local}})$。找$\mathbf{j}_{\text{local}}$只需找出$\mathbf{i}_{\text{local}}$和$\mathbf{k}_{\text{local}}$的叉积：$\mathbf{j}_{\text{local}} = \mathbf{k}_{\text{local}} \times \mathbf{i}_{\text{local}}$。

同样，叉积也可以用来求三角形表面或其他平面的法矢量。给定平面上任意3点$\mathbf{P}_1$、$\mathbf{P}_2$、$\mathbf{P}_3$，平面的法矢量就是$\mathbf{n} = \text{normalize}[(\mathbf{P}_2 - \mathbf{P}_1) \times (\mathbf{P}_3 - \mathbf{P}_1)]$。[8]

叉积也可应用在物理模拟中。当向一物体施加力（force），当且仅当其施力方向离开中心点时，该力会对物体的旋转运动产生影响。由此产生的旋转力称为**力矩**（torque），其计算方法如下：给定力$\mathbf{F}$、从质心（center of mass）至施力点的矢量$\mathbf{r}$，则产生的力矩为$\mathbf{N} = \mathbf{r} \times \mathbf{F}$。

4.2.5 点和矢量的线性插值

游戏中，时常要找两个已知矢量之间的矢量。例如，要在2秒内，以每秒30帧的速度，用动画形式顺滑地把物体从**A**点移动至**B**点，那么须计算**A**和**B**之间60个中间点（intermediate point）。

线性插值（linear interpolation）是一个简单的数学运算，用来计算两个已知点的中间点。此运算的名称通常简写成LERP。此运算定义如下，其中β介于并包含0～1：

$$\begin{aligned}\mathbf{L} = \mathbf{LERP}(\mathbf{A}, \mathbf{B}, \beta) &= (1-\beta)\mathbf{A} + \beta\mathbf{B} \\ &= [(1-\beta)A_x + \beta B_x, (1-\beta)A_y + \beta B_y, (1-\beta)A_z + \beta B_z]\end{aligned}$$

从几何上看，$\mathbf{L} = \mathbf{LERP}(\mathbf{A}, \mathbf{B}, \beta)$为**AB**线段间一点的位置矢量，该点距**A**点β百分比的位置，如图4.15所示。数学上，**LERP**函数只是两矢量的**加权平均**（weighted average），两矢量的权重分别为$(1-\beta)$和β。注意权重之和为1，此乃任何加权平均的一般要求。

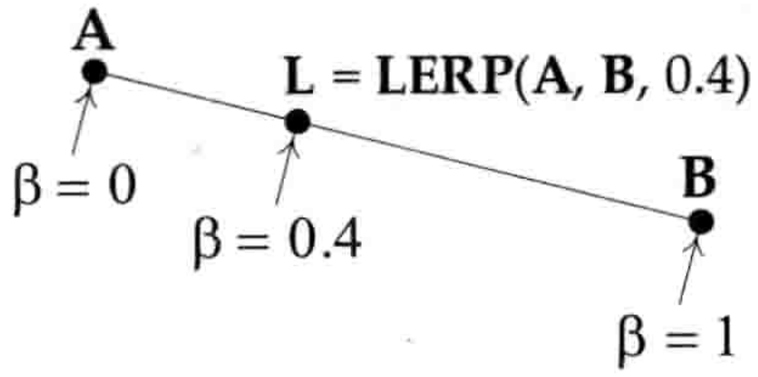


图 4.15: 对点**A**和点**B**进行线性插值（LERP），$\beta = 0.4$。

[8]译注：由于叉积不符合交换律，3点的次序会影响得出的法线方向（两个相反的方向）。这种次序称为缠绕顺序（winding order），详见10.1.1.3节。

4.3 矩阵

矩阵（matrix）是由$m \times n$个标量组成的长方形数组。矩阵便于表示线性变换（transformation），如平移（translation）、旋转（rotation）和缩放（scaling）。

矩阵$\mathbf{M}$通常写成由标量M_{rc}组成的栅格，以方括号包裹，其中下标r和c代表该项的行（row）和列（column）。例如，若$\mathbf{M}$是3×3矩阵，可写成：

$$\mathbf{M} = \begin{bmatrix} M_{11} & M_{12} & M_{13} \\ M_{21} & M_{22} & M_{23} \\ M_{31} & M_{32} & M_{33} \end{bmatrix}$$

我们可以视3×3矩阵的行和列为三维矢量。若某3×3矩阵中的所有行及列矢量为单位矢量，则该矩阵称为**特殊正交矩阵**（special orthogonal matrix）、**各向同性矩阵**（isotropic matrix）或**标准正交矩阵**（orthonormal matrix）。这种矩阵表示纯旋转。

在某些条件下，4×4矩阵可表示任意三维**变换**，包括**平移**、**旋转**和**缩放**。这种矩阵称为变换矩阵，对于身为游戏工程师的我们最为有用。利用矩阵乘法可以把表示为矩阵的变换，施于点或矢量。以下会介绍这些变换如何实现。

仿射矩阵（affine matrix）是一种4×4变换矩阵，它能维持直线在变换前后的平行性以及相对的距离比，但是不一定维持直线在变换前后的绝对长度及角度。由平移、旋转、缩放及/或切变（shear）所组合而成的变换都是仿射矩阵。

4.3.1 矩阵乘法

两矩阵$\mathbf{A}$和$\mathbf{B}$的积$\mathbf{P}$写作$\mathbf{P} = \mathbf{AB}$。若$\mathbf{A}$和$\mathbf{B}$为变换矩阵，则其积$\mathbf{P}$也是变换矩阵，而且以$\mathbf{P}$进行变换时，等同于进行$\mathbf{A}$和$\mathbf{B}$**两者**的变换。例如，若$\mathbf{A}$为缩放矩阵，$\mathbf{B}$为旋转矩阵，则矩阵$\mathbf{P}$能对点或矢量进行缩放**和**旋转变换。此特性对游戏编程特别有用，因为我们可以把一连串变换预先计算为单一矩阵，再用该矩阵高效地变换大批矢量。

要计算矩阵的积，只需简单地把$n_A \times m_A$矩阵$\mathbf{A}$的行，与$n_B \times m_B$ 矩阵$\mathbf{B}$的列进行点积计算。每个点积就是新生成矩阵中的元素。仅当两矩阵的**内维**（inner dimension）相等时（即$m_A = n_B$），两矩阵才可相乘。例如，当$\mathbf{A}$和$\mathbf{B}$都是3×3矩阵，则：

$$\mathbf{P} = \mathbf{AB}$$

$$P_{11} = \mathbf{A}_{\text{row1}} \cdot \mathbf{B}_{\text{col1}} \quad P_{12} = \mathbf{A}_{\text{row1}} \cdot \mathbf{B}_{\text{col2}} \quad P_{13} = \mathbf{A}_{\text{row1}} \cdot \mathbf{B}_{\text{col3}}$$
$$P_{21} = \mathbf{A}_{\text{row2}} \cdot \mathbf{B}_{\text{col1}} \quad P_{22} = \mathbf{A}_{\text{row2}} \cdot \mathbf{B}_{\text{col2}} \quad P_{23} = \mathbf{A}_{\text{row2}} \cdot \mathbf{B}_{\text{col3}}$$
$$P_{31} = \mathbf{A}_{\text{row3}} \cdot \mathbf{B}_{\text{col1}} \quad P_{32} = \mathbf{A}_{\text{row3}} \cdot \mathbf{B}_{\text{col2}} \quad P_{33} = \mathbf{A}_{\text{row3}} \cdot \mathbf{B}_{\text{col3}}$$

矩阵乘法并不符合交换律。换句话说，矩阵乘法的次序会影响结果：

$$\mathbf{AB} \neq \mathbf{BA}$$

在4.3.2节里，我们会探讨为何矩阵乘法次序会影响结果。

矩阵乘法有时称为**串接**（concatenation），因为n个矩阵的积是一个矩阵，此矩阵把原来的变换按矩阵相乘的次序串接起来。

4.3.2 以矩阵表示点和矢量

点和矢量都可以表示为**行矩阵**（row matrix）（$1 \times n$）或**列矩阵**（column matrix）（$n \times 1$），其中n为使用中的空间维度（通常是2或3）。例如，矢量$\mathbf{v} = (3, 4, -1)$可写成：

$$\mathbf{v}_1 = \begin{bmatrix} 3 & 4 & -1 \end{bmatrix}$$

或

$$\mathbf{v}_2 = \begin{bmatrix} 3 \\ 4 \\ -1 \end{bmatrix} = \mathbf{v}_1^{\mathrm{T}}$$

本来是可以任意选择行矢量或列矢量的，但此选择会影响矩阵乘法的书写次序。原因是进行矩阵乘法时，两矩阵的内部维数必须相等，所以：

- 要把$1 \times n$行矢量乘以$n \times n$ 矩阵，矢量必须置于矩阵的**左方**（$\mathbf{v}'_{1\times n} = \mathbf{v}_{1\times n}\mathbf{M}_{n\times n}$）。
- 而要把$n \times n$矩阵乘以$n \times 1$ 列矢量，矢量必须置于矩阵的**右方**（$\mathbf{v}'_{n\times 1} = \mathbf{M}_{n\times n}\mathbf{v}_{n\times 1}$）。

当多个变换矩阵**A**、**B**和**C**顺序施于矢量**v**，如使用**行**矢量则变换从**左至右**“阅读”，而使用**列**矢量则变换从**右至左**“阅读”。最容易的记忆方法是，**最接近**矢量的矩阵会最先进行变换。以下用括号显示这个关系：

$$\mathbf{v}' = (((\mathbf{v}\mathbf{A})\mathbf{B})\mathbf{C}) \qquad \text{行矩阵，从左至右阅读}$$
$$\mathbf{v}' = (\mathbf{C}(\mathbf{B}(\mathbf{A}\mathbf{v}))) \qquad \text{列矩阵，从右至左阅读}$$

本书采用**行矢量惯例**，因为从左至右的变换次序最符合说英语人群的阅读习惯[9]。话虽如此，对使用中的游戏引擎，或是阅读的书籍、文献、网页而言，必须小心检查它们采用哪个惯例。通常只要看矢量矩阵相乘时，矢量位于左方（行矢量）还是右方（列矢量）。当使用列矢量时，则须把本书所示的矢量做一次**转置**（transpose）才可使用。

4.3.3 单位矩阵

单位矩阵（identity matrix）是指，它乘以任何其他矩阵，都会得出和原来一样的矩阵。单位矩阵通常写作**I**。单位矩阵是正方形矩阵，对角线上的元素皆为1，其他元素为0：

$$\mathbf{I}_{3\times 3} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{AI} = \mathbf{IA} = \mathbf{A}$$

4.3.4 逆矩阵

矩阵**A**的**逆矩阵**（inverse matrix）（写作$\mathbf{A}^{-1}$）能**还原**矩阵**A**的变换。所以，若**A**把物体绕z轴旋转37°，则$\mathbf{A}^{-1}$会绕z轴旋转$-37°$。同样，若**A**把物体放大为原来的两倍，则$\mathbf{A}^{-1}$会把物体缩小为一半大小。若一个矩阵乘以它的逆矩阵，结果**必然**是单位矩阵，因此$\mathbf{A}(\mathbf{A}^{-1}) \equiv (\mathbf{A}^{-1})\mathbf{A} \equiv \mathbf{I}$。并非所有矩阵都有逆矩阵。然而，所有**仿射**矩阵（纯平移、旋转、缩放及切变的组合）都有逆矩阵。若矩阵的逆矩阵存在，则可用高斯消去法（Gaussian elimination）或LU分解（LU decomposition）求之。

由于我们大量使用矩阵乘法，所以要特别注意矩阵串接后求逆，这相当于**反向串接**各个矩阵的逆矩阵。例如：

$$(\mathbf{ABC})^{-1} = \mathbf{C}^{-1}\mathbf{B}^{-1}\mathbf{A}^{-1}$$

4.3.5 转置矩阵

矩阵**M**的**转置**（transpose）写作$\mathbf{M}^{\mathrm{T}}$。转置矩阵就是把原来矩阵以主对角线（diagonal）为对称轴做反射。换句话说，原来矩阵的行变成转置矩阵的列，反之亦然：

[9] 译注：对汉语人群亦然。

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

基于以下两个原因，转置矩阵很实用。首先，标准正交矩阵（纯旋转）的逆矩阵和转置矩阵是一样的——此特性非常好，因为计算转置矩阵比计算一般逆矩阵快得多；其次，当把数据从一个数学程序库送到另一个程序库时，转置矩阵也十分重要，因为有些库使用列矢量，有些则使用行矢量。对于基于行矢量的库和基于列矢量的库，两者的矩阵是**转置**关系。

和逆矩阵相同，矩阵串接的转置，为反向串接各个矩阵的转置。例如：

$$(\mathbf{ABC})^{\mathrm{T}} = \mathbf{C}^{\mathrm{T}}\mathbf{B}^{\mathrm{T}}\mathbf{A}^{\mathrm{T}}$$

当需要考虑矩阵怎样对点和矢量进行变换时，此等式就会显得有用。

4.3.6 齐次坐标

读者可能有印象，高中代数里谈及2×2矩阵可以用来表示二维中的旋转。要把矢量$\mathbf{r}$旋转ϕ度（正旋是反时针方向的），可以写作：

$$\begin{bmatrix} r'_x & r'_y \end{bmatrix} = \begin{bmatrix} r_x & r_y \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi \\ -\sin\phi & \cos\phi \end{bmatrix}$$

同样，三维中的旋转也可以用3×3矩阵表示。以上的二维例子其实就是三维中绕z轴的旋转，因此可写成：

$$\begin{bmatrix} r'_x & r'_y & r'_z \end{bmatrix} = \begin{bmatrix} r_x & r_y & r_z \end{bmatrix} \begin{bmatrix} \cos\phi & \sin\phi & 0 \\ -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

由此会引发问题——3×3矩阵是否能表示**平移**？可惜，答案是否定的。把一点$\mathbf{r}$平移$\mathbf{t}$需要对$\mathbf{t}$和$\mathbf{r}$的分量分别求和：

$$\mathbf{r} + \mathbf{t} = \begin{bmatrix} (r_x + t_x) & (r_y + t_y) & (r_z + t_z) \end{bmatrix}$$

矩阵乘法涉及对元素的相乘和相加，因此用矩阵乘法进行平移的想法貌似可行。然而，并没有办法可以把**t**的分量放置在3×3矩阵里，使其与矢量**r**相乘后产生如$(r_x + t_x)$的和。

好消息是，若采用4×4矩阵，则可以获得类似这种的和。此矩阵是何种样式？由于不需要旋转，所以左上的3×3部分应为单位矩阵。若把**t**置于末行，并把**r**的第4个元素（通常称为w）设为1，则**r**和矩阵首列的点积为$(1 \times r_x) + (0 \times r_y) + (0 \times r_z) + (t_x \times 1) = (r_x + t_x)$，刚好就是我们所需要的。若矩阵右下角的元素为1，而第4列的其他元素为0，那么相乘结果的w分量也为1。以下就是4×4平移矩阵的样式：

$$\begin{aligned}\mathbf{r}+\mathbf{t} &= \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} \\ &= \begin{bmatrix} (r_x + t_x) & (r_y + t_y) & (r_z + t_z) & 1 \end{bmatrix}\end{aligned}$$

当点或矢量从三维延伸至四维，便称为**齐次坐标**（homogeneous coordinates）。在游戏引擎中，大多数三维矩阵都采用4×4矩阵，与4元素的齐次坐标点或矢量进行运算。

4.3.6.1 变换方向矢量

在数学上，点（位置矢量）和方向矢量的处理方法有细微差异。当用矩阵变换一个点时，平移、旋转、缩放都会施于该点上。但是，当用矩阵变换一个方向矢量时，就要忽略矩阵的**平移**效果。因为方向矢量本身并无平移，加上平移会改变其模，这并非我们所要的。

在齐次坐标中，可以把点的w分量设为1，而把方向矢量的w分量设为**0**。以下的例子显示，矢量**v**中的$w = 0$，因此乘以矩阵的**t**矢量后，可在结果中消去平移的作用：

$$\begin{bmatrix} \mathbf{v} & 0 \end{bmatrix} \begin{bmatrix} \mathbf{U} & \mathbf{0} \\ \mathbf{t} & 1 \end{bmatrix} = \begin{bmatrix} (\mathbf{vU} + 0\mathbf{t}) & 0 \end{bmatrix} = \begin{bmatrix} \mathbf{vU} & 0 \end{bmatrix}$$

严格地说，（四维的）齐次坐标转换成为（三维的）非齐次坐标的方法是，把x、y、z分量除以w分量：

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \equiv \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix}$$

此公式表明，可设点的w分量为1，方向矢量的w分量为0。矢量除以$w = 1$，并不影响点的坐标；但矢量除以$w = 0$则会产生无穷大（infinity）。四维中位于无穷远的一点，可以旋转但不可以平移，因为无论怎样平移，该点还是位于无穷远。所以事实上，三维空间的纯方向

矢量，在四维齐次空间是位于无穷远的点。

4.3.7 基础变换矩阵

任何仿射变换矩阵都能由一连串表示纯平移、纯旋转、纯缩放及/或纯切变的4×4矩阵串接而成。以下逐一介绍这些基础变换矩阵。（下文略去切变，因为游戏中极少使用。）

注意4×4变换矩阵可切割为4个组成部分：

$$\begin{bmatrix} \mathbf{U}_{3\times3} & \mathbf{0}_{3\times1} \\ \mathbf{t}_{1\times3} & 1 \end{bmatrix}$$

- 左上的3×3矩阵$\mathbf{U}$，代表旋转及/或缩放。
- 1×3平移矢量$\mathbf{t}$。
- 3×1零矢量$\mathbf{0} = [0 \quad 0 \quad 0]^{\mathrm{T}}$。
- 矩阵右下角的标量1。

当一点乘以如此切割的矩阵时，结果会是：

$$\begin{bmatrix} \mathbf{r}'_{1\times3} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{r}_{1\times3} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{U}_{3\times3} & \mathbf{0}_{3\times1} \\ \mathbf{t}_{1\times3} & 1 \end{bmatrix} = \begin{bmatrix} (\mathbf{rU}+\mathbf{t}) & 1 \end{bmatrix}$$

4.3.7.1 平移

以下的矩阵能把一点向$\mathbf{t}$矢量方向平移：

$$\mathbf{r}+\mathbf{t} = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix} = \begin{bmatrix} (r_x+t_x) & (r_y+t_y) & (r_z+t_z) & 1 \end{bmatrix}$$

或可写成切割后的缩写：

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & \mathbf{0} \\ \mathbf{t} & 1 \end{bmatrix} = \begin{bmatrix} (\mathbf{r}+\mathbf{t}) & 1 \end{bmatrix}$$

为求纯平移变换矩阵的逆矩阵，只需把$\mathbf{t}$求反（negate）（即分别反转t_x、t_y及t_z的正负号）。

4.3.7.2 旋转

所有4×4纯旋转变换矩阵都是以下的形式：

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{R} & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{rR} & 1 \end{bmatrix}$$

矢量$\mathbf{t}$为0，而左上的3×3矩阵$\mathbf{R}$则包含旋转角度（弧度单位）的余弦和正弦。

以下矩阵代表绕x轴旋转角度ϕ：

$$\text{rotate}_x(\mathbf{r}, \phi) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & \sin\phi & 0 \\ 0 & -\sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

以下矩阵代表绕y轴旋转角度θ。注意，相对其余两个旋转矩阵，此矩阵是**转置**的——两个正负正弦是依主轴反射的：

$$\text{rotate}_y(\mathbf{r}, \theta) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

以下矩阵代表绕z轴旋转角度γ：

$$\text{rotate}_z(\mathbf{r}, \gamma) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

从这些矩阵中我们可观察到：

- 左上3×3矩阵中的1必然位于旋转轴上，正弦和余弦项则在轴以外。
- 正旋是自x至y（绕z轴）、自y至z（绕x轴）、自z至x（绕y轴）。因为z至x是“绕回去”了，所以绕y轴的旋转矩阵相对于其他两个是转置的。（可用右手或左手法则去记忆这点。）

- 纯旋转矩阵的逆矩阵，即是该旋转矩阵的转置矩阵。这是因为旋转的逆变换等同于用反向角度旋转，并且$\cos -\theta = \cos\theta$及$\sin -\theta = -\sin\theta$，所以把角度求反就等于把两个正弦项求反，余弦项则维持不变。

4.3.7.3 缩放

以下的矩阵缩放点$\mathbf{r}$，向x轴的缩放因子为s_x，向y轴的为s_y，向z轴的为s_z：

$$\begin{aligned}\mathbf{rS} &= \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\ &= \begin{bmatrix} s_x r_x & s_y r_y & s_z r_z & 1 \end{bmatrix}\end{aligned}$$

或可写成切割后的缩写：

$$\begin{bmatrix} \mathbf{r} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{S}_{3\times3} & \mathbf{0} \\ \mathbf{0} & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{rS}_{3\times3} & 1 \end{bmatrix}$$

从这种矩阵中我们可观察到：

- 把矩阵求逆，只需把s_x、s_y、s_z用其倒数代替（即$1/s_x$、$1/s_y$、$1/s_z$）。
- 当3个轴的缩放因子相等（$s_x = s_y = s_z$），此变换称为**统一缩放**（uniform scale）。球体在统一缩放后仍然是球体，若使用非统一缩放，结果则会变成椭球（ellipsoid）。为了保证包围球（bounding sphere）检测的数学运算能够简单快捷，许多游戏引擎都加上限制，只容许对渲染用的几何物体和碰撞图元使用统一缩放。
- 当把一个统一缩放矩阵$\mathbf{S}_u$和一个旋转矩阵$\mathbf{R}$串接，相乘的次序并不重要（即$\mathbf{S}_u\mathbf{R} = \mathbf{RS}_u$）。只有**统一**缩放才有这个特性！

4.3.8 4×3矩阵

4×4仿射矩阵的最右侧必然是一列$[0 \quad 0 \quad 0 \quad 1]^{\mathrm{T}}$的矢量。因此，游戏程序员可略去第4列，以节省内存。游戏数学库里经常会遇到4×3仿射矩阵[10]。

[10]译注：在使用GPU做蒙皮（skinning）时，要向顶点着色器（vertex shader）传递大量的变换，所以为节省空间、时间，通常会使用3×4的矩阵。

4.3.9 坐标空间

我们已经知道如何用4 × 4矩阵，把变换施于点和方向矢量。此概念可以延伸至刚体（rigid body），只需把物体当作无限个点。把变换施于物体，就如同把该变换施于物体里的每一点。例如，在计算机图形学里，物体通常由三角形网格表示，每个三角形的3个顶点是由点去表示的。在此情况下，只要把变换矩阵施于所有的顶点，就等于把物体变换了。

之前提及，所谓一个点，即是一个矢量把其尾置于某坐标系的原点。换句话说，一个点（位置矢量）是必须表示为**相对**于某组坐标轴的。若选择不同的坐标轴组，代表点的3个数字也随之改变。如图4.16所示，一点P可由两个不同的位置矢量来表示——矢量$\mathbf{P}_A$代表相对“A”轴组的点P位置，而矢量$\mathbf{P}_B$则代表相对“B”轴组的同一点。

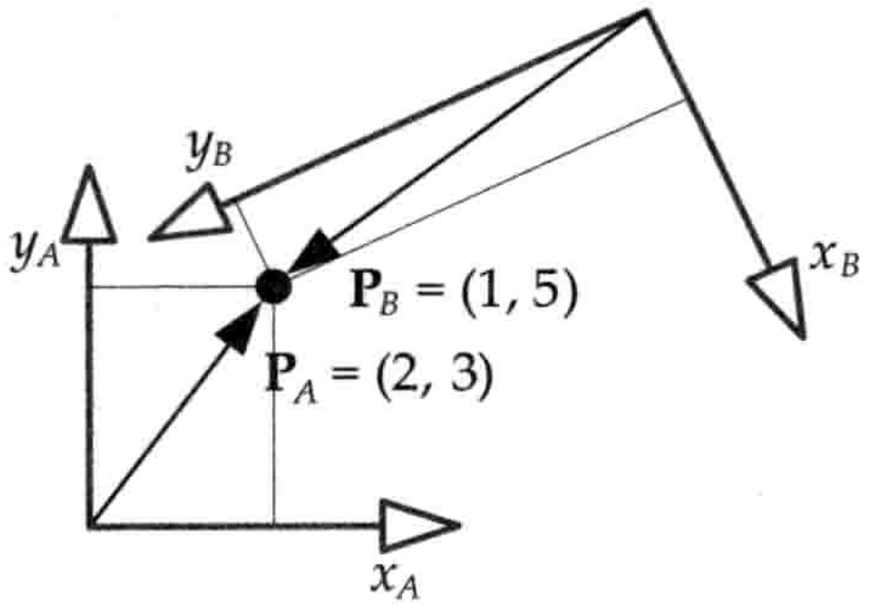


图 4.16: 点**P**对于不同坐标轴的位置矢量。

在物理学上，一组坐标轴代表一个参考系（frame of reference），所以有时候又会称一组轴为**坐标系**（coordinate frame，或简称为frame）。游戏业界则会使用**坐标空间**（coordinate space）一词，或简称**空间**（space），来表示一组坐标轴。以下将讨论游戏和计算机图形学中几个最常用的坐标空间。

4.3.9.1 模型空间

当使用Maya或3ds Max之类的工具去建立三角形网格，三角形顶点的位置是相对于一个笛卡儿坐标系的，我们称此坐标系为**模型空间**（model space），也可称为**物体空间**（object space）或**局部空间**（local space）。模型空间的原点可置于物体的中心位置，如物体的质心（center of mass），对于人形及动物角色，则可把模型空间的原点置于足部和地面之间。

多数游戏对象都有先天的定向性。例如，飞机的机头、垂直尾翼和机翼，可分别对应向前、向上，以及向左或向右。模型空间的轴通常会对准模型的自然方向，并会以直觉的标签

为这些轴命名，如图4.17所示。

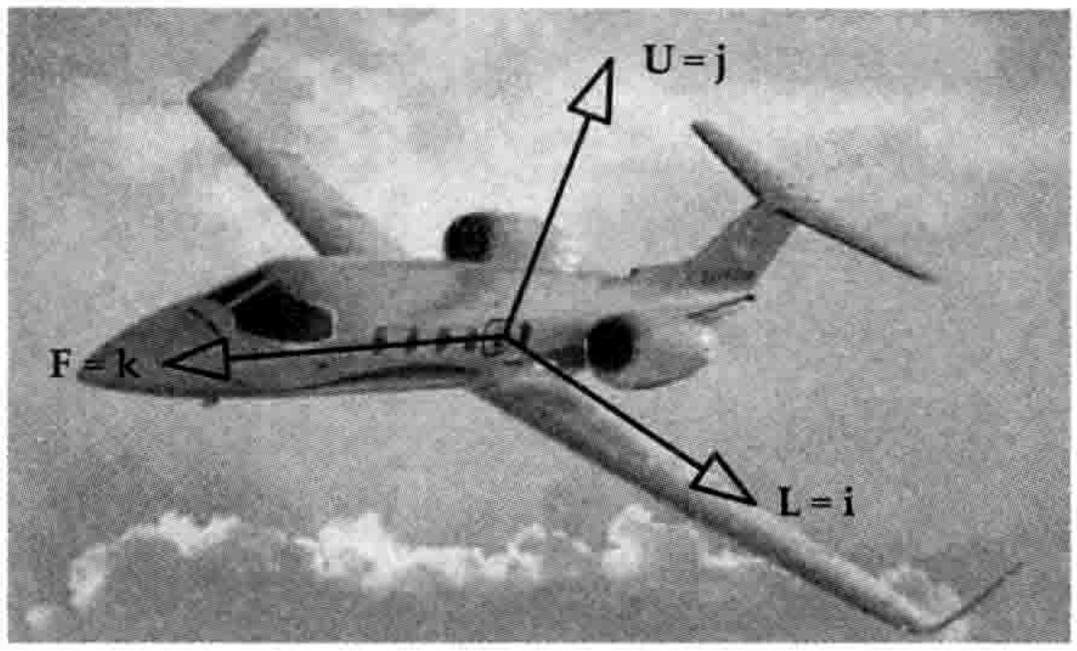


图 4.17: 对于一架飞机，这是模型空间前、左、上轴基矢量的可行选择之一。

- **向前**（front）：物体正常移动或朝向的方向，其轴称为向前轴。本书采用符号**F**代表前方轴的单位矢量。
- **向上**（up）：物体向上的轴称向上轴。本书采用符号**U**代表上方轴的单位矢量。
- **向左或向右**（left/right）：与物体的左边和右边对齐的轴分别称为向左轴和向右轴。使用哪个轴取决于游戏引擎是采用左手还是右手坐标系的。本书采用符号**L**和**R**分别代表这两个轴的单位矢量。

（向前、向上、向左）标签和(x, y, z)轴的映射完全是随心所欲的。使用右手坐标轴的时候，常见的映射是把**向前**对应正z轴、**向左**对应正x轴、**向上**对应正y轴（或以单元基矢量表示，$\mathbf{F} = \mathbf{k}$、$\mathbf{L} = \mathbf{i}$、$\mathbf{U} = \mathbf{j}$）。然而，同样常见的是$+x$代表**向前**、$+z$代表向右（$\mathbf{F} = \mathbf{i}$、$\mathbf{R} = \mathbf{k}$、$\mathbf{U} = \mathbf{j}$）。笔者也曾在工作上使用某些游戏引擎，采用z轴为垂直方向。对游戏引擎唯一的要求是贯彻使用统一协定。

使用合乎直觉的轴名称可减少混淆。比方说，3个**欧拉角**（Euler angle）——俯仰角（pitch）、偏航角（yaw）、滚动角（roll）——经常用来表示飞机的定向。可是，并不可能按照$(\mathbf{i}, \mathbf{j}, \mathbf{k})$基矢量去定义偏航角、俯仰角、滚动角，因为这些基矢量的方向是随意的。然而，我们**可以**按照$(\mathbf{L}, \mathbf{U}, \mathbf{F})$基矢量去定义偏航角、俯仰角、滚动角，因为这些基矢量的方向有清晰定义。确切地说，就是：

- **俯仰角**（pitch）是绕**L**或**R**旋转的角度；
- **偏航角**（yaw）是绕**U**旋转的角度；
- **滚动角**（roll）是绕**F**旋转的角度。

4.3.9.2 世界空间

世界空间（world space）是一个固定坐标空间。游戏世界中所有物体的位置、定向和缩放都会用此空间表示。此坐标空间把所有单个物体联系在一起，形成一个内聚的虚拟世界。

世界空间的原点可置于任何地方，但通常我们会把原点置于接近可玩游戏空间的中心，因为当(x, y, z)的值非常大时，浮点小数会出现精度问题，这样设置原点可使精度问题降至最低程度。尽管笔者遇到的大部分引擎都使用y轴向上或z轴向上的协定，但x、y、z轴的方向是可以随意的。若采用y轴向上协定，则通常是延伸大部分数学教科书的二维协定，当中y轴向上而x轴向右。z轴向上协定也是常见的，因为这样一来，游戏的俯览正射视角（top-down orthographic view）便会和传统的xy坐标图一样。

比方说，假设一架飞机的左翼尖位于模型空间的$(5, 0, 0)$。（另外，游戏的模型坐标采用$+z$轴向前、$+y$轴向上，如图4.17所示。）现在，想象飞机是朝向世界空间的$+x$轴方向，而飞机的模型空间原点则位于世界空间的某个位置，例如$(-25, 50, 8)$。由于飞机的$\mathbf{F}$矢量对应模型空间的$+z$轴，在世界空间里则朝向$+x$轴，所以可以得知飞机已绕世界空间的y轴旋转了90°。若飞机位于世界原点，则其左翼尖会位于世界空间的$(0, 0, -5)$。然而，因为飞机的原点已平移至$(-25, 50, 8)$，左翼尖的最终坐标便会是$(-25, 50, [8 - 5]) = (-25, 50, 3)$。图4.18描绘了这些关系。

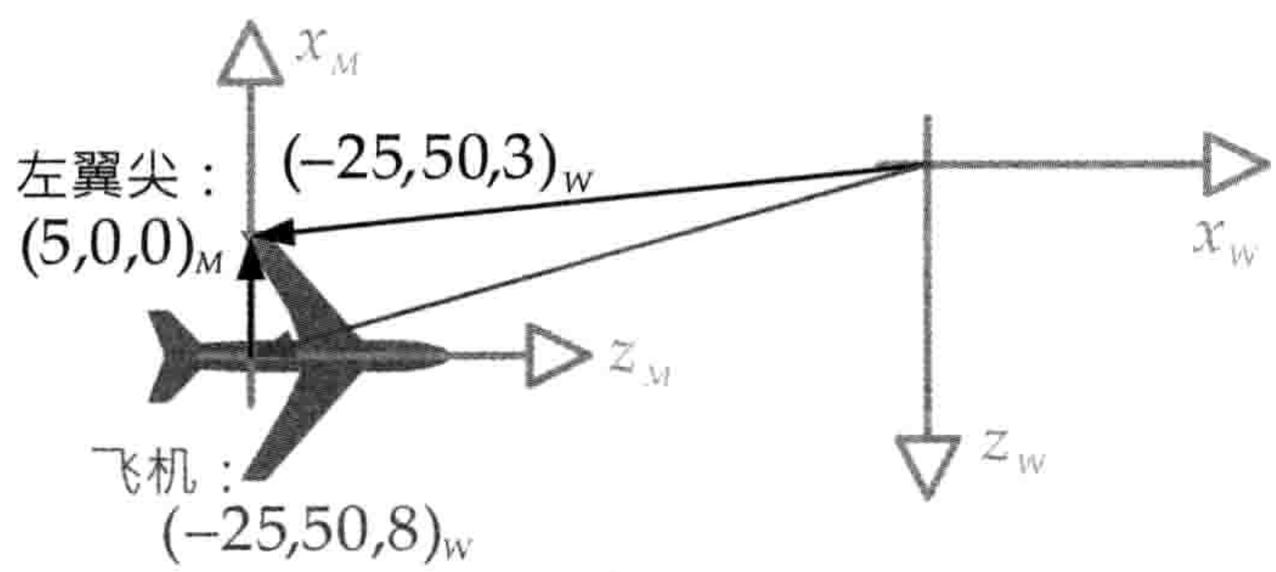


图 4.18: 这是一架里尔喷射机，其左翼尖位于模型空间的$(5, 0, 0)$。若喷射机绕世界空间y轴旋转90°，那么其模型空间原点就会平移至$(-25, 50, 8)$，而其左翼尖就会到达世界空间的$(-25, 50, 3)$。

我们可以在天空中多加几架飞机，这些飞机的左翼尖的模型空间坐标仍然是$(5, 0, 0)$，但是在世界坐标中，按照每架飞机的平移和定向，其左翼尖的坐标是完全不同的。

4.3.9.3 观察空间

观察空间（view space）又称为**摄像机空间**（camera space），是固定于摄像机的坐标系。观察空间原点置于摄像机的焦点（focal point）[11]。而且，观察空间也可采用不同的轴定向方案。但是，y轴向上、z轴顺着摄像机面对方向，是最典型的，因为$+z$轴代表着屏幕的深度[12]。其他引擎和API（如OpenGL）则用右手坐标系定义观察空间，使摄像机朝向的方向为$-z$轴，z坐标代表负深度。图4.19展示了观察空间的左、右手坐标系统。

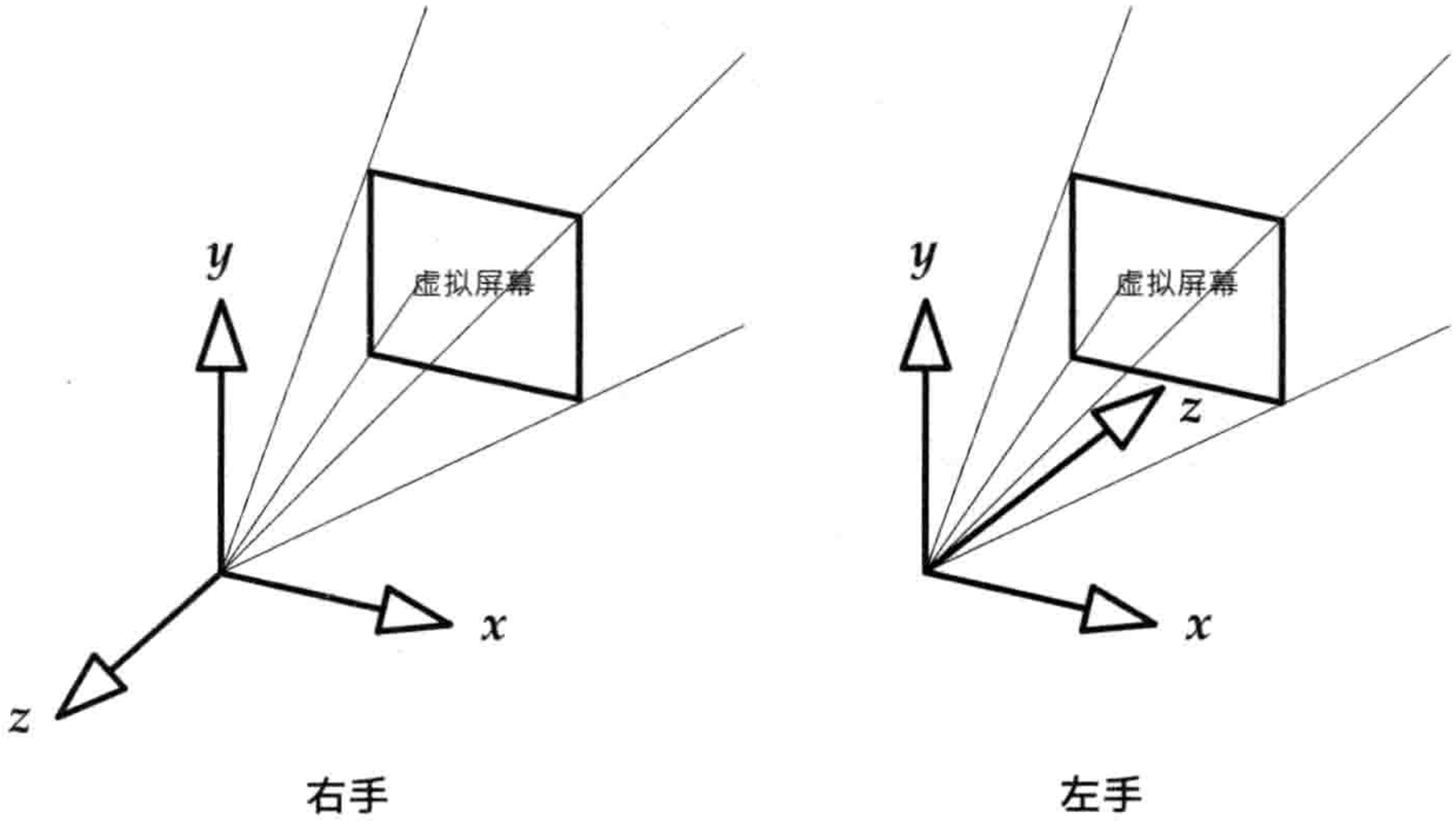


图 4.19: 观察空间（或称摄像机空间）的左、右手坐标系统例子。

4.3.10 基的变更

在游戏和计算机图形学里，经常把物体的位置、定向和缩放从某个坐标系转换至另一个坐标系。我们称此运算为**基的变更**（change of basis）。

4.3.10.1 坐标空间的层次结构

坐标系是相对的。即是说，若想在三维空间中定义一组轴，必须指明其位置、定向和缩放的数值是**相对**于另外一组轴的（否则那些数值是没意义的）。此意味着，坐标空间会形成

[11]译注：原文中的“焦点”可能产生歧义，因为焦点可以指镜头向景物对焦的位置。在计算机图形学中，通常采用眼睛（eye）或视点（view point）等术语表示观察空间的原点。

[12]x轴和y轴则是一般数学上习惯采用的二维坐标，$+x$向右，$+y$向上。

一个**层阶结构**——每个坐标空间都是某个坐标空间之**子**，而那个坐标空间则是**父**的角色。世界空间并无父，因为它是坐标空间树的根，其他坐标空间则直接或间接地相对于世界空间。

4.3.10.2 建构改变基的矩阵

把点或方向从任何子坐标系C变换至其父坐标系P的矩阵，可写作$\mathbf{M}_{C\to P}$（读作“C至P”）。此下标表示矩阵把点或方向从子空间变换至父空间。以下等式可把任何子空间位置矢量$\mathbf{P}_C$变换至父空间位置矢量$\mathbf{P}_P$：

$$\mathbf{P}_P = \mathbf{P}_C\mathbf{M}_{C\to P}$$

$$\mathbf{M}_{C\to P} = \begin{bmatrix} \mathbf{i}_C & 0 \\ \mathbf{j}_C & 0 \\ \mathbf{k}_C & 0 \\ \mathbf{t}_C & 0 \end{bmatrix} = \begin{bmatrix} i_{C_x} & i_{C_y} & i_{C_z} & 0 \\ j_{C_x} & j_{C_y} & j_{C_z} & 0 \\ k_{C_x} & k_{C_y} & k_{C_z} & 0 \\ t_{C_x} & t_{C_y} & t_{C_z} & 0 \end{bmatrix}$$

以上方程中：

- $\mathbf{i}_C$为子空间x轴的单位基矢量，此矢量以父空间坐标表示。
- $\mathbf{j}_C$为子空间y轴的单位基矢量，此矢量以父空间坐标表示。
- $\mathbf{k}_C$为子空间z轴的单位基矢量，此矢量以父空间坐标表示。
- $\mathbf{t}_C$为子坐标系相对于父坐标系的平移。

本结论应该是显而易见的。矢量$\mathbf{t}_C$只不过是子空间轴组相对于父空间的位置，因此若矩阵余下部分为单位矩阵，则在子空间的点$(0,0,0)$会变成父空间的$\mathbf{t}_C$。而矩阵左上3×3部分的单位矢量$\mathbf{i}_C$、$\mathbf{j}_C$和$\mathbf{k}_C$代表了纯旋转，因为这些都是单位长度矢量。以下用一个简单例子，可以做更清楚的解释。假设子空间绕z轴旋转角度γ，而没有平移。这种旋转的矩阵为：

$$\text{rotate}_z(\mathbf{r},\gamma) = \begin{bmatrix} r_x & r_y & r_z & 1 \end{bmatrix} \begin{bmatrix} \cos\gamma & \sin\gamma & 0 & 0 \\ -\sin\gamma & \cos\gamma & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{4.2}$$

如图4.20所示，$\mathbf{i}_C$和$\mathbf{j}_C$矢量能以父空间的坐标表达，分别为$\mathbf{i}_C = [\cos\gamma \quad \sin\gamma \quad 0]$及$\mathbf{j}_C = [-\sin\gamma \quad \cos\gamma \quad 0]$。当把这两个矢量，连同$\mathbf{k}_C = [0 \quad 0 \quad 1]$，代入$\mathbf{M}_{C\to P}$的公式，就可发现它与方程(4.2)的$\text{rotate}_z(\mathbf{r}, \gamma)$完全相等。

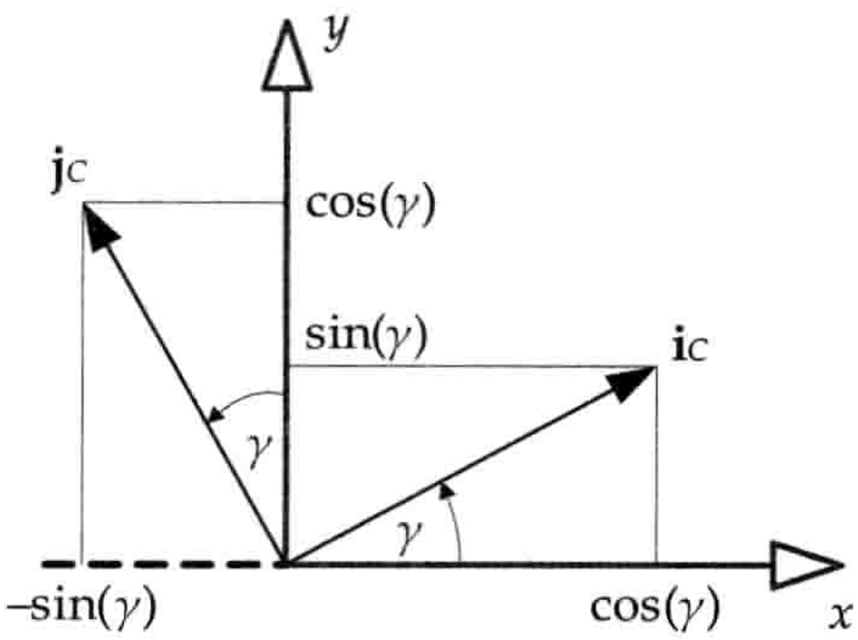


图 4.20: 子轴相对于父旋转γ角度所造成的基的变更。

缩放子轴

通过简单且恰当地缩放单位基矢量，便可以缩放子坐标系统。例如，若子空间放大两倍，则基矢量$\mathbf{i}_C$、$\mathbf{j}_C$、$\mathbf{k}_C$就会由单位长度变成长度为2。

4.3.10.3 从矩阵中获取单位基矢量

由于基变更矩阵是由平移及3个笛卡儿基矢量组成的，此事实可带来一个强大工具：给定**任何**4×4仿射矩阵，都可以用反向思维，从恰当的矩阵行（若使用列矢量则为矩阵列）中获取子空间基矢量$\mathbf{i}_C$、$\mathbf{j}_C$、$\mathbf{k}_C$。

例如，给定某车辆模型的世界变换为一个4×4仿射矩阵（这是十分常见的表示法）。该矩阵实际上只不过是基变更矩阵，把模型中的点从模型空间转换到世界空间。进一步假设游戏中z轴对着物体朝向的方向。那么，要取得车辆在世界空间里朝向的单位矢量，只需直接从模型至世界矩阵中抽取$\mathbf{k}_C$（即矩阵的第3行）[13]。这个矢量已经被归一化并可直接使用。

[13]译注：从另一个角度看，该阵列可把方向矢量从模型空间转换至世界空间。那么，只要用它来转换模型空间的z轴$(0,0,1,0)$便可得到世界空间的方向。可以看到，$(0,0,1,0)$乘以该矩阵之后，答案就是$(k_{Cx}, k_{Cy}, k_{Cz}, 0)$。

4.3.10.4 变换坐标系还是矢量

前文提及，矩阵$\mathbf{M}_{C\to P}$把点和方向从子空间变换至父空间。矩阵$\mathbf{M}_{C\to P}$的第4行包含$\mathbf{t}_C$，此即子坐标轴相对父坐标轴的平移。因此，可以把矩阵$\mathbf{M}_{C\to P}$想象为，把父**坐标轴**变换**至**子坐标轴。这是点和矢量变换的逆变换。换句话说，若某矩阵把**矢量**从子空间变换至父空间，那么该矩阵也同时把**坐标轴**从父空间变换至子空间。这是合理的，例如，可以想象，在固定的坐标轴里把点向右移20个单位，实质上等同于——把该点固定而把坐标轴向左移20个单位。此推论可参考图4.21进行理解。

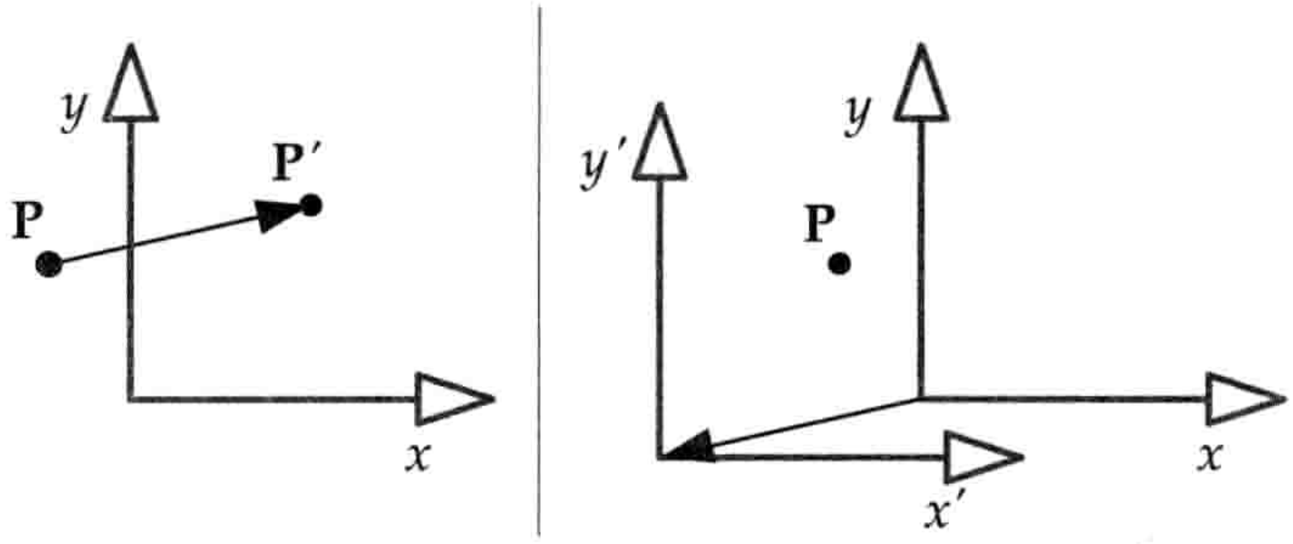


图 4.21: 对变换矩阵的两种诠释。左图中，点**P**在固定的坐标系中移动。右图中，点**P**维持静止，坐标系向相反方向移动。

当然，这一推论也会带来混淆。若基于坐标轴来思考，变换会是某一个方向；若基于点和坐标来思考，变换则是另一个方向！如同生命中会遇到很多让人混淆的事情，对其最佳的处理办法，可能是选择一个“规范”方式，并贯彻使用。例如，本书选择了以下的协定。

- 变换施于矢量（而非坐标轴）。
- 把矢量写成行（而非列）。

这两个协定，使我们可以从左至右阅读矩阵乘法，而且更容易理解，例如：

$$\mathbf{P}_D = \mathbf{P}_A\mathbf{M}_{A\to B}\mathbf{M}_{B\to C}\mathbf{M}_{C\to D}$$

显然，若想基于变换坐标轴思考（而非对点及矢量变换），要么从右至左阅读矩阵乘法，要么反转两个协定之一。读者可选择任意协定，只要其便于记忆和使用即可。

话虽如此，须注意，对某些问题，从矢量变换的角度去思考比较容易；而另一些问题，则适合用坐标轴变换。当逐渐善于基于三维矢量和矩阵数学去思考时，便会发现，针对眼前的问题灵活转换这些协定，并非难事。

4.3.11 变换法矢量

法矢量是一种特殊的矢量，因为它除了是单位矢量（通常情况是）外，法矢量还有附加要求——维持与对应的表面或平面**垂直**。变换法矢量时须特别留心，以确保维持其长度和垂直性。

一般来说，若点或（法矢量以外的）矢量可用3×3矩阵$\mathbf{M}_{A \to B}$将其从空间A旋转至空间B，则法矢量$\mathbf{n}$可使用该矩阵的**逆转置**矩阵$(\mathbf{M}_{A \to B}^{-1})^{\mathrm{T}}$做变换。本书不提供相关的推导或证明（[28] 3.5节含极优的推导）。然而，若矩阵$\mathbf{M}_{A \to B}$只含统一缩放而无切变，那么可以观察到，表面间和矢量间的夹角在A和B空间中是不变的。在此情况下，矩阵$\mathbf{M}_{A \to B}$可施于任何矢量，无论是法矢量还是其他矢量。然而，若$\mathbf{M}_{A \to B}$含非统一缩放或切变（即$\mathbf{M}_{A \to B}$**非正交**），则表面间和矢量间的夹角，从A空间变换到B空间后会改变。在A空间垂直于某表面的矢量，在B空间则未必如此。逆转置运算就是为此而设的，即使变换里含非统一缩放或切变，变换后的法矢量仍然垂直于其对应表面。

4.3.12 内存中存储矩阵

在C/C++语言中，通常使用二维数组储存矩阵。重温一下C/C++二维数组的语法，第1个索引值代表行，第2个索引值代表列。若在内存中顺序移动，列索引变化最快。

```
float m[4][4]; // [行][列], 列索引变化最快

// 使数组“平面化”以显示其次序
float *pm = &m[0][0];
ASSERT(&pm[0] == &m[0][0]);
ASSERT(&pm[1] == &m[0][1]);
ASSERT(&pm[2] == &m[0][2]);
// 以此类推
```

用C/C++二维数组存储矩阵有下面两个选择。

1. 把矢量$(\mathbf{i}_C, \mathbf{j}_C, \mathbf{k}_C, \mathbf{t}_C)$**连续**置于内存中（即每行含一个矢量）。
2. 把矢量在内存中**分散对齐**（stride）（即每列含一个矢量）。

方法1的好处是，要取得4个矢量任何一个，只需简单地索引矩阵，再把该位置的4个连续数值当作包含4个元素的矢量。此内存布局也有和行矢量等式匹配的好处（这是本书选择**行矢量**表示法的另一原因）。方法2也有用武之地，例如，在含矢量运算功能（如单指令多数据/SIMD）的微处理器中进行快速矢量矩阵乘法，本章稍后会对其进行详述。笔者遇到的大多数游戏引擎都会使用方法1，即使用C/C++二维组数中的每**行**去存储**矢量**，代码如下：

```
float M[4][4];

M[0][0]=ix;  M[0][1]=iy;  M[0][2]=iz;  M[0][3]=0.0f;
M[1][0]=jx;  M[1][1]=jy;  M[1][2]=jz;  M[1][3]=0.0f;
M[2][0]=kx;  M[2][1]=ky;  M[2][2]=kz;  M[2][3]=0.0f;
M[3][0]=tx;  M[3][1]=ty;  M[3][2]=tz;  M[3][3]=1.0f;
```

在调试器中，矩阵M的样式如下：

```
M[][]
     [0]
          [0] ix
          [1] iy
          [2] iz
          [3] 0.0000
     [1]
          [0] jx
          [1] jy
          [2] jz
          [3] 0.0000
     [2]
          [0] kx
          [1] ky
          [2] kz
          [3] 0.0000
     [3]
          [0] tx
          [1] ty
          [2] tz
          [3] 1.0000
```

要得知使用中的引擎采用了哪个布局，其中一个方法是，寻找4×4平移矩阵生成函数。（每个优良的三维数学库都应有此函数。）之后查看源代码，找出t矢量的元素是怎么存储的。若不能读取数学库的源代码（游戏业界里这种情况的可能性较小），则可以用容易辨认的平移矢量，如$(4,3,2)$，去调用该平移生成函数，再查看其传回的矩阵。若第3行含有值4.0，3.0，2.0，1.0，则矢量以行储存，否则以列储存。

4.4 四元数

我们知道，3 × 3矩阵可用来表示三维中任意的旋转。然而，矩阵并不总是理想的旋转表达形式，理由如下。

1. 矩阵需9个浮点值表示旋转，这显然是有冗余的，因为旋转只有3个自由度（degree of freedom，DOF）——偏航角、俯仰角、滚动角。
2. 用矢量矩阵乘法来旋转矢量，需3个点积，即共9个乘数及6个加数。若有可能，我们希望找到一种旋转表示方式，能加快旋转运算。
3. 在游戏和计算机图形学中，经常需要计算在两个已知旋转之间，某个比例的旋转。例如，若要平滑地把摄像机在几秒内从某起始定向A旋转到目标定向B，便需在其间找出A和B之间的许多中间旋转。若以矩阵表示A和B的定向，要计算这些中间值是很困难的。

幸好有一个旋转表达形式能克服以上3个问题。此数学对象称为**四元数**（quaternion）。四元数看似四维矢量，但**行为**上有很大区别。我们通常把四元数写成非斜非粗字体，例如：$\mathrm{q} = [q_x \quad q_y \quad q_z \quad q_w]$。

四元数是由威廉·哈密顿爵士（Sir William Rowan Hamilton）于1843年发明的，作为复数（complex number）的延伸，四元数最初是用于解决力学中的问题的。严格地说，四元数遵守一组规则，这些规则称为实数域上的四维**赋范可除代数**（normed division algebra）。幸好我们不用了解这些相当深奥的规则细节。对我们的应用来说，只需知道，**单位长度**的四元数[14]（即所有符合$q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$的四元数）能代表三维旋转。

网上有许多关于四元数的文献、网页及简报可供进一步阅读，笔者最喜爱的是加利福尼亚大学圣迭戈分校的教材[15]。

4.4.1 把单位四元数视为三维旋转

单位四元数可以视觉化为三维矢量加上第四维的标量坐标。矢量部分$\mathbf{q}_V$是旋转的单位轴乘以旋转半角的正弦；而标量部分q_S是旋转半角的余弦。那么单位四元数可写成：

$$\begin{aligned}\mathrm{q} &= \begin{bmatrix}\mathbf{q}_V & q_S\end{bmatrix} \\ &= \begin{bmatrix}\mathbf{a}\sin\frac{\theta}{2} & \cos\frac{\theta}{2}\end{bmatrix}\end{aligned}$$

[14]译注：下文称为单位四元数（unit quaternion）。

[15]http://graphics.ucsd.edu/courses/cse169_w05/CSE169_04.ppt

其中**a**为旋转轴方向的单位矢量，而θ为旋转角度。旋转方向使用**右手法则**，即是说，若使右手拇指朝向旋转轴的方向，正旋转角则是其余4只手指弯曲的方向。

当然，也可以把q写成简单的4个元素矢量：

$$\mathrm{q} = \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix}$$

其中：

$$q_x = q_{V_x} = a_x \sin\frac{\theta}{2}$$
$$q_y = q_{V_y} = a_y \sin\frac{\theta}{2}$$
$$q_z = q_{V_z} = a_z \sin\frac{\theta}{2}$$
$$q_w = q_{V_w} = \cos\frac{\theta}{2}$$

单位四元数和轴角（axis-angle）旋转表达方式很相似（即含4个元素的矢量形式为[**a** θ]）。然而，四元数在数学上比轴角更方便，稍后就会看到。

4.4.2 四元数运算

四元数提供许多矢量代数中常见的运算，例如，模及矢量加法。然而，必须谨记，两个四元数相加的和并不能代表三维旋转，因为该四元数并不是单位长度的。因此，在游戏引擎中不会看见四元数的和，除非它们用某方法缩放至符合单位长度的要求。

4.4.2.1 四元数乘法

用于四元数上的最重要运算之一就是乘法。给定两个四元数p和q，分别代表旋转**P**和**Q**，则pq代表两旋转的合成旋转（即旋转**Q**之后再旋转**P**）。其实四元数乘法有几种，但这里只讨论和三维旋转应用相关的乘法，此乘法称为格拉斯曼积（Grassmann product）。此定义下，pq之积为：

$$\mathrm{pq} = \begin{bmatrix} (p_S\mathbf{q}_V + q_S\mathbf{p}_V + \mathbf{p}_V \times \mathbf{q}_V) & (p_S q_S - \mathbf{p}_V \cdot \mathbf{q}_V) \end{bmatrix}$$

注意格拉斯曼积也是以矢量和标量部分来定义的，矢量部分的结果为四元数的x、y、z分量，标量部分则是w分量。

4.4.2.2 共轭及逆四元数

对四元数q求**逆**（inverse）写为q^{-1}，逆四元数和原四元数的乘积会变成标量1（即$qq^{-1} = 0\mathbf{i}+0\mathbf{j}+0\mathbf{k}+1$）。四元数[0 0 0 1]代表零旋转（从$\sin 0 = 0$代表前3个分量并且$\cos 0 = 1$代表第4个分量，可见其合理性）。

要计算逆四元数，先要定义一个称为**共轭**（conjugate）的量。共轭通常写成q^*，定义如下：

$$q^* = \begin{bmatrix} -\mathbf{q}_V & q_S \end{bmatrix}$$

换句话说，共轭是矢量部分求反（negation），但保持标量部分不变。

有了这个共轭定义，逆四元数q^{-1}的定义如下：

$$q^{-1} = \frac{q^*}{|q|^2}$$

由于我们使用的四元数都是用于代表三维旋转的，这些四元数都是单位长度的（即$|q| = 1$）。因此，这种情况下，共轭和逆四元数是相等的：

$$q^{-1} = q^* = \begin{bmatrix} -\mathbf{q}_V & q_S \end{bmatrix} \qquad \text{当} \qquad |q| = 1$$

这一结论是非常有价值的，因为它意味着计算逆四元数时，当知道四元数已被归一化，就不用除以模平方了（相对费时）。同时也意味着，通常计算逆四元数比计算3×3逆矩阵快得多，在某些情况下，我们可以利用这一特点优化引擎。

积的共轭及逆四元数

四元数积（pq）的共轭，等于求各个四元数的共轭后，以相反次序相乘：

$$(pq)^* = q^*p^*$$

类似地，四元数积的逆等于求各个四元数的逆后，以相反次序相乘：

$$(pq)^{-1} = q^{-1}p^{-1} \tag{4.3}$$

这种相反次序运算，同样适用于矩阵积的转置和逆。

4.4.3 以四元数旋转矢量

怎样以四元数旋转矢量？首先要把矢量重写为**四元数形式**。矢量是涉及基矢量**i**、**j**、**k**的和，四元数是涉及基矢量**i**、**j**、**k**以及第4个标量项之和。因此，把矢量写成四元数，并把标量项q_S设为0，合乎情理。给定矢量**v**，可把它写成对应的四元数$\mathrm{v} = [\mathbf{v} \quad 0] = [v_x \quad v_y \quad v_z \quad 0]$。

要以四元数q旋转矢量**v**，须用q前乘以矢量**v**（以**v**的对应四元数形式），再后乘以逆四元数q^{-1}。旋转后的矢量$\mathbf{v}'$可如下得出：

$$\mathrm{v}' = \mathrm{rotate}(\mathrm{q}, \mathbf{v}) = \mathrm{q}\mathrm{v}\mathrm{q}^{-1}$$

因为旋转用的四元数都是单位长度的，所以使用共轭也是等同的：

$$\mathrm{v}' = \mathrm{rotate}(\mathrm{q}, \mathbf{v}) = \mathrm{q}\mathrm{v}\mathrm{q}^{*} \tag{4.4}$$

只要从四元数形式的v′提取矢量部分，就能得到旋转后的矢量$\mathbf{v}'$。[16]

在真实的游戏中，众多不同的场合都适用四元数乘法。例如，求飞机飞行方向的单位矢量。再假设我们的游戏采用$+z$轴代表物体向前的协定。那么，按照定义，任何物体**在模型空间**的向前单位矢量都必然是$\mathbf{F}_M \equiv [0 \quad 0 \quad 1]$。要把此矢量变换至世界空间，只需轻松地把代表飞机定向的四元数q，用公式(4.4)去旋转模型空间的$\mathbf{F}_M$，就能得出世界空间$\mathbf{F}_W$（当然，要把这些矢量转换至四元数形式）：

$$\mathrm{F}_W = \mathrm{q}\mathrm{F}_M\mathrm{q}^{-1} = \mathrm{q}\begin{bmatrix}0 & 0 & 1 & 0\end{bmatrix}\mathrm{q}^{-1}$$

4.4.3.1 四元数的串接

和基于矩阵的变换一模一样，四元数可通过相乘**串接**旋转。例如，考虑3个四元数q_1、q_2、q_3分别表示不同旋转，并对应其等价的矩阵$\mathbf{R}_1$、$\mathbf{R}_2$、$\mathbf{R}_3$。我们希望首先进行旋转1，接着旋转2，最后旋转3。求合成旋转矩阵$\mathbf{R}_{\mathrm{net}}$和其旋转矢量**v**，等式如下：

[16]译注：上述公式可以简化为：$\mathbf{v}' = \mathbf{rotate}(\mathrm{q}, \mathbf{v}) = \mathbf{v} + 2\mathbf{q}_V \times (\mathbf{q}_V \times \mathbf{v} + q_S\mathbf{v})$。在Kavan等人的学术论文http://isg.cs.tcd.ie/kavanl/papers/sdq-tog08.pdf中的Lemma 4证明了该公式。此公式无须把**v**转换成四元数，其运算量也比原始的版本（使用两次公式(4.3)）有所减少。

$$\mathbf{R}_{\text{net}} = \mathbf{R}_1\mathbf{R}_2\mathbf{R}_3$$

$$\mathbf{v}' = \mathbf{v}\mathbf{R}_1\mathbf{R}_2\mathbf{R}_3 = \mathbf{v}\mathbf{R}_{\text{net}}$$

相似地，求合成旋转四元数q_{net}和其旋转矢量$\mathbf{v}$（以四元数形式表示的$\mathbf{v}$），等式如下：

$$\mathrm{q}_{\text{net}} = \mathrm{q}_3\mathrm{q}_2\mathrm{q}_1$$

$$\mathrm{v}' = \mathrm{q}_3\mathrm{q}_2\mathrm{q}_1\mathrm{v}\mathrm{q}_1^{-1}\mathrm{q}_2^{-1}\mathrm{q}_3^{-1} = \mathrm{q}_{\text{net}}\mathrm{v}\mathrm{q}_{\text{net}}^{-1}$$

注意，四元数的相乘次序和进行旋转的次序必须是**相反的**（$\mathrm{q}_3\mathrm{q}_2\mathrm{q}_1$）。因为旋转四元数会在矢量的**两边**相乘，没有求逆的四元数在左边，逆四元数在右边。从等式(4.3)可知，四元数积的逆等于求各个四元数的逆后，以相反次序相乘。因此，没有求逆的四元数从右至左阅读，而逆四元数则从左至右阅读。

4.4.4 等价的四元数和矩阵

任何三维旋转都可以从3 × 3矩阵表达方式$\mathbf{R}$和四元数表达方式q之间自由转换。若设$\mathrm{q} = [\mathbf{q}_V \quad q_S] = [q_{V_x} \quad q_{V_y} \quad q_{V_z} \quad q_S] = [x \quad y \quad z \quad w]$，则可用如下方式求$\mathbf{R}$：

$$\mathbf{R} = \begin{bmatrix} 1-2y^2-2z^2 & 2xy+2zw & 2xy-2yw \\ 2xy-2zw & 1-2x^2-2z^2 & 2yz+2xw \\ 2xz+2yw & 2yz-2xw & 1-2x^2-2y^2 \end{bmatrix}$$

相似地，给定$\mathbf{R}$也可用以下方式求q（当中q[0]= q_{V_x}，q[1]= q_{V_y}，q[2]= q_{V_z}，q[3]= q_S）。这段代码假设我们采用了C/C++中的行矢量（即R[row][col]对应以上的矩阵$\mathbf{R}$）。此段代码来自Nick Bobic于1998年7月5日发表的Gamasutra网站文章[17]。此外，可通过多个关于旋转矩阵的假设，更快地进行矩阵到四元数的转换，相关讨论可参考网页[18]。

[17] http://www.gamasutra.com/view/feature/3278/rotating_objects_using_quaternions.php

[18] http://www.euclideanspace.com/maths/geometry/rotations/conversions/matrixToQuaternion/index.htm

```
void MatrixToQuaterion(
    const float R[3][3],
    float       q[/*4*/])
{
    float trace = R[0][0] + R[1][1] + R[2][2];

    // 检测主轴
    if (trace > 0.0f)
    {
        float s = sqrt(trace + 1.0f);
        q[3] = s * 0.5f;

        float t = 0.5f / s;
        q[0] = (R[2][1] - R[1][2]) * t;
        q[1] = (R[0][2] - R[2][0]) * t;
        q[2] = (R[1][0] - R[0][1]) * t;
    }
    else
    {
        // 主轴为负
        int i = 0;
        if (R[1][1] > R[0][0]) i = 1;
        if (R[2][2] > R[i][i]) i = 2;

        static const int next[3] = { 1, 2, 0 };
        int j = next[i];
        int k = next[j];

        float s = sqrt((R[i][i]
                        - (R[j][j] + R[k][k]))
                        + 1.0f);
        q[i] = s * 0.5f;

        float t;
        if (s != 0.0f)      t = 0.5f / s;
        else                t = s;

        q[3] = (R[k][j] - R[j][k]) * t;
        q[j] = (R[j][i] + R[i][j]) * t;
        q[k] = (R[k][i] + R[i][k]) * t;
    }
}
```

4.4.5 旋转性的线性插值

在游戏引擎的动画、动力学及摄像机系统中，有许多场合都需要旋转性的插值。凭借四元数的帮助，对旋转插值与对矢量和点插值同样简单。[19]

最简单快速的旋转插值方法，就是套用四维矢量的线性插值（LERP）至四元数。给定两个分别代表旋转A和旋转B的四元数q_A和q_B，可找出自旋转A至旋转B之间β百分点的中间旋转$\mathrm{q}_{\mathrm{LERP}}$：

$$\begin{aligned}\mathrm{q}_{\mathrm{LERP}} &= \mathrm{LERP}(\mathrm{q}_A, \mathrm{q}_B, \beta) = \frac{(1-\beta)\mathrm{q}_A + \beta\mathrm{q}_B}{|(1-\beta)\mathrm{q}_A + \beta\mathrm{q}_B|} \\ &= \mathrm{normalize}\left(\begin{bmatrix}(1-\beta)q_{A_x} + \beta q_{B_x} \\ (1-\beta)q_{A_y} + \beta q_{B_y} \\ (1-\beta)q_{A_z} + \beta q_{B_z} \\ (1-\beta)q_{A_w} + \beta q_{B_w}\end{bmatrix}^{\mathrm{T}}\right)\end{aligned}$$

注意插值后的四元数需要再归一。这是因为LERP运算一般来说并不保持矢量长度。

从几何上来看，如图4.22所示，$\mathrm{q}_{\mathrm{LERP}} = \mathrm{LERP}(\mathrm{q}_A, \mathrm{q}_B, \beta)$是位于自定向A到定向B之间$\beta$百分点的中间定向的四元数。数学上，LERP运算是两个四元数的加权平均，加权值为$(1-\beta)$和β（注意$(1-\beta)+\beta=1$）。

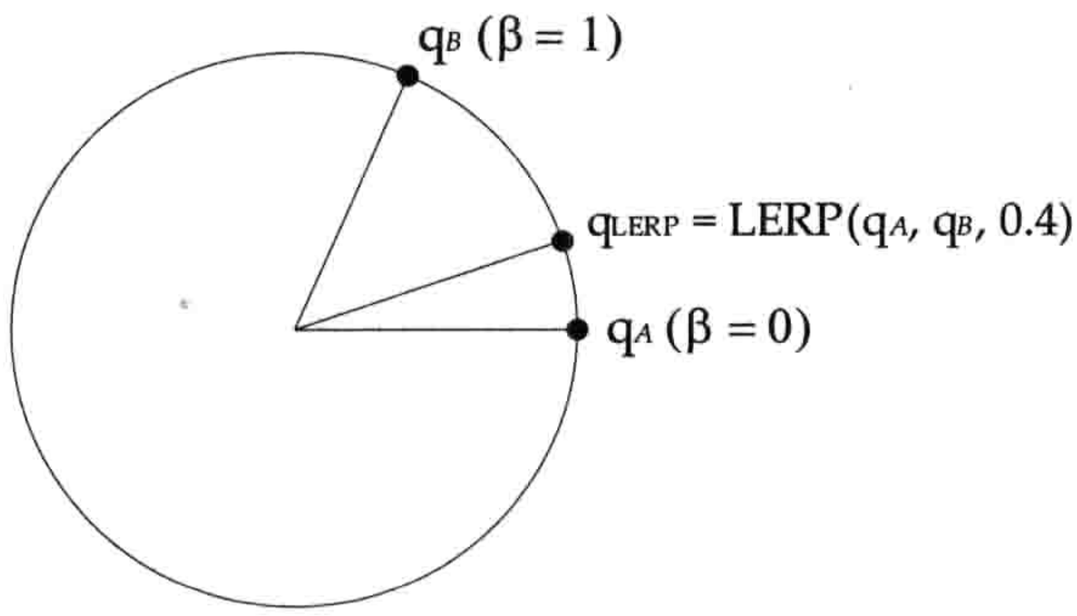


图 4.22: 对四元数q_A和q_B进行线性插值（LERP）。

[19]译注：对两个旋转矩阵插值虽然也是可行的，但却比四元数困难和慢很多。

4.4.5.1 球面线性插值

LERP运算的问题在于，它没考虑四元数其实是四维**超球**（hypersphere）上的点。LERP实际上是沿超球的**弦**（chord）[20]上进行插值，而不是在超球面上插值。这样会导致——当β以恒定速改变时，旋转动画并非以恒定角速度进行。旋转在两端看似较慢，但在动画中间就会较快。

解决此问题的方法是，采用LERP运算的变体——**球面线性插值**（spherical linear interpolation），简称SLERP。SLERP使用正弦和余弦在四维超球面的**大圆**（great circle）[21]上进行插值，而不是沿弦上插值，如图4.23所示。当β以常数速率变化，插值结果便会以常数角速率变化。

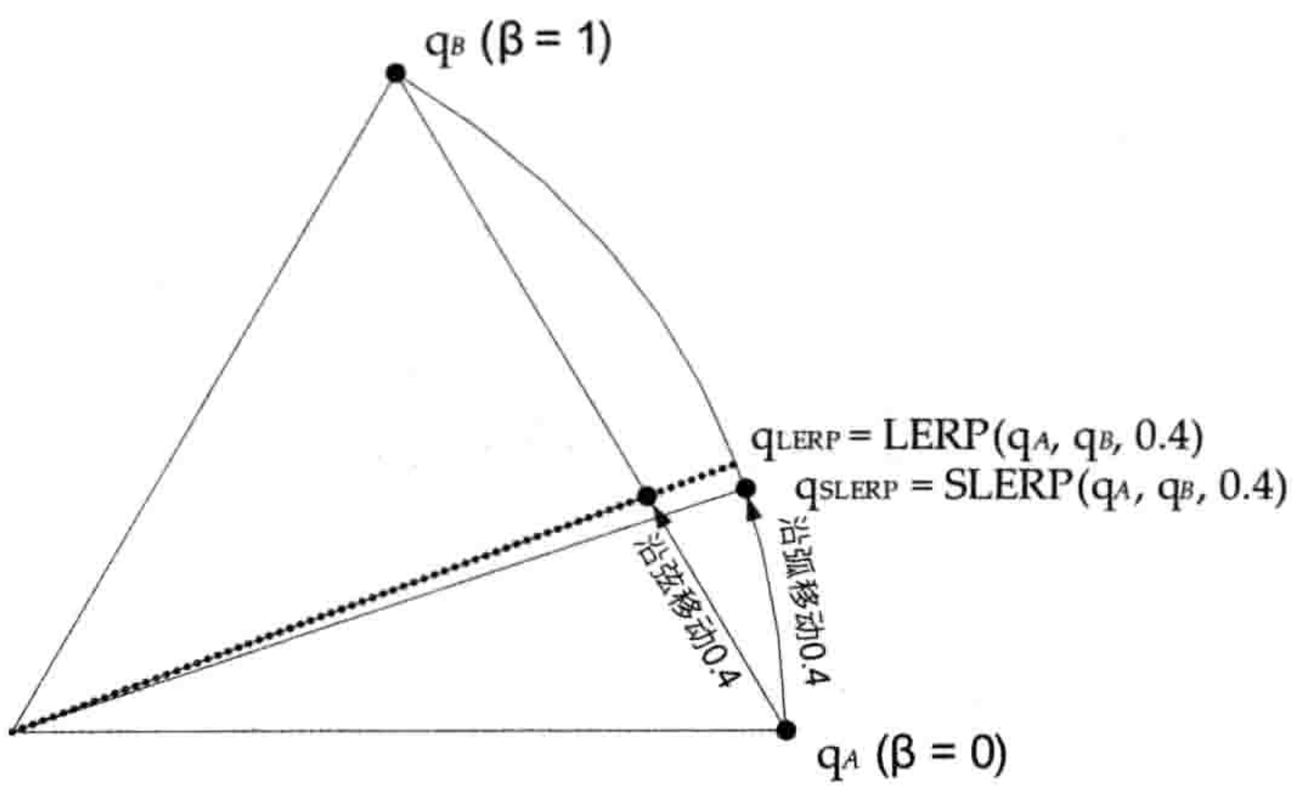


图 4.23: 球面线性插值（SLERP）沿四维超球面的大圆上插值。

SLERP公式和LERP公式相似，但其加权值以w_p和w_q取代$(1-\beta)$和β。w_p和w_q使用到两个四元数之夹角的正弦：

$$\mathrm{SLERP}(\mathrm{p}, \mathrm{q}, \beta) = w_p \mathrm{p} + w_q \mathrm{q}$$

其中：

$$w_p = \frac{\sin((1-\beta)\theta)}{\sin\theta}$$
$$w_q = \frac{\sin(\beta\theta)}{\sin\theta}$$

[20]译注：弦是圆形、球面、超球面上两点连接而成的直线线段。

[21]译注：大圆是（超）球面上半径等于球体半径的圆弧。大圆线是连接（超）球面上两点间最短路径的曲线。

两个单位四元数之间的夹角，可以使用四维点积求得。求得$\cos\theta$后就能轻易计算θ及几个正弦：

$$\cos\theta = \mathrm{p}\cdot\mathrm{q} = p_x q_x + p_y q_y + p_z q_z + p_w q_w$$

$$\theta = \cos^{-1}(\mathrm{p}\cdot\mathrm{q})$$

4.4.5.2 SLERP还是不SLERP（现在仍是个问题）

在游戏引擎中是否应使用SLERP还未成定论[22]。Jonathan Blow写了一篇出色的文章[23]，认为SLERP太昂贵，而LERP其实不差，因此，他建议应了解SLERP，但不把它应用于游戏引擎之中。另一方面，笔者在顽皮狗的同事则发现良好的SLERP实现，其效能接近LERP。（例如，顽皮狗Ice团队的SLERP实现为每关节20个周期，LERP则是16.25个周期。[24]）因此，笔者认为最好是先测试你的SLERP和LERP实现的效能，再做决定。但若你的SLERP真的慢（并且不能加快，或没时间去优化），通常用LERP取而代之还是可以的。

4.5 比较各种旋转表达方式

我们已了解到旋转可用好几种方式表示。本节叙述最常见的旋转表达方式，并介绍它们的优劣之处。由于并不存在适用于所有情况的完美旋转表达方式，本节内容可帮助读者在特定情况下选择最优的表达方式。

4.5.1 欧拉角

在4.3.9.1节里，我们已简单探讨了**欧拉角**。欧拉角能表示旋转，由3个标量值组成：偏航角、俯仰角、滚动角。有时候会用矢量$[\theta_\gamma \quad \theta_P \quad \theta_R]$表示这些量。

此表达方式的优势在于既简单又小巧（3个浮点数），还直观——很容易把偏航角、俯仰角、滚动角视觉化。而且，围绕单轴的旋转也很容易插值。例如，要从两个不同偏航角求中间的旋转，易如反掌，只需对标量θ_γ做线性插值。然而，对于任意方向的旋转轴，欧拉角则不能轻易插值。

---

[22]译注：原标题“To SLERP or not to SLERP （That's still the question）”是仿照莎士比亚《哈姆雷特》中的名句“To be, or not to be, that is the question”。

[23]http://number-none.com/product/Understanding%20Slerp,%20Then%20Not%20Using%20It/

[24]译注：id Software的Waveren曾发表《Slerping Clock Cycles》，此文章介绍如何使用SSE指令集优化SLERP，在降低少许精度的情况下，SSE版本比纯C版本快了近9倍。http://mrelusive.com/publications/papers/SIMD-Slerping-Clock-Cycles.pdf。

除此之外，欧拉角会遭遇称为**万向节死锁**（gimbal lock）的状况。当旋转90°时，三主轴中的一个会与另一主轴完全对齐，万向节死锁就会出现。例如，若绕x轴旋转90°，y轴便会与z轴完全对齐。那么，就不能再单独绕原来的y轴旋转了，因为绕y轴和z轴的旋转实际上已经等效。

欧拉角的另一个问题是，先绕哪根轴旋转，再绕哪根轴旋转，旋转的先后次序对结果是有差别的。次序可以是“俯偏滚”、“偏俯滚”、“滚偏俯”等，每个次序都会合成不同的旋转。欧拉角的旋转次序，并无所有领域通用的标准（当然，有些领域也有其特定规范）。因此，旋转角度$[\theta_\gamma \quad \theta_P \quad \theta_R]$并不能定义一个确定的旋转，必须知道旋转次序才能正确地诠释这些数字。

最后的问题是，对于要旋转的物体，欧拉角依赖从$x/y/z$轴和**前**/**左右**/**上**方向的映射。例如，偏航角总是指绕向上轴的旋转，但是若没有额外信息，就无法知道这是对应x、y或z轴的旋转。

4.5.2 3×3矩阵

基于几个原因，3×3矩阵是方便有效的旋转表达方式。3×3矩阵不受万向节死锁影响，并可独一无二地表达任意旋转。旋转可通过矩阵乘法（即一系列点积和加法），直接了当地施于点或矢量。对于硬件加速点乘和矩阵乘法，现在多数CPU及所有GPU都有内建支持。要反转方向的旋转，可求其逆矩阵，然而，纯旋转的转置矩阵即为逆矩阵，此乃非常简单的运算。而4×4矩阵更可用来表示仿射变换（旋转、平移、缩放）。

然而，旋转矩阵不太直观。当看见一个大数字表，并不容易把它们想象为对应的三维空间变换。而且，旋转矩阵不容易插值。最后一点，相对欧拉角，旋转矩阵需大量存储空间（9个浮点数）。

4.5.3 轴角

一个以单位矢量定义的旋转轴，再加上一个标量定义的旋转角，也可用来表示旋转。这称为**轴角**（axis-angle）表达方式，有时候会写成四维矢量形式$[\mathbf{a} \quad \theta]$，其中$\mathbf{a}$是旋转轴，$\theta$为弧度单位的旋转角。在右手坐标系中，正旋的方向由右手法则定义，而左手坐标系则采用左手法则。

轴角表达方式的优点在于比较直观，而且紧凑（轴角只需4个浮点数[25]，而3×3矩阵需

[25]译注：事实上，轴角等价于另一个更简洁的表达方式——旋转矢量（rotation vector）。旋转矢量是非归一化的三维矢量，矢量方向为旋转轴，模则是弧度单位的旋转角。旋转矢量只需存储为3个浮点数。

要9个）。

轴角的重要局限之一，是不能简单地进行插值。此外，轴角形式的旋转不能直接施于点或矢量，而须先把轴角转换为矩阵或四元数。

4.5.4 四元数

前文提及，单位长度的四元数可表示旋转，其形式和轴角相似。这两个表达方式的主要区别在于，四元数的旋转轴矢量的长度为旋转半角的正弦，并且其第4分量不是旋转角，而是旋转半角的余弦。

对比轴角，四元数形式带来两个极大的好处。第一，四元数乘法能串接旋转，并把旋转直接施于点和矢量。第二，可轻易地用LERP或SLERP运算进行旋转插值。四元数只需存储为4个浮点数，这也优于矩阵。[26]

4.5.5 SQT变换

单凭四元数只能表示旋转，而4×4矩阵则可表示任意仿射变换（旋转、平移、缩放）。当四元数结合**平移矢量**和**缩放因子**（对统一缩放而言是一个标量，对非统一缩放而言则是一个矢量），就能得到一个4×4仿射矩阵的可行替代形式。我们有时候称之为**SQT变换**，因为其包含缩放（scale）因子、表示旋转的四元数（**q**uaternion）和平移（**t**ranslation）矢量。

$$\mathrm{SQT} = \begin{bmatrix} s & \mathrm{q} & \mathbf{t} \end{bmatrix} \qquad (\text{统一缩放标量}s)$$

或

$$\mathrm{SQT} = \begin{bmatrix} \mathbf{s} & \mathrm{q} & \mathbf{t} \end{bmatrix} \qquad (\text{非统一缩放矢量}\mathbf{s})$$

SQT变换广泛地应用在计算机动画中，因为其体积较小（统一缩放需要8个浮点数，非统一需要10个，相对4×3矩阵则需要12个），并且SQT变换容易插值。插值时，平移矢量和缩放因子采用LERP，四元数则可使用LERP或SLERP。

---

[26] 译注：由于旋转用的四元数必为单位长度，即$q_x^2 + q_y^2 + q_z^2 + q_w^2 = 1$，基于此约束，存储单位四元数时可略去其中一个分量。例如，存储时略去q_w，然后读取时用$q'_w = \pm\sqrt{1 - q_x^2 + q_y^2 + q_z^2}$重建原来的四元数。但我们还需要知道原本$q_w$的符号。此问题有一个简单解决方法，就是利用四元数作为旋转时q等效于−q的特性，如果原本的$q_w < 0$，我们就存储其等效的−q，即存储$(-q_x, -q_y, -q_z)$，那么重建时就可确定q_w为非负数了（正数或0）。使用了这个技巧后，旋转用的单位四元数仅需存储为3个浮点数，所需空间与欧拉角和旋转矢量相同。译者是从Crytek公司2011年的一个简报得知此技巧的。http://www.crytek.com/cryengine/presentations/spherical-skinning-with-dual-quaternions-and-qtangents。

4.5.6 对偶四元数

还有一个数学对象可完整表示涉及旋转、平移、缩放的变换，称为**对偶四元数**（dual quaternion）。对偶四元数和普通四元数很像，区别在于对偶四元数的4个分量并非实数，而是**对偶数**（dual number）。对偶数可写成非对偶部（non-dual part）和对偶部（dual part）之和：$\hat{a} = a_0 + \varepsilon a_\varepsilon$。其中$\varepsilon$是一个魔法数字，称为**对偶单位**（dual unit），定义为$\varepsilon^2 = 0$。（这可比拟虚数$i = \sqrt{-1}$，用于把复数写成实部和虚部之和$c = a + ib$。）

因为每个对偶数都能表示为两个实数（非对偶部和对偶部），对偶四元数可表示为含8个元素的矢量。对偶四元数也可表示为两个普通四元数之和，第2个四元数要乘以对偶单位：$\hat{\mathrm{q}} = \mathrm{q}_0 + \varepsilon \mathrm{q}_\varepsilon$。

在本书范围内不能完整探讨对偶数和对偶四元数。然而，网上和文献都有这方面的优秀文章。笔者建议可先参考此技术报告[27]。

4.5.7 旋转和自由度

术语“**自由度**（degree of freedom，DOF）”是指物体有多少个互相独立的可变状态（位置和定向）。读者可能在力学、机器人学或航空学等专业里听过“6个DOF”这种说法。这是指，一个三维物体（在其运动没受人工约束的情况下）在平移上有3个DOF（沿$x/y/z$轴），在旋转上也有3个DOF（绕$x/y/z$轴），共计6个DOF。

DOF的概念可以让我们了解到——虽然旋转本身是3个DOF，但各种旋转表达方式却有不同数目的浮点参数。例如，欧拉角需要3个浮点数，轴角和四元数需要4个浮点数，3×3矩阵则需要9个浮点数。这些表示法为何都能表示3个DOF的旋转？

答案在于**约束**（constraint）。所有三维旋转表达方式都有3个或以上的浮点参数，但一些表达方式也会对参数加上一个或一个以上的约束。这些约束标明参数间并非**独立**的——改变某参数会导致其他参数需要改变，以维持约束的正确性。若从浮点参数个数中减去约束个数，就会得到DOF。三维旋转的DOF总是3：

$$N_{\mathrm{DOF}} = N_{\text{参数}} - N_{\text{约束}} \tag{4.5}$$

下面把等式(4.5)套用至本书介绍过的所有旋转表达方式。

- **欧拉角**：3个参数 $-$ 0个约束 $=$ 3个DOF。

[27] https://www.scss.tcd.ie/publications/tech-reports/reports.06/TCD-CS-2006-46.pdf

- **轴角**：4个参数 − 1个约束 = 3个DOF。约束：轴矢量限制为单位长度。
- **四元数**：4个参数 − 1个约束 = 3个DOF。约束：四元数限制为单位长度。
- **3 × 3矩阵**：9个参数 − 6个约束 = 3个DOF。约束：3个行矢量和3个列矢量都限制为单位长度（每个是三维矢量）。

4.6 其他数学对象

除了点、矢量、矩阵及四元数，游戏工程师还会遇到其他大量的数学对象。本节简单介绍一些常见的数学对象。

4.6.1 直线、光线及线段

一条无限长直线（line）可表示为直线上一点$\mathbf{P}_0$及沿直线方向的单位矢量$\mathbf{u}$。直线的**参数方程**（parametric equation）可从起点$\mathbf{P}_0$，沿单位矢量$\mathbf{u}$方向移动任意距离t，求出直线上任何一点$\mathbf{P}$。无穷大的点集$\mathbf{P}$成为标量t的**矢量函数**（vector function），如图4.24所示。

$$\mathbf{P}(t) = \mathbf{P}_o + t\mathbf{u} \qquad \text{其中} -\infty < t < +\infty$$

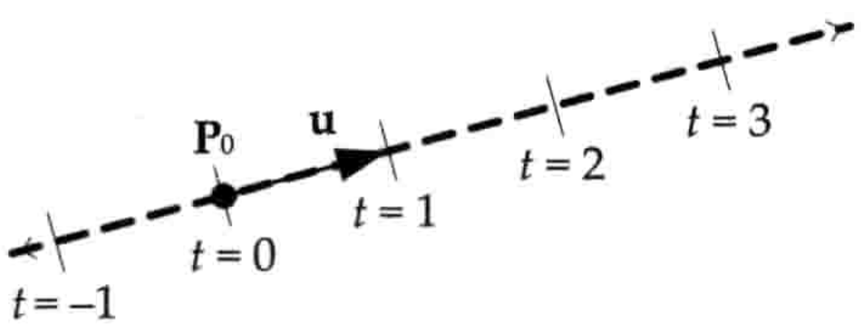


图 4.24: 直线的参数方程。

光线（ray）也是直线，但光线只沿一个方向延伸至无限远。光线可表示为$\mathbf{P}(t)$加上约束$t \geqslant 0$，如图4.25所示。

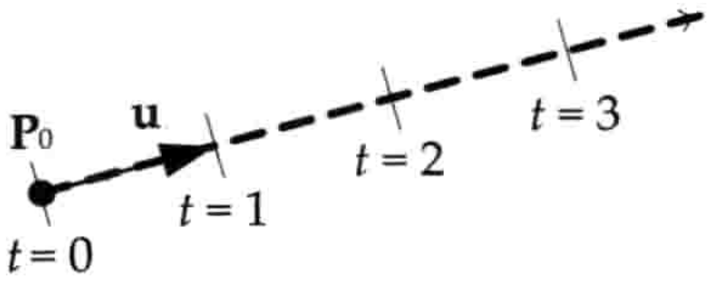


图 4.25: 光线的参数方程。

线段（line segment）受限于两个端点$\mathbf{P}_0$和$\mathbf{P}_1$。线段也可表示为$\mathbf{P}(t)$，配合以下两种形式之一（当中$\mathbf{L} = \mathbf{P}_1 - \mathbf{P}_0$，$L = |\mathbf{L}|$为线段长度）。

1. $\mathbf{P}(t) = \mathbf{P}_0 + t\mathbf{u}$, 其中$0 \leqslant t \leqslant L$。
2. $\mathbf{P}(t) = \mathbf{P}_0 + t\mathbf{L}$, 其中$0 \leqslant t \leqslant 1$。

第2种形式显示在图4.26中，此形式特别方便，因为参数t是正规化的。换句话说，无论对任何线段，t总是介乎0至1之间。这也意味着，不需要把L存储为另一个浮点参数，L已经编码进矢量$\mathbf{L} = L\mathbf{u}$里（反正$\mathbf{L}$本身需要存储）。[28]

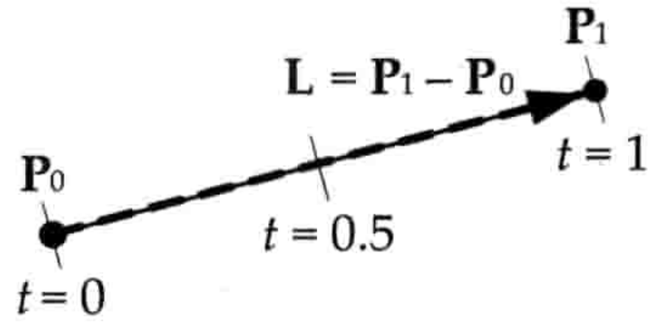


图 4.26: 线段的参数方程，采用了正规化的t参数。

4.6.2 球体

在游戏编程中，球体无处不在。球体（sphere）通常定义为中心点$\mathbf{C}$加上半径r，如图4.27所示。这恰好能置于一个四元素矢量$[C_x \quad C_y \quad C_z \quad r]$。稍后会看到，在SIMD矢量处理中，把数据打包为矢量（4个32位浮点数，共128位）有显著好处。

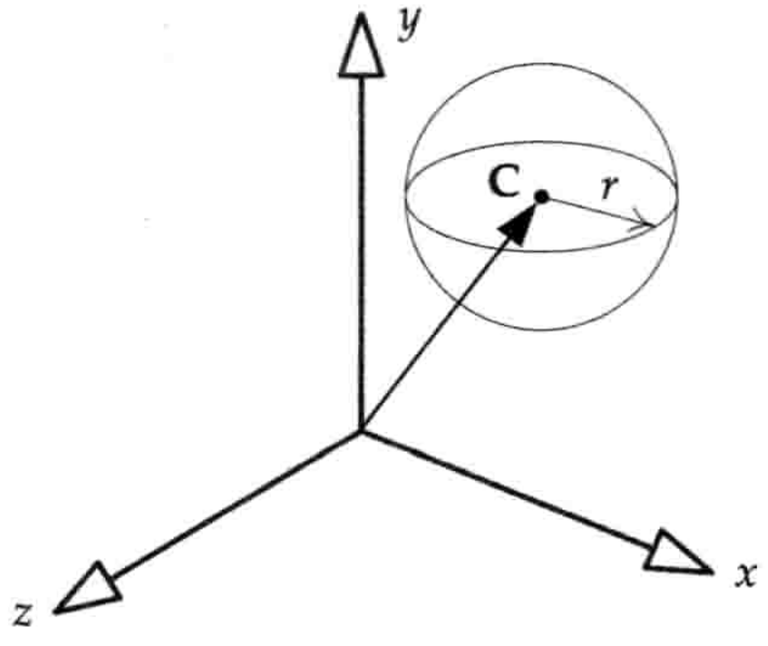


图 4.27: 以中心点和半径表示球体。

[28]译注：可以看出，这种线段参数方程式，其实等价于两点的线性插值（LERP）。用两端点$\mathbf{P}_0$和$\mathbf{P}_1$，或是$\mathbf{P}_0$加上$\mathbf{L}$表示线段，也是可行的。

4.6.3 平面

平面（plane）是三维空间中的二维表面。读者可能记得在高中代数中，平面方程[29]可写成：

$$Ax + By + Cz + D = 0$$

此方程只满足位于平面上的点$\mathbf{P} = [x \quad y \quad z]$的轨迹。平面也可用平面上一点$\mathbf{P}_0$和其单位法矢量$\mathbf{n}$来表示。此表达方式有时候称为**点法式**（point-normal form），如图4.28所示。

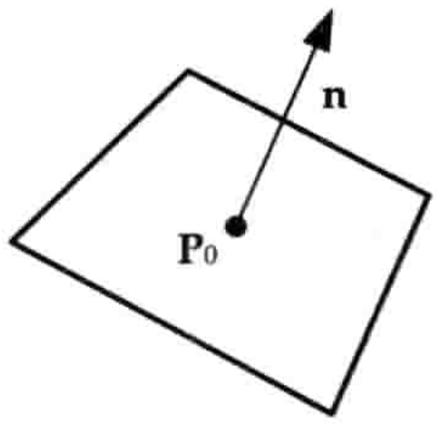


图 4.28: 以点法式表示的平面。

有趣的是，传统平面方程的参数A、B、C可诠释为三维矢量，此矢量沿平面的法线方向。若把$[A \quad B \quad C]$归一化至单位长度，则单位矢量$[a \quad b \quad c] = \mathbf{n}$，并且参数$D$正规化为$d = D/\sqrt{A^2 + B^2 + C^2}$后，$d$就是平面到原点的距离。若法矢量（$\mathbf{n}$）指向原点（即原点位于平面的“正面”），则$d$为正数；相反，若法矢量指向远离原点的方向（即原点位于平面的“背面”），则d为负数。事实上，正规化后的$ax + by + cz + d = 0$形式[30]也可写成$\mathbf{n} \cdot \mathbf{P} = -d$，意味着当任何点$\mathbf{P}$投影在矢量$\mathbf{n}$的方向时，投影的距离为$-d$。

如同球体，平面实际上也可包裹为四元素矢量。若要唯一地表示平面，可使用单位法矢量$\mathbf{n} = [a \quad b \quad c]$和平面至原点的距离$d$。四元素矢量$\mathbf{L} = [\mathbf{n} \quad d] = [a \quad b \quad c \quad d]$，是既紧凑又方便的表达和内存存储方式。注意，如果$\mathbf{P}$写成$w = 1$的齐次坐标，方程$\mathbf{L} \cdot \mathbf{P} = 0$其实等价于$\mathbf{n} \cdot \mathbf{P} = -d$。（这些方程都满足平面$\mathbf{L}$上的所有点$\mathbf{P}$。）

用四元素矢量形式定义的平面可以很容易地从某个坐标系变换至另一个坐标系。给定矩阵$\mathbf{M}_{A \to B}$，能把点和（非法线）矢量从空间A变换至空间B。前文曾述，要变换**法矢量**，如平面的矢量$\mathbf{n}$，可使用该矩阵的逆转置矩阵$(\mathbf{M}^{-1}_{A \to B})^{\mathrm{T}}$。无须惊讶，把逆转置矩阵施于四元素平面矢量$\mathbf{L}$，实际上也能正确地把平面从空间A变换至空间B。此处不会进一步推导或证明这一结果，若想详细了解此“诀窍”为何可行，可参考[28]的4.2.3节。

---

[29]译注：此方程称为一般式（general form）。除了以下提及的点法式（point-normal form），还有三点式（three point form）、参数式（parametric form）、截距式（intercept form）等。

[30]译注：正式术语是海赛正规式（Hessian normal form）。

4.6.4 轴对齐包围盒

轴对齐包围盒（axis-aligned bounding box，AABB）是三维**长方体**，其6个面都与某坐标系的正交轴对齐。因此，AABB可用六元素矢量$[x_{\min} \quad x_{\max} \quad y_{\min} \quad y_{\max} \quad z_{\min} \quad z_{\max}]$表示，即3个主轴上的最大值和最小值，又或以两点$\mathbf{P}_{\min}$和$\mathbf{P}_{\max}$表示。

此简单表达方式，可以用来简单检测一点$\mathbf{P}$是否在给定的AABB之内，检测时只需测试以下所有条件是否成立：

$$\begin{array}{lll} P_x \geqslant x_{\min} & \text{及} & P_x \leqslant x_{\max} \quad \text{及} \\ P_y \geqslant y_{\min} & \text{及} & P_y \leqslant x_{\max} \quad \text{及} \\ P_z \geqslant z_{\min} & \text{及} & P_z \leqslant x_{\max} \end{array}$$

因为AABB的交集测试这么高效，AABB常会用作碰撞检测的“早期淘汰”测试。若两个AABB 不相交，则不用再做更详细（也更费时）的检测。[31]

4.6.5 定向包围盒

定向包围盒（oriented bounding box，OBB）也是三维**长方体**，但其定向与其包围的物体按照某逻辑方式对齐。通常OBB与物体的局部空间轴对齐。这样的OBB在局部空间中如同AABB，但不一定会和世界空间轴对齐。[32]

有多种方法测试一点是否在OBB之内。常见方法是把点变换至OBB的“对齐”坐标空间，再运用上节中的AABB相交测试。

4.6.6 平截头体

如图4.29所示，**平截头体**（frustum[33]）由6个平面构成，以定义**截断头的四角锥形状**。平截头体常见于三维渲染，因为透视投影由虚拟摄像机视点造成，所以其三维世界中的可视范围是一个平截头体。平截头体的上下左右4个面代表屏幕的4边，而前后两面则代表近 / 远剪切平面（near/far clipping plane）（即所有可视点的最小/最大z坐标[34]）。

[31]译注：注意上面的公式是AABB对点的相交测试，本段谈及的是AABB对AABB的相交测试。关于后者可参阅12.3.5.4节。

[32]译注：12.3.4.4节提及OBB的表达方式。

[33]译注：本节原文为frusta，即frustum的复数。

[34]译注：此坐标是指在观察空间（view space）里。

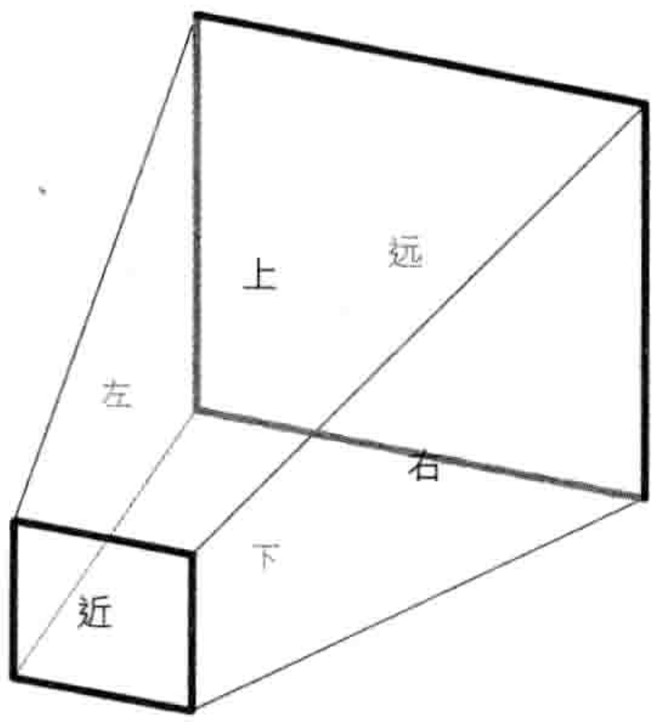


图 4.29: 平截头体[35]。

平截头体可方便地表示为6个平面的数组，而每个平面则以点法式表示（一点加上一法矢量）。[36]

要测试一点是否在平截头体里有点复杂，但基本上是用点积去测出该点是在每个平面的前面还是后面。若该点皆在6平面之内[37]，则该点在平截头体内。

有一个有用的技巧，是把要测试的世界空间点，通过摄像机的透视投影变换至另一空间，此空间称为**齐次裁剪空间**（homogeneous clip space）。世界空间的平截头体在此空间中变成AABB。那么就可以更简单地进行内外测试。

4.6.7 凸多面体区域

凸多面体区域（convex polyhedral region）[38]由任意数量的平面集合定义，平面的法线全部向内（或全部向外）。测试一点是否在平面构成的体积内，方法很简单、直接。与平截头体测试类似，只不过面的数量可能更多。游戏中，凸多面体区域非常适合做任意形状[39]的触发区域（trigger region）。许多游戏引擎也使用此技术，例如，雷神之锤引擎里无处不在的**笔刷**（brush）也正是用以平面包围而成的体积。

---

[35]译注：原图的左和右错误地反转标示。

[36]译注：用一般式也可以。

[37]译注：须定义平截头体的平面法线是向内的还是向外的。

[38]译注：或简单称之为凸多面体（convex polyhedron）。

[39]译注：单个凸多面体区域必须为凸，但若使用多个凸多面体区域的并集（union），则可表示凹多面体（concave polyhedron）。

4.7 硬件加速的SIMD运算

单指令多数据（single instruction multiple data，SIMD）是指，现代微处理器能用一个指令并行地对多个数据执行数学运算。例如，CPU可通过一个指令，把4对浮点数**并行地**相乘。SIMD广泛地应用在游戏引擎的数学库中，因为它能极迅速地执行常见的矢量运算，如点积和矩阵乘法。

1994年，英特尔（Intel）首次把多媒体扩展（multimedia extension，MMX）指令集加进奔腾CPU产品线中。把多个8/16/32位整数载入特设的64位MMX寄存器后，MMX指令集就能对那些寄存器进行SIMD运算。英特尔陆续加入多个版本的扩展指令集，称为单指令多数据流扩展（streaming SIMD extensions，SSE），其中第一个SSE版本出现于奔腾III处理器。SSE指令采用128位寄存器，可储存整数或IEEE浮点数。

游戏引擎中最常用的SSE模式为**32位浮点数打包模式**（packed 32-bit floating-point mode）。此模式中，4个32位`float`值被打包进单个128位寄存器，单个指令可对4对浮点数进行并行运算，如加法或乘法。当要计算四元素矢量和4×4矩阵相乘，这个模式正合我们所需[40]！

4.7.1 SSE寄存器

在32位浮点包裹模式中，每个SSE寄存器含4个32位`float`。为方便起见，我们将SSE寄存器中的4个`float`称作$[x \quad y \quad z \quad w]$，就如同齐次坐标的矢量/矩阵运算时的表示方式（如图4.30所示）。

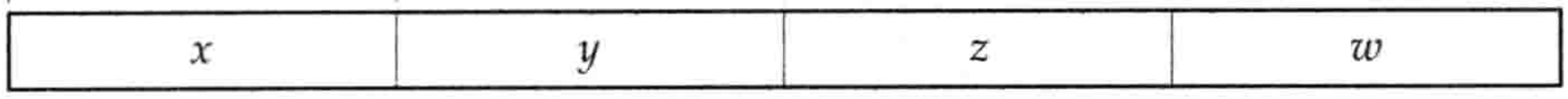


图 4.30: 使用32位浮点数模式的SSE寄存器，当中含4个分量。

为示范SSE寄存器如何运作，以下举出一个SIMD指令的例子：

```
addps xmm0, xmm1
```

`addps`指令把128位XMM0寄存器中的4个`float`分别与XMM1寄存器的4个`float`相加，4个运算结果写回XMM0。换一个方式：

[40]译注：原文“This is just what the doctor ordered”为成语，意思是调侃某事是医生的吩咐，其实是指“正合我意”。

$$\text{xmm0}.x = \text{xmm0}.x + \text{xmm1}.x$$
$$\text{xmm0}.y = \text{xmm0}.y + \text{xmm1}.y$$
$$\text{xmm0}.z = \text{xmm0}.z + \text{xmm1}.z$$
$$\text{xmm0}.w = \text{xmm0}.w + \text{xmm1}.w$$

储存于SSE寄存器的4个浮点数，可以个别抽出存进内存，或从内存载入，但是这类操作速度相对较慢。在x87 FPU[41]寄存器和SSE寄存器之间传送数据很糟糕，因为CPU须等待x87单元或SSE单元完成所有正在进行的工作。这样会令CPU的整个指令执行流水线停顿（stall），导致大量CPU周期被浪费。简而言之，应把普通`float`运算和SIMD运算的混合代码视作瘟疫，避之不迭。

为了把内存、x87 FPU寄存器和SSE寄存器之间的数据传输量降至最低，多数SIMD数学库都会尽量把数据保存在SSE寄存器中，而且越久越好。这意味着，即使标量值也保留在SSE寄存器里，而不把它传送至`float`变量。例如，两矢量点积的结果是一个标量，但若把该标量留在SSE寄存器里，就可供稍后的矢量运算，而不会带来额外传输成本。可把单个浮点值复制至SSE寄存器的4个“位置”以表示标量。因此若要存储一个标量s至SSE寄存器，就会设$x = y = z = w = s$。

4.7.2 __m128数据类型

在C/C++中，使用这些神奇的SSE 128位值颇为容易。微软Visual Studio编译器提供了内建的`__m128`数据类型[42]。此数据类型可用来声明全局变量、自动变量，甚至是类或结构里的成员变量。大多数情况下，此数据类型的变量会存储于内存中，但在计算之时，`__m128`的值会直接在CPU的SSE寄存器中运用[43]。事实上，以`__m128`声明的自动变量或函数参数，编译器通常会把它们直接置于SSE寄存器中，而非置于内存中的程序堆栈。

---

[41]译注：FPU为浮点运算器（floating-point unit）。

[42]译注：注意前缀__是两个下画线符（underscore）。

[43]译注：这其实和一般内建数据型别相似，平时存于内存，计算时可能要载入寄存器。有些SIMD指令的寻址模式（addressing mode）可直接存取内存中的`__m128`数据。

__m128变量的对齐

当一个__m128变量储存在内存中，程序员有责任确保该变量是16字节**对齐**[44]的。这意味着，当把__m128变量的地址以十六进制表示，其最低有效半字节必须总是0x0。编译器会自动为类和结构加入填充（padding），因此，若整个类和结构是16字节对齐的，置于其中的所有__m128成员变量也会正确地对齐。若声明含一个或多个__m128的自动或全局类/结构，则编译器会自动把对象对齐。然而，当编译器要动态地分配数据结构（即用malloc()或new分配数据）时，程序员就须负责对齐，编译器帮不上忙。

4.7.3 用SSE内部函数编码

SSE运算可用原始的汇编语言实现，也可通过使用C/C++中的内联汇编（inline assembly）。然而，这么做不但缺乏可移植性，而且编程也令人头疼。为了更加简便，现今的编译器提供**内部函数**（intrinsic）。内部函数是一些特殊指令，其形式和作用都很像普通C函数，但编译器会把它们转化为内联汇编代码。多数内部函数会翻译成单个汇编语言指令，但有些内部函数是**宏**，这些宏会被翻译为一串指令。

.cpp文件需#include <xmmintrin.h>才能使用__m128数据类型和SSE内部函数。

我们再从另一个角度看一下addps汇编语言指令。在C/C++中可用_mm_add_ps()内部函数执行这条指令。以下并列比较使用内联汇编和内部函数的代码。

```
__m128 addWithAssembly(
    __m128 a,
    __m128 b)
{
    __m128 r;
    __asm
    {
        movaps xmm0, xmmword ptr [a]
        movaps xmm1, xmmword ptr [b]
        addps xmm0, xmm1
        movaps xmmword ptr [r], xmm0
    }
    return r;
}
```

```
__m128 addWithIntrinsics(
    __m128 a,
    __m128 b)
{
    __m128 r =
        _mm_add_ps(a, b);
    return r;
}
```

在汇编语言版本中，以__asm关键字使用内联汇编，必须手动地用movaps指令建立输入参数a/b和SSE寄存器xmm0/xmm1的关联。另一方面，使用内部函数的版本则直观、清楚

[44]译注：有关对齐的问题，可重温本书第3.2.5.1节。

得多，代码也较少。没有内联汇编，SSE指令看上去和普通函数调用一样。[45]

读者可对这些例子里的函数做实验，下面的main()函数可作为测试平台。留意其中使用了另一个内部函数_mm_load_ps()，其作用是把内存中的float数组载入__m128变量（即SSE寄存器）。也要留意该4个全局float数组都使用了__declspec(align(16))强制声明16字节对齐。若略去这个指令（directive），则程序运行时会崩溃。

```
#include <xmmintrin.h>

// ……定义之前那两个函数……
__declspec(align(16)) float A[] = {  2.0f, -1.0f, 3.0f, 4.0f };
__declspec(align(16)) float B[] = { -1.0f,  3.0f, 4.0f, 2.0f };
__declspec(align(16)) float C[] = {  0.0f,  0.0f, 0.0f, 0.0f };
__declspec(align(16)) float D[] = {  0.0f,  0.0f, 0.0f, 0.0f };

int main(int argc, char* argv[])
{
    // 从以上的浮点数据数组载入a和b
    __m128 a = _mm_load_ps(&A[0]);
    __m128 b = _mm_load_ps(&B[0]);

    // 测试那两个函数
    __m128 c = addWithAssembly(a, b);
    __m128 d = addWithIntrinsics(a, b);

    // 把a和b的值储存回原来的数组，确保它们没被改动
    _mm_store_ps(&A[0], a);
    _mm_store_ps(&B[0], b);

    // 把两个结果储存至数组，以便打印
    _mm_store_ps(&C[0], c);
    _mm_store_ps(&D[0], d);

    // 检查结果
    printf("%g %g %g %g\n", A[0], A[1], A[2], A[3]);
    printf("%g %g %g %g\n", B[0], B[1], B[2], B[3]);
    printf("%g %g %g %g\n", C[0], C[1], C[2], C[3]);
    printf("%g %g %g %g\n", D[0], D[1], D[2], D[3]);

    return 0;
}
```

[45]译注：使用内部函数还能让编译器有更大的优化空间，例如，优化寄存器的分配、调乱指令的次序等。此外，VC和GCC都提供相同的SSE内部函数，内联汇编则完全不一样，所以用SSE内部函数的代码更容易跨平台（编译器及操作系统）。此外，VC 64位编译器暂时仍不支持内联汇编，只支持内部函数。译者认为，一般情况下都应尽量使用内部函数。

4.7.4 用SSE实现矢量对矩阵相乘

让我们来看看如何用SSE实现矢量对矩阵的相乘。目的是把1×4矢量$\mathbf{v}$和4×4矩阵$\mathbf{M}$相乘，得出乘积矢量$\mathbf{r}$。

$$\mathbf{r} = \mathbf{vM}$$

$$\begin{bmatrix} r_x & r_y & r_z & r_w \end{bmatrix} = \begin{bmatrix} v_x & v_y & v_z & v_w \end{bmatrix} \begin{bmatrix} M_{11} & M_{12} & M_{13} & M_{14} \\ M_{21} & M_{22} & M_{23} & M_{24} \\ M_{31} & M_{32} & M_{33} & M_{34} \\ M_{41} & M_{42} & M_{43} & M_{44} \end{bmatrix}$$

$$= \begin{bmatrix} v_x M_{11} + v_y M_{21} + v_z M_{31} + v_w M_{41} \\ v_x M_{12} + v_y M_{22} + v_z M_{32} + v_w M_{42} \\ v_x M_{13} + v_y M_{23} + v_z M_{33} + v_w M_{43} \\ v_x M_{14} + v_y M_{24} + v_z M_{34} + v_w M_{44} \end{bmatrix}^{\mathrm{T}}$$

此乘法涉及计算**行**矢量$\mathbf{v}$和$\mathbf{M}$矩阵**列**矢量的点积。若要使用SSE指令来计算，可先把$\mathbf{v}$存储至SSE寄存器（__m128），再把$\mathbf{M}$矩阵的每个**列**矢量存储至SSE寄存器。那么就可利用mulps指令，并行计算所有$v_k M_{ij}$：

```
__m128 mulVectorMatrixAttemp1(__m128 v,
    __m128 Mcol1, __m128 Mcol2, __m128 Mcol3, __m128 Mcol4)
{
    __m128 vMcol1 = _mm_mul_ps(v, Mcol1);
    __m128 vMcol2 = _mm_mul_ps(v, Mcol2);
    __m128 vMcol3 = _mm_mul_ps(v, Mcol3);
    __m128 vMcol4 = _mm_mul_ps(v, Mcol4);
    // ……然后呢?
}
```

以上代码能求出以下这些中间结果：

$$\mathtt{vMcol1} = [v_x M_{11} \quad v_y M_{21} \quad v_z M_{31} \quad v_w M_{41}]$$

$$\mathtt{vMcol2} = [v_x M_{12} \quad v_y M_{22} \quad v_z M_{32} \quad v_w M_{42}]$$

$$\mathtt{vMcol3} = [v_x M_{13} \quad v_y M_{23} \quad v_z M_{33} \quad v_w M_{43}]$$

$$\mathtt{vMcol4} = [v_x M_{14} \quad v_y M_{24} \quad v_z M_{34} \quad v_w M_{44}]$$

但问题是这么做的话，就需要在寄存器内做加法，才能计算所需结果。例如，$r_x = v_xM_{11} + v_yM_{21} + v_zM_{31} + v_wM_{41}$，这需要把vMcol1的4个分量相加。把寄存器内的分量相加，是既困难又低效的。再者，相加后的结果将分散在4个SSE寄存器中，那么还需要把它们结合到单个结果矢量**r**。好在还有更好的做法。

这里的“技巧”是，使用**M**的**行**矢量相乘，而不是用列矢量。这样，就可以并行地进行加法，最终结果也会置于代表输出矢量**r**的**单个**SSE寄存器中。然而，在本技巧中不能直接用矢量**v**乘以**M**的行，而是需要用v_x乘以第1行，v_y乘以第2行，v_z乘以第3行，v_w乘以第4行。要这么做，就需要把**v**里的单个分量如v_x，**复制**（replicate）到其余的分量里去，生成一个$[v_x \quad v_x \quad v_x \quad v_x]$的矢量。之后就可以用已**复制**某分量的矢量，乘以**M**中适当的**行**。

幸好，有强大的SSE指令shufps（对应内部函数为_mm_shuffle_ps()）支持这种复制运算[46]。这个强大指令比较难理解，因为它是通用的指令，可把SSE寄存器的分量次序任意调乱。然而，这里只需知道以下的宏可用来复制x、y、z或w分量至整个寄存器：

```
#define SHUFFLE_PARAM(x, y, z, w) \
    ((x) | ((y) << 2) | ((z) << 4) | ((w) << 6))

#define _mm_replicate_x_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(0, 0, 0, 0))

#define _mm_replicate_y_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(1, 1, 1, 1))

#define _mm_replicate_z_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(2, 2, 2, 2))

#define _mm_replicate_w_ps(v) \
    _mm_shuffle_ps((v), (v), SHUFFLE_PARAM(3, 3, 3, 3))
```

给定这些方便的宏，就可以编写矢量矩阵乘法函数如下：

```
__m128 mulVectorMatrixAttempt2(__m128 v,
    __m128 Mrow1, __m128 Mrow2, __m128 Mrow3, __m128 Mrow4)
{
    __m128 xMrow1 = _mm_mul_ps(_mm_replicate_x_ps(v), Mrow1);
    __m128 yMrow2 = _mm_mul_ps(_mm_replicate_y_ps(v), Mrow2);
    __m128 zMrow3 = _mm_mul_ps(_mm_replicate_z_ps(v), Mrow3);
    __m128 wMrow4 = _mm_mul_ps(_mm_replicate_w_ps(v), Mrow4);
```

[46]译注：Shuffle为洗牌、变换位置的意思。但实际上这条指令可把来源矢量的某些分量组合，复制到目标矢量的分量。

```
    __m128 result = _mm_add_ps(xMrow1, yMrow2);
    result        = _mm_add_ps(result, zMrow3);
    result        = _mm_add_ps(result, wMrow4);
}
```

这段代码产生以下的中间矢量：

$$\mathtt{xMrow1} = [v_x M_{11} \quad v_x M_{12} \quad v_x M_{13} \quad v_x M_{14}]$$

$$\mathtt{yMrow2} = [v_y M_{21} \quad v_y M_{22} \quad v_y M_{23} \quad v_y M_{24}]$$

$$\mathtt{zMrow3} = [v_z M_{31} \quad v_z M_{32} \quad v_z M_{33} \quad v_z M_{34}]$$

$$\mathtt{wMrow4} = [v_w M_{41} \quad v_w M_{42} \quad v_w M_{43} \quad v_w M_{44}]$$

把这4个中间矢量相加，就能求得结果**r**：

$$\mathbf{r} = \begin{bmatrix} v_x M_{11} + v_y M_{21} + v_z M_{31} + v_w M_{41} \\ v_x M_{12} + v_y M_{22} + v_z M_{32} + v_w M_{42} \\ v_x M_{13} + v_y M_{23} + v_z M_{33} + v_w M_{43} \\ v_x M_{14} + v_y M_{24} + v_z M_{34} + v_w M_{44} \end{bmatrix}^{\mathrm{T}}$$

对某些CPU来说，以上代码还可以进一步优化，方法是使用相对简单的**乘并加**（multiply-and-add）指令，通常表示为madd。此指令把前两个参数相乘，再把结果和第3个参数相加。可惜SSE并不支持madd指令，但我们可以用宏代替它，效果也不错：

```
#define _mm_madd_ps(a, b, c) \
    _mm_add_ps(_mm_mul_ps((a), (b)), (c))

__m128 mulVectorMatrixFinal(__m128 v,
    __m128 Mrow1, __m128 Mrow2, __m128 Mrow3, __m128 Mrow4)
{
    __m128 result;
    result = _mm_mul_ps (_mm_replicate_x_ps(v), Mrow1);
    result = _mm_madd_ps(_mm_replicate_y_ps(v), Mrow2, result);
    result = _mm_madd_ps(_mm_replicate_z_ps(v), Mrow3, result);
    result = _mm_madd_ps(_mm_replicate_w_ps(v), Mrow4, result);
    return result;
}
```

当然，矩阵对矩阵的乘法也可以用类似方法实现。对于微软Visual Studio编译器提供的所有SSE内部函数，可参阅MSDN。

4.8 产生随机数

随机数（random number）在游戏引擎中无处不在。因此，本节主要介绍两个最常见的随机数产生器：线性同余产生器和梅森旋转算法。我们可以看到，随机数产生器所产生的序列仅仅是非常复杂而已，这些序列其实是完全确定性的（deterministic）。因此，这些序列称为**伪随机**（pseudo-random）序列[47]。随机数产生器的好坏，在于其产生多少个数字之后会重复（即序列的**周期**/period），以及该序列在多个著名测试中的表现。

4.8.1 线性同余产生器

线性同余产生器（linear congruential generator，LCG）可以很简捷地产生伪随机序列。有些平台会使用此算法来实现标准C语言库的`rand()`函数。然而，实际情况在各平台上可能有所不同，因此不要认为`rand()`总会基于某一特定算法。若要确定的算法，最好是实现自己的随机数产生器。

LCG在《C数值算法（Numerical Recipes in C）》中有详细介绍，本书不做深入讨论。

笔者想指出的是，LCG并不能产生特别高质量的伪随机序列。若给定相同的初始种子值，则产生的序列会完全相同[48]。LCG产生的序列并不符合一些广泛接受的准则，比如长周期、高低位有接近的长周期、产生的值在序列上和空间上都无关联性。

4.8.2 梅森旋转算法

梅森旋转（Mersenne Twister，MT）[49]伪随机产生器的算法是特别为改进LCG的众多问题而设计的。以下是维基百科对MT优点的描述。

1. MT设计成有庞大的周期：$2^{19937}-1$（MT的始创人证明了此特性）。在实际应用中，只在很少情况下需要更长的周期，因为大部分应用都不需要2^{19937}个唯一组合（$2^{19937}\approx 4.3\times 10^{6001}$）。
2. MT有非常高阶的均匀分布维度（dimensional equidistribution）。这是指，输出序列里的连续数字，其序列关联性微不足道。
3. MT通过了多个统计随机性的测试，包括严格的Diehard测试。
4. MT很快。

---

[47]译注：伪随机数产生器（pseudo-random number generator）的常见缩写是PRNG。

[48]译注：这其实是所有确定性算法的特点，有时候也是重要的需求。这句和上一句关于随机数列的质量无关。

[49]译注：MT是由松本真和西村拓士于1997年发表的。梅森素数是2^n-1形式的素数，$2^{19973}-1$是第24个梅森素数。

网上有多个MT的实现，其中有一个特别酷、采用SIMD矢量指令做进一步优化的**SFMT**（SIMD-oriented Fast Mersenne Twister），可在官网下载[50]。

4.8.3 所有之母及Xorshift

因开发Diehard随机性测试组[51]而闻名的计算机科学家和数学家乔治·马尔萨利亚（George Marsaglia，1924—2011），于1994年发表了一个伪随机数产生器算法，此算法和MT相比，更易实现而且运行得更快。他声称此算法能产生的32位数列，其不重复周期为2^{250}。此算法通过所有Diehard测试，并且仍是当今为高速应用而设计的最佳伪随机数产生器之一。设计者把此算法称为**所有伪随机数产生器之母**（mother of all pseudo-random number generator），可见设计者认为此算法是所有人对PRNG的唯一所需。

之后，Marsaglia发布了另一个产生器Xorshift，其随机性介乎MT和所有之母之间，但运行速度稍快于所有之母。

读者可在维基百科找到有关马尔萨利亚的信息[52]，所有之母产生器可在两个网站找到相关信息[53,54]，Xorshift的PDF文章可于网站[55]下载。

---

[50] http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/SFMT/index.html

[51] http://www.stat.fsu.edu/pub/diehard

[52] http://en.wikipedia.org/wiki/George_Marsaglia

[53] ftp://ftp.forth.org/pub/C/mother.c

[54] http://www.anger.org/random

[55] http://www.jstatsoft.org/v08/i14/paper

第二部分

低阶引擎系统

第5章　游戏支持系统

每个游戏都需要一些底层支持系统，以管理一些例行却关键的任务。例如启动及终止引擎、存取（多个）文件系统、存取各种不同资产类型（网格、纹理、动画、音频等），以及为游戏团队提供调试工具。本章重点讨论多数游戏引擎中都会出现的底层支持系统。后续章节会探索一些较大型的核心系统，包括资源管理、人体学接口设备及游戏内置调试工具。

5.1　子系统的启动和终止

游戏引擎是复杂软件，由多个互相合作的子系统结合而成。当引擎启动时，必须依次配置及初始化每个子系统。各子系统间的相互依赖关系，隐含地定义了每个子系统所需的启动次序。例如子系统B依赖子系统A，那么在启动B之前，必须先启动A。各子系统的终止通常会采用反向次序，即先终止B，再终止A。

5.1.1　C++的静态初始化次序（是不可用的）

由于多数新式游戏引擎皆采用C++为编程语言，我们应考虑一下，C++原生的启动及终止语意是否可做启动及终止引擎子系统之用。C++中，在调用程序进入点（`main()`或Windows下的`WinMain()`）之前，全局及静态对象已被构建。然而，我们完全不可预知这些构造函数的调用次序[1]。在`main()`或`WinMain()`结束返回之后，会调用全局及静态对象的析构函数，而这些函数的调用次序也是无法预知的。显而易见，此C++行为并不适合用来初始化及终止游戏引擎的子系统，实际上，对任何含互相依赖全局对象的软件都不适合。

这事实令人遗憾，因为要实现各主要子系统，例如游戏引擎中的子系统，常见的设计模式是为每个子系统定义**单例类**（singleton class）（通常称作**管理器**/manager）。若C++能

[1]译注：在GCC中可使用`init_priority()`属性去设定变量的初始化次序。

给予我们更多控制能力，指明全局或静态实例的建构、析构次序，那么我们就可以把单例定义为全局变量，而不必使用动态内存分配。例如，各子系统可写成以下形式：

```
class RenderManager
{
public:
    RenderManager()
    {
        // 启动管理器……
    }

    ~RenderManager()
    {
        // 终止管理器……
    }
};

// 单例实例
static RenderManager gRenderManager;
```

可惜，由于没法直接控制建构、析构次序，此方法行不通。

5.1.1.1 按需建构

对付此问题，可使出一个C++的花招：函数内声明的静态变量并不会于main()之前建构，而是在第一次调用该函数时才建构。因此，若把全局单例改为静态变量，我们就可以控制全局单例的建构次序。[2]

```
class RenderManager
{
public:
    // 取得唯一实例
    static RenderManager& get()
    {
        // 此函数中的静态变量将于函数被首次调用时建构
        static RenderManager sSingleton;
        return sSingleton;
    }

    RenderManager()
```

[2]译注：这称作Meyers单例，沿于Scott Meyers的《More Effective C++》。

```
    {
        // 对于需依赖的管理器，先通过调用它们的get()启动它们
        VideoManager::get();
        TextureManager::get();

        // 现在启动渲染管理器
        // ……
    }

    ~RenderManager()
    {
        // 终止管理器
    }
};
```

你会发现，许多软件工程教科书都会建议此方法，或以下这种含动态分配单例的变种：

```
static RenderManager& get()
{
    static RenderManager* gpSingleton = NULL;
    if (gpSingleton == NULL)
    {
        gpSingleton = new RenderManager;
    }
    ASSERT(gpSingleton);
    return *gpSingleton;
}
```

遗憾的是，此方法不可控制析构次序。例如，在RenderManager析构之前，其依赖的单例可能已被析构。而且，很难预计RenderManager单例的确切建构时间，因为第一次调用RenderManager::get()时，单例就会进行建构，天知道那是什么时候！此外，使用该类的程序员可能不会预期，貌似无伤大雅的get()函数可能会有很高的开销，例如，分配及初始化一个重量级的单例。此法仍是难以预计且危险的设计。这促使我们诉诸更直接、有更大控制权的方法。

5.1.2 行之有效的简单方法

假设我们对于子系统继续采用单例管理器的概念。最简单的“蛮力”方法就是，明确地为各单例管理器类定义启动和终止函数。这些函数取代了建构和析构函数，实际上，我们会让建构和析构函数**完全不做任何事情**。这样的话，就可以在main()中（或某个管理整个引擎的单例中），按所需的**明确次序**调用各启动和终止函数。例如：

```
class RenderManager
{
public:
    RenderManager()
    {
        // 不做事情
    }

    ~RenderManager()
    {
        // 不做事情
    }

    void startUp()
    {
        // 启动管理器
    }

    void shutDown()
    {
        // 终止管理器
    }
};

class PhysicsManager    { /* 类似内容 …… */ };
class AnimationManager  { /* 类似内容 …… */ };
class MemoryManager     { /* 类似内容 …… */ };
class FileSystemManager { /* 类似内容 …… */ };

// ……

RenderManager        gRenderManager;
PhysicsManager       gPhysicsManager;
AnimationManager     gAnimationManager;
TextureManager       gTextureManager;

VideoManager         gVideoManager;
MemoryManager        gMemoryManager;
FileSystemManager    gFileSysteManager;

// ……

int main(int argc, const char* argv)
{
    // 以正确次序启动各引擎系统
```

```
    gMemoryManager.startUp();
    gFileSystemManager.startUp();
    gVideoManager.startUp();
    gTextureManager.startUp();
    gRenderManager.startUp();
    gAnimationManager.startUp();
    gPhysicsManager.startUp();
    // ……

    // 运行游戏
    gSimulationManager.run();

    // 以反向次序终止各引擎系统
    // ……
    gPhysicsManager.shutDown();
    gAnimationManager.shutDown();
    gRenderManager.shutDown();
    gTextureManager.shutDown();
    gVideoManager.shutDown();
    gFileSystemManager.shutDown();
    gMemoryManager.shutDown();

    return 0;
}
```

此法还有“更优雅的”实现方式。例如，可以让各管理器把自己登记在一个全局的优先队列（priority queue），之后再按恰当次序逐一启动所有管理器。此外，也可以通过每个管理器列举其依赖的管理器，定义一个管理器间的依赖图（dependency graph），然后按互相依赖关系计算最优的启动次序。上节提及的按需建构也是可行方式[3]。根据笔者经验，蛮力方法总优于其他方法，因为：

- 此方法既简单又容易实现。
- 此方法是明确的。看看代码就能立即得知启动次序。
- 此方法容易调试和维护。若某子系统启动时机不够早或过早，只需移动一行代码。

用蛮力方法手动启动及终止子系统，还有一个小缺点，就是程序员有可能意外地终止一些子系统，而非按启动之相反次序。但这一缺点并不会使笔者失眠，只要能成功启动及终止引擎的各子系统，你的任务就完成了。

[3]译注：要解决按需建构方式的析构次序问题，可以在单例建构时，把自己登记在一个全局堆栈中，在main()结束之前，逐一把堆栈弹出并调用其终止函数。此方法假设单例的终止次序可以为启动次序的相反，但理论上不能解决所有情况。

5.1.3 一些实际引擎的例子

看看一些来自实际引擎的启动、终止例子。

5.1.3.1 OGRE

OGRE的作者承认OGRE本质上是渲染引擎而非游戏引擎。但它也必须提供许多完整游戏引擎都有的底层功能，包括一个简单优雅的启动/终止机制。OGRE中的一切对象都由Ogre::Root单例控制。此单例含有指向其他OGRE子系统的指针，并负责启动和终止这些子系统。此设计使程序员能轻松启动OGRE，只需new一个Ogre::Root实例就可以了。

以下是一些OGRE的代码片段，从中可以理解其工作方式：

OgreRoot.h

```
class _OgreExport Root : public Singleton<Root>
{
    // <忽略一些代码……>

    // 各单例
    LogManager* mLogManager;
    ControllerManager* mControllerManager;
    SceneManagerEnumerator* mSceneManagerEnum;
    SceneManager* mCurrentSceneManager;
    DynLibManager* mDynLibManager;
    ArchiveManager* mArchiveManager;
    MaterialManager* mMaterialManager;
    MeshManager* mMeshManager;
    ParticleSystemManager* mParticleManager;
    SkeletonManager* mSkeletonManager;
    OverlayElementFactory* mPanelFactory;
    OverlayElementFactory* mBorderPanelFactory;
    OverlayElementFactory* mTextAreaFactory;
    OverlayManager* mOverlayManager;
    FontManager* mFontManager;
    ArchiveFactory* mZipArchiveFactory;
    ArchiveFactory* mFileSystemArchiveFactory;
    ResourceGroupManager* mResourceGroupManager;
    ResourceBackgoundQueue* mResourceBackgroundQueue;
    ShadowTextureManager* mShadowTextureManager;
    // 等等
};
```

OgreRoot.cpp

```
Root::Root(const String& pluginFileName,
           const String& configFileName,
           const String& logFileName) :
        mLogManager(0),
        mCurrentFrame(0),
        mFrameSmoothingTime(0.0f),
        mNextMovableObjectTypeFlag(1),
        mIsInitiailised(false)
{
    // 基类会检查单例
    // 初始化
    mActiveRenderer = 0;
    mVersion = StringConverter::toString(OGRE_VERSION_MAJOR) + "."
             + StringConverter::toString(OGRE_VERSION_MINOR) + "."
             + StringConverter::toString(OGRE_VERSION_PATCH)
             + OGRE_VERSION_SHUFFIX + " "
             + "(" + OGRE_VERSION_NAME + ")";
    mConfigFileName = configFileName;

    // 若没有日志管理员，建立日志管理员及默认日志文件
    if (LogManager::getSingletonPtr() == 0)
    {
        mLogManager = new LogManager();
        mLogManager->createLog(logFileName, true, true);
    }

    // 动态库管理员
    mDynLibManager = new DynLibManager();
    mArchiveManager = new ArchiveManager();
    // 资源群管理员
    mResourceGroupManager = new ResourceGroupManager();
    // 资源背景队列
    mResourceBackgrounQueue = new ResourceBackgroundQueue();
    // 等等
}
```

OGRE提供一个Ogre::Singleton模板基类，所有管理器都派生自此基类。在Ogre::Singleton的实现里，并不会进行延迟构建，而是依赖于Ogre::Root明确地new每个单例。如上所述，这样可以确定每个单例会以定义的明确次序创建及毁灭这些单例。[4]

[4]译注：此方法也有一个缺点，扩展引擎时须更改Ogre::Root的代码。此违反了开闭原则（open-closed principle），尤其会影响闭源引擎的可扩展性。

5.1.3.2 顽皮狗的《神秘海域：德雷克船长的宝藏》

顽皮狗开发的《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》引擎采用一个相似的明确方法去启动和终止各子系统。读者会发现，以下的引擎启动代码并非总是一串简单的单例分配。许多不同的操作系统服务、第三方库等，都必须在引擎初始化时启动。并且在可行情况下，代码中会尽量避免动态内存分配，所以许多单例是静态分配的对象（如g_fileSystem、g_languageMgr等）。这段代码不一定好看，但总可以完成任务。

```
Err BigInit()
{
    init_exception_handler();

    U8* pPhysicsHeap = new (kAllocGlobal, kAlign16)
        U8[ALLOCATION_GLOBAL_PHYS_HEAP];
    PhysicsAllocatorInit(pPhysicsHeap, ALLOCATION_GLOBAL_PHYS_HEAP);

    g_texDb.Init();
    g_textSubDb.Init();
    g_spuMgr.Init();

    g_drawScript.InitPlatform();

    PlatformUpdate();

    thread_t init_thr;
    thread_create(&init_thr, threadInit, 0, 30, 64*1024, 0, "Init");

    char masterConfigFileName[256];
    snprintf(masterConfigFileName, sizeof(masterConfigFileName),
        MASTER_CFG_PATH);
    {
        Err err = ReadConfigFromFile(masterConfigFileName);
        if (err.Failed())
        {
            MsgErr("Config file not found (%s).\n",
                masterConfigFileName);
        }
    }

    memset(&g_discInfo, 0, sizeof(BootDiscInfo));
    int err1 = GetBootDiscInfo(&g_discInfo);
    Msg("GetBootDiscInfo()" : 0x%x\n", err1);
    if (err1 == BOOTDISCINFO_RET_OK)
```

```
    {
        printf("titleId        : [%s]\n", g_discInfo.titleId);
        printf("parentalLevel : [%d]\n", g_discInfo.parentalLevel);
    }

    g_fileSystem.Init(g_gameInfo.m_onDisc);
    g_languageMgr.Init();
    if (g_shouldQuit) return Err::kOK;
    // 等等
}
```

5.2 内存管理

游戏程序员总希望把代码变得更快。任何软件的效能，不仅受算法的选择和算法编码的效率所支配，程序如何运用内存（RAM）也是重要因素。内存对效能的影响有两方面。

1. 以`malloc()`或C++的全局`new`运算符进行**动态内存分配**（dynamic memory allocation），是非常慢的操作。要提升效能，最佳方法是尽量避免动态分配内存，不然也可利用自制的内存分配器来大大减低分配成本。
2. 许多时候在现代的CPU上，软件的效能受其**内存访问模式**（memory access pattern）主宰。我们将看到，把数据置于细小**连续**的内存块，相比把数据分散至广阔的内存地址，CPU对前者的操作会高效得多。就算采用最高效的算法，并且极小心地编码，若其操作的数据并非高效地编排于内存中，算法的效能也会被搞垮。

本节会介绍如何从这两个方向优化内存的运用。

5.2.1 优化动态内存分配

通过`malloc()/free()`或C++的全局`new/delete`运算符动态分配内存——又称为**堆分配**（heap allocation）——通常是非常慢的。低效主要来自两个原因。首先，堆分配器（heap allocator）是通用的设施，它必须处理任何大小的分配请求，从1字节至1000兆字节亦然。这需要大量的管理开销，导致`malloc()/free()`函数变得缓慢。其次，在多数操作系统上，`malloc()/free()`必然会从用户模式（user mode）切换至内核模式（kernel mode），处理请求，再切换至原来的程序。这些上下文切换（context-switch）可能会耗费非常多的时间。因此，游戏开发中一个常见的经验法则是：

维持最低限度的堆分配，并且永不在紧凑循环中使用堆分配

当然，任何游戏引擎都无法完全避免动态内存分配，所以多数游戏引擎会实现一个或多个定制分配器（custom allocator）。定制分配器能享有比操作系统分配器更优的性能特征，原因有二。第一，定制分配器从预分配的内存中完成分配请求（预分配的内存来自malloc()、new，或声明为全局变量）。这样，分配过程都在用户模式下执行，完全避免了进入操作系统的上下文切换。第二，通过对定制分配器的使用模式（usage pattern）做出多个假设，定制分配器便可以比通用的堆分配器高效得多。

以下几节中，我们会看看几类常见的定制分配器。关于本题目的更多信息，可参阅Christian Gyrling的杰出博文[5]。

5.2.1.1 基于堆栈的分配器

许多游戏会以堆栈般的形式分配内存。当载入游戏关卡时，就会为关卡分配内存；关卡载入后，就会很少甚至不会动态分配内存。在玩家完成关卡之际，关卡的数据会被卸下，所有关卡占用的内存也可被释放。对于这类内存分配，非常适合采用堆栈形式的数据结构。

堆栈分配器（stack allocator）是非常容易实现的。我们要分配一大块连续内存，可简单地使用malloc()、全局new，或是声明一个全局字节数组（最后的方法，实际上会从可执行文件的BSS段里分配内存）。另外要安排一个指针指向堆栈的顶端，指针以下的内存是已分配的，指针以上的内存则是未分配的。对于每个分配请求，仅需把指针往上移动请求所需的字节数量。要释放最后分配的内存块，也只需要把指针向下移动该内存块的字节数量。

必须注意，使用堆栈分配器时，不能以任意次序释放内存，必须以分配时相反的次序释放内存。有一个方法可简单地实施此限制，这就是完全不容许释放个别的内存块。取而代之，我们提供一个函数，该函数可以把堆栈顶端指针回滚至之前标记了的位置，那么其实际上的意义就是，释放从回滚点至目前堆栈顶端之间的所有内存。

回滚顶端指针的时候，回滚的位置必须位于两个分配而来的内存块之间的边界，否则，写入新分配的内存时，会重写进之前最高位置内存块的末端。为保证能正确地回滚指针，堆栈分配器通常提供一个函数，该函数传回一个**标记**（marker），代表目前堆栈的顶端。而回滚函数则使用这个标记作为参数。图5.1展示了此过程。通常堆栈分配器的接口类似这样：

[5] http://www.swedishcoding.com/2008/08/31/are-we-out-of-memory

```
class StackAllocator
{
public:
    // 堆栈标记：表示堆栈的当前顶端
    // 用户只可以回滚至一个标记，而不是堆栈的任意位置
    typedef U32 Marker;

    // 给定总大小，构建一个堆栈分配器
    explicit StackAllocator(U32 stackSize_bytes);

    // 给定内存块大小，从堆栈顶端分配一个新的内存块
    void* alloc(U32 size_bytes);

    // 取得指向当前堆栈顶端的标记
    Marker getMarker();

    // 把堆栈回滚至之前的标记
    void freeToMarker(Marker marker);

    // 清空整个堆栈 (把堆栈归零)
    void clear();

private:
    // ......
};
```

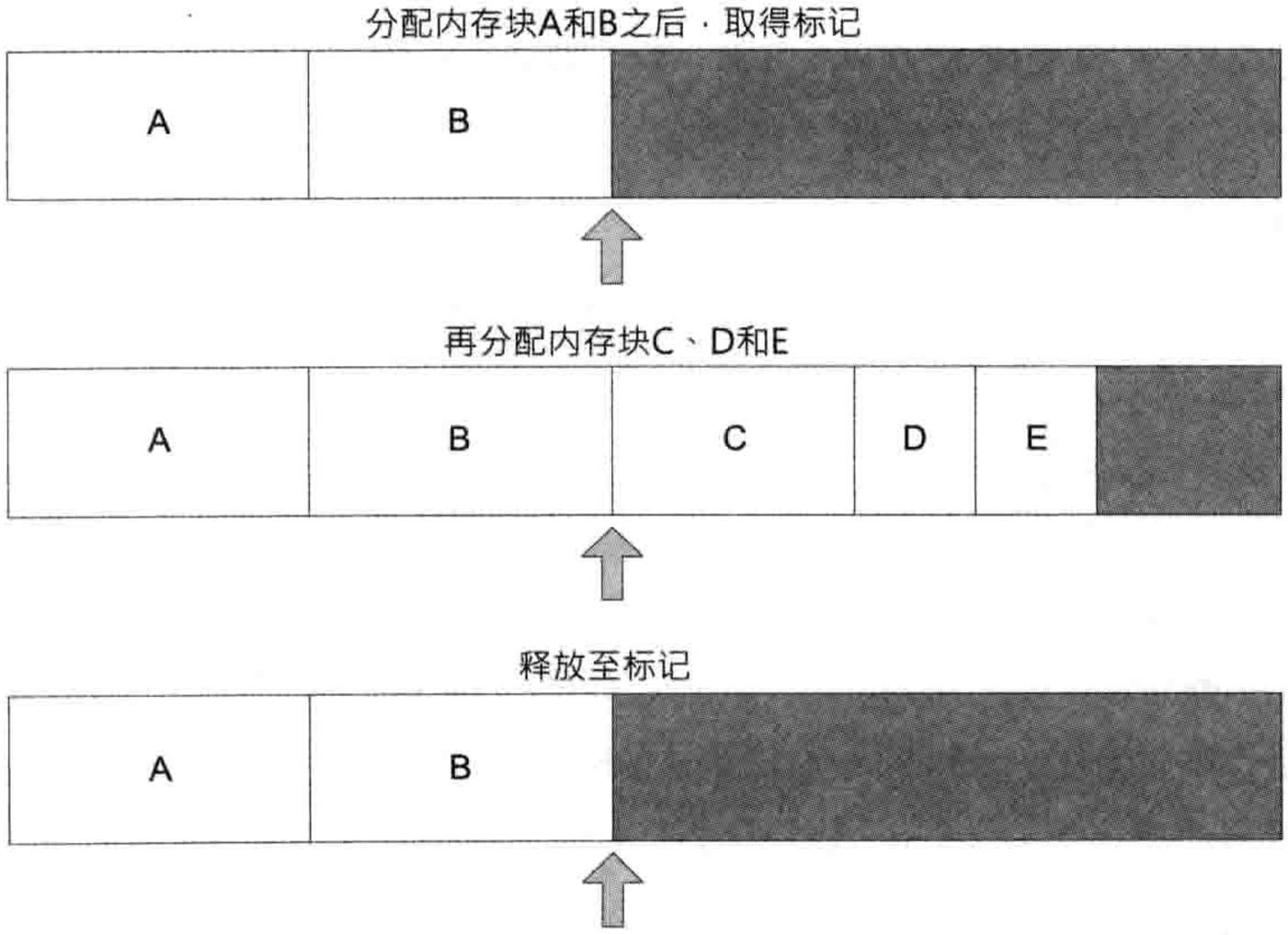


图 5.1: 向堆栈分配，以及释放至一个标记。

双端堆栈分配器

一块内存其实可以给两个堆栈分配器使用，一个从内存块的底端向上分配，另一个从内存块的顶端向下分配。双端堆栈分配器（double-ended stack allocator）很实用，因为它容许权衡底端堆栈和顶端堆栈的使用，使它更有效地运用内存。某些情况下，两个堆栈使用差不多相等的内存，那么两个堆栈指针大约会接近内存的中间。其他情况下，其中一个堆栈可能占用大部分的内存空间，但只要总共的分配总量不大于两个堆栈共享的内存块，则仍然可以满足所有分配要求。图5.2说明了这种情况。

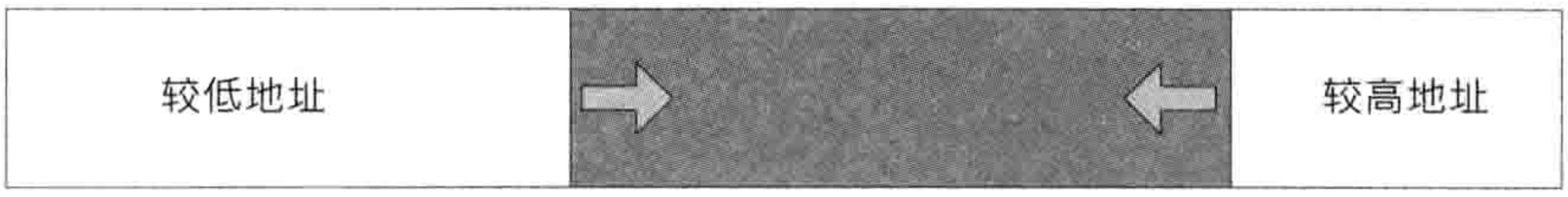


图 5.2: 一个双端堆栈分配器。

在Midway的《Hydro Thunder》街机游戏里，所有内存都是分配自单个巨大内存块，以双端堆栈分配器管理的。底堆栈用来载入及卸下游戏关卡（跑道）；而顶堆栈则用来分配临时内存块，这些临时内存会在每帧中分配及释放。此分配方案非常优异，保证《Hydro Thunder》从不会产生内存碎片问题（见5.2.1.4节）。《Hydro Thunder》的首席工程师Steve Ranck在[6]的第1.9节描述了此分配技巧。

5.2.1.2 池分配器

在游戏引擎编程（及普遍的软件工程）中，常会分配大量同等尺寸的小块内存。例如，我们可能要分配及释放矩阵、迭代器（iterator）、链表中的节点、可渲染的网格实例等。**池分配器**（pool allocator）是此类分配模式的完美选择。

池分配器的工作方式如下。首先，池分配器会预分配一大块内存，其大小刚好是分配元素的倍数。例如，4×4矩阵池的大小设为64字节的倍数（每矩阵16个元素，再乘以每元素4字节）。池内每个元素会加到一个存放自由元素的链表；换句话说，在对池进行初始化时，自由列表（free list）包含所有元素。池分配器收到分配请求时，就会把自由链表的下一个元素取出，并传回该元素。释放元素之时，只需简单地把元素插回自由链表中。分配和释放都是$O(1)$的操作。这是因为无论池内有多少个元素，每个操作都只需几个指针运算。（$O(1)$是Big-O表示法的例子。此例子代表分配和释放操作的执行时间大概是常数，和池中的元素数量无关。5.3.3节会解释Big-O表示法。）

储存自由元素的链表可实现为单链，即每个自由元素需要储存一个指针（在多数机器上是4字节）。我们应该如何取得储存这些指针的内存呢？无疑我们可以再预分配另一块内存储

存这些指针，这个内存块的大小为（sizeof(void*)×池里元素数量）字节。然而，这太浪费了。只需要意识到，自由列表内的内存块，按定义来说就是可用的内存。那为什么不用这些内存本身来储存自由列表的“next”指针呢？只要元素尺寸≥sizeof(void*)，就可以使用这个小诀窍了。

若元素尺寸小于指针，则可以使用池元素的索引代替以指针去实现链表。例如，若池是用来存放16位整数的，那么便可在链表中使用16位索引作为“next指针”。只要池里不超过$2^{16} = 65,536$个元素便没问题。

5.2.1.3 含对齐功能的分配器

我们在3.2.5.1节提及，每个变量和数据对象都有对齐要求。8位整数可对齐至任何地址，但32位整数或浮点变量则必须4字节对齐，即其地址的最低有效半字节必须为0x0、0x4、0x8或0xC。128位SIMD矢量值通常需要16字节对齐，即其地址的最低有效半字节只能为0x0。在PS3上，使用直接内存访问（direct memory access，DMA）控制器传送数据到SPU时，128字节对齐的内存能获得最大DMA吞吐量，即其地址的最低有效字节（LSB）为0x00或0x80。

所有内存分配器都必须能传回对齐的内存块。要实现这个功能十分容易。只要在分配内存时，分配比请求所需多一点的内存，再向上调整其内存地址至适当的对齐，最后传回调整后的地址。由于我们分配了多一点的内存，即使把地址往上调整，传回的内存块仍够大。

在大多数的实现中，额外分配的字节等于对齐字节。例如，若请求为16字节对齐的内存块，就可以额外分配多16字节。最坏的情况下要把地址往上移动15字节。多出的1字节，促使我们在任何情况都可使用相同的计算，就算原来分配的内存已符合对齐。这样做简化并加速了代码，其代价是每次分配浪费1字节。另一方面，我们下面会看到，需要这些多出的字节，以储存释放内存时所需的额外信息。

计算调整偏移量的方法如下。首先用掩码（mask）把原本内存块地址的最低有效位取出，再把期望的对齐减去此值，结果就是调整偏移量。对齐应该总是**2的幂**（通常是4或16字节），因此要计算掩码，只要把对齐减1就行了。例如，若请求16字节对齐的内存块，掩码就是$(16 - 1) = 15 =$ 0x0000000F。把未对齐的地址与掩码进行位并（bitwise AND）操作，就可得到错位（misalignment）的字节数目。例如，如果原来分配到的内存块地址为0x50341233，位并掩码0x0000000F后，得出0x00000003，即还差3字节才能对齐。要把这个地址对齐，只要加上(对齐字节 − 错位字节) $= (16 - 3) = 13 =$ 0xD字节。因此，最终对齐地址为0x50341233+0xD = 0x50341240。

以下是实现对齐内存分配器的其中一个方式：

```
// 对齐分配函数。注意:“alignment”必须为2的幂 (一般是4或16)
void *allocateAligned(U32 size_bytes, U32 alignment)
{
    // 计算总共要分配的内存量
    U32 expandedSize_bytes = size_bytes + alignment;

    // 分配未对齐的内存块，并转换为U32类型
    U32 rawAddress = (U32)allocateUnaligned(expandedSize_bytes);

    // 使用掩码去除地址低位部分，计算“错位”量，从而计算调整量
    U32 mask = (alignment - 1);
    U32 misalignment = (rawAddress & mask);
    U32 adjustment = alignment - misalignment;

    // 计算调整后的地址，并把它以指针类型返回
    U32 alignedAddress = rawAddress + adjustment;
    return (void*)alignedAddress;
}
```

当稍后要释放此内存块时，代码会传给分配器调整后的地址，而非原本我们分配的地址。那么，怎样才可以释放原本分配的内存呢？我们要找到某种方法，把调整后的地址转换回原本的、可能未对齐的地址。

要完成这个转换，我们可以储存一些元信息（meta-information）至额外分配的内存，这些内存原来只是做对齐之用。最少的调整量为1字节，这1字节足够储存偏移量（因为偏移量永不会超过256）。我们可以把偏移量储存至调整后地址之前的1字节（无论偏移量为多少字节），这么做，就可以简单地从调整后地址取回偏移量，并计算原本的地址。以下是修改后的allocateAligned()函数：

```
// 对齐分配函数。注意:“alignment”必须为2的幂 (一般是4或16)
void* allocateAligned(U32 size_bytes, U32 alignment)
{
    // 若alignment == 1，使用方必须调用
    // allocateUnaligned()及freeUnaligned()
    ASSERT(alignment > 1);

    // 计算总共要分配的内存量
    U32 expandedSize_bytes = size_bytes + alignment;

    // 分配未对齐的内存块，并转换为U32类型
    U32 rawAddress = (U32)allocateUnaligned(expandedSize_bytes);
```

```
    // 使用掩码去除地址低位部分，计算“错位”量，从而计算调整量
    U32 mask = (alignment - 1);
    U32 misalignment = (rawAddress & mask);
    U32 adjustment = alignment - misalignment;

    // 计算调整后的地址，并把它以指针类型返回
    U32 alignedAddress = rawAddress + adjustment;

    // 把alignment储存在调整后地址的前4字节
    U32* pAdjustment = (U32*)(alignedAddress - 4);
    *pAdjustment = adjustment;

    return (void*)alignedAddress;
}
```

而对应的freeAligned()函数可实现如下：

```
void freeAligned(void* p)
{
    U32 alignedAddress = (U32)p;
    U8* pAdjustment = (U8*)(alignedAddress - 4);
    U32 adjustment = (U32)*pAdjustment;

    U32 rawAddress = alignedAddress - adjustment;

    freeUnaligned((void*)rawAddress);
}
```

5.2.1.4 单帧和双缓冲内存分配器

几乎所有游戏都会在游戏循环中分配一些临时用数据。这些数据要么可在循环迭代结束时丢弃，要么可在下一迭代结束时丢弃。很多游戏引擎都支持这两种分配模式，分别称为**单帧分配器**（single-frame allocator）和**双缓冲分配器**（double-buffered allocator）。

单帧分配器

要实现单帧分配器，先预留一块内存，并以前文所述的简单堆栈分配器管理。在每帧开始时，都把堆栈的顶端指针重置到内存块的底端地址。在该帧中，分配要求会使堆栈向上成长。此过程不断重复。

```
StackAllocator g_singleFrameAllocator;

// 主游戏循环
while (true)
{
    // 每帧清除单帧分配器的缓冲区
    g_singleFrameAllocator.clear();

    // ……

    // 从单帧分配器分配内存
    // 我们永不需要手动释放这些内存！但要确保这些内存仅在本帧中使用
    void* p = g_singleFrameAllocator.alloc(nBytes);

    // ……
}
```

单帧分配器的主要益处是，分配了的内存永不用手动释放，我们依赖于每帧开始时分配器会自动清除所有内存。单帧分配器也极其高效。然而，单帧分配器的最大缺点在于，程序员必须有不错的自制能力。程序员需要意识到，从单帧分配器分配的内存块只在目前的帧里有效。程序员**绝不能**把指向单帧内存块的指针跨帧使用！

双缓冲分配器

双缓冲分配器容许在第i帧分配的内存块用于第$(i+1)$帧。实现方法就是建立两个相同尺寸的单帧堆栈分配器，并在每帧交替使用。

```
class DoubleBufferedAllocator
{
    U32            m_curStack;
    StackAllocator m_stack[2];

public:
    void swapBuffers()
    {
        m_curStack = (U32)!m_curStack;
    }

    void clearCurrentBuffer()
    {
        m_stack[m_curStack].clear();
    }
```

```
    void* alloc(U32 mBytes)
    {
        return m_stack[m_curStack].alloc(nBytes);
    }

    // ……
};

DoubleBufferedAllocator g_doubleBufAllocator;

// 主游戏循环
while (true)
{
    // 和之前一样，每帧清除单帧分配器的缓冲区
    g_singleFrameAllocator.clear();

    // 对双缓冲分配器交换现行和无效的缓冲区
    g_doubleBufAllocator.swapBuffers();

    // 清空新的现行缓冲区，保留前帧的缓冲不变
    g_doubleBufAllocator.clearCurrentBuffer();

    // ……

    // 从双缓冲分配器分配内存，不影响前帧的数据
    // 要确保这些内存仅在本帧或次帧中使用
    void* p = g_doubleFrameAllocator.alloc(nBytes);

    // ……
}
```

在多核游戏机，如Xbox 360或PS3上，在缓存非同步处理的结果时，这类分配器极有用。在第i帧，我们可以在某个SPU上启动一个任务，并从双缓冲分配器分配一块内存，给予该任务作为目的缓冲。该任务在第i帧完结之前完成，并把产生的结果写进我们提供的缓冲。在第$(i+1)$帧，两个缓冲互换。那么任务结果的缓冲就会在非活动状态，并不会被本帧进行的双缓冲分配所重写。在第$(i+2)$帧之前，便可安心使用任务结果，不怕数据被重写。

5.2.2 内存碎片

动态堆分配的另一问题在于，会随时间产生**内存碎片**（memory fragmentation）。当程序启动时，其整个堆空间都是自由的。当分配一块内存时，一块合适尺寸的连续内存块便会

被标记为“使用中”，而其余的内存仍然是自由的。当释放内存块时，该内存块便会与相邻的内存块合并，形成单个更大的自由内存块。随着时间的推移，鉴于以随机次序分配及释放不同尺寸的内存块，堆内存开始变成由自由块和使用中块所拼砌而成的拼布模样。我们可视自由区域为使用内存块之间的“洞”。如果洞的数量增多，并且洞的尺寸相对很小，就会称之为**内存碎片状态**，如图5.3所示。

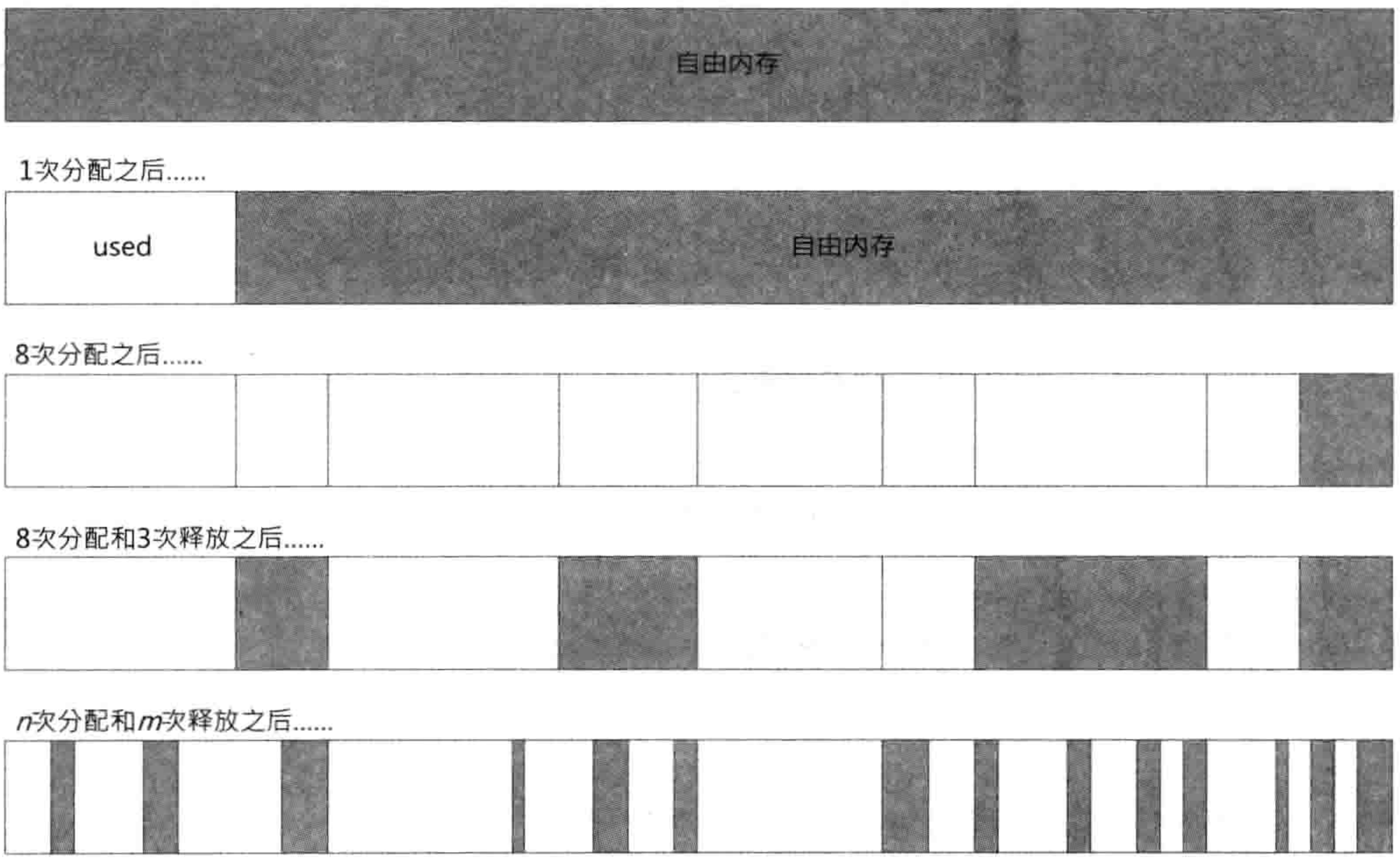


图 5.3: 内存碎片。

内存碎片的问题在于，就算有足够的自由内存，分配请求仍然可能会失败。问题的症结是，分配的内存必须为**连续**的。例如，要满足一个128KB的分配请求，必须有一个自由的“洞”，其尺寸大约要128KB，或更大。若有两个各64KB的洞，虽然总共有足够的字节数，但由于它们并非**连续的字节**，该请求仍会失败。

在支持**虚拟内存**（virtual memory）的操作系统上，内存碎片并非大问题。虚拟内存系统把不连续的物理内存块——每块称为**内存页**（page）——映射至**虚拟地址空间**（virtual address space），使内存页对于应用程序来说，看上去是连续的。在物理内存不足时，久没使用的内存页便会写进磁盘，有需要时再重载到物理内存。关于虚拟内存的运作方式，可参考此课件[6]。多数嵌入式设备并不能负担得起虚拟内存的实现。有些当代的游戏机，虽然技术上能支持虚拟内存，但由于其导致的开销，多数游戏引擎不会使用虚拟内存。

[6] http://lyle.smu.edu/~kocan/7343/fall05/slides/chapter08.ppt

5.2.2.1 以堆栈和池分配器避免内存碎片

使用堆栈和/或池分配器，可以避免一些内存碎片带来的问题。

- 堆栈分配器完全避免了内存碎片的产生。这是由于，用堆栈分配器分配到的内存块总是连续的，并且内存块必然以反向次序释放，如图5.4所示。
- 池分配器也无内存碎片问题。虽然实际上池会产生碎片，但这些碎片不会像一般的堆，提前引发内存不足的情况。向池分配器做分配请求时，不会因缺乏足够大的连续内存块，而造成分配失败，因为池内所有内存块是完全一样大的。图5.5显示了这种情况。

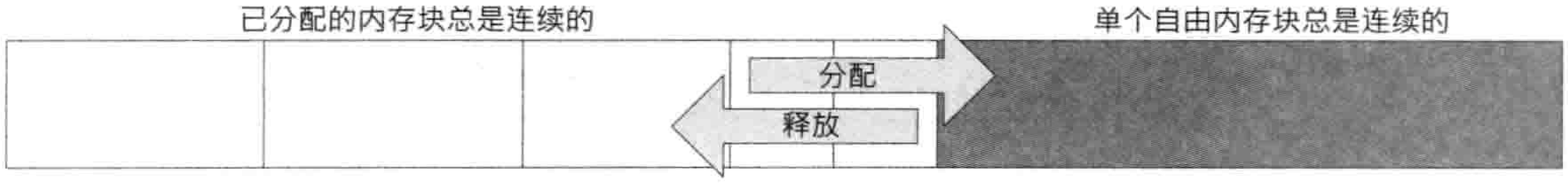


图 5.4: 堆栈分配器完全避免了内存碎片。

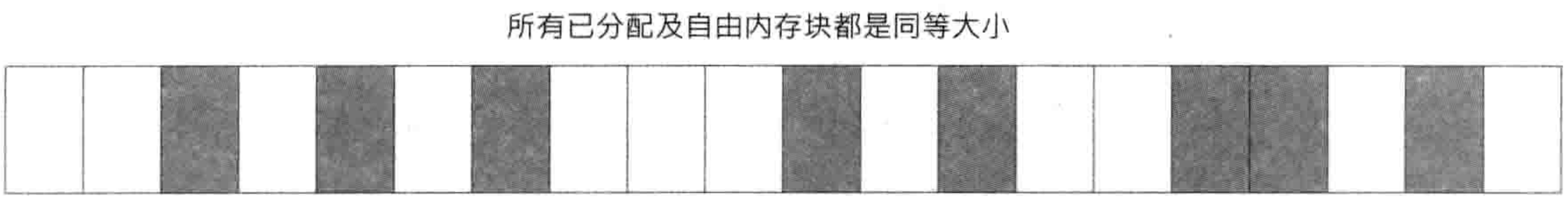


图 5.5: 池分配器不会受内存碎片影响。

5.2.2.2 碎片整理及重定位

若要分配及释放不同大小的对象，并以随机次序进行，那么堆栈和池分配器也不适用。对付这种情况，可以对堆定期进行**碎片整理**（defragmentation）。碎片整理能把所有自由的“洞”合并，其方法是把内存从高位移至低位，也即是把“洞”移至内存的高地址。一个简单的算法是，搜寻第一个“洞”，之后把洞上方紧接的已分配内存块往下移至洞的开始地址。实质上，这会把洞好像气泡一样浮升至内存中较高的地址。若一直进行这个过程，最后所有已分配内存块都会连续地凑在堆内存空间的底端，而所有洞都会浮升至空间的顶端，结合成一块连续的自由空间，如图5.6所示。

按以上介绍的方法，把内存这样移动是简单容易的事情。棘手的**是**，事实上我们移动了**已分配**的内存块，若有**指针**指向这些内存块，移动内存便会使这些指针失效。

其中一个解决方案就是，把指向这些内存块的指针逐一更新，使移动内存块后这些指针能指到新的地址。此过程称为指针**重定位**（relocation）。遗憾的是，在C/C++中并没有方法可以**搜寻**所有指向某地址范围的指针。若要在游戏引擎中支持碎片整理功能，程序员必须

小心手动维护所有指针，在重定位时正确更新指针；另一个选择是，舍弃指针，取而代之，使用更容易重定位时修改的构件，例如**智能指针**（smart pointer）或**句柄**（handle）。

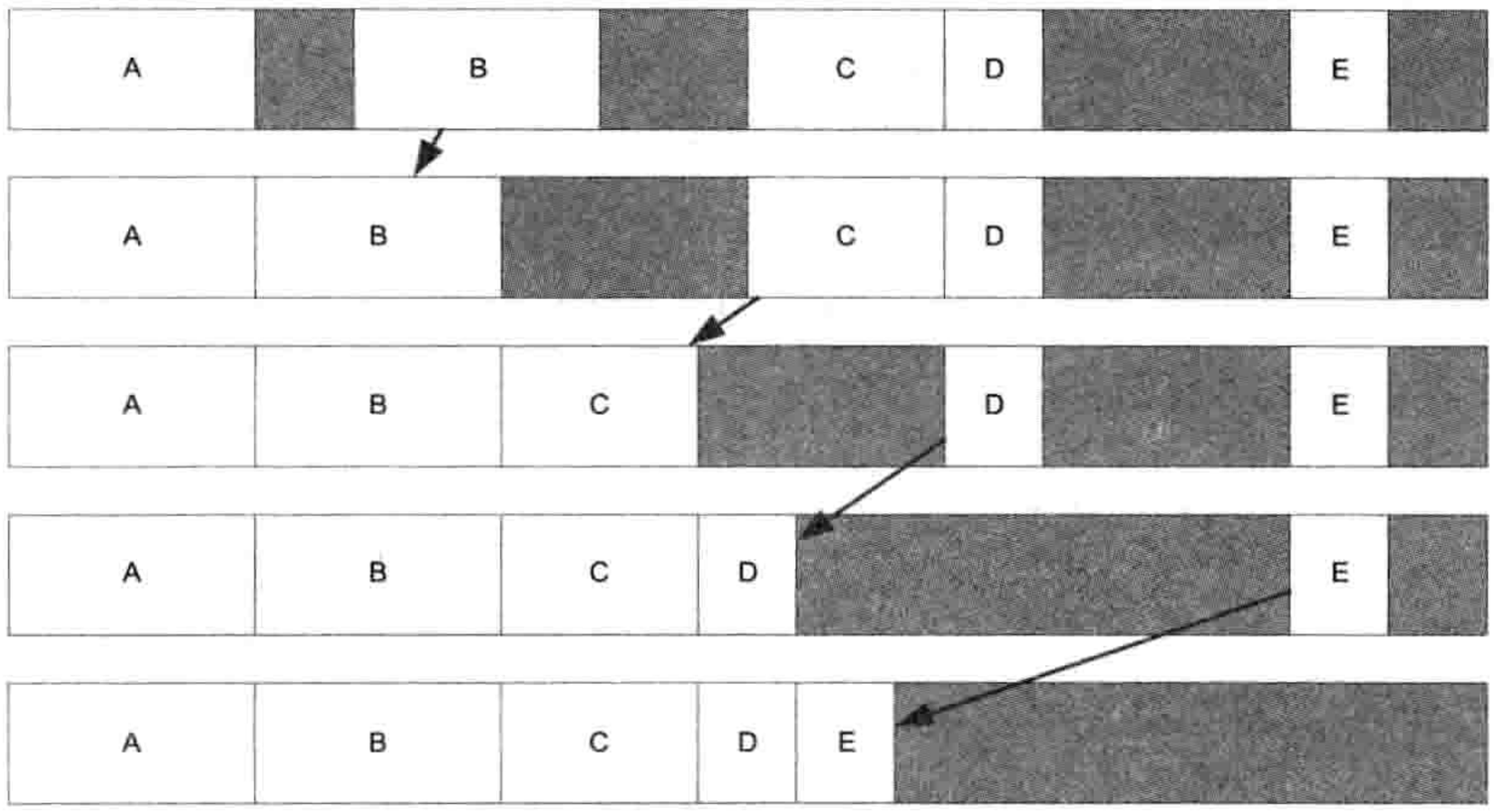


图 5.6: 通过搬移已分配的内存块来整理碎片。

智能指针是细小的类，它包含一个指针，并且其实际行为几乎和普通指针完全相同。但是由于智能指针是用类实现的，可以编写代码正确处理内存重定位。其中一个方法是，让所有智能指针把自己加进一个全局链表里。当要移动某块内存，便可扫描该全局链表，更新每个指向该块内存的智能指针。

句柄通常实现为索引，这些索引指向句柄表内的元素，每个元素储存指针。句柄表本身不能被重定位。当要移动某已分配内存块时，就可以扫描该句柄表，并自动修改相应的指针。由于句柄只是句柄表的索引，无论如何移动内存块，句柄的值都是不变的。因此，使用句柄的对象永不会受内存重定位影响。

重定位的另一难题是，某些内存块可能不能被重定位。例如，若使用第三方库，而该库不使用智能指针或句柄，那么，指向库内数据结构的指针就可能不能被重定位。要解决此问题，最好的办法是，让这些库在另一个特别缓冲区里分配内存，此缓冲区位于可重定位内存范围以外。另一可行选择是，干脆容许一些内存块不能被重定位。若这种内存块数量少且体积小，重定位系统仍可运行得相当好。

有趣的是，顽皮狗的所有引擎皆支持碎片整理。我们会尽可能使用句柄，以避免重定位指针。然而有些情况还是无法避免的，须使用原始指针（raw pointer）。我们需要小心地维护这些指针，当移动内存块时要人手重定位。由于不同原因，几个顽皮狗的游戏对象是不能重定位的。然而，如上所述，这一般不会造成实际问题，因为这种对象数量少，其体积相对整个重定位内存来说也很小。

分摊碎片整理成本

因为碎片整理要复制内存块，所以其操作过程可能很慢。然而，我们无须一次性把碎片完全整理。取而代之，我们可以把碎片整理成本分摊（amortize）至多个帧。我们容许每帧进行多达N次内存块移动，N是个小数目，如8或16。若游戏以每秒30帧运行，那么每帧会持续1/30s（33ms）。这样，堆通常能在少于1s内完全整理所有碎片，而不会对游戏帧率产生明显影响[7]。只要分配及释放的次数低于碎片整理的移动次数，那么堆就会经常保持接近完全整理的状态。

此方法只对细小的内存块有效，使移动内存块的时间短于每帧配给的重定位时间。若要重定位非常大的内存块，有时候可以把它分拆为两个或更多的小块，而每个小块可以独立被重定位。在顽皮狗的引擎中，这并不是问题，因为重定位只应用在游戏对象，而游戏对象一般很小，从不会超过数千字节。

5.2.3 缓存一致性

要了解内存存取模式为何影响效能，我们须先了解现代处理器如何读 / 写内存。存取主系统内存是缓慢的操作，通常需要几千个处理器周期才能完成。和CPU里的寄存器相比，存取寄存器只需数十个周期，甚至有时只需要一个周期。为了降低读 / 写主内存的平均时间，现代处理器会采用高速的内存**缓存**（cache）。

缓存是一种特殊的内存，CPU读 / 写缓存的速度比主内存快得多。内存缓存的基本概念是这样的，当首次读取某区域的主内存，该内存小块会载入高速缓存。这个内存块单位称为**缓存线**（cache line），缓存线通常介乎8至512字节，具体值视微处理器架构而定。若后来再读取内存，而该数据已在缓存中，那么数据就可以直接从缓存载入寄存器，这比读取主内存快得多。仅当要求的数据不在缓存中，才必须存取主内存。这种情况名为**缓存命中失败**（cache miss）。每当出现缓存命中失败，程序便要被逼暂停，等待缓存线自主内存更新后才能继续运行。

写数据到主内存也可应用相似的规则。最简单的缓存写入设计称为**透写式缓存**（write-through cache）。在这种设计中，写入数据到缓存时，会立即把数据同时写入主内存。然而，在另一种称为**回写式**（write-back或copy-back）的缓存设计中，数据会先写在缓存中，在某些情况下才会把缓存线回写到主内存。这些情况包括：一条曾写过新数据的缓存线需要逐出缓存，以自主内存载入新的缓存线；程序明确要求清除缓存。

显而易见，我们无法完全避免缓存命中失败，因为数据始终要在缓存和主内存之间移

[7]译注：若在1帧里进行完整的碎片整理，可能会令游戏片刻停顿，产生“很卡”的不良体验。

动。然而，高效计算的诀窍在于，以最优的方式安排内存中的数据及为算法编码，尽量减少缓存命中失败的次数。以下将会介绍如何达到这个目的。

5.2.3.1 一级及二级缓存

在发展缓存技巧之初，缓存内存是置于主板上的。缓存由比主内存更快、更贵的内存模组构成，从而能达至提升速度之效。然而，当时的缓存内存很昂贵，所以其容量通常很小，约是16KB的数量级。随着缓存技术的演进，发展出更快的缓存内存种类，这种缓存直接置于CPU芯片上。这样产生了两种缓存：在CPU芯片上的**一级**（level 1，L1）缓存、在主板上的**二级**（level 2，L2）缓存。近来，L2缓存也移至CPU芯片上（图5.7）[8]。

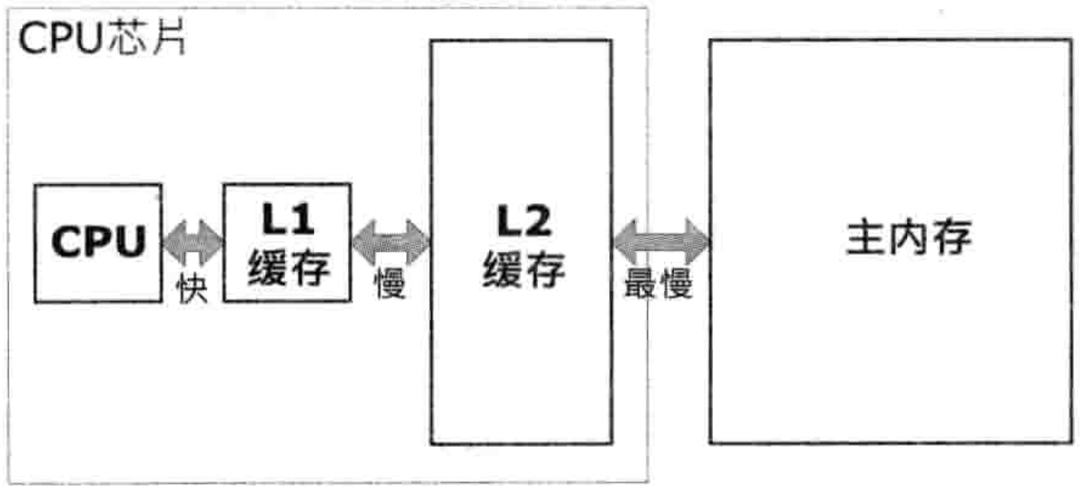


图 5.7: 一级及二级缓存。

由于L2缓存的出现，存取内存的规则变得更复杂。以前存取数据要从主内存经过缓存才能到达CPU（或相反方向），现在则要经过两级缓存——数据先从主内存到L2缓存，再从L2缓存到L1缓存，最后才到达CPU。本书不会深入探讨细节规则。（反正不同CPU的规则都有差异。）笔者只想指出，主内存比L2缓存慢，L2缓存比L1缓存慢。因此，L2缓存命中失败通常比L1缓存命中失败的成本高。

有一种特别差的缓存命中失败称为**load-hit-store**，此问题在PowerPC架构上（如Xbox 360和PS3）极为普遍。其出现过程是，CPU往某内存地址写入数据，随即又读取该地址，而此时要等待L1缓存写回数据至主内存，造成CPU的流水线停顿。详情可参阅网页[9]。

5.2.3.2 指令缓存和数据缓存

在为游戏引擎或任何性能关键系统编写高性能代码时，必须意识到数据和代码都会置于缓存内。指令缓存（instruction cache，I-cache）会预载即将执行的机器码，而数据缓存

[8]译注：Intel在1995年推出的Pentium Pro已开始有芯片上的L2缓存。2010年的i7芯片上含有高达12MB的L3缓存。

[9]http://assemblyrequired.crashworks.org/2008/07/08/load-hit-stores-and-the-__restrict-keyword

（data cache，D-cache）则用来加速自主内存读 / 写数据。大多数处理器会在物理上独立分开这两种缓存。因此，程序变慢，有可能因为指令缓存命中失败，或是数据缓存命中失败。

5.2.3.3 避免缓存命中失败

避免数据缓存命中失败的最佳办法就是，把数据编排进连续的内存块中，尺寸越**小**越好，并且要**顺序**访问这些数据。这样便可以把数据缓存命中失败的次数减至最少。当数据是连续的（即不会经常在内存中“跳来跳去”），那么单次命中失败便会把尽可能最多的相关数据载入单个缓存线。若数据量少，更有可能塞进单个缓存线（或最少数量的缓存线）。并且，当顺序存取数据时（即不会在连续的内存块**中**“跳来跳去”），便能造成最少次缓存命中失败，因为CPU不需要把相同区域的内存重载入缓存线。

要避免指令缓存命中失败，其基本原理和数据缓存的情况一样。然而，两者的实践方法不一样。由于编译器和链接器决定了代码的内存布局，读者可能会觉得自己对指令缓存命中失败几乎无法控制。然而，多数C/C++链接器都有一些简单规则，知悉并运用它们就能控制代码的内存布局。

- 单个函数的机器码几乎总是置于连续的内存。绝大多数情况下，链接器不会把一个函数切开，并在中间放置另一个函数。（内联函数除外，这点之后再解释。）
- 编译器和链接器按函数在翻译单元源代码（.cpp文件）中的出现次序排列内存布局。
- 因此，位于一个翻译单元内的函数总是置于连续内存中。即链接器永不会把已编译的翻译单元切开，中间加插其他翻译单元的代码。[10]

因此，按照数据缓存避免命中失败的原理，我们可以使用以下的经验法则。

- 高效能代码的体积**越小越好**，体积以机器码指令数目为单位。（编译器和链接器会负责把函数置于**连续**内存。）
- 在性能关键的代码段落中，**避免调用函数**。
- 若要调用某函数，就把该函数置于**最接近**调用函数的地方，最好是紧接调用函数的前后，而**不要**把该函数置于另一翻译单元（因为这样会完全无法控制两个函数的距离）。
- 审慎地使用内联函数。内联小型函数能增进效能。然而过多的内联会增大代码体积，使性能关键代码再不能完全装进缓存。假设有一个处理大量数据的紧凑循环，若循环内的代码不能完全装进缓存，每个循环迭代便会产生至少两次指令缓存命中失败。遇到这种情况，最好重新思考算法及其代码实现，看看能否减少关键循环中的代码量。

[10]译注：在Visual C++编译器中可使用函数级链接（function-level linking） /Gy选项，那么编译的输出单位为函数，链接时各个函数并不一定以翻译单元内的次序进行布局。事实上，还可以在链接时使用/ORDER选项自定义函数的布局次序。

5.3 容器

游戏程序员使用各式各样的集合型数据结构，也称为**容器**（container）或**集合**（collection）。各种容器的任务都一样——安置及管理0至多个数据元素。然而，细节上各种容器的运作方式有很大差异，每种容器也各有优缺点。常见的容器数据类型包括但肯定不限于以下所列。

- **数组**（array）：有序、连续储存数据的元素集合，使用索引存取元素。每个数组的长度通常是在编译期静态定义的。数组可以是多维的。C/C++原生支持数组（如`int a[5]`）。
- **动态数组**（dynamic array）：可在运行期动态改变长度的数组（如`std::vector`）。
- **链表**（linked list）：有序集合，但其数据在内存中是以非连续方式储存的（如`std::list`）。
- **堆栈**（stack）：在新增和移除数据时，采用后进先出（last-in-first-out，LIFO）的模式，也即压入（push）和弹出（pop）操作（如`std::stack`）。
- **队列**（queue）：在新增和移除数据时，采用先进先出（first-in-first-out，FIFO）的模式（如`std::queue`）。
- **双端队列**（double-ended queue，deque）：可以在两端高效地插入及移除数据（如`std::deque`）。
- **优先队列**（priority queue）：加入元素后，可用事先定义了的优先值计算方式，高效地弹出队列中优先值最高的元素。优先队列通常使用二叉堆（见下文）来实现（如`std::priority_queue`）。
- **树**（tree）：以层阶结构组织元素。每个元素（节点）有0个或1个父节点，以及0个至多个子节点。树是DAG（见下文）的特例。
- **二叉查找树**（binary search tree，BST）：二叉查找树中的每个节点最多含两个子节点。由于节点按预先定义的方式排列，任何时候都可以按该排列方式遍历整棵树。二叉查找树有多种类型，包括红黑树（red-black tree）、伸展树（splay tree）、AVL树（AVL tree）。[11]
- **二叉堆**（binary heap）：采用完全（或接近完全）二叉树的数据结构，通常使用（静态或动态）数组储存。根节点必然是堆中最大（或最小）的元素。二叉堆一般用来实现优先队列。[12]

---

[11]译注：如STL的`std::set`、`std::multiset`、`std::map`、`std::multimap`一般会用红黑树实现，能按照元素定义的顺序遍历。

[12]译注：原文描述二叉堆如同二叉查找树都是有序的，并不正确，译者自行改写本段描述。

- **字典**（dictionary）：由键值对（key-value pair）组成的表。通过键可以高效地查找到对应的值。字典又称为**映射**（map）或**散列表**（hash table），但其实从技术上来说，散列表只是字典的其中一个实现方式（如`std::map`、`std::hash_map`[13]）。
- **集合**（set）：保证容器内没有重复元素。集合好像字典，但只有键没有值。
- **图**（graph）：节点的集合，节点之间可任意以单向或双向路径连接。
- **有向非循环图**（directed acyclic graph，DAG）：图的特例，节点间以单向连接，并且无**循环**（即每条非空的路径里不能有相同的节点）。

5.3.1 容器操作

游戏引擎使用容器，必然也会利用多种常见算法。一些操作例子如下。

- **插入**（insert）：在容器中新增元素。新元素可置于表容器的开端、末端或其他位置。也有可能，容器本身根本无次序可言。
- **移除**（remove）：从容器中移除元素，当中可能需要查找操作（见下文）。然而，若有迭代器指向要移除的元素，使用该迭代器移除元素可能比较高效。
- **顺序访问 / 迭代**（sequential access/iteration）：按某“自然”次序访问容器内每个元素。
- **随机访问**（random access）：以任意次序访问容器的元素。
- **查找**（find）：从容器中寻找合乎条件的元素。有各式各样的查找操作，例如，逆向查找、查找多个元素等。此外，每种数据结构及每种情况，可能需要不同的算法（可参考维基百科[14]）。
- **排序**（sort）：把容器的元素以某方式排序。排序有很多种算法，例如冒泡排序（bubble sort）、选择排序（selection sort）、快速排序（quicksort）等（详情可参考维基百科[15]）。

5.3.2 迭代器

迭代器是一种细小的类，它“知道”如何高效地访问某类容器中的元素。迭代器像是数组索引或指针——每次它都会指向容器中某个元素，可以移至下一个元素，并能用某方式

[13]译注：C++ ISO标准并不含`std::hash_map`。在TR1和C++11的散列表模板是`std::unordered_map`，而使用散列表的集容器为`std::unordered_set`和`std::unordered_multiset`。

[14]`http://en.wikipedia.org/wiki/Search_algorithm`

[15]`http://en.wikipedia.org/wiki/Sorting_algorithm`

表示是否已访问容器中所有元素。例如，以下首段代码使用指针迭代访问C风格的数组，而第2段代码则用迭代器访问STL链表，两段代码的语法几乎完全相同。

```
void processArray(int container[], int numElements)
{
    int* pBegin = &container[0];
    int* pEnd = &container[numElements];
    for (int* p = pBegin; p != pEnd; ++p)
    {
        int element = *p;
        // 处理元素
    }
}
```

```
void processList(std::list<int>& container)
{
    std::list<int>::iterator pBegin = container.begin();
    std::list<int>::iterator pEnd = container.end();
    std::list<int>::iterator p;

    for (p = pBegin; p != pEnd; ++p)
    {
        int element = *p;
        // 处理元素
    }
}
```

相比直接访问容器的元素，采用迭代器的好处包括：

- 直接访问会破坏容器类的封装。而迭代器通常是容器类的**友元**（friend），因此它可高效迭代访问容器，同时不向外面暴露容器类的实现细节。（事实上，多数优良的容器类都会隐藏其内部细节，**不用**迭代器便不容许迭代访问内容。）
- 迭代器简化了迭代过程。大部分迭代器的行为和数组索引或指针相似，因此，无论构成容器的数据结构有多复杂，用户也可以编写一个简单的循环，每次把迭代器递增，并检查终止条件便可。例如，某迭代器可能使用中序（in-order）深度优先遍历（depth-first traversal），但使用起来和数组迭代一样简单。

5.3.2.1 前置递增与后置递增

读者可有留意，以上代码例子中采用了C++的**前置递增**（preincrement）运算符++p，而非**后置递增**（postincrement）运算符p++。此微小差异有时候对优化很重要。前置递增运

算符对运算子递增后，再传回其值；后置递增运算符则传回之前未递增的值。所以前置递增只需简单地把指针或迭代器就地递增，再传回它的参考。后置递增必须先备份旧值，把指针或迭代器递增，并传回之前的备份。对指针或整数索引而言，除了在紧凑的循环中，一般不会造成大问题[16]。然而，对于迭代器来说，后置递增可能导致效能损失，因为运算子备份及返回旧值时，或须进行复杂的迭代器对象建构及复制。（关于此问题的更详细讨论，可参阅[31]。）因此，最好习惯**任何时刻**都使用前置递增，除非真的需要后置递增的语意。

5.3.3 算法复杂度

为某应用场合选择容器时，要考虑容器的效能和内存特性。我们可以为每种容器的常见操作——如插入、移除、查找、排序，计算其理论效能。

我们通常把某操作的时间T，以容器内元素数目n的函数表示：

$$T = f(n)$$

我们通常不会想找出精确的函数f，而只对f的综合**数量级**（order）感兴趣。例如，若实际的函数是以下之一：

$$T = 5n^2 + 17,$$
$$T = 102n^2 + 50n + 12,$$
$$T = \frac{1}{2}n^2$$

无论任何情况下，我们都把表达式简化至其最重要项，上述3个例子都是n^2。为了表示函数的数量级，而非精确的方程，我们会采用大O记法（big-O notation），写成：

$$T = O(n^2)$$

算法的数量级通常可从其伪代码（pseudo code）中得知。若算法的运行时间和容器中的元素数目无关，我们称该算法为$O(1)$（即算法能在**常数时间**完成）。若算法会**循环**访问容器中的元素，则每个元素访问一次，例如，对无序表进行线性搜寻（linear search），那么我们称该算法为$O(n)$。（注意就算循环可能提早结束，仍然使用这个数量级[17]。）若算法有两层的嵌套循环（nested loop），每层循环可能会访问每个元素一次，那么我们称该算法

[16]译注：实际上，若编译后的语意没有分别（如上页代码例子中对指针递增），两者产生的机器码是一样的。

[17]译注：在算法分析中，最基本的方法是描述算法执行最坏情况时所需的时间数量级。例如，线性搜寻的最坏情况是，最后一个元素才符合搜寻条件，因此称它为$O(n)$。

为$O(n^2)$。若算法使用分治法（divide-and-conquer），例如**二分搜寻**（binary search）（当中每步能消去余下元素的一半），那么我们会预料该算法实际上最多访问$\log_2 n$个元素，因此称该算法为$O(\log n)$。若算法执行一个子算法n次，而该子算法本身是$O(\log n)$的，那么整个算法就是$O(n \log n)$了。

要选择合适的容器类，我们应观察预料中最常用的操作，选择对那些操作有最理想效能特性的容器。最常预见的数量级，由最快到最慢列表如下：$O(1)$、$O(\log n)$、$O(n)$、$O(n \log n)$、$O(n^2)$、$O(n^k)$，对于$k > 2$。

我们也要一并考虑容器的内存布局和内存使用特性。例如，数组（如`int a[5]`或`std::vector`）把元素**连续地**置于内存，而且除了储存那些元素本身，并**无额外内存开销**。（注意**动态**数组需要很小的固定额外开销。）而另一方面，链表（如`std::list`）会把元素包裹进“链节点”数据结构，此结构含指向下一元素的指针，也有可能有指向上一元素的指针，那么，每元素总共有**8字节**的额外开销。而且，链表内的元素无须置于连续的内存，实际上一般来说都不会连续。相比分散的内存块，连续的内存块通常更缓存友好。因此，对于高速算法，以缓存效能来说，数组通常比链表优胜（除非链表的节点置于细小、连续的内存，虽然这种情况很少有，但也不是闻所未闻）。然而，若插入及移除元素的速度最为重要，那么这些情况仍应使用链表。

5.3.4 建立自定义的容器类

许多游戏引擎都会提供常见容器数据结构的自定义实现。此惯例在游戏机引擎及移动电话/PDA上的游戏中尤其普遍。要自行建立容器类的各种原因如下。

- **完全掌控**：程序员能控制数据结构的内存需求、算法、何时 / 如何分配内存等。
- **优化的机会**：某些游戏机可能有某些硬件功能，可借这些功能优化数据结构和算法，或基于引擎中某个应用去做出微调。
- **可定制性**（customizability）：在第三者库如STL不常见的功能，可自行提供。（例如，搜寻n个最有关的元素，而非单个最有关元素。）
- **消除外部依赖**：你可能不会接触到第三方库的开发团队，若那些库出现问题，则可能无法立即自行调试和修正，而要等待该库的下一个发行版本。（可能等到游戏发行，该库还没有新版本！）

本书不能讲解所有数据结构，但我们会看看游戏程序员应付容器的一些常见方法。

5.3.4.1 建还是不要建

我们不会详细讨论实现所有这些数据类型的细节，相关的专著和线上资源多得泛滥。然而，我们会关注在哪里能取得所需的类型和算法实现。游戏引擎设计师有以下几种选择。

1. 自行建立所需的数据结构。
2. 依赖第三方的实现。常见的选择包括：
   (a) C++标准模板库（STL）；
   (b) STL的变种（variant），如STLport[18]；
   (c) 强大健壮的Boost库[19]。

STL和Boost都很有吸引力，因为它们提供丰富且强大的容器类，涵括大部分能想象得到的数据结构类型。除此以外，这两个软件包也提供强大、基于模板的**泛型算法**（generic algorithm）套件。当中有许多常见的算法实现，例如，在容器中寻找某元素，而这些算法都能应用至几乎任何数据对象的类型。然而，这类第三方库可能并不适合某些游戏引擎。而且，就算我们决定使用第三方库，也要从Boost、各种STL实现及其他第三方库中挑选。因此，我们先花点时间研究每种方案的优缺点。

STL

标准模板库（standard template library，STL）的优势包括：

- STL提供了丰富的功能。
- 在许多不同平台上也有尚算健壮的实现。
- 几乎所有C++编译器都带有STL。

然而，STL也有许多缺点，包括：

- 陡峭的学习曲线。虽然文档质量不错，但大部分平台的STL头文件都晦涩难懂。
- 相比为某问题而打造的数据结构，STL通常会较慢。
- 相比自行设计的数据结构，STL几乎总会占用更多内存。
- STL会进行许多动态内存分配。对于高性能、内存受限的游戏机游戏来说，控制STL的内存食欲是富有挑战性的工作。
- STL的实现和行为在各编译器上有微小差异，增加了多平台引擎上应用STL的难度。

[18]译注：实际上STLport合乎ISO C++标准库的实现，并加上一些扩展，因此译者认为这不算是STL变种。真正的STL变种，例如有EASTL。

[19]http://www.boost.org

只要程序员意识到STL的陷阱，并且审慎地使用，STL在游戏引擎编程中可占有一席之地。STL比较适合PC上运行的游戏引擎，因为现代PC的高级虚拟内存系统使内存分配变得高效，而且通常也能忽略物理内存不足的可能性。但另一方面，STL一般不适合游戏主机，因为游戏主机内存受限、缺乏高级CPU和虚拟内存。同时，使用STL的代码可能较难移植至其他平台。以下是笔者的经验法则。

- 首要的是，使用某STL类前，要认识其效能和内存特性。
- 若认为代码中的重量级STL类会造成瓶颈，尝试避免使用它们。
- 占小量内存的情况下才使用STL。例如，在游戏对象内加一个`std::list`是可以的，但在三维网格中的每个顶点加一个`std::list`，则应该不是个好主意。把三维网格的每个顶点加进一个`std::list`也并非好事，因为`std::list`类为每个元素动态分配细小的节点对象，会形成很多细小的内存碎片。
- 若引擎需要支持多平台，则笔者极力推荐使用**STLport**[20]。STLport是为了兼容多个编译器和目标平台而特别设计的，而且比原来的STL实现更高效、功能更丰富。

在《荣誉勋章：血战太平洋(Medal of Honor: Pacific Assault)》的PC版引擎里，大量使用了STL。虽然STL曾对此游戏的帧率有影响，但开发团队能解决这些由STL产生的效能问题（主要是通过小心地限制及控制STL的使用）。本书经常作为例子的，流行的面向对象渲染库OGRE，也都大量使用了STL。这只是笔者对STL的看法，读者的看法可能会不一样。笔者认为，在游戏引擎中使用STL是可行的，但必须用得审慎。

Boost

Boost是由几位C++标准委员会库工作小组成员发起的项目，但现时已成为有大量全球贡献者的开源项目。Boost的目标是制作一些库，能扩展STL并与STL联合工作，供商业或非商业使用。许多Boost库已纳入C++标准委员会的库技术报告（Library Technical Report，TR1），这是跃升为未来C++标准的一步。以下是Boost功能的简要。

- Boost提供许多有用但STL没有的功能。
- 某些情况下，Boost提供了替代方案，能解决一些STL设计上或实现上的问题。
- Boost能有效地处理一些非常复杂的问题，例如智能指针。（记住智能指针是复杂的东西，并且可能会严重影响性能。通常句柄是较好的选择，详见14.5节。）
- 大部分Boost库的文档都写得很好。这些文档不单解释每个库做什么和如何使用，很多时候还会深入探讨开发该库的设计决定、约束及需求。因此，阅读Boost文档也是学习

[20] `http://www.stlport.org`

软件设计原则的好方法。

若读者已使用STL，那么Boost可作为STL的扩展及/或部分STL功能的替代品。然而，必须注意以下列举的告诫。

- 大部分Boost核心类都是模板，因此，使用多数Boost功能时只需要包含一些头文件。然而，有些Boost库会生成颇大的.lib文件，可能不适合非常小型的游戏项目。
- 虽然全球规模的Boost社区是极好的支援网络，但Boost库并不提供任何保证。若读者碰到bug，你的团队有最终责任去避开问题或修正bug。
- 不保证支持向后兼容。
- Boost库是按Boost软件许可证发布的。若在引擎中使用，请小心阅读许可证内容[21]。

Loki

C++编程中有一门比较深奥的分支，称为**模板元编程**（template metaprogramming，TMP）。TMP的核心概念是利用编译器做一些通常在运行期才会做的工作，它运用C++模板功能诱使编译器做一些原本并非为此而设的事情。这促使TMP成为出奇强大又有用的工具。

至今，最知名且可能是最强大的C++ TMP库是Loki。Loki由Andrei Alexandrescu设计及实现（可参考其个人网站[22]），而Loki可从SourceForge获取[23]。

Loki极其强大，其迷人的代码也是值得学习的。然而，在实际应用时，Loki有两大缺点：（a）它的代码可能望而生畏，难以使用及全面理解；（b）有些元件依赖某些编译器的“副作用”行为，须细心调整才能应用在新的编译器上。因此，使用Loki较为棘手，而且相比其他“较不极端的”库来说，其移植能力较弱。Loki不适合胆小者。话虽如此，就算不使用Loki本身，一些Loki概念，例如**基于原则的设计**（policy-based design），也可应用到任何C++项目。笔者极力推荐所有软件工程师阅读Andrei的开创性著作《C++设计新思维（Modern C++ Design）》[2]，Loki库延生于此书。

5.3.4.2 动态数组和大块分配

在游戏编程中，经常大量使用C风格的固定大小数组，因为这种数组无须内存分配，又因连续而对缓存友好。数组的常用操作，例如添加数据和查找，也是非常高效的。

[21] http://www.boost.org/more/license_info.html

[22] http://www.erdani.org

[23] http://loki-lib.sourceforge.net

当数组的大小不能在编译时决定时，程序员会倾向转用**链表**或**动态数组**。若我们想维持固定大小数组的效能和特性，则通常会选用动态数组作为数据结构。

实现动态数组的最简单方法就是，在开始时分配n个元素的缓冲区；当缓冲区已含有n个元素，再加入新元素时，就把缓冲区**扩大**。这带来固定大小数组的优良效能特性，但又去除了元素上限。扩大缓冲区的实现方法为，分配一个更大的新缓冲区，再把原来的数据复制过去，最后释放原来的缓冲区。增加的大小按规则而定，可以每次增加n个元素，也可以每次把原来元素数量加倍。笔者遇见过的大多数实现都只会扩大而不会缩小（一个值得关注的例外是，当把数组清空为0个元素，有些实现可能会释放缓冲区，有些不会）。因此，数组的大小成为一种“高水位线”。STL的`std::vector`类就是如此运作的。[24]

若能为数据设立一个高水位线，最好当然是在引擎开始时就分配该大小的缓冲区。扩大动态数组时由于要分配内存及复制数据，其代价可能非常高。它对效能的影响力视缓冲区大小而定。扩大缓冲区时，释放旧缓冲区也会导致内存碎片。因此，和其他需要分配内存的数据结构一样，使用动态数组时也要十分审慎。动态数组在开发期可能是最优选择，因为当时还未能确定缓冲区所需的大小。当能确定适当的内存预算时，总可以把动态数组改为固定大小数组[25]。

5.3.4.3 链表

在选择数据结构时，若主要考虑因素并非内存的连续性，而是希望能在任何位置高效插入及移除元素，那么**链表**是常见之选。实现链表的难度不高，但不小心的话仍然会出错。要实现健壮的链表，本节会介绍一些提示和窍门。

链表基础

链表是非常简单的数据结构。链表中每个元素都有指针指向下一个节点；在双向链表（doubly-linked list）中，每个元素还有指针指向上一个节点。这两种指针称为**链接**（link）。为了跟踪整个链表，还需要另一对称为**头**（head）和**尾**（tail）的指针，分别指向首节点和末节点。

在双向链表中插入新节点时，须把前一节点的“后节点指针”，以及后一节点的“前节点指针”，都改为指向新节点，并且也要相应更改新节点中的两个指针。共有4种情况要处理。

---

[24]译注：在STL中，一个`std::vector<T> v`变量可以使用以下的常用手法将之缩小：`std::vector<T>(v).swap(v)`。

[25]译注：建议读者真的要用固定大小数组替代动态数组时，自行建立一个兼容`std::vector`接口的模板。

- 在空链表中加入节点；
- 在首节点前加入节点；
- 在末节点后加入节点；
- 在链表内部插入节点。

图5.8说明了这4种情况。

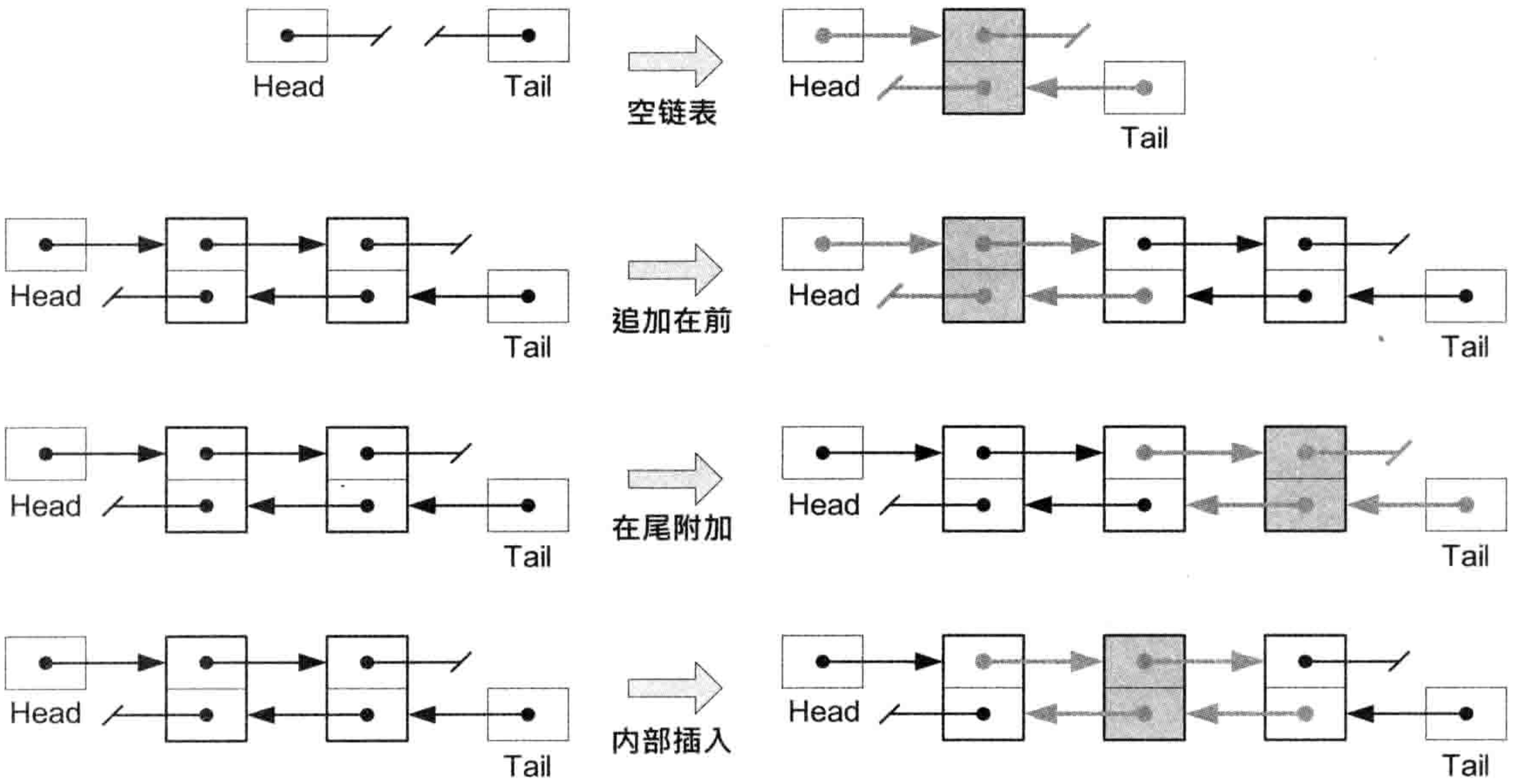


图 5.8: 把元素加入链表时必须处理4种情况：空链表、追加在前、在尾附加及内部插入。

移除节点也涉及类似的操作，须改动要移除节点及其前后节点中的指针。移除节点也有4种情况：移除首节点、末节点、内部节点，以及移除链表中只有一个节点的情况（即清空链表）。

节点数据结构

实现链表的代码并不特别难写，只是容易出错。因此，编写一个能管理任何元素类型的通用的链表，一般来说是个好主意。要这么做，首要是把元素的数据结构和储存链接（即“后节点指针”和“前节点指针”）的数据结构分开。链表节点的数据结构一般是简单的`struct`或`class`，命名为`Link`、`Node`、`LinkNode`之类，并会以元素类型作为模板参数。一般样子是这样的：

```
template< typename ELEMENT >
struct Link
{
    Link<ELEMENT>*  m_pPrev;
    Link<ELEMENT>*  m_pNext;
    ELEMENT*        m_pElem;
};
```

外露式表

外露式表（extrusive list）是一种链表，其节点数据结构完全和元素的数据结构分离。每个节点含指针指向元素，如上述的例子。当要在链表内加入元素时，便要为该元素分配一个节点，并适当地设置元素指针、前节点指针和后节点指针。移除元素时，就能释放其节点。

外露式设计的优点是，一个元素能同时置于多个链表，只需为每个链表分配独立的节点，指向该共享元素。而其缺点是，必须动态分配节点。许多时候，会使用池分配器（见5.2.1.2节）分配节点，因为每个节点是同等大小的（在32位机器上是12字节）。由于池分配器有高效及避免内存碎片的特性，在此应用中是极佳选择。[26]

侵入式表

侵入式表（intrusive list）是另一种链表，其节点的数据结构被嵌进目标元素本身。此方式的最大好处是无须再动态分配节点，每次分配元素时已“免费”获得节点。例如，可以把元素类编成这样：

```
class SomeElement
{
    Link<SomeElement>  m_link;

    // 其他成员
}
```

也可从Link类派生元素类。这样使用继承，和把一个Link对象作为类的第1个元素，几乎是等同的。但使用继承有额外好处，就是可以把节点指针（Link<SomeElement>*）向下转型至指向元素本身的指针（SomeElement*）。这意味着我们能消去节点中指向元素的指针。以下是C++中的一个可行设计。

[26]译注：STL的所有容器都是外露式的。

```
template< typename ELEMENT >
struct Link
{
    Link<ELEMENT>*  m_pPrev;
    Link<ELEMENT>*  m_pNext;
    // 由于继承的关系，无须 ELEMENT* 指针
}

class SomeElement : public Link<SomeElement>
{
    // 其他成员
};
```

侵入式表的最大缺陷在于，每个元素不能同时置于多个链表中（因为每个元素只有一个节点数据）。若要把元素同时加进N个链表，可在元素中加入N个节点成员（但此情况下就不能使用继承方式了）。然而，N的值必须事前固定，所以侵入式表并不及外露式表有弹性。

选外露式表还是侵入式表，要看实际应用以及操作上的限制。若不惜一切代价都要避免动态内存分配，那么侵入式表大概是最佳的。若能负担得起池分配的开销，则外露式表可能更适合。有时候，二者中只有唯一的可行方案。例如，我们希望在链表中储存一些实例，而这些实例的类是来自第三方库的，若不能或不想修改该库的源码，外露式表便会成为唯一选择。

头尾指针：循环链表

完整的链表实现还须提供头尾指针。最简单的做法是把这两个指针包装成为一个独立的数据结构，例如称为LinkedList，如下：

```
template< typename ELEMENT >
class LinkedList
{
    Link<ELEMENT>*  m_pTail;
    Link<ELEMENT>*  m_pHead;

    // 操作链接的成员函数
};
```

读者可能会发现，LinkedList和Link的分别并不大，两者都各含一对指向Link的指针。我们会发现，使用Link类管理头尾指针（如以下代码），有些显著的好处。

```
template< typename ELEMENT >
class LinkedList
{
    Link<ELEMENT>    m_root;     // 包含头和尾

    // 操作链接的成员函数
};
```

嵌入的m_root成员是一个Link，如同链表中的其他Link（除了m_root.m_pElement一直会是NULL）。如图5.9所示，这样会形成一个循环，故称为**循环链表**（circular linked list）。换句话说，链表中“真正”最后节点的m_pNext指针和“真正”首个节点的m_pPrev指针，都会指向m_root。

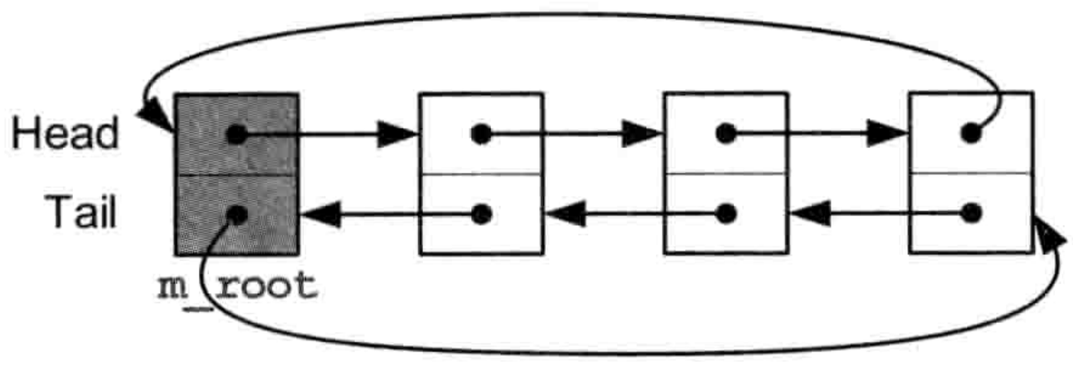


图 5.9: 把头尾指针储存于节点中就能实现循环链表。这样可以简化实现，并有其他好处。

相比先前的使用两个独立头尾指针的设计，此设计更优，因为它能简化插入及移除元素的逻辑。要明白其中的奥妙，先看在独立头尾指针设计中移除元素的代码：

```
void LinkedList::remove(Link<ELEMENT>& link)
{
    if (link.m_pNext)
        link.m_pNext->m_pPrev = link.m_pPrev;
    else
        m_pTail = link.m_pPrev; // 正在移除链表中的末元素

    if (link.m_pPrev)
        link.m_pPrev->m_pNext = link.m_pNext;
    else
        m_pHead = link.m_pNext; // 正在移除链表中的首元素

    link.m_pPrev = link.m_pNext = NULL;
}
```

若使用m_root的设计，则代码会变得稍微简单一些：

```
void LinkedList::remove(Link<ELEMENT>& link)
{
    // link 必然为链表中的成员
    ASSERT(link.m_pNext != NULL);
    ASSERT(link.m_pPrev != NULL);

    link.m_pNext->m_pPrev = link.m_pPrev;
    link.m_pPrev->m_pNext = link.m_pNext;

    // 这么做以表示link已不属任何链表了
    link.m_pPrev = link.m_pNext = NULL;
}
```

以上代码的粗体部分更揭示了循环链表的另一优点：节点的m_pPrev和m_pNext**不会为空指针**，除非该节点不属于任何链表（即该节点并未使用）。这能简单检测节点是否属于一个链表。

再来对比独立头尾指针的设计，该设计的链表中，首节点的m_pPrev必然是空指针，末节点的m_pNext亦然。若链表中只有一个节点，其两个指针**都会**是空指针。那么，就不能单凭节点本身，得知它是否已隶属一个链表。

单向链表

单向链表（singly-linked list）中的节点只有后节点指针而没有前节点指针。（整个链表可能同时有头尾指针，或只有头指针。）此设计明显地能节约内存，但其代价在于插入或移除元素。由于没有m_pPrev指针，所以需要从头遍历才能找到前节点，才能适当地更新其m_pNext指针。因此，双向链表的移除操作是$O(1)$，而单向链表的则是$O(n)$。[27]

这固有的插入及移除代价通常是难以承受的，因此大多数链表都是双向的。然而，若读者肯定只会加入或移除链表的首元素（例如用来实现堆栈），或只会加入首元素并移除末元素（例如用来实现队列，并且链表同时含头尾指针），那么便可避开单链表的问题，并节省一些内存。

---

[27]译注：在单向链表中，虽然移除某一节点的复杂度是$O(n)$，但移除某节点的下一节点则是$O(1)$。因此，在设计单向链表类时，可加入removeAfter(Link<ELEMENT>& link)这类API，并在应用时尽量使用。单向链表的最大缺点在于不能高效地逆向遍历。

5.3.4.4 字典和散列表

字典是由键值对组成的表。在字典中，用键能快速查找出对应的值。键和值可以是任何数据类型[28]。此类数据结构通常是使用二叉查找树或散列表来实现的。

在二叉查找树的实现中，键值对储存在二叉树的节点里，而整棵树则是按键值排序节点。用键查找值时，需要$O(\log n)$的二分查找操作。

在散列表的实现中，所有值储存于固定大小的表里，表中的每个位置表示一个或多个键。要插入键值对时，首先要把键转换为整数形式（若键原本并非整数），此转换过程称为**散列**（hashing）。然后，把散列后的键**模除**（modulo）表的大小来求得表的索引。最后，把键值对储存在该索引的位置上。**模除**运算（C/C++中的%）可以用来计算整数键除以表的大小后得出的余数。所以，若散列表有5个位置，键是3的话就会储存至索引3的位置（`3 % 5 == 3`），键是6的话就会储存至索引1的位置（`6 % 5 == 1`）。若无碰撞发生，用键查找散列表的复杂度为$O(1)$。

碰撞：开放和闭合散列表

有时候，两个或以上的键最终会占用散列表的同一位置，此情况称为**碰撞**（collision）。有两种基本方法**解决**碰撞，这两种方法引伸出两种散列表。

- **开放式散列**（open hashing）[29]：在开放式散列表中（见图5.10），碰撞发生时，多个键值对会储存在同一位置上，这些键值对通常以链表形式储存。此方法容易实现，并且储存于表中的键值对数目并无上限。然而，每次对这种散列表加入新键值对时都要动态分配内存。
- **闭合式散列**（closed hashing）[30]：在闭合式散列表中（见图5.11），解决碰撞的方法是进行**探查**（probing）过程，直至找到空位。（“探查”是指使用明确定义的算法找出空位。）此方法比较难实现，并且必须要设定表的键值对数目上限（因为每个位置只能储存一个键值对）。但其主要优点为，所需内存是固定的，散列表建立之后不用再分配动态内存。因此，这种散列表通常是游戏机引擎的好选择。

---

[28]译注：其实也有些限制，例如二叉查找树的实现中，键类型必须是可比较的（comparable），并且键集为全序关系（total order）。而散列表中的键类型必须提供散列函数，把键转换为整数形式。

[29]译注：开放式又名分离链式（separate chaining）。

[30]译注：非常容易混淆的是，闭合式散列又名为开放定址（open addressing）。

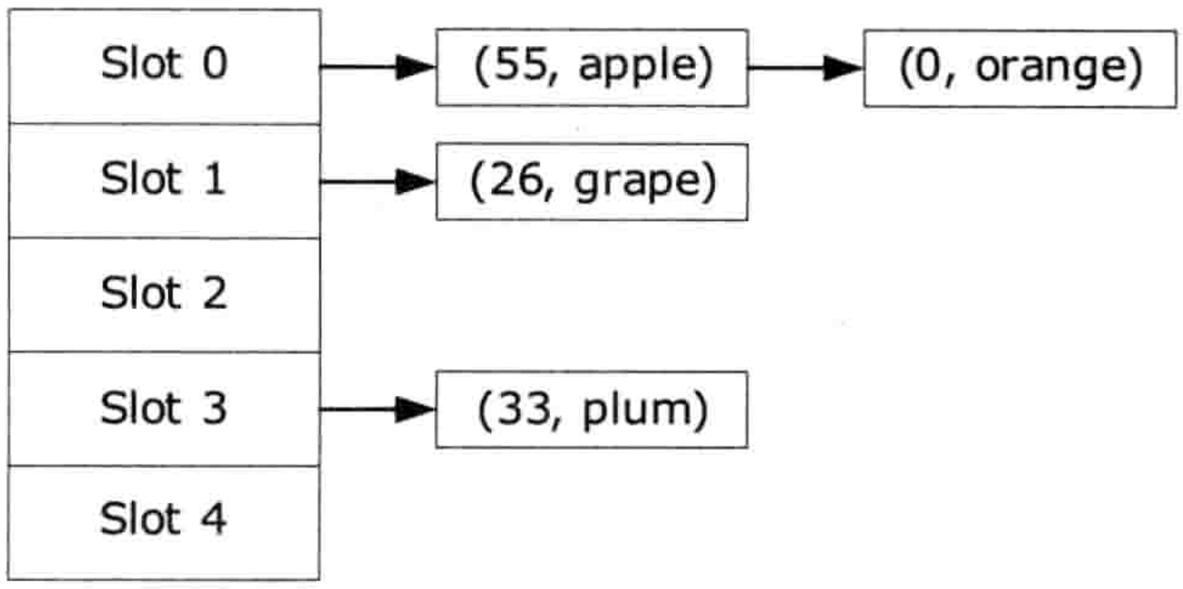


图 5.10: 开放式散列。

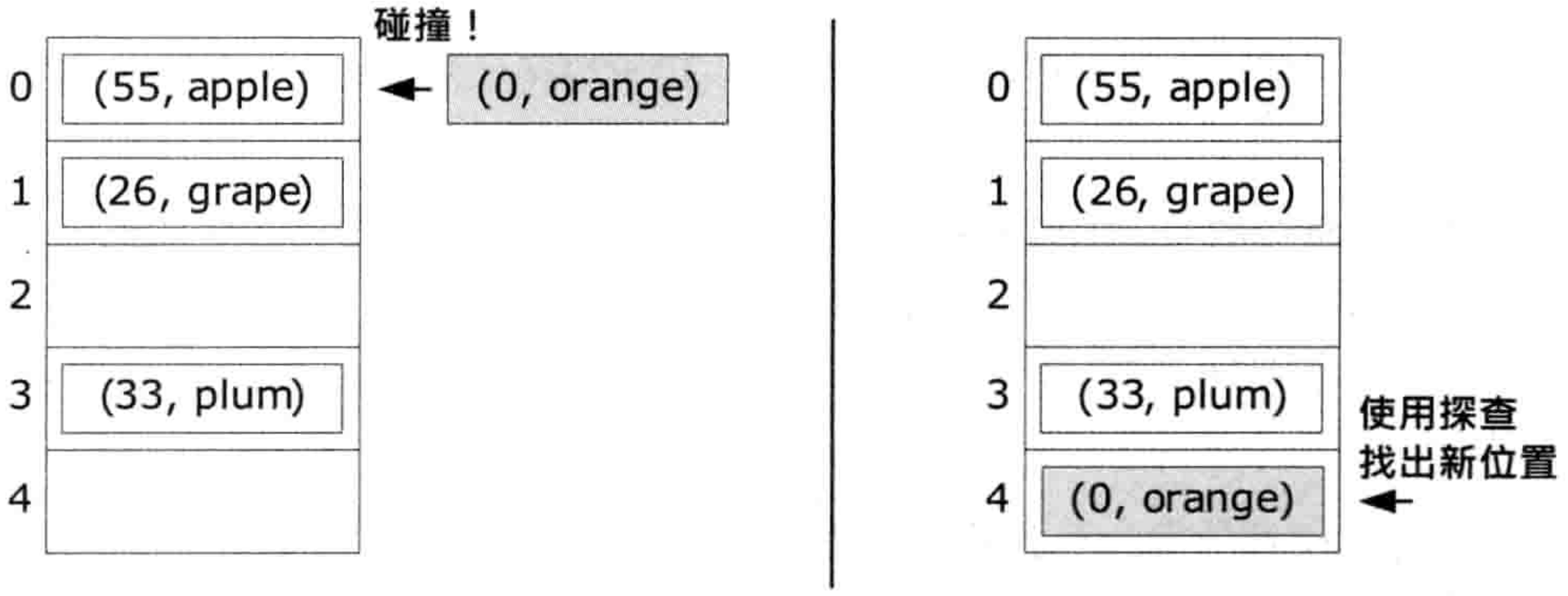


图 5.11: 闭合式散列。

散列法

散列法（hashing）是把任意数据类型的键转换为整数的过程，该整数模除表的大小就能求得表的索引。数学上可以表示为，给定键k，我们希望可以使用散列函数（hash function） H，产生整数散列值h，然后再求出表的索引i，如下：

$$h = H(k),$$
$$i = h \bmod N,$$

当中N是表的位置数目，而mod表示**模除**运算，即是求h整除N的**余数**。

若键本身为整数类型，则散列函数可以是恒等函数$H(k) = k$。若键为32位浮点数类型，则散列函数可以仅仅把其位模式（bit pattern）诠释为32位整数。

```
U32 hashFloat(float f)
{
    union
    {
        float   asFloat;
        U32     asU32;
    } u;

    u.asFloat = f;
    return u.asU32;
}
```

若键为字符串，就要使用**字符串散列函数**，把字符串中所有字符的ASCII或UTF码合并为单个32位整数。

散列函数的**质量**对散列表的效能极为重要。优良的散列函数，是指那些能把所有有效键平均分报至整个散列表的函数，从而能使碰撞机会减至最低。散列函数的运算时间也要快。另外，散列函数必须是**决定型的**（deterministic），换言之，每次相同的输入都会产生完全等同的输出。

字符串可能是读者最常会遇到的键类型，所以懂得一些优良的字符串散列函数特别有用。以下是几个优良的算法。

- LOOKUP3，由Bob Jenkins开发[31]。
- 循环冗余校验（cyclic redundancy check）函数，例如CRC32。[32]
- 信息摘要算法5（message-digest algorithm 5，MD5）是密码用的散列函数，能产生极好的结果，但其运算成本比较高[33]。
- Paul Hsieh的文章[34]列出一些其他的优良选择。

实现闭合散列表

在闭合散列表中，键值对直接储存于表里，而非储存于每个位置的链表。此方法使程序员预先定义散列表所用到的精确内存量[35]。要解决**碰撞**问题（即两个键映射到相同的位置），就要使用**探查法**。

[31] http://burtleburtle.net/bob/c/lookup3.c

[32] http://en.wikipedia.org/wiki/Cyclic_redundancy_check

[33] http://en.wikipedia.org/wiki/MD5

[34] http://www.azillionmonkeys.com/qed/hash.html

[35] 译注：如同动态数组，闭合散列表也可以实现动态扩大及缩小。只需要建立一个新大小的散列表，把原有散列表的键值对重新加到新散列表。由于键会重新进行散列，其分布会和旧的完全不同。

最简单的探查法是**线性探查**（linear probing）。假设散列函数产生了表索引i，但该位置已被占，那么线性探查法就是继续去找$(i+1)$、$(i+2)$等位置，直至找到空的位置（到了$i=N$时就把索引设到表的开端）。另一种线性探查的变体是交替向前和向后搜索，即$(i+1)$、$(i-1)$、$(i+2)$、$(i-2)$以此类推，记得要对产生的索引用模除法使其符合表的有限范围。

线性探查往往使键值对聚集成群。要避免产生这些集群，可使用名为**二次探查**（quadratic probing）的算法。从已占位置索引i开始，探查数列$i_j \pm j^2, j=1,2,\cdots$，换言之，即探查$(i+1^2)$、$(i-1^2)$、$(i+2^2)$、$(i-2^2)$，以此类推。记得同样要对产生的索引用模除法使其符合表的有限范围。

在使用闭合散列时，把散列表设为**质数**大小是个好主意。结合使用质数大小的表和二次探查，往往能得出表位置的最佳覆盖，并有最少的集群。详见此文[36]中讨论为何质数大小的表更可取。

5.4 字符串

字符串（string）几乎在所有软件项目中无处不在，游戏引擎也不例外。表面上，字符串看似只是个简单基本的数据类型，但当读者开始在项目中应用字符串，很快便会发现大量的设计问题和限制，当中每个设计决定都要细心考究。

5.4.1 字符串的使用问题

首先，最基本的问题是，如何在程序中储存和管理字符串。在C和C++中，字符串甚至不是一个原子数据类型，而是实现为**字符数组**。面对可变长度的字符串，若不是硬设置字符串的长度限制，就必须动态分配内存作为字符串缓冲区。C++程序员通常不直接处理字符数组，而较喜欢使用**字符串类**。那么，该用哪一个字符串类呢？STL提供了不错的字符串类，但若读者已决定弃用STL，便免不了要自己重新实现。

另一个字符串相关的问题是**本地化**（localization）——更改软件以发布其他语言的过程。这也称为**国际化**（internationalization），或简称I18N[37]。对每个向用户显示的字符串，都要事先翻译为需要支持的语言。（在程序内部使用、永不显示于用户的字符串，当然无须本地化。）除了通过使用合适的字体（font），为所有支持语言准备字符字形（character

[36] http://www.cs.utk.edu/~eijkhout/594-LaTeX/handouts/hashing-slides.pdf

[37] 译注：因I和N之间有18个字母。同样地，本地化也简称为L10N。

glyph），游戏还需要处理不同的文本方向（text orientation）。例如，希伯来文是由右至左阅读的[38]。游戏也需要优雅地处理译文比原文长很多或短很多的情况。

最后，需要知道，游戏引擎内部还会使用一些字符串，用作资源文件名、对象标识符等用途。例如，当游戏设计师设计一个关卡时，容许他为关卡中的对象命名，是非常方便的，例如，把一些对象命名为“玩家摄像机”、“敌方-坦克-01”、“爆炸触发器”等。

如何处理这些内部字符串，对游戏的性能举足轻重。因为在运行期操作字符串本身的开销花费不菲。比较或复制`int`或`float`数组，可使用简单的机器语言指令完成。然而，比较字符串需要$O(n)$的字符数组遍历（n为字符串的长度），例如使用`strcmp()`。复制字符串也需要$O(n)$的内存遍历，这还未考虑到需要为复制分配内存[39]。笔者曾参与一个项目，从剖析游戏的性能中发现，当中`strlen()`和`strcpy()`是两个开销最高的函数！之后我们改进程序，避免了不必要的字符串操作，并使用本节介绍的一些技巧，最终能从性能剖析中剔除这两个函数，并令游戏帧数显著提升。（笔者也从多个工作室的开发者那听闻过类似的故事。）

5.4.2 字符串类

字符串类大大方便了程序员使用字符串。然而，字符串类含有隐性成本，在性能分析之前难以预料。例如，用C风格字符数组形式，把字符串传递给函数，过程非常迅速，因为这通常只要把字符串首字符的地址存于寄存器再传递过去即可。然而，传递字符串对象时，若函数的声明或使用不恰当，可能会引起一个或多个拷贝构造函数的开销[40]。复制字符串时可能涉及动态内存分配，这会导致一个看似无伤大雅的函数调用，最终可能花费几千个机器周期。

因此，作者在游戏编程中一般会避免字符串类。然而，若读者强列希望使用字符串类，在选择或实现字符串类时，则务必查明其运行性能特性在可接受的范围，并让所有使用它的程序员知悉其开销。了解你的字符串类：它是否把所有字符缓冲区当作只读的？它是否使用了**写入时复制**（copy-on-write）优化？（参考维基百科[41]）一个经验法则是，经常以参考形式传递对象，不要以值来传递（因为后者通常会导致调用拷贝构造函数）。尽早剖析代码的性能，并确定字符串不是掉帧的主要原因。

---

[38]译注：原文还提及中文是竖排的，但是游戏软件中很少会使用传统的竖排中文。

[39]译注：其实`int`和`float`数组的比较或复制也是$O(n)$。字符串之所以相对较慢，在于其可变长度。C/C++中常用的字符串是空结尾字符串（**null-terminated string**），相关的操作如`strcmp`和`strcpy`都要检测扫描中的字符是否为空字符（`'\0'`），因此通常比`memcmp`和`memcpy`慢。

[40]译注：字符串对象以值传递（pass by value）时需要调用拷贝构造函数。若采用以参考传递（pass by reference）或以地址传递（pass by address）则不会有这个开销。

[41]`http://en.wikipedia.org/wiki/Copy-on-write`

笔者认为，有一种情况，有理由使用特化的字符串类，这便是储存和管理文件系统路径。假设有一个Path类，相比原始C风格字符数组，可以加入很多有意义的功能。例如，Path类能提供函数从路径中提取文件名、文件扩展名或目录。它也可以隐藏操作系统之间的差异，如自动转换Windows风格的反斜线至UNIX风格的正斜线，或其他操作系统的路径分隔符（path separator）。以跨平台方式撰写这种Path类在游戏引擎中是很有价值的。（关于这方面的细节详见6.1.1.4节）

5.4.3 唯一标识符

在任何虚拟游戏中，**游戏对象**都需要某种唯一标识方法。例如，吃豆人里的游戏对象可能被命名为“pac_man”、“binky”、“pinky”、“inky”、“clyde”等。使用唯一标识符（unique identifier），游戏设计师便能逐一记录组成游戏世界的无数个对象，而在运行时，游戏引擎也能借唯一标识符寻找和操控游戏对象。此外，组成游戏对象的**资产**（asset），如网格、材质、纹理、音效片段、动画等，也需要唯一标识符。

字符串似是这类标识符的天然选择。游戏资产通常储存为磁盘上的个别文件，因此它们通常可以用路径作为唯一标识符，而路径理所当然是字符串。游戏对象是由游戏设计师创建的，游戏设计师也顺理成章会为对象指派一个清晰明了的名字，而不希望记忆一些整数的对象索引，如64位或128位全局唯一标识符（globally unique identifier，GUID）。然而，**比较**标识符的速度在游戏中可能极有影响，`strcmp()`完全不能达到要求。我们想找到方法，“既要鱼，又要熊掌”[42]——既有字符串的表达能力和弹性，又要有整数操作的速度。

5.4.3.1 字符串散列标识符

把字符串**散列**（hash）是好方案。如之前提及，散列函数能把字符串映射至半唯一整数。字符串散列码能如整数般比较，因此其比较操作很迅速。若把实际的字符串存于散列表，那么就可以凭散列码取回原来的字符串。这在调试时非常有用，并且可以把字符串显示在屏幕上或写入日志文件中。游戏程序员常使用**字符串标识符**（string id）一词指这种散列字符串。虚幻引擎则称为**name**（由FName类实现[43]）。

如同许多散列系统，字符串散列也有散列**碰撞**的机会（即两个不同的字符串可能有相同的散列值）。然而，若有恰当的散列函数，则我们可以保证，游戏中可能用到的合理字符串

[42]译注：原文使用英文谚语“have a cake and eat it too”，直译是既要保留蛋糕，又要吃掉它。

[43]译注：虚幻的FName采用的方式和上述有些差别。虚幻把不同的FName对象储存在一个全局数组里，而每个FName对象储存了它置于数组中的索引。另设一个全局散列表映射字符串至数组里的FName对象。两个FName对象比较时，使用索引比较，而不是散列码，因此不会出现下文的散列碰撞问题。另外，FName对象在建构以后，可以直接取得原来的字符串，而不需要使用散列码取回原来的字符串。

输入不会做成碰撞。毕竟，32位散列码能表示超过40亿个值。因此，若散列函数能在此广大范围中平均分布字符串，则很少机会产生碰撞。在顽皮狗开发《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》的两年里，一次碰撞也未出现过。[44]

5.4.3.2 一些关于实现的主意

概念上，用散列函数使字符串产生字符串标识符十分容易。然而实际上，**何时**去计算散列是个问题。多数采用字符串标识符的游戏引擎会在运行时进行散列。在顽皮狗，我们容许在运行时进行散列，但也使用简单工具去预处理源代码，把每个SID（**任何字符串**）的宏直接翻译为相对的散列值。这么做，任何整数常数能出现的地方，都可以使用字符串标识符，包括switch语句中case标签的常数。（在运行时调用函数去产生的字符串标识符并非常数值，所以不能用于case标签。）

从字符串产生字符串标识符的过程，有时候称为**字符串扣留**（string interning），因为此过程除了会散列字符串，还会把它加进一个全局字符串表里。那么就能以散列值取回原来的字符串。另外，在工具中也可以加入把字符串散列为字符串标识符的能力，那么其产生的数据，在送交引擎前，当中的字符串便能被散列了。

字符串扣留的主要问题是其速度缓慢。首先要对字符串进行散列，这本身已是昂贵的操作，尤其是在扣留大量字符串的时候。此外，需要为字符串分配内存，并复制至查找表中。因此（若非在编译时产生字符串标识符），最好只对字符串扣留仅一次，并把结果储存备用。例如，以下首段代码比第2段代码好，因为第2段每次调用函数f()都会不必要地重新扣留字符串。

```
static StringId     sid_foo = internString("foo");
static StringId     sid_bar = internString("bar");

// ......

void f(StringId id)
{
    if (id == sid_foo)
    {
        // 处理 id == "foo" 的情况
    }
    else if (id == sid_bar)
```

[44]译注：根据著名的“生日问题（birthday problem）”，散列产生碰撞的概率是很高的。例如，用32位散列77,000次，含至少一次碰撞的概率已达50%。译者建议设计时应容许碰撞，并使用如FName的方式，使用唯一索引做比较。关于生日问题对于散列的影响可参考http://en.wikipedia.org/wiki/Birthday_attack。

```
    {
        // 处理 id == "bar" 的情况
    }
}
```

下面的方式则低效得多：

```
void f(StringId id)
{
    if (id == internString("foo"))
    {
        // 处理 id == "foo" 的情况
    }
    else if (id == internString("bar"))
    {
        // 处理 id == "bar" 的情况
    }
}
```

以下是internString()的实现方式之一：

```
// stringid.h
typedef U32 StringId;

extern StringId internString(const char* str);

// stringid.cpp
static HashTable<StringId, const char*> gStringIdTable;

StringId internString(const char* str)
{
    StringId sid = hashCrc32(str);

    HashTable<StringId, const char*>::iterator it =
        gStringIdTable.find(sid);

    if (it == gStringTable.end())
    {
        // 此字符串未加入表里，把它加入表
        // 记得要复制字符串，以防原来的字符串是动态分配的并将被释放
        gStringTable[sid] = strdup(str);
    }

    return sid;
}
```

虚幻引擎采用的另一个主意是，把字符串标识符和相应的C风格字符串包装进一个细小的类。在虚幻引擎中，此类名为FName。

利用调试内存储存字符串

当采用字符串标识符时，字符串本身只供开发人员使用。在发行游戏时，那些字符串几乎是不需要的——游戏本身应只使用字符串标识符。因此，可以把字符串表放在零售版游戏不会使用的内存空间里。例如，PS3开发机有256MB的零售版内存（retail memory），加上256MB额外的“调试版”内存（debug memory），后者不存在于零售版的游戏机。若把字符串置于调试内存，就不用担心其内存印迹影响最终版的游戏。（我们必须小心编写生产代码，永不依赖那些字符串！）

5.4.4 本地化

把游戏（或任何软件项目）本地化是件艰巨任务。这种任务最好在项目开始时就规划好，制定每个开发阶段的本地化工作。然而，有时候事与愿违。以下一些提示，有助于规划游戏引擎项目的本地化工作。关于软件本地化的深入讨论，可参考[29]。

5.4.4.1 Unicode

多数说英语的开发者的最大问题是，他们自小认定了字符串是8位ASCII字符码数组[45]（即基于ASCII[46]标准的字符）。ASCII字符串对于只有少量字母的语言（如英文）是没问题的。但对于含大量字符的语言，其字形（glyph）和英语的26个字母完全不同，ASCII字符串便无能为力了。针对ASCII标准所限，Unicode应运而生。[47]

现请读者暂放下本书，先阅读Joel Spolsky的文章《每位软件开发者都绝对必知的Unicode及字元集知识（没有借口！）/The Absolute Minimum Every Software Developer Absolutely，Positively Must Know About Unicode and Character Sets (No execuses!)》[48]。（读毕后可继续阅读本书！）

---

[45]译注：ASCII只有128个字符，应该只是7位。

[46]译注：从这里开始，原文采用ANSI一词，应该是指Windows上的Windows-1252编码，它含256个字符，所以其实不应混淆ANSI和ASCII，译者做出修正。

[47]译注：Unicode的中文名称有统一码、万国码、单一码、标准万国码。为避免混淆，此处采用原名。

[48]http://joelonsoftware.com/articles/Unicode.html。译注：中文译本是http://local.joelonsoftware.com/wiki/The_Joel_on_Software_Translation_Project:%E8%90%AC%E5%9C%8B%E7%A2%BC。

按Joel在文中提及的，Unicode并非单一标准，实际上是一篮子相关标准。读者需要为项目选择合适的特定标准。笔者在游戏引擎中最常采用的是UTF-8和UTF-16。

UTF-8

在UTF-8编码中，每个字符占1～3字节。因此UTF-8字符串所占的字节数量不一定等于其长度（字符个数）。此称为**多字节字符集**（multibyte character set，MBCS），因为每个字符占一至多个字符的储存空间[49]。

UTF-8的优点之一是向后兼容ASCII编码。可以向后兼容，是因为UTF-8的多字节字符，其首字节的最高有效位必然是1（即首字节于128至255之间）。由于标准ASCII字符码皆少于128，所以简单的旧ASCII字符串都是合法、无歧义的UTF-8字符串。

UTF-16

UTF-16标准采用更简单但较昂贵的方法进行编码。UTF-16中每个字符都确切地使用16位（无论该字符是否真的需要那么多位）[50]。因此，把UTF-16字符串所占的字节除以2，便可以得到字符个数。UTF-16是宽字符集（wide character set，WCS），因为每个字符是16位宽，有别于“通常”的ASCII `char`。[51]

Windows下的Unicode

在Windows下，`wchar_t`数据类型能用来表示单个“宽”UTF-16字符（WCS），而`char`则用作ANSI字符串及多字节UTF-16字符串（MBCS）。此外，Windows容许程序员编写**字符集无关**（character set independent）的代码。为此Windows提供名为`TCHAR`的数据类型。当用ANSI模式生成应用程序，`TCHAR`便会`typedef`为`char`；当采用UTF-16（WCS）时，则`typedef`为`wchar_t`。（为统一起见，也提供`WCHAR`作为`wchar_t`的同义词。）

在云云Windows API中，前缀或后缀为“w”、“wcs”或“W”表示宽（UTF-16）字符；前缀或后缀为“t”、“tcs”或“T”则表示目前的字符类型（可能是ANSI或UTF-16，

---

[49]译注：Unicode Character Set（UCS）和Multibyte Character Set（MBCS）是不同的字符集。UCS的目标是以一套字符集包揽各种文字，因此其编码（如UTF-8、UTF-16）可以容许在同一个字符串里包含中、日、韩等字符。而MBCS是指一些使用多于1字节的字符集，例如GBK、Big5、Shift-JIS等，通常不能实现多国文字混合使用。

[50]译注：严格地说，UTF-16和UTF-8都是可变长度编码。UTF-16的编码点（code point）需要一或两个16位空间（即2或4字节）。若只支持基本多文种平面（Basic Multilingual Plane，BMP），则每编码点只需16位空间。较旧的UCS-2编码就是只支持BMP的固定长度编码，而这种编码已被UTF-16取代。

[51]译注：“宽”是指每字符占多于8位，在Windows、Java和.Net里都定义为16位，但在一些UNIX上定义为32位。

视乎生成设置）；无前缀、后缀表示旧式普通ANSI。STL也使用相似的命名方式，例如，`std::string`是STL的ANSI字符串类，而`std::wstring`是宽字符串版本。

在Windows下，几乎所有涉及字符串的C标准库函数都有WCS和MBCS版本。遗憾的是API调用不使用UTF-8或UTF-16等术语，而且函数的命名并非完全一致。这对不清楚底蕴的程序员可能造成混淆。（但**读者**不会是其中之一！）表5.1列出一些例子。

ANSI	WCS	MBCS
`strcmp()`	`wcscmp()`	`_mbscmp()`
`strcpy()`	`wcscpy()`	`_mbscpy()`
`strlen()`	`wcslen()`	`_mbslen()`

表 5.1: 常用C标准库字符串函数，其ANSI、宽字符集及多字节字符集的版本。

Windows也提供函数对ANSI、多字节UTF-8、UTF-16字符串之间做转换。例如，`wcstombs()`把宽UTF-16字符串转换为多字节UTF-8字符串。

这些函数的完整说明可参阅微软的MSDN网站。以下是`strcmp()`及其家族的网址，在该页面也可以浏览或搜寻其他相关的字符串操作函数[52]。

游戏机上的Unicode

Xbox 360开发套件（Xbox 360 software development kit，XDK）几乎完全采用WCS字符串，连内部字符串如路径也不例外。这肯定是解决本地化问题的其中一种合理方案，使整个XDK内的字符串处理都是一致的。然而，UTF-16编码有点浪费内存[53]，因此各游戏引擎可能采用不同的处理方式。在顽皮狗，引擎全部采用8位的`char`字符串，并通过UTF-8编码来处理外国文字。选择哪种编码其实并不重要，重要的是能在项目中尽早决定，并贯彻始终地使用。

5.4.4.2 其他本地化要考虑之事

即使读者已在软件上采用Unicode字符，还有许多其他本地化问题要克服。首先，字符串并非本地化问题的全部所在。音频片段，包括录制的语音，也需要翻译。纹理中可能也会绘进英文文字，也需要翻译。许多符号在不同文化中也有不同的意义。就算是一些如禁止吸

[52] `http://msdn.microsoft.com/en-us/library/kk6xf663(VS.80).aspx`

[53] 译注：浪费只对中、日、韩等以外的文字而言。

烟的标志等看似无伤大雅的东西，也可能在另一种文化中造成误解。此外，有些市场的游戏评级界限并不一样。例如，日本少年级别的游戏不容许显示任何种类的血液，而北美则可接受少量血溅。

字符串也有一些细节需要关注。读者需要管理一个数据库，储存所有游戏中玩家能看到的字符串，那么才可以进行完整翻译。字符串的格式可能不一样，例如希伯来文是从右向左阅读的。字符串长度在不同语言中也有很大差异。读者也需要决定要发行单款DVD或蓝光光盘，包含所有语言数据，或是为各地区发行不同的光盘。

本地化系统中，最关键的组件是储存人类可读字符串的数据库，以及在游戏运行时，用标识符查找那些字符串的系统。例如，假设在平视显示器（heads-up display，HUD）中，在“Player 1 Score:”和“Player 2 Score:”标签后显示每位玩家的分数，并在每回合完结时显示“Player 1 Wins”或“Player 2 Wins”。这4段文字储存在本地化数据库时，每段文字要附上唯一并清楚的标识符，例如分别使用“p1score”、“p2score”、“p1wins”、“p2wins”。当把游戏的字符串翻译成法文，数据库就变为如表5.2所示的简单例子。可以为每种语言新增额外的列。

Id	**英文**	**法文**
p1score	“Player 1 Score”	“Grade Joueur 1”
p2score	“Player 2 Score”	“Grade Joueur 2”
p1wins	“Player 1 Wins!”	“Joueur un gagne!”
p2wins	“Player 2 Wins!”	“Joueur deux gagne!”

表 5.2: 用作本地化的字符串数据库例子。

数据库实际使用什么格式，悉随尊便。简单的做法是，可把微软Excel工作表储存为逗号分隔型取值（comma-separated values，CSV）文件，供游戏引擎读取，要复杂的话也可以使用强大的Oracle数据库。字符串数据库的细节对游戏引擎并不太重要，只要能读取字符串标识符，以及按游戏所支持的语言读取对应的Unicode字符串便可。（然而，从实践角度看，按某些游戏工作室的架构情况，一些数据库细节也可能**非常**重要。对有内部翻译人员的小型工作室，可能只需要使用置于网盘上的Excel工作表便行。但对大型工作室，其分部可能遍及英、欧、南美、日本，那么采用某种分布型数据库才更可行。[54]）

[54]译注：译者认为，由于CSV是纯文字文件，因此把CSV文件置于版本控制系统中，便能使用签入 / 签出、合并等功能，一般来说已经足以应付。另外，也可考虑使用如Google Sheets等线上协作试算表软件。

在运行时，须提供简单的函数，按字符串标识符及“当前”的语言，传回相应的Unicode字符串。该函数可声明成这样子[55]：

```
const wchar_t* getLocalizedString(const char* id);
```

使用时的方法是：

```
void drawScoreHud(const Vector3& score1Pos,
                  const Vector3& score2Pos)
{
    renderer.displayTextOrtho(
        getLocalizedString("p1score"), score1Pos);
    renderer.displayTextOrtho(
        getLocalizedString("p2score"), score2Pos);
    // ......
}
```

当然，这也需要某全局方式设定的“当前”语言。其做法可以是在安装游戏时，把配置设定固定下来。或者，可容许玩家在游戏的菜单中即时更改语言。两种方法也容易实现，例如使用一个全局整数变量，表示读取字符串表中的行索引（例如，第1行是英文，第2行是法文，第3行是西班牙文等）。

有了此基础设施之后，所有程序员都要切记，**不要向玩家显示原始字符串**。程序员必须使用数据库中的字符串标识符，并通过查找函数取得所需字符串。[56]

5.5 引擎配置

游戏引擎非常复杂，总是跟随着大量的可调校选项。有些选项通过游戏中的选项菜单提供给玩家调校。例如，游戏中可能会提供有关图形质量、音乐和音效的音量、控制等选项。而另一些选项，则只是为游戏开发团队而设置的，在游戏发行时，这些选项会被隐藏或去除。例如，玩家角色的最高行走速度，在开发期间可以作为选项供微调之用，但在游戏发行前则改为硬编码的值。

---

[55]译注：原文的返回类型是wchar_t，译者做出修正。

[56]译注：需要显示的字符串可包装为一个类，例如名为LocString。查找字符串的函数传回LocString对象，显示字符串相关函数的参数也使用LocString对象。那么，便能避免程序员错误使用原始字符串。但是，此类可能还需要有一些字符串格式化功能，例如把某变量的值代入字符中，如L"Player {0} Score: {1}"。

5.5.1 读 / 写选项

可配置选项可简单实现为全局变量或单例中的成员变量。然而，可配置选项必须供用户配置，并储存至硬盘、记忆卡（memory card）或其他储存媒体，使往后游戏能重读这些选项，否则这些配置选项的用途不大。以下是一些简单读 / 写可配置选项的方法:

- **文本配置文件**：现在最常见的读 / 写配置选项的方法就是，把选项置于一个或多个文本文件。在各游戏引擎中，这类文件的格式不尽相同，但格式通常是非常简单的。例如，Windows的INI文件（也用于OGRE渲染器）是由键值对所构成的，这些键值对以逻辑段分组。

```
[SomeSection]
Key1=Value1
Key2=Value2
[AnotherSection]
Key3=Value3
Key4=Value4
Key5=Value5
```

  此外，在游戏配置选项文件中，XML格式是另一种常见选择。

- **经压缩二进制文件**：多数现代的游戏主机都配备硬盘，但较旧的主机并不享有硬盘这种奢侈品。因此，自超级任天堂（Super Nintendo Entertainment System，SNES[57]）以来的主机都配有专门的可取出记忆卡，做读 / 写数据之用[58]。游戏选项有时会连同游戏存档一起写到这些记忆卡上。使用记忆卡时，常用经压缩二进制文件作为格式，皆因这些记忆卡的储存空间非常有限。

- **Windows注册表**（Windows registry）：微软Windows操作系统提供一个全局选项数据库，名为注册表。注册表以树形式储存，当中的内部节点称为**注册表项**（registry key），作用如文件夹，而叶节点则以键值对储存个别选项。任何应用程序、游戏或其他软件都可预留一整个子树（即注册表项），供该软件专用，并在该子树下存储任意的选项集。Windows注册表好像一个悉心管理的INI文件集合，并且实际上，Windows引进注册表的目的是取缔供操作系统和应用程序所使用的无限膨胀INI文件。

- **命令行选项**：可扫描命令行去取得选项设置。引擎可提供机制，使所有游戏中的选项都能经命令行设置；或者，引擎只向命令列显露所有选项中的一小子集。

- **环境变量**（environment variable）：在运行Windows、Linux或Mac OS的个人电脑中，环境变量有时候也用于存储一些配置选项。

- **线上用户设定档**（online user profile）：随着如Xbox Live等线上游戏社区的来临，每

[57]译注：此为欧美采用的名字，日本则称为スーパーファミコン（Super Famicom），或简称SFC。

[58]译注：事实上，超级任天堂并不支持可取出的记忆卡，但其卡带有内置数据储存功能。

位用户都能建立设定档，并用它来存储成就（achievement）、已购买或解锁的游戏内容、游戏选项及其他信息。由于这些数据储存在中央服务器中，只要连上互联网，无论何地玩家都能存取数据。

5.5.2 个别用户选项

多数游戏引擎会区分全局选项和个别用户选项（per-user option）。这是有需要的，因为多数游戏容许每个玩家配置其喜欢的选项。此概念对游戏开发其间也十分有用，因为每位程序员、美术设计师、游戏设计师都能自定义其工作环境，而不会影响其他队员。

显然，储存个别用户选项必须小心，每个玩家只能“看见”自己的选项，而不会遇见其他玩家在同一计算机或游戏主机上的选项。在游戏主机上，用户通常可以把游戏进度，以及如控制器等个别用户选项，一并储存至记亿卡或硬盘的“位置（slot）”中。这些位置通常实现为储存媒体上的文件。

在Windows机器上，每位用户在**C:\Documents and Settings**中各有其文件夹，用以储存该用户的信息，如该用户的桌面、“我的文件”文件夹、互联网浏览历史、临时文件等。当中有一个名为**Application Data**的隐藏文件夹，用来储存个别应用程序的个别用户数据。每个应用程序在**Application Data**文件夹下建立文件夹，并把所需的个别用户数据储存至此。

Windows游戏有时候会把个别用户选项储存至注册表。注册表以树的形式构成，在根节点下，有一个称为HKEY_CURRENT_USER的顶层子节点，用来储存登入后用户的设置。在注册表中，每位用户有其个别的注册表子树（位于顶层子节点HKEY_USERS之下），而HKEY_CURRENT_USER其实只是当前用户子树的别名。因此，游戏及其他应用软件可以通过读 / 写HKEY_CURRENT_USER下的注册表项，管理当前用户的配置选项。

5.5.3 真实引擎中的配置管理

本节将检阅几个真实游戏引擎如何管理其配置选项。

5.5.3.1 例子：雷神之锤的CVAR

雷神之锤引擎家族使用一个名为**主控台变量**（console variables，CVAR）的配置管理系统。其实，CVAR只不过是一个储存浮点数或字符串的全局变量，其值可于雷神之锤的主控台下查看及修改。部分CVAR的值也可储存至硬盘上，供引擎在之后重新载入。

在运行时，多个CVAR储存为全局键表。每个CVAR是动态配置的`struct cvar_t`实例，含变量的名字、浮点数或字符串值、标志位（bit flag）集合，以及指向链表中下一个CVAR的指针。读取CVAR的方法是调用`Cvar_Get()`函数，若该名字的变量不存在就会创建一个；修改CVAR则调用`Cvar_Set()`。其中一个标志位`CVAR_ARCHIVE`控制变量是否储存至配置文件**config.cfg**。若设置了此标志，该CVAR的值就能在多次游戏中持续。

5.5.3.2 例子：OGRE

OGRE渲染引擎采用一组Windows INI格式的文本文件做配置选项之用。选项默认会存进3个文件，这些文件都是位于可执行文件的文件夹中。

- **plugins.cfg**含要启用的可选插件，以及插件在盘上的位置。
- **resources.cfg**含游戏资产（即媒体、资源）的搜寻路径。
- **ogre.cfg**含丰富的选项，设置使用哪款渲染器（DirectX或OpenGL），以及喜好的视频模式、屏幕大小等。

OGRE并无开箱即用的储存个别用户选项机制。然而，OGRE提供免费源代码，因此很容易可以修改代码，以在用户的**C:\Documents and Settings**文件夹搜寻配置文件，而非在可执行文件的文件夹中搜寻。此外，`Ogre::ConfigFile`类也能用来轻易地读/写全新的配置文件。

5.5.3.3 例子：《神秘海域：德雷克船长的宝藏》

顽皮狗的神秘海域引擎使用多种配置机制。

游戏内置菜单设置

神秘海域引擎支持强大的游戏内置菜单系统，让开发者掌控全局配置选项及调用命令。可配置选项的数据类型必须相对简单（主要为布尔、整数、浮点数变量），但此并不会限制神秘海域开发者建立数以百计的有用菜单驱动选项。

每个可配置选项都实现为全局变量。为配置选项创建菜单项目时，会提供该全局变量的地址，之后菜单项目就能直接控制该全局变量的值。例如，以下函数创建一个子菜单项，当中的几个选项用于神秘海域路轨车辆（即用于“逃走”一章，吉普车追逐关卡[59]的车辆）。此函数定义的菜单项目，负责控制3个全局变量：两个布尔值及一个浮点值。函数中把3个菜

[59]译注：即《神秘海域》的第7章“Out of the Frying Pan”。吉普车是由非玩家角色控制的，按预定路线移动。

单项目组成一个菜单，并传回一个特殊菜单项目，当该项目被选取时就会打开菜单。可以假设，在建立菜单时，调用此函数的代码会把传回的菜单项目加进父菜单里。

```
DMENU::ItemSubMenu* CreateRailVehicleMenu()
{
    extern bool g_railVehicleDebugDraw2D;
    extern bool g_railVehicleDebugDrawCameraGoals;
    extern float g_railVehicleFlameProbability;

    DMENU::Menu* pMenu = new DMENU::Menu("RailVehicle");

    pMenu->PushBackItem(new DMENU::ItemBool(
        "Draw 2D Spring Graphs", DMENU::ToggleBool,
        &g_railVehicleDebugDraw2D));

    pMenu->PushBackItem(new DMENU::ItemBool(
        "Draw Goals (Untracked)", DMENU::ToggleBool,
        &g_railVehicleDebugDrawCameraGoals));

    DMENU::ItemFloat* pItemFloat = new DMENU::ItemFloat(
        "FlameProbability", DMENU::EditFloat, 5, "%5.2f",
        &g_railVehicleFlameProbability));

    pItemFloat->SetRangeAndStep(0.0f, 1.0f, 0.1f,  0.01f);
    pMenu->PushBackItem(pItemFloat);

    DMENU::ItemSubmenu* pSubmenuItem = new DMENU::ItemSubmenu(
        "RailVehicle...", pMenu);

    return pSubmenuItem;
}
```

当轻松按下PS3手柄的圆形按钮时，引擎便会储存当前菜单项目所对应的选项内容。菜单设置会写进INI风格的文本文件，使存了档的全局变量在多次游戏运行中能继续保持。系统中可**按个别菜单项目**设置是否存档，这是非常有用的功能，因为没有存档的选项会使用程序员的预设值。若程序员改变预设置，所有用户便能“看到”新的值，当然，除非某用户曾更改并储存该选项。

命令行参数

神秘海域引擎对命令行扫描一组预定义的特殊选项。当中可指定要载入的关卡名称，再加上一些常用参数。

Scheme数据定义

神秘海域引擎中，绝大多数的引擎和游戏配置信息，采用了Scheme语言（Lisp的方言之一）定义。利用自建的数据编译器，将Scheme语言里定义的数据结构转换为二进制文件，供游戏引擎读取。数据编译器也会生成头文件，内含C `struct`声明，对应每个Scheme中定义的数据类型。引擎利用这些头文件正确地解释二进制文件内的数据，甚至可以在运行期间，重编译及重载这些二进制文件，使开发者能修改Scheme中的数据后立即看到其运行效果（只要修改不涉及增加、减少数据成员，因为这种改动必须重新编译引擎）。

以下是一个为动画定义属性的例子，向游戏导出3个动画。读者可能未曾读过Scheme代码，但这段代码应该很简单，应该不解自明。其中较奇特的是，Scheme容许在命名中使用连字号（不像C/C++中，`simple-animation`会被理解成`simple`减去`animation`）。

```
;; simple-animation.scm
;; 定义一个新的数据类型，名为simple-animation
(deftype simple-animation ()
    (
        (name                string)
        (speed               float   :default 1.0)
        (fade-in-seconds     float   :default 0.25)
        (fade-out-seconds    float   :default 0.25)
    )
)

;; 定义此数据结构的3个实例
(define-export anim-walk
    (new simple-animation
        :name   "walk"
        :speed  1.0
    )
)

(define-export anim-walk-fast
    (new simple-animation
        :name   "walk"
        :speed  2.0
    )
)

(define-export anim-jump
    (new simple-animation
        :name               "jump"
        :fade-in-seconds    0.1
```

```
        :fade-out-seconds    0.1
    )
)
```

此Scheme代码会产生以下的C/C++头文件：

```
// simple-animation.h
// 警告: 本文件是Scheme自动生成的，不要手工修改
struct SimpleAnimation
{
    const char*     m_name;
    float           m_speed;
    float           m_fadeInSeconds;
    float           m_fadeOutSeconds;
};
```

在游戏中，可调用LookupSymbol()函数读取数据，该函数以返回类型为模板参数：

```
#include "simple-animation.h"

void someFunction()
{
    SimpleAnimation* pWalkAnim =
        LookupSymbol<SimpleAnimation*>("anim-walk");

    SimpleAnimation* pFastWalkAnim =
        LookupSymbol<SimpleAnimation*>("anim-walk-fast");

    SimpleAnimation* pJumpAnim =
        LookupSymbol<SimpleAnimation*>("anim-jump");

    // 在此使用这些数据……
}
```

此系统给予程序员巨大的弹性，定义不同种类的配置数据，无论是简单的布尔、浮点、字符串选项，以至于复杂、巢状或互相连接的数据结构。此系统可用来定义细致的动画树[60]、物理参数、游戏机制等。

[60]译注：在11.11节会详述神秘海域引擎如何利用Scheme做动画相关的配置。

第6章　资源及文件系统

游戏本质上是多媒体体验。因此，载入及管理多种媒体，是游戏引擎必须具备的能力。这些媒体包括纹理位图、三维网格数据、动画、音频片段、碰撞和物理数据、游戏世界布局等许多种类。除此以外，由于内存空间通常不足，游戏引擎要确保在同一时间，每个媒体文件只可载入一份。例如，5个网格模型都共享同一张纹理，那么该纹理在内存中只应有1份，而不是5份。多数游戏引擎会采用某种类型的**资源管理器**（resource manager）（又称作**资产管理器**/asset manager，或**媒体管理器**/media manager），载入并管理构成现代三维游戏所需的无数资源。

每个资源管理器都会大量使用文件系统。在个人计算机中，程序员是通过操作系统调用的程序库存取文件系统的。然而，游戏引擎有时候会“包装”原生的文件系统API，成为引擎私有的API，其主要原因有二。首先，引擎可能需要跨平台，此需求下，引擎自己的文件系统API就能对系统其他部分产生隔离作用，隐藏不同目标平台之间的区别。其次，操作系统的文件系统API未必能提供游戏引擎所需的功能。例如，许多引擎支持**串流**（streaming）（即能够在游戏运行中，同时载入数据），但多数操作系统不直接提供流功能的文件系统API。游戏机用的游戏引擎也要提供多种可移动和不可移动的储存媒体，从内存棒、可选购的硬盘、DVD盘、蓝光光盘以至网络文件系统（如Xbox Live或PlayStation网络PSN）。多种媒体之间的区别也同样可以用游戏引擎自身的文件系统API加以“隐藏”。

本章我们会探索现代三维游戏引擎中的各种文件系统API，再分析典型资源管理器的运作方式。

6.1　文件系统

游戏引擎的文件系统API通常提供以下几类功能。

- 操作文件名和路径。

- 开、关、读、写个别文件。
- 扫描目录下的内容。
- 处理异步文件输入 / 输出（I/O）请求（做串流之用）。

以下各节将分别简述这些功能。

6.1.1 文件名和路径

路径（path）是一种字符串，用来描述文件系统层次中文件或目录的位置。每个操作系统都有少许不同的路径格式，但所有操作系统的路径本质上有相同的结构。路径一般是以下的形式：

卷/目录1/目录2/.../目录N/文件名

或

卷/目录1/目录2/.../目录$(N-1)$/目录N

换言之，路径通常包含一个可选的**卷指示符**（volume specifier）紧接一串**路径成分**，它们之间以路径分隔符（path separator）分隔，例如用正斜线符（/）或反斜线符（\）。每个路径成分是从根目录至目标目录或文件之间的目录名称。若路径指向文件，则最后的是文件名，否则最后的是目标目录名称。路径中要指明根目录，通常是由可选的卷指示符接连路径分隔符（例如UNIX上的/，Windows上的C:\）的。

6.1.1.1 操作系统间之区别

每个操作系统在常规的路径结构上都会加入少许变化。以下列出微软DOS、微软Windows、UNIX操作系统家族及苹果Macintosh操作系统之间的一些重要区别。

- UNIX使用正斜线符（/）作为路径分隔符，而DOS及早期版本的Windows则采用反斜线符（\）。较新版本的Windows容许以正反斜线符分隔路径成分，然而有些应用程序仍然不接受正斜线符。
- Mac OS 8和9采用冒号（:）作为路径分隔符。而Mac OS X是基于UNIX的，因此它支持UNIX的正斜线符记号法。
- UNIX及其变种不支持以卷分开目录层次。整个文件系统都是以单一庞大的层次所组成的。本机磁盘、网络磁盘以及其他资源都是**挂接**（mount）为主层次中的某个子树。因此，UNIX不会出现卷指示符。

- 在微软Windows上，可以用两个方法定义卷。本机磁盘以单英文字母再加冒号指明（例如无处不在的C:）。而远端网络分享则可以挂接成为像本机磁盘一样，或是可以用双反斜线号加上远端计算机名字和分享目录/资源名字指明（如\\some-computer\some-share）。这种双反斜线号是通用命名规则（universal naming convention，UNC）的例子。
- DOS和早期版本的Windows下，文件名只能含最多8个字符，以点号分隔后有3字符**扩展名**。扩展名描述文件的类型，例如.txt是文本文件（text file）、.exe是可执行文件（executable file）。后期的Windows下，文件名可包含多个点号（如UNIX一样），但是，许多应用程序（包括Windows资源管理器）仍然会把最后一个点号后的字符诠释为文件名的扩展名。
- 每个操作系统都会禁止某些字符出现在文件和目录名称中。例如，在Windows或DOS路径中，除了卷指示符后，冒号不能置于其他地方。有些操作系统容许部分保留字符出现于路径内，但整个路径要加上引号，或是在违规字符前加上**换码符**（escape character），如反斜线号。例如，Windows下的文件名和目录名可含空白符，某些情况下此类路径必须加上双引号。
- UNIX和Windows皆有**当前工作目录**（current working directory/CWD或present working directory/PWD）的概念。在这两个操作系统的命令壳层（command shell）里，都可以用cd（**更改目录**）命令设置当前工作目录。要取得当前工作目录，Windows下可键入无参数的cd命令，而在UNIX下则可以执行pwd命令。在UNIX下只有一个当前工作目录，而在Windows下，每个卷有其个别当前工作目录。
- 支持多卷的操作系统（如Windows）也有**当前工作卷**（current working volume）的概念。在Windows 命令行，输入盘的字母再加上冒号，按Enter键，就能改变当前工作卷（如C:<Enter>）。
- 游戏机通常有一组预定义的路径前缀去表示多个卷。例如，PS3使用/dev_bdvd/前缀去指明蓝光驱动，而/dev_hdd$x$/则代表多个硬盘（x为设备的索引）。在PS3开发机（PS3 development kit）上，/app_home/会映射至用户定义的开发主机路径[1]。在开发期间，游戏通常从/app_home/读取资产，而不是从游戏主机本身的蓝光光盘或硬盘中读取。

[1]译注：例如用Windows机器连接PS3开发机，可以把D:\alice2设为开发机上的/app_home/。当开发机要打开/app_home/Config/AliceGame.ini，就会通过网络读取D:\alice2\Config\AliceGame.ini。

6.1.1.2 绝对和相对路径

所有路径都对应文件系统中的某个位置。当路径相对于根目录，我们称其为**绝对路径**（absolute path）。当路径相对于文件系统层次架构中的**其他**目录，则称之为**相对路径**（relative path）。

在UNIX和Windows下，绝对路径的首字符为路径分隔符（/或\），而相对路径则不会以路径分隔符作为首字符。Windows中，绝对路径和相对路径都可以加入卷指示符，不加入卷指示符代表使用当前工作卷。

以下是一些绝对路径的例子。

Windows

- `C:\Windows\System32`
- `D:\` （D:卷的根目录）
- `\` （当前工作卷的根目录）
- `\game\assets\animation\walk.anim` （当前工作卷）
- `\\joe-dell\Shared\_Files\Images\foo.jpg` （网络路径）

UNIX

- `/usr/local/bin/grep`
- `/game/src/audio/effects.cpp`
- `/` （根目录）

以下是相对路径的例子。

Windows

- `System32` （若当前工作目录为\Windows，则是指当前工作卷的\Windows\System32）
- `X:animation\walk.anim` （若X:的当前工作目录是\game\assets，则是指X:\game\assets\animation\walk.anim）

UNIX

- `bin/grep` （若当前工作目录为/usr/local，则是指/usr/local/bin/grep）
- `src/audio/effects.cpp` （若当前工作目录为/game，则是指/game/src/audio/effects.cpp）

6.1.1.3 搜寻路径

不要混淆**路径**和**搜寻路径**（search path）这两个术语。**路径**是代表文件系统下某文件或目录的字符串。**搜寻路径**是含一串路径的字符串，各路径之间以特殊字符（如冒号或分号）分隔，找文件时就会从这些路径进行搜寻。例如在命令行下执行程序，操作系统会首先看看当前目录下有没有该可执行文件，若没有则会从PATH环境变量中的路径搜寻该可执行文件。

有些游戏引擎会使用搜寻路径找资源文件。例如，OGRE渲染引擎有一个resources.cfg文本文件，它会使用当中的搜寻路径。此文件包含目录及ZIP存档列表，要找资产时，就会从该列表按序进行搜寻。然而，在运行时期搜寻资产，可能是费时的做法。通常，资产路径没理由会在运行时期之前无法得知。若然如此，我们应能完全避免搜寻资产，这显然是更优越的做法。

6.1.1.4 路径API

路径显然比简单字符串复杂得多。程序员需要对路径进行多种操作，例如，从路径分离目录 / 文件名 / 扩展名、使路径规范化（canonicalization）、绝对路径和相对路径之间进行转换等。含丰富功能的路径API对这些任务非常有用。

Windows为此提供一组API，由shlwapi.dll动态程序库实现，并提供shlwapi.h头文件。MSDN提供其详细说明文件[2]。

当然，shlwapi只能用于win32平台。索尼也为PS3提供了类似的API。但若要开发跨平台游戏引擎，便不能直接使用平台相关的API。游戏引擎也未必需要shlwapi的所有功能。因此，游戏引擎通常会实现轻量化的路径处理API，符合引擎专门需求，并能为引擎的所有目标平台工作。这种API可以是对原生API的简单包装，也可以从零开始实现。

6.1.2 基本文件I/O

C标准程序库（standard C library）提供两组API以开启、读取及写入文件内容，两组API中一组有缓冲功能（buffered），另一组无缓冲功能（unbuffered）。每次调用输入 / 输出（input/output，I/O），都需要称为**缓冲区**的数据区块，以供程序和磁盘之间传送来源或目的字节。当API负责管理所需的输入 / 输出数据缓冲，就称之为有缓冲功能的I/O API。相反，若需要由程序员负责管理数据缓冲，则称为无缓冲功能的API。C标准程序库

[2] http://msdn2.microsoft.com/en-us/library/bb773559(VS.85).aspx

中，有I/O缓冲功能的函数有时候会称为**流输入 / 输出**（stream I/O）API，因为这些API把磁盘文件抽象成字节流。

C标准程序库中，含缓冲功能和没有缓冲功能的API列于表6.1。[3]

操作	有缓冲功能	无缓冲功能
开启文件	`fopen()`	`open()`
关闭文件	`fclose()`	`close()`
读取文件	`fread()`	`read()`
写入文件	`fwrite()`	`write()`
移动访问位置	`fseek()`	`seek()`
返回当前位置	`ftell()`	`tell()`
读入单行	`fgets()`	无
写入单行	`fputs()`	无
格式化读取	`fscanf()`	无
格式化写入	`fprintf()`	无
查询文件状态	`fstat()`	`stat()`

表 6.1: 有缓冲功能和无缓冲功能的标准C程序库。

C标准程序库的I/O函数有详细说明文档，在此就不重复赘述其细节了。关于微软实现的有缓冲（流I/O）API，可参考此网页[4]；其无缓冲（低阶I/O）API，则可参考此网页[5]。

在UNIX及其变体中，C标准库的无缓冲I/O函数是原生的操作系统调用。然而，在微软Windows上，这些函数仅仅是底层API的包装。Win32 API的`CreateFile()`能建立或开启文件做读 / 写之用，`ReadFile()`及`WriteFile()`分别负责读 / 写数据，而`CloseFile()`则负责关闭已开启文件的句柄。相对于C标准程序库函数，使用低阶系统调用的优点是能运用原生文件系统的所有细节功能。例如，用Windows的原生API可以询问及改变文件的安全属性，C标准库则不可行。

有些游戏开发团队认为，管理自己的缓冲区是有帮助的。例如，艺电的《红色警戒3

[3]译注：实际上C标准程序库并没有定义`open()`系列函数，只有`fopen()`系列函数。这类API是个别平台提供的，例如，UNIX下的POSIX标准、微软Visual C++的CRT。而CRT提供的API则加上下画线前缀，如`_open()`。实际上，`open()`系列是为了提供更多平台相关特性的低阶函数，与是否支持缓冲无直接关系。

[4]http://msdn2.microsoft.com/en-us/library/c565h7xx(VS.71).aspx

[5]http://msdn2.microsoft.com/en-us/library/40bbyw78(VS.71).aspx

(Red Alert 3)》团队观察到，往日志里写数据会显著降低性能。他们更改日志系统，先把数据累积在内存缓冲，满溢后才写进盘内。之后，他们再把缓冲输出函数置于另一线程里，以避免令主游戏循环发生流水线停顿（stall）。

6.1.2.1 包装还是不包装

开发游戏引擎时，可使用C标准库的I/O函数，或是操作系统的原生API。然而，许多游戏引擎都会把文件I/O API**包装**成为自定义的I/O函数。包装操作系统的I/O API，最少有3个好处。第一，引擎程序员能保证这些自定义的API在所有目标平台上均有相同行为，就算某平台上的原生程序库本身有不一致性或臭虫也如是。第二，API可以简化，只剩下引擎实际需要的函数，使维护开支维持最少限度。第三，可提供延伸功能。例如，引擎的自定义包装API可处理不同媒体上的文件，无论是硬盘、游戏机上的DVD-ROM / 蓝光光盘，或网络上的文件（例如，由Xbox Live或PSN管理的远端文件），也可处理记忆棒（memory stick）或其他类型的可移除媒体。

6.1.2.2 同步文件I/O

C标准库的两种文件I/O库都是**同步的**（synchronous），即是说程序发出I/O请求以后，必须等待读 / 写数据完毕，程序才继续运行。以下代码片段示范如何使用同步I/O函数`fread()`，把整个文件的内容读入内存中的缓冲里。注意`syncReadFile()`函数直至所有数据读进缓冲后才返回。

```
bool syncReadFile(const char* filePath,
    U8* buffer, size_t bufferSize, size_t& rBytesRead)
{
    FILE* handle = fopen(filePath, "rb");
    if (handle)
    {
        // 在这里阻塞，直至所有数据都读取完毕
        size_t bytesRead = fread(buffer, 1, bufferSize, handle);
        int err = ferror(handle);    // 若过程出错，取得错误码
        fclose(handle);
        if (0 == err)
        {
            rBytesRead = bytesRead;
            return true;
        }
    }
}
```

```
    return false;
}

void main(int argc, const char* argv[])
{
    U8 testBuffer[512];
    size_t bytesRead = 0;

    if (syncReadFile("C:\\testfile.bin",
        testBuffer, sizeof(testBuffer), bytesRead))
    {
        printf("success: read %u bytes\n", bytesRead);
        // 可以在此使用缓冲的内容……
    }
}
```

6.1.3 异步文件I/O

串流（streaming）是指在背景载入数据，而主程序同时继续运行。为了让玩家领略无缝（seamless）、无载入画面的游戏体验，许多游戏在游戏进行的同时使用串流从DVD-ROM、蓝光光盘或硬盘读取即将来临的关卡数据。最常见的串流数据类型可能是音频和纹理，但其他数据也可以串流，例如几何图形、关卡布局、动画片段等。

为了支持串流，必须使用**异步**（asynchronous）文件I/O库。这种库能让程序在请求I/O后，不需要等待读 / 写完成，程序便立即继续运行。有些操作系统自带提供异步文件I/O库。例如，Windows通用语言运行平台（common language runtime，CLR）（CLR即Visual Basic.Net、C#、managed C++及J#等语言所采用的虚拟机器）提供了`System.IO.BeginRead()`及`System.UI.BeginWrite()`等函数。PlayStation 3也提供了名为`fios`的异步API。若目标平台不提供异步I/O库，也可自行开发一个[6]。即使非必要从零开始实现，包装系统API以提高可移植性也是好主意。

以下代码片段展示如何使用异步读取操作，把整个文件内容读入内存中的缓冲里。注意`asyncReadFile()`是立即返回的，要等待I/O库调用`asyncReadComplete()`回调函数时，才确保全部数据已读入缓冲里。

[6]译注：假设目标平台能提供线程或类似的功能。

```
AsyncRequestHandle g_hRequest;   // 异步I/O请求的句柄
U8 g_asyncBuffer[512];           // 输入缓冲

static void asyncReadComplete(AsyncRequestHandle hRequest);

void main(int argc, const char* argv[])
{
    // 注意：在此调用asyncOpen()可能本身是异步的，但这里会忽略此细节，
    // 假设该函数是阻塞的
    AsyncFileHandle hFile = asyncOpen("C:\\testfile.bin");

    if (hFile)
    {
        // 此函数做读取请求，然后立即返回(非阻塞)
        g_hRequest = asyncReadFile(
            hFile,                      // 文件句柄
            g_asyncBuffer,              // 输入缓冲
            sizeof(g_asyncBuffer),      // 缓冲大小
            asyncReadComplete);         // 回调函数
    }

    // 然后我们就可以开始循环 (此循环模拟等待读取时要做的一些真实的工作)
    for (;;)
    {
        OutputDebugString("zzz...\n");
        Sleep(50);
    }
}

// 当数据都读入时，就会调用此函数
static void asyncReadComplete(AsyncRequestHandle hRequest)
{
    if (hRequest == g_hRequest && asyncWasSuccessful(hRequest))
    {
        // 现在数据已读进g_asyncBuffer[]。查询实际读入的字节数量
        size_t bytes = asyncGetBytesReadOrWritten(hRequest);

        char msg[256];
        sprintf(msg, "async success, read %u bytes\n", bytes);
        OutputDebugString(msg);
    }
}
```

多数异步I/O库容许主程序在请求发出后一段时间，等待I/O操作完成才继续运行。若然只需要在等待I/O期间做些有限的工作，这种方式就显得有用。以下代码示范这种应用。

```
U8 g_asyncBuffer[256];          // 输入缓冲

void main(int argc, const char* argv[])
{
    AsyncRequestHandle hRequest = ASYNC_INVALID_HANDLE;
    AsyncFileHandle hFile = asyncOpen("C:\\testfile.bin");

    if (hFile)
    {
        // 此函数做读取请求，然后立即返回(非阻塞)
        hRequest = asyncReadFile(
            hFile,                          // 文件句柄
            g_asyncBuffer,                  // 输入缓冲
            sizeof(g_asyncBuffer),          // 缓冲大小
            NULL);                          // 无回调函数
}

// 现在做一点工作……
for (int i = 0; i < 10; ++i)
{
    OutputDebugString("zzz...\n");
    Sleep(50);
}

// 直至数据预备好之前，我们不能继续下去，所以要在此等待
asyncWait(hRequest);

if (asyncWasSuccessful(hRequest))
    {
        // 现在数据已读进g_asyncBuffer[]。查询实际读入的字节数量
        size_t bytes = asyncGetBytesReadOrWritten(hRequest);

        char msg[256];
        sprintf(msg, "async success, read %u bytes\n", bytes);
        OutputDebugString(msg);
    }
}
```

有些异步I/O库容许程序员取得某异步操作所需时间的估算。一些API也可以为请求设置**时限**（deadline）（实际上会为待完成的请求划分优先次序），并可设置请求超出时限时的安排（例如取消请求、通知程序、继续尝试等）。

6.1.3.1 优先权

必须谨记文件I/O是实时系统（real-time system），如同游戏的其他部分也要遵循时限。因此，异步I/O操作常有不同的优先权（priority）。例如，当要从硬盘或蓝光光盘串流音频，并要在串流时播放，那么，载满下个音频缓冲的优先权，和载入纹理或游戏关卡块的优先权相比，前者显然高于后者。异步I/O系统必须能暂停较低优先权的请求，才可以让较高优先权的I/O请求有机会在时限前完成。

6.1.3.2 异步文件I/O如何工作

异步文件I/O是利用另一线程处理I/O请求的。主线程调用异步函数时，会把请求放入一个队列，并立即传回。同时，I/O线程从队列中取出请求，并以阻塞（blocking）I/O函数如`read()`或`fread()`处理这些请求。请求的工作完成后，就会调用主线程之前提供的回调函数，告之该操作已完成。若主线程选择等待完成I/O请求，就会使用**信号量**（semaphore）处理。（每个请求对应一个信号量，主线程把自身处于休眠状态，等待I/O线程在完成请求工作后通知信号量。）

几乎任何可以想象到的同步操作，都能通过把代码置于另一线程而转变为异步操作。除了线程，也可以将代码移至物理上独立的处理器，例如PS3上的6个协同处理器（synergistic processing unit，SPU）之一。7.6节会有详细说明。

6.2 资源管理器

每个游戏都是由种类繁多的**资源**（有时称为**资产**或**媒体**）构成的，例如有网格、材质、纹理、着色器程序、动画、音频片段、关卡布局、碰撞数据、物理参数等。游戏资源必须妥善管理，这包括两方面，一方面是建立资源的离线工具，另一方面是在执行期载入、卸下及操作资源。因此，每个游戏都有某形式的**资源管理器**（resource manager）。

每个资源管理器都由两个元件组成，这两个元件既独立又互相整合。其一负责管理离线工具链，用来创建资产及把它们转换成引擎可用的形式。另一元件在执行期管理资源，确保资源在使用之前已载入内存，并在不需要的时候把它们从内存卸下。

在某些引擎中，资源管理器是一个具清晰设计、统一、中心化的子系统，负责管理游戏中用到的所有类型资源。其他引擎的资源管理器本身不是单独子系统，而是散布于不同的子系统中，或许这些子系统是由不同作者，经历过引擎漫长的、也许多姿多彩的历史而写成的。但无论资源管理器是如何实现的，它总是要负起某些责任，并解决一些有明确定义的问

题。本节会探讨典型游戏引擎资源管理器的功能，以及其实现细节。

6.2.1 离线资源管理及工具链

6.2.1.1 资产的版本控制

小型游戏项目中，游戏资产的管理方式可以是把组织不严谨的文件以项目特设的目录结构置于共用网盘中。但此方式对于现代商业三维游戏来说并不可行，因为这些游戏包含海量的各种资产。所以这类项目的开发团队需要一套更正式的方法来跟踪及管理资产。

有些游戏团队使用源代码版本控制系统管理资源。美术人员会把美术源文件（Maya场景、Photoshop .PSD文件、Illustrator文件等）签入Perforce或类似的套件。这种方法尚算行之有效，虽然有些游戏团队仍须为美术人员组建量身打造的软件，以压低他们的学习曲线。这类软件可以仅为商用版本控制系统的包装，也可以是完全定制的。

解决数据量的问题

艺术资产的版本控制，其最大问题在于极大的数据量。尽管相对于项目的影响，C++和脚本源程序文件都很小，而艺术文件则大得多。因为许多源文件控制系统都需要把文件从中央版本库复制至用户的本地机器，极大的资产文件可以导致这些套件完全无用。

笔者受雇于不同工作室之时，曾看过此问题的不同解决方案。有些工作室转而使用如Alienbrain这种特别针对极大量数据而设的商业版本控制系统。有些团队则简单承受这些数据量，让版本控制工具复制资产到本地机器。只要磁盘空间足够、网络频宽足够，此法还是可行的，但是它也可能是低效的，降低团队的生产力。有些团队在其版本控制工具上精心制作了一个系统，保证某终端用户只会取得其真正需要的文件至本地机器。此模型中，用户不是无权取得余下的版本库，就是按需从共享网盘中存取。

顽皮狗使用私有工具解决此问题。该工具利用UNIX的符号链接（symbolic link）去消除数据复制，同时容许每位用户拥有资产版本库的完整本地视图。只要文件未签出，该文件其实是符号链接，连至共享网盘上的主文件。符号链接占用很小的本地磁盘空间，因为它仅是一个目录条目。当用户为了编辑而签出文件，就会移除符号链接，并更换为一个本地副本。当用户完成编辑并签入文件，本地副本就会成为新的主文件，其版本记录会在主数据库中被更新，而本地文件又再一次变为符号链接。此系统非常有效，但需要团队从零开始自建其版本控制系统。笔者未发现任何商业工具能以此形式运作。并且，符号链接是UNIX的特性，这类工具或可使用Windows的junction实现，但笔者未曾见过有人做过这项尝试。

6.2.1.2 资源数据库

对于大部分资产来说，游戏引擎并不会使用其原本的格式，下节会深入探讨这部分。资产需要经过一些资产调节管道（asset conditioning pipeline，ACP），用来把资产转换为引擎所需的格式。当流经资产调节管道时，每个资源都需要有些**元数据**（metadata）描述**如何**对资源进行处理。例如，要压缩某纹理位图，便要得悉该用哪种**类型**的压缩方法才最合适。要导出动画片段，便需要知道要导出Maya中哪个范围的帧。若要从含有多个角色的Maya场景中导出角色网格，便需要知道每个网格对应哪个游戏角色。

为了管理所有这类元数据，便需要有某种形式的数据库。若只是制作非常小型的游戏，此数据库可能只需要放在开发者的脑海中。此时此刻，笔者好像能听到脑中数据库说："谨记：玩家的动画需要开启'反转X轴'设置，而其他角色必须**关闭**此设置……噢，应该是相反么？"

显然，对于任何有相当大小的游戏，我们不可依赖开发者的记忆，以这种方式工作。一方面，极大量的资产很快会变得无法驾驭。而逐一人手处理每个资产也太费时费事，对于大型的商业游戏制作是不现实的。因此，每个专业的游戏团队都有某种半自动资源管道，而管道所需的数据则储存在某种**资源数据库**（resource database）中。

在各游戏引擎中，资源数据库的形式有巨大差异。某引擎中，元数据可能会被嵌入至资产源文件本身（如把元数据储存在Maya文件中的所谓blind data里）。另一引擎中，每个源资源文件可能会伴随一个小文本文件，该文件描述应如何处理对应的资源。又另一引擎中，资源生成元数据会写进一组XML文件，或许再通过一些自建的图形界面去管理这些文件。有些引擎则采用真正的关联式数据库（relational database），如微软Access、MySQL，甚至是重量级的数据库，如Oracle。

无论资源数据库采用什么形式，它都必须提供以下的功能。

- 能处理多种**类型**的资源，理想地（但肯定非必要）是以一致的方式处理。
- 能创建新资源。
- 能删除资源。
- 能查看及修改现存的资源。
- 能把资源从一个位置移至磁盘上另一位置。（这是非常有用的，因为美术人员及游戏设计师经常要重新安排资产，以反映项目目标的改动、重新思考游戏设计、新增或取消特性等）。
- 能让资源交叉引用其他资源（例如，网格引用材质、某关卡引用一组动画）。交叉引用通常同时驱动资源管理生成过程及运行时的载入过程。

- 能维持数据库内所有交叉引用的**引用完整性**（referential integrity）。执行所有常见操作后，如删除或移动资源，仍能保持引用完整性。
- 能保存版本历史，并含完整日志记录改动者及事由。
- 资源数据库若能支持不同形式的搜寻及查询，将十分有用。例如，开发者可能想了解哪一关用了某动画、哪些材质引用了某纹理。或是开发者只是找到一个正好遗忘了名字的资源。

从以上列表可以得知，建立一个可靠及稳健的资源管理器并非小事一桩。若能完善地设计及实现，资源管理器简直能令项目成果天差地别——让团队发行一款热门游戏，或是让团队白花18个月最后被管理层中止项目（甚至更坏的情况）。笔者对此坚信不移，皆因这两种情况都亲身经历过。

6.2.1.3 一些成功的资源数据库设计

每个游戏团队都有不同的需求，导致在设计其资源数据库时有不同的抉择。然而，无论是否有价值，以下是笔者亲身用过、能有效工作的设计。

虚幻3

虚幻的资源数据库由其万用工具UnrealEd所管理。UnrealEd几乎负责一切事项，无论是资源元数据管理、资产创建、关卡布局等都一律包办。UnrealEd虽有缺点，但其最重要的好处在于，UnrealEd是游戏引擎本身的一部分。这种设计使UnrealEd能在创建资产之后，立即看到资产在游戏中运行时的模样。游戏也可以在UnrealEd中运行，便能观察资产在其自然环境中的样子，并能看到资产如何在游戏中运作。

笔者称UnrealEd的另一大优点为**一站式购物**（one-stop shopping）。UnrealEd的通用浏览器（generic browser）能让开发者存取引擎支持的一切资源（图6.1）。以单一、整合、一致的界面创建和管理所有类型的资源，是UnrealEd的一大优势。对比其他大部分引擎，资源数据往往由无数个不一致、部分晦涩难懂的工具分散管理，UnrealEd的此设计特色更显优越。仅仅是可以在UnrealEd中**寻找**任何资源这一点，已是巨大优点。

虚幻比其他引擎较难出岔子，因为资产必须明确地导入虚幻的资源数据库。那么在制作初期已经可以检查资源的有效性。而其他大部分游戏引擎中，任何旧数据都可以放进资源数据库里，直到最后在生成时才会知道有没有数据失效，甚至有时在游戏载入数据时才发现问题。对虚幻来说，资产导入UnrealEd时就能检查其有效性。这意味着，建立资产者能获得即时反馈，得知其资产是否配置正确。

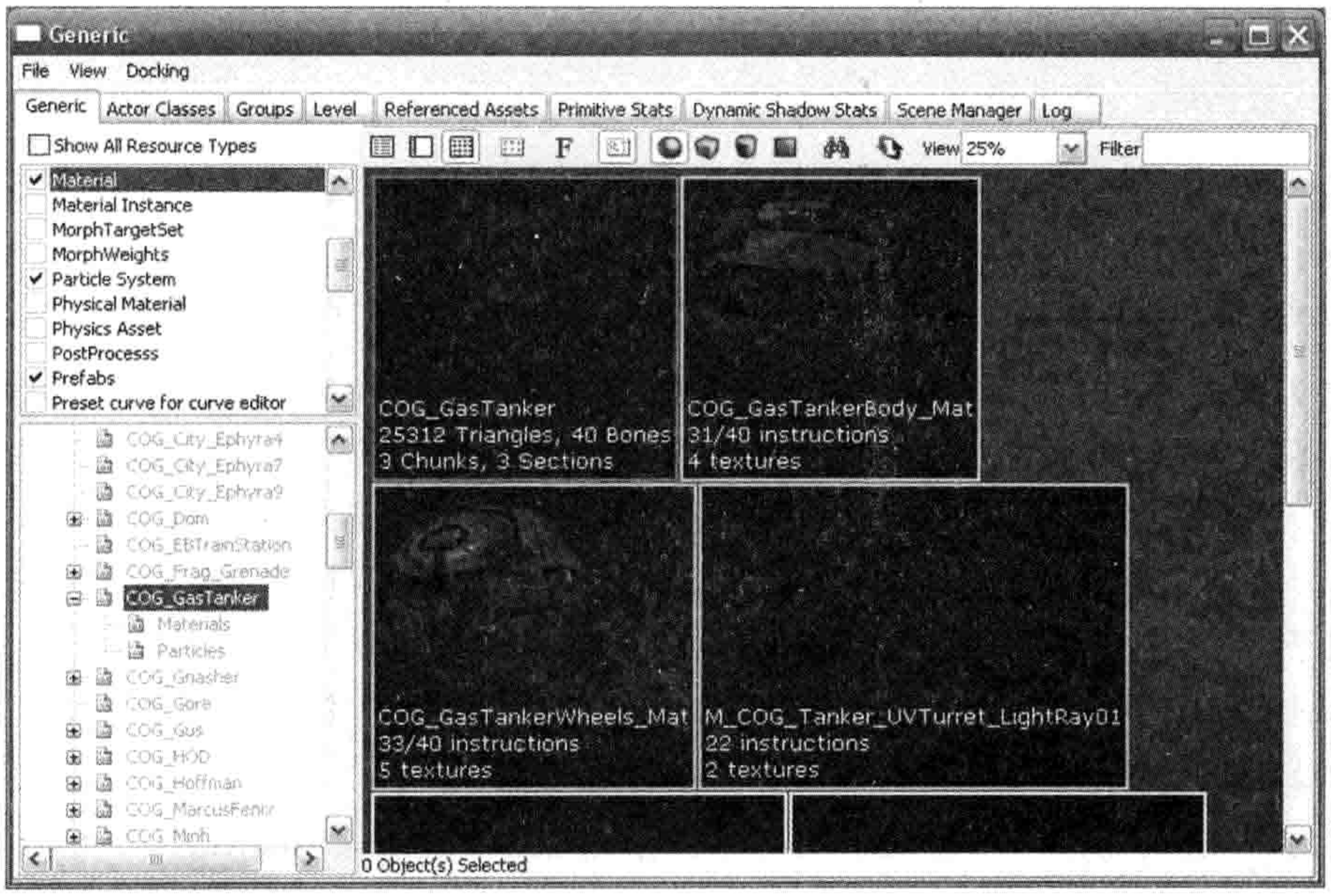


图 6.1: UnrealEd的通用浏览器。

当然，虚幻的方式也有一些重要缺点。首先，所有资源数据存于少量的大型包文件（package file）。这些文件是二进制的，因此并不易利用如CVS、Subversion或Perforce等版本控制包进行合并。当多位用户希望修改某包文件中的资源时，不能合并就会是重要问题。就算用户们尝试修改**不同的**资源，同一时间只有一位用户能锁定该包，因此其他人须轮候。舒缓此问题的方法之一就是，把资源划分为较小的包，然而实际上并不能完全根治。

UnrealEd的引用完整性相当不错，但仍有些问题。当某资源重新命名或移动后，所有对该资源的引用会自动维护，方法是产生一个虚拟对象（dummy object），把旧的资源映射至其新名称/位置。问题是这些虚拟映射对象会闲置、累积起来并造成问题，尤其是删除资源的时候问题变得严重。总体上，UnrealEd的引用完整性相当不错，尽管未臻完美。

撇除这些问题，UnrealEd是笔者用过最用户友好、整合良好、一条龙的资产创建工具包、资源数据库及资产调节管道。

顽皮狗的《神秘海域：德雷克船长的宝藏》引擎

在《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》中，顽皮狗储存其资源元数据至MySQL数据库，并编写了自制的图形用户界面（graphical user interface，GUI）管理数据库的内容。此工具给美术人员、游戏设计师和程序员等使用，以创建、删

除、查看及修改资源。此GUI是整个系统的关键组件，避免用户学习错综复杂的SQL语言去操作关联式数据库。

《神秘海域》最初的MySQL数据库并没有提供有用的版本历史功能，也没有方法去回滚（rollback）“坏”的改动。该数据库也不支持多人同时修改同一个资源，并难以管理。顽皮狗于是舍弃MySQL方案，改用由Perforce管理、基于XML文件的资产数据库。

图6.2为顽皮狗的资源数据库GUI，名为Builder。Builder的视窗分为两个主要部分：左方为树视图，显示游戏中的所有资源；右方为属性编辑视窗，用于显示及修改已选的一个或多个资源。资源树含文件夹，形成层次结构，方便美术人员及游戏设计师按自己喜欢的方式组织资源。任何文件夹都能创建及修改多种资源类型，包括演员（actor）[7]和关卡，以及组成这些的子资源（主要是网格、骨骼和动画）。动画可以组成伪文件夹，这种伪文件夹称为动画包（buddle）。那么，就能建立一大组动画，并且以这种单位进行管理，避免费时地在树视图中逐个动画拖动。

《神秘海域》的资产调节管道含有一组资源导出器、编译器、链接器，这些工具都是在命令行执行的。引擎能处理很多种类的数据对象，但数据对象会打包成为演员或关卡这两种资源文件。演员可以含有骨骼、网格、材质、纹理、动画。关卡则含有静态背景网格、材质、纹理，以及关卡布局信息。生成演员时，只需在命令行输入`ba`〈演员名字〉；生成关卡则输入`bl`〈关卡名字〉。这些命令行工具查询数据库以决定**如何**生成某演员或关卡。查询内容包括如何从数字内容创作（digital content creation，DCC）工具如Maya、Photoshop等导出数据，如何处理数据，以及如何把数据打包为二进制.pak文件供游戏引擎载入。这比许多游戏引擎简单得多，因为一般的引擎都要求美术人员**手工导出**资源，此乃费时费事、乏味、易错的任务。

顽皮狗设计的这个资源管道有以下优点。

- **粒度小的资源：**资源以逻辑实体的形式进行管理，这些逻辑实体是指网格、材质、骨骼、动画。这种资源粒度足够细小，使团队几乎不会出现两位成员想同时修改同一资源的冲突情况。
- **必需的特性（并无更多）：**Builder工具提供强大的特性，满足团队需求，而顽皮狗没有耗费任何资源去开发不需要的特性。
- **显而易见的源文件映射：**用户很容易得知某资源由哪些资产而来（原生DCC文件，如Maya .ma文件、Photoshop .psd文件）。
- **容易更改DCC数据的导出及处理方式：**只需在资源数据库GUI中单击相关资源，更改其处理属性便可。

[7]译注：在游戏业界中，actor通常是指游戏中含行为的动态对象，例如人物角色、载具、可动的门窗、开关等。

- **容易生成资产：**只需在命令行输入ba或bl，再加上资源名称。依赖系统（dependency system）便会处理余下事情。

当然，《神秘海域》的工具也有其缺点，包括如下几点。

- **欠缺可视化工具：**要预览资产，唯一方法是把资产载入游戏或模型/动画监示器（后者只是游戏的一种特别模式）。
- **工具没有完全整合：**顽皮狗使用一个工具为关卡布局，另一工具设置材质和着色器（此部分并非属于资源数据库GUI）。生成资产需利用命令行。若所有这些功能都整合至单一工具，应该比较方便。然而，顽皮狗没计划这么做，因为其效益很可能低于所需的成本。

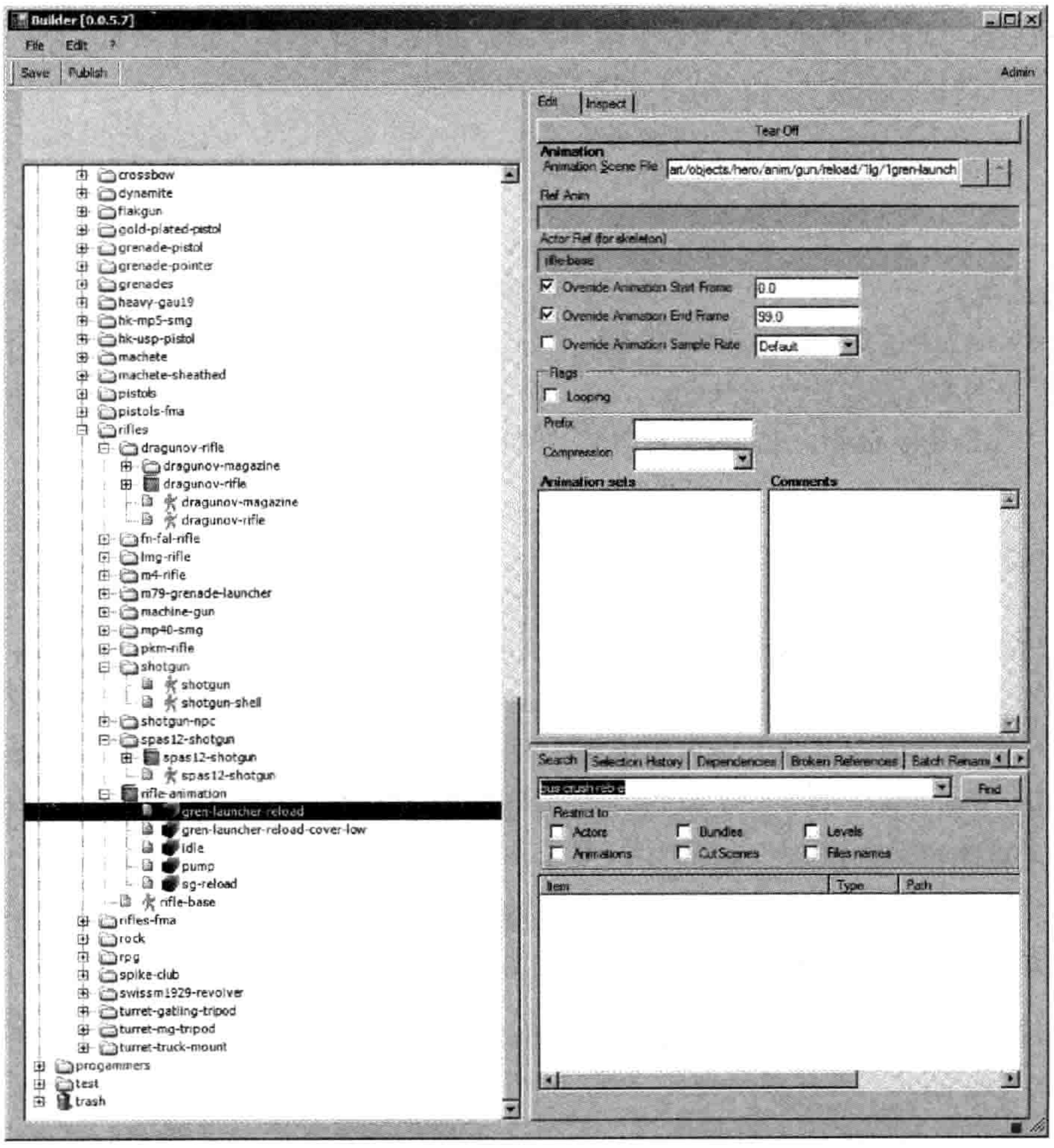


图 6.2: 顽皮狗的离线资源数据库前端Builder的图形用户界面。

OGRE的资源管理系统

OGRE是渲染引擎而非完整的游戏引擎。然而，OGRE却拥有一个颇完备、设计非常好的运行时资源管理器。通过一组简单一致的接口就能载入任何类型的资源。而且设计此系统时预留了扩展性（extensibility）。任何程序员都能在OGRE的资源框架下，为新的资产种类实现资源管理器，并将其轻松整合至框架中。

OGRE资源管理器的一个缺点在于，它仅是运行时方案。OGRE缺乏任何形式的离线数据库。OGRE只提供导出器（exporter），让Maya文件换为OGRE支持的网格格式（附有材质、着色器、骨骼，并可选择导出动画）。可是，导出器必须以人手方式在Maya里操作。更糟的是，描述某Maya文件怎样导出及处理的所有元数据，皆必须由用户每次导出时输入。

总括而言，OGRE的运行时资源管理器是强大且设计精良的。但若能加入同样强大的现代资源数据库及资产调节管道，就能令OGRE更完善。

微软的XNA

XNA是微软的游戏开发工具套件，以PC和Xbox 360为目标平台。XNA的资源管理系统别树一格，它通过Visual Studio IDE的项目管理及生成系统，把游戏资产以同样形式管理及生成。XNA的游戏开发工具Game Studio Express，其实只不过是Visual Studio速成版的插件。关于Game Studio Express可参阅MSDN[8]。

6.2.1.4 资产调节管道

在1.7节我们学到，资源数据通常是由先进的数字内容创作（digital content creation，DCC）工具所制作的，例如，Maya、ZBrush、Photoshop、Houdini等。然而，这些工具的数据格式一般并不适合游戏引擎直接使用。因此多数资源数据会经由**资产调节管道**（asset conditioning pipeline，ACP）才成为游戏引擎所用的数据。ACP有时也称为**资源调节管道**（resource conditioning pipeline，RCP）或**工具链**（tool chain）。

每个资源管道的开端都是DCC原生格式的源文件（如Maya .ma或.mb文件、Photoshop .psd文件等）。这些资产通常会经过3个处理阶段，才到达游戏引擎。

1. **导出器**（exporter）：为了要把DCC的源生格式转换为我们能够处理的格式，通常解决办法是为DCC工具撰写自定义插件，其功能就是把DCC工具里的数据导出为某种中间格式，供管道后续阶段使用。多数DCC应用软件都会为此提供尚算方便的机制。

[8] http://mdsn.microsoft.com/en-us/library/bb203894.aspx

实际上Maya提供3个机制：C++ SDK、名为MEL的脚本语言及近期新增的Python接口。

若然遇到某DCC软件不提供任何自定义方法，那总可以把数据储存为DCC工具的原生格式。幸运的话，其中可能有开放格式、尚算直观的文本格式，或其他可做反向工程的格式。若然如此，就可以把文件直接传送到管道里下一个阶段。

2. **资源编译器**（resource compiler）：我们通常要为由DCC工具导出的数据，以不同方式做一点“按摩”，才能让引擎使用。例如，可能要把网格的三角形重新排列成三角形带（triangle strip），或是要压缩纹理，或是要计算Catmull-Rom样条（spline）中每段的孤长。并非所有数据都需要编译，有些数据在导出后可能已经能直接供引擎使用。

3. **资源链接器**（resource linker）：有时候，多个资源需要先结合成为单个有用的包，然后才载入至游戏引擎。这个过程类似把C++源文件产生的对象文件，链接成为可执行文件。因此这个过程有时候称为**资源链接**。例如，要生成复杂的合成资源，如三维模型，可能会把多个导出的网格文件、多个材质文件、一个骨骼文件、多个动画文件内的所有数据，结合成为单一的资源。并非所有资源都需要链接，有些资产在导出或编译步骤之后已能供游戏使用。

资源依赖关系及生成规则

很类似C/C++项目中编译源文件再链接成可执行文件的过程，ACP会处理源资产（以Maya几何数据及动画文件、Photoshop PSD文件、原始音频片段、文本文件等形式），将其转换为游戏可使用的形式，再链接为内聚的整体供引擎使用。如同程序的源文件，各资产之间也有互相依赖的关系。（例如，某网格可能引用一个或多个材质，这些材质又引用多个纹理。）这些依赖关系通常会影响资产在管道内的处理次序。（例如，可能需要先生成角色的骨骼，才能处理该角色的任何一个动画。）除此以外，资产间的依赖关系也可告诉我们，当某个源资产做出改动后，要重新生成哪些资产。

生成依赖不单围绕资产本身的改动，也关系到数据格式的改动。例如，若储存三角形网格的文件格式改变了，那么整个游戏中的所有网格就要重新导出并重新生成。有些游戏引擎使用的数据格式，能强健地应付版本改动。例如，资产可能含版本编号，游戏引擎可包含一些代码以载入及使用遗留资产（legacy asset）。此方针的坏处在于，资源文件和代码都会趋于臃肿。若数据格式改动相对较少，当改动数据格式时，最好还是硬着头皮重新处理所有文件[9]。

[9]译注：若引擎有自定义工具创作游戏数据（如地图编辑器），由于这些文件是ACP的源文件，其自定义格式最好加入版本信息，以免改动格式时丢失用户的数据。

每个ACP都需要一组规则来描述资产间的依赖关系。当某资产做出改动后，有些生成工具可以利用这些依赖关系信息，确保以正确次序为所需的资产进行生成。一些游戏团队自己组建这类系统，另一些团队使用悠久的工具，例如make。无论选择哪个工具，团队都应该小心翼翼地看管他们的生成依赖系统。不然，某源文件的变动可能不会适当地触发重新生成某些资产。结果会造成不一致的游戏资产，导致游戏外观异常，甚至造成引擎崩溃。按笔者个人经验，曾目击团队为找出资产问题耗费无数小时，其实只要通过正确设置资产依赖关系，并实现可靠的生成系统，这些问题实际上是能避免的。

6.2.2 运行时资源管理

接着我们关注有关引擎运行时，资产怎样从资源数据库载入、管理并卸载。

6.2.2.1 运行时资源管理器的责任

游戏引擎的运行时资源管理器承担许多责任，全部都和其主要功能——载入资源至内存——有关。

- 确保任何时候，同一个资源在内存中只有**一份副本**。
- 管理每个资源的**生命期**（lifetime），载入需要的资源，并在不需要的时候卸载。
- 处理**复合资源**（composite resource）的载入。复合资源是由多个资源组成的资源。例如，**三维模型**是复合资源，含有网格、一个或多个材质、一个或多个纹理，并可能有骨骼和多个骨骼动画。
- 维护**引用完整性**。这包括**内部**引用完整性（单个资源内的交叉引用）及**外部**引用完整性（资源间的交叉引用）。例如，一个模型引用其网格和骨骼；网格引用其材质，材质又引用纹理；动画则引用骨骼，而最终骨骼又会绑定到一个或多个模型。当载入复合资源时，资源管理员必须确保所有子资源也被载入，并正确地修补所有交叉引用。
- 管理资源载入后的**内存用量**，确保资源储存在内存中合适的地方。
- 容许按资源类型，载入资源后执行**自定义的处理**。这种处理有时候又称为资源**登录**（log in）或资源**载入初始化**（load-initializing）。
- 通常（但非总是）提供单一**统一接口**管理多种资源类型。理想地，资源管理器应该要容易扩展，以便游戏开发团队需要新种类的资源时，也可以扩展处理。
- 若引擎支持，则要处理**串流**（streaming）（即异步资源载入）。

6.2.2.2 资源文件及目录组织

在一些游戏引擎（通常是PC上的引擎）中，每个资源储存为磁盘上的独立文件。这些文件通常位于树状目录中，而目录的组织主要是由资产创作者为方便而设计的。游戏引擎通常不会理会资源被放置于资源树中的哪个位置。以下是一个虚构游戏《太空逃亡者（Space Evaders）》[10]的典型资源目录树。

```
SpaceEvaders        整个游戏的根目录
  Resources         所有资源的根目录
    Characters      非玩家角色的模型和动画
      Pirate        海盗的模型及动画
      Marine        水兵的模型及动画
      ...
    Player          玩家角色的模型和动画
    Weapons         武器的模型和动画
      Pistol        手枪的模型和动画
      Rifle         步枪的模型及动画
      BFG           他X的大枪[11]的模型及动画
      ...
    Levels          背景几何及关卡布局
      Level1        第一关的资源
      Level2        第二关的资源
      ...
    Objects         其他三维物体
      Crate         无处不在的可破坏箱子
      Barrel        无处不在的可爆炸木桶
```

其他引擎会把多个资源包裹为单一文件，如ZIP存档（archive）或其他复合文件（也许是自定义格式）。这个手法的优点是减少载入时间。从文件载入数据时，三大开销为**寻道时间**（seek time）（即是把磁头移动至物理媒体上正确位置的时间）、开启每个文件的时间及从文件读入数据至内存的时间。三项之中，寻道时间和开启文件时间在许多操作系统里并非是微不足道的。使用单一大文件，这些开销都能减至最低。单一文件在盘上可以是连续的形式，这样寻道时间便能降至最低。而仅开启一个文件，也能消除为每个文件开启的开销。

---

[10] 译注：此处应该是对经典游戏《太空侵略者（Space Invaders）》的致敬。

[11] 译注：BFG 9000是《毁灭战士》中的武器，此处原文为“big ... uh ... gun”，读者意会或参考维基吧。

OGRE渲染引擎的资源管理器同时支持两种模式，可把资源文件各自置于硬盘上，也可以把资源置于庞大的ZIP存档中。使用ZIP格式的好处有：

1. **ZIP是开放格式**。用来读 / 写ZIP压缩文件的zlib和zziplib程序库都可供免费使用。zlib SDK是完全免费的[12]，而zziplib SDK则以LGPL授权[13]。
2. **ZIP存档内的虚拟文件也有相对路径**。换句话说，ZIP压缩文件蓄意设计成“像”文件系统的样子。OGRE资源管理员以貌似文件系统路径的字符串去识别所有资源。然而，这些路径有时是用来识别ZIP存档里的虚拟文件的，而非硬盘上的普通文件，使程序员在大多数情况下无须理会两者区别。
3. **ZIP存档可被压缩**。这样可缩小资源占用盘上的空间。但更重要的是，这么做可加速载入时间，因为从硬盘上载入的数据量减少了。当要从DVD-ROM或蓝光光盘读取数据时，就更见其效，因为这类设备的传输性能比硬盘慢得多。因此，载入数据后再解压所花的时间，通常比读取原来无压缩数据所花的时间少。
4. **ZIP存档可视为模块**。多个资源能组成ZIP文件，并以这些文件作为资源管理的单位。此想法可优雅地应用于产品本地化工作。所有需要本地化的资产（如含对话的音频片段、含文字或区域相关符号的纹理）可置于单一ZIP文件中，并为不同语言或地区制作该ZIP文件的不同版本。为某地区执行游戏时，引擎只要载入对应版本的ZIP文件。

虚幻引擎3采取类似的手法，但也有几个重要区别。虚幻中，所有资源都必须置于大型合成文件之中，这些文件称为**包**（package，又称为“pak文件”），并不容许资源以盘上独立文件出现。包文件采用自定义格式。虚幻引擎的编辑工具UnrealEd让开发者在这些包里创建及管理资源。

6.2.2.3 资源文件格式

每类资源都可能有不同的文件格式。例如，网格文件的储存格式通常异于纹理位图[14]。有些资产会储存为开放标准的格式。例如，纹理通常储存为TARGA[15]文件（TGA）、便携式网络图形（Portable Network Graphics，PNG）文件、标记图像文件格式（Tagged Image File Format，TIFF）文件、联合图像专家小组（Joint Photographic Experts Group，JPEG）文件，或视窗位图（Windows Bitmap，BMP）文件，也可储存为标准纹理压缩格

---

[12] http://www.zlib.net

[13] http://zziplib.sourceforge.net

[14] 译注：这例子通常是对的，但Hoppe在2002年发表了一篇文章《Geometry Images》，其重点刚好就是把网格储存为位图，能使用映像压缩及渐进式传输，可参阅http://research.microsoft.com/en-us/um/people/hoppe/proj/gim/。

[15] 译注：TARGA全称为Truevision Advanced Raster Graphics Adapter，而TGA为Truevision Graphics Adapter。此格式是20世纪90年代显卡厂商Truevision的图形格式，并非真正的开放标准。

式，如DirectX的S3纹理压缩家族格式（即S3TC，又称DXTn或DXTC）。同样，建模工具，如Maya或LightWave里的三维网格数据，也会导出为标准格式，如OBJ[16]或COLLADA，以供引擎使用。

有时候，单一文件格式可储存多种不同类型的资产。例如，Rad Game Tools公司的**Granny** SDK[17]实现了一个弹性开放式文件格式，此格式能储存三维网格数据、骨骼层次结构及骨骼动画数据。（实际上，Granny文件格式可以轻易用来储存任何种类的数据。）

许多游戏引擎程序员会定义自设的文件格式，当中有几个原因。其一，引擎所需的部分信息可能没有标准格式可以储存。此外，许多游戏引擎会尽力对资源做脱机处理，借以降低在运行时载入及处理资源数据的时间。数据须遵从某内存布局，例如，可选择原始二进制格式，在脱机时利用工具进行数据布局，而非载入数据时才做转换。

6.2.2.4 资源全局统一标识符

游戏中所有资源都必须有某种**全局唯一标识符**（globally unique identifier，GUID[18]）。最常见的GUID选项就是资源的文件系统路径（储存为字符串或其32位散列码）。这种GUID很直觉，因为它直接映射每个资源至硬盘上的物理文件。而且它能确保在整个游戏中是唯一的，因为操作系统已能保证两个文件不能有相同的路径。

然而，文件系统路径绝对不是资源GUID的唯一选择。有些引擎使用较不直觉的GUID类型，例如128位散列码，可能会利用工具来保证其唯一性。另一些引擎中，以文件系统路径作为GUID类型是不可行的。例如，虚幻引擎3储存多个资源在一个大文件里（称为**包**），所以包文件的路径并不能唯一地识别每个资源。虚幻的解决方案是，每个包里的资源以文件夹层次结构组织起来，并给予包里的资源唯一的名字，如同文件系统路径。因此虚幻的资源GUID格式是由包名字和包内资源路径串接而成的。例如在《战争机器（Gears of War）》中，资源GUID `Locust_Boomer.PhysicalMaterials.LocustBommerLeather`是用来标识一个位于`Locust_Boomer`包里`PhysicalMaterials`路径下名为`LocustBommerLeather`的材质。

6.2.2.5 资源注册表

为了保证在任何时间，载入内存的每个资源只会有一份副本，大部分资源管理器都含某种形式的**资源注册表**（resource registry），记录已载入的资源。最简单的实现模式就是使用

[16]译注：这里是指Wavefront公司的三维几何文件格式，而非对象文件格式。

[17]`http://www.rad-gametools.com`

[18]译注：注意这里不是指128位的GUID/UUID，而只是一个广义的概念——在某软件内不重复的标识符。

字典（dictionary），即**键值对**（key-value pair）的集合。键为资源的唯一标识符，而值通常就是指向内存中资源的指针。

资源载入内存时，就会以其GUID为键，加进资源注册表字典。卸下资源时，就会删除其注册表记录。游戏请求某资源时，资源管理员会先用其GUID查找资源注册表字典。若能寻获，就直接传回资源的指针；否则，就会自动载入资源，或是返回失败码。

乍看之下，若不能从资源注册表找到请求的资源，最直觉的处理手法就是自动载入该资源。而事实上，有些游戏引擎也是如此。然而，此手法也有一些严重问题。载入资源是缓慢操作，因为这涉及对硬盘上的文件定位及开启，读取可能大量的数据至内存（也可能是从很慢的设备读取，如DVD-ROM驱动），并且有机会在资源数据载入后，执行其载入后初始化工作。若请求来自运行中的游戏过程，载入资源可能会对游戏帧率造成非常明显的影响，甚至是几秒的停顿。因此，引擎可采取以下两个取代手法。

1. 在游戏进行中，完全禁止加载资源。此模式下，游戏关卡的所有资源在游戏进行前**全部**加载，那时候通常玩家正在观看载入画面或某种载入进度栏。
2. 资源以**异步**形式加载（即数据采用**串流**）。此模式下，当玩家在玩关卡A时，关卡B的资源就会在背景加载。此方式更可取，因为玩家能享受无载入画面的游戏体验。然而，这是相对较难实现的。

6.2.2.6 资源生命期

资源的**生命期**（lifetime）定义为该资源载入内存后至内存被归还做其他用途之间的时段。资源管理器的职责之一就是管理资源生命期——可能是自动的，也可能是通过对游戏提供所需的API函数，供手动管理资源生命期。

每个资源对生命期各有不同需求。

- 有些资源在游戏开始时便必须载入，并驻留在内存直至整个游戏结束。换言之，其生命期实际上是无限的。这些资源有时候称为载入并驻留（load-and-stay-resident，LSR）资源。典型例子包括：玩家角色的网格、材质、纹理及核心动画，抬头显示器（HUD）的纹理及字型，整个游戏都会用到的所有常规武器的资源。在整个游戏过程中，玩家能一直听到或看到的任何资源（以及不能按需载入的资源），都应该归为LSR资源。
- 有些资源的生命期能对应某游戏关卡。在玩家首次看到关卡时，对应的资源便须留在内存，直至玩家永久地离开关卡，资源才能被弃置。
- 有些资源的生命期短于其所在关卡的时间。例如，游戏中过场动画（in-game cut-

scene）（推进剧情或向玩家提供重要信息的迷你电影）里使用到的动画及音频短片，可能在玩家观看过场动画之前载入，播放后就能弃置。

- 有些资源如背景音乐、环境音效（ambient sound effect）或全屏电影，可以在播放时即时串流。这类资源的生命期很难定义，因为每字节只短暂留在内存中，但整首音乐却会延续很长的时间。这类资源通常以特定大小的区块为单位载入，区块大小根据硬件需求而定。例如，音轨可能会以4KB区块读入，因为某低阶声音系统的缓冲区可能是4KB。内存中某一刻只需两区块的数据，分别是目前播放中的区块，及载入中、紧接前者的区块。

某资源需要在何时载入，通常不是难题，只要按玩家第一次看见该资源的时间便能决定。然而，何时卸下资源并归还内存，就不是容易回答的了。问题在于，许多资源会在多个关卡中共享。完成关卡A时，我们不希望卸下一些资源后，在关卡B又立即再重新加载相同的资源。

解决方案之一就是对资源使用引用计数。首先，载入新关卡时，遍历该关卡所需的资源，并把这些资源的引用计数加1（那时候这些资源还未载入）。然后，遍历即将要卸下的（一个或多个）关卡里的所有资源，把这些资源的引用计数减1。那么，引用计数跌至0的资源就可被卸下了。最后，再把引用计数刚刚由0变成1的资源载入内存。

例如，假设关卡1使用资源A、B、C，而关卡2使用资源B、C、D、E（两关卡共享B和C）。表6.2列出玩家从关卡1玩至关卡2时，这5个资源的引用计数变化。表中加粗的引用参数代表该资源已载入内存，灰色背景代表资源不在内存。有括号的引用计数代表资源正要载入或卸下。

事件	A	B	C	D	E
起始状态	0	0	0	0	0
关卡1的引用计数加1	1	1	1	0	0
载入关卡1	(1)	(1)	(1)	0	0
玩关卡1	**1**	**1**	**1**	0	0
关卡2的引用计数加1	**1**	**2**	**2**	1	1
关卡1的引用计数减1	**0**	**1**	**1**	1	1
卸下关卡1，载入关卡2	(0)	**1**	**1**	(1)	(1)
玩关卡2	0	**1**	**1**	**1**	**1**

表 6.2: 当载入/卸下两个关卡时，资源的引用变化。

6.2.2.7 资源所需的内存管理

资源管理和内存管理息息相关，因为载入资源时，不可避免要决定资源加载至**哪个地方**。每个资源加载的目的地可能不同。首先，某些资源必须驻留在显存（video RAM）。典型例子包括纹理、顶点缓冲（vertex buffer）、索引缓冲（index buffer）、着色器。大部分其他资源可能都会驻留在主内存（main RAM），但不同的资源可能须置于不同的地址范围。例如，载入并驻留资源（LSR）可能会载入某内存区域，而经常载入卸下的资源可能会置于其他地方。

游戏引擎中，内存分配子系统的设计，通常与资源管理的设计有密切关系。有时候，我们会尽量运用已有的内存分配器设计资源系统；或倒过来，我们可以设计内存分配器，以配合资源管理所需。

5.2.1.4节中我们已提及资源管理会遇到的主要难题在于，载入 / 卸下资源时避免形成内存碎片。针对此问题，以下会探讨一些常用的解决方案。

基于堆的资源分配

处理方法之一是简单地忽略内存碎片问题，仅使用通用的堆分配器分配资源所需的内存（如使用C的`malloc()`或C++的全局`new`运算符）。若你的游戏只需运行在个人计算机上，这个方法还算可以，因为操作系统支持高级的虚拟内存分配。这种系统里，物理内存会裂成碎片，但操作系统有能力把不连续的物理内存页映射为连续的虚拟内存空间，有助于缓和内存碎片引起的不良影响。

若你的游戏要运行于物理内存有限的游戏机上，只配上最基础的虚拟内存管理器（甚至这都没有），那么内存碎片就会是个问题。此情况下，另一方案是定期整理内存碎片，这已于5.2.2.2节介绍如何实现。

基于堆栈的资源分配

堆栈分配器并不会有内存碎片问题，因为内存是连续分配的，而释放内存时则以分配的反方向进行。若以下两个条件成立，堆栈分配器便能用于载入资源。

- 游戏是线性及以关卡为中心的（即玩家观看载入画面，之后玩一个关卡，再观看另一个载入画面，之后再玩另一关卡）。
- 内存足够容纳各个完整关卡。

假设这些条件皆成立，便可使用堆分配器载入资源，详情如下。游戏启动时，先分配给载入

并驻留资源（LSR）。标记栈的顶端位置，那么之后便可以释放资源至此位置。载入关卡时，只需要简单地在栈的顶端分配资源所需的内存。玩家完成关卡后，就可以把栈的顶端位置移到之前标记的位置，那么就可以迅速释放关卡的所有资源，仅留下LSR资源。此过程能对无数关卡不断重复，而永不会导致内存碎片。图6.3说明了这个过程。

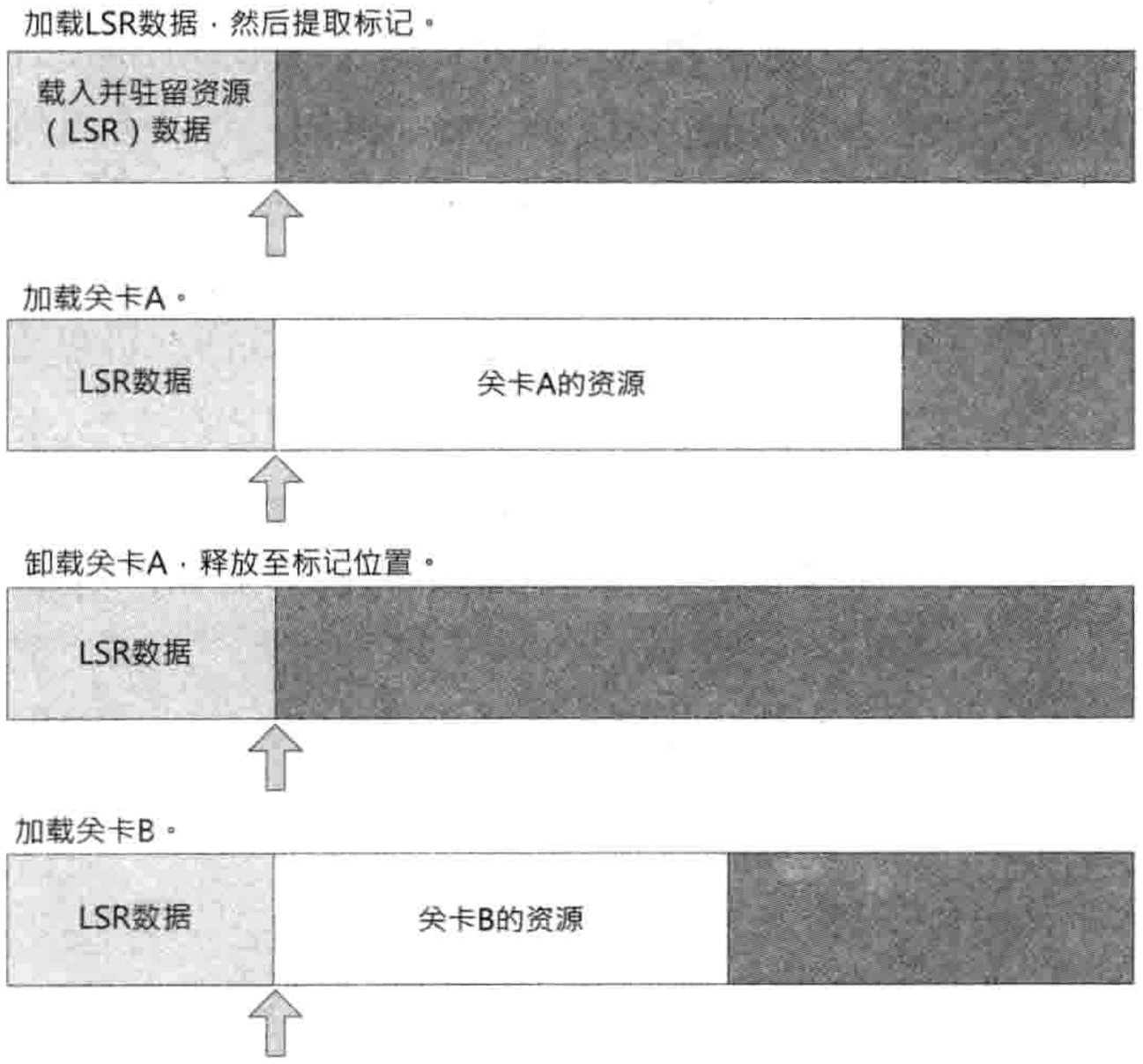


图 6.3: 使用堆栈分配器载入资源。

双端堆栈分配器也可用于此手法。两个栈定义在一大块内存里。其中一个由内存的底端往上成长，另一个则由顶端往下成长。只要两个栈永不重叠，它们就能自然地共享内存资源——若每个堆栈都各自有固定的大小，这种协调便不可能。

在《迅雷赛艇》中，游戏开发商Midway采用了双端堆栈分配器。底端堆栈用来载入持久的数据，而顶堆端栈则用作每帧的临时分配内存，每帧结束后就会释放整个顶端栈。另一种双端堆栈分配器的用法是来回（ping-pong）地载入关卡。Bionic Games曾在某项目上使用这种方式。其基本思路是，载入关卡B的压缩后版本至顶端堆栈，而当前关卡A（无压缩版本）则驻于底端堆栈。由关卡A进入关卡B时，就简单地释放关卡A的资源（实际上就是清除底端堆栈），之后就把关卡B从顶端堆栈解压至底端堆栈。解压缩通常比从硬盘读入数据快得多，因此这一方法排除了载入时间，使玩家过关时更感顺畅。

基于池的资源分配

在支持串流的游戏引擎中，另一个常见资源分配技巧是，把资源数据以同等大小的组块（chunk）载入。因为全部组块的大小相同，所以可以使用**池分配器**（见5.2.1.2节）。之后资源卸下时就不会造成内存碎片。

当然，基于组块的分配方式，需要所有资源以某方式布局，以容许资源能被切割成同等大小的组块。我们不能简单地把随意资源以组块方式载入，因为那些文件里可能含有连续的数据结构，例如数组或大于单个组块的巨形`struct`。例如，若多个组块含有一个数组，而组块又不是在内存中顺序排列的，那么数组的连续性就会消失，不能正常地用索引存取数组。这意味着，设计所有资源数据时都要考虑到“组块特性”。必须避免大型连续数据结构，取而代之，要使用小于单个组块的数据结构，或是不需要连续内存仍可正常运作的数据结构（如链表）。

池里的每个组块通常对应某个游戏关卡。（简单的实现方法就是给每个关卡一个组块链表。）那么，就算内存中同时有多个关卡，各有不同的生命期，引擎也可以适当地管理每个组块的生命期。例如，关卡A在载入后占用了N个组块。之后，又为关卡B另外分配了M个组块。当关卡A最后被卸下时，其N个组块就获释放。若关卡B仍在使用中，其M个组块就继续驻于内存。通过关联每个组块至特定关卡，就能简单有效地管理组块的生命期。图6.4展示了这种方式。

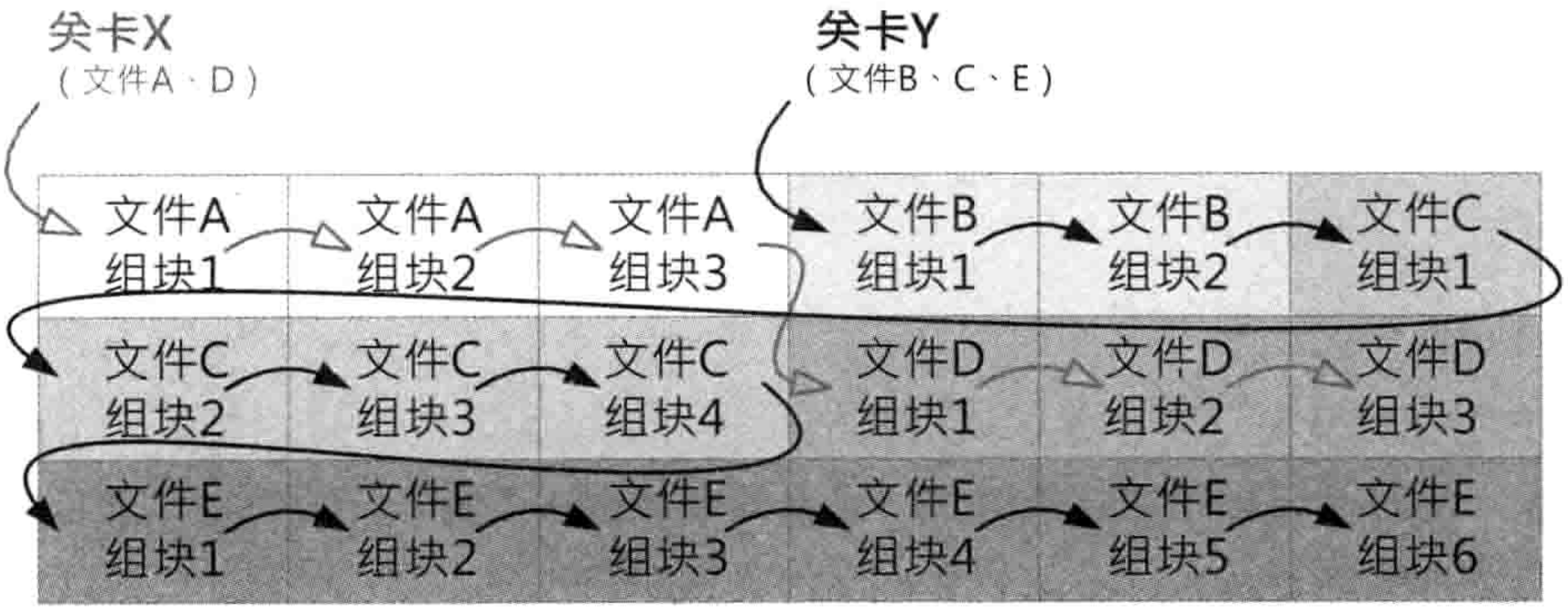


图 6.4: 关卡A和B的资源以组块方式分配。

“组块式”资源分配天生具有一个取舍问题，这就是空间浪费。除非资源文件大小刚好是组块大小的倍数，否则文件内最后的组块便不能充分利用所有空间（见图6.5）。选择较小的组块大小能舒缓问题，但组块越小，资源数据的布局限制就会变得更烦锁。（举一个极端例子，若组块大小选为1字节，那么所有数据结构都不能大于1字节，这显然是站不住脚的选择。）典型的组块大小大约是数千字节。例如，于顽皮狗，我们使用的组块式资源分配器，

是资源流系统的一部分，而组块大小则定为512KB。读者在选择组块大小时，可以考虑把其大小设为操作系统输入 / 输出缓冲区大小的倍数，借以期望在载入个别组块时能提供最大效能。

图 6.5: 我们通常不能充分利用资源文件的最后一个组块。

资源组块分配器

要限制因组块而导致浪费的内存，办法之一是设立特殊内存分配器，此分配器能利用组块内未用的内存。据笔者所知，这类分配器并无标准命名，笔者在此称它为**资源组块分配器**（resource chunk allocator）。

资源组块分配器并不难实现。只需管理一个链表，内含所有未用满内存的组块，每笔数据也包含自由内存块的位置及大小。然后就可以从这些自由内存块中，按需分配。例如，可以使用通用堆分配器管理这些自由内存块链表。或是可以为每个自由内存块设立小型栈分配器，面对内存分配请求时，就扫描每个栈分配器，遇到有足够内存空间的，就用该栈完成分配请求。

遗憾的是，此方案有美中不足的地方。若在资源组块未使用的区域分配内存，那么释放组块的时候又怎么办？不可能只释放组块的一部分，只能选择全部释放或不释放。因此，从那些未用区域分配来的内存，会在资源卸下时神奇地消失。

一个简单方案是，只利用资源组块分配器分配一些和对应关卡生命期相同的内存。换句话说，关卡A组块的自由内存只供属于关卡A的数据分配，关卡B组块的内存就只供关卡B的数据分配。这需要资源组块分配器独立地管理每个关卡的组块。用户请求分配时需要指明，要从哪个关卡分配内存才可以让分配器选择正确的链表来满足请求。

幸亏大部分游戏引擎在载入资源时都需要分配动态内存，内存需求可能大于那些资源文件本身。所以资源组块分配器可以成为有效的方法，重新利用组块原来浪费了的内存。

分段的资源文件

另一个和“组块式”资源相关的有用概念是**文件段**（file section）。典型的资源文件可能包含1～4段，每段分为一个或多个组块，以配合上述基于池的资源分配。某段可能含有为主内存而设的数据，而其他段则是视频内存的数据。另一段可能含有临时数据，仅于载入过程中使用，整个资源载入后这些临时数据就会被弃置。再另一段可能含有调试信息。游戏在调试模式下运行会载入这些调试数据，而在最终的发行版本则不会载入。Granny SDK的文件系统[19]是个优秀的例子，能说明如何把分段文件实现得又简单又有弹性。

6.2.2.8 复合资源及引用完整性

通常游戏的资源数据库包括多个**资源文件**，每个文件含有一个或多个**数据对象**（data object）。这些数据对象能用不同方式，引用或依赖于其他数据对象。例如，网格数据结构可能含引用，指向其材质；材质又含一组引用，指向多个纹理。通常交叉引用意味着依赖性（即是，若资源A引用资源B，则A和B必须同时在内存里才能使游戏正常运作）。总的来说，游戏资源数据库可表达为，由互相依赖的数据对象所组成的**有向图**（directed graph）。

数据对象之间的交叉引用可以是**内部的**（单个文件里两个对象的引用）或**外部的**（引用另一个文件的对象）。这种区别是重要的，因为内部和外部引用的实现方式通常各有不同。以下我们尝试把游戏的资源数据库视觉化，每个资源文件以虚线框表示，凸显内部/外部引用之分别——越过虚线文件边界的箭头就是外部引用，没有越过的是内部引用。如图6.6所示为该例。

有时候我们会把一组自给自足、由相互依赖资源所合成的资源称为**复合资源**（composite resource）。例如，**三维模型**是复合资源，内含一个至多个**三角形网格**、可选的**骨骼**、可选的**动画**集合。每个网格还对应一个**材质**，每个材质又引用一个或多个**纹理**。要完整地载入复合资源（如三维模型）至内存，也必须载入所有其依赖的资源。

6.2.2.9 处理资源间的交叉引用

实现资源管理的难点之一在于，管理处理资源对象间的交叉引用，并确保维系引用完整性。要理解资源管理器如何达成此需求，可以先看看交叉引用如何储存于内存和磁盘中。

在C++中，两数据对象间的交叉引用，通常以**指针**或**引用**实现。例如，网格可能含有数据成员`Material* m_pMaterial`（一个指针）或`Material& m_material`（一个引

[19] http://www.radgametools.com

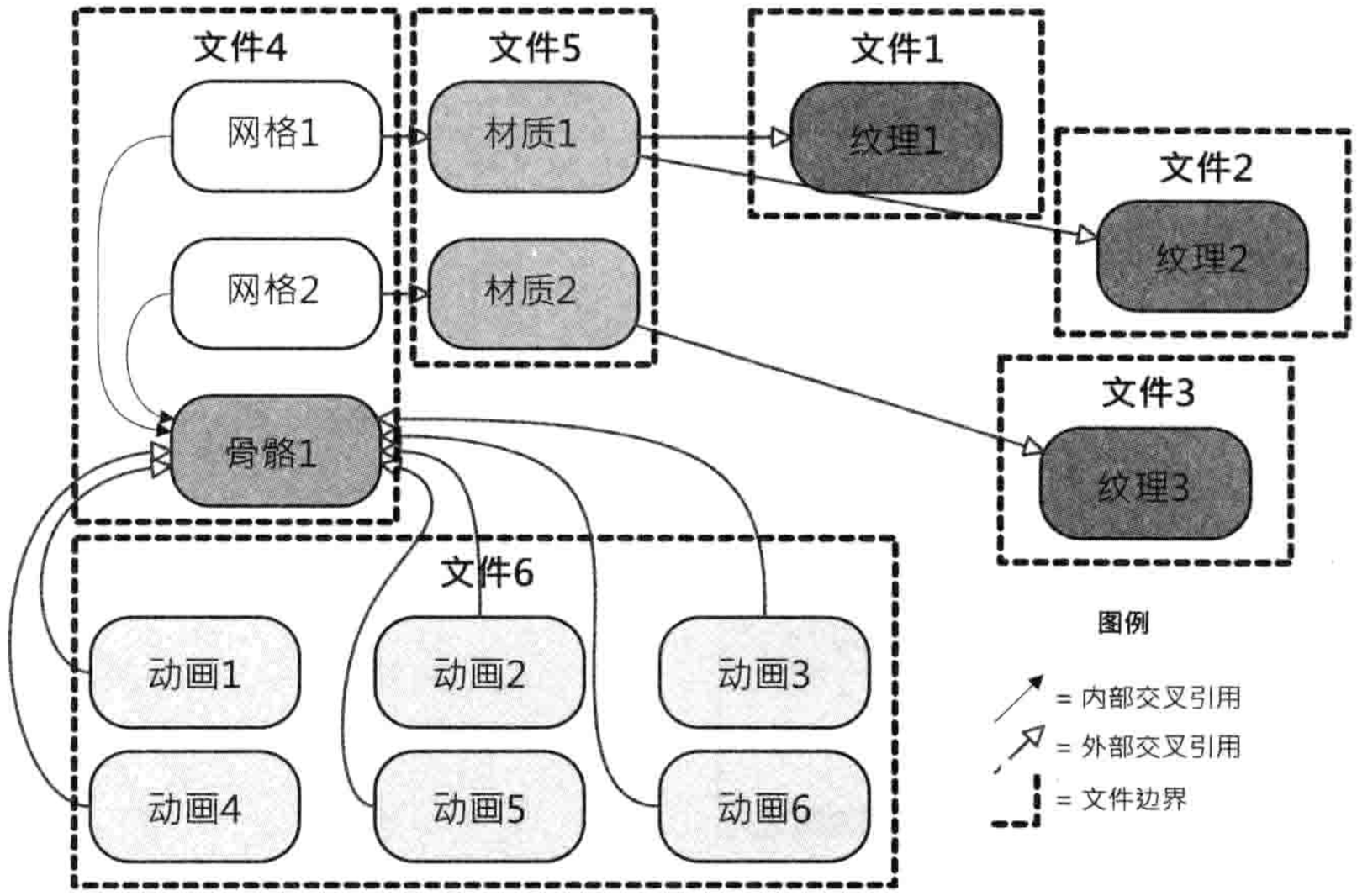


图 6.6: 资源数据库的依赖图例子。

用），用来引用其材质。然而，指针只是内存地址，其值在离开运行中的程序时就会失去意义。事实上，再多次运行相同程序，内存地址也会改变。显然，储存数据至文件时，不能使用指针表示对象之间的依赖性。

使用全局统一标识符做交叉引用

优良的解决方案之一，就是把交叉引用储存为字符串或散列码，内含被引用对象的唯一标识符。这意味着每个可能被引用的资源对象，都必须具有**全局唯一标识符**（GUID）。

这种交叉引用方式要行得通，资源管理器要维护一个全局资源查找表。每次载入资源对象至内存后，都要把其指针以GUID为键加进查找表中。当所有资源对象都载入内存后，就可以扫描所有对象一次，对其交叉引用的资源对象GUID，通过全局资源查找表换成指针。

指针修正表

储存对象至二进制文件的另一常用方法就是，把**指针**转换为**文件偏移值**（file offset）。假设有一组C struct或C++对象，它们之间利用指针做交叉引用。要储存这组对象至二进制文件，只需以任意次序访问每个对象一次（且仅一次），把每个对象的内存映像顺序写至文件。其效果就是把所有对象序列化（serialize）为文件中的**连续**映像，即使对象在内存中并非连续。参考图6.7。

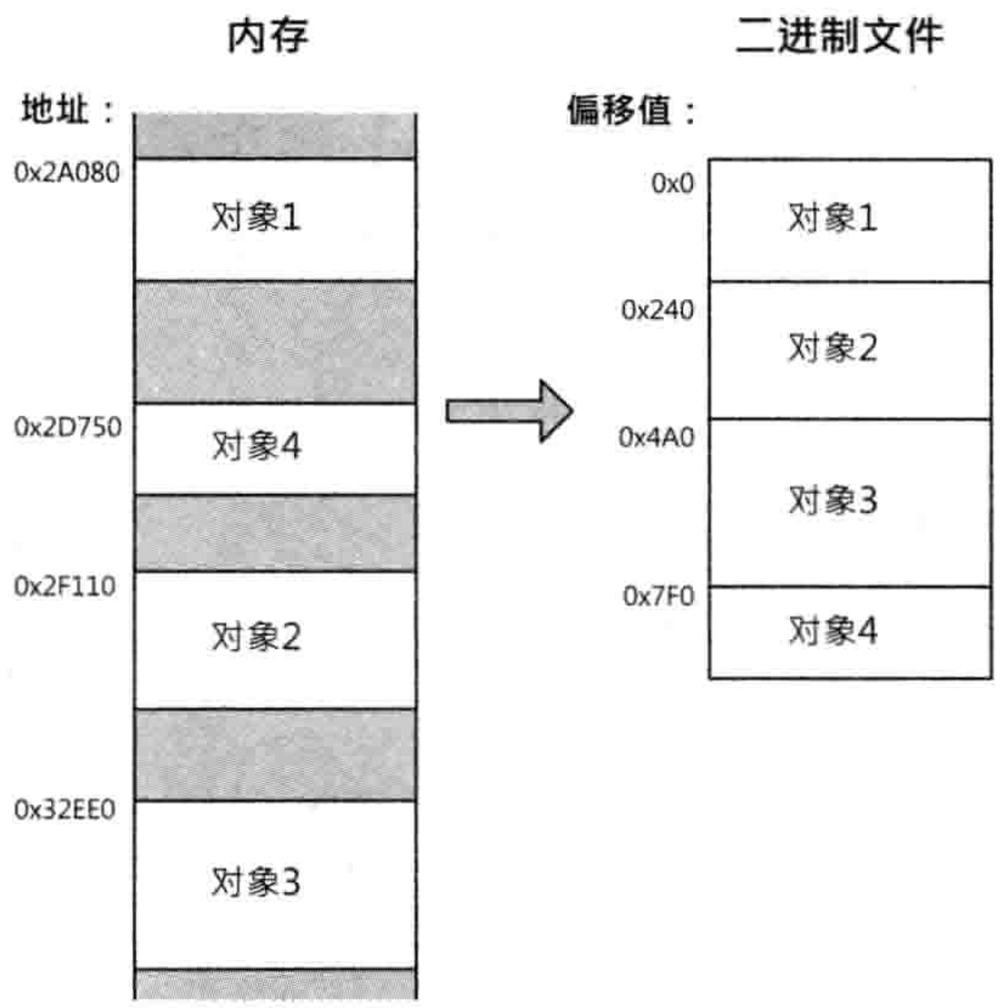


图 6.7: 当储存为二进制文件时，内存里的对象映像就能变成连续的。

对象原来的内存映像已经写进文件中，因而可以得知每个对象映像相对文件开始的**偏移值**。在写入二进制文件映像的过程中，要找出对象中所有的指针，并用偏移值原地取代那些指针。我们可以简单地写入偏移值，取代指针，是因为指针总有足够的位存放偏移值。实际上，二进制文件的**偏移值**等同内存的**指针**。（需要注意开发平台和目标平台的区别。若在64位Windows下写入文件，其指针是64位的，因此该文件不能和32位的游戏机兼容[20]。）

当然，之后载入文件至内存时，也需要把偏移值转回指针。这种转换称为**指针修正**（pointer fix-up）。当载入文件二进制映像时，映像内对象仍然保持连续。所以，把偏移值转回指针是易如反掌的。以下列出相关代码，并以图6.8说明。

```
U8* ConvertOffsetToPointer(U32 objectOffset,
                           U8* pAddressOfFileImage)
{
    U8* pObject = pAddressOfFileImage + objectOffset;
    return pObject;
}
```

偏移值和指针之间互相转换很简单，问题在于如何**找出**需要转换的指针。通常我们会在写二进制文件时解决此问题。负责把数据对象映像写进文件的代码，清楚知道对象的数据类型和类，因此这些代码也知道每个对象中的所有指针位于哪里。可以把指针的位置储存到一

[20]译注：若要支持游戏机，还要考虑之前提及的字节序、对齐等问题。

个简单列表，此表就是**指针修正表**（pointer fix-up table）。指针修正表连同对象映像一起写进二进制文件。之后，当载入文件至内存时，就能凭这个表修正所有指针。指针修正表的内容只是文件里的偏移值，每个偏移值代表一个需要修正的指针。图6.9解释了此机制。

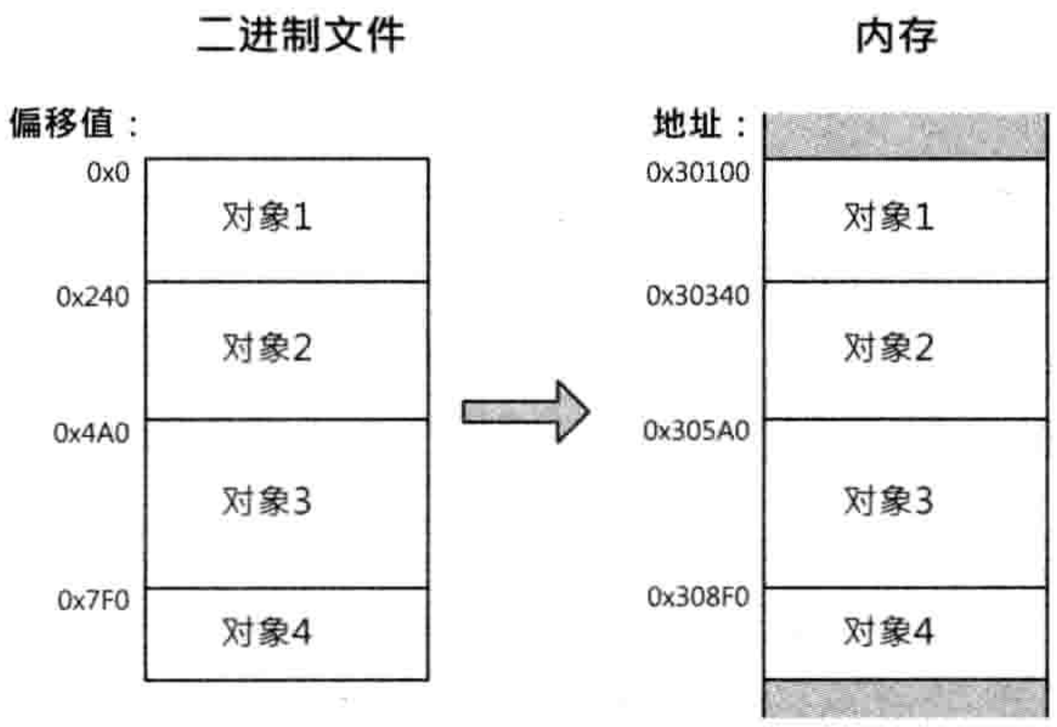


图 6.8: 当文件载入至内存后，资源文件映像仍然是连续的。

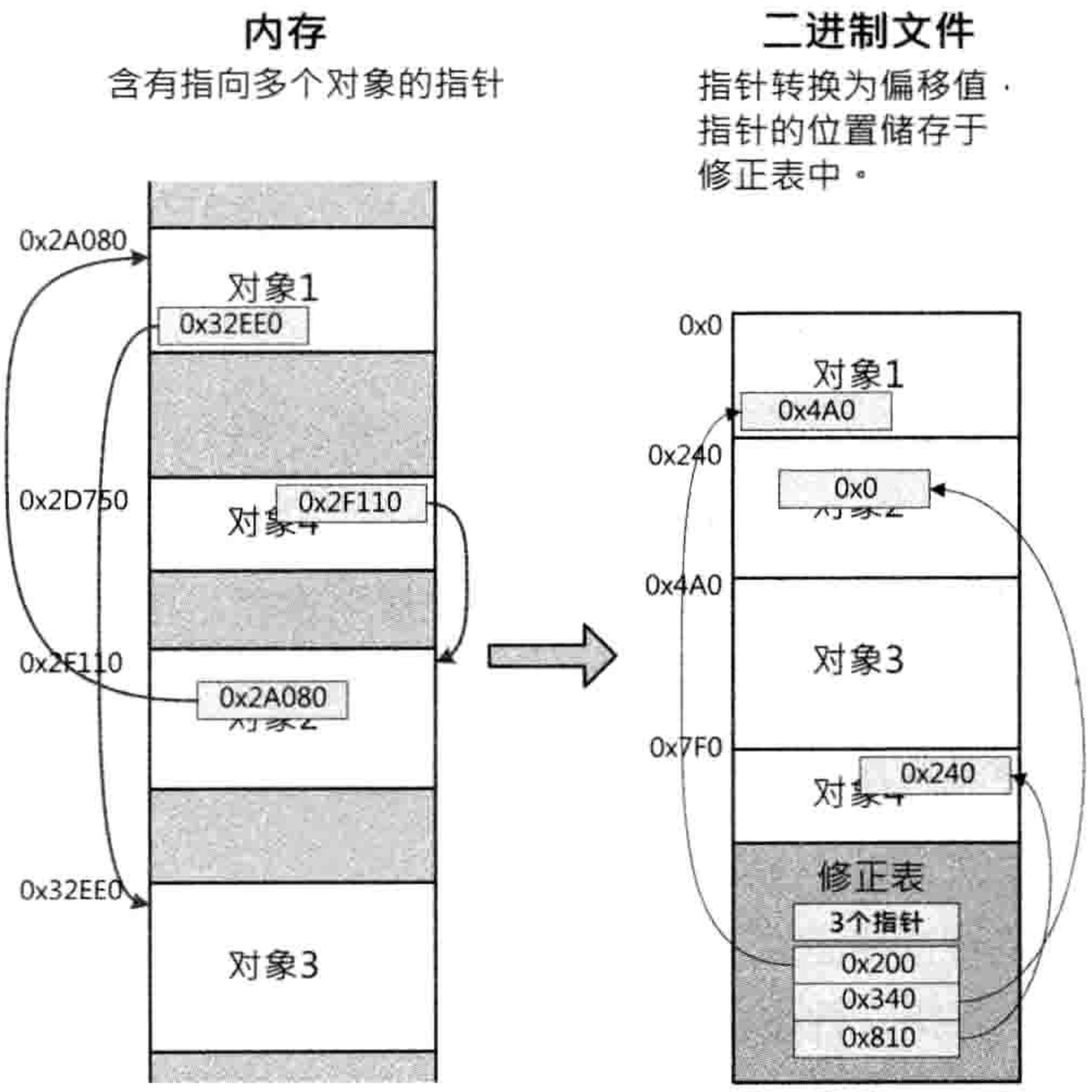


图 6.9: 指针修正表。

储存C++对象为二进制映像：构造函数

从文件载入C++对象，很容易忽略一个重要步骤——必须调用对象的构造函数。例如，若从某二进制文件载入3个对象，一个是A类的实体，一个是B类的实体，一个是C类的实体，那么就要对这3个对象分别调用正确的构造函数。

这个问题有两个常见解决方案。其一，读者可以简单决定，二进制文件完全不支持C++对象。换句话说，就是限制系统仅支持POD结构（plain old data structure，PODS）——C `struct`，以及**无虚函数、只含不做事情的平凡构造函数**（trivial constructor）的C++ `struct/class`。（关于POD结构的完整讨论，可参考维基百科[21]。）

其二，可以把非PODS对象的偏移值组成一个表，并在表里记录对象属于哪个类，最后把此表写进二进制文件。之后，载入二进制映像之时，就可以遍历此表，并对每个对象使用**placement new**语法调用适当的构造函数（即是对已分配的内存块调用构造函数）。例如，给定对象在二进制映像中的偏移值可以这样编码：

```
void* pObject = ConvertOffsetToPointer(objectOffset);
::new(pObject) ClassName; // placement-new 语法
```

当中**ClassName**是该对象所属的类。

处理外部引用

以上提及的两个方案，对引用**内部**资源非常有效。所谓内部资源，即是指单个资源文件内的对象。在最简单的情况下，可以载入二进制映像至内存，再用指针修正去解析所有交叉引用。但是，当交叉引用至其他资源文件时，就得采取稍有不同的手法。

要正确表示**外部**交叉引用，除了要指明偏移值或GUID，还需加上资源对象所属文件的路径。

载入由多个文件组成的资源，关键在于要先载入**所有**互相依赖的文件。可行的做法是，载入每个资源文件时，扫描文件中的交叉引用表，并载入所有被外部引用但未载入的资源文件。载入每个数据对象至内存时，可以把其地址加进主查找表。当载入所有互相依赖的资源文件后，所有对象已驻于内存，就可以使用主查找表把所有指针转换一遍，从GUID或文件偏移值转换为真实的内存地址。

---

[21] http://en.wikipedia.org/wiki/Plain_Old_Data_Structures

6.2.2.10 载入后初始化

理想地，每个资源都能经离线工具完全处理，载入后内存立即能够使用。实际而言，这并不经常可行。许多资源种类在载入后，至少需要一些“按摩”才能供引擎使用。本书中，笔者使用**载入后初始化**（post-load initialization）这个术语来描述资源数据载入后的任何处理。其他引擎或会使用其他术语。（例如在顽皮狗里，我们称之为**登入**资源。）许多资源管理器也支持释放资源的内存之前，执行某种拆除（tear-down）步骤。（在顽皮狗里，我们称之为**登出**资源。）

载入后初始化通常有两种情况。

- 某些情况下，载入后初始化是无法避免的步骤。例如，定义三维网格的顶点和索引值，载入主内存以后，几乎总是要传送至显存。这个步骤只能在运行时进行，过程包括建立Direct3D顶点或索引缓冲，锁定缓冲，复制或读入数据至缓冲，并解锁缓冲。
- 其他情况下，载入后初始化的处理过程是可避免的（即能把处理过程移至工具），但为了方便起见成为权宜之策。例如，程序员可能想在引擎的样条（spline）库中，加入精确的弧长计算。与其花时间更改工具并生成孤长数据，该程序员可能选择简单地在载入后初始化时才计算这些数据。之后，当计算结果完美了，才把代码搬到工具里，以避免运行期计算的开销[22]。

显然，每类资源的载入后初始化及拆除，都各有其独特的需求。因此，资源管理器通常可以按各资源类型，个别设置这两个步骤。在非面向对象语言中，例如C，可以使用查找表，把每资源类型映射至一对函数指针，一个负责载入后初始化，一个负责拆除。在面向对象语言中，例如C++，实现就更为简单了，只需使用多态为每个类独立处理其载入后的初始化和拆除。

在C++中，载入后初始化可以实现为一个特别的构造函数，而拆除则可置于类的析构函数中。然而，为此使用构造函数和析构函数会产生一些问题。（例如，C++的构造函数不能是虚函数，因此派生类并不能改变其基类的载入后初始化过程。）许多开发者比较喜欢把载入后初始化和拆除置于普通的虚函数中。例如，可选择一对虚函数，命名为`Init()`及`Destroy()`之类。

载入后初始化和资源内存分配策略息息相关，因为初始化时经常会产生新数据。某些情况下，载入后初始化步骤会在文件载入的数据上**新增**数据。（例如，读取Catmull-Rom样条之后，为每段计算孤长，那么就要分配额外的内存空间存放这些计算结果。）另一些情况是，载入后初始化步骤所产生的数据用来取代已载入的数据。（例如，为了向后兼容，可能会容

[22]译注：若是简单的计算，可能在运行时载入后计算，比读取数据的I/O时间还要快。应按实际情况决定在线或离线处理。

许引擎载入过时格式的网格数据，之后自动转换为最新格式。）此情况下，载入后产生新数据之后，部分或全部旧数据就会被弃置。

《迅雷赛艇》引擎有一简单强大的方法处理此事。该引擎容许两种载入资源方式：（a）直接载入至最终的内存位置，（b）载入至临时的内存区域。在（b）的情况下，载入后初始化须负责把处理后的数据复制至最终内存位置，之后位于临时内存的资源数据就会被弃置。这是十分有效的方法，载入同时含相关和不相关数据的资源。相关数据会复制至内存的最终目的地，不相关数据会被弃置。例如，过时格式的网格数据可以载入临时内存，在载入后初始化步骤中转换成最新格式，而不需要浪费内存保留旧格式数据。

第7章　游戏循环及实时模拟

游戏是实时的、动态的、互动的计算机模拟。由此可知，**时间**在电子游戏中担当非常重要的角色。游戏中有不同种类的时间——实时、游戏时间、动画的本地时间线、某函数实际消耗的CPU周期等。每个引擎系统中，定义及操作时间的方法各有所不同。我们必须透彻理解游戏中所有时间的使用方法。本章会谈及实时、动态模拟软件如何运作，并探讨这类模拟中运用时间的常见方法。

7.1　渲染循环

在图形用户界面（graphical user interface，GUI）中，例如Windows和Macintosh的机器上的GUI，画面上大部分的内存是静止不动的。在某一时刻，只有少部分的视窗会主动更新其外貌。因此，传统上绘画GUI界面会利用一个称为**矩形失效**（rectangle invalidation）的技术，仅让屏幕中有改动的内容重绘。较老的二维游戏也会采用相似的技术，尽量降低需重画的像素数目。

实时三维计算机图形以完全另一方式实现。当摄像机在三维场景中移动时，屏幕或视窗上的**一切内容**都会不断改变，因此再不能使用失效矩形法。取而代之，计算机图形采用和电影相同的方式产生运动的错觉和互动性——对观众快速连续地显示一连串静止影像。

要在屏幕上快速连续地显示一连串静止影像，显然需要一个循环。在实时渲染应用中，此循环又称为**渲染循环**（render loop）。渲染循环的最简单结构如下：

```
while (!quit)
{
    // 基于输入或预设的路径更新摄像机变换
    updateCamera();

    // 更新场景中所有动态元素的位置、定向及其他相关的视觉状态
    updateSceneElements();
```

```
    // 把静止的场景渲染至屏幕外的帧缓冲(称为“背景缓冲”)
    renderScene();

    // 交换背景缓冲和前景缓冲，令最近渲染的影像显示于屏幕之上
    // (或是在视窗模式下，把背景缓冲复制至前景缓冲)
    swapBuffers();
}
```

7.2 游戏循环

游戏由许多互动的子系统所构成，包括输入/输出设备、渲染、动画、碰撞检测及决议、可选的刚体动力学模拟、多玩家网络、音频等。在游戏运行时，多数游戏引擎子系统都需要周期性地**提供服务**。然而，这些子系统所需的服务**频率**各有不同。动画子系统通常需要30Hz或60Hz的更新率，此更新率是为了和渲染子系统同步。然而，动力学模拟可能实际需要更频繁地更新（如120Hz[1]）。更高级的系统，例如人工智能，就可能只需要每秒1、2次更新，并且完全不需要和渲染循环同步。

有许多不同方法能实现游戏引擎子系统的周期性更新。我们即将探讨一些可行的架构方案。但首先，我们会以最简单的方法更新引擎子系统——采用单一循环更新所有子系统。这种循环常称为**游戏循环**（game loop），因为它是整个游戏的主循环，更新引擎中所有子系统。

7.2.1 简单例子：《乒》

《乒（Pong）》是著名的乒乓球类型电子游戏。这种游戏类型起源于William A. Higinbotham在1958年于布鲁克黑文（Brookhaven）国家实验室创作的《双人网球（Tennis for Two）》游戏，该游戏使用示波器（oscilloscope）显示游戏图形。此类型游戏变得闻名，皆因其后的数字计算机版本——Magnavox Oddysey公司的《Table Tennis》和Atari公司的《乒》。

《乒》游戏里有一个球，它于两块垂直球拍和上下两幅固定的横墙之间来回反弹。玩家使用旋转钮控制球拍的位置。（现在的重制版本会使用手柄、键盘或其他人体学接口设备去操控。）若球来到时球拍击不中，对方就会得分，球就会被重置，并开始新的游戏回合。

以下的伪代码演示了《乒》的游戏循环核心的可行形式：

[1]译注：Xbox 360上的Forza Motorsport 3声称采用360Hz的物理更新率。

```
void main() // 乒
{
    initGame();

    while (true)      // 游戏循环
    {
        readHumanInterfaceDevices();

        if (quiteButtonPressed())
        {
            break; // 离开游戏循环
        }

        movePaddles();
        moveBall();
        collideAndBounceBall();

        if (ballImpactedSide(LEFT_PLAYER))
        {
            incrementScore(RIGHT_PLAYER);
            resetBall();
        }
        else if (ballImpactedSide(RIGHT_PLAYER))
        {
            incrementScore(LEFT_PLAYER);
            resetBall();
        }

        renderPlayerfield();
    }
}
```

显然此例子是纯粹虚构出来的。原来的《乒》肯定不是以每秒30帧速率重绘整个屏幕的。在当时，CPU非常慢，用尽其能力只仅仅够实时渲染两条代表球拍的直线和一个代表球的方格。后来一些游戏机通常会加入二维精灵（sprite）的专门硬件绘画会移动的对象。然而，这里只需要关注概念，而不是原来的《乒》的实现细节。

正如代码中显示，当游戏开始运行时，会调用`initGame()`进行各子系统的设置，当中可能包括图形系统、人体学接口设备、音频系统等，然后就会进入游戏循环。第一个`while(true)`语句告诉我们该循环会永远继续执行，除非此循环被内部中断。循环中第一个任务是读取人体学接口设备的数据。我们会检查玩家是否按了“离开（quit）”按钮，若按了会用`break`语句退出游戏。然后，`movePaddles()`就会根据旋转钮的偏转、手柄或

其他输入设备的数据，向上、向下调整球拍位置。之后，moveBall()把球的速度加进当前位置[2]，求出下一帧的新位置。在collideAndBounceBall()中，球的位置会对墙和球拍进行碰撞检测。若有碰撞，便要根据碰撞重新计算球的位置。之后，还要检测球是否碰到屏幕的左右边界。碰到左右边界代表有一方失球，所以就应给对方加分，并重置球的位置及开始下个回合。最后，renderPlayfield()负责绘画整个屏幕的游戏内容。

7.3 游戏循环的架构风格

有多种方式可以实现游戏循环，但其核心通常都会有一个或多个简单循环，再加上不同的修饰。以下我们会探讨几种常见的架构。

7.3.1 视窗消息泵

在Windows平台，游戏除了要服务引擎本身的子系统，还要处理来自Windows操作系统的消息。因此，Windows上的游戏都会有一段代码称为**消息泵**（message pump）。其基本原理是先处理来自Windows的消息，无消息时才执行引擎的任务。典型消息泵的代码如下：

```
while (true)
{
    // 处理所有待处理的Windows消息

    MSG msg;

    while (PeekMessage(&msg, NULL, 0, 0 ) > 0)
    {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }

    // 再无Windows消息需要处理，执行我们“真正”的游戏循环迭代一次
    RunOneIterationOfGameLoop();
}
```

以上这种实现游戏循环的方式，其副作用是设置了任务的优先次序，处理Windows消息为先，渲染和模拟游戏为后。这带来的结果是，当玩家在桌面上改变游戏的视窗大小或移动视窗时，游戏就会愣住不动。

---

[2]译注：在简单的运动学中，$\mathbf{x}(t+\Delta t)=\mathbf{x}(t)+\mathbf{v}(t)\Delta t$，若帧率是固定的，即$\Delta t$为常数，那么就能把$\mathbf{v}(t)$当成是$\Delta t$时间内的偏移，此公式就只剩下一个加法。

7.3.2 回调驱动框架

多数游戏引擎子系统和第三方游戏中间套件都是以**程序库**（library）的方式构成的。程序库是一组函数及/或类，这些函数和类能被应用程序员随意调用。程序库对程序员提供最大限度的自由。但程序库有时候比较难用，因为程序员必须理解如何正确使用那些函数和类。

相比之下，有些游戏引擎和游戏中间套件则是以**框架**（framework）构成的。框架是半完成的应用软件——程序员需要提供框架中空缺的自定义实现（或覆写框架的预设行为）。但程序员对应用软件的控制流程只有少量控制（甚至完全不能控制），因为那些都是由框架控制的。

在基于框架的渲染引擎或游戏引擎之下，主游戏循环已为我们准备好了，但该循环里大部分是空的。游戏程序员可以编写回调函数（callback function）以“填充”当中缺少的细节。例如，OGRE渲染引擎本身是一个以框架包装的库。在底层，OGRE提供给程序员直接调用的函数。然而，OGRE也提供一套框架，框架封装了如何有效地运用底层OGRE库的知识。若选择使用OGRE框架，程序员便需要自`Ogre::FrameListener`派生一个类，并覆写两个虚函数：`frameStarted()`和`frameEnded()`。读者可能已猜出来，OGRE在渲染主三维场景的前后会调用这两个函数。OGRE框架对游戏循环的实现方式像以下的伪代码（可参考实际源代码OgreRoot.cpp中的`Ogre::Root::renderOneFrame()`）：

```
while (true)
{
    for (each frameListener)
    {
        frameListener.frameStarted();
    }

    renderCurrentScene();

    for (each frameListener)
    {
        frameListener.frameEnded();
    }

    finalizeSceneAndSwapBuffers();
}
```

某游戏的FrameListener实现可能是这样子的：

```
class GameFrameListener : public Ogre::FrameListener
{
public:
    virtual void frameStarted(const FrameEvent& event)
    {
        // 于三维场景渲染前所需执行的事情 (如执行所有游戏引擎子系统)
        pollJoypad(event);
        updatePlayerControls(event);
        updateDynamicsSimulation(event);
        resolveCollisions(event);
        updateCamera(event);
        // 等等
    }

    virtual void frameEnded(const FrameEvent& event)
    {
        // 于三维场景渲染后所需执行的事情
        drawHud(event);
        // 等等
    }
}
```

7.3.3 基于事件的更新

在游戏中，**事件**（event）是指游戏状态或游戏环境状态的有趣改变。事件的例子有：人类玩家按下手柄上的按钮、发生爆炸、敌方角色发现玩家等。多数游戏引擎都有一个**事件系统**，让各个引擎子系统登记其关注的某类型事件，当那些事件发生时就可以一一回应（详见14.7节）。游戏的事件系统通常和图形用户界面里的事件/消息系统非常相似（如微软的Windows视窗消息、Java AWT的事件处理、C#的delegate和event关键字）。

有些游戏引擎会使用事件系统来对所有或部分子系统进行周期性更新。要实现这种方式，事件系统必须容许发送未来的事件。换句话说，事件可以先置于队列，稍后才取出处理。那么，游戏引擎在实现周期性更新时，只需要简单地加入事件。在事件处理器里，代码便能以任何所需的周期进行更新。接着，该代码可以发送一个新事件，并设定该事件在未来1/30s或1/60s生效，那么这个周期性更新就能根据需要一直延续下去。

7.4 抽象时间线

游戏编程中，使用**抽象时间线**（abstract timeline）思考问题有时候极为有用。时间线是连续的一维轴，其原点（$t = 0$）可以设置为系统中其他时间线的任何相对位置。时间线可以用简单的时钟变量实现，该变量以整数或浮点数格式储存绝对时间值。

7.4.1 真实时间

我们可以直接使用CPU的高分辨率计时寄存器（见7.5.3节）来量度时间，这种时间在所谓的**真实时间线**（real timeline）上。此时间线的原点定义为计算机上次启动或重置之时。这种时间的量度单位是CPU周期（或其倍数），但其实只要简单地乘以CPU的高分辨率计时器频率，此单位便可以转换为秒数。

7.4.2 游戏时间

我们不应限制自己只使用真实时间线。我们可以为解决问题定义许多所需的时间线。例如，我们可以定义**游戏时间线**（game timeline），此时间线在技术上来说独立于真实时间。在正常情况下，游戏时间和真实时间是一致的。若希望暂停游戏，就可以简单地临时停止对游戏时间的更新。若要把游戏变成慢动作，可以把游戏时钟更新得慢于实时时钟。通过相对某时间线去缩放和扭曲另一时间线，就可以实现许多不同效果。

暂停或减慢游戏时间也是非常有用的调试工具。在追查不正常的渲染时，开发者可以暂停游戏时间，使所有动作冻结。同一时间，渲染引擎及调试用的飞行摄像机继续运作，只要它们采用另一个时钟（可以用**实时时钟**，或另一独立**摄像机时钟**）。那么开发者就可以利用摄像机在游戏世界中飞行，并从任意角度视察问题所在。此外，我们也可以实现逐步更新游戏时钟，其实现方法是，当游戏在暂停模式下，每次按下手柄或键盘上的“逐步更新”按钮，就把游戏时钟推前目标帧率的时间（如1/30s）。

当使用上述方法时，必须谨记，游戏暂停时游戏循环是继续进行的，仅仅是游戏时钟停止。而通过对暂停的时钟加上1/30s去实现单步更新，并不等于在游戏主循环设置断点，再按F5键运行一次迭代。两种单步操作可用来追踪不同类型的问题。需要记住两者的区别。

7.4.3 局部及全局时间线

我们可以想象其他各种时间线。例如，每个动画片段或音频片段都可以含有一个**局部时**

间线（local timeline），该时间线的原点（$t = 0$）定义为片段的开始。局部时间线能按原来制作或录制片段的时间量度播放时的进展时间。当在游戏中播放片段时，我们可用原来以外的速率来播放。例如，我们可以加速一个动画，或减慢一个音频片段。甚至可以反向播放动画，只要把时间逆转就行了。

所有这些效果都可以视觉化为局部和全局时间线之间的**映射**，如同真实时间和游戏时间的关系。要以原来的速率播放某个动画，只需简单地把该动画的局部时间线之始（$t = 0$）映射至全局时间线的某一刻（$\tau = \tau_{\text{start}}$），如图7.1所示。

要以一半速率播放动画片段，我们可以把它想象为，在映射局部时间线到全局时间线之前，放大局部时间线至原来的两倍长度。为达至此目的，我们除了记录片段的全局时间τ_{start}，还需记录时间缩放因子或播放速率R，如图7.2所示。片段甚至可以反向播放，只要使用负数的时间比例（$R < 0$），如图7.3所示。

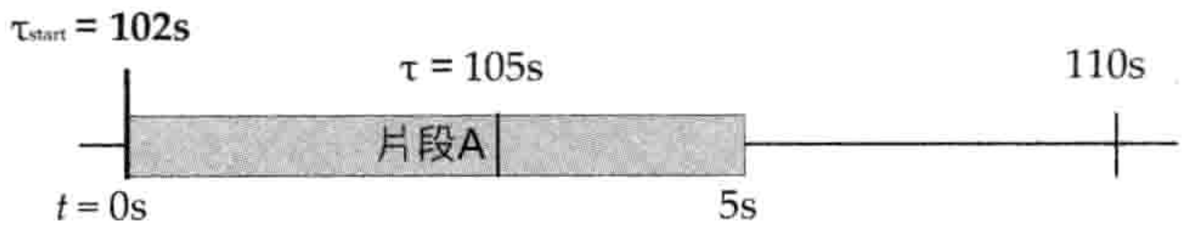


图 7.1: 播放一个动画片段，可当作是从局部时间映射至全局游戏时间。

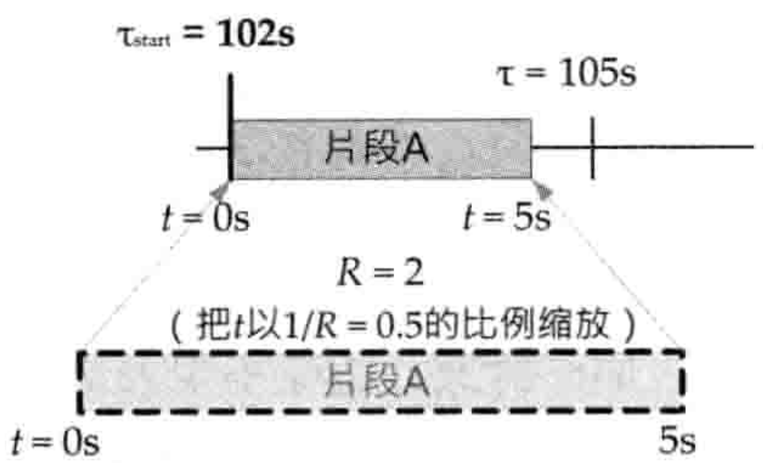


图 7.2: 为了控制动画播放速率，可简单地先缩放局部时间线，然后才映射至全局时间线。

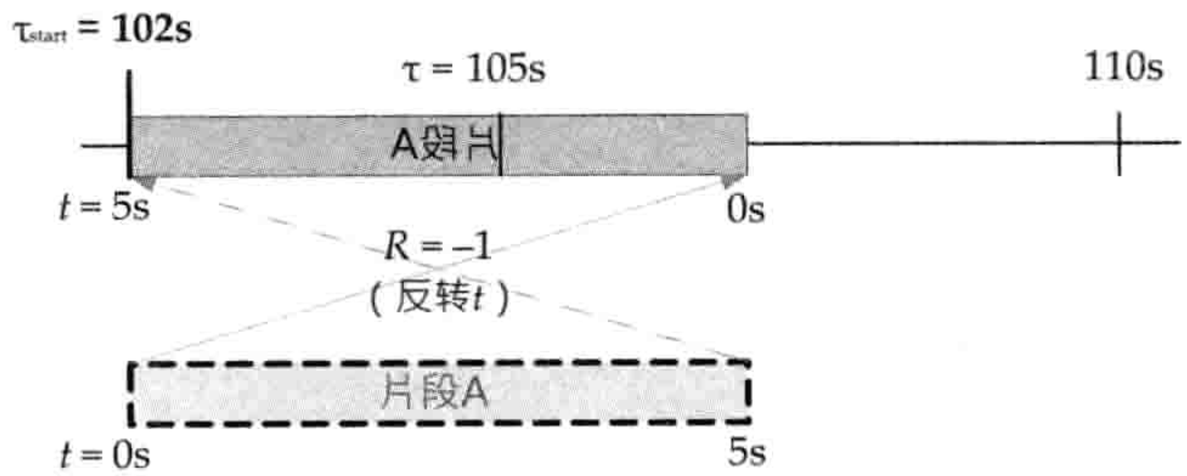


图 7.3: 为了把动画反向播放，可以用$R = -1$的时间比例来把片段映射至全局时间线。

7.5 测量及处理时间

本节会研究各种时间线和时钟之间一些细微与不那么细微的区别，并介绍在游戏中如何实现这些时间线和时钟。

7.5.1 帧率及时间增量

实时游戏的**帧率**（frame rate）是指一连串三维帧以多快的速度向观众显示。帧率的单位为**赫兹**（Hertz，Hz），即每秒的周期数量，这个单位可以用来描述任何周期性过程的速率。在游戏和电影里，帧率通常以**每秒帧数**（frame per second，FPS）来量度，其意义与赫兹完全相等。传统上电影以24 FPS播放。北美和日本的游戏通常以30或60FPS渲染，因为此帧率和这些地区的NTSC制式彩色电视的更新频率匹配。在欧洲及世界上其他地方，游戏以50FPS更新，因为这是PAL或SECAM制式彩色电视信号的更新频率。

两帧之间所经过的时间称为**帧时间**（frame time）、**时间增量**（time delta）或**增量时间**（delta time）。最后一个英文写法（delta time）很常见，因为两帧之间的持续时间在数学上常写成Δt。（技术上来说，Δt应该称为**帧周期**（frame period），因为它是**帧频率**（frame frequency）的倒数：$T = 1/f$。但是，在这种语境中，游戏程序员鲜会使用周期这个术语。）若游戏以准确30FPS渲染，那么其时间增量为1/30s，约33.3ms（毫秒/millisecond）。以60FPS渲染时，则时间增量会变成一半，即1/60s，约16.6ms。若需要知道游戏在游戏循环中每迭代实际上经过的时间，就需要进行测量，以下将做详细介绍。

笔者在此处也想指出，毫秒是游戏中常用的时间单位。例如，我们可能会说动画花费4ms，那就意味着动画占用整个帧预算的12% （$4/33.3 \approx 0.12$）[3]。其他常用时间单位还包括秒和机器周期。以下也会更深入地介绍时间单位和时钟变量。

7.5.2 从帧率到速率

假设我们想造一艘太空船，让它在游戏世界中以恒定速率每秒40m飞翔。（在二维游戏中，我们可能用每秒40个**像素**来设定速率！）实现此目标的简单方法是，把船的速率v（单位为米每秒）乘以一帧的经过时间Δt（单位为秒），就会得出该船的位置变化$\Delta x = v\Delta t$（单位为**米每帧**）。之后，此位置增量就能加到船的目前位置x_1，求得其次帧的位置：$x_2 = x_1 + \Delta x = x_1 + v\Delta t$。

以上例子其实是**数值积分**（numerical integration）的简单形式，名为**显式欧拉法**

[3]译注：这里是按30FPS为目标帧率。

(explicit Euler method)（详见12.4.4节）。若速率大致维持常数，此法可以正常运作。但是对于可变的速率，我们需要一些更复杂一点的积分方法。不过所有数值积分技术都需要使用帧时间Δt[4]。一个安全的说法是，游戏中物体的**感知速度**（perceived speed）依赖于帧时间Δt。因此，计算Δt的值仍是游戏编程的核心问题之一。以下数节会讨论几个计算方法。

7.5.2.1 受CPU速度影响的早期游戏

在许多早期的电视游戏中，并不会尝试在游戏循环中准确量度真实经过的时间。实质上，程序员会完全忽略Δt，取而代之，以米（或像素等其他距离单位）**每帧**设定速率。换言之，那些程序员可能在不知不觉下，以$\Delta x = v\Delta t$设定速率，而非使用v。

此简单方法造成的后果是，游戏中物体看上去的速度完全依赖于运行机器能产生的帧率。若在较快的CPU上运行这类游戏，游戏看上去就会像快速进带一样。因此，笔者称这类游戏为**受CPU速度影响的游戏**。

有些旧式PC带有“turbo”按钮，用来支持这类游戏。按下turbo按钮后，PC就会以其最高速度运行，但受CPU速度影响的游戏这时可能运行成快速进带的样子。当关上turbo按钮，PC就会模拟成上一代处理器的运行速度，使那些为上一代PC而设计的游戏能正常运行。

7.5.2.2 基于经过时间的更新

要开发和CPU速度脱钩的游戏，我们必须以某些方法量度Δt，而非简单地忽略它。量度Δt并非难事，只需读取CPU的高分辨率计时器取值两次——一次于帧开始之时，一次于结束之时。然后，取二者之差，就能精确量度上一帧的Δt。之后，Δt就能供所有引擎子系统使用，或可把此值传给游戏循环中调用到的函数，或把此值变成全局变量，或把此值包装进某种单例里。（7.5.3节会讨论有关高分辨率计时器的进一步细节。）

许多游戏引擎都会使用以上所说的方法。事实上，笔者大胆预测，**绝大部分**游戏引擎都使用以上的方法。然而，此方法有一大问题：我们使用第k帧量度出来的Δt去估计**接着的**第$k+1$帧的所需时间。这么做不一定准确。（如投资中常说：“过往表现不能作为日后表现的指标”。）下一帧可能因为某些原因，比本帧消耗更多（或更少）时间。我们称此类事件为**帧率尖峰**（frame-rate spike）。

使用上一帧的Δt来估计下一帧的时间，会产生非常坏的效果。例如，万一不小心，就

[4]译注：此处原文是elapsed frame time，即经过的帧时间。但较准确的说法是，在计算数值积分的时候，采用的Δt应为本帧所需模拟的时间，理论上和上一帧的帧时间无关，也并不能“量度”出来。之后的数节都是讲述如何决定这个Δt。

会使游戏进入低帧率的“恶性循环”。此情况可举例解释。假设当游戏以每33.3ms更新一次（即30Hz）时，物理模拟最为稳定。若然遇到有一帧特别慢，假设是57ms，那么我们便要在下一帧对物理系统步进**两次**，用以“掩饰”刚才经过的57ms。但步进两次会比正常消耗大约多一倍的时间，导致**下一帧**变成如本帧那么慢，甚至更慢。这样只会使问题加剧及延长。

7.5.2.3 使用移动平均

事实上，游戏循环中每帧之间是有一些连贯性的。例如，若本帧中摄像机对着某走廊，走廊出口含许多耗时渲染的物体，那么下一帧有很大机会仍然指向该走廊。因此，其中一个合理的方法是，计算连续几帧的平均时间，用来估计下一帧的Δt。此方法能使游戏适应转变中的帧率，同时缓和瞬间效能尖峰所带来的影响。平均的帧数越多，游戏对帧率转变的应变能力就越小，但受尖峰的影响也会变得越小。

7.5.2.4 调控帧率

使用上一帧的Δt估计本帧的经过时间，此做法带来的误差问题是可以避免的，只要我们把问题反转过来考虑。与其尝试**估算**下一帧的经过时间，不如尝试**保证**每帧都准确耗时33.3ms （若以60FPS运行就是16.7ms）。为达到此目标，我们仍然要量度本帧的耗时。若耗时比理想时间还要短，我们只需让主线程休眠，直至到达目的时间。若量度到的耗时比理想时间长，那么只好白等下一个目标时间。此方法称为**帧率调控**（frame-rate governing）。

显然，只当游戏的平均帧率接近目标帧率，此方法才有效。若因经常遇到“慢”帧，而导致游戏不断在30FPS和15FPS之间徘徊，那么就会明显降低游戏质量。因此，我们仍然需要让所有引擎系统设计成能接受任意的Δt。在开发时，可以把引擎停留在“可变帧率”模式，一切如常运作。之后，游戏能一贯地达到目标帧率，这样就能开启帧率调控，获其好处。

使帧率连续维持稳定，对游戏多方面都很重要。有些引擎系统，例如物理模拟中使用的数值积分，以固定时间更新运作最佳。稳定帧率也会较好看，因为如下一节的详述，更新视频的速率若不配合屏幕的刷新率会导致**画面撕裂**（tearing），而稳定帧率则可避免画面撕裂发生。

除此以外，当帧率连续维持稳定，一些如游戏**录播**功能会变得更可靠。游戏录播功能，如字面所指，能把玩家的游戏过程录制下来，之后再精确地回放出来。此功能既是供玩家用的有趣功能，也是非常有用的测试和调试工具。例如，一些难以找到的缺陷，可以通过游戏录播功能轻易重现。

为了实现游戏录播功能，需要记录游戏进行时的所有相关事件，并把这些事件及其时间戳（timestamp）储存下来。然后在播放时，使用相同的初始条件和随机种子，就能准确地按时间重播那些事件。理论上，这么做能产生和原来游戏过程一模一样的重播。然而，若帧率不稳定，事情可能以不完全相同的次序发生。因而造成一些“漂移”，很快就会使原来应在后退的AI角色变成在攻击状态中。[5]

7.5.2.5 垂直消隐区间

有一种显示异常现象，称为**画面撕裂**（tearing）。此现象的成因，是由于CRT显示器的电子枪在扫描中途交换背景缓冲区和前景缓冲区所引致[6]。当发生画面撕裂，屏幕上半部分显示了旧的影像，而下半部分则显示了新的影像。为避免画面撕裂，许多渲染引擎会在交换缓冲区之前，等待显示器的垂直消隐区间（vertical blanking interval，即电子枪重归到屏幕上角的时间区间）。

等待垂直消隐区间是另一种**帧率调控**。实际上它能限制主游戏循环的帧率，使其必然为屏幕刷新率的倍数。例如，在以60Hz刷新的NTSC显示器上，游戏的真实更新率实际会被量化为1/60s的倍数。若两帧之间的时间超过1/60s，便必须等待下一次垂直消隐区间，即该帧共花了2/60s（30FPS）。若错过两次垂直消隐，那么该帧共花了3/60s（20FPS），以此类推。此外，就算与垂直消隐同步，也不要假设游戏会以某特定帧率运行；谨记PAL和SECAM标准是基于大约50Hz的刷新率，而非60Hz。[7]

7.5.3 使用高分辨率计时器测量实时

我们已经谈及许多有关量度每帧所经过的真实“挂钟时间（wall clock time）”之事。本节会研究量度这种时间的方法细节。

大多数操作系统都提供获取系统时间的函数，例如标准C程序库函数`time()`。然而，因为这类函数所提供的量度分辨率不足，所以并不适合用在实时游戏中量度经过时间。再以`time()`为例，其传回值为整数，该值代表自1970年1月1日午夜至今的**秒数**，因此`time()`的分辨率为秒。考虑到游戏中每帧仅耗时数十毫秒，此量度分辨率实在太粗糙。

所有现代CPU都含有**高分辨率计时器**（high-resolution timer）。这种计时器通常会实现为硬件寄存器，计算自启动或重置计算机之后总共经过的CPU周期数目（或周期的倍数）。

---

[5]译注：此问题的简单解决方法是，同时记录每帧的Δt，使游戏性的逻辑模拟部分能完全重播录制时的状态。若播放时的帧率不能维持原来的速度，可选择以较慢的速度播放，或选择略过渲染一些帧。

[6]译注：实际上，LCD显示器也会出现同样情况，因为从显卡输出的视频信号本身已经可以有撕裂现象。

[7]译注：对于计算机游戏来说，显示器的刷新率有更多可能性。

量度游戏中经过的时间该使用这种计时器，因为其分辨率通常是几个CPU周期时间的级数。例如，在3GHz奔腾处理器上，其高分辨率计时器每周期递增一次，也就是每秒30亿次。因此其分辨率是30亿分之一 $= 3.33 \times 10^{-10}$s$= 0.333$ ns （纳秒/nanosecond）。此分辨率对于游戏中所有时间测量已绰绰有余。

各微处理器及操作系统中，查询分辨率计时器的方法各有差异。奔腾的特殊指令`rdtsc`（read time-stamp counter/读取时戳计数器）可供使用。但也可以使用经Windows封装的Win32 API函数：`QueryPerformanceCounter()`读取本CPU的64位计数寄存器，以及`QueryPerformanceFrequency()`传回本CPU的每秒计数器递增次数。一些PowerPC架构中（如Xbox 360及PS3）提供`mftb`（move from time base register/读取时间基寄存器）指令，用来读取两个32位时间基寄存器。另一些PowerPC架构则以`mfspr`（move from special-purpose register/读取特殊用途寄存器）代替。

大多数CPU的高分辨率计时器都是64位的，以免经常造成计时器溢出归零。64位无符号整数的最大值是0xFFFFFFFFFFFFFFFF，大约是1.8×10^{19}个周期。因此，以每CPU周期更新高分辨率计时器的3GHz奔腾处理器来说，其寄存器每次约195年才会溢出归零——肯定不是我们需要为此而失眠的问题。对比之下，32位整数时钟在3GHz下约每1.4s就会溢出归零。

7.5.3.1 高分辨率计时器的漂移

要注意，在某些情况下高分辨率计时器也会造成不精确的时间测量。例如，在一些多核处理器中，每个核有其独立高分辨率计时器，这些计时器可能（实际上会）彼此漂移（drift）。若比较不同核读取的绝对计算器读数，可能会出现一些奇异情况——甚至是负数的经过时间。对于这种问题必须加倍留神。

7.5.4 时间单位和时钟变量

每当要量度或指定持续时间，我们需要做两个决定。

1. 应使用什么**时间单位**？我们要把时间储存为秒、毫秒、机器周期，或是其他单位？
2. 应使用什么**数据类型**储存时间？应使用64位整数、32位整数，还是32位浮点数变量？

这些问题的答案在于量度时间的目的。这样又会引申两个问题：我们需要多少精度？以及我们期望能表示多大的范围？

7.5.4.1 64位整数时钟

我们之前已谈及以机器周期量度的64位无符号整数时钟，它同时支持非常高的精度（3GHz CPU上每周期是0.333ns）及很大的数值范围（3GHz CPU需约195年才循环一次）。因此这种时钟是最具弹性的表示法，只要你能负担得起64位的存储。

7.5.4.2 32位整数时钟

当要量度高精度但较短的时间，就可用以机器周期量度的32位整数时钟。例如，要剖析一段代码的效能，可以这么做：

```
// 抓取一个时间值
U64 tBegin = readHiResTimer();

// 以下是我们想量度性能的代码
doSomething();
doSomethingElse();
nowReallyDoSomething();

// 量度经过时间
U64 tEnd = readHiResTimer();
U32 dtCycles = static_cast<U32>(tEnd - tBegin);

// 现在可以使用或储存dyCycles的值……
```

注意我们仍然使用64位整数变量储存原始的时间量度。只有持续时间dt才用32位变量储存。这么做可以避免一些整数溢出问题。例如，若tBegin = 0x12345678FFFFFFB7及tEnd = 0x12345679**0**0000039，如果在相减之前先把这两个时间缩短为32位整数，那么就会得到一个负值的时间量度。

7.5.4.3 32位浮点时钟

另一常见方法是把较小的持续时间以秒为单位储存为浮点数。实现方法就是把以CPU周期为单位的时间量度除以CPU时钟频率（单位是每秒周期次数）[8]。例如：

[8]译注：原文误写为“乘以（multiply）”。原文代码也有同样错误，译者已修正。为了使用乘数，应该储存CPU时钟周期。

```
// 开始时假设为理想的帧时间 (30 FPS)
F32 dtSeconds = 1.0f / 30.0f;

// 在循环开始前先读取当前时间
U64 tBegin = readHiResTimer();

while (true) // 主游戏循环
{
    runOneIterationOfGameLoop(dtSeconds);

    // 再读取当前时间，计算增量
    U64 tEnd = readHiResTimer();
    dtSeconds = (F32)(tEnd - tBegin)
              / (F32)getHiRestTimerFrequency();

    // 把tEnd用作下一帧新的tBegin
    tBegin = tEnd;
}
```

再次注意我们必须先使64位的时间相减，**之后**才把两者之差转换为浮点格式。这样能避免把很大的数值储存进32位浮点数变量里。

7.5.4.4 浮点时钟的极限

回想在32位IEEE浮点数中，能通过指数把23位尾数动态地分配给整数和小数部分（见3.2.1.4节）。小数值中，整数部分占用较少位，于是便留下更多位精确地表示小数部分。但当时钟的值变得很大，其整数部分就会占用更多的位，小数部分剩下更少的位。最终，甚至整数部分的较低有效位都变成零。换言之，我们必须小心，避免用浮点时钟变量储存**很长的持续时间**。若使用浮点变量储存自游戏开始至今的秒数，最后会变得极不准确，无法使用。

浮点时钟只适合储存相对较短的持续时间，最多能量度几分钟，但更常见的是用来储存单帧或更短的时间。若在游戏中使用储存绝对值的浮点时钟，便需要定期将其重置为零，以免累加至很大的数值。[9]

[9]译注：例如，某游戏中的河流是用纹理滚动（texture scrolling）的方式产生流动效果的，常见的实现方法是把一个浮点时钟乘以移动速度，再把其位移结果传至着色器，然后在着色器中使纹理坐标加上这个位移。若放下不管，时间增大也导致位移一直增大，动画效果最终就会变得抖动、变形，甚至停顿。对于这类周期性的时钟，应把时钟按周期手动循环，例如使用代码`time = (time + dt) % period`来更新时钟变量。

7.5.4.5 其他时间单位

有些游戏引擎支持把时间值设定为游戏自定义单位，使32位时钟既有足够的精度，也不会很快就溢出循环。其中一个常见的选择是以1/300s为时间单位。此选择也有几个优点：（a）在许多情况之下也足够精确，（b）约165.7天才会溢出，（c）同时是NTSC和PAL刷新率的倍数。在60FPS下，每帧就是5个这种单位；在50FPS下，每帧就是6个这种单位。

显然1/300s时间单位并不足够精确地处理一些细微的效果，例如动画的时间缩放。（若尝试把30FPS的动画减慢至正常的1/10速度，这种单位产生的精度就已经不行了！）所以对很多用途来说，浮点数或机器周期仍是比较合适之选。而1/300s这种单位，能有效应用于诸如自动枪械每次发射之间的空档时间、由AI控制的角色要等多久才开始巡逻，或玩家留在硫酸池里能存活的时间期限。

7.5.5 应付断点

当游戏在运行时遇到断点，游戏循环便会暂停，由调试器接手控制。然而，这时候CPU还在运行，实时时钟仍然会继续累积周期次数。当程序员在断点里查看代码时，挂钟时间同时大量流逝。直至程序员继续执行程序时，该帧的持续时间才可能会量度为几秒、几分钟，甚至几小时！

显然，若把这么大的增量时间传到引擎中各子系统，必然有坏事发生。若我们幸运，游戏在一帧里蹒跚地执行很多秒的事情后，仍可继续运作。更糟的情况是导致游戏崩溃。

有一个简单方法可以避开此问题。在主游戏循环中，若量度到某帧的持续时间超过预设的上限（如1/10s），则可假设游戏刚从断点恢复执行，于是我们把增量时间人工地设为1/30s或1/60s（或其他目标帧率）。其结果是，游戏在一帧里锁定了增量时间，从而避免一个巨大的帧时间量度尖峰。

```
// 开始时假设为理想的帧时间 (30 FPS)
F32 dtSeconds = 1.0f / 30.0f;

// 在循环开始前先读取当前时间
U64 tBegin = readHiResTimer();

while (true)     // 游戏主循环
{
    updateSubsystemA(dt);
    updateSubsystemB(dt);
    // ……
```

```
renderScene();
swapBuffers();

// 再读取当前时间，估算下帧的时间增量
U64 tEnd = readHiResTimer();
dtSeconds = (F32)(tEnd - tBegin)
          / (F32)getHiRestTimerFrequency();

// 若 dt 过大，一定是从断点中恢复过来的，那么我们锁定dt至目标帧率
if (dt > 1.0f / 30.0f)
{
    dt = 1.0f / 30.0f;
}

    // 把tEnd用作下一帧新的tBegin
    tBegin = tEnd;
}
```

7.5.6 一个简单的时钟类

有些游戏引擎会把时间变量封装为一个类。引擎可能含此类的数个实例——一个用作表示真实“挂钟时间”、另一个表示“游戏时间”（此时间可以暂停，或相对真实时间减慢/加快）、另一个记录全动视频的时间等。实现时钟类很简单直接。以下笔者将介绍一个简单实现，并提示当中几个常见窍门、技巧及陷阱。

时钟类通常含有一个变量，负责记录自时钟创建以来经过的绝对时间。如上文所述，选择合适的数据类型和单位储存此变量，至关重要。在以下的例子中，笔者使用和CPU相同的储存绝对时间方法——以机器周期为单位的64位无符号整数。当然，可以有其他各种实现，但此例子大概是最简单的。

时钟类也可以支持一些很棒的特性，例如时间缩放。实现此功能并不困难，只需把量度得来的时间增量先乘以时间缩放因子，然后才进时钟变量。我们也可以暂停时间，只要在暂停时忽略更新便可以了。要实现单步时钟，只需在按下某按钮或键时，把固定的时间区间加到暂停中的时钟。以下的Clock类能示范所有这些特性：

```
class Clock
{
    U64     m_timeCycles;
    F32     m_timeScale;
    bool    m_isPaused;
```

```
static F32      s_cyclesPerSecond;

static inline U64 secondsToCycle(F32 timeSeconds)
{
    return (U64)(timeSeconds * s_cyclesPerSecond);
}

// 警告：危险——只能转换很短的经过时间至秒
static inline F32 cyclesToSeconds(U64 timeCycles)
{
    return (U64)(timeCycles / s_cyclesPerSecond);
}
```

```
public:
    // 在游戏启动时调用此函数
    static void init()
    {
        s_cyclesPerSecond = (F32)readHiResTimerFrequency();
    }

    // 构建一个时钟
    explicit Clock(F32 startTimeSeconds = 0.0f) :
        m_timeCycles(secondToCycles(startTimeSeconds)),
        m_timeScale(1.0f), // 默认为无缩放
        m_isPaused(false)  // 默认为运行中
    {
    }

    // 以周期为单位返回当前时间。注意我们并不是返回以浮点秒表示的绝对时间，
    // 因为32位浮点没有足够的精确度。参考calcDeltaSeconds()
    U64 getTimeCycles() const
    {
        return m_timeCycles;
    }

    // 以秒为单位，计算此时钟与另一时钟的绝对时间差
    // 由于32位浮点的精度所限，传回的时间差是以秒表示的
    F32 calcDeltaSeconds(const Clock& other)
    {
        U64 dt = m_timeCycles - other.m_timeCycles;
        return cyclesToSeconds(dt);
    }

    // 应在每帧调此函数一次，并给予真实量度帧时间（以秒为单位）
    void update(F32 dtRealSeconds)
```

```
{
    if (!m_isPaused)
    {
        U64 dtScaledCycles = secondsToCycles(
            dtRealSeconds * m_timeScale);

        m_timeCycles += dtScaledCycles;
    }
}

void setPaused(bool isPaused)
{
    m_isPaused = isPaused;
}

bool isPaused() const
{
    return m_isPaused;
}

void setTimeScale(F32 scale)
{
    m_timeScale = scale;
}

F32 getTimeScale() const
{
    return m_timeScale;
}

void singleStep()
{
    if (m_isPaused)
    {
        // 加上理想帧时间：别忘记把它缩放至我们当前的时间缩放率!
        U64 dtScaledCycles = secondToCycles(
            (1.0f / 30.0f) * m_timeScale);

        m_timeCycles += dtScaledCycles;
    }
}
};
```

7.6 多处理器的游戏循环

至今我们已经研究过基本的单线程游戏循环，并学习了游戏引擎中量度及处理时间的常用方法，现在可以开始讨论一些较复杂的游戏循环类型。在本节中，我们会探讨如何让游戏循环进化至使用现代多处理器（multiprocessor）硬件。下一节就会讨论网络游戏通常如何架构其游戏循环。

2004年，所有微处理器生产商都碰上芯片散热问题，导致无法制造更快的CPU。但摩尔定律（Moore's Law）——预计每18～24个月芯片的晶体管（transistor）会增加1倍——仍然有效。但到了2004年，此定律与处理器速度加倍的相关性已经告吹。因此，微处理器生产商转移了注意力，集中开发多核（multicore）CPU。（关于这个趋势的详情，可参阅微软的“The Manycore Shift White Paper/众核的转变白皮书”[10]；另一篇是Dean Dauger的“Multicore Eroding Moore's Law/多核在削弱摩尔定律”[11]）这直接影响软件行业，纷纷转向并行处理（parallel processing）技术。因此，运行于多核系统如Xbox 360及PlayStation 3的现代游戏引擎，已不能再使用单个游戏主循环去服务其多个子系统。

从单核到多核的转变是个痛苦历程。设计多线程程序比单线程的难得多。多数游戏公司逐步进行此转变，其做法是选择几个引擎子系统做并行化，并保留用旧的单线程主循环控制余下的子系统。至2008年，多数游戏工作室已完成引擎大部分的转变，对每个引擎带来不同程度的并行性。

本书篇幅未能完整详述并行编程的架构和技术。（关于此题目的深入探讨可参阅[20]。）然而，我们会扼要说明一些让游戏引擎利用多核硬件的最常见方法。有许多不同的软件架构都是可行的，它们的目标都是要最大化硬件使用率（即尝试令硬件线程、核或CPU的闲置时间变得最少）。

7.6.1 多处理器游戏机的架构

Xbox 360和PlayStation 3皆为多处理器游戏机。为了有意义地讨论并行软件架构，我们先简单了解此这款游戏机的内部结构。

---

[10] http://www.microsoft.com/en-us/download/details.aspx?id=17702（译注：此为新的下载链接。）

[11] http://www.macresearch.org/multicore_eroding_moores_law

7.6.1.1 Xbox 360

Xbox 360游戏机含3个完全相同的PowerPC处理器核。每个核有其专用的L1指令缓存和L1数据缓存，而3个核则共用一个L2缓存。此3个核和图形处理器（graphics processing unit，GPU）共用一个统一的512MB内存[12]。这些内存可用来存放可执行代码、应用数据、纹理、显存等。关于Xbox 360架构的更详尽说明，可参考Xbox半导体技术组的Jeff Andrews和Nick Baker所写的“Xbox 360 System Architecture/Xbox 360系统架构”[13]。然而，以上谈及的极简介绍对本节来说已经足够。图7.4描绘了非常简化的Xbox 360架构。

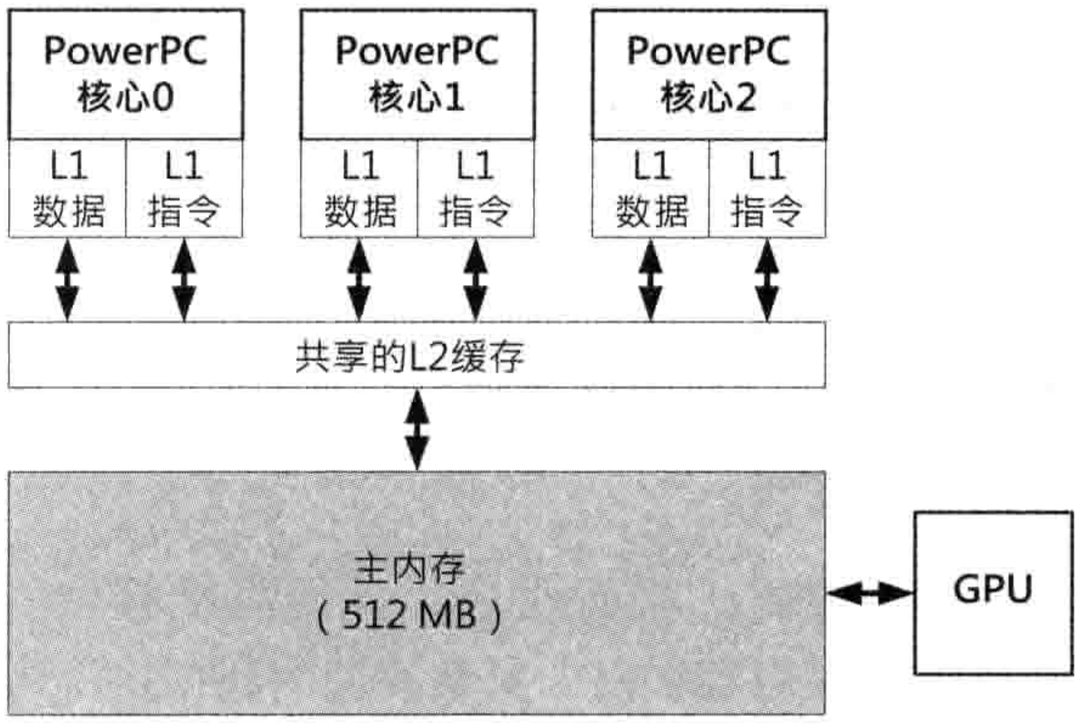


图 7.4: 简化了的Xbox 360硬件架构。

7.6.1.2 PlayStation 3

PlayStation 3硬件采用由索尼、东芝和IBM共同开发的Cell Broadband Engine（CBE）架构（见图7.5）。PS3采用了跟Xbox 360彻底不同的架构设计。PS3不采用3个相同处理器，而是提供不同种类处理器，每种处理各为特定任务而设计。PS3也不采用统一内存架构（unified memory architecture，UMA），而是把内存切割为多个区块，每块为提升系统中特定处理器的效率而设计。有关PS3架构的详细说明可参阅此网页[14]，然而以下的简介和图7.5对本节来说已经足够。

PS3的主CPU称为Power处理部件（Power Processing Unit，PPU）。此乃一个PowerPC处理器，和Xbox 360中的分别不大。除此处理器之外，PS3还有6个副处理器，名为协同处理部件（Synergistic Processing Unit，SPU）。这些副处理器是基于PowerPC指令集的，但它们经特别设计以提供最大效能。

[12]译注：GPU另外还内置10MB EDRAM，做渲染目标缓存区之用。

[13]http://www.hotchips.org/archives/hc17/3_Tue/HC17.S8/HC17.S8T4.pdf

[14]http://www.blachford.info/computer/Cell/Cell1_v2.html

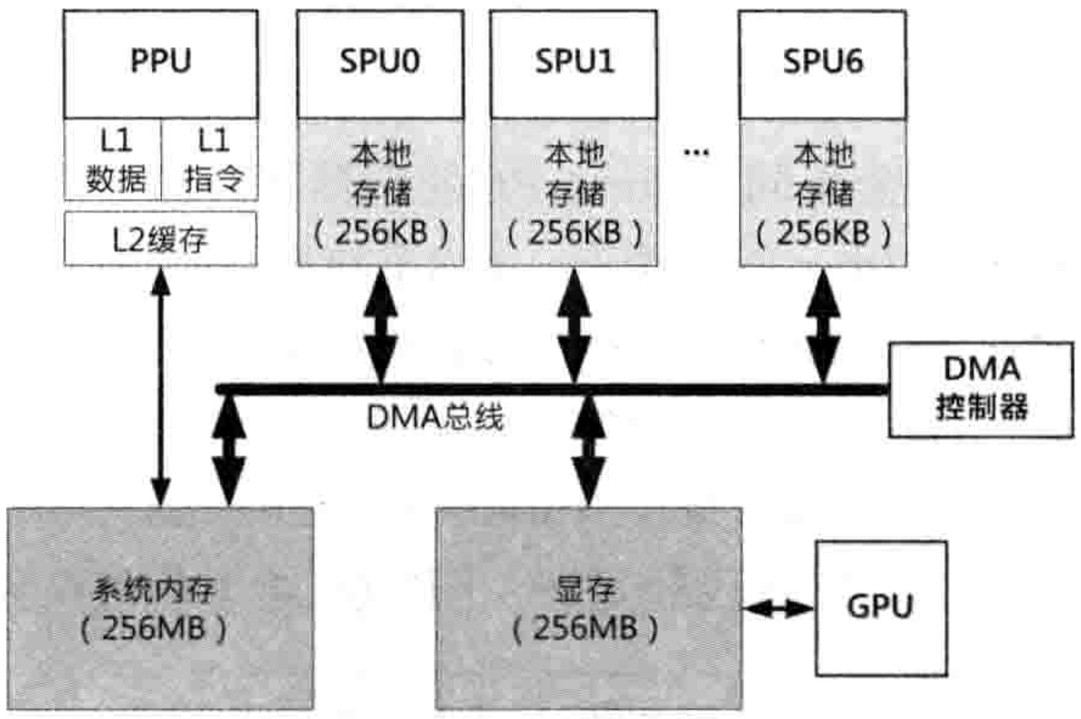


图 7.5: 简化了的PS3 Cell Broadband架构。

PS3的GPU含专用的256MB显存，而PPU则能存取256MB系统内存。此外，每个SPU含专用高速的256KB内存区，称为SPU的**局部存储**（local store，LS）。局部存储内存如L1缓存那么高效，使SPU运作得极其之快。

SPU不能直接读取主内存数据。取而代之，要使用直接内存访问（direct memory access，DMA）控制器来回复制主内存和SPU局部存储的数据块。这些数据传输是并行执行的，因此PPU和SPU在等待数据到达前仍能进行运算。

7.6.2 SIMD

第4.7节曾提及，多数现代的CPU（包括Xbox 360中3个PowerPC处理器、PS3的PPU和SPU）都会提供**单指令多数据**（single instruction multilpe data，SIMD）指令集。这类指令能让一个运算同时执行于多个数据之上，此乃一种细粒度形式的硬件并行。CPU一般提供几类不同的SIMD指令，然而游戏中最常用的是并行操作4个32位浮点数值的指令，因为相比单指令单数据（single instruction single data，SISD）指令，这种SIMD指令能使三维矢量和矩阵数学的运算加速至4倍。

把现存三维数学代码改为使用SIMD指令可能会有点棘手。若原来的代码使用到封装良好的三维数学库，那么转换至SIMD的工作会容易很多。例如，若源程序到处都有手写的点积计算（如`float d = ax * bx + ay * by + az * bz;`），那么便需要重写很大量的代码。相反，若源程序是以调用函数来计算点积的（如`float d = Dot(a, b);`），并且矢量在源程序中基本上是当作黑盒处理的，那么改用SIMD时只需改变三维数学库，而不用更改调用该库的代码（不过大概需要确保矢量数据是以16字节对齐的）。

7.6.3 分叉及汇合

另一种利用多核或多处理器硬件的方法是，采用并行的分治（divide-and-conquer）算法。这通常称为**分叉 / 汇合**（fork/join）法。其基本原理是把一个单位的工作分割成更小的子单位，再把这些工作量分配到多个核或硬件线程（分叉），最后待所有工作完成后再合并结果（汇合）。把分叉/汇合法应用至游戏循环时，其产生的架构看上去和单线程游戏循环很相似，但是更新循环的几个主要阶段都能并行化。图7.6说明了此架构。

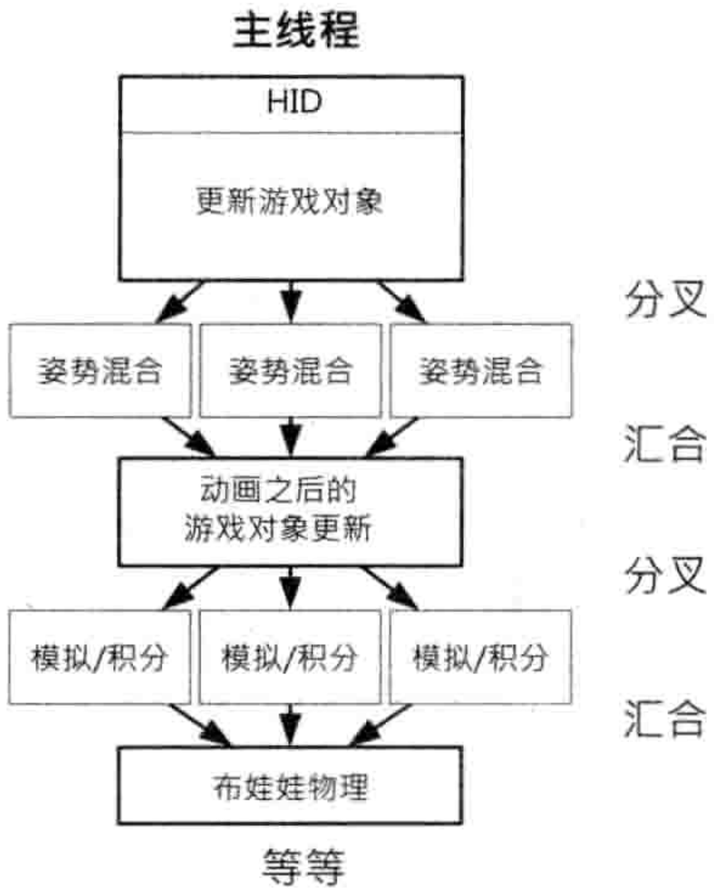


图 7.6: 在游戏循环中，使用分叉/汇合方式去并行化数个选定的CPU密集阶段。

我们再看一个实际例子。若动画混合（animation blending）使用线性插值（linear interpolation，LERP），其操作可以独立地施于骨骼上所有关节（见11.5.2.2节）。假设有5个角色，要混合每个角色的一对骨骼姿势（skeletal pose），当中每个骨骼有100个关节（joint），那么总共需要处理500对关节姿势（joint pose）。

要把此工作并行化，可以切割工作至N个批次，每批次含约$500/N$个关节姿势对，而N是按可用的处理器资源来设定的。（在Xbox 360上，N应该会是3或6，因为该游戏机有3个核，每个核有两个硬件线程。而在PS3上，N可以是1～6，视乎有多少个SPU可以使用。）然后我们“分叉”（即建立）N个线程，让每个线程各自执行分组后的姿势对。主线程可以选择继续工作，做一些和该次动画混合无关的事情；主线程也可选择等待信号量（semaphore）直至所有其他线程完成工作。最后，我们把各个关节姿势结果“汇合”成整体结果——在这例子里，就是要计算成5个骨骼的最终全局姿势。（每个骨骼计算全局姿势时，需用上所有关节的局部姿势，因此，对单个骨骼进行这种计算并不能并行化。然而，我们可以考虑再次分叉计算全局姿势，不过这次每线程负责计算一个或多个完整的骨骼。）

网页[15]含示范代码，说明如何在Win32系统调用来实现分叉 / 汇合工作线程。

7.6.4 每个子系统运行于独立线程

另一个多任务方法是把每个引擎子系统置于独立线程上运行。主控线程（master thread）负责控制及同步这些子系统的次级子系统，并继续应付游戏的大部分高级逻辑（游戏主循环）。对于含多个物理CPU或物理线程的硬件平台来说，此设计能让这些子系统并行执行。此设计适合某些子系统，那些子系统需重复地执行较有隔离性的功能，例如渲染引擎、物理模拟、动画管道、音频引擎等。图7.7说明了这种架构。

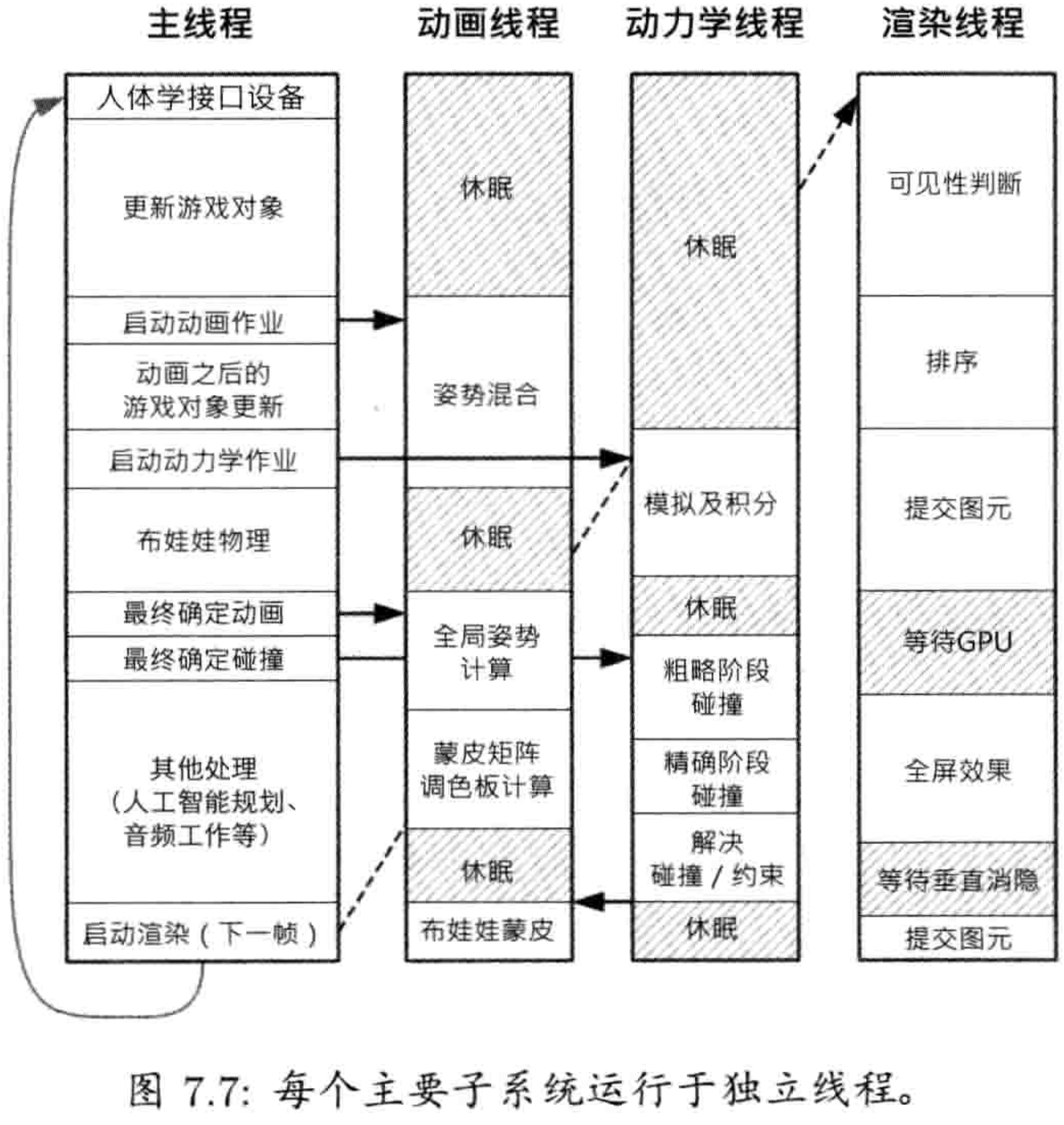


图 7.7: 每个主要子系统运行于独立线程。

多线程架构通常需要由目标硬件平台上的线程库所支持。在运行Windows的个人计算机上，通常会使用Win32的线程API。在基于UNIX的平台上，类似**pthread**的库可能是最佳选择。在PlayStation 3上，有一个名叫SPURS的库，可把工作运行于6个SPU之上。SPURS提供两种在SPU运行代码的基本方法——**任务模型**（task model）和**作业模型**（job model）。任务模型可用来把引擎子系统分离为粗颗粒度的独立执行单位，运作上与线程相似。下一节将讨论SPURS的作业模型。

[15] http://msdn.microsoft.com/en-us/library/ms682516(VS.85).aspx

7.6.5 作业模型

使用多线程的问题之一就是，每个线程都代表相对较粗粒度的工作量（例如，把所有动画任务都置于一个线程，把所有碰撞和物理任务置于另一线程），这么做会限制系统中多个处理器的利用率。若某个子系统线程未完成其工作，就可能会阻塞主线程和其他线程。

为充分利用并行硬件架构，另一种方法是让游戏引擎把工作分割成多个细小、比较独立的**作业**（job）。作业最好理解为，一组数据与操作该组数据的代码结合成对。作业准备就绪后，就可以加入队列中，待至有闲置的处理器，作业才会从队列取出执行。PS3的SPURS库的**作业模型**就是实现这种方法。使用该模型时，游戏主循环在PPU上执行，而6个SPU则为作业处理器。每个作业的代码和数据会通过DMA传送至SPU的局部存储，然后SPU执行作业，并把结果以DMA传回主内存。

如图7.8所示，作业较为细粒度且独立，因而有助于最大化处理器的利用率。相比“每个子系统运行于独立线程”的设计，这种方法也可减少或消除对主线程的一些限制。此架构也能自然地对任何数量的处理单元向上扩展（scale up）或向下缩减（scale down）（“每个子系统运行于独立线程”的架构就不太能做到）。

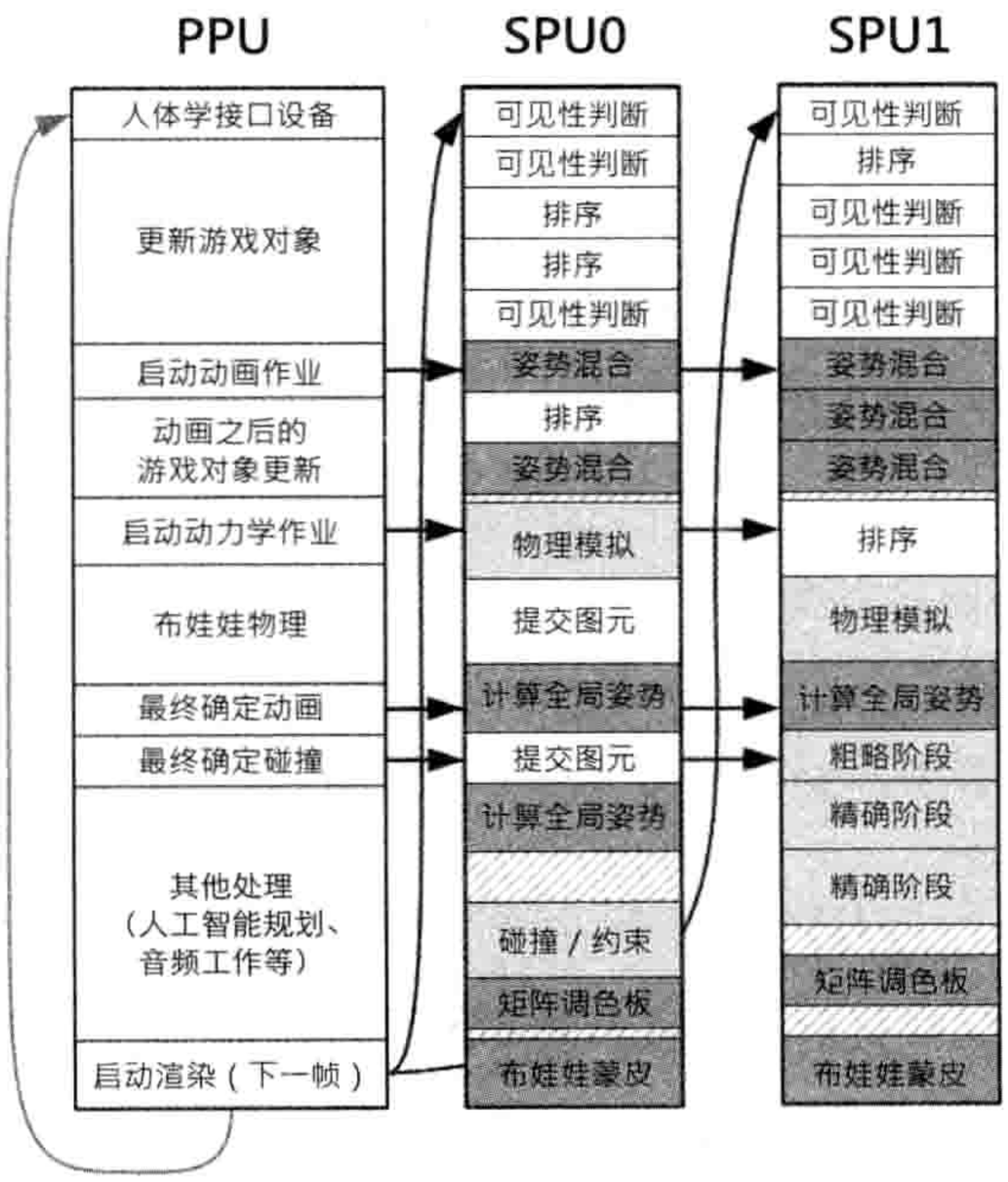


图 7.8: 在作业模型中，工作被拆分为小颗粒度的作业，这些作业可于任何闲置的处理器运行。这样能最大化处理器的使用率，并能增加主游戏循环的弹性。

7.6.6 异步程序设计

为了利用多处理器硬件而编写或更新游戏引擎，程序员必须小心设计异步方式的代码。这里所谓的异步，指发出操作请求之后，通常不能立即得到结果。而平时的同步设计，就是程序等待结果之后才继续运行。例如，某游戏可能会通过向世界进行光线投射（ray cast），以得知玩家角色是否能看见敌人。使用同步设计时，提出光线投射请求后便会立即执行，当光线投射函数执行完毕，就会传回结果，如以下代码所示：

```
while (true)      // 游戏主循环
{
    // ……
    // 投射一条光线以判断玩家能否看见敌人
    RayCastResult r = castRay(playerPos, enemyPos);

    // 现在处理结果
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // 玩家能看见敌人
        // ……
    }
    // ……
}
```

而使用异步设计，提出光线投射请求时，调用的函数只会建立一个光线投射作业，并把该作业加到队列中，然后该函数就会立即返回。主线程可继续做其他跟该作业无关的工作，同一时间另一个CPU或核就会处理那个作业。之后，当作业完成，主线程就能提取并处理光线投射的结果：

```
while (true)      // 游戏主循环
{
    // ……
    // 投射一条光线以判断玩家能否看见敌人
    RayCastResult r;
    requestRayCast(playerPos, enemyPos, &r);

    // 当等待其他核做光线投射时，我们做其他无关的工作
    // ……

    // 好吧，我们不能再做更多有用的事情了，等待光线投射作业的结果
    // 若作业完毕，此函数会立即返回。否则，主线程会闲置直至有结果
    waitForRayCastResults(&r);
```

```
    // 处理结果
    if (r.hitSomething() && isEnemy(r.getHitObject()))
    {
        // 玩家能看见敌人
        // ……
    }
    // ……
}
```

许多时候，异步代码可以在某帧启动请求，而在下一帧才提取结果。这种情况的代码可能是这样的：

```
RayCastResult r;
bool rayJobPending = false;

while (true)     // 游戏主循环
{
    // ……
    // 等待上一帧的光线投射结果
    if (rayJobPending)
    {
        waitForRayCastResults(&r);

        // 处理结果
        if (r.hitSomething() && isEnemy(r.getHitObject()))
        {
            // 玩家能看见敌人
            // ……
        }
    }

    // 为下一帧投射一条光线
    rayJobPending = true;
    requestRayCast(playerPos, enemyPos, &r);

    // 做其他事情
    // ……
}
```

7.7 网络多人游戏循环

网络多人游戏的游戏循环特别有趣，因此我们会简单介绍其架构。基于篇幅有限，本节未能谈及多人游戏的所有运作细节。（想深入了解本题目可参考[3]。）然而，本节会简单介绍两种最常见的多人架构，并探讨这些架构如何影响游戏循环的结构。

7.7.1 主从式模型

在**主从式**模型（client-server model）中，大部分游戏逻辑运行在单个服务器（server）上。因此服务器的代码和非网络的单人游戏很相似。多个客户端（client）可连接至服务器，以一起参与线上游戏。客户端基本上只是一个“非智能（dumb）”渲染引擎，客户端会读取人体学接口设备数据，以及控制本地的玩家角色，但除此以外，客户端要渲染什么都是由服务器告之。但这么做最痛苦的是，客户端代码需要即时把玩家的输入转换成玩家角色在屏幕上的动作。不然，玩家会觉得他控制的游戏角色反应非常缓慢，非常恼人。除了这些称为**玩家预测**（player prediction）的代码，客户端通常仅为渲染和音频引擎，加上一些网络代码。

服务器可以单独运行于一个机器上，此运行方式称为**专属服务**模式（dedicated server mode）。然而，客户端和服务器不一定要运行于两个独立的机器上，其实客户端机器同时运行服务器也是十分普遍的。实际上，在许多主从式多人游戏中，单人游戏模式其实是退化的多人游戏——当中只有一个客户端，并且把客户端和服务器运行在同一个机器上。这种运行方式又称为**客户端于服务器之上**模式（client-on-top-of-server mode）。

主从多人游戏的游戏循环有多种不同的实现方法。由于客户端和服务器理论上是独立的实体，两者可分别实现为完全独立的行程（process）（即不同的应用程序）。另一种实现方式是，把两者置于同一行程内的两个独立线程。但是，当采用客户端置于服务器之上模式时，以上两个方法都会带来不少本地通信方面的额外开销。因此，许多多人游戏会把客户端和服务器都置于单个线程中，并由单个游戏循环控制。

必须注意，客户端和服务器的代码可能以不同频率进行更新。例如，在《雷神之锤》中，服务器以20FPS运行（每帧50ms），而客户端通常以60FPS运行（每帧16.6ms）。其实现方式是，把主游戏循环以两帧速中较快的频率（60FPS）运行，并让服务器代码大约每3帧才运行一次。真正实现时，会计算上次服务器更新至今的经过时间，若超过50ms，服务器就会运行一帧，然后重置计时器。这种游戏循环大概是以下这样的：

```
F32 dtReal = 1.0f / 30.0f;   // 真实的帧时间增量
F32 dtServer = 0.0fl         // 服务器的时间增量

U64 tBegin = readHiResTimer();

while (true)     // 主游戏循环
{
    // 以50ms区间运行服务器
    dtServer += dtReal;

    if (dtServer >= 0.05f)   // 50ms
    {
        runServerFrame(0.05f);
        dtServer -= 0.05f;   // 重置供下次更新
    }

    // 以最大帧率执行客户端
    runClientFrame(dtReal);

    // 再读取当前时间，估算下帧的时间增量
    U64 tEnd = readHiResTimer();
    dtReal = (F32)(tEnd - tBegin)
           / (F32)getHiRestTimerFrequency();

    // 把tEnd用作下一帧新的tBegin
    tBegin = tEnd;
}
```

7.7.2 点对点模型

在**点对点**（peer-to-peer）多人架构中，线上世界中的每部机器都有点像服务器，也有点像客户端。游戏中每个动态对象，都由其对应的单一机器所管辖。因此，每个机器对其拥有管辖权（authority）的对象就如同服务器。对于其他无管辖权的对象，机器就如同是客户端，只负责渲染由对象的远端管辖者所提供的状态。

点对点多人游戏循环的结构比主从游戏的简单得多。从最高级的角度来看，点对点多人游戏循环的结构和单人游戏的相似。然而，其内部代码细节可能较难理解。在主从模型中，较能清楚知道哪些代码运行于服务器，哪些运行于客户端。但在点对点架构里，许多代码都要处理两个可能情况：本地机器拥有某对象状态的管辖权，或者本地某对象只是有其管辖权远端机器的哑代理（dumb proxy）。此两种模式通常实现为两种游戏对象，一种是本机有管辖权的完整“真实”游戏对象，另一种是“代理版本”，仅含远程对象状态的最小子集。

点对点架构可以设计得更复杂，因为有时候需要把对象的管辖权从某机器转移至另一机器。例如，若其中一部计算机离开游戏，该计算机所有对象的管辖权必须转移至其他参与该游戏的机器。相似地，若有新机器加入游戏，理想地该机器应接管其他机器的一些游戏对象，以平衡每部机器的工作量。这些细节超出本书范围。本节希望带出的重点是，多人架构对于游戏主循环的结构有深远影响。

7.7.3 案例分析：《雷神之锤II》

以下是《雷神之锤II》游戏循环的节录。《雷神之锤》、《雷神之锤II》、《雷神之锤III竞技场》的源代码都可以在id Software的网站取得[16]。读者可以看到，本章谈及的元素都会出现在以下的代码节录中，包括Windows消息泵（在游戏的Win32版本中）、计算两帧之间的真实时间增量、操作固定时间和时间缩放模式，以及更新服务器端和客户端的引擎系统。

```
int WINAPI WinMain (HINSTANCE hInstance,
                    HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    MSG     msg;
    int     time, oldtime, newtime;
    char    *cddir;

    ParseCommandLine (lpCmdLine);

    Qcommon_Init (argc, argv);
    oldtime = Sys_Milliseconds ();

    /* Windows 主消息循环 */
    while (1)
    {
        // Windows 消息泵
        while (PeekMessage (&msg, NULL, 0, 0, PM_NOREMOVE))
        {
            if (!GetMessage (&msg, NULL, 0, 0))
                Com_Quit ();
            sys_msg_time = msg.time;
            TranslateMessage (&msg);
            DispatchMessage (&msg);
        }

        // 以毫秒为单位量度真实的时间增量
```

[16] http://www.idsoftware.com

```
        do
        {
            newtime = Sys_Milliseconds ();
            time = newtime - oldtime;
        } while (time < 1);

        // 执行1帧游戏
        Qcommon_Frame (time);
        oldtime = newtime;
    }

    // 永不会到达这里
    return TRUE;
}

void Qcommon_Frame (int msec)
{
    char    *s;
    int     time_before, time_between, time_after;

    // 这里忽略一些细节……

    // 处理固定时间模式及时间缩放
    if (fixedtime->value)
        msec = fixedtime->value;
    else if (timescale->value)
    {
        msec *= timescale->value;
        if (msec < 1)
            msec = 1;
    }

    // 处理游戏中的主控台
    do
    {
        s = Sys_ConsoleInput ();
        if (s)
            Cbuf_AddText (va("%s\n", s));
    } while (s);
    Cbuf_Execute ();

    // 执行1帧服务器
    SV_Frame (msec);

    // 执行1帧客户端
```

```
        CL_Frame (msec);

        // 这里忽略一些细节……
}
```

第8章　人体学接口设备（HID）

游戏是有互动性的计算机模拟，因此玩家需要以某些方法把输入送往游戏。为游戏而设的**人体学接口设备**（human interface device，HID）种类繁多，包括摇杆（joystick）、手柄（joypad）、键盘、鼠标、轨迹球（trackball）、Wii摇控器（Wii Remote/WiiMote），以及专门的输入设备，如方向盘、鱼杆（fishing rod）、跳舞毯、电子吉他等。本章会探讨游戏引擎如何自人体学接口设备读取输入，以及把输入处理和应用的常用方法，也会介绍怎样从这些设备向玩家输出反馈。

8.1　各种人体学接口设备

有很多不同种类的人体学接口设备可供游戏使用。Xbox 360和PS3等游戏机配备如图8.1所示的手柄。众所周知，任天堂Wii游戏机备有独特、创新的Wii遥控器（图8.2）。PC游戏通常使用键盘、鼠标或手柄。（微软把Xbox 360手柄设计为可供Xbox 360和Windows PC平台使用。）街机机器有一个或多个内置控制器，如摇杆及多个按钮（图8.3），或是轨迹球、方向盘等。街机机器的输入设备通常会为游戏量身打造，但同一厂商的街机机器也会重用相同的输入硬件。

在游戏机平台上，除了提供如手柄这种“标准”输入设备，也会生产一些专门的输入设备和适配器。例如，《吉他英雄（Guitar Hero）》游戏系列就提供吉他和鼓等设备；赛车游戏有方向盘可供选购；《劲舞革命（Dance Dance Revolution/DDR）》类的游戏可使用特殊的跳舞毯。图8.4有部分相关设备。

任天堂的Wii遥控器是现今市场上最有弹性的输入设备之一。Wii遥控器时常可以通过配接额外硬件而获得新的用法，而不需要被全新设备取代。例如，《马里奥赛车Wii（Mario Kart Wii）》配备塑料方向盘配接器，能把Wii遥控器安装于其中（见图8.5）。

图 8.1: Xbox 360和PS3游戏机的标准手柄。

图 8.2: 任天堂的创新Wii遥控器。

图 8.3: Midway街机游戏《真人快打》的专用输入设备。

图 8.4: 游戏机还可使用很多特制的输入设备。

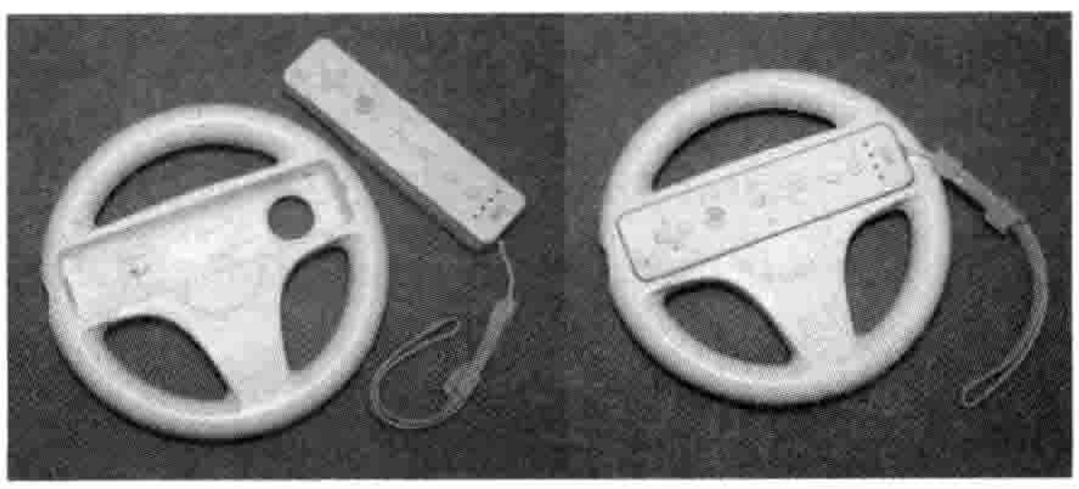


图 8.5: 任天堂Wii的方向盘配接器。

8.2 人体学接口设备的接口技术

所有人体学接口设备都能向游戏软件提供输入，有些设备也能通过多种输出为玩家提供反馈。按设备的具体设计，游戏软件可用多种方式读取输入及写进输出。

8.2.1 轮询

一些简单设备，如手柄和老式摇杆，可通过定期**轮询**（poll）硬件来读取输入（通常在主游戏循环里每次迭代轮询一次）。那就意味着明确地查询设备的状态，方法其一是直接读取硬件寄存器，其二是读取经内存映射的I/O端口，或是通过较高级的软件接口（该接口再转而读取适当的寄存器或内存映射I/O端口）。同样，也可使用上述方式把输出传到设备。

微软为Xbox 360手柄而设的XInput API能用于Xbox 360和Windows上。此API是简单轮询的好例子，以下简述其使用方法。游戏在每帧调用`XInputGetState()`函数时，该函数便会与硬件/驱动通信，适当地读取数据，并把所有结果包装以方便软件使用。`XInputGetState()`函数会把结果填入类型为`XINPUT_STATE`结构，而`XINPUT_STATE`又包含`XINPUT_GAMEPAD`结构。此结构含有手柄设备上所有输入的当前状态，包括按钮、拇指摇杆（thumb stick）及扳机（trigger）。

8.2.2 中断

有些HID只会当其状态有某些改变时，才会把数据传至游戏引擎。例如，鼠标有大部分时间都在鼠标垫上静止不动，无理由在其静止的时候，还要把数据不断传至计算机。只有在鼠标移动时、按下或释放按钮时，才需要传送数据。

这类设备通常和主机以**硬件中断**（hardware interrupt）方式通信。所谓中断，是由硬件生成的信号，能让CPU暂停主程序，并执行一小段称为**中断服务程序**（interrupt service routine，ISR）的代码。中断能应用在各个方面，但以HID来说，ISR代码大概就是用来读取设备状态，把状态储存以供后续处理，然后交还CPU给主程序。游戏引擎就可以在合适的时候提取那些数据。

8.2.3 无线设备

蓝牙（Bluetooth）设备，如Wii遥控器、DualShock 3和Xbox 360无线手柄，并不能简单地通过访问寄存器或内存映射I/O去读 / 写。软件必须以蓝牙协议（Bluetooth protocol）

和设备“交流”。软件可请求HID传送输入数据（如按钮状态）回主机，或传送输出（如震动设置或音频数据流）至设备。这种通信一般会由游戏引擎主线程以外的线程负责处理，或至少被封装为相对简单的接口供主循环调用。从游戏程序员的角度来说，蓝牙设备的状态，基本上和其他传统轮询设备的状态并无二致。

8.3 输入类型

虽然游戏用HID的硬件规格和布局设计繁多，但其提供的大部分输入都可归类为几个类型，以下逐一类型深入讨论。

8.3.1 数字式按钮

几乎每个HID都至少有几个**数字式按钮**（digital button）。这些按钮只有两个状态——**按着**或**释放**。游戏程序员常称按着的按钮为**向下**（down），释放的按钮为**向上**（up）[1]。

电子工程师会说电路中的开关（switch）有两种状态，其中**闭合**（closed）是指电流流通电路，而**断开**（open）是指电流不流通——即电路有无穷大的**电阻**（resistance）。**闭合**是表示按钮**按着**还是**释放**，取决于硬件设计。若按钮**正常是断开**的，那么释放时电路是断开的，按着时电路是**闭合**的。若按钮**正常是闭合的**，情况则相反——按下按钮会断开电路。

软件中，数字式按钮的状态（按着或没按着）通常以一个单独位表示，一般以0表示没按着（向上），1表示按着（向下）。但是此表示方式取决于电路设计，以及驱动程序员的决定，该值的意思也可以是相反的。

有时候，设备上所有按钮的状态会结合为一个无符号整数值。例如，在微软的XInput API中，XBox 360手柄的状态是以XINPUT_GAMEPAD结构传回的，该结构的定义如下：

```
typedef struct _XINPUT_GAMEPAD {
    WORD wButtons;
    BYTE bLeftTrigger;
    BYTE bRightTrigger;
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

[1]译注：键盘按键和鼠标按钮都是数字型按钮，许多API使用如OnKeyDown/Up、OnMouseDown/Up等命名，表示按钮状态改变的事件。但英文中up/down并非对按钮状态的描述，较正确的说法应该是pressed/released，故作者才特意指出。

此结构含16位无符号整数（WORD）变量wButtons，存放所有按钮的状态。以下的掩码定义了物理按钮在该字里所对应的位。（注意第10和11位是未用的。）

```
#define XINPUT_GAMEPAD_DPAD_UP          0x0001 // bit 0
#define XINPUT_GAMEPAD_DPAD_DOWN        0x0002 // bit 1
#define XINPUT_GAMEPAD_DPAD_LEFT        0x0004 // bit 2
#define XINPUT_GAMEPAD_DPAD_RIGHT       0x0008 // bit 3
#define XINPUT_GAMEPAD_START            0x0010 // bit 4
#define XINPUT_GAMEPAD_BACK             0x0020 // bit 5
#define XINPUT_GAMEPAD_LEFT_THUMB       0x0040 // bit 6
#define XINPUT_GAMEPAD_RIGHT_THUMB      0x0080 // bit 7
#define XINPUT_GAMEPAD_LEFT_SHOULDER    0x0100 // bit 8
#define XINPUT_GAMEPAD_RIGHT_SHOULDER   0x0200 // bit 9
#define XINPUT_GAMEPAD_A                0x1000 // bit 12
#define XINPUT_GAMEPAD_B                0x2000 // bit 13
#define XINPUT_GAMEPAD_X                0x4000 // bit 14
#define XINPUT_GAMEPAD_Y                0x8000 // bit 15
```

要读取某按钮的状态，可对wButtons字及该按钮的对应掩码进行C/C++的位并（&）运算，并检查运算结果是否为非零值。例如，要检测A按钮是否按着（向下），可以编写：

```
bool IsButtonADown(const XINPUT_GAMEPAD& pad)
{
    // 遮掩第12位以外的位
    return ((pad.wButtons & XINPUT_GAMEOAD_A) != 0);
}
```

8.3.2 模拟式轴及按钮

模拟式输入（analog input）是指可获取一个范围以内的数值（而非仅0和1）。此类输入通常用来代表扣压扳机的程度，或摇杆的二维位置（使用两个模拟输入，一个用作x轴，一个用作y轴，如图8.6所示）。由于模拟式输入经常用来代表某些轴的旋转角度，所以模拟式输入又称为**模拟式轴**（analog axis），或简称为**轴**（axis）。

有些设备的按钮也是模拟式的，这意味着游戏能检测玩家按下那些按钮的压强。然而，模拟式按钮所产生的信号通常有太多噪声（noise），导致这些按钮不太有用。笔者暂时还没遇见能有效运用模拟式按钮的游戏。（说不定实际上是有这样的游戏！）

严格来说，模拟式输入到达游戏引擎时并非是模拟的。每个模拟式输入信号通常都要被**数字化**（digitize），意指信号被量化（quantize），再被表示为软件中的整数。例如，某模拟式输入若使用16位整数表示，其范围可能是$-32,768 \sim 32,767$。有时候模拟式输入也会转

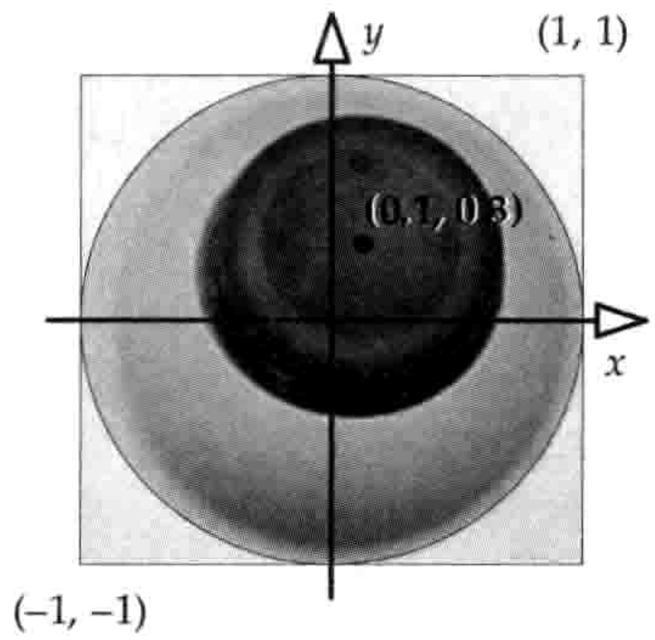


图 8.6: 可用两个模拟输入表示摇杆的x和y偏转量。

换为浮点数，例如−1～1范围的值。但3.2.1.3节曾提及，浮点数也只是数值化后的数字值。

回顾XINPUT_GAMEPAD的定义，会发现微软采用16位带符号整数表示Xbox 360手柄拇指摇杆的偏转量（左摇杆是sThumbLX/sThumbLY，右摇杆是sThumbRX/sThumbRY）。因此，这些值的范围为$-32,768$（向左或向下）至$32,767$（向右或向上）。但是，左右扳机却是以8位无符号整数表示的（bLeftTrigger和bRightTrigger）。这些输入的范围为0（没扣压）～255（完全扣压）。其他游戏机的模拟式轴会采用不同的数字表示法。

```
typedef struct _XINPUT_GAMEPAD {
    WORD wButtons;
    // 8位无符号
    BYTE bLeftTrigger;
    BYTE bRightTrigger;
    // 16位有符号
    SHORT sThumbLX;
    SHORT sThumbLY;
    SHORT sThumbRX;
    SHORT sThumbRY;
} XINPUT_GAMEPAD;
```

8.3.3 相对性轴

模拟式的按钮、扳机、摇杆和拇指摇杆的位置都是**绝对的**（absolute），即是说在某个明确定义的位置其输入值为0。然而，有些设备的输入是**相对性的**（relative）。这类设备并不能界定在哪个位置的输入值为0。反过来，输入值为0代表设备的位置没变动，而非零值代表自上次读取输入至今的增量（delta）。这样的例子有鼠标、鼠标滚轮和轨迹球。

8.3.4 加速计

PlayStation 3的Sixaxis及DualShock 3手柄，以及任天堂的Wii遥控器，都内含加速传感器（加速计/accelerometer）。这些设备能感应3个主轴（x, y, z）方向的加速，如图8.7所示。这些是**相对性**模拟式输入，就好像鼠标的二维轴。当控制器没被加速时，其输入为0；[2]当控制器被加速时，它就能量度每个轴上最高±3g的加速度，并把每轴的量度数值化为8位带符号整数。

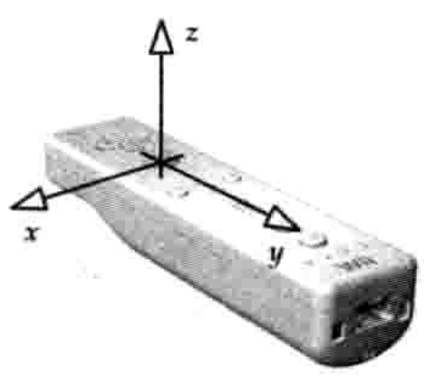


图 8.7: Wii摇控器的加速计轴。

8.3.5 以Wii遥控器或Sixaxis做三维定向

有些Wii和PS3游戏会分别使用Wii遥控器和Sixaxis手柄中的3个加速计去估算控制器在玩家手上的定向（orientation）。例如，在《超级玛里奥银河》的一些关卡中，玛里奥需跳上一个大球，并用脚去滚动该球。在此模式下，玩家的控制方式是首先把Wii遥控器指向天花板，再把Wii遥控器向前后左右倾斜来控制该球向相应的方向加速。

3个加速计可以用来检测Wii遥控器或Sixaxis手柄的定向，其原理是基于我们在地球表面上玩这些游戏，而地球的1g（$\approx 9.8\text{m/s}^2$）引力（gravity）能对物体产生固定的向下加速度。若把控制器完美地水平放置，并指向电视方向，那么垂直方向（z）的加速计应量度到大约-1g。

若垂直地握着控制器，使其指向上方，则可以预期z传感器测量到的加速度应为0g，而y传感器应为$+1$g （因为y传感器会感受到完整的引力效果）。若把手柄以45°握着，则x和y输入的值大约都会是$\sin 45° = \cos 45° = 0.707$g。当我们校准（calibrate）了加速计，得知每个轴的零点，就可以使用逆正弦和逆余弦，轻松求得偏航角（yaw）、俯仰角（pitch）和滚动角（roll）。

---

[2]译注：加速计量度的是固有加速度（proper accleration），即其感受到、相对于自由落体的加速度。在地球近地表的位置，自由落体的加速度是向地心1g（$g \approx 9.8\text{ms}^{-2}$）的。所以加速计在相对地球而言静止（如置于桌上）或均速直线移动时（如在笔直路轨上以均速前进的火车中），其量度到的值是向上1g；当其作为自由落体时（如使手柄从空中掉下），其量度到的值才是零矢量。但当然，操作系统或驱动可减去向上1g，使其传回相对静止状态的加速度量度。

但是这种做法有两个问题：第一，若玩家不能把控制器握稳，控制器往各个方向移动所产生的加速度也会被加速计侦测进去，使上述的数学计算变得无效。第二，由于z加速计已为引力校准，而其余两个没有，所以z轴对侦测定向有较少的精确度。许多Wii游戏需要玩家以非标准定向掌握Wii遥控器，例如，把按钮面向玩家的胸口，并把控制器指向天花。这样x或y加速计的轴可以和引力方向平行，而非已为引力校准的z加速计，从而能使定向读数的精度最大化。关于此题目的更多资料，可参考这两个网页[3,4]。

8.3.6 摄像机

Wii遥控器有一项其他游戏机标准HID欠缺的独特功能，这就是其红外线（infrared，IR）传感器。此传感器本质上是一个低分辨率摄像机，能捕捉Wii遥控器指向的二维红外线影像。Wii游戏机还附有一条需置于电视上的“Wii专用感应条[5]（Wii sensor bar）”，此设备两端各含一个红外线发光二极管（light emitting diode，LED）。在IR摄像机获取的影像中，这两个LED成为两个亮点，而其他背景则是黑暗的。Wii遥控器的影像处理软件会分析这一影像，并分离出该两点的尺寸及位置。（实际上，该软件能检测和传送多至4个亮点的位置和尺寸。）然后，游戏机就能通过蓝牙接收这些尺寸和位置信息。

把两个亮点连成直线后，该直线的位置和定向能用来计算Wii遥控器的偏航角、俯仰角和滚动角（只要Wii遥控器指向感应条）。根据两点的分隔距离，还可以计算出Wii遥控器和电视之间的距离。此外，有些软件也会利用亮点的尺寸信息。图8.8说明了Wii遥控器摄像机的基本原理。

另一个流行的摄像机设备，是索尼为PlayStation系列游戏机而设计的EyeToy（图8.9）。此设备基本上是一个有多种应用场合的高分辨率彩色摄像机。它可以进行视频会议，像普通的网络摄像机（web cam）一样。也可用作跟Wii摇控器的IR摄像机差不多的用途，去感应位置、定向和深度距离。现时游戏开发者们才刚开始尝试这些高级输入设备的各种可能性。

8.4 输出类型

HID通常用来把玩家的输入传送至游戏软件。然而，有些HID也可以通过各类型输出，向玩家反馈。

---

[3] http://druid.caughq.org/presentations/turbo/Wiimote-Hacking.pdf

[4] http://www.wiili.org/index.php/Motion_analysis

[5] 译注：此乃香港任天堂中文网站上所用的字眼。

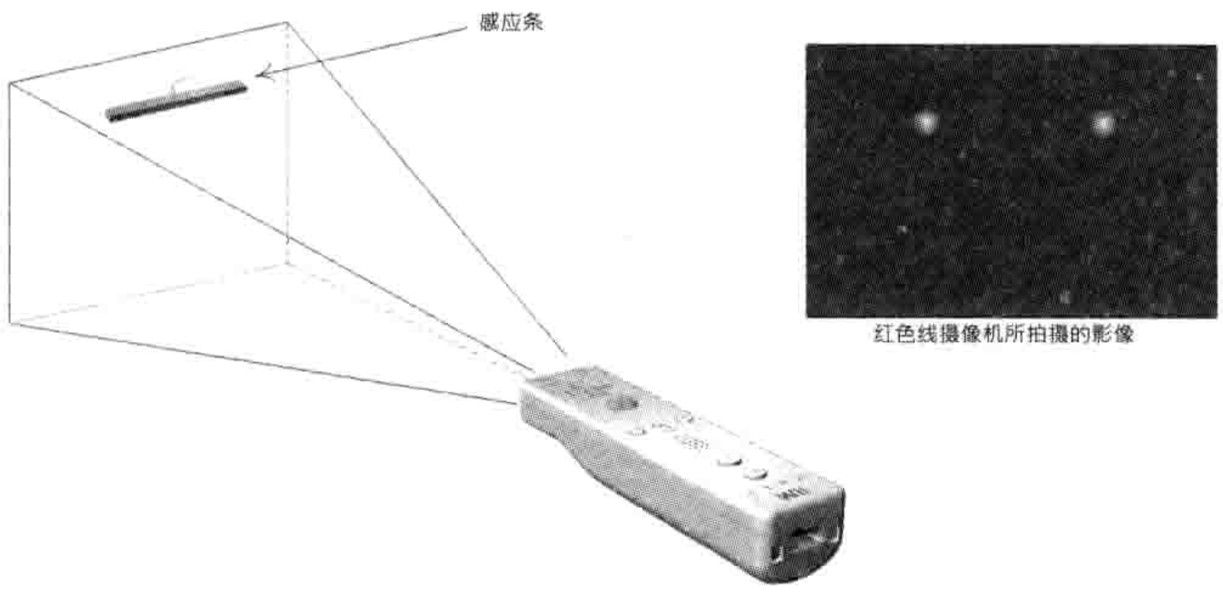


图 8.8: Wii专用感应条里装置了两个红外线LED，在Wii遥控器的红外线摄像机所拍摄的影像中产生了两个亮点。

图 8.9: 索尼供PS3使用的EyeToy。

8.4.1 震动反馈

一些手柄提供**震动反馈**（rumble）[6]功能，例如，PlayStation的DualShock系列手柄、Xbox及Xbox 360手柄。此功能让手柄在玩家手中震动（vibrate），以模拟游戏角色在游戏中受到扰动或撞击等感觉。震动通常由一个至多个马达（motor）驱动，每个马达带有稍不平衡的负重，以不同速度旋转。然后，游戏可开关这些马达，并通过调节其旋转速度来向玩家双手产生不同的触觉效果。

8.4.2 力反馈

力反馈（force-feedback）是另一种输出类型，其原理是通过由马达驱动的致动器（actuator），以其产生的力对抗玩家施于HID上的力。此功能常见于街机赛车游戏——当玩家尝试转方向盘时，方向盘会产生阻力，以模拟困难的行车条件或急转弯。如同震动反馈，游戏通常也可开关力反馈马达，并控制施于致动器的力和方向。

8.4.3 音频

音频（audio）通常是独立的引擎系统。可是，有些HID却能提供音频输出，供音频系统使用。例如，Wii遥控器就含有一个细小、低质量的扬声器（speaker）。而Xbox 360手柄也有耳机（headset）插口，能提供如USB音频设备的输出（扬声器）和输入（麦克风/microphone）功能。USB耳机的常用情境之一是，在多人游戏中让玩家通过网络语音

[6]译注：Rumble通常是指一种低频噪声，形容“隆隆”作响。游戏界中常以rumble一词指手柄的震动功能。此功能的正式术语为vibration feedback（震动反馈），是一种触觉技术（haptic technology）。

（voice over IP，VoIP）通信。

8.4.4 其他输入 / 输出

HID还可以支持许多其他类型的输入 / 输出。在一些较老一点的游戏机上，如世嘉Dreamcast，其手柄能插入记忆卡。而Xbox 360手柄、Sixaxis/DualShock 3手柄和Wii遥控器都分别带有4个LED，由软件控制开关。当然，一些特殊的设备，如乐器、跳舞毯等，有其专门的输入 / 输出类型。

人体学接口正在不断进行创新。当今最有趣的相关题目莫过于姿势界面（gesture interface）和思想控制设备（thought-controlled device）。我们肯定可以预见在几年之内，游戏机和HID制造商将带来更多创新。

8.5 游戏引擎的人体学接口设备系统

多数游戏引擎不会直接使用HID的原始输入数据。这些来自HID的输入数据通常会经过多重处理，确保数据能转化为游戏内又流畅、又直觉、又令人满意的行为。此外，大部分游戏引擎会引入至少一个在HID和游戏之间的间接层，以把HID输入以多种形式抽象化。例如，按钮映射表（button-mapping table）可用来把原始按钮输入转化为游戏逻辑的动作（action），那么玩家就能按喜好自定义按钮的功能。本节会先概述游戏引擎HID系统的典型需求，然后再深入探讨每个需求。

8.5.1 典型需求

游戏引擎的HID系统通常提供以下部分或全部功能。

- 死区（dead zone）。
- 模拟信号过滤（analog signal filtering）。
- 事件检测（event detection）（如按下和释放按钮）。
- 检测按钮的**序列**（sequence），以及多按钮的组合（又称为**弦**/chord）。
- 手势检测（gesture detection）。
- 为多位玩家管理多个HID。
- 多平台的HID支持。

- 控制器输入的重新映射（re-mapping）。
- 上下文相关输入（context sensitive input）。
- 临时禁用某些输入。

8.5.2 死区

模拟轴（如摇杆、拇指摇杆、扳机等）所产生的输入值都是介于一些预设的最小、最大范围内，以下使用$I_{\min}$和$I_{\max}$代表此范围。当玩家未触碰那些模拟轴时，我们希望能获得稳定及清晰的“未扰动（undisturbed）”输入值，此值以下称为I_0。通常未扰动值在数值上等于0，并且对于双向控制（如摇杆轴）来说，其值会位于$I_{\min}$和$I_{\max}$的正中间，而对单向控制（如扳机）来说，则会等于$I_{\min}$。

可惜，由于HID本质上是模拟式设备，其产生的电压含有噪声，以致实际上量度到的输入会轻微在I_0附近浮动。此问题的常见解决办法是引入一个围绕I_0的小**死区**（dead zone）。对于摇杆，死区可以定义为$[I_0 - \sigma, I_0 + \sigma]$；对于扳机，则定义为$[I_0, I_0 + \sigma]$。任何位于死区的输入值都可以简单地被钳制为$I_0$。死区必须足够大以容纳未扰动控制的最大噪声，同时死区也必须足够小以免影响玩家对HID的反应手感。

8.5.3 模拟信号过滤

就算控制器不在死区范围，其输入仍然会有信号噪声问题。这些噪声有时候会导致游戏中的行为显得抖动或不自然。因此，许多游戏会过滤来自HID的原始信号。噪声信号的频率通常比玩家产生的要高。所以，解决办法之一是，先利用**低通滤波器**（low pass filter）过滤原始输入数据，然后才把结果传送至游戏中使用。

离散低通滤波器的实现方法之一是，结合目前未过滤输入值和上一帧的已过滤输入。设未过滤输入为时变函数$u(t)$，并设已过滤输入为$f(t)$，当中t为时间，则它们的关系可写成：

$$f(t) = (1 - a)f(t - \Delta t) + au(t) \tag{8.1}$$

当中参数a是按帧持续时间Δt和过滤常数RC所确定（RC是传统以阻容电路实现的模拟低通滤波器中，电阻值和电容值的积）：

$$a = \frac{\Delta t}{RC + \Delta t} \tag{8.2}$$

这些等式可以简单地用以下的C/C++代码实现。此实现假设调用方保存了上一帧的已过滤输入。更多关于低通滤波器的信息可参阅维基百科[7]。

```
F32 lowPassFilter(F32 unfilteredInput,
                  F32 lastFramesFilteredInput,
                  F32 rc, F32 dt)
{
    F32 a = dt / (rc + dt);
    return (1 - a) * lastFramesFilteredInput + a * unfilteredInput;
}
```

另一个过滤HID输入数据的方法是计算移动平均（moving average）。例如，若要计算3/30s（3帧）时间范围内的输入数据平均，只需把原始输入数据简单地储存于3个元素大小的循环缓冲区里，把此数组的值求和除3，就是过滤后的输入值。实现此过滤器时还要留意一些细节。例如，需要正确地处理前两帧的输入，因为当时该数组并未填满有效数据。但实现并不特别复杂。以下代码示范计算N个元素的移动平均：

```
template< typename TYPE, int SIZE >
class MovingAverage
{
    TYPE        m_samples[SIZE];
    TYPE        m_sum;
    U32         m_curSample;
    U32         m_sampleCount;

public:
    MovingAverage() :
        m_sum(static_cast<TYPE>(0)),
        m_curSample(0),
        m_sampleCount(0)
    {
    }

    void addSample(TYPE data)
    {
        if (m_sampleCount == SIZE)
        {
            m_sum -= m_samples[m_curSample];
        }
        else
        {
```

[7] http://en.wikipedia.org/wiki/Low-pass_filter

```
            ++m_sampleCOunt;
        }

        m_samples[m_curSample] = data;
        m_sum += data;
        ++m_curSample;

        if (m_curSample >= SIZE)
        {
            m_curSample = 0;
        }
    }

    F32 getCurrentAverage() const
    {
        if (m_sampleCount != 0)
        {
            return static_cast<F32>(m_sum)
                / static_cast<F32>(m_sampleCount);
        }
        return 0.0f;
    }
};
```

8.5.4 输入事件检测

低级的HID接口通常会为游戏提供设备中各输入的当前状态信息。然而在许多时候，游戏需要检测**事件**（event）——即状态之改变，而非每帧的当前状态。最常见的HID事件大概就是按下和释放按钮，但当然，也可检测其他种类的事件。

8.5.4.1 按下和释放按钮

假设按钮[8]的输入位在没按下时为0，按下时为1。检测按钮状态改变的最简单方法就是，记录上一帧的状态，用以和本帧的状态比较。若两状态有所不同，便能得知有事件发生。接着，凭每个按钮的当前状态，就能知道是按下或释放按钮事件。

此过程可以使用简单的位运算符（bitwise operator）来检测按下和释放按钮事件。假设有一个32位字buttonStates，内含有最多32个按钮的当前状态，我们希望计算出两个32位字——buttonDowns含按下按钮的事件，buttonUps含释放按钮的事件。在这两

[8]译注：此处的按钮，除了指手柄和鼠标的按钮，也可指键盘上的按键和其他相似的输入。

个字中，0代表事件没发生，1代表事件已发生。要进行此计算，还需要上一帧的按钮状态prevButtonStates。

我们都知道，异或（exclusive OR，XOR）运算对两个相同的输入会产生0，对两个不同的输入会产生1。因此，若把上一帧和本帧的按钮状态字进行位异或，某个位产生1代表该按钮的状态在两帧中有所改变。而要测定该事件是按下或释放按钮，可以再审视每个按钮的当前状态。若某按钮的状态有改变，而当前的状态是按下，那么就产生按下事件，否则就是个释放事件。以下代码使用了以上的想法来产生两个按钮事件：

```
class ButtonState
{
U32 m_buttonStates;      // 当前帧的按钮状态
U32 m_prevButtonStates; // 上一帧的按钮状态
U32 m_buttonDowns;       // 1 表示本帧按下的按钮
U32 m_buttonUps;         // 1 表示本帧释放的按钮

void DetectButtonUpDownEvents()
{
    // 假设m_buttonStates及m_prevButtonStates都是有效的，
    // 生成m_buttonDowns及m_buttonUps

    // 首先判断哪些位有改变
    U32 buttonChanges = m_buttonStates ^ m_prevButtonStates;

    // 然后用AND去取得DOWN的各个位
    m_buttonDowns = buttonChanges & m_buttonStates;

    // 再用AND-NOT去取得UP的各个位
    m_buttonUps = buttonChanges & (~m_buttonStates);
    }

    // ……
};
```

8.5.4.2 弦

弦（chord）是指一组按钮，当同时被按下时，会产生在游戏中另一个独特行为。以下是一些弦的例子。

- 在《超级玛里奥银河》的开始画面中要求玩家同时按下Wii遥控器上的A和B按钮来开始游戏。

- 无论在玩任何Wii游戏，同时按下Wii遥控器的1和2按钮，就会使该遥控器进入蓝牙发现模式。
- 许多格斗游戏中，同时按下某两个按钮是用来使出捕捉技（grapple）。
- 在《神秘海域：德雷克船长的宝藏》的开发版本中，同时按下DualShock 3手柄的左右扳机，能使玩家角色飞越游戏世界的任何地方，不受碰撞。（抱歉，此功能在发行版本无效!）许多游戏也有类似的秘技（cheat）以方便开发。（那些秘技也可以不使用弦。）在雷神之锤引擎中，此秘技称为**穿墙模式**（no-clip mode），意指角色的碰撞体不**受限**于世界的可玩地域。其他引擎使用另一些术语。

弦的检测在理论上颇简单——监察两个或以上的按钮状态，当该组按钮全部同时被按下，才执行操作。

但是，当中还要处理许多细节。首先，若弦里的按钮在游戏中有其他用途，便要小心地避免**同时**产生个别按钮的动作和弦的动作。通常可以这样解决：检测个别按钮的时候，同时检查弦里的其他按键并**没有**被按下。

另一个美中不足的地方，在于人类总非完美，人们常常会按下弦中的某一个按钮稍早于其他按钮。因此弦的检测代码必须更健壮，处理玩家可能在第i帧按下一个或数个按钮，而在第$n+1$帧（甚至更多帧之后）按下弦的其他按钮。有几种方法可以处理这些情况。

- 可以把按钮输入设计为，弦总是作用于某个按钮的动作再**加上**额外的动作。例如，若按L1是令主要武器开火，按L2投射手榴弹，可能L1 + L2的弦是令主要武器开火、投射手榴弹，**并**发送能量波使这些武器的伤害力加倍。那么，就算个别按钮于弦之前被检测，从玩家的角度来说游戏表现出的行为没有不同。
- 可以在个别按钮按下之后，加入一段延迟时间，然后才算作是一个有效的游戏事件。在延迟期间（如2或3帧），若检测到一个弦，那么那个弦就会凌驾个别按钮按下事件。这种方法给玩家按弦时留有余地。
- 可以在按下按钮时检测弦，但当之后释放按钮时才产生效果。
- 可以在按下单个按钮时立即执行其动作，但容许这些动作被之后弦的动作所抢占。

8.5.4.3 序列和手势检测

在实际按下按钮后，程序延迟一段时间才把它算作一个按下事件，此过程乃**手势检测**（gesture detection）的特例。手势是指玩家通过HID，在一段时间内完成一串动作。例如，在格斗游戏或动作游戏中，可能需要检测按钮**序列**（sequence），例如“ABA”。我们也可以扩大此机制至按钮以外的输入。例如“ABA左右左”，当中最后3个动作是指手柄其中一个

摇杆从左到右再到左的移动。通常序列或手势需要在限定时间内完成，否则当作无效。所以可能在1/4s内按“ABA”是算有效的，超过1或2s则为无效。

一般实现手势检测的方法是，保留玩家通过HID输入的动作短期记录。当检测到手势中第一个成分，就会把该成分及其产生的时间戳记录在历史缓冲区中。之后，检测到每个后续成分时，需要检查距上一个成分所经过的时间，若时间仍在容许范围内，就继续把该成分加入历史缓冲区中。若整个序列于限定时间内完成，就会产生对应的手势事件，以通知游戏引擎的其余部分。然而，若在过程中检测到无效输入，或手势成分于允许时间外产生，那么整个历史缓冲区会被重置，玩家需要重新输入手势。

以下通过3个具体例子，使读者能确切领会这些机制如何运作。

迅速连打按钮

许多游戏要求玩家迅速连打按钮以执行某些动作。连打按钮的频率有时候会转化为游戏内某些数值，例如玩家角色的跑步速度或其他动作。此频率通常也会用来定义手势判定是否成立——当频率降低至某个最小值，手势判定就不再成立。

要侦测连打按钮频率，只须记录该按钮上一次被按下事件的时间T_{last}。那么，频率f就是两次按下按钮的时间间隔的倒数（$\Delta T = T_{\text{cur}} - T_{\text{last}}$，$f = 1/\Delta T$）。每次侦测到新的按下按钮事件，都要计算新的频率$f$。要实现最小有效频率，只需要比较$f$和最小频率$f_{\min}$（另一个方法是直接比较$\Delta T$和最大周期$\Delta T_{\max} = 1/f_{\min}$）。若结果合乎阈值，便更新$T_{\text{last}}$的值。若结果不合乎阈值，只要不更新$T_{\text{last}}$便可。那么，在有一对新的够迅速的按钮按下事件产生之前，手势会一直判定为无效。以下的伪代码展示了此过程：

```
class ButtonTapDetector
{
    U32     m_buttonMask;    // 需检测的按钮 (位掩码)
    F32     m_dtMax;         // 按下事件之间的最长容许时限
    F32     m_tLast;         // 最后按下按钮的时间，以秒为单位

public:
    // 构建一个对象，用于检测快速连打指定的按钮(以索引标识)
    ButtonTapDetector(U32 buttonId, F32 dtMax) :
        m_buttonMask(1U << buttonID),
        m_dtMax(dtMax),
        m_tLast(CurrentTime() - dtMax) // 开始时是无效的
    {
    }
```

```
    // 随时调用此函数查询玩家是否做出这个手势
    void IsGestureValid() const
    {
        F32 t = CurrentTime();
        F32 dt = t - m_tLast;
        return (dt < m_dtMax);
    }

    // 每帧调用此函数
    void Update()
    {
        if (ButtonJustWentDown(m_buttonMask))
        {
            m_tLast = CurrentTime();
        }
    }
};
```

在上面的代码片段中，每个按钮有唯一的标识符。此标识符仅仅是一个索引，范围是0～$N-1$（N为该HID的按钮总数）。把1向左移动标识符的位数（1U << buttonId），就能把标识符转变为位掩码。ButtonsJustWentDown()函数用来侦测本帧刚被按下的按钮，若位掩码指定的按钮中有**任意一个**按钮刚才按下，此函数就会回传非零值。在此类中，我们只是利用ButtonsJustWentDown()函数来检查某个按钮的按下事件，但此函数之后会用作检查多个同时按下按钮事件。

多按钮序列

假设我们想检测在1s内连续按下ABA的序列，其做法如下。首先，我们使用一变量记录在序列中预期要按下的按钮。例如，如果使用按钮标识符数组来定义序列（如aButtons[3]={A, B, A}），那么该变量就是此数组的索引i。该变量最初设置为序列的第一个按钮，即$i=0$。此外，如同前文中迅速连打按钮的例子，我们也需要另一变量T_{start}，记录整个序列的开始时间。

之后的做法就是，当接收到一个合乎序列目前预期的按钮按下事件，就把事件的时间戳与序列开始时间T_{start}进行比较。若事件仍在有效时间窗（time window）之内，便要把目前的按钮移至序列下一按钮；仅当处理第一个按钮（$i=0$）时，还须更新T_{start}。若按钮按下事件不合乎序列的目前按钮，或是时间差太大，就要把按钮索引重置为序列开端，并把T_{start}设为某个无效值（例如0）。以下代码演示了这个逻辑。

```
class ButtonSequenceDetector
{
    U32*     m_aButtonIds;    // 检测的序列
    U32      m_buttonCount;   // 序列中的按钮数目
    F32      m_dtMax;         // 整个序列的最大时限
    EventId  m_eventId;       // 完成序列的事件
    U32      m_iButton;       // 要检测的下一个按钮
    F32      m_tStart;        // 序列的开始时间，以秒为单位

public:
    // 构建一个对象，用于检测指定的按钮序列
    // 当成功检测到序列，就会广播指定事件，令整个游戏能适当回应事件
    ButtonSequenceDetector(U32* aButtonIds,
                           U32 buttonCount,
                           F32 dtMax,
                           EventId eventIdToSend) :
        m_aButtonIds(aButtonIds),
        m_buttonCount(buttonCount),
        m_dtMax(dtMax),
        m_eventId(eventIdToSend),    // 完成序列后会广播的事件
        m_iButton(0),                // 序列之始
        m_tStart(0)                  // 初始值(无作用)
{
}

// 每帧调用此函数
void Update()
{
    ASSERT(m_iButton < m_buttonCount);

    // 计算下个预期的按钮，以位掩码表示(把1左移至正确的位索引)
    U32 buttonMask = (1U << m_aButtonId[m_iButton]);

    // 若玩家按下预期以外的按钮，废止现时的序列
    // (使用位取反运算检测所有其他按钮。)
    if (ButtonsJustWentDown(~buttonMask))
    {
        m_iButton = 0;   // 重置
    }
    // 否则，若预期按钮刚被按下，检查dt及适当地更新状态
    else if (ButtonsJustWentDown(buttonMask))
    {
        if (m_iButton == 0)
        {
            // 此乃序列中第一个按钮
```

```
            m_tStart = CurrentTime();
            ++m_iButton;            // 移至下一个按钮
        }
        else
        {
            F32 dt = CurrentTime() - m_tStart;

            if (dt < m_dtMax)
            {
                // 序列仍然有效。
                ++m_iButton;    // 移至下一个按钮

                // 序列是否完成?
                if (m_iButton == m_buttonCount)
                {
                    BroadcastEvent(m_eventId);
                    m_iButton = 0;  // 重置
                }
            }
            else
            {
                // 对不起，按得不够快
                m_iButton = 0;  // 重置
            }
        }
    }
};
```

旋转摇杆

我们再看一个更复杂的手势例子，检测玩家把左拇指摇杆沿顺时针方向旋转一周。这种检测颇为容易，如图8.10所示，把遥杆位置的二维范围分割成4个象限（quadrant）。顺时针方向旋转时，其中一个情况是摇杆先经过左上象限，然后至右上象限，再到右下象限，最后到左下象限。只要把象限检测当作按钮处理，就可稍修改上文按钮序列检测代码来完成任务。笔者把此实现留给读者作为练习。要尝试啊!

8.5.5 多HID和多玩家的管理

在玩多人游戏时，多数游戏机器容许接上两个或更多个HID。引擎需要追踪目前连接了哪些设备，并把每个设备的输入发送给游戏中适当的玩家。这意味着，我们需要某方式映射

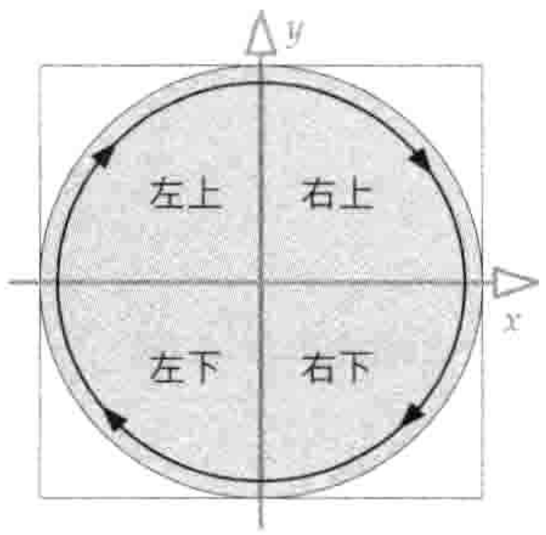


图 8.10: 把二维的摇杆输入分割为4个象限，借以检测循环旋转。

控制器至玩家。这也许简单到只是控制器索引至玩家索引的一一映射，也许是更复杂的，例如把控制器映射至按下Start按钮的玩家。

即使在单人游戏、仅有一个HID的情况下，引擎也需要稳健地处理多种异常情形，例如意外地拔掉控制器，或控制器电池耗尽。当控制器断线，多数游戏会暂停进度，显示信息，然后等待控制器重新连接。大部分多人游戏中，对应断线控制器的化身会被暂停或临时移除，但容许其他玩家继续玩；当重新连接控制器之后就会再激活该化身。

若系统中有使用电池运作的HID，游戏或操作系统便需要负责检测低电力状况。出现这种状况时，通常要用某些方法通知玩家，例如显示一个不会被游戏内容遮挡的信息，及/或播放一个音效。

8.5.6 跨平台HID系统

许多游戏引擎是跨平台的。这些引擎中，处理HID输入及输出的方法之一，就是如下面的例子，在所有和HID相关的代码中散布条件编译指令。显然这是可行的，但非理想方案。

```
#if TARGET_XBOX360
    if (ButtonsJustWentDown(XB360_BUTTONMASK_A))
#elif TARGET_PS3
    if (ButtonsJustWentDown(PS3_BUTTONMASK_TRIANGLE))
#elif TARGET_WII
    if (ButtonsJustWentDown(WII_BUTTONMASK_A))
#endif
{
    // 做些事情……
}
```

更好的方案是提供某形式的硬件抽象层，使游戏代码和硬件相关细节隔离。

若运气佳，我们可以把不同平台HID的大部分差异，通过细心选择的抽象按钮及轴抽象出来。例如，若我们的游戏发行的是Xbox 360及PS3版本，两款手柄的控制布局（按钮、轴及扳机）几近相同。每个平台上，控制的标识符有差异，但我们可以轻易地采用泛化的控制标识符以供两种手柄使用。例如：

```
enum AbstractControlIndex
{
    // 开始及返回按钮
    AINDEX_START,              // Xbox 360 Start, PS3 Start
    AINDEX_BACK_PAUSE,         // Xbox 360 Back, PS3 Pause

    // 左方十字按钮
    AINDEX_LPAD_DOWN,
    AINDEX_LPAD_UP,
    AINDEX_LPAD_LEFT,
    AINDEX_LPAD_RIGHT,

    // 右方4个按钮
    AINDEX_RPAD_DOWN,          // Xbox 360 A, PS3 交叉
    AINDEX_RPAD_UP,            // Xbox 360 Y, PS3 三角
    AINDEX_RPAD_LEFT,          // Xbox 360 X, PS3 正方
    AINDEX_RPAD_RIGHT,         // Xbox 360 B, PS3 圆形

    // 左右拇指遥杆按钮
    AINDEX_LSTICK_BUTTON,      // Xbox 360 LThumb, PS3 L3, Xbox 白
    AINDEX_RSTICK_BUTTON,      // Xbox 360 RThumb, PS3 R3, Xbox 黑

    // 左右肩按钮
    AINDEX_LSHOULDER,          // Xbox 360 L肩, PS3 L1
    AINDEX_RSHOULDER,          // Xbox 360 R肩, PS3 R1

    // 左拇指遥杆轴
    AINDEX_LSTICK_X,
    AINDEX_LSTICK_Y,

    // 右拇指遥杆轴
    AINDEX_RSTICK_X,
    AINDEX_RSTICK_Y,

    // 左右板机轴
    AINDEX_LTRIGGER,           // Xbox 360 -Z, PS3 L2
    AINDEX_RTRIGGER,           // Xbox 360 +Z, PS3 R2
};
```

此抽象层能把目标硬件的原始控制标识符转化为抽象的控制索引。例如，每当把按钮状态读入成为32位字，便可使用位重组（bit swizzling）指令按抽象索引次序重新排列。而模拟输入也可以同样地按适当次序重新排列。

映射物理到抽象控制的时候，有时还需一些小聪明。例如，在Xbox上，左右扳机合成单一轴，扣上左扳机时该轴产生负值，两个扳机都不扣时是0，而扣上右扳机则是正值。但为了匹配PlayStation的DualShock控制器，我们可能想要把那个Xbox上的轴分割成两个独立的轴，并把其值适当地缩放，使有效值的范围在所有平台上统一。

在多平台引擎上处理HID输入／输出，以上所说的当然不是唯一方法。我们可以采取更功能性的方式，例如把抽象控制按照它们在游戏中的功能来命名，而非使用手柄上的物理位置。我们也可以加入更高级的函数，使用平台定制的检测代码检测抽象手势；或是我们可以咬紧牙关，在所有需要HID输入／输出的游戏代码中编写各个平台的版本。虽然有无穷的可行做法，但是几乎所有跨平台游戏引擎都会在游戏代码和硬件细节之间加入**某种**隔离方式。

8.5.7 输入的重新映射

在物理HID的控制功能上，很多游戏提供玩家某程度的选择权。在视频游戏中有一常见选项，就是决定右拇指摇杆的垂直轴对于摄像机控制的意义。有些玩家喜欢向上推摇杆时令摄像机往上转，而其他玩家则喜爱倒转的控制方式，即向下推摇杆令摄像机往上转（像飞机的操控杆）。此外，有些游戏提供两套或以上的预定义按钮映射，供玩家选择。而有些计算机游戏则让玩家以完全的控制权设置键盘的每个键、鼠标的每个按钮和滚轮的功能，而鼠标的两个轴也可以有不同的控制方式。

实现这些功能的方法，可参考我的滑铁卢大学老教授Jay Black的一句名言：“计算机科学中的每个问题都可以用一间接层解决。[9]”我们可以给每个游戏功能一个唯一标识符，然后加一个简单的表，把每个物理或抽象的控制索引映射至游戏中的逻辑功能。每当游戏要判断是否应激活某个逻辑游戏功能，就可以查表找到对应的抽象或物理控制标识符，再而读取该控制的状态。要改变映射，可以更换整个表，或是让玩家设置该表中的个别条目。

在此我们再探讨一些技术细节。首先，各个控制产生有不同的输入种类。模拟轴所产生的值，其范围可能是$-32,768 \sim 32,767$，或是$0 \sim 255$，又或是其他范围。而HID上的数字式按钮的状态，通常会打包为单个计算机字。因此，我们必须小心，只容许合理的输入映射。例如，某个游戏逻辑需要轴，就不能改用按钮操控。解决方法之一就是把所有输入归一化。例如，可以把所有模拟轴和按钮的输入都重新缩放为$[0, 1]$范围。读者起初可能觉得这么做的

[9]译注：此句子应沿自David John Wheeler。见http://en.wikipedia.org/wiki/David_Wheeler_(computer_scientist)。

用途不大，因为有些轴本质是双向的（如摇杆），有些轴则是单向的（如扳机）。然而，只要把控制分为几类，就能对这些同类的输入进行归一化，并只容许相容的类型做重新映射。标准游戏机手柄的合理分类，以及其归一化输入可以如下。

- **数字式按钮**：按钮状态打包成32位字，每一位代表一个按钮的状态。
- **单向绝对轴**（如扳机、模拟式按钮）：产生$[0,1]$范围的浮点数输入值。
- **双向绝对轴**（如摇杆）：产生$[-1,1]$范围的浮点数输入值。
- **相对轴**（如鼠标轴、滚轮、轨迹球）：产生$[-1,1]$范围的浮点数输入值，当中± 1代表单帧（即1/30s或1/60s）内最大的相对偏移值。

8.5.8 上下文相关控制

在许多游戏里，一个物理控制会根据上下文（context）有着不同功能。例子之一就是无处不在的“使用”按钮。若游戏角色站在门前，按“使用”按钮可能会令角色开门。若游戏角色附近有一个物体，按“使用”按钮可能会令角色拾起该物体。另一个常见例子是模态（modal）控制模式。当玩家走动时，一些控制是用来导航和操控摄像机的。当玩家驾驶载具时，那些控制就会用来操控载具的转向，而摄像机的操控方式也可能不同。

上下文相关（context-senstive）控制可简单地采用状态机来实现。根据当前状态，个别HID控制可能有不同用途。而最棘手的部分，就是要判断现时在哪个状态中。例如，当按下上下文相关的“使用”按钮时，角色可能站立的位置，刚好与一件武器和一个医疗包距离相等，并面向着两物体的中间点。那么，应拾起哪一个物体呢？有些游戏实现了优先系统，以打破这种不分胜负的情况。或许是武器的权值高于医疗包，那么此例子中武器就会“胜出”。实现上下文控制并不复杂，但是必须反覆多尝试，才会感到有良好的手感。规划多次迭代和焦点测试吧！

另一个相关概念是**控制拥有权**（control ownership）。有些HID上的控制可能由游戏中不同部分所“拥有”。例如，有些输入是玩家角色控制，有些是摄像机控制，另有一些是供游戏的包装和菜单系统使用（暂停游戏等）。有些引擎引入逻辑设备的概念，这种设备只是由物理设备上的输入子集所组成的。例如，一个逻辑设备可能供一个玩家角色使用，另一个供摄像机使用，而另一个供菜单系统使用。

8.5.9 禁用输入

在多数游戏中，有时候需要禁止玩家控制其角色。例如，当玩家角色参与游戏内置电影时，我们可能希望暂停所有玩家操控；或是当玩家经过一条窄巷，我们可能希望暂停自由地

转动摄像机。

一个较拙劣的方法是，使用位掩码禁用设备上的个别控制。每当读取控制时，检查该掩码中对应的位，若该位被设置则传回零值或中性值，否则就传回实际从设备获得的值。然而，禁用设备时须特别谨慎，若忘记重置禁用掩码，游戏可能会进入一个状态——玩家持续失去对游戏的控制，并必须要重启游戏。因此我们须小心检查游戏逻辑，加入一些防故障机制也是一个好主意，例如在玩家角色死亡及重生时把禁用掩码清零。

直接禁用HID某些输入，对游戏来说可能是过大的限制。另一个可能更好的做法是，把禁用某玩家动作及行为的逻辑写进玩家或摄像机的代码里。例如，若摄像机某时刻决定要忽略右拇指轴的输入，游戏引擎内其他系统仍然有自由读取该输入做其他用途。

8.6 人体学接口设备使用实践

正确及流畅地处理人体学接口设备，是任何好游戏的重要一环。概念上，HID好像是颇为直接了当的事情。然而，实际上或会遇上一些“疑难杂症（gotcha）”，包括不同物理输入设备的差异、低通过滤器的正确实现、无缺陷的控制方式映射处理、理想的震动反馈手感、游戏机厂商的技术要求清单（technical requirements checklist，TRC）所引申的限制等。游戏开发团队应投放足够的时间和人力，实现一个又谨慎又完整的人体学接口设备系统。这是极其重要的，因为HID系统支撑着游戏的最宝贵资源——游戏的玩家机制。

第9章　调试及开发工具

开发游戏软件是一项错综复杂、数学密集、容易出错的工作。因此，几乎所有专业游戏团队都会制作一套工具自用，使游戏开发过程更容易、更少出错。在本章中，我们会看一看，专业级游戏引擎中最常见的开发及调试工具。

9.1　日志及跟踪

你可否记起，曾经用BASIC或Pascal编写第一个程序？（好吧，或许你不曾用过。若你比我年轻许多，很有可能没用过这些古老的编程语言——或许你是用Java、Python或Lua写第一个程序吧。）无论如何，你应该记得当时如何调试程序。用**调试器**（debugger）？或许当时你还以为debugger是那些泛出蓝光的除虫器呢。那时候，你大概更会使用打印语句（print statement）去显示程序的内部状态。C/C++程序员称此为**printf调试法**（此名沿于标准C程序库的`printf()`函数）。

即使你已知道debugger不是指那些用来在晚上烧烤倒霉昆虫的设备，事实证明，**printf调试法**仍是非常有效的方法。尤其在编写实时程序时，某些bug难以使用断点和监视窗口来跟踪。有些bug是有时间依赖性的，仅当程序在全速运行时才会出现。另一些bug由很大一串复杂事件导致，即使逐一手动跟踪也十分困难。在这种情况下，最强大的调试工具通常就是一组打印语句。

各个游戏平台都有某种主控台（console）或电传打字机（teletype，TTY）输出设备。以下是一些例子。

- 在Linux或Win32下运行由C/C++编写的主控台应用程序，可以使用`printf()`、`fprintf()`或STL的`iostream`接口，往`stdout`打印。
- 可惜，若游戏生成为Win32下的窗口应用程序，`printf()`和`iostream`就不能工作了，因为那时候并没有主控台供显示输出。然而，若在Visual Studio调试器之下运行

程序，就能使用Win32函数OutputDebugString()，向Visual Studio的调试主控台（debug console）打印信息。

- 在PlayStation 3开发套件中，有一个名为Target Manager的应用程序运行于PC端，该程序可以用来起动开发机端的程序。Target Manager含有一组TTY窗口，供游戏引擎打印消息。

因此，为调试而打印信息，通常都如在代码中加入一些printf()调用般简单。然而，多数游戏引擎会更进一步，提供更完善的打印功能。以下几个小节将会考察这些功能。

9.1.1 使用OutputDebugString()做格式化输出

Win32的OutputDebugString()函数能有效地把调试信息打印至Visual Studio的调试窗口。然而，与printf()不同，OutputDebugString()不支持格式化输出，它只能打印char数组形式的字符串。因此，多数Windows游戏引擎以自定义函数包装此函数：

```
#include <stdio.h>  // 为了 va_list 等声明
#ifndef WIN32_LEAN_AND_MEAN
#define WIN32_LEAN_AND_MEAN 1
#endif
#include <windows.h>    // 为了 OutputDebugString()

int VDebugPrintF(const char* format, va_list argList)
{
    const U32 MAX_CHARS = 1023;
    static char s_buffer[MAX_CHARS + 1];
    int charsWritten
        = vsnprintf(s_buffer, MAX_CHARS, format, argList);
    s_buffer[MAX_CHARS] = '\0';      // 确保字符串是以空字符结尾

    OutputDebugString(s_buffer);  // 得到格式化字符串后调用Win32 API
    return charsWritten;
}

int DebugPrintF(const char* format, ...)
{
    va_list argList;
    va_start(argList, format);
    int charsWritten = VDebugPrintF(format, argList);
    va_end(argList);
    return charsWritten;
}
```

注意上述代码实现了两个函数：DebugPrintF()接受用可变长度参数表（用省略号指明），而VDebugPrintF()则接受va_list参数。那么程序员可基于VDebugPrintF()来编写其他打印函数。（C/C++不能把省略号的内容由一个函数传递至另一个函数，但传递va_list就没问题。）

9.1.2 冗长级别

当你成功在代码中策略地加入一堆打印语句，最好能保留这些语句，做日后需要时之用。为此，多数引擎会提供一些机制来控制**冗长**级别（verbosity level），例如通过命令行或在运行时动态设定。当冗长级别设为最小值时（通常为0），只有严重错误消息才会被打印。若把冗长级别提高，则代码中更多的打印语句会做输出。

实现冗长级别的最简单方法就是，把当前的冗长级别存储在一个全局整数变量中，或可命名为g_verbosity。然后要提供一个VerboseDebugPrintF()函数，其首个参数为冗长级别，若当前冗长级别高于此参数就会打印该消息。此函数可这样实现：

```
int g_verbosity = 0;

void VerboseDebugPrintF(int verbosity, const char* format, ...)
{
    if (g_verbosity >= verbosity)  // 仅当全局冗长级别足够高才打印
    {
        va_list argList;
        va_start(argList, format);
        VDebugPrintF(format, argList);
        va_end(argList);
    }
}
```

9.1.3 频道

把调试输出分类为**频道**（channel）是另一个极为有用的功能。如某频道接收动画系统的消息，而另一频道接收物理系统的消息等。

有些游戏平台，如PlayStation 3，可以把调试输出送到14个不同TTY窗口之一。此外，每条消息还会抄送至一个特别的TTY窗口，使其包含其他14个窗口的所有输出。那么开发人员就能很容易地重点查看某一类消息。如要追查动画问题，就可以切换至动画的TTY窗口，忽略其他输出。而当出现一些未知原因的问题时，就可以在那个“全部”TTY中寻找线索。

其他平台如Windows仅提供单个调试输出主控台。然而，就算是在这些系统上，把输出划分为频道也是很有用的。每个频道的输出可以用不同颜色显示。另外，也可以实现**过滤器**（filter），这些过滤器能在运行时开关，并可设定仅某条或某组频道能输出。使用这种功能的话，若开发人员要追查动画问题，便可轻易把除了动画以外的频道过滤掉。

只需在调试打印函数里加入频道参数就可实现这种功能。频道可以用数字表示，但更好的做法是使用C/C++的enum声明来命名。另一做法是以字符串或字符串散列标识符（hashed string id）来命名频道。打印函数便可以根据当前作用的频道表，仅当指定的频道包含于该表才打印该消息。

若频道总数少于32或64个，就可以使用32位或64位掩码来指明要过滤的频道。当掩码中的某位为1，对应的频度便是开启的，否则该频度就是关闭的。

9.1.4 把输出同时抄写至日志文件

把所有调试信息同时抄写至一个或多个日志文件（如每个频道各用一个文件），是个不错的主意，因为这样可以在事后诊断问题所在。要是可行的话，不管当前的冗长级别及频道过滤设置为何，最好把**所有**调试输出都写进日志文件。那么遇到**预料之外**的问题，就能轻松地通过查核最近的日志文件以追查问题来源。

另一个酌量之事在于，是否每次调用调试输出函数后都对日志文件**清空缓冲**（flush），以确保万一游戏崩溃时日志文件仍会包含最后的输出。最后一笔打印出来的数据通常对确定崩溃原因起到关键作用，所以我们希望确保日志文件一直都含有最新的输出。当然，清空输出缓冲可能需要很高的成本。因此，仅当以下两种情形才应该清空输出缓冲：（a）程序输出的日志量不多，（b）你发现在某平台上确有必要这样做。若认为确有这种需求，可以在引擎配置提供清空缓冲的开关选项。

9.1.5 崩溃报告

有些游戏引擎会在崩溃时放出特别的文本输出或日志文件。多数操作系统都可以设置一个顶层异常处理函数（top-level exception handler），此函数能捕获大部分的崩溃情形。你可以在此函数中打印各种各样有用的信息，甚至可以考虑把崩溃报告以电邮形式送交整个程序团队。这样做对程序员有很好的启发性：当他们知道美术和设计团队经常遇到崩溃，就会意识到自己的调试工作到底有多迫切!

崩溃报告可包含各种信息，例如：

- 崩溃时玩家在玩的关卡。
- 崩溃时玩家角色所在的世界空间位置。
- 游戏崩溃时玩家角色的动画/动作状态。
- 崩溃时正在运行的一个或多个游戏脚本（当脚本是崩溃起因，此信息有莫大帮助）。
- 堆栈跟踪（stack trace）：多数操作系统都提供一些机制去取得调用堆栈（虽然那些机制没有标准而且跟平台非常相关）。通过这些机制，你可以在崩溃时，把堆栈中所有非内联函数的符号打印出来。
- 引擎中所有内存分配器的状态（余下内存的大小、内存碎片程度等）。若崩溃和造成内存不足的bug有关，这类数据就可能很有用。
- 你在查找崩溃原因时，想到的和崩溃有可能相关的其他信息。

9.2 调试用的绘图功能

现时的互动游戏几乎全由数学驱动。在游戏世界里，用数学来定位定向物体、移动它们、检测碰撞、投射光线（cast ray）以检测视线（line of sight），当然少不了要用矩阵乘法来转换物体坐标，把坐标从物体空间转换至世界空间，或再转换至屏幕空间做渲染之用。几乎所有现时的游戏都是三维的，但即使是二维的游戏，也难以把所有这些数学计算结果变成头脑内的影像。因此，大部分游戏引擎都会提供一组API，去绘画有颜色的线条、简单图形及三维文本。作者称这些API为**调试绘图**（debug drawing）功能。之所以称其为调试绘图，是因为绘画这些线、形状与文字仅为了在开发及调试期间做可视化，这些功能会在游戏发布版本中移除。

调试绘图API可节省开发人员大量时间。例如，若要找出为何一个抛射物没有击中敌方角色，哪种方法会较快？在调试器中解读一堆数字？或是在游戏中绘画一条三维的曲线以显示抛射物的轨迹呢？使用调试绘图API，逻辑和数学错误立即无所遁形。这或可称作一图抵千调试分钟[1]。

以下是顽皮狗《神秘海域：德雷克船长的宝藏》引擎中的调试绘图演示。下面的全部截图都是在**游戏性测试**（playtest）关卡里拍下来的，此关卡是许多特殊关卡之一，用作测试新功能，以及调试游戏中的问题。

- 图9.1展示单条直线如何帮助开发人员得知目标是否在敌人视线范围内。你也可以看到一些调试文本在敌人头顶上渲染，这个例子中文本显示了武器射程、伤害倍增器、与

[1] 译注：此句子沿于谚语“A picture is worth a thousand words.”（一图抵千言）。

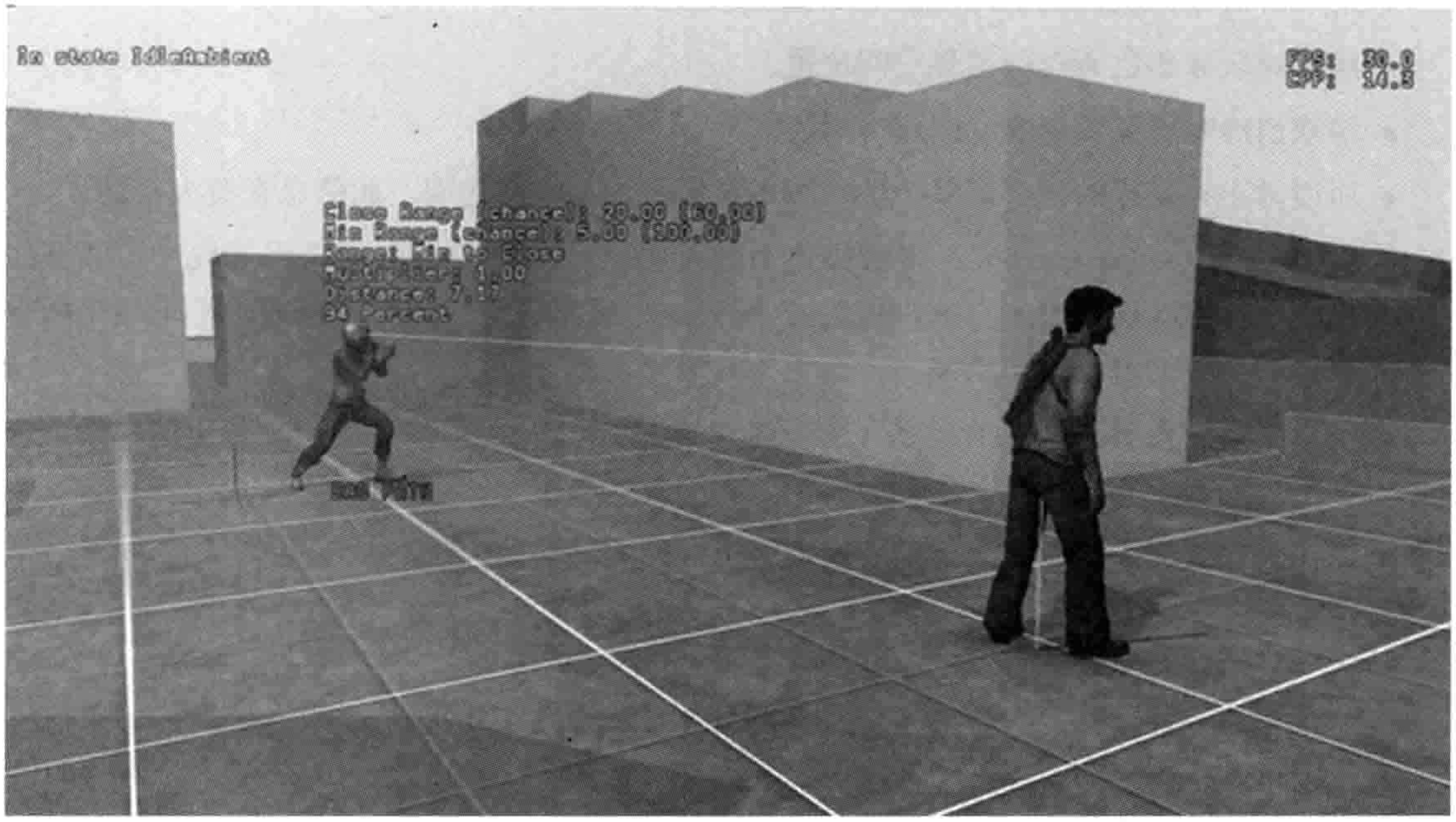


图 9.1: 视觉化NPC对玩家的视线。

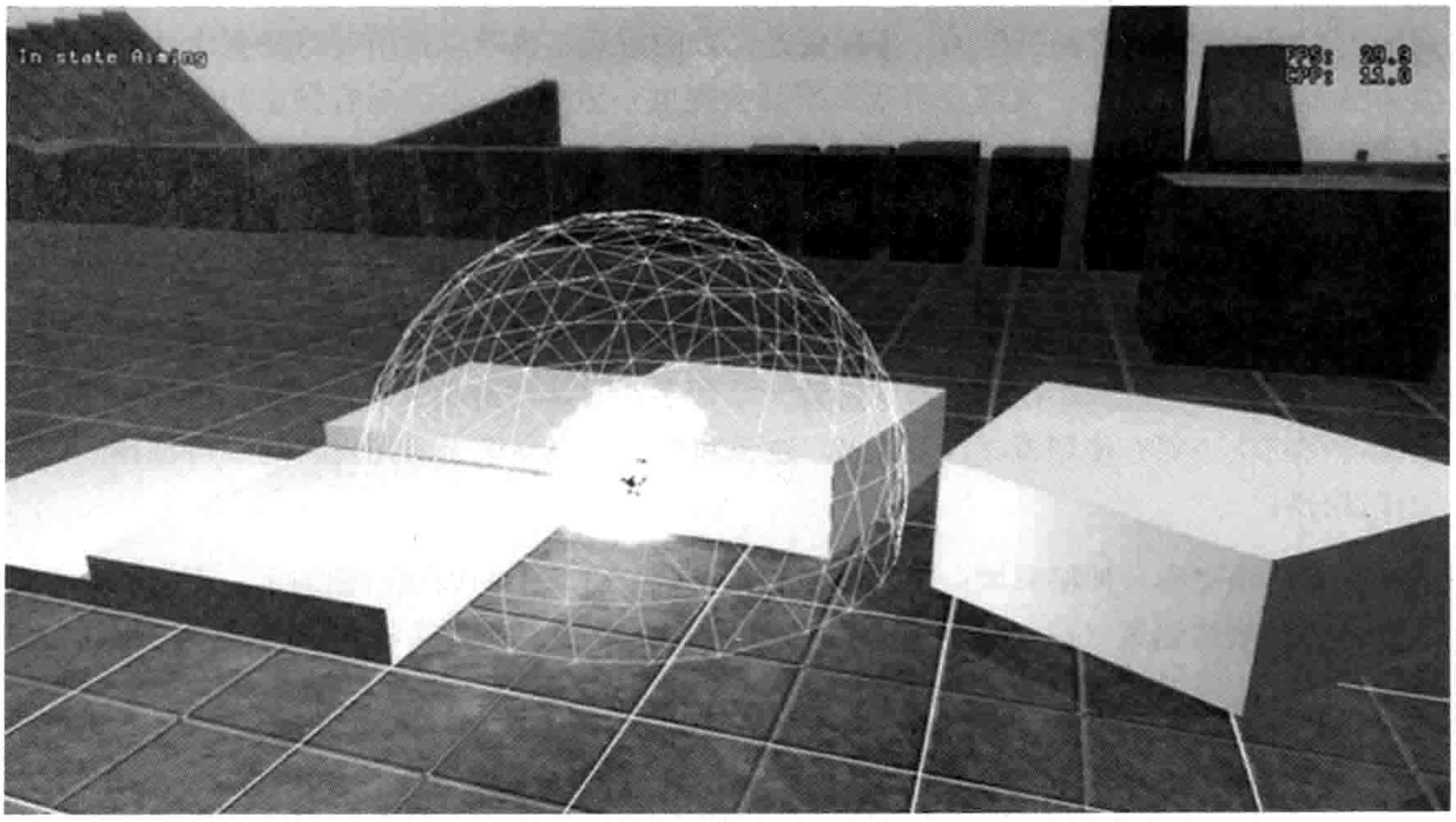


图 9.2: 视觉化爆破扩张的球体。

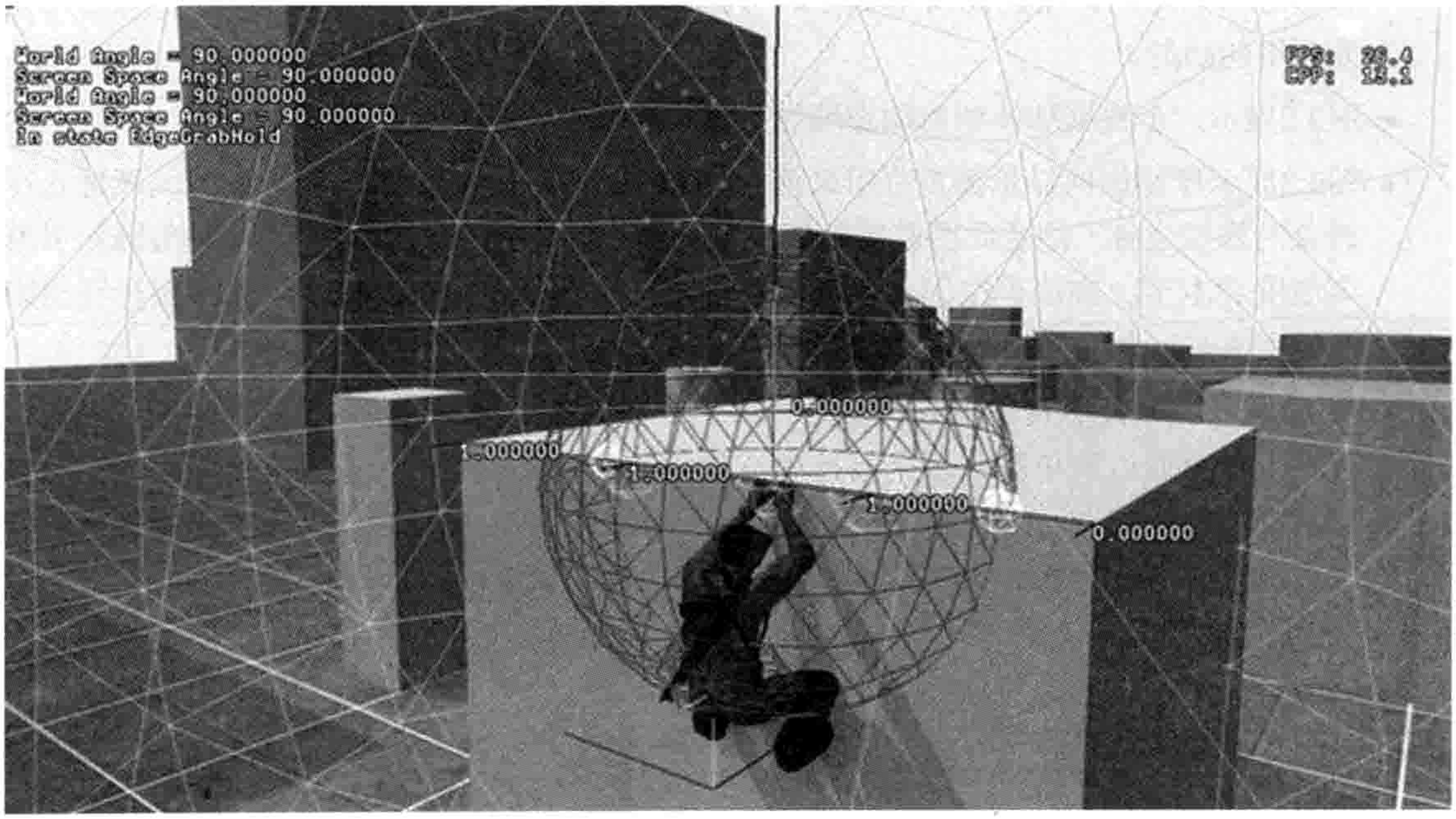


图 9.3: 供德雷克边缘攀抓及摆动系统使用的球体和矢量。

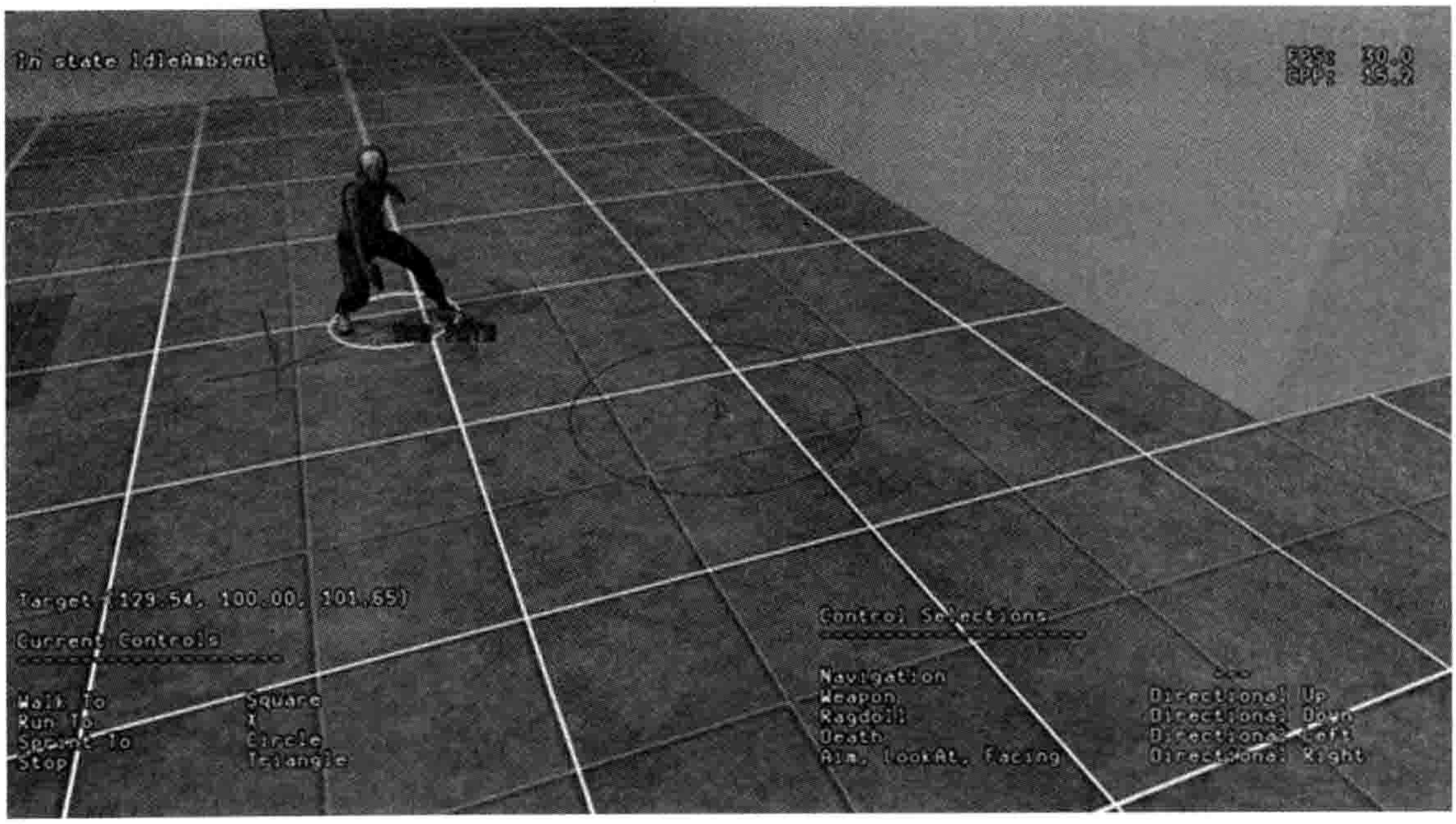


图 9.4: 为了调试，手工控制NPC的动作。

目标之间的距离、角色能击中目标的可能性百分点。在三维空间中打印任意信息是极为有用的功能。

- 图9.2展示一个线框球体如何动态地显示爆炸的爆破扩张范围。
- 图9.3展示球体如何显示德雷克用来搜索可攀抓边缘（ledge）的半径。红线代表他正在抓着的攀抓边缘。注意此图左上角的白色文本。《神秘海域：德雷克船长的宝藏》引擎可渲染文本至二维屏幕空间，也可渲染于全三维的空间。当要显示不受摄像机视角影响的文本时，此功能就十分实用了。
- 图9.4展示一个处于特殊调试模式的AI角色。在此模式下，角色的脑袋实际上被关掉，开发人员可通过简单的菜单全权控制该角色的移动及动作。开发人员可简单地把摄像机对准某些目标点，然后命令该角色步行、奔跑或疾跑（sprint）至那些指定地点。开发人员也可指示角色进入或离开附近的掩护地点，让他用武器开火等。

9.2.1 调试绘图API

调试绘图API通常需要满足下列要求。

- API应该又简单又容易使用
- API应支持一组有用的图元（primitive），包括但并不限于
  - 直线。
  - 球体。
  - 点（通常表示为小交叉或球体，因为单个像素很难看得见）。
  - 坐标轴（通常x轴画成红色，y轴绿色，z轴蓝色）。
  - 包围盒（bounding box）。
  - 格式化文本。
- API应该能弹性控制图元如何绘画，包括：
  - 颜色。
  - 线的宽度。
  - 球体半径。
  - 点的大小、坐标轴的长度，以及其他预设图元的尺寸。
- API应该可以把图元绘画至世界空间（全三维、使用游戏摄像机的透视投影矩阵），或屏幕空间（选用正射投影，甚至透视投影）。世界空间图元适合注释三维场景的物体。屏幕空间图元则适合显示一些调试信息，如HUD一样不受摄像机位置和方向影响。

- API应可选择是否使用**深度测试**（depth testing）来绘画图元。
  - 当开启深度测试，图元会被场景中的真实物体所遮挡。这样会较容易显示图元和物体的前后关系，但同时也意味着图元有时会难以观察，甚至完全被场景中的物体遮掩。
  - 当关闭深度测试，图元便会“漂浮”在场景中所有真实物体之前。这样会难以判断图元和物体的前后关系，但能肯定图元永不会在视域范围里隐藏起来。
- 应该可以在代码里的任何地方调用此API。多数渲染引擎都会要求，只能在游戏循环的某个阶段才能渲染几何体，而通常是在每帧的最后阶段。所以，此需求意味着系统必须把所有调试绘画请求排进一个队列中，以等待合适的时机才渲染这些请求。
- 理想地，每个调试图元都应包含其**生命期**（lifetime）。生命期控制图元在提交后维持于屏幕上的时间。若某段绘画调试图元的代码在每帧都会执行，那么生命期应设为1帧——因为那些图元每帧都会被刷新，所以能一直显示在屏幕上。然而，若某段绘画调试图元的代码很少或间歇地执行（例如某函数负责计算抛射物的初始速度），那么你不会希望那些图元闪现1帧然后就消失。在这种情况下，程序员最好可以给定调试图元更长的生命期，大约以数秒计。
- 调试绘图系统应能高效地处理大量的调试图元。当要为数千个游戏对象绘画调试信息，图元数量就累积得很庞大了，总不希望开启调试绘图时游戏变得无法正常运行。

以下是顽皮狗《神秘海域：德雷克船长的宝藏》引擎调试绘图API大约的样子：

```
class DebugDrawManager
{
public:
    // 在调试绘图队列中加入一条线段
    void AddLine(    const Point& fromPosition,
                     const Point& toPosition,
                     Color color,
                     float lineWidth = 1.0f,
                     float duration = 0.0f,
                     bool depthEnabled = true);

    // 在调试绘图队列中加入一个轴对齐十字（3条线汇集于1点）
    void AddCross(   const Point& position,
                     Color color,
                     float size,
                     float duration = 0.0f,
                     bool depthEnabled = true);

    // 在调试绘图队列中加入一个球体
```

```
void AddSphere( const Point& centerPosition,
                float radius,
                Color color,
                float duration = 0.0f,
                bool depthEnabled = true);

// 在调试绘图队列中加入一个圆形
void AddCircle( const Point& centerPosition,
                const Vector& planeNormal,
                float radius,
                Color color,
                float duration = 0.0f,
                bool depthEnabled = true);

// 在调试绘图队列中加入一组坐标轴，表示位置及定向
void AddAxes(   const Transform& xfm,
                Color color,
                float size,
                float duration = 0.0f,
                bool depthEnabled = true);

// 在调试绘图队列中加入一个线框三角形
void AddTriangle(   const Point& vertex0,
                    const Point& vertex1,
                    const Point& vertex2,
                    Color color,
                    float lineWidth = 1.0f,
                    float duration = 0.0f,
                    bool depthEnabled = true);

// 在调试绘图队列中加入一个轴对齐包围盒
void AddAABB(   const Point& minCoords,
                const Point& maxCoords,
                Color color,
                float lineWidth = 1.0f,
                float duration = 0.0f,
                bool depthEnabled = true);

// 在调试绘图队列中加入一个定向包围盒(OBB)
void AddOBB(    const Mat44& centerTransform,
                const Vector& scaleXYZ,
                Color color,
                float lineWidth = 1.0f,
                float duration = 0.0f,
                bool depthEnabled = true);
```

```
    // 在调试绘图队列中加入一个字符串
    void AddString( const Point& pos,
                    const char* text,
                    Color color,
                    float duration = 0.0f,
                    bool depthEnabled = true);
};

// 此全局调试绘图管理器是为了在全三维透视投影中绘图
extern DebugDrawManager g_debugDrawMgr;

// 此全局调试绘图管理器负责在二维屏幕空间中渲染图元。坐标(x, y)用于指定屏幕上
// 的二维位置，而z坐标则是储存一个代号，用来表示(x, y)坐标是以绝对像素为单位的，
// 还是以范围于0.0～1.0的归一化坐标为单位的（后者令绘图与屏幕实际分辨率无关）
extern DebugDrawManager g_debugDrawMgr2D;
```

以下是游戏代码使用此API时的例子：

```
void Vehicle::Update()
{
    // 做一些计算……

    // 调试绘画我的速度矢量
    Point start = GetWorldSpacePosition();
    Point end = start + GetVelocity();
    g_debugDrawMgr.AddLine(start, end, kColorRed);

    // 做另一些计算……

    // 调试绘画我的名字及乘客数量
    {
        char buffer[128];
        sprintf(buffer, "Vehicle %s: %d passengers",
            GetName(), GetNumPassengers());
        g_debugDrawMgr.AddString(GetWorldSpacePosition(),
            buffer, kColorWhite, 0.0f, false);
    }
}
```

注意绘画函数的名字并非使用动词“draw”而是“add”。因为调试图元一般并不是在调用函数后立即绘画的，而是把数据加进一个列表，在稍后才绘画出来。多数高速的三维渲染引擎都会要求，用一个**场景**数据结构管理所有视觉元素，使引擎能更高效地渲染，渲染通常在游戏循环末进行。第10章会更深入探讨渲染引擎如何运作。

9.3 游戏内置菜单

每个游戏引擎都有大量的配置选项及功能。事实上，每一个主要的子系统，包括渲染、动画、碰撞、物理、音频、网络、玩家机制、人工智能等子系统，都有其专门的配置选项。在游戏运行期间，若程序员、美术人员、游戏设计师等能直接配置这些选项，将会是非常有用的功能。否则可能要编辑源代码，重新编译代码，链接游戏可执行文件，再重回游戏，才能看到配置选项的效果。所以在游戏进行中配置选项，就能大幅缩减游戏开发团队用于调试问题、设置新关卡和游戏机制所需的时间。

在游戏进行中配置选项，最简单及方便的办法便是提供**游戏内置菜单**（in-game menu）系统。游戏内置菜单中的项目可做很多事情，例如（但肯定不限于）：

- 切换（toggle）全局布尔设定。
- 调校全局整数及浮点数值。
- 调用一些引擎提供的函数，执行任何任务。
- 开启副菜单，使菜单系统按层阶式管理，以方便浏览。

引擎应该能使用户轻松地激活游戏内置菜单，例如通过按手柄上的按钮。（当然，要选择一些正常游戏过程不会使用到的按钮组合。）开启游戏内置菜单时，通常会把游戏暂停。那么开发者就能在测试游戏时，直至问题出现前的一刻才开启游戏内置菜单，接着在游戏暂停时，调校一些引擎选项以更清楚地显示游戏的问题，然后就可以取消暂停，继续深入调查该问题。

以下我们看一看顽皮狗《神秘海域：德雷克船长的宝藏》引擎的菜单系统如何运作。图9.5展示了其顶级菜单，每个子菜单对应引擎中的主要子系统。图9.6中我们进入了“**Rendering...**”子菜单。由于渲染引擎是极复杂的系统，因此该菜单内还有许多子菜单去控制多个方面的渲染选项。例如要控制三维网格的渲染方式，可以再进入图9.7中的“**Mesh Options...**”子菜单。此子菜单能关上所有静态背景网格的渲染，剩下只会渲染动态的前景网格，效果如图9.8所示。

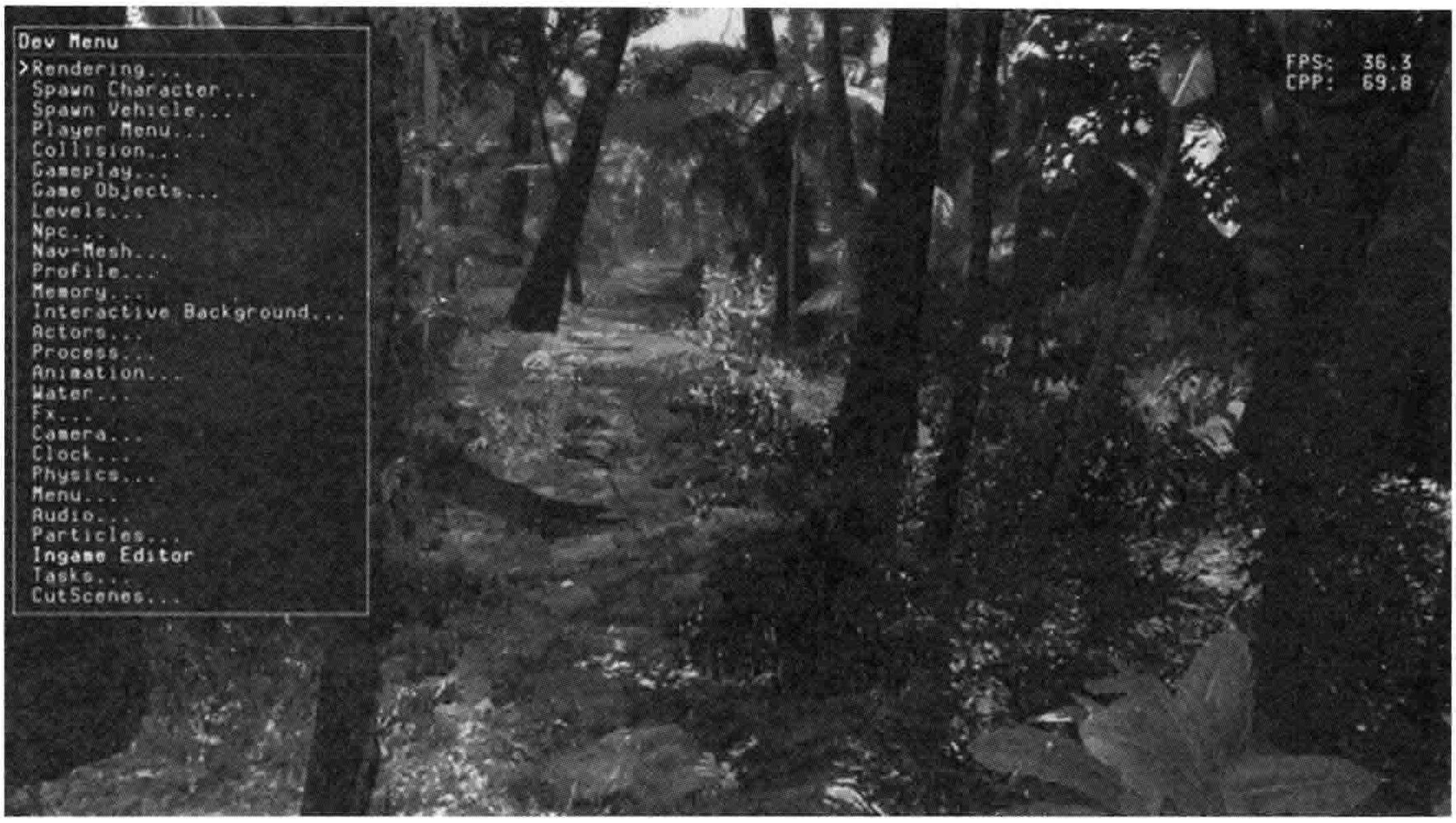


图 9.5: 《神秘海域》的主开发菜单。

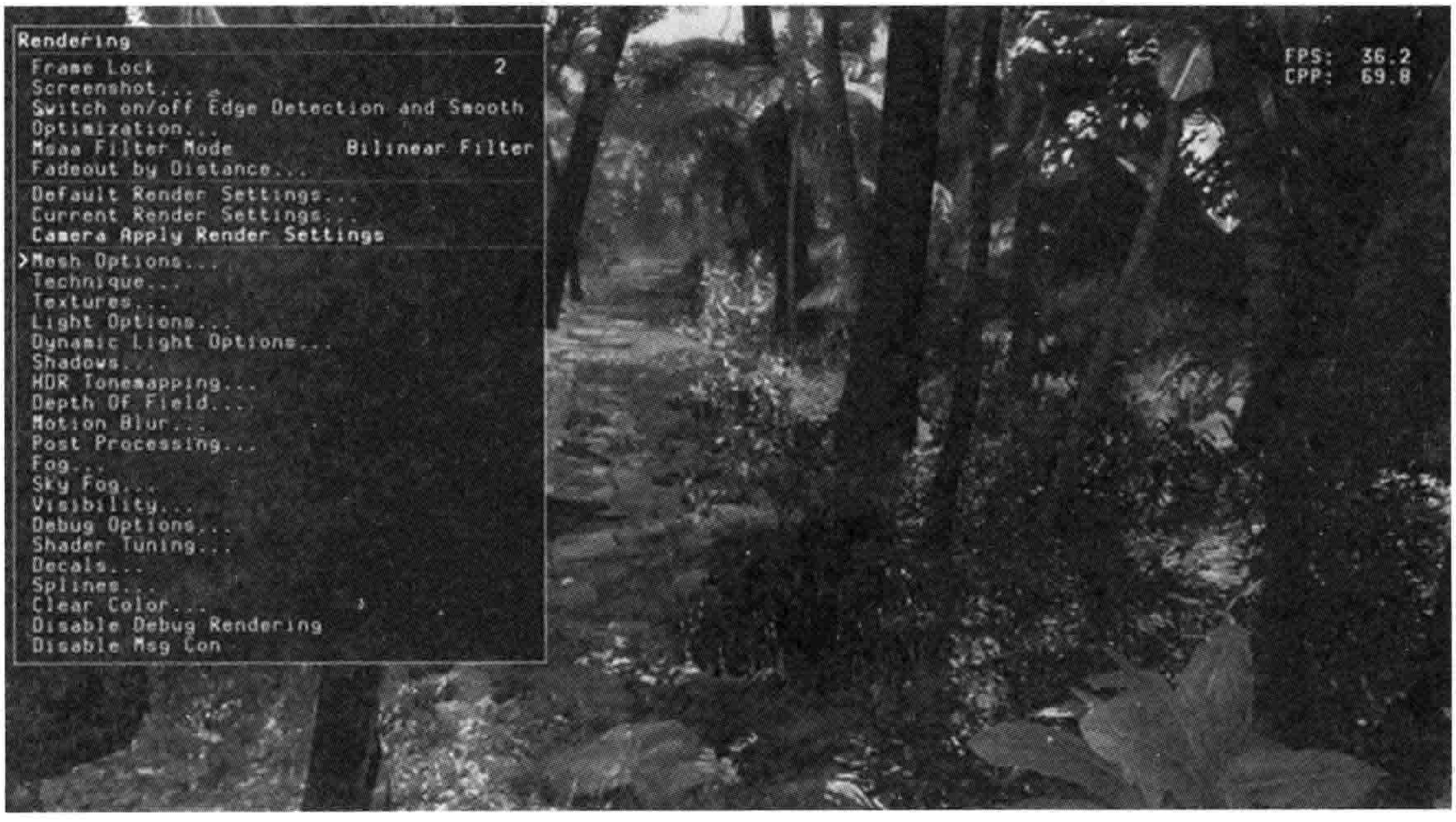


图 9.6: 渲染子菜单。

图 9.7: 网格选项菜单。

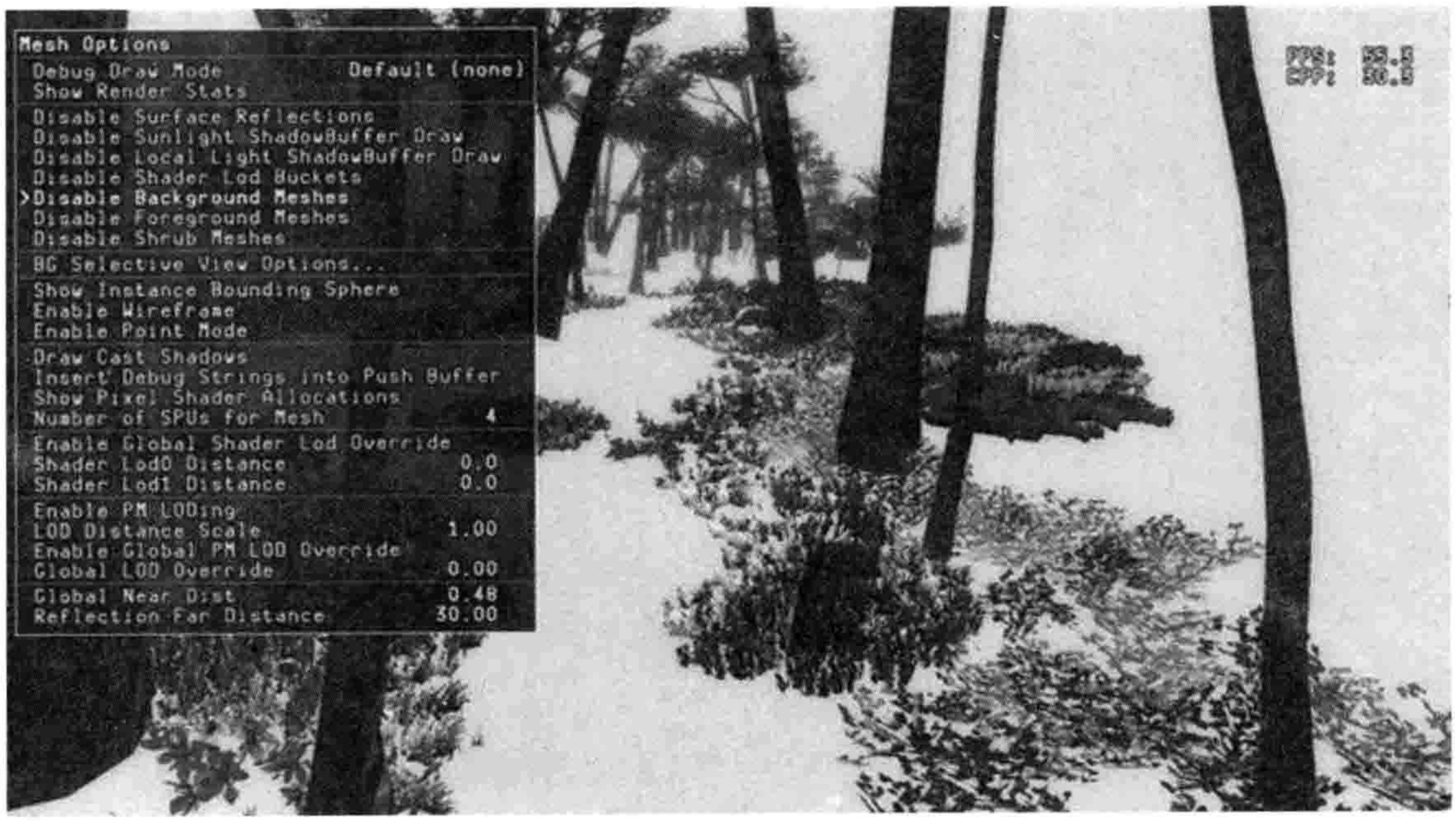


图 9.8: 关闭背景网格。

9.4 游戏内置主控台

有些引擎提供游戏内置主控台（in-game console），或会取代游戏内菜单，或会和菜单并存。游戏内置主控台提供命令式接口以让用户使用游戏引擎的功能，就如同DOS**命令提示符**让用户使用Windows操作系统的功能，又如同csh、tcsh、ksh、bash**壳层**让用户使用类UNIX的操作系统。与菜单系统相似，游戏引擎主控台可提供一些命令，使开发人员能检视及操控全局引擎设置，以及执行各种命令。

主控台和菜单系统相比较不方便，尤其是对于打字不快的用户来说。然而，主控台可以比菜单更强大。有些游戏内置主控台仅提供一组基本的硬编码命令，这种主控台的弹性和菜单差不多。但是，另一些主控台提供丰富的接口可使用几乎所有引擎功能。图9.9展示了《雷神之锤4》的游戏内置主控台。

一些游戏引擎提供强大的脚本语言，供程序员和游戏设计师延伸引擎的功能，甚至用以制作全新的游戏。若游戏内置主控台也是使用相同的脚本语言来“沟通”的，那么任何脚本可以做的事，都可以通过主控台互动地执行。14.8节里我们会深入探讨脚本语言。

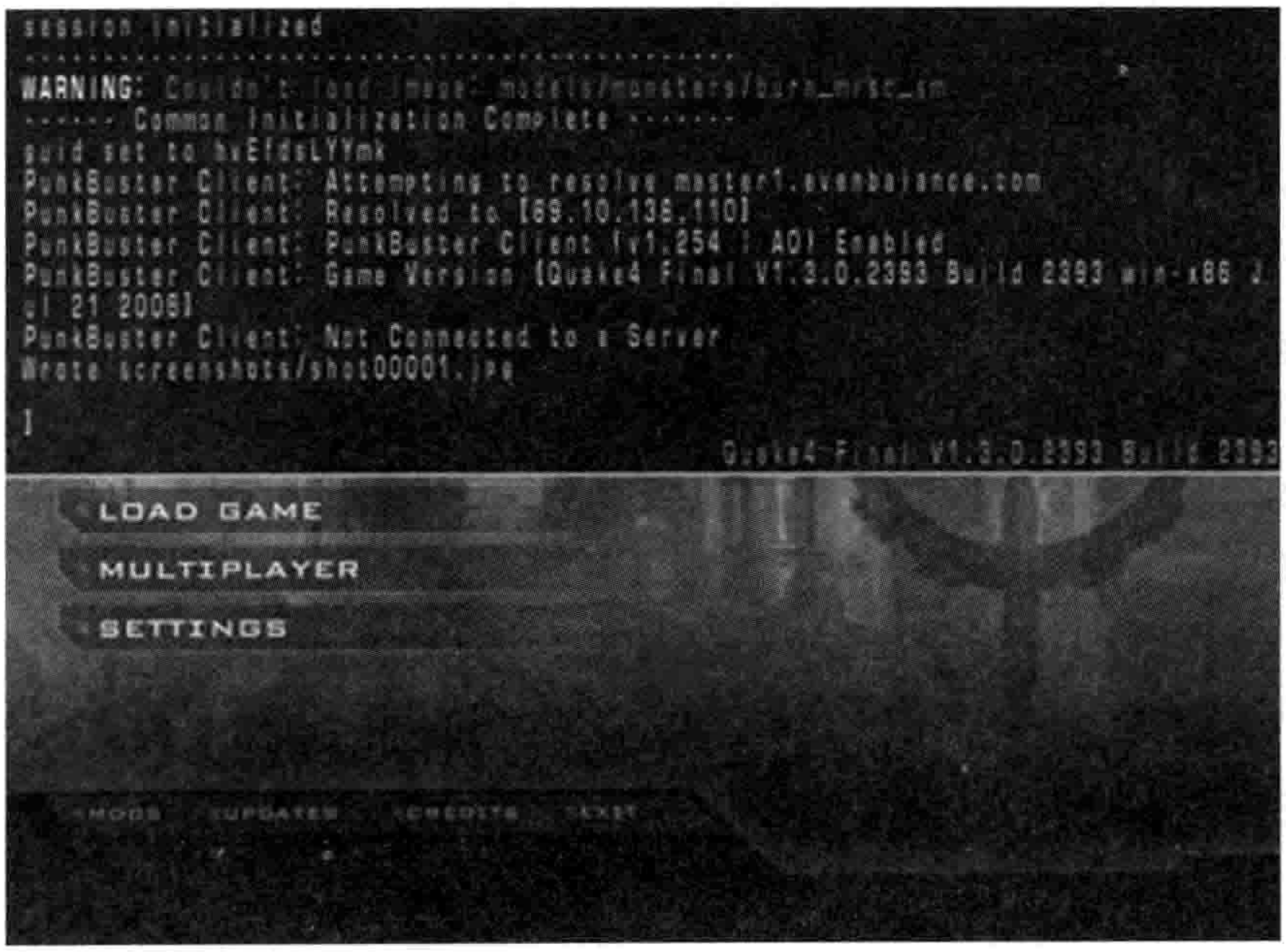


图 9.9: 《雷神之锤4》的游戏内置主控台，叠加于主游戏菜单之上。

9.5 调试用摄像机和游戏暂停

游戏内置菜单或主控台最好能附带两个重要功能：（a）把摄像机从游戏角色分离出来，控制摄像机在游戏世界里飞驰，以细察场景的所有环节；（b）暂停、恢复暂停、单步执行游戏（参见7.5.6节）。当游戏暂停时，必须仍然可以控制摄像机。实现此功能的方法是，即使游戏的逻辑时钟是暂停的，仍保持执行渲染引擎和摄像机控制系统。

慢动作模式（slow motion mode）是另一个极为有用的功能，用以细察动画、粒子效果、物理行为、碰撞行为、人工智能行为等。此模式也容易实现。假设我们使用一个独立于实时的时钟，来更新所有游戏性元素，那么只要使该时钟的更新速率慢于正常，就能令游戏进入慢动作模式。此方式也可以用来实现快动作模式（fast motion mode），可于费时的游戏部分高速移动，更快地走到目标地点。

9.6 作弊

在开发和调试游戏时，让用户为了方便而打破游戏规则，是很重要的一件事。这些功能恰当地命名为**作弊**（cheat）。例如，许多引擎可以让用户“拾起”玩家角色，将角色飞到游戏世界任何地方，而且碰撞会被关闭，使角色能穿越所有障碍物。在测试游戏性时，此功能极为有用。与其为了把角色带到某目的地而浪费时间去玩游戏，不如简单地拾起角色，使角色飞到想去的地方，再放下角色恢复至正常游戏模式。

其他有用的作弊包括但不限于：

- **不死身**（invincible）：在测试功能或调试bug时，开发者通常不希望为了在大量敌人中保护自己的角色而烦恼，或是担心角色从高处掉下来而伤亡。
- **给玩家武器**：为了测试，引擎通常可以给予玩家任何游戏中的武器。
- **无尽弹药**：当开发者在尝试杀敌，以测试武器系统或AI被击中的行为时，一定不会想四处找弹夹。
- **选择角色网格**：若玩家角色有多于一套“装束”，在测试时可任选一套也是很有用的。

显然这个作弊表可延伸多页。读者可以加入任何所需的作弊，以帮助开发或调试游戏。有时甚至会把一些最爱的作弊留给玩发行版本的玩家。玩家要激活那些作弊，通常要用手柄或键盘输入一些没有公布的**作弊码**（cheat code），或是完成游戏中某些目标。

9.7 屏幕截图及录像

另一个极为有用的工具是获取屏幕截图，并把截图存储为合适的图像格式，例如.bmp或.tga格式。实际的截图方法细节，每个平台都不一样，但一般是通过调用图形API去把帧缓冲由显存传送至主内存，在主内存的图像就可以经扫描转换为心目中的格式。图像文件通常会被写到某个预设文件夹，并以日期时间来命名以保证文件名的唯一性。

引擎也可以为用户提供多种选项，控制如何获取屏幕截图。常见例子如下。

- 屏幕截图中是否包含调试用的图形及文本。
- 屏幕截图中是否包含HUD元素。
- 屏幕截图的分辨率：有些引擎可以获取高分辨率的屏幕截图，其做法是修改投射矩阵，例如，每次以正常分辨率获取四分之一的屏幕，最后合成一张高分辨率的图。
- 简单的摄像机动画：例如，可以让玩家标记摄像机的开始、结束位置及定向。然后就可以把摄像机从开始至结束对位置和定向慢慢插值，把一连串的屏幕截图存储下来。

有些引擎也提供全面的录像模式。这些系统以游戏的目标帧率获取屏幕截图，通常这些截图会经离线处理生成如AVI或MP4等格式的视频文件。

获取屏幕截图通常是比较慢的操作，部分原因涉及从显存传送帧缓冲至主内存的时间开销（图形硬件通常不会优化此操作），而另一大原因与图像存盘有关。若希望以实时录像（或至少**接近**实时），那么几乎必须要把获取后的屏幕截图存至主存的一个缓冲里去，当缓冲被填满后才把内容存盘（那个时候通常游戏会顿卡）。

9.8 游戏内置性能剖析

游戏是实时系统，要达到及维持高帧率（通常30FPS或60FPS）是重要的目标。因此，任何游戏程序员都有职责确保其代码能够在预算内高效运行。如在第2章讨论过的80-20及90-10规则，大百分比的代码并不需要优化。而唯一能得悉**何处**要进行优化的方法，就是**量度游戏的性能**。在第2章我们已讨论过多个第三方剖析工具。然而，这些工具都有多个限制，而且并不一定能在游戏机上运行。由于这些原因，以及为了更方便起见，许多游戏引擎提供某种形式的游戏内置性能剖析工具（in-game profiling tool）。

游戏内置剖析器让程序员标记一些代码段落，并且让程序员将每个段落命名为易懂的名字，然后剖析器会为这些段落计时。剖析器使用CPU的高分辨率时钟去量度每段代码所花的运行时间，并把数据记录在内存（如图9.10、图9.11、图9.12所示）。数据通常能以多种

方式显示，包括原始的周期、以微秒为单位的运行时间，以及相对整帧的运行时间百分比。

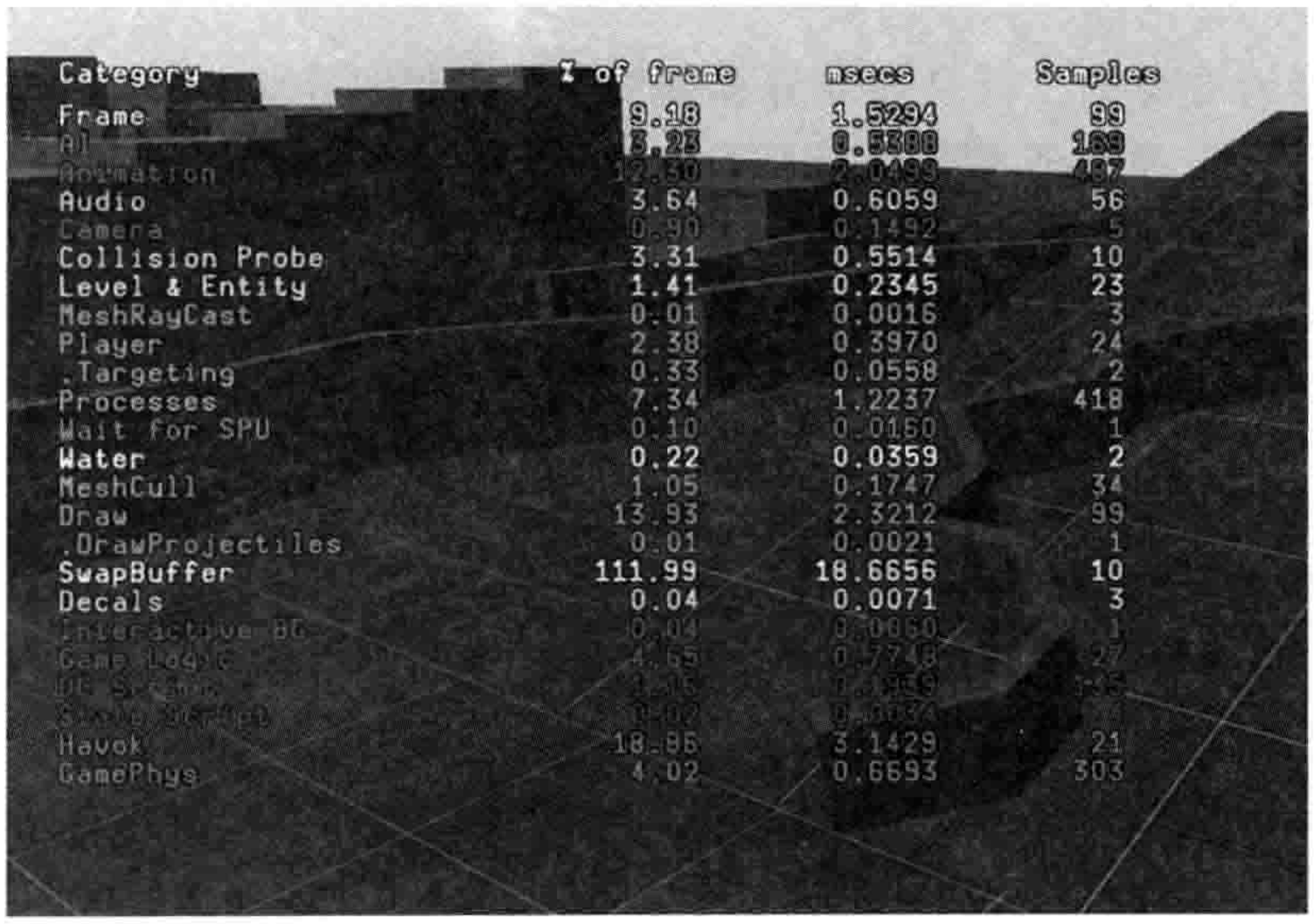


图 9.10: 《神秘海域2：纵横四海》引擎中的剖析分类显示，展示了多个顶层引擎系统的粗略计时。

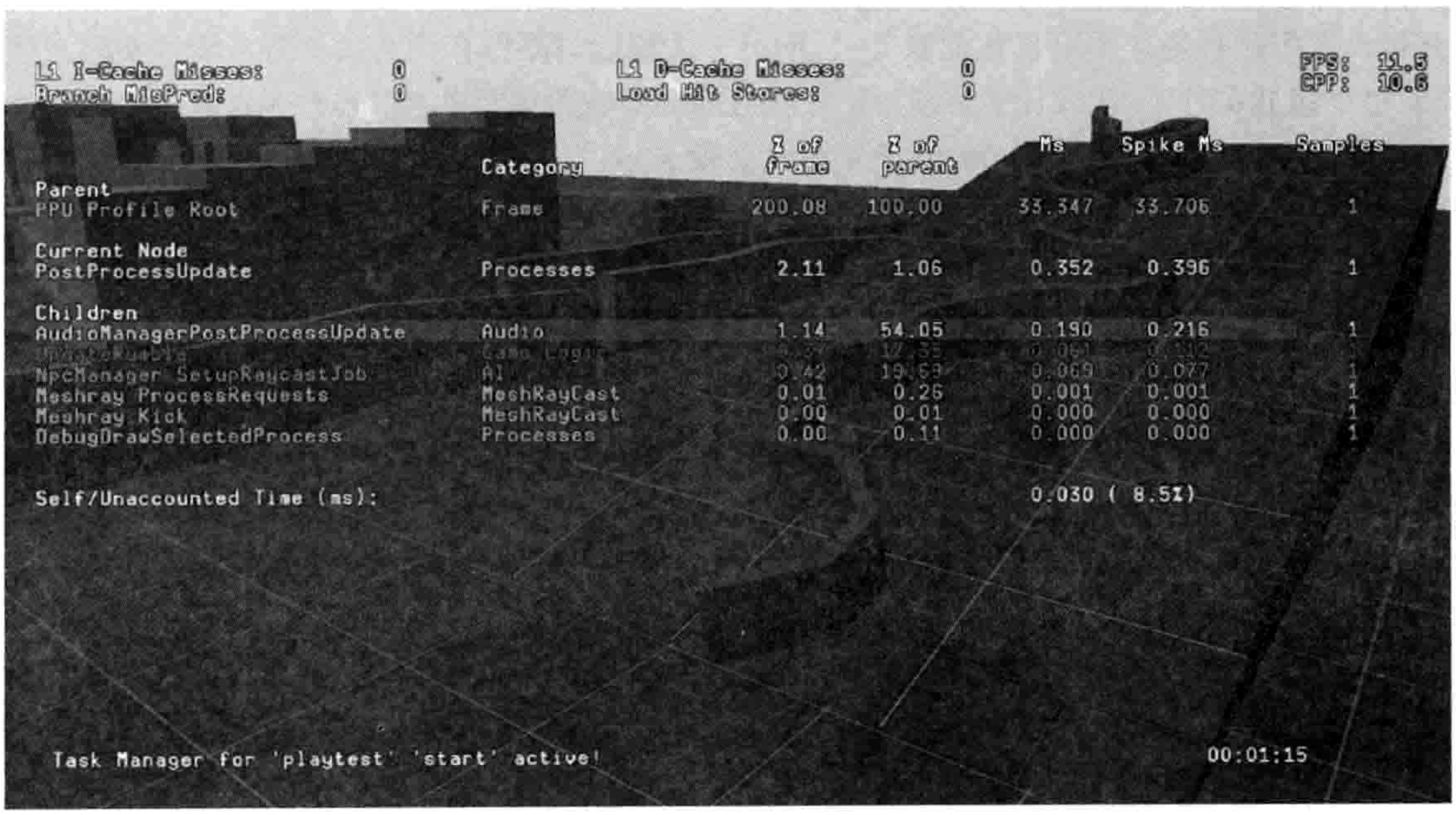


图 9.11: 《神秘海域2》引擎也提供一个层阶式的剖析显示，让用户可以深入探究至某个函数调用的开销。

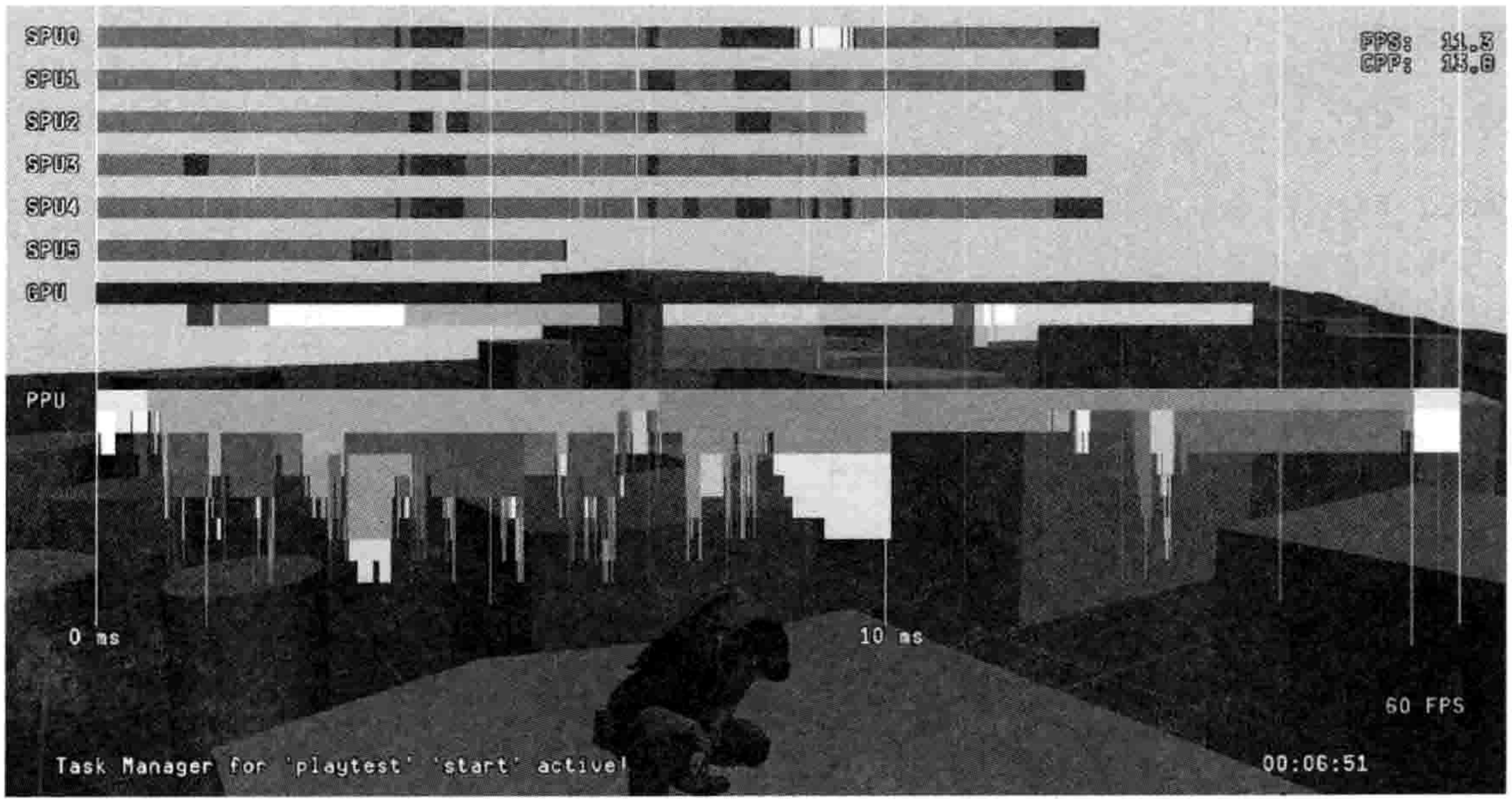


图 9.12: 《神秘海域2》引擎的时间线模式，可完全展示单帧里多个操作在PS3 SPU、GPU、CPU上的运行情况。

9.8.1 层阶式剖析

由命令式语言（imperative language）所写的计算机程序，天生就是**层阶式的**——一个函数调用某函数，该函数又再调用其他函数。例如，假设函数a()调用函数b()及c()，而函数b()又调用函数d()、e()和f()。那么，其伪代码可以写成：

```
void a()
{
    b();
    c();
}

void b()
{
    d();
    e();
    f();
}

void c() { ... }
```

```
void d() { ... }

void e() { ... }

void f() { ... }
```

假设main()直接调用函数a()，那么函数调用层阶就可以绘画成图9.13。

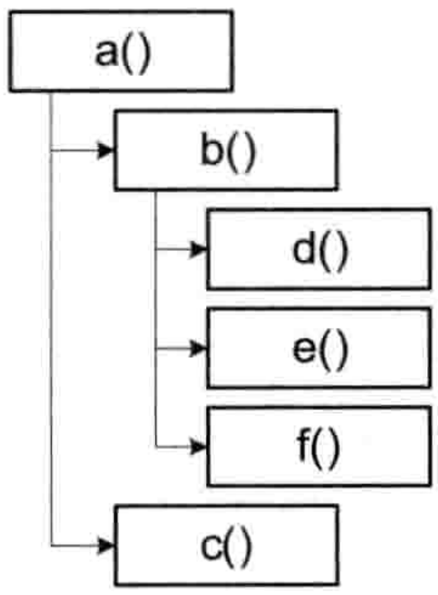


图 9.13: 一个假想的函数调用层阶。

当调试程序时，调用堆栈（call stack）只显示此树的一个快照。具体来说，调用堆栈显示在此树中，从**当前执行**的函数到达根函数的路径。在C/C++中，根函数一般会是main()或WinMain()，虽然从技术上来说，这些函数是由标准C运行时库（C runtime library，CRT）里的启动函数所调用，因此启动函数才是层阶中真正的根。例如，若在函数e()设置断点，那么断点命中时调用堆栈的情况是这样的：

```
e()                 ←当前执行中的函数。
b()
a()
main()
_crt_startup()      ←调用层阶中的根。
```

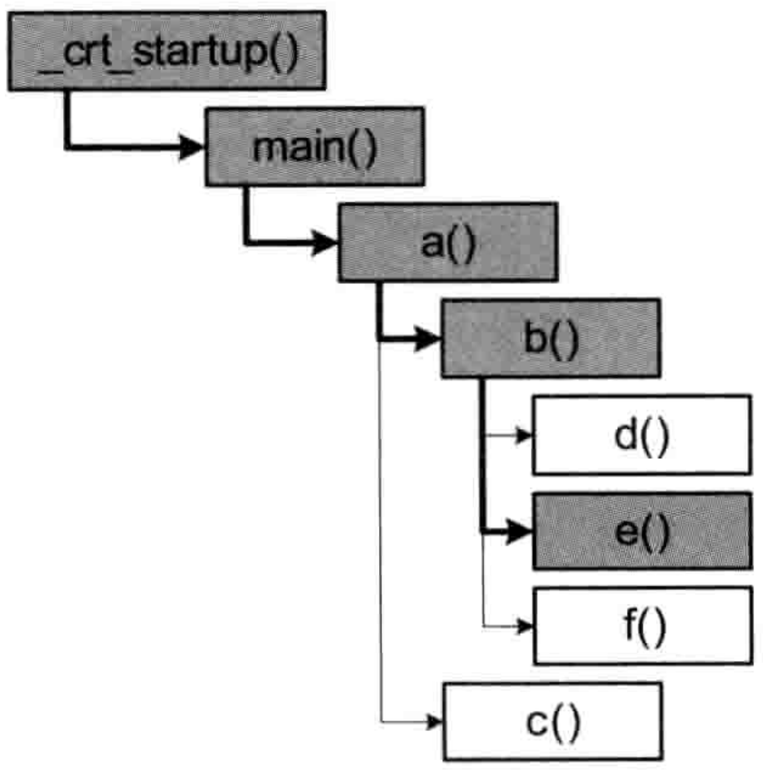


图 9.14: 从函数e()加入断点后的调用堆栈。

图9.14显示函数调用树中，该调用堆栈由函数e()到达的根的路径。

9.8.1.1 以层阶形式量度执行时间

量度单个函数的执行时间，其结果会包含该函数调用的子函数、孙函数、曾孙函数……的执行时间。要正确地理解剖析数据，必须顾及函数的调用层阶。

许多商用性能剖析工具能**自动地**在被剖析程序的每个函数中加入测控（instrumentation）。此功能可以同时量度在剖析期间每个函数调用的**包含**（inclusive）执行时间和**排他**（exclusive）执行时间。如字面之意，包含时间量度函数本身以及其调用的所有子函数的执行时间；排他时间仅量度函数本身所花的时间。（某函数的排他时间，可由其包含时间减去其所有子函数的包含时间而得出。）此外，有些剖析工具能记录每个函数的调用次数。此仍优化程序时的重要信息，让你分辨两类耗时的函数，一类是因为每次调用所花的时间长，一类是因为调用了非常多次。

相比之下，游戏内的性能剖析工具并没有那么强大，而通常是使用**手工方式**在程序中加入测控。若游戏的主循环的结构够简单，就可以在较粗尺度上取得数据，而不需要考虑函数的调用层阶。例如，一个典型游戏循环大概是这样子的：

```
while (!quitGame)
{
    PollJoypad();
    UpdateGameObjects();
    UpdateAllAnimations();
    PostProcessJoints();
    DetectCollisions();
    RunPhysics();
    GenerateFinalAnimationPoses();
    UpdateCameras();
    RenderScene();
    UpdateAudio();
}
```

剖析此游戏的性能时，可以先量度每个主要阶段的执行时间：

```
while (!quitGame)
{
    {
        PROFILE("Poll Joypad");
        PollJoypad();
    }
    {
        PROFILE("Game Object Update");
        UpdateGameObjects();
```

```
    }
    {
        PROFILE("Animation");
        UpdateAllAnimations();
    }
    {
        PROFILE("Joint Post-Processing");
        PostProcessJoints();
    }
    {
        PROFILE("Collision");
        DetectCollisions();
    }
    {
        PROFILE("Physics");
        RunPhysics();
    }
    {
        PROFILE("Animation Finaling");
        GenerateFinalAnimationPoses();
    }
    {
        PROFILE("Cameras");
        UpdateCameras();
    }
    {
        PROFILE("Rendering");
        RenderScene();
    }
    {
        PROFILE("Audio");
        UpdateAudio();
    }
}
```

以上代码内的PROFILE()宏会以一个类去实现，该类的构造函数负责开始计时，而析构函数则停止计时，并以指定的名字记录执行时间。因此该类只会为其块作用域（block scope）内的代码计时，这是由于C++的特质，对象进出作用域时会自动构造和析构。

```
struct AutoProfile
{
    AutoProfile(const char* name)
    {
        m_name = name;
```

```
        m_startTime = QueryPerformanceCounter();
    }

    ~AutoProfile()
    {
        __int64 endtime = QueryPerformanceCounter();
        __int64 elapsedTime = endtime - m_startTime;
        g_profileManager.storeSample(m_name, elapsedTime);
    }

    const char*     m_name;
    __int64         m_startTime;
};

#define PROFILE(name) AutoProfile p(name)
```

然而，用于深层的巢状函数调用时，此简单方式就无用武之地了。例如，若要在RenderScene()函数内加入几个PROFILE()，我们就必须理解函数的调用层阶关系，才能正确解读这些量度数据。

解决此问题的方法之一就是，让程序员加入一些代码去描述剖析采样（profile sampling）的层阶关系。例如，在RenderScene()函数之下的每个PROFILE(...)采样，都定义为PROFILE(“Rendering”)采样的子采样。这些关系通常在采样代码以外的地方，预先声明所有样本箱（sample bin）。例如，我们可以在引擎初始化时如下设置游戏内性能剖析器：

```
// 此代码声明多个剖析样本箱，指明样本箱的名字，以及父样本箱的名字（若有）
ProfilerDeclareSampleBin("Rendering", NULL);
    ProfilerDeclareSampleBin("Visibility", "Rendering");
    ProfilerDeclareSampleBin("ShaderSetUp", "Rendering");
        ProfilerDeclareSampleBin("Materials", "ShaderSetUp");
    ProfilerDeclareSampleBin("SubmitGeo", "Rendering");
ProfilerDeclareSampleBin("Audio", NULL);
// ……
```

此方法仍有一些问题。具体来说，若每个函数在调用层阶中都仅有一个父函数，那么此方式行之有效。但若要剖析某函数，而此函数会被多于一个父函数调用，此方法就不行了。原因显而易见，我们静态地声明样本箱，**犹如**指明每个函数只会在调用层阶中出现一次，但是实际上同一函数在调用层阶中会出现多次，每次有不同的父函数。这会产生一些误导数据，因为一个函数的时间会包含在单个父样本箱内，但实际上那些时间应该分配在多个父样本箱内。许多游戏引擎没有解决此问题，因为那些引擎主要目标是剖析一些粗粒度的函数，

而那些函数只出现在调用层阶特定位置。然而，当要用这些引擎中的剖析器时，要特别注意此限制。

我们也希望知道每个函数的调用**次数**。在上述例子中，1帧内每个被剖析的函数仅被调用一次且仅一次。然而有些调用层阶中较深的函数，可能在每帧内会被调用多次。当量度到函数x()花了2ms执行，我们需要知道，2ms是x()每次执行的时间，还是x()仅耗2ms但执行了1000次。记录函数在每帧的调用次数，十分简单——剖析系统可以在每次收到时间样本时，使该样本箱里的计数器加1，而在每帧开始时把计数器重置。

9.8.2 导出至Excel

有些引擎可以把游戏内置性能剖析工具的数据导出至文本文件，以供往后分析。笔者发现，逗号分隔型取值（comma-separated values，CSV）格式最好用，因为Excel可以读取这些文件，并进行各种数据操作和分析。笔者为《荣誉勋章之血战太平洋》引擎编写过这样的导出器（exporter）。每列对应多个评注部分，而每行则代表游戏执行时某帧的剖析采样。第1列是帧的序号，第2列是以秒为单位的实际游戏执行时间。团队就能利用此文件，绘制效能对于时间的图表，并分析每帧的执行时间分布。在导出的Excel文件中增添一些简单的公式，还可以计算帧率、执行时间百分比等数据。

9.9 游戏内置的内存统计和泄漏检测

除了运行时性能（即帧率）之外，多数游戏引擎也受目标平台上的内存所限。PC游戏对这方面的限制最少，因为现时的PC有强大的虚拟内存系统。但即使如此，PC游戏还是会受“最低配置”计算机的内存所限。最低配置印刷在游戏包装上，是游戏发行商保证游戏能运行的最低级计算机。

因此，多数游戏引擎都会实现自定义的内存追踪工具（memory tracking tool）。这些工具使开发人员得知每个引擎子系统花费了多少内存，以及是否有内存泄漏（memory leak，即某些内存在分配后没有释放）。当要减少内存使用量以适应目标游戏机或PC类型时，这些信息就可以协助做出明智的决定。

然而，跟踪游戏实际使用了多少内存，是个出奇棘手的任务。读者或许以为只要简单地把`malloc()/free()`或`new/delete`包装为一对函数或宏，就可以跟踪已分配和已释放的内存。可是，现实非如此简单，原因如下。

1. **你不能控制他人代码的分配行为：**除非自己完全从无到有写操作系统、驱动程序及游

戏引擎，否则你的游戏最终很大机会要链接一些第三方库。多数优良的库会提供**内存分配钩子**（memory allocation hook），那么就可用自己的分配器取代库内预设的。但有一些库并不提供钩子。跟踪引擎用到的每个第三方库所分配的内存，并不容易。但是，若严格筛选第三方库，这通常还是**可行**的。

2. **内存有不同形式：**例如，PC有主内存及显存（即位于显卡的内存，主要用作保存几何及纹理数据）两种内存。即使可以自己跟踪在主内存里进行的内存分配及释放，仍几乎不可能跟踪显存的使用。因为图形API如DirectX会向开发者隐藏显存分配及释放的细节。游戏机的情况会比较好一点，只因开发者通常要自行编写显存的管理系统。这比使用DirectX困难，但至少能知悉所有运行的细节。[2]

3. **分配器有不同形式：**许多游戏使用特殊分配器做不同用途。例如，《神秘海域：德雷克船长的宝藏》引擎含多个分配器：一个**全局堆**作为通用分配器；一个供分配**游戏对象**的堆，因为在游戏世界中会不断产生及销毁游戏对象；一个**关卡载入堆**，用于分配在游戏中以**流**（stream）读入的数据；一个堆栈分配器供**单帧分配**（栈在每帧后自动清空）；一个**显存**分配器；以及一个**调试用的堆**，只用来分配发行版本中不需要的内存。这些分配器各自在游戏开始时取得一大块内存，然后自行管理那些内存。若要跟踪所有new/delete的调用，那么只能看见这6个分配器各调用了一次new，就没其他信息了。要取得有用的信息，我们需要跟踪每个分配器**内**的分配情况。

大多数专业的游戏开发团队会花上大量精力制作游戏内的内存追踪工具，以提供准确及详细的信息。这些工具通常能以多种形式输出。例如，引擎可以制作游戏中某段时期内所有内存分配的详细日志。这些数据可能包含按分配器或按游戏系统划分的内存峰值，以显示每部分所需的最大物理内存量。有些引擎也提供在游戏运行时显示内存用量的HUD。这些数据可以表格方式显示（如图9.15所示），或是用图表方式显示（如图9.16所示）。

此外，当出现低内存或内存不足的情况时，优良的引擎会尽力以最有用的方式输出此信息。当开发PC游戏时，开发团队通常会用高端的计算机，其内存会多于最低配置的计算机。相似地，开发视频游戏时会使用特别的**开发机**，其内存会比零售的游戏机大。所以在这两种情况下，就算从技术上来说游戏已经内存不足，游戏还是可以继续运行的（即已无法适应最低配置的PC或零售游戏机）。当出现这类内存不足的情况，游戏引擎可显示消息，如“内存不足——本关卡无法于零售游戏机上运行”。

游戏引擎的内存追踪系统还可以从多方面协助开发者及早且方便地定位问题。以下是一些例子。

---

[2]译注：另一个方法是，Direct3D 9在创建设备时，可加入D3DCREATE_DISABLE_DRIVER_MANAGEMENT选项，之后便可以用D3DQUERYTYPE_ResourceManager取得一些关于Direct3D对象的主内存和显存统计。但要注意这种Query只能在Direct3D Debug Runtime中使用。

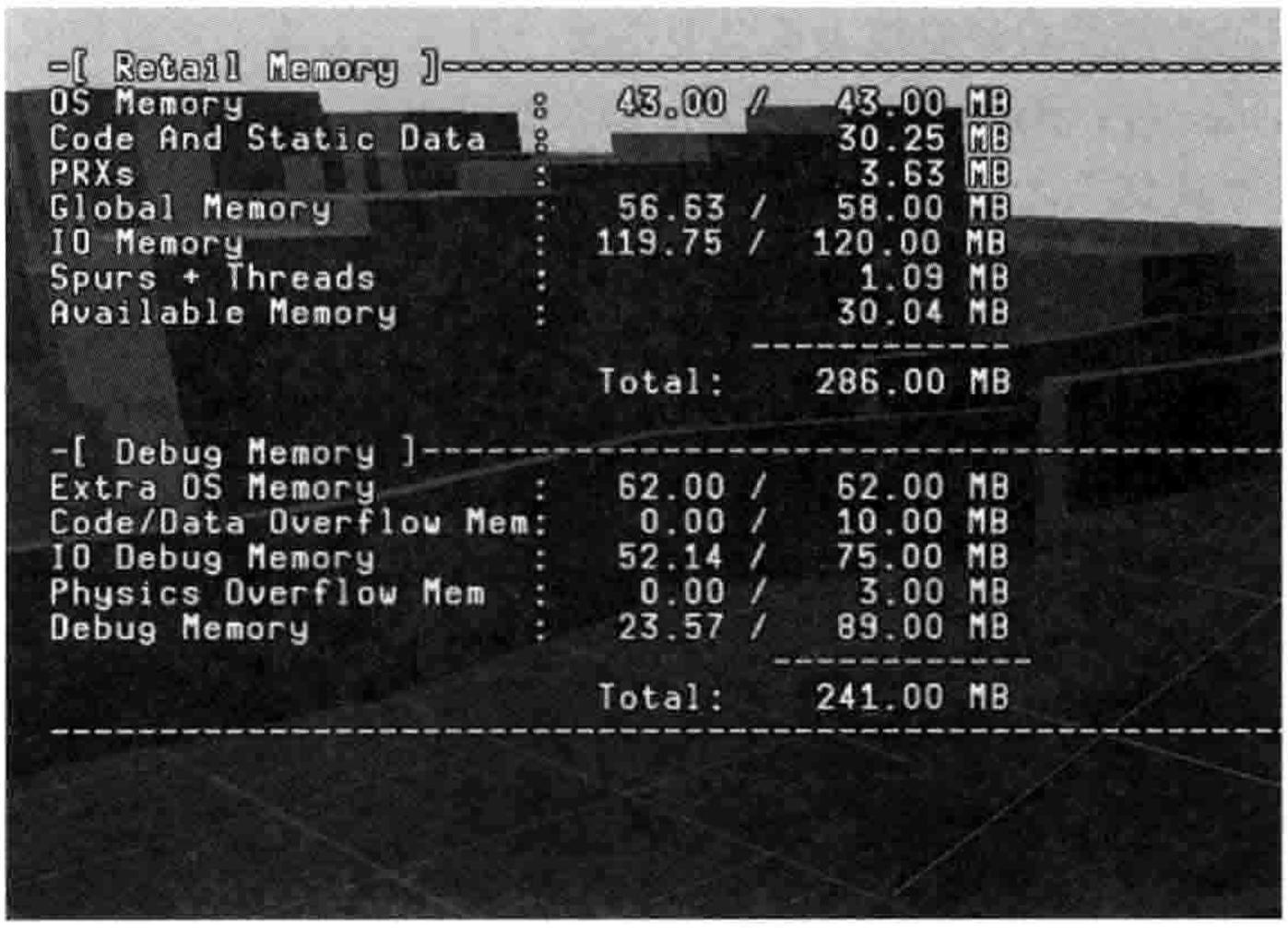


图 9.15: 《神秘海域2：纵横四海》引擎的表格方式内存统计。

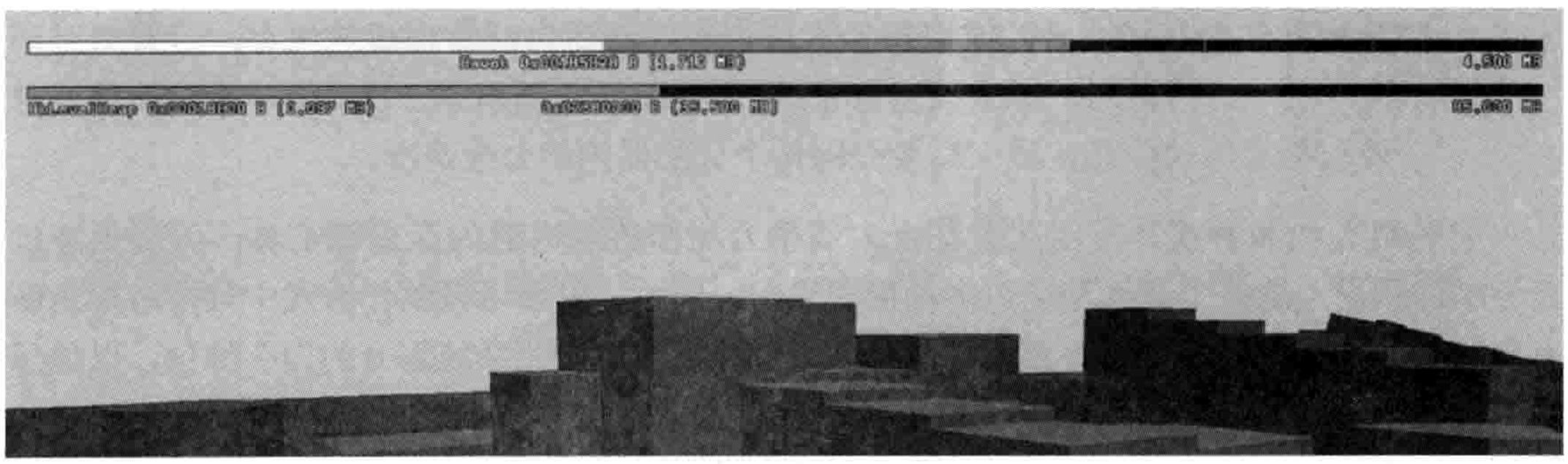


图 9.16: 《神秘海域2》引擎的图形化内存用量显示。

- 若模型载入失败，可用鲜红的文本字符串以三维形式渲染于原来模型所在之处。
- 若纹理载入失败，可用丑陋的粉红色纹理来取代，使物体渲染得显然不像最终版本的一部分。
- 若动画载入失败，游戏角色可以摆出一个特别（或许是可笑的）姿势，以表示欠缺一个动画，并且欠缺的动画名字应飘浮于角色的头上。

编写优良内存分析工具的要点有：（a）提供准确信息，（b）把数据以方便及令问题显而易见的方式呈现，（c）提供上下文信息以协助团队追踪问题根源。

第三部分

图形及动画

第10章　渲染引擎

多数人在想到计算机和视频游戏时，第一个进入脑海的印象就是惊艳的三维图形。实时三维渲染是极其广且深的题目，因此无法在短短一章里包括所有细节。有幸坊间有关于此题目的大量卓越书籍及其他资源可供参考。事实上，在开发游戏引擎的所有技术中，实时三维图形可能是最获详尽讨论的技术之一。因此，本章的目标是为读者提供实时渲染技术的概述，以及作为进一步学习的跳板。读毕本章后，阅读其他三维图形书籍会更得心应手。读者或许还可以在派对上给朋友留下深刻印象（……或许会变得更“宅”）。

本章首先会铺垫所有实时三维渲染引擎都必备的概念、理论和数学根基，然后会谈及如何用软硬件管道把这些理论框架变为现实，之后再讨论常见的优化技巧，以及这些技巧如何驱动多数引擎中的工具管道架构及运行时的渲染API。最后纵览现今游戏引擎所采用的一些高级渲染技巧及光照模型。本章处处会引用一些笔者最爱的书籍及其他资源，应能帮助读者更深入了解相关题目。

10.1　采用深度缓冲的三角形光栅化基础

三维场景渲染的本质涉及以下这几个基本步骤。

- 描述一个**虚拟场景**（virtual scene）。这些场景一般是以某数学形式表示的三维表面。
- 定位及定向一个**虚拟摄像机**（virtual camera），为场景取景。摄像机的常见模型是这样的：摄像机位于一个理想化的焦点（focal point），在焦点前的近处悬浮着一个影像面（image surface），而此影像面由多个虚拟感光元件（virtual light sensor）所组成，每个感光元件对应着目标显示设备的像素（picture element/pixel）。
- **设置光源**（light source）。光源产生的光线会与环境中的物体交互作用并反射，最终会到达虚拟摄像机的感光像面。
- 描述场景中物体表面的**视觉特性**（visual property）。这些视觉特性决定光线如何与物

体表面产生交互作用。

- 对于每个位于影像矩形内的像素，渲染引擎会找出经过该像素而聚焦于虚拟摄像机焦点的（一条或多条）光线，并计算其颜色及强度（intensity）。此过程称为**求解渲染方程**（solving the rendering equation），也叫作**着色方程**（shading equation）。

图10.1描绘了此高级渲染过程。

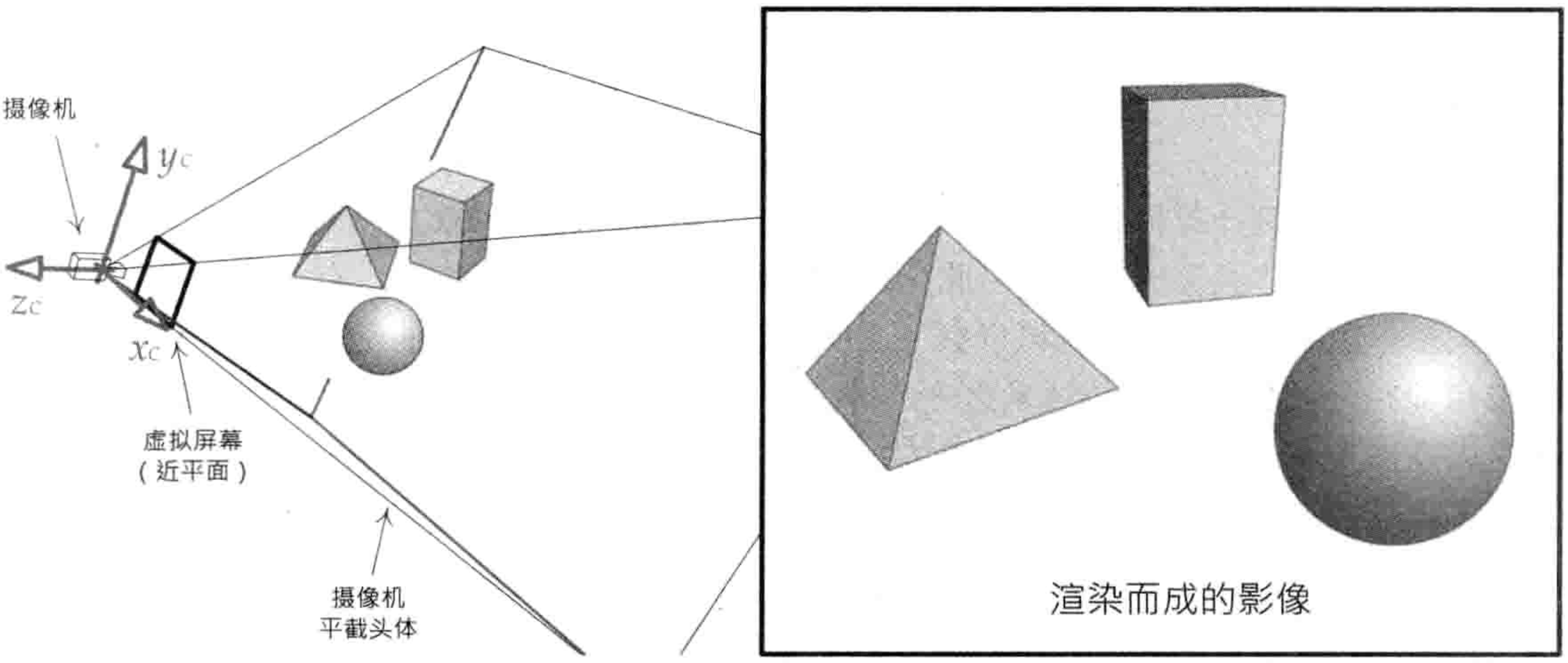


图 10.1: 几乎所有三维计算机图形技术，在高层次来看都会使用这种渲染方法。

有多种不同的技术可运行上述的基本渲染步骤。游戏图形一般是以**照相写实主义**（photorealism）为主要目标，但也有些游戏以特别风格为目标（如卡通、炭笔素描等）。因此，渲染工程师和美术人员通常会把场景的属性描述得尽量真实，并使用尽量接近物理现实的光传输模型（light transport model）。在此语境下，整个渲染技术的领域，包含为了视觉流畅而设计的实时渲染技术，以及为照相写实主义而设计但非实时运行的渲染技术。

实时渲染引擎重复地进行上述的步骤，以每秒30、50或60帧的速度显示渲染出来的影像，从而产生运动的错觉。换句话说，实时渲染引擎以最长33.3ms内产生每幅影像（以达至30FPS的帧率）。通常实际上可用的时间更少，因为其他如动画、人工智能、碰撞检测、物理模拟、音频、玩家机制、其他游戏性等引擎系统都会耗费时间资源。对比电影渲染引擎通常要花许多分钟以至于许多小时来渲染1帧，现时实时计算机图形的品质可谓非常惊人。

10.1.1 场景描述

现实世界的场景由物体所组成。有些物体是固态的，例如一块砖头，有些物体无固定形

状，例如一缕烟，但所有物体都占据三维空间的体积。物体可以是**不透明的**（opaque），即光不能通过该物体；也可以是**透明的**（transparent），即光能通过该物体，过程中不被散射（scatter），因此可以看见物体后面的清晰影像[1]；还可以是**半透明的**（translucent），即光能通过该物体，但过称中会被散射至各个方向，使物体背后的影像变得朦胧。

渲染不透明物体时，只需要考虑其**表面**（surface）。我们无须知道不透明物体内部是怎样的，便足以渲染该物体，因为光能不穿越其表面。当渲染透明或半透明物体时，便需要为光线通过物体时所造成的反射、折射、散射、吸收行为建模。此模型需要该物体内部结构及属性的知识。然而，多数游戏引擎不会达至这么麻烦的地步。游戏引擎通常只会用跟渲染不透明物体差不多的方法，去渲染透明和半透明物体。游戏引擎通常会采用名为**alpha**的简单不透明度（opacity）量度数值表达物体表面有多不透明或透明。此方法能导致多种视觉异常情况（例如，物体离摄像机较远的一面可能渲染得不正确），但可采用近似法来使大部分情况看上去都足够真实。就算是烟这种无固定形状的物体，通常也会用粒子效果去表现，而这些效果实际上是由大量半透明的矩形卡板所合成的。因此，我们完全可以说，大多数游戏渲染引擎主要着重于渲染物体的**表面**。

10.1.1.1 高端渲染软件所用的表示法

理论上，一块表面是由无数三维空间中的点所组成的一张二维薄片。然而，此描述显然无实际用途。为了让计算机处理及渲染任意的表面，我们需要以一个紧凑的方式用数学表示表面。

有些表面可用分析式来精确表示[2]。例如，位于原点的球体表面可用$x^2 + y^2 + z^2 = r^2$表示。然而，为任意形状建模时，分析式的方程并非十分有用。

在电影产业里，表面通常由一些矩形的**面片**（patch）所组成，而每个面片则是由小量的控制点定义的三维样条（spline）所构成的。可使用多种样条，包括各Bézier曲面（如双三次面片/bicubic patch，是一种三阶Bézier曲面[3]）、非均匀有理B样条（nonuniform rational B-spline/NURBS）[4]、N面片（N-patches，又名为**normal patches**）[5]。用面片建模，有点像用小块的长方形布或纸糊[6]去遮盖一个雕像。

---

[1]译注：此处有一点矛盾。如果物体是完全透明的，它不和光线有互动的话，那么完全可以在渲染过程中予以忽略。在真实物理中，所有物质都不完全透明的。例如，空气也不是完全透明，白昼天空的颜色也是由于太阳光线被大气散射所造成的。

[2]译注：原文还有“使用参数式表面方程（using parametric surface equation）”一段。因为后面举的例子并非这种方程，而是隐式表面方程（implicit surface equation），故删之。

[3]http://en.wikipedia.org/wiki/Bezire_surface

[4]http://en.wikipedia.org/wiki/Nurbs

[5]http://www.gamasutra.com/view/feature/2980/b%C3%A9zier_triangles_and_npatches.php

[6]译注：原文为paper maché（Papier-mâché），指用碎纸糊贴去造立体作品的艺术手法。

高端电影渲染引擎如Pixar的RenderMan，采用**细分曲面**（subdivision surface）定义几何形状。每个表面由控制多边形网格（如同样条）表示表面，但这些多边形会使用Catmull-Clark算法逐步细分成更小的多边形。细分过程通常会进行至每个多边形小于像素的大小。此方法的优点是，无论摄像机距离表面有多接近，都能再细分多边形，使轮廓边线显得圆滑。关于细分曲面可参阅Gamasutra的这篇文章[7]。

10.1.1.2 三角形网格

传统来说，游戏开发者会使用三角形网格来为表面建模。三角形是表面的分段线性逼近（piecewise linear approximation），如同用多条相连的线段分段逼近一个函数或曲线（见图10.2）。

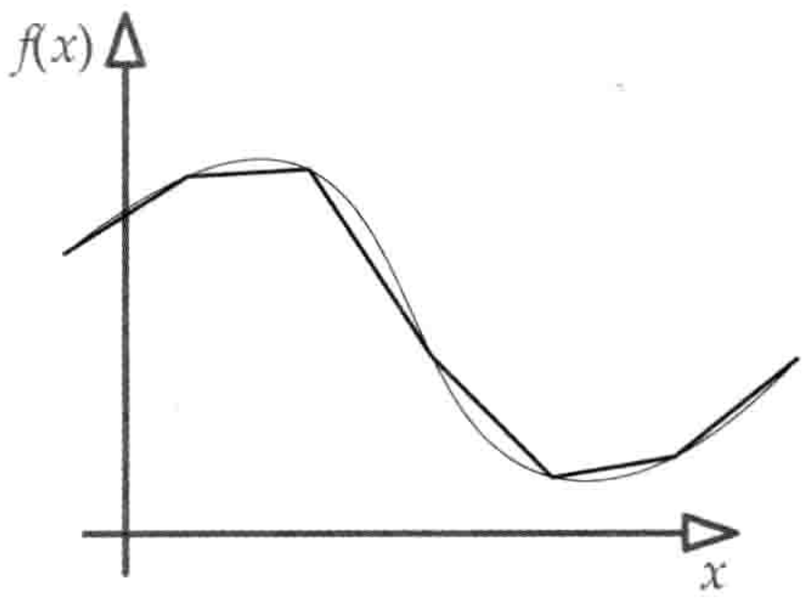


图 10.2: 三角形网格是表面的线性逼近，如同以一系列互连的线段去线性逼近曲线或函数。

在各种多边形中，实时渲染之所以选用三角形，是因为三角形有以下的优点。

- **三角形是最简单的多边形**。少于3个顶点就不能成为一个表面。
- **三角形必然是平坦的**。含4个或以上顶点的多边形不一定是平坦的，因为其前3个顶点能定义一个平面，第4个顶点或许会位于该平面之上或之下。
- **三角形经多种转换之后仍然维持是三角形，这对于仿射转换和透视转换也成立**。最坏的情况下，从三角形的边去观看，三角形会退化为线段。在其他角度观察，仍能维持是三角形。
- **几乎所有商用图形加速硬件都是为三角形光栅化而设计的**。从最早期的PC三维图形加速器开始，渲染硬件一直几乎只专注为三角形光栅化而设计。此决策还可追溯至最早期使用软件光栅化的三维游戏，如《德军师令部》和《毁灭战士》。无论个人喜恶，基于三角形的技术已牢牢确立在游戏业界，在未来几年应该还不会有大转变。

[7] http://www.gamasutra.com/view/feature/3177/subdivision_surface_theory.php

镶嵌

镶嵌（tessellation）是指把表面分割为一组离散多边形的过程，这些多边形通常是三角形或四边形（quadrilateral，简称**quad**）。**三角化**（triangulation）专指把表面镶嵌为三角形。

这种三角形网格在游戏中有一常见问题，就是其镶嵌程度是由制作的美术人员决定的，不能中途改变。固定的镶嵌会使物体的**轮廓边缘**显得不圆滑（图10.3），此问题在摄像机接近物体的时候更加明显。

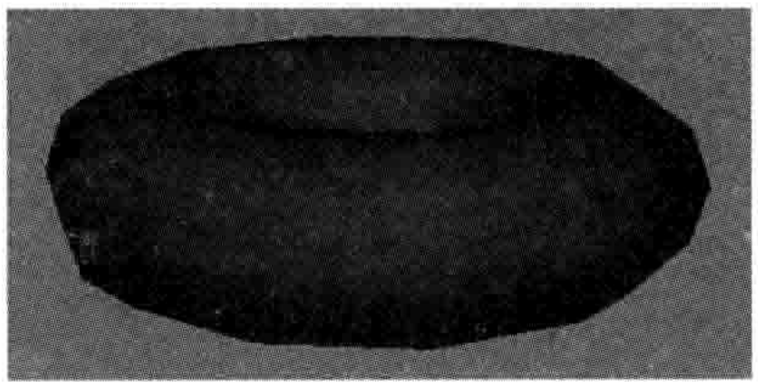


图 10.3: 固定的镶嵌会使物体的轮廓边缘显得块状，尤其当物体接近摄像机之时。

理想地，我们希望有一方案能按物体与虚拟摄像机距离的缩减而增加密辅程度。换句话说，我们希望无论物体是远是近，都能有一致的三角形对像素密度。细分曲面能满足此愿望，表面能根据与摄像机的距离来进行镶嵌，使每个三角形的尺寸都少于一个像素。

游戏开发者经常尝试以一串不同版本的三角形网格链去逼近此理想的三角形对像素密度，每一版本称为一个**层次细节**（level-of-detail，LOD）。第一个LOD通常称为LOD 0，代表最高程度的镶嵌，在物体非常接近摄像机时使用。后续的LOD的镶嵌程度不断降低（图10.4）。当物体逐渐远离摄像机，引擎就会把网格从LOD 0换为LOD 1、LOD 2等。这样渲染引擎便可以花费更多时间在接近摄像机的物体上（即占据屏幕中更多像素的物体），进行顶点的转换和光照运算。

有些游戏引擎会应用**动态镶嵌**（dynamic tessellation）技术到可扩展的网格上，例如水面和地形。在这种技术中，网格通常以高度场（height field）来表示，而高度场则在某种规则栅格模式上定义。最接近摄像机的网格区域会以栅格的最高分辨率来镶嵌，距摄像机较远的区域则会使用更少的栅格点来进行镶嵌。

渐进网格（progressive mesh）是另一种动态镶嵌及层次细节技术。运用此技术时，当物体很接近摄像机时采用单个最高分辨率网格。（这个本质上就是LOD 0网格。）当物体自摄像机远离，这个网格就会自动密辅，其方法是把某些棱收缩为点。此过程能自动生成半连续的LOD链。关于渐进网格技术的详细讨论，可参阅此文[8]。

[8] http://research.microsoft.com/en-us/um/people/hoppe/pm.pdf

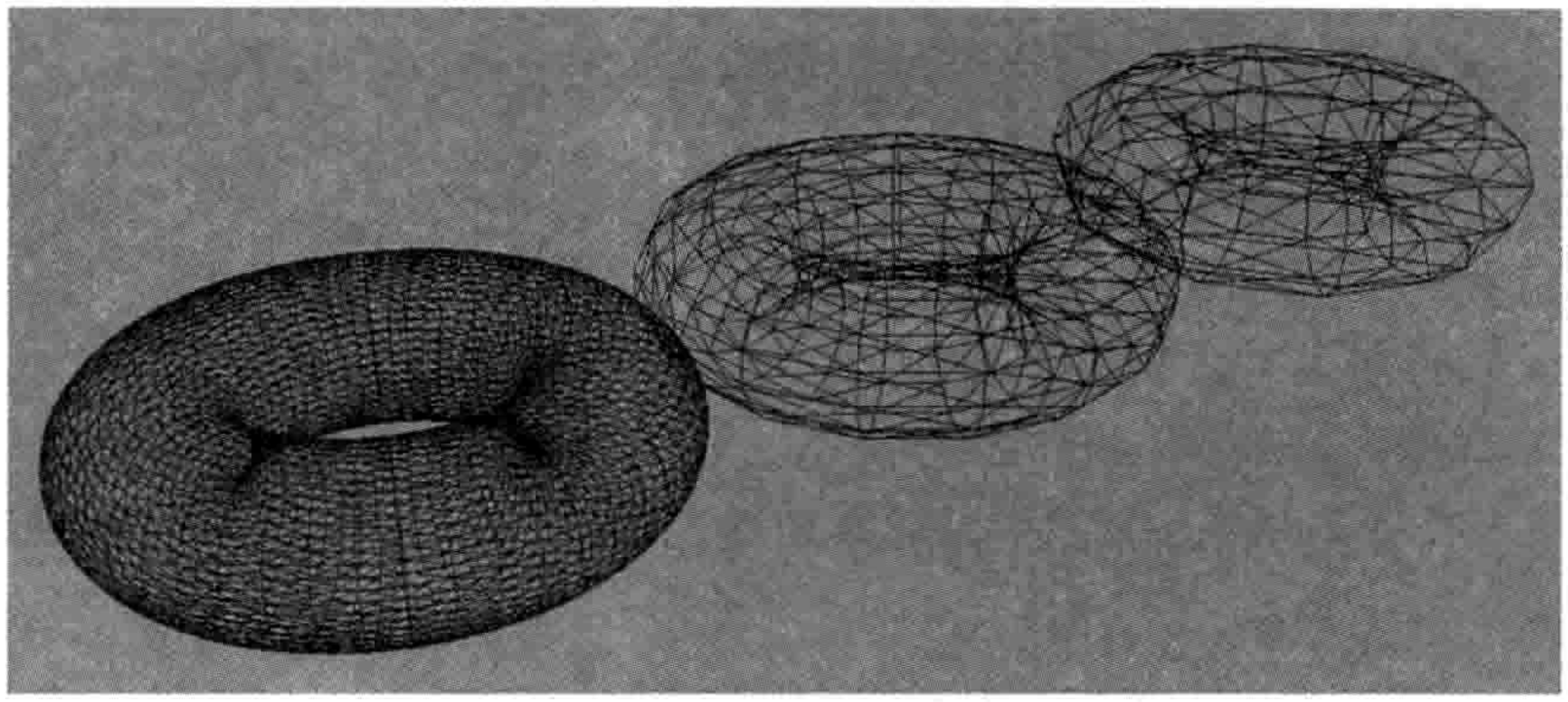


图 10.4: 一串不同LOD的网格，每个有固定的镶嵌程度，可用于逼近均匀的三角形对像素比。左方的环形（torus）由5000个三角形组成，中间的是450个，右方的是200个。

10.1.1.3 构造三角形网格

我们已理解三角形网格是什么以及为什么使用它们，再看看如何构造三角形网格。

缠绕顺序

三角形由3个顶点的位置矢量定义，此3个矢量设为$\mathbf{p}_1$、$\mathbf{p}_2$、$\mathbf{p}_3$。每条棱（edge）的相邻顶点的位置矢量相减，就能求得3条棱的矢量。例如：

$$\mathbf{e}_{12} = \mathbf{p}_2 - \mathbf{p}_1$$

$$\mathbf{e}_{13} = \mathbf{p}_3 - \mathbf{p}_1$$

$$\mathbf{e}_{23} = \mathbf{p}_3 - \mathbf{p}_2$$

任何两棱的叉积，归一化后就能定义为三角形的单位**面法线**（face normal）$\mathbf{N}$:

$$\mathbf{N} = \frac{\mathbf{e}_{12} \times \mathbf{e}_{13}}{|\mathbf{e}_{12} \times \mathbf{e}_{13}|}$$

图10.5描绘了这些推导。要知道面法线的方向（即棱叉积的目的），我们需要定义哪一面才是三角形的正面（即物体表面），哪一面是背面（即表面之内）。这个可以简单用**缠绕顺序**（winding order）来定义，缠绕顺序用来定义表面方向有两种方式，分别是顺时针方向（clockwise，CW）和逆时针方向（counterclockwise，CCW）。[9]

[9]译注：wind作为动词是指缠绕。可这么想象，把3根有序号的钉子打在墙上，然后用绳子按序号缠绕钉子来做出三角形。

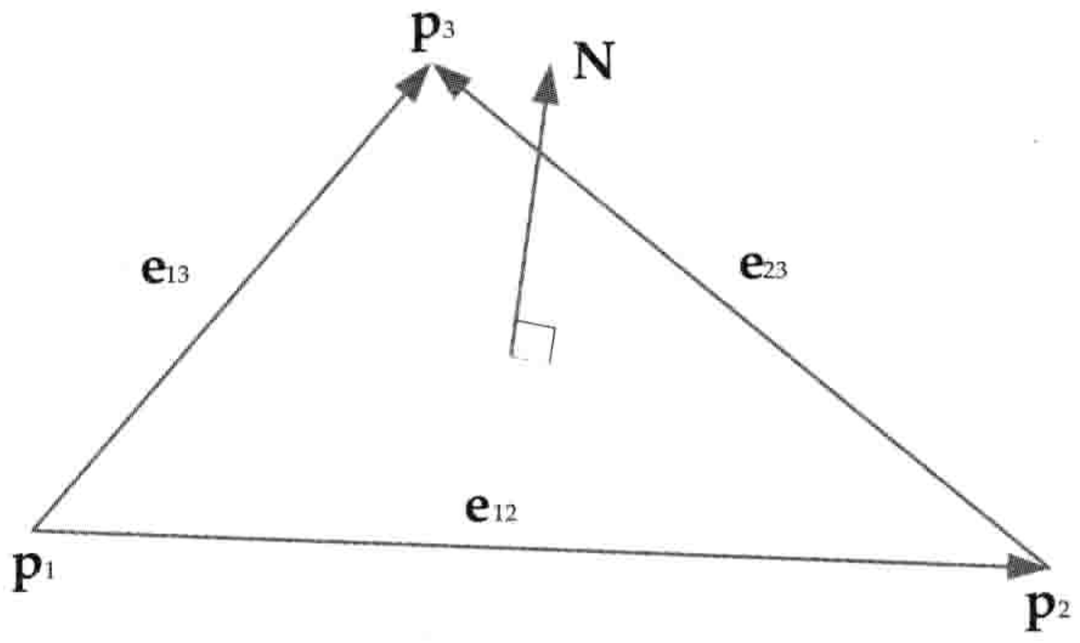


图 10.5: 从三角形的顶点推导其棱和平面。

多数底层图形API提供基于缠绕顺序来**剔除**背面三角形（backface triangle culling）。例如，若在Direct3D内把剔除模式参数（D3DRS_CULL）设置为D3DCULLMODE_CW，那么所有在屏幕空间里缠绕顺序为顺时针方向的三角形就会视为背面，不被渲染。

背面剔除的重要性在于，我们通常不需要浪费时间渲染看不见的三角形。而且，渲染透明物体的背面还会做成视觉异常。可以随意选择两种缠绕顺序之一，只要整个游戏的资产都是一致的就行。不一致的缠绕顺序是三维建模新手的常见错误。

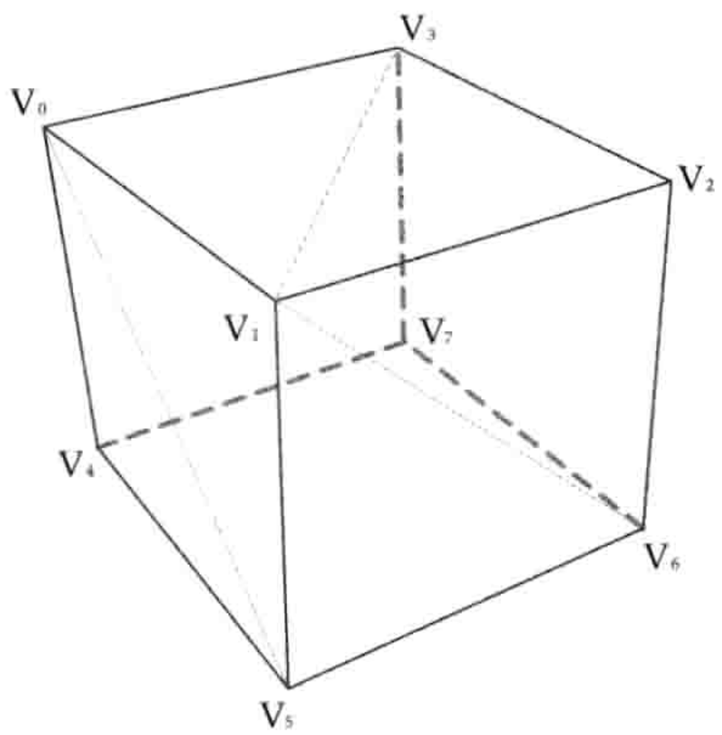
三角形表

定义网格的最简单方法是以每3个顶点为一组列举，当中每3个顶点对应一个三角形。此数据结构称为**三角形表**（triangle list），如图10.6所示。

索引化三角形表

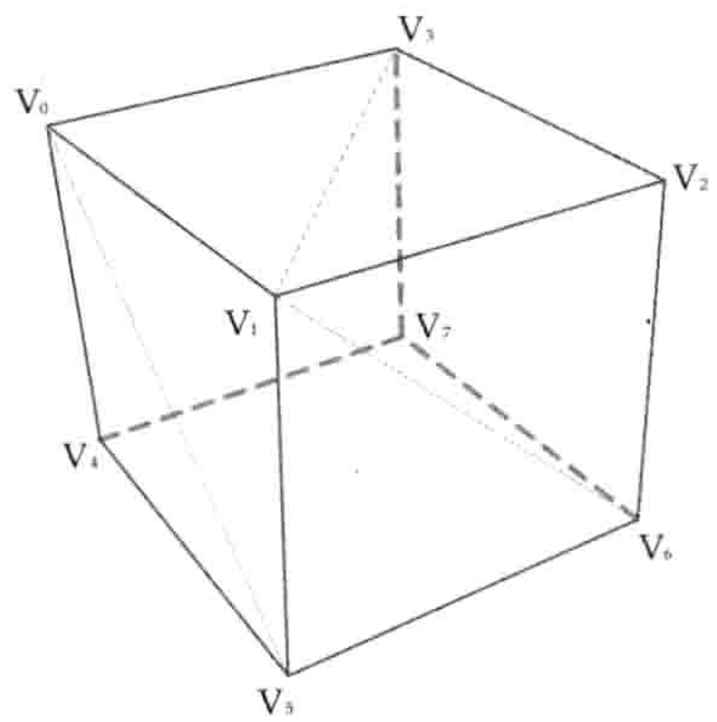
读者可能注意到，在图10.6的三角形表中有许多重复的顶点，而且经常重复多次。之后在10.1.2.1节会谈及，每个顶点要储存颇多的元数据，因此在三角形表中重复的数据会浪费内存。这同时也会浪费GPU的资源，因为重复的顶点会计算变换及光照多次。

由于上述原因，多数渲染引擎会采用更有效率的数据结构——**索引化三角形表**（indexed triangle list）。其基本思想就是每个顶点仅列举一次，然后用轻量级的顶点**索引**（通常每个索引只占16位）来定义组成三角形的3个顶点。在DirectX下顶点储存于**顶点缓冲**（vertex buffer），在OpenGL下则称其为**顶点数组**（vertex array）。而索引会储存于另一单独缓冲，称为**索引缓冲**（index buffer）或**索引数组**（index array）。图10.7展示了此数据结构。



V_0	V_1	V_3	V_1	V_2	V_3	V_0	V_5	V_1	...	V_5	V_7	V_6

图 10.6: 三角形表。



顶点	V_0	V_1	V_2	V_3	V_4	V_5	V_6	V_7

索引	0	1	3	1	2	3	0	5	1	...	5	7	6

图 10.7: 索引化三角形表。

三角形带及三角形扇

在游戏渲染中，有时候还会用到两种特殊网格数据结构，分别为**三角形带**（triangle strip）及**三角形扇**（triangle fan）。这两种数据结构不需要索引缓冲，但同时能降低某程度的顶点重复。它们之所以有这些特性，其实是通过预先定义顶点出现的次序，并预先定义顶点组合成三角形的规则。

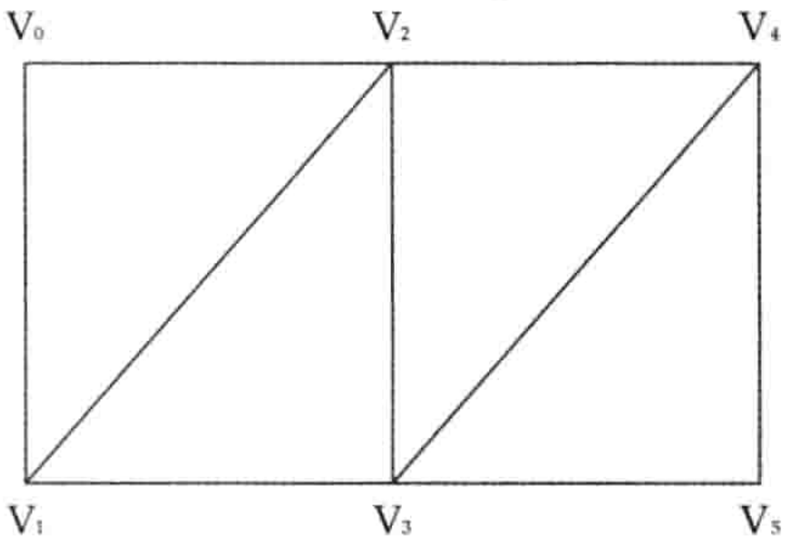
在三角形带中，前3个顶点定义了第一个三角形。之后的每个顶点都会连接其前两个顶点，产生全新的三角形。为了统一三角形带的缠绕顺序，产生每个新三角形时，其前两个相邻顶点会互换次序。图10.8展示了一个三角形带的例子。

在三角形扇中，前3个顶点定义了第一个三角形，之后每个顶点与前一顶点及该三角形扇的首顶点组成三角形。图10.9是三角形扇的例子。

顶点缓存优化

当GPU处理索引化三角形表时，每个三角形能引用顶点缓冲内的任何顶点。为了在光栅化阶段保持三角形的完整性，顶点必须按照其位于三角形中的次序来处理。当顶点着色器处理每个顶点后，其结果会被缓存以供重复使用。若之后的图元引用到存于缓存的顶点，就能直接使用结果，而无须重复处理该顶点。

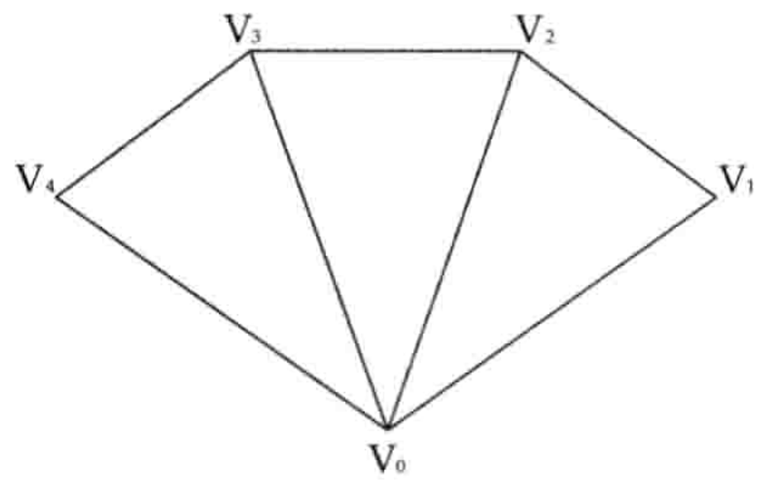
使用三角形带及三角形扇，一个原因是能节省内存（无须索引缓冲），另一原因是基于它们往往能改善GPU存取显存时的缓存一致性（cache coherency）。我们甚至可以使用**索**



顶点	V_0	V_1	V_2	V_3	V_4	V_5

解读成三角形	**0 1 2**	1 **3** 2	2 3 **4**	3 **5** 4

图 10.8: 三角形带。



顶点	V_0	V_1	V_2	V_3	V_4

解读成三角形	**0 1 2**	0 2 **3**	0 3 **4**

图 10.9: 三角形扇。

引化三角形带及**索引化三角形扇**以消除所有顶点重复（这样通常比不用索引缓冲更省内存），而同时仍能受益于三角形带及三角形扇次序所带来的缓存一致性。

除了次序受限的三角形带及三角形扇，我们也可以优化索引化三角形表以提升缓存一致性。**顶点缓存优化器**（vertex cache optimizer）就是为此而设的一种离线几何处理工具，它能重新排列三角形的次序，以优化缓存内的顶点复用。顶点缓存优化器一般会根据多种因素来进行优化，例如个别GPU类型的顶点缓存大小、GPU选择缓存或舍弃顶点的算法等。以Sony的Edge几何处理库为例，其顶点缓存优化器能使三角形表的渲染吞吐量达至高于三角形带的4%。

10.1.1.4 模型空间

三角形网格的位置矢量，通常会被指定于一个便利的局部坐标系，此坐标系可称为**模型空间**（model space）、**局部空间**（local space）或**物体空间**（object space）。模型空间的原点一般不是物体中心，便是某个便利的位置，例如，角色脚掌所在地板的位置、车辆轮子在地上的水平质心（centroid）。

如4.3.9.1节提及，模型空间的轴可随意设置，但这些轴通常会和自然的“前方”、“左/右方”及“上方”对齐。在数学上再严谨一些的话，可以定义3个单位矢量**F**、**L**（或**R**）、**U**，并把这3个矢量映射至模型空间的单位基矢量**i**、**j**、**k**（即各自对应x、y、z轴）。例如，一个常见的映射为$\mathbf{L}=\mathbf{i}$、$\mathbf{U}=\mathbf{j}$、$\mathbf{F}=\mathbf{k}$。这些映射可以随意设定，只要引擎中所有模型的映射都是始终如一的。图10.10展示了一架飞机的模型空间轴的可行映射之一。

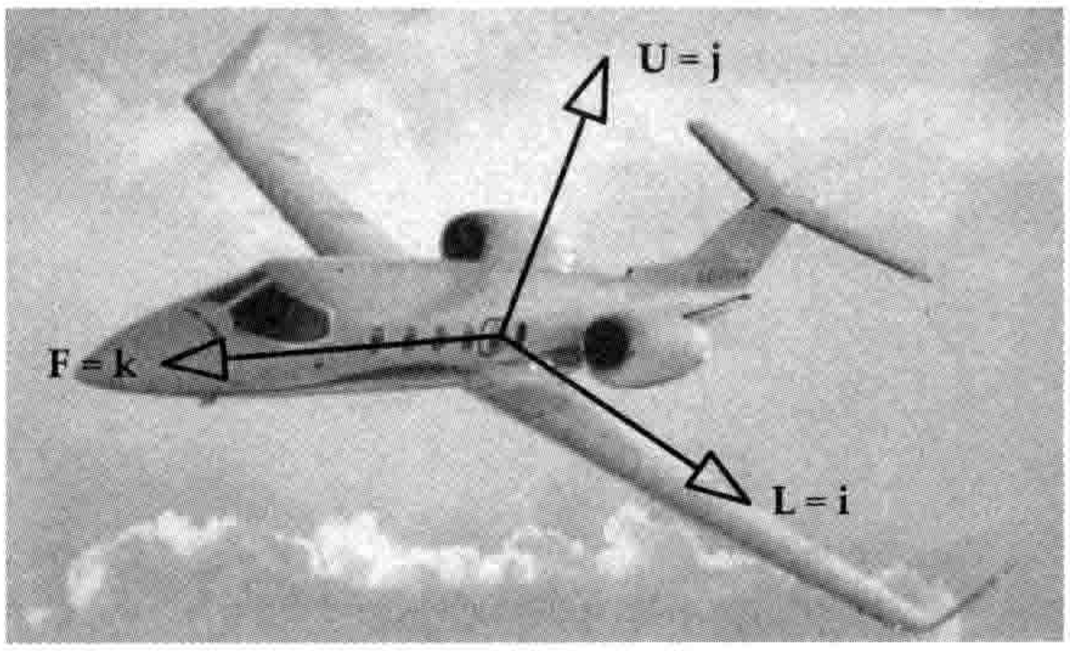


图 10.10: 模型空间轴的可行映射之一。

10.1.1.5 世界空间及网格实例化

使用网格组成完整场景时，会在一个共同的坐标系里放置及定向多个网格，此坐标系称为**世界空间**（world space）。每个网格可在场景中多次出现，例如，街上排列着同款街灯、一堆看不见面目的士兵、攻击玩家的一大群蜘蛛等。每个这些物体称为**网格实例**（mesh instance）。

每个网格实例含共享网格的引用，此外也包含一个变换矩阵，用以把个别实例的网格顶点从模型空间转换至世界空间。此矩阵名为**模型至世界**矩阵（model-to-world matrix），有时候仅简单称为**世界矩阵**（world matrix）。若采用4.3.10.2节的表示方式，此矩阵可写成：

$$\mathbf{M}_{M\to W} = \begin{bmatrix} (\mathbf{RS})_{M\to W} & \mathbf{0} \\ \mathbf{t}_M & 1 \end{bmatrix}$$

当中左上的3×3矩阵$(\mathbf{RS})_{M\to W}$用来旋转和缩放模型空间顶点至世界空间，而$\mathbf{t}_M$则是模型空间轴在世界空间的位移。若用世界空间坐标表示单位模型空间的基矢量$\mathbf{i}_M$、$\mathbf{j}_M$、$\mathbf{k}_M$，则该矩阵也可写成：

$$\mathbf{M}_{M\to W} = \left[\begin{array}{c|c} \mathbf{i}_M & 0 \\ \mathbf{j}_M & 0 \\ \mathbf{k}_M & 0 \\ \hline \mathbf{t}_M & 1 \end{array}\right]$$

给定一个模型空间的顶点坐标，渲染引擎会用以下的方程计算其世界空间坐标：

$$\mathbf{v}_W = \mathbf{v}_M \mathbf{M}_{M\to W}$$

$\mathbf{M}_{M\to W}$可以看成是模型空间轴的位置和定向的描述，此描述是以世界空间坐标表示的。或是把它看成是把顶点从模型空间变换至世界空间的矩阵。

当渲染模型时，模型至世界矩阵也可用来变换网格的表面法矢量（见10.1.2.1节）。4.3.11节曾提及，为了正确变换法矢量，必须把法矢量乘以模型至世界矩阵的逆转置矩阵。而4.3.6.1节提及，若矩阵不含缩放及切变，可简单地把法矢量的w设为0，再乘以模型至世界矩阵完成变换。

有些网格是完全静止及独一无二的，例如建筑物、地形，以及其他背景元素。这些网格的顶点通常以世界空间表示，因此其模型至世界矩阵是单位矩阵，可以忽略。

10.1.2 描述表面的视觉性质

为了正确地渲染及照明表面，我们需要有描述表面的视觉性质（visual property）。表面性质包括几何信息，例如表面上不同位置的法矢量。表面性质也包括描述光和表面交互作用的方式，包括漫反射颜色（diffuse color）、粗糙度（roughness）/光滑度（shininess）、反射率（reflectivity）、纹理、透明度/不透明度、折射率（refractive index）等。表面性质也可能含有表面随时间变化的描述（例如，有动画的角色皮肤应如何追踪其骨骼的关节，水面如何移动等）。

渲染照相写实影像的关键在于，正确地模拟光和场景中物体交互作用时的行为。因此渲染工程师需要理解光如何工作、光如何在环境中传递，以及虚拟摄像机如何“感光”，并把结果转换成屏幕上像素的颜色。

10.1.2.1 光和颜色的概论

光是电磁辐射，在不同情况下其行为既像波也像粒子。光的颜色是由其**强度**（intensity）I及**波长**（wavelength）λ（或频率$f = 1/\lambda$）所决定。可见光的波长范围是740nm～380nm（频率是430THz～750THz）。一束光线可能含单一纯波长，这即是彩虹的颜色，又称为**光谱颜色**（spectral color）。或是，一束光线可能由多种波长的光混合而成。我们可以把一束光线中各波长的强度绘成图表，这种图称为**光谱图**（spectral plot）。白光含所有波长，因此其光谱图大约像一个矩形，横跨整个可见光波段[10]。纯绿光则只有一个波长，因此其光谱图会显示在570THz有一个极窄的尖峰。

[10]译注：此处的描述可能不太正确。事实上，日常用语中的“白光”和人类视觉系统相关，不同的波长组合也可以产生视觉上的“白光”。而应用上通常以黑体辐射（black body radiation）的温度——即色温（color temperature）——来定义“白光”。例如，某个屏幕发出的白光是6500K，表示其颜色由人类看上去像绝对温度为6500K黑体发射出来的光。

光和物体的交互作用

光和物质之间能有许多复杂的交互作用（interaction）。光的行为，部分是由其穿过的**介质**（medium）所控制的，部分是由两种不同介质（如空气/固体、空气/水、水/玻璃等）之间的**界面**（interface）所控制的。从技术上来说，一个表面只不过是两种不同介质的界面。

不管光的行为有多复杂，其实光只能做4件事[11]。

- 光可被**吸收**（absorb）。
- 光可被**反射**（reflect）。
- 光可在物体中**传播**（transmit），过程中通常会被**折射**（refract）。
- 通过很窄的缺口时，光会被**衍射**（diffract）。

多数照相写实渲染引擎会处理以上前3项行为，而衍射通常会被忽略，因为在多数场景中衍射的效果并不明显。

一个平面只会吸收某些波长的光，其他波长的光会被反射。这个特性形成我们对物体颜色的感知（perception）。例如，若白光照射一个物体，红色以外的所有波长被吸收，那么该物体就显得是红色的。同样的感知效果会出现在红光照射在白色物体上，我们的眼睛无法区分这两种情况。[12]

光的反射可以是**漫反射**（diffuse），这是指入射光会往所有方向平均散射。而反射也可以是**镜面反射**（specular），这是指入射光会直接被反射，或在反射时展开成很窄的锥形。反射可以是**各向异性**（anisotropic）的，这是指在不同角度观察表面时光的反射有所不同。

当光穿过物体时，光可能会被**散射**（如半透明物质），部分被**吸收**（如彩色玻璃），或被**折射**（如三棱镜）。不同波长的光折射角度会有差异，产生散开的光谱。这就是光经过雨点或三棱镜能产生彩虹的原因。光也能进入半固态的表面，在表面下反弹，再从另一个位置离开表面。这个现象称为**次表面散射**（subsurface scattering，SSS）。此效果能使皮肤、蜡、大理石等物质显示其柔和的特性。

---

[11]译注：此外，光的行为还有干涉（interference）、极化（polarization）等。干涉现象产生光盘的七彩颜色，一般在计算机图形学中要使用特殊的着色模型才能模拟。而极化在计算机图形学中通常会被忽略。

[12]译注：如译者之前所注，“白光”的定义是比较模糊的；然而，理想的白色物体却能够定义为完全反照所有可见光波长的物体。自然界中反照率（albedo）最高的物体是雪，新雪的反照率能高达0.9。相反，理想的黑色物体就是反照率为0的物体（即黑体），自然界的例子是沥青，可低至0.04。

颜色空间和颜色模型

颜色模型（color model）是量度颜色的三维坐标系统。而**颜色空间**（color space）是一个具体标准，描述某颜色空间内的数值化颜色如何映射至人类在真实世界中看到的颜色。颜色模型通常是三维的，原因是我们眼睛里有3种颜色感应器（锥状细胞），每种感应器对不同波长的光敏感。

计算机图形学中最常用的颜色模型是RGB模型。此模型中，由一个单位立方体表示颜色空间，其3个轴分别代表红、绿、蓝光的量度。这些红、绿、蓝分量称为**颜色通道**（color channel）。在标准的RGB颜色模型中，每个颜色通道的范围都是0～1。因此，颜色$(0,0,0)$代表黑色，$(1,1,1)$则代表白色。

当颜色存储于位图时，可使用多种不同的颜色格式（color format）。颜色格式的定义，部分由**每像素位数**（bits per pixel，BPP）决定，更具体地说，是由表示每颜色通道的位数决定的。RGB888格式使用每颜色通道8位，共24位/像素。此格式中，每个通道的范围是0～255，而非0～1。在RGB565中，红色和蓝色使用5位，绿色使用6位，总共16位/像素。调色板格式（paletted format）可使用每像素8位储存索引，再用这些索引查找一个含256色的调色板，调色板的每笔记录可能存储为RGB888或其他合适的格式。

在三维渲染中，还会用到其他一些颜色模型。在10.3.1.5节会介绍如何使用对数LUV颜色模型做**高动态范围**（high dynamic range，HDR）渲染。

不透明度和alpha通道

常会在RGB颜色矢量之后再补上一个名为**alpha**的通道。如10.1.1节曾提及，alpha值用来量度物体的不透明度。当存储为像素时，alpha代表该像素的不透明度。

RGB颜色格式可扩展以包含alpha通道，那时候就会称为RGBA或ARGB颜色格式。例如，RGBA8888是每像素32位的格式，红、绿、蓝、alpha都使用8位。又例如，RGBA5551是16位格式，含1位alpha。此格式中，颜色只能指定为完全不透明或完全透明。

10.1.2.2 顶点属性

要描述表面的视觉特性，最简单的方法就是把这些特性记录在表面的离散点上。网格的顶点是存储表面特性的便利位置，这种存储方式称为**顶点属性**（vertex attribute）。

一个典型的三角形网格中，每个顶点包含部分或全部以下所列举的属性。身为渲染工程师，我们当然能自由地定义额外所需的属性，达致在屏幕上想要的视觉效果。

- **位置矢量**（position vector） $\mathbf{p}_i = [p_{ix} \quad p_{iy} \quad p_{iz}]$：这是网格中第$i$个顶点的三维位置。位置矢量通常以物体局部空间的坐标表示，此空间名为**模型空间**（model space）。
- **顶点法矢量**（vertex normal） $\mathbf{n}_i = [n_{ix} \quad n_{iy} \quad n_{iz}]$：这是顶点$i$位置上的表面单位矢量。顶点法矢量用于每顶点动态光照（per-vertex dynamic lighting）的计算。
- **顶点切线矢量**（vertex tangent） $\mathbf{t}_i = [t_{ix} \quad t_{iy} \quad t_{iz}]$：这是和**顶点副切线矢量**（vertex bitangent）$\mathbf{b}_i = [b_{ix} \quad b_{iy} \quad b_{iz}]$互相垂直的单位矢量，它们也同时垂直于顶点法矢量$\mathbf{n}_i$。这3个矢量$\mathbf{n}_i$、$\mathbf{t}_i$、$\mathbf{b}_i$能一起定义称为**切线空间**（tangent space）的坐标轴。此空间能用于计算多种逐像素光照（per-pixel lighting），例如法线贴图（normal mapping）及环境贴图（environment mapping）。（副切线矢量有时候被称为**副法矢量**（binormal），尽管它并非垂直于表面。[13]）
- **漫反射颜色**（diffuse color） $\mathbf{d}_i = [d_{Ri} \quad d_{Gi} \quad d_{Bi} \quad d_{Ai}]$：漫反射颜色是一个四元素矢量，以RGB颜色空间描述表面的漫反射颜色。此顶点属性通常附有不透明度，即**alpha**（A）。此颜色可能在脱机时计算（静态光照），或运行时计算（动态光照）。
- **镜面颜色**（specular color） $\mathbf{s}_i = [s_{Ri} \quad s_{Gi} \quad s_{Bi} \quad s_{Ai}]$：当光线由光滑表面反射至虚拟摄像机影像平面，这个矢量就是描述其镜面高光的颜色。
- **纹理坐标**（texture coordinates） $\mathbf{u}_{ij} = [u_{ij} \quad v_{ij}]$：用来把二维（有时候三维）的位图“收缩包裹”网格的表面，此过程称为**纹理贴图**（texture mapping）。纹理坐标(u, v)描述某顶点在纹理二维正规化坐标空间里的位置。每个三角形可贴上多张纹理，因此网格可以有超过一组纹理坐标。我们采用下标j去表示不同的纹理坐标组。
- **蒙皮权重**（skinning weight） (k_{ij}, w_{ij})：在骨骼动画里，网格的顶点依附在骨骼的个别关节之上。这种情况下，每个顶点需指明其依附着的关节索引k。另一种情况是，一个顶点受多个关节所影响，最终的顶点位置变为这些影响的**加权平均**（weighted average）。我们把每个关节的影响以权重因子w表示。概括地说，顶点i可由多个关节j所影响，每个影响关系可存储为两个数值$[k_{ij}, w_{ij}]$。

10.1.2.3 顶点格式

顶点属性通常储存于如C struct或C++ class的数据结构。这样的数据结构的布局称为**顶点格式**（vertex format）。不同的网格需要不同的属性组合，因而需要不同的顶点格式。以下是一些常见的顶点格式例子：

---

[13]译注：有些意见认为（如http://www.terathon.com/code/tangent.html的Bitangent versus Binormal一节），图形学中常出现的副法矢量（binormal）实际上是错误地使用了微分几何中用来描述三维曲线的弗莱纳标架（Frenet frame）所采用的术语。更恰当的术语应该是逼切线矢量（bitangent）。

```
// 最简单的顶点，只含位置。(可用于阴影体伸展/shadow volume extrusion、
// 卡通渲染中的轮廓棱检测/silhouette edge detection、z预渲染/z-prepass等)
struct Vertex1P
{
    Vector3     m_p;            // 位置
};

// 典型的顶点格式，含位置、顶点法线及一组纹理坐标
struct Vertex1P1N1UV
{
    Vector3     m_p;            // 位置
    Vector3     m_n;            // 顶点法矢量
    F32         m_uv[2];        // (u, v) 纹理坐标
};

// 蒙皮用的顶点，含位置、漫反射颜色、镜面反射颜色及4个关节权重
struct Vertex1P1D1S2UV4J
{
    Vector3     m_p;            // 位置
    Color4      m_d;            // 漫反射颜色及透明度
    Color4      m_S;            // 镜面反射颜色
    F32         m_uv0[2];       // 第1组纹理坐标
    F32         m_uv1[2];       // 第2组纹理坐标
    U8          m_k[4];         // 蒙皮用的4个关节索引及
    F32         m_w[3];         // 3个关节权重(第4个由其他3个求得)
};
```

显然，顶点属性的可行排列数目，以至于不同的顶点格式数目，都可增长至非常庞大。(实际上，若能使用任意数目的纹理坐标或关节权重，格式数目在理论上是无上限的。)管理所有这些顶点格式，经常是图形程序员的头痛之源。

以下列举一些步骤，可以减少引擎所需支持的顶点格式数目。在实际的图形应用中，许多理论上可行的顶点格式根本无用，或不受图形硬件或游戏的着色器的支持。有些游戏团队会设置限制，仅使用一组有用并可行的顶点格式，以便管理。例如，团队可能只容许每个顶点使用0、2或4个关节权重，又或决定只支持最多两组纹理坐标。现在的GPU能够从顶点数据结构中抽取部分属性，因此有些游戏团队也会选择给所有网格使用单一的“über格式”[14]，然后按着色器所需选取相关的属性。

[14]译注：这里是指包含所有所需顶点属性并集的顶点格式。业界比较少用über一词来形容这种顶点格式。实际上这种做法为了简单而牺牲了内存容量及内存带宽。

10.1.2.4 属性插值

三角形顶点的属性仅仅是整个表面的视觉特性的粗糙、离散近似值。当渲染三角形时，重要的是三角形内点的视觉特性，这些内点最终成为屏幕上的像素。换言之，我们需要取得**每像素**（per-pixel）的属性，而非每顶点（per-vertex）的。

要取得网格表面的每像素属性，最简单的方法是对每顶点属性进行**线性插值**（linear interpolation）[15]。当把线性插值施于顶点颜色，这种属性插值便称为**高氏着色法**（Gouraud shading）。图10.11是以高氏着色法渲染三角形的例子，图10.12则是把它应用在三角形网格时的效果。插值法也会应用至其他各种顶点属性，例如，顶点法矢量、纹理坐标、深度等。

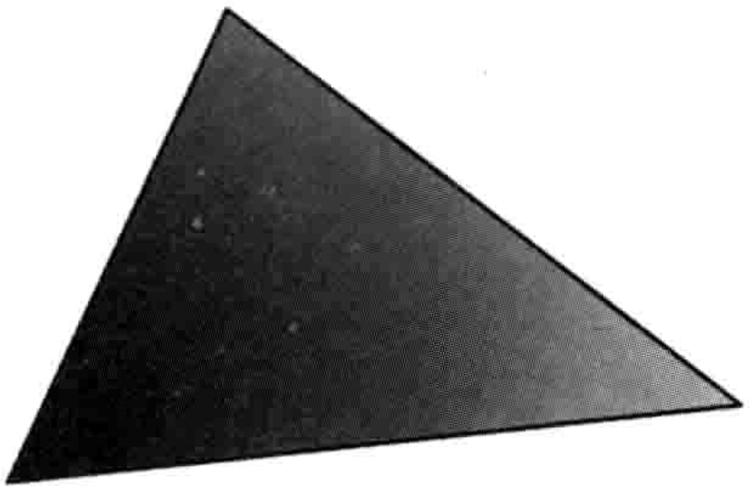


图 10.11: 以高氏着色法渲染三角形，每顶点有不同深浅的灰色。

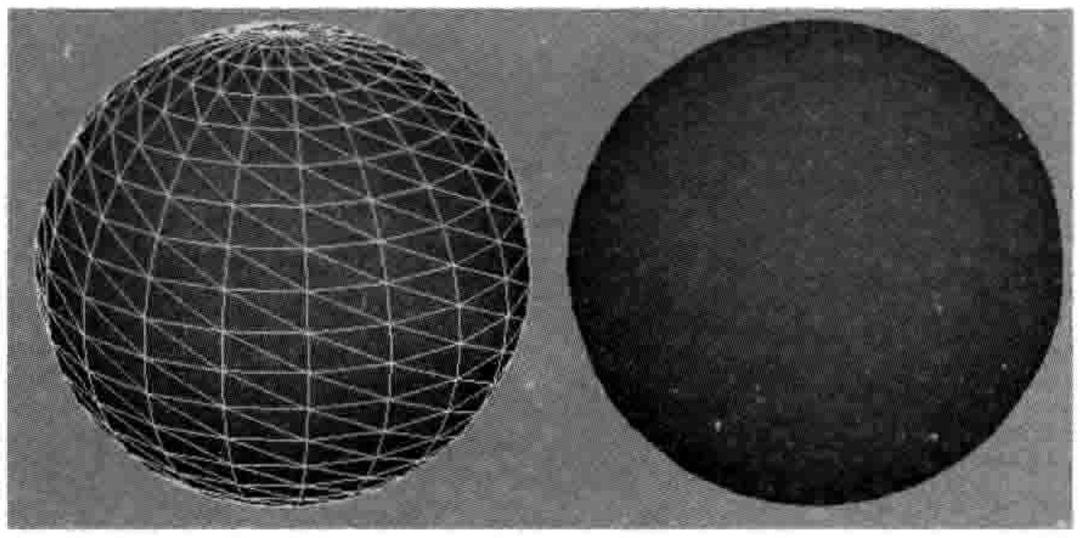


图 10.12: 高氏着色法能令物体的小平面显得圆滑。

顶点法线及圆滑化

10.1.3节将会提及，**光照**（lighting）是基于物体表面的视觉特性以及到达该表面的光线特性，来计算物体表面上各点的颜色的过程。光照网格的最简单方法就是，**逐顶点**（per-vertex）计算表面的颜色。换句话说，我们使用表面特性及入射光计算每个顶点的漫反射颜色（$\mathbf{d}_i$）。然后，这些顶点颜色会经由高氏着色法，在网格的三角形上插值。

为了计算表面某点的光线反射量，多数光照模型会利用在该点垂直于表面的**法矢量**。由于我们以逐顶点方式计算光照，所以此处可使用顶点法矢量$\mathbf{n}_i$。也因此，顶点法矢量的方向对于网格的最终外观有重要影响。

例如，假设有一个高瘦的长方体。若我们想使长方体的边缘显得锐利，那么可以使每个顶点法矢量与长方体的面垂直。计算每个三角形的光照时，3个顶点的法矢量是一模一样的，

[15] 译注：实际上，顶点属性需要透视校正插值（perspective-correct interpolation），而非简单地在屏幕空间进行线性插值。10.1.4.4节有相关介绍。

因此光照的结果显示为平面，并且在长方体顶点上的光照会如同顶点法矢量一样地做出突然转变。

我们也可以令相同的长方体网格显得更像一个圆滑的圆柱体，方法是把顶点法矢量改为自长方体的中线向外发散。此情况下，每个三角形上的顶点法矢量变得不同，引致其光照结果也不一样。利用高氏着色法为这些顶点颜色进行插值时，会使光照效果顺滑地在表面上过渡。图10.13比较了两种顶点法矢量设置所形成的光照效果。

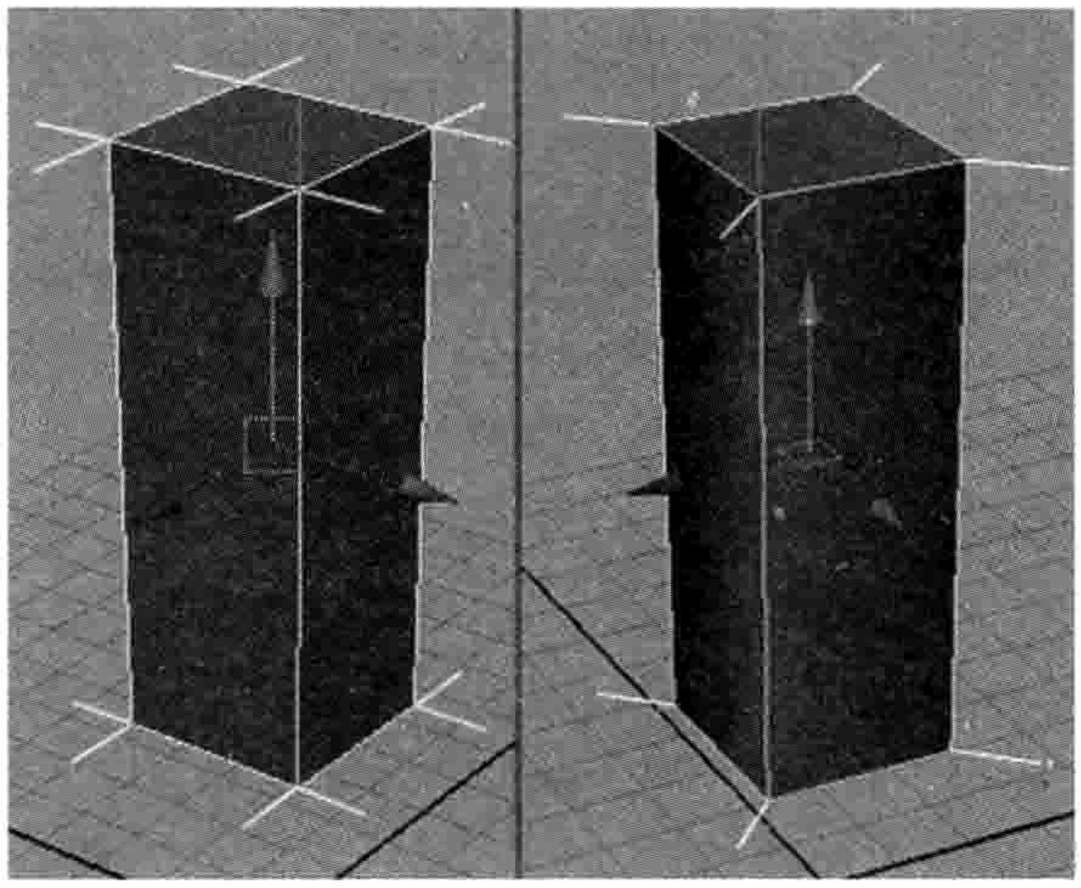


图 10.13: 网格的顶点法线方向，对于逐顶点光照计算出的颜色有重大影响。

10.1.2.5 纹理

若三角形比较大，以逐顶点方式设置表面性质可能会太过粗糙。线性的属性插值也非总是我们想要的，并且这种插值会引起一些视觉上的问题。

例如，渲染光滑物体的**镜面高光**（specular highlight）时，使用逐顶点光照会出现问题。通过把网格高度进行镶嵌，再使用逐顶点光照配合高氏着色法，可以做出相当不错的效果。然而，当三角形太大时，对镜面高光做线性插值所形成的误差便会非常明显，如图10.14所示。

要克服逐顶点表面属性的限制，渲染工程师通常会使用称为**纹理贴图**（texture map）的位图影像。纹理通常含有颜色信息，并且一般会投射在网格的三角形上。这种使用纹理的方法，就好像我们小时候把那些假纹身印在手臂上。但其实纹理也可以存储颜色以外的视觉特性。而且纹理也不一定用来投射于网格上，例如，可以把纹理当作存储数据的独立表格。纹理中的每个单独像素称为**纹素**（texel），用以区分于屏幕上的像素。

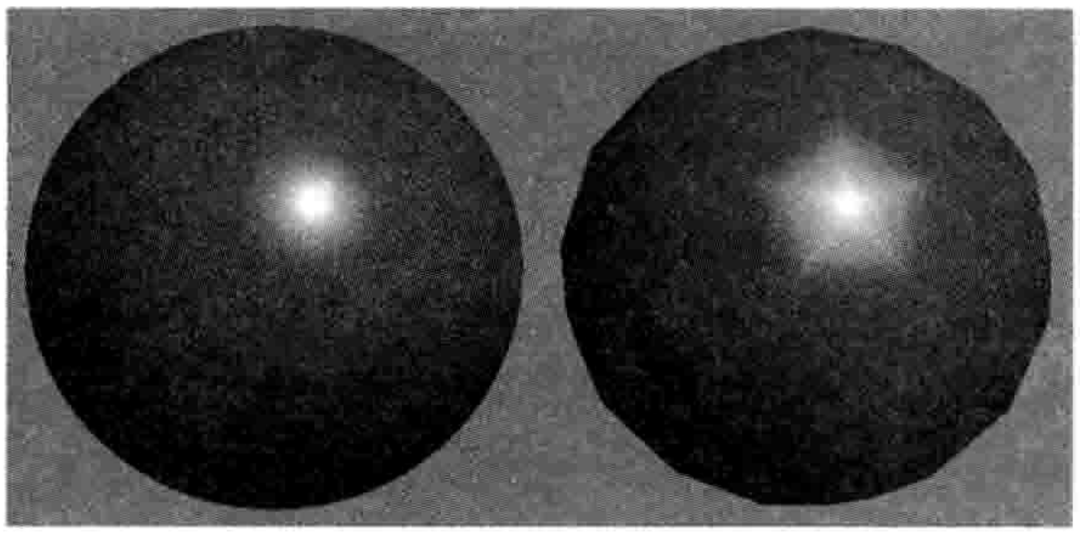


图 10.14: 对顶点属性做线性插值，有时候并不能准确描述表面的视觉特性，尤其是镶嵌不足的情况。

在某些图形硬件上，纹理位图的尺寸必须为2的幂。虽然纹理通常只要能塞进显存，多数硬件没有对其尺寸设限，但一般纹理的尺寸会是256×256、512×512、1024×1024及2048×2048等。有些图形硬件会加一些额外限制，例如要求纹理必须为正方形；有些硬件会解除一些限制，例如能接受2的幂以外的尺寸。

纹理种类

最常见的纹理种类为**漫反射贴图**（diffuse map），又称作**反照率贴图**（albedo map）。漫反射贴图的纹素储存了表面的漫反射颜色，这好比在表面上贴上贴纸或涂上漆油。

计算机图形学里也会使用其他种类的纹理，包括**法线贴图**（normal map）——每个纹素用来储存以RGB值编码后的法矢量、**光泽贴图**（gloss map）——在每个纹素上描述表面的光泽程度、**环境贴图**（environment map）——含周围环境的图像以渲染反射效果，此外还有各式各样的纹理种类。10.3.1节会讨论如何用几种纹理实现基于图像的光照（image-based lighting），以及其他效果。

事实上，纹理贴图可以储存任何在计算着色时所需的信息。例如，可以用一维的纹理储存复杂数学函数的采样值、颜色对颜色的映射表，或其他查找表（lookup table，LUT）。

纹理坐标

我们现在讨论如何将二维的纹理投射至网格上。首先，我们要定义一个称为纹理空间（texture space）的二维坐标系。纹理坐标通常以两个归一化的数值(u, v)表示。这些坐标的范围是从纹理的左下角$(0, 0)$伸展至右上角$(1, 1)$。[16]使用这样的归一化纹理坐标，好处是这

[16]译注：Direct3D的纹理空间是和上述所说的上下倒转，即$(0, 0)$为左上角，$(1, 1)$为右下角。两种惯例可以使用$(u', v') = (u, 1 - v)$互相转换。网格顶点中的纹理坐标可以在导出或转换网格时转换，或是在加载网格时转换，或是在着色器中转换。

些坐标不会受纹理尺寸影响。

要把三角形映射至二维纹理，只需要在每个顶点i上设置纹理坐标(u_i, v_i)。这样实际上就是把三角形映射至纹理空间的影像平面上。图10.15是一个纹理贴图的例子。

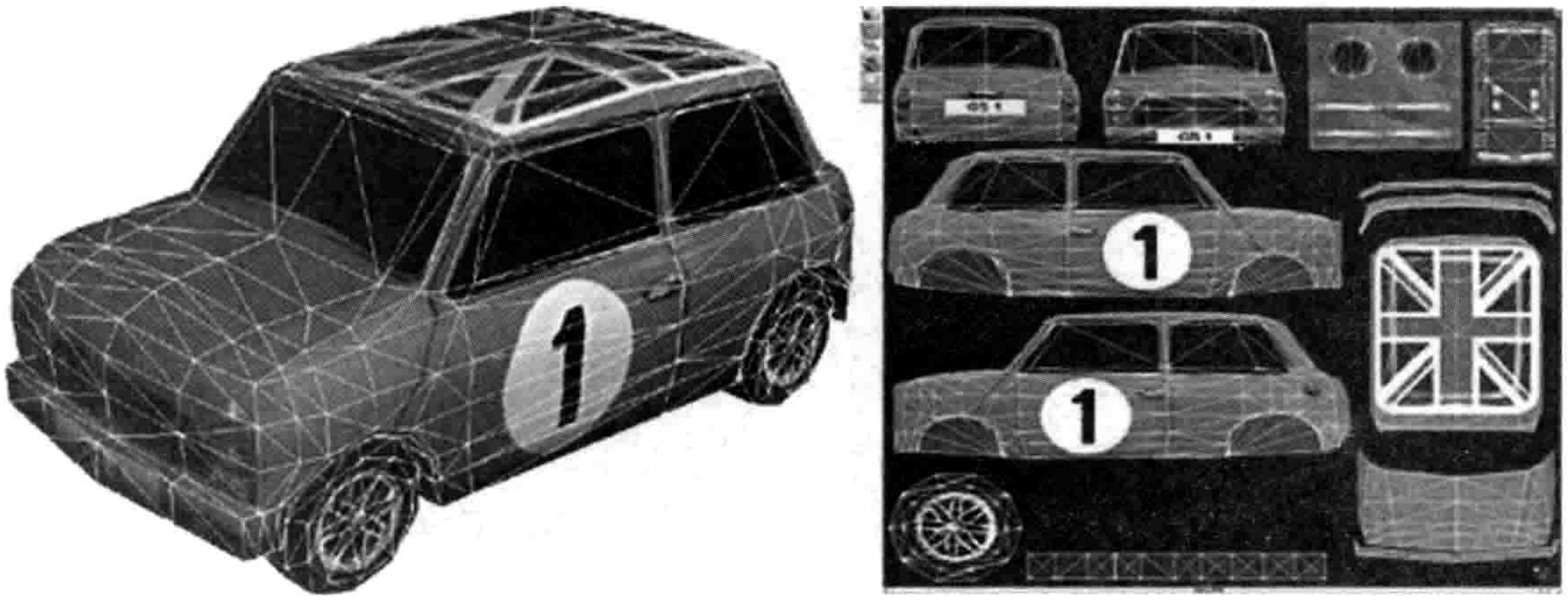


图 10.15: 纹理贴图的例子。同时展示网格三角形在三维空间和纹理空间之中。

纹理寻址模式

纹理坐标可以延伸至$[0, 1]$范围之外。图形硬件可用以下几种方式处理范围以外的纹理坐标。这些处理方式称为**纹理寻址模式**（texture addressing mode），可供用户选择。

- **缠绕模式**（wrap mode）[17]：此模式中，纹理在各方向重复又重复。所有形式为(ju, kv)的纹理坐标等价于(u, v)，当中j和k是任何整数。[18]
- **镜像模式**（mirror mode）：此模式和缠绕模式相似，不同之处在于，在u为奇数倍数上的纹理会在v轴方向形成镜像，在v为奇数倍数上的纹理会在u轴方向形成镜像。
- **截取模式**（clamp mode）：此模式中，当纹理坐标在正常范围之外，纹理的边缘纹素会简单地延伸。
- **边缘颜色模式**（border color mode）：此模式下用户能指定一个颜色，当纹理坐标在$[0, 1]$范围以外时使用。

图10.16展示了这些纹理寻址模式。

[17]译注：Direct3D和OpenGL的术语有许多分歧。Direct3D的纹理寻址模式（texture addressing mode）在OpenGL则称为纹理缠绕模式（texture wrap mode）；而Direct3D的寻址模式之中有一个缠绕模式（wrap mode），OpenGL相应的术语则是重复模式（repeat mode）。更复杂的是，Direct3D还有一个称为纹理缠绕（texture wrapping）的渲染状态D3DRS_WRAP0 7，又是另一个概念。

[18]译注：多数图形API都可以分别设置u和v的寻址模式，例如，u用缠绕模式，v用镜像模式。

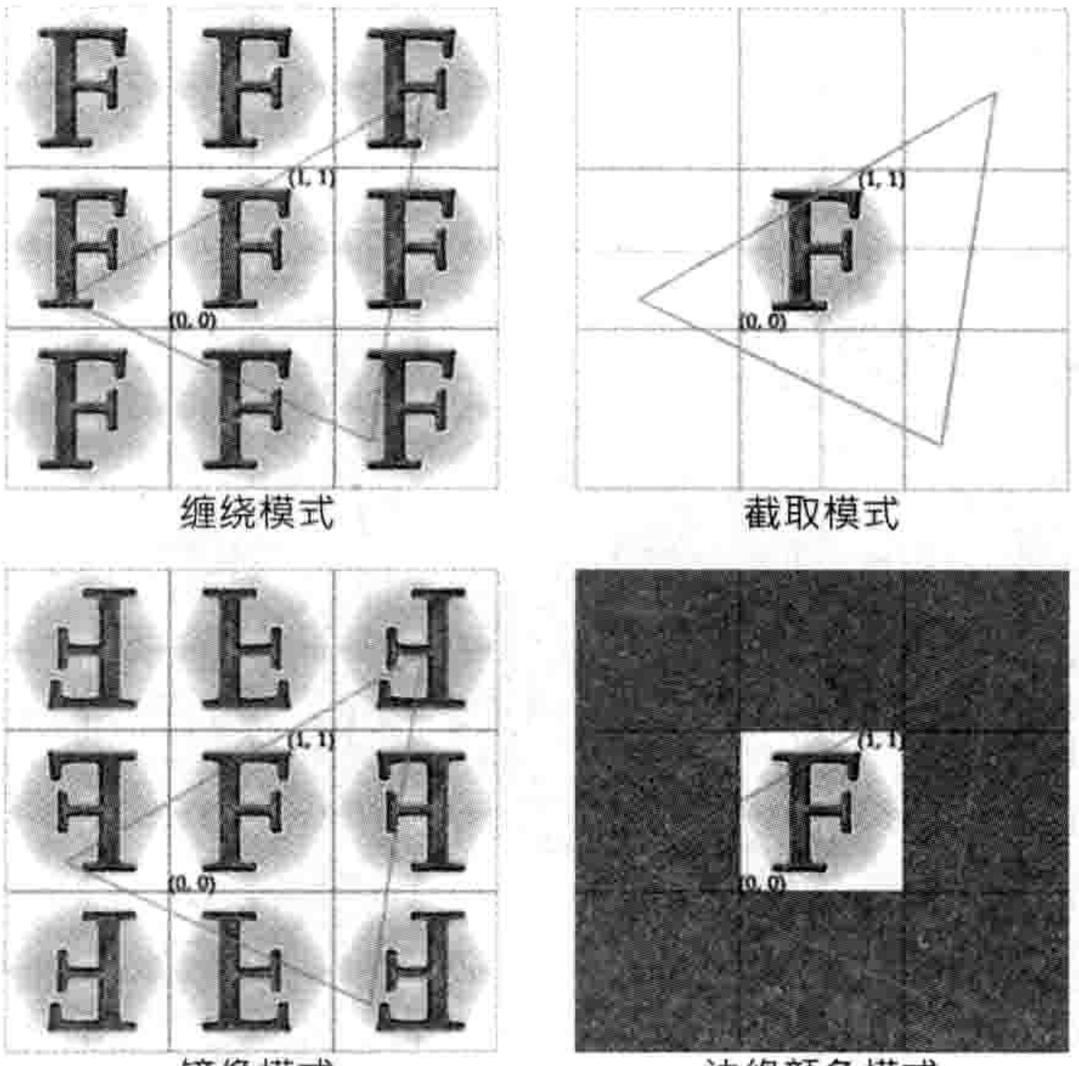


图 10.16: 纹理寻址模式。

纹理格式

纹理位图可在磁盘上储存为任何格式的文件，只要你的游戏引擎含有读取该文件至内存的代码便可。常见的文件格式有Targa（TGA）、便携式网络图形（Portable Network Graphics，PNG）、视窗位图（Windows bitmap，BMP）、标记图像文件格式（Tagged Image File Format，TIFF）。纹理存于内存时，通常会表示为二维像素数组，当中像素使用某种颜色格式，例如，RGB888、RGBA8888、RGB565、RGBA5551等。

多数现在的显卡及图形API都会支持**压缩纹理**（compressed texture）。DirectX支持一系列称为DXT或S3纹理压缩（S3 Texture Compression，S3TC）的压缩格式[19]。此处不详述其细节，但其基本原理是把纹理切割成多个4像素×4像素的小块[20]，并使用一个小型调色板（color palette）储存每个小块的颜色。关于S3TC格式可参考维基百科[21]。

显然，压缩纹理的优点是比无压缩纹理使用较少内存。其额外的好处，或许读者想不到。事实上，使用压缩纹理渲染也较高效。S3TC纹理能够提速，皆因其内存存取模式更缓存友好——每个小块把4 × 4个相邻像素储存至单个64或128位字，因此能够更充分利用缓

[19]译注：从Direct3D 10开始，DXT1/3/5分别改称为BC1/2/3，并加入BC4/5（BC为block compression的缩写）。Direct3D 11又增加了BC6H和BC7纹理压缩格式。

[20]译注：原文2 × 2应为笔误。

[21]http://en.wikipedia.org/wiki/S3_Texture_Compression

存[22]。然而，压缩纹理会导致一些失真，这些失真有时候不易察觉，但当失真太严重时，就必须以无压缩纹理取代。

纹素密度及多级渐远纹理

想象我们要渲染一个满屏的四边形（两个三角形组成的长方形），此四边形还贴上一张纹理，其尺寸刚好配合屏幕的分辨率。在这种情况下，每个纹素刚好对应一个屏幕像素，我们称其**纹素密度**（texel density，即纹素和像素之比）为1。当在较远距离观看该四边形时，其屏幕上的面积就会变小。由于纹理的尺寸不变，该四边形的纹素密度就会大于1，即每个像素会受多于一个纹素所影响。

显然纹素密度并不是一个常量，它会随物体相对摄像机的距离而改变。纹素密度影响内存使用量，也影响三维场景的视觉品质。当纹素密度远低于1，每个纹素就会显著比屏幕像素大，那么就会开始察觉到纹素的边缘。这会毁灭游戏的真实感。当纹素密度远高于1，许多纹素会影响单个屏幕像素。这样会产生如图10.17所示的**莫列波纹**（moiré banding pattern）。更甚者，由于像素边缘内的多个纹素会按细微的摄像机移动而不断改变像素的颜色，像素的颜色就会显得浮动不定及闪烁。而且，若玩家永不会接近一些远距离的物体，用非常高的纹素密度渲染那些物体只是浪费内存。毕竟若无人能看见其细节，在内存保留高分辨率纹理又有何用？

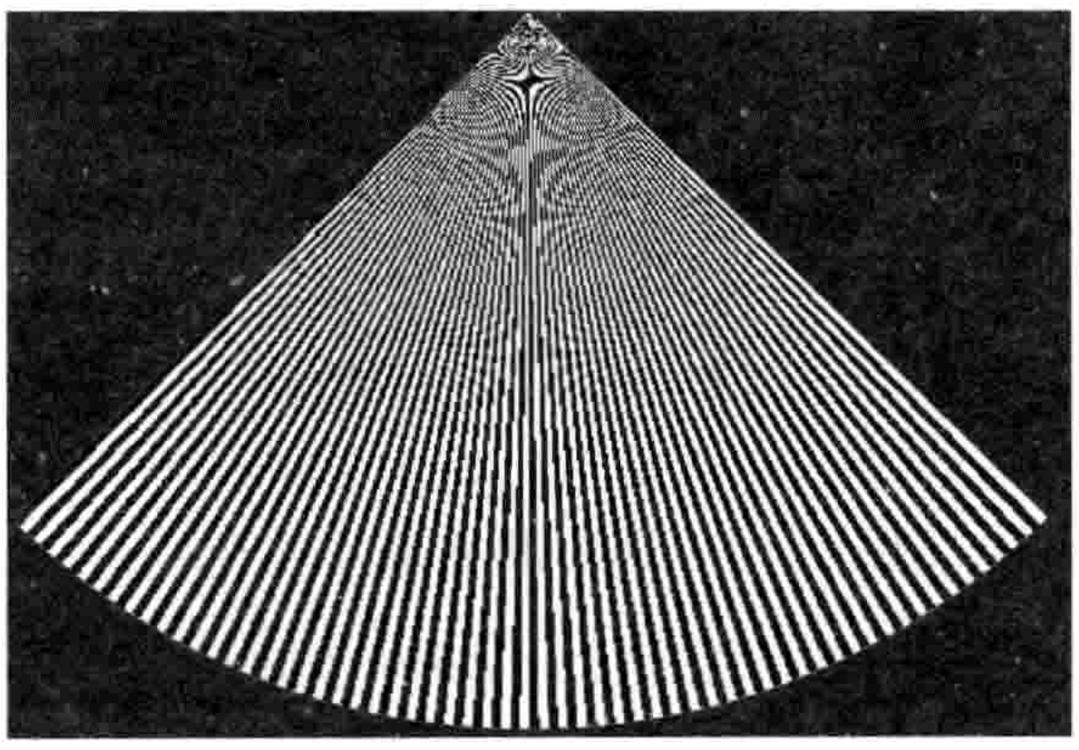


图 10.17: 纹理密度大于1可引致莫列波纹。

理想地，我们希望无论物体是近是远，仍然维持纹素密度接近于1。要准确地维持此约束是不可能的，但可以使用**多级渐远纹理**（mipmapping）技术来逼近。其方法是，对于每张纹理，我们建立较低分辨率位图的序列，当中每张位图的宽度和高度都是前一张

[22]译注：如果是一般的线性储存方式，从某像素往上或往下存取时，其内存位置的差距较大，缓存效率较低。除了以小块方式储存，有些图形硬件还能提供（或内部自动转换成）一种swizzle纹理格式，取代线性的扫描方式，以增加缓存效率。

位图的一半。我们称这些影像为**多级渐远纹理**（mipmap）或**渐远纹理级数**（mip level）。如图10.18所示的例子，一张64 × 64的纹理会有以下的渐远纹理级数：64 × 64、32 × 32、16 × 16、8 × 8、4 × 4、2 × 2和1 × 1。当使用多级渐远纹理时，图形硬件便会按照三角形[23]与摄像机的距离，选择合适的渐远纹理级数，以尝试维持纹素密度接近于1。例如，若纹理占据屏幕40 × 40像素的面积，那么就可能会选择64 × 64的渐远纹理级数；若同样的纹理只占10 × 10像素，就可能会选择16 × 16的渐远纹理级数。我们以下将会提及，硬件可用**三线性过滤**（trilinear filtering）对两张相邻级数的渐远纹理采样，再混合采样结果。在上述例子中，在10 × 10像素面积贴图时，可以分别对16 × 16及8 × 8渐远纹理采样并混合结果。

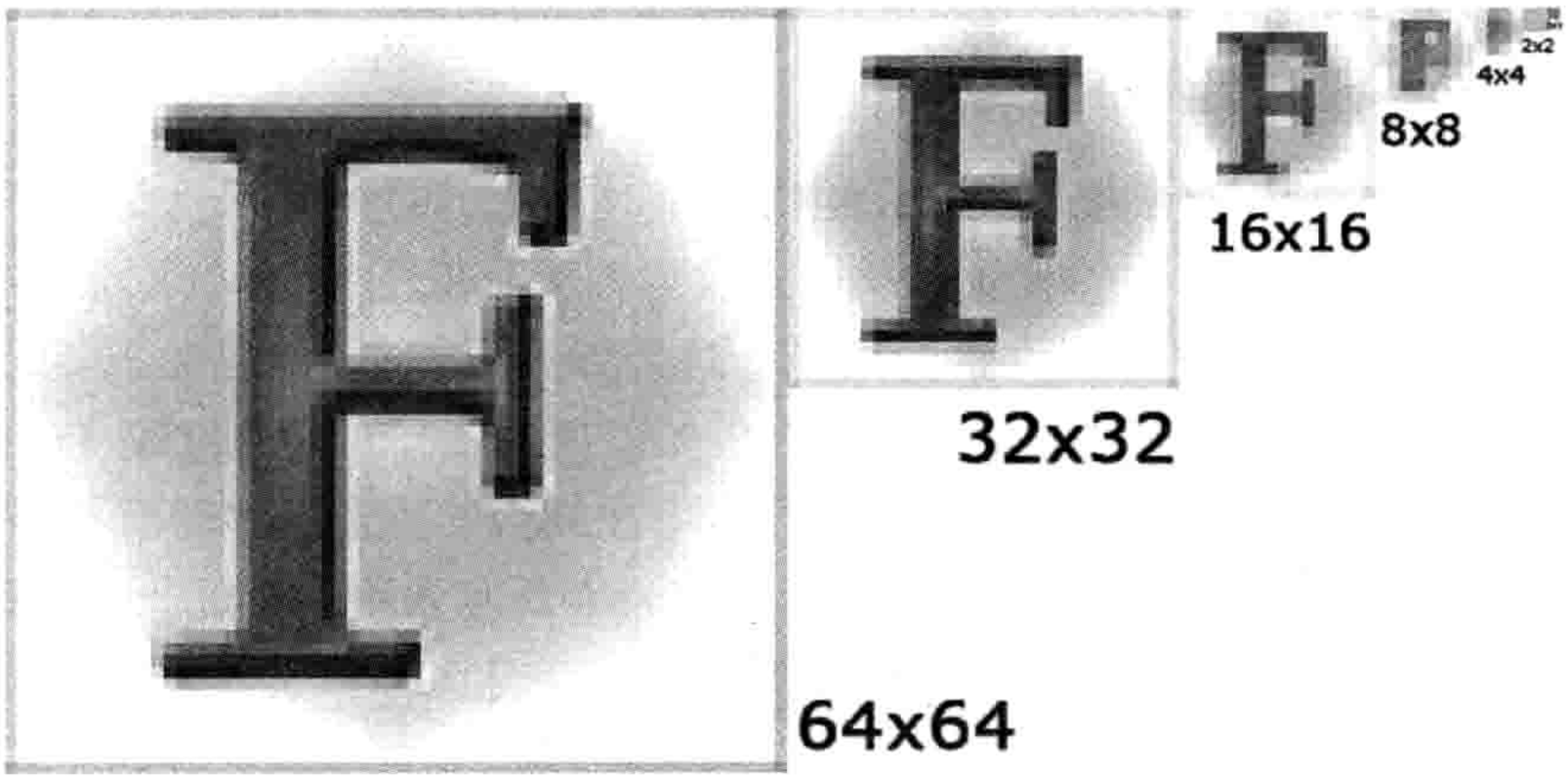


图 10.18: 64 × 64纹理的渐远纹理级数。

世界空间纹素密度

纹素密度一词，也可用于描述纹素和贴图表面的世界空间面积之比。例如，2米宽的正方形贴上256 × 256纹理，其纹素密度就是$256^2/2^2 = 16384$。[24]为了和之前所谈的屏幕空间纹素密度区别，笔者把此密度称为**世界空间纹素密度**（world space texel desnity）。

世界空间纹素密度不需要接近1，实际上此数值许多时候会比1大很多，具体要视乎所选择的世界单位（world unit）。然而重要的是，在物体贴上纹理时，应使用大概一致的世界空间纹理密度。例如，我们期望一个立方体的六个面应该占用相同的纹理面积。若然不是，立方体的其中一面的纹理解像度可能较另一面低，或会让玩家察觉出来。许多游戏工作室会为其美术团队制定指引，并会提供引擎内的纹素密度可视化工具，使游戏内所有物体都有相对一致的世界空间纹素密度。

[23]译注：更准确地说，图形硬件会逐像素（或逐采样）计算渐远纹理级数。

[24]译注：此数字的单位是纹素每平方米（texel m^{-2}）。或许以下的方法能更直观理解，假设内存有其成本，若每个纹素需要成本1元，那么上述例子每平方米就要16,384元……

纹理过滤

当渲染纹理三角形上的像素时，图形硬件会计算像素中心落入纹理空间的位置，来对纹理贴图采样。通常纹素和像素之间并没有一对一的映射，像素中心可以落入纹理空间的任何位置，包括在两个或以上纹素之间的边缘。因此，图形硬件通常需要采样出多于一个纹素，并把采样结果混合以得出实际的采样纹素颜色。此过程称为**纹理过滤**（texture filtering）。

多数显卡支持以下的纹理过滤种类。

- **最近邻**（nearest neighbor）：这种粗糙方法会挑选最接近像素中心的纹素。当使用多级渐远纹理时，此方法会挑选一个渐远纹理级数，该级数最接近但高于理想的分辨率。理想分辨率是指达到屏幕空间纹素密度为1。
- **双线性**（bilinear）：此方法会对围绕像素中心的4个纹素采样，并计算该4个颜色的加权平均（权重是基于纹素和像素中心的距离）。当使用多级渐远纹理时，也是选择最接近的级数。
- **三线性**（trilinear）：此方法把双线性过滤法施于最接近的两个渐远纹理级数（一个高于理想分辨率，一个低于理想分辨率），然后把两个采样结果线性插值。这样便能消除屏幕上碍眼的、相邻渐远纹理级数之间的边界。
- **各向异性**（anisotropic）：双线性和三线性过滤都是对2×2的纹素块采样。如果纹理表面是刚好面对着摄像机的，这样是正确的做法。然而，若表面倾斜于虚拟屏幕平面，这就不太正确了。各向异性过滤法会根据视角，对一个梯形范围内的纹理采样，借以提高非正对屏幕的纹理表面的视觉品质。

10.1.2.6 材质

材质（material）是网格视觉特性的完整描述。这包括贴到网格表面的纹理设置，也包含一些高级特性，例如选用哪一个着色器、该着色器的输入参数，以及控制图形加速硬件本身的功能参数。

虽然顶点属性从技术上来说也是表面特性描述的一部分，但是顶点属性并不算是材质的一部分。然而，顶点属性随网格而来，因此网格和材质结合后包含所有需要渲染物体的信息。“网格—材质对”有时称为“**渲染包**（render packet）”，而“几何图元（geometric primitive）”一词有时候也会延伸至包含“网格—材质对”的意思。

三维模型通常会使用多于一个材质。例如，一个人类模型可能有多个材质，供头发、皮肤、眼睛、牙齿、多种服饰等之用。因此，一个网格通常会切割成**子网格**（submesh），每个子网格对应一个材质。OGRE渲染引擎在其`Ogre::SubMesh`类实现了此设计。

10.1.3 光照基础

光照是所有计算机图形渲染的中心。欠缺良好的光照，原本精致建模的场景会显得平面及不自然。同样地，就算是极简单的场景，当准确地照明时也会显得极为真实。如图10.19所示的经典“康乃尔盒子（Cornell box）[25]”场景，就是这种情况的好例子。

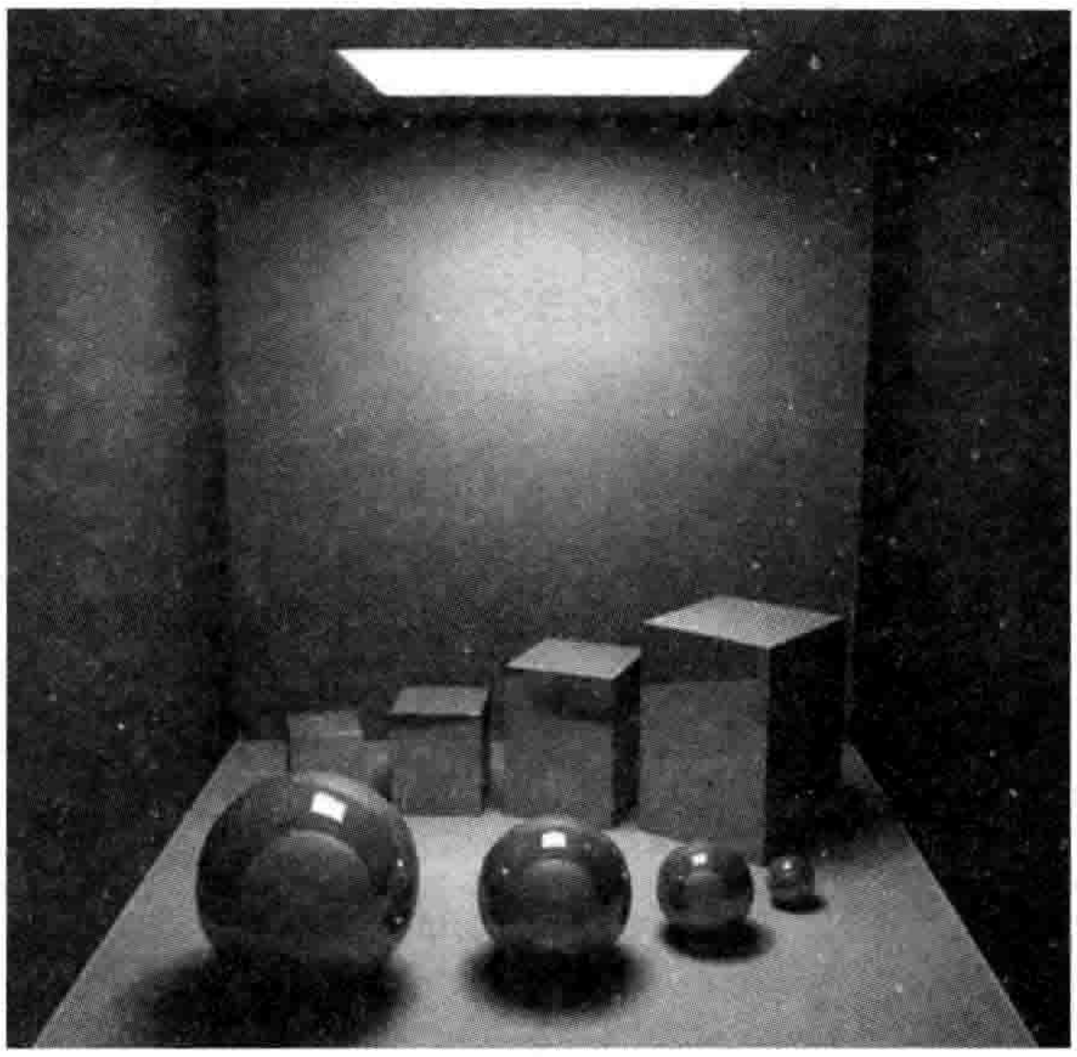


图 10.19: 这是经典“康乃尔盒子”场景的变种，它展示了真实光照可以令最简单的场景都变得真实的照片。

以下是一组顽皮狗《神秘海域：德雷克船长的宝藏》的截图，它们凸显了光照的重要性。图10.20中，场景以无贴图方式渲染。图10.21则是同一场景，但仅渲染漫反射贴图。图10.22是综合贴图和完整光照的效果。留意应用光照后，能令场景的真实性顿时提高。

着色（shading）[26]一词，通常是光照加上其他视觉效果的泛称。因此，着色还包含了以过程式的顶点变形表现水面动态、生成毛发曲线或皮毛外壳（fur shell）[27]、高次曲面（high order surface）的镶嵌，以及许多渲染场景所需的计算。

以下几个小节会介绍光照的基础，以便理解图形硬件及渲染管道。在10.3节会再介绍一些高级光照及着色技术。

---

[25]译注：康乃尔盒子是康乃尔大学的几位学者在1984年论文中所设计的场景，其原来功能是研究漫反射的全局光照模型。可于http://www.graphics.cornell.edu/online/box/阅读有关历史及实验数据。

[26]译注：也可译作“浓淡处理”。此词应源于绘画艺术中对光影和材质的表现手法。

[27]译注：这是渲染皮毛（如兽皮或毛绒玩具上的短毛）的技术之一，基本方法是把皮肤的网格往外拉伸（extrude），形成多层的外壳，然后每层外壳以同一张毛发横切面贴图（就是透明背景上有许多点状）渲染。详情可参考http://research.microsoft.com/en-us/um/people/hoppe/fur.pdf。

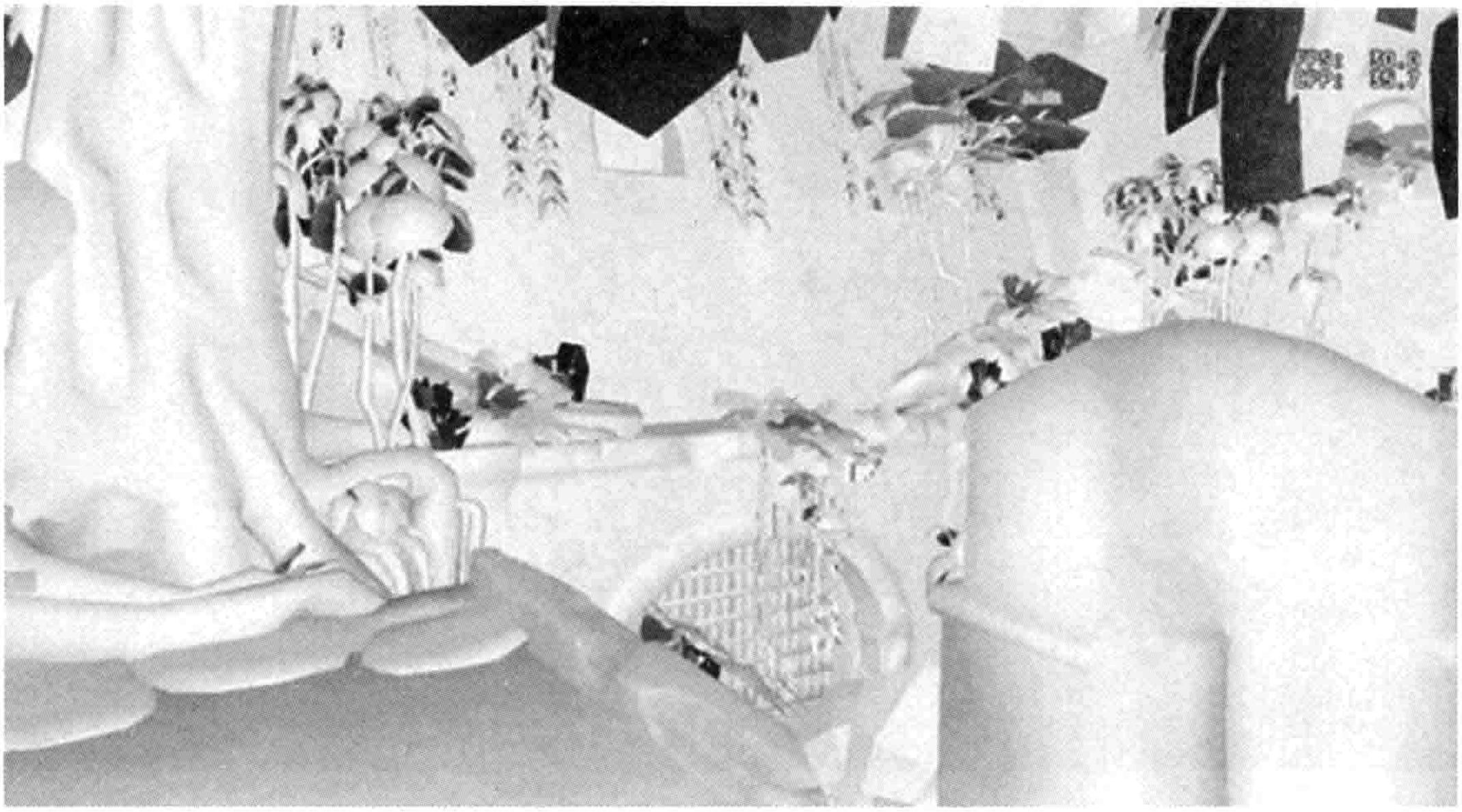


图 10.20: 以无纹理的方式渲染《神秘海域：德雷克船长的宝藏》的一个场景。

图 10.21: 以只含纹理的方式渲染同一个《神秘海域》场景。

图 10.22: 以完整光照方式渲染同一个《神秘海域》场景。

10.1.3.1 局部及全局光照模型

渲染引擎使用多种数学模型来模拟光和表面/体积的交互作用，这些模型称为**光传输模型**（light transport model）。最简单的模型只考虑**直接光照**（direct lighting）。此模型中，光发射后，碰到场景中某个物体后会反射，然后直接进入虚拟摄像机的虚拟平面。这种简单模型又称为**局部光照模型**（local illumination model），因为其仅考虑光对于单个物体的局部影响，换句话说，此模型中每个物体不会影响其他物体的光照。局部光照模型是游戏历史中最早应用到的模型，不足为奇；事实上这个模型还在现时的游戏中使用，在某些情况下可以产生极为真实的效果。

而要达致真正的照相写实，就必须考虑到**间接光照**（indirect lighting），即光被多个表面反射后才进入摄像机。照顾到间接光照的模型称为**全局光照模型**（global illumination model）。有些全局光照模型针对某种视觉现象，例如产生逼真的阴影、模拟反射性表面、考虑物体间的互相反射（某物体的颜色会影响其邻近物体的颜色）、模拟焦散（caustics）效果（如水面或光滑金属表面的强烈反射）。其他全局光照模型尝试模拟多种光学现象，例如光线追踪（ray tracing）和辐射度算法（radiosity）。

全局光照模型能够完全[28]由单一数学公式描述，此公式称为**渲染方程**（the rendering equation）[29]或**着色方程**（shading equation）。此公式由Jim Kajiya在1986一篇开创性的SIGGRAPH论文中提出。从某意义上，所有渲染技术都可视为此渲染方程的完全或部分解，尽管每个技术的基本方法、假设、简化方式、逼近方式有所不同。关于渲染方程的详情，可参阅维基百科[30]或任何高级渲染和光照的文章。

10.1.3.2 Phong氏光照模型

在游戏渲染引擎中，最常用的局部光照模型就是**Phong氏反射模型**（Phong reflection model）[31]。此模型把从表面反射的光分解为3个独立项。

- **环境**（ambient）项模拟场景中的整体光照水平。此乃场景中间接反射光的粗略估计。间接反射的光使阴影部分不会变成全黑。[32]
- **漫反射**（diffuse）项模拟直接光源在表面均匀地向各个方向反射。此项能逼近真光源照射至哑光表面（matte surface）的反射，例如木块或布料。
- **镜面反射**（specular）项模拟在光滑表面会看到的光亮高光。镜面高光会出现在光源的直接反射方向。

图10.23显示了环境项、漫反射项和镜面反射项结合后，形成最终的表面强度和颜色。

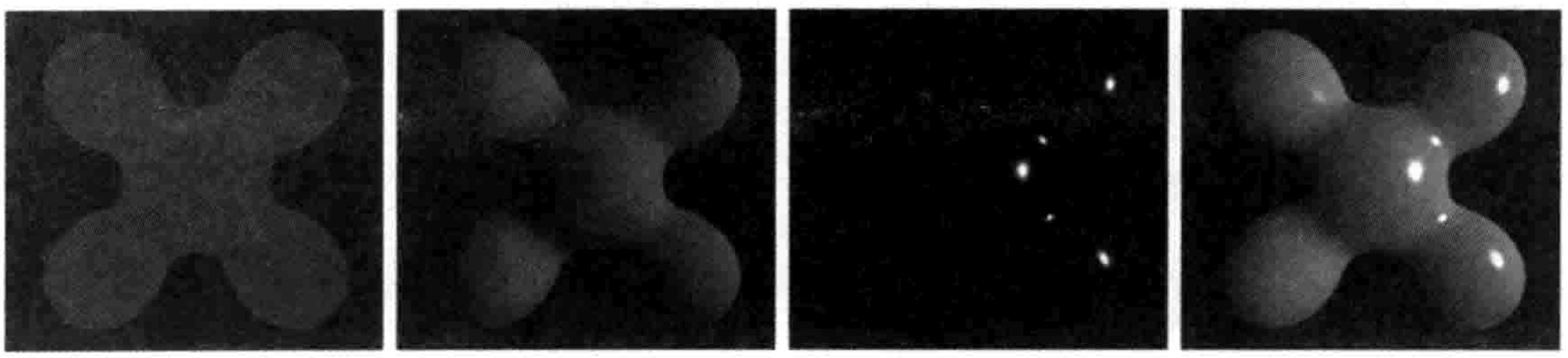


图 10.23: 把环境、漫反射及镜面反射项相加计算Phong反射。

[28]译注：其实并不“完全”，还有几项光学现象未能描述，包括磷光（phosphorescence）、萤光（fluorescence）、干涉（interference）和次表面散射（subsurface scattering）。但此公式在一般图形学中已经非常全面了。

[29]译注：此方程特有the这个冠词，是代表它是唯一的，而不是多种方程之一，有唯我独专的气焰，译者无法译出此层意思。

[30]http://en.wikipedia.org/wiki/Rendering_equation

[31]译注：Bui Tuong Phong（1942—1975）是越南裔美籍计算机图形学先驱。维基百科上的中文译名是裴祥风，但在图形学里Phong reflection model却通常译作冯氏反射模型。为避免混淆，本译作采用其英文姓氏。另一要注意的是，Phong反射模型和Phong着色法（Phong shading）是完全两回事。

[32]译注：环境光还会用来模拟散射形成的光源，例如，蓝天会使阴影变成蓝色。

计算表面上某点的Phong反射，需要几个输入参数。Phong氏模型一般会独立地施于3个颜色通道(R,G,B)，因此以下谈及的颜色参数都是三维矢量。Phong氏模型的输入包括：

- 视线方向矢量$\mathbf{V} = [V_x \quad V_y \quad V_z]$是从反射点延伸至虚拟摄像机焦点的方向。
- 3个颜色通道的环境光强度$\mathbf{A} = [A_R \quad A_G \quad A_B]$。
- 光线到达表面上那一点的表面法线$\mathbf{N} = [N_x \quad N_y \quad N_z]$。
- 表面的反射属性，包括：
  - 环境反射量k_A。
  - 漫反射量k_D。
  - 镜面反射量k_S。
  - 镜面的“光滑度（glossiness）”幂α。
- 每个光源i的属性，包括：
  - 光源的颜色及强度$\mathbf{C}_i = [C_{Ri} \quad C_{Gi} \quad C_{Bi}]$。
  - 从反射点至光源的方向矢量$\mathbf{L}_i$。

在Phong氏模型中，从表面上某点反射的光强度$\mathbf{I}$可以表示为以下的矢量方程：

$$\mathbf{I} = k_A\mathbf{A} + \sum_i \left[k_D(\mathbf{N}\cdot\mathbf{L}_i) + k_S(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right]\mathbf{C}_i$$

当中，求和部分计算所有能影响到该点的光源所产生的反射。此方程能分解为3个标量方程，每个方程对应一个颜色通道：

$$I_R = k_A A_R + \sum_i \left[k_D(\mathbf{N}\cdot\mathbf{L}_i) + k_S(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right]C_{Ri}$$

$$I_G = k_A A_G + \sum_i \left[k_D(\mathbf{N}\cdot\mathbf{L}_i) + k_S(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right]C_{Gi}$$

$$I_B = k_A A_B + \sum_i \left[k_D(\mathbf{N}\cdot\mathbf{L}_i) + k_S(\mathbf{R}_i\cdot\mathbf{V})^\alpha\right]C_{Bi}$$

在这些方程中，矢量$\mathbf{R}_i = [R_{xi} \quad R_{yi} \quad R_{zi}]$是光线方向$\mathbf{L}_i$对于表面法线的反射。

矢量$\mathbf{R}_i$可以用一些矢量运算求得。任何矢量都可以表示为切线分量和法线分量之和。例如，光线方向矢量$\mathbf{L}$可分解为：

$$\mathbf{L} = \mathbf{L}_T + \mathbf{L}_N$$

由于点积$\mathbf{N}\cdot\mathbf{L}$表示$\mathbf{L}$在法线方向的投影（此值为标量），因此法线分量$\mathbf{L}_N$就是单位法线以此

点积缩放的结果：

$$\mathbf{L}_N = (\mathbf{N} \cdot \mathbf{L})\mathbf{N}$$

而反射矢量$\mathbf{R}$和$\mathbf{L}$有同一个法线分量，但有相反的切线分量（$-\mathbf{L}_T$）。因此，可以这样求$\mathbf{R}$：

$$\begin{aligned}\mathbf{R} &= \mathbf{L}_N - \mathbf{L}_T \\ &= \mathbf{L}_N - (\mathbf{L} - \mathbf{L}_N) \\ &= 2\mathbf{L}_N - \mathbf{L} \\ &= 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L}\end{aligned}$$

此方程能计算出所有光源方向$\mathbf{L}_i$的对应反射方向$\mathbf{R}_i$。

Blinn-Phong

Blinn-Phong反射模型[33]是Phong反射模型的变种，两者在计算镜面反射时有些微差别。我们定义中间方向矢量（halfway vector）$\mathbf{H}$为视线方向矢量$\mathbf{V}$和光线方向矢量$\mathbf{H}$的中间。Blinn-Phong模型的镜面分量为$(\mathbf{N} \cdot \mathbf{H})^\alpha$，异于Phong模型的$(\mathbf{R} \cdot \mathbf{V})^\alpha$。而当中的幂$\alpha$也和Phong的$\alpha$有些差别，但可以选择一些合适的值使结果接近Phong的镜面反射项。

Blinn-Phong模型以降低准确度来换取更高的性能，然而Blinn-Phong模型实际上模拟某些材质时，比Phong模型更接近实验测量的数据。Blinn-Phong模型几乎是早期计算机游戏的唯一之选，并且以硬件形式进驻早期GPU的固定管线。此模型的细节可参考维基百科[34]。

BRDF图表

Phong反射模型中，其3个项是通用的**双向反射分布函数**（bidirectional reflection distribution function，BRDF）之特例。BRDF是沿视线方向$\mathbf{V}$的向外（反射）辐射与沿入射光线$\mathbf{L}$的进入辐射之比。

BRDF可以显示为一个半球图表，当中距原点的径向距离代表从该角度观察到的反射光强度。Phong的**漫反射**项为$k_D(\mathbf{N} \cdot \mathbf{L})$。此反射项只顾及入射方向$\mathbf{L}$，和视线角度$\mathbf{V}$无关。因此，此反射项的值在不同视线角度都是一样的。若把此项以三维视线角度的函数作图，那么结果就像一个半球体，球心位于计算Phong反射的位置。图10.24用二维方式显示此函数。

---

[33]译注：Jim Blinn是计算机图形学的先驱，为人熟知的研究成果还有环境贴图（environment mapping）、凹凸贴图（bump mapping）等。

[34]http://en.wikipedia.org/wiki/Blinn-Phong_shading_model

而Phong模型的镜面反射项是$k_S(\mathbf{R}\cdot\mathbf{V})^\alpha$。此项同时取决于光源方向$\mathbf{L}$及视线方向$\mathbf{V}$。当视角接近$\mathbf{L}$对表面法线的反射方向$\mathbf{R}$时，此函数产生镜面“热点”。然而，当视角离开光源反射的方向，其作用瞬即骤减。图10.25显示了此函数的二维情况。

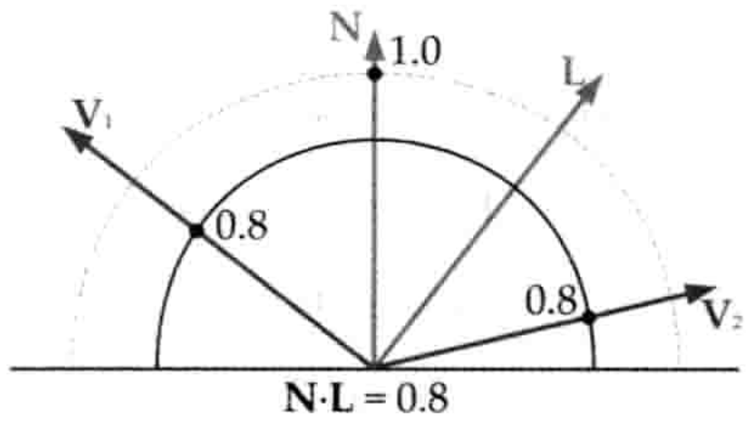


图 10.24: Phong反射模型的漫反射项取决于$\mathbf{N}\cdot\mathbf{L}$，但和视线方向$\mathbf{V}$无关。

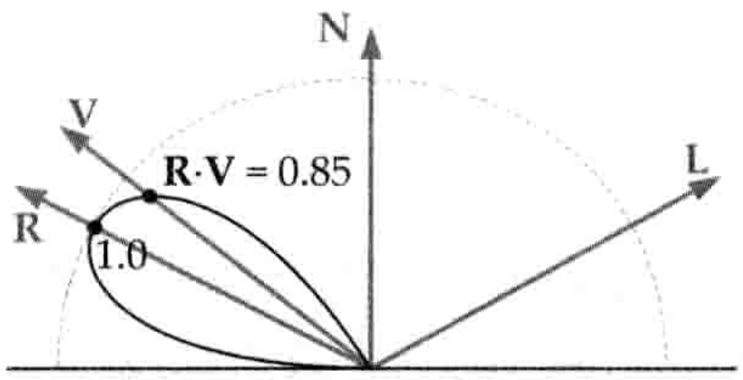


图 10.25: 在视线方向$\mathbf{V}$与反射方向$\mathbf{R}$重叠时，Phong反射模型的镜面反射项到达最大值，然后随$\mathbf{V}$偏离$\mathbf{R}$急速下降。

10.1.3.3 光源模型

除了为光线和平面之间的交互作用建模，我们还需描述场景中的光源。如同所有实时渲染的做法，我们会使用多个简化的模型逼近现实中的光源。

静态光照

最快的光照计算就是不计算。因此光照最好能尽量在游戏运行前计算。我们可以在网格的顶点预计算Phong反射，并把结果储存于顶点漫反射颜色属性中。我们也可以逐像素预计算光照，把结果储存于一类名为**光照贴图**（light map）的纹理贴图上。在运行时，把光照贴图纹理投影在场景中的物体，以显示光源对物体的影响。

读者或许会问，为何不把光照信息直接烘焙到场景中的漫反射纹理中？原因有几个。首先，漫反射纹理贴图通常会在场景中密铺或重复使用，所以把光照烘焙在它们上是不可行的。取而代之，我们会使用一张光照纹理贴到所有受光源影响范围内的物体上。这样做能令动态物体经过光源时得到正确的光照。而光照贴图的分辨率也可以异于（通常是低于）

漫反射纹理的分辨率。最后一点，“纯”光照贴图通常比包含漫反射颜色信息的贴图更易压缩[35]。

环境光

环境光（ambient light）对应Phong光照模型中的环境项。此项独立于视角，并且不含方向。因此，环境光由单个颜色表示，该颜色对应Phong方程中的**A**颜色项（在运行时会以表面的环境反射项k_A来缩放）。在游戏世界的不同区域中，环境光的强度和颜色可以改变。

平行光

平行光（directional light）模拟距离受光表面接近无限远的光源，例如太阳。自平行光发射的光线是平行的，而光源本身在游戏世界中并无特定位置。因此，平行光以光源的颜色**C**和方向**L**表示。图10.26显示了一个平行光。

点光/全向光

点光（point light）又称为**全向光**（omni-directional light），在游戏世界中有特定位置，并向所有方向均匀辐射。光的强度通常设定为以光源距离做平方衰减，超出预设的最大有效半径就会把强度设为0。点光由其位置**P**、光源颜色/强度**C**及最大半径$r_{\max}$表示。渲染引擎只需要把点光的效果施于其球体范围的表面（此仍重要的优化）。图10.27显示了一个点光。

聚光

聚光（spot light）的行为等同于发射光线受限于一个圆锥范围的点光，如手电筒一样。通常会用内角和外角设置两个圆锥。在内圆锥里，光线以最高强度发射。而在内角和外角之间，光线的强度会衰减，直至外圆锥强度归零。在两个圆锥里，光的强度也会按径向距离衰减。聚光以位置**P**、光源颜色**C**、中央方向矢量**L**、最大半径$r_{\max}$、内外圆锥角$\theta_{\min}$和$\theta_{\max}$表示。图10.28显示了一个聚光。

[35]译注：因为光照贴图只含低频的漫反射信息。

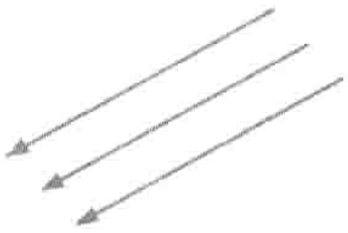


图 10.26: 平行光源的模型。

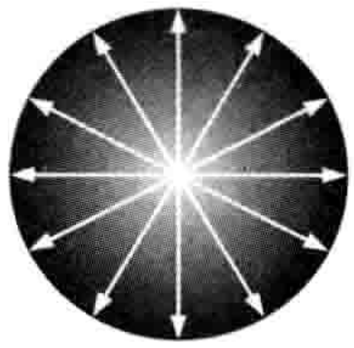


图 10.27: 点光源的模型。

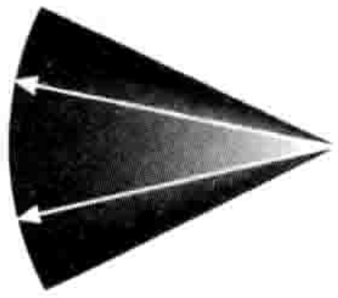


图 10.28: 聚光源的模型。

面积光

以上所述之光源都是从一个理想化的点，自无限远或本地进行辐射。但现实的光源几乎必定有非零的面积，因此才会使其产生的阴影含有本影（umbra）和半影（penumbra）。

与其显式地为面积光（area light）建模，计算机图形工程师通常会用多种“小技巧”模拟其行为。例如，要模拟半影，可以投下多个阴影，再把结果混合；又或是以某种方式把锐利的阴影边缘模糊化。

发光物体

场景中有些表面本身也是光源。例如手电筒、发光的水晶球、火箭喷出的火焰等。发光表面可用**放射光贴图**（emissive texture map）来模拟，此纹理的颜色永远以完全强度发射，不受附近的光照环境所影响。这种纹理可以用来定义霓虹灯标志、车头灯等。

有些发光物体（emissive object）会结合多种技术来渲染。例如，渲染手电筒时，在直望的方向可以使用放射光贴图，在同一位置加入聚光以照亮场景，加入半透明的黄色网格模拟光锥，渲染面向摄像机的卡片以模拟镜头光晕（lens flare）（若引擎支持高动态范围光照可用**敷霜效果**/bloom代替[36]），以及用投射纹理把手电筒的焦散效果投射至受光的表面上。《路易士鬼屋（Luigi's Mansion）》中的手电筒是这类效果组合的绝佳例子，见图10.29。

10.1.4 虚拟摄像机

在计算机图形学中，虚拟摄像机比现实的摄像机或人类眼睛简单得多。我们把摄像机当作一个理想的焦点，并有一个矩形虚拟感光表面——称为**成像矩形**（imaging rectangle）——悬浮在焦点的近距离前面。成像矩形由正方或矩形虚拟感光元件的栅格所组成，每个感

[36]译注：原文虽然说是代替，其实两种效果并不接近。镜头光晕的成因是由强光在镜头内反射所形成的，因此会出现在屏幕中不同位置（通常由一串穿过屏幕中心的光晕组成）。而敷霜效果的成因包括不完美的对焦，或是由于强光在相邻感光元件溢出（在视网膜和胶卷也常会出现），所以此效果主要会在屏幕中光亮的部分出现。

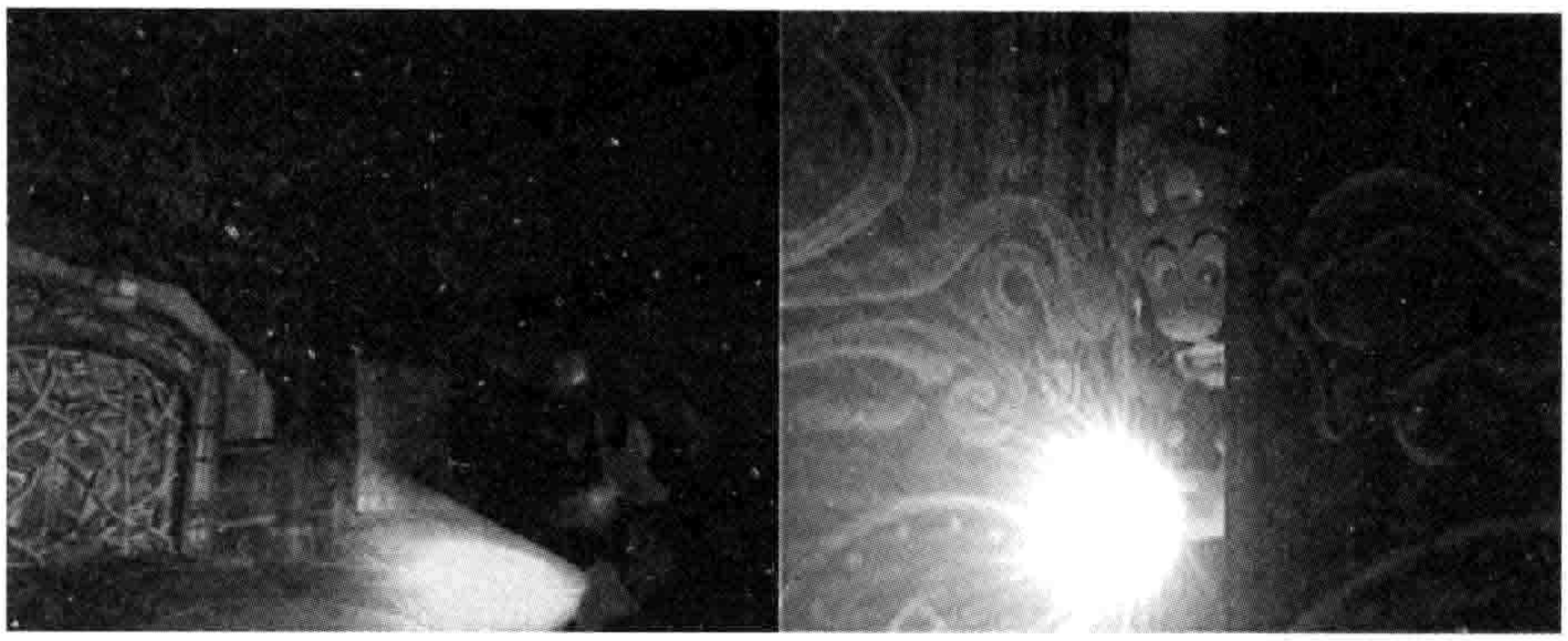


图 10.29: 《路易士鬼屋》中的手电筒由多个视觉效果组成，包括一个模拟光线的圆锥形半透明几何体、一个动态聚光源照亮场景、一个位于透镜的放射光贴图，以及朝向摄像机的镜头光晕。

光元件对应屏幕中一个像素。所谓渲染，可以理解为每个虚拟感光元件记录光强度和颜色的过程。

10.1.4.1 观察空间

虚拟摄像机的焦点，是**观察空间**（view space）或称为**摄像机空间**（camera space）的三维坐标系统的原点。摄像机通常“看着”观察空间中的正或负z轴，其y轴向上，x轴则向左或右。图10.30显示了典型的左手、右手观察空间的各轴。

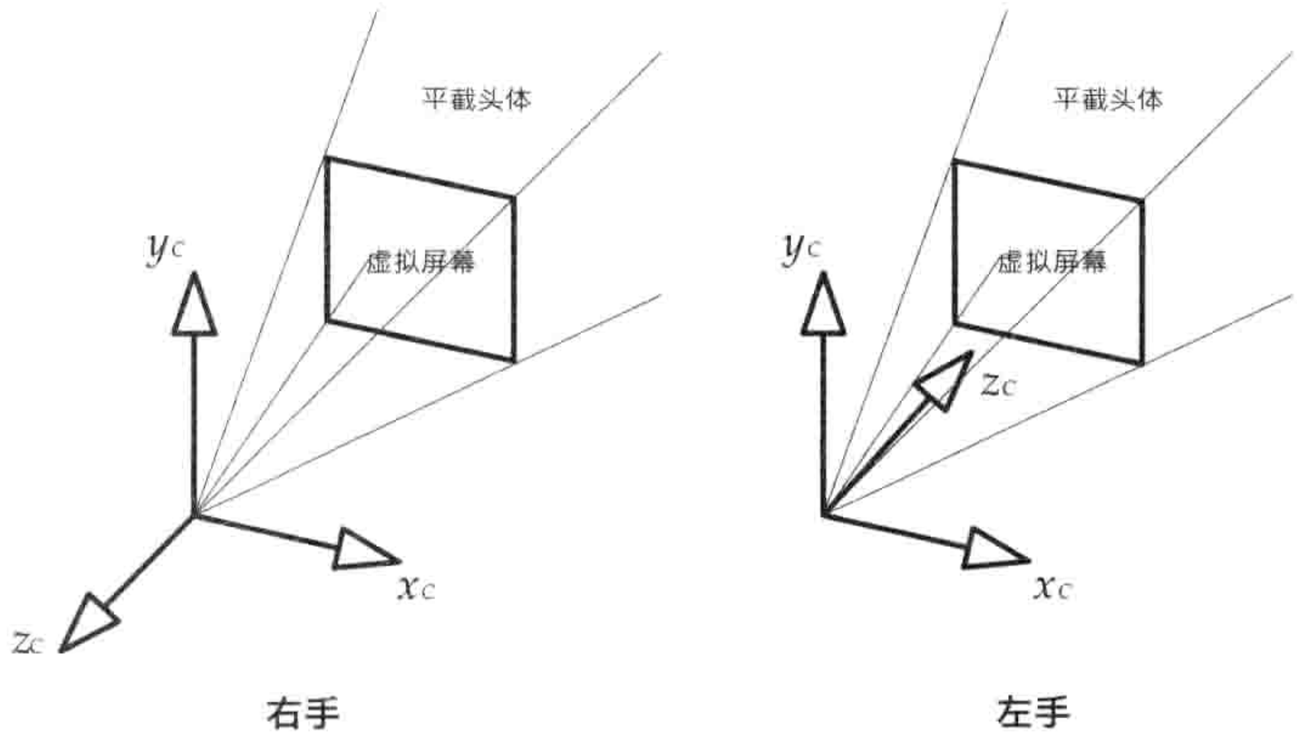


图 10.30: 左手和右手摄像机空间的各轴。

摄像机位置及定向可以用观察至世界矩阵（view-to-world matrix）表示，如同一个场景中的网格实例使用模型至世界矩阵（model-to-world matrix）表示其位置及定向。若我们

知道摄像机的位置矢量，以及摄像机空间的3个单位基矢量，而这些矢量都是以世界坐标表示的，那么观察至世界矩阵可写成以下形式，和建立模型至世界矩阵[37]的方式相同：

$$\mathbf{M}_{V\to W} = \left[\begin{array}{c|c} \mathbf{i}_V & 0 \\ \mathbf{j}_V & 0 \\ \mathbf{k}_V & 0 \\ \hline \mathbf{t}_V & 1 \end{array}\right]$$

当要渲染三角形网格，其顶点首先从模型空间变换至世界空间，然后再从世界空间变换至观察空间。进行后者时，我们需要世界至观察矩阵（world-to-view matrix），这其实就是观察至世界矩阵的逆矩阵。此矩阵有时候称作**观察矩阵**（view matrix）：

$$\mathbf{M}_{W\to V} = (\mathbf{M}_{V\to W})^{-1} = \mathbf{M}_{\text{view}}$$

这里需要小心。摄像机矩阵相对场景中的物体矩阵来说是逆矩阵，这点经常会让游戏开发新手困惑及造成软件缺陷。

在渲染个别网格实例前，通常会预先串接世界至观察矩阵和该实例的模型至世界矩阵。在OpenGL中这个串接后的矩阵称为模型观察矩阵（model-view matrix）。预计算此矩阵后，渲染引擎就只需对每个顶点做一次矩阵乘法，便能把顶点由模型空间变换至观察空间：

$$\mathbf{M}_{M\to V} = \mathbf{M}_{M\to W}\mathbf{M}_{W\to V} = \mathbf{M}_{\text{model-view}}$$

10.1.4.2 投影

为了把三维场景渲染成二维影像，我们需使用一种特别的变换——**投影**（projection）。计算机图形学中，**透视**投影（perspective projection）是最常见的投影，因为它能模仿典型摄像机的成像方式。使用这种投影时，物体显得远小近大，此效果称为**透视收缩**（perspective foreshortening）。

有些游戏采用能维持长度不变的**正射**投影（orthographic projection），主要用作渲染三维模型或游戏关卡的**平面图**（plan view，如正面图、侧面图、俯视图等），供编辑之用；也会用来在屏幕上叠加二维图形，以渲染HUD或类似的显示。图10.31比较了用这两种投影渲染立方体的结果。

[37]译注：原文为model-to-view matrix，应为笔误。

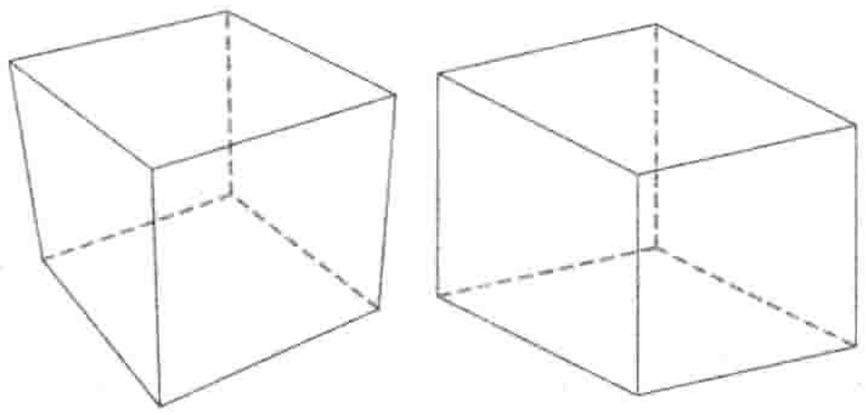


图 10.31: 使用透视投影（左图）和正射投影（右图）渲染的立方体。

10.1.4.3 观察体积及平截头体

摄像机能“看到”的空间范围称为观察体积（view volume）。观察体积由6个平面定义。**近平面**（near plane）对应于虚拟影像感光元件的表面。上、下、左、右4个平面对应虚拟屏幕的边缘。而**远平面**（far plane）则用作渲染优化，确保很远的物体不获渲染。远平面也作为深度缓冲的深度上限（见10.1.4.8节）。[38]

当使用**透视**投影渲染场景时，其观察体积的形状是截断的四角锥体，此形状有其独特名称**平截头体**（frustum）。当使用正射投影，其观察体积就是长方体。图10.32和图10.33分别说明了透视投影和正射投影。

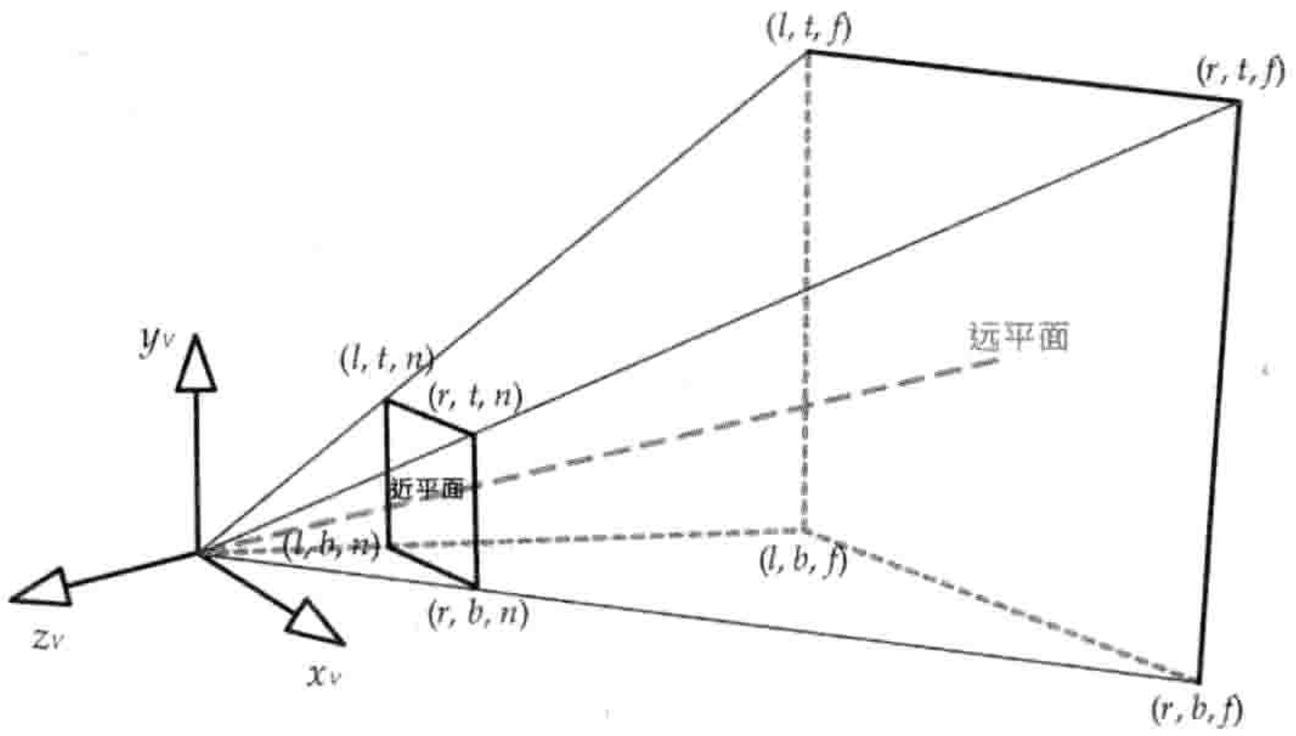


图 10.32: 一个透视的观察体积（平截头体）。

观察体积的6个平面可紧凑地用6个四维矢量$(n_{xi}, n_{yi}, n_{zi}, d_i)$表示，当中$\mathbf{n} = n_x, n_y, n_z$为平面法线，而$d$为平面和原点的垂直距离。若使用点法式（point-normal form）去表示平面，则可以用6对矢量$(\mathbf{Q}_i, \mathbf{n}_i)$去表示该6个平面，当中$\mathbf{Q}$为平面上任意的点，而$\mathbf{n}$为平面法矢量。（在两种表示法中，$i$都是指平面的索引。）

[38]译注：这里的说明主要是基于光栅化管道的常见做法。在光栅化管道中，远平面也可设置成无限远（http://www.terathon.com/gdc07_lengyel.pdf）。而光线追踪式渲染没必要设置近平面，其观察体积可以是无限长的四角锥体。

10.1.4.4 投影及齐次裁剪空间

透视及正射投影能把点从观察空间变换至一个称为**齐次裁剪空间**（homogeneous clip space）的坐标系。此三维空间其实仅是观察空间的变形版本。裁剪空间是用来把摄像机空间的观察体积转换成标准的观察体积，转换后的体积不仅独立于把三维场景转换成二维屏幕空间的**投影**类型，也独立于屏幕的**分辨率**（resolution）及**长宽比**（aspect ratio）。

在裁剪空间中，标准的观察体积是一个长方体，其x轴和y轴的范围都是-1～$+1$。在z轴方向，观察体积不是从-1延伸至$+1$（OpenGL），就是0～1（DirectX）。我们称此坐标系统为“裁剪空间”，因为观察体积的平面是和坐标系统里的轴对齐的，使按观察体积裁剪三角形变得方便（即使采用透视投影亦然）。图10.34显示了OpenGL的标准裁剪空间观察体积。注意让裁剪空间的z轴往屏幕里延伸，y轴向上，x轴向右。换句话说，齐次裁剪空间通常是**左手**坐标系统。

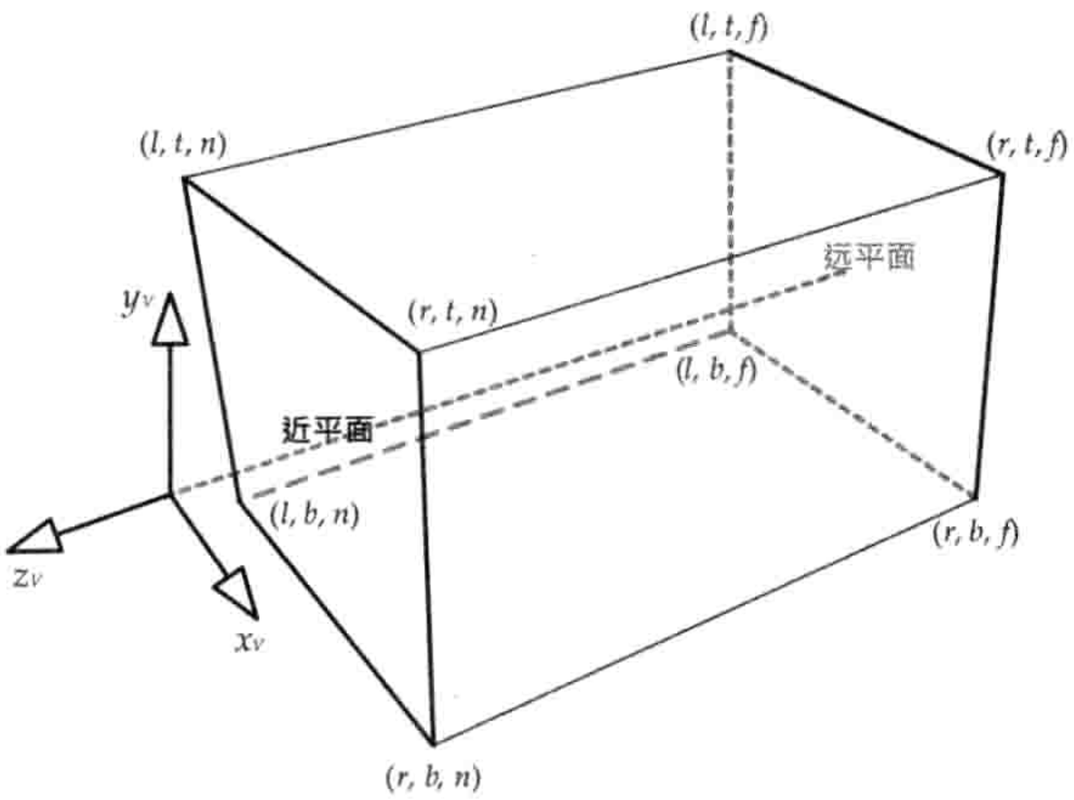


图 10.33: 一个正射的观察体积。

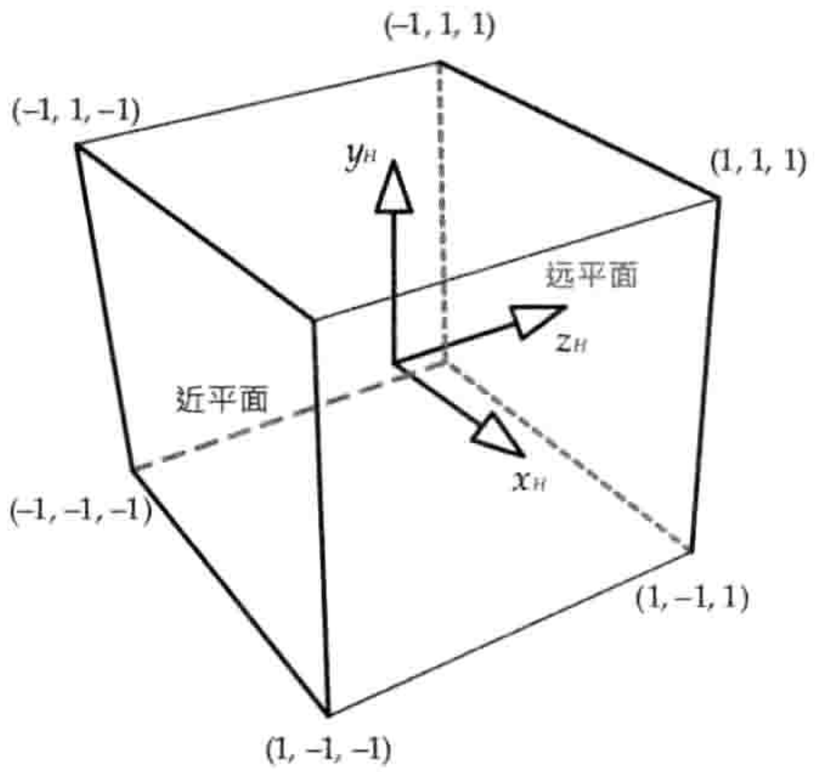


图 10.34: 齐次裁剪空间的标准观察体积。

透视投影

在[28]的4.5.1节里极好地解释了透视投影，因此我们不在此重复。取而代之，以下我们会简单地介绍透视投影矩阵$\mathbf{M}_{V\to H}$。（下标$V \to H$表示此矩阵用于把点从观察空间变换至齐次裁剪空间。）若我们的观察空间为右手坐标系，那么近平面在z轴相交于$z = -n$，而远平面则和z轴相交于$z = -f$。虚拟屏幕的左右下上边缘则分别位于近平面的$x = l$、$x = r$、$y = b$及$y = t$。（通常虚拟屏幕的中心位于摄像机空间的z轴，这种情形下$l = -r$、$b = -t$，但这并非最一般化的情形。）使用这些定义后，OpenGL用的透视投影矩阵为：

$$\mathbf{M}_{V\to H}=\begin{bmatrix}\frac{2n}{r-l} & 0 & 0 & 0\\ 0 & \frac{2n}{t-b} & 0 & 0\\ \frac{r+l}{r-l} & \frac{t+b}{t-b} & -\frac{f+n}{f-n} & -1\\ 0 & 0 & -\frac{2nf}{f-n} & 0\end{bmatrix}$$

DirectX裁剪空间的z轴范围是$[0,1]$，而非OpenGL所采用的$[-1,1]$。我们可以轻易地调整透视矩阵，使其适用于DirectX：

$$(\mathbf{M}_{V\to H})_{\text{DirectX}}=\begin{bmatrix}\frac{2n}{r-l} & 0 & 0 & 0\\ 0 & \frac{2n}{t-b} & 0 & 0\\ \frac{r+l}{r-l} & \frac{t+b}{t-b} & -\frac{f}{f-n} & -1\\ 0 & 0 & -\frac{nf}{f-n} & 0\end{bmatrix}$$

除以Z

进行透视投影后，每个顶点的x和y坐标会除以其z坐标。这个除法是产生透视收缩的方法。要了解为何该距阵能做成这样的结果，可以尝试把观察空间的点$\mathbf{p}_V$以四维齐次坐标表示，再乘以OpenGL的透视投影矩阵：

$$\mathbf{p}_H=\mathbf{p}_V\mathbf{M}_{V\to H}=\begin{bmatrix}p_{Vx} & p_{Vy} & p_{Vz} & 1\end{bmatrix}\begin{bmatrix}\frac{2n}{r-l} & 0 & 0 & 0\\ 0 & \frac{2n}{t-b} & 0 & 0\\ \frac{r+l}{r-l} & \frac{t+b}{t-b} & -\frac{f+n}{f-n} & -1\\ 0 & 0 & -\frac{2nf}{f-n} & 0\end{bmatrix}$$

相乘的结果有以下形式：

$$\mathbf{p}_H=\begin{bmatrix}a & b & c & -p_{v_z}\end{bmatrix}\tag{10.1}$$

当要把齐次坐标转换为三维坐标时，要把其x、y、z分量除以w分量：

$$\begin{bmatrix} x & y & z & w \end{bmatrix} \equiv \begin{bmatrix} \frac{x}{w} & \frac{y}{w} & \frac{z}{w} \end{bmatrix}$$

因此，把等式(10.1)除以其齐次w分量——即实际上是观察空间z坐标的负数$-p_{V_z}$，得出：

$$\begin{aligned} \mathbf{p}_H &= \begin{bmatrix} \frac{a}{-p_{Vz}} & \frac{b}{-p_{Vz}} & \frac{c}{-p_{Vz}} \end{bmatrix} \\ &= \begin{bmatrix} p_{H_x} & p_{H_y} & p_{H_z} \end{bmatrix} \end{aligned}$$

所以，齐次裁剪空间坐标都是除以观察空间的z坐标，而这样能产生透视收缩。

透视正确的顶点属性插值

在10.1.2.4节中，我们已学过对顶点属性插值，从而得出三角形内合适的属性值。属性插值是在**屏幕空间**中进行的。我们向屏幕上的每个像素迭代，并计算对应于**三角形表面**位置的每个属性值。当以透视投影渲染场景时，我们必须谨慎地进行此插值，把透视收缩的影响计算在内。这种插值称为**透视校正**插值（perspective-correct interpolation）。

推导透视正确插值已超出本书范围，但简单地说，需要把插值后的每顶点属性值除以对应的z坐标（深度）。对于每对顶点属性A_1及A_2，可写出位于它们之间t百分比的插值属性：

$$\frac{A}{p_z} = (1-t)\frac{A_1}{p_{z1}} + t\frac{A_2}{p_{z2}} = \text{LERP}\left(\frac{A_1}{p_{z1}}, \frac{A_2}{p_{z2}}, t\right)$$

关于透视正确插值的数学推导可参考[28]。

正射投影

正射投影可使用以下的矩阵：

$$(\mathbf{M}_{V\to H})_{\text{ortho}} = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & -\frac{2}{f-n} & 0 \\ -\frac{r+l}{r-l} & -\frac{t+b}{t-b} & -\frac{f+n}{f-n} & 1 \end{bmatrix}$$

这其实只是常见的缩放并平移矩阵。（其左上3×3含有对角非统一缩放矩阵，而最下面一行含有平移。）由于此观察体积在观察空间和裁剪空间都是长方体，我们仅需要缩放及平移使顶点在这两个空间中转换。

10.1.4.5 屏幕空间及长宽比

屏幕空间是二维坐标系统，其轴是以屏幕像素来量度的。原点通常位于屏幕左上角，x轴向右，y轴向下。（y轴方向和一般数学定义相反，这是由于CRT显示器是从屏幕上方往下扫描的。）屏幕的宽度和高度之比称为**长宽比**（aspect ratio）。最常见的长宽比为$4:3$（传统电视屏幕的长宽比）及$16:9$（电影屏幕[39]或高清电视的长宽比。）图10.35展示了这些长宽比。

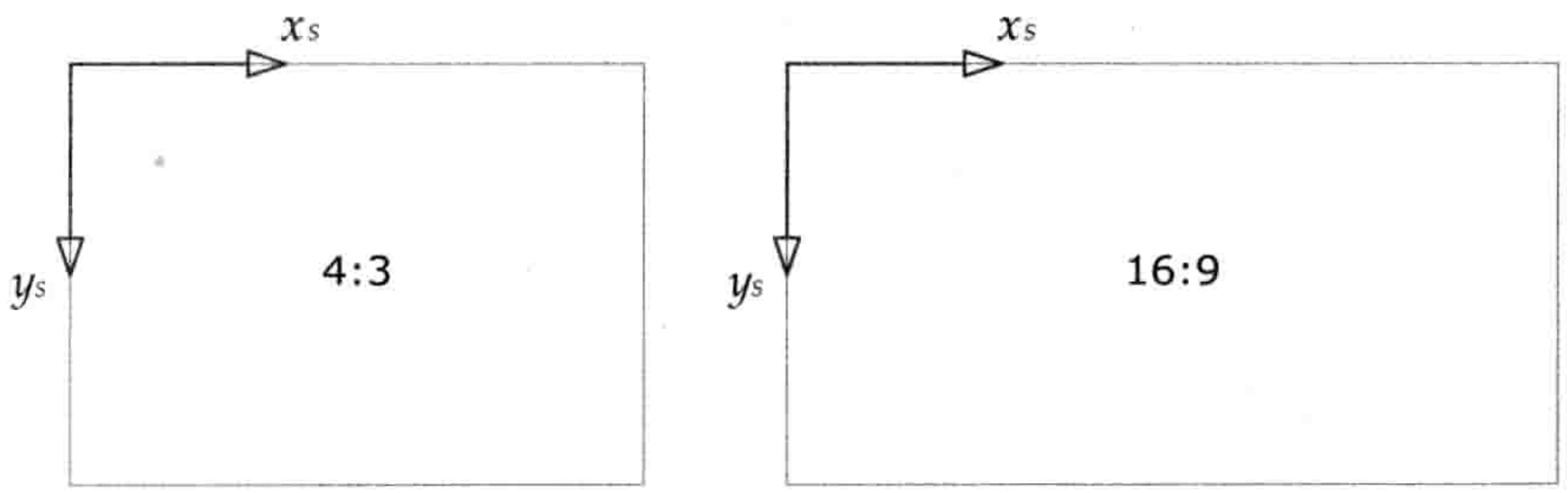


图 10.35: 两个最常见的屏幕长宽比是$4:3$和$16:9$。

渲染以齐次裁剪空间表示的三角形时，可以只用其(x, y)坐标而忽略z。但在渲染之前，我们要缩放及平移裁剪空间坐标，使这些坐标变换成在屏幕空间而非在单位正方形里。此缩放平移操作称为**屏幕映射**（screen mapping）。

10.1.4.6 帧缓冲

最终渲染后的影像会储存在一个名为**帧缓冲**（frame buffer）的颜色位图缓冲里。虽然多数显卡支持多种帧缓冲格式，但像素颜色通常以RGBA8888格式储存。其他常用的格式包括RGB565、RGB5551，以及一些调色板模式。

显示硬件（CRT、平板屏幕显示器、高清电视等）会周期性地读取帧缓冲的内容，北美及日本所使用的NTSC电视的读取频率是60Hz，而欧洲及许多其他地区所使用的PAL/SECAM电视则是50Hz。渲染引擎通常会维护至少两个帧缓冲。当中，显示硬件扫描一个帧缓冲时，渲染引擎则更新另一个帧缓冲。此称为**双缓冲法**（double buffering）。

[39]译注：电影的常见长宽比为$1.85:1$和$2.39:1$，前者较接近$16:9$（$16:9$即约$1.77:1$）。

通过在**垂直消隐区间**（vertical blanking interval，即CRT电子枪向屏幕左上角重置期间）互换两个缓冲，双缓冲就能确保显示硬件一直能扫描完整的帧缓冲。这么做能避免一个名为**撕裂**（tearing）的不良效果，当中屏幕上半部分含有最新渲染的影像而下半部分则仍是上一帧的残留影像。

有些引擎共使用3个帧缓冲，此技术称为**三缓冲法**（triple buffering）。这么做的原因是，就算显示硬件仍在扫描上一帧，渲染引擎已经能开始渲染下一帧。例如，当引擎完成缓冲B的渲染时，硬件可能仍在扫描缓冲A，这时候引擎就可以继续把新的帧渲染至缓冲C，而不需要闲置等待显示硬件完成缓冲A的扫描。

渲染目标

任何供渲染引擎绘画图形的缓冲皆称为**渲染目标**（render target）。在本章稍后会提及，渲染引擎除了使用帧缓冲，还会使用许多种类的屏幕外（off-screen ）渲染目标。这些渲染目标包括深度缓冲（depth buffer）、模板缓冲（stencil buffer），以及其他用来储存中间渲染结果的缓冲。

10.1.4.7 三角形光栅化及片段

要在屏幕上产生一个三角形的影像，我们需要给该三角形范围内的像素填充数据。此过程称为光栅化（rasterization）。在光栅化过程中，三角形表面会分拆成名为**片段**（fragment）的小块，每个片段对应三角形表面中的一个细小区域，而每个这些细小区域对应于单个屏幕像素。（在使用多采样抗锯齿时，每个片段对应于像素的**一部分**，详情见下文。）

片段像是培训过程中的像素。把一个片段写进帧缓冲之前，它要通过许多测试（下文会更详细讨论）。若片段在任何一个测试中不获通过，它就会被丢弃。能通过所有测试的片段就会被着色（即决定其颜色），接着其颜色就会写进帧缓冲，或是和已经在帧缓冲的颜色进行混合。图10.36说明如何把片段变成像素。

抗锯齿

当光栅化一个三角形时，其边缘可能显得像锯齿般不圆滑。我们都熟悉并喜爱（或憎恨）这种“梯级”效果。从技术上来说，这些混叠（aliasing）之所以产生，是由于我们使用离散的像素集对原本圆滑的、连续的二维信号进行**采样**（sample）。（在频域中，采样导致信号偏移并在频率轴方向多次复制。混叠的字面意思就是指信号重叠后，使信号之间产生混淆。）

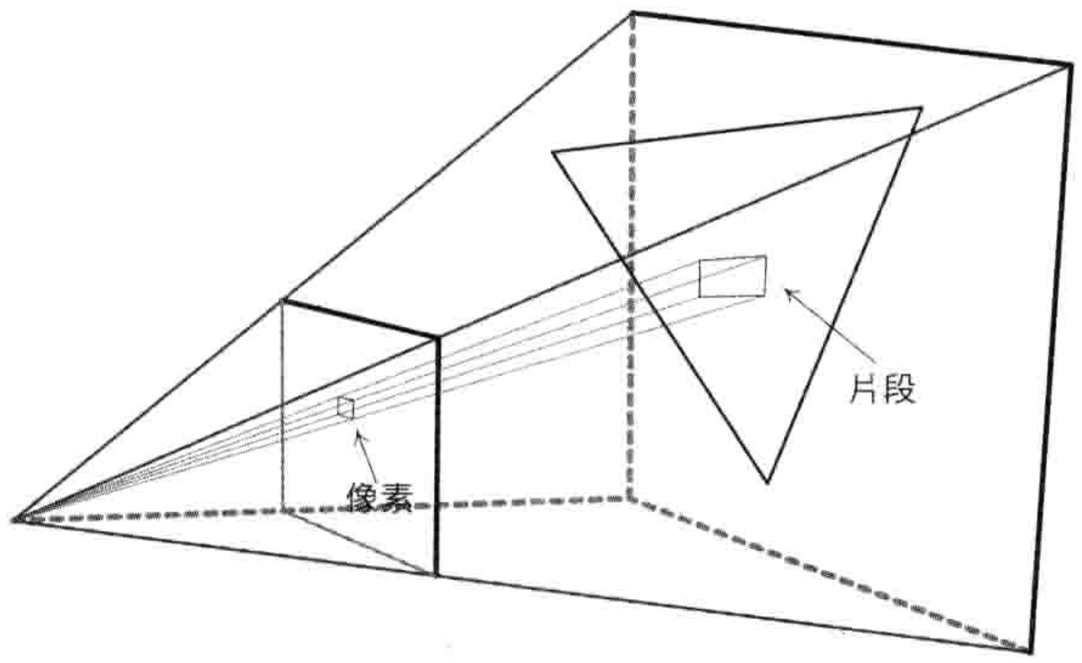


图 10.36: 片段是三角形对应屏幕像素的一个细小区域。在渲染管道中，一个片段若没有被丢弃，其颜色就会写入帧缓冲之中。

抗锯齿（antialiasing）[40]是一种降低由混叠所形成的视觉缺乏的技术。抗锯齿的效果是，三角形边缘与帧缓冲附近的颜色混合起来。

有多种方法对三维渲染影像进行抗锯齿处理。在**全屏抗锯齿**（full-screen antialiasing，FSAA）[41]中，影像渲染至比实际屏幕宽一倍、高一倍[42]的帧缓冲里。然后再把该帧缓冲缩减采样（downsample）至所需的分辨率[43]。FSAA很耗时，因为渲染长宽两倍大小的帧等于渲染4倍像素。FSAA的帧缓冲也需消耗正常帧缓冲4倍的内存。

多数现在的图形硬件能够进行抗锯齿渲染，而又不需要实际渲染长宽两倍大小的影像，当中用到的技术称为**多重采样抗锯齿**（multisample antialiasing，MSAA）。其基本原理是把每个像素分拆为多个片段。这些片段在管道最后阶段结合成单个像素[44]。（现在的GPU通常能支持4x及8x的多重采样。）[45]

---

[40]译注：此术语能意译为反锯齿、抗锯齿、边缘柔化，或是直译为抗混叠。

[41]译注：此术语也称为超采样抗锯齿（super sample antialiasing，SSAA）。

[42]译注：两倍宽、两倍高的FSAA称为2 × 2或4x。实际上只要内存和时间足够，多少倍都可行。

[43]译注：以2 × 2 FSAA为例，缩减采样是指计算每组2 × 2像素的平均值，再把这些值写入原来大小的帧缓冲。

[44]译注：此处原文称MSAA不用两倍宽、两倍高的帧缓冲，并不正确。译者在此补充一下MSAA的原理。其实MSAA仅仅是FSAA的一种优化。其优化的地方在于，深度和stencil采用超采样方式进行测试（如每像素做2 × 2个采样），但像素着色只是正常采样（即每像素只做1个采样）。最后如同超采样抗锯齿方法一样，把帧缓冲缩减采样至目标的分辨率。MSAA的优化能显著减少着色计算。

[45]译注：由于近年流行的延迟渲染（deferred rendering）不能使用MSAA，因此多种新的抗锯齿技术应运而生，例如，形态抗锯齿（morphological antialiasing，MLAA）、方向局部化抗锯齿（directionally localized antialiasing，DLAA）、快速逼近抗锯齿（fast approximate antialiasing，FXAA）、次像素重建抗锯齿（subpixel reconstruction antialiasing，SRAA）等。

10.1.4.8 遮挡及深度缓冲

当渲染两个在屏幕空间重叠的三角形时，我们需要一些方法来保证较近摄像机的三角形渲染在另一个之上。为达到此目的，我们可以把三角形以从后往前的次序渲染，此方法称为画家算法（painter's algorithm）。然而，如图10.37所示，当三角形互相穿插时此方法并不适用。[46]

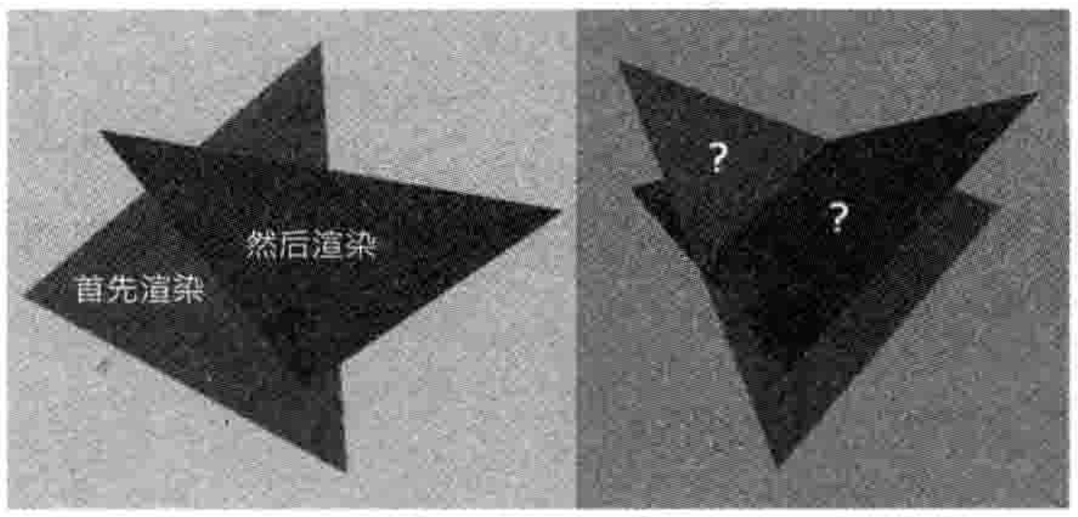


图 10.37: 画家算法以从后往前的次序渲染三角形，以产生正确的三角形遮挡。然而，当三角形互相穿插时此算法就不适用了。

要正确地实现三角形的遮挡（occlusion）关系，而不需要理会三角形的渲染次序，渲染引擎会使用称为**深度缓冲**（depth buffer，或称*z*缓冲/*z*-buffer）的技术。深度缓冲是全屏缓冲，当中每个像素含16或24位的深度数据。每个片段含有一个*z*坐标，以量度其“深入”屏幕的深度。（片段的深度是从对三角形顶点深度插值所得。）当片段的颜色写至帧缓冲，其深度就会储存在对应的深度缓冲像素里。而当另一片段（自另一个三角形）渲染在同一像素时，引擎就会比较新的片段深度和深度缓冲里的现存深度。若新片段比较接近摄像机（即其深度较小），其颜色及深度就会写进帧缓冲。否则该片段就会被丢弃。

深度冲突及W缓冲

当渲染互相非常接近的平行表面时，渲染引擎必须能够分开这两个平面的深度。若深度缓冲有无限的精确度，这并不会造成问题。可惜，现实的深度缓冲仅含有限的精确度，因此两个足够接近的平面，其两个深度可能会变成相同的离散值。若发生这种情形，较远的平面像素就会“刺穿”较近的平面，造成一个称为**深度冲突**（*z*-fighting）的噪点效果。

为了使整个场景的深度冲突降至最低，我们希望无论要渲染的平面是远是近，其深度都有相同的精确度。然而，*z*缓冲并非如此。裁剪空间的*z*深度（p_{Hz}）并非均匀分布于近平面和远平面之间，因为该深度是观察空间的*z*坐标的倒数。而由于$1/z$曲线的形状，深度缓冲的

[46]译注：另一个例子是4个三角形以“井”字形排列，每个三角形同时在另一个三角形之上，也同时在另一个三角形之下，虽然没有相交，也不能从后往前排序。

大部分精确度集中于近摄像机的地方。

图10.38绘画的函数$p_{Hz} = 1/p_{Vz}$显示了此特性。接近摄像机时，两平面的观察空间距离Δp_{Vz}变换成相当大的裁剪空间距离Δp_{Hz}。但当离开摄像机时，同样的分隔在变换后变成细小的裁剪空间距离。此结果造成深度冲突，并且当物体越离开摄像机情况越见严重。

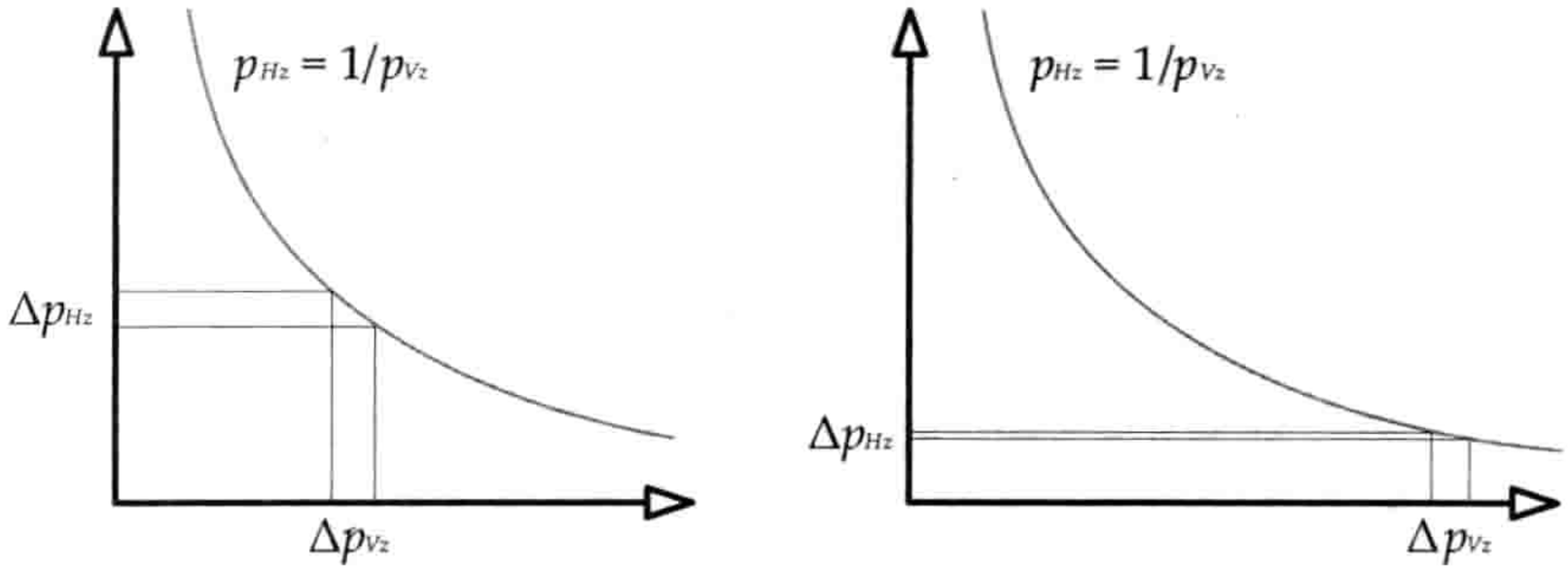


图 10.38: 为函数$1/p_{Vz}$作图，显示出大部分的精确度是置于近摄像机之处。

要克服此问题，我们希望在深度缓冲储存**观察空间**的z坐标（p_{Vz}），而非裁剪空间的z坐标（p_{Hz}）。观察空间z坐标随摄像机距离线性变化，因此使用观察空间z坐标能达到整个深度范围内都具均匀精确度。此技术称为w**缓冲**（w-buffer），因为观察空间z坐标恰好出现在齐次裁剪空间坐标的w分量里。（回想方程(10.1)中$p_{Hw} = -p_{Vz}$）。

此处的术语可能非常令人混淆。z缓冲和w缓冲储存的是**裁剪空间**的坐标。但依**观察空间**的角度来说，z缓冲储存的是$1/z$（即$1/p_{Vz}$），而w缓冲储存的是z（即p_{Vz}）!

在此也要注意一点，w缓冲方式比z缓冲方式稍耗时一点。因为在w缓冲中我们不能直接对深度插值。深度必须先计算其倒数才能插值，然后再算倒数以储存于w缓冲。

10.2 渲染管道

在对三角形光栅化的主要理论及实践基础有大概理解之后，我们将注意力转向如何实现整个渲染过程。在实时游戏渲染引擎中，10.1节已提及，高级的渲染步骤是由名为**管道**（pipeline）的软件架构所实现的。管道只是一连串的顺序计算阶段（stage），每个阶段有其具体目的，各个阶段会操作输入流中的数据项，并对输出流产生数据。

管道中每个阶段的操作过程通常独立于其他阶段。因此，管道化架构的最大优点在于非常适合并行化（parallelization）。正当第1个阶段在咀嚼一个数据元素，第2个阶段可以处理先前第1个阶段的结果，这种并行性一直延伸至往后的阶段。

并行也可以在管道的个别阶段中实现。例如，在计算机硬件中可以把某阶段在芯片上复制N份，那么N个数据元素就能在该阶段并行。图10.39展示了一个并行管道。理想地，所有阶段都能（大部分时间里）并行运行，而一些阶段能够同时操作多个数据项。

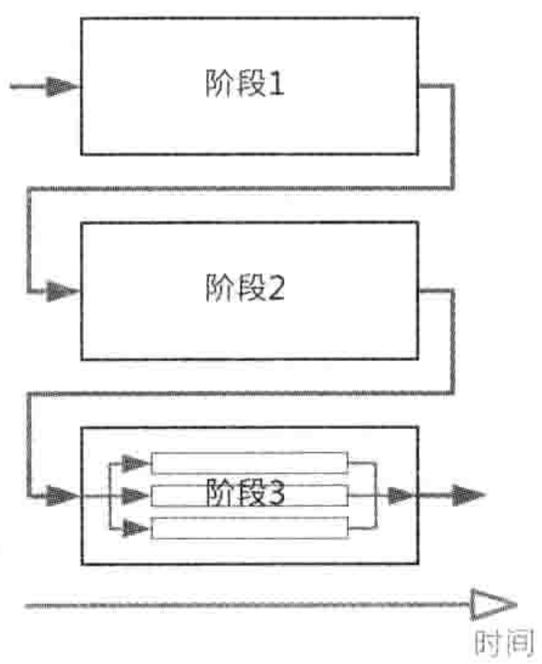


图 10.39: 一个并行管道中所有阶段都会并行执行，而且有些阶段能同时操作多个数据项。

管道的**吞吐量**（throughput）量度总体每秒可产生的数据量。而管道的**潜伏期**（latency）量度单个数据需要花多少时间才能走完整个管道。个别阶段的潜伏期量度该阶段需花多少时间处理单个数据。最低吞吐量的阶段成为整个管道的吞吐量上限，同时也对整个管道的平均潜伏期有影响。因此，设计渲染管道时，应尽量降低及平衡整个管道里的潜伏期，以消除瓶颈[47]。在优良设计的管道中，所有阶段同时运作，没有阶段需要长时间闲置等待另一个阶段。

10.2.1 渲染管道概观

有些图形学文献会把渲染管道分割为3个概要阶段。在本书中，我们会把管道往后延伸，

[47]译注：理论上来说，管道的设计必然出现瓶颈，目标应该是减少瓶颈造成的影响。

以包含脱机工具所创建场景的阶段，这些场景最终会由游戏引擎渲染。我们的管道中，最高级的阶段包括：

1. **工具阶段（脱机）**：定义几何和表面特性（材质）。
2. **资产调节阶段（脱机）**：资产调节管道处理几何和材质数据，生成引擎可用的格式。
3. **应用程序阶段（CPU）**：识别出潜在可视的网格实例，并把它们及其材质呈交至图形硬件以供渲染。
4. **几何阶段（GPU）**：把顶点变换、照明，然后投影至齐次裁剪空间。可选择用几何着色器处理三角形，然后对三角形根据平截头体进行裁剪。
5. **光栅化阶段（GPU）**：把三角形转换为片段，并对片段着色。片段经过多种测试（深度测试、alpha测试、模板测试等）后，最终和帧缓冲混合。

10.2.1.1 渲染管道如何变换数据

我们有兴趣知道几何数据经过渲染管道时，其数据格式是如何改变的。工具和资产调节阶段负责处理网格和材质。应用程序阶段负责处理网格实例和子网格，每个子网格关联至一个材质。而在几何阶段，每个子网格分解成个别顶点，顶点获大规模并行处理。在这一阶段的结尾，完全变换后、着色后的顶点会重新构成三角形。在光栅化阶段，每个三角形分解为片段，这些片段若没被丢弃，其颜色便会最终写进帧缓冲。图10.40说明了此过程。

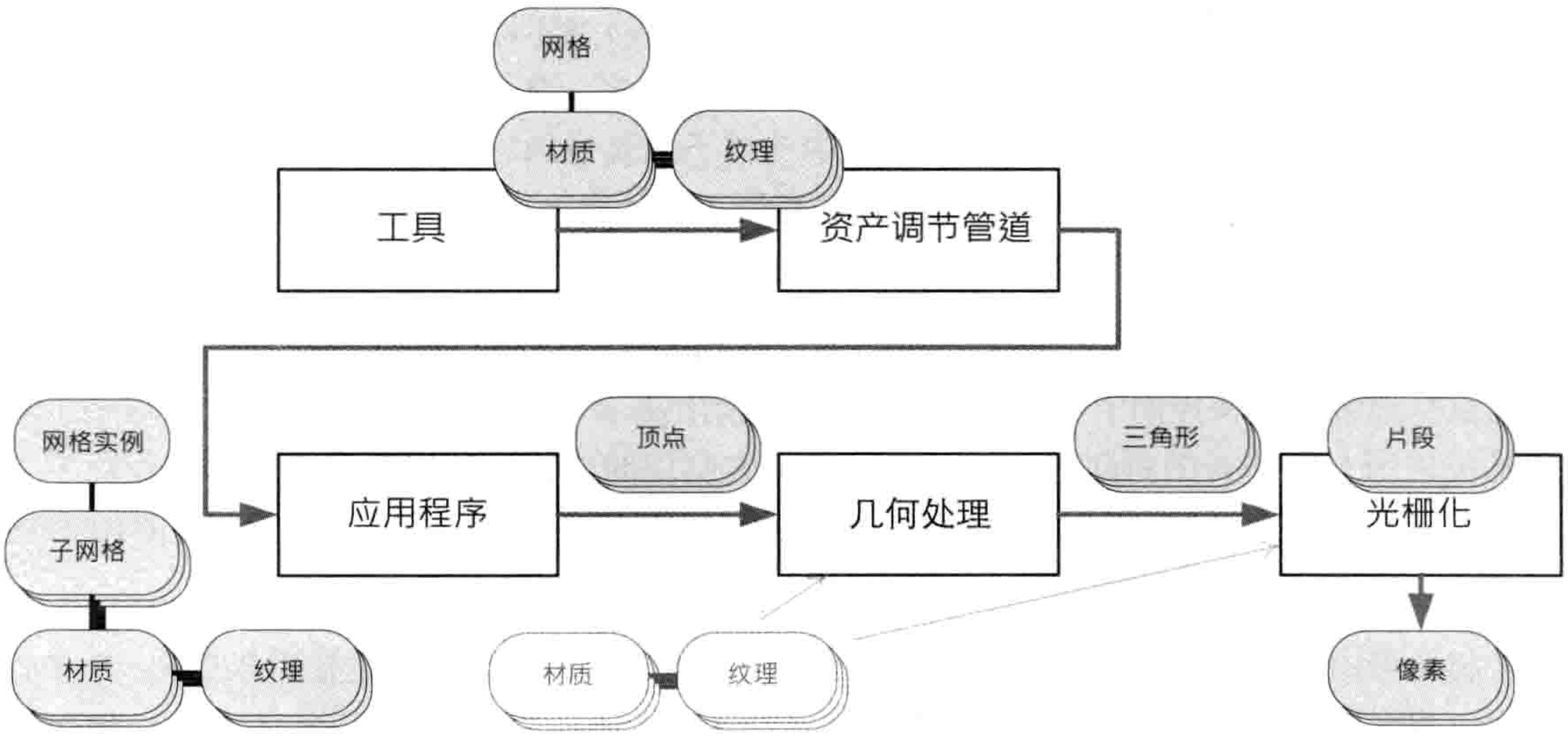


图 10.40: 当几何数据通过渲染管道的多个阶段时，其格式会彻底改变。

10.2.1.2 管道的实现

前两个渲染管道阶段以脱机方式实现，通常由PC或Linux机器执行。应用程序阶段则是由游戏机或PC的主CPU执行，或由像PS3 SPU这些并行处理器执行。几何及光栅化阶段通常由图形处理器（graphics processing unit，GPU）处理。以下我们会探索实现这些阶段的部分细节。

10.2.2 工具阶段

在工具阶段，三维建模师在**数字内容创作**（digital content creation，DCC）软件里制作三维模型，这些软件例如有Maya、3ds Max、LightWave、Softimage/XSI、SketchUp等。这些模型可以由任何方便的表面描述方式定义，例如NURBS、四边形、三角形等。然而，在管道的运行时渲染前，这些模型总要先镶嵌成三角形。

网格的顶点也可以蒙皮。蒙皮需要把每个顶点关联至骨骼结构上的一个或多个关节，每个顶点含有对每个关联关节的影响权重。动画系统会使用蒙皮信息及骨骼驱动模型的动作，详情请参阅第11章。

在工具阶段，美术人员也需要定义材质。这包括为每个材质选择着色器，选取该着色器所需的纹理，以及设置着色器提供的配置参数及选项。纹理会被贴至表面。另外也可以定义一些顶点属性，通常会使用DCC应用提供的直觉工具“涂上”这些属性。

制作材质时，通常会使用商用或公司内定制的材质编辑器。材质编辑器有时候会以插件方式直接整合在DCC工具里，也可以是一个独立程序。有些材质编辑器能现场连接至游戏，使材质制作人能观看材质在真实游戏中的样子。其他的材质编辑器提供脱机的三维预览视域。有些编辑器甚至能让美术人员和着色器工程师直接在编辑器中编写及调试着色器。NVIDIA的FX Composer是其中的一个例子，见图10.41。

虚幻引擎3提供强大的图形化着色语言（graphical shading language）[48]。这种工具可以用鼠标连接不同种类的节点，以建立视觉效果的迅速原型（rapid prototype）。这些工具通常提供所见即所得的材质显示。以图形化语言制作的着色器通常需要由图形工程师手工优化，因为图形化语言总是以性能换取其难以置信的弹性、通用性及易用性[49]。图10.42展示了虚幻引擎3的材质编辑器。

材质可由个别网格储存及管理。然而，这样会导致重复数据，以及重复的工作。许多游戏中，相对较少的材质能应用至游戏中的许多物体。例如，我们可以制定一些标准，重复使

[48] 译注：原文说FX Composer也提供图形化着色语言，并不正确，故删之。

[49] 译注：在较新版本的虚幻引擎3里，提供了一种节点可供直接编写着色器语言，以弥补视觉化编程的局限。

用木材、塑料、金属、布料、皮肤等材质。没有必要把这些材质复制至每个网格中。取而代之，许多游戏团队会建立材质库，从中给每个网格挑选合适的材质，让网格和材质维持松散的耦合。

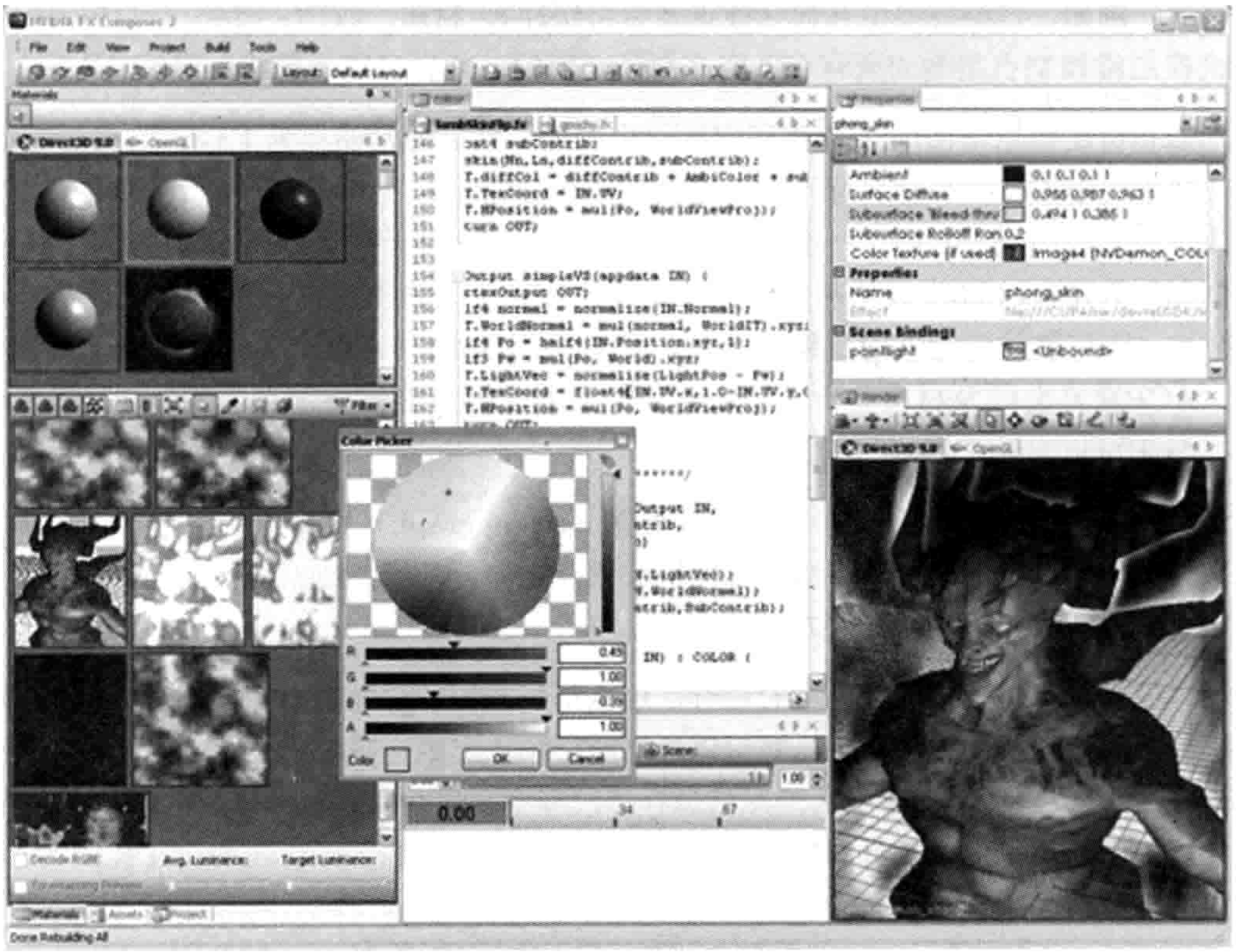


图 10.41: NVIDIA的FX Composer软件可用来轻松地编写、可视化及调试着色器。

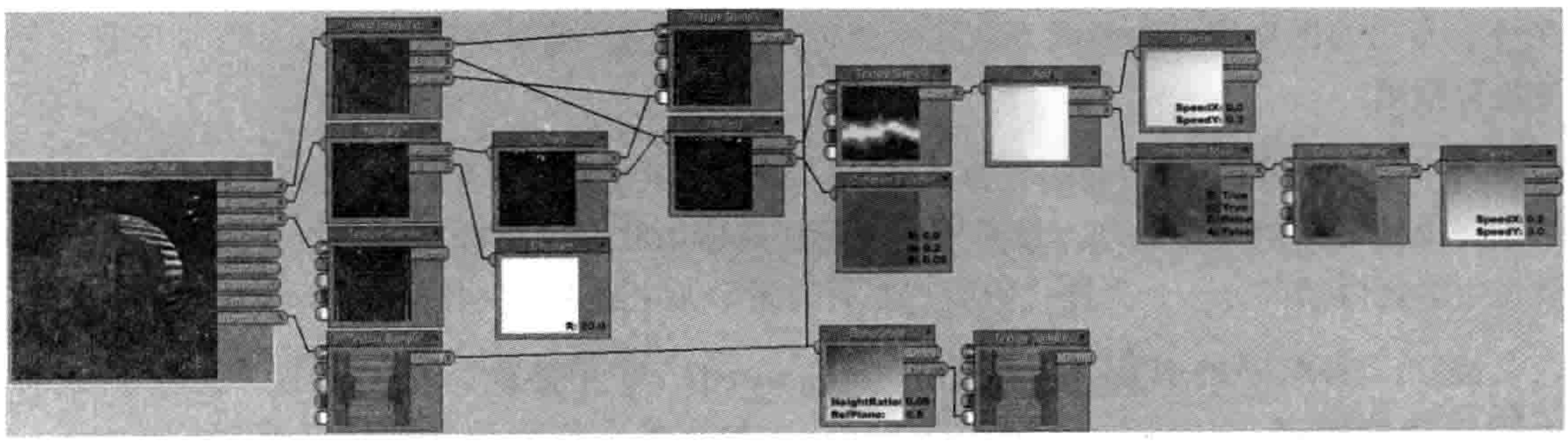


图 10.42: 虚幻引擎3的图形化着色语言。

10.2.3 资产调节阶段

资产调节阶段本身也是一个管道，有时候称为**资产调节管道**（asset conditioning pipeline，ACP）。如6.2.1.4节所述，其工作是导出、处理、链接多个种类的资产，生成内聚的整体。例如，三维模型由几何（顶点及索引缓冲）、材质、纹理、骨骼（可选）所组成。ACP确保三维模型引用到的所有个别资产都是可用的，并且已准备就绪供引擎载入。

几何和材质数据是由DCC应用程序抽取出来的，然后通常储存为平台无关的中间格式。接着，视乎引擎支持多少种目标平台，这些数据会被处理成一个或多个平台专用格式。理想地，此阶段生成的平台专用资产能直接载入内存，在运行时无须后处理，或只需要少量后处理。例如，为Xbox 360生成的网格数据应该输出成顶点及索引格式，能直接上传至显存[50]；在PS3上，几何数据可能生成压缩数据流，能直接用DMA传送至SPU进行解压。在生成资产时，ACP通常会顾及材质和着色器。例如，某着色器可能需要顶点法线、切线及副切线矢量，ACP可以自动产生这些矢量。

资产调节阶段也可能会计算高级的**场景图**（scene graph）数据结构。例如，可能会处理静态关卡几何以生成BSP树（给定摄像机位置及定向，场景图数据结构能帮助渲染引擎迅速判断哪些物体需要渲染，10.2.7.4节会再探讨）。

耗时的光照计算通常会在脱机时进行，这也是资产调节阶段的一部分。这种计算称为**静态光照**（static lighting）。静态光照可以计算网格顶点上的光照颜色（此称为顶点光照“烘焙/baking”）；也可以把每像素的光照信息存于纹理中，这些纹理称为**光照贴图**（light map）；除此以外，还可以生成**预计算辐射传输**（precomputed radiance transfer，PRT）的系数，这些通常是球谐（spherical harmonic）函数的系数。

10.2.4 GPU简史

在游戏开发的早期，所有渲染都在CPU上进行。游戏如《德军司令部》和《毁灭战士》在没有专门的图形硬件的情况下（除标准VGA显卡外），把早期PC渲染互动三维场景的程度推至极限。

由于这些及其他PC游戏的普及，硬件厂商开始开发图形硬件，把一些原来由CPU执行的工作交给专门的硬件处理。最早期的图形加速器如3Dfx的巫毒（Voodoo）系列，能处理管道中最耗时的部分——光栅化阶段。之后的图形加速器也提供几何处理阶段的支持。

最初，图形硬件只提供硬接线（hard-wired）却可配置的管道实现，这种管道称为**固定功能管道**（fixed-function pipeline）。此技术也称为**硬件变换及光照**（hardware

[50]译注：之前提及，实际上Xbox 360采用统一内存架构，基本上无主存、显存之分。

transform and lighting，hardware T & L）。之后，管道中的数个子阶段变为可编程的（programmable）。工程师能编写名为**着色器**的程序控制管道如何处理顶点（**顶点着色器**/vertex shader）及片段（**片段着色器**/fragment shader，或更常称为**像素着色器**/pixel shader）。DirectX 10又增加了第3种着色器，名为**几何着色器**（geometry shader）。这种着色器可以允许渲染工程师修改、剔除和创建整个图元（三角形、线段和点）。[51]

图形硬件已进化成一种专门的微处理器，称为**图形处理器**（graphics processing unit，GPU）。GPU为最大化管道吞吐量而设计，当中利用了庞大的并行性处理。例如，现在的GPU，如GeForce 8800，可以处理128个顶点或片段[52]。

就算GPU在完全可编程的形式下，GPU也不是通用微处理器——也不应如此。GPU能达至高处理速度（现时是每秒万亿次浮点运算（teraflop）的级数）的原因，在于仔细控制管道的数据流。有些管道阶段是完全固定功能的，有些是可配置但不能编程的。内存只能在控制范围内存取，并且使用专门的缓存把不需要的重复计算减至最少。

在以下几个小节中，我们会概要地探索当代GPU的架构，并察看渲染管道的运行时部分通常如何实现。我们主要会谈现时的GPU架构，这些架构应用于PC的最新显卡，以及Xbox 360和PS3等游戏机平台上。然而，并非所有平台都支持我们在这里会讨论到的功能。例如，Wii并不支持可编程着色器，并且大部分PC游戏也需要有后备渲染方案以支持仅有有限可编程着色器的旧显卡。

10.2.5 GPU管道

几乎所有GPU都会把管道分拆为以下所述的子阶段，如图10.43所示。图中每个阶段的各灰阶代表它的功能是可编程的、固定但能配置的，或是固定而不能配置的。

10.2.5.1 顶点着色器

此阶段是完全可编程的。顶点着色器负责变换及着色/光照顶点。此阶段的输入是单个顶点（虽然实际上会并行处理多个顶点）。顶点位置及法矢量通常以模型空间或世界空间表示。此阶段也会进行透视投影、每顶点光照及纹理计算，以及为动画角色计算蒙皮。顶点着色器也可以通过修改顶点位置来产生程序式动画（procedural animation）。例如模拟风吹草动或碧波荡漾。此阶段的输出是完成变换及光照后的顶点，其位置及法矢量是以齐次裁剪

[51] 译注：DirectX 11又增加外壳着色器（hull shader）、域着色器（domain shader）和计算着色器（compute shader）。前两者用于自定义镶嵌，后者则用于GPU通用计算。

[52] 译注：更新信息，GeForce GTX 590有1024个流处理器，Radeon HD 6970有1536个。

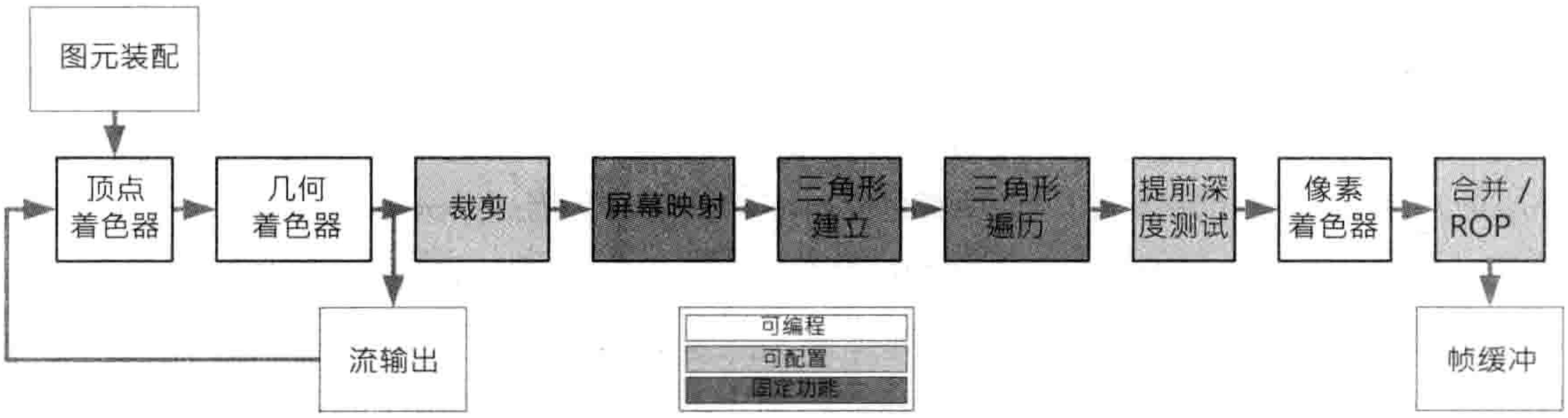


图 10.43: 典型GPU渲染管道实现的几何处理及光栅化阶段。白色阶段是可编程的，浅灰色阶段是可配置的，而深灰色阶段是固定的功能。

空间表示的（见10.1.4.4节）[53]。

在较新的GPU中，顶点着色器能完全存取纹理数据[54]，而较旧的GPU只容许像素着色器存取纹理数据。当把纹理作为独立的数据结构，例如高度图或查找表，此功能特别有用。

10.2.5.2 几何着色器

可选的几何着色阶段也是完全可编程的。几何着色器处理以齐次裁剪空间表示的整个图元（三角形、线段、点）。它能剔除或修改输入的图元，又能生成新的图元。其典型应用包括阴影体积拉伸（shadow volume extrusion，见10.3.1.1节）、渲染立方体贴图（cube map）的6个面（见10.3.1.4节）、在网格的轮廓边拉伸毛发的鳍（fur fin）、从点数据生成粒子四边形（见10.4.1节）、动态镶嵌、把线段以分形细分（fractal subdivision）模拟闪电效果、布料模拟等。

10.2.5.3 流输出

现在的GPU容许把达至此管道阶段的数据写回内存。数据能从那里回到管道之始做进一步处理。此功能称为**流输出**（stream out）。

通过使用流输出，许多迷人的视觉效果可以不经CPU实现。其中一个绝佳的例子是头发渲染。头发通常是由三次样条曲线（cubic spline curve）的集合表示的。以往头发通常在CPU上进行物理模拟，然后在CPU上把样条镶嵌为线段，最后由GPU渲染那些线段。

有了流输出，GPU便可于顶点着色器内，在头发样条的控制点上进行物理模拟。几何

[53]译注：位置必须以齐次裁剪空间表示，但法矢量的空间则可自由决定。顶点着色器输出的法线供后续的着色器所使用，例如，在像素着色器计算逐像素光照时，通常会使用世界空间或切线空间的法矢量。

[54]译注：此功能的正式术语为顶点纹理拾取（vertex texture fetch，VTF），自着色器模型3.0开始支持。

着色器则把样条镶嵌成线段，并用流输出功能把镶嵌后的顶点数据写入内存。最后那些线段被重新流入管道之始去渲染。

10.2.5.4 裁剪

裁剪（clipping）阶段把三角形在平截头体以外的部分切掉。其原理是先判定哪些顶点在平截头体以外，然后求出三角形的棱和平截头体的平面之间的相交点。这些相交点会成为一个或多个裁剪后三角形的新顶点。

此阶段是固定功能，但提供有限度配置。例如，除了平截头体的平面外，用户能定义额外的裁剪平面。此阶段也能配置剔除完全在平截头体以外的三角形。

10.2.5.5 屏幕映射

屏幕映射（screen mapping）只是简单地缩放和平移顶点，使之从齐次裁剪空间变换至屏幕空间。此阶段是完全固定且不能配置的。

10.2.5.6 三角形建立

自三角形建立（triangle setup）阶段，光栅化硬件开始迅速地把三角形转换成片段。此阶段是不能配置的。

10.2.5.7 三角形遍历

三角形遍历（triangle traversal）阶段把三角形分解为片段（即光栅化）。通常每个像素会产生一个片段，除非是使用MSAA，那么每个像素会产生多个片段（见10.1.4.7节）。三角形遍历也会对顶点属性进行插值，以产生每片段（per-fragment）属性，供像素着色器使用。有需要时，此过程会采用透视校正插值。此阶段的功能也是固定且不能配置的。

10.2.5.8 提前深度测试

许多显卡能够在管道的此时间点检查片段的深度，若某片段会被帧缓冲的像素遮挡，就在此丢弃该片段。这么做对于所有被遮挡片段，能直接跳过（可能非常耗时的）像素着色器阶段。

难以置信的事，并非所有图形硬件都支持在此管道阶段进行深度测试。在过去的GPU设计中，深度测试和alpha测试都是在像素着色器之后才进行的。因此，此阶段称为**提前z测试**（early z-test）或**提前深度测试**（early depth test）[55]。

10.2.5.9 像素着色器

像素着色是完全可编程的阶段。其工作是替每个像素着色（即光照及其他处理）。像素着色器也能丢弃一些片段，例如，某些片段被判断为完全透明的。像素着色器可以对多个纹理采样、计算每像素光照，以及任何会影响片段颜色的计算。

此阶段的输入是一组每片段属性（这些属性是在三角形遍历中通过对顶点属性插值所得）。而输出则是一个颜色矢量，代表所要的片段颜色。

10.2.5.10 合并/光栅运算阶段

管道的最终阶段为**合并阶段**（merge stage）或**混合阶段**（blending stage），NVIDIA称之为**光栅运算阶段**（raster operations stage，ROP）。此阶段不能编程，但能高度配置。此阶段负责执行多个片段测试，包括深度测试（见10.1.4.8节）、alpha测试（片段的alpha通道值能用于丢弃某些片段），以及模板测试（stencil test，见10.3.3.1节）。

若片段通过了所有测试，其颜色就会与帧缓冲原来的颜色进行混合（合并）。混合方式是由**alpha混合函数**（alpha blending function）控制，此函数的结构是固定的，但可以通过配置其运算符及参数产生各种各样的混合运算。

Alpha混合最常用于渲染半透明几何物体。进行这种渲染时，会采用以下的混合函数：

$$\mathbf{C}'_D = A_S\mathbf{C}_S + (1 - A_S)\mathbf{C}_D$$

下标S和D分别指“来源地（source，即传入的片段）”及“目的地（destination，即帧缓冲的像素）”。因此，写进帧缓冲的颜色（$\mathbf{C}'_D$），其实就是目前帧缓冲内容（$\mathbf{C}_D$）及片段颜色（$\mathbf{C}_S$）的**加权平均**。而混合权重（A_S）则是传入片段的来源alpha。

要令alpha混合显示正常，必须先渲染场景中的不透明几何物体至帧缓冲，**然后**把半透明表面从后往前排序渲染。要这样做的原因是，进行alpha混合之后，新片段的深度会**覆写**原来被混合的像素深度。换句话说，深度缓冲会忽略透明度（当然，除非关掉深度写入）。如果要在不透明背景上渲染一堆半透明物体，那么理想地最终像素颜色要与那堆半透明物

[55]译注：通常显卡会自动开启提前深度测试，但若用户开启了alpha测试，在像素着色器改变深度或自我丢弃（discard），就会自动关闭提前深度测试。传统的深度测试是最简单通用的方案，而提前深度测试是个别硬件商后来才加进来的优化性功能。

体的**所有**表面混合。若使用任何从后至前以外的次序渲染，有些半透明片段的深度测试便会失败，导致那些片段被丢弃，最终造成不完整的混合（并且是比较奇怪的影像）[56]。

除了半透明混合，也可以定义其他的混合函数。通用混合函数的形式为$\mathbf{C}'_D = (\mathbf{w}_S \bigotimes \mathbf{C}_S) + (\mathbf{w}_D \bigotimes \mathbf{C}_D)$，当中权重因子$\mathbf{w}_S$及$\mathbf{w}_D$可由程序员设置，其可选值包括0、来源地或目的地颜色、来源地或目的地alpha，以及1减去来源地或目的地的颜色或alpha。运算符$\bigotimes$根据$\mathbf{w}_S$及$\mathbf{w}_D$的数据类型，可以是普通的标量对矢量乘法，或是矢量对矢量的分量乘法（即阿达马积/Hadamard product，见4.2.4.1节）。

10.2.6 可编程着色器

现在我们对GPU管道从头到尾有了个基本概念，让我们深入探讨当中最有趣的部分——可编程着色器。着色器自DirectX 8引入以来，其架构有重大的演进。早期的**着色器模型**（shader model）只支持底层的汇编语言编程，而且像素着色器的指令集和寄存器集和顶点着色器有很大区别。DirectX 9时代带来了高级的、近似C的着色语言，例如Cg（C for graphics）、HLSL（高级着色语言/high-level shading language——微软的Cg语言实现）及GLSL（OpenGL着色语言/OpenGL shading language）。DirectX 10引进了几何着色器，并带来了统一着色器架构，其DirectX术语为**着色器模型4.0**。在此统一着色器模型中，3种着色器支持差不多相同的指令集，以及差不多相同的能力，这些能力包括读取纹理内存。

着色器从输入数据取得一个元素，把该元素变换为输出数据的一个或多个元素。

- 顶点着色器的输入为顶点，包含以模型空间或世界空间表示的位置及法矢量。而输出为已变换及照明的顶点，以齐次裁剪空间表示。
- 几何着色器的输入为单个n顶点几何图元——点（$n = 1$）、线段（$n = 2$）或三角形（$n = 3$），以及最多n个作为控制点的额外顶点。其输出为0或多个图元，这些图元的种类可与输入的有所不同。例如，几何图元可以把点转换为由两个三角形组成的四边形；也可以把三角形变换为三角形，但当中部分三角形可以被丢弃。
- 像素着色器的输入为片段，其属性来自对三角形顶点属性的插值。其输出是将要写至帧缓冲的颜色（假设片段能通过深度测试及其他可选测试）。像素着色器也能明确地丢弃片段，那些情形下便无任何输出。

---

[56]译注：前文也提及，画家算法无法完美地把物体从后往前排序，所以才会使用深度缓冲解决物体遮挡问题。这里使用画家算法渲染多个半透明物体，也会产生相似的问题。而且，如果没有以三角形为单位排序，网格内的三角形也不会以从后往前的次序渲染，形成文中所说的不完整混合。因此，一般渲染半透明物体的做法是，以物体为单位从后往前排序，然后关掉深度写入（depth write）去渲染。那么每个半透明片段只会与不透明物体进行深度测试，能尽量避免出现闪烁。更理想的解决方法是次序无关透明（order-independent transparency，OIT）的渲染技术，例如深度剥离（depth peeling）、alpha至覆盖掩码转换（alpha to coverage）、片段链表（fragment linked list）等。

10.2.6.1 内存访问

由于GPU实现了数据处理管道，必须非常谨慎地控制内存访问。着色器程序不能直接读 / 写内存。取而代之，其内存访问只限于两个方法：寄存器和纹理贴图。

着色器寄存器

着色器可用**寄存器**间接地存取内存。所有GPU寄存器是128位SIMD格式。每个寄存器能保存4个32位浮点数（在Cg语言里由`float4`数据类型表示）或4个32位整数。这些寄存器能包含一个齐次坐标的四维矢量，或是一个RGBA格式的颜色，当中每个分量为32位浮点数格式。矩阵可以由一组3或4个寄存器表示（在Cg里以`float4x4`等内置矩阵类型表示）。GPU寄存器也可以用来保存单个32位标量，这样用的时候，通常会把该值复制至所有4个32位字段。有些GPU能在16位字段上运算，这种数据类型称为**half**。（Cg为此提供多种内置类型，如`half4`及`half4x4`等。）

寄存器有以下4大类。

- **输入寄存器**（input register）：这些寄存器是着色器的主要数据输入来源。在顶点着色器中，输入寄存器含有顶点的属性数据。在像素着色器中，输入寄存器含有对应某片段的顶点属性插值数据。在调用着色器之前，GPU会自动设置这些输入寄存器的值。
- **常数寄存器**（constant register）：常数寄存器的值是由应用程序设置的，应用程序按不同图元会设置不同的值。所谓常数，只是对着色器而言。常数寄存器是着色器的另一种输入。其典型内容包括模型观察矩阵、投影矩阵、光照参数，以及其他着色器所需但顶点属性不包含的数据。
- **临时寄存器**（temporary register）：这些寄存器只供着色器程序内部使用，通常用于储存中间计算结果。
- **输出寄存器**（output register）：这些寄存器的内容由着色器填充，作为着色器仅有的输出形式。在顶点着色器中，输出寄存器含有顶点属性，例如，以齐次裁剪空间表示的已变换的位置及法矢量、可选的颜色、纹理坐标等。在像素着色器中，输出寄存器包含正在着色的片段的最终颜色。

当提交几何图元渲染时，应用程序要提供常数寄存器的值。在调用着色器程序之前，GPU会从显存自动复制顶点或片段属性数据至适当的输入寄存器；当程序执行完成，GPU会把输出寄存器的内存写入显存，使数据能给予下个管道阶段。

GPU通常会把输出数据储存至缓存，使这些数据能重用而不需要重新计算。例如，**变换后顶点缓存**（post-transform vertex cache）是用来储存顶点着色器最近处理过的结果。

如果某三角形刚好引用到之前已处理过的顶点，那么可行的话GPU就会读取变换后顶点缓存——只有当缓存中的顶点被丢弃后，才需要再对相同的顶点执行顶点着色器。

纹理

着色器也能够直接读取**纹理贴图**。纹理数据是以纹理坐标寻址的，而不是使用绝对内存地址。GPU的纹理采样器会自动**过滤**纹理数据，适当地混合相邻纹素及相邻渐远纹理级数的值。也可以关上纹理过滤，以直接存取某纹素的值。当纹理贴图作为查找表之用，关上过滤功能就很有用。

着色器只能用间接方法**写**数据进纹理——把场景渲染至屏幕外帧缓冲，再在后续的渲染阶段把该帧缓冲当作纹理贴图使用。此功能称为**渲染到纹理**（render to texture，RTT）。

10.2.6.2 高级着色器语言的语法入门

高级着色器语言如Cg和GLSL仿照了C语言来制定。程序员能声明函数、定义简单的`struct`，以及做算术运算。然而，如前所述，着色器程序只能存取寄存器和纹理。因此，在Cg或GLSL所定义的`struct`及变量都会由着色器编译器把它们直接映射至寄存器。我们会用以下方式定义这些映射。

- **语义**（semantic）：我们可以在变量或`struct`成员之后加入冒号和一个名为**语义**的关键词。语义告诉编译器如何把变量或数据成员绑定至个别顶点或片段属性。例如，在顶点着色器中我们可以声明一个输入`struct`，其成员映射至顶点的**位置**及**颜色**属性：

```
struct VtxOut
{
    float4 pos   : POSITION;    // 映射至位置属性
    float4 color : COLOR;       // 映射至颜色属性
};
```

- **输入和输出**：编译器会根据使用个别变量或`struct`的上下文，判断它们应映射至输入或输出寄存器。若变量是以参数形式传入着色器的主函数，那么它就会被当作输入；若变量是主函数的传回值，那么它就会被当作输出。

```
VtxOut vshaderMain(VtxIn in)      // in 映射至输入寄存器
{
    VtxOut out;
    // ……
    return out;                   // out 映射至输出寄存器
}
```

- **uniform声明**：要从应用程序经常数寄存器取得数据，可以在声明变量时加入uniform关键字。例如，模型观察矩阵可用以下方式传入顶点着色器：

```
VtxOut vshaderMain(VtxIn in,
                   uniform float4x4 modelViewMatrix)
{
    VtxOut out;
    // ……
    return out;
}
```

进行算术运算，可使用C风格的运算符，或调用适当的内部函数（intrinsic）。例如，要把输入顶点位置乘以模型观察矩阵，可写成：

```
VtxOut vshaderMain(VtxIn in,
                   uniform float4x4 modelViewMatrix)
{
    VtxOut out;
    out.pos = mul(modelViewMatrix, in.pos);
    out.color = float4(0, 1, 0, 1); // RGBA 绿色
    return out;
}
```

要从纹理获取数据，需要调用特殊的内部函数，这些函数会从指定的纹理坐标读取纹素的值。这些函数有多种变种，以供读取不同格式的一维、二维或三维纹理，并可选择是否使用过滤。也会提供特殊的寻址模式存取立方体贴图及阴影贴图。引用纹理需要使用特别的数据类型声明方式，此方式称为**纹理采样器**（texture sampler）声明。例如，sampler2D数据类型代表对二维纹理的引用。以下是一个简单的Cg像素着色器，把漫反射贴图贴于三角形之上：

```
struct FragmentOut
{
    float4 color : COLOR;
}

FragmentOut pshaderMain(float2 uv : TEXCOORD0,
                        uniform sampler2D texture)
{
    FragmentOut out;
    out.color = tex2D(texture, uv); // 查找位于(u,v)的纹素
    return out;
}
```

10.2.6.3 效果文件

一个孤立的着色程序并不十分有用。GPU管道还需要一些额外信息，才能为着色器程序调用提供有意义的输入。例如，我们需要指定应用程序相关的参数（如模型观测矩阵、光源参数等）如何映射至着色器程序中声明的uniform变量。此外，有些视觉效果需要两个或以上的渲染步骤（render pass），但一个着色器只能描述单个渲染步骤内的运算。另外，若在PC平台上开发游戏，我们需要为高端的渲染效果定义一个“撤退（fallback）”版本，以供旧显卡正常运作。为了把（多个）着色器结合成完整的视觉效果，我们可以使用名为**效果文件**（effect file）的文件格式。

不同的渲染引擎会用稍有差异的方式实现这些效果文件。在Cg里，其效果文件格式名为**CgFX**。OGRE采用和CgFX相似的格式，名为**材质文件**（material file）。GLSL可以使用COLLADA文件描述效果，此格式是基于XML的。虽然有这些差异，但是效果文件通常会使用到以下的层次结构。

- 在全局作用域定义struct、着色器程序（实现为多个“主”函数）和全局变量（映射至应用程序相关的常数参数）。
- 定义一个或多个**技术**（technique）。每个技术代表渲染某视觉效果的方法。一个效果通常提供一个主要技术，作为效果的最高品质实现，另外再加上多个回退（fall back）技术，供较低级的图形硬件使用。
- 每个技术内定义一个或多个**步骤**（pass）。每个步骤描述如何渲染一整帧影像。通常一个步骤包含顶点/几何/像素着色器程序的“主函数”引用、多个参数绑定及可选的渲染状态设置。

10.2.6.4 延伸阅读

本节仅让我们浅尝高级着色器编程的滋味——完整教程超出本书范围。关于Cg着色器编程的详细介绍，可参阅NVIDIA网站提供的Cg教程[57]。

10.2.7 应用程序阶段

现在我们已理解GPU如何运作，接下来我们会讨论负责驱动GPU的管道阶段——应用程序阶段（application stage）。本阶段有3个角色。

1. **可见性判别**：应该仅把可见（或至少潜在可见）的物体提交GPU，以免浪费宝贵的资

[57] http://developer.nvidia.com/object/cg_tutorial_home.html

源渲染总是看不见的物体。

2. **提交几何图元至GPU以供渲染：**使用DirectX的`DrawIndexedPrimitive()`或OpenGL的`glDrawArrays()`之类的渲染调用把子网格材质对传送至GPU。另一个提交方法是建立GPU命令表。几何图元要适当地排序以优化渲染性能。若场景需要用多个步骤渲染，几何图元便需要被提交多次。
3. **控制着色器参数及渲染状态：**uniform参数通过常数寄存器传送至着色器时，应用程序阶段需按每个图元为单位进行配置。此外，应用程序阶段必须设置所有不可编程但可配置的管道阶段参数，以确保每个图元能正确地渲染。

在以下几个小节中，我们会概要地探索应用程序阶段如何执行这些任务。

10.2.7.1 可见性判断

最不费时的三角形就是那些不用渲染的。因此，在把场景中的物体提交至GPU之前，**剔除**（cull）不会对最终影像有任何贡献的物体，是极其重要的事。建立可见网格实例表的过程称为**可见性判断**（visibility determination）。

平截头体剔除

在平截头体剔除（frustum culling）里，完全位于平截头体之外的物体便会排除在渲染表之外。给定一个候选网格实例，我们可以通过一些简单测试判别它是否在平截头体之内，这些测试会使用到物体的**包围体积**（bounding volume）及6个平截头体平面。包围体积通常是球体，因为球体特别容易进行剔除运算。对每个平截头体平面，我们把该平面往内移动球体半径的距离，然后就可以判别球的中心点是位于修改后平面的哪一方。若球体在所有6个修改后平面的前方，球体就是在平截头体之内[58]。

10.2.7.4节描述的场景图数据结构可以优化平截头体剔除，把包围体积不接近平截头体的物体完全忽略。

遮挡及潜在可见集

就算物体完全在平截头体之内，也可能会被其他物体遮挡。把可见表中完全被其他物体遮挡的物体移除，称为**遮挡剔除**（occlusion culling）。在拥挤的环境中，从地表上观察，会

[58]译注：更正确地说，这个测试只能判别部分完全在平截头体之外的球体。有些情况球体位于平截头体之外，但还是会误判。然而这种方法是保守的（conservative），不会错误剔除一些潜在可见的物体。详情可参阅《Real-Time Rendering》第3版16.14.2节。

出现许多物体互相遮挡的情形，那么遮挡剔除就非常重要。而在较不拥挤的环境中，或俯览场景，就只会有较少的遮挡出现，那么遮挡剔除的成本便可能超过其得益。

大型环境的总体遮挡剔除可以通过预计算**潜在可见集**（potentially visible set，PVS）实现。给定一个摄像机位置，PVS能列出可能可见的物体。PVS会不准确地包含一些实际上不可见的物体，但不会错误地排除一些应该对渲染有贡献的物体。

实现PVS系统的方法之一，就是把场景切割成某类型的区域。每个区域提供摄像机在该区域内能看见的其他区域列表。这些PVS可以由美术人员或游戏设计师手工设置。更常见的方法是采用自动的脱机工具，在人工设置的区域上生成PVS。这些工具的运作原理，通常是随机挑选区域内的不同视点来渲染场景。只要把每个区域的几何物体用颜色编码，便可以通过扫描渲染结果的帧缓冲来得出可见区域表。但由于自动化PVS工具并非完美，所以它们会向用户提供一些机制调整结果，例如手工设置摄像机位置，或是手工指定一些区域应该要包含或排除于某个PVS的区域内。

入口

另一个判别场景中哪些部分可见的方法是使用**入口**（portal）。使用入口渲染时，游戏世界会划分为半封闭的区域，这些区域以孔洞互相连接，例如窗户或门户。这些孔洞称为入口，通常会以其边界的多边形表示。

要渲染一个含入口的场景时，首先渲染包含摄像机的区域。然后，对于每个连接着该区域的入口，我们建立对应的、像平截头体的体积。该体积含有多个平面，每个平面都是延伸自摄像机焦点及入口的包围多边形的棱。相邻区域的物体就利用该入口体积来进行剔除，方法和平截头体剔除一模一样。那么便可以确定相邻区域中只有可见物体才被渲染。图10.44说明此技术。

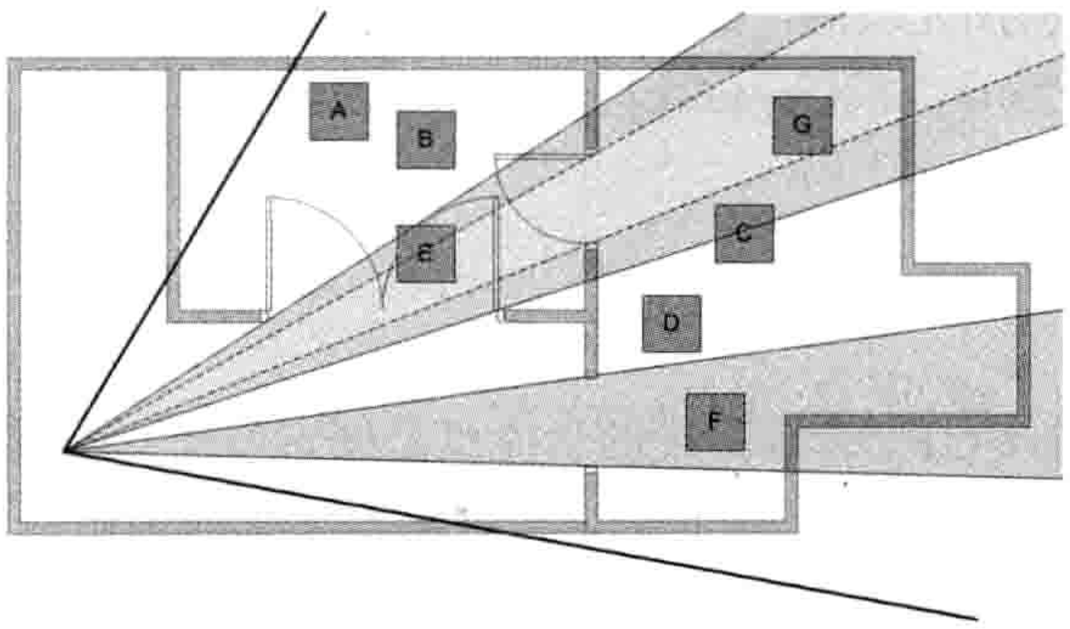


图 10.44: 入口是用于定义像平截头体一样的体积的，这些体积用来剔除相邻的区域。在此例子中，物体A、B及D会被剔除，因为它们在入口体积以外。其他物体则是可见的。

遮挡体积（反入口）

若我们把入口的概念反转，锥形的体积也可用于描述某物体遮挡的区域，这些区域内的物体**不会**被看见。这种体积称为**遮挡体积**（occlusion volume）或**反入口**（anti-portal）。构建遮挡体积时，我们找出遮挡物的每个轮廓边缘（silhouette edge），并把平面自摄像机焦点延伸至这些棱。当测试较远的物体是否被遮挡时，若它们完全在这些遮挡体积之内，则可以把它们剔除。见图10.45。

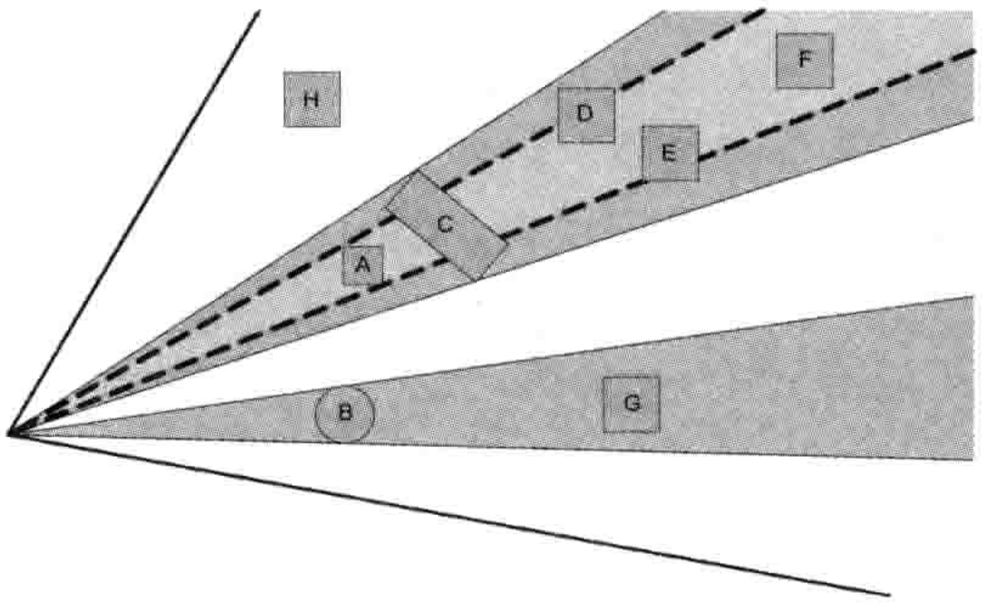


图 10.45: 基于对应物体A、B和C的反入口，物体D、E、F和G被剔除。因此，只有A、B、C和H是可见的。

入口最适用于渲染封闭的室内环境，当中“房间”之间只有少量的门窗。在这类场景中，入口体积只占摄像机平截头体体积很小的百分比，以致可以剔除大量在入口体积以外的物体。在此情况下，反入口占平截头体体积很大的百分比，所以也能剔除大量的物体。

10.2.7.2 提交图元

当产生了可见的几何图元表后，必须**提交**至GPU管道进行渲染。方法是调用DirectX的`DrawIndexedPrimitive()`或OpenGL的`glDrawArrays()`。

渲染状态

如10.2.5节提及，许多GPU管道阶段的功能虽是固定但却可配置的。即使可编程的阶段也有些部分由可配置参数所驱动。以下是这些可配置参数的例子（虽然这决不是完整的列表）。

- 世界观察矩阵。
- 光源方向矢量。

- 纹理绑定（即供某材质/着色器所用到的纹理）。
- 纹理寻址及过滤模式。
- 基于时间的纹理滚动及其他动画效果。
- 深度测试（启用或禁用）。
- alpha混合选项。

GPU管道内的所有可配置参数称为**硬件状态**（hardware state）或**渲染状态**（render state）。应用程序阶段有责任确保提交每个图元时，正确地及完整地配置硬件状态。理想地，这些状态设置完全由每个子网格对应的材质所描述。那么应用程序阶段的工作可归结为：遍历可见的网格实例列表，遍历每个子网格材质对，以材质规格设置渲染状态，并调用底层的提交几何图元函数（`DrawIndexedPrimitive()`、`glDrawArrays()`或其他类似函数）。

状态泄漏

如果在提交图元之间，我们忘记设置某方面的渲染状态，那么上一图元的设置便会“泄漏”至下一图元。**渲染状态泄漏**（render state leak）的结果，可能会是物体出现错配的纹理或不正确的光照效果。显然应用程序阶段绝不应产生状态泄漏。

GPU命令表

应用程序阶段实际上使用命令表（command list）和GPU进行沟通。这些命令包含交错的渲染状态设置及渲染几何图元的引用。例如，要用材质1去渲染物体A和B，然后用材质2去渲染物体C、D和E，其命令表大概是这样的：

- 设置材质1的渲染状态（含多个命令，每个命令设置一个状态）。
- 提交图元A。
- 提交图元B。
- 设置材质2的渲染状态（含多个命令）。
- 提交图元C。
- 提交图元D。
- 提交图元E。

在底层里，如`DrawIndexedPrimitive()`之类API函数实际上仅仅是构建及提交GPU命令表。这些API的调用成本本身对于某些应用程序来说可能太高。为了优化性能，有些游戏

引擎会手工建立GPU命令表[59]，或调用底层的渲染API，如PS3的libgcm库。

10.2.7.3 几何排序

渲染状态设置是全局的——它们在整个GPU中有效。因此，改变渲染状态时，整个GPU管道必须完成目前工作，才能换上新的设置。若不妥善管理，会令效能严重下降。

显然我们希望渲染状态的改变次数越少越好。最好的解决方法是按材质来排序几何物体。按此方法，我们先设置材质A的渲染状态，然后渲染所有采用材质A的几何物体，再轮到材质B。

可惜的是，把几何物体按材质排序会对渲染性能产生不利影响，因为它会增加**覆绘**（overdraw）——多个互相重叠的三角形重复填充同一像素。无疑有些像素覆绘是必须且我们期望的，因为这是唯一正确的方法把半透明表面alpha混合至场景中。然而，**不透明**像素的覆绘总是浪费GPU时间。

提前深度测试的设计是为了预先丢弃将被遮挡的像素，以避免执行耗时的像素着色器。但要充分利用其优势，我们需要按从前至后的顺序渲染三角形。那么，最近摄像机的三角形才会立即填充深度缓冲，而后续来自较远三角形的所有片段才能迅速被丢弃，最终导致很少甚至全无覆绘。

深度预渲染步骤是救星

又要按材质排序渲染几何物体，又要按从前至后的顺序渲染不透明几何物体，我们怎样才能排解此冲突？答案是使用GPU的**深度预渲染步骤**（z prepass）功能。

深度预渲染步骤的基本概念是渲染场景两次：第1次尽速产生深度缓冲的内容，第2次才用完整的颜色填进帧缓冲（受惠于深度缓冲的内容，此次不会有覆绘）。当关闭像素着色器并仅更新深度缓冲，GPU便会使用特设的双倍速度的渲染模式。在此次渲染步骤中，不透明物体可按从前至后的顺序渲染，使深度缓冲的写入次数变得最少。然后几何物体按材质来重新排序，用最少的状态改变渲染颜色，使管道吞吐量最大化。

当渲染不透明几何物体之后，就可以用从后至前的顺序渲染半透明表面。可惜的是，半透明几何物体的材质排序问题并无通用解决方案。我们必须用从后至前的顺序去渲染，才能得到正确的alpha混合结果。因此渲染半透明几何物体时，我们必须接受频繁切换状态的成本（除非某游戏使用的半透明几何物体能实现特殊优化[60]）。

---

[59]译注：Direct3D 11容许用户建立命令表，目的是让多个线程同时生成命令表，以提高在多核CPU上的性能。

[60]译注：例如，可采用纹理图谱（texture atlas）技术把多个材质合成为一个，那么便能减少切换渲染状态。

10.2.7.4 场景图

现在的游戏世界能达到非常大的规模。由于多数场景中，大部分的几何物体会在摄像机平截头体的范围之外，因此，明确地用平截头体剔除每个这些物体，通常难以置信地耗费时间资源。取而代之，我们希望能设计一些数据结构管理场景中所有几何物体，并能迅速丢弃大量完全不接近摄像机平截头体的世界部分，这样才能进行更仔细的平截头体剔除。理想地，此数据结构更可以帮助对场景中的几何物体排序；排序次序方面，可以为了深度预渲染步骤做前至后或后至前排序，或是为了颜色渲染而采用材质排序。

这样的数据结构通常称作**场景图**（scene graph），此名称与电影渲染引擎或Maya之类的DCC工具所采用的数据结构相关。然而，游戏的场戏图不必是图，而实际上其数据结构通常会选择某种树。多数这类数据结构的理念在于，把三维空间以某形式划分为区域，使不与平截头体相交的区域能尽快丢弃，而无须逐一物体进行平截头体剔除。这些数据结构的例子有四叉树（quadtree）、八叉树（octree）、BSP树、kd树、空间散列（spatial hashing）技术等。

四叉树和八叉树

四叉树以递归方式把空间分割成象限（quadrant）。每层递归以四叉树的节点表示，每个节点有4个子节点，每个子节点代表一个象限。这些象限通常是由轴对齐的平面切割而成的，所以每个象限是正方形或长方形的。然而，有些四叉树用任意形状的区域来细分空间。

四叉树可用于储存及组织几乎任何在空间中分布的数据。在渲染引擎中，四叉树通常用来储存可渲染图元，例如网格实例、地形几何的子区域、大型静态网格的个别三角形等，目的是加速平截头体剔除。可渲染图元储存于树的叶节点，我们通常尽量令每个叶节点有均匀的图元数目。要实现此目标，可以基于区域内的图元数目来决定继续或终止细分区域。

要判断哪些区域在摄像机平截头体中可见，我们从根节点往叶节点遍历，检查每个中间区域是否位于平截头体内。若某个象限不与平截头体相交，那么其子区域也不会相交，所以我们可以停止遍历该分支。这样，我们搜寻潜在可见图元的速度会比线性搜寻快得多（通常是$O(\log n)$时间）。图10.46展示了一个四叉树细分的例子。

八叉树是四叉树的三维版本。在每层递归细分时，八叉树把空间分割为8个子区域[61]。八叉树的子区域通常是正方体或长方体，但也可以是任意的三维区域。

[61] 译注：正式术语为挂限（octant）。

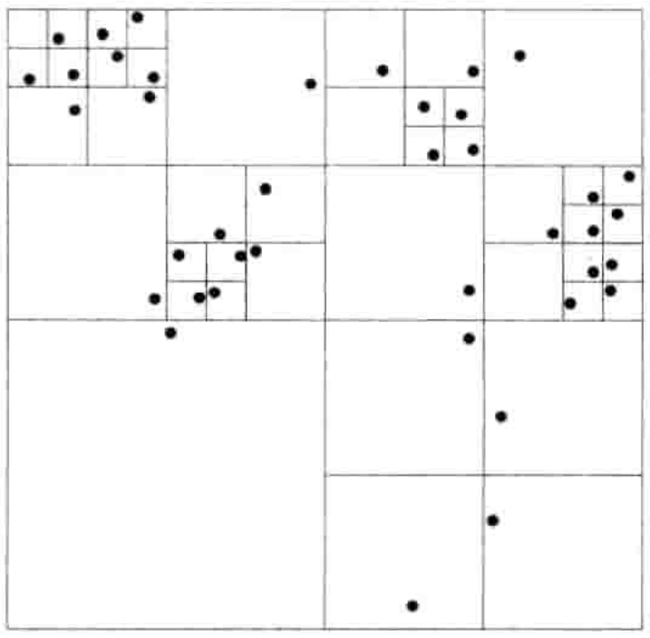


图 10.46: 从此俯瞰图显示，一个空间被递归分割成象限，以储存为四叉树。空间一直被分割直至每个区域只含一点。

包围球树

如同四叉树和八叉树把空间细分为（通常）矩形区域，**包围球树**（bounding sphere tree）把空间以层次结构分割为球状区域。包围球树的子节点含有场景中可渲染图元的包围球。我们首先把图元分成小组，计算每组的包围球。然后这些小组又再结合成较大的组，重复此过程，直至得到一个包围整个虚拟场景的包围球。求潜在可见图元时，也是从根节点往子节点遍历，测试每个包围球是否和平截头体相交，仅对相交分支继续递归遍历。

BSP树

二元空间分割（binary space partitioning，BSP）树把空间递归分割为一半，直接每个半空间（half-space）里的物体符合某些预定条件（有如四叉树把空间分割成象限）。BSP树有多种用途，包括碰撞检测和构造实体几何（constructive solid geometry，CSG），以及其最知名的优化三维图形用途——平截头体剔除及几何物体排序。kd树是BSP树的特殊情况，BSP的分割平面可以是任意方向的，而kd树的分割平面会依次序与k维空间的轴对齐（例如在某2-d树中，先以某x坐标把空间分割为两个“半空间”，再用某y坐标把其中一个“半空间”分割，以此类推）[62]。

以渲染应用来说，BSP树在每层递归中用单个平面把空间二分。这些分割平面可以是轴对齐的，但更常用的方法是每次细分时，都以场景中某三角形的平面分割空间。所有其他三角形就会分成两类，一类在该分割平面之前，另一类在之后。任何与分割平面相交的三角形，都会被切割为3个三角形，使每个三角形都只会在分割平面之前或之后，或是和分割平面共面。

[62]译注：此句原文意思为kd树是把BSP树概念推广至k维，并不正确，故简单改写之。

BSP树可用于平截头体剔除，其实现方法基本上和四叉树、八叉树、包围球树无大区别。然而，若使用上述以个别三角形生成BSP的方法，这种BSP树也可以使三角形按从后至前或从前至后的严格次序排序[63]。此排序功能对早期的三维游戏如《毁灭战士》特别重要，因为那个时候还没有硬件深度缓冲，而被逼使用画家算法（即把场景从后至前渲染）去保持三角形间之遮挡关系正确。

要从后往前排序BSP树，首先需给定一个三维空间中的摄像机视点，然后由根节点开始遍历。在每个节点上，我们检测视点是位于该节点的分割平面之前或之后。若摄像机位于节点平面之前，那么我们先遍历往后的子节点，然后渲染和该节点平面共面的三角形，最后再遍历往前的子节点。同样地，若摄像机视点位于节点的分割平面之后，则先遍历往前的子节点，渲染共面的三角形，再遍历往后的子节点。此遍历方式保证先遍历离摄像机最远的三角形，才会遍历较近的三角形，因而能产生从后往前的次序。由于此算法会遍历场景中所有三角形，遍历次序与摄像机观察方向并无关系。为了筛选可见的三角形，还需再进行平截头体剔除步骤。图10.47是一个简单BSP树的图例，旁边列出按该摄像机位置遍历的步骤。

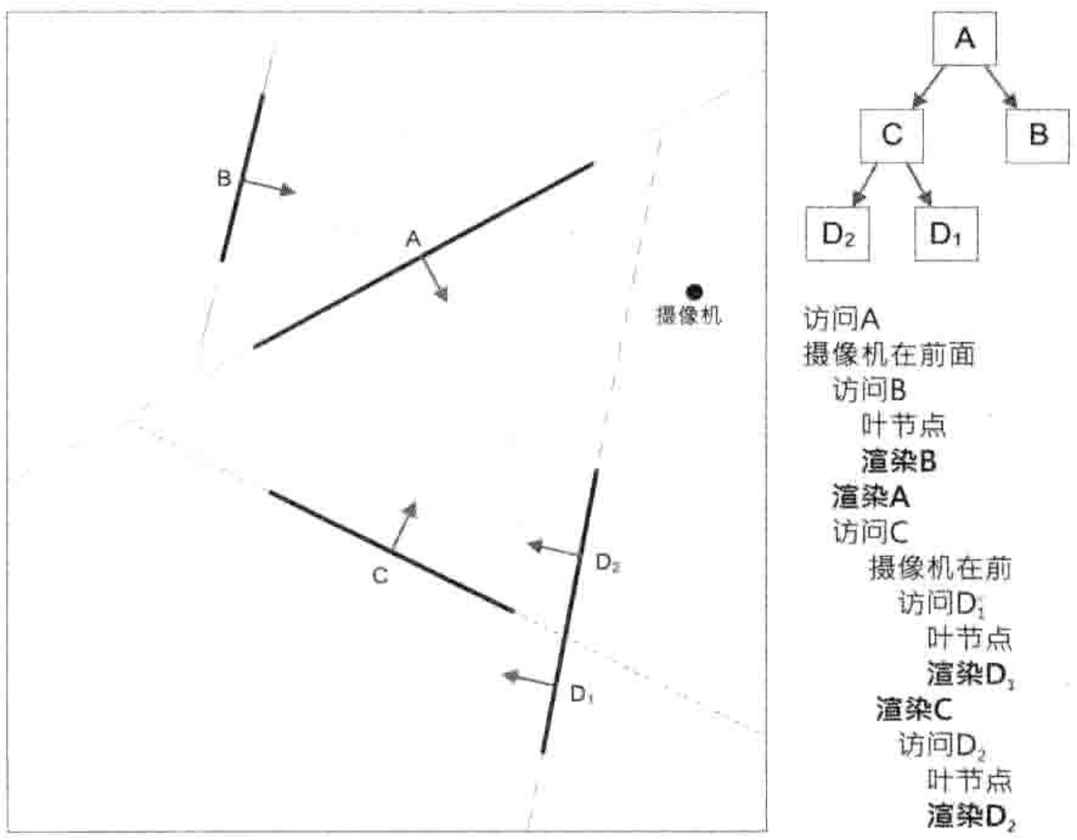


图 10.47: 这是BSP从后往前遍历三角形的例子。为简单起见，三角形显示为二维的线段，但在真正的BSP树中，三角形及分割平面可以在空间中任意定向。

生成及使用BSP树的完整描述超出本书范围，详情可参考这两个网站[64,65]。

[63]译注：四叉树、八叉树、kd树也可以用类似的方法进行排序，但由于使用这些数据结构时并不会如BSP树般切割和平面相交的物体（如包围体积），所以通常其排序次序并不如BSP般严格。

[64]http://www.ccs.neu.edu/home/donghui/teaching/slides/geometry/BSP2D.ppt

[65]http://www.gamedev.net/reference/articles/article657.asp

10.2.7.5 选择场景图

世上有许多不同种类的场景图。哪一款适合你的游戏，视乎游戏要渲染的场景特质而定。为了做出明智的抉择，必须清楚了解特定游戏的渲染场景需求——更重要是哪些不是需求。

例如，若读者正在开发格斗游戏，游戏中两个角色在擂台上战斗，擂台外主要是静态的场景，那么这游戏可能根本不需要场景图。若游戏主要在室内环境进行，BSP树或入口系统可能是不错的选择。若游戏于比较平坦的室外地形上进行，并且场景以俯览为主（或许如实时策略游戏或上帝模拟游戏的情况），简单的四叉树也许已能达到高效的渲染性能。另一方面，若室外场景主要是以地上角色的视点观察，就可能需要额外的剔除机制。内容密集场景可受益于遮挡体积（反入口）系统，因为场景中会有许多遮挡物。若你的室外场景的物体较为分散，加入反入口系统或无裨益（甚至有损帧率）。

最后，应该以统计数据选择场景图，统计数据来自你的渲染引擎实际的性能量度。读者可能会对实际的性能瓶颈位置感到惊讶。但当得知事实后，就能针对手头上的问题，选择合适的场景图及其他优化方法。

10.3 高级光照及全局光照

为了渲染逼真的场景，我们需要物理上精确的全局光照算法，完整地介绍这些算法会超出本书范围。以下几节中，我们会概要简介今天游戏产业中最流行的相关技术。本章之目的在于让读者知道有相关的技术，并以此为继续学习的起点。此题目的深入探讨可参考[8]。

10.3.1 基于图像的光照

许多高级的光照及着色技术都会大量使用影像数据，这些数据通常是以二维纹理贴图形式表示的。这些技术统称为**基于图像的光照**（image-based lighting）算法。

10.3.1.1 法线贴图

法线贴图（normal map）中，每个纹素代表表面法矢量的方向。利用这种贴图，三维建模师可以为渲染引擎细致地描述表面的形状，而无须把模型高度镶嵌（那么可以把相同的数据存于顶点法线）。使用法线贴图的单个平面三角形，看上去有如百万个细小三角形做成的效果。图10.48是一个法线贴图例子。

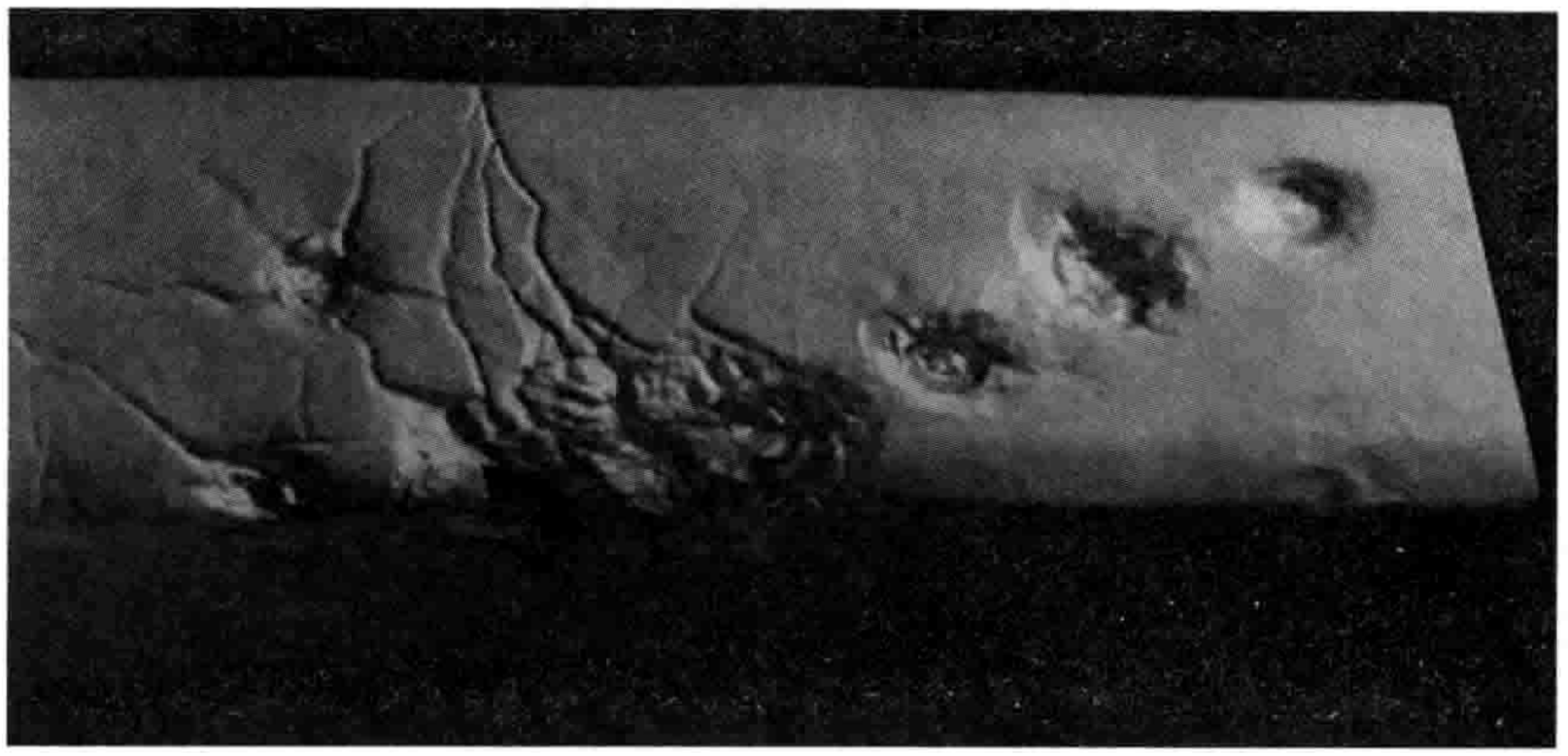


图 10.48: 在表面应用法线贴图的例子。

法矢量通常会在纹理的RGB颜色通道中编码。由于RGB颜色通道必须为正数，而法矢量的分量可为负数，所以为法矢量编码时会加上合适的偏置（bias）[66]。假设表面法矢量都是单位矢量，那么有时候只需在纹理中储存两个坐标，第3个坐标能较简易地在运行时计算得出[67]。

10.3.1.2 高度贴图：视差贴图和浮雕贴图

高度贴图（height map）一如字面之意思，是用来编码高于或低于三角形表面的理想高度。因为每个纹理只需单个高度值，所以高度贴图通常编码为灰阶影像。

高度贴图通常用于**视差贴图法**（parallax mapping）[68]及**浮雕贴图**（relief mapping）[69]技术。这两个技术都能令平面表面显得有强烈的高度变化，能制造出自遮挡（self-occlusion）及自阴影（self-shadow）的效果。图10.49展示了一个以DirectX 9实现的视差遮挡贴图法（parallax occlusion mapping，POM）[70]例子。

---

[66]译注：最常见的法线贴图是储存切线空间（tangent space）的法矢量。使用这种贴图时还需要一些顶点数据的配合，建议读者可参考《Cg教程（The Cg Tutorial）》第8章，其英文版本可在网上免费阅读http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter08.html。

[67]译注：法线贴图的编码方法在近年有不少进展，Crytek在2010 SIGGRAPH课程课件《CryENGINE 3: Reaching the Speed of Light》中介绍了他们的成果。可于http://advances.realtimerendering.com/s2010/下载。

[68]译注：可参考https://www8.cs.umu.se/kurser/5DV051/VT09/lab/parallax_mapping.pdf。

[69]译注：可参考http://www.inf.ufrgs.br/~comba/papers/2005/rtrm-i3d05.pdf。

[70]译注：有关POM及其进展版本四叉树位移贴图法（quadtree displacement mapping，QDM）的介绍，可参考《GPU Pro》第3部分第1章，或其GDC课件http://www.drobot.org/pub/M_Drobot_Programming_Quadtree%20Displacement%20Mapping.pdf。

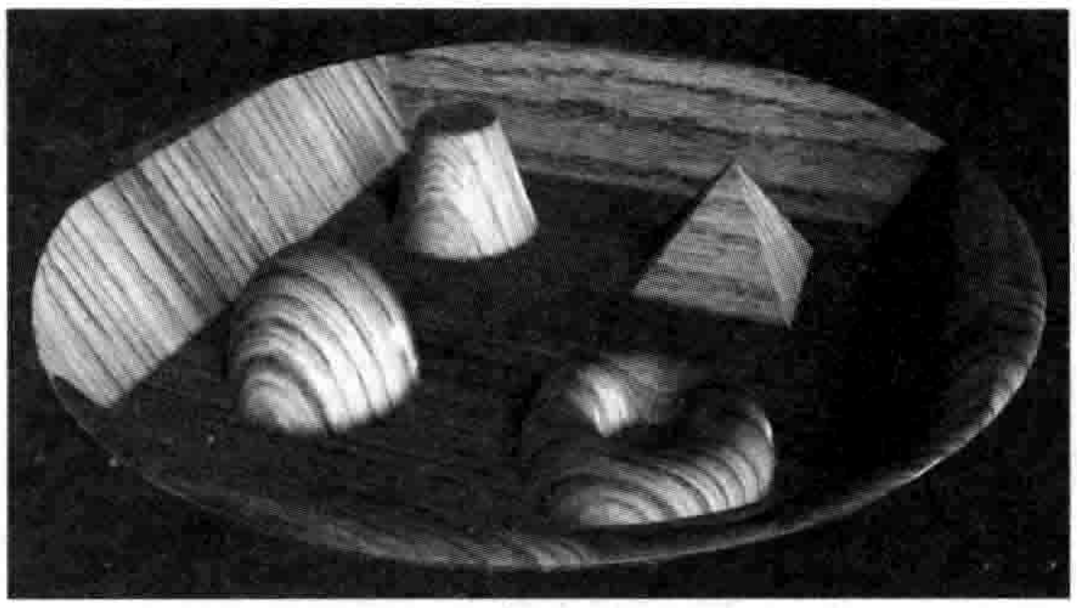


图 10.49: DirectX 9视差遮档贴图。此表面实际上是一个平盘，但使用了高度贴图去定义其表面细节，。

高度贴图也可用于快速生成法矢量。此技术用于早期的凹凸贴图。现在大多数引擎都会把法矢量直接储存于法线贴图，而不会通过高度贴图去计算法矢量。

10.3.1.3 镜面/光泽贴图

当光直接从光滑表面反射时，这称为**镜面**（specular）反射。镜面反射的强度取决于观察者、光源和法矢量之间的相对角度。如10.1.3.2节所提及，镜面强度的数学形式为$k_S(\mathbf{R}\cdot\mathbf{V})^{\alpha}$，当中$\mathbf{R}$为光源方向矢量经表面法矢量反射后的反射方向，$\mathbf{V}$则是往观察者的方向，$k_S$为表面整体镜面反射率，而$\alpha$称为镜面幂（specular power）[71]。

许多镜面并不是均匀光滑的。例如，当人的脸上有汗或污垢时，湿润的区域显得有光泽，而干燥的区域则显得暗哑。我们可以把非常细致的镜面信息编码至一张贴图中，此贴图称为**镜面贴图**（specular map）。

若我们把k_S的值存进镜面贴图的纹素，就能控制每个纹素的位置能造成多少镜面反射。这一种镜面贴图有时会称为**光泽贴图**（gloss map）。这种贴图也会称为**镜面遮罩**（specular mask），因为0值的纹素能“遮盖”不想要有镜面反射的表面部分。若在镜面贴图中储存α的值，那么我们可以控制每纹素位置镜面高光的集中程度。这种纹理称为**镜面幂贴图**（specular power map）。图10.50是一个光泽贴图应用例子。

10.3.1.4 环境贴图

环境贴图（environment map）的样子，有如以场景中某物体的视点拍摄其四周环境的全景照片（panoramic photograph），视域涵括全360°水平方向，以及垂直方向180°或360°。

[71] 译注：前文曾称α为光滑度（glossiness）。

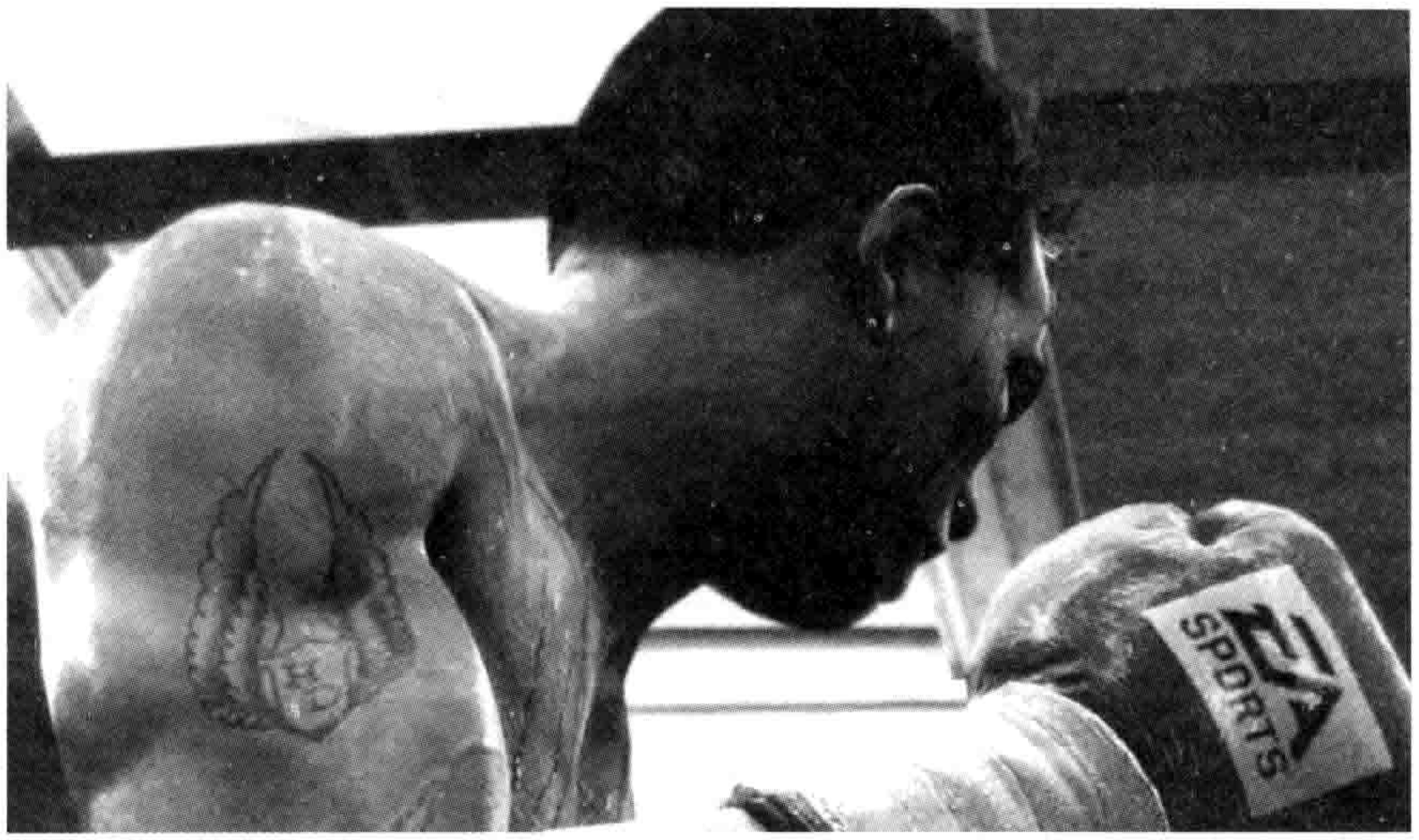


图 10.50: 在艺电《拳击之夜3（Fight Night Round 3）》的截图中，展示了光泽贴图如何在表面的每个纹素控制镜面反射程度。

环境贴图可作为物体四周光照环境的通用描述。环境贴图通常用作低成本反射渲染。

最常见的两种环境贴图格式是**球面环境贴图**（spherical environment map）及**立方环境贴图**（cubic environment map）[72]。球面贴图像是一张由鱼眼镜头拍摄的照片，又有如贴于一个无穷大球体之内，球心为要渲染物体的位置。球面贴图的问题之一在于使用球坐标（spherical coordinates）进行寻址。在赤道附近，水平和垂直方向都有充足的分辨率。然而，当垂直角度（方位角/azimuthal angle）接近垂直时，水平（天顶/zenith）轴方向的纹理分辨率便会降至单个纹素。立方环境贴图就是为解决此问题而设计的。[73]

立方贴图像是从6个主要方向（上下左右前后）拍摄照片后再拼合而成。在渲染时，立方贴图像是贴至一个无穷大立方体的6个内面，该立方体的中心位于要渲染的物体。

要在某物体表面的点**P**上读取环境贴图纹素，就要计算从摄像机到**P**的光线方向，再根据**P**的法矢量计算反射方向。然后，再计算反射光线与环境贴图的球体或立方体之相交，该相交点的纹素就能用于**P**的着色。[74]

---

[72]译注：另一种用于实时渲染的格式是双抛物面环境贴图（dual paraboloid environment map）。立方贴图（cube map）和双抛物面贴图（dual paraboloid map）还能用来渲染点光源的阴影贴图。

[73]译注：此段描述并不符合一般计算机图形学中的球面贴图，读者或可参考http://www.opengl.org/resources/code/samples/sig99/advanced99/notes/node176.html。另外，球面贴图现时主要是用于不支持立方贴图的硬件，例如Wii。球面贴图大概只有寻址简单的优点，其缺点甚多，缺点之一是不适合动态渲染环境贴图。

[74]译注：环境贴图除了用于渲染反射效果，也能用于折射效果。这些效果通常只适用于有曲面的物体（如汽车、盔甲），但不适用于大面积平面的物体（如平面镜、水面）。

10.3.1.5 三维纹理

现在的图形硬件也支持三维纹理[75]。三维纹理可以想象是一叠二维纹理。给定一个三维坐标(u, v, w)，GPU懂得如何对三维纹理进行寻址及过滤。

三维纹理非常适合描述物体的体积特性。例如，我们可以用三维纹理去渲染一个大理石球体，并可以用任意平面切割该球体。其纹理会显得连贯，无论在哪里切割，纹理仍看似正确，因为该纹理在整个球体的体积内都有明确界定而且连续。[76]

10.3.2 高动态范围光照

显示设备（如电视或CRT显示器）只能产生有限的强度范围。这是为何帧缓冲里的色彩通道限于0～1范围的原因。然而在真实世界，光的强度可以任意增大[77]。**高动态范围**（high dynamic range，HDR）光照尝试捕捉如此大范围的光照强度。

使用HDR光照时，不会把计算的强度结果随意截取。其结果影像会以某种格式保存，该格式容许储存大于1的强度。这么做就能无失真地同时保存亮区和暗区的细节。

在把影像显示于屏幕之前，还需进行一个**色调映射**（tone mapping）处理，把影像的强度调整至显示设备所支持的范围。在渲染引擎中运用色调映射，可以仿造许多现实世界的效果，例如从黑房里走到明亮区域所造成的短暂失明现象，或是看到很强的背光从物体边缘溢出（此称为**敷霜效果**/bloom effect）。

HDR影像的表示方法之一就是，把红绿蓝通道各储存为32位浮点数，而不使用8位整数。另一种方法是使用完全不同的色彩模型。例如log-LUV色彩模型就是HDR光照的流行之选。在此模型中，色彩由一个强度（intensity）通道（L）和两个色度（chromaticity）通道（U及V）去表示。由于人眼对强度改变的敏感程度高于色度，所以L通道以16位储存，而U和V各用8位。此外，L是以对数比例（底数为2）表示的，以捕捉非常大范围的光强度。

10.3.3 全局光照

我们在10.1.3.1节中曾提及，全局光照（global illumination，GI）是指一类光照算法，这些算法考虑到光从光源传送至虚拟摄像机之间与多个物体的互动。全局光照可营造不同效果，例如，遮挡物产生的阴影、反射、焦散，以及一个物体的颜色能溢出至其附近的物

[75] 译注：也称为体积纹理（volume texture）。

[76] 译注：体积纹理所需的内存很高，例如，$256 \times 256 \times 256$的32位纹理便需要64MB。因此体积纹理并不常用于现在的游戏中。另一个可行方法是程序纹理（procedural texture），例如，大理石的体积纹理可以由Perlin噪声合成。

[77] 译注：例如，满月月光的强度大约是太阳光的50万分之一。

体。以下我们会简述几个最常见的全局光照技巧。有些方法是为某独立效果而设，例如阴影或反射；有些方法则是为整体的全局光传输而设，例如辐射度算法及光线追踪法。

10.3.3.1 阴影渲染

表面遮挡光源路径就会产生阴影。由理想点光源产生的阴影应有锐利的边缘，而现实世界的阴影边缘却是模糊的，该模糊部分称为半影（penumbra）。半影的出现，是由于现实世界的光源会覆盖一定面积，因此会产生以不同角度掠过物体边缘的光线。

两个最流行的阴影渲染技巧为**阴影体积**（shadow volume）和**阴影贴图**（shadow map），以下会分别做简单介绍。在两种技巧中，都会把场景中的物体分为3个类别：投射阴影的物体、接收阴影的物体，以及完全被阴影渲染忽略的物体[78]。同样地，光源也可以标识为产生或不产生阴影的。这些是重要的优化，令生成场景中的阴影时，能限制所需处理的光源和物体组合数量。

阴影体积

在阴影体积技术中，会从产生阴影的光源位置观察每个投射阴影的物体，在那个视角判断物体的轮廓边缘（silhouette edge）。这些边缘沿光线方向伸出（extrude），产生一个几何立体，该几何立体代表着光线被投射阴影物体遮挡所造成的空间体积，如图10.51所示。

图 10.51: 光线视点所见的轮廓边缘向光线方向伸出，产生阴影体积。

[78] 译注：还有一类既投射阴影也接收阴影的物体。若物体能接收自身投射的阴影，那些阴影称为自阴影（self-shadow）。

阴影体积使用一种特殊的全屏缓冲产生阴影，此缓冲称为模板缓冲（stencil buffer），它对应屏幕每个像素储存一个整数值。渲染时可用模板缓冲作为遮罩，例如，我们可以把GPU配置成，渲染某片段时，仅当模板缓冲中对应的值不是0才进行渲染。此外，也可配置GPU，使渲染几何物体时以几种不同的方式更新模板缓冲里的值。

让我们回到用阴影体积渲染阴影的话题。首先，我们要渲染一遍没有阴影的场景，同时填充准确的深度缓冲。然后把模板缓冲清空，使当中每个像素的值都设为0。之后就可以从摄像机的视角渲染阴影体积，渲染该体积时，若像素属正向的三角形，便要令模板缓冲的值加1，相反背向的三角形要令模板缓冲的值减1。在屏幕空间中，阴影体积以外的像素，其对应模板缓冲的值当然维持是0。有趣的是，阴影体积的正向三角形与背向三角形重叠的屏幕位置，其模板缓冲值也是0，因为正向的面片使该值加1，而背向的面片又使该值减1。在背向面片被“真实”场景几何物体遮挡的部分，模板值就是1。这就告诉我们屏幕中哪些像素是在阴影之中。所以我们可以再渲染第3个步骤，把那些模板值非0的区域加深颜色。[79]

阴影贴图

所谓的阴影贴图技术，实际上是进行每片段的深度测试，但该“深度”非从摄像机的视角去计算，而是从光源的视角计算。使用阴影贴图时，需要把场景渲染两次。首先，从光源视角渲染场景，把渲染结果的深度缓冲储存为**阴影贴图**纹理。然后，以正常方式渲染场景，渲染每个片段时使用阴影贴图判断该片段是否在阴影内。而判断方法是，若该片段自光源的距离比阴影贴图里的对应深度值远，那么便代表该片段被遮挡，也就是位于阴影范围内。这一过程的原理，等同于使用深度缓冲判断片段是否被较近的三角形遮挡。

阴影贴图仅含深度信息，其每个纹素记录从光源方向来说最近遮挡物的深度[80]。因此，阴影贴图通常会使用硬件的双倍速模式，仅填充深度缓冲（由于我们只关心深度）[81]。点光源需要使用透视投影来渲染阴影贴图[82]，而平行光则使用正射投影。

---

[79]译注：用加深颜色方式去使用阴影体积，并不能模仿真实的情况，例如，这样可能会看到阴影中的高光形状，也不能正确渲染多个光源的阴影。较正确的做法是，第一次渲染场景时，只渲染环境（ambient）和放射（emissive）的光照项。之后对于每个投射阴影的光源，先把模板清0，再渲染受其影响又投射阴影的几何物体的阴影体积，然后以模板为遮罩再渲染那些几何物体一次，这次只渲染漫反射（diffuse）和镜面反射（specular）的光照项。实践上，阴影体积还有很多必须注意的细节，读者可参考http://http.developer.nvidia.com/GPUGems3/gpugems3_ch11.html。然而，由于阴影体积的填充率较阴影贴图高，阴影贴图目前较适合现在的硬件，故成为主流技术。

[80]译注：此句原文对深度贴图纹素的意义解释得不够清楚，译者稍做补充。

[81]译注：在PC的Direct3D 9中，并没有标准方法可以把深度缓冲当作阴影贴图，必须使用个别硬件厂商提供的特殊方法。若要简单支持所有厂商，须把深度写进色彩缓冲里。

[82]译注：一般来说，小于180°角的聚光灯才可使用透视投影来渲染阴影贴图。对于点光源的阴影贴图，可使用立方贴图（渲染6次场景），或是使用双抛物面贴图（渲染2次场景）。但后者也会造成一些精度问题，可参考http://www.mpi-inf.mpg.de/~tannen/papers/cgi_02.pdf。

使用阴影贴图渲染场景时，要使场景以摄像机视角正常地渲染。对于每个三角形的顶点，我们计算该顶点在**光源空间**（light space）中的位置，光源空间即是指渲染阴影贴图时的“观察空间”[83]。如同其他顶点属性，这些光源空间坐标会在三角形上进行插值。这样可以获得每个片段在**光源空间**中的位置。判断某片段是否在阴影之内，首先把片段的光源空间位置(x, y)转换为阴影贴图的纹理坐标(u, v)，然后就用该片段的光源空间z值和阴影贴图的纹素比较。如果片段的光源空间z值远于阴影贴图纹素所储存的距光源距离，那么该片段必然是被其他较近的几何图形所遮挡，也即是在阴影之中。否则，片段就不在阴影之中。片段的颜色可基于此信息做出调整。图10.52展示了整个阴影贴图的过程。[84]

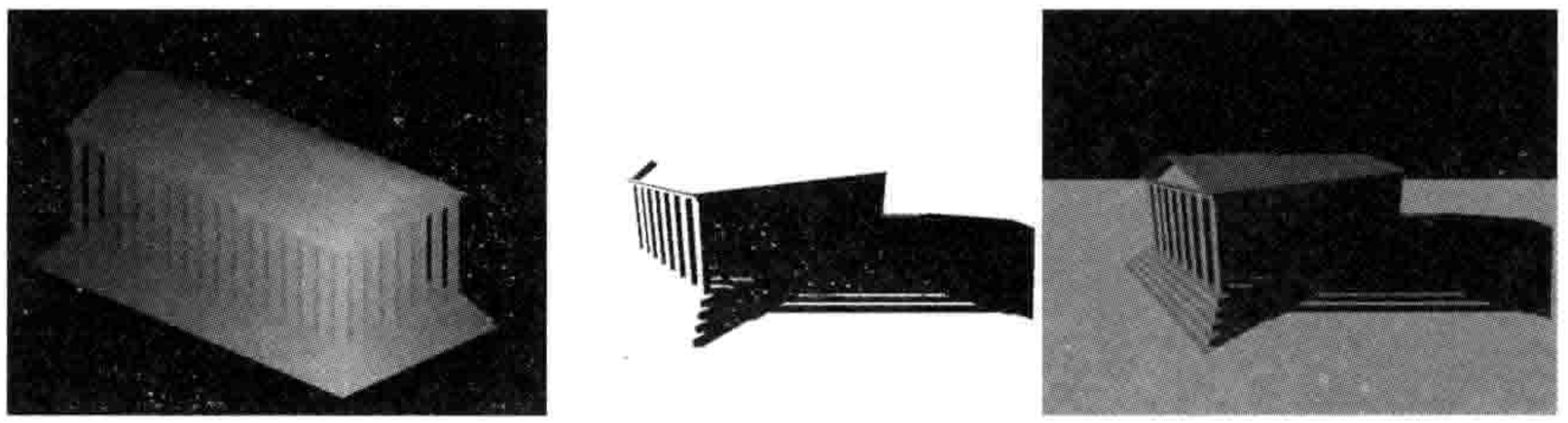


图 10.52: 左图是阴影贴图，它是从某点光源视点所渲染的z缓冲内容。中间的图中，黑色像素表示该像素在光源空间的深度测试失败（片段在阴影内），白色像素代表成功通过测试（片段不在阴影内）。右图展示使用阴影后的最终的渲染结果。

10.3.3.2 环境遮挡

环境遮挡（ambient occlusion，AO）是一种用于渲染**接触阴影**（contact shadow）的技术，所谓接触阴影是指场景仅以环境光照明时所产生的软阴影。实质上，AO描述表面上每点“可接触光线”的程度。例如，一根管子的内部表面比其外部表面能接受到的环境光照要少。若在阴天把该管子置于室外，其管内通常比管外显得较暗。

图10.53展示了一个物体表面的AO项。量度某点AO值的方法是，以该点为球心设一个非常大半径的半球体，然后计算从该点可见的半球表面面积百分比。由于AO与观察方向及入射光方向无关，静态物体的AO可以在脱机时预计算，计算结果通常会储存为纹理。[85]

[83]译注：实际上应使用投射变换，把坐标变换为光源的齐次裁剪空间（$[-1, 1]^3$），再用缩放及平移变换为纹理空间$[0, 1]^3$。详情可参考《Cg教程（The Cg Tutorial）》第9.3节关于投影贴图的变换，其做法和阴影贴图相同，详见http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html。

[84]译注：虽然阴影贴图的原理简单，但是它也有非常多的问题和对应的改善方法。尤其在2000年之后，业界和学术界对阴影贴图投入了大量的研究，产生许多新技术（和缩写）。在投影空间方面，有LiSPSM、TSM、PSSM、CSM等；过滤相关的有VSM、ESM等；软阴影相关的有PCF、PCSS、SSSS等。建议读者可先参考[1]。

[85]译注：本段所描述的AO计算是基于几何方法的，而另一种近年流行的方法是在屏幕空间计算。后者称为屏幕空间环境遮挡（screen-space ambient occlusion，SSAO），由Crytek公司的Vladimir Kajalin发明，率先应用于2007年发行的《孤岛危机（Crysis）》。其后也出现多种改进方案，例如Horizon-based AO （HBAO）、Volumetric Obscurance （VO）等。也有研究把SSAO扩展至模拟全局照明，例如screen-space directional occlusion （SSDO）。

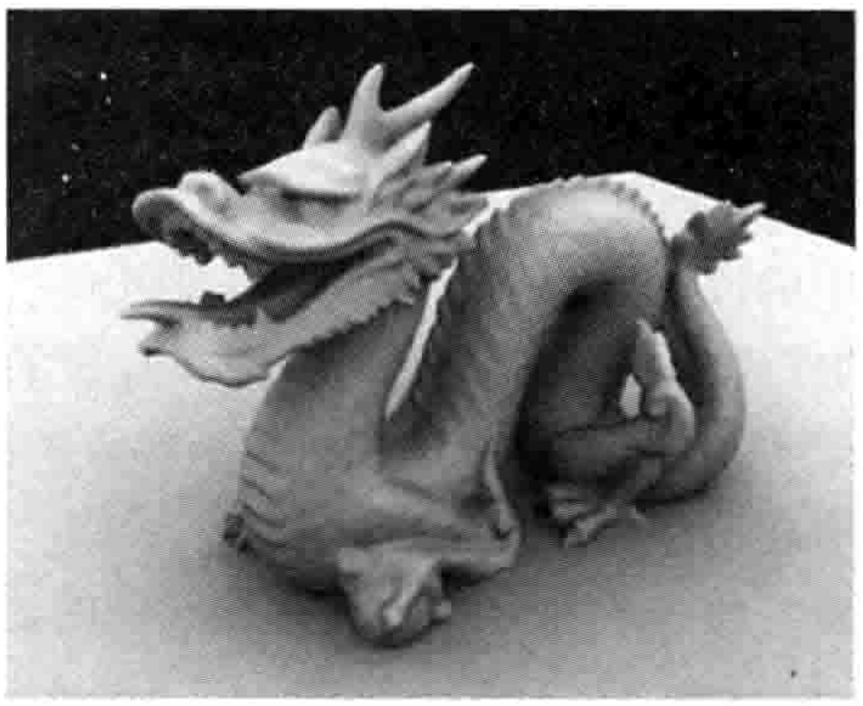


图 10.53: 使用环境遮挡渲染的龙。

10.3.3.3 镜像

当光线自非常光滑的表面反射时，就会在该表面产生一部分场景的镜像（reflection）。我们可以用多种方法实现镜像。环境贴图可用于一般光滑物体表面上产生附近环境的镜像。而平面物体（如镜子）的直接镜像，则可以把摄像机位置按该反射性表面平面进行反射变换，然后从反射的视点渲染场景至一张纹理，最后把该纹理再渲染至该反射性表面上，如图10.54所示的例子。

图 10.54: 《路易士鬼屋（Luigi's Mansion）》的镜中镜像实现方式是，先把场景渲染至纹理，再运用于镜的表面。

10.3.3.4 焦散

焦散（caustics）是指强烈反射或折射所产生的光亮高光，通常出现于非常光滑的表面，如水面或抛光金属。当反射的表面在移动时，例如水面的抖动，焦散会产生闪烁及在投射的表面上“摇曳”。要渲染焦散效果，可通过投影一张含有（具动画的）半随机亮点的纹理至受影响的物体表面。图10.55展示了此技术的例子。

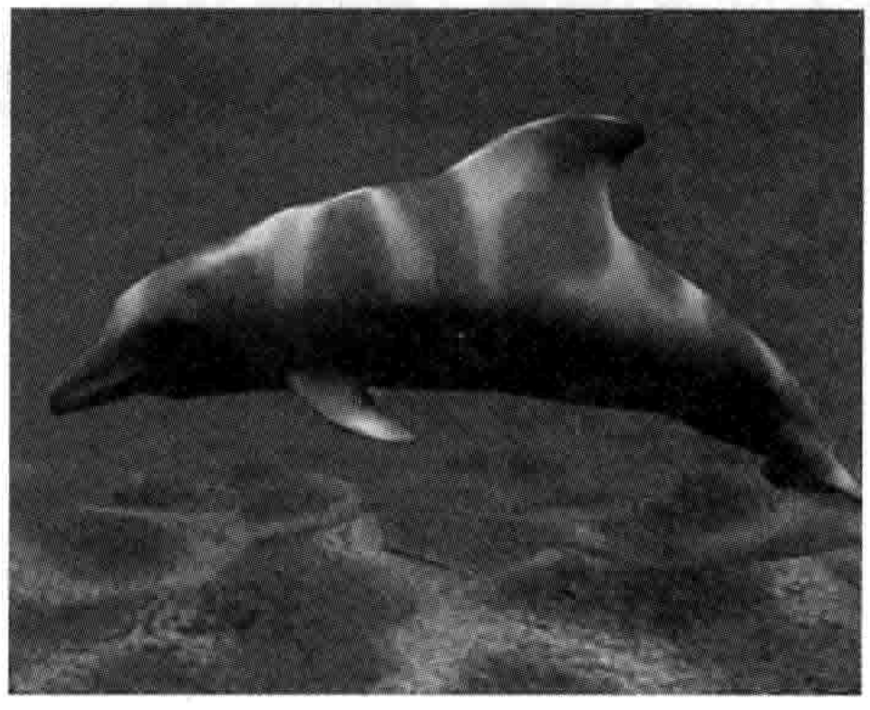


图 10.55: 通过把动画纹理投影至受影响的表面，制造焦散效果。

10.3.3.5 次表面散射

当光线到达物体表面上的一点时，光线会在表面下散射，然后在表面的其他位置离开，此现象称为**次表面散射**（subsurface scattering，SSS）。SSS现象会令人体皮肤、蜡、大理石等材质表面产生“温暖、淡淡泛光”之效果（图10.56展示了加入次表面散射前后的渲染效果）。SSS可以用比BRDF（见10.1.3.2节）更高阶一点儿的变种表示，此函数称为**双向表面散射反射分布函数**（bidirectional surface scattering reflectance distribution function，BSSRDF）。

有多种方法可以模拟SSS。基于深度贴图（depth map）的方法会渲染一张阴影贴图（见10.3.3.1节），但该阴影贴图并非用作判断像素是否在阴影之内，而是用来量度光线需要经过多少距离才能通过遮挡物。然后，在物体阴影面加入人造的漫射光照项，其光照强度与光线经过物体至另一面的距离成反比。这样，在物体相对较薄的地方，光源另一面的表面便会产生淡淡泛光。更多SSS相关信息可参考网页[86]。

[86] http://http.developer.nvidia.com/GPUGems/gpugems_ch16.html

图 10.56: 左图没有使用次表面散射渲染（即采用BRDF光照模型）。右图使用次表面散射渲染相同的龙（即使用了BSSRDF模型）。这两张图由弗吉尼亚大学的Rui Wang渲染。

10.3.3.6 预计算辐射传输

预计算辐射传输（precomputed radiance transfer，PRT）是一项较新的技术，可以实时模拟基于辐射度算法的渲染方法。其做法是，预先计算来至所有方向的入射光和表面的互动（反射、折射、散射），并把那些描述储存下来。在运行时，根据某入射光线查表，并把该光线的反射迅速地转换为准确的光照结果。

一般来说，光于某点的反射是一个复杂的函数，该函数定义于以该点为球心的半球范围。我们需要一个紧凑表示此函数的方法，才能使PRT技术实用化。常见的方法是用球谐基函数（spherical harmonic/SH basis function）的线性组合逼近此函数。此做法本质上等同于把简单的标量函数$f(x)$表示为多个经偏移及缩放的正弦波的线性组合，分别只在于SH是三维的。

PRT的细节超出本书范围，请读者参考相关文献[87]。DirectX SDK里也有展示PRT技术的范例[88]。[89]

10.3.4 延迟渲染

在传统基于三角形光栅化的渲染中，所有光照和着色计算都是在观察空间中的三角形片段上计算的。此技术有效能较差的问题。首先，我们可能做了许多不必要的工作。我们替

[87] http://web4.cs.ucl.ac.uk/staff/j.kautz/publications/prtSIG02.pdf

[88] http://msdn.microsoft.com/en-us/library/bb147287.aspx

[89] 译注: PRT的原祖论文出自2002年Slone等作者的“Precomputed Radiance Transfer for Real-Time Rendering in Dynamic, Low-Frequency Lighting Environments”, http://research.microsoft.com/en-us/um/people/johnsny/papers/prt.pdf。

三角形顶点着色，但可能在光栅化阶段才发现整个三角形会被深度测试所剔除。早期深度测试可协助消除像素着色器的计算，但仍欠完美。此外，为了处理含多个光源的复杂场景，我们最终会产生大量不同的顶点及像素着色器版本，每个版本处理不同数量的光源、不同种类的光源、不同数量的蒙皮权重等。

延迟渲染（deferred rendering）是解决这些问题的另一种场景着色方法。在延迟渲染中，主要的光照计算是在屏幕空间进行的，而非观察空间。我们首先迅速地渲染不含光照的场景。在此阶段，我们把所有将用于光照计算的信息储存在一个“深厚的”帧缓冲里，此缓冲称为**几何缓冲**（geometry buffer，G-buffer）。完成场景渲染后，就使用几何缓冲的信息来计算光照和着色。这样做通常比观察空间光照更高效，又避免了着色器版本的增长，并且可以相对容易地渲染一些非常悦目的效果，如图10.57所示。

图 10.57: 这些图片为《杀戮地带2（Kill Zone 2）》的截图，展示了延迟渲染中几何缓冲的典型成分。上方的图是最终的渲染影像。在该图之下，从上至下、左至右，分别是反照率（漫反射）颜色、深度、观察空间法线、屏幕空间运动矢量（供运动模糊之用）、镜面幂，及镜面强度。

几何缓冲在物理上是由一组缓冲实现的，但理论上它是含有丰富的光照和表面信息的单个缓冲。典型的几何缓冲可能含有以下属性：深度、剪切空间的表面法矢量、漫反射颜色，甚至是预计算辐射（PRT）系数。[90]

深入探讨延迟渲染超出本书范围，读者可参看Guerrilla Games员工的精彩报告[91]。

10.4 视觉效果和覆盖层

至此所谈及的渲染管道，主要是用于渲染三维固体物体的。通常在此渲染管道之上，还有一些专门渲染视觉效果的渲染系统，例如粒子效果、贴花（decal，用于渲染细小的几何覆盖物，例如弹孔、裂缝、抓痕，以及其他表面细节）、头发皮毛、降雨降雪、水，以及其他专门的视觉效果。另外，也可应用全屏后期处理效果，例如晕影（vignette，在画面边缘稍模糊的效果[92]）、动态模糊、景深模糊、人工性/增强性色彩处理等。最后，实现游戏的菜单系统及平视显示器（HUD）的方法，一般是通过渲染文本及其他二维/三维图形，覆盖在原来的三维场景之上。

深入讨论这些引擎系统超出本书范围。以下几节会简介这些渲染系统，并提供相关的参考信息。

10.4.1 粒子效果

粒子渲染系统是为了渲染无固定形状的物体而设的，如烟、火花、火焰等。这些通称为**粒子效果**（particle effect）。粒子效果和其他的可渲染几何物体的区别在于如下几点。

- 粒子系统由**大量相对简单**的几何物体所组成。这些几何物体通常是称为**quad**[93]的简单卡片，每个quad由两个三角形组成。
- 几何物体通常是**朝向摄像机**的（即公告板/billboard），引擎必须做相应的工作，确保面片的法矢量总是朝向摄像机的焦点。
- 其材质几乎都是**半透明**的。因此，粒子渲染系统有严格的**渲染次序**，此与场景中大部

[90]译注：一般的延迟渲染由于在每像素只能储存一组数据，因此只能渲染不透明的物体。半透明的物体要在延迟渲染完成后，才用传统的“正向着色（fordward shading）”渲染和混合。一般的延迟渲染不能使用MSAA，所以后来有许多特别为延迟渲染而开发的抗锯齿方法。此外，视乎几何缓冲的大小，延迟渲染可能需要较高的显存频宽。

[91]http://www.guerrilla-games.com/publications/dr_kz2_rsx_dev07.pdf

[92]译注：vignetting有时候会译作暗角，此译法比较容易理解此现象的原始意思，就是画面外缘比画面中心暗。vignetting几乎出现在所有照片或视频中，在美感上也有突出主体的作用，关于其详细介绍，可参看http://toothwalker.org/optics/vignetting.html。

[93]译注：quad是quadrilateral（四边形）的缩写。

分不透明物体不同。

- 粒子以多种丰富方式表现**动画**。它们的位置、定向、大小（缩放）、纹理坐标，以及许多其他着色器参数都是于每帧有所变化的。这些改动通常用手工制作的动画曲线或程式方法来定义。
- 粒子通常会不断**出生及湮灭**。粒子发射器是游戏世界中的逻辑实体（logical entity），以用户设置的速率创造粒子。粒子灭亡的原因包括：碰到预先定义的死亡平面、已存活超过用户定义的时间，或是其他用户设置的条件。

粒子效果可以用正常的三角形网格几何物体配合适当的着色器进行渲染。然而，由于上述列出的独特性质，真实的游戏引擎总是会以专门的动画及渲染系统来实现粒子效果。图10.58是一些粒子效果的例子。

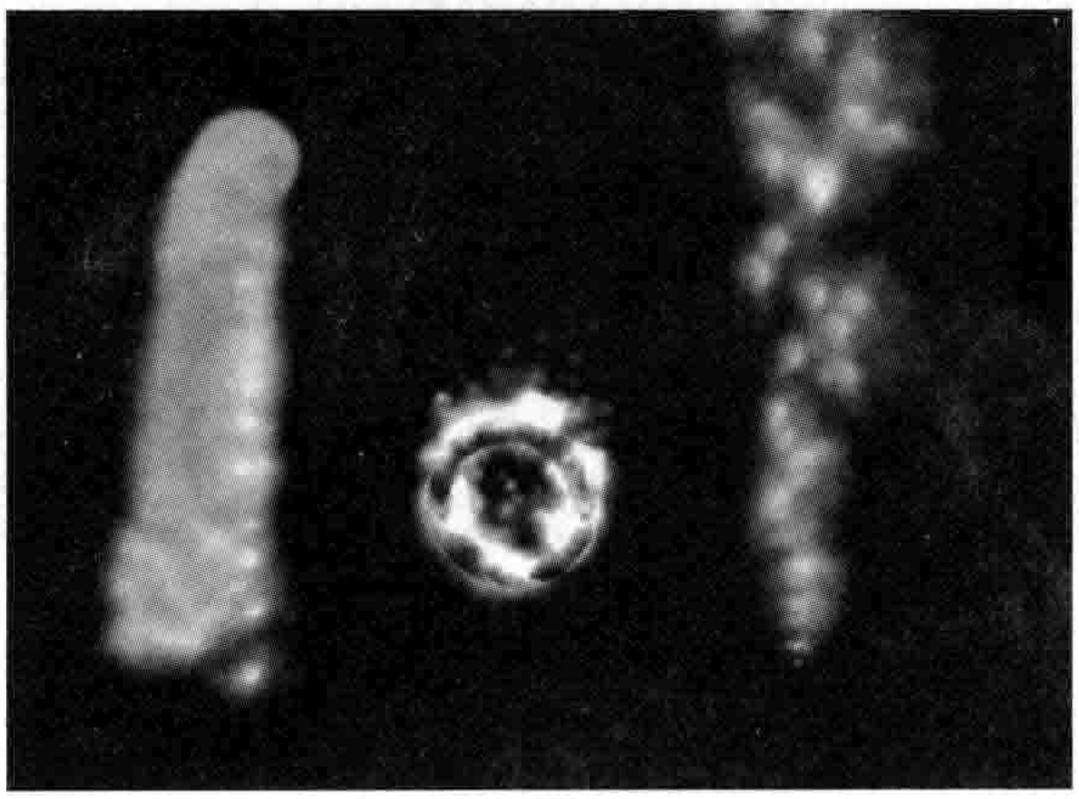


图 10.58: 一些粒子效果。

粒子系统的设计和实现是一个大题目，本身可用多个章节进行讲解。关于粒子系统的更多信息，可参考[1]的10.7节、[14]的20.5节、[9]的13.7节、[10]的4.1.2节。

10.4.2 贴花

贴花（decal）是覆盖在场景中正常物体上，相对较小的几何物体，用于动态改变物体表面的外观。弹孔、脚印、抓痕、裂缝等都是贴花的例子。

现代引擎最常用的贴花实现方法就是，把贴花设为长方形区域，按某方向投影在场景中。这会形成一个三维空间中的长方体。长方体在投射方向与表面第一次相交的地方就成为贴花的面片。从相交的表面中提取三角形后，用投影长方体的四块包围平面裁切这些三角形。通过生成适当的顶点纹理坐标，把三角形贴上所需的贴花纹理。这些含贴图的三角形渲

染在正常场景之上，有时候会使用视差贴图带出深度的错觉，并且用少许深度偏移（z-bias）（通常是往近平面稍做移动）避免和原来的表面产生深度冲突（z-fighting）。最终就能创造出弹孔或抓痕等表面改造。图10.59展示了弹孔贴花的效果。

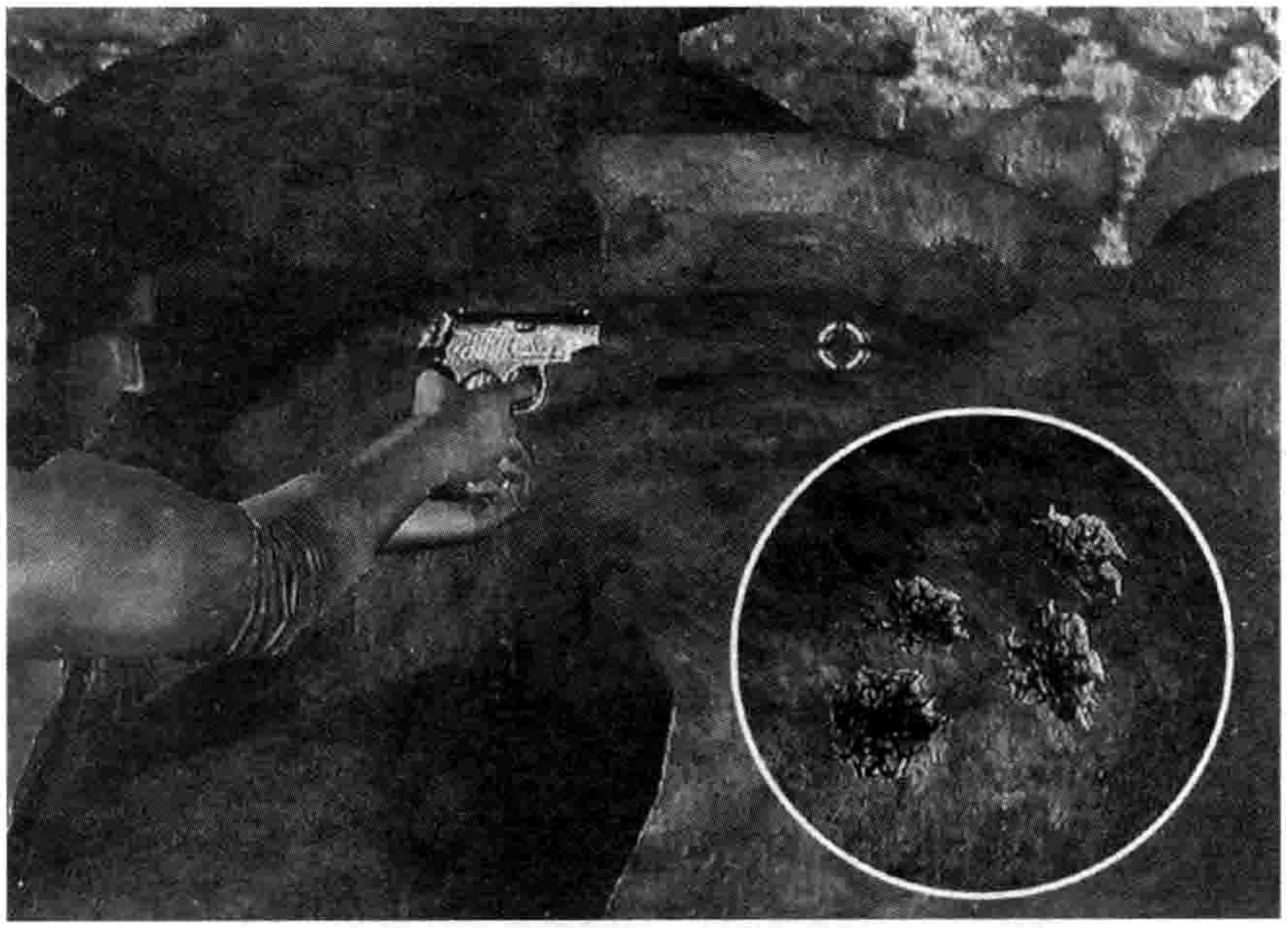


图 10.59: 《神海》中的视差贴图贴花。

关于生成和渲染贴花的更多信息，可参考[7]的4.8节、[28]的9.2节。

10.4.3 环境效果

采用颇为自然或真实的场景作为背景的游戏，都需要一些环境渲染效果（environmental rendering effect）。这些效果通常由专门的渲染系统实现。我们以下会简介当中几个最常见的系统。

10.4.3.1 天空

游戏世界中的天空需要栩栩如生、细致分明，但从技术上来说天空和摄像机之间的距离非常远。因此我们不可能用现实中的方式为天空建模，取而代之，需要使用一些专门的渲染技术。

其中一个简单的方式就是，先把帧缓冲填满天空的纹理，才去渲染三维几何图形。该天空纹理应该有接近1 : 1的纹理像素比，使纹理逼近屏幕的分辨率。天空纹理可根据游戏摄像机的移动而相应旋转及卷动。在渲染天空时，我们必须确认把所有像素的深度设置为最大值。这样可以确保所有三维场景物体都会排于天空之前[94]。《迅雷赛艇（Hydro Thunder）》就是完全采用这种方式渲染天空的。

有些游戏玩家可以任意改变视角方向，这种情况就需要**天空穹顶**（sky dome）或**天空盒**（sky box）。渲染穹顶或盒子时，总是把它们的中心置于摄像机的位置，这样无论摄像机在游戏中如何移动，天空看起来就像在无限远的地方。如同天空纹理般，渲染天空时要把帧缓冲的所有像素设为最大深度值。这样天空穹顶或天空盒相对其他物体可以很细小。它的大小并不重要，只要它能填满整个帧缓冲就可以了。关于天空渲染可参考[1]的10.3节及[38]的第253页。

另一方面，云通常也需要使用专门的渲染及动画系统实现。在早期的游戏如《毁灭战士（Doom）》和《雷神之锤（Quake）》中，云仅是几块平面，贴上卷动的半透明云纹理。较近期的云渲染技术，包括使用面向摄像机的卡板（公告板）、基于粒子效果的云，以及体积云（volumetric cloud）效果。

10.4.3.2 地形

地形（terrain）系统是为了建立地表的模型，并作为摆置各式各样静态动态元素的画布。有时候我们会使用如Maya的软件为地形建模。但若玩家能看见很远的景物，我们通常需要某种动态镶嵌或其他层次细节（level-of-detail，LOD）系统。我们也要限制所需的数据量，以表示非常大型的户外区域。

高度场地形（height field terrain）是大型地形建模的流行之选。因为高度场地形通常储存为灰阶纹理贴图，其数据量相对较少。大多数基于高度场的地形系统中，会用规则的栅格模式来镶嵌水平面（$y = 0$），然后以高度场纹理的采样决定地形顶点的高度。每个区域单元的三角形数量可以按摄像机距离来调整，使大尺度的地形特征能在远处观看，而同时能表现近距离地形的层次细节。图10.60是以高度场位图定义地形的例子[95]。

地形系统通常会提供专门的工具“粉刷”地形数据，雕刻不同地形特征，如路面、河流

---

[94]译注：若采用先渲染天空后渲染场景的次序，渲染天空时其实可以同时关上深度测试（z-test）及深度写入（z-write）。但若天空采用较复杂的像素着色器，而且又会被大量前景所覆盖，可考虑最后才渲染天空，这时候需要开启深度测试，但仍可关上深度写入，因为写入深度不会再有用途。

[95]译注：通常8位通道的纹理的精度不足以表示一般的地形，例如，用0～255表示水平面至海拔255m，那么高度的精确度就只有1m。所以一般来说需要16位或更高的精度。而另一个方法是利用地形高度的局部性，把地形分区，每区各自定义其高度的最小至最大值，那么便可以用较少的数据量表示更高的精确度。

等。地型系统的贴图方法，一般是用4张或以上的纹理互相混合。那么美术人员只需把某层纹理显露出来，便可以“粉刷”出草地、泥地、碎石地，或其他地形特征。这些纹理层也可以从一层渐变混合至另一层，来做到平滑的纹理过渡。有些地形工具容许把地形镂空，以加入建筑物、渠沟，或其他用网格建模的特殊地形特征。有时候，地形编辑工具会直接整合至游戏世界编辑器，而在其他引擎中就可能是独立的工具。

当然，高度场地形仅是多种游戏地表建模的方法之一。对于地形渲染的更多信息，可参考[6]的4.16～4.19节，以及[7]的4.2节。

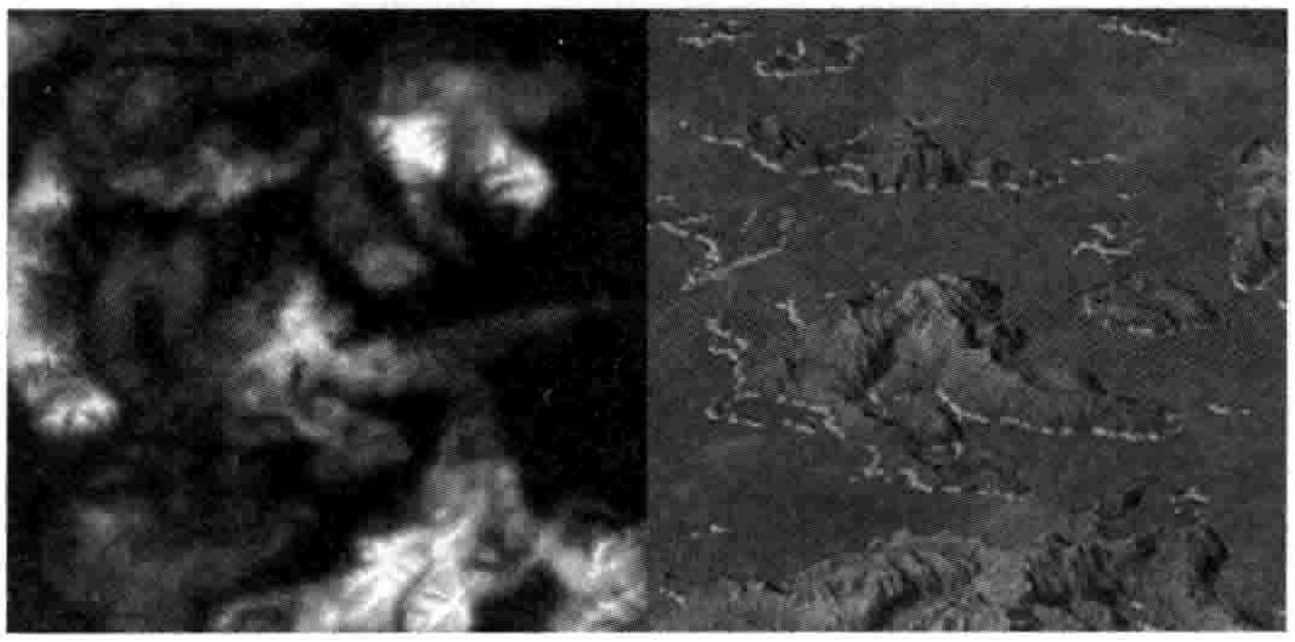


图 10.60: 灰阶高度场位图（左图）可用来控制地形栅格网格（右图）的顶点垂直位置。在此例子中，使用了一个水体平面创造岛屿。

10.4.3.3 水体

现在的游戏中，水体渲染器随处可见。水体有很多不同种类，包括海洋、池塘、河流、瀑布、喷水池、射水、水坑、潮湿的硬地等。每种水体通常需要一些专门的渲染技术。有些也需要力学运动模拟。大型的水体可能需要动态镶嵌或其他类似地形系统的LOD技术。

水体系统有时候会与游戏的刚体动力学系统互动（漂浮、喷射产生的力等），有时也会与游戏性系统互动（如湿滑路面、游泳机制、潜水机制、被向上喷射的水弹起等）。水体效果通常由不同的渲染技术及子系统结合创造。例如，瀑布可能使用专门的水体着色器、卷动纹理、在瀑布底处放置模拟雾的粒子效果、模拟水泡的类贴花覆盖等。今时今日，游戏提供惊人的水体效果，而且从实时流体动力学技术等活跃研究中，能预见未来几年内的水体模拟变得更丰富及真实。关于水体模拟及渲染技术的进一步信息，可参考[1]的9.3、9.5、9.6节，[13]，以及[6]的2.6、5.11节。[96]

[96]译注：译者在此多推荐一本相关专著——Robert Bridson的《Fluid Simulation for Computer Graphics》，于2008年由AK Peters出版。据说书中的内容已被应用于《小小大星球3（Little Big Planet 3）》（http://advances.realtimerendering.com/s2011/index.html）。

10.4.4 覆盖层

多数游戏都会有平视显示器（HUD）、游戏内图形用户界面及菜单系统。这些**覆盖层**（overlay）通常是用二维或三维的图形直接渲染在观察空间或屏幕空间中的。

覆盖层通常在主场景之后渲染，并关上深度测试以确保它们会显示在三维场景之上。二维覆盖层的实现方法通常是，使用正射投影渲染屏幕空间的四边形（一对三角形），也可以使用正常的透视投影并把几何物体置于观察空间，使这些几何物体随摄像机移动。

10.4.4.1 归一化屏幕坐标

二维覆盖层的坐标可使用屏幕像素为单位。然而，若读者的游戏要支持多个屏幕解分辨率（在PC游戏中很常见），那么更好的办法是使用**归一化**屏幕坐标（normalized screen coordinates）。归一化坐标中，两个轴其中**一个**轴的范围是由0至1（但不能两个都是0～1，见下文），而且能轻易缩放至不同分辨率下的像素单位。这么做我们放置视觉元素时，就完全无须顾虑屏幕分辨率（仅需考虑少许有关长宽比的事情）。

最容易定义归一化坐标的方法是，把y轴的范围设置为0.0～1.0。当使用4：3长宽比，x轴的范围就是0.0至1.333($=4/3$)，而16：9时x轴范围则为0.0～1.777($=16/9$)。重要的是，不要定义两轴的范围都是0～1。这样做会使正方形的视觉元素的x方向尺寸异于其y方向的尺寸；反过来说，长宽尺寸同值的元素，在屏幕上看上去并不是正方形的！此外，“正方形”的元素在不同长宽比中会拉伸成不同形状——这并非是可接受的事。

10.4.4.2 屏幕相对坐标

要完善归一化屏幕坐标，应令它可以使用绝对或相对坐标。例如，正数的坐标可理解为相对于屏幕左上角计算，而负数的坐标是相对右下角计算。那么，若我想把一个HUD元素置于离右缘或下缘的某个距离，在长宽比改变时就不必改变其归一化坐标。我们还可以建立更丰富的对齐方式，例如，对齐至画面中心，或是对齐至另一视觉元素。

然而，有些覆盖层元素并不能轻松地使用归一化坐标，同时在4：3及16：9长宽比下正常显示。或许可以考虑为每种长宽比设置不同的布局，那么便可以独立地做微调。

10.4.4.3 文本及字体

游戏引擎的文本/字体系统通常会实现为一种特殊的二维（或有时候三维）的覆盖层。在其核心中，文本渲染系统需要按字符串显示一串文字字形（glyph），并以某种方向在屏

幕上排列。字体（font）通常以含有字形的纹理贴图实现。另外，再保存一个字体描述文件，内含每个字形在纹理中的包围盒、字体布局的信息，如字距调整（kerning）、基线偏移（baseline offset）等。[97]

优良的文本/字体系统必须处理不同字符集的区别，以及各语言固有的阅读方向。有些文本系统也会提供一些有趣的功能，例如在屏幕上产生字符的多种动画，或对个别字符产生动画等。有些游戏引擎甚至会实现Adobe Flash标准的子集，为覆盖层提供丰富的二维效果[98]。然而，谨记当实现游戏字体系统时，只实现游戏实际上需要的功能。若引擎实现了高级文本动画，而游戏根本不需要文本动画，那是毫无意义的!

10.4.5 伽马校正

阴极射线管（cathode ray tube，CRT）显示屏往往有非线性的亮度响应曲线。即是说，若送往CRT显示屏的红、蓝、绿值以线性递增，屏幕上显示出来的结果从人眼的感知上的亮度则并非线性的。视觉上来说，较暗的区域显得比理论上来说还暗。图10.61显示了这种情况。

一般CRT显示屏的伽马响应曲线（gamma responsive curve）可用简单公式建模：

$$V_{\text{out}} = V_{\text{in}}^{\gamma}$$

其中$\gamma_{\text{CRT}} > 1$。要校正此情况，颜色传送至CRT显示器之前，通常会进行一个逆变换（即使用$\gamma_{\text{CRT}} < 1$）。一般CRT显示屏的γ_{CRT}值为2.2，所有校正值通常是$\gamma_{\text{corr}} = 1/2.2 = 0.455$。图10.62展示了这些伽码编码及解码曲线。

三维渲染引擎可执行伽码编码，以确保最终影像中的值是正确地获伽马校正的。然而当中有一个问题，就是纹理贴图所使用的位图通常本身已获伽马校正。高质量的渲染引擎会考虑到此实际情况，所以会在渲染前先把纹理进行伽码解码，并在最终渲染场景后才重新进行伽码编码，使颜色能正确地重现在屏幕上。[99]

---

[97]译注：本段谈及的主要是指拼音文字（如英文）的字体渲染方式。而由于中、日、韩的文字字形较多，如需显示大量文字（或由玩家输入的文字），就不会使用纹理储存所有字形，而是把最近用到的字形缓存至纹理中。此过程可以使用一些字体渲染程序库达成，例如freetype（http://www.freetype.org/）。

[98]译注：现时游戏业界最常用的Flash渲染中间件是Scaleform（http://gameware.autodesk.com/scaleform），另一个选择是IGGY（http://www.radgametools.com/iggy.htm）。此外，也可采用一个开源的Flash渲染库gameswf（http://tulrich.com/geekstuff/gameswf.html）。

[99]译注：如果纹理不经任何计算便直接写在色彩缓冲（例如二维的GUI），那么并不需要这样做。但在渲染三维场景时，光照及alpha混合等计算都需要使用线性的贴图。所以一般是使用线性贴图完成所有渲染后（包括全屏后期处理），才把结果进行伽码编码。

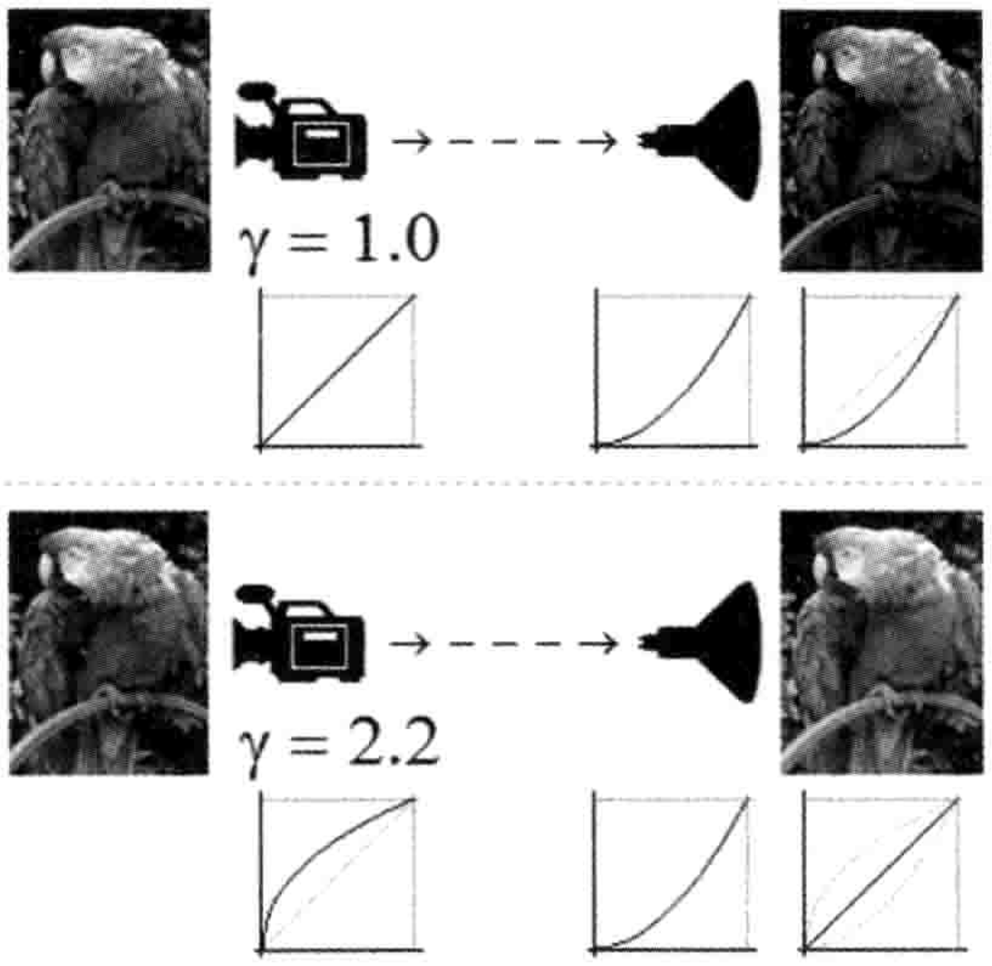


图 10.61: CRT的伽马响应对影像品质的影响，并如何做出校正。图片来自维基百科。

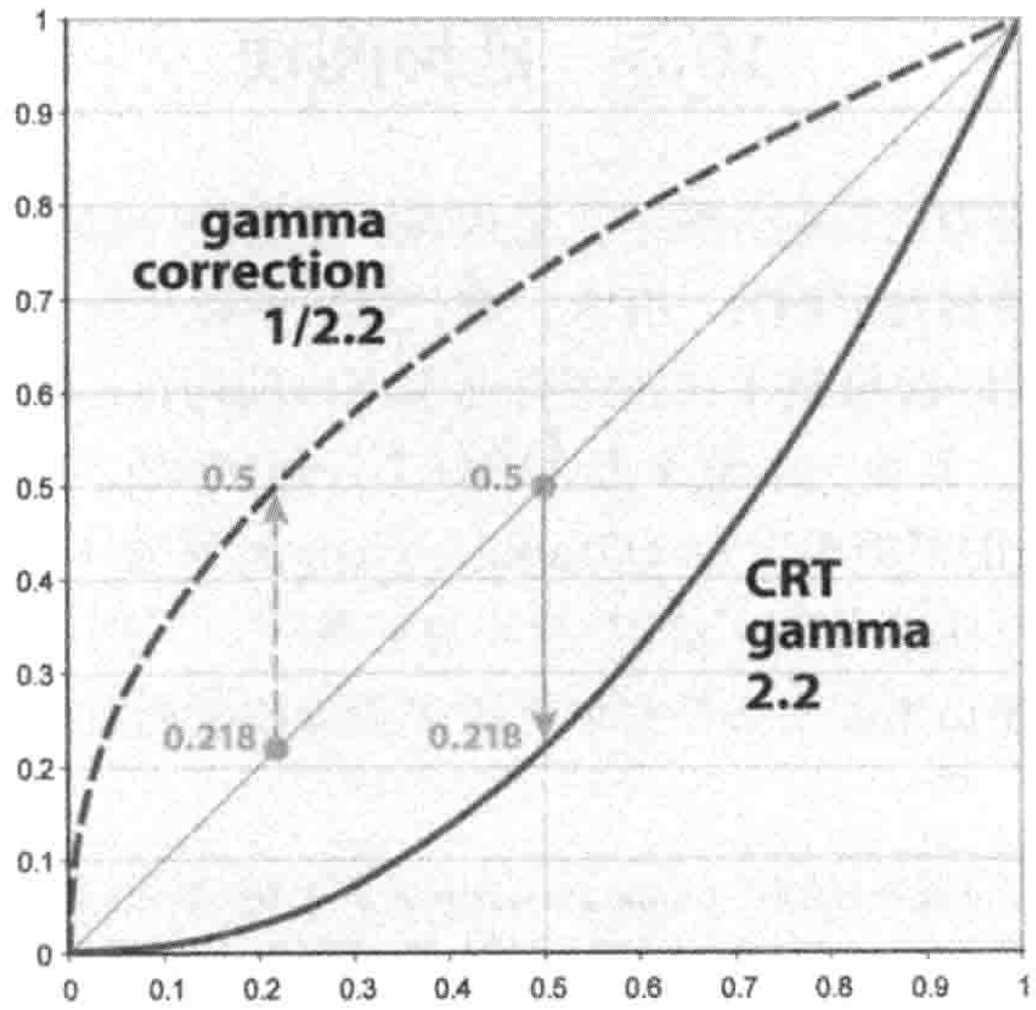


图 10.62: 伽马编码及解码曲线。图片来自维基百科。

10.4.6 全屏后期处理效果

全屏**后期处理效果**（full-screen post effect）应用在已渲染的三维场景上，以增加真实感或做出特殊的风格。这些效果的实现方法，通常是把屏幕的所有内容传送至含所需效果的像素着色器。而实际过程就是把含未处理场景的贴图，以一个全屏四边形进行渲染。以下列举出一些全屏后期处理效果的例子。

- **动态模糊**（motion blur）：此效果通常的实现方法为，渲染一个屏幕空间的速度矢量缓冲区，并使用此矢量场选择性地模糊已渲染的影像。产生模糊的方法是把一个卷积核（convolution kernel）施于影像[100]。
- **景深模糊**（depth-of-field blur）：此模糊效果使用深度缓冲区的内容调整每像素的模糊程度。
- **晕影**（vignette）：此效果通过降低屏幕四角的亮度和饱和度，产生类似电影的戏剧性效果。实现此效果的方法，可以简单地在屏幕上覆盖一张贴图。此效果的另一变种是用来产生玩家使用双筒望远镜或武器观景器的效果。
- **着色**（colorization）：可用后期处理效果以任意方式修改屏幕上的颜色。例如，所有红色以外的颜色可以去饱和度至灰色，产生有如电影《辛德勒的名单（Schindler's List）》中红衣小女孩一幕一样的震憾效果。[101]

10.5 延伸阅读

我们在本章以非常有限的篇幅介绍了大量内容，但这些内容都只是很基础的知识。读者必然希望更仔细地深度探索这些题目。有关三维计算机图形及动画的整个过程概略，笔者高度推荐[23]。而当代的实时渲染技术在[1]中有非常深入的探讨，而[14]是所有关于计算机图形学的权威参考指南[102]。其他三维渲染方面的好书还包括[42]、[9]、[10]。三维渲染的数学可参考[28]。图形程序员的书柜中没有《Graphics Gems》系列（[18]、[4]、[24]、[19]、[36]）及《GPU Gems》系列（[13]、[38]、[35]）就不算完整[103]。当然，此简短的参考列表只是一个开始，读者在游戏程序员生涯中必然会遇到更多渲染和着色器方面的优良读物。[104]

---

[100]详见Dale A. Schumacher于[4]发表的文章“Image Smoothing and Sharpening by Discrete Convolution”。

[101]译注：在《爱丽丝：疯狂回归（Alice: Madness Returns）》中，当主角进入“歇斯底里”模式时，所有除了红色鲜血以外的东西都会变成灰色。但做法并非单靠缓冲区的颜色进行变换，而是要指明哪些部分是血。

[102]译注：此书自1990年的第2版后（1995年有C语言的第2版），终于在2013年推出了第3版，其大纲及内容几乎全部重写。有关非实时渲染也可阅读Matt Pharr等人的《Physically Based Rendering: From Theory to Implementation》第2版。

[103]译注：游戏图形方面现在还有《ShaderX》系列和更新的《GPU Pro》系列。

[104]译注：译者长期搜集计算机图形学相关的一些书籍，可访问http://book.douban.com/doulist/1445680/。

第11章　动画系统

现在的游戏多数会围绕一些**角色**（character）——通常是人类或人形角色，有时候也会是动物或异形。角色是独特的，因为他们需要流畅地以有机方式移动。此需求成为新的技术难点，其困难程度远超模拟载具、抛射体、足球、俄罗斯方块等刚体物体。引擎中的**角色动画系统**（character animation system）负责为角色灌输自然的动作。

以下我们将会看到，动画系统给予游戏设计师一套强大的工具，这些工具除了用于角色外，也能用于非角色的物体。任何非百分百刚性的物体都可利用动画系统。所以当读者看到载具上的可移动组件、以关节联系的机械、微风中摇曳的树木，甚至是游戏中会爆炸的建筑物，这些物体有很大机会会利用游戏引擎的动画系统。

11.1　角色动画的类型

角色动画技术自《大金刚（Donkey Kong）》以来经历一段漫长的发展过程。起初，游戏采用非常简单的技巧去产生栩栩如生的动作。随着游戏硬件的改进，更多高级技巧可以实时应用。今天，游戏设计师手上有许多强大的动画方法。我们在本节看看角色动画的演变，以及列出现在游戏引擎中3种最常用的动画技术。

11.1.1　赛璐璐动画

所有游戏动画技术的前身是**传统动画**（traditional animation）或**手绘动画**（hand-drawn animation）。此技术用于最早期的卡通动画。这种动画的动感由连续快速显示一串静止图片所产生，这些图片称为**帧**（frame）。实时三维渲染可想象为传统动画的电子形式，把一串静止的全屏影像不断地向观众展示，以产生动感。

赛璐璐动画（cel animation）是传统动画的一个种类。**赛璐璐**是透明的塑料片，上面可以绘画。把一连串含动画的赛璐璐放置于固定的手绘背景之上，就能产生动感，而无须不断

重复绘画静态的背景。

赛璐璐动画的电子版本是称为**精灵动画**（sprite animation）的技术。所谓精灵，其实是一张细小的位图，叠在全屏的背景影像之上而不会扰乱背景，通常由专门的图形硬件绘画。因此，精灵之于二维游戏动画，犹如赛璐璐之于传统动画。精灵是二维游戏时代最主要的技术。图11.1展示了一组著名的精灵位图，这组人形角色跑步精灵几乎用在所有美泰公司Intellivision游戏之中。这组帧被设计成就算不断重复播放也会显得顺畅——这种动画称为**循环动画**（looping animation）。而此组动画以现在的说法可称为一个跑步周期（run cycle），因为它用于显示角色跑动。角色通常有多组循环动画周期，包括多种闲置周期（idle cycle）、步行周期（walk cycle）及跑步周期（run cycle）。

图 11.1: 多数Intellivision游戏都使用到的精灵动画序列。

11.1.2 刚性层阶式动画

自三维图形技术的来临，精灵技术开始失去其吸引力。《毁灭战士》使用类似精灵的动画系统，游戏中的怪兽仅是面向摄像机的四边形，每个四边形贴上一连串纹理位图（这种纹理称为**动画纹理**/animated texture）以产生动感。这种技术在今天仍然用于低分辨率或远距离物体，例如体育馆里的群众、背景中的千兵万马对战等。然而，对于高质量的前景角色，其三维图形需要使用更进一步的角色动画方法。

实现三维角色动画，最初的方法称为**刚性层阶式动画**（rigid hierarchical animation）。此方法中，角色由一堆刚性部分建模而成。人形角色通常会分拆成骨盘（pelvis）、躯干（torso）、上臂（upper arm）、下臂（lower arm）、大腿（upper leg）、小腿（lower leg）、手部（hand）、脚部（feet）及头部（head）。这些刚性部分以层阶形式彼此约束，类似于哺乳类动物以关节连接骨骼。这样能使角色自然地移动。例如，当移动上臂时，下臂和手部会随之而动。一般的层阶会以骨盘为根，躯干和大腿是其直接子嗣，其他部分如下连接[1]：

```
Pelvis（髋关节/骨盘）
    Torso（躯干）
        UpperRightArm（右上臂）
            LowerRightArm（右前臂）
                RightHand（右手）
```

[1]译注：原文的层阶中出现了两次UpperLeftArm和UpperLeftLeg，应为笔误。

```
        UpperLeftArm（左上臂）
            LowerLeftArm（左前臂）
                LeftHand（左手）
        Head（头）
    UpperRightLeg（右大腿）
        LowerRightLeg（右小腿）
            RightFoot（右脚）
    UpperLeftLeg（左大腿）
        LowerLeftLeg（左小腿）
            LeftFoot（左脚）
```

刚性层阶技术的最大问题在于，角色的身体会在关节位置产生碍眼的“裂缝”，如图11.2的情形。对于确是由刚性部件组成的机器人及机械，刚性层阶动画能好好配合，但对于“有血有肉”的角色，仔细察看时就会出现问题。

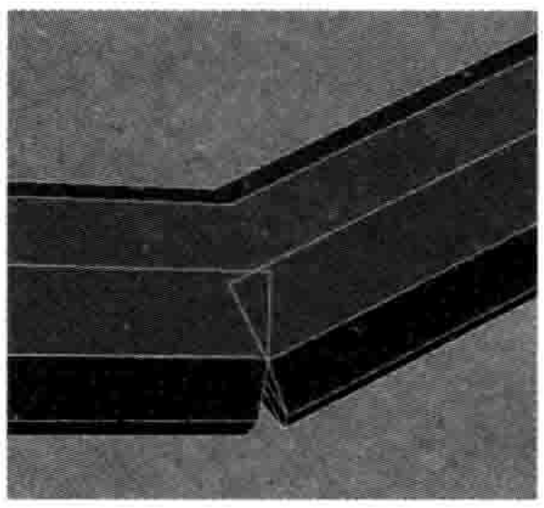


图 11.2: 在刚性层阶动画中，一个大问题是在关节位置产生裂缝。

11.1.3 每顶点动画及变形目标

刚性层阶动画由于是刚性的，往往会显得不自然。我们真正希望的是能移动每个顶点，使三角形拉伸以产生更自然生动的动作。

方法之一是使用称为**每顶点动画**（per-vertex animation）的蛮力技术。在此方法中，动画师为网格的顶点添加动画，这些动作数据导出游戏引擎后，就能告诉引擎在运行时如何移动顶点。此技术能产生任何能想象得到的网格变形（仅受表面的镶嵌所限）。然而，这是一种数据密集的技术，因为每个顶点随时间改变的动作信息都需要储存下来。因此，在实时游戏中很少会用上此技术[2]。

此技术的一个变种——**变形目标动画**（morph target animation）——应用于一些实时引擎。此方法也是由动画师移动网格的顶点，但仅制作相对少量的固定极端姿势（extreme

[2]译注：《雷神之锤III竞技场》采用每顶点动画技术，用MD3格式储存随时间改变的顶点位置和法线。

pose）。在运行时把两个或以上的这些姿势**混合**，就能生成动画。每个顶点的位置是简单地把每个极端姿势的顶点位置线性插值（linear interpolation，LERP）而得。

变形目标技术通常用于面部动画（facial animation），因为人脸具有非常复杂的解剖结构，其动作由大约50组肌肉所驱动。动画师能使用变形目标动画去完全控制脸上的每个顶点，制作出细微及极端的移动，模拟面部肌肉组织。图11.3展示了一组面部变形目标。

图 11.3: NVIDIA《Dawn》中的角色面部变形目标。

11.1.4 蒙皮动画

随着游戏硬件的能力更进一步，称为**蒙皮动画**（skinned animation）的技术就应运而生了。此技术含有许多每顶点动画及变形目标动画的优点，允许组成网格的三角形做出变形。但蒙皮动画也有刚性层阶式动画的高效性能及内存使用量特性。蒙皮动画能产生相当接近真实的皮肤和衣着移动。

蒙皮动画率先应用在如《超级玛里奥64（Super Mario 64）》的游戏中，并且仍是当今最流行的技术，它不单只应用于游戏，还应用于电影工业。许多知名的现代游戏及电影角色，如《侏罗纪公园（Jurrassic Park）》中的恐龙、《潜龙谍影4（Metal Gear Solid 4）》中的Solid Snake、《魔戒（Lord of the Rings）》中的咕噜（Gollum）、《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》中的Nathan Drake、《玩具总动员（Toy Story）》中的巴斯光年（Buzz Lightyear）、《战争机器（Gears of War）》中的Marcus Fenix 等，都是完全或部分采用蒙皮动画技术。本章余下的内容都会专注研究蒙皮/骨骼动画。

在蒙皮动画中，**骨骼**（skeleton）是由刚性的“骨头（bone）”所建构而成的，这与刚性层阶动画是一样的。然而，这些刚性的部件并不会渲染显示，始终都是隐藏起来的。称为**皮肤**（skin）的圆滑三角形网格会绑定于骨骼上，其顶点会追踪关节（joint）的移动。蒙皮上每个顶点可按权重绑定至多个关节，因此当关节移动时，蒙皮可以自然地拉伸。

图11.4中的是Crank the Weasel，它是2001年由Eric Browning为Midway家庭娱乐公司设计的游戏角色。Crank的外皮如同其他三维模型一样，都是由三角形网格组成的。然而，角色内有刚性的骨头及关节来驱动蒙皮的移动。

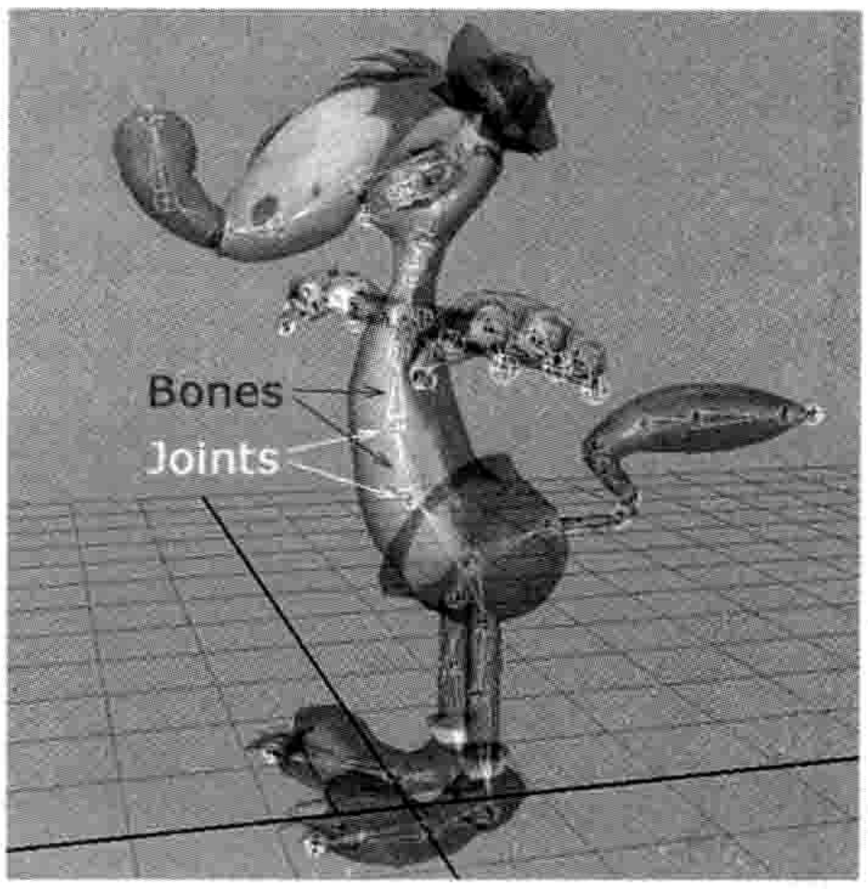


图 11.4: Eric Browning的Crank the Weasel角色及其内部骨骼结构。

11.1.5 把动画方法视为数据压缩技术

最有弹性的动画系统，可想象成动画师能控制物体表面上无穷多的点。当然，用这种方法制作动画，其结果会是无穷大量的数据！此理想的简化版本是控制三角形网格的顶点，那么实际上，我们是把描述动画的信息加以**压缩**，限制了只能移动顶点。（在控制点上加入动画，可以类比为由高次面片组成的模型的顶点动画。）而变形目标也可想象为更进一步的压缩，其压缩方法是在系统中加入更多的约束——顶点只能在一组固定数目的预定义顶点位置间的线性路径中移动。骨骼动画也是另一种通过加入约束来压缩顶点动画的方法。在此方法中，相对大量的顶点只能跟随相对少量的骨骼关节移动。

当要权衡各种动画技术时，把它们当成压缩方法来考虑会有所帮助，这种思考方式可和视频压缩技术类比。一般来说，我们选择动画技术的目标，是能提供最佳压缩而又不会产生不能接受的视觉瑕疵。骨骼动画能提供最佳的压缩，因为每个关节的移动会扩大至多个顶点的移动。角色的四肢大部分行为像刚体，所以能非常有效地使用骨骼移动。然而，面部的动作往往更为复杂，每个顶点的移动更为独立。若要使用骨骼方式制作有说服力的动画，所需的关节就会接近网格的顶点数量，因而降低了骨骼动画作为压缩方法的效能。这也是为何动画师偏爱使用变形目标而非骨骼方法制作面部动画的一个原因。（另一原因是，动画师用变形目标技术制作面部动画，工作更为自然。）

11.2 骨骼

骨骼（skeleton）由刚性的**关节**（joint）**层阶结构**所构成。在游戏业界，“关节”和“骨头（bone）”这两个术语通常会交替使用，但**骨头**一词其实名不副实。技术上来说，关节是动画师直接控制的物体，而骨头只是关节之间的空位。以Crank the Weasel角色模型的骨盘关节为例，它是单个关节，但由于它连接至4个其他关节（尾、脊柱、左右髋关节），骨盘关节看上去有如连接着4根骨头。图11.5详细展示了此例子。游戏引擎并不在意骨头，只在乎关节。因此每当读者在业界听到“骨头”，99%的情况实际上是指关节。

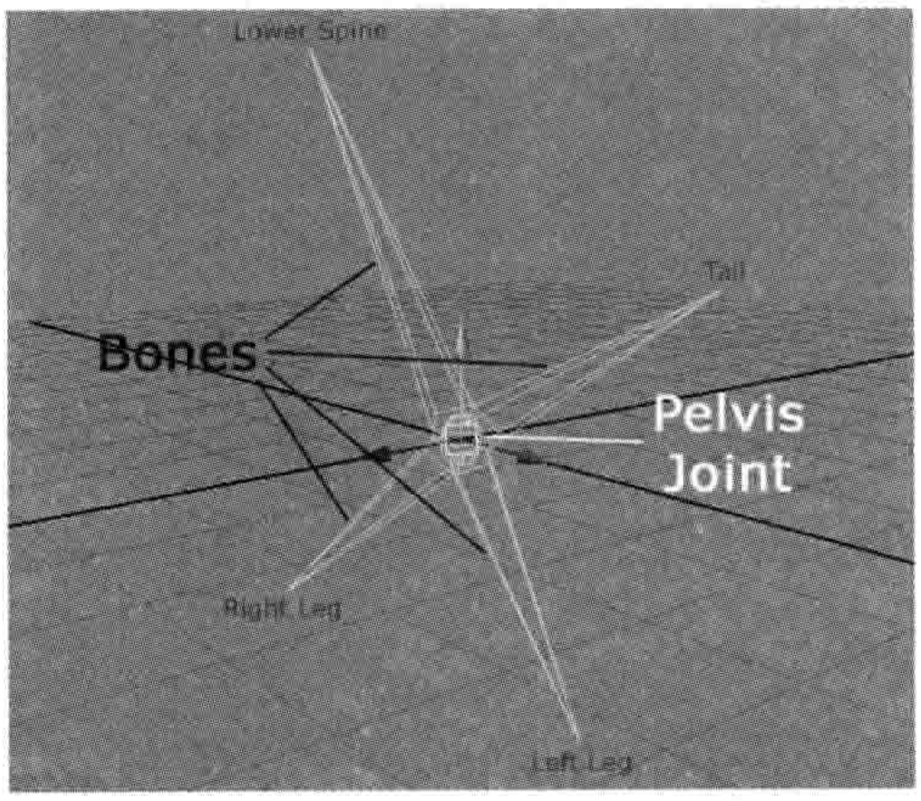


图 11.5: 角色的髋关节连接了4个其他关节（脊椎下部、尾、双腿），因而产生了4根骨头。

11.2.1 骨骼层阶结构

如前所提及，骨骼的关节形成层阶结构，也即树结构。选择其中一个关节为根，其他关节则是根关节的子孙。蒙皮动画所用的关节层阶结构，通常和刚性层阶相同。例如，人形角色的关节层阶结构可能是这样的：

```
Pelvis（髋关节 / 骨盘）
    LowerSpine（脊椎下部）
        MiddleSpine（脊椎中部）
            UpperSpine（脊椎上部）
                RightShoulder（右肩）
                    RightElbow（右肘）
                        RightHand（右手）
                            RightThumb（右拇指）
                            RightIndexFinger（右食指）
                            RightMiddleFinger（右中指）
```

```
                    RightRingFinger（右无名指）
                    RightPinkyFinger（右小指）
LeftShoulder（左肩）
    LeftElbow（左肘）
        LeftHand（左手）
            LeftThumb（左拇指）
            LeftIndexFinger（左食指）
            LeftMiddleFinger（左中指）
            LeftRingFinger（左无名指）
            LeftPinkyFinger（左小指）
Neck（脖）
    Head（头）
        LeftEye（左眼）
        RightEye（右眼）
        多个面部关节
RightThigh（右大腿）
    RightKnee（右膝）
        RightAnkle（右脚踝）
LeftThigh（左大腿）
    LeftKnee（左膝）
        LeftAnkle（左脚踝）
```

我们通常会把每个关节赋予0～$N-1$的索引。因为每个关节有一个且仅一个父关节，只要在每个关节储存其父关节的索引，即能表示整个骨骼层阶结构。由于根关节并无父，其父索引通常会设为无效的索引，例如−1。

11.2.2 在内存中表示骨骼

骨骼通常由一个细小的顶层数据结构表示，该结构含有关节数组。关节的储存次序通常会保证每个子关节都位于其父关节之后。这也意味着，数组中首个关节总是骨骼的根关节。

在动画数据结构中，通常会使用**关节索引**（joint index）引用关节。例如，子关节通常以索引引用其父关节。同样地，在蒙皮三角形网格中，每个顶点使用索引引用其绑定关节。使用索引引用关节，无论在储存空间上（关节索引通常用8位整数）或查找引用关节的时间上（索引可直接存取数组中所需的关节），都比使用关节名字高效得多。

每个关节的数据结构通常含以下信息。

- 关节**名字**，可以是字符串或32位字符串散列标识符。
- 骨骼中其**父**节点的**索引**。
- 关节的**绑定姿势之逆变换**（inverse bind pose transform）。关节的绑定姿势是指蒙皮网

格顶点绑定至骨骼时，关节的位置、定向及缩放。我们通常会储存此变换之逆矩阵，其原因会在稍后深入探讨。

典形的骨骼数据结构可能是这样的：

```
struct Joint
{
    Matrix4x3   m_invBindPose;   // 绑定姿势之逆变换
    const char* m_name;          // 人类可读的关节名字
    U8          m_iParent;       // 父索引，或0xFF代表根关节
};

struct Skeleton
{
    U32         m_jointCount;    // 关节数目
    Joint*      m_aJoint;        // 关节数组
};
```

11.3 姿势

无论采用哪种制作动画的技术，赛璐璐、刚性层阶、蒙皮 / 骨骼，每个动画都是随时间推移的。通过把角色身体摆出一连串离散、静止的**姿势**（pose），并以通常30或60个**姿势每秒**的速率显示这些姿势，就能令角色产生动感。（实际上，如第11.4.1.1节所提及，我们会为相邻的姿势**插值**，而非逐个姿势显示。）在骨骼动画中，骨骼的姿势直接控制网格顶点，而且摆姿势是动画师为角色带来生命气息的主要工具。因此，很明显，要为骨骼加入动画之前，先要了解如何为骨骼摆姿势。

把关节任意旋转、平移，甚至缩放，就能为骨骼摆出各种姿势。一个关节的**姿势**定义为关节相对某参考系（frame of reference）的位置、定向和缩放。关节的姿势通常以4×4或4×3矩阵表示，或表示为SQT数据结构（缩放 / scale、四元数旋转 / quaternion及矢量平移 / translation）。骨骼的姿势仅仅是其所有关节的姿势之集合，并通常简单地以SQT数组表示。

11.3.1 绑定姿势

图11.6显示了一个骨骼的两个不同姿势。左图是一个特别的姿势，称为**绑定姿势**（bind pose），有时候叫作**参考姿势**（reference pose）或**放松姿势**（rest pose）。这是三维网格绑定至骨骼之前的姿势，因而得名。换句话说，这就是把网格当作正常、没有蒙皮、完全不涉

及骨骼的三角形网格来渲染的姿势。绑定姿势又叫作**T姿势**（T-pose），这是由于角色通常会站着，双腿稍分开，并把双臂向左右伸直，形成T字形。特别选择此姿势，是因为此姿势中的四肢远离身体，较容易把顶点绑定至关节。

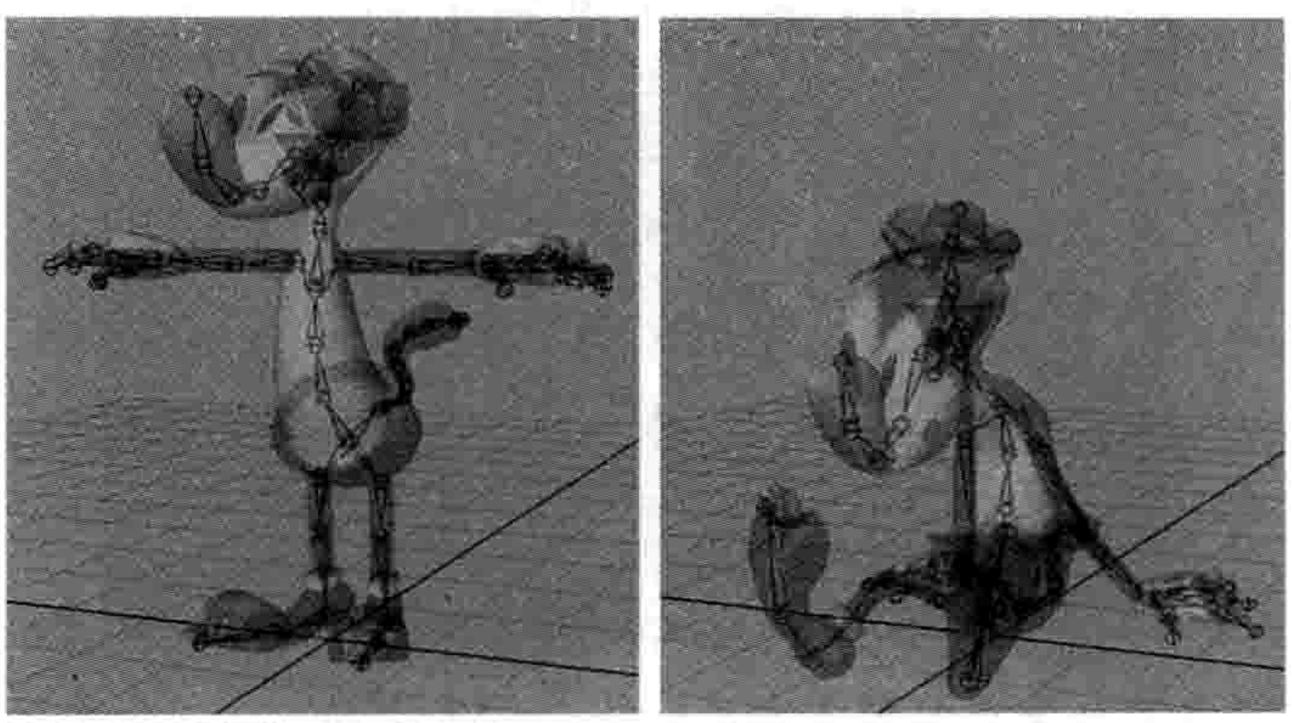


图 11.6: 同一骨骼的两个不同姿势。左图是一个特别姿势，称为**绑定姿势**。

11.3.2 局部姿势

关节姿势最常见是相对于**父**关节来指定的。相对父关节的姿势能令关节自然地移动。例如，若旋转肩关节时，不改动肘、腕及手指相对父的姿势，那么如我们所料，整条手臂就会以肩关节为轴刚性地旋转。我们有时候用**局部姿势**（local pose）描述相对父的姿势。局部姿势几乎都储存为SQT格式，其原因将在稍后谈及动画混合时解释。

在图形表达上，许多三维制作软件，如Maya，会把关节表示为小球。然而，关节含旋转及缩放，不仅限于平移，所以此可视化方式或会有点误导成分。事实上，每个关节定义了一个坐标空间，原理上无异于其他我们曾遇到的空间（如模型空间、世界空间、观察空间）。因此，最好把关节显示为一组笛卡儿坐标轴。Maya提供了一个选项显示关节的局部坐标轴，如图11.7所示。

数学上，关节姿势就是一个**仿射变换**（affine transformation）。第j个关节可表示为4×4仿射变换矩阵$\mathbf{P}_j$，此矩阵由一个平移矢量$\mathbf{T}_j$、3×3对角缩放矩阵$\mathbf{S}_j$，及3×3旋转矩阵$\mathbf{R}_j$所构成。整个骨骼的姿势$\mathbf{P}^{\text{skel}}$可写成所有姿势$\mathbf{P}_j$的集合，当中j的范围是$0 \sim N-1$：

$$\mathbf{P}_j = \begin{bmatrix} \mathbf{S}_j\mathbf{R}_j & \mathbf{0} \\ \mathbf{T}_j & 1 \end{bmatrix}$$

$$\mathbf{P}^{\text{skel}} = \{\mathbf{P}_j\}\,|_{j=0}^{N-1}$$

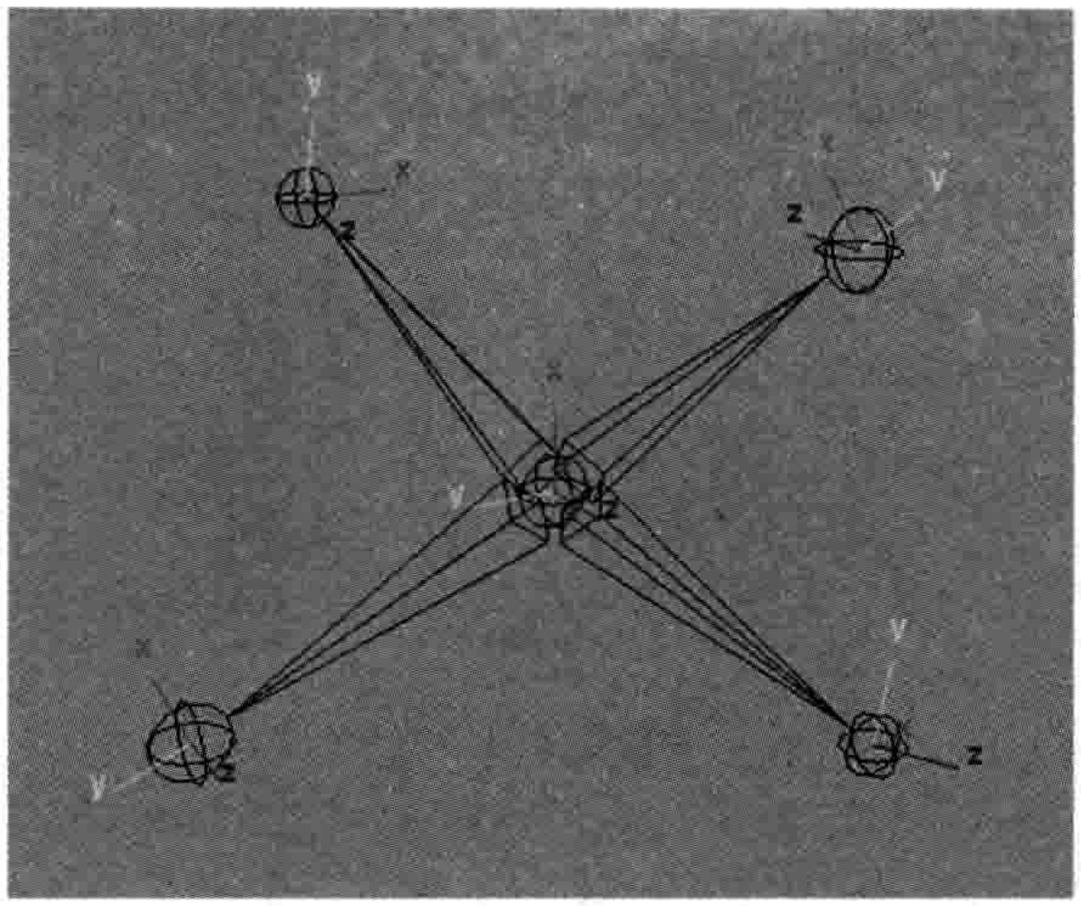


图 11.7: 骨骼层阶中的每个关节都各定义一组局部坐标轴。这组坐标轴称为关节空间。

11.3.2.1 关节缩放

有些游戏引擎不容许关节缩放，那么就会忽略$\mathbf{S}_j$，把它假定为单位矩阵。有些引擎会假设，若使用缩放，其必须为**统一**缩放，即3个维度上的缩放都相同。此情况下，缩放能用单个标量s_j表示。有些引擎甚至支持**非统一**缩放，那么缩放可紧凑地表示为3个元素的矢量$\mathbf{s}_j = [s_{jx} \quad s_{jy} \quad s_{jz}]$。矢量$\mathbf{s}_j$的元素对应$3 \times 3$缩放矩阵$\mathbf{S}_j$的3个对角元素，因此**本身**并不是真正的矢量。游戏引擎几乎不会容许切变（shear），因此$\mathbf{S}_j$几乎永不会以3×3缩放 / 切变矩阵表示，虽然**可以**这么做。

在姿势或动画中忽略或限制缩放有许多好处。显然使用较低维度的缩放表示法能节省内存。（使用统一缩放，每动画帧每关节只需储存1个浮点标量；非统一缩放需要3个浮点数；完整的3×3缩放 / 切变矩阵需要9个浮点数。）限制引擎使用统一缩放，还有另一个好处，它能确保包围球不会变换成椭球体（ellipsoid），而使用非统一缩放则会出现此情况。避免了椭球体就能大幅度简化按每关节计算的平截头体剔除及碰撞测试。

11.3.2.2 在内存中表示关节姿势

如前所述，关节姿势通常表示为SQT格式。在C++中，此数据结构可以是以下这样的，注意当中Q为第1个字段以确保正确的对齐及最优的包裹。（你知道为什么吗？）

```
struct JointPose
{
    Quaternion  m_rot;        // Q
```

```
    Vector3     m_trans;    // T
    F32         m_scale;    // S (仅为统一缩放)
};
```

若容许非统一缩放，我们就会这么定义关节姿势：

```
struct JointPose
{
    Quaternion  m_rot;      // Q
    Vector3     m_trans;    // T
    Vector3     m_scale;    // S
    U8          m_padding[8];
};
```

整个骨骼的局部姿势可表示如下，当中m_aLocalPose数组是动态分配的，该数组刚可容纳匹配骨骼内关节数目的JointPose。

```
struct SkeletonPose
{
    Skeleton*   m_pSkeleton;    // 骨骼 + 关节数量
    JointPose*  m_aLoclPose;    // 多个局部关节姿势
};
```

11.3.2.3 把关节姿势当作基的变更

谨记**局部**关节姿势是相对直属父关节而指定的。任何仿射变换都可想象为把点或矢量从一个坐标系变换至另一坐标系。因此，当把关节姿势变换$\mathbf{P}_j$施于以关节j坐标系表示的点或矢量时，其变换结果是以父关节空间表示的该点或矢量。

如前数章的惯用法，我们会使用下标表示变换的方向。因为关节姿势能把点及矢量从**子**关节的空间（C）变换至其**父**关节的空间（P），我们会把此变换写成$(\mathbf{P}_{\mathrm{C}\to\mathrm{P}})_j$。另一方式是引入一个函数$\mathrm{p}(j)$，它会回传关节$j$的父索引，那么就可以把关节$j$的局部姿势写成$\mathbf{P}_{j\to\mathrm{p}(j)}$。

偶尔我们要以相反方向变换点及矢量，即由**父关节**的空间变换至**子关节**的空间。此变换就是局部关节姿势的逆变换。数学上表示为$\mathbf{P}_{\mathrm{p}(j)\to j} = (\mathbf{P}_{j\to\mathrm{p}(j)})^{-1}$。

11.3.3 全局姿势

有时候，把关节姿势表示为模型空间或世界空间会很方便。这称为**全局姿势**（global pose）。有些引擎用矩阵表示全局姿势，有些引擎则使用SQT格式。

数学上，某关节的模型空间姿势（$j \to \mathrm{M}$），可通过从该关节遍历至根关节时，在每个关节乘上其局部姿势（$j \to \mathrm{p}(j)$）算出。以图11.8的层阶为例，把根关节的父节点定义为模型空间，即$\mathrm{p}(0) \equiv \mathrm{M}$。关节$\mathrm{J}_2$的模型空间姿势便可写成：

$$\mathbf{P}_{2\to\mathrm{M}} = \mathbf{P}_{2\to1}\mathbf{P}_{1\to0}\mathbf{P}_{0\to\mathrm{M}}$$

类似地，关节J_5的模型空间姿势可写成：

$$\mathbf{P}_{5\to\mathrm{M}} = \mathbf{P}_{5\to4}\mathbf{P}_{4\to3}\mathbf{P}_{3\to0}\mathbf{P}_{0\to M}$$

任何关节j的全局姿势（关节至模型空间的变换）可写成：

$$\mathbf{P}_{j\to\mathrm{M}} = \prod_{i=j}^{0} \mathbf{P}_{i\to\dot{\mathrm{p}}(i)} \tag{11.1}$$

当中，每次乘法迭代意味着i变成$\mathrm{p}(i)$（即关节i的父关节），并且$p(0) \equiv M$。

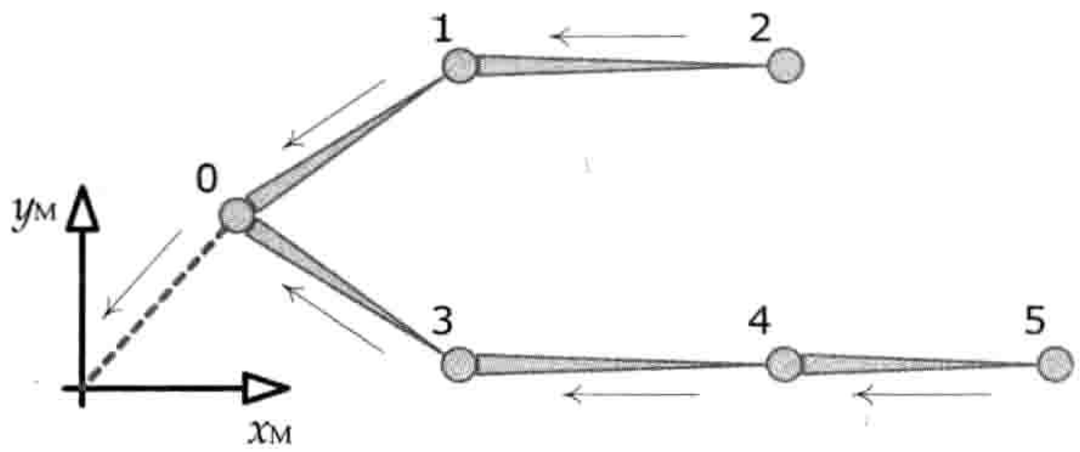


图 11.8: 为计算关节的全局姿势，可从该关节往根关节及模型空间原点遍历，过程中把每个关节的子至父（局部）变换串接起来。

11.3.3.1 在内存中表示全局姿势

我们可以扩展SkeletonPose的数据结构，以包含全局姿势。当中我们再次基于骨骼中的关节数目动态分配m_aGlobalPose数组：

```
struct SkeletonPose
{
    Skeleton*   m_pSkeleton;     // 骨骼 + 关节数量
    JointPose*  m_aLocalPose;    // 多个局部关节姿势
    Matrix44*   m_aGlobalPose;   // 多个全局关节姿势
};
```

11.4 动画片段

在动画电影中，每个场景的方方面面都会先仔细规划，然后才开始制作动画。这些规划包括场景中每个角色和道具的移动，甚至包括摄像机的移动。换句话说，整个场景会以一串很长的、连续的帧来产生动画。当角色在镜头之外时，无须为它们制作动画。

然而，游戏的动画与此不同。游戏是互动体验，所以无人能预料角色会移动至哪里、做些什么。玩家能全权控制其角色，通常也能控制部分摄像机的行为。甚至乎，人类玩家不可预知的行动，也会大大影响由计算机驱动的非玩家角色。因此，游戏的动画几乎都不可能制作成一串很长的、连续的帧。取而代之，游戏角色的移动都必须拆分为大量小粒度的动作。我们称这些个别的动作为**动画片段**（animation clip），有时或简称**动画**（animation）。

每个片段都能令角色表现一个有明确界定的动作。有些片段会设计成循环形式，例如步行周期、跑步周期。其他片段则只会播放一次，例如掷物、绊倒并跌在地上。有些片段会影响角色全身，例如跳跃。其他的则只会影响身体某部分，如挥动右手。一个角色的动作一般会分拆成上千个片段。

唯一例外的情况是，当角色进入游戏中非互动的部分，这些部分称为**游戏内置电影**（in-game cinematics，IGC）、**非交互连续镜头**（noninteractive sequence，NIS）、或**全动视频**（full-motion video，FMV）。非互动序列通常用于交代难于在互动游戏过程中表现的故事情节，而这些序列的制作方法基本上和计算机动画电影相同（虽然非互动序列经常会使用游戏内的资产，如角色网格、骨骼、纹理等）。术语IGC和NIS通常是指用游戏引擎来渲染的非互动序列。FMV则是指预先渲染至MP4、WMV或其他的视频文件类型，然后在运行时由引擎内的全屏电影播放器播放。

这类动画的另一变种是半互动的序列——**快速反应事件**（quick time event，QTE）。在QTE里，玩家必须在非互动序列中的正确时间按键，才能见到成功的动画并继续下去；否则会播放失败的动画，要玩家再来一次，并可能会扣命或带来其他不良后果。

11.4.1 局部时间线

我们可以想象每个动画片段各自有一条局部时间线（local timeline），该时间线通常使用自变量t表示。在片段开始时$t = 0$，在结束时$t = T$，当中T为片段的持续时间。变量t的每个值称为**时间索引**（time index）。图11.9是一个局部时间线的例子。

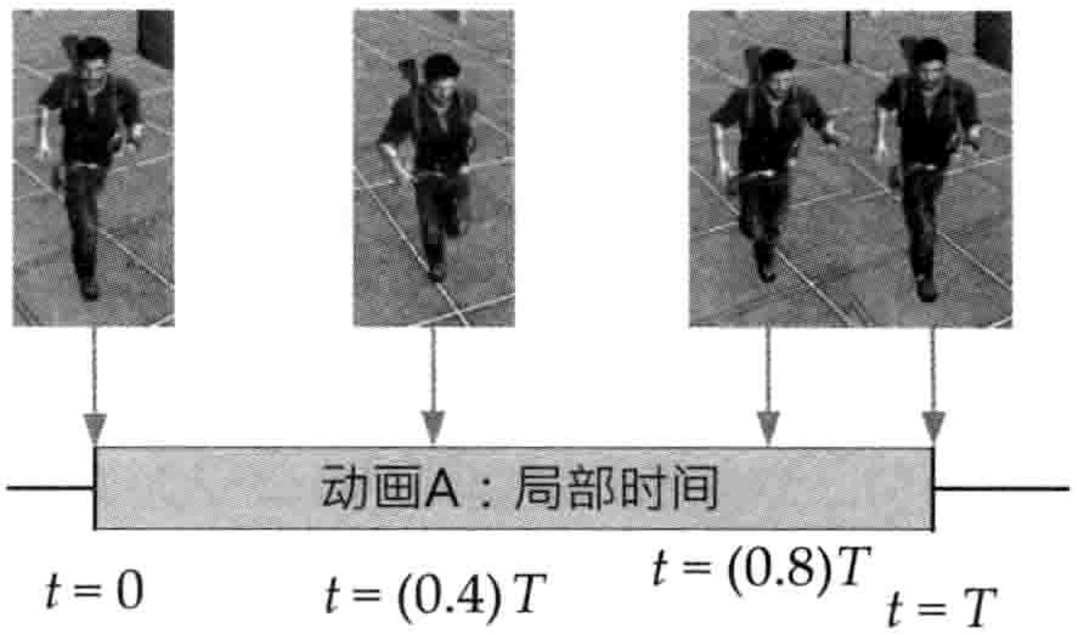


图 11.9: 在一个动画的局部时间线上，展示了位于几个时间索引的姿势。

11.4.1.1 姿势插值及连续时间

我们要意识到，把帧展示给观众的速率，并不一定要等于由动画师所制作的姿势的播放速率。在电影和游戏中，动画师几乎都不会以每秒30或60次设定角色的姿势。取而代之，动画师会在片段中指定的时间点上设定一些重要的姿势，这些姿势称为**关键姿势**（key pose）或**关键帧**（key frame），然后计算机会采用线性或基于曲线的插值计算中间的姿势。图11.10说明这点。

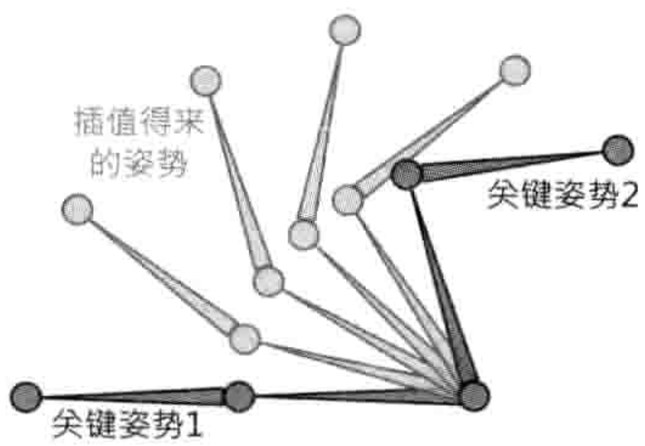


图 11.10: 动画师制作少量的关键姿势，然后引擎用插值法填补其余的姿势。

由于动画引擎能够对姿势**插值**（稍后本章会做详细介绍），我们实际上能在片段间**任何时间**采样，不一定要在整数帧索引上采样。换言之，动画片段的时间线是**连续的**。在计算机动画中，时间变量t是**实数**（浮点数），而非**整数**。

动画电影并不会充分利用动画时间线连续性所带来的好处，因为电影的帧率会锁定为每秒24、30或60帧。例如，在电影中观众只会看见角色在第1、2、3等帧的姿势，永不需要找寻角色第3.7帧的姿势。因此在动画电影中，动画师不需要很在意（若非从没关注）角色在两个整数帧索引之间的样子。

相反，因CPU或GPU的负载，实时游戏的帧率经常有少许变动。而且，有时候会调节

游戏动画的**时间比例**（time scale），使角色的动作显得快于或慢于原来制作动画时的速率。因此在实时游戏中，动画片段几乎**永远不会**在整数帧索引上采样。理论上，若时间比例为1.0，便应在第1、2、3等帧上对片段采样。但实际上，玩家可能会见到第1.1、1.9、3.2等帧。并且若把时间比例设为0.5，玩家可能实际上会见到第1.1、1.4、1.9、2.6、3.2等帧。甚至可使用负值的时间比例播放前后倒转的动画。因此，游戏动画的时间是**连续的**，并**可改变比例的**。

11.4.1.2 时间单位

由于动画的时间线是连续的，最好使用秒作为时间量度单位。若我们定义了帧的持续时间，那么时间也可以使用**帧**作为量度单位。游戏动画中，典型的帧持续时间为1/30s或1/60s。然而，切记不要把时间变量t定义为整数，只算完整的帧。无论选择哪种时间单位，t都应该是实数（浮点数）、定点数或量度子帧（subframe）时间间隔的整数。归根究底，目标是令时间的量度值有足够的分辨率，以计算帧之间的结果或改变动画播放速率。

11.4.1.3 比较帧与采样

遗憾的是，**帧**这个术语在游戏业界中有多个意思，引致许多混淆。有时候一帧是指**一段时间**，如1/30s或1/60s。但在其他语境中，帧又会指**某一时间点**（如我们会说角色的第42帧姿势）。

笔者较喜爱使用术语**采样**（sample）代表某时间点，而保留**帧**一词描述1/30s或1/60s的持续时间。图11.11的例子说明，以每秒30帧制作的1s动画中，含有31个**采样**，持续时间是30**帧**。“采样”一词源自信号处理。一个时间上连续的信号（即一个函数$f(t)$），可转换为一组时间上均匀相隔的离散数据点。关于采样的更多信息可参阅维基百科[3]。

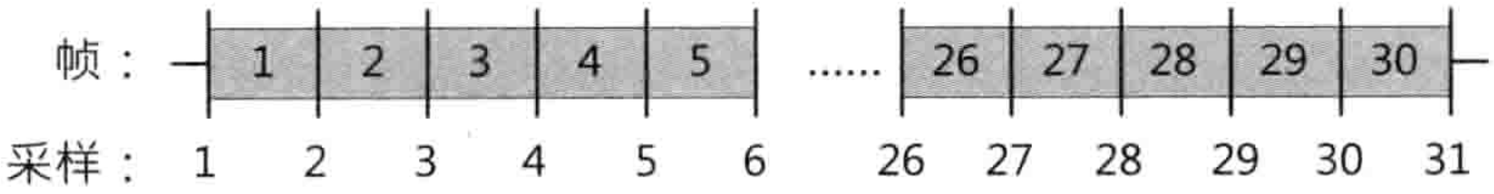


图 11.11: 1s的动画以每秒30帧采样，其时长是30帧，并含有31个采样。

[3] http://en.wikipedia.org/wiki/Sampling_(signal_processing)

11.4.1.4 帧、采样及循环片段

当把动画片段设计为不断重复播放时，我们称之为**循环动画**（looping animation）。假设我们把一个1s（30帧/31采样）的动画复制两份，前后连接，然后如图11.12所示，把首个片段的第31个采样和第2个片段的第1个采样重叠。要令片段好好地循环，我们会发现，片段最后的角色姿势必须完全和最初的姿势匹配。那么也即意味着，循环片段的最后一个采样是冗余的（参看例子中的第31个采样）。因此许多游戏引擎会略去循环片段的最后一个采样。

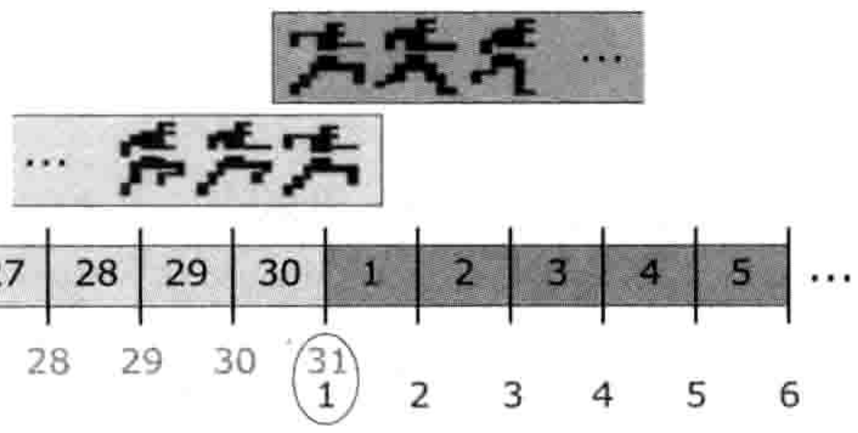


图 11.12: 循环片段的最后一个采样与第一个采样是重合的，所以它是冗余的。

以上的分析可带出用于所有动画片段的采样数目和帧数目的规则。

- 若片段是**非循环的**，N个帧的动画有$N+1$个独一无二的采样。
- 若片段是**循环的**，那么最后一个采样是冗余的，因此N个帧的动画有N个独一无二的采样。

11.4.1.5 归一化时间（相位）

有时候，使用归一化的时间单位u是比较方便的。在这种时间单位中，无论动画的持续时间T是多长，$u=0$代表动画的开始，$u=1$代表结束。我们有时候称归一化的时间为动画的**相位**（phase），因为当动画在循环时，u有如正弦波的相位。图11.13说明了这种时间单位。

当要同步两个或以上的动画片段，而它们的持续时间又不相同，归一化时间就很适用。例如，假设我们希望能圆滑地把2s（60帧）的跑步周期淡入 / 淡出至一个3s（90帧）的步行周期。要令淡入 / 淡出的过程自然，我们要确保两个动画能一直维持同步，使两个片段中的步伐是一致的。简单的解决方法是把步行片段的归一化起始时间u_{walk}与跑步片段的归一化时间u_{run}匹配。然后我们使两个片段以相同的归一化速率推进，使两个片段保持同步。此做法较使用绝对时间索引t_{walk}和t_{run}容易实现，并较难出错。

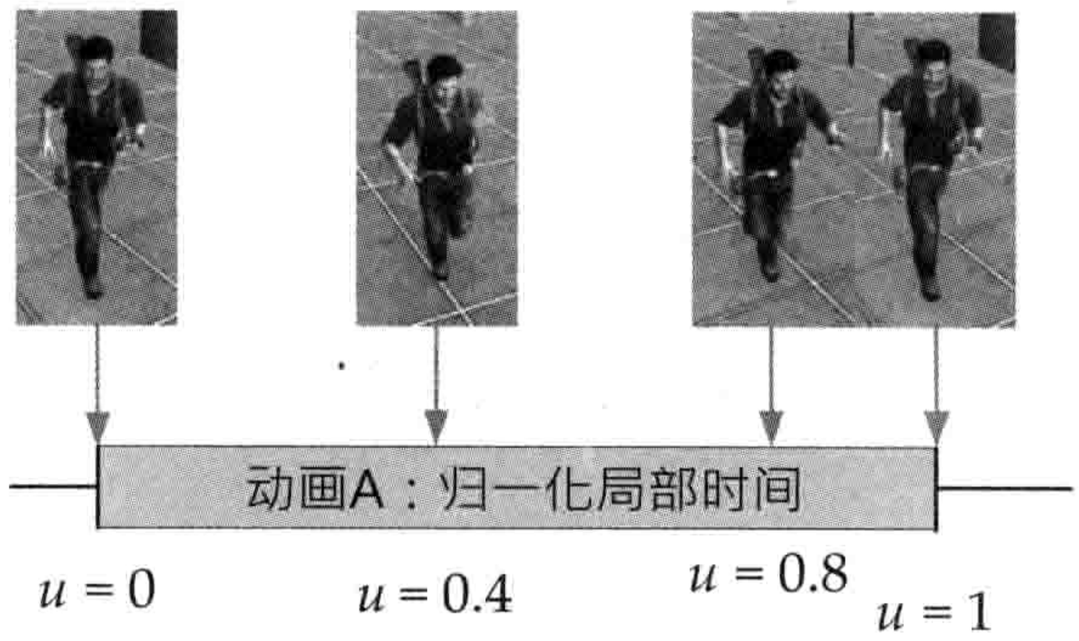


图 11.13: 一个动画片段的归一化时间单位。

11.4.2 全局时间线

正如每个动画片段都有一个局部时间线（其时钟在动画开始时为0），游戏里每个角色都有一个全局时间线（global timeline，其时钟在角色诞生于游戏世界时启动，或是在关卡或整个游戏开始时启动）。本书采用时间变量τ表示全局时间，与局部时间t区分。

我们可以把**播放**动画简单想象成把片段的局部时间**映射**至角色的全局时间。例如，图11.14中，动画片段A从全局时间$\tau_{\text{start}} = 102\text{s}$开始播放。

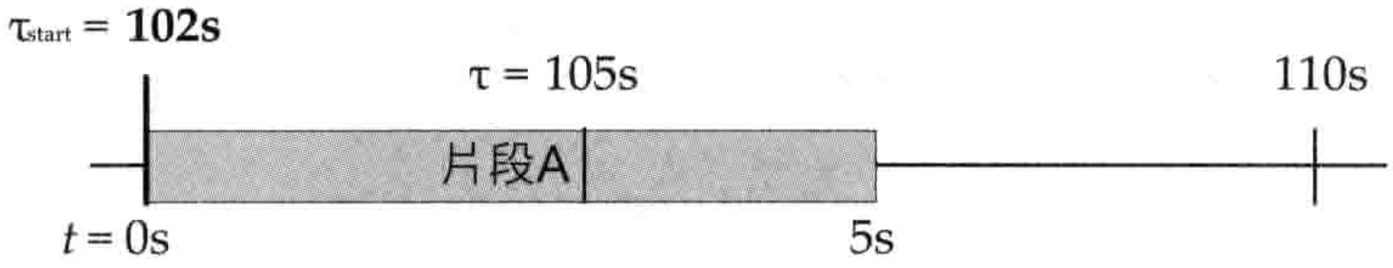


图 11.14: 在全局时间线的102s开始播放动画片段A。

如前所述，播放循环动画就好像把片段复制无限次，并前后相连至全局时间线。我们也可把动画循环有限次数，那么即是把片段复制有限次，并置于全局时间线，如图11.15所示。

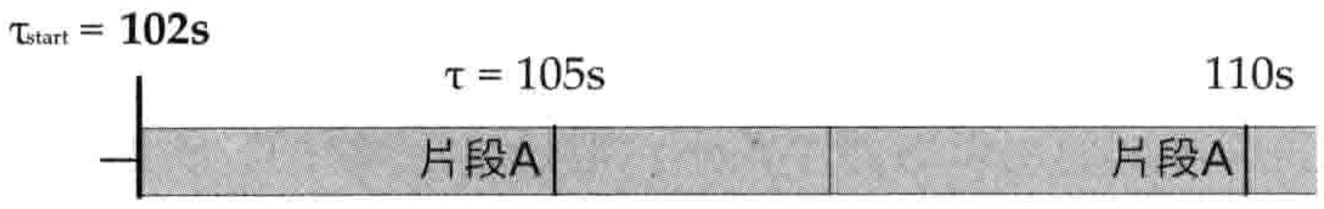


图 11.15: 播放循环动画，等于把片段复制多次，每个紧接一起地置于时间线上。

在片段中调整**时间比例**（time scale），可以把片段播放得比原来设定时更快或更慢。要实现此功能，只需要把片段置于全局时间线之时缩放其比例。此功能最自然的表示方式为**播放速率**（playback rate），我们使用变量R代表它。例如，如果动画以两倍速率播放

（$R = 2$），那么当要把局部时间线放置在全局时间线时，我们把该局部时间线缩短至原来正常长度的一半（$1/R = 0.5$），如图11.16所示。

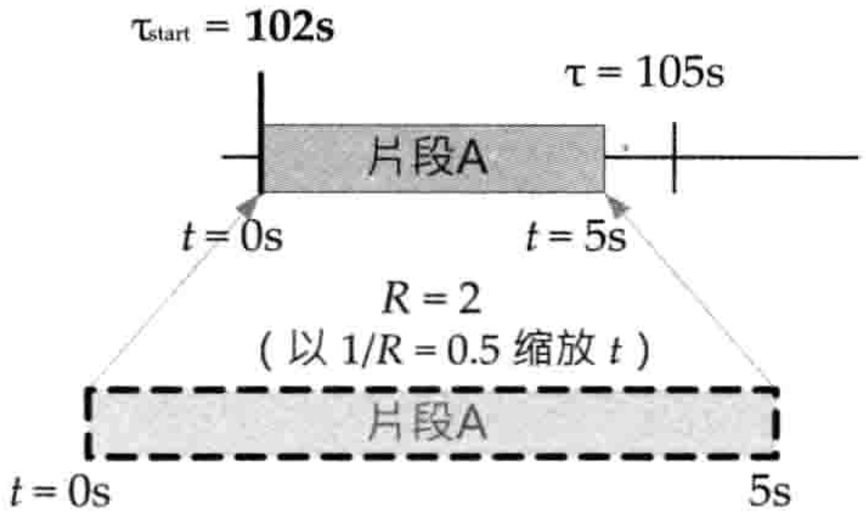


图 11.16: 以两倍速度播放动画，等于把动画的局部时间线的比例缩短为原来的一半。

要把片段倒转播放，可把时间比例设为−1，如图11.17所示。

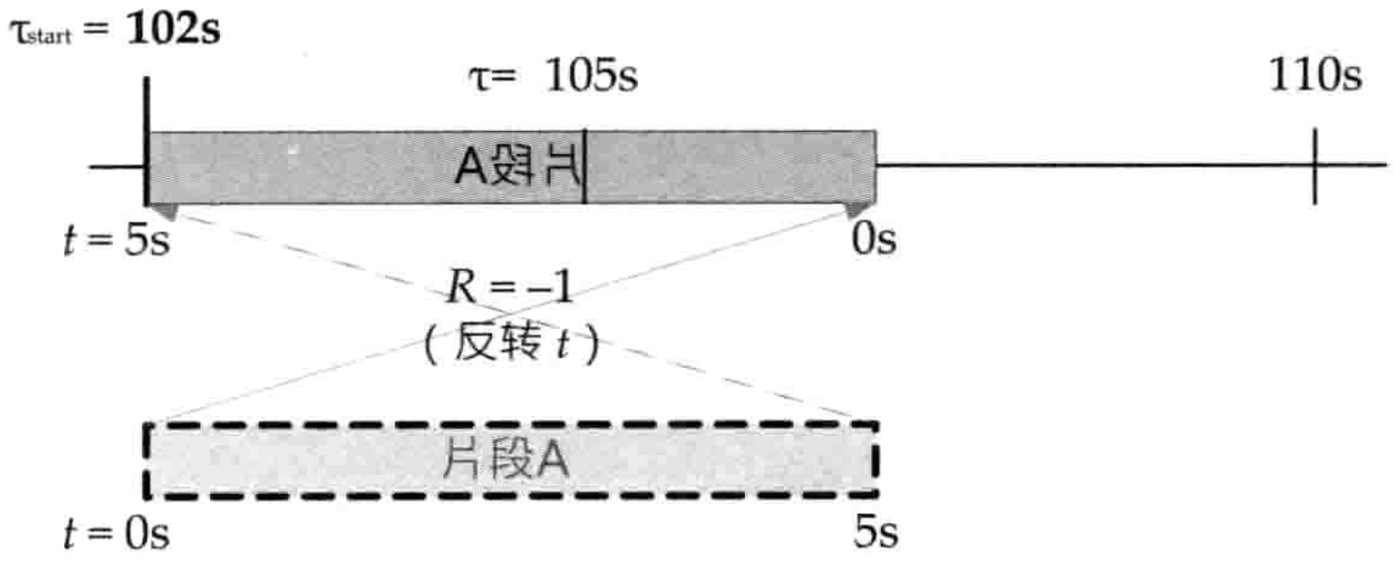


图 11.17: 倒转播放动画等于采用−1的时间缩放比例。

要把动画片段映射至全局时间线，我们需要以下有关片段的几个信息。

- 其全局起始时间τ_{start}
- 其播放速率R
- 其持续时间T
- 循环次数N

有了以上的信息，我们可以使用以下两个关系，把任何全局时间τ映射至对应的局部时间t，及反方向的映射：

$$t = R(\tau - \tau_{\text{start}}) \tag{11.2}$$

$$\tau = \tau_{\text{start}} + \frac{1}{R}t$$

若动画是非循环的（$N = 1$），那么在使用t为片段采样一个姿势之前，我们应把t裁剪至

合法范围$[0, T]$：

$$t = \text{clamp}\left[R(\tau - \tau_{\text{start}})\right]\Big|_0^T$$

若动画不停循环（$N = \infty$），那么可以把t除以持续时间T，得出的**余数**便是合法范围。实现时可使用**模除**（modulo）运算（mod或C/C++中的%[4]）如下：

$$t = \left(R(\tau - \tau_{\text{start}})\right) \bmod T$$

若动画片段循环有限的次数（$1 < N < \infty$），那么我们必须先把时间t裁切至$[0, NT]$的范围，然后再把结果模除以T，以把t变成能对片段采样的合法范围：

$$t = \left(\text{clamp}\left[R(\tau - \tau_{\text{start}})\right]\Big|_0^{NT}\right) \bmod T$$

多数游戏引擎都会直接使用局部动画时间线，而不直接使用全局时间线。然而，直接使用全局时间线也能带来一些极其有用的好处。例如，这会令同步动画变得直观、容易。

11.4.3 比较局部和全局时钟

动画系统必须记录每个正在播放的动画的时间索引。我们有两种记录时间索引的方法。

- **局部时钟**：在此方法中，每个片段都有其局部时钟，时钟通常用秒、帧或归一化时间为单位（后者常称为动画的**相位**），以浮点小数形式储存。片段开始播放时，通常局部时间t被设为0。随时间推移，我们把每个片段各自的局部时钟向前推进。若片段有非正常的播放速率R，局部时钟在推进时要以R缩放。
- **全局时钟**：在此方法中，角色含有全局时钟，时钟通常以秒为单位。每个片段记录其开始播放时的全局时间τ_{start}。片段的局部时钟由公式(11.2)**计算**出来，而非直接储存在片段之中。

局部时钟方法的优点在于简单，并且是设计动画系统时最显然的选择。然而，全局时钟方法有其过人之处，特别适用于同步动画，无论是单个角色本身的同步或是场景中多个角色的同步。

11.4.3.1 用局部时钟同步动画

使用局部时钟方法时，我们通常会把片段局部时间的原点（$t = 0$）定义为片段开始播

[4]译注：前文说明了时间变量须为实数，所以这里应使用fmod()或fmodf()函数。C/C++的%运算符只能用于整数。

放那一刻。因此，要同步两个或以上的片段时，必须于完全相同的游戏时间播放它们。虽然这好像很简单，然而，若播放动画的命令是来自多个不同的引擎子系统的，这就会变得棘手。

例如，我们要同步玩家角色的出拳动画与NPC的相应受击反应动画。问题在于，玩家的出拳动画是由玩家子系统在侦测到按下手柄按钮后做出的反应，而NPC的受击动画是由人工智能（AI）子系统播放的。若在游戏循环中，AI子系统的代码在玩家子系统代码**之前**执行，那么玩家出拳和NPC的反应就会有1帧的延迟。若玩家子系统的代码在AI子系统的代码之前执行，当NPC打击玩家角色时，相反的问题也会出现。若两个子系统之间的通信是使用消息传送（事件）系统，更会有额外的延迟（详见14.7节）。图11.18说明了此问题。

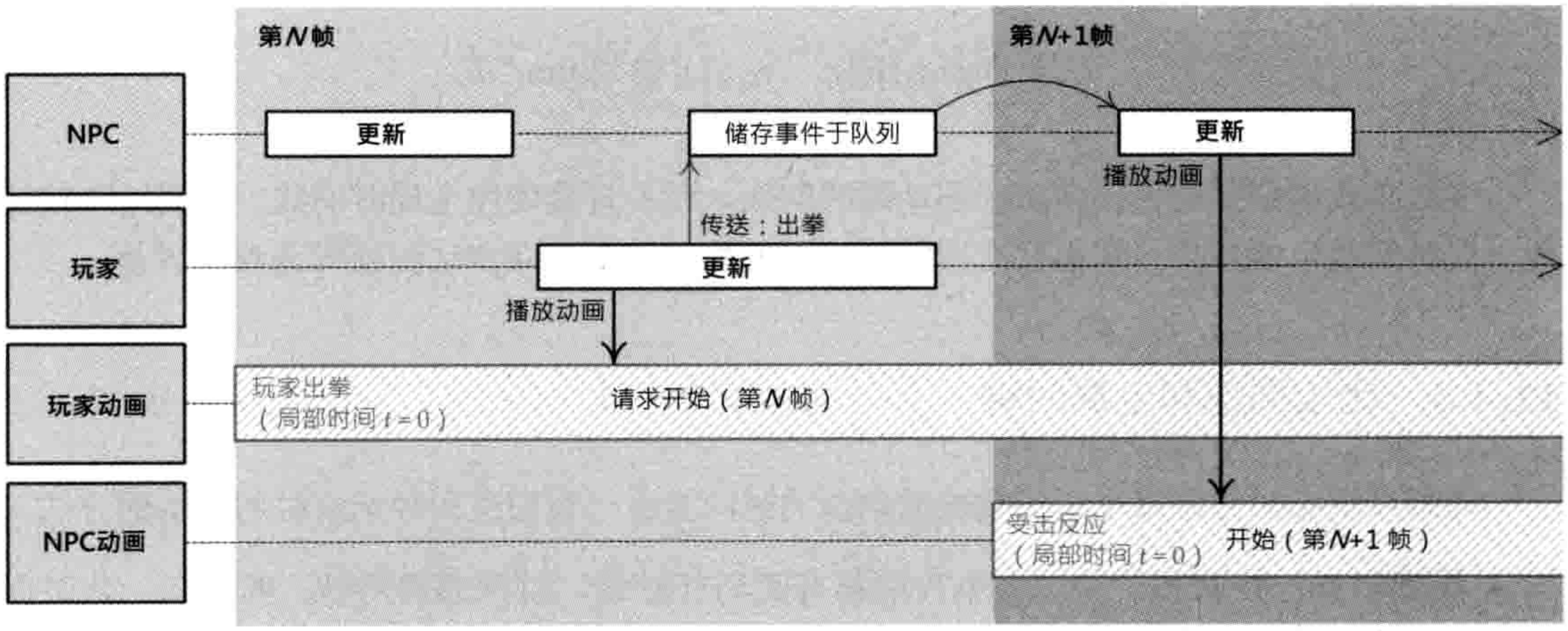


图 11.18: 使用局部时钟来播放动画，各游戏性系统的执行次序就会造成动画同步问题。

```
void GameLoop()
{
    while (!quit)
    {
        // 初步更新……
        UpdateAllNpcs();      // 对上帧的出拳事件做出反应
        // 其他更新……
        UpdatePlayer();       // 按下出拳按钮，开始出拳动画，
                              // 并发送事件给NPC回应
        // 更多更新
    }
}
```

11.4.3.2 用全局时钟同步动画

全局时钟方法有助于解决许多同步问题，因为依照定义，所有片段的时间线都有共同的原点（$\tau = 0$）。在两个或以上的动画中，若其全局的开始时间在数值上相同，那么这些片段在开始时是完全同步的。若片段的播放速率都相同，片段就会一直同步，不会慢慢互相偏离。从此，在**何时**执行播放动画的代码就不成问题。就算玩家出拳1帧后，AI代码才播放受击反应，仍然可以简单地令两个动画同步，只要把这两个动画的全局开始时间匹配就可以了。图11.19说明了此解决方法。

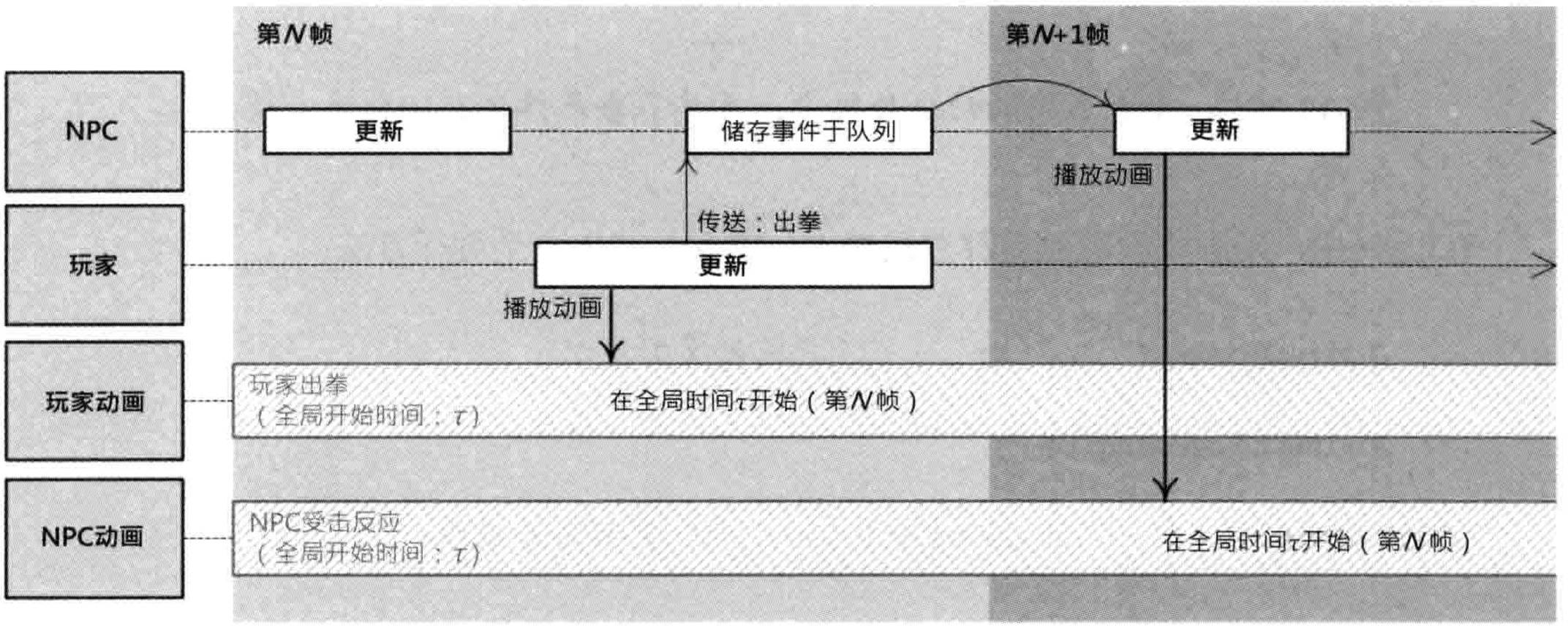


图 11.19: 使用全部时钟可以缓和动画同步问题。

当然，我们需要确保两个角色的全局时间互相匹配，但这是简单的工作。我们可以根据两个角色全局时钟之差，调整两个全局开始时间；或是可以简单地令游戏中所有角色共享一个相同的主时钟。

11.4.4 简单的动画数据格式

一般来说，动画数据是从Maya场景文件中，通过离散地以每秒30或60个骨骼姿势采样的速率采样而得。一个采样由骨骼中每个关节的完整姿势所组成。这些关节姿势通常储存为SQT格式：对于每个关节j，其缩放部分不是一个标量S_j，就是一个三维矢量$\mathbf{S}_j = [S_{jx} \quad S_{jy} \quad S_{jz}]$；旋转部分当然是一个四元数$Q_j = [Q_{jx} \quad Q_{jy} \quad Q_{jz} \quad Q_{jw}]$；而平移是三维矢量$\mathbf{T}_j = [T_{jx} \quad T_{jy} \quad T_{jz}]$。我们有时候称一个动画由每关节多至10个通道（channel）所组成，实际是指$\mathbf{S}_j$、Q_j、$\mathbf{T}_j$的10个分量。图11.20展示了这个情况。

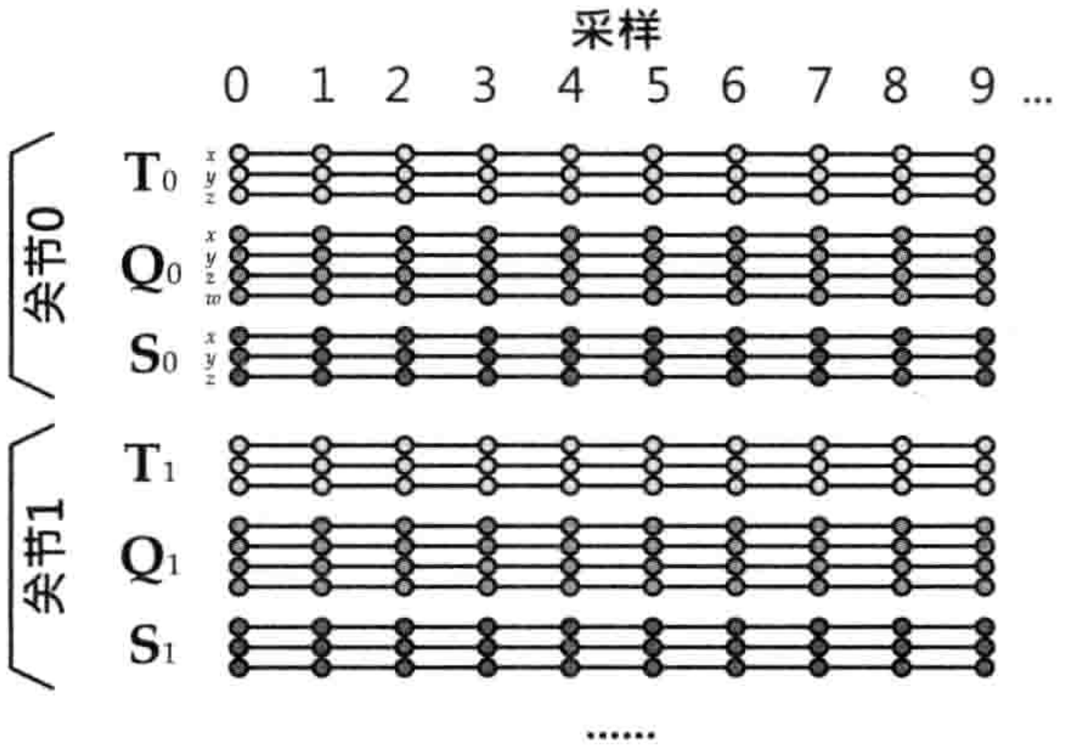


图 11.20: 一个未压缩的动画片段中，每关节每采样含有10个浮点数据。

在C++中，动画片段可以多种方式表示。以下是其中一个可行方法：

```
struct JointPose { ... };    // 之前定义的SQT

struct AnimationSample
{
    JointPose*        m_aJointPose; // 关节姿势数组
};

struct AnimationClip
{
    Skeleton*         m_pSkeleton;
    F32               m_framesPerSecond;
    U32               m_frameCount;
    AnimationSample*  m_aSamples; // 采样数组
    bool              m_isLooping;
};
```

每个动画片段是为特定骨骼而设计的，通常不会用于其他骨骼。因此，以上的例子中，AnimationClip数据结构含有其骨骼之引用m_pSkeleton。（在真实的引擎中，可能会使用独一无二的骨骼标识符，而非Skeleton*指针。在此情况下，引擎必须提供既快速又方便的方法用标识符查找骨骼。）

m_aJointPose的数组长度已假定和骨骼的关节数目相同。而m_aSamples数组的采样数目则是由帧数及该片段是否用于循环所决定的。非循环动画的采样数目是(m_frameCount + 1)。循环动画的最后一个采样等同于第一个采样，所以通常会被略去。在这种情况下，采样数目便会等于m_frameCount。

必须要知道，在真实的游戏引擎中，动画数据不会储存于这般简单的格式中。在11.8节中将会提及，动画数据通常会以多种方式**压缩**，以节省内存。

11.4.4.1 动画重定目标

上文说到，一个动画通常只兼容于特定骨骼。若多个骨骼是很近似的，那么也可打破此规则。例如，若一组骨骼基本上是相同的，除了有些骨骼含有不会影响主要层次结构的子关节，那么为其中的一个骨骼所设计的动画，应能用于其余的骨骼。唯一的要求就是，引擎把动画播放于某骨骼时，需要忽略动画中未能与骨骼匹配的关节。

还有更先进的技术，可把为一个骨骼而设计的动画，重定目标（retarget）至不同的骨骼。这是一个活跃的研究领域，但完整的讨论已超出本书范围。读者可参考两篇论文[5,6]。

11.4.5 连续的通道函数

动画片段中的采样其实就是用来定义随时间改变的连续函数。读者可以把动画片段想象为每关节有10个标量值的函数，或是每关节有两个矢量值的函数加上四元数矢量值的函数。理论上，这些**通道函数**（channel function）在整个片段的时间线上是圆滑并连续的，如图11.21所示（除了故意编辑成不连续的，例如镜头切换）。然而在实践中，许多游戏引擎只会在采样间进行**线性**插值，那么实际上用到的是原来连续函数的**分段线性逼近**（piecewise linear approximation），如图11.22所示。

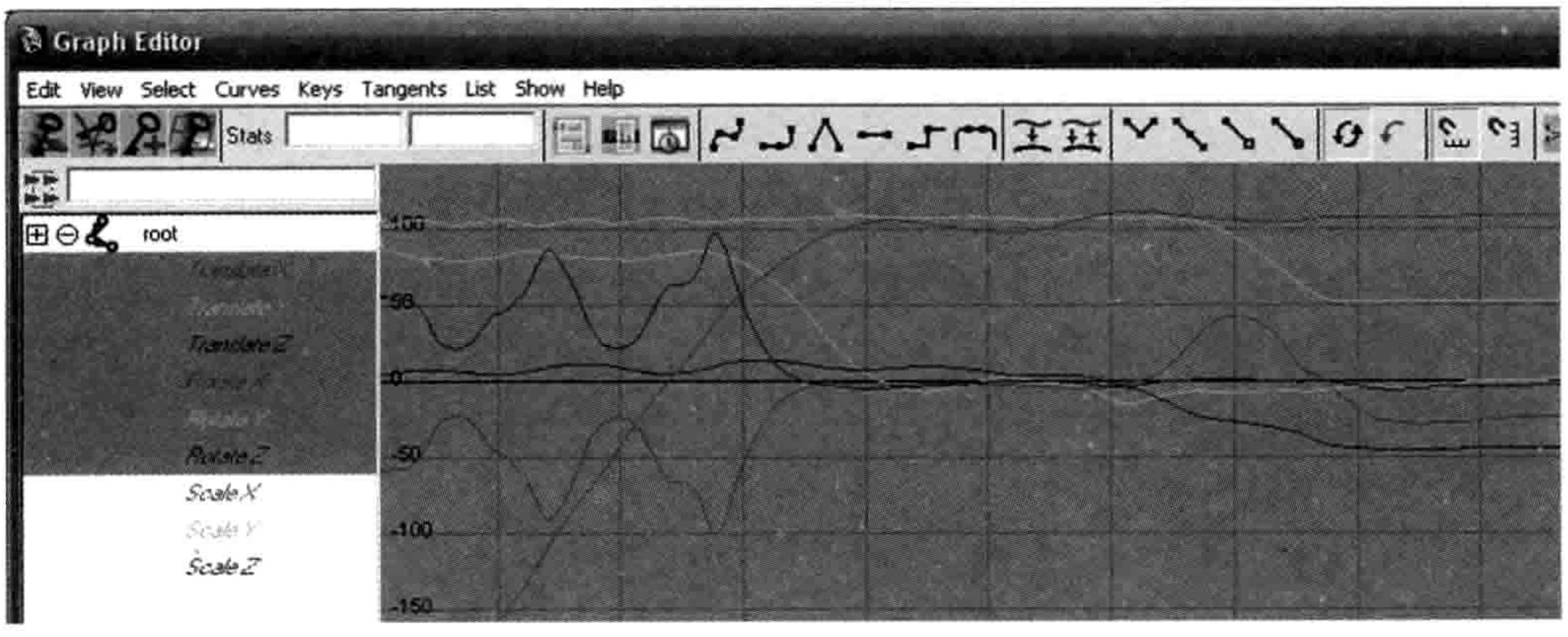


图 11.21: 动画片段中的采样定义了时间上的连续函数。

[5]http://portal.acm.org/citation.cfm?id=1450621

[6]http://chrishecker.com/Real-time_Motion_Retargeting_to_Highly_Varied_User-Created_Morphologies

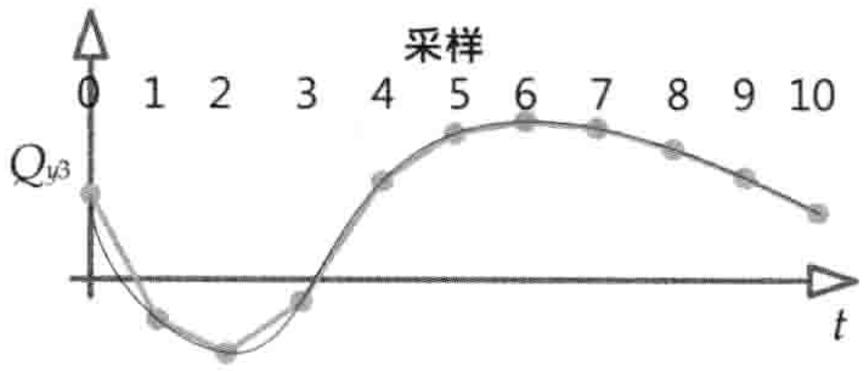


图 11.22: 为通道函数插值时，许多游戏引擎使用分段线性逼近。

11.4.6 元通道

许多游戏容许在动画中加入额外的“元通道（metachannel）”数据。这些通道可以把游戏专用的信息编码，同时能和动画同步，而又无须把这些信息以骨骼姿势储存。

较常见的一种特殊通道是在多个时间点上储存**事件触发器**（event trigger），如图11.23所示。当动画的局部时间索引经过这些触发器时，触发器的**事件**便会送交游戏引擎，引擎可按需处理这些事件。（第14章会详细讨论事件。）事件触发器常用于记录在动画中哪些时间点要播放音效或粒子效果。例如，当左或右脚接触地面时，便可以播放一个脚步声及一个“尘雾”粒子效果。

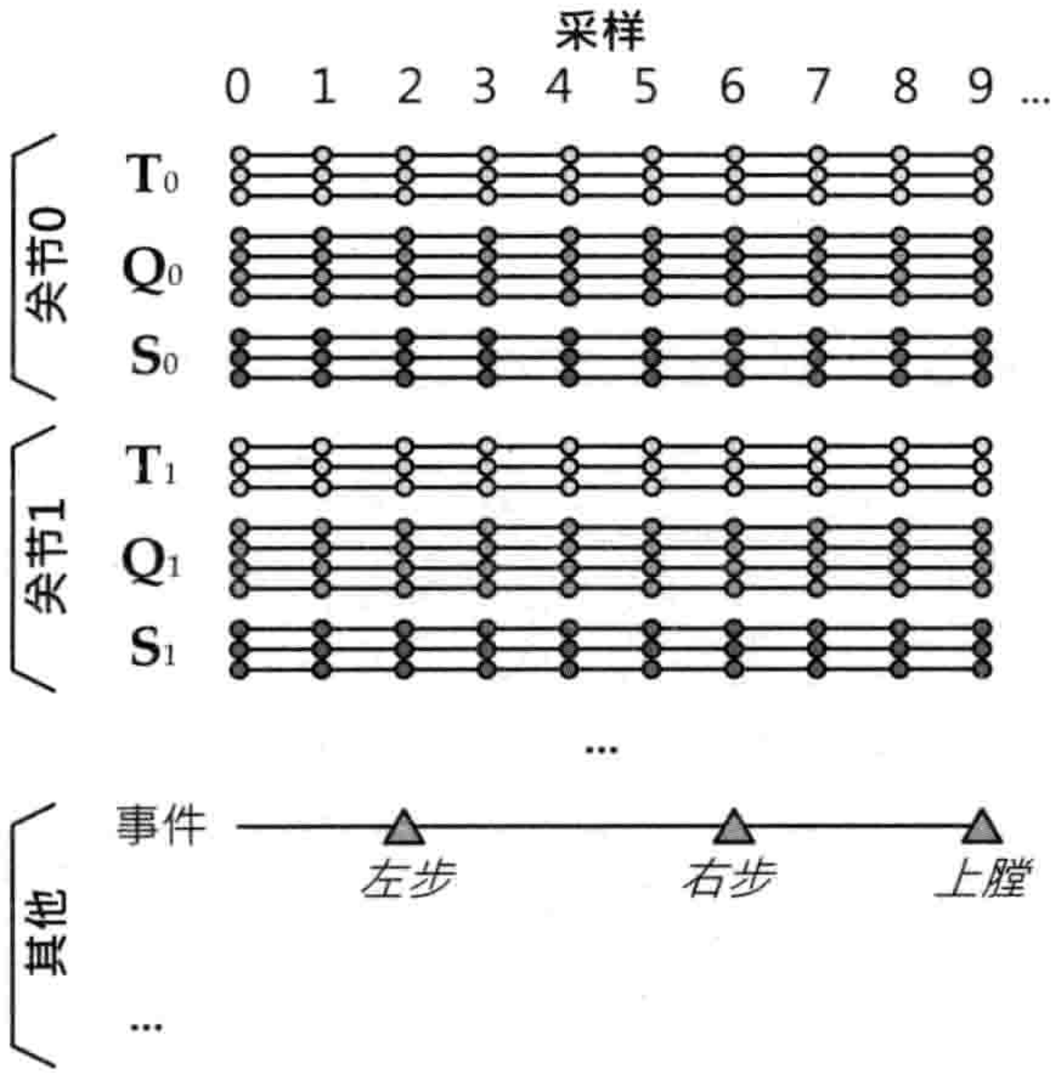


图 11.23: 可以在动画片段中加入特殊的事件触发器通道，用来同步音效、粒子效果，以及其他与动画相关的游戏事件。

另一种常见做法是提供一种在Maya中称为定位器（locator）的特殊关节，定位器可以

和骨骼关节一起设置动画。由于定位器和关节一样，仅仅是一个仿射变换，所以这些特殊关节可用于记录游戏中任何物体的位置及定向。

定位器的典型用法是在动画中设置摄像机的位置及角度。在Maya中，可把摄像机绑定至某定位器，然后与角色（们）的关节一起设置动画。把摄像机的定位器导出之后，就能在游戏中播放动画时移动摄像机。摄像机的视野（field of view）及其他摄像机属性也可设置动画，其数据储存于一个或以上的**浮点通道**（floating-point channel）。

以下是其他非关节动画通道的例子。

- 纹理坐标滚动。
- 纹理动画（此仍纹理坐标滚动的一种特例。多个动画帧会排列在纹理中，然后每迭代滚动一整个帧，以达到动画效果）。
- 含动画的材质参数（颜色、镜面程度、透明度等）。
- 含动画的光源参数（半径、圆锥角度、强度、颜色等）。
- 其他随时间改变，并以某形式和动画同步的参数。

11.5 蒙皮及生成矩阵调色板

我们已了解如何用旋转、平移及缩放设置骨骼的姿势，也知道任何骨骼姿势都可以用一组局部（$\mathbf{P}_{j\to p(j)}$）或全局（$\mathbf{P}_{j\to M}$）关节姿势变换表示。之后，我们会探讨把三维网格顶点联系至骨骼的过程。此过程称为**蒙皮**（skinning）。

11.5.1 每顶点的蒙皮信息

蒙皮用的网格是通过其顶点系上骨骼的。每个顶点可**绑定**（bind）至一个或多个关节。若某顶点只绑定至一个关节，它就会完全跟随该关节移动。若绑定至多个关节，该顶点的位置就等于把它逐一绑定至个别关节后的位置，再取其**加权平均**。

要把网格蒙皮至骨骼，三维建模师必须替每个顶点提供以下的额外信息：

- 该顶点要绑定到的（一个或多个）关节**索引**。
- 对于每个绑定的关节，提供一个**权重因子**（weighting factor），以表示该关节对最终顶点位置的影响力。

如同计算其他加权平均时的习惯，每个顶点的权重因子之和为1。

通常游戏引擎会限制每个顶点能绑定的关节数目。典型的限制为每顶点4个关节，原因如下。首先，4个8位关节索引能方便地包裹为一个32位字。此外，每顶点使用2个、3个及4个关节所产生的质量很容易区分，但多数人并不能分辨出每顶点4个关节以上的质量差别。

因为关节权重的和必须为1，所以最后一个权重可以略去，也通常会被略去。（该权重可以在运行时用$w_3 = 1 - (w_0 + w_1 + w_2)$计算出来。）因此，典型的蒙皮顶点数据结构可能如下：

```
struct SkinnedVertex
{
    float m_position[3];     // (Px, Py, Pz)
    float m_normal[3];       // (Nx, Ny, Nz)
    float m_u, m_v;          // 纹理坐标 (u, v)
    U8    m_jointIndex[4];   // 关节索引
    float m_jointWeight[3]; // 关节权重，略去最后一个
};
```

11.5.2 蒙皮的数学

蒙皮网格的顶点会追随其绑定的关节而移动。要用数学实践此行为，我们需要求一个矩阵，该矩阵能把网格顶点从原来位置（绑定姿势）变换至骨骼的当前姿势。我们称此矩阵为**蒙皮矩阵**（skinning matrix）。

如同所有网格顶点，蒙皮顶点的位置也是在模型空间定义的。无论其骨骼是绑定姿势或任何其他姿势亦然。所以，我们所求的矩阵会把顶点从绑定姿势的模型空间变换至当前姿势的模型空间。不同于之前所见的矩阵（如模型至世界矩阵），蒙皮矩阵并**非**基变更（change of basis）的变换。蒙皮矩阵把顶点变形至新位置，顶点在变换前后都在模型空间。

11.5.2.1 单个关节骨骼的例子

我们开始推导蒙皮矩阵的基本方程。由浅入深，我们先使用含单个关节的骨骼。那么，我们会使用两个坐标空间：模型空间（以下标M表示）及唯一关节的关节空间（以下标J表示）。关节的坐标轴最初为绑定姿势（以下标B表示）。在动画的某个时间点上，关节的轴会移至模型空间中另一位置及定向，我们称此为**当前姿势**（以下标C表示）。

现在我们考虑一个蒙皮至这个关节的顶点。在绑定姿势时，该顶点的模型空间位置为$\mathbf{V}_{\mathrm{M}}^{\mathrm{B}}$。蒙皮过程要计算出该顶点在当前姿势的模型空间位置$\mathbf{V}_{\mathrm{M}}^{\mathrm{C}}$，如图11.24所示。

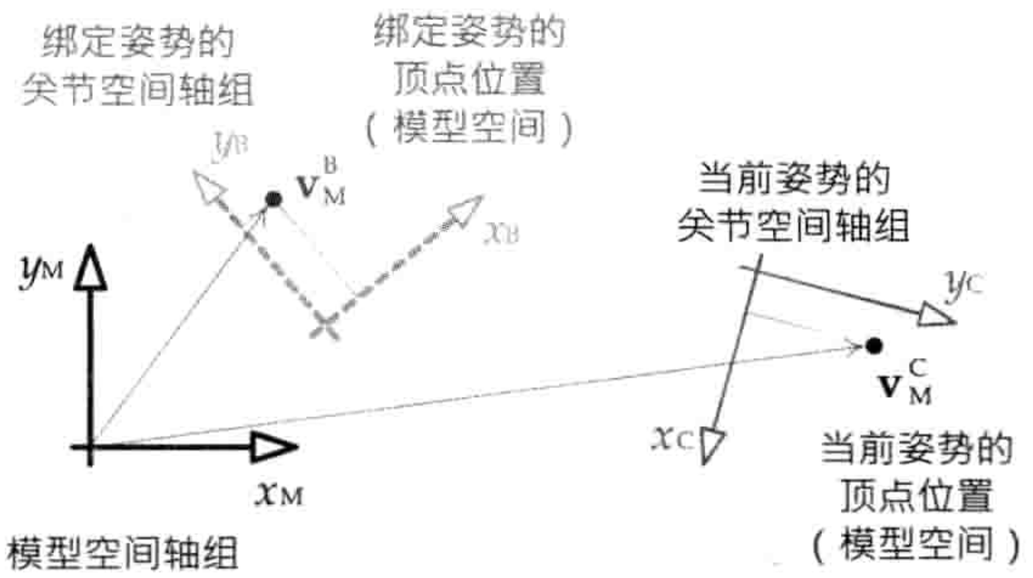


图 11.24: 含单个关节的骨骼，其绑定姿势及当前姿势，以及一个顶点绑定至该关节。

求蒙皮矩阵的“诀窍”在于领会到，顶点绑定至关节的位置时，在**该关节空间**中是**不变的**。因此我们可以把顶点于模型空间的绑定姿势位置转换至关节空间，再把关节移至当前姿势，最后把该顶点转回模型空间。此模型空间至关节空间再返回模型空间的变换过程，其效果就是把顶点从绑定姿势“变形”至当前姿势。

参考图11.25，假设$\mathbf{V}_{\mathrm{M}}^{\mathrm{B}}$在绑定姿势的模型空间坐标是$(4,6)$。我们把此顶点变换至对应的关节空间坐标$\mathbf{v}_j$，在图中大约是$(1,3)$。由于此顶点绑定至该关节，无论该关节怎样移动，此顶点的关节空间坐标**一直**会维持$(1,3)$。当我们把关节设置为希望得到的当前姿势，我们把顶点的坐标转换至模型空间，以$\mathbf{V}_{\mathrm{M}}^{\mathrm{C}}$表示。在图中，此坐标大约是$(18,2)$。因此蒙皮变换把顶点从模型空间的$(4,6)$变形至$(18,2)$，整个过程完全是由于该关节从其绑定姿势移动至图中的当前姿势所驱动的。

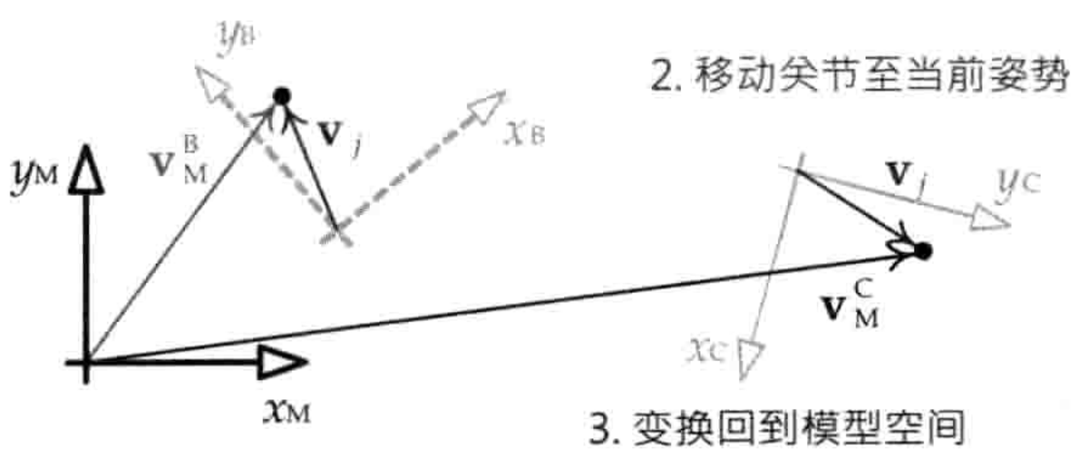


图 11.25: 把顶点位置变换至关节空间，以“追踪”关节的移动。

从数学上查看此问题，我们以矩阵$\mathbf{B}_{j\to\mathrm{M}}$表示关节$j$在模型空间的**绑定姿势**。此矩阵把点或矢量从关节j的空间变换至模型空间。现在，考虑一个以模型空间表示的绑定姿势顶点$\mathbf{V}_{\mathrm{M}}^{\mathrm{B}}$。要把此顶点变换至关节$j$的空间，我们只需简单地乘以绑定姿势矩阵之逆矩阵，

即$\mathbf{B}_{\mathrm{M}\to j}=(\mathbf{B}_{j\to\mathrm{M}})^{-1}$：

$$\mathbf{v}_j=\mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{B}_{\mathrm{M}\to j}=\mathbf{v}_{\mathrm{M}}^{\mathrm{B}}(\mathbf{B}_{j\to\mathrm{M}})^{-1} \tag{11.3}$$

类似地，我们以矩阵$\mathbf{C}_{j\to\mathrm{M}}$表示关节的当前姿势。那么要把$\mathbf{v}_j$从关节空间转回模型空间，我们只需把它乘以当前姿势矩阵：

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}}=\mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}}$$

若使用方程(11.3)展开$\mathbf{v}_j$，就能得出把顶点直接从绑定姿势变换至当前姿势的方程：

$$\begin{aligned}\mathbf{V}_{\mathrm{M}}^{\mathrm{C}}&=\mathbf{v}_j\mathbf{C}_{j\to\mathrm{M}}\\&=\mathbf{v}_{\mathrm{M}}^{\mathrm{B}}(\mathbf{B}_{j\to\mathrm{M}})^{-1}\mathbf{C}_{j\to\mathrm{M}}\\&=\mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{K}_j\end{aligned} \tag{11.4}$$

联合后的矩阵$\mathbf{K}_j=(\mathbf{B}_{j\to\mathrm{M}})^{-1}\mathbf{C}_{j\to\mathrm{M}}$称为**蒙皮矩阵**。

11.5.2.2 扩展至多个关节的骨骼

在以上的例子中，我们只考虑单个关节。然而，我们所推导的方程，其实可施于任何骨骼中的任何关节，因为我们是以全局姿势（即关节至模型空间变换）来进行推导的。要把以上的方程扩展至含多关节的骨骼，我们只需做两个小调整。

1. 我们须确保，矩阵$\mathbf{B}_{j\to\mathrm{M}}$及$\mathbf{C}_{j\to\mathrm{M}}$使用方程(11.1)正确计算。$\mathbf{B}_{j\to\mathrm{M}}$及$\mathbf{C}_{j\to\mathrm{M}}$仅分别等价于该方程中，$\mathbf{P}_{j\to\mathrm{M}}$的绑定姿势及当前姿势。
2. 我们须计算一组蒙皮矩阵$\mathbf{K}_j$，当中每个矩阵对应第j个关节。此数组称为**矩阵调色板**（matrix palette）[7]。当要渲染一个蒙皮网格时，矩阵调色板便要传送至渲染引擎。渲染器会为每个顶点查找调色板中合适的关节蒙皮矩阵，并用该矩阵把顶点从绑定姿势变换至当前姿势。

这里我们再补充一下，假设角色的姿势随时间改变，其当前姿势矩阵$\mathbf{C}_{j\to\mathrm{M}}$便需要每帧更新。然而，绑定姿势逆矩阵在整个游戏中都是常量，因为骨骼的绑定姿势是模型创建时确定下来的。因此，$(\mathbf{B}_{j\to\mathrm{M}})^{-1}$矩阵通常会缓存于骨骼，并不需要在运行时计算。动画引擎通常先计算每个关节的局部姿势$\mathbf{C}_{j\to p(j)}$，然后用方程(11.1)把这些矩阵转换至全局姿势$\mathbf{C}_{j\to\mathrm{M}}$，最后把全局姿势乘以对应的绑定姿势逆矩阵$(\mathbf{B}_{j\to\mathrm{M}})^{-1}$，以生成每个关节的蒙皮矩阵$\mathbf{K}_j$。

[7]译注：此术语虽然称为调色板（palette），但实际上与颜色无关。调色板在这里的意思源于画家把有限数量的颜料置于调色板上，用于挑选及混合。在蒙皮时，关节也是有限数量的，网格的每个顶点可挑选采用哪一个关节，或从数个关节中混合。

11.5.2.3 引入模型至世界变换

每个顶点最终会由模型空间变换至世界空间。因此有些引擎会把蒙皮矩阵调色板预先乘以物体的模型至世界变换。这是个很有用的优化，因为渲染引擎渲染蒙皮几何时，每个顶点能节省一个矩阵乘法。（当要处理几十万个顶点时，这些节约就能积少成多！）

要把模型至世界变换引入蒙皮矩阵，只需把它简单地串接至正常的蒙皮矩阵方程：

$$(\mathbf{K}_j)_{\mathrm{W}} = (\mathbf{B}_{j\to\mathrm{M}})^{-1}\mathbf{C}_{j\to\mathrm{M}}\mathbf{M}_{\mathrm{M}\to\mathrm{W}}$$

有些引擎把模型至世界变换如此烘焙至蒙皮矩阵，但有些引擎则不会这么做。选择权完全由工程团队决定，当中受多种因素所左右。例如，有一个情况我们绝**不会**这么做，这就是我们要多个角色同时播放单个动画——此技术通常称为**动画实例**（animation instancing），常用于大规模的群众动画。在此情况下，模型至世界变换需要分离出来，以便把单个矩阵调色板应用至群众中的所有角色。

11.5.2.4 把顶点蒙皮至多个关节

要把顶点蒙皮至多个关节，我们可以计算顶点分别蒙皮至每个关节，产生对于每个节点的模型空间位置，然后把这些结果进行**加权平均**来求出最终位置。这些权重是由角色绑定师（character rigging artist）提供的，并且每个顶点的权重之和必为1。（若此和并非为1，工具管道应该把它们归一化。）

N个数值a_0至a_{N-1}含对应权重w_0至w_{N-1}，并且$\sum_{i=0}^{N-1} w_i = 1$，那么这些数值的通用加权平均公式是：

$$\hat{a} = \sum_{i=0}^{N-1} w_i a_i$$

此公式同样适用于矢量数值$\mathbf{a}_i$。因此，对于一个顶点蒙皮至N个关节，关节索引为j_0至j_{N-1}，权重为w_0至w_{N-1}，我们可以把方程(11.4)延伸如下：

$$\mathbf{v}_{\mathrm{M}}^{\mathrm{C}} = \sum_{i=0}^{N-1} w_{ij}\mathbf{v}_{\mathrm{M}}^{\mathrm{B}}\mathbf{K}_{j_i}$$

当中$\mathbf{K}_{j_i}$是关节j_i的蒙皮矩阵。

11.6 动画混合

动画混合（animation blending）是指能令一个以上的动画片段对角色最终姿势起作用的技术。更准确地说，混合是把两个或更多的**输入姿势**结合，产生骨骼的**输出姿势**。

混合通常会结合某个时间点的两个或两个以上姿势，并生成同一时间点的输出。在此语境中，混合用作结合两个或两个以上的动画，自动产生大量新动画，而无须手工制作这些动画。例如，通过混合负伤的及无负伤的步行动画，我们可以生成二者之间不同负伤程度的步行动画。又例如，我们可以混合某角色的向左瞄准及向右瞄准动画，就能令角色瞄准左右两端之间的所需方位。动画混合可用于对面部表情、身体站姿、运动模式等的极端姿势之间插值。

动画混合也可以用于求出**不同**时间点的两个已知姿势之间的姿势。当我们要取得角色在某时间点的姿势，而该时间点并非刚好对应动画数据中的采样帧，那么就可使用这种动画混合。我们也可使用时间上的动画混合——通过在短时段内把来源动画逐渐混合至目标动画，就能把某动画圆滑地过渡至另一动画。

11.6.1 线性插值混合

给定两个骨骼姿势$\mathbf{P}_A^{\text{skel}} = \{(\mathbf{P}_A)_j\}\,|_{j=0}^{N-1}$及$\mathbf{P}_B^{\text{skel}} = \{(\mathbf{P}_B)_j\}\,|_{j=0}^{N-1}$，我们希望求得此两极端的中间姿势$\mathbf{P}_{\text{LERP}}^{\text{skel}}$。方法之一是，对这两个来源姿势中每个关节的局部姿势进行线性插值（linear interpolation，LERP），可表示如下：

$$\begin{aligned}(\mathbf{P}_{\text{LERP}})_j &= \text{LERP}\left[(\mathbf{P}_A)_j, (\mathbf{P}_B)_j, \beta\right] \\ &= (1-\beta)(\mathbf{P}_A)_j + \beta(\mathbf{P}_B)_j\end{aligned} \tag{11.5}$$

而整个骨骼的插值后姿势，仅仅是所有关节插值后姿势的集合：

$$\mathbf{P}_{\text{LERP}}^{\text{skel}} = \{(\mathbf{P}_{\text{LERP}})_j\}\,|_{j=0}^{N-1} \tag{11.6}$$

在这些方程中，β称为**混合百分比**（blend percentage）或**混合因子**（blend factor）。当$\beta = 0$，骨骼的最终姿势便会完全和$\mathbf{P}_A^{\text{skel}}$匹配；当$\beta = 1$，最终姿势就会和$\mathbf{P}_B^{\text{skel}}$匹配。当$\beta$介乎0～1，最终姿势便是两个极端某中间姿势。图11.10展示了此效果。

我们在此再谈一些细节。我们对**关节姿势**进行线性插值，即是指对4×4**变换矩阵**进行插值。然而，如第4章所说，直接对矩阵插值并非切实可行。这也是通常用SQT格式表示局部姿势的原因之一，那么我们就可用4.2.5节定义的LERP运算，分别对SQT中每个部分插

值。SQT中的位移部分**T**只是一个直截了当的矢量LERP：

$$\begin{aligned}(\mathbf{T}_{\text{LERP}})_j &= \text{LERP}\left[(\mathbf{P}_A)_j, (\mathbf{P}_B)_j, \beta\right] \\ &= (1-\beta)(\mathbf{T}_A)_j + \beta(\mathbf{T}_B)_j\end{aligned} \tag{11.7}$$

旋转部分可使用四元数LERP或SLERP（球面线性插值）：

$$\begin{aligned}(\mathrm{q}_{\text{LERP}})_j &= \text{LERP}\left[(\mathrm{q}_A)_j, (\mathrm{q}_B)_j, \beta\right] \\ &= (1-\beta)(\mathrm{q}_A)_j + \beta(\mathrm{q}_B)_j\end{aligned} \tag{11.8a}$$

或

$$\begin{aligned}(\mathrm{q}_{\text{SLERP}})_j &= \text{SLERP}\left[(\mathrm{q}_A)_j, (\mathrm{q}_B)_j, \beta\right] \\ &= \frac{\sin((1-\beta)\theta)}{\sin\theta}(\mathrm{q}_A)_j + \frac{\sin(\beta\theta)}{\sin\theta}(\mathrm{q}_B)_j\end{aligned} \tag{11.8b}$$

最后，缩放部分根据引擎支持的缩放类型（统一或非统一），使用标量或矢量LERP：

$$\begin{aligned}(\mathbf{s}_{\text{LERP}})_j &= \text{LERP}\left[(\mathbf{s}_A)_j, (\mathbf{s}_B)_j, \beta\right] \\ &= (1-\beta)(\mathbf{s}_A)_j + \beta(\mathbf{s}_B)_j\end{aligned} \tag{11.9a}$$

或

$$\begin{aligned}(s_{\text{LERP}})_j &= \text{LERP}\left[(s_A)_j, (s_B)_j, \beta\right] \\ &= (1-\beta)(s_A)_j + \beta(s_B)_j\end{aligned} \tag{11.9b}$$

对两个骨骼姿势进行线性插值时，最自然的中间姿势通常是令关节独立在其父关节中间进行插值。换句话说，姿势混合通常在局部姿势进行。若直接在模型空间混合全局姿势，其结果从生物力学上看显得不真实。[8]

由于姿势混合是在局部姿势进行的，每个关节姿势的线性插值完全独立于同一骨骼上的其他关节插值。这意味着，线性插值可完全并行地在多处理架构上运行。[9]

[8]译注：其中一个显而易见的问题在于，就算在局部姿势中只有旋转，用全局姿势插值后，关节之间的距离（骨头的长度）会改变。

[9]译注：虽然如此，但一般来说每个骨骼实际上只有上百个关节，把一个角色内的关节并行插值并不划算，因为可能会减低内存存取的连贯性，反而对性能有影响。如游戏中有大量角色，可以考虑以角色（骨骼）为最小单位进行并行。

11.6.2 线性插值混合的应用

现在我们对线性插值混合已有基本认识，接下来看看它在游戏中的典型应用。

11.6.2.1 时间性混合

我们在11.4.1.1节提及，游戏动画几乎永不会在整数的帧索引上采样。由于浮动的帧率，玩家可能实际上会看到第0.9、1.85、3.02帧，而非刚好是期望看到的第1、2、3帧。此外，有些动画压缩法仅仅储存不一样的关键帧，这些关键帧非均匀地分布在动画片段的局部时间线上。无论是以上哪一种情况，我们都需要求出动画片段中各个采样姿势之间的中间姿势。

我们可使用LERP混合求得这些中间姿势。例如，假设我们的动画片段的姿势采样是均匀分布在0、Δt、$2\Delta t$、$3\Delta t$等时间点之上的。为了求时间点$t = (2.18)\Delta t$的姿势，我们只需简单地采用$\beta = 0.18$的混合百分比，对时间点$2\Delta t$及$3\Delta t$的姿势进行线性插值。

给定两个于时间点t_1及t_2的姿势采样，以下方程可以求得位于此期间时间点t的姿势：

$$\begin{aligned}\mathbf{P}_j(t) &= \text{LERP}\left[\mathbf{P}_j(t_1), \mathbf{P}_j(t_2), \beta(t)\right] \\ &= (1-\beta(t))\mathbf{P}_j(t_1) + \beta(t)\mathbf{P}_j(t_2)\end{aligned} \tag{11.10}$$

当中混合因子$\beta(t)$为比率：

$$\beta(t) = \frac{t - t_1}{t_2 - t_1} \tag{11.11}$$

11.6.2.2 动作连续性：淡入 / 淡出

游戏角色上的动画，是大量细粒度动画片段所拼砌而成的。若你的动画师有足够能力，就能令动画角色在个别片段**之内**的动作自然真实。然而，众所周知，把片段过渡至另一片段，要达到同样的质量是极难的。在游戏动画中常见到的“跳帧（pop）”，多数出现于角色从一个片段过渡至另一个片段的时候。

理想地，我们希望角色身体每个部分的动作都是完全流畅的，就算在过渡中亦然。换言之，骨骼中每个关节移动时所描绘出的三维路径不应含突然“跳跃”。我们称此为C^0**连续**（C^0 continuity），如图11.26所示。

不单路径本身应该连续，其第一导数（速度曲线）也应连续。此称为C^1**连续**（又称速度及动量的连续性）。若使用更高阶的连续性，角色的动作会显得更佳及更真实。例如，若我们可能希望移动路径达至C^2连续，即路径的第二导数（加速度曲线）也为连续。

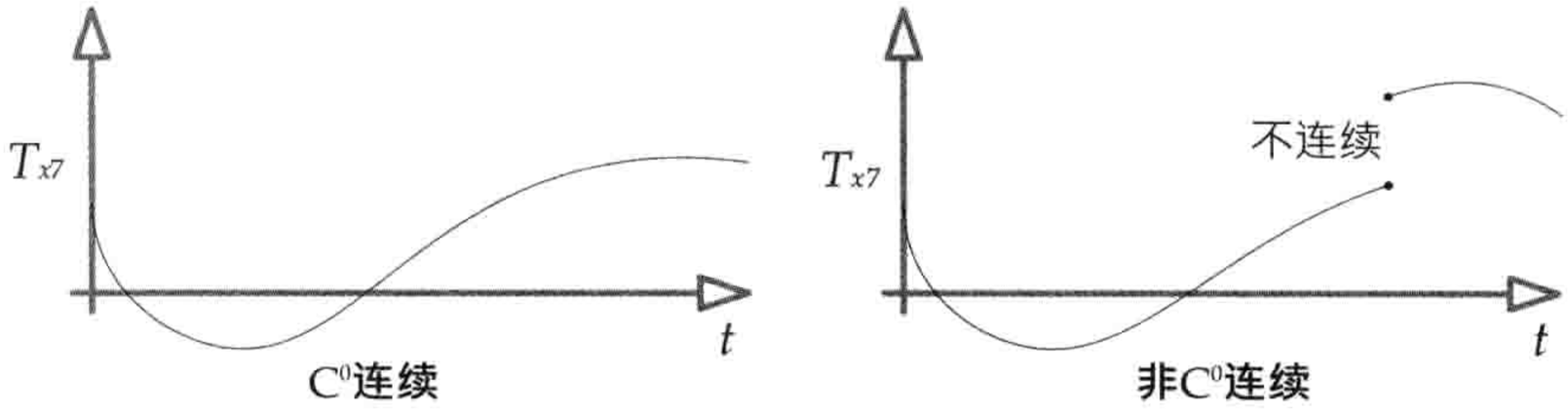


图 11.26: 左图的通道函数是C^0连续的，右图则不是。

我们通常难以达到严格数学上的C^1或以上的连续性。然而，我们可使用LERP的动画混合达到相当不错的C^0动作连续性。这种混合也可以相当好地逼近C^1连续性。当应用至过渡片段时，LERP混合有时称为淡入 / 淡出（cross-fading）。LERP混合可能会产生一些瑕疵，例如忌讳的“滑脚”问题，因此使用时必须适如其分。

要对两个动画进行淡入 / 淡出，我们要把两个片段的时间线适度地重叠。在开始时间t_{start}时，混合百分比β为0，即混合之始只能看到片段A。然后我们逐步递增β，直至时间t_{end}时β为1。此时只能看到片段B，我们可以把片段A完全撤去。淡入 / 淡出的持续时间（$\Delta t_{\text{blend}} = t_{\text{end}} - t_{\text{start}}$）有时候称为**混合时间**（blend time）。

淡入 / 淡出的种类

两种常见的淡入 / 淡出过渡方法如下。

- **圆滑过渡**（smooth transition）：播放片段A及B的同时把β从0至1递增。要效果好，两个片段都必须为循环动画，并且两个片段应该同步至手脚位置大致匹配。（若不这么做，淡入 / 淡出的结果常会显得完全不自然。）图11.27展示了此技术。
- **冻结过渡**（frozen transition）：片段A的局部时钟停顿于片段B开始播放之时。那么片段A的骨骼姿势就会冻结起来，而片段B则渐渐取缔角色的动作。这种技术适合于混合两个不相关且不能在时间上同步的片段，因为片段必须在时间上同步才能使用圆滑过渡。图11.28显示了冻结过渡。

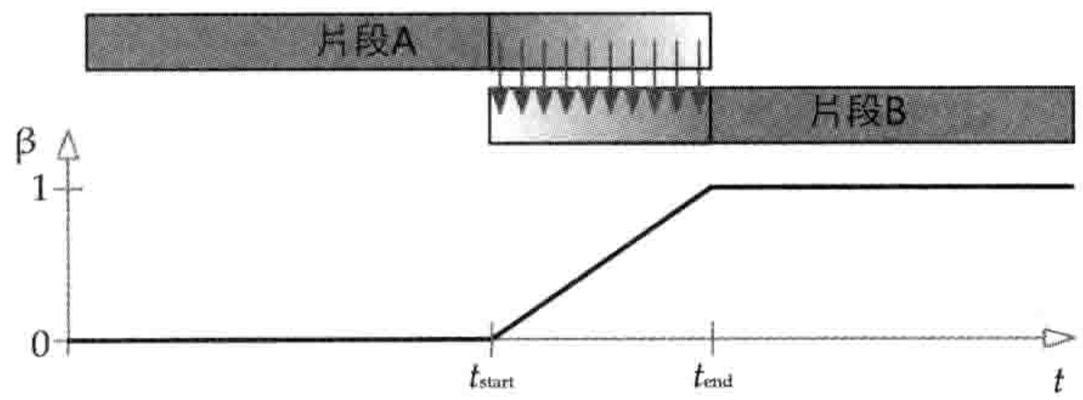


图 11.27: 在圆滑过渡中，两片段的局部时钟在过渡期间也保持运行。

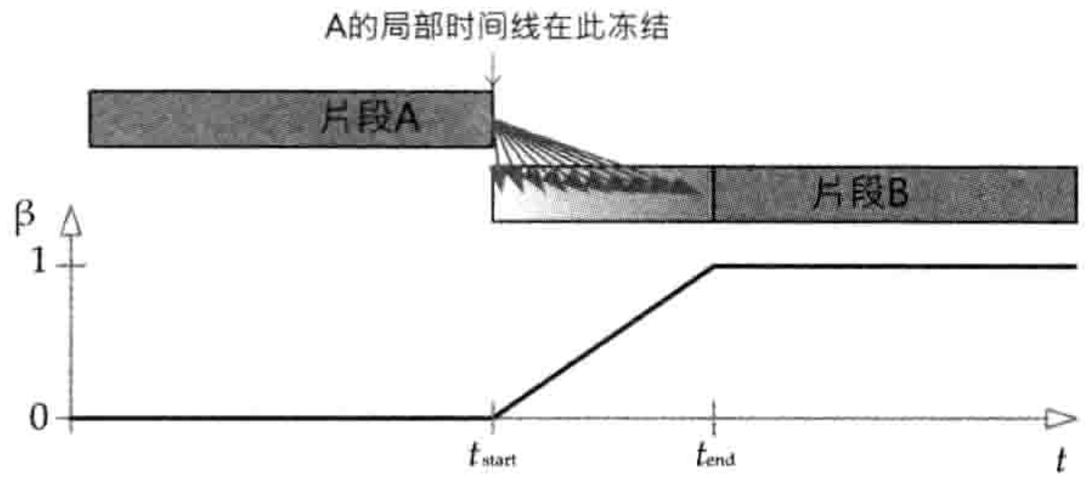


图 11.28: 在冻结过渡中，片段A的局部时钟在过渡期间停止运行。

我们也可以控制混合因子β过渡过程中的变化方式。在图11.27及图11.28中，混合因子按时间线性变化。为了达致更圆滑的过渡，我们可以令β按时间的三次函数变化，例如用一维Bézier曲线。当把这些曲线应用到正在淡出的当前片段时，该曲线就称为**缓出曲线**（ease-out curve）；当应用到正在淡入的新片段时，就称作**缓入曲线**（ease-in curve）。这种曲线如图11.29所示。

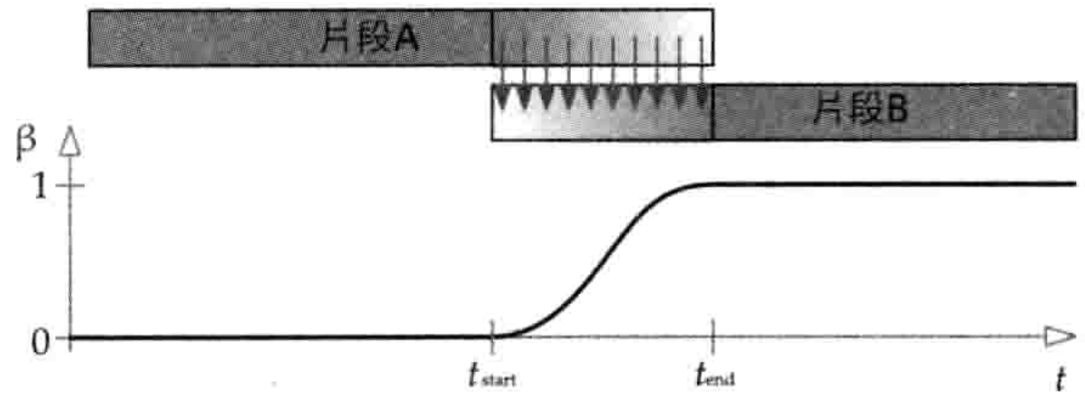


图 11.29: 采用缓入/缓出三次曲线为混合因子的圆滑过渡。

以下列出了Bézier缓入/缓出曲线的方程。此方程能传回混合时程中任何时间t的β值。β_{start}为混合之始t_{start}的混合因子，β_{end}为时间t_{end}时的最终混合因子。参数u是t_{start}和t_{end}之间的**归一化时间**，为方便起见我们设$v = 1 - u$（即**逆向**归一化时间）。注意Bézier切线T_{start}和T_{end}设为对应的混合因子β_{start}及β_{end}，因为这样能产生符合所需的良好曲线：

$$
\begin{aligned}
&\text{设 } u = \frac{t - t_{\text{start}}}{t_{\text{end}} - t_{\text{start}}} \\
&\text{及 } v = 1 - u\text{，则：} \\
&\beta(t) = (v^3)\beta_{\text{start}} + (3v^2u)T_{\text{start}} + (3uv^2)T_{\text{end}} + (u^3)\beta_{\text{end}} \\
&\qquad = (v^3 + 3v^2u)\beta_{\text{start}} + (3vu^2 + u^3)\beta_{\text{end}}
\end{aligned}
$$

核心姿势

现在是适当时间谈谈另一种**无须**混合就能产生连续动作的方法，这就是动画师确保某片段的最后姿势能匹配后续片段的首个姿势。实践上，动画师通常会制定一组**核心姿势**（core pose），例如包括一个直立的核心姿势、一个蹲下姿势、一个躺下姿势等。只要能确保角色的每个动画片段以某核心姿势开始，并以某核心姿势结束，就能简单地把核心姿势匹配的片段连接成具C^0连续性的动画。C^1或更高阶的连续性的动画也可做到，只要确保角色在片段结束时动作能圆滑地过渡至后续片段开始时的动作。具体做法也很简单，只需创作一段圆滑的动画，然后把它切为两个或两个以上的动画片段。

11.6.2.3 方向性运动

基于LERP的动画混合通常应用在角色运动（character locomotion）。真实的人类在步行或跑步时，他有两种方式改变移动方向。第一种方法，他能转身改变方向，那么他能一直面向移动的方向。笔者称这种为**轴转移动**（pivotal movement），因为他转身时是按其垂直轴旋转的。另一种方法，他能保持面向某方向，而同时向前后左右步行（在游戏世界中称为**strafing**），使移动方向和面向方向互相独立。笔者称此为**靶向移动**（targeted movement），因为这种移动通常用于在移动同时保持角色的眼睛或武器瞄准某个目标。图11.30显示了这两种移动方式。

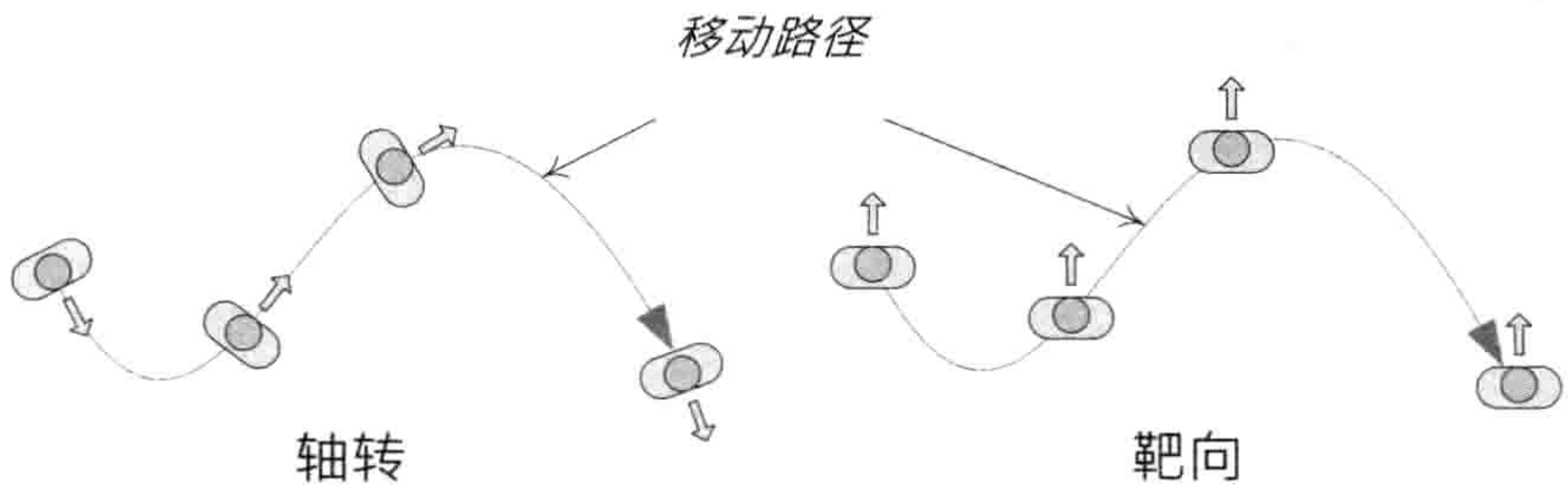


图 11.30: 在轴转移动中，角色面向他移动的方向，并以他的垂直轴为轴心旋转。在靶向移动中，移动方向不需要匹配面向方向。

靶向移动

为了实现**靶向移动**，动画师会制作3种不同的循环动画片段，包括向前、向左及向右移动。笔者称这些为**方向性运动片段**（directional locomotion clip）。我们把这3个片段排列在一个半圆圆周之上，向前位于0°，向左位于90°，向右位于−90°。只要使角色面对方向对齐

至0°，就能在半圆上求得移动方向，然后选择该角度上的两个相邻片段以LERP方式混合。混合百分比β由移动角度和相邻片段的角度求得。图11.31说明了这种技术。

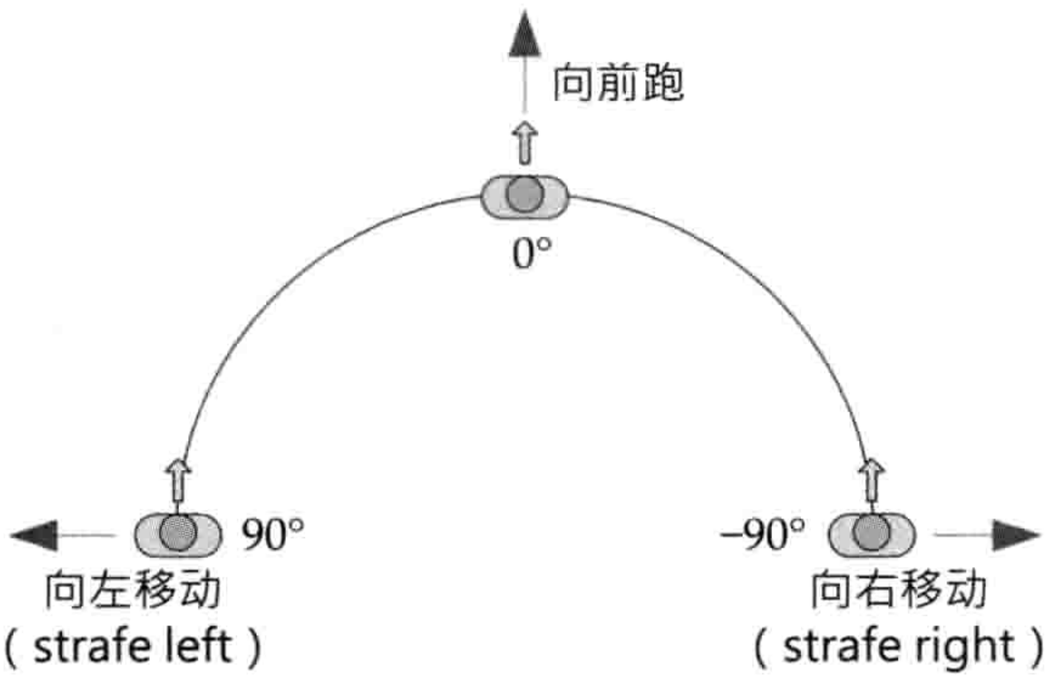


图 11.31: 把往4个主方向移动的循环运动片段互相混合，便可实现靶向移动。

注意我们并没有在混合中包含向后移动，来形成一个全周的混合。这是因为向左右移动和向后移动的混合，一般来说会显得不自然。问题在于，向左移动时，角色会把右脚跨于左脚前方，这么做能令向左与向前移动的混合动画显得正确。同样，向右移动的动画通常会制作成左脚跨于右脚前方。但是，当尝试把这些左右移动直接和向后移动混合，一条腿就会穿过另一条腿，这是极其尴尬、不自然的。此问题有几个解决方法。其中一个可行方法就是定义两个半圆混合，一个用作向前移动，一个用作向后移动，再为这两个半圆分别制作两组适合混合的左右移动动画。当从一个半圆过渡至另一个半圆时，我们可以加入明确的过渡动画，使角色有机会适当地调整步伐及跨步问题。

轴转移动

为了实现**轴转移动**，我们可简单播放向前运动循环片段，并同时以垂直轴旋转整个角色，以达至转向目的。若角色在转向时不保持完全笔直，轴转移动会显得更自然——真实人类在转向时会造成少许倾斜[10]。我们可以令整个角色的垂直轴倾斜一点，但这会造成一些问题，内脚会插进地下，外脚则会离地。要做出更自然的效果，可使用3个向前步行或跑步的动画，一个完全向前，一个是极端向左转，一个是极端向右转。然后就可以使用LERP混合这些动画，产生想要的倾斜角度动画了。

[10]译注：实际上，这在物理上是必然的，必须有向心力才能形成转向。在水平平面移动时，要靠倾斜身体产生脚底的磨擦力来产生向心力。原理上和摩托车转向时倾斜的情况一样。具体关系是$\tan\theta = v^2/gr$，当中θ是倾斜角，v是向前的速度，g是引力常数，r是圆周运动的半径。

11.6.3 复杂的线性插值混合

在真实的游戏引擎中，角色会使用广泛不同种类的复杂混合，以完成不同目的。为了更易使用，我们会把几个常见的复杂混合“预包装”起来。以下几节会探讨一些流行的预包装复杂混合类型。

11.6.3.1 泛化的一维线性插值混合

LERP混合可以扩展至多于两个动画片段，笔者称此技术为**一维线性插值混合**（one-dimensional LERP blending）。我们定义一个新的混合参数b，此参数可任意指定范围（如$-1\sim+1$、$0\sim1$，甚至$27\sim136$）。任意数量的片段置于此范围中的某点之上，如图11.32所示。给定任意的b值，我们选取两个最近于该值的片段，并用方程(11.5)混合两者。若那两个片段分别位于点b_1及b_2，那么混合百分比β可用类似方程(11.10)的形式求出：

$$\beta = \frac{b - b_1}{b_2 - b_1} \tag{11.12}$$

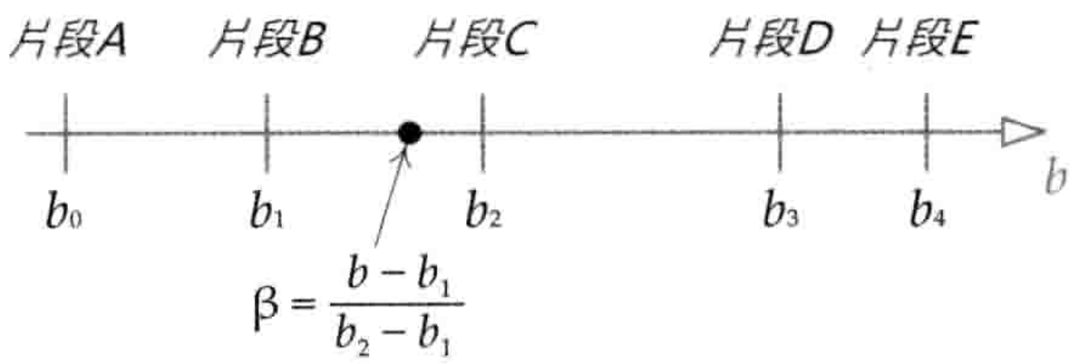


图 11.32: 对N个动画片段的泛化线性混合。

方向性移动仅是一维LERP混合的特例。我们只需简单弄直放置方向性动画片段的半圆，并把移动方向角色θ作为参数b（范围为$-90°\sim90°$）。此混合范围内可放置任意数量的动画片段于不同角度之上。图11.33显示了这种用法。

11.6.3.2 简单的二维线性插值混合

有时候我们想圆滑地同时改变角色动作的**两个**方面。例如，我们可能希望角色能用武器在水平方向及垂直方向瞄准。又例如，我们可能希望改变角色走动时的步伐长度及双脚分隔的距离。我们可以把一维LERP混合扩展至二维，以达成上述的效果。

若我们得知，所需的二维混合只涉及4个动画片段，并且这些片段位于一个正方形区域的4角，那么我们可用3个一维混合求得混合姿势。我们的广义混合因子b变成二维混合矢

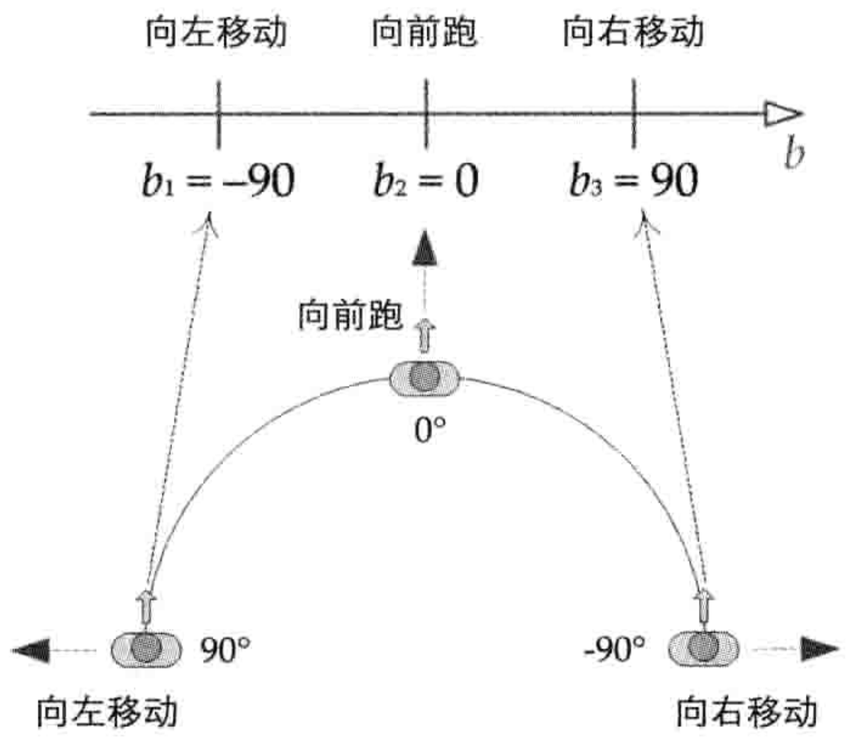


图 11.33: 靶向移动所使用的方向性动画片段，可理解为特殊的一维线性插值混合。

量$\mathbf{b} = [b_x \quad b_y]$。若$\mathbf{b}$位于4个片段所包围的正方形区域中，那么就可以用以下的步骤求得所需的混合姿势。

- 利用水平混合因子b_x求出两个中间姿势，一个在顶边两个动画片段之间，一个在底边两个片段之间。这两个姿势可以用简单的一维LERP混合求得。
- 然后，使用垂直混合因子b_y，把两个中间姿势用一维LERP混合求出最终姿势。

图11.34展示了此技术。

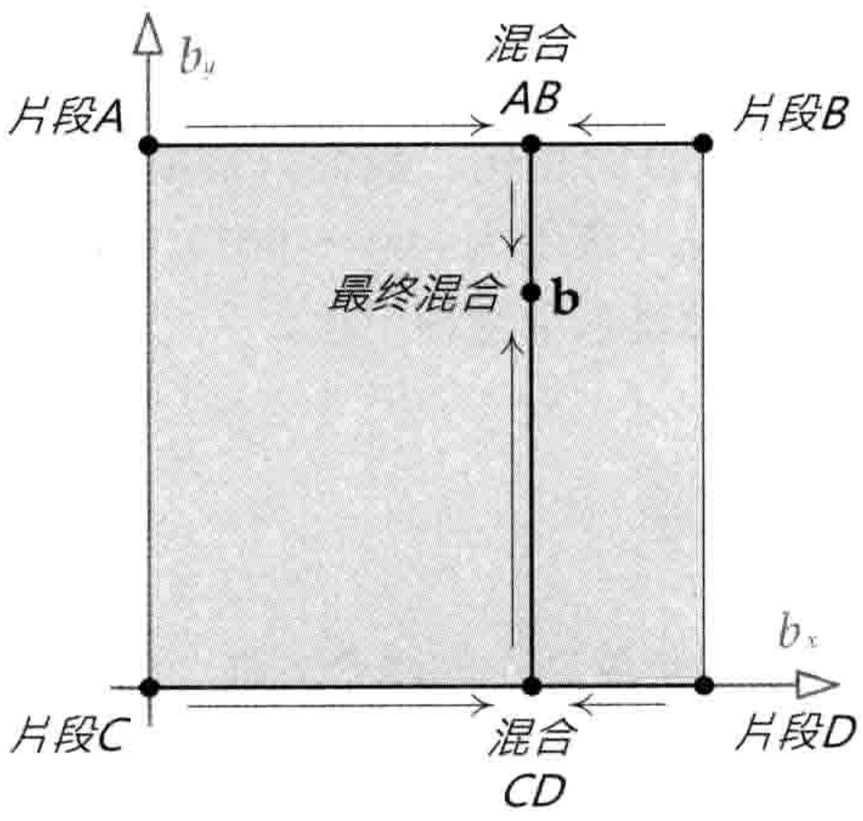


图 11.34: 对正方形区域4角的动画片段做二维动画混合。

11.6.3.3 三角形的二维线性插值混合

以上所述的简单二维混合技术，只能用于动画片段置于正方形4角的混合。那么如何能混合任意数量置于混合空间任意二维位置的动画片段呢？

首先我们想象有3个需混合的动画片段。第i个片段对应二维混合空间中的一个混合坐标$\mathbf{b}_i = [b_{xi} \quad b_{yi}]$，这3个坐标形成二维混合空间中的三角形。每个片段$i$定义一组关节姿势$\{(\mathbf{P}_{ij})\}|_{j=0}^{N-1}$，当中$j$是关节索引而$N$是骨骼中的关节总数。我们想求出对于三角形内任意点$\mathbf{b}$的插值后骨骼姿势，如图11.35所示。

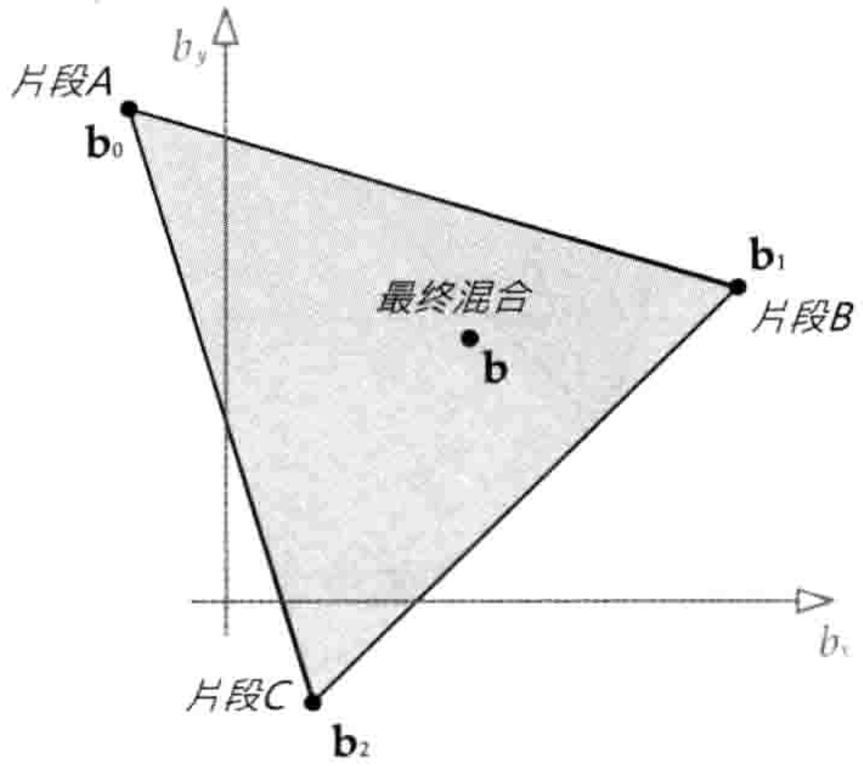


图 11.35: 对3个动画片段的二维动画混合。

然而，我们怎样才能用LERP混合3个动画片段呢？庆幸我们有一个简单答案：LERP函数实际上可以用于任何数量的输入，因为它仅仅是一个**加权平均**（weighted average）。如同其他加权平均数，权重之和必须为1。以两个输入的LERP混合来说，我们使用β和$1-\beta$的权重，显然它们之和是1。那么对于3个输入的LERP，我们可简单地使用3个权重，α、β，以及$\gamma = 1 - \alpha - \beta$。

$$(\mathbf{P}_{\text{LERP}})_j = \alpha(\mathbf{P}_0)_j + \beta(\mathbf{P}_1)_j + (1 - \alpha - \beta)(\mathbf{P}_2)_j \tag{11.13}$$

给定二维混合矢量$\mathbf{b}$，我们可使用$\mathbf{b}$相对于3个片段所形成的三角形的**重心坐标**（barycentric coordinates）[11]求得混合权重α、β、γ。一般来说，在顶点为$\mathbf{b}_0$、$\mathbf{b}_1$、$\mathbf{b}_2$的三角形中，某点的重心坐标$\mathbf{b}$及3个标量值（α、β、γ）满足以下关系：

$$\mathbf{b} = \alpha\mathbf{b}_0 + \beta\mathbf{b}_1 + \gamma\mathbf{b}_2 \tag{11.14}$$

[11] http://en.wikipedia.org/wiki/Barycentric_coordinate_system_(mathematics)

及

$$\alpha + \beta + \gamma = 1$$

这些就正是我们要寻找的3片段加权平均的权重。图11.36显示了重心坐标。

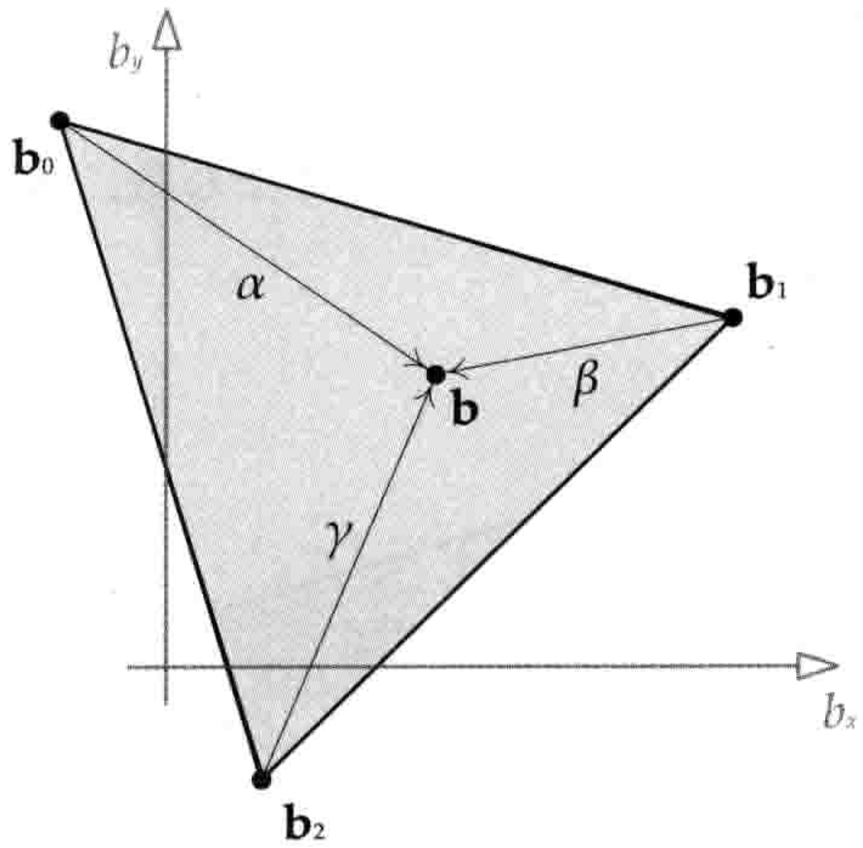


图 11.36: 三角形中的多个重心坐标。

注意，把重心坐标$(1,0,0)$代入方程会得出$\mathbf{b}_0$，代入$(0,1,0)$得$\mathbf{b}_1$、代入$(0,0,1)$得$\mathbf{b}_2$。类似地，把这些混合权重代入方程(11.13)会分别得出姿势$(\mathbf{P}_0)_j$、$(\mathbf{P}_1)_j$及$(\mathbf{P}_2)_j$。再者，重心坐标$(1/3,1/3,1/3)$位于三角形的重心，并会产生3个姿势的相同比重混合。这完全合乎我们的预期。

11.6.3.4 泛化的二维线性插值混合

重心坐标技术可扩展至任意数目的动画片段，这些片段可置于二维混合空间的任意位置。我们不在此详细展开，但其想法是利用**Delaunay三角剖分**（Delaunay triangulation）技术[12]，从多个动画片段位置$\mathbf{b}_i$求出一组三角形。得到这些三角形后，从中找寻包围混合点$\mathbf{b}$的三角形，然后在该三角形使用上述3片段的LERP，见图11.37。[13]

[12] http://en.wikipedia.org/wiki/Delaunay_triangulation

[13] 译注：这里所述的方法，每次只能混合3个片段。另一个混合多个动画片段的方法是使用更高维的空间，例如，4个片段可想象为四面体（tetrahedra）之4个顶点，同样可使用重心坐标计算混合。从另一角度看此问题，使用加权平均就能混合多个片段。

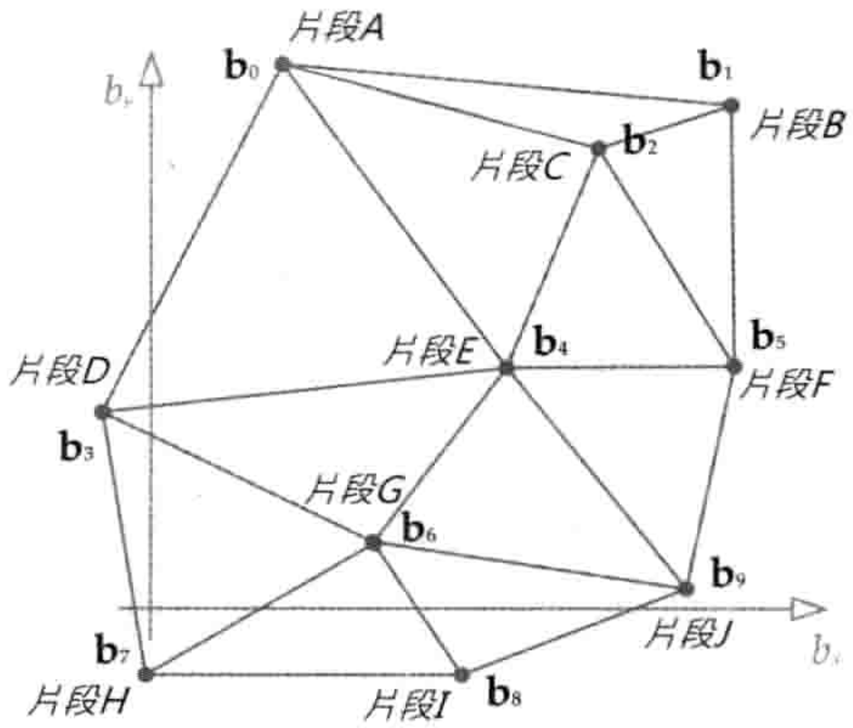


图 11.37: 对任意数目、置于任意二维混合空间位置的动画片段，使用Delaunay三角剖分。

11.6.4 骨骼分部混合

人可独立控制身体不同部位。例如，我可以在步行时挥动右臂，并同时令左臂指着某物。在游戏中实现这种动作的方法之一是，使用名为**骨骼分部混合**（partial-skeleton blending）的技术。

回想方程(11.5)及(11.6)，进行正常LERP混合的时候，混合百分比β应用在骨骼中每个关节的混合中。骨骼分部混合延伸此做法，容许每关节设置不同的混合百分比。换言之，我们对每个关节j定义一个独立的混合百分比β_j。整个骨骼的混合百分比集合$\{\beta_j\}|_{j=0}^{N-1}$有时候称为**混合遮罩**（blend mask），因为这可以把某些关节的混合百分比设为0，来“掩盖”那些关节。

例如，我们要令角色向某人挥动右臂及右手，并令他在步行、跑步及站立时都能挥手。要用骨骼分部混合实现这个效果，动画师要制作3个全身动作：**步行**、**跑步**及**站立**。动画师也要制作一个**挥手**动作。然后，创建一个混合遮罩，遮罩中除了将右肩、右肘、右腕及右手手指关节的混合百分比设为1，其他关节的混合百分比都设为0：

$$\beta_j = \begin{cases} 1 & j \in \text{右手} \\ 0 & \text{其他} \end{cases}$$

当**步行**、**跑步**或**站立**片段与**挥手**片段混合时使用这个混合遮罩，结果就是角色在步行、跑步或站立的同时挥动右手。

虽然骨骼分部混合有其用途，但它也倾向造成不自然角色动作。此问题的原因有二。

- 若相连关节的混合因子突兀改变，可导致身体一部分的动作与其他部分分离。在上述例子中，右肩关节的混合因子突兀地改变，这样会造成上脊椎、颈及头的关节由一个动画所驱动，而右肩及右臂则完全由另一动画所驱动，看上去会很奇怪。减轻此问题的方法之一，可以逐渐改变混合因子，而非突兀地改变。（在例子中，可以在右肩用0.9，上脊椎用0.5，颈和中脊椎用0.2。）
- 现实中的人体动作并不是完全独立的。例如，一个人在跑步中挥手，我们会预期他的挥手动作比站立时更“晃动”及不受控制。但是，使用骨骼分部混合时，手臂的动画无论其他身体部分在做什么都是等同的。此问题难以用分部混合解决。取而代之，近年许多游戏开发者改用另一看上去更自然的技术，该技术称为**加法混合**。

11.6.5 加法混合

加法混合（additive blending）为动画结合问题带来全新的方式。加法混合引入一种称为**区别片段**（difference clip）的新类型动画。如名字所暗示，区别片段代表两段正常动画的区别。区别动画可以加进普通的动画片段，以产生一些有趣的姿势和动作**变化**。本质上，区别片段储存了一个姿势变换至另一个姿势所需的改变。区别动画在游戏业界常称为**加法动画片段**。本书采用**区别片段**，因为这更准确地描述了它的本质。

我们先考虑两个输入片段，分别为**来源片段**（source clip，S）及**参考片段**（reference clip，R）。概念上，区别片段是$D = S - R$。若区别片段D加进原来的参考片段，我们就会取回来源片段（$S = D + R$）。只需要把某百分比的D加进R，我们也可以产生介乎R和S之间的动画，这如同使用LERP为两个极端片段找出中间动画。然而，加法混合技术之美在于，制作一个区别片段之后，可以把该片段加进其他不相关的片段，而不仅限于原来的参考片段。我们称这些动画为**目标动画**（target clip，T）。

例如，若参考片段是角色正常跑步，而来源片段是疲惫下跑步，那么区别片段只含有令角色在跑步中显得疲惫所需的改变。若此区别片段应用至角色步行，结果会是一个疲惫下步行的结果。通过加入一个区别片段至多个“正常”的动画片段，便能创建许多有趣且自然的动画。又或是把不同的区别动画加进一个目标动画，也能产生不同的效果。

11.6.5.1 数学公式

区别动画D的定义为某来源动画S和某参考动画R间之差异。因此概念上，（在某时间点的）区别姿势是$D = S - R$。当然，我们要处理的是关节姿势，而不是标量，不能简单地把姿势相减。一般来说，关节姿势是一个4×4仿射矩阵$\mathbf{P}_{C \to P}$，此矩阵会把点或矢量从子关节

的局部空间变换至其父关节的空间。以矩阵来说，等价于标量减法的运算是乘以逆矩阵。因此，给定骨骼中每个关节j的来源姿势$\mathbf{S}_j$及参考姿势$\mathbf{R}_j$，区别姿势$\mathbf{D}_j$可定义如下（在本讨论中，由于我们在处理从子至父的姿势矩阵，所以舍弃了$C \to P$或$j \to \mathrm{p}(j)$下标）：

$$\mathbf{D}_j = \mathbf{S}_j \mathbf{R}_j^{-1}$$

把区别姿势$\mathbf{D}_j$“加进”目标姿势$\mathbf{T}_j$会产生新的加法姿势$\mathbf{A}_j$。具体方法是简单地串接区别变换及目标变换：

$$\mathbf{A}_j = \mathbf{D}_j \mathbf{T}_j = (\mathbf{S}_j \mathbf{R}_j^{-1}) \mathbf{T}_j \tag{11.15}$$

我们可以通过把区别姿势“加到”原来的参考姿势，验证上面的方程：

$$\begin{aligned} \mathbf{A}_j &= \mathbf{D}_j \mathbf{R}_j \\ &= \mathbf{S}_j \mathbf{R}_j^{-1} \mathbf{R}_j \\ &= \mathbf{S}_j \end{aligned}$$

换句话说，把区别动画D加到原来的参考动画R，便会得出我们预期的来源动画S。[14]

不同动画短片的时间性插值

我们在11.4.1.1节中知悉，游戏动画几乎永不会在整数帧索引上采样。要求得任意时间t的姿势，我们必须对置于时间t_1及t_2的两个相邻姿势样本进行**时间性插值**。幸好区别片段如同其他非加法片段一样，可使用时间性插值。我们只需简单地把方程(11.10)及(11.11)套用在区别片段，如同应用在一般动画一样。

注意，仅当输入片段S和R的持续时间相同，才能求得它们的区别动画。否则会有一段时间缺乏S或R的定义，那么该段时间的D也无定义。

加法混合百分比

在游戏中，我们经常希望可以混合某百分比的区别动画，以产生不同程度的效果。例如，若区别片段能使角色的头部向右转80°，混合该片段的50%应该可令他的头部仅向右转40°。

[14] 译注：本节中的**D**、**S**、**R**其实也可使用SQT来表示。一般来说，SQT的串接和求逆运算都比矩阵高效。

我们可再次使用LERP实现这种效果。我们希望在无修改的目标动画和完全应用区别动画后的新动画之间插值。为此我们把方程(11.15)扩展如下：

$$\begin{aligned}\mathbf{A}_j &= \text{LERP}(\mathbf{T}_j, \mathbf{D}_j\mathbf{T}_j, \beta) \\ &= (1-\beta)(\mathbf{T}_j) + \beta(\mathbf{D}_j\mathbf{T}_j)\end{aligned} \tag{11.16}$$

如第4章所谈及，我们不能直接对矩阵LERP。因此方程(11.16)必须分拆为3个分别对S、Q和T的插值，如同方程(11.7)、(11.8)、(11.9)那么做。

11.6.5.2 比较加法混合和分部混合

加法混合在多方面和分部混合相似。例如，我们可以取得站立片段和站立并挥右臂片段的区别。其结果差不多等同于用分部混合令右臂挥动。然而，相对于分部混合看上去的“分离”问题，加法混合的同类问题较少。这是由于用加法混合时，我们不是取代一组关节的动画，也不是对两个可能无关的姿势进行插值。取而代之，我们把动作加进原来的动画中，动作可能横跨整个骨骼。其效果为，区别动画“得悉”怎样改变角色的姿势来做到某些具体效果，例如表现疲惫、把头转向某方向或挥手。这些改动能施于各式各样的动画上，而且效果通常会很自然。

11.6.5.3 加法混合之局限

当然，加法动画也非银弹。由于它把动作加进已有的动画，会产生骨骼中关节旋转过度的倾向，尤其是同时加入多个区别动画的情况。例如，想象有一个角色左臂弯曲90°的目标动画。若我们再加入一个弯曲臂肘90°的区别动画，那么整体效果便是弯曲了$90° + 90° = 180°$。这样会令下臂插进上臂，当然不是正常的位置!

显然我们必须小心地选择参考片段，并决定它可以应用在哪些目标片段之上。以下是一些简单的经验法则。

- 在参考片段中，尽量减少髋关节的旋转。
- 在参考片段中，肩及肘关节应该一直维持中性姿势。那么把区别片段加入其他目标时，就能减轻手臂过度旋转的情况。
- 动画师应为每个核心姿势（站立、蹲下、躺下等）创建新的区别动画。那么动画师就能为这些姿势表现出现实人类应有的动作方式。

这些法则可以作为有用的起点，然而，如何制作及应用区别动画，只能从反复尝试及错误中学习，或从有这方面经验的动画师及工程师那学习。若读者的团队过去未使用过加法混合，应预期要花上大量时间学习加法混合的技艺。

11.6.6 加法混合的应用

11.6.6.1 站姿变化

加法混合的触目应用之一就是**站姿变化**（stance variation）。动画师对每个站姿创建1帧区别动画。当这些单帧片段用加法混合至一个基本动画时，角色的整个站姿就会戏剧性地发生变化，但角色又能继续表现原来所需的动作，如图11.38所示。

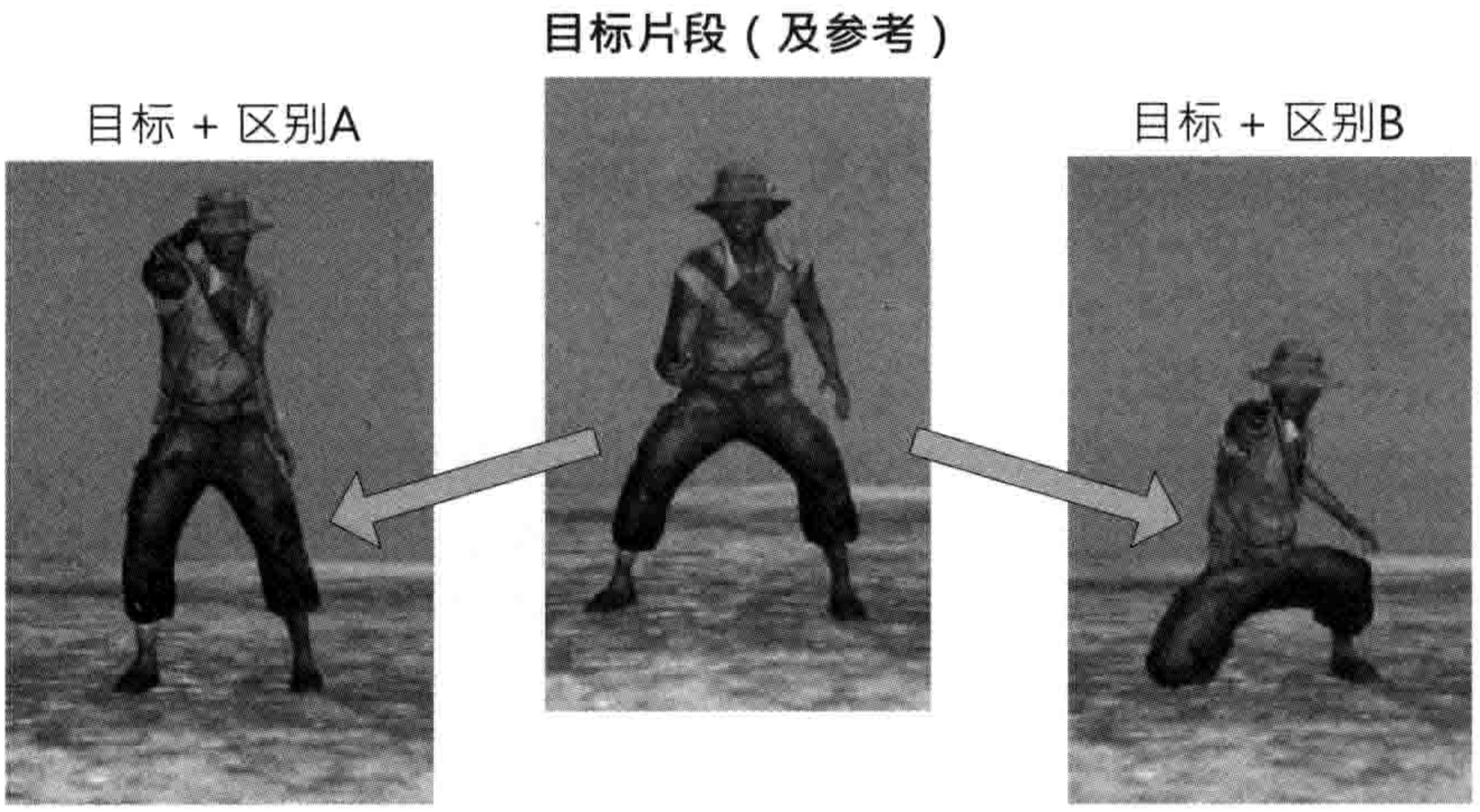


图 11.38: 两个单帧区别动画A和B，可令目标动画片段变成两个完全不同的站姿。（角色来自顽皮狗的《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune)》。）。

11.6.6.2 移动噪声

现实人类跑步时每个脚步不会完全一样——动作总是会有些变化的。人在分心的时候（例如在攻击敌人时）变化特别明显。加法混合可用于在完全重复的移动周期上叠加随机性、反应和分心的表现，如图11.39所示。

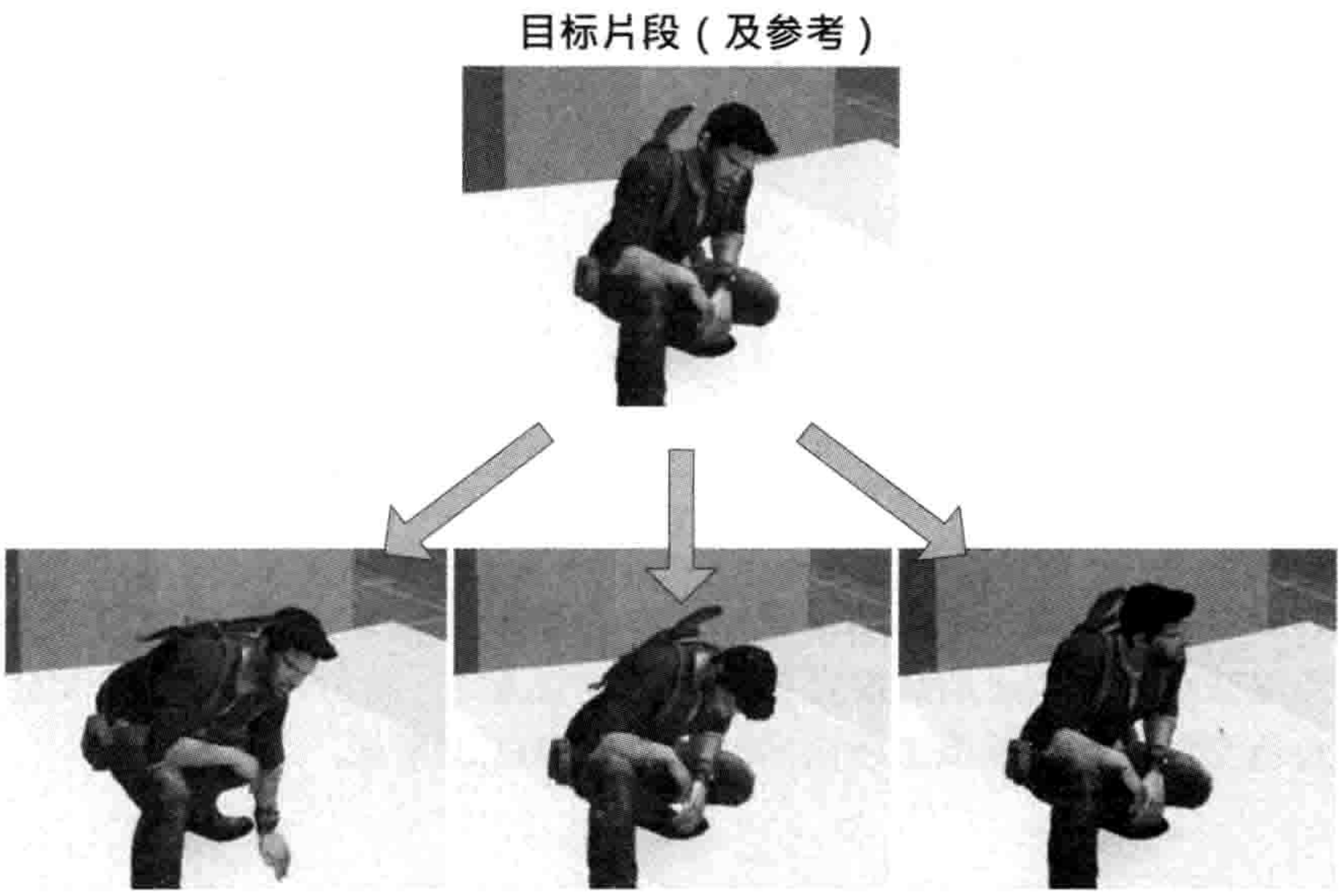


图 11.39: 加法混合可对重复的休闲动画加入变化。图片由顽皮狗提供。

11.6.6.3 瞄准及注视

另一加法混合常用之处在于，让角色注视四周或用武器瞄准。实现此效果时，首先用动画令角色执行一些动作，例如跑步，当中头部和武器都朝前面方向。然后，动画师改变头部或武器的方向至最右的角度，并储存该帧或多帧的区别动画。对最左、最上、最下方向重复此过程。那么这4个区别动画就能加法混合至原来的向前动画，产生向上下左右及之间的注视或瞄准动画。

瞄准的角度由每个片段的加法混合因子决定。例如，把向右的加法动画以100%混合，角色便会瞄准至最右的方向。混合50%向左加法动画的话，角色便会瞄准至最左方向和正面方向的中间。我们也可以再混合向上、向下加法动画，使角色向对角方向瞄准。图11.40展示了此应用。

11.6.6.4 时间轴的另类用途

其实有趣的是，动画片段的时间轴并不一定用于表示时间。例如，3帧动画片段可为引擎提供3个瞄准姿势——第1帧是向左瞄准的姿势、第2帧向前、第3帧向右。要令角色向右瞄准，我们可以把瞄准动画的局部时钟固定至第3帧。要产生50%向前和向右瞄准的混合，只

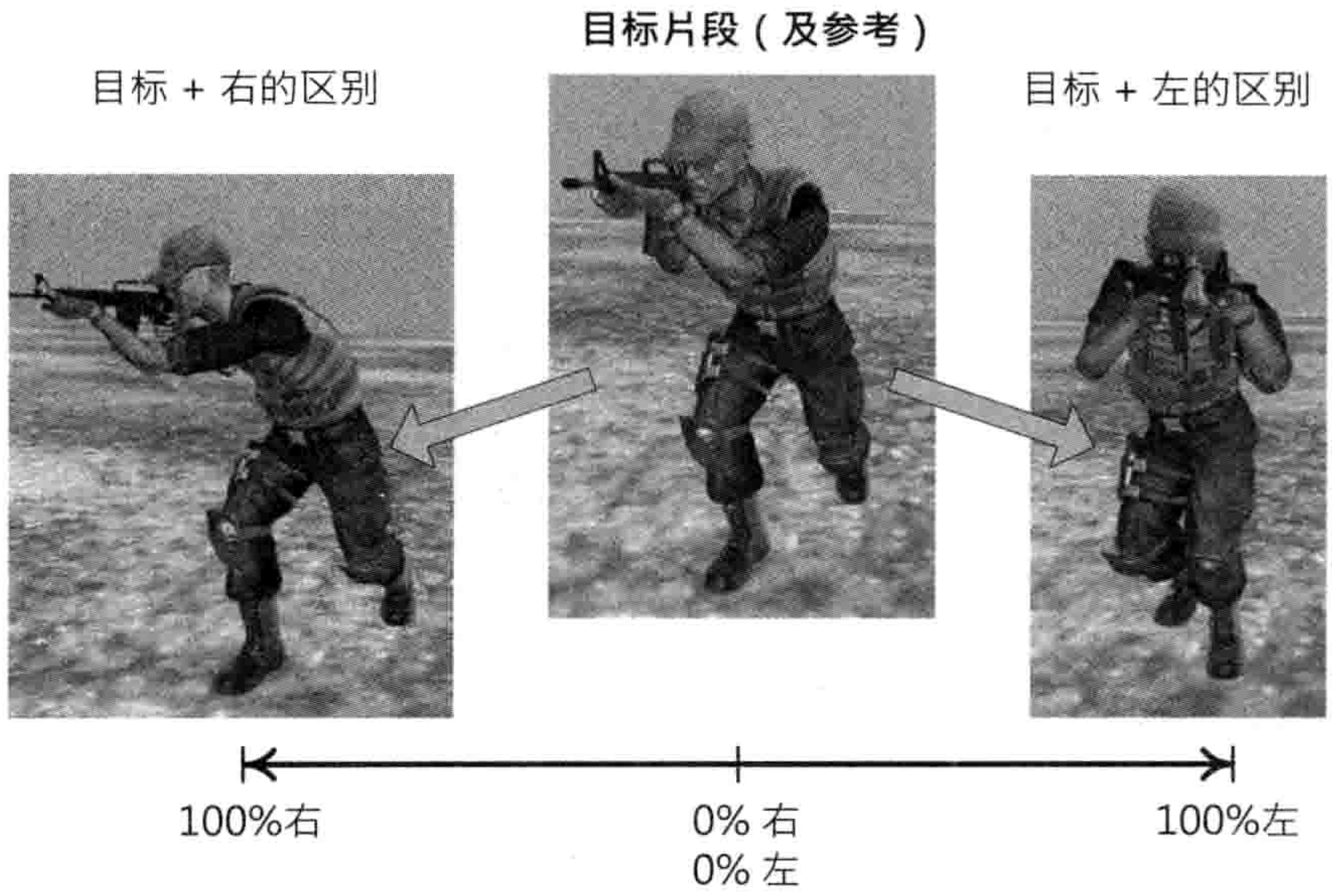


图 11.40: 使用加法混合作武器瞄准。图片由顽皮狗提供。

需把时钟拨至2.5帧。这是利用引擎现有功能来达成新目的的好例子。

11.7 后期处理

一旦一个或多个动画片段生成骨骼的姿势，然后通过线性插值或加法混合把结果混合成为一个姿势，在渲染角色之前，通常还需要再修改姿势。此修改称为**动画后期处理**（animation post-processing）。本节中，我们会看看几个常见的动画后期处理类型。

11.7.1 程序式动画

程序式动画（procedural animation）是指任何在运行时生成的动画，这些动画并非由动画工具（如Maya）导出的数据所驱动的。有时候，手工制作的动画片段用于设置骨骼最初的姿势，然后程序动画会作为后期处理的形式修改此姿势。

例如，想象有一个普通动画片段，用于令一车辆在崎岖的地形上行驶时显得颠簸。车辆行进的方向由玩家控制。我们希望当车辆转弯的时候，调整前轮和方向盘的转向令它们更显真实。这些调整可在动画产生姿势之后以后期处理方式进行。假设在原来的动画中，前轮是朝向正前方的，而方向盘则是正中位置。那么我们使用目前的转向角度创建一个依垂直轴旋

转的四元数，令前轮转向想要的角度。此四元数可乘以前轮的Q通道达至转向。同样，我们可生成依方向盘轴旋转的四元数，并把它乘以方向盘的Q通道令方向盘转向。这些调整于全局姿势计算及矩阵调色盘生成之前，在局部姿势间进行。

又例如，我们希望令游戏世界中的树木及灌木在风中自然摇曳，并且角色经过时会被拨开。要实现此效果，我们可以把树木和灌木建模为有简单骨骼的蒙皮网格。然后可以用程序动画取代手工动画，或在手工动画中加入程序动画，令那些关节自然地移动。我们可以在多个关节的旋转中加入一个或多个正弦波（sinusoid）[15]，模拟它们在风中摇曳。当角色经过含灌木或草丛的区域时，我们可以把植物根部的四元数以角色为中心向外偏转，令它们显得像被角色推开一样。

11.7.2 逆运动学

假设我们有一个动画片段是令角色弯腰拾取地上的物体。在Maya中，该片段看起来非常好；但在游戏关卡中，由于地面不是完全平坦的，有时候角色的手会碰不到物体，有时候又会穿过物体。在这种情况下，我们希望可以调整骨骼的最终姿势，令角色的手能完全与目标物体对齐。名为**逆运动学**（inverse kinematics，IK）的技术可以达成此事。

普通的动画片段是**正向运动学**（forward kinematics，FK）的例子。在正向运动学中，其输入是一组局部姿势，而输出是一个全局姿势，以及每关节的蒙皮矩阵。逆运动学的流程则是相反方向的：输入是某关节想要的全局姿势，此输入称为**末端受动器**（end effector）。我们要求出其他关节的**局部**姿势，使末端受动器能到达指定的位置。

数学上，IK可归结为**误差最小化**（error minimization）问题。如同其他最小化问题，问题可能会有一个解、多个解或无解。这是很符合直觉的：若要手握房间另一面的门柄，不走过去是无法做到的。要令IK发挥最好效果，开始时骨骼最好摆出接近目标的姿势。这样有助于算法专注“最接近”的解，并能在合理的时间内完成计算。图11.41展示了IK的工作情形。

想象一个含两个关节的骨骼，当中每个关节只能对一个轴旋转。这两个关节的旋转可写成二维角度矢量$\theta = [\theta_1 \quad \theta_2]$。两个关节的可行角度是一个集合，此集合所组成的二维空间称为**位形空间**（configuration space）。显然，对于更复杂、每关节含更多自由度的骨骼，位形空间会变成多维的，但无论有多少维的，这里所谈的概念同样适用。

现在我们对于每个关节旋转组合（即二维位形空间中的每个点）绘制一个三维图，该图的第3个轴是末端受动器和目标位置的距离。图11.42展示了此图的一个例子。此三维面的

[15] 译注：这里指以时间为参数的正弦函数，例如这种形式：$A\sin(\omega t - \phi)$，其中A是波幅，ω是角速度、t是时间、ϕ是相位。

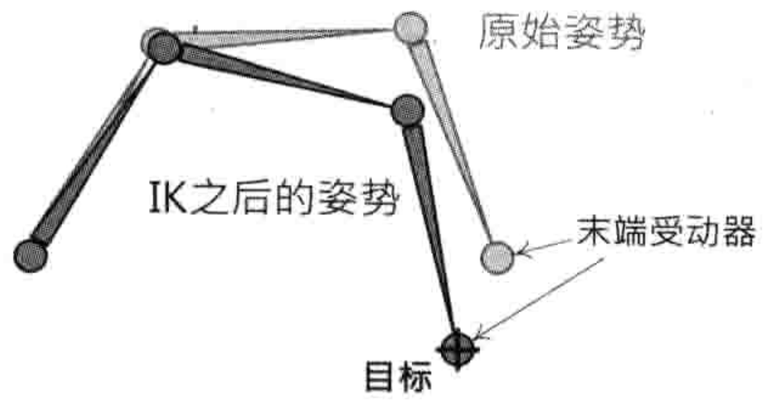


图 11.41: 逆运动学通过误差最小化，尝试把末端受动器关节移至目标的全局姿势。

“谷底”代表末端受动器最接近目标的地方。三维面上某点的高度为0，代表末端受动器已到达目标。逆运动学就是尝试找出此三维面的最小值（最低点）。[16]

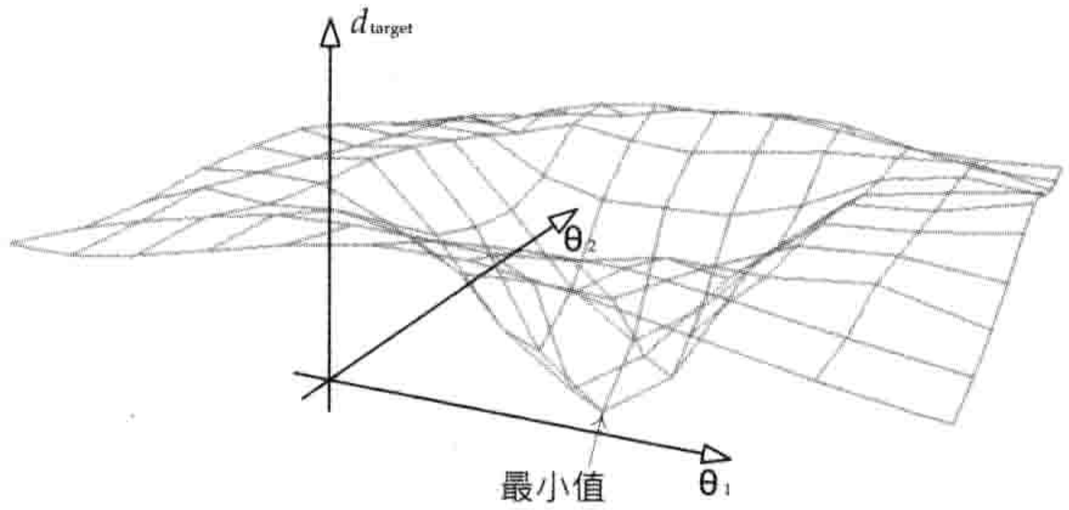


图 11.42: 此三维图绘画了二维位形空间中末端受动器与目标的距离。IK寻找这个距离的局部最小值。

这里不介绍反向动力学问题的求解细节。读者可参阅维基百科[17]或Jason Weber之文章《受限的反向动力学（Constrained Inverse Kinematics）》[40]。

11.7.3 布娃娃

当角色死去或失去意识之时，其身体会变得瘫软。在此情形下，我们希望该身体能与周边环境以真实的物理方式互动。为此我们可使用**布娃娃**（ragdoll）。布娃娃是一组由物理模拟的刚体，每个刚体代表角色的半刚体身体部分，例如下臂或上腿。这些刚体彼此受限于角色的关节位置，这些受限方式要设置成能产生自然的“无生气”身体移动。刚体的位置和定向都是由物理系统计算的，然后用于驱动角色骨骼中某几个重要关节的位置和定向。通常，把数据自物理系统传输至骨骼是一个后期处理步骤。

要真正了解布娃娃物理，须先理解碰撞及物理系统如何运作。12.4.8.7及12.5.3.8节会更深入讨论布娃娃。

[16]译注：当然，反向动力学还需要找出从当前位置移动至最低点的最短路径。

[17]http://en.wikipedia.org/wiki/Inverse_kinematics

11.8 压缩技术

动画数据可占去大量内存。单个关节姿势可能由10个浮点数通道（3个用作平移、4个用作旋转，再加上最多3个用作缩放）组成。由于每个通道含4字节浮点数，以每秒30个样本的1s片段便需要4字节 × 10通道 × 30样本每秒 = 1200字节每关节，也即等于每关节每秒1.17KB的数据流量。对于由100个关节组成的骨骼（以现今标准来说算是很少的关节），无压缩的动画会占每秒117KB。若游戏含1000s的动画（在现今游戏中这是偏低的估算），那么整个数据集就是庞大的114.4MB。多数游戏没有如此大的空间储存此数据量，以PlayStation 3为例，它只有256MB主存及256MB显存。因此，游戏工程师需为此投放精力，压缩动画数据，以最少的内存成本提供最丰富及多元化的动作。

11.8.1 通道省略

降低动画片段尺寸的简单方法之一就是省略无关的通道。多数角色都不需要非统一缩放，因此3个缩放通道可缩减至单个统一缩放通道。有些游戏甚至可以省去所有关节的缩放通道（可能除了面部的一些关节）。人形角色的骨头通常是不能伸缩的，所以大部分关节的平移通道也可略去，只有根关节、面部关节，或一些颈关节需要保留。最后，因为四元数要一直保持归一，所以只需储存3个分量（如x、y、z），第4个分量（如w）可在运行时重建。

作为更进一步的优化，若一些姿势在整个动画片段期间没有变化，那么可以只存储该姿势位于时间$t = 0$的第一个样本，再加一位的标记表示该通道所有其他t值都是常数。

通道省略可大幅降低动画片段的尺寸。一个无缩放、无平移、含100关节的骨骼只需要303个通道，这包括每个关节的四元数所需的3个通道，以及根关节平移的3个通道。可以对比一下，若100个关节都包含10个通道，那么总共要1000个通道。

11.8.2 量化

另一个降低动画尺寸的方法是缩减每通道的尺寸。浮点小数值正常会储存为32位IEEE格式。此格式提供23位的尾数精确度，以及8位的指数。然而，在动画片段中我们经常并不需要保持这种精确度及范围。储存四元数时，可保证其通道值的范围必然是$[-1, 1]$。当一个32位IEEE浮点数的绝对值为1时，其指数是0，而且23位精确度能准确至7个小数位。经验告诉我们，四元数可以仅用16位精确度编码，因此我们若为每通道使用32位浮点数其实白白浪费了16位。

把32位IEEE浮点数转换成n位整数表示法的运算称为**量化**（quantization）。实际上此运算有两部分：**编码**（encode）是把原来的浮点小数值转换为量化后的整数表示法的过程，**解码**（decode）是把量化整数还原为原来浮点数的近似值。（我们只能够还原一个**近似值**——量化是**有损**压缩（lossy compression），因为它实质上降低了用于表示一个值的精确度位数。）

要把浮点数编码成整数，我们首先要把合法范围切割成N个同等大小的**区间**。然后我们找出某浮点数值属于哪一个区间，并用该区间的**整数索引**值表示该值。解码时，我们只需简单地把整数索引转换至浮点数格式，并用偏移及缩放把该值还原至原来的范围。选择N的大小时，通常会选用对应于n位整数能表示的整数值范围。例如，若把32位浮点数值编码为16位整数，那么区间的数目便会是$N = 2^{16} = 65536$。

Jonathan Blow在《游戏开发者杂志》的“The Inner Loop”专栏撰写了一篇关于浮点标量量化的优秀文章[18]（该文的源代码也可下载[19]）。该文介绍了编码过程中从浮点数映射至区间的两个方法：我们可以把浮点数**截尾**（truncate）至紧接的最低区间边界（**T编码**），或是可以把浮点数**舍入**（round）至包围区间之中值（**R编码**）。类似地，该文描述了从整数表示法重建浮点数的两个方法：我们可以传回原值映射到的区间的**左值**（**L重建**），或是传回区间的中值（**C重建**）。这样我们有4种编码解码的可行方法：TL、TC、RL、RC。当然，应避免用TL及RC，因为这种组合会趋向增加或减少数据中的能量，这通常会产生灾难性后果。TC的好处在于时间上高效，但也会有些严重问题——无法准确地表示0值。（若为`0.0f`编码，解码后会是一个细小的正数。）因此RL通常是最好的选择。我们将会在此示范这种方法。

该文中只谈及量化正浮点数，并且在例子中，为简单起见，输入范围都假设是$[0, 1]$。然而，我们必然可以用偏移及缩放令任何浮点范围变成$[0, 1]$。例如，四元数通道的范围是$[-1, 1]$，但我们可以把这些值加1再除以2，令数值变成$[0, 1]$的范围。

以下一个函数把$[0, 1]$范围的输入浮点数值编码至n位整数，另一个则解码还原，两个函数都是根据Jonathan Blow的RL法编写的。量化值会以32位无符号整数（`U32`）传回，但实际上只会用到由`nBits`参数指定的最低n个有效位。例如，若传入`nBits==16`，那么就可以安全地把结果转型至`U16`。

```
U32 CompressUnitFloatRL(F32 unitFloat, U32 nBits)
{
    // 基于要求的输出位数，判断区间数量
    U32 nIntervals = 1u << nBits;
```

[18] http://number-none.com/product/Scalar%20Quantization/index.html

[19] http://www.gdmag.com/src/jun20.zip

```
    // 把输入值从[0, 1]范围缩放至[0, nIntervals - 1]范围
    // 这里需要减1是由于我们希望最大的输出值能储存于nBits个位之内
    F32 scaled = unitFloat * (F32)(nIntervals - 1u);

    // 最后，我们需要加0.5f，再四舍五入至最近的区间中点
    // 然后，把该值截尾，取得区间索引(通过转型至U32)
    U32 rounded = (U32)(scaled + 0.5f);    // 译注：原文误写为 *

    // 为无效的输入值做出保护
    if (rounded > nIntervals - 1u)
        rounded = nIntervals - 1u;
    return rounded;
}

F32 DecompressUnitFloatRL(U32 quantized, U32 nBits)
{
    // 基于编码时的位数，判断区间数量
    U32 nIntervals = 1u << nBits;

    // 解码只需简单地把U32转成F32，并按区间大小缩放
    F32 intervalSize = 1.0f / (F32)(nIntervals - 1u);

    F32 approxUnitFloat = (F32)quantized * intervalSize;
    return approxUnitFloat;
}
```

要处理任意在$[min, max]$范围内的输入值，我们可使用以下这些函数：

```
U32 CompressFloatRL(F32 value, F32 min, F32 max, U32 nBits)
{
    F32 unitFloat = (value - min) / (max - min);
    U32 quantized = CompressUnitFloatRL(unitFloat, nBits);
    return quantized;
}

F32 DecompressFloatRL(U32 quantized, F32 min, F32 max, U32 nBits)
{
    F32 unitFloat = DecompressUnitFloatRL(quantized, nBits);
    F32 value = min + (unitFloat * (max - min));
    return value;
}
```

我们回到最初的动画通道压缩问题。要压缩及解压四元数的4个分量至每通道16位，我们只需调用CompressFloatRL()及DecompressFloatRL()，传入$min = -1$、$max =$

1、$n = 16$:

```
inline U16 CompressRotationChannel(F32 qx)
{
    return (U16)CompressFloatRL(qx, -1.0f, 1.0f, 1.0f, 16u);
}

inline F32 DecompressRotationChannel(U16 qx)
{
    return DecompressFloatRL((U32)qx, -1.0f, 1.0f, 16u);
}
```

平移的压缩比旋转稍棘手，因为和四元数通道不同，平移通道的范围在理论上是无界的。好在于实践中角色的关节不会移得很远，我们可设定一个合理的移动范围，若出现超越该合法范围的动画便报错。游戏内置电影是对此规则的一个特例，当游戏内置电影在世界空间制作动画，角色根关节的平移值可以变得很大。为解决此问题，我们可以以动画或关节为单位，针对每片段实际的最大平移值，选择合法的平移范围。由于每个动画或每个关节中有不同的数据范围，我们必须把这些范围储存在压缩片段数据中。这样会增加每个动画的数据，因此是否值得要做出权衡。

```
// 我们使用2m的范围，你可能使用不同的
F32 MAX_TRANSLATION = 2.0f;

inline U16 CompressTranslationChannel(F32 vx)
{
    // 钳制至合法范围……
    if (value < -MAX_TRANSLATION)
        value = -MAX_TRANSLATION;
    if (value >  MAX_TRANSLATION)
        value =  MAX_TRANSLATION;

    return (U16)CompressFloatRL(vx,
        -MAX_TRANSLATION, MAX_TRANSLATION, 16);
}

inline F32 DecompressTranslationChannel(U16 vx)
{
    return DecompressFloatRL((U32)vx,
        -MAX_TRANSLATION, MAX_TANSLATION, 16);
}
```

11.8.3 采样频率及键省略

动画数据偏大的原因有三：第一，每个关节的姿势含最多10个通道；第二，骨骼含大量的关节（人形角色需100个或以上）；第三，角色的姿势通常采用高频率采样（例如每秒30帧）。我们已看过第1个问题的一些解决方案。另一方面，我们几乎不能减少高分辨率角色的关节数目，因此我们无法解决第2个问题。要对付第3个问题，可以做如下两件事。

- **降低整体的采样率**：有些动画以每秒15帧导出，效果还是可以的。这么做能使动画数据大小降为一半。
- **省略一些样本**：若片段在某个时间区间中，通道数据的变化大约呈线性，那么我们可以省略此区间中除首尾以外的所有样本。然后在运行时，使用线性插值还原删掉了的样本。

第2个技术比较复杂，而且需要在每个样本上储存关于**时间**的信息。这些额外数据会蚕食我们从一开始通过省略样本而节省的数据量。然而，有些游戏引擎曾成功地使用此技术。

11.8.4 基于曲线的压缩

笔者在工作生涯中使用过的，数一数二最强、最容易使用的，各方面都优良的动画API是Rad Game Tools公司的Granny[20]。Granny并不是用等距的姿势样本序列来储存动画的，取而代之，它采用一组n阶非均匀、非有理B样条描述关节S、Q、T通道随时间的路径。使用B样条的好处是可用少量数据点为高曲率的通道编码。

Granny导出的动画数据，也如其他传统动画数据一样，在等距的时间点上对关节姿势采样。然后，Granny会对每个通道的样本数据集用B样条拟合，拟合的容忍度由用户设置。最终产生的动画片段的尺寸会比一般均匀采样、线性插值的片段显著减小。图11.43说明了此过程。

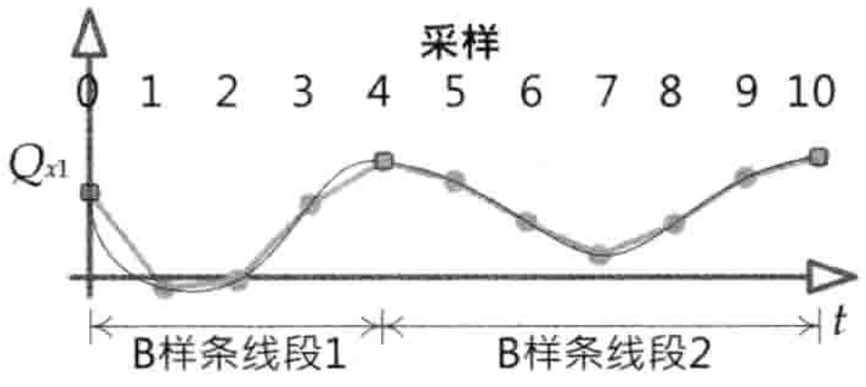


图 11.43: 动画压缩的方法之一，是采用B样条拟合动画通道数据。

[20]译注：该产品的正式名字是Granny 3D。Granny是老奶奶的意思。

11.8.5 选择性载入及串流

最省内存的动画片段就是不在内存的片段。多数游戏不需要所有动画同时置于内存。有些片段只应用在某角色职业，所以如果某关卡中不会遇到该职业的角色，那些片段便不用载入。另外有一些动画片段在游戏中只会使用一次，这些片段可以在需要播放之前才载入或串流，播放结束后就从内存释放。

多数游戏会在游戏开始时载入一组核心动画片段，并一直把它们保留在内存。这些片段包括玩家角色的核心移动集，以及在游戏中不断应用的物体动画，例如武器及补给品。所有其他动画则通常是按需载入的。有些游戏引擎按逐个动画片段载入，但很多游戏引擎会把动画片段打包成逻辑组，并以此为单位载入及卸载。

11.9 动画系统架构

我们已了解游戏动画系统里的理论，现在我们转向讨论从软件架构的立足点如何架构这种系统。我们会探讨在典型的游戏引擎中，动画系统和其他系统间有哪些类型的接口。

多数动画系统由3个分明的软件层所组成。

- **动画管道**（animation pipeline）：对于游戏中每个含动画的角色及物体，动画管道为它们取得一个或多个动画片段及对应的混合因子作为输入，把这些片段混合后产生一个局部骨骼姿势作为输出。动画管道也会为骨骼计算一个全局姿势，以及生成蒙皮矩阵调色板供渲染引擎使用。动画管道通常会提供后期处理钩子，以便在生成全局姿势及蒙板矩阵前可以修改局部姿势。此处可将逆运动学、布娃娃物理，以及其他形式的程序动画施于骨骼之上。
- **动作状态机**（action state machine，ASM）：游戏角色的动作（站、行、跑、跳等）通常最好建模为有限状态机，此状态机常称为**动作状态机**。ASM子系统位于动画管道之上，并提供以状态驱动的动画接口供所有高层游戏代码之用。ASM确保角色能从一个状态圆滑地**过渡**至另一状态。此外，多数动画引擎容许角色身体的不同部分同时做不同、独立的事情，例如边跑边瞄准边开火。要实现此功能，可通过**状态层**（state layer）使用多个独立的状态机控制单个角色。
- **动画控制器**（animation controller）：在许多游戏引擎中，玩家与非玩家角色的行为最终是由**动画控制器**所组成的高级系统控制的。每个控制器是特别为管理某个角色行为模式而设的。例如一个控制器用于角色在开放空间移动及战斗中处理其行为（“边走边打”模式），其他控制器可能用于躲避、驾车、爬梯等。这些高级动画控制器能封装

绝大部分的动画相关代码，令高级的玩家控制及AI逻辑不会因动画的微观管理而变得杂乱。

有些游戏引擎会以上述不同的方式分割软件层。其他引擎可能会把两层或以上融合成一个系统。然而，所有动画引擎都需要以某种形式执行这些工作。以下几节中，我们会按照这个软件层探索动画系统的架构，若例子中的游戏引擎采用不同的方式我们会加以注释。

11.10 动画管道

底层动画引擎所做的运算，构成了一个把输入（动画片段及混合设置）变换成输出（局部及全局姿势、渲染用的矩阵调色板）的管道。此管道的各个阶段如下。

1. **片段解压及姿势提取：**在此阶段中，每个片段的数据会被解压，并且提取所需时间索引的静态姿势。此阶段的输出是每个输入片段一个局部骨骼姿势。此姿势可能包含每个关节的姿势信息（**全身的姿势**），或仅含部分关节的信息（**分部姿势**），或可能是用作加法混合的**区别姿势**。
2. **姿势混合：**在此阶段中，通过全身LERP混合、分部LERP混合，及/或加法混合，把输入姿势结合在一起。本阶段的输出是一个对应骨骼中所有关节的局部姿势。只有当需要混合超过一个动画片段时才需要执行本阶段，否则只需直接使用阶段1的输出。
3. **全局姿势生成：**此阶段遍历骨骼层次结构，把局部关节串接以产生骨骼的全局姿势。
4. **后期处理：**这是一个可选的阶段，使输出最终姿势之前，有机会修改骨骼的局部及/或全局姿势。后期处理用于逆运动学、布娃娃物理，以及其他形式的程序动画。
5. **重新计算全局姿势：**许多种类的后期处理都需要全局姿势作为输入，但却只生成局部姿势作为输出。当执行了这种后期处理步骤，我们必须从修改后的局部姿势重新计算全局姿势。显然，若某后期处理运算并不需要使用全局姿势信息，该运算可于第2及第3阶段之间运行，那么就可以避免重新计算全局姿势。
6. **矩阵调色板生成：**生成最终全局姿势后，本阶段把每个关节的全局姿势矩阵乘以对应的逆绑定姿势矩阵。本阶段的输出为供渲染引擎所用的蒙皮矩阵调色板。

图11.44展示了一个典型的动画管道。

11.10.1 数据结构

每个动画管道都有不同的架构，但它们的数据结构都会和本节所示的有相似之处。

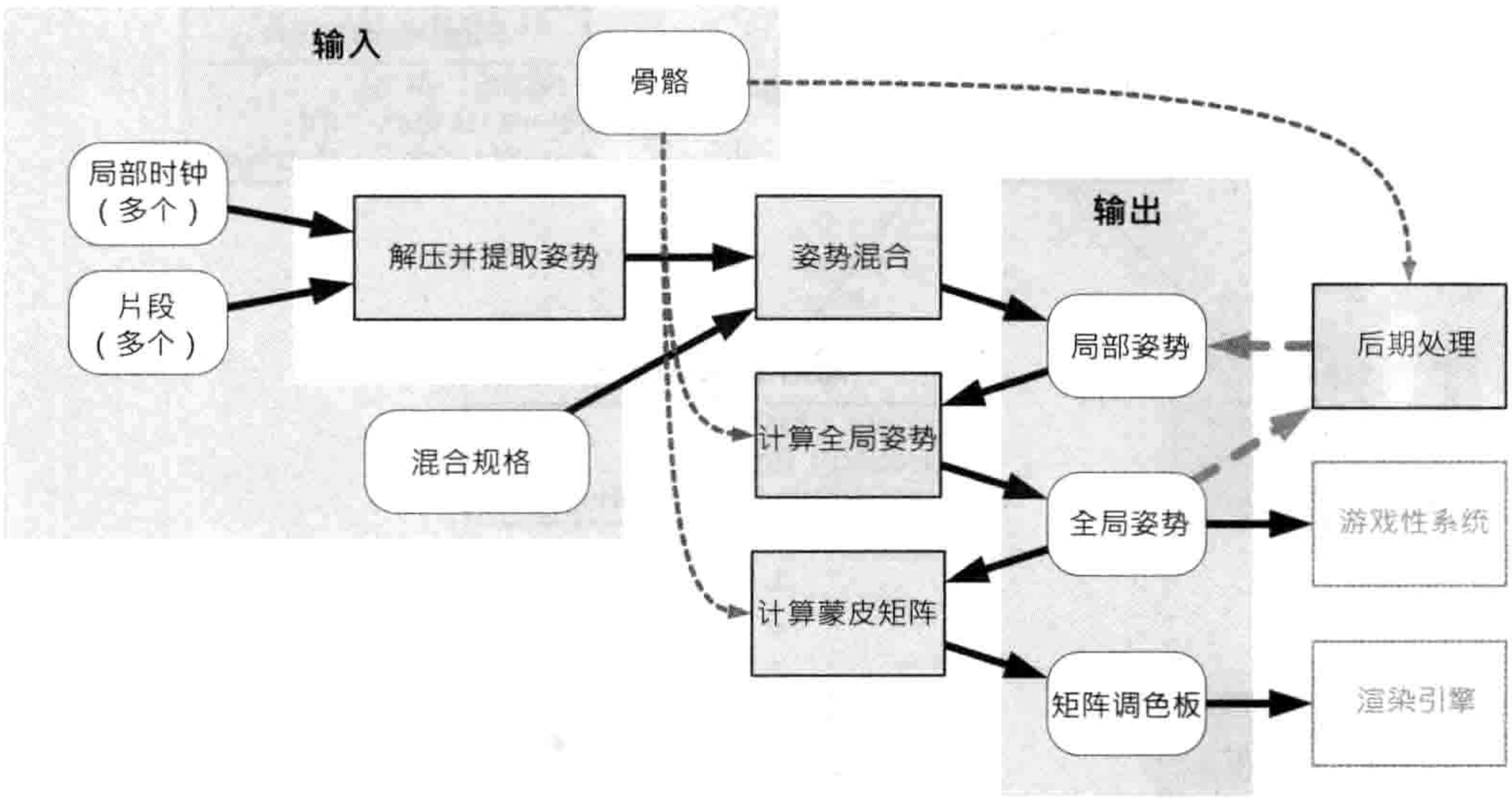


图 11.44: 典型的动画管道。

11.10.1.1 共享资源数据

如同所有游戏引擎系统，我们必须清楚分开**共享资源数据**（shared resource data）及**每实体状态信息**（per-instance data structure）。游戏中每个个别角色或物体有其每实体数据结构，但相同类型的角色或物体都会共享一组资源数据。这些共享数据通常包括：

- **骨骼**：骨骼描述关节层次结构及其绑定姿势。
- **蒙皮网格**：一个或多个可蒙皮至单个骨骼的网格。蒙皮网格中的每个顶点都包含一个或多个关节索引，加上那些关节对该顶点的影响力。
- **动画片段**：为每个角色骨骼制作的数百甚至数千个动画片段。这些动画片段可以是全身片段、分部片段或用于加法混合的区别片段。

图11.45是这些数据结构的UML图。特别注意这些类之间关系的**基数**（cardinality）及**方向**。在UML图中，基数置于类间的关系箭头及箭尾之旁，1代表类的一个实体，星号代表多个实体。例如，某种类的角色会引用一个骨骼、一个或多个网格、一个或多个动画片段。当中，骨骼是统合各方的元素——蒙皮绑定至骨骼，但蒙皮和动画片段没有关系。同样，动画片段应用至某特定骨骼，但它们完全不“知悉”蒙皮网格。图11.46说明了这些关系。

游戏设计师通常会尽量令骨骼数目减至一个或仅几个，因为每个新骨骼通常需要一组全新的动画片段。为了令游戏看起来有许多不同类型角色的错觉，通常会尽量制作多个绑定至单个骨骼的网格，使所有角色能共享同一组动画。

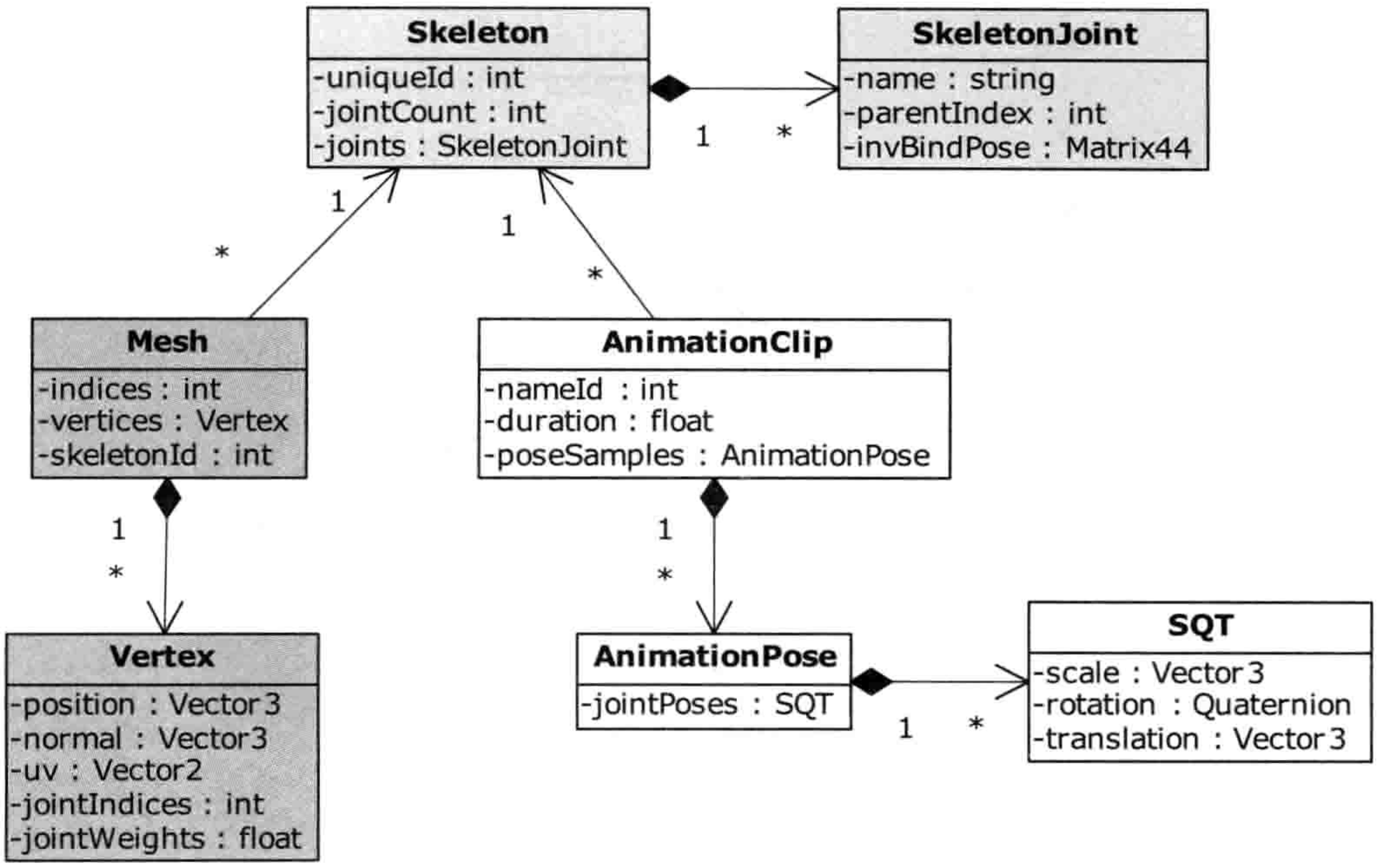


图 11.45: 共享动画资源的UML图。

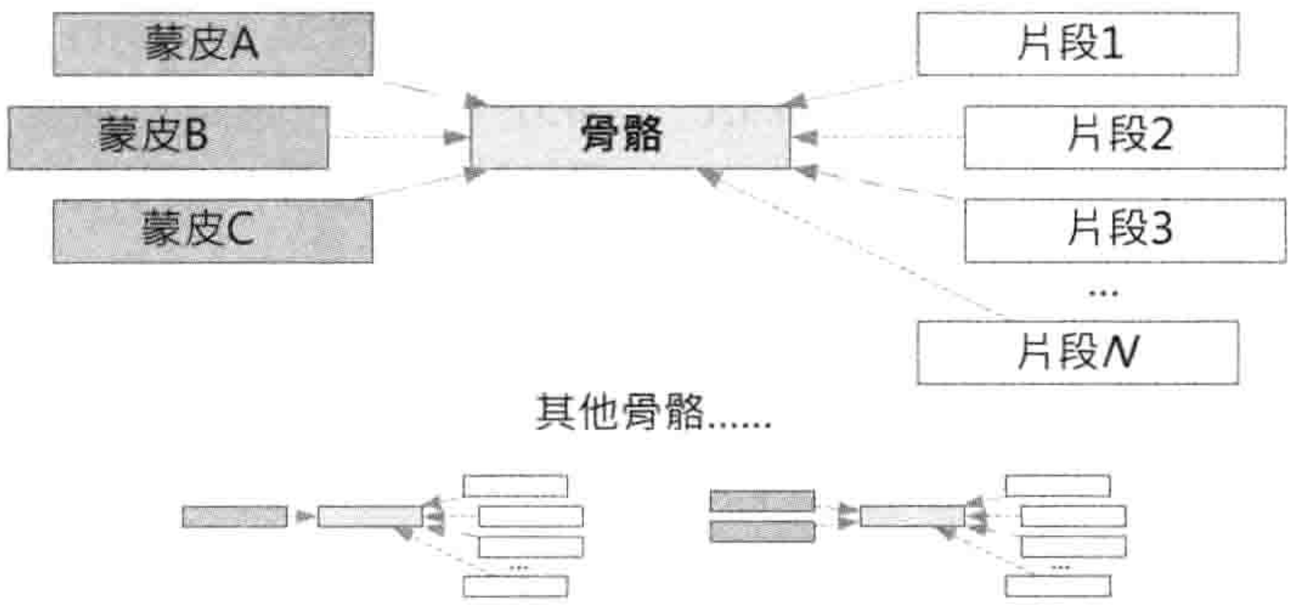


图 11.46: 多个动画片段及一个至多个网格都指向单个骨骼。

11.10.1.2 每实例数据

在大多数游戏中，每个角色种类的多个**实体**可在同屏上出现。某个角色种类的每个实体都需要其私有数据结构，以记录当前播放中的动画片段、片段混合的方式（如果有超过一个片段），以及其当前的骨骼姿势。

现在还没表示每实例数据的统一方法。然而，几乎所有动画引擎会记录以下的信息。

- **片段状态**：每个播放中的片段都需要维护以下信息。
  - **局部时钟**：片段的局部时钟描述其局部时间线上的一点，该时间点用于提取当前姿势。在一些游戏引擎中，局部时钟由全局起始时间所替代。（11.4.3节含局部时钟及全局时钟的比较。）
  - **播放速率**：片段可以任何速率播放，此变量在11.4.2节中表示为R。
- **混合规格**：混合规格描述哪些动画片段正在播放，以及这些片段如何混合在一起。每个片段通过混合权重设置它对最终姿势的影响。主要有两种方式描述片段混合的方式：**统一加权平均法**及**混合树**。当使用树状方式时，混合树的结构会视为**共享资源**，而当中的混合权重则储存为**每实例**状态数据。
- **分部骨骼关节权重**：若应用分部骨骼混合，每个节点对最终姿势的影响力会储存为一组**关节权重**。一些动画引擎的关节权重是二元的——关节有影响力或是没有影响力。另一些动画引擎可设置权重为0（无影响力）至1（完全影响）。
- **局部姿势**：局部姿势通常是一个SQT数组的数据结构，当中每个SQT对应一个关节，储存成相对于父关节的骨骼最终姿势。此数组也可能会用于储存中间姿势，以作为管道中后期处理阶段的输入及输出。
- **全局姿势**：全局姿势可以是SQT、4×4或4×3的数组，当中每个元素对应一个关节，储存模型空间或世界空间的最终骨骼姿势。全局姿势可能会用作后期处理的输入。
- **矩阵调色盘**：矩阵调色盘是4×4或4×3矩阵，当中每个元素对应一个关节，储存蒙皮矩阵，供渲染引擎之用。

11.10.2 扁平的加权平均混合表示法

就算是最入门级的游戏引擎，也会支持某种形式的动画混合。这意味着在某一指定时间，多个动画片段能对角色骨骼的最终姿势产生影响。描述如何混合作用中的片段，最简单的方法之一就是使用**加权平均**。

在此方法中，每个动画片段会对应一个混合权重，此权重描述该片段的角色最终姿势的影响力。引擎维护一个**作用**动画片段的扁平列表（即非零权重的片段）。要计算骨骼的最终姿势，我们首先从N个作用片段中，对每个片段在恰当的时间索引上提取姿势。然后，对于骨骼中每个关节，我们简单地从N个作用动画提取到的平移矢量、旋转四元数、缩放因子计算N点加权平均数。这就能产生骨骼的最终姿势。

N个矢量$\mathbf{v}_i$的集合之加权平均方程如下：

$$\mathbf{v}_{\text{avg}} = \frac{\sum_{i=0}^{N-1} w_i \mathbf{v}_i}{\sum_{i=0}^{N-1} w_i}$$

若权重已**归一化**，即它们之和为1，此方程可简化为：

$$\mathbf{v}_{\text{avg}} = \sum_{i=0}^{N-1} w_i \mathbf{v}_i \qquad (\text{当} \sum_{i=0}^{N-1} w_i = 1)$$

在$N = 2$的情况下，若我们设$w_1 = \beta$及$w_0 = (1 - \beta)$，加权平均便会变成熟悉的对两矢量的线性插值（LERP）：

$$\begin{aligned}\mathbf{v}_{\text{LERP}} &= \text{LERP}(\mathbf{v}_A, \mathbf{v}_B, \beta) \\ &= (1 - \beta)\mathbf{v}_A + \beta \mathbf{v}_B\end{aligned}$$

我们可以简单地把相同的加权平均公式应用至四元数，只要视四元数为含4个分量的矢量。

11.10.2.1 例子：OGRE

OGRE的动画系统完全以这种方式运作。`Ogre::Entity`代表某个三维网格的一个实例（例如游戏世界中的某个步行中的角色）。此类聚集一个`Ogre::AnimationStateSet`，而`AnimationStateSet`又维护一组`Ogre::AnimationState`，当中每个`AnimationState`对象对应一个作用中的动画。以下是`Ogre::AnimationState`的代码片段（为清楚起见，已剔除一些不相关的细节）：

```
/** 表示动画片段的状态，以及该片段对角色整体姿势的影响权重
*/
class AnimationState
{
protected:
    String      mAnimationName; // 片段的引用
    Real        mTimePos;       // 局部时间位置
    Real        mWeight;        // 混合权重
    bool        mEnabled;       // 本动画是否播放中
    bool        mLoop;          // 应否循环播放

public:
    /// 取得本动画的名字
    const String& getAnimationName() const;
```

```
    /// 取得本动画的(局部)时间位置
    Real getTimePosition() const;

    /// 设置动画的(局部)时间位置
    void setTimePosition(Real timePos);

    /// 取得本动画的权重(影响力)
    Real getWeight() const;

    /// 设置本动画的权重(影响力)
    void setWeight(Real weight);

    /// 修改时间位置，调整动画长度。若启用循环，此方法会引致循环
    void addTime(Real offset);

    /// 若动画到达局部时间线的终点，并且不循环，传回true
    bool hasEnded() const;

    /// 传回启动状态
    bool getEnabled() const;

    /// 设置启动状态
    void setEnabled(bool enabled);

    /// 取得动画应否循环播放的状态
    bool getLoop() const;

    /// 设置动画应否循环播放的状态
    void setLoop(bool loop);
};
```

每个AnimationState记录了动画片段的局部时间及其混合权重。当为某Ogre::Entity计算骨骼最终姿势时，OGRE动画系统简单地遍历其AnimationStateSet中的每个作用AnimationState。对每个状态，它会按照该状态的局部时间所计算出来的时间索引，从该状态对应的动画片段中提取骨骼姿势。最后，对于骨骼中的每个关节，动画系统为其平移矢量、旋转四元数、缩放因子计算N点加权平均，以产生最终骨骼姿势。

OGRE及播放速率

有趣的是，OGRE并无播放速率（R）的概念。若它有此概念，我们应该能在Ogre::AnimationState看见像这样的数据成员：

```
Real     mPlaybackRate;
```

当然，我们仍然可以简单地把传入addTime()函数的时间进行缩放，令动画片段播放得更快或更慢。但遗憾的是，OGRE没有直接支持动画时间缩放的功能。

11.10.2.2 例子：Granny

Rad Game Tools的Granny动画系统[21]提供了一个扁平的加权平均动画混合系统，该系统与OGRE的相似。Granny容许在同一角色身上同时播放任意数目的动画。每个作用动画的状态由granny_control数据结构所维护。Granny计算加权平均来产生最终姿势，并会自动地令所有作用片段的权重归一化。在此意义上，Granny的架构简直和OGRE的动画系统一模一样。然而，Granny突出之处在于其处理时间的方式。Granny采用11.4.3节所述的全局时间方式，此方式可以令每个片段循环任意次数，或无限循环。片段也可以在时间上缩放，负数的时间缩放因子可以令动画倒转播放。

11.10.3 混合树

有些动画引擎并不是以扁平加权平均描述混合方式的，而是采用混合操作树，我们将会探讨其中原因。动画混合树（animation blend tree）是编译理论中的**表达式树**（expression tree）或**语法树**（syntax tree）的例子。该树的内节点是运算符，而叶节点则是那些运算符的输入。（更正确的说法是，内节点表示文法的**非终止符** / non-terminal，而叶节点则表示**终止符** / terminal。）在以下几个小节中，我们会重温11.6.3及11.6.5节提及的多种动画混合方式，并看看这些混合方式怎样表示为表达式树。

11.10.3.1 二元LERP混合

如11.6.1节所述，二元线性插值（LERP）混合从两个输入姿势混合成一个输出姿势。混合权重（β）控制第2个输入姿势显示于输出姿势的百分比，而$(1-\beta)$则是第1个姿势的百分比。此混合可以表示为图11.47中的二叉表达式树。

11.10.3.2 泛化一维LERP混合

在11.6.3.1节中，我们得悉泛化一维LERP混合可以很方便地混合任意数量的片段，当中的片段置于一条线性的轴之上。混合因子b指明在此轴上的所需混合。这种混合可表示

[21] http://www.radgametools.com/granny.html

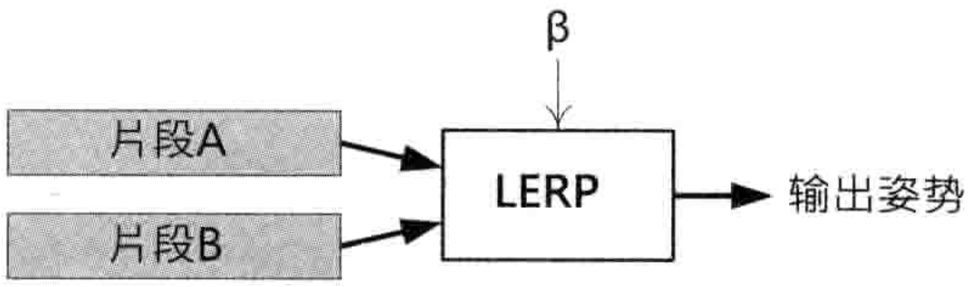


图 11.47: 以二元表达式树表示二元LERP混合。

为n个输入的运算符，如图11.48所示。

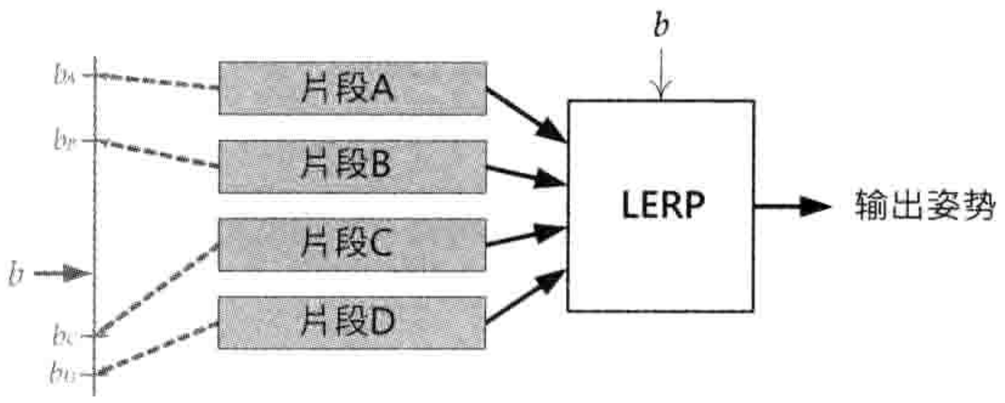


图 11.48: 以多个输入的表达式树表示泛化的一维混合。指定一个混合因子b之后，这种树总是可以转换为二元表达式树。

给定一个b值，这种线性混合总是可变为一个二元LERP混合。我们只需使用最接近b的两个片段作为二元混合的输入，并以方程(11.12)计算β值。图11.48显示了此运算符。

11.10.3.3 简单二维LERP混合

11.6.3.2节描述了可通过层叠两个二元LERP混合的结果实现二维LERP混合。给定一个二维混合点$\mathbf{b} = [b_x \quad b_y]$，图11.49显示了这种混合如何用树的形式表示。

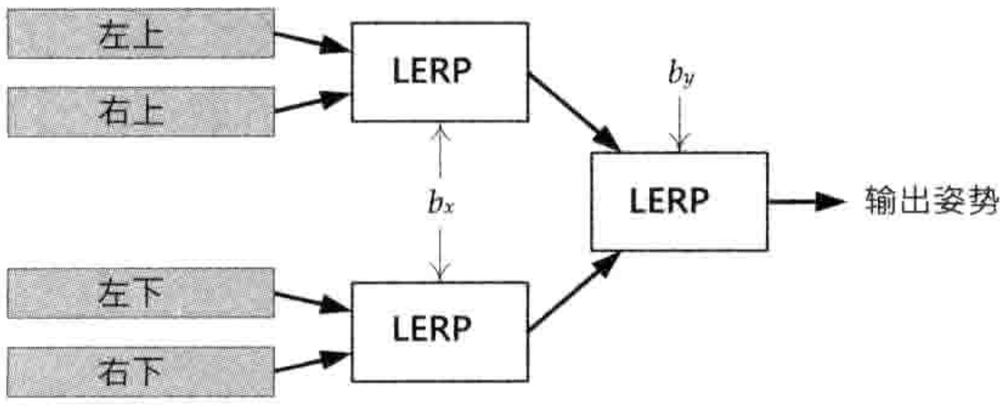


图 11.49: 以层叠二元混合实现简单的二维LERP混合。

11.10.3.4 三角LERP混合

11.6.3.3节介绍了三角LERP混合，该混合使用重心坐标α、β及$1-\alpha-\beta$作为混合权重。为了以树的方式表示这种混合，我们需要三元（ternary，3个输入）表达式树节点，如图11.50所示。

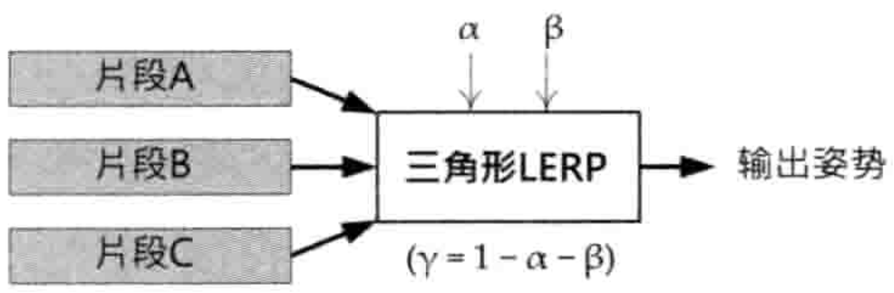


图 11.50: 以三元表达式树表示三角LERP混合。

11.10.3.5 泛化三角LERP混合

在11.6.3.4节中，我们介绍了通过在二维平面上任意位置放置片段来指定的泛化三角LERP混合。给定平面上的一点$\mathbf{b} = [b_x \quad b_y]$就能生成所需的输出姿势。这种混合可表示为含任意输入的树节点，如图11.51所示。

泛化三角LERP混合总是可转换成一个三叉树，其方法是使用Delaunay三角剖分找出包含点$\mathbf{b}$的三角形。然后，把该点转换为重心坐标α、β及$1-\alpha-\beta$，并使用这些坐标作为三元混合节点的混合权重，三角形顶点对应的3个片段则作为节点的输入，如图11.51所示。

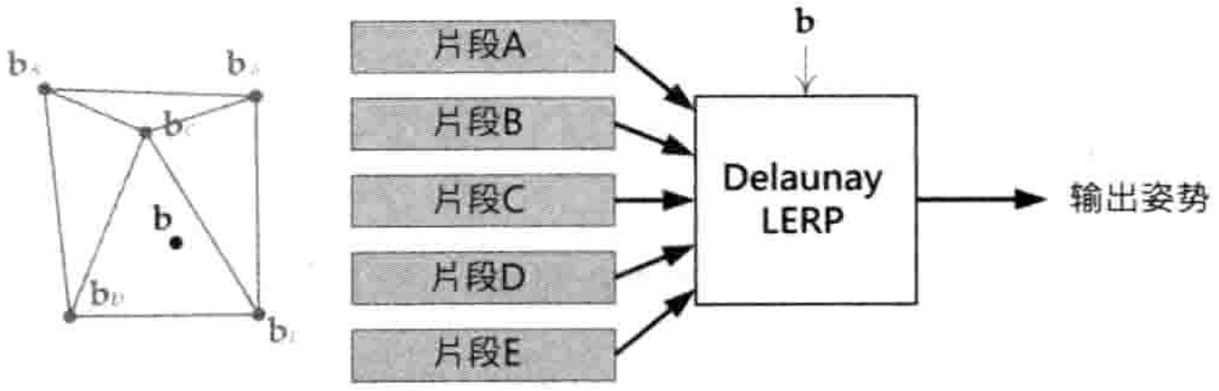


图 11.51: 多个输入的表达式树节点可表示泛化二维混合。通过Delaunay三角化总是能把这种树转换为三元树。

11.10.3.6 加法混合

11.6.5节描述了加法混合。加法混合是一个二元操作，因此可表示为图11.52中的二叉树节点。混合权重β控制区别动画在输出的程度——当$\beta = 0$，区别片段完全不影响输出；当$\beta = 1$，区别动画对输出产生最大影响力。

加法混合节点必须谨慎处理，因为其输入是不可互换的（大部分其他混合种类是可以的）。两个输入之一是正常的骨骼姿势，而另一个输入则是**区别姿势**（也称为**加法姿势**）。区别姿势**只可**施于正常姿势，而加法混合结果是另一个正常姿势。这也意味着，加法混合必须为一个叶节点，而其他的混合可以是叶节点或内节点。若我们要把多个区别动画施于角色，我们必须使用层叠式的二叉树，并把区别动画置于区别动画的输入，如图11.53所示。

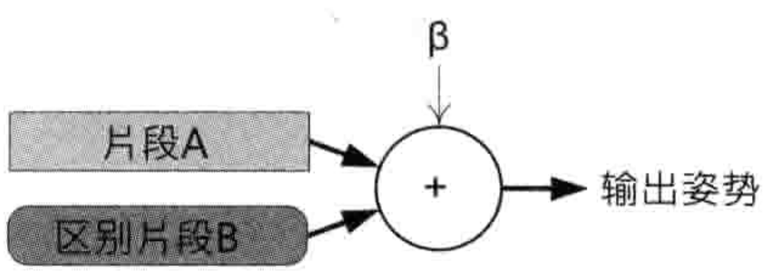


图 11.52: 以二元树表示加法混合。

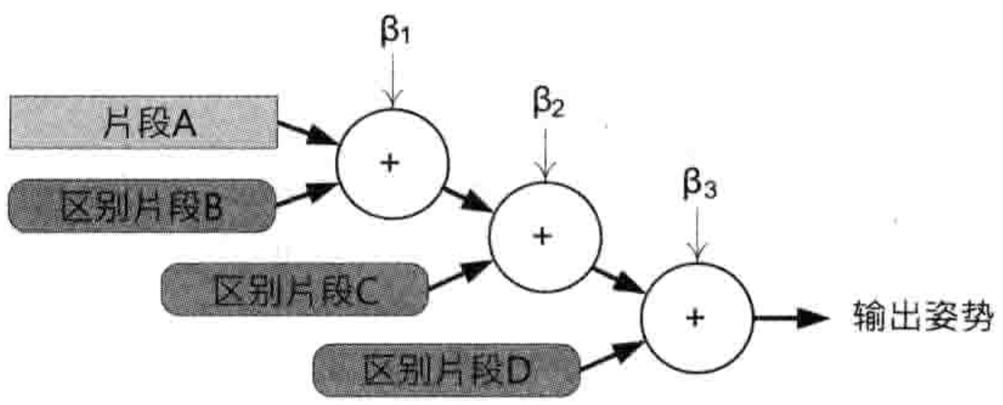


图 11.53: 为了加法混合多个区别姿势至正常“基础”姿势，必须使用层叠式二元表达式树。

11.10.4 淡入 / 淡出架构

如11.6.2.2节所提及的，动画间的淡入/淡出通常是把之前的动画与之后的动画做线性插值来实现的。淡入 / 淡出可用两种方式实现，视乎你的动画引擎采用扁平加权平均架构，还是表达式树架构。在本节中，我们会看看这两种实现。

11.10.4.1 扁平加权平均的淡入 / 淡出

采用扁平加权平均架构的动画引擎中，淡入 / 淡出是由调整片段权重本身实现的。回想权重$w_i = 0$的片段并不会影响角色的当前姿势，而非零权重的片段则会一起计算平均产生

最终姿势。若我们希望从片段A圆滑地过渡至片段B，只需简单地逐渐提升片段B的权重w_B，并同时逐渐降低片段A之权重w_A，如图11.54所示。

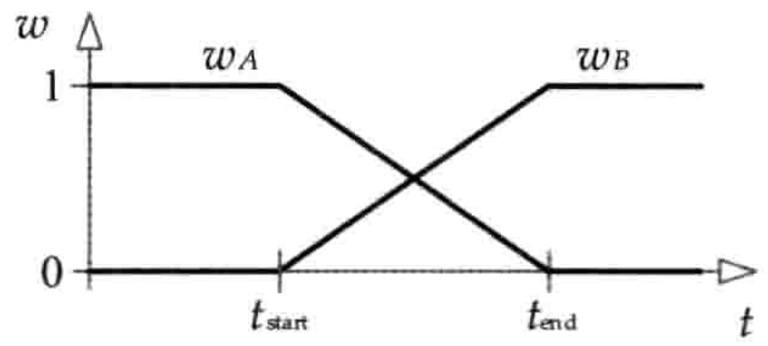


图 11.54: 在加权平均动画架构中，实现片段A至片段B的淡入 / 淡出。

然而，当我们需要从一个复杂的混合过渡至另一个复杂的混合，采用加权平均架构的淡入 / 淡出便会有点棘手。例如，我们希望把角色从步行过渡至跳跃，假设步行动作是由片段A、B、C平均而成的，而跳跃动作则由平均D及E片段而成。

我们希望圆滑地从步行过渡至跳跃，而不仅影响步行及跳跃动作本身的样子。因此在过渡时，我们要逐渐降低ABC的权重，同时逐渐调升DE的权重，并要保持ABC和DE内的**相对**权重维持不变。设淡入 / 淡出的混合因子为λ，要满足此要求，我们可以简单地按原来的方式**同时**设置这两组片段的权重，然后把来源组中的权重乘以$(1-\lambda)$，目标组中的权重乘以λ。

以下看一个实际例子，以说服我们这是正确的做法。假设从ABC过渡至DE之前，非零的权重为$w_A = 0.2$、$w_B = 0.3$、$w_C = 0.5$。而在过渡之后，非零的权重为$w_D = 0.33$、$w_E = 0.66$。因此我们对权重的设定为：

$$\begin{aligned} w_A &= (1-\lambda)(0.2), & w_D &= \lambda(0.33), \\ w_B &= (1-\lambda)(0.3), & w_E &= \lambda(0.66), \\ w_C &= (1-\lambda)(0.5) \end{aligned} \tag{11.17}$$

在以上的方程中，读者应能说服自己以下事项。

1. 当$\lambda = 0$，输出姿势是A、B、C片段的正确混合，D、E片段不影响输出。
2. 当$\lambda = 1$，输出姿势是D、E片段的正确混合，A、B、C片段不影响输出。
3. 当$0 < \lambda < 1$，ABC组及DE组内的相对权重仍然正确，虽然组内权重之和并不是1。（事实上，ABC组的权重之和为$1-\lambda$，DE组的权重之和为λ。）

要令这种方法正确运作，实现上需要记录片段的逻辑分组（虽然在底层所有片段仍然是由一个大扁平数组所维护的，例如OGRE中的`Ogre::AnimationStateSet`）。在以上的例子中，系统必须“得悉”A、B、C分为一组，D、E是另一组，以及我们希望从ABC组过

渡至DE组。这需要在扁平片段状态数组以上，再维护一些额外的元数据。

11.10.4.2 表达式树的淡入 / 淡出

在基于表达式树的动画引擎中实现淡入 / 淡出，相比加权平均架构更为直观。无论要从一个片段至另一个片段，还是从一个复杂混合过渡至另一混合，方法始终如一：我们只需在淡入 / 淡出时，在混合树根节点加入一个新的二叉LERP节点。

如前文，我们把淡入 / 淡出节点的混合因子表示为λ。淡入 / 淡出混合节点上方的输入为来源树（可以是一个片段或复杂混合），下方输入为目标树（也可以是片段或复杂混合）。在过渡期间，λ由0逐渐提升至1。当达至$\lambda = 1$时，过渡便告完成，那个淡入 / 淡出混合节点以及上方的输入树便可功成身退。剩下的下方输入树变成整个混合树的根节点，从而完结过渡。图11.55展示了此过程。

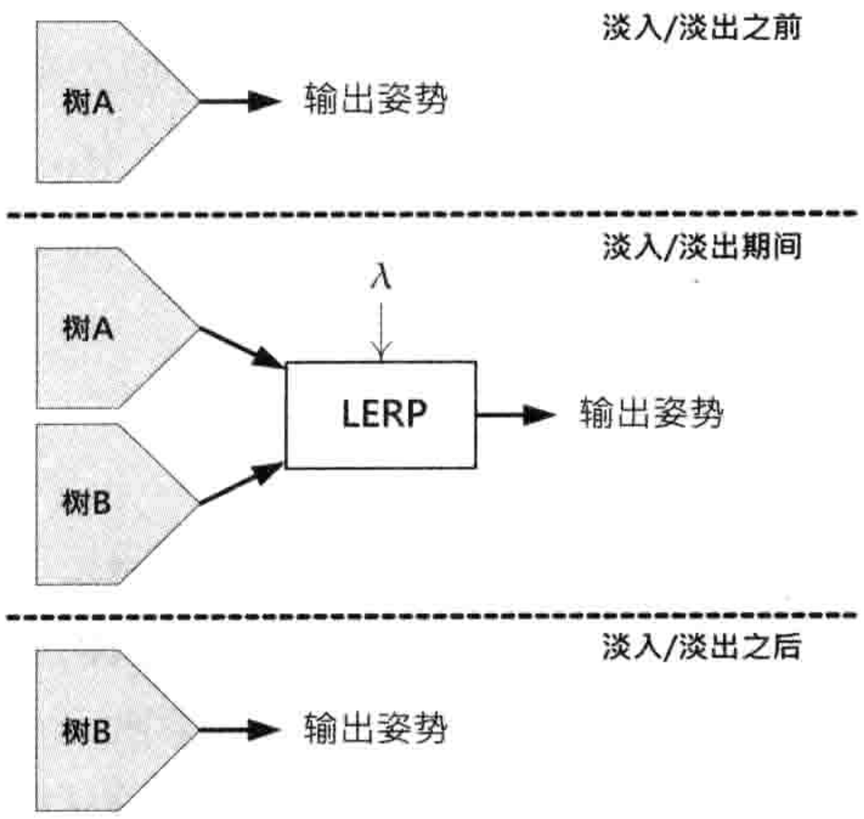


图 11.55: 两个任意混合树之间的淡入 / 淡出。

11.10.5 动画管道的优化

优化是动画管道的关键要素。有些管道暴露出它们所有的优化本质细节，那么就会把正确优化的责任交给调用的代码。另一些管道在方便的API之下封装大部分的优化细节，但即使这样，API还是需要用某方式设计，令所需的优化能在背后实现。

动画管道的优化通常和游戏运行的硬件架构相关。例如，现在的硬件架构中，内存存取模式会大大影响代码的效率。必须尽量避免缓存命中失败及load-hit-store，以确保代码以最快速度运行。但是在另一些硬件，浮点运算可能是瓶颈，那么代码应该设计成尽

量利用SIMD矢量运算。每个硬件平台对程序员有其独特的优化挑战。因此，有些动画管道API是非常专门地针对某平台而设的。另一些管道则尝试提供一个可在不同平台用不同方式优化的API。以下我们看一些平台优化的例子。

11.10.5.1 PlayStation 2上的优化

PS2内有一个非常快的内存区域，称为**便笺内存**（scratch pad）。PS2也有一个高速的**直接内存访问**（direct memory access，DMA）控制器，可以高效地把数据复制至便笺内存，或从便笺内存复制至主存。有些动画管道利用了此硬件架构，把所有动画混合置于便笺内存上执行。当要混合两个骨骼姿势时，便把这些数据通过DMA从主存传输至便笺内存。混合后，其结果也是写到便笺内存的另一缓冲区内。最后，结果姿势便会通过DMA传回主存。

PS2的DMA控制器可与主CPU并行地传输数据。因此，要把吞吐量最大化，PS2程序员经常要想办法令CPU和DMA控制器同时有工作。有时候，要达至此目的的最好方法是使用**批次风格**API，当中游戏会把动画混合的请求排列成一个很大的队列，然后才开始执行。那么动画管道便可以同时令CPU和DMA控制器的使用率最大化，因为这能令大量的姿势请求传给管道，请求之间不含“空档”，甚至可在DMA传送一些请求的同时处理不相关的请求。

11.10.5.2 PlayStation 3上的优化

如7.6.1.2节所述，PS3有6个称为**协同处理器**（synergistic processing unit，SPU）的特殊处理器。SPU执行多数代码时比主CPU（即**Power处理器**/Power processing unit/PPU）更快。每个SPU也含256KB的特快**局部存储**内存，供其独占使用。和PS2相似，PS3也有一个强大的DMA控制器，可以在各处理器执行计算工作的同时，并行地在主存和SPU内存间传输数据。若要在PS3上实现一个理想的动画管道，SPU处理的数据越多越好，并且令PPU和SPU尽量无须闲置地等待DMA传输。

此架构令动画管道的API和PS2相似，相似之处在于需要把动画请求以批次形式进行，令请求能高效地交织执行。此外，PS3的动画API通常会使用**动画作业**（animation job）的概念，因为**作业**（job）是SPU上的基本执行单位。

11.10.5.3 Xbox及Xbox 360上的优化

Xbox及Xbox 360皆采用统一内存架构（unified memory architecture），而非采用专门的内存区域及DMA控制器的传输区域间的数据。所有处理器包括主CPU（360中的是3个PowerPC核）、GPU，以及其他硬件系统都是共享一整块主存。

理论上，Xbox架构需要一组和PlayStation完全不同的优化方式，因而应该会有非常不同的动画API。然而，Xbox可以作为一个例子，说明**有时候**在一个平台上的优化也可对其他平台有所帮助。事实证明，Xbox和PlayStation在缓存命中失败及load-hit-store的内存存取模式会引致严重的性能下降。因此，两个系统最好都能尽量令动画数据在主存中局部化。若动画管道以批次形式处理动画，并在较小的内存区域内进行运算（如PS2的便笺内存及PS3的SPU内存），那么也会有利于Xbox的统一内存架构。我们并非总能达到这种平台间的协同效应，每个平台仍需专门的优化。然而，若有这种机会，我们应该尽量把握。

作为一条经验法则，我们应该在效能最有限的平台上优化引擎。当这些优化后的代码移植至其他限制较少的平台时，很有可能那些优化仍然会有效，最坏的情况下也只会对效能有少许反效果。若倒转次序，把在最不严峻的平台所优化的代码移植至最严峻的平台，结果几乎必然不能产生最严峻平台下的最优效能。

11.11 动作状态机

底层管道等于动画系统中的Direct3D或OpenGL——底层管道很重要，但直接供游戏代码使用会有所不便。因此，为方便起见，在底层管道和游戏角色/其他动画系统客户端之间，通常会引入一个软件层。此软件层通常会实现为状态机，称为**动作状态机**（action state machine）或**动画状态机**（animation state machine，ASM）。

ASM置于动画管道之上，它能以直接了当、状态驱动的方式控制游戏中角色的动作。ASM也负责确保状态间能以平滑、自然的方式过渡。有些动画引擎容许使用多个独立的状态机控制角色不同方面的动作，例如全身的运动、上半身的姿态、面部动画等。这通常是以状态层的概念实现的。在本节中，我们会探讨如何架构一个典型的动画状态机。

11.11.1 动画状态

ASM中每个状态对应一个任意复杂的动画片段混合。在混合树架构中，每个状态对应至某个预先定义的混合树。而在扁平加权平均架构中，一个状态代表一组片段及一组相对权重。由于按照混合树来思考会比较方便及容易表达，我们在接下来的讨论中会以混合树为例。然而，只要不涉及加法混合或四元数SLERP运算，我们在此所述的都能用扁平加权平均方式实现。

按游戏设计的需求，对应至动画状态的混合树可以很简单也可以很复杂（只要能合乎引擎的内存及效能限制）。例如，一个“空闲（idle）”的状态可能只对应一个全身动画，而一

个“跑步中”的状态可能会对应至一个半圆混合，当中“左走”、“前奔”、“右走”片段各置于−90°、0°、+90°的位置。而“跑步中开火”状态会包含半圆方向混合，加上瞄准武器用的加法混合或分部骨骼混合，再加上额外的混合供角色用眼、头、肩去注视目标。除此以外，还可加入更多的加法动画，控制角色走动时的整体姿势、步态、步距，并加上一些随机的变化令角色更显“人性化”。

11.11.1.1 状态及混合树规格

动画师、游戏设计师和程序员通常会一起合作创造游戏里中心角色的动画及控制系统。这些开发者需要一种方式描述角色ASM的状态，编排每个混合树的结构，并在混合树中选择片段做输入。虽然状态及混合树可以是硬编码的，但多数现在的游戏引擎会提供**数据驱动**（data-driven）的方式定义动画状态。数据驱动的目标是让用户创建新的动画状态，移除不需要的状态，微调现存的状态，并能相当快地看到修改后的结果。换句话说，数据驱动动画引擎的中心目标是**快速迭代**（rapid iteration）。

用户输入动画状态的方式有很多种。有些引擎采用简单、最基础的方式，把动画状态以简单语法写于文本文件里。另一些引擎提供华丽[22]的图形编辑器，建构动画状态的方式是通过拖曳原子性元件（如片段及混合节点）至画布上，并把它们以任意方式连接在一起。这种编辑器通常可提供角色实况预览，令用户即时见到角色在最终游戏中的样子。笔者认为，具体选用哪种方式对最终游戏的质量只有少许影响，最重要的是使用者能相当容易快速地修改及看见修改后的结果。

11.11.1.2 自定义混合树节点类型

我们只需要4种原子混合节点类型便可构建任意复杂的混合树。这4种节点类型包括片段、二元LERP混合、二元加法混合、三元（三角）LERP混合。几乎所有可想象到的混合树都可由这几种原子节点构成。

仅由原子节点构成的混合树很快会变得又巨大又笨重。因此为方便起见，许多游戏引擎容许预定义一些自定义的复合节点类型。11.6.3.4节及11.10.3.2节所提及的N维线性混合节点是复合节点的例子。我们可以想象无数个复杂的混合节点类型，每个节点都是为解决某个游戏中的独特问题而设的。足球游戏可能需要定义一个节点令角色运球。战争游戏可能要定义一个特殊节点处理瞄准武器及开火。格斗游戏可能要为角色每个打击动作定义自定义节点。当能够定义自定义节点类型时，可做之事无可限量[23]。

[22]译注：此处原文使用形容词slick，也有华而不实、花巧的负面意思。作者似乎比较推崇他们引擎采用的文本方式。

[23]译注：原文为谚语“the sky's the limit”。

11.11.1.3 例子：顽皮狗的神秘海域引擎

顽皮狗的《神秘海域：德雷克船长的宝藏（Uncharted: Drake's Fortune）》及《神秘海域：纵横四海（Uncharted: Among Thieves）》采用一个简单的、基于文本的方式描述动画状态。因顽皮狗有丰厚的Lisp语言历史背景，神秘海域引擎使用了一个客制化的Scheme程序语言（Scheme本身是一种Lisp方言）描述状态。状态规格有两个基本类型：**简单**及**复杂**。

简单状态

简单状态含一个动画片段。例如：

```
(define-state simple
    :name "pirate-b-bump-back"
    :clip "pirate-b-bump-back"
    :flags (anim-state-flag no-adjst-to-ground)
)
```

不要被Lisp风格的语法吓跑。这整段代码仅仅定义了一个名为“pirate-b-bump-back”的状态，其动画片段的名称也是“pirate-b-bump-back”。而`:flags`参数容许用户在状态中设定多个布尔选项。

复杂状态

复杂状态含有LERP或加法混合所组成的树。例如，以下的状态定义了一棵树，该树含有一个二元LERP混合节点，以及两个作为输入的片段（“walk-l-to-r”及“run-l-to-r ”）：

```
(define-state complex
    :name "move-l-to-r"
    :tree
        (anim-node-lerp
            (anim-node-clip "walk-l-to-r")
            (anim-node-clip "run-l-to-r")
        )
)
```

代码中的`:tree`参数可让用户任意构建混合树，混合树以LERP、加法混合节点及播放个别片段的节点所构成。

在此我们可以知道上面`(define-state simple ...)`的例子背后的真实工作方式，它可能只是定义了一个复杂混合树，内含一个“片段”节点而已：

```
(define-state complex
    :name "pirate-b-unimog-bump-back"
    :tree (anim-node-clip "pirate-b-unimog-bump-back")
    :flags (anim-state-flag no-adjust-to-ground)
)
```

以下的复杂状态展示如何把混合节点层叠成任意深度的混合树：

```
(define-state complex
    :name "move-b-to-f"
    :tree
        (anim-node-lerp
            (anim-node-additive
                (anim-node-additive
                    (anim-node-clip "move-f")
                    (anim-node-clip "move-f-look-lr")
                )
                (anim-node-clip "move-f-look-ud")
            )
            (anim-node-additive
                (anim-node-additive
                    (anim-node-clip "move-b")
                    (anim-node-clip "move-b-look-lr")
                )
                (anim-node-clip "move-b-look-ud")
            )
        )
)
```

此段代码对应图11.56中的树。

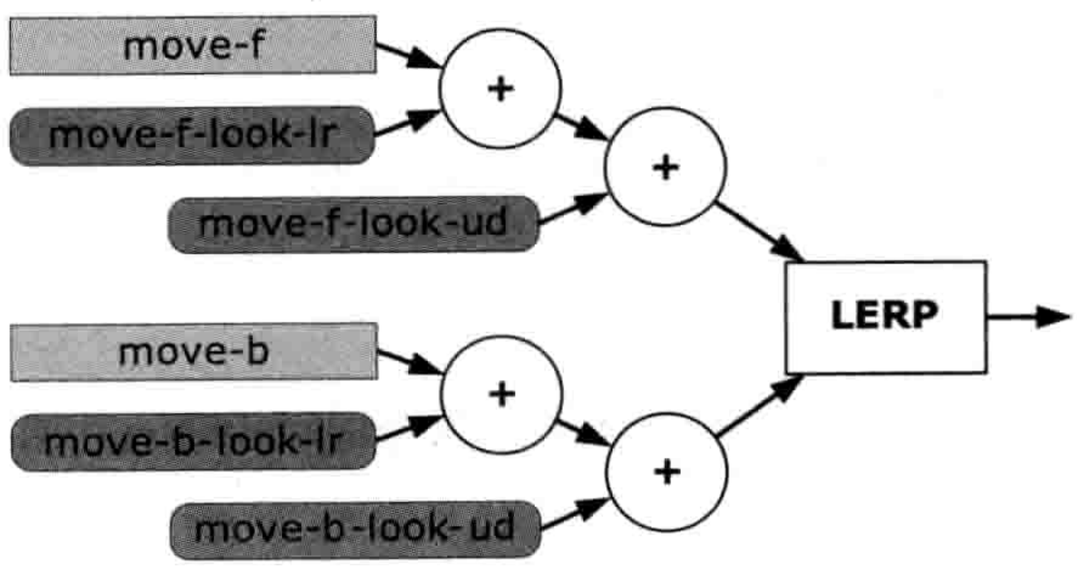


图 11.56: 对应“move-b-to-f”状态例子的混合树。

自定义树语法

感谢Scheme的强大宏语言，用户可利用基本片段、LERP及加法混合节点定义自定义混合树。这样就能定义多个状态，这些状态几乎等同但有不同输入片段或其他变化。例如，上面例子中的“move-b-to-f”复杂状态可通过宏来分部定义：

```
(define-syntax look-tree
    (syntax-rules ()
        ((look-tree base-clip look-lr-clip look-ud-clip)
            ;; 这是表达“每当编译器见到(look-tree b lr ud)形式的代码时，
            ;; 用以下代码取代之……”
            (anim-node-additive
                (anim-node-additive
                    (anim-node-clip base-clip)
                    (anim-node-clip look-lr-clip)
                )
                (anim-node-clip look-ud-clip)
            )
        )
    )
)
```

而原来的“move-b-to-f”状态能以这个宏重新定义：

```
(define-state complex
    :name "move-b-to-f"
    :tree
        (anim-node-lerp
            (look-tree "move-f" "move-f-look-lr" "move-f-look-ud")
            (look-tree "move-b" "move-b-look-lr" "move-b-look-ud")
        )
)
```

(look-tree ...)宏可用于定义任意数量的状态，只要那些状态需要相同的树结构但不同的动画片段输入。状态也可以用各种方式组合“look-tree”。

快速迭代

《神秘海域（Uncharted）》通过两个重要工具达到快速迭代。游戏内置的动画观察工具可以通过菜单产生（spawn）游戏角色，并控制其动画。此外，我们有一个命令行工具，负责重新编译动画脚本，并令运行中的游戏重新载入编译结果。调整角色动画时，只要修改含该动画规格的文本文件，再重新载入动画状态，便可在游戏中迅速看到改动后的效果。

11.11.1.4 例子：虚幻引擎3

虚幻引擎3（Unreal Engine 3，UE3）为动画系统提供了一个图形用户界面。如图11.57所示，UE3的动画混合树含有一个AnimTree根节点。此节点有3个输入：动画（animation）、变形（morph），以及一个称为**骨骼控制**（skel control）的特别节点。动画输入可连接至任意复杂的混合树（那么姿势就会从右至左流动，此方向和本书的惯例相反）。而变形输入则是连接驱动角色的基于变型目标动画，常用面部动画。最后，骨骼控制输入用于各种后期处理，例如反动力学。后期处理是在动画树及/或变形树生成姿势后才进行的。

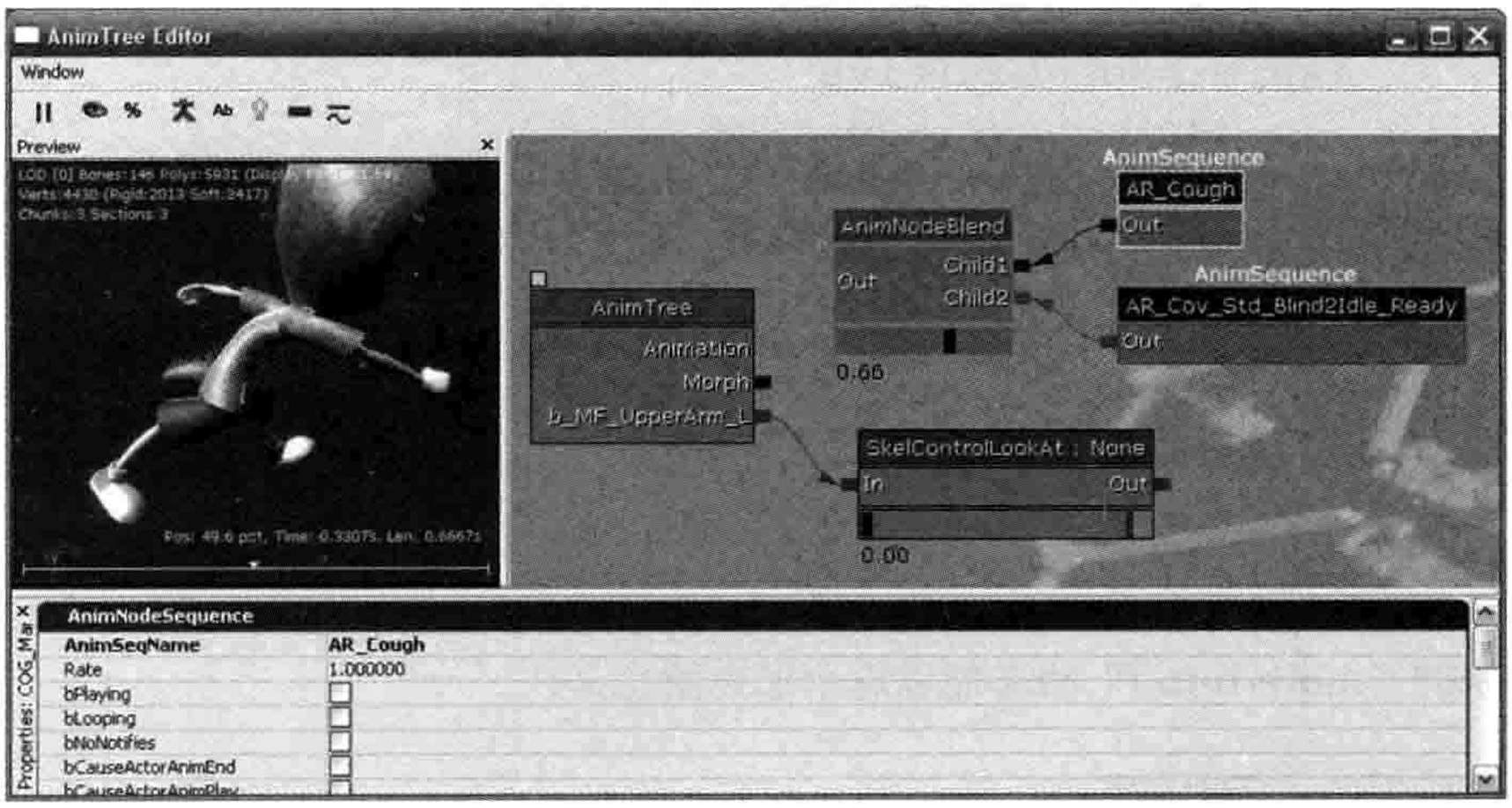


图 11.57: 虚幻引擎3的图形化动画编辑器。

虚幻引擎3的动画树

UE3的动画树本质上就是混合树。每个动画片段（UE3称为**序列**/sequence）由AnimSequence类型的节点表示。每个序列节点只有一个输出，它可直接连接到AnimTree的“动画”输入，或其他复杂节点类型。UE3提供许多开包即用的混合节点，包括二元混合、四向二维混合（称为**按瞄准混合**/blend by aim）等。UE3也提供许多特殊节点，例如用于缩放片段的播放速率（R）、把动画镜像化（如把右手动作变为左手动作）等。

UE3动画树的自定义化能力也非常高。程序员可以创建节点类型做到任意复杂的操作。因此，UE3开发者不限于使用简单的二元及三元LERP混合。在撰写本章时，UE3还未直接支持加法动画混合，但游戏队伍肯定有可能扩展UE3以支持加法动画[24]。

[24]译注：现时UE3已支持加法动画，见http://udn.epicgames.com/Three/AdditiveAnimations.html。

有趣的是，UE3所实现的角色动画方式并非明确地基于状态的。UE3让开发者建立一个巨型树，而不是通过定义多个状态，各状态含各自的混合树。UE3是通过开关树里的不同部分，来把角色转换至不同“状态”的。有些游戏团队实现了一个系统，用来动态地更替混合树的一部分，那么便可把巨大的树分拆为更易管理的子树。

虚幻引擎3的后期处理树（骨骼控制）

如前文所述，动画后期处理涉及用程序修改混合树所生成的骨骼姿势。在UE3里，骨骼控制（skel control）正是为此而设的。要使用骨骼控制，用户首先要为想要控制的关节在AnimTree上创建一个输入，然后建立合适的骨骼控制节点，再把该节点连接至AnimTree上的新输入。

UE3提供一些开包即用的骨骼控制，包括脚部反向动力学（foot IK，令脚部位置符合地表轮廓）、程序化“注视（look-at）”（令角色注视空间中某点），以及其他反向动力学等。如同动画节点，程序员可以轻松地创建自定义的骨骼控制，以满足开发中游戏的个别需求。

11.11.2 过渡

为了制作高质量的角色动画，我们必须小心处理动作状态机中的状态**过渡**，以确保动画连接之处不会出现兀突或粗糙的感觉。多数现在的动画引擎提供数据驱动的方式指定如何处理过渡。我们以下探讨此机制如何运作。

11.11.2.1 过渡的种类

状态之间的过渡有几种不同做法。若我们知道来源状态的最终姿势能完全匹配目标状态的初始姿势，那么我们可以简单地从一个状态“跳到”另一个状态。若非如此，我们便需要加入淡入 / 淡出。然而，淡入 / 淡出并不总是最佳之选。例如，我们无法用淡入 / 淡出从一个躺下状态真实地过渡至站立状态。实现这类过渡时，通常需要在状态机中引入特殊的**过渡状态**（transitional state）。过渡状态只为把某状态过渡至另一状态而设，不会作为一个稳定状态的节点。但由于过渡状态本身也是一个含完整功能的状态，它可以由任意复杂的混合树构成。因此，为过渡制作自定义动画时，过渡状态可提供最大的弹性。

11.11.2.2 过渡的参数

描述某两个状态间的过渡，需要指明多个参数，以完全控制过渡的过程。这些参数不限

于以下所列。

- **来源及目标状态**：过渡是施于哪对状态的？
- **过渡类型**：这是一个即时过渡、淡入 / 淡出，还是使用过渡状态？
- **持续时间**：淡入 / 淡出的过渡需要指明淡入 / 淡出的持续时间。
- **缓入 / 缓出曲线类型**：在淡入 / 淡出过渡中，我们希望指明混合因子的变化使用哪一种缓入 / 缓出曲线类型。
- **过渡窗口**（transition window）：某些过渡只能在来源动画的局部时间位于某个窗口内才能进行。例如，从出拳动画至击中反应动画的过渡，只有当出拳动画后半手臂收回的时候才有意义。在动画前半拳头向前击出时，不容许此过渡（或应选另一过渡）。

11.11.2.3 过渡矩阵

指定各状态之间的过渡，可能并非容易之事，因为过渡的潜在数量通常很大。在一个含有n个状态的状态机中，最多可以有n^2个过渡。我们可以想象有一个正方形矩阵，它的行及列都分别列举所有状态。这样的矩阵可用于指定所有可能的过渡，从任何行的状态过渡至任何列的状态。

在真实的游戏中，**过渡矩阵**（transition matrix）通常是颇稀疏的（sparse），因为并非每个状态都能过渡至所有状态。例如，死亡状态通常不能过渡至任何其他状态。又例如，驾驶状态通常不能过渡至游泳状态（至少要经过一个中间状态令角色跳出车辆!）。矩阵中独一无二的过渡数目可能大幅少于合法的过渡数目。因为我们经常可以重复使用过渡规格，套用于多对状态。

11.11.2.4 实现过渡矩阵

实现过渡矩阵有许多方法。可以用表格（spreadsheet）程序，把所有过渡填写为矩阵式的表格。也可以把过渡写进动作状态的文本文件中。若有图形界面的状态编辑器，我们可以在那里加入过渡。在以下几个小节中，我们浅谈几个真实游戏中的过渡矩阵实现方式。

例子：《荣誉勋章：血战太平洋》的通配符

在《荣誉勋章：血战太平洋（Medal of Honor: Pacific Assault，MOHPA）》中，我们利用过渡矩阵的稀疏性质，在过渡规格中加入通配符（wild-card）的支持。对于每个过渡规格，来源及目标状态的名字都可包含作为通配符的星号（*）。这样，我们就可以指定一个

从任何状态至任何状态的默认过渡（通过语法from="*" to="*"），然后便可以简单地从这个全局默认过渡细分至不同种类的过渡。这个细分过程可以一直进行，有需要时为个别状态对指定自定义过渡。MOHPA的过渡矩阵大约是这样的：

```
<transitions>
    // 全局默认过渡
    <trans from="*" to="*" type=frozen duration=0.2>
    ...
    // 任何步行至任何跑步的默认过渡
    <trans from="walk*" to="run*" type=smooth duration=0.15>
    ...
    // 任何俯卧至任何起身动作的专门处理
    // （仅在局部时间线上的2s至7.5s有效）
    <trans from="*prone" to="*get-up" type=smooth duration=0.1
        window-start=2.0 window-end=7.5>
    ...
    // 蹲下步行至跳跃的特殊情况
    <trans from="walk-crouch" to="jump" type=frozen duration=0.3>
    ...
</transitions>
```

例子：神秘海域引擎的第一类过渡

一些动画引擎中，高层游戏代码以明确指定目标状态的方式，从当前状态过渡至目标状态。此方法的问题在于，调用方代码必须知道在某个状态之下哪些是合法目标状态的名字。

在顽皮狗的**神秘海域**引擎中，我们通过把过渡从第二级的实现细节变为第一类实体来克服这个问题。每个状态提供能合法过渡的目标状态列表，并给予每个过渡唯一的名字。我们把过渡的名字标准化，令能预计到每个过渡的**效果**。例如，若有一个名为“步行”的过渡，无论当前状态是什么，此过渡**总是**能把当前的状态转换至某个步行状态。每当高层的动画控制代码希望由状态A过渡至状态B，它需要使用过渡的名字（而非明确指定目标状态）。若能找到该过渡，并且该过渡是合法的就会采用；否则，过渡请求失败。

以下的例子定义了4个过渡，分别名为“reload（装弹）”、“step-left（向左踏步）”、“step-right（向右踏步）”及“fire（开火）”。当中，(transition-group …)命令调用之前已定义的过渡，当多个状态要使用同一组过渡时就很有用。(transition-end …)命令用于指定最终过渡，若状态的局部时间线到达终点前没执行其他过渡，就会自动使用最终过渡。

```
(define-state complex
    :name "s_turret-idle"
```

```
:tree (aim-tree (anim-node-clip
                    "turret-aim=all--base")
                "turret-aim-all--left-right"
                "turret-aim-all--up-down")
:transitions (
    (transition "reload" "s_turret-reload"
        (range - -) :fade-time 0.2)

    (transition "step-left" "s_turret-step-left"
        (range - -) :fade-time 0.2)

    (transition "step-right" "s_turret-step-right"
        (range - -) :fade-time 0.2)

        (transition "fire" "s_turret-fire"
            (range - -) :fade-time 0.1)

        (transition-group "combat-gunout-idle^move")

        (transition-end "s_turret-idle")
    )
)
```

此方式的优美之处，可能并非一眼就能看得出来。其主要目的在于，以数据驱动方式修改过渡和状态，在许多情况下无须修改C++源代码。这种程度的弹性来自于分隔开动画控制代码和状态图的结构。例如，假设有10个步行状态（正常的、受惊的、蹲下的、受伤的等）。所有这些状态都可以过渡至跳跃状态，但不同的步行需要不同的跳跃（例如正常跳跃、受惊的跳跃、从蹲下起跳、受伤的跳跃等）。在这10个步行状态中，我们都简单定义一个名为“跳跃”的过渡。最初，我们把这些过渡都指向一个通用的“跳跃”状态，仅仅用来令游戏可以运行。然后，我们微调部分状态，令它们过渡至自定义的跳跃状态。我们甚至能在某些“步行”及对应的“跳跃”状态之间加入过渡状态。改动这些状态图的结构及过渡参数，只要过渡的**名字**没改便不会影响C++源代码。

11.11.3 状态层

多数生物可以利用它们的身体在同一时间做多件事情。例如，一个人类在使用下半身走路时，可以控制肩、头及眼察看某东西，并同时用手和臂做手势。身体不同部位的动作通常并不完全同步，某些身体部位常常会“引领”其他部位的动作（例如，头部引领转身时，肩部、髋部、脚部按序跟随）。在传统动画中，此知名技巧称为**预备动作**（anticipation）[44]。

这种动作好像和基于状态机的动画不太协调。毕竟，我们在某时刻只能设置一个状态。那么我们怎样能够独立地操控身体的不同部位呢？此问题的解决方案之一就是，引入**状态层**（state layer）的概念。每个状态层在某一刻只能有一个状态，但不同状态层之间在时间上是独立的。骨骼最终姿势的计算方法是，对n个层各自的混合树取值，产生n个骨骼姿势，再用预先设定的方法把这些姿势混合在一起，如图11.58所示。

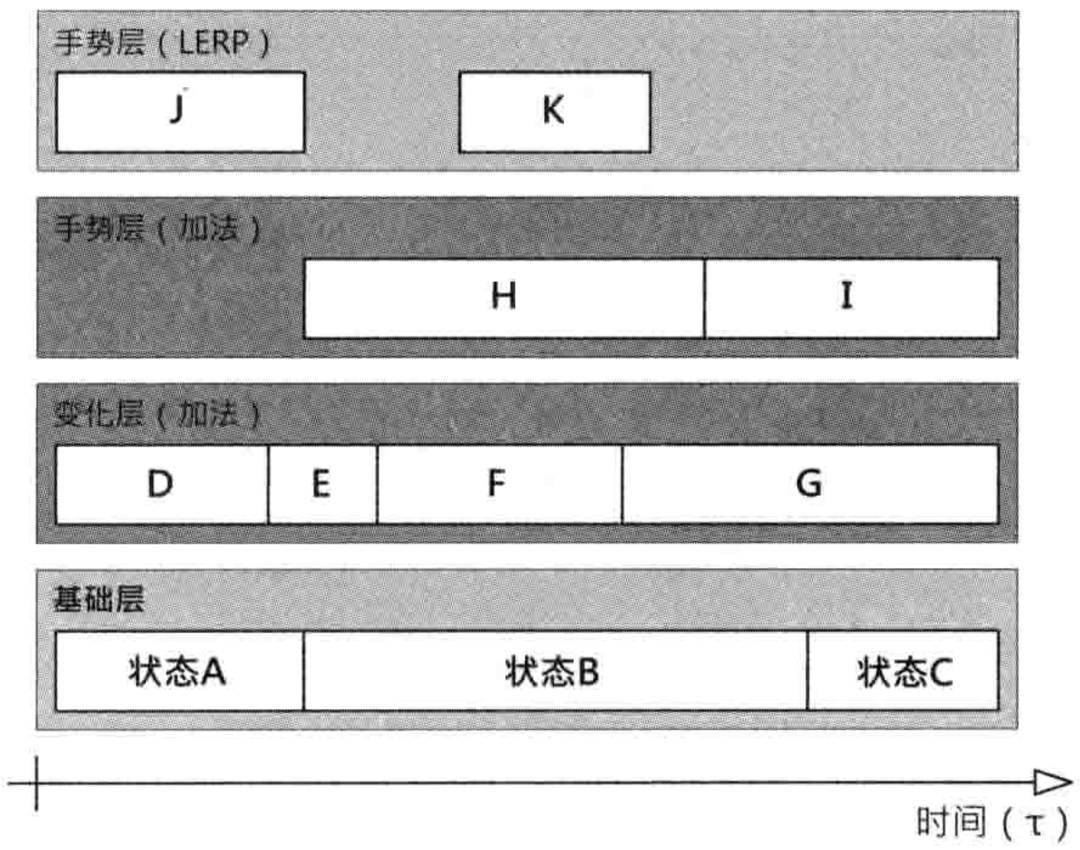


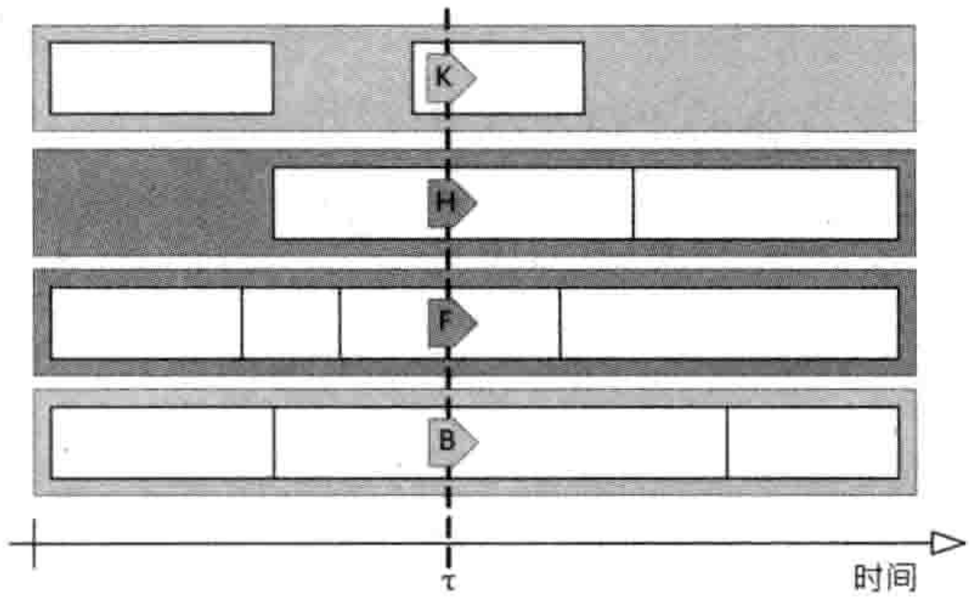
图 11.58: 含有层的动画状态机，展示每层的状态过渡如何可以在时间线上互相独立。

神秘海域引擎使用分层状态架构。这些层形成一个堆栈，当中底层（称为**基层**/base layer）总是产生全身的骨骼姿势，而其上的层可以产生全身、分部骨骼或加法姿势。该引擎支持两种层：LERP层和加法层。LERP层把其输出与下层所产生的姿势以LERP混合。加法层则假设其输出混合必然是一个区别姿势，并使用加法混合和下层的输出合并。此架构的最终效果是，分层状态机把多个时间上独立的混合树（每层一个）转换为单个统一混合树，如图11.59所示。

11.11.4 控制参数

把所有混合权重、播放速率，以及复杂动画角色的其他控制参数和谐地结合在一起，从软件工程的角度来说是一件富挑战性的事情。不同的混合权重对角色动画有不同影响。例如，某权重可能控制角色的移动方向，而另一些权重则控制移动速率、水平/垂直武器瞄准、头/眼注视方向等。我们需要某种方式把这些混合权重显露给代码，以供代码操控。

在扁平加权平均架构中，我们有一个扁平列表，内含所有能播放于角色的动画。每个片段状态都有一个混合权重、播放速率，可能还有其他控制参数。控制角色的代码必须使用名字取得某个片段，然后再适当地调整其混合权重。这种方式令程序接口变得很简单，然而，



于时间τ的净混合树

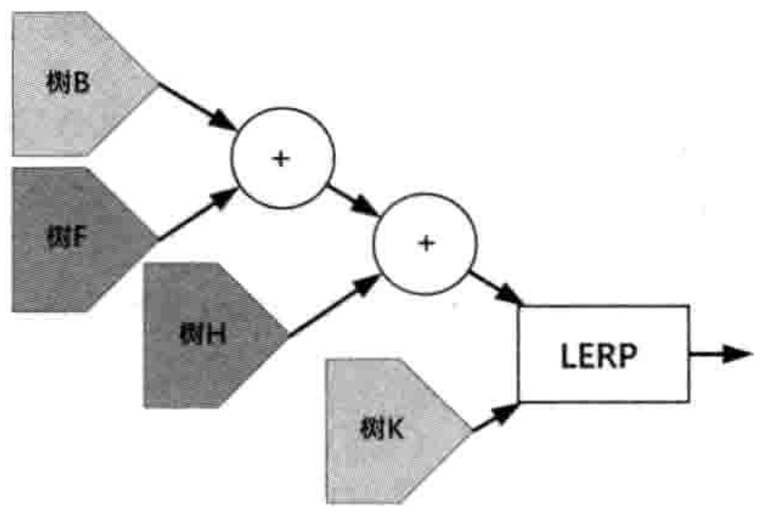


图 11.59: 含有层的动画状态机，可以把多个状态的混合树转换为单个统一的混合树。

它会把大部分控制混合权重的责任转移给角色控制系统。例如，要调整角色正在跑步的方向，角色控制代码必须知道“跑”这个动作是如何从一组动画片段组成的，这些片段可能称为“左移”、“向前跑”、“右移”、“向后跑”。那段代码必须用名字查找这些片段，并手工控制这4个混合权重来实现某个角度的跑步动画。不用多说，用这么细粒度的方式控制动画参数，不但繁重乏味，还会导致难以理解的源代码。

混合树架构也会产生另一些问题。多亏其树结构，片段自然地组成功能单位。自定义的树节点可以封装复杂的角色动作。这些方便都是优于扁平加权平均方式的。然而，这些控制参数深埋在树之内。代码若要控制头和眼的水平注视方向，必须要有混合树的**先验**（a priori）知识，才可以在树中找到合适的节点并控制其参数。

不同的动画引擎对此有不同的解决方法。以下是一些例子。

- **节点搜寻**：有些引擎提供**搜寻**混合节点的方式，供高层代码使用。例如，给予树中相关节点特别名字，如名为“HorizAim”的节点是用于控制水平瞄准的。控制代码可简单地在树中搜寻特定名字，若能找到，那么我们便知道调整其参数的效用。
- **命名变量**：有些引擎可以为个别控制参数命名。控制代码便可以用名字查找控制参数并调整其值。
- **控制结构**：另一些引擎使用简单的数据结构，例如浮点数数组或C `struct`，以储存整

个角色的控制参数。混合树中的节点连接至某些控制参数，连接方式可以用硬编码使用struct的成员，或是用名字或索引查找参数。

当然，还有许多其他不同的解决方案。每个动画引擎解决问题的方式有些微差异，但总体效果基本上是一样的。

11.11.5 约束

我们已经知道动作状态机如何用于指定复杂混合树，以及过渡矩阵如何控制状态之间的过渡方式。角色动画控制的另一重要方面在于，以多种方式约束角色及/或物体的移动。例如，我们可能要令武器总是常约束至携带者的手中。又例如，我们希望两个角色握手时，两只手被约束在一起。另外，角色的脚部通常会约束至和地面贴齐，而手部则会约束对齐至梯子的横档上或车辆的方向盘上。本节会简介在一般动画系统中如何处理这些约束（constraint）。

11.11.5.1 依附

几乎所有现在的游戏引擎都支持把物体依附至另一物体之上。在其最简单模式中，物体对物体的依附（attachment）涉及约束物体A骨骼某关节J_A的位置及/或定向，使其与物体B骨骼某关节J_B重叠。依附通常是一个父子关系，当父骨骼移动时，子物体应调整以满足约束，如图11.60所示。

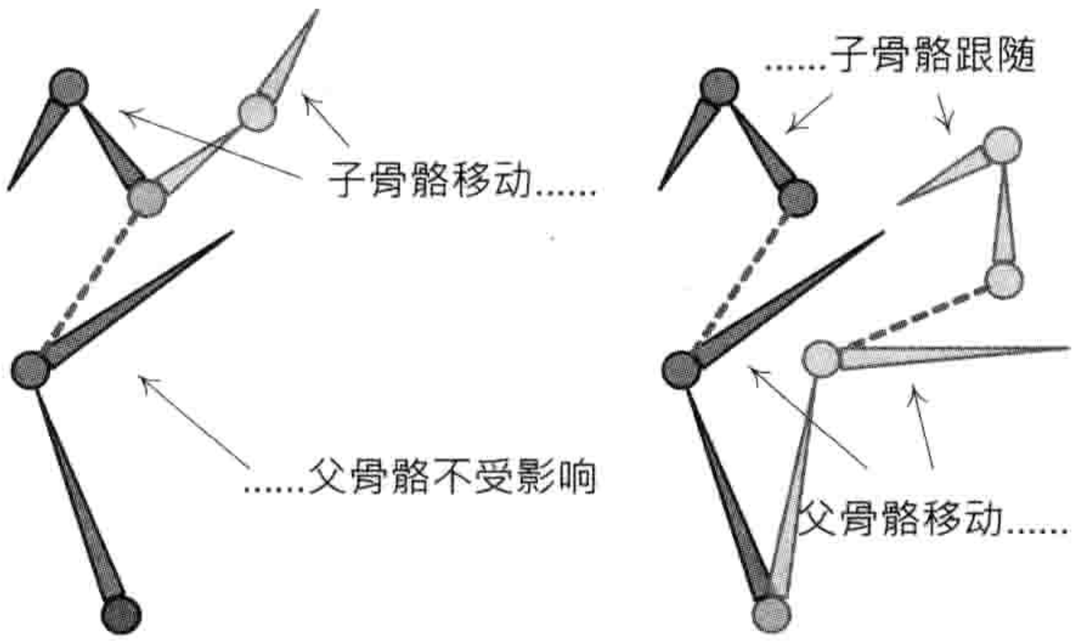


图 11.60: 一个父骨骼依附一个子骨骼。展示了父骨骼的移动如何自动地产生子骨骼的移动，但反之则不然。

有时候，在父关节和子关节间加入一个**偏移**（offset），可能会很方便。例如，当要把一支枪放在角色手上时，我们可以把枪的“把手（grip）”关节与角色的“右腕”重叠。然而

这样不一定能对齐手和枪的位置。此问题的解决方法之一，可以在角色骨骼中在“右腕”下加入一个“右手握枪”关节，再调整此关节位置使与枪的“把手”关节重合，令角色很自然地掌握着枪。然而，此解决方法会增加骨骼的关节数目。每个关节在动画混合及矩阵调色板计算中有处理成本，在储存动画键时也有内存成本。因此，增加新关节有时候并不一定是可行选择。

我们知道新增用于依附的关节并不影响角色的姿势，它仅仅为依附中的父子关节间加入额外的变换。因此我们真正需要做的，就是为一些关节打标记，使动画混合管道忽略它们，但却能用于依附之用。这种特别关节有时候称为**依附点**（attach point）。图11.61展示了这种关节。

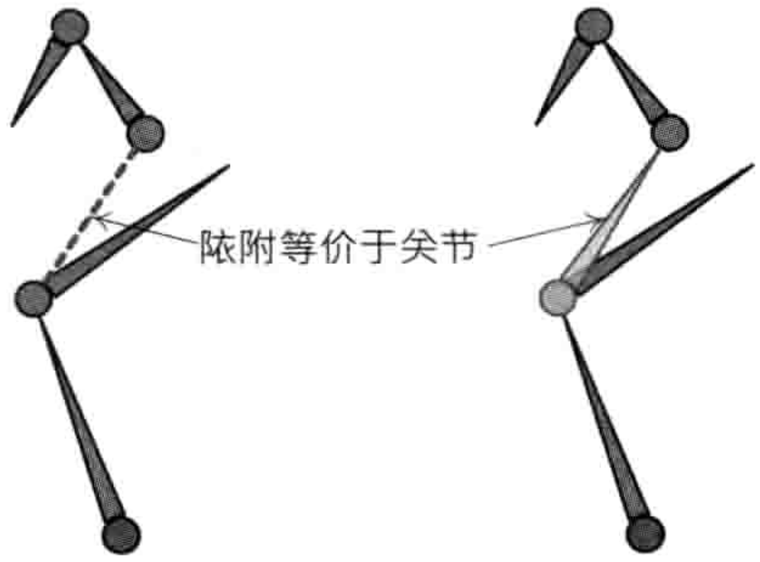


图 11.61: 依附点等同于在父与子之间加入一个额外的关节。

在Maya中可以把依附点当作一般关节或定位器（locator）般建模，然而多数游戏引擎能更方便地定义依附点。例如，依附点可以在动作状态机中指定，或是在动画制作工具中的自定义GUI里指定。那么动画师就能专注于影响角色外观的关节，而其他需要控制依附点的组员（如游戏设计师及工程师）则可以方便地控制依附设置。

11.11.5.2 跨物体对准

新游戏的角色和环境之间的互动越来越复杂及细致。因此，我们必须要有一个系统可以令动画中的物体对齐至另一物体。这种系统可用于游戏内置电影及互动游戏性元素。

想象一个动画师在Maya或其他动画工具中设置了两个角色和一道门。两个角色握手后，其中一个角色打开门，然后两个角色走过那道门。动画师必须确保这3个场景中的参与者能完美地对齐。然而，这些动画导出后会变成3个独立的动画片段，要分别施于游戏世界中3个独立的演员（actor）。在动画开始之前，两个角色可能由AI或玩家所操控。那么我们怎样才能令3个演员在播放3个片段时完美地对齐呢？

参考定位器

良好的解决方法之一，就是为3个动画片段加入一个共同的参考点。在Maya中，动画师可以在场景中放置一个**定位器**（locator，其实仅为一个三维变换，像骨骼关节一样），放置地点随意，觉得方便即可。其位置与定向其实无所谓，这点在以下就能明白。定位器会加上标签，以告诉导出器该定位器要特别处理。

当导出3个动画片段时，导出工具会储存参考定位器的位置及定向，这些位置及定向表示为相对于每个演员的局部物体空间，并储存至所有3个片段数据文件中。之后，当要在游戏中播放此3个片段时，动画引擎能查找到3个片段中相对于参考定位器的位置和定向。动画引擎可以把3个物体的原点变换，令3个参考定位器在世界空间重合。参考定位器的作用如同**依附点**（11.11.5.1节），而实际上它们可实现为同一功能。实际结果就是所有3个演员互相对齐，完全与原始在Maya场景中对齐时一样。

图11.62展示例子中的门和两个角色如何在Maya场景中设置。在图11.63中，把参考定位器置于每个片段之中（表示为演员的局部空间）。在游戏中，这些局部空间的参考定位器与世界空间的固定定位器对齐，从而重新把演员对齐，如图11.64所示。

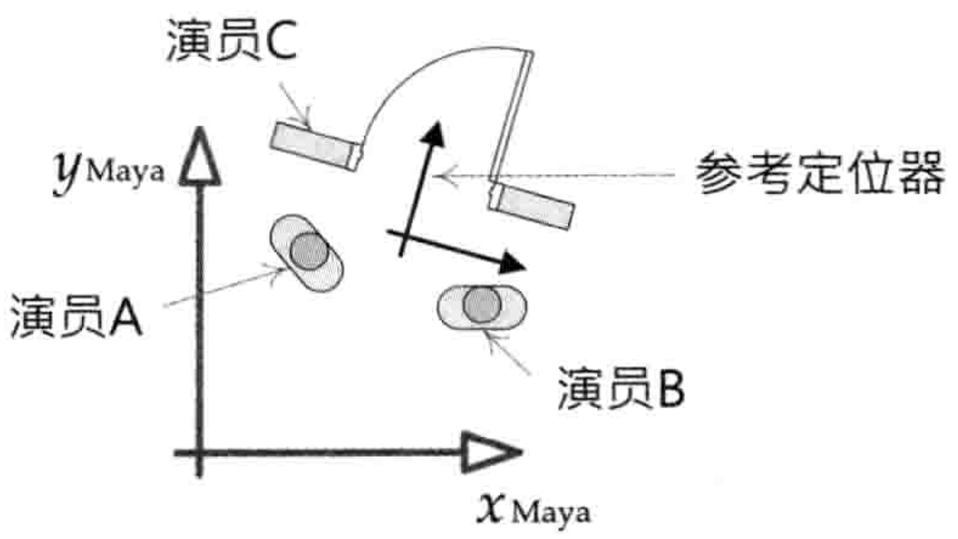


图 11.62: 原始的Maya场景包含3个演员和1个参考定位器。

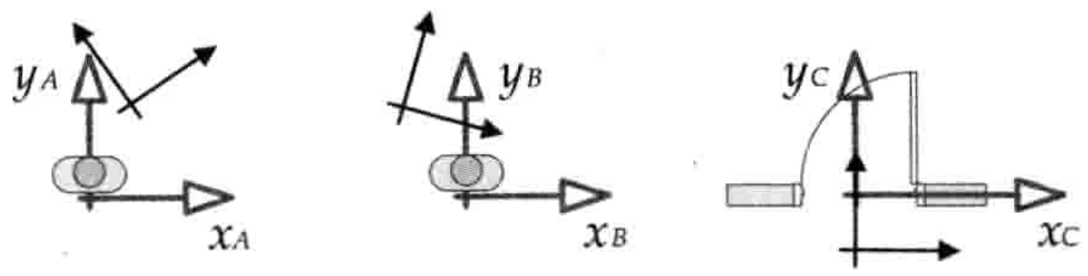


图 11.63: 把参考定位器置于每个演员的片段之中。

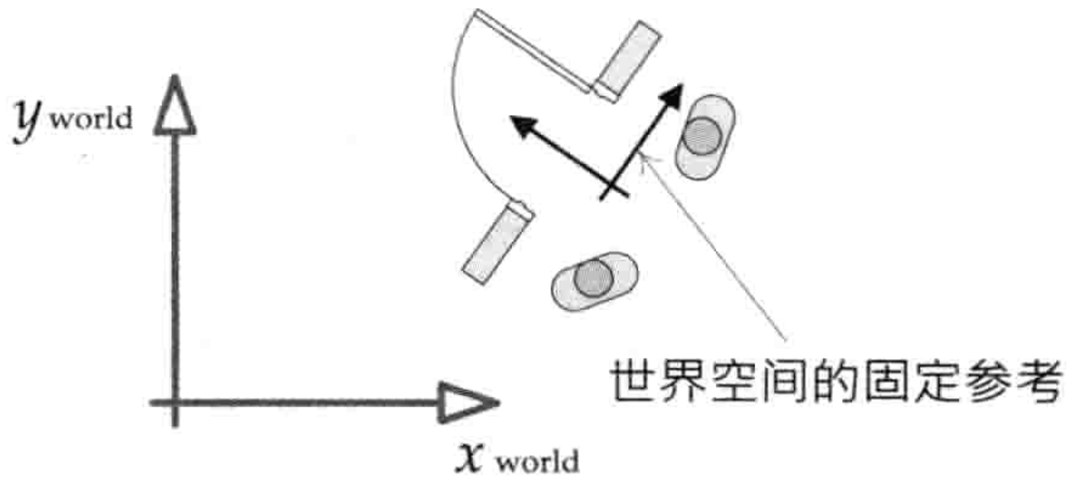


图 11.64: 在运行时，这些局部空间的参考定位器与世界空间的定位器对齐，从而重新把演员对齐。

求出世界空间的参考定位器

以上我们忽略了一个重要细节——谁决定参考定位器的世界空间位置和定向？每个动画片段提供**其演员**空间的参考定位器变换。但我们需要某种方式定义该参考定位器应怎样置于世界空间中。

在我们的例子中，其中一个演员是固定于世界中（那道门）的。因此，一个可行的方案是从那道门求出参考定位器的世界空间位置和定向，并令两个角色与其对齐。此过程所需的命令大概如以下的伪代码：

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB)
{
    // 从该门的动画，取得其参考定位器的世界空间变换
    Transform refLoc = GetReferenceLocatorWs(
        door, "shake-hands-door");

    // 就地播放门的动画（它本身已位于世界中正确的位置）
    PlayAnimation("shake-hands-door", door);

    // 采用相对于从门取得的参考定位器，播放两个角色的动画
    PlayAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);

    PlayAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

另一个选择是独立于场景中的3个演员定义参考定位器的世界空间变换。例如，我们可

以用世界编辑工具把参考定位器放置在场景中（见13.3节）。那么在此情况下，以上的伪代码可改成这样：

```
void playShakingHandsDoorSequence(
    Actor& door,
    Actor& characterA,
    Actor& characterB,
    Actor& refLocatorActor)
{
    // 简单地查询一个独立演员的变换，来取得参考定位器的世界空间变换
    // （假设该独立演员是被手工置于世界之中的）
    Transform refLoc = GetActorTransformWs(refLocatorActor);

    // 采用相对于世界空间参考定位器，播放3个角色的动画
    PlayAnimationRelativeToReference(
        "shake-hands-door", door, refLoc);

    PlayAnimationRelativeToReference(
        "shake-hands-character-a", characterA, refLoc);

    PlayAnimationRelativeToReference(
        "shake-hands-character-b", characterB, refLoc);
}
```

11.11.5.3 抓取及手部IK

就算经过依附连接两个物体后，有时候我们会发现那个对齐在游戏中可能仍然显得不太正确。例如，某角色右手可能拿着步枪，并让左手扶着枪托。当角色用武器瞄准不同方向时，我们可能会发现在某些瞄准角度，左手不再完全与枪托对齐。这种关节不对齐情况由LERP混合而生。就算问题关节在片段A和片段B中完全对齐，A和B的LERP混合并不能保证这些关节仍然对齐。

此问题的解决办法之一是，使用**逆向运动学**（inverse kinematics，IK）修正左手的位置。其基本方式是指定关节的目标位置，然后把IK施于由该关节起往上的一小串关节链（通常是2～4个关节）。需要修正的关节称为**末端受动器**（end effector）。IK求解程序会调整末端受动器的（多个）父关节的定向，令末端受动器尽量接近目标。

IK系统的API形式通常是对某个关节链开关IK，再指定目标点。IK通常是在底层动画管道中计算的，这种设计令IK计算可以在合适的时机执行——在计算中间局部及全局骨骼姿势之后，但在计算最终矩阵调色盘之前。

有些动画引擎容许预先定义IK链。例如，我们可以为左臂、右臂、左腿、右腿各定义一个IK链。假设在我们的例子中，一串IK链是由其末端受动关节的名字定义的。（其他引擎可能使用索引，或句柄，或其他标识符，但概念是一样的。）那么开启IK计算的函数可能是这个样子的：

```
void enableIkChain(
    Actor& actor,
    const char* endEffectorJointName,
    const Vector3& targetLocationWs);
```

而关闭IK计算的函数可能是这样子的：

```
void disableIkChain(
    Actor& actor,
    const char* endEffectorJointName);
```

IK通常相对较少开关，但世界空间的目标位置必须每秒更新（若目标正在移动）。因此，底层动画管道必须提供更新作用中IK目标点的机制。例如，管道可能会容许多次调用`enableIkChain()`。第一次调用时，会开启IK并设置目标点，而之后的调用只更新目标点。

IK适合用于关节和目标本身已相当接近，仅对关节对齐做出细微修正。若关节的实际位置和目标位置相距甚远，IK的表现就会不理想。也需要注意多数的IK算法只求关节的**位置**，读者可能需要编写额外的代码保证末端受动关节的**定向**也与目标对齐。IK不是万灵药，也可能有显著的效能消耗，所以必须用得明智。

11.11.5.4 动作提取及脚部IK

在游戏中，我们经常希望角色的走路动画能显得真实并“脚踏实地”。走路动画是否显得真实，最大的因素在于脚部是否在地上滑动。**滑步**（foot sliding）有多种解决方法，最常见的是**动作提取**及**脚部IK**。

动作提取

首先我们想象如何做角色向前直线步行的动画。在Maya（或其他动画工具）中，动画师首先令角色左脚向前踏出完整一步，然后到右脚。这样的动画称为**运动周期**（locomotion cycle），因为它设计做循环之用。动画师必须小心确保角色的脚显得与地面接触，并且不会在移动时滑步。角色在首帧的初始位置移动至周期末的新位置。图11.65展示了此周期。

注意在整个步行周期中，角色的局部空间原点维持不动。其效果是角色随着向前步行“远离背后的原点”。那么我们想象把这一动画循环播放。我们会见到角色向前步行一个完整周期后，跳回角色在动画中第1帧的位置。显然这在游戏中是不行的。

要正确播放，我们需要消除角色向前的移动，令角色的局部空间原点一直维持大约于角色重心之下。实现此方法可简单把角色骨骼根节点的向前平移全设成0。那么得出的动画片段就有如角色在“月球漫步（moonwalk）[25]”，见图11.66。

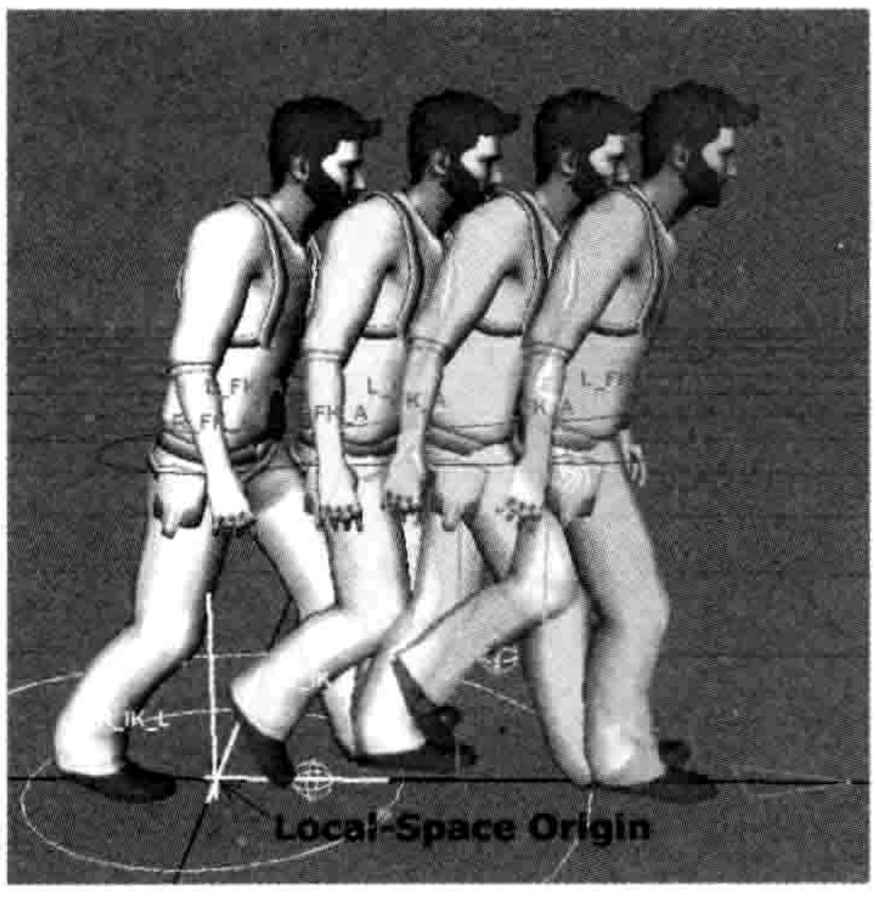


图 11.65: 在动画制作软件中，角色在空间中往前移动，显得“脚踏实地”。

图 11.66: 把根节点的向前移动清零后的步行周期。

要令角色的脚如在原始Maya场景中紧贴地面，我们需要令角色每帧向前移动合适的距离。我们可以计算角色在一个周期中移动的总距离，把它除以周期的总时间，就能得出平均移动速率。但角色步行时的向前移动速率并非常数。这一情况在角色蹒跚而行时特别明显（角色受伤的脚向前急速移动，而“好”的那只脚则接着缓慢移动），但所有自然的步行周期都是如此。

因此，在把根关节的向前移动设为0之前，我们预先把动画数据储存为一个特别的“提取动作（extracted motion）”通道。此数据能在游戏中用于把角色的局部空间原点向前移动，移动幅度完全按照Maya中每帧根节点的移动。结果就是角色向前步行得和制作时一样，但现在他的局部空间原点随步行而移动，令动画正确地循环，如图11.67所示。

若角色在动画片段中向前步行4英尺，而动画需时1s，那么我们知道角色的平均移动速率为每秒4英尺。要令角色以不同速率步行，我们只需要简单地缩放步行周期的播放速率。例如，我们希望角色以每秒2英尺速率步行，可以把播放速率设为一半（$R = 0.5$）。

[25]译注：这种舞步因迈克尔·杰克逊（Michael Jackson）而闻名。

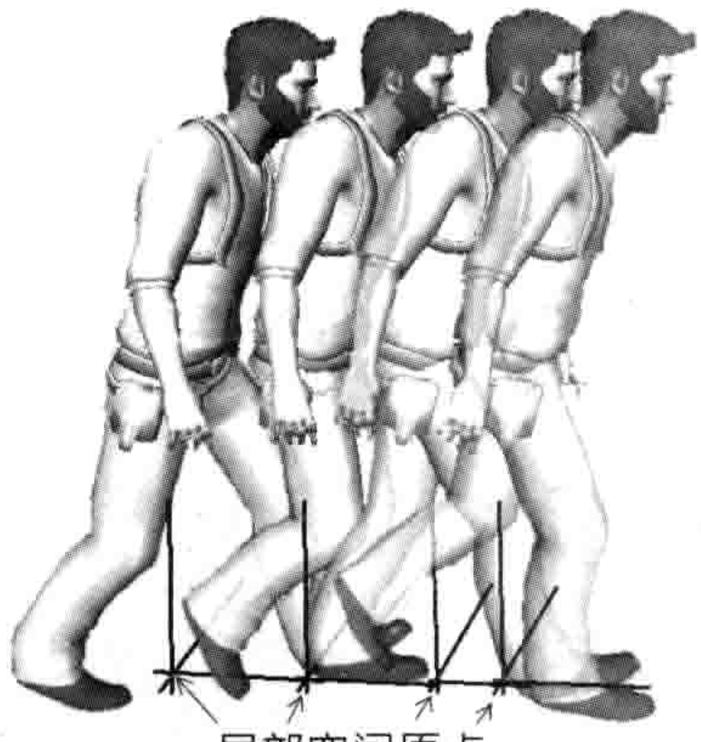


图 11.67: 提取根节点移动数据后，把数据应用至角色的局部空间原点，最后得出游戏中的步行周期。

脚部IK

当角色以直线移动（或更准确地说，角色以动画师制作时的路径而行），动作提取可以很好地令角色的脚显得贴地。然而，真实游戏的角色必须转身，并以原来手工制作时不同的方式移动（例如在不平坦的地表移动）。这会产生额外的滑步。

此问题的解决方法之一是使用IK修正滑步。其基本理念是分析动画去找出每只脚全部和地面接触的时间区间。在脚部接触地面时，我们记下其世界空间位置。之后在该脚仍然贴地的帧里，我们用IK去调整该腿的姿势，令脚部依然固定于正确位置。此技巧骤耳听上去好像很容易，但要令其美观和感觉良好是非常具挑战性的。这需要很多迭代及微调。而且有些自然的人类动作，例如增大跨步以领导转向，并不能单靠IK实现。

此外，角色动画的**美观**和角色的**操控感**是一个很大的权衡，当中玩家操作的角色尤是。通常，令玩家角色控制系统变得反应灵敏及好玩，比角色动画美观完美更为重要。结论是，不要看轻在游戏中加入脚步IK或动作提取的工作。要预留足够的时间反复试错，并要预备有所折中，确保玩家角色不单要好看，也要**操控感**良好。

11.11.5.5 其他类型的约束

还有许多其他类型的约束系统可以加进游戏动画引擎。一些例子包括：

- **注视**（look-at）：角色能注视环境中的兴趣点。角色可以仅用眼睛注视，又或同时用眼和头，又或加入上半身的扭动。注视约束有时候是以IK或程序式关节偏移实现的，但

更自然的观感可用加法混合实现。

- **掩护对准**（cover registration）：角色在掩护时要和掩护物完美地对齐。这通常是用上述谈及的参考定位器技术来实现的。
- **进入及离开掩护**：当角色使用掩护物时，我们通常必须使用自定义的进入及离开掩护动画混合。
- **通行协助**：令角色探索上下或周围的障碍物，或通过障碍物。这样能为游戏添加许多生气。通常的做法是要提供自定义动画，并加入参考定位器对准要克服的障碍物。

11.12 动画控制器

动画管道提供高速的动画姿势设置及混合功能，但于游戏性代码直接使用其接口通常会太累赘。动作状态机通过使用复杂的混合树描述及数据驱动方式，一般能提供更方便的接口，并能封装成易于理解的逻辑状态。同时，也可以用数据驱动方式定义状态间的过渡，使游戏性代码使用过渡后无须对过渡进行微观管理[26]。ASM系统也可提供分层机制，使角色的动作由多个并行的状态机所描述。但尽管动作状态机能提供相对较方便的接口，有些游戏团队为了方便会引入第三层软件，旨在提供更高层的角色动画控制。这层软件通常会实现为一组名为**动画控制器**（animation controller）的集合。

控制器管理趋向相对长时间的行为，通常是数秒或以上的级别。每个动画控制器通常负责一个类型的角色行为，例如，掩护时的行为、从游戏世界中一个地方移动至另一地方、驾驶汽车等。控制器通常要和谐地安排角色各方面动画相关行为。控制器要调整混合因子控制移动方向、瞄准等，也要管理状态过渡、层的淡入 / 淡出，以及做其他能令角色表现出所需行为的工作。

基于控制器设计的好处之一就是，所有和某行为类型相关的代码都置于同一地方。此设计也能令高层游戏性系统（例如玩家机制及AI）写得更简单，因为所有动画的微观管理细节都要抽取出来并埋于控制器之中。

动画控制器层可以用很多方式实现，并且跟游戏需求和工程团队的软件设计哲学有很大依赖关系。有些团队完全不用动画控制器。一些团队会把动画控制器紧紧整合至AI及/或玩家机制系统中。另一些团队会实现相对较通用的控制器，供玩家角色和非玩家角色共同使用。不论是好事还是坏事，游戏产业中并无标准的动画控制器实现方式（至少暂时如此）。

[26] 译注：原文fire-and-forgot（射后不理）原指导弹发射后便不需要控制的模式。

第12章　碰撞及刚体动力学

在真实世界中，固体物体本质上……就是固态的。它们通常不会做出不可能的事情，例如互相穿透对方。但是在虚拟游戏世界中，除非我们告诉物体如何做某些事情，否则它们不会有这些性质。游戏程序员需要花许多精力，才能确保物体不会互相穿透。这是任何游戏引擎的核心元件之一的角色——**碰撞检测系统**（collision detection system）。

游戏引擎的碰撞系统通常紧密地和**物理引擎**（physics engine）整合。当然，物理的范畴很广，而现在游戏引擎所指的“物理”，更精确地说应称为**刚体动力学**（rigid body dynamics）模拟。**刚体**（rigid body）是理想化、无限坚硬、不变形的固体物体。**动力学**（dynamics）是一个过程，计算刚体怎样在**力**（force）的影响下随时间**移动**及**相互作用**。刚体动力学模拟令游戏中的物体移动得高度互动及混沌自然，这种效果难以用预制的动画片段达成。

动力学模拟需大量使用碰撞检测系统，以正确地模拟物体的多种物理行为，包括从另一物体弹开，在摩擦力下滑行、滚动，并最终静止。当然，碰撞检测系统也可以不结合动力学模拟，仅单独使用。许多游戏甚至没有“物理”系统。但当涉及在二维或三维空间中移动物体，这些游戏都需要某种形式的碰撞检测。

我们在本章会探讨典型碰撞检测系统及典型物理（刚体动力学）系统的架构。在我们探讨这两个密切相关系统的组件时，我们也会研究它们背后的数学及理论。

12.1　你想在游戏中加入物理吗

今天，多数游戏引擎都富有某种物理模拟能力。有些物理效果是玩家们觉得必然有的，例如布娃娃式死亡。另一些物理效果则加入游戏后，例如绳子、布料、头发或复杂的物理驱动机械，可以令游戏获得与众不同的**难以言喻的特质**（je ne sais quoi）。近年许多游戏工作室开始为高级物理模拟做实验，包括近似的实时流体机制效果及可变形的物体模拟。然而，

在游戏中加入物理并非零成本的，在我们下定决心为我们的游戏实现林林总总的物理驱动功能之前，我们应该（至少）了解一下当中的取舍。

12.1.1 物理系统可以做的事情

以下是一些游戏物理系统可以做的事情。

- 检测动画物体和静态世界几何物体之间的碰撞。
- 模拟在引力及其他力影响下的自由刚体。
- 弹簧质点系统（spring-mass system）。
- 可破坏的建筑物和结构。
- 光线（ray）及形状的投射（用以判断视线、弹道等）。
- 触发体积（trigger volume）（判断物体进入 / 离开游戏世界中定义的区域，或逗留在那些区域的时间）。
- 容许角色拾起刚体。
- 复杂机器（起重机、移动平台谜题等）。
- 陷阱（例如山崩的泥石）。
- 带有逼真悬挂系统的可驾驶载具。
- 布娃娃式死亡。
- 富动力的布娃娃：真实地混合传统动画及布娃娃物理。
- 悬挂道具（水壶、项链、配剑）、半真实的头发/衣服移动。
- 布料模拟。
- 水面模拟及浮力（buoyancy）。
- 声音传播。

注意，除了在游戏运行时运行物理模拟，我们也可以在离线预处理步骤中运行模拟以生成动画片段。Maya等动画工具有许多插件可用。这也是NaturalMotion公司的Endorphin软件包[1]所采用的方式。我们在本章中只讨论运行时的刚体动力学模拟。然而，离线工具也是很强大的，在策划游戏项目时应记得有此选择。

[1] http://www.naturalmotion.com/endorphin.htm

12.1.2 物理好玩吗

在游戏中使用刚体动力学系统，不一定能令游戏变得好玩。物理模拟天生的混沌行为实际上可能会干扰游戏体验，而非增强游戏体验。由物理产生的乐趣受多个因素影响，包括模拟本身的质量、与其他引擎系统整合的程度、选择哪些游戏元素采用物理驱动哪些直接操控、物理性元素如何与游戏目标及游戏角色能力互动，以及游戏的类型。

以下我们看看一些游戏类型如何整合刚体动力学系统。

12.1.2.1 模拟类游戏

模拟类游戏（simulation game）的主要目标在于准确地模仿出现实世界的体验。例子包括《模拟飞行（Flight Simulator）》、《跑车浪漫旅（Gran Turismo）》、《NASCAR赛车（NASCAR Racing）》游戏系列。显然，刚体动力学系统所提供的真实性完全合乎这类型游戏。

12.1.2.2 物理解谜游戏

物理解谜（physics puzzle）就是供玩家玩一些以动力学模拟的玩具。因此这类游戏显然需要完全依赖物理作为其核心机制。这类游戏的例子包括《Bridge Builder》、《The Incredible Machine》、页游《Fantastic Contraption》、iPhone上的《Crayon Physics》[2]。

12.1.2.3 沙箱游戏

在沙箱游戏（sandbox game）中，可能没有任何目标，或者可能有大量可选的目标。玩家的主要目标通常是“到处胡闹”，并探索游戏世界中的物体可以用来做什么。沙箱游戏的例子有《侠盗猎车手（Grand Theft Auto）》、《孢子（Spore）》和《小小大星球（LittleBigPlanet）》。

沙箱游戏能好好利用真实的动力学模拟，尤其当游戏的乐趣是来自用游戏中物体真实的（或半真实的）互动。因此在这些情况下，物理本身就是乐趣所在。许多游戏舍弃一些真实性换取更多乐趣（例如，比现实更大规模的爆炸、比正常更强或更弱的地心引力等）。因此动力学模拟可能需要以多种方式调整达至良好的“感觉”。

[2]译注：现在最著名的例子大概是《愤怒的小鸟（Angry Birds）》。它采用Box2D引擎。

12.1.2.4 基于目标及基于故事的游戏

基于目标的游戏含一些规则，以及一些玩家必须完成才能继续游戏的指定目标；在基于故事的游戏中，讲故事乃是最重要的。把物理系统整合至这些类型的游戏，可能会很棘手。我们通常会因模拟的**真实性**而失去一些**操控性**，降低操控性会阻碍玩家完成游戏目标的能力，以及游戏讲说故事的能力。

例如，在基于角色的平台游戏中，我们希望玩家角色能以又好玩又易操控的方式移动角色，而不需要特别真实。在战争游戏中，我们可能想炸一座桥，但希望确保散落的碎片不会阻塞玩家的唯一前进路径。在这类型游戏中，物理通常不需要好玩，事实上当玩家“卡关”时游戏的物理行为可能令玩家更没趣。因此，开发者必须谨慎明智地应用物理，并采取步骤控制模拟的行为，确保物理不会妨碍游戏性。

12.1.3 物理对游戏的影响

在游戏中加入物理模拟，对项目及游戏性可能有多种影响。以下是对几个游戏开发范畴影响的例子。

12.1.3.1 对设计的影响

- **可预测性**（predictability）：物理模拟行为与手工动画的区别在于，其天生的混沌性及多变性，而这也成为物理模拟不可预测的原因之一。若某事情必须每次都以某种方式发生，最好还是使用动画，而不需要逼动力学模拟确保每次产生相同的结果。
- **调校及控制**：物理定律（若是现实的正确模型时）是恒常不变的。游戏中，我们可以调整重力的值或某刚体的恢复系数（restitution coefficient），来重夺某种程度的操控性。然而，调整物理参数的效果并不直观而且难以可视化。要令一个角色向某个方向走，用调整力的方式比调整角色走路动画要难得多。
- **意外行为**：有时候物理会产生游戏中预料之外的特征，例如《军团要塞（Team Fortress Classic）》中的火箭筒跳跃秘技、《光环（Halo）》中的疣猪号战车空中爆炸术、《超能力战警（Psi-Ops）》中的飞行“滑板”。

总括来说，游戏设计应驱动游戏引擎物理方面的需求，而不是倒过来。

12.1.3.2 对工程的影响

- **工具管道**：优良的碰撞/物理管道需要花长时间去建设及维护。

- **用户界面**：玩家如何操控世界中的物理物体？能射击它们吗？能走进它们内？能拾起它们？用《重返侏罗纪（Trespasser）》的虚拟手臂？用《半条命2（Half Life 2）》的“重力枪”？
- **碰撞侦测**：用于动力学模拟的碰撞模型，可能需要比非物理驱动的模型更细致，建模时也要更谨慎。
- **人工智能**：使用物理模拟的物体后，路径可能无法预测。引擎可能需要处理动态的掩护点，这些掩护点可能会移动或遭炸毁。人工智能可否利用物理取得优势？
- **动画及角色动作**：以动画驱动的物体可以轻微与另一个物体碰撞而不产生不良效果，但使用动力模拟的话，物体可能会从另一物体弹开，而且是以预料之外的形式，或产生严重的抖动。或许需要加入碰撞过滤，容许一些物体可以轻微地互相重叠。此外也可能需要一些机制确保物体正确地平息下来及进入休眠模式。
- **布娃娃物理**：布娃娃需要大量微调，而且有时候会受模拟的不稳定所影响。由于动画可能会令部分身体与其他碰撞体积互相重叠，当角色转换成布娃娃时，这些重叠可能造成极大的不稳定性。必须采取措施避免这种情况发生。
- **图形**：物理驱动的动作或会影响可渲染物体的包围体积（否则包围体积就是固定的或更可预测的）。使用可破坏建筑及物体，可能会令一些预计算光照及阴影方法失效。
- **网络及多人**：不影响游戏性的物理效果可以仅仅在每个客户端机器（独立地）模拟。然而，会影响到游戏性的物理（例如手榴弹的轨道）则必须在服务器上模拟，并且准确地复制至所有客户端。
- **记录及重播**：记录游戏过程及在稍后重播的能力，对除错/测试很有帮助，也可作为一个有趣的游戏功能。此功能在含动力学模拟的游戏上更难实现，因为物理的混沌行为（在初始条件的少许改动会产生非常不同的模拟结果）及物理更新的时间差异会导致重播结果与记录有所出入。

12.1.3.3 对美术的影响

- **额外的工具及工作流程复杂度**：美术部门要在物体中加入质量、摩擦力、约束，以及其他动力学模拟所需的参数，令部门的工作变得更困难。
- **更复杂的内容**：我们可能需要对一个物体建立多个不同用途的版本，每个版本含不同的碰撞及动力学设置，例如无损坏的版本和可被破坏的版本。
- **失控**：物理驱动物体的不可预测性可能令美术人员难以控制场景的艺术构图。

12.1.3.4 其他影响

- **跨部门的影响：**在游戏中加入动力学模拟需要工程、美术、设计部门的紧密合作。
- **对制作的影响：**物理可能会增加项目的开发成本、技术/组织的复杂度，以及风险。

虽然物理对游戏会造成各种影响，但是今天大多数的团队还是选择整合刚体动力学系统至游戏中。只要在过程中配合谨慎的计划及明智的选择，在游戏中加入物理可能会是值得且卓有成效的。而且如下节所述，第三方中间件令物理比以前更平易近人。

12.2 碰撞/物理中间件

碰撞系统及刚体动力学模拟的开发是一件富挑战性及耗时的工作。游戏引擎的碰撞/物理系统可占一个游戏引擎源代码中显著的百分比。此系统需要写很多代码，而且还要维护!

庆幸坊间有许多健壮的、高质量的碰撞/物理引擎可供选择，当中一些是商业产品，一些是开源形式的。我们以下会列出几个做介绍。关于各个物理SDK的优缺点，可参考一些游戏开发论坛[3]。

12.2.1 I-Collide、SWIFT、V-Collide及RAPID

I-Collide是由北卡罗来纳大学教堂山分校（UNC）开发的一个开源碰撞检测程序库。它可以检测**凸体积**（convex volume）之间是否相交。后来一个更快、更多功能的SWIFT程序库取代了I-Collide。UNC也开发了一些能处理复杂非凸形状的库，包括V-Collide及RAPID。这些库不能在游戏中开箱即用，但可以作为一个很好的基础组建一个功能齐全的游戏碰撞检测引擎[4]。读者可以在官方网站[5]获知I-Collide、SWIFT及其他UNC几何程序库。

12.2.2 ODE

ODE是“Open Dynamics Engine（开放动力学引擎）”的缩写，是一个开源碰撞及刚体动力学SDK[6]。其功能和一些商用产品（如Havok）相近。其优点包括免费（对于小型游戏工作室和学生项目来说是一大优点!）而且有完整代码（调试更容易，也可以为游戏的特别需求修改物理引擎）。

---

[3]例如http://www.gamedev.net/community/forums/topic.asp?topic_id=463024

[4]译注：这些程序库中，部分虽然开源，但仅限非商业使用。商业上使用要另外商洽。

[5]http://gamma.cs.unc.edu/I-COLLIDE/

[6]http://www.ode.org

12.2.3 Bullet

Bullet是一个同时用于游戏及电影行业的开源碰撞检测及物理程序库。其碰撞引擎和动力学模拟整合在一起，但碰撞引擎也提供了钩子供独立使用或整合至其他物理引擎。Bullet支持**连续碰撞检测**（continuous collision detection，CCD），又称为**冲击时间**（time of impact，TOI）碰撞检测。此功能对检测细小高速移动的物体很有帮助，后文会再做解释。Bullet SDK可于这里[7]下载，另设维基[8]可供参考。

12.2.4 TrueAxis

TrueAxis[9]是另一个碰撞/物理SDK，非商业使用是免费的。

12.2.5 PhysX

PhysX起初是一个名为NovodeX的程序库。NovodeX由Ageia公司开发及发行，作为他们的专用物理协处理器的市场策略。后来NVIDIA收购了Ageia，并把PhysX改造成可使用NVIDIA GPU作为协处理器运行。（它也可以不使用GPU，完全在CPU上运行。）PhysX SDK可在官网下载[10]。Ageia和NVIDIA的部分市场策略是通过提供免费的CPU版本SDK，去驱动往后的物理协处理器市场[11]。开发者也可以付一定费用去获取完整的源代码，以供按需修改程序库。PhysX提供PC、Xbox 360、PlayStation 3及Wii的版本。[12]

12.2.6 Havok

Havok是商业物理SDK的绝对标准，它提供最丰富的功能集，并自夸在所有支持平台上都有极好的性能特征。（它也是最昂贵的解决方案。）Havok由一个核心碰撞/物理引擎，加上数个可选的产品所构成。这些可选产品包括载具物理系统、为可破坏环境建模的系统，以及一个全功能动画SDK，此动画SDK直接与Havok的布娃娃物理系统整合。Havok可以

---

[7] http://code.google.com/p/bullet/

[8] http://bulletphysics.org/mediawiki-1.5.8/index.php/Main_Page

[9] http://trueaxis.com/

[10] http://developer.nvidia.com/physx （译注：现时PhysX SDK不提供直接下载，需注册及审批。）

[11] 译注：后来的GPU架构加入更弹性的通用计算功能，所以暂时没有再推出专门为物理而设的协处理器。

[12] 译注：Matthias Müller-Fischer是PhysX的首席研究员，他也是NovodeX AG（被Ageia收购）的创办人之一。他发表过多篇对物理模拟方面具影响力的论文，这些论文及相关讲座简报可从其个人网站http://www.matthiasmueller.info/取得，非常值得参考。

在PC、Xbox 360、PlayStation 3及Wii上运行，并且已为这些平台个别优化。可以在其官网[13]获取更多信息。

12.2.7 PAL

PAL（Physics Abstraction Layer/物理抽象层）是一个开源程序库，让开发者可以在项目上用多于一个物理SDK。它提供PhysX（NovodeX）、Newton、ODE、OpenTissue、Tokamak、TrueAxis及其他几个SDK的钩子。可于官网[14]阅读更多信息。

12.2.8 DMM

位于瑞士日内瓦的Pixelux Entertainment公司开发了一个独一无二的物理引擎DMM（Digital Molecular Matter/数字分子物质）。DMM使用有限元素法（finite element method）去模拟可变形及可破坏的物体。DMM包含离线及运行时组件。它于2008年面世，曾用于卢卡斯艺能（LucasArts）的《星球大战：原力解放（Star Wars: The Force Unleashed）》中。可变形体的机制超出我们的讨论范围，但读者可在该公司的官网[15]获得更多信息。

12.3 碰撞检测系统

游戏引擎碰撞系统的主要用途在于，判断游戏世界中的物体有否**接触**（contact）。要解答此问题，每个逻辑对象会以一个或多个几何**形状**代表。这些图形通常较简单，例如球体、长方体、胶囊体等。然而，也可使用更复杂的形状。碰撞系统判断在某指定时刻中，这些图形有否**相交**（即重叠）。因此，美其名曰碰撞检测系统，本质上是几何相交测试器。

当然，碰撞系统不仅要回答图形是否相交，也提供接触的相关信息。接触信息可用于避免在屏幕上出现不真实的视觉异常情况，例如，两个物体**互相穿插**（interpenetrate）。其解决办法通常是在渲染前移动互相穿插的物体，使它们分离。碰撞也能提供对物体的**支撑**（support）——一个或多个接触合力令物体静止，施于物体的引力及/或其他力达至平衡。碰撞也可以用于其他用途，例如，当导弹击中目标时令其爆炸，或是当角色通过悬浮中的药包时为角色补血。通常，刚体动力学模拟是碰撞系统的最苛求客户，它要利用碰撞系统模仿真实的物理行为，例如反弹、滚动、滑动、达至静止等。然而，就算一些游戏无物理系统，也可能会大量使用碰撞检测引擎。

[13] http://www.havok.com

[14] http://www.adrianboeing.com/pal/index.html

[15] http://www.pixeluxentertainment.com

本章中，我们会从一个高层次的角度简介碰撞检测如何运作。对于本题目更深入的讨论，坊间有许多实时碰撞检测的好书，例如[12]、[41]和[9]。[16]

12.3.1 可碰撞的实体

若希望游戏中某逻辑对象需要和其他对象碰撞，我们便需要为该对象提供一个**碰撞表达形式**（collision representation），以描述对象的形状及其在游戏世界的位置和定向。碰撞表达形式是一个独特的数据结构，分离于对象的**游戏性表达形式**（gameplay representation）及**视觉表达形式**（visual representation，可能是一个三角形网格、细分曲面、粒子效果，或其他视觉表达形式）。

从检测相交的角度，我们通常希望形状在几何上和数学上是简单的。例如，供碰撞用途时，石头可能会建模为球体，车头罩可能会建模为长方体，人体可能会由一组互相连接的**胶囊体**（capsule，即药丸形的体积）逼近。理想地，我们只有当简单的表达形式不足以达成游戏中所需的行为，才会诉之于更复杂的形状。图12.1展示了几个使用简单形状逼近物体体积做碰撞用途的例子。

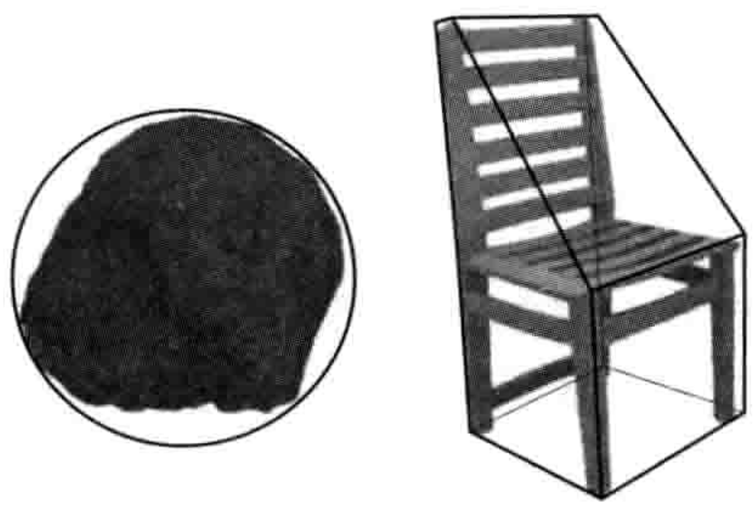


图 12.1: 游戏通常使用简单形状逼近物体体积。

Havok采用**可碰撞体**（collidable）一词，描述一个参与碰撞检测的单独刚体物体。Havok使用C++的`hkpCollidable`类的实例表示每个可碰撞体。PhysX称它的刚体为**演员**（actor），并表示为`NxActor`类的实例。在这些库中，可碰撞实体包含两个基本信息，**形状**及**变换**。形状描述可碰撞体的几何外形，而变换则描述形状在游戏中的位置及定向。碰撞物需要变换，原因有三。

1. 从技术上来说，形状只描述物体的外形（即它是一个球体、长方体、胶囊体，或其他类型的体积）。形状也描述物体的尺寸（例如球体的半径、长方体的长/宽/高等）。然而，形状通常以其位于原点的中心来定义，并以相对于坐标轴的某类典范定向来定义。为了使形状有其用，形状必须被变换，使其合适地放置及转向于世界空间中。

[16]译注：译者再推荐一本较近期的《Game Physics Pearls》。

2. 游戏中许多对象是动态的。如果必须把任意复杂形状的**特征**（如顶点、平面等）逐一移动，才能在空间中移动形状，那便会很耗时。然而使用变换的话，无论形状的特征是简单还是复杂，形状都能快速地移动。
3. 表示复杂类型的形状可能会占据不少内存。因此，多个可碰撞体共享一个形状描述，能节省内存空间。例如，在赛车游戏中，许多车辆的形状信息可能是相同的。在这种情况下，所有的车辆可碰撞体可使用单个车辆形状。

游戏中的对象可以完全没有可碰撞体（若它不需要碰撞检测服务），又可以含单个可碰撞体（若该对象是一个刚体），还可以含多个可碰撞体（例如，当中每个可碰撞体代表有关节机器人的每个刚体组件）。

12.3.2 碰撞/物理世界

碰撞系统通常会通过一个名为**碰撞世界**（collision world）的数据结构，管理其所有的可碰撞实体。碰撞世界是专门为碰撞检测而设的游戏世界完整表达方式。Havok的碰撞世界是`hkpWorld`类的实例。类似地，PhysX中碰撞世界是`NxScene`的实例。ODE使用`dSpace`类的实例表示碰撞世界，它实际上是几何体积层阶结构的根，代表游戏中所有可碰撞形状。

相比储存碰撞信息在游戏对象本身，把所有碰撞信息维护于私有的数据结构有几个优点。其一，碰撞世界只需包含游戏对象中有可能碰撞的可碰撞体。那么碰撞系统便不需要遍历无关的数据结构。此设计也能令碰撞数据以最高效的方式组织。例如，碰撞系统可以利用缓存一致性以增强性能。碰撞世界也是一个有效的封装机制，此机制通常从可理解性、可维护性、可测试性、可重用性来说都有帮助。

12.3.2.1 物理世界

若游戏含刚体动力学系统，该系统通常会紧密地与碰撞系统整合。动力学系统的“世界”数据结构通常会与碰撞系统共享，模拟中每个刚体通常会关联至碰撞系统里的一个可碰撞体。此设计在物理引擎中屡见不鲜，因为物理系统需要频繁地使用细致的碰撞查询。实际上，通常是由物理系统**驱动**碰撞系统的运作，物理系统在每个模拟时步中指挥碰撞系统执行至少一次、有时几次的碰撞测试。因此，碰撞世界有时候称为**碰撞/物理世界**，或直接称之为**物理世界**（physics world）。

在物理模拟中的每个动力学刚体通常关联至碰撞系统里的单个可碰撞体（虽然每个可碰撞体不一定需要一个动力学刚体）。例如，在Havok中，刚体由`hkpRigidBody`类的实例

表示，而每个刚体都含一个指针指向正好一个hkpCollidable。在PhysX中，可碰撞体和刚体的概念混合在一起——NxActor类身兼两个用途（虽然刚体的物理性质是分开储存于NxBodyDesc的）。在两个SDK中，都可以令一个刚体的位置及定向固定于空间中，其意义是让该刚体不参与动力学模拟，仅作为可碰撞体之用。

尽管这是一个紧密的整合，多数物理SDK都会尝试分离碰撞库和刚体动力学模拟。这么做可以把碰撞系统作为独立库使用（对于不需要物理但需要检测碰撞的游戏尤其重要）。另外**理论上**，游戏工作室也可以更换物理SDK的整个碰撞系统，而无须重写动力学模拟。（实践上这可能比所说的困难!）

12.3.3 关于形状的概念

在**形状**的日常概念背后有丰富的数学理论（见维基百科[17]）。在我们的应用上，形状可以理解为一个由边界所指明的空间区域，能清楚界定形状之**内外**。在二维空间中，形状含其面积，而其边界则是由1条曲线、或3条或以上直线（这就是**多边形**/polygon）所定义的。在三维空间中，形状含体积，其边界不是曲面便是由多边形所组成的（这就称为**多面体**/polyhedron）。

必须注意，有些游戏对象的类型，如地形、河流或薄墙，最好以**表面**（surface）来表示。在三维空间，表面是一个二维几何实体，有**前后**之分，但无内外之分。表面的例子有平面（plane）、三角形、细分表面，以及由一组相连的三角形或多边形所构成的表面。多数碰撞SDK提供表面原型的支持，并把形状的语意扩展至包含闭合体积及开放表面。

碰撞库常会提供可选的挤压（extrusion）参数，使表面含有体积。这种参数指明表面该有多“厚”。这么做也能帮助减少细小高速物体与无穷薄表面错失碰撞的情况（这俗称为“子弹穿纸”问题，见12.3.5.7节）。

12.3.3.1 相交

我们对**相交**都有直观的概念。技术上来说，相交/交集（intersection）[18]术语来自集合论（set theory[19]）。两集合的交集是它们共有成员所组成之集合。在几何学上，两形状的交集仅仅是同时位于两形状中所有点的（无穷大的）集合。

---

[17] http://en.wikipedia.org/wiki/Shape

[18] 译注：intersection在几何学上通常译作相交，而在集合论中译作交集。

[19] http://en.wikipedia.org/wiki/Intersection_(set_theory)

12.3.3.2 接触

在游戏中，我们通常没有兴趣求严格意义上、以点集合表示的交集。反而，我们只是希望得知两个物体是否相交。在碰撞事件发生时，碰撞系统通常会提供额外关于接触性质的信息。例如，这些信息让我们能高效地分离物体，而且能令此过程显得真实。

碰撞系统通常会把接触信息打包成方便的数据结构，对每个检测到的接触生成该结构的实例。例如，Havok以`hkContactPoint`类的实例来传回接触信息。接触信息一般会包含一个**分离矢量**（separating vector），我们可以把物体沿这个矢量移动，就能高效地把物体脱离碰撞状态。接触信息通常也会包含这两个正接触的碰撞体的信息，包括双方的形状，甚至可能包含这些形状接触的特征[20]。碰撞系统也可能会传回额外的信息，例如两个碰撞体投影在分离矢量上的速度。

12.3.3.3 凸性

在碰撞检测范畴里，最重要的概念之一是分辨**凸**（convex）和**非凸**（non-convex，即**凹**/concave）的形状。技术上来说，凸形状的定义是，由形状内发射的光线不会穿越形状表面两次或以上。有一个简单办法判断形状是否为凸，我们可以想象用保鲜膜包裹形状，若形状是凸的，那么保鲜膜下便不会有气囊。在二维空间，圆形、矩形、三角形都是凸的，但“吃豆人（Pac Man）”不是凸的。此概念可同样地推广至三维。

凸性（convexity）是很重要的属性。我们稍后会见到在凸形状之间检测相交，一般会比凹形状简单而且需较少运算。要阅读更多关于凸形状的信息，可参考维基百科[21]。

12.3.4 碰撞原型

碰撞检测系统一般可能只支持有限的形状类型。有些碰撞系统称这些形状为**碰撞原型**（collision primitive），因为这些形状可作为基本组件构成更复杂的形状。本节会简单介绍几种最常见的碰撞原型。[22]

---

[20]译注：这里的特征（feature）是指顶点、棱、面。例如，一个多面体的顶点和另一个多面体的面接触。

[21]`http://en.wikipedia.org/wiki/Convex`

[22]译注：以下介绍的形状中，有几种称为包围体积（bounding volume）。但是在碰撞检测中，不一定要用一个完全包围原本物体的体积，只需按游戏性所需逼近原来的形状便可。例如，在格斗游戏中，角色的受击体积可以比原来的网格小，不然，打到包围体积内的空气部分也会当作命中。然而，在可见性判断（见10.2.7.1节）中，必须构建完全包围原本物体的包围体积。

12.3.4.1 球体

球体（sphere）是最简单的三维体积。如读者所料，球体是最高效的碰撞原型。球体以其球心和半径表示。这些信息可以方便地包裹在一个四元素浮点矢量中，这种矢量对SIMD数学库能运作得特别好。

12.3.4.2 胶囊体

胶囊体（capsule）是药丸形状的体积，由一个圆柱体加上两端的半球所组成。胶囊体可想象为一个**扫掠球体**（swept sphere）[23]——把一个球体从A点移到B点所勾勒的形状。（然而，对比静态胶囊体和随时间移动的球体所产生的胶囊体形状，两者有些重要区别，不尽等同。）胶囊体通常是由两点和半径表示的（图12.2）。计算胶囊体的相交比圆柱体和长方体高效，因此胶囊体常用来为接近圆柱状的物体建模，例如人体的四肢。

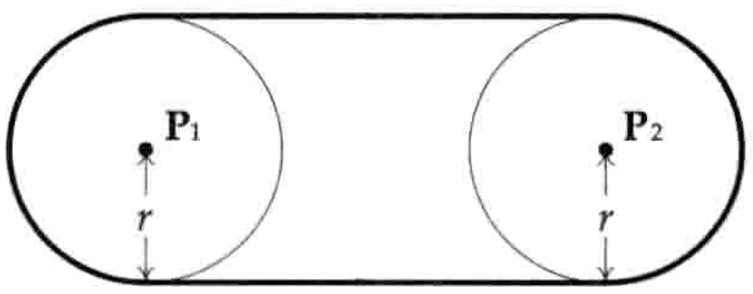


图 12.2: 可用两点和半径表示胶囊体。

12.3.4.3 轴对齐包围盒

轴对齐包围盒（axis-aligned bounding box，AABB）是一个矩形的体积（技术上称为**长方体**/cuboid），其6个面都与坐标系统的轴平行。当然，一个盒子与某坐标系的轴对齐，并不一定会与另一坐标的轴对齐。所以我们谈及AABB时，只是指它在某特定（一个或多个）坐标帧里与轴对齐。

AABB可以方便地由两个点定义：一个点是盒子在3个主轴上最小的坐标，而另一个则是最大的坐标[24]，参见图12.3。

AABB的主要优点在于，可以高速地测试和另一AABB是否相交。而AABB的最大限制在于，它们必须一直保持与轴对齐，才能维持这个运算上的优势。这意味着，若使

[23]译注：更准确的名字是线段扫掠球体（line swept sphere，LSS）。

[24]译注：另一可行的AABB表示方式是储存其最小坐标及尺寸。例如，Win32里的RECT结构和Java AWT的Rectangle类都可当作是二维的AABB，前者使用最小最大坐标(left, top, right, bottom)，而后者则是以最小坐标及尺寸表示(left, top, width, height)。两种表达方式在不同的运算上的效能各有优劣，但最小最大表示法的相交运算通常较快，所以多获游戏引擎采用。此外，AABB还可采用如OBB的中心点、半长度方式表示。

用AABB逼近游戏中某物体的形状，当物体旋转时便需重新计算其AABB。就算某物体大概是长方体形状的，当它旋转至偏离原来的轴时，其AABB便会退化为很差的逼近形状，如图12.4所示。

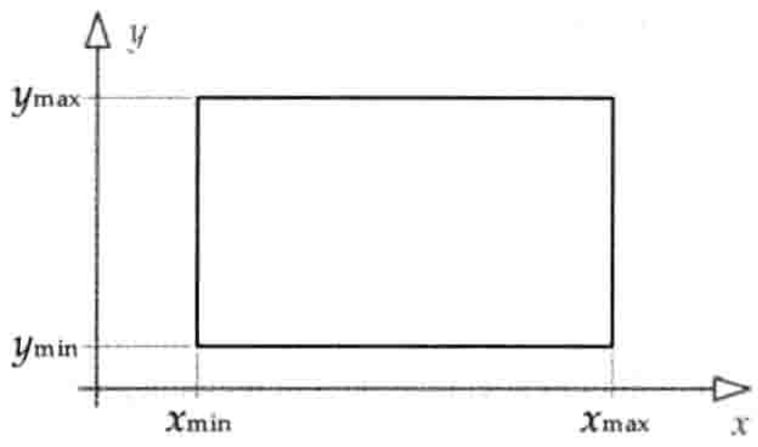


图 12.3: 一个轴对齐包围盒。

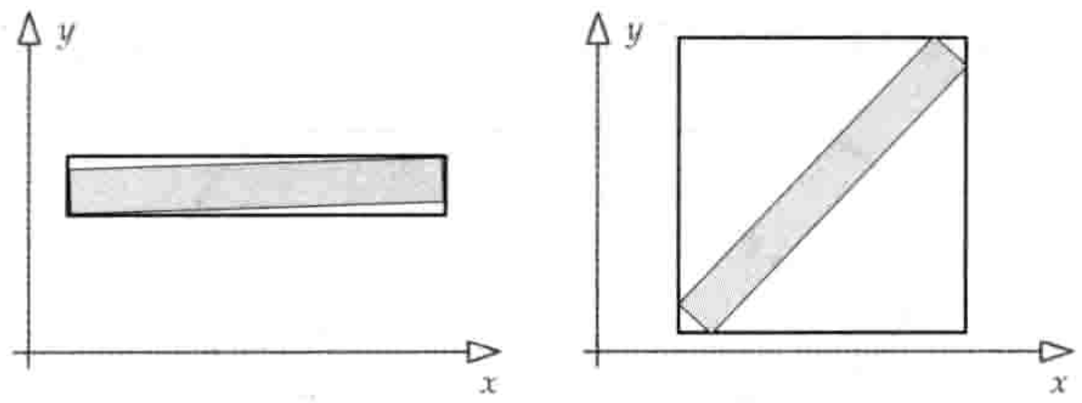


图 12.4: AABB只适用于逼近长方形物体，而且物体的主轴大约与坐标系统的轴对齐。

12.3.4.4 定向包围盒

若我们容许把轴对齐的盒子于其坐标系中旋转，便会得到定向包围盒（oriented bounding box，OBB）。OBB通常会表示为3个“半尺寸”（半宽、半长、半高）再加上一个变换，该变换把盒子的中心定位，并定义了盒子相对坐标系的定向[25]。OBB是一种常用的碰撞原型，因为它能较好地切合任意定向的物体，而其表示方式仍算简单。

12.3.4.5 离散定向多胞形

离散定向多胞形（discrete oriented polytope，DOP）是比AABB及OBB更泛化的形状。DOP是凸的胞形，用来逼近物体的形状。构建DOP的方法之一是，把多个置于无穷远的平面依其法矢量滑动，直至与所需逼近的物体接触。AABB是一个6-DOP，其各个平面法矢量与坐标轴平行。OBB也是6-DOP，但其平面法矢量与物体天然的主轴平行。而k-DOP是由

[25]译注：其实“半尺寸”也可以作为变换中的缩放。那么就可以用单个仿射矩阵表示OBB，使一个单位立方体变换至所需的OBB。这个表示方式比较紧凑，可置于3个SIMD矢量中。

任意k个平面所构成的形状。一个构建DOP的方法是，先建立物体的OBB，再把角及/或边以45°切割，加入更多的平面试图做一个更紧密的逼近。图12.5展示了一个k-DOP例子。

12.3.4.6 任意凸体积

多数碰撞引擎容许三维美术人员在Maya类软件中构建任意凸体积。美术人员用多边形（三角形或四边形）制作形状，然后使用一个离线工具分析那些三角形，确保它们实际上组成了一个凸多面体[26]。若形状合乎凸性测试，其三角形就可以转换为一组平面（本质上是k-DOP），以k个平面方程表示，或k个点加上k个法矢量。（若发现它是非凸的，可以用多边形汤表示，详见下一节。）图12.6展示了一个任意凸体积（arbitrary convex volume）。

凸体积的相交测试比我们介绍至今的简单几何原型都更耗时。然而，我们将会在12.3.5.5节看到，几种高效的求相交算法如GJK，都可以用于这些形状，因为这些都是凸形状。

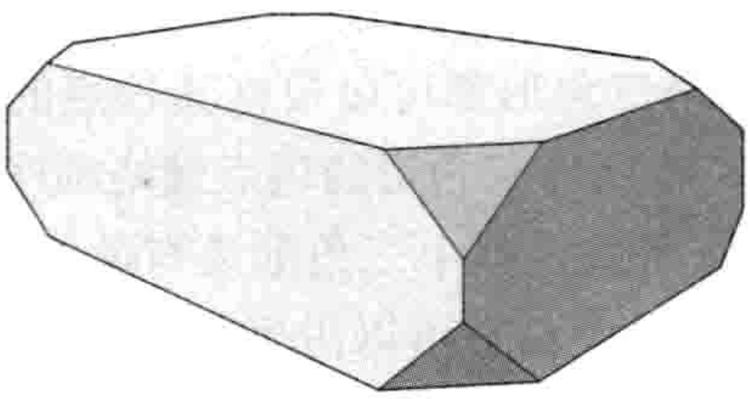


图 12.5: 切去OBB的8个角就成为14-DOP。

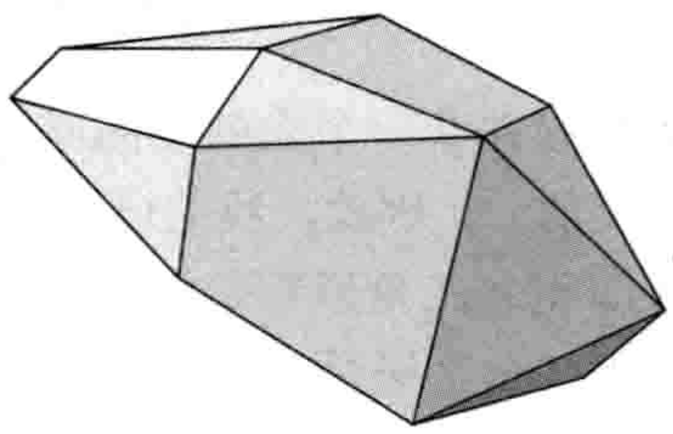


图 12.6: 由相交平面组成的任意凸体积。

12.3.4.7 多边形汤

有些碰撞系统也会支持完全任意、非凸的形状。这些形状通常是由三角形或其他简单多边形所构成的。因此，这类形状通常称为**多边形汤**（polygon soup或poly soup）。多边形汤常用于为复杂的静态几何建模，例如地形或建筑物（图12.7）。

如读者所料，使用多边形汤做碰撞检测是最费时的碰撞检测。碰撞引擎必须对每个三角形进行测试，并且要处理相邻三角形的共棱所做成的伪相交。因此，多数游戏会尝试做出限制，仅把多边形汤应用在不参与动力学模拟的物体。

[26] 译注：具体来说，要检测每个三角形所表示的平面，其法线方向外不可以有其他三角形。另外，还需检查形状是闭合的，以及无退化三角形等。

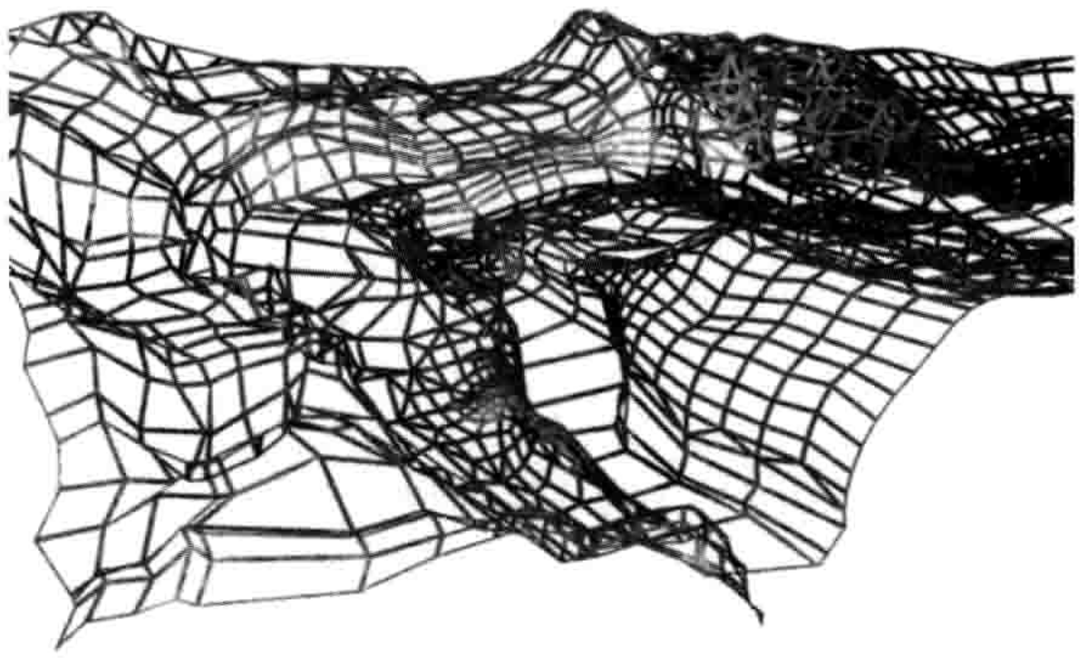


图 12.7: 多边形汤常用于为复杂的静态表面建模，如地形或建筑。

多边形汤有内外之分吗

与凸、简单形状不同，多边形汤不一定表示一个体积，它也可以表示一个开放的表面。多边形汤通常不包含足够信息，供碰撞系统判断它是闭合体积还是开放表面。这样会导致难以决定该用什么方向分离穿透中的物体。

还好这并非一个太棘手的问题。多边形汤里的每个三角形都可以根据其顶点的缠绕顺序来定义前后。因此，我们可以小心地构建多边形汤，令所有三角形的顶点缠绕顺序都是一致的（即相邻三角形都是面向大致相同的方向）。那么可以令整个三角形汤有前后的定义。若我们把三角形汤是开放或闭合的信息储存下来（假设这个信息可以由离线工具查明），那么可以把闭合形状的“前”和“后”理解为“外”和“内”（或是相反，视乎建构多边形汤时的惯例而定）。

对于某些**开放**的多边形汤形状（即表面），我们也可以“仿造”内外的信息。例如，若游戏中的地形是由开放的多边形汤所表示的，那么可以悉随专便，设定表面的前面是指向远离地球的方向。这就意味着“前”一直对应着“外”。要成功实践，我们可能需要定制碰撞引擎，令引擎知悉我们所选择的惯例。

12.3.4.8 复合形状

有些物体用单个形状来逼近并不足够，用**一组**形状来逼近就不错。例如，一张椅子的碰撞体可以用两个盒子建模——一个盒子包围椅背，另一个盒子包围椅座位及四脚，如图12.8所示。

为非凸物体建模时，复合形状（compound shape）经常可作为多边形汤的高效代替品。而且，一些碰撞系统在碰撞测试时，会把复合形状的凸包围体积作为整体，从而获益。

在Havok中，这称为**中间阶段**（midphase）碰撞检测。如图12.9所示，碰撞系统首先测试两个复合形状的凸包围体积。如果它们不相交，便完全无须测试子形状间的碰撞。

图 12.8: 把椅子建模为两个相连的盒子形状。

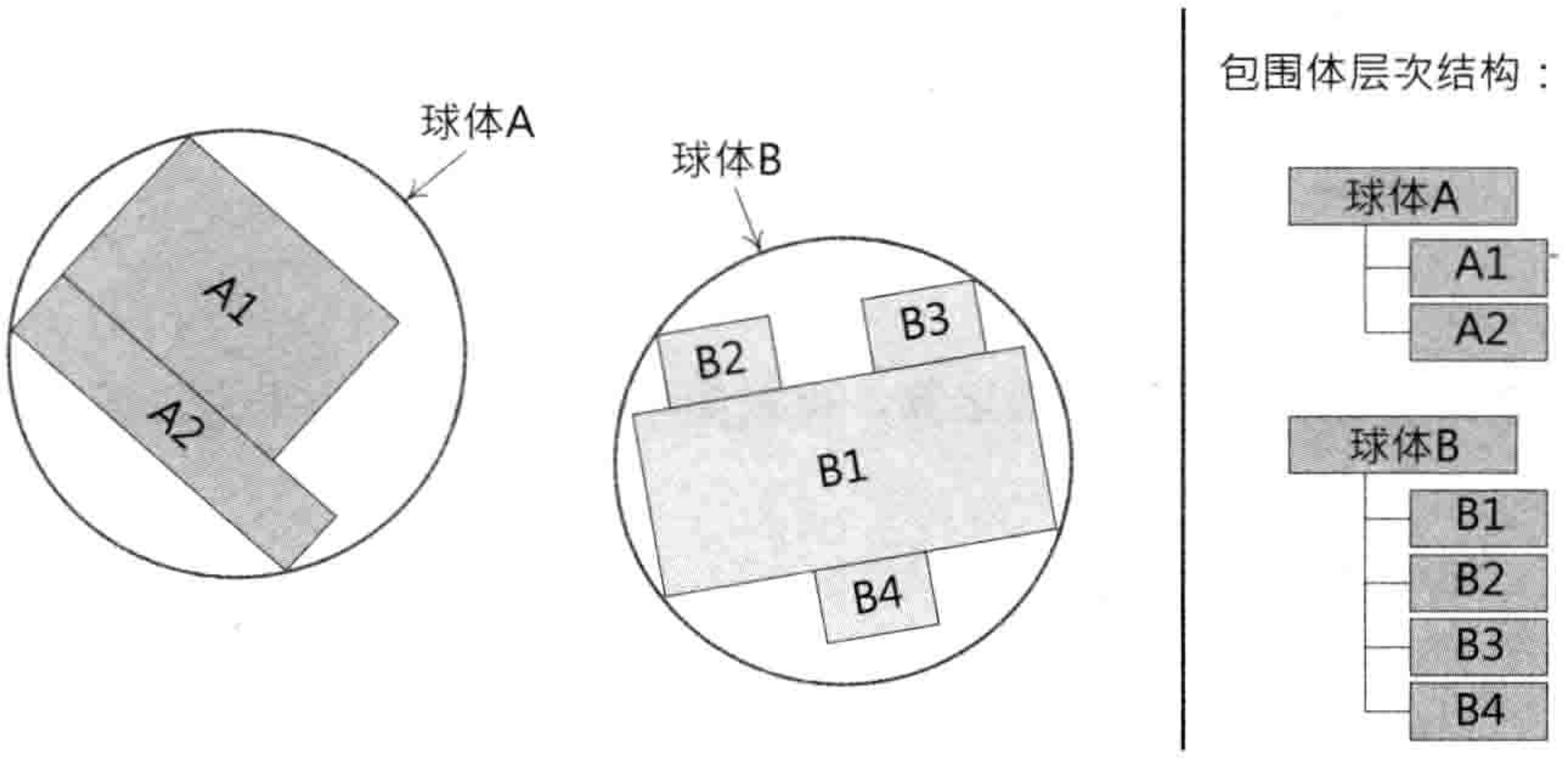


图 12.9: 当一对复合形状的凸包围体积相交（图中是球体A和B），碰撞系统才需检测它们的子形状是否相交。

12.3.5 碰撞测试及解析几何

碰撞系统应用到**解析几何**（analytical geometry）中三维体积及表面的数学描述，计算形状间是否相交。这个既深且广的研究领域的更多详情可参考维基百科[27]。在本节中，我们会简介解析几何背后的概念，展示一些常见例子，然后讨论为任意凸多面体[28]而设的泛用GJK相交测试算法。

---

[27] http://en.wikipedia.org/wiki/Analytic_geometry

[28] 译注：其实只要是凸体积都可使用GJK算法，例如球体、胶囊体。

12.3.5.1 点与球体的相交

要判断一个点**p**是否在球体中，只需生成一个自球心至该点的分离矢量**s**，然后量度该矢量的长度[29]。若长度大于球体半径，则该点位于球体之外，否则就是球体之内：

$$\mathbf{s} = \mathbf{c} - \mathbf{p}$$

若$|\mathbf{s}| \leqslant r$, 则**p**位于球体内。

12.3.5.2 球体与球体的相交

判断两个球体是否相交，几乎和判断点是否在球体中一样简单。再次，我们生成连接两个球心的分离矢量**s**。取其长度，与两球体半径之和做比较。若分离矢量的长度小于或等于两半径之和，那么球体是相交的，否则两者不相交：

$$\begin{aligned} &\mathbf{s} = \mathbf{c}_1 - \mathbf{c}_2 \\ &\text{若}|\mathbf{s}| \leqslant (r_1 + r_2), \text{则两球体相交。} \end{aligned} \tag{12.1}$$

要避免计算长度时所需的平方根运算，可以简单地把方程两边平方。那么方程(12.1)就会变成：

$$\begin{aligned} &\mathbf{s} = \mathbf{c}_1 - \mathbf{c}_2 \\ &|\mathbf{s}|^2 = \mathbf{s} \cdot \mathbf{s} \\ &\text{若}|\mathbf{s}|^2 \leqslant (r_1 + r_2)^2, \text{则两球体相交。} \end{aligned}$$

12.3.5.3 分离轴定理

多数碰撞检测系统都会大量使用**分离轴定理**（separating axis theorem）[30]。此定理指出，若能找到一个轴，两个凸形状于该轴上的**投影**不重叠，就能确定两个形状不相交。若这样的轴并不存在，**并且**那些形状是凸的，则可以确定两个形状相交。（若形状为凹，那么就算找不到分离轴，形状也可能不相交。这是我们偏好在碰撞系统使用凸形状的原因之一。）

此定理最容易在二维空间上图解说明。直觉地，若能找到一条直线，令物体A完全在直线的一方，而物体B完全在另一方，那么A和B便不重叠。这样的直线称为**分离线**，它必定

[29]译注：如同下节所述，可比较分离矢量长度的平方和球体半径的平方，这样可节省较耗时的平方根运算。

[30]http://en.wikipedia.org/wiki/Separating_axis_theorem

垂直于分离轴。因此若我们能找到一条分离线，只要观察垂直于分离线的轴上的形状投影，就能更容易说服我们此定理的正确性。

二维**凸**形状在一个轴上的投影，就有如物体在一条细线上的阴影。这些投影必然是在轴上的一个线段，代表物体在轴方向上的最大范围。我们也可以把投影视为轴上的最小及最大坐标，这可以写成闭合区间$[c_{\min}, c_{\max}]$。如图12.10所示，若在两形状之间存在分离线，它们在分离轴上的投影不会重叠相交。然而，若投影重叠，那就不算一个分离轴。

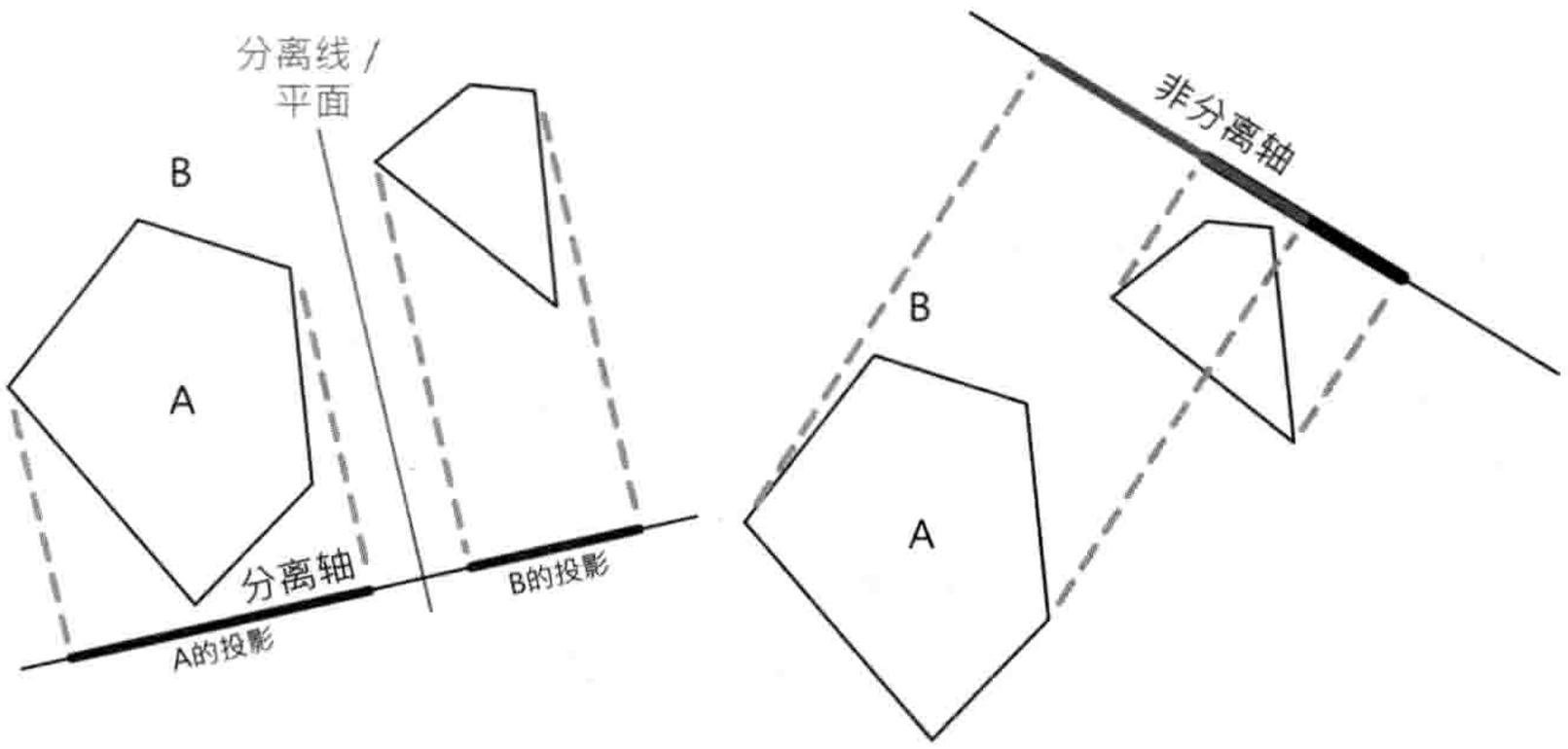


图 12.10: 两个形状在分离轴上的投影总是不相交的。同样的形状投影在非分离轴上可能是相交的。若不存在分离轴，两个形状就是相交的。

在三维空间中，分离线变成分离平面，但分离轴仍然是一个轴（即一个三维方向[31]）。而三维**凸**形状在轴上的投影是一线段，可用完全闭合区间$[c_{\min}, c_{\max}]$表示。

有些形状类型的特性，使我们可以容易地找到潜在的分离轴。要检测A和B两个形状是否相交，我们可以把这些形状逐一投影到各个潜在分离轴，并检查两个投影区间$[c^A_{\min}, c^A_{\max}]$、$[c^B_{\min}, c^B_{\max}]$是否相交（即是否重叠）。数学上来说，只有当$c^A_{\max} < c^B_{\min}$或$c^B_{\max} < c^A_{\min}$时，两个区间不相交。若于潜在分离轴上的投影区间不相交，那么我们就算是找到一个合法的分离轴，并得知两个形状不相交。

此原理可在球体与球体测试上示范。若球体不相交，那么连接两球心的线段，其方向必然是一个合法的分离轴（虽然视乎球体分隔的距离，还可能有其他分离轴）。要用图说明这个情况，考虑两球体极为接近，但又未接触。在此情况下，**唯一**的分离轴便是球心至球心线段的方向。随球体互相分开，便可用更大幅度向两个方向旋转分离轴，如图12.11所示。

[31] 译注：原文“i.e., an infinite line（无穷直线）”并不准确。因为直线会通过固定的点，而分离轴仅表示一个方向。

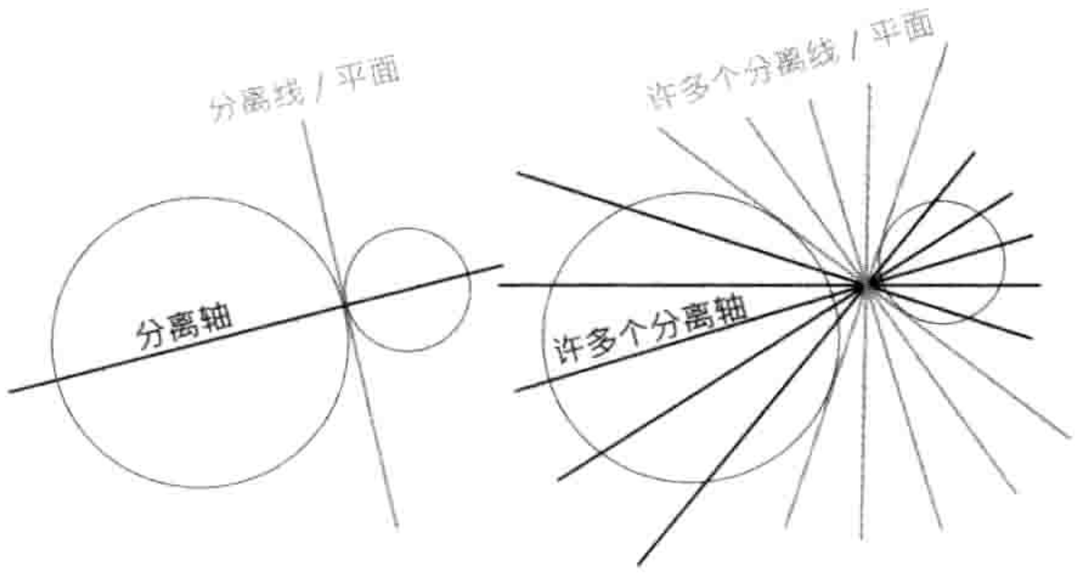


图 12.11: 当两个球体被无穷短的距离分开，唯一的分离轴就是通过两个球心的线段。

12.3.5.4 AABB与AABB的相交

要判断两个AABB是否相交，我们可再使用分离轴定理。由于AABB的面与一组坐标轴平行，所以若存在分离轴，它必然是3个坐标轴之一。

因此，要检测两个AABB的相交，我们只需要检查它们在每个轴上的最小、最大坐标。设两个AABB称为A、B，在x轴上两个AABB的区间为$[x^A_{\min}, x^A_{\max}]$及$[x^B_{\min}, x^B_{\max}]$，在y轴和z轴上也有相应的区间。若在**3个轴**上的区间**都**重叠，那么两个AABB是相交的；否则它们不相交。图12.12显示相交和非相交的例子（为用图表示，简化为二维）。对于AABB碰撞的更深入讨论，可参考这篇文章[32]。

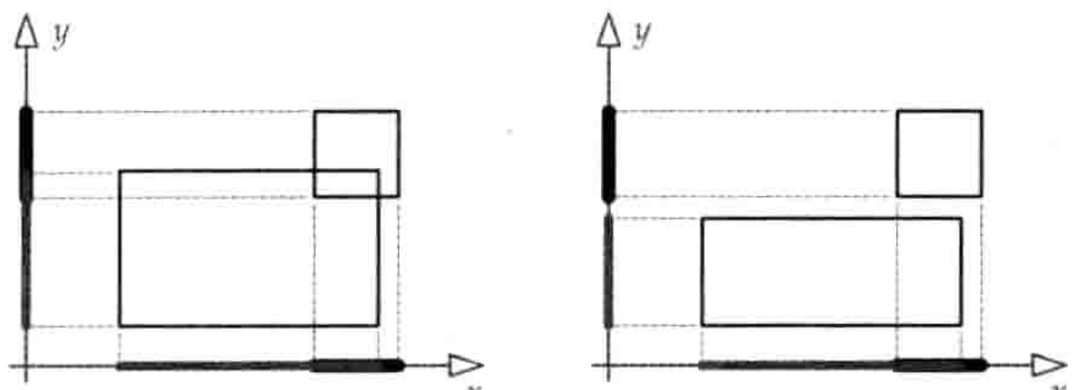


图 12.12: 二维的AABB相交 / 不相交的例子。注意右图中的AABB仅在x轴上相交，在y轴上不相交。

12.3.5.5 检测凸碰撞：GJK算法

有一个非常高效的算法，可检测任意凸**多胞形**（polytope，即二维中的凸多边形、三维中的凸多面体[33]）。该算法的3位发明者是密歇根大学的E. G. Gilbert、D. W. Johnson及S.

[32] http://www.gamasutra.com/features/20000203/lander_01.htm

[33] 译注：多胞体是由线/平面/超平面组成的形状，但如之前的译注所述，GJK还适用于其他凸形状，如球体、胶囊体。唯一要求是替形状提供一个支持函数（supporting function），详见后文。

S. Keerthi，所以称之为GJK算法。已有许多文献描述此算法及其变种，包括原始论文[34]、由Christer Ericson制作的优秀SIGGRAPH PPT演示[35]、Gino van den Bergen制作的另一优秀PPT演示[36]。然而，对此算法最易理解（及最富娱乐性）的描述可能是Casey Muratori的线上教学视频“实现GJK（Implementing GJK）”[37]。因为这些材料都非常好，笔者希望在此让读者浅尝此算法的精髓，余下的细节可参考上述的文献及网站。

GJK算法依赖一个称为**闵可夫斯基差**（Minkowski difference）的几何运算。这个听上去好像很厉害的运算，其实十分简单。把A图形中的所有点，与B图形的所有点成对相减，得出的集合$\{(\mathbf{A}_i - \mathbf{B}_j)\}$便是闵可夫斯基差。

闵可夫斯基差的用处在于，当应用至两个形状[38]，当且仅当两个形状相交，其闵可夫斯基差会**包含原点**。此命题的证明有点儿超出本书范围，然而我们可以意会为何它是对的。若两个形状相交，实际上就是有些在A里的点**也在**B的范围内。那么当A里的点减去B里的点时，两形状共有的点相减后便会是0，所以两个形状的闵可夫斯基差会含有原点，当且仅当它们含共有的点。图12.13说明了这一情况。

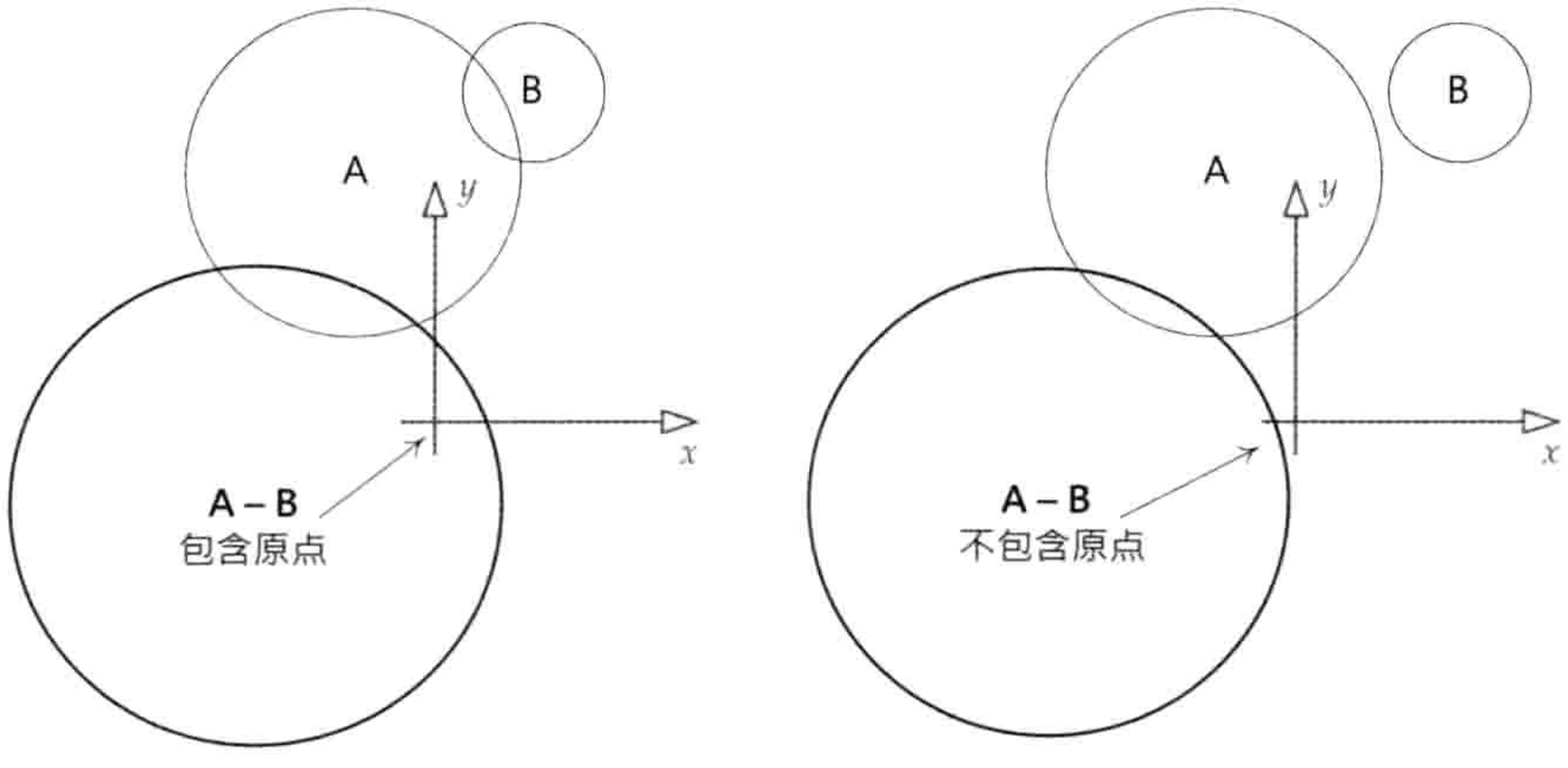


图 12.13: 两个相交凸形状的闵可夫斯基差包含了原点，但不相交形状的闵可夫斯基差不包含原点。

两个凸图形的闵可夫斯基差都是一个凸图形。我们只关心闵可夫斯基差的**凸包**，而不是所有内点。GJK的基本程序是从闵可夫斯基差的凸包内，尝试找出一个包含原点的**四面体**（tetrahedron，即由三角形构成的四边形状）。若可以找到这样的四面体，则两个形状相交；若不能找到这样的四面体，则两个形状不相交。

[34] http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?&arnumber=2083

[35] http://realtimecollisiondetection.net/pubs/SIGGRAPH04_Ericson_the_GJK_algorithm.ppt

[36] http://www.laas.fr/~nic/MOVIE/Workshop/Slides/Gino.vander.Bergen.ppt

[37] http://mollyrocket.com/353

[38] 译注：原文限定为凸形状，但在此段关于闵可夫斯基差的描述中，凸性并非必要条件。而GJK则需要两个形状为凸。

四面体其实只是名为**单纯体**（simplex）的几何物体之一。不要让这个名字吓跑，单纯体只是点的集合。单点单纯体是一个点，两点单纯体是一条线段，3点单纯体是一个三角形，4点单纯体是一个四面体，见图12.14。

图 12.14: 含有1～4个点的单纯体。

GJK是一个迭代式算法，可首先从闵可夫斯基差凸包内任意的四面体开始，然后再尝试建立更高阶、潜在包含原点的单纯体。在循环的每个迭代中，我们从当前的单纯体考虑，判断原点在该单纯体的哪一个方向。然后我们在那个方向搜寻闵可夫斯基差的**支持顶点**（supporting vertex），即凸包中在该方向最接近原点的顶点。我们把该点加进单纯体，产生一个更高阶的单纯体（即点变成线段，线段变成三角形，三角形变成四面体）。若加入新点后能令单纯体包含原点，那么工作完成——两个形状相交。反过来说，若不能找到比当前单纯体更接近原点的支持顶点，那么便可知道此任务永不能完成，也即意味着两个形状不相交。图12.15说明了这个想法。

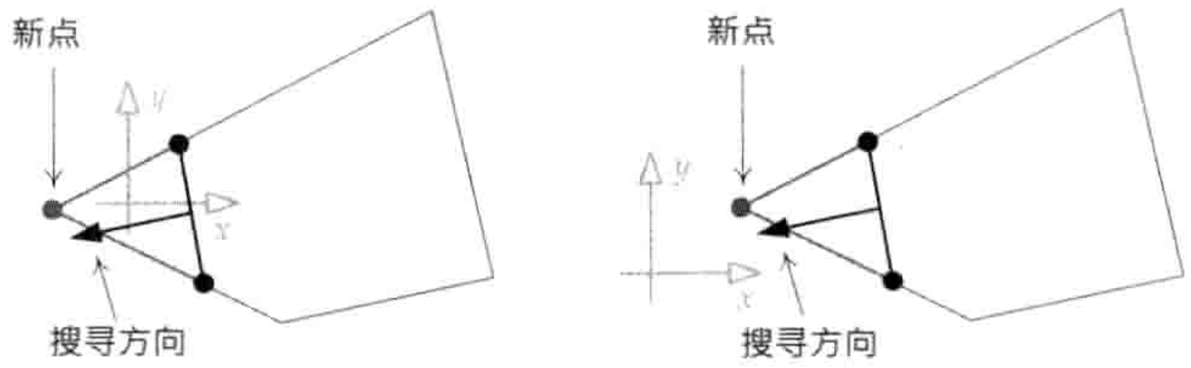


图 12.15: 在GJK算法中，若在当前的单纯体中加入一点就能包含原点，我们就能知道两形状相交。若无支持顶点能令单纯体接近原点，那么两形状就不相交。

要真正理解GJK算法，读者需要参考前面提及的文献和视频。然而，希望本节能激起读者更深入研究的欲望。或者，至少能让读者在派对中抛出“GJK”这个名称去炫耀一番。

12.3.5.6 其他形状对形状组合

本书不会说明更多的不同形状的相交组合，因为在其他著作中已有详细描述，例如[12]、[41]、[9]。然而，读者需要意识到一个重点，就是形状对形状的组合的**数目**十分庞大。事实上，对于N种形状，所需的成对测试便需要$O(N^2)$个。碰撞引擎的复杂度，很大程度上是由大量所需处理的相交组合所造成的。这也是碰撞引擎的作者总是尽量限制碰撞

原型数量的原因，这样做可以大幅度降低所需处理的相交组合数目。（这也是GJK流行的原因——GJK能一举处理**所有**凸形状之间的碰撞检测。而不同形状的唯一区别只在于算法所使用的**支持函数** / support function）。

给定两个任意形状，应如何实现代码选择合适的碰撞测试函数，这是一个实际问题。许多引擎使用**双分派**（double dispatch）方法[39]。单分派（即虚函数）在运行时使用单个对象类型，决定对某抽象函数调用哪一个具体实现。双分派把虚函数的概念扩展至两个对象类型。双分派可以通过二维查找表实现，表的键由两个检测对象的类型组成。此外，双分派也可以实现为两次虚函数调用，第一次由对象A的类型决定调用哪个具体函数，在该函数中再由对象B的类型决定调用哪个具体函数。

接下来看一个真实例子。Havok使用**碰撞代理人**（collision agent，为`hkCollisionAgent`的子类）处理某特定相交测试。具体碰撞代理人包括`hkSphereSphereAgent`类、`hkSphereCapsuleAgent`类、`hkpGskConvexConvexAgent`类等。而`hkpCollisionDispatcher`类负责管理一个二维分派表，内含这些代理人的类型。如读者所料，分派器的任务就是从一对要做碰撞检测的碰撞体，高效地查找出合适的代理人，然后以这两个碰撞体作为参数调用该代理人的函数[40]。

12.3.5.7 检测运动物体之间的碰撞

至今我们只考虑两个静止物体的静态相交测试，物体移动时会更加复杂。在游戏中，运动通常是以离散时间步来模拟的。因此，简单方法是在每个时间步中，将每个刚体的位置和定向当作是静止的，然后把静态相交测试施于这些碰撞世界的“快照（snapshot）”。若物体的移动速度相对其尺寸来说不是太快，此方法是可行的。事实上，此技巧在许多碰撞/物理引擎上行之有效，Havok也是预设使用此方法。

然而，对于较小的高速移动物体，此方法便会失效。现在想象有一个物体，在时间步之间的移动幅度**大于其尺寸**（以移动方向来计算）。若我们把两个相邻的碰撞世界快照重叠观看，便会注意到该快速移动物体在两个快照的像之间会有一段空隙。如果另一物体刚好在空隙之间，便会完全错过碰撞。图12.16展示了这个“子弹穿纸（bullet through paper）”的问题，也可称作“隧穿（tunneling）[41]”。以下会讲述几个常见的应对方案。

---

[39] http://en.wikipedia.org/wiki/Double_dispatch

[40] 译注：译者认为，如果二维分派表所存储的是函数指针，可能会更为高效，因为这样会省去一次虚函数调用。

[41] 译注：此术语可能是借用自量子隧穿效应（quantum tunnelling effect），故用上此译法。这效应指粒子在量子力学中，能穿过经典力学中无法穿过的“墙壁”。

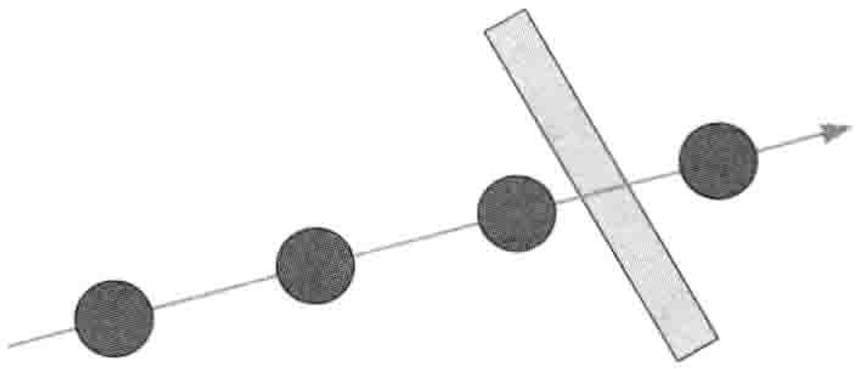


图 12.16: 细小快速的物体移动时，其路径可能会在碰撞世界的连续快照中留下空隙，这意味着有可能完全错过碰撞。

扫掠形状

避免隧穿的方法之一是利用**扫掠形状**（swept shape）。扫掠形状是一个形状随时间从某点移动至另一点所形成的形状。例如，扫掠球体是胶囊体，而扫掠三角形则是一个三角柱体（见图12.17）。

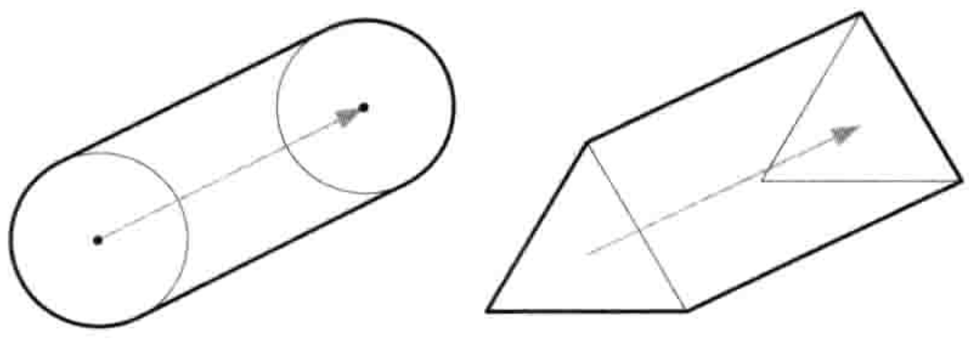


图 12.17: 扫掠球体是胶囊体，扫掠三角形是一个三角柱体。

检测相交时，由测试碰撞世界的静态快照，改为测试形状从上一个快照的位置及定向移动至当前快照所形成的扫掠形状。此方法等同对快照间的形状做**线性插值**，因为我们通常以快照间的直线线段扫掠。

当然，线性插值不一定是高速移动碰撞体的良好逼近值。若碰撞体以曲线路径移动，那么理论上我们应该按其曲线路径扫掠该形状。然而，凸形状以曲线扫掠所产生的形状并不是凸的，所以这会令碰撞测试更复杂及需要更多运算。

此外，若要扫掠的形状正在旋转，即使扫掠的路径为直线，所得的扫掠形状也不一定是凸的。如图12.18所示，我们总是**可以**对形状在之前及当前的快照进行线性外插，尽管结果并不一定能准确表示形状在时间步中的移动范围。换句话说，线性插值不一定适合旋转中的形状。因此，除非形状不能旋转，扫掠形状的相交测试相比基于静态快照的来说，更复杂又需要更多运算。

扫掠形状是有用的技术以确保碰撞不会在快照间错过。然而，若是沿曲线路径进行线性插值，又或涉及旋转的碰撞体，其测试结果一般是不准确的，所以可能要根据游戏所需使用更细密的技术。

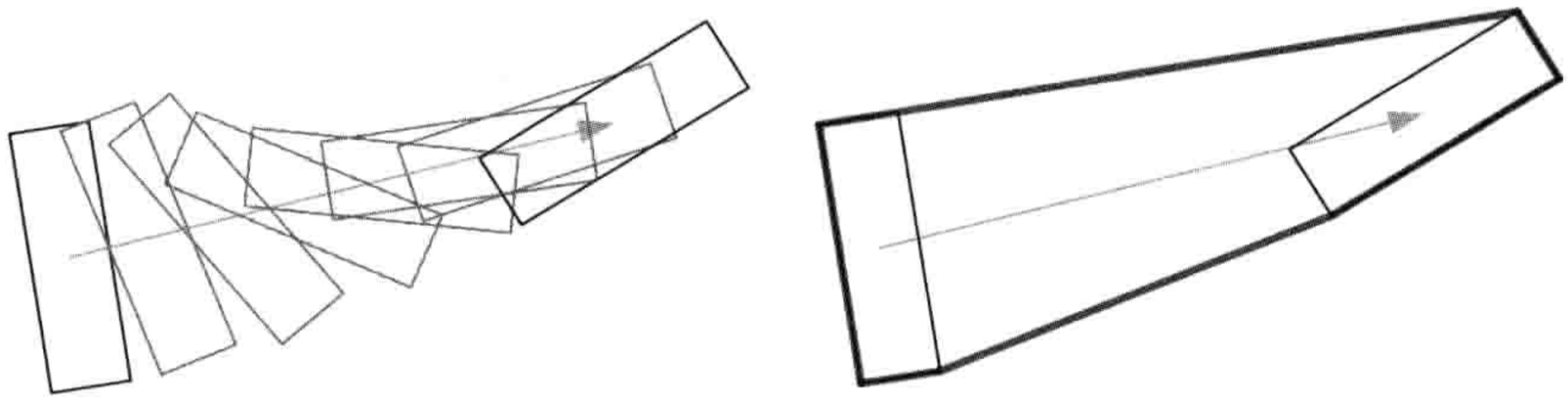


图 12.18: 通过线段扫掠的旋转物体，不一定产生凸形状（左图）。运动的线性插值能生成凸形状（右图），但可能相当不接近时间步内的发生情况。

连续碰撞检测

处理隧穿的另一方法是使用**连续碰撞检测**（continuous collision detection，CCD）技术。CCD的目标是对两个移动物体于某时间区间内，求得最早的**冲击时间**（time of impact，TOI）。

CCD算法一般是迭代式的。我们维护每个碰撞体于上个时间步及当前时间的位置及定向。这些信息可以用来各自对位置及定向进行线性插值，以产生在该时间区间内任一时间点上的碰撞体变换。然后算法搜寻在移动路径上的最早TOI。常用的搜寻算法包括Brian Mirtich的**保守前进法**（conservative advancement）、向闵可夫斯基和（Minkowski sum）投射光线，以及考虑每对形状特征的最小TOI。索尼娱乐的Erwin Coumans在一篇文章[42]中讲述了这些算法，并提供了一个保守前进法的变种。

12.3.6 性能优化

碰撞检测是CPU密集的工作，原因有二。

1. 判断两个形状是否相交，所需的计算是非平凡的。
2. 多数游戏世界含有大量的物体，随物体数量递增，所需的相交测试会迅速增长。

要检测n个物体之间的相交，蛮力法需要测试逐对物体，造成一个$O(n^2)$算法。然而，实践上有更高效的算法。碰撞引擎通常会使用某种空间散列（spatial hashing）方法[43]、空间细分（spatial subdivision）或层次式包围体积（hierarchical bounding volume），以降低所需的相交测试次数。

---

[42] http://www.continousphysics.com/BulletContinousCollisionDetection.pdf

[43] http://research.microsoft.com/~hoppe/perfecthash.pdf

12.3.6.1 时间一致性

常见的优化技巧之一是利用**时间一致性**（temporal coherency），也称为**帧间一致性**（frame-to-frame coherency）。当碰撞体以正常速率移动，在两时步中其位置及定向通常会很接近。通过跨越多帧把结果缓存，我们可以避免每帧重新计算一些类型的信息。例如在Havok中，碰撞代理人（`hkpCollisionAgent`）通常会在帧之间延续，令它们能重复使用之前时间步的运算，只要相关的碰撞体运动没导致这些计算结果无效。

12.3.6.2 空间划分

空间划分（space partitioning）的基本思路是通过把空间切割成较小的区域，以大幅降低需要做相交测试的碰撞体。若我们（无须很费时就）能判断一对碰撞体不属同一区域，那么我们就不需要对它们进行更细致的相交测试。

有多种层阶式的方案可以为优化碰撞检测划分空间，例如八叉树（octree）、二元空间分割树（binary space partitioning/BSP tree）、kd树、球体树（sphere tree）等。这些方案把空间以不同方式细分，但它们都是层阶式的，由位于树根的大分区域逐层细分，直至细分至足够细致的分区。然后就可以遍历该树，以找出及测试潜在碰撞的物体组别，做实际的相交测试。因为树把空间剖分了，所以当向下遍历一个分支时，该分支的物体不可能与其他兄弟分支的物体碰撞。

12.3.6.3 粗略阶段、中间阶段、精确阶段

Havok使用三阶段的方式，缩减每时步中所需检测的碰撞体集合。

- 首先，用粗略的AABB测试判断哪些物体有机会碰撞。这称为**粗略阶段**（broad phase）碰撞检测。
- 然后，检测复合形状的逼近包围体。这称为**中间阶段**（midphase）碰撞检测。例如，某复合形状由3个球体所构成，该复合形状的包围体积可以是包围那3个球体的更大球体。复合形状可能含有其他复合形状，因此，一般化来说，复合碰撞体含有一个包围体积层阶结构。中间阶段遍历此层阶结构，以找出可能会碰撞的子形状。
- 最后测试碰撞体中个别碰撞原形是否相交。这称为**精确阶段**（narrow phase）碰撞检测。

扫掠裁减算法

所有主要的碰撞/物理引擎（如Havok、ODE、PhysX）的粗略阶段碰撞检测会使用一个名为**扫掠裁减**（sweep and prune）[44]算法。其基本思路是对各个碰撞体的AABB的最小、最大坐标在3个主轴上排序，然后通过遍历该有序表检测AABB间是否重叠。扫掠裁减算法可以利用帧间一致性把$O(n \log n)$的排序操作缩减至$O(n)$的预期运行时间。帧间一致性也可以帮助旋转物体时更新其AABB。

12.3.7 碰撞查询

碰撞检测的另一任务是回答有关游戏世界中碰撞体积的假想问题（hypothetical question），例如：

- 若从玩家武器的某方向射出子弹，若能击中目标，那目标是什么？
- 汽车从A点移动至B点是否会碰到任何障碍物？
- 找出玩家在给定半径范围内的所有敌人对象。

一般而言，这些操作称为**碰撞查询**（collision query）[45]。

最常用的查询类型是**碰撞投射**（collision cast），或简称作**投射**（cast）。（常见同义词还有**追踪**/trace、**探查**/probe。）投射用于判断，若放置某假想物体于碰撞世界，并沿光线（ray）或线段移动，是否会碰到世界中的物体。投射与正常的碰撞检测操作有别，因为投射的实体并不真正存在于碰撞世界，它完全不会影响世界中的其他物体。这就是为什么我们称，碰撞投射是回答关于世界中碰撞体的**假想**问题。

12.3.7.1 光线投射

最简单的碰撞投射是**光线投射**（ray cast），虽然此术语实际上有点不准确。我们实际上要投射的是有向线段（directed line segment），换句话说，我们的投射是有起点（$\mathbf{p}_0$）和终点（$\mathbf{p}_1$）的。（多数碰撞系统不支持无限长的光线，因为它们使用到以下介绍的参数公式。）投射线段会用于检测与碰撞世界物体是否相交。若发生相交，就会传回接触点或点集。

光线投射系统的线段通常以起点（$\mathbf{p}_0$）以及增量矢量（$\mathbf{d}$）来描述，而起点加上增量矢量后就会得出终点（$\mathbf{p}_1$）。此线段上的任何点都可在以下的**参数方程**（parametric equation）

[44] http://en.wikipedia.org/wiki/Sweep_and_prune

[45] 译注：在计算机几何中，这类问题可归纳为几何查询（geometric query）问题。

中求得，当中参数t值的范围是0～1：

$$\mathbf{p}(t) = \mathbf{p}_0 + t\mathbf{d}, \qquad t \in [0, 1]$$

显然，$\mathbf{p}_0 = \mathbf{p}(0)$，$\mathbf{p}_1 = \mathbf{p}(1)$。此外，沿线段上的每点都可以用唯一的$t$值来指明。许多光线投射API会传回“$t$值”，或是提供一个函数把接触点转换为对应的$t$。

多数碰撞检测系统可以传回**最早的接触**，即最近于$\mathbf{p}_0$的接触点，又即对应t最小值的点。有些系统也能够传回与光线或线段相交的所有接触点的完整列表。其传回的每个接触点信息，通常都会包含t值、其相交的碰撞体的唯一标识符，或会包含其他信息，如该接触点的表面法线、形状或表面的其他相关属性。接触点的数据结构可能是这样的：

```
struct RayCastContact
{
    F32      m_t;                // 此接触点的t值
    U32      m_collidableId;     // 击中哪个可碰撞体?
    Vector   m_normal;           // 接触点的法矢量
    // 其他信息
};
```

光线投射之应用

光线投射在游戏中大量使用。例如，我们可能希望询问碰撞系统，角色A是否能直接看见角色B。为了做出判断，可以简单地从角色A的双眼投射有向线段至角色B的胸口。若光线触碰角色B，那么A就能“看见”B。但若光线在到达角色B**之前**碰到其他物体，便能得知该视线正被阻挡。光线投射也应用在武器系统（如判断子弹是否命中）、玩家机制（如判断角色脚底下是否为地面）、人工智能系统（如视线检测、瞄准、移动查询等）、载具系统（如把车轮依附地面）等。

12.3.7.2 形状投射

另一种对碰撞系统的常见查询，是问一个假想凸形状可以沿一有向线段移动多远才会碰到其他物体。若投射的体积是球体，则称为**球体投射**（sphere cast），更一般的情况称为**形状投射**（shape cast）。（Havok称这些为**线性投射**。）如同光线投射，形状投射也是指定起点$\mathbf{p}_0$、行程距离$\mathbf{d}$，当然还要提供投射的形状类型、大小、定向的信息。

投射一个凸形状时有两个情况要考虑。

1. 投射形状已插入或接触到至少一个其他的碰撞体，因而阻止投射形状移离起点。

2. 投射形状在起点没有与其他碰撞体相交，因此可以自由地沿路径移动一段非零距离。

在第一种情况下，碰撞系统通常会汇报在起始时投射形状与所有相交碰撞体的接触点。这些接触点可能位于投射形状**之内**或于其**表面**，如图12.19所示。

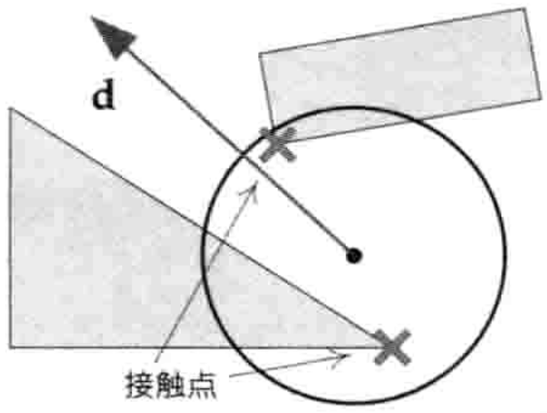


图 12.19: 球体在开始投射时已穿透其他物体，无法再移动，还可能有多于一个接触点。

在第二种情况下，投射形状可能会在碰到物体前，沿线段移动一段非零距离。假设投射形状碰到一些物体，通常是单个碰撞体。然而，若轨道刚刚好，投射形状也有可能会在同时撞击到多个碰撞体。当然，若受击的碰撞体是非凸多边形汤，投射形状便可能会同时碰到多边形汤的多个部分[46]。我们可以安全地说，无论投射什么类型的凸形状，都有**可能**（尽管可能性不大）会产生多个接触点。然而在这种情况下，接触点必然会在投射形状的**表面**，而永不会在形状里面（因为我们知道投射形状在开始移动之前没有插入其他形状中）。见图12.20。

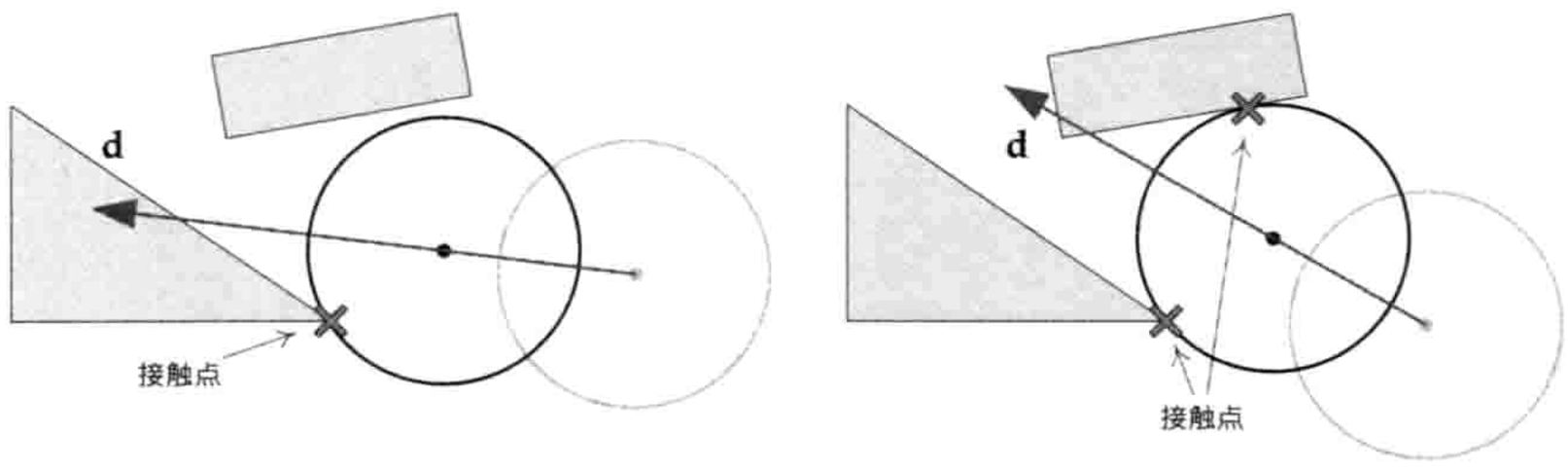


图 12.20: 若投射开始时形状没有穿透任何物体，那么形状就可以沿线段移动一段非零距离，并且接触点（若有）总会在形状的表面。

如同光线投射，有些形状投射的API仅传回投射形状最早碰到的（一个或多个）接触，而其他API可以让形状继续投射，传回所有碰到的接触。图12.21展示了后者。

[46]译注：若两个形状皆凸，它们的接触特征（接触点或棱、面）只有一对，例如一个形状的顶点接触到另一形状的面。

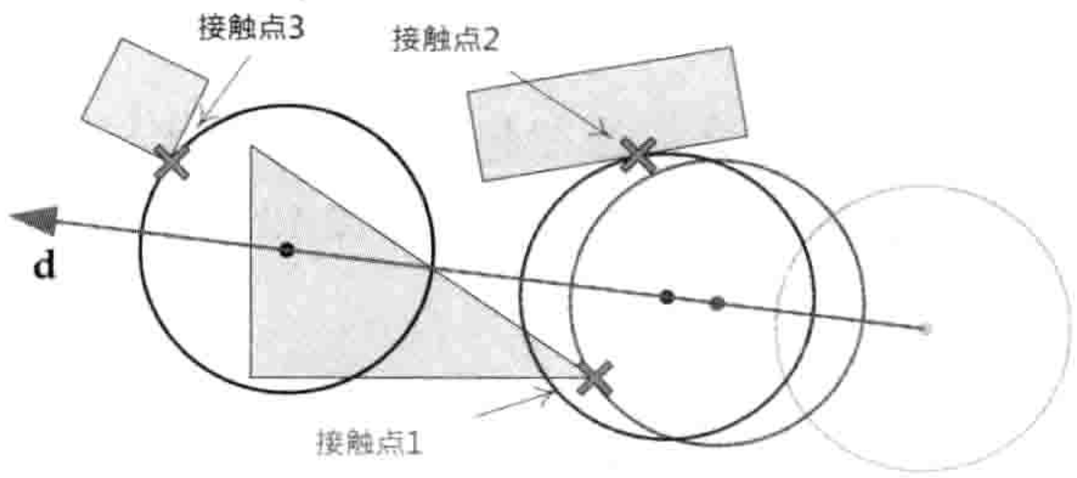


图 12.21: 形状投射API可能传回最早接触到的所有接触点。

形状投射所传回的接触信息会比光线投射的更复杂一点。我们再不能简单地传回一个或多个t值，因为t值只能表示形状沿路径的中心点位置，而不能告之碰撞的表面位置或内部位置。因此，多数形状投射API会同时传回t值及实际接触点，再加上其他相关信息（如受击的碰撞体、接触点的表面法线等）。

不同于光线投射，形状投射系统必须能传回多个接触点。这是由于，即使至传回最早的t值，投射形状仍可能同时接触到游戏世界中多个不同的碰撞体，或是触碰到单个非凸碰撞体的多个点。因此，碰撞系统通常传回接触点的数组或列表数据结构，当中每个元素可能像如下所示：

```
struct ShapeCastContact
{
    F32      m_t;                // 此接触点的t值
    U32      m_collidableId;     // 击中哪个可碰撞体?
    Point    m_contactPoint;     // 实际接触点的位置
    Vector   m_normal;           // 接触点的法矢量
    // 其他信息
};
```

在一些接触点中，我们通常希望分辨每个t值所对应的接触点集合。例如，最早的接触，实际上是以含相同t值的接触点来描述的。必须注意，碰撞系统传回的接触点可能会以t值排序，或是不做排序。如果没有，还是最好手动把结果以t值排序。这样可确保列表第一个接触点会是沿投射路径最早的接触点集合之一。

形状投射之应用

形状投射在游戏中极为有用。球体投射可用于判断虚拟摄像机是否与游戏世界中的物体碰撞。球体或胶囊体投射也常用于实现角色移动。例如，要令角色在崎岖不平的地形上滑行前进，我们可以从角色脚部向移动方向投射一个球体或胶囊体。然后通过第二次投射向上

或向下调整该形状，确保形状仍然保持和地面接触。若球体遇到很短的垂直障碍物，例如行人路的缘石，就可以把球体“突然提升”至缘石之上。若垂直障碍物很高，例如碰到墙壁，那么投射球体就可以沿墙壁水平滑动。投射球体的最终位置就成为角色在下一帧的新位置。

12.3.7.3 Phantom

有时候，游戏需要判断碰撞体是否位于游戏中某些指定体积里。例如，我们可能需要获取玩家角色某半径范围内的所人敌人列表。为此，Havok提供一种特别的，称为**phantom**（幻影）[47]的碰撞体。

phantom的行为有如零距离的形状投射。在任何时刻，我们都可以用phantom查询与其接触的其他碰撞体。查询的结果数据格式，实质上与零距离形状投射的相同。

然而，phantom与形状投射的区别是，phantom会持续在碰撞世界里存在。这样，检测phantom与“真实”碰撞体时，就能充分利用碰撞引擎的时间一致性优化。事实上，phantom和正常碰撞体的唯一区别是，phantom对于碰撞世界中其他碰撞体来说是“透明”的（phantom也不参与动力学模拟）。那么就可以回答phantom与哪些真实碰撞体碰撞的假设性问题，又确保phantom完全不会影响到碰撞世界的其他碰撞体（包括phantom）。

12.3.7.4 其他查询类型

除了投射，有些碰撞引擎还支持其他种类的查询。例如，Havok提供最近点（closest point）查询[48]，给定一个碰撞体，就能找出其他接近的碰撞体上的最近点集合。

12.3.8 碰撞过滤

游戏开发者经常需要启用或禁用对某类型物体的碰撞检测。例如，多数物体可以穿过水体的表面。我们可加入浮力模拟（buoyancy simulation）令那些物体上浮或下沉，但无论是哪一种情况，我们都不希望水面像固体一样。多数碰撞引擎可以根据游戏的具体准则，来决定碰撞体之间的接触是否成立。这就是**碰撞过滤**（collision filtering）功能。

12.3.8.1 碰撞掩码及碰撞层

常见的碰撞过滤方法之一，就是对世界中的物体进行分类，然后用一个查找表判断某类

[47]译注：这仅是Havok引擎的术语，由于无普遍性，不硬译作“幻影”。

[48]译注：更一般性的术语为邻近查询（proximity query）。

碰撞体能否与另一些分类碰撞。例如，在Havok中，碰撞体可以属于（唯一）一个碰撞层。Havok的默认碰撞过滤器是hkpGroupFilter的实例，它对每个碰撞层维护一个32位的掩码，当中每个位指定某碰撞层是否能与另一碰撞层碰撞。

12.3.8.2 碰撞回调

另一种过滤技术是，当碰撞库检测到碰撞时调用**回调函数**（callback function）。回调函数可以检查碰撞的具体信息，然后按自己所定的条件决定接受或拒绝碰撞。Havok也支持这种过滤。Havok中，当接触点一开始加进世界时，就会调用contactPointAdded()回调函数。若后来判断接触点是合法的（若找到更早碰撞时间的接触，此接触点可能会获判失效），就会调用contactPointConfirmed()回调函数。若有需要，应用程序可以在这些回调中拒绝这些接触点。[49]

12.3.8.3 游戏专门的碰撞材质

游戏开发者通常需要对游戏世界中的物体进行分类，除了用来控制它们如何碰撞（如使用碰撞过滤），也可以控制其他效果，如某类物体撞到另一类物体时所产生的声音或粒子效果。例如，我们可能希望区分木、石、金属、泥、水及血肉之躯的效果。

要达到此目的，许多游戏实现了一种碰撞形状分类机制，它在多方面与图形的**材质系统**相似。事实上，有些游戏团队使用**碰撞材质**（collision material）这个术语形容这种分类机制。其基本思路是把每个碰撞表面关联至一组属性，这组属性定义了某种表面在物理上和碰撞上的行为。碰撞属性可包含音效、粒子效果、物理属性（如恢复系数和摩擦系数[50]）、碰撞过滤信息，以及其他游戏所需的信息。

简单凸碰撞原型通常只会使用一组碰撞属性。而多边形汤形状可能需要在每个三角形上设置属性。在后者的使用方式中，我们通常希望碰撞原型及其碰撞属性的绑定能尽量紧凑。典型的做法是碰撞原型以8、16或32位整数指定碰撞材质。此整数用于索引至某数据结构数组，内含详细的碰撞属性。

---

[49]译注：这种方法提高了应用程序的自由度，但效率会较低。因为以分类判断的话，可以在粗略阶段判断是否继续较细致的相交计算。碰撞引擎可以在粗略阶段提供回调函数，能尽早剔除一些不需要的运算。

[50]译注：恢复系数和摩擦系数分别在12.4.7.2及12.4.7.5节中介绍。

12.4 刚体动力学

许多游戏引擎都包含**物理系统**（physics system），以模拟虚拟游戏世界中的物体，使其运动接近现实世界的方式。技术上来说，游戏物理引擎通常关注物理学的其中一门——**力学**（mechanics）。力学研究力（force）怎样影响物体的行为。在游戏引擎中，我们通常特别关注物体的**动力学**（dynamics），即它们如何随时间移动[51]。一直以来，游戏中几乎只关注动力学中的**经典刚体动力学**（classical rigid body dynamics）。此术语意味着在游戏物理模拟中，做了两个重要的简化假设。

- **经典（牛顿）力学**：模拟中假设物体服从牛顿运动定律（Newton's laws of motion）。物体足够大，不会产生量子效应（quantum effect）；物体的速度足够低，不会产生相对论性效应（relativistic effect）。
- **刚体**：模拟中的物体是完美的固体，不会变形。换言之，其形状是恒常固定的。这种假设能良好配合碰撞检测系统。而且刚性能大幅简化模拟固体动力学所需的数学。

游戏物理引擎也能确保游戏世界中的刚体运动符合多个**约束**（constraint）。最常见的约束是非穿透性（non-penetration），即物体不能互相穿透。因此，当发现物体互相穿透时，物理系统要尝试提供真实是**碰撞响应**（collision response）。这是物理引擎与碰撞系统紧密关联的主因之一。

多数物理系统也容许游戏开发者设置其他类型的约束，以定义物理模拟刚体之间的真实互动。这些约束包括有铰链（hinge）、棱柱关节（prismatic joint，滑动块/slider）、球关节（ball joint）、轮、“布娃娃”，用于模拟失去知觉或死亡的角色，诸如此类。

物理系统通常会共享碰撞世界的数据结构，而且事实上物理引擎通常会驱动碰撞检测算法的执行，作为物理时步更新的一个环节。动力学模拟的刚体和碰撞引擎的碰撞体通常是一对一的映射关系。例如，在Havok中，`hkpRigidBody`物体引用至（唯一）一个`hkpCollidable`（虽然也可以创建无对应刚体的碰撞体）。在PhysX中，这两个概念更紧密地整合在一起，一个`NxActor`同时用作碰撞体及动力学模拟的刚体。这些刚体及其对应的碰撞体通常会由一个单例[52]数据结构管理，例如称作**碰撞/物理世界**（collision/physics world），或简单地称为**物理世界**（physics world）。

物理引擎中的刚体通常独立于游戏性虚拟世界中的逻辑对象。游戏对象的位置和定向可以由物理模拟驱动。要达此目的，我们每帧向物理引擎查询刚体的变换[53]，然后把该变换

---

[51] 译注：与动力学相对的是静力学（statics），研究物体在静止平衡状态下的负载，并不考虑时间上的变化。不过游戏中很少会应用到静力学。

[52] 译注：物理引擎通常可创建多个互不相关的物理世界，所以这里并不是严格意义上的单例。

[53] 译注：由于是刚体模拟，这里的变换是刚性变换，即只含有位移和旋转，不含缩放及切变。

以某方式施于对应游戏对象的变换。游戏对象的动作也可以由其他引擎系统（如动画系统或角色控制系统）所决定，再去驱动物理世界中刚体的位置及定向。12.3.1节提及，一个逻辑游戏对象可由物理世界中的一个或多个刚体所表示。简单的对象（如石头、武器、木桶）可对应至单个刚体。但是，含关节的角色或复杂的机器可能由多个互相连接的刚体所构成。

本章以下内容致力研究游戏物理引擎如何运作。我们简介刚体动力学模拟的理论后，便会研究游戏物理系统中一些最常见的功能，并会察看物理引擎如何整合至游戏中。

12.4.1 基础

关于经典刚体动力学这一主题，有许多非常优秀的书籍、文章、简报。[15]提供解析力学的坚实基础。[34]、[11]及[25]等是更贴近我们讨论的文献，内容针对游戏所用的物理模拟。[1]、[9]、[28]等包含游戏的刚体动力学章节。Chris Hecker在《游戏开发者杂志（Game Developer Magazine）》上撰写了一系列关于游戏物理的文章。他把这些文章和相关的资源发布于其个人网站[54]。Russell Smith（ODE的主要开发者）也提供了一个关于游戏动力学模拟的简报[55]。

在本节中，笔者将概述多数游戏物理引擎背后的基础理论概念。这只是一个旋风之旅，笔者必须略去一些细节。当读者阅读本章时，强烈建议读者至少读一些上面提到的资源。

12.4.1.1 单位

多数刚体动力学模拟都会使用MKS单位系统。此系统中，距离以米（meter/m）量度，质量以千克（kilogram/kg）量度，时间以秒（second/s）量度。MKS系统因这些单位缩写而得名。

读者可以设置物理系统使用其他单位。但若是如此，必须确保模拟中所有量纲都保持一致。例如，因引力而产生的加速度常量**g**，在MKS系统中使用m/s^2量度，若读者选择其他单位系统就要适当地转换。多数游戏团队都会坚持使用MKS，使生活能过得轻松一点。

12.4.1.2 分离线性及旋转动力学

无约束刚体（unconstrained rigid body），指可以在3个笛卡儿轴上自由位移，并绕这3个轴自由旋转的刚体。我们称这种刚体含6个**自由度**（degree of freedom，DOF）。

[54] http://chrishecker.com/Rigid_Body_Dynamics

[55] http://www.ode.org/slides/parc/dynamics.pdf

这有可能是有点出乎意料，但无约束刚体的运动可以分离为两个独立的部分。

- **线性动力学**（linear dynamics）描述刚体除旋转以外的运动。（我们可单单使用线性动力学描述理想化**质点**/point mass的运动。所谓质点是指无穷小、无法旋转的物质。）
- **旋转动力学**（angular dynamics）描述刚体的旋转性运动。

读者可以很容易想象，能够分离线性及旋转部分，对于分析或模拟刚体的运动是有万分帮助的。这意味着我们无须顾及一个刚体的旋转运动，就能计算其线性运动，这有如把刚体视作理想化质点。然后，再叠加上旋转运动就可以完整地描述刚体运动。

12.4.1.3 质心

以**线性**动力学的需求来说，无约束刚体的行为有如把所有其物质集中于一个点，此点称为**质心**（center of mass，CM/COM）。质心本质上是刚体在所有定向的平衡点。换句话说，刚体的质量在所有方向上均匀分布地围绕质心。

在均匀密度的刚体中，其质心位于刚体的**几何中心**（centroid）。那即是说，如果把刚体切割成N个非常小的等分，再把它们的位置之和除以N后就会逼近质心位置。若刚体的密度并非均匀，那么就要令每个小块以其质量作为权值，也即一般化来说质心是这些小块位置的加权平均值。因此：

$$\mathbf{r}_{\mathrm{CM}} = \frac{\sum_{\forall i} m_i \mathbf{r}_i}{\sum_{\forall i} m_i} = \frac{\sum_{\forall i} m_i \mathbf{r}_i}{m}$$

当中，$\mathbf{r}$表示矢径（radius vector）或位置矢量（position vector），即从世界空间原点至该点的矢量。（当这些小块的大小和质量趋向0，此和便变成积分。）

对于凸的刚体，其质心总是位于刚体之内。凹的刚体的质心可能位于刚体之外。（例如，字母“C”的质心位于哪里呢？）

12.4.2 线性动力学

以线性动力学的需求来说，刚体的位置可以完全由一个位置矢量$\mathbf{r}_{\mathrm{CM}}$描述，如图12.22所示，该矢量由世界空间原点延伸至刚体质心。由于我们使用MKS系统，位置以米量度。因为正在描述刚体质心的运动，以下讨论会略去CM下标。

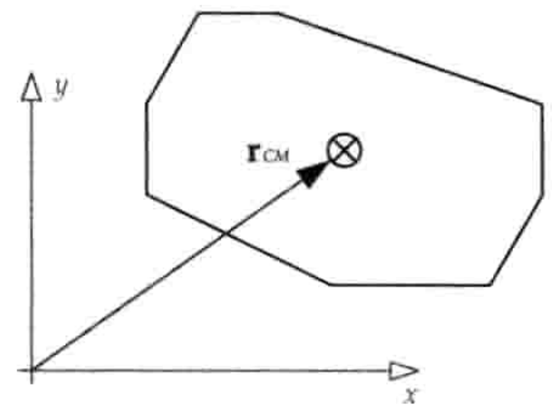


图 12.22: 以线性动力学的需求来说，刚体的位置可以完全由其质心位置所描述。

12.4.2.1 线性速度和加速度

刚体的**线性速度**（linear velocity）定义了刚体质心的移动速率和方向。线性速度是矢量，通常以米每秒（m/s）量度。速度[56]是位置对于时间的第一导数，因此可以写成：

$$\mathbf{v}(t) = \frac{\mathrm{d}\mathbf{r}(t)}{\mathrm{d}t} = \dot{\mathbf{r}}(t)$$

当中矢量$\mathbf{r}$上面的点代表对于时间的导数[57]。对矢量微分等于对其分量独立地微分，因此：

$$v_x(t) = \frac{\mathrm{d}r_x(t)}{\mathrm{d}t} = \dot{r}_x(t)$$

y和z分量也是如此。

线性加速度（linear acceleration）是线性速度对于时间的第一导数，又等于刚体质心位置对于时间的第二导数。加速度也是矢量，通常使用符号$\mathbf{a}$表示，可写成：

$$\begin{aligned}\mathbf{a}(t) &= \frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t} = \dot{\mathbf{v}}(t) \\ &= \frac{\mathrm{d}^2\mathbf{r}(t)}{\mathrm{d}t^2} = \ddot{\mathbf{r}}(t)\end{aligned}$$

12.4.2.2 力及动量

力（force）定义为任何能使含质量物体加速或减速的东西。力含模（magnitude）及空间中的方向，因此所有力都以矢量表示。力通常标记为符号$\mathbf{F}$。当N个力施于一个刚体时，

[56]译注：严格地说，应称为瞬时速度（instantaneous velocity）。

[57]译注：此乃牛顿标记法（Newton's notation），常见于力学中。而$\mathrm{d}x/\mathrm{d}t$是莱布尼茨标记法（Leibniz's notation）。

其对刚体线性运动的净效应为那些力的矢量和：

$$\mathbf{F}_{\text{net}} = \sum_{i=1}^{N} \mathbf{F}_i$$

著名的牛顿力学第二定律指出，力与加速度和质量成正比：

$$\mathbf{F}(t) = m\mathbf{a}(t) = m\ddot{\mathbf{r}}(t) \tag{12.2}$$

此定律意味着，力的量度单位是千克米每平方秒（kg-m/s^2），又称为牛顿（Newton）。

当把刚体的线性速度和质量相乘，就会得出**线性动量**（linear momentum）。线性动量习惯性以$\mathbf{p}$表示[58]：

$$\mathbf{p}(t) = m\mathbf{v}(t)$$

当质量是常数，方程(12.2)是正确的。但若质量不是常数，例如，火箭的燃料会逐渐转变为能量，方程(12.2)就不完全正确。下面的公式才是正确的：

$$\mathbf{F}(t) = \frac{\mathrm{d}\mathbf{p}(t)}{\mathrm{d}t} = \frac{\mathrm{d}(m\mathbf{v}(t))}{\mathrm{d}t}$$

当质量是常数时，m就可以抽出微分之外，变成我们所熟悉的$\mathbf{F} = m\mathbf{a}$。[59]我们并不怎么需要关注线性动量。然而，在旋转动力学中，动量的概念就变得更相关。

12.4.3 运动方程求解

刚体动力学的中心问题是，给定一组施于刚体的已知力，对刚体的运动求解。对于线性动力学而言，这是指给定合力$\mathbf{F}_{\text{net}}(t)$及其他信息（如之前某刻的位置及速度），求出$\mathbf{v}(t)$及$\mathbf{r}(t)$。以下会见到，这等于对两个常微分方程求解，一个是给定$\mathbf{a}(t)$求$\mathbf{v}(t)$，另一个是给定$\mathbf{r}(t)$求$\mathbf{v}(t)$。

---

[58]译注：在牛顿引入动量之前，科学界有一个和动量相似、称为impetus（词源来自拉丁文in + petere）的概念。可能是因此后人才使用p来代表动量，而这样也避免了使用m和质量冲突。

[59]译注：如果质量是t的函数，那么$\mathbf{F}(t) = \dfrac{\mathrm{d}(m(t)\mathbf{v}(t))}{\mathrm{d}t} = \mathbf{v}(t)\dfrac{\mathrm{d}m(t)}{\mathrm{d}t} + m(t)\dfrac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t}$。

12.4.3.1 把力作为函数

力可以是常数，又可以是随时间变化的函数，如上节所示的$\mathbf{F}(t)$。力也可以是刚体位置、速度或其他多个变量的函数。因此，广义上力可以表达为：

$$\mathbf{F}(t, \mathbf{r}(t), \mathbf{v}(t), \cdots) = m\mathbf{a}(t) \tag{12.3}$$

这又可以写成按位置矢量及其第一、第二导数表示的函数：

$$\mathbf{F}(t, \mathbf{r}(t), \dot{\mathbf{r}}(t), \cdots) = m\ddot{\mathbf{r}}(t)$$

例如，弹簧所产生的力，与其从自然静止位置拉伸至的距离成正比。在一维中，若弹簧的静止位置是$x = 0$，弹簧的力可以写成：

$$F(t, x(t)) = -kx(t)$$

当k是**弹簧常数**（spring constant），用于量度弹簧的刚度（stiffness）。

举另一个例子，机械黏滞阻尼器（mechnical viscous damper，也称为减震器/dashpot）所产生的力，与阻尼器的活塞速度成正比。因此在一维中，这个力可以写成：

$$F(t, v(t)) = -bv(t)$$

当中b是**黏滞阻尼系数**（viscous damping coefficient）。

12.4.3.2 常微分方程

广义来说，**常微分方程**（ordinary differential equation，ODE[60]）是涉及一个自变量（independent variable）的函数及多个该函数导数的方程。若自变量是时间，而该函数是$x(t)$，那么一个ODE是以下形式的关系：

$$\frac{\mathrm{d}^n x}{\mathrm{d}t^n} = f\left(t, x(t), \frac{\mathrm{d}x(t)}{\mathrm{d}t}, \frac{\mathrm{d}^2 x(t)}{\mathrm{d}t^2}, \cdots, \frac{\mathrm{d}^{n-1} x(t)}{\mathrm{d}t^{n-1}}\right)$$

另一种说法是，$x(t)$的第n导数表示为函数f，而函数f的参数为时间（t）、位置（$x(t)$），以及任意数量低于n阶的$x(t)$导数。

[60]译注：请勿与开放物理引擎（open dynamics engine，ODE）混淆。

如方程(12.3)所示，力在广义上是时间、位置及速度的函数：

$$\ddot{\mathbf{r}}(t) = \frac{1}{m}\mathbf{F}(t, \mathbf{r}(t), \dot{\mathbf{r}}(t))$$

这显然符合常微分方程的资格。我们希望能对这个常微分方程求解，以得出$\mathbf{v}(t)$及$\mathbf{r}(t)$。

12.4.3.3 解析解

在很罕见的情况下，运动的微分方程可以求出**解析解**（analytical solution）。即是说，可以找到一个简单的闭合式函数，描述所有可能的时间值t的刚体位置。一个常见的例子是抛射物（projectile）受引力加速度影响的垂直运动，当中加速度为$\mathbf{a}(t) = [0, g, 0]$，而$g = -9.8\text{m/s}^2$。在这个例子中，常微分方程可归结为：

$$\ddot{y}(t) = g$$

对此求积分，得出：

$$\dot{y}(t) = gt + v_0$$

当中v_0是抛射物的初始垂直速度。再求第二次积分就会得出熟悉的解：

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0$$

当中y_0为抛射物的初始垂直位置。

然而，在游戏中几乎永不可能有解析解。此结论部分原因是由于一些微分方程根本无闭合式解。此外，游戏是一个互动模拟，我们不可能预知游戏中的力如何随时间变化。因此，我们不可能求得简单的闭合式、随时间变化的函数，表示游戏中物体的位置及速度。

12.4.4 数值积分

由于上述原因，游戏物理引擎改为使用**数值积分**（numerical integration）技术。使用这种技术，我们能对微分方程以**时步**（time step）的方式求解，即以上一时间步的解求得本时间步的解。时间步的长度通常（大约）是常数，并标记为Δt。给定已知某刚体在t_1时间的位置及速度，并且力是时间、位置及/或速度的函数，希望求得在下一时间步$t_2 = t_1 + \Delta t$的位置及速度。换句话说，给定$\mathbf{r}(t_1)$、$\mathbf{v}(t_1)$及$\mathbf{F_1}(t, \mathbf{r}, \mathbf{v})$，求出$\mathbf{r}(t_2)$及$\mathbf{v}(t_2)$。

12.4.4.1 显式欧拉

对常微分方程的最简单数值求解方法之一是**显式欧拉法**（explicit Euler method）。这是游戏程序新手最常使用的直觉方法。假设在某一时刻我们得悉当前的速度，而我们希望对以下的常微分方程求解，以得出次帧的刚体位置：

$$\mathbf{v}(t) = \dot{\mathbf{r}}(t) \tag{12.4}$$

使用显式欧拉法，我们只需简单地把速度乘以时间步，即从米每秒的单位转换为米每帧，然后把1帧所移动的距离加至当前位置，以求出物体在次帧的新位置。这就能得出以下对该常微分方程(12.4)的近似解：

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t \tag{12.5}$$

我们可以用类似方式，给定施于本帧的合力，求物体次帧的速度。此常微分方程为：

$$\mathbf{a}(t) = \frac{\mathbf{F}_{\text{net}}(t)}{m} = \dot{\mathbf{v}}(t)$$

而使用显式欧拉法对此常微分方程的近似解为：

$$\mathbf{v}(t_2) = \mathbf{v}(t_1) + \frac{\mathbf{F}_{\text{net}}(t_1)}{m}\Delta t \tag{12.6}$$

显式欧拉的诠释

方程(12.5)所做之事，实际上是假设在该时间步中，物体的速度维持不变。因此，我们可使用**当前**的速度去预计**次**帧的物体位置。位置在时间t_1及t_2之间的改变量$\Delta\mathbf{r} = \mathbf{v}(t_1)\Delta t$。若我们绘制刚体位置对时间的图，我们实际上是使用$t_1$时函数的**斜率**（就是$\mathbf{v}(t)$），线性外插出下一时间步$t_2$时该函数的值。如图12.23所示，线性外插并不一定可以好好预测下一时间步的真正位置$\mathbf{r}(t_2)$，但如果速度大约是常数，这种方法可得出不错的结果。

图12.23也暗示了显式欧拉的另一种诠释——导数的逼近。根据定义，任何导数都是两个无穷小的差的商数（在我们的例子中是$\mathrm{d}\mathbf{r}/\mathrm{d}t$）。显式欧拉法使用两个**有限**差的商数逼近这个值。换句话说，$\mathrm{d}\mathbf{r}$变成$\Delta\mathbf{r}$，$\mathrm{d}t$变成Δt就得出：

$$\frac{\mathrm{d}\mathbf{r}}{\mathrm{d}t} \approx \frac{\Delta\mathbf{r}}{\Delta t},$$
$$\mathbf{v}(t_1) = \frac{\mathbf{r}(t_2) - \mathbf{r}(t_1)}{t_2 - t_1} = \frac{\mathbf{r}(t_2) - \mathbf{r}(t_1)}{\Delta t}$$

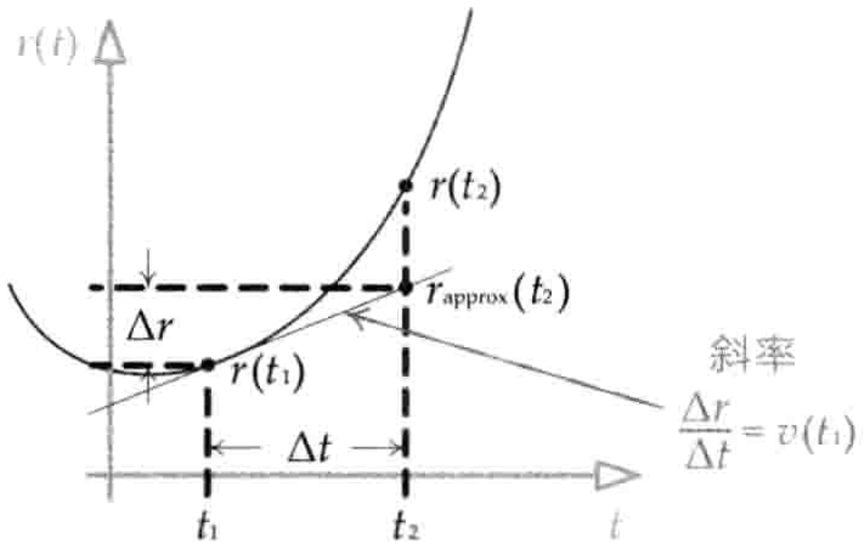


图 12.23: 在显式欧拉法中，使用t_1时函数$\mathbf{r}(\mathbf{t})$斜率，从$\mathbf{r}(t_1)$线性外插出$\mathbf{r}(t_2)$的估值。

此方程能再次简化为方程(12.5)。仅当速度在时间步中维持不变，此逼近法才是合法的。若Δt趋近0，其极限也是合法的（那么会变成**完全**准确）。显然，相同的分析方法也可以用在方程(12.6)。

12.4.4.2 数值方法的特性

笔者已经指出，显式欧拉法并不太准确。我们再具体地研究此问题。常微分方程的数值解实际上有3个重要且互相相连的特性。

- **收敛性**（convergence）：当时间步Δt趋向0，近似解是否逐渐接近真实解？
- **阶数**（order）：给定常微分方程的某个数值逼近解，误差有多“差”？常微分方程数值解的误差，通常与时间步长Δt的某个幂成正比，因此通常会把这些误差写成大O标记法（例如$O(\Delta t^2)$）。当某数值方法的误差为$O(\Delta t^{(n+1)})$，我们称它为n阶的数值方法。
- **稳定性**（stability）：数值解是否趋时间“安顿下来”？若某数值方法在系统中加进能量，物体的速度会最终“爆炸”，系统变得**不稳定**。相反，若某数值方法趋向从系统中消去能量，那么会形成全局阻尼的效果，系统会是**稳定的**。

阶数的概念需要多一点解释。通常在量度数值方法的误差时，会比较其逼近方程与常微分方程精确解的泰勒级数（Taylor series）展开式。然后通过相减这两个方程消除通项。余下的泰勒项表示该数值方法固有的误差。例如，显式欧拉方程为：

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t$$

而精确解的泰勒级数展开式为：

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + \frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \frac{1}{6}\dddot{\mathbf{r}}(t_1)\Delta t^3 + \cdots$$

因此，欧拉方法的误差可表示为$\mathbf{v}\Delta t$之后的所有项，而这些项的阶数为$O(\Delta t^2)$（因为余下更高阶的项比这项的影响少）：

$$\begin{aligned}\mathbf{E} &= \frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2 + \frac{1}{6}\dddot{\mathbf{r}}(t_1)\Delta t^3 + \cdots \\ &= O(\Delta t^2)\end{aligned}$$

为了显示某数值方法的误差，我们可以在写下其方程时在末端加入以大O标记法表示的误差项。例如，显式欧拉法的方程能最准确地写成：

$$\mathbf{r}(t_2) = \mathbf{r}(t_1) + \dot{\mathbf{r}}(t_1)\Delta t + O(\Delta t^2)$$

我们称显式欧拉法是“一阶”方法，因为其准确度达到**并包括**Δt一次方的泰勒级数。概括而论，若某数值方法的误差项是$O(\Delta t^{(n+1)})$，它便称为n阶方法。

12.4.4.3 显式欧拉以外的选择

显式欧拉法常出现于游戏中的简单积分工作，当速度接近常数时它能产生最好的结果。然而，通用的动力学模拟并不会使用显式欧拉法，因为此法误差又高、稳定性又低。对常微分方程求解还有许多方法，包括向后欧拉（backward Euler，另一种一阶方法）、中点欧拉（mid-point Euler，二阶方法），以及Runge-Kutta方法族。（四阶Runge-Kutta是尤其流行的一员，其缩写为RK4。）在此笔者不详细描述这些方法，因为读者可以在线上和文献中找到大量相关信息。维基百科[61]可以作为学习这些方法的优良跳板。

12.4.4.4 韦尔莱积分

现在，游戏最常用的常微分方程数值方法大概是韦尔莱积分（Verlet integration）[62]，因此笔者会为它花上一些篇幅。此方法实际上有两个变种：正常韦尔莱及**速度韦尔莱**（velocity Verlet）。笔者会简介这两个变种，但其理论及深入的解释则留给相关的大量文献及网页。（读者可阅读维基百科[63]作为起点。）

正常韦尔莱积分非常吸引人，因为它能达致高阶（少误差），相对简单，求值又快，而且能直接使用加速度在单个步骤中求出位置（而不是一般做法中先用加速度求速度，再用速度求位置）。推导韦尔莱积分公式的方法是把两个泰勒级数求和，一个是往后的时间，一个

[61] http://en.wikipedia.org/wiki/Numerical_ordinary_differential_equations

[62] 译注：韦尔莱积分以其发明者、法国物理学家Loup Verlet命名。此姓氏的法语读音为[vɛʁ'lɛ]，注意它无尾音t。

[63] http://en.wikipedia.org/wiki/Verlet_integration

是往前的时间：

$$\mathbf{r}(t_1+\Delta t)=\mathbf{r}(t_1)+\dot{\mathbf{r}}\Delta t+\frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2+\frac{1}{6}\dddot{\mathbf{r}}(t_1)\Delta t^3+O(\Delta t^4)$$
$$\mathbf{r}(t_1-\Delta t)=\mathbf{r}(t_1)-\dot{\mathbf{r}}\Delta t+\frac{1}{2}\ddot{\mathbf{r}}(t_1)\Delta t^2-\frac{1}{6}\dddot{\mathbf{r}}(t_1)\Delta t^3+O(\Delta t^4)$$

把这两个算式相加，便可以消去一些正负相反的项。其结果是，以加速度和当前及上一个时间步的（已知）位置表示下一个时间步的位置。这就是正常韦尔莱积分[64]：

$$\mathbf{r}(t_1+\Delta t)=2\mathbf{r}(t_1)-\mathbf{r}(t_1-\Delta t)+\mathbf{a}(t_1)\Delta t^2+O(\Delta t^4)$$

若使用合力表示，韦尔莱积分可写成：

$$\mathbf{r}(t_1+\Delta t)=2\mathbf{r}(t_1)-\mathbf{r}(t_1-\Delta t)+\frac{\mathbf{F}_{\text{net}}(t_1)}{m}\Delta t^2+O(\Delta t^4)$$

从这个公式消失的速度引人注意。然而，速度可用以下（从多个选择里挑选的）不太准确的方法逼近：

$$\mathbf{v}(t_1+\Delta t)=\frac{\mathbf{r}(t_1+\Delta t)-\mathbf{r}(t_1)}{\Delta t}+O(\Delta t)$$

12.4.4.5 速度韦尔莱

更常用的**速度韦尔莱**是一个含4个步骤的过程，当中时间步切割为两部分去求解。给定已知的$\mathbf{a}(t_1)=m^{-1}\mathbf{F}(t_1,\mathbf{r}(t_1),\mathbf{v}(t_1))$，我们这样执行速度韦尔莱。

1. 计算$\mathbf{r}(t_1+\Delta t)=\mathbf{r}(t_1)+\mathbf{v}(t_1)\Delta t+\frac{1}{2}\mathbf{a}(t_1)\Delta t^2$
2. 计算$\mathbf{v}(t_1+\frac{1}{2}\Delta t)=\mathbf{v}(t_1)+\frac{1}{2}\mathbf{a}(t_1)\Delta t$
3. 确定$\mathbf{a}(t_1+\Delta t)=\mathbf{a}(t_2)=m^{-1}\mathbf{F}(t_2,\mathbf{r}(t_2),\mathbf{v}(t_2))$
4. 计算$\mathbf{v}(t_1+\Delta t)=\mathbf{v}(t_1+\frac{1}{2}\Delta t)+\frac{1}{2}\mathbf{a}(t_1+\Delta t)\Delta t$

注意第3步中的力函数是依赖于下一时间步的位置$\mathbf{r}(t_2)$及速度$\mathbf{v}(t_2)$的。我们已在第1步计算了$\mathbf{r}(t_2)$，那么只要力并不是依赖于速度的，便有所有的数据执行第3步。若力是依赖于速度的，那么便必须先计算次帧的速度近似值，例如使用欧拉方法。

---

[64]译注：译者在开发《爱丽丝疯狂回归（Alice: Madness Returns）》时，也是利用这种韦尔莱积分模拟头发的，这个方法非常稳定及高效，但其中一个缺点是连续两帧的Δt必须接近不变，否则会有明显的异常结果。其详细原理及实现可参考博文http://www.cnblogs.com/miloyip/archive/2011/06/14/alice_madness_returns_hair.html。

12.4.5 二维旋转动力学

直至这里，我们都集中分析刚体质心的线性运动（其行为和点质心一样）。如笔者较早前提及，无约束刚体会绕其质心旋转。这意味着我们可以在质心线性运动上，叠加刚体的旋转运动，从而得出刚体整体运动的完整描述。响应施力的刚体旋转运动的学问，称为**旋转动力学**（angular dynamics）。

在二维中，旋转动力学几乎等同于线性动力学。每个线性量都可对应一个类似的旋转量，而且两种数学计算都非常工整。因此，我们会先探讨二维旋转动力学，然后再扩展至三维。三维比较复杂一点，我们由浅入深。

12.4.5.1 定向及角速率

在二维中，每个刚体可当作一块材料薄片。（有些物理文献称这些刚体为**平面薄片**/plane lamina。）所有线性运动在xy平面上发生，而所有旋转则是绕z轴发生的。（想象一块拼图在气垫曲棍球桌面上滑行。）

二维刚体的定向（orientation）可以完全用一个角度θ表示，此角度相对于某事先订立的零旋转，以孤度量度。例如，我们可以定义，当车辆向着世界空间的正x轴时，设$\theta = 0$。此角度当然是随时间变化的，所以我们把它标记为$\theta(t)$。

12.4.5.2 角速率与加速度

角速度（angular velocity）是量度刚体旋转角度随时间的变化率。在二维中，角速度是标量，更正确来说应该称之为角**速率**，因为“速度”一词只能用于矢量。角速率以标量函数$\omega(t)$标记，而单位是弧度每秒（rad/s）。角速率是定向角度$\theta(t)$对于时间的导数：

$$\text{旋转：}\quad \omega(t) = \frac{\mathrm{d}\theta(t)}{\mathrm{d}t} = \dot{\theta}(t) \qquad \text{线性：}\quad \mathbf{v}(t) = \frac{\mathrm{d}\mathbf{r}(t)}{\mathrm{d}t} = \dot{\mathbf{r}}(t)$$

如我们所料，角加速率（angular acceleration）是角速率的变化率，标记为$\alpha(t)$，量度单位为弧度每平方秒（$\mathrm{rad/s^2}$）：

$$\text{旋转：}\quad \alpha(t) = \frac{\mathrm{d}\omega(t)}{\mathrm{d}t} = \dot{\omega}(t) = \ddot{\theta}(t) \qquad \text{线性：}\quad \mathbf{a}(t) = \frac{\mathrm{d}\mathbf{v}(t)}{\mathrm{d}t} = \dot{\mathbf{v}}(t) = \ddot{\mathbf{r}}(t)$$

12.4.5.3 转动惯量

相当于质量，在旋转动力学中有称为**转动惯量**（moment of inertia）的概念。就如同质量代表改变点质量线性速度的难易程度一样[65]，转动惯量是量度刚体在某轴上改变角速率的难易程度。若刚体的质量集中于旋转轴附近，那么它会较容易绕该轴旋转，而它的转动惯量会比另一个质量远离旋转轴的物体小。

由于现在集中讨论二维旋转动力学，旋转轴总为z轴，而刚体的转动惯量是一个简单标量。转动惯量通常标记为英文字母I。笔者不会在此详细说明转动惯量的计算方法，完整推导可参考[15]。

12.4.5.4 力矩

直至现在，我们假设所有力都是施于刚体的质心的。然而，在一般情况下，力可以施于刚体的任何位置。若某个力的施力线穿过刚体的质心，则该力仅产生之前所述的线性运动。在其他情况下，该力除了产生线性运动，还会同时产生一个称为**力矩**（torque）[66]的旋转力，如图12.24所示。

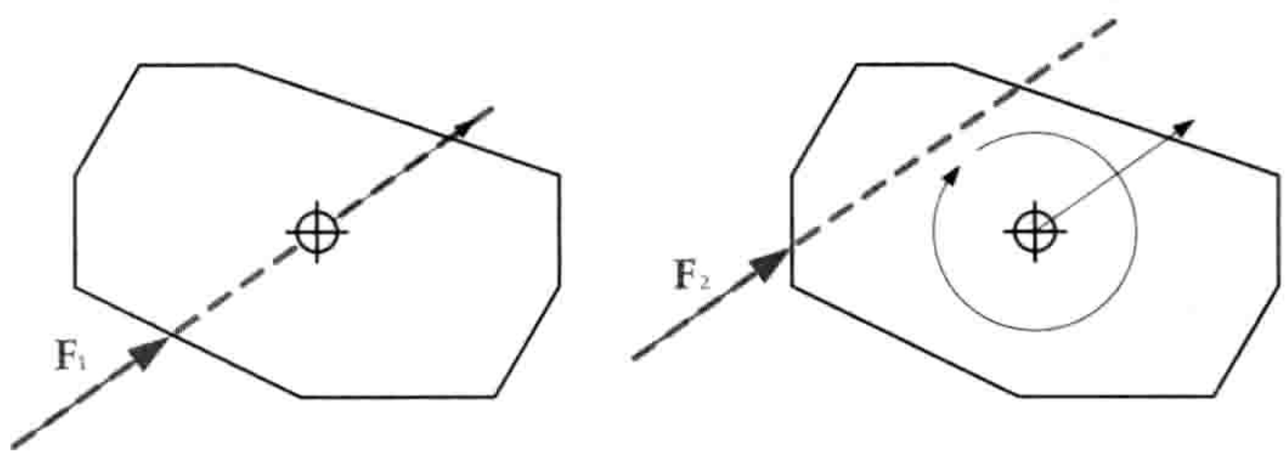


图 12.24: 左图中，对刚体的质心施力会产生纯粹的线性运动。右图中，对质心外施力会形成力矩，同时产生旋转和线性运动。

我们可以使用叉积计算力矩。首先，我们把施力的位置表示为矢量$\mathbf{r}$，此矢量是由质心延伸至施力点，（换句话说，矢量$\mathbf{r}$位于**刚体空间**（body space），该空间的原点位于质心。）如图12.25所示。由力$\mathbf{F}$施于位置$\mathbf{r}$所产生的力矩$\mathbf{N}$为：

$$\mathbf{N} = \mathbf{r} \times \mathbf{F} \tag{12.7}$$

---

[65]译注：质量又可称为惯性质量（inertial mass），意味着它是量度改变惯性（inertia）的阻力。注意在力学中，质量（mass）有别于重量（weight），后者是物体因引力而产生的力，即$W = mg$。由于重量是一种力，其单位是牛顿（N）而不是千克（kg）。

[66]译注：力矩又称为转矩、扭矩、转动力矩。

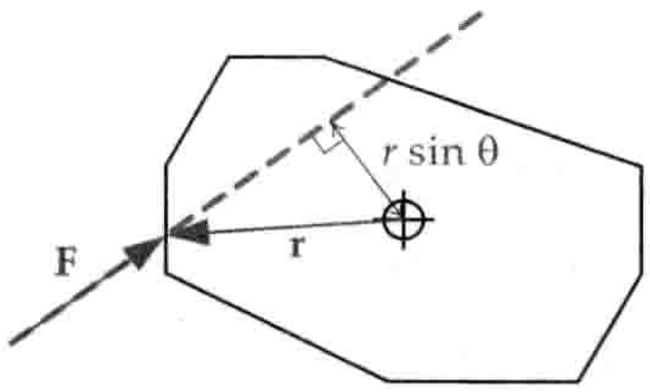


图 12.25: 力矩的计算方法是求刚体空间（即相对于质点）的施加点与力矢量的叉积。为方便作图，图中的矢量都以二维展示，力矩矢量实际上是插进页面的方向。

方程(12.7)意味着，施力点离质心越远，力矩越大。这解释了杠杆为何可以帮助移动重物，这也解释了为何施于质心的力不产生力矩及旋转——因为这种情况下矢量**r**为**0**。

当两个或以上的力施于一个刚体时，可以把各个力矩求和，如同把力求和一样。因此我们最后关心的是净力矩（net torque）$\mathbf{N}_{\text{net}}$。

在二维中，矢量**r**和**F**必须位于xy平面上，因此**N**必然会指向正或负z轴。因此，我们可以把二维力矩标记为N_z，就是矢量**N**的z分量[67]。

力矩对于角加速率和转动惯量，有如力对于线性加速度和质量：

$$\begin{array}{l|l} \text{旋转：} \quad \begin{aligned} N_z &= I\alpha(t) \\ &= I\dot{\omega}(t) = I\ddot{\theta}(t) \end{aligned} & \text{线性：} \quad \begin{aligned} \mathbf{F} &= m\mathbf{a}(t) \\ &= m\dot{\mathbf{v}}(t) = m\ddot{\mathbf{r}}(t) \end{aligned} \end{array} \tag{12.8}$$

12.4.5.5 二维旋转方程求解

在二维的情况下，我们可以使用和线性动力学问题完全相同的数值方法，为旋转运动方程求解。我们希望求解的一对常微分方程如下：

$$\begin{array}{l|l} \text{旋转：} \quad \begin{aligned} N_{\text{net}}(t) &= I\dot{\omega}(t), \\ \omega(t) &= \dot{\theta}(t) \end{aligned} & \text{线性：} \quad \begin{aligned} \mathbf{F}_{\text{net}}(t) &= m\dot{\mathbf{v}}(t), \\ \mathbf{v}(t) &= \dot{\mathbf{r}}(t) \end{aligned} \end{array}$$

而它们的显式欧拉逼近解为：

$$\begin{array}{l|l} \text{旋转：} \quad \begin{aligned} \omega(t_2) &= \omega(t_1) + I^{-1}N_{\text{net}}(t_1)\Delta t, \\ \theta(t_2) &= \theta(t_1) + \omega(t_1)\Delta t \end{aligned} & \text{线性：} \quad \begin{aligned} \mathbf{v}(t_2) &= \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t_1)\Delta t, \\ \mathbf{r}(t_2) &= \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t \end{aligned} \end{array}$$

当然，也可以使用其他更精确的数值方法，例如速度韦尔莱（笔者在此省略了线性情况，读者可对12.4.4.5节中的步骤做比较）。

[67] 译注：$N_z = r_x F_y - r_y F_x$。

1. 计算$\theta(t_1+\Delta t)=\theta(t_1)+\omega(t_1)\Delta t+\frac{1}{2}\alpha(t_1)\Delta t^2$
2. 计算$\omega(t_1+\frac{1}{2}\Delta t)=\omega(t_1)+\frac{1}{2}\alpha(t_1)\Delta t$
3. 确定$\alpha(t_1+\Delta t)=\alpha(t_2)=I^{-1}N_{net}(t_2,\theta(t_2),\omega(t_2))$
4. 计算$\omega(t_1+\Delta t)=\omega(t_1+\frac{1}{2}\Delta t)+\frac{1}{2}\alpha(t_1+\Delta t)\Delta t$

12.4.6 三维旋转动力学

三维的旋转动力学较二维的复杂，然而其基本概念是非常相近的。本节会非常扼要地介绍旋转动力学如何在三维中运作，集中讨论初接触此课题常会困惑的地方。有许多网上文章可以参考，包括Glenn Fiedler关于此课题的系列文章[68]、卡内基梅隆大学机械人研究所David Baraff的一篇文章“An Introduction to Physically Based Modeling（基于物理的建模概论）”[69]。

12.4.6.1 惯性张量

刚体在3个坐标轴上的质量分布可以很不相同。因此，我们可以预料，不同轴的转动惯量会有所不同。例如，一支棍子应该可以较容易地绕其长轴旋转，因为所有质量都很接近此旋转轴；绕其短轴旋转则较困难，因为棍子的质量分布离开旋转轴很远。现实的确是这样的，也说明为何花样滑冰运动员自转时把双手拉近身体旋转速度可以提升[70]。

在三维中，刚体的旋转质量由3×3矩阵表示的，此矩阵称为**惯性张量**（inertia tensor），通常标记为$\mathbf{I}$（如前，我们不在此描述如何计算惯性张量，请参考[15]）：

$$\mathbf{I}=\begin{bmatrix}I_{xx} & I_{xy} & I_{xz}\\ I_{yx} & I_{yy} & I_{yz}\\ I_{zx} & I_{zy} & I_{zz}\end{bmatrix}$$

此矩阵主对角线上的元素（I_{xx}、I_{yy}、I_{zz}）是刚体绕3个主轴的转动惯量。此矩阵的主对角线以外元素称为**惯量积**（product of inertia）。若刚体对3个主轴都是对称的（例如一个长方体），这些惯量积会为0。当惯量积不是0，虽然会倾向产生物理上真实的运动，但这些运动可能有点儿违反直觉，会令普通玩家以为这些运动是“错误”的。因此，在游戏物理引擎中，惯性张量常常会简化为三元素矢量$[I_{xx}\quad I_{yy}\quad I_{zz}]$。

---

[68] http://gafferongames.wordpress.com/game-physics

[69] http://www-2.cs.cmu.edu/~baraff/sigcourse/notesd1.pdf

[70] 译注：这个例子是基于角动量守恒的，角动量不变而转动惯性降低，导致角速度上升。

12.4.6.2 三维中的定向

在二维中，刚体的定向可以用单个角度θ描述，此角度量度绕z轴的旋转（假设运动在xy平面上进行）。在三维中，刚体的定向可以表示为3个欧拉角$[\theta_x \quad \theta_y \quad \theta_z]$，当中每个代表刚体绕3个笛卡儿轴之一旋转的角度。然而，如第4章所提及的，欧拉角有万向节死锁（gimbal lock）的问题，可能难以进行运算。因此，刚体的定向更常使用3×3矩阵$\mathbf{R}$或单位四元数q表示。本章统一使用后者。

试回想，四元数是一个四元素矢量。当中x、y、z分量可理解为一个代表旋转轴的单位矢量$\mathbf{u}$乘以旋转半角的正弦，而w分量则是旋转半角的余弦：

$$\begin{aligned} \mathrm{q} &= \begin{bmatrix} q_x & q_y & q_z & q_w \end{bmatrix} = \begin{bmatrix} \mathbf{q} & q_w \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{u} \sin\frac{\theta}{2} & \cos\frac{\theta}{2} \end{bmatrix} \end{aligned}$$

刚体的定向当然是随时间改变的函数，因此我们应把它写作$q(t)$。

再一次，我们需要选择某任意指定的方向代表零旋转。例如，我们可以说，在默认情况下，每个物体的前方与世界空间正z轴对齐，y向上，x向左。任何“非单位”（non-identity）四元数[71]都会把物体转离这规范的世界定向。可任意选择规范的世界定向，但当然游戏中所有资产都必须统一。

12.4.6.3 三维中的角速度及角动量

在三维中，角速度是一个矢量，标记为$\boldsymbol{\omega}(t)$。角速度矢量可以想象为，以一个单位矢量$\mathbf{u}$定义旋转轴，再乘以旋转平面上刚体的角速度标量$\omega_u = \dot{\theta}_u$。因此：

$$\boldsymbol{\omega}(t) = \boldsymbol{\omega}_u(t)\mathbf{u}(t) = \dot{\boldsymbol{\theta}}_u(t)\mathbf{u}(t)$$

在线性动力学中，若没有力施于刚体，则其线性加速度为0，线性速度维持不变。在二维旋转动力学中，这仍然是正确的：若无力矩施于二维刚体，那么角加速率α是0，而绕z轴的角速率ω维持不变。

可惜，在三维旋转动力学中，并**不是**如此的。事实证明，就算一个刚体在没有外力的情况下旋转，其角速度矢量$\boldsymbol{\omega}(t)$可能并不是常量，因为该旋转轴可能会不断改变方向。读者可以做一个简单实验看到这个情况。把一个长方物体（如一块长方形的木头）抛到空中。若抛

[71] 译注：这里的单位（identity）并非指长度为1，而是指[0 0 0 1]，因为用此四元数旋转矢量时，结果矢量会维持不变。

起物体时，以它的最短轴旋转，那么旋转轴的方向会大致保持不变。若尝试以它的长轴旋转，情况也会一样。然而，若以其中间长度的轴来旋转，旋转就会变得完全不稳定。旋转轴会疯狂地改变方向，如图12.26所示。

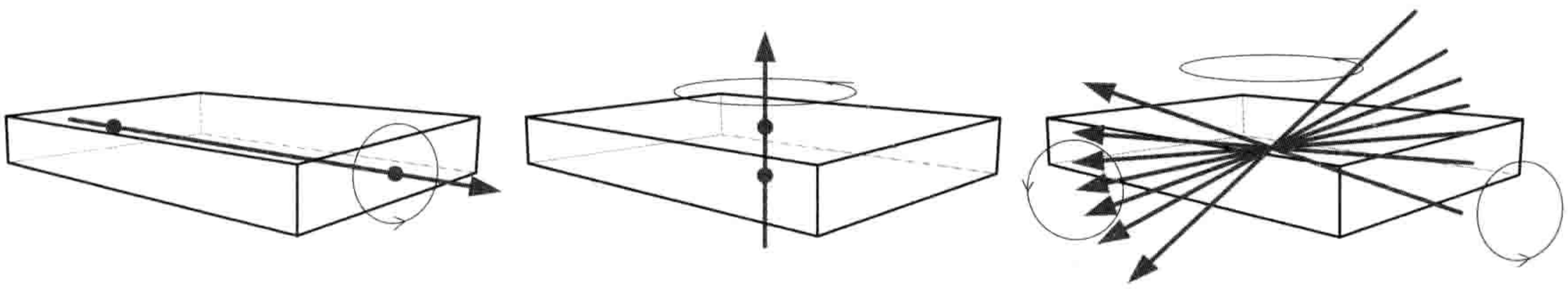


图 12.26: 长方物体以其最短或最长轴旋转时，能产生均角速度矢量。然而，若以其中间长度的轴旋转，其角速度矢量就会疯狂地改变。

在无力矩情况下角速度矢量能改变的事实，说明角速度并不是**守恒**的。然而，另一个相关称为**角动量**（angular momentum）的量，在无力矩下仍维持不变，即角动量是守恒的。旋转动力学的角动量如同线性动力学的线性动量：

$$\text{旋转：} \quad \mathbf{L}(t) = \mathbf{I}\boldsymbol{\omega}(t) \quad \Big| \quad \text{线性：} \quad \mathbf{p}(t) = m\mathbf{v}(t)$$

犹如线性的情况，角动量$\mathbf{L}(t)$是一个三维矢量。然而，与线性动量不一样，旋转质量（惯性张量）并不是标量，而是3×3矩阵。因此，$\mathbf{I}\omega$是一个矩阵积：

$$\begin{bmatrix} L_x(t) \\ L_y(t) \\ L_z(t) \end{bmatrix} = \begin{bmatrix} I_{xx} & I_{xy} & I_{xz} \\ I_{yx} & I_{yy} & I_{yz} \\ I_{zx} & I_{zy} & I_{zz} \end{bmatrix} \begin{bmatrix} \omega_x(t) \\ \omega_y(t) \\ \omega_z(t) \end{bmatrix}$$

由于角速度$\boldsymbol{\omega}$并不守恒，我们在动力学模拟中不会像线性速度$\mathbf{v}$那般，视角速度$\boldsymbol{\omega}$为一个基本的量。角速度是第二级别的量，会在每个模拟时间步中确定了角动量$\mathbf{L}$之后，才计算出$\boldsymbol{\omega}$。

12.4.6.4 三维力矩

计算三维力矩的方法，仍可计算施力点的半径位置矢量与力矢量的叉积（$\mathbf{N} = \mathbf{r} \times \mathbf{F}$）。方程(12.8)仍然有效，但我们需要用角动量表示此方程，因为角速度并不是守恒的量：

$$\mathbf{N} = \mathbf{I}\alpha(t) = \mathbf{I}\frac{\mathrm{d}\boldsymbol{\omega}(t)}{\mathrm{d}t} = \frac{\mathrm{d}}{\mathrm{d}t}(\mathbf{I}\boldsymbol{\omega}(t)) = \frac{\mathrm{d}\mathbf{L}(t)}{\mathrm{d}t}$$

12.4.6.5 三维旋转运动方程求解

当要对三维中的旋转运动求解时，我们可能会想使用与线性运动及二维旋转运动完全相同的方式进行。我们可能猜想，运动的微分方程应该写成：

$$\text{三维旋转（?）：}\quad \begin{aligned}\mathbf{N}_{\text{net}} &= \mathbf{I}\dot{\boldsymbol{\omega}}(t),\\ \boldsymbol{\omega}(t) &= \dot{\boldsymbol{\theta}}(t)\end{aligned} \quad\Big|\quad \text{线性：}\quad \begin{aligned}\mathbf{F}_{\text{net}} &= m\dot{\mathbf{v}}(t),\\ \mathbf{v}(t) &= \dot{\mathbf{r}}(t)\end{aligned}$$

并且，若使用显式欧拉法，我们猜想这些常微分方程的近似解会如此这般：

$$\text{三维旋转（?）}\quad \begin{aligned}\boldsymbol{\omega}(t_2) &= \boldsymbol{\omega}(t_1) + \mathbf{I}^{-1}\mathbf{N}_{\text{net}}(t_1)\Delta t,\\ \boldsymbol{\theta}(t_2) &= \boldsymbol{\theta}(t_1) + \boldsymbol{\omega}(t_1)\Delta t\end{aligned} \quad\Big|\quad \text{线性}\quad \begin{aligned}\mathbf{v}(t_2) &= \mathbf{v}(t_1) + m^{-1}\mathbf{F}_{\text{net}}(t_1)\Delta t,\\ \mathbf{r}(t_2) &= \mathbf{r}(t_1) + \mathbf{v}(t_1)\Delta t\end{aligned}$$

然而，实际上这**并不**正确。三维旋转的微分方程有别于线性及二维旋转的微分方程，两个重要分别如下。

1. 我们直接对$\mathbf{L}$求解，而不是对角速度$\boldsymbol{\omega}$求解。然后才使用$\mathbf{I}$和$\mathbf{L}$计算角速度。这是因为角动量是守恒的，角速度不是。
2. 给定角速度对定向求解时会遇到一个问题：角速度是一个三元素**矢量**，而定向是4个元素的**四元数**。那么怎样写一个常微分方程联系四元数和矢量呢？答案是不可以，至少不能直接做到。但我们可以把角速度转换为四元数形式，做一个较奇特的方程去联系定向四元数和角速度四元数。

事实上，当我们以四元数表示刚体的定向，该四元数的第一导数便和刚体的角速度有关，其关系如下说明。首先，我们构建一个角速度四元数，此四元数的x、y及z为角速度矢量的分量，而w分量则是0：

$$\omega = \begin{bmatrix}\omega_x & \omega_y & \omega_z & 0\end{bmatrix}$$

然后，联系定向四元数和角速度四元数的微分方程就是（在此不谈其中原因）：

$$\frac{\mathrm{dq}(t)}{\mathrm{d}t} = \dot{\mathrm{q}}(t) = \frac{1}{2}\omega(t)\mathrm{q}(t)$$

很重要的是，如上所述这里的$\omega(t)$是角速度**四元数**，而$\omega(t)\mathrm{q}(t)$是**四元数**积（见4.4.2.1节）。

因此，我们实际上需要把运动的常微分方程写成（笔者同时列出线性常微分方程，以显两者的相似性）：

$$\text{三维旋转：}\quad \begin{aligned} \mathbf{N}_{\text{net}}(t) &= \dot{\mathbf{L}}(t), \\ \boldsymbol{\omega}(t) &= \mathbf{I}^{-1}\mathbf{L}(t), \\ \omega(t) &= \begin{bmatrix} \boldsymbol{\omega}(t) & 0 \end{bmatrix}, \\ \frac{1}{2}\omega(t)\mathrm{q}(t) &= \dot{\mathrm{q}}(t) \end{aligned} \qquad \text{线性：}\quad \begin{aligned} \mathbf{F}_{\text{net}}(t) &= \dot{\mathbf{p}}(t), \\ \mathbf{v}(t) &= \frac{\mathbf{p}(t)}{m}, \\ \mathbf{v}(t) &= \dot{\mathbf{r}}(t) \end{aligned}$$

若使用显式欧拉法，三维旋转运动常微分方程的最终近似解是：

$$\mathbf{L}(t_2) = \mathbf{L}(t_1) + \mathbf{N}_{\text{net}}(t_1)\Delta t = \mathbf{L}(t_1) + \Delta t \sum(\mathbf{r}_i \times \mathbf{F}_i(t_1)), \qquad \text{(矢量)}$$

$$\omega(t_2) = \begin{bmatrix} \mathbf{I}^{-1}\mathbf{L}(t_2) & 0 \end{bmatrix}, \qquad \text{(四元数)}$$

$$\mathrm{q}(t_2) = \mathrm{q}(t_1) + \frac{1}{2}\omega(t_1)\mathrm{q}(t_1)\Delta t \qquad \text{(四元数)}$$

我们需要定期把定向$\mathrm{q}(t)$重新归一化，以消除浮点小数累计无法避免的误差。

一如既往，这里使用显式欧拉只是做例子之用，我们也可使用速度韦尔莱、RK4或其他更稳定、更精确的数值方法。

12.4.7 碰撞响应

至今的讨论，都假设了刚体没有遇到碰撞，以及其运动不受限于其他形式。当刚体互相碰撞时，动力学模拟必须采取步骤确保它们对碰撞做出真实的响应，并且确保它们永不在模拟步完成之后处于互相穿插的状态。此为**碰撞响应**（collision response）。

12.4.7.1 能量

在讨论碰撞响应之前，我们必须理解**能量**（energy）这个概念。当施力令刚体移动一段距离，我们称该力做了**功**（work）。功代表能量的改变，即是说，力可以对一个刚体系统增加能量（如爆炸），或是从系统中移除能量（如摩擦力）。能量有两种形式。刚体的势能（potential energy）V仅仅是来自其相对力场（如引力场或磁场）的位置而形成的能量。（例如，离地球表面越高的物体含有越多引力势能。）刚体的动能（kinetic energy）T代表该刚体相对于系统中其他刚体的能量。由刚体形成的孤立系统，其总能量$E = V + T$是守恒量。其意义是，若没有在系统中取出能量，或从外界对系统加入能量，孤立系统总能量会维持不变。

线性运动所形成的能量可写成：

$$T_{\text{linear}} = \frac{1}{2}mv^2$$

或以线性动量和速度矢量表示：

$$T_{\text{linear}} = \frac{1}{2}\mathbf{p} \cdot \mathbf{v}$$

类似地，刚体旋转运动所形成的动能为：

$$T_{\text{angular}} = \frac{1}{2}\mathbf{L} \cdot \boldsymbol{\omega}$$

在对所有不同种类的物理问题求解时，能量及能量守恒是极有用的概念。我们将会在下文看到能量在碰撞响应中的角色。

12.4.7.2 冲量碰撞响应

两个刚体在真实世界中碰撞，会产生一连串复杂的事件。刚体会被轻微压缩，然后反弹，改变其速度，并在过程中因为生成热和声音而损失一些能量。多数实时刚体动力学模拟对这些细节进行逼近，当中使用到一个简单的模型。此模型基于分析碰撞体的动量及动能，称为**无摩擦力下瞬时碰撞的牛顿恢复定律**（Newton's law of restitution for instantaneous collisions with no friction）。此定律使用以下的简化碰撞假设。

- 碰撞的力作用于无穷短的时间之内，这个力转化为理想化的**冲量**（impulse）。这样令物体在碰撞时**瞬时**改变速度。
- 物体表面上的接触点无摩擦力。这即等同于说，在碰撞时分离物体的冲量垂直于这两个表面，碰撞冲量无切线上的分量。（当然这仅是一个理想化的假设，12.4.7.5节会再探讨摩擦力）。
- 刚体间的复杂分子下互动，可由单个简单的**恢复系数**（coefficient of restitution）所逼近，习惯上标记为ε。[72]当$\varepsilon = 1$，碰撞是**完全弹性**（perfectly elastic）的，无任何能量损失。（想象两个桌球在空中相撞。）当$\varepsilon = 0$，碰撞是**完全非弹性**（perfectly inelastic）的，或称**完全塑性**（perfectly plastic），一起失去两个刚体的动能。碰撞后两个刚体会黏在一起，继续沿它们在碰撞前的共同质心方向移动。（想象两块黏土碰在一起。）

[72]译注：ε（epsilon）为第5个希腊字母的小写。

所有碰撞分析都基于线性动量守恒。因此，对两个刚体，我们可以写下：

$$\mathbf{p}_1 + \mathbf{p}_2 = \mathbf{p}_1' + \mathbf{p}_2', \qquad \text{或}$$
$$m_1\mathbf{v}_1 + m_2\mathbf{v}_2 = m_1\mathbf{v}_1' + m_2\mathbf{v}_2'$$

当中附撇号（′）的符号代表碰撞后的动量及速度。动能也是守恒的，但我们必须考虑到因热及声音而失去的能量，这些失去的能量标记为T_{lost}项：

$$\frac{1}{2}m_1v_1^2 + \frac{1}{2}m_2v_2^2 = \frac{1}{2}m_1v_1'^2 + \frac{1}{2}m_2v_2'^2 + T_{\text{lost}}$$

若碰撞是完全**弹性**的，能量散失T_{lost}便为0。若完全**塑性**，则能量的散失等同于系统原来的动能，附撇号的动能会变成0，刚体在碰撞后黏在一起。

要使用牛顿恢复定律进行碰撞决议，我们会在两个刚体上施以理想化的**冲量**。冲量像是一个在无穷短时间内作用的力，令施予的刚体瞬间改变速度。冲量可以标记为$\Delta\mathbf{p}$，因为它是动量的改变（$\Delta\mathbf{p} = m\Delta\mathbf{v}$）。然而，多数物理文献会使用$\hat{\mathbf{p}}$（读作“p-hat”），而我们也会使用此符号。

由于我们假设碰撞没有摩擦力，所以冲量矢量必然于接触点上垂直于两个表面。换言之，$\hat{\mathbf{p}} = \hat{p}\mathbf{n}$，当中$\mathbf{n}$是垂直于两个表面的单位法矢量，如图12.27所示。若我们假设表面法矢量指向刚体1，那么刚体1会受到冲量$\hat{\mathbf{p}}$，而刚体2则受到相反方向的冲量。因此，两个刚体在碰撞后的动量，可按碰撞前的动量及冲量表示：

$$\begin{aligned}
\mathbf{p}_1' &= \mathbf{p}_1 + \hat{\mathbf{p}}, & \mathbf{p}_2' &= \mathbf{p}_2 - \hat{\mathbf{p}}, \\
m_1\mathbf{v}_1' &= m_1\mathbf{v}_1 + \hat{\mathbf{p}}, & m_2\mathbf{v}_2' &= m_2\mathbf{v}_2 - \hat{\mathbf{p}}, \\
\mathbf{v}_1' &= \mathbf{v}_1 + \frac{\hat{p}}{m_1}\mathbf{n}, & \mathbf{v}_2' &= \mathbf{v}_2 - \frac{\hat{p}}{m_2}\mathbf{n}
\end{aligned} \tag{12.9}$$

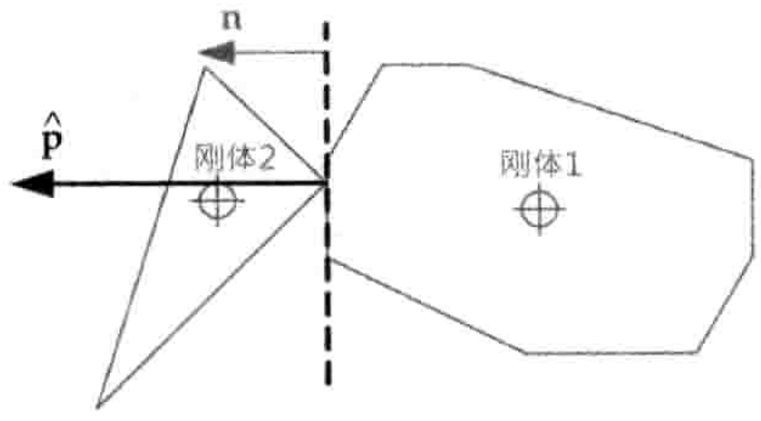


图 12.27: 在无摩擦力的碰撞中，冲量作用于垂直于两表面接触点的直线。此直线以单位法矢量定义。

恢复系数是刚体碰撞前后关系的关键。给定两个刚体质心在碰撞前的速度$\mathbf{v}_1$、$\mathbf{v}_2$，碰撞后的速度$\mathbf{v}_1'$、$\mathbf{v}_2'$，那么恢复系数ε定义为：

$$\mathbf{v}_2' - \mathbf{v}_1' = \varepsilon(\mathbf{v}_2 - \mathbf{v}_1) \tag{12.10}$$

若暂时假设刚体不能旋转，对方程(12.9)及(12.10)求解可得出：

$$\hat{\mathbf{p}} = \hat{p}\mathbf{n} = \frac{(\varepsilon+1)(\mathbf{v}_2 \cdot \mathbf{n} - \mathbf{v}_1 \cdot \mathbf{n})}{\frac{1}{m_1} + \frac{1}{m_2}}\mathbf{n}$$

注意，若恢复系数是1（完全弹性碰撞），并且刚体2的有效质量是无穷大（例如一条混凝土道路），那么$(1/m_2) = 0$、$\mathbf{v}_2 = 0$，那么如我们所料，此表达式会简化为另一刚体的速度矢量对接触法矢量的反射：

$$\begin{aligned}\hat{\mathbf{p}} &= -2m_1(\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n}, \\ \mathbf{v}_1' &= \frac{\mathbf{p} + \hat{\mathbf{p}}}{m_1} = \frac{m_1\mathbf{v}_1 - 2m_1(\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n}}{m_1} \\ &= \mathbf{v}_1 - 2(\mathbf{v}_1 \cdot \mathbf{n})\mathbf{n}\end{aligned}$$

若我们考虑到刚体的旋转，上述的方程解会更复杂一点。在这种情况下，我们需要知道两刚体接触点上的速度，而非质心的速度，然后要计算碰撞后真实旋转效果所需的冲量。本文不详述这些计算方法，Chris Hecker撰写了一篇优良文章[73]，同时描述线性及旋转方面的碰撞响应。有关碰撞响应的背后理论在[15]有更完整的说明。

12.4.7.3 惩罚性力

另一个碰撞响应方法是，在模拟中引入称为惩罚性力（penalty force）的虚构力。惩罚性力的行为，有如在两个刚互相穿透刚体的接触点之间，系上一个坚硬的阻尼弹簧。这种力会在短但有限的时间内，产生所需的碰撞响应。使用此方法时，弹簧常量k实际上能控制互相穿透的持续时间，而阻尼系数b有一些恢复系数的作用。当$b = 0$，无阻尼，没有能量散失，碰撞就是完全弹性的。b越大，碰撞就更塑性。

接下来，我们简单看看用惩罚性力方法解决碰撞的优缺点。优点方面，惩罚性力容易实现及理解。当有3个或以上的刚体互相穿透时，此法也能良好工作。若以刚体结对方式做

[73] http://chrishecker.com/images/e/e7/Gdmphys3.pdf

碰撞决议，是非常难求解的。有一个好例子是索尼的PS3示范，当中把大量橡皮鸭子掉进浴缸，尽管有大量碰撞，但其物理模拟显得又好又稳定。惩罚性力是成就此效果的好方法。

可惜，由于惩罚性力仅响应刚体间的穿透（即相对位置），而非响应相对速度，因此力的方向可能会出乎意料之外，对高速碰撞尤是。有一个经典例子是关于一辆汽车与货车迎头相撞的。由于汽车较货车矮，若仅使用惩罚性力的方法，很容易令惩罚性力垂直向上，而非根据相对速度形成水平方向的惩罚性力。这导致货车车头弹起来，而汽车则从下面驶过。

一般而言，惩罚性力技术对于低速撞击能良好运作，但不适合高度移动的物体。一个可行的办法是结合惩罚性力与其他碰撞决议方法，以权衡大量互相穿透情况的稳定性、响应性，以及高速下更直觉的行为。

12.4.7.4 使用约束解决碰撞

我们将会在12.4.8节中探讨，多数物理系统能把多种约束施于模拟中的刚体运动。如果把碰撞作为不容许穿透的约束条件，那么碰撞就能简单地使用通用的约束求解程序解决。若约束求解程序能迅速地产生高品质的视觉效果，那么就可作为有效的碰撞决议方法。

12.4.7.5 摩擦力

摩擦力（friction）出现于两个持续接触中的刚体之间，阻碍它们的相对移动。摩擦力有许多种类。**静摩擦力**（static friction）是当尝试在表面上滑动静止物体时所受到的阻力。**动摩擦力**（dynamic friction）是物体相对其他物体实质移动时的抵抗力。**滑动摩擦力**（sliding friction）是一种动摩擦力，阻碍物体沿表面滑动。**滚动摩擦力**（rolling friction）是一种静或动摩擦力，施于轮子或其他圆形物体在表面滚动时的接触点。若表面非常粗糙，滚动摩擦力刚足够令轮子滚动而不滑动，那么此滚动摩擦力就会作为一种静摩擦力。若表面比较平滑，轮子可能会打滑，那么滚动摩擦力的动摩擦力形式就产生作用。**碰撞摩擦力**（collision friction）是在两个物体移动中碰撞时，瞬间施于接触点的摩擦力。（这是在12.4.7.1节中讨论牛顿恢复定律时所要忽略的摩擦力。）不同种类的**约束**也可含有摩擦力。例如，生锈的铰链或轮轴在转动时可能受摩擦力矩阻碍。

以下用一个例子重点讲述摩擦力如何运作。线性滑动摩擦力，与滑动物体施于滑动平面法线上的重量分量成正比。物体的重量仅是引力所造成的力$\mathbf{G} = m\mathbf{g}$，此力总是朝下。在与水平面成θ角的斜面上，此力于斜面法线上的分量为$G_N = mg\cos\theta$。而摩擦力f是：

$$f = \mu mg\cos\theta$$

当中的正比常数μ称为**摩擦系数**（coefficient of friction）。此力作用于平面的切线上，其方向与物体尝试或实质运动的方向相反，如图12.28所示。

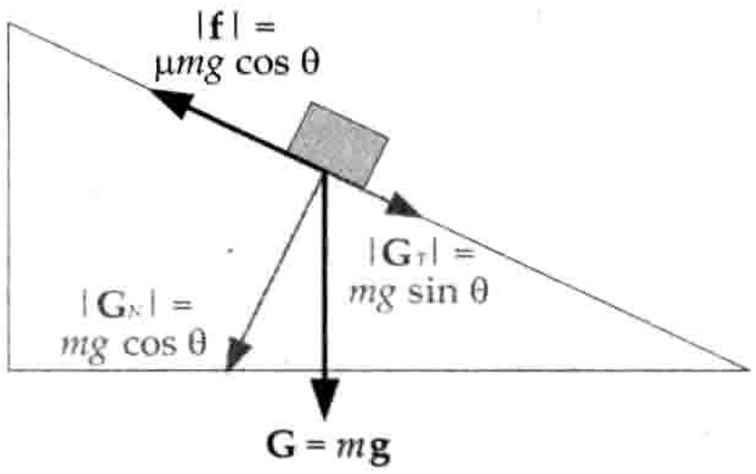


图 12.28: 摩擦力f与物体重量的法矢量分量成正比。该正比常数μ称为摩擦系数。

图12.28也显示了引力施于表面切线的分量$G_T = mg\sin\theta$。此力倾向把物体沿平面往下加速，但由于有滑动摩擦力，此力受f阻碍。因此，在表面切线上的净力为：

$$F_{\text{net}} = G_T - f = mg(\sin\theta - \mu\cos\theta)$$

若倾斜角度令括号内的表达式归零，那么物体会以均速滑动（若物体正在移动），或是静止。若表达式大约为0，物体便会沿表面向下加速。若表达式少于0，物体便会减速至最终静止。

12.4.7.6 焊接

当一个物体在多边形汤上滑动时，会衍生另一问题。回想多边形汤，其名字暗示着，它是一堆无关系的多边形（通常是三角形）。当物体从一个三角形滑动至下一个三角形时，碰撞检测系统会产生额外的伪接触点，因为系统会认为该物体碰到下一个三角形的边缘，如图12.29所示。

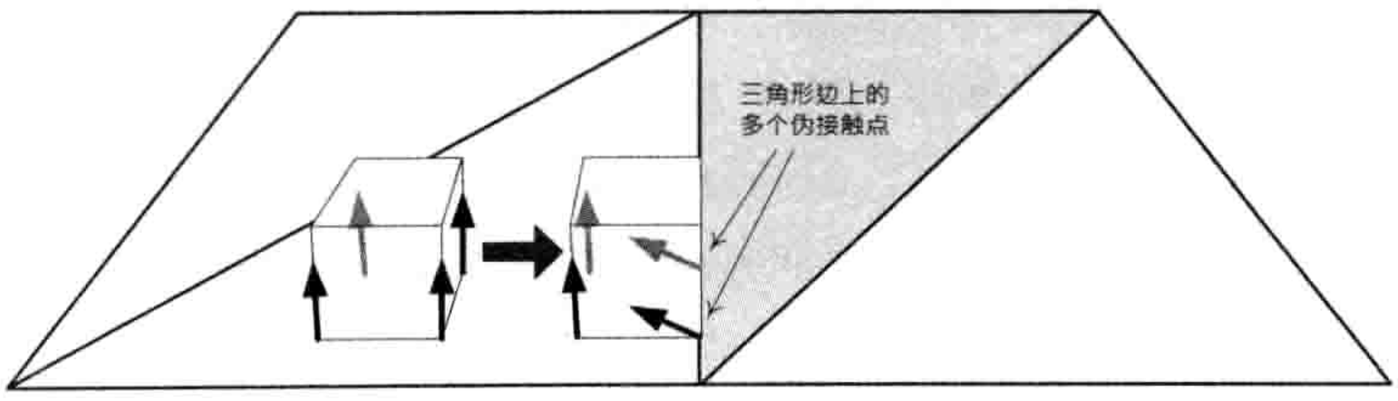


图 12.29: 当物体在两个相邻三角形之间滑动时，可能会对下一个三角形边缘产生伪接触点。

此问题有多个解决方法。其一是分析接触集合，以多种启发法（heuristic）及前一帧的物体接触信息，移除一些伪接触点（例如，若我们知道物体之前沿着一个表面滑行，而现在

有一个接触点靠着当前三角形的边缘，那么就可以丢弃该接触点）。Havok版本4.5之前使用此方法。

自Havok版本4.5开始，使用了一个新技术，就是在网格上加入三角形邻接（adjacency）信息。因此碰撞检测系统可以“得悉”哪些边是内边，并能可靠、迅速地丢弃伪接触点。因为多边形汤中的三角形互相连接在一起，所以Havok称此解决方案为**焊接**（welding）。

12.4.7.7 休止、岛屿及休眠

当通过摩擦力、阻尼或其他方式削减模拟系统中的能量时，移动中的物体会最终静止。这好像是模拟的自然结果，此结果应该能从运动微分方程中自然得出。可惜，在真实的计算机模拟中，休止并非那么容易的事。多种因素会令物体永远抖动而不休止，例如浮点小数误差、计算恢复力时的误差等。因此，多数物理引擎会使用启发式方法，检测物体是否在振荡，是否应该休止下来。可以通过从系统中削减更多的能量确保物体最终休止，或是当物体的平均速度降致一个阈值时就猛然把它停下来。

当物体真正停止移动时（发现自己在平衡状态/equilibrium state），便没有理由继续在每帧对其方程进行积分。为了优化性能，多数物理引擎容许模拟中的动力学物体**进入休眠**（sleep）状态。这样会令那些休眠物体暂时撇除在模拟之外，但这些物体仍会参与碰撞检测。若有任何力或冲量施于休眠物体，或该物体失去令其处于平冲状态的接触点，那么便要唤醒该物体，继续其动力学模拟。

休眠条件

可使用多种条件判断刚体是否合乎休眠的资格。然而并不总是在所有情况下能容易、健壮地做出判断。例如，一支长钟摆可能有非常少的角动量，但仍能在屏幕上看见它的移动。

最常用的平冲判断条件包括：

- 刚体受**支持**：这是指刚体含有3个或以上的接触点（或一个或以上的**平面**接触），能在引力或其他施力下令刚体处于平冲状态。
- 刚体的**线性及角动量**低于预设阈值。
- 线性及角动量的**移动平均**（running average）低于预设阈值。
- 刚体的**总动能**（$T = \frac{1}{2}\mathbf{p} \cdot \mathbf{v} + \frac{1}{2}\mathbf{L} \cdot \boldsymbol{\omega}$）低于预设阈值。动能通常以质量归一化，从而可以不论质量大小把单个域值用于所有物体。
- 对于将休眠的刚体，可以**逐渐减慢**其运动，令它能平滑地停止而不是突然停止。

模拟岛

Havok和PhysX都会自动地把物体分组，以进一步优化性能。可按正在互动的物体分组，或是把近期潜在会互动的物体组成**模拟岛**（simulation island）。每个模拟岛能独立于其他岛进行模拟，这点非常有利于缓存一致性及并行处理。

Havok和PhysX都会以整个岛为单位进入休眠，而非以独立刚体为单位。此方法有优点也有缺点。当一组互动中的物体进入休眠，其性能提升显然更多。另一方面，即使岛中只有一个苏醒的物体，整个岛都必须苏醒。整体而言，优点应该足以抵消其缺点，因此我们很有可能在以后的SDK版本中仍然看到模拟岛的设计。

12.4.8 约束

无约束的刚体有6个自由度（degree of freedom，DOF）：它能在3个维度上平移，并能绕3个笛卡儿轴旋转。**约束**（constraint）限制了物体的运动，削减了部分或完整的DOF。约束可以为游戏中多种有趣行为建模。以下是一些例子。

- 摇晃的吊灯（点对点约束）。
- 可踢、可撞的门，其铰链可被破坏（铰链约束）。
- 汽车的轮子装配（轮轴约束，备有供悬挂系统用的阻尼弹簧）。
- 火车或车轮拉动挂车（刚性弹簧/杆状约束）。
- 绳子或锁链（一串刚性弹簧/杆状约束）。
- 布娃娃（以特殊约束模仿人类骨骼的行为）。

以下数节会简介这些及其他在物理SDK中最常见的约束。

12.4.8.1 点对点约束

点对点约束（point-to-point constraint）是最简单的约束类型。它的作用如同球窝关节（ball and socket joint），刚体中某个指定的点与其他刚体的指定点对齐，除此以外能自由移动，如图12.30所示。

12.4.8.2 刚性弹簧

刚性弹簧约束（stiff spring constraint）很像点对点约束，区别是前者要保持两点分隔一段指定距离。这种约束如像在两个约束点间放置一支隐形杆，如图12.31所示。

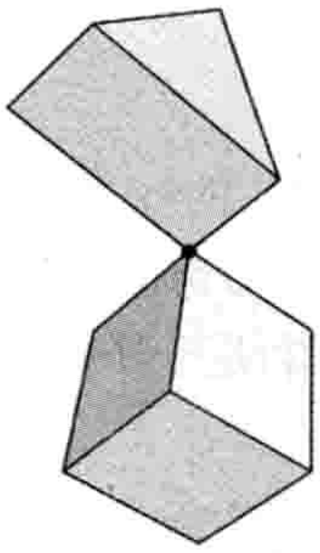


图 12.30: 点对点约束要求刚体A的某点与刚体B的某点对齐。

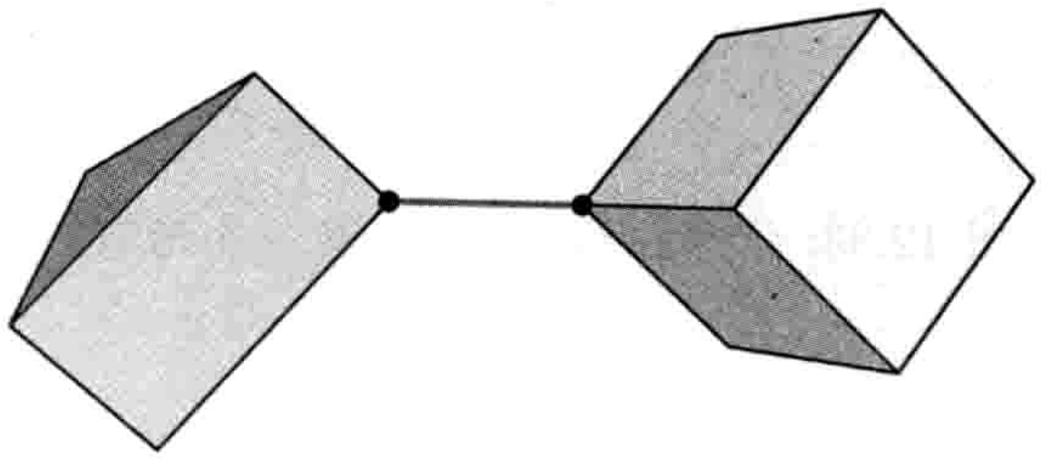


图 12.31: 刚性弹簧约束要求刚体A的某点离开刚体B的某点一段用户指定的距离。

12.4.8.3 铰链约束

铰链约束（hinge constraint）限制旋转运动只能绕铰链旋转（只余一个旋转DOF）。**无限制铰链**（unlimited hinge）如同轮轴，容许物体旋转无限个圈。而**有限制铰链**（limited hinge）令物体只能在预设的角度内绕轴旋转。例如，单向门只能在180°圆弧上移动，否则它就会穿过相连的墙壁了。相似地，双向门的旋转被限制在±180°的圆弧上。铰链约束也可给定某程度的摩擦力，这种摩擦力是以力矩的形式阻碍绕轴的旋转。图12.32展示了一个有限制铰链。

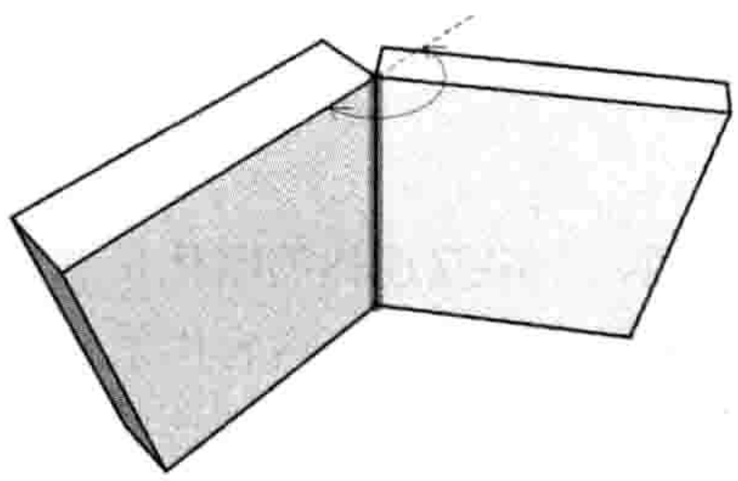


图 12.32: 限制铰链约束能模拟一道门的行为。

12.4.8.4 滑移铰

滑移铰约束（prismatic constraint[74]）的行为如活塞：受限的刚体运动只余一个平移自由度。滑移铰可选择容许或不容许刚体绕活塞的平移轴旋转。当然滑移铰也可以是有限制或是无限制的，含有或不含摩擦力。图12.33是一个滑移铰的例子。

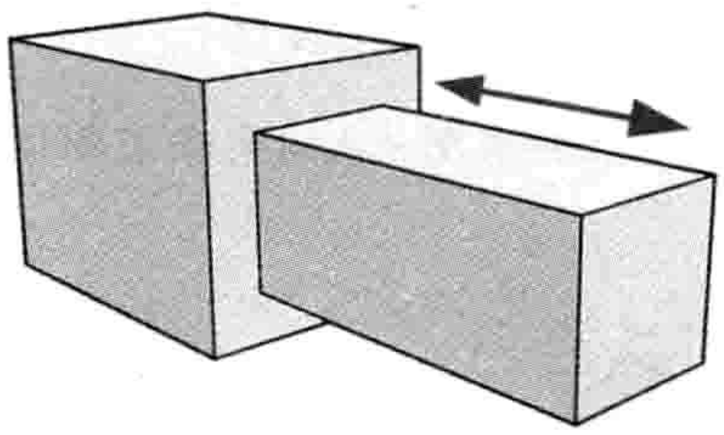


图 12.33: 滑移铰约束的行为像一个活塞。

12.4.8.5 其他常见约束

当然还有许多其他的可行约束类型。以下仅是一些例子。

- **平面**：物体被约束在二维平面上移动。
- **轮子**：通常是无限旋转的铰链，再通过阻尼弹簧加入某形式的垂直悬挂系统[75]。
- **滑轮**（pulley）：在此特殊约束中，一条假想的绳子经过滑轮连接两个物体。这两个物体沿绳子按杠杆比率移动。

约束或许是可破坏的。所谓破坏，是指当施与足够的力时，约束会自动失效。另外，游戏也可按自定义何时破坏的条件开关约束。

12.4.8.6 约束链

由于约束求解程序的迭代性质，一串互连体的刚体长链有时并不容易稳定地模拟。**约束链**（constraint chain）是一个特殊的约束群组，内含供求解程序使用的物体连接信息。这些信息令求解程序能更稳定地处理这些约束链，若欠缺这些信息就难以达到相同效果。

---

[74]译注：Prismatic constraint直译是棱状约束，但似乎滑移铰更合适且更易理解。

[75]译注：如果是汽车的前轮，还要容许转向。所以这基本上只完全去除一个旋转DOF。

12.4.8.7 布娃娃

布娃娃（ragdoll）是模拟人体在死亡或失去知觉时的动作，即整个身体是瘫软的。制作布娃娃的方法是，把一组刚体连接起来，每个刚体代表身体的半刚性部位。例如，我们可能使用胶囊体模拟足、小腿、大腿、手、上臂、下臂、头，或可能包括几个供躯干所用以模拟脊椎的灵活性。

布娃娃中的刚体使用约束互相连接。布娃娃的约束为了模仿真实人体的关节运动而特别设计。我们通常使用约束链提高模拟的稳定性。

布娃娃总是紧密地和动画系统整合。随着布娃娃在物理世界中移动，我们抽取各刚体的位置及定向，并用这些信息驱动动画骨骼中对应关节的位置及定向。所以实质上，布娃娃其实是一种由物理系统驱动的**程序式动画**（procedural animation）。

当然，实现布娃娃并非笔者所说的这么简单。首先，布娃娃中的刚体和动画骨骼的关节未必是一对一的关系，骨骼关节的数量通常比布娃娃刚体的数量多。因此，我们需要一个能够映射刚体至关节的系统（即是需要“知悉”布娃娃中每个刚体所对应的关节）。骨骼中受布娃娃刚体驱动的关节之间，可能还有一些额外关节，因此映射系统必须能判断这些额外关节的正确姿势。然而，这并非精密科学。我们必须应用审美观及/或一些人体生物力学的知识，以达到自然的布娃娃效果。

12.4.8.8 富动力约束

约束也可以加入动力（powered constraint/富动力约束），即外部引擎系统（如动画系统）可以间接地控制布娃娃刚体的平移及定向。

以肘关节为例，它的行为差不多是一个有限制的铰链，有稍少于180°的自由旋转能力。（实际上，肘关节也可以做轴向旋转，但在此讨论中会忽略此自由度。）为了向此约束加入动力，我们把肘关节建模为一个**旋转弹簧**（rotational spring）。这种弹簧会按其偏离预设静止位置的角度，产生与该角度成正比的力矩，$N = -k(\theta - \theta_{\text{rest}})$。现在想象从外面改变这个静止角度，令它总是匹配动画骨骼的关节角度。由于静止角度被改变，弹簧会认为它处于非平衡状态，并施力矩旋转手肘，令关节角度回复至与θ_{rest}对齐。在无其他力或力矩的影响下，各刚体会完全追踪动画骨骼的肘关节运动。但若有其他力的介入（例如下臂接触到不能移动的物体），那么这些力会影响肘关节的整体运动，令运动自然地偏离原来的动画运动。如图12.34所示，这样能产生一个错觉——角色努力以某种方式移动（即动画提供的“理想”动作），但又有时因物理世界的限制而无法如愿（例如，角色的手臂想向前摆动，但被某些东西缠着）。

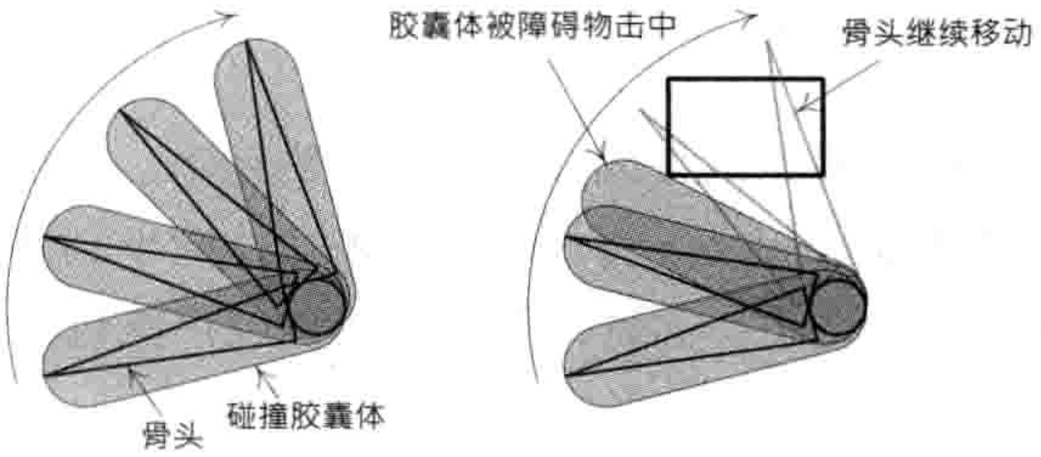


图 12.34: 左图：通过富动力的布娃娃约束，在无外力或力矩的情况下，表示前臂的刚体能完全追踪肘关节的动画。右图：若有障碍物阻挡刚体的运动，就能令运动以真实的方式偏离原来的肘关节动画。

12.4.9 控制刚体的运动

刚体除了要自然地受引力影响移动、响应对场景中其他物体的碰撞，多数游戏设计师还会要求对刚体有某程度的控制能力。例如：

- 通风口对进入其影响范围的物体，施以向上的力。
- 车辆连接挂车后，移动时会对挂车施以拉力。
- 牵引光束可施力吸引未觉察的太空飞船。
- 反引力设备可以令物体浮起。
- 河水的流动产生力场，令浮在河面的物体移往下游。

例子不胜枚举。多数物理引擎通常会对其用户提供多种控制模拟中刚体的方法。以下几节会概述当中最常见的机制。

12.4.9.1 引力

以地球或其他行星的表面为背景的多数游戏中，引力（gravity）普遍存在（就算是太空船上也可能有模拟的引力）。引力从技术上来说不是力，而是（大概为）常数的加速度，因此它对不同质量的物体的效果是相同的。由于引力的普遍存在以及其特殊性，多数物理引擎会把引力加速度作为全局设置。（若读者要做太空游戏，可以把引力设定为0，以消除引力在模拟中的作用。）

12.4.9.2 施力

游戏物理模拟中的刚体可施以任意数量的力。施力总是在有限时间区间中进行的。

（若只施于瞬间，那应称为**冲量**，见下文。）游戏中的力一般是动态的，这些力可以在每帧改变其方向及绝对值。因此，多数物理引擎的施力函数会设计为每帧调用一次，而力的持续影响时间也是在该帧之内。这类函数的签名通过如`applyForce(const Vector& forceInNewtons)`，而持续时间假设为Δt。

12.4.9.3 施力矩

当施力作用于刚体的质心，不会产生力矩，仅影响刚体的线性加速度。若力施于质心之外，就会同时产生线性及旋转加速度。此外，也可以产生**纯力矩**（pure torque），方法是在离质心相同距离的对点上施以相反方向等额的力。这对力所产生的线性运动会互相抵消（因为以线性动力学来说，两个力都是施于质心的），因而只留下旋转效果。这种产生力矩的一对力，称为**力偶**（couple）[76]。引擎可能提供像`applyTorque(const Vector& torque)`这样的特殊函数。然而，若读者用的物理引擎不提供`applyTorque()`函数，读者可自行编写，令它产生适当的力偶[77]。

12.4.9.4 施以冲量

如12.4.7.2节所提及，**冲量**是速度的瞬间改变（或实际上，是动量的改变）。技术上来说，冲量是无穷短时间内的施力。然而，基于时间步的动力模拟中，最短的持续时间为Δt，而Δt不够小去充分模拟冲量。因此，多数物理引擎会提供签名像`applyImpulse(const Vector& impulse)`这样的函数施冲量于刚体。当然，冲量也有两种形式——线性与旋转，良好的引擎会提供函数施以这两种冲量。

12.4.10 碰撞/物理步

我们已谈及实现碰撞及物理系统时，其背后的一些理论及技术细节。现在将会介绍这些系统实际上怎样在每帧进行更新。

每个碰撞/物理引擎都会在更新步时执行以下的基本工作。不同物理SDK可能会以不同次序执行这些工作阶段。然而，以笔者所见，最常用到的方法大概是这样的。

1. 以Δt对施于物理世界刚体的力及力矩计算向前积分，求出次帧的暂定位置及定向。

---

[76] http://en.wikipedia.org/wiki/Couple_(mechanics)

[77] 译注：为此引擎需提供像`applyForceAtPosition(const Vector& force, const Vector& position)`这样的函数以对质心以外的位置施力。注意，施力位置可能以局部空间坐标或世界空间坐标表示。

2. 调用碰撞检测程序库，判断暂定移动是否有令物体间产生新的接触点。（刚体通常会记录其接触点，从而利用时间相干性。因此在模拟的每步中，碰撞引擎只需判断是否有失去之前的接触点，以及是否有加入新的接触点。）
3. 进行碰撞决议。常用的方法是使用冲量、惩罚性力，或作为以下约束求解的一部分。
4. 以约束求解程序满足约束条件。

在第4步完结之时，有些刚体可能已移离第1步所计算出来的暂定位置。这种移动可能会导致物体间有其他互相穿插的情况，或可能破坏了其他之前已满足的约束。因此，需要重复第1步至第4步（根据碰撞及约束的决议方式，有时候仅需重复第2步至第4步），直至（a）成功决议所有碰撞并且满足所有约束，或（b）超过预设的迭代数目。对于后者，求解程序实际上是“放弃”了，期望未解决的问题在之后的模拟帧能自然地解决。这样也能在多个帧里分摊碰撞及约束决议的成本，避免产生性能尖峰（spike）[78]。然而，若错误太大、时步太长或不稳定，这种做法也可导致貌似不正确的行为。可以在模拟中混合惩罚性力，以使模拟能随时间逐渐解决这些问题。

12.4.10.1 约束求解程序

约束求解程序（constraint solver）本质上是一个迭代算法，尝试**最小化**刚体在物理空间的实际位置/定向与约束所定义的理想位置/定向的**误差**，以同时满足大量的约束。因此，约束求解程序本质上是迭代式误差最小化算法。

我们首先看看约束求解程序如何解决平凡（trivial）的情况——一个铰链约束连接两个刚体。在每次物理模拟步中，数值积分器会求出两个刚体的新暂定变换。然后，约束求解程序先计算两个刚体间的相对位置，然后计算它们对共有旋转轴的位置/定向的误差。若检测到任何误差，求解程序就以最小化或消除这个误差的方式移动刚体。由于系统中无其他刚体，第2个迭代中应该找不到新的碰撞接触，并且约束求解程序会发现那唯一的铰链约束现在已被满足。因此，结束循环，不需要再进行更多迭代。

当必须同时满足超过一个约束时，可能需要更多的迭代。在每个迭代中，数值积分器有时候会移动刚体，趋向令它们脱离约束，而约束求解程序则会趋向令它们回复至合乎约束。幸运的话，通过在求解程序中使用谨慎设计的错误最小化方式，这种反馈循环最终应该可以求得合法解。然而，解答并不一定是精确的。这就是为什么含物理引擎的游戏中，有时能看见一些貌似不可能的行为，例如，可延长的锁链（铁环之间有空隙）、物体间轻微互相穿插，或铰链在一瞬间移动至可动范围以外。约束求解程序的目的为最小化误差，并不总是能完全消除误差。

[78] 译注：这是指某帧的持续时间比平均值高许多，做成“顿卡”，降低游戏的流畅性。

12.4.10.2 各引擎的差异

以上所说，当然过度简化了物理/碰撞引擎在每帧所做的事。在不同的物理SDK中，多个计算阶段的方式，以及它们的相对次序，也会有所出入。例如，有些约束类型是以力和力矩建模的，这些约束不是由约束求解程序处理，而是由数值积分器处理。碰撞可以在数值积分之前运行，而非之后。碰撞也可能用不同方法解决。本节的目的，仅让读者浅尝这些系统的工作方式。要更深入理解某一SDK的详细运作方式，读者必须要阅读其文档，并可能需要参考它的源代码（假设读者能得到相关的代码）。富好奇心及勤奋的读者，可以先下载ODE[79]及PhysX，对它们进行实验，因为这两个SDK都是免费的。读者可以从ODE的维基网站[80]中学习许多相关知识。

12.5 整合物理引擎至游戏

显然，仅仅碰撞/物理引擎本身并无大用处，它必须要整合至游戏引擎中才能发挥所长。本节讨论碰撞/物理引擎和其他游戏代码之间的最常见整合点。

12.5.1 连接游戏对象和刚体

碰撞/物理世界中的刚体和碰撞体，只不过是一些抽象的数学描述。要在游戏中利用它们，需把它们和在屏幕上的视觉表示连接起来。通常我们不会直接绘画刚体（除了作为调试之用）。取而代之，组成虚拟游戏世界的逻辑对象，会使用刚体来描述其形状、尺寸及物理行为。我们在第14章会深入探讨游戏对象，此时此刻，我们只需要有一个游戏对象的直觉形象——游戏世界中的逻辑实体，如一个角色、一架载具、一把武器、一个飘浮中的补血品等。因此，物理世界的刚体和屏幕上的视觉表示之间并非直接相连，而是靠逻辑对象作为两者之间的枢纽，如图12.35所示。

一般来说，游戏对象在碰撞/物理系统中会表示为零或多个刚体。以下列出3个可能的情形。

- **零个刚体：**在物理世界中不含刚体的游戏对象是当作非固体的，因为它们完全无任何碰撞表示方式。玩家或非玩家不能互动的装饰性对象可能无碰撞，例如空中飞翔的雀鸟，或可观但不可达的游戏世界局部。有些对象基于某种原因需手工处理碰撞（而不使用碰撞/物理引擎），也适用于这种方式。

[79]译注：译者更建议尝试Bullet Physics。

[80]http://opende.sourceforge.net/wiki/index.php/Main_Page

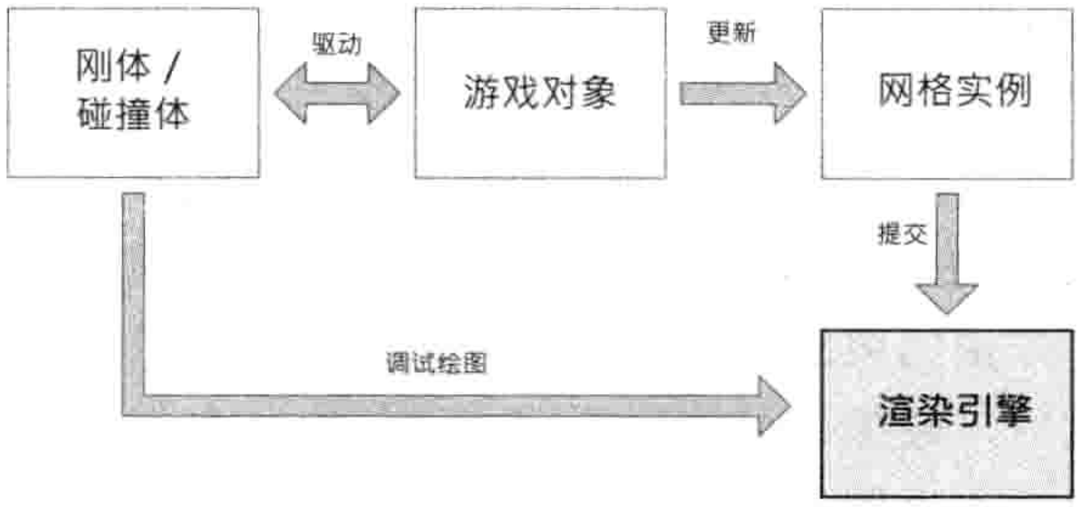


图 12.35: 通过游戏对象把刚体连接其视觉表。引擎可能会提供另一个可选的渲染途径，为调试目的视觉化刚体的位置。

- **一个刚体**：大多数简单游戏对象只需由单个刚体表示。这种情形中，刚体的碰撞形状会尽量逼近游戏对象的视觉表示，而刚体的位置/定向则完全匹配游戏对象本身的位置/定向。
- **多个刚体**：有些复杂的游戏对象是由碰撞/物理世界中的多个刚体所表示的。例如，角色、机械、载具，或任何由多件固体组成的对象。这些游戏对象通常会利用骨骼（即仿射变换的层阶结构）跟踪各组件的位置（当然也可使用其他可行方法）。刚体通常连接至骨骼关节，令每个刚体的位置/定向对应至其中一个关节的位置/定向。骨骼的一些关节可能由动画驱动，那么对应的刚体只会简单跟随动画。相反，物理系统可能会驱动刚体的位置，间接地控制关节的位置。从关节至刚体的映射可以是一对一关系，也可以不是；有些关节可以完全由动画控制，有些则连接至刚体。

游戏对象及刚体之间的连接，当然必须由引擎管理。通常，每个游戏对象会管理自己的刚体，需要时创建或销毁刚体，把这些刚体加进物理世界或从物理世界中移除，并维护刚体位置和游戏对象或其关节位置的联系。对于含多个刚体的复杂游戏对象，便可使用某种包裹类（wrapper class）管理它们。这样做可以避免游戏对象直接管理一组刚体的细节，而且令多种游戏对象使用统一方式管理它们的刚体。

12.5.1.1 物理驱动的刚体

若游戏含有刚体动力学系统，那么我们会假设游戏中至少有一些对象会完全由模拟驱动。这些游戏对象称为**物理驱动**（physics-driven）对象，例如瓦砾碎片、建筑物爆炸、山坡上的滚石、空弹匣及弹壳。

物理驱动刚体连接其游戏对象的方式，是通过步进模拟后，向物理系统查询刚体的位置/定向。之后，把这个变换施于整个游戏对象，或施于某个关节，或施于游戏对象中的某些其他数据结构。

例子：构造一个含可拆门的保险箱

当物理驱动的刚体连接至骨骼的关节，刚体通常在**受约束**的情况下去产生所需的运动。作为例子，我们看看如何建模一个有可拆门的保险箱。

视觉上，我们假设那个保险箱只含一个三角形网格模型，此模型有两个子网格，一个为保险箱外壳而设，一个为保险箱门而设。我们采用含两个关节的骨骼去控制这两块部件的运动。根关节绑定至保险箱外壳，而子关节则绑定至门，旋转该关节时会令门适当地开合。

保险箱的碰撞几何也拆分成两块独立的部件，一个供保险箱外壳所用，一个供保险箱门所用。这两块部件用于在碰撞/物理世界中创建两个完全独立的刚体。模拟保险箱外壳的刚体绑定至骨骼中的根关节，而模拟门的刚体则连接至门关节。然后，在物理世界中加入一个铰链约束，确保两个刚体在动力学模拟时，门的刚体能相对于保险箱外壳正确开合。表示外壳和门的两个刚体，其运动会用于更新骨骼中两个关节的变换。动画系统产生骨骼矩阵调色板之后，渲染引擎便可使用物理世界的刚体的位置渲染外壳和门的子网格。

若在某刻要把门炸开，可破坏铰链，并在刚体上施以冲量令它们吹飞。视觉上，玩家会觉得门和外壳变成独立的物体。但事实上，它仍然是单个游戏对象，以及单个含两关节两刚体的三角形网格。

12.5.1.2 游戏驱动刚体

多数游戏中，游戏世界的某些物体需要以非物理方式移动。这种物体的运动可以由动画、样条路径或人类玩家所控制。我们通常希望这些物体参与碰撞检测，例如，令它们能推开物理驱动的物体，但我们不希望物理系统对这些物体有任何影响。为了支持这种物体，多数物理SDK提供一种特别的刚体类型，称为**游戏驱动刚体**（game-driven body）。（Havok称之为“受关键帧控制”的刚体。）

游戏驱动刚体不受引力所影响。物理系统也把它们当作含有无穷质量（通常会把质量标示为0，因为这是物理驱动刚体的无效值。）无穷质量能确保模拟中的力和力矩无法改变游戏驱动刚体的速度。[81]

要在物理世界中移动游戏驱动刚体，我们不能简单地在每帧设置其位置及定向，以配合对应游戏对象的位置。因为这么做会产生不连贯性，令物理模拟非常难求解。（例如，物理驱动刚体可能会发现它突然穿插进一个游戏驱动刚体，但却无法知悉该游戏驱动刚体的动量做出碰撞决议。）因此，通常会使用冲量移动游戏驱动刚体。冲量即速度的瞬时改变，

[81] 译注：通常在计算时会把力乘以质量的倒数以求出加速度。物理引擎可储存质量的倒数（而非质量本身），那么这种刚体可设其质量倒数为0，求出的加速度也会是0。

以时间向前积分就能令刚体在时步末时到达所需的位置。多数物理SDK会提供简便的函数，计算所需的线性及旋转冲量，以使刚体在次帧时能到达所需的位置及定向。当移动游戏驱动刚体时，若该刚体需要停下来，我们必须小心地把其速度清零。否则，刚体会无止境地沿其轨道移动。

例子：含动画的保险箱门

我们沿用含可拆门的保险箱例子。假设我们希望某角色走到保险箱前，拨动密码组合，打开保险箱门，存放一些金钱，再关门并锁上保险箱。稍后，我们希望另一角色以不太文明的方法——炸开保险箱门——取得保险箱里的金钱。要达成这些效果，保险箱的模型要加入转盘的子网格，以及令它旋转的关节。无须把转盘当作刚体，除非我们希望炸开保险箱时转盘会和门分离弹开。

在角色打开及关闭保险箱的动画时段内，保险箱的刚体会处于游戏驱动模式。这时候动画会驱动关节，关节驱动刚体。然后，当炸开保险箱时，我们把刚体转回物理驱动模式，断开铰链约束，施以冲量，就可以看到门飞脱出来。

读者可能已注意到，本例中的铰链并不是实际上需要的。除非我们希望保险箱门在某刻是开启的，并且要看到保险箱门因移动保险箱或碰到其他东西时会自然地摇晃，才会需要使用铰链约束。

12.5.1.3 固定刚体

多数的游戏世界是由静态几何物体和动态物体所组成的。为了模拟游戏世界的静态组件，多数物理SDK会提供一种特别的刚体，称为**固定刚体**（fixed body）。固定刚体的行为有如游戏驱动刚体，但固定刚体并不参与动力学模拟。它们实际上就是只有碰撞的刚体。此优化能大幅提升多数游戏的性能，对只含少量动态物体于大型静态世界中移动的游戏尤有帮助。

12.5.1.4 Havok的运动类型

在Havok中，所有刚体的类型都是由`hkpRigidBody`类的实例所表示的。每个实例含有一个**运动类型**（motion type）的字段，用以告诉系统该刚体是固定的、游戏驱动的（Havok称之为“用关键帧的/keyframed”），或是物理驱动的（Havok称之为“动力学的/dynamic”）。若以固定运动类型来创建一个刚体，其运动类型就再也不能改变。除此以外，可以在运动时改变刚体的运动类型。例如，一件在角色手上的物体应该使用游戏驱动类

型。但当该物体从角色手上丢掉时，它就应该改为物理驱动，令动力学模拟接管其运动模式。在Havok中，只须简单地在物体离手时改变其运动类型便可。

为了给予Havok一些刚体的惯性张量的信息，运动类型数目其实还翻了一倍。“动力学”运动类型拆分为数个子类别，包括“球形惯量的动力学”、“矩形惯量的动力学”等。Havok利用刚体的运动类型，基于惯性张量内部结构的假设来判断该应用哪些优化。

12.5.2 更新模拟

物理模拟当然必须定期更新，通常每帧一次。这不仅涉及步进模拟（数值积分、碰撞决议及施以约束），也需要维持游戏对象和其刚体的联系。若游戏需要对任何刚体施力或冲量，也必须每帧进行。以下是完整地更新物理模拟所需的步骤。

- **更新游戏驱动刚体：**更新物理世界中所有游戏驱动刚体的变换，令这些变换匹配其相连的游戏世界中对象（游戏对象或关节）的变换。
- **更新phantom：**每个phantom形状的行为，如同欠缺刚体的游戏驱动碰撞体。它用作几种碰撞查询。在物理步进之前，需要更新所有phantom的位置，那么执行碰撞检测时这些phantom就会处于合适的位置。
- **施以力、冲量，并调整约束：**更新游戏正在施行的力。本帧所发生的游戏事件，其产生的冲量也在此时施行。按需调整约束。（例如，检查可破坏的铰链是否受损毁，若然则令物理引擎移除该约束。）
- **步进模拟：**如12.4.10节所述，必须定期更新碰撞及物理引擎。更新内容包括：对运动方程进行**数值积分**，以求出所有刚体的次帧物理状态；执行**碰撞检测**算法，以求出物理世界中要增加或删减的刚体接触；**碰撞决议**；**施行约束**。视不同SDK而定，这些更新可能藏于单个不能分割的`step()`函数中，也可能可以逐一执行。
- **更新物理驱动的游戏对象：**从物理世界中获取物理驱动物体的变换，然后用这些变换更新相对的游戏对象或关节，使两者匹配。
- **phantom查询：**在物理步进之后，可读取phantom形状的接触信息，以做出游戏中的决定。
- **执行碰撞投射查询：**以同步或异步方式启动光线及形状投射。当可以获得这些查询的结果时，多个引擎系统就可利用这些结果来做出决定。

这些任务通常以上述次序执行，但光线和形状投射理论上可以置于游戏循环中的任何位置。显然，在步进模拟前更新游戏驱动物体并施以力/冲量，仍是合理的次序，这样才能令模拟“见到”这些更新的后果。相似地，物理驱动的游戏对象应该总是置于步进模拟之

后，以确保使用到最新的刚体变换。通常渲染置于游戏循环之末，这样才能确保我们渲染的是某刻一致的游戏世界视图。

12.5.2.1 安排碰撞查询的时间

为了对碰撞系统查询最新的信息，在每一帧中，我们需要在物理步进后才执行碰撞查询（光线及形状投射）。然而，物理步通常执行至帧末，在此之前游戏逻辑已做好大部分决定，并且也已决定好所有游戏驱动刚体的新位置。那么，应该在什么时候执行碰撞查询？

此问题并无易解。我们有多种选择，多数游戏会采用以下的一些或全部选择。

- **基于前一帧的状态做决定：**许多情况下，我们可使用前一帧的碰撞信息正确地做出决定。例如，我们可能希望知悉玩家是否站在一些物体之上，以决定本帧他应否开始掉下来。在此情况下，我们可以安全地在物理步进之前执行碰撞查询。
- **接受1帧延迟：**就算真的想知道**本帧**发生的事情，我们可能要忍耐1帧的延迟才取得碰撞查询结果。通常对于移动得不太快的物体，可以做出这种让步。例如，我们可能要移动一物体，然后希望知悉该物体是否在玩家的视线中。玩家可能不会注意得到这种查询中的差1帧错误。若然这样，我们可以在物理步进前执行碰撞查询（取得前一帧的碰撞信息），把这些结果当作本帧末碰撞状态的近似值来运用。
- **在物理步进后执行查询：**另一个方法是在物理步进之后才执行某些查询。当基于这些查询结果的决定可延至帧后期才做出，这种做法便可行。例如，依赖碰撞查询的渲染效果便可以这样实现。

12.5.2.2 单线程更新

非常简单的单线程游戏循环可能是这样子的：

```
F32 dt = 1.0f / 30.0f;

for(;;) // 主游戏循环
{
    g_hidManager->poll();

    g_gameObjectManager->preAnimationUpdate(dt);
    g_animationEngine->updateAnimations(dt);
    g_gameObjectManager->postAnimationUpdate(dt);
    g_physicsWorld->step(dt);
    g_animationEngine->updateRagDolls(dt);
```

```
    g_gameObjectManager->postPhysicsUpdate(dt);
    g_animationEngine->finalize();

    g_effectManager->update(dt);
    g_audioEngine->update(dt);
    // 等等
    g_renderManager->render();
    dt = calcDeltaTime();
}
```

在此例子中，游戏对象分3阶段进行更新：第1阶段是于动画运行之前（例如在该阶段编排新的动画），第2阶段是于动画系统计算最终局部姿势及暂定全局姿势之前（但未生成最终全局姿势及矩阵调色板），第3阶段是在物理步进之后。

- 游戏驱动刚体的位置通常于`preAnimationUpdate()`或`postAnimationUpdate()`中更新。对于每个游戏驱动刚体，其变换通常会设置为匹配拥有该刚体的游戏对象，或是匹配骨骼中的一个关节。
- 物理驱动刚体的位置通常是在`postPhysicsUpdate()`中被读取，其位置会用于更新游戏对象的位置，或是其骨胳中的一个关节位置。

步进物理模拟的频率是一个重要的考虑事项。常数时步（Δt）对大部分数值积分方法、碰撞检测算法及约束求解程序来说都是最好的。一般而言，可用理想的1/30s或1/60s步进物理/碰撞SDK，并管理整个游戏循环的帧率。

12.5.2.3 多线程更新

当物理引擎整合至支持多处理器或多线程的游戏引擎时，事情会变得更复杂一点。在7.6节中我们谈及，为了从多处理器硬件中获益，有多种构造游戏循环的可行方法。现在我们关注应用这些技巧时，一些和物理引擎相关的专门问题。

在另一线程运行物理

其中一个选择是在专用的线程上运行物理/碰撞引擎。读者可能会想到，这类设计可能会引起竞态条件（race condition）。若在不合适的时候更新游戏驱动刚体，物理线程可能会在模拟中使用到过期的位置信息。同样，若模拟不能在更新物理驱动对象前完成，游戏对象也可能取得过期的位置信息。

此问题的解决方法之一是，安排物理及主线程互相等待——这称为**线程同步**（thread

synchronization）。线程同步由互斥锁（mutex）、信号标（semaphore）或临界区域（critical section）实现。线程同步通常是相对高昂的操作，因此我们一般会尽量减少线程间的同步点。以物理引擎来说，我们至少需要两个同步点：一个容许启动该帧的物理模拟（于更新所有游戏驱动刚体之后），另一个是当模拟完成后通知主线程（从而容许主线程查询物理驱动刚体的信息）。

为了降低同步点的数目，策略之一是使用命令队列（command queue）做线程间通信。首先，主线程锁定一个临界区域，再写一些命令至队列，然后尽快解锁。物理引擎遇到合适时机，就从队列中取得下一批的命令，这时也需要锁定临界区域，以确保主线程不会同时覆写正在读取的队列。

使用碰撞查询会使问题变得更为复杂。为了令多个线程能同时存取碰撞/物理世界，一些物理引擎（如Havok）会容许对世界分别以读取或写入模式进行锁定。那么碰撞查询可以在游戏循环中任何时间执行（那时候以读取模式锁定世界），除非当时物理世界正在更新（那时候会以写入模式进行锁定）。

分叉及汇合

在物理中使用分叉/汇合（fork/join）架构的好处之一在于，能够从本质上消除所有线程同步问题。主线程如常执行，直至需要步进物理时，我们把步进的过程fork成独立的线程（最理想地是，为每个处理器核或硬件线程配置一个线程），使步进能尽快执行。当所有线程完成工作后，就要整理合并结果，然后主线程便能如单线程的情况继续运行。当然，这要行得通，物理系统必须要设计成支持fork/join。多数物理SDK（包括Havok及PhysX），都使用到的概念，即把互不影响的刚体分组。此设计适合fork/join架构，因为每个岛屿可以动态地分布至可用的线程上。

作业模型

若物理SDK容许更新步的个别阶段（积分、碰撞检测、约束求解、CCD等）独立地执行，那么就可以充分利用到作业模型（job model）。作业模型能令每个阶段在最方便的时刻起动，然后主线程在等待物理引擎的工作时能执行其他不相关的工作。

作业模型对碰撞查询（光线/形状投射）更为有用。因为游戏引擎通常只需要在每帧步进物理模拟一次，但可能要在游戏循环的不同位置进行多次碰撞查询。若使用轻量的作业去执行查询，我们能在任何有需要的时候起动这些作业。另一方面，若碰撞查询只能在帧的某些时候运行（因为它们以分叉或专用线程执行），游戏程序员的工作就变得更困难。

游戏程序员需要把碰撞查询置于一个队列中，而这些查询在帧的某时候以批次方式执行。图12.36比较了这两种方式。

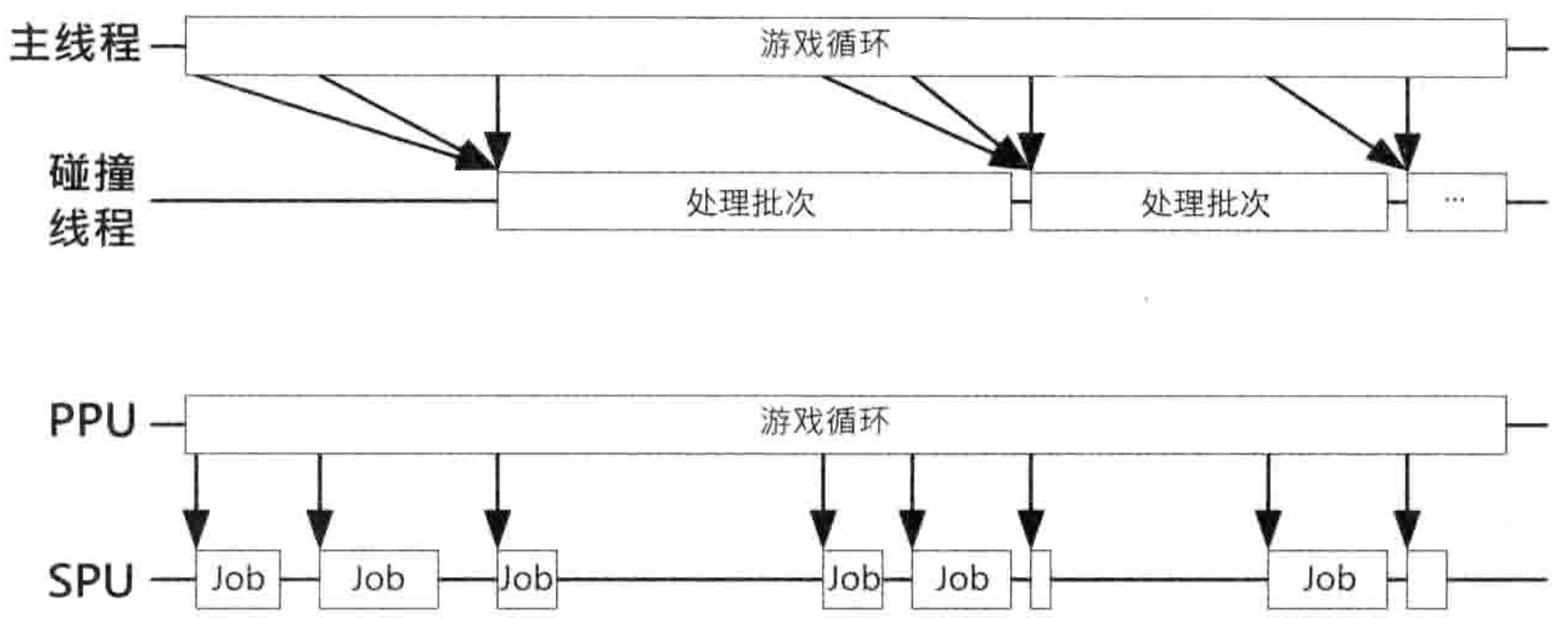


图 12.36: 碰撞查询通常是批次进行的，在游戏循环中几个特选的时间点执行。然而，采用作业模型时，查询可以在任何时刻启动，不需要以批次方式执行。

12.5.3 游戏中碰撞及物理的应用例子

为了更具体地讨论碰撞及物理，以下会俯览几个常见例子，说明碰撞及/或物理通常如何应用于真实游戏之中。

12.5.3.1 简单刚体游戏对象

许多游戏包含简单的物理模拟物体，例如武器、可拾取及投掷的石块、空弹匣、家具、搁板上可射击的物体等。这些物体的实现方式，可以通过创建自定义的游戏对象类，并在其中加入对物理世界刚体（如Havok中的`hkpRigidBody`）的参考。或者我们可以创建一个附加组件类，负责处理简单的刚体碰撞及物理，并容许此功能加至引擎中几乎任何类型的游戏对象。

简单的物理物体常会在运行时改变其运动类型。当某个对象在角色手中时，它是游戏驱动的；但当它离手自由掉落时，就会变成物理驱动的。

想象有一件简单的物理物体置于桌上或搁板之上，在某刻被子弹或其他东西击中。那么该物体应采用哪种运动类型？我们应该设置它成为物理驱动，由模拟使它进入休眠，还是应该把它设置为游戏驱动，直至被击中才改变为物理驱动？答案主要取决于游戏的设计。若我们需要严格控制物体是否可以被击落，那么就应该使用游戏驱动方式；否则物理驱动可能已足够。

12.5.3.2 弹道

无论读者是否赞同游戏暴力，子弹及其他形式的抛射物（projectile）仍然是大量游戏的重要元素。以下介绍这些物体通常如何实现。

有时候子弹是以光线投射实现的。在开火的那一帧，我们进行一次光线投射，判断击中了什么物体，并立即把相关影响施于受击的物体。

可惜，光线投射方式并没有交待抛射物的行程时间。它也没有处理轨道因引力而造成的稍微下降[82]。若这些细节对某游戏而言是重要的，我们就可以用真正的刚体模拟抛射物，这些抛射物会随时间于碰撞/物理世界中移动。此方法特别适合较慢的抛射物，例如投掷物体或火箭炮。

实现子弹及抛射物要注意及处理许多问题，以下是几个常见的例子：

- 光线来自摄像机的焦点，还是来自玩家手中武器的前端？此问题对于第三人称射击游戏更为突出，因为发射自玩家枪械的光线，与投射自摄像机焦点通过屏幕中心十字线（reticle）的光线，往往并不是同一直线。这样会导致一些问题状况，例如十字线对准了目标，但第三人称的角色明显在障碍物之后，从角色的角度来说并不能射击到目标。通常必须使用多种“小技巧”确保玩家体验到所瞄准的就能击中，并同时维持在屏幕上显示貌似真实的画面。
- 碰撞几何体和渲染几何体不匹配，也会导致一些问题状况。例如，玩家可以从一件物体的隙缝或其边缘看到目标，但其碰撞几何体是实心的，所以子弹不能到达目标。（此问题通常只对玩家角色造成问题。）解决办法之一是弃碰撞查询，用渲染查询判断光线是否能击中目标。例如，在一个渲染阶段中，我们生成一张纹理，其每个像素储存对应游戏对象的唯一标识符。然后就可以读取此纹理，判断十字线下是否为敌人角色或其他合适的目标。
- 若抛射物需要一定时间才能达至目标，AI角色射击时可能需要计算时间差。
- 当子弹击中目标，我们可能要触发一个音效或粒子效果，放置贴花（decal），或执行其他工作。

12.5.3.3 手榴弹

游戏中的手榴弹（grenade）有时候会实现为自由移动的物理对象。然而，这样会令受控能力大幅削减。为了重拾部分控制能力，可以对手榴弹施以多个人造的力或冲量。例如，

[82] 译注：子弹发射后也是沿抛物线运动的，其实下降幅度也可以很大。2007年5月1日澳门一位司警意图驱散人群，向天开枪，子弹怀疑越过30m高的望厦山，击中300m外一位摩托车司机的肩背。

我们可以在手榴弹初次碰到地面时，施以极大的空气拖曳力，试图限制它弹离目标的距离。

有些游戏团队则完全手工地管理手榴弹的运动。他们可以预先计算好手榴弹的弹道，然后通过一连串的光线投射判断掷出手榴弹后会击中哪个目标。弹道甚至可以在画面上显示，供玩家知悉。当掷出手榴弹后，游戏令它沿其弧形弹道前进，然后可以小心控制反弹，令它不会弹离目标太远，同时仍保持自然。

12.5.3.4 爆炸

在游戏中，爆炸（explosion）通常由几个部分组成：一些视觉效果，如火球烟雾；一些音效模仿爆炸的声响及对世界中游戏对象的影响；往外增长的破坏半径，影响半径范围内的任何对象。

当一个对象处于爆炸半径之内，我们通常会扣减其生命值，并产生一些运动模拟冲击波（shock wave）的效果。这些效果或可用动画制作。（例如，角色对爆炸的反应最好用动画表现。）我们也可能想完全用动力学模拟驱动冲击的反应。为此，我们可以令爆炸向范围内所有合适的物体施以冲量。由于这些冲量是辐射状的，其方向很容易计算，只需要把受影响的物体中心减去爆炸中心的矢量归一化，然后再以爆炸强度缩放该单位矢量（并可能按距离做衰减）。

爆炸也可以与其他引擎系统互动。例如，我们可以把“力”施与植物动画系统，令花草树木都会受爆炸影响而瞬间弯曲。

12.5.3.5 可破坏物体

可破坏物体（destructible object）[83]常见于许多游戏中。这些物体奇特之处在于，它们起初未损坏时看上去是单个聚合物体，但它们可以破裂为多个碎片。我们可能希望这些碎片一块一块地剥落，令物体逐渐地“削除”，或是我们可能只需要单次的灾难性的爆炸。

形变体（deformable body）模拟（如DMM 引擎里的形变体）能自然地处理可破坏物体。然而，我们也可以用刚体动力学实现可破坏物体。通常的做法是，把模型切割为多个碎片，并为每个碎片设置一个刚体。有鉴于性能优化及/或观感，我们可能会制作一个未破坏的渲染及碰撞几何版本，作为单独的实物。当物体要开始破裂时，就会用破坏版本替代此模型。在其他情况下，我们或可以一直使用已分割的模型。例如，这种做法适用于一堆砖块、叠在一起的碗碟。

[83]译注：在图形学中，这种技术称为破裂模拟（fracture simulation）。

要模拟由多块东西组成的物体，我们可以简单地叠起一堆刚体，由物理模拟自行处理。这种方式对于高质量的物理引擎或是可行的（但并不总是容易做得好）。然而，若我们想要一些好莱坞式的特效，简单地叠起刚体未必凑效。

例如，我们可能希望定义物体的结构。有些部分可能是**不能被破坏的**（indestructible），如墙基或汽车的底盘。其他部分可能是**非结构性的**（non-structural），当被子弹或其他物体击中时只会简单地剥落。而另一些部分可能是**结构性的**（structural），当受击时，不单自己会剥落，也会把力传授至置于其上方的部分。还有一些部分是易爆炸的（explosive），受击后可引至连锁爆炸，或是把破坏力传送至整个结构。我们可能会把一些碎片设为角色可使用的合法**掩护**点（cover point）。这样意味着可破坏物体系统可能会关联至掩护系统。

我们也可能希望可破坏物体有生命值。物体的损害程度可逐渐提升，直至最终完全崩溃。或是每个部分有其生命值，可能需要多枪或多次撞击才会破裂。我们也可以加入约束，令物体悬挂着受破坏的部分，而不是把它们完全分离。

我们也可能希望这些结构要花上一段时间才会完全崩塌。例如，若一座长桥的一端爆炸，崩塌会慢慢地从一端扩散至另一端，那么那座桥才会显得宏大。物理系统不一定能简单地模拟这种效果，因为它会同时唤醒模拟岛中的所有刚体[84]。取而代之，这类效果可通过审慎使用游戏驱动运动类型实现。

12.5.3.6 角色机制

一些游戏如保龄球、弹珠台或《Marble Madness》，其“主角”是一个在虚拟游戏世界中滚动的球。对这类游戏而言，我们可以很好地用物理模拟，以自由运动的刚体模拟这些球，并通过施力或冲量控制它在游戏中的移动。

然而，在基于人物角色的游戏中，我们通常不会使用这种方式。人形或动物角色的移动太过复杂，通常不能用力和冲量控制其移动。取而代之，我们通常会使用一组游戏驱动的胶囊形刚体模拟角色，每个刚体连接至角色骨骼中的关节。这些刚体主要用于子弹受击检测，或是用于产生一些二次效果，例如角色的手臂撞掉桌上的物体。由于这些刚体是游戏驱动的，它们不会避免穿插进物理世界中的固定物体中，因此确保角色动作可信性的责任便落在动画师身上。

为了在游戏世界中移动角色，多数游戏会使用球体或胶囊体投射，往欲移动的方向进行探测。碰撞会使用手工方式求解。这样可以做出许多很酷的事情，例如：

- 当角色以斜向角度跑往墙壁，可以令角色沿墙滑动。

[84] 译注：这里是指游戏性能瞬间降低，不能良好地模拟出此效果。

- 角色走到路缘时，可以“提起”角色，而不会被卡住。
- 避免角色走下路缘时令角色进入“掉下”状态。
- 避免角色走上太陡的斜坡（多数游戏都会设置一个界限角度，角色在超过该角度的斜坡上会滑下来，不容许角色往上走）。
- 为迁就碰撞情况调整动画。

我们为最后一点给出一个例子，若角色以约90°角直冲向墙壁，我们可以令角色不停地对着墙“月球漫步（moonwalk）”，或是可以减慢该动画，或是可以做得更好，例如，播放一个双手伸出触碰墙壁的动画，然后令角色合理地停下来，直至他改变移动方向。

Havok提供一个角色控制系统处理这些事情。如图12.37所示，Havok系统中的角色以胶囊体phantom模拟，此phantom在每帧移动以求出新的位置。Havok会为每个角色维护一个碰撞接触流形（collision contact manifold，是指降噪后的撞触平面集合）。此流形可用于在每帧判断角色最优的移动、调整动画等。

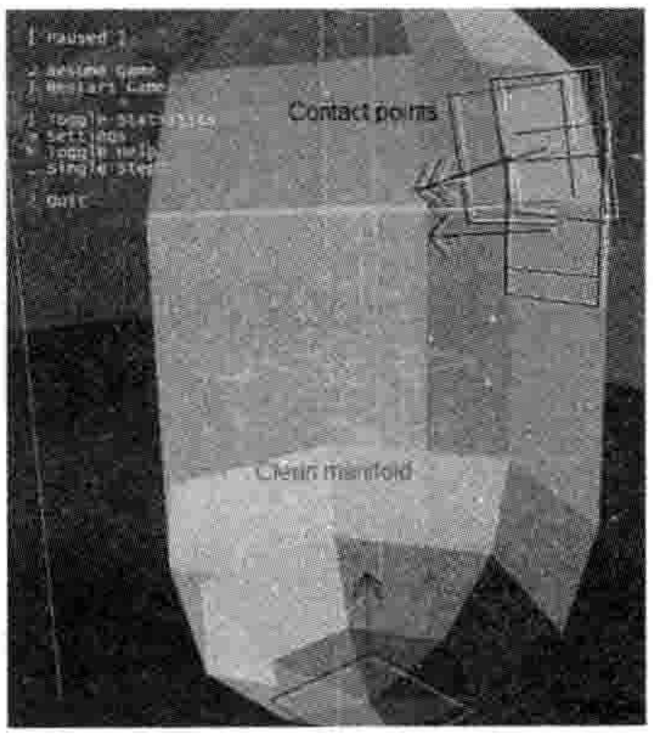


图 12.37: Havok的角色控制器把角色建模成胶囊体的phantom。phantom维护一个降噪的碰撞流形（一组平面），游戏可用此信息来做移动决策。

12.5.3.7 摄像机碰撞

许多游戏的摄像机会追随玩家在游戏世界中的角色或载具，玩家通常可以有限地操控或旋转摄像机。这类游戏中的一个要点，是令摄像机永不穿透场景中的几何体，否则会打破游戏的真实感。因此，在许多游戏中，摄像机系统成为碰撞引擎的重要使用方法。

对于大部分摄像机碰撞系统，其背后的基本原理是用一个或多个球形phantom包围虚拟摄像机，或是用球体投射查询检测是否和物体产生碰撞。在摄像机碰撞到物体之前，该系统会以一些方式调整摄像机的位置及/或定向，以避免摄像机穿透该物体。

说来简单，但实际上这是一个难以置信的棘手问题，需要花许多反覆试验才能得到满意的效果。为了让读者了解当中所需的工夫，可以告诉大家一个事实，许多游戏团队有一个专门的工程师，在项目整个周期中做摄像机系统的事情。我们不可能在此深入探讨摄像机碰撞检测及决议，但以下的内容应该能供读者了解当中最切题的要点。

- 拉近镜头（zoom in）在许多情况下可以避免碰撞问题。在第三人称游戏中，玩家可以拉近镜头成为第一人称视角，而不会产生太大的问题（但其中一个问题是要令拉近过程中摄像机不穿透角色的头部）。
- 在遇到碰撞时，大幅度改变摄像机的水平角度，并不是个好主意。因为这样会搞乱取决于摄像机角度的玩家操作。然而，当合乎玩家预期要进行的事情时，某程度的水平调整也能良好运作。例如，当玩家正在瞄准目标时，若因摄像机碰撞而失去目标，玩家必会破口大骂。但如果玩家只是在世界中移动，改变摄像机定向就自然不过了。
- 读者可以在某程度上调整摄像机的垂直角度，但最重要的是不要调整过度，否则玩家会失去方向感，最终令玩家向下看着角色的头顶!
- 有些游戏容许摄像机在一垂直平面上的弧上移动，该弧可能使用样条（spline）定义。这么做可以使单一HID控制（如左拇指摇杆的上下移动）直观地同时控制摄像机的缩放及垂直角度。（《神秘海域：德雷克船长的宝藏》的摄像机就是如此运作的。）当摄像机碰到世界中的物体，摄像机就可自动地沿此弧移动以避开碰撞，也可以把弧往水平方向挤压，或采用其他方法。
- 除了要考虑摄像机后方及旁边有什么，还必须考虑摄像机前方的事物。例如，若一个柱子或一个角色移动至玩家角色及摄像机之间，应如何处理？有些游戏会把造成问题的物体变成透明，有些游戏会拉近镜头或摆动摄像机避免碰撞。这么做可能会令玩家舒服或难受! 处理这些情况的方法，可能提高或降低游戏的体验品质。
- 读者或希望摄像机按不同的碰撞情况做不同反应。例如，当主角在非战斗的情况下，玩家或可接受用水平摆动摄像机的方式去避免碰撞。但当玩家正在向目标开火，水平或垂直的摄像机摆动会影响玩家瞄准，届时只可选择用拉近的方法。

就算考虑了这些及许多其他问题情况，你的摄像机或许仍会有问题! 在实现摄像机碰撞系统时，总是要预留大量的时间以备反复试错。

12.5.3.8 整合布娃娃

在12.4.8.7节中，我们学习了如何使用特殊类型的约束连接一组刚体以模仿瘫软（死亡或失去知觉）的人体行为。本节将会探讨在游戏中整合布娃娃时会遇到的一些问题。

如12.5.3.6节所述，对于有知觉的角色，其整体移动一般是通过在游戏世界做形状投射或移动phantom来达成的。而角色身体的细节动作则通常由动画驱动。另外，游戏驱动刚体有时候会绑定至四肢，借以做武器瞄准，或是令角色可以碰掉世界中的物体。

当角色失去知觉时，布娃娃系统就会介入。此时，可用胶囊体形状的刚体去模拟角色四肢，并把它们用约束连接至角色骨骼的关节。然后物理系统会模拟这些刚体的运动，并更新其对应的骨骼关节，令物理可以驱动角色的身体。

用作布娃娃物理的那组刚体，并不一定和角色有知觉时所绑定的一样。因为两个碰撞模型的需求有非常大的差异。当角色有知觉时，其刚体是游戏驱动的，因此我们不用在乎它们是否互相穿插。事实上，我们通常希望他们会重叠，以保证碰撞体之间没有空隙，使敌人的射击不会穿过身体。然而，当角色变成布娃娃，必须保证刚体间不会互相穿插，不然可能会令碰撞决议产生过大的冲量，导致四肢往外“爆炸”！因此，对于角色有知觉和失去知觉的状态，通常会有两套截然不同的碰撞/物理表示。

另一问题是，如何从有知觉状态过渡至失去知觉状态。简单用LERP把动画驱动和物理驱动的姿势做混合，其效果通常不太好，因为物理姿势会很快偏离动画姿势。（把两个完全不相关的姿势混合通常会不自然。）因此，我们或可考虑在过渡中使用富动力约束（见12.4.8.8节）。

当角色有知觉的时候（即他们的刚体是游戏驱动的），他们常会穿插入背景的几何体中。也即是说，当角色过渡至布娃娃（物理驱动）模式时，这些刚体可能会在其他实心物体之中。这样可能会产生巨大的冲量，令布娃娃的行为变得怪异。要避免这种问题，最理想是谨慎地制作死亡动画，尽量令角色的四肢远离碰撞。除此以外，在游戏驱动模式时，要使用phantom或碰撞回调检测角色的身体有没有与物体碰撞，没有碰撞的时候就可以安全地转换成布娃娃模式。

然而，即使采用了以上的手法，布娃娃还是有机会卡在其他物体中。要令布娃娃显得自然，可以利用单面碰撞功能。若四肢的一部分嵌入在墙中，单面碰撞会尽量推动四肢离开墙身，而不会使其保持卡着的状态。可是，单面碰撞也不能解决所有问题。例如，当角色快速移动，或是布娃娃的过渡不能正常执行，某个刚体最终可能会置于一面薄墙的另一面。那么角色可能就会悬挂在空中，而不能正常地掉到地上。

最近有一项流行的布娃娃功能，就是令失去知觉的角色重获知觉，并重新站起来。为了实现此功能，我们需要用某种方法搜寻合适的“站立”动画。我们希望能找到一个动画，其首帧的姿势是最匹配静止下来后的布娃娃姿势（通常后者是完全不能预计的）。我们可以只匹配几个关键的关节，例如大腿及上臂。另一个方法是使用富动力约束，手工地引导静止的布娃娃移动至适合站立的姿势。

最后要提一点，设定布娃娃的约束是一件很麻烦的工作。我们通常希望四肢能自由活动，但不会做成一些生物力学上不可能的姿势。所以建构布娃娃时通常会使用特别类型的约束。尽管如此，要令布娃娃好看还得费上不少心神。高质量的物理引擎（如Havok）提供丰富的内容创作工具，供美术人员在DCC工具（如Maya）中设置约束，并能即时测试布娃娃在游戏中的表现。

总而言之，令布娃娃物理在游戏中**运作**并不难，但要令它**好看**就是艰巨的工作！如同许多游戏编程的工作，最好能预留充足的时间反复试错，若读者首次做布娃娃就更加要注意。

12.6 展望：高级物理功能

在游戏中利用受约束的刚体动力学模拟，可创造非常多的物理驱动效果。但它的局限也是显而易见的。近年的研发不断寻求超越刚体动力学的物理引擎功能。以下是一些例子。

- **形变体**（deformable body）：随着硬件能力的提升，以及开发了更高效的算法，物理引擎开始提供对可变形体的支持。DMM引擎是在这方面极佳的例子。
- **布料**（cloth）：布料可建模为以刚性弹簧连接的一堆点质量。众所周知，布料模拟非常难以做好，因布料与其他物体的碰撞、模拟的数值稳定性等都会引致诸多问题。
- **头发**（hair）：头发可建模为大量物理模拟的细丝。更简单的方法是使用绳子或可变形体模拟角色头发。这是一个活跃的研究题目，而且游戏中的头发质量也一直提升。[85]
- **水面模拟**（water surface simulation）及**浮力**（buoyancy）：游戏中使用水面模拟及浮力已有一段日子。这些功能拟可以使用特设的系统（在物理引擎之外）达成，或是在物理模拟中加入力去建模。自然的水面运动通常只是一个渲染效果，不会影响物理模拟。从物理引擎的角度看，水面通常是建模为一个平面。水面大幅移动时，通常整个平面是跟着移动的。然而，有些游戏团队及研究学者在尝试打破这些极限，制作动态水面、浪尖、真实感的水流模拟等。
- **通用流体动力学模拟**（general fluid dynamics simulation）：现时流体动力学实现于专门的模拟库。然而，这也是一个活跃的研究题目，可能最终会以某种形式进入主流的物理引擎。[86]

---

[85]译注：对这方面有兴趣的读者，可参阅译者的博文“爱丽丝的发丝——《爱丽丝惊魂记：疯狂再临》制作点滴”http://www.cnblogs.com/miloyip/archive/2011/06/14/alice_madness_returns_hair.html

[86]译注：现时，流体动力学模拟已经常出现在游戏中，例如《小小大星球2》（见“Two Uses of Voxels in LittleBigPlanet2's Graphics Engine”，http://advances.realtimerendering.com/s2011/index.html）、《鳄鱼小顽皮爱洗澡》（译者估计此作品采用了smoothed particle hydrodynamics/SPH方法做流体模拟）。

第四部分

游戏性

第13章　游戏性系统简介

迄今本书主要集中谈技术。我们了解到游戏引擎是复杂、多层的软件系统，建于目标机器的硬件、驱动及操作系统之上。我们已得知，低阶引擎系统如何满足引擎其他部分的需求，人类玩家如何使用人体学接口设备（手柄、键盘、鼠标及其他设备）向引擎提供输入信息，渲染引擎如何产生屏幕上的三维图形，碰撞系统如何检测及解决各形状之间的穿插，物理模拟如何令物体以物理上真实的方式移动，动画系统如何令角色及物体自然地移动。尽管这些组件提供了如此广泛强大的功能，若不把它们整合在一起，那仍然不是一个游戏!

游戏的本质，并非在于其使用的技术，乃是其**游戏性**（gameplay）。所谓游戏性，可定义为玩游戏的整体体验。**游戏机制**（game mechanics）一词，把游戏性这个概念变得更为具体。游戏机制通常定义为一些**规则**，这些规则主宰了游戏中多个实体之间的互动。游戏机制也定义了玩家（们）的**目标**（objective）、成败的**准则**（criteria）、玩家角色的各种**能力**（ability）、游戏虚拟世界中**非玩家实体**（non-player entity）的数量及类型，以及游戏体验的**整体流程**（overall flow）。在许多游戏中，扣人心弦的故事和丰富的角色，与这些游戏机制元素交织在一起。然而，并非所有游戏都必须有故事及角色，从极为成功的解谜游戏如《俄罗斯方块（Tetris）》可见一斑。谢菲尔德大学（University of Sheffield）的Ahmed BinSubaih、Steve Maddock及Daniela Romano曾发表了一篇论文，题目为《“游戏”可移植性研究（A Survey of “Game” Portability）》[1]，文中把实现游戏性的软件系统集合称为游戏的**G因子**（G-factor）。在以下3章中，我们会探讨用于定义及管理**游戏机制**（或称游戏性，或称G因子）的关键工具及引擎系统。

13.1　剖析游戏世界

对于不同的游戏类型、林林总总的游戏，游戏性的设计有很大差异。然而，大多数的三维游戏，以及不少的二维游戏，都会有几个基本的结构模式。以下几节会讨论这些模式，但

[1] http://www.dcs.shef.ac.uk/intranet/research/public/resmes/CS0705.pdf

请记得这些模式有其局限，并非所有游戏都能套用。

13.1.1 世界元素

多数电子游戏都会在二维或三维虚拟**游戏世界**（game world）中进行。这些世界通常是由多个离散的元素所构成的。一般来说，这些元素可分为两类——静态元素及动态元素。静态元素包括地形、建筑物、道路、桥梁，以及几乎任何不会动或不会主动与游戏性互动的物体。而动态元素则包括角色、车辆、武器、补血包、能力提升包、可收集物品、粒子发射器、动态光源、用来检测游戏中重要事件的隐形区域、定义物体移动路径的曲线样条等。图13.1展示了以这种方式分拆的游戏世界。

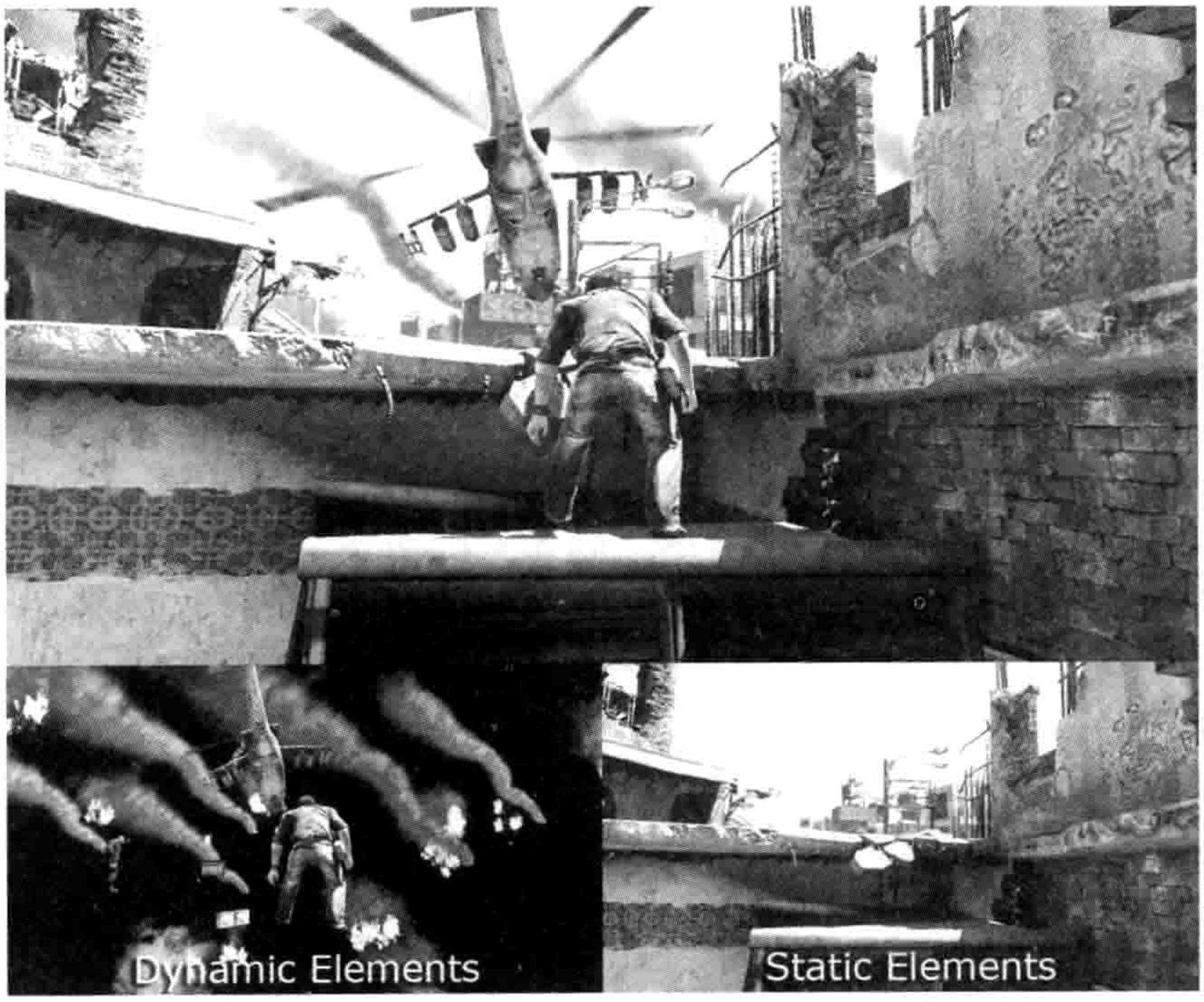


图 13.1: 典型的游戏世界是由静态和动态元素所组成的。

谈到游戏性，我们通常会集中讨论游戏的动态元素。但是显然，静态的背景对于游戏有重大影响。例如，对于着重掩护的射击游戏，若在空旷的长方形房间里进行，就毫无乐趣可言了。另一方面，实现游戏性的软件系统，其主要关心的是动态元素的位置、定向、内部状态更新，因为这些是随时间改变的元素。**游戏状态**（game state）一词，是指所有动态游戏世界元素的当前整体状态。

动态和静态元素之比，在各游戏中有所不同。多数三维游戏只有相对少量的动态元素，这些元素在相对广大的静态背景范围中移动。另一些游戏，如经典的街机游戏《爆破彗星（Asteroids）》或Xbox 360上的复古热作《几何战争（Geometry Wars）》，就完全没有静态元素可言（除了空白的屏幕）。通常，游戏的动态元素比静态元素更耗CPU资源，因此多数三维游戏被迫使用有限的动态元素。然而，动态元素的比例越高，玩家感受到的世界越“生动”。随着游戏硬件性能的进步，游戏的动态元素比例也在不断提升。

有一点要留意，游戏世界的动态及静态元素时常并非黑白分明。例如，在街机游戏《迅雷赛艇（Hydro Thunder）》中，瀑布的纹理有动画效果，其底下有薄雾效果，而且游戏设计师可以独立于地形及水体外随意放置这些瀑布，在这个意义上这些瀑布是动态的。然而，从工程的角度看，瀑布是以静态元素方式处理的，因为它们并不会以任何形式与赛艇互动（除了会阻碍玩家看到加速包及秘密通道）。各游戏引擎会以不同基准区分静态和动态元素，有些引擎甚至不做区分（即所有东西都可能成为动态元素）。

分开静态与动态元素的目的，主要是做优化之用——若物体的状态不变，我们就可以减少对它的处理。例如，静态三角形网格的顶点可使用世界空间坐标，借以省去对每顶点的矩阵乘法[2]，而正常渲染时是需要用矩阵乘法把模型空间变换为世界空间的。光照也可以预计算，其结果可存于顶点、光照贴图、阴影贴图、静态环境遮挡（ambient occlusion）信息，或预计算辐射传输（precomputed radiance transfer，PRT）的球谐系数（spherical harmonics coefficient）[3]。在运行时游戏世界中动态元素所需的运算，对于静态元素来说，都可以预先计算或忽略。

有一些游戏含有可破坏环境，这算是模糊静态和动态元素之分界的例子。例如，我们可能给予每个静态元素3个版本，完好的，受损的，完全被毁坏的。这些背景元素在大部分时间中是静态的，但在爆炸中可能被替换至不同版本，以产生其受到破坏的视觉效果。实际上，静态和动态世界元素只是许多可能性的两个极端。我们为两者定分界（如果真的这么做），只是用作改变优化方法及跟随游戏设计所需。

13.1.1.1 静态几何体

静态世界元素通常在Maya之类的工具中制作。这些元素可能是一个巨形的三角形网格，或是分拆为多个细块。场景中的静态部分有时候会采用**实例化几何体**（instanced geometry）制作。实例化是一个节省内存的技术，当中，较少数目的三角形网格会在不同位置及定向被渲染多次，以产生一个丰富的游戏场景。例如，三维建模师可能制作了5款矮墙，然后以随

[2]译注：一般来说，每顶点的世界空间坐标也须乘以view-projection矩阵，所以预先计算出world-view-projection矩阵去给顶点相乘并不是问题。这里可能假设模型空间坐标需要独立乘以world矩阵，用于世界空间的光照或其他计算。

[3]译注：见10.3.3.6节。

机方式把它们拼砌成数里长、独一无异的城墙。

静态视觉元素及碰撞数据也可以用**笔刷几何图形**（brush geometry）方式构建。这种几何体源自于雷神之锤（Quake）系列引擎。所谓**笔刷**，是指多个凸体积组成的形状，每个凸体积由一组平面所包围。建构笔刷几何图形是容易快捷的，而且这种几何体能很好地整合至基于BSP树的渲染引擎。笔刷非常合适于快速堆砌游戏内容的初形。由于这么做成本不高，可以在初始阶段就测试游戏性。如果证实了关卡的布局恰当，美术团队便可以加入纹理及微调那些笔刷几何图形，或是用更细致的网格资源取代它们。相反，若关卡需要重新设计，那些笔刷几何图形可以简单地修改，而无须美术团队大量重做资源。

13.1.2 世界组块

当游戏在非常巨大的虚拟世界中进行，这些世界通常会被拆分成为独立可玩的区域，我们称之为**世界组块**（world chunk）。有时候组块也称为**关卡**（level）、**地图**（map）、**舞台**（stage）或**地区**（area）。玩家在进行游戏时，通常同时只能见到一个，或最多几个组块。随着游戏的发展，玩家从一个组块进入另一个组块。

起初，发明“关卡”的概念是为了在内存有限的游戏硬件上提供更多游戏性的变化。同时间只会有一个关卡存于内存，但随着玩家从一个关卡到达另一个关卡，可以获得更丰富的整体体验。从那时候开始，游戏设计形成多个分支，到现在这种基于线性关卡的游戏少了很多。有些游戏实质上仍然是线性的，但对玩家来说，世界组块之间已没像以前那般地明显分界。另一些游戏使用星状拓扑（star topology），其中玩家在一个中央枢纽地区，并可以在那里选择前往其他的地区（可能需要先为那些地区解锁）。还有一些游戏使用图状拓扑，即地区之间以随意方式连接。也有一些游戏会提供一个貌似广大、开放的世界。

无论现代游戏设计如何丰富，除了最小型的游戏世界，多数游戏世界都仍然会分割为某形式的组块。这么做有几个原因。首先，内存限制仍然是一个重要的约束（直至有无限内存的游戏机充斥市面）。世界组块也是一个控制游戏整体流程的方便机制。组块作为一个分工的单位，每个组块可以由较小的游戏设计师及美术团队分别建构及管理。图13.2展示了一些世界组块。

13.1.3 高级游戏流程

游戏的**高级流程**（high-level flow）是指由玩家**目标**（objective）所组成的序列、树或图。目标有时候也称作**任务**（task）、**舞台**（stage）或**关卡**（level）（此术语和世界组块相同），又或是**波**（wave）（若游戏的主要目标是击败一波接一波敌人）。高级流程也会定义每

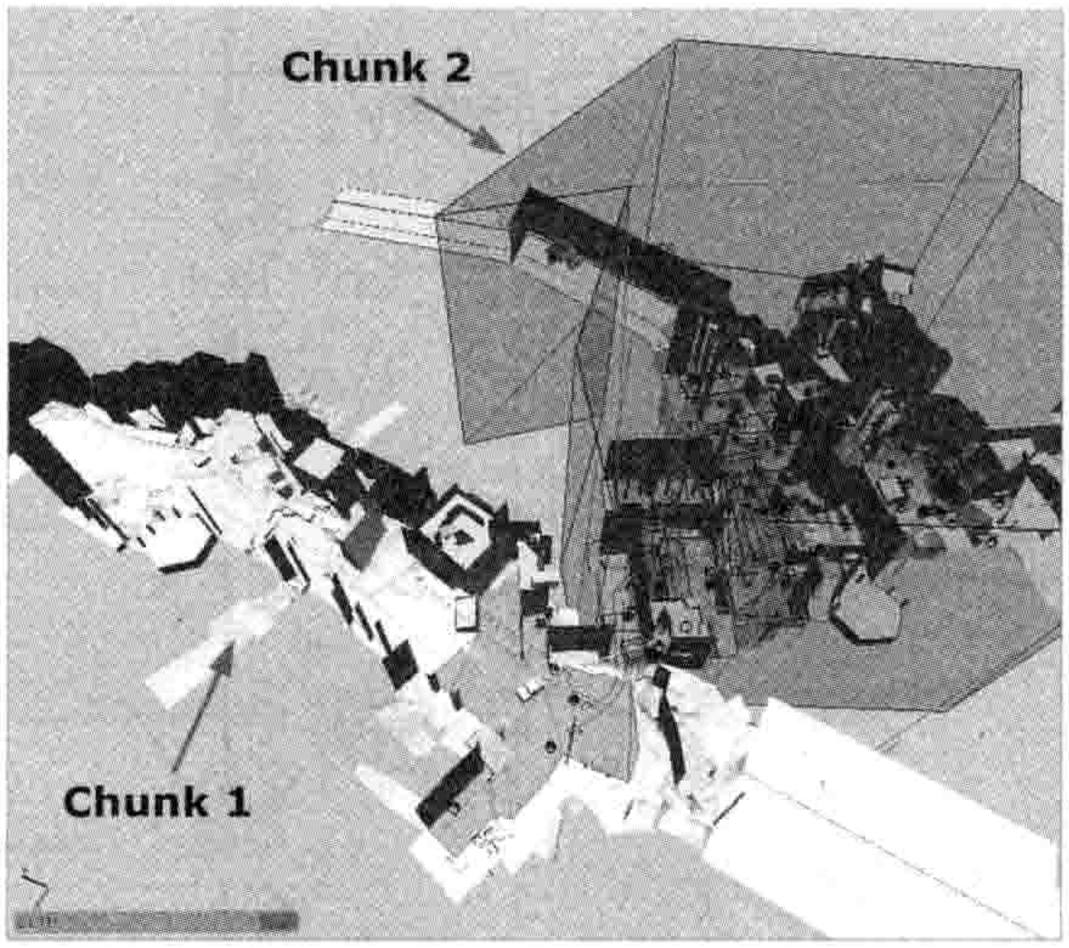


图 13.2: 基于各种原因，许多游戏世界会被分割成组块。这些原因包括内存限制、需要通过游戏世界控制游戏的流程，以至于在开发期做分工之用。

个目标的胜利条件（如肃清所有敌人并取得钥匙），以及失败的惩罚（如回到当前地区的起点，当中可能会扣减一条“生命”）。在故事驱动的游戏中，流程可能也包含多个游戏内置电影，使玩家得知故事的进展。这些连续镜头段有时候称为**过场动画**（cut-scene）、**游戏内置电影**（in-game cinematics，IGC）或**非交互连续镜头**（noninterative sequence，NIS）。若这些镜头是在脱机时渲染的，然后以全屏电影方式播放，则会称之为**全动视频**（full-motion video，FMV）。

早期游戏中，玩家的目标会一一对应至某个世界组块（也因此“关卡”一词具双重含意）。例如，在《大金刚（Donkey Kong）》中，每个关卡给予玛里奥一个新的目标（即走到天台达至下一关）。然而，这种目标和组块一一对应的关系在现代游戏设计中已式微。每个目标可能与一个或多个世界组块有所关联，但目标和组块的耦合会被刻意减弱。这种设计提供弹性，可以独立地改动游戏的目标及世界组块，这样从游戏开发的后勤及实践角度上来说都是极为有用的。许多游戏把目标归类为更粗略的游戏性段落，例如称之为**章**（chapter）或**幕**（act）。图13.3展示了一个典型游戏性的架构。

13.2 实现动态元素：游戏对象

游戏的动态元素通常会以面向对象方式设计。此方式不但直观自然，而且能很好地对应至游戏设计师建构世界的概念。游戏设计师能想象出游戏中的角色、载具、悬浮血包、爆炸

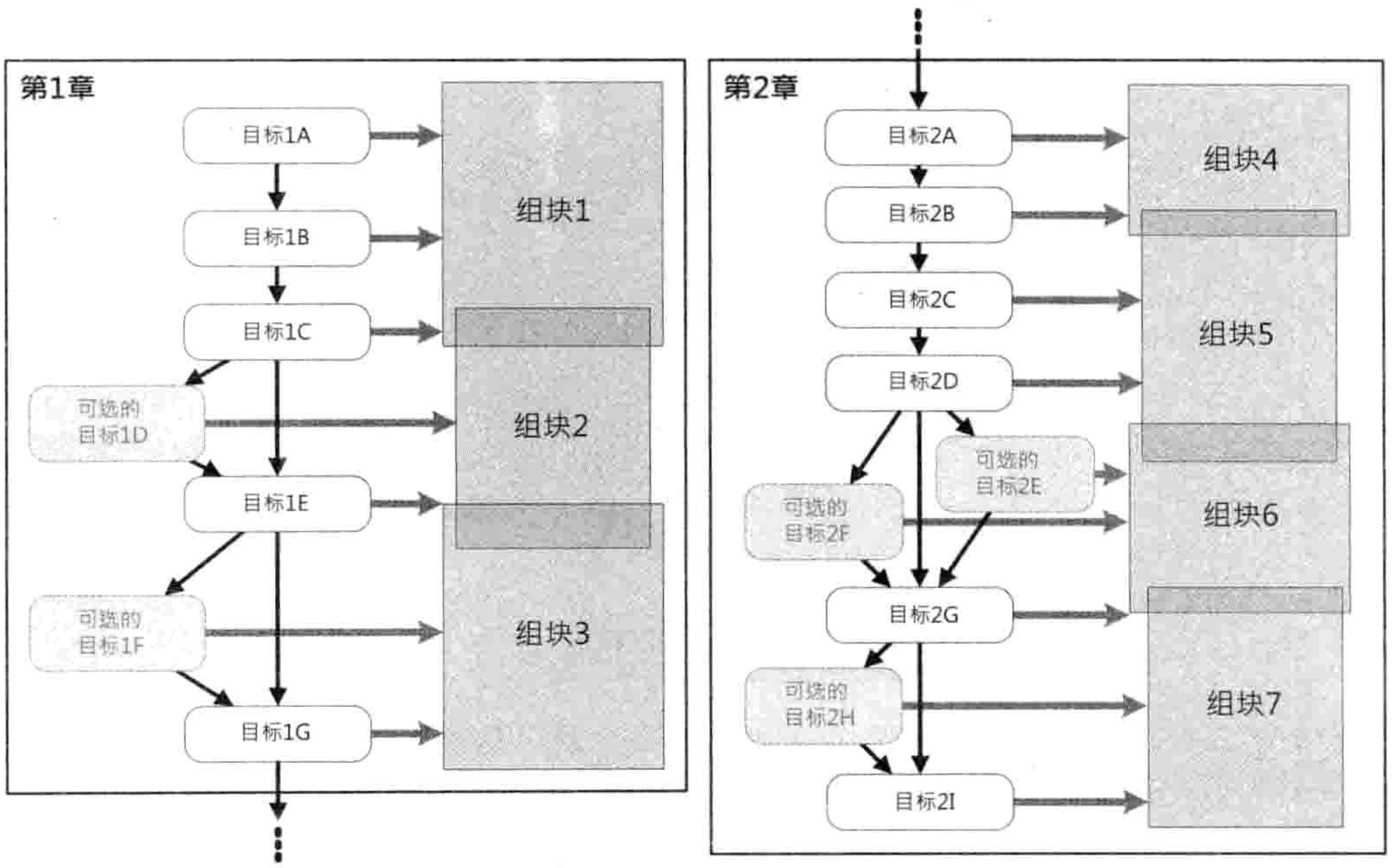


图 13.3: 游戏目标通常设置为序列、树和图，每个目标映射至一个或多个游戏组块。

木桶，以及无数的动态对象在游戏世界中移动。因此，很自然会想到在游戏世界编辑器中创建及处理这些元素。相似地，程序员通常也会觉得，把动态元素实现为运行时的自动代理人是十分自然的事情。本书会使用**游戏对象**（game object，GO）这一术语，去描述游戏世界中几乎任何的动态元素。然而，此术语在业界并非标准，有时候也称作**实体**（entity）、**演员**（actor）或**代理人**（agent）等。

如面向对象的习惯，游戏对象本质上是**属性**（attribute，对象当前的状态）及**行为**（behavior，状态如何应对事件、随事件变化）的集合。游戏对象通常以类型（type）做分类。不同类型的对象有不同的属性及行为。某类型的所有**实例**（instance）都共享相同的属性及行为，但每个实例的属性的**值**（value）可以不相同。（注意，若游戏对象的行为是数据驱动的，例如，用脚本代码，或由一组数据驱动的规则回应事件，那么行为也可以按实例有所差异。）

类型和**实例**的分别是十分关键的。例如，《吃豆人（Pac-Man）》中有4个游戏对象类型：鬼魂、豆子、大力丸和吃豆人。然而，在某时刻，只会最多有4个鬼魂实例、50～100个豆子实例、4个大力丸实例和1个吃豆人的实例。

多数面向对象系统都会提供一些机制实现属性、行为，或两者的**继承**（inheritance）。继承促进重用代码及设计。使用继承的具体形式，每个游戏都不太一样，但多数游戏引擎都

会支持某些形式的继承功能。

13.2.1 游戏对象模型

在计算机科学中，**对象模型**（object model）一词有两个相关但不一样的意思。它可以是指某编程语言或形式设计语言所提供的特性集。例如，我们可以说**C++对象模型**或**OMT对象模型**[4]。对象模型的另一个意义是指，某面向对象编程接口（如类和方法的集合，以及为解决特定问题所设计的相互关系）。这个意义的一个例子是**微软Excel对象模型**，此模型供外在程序以多种方式控制Excel。（见维基百科对**对象模型**[5]的更多讨论。）

本书中，**游戏对象模型**（game object model）一词专指由游戏引擎所提供的、为虚拟世界中动态实体建模及模拟的设施。按此意义，游戏对象模型含有前面所及的两方面定义。

- 游戏的对象模型是一种特定的面向对象编程接口，用于解决开发某个游戏中一些具体实体的个别模拟问题。
- 此外，游戏的对象模型常会扩展编写引擎本身的编程语言。若游戏是以非面向对象语言（如C）实现的，程序员可自行加入面向对象的设施。即使游戏是以面向对象语言（如C++）实现的，通常也会加入一些高级功能，例如反射（reflection）、持久性（persistence）及网络复制（network replication）等。游戏对象模型有时候会融合多个语言的功能。例如，某游戏引擎可能会合并编译式语言（如C/C++）和脚本语言（如Python、Lua或Pawn）来使用，并提供统一的对象模型供这两类语言访问。

13.2.2 工具方的设计和运行时的设计

以世界编辑器（以下详述）呈现给设计师的对象模型，不必和用于实现运行时游戏的对象模型相同。

- 工具方的游戏对象模型，当要实现为运行时的模型时，可以使用无原生面向对象功能的语言（如C）。
- 工具方的某单个游戏对象，在运行时可能被实现为一组类（而非预期的一个类）。
- 每个工具方的游戏对象，在运行时可能仅是唯一标识符，其全部状态则储存至多个表或一组松耦合的对象。

---

[4]译注：OMT（Object Modeling Technique）是UML（Unified Modeling Language）的3个前身之一，另外两个是Booch方法及OOSE（Object Oriented Software Engineering）方法。这3个方法论的发明者James Rumbaugh（OMT）、Grady Booch及Ivar Jacobson（OOSE）被专称为UML三巨头。

[5]http://en.wikipedia.org/wiki/Object_model

因此，一个游戏实在是有两个虽不同但密切相关的对象模型。

- **工具方对象模型**（tool-side object model）是一组设计师在世界编辑器里看到的游戏对象类型。
- **运行时对象模型**（runtime object model）是程序员用任何语言构成成分或软件系统把工具方对象模型实现于运行时的对象模型。运行时对象模型可能和工具方模型相同，或有直接映射，又或是完全不同的实现。

有些游戏引擎对两种模型并没有很清晰的分界，甚至没有分别。其他游戏引擎则会清楚地划定分界。在一些引擎中，工具和运行时会共享游戏对象模型的实现。其他引擎中，运行时的游戏对象模型看上去完全和工具方的实现相异。有些模型的实现会偏重于工具方，游戏设计师需要知悉他们所设计的游戏性规则和对象行为对性能和内存消耗的影响。然而，几乎所有游戏引擎都会有某形式的工具方对象模型及对应的运行时实现。

13.3 数据驱动游戏引擎

在游戏开发的早期年代，游戏的大部分内容都是由程序员硬编码而成的。就算有工具，也都是非常简陋的。这样之所以行得通，是因为当时典型的游戏只有少量内容，而且当时游戏的标准并不高，部分能归咎于早期游戏硬件对图形及声音性能的限制。

今天，游戏的复杂性以数量级增长，而且品质要求很高，甚至经常要和好莱坞大片的计算机特效比较。游戏团队也变大许多，但游戏内容量比团队增长得更快。把这一代游戏机（Wii、Xbox 360、PS3）的游戏对比上一代，游戏团队需要产出约10倍的内容，但团队最多只增加了25%。此趋势意味着，团队必须以极高效的方式生产非常大量的内容。

工程方面的人力资源通常是制作的瓶颈，因为优秀的工程师非常有限和昂贵，而且工程师产出内容的速度通常比美术设计师及游戏设计师慢（源于计算机编程的复杂性）。现在多数团队相信，应该尽量把生产内容的权力交予负责该内容的制作者之手——即美术设计师和游戏设计师。当游戏的行为可以全部或部分由美术设计师及游戏设计师所提供的**数据**所控制，而不是由程序员所编写的**软件**完全控制，该引擎就称为是**数据驱动**（data-driven）的。

通过发挥所有员工的全部潜能，并为工程团队工作降温，数据驱动架构因而能改善团队的效率。数据驱动也可以促进**迭代次数**。当开发者想要微调游戏的内容或完全重制整个关卡时，数据驱动的设计能令开发者迅速看到改动的效果，理想的情况下无须或仅需工程师的少量帮助。这样能节省宝贵的时间，并促使团队把游戏打磨至最高品质。

然而必须注意到，数据驱动通常有较大的代价。我们必须为游戏设计师及美术设计师提供工具，以使用数据驱动的方式制作游戏内容。也必须更改运行时代码，以健壮地处理更大的输入范围。在游戏内也要提供工具，让美术设计师及游戏设计师预览工作成果及解决问题。这些软件都需要花大量时间及精力去编写、测试及维护。

可惜，许多团队匆忙地采用数据驱动架构，而没有静心下来研究这项工作对他们的游戏设计，甚至团队成员个别需求的影响。这种急进的方式，使他们有时候会走得太过火，制作出过于复杂的工具及引擎系统，这些软件可能难以使用、臭虫满载，并且几乎无法适应项目的需求变动。讽刺的是，为了实现数据驱动设计的好处，团队很容易变得比老式硬编码方式生产力更低。

每个游戏引擎都应该有些数据驱动的部件，但是游戏团队必须非常谨慎地选择把哪些引擎部分设为数据驱动的。我们需要衡量制作数据驱动或迅速迭代功能的成本，对比该功能预期可以节省团队在整个项目过程的时间。在设计及实现数据驱动的工具和引擎时，要牢记KISS咒语（“Keep it simple，stupid”）。改述爱因斯坦名言：游戏引擎中的一切应尽量简单，至不能再简化为止。

13.4 游戏世界编辑器

我们曾讨论过数据驱动的资产创作工具，例如Maya、Photoshop、Havok内容工具等。这些工具产生的资产，会供渲染引擎、动画系统、音频系统、物理系统等使用。对游戏性内容来说，对应的工具便是**游戏世界编辑器**（game world editor），这些编辑器用于定义世界组块，并填入静态及动态元素。

所有商用游戏引擎都有某种形式的世界编辑工具。当中闻名于世的有**Radiant**，它是用来制作雷神之锤和毁灭战士引擎系列的地图，见图13.4。Valve公司的Source引擎（即《半条命2（Half Life 2）》、《橙盒（The Orange Box）》、《军团要塞2（Team Fortress 2）》所使用的引擎）也提供了一个名为**Hammer**的编辑器（曾命名作**Worldcraft**和**The Forge**），见图13.5。

游戏世界编辑器通常可以设置游戏对象的初始状态（即其属性值）。多数游戏世界编辑器也会以某种形式，让用户控制游戏世界中动态对象的**行为**。控制行为的方式可以是通过修改数据驱动的组态参数（例如，对象A最初应是隐形状态，对象B在诞生后应立即攻击玩家，对象C是可燃的），又或是使用脚本语言，从而让游戏设计师的工作进入编程境界。有些世界编辑器甚至能定义全新的游戏对象类型，过程无须或只需少许程序员介入。

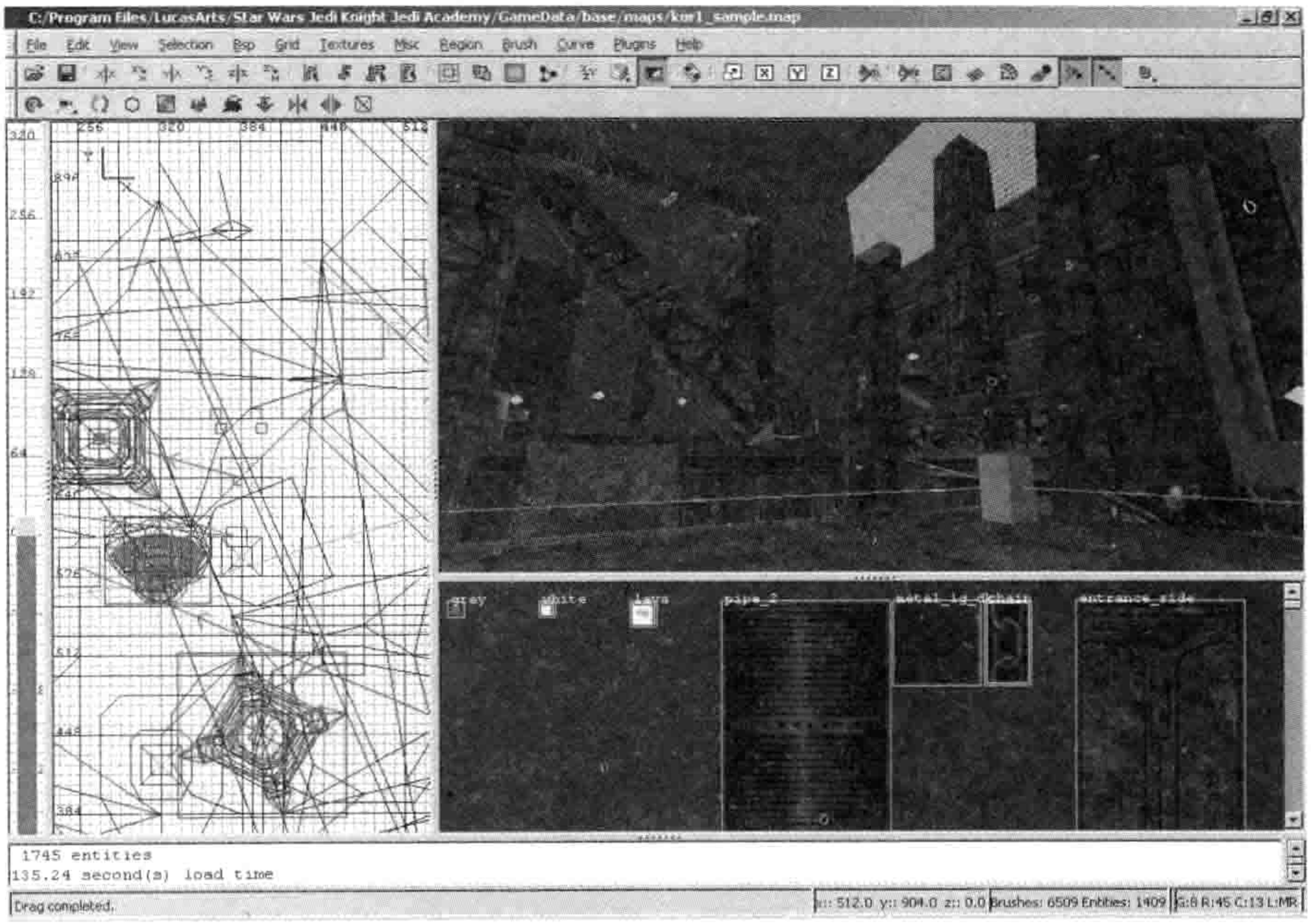


图 13.4: 为《雷神之锤》和《毁灭战士》系列引擎而设的Radiant世界编辑器。

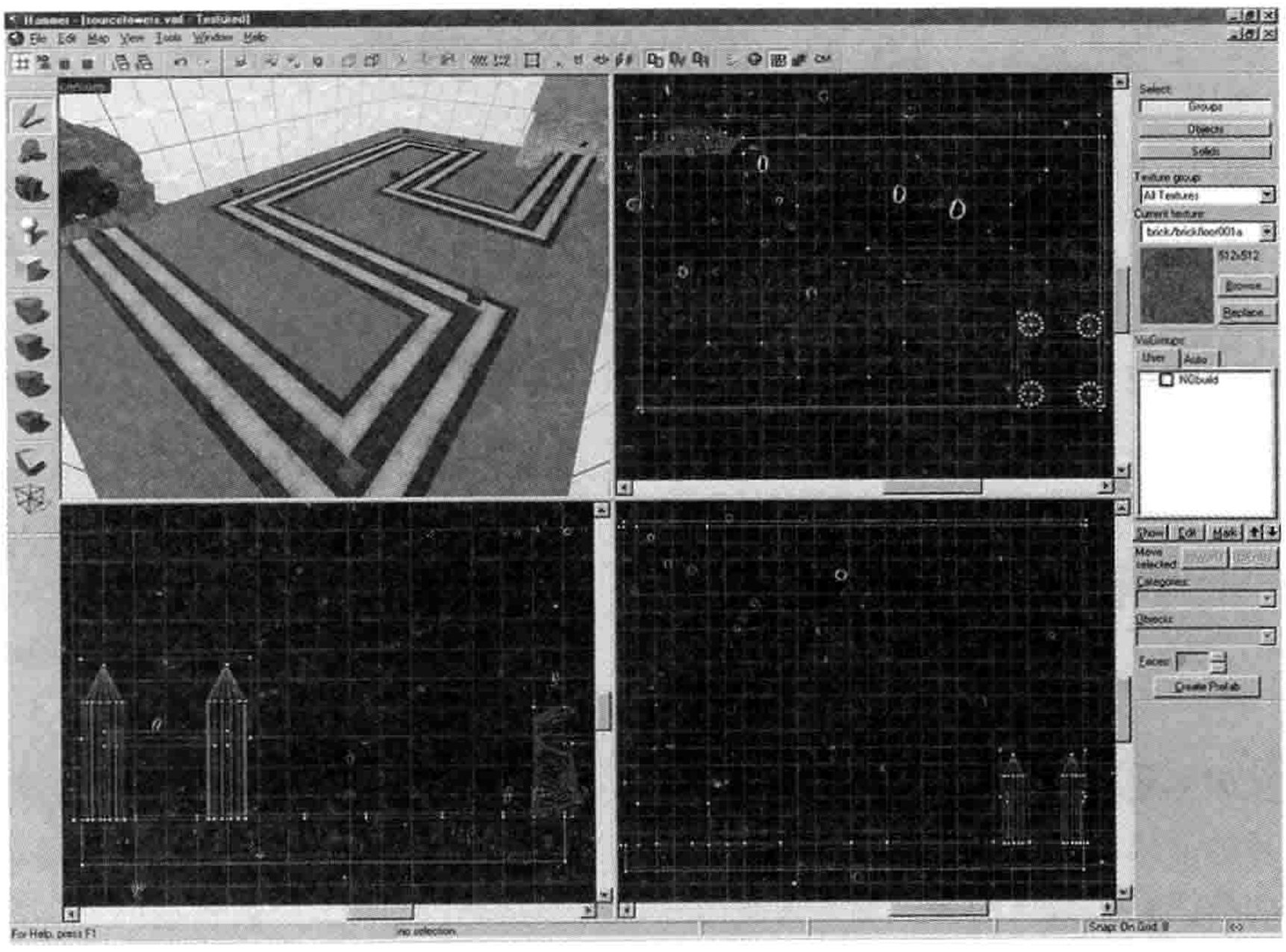


图 13.5: Valve公司为Source引擎而设的Hammer编辑器。

13.4.1 游戏世界编辑器的典型功能

各个游戏世界编辑器的设计及布局有很大差异，但大部分都会提供一组相当标准的功能集。这些功能包括但不限于以下之列。

13.4.1.1 世界组块创建及管理

世界创建的单位通常是组块（chunk，或称为关卡/level或地图/map，见13.1.2节）。游戏世界编辑器通常可以创建多个新的组块，以及把现有组块更名、分割、合并及删除。每个组块可以连接至一个或多个静态网格，以及其他静态数据元素，例如人工智能用的导航地图、玩家可攀抓边缘信息、掩护点等。有些引擎的组块必须以一个背景网格来定义，不能缺少。而另一些引擎则可以独立存在，或许是用一个包围体（如AABB、OBB或任意多边形区域）来定义，并可填入零至多个网格及/或笔刷几何（见1.7.3.1节）。

有些世界编辑器提供专门的工具制作地形、水体，以及其他专门的静态元素。在另一些引擎中，这些元素可能都是用标准的DCC应用程序来制作的，但会以某种方式加入标签，以对资产调节管道及/或运行时引擎说明它们是特别的元素。（例如，在《神秘海域：德雷克船长的宝藏》中，水体是以普通三角形网格方式制作的，但会贴上特殊的材质，以说明它们应以水体方式处理。）有时候，我们会使用另一独立工具来创建及编辑特殊的世界元素。例如《荣誉勋章：血战太平洋》的高度场地形，其制作工具便是来自艺电另一团队的自定义化版本。由于项目当时使用了Radiant引擎，比起在Radiant中集成一个地形编辑器，这么做更为合适。

13.4.1.2 可视化游戏世界

世界编辑器把游戏世界的内容可视化（visualize），对用户来说是很重要的功能。因此，几乎所有游戏编辑器都提供世界的三维透视视角，及/或二维的正射视角。很常见的方式是把视图面板分割为4部分，3个用作上、侧、前方的正射正视图（orthographic elevation），另一个用作三维透视视图。

有些编辑器直接整合自制的渲染引擎至工具中，去提供这些世界视图。另一些编辑器则是把自身整合至三维软件，如Maya或3ds Max，因而可以简单地利用这些工具的视区。也有些编辑器的设计，会通过与实际的游戏引擎通信，利用游戏引擎来渲染三维视图。更甚者，有些引擎会整合至引擎本身。

13.4.1.3 导航

若用户不能在世界编辑器的世界中到处移动，这个编辑器显然无所用。在正射视图中，必须能够滚动及缩小放大。而三维视图则可使用数个摄像机控制方式。例如可以聚焦某个对象，然后绕它旋转。也可以切换至“飞行”模式，当中，摄像机以自身的焦点旋转，并可向前后上下左右移动。

有些编辑器提供许多方便导航的功能，包括用单个按键就可以选取及聚焦对象、储存多个相关的摄像机位置、在那些位置中跳转、多个摄像机移动速率模式、如网页浏览器的导航历史记录般在游戏世界中跳转等。

13.4.1.4 选取

游戏世界编辑器的主要设计目的是，供用户利用静态及动态元素填充游戏世界。因此，让用户选择个别元素来编辑，是很重要的功能。有些引擎只容许同时间选取一个对象，而更先进的编辑器则可以多选。用户可以使用方形橡皮筋在正射视图中选取对象，或在三维视图中用光线投射方式进行选取。多数编辑器也会以滚动表或树视图展示世界中的元素列表。有些编辑器也可以把选取集命名及储存，供以后取回使用。

游戏世界通常填充了很密集的内容，因而有时候可能难以选取心目中的对象。此问题有几个解决方法。当使用光线投射方式选取三维中的对象时，编辑器可让用户循环选取与光线相交的所有对象，而不是总选取最近者。许多编辑器可以在视图中暂时隐藏当前所选的对象。那么，若用户选不到所需的对象，可以先把选取的对象隐藏再试。下节所述的图层，也是有效疏理对象并改善选取成功率的功能。

13.4.1.5 图层

在一些编辑器中，可以把对象用预设或用户自定义的**图层**来分组。此功能非常有用，能把游戏世界中的内容有条理地组织起来。可以把整个图层隐藏或显示整理凌乱的屏幕内容。也可以把图层设置色彩，令图层内容更易识别。图层也是分工的重要工具，例如，负责灯光的同事在某个世界组块上工作时，他们可以隐藏所有和灯光无关的元素。

更重要的是，若编辑器能独立地载入及储存图层，就能避免多人在同一个世界组块上工作所产生的冲突。例如，所有光源可能储存在一个图层里，背景几何体在另一图层，所有AI角色又置于另一图层。由于每个图层完全独立，灯光、背景及NPC小组可以同时在同一世界组块上工作。

13.4.1.6 属性网格

填充游戏世界组块的静态和动态元素，通常会有多个能让用户编辑的属性（property，也称作attribute）。属性可以是简单的键值对，并仅限使用简单的原子数据类型，如布尔、整数、浮点数及字符串。有些编辑器支持更复杂的属性，包括数组、嵌套的复合数据结构。

多数世界编辑器会使用能滚动的属性网格（property grid）显示当前选取（单个或多个）对象的属性，如图13.6所示。用户能在网格中看到每个属性现时的值，还可以用键入方式、复选框、下拉组合框、微调控制项（spinner control）等编辑属性的值。

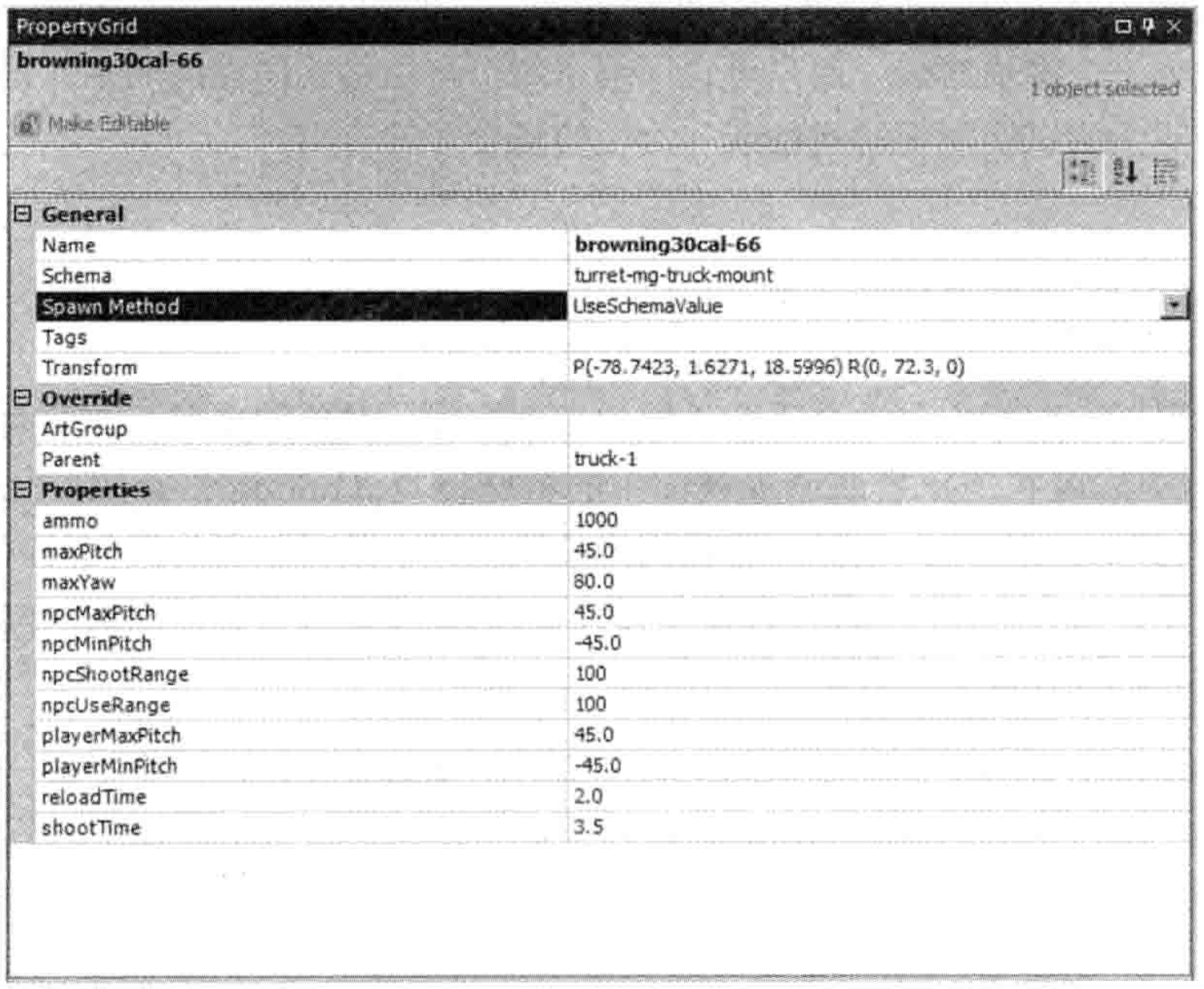


图 13.6: 典型的属性网格编辑器。

选取多个对象后的编辑方式

在支持多对象选取的世界编辑器中，其属性编辑器也可支持多对象编辑。此高级特性会把选取中所有对象的属性混合在一起来显示。若某个属性在选取中所有对象都相同，就会正常地显示该值，在网格中编辑该值时，也会把新值更新至所有选取对象的属性中。若某属性在各个对象中并不一致，则网格不予以显示。在这种情况下，若在网格中输入新值，它便会覆盖所有选取对象的值，令它们变成一致。还有一个问题，那就是如何处理选取中不同类型的对象。每个对象可能会有不同的属性集，因此属性网格必须仅显示所有选取对象共通的属

性。由于游戏对象经常会继承自一个基类，所以这种做法还是有用的。例如，多数对象含位置及定向这两个属性，当选取不同类型的对象时，虽然编辑器会暂时隐藏特化的属性，但用户仍可以编辑这些共有的属性。

自由格式属性

通常，这些属性集关联至某个对象，属性的数据类型是按每个对象来定义的。例如，渲染对象含有位置、定向、比例及网格的属性，光源含有位置、方向、颜色、强度及光源类型属性。有些编辑器还会额外让用户在个别对象上定义“自由格式（free-form）”属性。这些属性通常实现为一个键值对表。用户可自由编辑每个自由格式属性的名字（键）、数据类型，以及其值。这对于尝试实现新游戏性或一次性的场合非常有用。

13.4.1.7 安放对象及对齐辅助工具

世界编辑器对一些对象属性会采取不同的处理方式。对象的位置、定向及缩放通常如同在Maya和Max中，可利用正射或透视视图中的特殊锚点（handle）操控。此外，资产的连接通常需要用特殊方式处理。例如，若我们修改了世界中某对象所使用到的网格，编辑器应该在正射及三维透视视区中显示该网格。因此，游戏世界编辑器必须知悉这些属性需特殊处理，而不能把它们当作其他属性般统一处理。

许多世界编辑器会除了提供基本的平移、旋转、缩放工具外，还会提供一篮子的对象安放及对齐辅助工具。这些功能中，大部分都借鉴自商用图形及三维建模工具，如Photoshop、Maya、Visio等。这些功能的例子有对齐（snap）至网格、对齐至地形、对齐至对象等。

13.4.1.8 特殊对象类型

如同世界编辑器对于一些属性需要特殊处理，某些对象类型也需要特殊处理。例如：

- **光源**（light source）：世界编辑器通常使用特殊的图标来表示光源，因为它们本身并无网格。编辑器可能会尝试显示光源对场景中几何体的近似效果，令设计师可以实时移动光源并能看到场景最终效果的大概。
- **粒子发射器**（particle emitter）：如果编辑器是建立在独立的渲染引擎之上的，那么在编辑器中可视化粒子的发射器也可能会遇到问题。在此情况下，粒子发射器可简单地用图标显示，或是在编辑器中尝试模拟粒子效果。当然，若编辑器是置于游戏内的，或是能与运行中的游戏通信的，这便不成问题。

- **区域**（region）：区域是空间中的体积，供游戏侦测相关事件，诸如对象进入或离开体积，或是就某些目的做分区。有些游戏引擎限制了区域，只能为球体或定向盒，而另一些引擎可能支持一些形状，其俯瞰视图是任意的凸多边形，而其边必须是水平的。还有另一些引擎支持用更复杂的形状构建区域，例如k-DOP（见12.3.4.5节）。若区域总是球形的，设计师可能只需要在属性网格中修改“半径”属性，但要定义或修改任意形状的范围，就几乎必须要有特设的编辑工具了。
- **样条**（spline）：样条是由控制点集所定义的立体曲线，在某些数学曲线中，还会加入控制点上的切线来定义样条。Catmull-Rom是常用样条之一，因为它只需一组顶点来定义（无须切线），而且样条会经过所有控制点。但无论支持哪一种样条类型，世界编辑器通常都需要在视区中显示样条，以及该用户选取及操控个别控制点。有些世界编辑器实际上还支持两种选取模式——“粗略”模式用于选取场景中的对象，以及“细致”模式用于选择已选对象的个别组件，例如样条的控制点或区域的顶点。

13.4.1.9 读 / 写世界组块

当然，无法读 / 写世界组块的世界编辑器并不完整。不同的引擎对于世界组块的读 / 写粒度，差异很大。有些引擎把每个组块储存为单个文件，而另一些引擎则可以独立读 / 写个别的图层。数据格式也有很多选择。有些引擎使用自定义的二进制文件格式，有些则使用如XML的文本格式。每个设计都有其优缺点，但所有编辑器都必须提供某形式的世界组块读 / 写功能，而每个游戏引擎都能够读取世界组块，从而能在运行时于这些组块中进行游戏。

13.4.1.10 快速迭代

优秀的游戏世界编辑器通常都会支持某程度的动态微调功能，供快速迭代（rapid iteration）之用。有些编辑器在游戏本身内执行，让用户即时看到改动的效果。另一些编辑器能连接至运行中的游戏。也有一些世界编辑器完全在脱机状态下运行，它可能是一个独立的工具，或是某DCC工具（如LightWave或Maya）的插件。这些工具有时可以令运行中的游戏动态更新被修改的数据。具体的机制并不重要，最重要的是给用户足够短的**往返迭代时间**（round-trip iteration time，即修改游戏世界，与该改动在游戏中显现效果之间的时间）。迭代并非必须是即时见到结果的。迭代时间应与改动的范围及频率相符。例如，我们或许会期望调整角色的最大血量是一个非常快的操作，但当改动影响整个世界组块光照环境时，就可忍受更长的迭代时间。

13.4.2 集成的资产管理工具

有些引擎中，游戏世界编辑器会整合游戏资产数据库的其他方面功能，例如设定网格/材质的属性、设定动画/混合树/动画状态机、设置对象的碰撞/物理属性、管理材质资源等（关于游戏资产数据库请见6.2.1.2节）。

或许，这种设计最著名的例子是**UnrealEd**，它是虚幻引擎中制作游戏内容的编辑器。UnrealEd直接整合至游戏引擎中，因此编辑器中的任何改动都能即时套用在运行中游戏的动态元素里。这种做法能轻易实现快速迭代。但UnrealEd并非仅是一个世界编辑器，它实际上是一个完整的内容创作软件包。它管理整个游戏资产数据库，无论是动画、音频片段、三角形网格、纹理、材质、着色器……都一手包办。UnrealEd也对用户提供统一、实时、所见即所得的整个资产数据库视图，使它能促进快速、高效的游戏开发过程。图13.7及图13.8展示了几个UnrealEd的截屏。

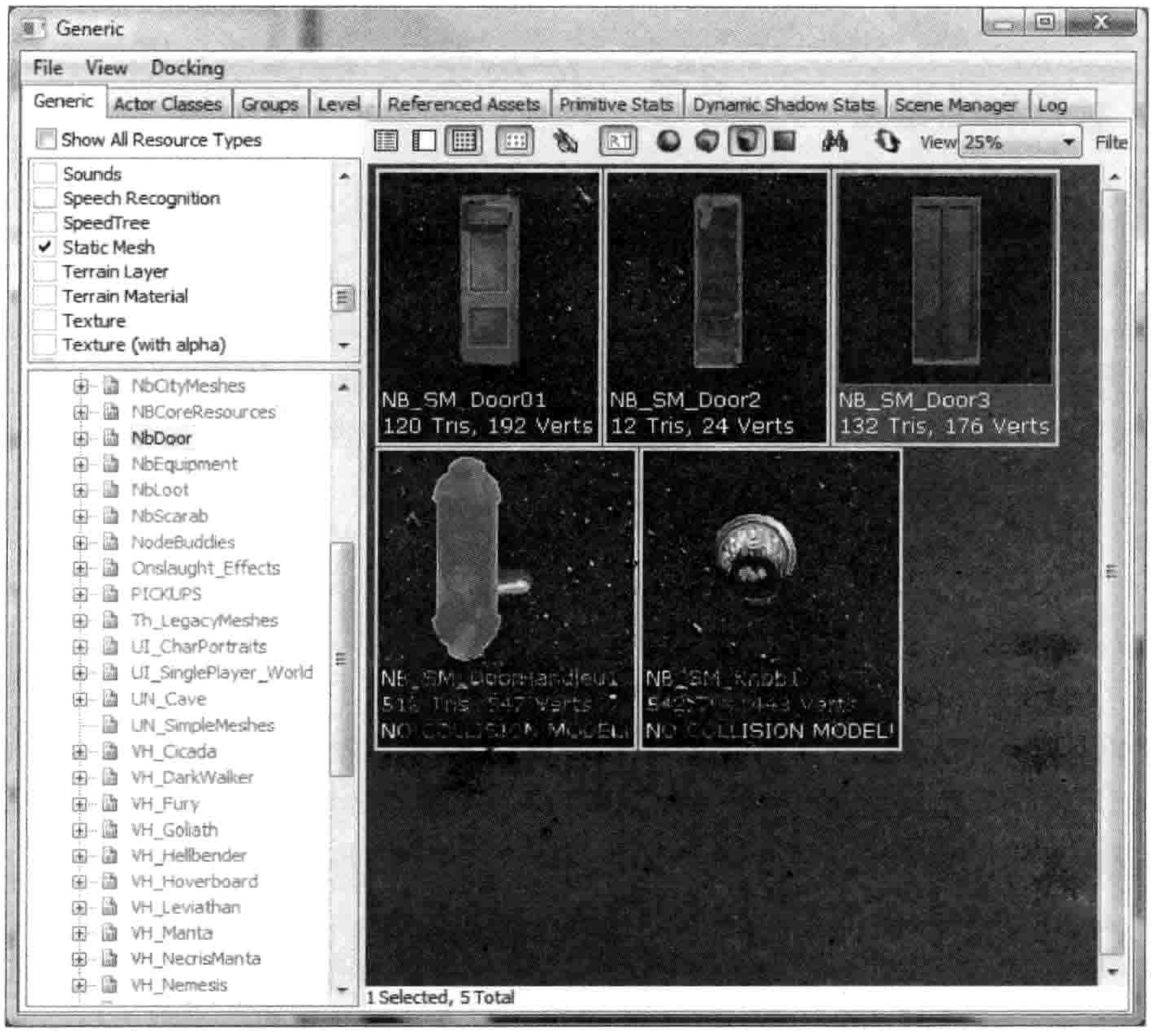


图 13.7: UnrealEd的通用浏览器能访问游戏的整个资产数据库。

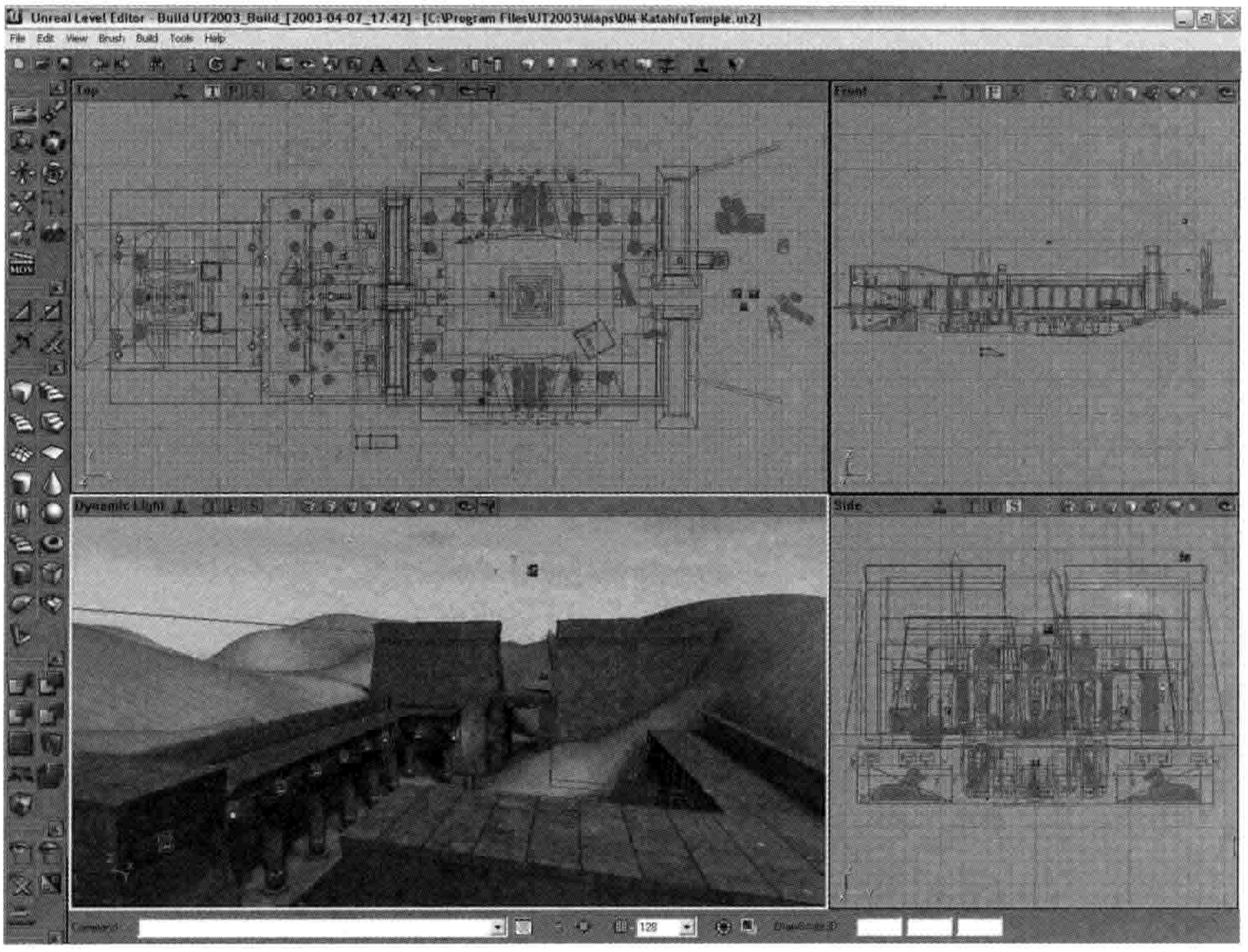


图 13.8: UnrealEd也提供世界编辑器。

13.4.2.1 数据处理成本

在6.2.1节中，我们学习到资产调节管道（asset conditioning pipeline，ACP）是负责把游戏资产从多种来源格式转换至游戏引擎所需的格式。此过程通常含两个步骤。首先，资产从DCC应用程序导出，成为平台无关的中间格式，其中仅含游戏所关注的数据。然后，资产会转换成为特定平台优化的格式。对于多游戏平台的项目，在这第二步中会从单个平台无关的资产生成多个特定平台的资产。

不同工具的关键区别之一在于，这第二步的优化过程在哪个时间点执行。UnrealEd在导入资产时就会对资产优化[6]。此方法在关卡设计上能缩短迭代时间。然而，改动网格、动画、音频片段等来源资产会变得更痛苦。另一些引擎，如Source及雷神之锤引擎，把资产优化延后至烘焙关卡、执行游戏之前。《光环（Halo）》则给用户选择在任意时刻转换原始资源——这些资源在第一次载入至引擎前转换至优化格式，并把结果缓存，避免每次执行游戏时都要无意义地再做转换。

[6]译注：虚幻引擎（至少在译者曾使用过的第3代）在导入资产时，该资产仍然是平台无关的，一些平台可以直接载入这些格式。而虚幻引擎也可以把资产再为平台优化，它称此步骤为cook（烹调）。

第14章　运行时游戏性基础系统

14.1　游戏性基础系统的组件

多数游戏引擎都会带有一套运行时软件组件，它们合作提供一套框架实现游戏独特的规则、目标、动态世界元素。游戏业界对这些组件并无标准命名，但我们把它们总称为引擎的**游戏性基础系统**（gameplay foundation system）。如果我们可以合理地画出游戏与游戏引擎的分界线，那么游戏性基础系统就是刚刚位于该线之下。理论上，我们可以建立一个游戏性基础系统，其大部分是各个游戏皆通用的。然而，实践中这些系统几乎总是包含一些跟游戏类型或具体游戏相关的细节。事实上，引擎和游戏之间的分界，或应视为一大片的模糊区域——这些组件构成的网络一点一点把游戏和引擎连接在一起。有一些游戏引擎更会把游戏性基础系统完全置于引擎/游戏分界线**之上**。游戏引擎之间的重要差异，莫过于其游戏性组件设计与实现的差别。然而，不同引擎之间也有出奇多的共有模式，而这些共有部分正是本章的主要讨论题目。

每个引擎的游戏性软件设计方法都有点不同。然而，多数引擎都会以某种形式提供这些主要的子系统。

- **运行时游戏对象模型**（runtime game object model）：抽象游戏对象模型的实现，供游戏设计师在世界编辑器中使用。
- **关卡管理及串流**（level management and streaming）：此系统负责载入及释放下游戏性用到的虚拟世界内容。许多引擎会在游戏进行时，把关卡数据串流至内存中，从而产生一个巨大无缝世界的感觉（但实际上关卡被分拆成多个小块）。
- **更新实时对象模型**（real-time object model updating）：为了令世界中的游戏对象能有自主（autonomous）的行为，必须定期更新每个对象。这里就是令游戏引擎中所有浑然不同的系统真正合而为一的地方。
- **消息及事件处理**（messaging and event handling）：大多数游戏对象需与其他对象通

信。对象间的消息（message）许多时候是用来发出世界状态改变的信号的，此时就会称这种消息为**事件**（event）。因此，许多工作室会把消息系统称为**事件系统**。

- **脚本**（scripting）：使用C/C++等语言来编写高级的游戏逻辑，或会过于累赘。为了提高生产力、提倡快速迭代，以及把团队中更多工作放到非程序员之手，游戏引擎通常会整合一个脚本语言。这些语言可能是基于文本的，如Python或Lua，也可以是图形语言，如虚幻的Kismet。
- **目标及游戏流程管理**（objectives and game flow management）：此子系统管理玩家的目标及游戏的整体流程。这些目标及流程通常是以玩家目标构成的序列（sequence）、树（tree）或图（graph）所定义的。目标又常会以章（chapter）的方式分组，尤其是一些主要以故事驱动的游戏，许多现代的游戏都是这般。游戏流程管理系统负责管理游戏的整体流程，追踪玩家对目标的完成程度，并且在目标未完成之前阻挡玩家进入另一游戏世界区域。有些设计师称这些为游戏的“脊柱（spine）”。

在这些主要系统之中，运行时对象模型可能是最复杂的。通常它要提供以下大部分（或是全部）功能。

- **动态地产生（spawn）及消灭（destroy）游戏对象**：游戏世界中的动态元素经常需要随游戏性创建及消去。拾起补血包后便会消失；爆炸发生后就会灰飞烟灭；当你以为肃清了整个关卡后，敌方增缓从某个角落神不知鬼不觉地出现。许多游戏引擎会提供一个系统，为动态产生的游戏对象管理内存及相关资源。另一些引擎简单地完全禁止动态地创建、销毁游戏对象。
- **联系底层引擎系统**：每个游戏对象都会联系至一个或多个下层的引擎系统。多数游戏对象在视觉上以可渲染的三角形网格表示，有些游戏对象有粒子效果，有些有声音，有些有动画。多数游戏对象有碰撞信息，有些需要物理引擎做动力学模拟。游戏基础系统的重要功能之一就是，确保每个游戏对象能访问它们所需的引擎系统服务。
- **实时模拟对象行为**：游戏引擎的核心，仍基于代理人模型的实时动态计算机模拟。这句话只不过是花哨地说出，游戏引擎需要随时间更动态地更新所有游戏对象的状态。对象可能需要以某特定次序进行更新。此次序部分由对象间的依赖性所支配，部分基于它们对多个引擎子系统的依赖性，也有部分基于那些子系统本身的相互依赖性。
- **定义新游戏对象类型**：游戏在开发过程中，伴随着每个游戏需求的改变及演进。游戏对象模型必须有足够的弹性，可以容易地加入新的对象类型，并在世界编辑器中显示这些新对象类型。理想地，新的游戏类型应可以完全用数据驱动方式定义。然而，在许多引擎中，新增游戏类型需要程序员的参与。
- **唯一的对象标识符（unique object id）**：典型的游戏世界包含数百上千的不同类型游戏对象。在运行时，必须能够识别或找到想要的对象。这意味着，每个对象需要有

某种唯一标识符。人类可读的名称是最方便的标识符类型，但我们必须警惕在运行时使用字符串所带来的性能成本。整数标识符是最高性能之选，但对人类游戏开发者来说最难使用。也许使用字符串散列标识符（hashed string id，见5.4.3.1节）作为对象标识符是最好的方案，因为它们的性能如整数标识符，但又能转化为字符串，容易供人类辨识。

- **游戏对象查询（query）**：游戏性基础系统必须提供一些方法去搜寻游戏世界中的对象。我们可能希望以唯一标识符取得某个对象，或是取得某类型的所有对象，或是基于随意的条件做高级查询（例如寻找玩家角色20m以内的所人敌人）。
- **游戏对象引用（reference）**：当找到了所需的对象，我们需要以某种机制保留其**引用**，或许只是在单个函数内做短期保留，也有可能需要保留更长的时间。对象引用可能简单到只是一个C++类实例指针，也可能使用更高级的机制，例如句柄或带引用计数的智能指针。
- **有限状态机（finite state machine，FSM）的支持**：许多游戏对象类型的最佳建模方式是使用有限状态机。有些游戏引擎可以令游戏对象处于多个状态之一，而每个状态下有其属性及行为特性。
- **网络复制（network replication）**：在网络多人游戏中，多个游戏机器通过局域网或互联网连接在一起。某个对象的状态通常是由其中一台机器所拥有及管理的。然而，对象的状态也必须**复制**（通信）至其他参与该多人游戏的机器，使所有玩家能见到一致的对象。
- **存档及载入游戏、对象持久性（object persistence）**：许多游戏引擎能把世界中游戏对象的当前状态储存至磁盘，供以后读入。引擎可以实现“任何地方存档”的游戏存档系统，或实现网络复制的方式，或是简单地使用世界编辑器储存/载入游戏世界组块的方式。对象特久性通常需要一些编程语言的功能，例如，**运行时类型识别**（runtime type identification，RTTI）、**反射**（reflection），以及**抽象构造**（abstract construction）。RTTI及反射令软件在运行时能动态地判断对象的**类型**，以及类里有哪些**属性**及**方法**。抽象构造可以在不硬编码类的名称的同时，创建该类的实例。此功能在把对象从磁盘序列化一个对象至内存时十分有用。若你所选用的语言没有RTTI、反射或抽象构造的原生支持，可以手工加入这些功能。

本章余下之篇幅将会逐一深入探究这些子系统。

14.2 各种运行时对象模型架构

游戏设计师使用世界编辑器时，会面对一个抽象的游戏对象模型。该模型定义了游戏世界中能出现的多种动态元素，指定它们的行为是怎样的，它们有哪些属性。在运行时，游戏性基础系统必须提供这些对象模型的具体实现。此模型是任何游戏性基础系统中最巨大的组件。

运行时对象模型的实现，可能与工具方的抽象对象模型相似，也可能不相似。例如，运行时对象模型可能完全不是用面向对象编程语言来实现的，它也可能是用一组互相连接的实例表示的单个抽象游戏对象。无论设计是怎样的，运行时对象模型必须忠实地复制出世界编辑器所展示的对象类型、属性及行为。

相对设计师所见的工具方抽象对象模型，运行时对象模型是其游戏中的表现。运行时对象模型有不同设计，但多数游戏引擎会采用以下两种基本架构风格之一。

- **以对象为中心**（object-centric）：此风格中，每个工具方游戏对象，在运行时是以单个类实例或数个相连的实例所表示。每个对象含一组**属性**及**行为**，这些都会封装在那些对象实例的类（或多个类）之中。游戏世界只不过是游戏对象的集合。
- **以属性为中心**（property-centric）：此风格中，每个工具方游戏对象仅以唯一标识符表示（可实现为整数、字符串散列标识符或字符串）。每个对象的**属性**分布于多个数据表，每种属性类型对应一个表，这些属性以对象标识符为键（而非集中在单个类实例或相连的实例集合）。属性本身通常是实现为硬编码的类之实例。而游戏对象的**行为**，则是隐含地由它组成的属性集合所定义的。例如，若某对象含“血量”属性，该对象就能被攻击、扣血，并最终死亡。若对象含“网格实例”属性，那么它就能在三维中渲染为三角形网格的实例。

以上的两个架构风格都有其独特的优缺点。我们将逐一探究它们的一些细节，当在某方面其中一个风格可能极优于另一风格时，我们会特别指明。

14.2.1 以对象为中心的各种架构

在以**对象为中心**的游戏世界对象架构中，每个逻辑游戏对象会实现为类的实例，或一组互相连接的实例。在此广阔的定义下，可做出多种不同的设计。以下我们介绍几种最常见的设计。

14.2.1.1 一个简单以C实现的基于对象的模型：《迅雷赛艇》

游戏对象模型并不一定要使用如C++等面向对象语言来实现。例如，圣迭戈Midway公司的街机游戏《迅雷赛艇》就是完全用C写成的。《迅》采用了一个非常简单的游戏对象模型，当中只含几个对象类型。

- 赛艇（玩家及人工智能所控制的）。
- 飘浮的红、蓝加速图标。
- 背景具动画的物体（如赛道旁的动物）。
- 水面。
- 斜台。
- 瀑布。
- 粒子效果。
- 赛道板块（多个二维多边区域连接在一起，用于定义赛艇能跑的水域）。
- 静态几何（地形、植皮、赛道旁的建筑物等）。
- 二维平视头显示器（HUD）元素。

图14.1是几张《迅雷赛艇》的截图。注意两张图中都有加速图标，而左图中有鲨鱼经过（这是一个具动画的背景物体例子）。

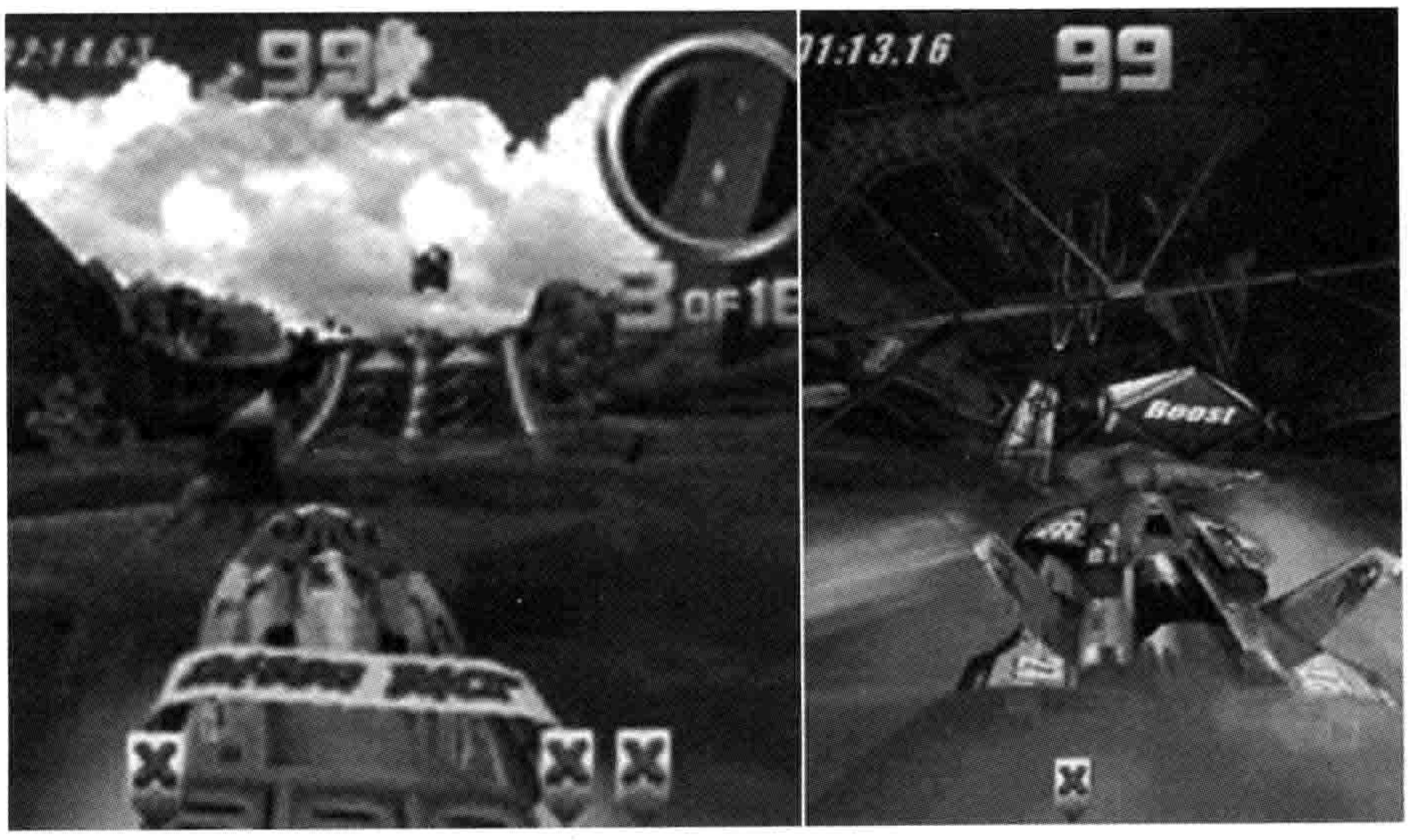


图 14.1: 圣迭戈Midway公司的街机游戏《迅雷赛艇》截图。

《迅》中有一个名为World_t的C struct，用于储存及管理游戏世界的内容（即一个赛道）。世界内包含各种游戏对象的指针。当中，静态几何仅仅是单个网格实例。而水面、瀑布、粒子效果各有自己的数据结构。赛艇、加速图标及游戏中其他动态对象则表示为WorldOb_t（即世界对象）这个通用struct的实例。《迅》中的这种对象就是本章所定义的**游戏对象**的例子。

WorldOb_t数据结构内的数据成员包括对象的位置和定向、用于渲染该对象的三维网格、一组碰撞球体、简单的动画状态信息（《迅》只支持刚体层次式动画）、物理属性（速度、质量、浮力），以及其他动态对象都会拥有的数据。此外，每个WorldOb_t还含有3个指针：一个void* “用户数据（user data）”指针、一个指向“update”函数的指针及一个“draw”函数的指针。因此，虽然《迅》并不是严格意义上的面向对象，但《迅》的引擎实质上扩展了非面向对象语言（C），基本地实践两个重要的OOP特征：**继承**（inheritance）和**多态**（polymorphism）。用户数据指针令每个游戏对象可维系一些对游戏对象类型相关的自定义状态信息，也同时能令所有世界对象继承一些共有的功能。例如“**Banshee**”赛艇的加速机制不同于“**Rad Hazard**”，并且每种加速机制需要不同的状态信息去管理其起动及结束动画。这两个函数指针的用途如同**虚函数**，使世界对象有多态的行为（通过“update”函数），以及多态的视觉外观（通过“draw”函数）[1]。

```
struct WorldOb_s
{
    Oreint_t    m_transform;     /* 位置/定向 */
    Mesh3d*     m_pMesh;         /* 三维网格 */
    /* ... */
    void*       m_pUserData;     /* 自定义状态 */
    void        (*m_pUpdate)();  /* 多态更新 */
    void        (*m_pDraw)();    /* 多态绘制 */
};
typedef struct WorldOb_s WorldOb_t;
```

14.2.1.2 单一庞大的类层次结构

很自然地我们会用分类学的方式把游戏对象类型归类。此思考方式会促使游戏程序员选择一个支持继承功能的面向对象语言。表示一组互相有关联的游戏对象类型，最直观、明确的方式就是使用类层次结构。因此，大部分商业游戏引擎都采用类层次结构，这是完全意料中的事。

[1]译注：这种多态行为的实现方式和C++的虚函数有所分别。C++对象模型为每个多态类储存一个静态的虚函数表，其中储存一个至多个函数指针，对象则拥有指向某个虚函数表的指针。而这个例子则是每个对象直接拥有多个不同的函数指针，可以为对象动态地合成函数。

图14.2展示了一个可用于实现《吃豆人》的简单类层次结构。此层次结构（如同许多游戏引擎）都是以名为GameObject的类为根的，它可能提供所有对象都共同需要的功能，例如RTTI或序列化。而MovableObject类则用于表示所有含位置及定向的对象[2]。RenderableObject给予对象获渲染的能力（如果是传统的《吃豆人》，就会使用精灵/sprite；如果是现代三维的版本，就可能是使用三角形网格）。从RenderableObject派生了鬼、吃豆人、豆子及大力丸等类，构成了整个游戏。这只是一个假想例子，但它展示了多数游戏对象类层次结构背后的基本概念——共有的、通用的功能会接近层次结构的根，而越接近层次结构叶端的类则会加入越多的专门功能。

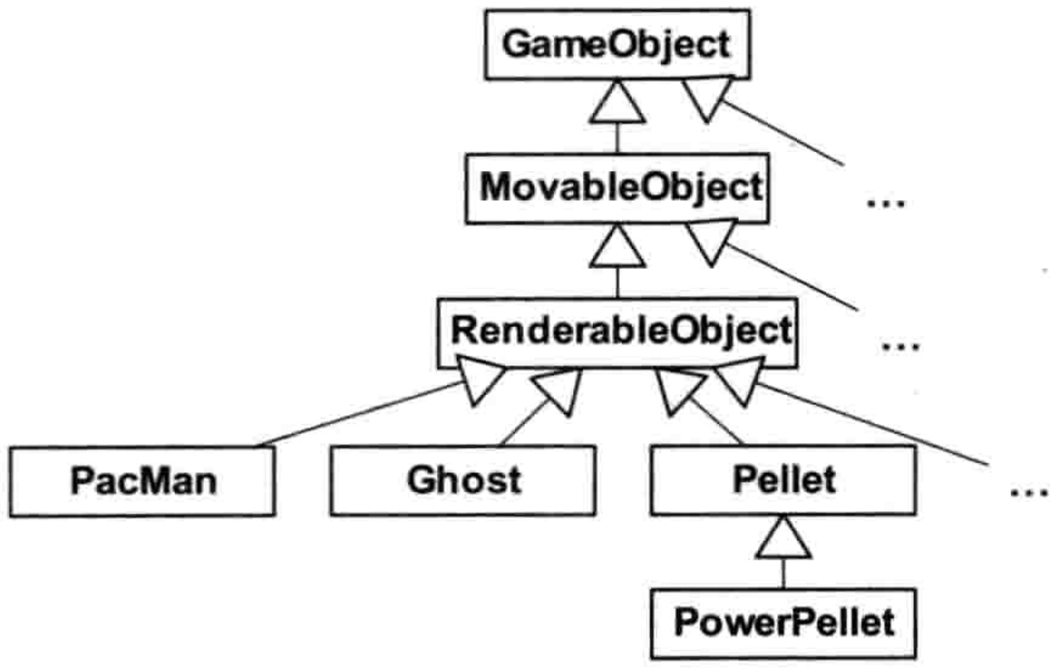


图 14.2: 《吃豆人》的假想类层次结构。

一开始时，游戏对象类层次结构通常是简单、轻盈的，在这种情况下的层次结构可能是一个十分强大而且直觉的游戏对象类型描述方式。然而，随着类层次结构的成长，它会倾向同时往纵、横方向发展，形成笔者称之为**单一庞大的类层次结构**（monolithic class hierarchy）。当游戏对象模型中几乎所有的类都是继承自单个共通的基类时，就会产生这种层次结构。虚幻引擎的游戏对象模型就是一个经典例子（图14.3）。

14.2.1.3 深宽层次结构的问题

单一庞大的类层次结构对游戏开发团队来说，可导致很多不同类型的问题。类层次结构成长得越深越宽，这些问题就变得越极端。我们利用以下几个部分探讨深宽层次结构的最常见问题。

类的理解、维护及修改

一个类越是在类层次结构中越深的地方，就越难理解、维护及修改。因为要理解一个

[2]译注：这个命名可能不甚理想，因为有位置及定向的对象不一定是可以移动的。

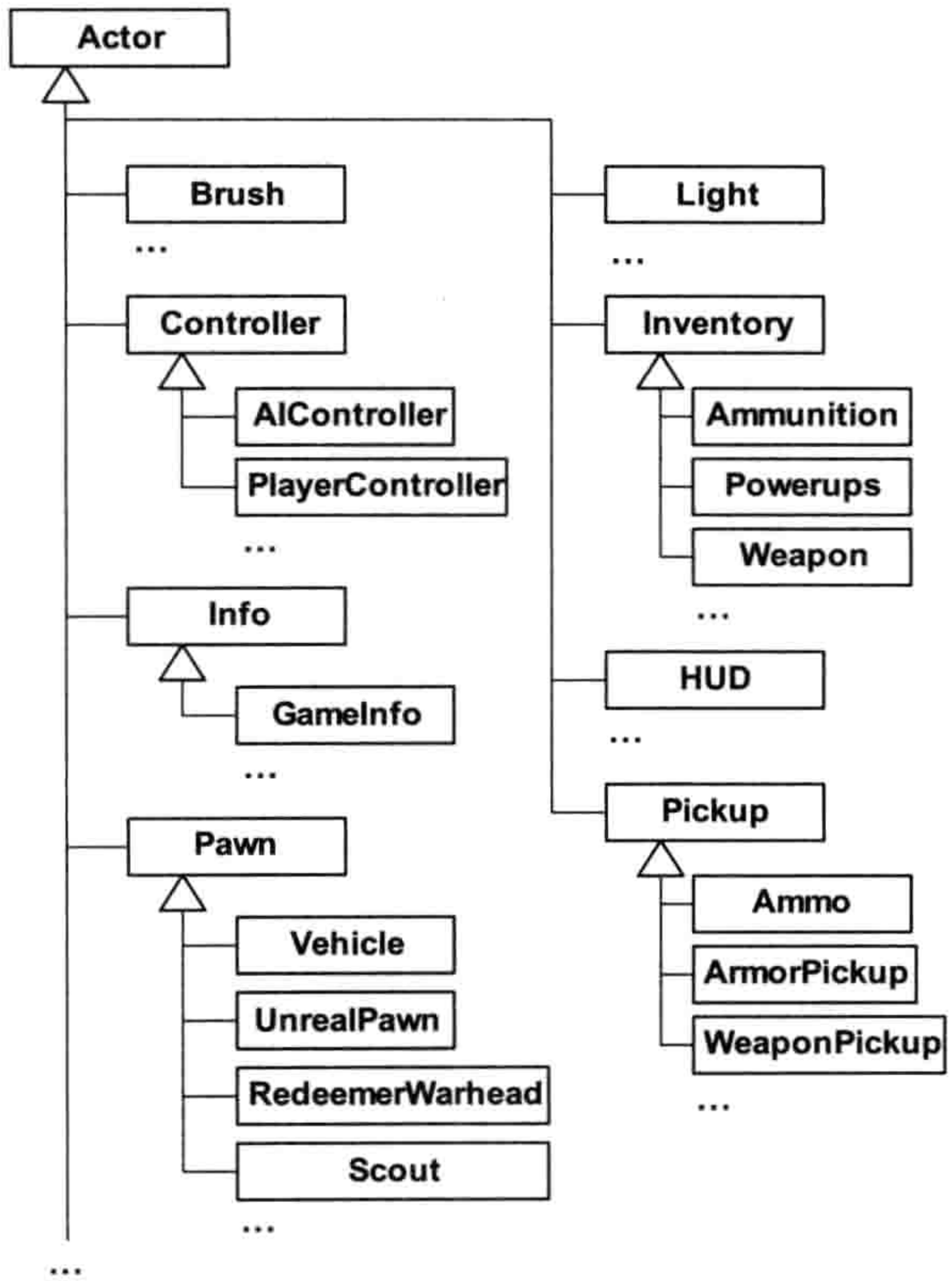


图 14.3: 《虚幻竞技场2004（Unreal Tournament 2004）》的游戏对象类层次结构摘录。

类，就需要理解其所有父类。例如，在派生类中修改一个看似无害的虚函数，就可能会违背了众基类中某个基类的假设，从而产生微妙又难以找到的bug。

不能表达多维的分类

每个层次结构都使用了某种标准分类对象，这些标准称为**分类学**（taxonomy）。例如，**生物分类学**（biological taxonomy，又称作alpha taxonomy）基于遗传的相似性分类所有生物，它使用了8层的树：域（domain）、界（kingdom）、门（phylum）、纲（class）、目（order）、科（family）、属（genus）、种（species）。在树中的每一层，会采用不同的指标把地球上无数的生命形式分割成越来越仔细的群组。

任何层次结构的最大问题之一就是，它只能把对象在每层中用单个“轴”分类——即基于某单一特定的标准做分类。当设计层次结构时选择了某个标准，就很难，甚至不可能用另一个完全不同的“轴”分类。例如，生物分类学是基于遗传特性分类生物的，它并没有说明

生物的颜色。若要以颜色为生物分类，则需要另一个完全不同的树结构。

在面向对象编程中，层次结构分类所形成的这种限制很多时候会展现在深、宽、令人迷惘的类层次结构中。当分析一个真实游戏的类层次结构时，许多时候我们会发现它会把多种不同的分类标准尝试合并在单一的类树中。在另一些情况下，若某个新对象类型的特性是在原有层次结构设计的预料之外，就可能会做出一些让步令该新类型可于置于层次结构中。例如，图14.4所展示的类层次结构，好像能合乎逻辑地把不同的载具（vehicle）分类。

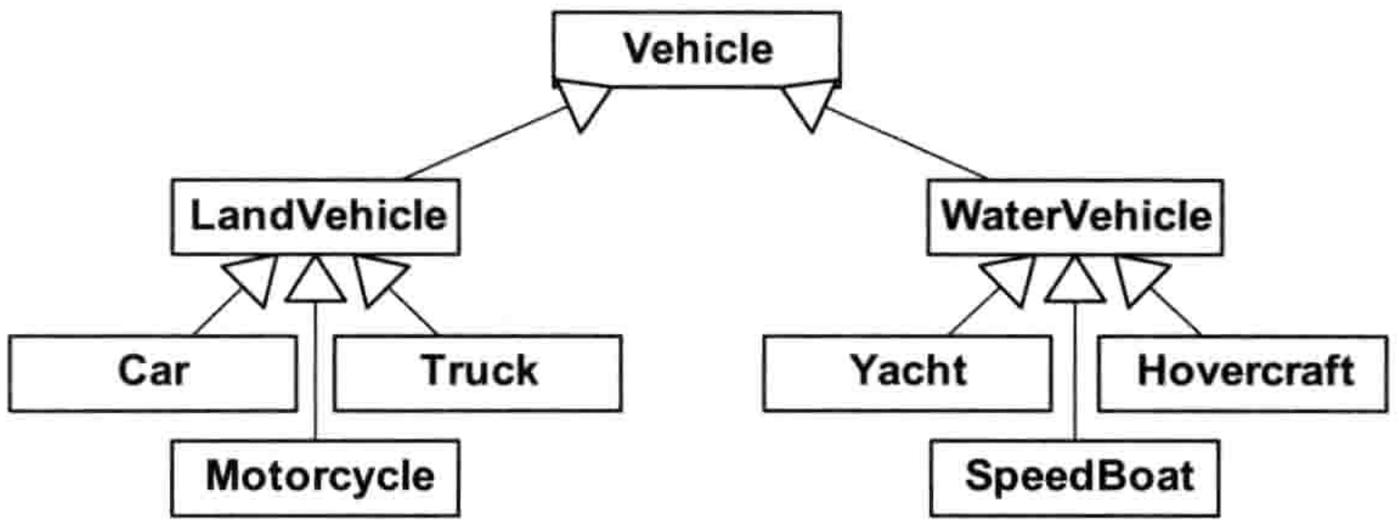


图 14.4: 好像能合乎逻辑地描述各类载具的类层次结构。

那么，当游戏设计师对程序员宣布，他们要在游戏中加入**水陆两用载具**（amphibious vehicle）时，可以怎么办？那种载具不能套进现有的分类系统。这可能会令程序员惊惶失措，或更有可能的是把该类结构“强行修改（hack）”成丑陋、易错的方式。

多重继承：致命钻石

水陆两用载具的问题，解决方法之一是利用C++的多重继承（multiple inheritance，MI）功能，如图14.5所示。然而，C++的多重继承又会引致一些实践上的问题。例如，多重继承会令对象拥有基类成员的多个版本——此情况称为“致命钻石（deadly diamond）”或“死亡钻石（diamond of death）”。详情可参阅本书3.1.1.3节。

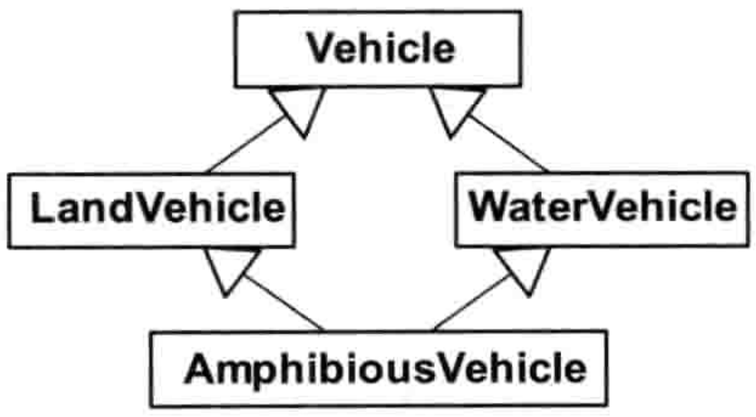


图 14.5: 水陆两用载具的钻石形类层次结构。

实现一个又可工作、又易理解、又能维护的多重继承类层次结构，其难度通常超过其得益。因此，多数游戏工作室禁止或严格限制在类层次结构中使用多重继承。

mix-in类

有些团队容许使用多重继承的一种形式——一个类可以有任意数量的父类但只能有一个祖父类。换言之，一个类可以派生自主要继承层次结构中的一个且仅一个类，但也可以继承任意数量的**mix-in类**（无基类的独立类）。那么共用的功能就能抽出来，形成mix-in类，并把这些功能在需要的时候定点插入主要继承层次结构中。图14.6显示了一个例子。然而，下面将提及，通常更好的做法是**合成**（composition）或**聚合**（aggregation）那些类，而不是**继承**它们。

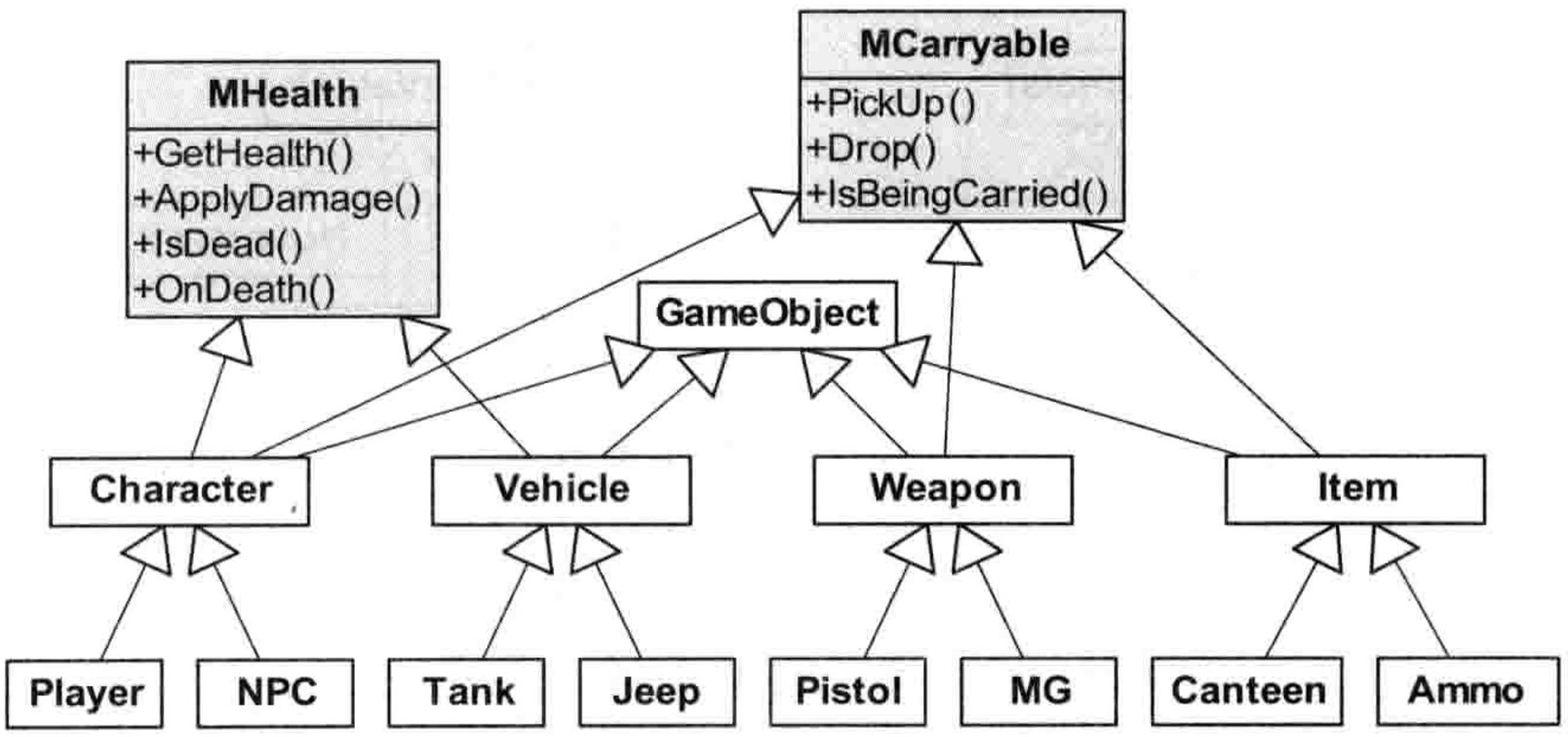


图 14.6: 含mix-in类的类层次结构。任何继承MHealth mix-in类的类会加血量信息，并可以被杀。MCarryable mix-in类可以令其派生类的对象被角色携带。

冒泡效应

在设计庞大类层次结构之初，其一个或多个根类通常非常简单，每个根类有最低限度的功能集。然而，当游戏加入越来越多的功能，就可能越容易尝试**共享**两个或更多个**无关**类的**代码**，这种欲望会令功能沿层次结构往上移，笔者称之为“冒泡效应（bubble up effect）”。

例如，开始时我们可能做出这么一个设计，只有木箱能浮于水面。然而，当游戏设计师见到那些很酷的漂浮箱子，他们就会要求加入更多的漂浮对象，例如角色、纸张、载具等。因为“可浮与不可浮”并非原来设计时的分类标准，程序员们很快就会发现有需要把漂浮功能加至类层次结构中毫不相关的类之中。由于不想使用多重继承，程序员们就决定把漂浮相关的代码往层次结构上方搬移，那些代码会置于全部漂浮对象所共有的基类之中。事实上一些派生自该基类的对象并**不能漂浮**，但此问题的程度不及把代码在多个类各复制一次的问题。（也可加入如m_bCanFloat这种布尔成员变量以分开两种情况。）最后，漂浮功能（以及许多其他游戏功能）会置于继承层次结构的根类。

虚幻引擎的Actor（演员）类可说是此“冒泡效应”的经典例子。它包含的数据成员及代码涵盖管理渲染、动画、物理、世界互动、音效、多人游戏的网络复制、对象创建及销毁、演员更新（即基于某些条件迭代所有演员，并对他们进行一些操作），以及消息广播。当我们容许一些功能在单一庞大的层次结构中像泡沫般上移，多个引擎子系统的封装工作会变得困难。

14.2.1.4 使用合成简化层次结构

或许，单一庞大层次结构的最常见成因就是，在面向对象设计中过度使用“是一个（is-a）”关系。例如，在游戏的GUI中，程序员可能基于GUI视窗总是长方形的逻辑，把Window类派生自Rectangle类。然而，一个视窗**并不是**一个长方形，它只是**拥有**一个长方形，用于定义其边界。因此，这个设计问题的更好解决方法是把Rectangle的实例安置于Window类之中，或是令Window拥有一个Rectangle的指针或参考。

在面向对象设计中，“有一个”关系称为**合成**（composition）。在合成中，A类不是直接拥有B类**实例**，便是拥有B类实例的**指针**或**参考**。严格来说，使用“合成”一词时，必须指A类**拥有**B类。这即是说，当构造A类实例时，它也会自动创建B类的实例；当销毁A类的实例时，也会自动销毁B类的实例。我们也可以用指针或参考把两个类连接起来，而当中的一个类并不管理另一个类的生命周期，这种技术称之为**聚合**（aggregation）。

把“是一个”改为“有一个”

要降低游戏类层次结构的宽度、深度、复杂度，一个十分有用的方法是把“是一个”关系改为“有一个”关系。我们使用图14.7中的单一层次结构假想例子说明此技巧。GameObject根类提供所有游戏对象所需的共有功能（如RTTI、反射、通过序列化实现持久性、网络复制等）。MovableObject类用于表示任何含空间**变换**（即位置、定向，以及可选的比例）的对象。RenderableObject加入了在屏幕上渲染的功能。（非所有游戏对象都需要被渲染，例如，隐形的TriggerRegion类就可以直接继承自MovableObject。）CollidableObject类对其实例提供碰撞信息。AnimatingObject类给予其实例一个通过骨骼关节结构播放动画的能力。最后，PhysicalObject类给予其实例被物理模拟的能力（例如，一个刚体能受引力影响往下掉，并被游戏世界反弹）。

此类继承结构的一大问题在于，它限制了我们创造新游戏类型的设计选择。若我们想定义一个能受物理模拟的对象类型，我们被迫把该类派生自PhysicalObject，即使它并不需要骨骼动画。若我们希望一个游戏对象类有碰撞功能，它必须要派生自CollidableObject，即使它可能是隐形的，并不需要RenderableObject的功能。

图14.7中的类继承结构的第2个问题在于，难以扩展现存类的功能。例如，假设我们希望支持变形目标动画，那么我们会令AnimatingObject派生两个新类，SkeletalObject及MorphTargetObject。若我们要令这两个类都支持物理模拟，就必须重构PhysicalObject成为两个近乎相同的类，一个派生自SkeletalObject，一个派生自MorphTargetObject，或是改用多重继承。

这些问题的一个解决方法是，把GameObject不同的功能分离成为独立的类，每个类负责单一、定义清楚的服务。这些类有时候称为**组件**（component）或**服务对象**（service object）。组件化的设计令我们可以只选择游戏对象所需的功能。此外，每项功能可以独立地维护、扩充或重构，而不影响其他功能。这些独立的组件也更易理解及测试，因为它们和其他组件没有耦合。有些组件类直接对应单个引擎子系统，例如渲染、动画、碰撞、物理、音频等。当某个游戏对象整合多个子系统时，这些子系统能互相保持距离及良好的封装。

图14.8展示了把类层次结构重构为组件后的可行设计。在此设计中，GameObject变成一个枢纽（hub），含有每个可选组件的指针。MeshInstance组件取代了RenderableObject类，它表示一个三角形网格的实例，并封装了如何渲染该网格的知识。类似地，AnimationController组件替代了AnimatingObject，把骨骼动画服务提供给GameObject。Transform类取代MovableObject维护对象的位置、定向及比例。RigidBody类展示游戏对象的碰撞几何，并为GameObject提供对底层碰撞及物理系统的接口，从而代替了CollidableObject及PhysicalObject。

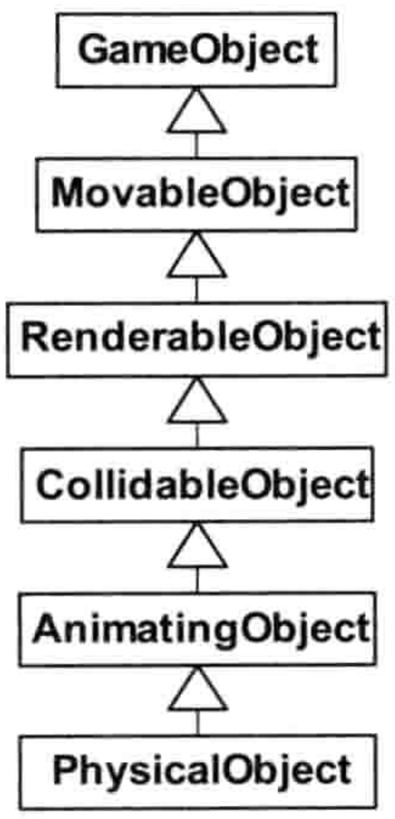


图 14.7: 假想游戏对象层次结构，仅以继承连接各类。

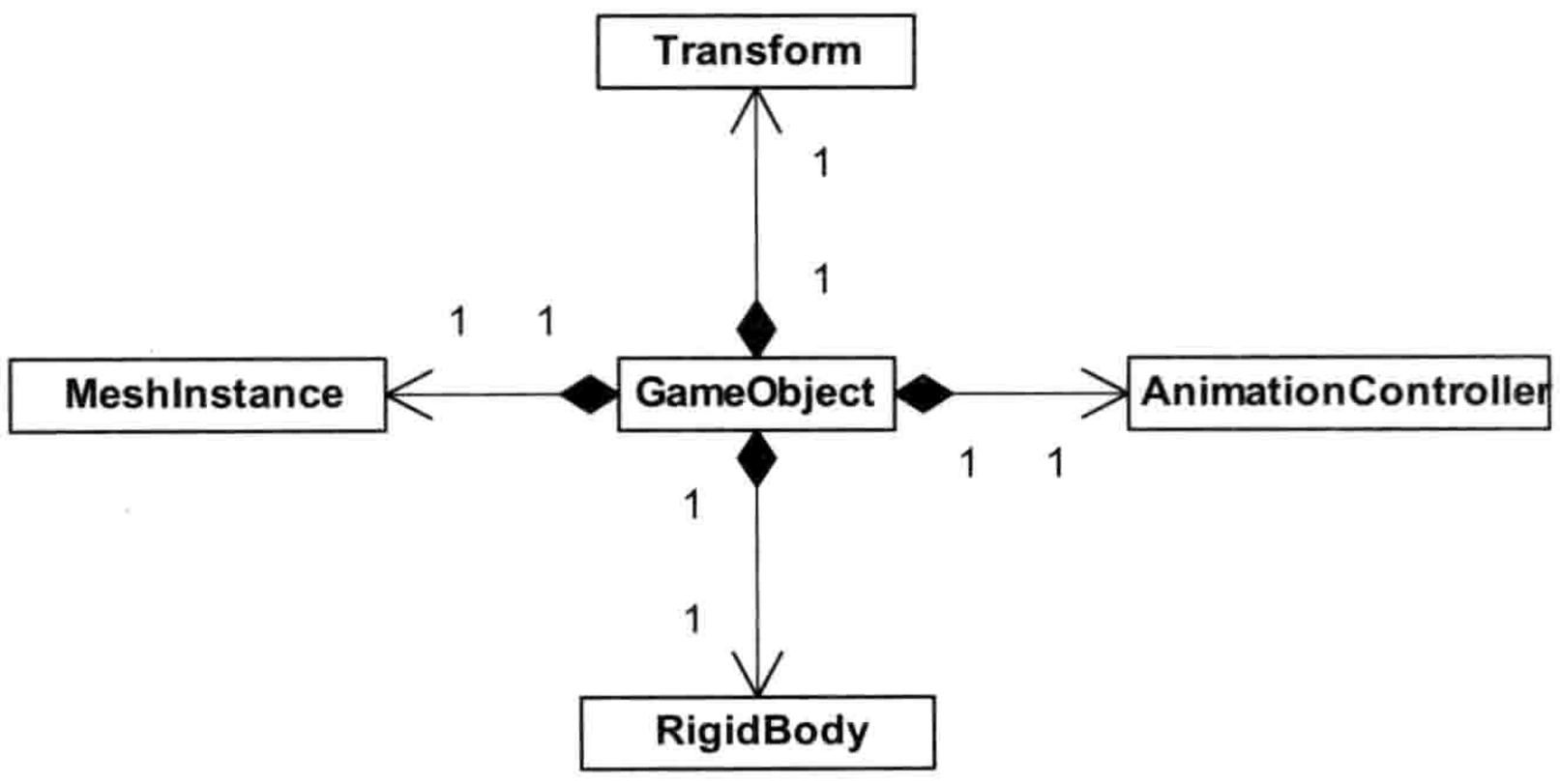


图 14.8: 重构假想的游戏对象层次结构，相对继承更偏爱类的合成。

组件的创建及拥有权

在这种设计中，通常“枢纽”类拥有其组件，即是说它管理其组件的**生命周期**。但GameObject怎么知道要创建哪些组件？对此有多个解决方案，最简单的就是令GameObject根类拥有所有可能组件的指针。每个游戏对象类型都派生自GameObject类。GameObject的构造函数把所有组件指针初始化为NULL。而在派生类的构造函数中，就能自由选择创建其所需的组件。方便起见，默认的GameObject析构函数可以自动地清理所有组件。在这种设计中，派生自GameObject的类层次结构成为了游戏对象的主要分类法，而组件类则作为可选的增值功能。

以下展示了一个组件创建销毁逻辑的可行实现。然而，记住这段代码仅是作为例子之用，实现细节可能会有许多细节变化，甚至采用实质相同类层次结构的引擎也会有许多实现上的出入。

```
class GameObject
{
protected:
    // 我的变换 (位置、定向、比例)
    Transform                   m_transform;

    // 标准组件
    MeshInstance*               m_pMeshInst;
    AnimationController*        m_pAnimController;
    RigidBody*                  m_pRigidBody;

public:
    GameObject()
    {
        // 默认无组件。派生类可以覆写
        m_pMeshInst = NULL;
        m_pAnimController = NULL;
        m_pRigidBody = NULL;
    }

    ~GameObject()
    {
        // 自动删除被派生类创建的组件
        delete m_pMeshInst;
        delete m_pAnimController;
        delete m_pRigidBody;
    }
```

```
    // ……
};

class Vehicle : public GameObject
{
protected:
    // 加入载具的专门组件
    Chassis*    m_pChassis;
    Engine*     m_pEngine;

    // ……

public:
    Vehicle()
    {
        // 构建标准GameObject组件
        m_pMeshInst = new MeshInstance;
        m_pRigidBody = new RigidBody;

        // 注意：我们假设动画控制器必须引用网格实例，
        // 才能令控制器取得矩阵调色板
        m_pAnimController
            = new AnimationController(*m_pMeshInst);

        // 构建载具的专门组件
        m_pChassis = new Chassis(*this, *m_pAnimController);
        m_pEngine = new Engine(*this);
    }

    ~Vehicle()
    {
        // 只需析构载具的专门组件，因为GameObject会为我们析构标准组件
        delete m_pChassis;
        delete m_pEngine;
    }
};
```

14.2.1.5 通用组件

另一个更有弹性（但实现起来更棘手）的方法是，于根游戏对象类加入通用组件的链表。在这种设计中，组件通常都会继承自一个共有的基类，使迭代链表时能利用该基类的多态操作，例如，查询该类的类型，或逐一向组件传送事件以供处理。此设计令根游戏对象类

几乎不用关心有哪些组件类型，因而在大部分情况下，可以无须修改游戏对象就能创建新的组件类型。此设计也能让每个游戏对象拥有任意数量的同类型组件实例。（硬编码的设计只容许固定的数量，具体视乎游戏对象类里每个组件类型有多少个指针。）

图14.9展示了这种设计。相比硬编码的组件模型，这种设计较难实现，因为我们必须以完全通用的方式来编写游戏对象的代码。同样地，组件类也不可以假设在某游戏对象中有哪些组件。是使用硬编码组件指针的设计，还是使用通用组件的链表，并不是一个简单的决策。两者各有优缺点，各游戏团队会有不同之选。

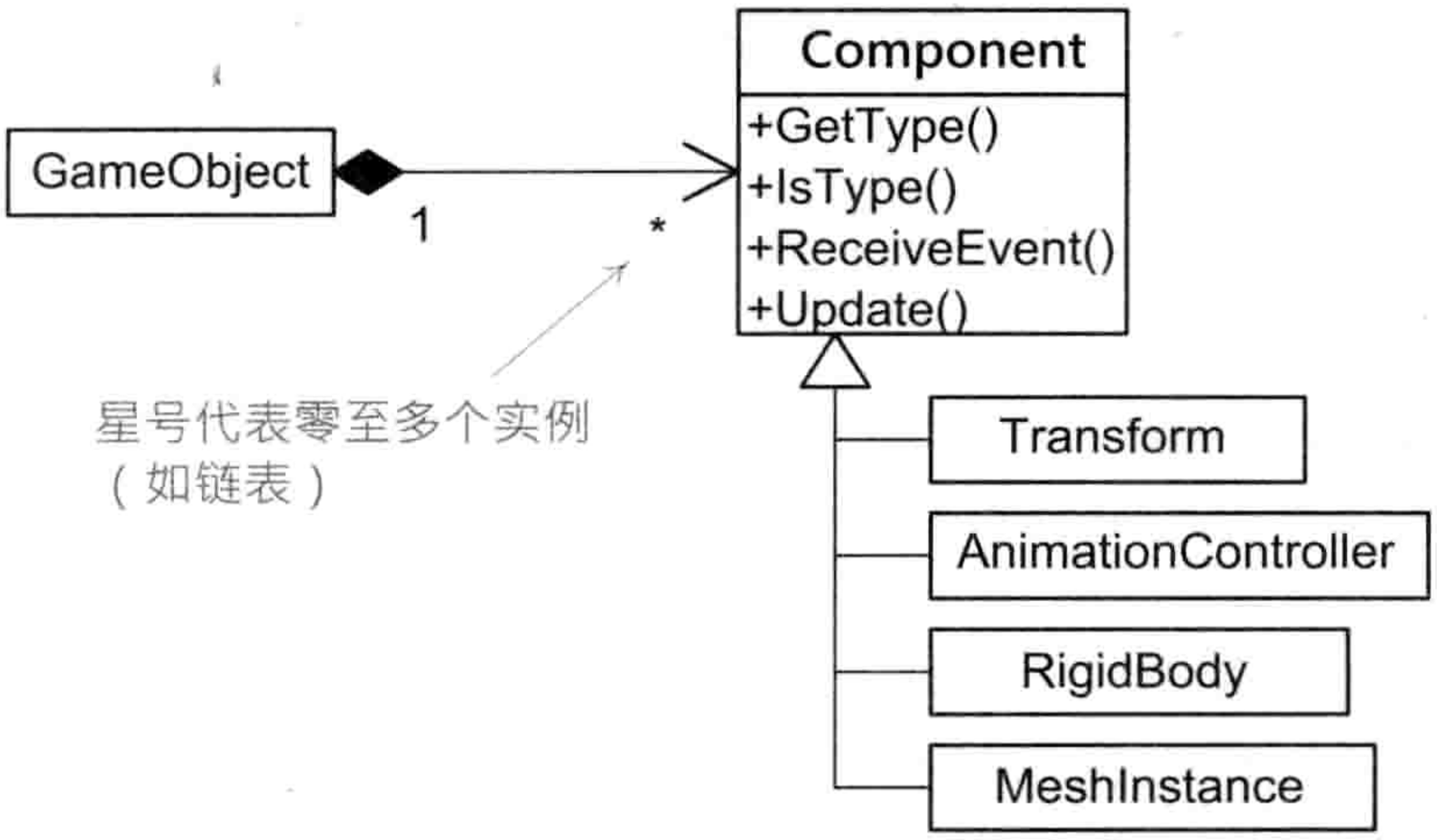


图 14.9: 组件链表可提升弹性，因为枢纽游戏对象不用关注各组件的细节。

14.2.1.6 纯组件模型

若我们把组件的概念发挥至极致，会是如何的呢？我们可以把GameObject根类的几乎**所有**功能移到多个组件类中。那么，游戏对象类就差不多变成一个无行为的容器，它含有唯一标识符及一些组件的指针，但自己却不含任何逻辑。既然如此，何不删去那个根类呢？要这么做，其中一个方法是把游戏对象的标识符复制至每个组件中。那么组件就能逻辑地以标识符分组方式连接起来。若能提供一个以标识符查找组件的快速方法，我们便无须GameObject这个枢纽。笔者称这种架构为**纯组件模型**（pure component model），如图14.10所示。

刚开始时，可能会觉得纯组件模型并不简单，而且它也带有一些问题。例如，我们仍要定义游戏所需的具体游戏对象类型，并且在创建那些对象时安插正确的组件实例。之前的GameObject的层次结构可以帮助我们处理组件创建。若使用纯组件模型，取而代之，我们可以用工厂模式（factory pattern），对每个游戏对象定义一个工厂类（factory class），

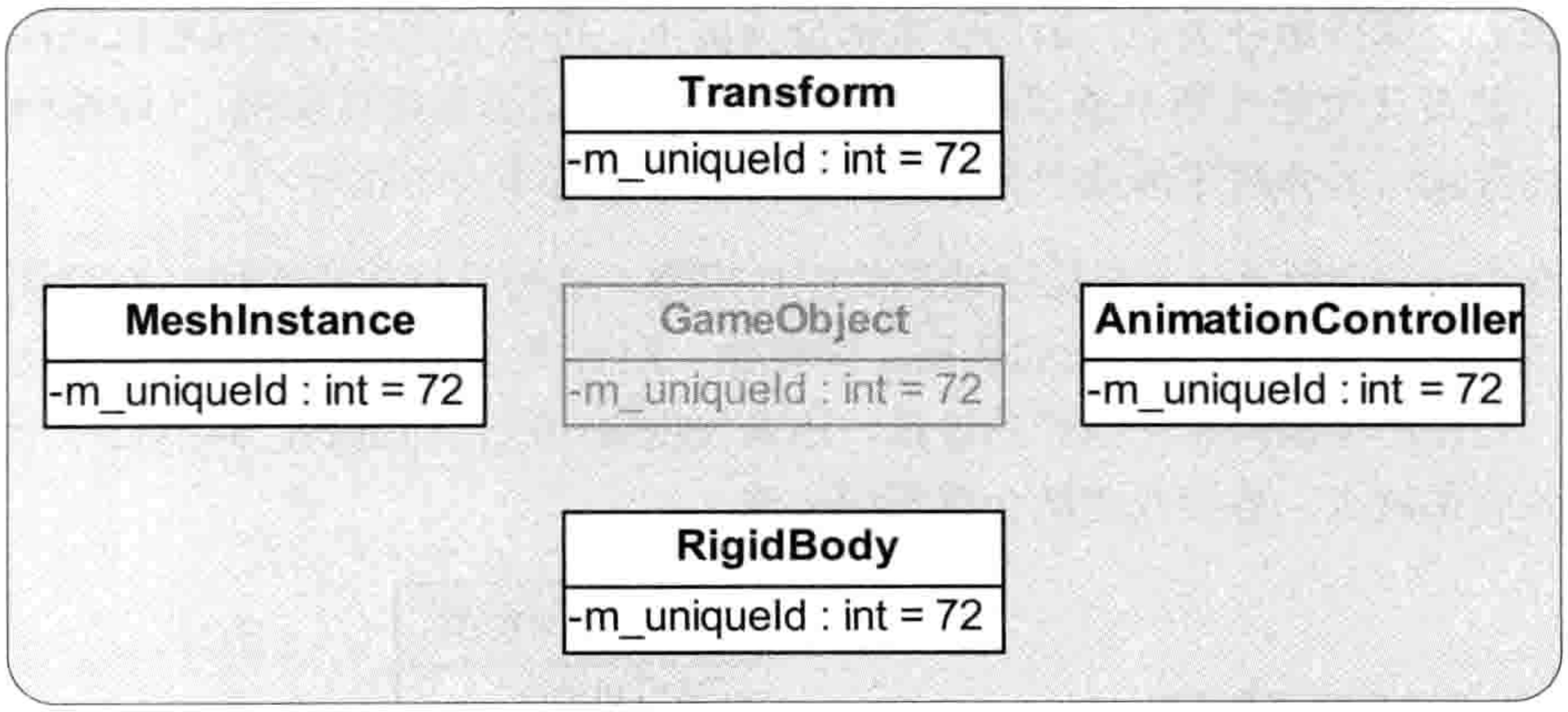


图 14.10: 在纯组件模型中，逻辑游戏对象是由许多组件组成的，但组件只是通过唯一标识符间接地连接在一起。

内含一个虚拟建构函数创建该对象类型所需的组件。又或者，我们可以改用数据驱动模型，通过由引擎读取文本文件所定义的游戏对象类型，决定为游戏对象创建哪些组件。

另一个纯组件模型的问题，在于组件间的通信。我们的中央GameObject当作“枢纽”，可编排多个组件间的通信。在纯组件架构中，我们需要一个高效的方法，令单个对象中的组件能互相通信。当然，组件可以使用游戏对象的唯一标识符来查找该对象的其他组件。然而，我们很可能需要更高效的机制，例如，预先把组件连接成循环链表。

在这种意义上，纯组件模型中，某游戏对象与另一游戏对象的通信也面对相同困难。我们不能再通过GameObject实例做通信媒介，而必须事前知道我们要与哪一个组件通信，又或是对目标游戏对象的所有组件广播信息。这两种方法都不甚理想。

纯组件模型可以在真实游戏项目实施，或许也有成功的实例。这类模型都有其优缺点，再次，我们不能清楚确定这些设计是否比其他设计更好。除非读者是研发团队的成员，那么应该会选择自己最方便且最有信心的架构，而该架构又是最能配合开发中的游戏的。

14.2.2 以属性为中心的各种架构

惯用面向对象语言的程序员，常会自然地使用对象属性（数据成员）和行为（方法、成员函数）去思考问题。这称为**以对象为中心的视图**（object-centric view）：

- 对象1
  - 位置= $(0, 3, 15)$
  - 定向= $(0, 43, 0)$

- 对象2
  - 位置= $(-12, 0, 8)$
  - 血量= 15
- 对象3
  - 定向= $(0, -87, 10)$

然而，我们也可以属性为中心来思考，而不是对象。我们先定义游戏对象可能含有的属性集合，然后为每个属性建表，每个表含有各个对象对应该属性的值，这些属性值以对象唯一标识符为键。这称为**以属性为中心的视图**（property-centric view）。

- 位置
  - 对象1 = $(0, 3, 15)$
  - 对象2 = $(-12, 0, 8)$
- 定向
  - 对象1 = $(0, 43, 0)$
  - 对象3 = $(0, -87, 10)$
- 血量
  - 对象2 = 15

以属性为中心的对象模型曾成功地应用在许多商业游戏中，包括《杀出重围2（Deus Ex 2）》及《神偷（Thief）》系列。延伸阅读（14.2.2.5节）提供这些项目中对象模型设计的更多细节。

相对于对象模型，以属性为中心的设计更类似关系数据库。每个属性像是数据库表的一列（或独立的表），以游戏对象的唯一标识符为**主键**（primary key）。当然，在面向对象模型中，对象不仅以**属性**定义，还需要定义其**行为**。若我们有了属性的表，如何实现行为呢？各游戏引擎给出不同的答案，但最常见的方法是把行为实现在两个地方：（a）在属性本身，及/或（b）通过脚本。我们进一步探讨这两种做法。

14.2.2.1 通过属性类实现行为

每种属性可以实现为**属性类**（property class）。属性可以是简单的单值，如布尔值或浮点数，也可以复杂到如一个渲染用的三角形网格，或是一个人工智能“脑”。每个属性类可以通过硬编码方法（成员函数）来产生行为。某游戏对象的整体行为仍是由其全部属性的行为结集而得。

例如，若游戏对象含有Health（血量）属性的实例，该对象就能受损，并最终被毁或被杀。对于游戏对象的任何攻击，Health对象都能扣减适当的血量作为回应。属性对象也可以与该游戏对象中的其他属性对象交流，以产生合作行为。例如，当Health属性检测并回应了一个攻击，它可以发一个消息给AnimatedSkeleton（带动画的骨骼）属性，从而令游戏对象播放一个合适的受击动画。相似地，当Health属性检测到游戏对象快要死去或被毁，它能告诉RigidBodyDynamics（属性）触发物理驱动的自爆，或是“布娃娃”模拟。

14.2.2.2 通过脚本实现行为

另一选择，是把属性值以原始方式储存于一个或多个如数据库的表里，然后用脚本代码实现对象的行为。每个游戏对象可能有一个名为ScriptId的特殊属性，若对象含该属性，那么它就是用来指定管理对象行为的脚本部分（指脚本函数，若脚本支持面向对象则是指脚本对象）。脚本代码也可能用于回应游戏世界中的事件。14.7节将会谈及更多有关事件系统的细节，而14.8节则会讨论有关游戏脚本语言。

在一些以属性为中心的引擎里，核心属性是由工程师硬编码的类，但引擎还会提供一些机制给游戏设计师及程序员，以完全使用脚本实现一些新的属性。这种方法曾成功应用到一些游戏，例如《末日危城（Dungeon Siege）》。

14.2.2.3 对比属性与组件

笔者需要交待一下，14.2.2.5节所参考的文章中，许多作者使用“组件”一词去代表笔者在此所指的“属性对象”。在14.2.1.4节中，笔者使用“组件”一词指以对象为中心的设计中的子对象，而这个“组件”和属性对象并不怎么相似。

然而，属性对象和组件在很多方面都是密切相关的。在两种设计中，单个逻辑游戏对象都是由多个子对象所组成的。主要的区别在于子对象的角色。在以属性为中心的设计中，每个子对象定义游戏对象本身的某个属性（如血量、视觉表示方式、物品清单、某种魔法能量等）；而在以组件为中心（以对象为中心）的设计中，子对象通常用作表示某底层引擎子系统（渲染器、动画、碰撞及动力学等）。这个区别如此细微，在许多情况下这个区别的存在都几乎无所谓了。读者可称自己的设计为**纯组件模型**（14.2.1.6节），或是**以属性为中心模型**，看你觉得哪一个名称较为合适。但是到了最后，读者应会得到实质上相同的结果——一个由一组子对象所合成而成的逻辑游戏对象，并从这组子对象中获取所需的行为。

14.2.2.4 以属性为中心的设计的优缺点

以属性为中心的方式富有许多潜在优点。它趋向更有效地使用内存，因为我们只需储存实际上用到的属性（即是说，我们不会有一些对象，内含未用的数据成员）。它也更容易使用数据驱动的方式来建模，设计师能轻松定义新的属性，无须重新编译游戏，因为根本不用改变游戏对象类的定义。仅当定义新的属性类型时，才需要程序员的介入（假设属性不能通过脚本定义）。

属性中心设计也可能比对象中心模型更缓存友好，因为相同类型的数据在内存中是**连续**储存的。这是在当今游戏硬件中的常用的优化技巧，因为这些硬件的内存存取成本远高于执行指令和运算[3]。（例如，在PS3上缓存命中失败的成本，等同于执行数千条CPU指令的成本。）把数据连续储存于内存之中，能减少或消除缓存命中失败，因为当我们存取数据数组的某元素时，其附近的大量元素也会被载入相同的缓存线（cache line）之中。此数据布局设计方式有时候称为**数组之结构**（struct of array，SoA），相比更传统的方式为**结构之数组**（array of struct，AoS）。以下的代码片段展示了这两种内存布局方式的区别。（注意，我们并不会完全以这种方式来实现游戏对象模型，此例子是为了展示以属性为中心的设计可以产生连续的类型数组，而不是复杂对象的单个数组。）

```
static const U32 MAX_GAME_OBJECTS = 1024;

// 传统结构之数组（AoS）方式

struct GameObject
{
    U32         m_uniqueId;
    Vector      m_pos;
    Quaternion  m_rot;
    float       m_health;
    // ......
};

GameObject g_aAllGameObjects[MAX_GAME_OBJECTS];
```

[3]译注：此现象称为内存墙（memory wall）。维基百科条目指出，从1986至2000年，CPU每年提速55%，而内存仅为10%。http://en.wikipedia.org/wiki/Random-access_memory#Memory_wall。

```
// 对缓存更友好的数组之结构（SoA）方式

struct AllGameObjects
{
    U32         m_aUniqueId[MAX_GAME_OBJECTS];
    Vector      m_aPos[MAX_GAME_OBJECTS];
    Quaternion  m_aRot[MAX_GAME_OBJECTS];
    float       m_aHealth[MAX_GAME_OBJECTS];
    // ……
};

AllGameObjects g_allGameObjects;
```

以属性为中心的模型也有其缺点。例如，当游戏对象只是属性的大杂烩，就会更难以维系那些属性之间的关系。单凭凑齐一些细粒度的属性去实现一个大规模的行为，并非易事。这种系统也可能更难以除错，因为程序员不能一次性地把游戏对象拉到监视视窗中检查它的属性。

14.2.2.5 延伸阅读

一些游戏业界的杰出工程师曾在各个游戏开发会议上发表过有关属性为中心的架构的简报，这些简报可以通过以下网址取得。

- Rob Fermier，“Creating a Data Driven Engine”，Game Developer's Conference，2002.[4]
- Scott Bilas，“A Data-Driven Game Object System”，Game Developer's Conference，2002.[5]
- Alex Duran，“Building Object Systems: Features，Tradeoffs，and Pitfalls”，Game Developer's Conference，2003.[6]
- Jeremy Chatelaine，“Enabling Data Driven Tuning via Existing Tools”，Game Developer's Conference，2003.[7]
- Doug Church，“Object Systems”，于2003年韩国首尔的一个游戏开发会议发表；会议由Chris Hecker、Casey Muratori、Jon Blow和Doug Church组织。[8]

[4] http://www.gamasutra.com/features/gdcarchive/2002/rob_fermier.ppt
[5] http://www.drizzle.com/~scottb/gdc/game-objects.ppt
[6] http://www.gamasutra.com/features/gdcarchive/2003/Duran_Alex.ppt
[7] http://www.gamasutra.com/features/gdcarchive/2003/Chatelaine_Jeremy.ppt
[8] http://chrishecker.com/images/6/6f/ObjSys.ppt

14.3 世界组块的数据格式

如前所述，世界组块通常同时包含了**静态**和**动态**世界元素。静态几何体可能使用一个巨型三角形网格表示，或是由许多较细小的网格所组合而成。每个网格可产生多个**实例**，例如，一道门的网格会重复地用于组块中所有门。静态数据通常包含了碰撞信息，其形式可以是三角形汤、凸形状集，及/或其他更简单的几何形状，如平面、长方体、胶囊体和球体。静态元素还有**体积区域**（volumetric region），用于侦测事件或勾画游戏中不同地域。另外，静态元素也可能包含人工智能**导航网格**（navigation mesh），这些导航网格是一组线段，勾画出背景几何中角色可行走的**路段**[9]。因为我们已经在之前的章节中讨论过大部分这些格式，我们不会在此再详述。

世界组块里的动态部分包含该组块内游戏对象的某种表示形式。游戏对象以其**属性**及**行为**来定义，而对象的行为则是直接或间接地取决于它的**类型**。在以对象为中心的设计中，游戏对象的类型直接决定要实例化哪一个类（或哪些类），以在运行时表示该游戏对象。而在以属性为中心的设计中，游戏对象的行为是由其属性的行为融合而成的，但其类型仍然决定了哪个对象应含什么属性（另一种讲法是对象的属性定义其类型）。因此，一般对每个游戏对象而言，世界组块数据文件包含了：

- **对象属性的初始值**：世界组块定义了每个对象在游戏世界中诞生时应有的状态。对象的属性数据可储存为多种格式。以下我们会探讨几种常见格式。
- **对象类型的某种规格**：在以对象为中心的引擎中，此规格可能是字符串、字符串散列标识符，或其他唯一的类型标识符。而在以属性为中心的设计中，类型可能会显式储存，或是定义为组成对象的属性集合。

14.3.1 二进制对象映像

要把一组游戏对象储存于磁盘，其中一个方法是把每个对象的二进制映像（binary image）写入文件，映像和对象在运行时于内存中的样子完全相同。这么做，产生对象似乎是极简单的工作。当游戏世界组块读入内存后，我们已获得所有对象已预备好的映像，所以能简单地令它们运作。

嗯，实际上并非如此简单。把“现场”的C++类实例储存为二进制映像，会遇到几个难题。例如需要对指针和虚表做特殊处理，也有可能要为字节序问题交换实例中的数据。（第6.2.2.9节详述了这些技巧。）而且，二进制对象映像并无弹性，难以恰当地对其内容进行修改。游戏性是游戏项目中最充满变数、不稳定的部分，因此，选择能支持快速开发及能健

[9]译注：原文对导航网格的解释不太正确，译者做出补充。

壮地经常修改的数据格式最为明智。所以，二进制映像格式通常并不是储存游戏对象的最佳之选（虽然此格式可能适合更稳定的数据结构，例如网格数据或碰撞几何。）

14.3.2 游戏对象描述的序列化

序列化（serialization）是另一种把游戏对象内部状态表示方式储存至磁盘文件的方法。此方法相比二进制对象技术，往往更可携及更容易实现。要把某对象序列化至磁盘，就需要该对象产生一个数据流，当中要包含足够的细节，供日后重建原本的对象。要从磁盘的数据反序列化[10]至内存时，首先要创建适当的类的实例，然后读入属性数据流，以初始化新对象的内部状态。若序列化数据是完整的，那么以我们所需的用途来说，新建对象应该等同于原本的对象[11]。

有些编程语言原生支持序列化。例如，C#和Java都提供标准机制序列化对象至XML文本格式，以及其反序列化。可惜C++语言并没有标准化的序列化机制。然而，在游戏业行内或行外，也开发了许多成功的C++序列化系统。我们不会在此讨论如何编写C++对象序列化系统的细节，但我们会讨论一下关于数据格式及开发C++序列化系统所必需的几个主要系统。

序列化数据并不是对象的二进制映像。取而代之，序列化数据通常会储存为更方便及更可携的格式。XML是流行的对象序列化格式，因为它既有良好的支持也获标准化，又较易于供人阅读。XML对层次数据结构有非常优秀的支持，这是序列化游戏对象集合时经常需要的。然而，解析XML之慢众所周知，这可能增加世界组块的加载时间。因此，有些游戏引擎采用自定义的二进制格式，解析时比XML快而且紧凑[12]。

把对象序列化至磁盘，以及从磁盘反序列化，通常可以实现为以下两种机制之一。

- 在基类加入一对虚函数，如`SerializeOut()`和`SerializeIn()`，然后在每个派生类实现这两个函数，说明如何序列化该类[13]。
- 实现一个C++类的**反射**（reflection）系统。那么就可以开发一个通用的系统去自动序列化任何包含反射信息的C++对象。

---

[10]译注：原文在此仍使用了序列化（serialize）一词，译者认为用反序列化（deserialize）较为恰当。

[11]译注：对象是否等同（identical）可以是类或应用本身所定义的。反序列化后的对象可以和原来的对象拥有不同的内存布局，甚至加入新的属性，或是不同程序语言/运行时所实现的对象，但在应用逻辑上仍然可认为两者是等同的。这也是序列化较二进制对象映射更具弹性的地方。

[12]译注：现时另一个流行的格式是JSON，可能比XML更为简单而且容易映射至面向对象的数据结构。容许在此宣传一下译者的C++开源库rapidjson（`https://code.google.com/p/rapidjson/`），它支持JSON的解析及生成，并考虑了许多游戏的性能和内存需求，可以用于储存数据，或是用于编写序列化系统。

[13]译注：也可以参考Boost的Serialization模块，它可以选择只为每个类撰写一个函数，同时负责序列化和反序列化。在较简单的情况下可保持DRY规则。

反射是C#及其他一些语言的术语。概括地说，反射数据描述了类在运行时的内容。这些数据所储存的信息包括类的名称、类中的数据成员、每个数据成员的类型、每个成员位于对象内存映像的偏移（offset），此外，它也包含类的所有成员函数信息。若能获取任何一个C++类的反射信息，开发通用的对象序列化系统是挺简单的一回事。

然而，C++反射系统中最棘手的地方在于，生成所有相关类的反射数据。其中一个方法是，使用`#define`对类中每个数据成员抽取相关的反射数据，然后让每个派生类重载一个虚函数以返回该类相关的反射数据。也可以手工地为每个类编写反射的数据结构，又或是使用其他别出心裁的方法[14]。

除了属性信息，序列化数据流中的每个对象总是会包含该**类/类型**的名字或唯一标识符。类标识符的作用是，当把对象反序列化至内存时，用来实例化适当的类。类标识符可以是字符串、字符串散列标识符，或是其他种类唯一标识符。

遗憾的是，C++并没有提供以字符串或标识符去实例化的方法。类的名称必须在编译时决定，因此程序员必须要硬编码类的名称（如`new ConcreteClass`）[15]。为了绕过此语言限制，C++对象序列化系统总是含有某种形式的**类工厂**（class factory）。工厂可以用任何方式实现，但最简单的方法是建立一个数据表，当中把类的名称/标识符映射至一个函数或仿函数对象（functor object），后者用硬编码方式去实例化该类。给定一个类的名称或标识符，我们可以在那个表里简单地查找到对应的函数或仿函数，并调用它来实例化该类。

14.3.3 生成器及类型架构

二进制对象映像和序列化格式都有一个致命要害[16]。这两种储存格式都是由对象类型的运行时实现所定义的，因此世界编辑器需要深入知道游戏引擎运行时实现才能运作。例如，为了令世界编辑器写出由多种游戏对象组成的集合，世界编辑器必须直接链接运行时游戏引擎代码，或是费尽苦心硬编码，以生成和游戏对象运行时完全相同的数据块。序列化数据与游戏对象实现之间的耦合比较低一点，但同样地，世界编辑器不与运行时游戏对象代码链接以使用其`SerializeIn()`及`SerializeOut()`函数，便需要以某种方式取得类的反射信息。

为了解耦游戏世界编辑器和运行时引擎代码，我们可以把实现无关的游戏对象描述抽象出来。对于世界组块数据文件中的每个游戏对象，我们储存多一点数据，这组数据常称为**生成器**（spawner）。生成器是游戏对象的轻量、仅含数据的表示方式，可用于在运行时实例

[14]译注：例如，宏可以用模板取代，也可以像SWIG那样编译头文件（或自定义的文件格式）生成反射数据，也可以改造一些C++编译器，在编译之余生成这些信息。

[15]译注：即不可以这样`char* name = …; BaseClass* b = new name;`。

[16]译注：原文为阿喀琉斯之踵（Achilles heel）。

化及初始化游戏对象。它含有游戏对象在工具方的**类型**标识符，也包含有一个简单键值对表描述游戏对象的属性初始值。这些属性通常包含了模型至世界变换，因为大多数游戏对象都有明确界定的世界空间位置、定向及缩放比例。当要生成对象时，就可以凭生成器的类型来决定实例化哪一个或多个类。然后这些运行时对象通过查表合适地初始化其数据成员。

我们可以设置生成器在载入后立即生成对象，或是休眠等待，直至稍后需要时才生成对象。生成器可以实现为第一类对象（first-class object），令它能有一个方便的功能接口，又能在对象属性以外再储存一些有用的元数据。生成器甚至还有生成对象以外的用途。例如，在《神秘海域：德雷克船长的宝藏》中，设计师采用生成器定义一些游戏中重要的点或坐标轴。我们称这些为**位置生成器**（position spawner）或**定位器生成器**（locator spawner）。定位器在游戏中有多种用途，例如：

- 定义人工智能角色的兴趣点。
- 定义一组坐标轴去令多个动画能完美地同步播放。
- 定义粒子效果或音效的起始位置。
- 定义赛道中的航点（waypoint）。
- 等等。

14.3.3.1 对象类型架构

游戏对象的类型定义了其属性和行为。在基于生成器设计的游戏世界编辑器中，游戏对象类型可以由数据驱动的**schema**所表示。schema定义了哪些属性会在创建或修改对象时显露于用户。要在运行时生成某个类型的游戏对象，其工具方的对象类型可以用硬编码或数据驱动的方式，映射至一个或多个需实例化的类型。

类型schema可储存为简单的文本文件，以供世界编辑器读取，并可供用户检视及编辑。以下是一个schema文件的样子：

```
enum LightType
{
    Ambient, Directional, Point, Spot
}

type Light
{
    String          UniqueId;
    LightType       Type;
    Vector          Pos;
    Quaternion      Rot;
```

```
    Float           Intensity : min(0.0), max(1.0);
    ColorARGB       DiffuseColor;
    ColorARGB       SpecularColor;
    ...
}

type Vehicle
{
    String          UniqueId;
    Vector          Pos;
    Quaternion      Rot;
    MeshReference   Mesh;
    Int             NumWheels : min(2), max(4);
    Float           TurnRadius;
    Float           TopSpeed : min(0.0);
    ...
}
```

此例子带出了几个重要细节。读者可以注意到，每个属性除了有名称，还定义了其**数据类型**。这些之中有简单的类型，如字符串、整数、浮点数，也有一些特殊的类型，如矢量、四元数、ARGB颜色，也有一些是对特殊资产类型（如网格、碰撞数据等）的参考。在此例子中甚至有机制定义列举类型，如`LightType`。另一个细微之处在于，对象类型的schema同时对世界编辑器提供一些额外信息。有时候属性的数据类型暗示了它需要哪种GUI控件，例如字符串一般会使用文本框来编辑，布尔值则使用复选框，而矢量则会使用3个对应于x、y、z坐标的文本框，或是特别为三维矢量编辑而设计的GUI控件。schema也可以设置一些元信息供GUI所用，例如，整数及浮点数属性的最小值和最大值、下拉组合框中可选的项目等。

有些游戏引擎容许对象类型schema采用继承，和类的继承相似。例如，所有游戏对象需要知道其**类型**，并对应一个**唯一标识符**，以便在运行时和其他游戏对象区分。这些属性可以在顶级schema中指定，其他schema则可以继承这个顶级schema。

14.3.3.2 属性默认值

读者可以想象得到，典型游戏对象schema中的属性数量可以增长至很多。那么，游戏设计师在游戏世界中放置每个游戏对象类型的实例时，便需要为它们设置大量的数据。在schema中为大量属性定义**默认值**（default value），对此设置实例属性有极大的帮助。设置默认值以后，设计师就能轻易地放置游戏对象类型的“寻常”实例，但仍然可以按需为某些实例微调。

然而，改变某属性的默认值会造成问题。例如，游戏设计师原本是希望兽人的HP为20。经过多个月的制作后，团队决定要把兽人的HP默认值调整为30。那么若不做修改，新放置的兽人便有30点HP。但之前已经放置在游戏世界组块里的兽人要怎么办？我们要搜寻所有之前创建的兽人，并把其HP值手工改为30吗？

理想地，我们希望设计一个生成器，可以自动地把改动了的默认值散布至所有现存、未曾覆写该属性的实例。要实现此功能，有一个容易的方法，就是对于和默认值相同的属性值，不储存其键值对。在载入时，若某个属性值不存在，就使用合适的默认值。（这里假设了游戏引擎能取得对象类型schema文件，从中能获取属性的默认值。）在我们的例子中，多数现存的兽人并没有储存其HP的键值对（当然除非我们手动把某些生成器的HP从默认值改为其他数值）。因此，当默认值从20改为30，这些兽人就会自动采用新的数值[17]。

有些引擎容许派生对象类型覆写默认值。例如，schema里的载具类型定义了TopSpeed的默认值为每小时80英里，而Motorcycle派生类可能把该默认值覆写为每小时100英里。

14.3.3.3 生成器及类型架构的好处

把生成器和游戏对象分开实现，其主要优点就是**简单**、**富弹性**和**健壮性**（robustness）。从数据管理的角度来说，处理键值对组成的表，相比管理需指针修正的二进制对象映像，或是自定义的对象序列化格式都简单得多。采用键值对也可为数据格式带来极大的弹性，而且可以健壮地做出改动。若游戏对象遇到预料之外的键值对，可以简单忽略它们。相似地，若游戏对象未能找到所需的键值对，可选择使用默认值。因此，游戏设计师和程序员改动游戏对象类型时，键值对的数据格式仍可以极健壮地配合。

生成器也简化了游戏世界编辑器的设计和实现，因为世界编辑器仅需要知道如何管理键值对及对象类型schema。它不需要与游戏引擎运行时共享代码，并且和引擎实现的细节维持非常松的耦合。

生成器和原型（archetype）令游戏设计师及程序员拥有高度弹性及强大力量。设计师可以在世界编辑器中定义新的游戏对象类型schema，过程中无须或只需少许程序员的介入。而程序员可以按自己的时间表实现运行时的对象。程序员无须为了防止游戏不能运行，每次加入新对象类型时便立即实现该对象。无论有没有运行时实现，新对象的数据都可以存于世界组块文件中；无论世界组块中有没有相关数据，运行时的实现都可以存在。

---

[17]译注：这里要注意，如果某兽人的HP已特别手动改为30，然后更改默认值为30再储存时，可能会因为当前值与默认值相同，而不储存其键值对。那么再更改默认值的时候，该兽人也会随着改变，这可能不合乎用户的初衷。另一个做法是，不采用比较的方式来决定是否储存键值对，而是用一个布尔标志表示属性有否被手动修改。如果用户希望属性回归默认值，更改数值之外也要清除此标志。

14.4 游戏世界的加载和串流

为了跨越离线世界编辑器与运行时游戏对象模型之间的鸿沟，我们需要一些方法把世界组块加载至内存，并且在用完后卸载它们。游戏世界加载系统有两个主要功能：管理所需的文件I/O，从磁盘加载游戏世界组块及其他用到的资产至内存中；管理这些资源的内存分配及释放。随着游戏对象在游戏中的出现和消失，引擎也需要管理其**生成**及**销毁**过程。这包括为对象分配及释放内存，以及确保每个游戏对象使用正确的类去实例化。以下几节会探讨游戏世界如何加载，并观察对象生成系统通常如何运作。

14.4.1 简单的关卡加载

最直截了当的游戏世界加载方法，也是全部早期游戏所用的方法，就是仅容许游戏每次加载一个游戏世界组块（又即是关卡）。当游戏开始或过关时，玩家需要等待关卡载入，期间会显示静态或含简单动画的二维加载画面。

这种设计的内存管理也是很简单直接的。如6.2.2.7节提及，堆栈分配器十分适合这种每次仅加载一个关卡的设计。当游戏开始运行时，首先加载全部游戏关卡都需要的资源至堆栈底端。笔者称之为**载入并驻留**（load-and-stay-resident，LSR）数据。我们记下LSR数据完全加载后的堆栈指针。然后，每个游戏世界组块及其相关的网格、纹理、音频、动画等资源都加载于堆栈中LSR数据之上。当玩家完成该关卡后，只需简单地把堆栈指针回复至LSR数据之上。接着就可以在该位置之上加载新关卡了。图14.11展示了此过程。

虽然此设计极为简单，它也有多个缺陷。首先，玩家看到的游戏世界是独立分割的组块，用这个方法不能实现辽阔、连续、无缝的世界。另一问题是在关卡资源数据加载期间，内存中并没有游戏世界，因而玩家会被逼看一些二维加载画面之类[18]。

14.4.2 往无缝加载进发：阻隔室

为了避免出现关卡加载画面，最好是在下一世界组块及相关资源数据加载时，让玩家继续进行游戏。一个简单的实现方式是把游戏世界资产所预留的内存分割为两个等同大小的块。我们可以把关卡A加载至第一块内存，加载后让玩家开始玩关卡A，而同时用串流的文件I/O程序库（即加载代码会在另一线程运行）加载关卡B至第二块内存。此技术的最大问题在于，要把原来可以一次加载的关卡切割成两半。

[18]译注：由于加载过程主要是I/O密集的，加载关卡期间其实可以用CPU做一些事情，例如《猎天使魔女（Bayonetta）》在加载关卡时会让玩家在一个空地随意做操控、出招等练习，这仅使用到本书所指的LSR数据（主角的模型和动画等）。

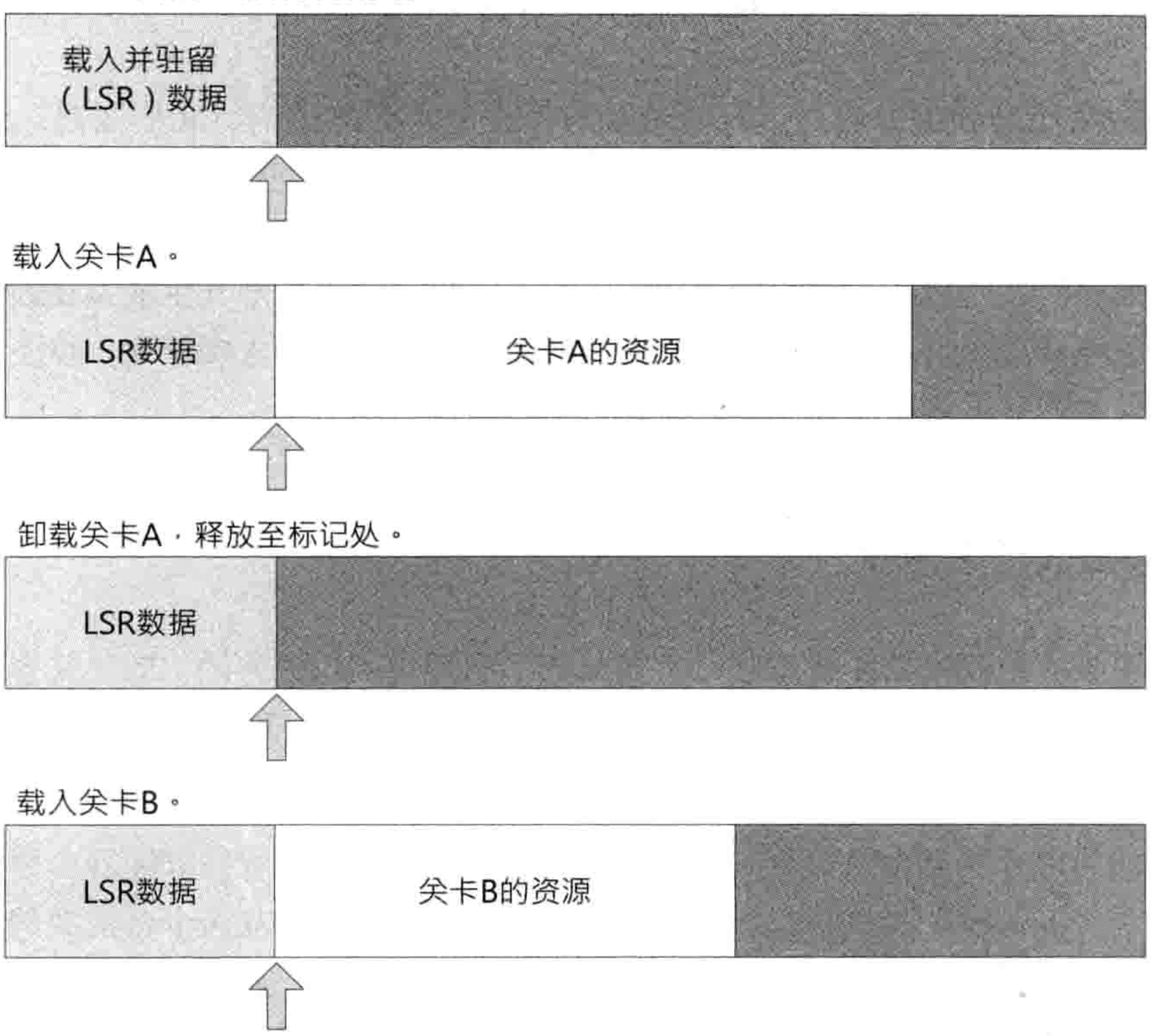


图 14.11：对于同时间只有一个关卡的世界加载系统，基于堆栈的内存分配器非常适合。

我们也可以做出另一个相似的效果，就是把游戏世界内存切割为两个不同大小的块，大的一块用来储存“完整”的游戏世界组块，小的一块只需足够储存一个小型的组块。那个小组块有时称为“阻隔室（air lock）”。

游戏开始时，先加载一个“完整”的组块及一个“阻隔室”组块。玩家在完整组块中前进，然后进入阻隔室。阻隔室内会有门闸或其他障碍物，防止玩家看到或返回之前的完整组块。这时候，就可以卸载之前的完整组块，并加载下一个完整世界组块。加载期间，别让玩家在阻隔室闲下来，可以让玩家简单地走过一条通道，或是执行更有趣的任务，例如，解决一个谜题或与敌人战斗。

能在玩家游玩时同时加载完整世界组块，关键是异步（asynchronous）文件I/O。详情可参考6.1.3节。采用阻隔室有一点值得注意，那就是当游戏开始时我们仍需要显示加载画面，因为那时候内存中并无任何游戏世界可供游玩。然而，当玩家已经进入游戏世界后，借着阻隔室和异步数据加载，就不再需要见到加载画面了。

Xbox的《光环》也采用了近似的手法。当中，大型的世界区域总是以较小的狭窄区域来桥接的。玩《光环》的时候，你会发现每玩5～10分钟就会遇到那些狭小区域，防止玩家折返。PS2的《杰克2（Jak 2）》也使用了阻隔室，其游戏世界的结构是以一个枢纽（主城）连接多个分支地区，枢纽和分支地区之间都有一个细小的阻隔室。

14.4.3 游戏世界的串流

许多游戏设计要求游戏令玩家感觉自己在一个庞大、连续的无缝世界中游玩。理想地，玩家应该不用定时局限在细小的阻隔区域，而是令世界尽量自然地、逼真地逐步显露于玩家。

现代游戏引擎支持这类无缝世界的技术称为**串流**（streaming）。世界串流可以用多种方式实现，其中有两个重要目标：（a）在玩家参与正常的游戏性任务时加载数据；（b）用某些方法管理内存，使玩家在游戏过程中不断加载、卸载数据也不会导致**内存碎片**问题。

近年的游戏机和计算机都配有比上一代大得多的内存，因此现在可以把多个世界组块保持在内存之中。我们可以把内存空间分割为3个同等大小的缓冲区。首先，我们分别加载A、B、C世界组块至这3个缓冲区，然后让玩家在A组块游玩。当玩家进入B组块进行游戏，直至无法看到A组块时，就可以卸载A组块并加载新的D组块至第一个缓冲区。当看不见B的时候，也就可以扔掉它并加载E组块。我们可以循环使用这些组缓冲区，直至玩家到了此连续游戏世界的尽头。

但是，这种粗粒度的世界串流方式有一个问题，那就是它对世界组块的大小设下麻烦的限制。游戏中所有的组块的大小必须大致相同，每个组块需要足够大以填充3个缓冲区之一，而又不能超载。

此问题的解决方法之一是，采用更细粒度的内存分割方式。原来我们会串流较大的内存组块，取而代之，我们把游戏中每个游戏资产（包括游戏世界组块、前景网格、纹理、动画等）都切割为相同大小的数据块。然后我们使用一个以块为单位、基于内存池的内存分配系统（见6.2.2.7节），按需加载及卸载这些资源数据，而无须担心造成内存碎片。这就是《神秘海域：德雷克船长的宝藏》所采用的技术。

14.4.3.1 判断要加载哪些资源

当我们为世界串流采用细粒度的块内存分配器方案时，引擎是如何得知在游戏过程中哪个时间需要加载哪些资源的呢？在《神秘海域：德雷克船长的宝藏》中，我们使用了一个相对简单的**关卡加载区域**（level load region）系统控制加载及卸载资产。

《神秘海域》主要有两个地理上分隔的相邻游戏世界：森林和岛。每个这些世界都存在于独立、一致的世界空间，但它们会切割为多个地理上相邻的组块。每个组块会被一个简单的凸体积所包围，我们称这些体积为**区域**（region），区域之间可能会有重叠的部分。每个区域配有一个表，列出玩家位于该区域时内存应该包含的世界组块。

在某任意时刻，玩家会位于一个或多个这些区域之中。我们可以求出这些区域的组块列表的**并集**，以决定内存中应有的世界组块集合。关卡加载系统定期检查此主控列表，与内存中现有组块比较。若主控列表的组块消失了，就可以卸载内存中的该组块；若列表中出现新的组块，就可以把它加载至任何一个闲置的内存块。我们细心设计关卡加载区域和世界组块，确保玩家永不会看到一个组块因卸载而在眼前消失，也会在玩家第一次见到组块之前有足够的时间加载，使组块能完整地串流至内存。图14.12展示了此技术。

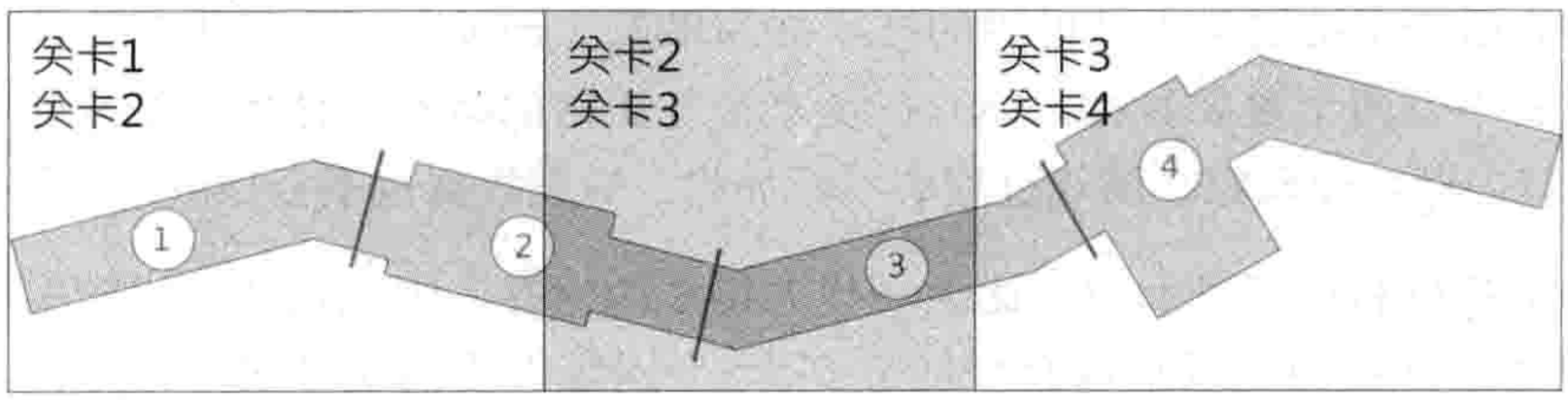


图 14.12: 把游戏世界分割成组块。每个关卡加载区域对应一个组块请求列表，细心整理这些信息以保证玩家永不会看到组块在视野内弹出或消失。

14.4.4 对象生成的内存管理

当游戏世界载入至内存后，我们需要管理世界中动态对象的**生成**（spawning）。多数游戏引擎都含某形式的游戏对象生成系统，负责管理实例化组成游戏对象的一个或多个类，以及当游戏对象不再需要后负责其销毁过程。对象生成系统的重要工作之一是，管理新生成游戏对象的动态内存分配。由于动态分配可能很慢，所以我们必须花一些工夫确保分配过程尽量高效。又由于游戏对象会有不同的大小，为它们做动态分配可能会导致内存碎片，最后形成过早的内存不足情况。其实现时有不同的游戏对象内存管理方法，以下几小节将会探讨当中几个常见的方案。

14.4.4.1 对象生成的离线内存分配

有些游戏引擎为了解决内存分配速度及碎片问题，采用了比较苛刻的方法，这就是简单地完全禁止在游戏途中动态分配内存。这些引擎容许游戏世界组块动态加载和卸载，但载入组块后立即生成所有动态游戏对象，然后就不再创建和销毁游戏对象了。读者可以把这种方

法想象为遵从“游戏对象守恒定律[19]”——加载世界组块后便不能创建或销毁游戏对象。

此技术避免了内存碎片，因为世界组块中的所有游戏对象的内存需求是先验得知（known a priori）且有界的（bounded）。这意味着，游戏对象的内存可以用世界编辑器离线分配，并置于世界组件的数据之中。因此，所有游戏对象，连同游戏世界及其资源，都可以加载于同一块内存中，而且这些游戏对象无异于其他资源，也不会造成内存碎片。此技术的另一优点，是能令游戏准确预测其内存用量情况。游戏世界不会在未预期的情况下，产生大量新游戏对象，造成内存不足。

缺点方面，此法对游戏设计师造成颇严重的限制。为了模拟动态对象生成，我们可以先在世界编辑器中创建一些对象，然后设计它们在加载后维持隐形及休眠状态。之后，这些对象可以启动自己及变成可见的，从而模拟它们在游戏中的“生成”。然而，在世界编辑器里，游戏设计师须预设每个游戏对象类型在游戏世界所需的总数。若设计师们希望有无限供应的补血包、武器、敌人，或其他游戏对象类型，他们可以想一些方法循环使用这些对象，否则就要倒霉了。

14.4.4.2 对象生成的动态内存管理

游戏设计师或较希望选择有真正动态对象生成的游戏引擎。虽然这比静态游戏对象生成更难实现，但也可以用几种不同方式实现。

再一次，我们面对的主要问题是内存碎片。由于不同类型的游戏对象（有时甚至是相同类型的不同实例）占用不同内存用量，不能使用我们最爱的无碎片分配器——池分配器（pool allocator）。由于游戏对象的生成和销毁次序一般来说是不相同的，我们也不能使用堆栈式分配器（stack-based allocator）。庆幸我们还有多个方法对付内存碎片问题，以下探讨几个常见方法。

为每个对象类型设内存池

若每个游戏对象类型的实例能保证占用相同的内存量，我们可以考虑为每个对象类型使用独立的池分配器。实际上，我们只需要为**相同大小的对象**共享一个池分配器，那么相同的对象类型也会用到同一个分配器。

这么做可以完全避免内存碎片，但此方法的限制在于我们要建立很多个池。我们也需要估算每个对象类型有多少实例。若一个池含有太多元素，最终会浪费了内存；若含有太少元素，我们就不能在运行时满足所有的生成请求，那么一些对象生成便会失败。然而，许多商

[19]译注：此处是借用物理上的“质量守恒定律（law of conservation of mass）”。

业游戏都成功地采用了这种内存管理方式。

小块内存分配器

我们可以把每游戏对象类型共享一个池的概念转化为更可行的方式，就是容许游戏对象使用元素大小大于对象大小的池。这样能显著减少内存池的数量，代价是每个池都可能浪费一些内存。

例如，我们可能会建立一组池分配器，每个分配器的元素大小是前一个的1倍，如8、16、32、64、128、256和512字节。我们也可能使用另一个序列以适应某些分配模式，又或是基于游戏运行的分配统计数据来决定序列中的数值。

那么当分配内存的时候，我们首先搜寻元素最小的池，看看其大小是否大于或等于分配对象的大小。我们容许池的元素比对象大，所以会浪费一些内存。作为回报，我们缓和了内存碎片问题。这是一个公平合理的交易。若我们遇到一些内存分配请求，其请求大小比最大元素的池还要大，我们总是可以把请求转发给通用的堆内存分配器。这样的问题不大，因为我们知道大块内存所形成的碎片问题远不及小块内存的严重。

以上所描述的分配器有时候称为**小块内存分配器**（small memory allocator）。对于能放进某个池的分配请求，此分配器能消除它可能形成的碎片。此分配器也可以显著加快小块数据的内存分配，因为此分配器只需进行两次指针改动以删除自由元素链表的元素，这个操作比通用的堆内存分配轻量得多。

内存重定位

另一个消灭内存碎片的方法是直捣问题核心。此法称为**内存重定位**（memory relocation），涉及把已分配的内存块移动至相邻的自由空隙，从而消灭碎片。内存块移动本身是很容易的，但由于我们要移动“现场”的已分配对象，我们需要非常小心地更改指向这些被移动内存块的指针。详情见5.2.2.2节。

14.4.5 游戏存档

许多游戏容许玩家进行进度存档，离开游戏后下次进入游戏时，能回复至与之前完全一样的状态。游戏存档系统（saved game system）与世界载入组件相似，后者也能从磁盘或记忆卡加载游戏世界状态。但两者的需求有些不同，所以通常会把两者作为独立的系统（或只是部分重叠）。

为理解两者的需求差异，不妨简单比较世界组块和游戏存档的差别。世界组块含有世界中动态对象的初始状态，但也包含所有静态世界元素的完整描述。大部分静态信息（如背景网格和碰撞数据）往往消耗许多磁盘空间。因此，世界组块有时候由多个磁盘文件组成，而世界组块所涉及的数据总量通常很庞大。

另一方面，游戏存档必须储存世界中游戏对象的状态信息。然而，它不需要储存从世界组块数据就能得知的重复信息。例如，我们无须把静态几何储存至游戏存档中。游戏存档也不需要储存每个游戏对象的所有状态细节。存档可以完全忽略不影响游戏性的对象。对于其他对象而言，也可能只需要储存部分状态信息。只要玩家不能分辨存档时及读档后的世界状态有何分别（或是那些分别不影响玩家），这就是一个成功的游戏存档系统。因此，游戏存档文件往往较世界组块文件细小得多，而且会更注重压缩及省略数据。尤其是要把大量游戏存档储存至上一代游戏机中的细小记忆卡时，细小的存档文件更显重要。时至今日，虽然游戏机已配置大硬盘，我们仍然最好把存档优化得越小越好。

14.4.5.1 储存点

游戏存档的方式之一是，限制只能在某些指定地点存档，这些地点称为**储存点**（checkpoint）。此方式的好处在于，每个储存点的游戏状态已储存在其附近的世界组块里。无论哪一个玩家走到储存点，这些数据永远不变，因此无须储存在存档中。因此，基于储存点的游戏存档可以极为细小。我们可能只需储存玩家最后到达的储存点的名字，再加上玩家角色的一些当前信息，例如血量、余下多少条命、库存中的物品、武器及其弹药量等。有些基于储存点的游戏甚至不用储存这些信息，因为玩家到达每个储存点都有游戏预设的状态。当然，基于储存点的游戏也有其缺点，就是玩家可能会感到沮丧，尤其是储存点的数量少，或是储存点之间的距离过远。

14.4.5.2 任何地方皆可存档

有些游戏支持一个功能，称为“**任何地方皆可存档**（save anywhere）”。顾名思义，这些游戏容许玩家在游戏过程中几乎任何地方储存游戏的状态。此功能必然导致存档文件显著变大，因为与游戏性相关的每个游戏对象位置和内部状态都需要储存下来，并且在之后载入并回复原来的状态。

在“任何地方皆可存档”的设计中，游戏存档文件基本上储存了如同游戏世界组块的信息，再减去世界的静态组件部分。我们可以为这两个系统采用相同的数据格式，虽然也有可能因为某些原因妨碍我们这么做。例如，世界组块数据格式可能是为弹性而设计的，但游戏存档格式可能需要压缩以令每个存档变得最小。

前面曾提及，缩减游戏存档的数据量的方法之一是，省略一些无关的游戏对象及一些无关的细节。例如，我们不需要记录每个播放中动画的时间索引，也不需要记录每个物理模拟刚体的动量和速度。我们可以依赖人类玩家的非完美记忆，只储存游戏状态的粗略大概。

14.5 对象引用与世界查询

每个游戏对象通常需要某种唯一的标识符，使游戏中的对象能互相区分，并且能在运行时找到所需的对象，也可用该标识符作为对象间通信的目标。唯一对象标识符对工具方同样重要，因为它们能在世界编辑器中用于识别及找出游戏对象。

在运行时，我们总需要多种方法以寻找游戏对象。我们可能希望用对象的唯一标识符、类型，或一组搜寻条件找对象。我们也经常需要做一些基于邻近性的查询（proximity-based query），例如，找出玩家角色10m半径以内的所有敌人。

当我们通过查询找到一个游戏对象时，我们需要以某种方式引用它。在C或C++等语言之中，对象的引用可以使用指针实现，也可以用更精密的方式，如句柄或智能指针。对象引用的生命周期可以有很大差异，它可以是单个函数调用的作用域，也可以到数分钟的时段。

在以下数节，我们首先探讨几个实现对象引用的方法，然后再探索我们实现游戏性时常会用到的查询，以及这些查询可以如何实现。

14.5.1 指针

在C和C++中，实现对象引用最简单直接的方法就是使用指针（或C++的引用类型）。指针很强大，而且也是最简单和直觉的方法。然而，指针带来许多问题。

- **孤立对象**（orphaned object）：理想地，每个对象都有一个**拥有者**，拥有者本身也是一个对象，负责管理其所拥有对象的生命周期，即创建对象并在不需要它的时候把它销毁。然而，指针并不能协助程序员强逼实施这些规则。这有可能造成**孤立**对象，即对象仍占据内存，但它本身已不被需要，或是不被系统内任何其他对象所引用。
- **过时指针**（stale pointer）：删除对象后，理想地，所有指向该对象的指针应该设为空指针。若我们忘记这么做，就会形成过时指针——指针指向以前正常对象所占的内存，但现在该内存块已被释放。若程序中某个部分使用过时指针读 / 写数据，有可能会做成崩溃或不正常的程序行为。过时指针的bug可能难以追踪，因为它们可能在对象销毁后的短暂时间内仍能如常运作。但再过一段时间，若有新对象分配于该内存块，就会改动数据并引发崩溃。

- **无效指针**（invalid pointer）：程序员能自由地储存任何地址于指针中，包括一些完全无效的地址。最常见的问题是对空指针解引用。这些问题可以使用断言宏来防护，在每次解引用前都先检查指针为非空。更坏的情况是，若一些数据被错误当作是指针，那么对它解引用实质上是对随机内存地址进行读 / 写。通常这种情况会导致崩溃，或其他很难除错的严重问题。

许多游戏引擎都大量使用指针，因为指针是实现对象引用最快、最高效并最容易使用的方式。然而，富经验的程序员总是对指针小心翼翼，有些游戏团队会转用更精密的对象引用类型，其原因包括希望采用更安全的编程惯例，或是认为有所必要。例如，若游戏引擎在运行时利用重定位来消灭内存碎片（见5.2.2.2节），就不能使用简单的指针。我们可能需要一种对重定位健壮的对象引用类型，或是需要手动修正每个指向重定位内存块的指针。

14.5.2 智能指针

智能指针（smart pointer）是一个小型对象，行为与指针非常接近，而它的目的是规避原始C/C++指针所衍生的问题。基本上，智能指针含有一个原始指针的数据成员，并提供一组重载运算符以令智能指针的行为在大多数情况下和原始指针一样。指针可以解引用，因此我们需要重载*和->运算符返回所需的地址。此外，指针也有算术运算，因此也需要重载+、 -、 ++和--运算符。

因为智能指针本身是对象，它能含有额外的元数据，又可以加入额外的操作步骤，这些步骤普通指针是做不到的。例如，智能指针可能含有信息，说明其指向的对象是否已被销毁，若已被销毁就可以返回NULL地址。

智能指针也可以帮助管理对象生命周期，方法是通过与其他智能指针合作来判定对象的引用个数。此技术称为**引用计数**（reference counting）。当指向某对象的智能指针个数降至零，我们就能知悉已经不再需要该对象，可以自动地销毁该对象。此做法能令程序员不用担心对象的拥有权及孤立对象。

智能指针也有其问题。首先，智能指针容易实现，但却极难完全无误。智能指针需要处理多种情况，但C++标准库所提供的`std::auto_ptr`类却普遍认为不足够应付很多情况。Boost C++模板库提供6个不同种类的智能指针。

- `scoped_ptr`：指向单个对象且该对象只有一个拥有者的指针。
- `scoped_array`：指向一组对象且那组对象只有一个拥有者的指针。
- `shared_ptr`：指向一个对象的指针，该对象的生命周期由多个拥有者共享。
- `shared_array`：指向一组对象的指针，该组对象的生命周期由多个拥有者共享。

- weak_ptr：指向一个对象的指针，但它不拥有该对象，也不会自动销毁该对象。（该对象的生命周期须由一个shared_ptr管理）。
- intrusive_ptr：其实现引用计数的方法是，假设指向的对象会维护该引用计数。侵入式指针非常有用，因为它们所占的空间和原始C++指针相同（因为它本身不储存引用计数相关的信息），另一个原因是它们能直接地从原始指针建构。

正确地实现智能指针类可能是一个艰巨的任务，读者看一看Boost的智能指针文档[20]便能了解其难度。当中要解决多个问题。

- 智能指针的类型安全性。
- 令智能指针可以使用不完整的类型[21]。
- 在异常（exception）出现时保持正确的智能指针行为。
- 运行时的成本可能很高。

笔者曾参与一个项目，该项目尝试实现自己的智能指针，直至项目结束前我们都在修正许多不同的恶心bug。笔者个人建议，尽量远离智能指针，就算必须使用它们，也要用一个成熟的实现，例如Boost，而不要尝试自己开发。

14.5.3 句柄

句柄（handle）在很多方面的行为都像智能指针，但它更易实现并且较少出现问题。基本上，句柄就是某全局**句柄表**（handle table）的整数索引，而句柄表则是储存指向引用对象的指针。要创建一个句柄，只需简单地用对象的地址去搜寻句柄表，并把结果索引储存在句柄中。要对句柄解引用，则只需把句柄作为索引去读取句柄表，并把该位置的指针解引用。图14.13说明了此数据结构。

通过加入句柄表这个简单的间接层，就能令句柄比指针更安全及更具弹性。若要删除一个对象，只须简单把句柄表中对应的记录清空。这会令所有现存该对象的句柄都立即且自动地变成空引用。另外，句柄也支持内存重定位。当对象在内存中重定位，其旧地址可以在句柄表中找到记录并进行相应更新。再次，所有现存引用该对象的句柄也能自动地更新。

虽然句柄可以实现为原始整数，然而，句柄表的索引通常会包装成一个简单类，以提供更方便创建句柄和解引用的接口。

---

[20] http://www.boost.org/doc/libs/1_54_0/libs/smart_ptr/smart_ptr.htm

[21] 译注：即是说所指向的类型只有前置声明。

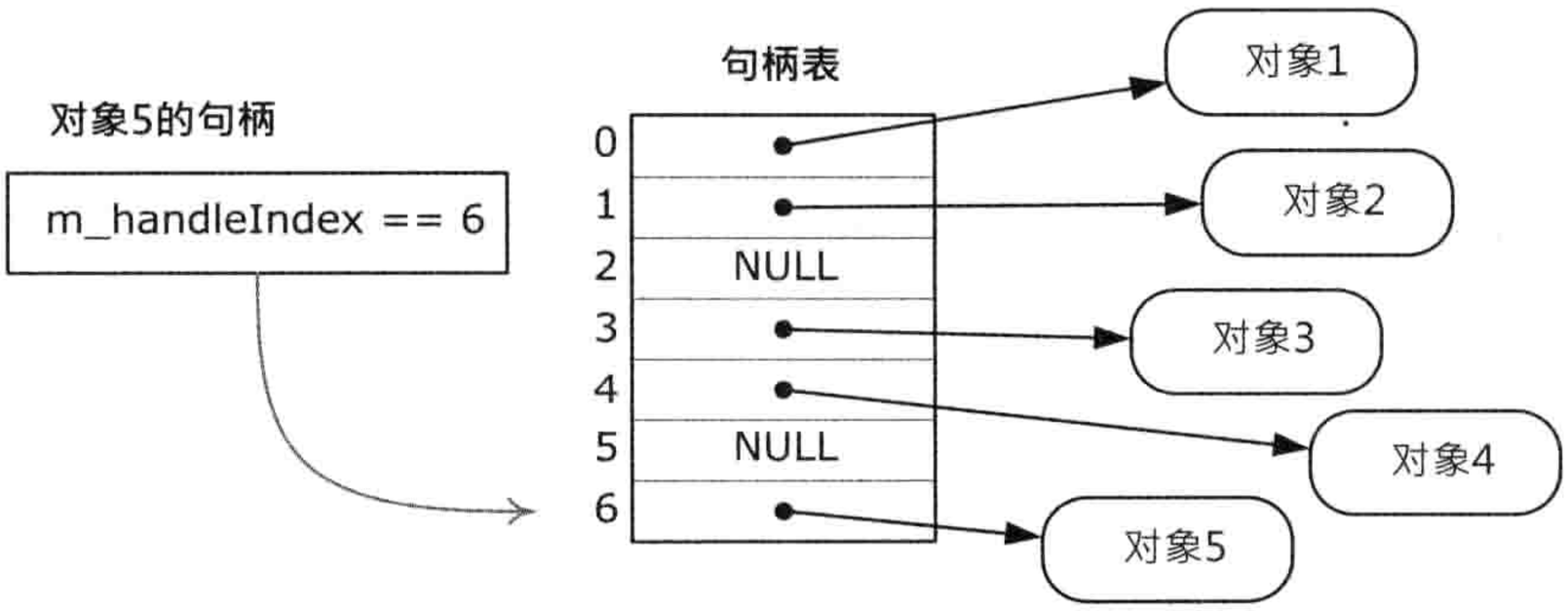


图 14.13: 句柄表含有原始指针。句柄仅是此表的索引。

句柄会有机会引用过时对象。例如，假设我们为对象A创建了一个句柄，该句柄占用了句柄表第17条记录。之后，该对象被删除了，所以第17条记录也设为空指针。再之后，有一个新的对象B被创建，恰巧它也占用句柄表第17条记录。那么所有原来引用对象A的句柄就突然变成引用对象B（而非空值）。几乎肯定这不是我们想要的行为。

过时对象的简单解决方案之一是，在每个句柄中加入唯一的对象标识符。那么，当创建引用对象A的句柄时，该句柄不仅含有记录索引17，也同时储存了对象标识符“A”。当对象B在句柄表中取代对象A之时，所有引用对象A的遗留句柄认同它使用索引17，但不认同句柄表中的对象标识符。那么，当对这些引用过时对象A的句柄解引用时，我们就可以返回空值，而不会错误地返回对象B的指针。

以下的代码段落可以示范怎样实现一个简单的句柄类。注意我们也在GameObject中储存了它的句柄索引，那么当要为GameObject创建新句柄时就不用以地址搜寻句柄表了。

```
// 在GameObject类之内，我们储存了唯一标识符
// 为了高效创建新句柄，也储存了对象的句柄索引
class GameObject
{
private:
    // ……
    GameObjectId    m_uniqueId;      // 对象的唯一标识符
    U32             m_handleIndex;   // 供更快地创建句柄

    friend class GameObjectHandle;   // 让它访问id及索引
    // ……

public:
    GameObject()    // 构造函数
    {
        // 唯一标识符来自世界编辑器，或是在运行时动态指定
```

```
        m_uniqueId = AssignUniqueObjectId();

        // 从句柄表中找一个闲置的句柄索引
        m_handleIndex = FindFreeSlotInHandleTable();

        // ......
    }

    // ......
};

// 此常数定义句柄表的大小，以及同时间的最大对象数目
static const U32 MAX_GAME_OBJECTS = ...;

// 这是全局句柄表，只是简单的数组，储存游戏对象指针
static GameObject* g_apGameObject[MAX_GAME_OBJECTS];

// 这是我们的简单游戏对象句柄类
class GameObjectHandle
{
private:
    U32             m_handleIndex;   // 句柄表的索引
    GameObjectId    m_uniqueId;      // 唯一标识符以防过时句柄
public:
    explicit GameObjectHandle(GameObject& object) :
        m_handleIndex(obejct.m_handleIndex),
        m_uniqueId(object.m_uniqueId)
    {
    }

    // 此函数为句柄解引用
    GameObject* ToObject() const
    {
        GameObject* pObject = g_apGameObject[m_handleIndex];
        if (pObject != NULL && pObject->m_uniqueId == m_uniqueId)
        {
            return pObject;
        }
        return NULL;
    }
};
```

此例子的实现虽然是可用的，但其功能并不完整。我们可能要实现复制语意，又或提供额外的构造函数。全局句柄表的每条记录除了包含对象的原始指针，也可以加入额外信息[22]。当然，这里的固定容量句柄表实现并非唯一的可行设计。不同引擎的句柄系统都有些差异。

我们也要注意，全局引用表有另一美好的副作用，这就是它提供了一个现成的活跃游戏对象列表。例如，我们可以通过全局引用表高效地迭代世界中的所有游戏对象。有些情况下也可以把它用于实现其他的查询种类。

14.5.4 游戏对象查询

每个游戏引擎至少要提供几个在运行时搜寻对象的方法，我们称这些方法为**游戏对象查询**（game object query）。最简单的查询种类是用对象的唯一标识符来找出它。然而，真实的游戏引擎需要很多其他种类的游戏对象查询。以下是一些游戏开发者可能需要的游戏对象查询例子。

- 找出玩家视线范围内的所有敌人角色。
- 对某类型的所有游戏对象进行迭代。
- 找出所有血量少于80%的可破坏游戏对象。
- 向所有在爆炸影响半径范围内的游戏对象做出伤害。
- 对子弹弹道或其他抛射体路径中的对象进行由近至远的迭代。

此表可以继续数页，它的内容当然是跟具体游戏的设计有关。

为了提供最有弹性的游戏对象查询，我们可以想象开发一个通用的游戏对象数据库，它可以编写任意的搜寻条件实现任意查询。理想地，我们的游戏对象数据库可以极高效、迅速地完成所有这些查询，并尽量利用到所有可用的软硬件资源。

现实中，弹性和速度是鱼与熊掌，不可兼得。取而代之，游戏团队通常要判断，在游戏开发过程中哪些是可能最常用到的查询类型，并实现专用的数据结构加速这些查询类型。当需要有新的查询类型，工程师可利用现有的数据结构实现这些查询，若速度不能达标就需要开发新的数据结构。以下列举了一些可用于加速某类游戏对象查询的专门的数据结构。

- **以唯一标识符搜寻游戏对象：**游戏对象的指针或句柄可储存于以唯一标识符为键的散列表或二叉查找树。
- **对合乎某条件的所有对象进行迭代：**游戏对象可预先以多个条件排序（假设我们能预先知道所需的条件），并把结果储存在多个链表里。例如，我们可以建立某游戏对象类

[22]译注：例如，可以储存引用计数，创建句柄时加1，删除句柄时减1，计数归零时可以自动删除对象。

型的表，或是维护一个在玩家某半径范围内的所有对象的列表等。

- **搜寻抛射体路径或对某目标点视线内的所有对象：**这种游戏对象查询通常会利用碰撞系统实现。多数碰撞系统会提供一些极快的光线投射功能，甚至能投射其他形状（如球体或任意的凸体积）判断这些形状碰到哪些对象。
- **搜寻某区域或半径范围内的所有对象：**我们可以用一些空间散列数据结构去储存游戏对象。这个结构可能是置于整个世界之中的简单平面栅格，也可以是更精密的方法，如四叉树、八叉树、kd树，或其他基于空间邻近性的数据结构。

14.6 实时更新游戏对象

无论是最简单的还是最复杂的游戏引擎，都须随着时间更新每个游戏对象的内部状态。游戏对象的**状态**（state）可由它的**属性**（attribute）定义（有时候称为**property**，在C++语言中为**成员数据**/data member）。例如，在《Pong》[23]游戏中，乒乓球的状态可以定义为它在屏幕上的(x, y)坐标及速度（速率及运动方向）。因为游戏是动态、基于时间的模拟，一个游戏对象的状态是描述它在**某一刻**的组态。另一个说法是，一个游戏对象的时间概念是**离散的**（discrete），而不是**连续的**（continuous）。但是，我们可以想象游戏对象状态是连续的，然后在引擎中离散地采样（sample）。这样做可以帮助解决一些常见的陷阱。

在以下的讨论中，我们用符号$\mathbf{S}_i(t)$表示对象i在时间t的状态。这里使用矢量表示方式在数学上并不完全正确，但这种表示方式提醒我们，一个对象的状态就好像一个异质（heterogeneous）的n维矢量，它包含了所有不同资料类别的讯息。注意，这里使用的“状态”一词并非**有限状态机**（finite state machine，FSM）里的状态。一个游戏对象可以由一个或多个FSM所组成，但在这种情况下，每个FSM的当前状态只是游戏对象总状态$\mathbf{S}(t)$的一部分。

大多数低阶引擎子系统（渲染、动画、碰撞、物理、声音等）都需要周期性更新，游戏对象也一样。正如第7章所提及的，通常是通过称为**游戏循环**的主循环来更新引擎子系统（或可使用多个线程，每个线程运行一个游戏循环）。差不多所有游戏引擎都在主游戏循环里更新游戏对象的状态，换句话说，它们把游戏对象模型当作另一个需要周期性运行的引擎子系统。

因此，更新游戏对象可视为一个过程，每个对象根据之前的状态$\mathbf{S}_i(t - \Delta t)$决定当前的状态$\mathbf{S}_i(t)$。当所有对象获更新，当前的时间$t$就成为新的之前时间$(t - \Delta t)$。这个过程在游戏运行中不断重复。一般来说，引擎会管理一个至多个**时钟**，其中一个时钟会对应实时

[23]译注：Pong有中文译名《乓》，是1972年由Atrai公司制作发行的一款模拟乒乓球的街机游戏。

(real-time)，而其他时钟可能不对应实时。这些时钟提供绝对时间t给引擎，也可能提供游戏循环中两个迭代的时间差Δt。我们通常会容许更新游戏对象状态的时钟偏离实时。这么做，可以按照游戏设计需求实现游戏对象的暂停、减速、加速，甚至时光倒流。这些功能也能促进游戏调试和开发。

如第1章提及的，游戏对象更新系统对计算机科学来说，是一个**动态**（dynamic）、**实时**（real-time）、**基于代理**（agent-based）的**计算机模拟**（computer simulation）。游戏对象更新系统也和**离散事件模拟**（discrete event simulation）有关（详见14.7节有关事件的内容）。在计算机科学里，这些都是成熟的研究领域，也有许多互动娱乐以外的应用。游戏是基于代理模拟中一个较复杂的应用。将可见到，要在一个动态、互动虚拟环境中，按时间正确地更新游戏对象是出奇地困难。通过学习基于代理及离散事件模拟，游戏程序员可以对游戏对象更新有更多的理解。这些领域的研究员也可能从游戏引擎设计中学到一些东西!

如同所有高阶的游戏引擎系统，每个引擎采用的设计方式都有轻微的（时而巨大的）差异。可是，如前，每个游戏团队都会面对一些常见问题，而某些设计模式总会出现在几乎所有引擎里。本节会研究这些常见问题，以及它们的常见解决方案。谨记，一些游戏引擎有可能会使用非常方案，此外，下文提及的方案未必能解决一些特殊游戏设计所产生的问题。

14.6.1 一个简单（但不可行）的方式

要更新一个游戏对象集合的状态，最简单的方法是遍历那个集合，并调用每个对象的虚函数（命名如`Update`之类[24]）。通常这个遍历是在游戏主循环的一个迭代中执行一次，也就是每**帧**一次。游戏对象的类可提供自定义的`Update()`函数，该函数把类对象的状态更新至下一个离散时刻。调用更新函数时，可传入当前距离上一帧的时间差，使对象可以恰当地考虑已流逝的时间。因此，最简单的`Update()`函数原型可能如下：

```
virtual void Update(float dt);
```

为方便以下讨论，我们假设游戏引擎采用一个庞大的类继承体系，各游戏对象都是当中的某个类的实例。但是，我们也可以把这里的概念延伸至几近任何以对象为中心的设计。例如，要更新一个基于组件的对象模型，我们可以调用该对象的每个组件的`Update()`，或者我们可以对该对象的“枢纽”对象调用`Update()`，让它更新认为适合的相关组件。我们也可以把这些概念延伸到基于属性的对象模型，每帧调用各个属性的`Update()`函数。

[24]译注：有些引擎把该函数命名为`Tick`、`Simulate`、`Think`等，然而如前文所述，不同引擎的做法不完全相同，这类函数的实际用途也有差异，必须注意。

有曰："魔鬼在细节里（devil is in the details）"，所以我们将在此研究两个重要细节：第一，我们如何管理所有对象的集合；第二，Update()函数应该负责什么事情。

14.6.1.1 管理所有对象的集合

游戏引擎通常会利用一个单例（singleton）管理所有活的游戏对象。这个类可能叫作GameWorld 或GameObjectManager。因为游戏进行中可以生产或销毁对象，所以游戏对象集合一般是动态的。一个简单有效的方式是用**链表**，而链表的元素指向对象的指针、智能指针或句柄。有些游戏引擎不容许动态生产或销毁对象，这些引擎便可用固定大小的**数组**而不需要链表。以下可以看到，大部分引擎会采用比链表更复杂的数据结构管理游戏对象。但在目前，为简单起见，我们可以把这个数据结构想象为链表。

14.6.1.2 Update()函数的责任

一个游戏对象的Update()函数主要负责从它之前的状态$\mathbf{S}_i(t-\Delta t)$决定它当前离散时间的状态$\mathbf{S}_i(t)$。这可能牵涉运行该对象的刚体动力学模拟、对动画采样、回应在该时步（time step）里产生的事件等。

大多数游戏对象会和一个或多个引擎子系统互动。这些游戏对象可能要播放动画、渲染图形、发出粒子特效、播放音效、侦测与其他对象/静态几何物体是否碰撞等。这些系统都须按时更新内部状态，一般每帧或数帧更新一次。很简单直觉的想法是于游戏对象的Update()函数里更新这些子系统。例如，考虑以下一个Tank类的虚构更新函数：

```
virtual void Tank::Update(float dt)
{
    // 更新坦克本身的状态
    MoveTank(dt); //移动
    DeflectTurret(dt); // 偏转炮塔
    FireIfNecessary(); // 按需发炮

    // 现在更新低阶的引擎子系统（这不是一个好方法，见下文）
    m_pAnimationComponent->Update(dt);
    m_pCollisionComponent->Update(dt);
    m_pPhysicsComponent->Update(dt);
    m_pAudioComponent->Update(dt);
    m_pRenderingComponent->Draw();
}
```

如果Update()函数如此构成，游戏循环差不多只需要更新游戏对象：

```
while (true)
{
    PollJoypad();

    float dt = g_gameClock.CalculateDeltaTime();

    for (each gameObject)
    {
        // 此虚构的Update()函数更新所有引擎子系统!
        gameObject.Update(dt);
    }

    g_renderingEngine.SwapBuffers();
}
```

以上游戏更新方式看上去不管有多简单，也不是一个商业级引擎的可行方案。以下几节会探讨这个简单方式所产生的问题，并研究解决这些问题的常用方法。

14.6.2 性能限制及批次式更新

大部分低阶引擎系统都有极严竣的性能限制。它们需要处理大量数据，并且要在每帧里尽快完成大量运算。因此，大多数引擎能受惠于**批次式更新**（batched update）。例如，相对于逐个对象更新动画及交错进行其他无关运算（如碰撞侦测、物理模拟及渲染等），把动画组成一个批次更新更高效。

在大多数商业游戏引擎中，主游戏循环会直接或间接更新引擎子系统，而非在每个游戏对象的Update()函数里，以对象为单位更新引擎子系统。当游戏对象需要某个引擎子系统的服务，游戏对象会通过引擎子系统分配该子系统需要的状态。例如，一个游戏对象希望渲染为一个三角形网格，它要求渲染子系统分配一个**网格实例**（mesh instance）以供使用。（一个网格实例为三角形网格的单个实例，拥有该实例在世界空间的位置、方位、缩放、材质数据及其他实例相关的资讯。）渲染引擎能使用对它来说最高效的方式[25]管理网格实例的集合。游戏对象可改变网格实例的属性控制其渲染效果，但它并不直接控制渲染的过程。取而代之，当游戏所有对象完成更新后，渲染引擎就能高效地批次渲染所有可见的网格实例了。

使用了批次式更新的游戏对象（如我们虚构的Tank），其Update()函数会像这样:

[25]译注: 渲染引擎可能使用10.2.7.4节中介绍的数据结构来管理这些实例，如四叉树、八叉树、包围球树、BSP树等。

```
virtual void Tank::Update(float dt)
{
    // 更新坦克本身的状态
    MoveTank(dt);
    DeflectTurret(dt);
    FireIfNecessary();

    // 改变游戏子系统组件的属性，但不在此更新它们
    if (justExploded)
    {
        m_pAnimationComponent->PlayAnimation("explode");
    }
    if (isVisible)
    {
        m_pCollisionComponent->Activate();
        m_pRenderingComponent->Show();
    }
    else
    {
        m_pCollisionComponent->Deactivate();
        m_pRenderingComponent->Hide();
    }
    // 等等
}
```

游戏循环就变成这样：

```
while (true)
{
    PollJoypad();

    float dt = g_gameClock.CalculateDeltaTime();

    for (each gameObject)
    {
        gameObject.Update(dt);
    }

    g_animationEngine.Update(dt);
    g_physicsEngine.Simulate(dt);
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_audioEngine.Update(dt);
    g_renderingEngine.RenderFrameAndSwapBuffers();
}
```

批次式更新带来很多性能效益，包括但不限于：

- **最高的缓存一致性**：批次式更新加强了游戏引擎子系统内的缓存一致性，因为子系统能把各对象的所需数据分配到一个连续的内存区里。
- **最少的重复运算**：可以先执行整体的运算，之后在各对象更新中重用，无须每次在对象中重新计算。
- **减少资源再分配**：游戏子系统经常要在更新期间分配及管理内存及/或其他资源。若某个子系统的更新与其他子系统的更新交错执行，处理每个对象时便须释放及再分配这些资源。若更新是批次式的，则只需每帧为批次中的所有对象分配这些资源一次。
- **高效的流水线**：很多引擎子系统对游戏世界中的每个对象执行近似的运算。当更新以批次式执行时，就可以做一些优化及利用硬件特设的资源。举例，PlayStation 3提供数个[26]称为SPU的高速微处理器，每个处理器有其私有高速内存。但批次处理重画时，在计算一个角色姿势的同时，可以用DMA传输下一个角色的数据到SPU内存。如果对象是独立更新的，这种并行就不能实现。

性能优势并不是使用批次式更新的唯一原因。一些引擎子系统从根本上不能以对象单位进行更新。例如，若一个动力学系统里有多个刚体，进行碰撞决议（collision resolution）时，孤立地逐一考虑对象，一般不能找到满意的解。对象间的相互穿透（interpenetration）一定要分组决议，可使用迭代法或对线性系统求解。

14.6.3 对象及子系统的相互依赖

即使我们不关注性能，当游戏对象**依赖**其他对象时，简单地逐个对象更新仍不可行。例如，一个人物角色抱着一只猫于怀中，为计算猫的骨骼的世界坐标姿势，必先计算该人物的世界坐标姿势。这意味着，要正确运行游戏，游戏对象更新的**次序**是重要的。

引擎**子系统**依赖另一个子系统是一个相关问题。例如，布娃娃（ragdoll）物理模拟系统须与动画系统协同更新。通常，动画系统会先产生局部空间骨骼姿势（local space skeletal pose），这些关节转换（joint transform）会变换到世界空间，然后物理系统会把它们设定为一个和骨骼相似的相连刚体系统。物理系统随时间模拟这些刚体，刚体的模拟结果会反过来设定骨骼的关节。最后，动画计算最终的世界空间姿势及蒙皮矩阵表（skinning matrix palette）。简而言之，更新动画及物理系统需特定的更新次序，以获得正确的结果。在游戏引擎设计中，子系统间的相互依赖是司空见惯的。

---

[26]译注：PS3的Cell芯片上有8个物理SPE，当中一个会在测试过程中锁掉以提高生产良率，一个预留给操作系统使用，所以实际上游戏代码中可使用6个SPE。每个SPE由SPU和记忆流控制器组成。http://en.wikipedia.org/wiki/Cell_(microprocessor)。

14.6.3.1 分阶段更新

要叙述子系统间的相互依赖，可以在游戏主循环中明确地编写代码以说明正确的子系统更新次序。例如，动画系统和布娃娃系统相互作用的代码可编写成：

```
while (true) // 游戏主循环
{
    // ……
    g_animationEngine.CalculateIntermediatePoses(dt);
    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // 包括布娃娃模拟
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
    g_animationEngine.FinalizePoseAndMatrixPalette();
    // ……
}
```

在游戏主循环中，要慎选时机更新游戏对象状态。通常不能简化成每帧每对象调用一次Update()函数。游戏对象可能需要使用多个引擎子系统的中间结果。例如，游戏对象须在动画系统更新前请求播放动画。然而，该对象也可能希望先用程序方式调整动画系统产生的中间姿势，之后才把调整后的姿势送到布娃娃物理系统去，甚至该对象也可能希望在最终姿势及蒙皮矩阵表生成前对姿势做出最后调整。这意味着，每个游戏对象可能需要多次更新，例如，在动画系统生成中间结果的前后、最终姿势生成前等。

很多游戏引擎容许游戏对象在1帧中的多个时机进行更新。例如，一个引擎可能更新对象3次，一次于动画混合前，一次于动画混合后，一次于最终姿势生成前。一个游戏对象类可以编写3个虚函数作为“挂钩”，以达此目的。这种系统的游戏循环可能会是这样的：

```
while (true) //游戏主循环
{
    // ……

    for (each gameObject)
    {
        gameObject.PreAnimUpdate(dt);
    }

    g_animationEngine.CalculateIntermediatePoses(dt);

    for (each gameObject)
    {
        gameObject.PostAnimUpdate(dt);
```

```
    }

    g_ragdollSystem.ApplySkeletonsToRagDolls();
    g_physicsEngine.Simulate(dt); // 包括布娃娃模拟
    g_collisionEngine.DetectAndResolveCollisions(dt);
    g_ragdollSystem.ApplyRagDollsToSkeletons();
    g_animationEngine.FinalizePoseAndMatrixPalette();

    for (each gameObject)
    {
        gameObject.FinalUpdate(dt);
    }

    // ……
}
```

游戏对象可按需增加更多更新阶段。但要小心，因为每次遍历所有游戏对象并调用各对象的虚函数，其开销可能很高。而且，非所有游戏对象都需要所有更新阶段，遍历不需要某个阶段的对象纯粹浪费CPU时钟周期。一个降低遍历成本的办法是管理多个游戏对象链表，每个链表代表一个更新阶段。一个对象如需在某个阶段中更新，就加到相应的链表中，以此避免遍历不关注某个更新阶段的对象。

14.6.3.2 桶式更新

当存在**对象间**的依赖时，便要轻微调整上述的阶段式更新技巧。因为对象间的依赖性可能抵触更新次序的规则。例如，想象对象A手持着对象B。假设**完全**更新（包括产生最终世界空间姿势及蒙皮矩阵表）对象A之后才能更新对象B，这就抵触了动画系统批次更新所有游戏对象动画以达至最高吞吐量的原则。

对象间的依赖可以想象为多个依赖树（dependency tree）组成的树林（forest）。没有父的游戏对象（即没有其他游戏对象依赖它）被视为树林中的一棵树的根。直接依赖根对象的游戏对象，就成为树林中第1阶度的子节点；直接依赖第1阶度子节点的对象，就成为第2阶度的子节点；以此类推，如图14.14所示。

为了解决更新次序的抵触问题，一个可行方案就是把对象编集成独立的群组。因为没有更好的术语，笔者称这些群组为**桶**（bucket）。第1个桶由树根的对象所构成；第2个桶由第1阶度子节点的对象所构成；第3个桶由第2阶度子节点的对象所构成；以此类推。先为第1个桶执行完整的游戏对象及引擎子系统更新，当中包含所有更新阶段。之后再执行下一个桶的完整更新，直至所有桶都更新了。

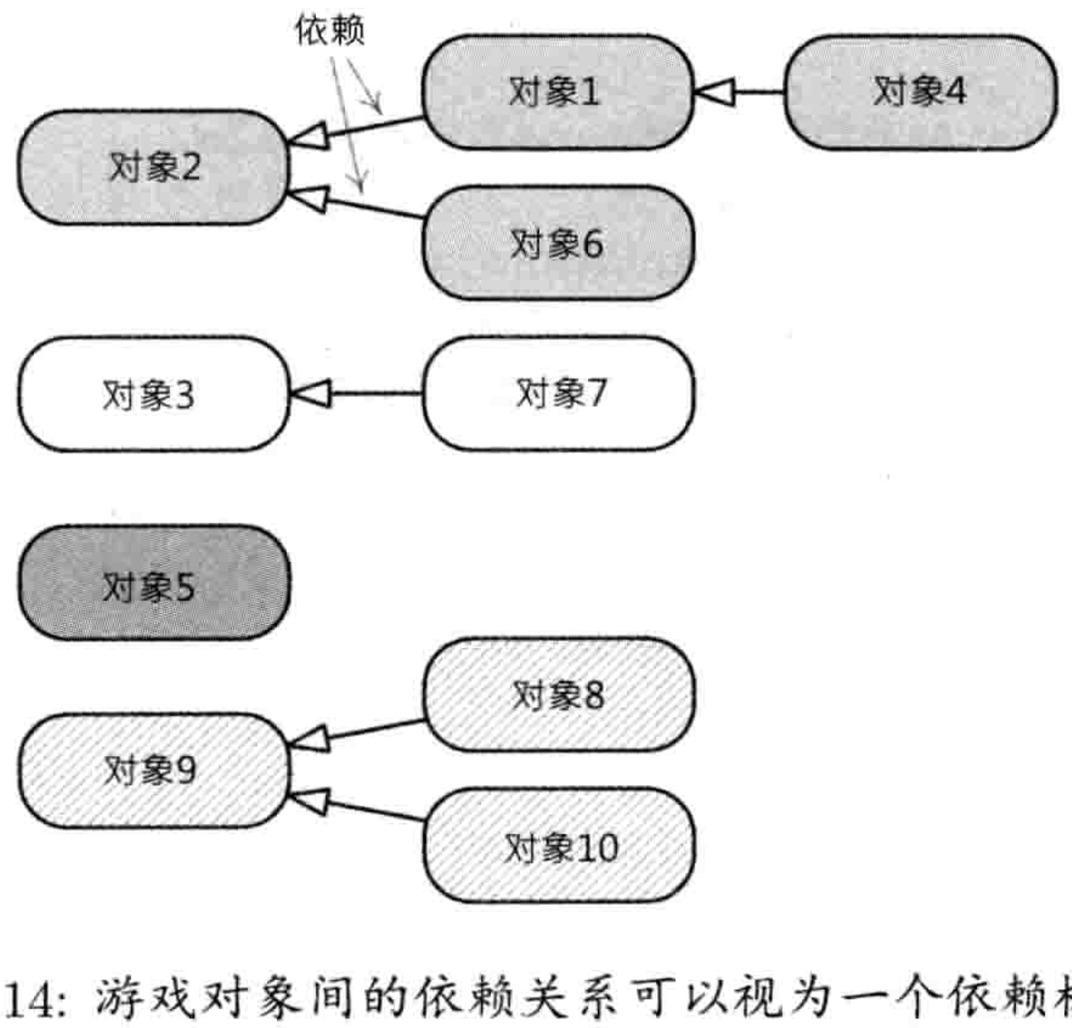


图 14.14: 游戏对象间的依赖关系可以视为一个依赖树林。

理论上，依赖树林中的树深度是无界的。但实践上，深度一般很浅。例如，某游戏的游戏角色可以手持武器，而这些角色又可能站在移动平台或坐在载具上。这个情况下，依赖树林只需要3层，也就是说只需要3个桶，分别对应移动平台/载具、角色、手持中的武器。一个游戏引擎可以明确地为依赖树林的深度设限，可以用固定数目的桶（假设使用桶式更新，当然也可以用其他方式架构游戏循环）。

以下是一个桶式、阶段式、批次式的更新循环示例：

```
void UpdateBucket(Bucket bucket)
{
    // ……

for (each gameObject in bucket)
{
    gameObject.PreAnimUpdate(dt);
}

g_animationEngine.CalculateIntermediatePoses(bucket, dt);

for (each gameObject in bucket)
{
    gameObject.PostAnimUpdate(dt);
}

g_ragdollSystem.ApplySkeletonsToRagDolls(bucket);
```

```
    g_physicsEngine.Simulate(bucket, dt); // 包括布娃娃模拟
    g_collisionEngine.DetectAndResolveCollisions(bucket, dt);

    g_ragdollSystem.ApplyRagDollsToSkeletons(bucket);
    g_animationEngine.FinalizePoseAndMatrixPalette(bucket);

    for (each gameObject in bucket)
    {
        gameObject.FinalUpdate(dt);
    }
    // ……
}

void RunGameLoop()
{
    while (true)
    {
        // ……
        UpdateBucket(g_bucketVehiclesAndPlatforms);
        UpdateBucket(g_bucketCharacters);
        UpdateBucket(g_bucketAttachedObjects);
        // ……

        g_renderingEngine.RenderSceneAndSwapBuffers();
    }
}
```

实践中，有时候事情会比这个更复杂一点。例如，一些物理引擎可能不支持“桶”这个概念，可能是因为它们是第三方SDK，或是因为它们不能以桶式更新。可是，桶式更新对我们很重要，在顽皮狗公司，它被应用于《神秘海域：德雷克船长的宝藏》，并再次应用于《神秘海域2：纵横四海》。这是一个获证实行之有效的方法。

14.6.3.3 对象状态及“差一帧”延迟

现在再重温游戏对象更新，但这次改用游戏对象的个别时间去表达。14.6节定义了$\mathbf{S}_i(t)$状态矢量为游戏对象i在时间t的状态。当更新对象i时，就会利用之前的状态矢量$\mathbf{S}_i(t_1)$去转换为新的状态矢量$\mathbf{S}_i(t_2)$（这里设$t_2 = t_1 + \Delta t$）。

理论上，所有游戏对象的状态是瞬间及并行地从时间t_1更新至时间t_2的，如图14.15所示。然而，实践上，只会逐个对象更新，比如遍历游戏对象并逐一调用它们的更新函数。若在更新中途暂停下来，有部分游戏对象的状态已更新至$\mathbf{S}_i(t_2)$，而其他对象可能还在前一个

状态$\mathbf{S}_i(t_1)$。这意味着，在更新循环里的某刻，询问两个对象当前的时间，结果可能会不同。更甚者，若在更新循环的某刻中断下来，有些对象可能在部分更新的状态。例如，某个对象可能已执行姿势动画混合，却未计算物理及碰撞决议。这可导出一个规则：

> “所有游戏对象的状态在更新循环**之前**和**之后**是一致的，但在更新**途中**是可能不一致的。”

图14.16阐明了这一规则。

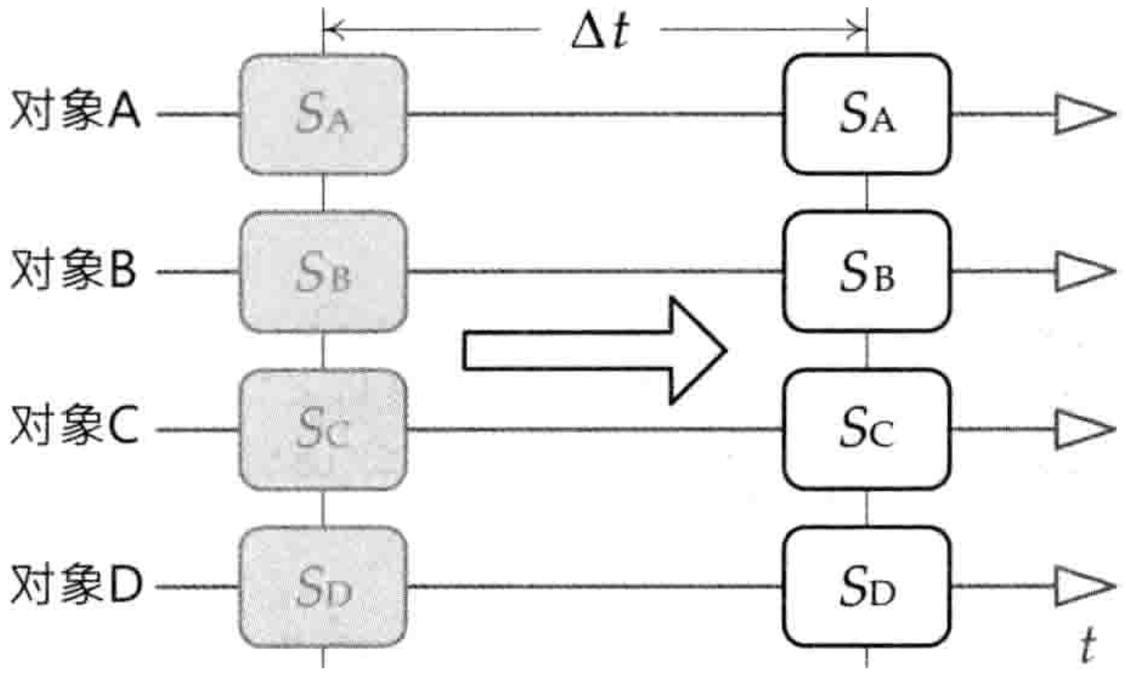


图 14.15: 理论上在每个游戏循环迭代中，所有游戏对象的状态是瞬间及并行地进行更新的。

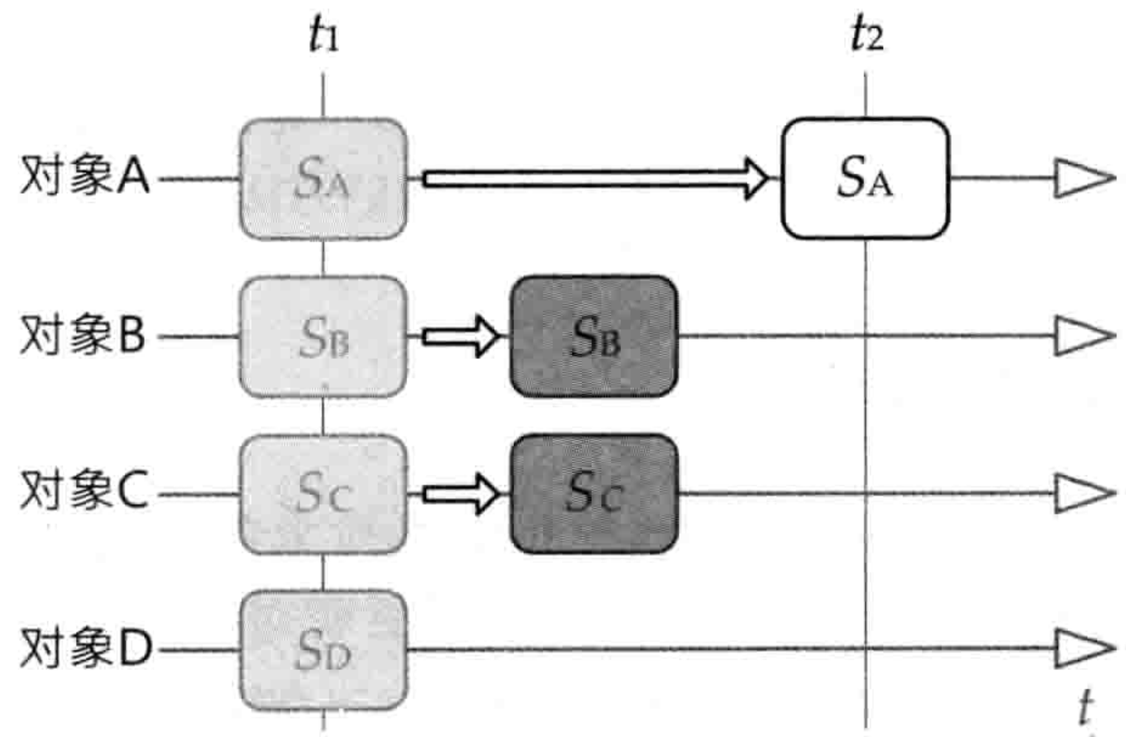


图 14.16: 实践上，游戏对象是逐一更新状态的。这意味着，在更新循环的某些时刻，一些对象（如A）认为时间是t_2而其他对象（如D）则认为仍是t_1。一些对象（如B、C）可能只更新了一部分，所以它们的状态是内部不一致的。这些对象的状态位于t_1和t_2之间。

在更新循环中，游戏对象的不一致状态是混淆和bug的主要来源，即使游戏业界从业人员也如此。更新一个游戏对象时，若它要查询其他对象的状态，就必须解决这个问题。例如，对象B要根据对象A的速度来决定自己于时间t的速度。那么，程序员必须清楚他/她需要的是对象A的之前状态$\mathbf{S}_{\mathrm{A}}(t_1)$，还是新状态$\mathbf{S}_{\mathrm{A}}(t_2)$。若需要新状态，而对象A却未更新，那么这是一个更新次序问题，会导致一类称为**“差一帧”延迟**的bug。在这类bug中，一些对象的状态会延后其他对象1帧，从屏幕上看就是对象之间不同步。

14.6.3.4 对象状态缓存

如14.6.3.2节所述，解决这个问题的方案之一是把游戏对象按桶分类。简单的桶式更新稍微专横地限制了游戏对象能询问哪些对象的状态。如果游戏对象A希望取得对象B的**已更新**状态矢量$\mathbf{S}_{\mathrm{B}}(t_2)$，那么对象B一定要置于**之前已更新**的桶里。同样，如果对象A希望取得对象B**之前**的状态矢量$\mathbf{S}_{\mathrm{B}}(t_1)$，那对象B一定要置于**之后才更新**的桶里。对象A不应查询与自己同属一桶的对象的状态矢量，否则会违反上节的规则，因为那些状态矢量可能只部分更新。

一个能改善一致性的办法如下。更新时，不要就地**覆写**新的状态到原来的矢量，而是保留之前的状态矢量$\mathbf{S}_i(t_1)$，并把新的状态写到另一个矢量$\mathbf{S}_i(t_2)$。这带来两个益处：第一，任何对象都可安全地查询其他对象的之前状态，不受对象更新次序影响；第二，就算是在更新的过程中，它保证永远有一个完全一致的状态$\mathbf{S}_i(t_1)$。以笔者所知，现时该技术并无标准术语，这里称它为**状态缓存**（state caching）[27]。

状态缓存还有一个好处就是，可以通过线性地向前后两个状态插值，得出该时段间任何时刻的状态近似值。Havok物理引擎就是纯粹为了这个原因而保存每个刚体的前后状态。

该技术的缺点是比就地更新状态多耗1倍内存。而且该技术也只能解决一部分问题。虽然之前的状态在t_1是完全一致的，但在t_2的新状态依然可能不一致。然而，明智地使用这一技术还是有帮助的。

14.6.3.5 加上时戳

为改善游戏对象状态的一致性，一个简单低成本的方法就是为对象加上时戳（time stamp）。那么就能轻易分辨游戏对象的状态是在之前还是当前时间。任何查询其他对象的代码，应用断言（assertion）或明确地检查对方的时戳，以确保取得的状态资讯是恰当的。

[27]译注：译者认为，用缓冲（buffer）会比缓存（cache）恰当。因为缓存是利用局部性做优化用途的，而这里是为了储存两个不同时间的数据。可参考维基百科条目http://en.wikipedia.org/wiki/Cache中的“The difference between buffer and cache”一节。

加上时戳并不能解决同一桶内对象的不一致问题。但我们可以设一个全局或静态变量反映目前正在更新哪一个桶。假设每个对象都知道它们置于哪个桶，那么，就可以断言查询对象不应置于现时正在更新的桶，以防查询到不一致的状态。

14.6.4 为并行设计

7.6节介绍了多个并行（parallelism）处理方式，使游戏引擎能受惠于近来有并行处理能力的常见游戏硬件。那么并行性是否影响游戏对象更新的方式呢？

14.6.4.1 使游戏对象模型本身并行

众所周知，使游戏对象模型并行是很困难的，当中有几个原因。通常，游戏对象间会大量相互依赖，游戏对象也可能和多个引擎子系统所产生的数据互相依赖。此外，游戏对象会与其他游戏对象交流（communicate），有时候在更新循环中会多次交流，而交流的模式（pattern）是不可预期且易受玩家输入所影响的。这使游戏对象在多线程中更新变得困难。例如对象间通信须使用线程同步机制，在效能角度上这通常是不容许的。并且，直接读取其他对象的状态矢量，便不能传送游戏对象到副处理器（如PlayStation 3的SPU）的隔离内存去更新。

话虽如此，理论上仍然**可以**并行更新游戏对象。为使它更符合实践，我们应谨慎设计整个对象模型，保证对象不能直接查看其他游戏对象的状态矢量。所有对象交流须使用消息传递（message-passing）。我们需要一个高效的系统传递消息，无论对象是分隔在不同内存还是不同CPU物理核。有些研究采用分布式（distributed）编程语言（如爱立信公司的Erlang[28]）编写游戏对象模型。这些语言提供内置的并行及消息传递功能，也比C/C++之类的语言更高效地处理线程的上下文切换（context switching），这些语言的编程惯用法（idioms）也可以协助防止程序员“打破规则”，使并发式（concurrent）、分布式、多代理（multiple agent）的设计恰当及高效。

14.6.4.2 与并发的引擎子系统接口

虽然尖端的并发分布式对象模型在理论上可行，并且是非常有趣的研究领域，但大部分游戏团队都不会使用。相反，大部分游戏团队仍然使用单线程的游戏对象及旧式的游戏循环。他们把注意力集中于使游戏对象依赖的许多低阶引擎子系统并行化。这么做会有更大的成本效益，因为相比游戏对象模型，低阶引擎子系统才是性能关键。这是由于低阶引

[28] http://www.erlang.org

擎子系统必会在每帧处理高容量的数据，而游戏对象模型所需的CPU资源通常比较小。这是80-20规则的实际应用例子。

当然，使用单线程的游戏对象模型并不代表游戏程序员可以完全忽略并行问题。游戏对象模型仍然要和引擎子系统互动，而引擎子系统本身是与游戏对象模型并发运行的。程序员要改变思维，避免一些过去在并行处理世代之前行之有效的编程范式，并采用新的范式。

最重要的思想改变，可能是程序员要用**异步**（asynchronous）的思维。如7.6.5节所提及，若游戏对象要执行耗时的操作，它应避免使用**阻塞型**（blocking）**函数**，这种函数直接在调用方的线程上下文中工作，即暂停了调用方的线程运行直至工作完成。取而代之，请求执行巨大或昂贵的工作时，应尽量调用**非阻塞型**（non-blocking）**函数**，这种函数把工作请求发送至其他线程、核心或处理器去执行，发送请求后这种函数立即返回至调用方，不等待工作完成。主游戏循环便可以继续处理其他不相关的工作，包括更新其他游戏对象，而原来的对象就继续等待。在这帧稍后时刻或下一帧，该对象才会获得刚才请求工作的结果，并利用它来继续处理。

另一个游戏程序员要改变的思维是批处理（batching）。如14.6.2节提及，把相近的任务集合成批次集中处理，会比独立地逐个任务执行更高效。批处理也能应用在游戏对象状态更新。举例，若一个游戏对象为不同原因，要投射100条光线到碰撞世界，最好能把那些光线排成队，再以批次执行。若一个现有引擎要为并行翻新，便可能要重写代码把单独的请求改为批次式请求。

要把同步非批次代码改为异步批次形式，最棘手的是决定在游戏循环中（a）**何时启动**请求及（b）**何时等待**并使用请求的结果。为此，程序员可问自己以下问题。

- **请求能在多早启动？**越早启动请求，就越有机会能在实际需要结果时完成工作。这样，主线程就不会闲着等待异步请求的结果，优化CPU的使用率。因此，我们应该分析每个请求，找出能有足够资讯启动它的帧内最早时刻，并在那一刻启动它。
- **在需要请求的结果前可以等多久？**或许我们可以等到更新循环稍后的地方继续执行下半部分。或许我们可以容忍1帧的滞后，并使用上一帧的结果更新本帧的对象状态。（有些子系统如人工智能，甚至能容忍更长的滞后时间，因为它们只需要每隔几秒更新一次。）很多情况下，只要些少思考、一点代码重构及一些额外的中间数据缓存，就能把请求结果的时机推迟至帧内稍后的时间。

14.7 事件与消息泵

游戏本质上就是事件驱动的。**事件**（event）是游戏过程中发生、希望关注的事情。发生爆炸、玩家被敌人看见、拾取补血包等都是事件。游戏通常需要一些方法做两件事——当事件发生时通知关注该事件的对象，以及安排那些对象回应所关注的事件，后者称为**事件处理**（event handling）。不同类型的游戏对象会以不同方式回应事件。个别类型的游戏对象处理事件的方式是其行为的关键特征，重要性如同对象在没有外来输入时的状态改变方式。例如，《乒（Pong）》中球的行为可分为几部分，一部分受其速度所支配，一部分通过回应球撞到墙/球拍的事件而令球反弹，也有一部分需回应一方玩家的失球事件。

14.7.1 静态函数类型绑定带来的问题

为了通知游戏对象一个事件已发生，最简单的方法是调用该对象的方法（成员函数）。例如，当爆炸发生时，我们可以对游戏世界查询，找出爆炸半径范围内的所有对象，并对这些对象调用名为OnExplosion()的虚函数。以下的伪代码说明这种处理方式。

```
void Explosion::Update()
{
    // ……
    if (ExplosionJustWentOff())
    {
        GameObjectCollection damagedObjects;
        g_world.QueryObjectsInSphere(GetDamageSphere(),
                                     damagedObjects);

        for (each object in damagedObjects)
        {
            object.OnExplosion(*this);
        }
    }
    // ……
}
```

对OnExplosion()的调用是**静态函数类型**的**后期绑定**（late binding）例子。函数绑定是指调用函数时会运行哪个函数实现，换言之，就是调用绑定到函数实现。如OnExplosion() 等虚函数，被称之为**后期绑定**。这是指编译器在编译期并不知道将会运行**哪个**函数实现，只有在运行时得知目标对象的类型，才能运行适当的实现。我们称虚函数是**静态类型**，因为给定一个对象类型，编译器能知道应调用哪个实现。例如，

编译器知道，若目标对象是Tank类，就应该运行Tank::OnExplosion()；若目标对象是Crate类，就应该运行Crate::OnExplosion()。

静态函数类型绑定带来的问题在于，它在某程度上降低了实现的弹性。首先，OnExplosion()函数需要所有游戏对象继承自同一个基类。此外，基类必须**声明**该虚函数，即使不是所有游戏对象都能对爆炸做出回应。实际上，使用静态类型的虚函数作为事件处理程序，会导致GameObject基类需要声明游戏中**所有可能出现的事件**！这样做会令创建新事件变得困难，也阻止了以数据驱动方式产生事件，例如用世界编辑工具产生事件。这种方式也无法让某些类或某几个实例仅登记自己希望关注的事件，而不登记其他事件。结果是，游戏中每个对象都知道所有可能发生的事件，即使它对一些事件的回应是不做任何事情（例如，实现空的、不做任何事情的事件处理程序）。

我们真正想要的事件处理程序，是**动态函数类型**的**后期绑定**。有些程序语言原生支持这种功能（例如C#的delegate）。在其他语言中，工程师必须自行实现这种绑定。有许多方案可以解决此问题，但其中部分方案归结到数据驱动方法。换言之，我们把函数调用封装成对象，并把对象传递至运行时的对象，以实现后期绑定的动态类型函数调用。

14.7.2 把事件封装成对象

事件实质上由两个部分组成：**类型**（爆炸、朋友受伤、玩家被发现、拾起补血包等）以及**参数**（argument）。参数为事件提供细节（爆炸会造成多少点伤害？哪个朋友受伤？玩家在哪里被发现？血包能补多少点血？）我们可以把这两个部分封装成对象，伪代码如下所示：

```
struct Event
{
    const U32 MAX_ARGS = 8;

    EventType    m_type;
    U32          m_numArgs;
    EventArg     m_aArgs[MAX_ARGS];
};
```

有些游戏引擎称这些为**消息**（message）或**命令**（command），而不称为**事件**。这些名称强调，本质上，把事件通知对象等于向对象发送消息或命令。

实践上，事件对象通常不止这么简单。例如，我们可能会从一个事件根类派生不同的事件类型。而参数也可能实现为链表或动态分配的数组，以容纳任意数量的参数，并且参数可以是不同的数据类型。

把事件（或消息）封装为对象有许多好处。

- **单个事件处理函数**：由于事件对象把其类型以内部方式编码，任何数量的事件类型都可以表示为单个类（或是继承层次中的根类）的实例。这意味着我们仅需要**单个**虚函数处理**所有**事件类型（如`virtual void OnEvent(Event& event);`）。
- **持久性**（persistence）：事件对象有一点和函数调用不相似，函数的参数在调用返回后就离开了作用域，但事件对象把其类型及参数储存为数据。因此，事件对象含有持久性，可储存于队列，稍后才做处理，也可以复制及广播至多个接收者等。
- **盲目地转发事件**：对象可以转发它收到的事件至另一对象，而不需要知道事件的内容。例如，若载具收到一个“下车”事件，它可以转发至其所有乘客，令他们可以下车，而载具本身可能完全不了解“下车”的意思。

把事件 / 消息 / 命令封装为对象的想法，在计算机科学其他领域中也是寻常事。这种做法不仅在游戏引擎中会使用，也可见于图形用户界面、分布式通信等系统中。著名的“四人组”设计模式著作[17]称这种做法为命令模式（command pattern）。

14.7.3 事件类型

我们有多种方法分辨不同事件类型。在C/C++中最简单的方法是使用一个全局的枚举（enum），把每个事件类型映射至一个唯一整数。

```
enum EventType
{
    EVENT_TYPE_LEVEL_STARTED,
    EVENT_TYPE_PLAYER_SPAWNED,
    EVENT_TYPE_ENEMY_SPOTTED,
    EVENT_TYPE_EXPLOSION,
    EVENT_TYPE_BULLET_HIT,
    // ……
};
```

此方法的优点在于简单及高效（因为整数通常能极快地读 / 写及比较）。然而，它也有3个弊病。首先，游戏中**所有**事件类型都要集中在一起，这可视为破坏封装的一种形式（是好是坏，见仁见智）。第二，事件类型是硬编码的，意味着新的事件类型不可通过数据驱动的方式来定义。第三，枚举仅是索引，它们是次序相关的。若某人意外地在列表的中间加入新的事件类型，其后的事件标识符都会改变，这对于储存在磁盘上数据文件中的事件标识符会造成问题。因此，基于枚举的事件类型系统能在小演示及原型程序中良好运作，却不太可以扩展至真正的游戏规模。

另一个事件类型编码方法是使用字符串。此方法是完全自由形式的，只要想一个名字就能加入新的事件类型。但它有几个问题，包括有较大的机会产生事件名称冲突，也有机会因拼错字而导致不能正常运作，字符串所消耗的内存也较多，而且比较字符串较整数来的慢。为了减少内存和提高性能，可以用字符串散列标识符来代替原始字符串，但这仍然解决不了名称冲突及拼错字问题。虽然如此，基于字符串或字符串标识符的事件系统富极高弹性，而且它们本质上支持数据驱动，所以有许多游戏团队都会考虑是否值得承担使用它的风险。

实现工具时，可以用一些方法降低用字符串表示事件所带来的风险。例如，可以使用一个中央数据库管理所有事件类型的名称，用户通过一个界面加入新的事件至数据库。那么，当加入新名称时就可以自动地检测有否造成名称冲突，可禁止用户加入重复的事件类型。当用户要输入一个已有的事件类型时，可以使用下拉组合框列出已排序的事件类型，而不需要用户记住名称并手工键入。事件数据库也可以储存每个事件类型的元数据，包括用途及正确用法的文档，也可以包含事件所支持的参数数量及类型。这种方法可以非常有效，但我们不应该忘记设置这种系统所需的成本，因为此成本并非微不足道的。

14.7.4 事件参数

事件的参数通常与函数的参数很相似，都是用来提供可能对接收者有用的事件相关信息的。事件参数可以用多种方式实现。

我们可以为每种事件类型从Event基类派生一个独立的类。那么就可以使用硬编码方式将事件参数作为这些类的数据成员。例如：

```
class ExplosionEvent : public Event
{
    float        m_damage;
    Point        m_center;
    float        m_radius;
};
```

另一个方法是把事件参数储存为**variant**的集合。variant是一种数据对象，可储存多于一种数据类型。variant通常会储存当前的数据类型以及数据本身。在事件系统中，我们可能希望参数是整数、浮点数、布尔值或字符串散列标识符。因此在C/C++中，我们可以定义一个类似下面代码的Variant类：

```
struct Variant
{
    enum Type
    {
        TYPE_INTEGER,
        TYPE_FLOAT,
        TYPE_BOOL,
        TYPE_STRING_ID,
        TYPE_COUNT  // 类型总数
    };

    Type        m_type;
    union
    {
        I32     m_asInteger;
        F32     m_asFloat;
        bool    m_asBool;
        U32     m_asStringId;
    };
};
```

Variant的集合可以实现为细小、设固定最大长度（如4、8、16个元素）的数组。这样限制了事件可以传递的参数数目，但可以避开为每个事件动态分配内存的消耗。这种做法尤其在内存有限的游戏机上是一个巨大优势。

Variant的集合也可以实现为动态改变大小的数据结构，如动态数组（如`std::vector`）或链表（如`std::list`）。这样可以比固定大小的实现提供更大的弹性，但会产生动态内存分配的成本。假定每个Variant都占同样的空间，我们可以在此利用池分配器。

14.7.4.1 以键值对作为事件参数

事件参数采用以**索引**为基础的集合，其中一个根本问题是这些参数的意义取决于储存的次序。发送方及接受方都必须理解事件以什么次序储存参数。这可能会导致混淆及bug。例如，我们可能意外地遗漏了一个参数，或是多加了一个。

若我们采用键值对（key-value pair）来实现事件参数，这个问题就可以避免。每个参数都以唯一的键来标识，因此参数能以任何次序储存，也可以忽略可选的参数。参数集合可以实现为闭合或开放式散列表，使用键来做散列；也可实现为键值对的数组、链表或二叉搜寻树。表14.1说明此做法。我们有许多选择，但具体采用哪一个方案并不是很重要，只要该方案能在功能上及性能上满足游戏的需求即可。

键	值	
	类型	
“event”	stringid	“explosion”
“radius”	float	10.3
“damage”	int	25
“grenade”	bool	true

表 14.1: 事件对象的参数可以实现为键值对的集合。使用键可避免依赖次序的问题，因为每个事件参数都能用其键来独一无异地识别。

14.7.5 事件处理器

当游戏对象接收到一个事件/消息/命令，它需要以某种方式做出回应。此过程称为**事件处理**（event handling），并通常实现成称为**事件处理器**（event handler）的函数或脚本段落。（我们稍后会再谈及脚本。）

事件处理器通常是一个原生的虚函数或脚本函数，它能处理所有类型的事件（如函数`OnEvent (Event& event)`）。此情况下，函数通常包含某种`switch`语句或一串if/else-if语句，以处理不同类型的、可能接收到的事件。典型的事件处理器函数大概是这样子的：

```
virtual void SomeObject::OnEvent(Event& event)
{
    switch (event.GetType())
    {
    case EVENT_ATTACK:
        ResponseToAttack(event.GetAttackInfo());
        break;

    case EVENT_HEALTH_PACK:
        AddHealth(event.GetHealthPack().GetHealth());
        break;

    // ......
    default:
        // 未认出的事件
        break;
    }
}
```

另一种方法，是实现一系列处理器函数，每个函数负责一种事件（如OnThis()、OnThat()……）。然而，如前所述，这种会增殖的事件处理器函数可能会很麻烦。

Microsoft Foundation Class（MFC）是一个Windows GUI工具集，它含有一个著名的**消息映射**（message map），可以在运行时把Windows消息绑定至任何非虚或虚函数。这种做法避免了在单个根类中声明所有可能的Windows消息处理器，同时也能避免使用一个巨大的switch语句（在非MFC的Windows消息处理函数通常会这么做）。但可能不值得实现这种系统，使用switch语句的方式既能良好运作而又简单又清晰[29]。

14.7.6 取出事件参数

以上的例子掩饰了一个重要细节——如何从事件的参数表中以类型安全的方法取出参数。例如，我们可能假设event.GetHealthPack()会传回一个HealthPack游戏对象，而该对象提供一个GetHealth()的成员函数。这意味着根Event类得悉补血包的接口（扩展下去，即它也得悉游戏中所有的事件参数类型!）这大概是一个不切实际的设计。在真实的引擎中，Event的派生类可能提供像GetHealthPack()这样的API，以便存取数据。第二种方式是处理器手工地取出数据，并把它们转化为合适的类型。第二种方式会引起类型安全问题，但从实践上来说，这通常不会造成大问题，因为在取出参数之前我们必然先知道事件的类型。

14.7.7 职责链

游戏对象几乎总是依赖于另一些游戏对象。例如，游戏对象通常处于一个变换层次关系之中，这样做可以令一个对象停在另一个对象之上，或是把对象绑定至角色的手中。游戏对象也可能由多个互动的组件所组成，形成以组件对象所组成的星状拓朴或松散地连接的“云”。体育类游戏可能会为每队维护其队员的列表。总括而言，我们可以把游戏类型之间的关系想象为一个或多个**关系图**（relationship graph）（回想表和树都是图的特例而已）。图14.17展示了一些关系图的例子。

在这些关系图中把事件从一个对象传递到另一对象，很多时候是讲得通的。例如，当载具接收到一个事件，它可以把该事件传递至所有乘客，而那些乘客或希望把事件再传递至他们的装备去。当一个多组件对象收到事件时，该对象或许也需要把事件传递至它的各个组件，令每个组件有机会处理该事件。在体育类游戏中，角色收到事件后，也可能要把它传递至他的队员。

[29]译注：C/C++编译器会使用branch table等技巧优化switch-case语句。http://en.wikipedia.org/wiki/Branch_table。

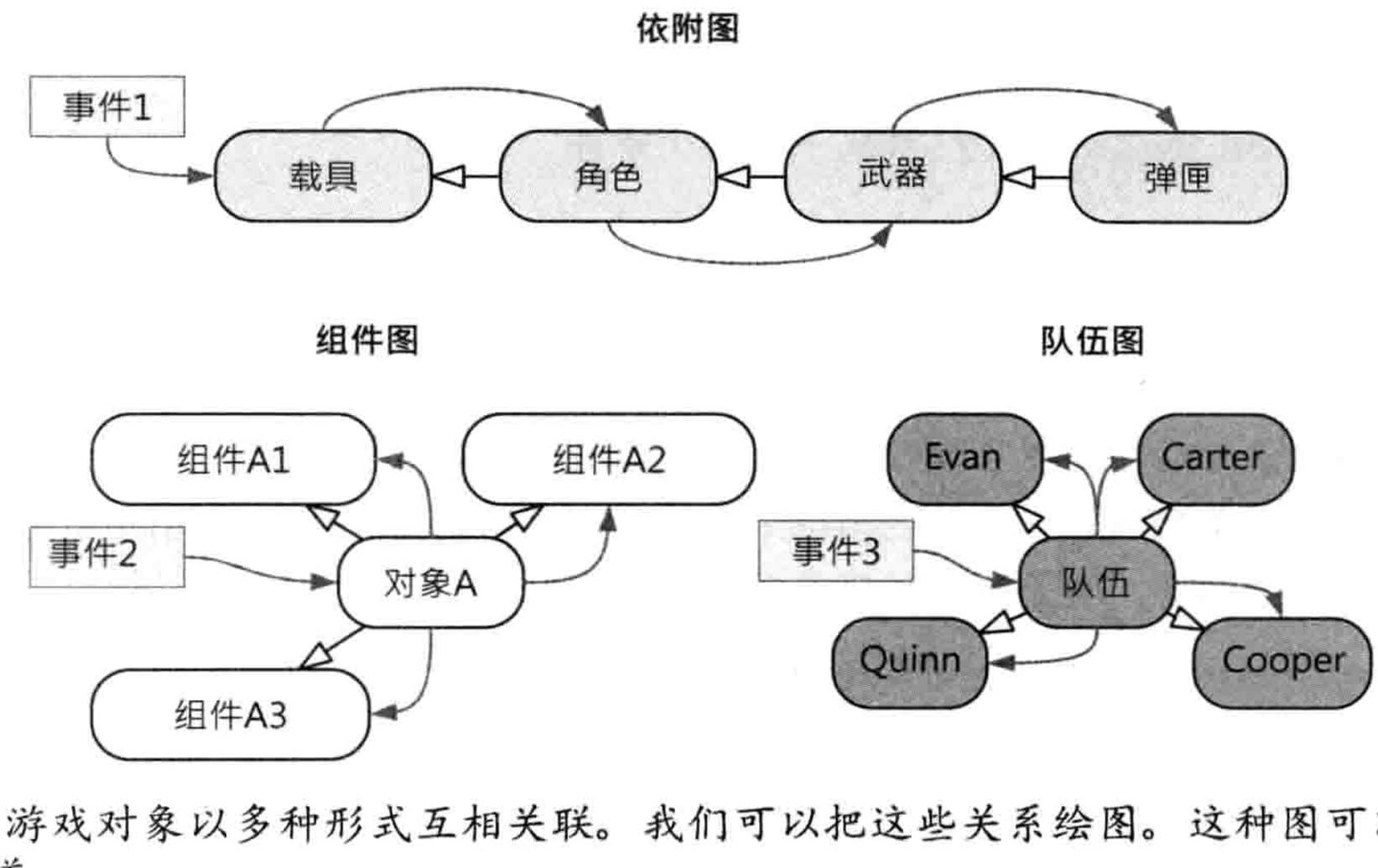


图 14.17: 游戏对象以多种形式互相关联。我们可以把这些关系绘图。这种图可以用作事件的分派通道。

在对象关系图中转发事件，仍面向对象、事件驱动编程中一种常见的设计模式，有时称为**职责链**（chain of responsibility）[17]。通常，事件传递的次序是预先由工程师决定的。事件首先传递至链的首个对象，然后该对象返回一个布尔或枚举值，以表示该对象是否认识并处理该事件。若事件被接收者消化了，事件转发就告一段落；否则，事件就再转发至链中下一个接收者。支持职责链的事件处理器大概是这个样子的：

```
virtual bool SomeObject::OnEvent(Event& event)
{
    // 先调用基类的处理器
    if (BaseClass::OnEvent(event))
    {
        return true;
    }

    // 现在试图自己处理事件
    switch (event.GetType())
    {
    case EVENT_ATTACK:
        ResponseToAttack(event.GetAttackInfo());
        return false; // 可以转发事件给其他对象

    case EVENT_HEALTH_PACK:
        AddHealth(event.GetHealthPack().GetHealth());
        return true; // 我消化了事件，不再转发
```

```
        // ……
        default:
            return false; // 我认不出这个事件
        }
}
```

当派生类覆盖一个事件处理器，若它只是想修改而不是取代原来的回应，那么派生类的处理器或可调用基类的实现。在其他情况下，派生类可以完全取代基类的回应，那么就不需要调用基类的处理器了。这是另一种职责链。

事件转发也有其他用途。例如，我们可能想把事件多播（multicast）至某半径范围内的所有对象（如爆炸）。要实现此功能，可以利用游戏世界的对象查询机制找出对应球体内的所有对象，然后把事件转发给查询结果中的每个对象。

14.7.8 登记对事件的关注

可以很稳妥地说，游戏中大部分对象都不需要接收所有可能的事件。大部分游戏对象类型只会“关注（interest）”很小的事件集合。那么多播或广播事件就是很低效的事情，因为我们需要对一组对象迭代，并逐一调用其事件处理器，即使当中一些对象并不关注某些事件。

为了提高事件处理的效率，我们可以容许对象**登记**它们所**关注**的事件。例如，每个事件类型维护一个链表，内含关注该事件类型的对象；或每个游戏对象维护一个位数组（bit array），当中每一位代表该对象是否关注某种事件。这样做可以避免调用一些不关注该事件的对象。调用虚函数可带来一些不简单的性能损耗，特别是在内存缓存较原始的游戏机之上，因此按对事件的关注情况来过滤对象，可以改善事件多播或广播的效能。

更好的做法是，对于要多播的事件，我们限定游戏对象查询只包含关注该事件的游戏对象。例如，当爆炸发生时，我们向碰撞系统查询，找出影响半径范围里**并且**能回应爆炸事件的对象。整体而言，此做法可能节省一些时间，因为对于不关注发送事件的对象，我们避免了对它们进行迭代。但这种做法是否带来净收益，要视乎查询机制如何实现，并要比较在查询中过滤和多播迭代时过滤的相对成本。[30]

[30]译注：此功能在分布式互动仿真（distributed interactive simulation）中又称为兴趣管理（interest management）。

14.7.9 要排队还是不要排队

多数游戏引擎都会提供一个机制立即处理刚发出的事件。除此以外，有些引擎也会容许把事件排队，留待未来某刻才进行处理。把事件排队有一些很吸引人的好处，但也会增加事件系统的复杂度并造成一些独有的问题。以下数小节将探讨事件排队的优缺点，并在过程中学习如何实现这些系统。

14.7.9.1 事件排队的好处

控制事件处理的时机

我们之前已看到，必须小心以某特定次序更新引擎子系统和游戏对象，才能确保有正确的行为及最大化运行时效率。同等意义上，有些事件类型对于它应该在游戏循环哪个时机处理非常敏感。若所有事件都在发出后立即处理，那么事件处理器函数最终会在游戏循环中不确定且难以控制的时机被调用。使用事件队列延后处理事件，工程师就可以采取措施确保事件在安全及合适的时机获得处理。

往未来投递事件的能力

当发出一个事件时，发送者通常可以设置一个送递时间，例如，我们可能希望该事件在同一帧的稍后时间、次帧或投递后数秒才做处理。此功能等于我们可以往未来投递事件，它有许多有趣的用途。通过往未来投递事件，我们可以实现一个简单闹钟。一些周期性任务（例如，每2s令灯闪一闪）可以实现为投递一个未来事件，其处理器执行了任务后再往未来一个周期后投递相同类型的事件。

为了实现往未来投递事件功能，每个事件进入队列前要指定一个所需的送达时间。仅当目前的游戏时钟等于或超过事件的送达时间，该事件才会被处理。实现此功能的简单方法是把队列中的事件按送达时间排序[31]。那么在每帧中，先检查队列中首个事件的送达时间。若还未到送达时间，就可立即终止处理，因为我们能肯定之后的事件也是未到时间的。我们处理时间已到的事件，直至无合乎条件的事件。以下的伪代码展示了此过程：

```
// 至少每帧调用此函数一次
// 它的任务是分派所有投递时间为现在或过去的事件
void EventQueue::DispatchEvents(F32 currentTime)
{
    // 取得队列中的下一个事件，但不从队列中移除它
```

[31] 译注：也可使用优先队列（priority queue）来实现。

```
Event* pEvent = PeekNextEvent();

while (pEvent &&
       pEvent->GetDeliveryTime() <= currentTime)
{
    // OK，现在从队列移除该事件
    RemoveNextEvent();

    // 把事件分派至接收者的事件处理器
    pEvent->Dispatch();

    // 取得队列中下一个事件（同样不移除它）
    pEvent = PeekNextEvent();
}
}
```

事件的优先次序

就算事件按送达时间在事件队列中排序，若两个或以上的事件都有相同的送达时间，这些事件的处理顺序仍然是有歧义的。这种情况的出现机会可能超出读者所想，因为事件的送达时间通常会量化为整数帧。例如，两个发送者请求事件在“本帧”、“次帧”或“7帧以后”被调度，那么那些事件便可能会有相同的送达时间。

消除这种歧义的方法之一是为事件设置**优先次序**（priority）。当两个事件的送达时间相同，就先处理优先次序较高的一个。实现时只需要在排序时先比较送达时间，若相同再比较优先次序。

若以32位整数来表示优先次序，就能得出约40亿个不同的优先级。也可以限制只有两三个不同的优先级（如低、中、高）。每个游戏引擎的优先级都只需要达到消除真实歧义的数量下限。通常越接近这个下限越好。若有大量的优先级，要了解各种情况哪一个事件先处理，就变得困难了。然而，每个游戏的事件处理系统的需求都不同，读者需要自行决定。

14.7.9.2 事件排队带来的问题

增加了事件系统的复杂度

为了实现事件排队系统，相对于即时处理的事件系统，我们需要更多代码、额外的数据结构、更复杂的算法。复杂度的增长，在游戏开发过程中通常会转化成更长的开发时间、更高的维护及扩展成本。

深度复制事件及其参数

使用即时处理事件的方法时，事件参数所存的数据只需要在事件处理函数（及其所调用的函数）中持续。即是说事件及其参数可以存于任何内存空间，包括调用堆栈。例如，我们可以编写这种函数：

```
void SendExplosionEventToObject(GameObejct& receiver)
{
    // 在堆栈中分配事件参数
    F32     damage = 5.0f;
    Point   centerPoint(-2.0f, 31.5f, 10.0f);
    F32     radius = 2.0f;

    // 在堆栈中分配事件
    Event event("Explosion");
    event.SetArgFloat("Damage", damage);
    event.SetArgPoint("Center", &centerPoint);
    event.SetArgFloat("Radius", radius);

    // 传送事件。这会立即调用接收者的事件处理器
    event.Send(receiver);
    //{
    //    receiver.OnEvent(event);
    //}
}
```

然而，当事件需要排队时，其参数就需要持续至发送者函数作用域之外。这意味着我们必须复制整个事件对象至队列。我们必须使用**深度复制**（deep-copy），即我们不仅要复制事件对象本身，也要复制事件所装载的参数，包括这些参数所持有的对象。深度复制确保不会有仅对发送者作用域数据的悬垂引用（dangling reference），并且容许事件无限期储存。上面的投寄函数例子改为使用排队事件系统后，基本上是差不多的，但读者可以注意下面代码的斜体部分，Event::Queue()函数的实现会比Send()更复杂一点。

```
void SendExplosionEventToObject(GameObejct& receiver)
{
    // 仍然在调用堆栈中分配事件参数
    F32     damage = 5.0f;
    Point   centerPoint(-2.0f, 31.5f, 10.0f);
    F32     radius = 2.0f;

    // 在堆栈中分配事件还是可行的
    Event event("Explosion");
```

```
    event.SetArgFloat("Damage", damage);
    event.SetArgPoint("Center", &centerPoint);
    event.SetArgFloat("Radius", radius);

    // 把事件储存在接收者的队列，供接收者在未来处理
    // 注意事件必须被深度复制，才能加到队列里
    // 因为原来的事件存于堆栈中，在此函数返回后就离开作用域
    event.Queue(receiver);
    //{
    //    Event* pEventCopy = DeepCopy(event);
    //    receiver.EnqueueEvent(pEventCopy);
    //}
}
```

为队列中的事件做动态内存分配

把事件对象深度复制意味着需要动态内存分配。如之前多次提及，游戏引擎并不欢迎动态内存分配，因为它除了有潜在的成本外，也会造成空间碎片。然而，若使用排队事件，我们便需要动态分配事件对象。

如同游戏引擎中所有的动态分配，我们最好能选择一个快速且不会造成碎片的分配器。我们可以选择**池分配器**，但这仅当所有事件对象是相同大小而且其参数也是由相同数量、相同大小的元素类型所组成的时。有些情况的确是这样的，例如，每个参数都使用上面所说的Variant类型。若事件对象及/或其参数的大小是可变的，**小型内存分配器**便可能适用。（记得小型内存分配器维护多个池，每个池有预先指定的分配大小范围。）当设计一个排队事件系统时，记得要细心考虑其动态分配需求。[32]

调试困难

使用排队事件时，事件处理器并不是由发送者直接调用的。因此，与即时事件处理不同，调用堆栈不能告诉我们事件从何而来。我们不能在调试器中的调用堆栈往上检查发送者的状态，甚至该事件发送时的环境情况。因此，调试这些延时事件会比较棘手，若事件会被对象传发的话就更为困难。

---

[32]译注：若事件系统不支持送达时间和优先次序，也即事件总是遵从FIFO，那么有一个简单的方法避免动态内存分配及深度复制，这就是建立一个循环队列（circular queue）储存事件对象及其参数，直接在队列上分配空间及写入数据。若需要支持送达时间和优先次序，可考虑只支持能重定位（relocation）的事件参数类型，如数字字、字符串/散列字符串和句柄，那么每个事件总是表示为循环队列中的内存块，若循环队列因碎片而满溢，就可以把每个事件的内存块搬移，重新成为连续内存（此过程通常称为夯实/compact）。

有些引擎会储存一些调试信息，表示事件在系统中传递的轨迹，但无论怎样仔细，调试没有排队的事件通常都简单得多。

排队事件也会导致一些有趣又难追踪的**竞态条件**（race condition） bug。我们可能需要在游戏循环中各处散布事件调度，以保证事件的处理不会导致讨厌的1帧延迟，仍能保持在该帧内以适当次序更新对象。例如，在更新动画时，我们可能检测到某个动画已放完。这可能会发出一个事件，令处理器可以播放新动画。显然，我们不希望在之前和之后的动画间有1帧的延迟。为了避开这个问题，我们可以先更新动画的时钟（那么就能在这时候检测到动画放完并发出事件），然后进行事件调度（那么处理器就有机会请求播放新动画），最终才开始动画混合（那么就能够处理和显示新动画的首帧）。以下的代码片段说明了这个方法：

```
while (true) // 主游戏循环
{
    // ……
    // 更新（多个）动画时钟。这可能会检测到片段完结，
    // 并发出EndOfAnimation事件
    g_animationEngine.UpdateLocalClocks(dt);

    // 然后分派事件
    // 若有需要，事件处理器可以在此帧开展一个新的动画
    g_eventSystem.DispatchEvents();

    // 最后，混合所有当前在播放的动画
    // （包括本帧较早前播放的新动画）
    g_animationEngine.StartAnimationBlending();

    // ……
}
```

14.7.10 即时传递事件带来的问题

即时传递事件也有其问题。例如，即时事件处理器可能引致非常深的调用堆栈。对象A向对象B传递一个事件，然后B的事件处理器又发出另一个事件，如此下去。支持即时传递事件的游戏引擎可能常会见到如下这种调用堆栈：

```
...
ShoulderAngle::OnEvent()
Event::Send()
Character::OnEvent()
Event::Send()
Car::OnEvent()
```

```
Event::Send()
HandleSoundEffect()
AnimationEngine::PlayAnimation()
Event::Send()
Character::OnEvent()
Event::Send()
Character::OnEvent()
Event::Send()
Character::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::OnEvent()
Event::Send()
Car::Update()
GameWorld::UpdateObjectsInBucket()
Engine::GameLoop()
main()
```

在极端情况下，这么深的调用堆栈有可能会用尽程序的堆栈空间（尤其造成无限循环的事件发送），但此问题的症结在于，每个事件处理器都必须实现为完全**可重入**（re-entrant）函数。这即是说，以递归方式调用事件处理器并不会有任何不良副作用。作为一个人为例子，想象有一个函数，它会把全局变量的值加1。若该全局变量在每帧只能加1，那么这个函数是**不**可重入的，因为多次的递归调用会导致多次使变量加1。[33]

14.7.11 数据驱动事件/消息传递系统

事件系统给予游戏程序员大量弹性，远远超越C/C++之类的语言所提供的静态函数类型调用机制所要做的事情。然而，我们可以做得更好。在之前的讨论中，收发事件的逻辑仍然是硬编码的，因此这些逻辑都是由工程师独家控制的。若我们令事件系统变成数据驱动的，就能把权力交到游戏设计师手中。

有很多方法可以把事件系统变成数据驱动的。从完全硬编码（非数据驱动）的事件系统为始，我们可以提供一些简单的数据驱动配置能力。例如，设计师可以容许对个别对象或某类的所有对象进行配置，决定它们能回应哪些事件。在世界编辑器中，我们可以选取一个对象，然后从弹出的所有事件列表中选择该对象能接收的事件。对于每个接收事件，设计师能在下拉组合列表框中选择一个或多个硬编码、预先定义好的对象反应方式。例如，对于

[33]译注：另一个常见的问题是，事件处理器在处理事件的期间，改变了一些状态，中间若触发另一些事件，那些事件处理器所取得的状态可能是不完整的。这类问题没有通解，但可以用防御式编程去发现问题。

“PlayerSpotted（玩家被发现）”事件，该人工智能角色可以设置一个以下的动作：逃跑、攻击，或不理会此事件。许多真实商用游戏引擎的事件系统本质上都采用这种实现方式。

引擎的另一个做法是向游戏设计师提供简单的脚本语言（14.8节会探讨此题目）。在这种情况下，设计师可以编写真正的代码来定义某类对象应该如何回应某种事件。在脚本模型下，设计师实质上等同于程序员（但脚本相比工程师用的语言，功能较有限、较容易学习、希望更少出错），因此一切皆可能实现。设计师可以任意定义新的事件类型、发送事件、接收事件、处理事件。

简单、可配置事件系统的问题在于，它限制了游戏设计师在没有程序员帮忙下所能做的事情。另一方面，全脚本的方案也有其问题：许多游戏设计师并没有经过专业软件工程师的培训，因此有些设计师对学习及使用脚本语言却步。相对于工程师，设计师也可能更易写出bug来，除非他们对写脚本或编程已实践过好一段日子。使用这种系统可能会在alpha版本遇见一些意料之外的问题。

因此，有些游戏引擎会寻找一些折中方案。它们可以使用高级的图形用户界面以提供大量弹性，而又不需要提供完整的、自由形式的脚本语言。其中一个方法是使用流程图风格的编程语言。这种系统的背后理念在于，提供一组受控的原子操作供用户选择，并让用户把操作按自己所想的方式自由地连接起来。例如，为了回应“PlayerSpotted”事件，设计师可以连接一个流程图，令角色退到最近的掩护点，然后播放一个动画，等5s再进行攻击。图形用户界面也可以提供错误检测及校验，以确保不会造成粗心大意的bug。

14.7.11.1 数据路径通信系统

把函数调用形式的事件系统转化为数据驱动，难点之一在于不同的事件类型会造成兼容问题。例如，假设在一个游戏中，玩家持有电磁脉冲枪（electromagnetic pulse gun，EMP gun），它发出的脉冲会令电灯和电子设备关掉、吓跑小动物，并造成冲击波（shock wave）令附近的植物摆动。上述的对象类型可能已经会各自回应一种事件，并做出反应表现所想的行为。小动物对“吓跑（scare）”事件的回应方式可能是迅速逃跑。电子设备对“关掉（turn off）”事件的回应方式可能是关掉自己。植物对“风（wind）”事件的回应方式可能是摆动。问题是EMP枪本身并不和这些对象的事件处理器兼容。因此，我们可能会实现一个新的事件类型，也许称为“EMP”事件，并在每种游戏对象类型中撰写特定的事件处理器，来回应这种事件。

此问题的一个解决方案是去掉事件类型，而仅考虑游戏对象**传送数据流**至其他对象。在这种系统中，每个对象含一个至多个**输入／输出端口**（port），输出端口可以用数据流连接至其他对象的输入端口，用来传送数据。如果我们有方法连接这些端口，例如在图形用户界

面中用连线方式连接端口，那么我们就能创建任意复杂的行为。继续刚才的例子，EMP枪会有一个输出端口，此端口可能命名为“射击”，它会输出一个布尔值。大部分时间里，此端口输出0值（即false），但当用枪射击时，该端口就会产生一个短暂（1帧）、值为1（true）的脉冲。世界中其他游戏对象也会有一些布尔输入端口，用来触发不同的反应。动物有“吓跑”端口，电子设备有“开启”端口，植物有“摆动”端口。当我们把EMP的“射击”端口连接至这些游戏对象的输入端口，就可以用枪击来触发所需的行为。（注意我们需要把“射击”输出先通过一个节点来**反转**信号，然后才接到电子设备的“开启”端口。这是由于我们希望射击时关掉这些电子设备。）图14.18展示了此例子的连线图。

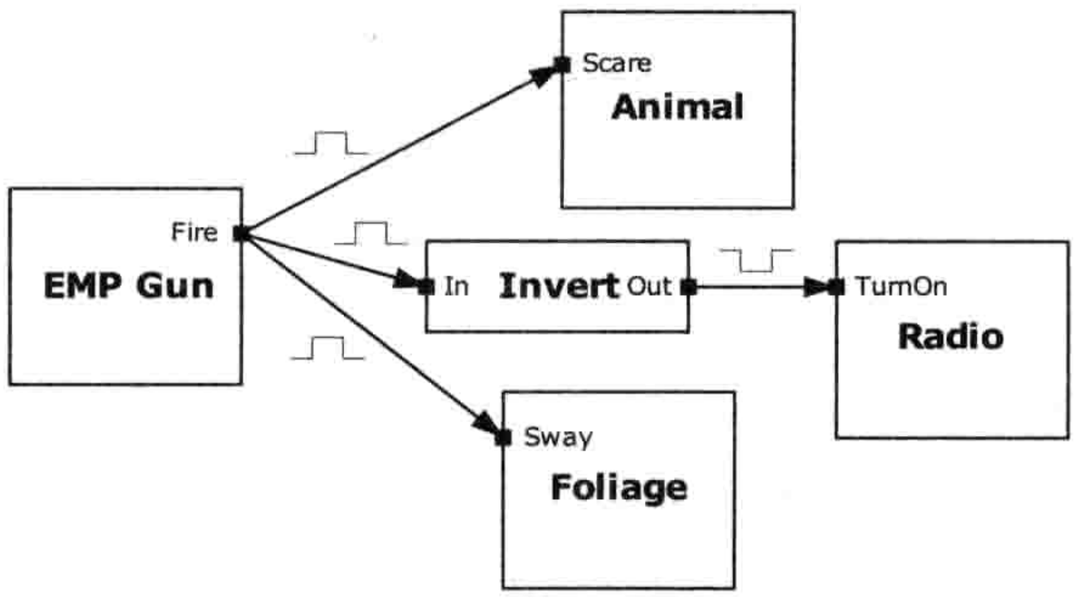


图 14.18: 当电磁脉冲枪开枪时，它于“射击”端口产生1的脉冲。此端口可连接至其他接受布尔值的输入端口，用来触发相关行为。

程序员可以制定每种游戏对象有什么端口。设计师就能使用图形用户界面，借着自由地连接这些端口来建立游戏所需的行为。程序员也会提供其他节点，例如，一个反转输入的节点、产生正弦波的节点、输出游戏时间（以秒为单位）的节点等。

数据路径也可以传输多种类型的数据。有些端口可能会产生或接收布尔数据，另一些可能会产生或接收单个浮点数。还有一些可能会是三维矢量、颜色、整数等。这种系统必须要确保连接的端口采用兼容的数据类型，或是必须提供一些机制为连接不同类型的端口时自动转型。例如，把单个浮点数输出连接至布尔输入时，可以把小于0.5的值转换为false，大于或等于0.5的值转换为true。这些就是基于图形用户接口事件系统的精髓，如图14.19所示的虚幻引擎Kismet系统。

14.7.11.2 视觉化编程的优缺点

图形用户接口相比文本式脚本语言，优势十分明显：容易使用，工具中的帮助和提示可以引导用户渐进学习，能有大量的错误检测。然而，流程图式的图形用户接口也有一些缺点，包括：开发、调试、维护这种系统的成本高，增加了复杂性，这些复杂可能会引致麻烦甚至影响进度的bug，而设计师的工作也会受工具所限。文本式编程语言有超越图形式编程

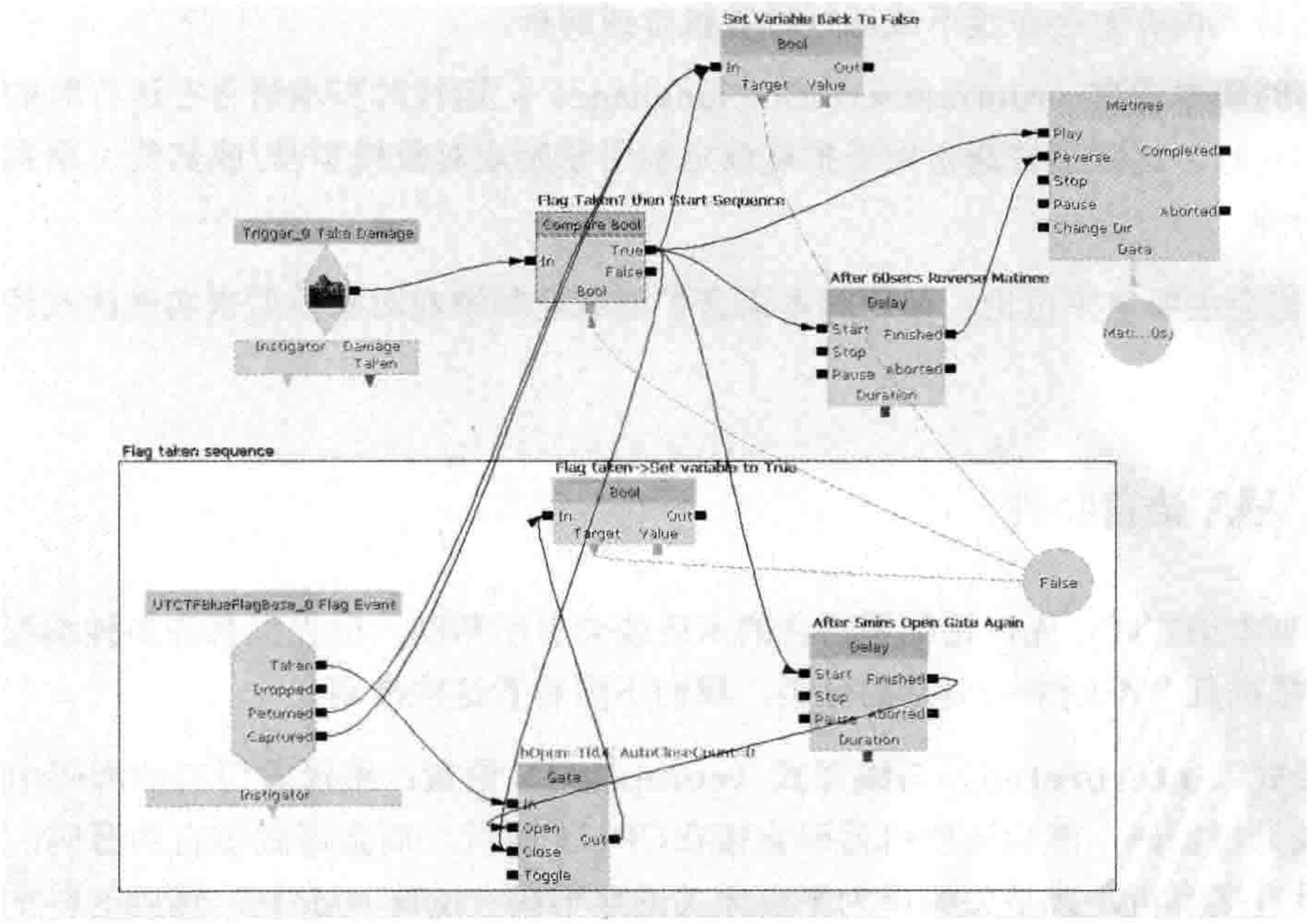


图 14.19: 虚幻引擎3的Kismet。

的优点，包括：相对简单（意味着更难出错），能简单地在代码中进行文字搜寻及取替，用户能自由选择喜爱的编辑器。

14.8 脚本

脚本语言（scripting language）可定义为一种编程语言，其主要目的在于让用户控制及定制应用软件程序的行为。例如，Visual Basic语言可用于定制Excel的行为，MEL和Python可用于定制Maya的行为。在游戏引擎的语境中，**脚本语言**是高级、相对容易使用的编程语言，供用户方便地使用引擎常用到的功能。因此，脚本语言可供程序员和非程序员开发新游戏，或是定制（或称为“mod”）一个现存的游戏。

14.8.1 对比运行时与数据定义

我们应谨慎地画一个重要的分界线。游戏脚本语言通常有两种。

- **数据定义语言**（data-definition language）：数据定义语言的主要功能在于让用户创建及填充数据结构，这些数据将会供引擎读取。这些语言通常是声明式的（见后文），当

数据载入内存时会在线下或运行时被执行或解析。

- **运行时脚本语言**（runtime scripting language）：运行时脚本语言在运行时的引擎上下文中执行。这些语言通常用于扩展或定制引擎游戏对象模型及/或其他引擎系统的硬编码功能。

在本节我们会主要集中讨论，使用脚本语言扩展或定制游戏对象模型来实现游戏性功能。

14.8.2 程序语言特性

讨论脚本语言时，先讨论编程语言的术语或会有所帮助。世界上有许多种编程语言，但它们可以根据几个准则来做概括的分类。我们下面看看这些准则。

- **直译式（interpreted）与编译式（compiled）语言：编译式**语言的源码由编译器翻译成为机器码，然后这些机器码直接在CPU上执行。而**直译式**语言的源码，可以在运行时直接解析，或是先编译为平台无关的**字节码**（byte code），然后这些字节码在**虚拟机**（virtual machine）中执行。虚拟机如同一个假想CPU的模拟，而字节码就如同此CPU可执行的机器码。虚拟机的好处在于它比较容易移植至几乎所有硬件平台，并可以嵌入至一个宿主应用程序（如游戏引擎）。此弹性带来的最大代价是运行速度，虚拟机执行字节码的速度通常比原生CPU执行机器码要慢得多。
- **命令式（imperative）语言：**在命令式语言中，程序是由指令序列所组成的，每个指令执行一个操作或是读 / 写内存中的数据。C/C++是命令式语言。
- **声明式（declarative）语言：**声明式语言描述要做**什么**但不指明**如何**取得结果。如何执行取决于语言实现。Prolog是声明式语言的例子。置标语言（markup language），如HTML和TeX，也可归类为声明式语言。
- **函数式（functional）语言：**函数式语言在技术上可算是声明式语言的子集，其目的是完全避免使用状态。在函数式语言中，程序由一组函数所定义。每个函数在不产生副作用的情况下输出结果（即除了输出数据外不会对系统产生可观察的改动）。程序的建构方法是由一个函数传递输入参数至另一函数，直至取得所需的结果。这些语言适合实现一些数据处理管道。Ocaml、Haskell、F#是函数式语言的例子。
- **过程式（procedural）与面向对象（object-oriented）语言：**在过程式语言中，程序建构的主要原子是**程序**（procedure），或是**函数**（function）。这些程序和函数执行操作、计算结果及/或改变内存中数据结构的状态。相比之下，面向对象语言的主要建构单位是**类**（class），它是一种数据结构，与一组程序/函数紧密耦合，这些程序/函数“懂得”如何管理及操作该数据结构中的数据。

- **反射式（reflective）语言：**在反射式语言中，数据类型、数据成员布局、函数、类层次结构关系等信息可以在运行时于系统取得。在非反射式语言中，大部分这些元信息只能在编译时获取，只有非常小的部分能供运行时取用。C#是反射式语言的例子，而C/C++是非反射式语言的例子。

14.8.2.1 游戏脚本语言的典型特性

游戏脚本语言相对于原生编程语言的主要特性包括：

- **直译式：**多数游戏脚本语言都是由虚拟机直译的，而非编译的。此选择是基于弹性、可移植性、快速迭代（见下文）的考虑。当把代码表示为平台无关的字节码，引擎就可以把这些代码当作数据处理。这些代码可以如同其他资产般载入内存，而不需要操作系统的帮忙（例如PC平台上的DLL或PS3的IRX，则需涉及操作系统）。因为这些代码由虚拟机执行而非直接由CPU执行，游戏引擎可以有很大的弹性以决定如何及何时执行这些脚本代码。
- **轻量：**多数游戏脚本语言都是为嵌入式系统设计的。因此，它们的虚拟机比较简单，而且内存消耗也比较少。
- **支持快速迭代：**每当更改原生源程序码，我们必须重新编译及链接程序；游戏也需要先关掉再重启，才能看到改动后的效果。另一方面，脚本代码改动后，通常可以非常快速地看到改动的效果。有些游戏引擎容许即时重新载入脚本，而无须关掉游戏。有一些引擎可能需要关掉再重启。但无论是哪一种方式，修改脚本至看到改动效果的时间，通常比更改原生语言源程序要短得多。
- **方便易用：**脚本语言通常会根据具体游戏的需求来定制。我们可以提供一些功能令常见的任务变得简单、直觉及较难出错。例如，游戏脚本语言可以提供一些功能，如以名字搜寻对象、发送及处理事件、暂停或控制时间、等待直至指定的时间、实现有限状态机、对游戏编辑器暴露可调参数供游戏设计师使用，或是为多人游戏处理网络复制。

14.8.3 一些常见的游戏脚本语言

当实现运行时游戏脚本系统时，我们有一个最基本的抉择：应选用第三方的商业或开源语言并定制成我们所需，还是自行从无到有设计及实现一门定制语言呢？

从零开始创造一门语言，其麻烦程度和在整个项目周期的维护成本通常造成得不尝失。聘请游戏设计师和程序员时，要求他们已熟识我们定制的内部语言，也是很困难或根本不可

能的事情，因此通常我们还需要一些培训成本。然而，显然创造一门语言是最富弹性和定制性的方法，而那些弹性可能是值得投资的。

对于大多数工作室而言，更合适的方法是选择一个知名且成熟的脚本语言，并针对具体的游戏引擎扩展其功能。坊间有许多第三方的脚本语言可供选择，许多都是成熟和健壮的，并曾成功应用在游戏行业内外许多项目之中。

以下数小节中，我们会介绍一些定制游戏脚本语言，以及一些常用于游戏引擎但本身和游戏无关的脚本语言。

14.8.3.1 QuakeC

id Software公司的John Carmack为《雷神之锤（Quake）》实现了一门自定义脚本语言，称为**QuakeC**（QC）。此语言本质上是C编程语言的简化版本，并直接整合至雷神之锤引擎中。此语言不支持指针或定义任意的struct，但它可以方便地操控**实体**（entity——雷神之锤引擎中游戏对象的名称），并可以用于收发游戏事件。QuakeC是直译式、命令式、过程式语言。

把QuakeC交给游戏玩家之后，导致我们现在称为**mod社区**的诞生。通过脚本语言及其他形式的数据驱动定制功能，玩家可以把许多商用游戏转化为各式各样的游戏体验，由原先题材轻微修改至完全新的游戏皆有可能。

14.8.3.2 UnrealScript

或许最知名的完全定制脚本语言就是虚幻引擎的**UnrealScript**。此语言基于C++语法风格，并支持许多C/C++程序员习惯的概念，包括类、局部变量、循环、以数组及struct组织数据、字符串、散列字符串（虚幻中称为FName）、对象参考（但不是自由形式的指针）等。此外，UnrealScript还提供一些极为强大的游戏专用功能，以下我们将就这些功能做简单探讨。UnrealScript是直译式、命令式、面向对象语言。

扩展类层次结构的能力

这点也许是UnrealScript最著名的地方。Unreal对象模型本质上是单一的类层次结构，加上组件提供各引擎系统的接口。层次结构的根是一些**原生类**（native class），因为它们是用原生C++语言实现的。但UnrealScript真正强大的地方在于，它能够纯粹用脚本实现新的派生类。

听上去这好像不是什么了不起的事情，除非读者亲身试过实现同样的东西！实际上，UnrealScript重新定义并扩展了C++的**原生对象模型**，这是令人吃惊的事情。对于原生的Unreal类，UnrealScript的源文件（其一般扩展名为.uc）取代了C++的头文件（.h文件）定义类，实际上UnrealScript编译器会从.uc文件**生成**C++的.h文件，而程序员只需在.cpp源文件中实现这些类。这么做，UnrealScript编译器就能为每个Unreal类加入一些额外的功能，而这些功能令新的脚本类可以派生自原生类或脚本类。

Latent函数

Latent函数可以跨越多个游戏性的帧去执行。Latent函数可以执行一些指令，然后“睡眠”，等待一个事件或一段时间。等相应的事件发生或到达指定时间，引擎就会令函数“苏醒”，在之前停下来的地方继续执行。对于依赖时间流逝的游戏，此功能对管理这种游戏内的行为非常有用。

方便与UnrealEd联系

基于UnrealScript的类数据成员都可以加入一个简单的注释，用来表示该数据成员如何在Unreal的世界编辑器——UnrealEd——中显示及编辑。这无须GUI编程。此功能使数据驱动游戏设计非常容易（只要UnrealEd内置的数据成员编辑GUI能满足所需）。

多人游戏的网络复制

UnrealScript中每个数据元素都可以标记为需要复制（replication）的。在虚幻网络游戏中，每个游戏对象在某个机器上有完整的表示，但其他机器只有一个轻量的对象版本，此版本称为**远程代理**（remote proxy）。当一个数据成员标示为需要复制，引擎就会自动地把对象主版本的这些数据复制至所有远程代理去。此功能令程序员或设计师可以容易控制网络上应该传播哪些数据，也间接地控制了游戏所需的网络流量。

14.8.3.3 Lua

Lua是著名且流行的脚本语言，它能容易地整合至应用软件，包括游戏引擎。Lua官网[34]称Lua是“游戏界脚本语言之首”。

根据Lua官网的介绍，Lua的关键优点包括：

[34] http://www.lua.org/about.html

- **健壮及成熟**：Lua已应用于许多商用产品，包括Adobe的Photoshop Lightroom，以及许多的游戏，例如《魔兽世界》。
- **优良文档**：Lua的参考手册[21]是完整、可理解的文档，提供了线上和书本格式。坊间也有许多关于Lua的书籍，例如[22]和[43]。
- **卓越的运行时性能**：Lua执行其字节码的速度要比许多其他脚本语言快及高效。
- **可移植性**：无须修改，Lua就可以运行不同版本的Windows和UNIX、手提设备、嵌入式微处理器等。Lua本身以可移植的方法编写，令它能很容易地适配新的硬件平台。
- **为嵌入设备而设计**：Lua占用非常少内存（其直译器及所有程序库大约只占350KB）。
- **简单、强大、可扩展**：Lua的语言核心是非常细小及简单的，但Lua设计了一些元机制（meta-mechanism）无限扩展它的核心功能。例如，Lua本身并不是面向对象语言，但我们可以通过元机制加入OOP的支持。
- **免费**：Lua是开源的，而且以非常自由的MIT许可证方式发布。

Lua是动态类型语言，意味着变量没有类型，只有值才有类型（每个值带有类型信息）。Lua的基本数据结构是**表**（table），也即是关联数组（associative array）。表本质上是一个键值对的列表，优化了以值索引的能力。

Lua提供一个C语言的方便接口，令Lua虚拟机可以调用及操作用C编写的函数，如同Lua本身提供的一些函数。

Lua把代码段（称为**chunk**）当作第一类对象，Lua程序也可以处理这些代码段。代码可以用源码方式执行，也可以预编译为字节码格式。所以虚拟机可以执行一个含Lua代码的字符串，如同该代码被编译进原来的程序中。Lua支持强大高级的编程功能，包括**协程**（coroutine）。这是合作式多任务（cooperative multitasking）的一种简单形式，每个线程必须明确地交出CPU供其他线程执行（而不是像抢占式多任务系统以时间片进行调度）。

Lua也有一些隐患。例如，其富弹性的函数绑定机制令我们可以（且很容易）重新定义一些重要的全局函数（如`sin()`）去执行完全不同的任务（而不是我们原来所想要的）。但总的来说，Lua证明了自己是游戏脚本语言的卓越之选。

14.8.3.4 Python

Python是过程式、面向对象、动态类型脚本语言，为容易使用、与其他程序语言整合、高度弹性而设计。和Lua相似，Python也常用作游戏脚本语言。根据Python官网[35]，Python的最优功能包括：

[35] `http://www.python.org`

- **清晰可读的语法**：Python代码容易阅读，部分原因是其语法强制使用一种专门的缩进方式。（Python根据解析空白来决定每行代码的作用域。）
- **反射性语言**：Python包含强大的运行时自省能力。Python中的类是第一类对象，意味着我们可以在运行时对类进行操作及查询，如同其他对象一样。
- **面向对象**：Python优于Lua的一点是Python把OOP功能建于其语言核心之内。这样会更容易把Python整合至游戏的对象模型。
- **模块化**：Python支持层次结构式程序包，鼓励清晰的系统设计及良好的封装。
- **基于异常的错误处理**：相比非基于异常的语言，Python的异常简化了错误处理代码，也令那些代码变得更优雅、更局部化。
- **包罗万象的标准库及第三方模块**：Python程序库几乎覆盖任何可想象到的任务。（真的！）
- **可嵌入**：Python比较容易嵌入应用软件中，如游戏引擎。
- **详尽的文档**：坊间有许多Python的文档及教材，线上和书本形式的都有。Python官网是一个好的出发点。

Python的语法在多方面会令人想起C语言（例如，它使用=运算符作为赋值，以及用==做相等测试）。然而，**代码缩进**（indentation）是在Python中定义**作用域**的唯一方法（而不是采用C语言的花括号）。Python的主要数据结构是**list**和**dictionary**，前者是线性可索引的序列，后者则是键值对的表。这两种数据结构都能储存原子值或是其他list/dictionary，所以我们可以很轻易地建立任意复杂的数据结构。除此以外，**类**（数据元素和函数的统一集合）也建立在该语言之内。

Python支持**鸭子类型**（duck typing），它是指一种动态类型风格，当中函数接口决定了对象的类型（而不是用静态的继承关系来决定的）。换言之，任何类只要提供某个接口（即一组指定签名的函数），就能替换其他支持相同接口的类。这是一个强大的范式，其效果是Python支持多态（polymorphism）而无须使用继承。鸭子类型在某些方面和C++的模板元编程相似，虽然鸭子类型大概是更富弹性的，因为调用者和被调用者的绑定是动态在运行时形成的。鸭子类型之名来自著名的谚语（出自James Whitcomb Riley）：“若它像鸭子般走路，也叫得像鸭子，那么我会称它为鸭子。（If it walks like a duck and quacks like a duck，I would call it a duck.）”。关于鸭子类型的更多信息，可参阅维基百科[36]。

总而言之，Python易用易学，容易嵌入游戏引擎，能良好地整合游戏对象模型，可以是一个游戏脚本语言的卓越强大之选。

[36] http://en.wikipedia.org/wiki/Duck_typing

14.8.3.5 Pawn/Small/Small-C

Pawn是一个轻量、动态类型、类C的脚本语言，由Marc Peter所创。此语言之前被称为**Small**，而Small本身演化自一个更早的C语言子集语言**Small-C**，Small-C是由Ron Cain和James Hendrix编写的。Pawn是一个直译式语言，它的源码先编译成字节码（称为P代码），然后再由运行时的虚拟机直译。

Pawn的设计尽量占用少量的内存，并且能快速地执行字节码。和C不一样，Pawn的变量是动态类型的。Pawn也支持有限状态机，包括状态内的局部变量。此独特功能令它非常适合很多游戏应用。网上有优良的Pawn文档[37]。Pawn是开源的，而且可以在Zlib/libpng许可证下免费使用[38]。

Pawn的语法近似C，令C/C++程序员容易学习，并且能容易地整合至以C编写的游戏引擎。它的有限状态机支持对游戏编程十分有用。它曾成功地应用在许多游戏项目中，包括Midway的《疯狂飞行员（Freaky Flyers）》。Pawn证明了自己是切实可行的游戏脚本语言。

14.8.4 脚本所需的架构

脚本代码在游戏引擎中可以扮演许多不同角色。有许多架构可供考虑，从简单的脚本代码片段去代表一个对象或引擎系统执行一些简单功能，到高层的脚本去管理游戏的操作，也是可能的。以下是几个可能的架构。

- **回调脚本**：在此架构下，引擎的主要功能大部分都是以原生编程语言硬编码的，只有某些关键的小功能设计成可定制的。这些部分通常实现为**钩子函数**（hook function），或称为**回调**（callback），都是指用户提供一个函数供引擎调用，借以做出定制。钩子函数可以用原生语言编写，但当然它们也能用脚本语言编写。例如，当在游戏循环里更新游戏对象，引擎可能会调用一个可选的回调函数，此函数可以用脚本编写。这样做能令用户有机会定制游戏对象更新的方式。
- **事件处理器脚本**：事件处理器其实只是一种特殊的钩子函数，其作用是令游戏对象回应游戏世界发生的相关事件（例如对爆炸做出反应），或是回应引擎本身的事件（如内存不足）。许多游戏引擎容许以脚本或原生语言编写事件处理器。
- **以脚本扩展游戏对象类型或定义新类型**：有些脚本语言可以用脚本扩展原来由原生语言实现的游戏对象类型。事实上，回调和事件处理器也是这种方式的小型例子，但此

[37] http://www.compuphase.com/pawn/pawn.htm

[38] http://www.opensource.org/licenses/zlib-license.php

概念可以扩展至完全由脚本定义新的游戏对象类型。我们可以使用**继承**（即自一个由原生语言编写的类派生一个由脚本语言编写的类），又或是采用**组合/聚合**方式（例如把脚本类的实例绑定至原生的游戏对象）。

- **组件或属性脚本：**在基于组件或基于属性的游戏对象模型中，只有容许用脚本或部分脚本创建新组件或属性对象，这样才有意义。Gas Powered Games的《末日危城（Dungeon Siege）》采用的就是这种方式[39]。其游戏对象模型是基于属性的，他们容许属性用C++或Gas Powered Games的定制脚本语言**Skrit**[40]来编写。在项目结束时他们大概有148个脚本属性类型和21个原生C++属性类型。
- **脚本驱动的引擎系统：**脚本可能用于驱动整个引擎系统。例如，我们可以想象游戏对象模型完全由脚本编写，仅当需要一些底层的引擎组件时才调用原生的引擎代码。
- **脚本驱动的游戏：**有些游戏引擎完全颠倒原生语言和脚本语言的关系。在这些引擎中，脚本代码是游戏运行的主体，原生引擎代码仅作为程序库，用来调用一些高速的引擎部分。Panda3D引擎[41]就是这种架构的例子。Panda3D的游戏完全使用Python语言编写，而其原生引擎（以C++实现）仅作为被脚本调用的库。（Panda3D的游戏也可以完全用C++编写。）

14.8.5 运行时游戏脚本语言的功能

许多游戏脚本语言的主要目的是用于实现游戏性功能，而达成此目的的方法通常是修改及定制游戏的对象模型。本节会探讨这些脚本系统的一些最常见需求及功能。

14.8.5.1 对原生编程语言的接口

为了令脚本语言有所作为，不能让它闭门造车。游戏引擎必须要能执行脚本代码，同样重要的是，脚本代码也需要发起引擎中的操作。

运行时脚本语言的虚拟机通常是嵌入游戏引擎中的。引擎启动虚拟机，需要时执行脚本代码，并管理脚本的执行情况。执行的单位根据具体语言和游戏实现有所不同。

- 在函数式脚本语言中，**函数**通常是执行的主要单位。为了调用一个脚本函数，我们需要按函数名称找到对应的字节码，然后生成一个虚拟机去执行该函数（或命令一个现存的虚拟机去执行）。

---

[39] http://www.drizzle.com/~scottb/gdc/game-objects.ppt

[40] http://ds.heavengames.com/library/dstk/skrit/skrit

[41] http://www.panda3d.org

- 在面向对象脚本语言中，**类**通常是执行的主要单位。这种系统可以创建和销毁对象，以及对个别实例调用其方法（成员函数）。

在脚本语言和原生代码之间，容许双向通信是甚有裨益的。因此，多数脚本语言也容许在脚本中调用原生代码。其调用方法细节是语言和实现相关的，但是最基本的方法是容许一些脚本函数用原生语言去实现，而不是用脚本语言。要调用一个引擎函数，脚本代码只需简单地进行一个正常的函数调用。然后虚拟机会检测到该函数有原生实现，它查找对应原生函数的地址（可能用名字或其他唯一函数标识符），并调用它。例如，在Python的类或模块中，部分或全部成员函数可以用C函数实现。Python维护了一个数据结构，称为**方法表**（method table），用来把每个Python函数的名字（以字符串表示）映射至对应的C函数地址。

个案研究：顽皮狗的DC语言

以下我们用顽皮狗的运行时脚本语言DC作例，看看它如何整合至引擎中。

DC是Scheme语言的变种（Scheme是Lisp的变种）。DC中的执行代码段称为**脚本lambda**，大约等于Lisp语言家族里的函数或代码块。DC程序员编写脚本lambda，并为它们用全局唯一名字命名。DC编译器把这些脚本lambda编译成字节码，然后这些字节码块就能当游戏运行时载入内存，并可用一个简单的C++函数接口以名字查找所需的脚本lambda。

当引擎取得脚本lambda的指针，就能调用一个引擎的函数并把该指针传入，执行脚本lambda。该函数的实现意想不到的简单，只是做一个循环，逐一读取字节码指令，并执行这些指令。当所有指令都执行完毕，函数就能返回。

在DC的虚拟机里有一组寄存器（register），用来储存脚本可能需要的任何数据。我们使用了**variant**数据类型，即所有数据类型的union（可参考14.7.4节的相关讨论）。有些指令会令数据载入寄存器，另一些指令可以查找寄存器并读取数据。指令中包含执行语言中的所有数学运算，还有执行条件检查的指令，例如为DC中`(if…)`、`(when…)`及`(cond…)`等实现的指令。

虚拟机里也支持**函数调用堆栈**（function call stack）。DC中的脚本lambda是脚本程序员使用DC的`(defun…)`语法定义的。脚本lambda可以调用其他脚本lambda。如同任何过程式编程语言，我们需要有一个堆栈去记录寄存器的状态及函数的返回地址。在DC的虚拟机里，调用堆栈实质上是寄存器组（register bank）的堆栈，即每个新函数调用拥有一个私有的寄存器组。这种做法令我们无须先备份寄存器，调用函数，然后当函数结束时再恢复寄存器状态。DC虚拟机的做法是，遇到调用另一脚本lambda的字节码时，首先用名字查找该脚本lambda，然后把一个新的寄存器组压入栈，之后就可以继续执行该脚本lambda的首

个指令。而当虚拟机遇到返回指令，就只需简单把寄存器组出栈，当中已包含返回“地址”（返回“地址”其实只是原来调用方在调用指令之后的字节码指令索引）。

以下的伪代码能让读者领略一下DC虚拟机的核心指令处理方式：

```
void DcExecuteScript(DCByteCode* pCode)
{
    DCStackFrame* pCurStackFrame = DcPushStackFrame(pCode);

    // 继续直至再没有堆栈帧(即顶层的脚本lambda“函数”返回)
    while (pCurStackFrame != NULL)
    {
        // 取得下一指令。我们永不会取不到指令，因为返回指令总是最后一个，
        // 并且返回指令会弹出堆栈帧
        DCInstruction& instr = pCurStackFrame->GetNextInstruction();

        // 执行指令
        switch (instr.GetOperation())
        {
        case DC_LOAD_REGISTER_IMMEDIATE:
            {
                // 载入此指令的立即值
                Variant& data = instr.GetImmediateValue();

                // 并判断要存入哪个寄存器
                U32 iReg = instr.GetDestRegisterIndex();

                // 从堆栈帧取得寄存器
                Variant& reg = pCurStackFrame->GetRegister(iReg);

                // 储存立即值至该寄存器
                reg = data;
            }
            break;

        // 其他载入及储存寄存器指令……

        case DC_ADD_REGISTERS:
            {
                // 判断要为哪两个寄存器相加
                // 结果会存于寄存器A
                U32 iRegA = instr.GetDestRegisterIndex();
                U32 iRegB = instr.GetSrcRegisterIndex();

                // 从堆栈取得两个寄存器variant
```

```
            Variant& dataA = pCurStackFrame->GetRegister(iRegA);
            Variant& dataB = pCurStackFrame->GetRegister(iRegB);

            // 把两个寄存器相加，存入寄存器A
            dataA = dataA + dataB;
        }
        break;

    // 其他数学指令……

    case DC_CALL_SCRIPT_LAMBDA:
        {
            // 判断脚本lambda的名字储存在哪个寄存器
            // （假设之前已用载入指令把名字载入该寄存器）
            U32 iReg = instr.GetSrcRegisterIndex();

            // 取得该寄存器，它内含要调用的lambda名字
            Variant& lambda = pCurStackFrame->GetRegister(iReg);

            // 按名字查找lambda的字节码
            DCByteCode* pCalledCode =
                DcLookUpByteCode(lambda.AsStringId());

            // 现在通过压入新的堆栈帧来“调用”lambda
            if (pCalledCode)
            {
                pCurStackFrame = DcPushStackFrame(pCalledCode);
            }
        }
        break;

        case DC_RETURN:
            {
                // 简单地弹出堆栈帧。若我们在顶层lambda，
                // 此函数会返回NULL，然后循环会被终止
                pCurStackFrame = DcPopStackFrame();
            }
            break;

        // 其他指令……
        // ……
        } // switch结束
    } // while结束
}
```

在以上的例子中，假设了全局函数DcPushStackFrame()和DcPopStackFrame()会以合适的方法管理寄存器组的堆栈，而全局函数DcLookUpByteCode()则可以用名字查找所需的脚本lambda。这里不会展示所有这些函数，因为此例子的目的仅用于示范一个脚本虚拟机的内部循环如何运作，而不是提供一个完整可跑的实现。

DC脚本lambda也可以调用原生函数，这些原生函数是用C++编写的全局函数，并作为钩子使用引擎的功能。当虚拟机遇到要调用原生函数的指令时，它会通过一个由引擎程序员硬编码的全局表，按名字查找出C++函数的地址。当虚拟机找到合适的C++函数时，就会提取当前堆栈中的寄存器参数，然后调用该函数。这意味着C++函数的参数总是Variant类型。当C++函数传回一个值，那个值也必须是一个Variant，然后该值就会储存至当前堆栈帧的一个寄存器中，以供后续的指令使用。

那个全局函数表可能是这个样子的：

```
typedef Variant DcNativeFunction(U32 argCount, Variant* aArgs);

struct DcNativeFunctionEntry
{
    StringId            m_name;
    DcNativeFunction*   m_pFunc;
};
DcNativeFunctionEntry g_aNativeFunctionLookupTable[] = {
    { SID("get-object-pos"), DcGetObjectPos },
    { SID("animate-obejct"), DcAnimateObject },
    // 等等
    // ……
}
```

以下展示了一个原生DC函数实现的大概样子。注意当中Variant参数是以数组的形式传递的。函数必须要核实参数的数量合乎预期，也需要核实每个参数的类型合乎预期，并要准备处理DC脚本程序员错误调用函数的情况。

```
Variant DcGetObjectPos(U32 argCount, Variant* aArgs)
{
    // 设定默认的返回值
    Variant result;
    result.SetAsVector(Vector(0.0f, 0.0f, 0.0f));

    if (argCount != 1)
    {
        DcErrorMessage("get-object-pos: Invalid arg count.\n");
        return result;
```

```
    }

    if (aArgs[0].GetType() != Variant::TYPE_STRING_ID)
    {
        DcErrorMessage("get-object-pos: Expected string id.\n");
        return result;
    }

    StringId objectName = aArgs[0].AsStringId();

    GameObject* pObject = GameObject::LookUpByName(objectName);

    if (pObject == NULL)
    {
        DcErrorMessage("get-object-pos: Object '%s' not found.\n",
            objectName.ToString());
        return result;
    }

    result.SetAsVector(pObject->GetPosition());
    return result;
}
```

14.8.5.2 游戏对象句柄

脚本函数通常需要与游戏对象互动，而游戏对象本身可能是部分或全部由引擎原生语言所实现的。原生语言的对象引用机制（如C++的指针和参考）未必能用于脚本语言。（例如脚本语言可能完全不支持指针。）因此，我们需要一个可靠的方法让脚本引用游戏对象。

有很多方法可达到此目的。其中一个方法是在脚本中以不透明的数值型**句柄**（handle）来引用对象。脚本对象可以用多种方法取得对象句柄。句柄可能是由引擎传递过来的，也可以通过一些查询取得，例如，查询玩家某半径范围内的所有对象句柄，或是以特定的对象名字查找其句柄。那么脚本就能把句柄作为参数来调用原生函数，借以对该对象进行一些操作。在原生语言方面，句柄会被转换为指向原生对象的指针，然后适当地处理该对象。

数值型句柄的优点是简单，能容易地应用于支持整数数据的脚本语言。然而，数值型句柄用起来可能有点不够直觉。另一选择是采用对象的名字，以字符串取代句柄。相对于数值型句柄，这种做法有一些有趣的优点。首先，字符串是人类可读而且能很直觉地使用。这些字符串也能直接对应至游戏世界编辑器中对象的名字。此外，我们可以选择保留一些特殊的对象名字，并给予这些名字一些“魔法”意义。例如，在顽皮狗的脚本语言中，保留名字“self”总是代表执行中脚本现时绑定至的对象。那么游戏设计师就能写一个脚本，并把它绑

定至游戏中的一个对象，然后简单地写（animate "self" 动画名称）来播放某动画。

当然，使用字符串作为对象句柄也会有些缺点。字符串比整数标识符占用更多的内存空间。而且字符串的长度是可变的，所以需要动态内存分配来复制它们。比较两个字符串也是较慢的操作。脚本程序员也很容易拼错对象的名字而产生bug。此外，若有人在游戏世界编辑器中修改对象的名字，而又忘记更新脚本中的对象名字，就会令脚本不能正常运作。

字符串散列标识符可以克服以上大部分的问题，它把字符串（无论任何长度）转换为整数。理论上，字符串散列标识符能取得两者之长，既能像字符串供用户阅读，又能有整数的运行时性能特征。然而，要令这种方法可行，脚本语言需要用某种方式支持字符串散列标识符。理想地，我们希望脚本编译器能把字符串转换为字符串散列标识符。那么运行时代码就完全不需要处理字符串，而只需使用字符串散列标识符（除了在调试时，我们希望能看到字符串散列标识符对应的字符串）。然而，不是所有脚本语言都能这么做。另一种方法是在脚本中使用字符串，当运行时需要调用原生函数时，就把字符串转换为字符串散列标识符。

14.8.5.3 接收及处理事件

在多数的游戏引擎中，事件是无处不在的通信机制。只要容许用脚本编写事件处理函数，就能对定制游戏中硬编码行为开启一个强大之门。

事件通常发送到个别对象，并在该对象的上下文中进行处理。因此脚本化的事件处理器需要用某方式关联至一个对象。有些引擎利用游戏对象类型系统实现此功能，就是令脚本化的事件处理器可以用类为单位的形式进行绑定。那么不同类型的对象可以用不同方式回应相同的事件，但确保了同类型的所有实例能一致地、统一地回应事件。事件处理函数本身可以是简单的脚本函数，若脚本语言是面向对象的，那么事件处理器也可以是一个子类。无论是哪一种方式，我们通常会把事件接收方的对象句柄传入事件处理器，如同调用C++成员函数时会传入this指针一样。

在另一些引擎中，脚本处理器关联至个别对象，而不是关联至对象类型。那么相同类型的不同实例就能以不同方式回应相同的事件。

当然，我们还有许多其他选择。例如，在顽皮狗的神秘海域引擎中，脚本本身就是对象。它们能关联至个别游戏对象，也可以绑定至区域（用来触发事件的凸空间），或是作为游戏世界中的独立对象。每个脚本可以有多个状态（即是说神秘海域引擎中的脚本是有限状态机）。而每个状态下可以有一个至多个事件处理代码块。当游戏对象接收到一个事件，它可选择用原生C++的方式处理该事件，也可选择检查该对象有没有绑定脚本对象，若有，则可以把事件发送到该脚本的当前状态。若该状态含有对应该事件的处理器，就会调用它。否则，那个脚本会忽略该事件。

14.8.5.4 发送事件

容许脚本处理引擎产生的游戏事件，无疑是一项强大的功能。另一更强大的功能是，从脚本代码产生事件，并把事件发送至引擎或其他脚本。

理想地，我们希望脚本不单能发送预先定义的事件类型，还可以用脚本定义全新的事件类型。如果事件仅仅是字符串的话，那就简单不过了。要定义一个新的事件类型，脚本程序员只需在代码中加入新事件的名字。这可以成为一种非常富弹性的脚本间通信方法。脚本A可以定义一个新的事件类型并发送至脚本B。若脚本B定义了此事件类型的事件处理器，我们就能实现脚本A向脚本B“传话”了。在一些游戏引擎中，事件或消息传递是用脚本实现对象间通信的唯一方式。此方式可以成为优雅、强大及弹性的方案。

14.8.5.5 面向对象脚本语言

有些脚本语言本质上是面向对象的。另一些语言不直接支持对象，但提供一些机制可用于实现类和对象。许多引擎中，游戏性是通过某形式的面向对象游戏对象模型来实现的。因此，容许脚本中使用某些形式的面向对象编程是很合理的。

脚本中定义类

类其实只是一些数据及其相关的函数。因此，任何脚本只要可以定义新的数据结构，并提供一些方法储存及处理函数，就可以用来实现类。例如，在Lua中，可以用表来建立类，把数据成员和成员函数储存于表中。

脚本中的继承

面向对象语言并非必须要支持继承。然而，若支持这个功能，可能发挥极大作用，如同原生编程语言如C++中的继承。

在游戏脚本语言的上下文中，存在两种继承方式：从脚本类派生另一脚本类、原生类派生脚本类。若脚本语言是面向对象的，那么前一种通常都会直接支持。然而，即使脚本语言支持继承，后一种也难以实现。问题在于，要整合两种语言及两种底层对象模型。我们在此不深入探究如何实现这种继承，因为这种实现是与需要整合的两种具体语言相关的。能无缝地从原生类派生脚本类的脚本语言，笔者只曾见过UnrealScript。

脚本中的合成/聚合

我们不需要依赖继承扩展类层次结构，也可以用合成（composition）或聚合（aggregation）做到相同的效果。在脚本中，我们所要做的，就是定义一些类，并把这些类关联至原生编程语言所定义的类。例如，游戏对象可以含有指针，指向一个可选的组件，而该组件完全由脚本编写。若那些组件存在，我们就把某些关键功能委派给脚本组件。脚本组件也可以有Update()函数，每当游戏对象更新时就会被调用。此外，脚本组件可以登记一些函数/方法为事件处理器。当事件发送至游戏对象，游戏对象就会调用脚本组件中的合适事件处理器，让脚本程序员有机会修改或扩展原生实现游戏对象的行为。

14.8.5.6 有限状态机脚本

游戏编程中许多问题可以用有限状态机（finite state machine，FSM）解决。因此，有些引擎把FSM的概念建立在核心游戏对象模型中。在这些引擎中，每个游戏对象可以有一个至多个状态，并且每个状态（而不是每个游戏对象本身）会含有更新函数、事件处理函数等。简单的游戏对象可能只会定义一个状态，但复杂的游戏对象可自由地定义多个状态，每个状态各有不同的更新及事件处理行为。

若引擎支持状态驱动游戏对象模型，那么在脚本中提供FSM的支持就很有意义了。当然，即使核心游戏对象模型不直接支持FSM，我们仍然可以在脚本中使用FSM来提供状态驱动行为。任何编程语言都可以使用类表示状态来实现FSM，但一些脚本语言可以提供一些专门的工具来实现FSM。面向对象脚本语言可以提供自制的语法，使一个类能包含多个状态，并且可以提供一些工具帮助脚本程序员把状态对象聚合为一个中央枢纽对象，然后直观地把更新及事件处理函数委托给它。然而，即使脚本语言没有提供这些功能，仍然总是能根据一些惯例在每个脚本中实现FSM。

14.8.5.7 多线程脚本

并行执行多个脚本的能力一般是很有用的，特别对现今高度并行的硬件架构而言尤其重要。若多个脚本能同时间执行，我们实质上要提供在脚本代码中**并行执行的线程**，如同大多数多任务操作系统所提供的线程。当然，脚本可能不是实际上并行的，若脚本在单CPU上运行，CPU必须轮流执行。然而，从脚本程序员的角度，这是一种并行多线程的范式。

多数脚本系统通过**合作式多任务**（cooperative multitasking）来提供并行性。这是指脚本会一直执行，直至它主动交出执行权。相反，**抢占式多任务**（preemptive multitasking）可以在任何时候中断正在执行的脚本，令其他脚本得以执行。

要在脚本中实现合作式多任务，一个简单方法是容许脚本主动睡眠，然后等待某些相关的事情发生。脚本可能会等待指定的秒数，或是等待接收到某个事件。脚本也可以等待另一线程到达一个预定义的同步点。无论是哪一种原因，当脚本进入睡眠状态时，它就会被置于睡眠脚本线程列表中，并告诉虚拟机可以执行其他合乎条件的脚本。系统会一直检查唤醒脚本线程的条件，当其中一个条件成立就会唤醒等待该条件的脚本，并容许该脚本继续执行。

为了展示实际上该工作如何运作，我们可以看看一个多线程脚本的例子。此脚本负责管理两个角色和一道门的动画。两个角色被命令走到门前，每个角色到达门前的时间是不同的，也不可预知。当脚本在等待角色到达门前时，我们使脚本进入睡眠状态。当两个角色都到达后，其中一个角色会开门，这时该角色要播放“开门”动画。注意我们不希望在脚本中硬编码动画播放的持续时间。这么做，若动画师修改该动画，我们便不需要回去改脚本代码。因此我们使线程再度睡眠，等待动画结束。以下是达成目的的脚本，它采用了简单的类C伪代码语法。

```
function DoorCinematic
{
    thread Guy1
    {
        // 要求guy1走到门前
        CharacterWalkToPoint(guy1, doorPosition);
        WaitUntil(ARRIVAL); // 睡眠直至他到达那里

        // 他已到达，用信号通知其他线程
        RaiseSignal("Guy1Arrived");

        // 等待另一人也到达
        WaitUntil(SIGNAL, "Guy2Arrived");

        // 现在叫guy1播放“开门”动画
        CharacterAnimate(guy1, "OpenDoor");
        WaitUntil(ANIMATION_DONE);

        // OK，门已开启告诉另一线程
        RaiseSignal("DoorOpen");

        // 现在走过门
        CharacterWalkToPoint(guy1, beyondDoorPosition);
}

thread Guy2
{
    // 要求guy2走到门前
```

```
        CharacterWalkToPoint(guy2, doorPosition);
        WaitUntil(ARRIVAL); // 睡眠直至他到达那里

        // 他已到达，用信号通知其他线程
        RaiseSignal("Guy2Arrived");

        // 等待另一人也到达
        WaitUntil(SIGNAL, "Guy1Arrived");

        // 等待guy1为我开门
        WaitUntil(SIGNAL, "DoorOpen");

        // OK，门已开启告诉另一线程
        RaiseSignal("DoorOpen");

        // 现在走过门
        CharacterWalkToPoint(guy2, beyondDoorPosition);
    }
}
```

在上面的代码中，我们假设这假想的脚本语言提供了简单的语法，用来在单个函数中定义线程。在此定义了Guy1和Guy2两个线程。

Guy1线程告诉角色走到门前，到达后就睡眠。这里我们省略了一些细节，但假设脚本语言可以魔法般地使线程进入睡眠，等待游戏中的角色走到指定的目标地点。在现实中，我们可能需要令角色回传一个事件至脚本，然后脚本收到事件后就会苏醒过来。

当Guy1到达门前，其线程会做两件事（稍后再解释）。首先，它树立（raise）一个“Guy1Arrived”信号（signal）。然后，它继续睡眠等待另一个“Guy2Arrived”信号。若我们观察Guy2的线程，读者会发现相似的模式，只是反转而已。树立一个信号后等待另一个信号，此模式之目的在于同步两个线程。

在我们的假想脚本语言中，**信号**只是一个具名的布尔旗标。旗标的初始值为`false`，但当一个线程调用`RaiseSignal(name)`，该名字的旗标就变为`true`。其他线程可以进入睡眠，等待某个具名信号变成`true`。当信号变成`true`，就会唤醒那些线程并令它们继续执行。在此例子中，两个线程使用“Guy1Arrived”及“Guy2Arrived”信号互相同步。每个线程树立自己的信号并等待另一线程的信号。哪个信号先树立并不重要，只有当两个信号都被树立后，两个脚本才会苏醒。当它们苏醒时，它们就完美地同步了。图14.20展示了两种可能的情况，一个是Guy1先到达，另一个是Guy2先到达。但如图所示，哪个信号先树立并无所谓，当两个信号都被树立后，两个线程最终总是同步的。

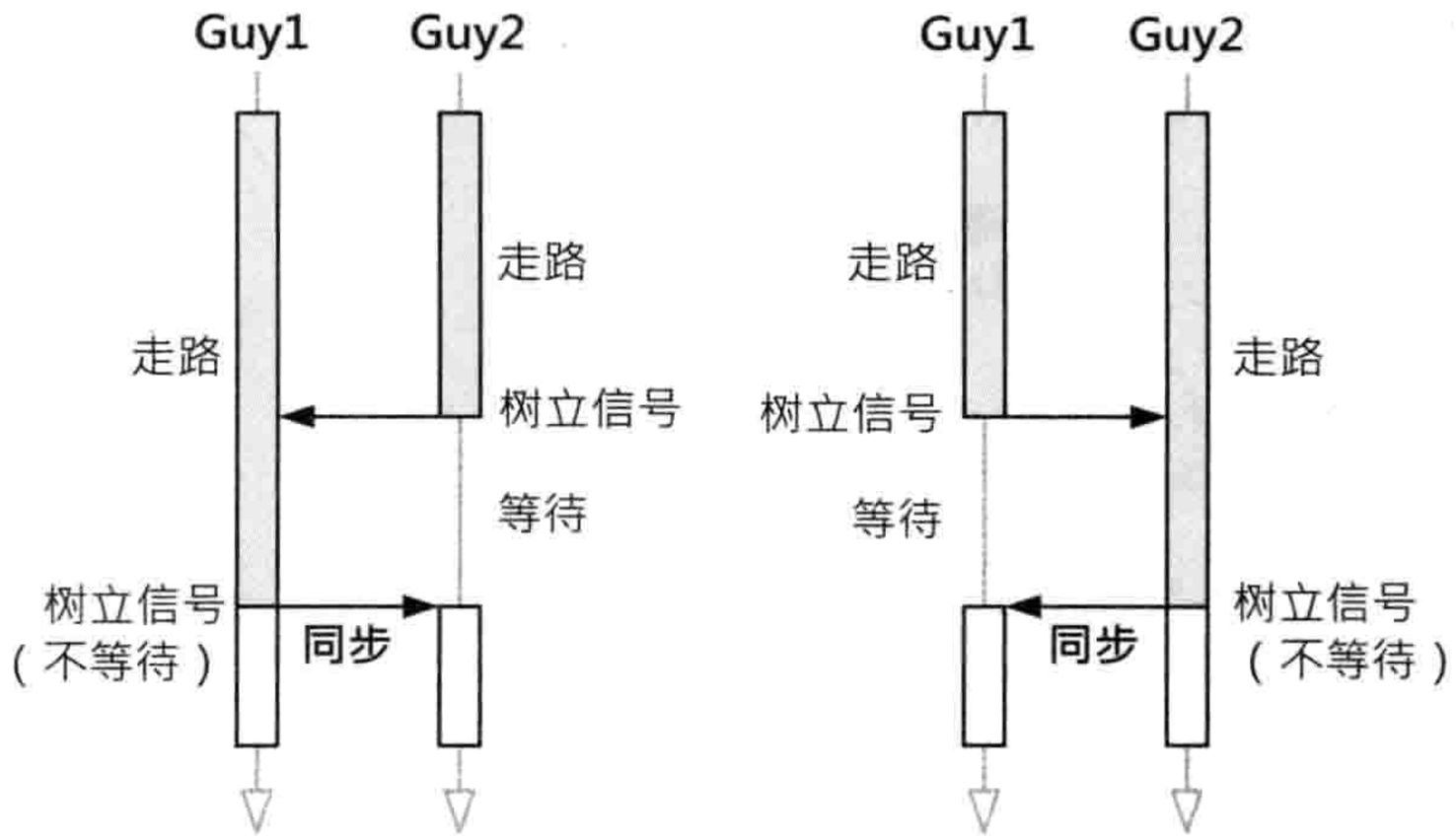


图 14.20: 这两个例子里，树立一个信号然后等待另一信号，这种简单模式可用于同步两个脚本线程。

14.9 高层次的游戏流程

游戏对象模型提供了一个基础，我们可以在此基础上实现丰富有趣的游戏对象类型，用以填充我们的游戏世界。然而，游戏对象模型本身只容许我们定义游戏世界中可存在的对象类型，以及它们各自的行为。游戏对象模型并没有说明玩家的目标，以及达成目标后有何事会发生，或是失败时什么命运会降临至玩家。

对于这些事情，我们需要有一个系统控制高层次的游戏流程（high-level game flow）。此系统通常可实现为有限状态机。每个状态代表单个玩家的目标或遭遇，每个目标或遭遇都关联到虚拟游戏世界中的某个地点。当玩家完成每个任务，状态机就会前进到下一个状态，游戏会向玩家展示下一组目标。此状态机也定义了玩家失败时会发生什么事情。通常，失败时会把玩家传送回当前状态的起点，让玩家再次挑战。有时候经过足够的失败次数之后，玩家的“命数”归零，那么就会返回主菜单，让玩家可以重新开始游戏。整个游戏的流程，从菜单至第一个关卡再至通关，都可以由此高层次的有限状态机控制。

顽皮狗的《杰克与达斯特》和《神秘海域》系列的任务系统都是基于状态机的高层次游戏流程系统。该系统容许状态（顽皮狗称之为任务）为线性序列。但它也容许并行的任务，当中一个任务可能分支为两个或以上的并行任务，最终再合并为主任务序列。这个并行任务的功能令顽皮狗的任务图异于一般状态机，因为状态机在同一时间通常只能处于单个状态。

第五部分

总结

第15章　还有更多内容吗

恭喜！你已经完满地通过游戏引擎架构的旅程（希望仍然完好无损）。但愿你能从中学习到不少有关典型游戏引擎的主要组件。然而，每个旅程之终仍是另一旅程之始。在本书以外，我们仍有许多可以学习的领域。由于技术与计算机硬件的不断进步，游戏可以做到的事情会越来越多，也就是说，我们需要研发更多的引擎系统支持这些事情。另一方面，本书专注讲述游戏引擎本身，还未开始讨论游戏性编程这个丰富的世界，其内容或可填满许多卷书。

在以下几节中，笔者会指出几个本书没有足够篇幅去讨论的引擎及游戏性系统，并为有兴趣学习这些内容的读者提供一些资源。

15.1　一些未谈及的引擎系统

15.1.1　音频

笔者在1.6.13节中提及音频通常居于游戏开发的次位，使音频工程师、音效设计师、配音演员及作曲家非常懊恼，尤其是他们努力为虚拟游戏世界创造了那么重要的第四维度。可是，遗憾的是，这情况也出现在本书中，笔者无篇幅及时间论述游戏音频系统，或许要等下一版。（符合这个漫长又痛苦的不幸传统，音频再次壮烈牺牲！）

幸好，坊间有许多书籍及线上资源提供音频开发的信息。首先，笔者建议阅读微软XACT音效制作工具及API的文档[1]。XACT支持几乎一般游戏程序员所需要的音频功能，而且其文档颇易阅读。另外，《游戏编程精髓（Game Programming Gems）》丛书也包括大量音频相关的文章，可参见[7]第6部分及[40]第6部分。

[1] http://msdn.microsoft.com/en-us/library/bb172329(VS.85).aspx

15.1.2 影片播放器

许多游戏含有影片播放器以显示预渲染的影片，这些影片也称作全动视频（full-motion video，FMV）。影片播放器的基本组件包括文件流I/O系统的接口（见6.1.3节）、用于解压视讯流的解码器，以及某种与音频系统音轨同步的机制。

市面上有许多不同的视频编码标准，适合不同的应用所需。例如，VCD和DVD分别采用MPEG-1和MPEG-2（H.262）编码。H.261和H.263标准是主要为视频会议而设计的。游戏通常用如MPEG-4 part 2（如DivX）、MPEG-4 Part 10（H.264）、Windows Media Video（WMV）或Bink Video（由Rad Game Tools公司为游戏特制的标准）。想了解更多视频编码的内容可参考维基百科[2]及Bink Video[3]。

15.1.3 多人网络

虽然我们已谈及多人游戏架构及网络（如1.6.14、7.7及14.8.3.2节），本书在这方面是远远不够全面的。关于多人网络游戏的论述可参考[3]。

15.2 游戏性系统

游戏当然不单只有引擎。在（第14章讨论的）游戏性基础层之上，还有许多形形色色与游戏类型/个别游戏相关的游戏性系统。这些系统配合本书所谈及的许多游戏引擎技术，凝聚成游戏的生命。

15.2.1 玩家机制

玩家机制当然是最重要的游戏性系统。玩家机制及游戏性的风格定义了每种游戏类型（genre），而游戏类型中的每个游戏也有其独特设计。因此，玩家机制是一个庞大的题目，它涉及人机界面设备（HID）系统、动作模拟、碰撞检测、动画、音频，不用说当然还要与其他游戏性系统整合，诸如游戏摄像机、武器、掩护点、专门的移动方式（梯子、摆动的藤蔓等）、载具系统、谜题机制等。

显然玩家机制对不同游戏有许多区别，所以不可能依靠某个资源就能学习所有的玩家机制。最佳的学习方式，可以是每次学习单个游戏类型。玩游戏，试图对其玩家机制做反向

[2] http://en.wikipedia.org/wiki/Video_codec

[3] http://www.radgametools.com/bnkmain.htm

工程。然后尝试自己实现它们！作为阅读的起点，读者可参看[7]中4.11节，当中讨论了玛里奥式平台游戏玩家机制。

15.2.2 摄像机

游戏的摄像机系统与玩家机制同样重要。事实上，摄像机可以成就游戏体验，也可以毁掉游戏体验。每个游戏类型往往有其摄像机控制风格，虽然同类型的游戏会有少许区别（也有些有巨大区别！）。参见[6]的4.3节，对基本游戏摄像机控制技术的讨论。以下概要列出一些三维游戏最流行的摄像机，但请注意这远非一个完整的列表。

- **注视摄像机**（look-at camera）：这类摄像机围绕一个目标点旋转，并能相对该点做前后移动。
- **跟随摄像机**（follow camera）：这类摄像机常用于平台游戏、第三人称游戏、基于载具的游戏。它的行为很类似注视摄像机，聚焦于玩家的角色/化身/载具，但其运动通常是滞后于玩家的。跟随摄像机也包含高级的碰撞检测及回避逻辑，并且给玩家对摄像机相对化身的定向有某程度的控制。
- **第一人称摄像机**（first-person camera）：随着玩家角色在游戏世界中的移动，第一人称摄像机维持固定于角色的虚拟眼睛位置。玩家通常能通过鼠标或手柄完全控制摄像机的方向。摄像机注视的方向也会直接转化为玩家武器的瞄准方向。这通常显示为屏幕下方一双手臂握着武器，以及画面中间的十字线。
- **即时战略摄像机**（RTS camera）：即时战略游戏及上帝模拟游戏通常会使用浮于地形上的摄像机，以某角度朝向下。玩家可以控制摄像机在地形上水平移动（pan），但通常不能直接控制偏航角和俯仰角。
- **电影摄像机**（cinematic camera）：多数三维游戏都至少有些电影时段，当中的摄像机会更类似电影中的运镜效果，而不受游戏本身的约束。

15.2.3 人工智能

多数基于角色的游戏含有另一个重要组件——人工智能（artificial intelligence，AI）。底层的AI系统技术通常包括路径搜寻（path finding，常使用著名的A*算法）、感知系统（perception system，视线感知、视锥感知、环境的理解等），以及某形式的记忆。

在这些基础之上会实现一些角色控制逻辑。角色控制系统判断角色如何做出一些具体动作，如角色运动（character locomotion）、通过特殊的地形特征、使用武器、驾驶载具、掩护等。这些动作通常涉及引擎中的碰撞、物理、动画系统等复杂接口。11.11及11.12节详

述了角色控制。

在角色控制层之上，通常AI系统还包含目标设定、决策逻辑、情感状态、群体行为（协调/coordination、包抄/flanking、群众/crowd、群集/flocking），也有可能含一些高级功能，如从过去错误中学习，或是适应一个改变中的环境。

当然，“人工智能”一词用于游戏是不尽准确的。游戏人工智能相比真正的人工智能，通常有点戏法、假象的成分。我们必须了解，游戏中真正重要的是玩家的体验。典型的例子之一是《光环》。当Bungie起初实现其人工智能系统时，加入了一简单规则：小兵的队长身亡时，全部小兵们会逃跑。经过一轮又一轮试玩（playtest），众人都不明白**为何**小兵们会逃跑。即使Bungie的团队多次修改动画及人工智能行为，玩家们仍然不了解两者的关联。最终，开发者令其中一个小兵喊：“队长死了！逃吧！”此故事只是想说明，若玩家不能感知游戏人工智能逻辑背后的意义，这些逻辑就没有什么意义。

人工智能编程是一个内容丰富的题目，本书肯定没法详述，建议读者参考[16]、[6]的第3部分、[7]的第3部分、及[40]的第3部分。

15.2.4 其他游戏性系统

除了玩家机制、摄像机、人工智能以外，游戏显然还有更多部分。有些游戏有可驾驶的载具，实现了特殊类型的武器，通过动力学物理模拟容许玩家破坏游戏中的环境，让玩家创作自己的角色，建立自定义的关卡，需要玩家解谜……当然，这些游戏类型或具体游戏相关的功能列表，以及为实现它们而设的专门软件系统，都是永无止尽的。游戏性系统如同游戏一样，非常丰富且多元化。也许，这里就是你作为游戏程序员新旅程之始!

参考文献

[1] Tomas Akenine-Moller, Eric Haines, and Naty Hoffman. *Real-Time Rendering (3rd Edition)*. Wellesley, MA: A K Peters, 2008. 中译本：《实时计算机图形学（第2版）》，普建涛译，北京大学出版社，2004.

[2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Resding, MA: Addison-Wesley, 2001. 中译本：《C++设计新思维：泛型编程与设计模式之应用》，侯捷/於春景译，华中科技大学出版社，2003.

[3] Grenville Armitage, Mark Claypool and Philip Branch. *Networking and Online Games: Understanding and Engineering Multiplayer Internet Games*. New York, NY: John Wiley and Sons, 2006.

[4] James Arvo (editor). *Graphics Gems II*. San Diego, CA: Academic Press, 1991.

[5] Grady Booch, Robert A. Maksimchuk, Michael W. Engel, Bobbi J. Young, Jim Conallen, and Kelli A. Houston. *Object-Oriented Analysis and Design with Applications (3rd Edition)*. Reading, MA: Addison-Wesley, 2007. 中译本：《面向对象分析与设计（第3版）》，王海鹏/潘加宇译，电子工业出版社，2012.

[6] Mark DeLoura (editor). *Game Programming Gems*. Hingham, MA: Charles River Media, 2000. 中译本：《游戏编程精粹1》，王淑礼译，人民邮电出版社，2004.

[7] Mark DeLoura (editor). *Game Programming Gems 2*. Hingham, MA: Charles River Media, 2001. 中译本：《游戏编程精粹2》，袁国忠译，人民邮电出版社，2003.

[8] Philip Dutré, Kavita Bala and Philippe Bekaert. *Advanced Global Illumination (2nd Edition)*. Wellesley, MA: A K Peters, 2006.

[9] David H. Eberly. *3D Game Engine Design: A Practical Approach to Real-Time Computer Graphics*. San Francisco, CA: Morgan Kaufmann, 2001. 国内英文版：《3D游戏引擎设计：实时计算机图形学的应用方法（第2版）》，人民邮电出版社，2009.

[10] David H. Eberly. *3D Game Engine Architecture: Engineering Real-Time Applications*

with Wild Magic. San Francisco, CA: Morgan Kaufmann, 2005.

[11] David H. Eberly. *Game Physics.* San Francisco, CA: Morgan Kaufmann, 2003.

[12] Christer Ericson. *Real-Time Collision Detection.* San Francisco, CA: Morgan Kaufmann, 2005. 中译本：《实时碰撞检测算法技术》，刘天慧译，清华大学出版社，2010.

[13] Randima Fernando (editor). *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics.* Reading, MA: Addison-Wesley, 2004. 中译本：《GPU精粹：实时图形编程的技术、技巧和技艺》，姚勇译，人民邮电出版社，2006.

[14] James D. Foley, Andries van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics: Principles and Practice in C (2nd Edition).* Reading, MA: Addison-Wesley, 1995. 中译本：《计算机图形学原理及实践——C语言描述》，唐泽圣/董士海/李华/吴恩华/汪国平译，机械工业出版社，2004.

[15] Grant R. Fowles and George L. Cassiday. *Analytical Mechanics (7th Edition).* Pacific Grove, CA: Brooks Cole, 2005.

[16] John David Funge. *AI for Games and Animation: A Cognitive Modeling Approach.* Wellesley, MA: A K Peters, 1999.

[17] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissiddes. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1994. 中译本：《设计模式：可复用面向对象软件的基础》，李英军/马晓星/蔡敏/刘建中译，机械工业出版社，2005.

[18] Andrew S. Glassner (editor). *Graphics Gems I.* San Francisco, CA: Morgan Kaufmann, 1990.

[19] Paul S. Heckbert (editor). *Graphics Gems IV.* San Diego, CA: Academic Press, 1994.

[20] Maurice Herlihy, Nir Shavit. *The Art of Multiprocessor Programming.* San Francisco, CA: Morgan Kaufmann, 2008. 中译本：《多处理器编程的艺术》，金海/胡侃译，机械工业出版社，2009.

[21] Roberto Ierusalimschy, Luiz Henrique de Figueiredo and Waldemar Celes. *Lua 5.1 Reference Manual.* Lua.org, 2006.

[22] Roberto Ierusalimschy. *Programming in Lua, 2nd Edition.* Lua.org, 2006. 中译本：《Lua程序设计（第2版）》，周惟迪译，电子工业出版社，2008.

[23] Isaac Victor Kerlow. *The Art of 3-D Computer Animation and Imaging (2nd Edition).* New York, NY: John Wiley and Sons, 2000.

[24] David Kirk (editor). *Graphics Gems III.* San Francisco, CA: Morgan Kaufmann, 1994.

[25] Danny Kodicek. *Mathematics and Physics for Game Programmers.* Hingham, MA: Charles River Media, 2005.

[26] Raph Koster. *A Theory of Fun for Game Design.* Phoenix, AZ: Paraglyph, 2004. 中译本:《快乐之道:游戏设计的黄金法则》,姜文斌等译,百家出版社,2005.

[27] John Lakos. *Large-Scale C++ Software Design.* Reading, MA: Addison-Wesley, 1995. 中译本:《大规模C++程序设计》,李师贤/明仲/曾新红/刘显明译,中国电力出版社,2003.

[28] Eric Lengyel. *Mathematics for 3D Game Programming and Computer Graphics (2nd Edition).* Hingham, MA: Charles River Media, 2003.

[29] Tuoc V. Luong, James S. H. Lok, David J. Taylor and Kevin Driscoll. *Internationalization: Developing Software for Global Markets.* New York, NY: John Wiley & Sons, 1995.

[30] Steve Maguire. *Writing Solid Code: Microsoft's Techniques for Developing Bug Free C Programs.* Bellevue, WA: Microsoft Press, 1993. 国内英文版:《编程精粹:编写高质量C语言代码》,人民邮电出版社,2009.

[31] Scott Meyers. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition).* Reading, MA: Addison-Wesley, 2005. 中译本:《Effective C++:改善程序与设计的55个具体做法(第3版)》,侯捷译,电子工业出版社,2011.

[32] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Reading, MA: Addison-Wesley, 1996. 中译本:《More Effective C++:35个改善编程与设计的有效方法(中文版)》,侯捷译,电子工业出版社,2011.

[33] Scott Meyers. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library.* Reading, MA: Addison-Wesley, 2001. 中译本:《Effective STL:50条有效使用STL的经验》,潘爱民/陈铭/邹开红译,电子工业出版社,2013.

[34] Ian Millington. *Game Physics Engine Development.* San Francisco, CA: Morgan Kaufmann, 2007.

[35] Hubert Nguyen (editor). *GPU Gems 3.* Reading, MA: Addison-Wesley, 2007. 中译本:《GPU精粹3》,杨柏林/陈根浪/王聪译,清华大学出版社,2010.

[36] Alan W. Paeth (editor). *Graphics Gems V.* San Francisco, CA: Morgan Kaufmann, 1995.

[37] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion (2nd Edition).* Sebastopol , CA: O'Reilly Media, 2008. (常被称作

"The Subversion Book"，线上版本http://svnbook.red-bean.com.）国内英文版：《使用Subversion进行版本控制》，开明出版社，2009.

[38] Matt Pharr (editor). *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Reading, MA: Addison-Wesley, 2005. 中译本：《GPU精粹2：高性能图形芯片和通用计算编程技巧》，龚敏敏译，清华大学出版社，2007.

[39] Bjarne Stroustrup. *The C++ Programming Language, Special Edition (3rd Edition).* Reading, MA: Addison-Wesley, 2000. 中译本《C++程序设计语言（特别版）》，裘宗燕译，机械工业出版社，2010.

[40] Dante Treglia (editor). *Game Programming Gems 3.* Hingham, MA: Charles River Media, 2002. 中译本：《游戏编程精粹3》，张磊译，人民邮电出版社，2003.

[41] Gino van den Bergen. *Collision Detection in Interactive 3D Environments.* San Francisco, CA: Morgan Kaufmann, 2003.

[42] Alan Watt. *3D Computer Graphics (3rd Edition).* Reading, MA: Addison Wesley, 1999.

[43] James Whitehead II, Bryan McLemore and Matthew Orlando. *World of Warcraft Programming: A Guide and Reference for Creating WoW Addons.* New York, NY: John Wiley & Sons, 2008. 中译本：《魔兽世界编程宝典：World of Warcraft Addons完全参考手册》，杨柏林/张卫星/王聪译，清华大学出版社，2010.

[44] Richard Williams. *The Animator's Survival Kit.* London, England: Faber & Faber, 2002. 中译本：《原动画基础教程：动画人的生存手册》，邓晓娥译，中国青年出版社，2006.

中文索引

C

D

E

F

H

I

J

K

L

M

Q

R

S

T

Z

英文索引

B

C

G

O

P

Q

R

S

T

U

V

[G e n e r a l I n f o r ma t i o n]
书名= 游戏引擎架构
作者= （美）格雷戈瑞著
页数= 7 7 2
S S 号= 1 3 5 2 2 5 8 3
出版日期= 2 0 1 4 . 0 2
出版社= 北京：电子工业出版社
I S B N 号= 9 7 8 - 7 - 1 2 1 - 2 2 2 8 8 - 7
中图法分类号= T P 3 9 1 . 4 1
原书定价= 1 2 8 . 0 0
参考文献格式= （美）格雷戈瑞著. 游戏引擎架构. 北京：电子工业出版社，2 0 1 4 . 0 2 .
内容提要= 本书入选2 0 0 9 年度游戏开发杂志前沿奖决赛名单，内容结合游戏引擎开发的理论及实践，并全面函括多个范畴。当中描述的概念及技巧是实际应用在游戏公司里，例如电艺E A 及N a u g h t y D o g 。一些例子有时候会建基于个别科技，但讨论的内容延伸超越个别引擎或A P I 。读者可跟据参考及引用？