Broadview
www.broadview.com.cn

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

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2月第1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

目录

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 Algoithms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

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：

$$y(t) = \frac{1}{2}gt^2 + v_0 t + y_0 \tag{1.1}$$

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细致程度技巧，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。

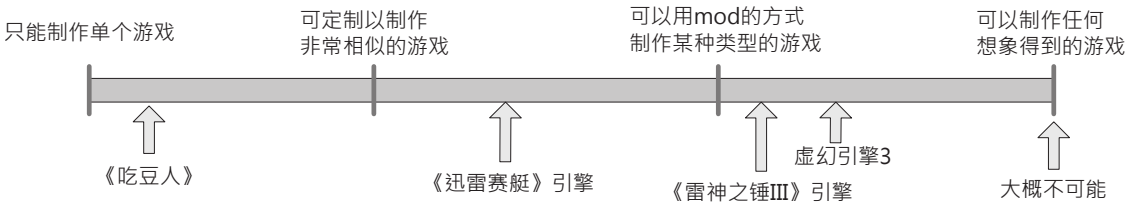

图 1.1: 游戏引擎复用性连续谱。

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）》、《战栗时空（Half-Life）》《反恐精英（Counter-Strike）》和《使命之唤（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: 《使命之唤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。
- 能侦测复杂按妞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

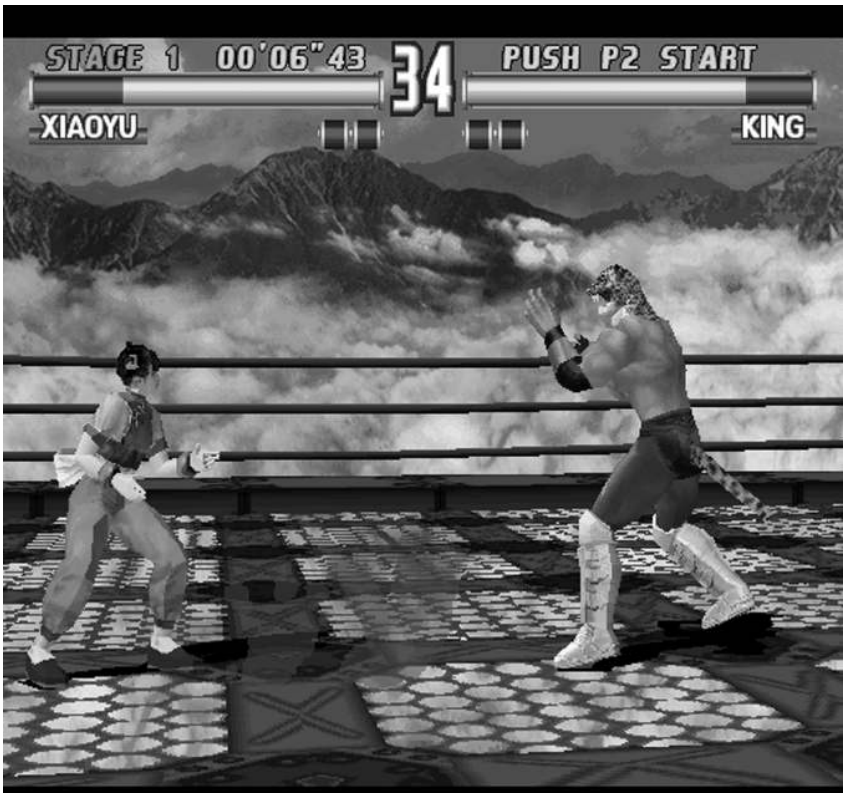

图 1.5: 《铁拳3（Tekken 3）》（PlayStation）。

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）。着色器模拟了表面下散射（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）。

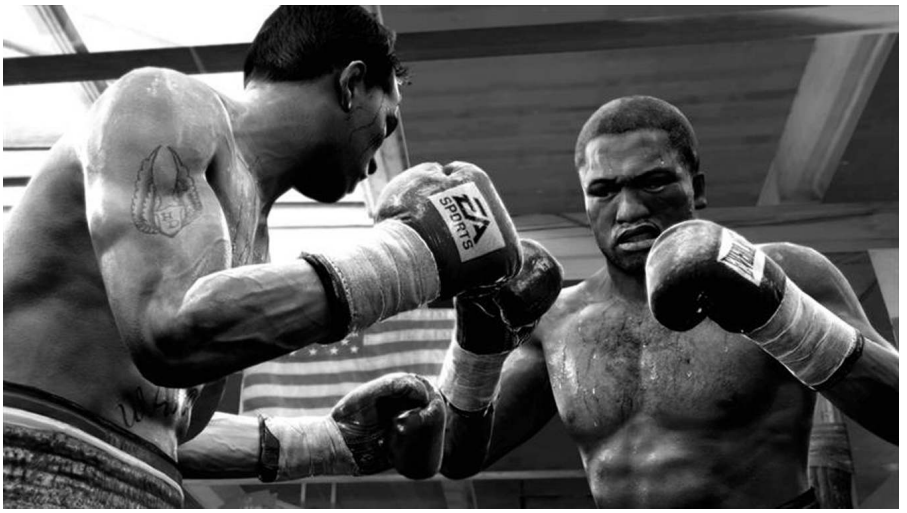

图 1.6: 《拳击之夜3（Fight Night Round 3）》（PlayStation 3）。

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）的次文化，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

[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

引擎上采用各种优化。

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。

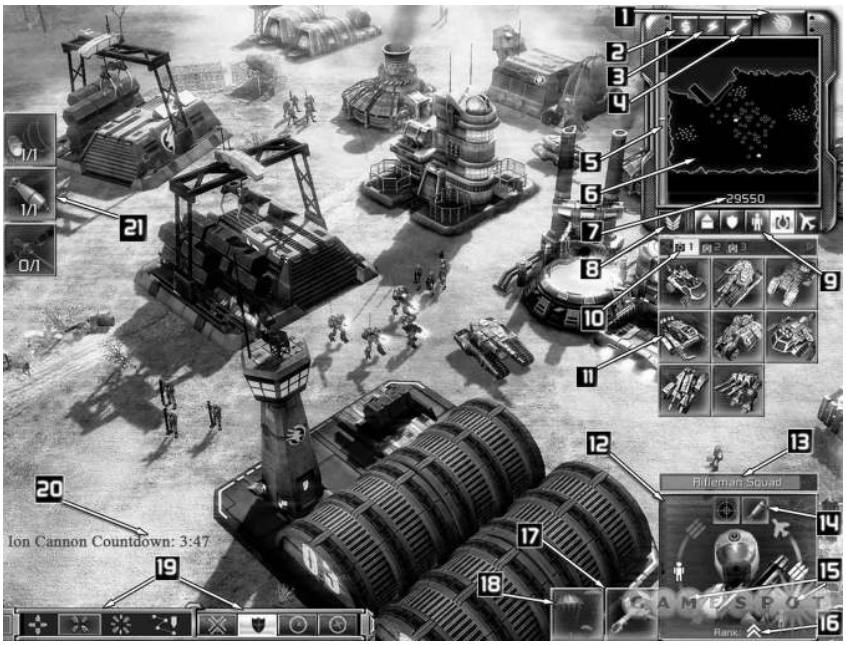

图 1.9: 《命令与征服3（Command & Conquer 3）》（PC）。

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《战栗时空》）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，制作了红极一时的游戏《战栗时空2（Half Life 2）》、其续作《战栗时空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。

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。

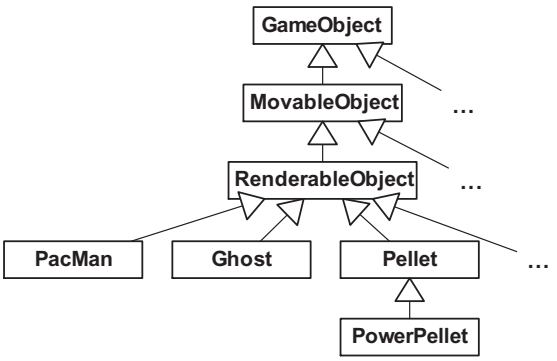

图 14.2: 《吃豆人》的假想类层次结构。

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。

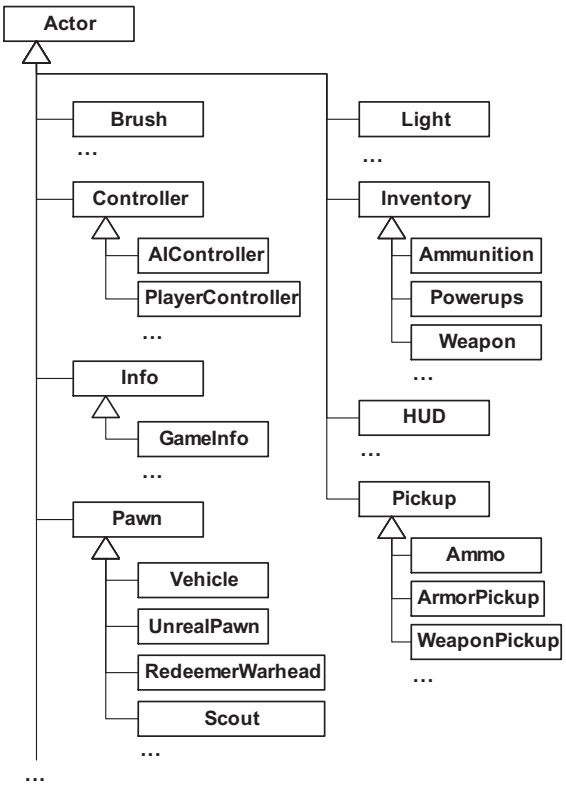

图 14.3: 《虚幻竞技场2004（Unreal Tournament 2004）》的游戏对象类层次结构摘录。

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。

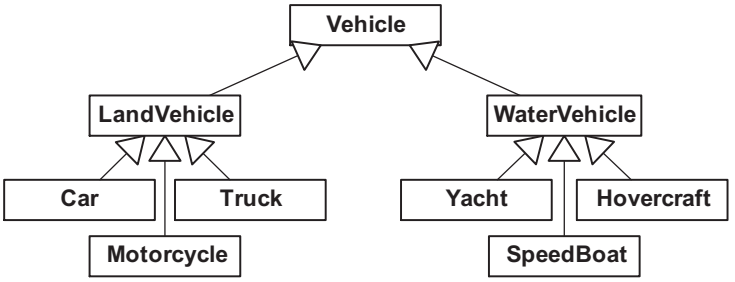

图 14.4: 好像能合乎逻辑地描述各类载具的类层次结构。

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。

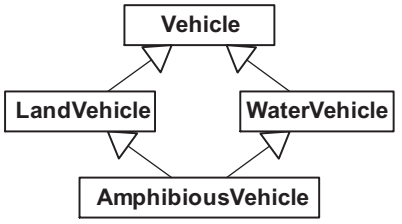

图 14.5: 水陆两用载具的钻石形类层次结构。

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。

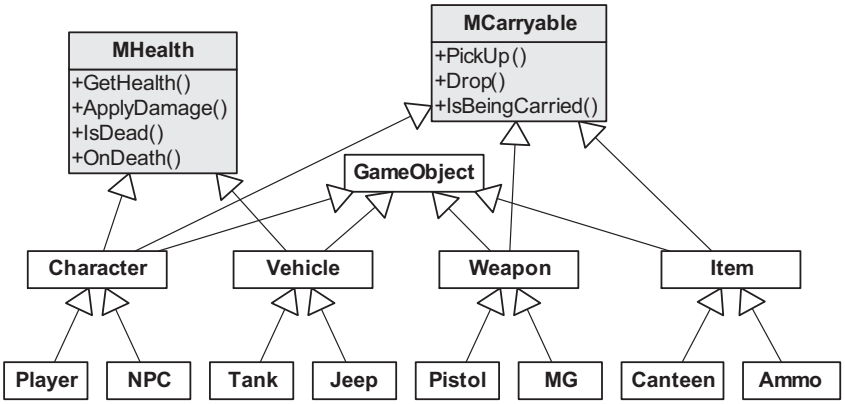

图 14.6: 含mix-in类的类层次结构。任何继承MHealth mix-in类的类会加血量信息，并可以被杀。MCarryable mix-in类可以令其派生类的对象被角色携带。

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。

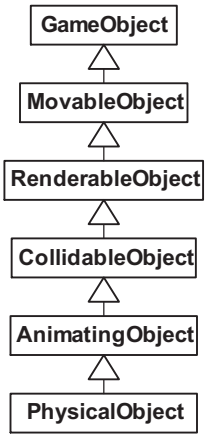

图 14.7: 假想游戏对象层次结构，仅以继承连接各类。

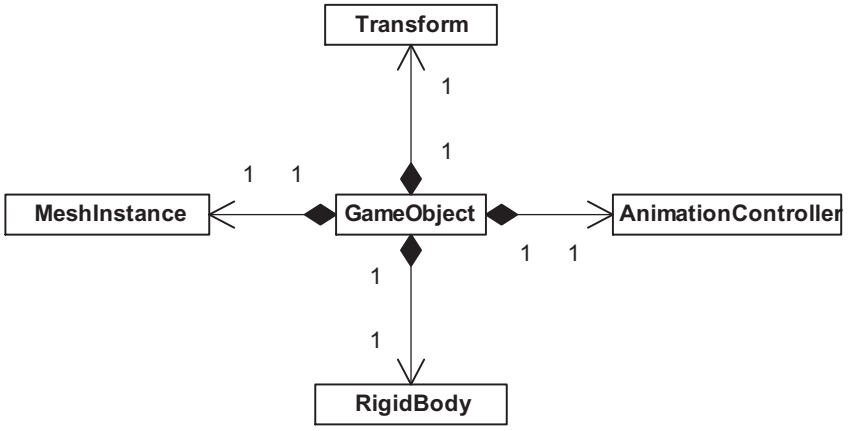

图 14.8: 重构假想的游戏对象层次结构，相对继承更偏爱类的合成。

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。

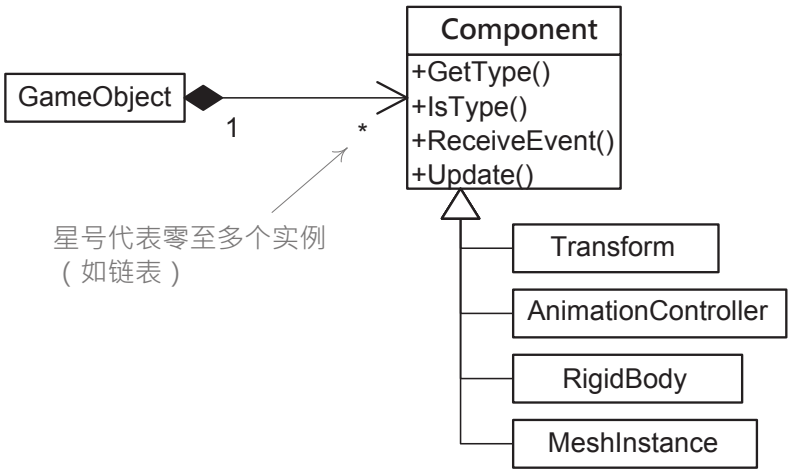

图 14.9: 组件链表可提升弹性，因为枢纽游戏对象不用关注各组件的细节。

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），

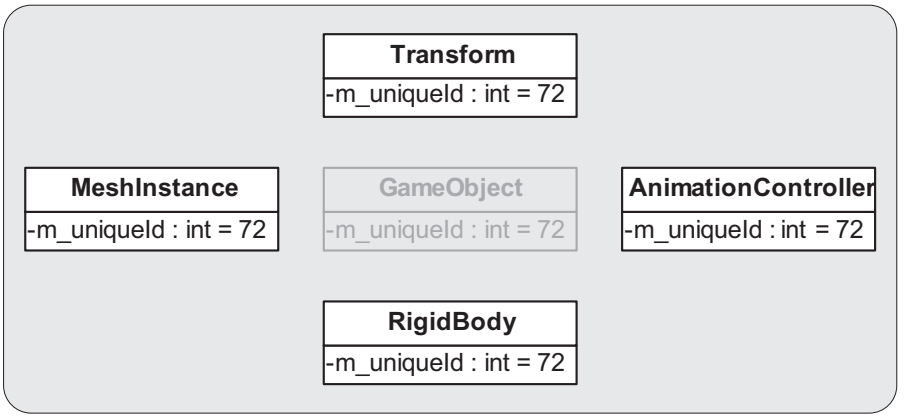

图 14.10: 在纯组件模型中，逻辑游戏对象是由许多组件组成的，但组件只是通过唯一标识符间接地连接在一起。

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建康值作为回应。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）。

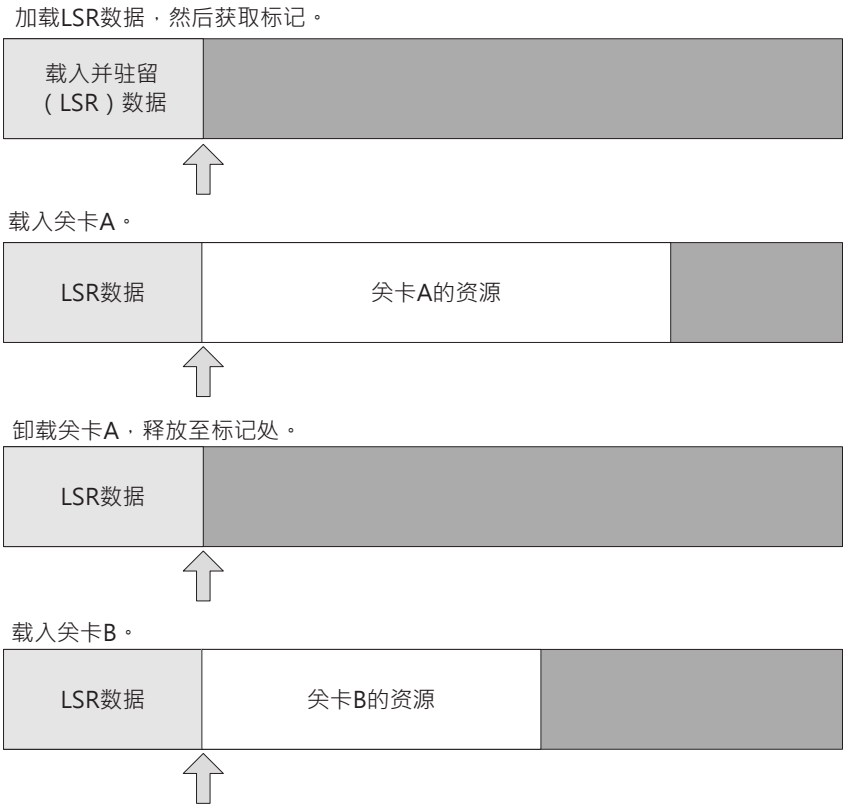

图 14.11: 对于同时间只有一个关卡的世界加载系统，基于堆栈的内存分配器非常适合。

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。

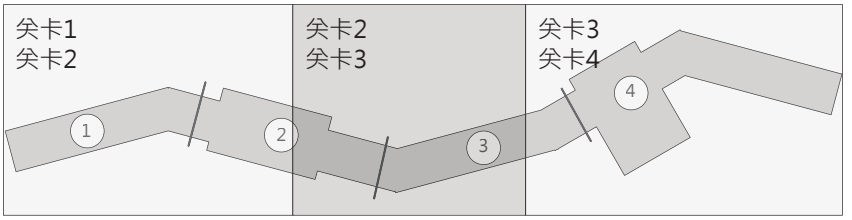

图 14.12: 把游戏世界分割成组块。每个关卡加载区域对应一个组块请求列表，细心整理这些信息以保证玩家永不会看到组块在视野内弹出或消失。

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。

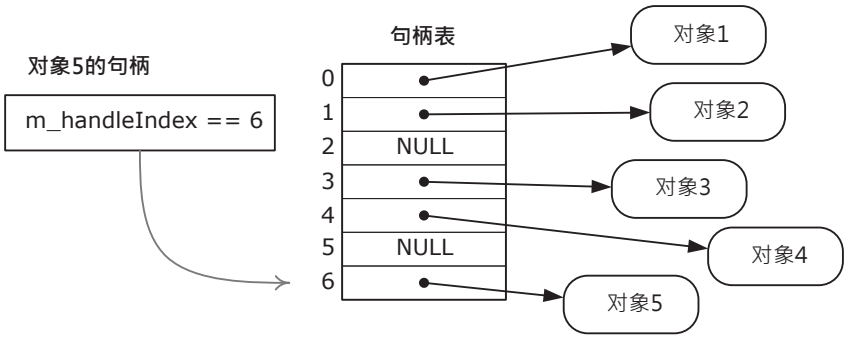

图 14.13: 句柄表含有原始指针。句柄仅是此表的索引。

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。

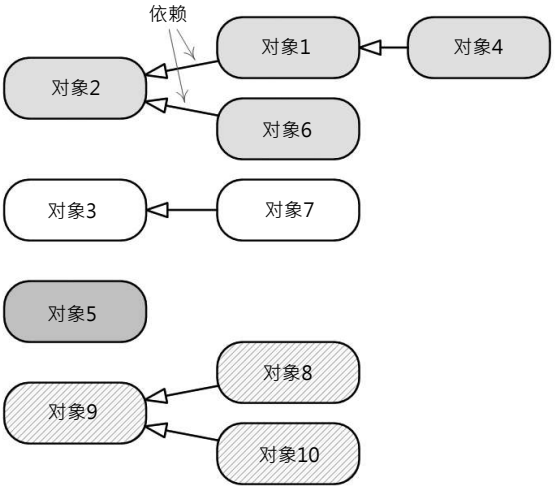

图 14.14: 游戏对象间的依赖关系可以视为一个依赖树林。

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。

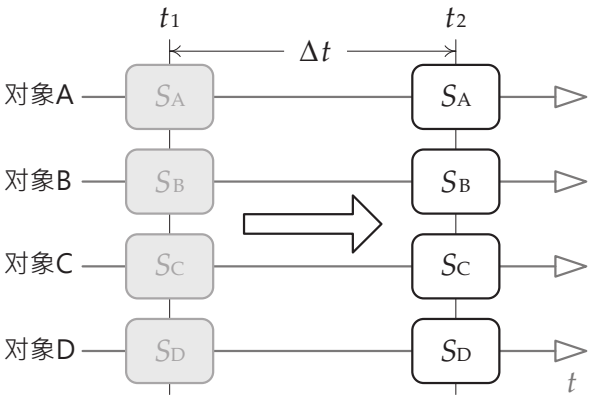

图 14.15: 理论上在每个游戏循环迭代中，所有游戏对象的状态是瞬间及并行地进行更新的。

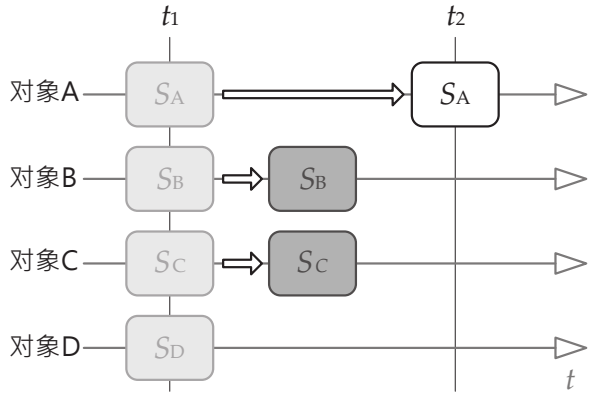

图 14.16: 实践上，游戏对象是逐一更新状态的。这意味着，在更新循环的某些时刻，一些对象（如A）认为时间是t_2而其他对象（如D）则认为仍是t_1。一些对象（如B、C）可能只更新了一部分，所以它们的状态是内部不一致的。这些对象的状态位于t_1和t_2之间。

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。

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

K

Q

R

S

T

Z

英文索引

B

C

G

Q

R

S

T

U

V

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《DirectX9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。

上架建议：游戏开发/游戏设计

新浪微博 weibo.com @博文视点Broadview

策划编辑：张春雨
责任编辑：付　睿
封面设计：侯士卿

ISBN 978-7-121-22288-7

9 787121 222887

定价：128.00元