THOMSON

•游戏开发经典丛书•



本书配光盘

(美) Brian Schwab 著
林龙信 张波涛 译

AI游戏引擎程序设计

AI Game Engine Programming

CENGAGE Learning™

AI Game Programming Engine

清华大学出版社

游戏引擎设计

AI Game Engine Programming

AI(Artificial Intelligence，人工智能)在游戏中是一个崭新的概念，如今得到了普遍的关注和重视。人们期望与具有更高智能水平的AI对手对抗，这促进了游戏AI技术的发展。特别是近来，已经出现了模仿人类玩家的游戏风格和反应的AI对手，为游戏问题提供创造性解决方案的AI对手，甚至还出现了具有人类心情和情感的AI对手。

本书为游戏开发人员创建现代游戏的AI引擎提供了工具和必要的指导。带领读者从理论进入实际的游戏开发，并给出可用的代码框架，详细说明技术的实现方法。另外，该书综合阐述了不同技术的使用范围，并囊括了普遍存在的瓶颈、设计上应该考虑的问题以及优化策略。所有这些内容对游戏AI引擎开发人员都是必不可少的参考资料。

有关CD-ROM的内容

本书附带的CD-ROM中包括了本书涉及到的所有在Microsoft Visual C++6.0下编译通过的源代码、编译得到的二进制文件、OpenGL的一个封装库GLUT和Lua语言库、一些网络资源的超链接和书中出现的所有图片等。

作者简介

Brian Schwab具备十几年的游戏程序设计经验，在Angel Studios公司和DreamWorks公司拥有游戏和AI程序设计的关键职位。现在他是索尼娱乐公司的资深AI程序员。

技术支持：http://www.charlesriver.com

ISBN 978-7-302-16312-1
9 787302 163121 >
定价：59.80元(含光盘)

·游戏开发经典丛书·

AI 游戏引擎程序设计

(美) Brian Schwab 著

林龙信 张波涛 译

清华大学出版社

北 京

Brian Schwab
AI Game Engine Programming
EISBN：1-58450-344-0
Copyright © 2004 by Thomson, a division of Thomson Learning
Original language published by Thomson Learning (a division of Thomson Learning Asia Pte Ltd).
All rights reserved.

Tsinghua University Press is authorized by Thomson Learning to publish and distribute exclusively this Simplified Chinese edition. This edition is authorized for sale in the People's Republic of China only (excluding Hong Kong, Macao SAR and Taiwan). Unauthorized export of this edition is a violation of the Copyright Act. No part of this publication may be reproduced or distributed by any means, or stored in a database or retrieval system, without the prior written permission of the publisher.

本中文简体字翻译版由汤姆森学习出版集团授权清华大学出版社独家出版发行。此版本仅限在中华人民共和国境内(不包括中国香港、澳门特别行政区及中国台湾地区)销售。未经授权的本书出口将被视为违反版权法的行为。未经出版者预先书面许可，不得以任何方式复制或发行本书的任何部分。

北京市版权局著作权合同登记号　图字：01-2006-7223

本书封面贴有 Thomson 防伪标签，无标签者不得销售。
版权所有，侵权必究。侵权举报电话：010-62782989　13501256678　13801310933

图书在版编目(CIP)数据

AI 游戏引擎程序设计/(美)施瓦布(Schwab, B.)著；林龙信，张波涛译. —北京：清华大学出版社，2007.11
(游戏开发经典丛书)
书名原文：AI Game Engine Programming
ISBN 978-7-302-16312-1

Ⅰ. A…　Ⅱ. ①施…　②林…　③张…　Ⅲ. 游戏—软件开发　Ⅳ. TP311.5

中国版本图书馆 CIP 数据核字(2007)第 159567 号

责任编辑：王　军　梁卫红
装帧设计：康　博
责任校对：胡雁翎
责任印制：何　芊
出版发行：清华大学出版社　　**地　　址**：北京清华大学学研大厦 A 座
http://www.tup.com.cn　**邮　　编**：100084
c—service@tup.tsinghua.edu.cn
社 总 机：010-62770175　**邮购热线**：010-62786544
投稿咨询：010-62772015　**客户服务**：010-62776969
印 刷 者：北京鑫丰华彩印有限公司
装 订 者：三河市李旗庄少明装订厂
经　　销：全国新华书店
开　　本：185×260　**印　张**：29.75　**字　数**：724 千字
附光盘 1 张
版　　次：2007 年 11 月第 1 版　　**印　　次**：2007 年 11 月第1 次印刷
印　　数：1～4000
定　　价：59.80 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：(010)62770177 转 3103　产品编号：023671－01

译 者 序

如今电脑游戏正变得越来越普及，与此相应，游戏编程也成了计算机行业内的一个热门领域。特别是随着人工智能理论研究的进展，许多现代游戏都具有了很多的智能成分。如何理解智能？游戏人工智能是什么？有哪些人工智能技术可以应用到游戏编程之中？它们又该如何实现？这些都是摆在智能游戏开发者面前的问题。

但是，现在市面上关于一般的游戏编程的书还不多，而关于“游戏人工智能”编程的书则更少。Brian Schwab 先生的这本 *AI Game Engine Programming* 可谓是给我们这些智能游戏开发者带来了一阵及时雨，它以一种独特深邃的视角，从众多游戏编程书籍中脱颖而出，是一本非常难得的佳作。

本书以严谨的理论说明、丰富的游戏示例以及翔实的程序代码对人工智能游戏引擎编程知识进行了很好的归纳和阐述。尽管本书是为职业的游戏 AI 程序员，以及有志于将其兴趣领域扩展到 AI 的程序员编写的，但其对各个层次的游戏编程人员都有很好的启发作用。希望通过本书的学习，读者能够快速地掌握人工智能游戏引擎的编程技巧，将人工智能真正地应用到所开发的游戏之中。

本书涉及到了相当多的游戏，有些可能是我们相当熟悉的，有些则可能非常陌生。为了防止翻译过来的游戏名称产生歧义，我们仍然采用游戏的英文原名(某些非常熟悉的游戏则在英文后标注了中文游戏名)。另外，由于本书中具有大量的游戏专业词汇，尽管借助了强大的网络搜索工具，但对于某些具体词汇的译法仍然比较模糊，同行的翻译也不是很一致，故在译稿中仅仅按实际含义给出。

本书由林龙信、张波涛翻译完成，但翻译工作是集体智慧的结晶。除了译者外，还得到了翻译工作的组织者以及许多网络游戏玩家的大量指导和帮助，在此表示衷心的感谢。

由于译者水平有限，书中难免会存在不妥之处，敬请广大读者批评指正。

译 者

2007 年 4 月于湖南长沙

前　言

现在市面上关于一般游戏编程的书还不多，而关于“游戏人工智能(Game Artificial Intelligence，游戏 AI)”编程的书则更少。本书将为读者介绍 4 个方面的内容，这些内容都是目前市面上的游戏编程书籍较少涉及到的。

(1) 对“游戏 AI”进行清楚的定义。很多书本使用一个概括的或非常宽泛的概念来对 AI 这个术语进行解释，这样不仅会使读者对解答感到不满意，而且还可能加深对 AI 的“神秘”感，而 AI 在一般大众和行业人士的常识中是一个非常普遍的词汇。这里，将游戏 AI 问题划分成确实符合真实 AI 解决方案的若干部分来考虑。

(2) 对 AI 元素和解决方案按游戏类型进行分解。很多书本仅仅依赖于一种类型的游戏，或一个有限的示例程序。本书将大多数现代游戏类型进行分解，并以实际发布的名称给出 AI 使用技巧的具体示例。通过了解不同游戏类型的 AI 范例选择背后的基本思想，读者将会对范例本身有更多理解。

(3) 对大多数通用的 AI 范例进行代码实现。在本书的后几章中，对于每种 AI 技术都会给出真实代码，这包括以框架的形式和作为真实示例应用的一部分。本书将会对代码进行分解，并对其进行充分的讨论，以帮助说明系统的实际处理过程。

(4) 对未来改进方向进行讨论。对于每种游戏类型和 AI 技术，本书都会给出系统如何扩展的示例。这将通过两种方式来完成：指出当前游戏和经典游戏中的一般的 AI 缺陷；详细给出在空间、速度或其他限制条件下实现系统最优的方式。

本书总揽

本书第 I 部分是对游戏 AI 的一个概述，它涵盖了贯穿于本书的基本术语，对游戏 AI 中的几个基本概念进行了探讨，并详细分析了游戏 AI 引擎的基本系统。第 II 部分对具体的游戏类型以及它们如何使用不同的 AI 范例进行了讨论。尽管本书并不是包罗万象的(即详细论述每种游戏是如何实现的)，但它将对各种类型游戏提出的问题的更一般解决方案进行讨论。第III部分给出了基本 AI 技术的真实代码实现。而第IV部分则对高级的 AI 技术进行了讨论。第 V 部分对许多概念和关注点进行了分解，着重处理“真正游戏 AI 开发”的各种事项，包括一般的设计与开发问题、作为一个综合范例的“分层式设计 AI”(有助于几乎所有 AI 引擎的组织性)、对 AI 系统的调试以及 AI 的未来。

本书读者

本书旨在向游戏开发人员提供设计现代游戏 AI 引擎所必需的工具，并对当前一些 AI 引擎上应用的技术进行分析。AI 编程是游戏开发过程中一个非常具有挑战性的方向，尽管已经有很多书对一般的游戏相关的数据结构和编码风格进行了阐述，但很少对“游戏 AI”这个重要而且技术性又强的学科领域进行专门论述。

本书是为职业的游戏 AI 程序员，以及有志于将其兴趣领域扩展到 AI 的程序员专门编写的。如果在决定使用哪种技术上存在困难，或者对一个特定游戏最适合的引擎的工作代码存在问题，或者需要这样的运行代码，那么可以从本书中获取答案。本书将为多种有用的游戏 AI 技术提供一个简洁而可用的接口。本书将强调主要的决策范例，因此不会对路径搜索(至少不会直接深入研究，事实上很多本书提到的技术都可用于运行一个路径搜索器)或感知等重要的领域进行深入研究——尽管也会对它们进行讨论。

本书假定读者具有 C++语言、经典数据结构以及面向对象编程基本知识的应用经验。本书的示例程序是在 Windows®平台下用 Microsoft Visual C++®编写的，但只有渲染是与具体平台相关的，并且所使用的渲染 API 是扩展到 OpenGL 中的 GLUT。因此，如果需要，可以很容易地移植到另外的系统。可参见光盘中关于 GLUT 和 OpenGL 的信息。

读过本书之后，读者将对作为一个游戏 AI 程序员必备的知识结构有个充分的了解。游戏类型的介绍将使得程序员能够了解在给定产品和时间表的情况下，如何从头至尾地构造一个 AI 系统。本书的代码涵盖了非常宽广的游戏类型，几乎可以构造任意类型的 AI 系统，并将详细说明如何将这些技术综合应用到更复杂和可用的具体游戏 AI 引擎中。

目　录

第 I 部分　概述

第 1 章　基本定义与概念 …… 3
1.1　什么是智能 …… 3
1.2　什么是游戏 AI …… 3
1.3　什么不是游戏 AI …… 6
1.4　该定义与人工智能理论定义的区别 …… 8
1.5　可应用的大脑科学与心理学理论 …… 9
1.5.1　大脑的组织结构 …… 9
1.5.2　知识库与学习 …… 10
1.5.3　认知 …… 12
1.5.4　心智理论 …… 13
1.5.5　有限最优 …… 19
1.5.6　来自机器人技术的启发 …… 20
1.6　小结 …… 22

第 2 章　AI 引擎的基本组成与设计 …… 23
2.1　决策与推理 …… 23
2.1.1　解决方案的类型 …… 24
2.1.2　智能体的反应能力 …… 24
2.1.3　系统的真实性 …… 24
2.1.4　游戏类型 …… 25
2.1.5　游戏内容 …… 25
2.1.6　游戏平台 …… 25
2.1.7　开发限制 …… 27
2.1.8　娱乐限制 …… 28
2.2　输入处理机与感知 …… 29
2.2.1　感知类型 …… 29
2.2.2　更新规则 …… 29
2.2.3　反应时间 …… 30
2.2.4　门限 …… 30
2.2.5　负荷平衡 …… 30
2.2.6　计算代价与预处理 …… 30
2.3　导航 …… 31
2.3.1　基于网格 …… 31
2.3.2　简单避免与位势场 …… 32
2.3.3　地图节点网络 …… 33
2.3.4　导航网格 …… 33
2.3.5　组合系统 …… 34
2.4　综合考虑 …… 35
2.5　小结 …… 36

第 3 章　Alsteroids: AI 试验平台 …… 37
3.1　GameObj 类 …… 38
3.2　GameObj 类的 Update()函数 …… 39
3.3　Ship 对象 …… 40
3.4　其他游戏对象 …… 41
3.5　GameSession 类 …… 42
3.5.1　主逻辑与碰撞检测 …… 43
3.5.2　对象清除 …… 45
3.5.3　主飞船和宝物的产生 …… 45
3.5.4　奖励生命 …… 46
3.5.5　级别和游戏的结束 …… 46
3.6　Control 类 …… 47
3.7　AI 系统钩子 …… 47
3.8　游戏主循环 …… 48
3.9　小结 …… 48

第 II 部分　游戏类型

第 4 章　角色扮演类游戏 …… 53
4.1　通用 AI 元素 …… 57
4.1.1　敌人 …… 57
4.1.2　头目 …… 58
4.1.3　非玩家角色 …… 58
4.1.4　店员 …… 59
4.1.5　队员 …… 59
4.2　有用的 AI 技术 …… 60
4.2.1　脚本 …… 60
4.2.2　有限状态机 …… 61
4.2.3　消息 …… 62
4.3　示例 …… 62
4.4　例外 …… 63
4.5　需要改进的具体游戏元素 …… 64
4.5.1　角色扮演不等于战斗 …… 64
4.5.2　语法机器 …… 64
4.5.3　任务发生器 …… 65
4.5.4　更好的队员 AI …… 65
4.5.5　更好的敌人 …… 66
4.5.6　完全真实的市镇 …… 67
4.6　小结 …… 68

第 5 章　冒险类游戏 …… 69
5.1　通用 AI 元素 …… 70
5.1.1　敌人 AI …… 70
5.1.2　非玩家角色 …… 70
5.1.3　协作元素 …… 71
5.1.4　感知系统 …… 71
5.1.5　摄像机 …… 71
5.2　有用的 AI 技术 …… 71
5.2.1　有限状态机 …… 71
5.2.2　脚本系统 …… 72
5.2.3　消息系统 …… 72
5.2.4　模糊逻辑 …… 72
5.3　示例 …… 73
5.4　需要改进的领域 …… 74
5.4.1　潜行目标的附加类型 …… 74
5.4.2　传统冒险根源的回归 …… 74
5.4.3　更好的 NPC 通信 …… 74
5.4.4　用户界面 …… 74
5.5　小结 …… 75

第 6 章　即时策略游戏 …… 77
6.1　通用 AI 元素 …… 77
6.1.1　个体单元 …… 77
6.1.2　雇佣个体单元 …… 78
6.1.3　指挥官与中级战略性元素 …… 78
6.1.4　高层战略性 AI …… 78
6.1.5　市镇构建 …… 79
6.1.6　本土生活 …… 79
6.1.7　路径搜索 …… 79
6.1.8　战术与战略支撑系统 …… 79
6.2　有用的 AI 技术 …… 81
6.2.1　消息 …… 81
6.2.2　有限状态机 …… 81
6.2.3　模糊状态机 …… 82
6.2.4　层次化 AI …… 82
6.2.5　规划 …… 82
6.2.6　脚本 …… 82
6.2.7　数据驱动 AI …… 83
6.3　示例 …… 84
6.4　需要改进的领域 …… 85
6.4.1　学习 …… 85
6.4.2　确定 AI 元素何时受困 …… 85
6.4.3　AI 助手 …… 86
6.4.4　对抗人物 …… 86
6.4.5　多战略少战术 …… 87
6.5　小结 …… 88

第 7 章　第一人称/第三人称射击游戏 … 89
7.1　通用 AI 元素 …… 91
7.1.1　敌人 …… 91
7.1.2　敌人头目 …… 92
7.1.3　死亡竞赛对手 …… 92
7.1.4　武器 …… 92
7.1.5　协作智能体 …… 93
7.1.6　分队成员 …… 93

7.1.7 路径搜索……93
7.1.8 空间推理……94
7.2 有用的 AI 技术……94
7.2.1 有限状态机……94
7.2.2 模糊状态机……97
7.2.3 消息系统……98
7.2.4 脚本系统……98
7.3 示例……98
7.4 需要改进的领域……99
7.4.1 学习与对手建模……99
7.4.2 个性……100
7.4.3 创造力……100
7.4.4 预测……100
7.4.5 更好的会话引擎……101
7.4.6 动机……101
7.4.7 更好的分队 AI……101
7.5 小结……101

第 8 章 平台游戏……103
8.1 通用 AI 元素……108
8.1.1 敌人……108
8.1.2 敌人头目……108
8.1.3 协作元素……108
8.1.4 摄像机……109
8.2 有用的 AI 技术……110
8.2.1 有限状态机……110
8.2.2 消息系统……110
8.2.3 脚本系统……110
8.2.4 数据驱动系统……110
8.3 示例……110
8.4 需要改进的领域……111
8.4.1 摄影技巧……111
8.4.2 帮助系统……111
8.5 小结……112

第 9 章 射击类游戏……113
9.1 通用 AI 元素……118
9.1.1 敌人……118
9.1.2 敌人头目……118
9.1.3 协作元素……119
9.2 有用的 AI 技术……119
9.2.1 有限状态机……119
9.2.2 脚本系统……119
9.2.3 数据驱动系统……119
9.3 例外……120
9.4 示例……120
9.5 需要改进的领域……121
9.6 小结……122

第 10 章 运动类游戏……123
10.1 通用 AI 元素……124
10.1.1 教练或团队级别 AI……124
10.1.2 玩家级别 AI……124
10.1.3 路径搜索……125
10.1.4 摄像机……125
10.1.5 混杂元素……126
10.2 有用的 AI 技术……127
10.2.1 有限状态机与模糊状态机……127
10.2.2 数据驱动系统……133
10.2.3 消息系统……134
10.3 示例……134
10.4 需要改进的领域……135
10.4.1 学习……135
10.4.2 游戏平衡……135
10.4.3 玩法创新……135
10.5 小结……136

第 11 章 赛车游戏……139
11.1 通用 AI 元素……140
11.1.1 轨迹 AI……140
11.1.2 交通……141
11.1.3 行人……142
11.1.4 敌人与战斗……142
11.1.5 非玩家角色……142
11.1.6 其他竞争行为……142
11.2 有用的 AI 技术……143
11.2.1 有限状态机……143
11.2.2 脚本系统……143
11.2.3 消息系统……143

11.2.4 遗传算法……143
11.3 示例……143
11.4 需要改进的领域……144
11.4.1 除犯罪以外的其他感兴趣领域……144
11.4.2 更多的智能 AI 敌人……144
11.4.3 永不落幕的游戏世界……144
11.5 小结……145

第 12 章 经典策略游戏……147
12.1 通用 AI 元素……156
12.1.1 对手 AI……156
12.1.2 AI 助手……157
12.2 有用的 AI 技术……157
12.2.1 有限状态机……157
12.2.2 Alpha-Beta 搜索……157
12.2.3 神经网络……158
12.2.4 遗传算法……158
12.3 例外……158
12.4 示例……158
12.5 需要改进的领域……159
12.5.1 创造力……159
12.5.2 速度……159
12.6 小结……159

第 13 章 格斗类游戏……161
13.1 通用 AI 元素……162
13.1.1 敌人……162
13.1.2 碰撞系统……163
13.1.3 敌人头目……163
13.1.4 摄像机……163
13.1.5 动作和冒险元素……164
13.2 有用的 AI 技术……164
13.2.1 有限状态机……164
13.2.2 数据驱动系统……164
13.2.3 脚本系统……164
13.3 示例……165
13.4 需要改进的领域……165
13.5 小结……166

第 14 章 著名的混杂游戏类型……167
14.1 文明游戏……167
14.2 天神游戏……174
14.3 战争游戏……177
14.4 飞行模拟游戏……181
14.5 音乐游戏……186
14.6 益智游戏……186
14.7 人工生命游戏……187

第III部分 基本的 AI 引擎技术

第 15 章 有限状态机……193
15.1 FSM 概述……193
15.2 FSM 骨架代码……197
15.2.1 FSMState 类……197
15.2.2 FSMMachine 类……198
15.2.3 FSMAIControl 类……199
15.3 在试验平台上实现 FSM 控制的飞船……200
15.4 示例实现……201
15.4.1 Control 类编码……202
15.4.2 状态编码……204
15.5 使用该系统的 AI 的性能……213
15.5.1 基于 FSM 系统的优势……213
15.5.2 基于 FSM 系统的劣势……214
15.6 范例扩展……215
15.6.1 层次化 FSM……215
15.6.2 基于消息和事件的 FSM……216
15.6.3 具有模糊转换的 FSM……216
15.6.4 基于堆栈的 FSM……216
15.6.5 多重并发 FSM……217
15.6.6 数据驱动 FSM……217
15.6.7 惯性 FSM……218
15.7 最优化……219
15.7.1 FSM 和感知的负荷平衡……219
15.7.2 LOD AI 系统……219
15.7.3 共享数据结构……220
15.8 设计上考虑的因素……220
15.8.1 解决方案的类型……221

15.8.2 智能体的反应能力……221
15.8.3 系统的真实性……221
15.8.4 游戏类型……221
15.8.5 游戏内容……222
15.8.6 游戏平台……222
15.8.7 开发限制……222
15.8.8 娱乐限制……222
15.9 小结……223

第 16 章 模糊状态机……225
16.1 FuSM 概述……225
16.2 FuSM 骨架代码……228
16.2.1 FuSMState 类……228
16.2.2 FuSMMachine 类……230
16.2.3 FuSMAIControl 类……231
16.3 在试验平台上实现 FuSM 控制的飞船……232
16.4 示例实现……232
16.4.1 添加 Saucer……232
16.4.2 其他的游戏修改……233
16.4.3 FuSM 系统……233
16.5 控制类编码……234
16.6 使用该系统的 AI 的性能……241
16.6.1 基于 FuSM 系统的优势……241
16.6.2 基于 FuSM 系统的劣势……242
16.7 范例扩展……243
16.7.1 有限数量当前状态的 FuSM……243
16.7.2 作为角色支持系统的 FuSM……243
16.7.3 在较大 FSM 中作为单一状态的 FuSM……244
16.7.4 层次化 FuSM……244
16.7.5 数据驱动 FuSM……244
16.8 最优化……245
16.9 设计上考虑的因素……245
16.9.1 解决方案的类型……245
16.9.2 智能体的反应能力……245
16.9.3 系统的真实性……245
16.9.4 游戏类型……246
16.9.5 游戏平台……246
16.9.6 开发限制……246
16.9.7 娱乐限制……246
16.10 小结……246

第 17 章 基于消息的系统……249
17.1 消息概述……249
17.2 消息的骨架代码……250
17.2.1 Message 对象……250
17.2.2 MessagePump 类……251
17.3 客户端句柄……255
17.4 在 AIsteroids 试验平台上的示例实现……256
17.4.1 MessState 类……256
17.4.2 MessMachine 类……257
17.4.3 MessAIControl 类……258
17.5 状态编码……262
17.6 使用该系统的 AI 的性能……265
17.6.1 消息系统的优势……265
17.6.2 消息系统的劣势……266
17.7 范例扩展……266
17.7.1 消息优先级……266
17.7.2 消息仲裁……267
17.7.3 自动和扩展的消息类型……267
17.8 最优化……268
17.9 设计上考虑的因素……268
17.9.1 解决方案的类型……268
17.9.2 智能体的反应能力……268
17.9.3 系统的真实性……268
17.9.4 游戏类型和平台……268
17.9.5 开发限制……269
17.9.6 娱乐限制……269
17.10 小结……269

第 18 章 脚本系统……271
18.1 脚本概述……271
18.2 AIsteroids 测试平台中的脚本实现……272
18.2.1 一种配置脚本语言……273

18.2.2 配置脚本系统的 AI 性能分析……278
18.2.3 游戏中 Lua 的嵌入……278
18.3 Lua 在 AIsteroids 测试平台中的实现……286
18.4 Lua 脚本系统的 AI 性能分析……290
18.5 脚本系统的优点……290
18.5.1 快速原型开发……290
18.5.2 更低的门槛……291
18.5.3 更快的 AI 调试速度……291
18.5.4 更多的用户扩展手段……291
18.5.5 更广的适用范围……291
18.6 脚本系统的缺点……291
18.6.1 执行速度……292
18.6.2 调试难度……292
18.6.3 脚本作用……292
18.6.4 宿主代码和脚本的功能划分……293
18.6.5 需维护的系统数量……293
18.7 范例扩展……294
18.7.1 自定义语言……294
18.7.2 内建调试工具……294
18.7.3 智能脚本 IDE……294
18.7.4 游戏脚本自动集成……295
18.7.5 自主修改脚本……295
18.8 优化……295
18.9 设计上考虑的因素……296
18.9.1 解决方案的类型……296
18.9.2 智能体的反应能力……296
18.9.3 系统的真实性……296
18.9.4 游戏类型和平台……297
18.9.5 开发限制……297
18.9.6 娱乐限制……297
18.10 小结……297
第 19 章 基于位置的信息系统……299
19.1 基于位置的信息系统概述……299
19.1.1 影响图技术(IM)……299
19.1.2 智能地形技术(Smart Terrain)……300
19.1.3 地形分析技术(Terrain Analysis，TA)……301
19.2 各种技术的使用方法……301
19.2.1 占用数据……301
19.2.2 场地控制……301
19.2.3 探路系统的辅助数据……302
19.2.4 危险预警……302
19.2.5 初步战场计划……303
19.2.6 简单战场分析……303
19.2.7 高级战场分析……303
19.3 影响图框架代码及测试平台实现……305
19.3.1 占用影响图……310
19.3.2 占用 IM 测试平台的使用……314
19.3.3 控制影响图……315
19.3.4 控制 IM 测试平台的使用……318
19.3.5 逐位影响图……319
19.3.6 逐位 IM 测试平台的使用……324
19.3.7 其他实现……324
19.4 基于位置的信息系统的优点……326
19.5 基于位置的信息系统的缺点……326
19.6 范例扩展……326
19.7 优化……326
19.8 设计上考虑的因素……327
19.8.1 解决方案的类型……327
19.8.2 智能体的反应能力……327
19.8.3 系统的真实性……327
19.8.4 游戏类型和平台……328
19.8.5 开发限制……328
19.8.6 娱乐限制……328
19.9 小结……328

第Ⅳ部分　高级 AI 引擎技术

第 20 章　遗传算法 …… 331

20.1　遗传算法概述 …… 331

20.1.1　自然进化规律 …… 331

20.1.2　游戏中的进化 …… 332

20.1.3　遗传算法基本过程 …… 333

20.2　问题的表示 …… 334

20.2.1　基因和基因组 …… 335

20.2.2　适应度函数 …… 336

20.2.3　繁殖 …… 337

20.3　AIsteroids 测试平台中遗传算法的实现 …… 342

20.4　遗传算法在测试平台中的性能 …… 354

20.5　基于遗传算法的系统的优点 …… 356

20.6　基于遗传算法的系统的缺点 …… 357

20.6.1　时间代价较大 …… 357

20.6.2　算法性能随机性大 …… 357

20.6.3　结果成败定义模糊 …… 358

20.6.4　最优解不能保证 …… 358

20.6.5　参数调试和扩展难度大 …… 358

20.7　范例扩展 …… 358

20.7.1　蚁群算法 …… 358

20.7.2　协同进化 …… 359

20.7.3　自适应遗传算法 …… 359

20.7.4　遗传程序设计 …… 359

20.8　设计上考虑的因素 …… 360

20.8.1　解决方案的类型 …… 360

20.8.2　智能体的反应能力 …… 360

20.8.3　系统的真实性 …… 360

20.8.4　游戏类型 …… 360

20.8.5　平台 …… 361

20.8.6　开发限制 …… 361

20.8.7　娱乐限制 …… 361

20.9　小结 …… 361

第 21 章　神经网络 …… 363

21.1　自然中的神经网络 …… 363

21.2　人工神经网络概述 …… 364

21.3　神经网络的使用 …… 366

21.3.1　结构 …… 366

21.3.2　学习机制 …… 367

21.3.3　创建训练数据 …… 368

21.4　神经网络活动 …… 368

21.5　在 AIsteroids 测试平台上实现神经网络 …… 371

21.5.1　NeuralNet 类 …… 371

21.5.2　NLayer 类 …… 375

21.5.3　NNAIControl 类 …… 379

21.6　测试平台的性能 …… 384

21.7　优化 …… 385

21.8　基于神经网络的系统的优点 …… 386

21.9　基于神经网络的系统的缺点 …… 386

21.10　范例扩展 …… 387

21.10.1　其他类型的神经网络 …… 388

21.10.2　神经网络学习的其他类型 …… 388

21.11　设计上考虑的因素 …… 389

21.11.1　解决方案的类型 …… 389

21.11.2　智能体的反应能力 …… 389

21.11.3　系统的真实性 …… 389

21.11.4　游戏类型和平台 …… 390

21.11.5　开发限制 …… 390

21.11.6　娱乐限制 …… 390

21.12　小结 …… 390

第 22 章　其他技术备忘录 …… 393

22.1　人工生命 …… 393

22.1.1　人工生命在游戏中的用途 …… 394

22.1.2　人工生命学科 …… 394

22.1.3　优点 …… 395

22.1.4　缺点 …… 396

22.1.5 游戏设计可以开发的领域……396
22.2 规划算法……396
22.2.1 当前在游戏中的使用状况……397
22.2.2 优点……398
22.2.3 缺点……399
22.2.4 游戏设计可以开发的领域……399
22.3 产生式系统……400
22.3.1 优点……401
22.3.2 缺点……401
22.3.3 游戏设计可以开发的领域……401
22.4 决策树……402
22.4.1 优点……403
22.4.2 缺点……403
22.4.3 游戏设计可以开发的领域……404
22.5 模糊逻辑……404
22.5.1 优点……406
22.5.2 缺点……406
22.5.3 游戏设计可以开发的领域……406
22.6 小结……406

第 V 部分 AI 实战游戏开发

第 23 章 分层式 AI 设计……411
23.1 基本回顾……411
23.2 分层式层结构……412
23.2.1 重现前述示例……413
23.2.2 感知和事件层……414
23.2.3 行为层……414
23.2.4 动画层……415
23.2.5 运动层……417
23.2.6 短期决策层(ST)……418
23.2.7 长期决策层(LT)……418
23.2.8 基于位置的信息层……418
23.3 BROOKS 包容式体系结构……419
23.4 游戏层次分解……419
23.4.1 目标……419
23.4.2 分层式超级玛莉……420
23.4.3 AI 怪物的实现……420
23.4.4 AI 玩家的实现……424
23.5 小结……427

第 24 章 AI 开发中普遍关心的问题……429
24.1 有关设计的问题……429
24.1.1 数据驱动系统设计时需考虑的问题……429
24.1.2 “一根筋”(OTM)综合症……431
24.1.3 多细节层次(LOD)AI……432
24.1.4 支持 AI……434
24.1.5 通用 AI 设计思想……435
24.2 有关娱乐的问题……436
24.2.1 所有重要的趣味性因素……436
24.2.2 随机感……437
24.2.3 一些令 AI 系统看上去非常愚蠢的因素……438
24.3 有关产品的问题……439
24.3.1 保持 AI 行为的一致性……439
24.3.2 提前思考游戏参数的调试问题……440
24.3.3 预防 AI 系统的未知行为……440
24.3.4 注意设计人员工具使用方式的差异性……441
24.4 小结……441

第 25 章 调试……443
25.1 AI 系统的通用调试……443
25.2 可视化调试……443
25.2.1 提供各种信息……443
25.2.2 有助于调试……444
25.2.3 时序信息……444
25.2.4 监视状态转变……444
25.2.5 有助于控制台调试……444

25.2.6 调试脚本语言 ……444
25.2.7 双功能影响图 ……444
25.3 Widget …… 445
25.3.1 实现 ……445
25.3.2 BasicButton ……448
25.3.3 Watcher ……449
25.3.4 RadioButton ……449
25.3.5 OnOffButton ……450
25.3.6 ScrubberWidget ……451
25.3.7 程序集成 ……452
25.4 小结 …… 456
第 26 章 总结与展望 …… 457
26.1 AI 引擎设计总结 …… 457
26.2 AI 游戏的未来展望 …… 457
附录 有关 CD-ROM 的说明 …… 459

第Ⅰ部分 概　　述

第 1 章将对“游戏人工智能(Game Artificial Intelligence，游戏 AI)”进行定义，并用一个合适的术语来代替这个含糊的表述。也将从心理学界和人工智能理论界的角度出发讨论一些适当可用的 AI 理论。

第 2 章将介绍构成一般的游戏 AI 引擎的基本系统，以及设计一个新的 AI 引擎时需要考虑的各个要素。

第 3 章将引用并讨论一个基本的应用示例。该示例在本书后面各部分中将作为一个 AI 实现的试验平台。

1 基本定义与概念

欢迎阅读《AI 游戏引擎程序设计》这本书。本书旨在向游戏 AI 的程序员提供开发面向现代商业游戏的 AI 引擎所需要的知识和工具。但究竟什么是“游戏 AI”？AI 是一门相对年轻的科学，其早期的一些工作完成于 19 世纪 50 年代初期。受早期游戏机计算能力和存储空间的限制，游戏采用真正 AI 技术的历史则更短。由于 AI 在游戏中是一个崭新的概念，所以关于游戏 AI 的定义对于大多数人，甚至对那些从事游戏开发的人来说也都还不大清楚。本章将对“游戏 AI”这个术语进行定义，区分游戏 AI 中常常混用的实践和技术问题，并对其未来可能的扩展领域进行讨论。在本章的后面，还会围绕游戏 AI 对认知科学、心理学和机器人技术等其他领域的相关概念进行讨论。

1.1 什么是智能

“智能”这个术语是相当模糊的。从字典里可以查知，它是指“获取和应用知识的能力”，但这个解释太笼统。从字面上解释，该定义可能意味着自动调温器是智能的。因为它可以获取房间太冷这个知识，从而应用它所学到的知识去调节加热器。字典接着表明，智能体现了“思考和推理的能力”。尽管这个定义稍微好些(有更多的限制，从而把自动调温器排除在外)，但它也仅仅是通过引入两个甚至更不清晰的术语“思考”和“推理”来增加我们定义的困难。事实上，对智能的“真正”定义是一个古老而又折磨人的争论，它远远超出了本文的范围。值得庆幸的是，设计一个好的游戏并不需要这个定义。实际上，本文赞同字典里的第一个定义，因为它与我们期望在开发的游戏系统中看到的被认为是智能的行为非常吻合。出于我们自己的目的，一个智能的游戏智能体(agent)是能够获取关于这个世界的知识，并对该知识做出反应的智能体。这种反应的质量和效果就是游戏需要权衡和设计的问题。

1.2 什么是游戏 AI

让我们先对 AI 作一个理论上的定义。在 AI 的经典著作 *Artificial Intelligence: A Modern Approach*(人工智能：一种现代方法)中，Russel 和 Norvig 指出，AI 就是设计计算机程序，

使得它可以像人一样行动和思考，同时也是理性地行动和思考。这个定义包含了智能的认知学和行为学观点，并涵盖了理性和“人性”(因为人有时候是很不理性的，比如冲进一座着火的建筑物去救自己的孩子，但这依然是智能的)。

游戏 AI 是游戏中的代码，它使得在给定情形下，当游戏有多种选择的时候，计算机控制的竞争对手(或合作伙伴)采取看起来是聪明的决策，从而产生相关的有效的和有用的行为。注意上述定义中“看起来”这个词。游戏中由 AI 产生的行为是“结果”导向的。因此，可以认为游戏界主要关注 AI 中的行为主义学派。的确，我们仅仅对系统将要做出的响应感兴趣，而并不太关心系统是如何做出这个响应的。我们关心系统如何行动，而不关心系统如何思考。人们在玩游戏时并不会在意这个游戏是否运用了庞大的决策脚本数据库，也不会在意是否对决策树进行直接搜索，或者是否建立了所处环境的精确知识库和是否基于逻辑规则进行推理决策。正如他们所说，推理过程全由游戏 AI 来实现。因此，纯粹的行为决策，如对手发动了哪项攻击，他是如何接近玩家的，他如何在环境中使用各种要素，以及其他的一些细节，全部由游戏 AI 系统来完成。

现代游戏发展过程中也使用“AI”这个术语来描述其他游戏行为，比如人类输入接口的工作方式。有时候，甚至是支配运动和碰撞的算法(如果游戏运动是通过动画驱动，而非物理模拟)也属于这一类。可见，AI 这个术语在游戏开发界是一个被广泛使用的名词。当与其他同行(或者在自己工作的公司内)讨论 AI 时，大家必须一致认同这个术语的内涵和范围。这点非常重要，否则，如果一个人关于 AI 的观念与其他人的观念相差太远，就容易产生误解。当本书涉及到 AI 时，将使用非常狭窄的定义，即“基于角色的行为智能”。其中的术语“角色(character)”源于大多数游戏的角色(或执行者、智能体)驱动本质。确实，很多策略游戏或所谓“天神”游戏都有一个“管理员(overseer)”的概念，在全局意义上看它是在进行决策，但也可看作只是另一个角色而已。

以前，AI 编程更普遍地被称为“游戏玩法编程”，因为 CPU 控制的角色所展现出来的行为的确不含有任何智能成分。图 1-1 所示是游戏 AI 的整个发展年表。在视频游戏早期的大多数程序员都为他们的敌人使用一些模式或重复运动(如 Galaga 或 Donkey Kong)，或者使用一些仅仅能移动而且是只在某些确定的“弱点”(如 R 型)上易受攻击的敌人。在某种程度上，这些游戏都是要找到预先确定的行为模式，以便玩家能够轻易击倒对手(或一群对手)并继续前往另一个地点。这是由早期处理器速度和存储器空间的极端限制所决定的。模式很容易存储，而且只需要最少的代码来驱动，并且几乎不需要进行计算；无论在顶层表现其他什么行为，它都简单地使对手按规定模式运动(例如，当玩家在他们下面并以某种模式移动时，Galaga 中的对手便进行射击)。事实上，一些游戏采用所谓随机的运动，但由于早期游戏中的随机数产生器使用伪随机数的硬编码表，因此这些模式也能够在整个游戏行为中最终预见。

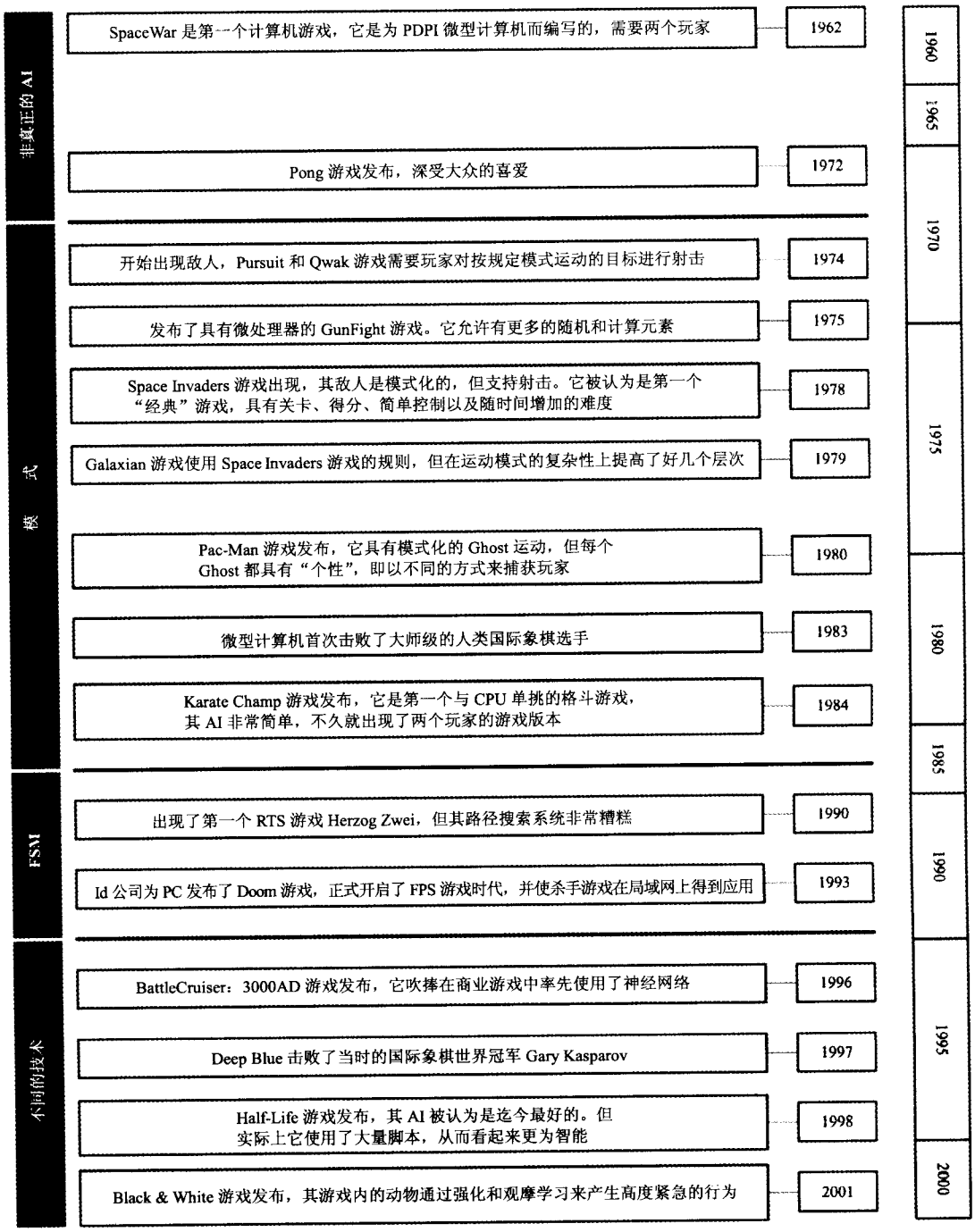


图 1-1　游戏 AI 的发展年表

在过去，为了让游戏看起来更智能而经常使用的另一种技术是让计算机对手作弊，也就是让他拥有人类玩家所不拥有的关于该游戏的额外信息，从而使得下一步所做的决策看起来非常地巧妙。计算机读入玩家所按下的按钮(甚至在玩家开始按钮所对应的动画之前)，

并以一个阻碍的走步作为系统的响应。一个实时策略作弊器可以使其使用者在游戏早期他们还没有找到有价值资源的区域时便前往这些地点。当游戏赋予计算机对手一些能力，给它提供额外的能力、资源，以及一些可直接使用的东西，而不是依靠自己来提前规划和了解对那些资源的需要，那么就可以实现 AI 作弊。这些策略能够带来更具挑战性但最终更不能让人满意的对手，因为人类玩家几乎总能意识到计算机正在完成一些他不可能完成的任务。关于这种不可能行为的一个很容易发现同时又很令人沮丧的例子是赛车游戏中所用到的决胜局欺诈。在赛车接近结束时，如果玩家超出 AI 控制的车辆太多，那么游戏就会简单地加快这些车辆的速度，直到它们追赶上玩家，那时它们又恢复到正常状态。的确，这使赛车更像是一场战斗，但是看着先前无能的赛车突然奇迹般地追上了玩家，这简直就是件十分荒谬的事情。

在现代游戏中，这些旧方法正逐渐地被淘汰。游戏的主要卖点正缓慢但确确实实地向能够展现 AI 成就和能力的领域发展，而不是像最近游戏发展那样注重图形化外表。在 AI 重要性和品质的新扩展中，强调视觉实际上是有原因的。早期对图形的强调最终导致几乎在每个平台上都出现了专门的图形处理器，并且主 CPU 正日益向越来越完善的 AI 例程开放。尽管游戏图形具有如此重要的作用，但游戏图形的吸引力还是注定在不断减弱，并且人们越来越多地把注意力集中到游戏本身的其他内容上。考虑到如今的消费者正推崇拥有更好 AI 控制对手的游戏，因此，我们拥有更多的 CPU 时间是非常有利的。在游戏的 8 位计算机时代或更早的时候，一般只有 1~2%的 CPU 总时间用于运行游戏的 AI 要素。现在，需要 AI 的游戏都预留 10~35%的 CPU 时间给 AI 系统[Woodcock01]，有的甚至更高。

这意味着，现在的游戏对手能够不用作弊就能找到更好的游戏解决方案，而且能够运用一些更具适应性和应急性的方法(如果没有诸如可获得更快和更强大的处理器对它们进行驱动等原因)。因此，现代游戏中定义的 AI 正逐渐地显示其真正的智能(AI 理论界定义的智能)，而不是陈旧的规定性模式或模仿智能行为的行为。传统的游戏 AI 工作正越来越多地注入来自 AI 理论界的技术(如启发搜索、学习、规划等)。随着游戏(以及玩家兴趣)变得越来越复杂，这种趋势将会继续下去。

1.3 什么不是游戏 AI

如上一节所述，游戏 AI 是一个使用非常广泛的术语，当涉及到除图形外的游戏的任何其他元素时都常常会用到它——尽管并不准确，如游戏的避障(或路径搜索)系统、玩家控制、用户界面以及有时的动画系统都与 AI 有关。在某种程度上，这些元素的确与 AI 有关，而且如果使用不当，会造成 AI 看起来“更愚蠢”。但是，它们并不是游戏中最初的 AI 系统。该情况的一个例外或许是一个玩法足够简单的游戏，该游戏中对手的全部智能就在于不停地走动或选择合适的动画进行播放。

本书将强调这个区别：当游戏有多个选项或玩法时，游戏 AI 将做出智能的决策。那些二级系统，尽管从一组解决方案、动画或路径中进行决策，但对于特定输入，更多的是找到最好(唯一)的解决方案。与此相反，主要的 AI 可能会面临很多同样好的解决方案，但还需要对规划、资源、玩家特征等进行考虑，以便从游戏的全局角度进行决策。

对这种区别的另一种思考方式是那些支持系统只具有很低层次的智能，而本书把目光更多地放在 AI 系统需要进行的高层决策之中。例如，一个人从椅子上站起来并走过这个房间。他脑子里的想法是“我想从冰箱里拿一瓶苏打水”。但看看为了完成这项任务他所使用的低层智能：他的大脑决定肌肉收缩的正确顺序以便让自己从椅子里站起来(动画获取)，开始朝冰箱走去，穿过地板上的所有物体(路径搜索)。此外，他可能会失去平衡但很快又重新恢复(物理学)或者在路上搔头弄耳，以及其他许多细微动作。这些情况都不会改变这个事实：他的整个计划是要得到苏打水并最终实现。然而，并不是所有的游戏都采用检查高层 AI 的分层决策系统。大多数游戏都是将不同层级的决策分成相互通信的独立系统。问题是，这些低层系统都支持智能体的智能，但对于本书来说，并不定义 AI 控制的智能体的智能。

另外需要注意的是，设计一个好的游戏 AI 并不一定意味着写出好的代码。很多程序员认为，AI 设计是一个通过纯粹的编程技术能够解决的技术问题，但这远远不够。当设计一个游戏 AI 时，优秀的软件设计者必须要从诸多方面进行权衡，比如玩法、审美学、动画、声音以及 AI 和游戏界面的行为。的确，AI 系统必须要解决大量的高技术含量的挑战。然而，AI 的最终目标是给玩家提供一种娱乐体验，而不是展示巧妙的代码。如果玩家觉得游戏一点也不够智能和有趣，那么他也不会去关心开发者那闪光的新算法。游戏 AI 不是最好的代码，它只是对代码以及大量“奏效的东西(whatever works，WW)”的最好运用。过去一些看起来非常智能的游戏使用一些很有问题的方法来达到他们的目的。本书并不纵容编写不好的代码，但只要是有助于增进智能成分并增强游戏趣味性的东西，就都不应该抛弃掉。另外，业内一些非常有名的游戏代码都以一通无意识的程序处理开头，这揭示了回溯和清除的一种智能算法。

WW 的思想已经走入大众之中，其原因在于在游戏开发过程中一般局面往往很迟才打开，并且这通常被认为是拙劣调度的标志。由于过去 AI 关注的是非常底层的东西，因此，过去绝望的 AI 程序员被迫使用一些有问题的方法来使他们的系统带有智能成分。这种填鸭式方法或许会带来更好看的行为，但它同时也将导致调试、维护和游戏代码扩展上的困难。对我们来说，幸运的是，这类事情已经降低到最小限度，因为在通往成功游戏的路上 AI 正起着越来越重要的作用，并能够在技术设计的扩展和必须要推敲的时间上进行预先计划。如果在尝试使用一种通用的简洁而又极好的 AI 算法时，发现它要么限制了引擎能够完成事情的类型，要么因为需要大量二级资源而给工作人员带来过度的压力，那么需要提出关于效用的问题。游戏 AI 领域跟宇宙中的其他领域一样，并不是对任何事情都有很好的解决方案。

还有一个不是很重要的问题。游戏 AI 并不是一种新的生命形式，不是一个分离的最终将接管人的工作站(PlayStation®)并命令人定期给它喂食的大脑。好莱坞告诉我们这就是 AI 将带给我们的东西，但事实却远没有那么戏剧性。在未来的 AI 研究中，我们很有可能得到能够学习从而胜任任何游戏的真正一般的 AI 范例，但现在我们还无法实现。现在，游戏 AI 还与具体游戏有关，并且在很大程度上由程序员控制。然而，这个领域仍然被不从事编程工作的公众所误解，甚至被那些从事游戏开发但不经常研究 AI 系统的人员所误解。因此，必须要小心，不要陷入管理层或游戏设计者的突发奇想、建议和“游戏想法”的困扰，因为他们并不完全理解 AI 系统在特定游戏上所具有的限制。

1.4 该定义与人工智能理论定义的区别

人工智能理论有两个主要的目标。第一个目标是要帮助我们理解智能实体，然后智能实体将反过来帮助我们理解我们自身。该目标也是一些更深奥领域(如哲学和心理学等)的研究目标，但在这里它更具功能性。相比哲学的“为什么我们是智能的？”和心理学的“智能来自大脑中的哪个部分？”，AI 更关心这个问题：“那个家伙是怎么找到正确答案的？”。第二个目标是构建智能实体。大家可以认为这是为了娱乐或者是利益，因为这些智能实体在我们的日常生活中非常有用。第二个目标反映了现实经济系统的本质(尤其是在所谓的西方世界里)，即最有可能获取最大利益的研究也是最有可能获得资助的研究。

Russel 和 Norvig [Russel95]把 AI 定义成能够仿效 4 种事件的计算机程序设计，即像人一样思考、有理性地思考、像人一样行动、有理性地行动。在理论研究中，上述定义中的 4 个部分都是构建智能程序的基础。图灵测试(Turing test)就是“像人一样行动”的例子，即如果人不能分辨出程序的动作和人的动作之间的差别，那么该程序就是智能的。认知理论有助于将传统的人类心智科学(mind science)与 AI 设计结合在一起，从而产生更多的像人一样的行为，并且我们希望它们能够以人类思考的方式进行“思考”。绝对逻辑系统试图通过纯粹的理性思考来解决问题，而不带个人偏见和情感。但这却成了 AI 理论定义的方法：有理性地行动——总是设法给出问题的答案。实际上，“有理性地行动”是上述定义的 4 个方面中最主要的一个，也是全世界研究者在实验室努力研究的方向。

从这个意义上说，游戏 AI 的主要目标和传统研究的主要目标很不一样。AI 理论通常关注理性的决策，即最优的或最正确的(当不存在最优时)情况。与此相反，游戏 AI 把焦点集中在像人一样行动，而不太依靠总体理性。这是因为游戏 AI 需要模仿人类任务性能的优劣变化，而不是在任何时候都对最优决策进行严格搜索。当然，这里有娱乐的原因。

以国际象棋游戏为例。如果玩这个游戏是作为理论研究的一部分，那么我们希望在给定时间和存储器约束下，它能有最好的表现。我们会想方设法获取完美的理性，运用高度调整的 AI 技术来帮助驾驭大量可能的动作。然而，如果设计国际象棋游戏是为了给一般的人类玩家一个可以对抗的娱乐对手，那么我们的目标将急剧转移。我们希望游戏能够给人提供一个合适的挑战，而不能一直采用最优走步来获取压倒性的胜利。是的，完成这两个程序所运用到的技术或许在很多方面都非常相似，但由于程序的主要目标不同，对这两个系统的编码也会存在很大的差异。对 Big Blue 进行编程的人不会在意 Kasparov 跟 Big Blue 对抗时是否会获得乐趣。但非常流行的 Chessmaster 游戏的设计人员肯定花了大量的时间来考虑娱乐因素，特别是在默认难度设置模式里。

国际象棋只是一个特殊的例子，因为人们在玩国际象棋游戏时都常常希望它能够有很好的表现(除非他们只是在学习，并把游戏的难度等级设置成一个较低的水平)。设想一下 Quake 游戏中 AI 控制的“机器人(bot)”死亡竞赛对手中用到的例子。如果机器人走入房间，就非常完美地躲藏、瞄准，并精确地知道宝物(powerup)将何时在地图中的何地出现，那么与它对抗就没有什么意思。相反，我们希望游戏 AI 对手只具有人类级别的性能。我们希

望与一个更像人的对手对抗，它们具有人类玩家的特点，比如偶尔会出现失误、在对抗中用完了弹药、跳跃失误而坠落等。我们同样需要有能力的对手，但由于我们无法对能力进行测量，因此在确定某些事情的智能性和真实性时，我们期望它们能有一些缺陷和不足。

另一方面，理论 AI 系统一般不对人性进行模仿——尽管有些是这样。它们主要是模仿智能，即在所有给定的可能决策和规则的条件下做出最理性的决策的能力。这通常是它们唯一的需求，也是不对我们所面临的限制(如时间和存储器)进行考虑的原因。同时，通过远离人性问题，它们不会遇到处理诸如智能构成和如何正确解决等恼人的问题。它们只是简单地从大量协议好的可能发生的事情上进行搜索，以得到最大的总体效益。

最后，计算能力、存储器容量以及软件工程具有非常重要的作用，它们可以使得 AI 研究中相互独立的两部分结合在一起。AI 系统可以具有很高的性能，甚至可以实时解决那些最复杂的问题。因此，对 AI 系统进行编程更像是简单地在问题与系统之间建立起联系。

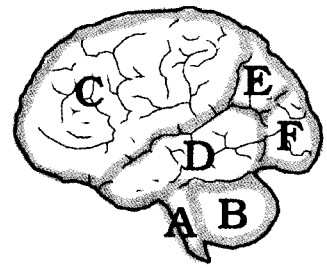
1.5　可应用的大脑科学与心理学理论

对人类大脑的工作方式进行思考可以使 AI 编程具有从现实生活中学到的结构化和程序化特点。有保留地阅读本节内容，并注意关于大脑的工作方式和组织形式存在着不同的理论。本节将介绍更多的关于如何像人的大脑那样分解智能任务的思想和概念。

1.5.1　大脑的组织结构

大脑是由一些子部分构成的，通常(然而却并不正确)可划分为 3 个主要的部分：后脑(或脑干)、中脑和前脑。很多人都听说过大脑按照爬行脑、哺乳动物脑和人脑来进行的划分，但最近的研究表明这种按照物种进行的明确划分是不正确的。几乎所有大脑都具有这 3 个部分，只是大小不一样而已；而且在某些情况下，位置也完全不一样(比如蛇就具有哺乳动物的大脑位置)。这些大脑区域相互独立工作，并通过使用局部内存区域和访问相邻的突触连接来完成有机体的一些具体工作(例如恐惧条件反射就集中在扁桃体上)。但这些区域又是相互连接的，甚至有些区域为了执行一些全局任务，连接还相当紧密(例如，扁桃体是情感数据最基本的首要采集点，之后通过脑丘和一些脑皮层区域，将搜集到的数据传输给海马状突起，与其他感官输入混合后进行最后存储)。然而，这使得大脑在某些方面是目标导向的，具有无限的存取功能并缺少真正的“父类”。

在设计 AI 引擎时，大脑的组织模型具有一定的价值。图 1-2 给出了大脑和游戏系统之间相对应的任务分配。通过将 AI 任务分解成几个具体的子模块(相互之间仅需要了解很少量的知识)，我们可以让其他类来采集这些子模块的输出，并将这些知识混合成游戏角色能够使用的更加复杂的表示。这也体现了我们的 AI 系统所追求的高效率。要避免单独使用计算和代码，以及那些很少以至于在实际应用中通过硬编码实现的输入条件。尽管我们不能完全克服低效率，但大多数的低效率都可以通过巧妙的思考和编程来减少。



	大 脑	游戏 AI 系统
A	脑干 - 反射 - 低层功能/生存	- 碰撞 - 动画选择
B	小脑 - 运动中枢/传感器混合与协调	- 物理学 - 导航
C	额叶 - 高层大脑功能 - 情感 - 学习	- 决策
D	颞叶 - 记忆(视觉和言语)	- 学习
E	顶叶 - 传感器皮层	- 感知
F	枕叶 - 视觉处理	- 感知

图 1-2 与游戏 AI 系统相关的大脑目标导向特性

1.5.2 知识库与学习

尽管人类存储系统的内部工作方式还不完全清楚，甚至还没有形成共识，但人们的普遍观点是：信息是以突触连接层级上大脑神经细胞的微小变化的形式进行存储的(这里对神经系统的描述进行了高度简化)。这些变化在网络中的不同路线上由于电导率的不同而有所差别，从而影响了特定神经细胞以及整个子网络的冲动势(firing potential)。如果使用某条特殊的神经路径，它将会变强，反之亦然。因此，存储系统采用了一种“可塑性(plasticity)”的技术，并且游戏可以从中学到很多东西。与设计一个 AI 行为以及对人类动作反应的固定列表不同，我们可以让行为混合在一起，而通过 AI 可塑性带来的延展性来进行体现。AI 能够跟踪它自己的行动，而不管人类是否始终通过选择一定行为来对它做出反应。它能够识别趋势并调整自身行为(或者如果是作为防御，则采取必要的反措施)，从而灵活地改变 AI 的整体行为。

当然，AI 系统需要一个可靠系统来确定哪些是值得学习的东西，然而人类大脑对所有事情都进行存储，这将导致误解、误传甚至是错觉。尽管语境上非常复杂，但对 AI 的学习进行筛选可以使人类玩家避免通过教系统一些误导的行为(系统也将同样响应这些误导的行为)来开发学习系统。是不是玩家连续三次拳击后 AI 都一直使用一个低位阻挡来阻止下一次即将到来的拳击？如果是这样的话，玩家会对此有所认识，并在三次拳击后使用一个高位拳击，从而击中一直是低位阻挡的 AI。

另一个从自然界中学到的有用经验是，人类大脑中记忆增强和降低的速率并不是对所有的系统都是一样的。例如，与疼痛和厌恶有关的记忆或许永远难以完全消除，尽管这个人只是经历过一次并且以后永不再经历。这是自然界使用动态硬编码的一个很好的例子。上面提到的可塑性变化能够被锁住(通过停止学习过程或者把那些变化转移到一个更长期的存储器中)，从而不允许由于时间的流逝而降低记忆。但像大脑一样，使用过多的硬编码或者使用不当，都将导致非常奇怪的行为，使人(或游戏角色)变得病态或遗忘。

需要考虑的一个更加深入的概念是长期存储器和短期存储器。短期存储器(或内存储器)可被认为是一种仅保持较短时间的感知，之后便根据其重要性进行过滤，然后存入长期存储器或简单地遗忘掉。这便产生了诸如注意广度(attention span)和单一思想(single mindedness)等概念。很多游戏都有数字存储器，当敌人看到玩家(或更坏的情况，即被玩家的武器击中)时，会在规定的时间周期内跟随在玩家后面，但一旦玩家隐藏起来，它便会彻底忘记玩家的存在并回到它原来的工作中去。这是典型的基于状态的 AI 行为，但也是很不符合实际的、缺乏智能的行为。通过为对手设计更多的模拟存储器模型，或者仅仅让它拥有记住长于一分钟时间内的任何事情的能力，它便仍然能够回到它原来的工作，但会对将来的攻击更敏感，而且可以事先寻求支援等。确实，有些游戏采用了这些类型的规则。但对于现实的 AI 行为，总存在着改进和扩展的空间。

大脑也使用“调节器(modulator)”，它们是释放到血液中的一些化学物质，并需经过一定时间才能降解。这些物质包括肾上腺素(adrenaline)或催产素(oxytocin)等。它们的主要作用是抑制或增强特定大脑区域内神经元的冲动(firing)。这将使得大脑接收更加集中，并根据语境对特殊情形下的存储器进行调节。在游戏 AI 系统中，调节器可以重载整个 AI 的状态，或仅在一定状态下所展现出来的行为。因此，传统的基于状态的 AI 可以借用调节(modulation)的概念，从而变得更加灵活。先前提到的被警告过的敌人角色可以转换到一个完全不同的 Alerted 状态，并在经过缓慢退化后又重新转换到 Normal 状态。但采用带修正器的状态系统时，可以用一个“攻击性(aggressive)”调节器来保持其正常的 Guard 状态。尽管保持角色的状态图比较简单，但需要一个更一般的方法来对 Guard 状态进行编码。

人类大脑通过在大脑中不同存储器中存储事情来进行学习，这可以认为是通过偏差和联想来扩展关于世界的知识库。通常有几种独立的实现方式：探测或直接体验、模仿、想象推断。推断或许是个例外，因为它需要有一个高度发展的智力模型，其游戏角色可以通过相同的方式来收集信息。但是，当决定在游戏中采用学习技术时，必须要谨慎地选择游戏的学习方式。对那些与人对抗时似乎奏效的行为进行统计是一种途径；记录人类玩家在跟 AI 对手对抗时的所作所为并对这些人类行为进行模仿或改进则是另一种途径。

采用经典 AI 学习算法(以及大脑物理模型)的游戏会遇到问题，即为了学习它们常常要对其方位进行多次迭代。在 AI 对手的快节奏短周期世界中进行学习将导致其性能急剧下降。使用这些技术的大多数游戏都是在使用前的制造过程中进行全部的学习，而在使用过程中则不具备学习能力，只有这样才能保持其行为的稳定。当有其他满足速度和精度要求的新方法问世并公开之后，这种情况将会有所改变。

但学习不一定都是有意识的。例如，很多种游戏用影响力地图(influence map)来进行无意识的学习，这可以使 AI 对手看起来更智能；且只有一两个应用能够用它进行有意识的学习。对地图上任意地点双方各有多少战斗单元死亡进行一个简单估计，能够给 RTS 游戏的路径搜索算法提供躲避“死亡区域(kill zone)”(对手(人类或其他)在一些经常走到的地图位置上设置的致命陷阱)的所有信息。这种学习将随着时间的推移将逐渐变弱，或受攻击单元的影响(它们传回的消息表明它们率先摧毁了那些将某个区域变成死亡区域的所有事物)。影响力地图在一些运动类游戏中也得到了成功应用。例如，通过微调玩家在足球场上的默认位置，可以更好地对过去人类玩家的传球进行定位；对防守队采用相同的系统，可以使得他们尽可能地阻碍对方传球。

该类型的系统允许对相同类型的信息进行累积，并以一种快速且易访问的方式对它们进行简单存储，同时保持迭代次数在很低水平(为了检查该类型系统学习的效果，迭代是必不可少的)。由于被存储信息的特性非常具体，因此存储错误信息的概率也减小到了最低程度。

1.5.3 认知

我们的感官时刻在接收大量的数据。大脑又是如何知道哪些信息要首先进行处理？哪些信息要丢掉？在更危险的形势下，何时将正在处理的任务挂起？大脑是通过使用各种不同的系统来对输入数据进行快速分类并设置优先级，从而实现上述功能的。认知可以认为是获取所有输入传感数据(称为“感知(perception)”)，并通过先天知识(包括本能和直觉)和推理中心(包括存储的记忆)，来获得用户意义下对这些感知的理解。逻辑、推理、文化以及所有个人存储的规则都仅仅是把所需要了解的感知从背景噪声中挑选出来的手段而已。想一想生活在大城市里的人的大脑每天要接收的绝对信息量。他必须要处理数以百万的人和车给他带来的视觉、声觉和嗅觉上的冲击，要从人群中、叫卖小贩中、无家可归的流浪汉中以及其他无数分心的事物中不停地进行路径搜索。如果他的大脑把所有这些都记住，那么它将无法充分集中注意力来完成任何工作。

感知不一定都来自外部。现代世界的压力导致了紧张和忧虑，从而分散了人们的注意力和打断了人们的思考。每个人都会经常遇到一些转瞬即逝的突发奇想，大脑需要对自己的那些突发奇想进行提炼，从而得到一些重要的思想。

在游戏 AI 中，我们不会因为海量的数据而烦恼，因为我们能够在处理过程中挑选出任意层级的感知。这使得整个程序看起来没那么神秘。图 1-3 所示是一个运动类游戏的实体模型(mock-up)，它对前景中 AI 玩家做出的不同决策使用了不同的感知。当对任意特殊的 AI 子系统进行编码时，应该确保只使用那些真正需要的感知。不能过度简化，否则将很容易预测子系统的输出行为。仅仅当声音位于敌人的一定范围之内时敌人角色才能听到它，这样的听觉子系统有时是非常奇怪的，比如玩家恰好在那范围之外的地方发出一个非常大的噪声。应该要考虑距离和初始音量，这样声音会自然地随着传播而衰减。或许同样得考虑环境的声学特性，因为声音在峡谷中会比在办公大楼中传播得更远，或者在水下比

在空气中传播得要远。这些都是非常简单的例子，但我们可以领会这其中的概念。感知不是单一的，因为对每个感知表示的数据往往可以通过多种方式进行解释。

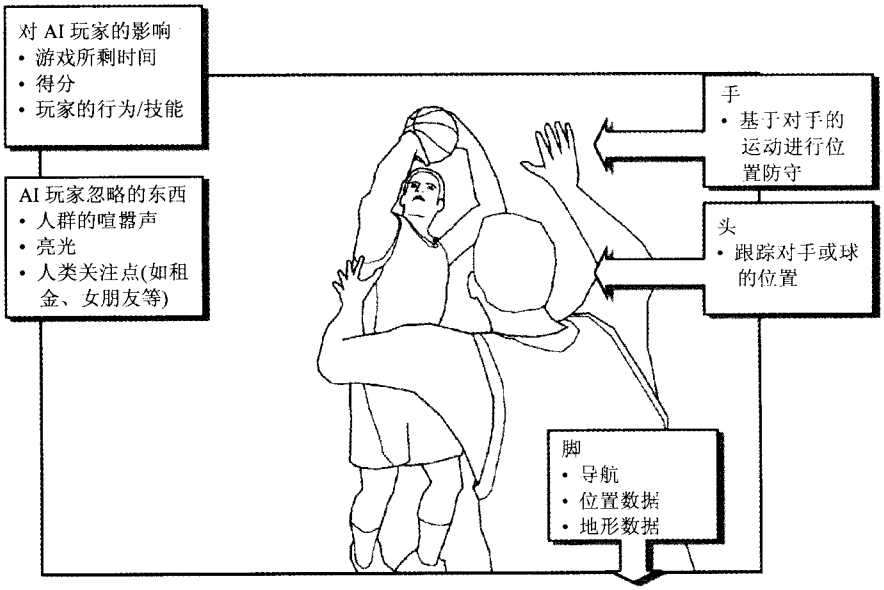


图 1-3　游戏角色需要考虑的各种感知的图形描述

我们可以把 AI 界中用到的系统也当作过滤器。决策系统用来确定所要执行的正确行动，所有作为我们主要决策系统的技术都只是一种过滤方法，即从 AI 能够做的所有可能事情(或由某些规则或游戏状态定义的这些事情的子集)中过滤出当前的游戏状态(依靠 AI 注册的感知变量)。因此，总的来说，许多人关于 AI 的主要观点是：它普遍地由某种方式的聚焦搜索构成。在一定程度上这是对的。大多数 AI 系统都仅仅是从大量可能性中进行搜索，只是方式不同而已。因此，游戏可能发生的事情的特征可用于从概念上考虑使用哪种最好的 AI 技术。这种特征通常被称为游戏的“状态空间(state space)”。如果游戏对不同感知的可能结果的响应大部分都是孤立的，不存在真正的灰色条件，那么可以采用基于状态的系统，因为面对的是许多可能的数字响应，而且几乎是一种枚举状态空间。然而，如果可能发生的响应是全范围连续的，就像存在偶尔下降的起伏的山坡(或另一个更加三维的比喻，读者应该能够理解)，那么基于神经网络的系统将会更合适，因为它们在连续响应域上能够更好地识别局部极值点。我们将在本书的第III部分和第IV部分对它们以及其他 AI 系统进行讨论，这里仅仅是给出一个说明。

1.5.4　心智理论

行为主义学家和认知科学家都采用一个心理学模型作为主要的研究领域，这就是所谓的心智理论(Theory of Mind，ToM)。这个概念在 AI 领域具有很多优点，因为我们的主要工作就是设计看起来智能的系统。实际上，ToM 更多的是认知特性，而不是一种理论。它深刻地指出了人具有理解他人的能力，并具有与其自身分离的思想和世界观。在技术层次

上，ToM 被定义为理解别的东西是有意识的智能体，并通过诸如信念(belief)和欲望(desire)的意识状态的理论概念来解释他们的思想[Premack78]。然而，它并不像听起来那么复杂。可以将它看作是拥有领会意图的能力，而不只是对行动的严格认知。我们一直在做这些事情，并对那些甚至是最不人性的环境因素也进行人性化处理。程序清单 1-1 列出了很早的一个 AI 程序 Eliza 的 Java 版本(由 Robert C. Goerlich 于 1997 年编写)的一段代码。在他那个时代，这是一个非常显著的工作，使人们相信它比实际具有更大的作用。

这对我们有什么启发呢？从人类看来，形成关于他人的 ToM 的能力通常在大约 3 岁的时候开始显现出来。判断一个小孩是否形成了这种认知特性通常采用的一个测试方法是向他询问经典的“错误信念任务(False Belief Task)”[Wimmer83]。在这个问题中，小孩被安排在一个场景中，同时一个名叫 Bobby 的角色把一个私人物品(比如一本书)放到他的壁橱里。之后 Bobby 离开。当他离开时，他弟弟走进来并把那本书拿出来放到食橱里。然后问这个小孩当 Bobby 回来时他会去哪儿找他的书。如果小孩回答是食橱，那么表明他还没有形成这种理解，即在 Bobby 的心智里并不存在与小孩所拥有的一样的信息。因此他还没有一个关于 Bobby 的心智的抽象的参考系，或者说理论，也因此没有关于 Bobby 的 ToM。如果小孩给出了正确的答案，那表明他不仅能够确定关于这个世界的事实，而且能够形成其他人的心智的理论和简化模型(该模型包括了他们可能拥有的事实、欲望和信念)，因此也就提供了他人心智的理论。

程序清单 1-1 Java 版 Eliza 程序中的一些示例代码

```
public class Eliza extends Applet

    {

    ElizaChat cq[];
    ElizaRespLdr ChatLdr;
    static ElizaConjugate ChatConj;
    boolean _started=false;
    Font _font;
    String _s;

    public void init()
      {

      super.init();
      ChatLdr = new ElizaRespLdr();
      ChatConj = new ElizaConjugate();

      //{{INIT_CONTROLS
      setLayout(null);
      addNotify();
      resize(425,313);
      setBackground(new Color(16776960));
      list1 = new java.awt.List(0,false);
```

```
        list1.addItem("Hi! I'm Eliza. Let's talk.");
        add(list1);
        list1.reshape(12,12,395,193);
        list1.setFont(new Font("TimesRoman", Font.BOLD, 14));
        list1.setBackground(new Color(16777215));
        button1 = new java.awt.Button
             ("Depress the Button or depress <Enter> to send to Eliza");
        button1.reshape(48,264,324,26);
        button1.setFont(new Font("Helvetica", Font.PLAIN, 12));
        button1.setForeground(new Color(0));
        add(button1);
        textField1 = new java.awt.TextField();
        textField1.reshape(36,228,348,24);
        textField1.setFont(new Font("TimesRoman", Font.BOLD, 14));
        textField1.setBackground(new Color(16777215));
        add(textField1);
        //}}
        textField1.requestFocus();
}

public boolean action(Event event, Object arg)
        {
        if (event.id == Event.ACTION_EVENT && event.target == button1)
             {
                clickedButton1();
                textField1.requestFocus();
                return true;
             }
        if (event.id == Event.ACTION_EVENT && event.target == textField1)
             {
                clickedButton1();
                textField1.requestFocus();
                return true;
             }
        return super.handleEvent(event);
}

public void clickedButton1()
        {
        parseWords(textField1.getText());
        textField1.setText("");
        textField1.setEditable(true);
        textField1.requestFocus();

}
public void parseWords(String s_)
        {
        int idx=0, idxSpace=0;
        int _length=0; // actual no of elements in set
        int _maxLength=200; // capacity of set
```

```
        int _w;

        list1.addItem(s_);
        list1.makeVisible(list1.getVisibleIndex()+1);
        s_=s_.toLowerCase()+" ";
        while(s_.indexOf("'")>=0)
            s_=s_.substring(0,s_.indexOf("'"))+
                  s_.substring(s_.indexOf("'")+1,s_.length());
            bigloop: for(_length=0; _length<_maxLength &&
                              idx < s_.length(); _length++)
                  {
                  // find end of the first token
                  idxSpace=s_.indexOf(" ",idx);
                  if(idxSpace == -1) idxSpace=s_.length();

                  String _resp=null;
                  for(int i=0;i<ElizaChat.num_chats && _resp == null;i++)
                        {
                        _resp=ChatLdr.cq[i].converse

                        (s_.substring(idx,s_.length()));
                        if(_resp != null)
                              {
                              list1.addItem(_resp);
                              list1.makeVisible(list1.getVisibleIndex()+1);
                              break bigloop;
                        }
                  }
                  // eat blanks
                  while(s_.length() > ++idxSpace &&
                              Character.isSpace(s_.charAt(idxSpace)));
                  idx=idxSpace;

                  if(idx >= s_.length())
                        {
                        _resp=ChatLdr.cq[ElizaChat.num_chats-1]
                                                .converse("nokeyfound");
                        list1.addItem(_resp);
                        list1.makeVisible(list1.getVisibleIndex()+1);
                  }
            }
        }
        //{{DECLARE_CONTROLS
        java.awt.List list1;
        java.awt.Button button1;
        java.awt.TextField textField1;
        //}}
}
//---------------------
class ElizaChat
```

```
{
static int          num_chats=0;
private String      _keyWordList[];
private String      _responseList[];
private int         _idx=0;
private int         _rIdx=0;
private boolean     _started=false;
private boolean     _kw=true;
public String       _response;
private String      _dbKeyWord;
public int          _widx = 0;
public int          _w = 0;
public int          _x;
private char        _space;
private char        _plus;

public ElizaChat()
     {
     num_chats++;
     _keyWordList= new String[20];
     _responseList=new String[20];
     _rIdx=0;
     _idx=0;
     _keyWordList[_idx]=" ";

     _space=" ".charAt(0);
     _plus="+".charAt(0);
}

public String converse(String kw_)
     {
     _response = null;
     for(int i=0; i <= _idx - 1;i++){
         _dbKeyWord = _keyWordList[i];

         if(kw_.length()>=_dbKeyWord.length()&&
               _keyWordList[i].equals
                   (kw_.substring(0,_dbKeyWord.length())))
              {

              _widx = (int) Math.round(Math.random()*_rIdx-.5);
              _response = _responseList[_widx];
              _x=_response.indexOf("*");
              if(_x>0)
                   {
              _response=_response.substring(0,_x)+

              kw_.substring(_dbKeyWord.length(),kw_.length());
              if(_x<_responseList[_widx].length()-1)
```

```
                        _response=_response+"?";
                    _response=Eliza.ChatConj
                                                .conjugate(_response,_x);
                    _response=_response.replace(_plus,_space);
                }
                break;
            }
        }

        return _response;
    }

    public void loadresponse(String rw_)
        {
        _responseList[_rIdx]=rw_;
        _rIdx++;
    }

    public void loadkeyword(String kw_)
        {
        _keyWordList[_idx]=kw_;
        _idx++;
    }
}
```

在哲学上这只是例行程序，但心智科学通常将这种能力看作是一种需要依赖语言能力的东西。毕竟，语言给我们提供了一种含义和意图的表达媒介。正是有了语言，我们可以以一种有意识的方式描述他人和自己的行动。这或许也是为什么 Alan Turing 提出他著名的测试来作为计算机程序所展现出来的智能的真正量度。如果程序可以成功地与另一个实体(人)进行沟通，并且人不能分辨出它是一台计算机，那么它就必定是智能的。从而，Turing 认为，我们能够成功开发 ToM 的任何事情都是智能的。如果能够让游戏在玩家中引发这种反应，对游戏来说这是个绝好消息。

有趣的是，对黑猩猩和一些更低级的灵长类动物的深入研究表明，它们在确定相互间以及人类的意图和预测方面具有非同寻常的能力，甚至不需要人类层次上的语言沟通。因此，形成关于他人心智的概念的能力要么是生物天生的，这可通过视觉线索来确定；要么是完全其他的东西。但不管是什么，可以确定的是，我们并不需要 AI 控制的智能体掌握完全的语言沟通技能，就能够使玩家具有关于 AI 的 ToM。

我们希望游戏具有这种与生俱来的本性。如果我们能够让玩家看到的不仅仅是具有 X 健康值和 Y 力量值的生物，而是一个有信念、欲望和意图的生命，那么我们已经取得了很大的胜利。而只有当讨论中的 AI 系统像人类一样进行决策，从而表现出它们的高级特性并超越那些相关的简单玩法时，才能使人类玩家停止怀疑。事实上，我们必须模仿思想，而不是行为。行为应该来自我们赋予的 AI 创造物的思想，而不是来自程序员的思想。

注意，这并不意味着我们需要赋予我们的创造物完美地解决问题的能力来实现这种状态。人们本能地试图去创造关于他们正在对付的其他实体的 ToM，从而可以预料其他实体的行动和思想。作为游戏 AI 的程序员，在设计我们希望人类赋予一定品质的实体时，我

们可以好好地利用这个特性。事实上，关于这类基本的、低级目标(生物体试图创造相互的ToM)的知识可以给程序员或设计者一些指导，从而让他们知道哪种类型的信息可以直接提供给玩家，哪种不应该提供，哪种可以模棱两可地处理。正如魔术师所说，“观众看到了我想让他看到的一切”。

举一个例子，考虑分队战斗游戏中 AI 控制的行为。图 1-4 中给出了一个简单战场的布置，人类玩家位于地图的底部，4 个 CPU 敌人对他进行了包围并在多个掩护点之间移动。这些敌人的简单游戏规则如下：

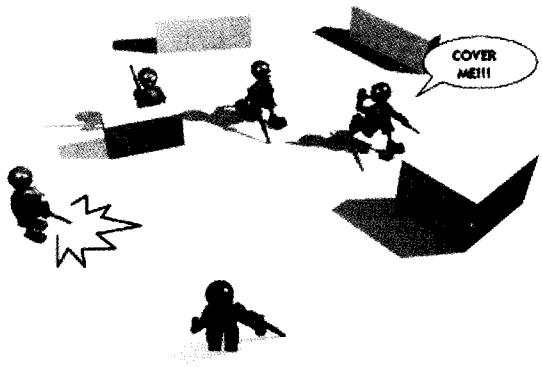


图 1-4　松散协同敌人分队中的自发 ToM

- 如果没有人朝玩家射击，如果我装满了子弹且准备完毕，我将开始射击。注意在这个游戏中每次只能有一个玩家可以射击。
- 如果我暴露在外，我将前往最近的未被占领的掩护位置，并随机地呼喊“掩护我！”或“在你左边！”或甚至只是咕哝几声。
- 如果我处于掩护位置，我将重新装弹，然后等待那个家伙射击完毕，或许可以通过播放一些类似扫描的动画，使得看起来更像他正准备狙击玩家。

现在想象一下这场游戏在人类玩家看来会是什么样子。4 个敌人士兵进入了视野。其中一个迅速开始射击，而其余三个寻找掩护点。然后，先前的士兵停止了射击，呼叫“掩护我!”并向前跑去寻求掩护。同时另一个士兵跳出来并开始射击。在这个系统里，士兵对相互之间、对玩家的意图、对他们执行的一个基本的交替前进和掩护的军事调遣这个事实都完全没有察觉。但由于人类玩家自然地试图形成一个关于敌人的 ToM，在他看来，这是一个高度协同和智能的行为。因此，该策略得以奏效。我们已经设计了一个智能系统，至少在娱乐界看来是这样。

1.5.5　有限最优

为了在我们的 AI 系统中实现一定程度的理性，必须要表明我们所争取的理性的程度，并允许它对系统设计进行约束。如果目标是接近完美的理性，那么最好能够接受程序将运行非常长时间这个事实，除非面对的决策状态空间确实非常小。对于大多数娱乐游戏来说，完美的理性是不希望也不必要的。就像前面所说的，游戏 AI 的目标是仿效人类性能的级别，而不是完美的理想。人类犯错误的原因之一就在于“有限最优(Bounded Optimality，

BO)”的概念。BO 意味着在给定计算约束(以及其他资源)的情况下，系统将做出它能够做出的最优决策。某种解决方案可能的总理性与限制的数目和数量直接相关。换句话说，付出多少就能得到多少。在对世界有限的观察以及时间约束下，我们能够构建一个在范围之内做出最优选择的决策系统。

跟计算机一样，人的决策能力也受很多因素的限制，包括相关知识的质量和深度、认知速度以及解决问题的能力。但这仅仅涵盖了硬件和软件。我们也受环境限制的影响，或许它们使得我们的大脑不可能得到完全地开发。我们生活在一个“实时”的世界，必须在短时间内做出挽救我们生命或者促成我们事业的决策。所有这些因素共同影响着我们的决定，使得不正确性被限制在可允许的范围内。因此，与其强迫程序找到理想的解决方案，还不如只是引导决策朝正确的方向前进，并在有限的时间内朝该方向努力。我们希望，所产生的决策将会更人性化(当然，直到计算能力达到时间片约束可减弱到零的水平)并在有限的平台和游戏类型约束下工作得更好。事实上，我们设计最优的程序而不是获取最优的行动。

BO 这个概念在 AI 理论界(以及游戏理论界和哲学界)也开始变得流行起来，因为对现实问题的所谓的最优解决方案在计算能力上往往都是难以实现的。另一个原因是没有限制条件的现实问题是很少的。根据世界的真实性，我们需要一种不要求绝对理性的测量成功的方法。

在很多类型的系统中使用 BO 方法都存在一个问题，那就是它们需要一个递增解决方案，即当被赋予的资源增加时解决方案也将逐步变得更好。递增解决方案绝对不是对所有问题都是通用的，但那些在计算上存在挑战性障碍并需要 BO 思想的类型往往可以按某种方式简化成一个递增层次。例如，路径搜索就可以设定几个复杂度层次。可以开始在很大的地图区域上进行路径搜索，然后在各个区域内，然后局部，最后在动态目标周围。每个连续的层次都逐渐地比上一个要好，但每个层次都使得玩家朝正确方向前进，至少大部分都这样。

1.5.6 来自机器人技术的启发

机器人技术是与游戏 AI 界具有大量的相似任务的为数不多的领域之一。理论工作试图解决能够使用彻底的搜索而找到最优解的大规模问题，与此不同，机器人不得不对诸如物理学、计算速度和环境感知等实时约束进行处理。实际上，机器人必须要处理一些智能地解决问题的计算难题，同时要掌握将技术应用到物理模型(这些物理模型需要对一定程度的真实世界进行处理)上的技巧。一个具有挑战性的任务往往运用理论知识，并将其与现实世界磨合，直至二者很好地匹配。由于其固有的最优性和在真实世界中的用途(该用途是机器人技术与在实验室完成的理论 AI 工作结合所产生的)，促进了很多来自机器人的技术被应用于游戏之中。在游戏中我们成功使用的路径搜索方法(包括非常重要的 A*算法)很大一部分来自于机器人研究。机器人技术给我们带来的主要启发包括如下几个方面。

(1) 设计与解决方案的简单性。很多机器人的方法论使用 WW 模型，这点游戏也完全认同。这是由于机器人技术总的来说是个非常困难的问题，具有很多类型的挑战，比如通过不确定的地形或识别一般的环境目标等。研究者赋予机器人的每一种真实的官能都可以

转化为数量巨大的技术和研究，它们是将系统分解成多个可使用部分所必需的。如果没有这种官能系统还能工作，那么该解决方案就算不是更好的，起码也是好的，因为必须考虑到没有引入一个复杂感知系统而节省的时间和成本代价。Rodney Brooks 的一些机器人设计对此有非常好的阐述：他的机器人设计并不试图依靠对障碍物的识别来通过某地区，也不去绕过或通过计算来克服这些障碍物，它们在很大程度上是无意识的，类似昆虫的生物，通过盲目地采用一般的搜索方法在障碍物上强行开道。我们受到的启发是，尽管别人花费数年的时间尝试高技术含量的方法来聪明地避开障碍物，但最终还是失败了；而 Brooks 的机器人设计却被采纳到面向火星的机器人中。

(2) 心智理论。ToM 在机器人技术中也是很先进的。研究人员发现，如果人们能够以某种方式赋予机器人一些人类的特性(不是人类思考过程)，那么就能够设计出更好的机器人。这种自然的人类过程(或人性化)对机器人研究人员来说是件好事，因为这实际上使得机器人在人看来更加智能，并在公众的心目中更加合适。设想一个简单地向白光移动的机器人。当人类对这个简单的行为进行描述时，他通常会说这个机器人“喜欢光”或“害怕黑暗”。通过不断地赋予机器人展示欲望和意图的能力，而不是自然状态的行为，研究人员希望可以制造出人们不仅能够忍受而且能够在真实世界中愉快相处的机器人。像 Cog 和 Kismet [Brooks98]等的机器人项目继续推动着人机交互领域的发展，它们主要是通过社会提示来完成的，如促进人们对机器人的 ToM 以及使得交互本身和机器人参与的学习更加生动。

(3) 多层决策体系。很多现代机器人平台都使用一个系统(有时称为子系统)，在它上面运行的机器人决策结构被分为多个层级，它们的排列体现了关于世界的从高级到低级的决策[Brooks91]。这种所谓至下而上的行为设计允许机器人在某种环境下实现一定程度的自主，因为它们总有万无一失的行为可以依靠。因此，一个机器人或许拥有很低的层级，其主要目标就是避免障碍或其他附近的危险。这个层级可以很快地从外界得到更新的信息。它也可以覆盖或修改来自顶层决策结构的行为，因为它代表了最高优先级的决策。层级越高，优先级越低，与环境交互的数量也降低，而整体目标的复杂性增加。因此，在最高层级，机器人对高层规划进行构思：“我需要离开房间。”系统内的玩家相互不了解(或了解极少)，他们简单地以这样的方式建立相互联系：通常与整个目标相联系的不同任务专门集中在不同的层级。层级间的这种独立性也带来了系统的高度鲁棒性，因为它意味着一个层级的混乱(或接收到错误数据)并不会破坏整体结构，因此，只要系统的其他部分返回到常态，机器人将仍然能够完成任务。

这种类型的结构非常适用于需要在多个层级复杂度上同时进行决策的游戏类型，如 RTS 游戏。按照采用分层(subsumption)技术的机器人团队给出的一般技巧，我们可以从这些系统所展现出来的相当多的好处中获益，包括自动容错(在系统各层级之间)和处理各层级上未知或部分已知信息的鲁棒性。分层体系结构不需要一个清晰的、从头至尾的行动规划，并且设计好的系统可以按代表环境允许的最好方式的顺序自动执行智能规划中的不同部分。本书将采用第 23 章中的一种类似方法来讨论分解 AI 引擎问题的一般方式。

1.6 小结

本章讨论了在后面各章中我们要用到的一些基本的 AI 术语、一般的心理学理论以及来自可应用到 AI 系统设计的其他领域的概念。

- 本书用游戏 AI 来表示基于角色的行为决策，进一步集中在那些需要在多个良好决策中进行选择的任务，而不是寻找最优的可能决策。
- 老的游戏使用一些模式，或通过赋予计算机对手一些人类选手所不拥有的隐秘知识来进行作弊，由于日益强大的 AI 系统被用于游戏中，这两种方法都正在逐渐地被淘汰。
- 随着玩家对更复杂的游戏需要更好的对手，在如今的游戏中 AI 正变得越来越重要。这的确是实情，尽管很多游戏开始在线进行，但大多数人还只是在玩单人模式的游戏。
- 由于主要是一种娱乐形式，游戏 AI 需要更智能和更具趣味性。因此，游戏 AI 需要展现一些人类的错误和个性，能够使用不同的难度等级，能够让人觉得有挑战但又不是太具挑战性。
- 大脑的组织结构给我们展示了以复杂的顺序彼此构建的目标导向系统的使用。
- 像大脑一样，AI 系统能够使用长期存储器和短期存储器，这将引导我们探究更真实的 AI 行为。
- 游戏中的学习跟在真实大脑中一样，可以是有意识或无意识的。通过使用这两种类型，我们能够模仿随着时间流逝而修正得更加真实的行为，同时仍然将我们的学习集中在我们所注意的地方。
- 认知研究引导我们将 AI 的推理系统看作是过滤器，它接收我们的输入并给我们一个明智的输出。对给定游戏状态空间的本质进行考虑，并将它与可用的 AI 技术进行对比，就会找到适合游戏的正确“过滤器”。
- 通过把给游戏玩家一个关于 AI 控制的主体的 ToM 作为目标，可以将智能体的特性延伸到基本需求和欲望，从而将其决策过程的真实性延伸到玩家。
- 有限理性是一个形式上的概念，它可以将游戏 AI 的目标形象化，即我们追求的并不是最优的行动，而是在存在诸多约束时能够给出好的解决方案的最优程序。
- 机器人技术给我们提供了设计和实现简单性的理念，延伸了赋予我们的创造物 ToM 的需求，并给我们提供了一个至下而上设计和实现自主智能体的一般分层体系结构。

2 AI引擎的基本组成与设计

本章将对一般 AI 引擎的基本组成进行分解和论述。图 2-1 给出了 AI 引擎的基本布局。尽管这个列表并不是包罗万象的，也不是唯一的分解方式，但几乎所有的 AI 引擎都将以某种形式使用这些基本系统：决策与推理、感知、导航。

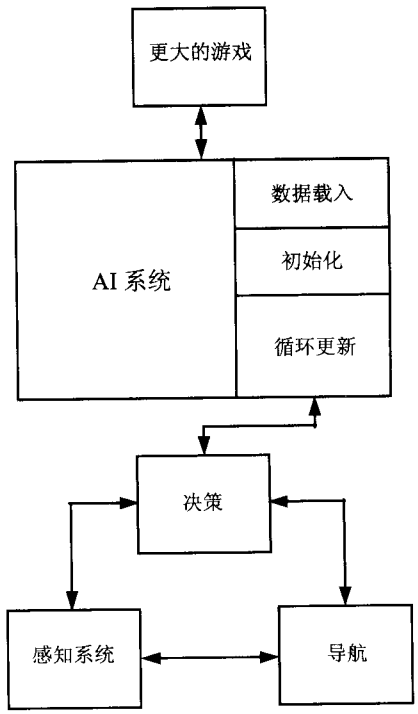


图 2-1　AI 引擎的基本布局

2.1　决策与推理

决策系统是引擎中使用最广泛的部分，也是本书的重点。决策系统是决定用户构建的 AI 引擎(或引擎子部分)类型的底层结构。推理可定义为从实际知识或假定为真的前提中获取逻辑的或合理的结论的行为。用游戏术语来说，这意味着 AI 控制的对手获取关于世界的信息(参见 2.2.1“感知类型”一节)并做出关于如何响应的智能的、合理的决策。因此，

AI 系统受它能获取的外界信息的限制，也受游戏设计时所定义的响应集的丰富程度的限制。游戏允许 AI 角色做的事情越多，游戏的响应集或状态空间也就越大。用户选择的技术应该由用户要构建的游戏的规模和状态空间的大小来决定(至少是部分决定)。关于此概念的更多信息将在第III部分和第IV部分给出，在那里将对不同的技术进行描述。

本书描述的所有类型的决策系统都可归结于这个定义：利用可得到的输入来获取解决方案。这些技术之间的差别将决定用户选择使用的类型(或组合)。我们关注的主要区别有：解决方案的类型、智能体的反应能力、系统的真实性、游戏类型、游戏内容、游戏平台、开发限制以及娱乐限制。

2.1.1 解决方案的类型

主要的游戏解决方案类型有两种：战略型和战术型。战略型解决方案通常针对长期、高层次并需要多个行动共同来完成的目标。而战术型解决方案则更多针对短期、低层次并通常只包含一个物理动作或技巧的目标。二者的区别可通过 Quake 式游戏中 HuntPlayer 和 CircleStrafe 这两个解决方案来进行说明。追捕玩家是一个高层目标，它包括了发现玩家、接近玩家以及在格斗中与玩家交战。绕行扫射只是在与敌人交战中所使用的一种移动方式。许多游戏同时需要战略型和战术型两种解决方案，因此开发人员根据这种划分将问题分离成几个独立的部分，并结合不同的技术来实现。

2.1.2 智能体的反应能力

游戏元素应该如何进行反应？脚本化系统趋向于设计具有更多程式化和语境相关响应的角色，但它们也容易受困于这些行为脚本，从而失去反应性。与此相反，完全反应式系统(它们接收输入便立即改变响应，而对之前的行为并不关心)容易被认为过于机械或作弊，并且不符合人类的感情。强响应系统也需要一个非常丰富的响应集，否则它们表现的行为将是可预测和无新意的。然而，这对街机式游戏(arcade style)或所谓的“twitch”游戏来说是非常合适的。这需要在所设计的游戏类型和对期望设计的玩法经验的正确权衡的基础上进行阐述。

2.1.3 系统的真实性

要被认为是“真实的”，AI 元素做出的决策和行动必须是类似于人类的。在游戏的限制条件下，每个 AI 实体都需要智能来决定所要做的正确事情，但类似于人类也意味着会犯错误。因此，AI 角色也需要很好地体现人类的弱点。化解玩家所有拳击或射击永远不会落空的对手，或 Scrabble 游戏中对整个字典了如指掌的对手都不会让玩家高兴，反而只会让他感到沮丧。我们的目标是要在竞赛和娱乐之间找到平衡，从而使玩家被游戏的挑战性所吸引，同时通过击败游戏形成一个持续的积极反馈。其他真实性问题还包括对游戏使用的物理定律的实际认同数量。玩家能跳得比实际生活中的更高吗？他能飞吗？是否能很快痊愈？所有这些都得由开发人员来决定。这意味着“真实性”可定义为“特定游戏世界中的真实性”。在幻想世界中尤其要小心，因为肆意破坏规则的敌人将被认为是在作弊而不是具有魔法。必须要采取措施以确保玩家了解并遵守游戏世界中的规则。还要记住，地球物

理定律对于大多数玩家来说通常都是知道的，但在他们设法习惯开发人员设定的规则时，那些特殊定律最初可能会成为他们的绊脚石。

2.1.4　游戏类型

不同类型的游戏需要不同类型的 AI 系统。本书第 II 部分将对游戏类型进行深入讨论。在游戏类型层次上，必须要记住下面几项：

- **输入(或感知)类型**。需要注意的事项包括：输入的数量、频率、信息传递方式(轮询、事件、回调函数等)以及输入之间的层次关系。街机式游戏只有非常有限的输入，而实时策略游戏中的角色可能需要非常多的关于世界的感知，以便导航地形、保持编队、帮助友好单元、接受人类命令并对攻击的敌人做出响应。
- **输出(或决策)类型**。根据感知系统赋予引擎决策部分的信息，AI 系统将做出一个决策或产生一个输出。输出可以是模拟的、数字的或者是在周边行为之上的一连串事件；它可以包含整个角色(如潜水以寻求掩护)，或只是部分角色(如扭头以响应噪声)，或多个角色(如让市民们挖更多的石头)。输出可以是具体的(以某种方式影响一个单一角色，如腾空一跃)，也可以是高层的(如“我们需要制造 Dragon 单元”)，其中后者将影响许多 AI 角色的行为并改变许多决策的进程。
- **游戏类型需要的决策整体结构**。有的游戏具有非常简单或单一性的决策。Robotron 游戏就是一个很好的例子。怪物以一定的速度和运动类型向玩家靠近，并试图杀死玩家。但在复杂的游戏中，如 Age of Empires(《帝国时代》)，需要在游戏中进行许多不同类型的决策。可以使用小组级策略、单元战术、一系列路径搜索问题以及其他更深奥的问题，如外交手段等。它们之中的每一个问题都可能代表了 AI 中的一个子系统，而且这些子系统采用完全不同的方法来完成这些工作。

2.1.5　游戏内容

上述游戏类型是游戏玩法关注的特例，它由特殊或新奇的游戏内容来决定。Black & White(《黑与白》)之类的游戏需要为它们基本的游戏玩法机制设计非常专业的 AI 系统，即通过对它引导或展示如何完成工作来教它学会主要的动物行为。当预先设计框架时，需要进行深思熟虑，同时可以向早期的原型性工作寻求帮助以克服设计上的缺陷。

2.1.6　游戏平台

游戏是为哪种平台创作的？个人电脑、家庭控制台、街机体系还是掌上电脑？尽管这些不同机器之间的界限已经开始变得模糊，但各自还是有它们自己的具体要求和限制的，这是我们必须要考虑到的。关于各个平台的一些 AI 考虑有：

- **个人电脑(PC)**。在线 PC 游戏可能需要用户的可扩展性(以包括级别或 AI 编辑器的形式)，因此 AI 系统需要对这些进行处理。单人 PC 游戏通常具有非常深的 AI 系统，因为 PC 游戏玩家年龄一般偏大，他们希望更大的复杂性和对手的真实性。PC 的标准输入设备是鼠标(除了飞行模拟游戏或赛车游戏外)，因此必须要记住，人类

玩家将用这个器件来执行各种具体的命令，而且如果让 AI 单调地执行那些任务或者执行一些鼠标不可能执行的任务，那么他们会大叫犯规。同时，持续变化的 PC 也意味着对大多数游戏来说，最小配置也逐步提高，因此，在做出设计决定时，AI 程序员需要预测游戏运行的最小配置(通常是游戏开始后的一到三年)。PC 游戏的体验通常也较长(超过 30 小时的游戏可玩性)；因此，AI 对手需要经常变化，这样与它对抗才不会觉得重复。

- **控制台(Console)**。由于控制台的游戏玩家通常较为年轻而且对虚幻的场景更加开放，因此对它的现实约束也有所提高。然而，由于玩家技能高低的范围很大，现在对难度级别设置有了更广泛的应用。存储器和 CPU 的预算通常更为严格，这是因为与 PC 相比这些机器(至少到现在为止)更为有限。从质量保证的角度而不是玩法体验的质量角度看，控制台游戏在很大程度上具有更高的质量标准。控制台上的程序通常不会崩溃，尽管“仅限 PC”的问题开始蔓延到控制台世界。然而，由于这种较高的标准，在被核准发行前 AI 系统需要承受更长时间和更艰辛的测试。很多公司在内部对他们的游戏进行测试，并且在游戏上架销售之前控制台的制造者也要对游戏进行测试。因此，游戏中使用的任何“外来”AI 样式(如学习系统)都有可能使得这个测试过程变得更为冗长，因为一些先进的 AI 技术往往具有内在的不可复制性。
- **街机(Arcade)**。在 20 世纪七八十年代，先进的图形硬件具有过高的成本，难以在普通家庭中普及，同时家庭控制台在其显示效果上也还非常简单(如 Atari® 2600™ 和 Coleco-Vision®)，这使得街机平台取得了巨大的成功。由于如今日益强大的家庭机器，街机行业必须要做出巨大的改变。现在，大多数街机机器都属于下列三种类型之一：大的、定制的控制柜(如坐着完成的赛车游戏或滑雪模拟器)，定制输入(轻武器游戏、音乐游戏)，以及可在酒吧的角落或其他非专门的街机环境中安放的小游戏。Golden Tee 高尔夫球游戏就是属于第三种类型的一个例子。由于是定制的街机机器，其最高等级往往就是硬件的上限。整个封装也是定制的，因此开发者可以自由安放尽可能需要的 RAM 和处理能力(当然是在合理的限制内)。小游戏事实上更多的是以“套件”的形式销售，这样其拥有者可以用老游戏的部分去换取一些较新游戏的相应部分。街机的 AI 仍然是“基于模式”的，因为当人们投入 2 角 5 分硬币时(在其他现代游戏中是 1 美元或更多)，他们可以设想将会发生的事情。对街机环境的 AI 进行调整通常涉及到在当地放置一台二级测试机器，并从机器中取回统计数字，以确定该游戏难度是太简单还是太复杂，或是其他可能对挣钱不利的事情。因此，街机世界中的 AI 通常都是简单的，但因为要设法权衡娱乐因素和经济效益，对它的调整将十分困难。
- **掌上电脑(Handheld)**。掌上电脑是最严格的平台，它曾经一度是 Nintendo® Gameboy®的天下，但近来变成了游戏开发的热门领域，像 PDA、移动电话以及其他所有现在能想到的小玩意几乎都开始进入游戏世界。这些机器通常具有非常小

的 RAM，输入键的数量也非常有限(甚至在移动电话中也是如此，它不是真正的游戏控制器，因此不能识别同时按下的多个键)，并且那些袖珍型机器的图形能力也非常弱。事实上，过去常与 8 位或 16 位技术打交道的人再一次找到了可以施展他们才华的地方。这些平台上的 AI 需要更智能，同时空间和速度也要最优。正因为如此，这些机器通常为其 AI 系统采用一些倒退的技术，比如模式化运动、类似于无意识障碍的敌人、作弊(由于它们是程序的一部分，故可运用只有它们才有的关于人类的知识)。然而，随着掌上电脑系统变得日益强大，这种情况将会改变，而且掌上电脑和控制台之间的界限也会变得模糊。

2.1.7　开发限制

开发限制包括预算相关事项、人力问题和时间长度。基本上，AI 程序员需要将所有这些事项转化为他的一个资源：时间。AI 程序员确实需要对时间有一个很好的判断。有多少时间可以投入到设计阶段、制造阶段以及最后的测试或调整阶段？已经反复证明，该过程的最后一个阶段是最为重要的，因为最好的游戏都不可避免地是高度修饰的。是的，设计一个系统也同样重要，因为一个设计得很好的引擎将使得程序具有快速并轻易向游戏添加必要的行为内容的能力。但是，即使游戏设计得再好，也需要对它进行大量的调整以得到正确的体验。

由于游戏中 AI 的角色本质上是高层的(而不是低层的引擎代码，如数学库或渲染器)而且新思想和行为似乎不可避免地出现在产品晚期，因此 AI 系统因“特征蠕变(feature creep)”而声名狼藉。它可定义为一种附加在项目末端的新特征，使得最后的数据蠕动到未来状态之中。这表明下列二者必有其一：糟糕的游戏需要额外的元素使它变得有趣或具有可玩性；优秀的游戏只是在那方面做得更好。如果是属于后面那种情况，那么做得不错。如果管理人员愿意加大额外的时间和金钱投资，使得产品获得超越最初设计的性能，那么也很不错。但如果极快地添加额外元素来使有问题或失败的游戏看起来更好，那么这将是一个灾难。一个好的、超前的游戏设计是解决该问题的最好方法，但生产人员需要仔细并严格遵守时间表，从而减少这个弊病。

就像将在第 II 部分看到的一样，几乎所有游戏都采用某种形式的基于状态的 AI(如果不是作为主要系统)。一般而言，这主要是由游戏的特性决定的。人们希望游戏中至少具有一定的可预测性——如果玩家经常玩一些永不停息、时刻变化的对抗游戏，那么他的精力将很快消耗殆尽。大多数游戏中的 AI(或通常的游戏玩法体验)都需要具备一定的周期性，即行动阶段，然后是休息阶段，然后再重复。这种节奏对基于状态的方法非常有利。然而，大多数游戏采用组合引擎，对于游戏范围内发现的不同 AI 问题采用不同的决策系统来进行处理。因此，不要以为基于状态的模型是唯一可行的方法。

由于基于状态的方法将整个游戏的状态空间有组织地划分成易处理的几部分，因此它非常流行。我们不需要对游戏中决策之间的逻辑联系进行整体处理，而实际上只需要将游戏划分成更小且更容易处理的子游戏。这就是为什么即使是那些总体上不大符合状态结构

的游戏也能够从高层状态机的划分中获得好处的原因。这些高层状态机能够通过定义只含有能提供划分的内部状态的状态，将解空间划分为便于处理的小块。例如，Joust 是一个动态性很强的游戏，每一个关卡都非常相像(除了 egg 阶段)，并且其 AI 系统更多的是基于规则而不是基于状态。但可以将 Joust 的一般关卡划分为 3 个状态：Spawning 状态(对敌人进行实例化)、Regular 状态(在正常游戏玩法中)和 Extended 状态(时间耗尽，Pterodactyl 在寻找人类玩家)。另外，还可以对 Regular 状态进行进一步划分，以提炼更多的行为。因此，可以确定 AI 角色是位于系统的底层、中层还是高层，并在事实上使它成为一个状态，这样 AI 系统就可以对这种位置状态做出不同的响应。很明显，这种信息在 Regular 状态(例如，Regular 状态可以具有一个基于角色放置划分行为判断的转换声明)下可作为一个简单的修正器。通过把它当作是另一系列的状态，产生的每个状态本身都非常简单，而不是更复杂的 Regular 状态。至于这样做会不会造成有组织的简单性与重复代码之间权衡的困难，则需要在实现中进行确定。

在游戏 AI 中状态机具有优势的另一个原因是其针对测试、调整和调试目的。质量保证人员(QA，或测试者)将要花费很多时间来确定游戏 AI 是否有缺陷，或者太难，或者当游戏 AI 系统在某些情况下不具有可再生性时计算机会不会崩溃。基于非状态技术的游戏调整将会更加艰苦，同时也很难给出具体的建议(我们知道制造者有着大量具体的建议，但有时在产品接近完成时这将非常危险)。本书的第III部分和第IV部分将在技术基础上对此类问题进行更详细的讨论。

2.1.8 娱乐限制

游戏也因一些假定的娱乐元素而出名，并且很多游戏玩家对缺少这些元素的游戏表现出了有意或无意的消极反应。这包括了诸如 rock-paper-scissors(RPS)的场景。在游戏设计中普遍用到的一个概念是“每一件可以完成的事情都应该有反制手段”，这导致了 RPS 比较。如果游戏对手具有人类玩家不能反制的能力，那么最好能有一个好的原因，否则游戏将不会引起太多的兴趣。但是，如果人类能够做一些 AI 不能反制的事情，那么这个游戏又太过简单，同样也会失败。这是一个经典的游戏平衡问题，它对于一个游戏的最后成功起着至关重要的作用。

难度级别问题是 AI 系统需要回答的另一个娱乐问题。一般认为静态技能级别(通常由玩家在游戏开始之前设定)要比动态技能等级(游戏级别随着玩家的进展而实时变化)好。这是因为大多数玩家希望知道他们试图打败的挑战级别，尽管也可以设置一个“静态”难度级别，但玩家知道它将随着游戏进展而进行调整。这是因为人们的技能级别差别很大；在具体的任务级别中，动态技能级别的调整很难进行；静态技能级别让人觉得游戏非常平稳并且没有作弊。有些人喜欢为游戏担忧，喜欢焦急等待的感觉，但其他人只是希望休息一下，像旅行者一样轻快地走过，走马观花。动态技能级别的另一个问题是必须以某种方式将那些探险式或非标准的行为剔除出去，这些行为是人们从与“被卡住”或失败(由于困难)相联系的行为中产生的。

由于我们是在设计计算机游戏，而不是电影，因此还有一个问题没有重点解决，即玩家理解 AI 角色的感情和意图问题。在电影或电视中，这可以通过生动的摄像机视角、很多对话以及面部表情来完成。在游戏中，使用摄像机视角非常困难，因为(尤其是在 3D 游戏中)控制方案可能会依赖于摄像机，否则为了玩游戏可能需要广角摄像机(例如，在足球游戏中，可能需要观察大部分场地，甚至是某个人脸的相当短时间的特写，这都会损害到游戏的玩法)。因此，我们只有有限的工具来理解此类信息。我们可以对表情进行滑稽模仿，这在大多卡通类游戏(如 Crash Bandicoot 或 Ratchet and Clank)中都有用。在那些游戏中运用经典的卡通式拉伸和压扁将有助于赋予从远处走来的角色以感情，而不用使用靠近的摄像机。对话也有帮助，但容易重复，同时也需要一定水平的假唱以看起来更好。脸上有着悲伤的表情，但说话时正常拍打下颌，这样的角色并不能传递我们所要表达的感情。我们需要意识到，大多数动作都要能很明显地觉察到。如今平台的更好的图形能力将使这个问题变得容易解决一些了，因为我们能模仿更多复杂的特征并使用更微妙的动画来使它们变得更生动，但家庭控制台仍然受常规电视的有限分辨率的困扰，因为在非高清晰度电视上那些微小细节往往都不复存在。

2.2　输入处理机与感知

AI 感知可定义为希望游戏中元素响应的环境中的事件。因此，它可以像玩家位置一样简单(在 Robotron 游戏中，除了敌人自己的位置外，这是这个著名的 AI 唯一的输入)，也可以像在实时策略(RTS)游戏中对计算机看到的人类使用进行仔细记录一样复杂。通常，如果可能，这些类型的数据寄存器被封装成一个单个的代码模块，从而能够更容易地对系统进行添加，可确保在 AI 系统的其他部分没有做重复操作，有助于调整，并将计算提炼到一个最容易优化的中心位置。

中枢感知系统也能够在每个输入寄存器中对额外的数据或需要考虑的事项进行标记，包括感知类型、更新规则、反应时间、门限、负荷平衡、计算代价与前置条件等。

2.2.1　感知类型

不同类型的输入可能包括布尔型、整型、浮点型等。它们还可能包括静态感知和动态感知。静态感知的一个例子是篮球游戏中逻辑所需要的感知，如“控球技能大于 75”，它只需要确定一次，除非游戏允许技能在游戏过程中进行调整。

2.2.2　更新规则

不同的感知需要以不同的频率进行更新，同时也不一定追求很高的更新频率，因为感知变化没有那么快或者始终重新计算的代价非常昂贵。这可以认为是一种形式的反应时间，但不完全正确。这更像是一种轮询式的感知，人们不会在意它是否有些过时。继续我们篮球游戏的例子，我们可以通过视线检测来确定持球者是否具有一条通向篮板的无障碍通道。这种检测的代价是非常昂贵的，特别是在要对所有的移动角色使用预测来确定他们是否会及时让出通道的情况下。因此，可以每隔一段时间检测一次，而不要一直检测。

2.2.3 反应时间

反应时间是在敌人确认环境变化之前的暂时停顿。当反应时间为零时，计算机就纯粹是一台计算机。但是在敌人确认事情之前安排一点随机大小(或根据技能特性来决定)的暂停时间，将会使得系统的整体行为看起来更具人性和更公平。这也可以用于调整难度级别，使整个游戏的难度跟期望值差不多。反应时间也能使角色具有一点点的个性，从而使那些快的角色比慢的角色反应更快。

2.2.4 门限

门限是 AI 能够响应的最小值和最大值，这可用于简单的数据范围检测，也可用于模仿稍微有点耳背的角色(其最小听觉门限比其他角色的最小听觉门限高)或目光敏锐的角色(能看见所有的运动，而不仅仅是大的或快速的运动)。响应游戏事件的门限也可以下降或升高，同样地，这可以用于模仿感知的退化或增强。因此，一个闪光弹会暂时使对手什么也看不见，但巡逻的守卫由一个未经辨识的声音实际上变得更加敏锐，因为他暂时集中了更多的注意力。

2.2.5 负荷平衡

在一些游戏中，AI 需要考虑数据的数量可能非常庞大或者计算量非常繁重，从而难以对游戏的任何时间点进行评估。建立一个感知系统，以便可以指定具体输入变量的更新时间。这是一种使系统负荷平衡的简单方式，它可以防止那些很少变化的事件占用太多的 CPU 时间。

2.2.6 计算代价与预处理

除了刚提到的使计算负荷平衡外，还需要对计算代价进行粗略考虑。因此，在我们脑海中要设计一个具有分层连接计算的系统，从而首先进行简单的预处理计算，如果能够基于这些简单计算，则可以完全不用进行任何复杂的判决。为了给出一个极度简化的例子，我们假定在 Pac-Man 游戏中，控制主要角色的 AI 程序需要进行两个计算：能量丸(power pill)的数量以及与每个能量丸的距离。Pac-Man 游戏最好能够在计算它与所有能量丸的距离之前先检查能量丸的总数量(通过检查 Power Pill Count 之类的变量，或对不同的能量丸进行轮询从而确定有多少处于活动状态)，以确定至少有一个，因为计算距离需要耗费更多的计算代价。

为游戏选择的感知系统很可能是针对具体游戏的，因为 AI 系统响应的输入在很大程度上取决于游戏的类型、玩法强调的重点、角色或敌人具有的所有特殊能力以及其他事情。AI 系统需要的一些数据来源于对人类传感系统(如视线或听力范围)的模仿，而其他数据则仅仅是直接使用游戏中的信息。要确保不要太远离后一种情况，否则将会有作弊的危险。很有可能需要使用扩展信息来为此类输入设计一个好的算法，因为它在计算或空间上的代价非常昂贵(例如 AI 所到之处的详细地图，或对某人在玩家后面这种感觉进行建模)。

更新感知寄存器的两种主要样式如下：

- **轮询(polling)**。轮询涉及到在游戏循环基础上对每一个游戏循环中要变化的具体值进行检测或计算，例如时刻检测篮球玩家是否有机会传球。这对 AI 响应的大多数数据来说是必要的，但这种数据也更需要进行负荷平衡(如前所述)。对模拟输入(连续的或实值)或可能以其他形式表示的输入都要始终使用这种方法。
- **事件(event)**。事件在某些方面刚好与轮询相反。输入本身告诉感知系统它已经发生了变化，而感知系统注意到这种变化。如果感知系统没有接收到事件，那么它不会做任何事情。这对于不会经常变化(而不是像人类玩家位置那样每秒变化 30 次或更多)的数字输入(开/关，或枚举状态)是个首选方法。原因在于，如果将一串源源不断的事件注册、排队，然后执行，那么只是给轮询系统(针对这个特殊的输入)增加开销而已。

一些游戏(尤其是秘密类游戏)广泛使用先进的感知系统。因为敌人的感觉是对付玩家的一种武器，并且除了游戏目标外，大部分游戏体验都是关于如何击败感知系统的。关于它们的更多知识，可以参见第 5 章“冒险类游戏”。

2.3　导航

AI 导航是关于如何从 A 点到达 B 点的技术。在对一些更真实、恐怖或戏剧性游戏的研究中，现代游戏世界一般都包含了巨大且复杂的环境，比如多种类型的地形、障碍、活动目标等。存在非常成熟的 AI 算法能够用于解决上述问题，这得益于机器人领域，因为该领域需要处理机器人在极其困难的环境中调遣的问题。导航包括两个主要的任务：路径搜索和避障。

路径搜索是一个有趣、复杂同时又相当令人沮丧的问题。在以前，几乎不存在路径搜索，因为环境非常简单，或者广泛开放(像在 Defender 游戏中那样，敌人仅仅是朝玩家所在的方向前进)，或者敌人并不朝玩家的方向前进，而是玩家必须要避免的随机方向(就像 Donkey Kong 游戏中的桶一样)。但当游戏开始具有可以在其中走来走去的真实世界后，所有这些都发生了变化。为了让 AI 角色从游戏世界中的 A 点走到 B 点，需要一个可以帮助它找到路径的系统。人们已经提出了一些不同的方案来做这些事情，包括基于网格、简单避免与位势场、地图节点网络导航网格以及组合系统。

2.3.1　基于网格

在基于网格的系统中，世界被分成同样大小的网格(可以是方形或六角形)，并使用 A* 算法(路径搜索的一种非常重要的算法)或其他派生算法来寻找使用网格的最短路径。每个网格块都有一个“通过可能性(traversal possibility)”值，通常是从 0(完全不能通过)到 1(完全开放，可以通行)。简单系统可以仅对网格使用二元值，而复杂的设置则可能需要使用完全模拟值来表示网格的高度(以便能够模仿上山比下山难)或网格块的特殊属性，如“水”或“有人站在这儿”等，如图 2-2 所示。基于网格的解决方案关注的是网格的绝对存储空间以及系统找到最短路径所需临时数据的存储空间。如果网格的分辨率很高，那么由于搜索算法的工作量随着网格的变细而急剧增长，其代价也是难以承受的。

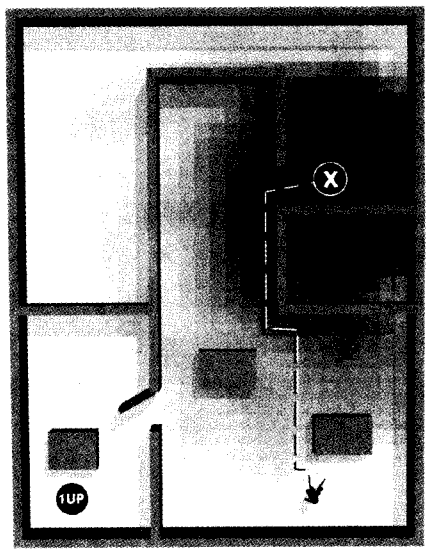


图 2-2 网格方块示例

2.3.2 简单避免与位势场

在简单避免与位势场方法中，也需要将地图划分成网格，并对每个网格区域进行赋值，这样 AI 角色将从高位势值区域向低位势值区域行进。在具有凸起障碍物的开放世界中，可以对这项技术进行预处理，得到该空间的一个几乎最优的 Voronoi 图(例如，许多研究利用 Voronoi 方法对空间进行了数学上充分最优的划分)，从而使路径搜索具有更高的品质(由于路径不是通过繁重的搜索，而是通过简单跟随位势值减小方向从现有数据中提取出来的，因此它的速度更快)。然而，就算有凸起障碍物，我们也不能对它进行预处理，因为并不知道行进的方向。如图 2-3 所示，在这种情况下，问题在于如何产生位势场。

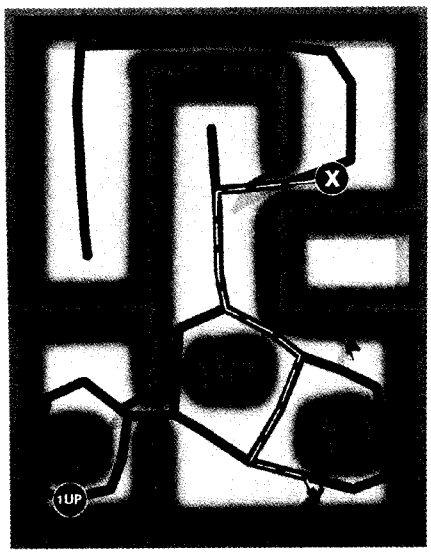


图 2-3 预处理后的位势场

2.3.3　地图节点网络

地图节点网络用在比网格系统能简单表示的世界更庞大的世界，或是更三维(3D)的世界。根据这种方法，当关卡设计人员设计关卡时，实际上是设置一系列的轨迹点(waypoint)，这些轨迹点代表了组成关卡的房间和大厅之间的相互连接关系，如图 2-4 所示。之后，与基于网格的方法一样，使用某种搜索算法(大多数是 A*算法)找到这些点之间的最短连接路径。事实上，这里使用的是与前面描述的一样的方法，只不过是对算法难以操作的状态空间进行了简化。该系统的存储代价非常低，但是这里还有另一种代价。应该正确地设计节点网络，从而能够更好地进行路径搜索，同时就算是关卡发生了变化，该节点网络也会继续存在。另外，该方法不能很好地适用于动态障碍，除非不介意在节点网络中插入动态目标位置。较好的方式是采用某种形式的避障系统来对付移动目标，而利用节点网络来实现在环境中穿行。当玩家足够靠近某物时，避障系统会帮忙躲开它，但只要有局部障碍挡在路上，避障系统就不知道哪个才是通往下一个路径节点的方向。

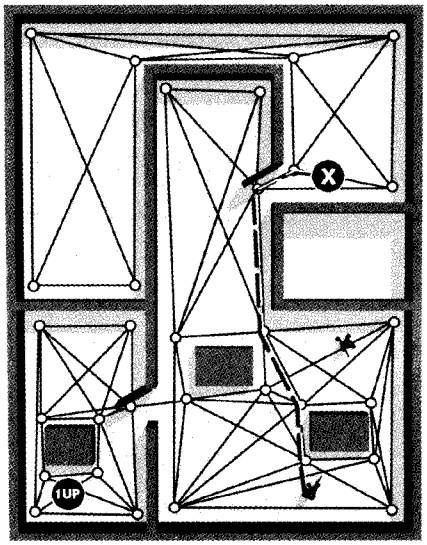


图 2-4　地图节点网络系统

2.3.4　导航网格

导航网格系统试图具有地图节点系统的所有优点，却不需要产生或维持节点网络。通过使用用于构造地图的实际多边形，该系统从算法上构建了一个 AI 能够使用的路径节点网络，如图 2-5 所示。这是一个非常强大的系统，但如果构建导航网格的方法本身不够智能，或在建造关卡时没有用到这个过程将会被执行的知识，那么会出现一些看起来非常奇怪的路径。此类系统最适合于简单的导航，因为从游戏玩法的特定路径特征(比如远距传输器或升降机)很难提取出一个一般的算法，除非关卡设计者设置一些具体的与那些特定玩法元素相联系的且用于构建网络的连接数据。如果要设法让关卡设计者从那些处理导航问题的麻烦中解脱出来，那么首要的就是要摈弃自动产生导航网格的意图。

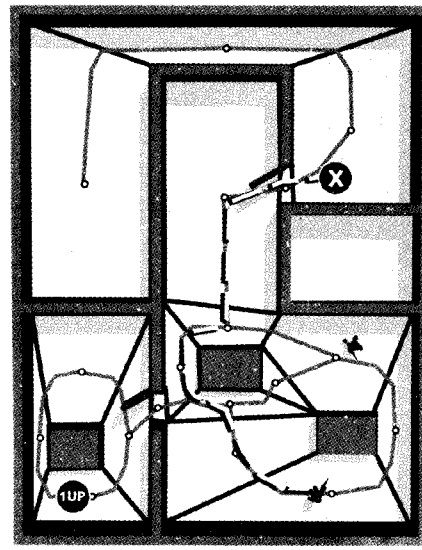


图 2-5 导航网格系统

2.3.5 组合系统

有些游戏使用上述技术的组合。相对开放的世界可以使用导航网络，但其地下通道则依赖于路径节点网络。具有很多有机生物运动(如大量的鸟或成群的动物)的游戏可以使用位势场的解决方案来强调群体行为，但对更具有人类特点的生物则采用固定的路径搜索系统，或当在天空中飞行时则采用只有 UFO 才会使用的特殊节点网络。通过组合使用各种技术，可以使系统的任何部分都避免负担过重，因为这是让系统做它最擅长做的事情。当最初的技术失败后，可以依赖另一种技术。

另一方面，避障是一个更简单的导航任务。它涉及到避开在玩家行进路线上的物体。避障与“闪避(dodging)”类似，都是为了要避开物体而临时改变路径。路径搜索系统会为玩家找到一条可以到达目标位置的合法路径，但由于某个物体恰好挡住了他的去路，因此需要暂时调整行进方向。这样，世界中的动态障碍可以与静态路径搜索系统分开进行单独处理。

避障通常可通过几种不同的方式来完成：

- **位势场**。如果已经使用位势场方法进行路径搜索，那么也可以使用类似方法进行避障。需要做的仅仅是对各种动态障碍赋予一个额外的排斥力(repellant)，该排斥力以障碍的当前位置为中心，在远离它们的各个方向上都存在，从而使得任何人都不能太靠近它们。越靠近障碍时，排斥力也越强，直到在某个最小距离位置它最终停止了入侵者的所有运动。
- **操控行为**。早在 1987 年，Craig Reynolds 就发表了一篇文章[Reynolds87]，详细阐述了一种他称为“boids”的行为系统。在这个系统里，生物以群体方式移动并具有不需要复杂规划的有组织的行为。1999 年，他发表了另一篇题为“Steering Behaviors for Autonomous Characters(自主角色的操控行为)”[Reynolds99]的文章对他的研究进行了更新。一直到现在，游戏都在借用这篇文章的思想。在文章中，

他阐明了只需要少数的精确力，就可以轻易地让 AI 控制的角色模仿真实的运动模式。尽管 Reynolds 的方法被普遍应用于“群聚”系统(处理大量的生物，如鸟和鱼)的实现，但同样的系统也可以用于一般的运动，包括避障。系统使用很简单的传感器来确定可能的碰撞，然后用非常容易理解的基本矢量数学做出相应的反应，并最终实现。

关于路径搜索有非常多的文献。很多早期游戏在这方面做得不好，受到吹毛求疵者的批评。但事实上，该 AI 任务是 AI 界研究得非常多的问题之一。本书不会对具体路径搜索问题的实现进行深入钻研，但读者可以参考随书光盘中对这个重要的 AI 引擎子系统的材料的链接。

2.4　综合考虑

通过考虑所有这些应该考虑的事项，并了解不同 AI 技术(见本书后面几部分的描述)的优缺点，我们可以为游戏的 AI 需求找到一个合适的解决方案。

AI 引擎设计的基本步骤如下：

(1) 确定 AI 系统的不同部分，并使它们成为引擎中完全独立的部分，且每一部分都由某一特定的 AI 技术来加以解决，有些是游戏类型相关的。如果要对简单的格斗游戏进行编码，则只需要一种真正的 AI 系统。或许读者正在设计一个格斗游戏，其大部分工作都是数据驱动的，而且角色编剧将完成大部分的工作。但如果要对大的 RTS 游戏进行编码，则可能需要好几个子系统来实现构成这种类型游戏的许多级别的 AI。

(2) 确定系统输入的类型，即它们仅仅是数字量(开/关)，还是一系列的枚举状态，或者是全浮点模拟值，或是上述的组合。

(3) 确定系统使用的输出。与输入一样，或许会有非常不同的输出，比如播放一段具体的动画或在一个很强约束的行为中执行动作。可以有许多的模拟输出，比如速度就可以是 1.5 英里/小时或 157.3 英里/小时。但也可以有多层输出，能在身体的上部和下部播放不同动画的角色就是一个例子。该角色的下半部与运动有很强的联系，而其身体的上部分主要是掌握武器、瞄准或播放其他嘲弄的动画。实际上，这样是同时控制两个输出，而且它们以某种形式在角色中进行了分层处理。

(4) 确定所需的将输入和输出连接的主要逻辑。是否具有实际存在的、强健的、坚定不变的规则？是否具有一般的规则并有许多例外？或者是否根本就没有规则，而只是一些相互堆砌以表示所有逻辑的模式？所有这些在如今的游戏中都非常普遍。

(5) 确定游戏中对象之间、需要编码的 AI 系统之间以及其他游戏系统之间通信的类型。需要连续通信，还是更多事件驱动情形？是偶尔还是始终在任意特定游戏时间内从对象中取回多个消息？此类考虑将会给我们带来一系列我们需要的来自个别 AI 实体的额外需求。

(6) 注意游戏将要承受的所有其他限制。平台相关的限制是很大的一类。时间长度也是如此，当第一次着手处理 AI 项目时，它将是很难处理的一类。这里存在很多容易造成

紊乱的地方，并且 AI 工作的高层特性意味着也需要依赖团队中的其他人，让他们在工作过程中提供技术或策略资源。在考虑这些类型的限制后，我们必须要理性对待我们所能完成的工作量。同时记住，如果自己承担的任务过多，那将会发狂或崩溃。

(7) 考虑各种 AI 技术的优缺点，这在本书第III部分和第IV部分会有详细阐述。我们希望找到一些可以用于实现我们的系统的东西。如果没有找到，那可能是因为还没有对问题进行足够的分解，并试图一次处理很大一部分。试着观察所要设计的系统(或子系统)，并确保它足够地小，从而不会想方设法将太多的功能交由一个单一的 AI 技术来解决，并让它充满复杂性或例外。

然而，纯粹理论只会使我们更偏离目标。应该利用这些骨架代码在游戏中建立一些原型。或许会找到某种具体方法的缺陷，并发现很难根据所需要的级别来选择合适的技术，以及对于 AI 的一些枝节问题需要额外的元素。将这种原型作为 AI 引擎设计阶段的一部分，它将有助于找到计划中的漏洞，以及对类和结构设计上的那些冗长任务进行分解。最终的产品将会因为有它而变得更好。

2.5 小结

本章覆盖了游戏 AI 引擎中内在的基本系统，并对在设计和建造引擎时需要考虑的要点进行了描述。

- AI 引擎的三个主要部分是决策、感知和导航。
- 所使用的决策技术的类型取决于具体游戏相关的因素，如解决方案的类型、智能体的反应能力、系统的真实性、游戏类型、游戏的特殊内容、游戏平台、以及开发和娱乐限制。
- 感知系统在 AI 角色的输入数据计算上通常处于中心位置。通过保持它在中心位置，AI 系统可以防止过多的重复计算并有助于调试和开发。
- 感知系统也可以考虑低层次的细节，包括更新规则、反应时间、门限、负荷平衡以及计算代价与预处理。
- 游戏 AI 的导航系统通常属于下列 4 种样式之一：基于网格、简单避障与位势场、地图节点网络、导航网格。有些游戏则分层对它们进行组合。避障是一种处理短期目标的局部系统。
- 在设计 AI 系统时，要将整个系统分解成几个子系统，确定输入和输入类型，确定输出和输出类型，确定将输入和输出进行连接的逻辑，确定需要的通信类型，确定其他的系统限制，然后再考虑每种 AI 技术的特点。如果在系统和技术匹配上遇到麻烦，那么需要将正在处理的系统进行简化(通过细分)，或者采用另一种不同的技术会更好。
- 把对 AI 系统进行原型化当作是设计阶段的一部分，将有助于确保系统足够灵活地处理任何需要的事情，并迅速指出设计或实现中的漏洞；这些漏洞更容易在完整的产品周期开始之前确定。

3 AIsteroids: AI 试验平台

本章将介绍一个小的应用程序，即 AIsteroids，它将成为各种 AI 技术的试验平台。正如它的名字所暗示的那样，它是 AIsteroids 式游戏的简化版本，开始只具有若干礁石(由圆圈表示)、一个 AI 或人类控制的飞船(由三角形表示)、增加玩家射击力的宝物(由方块表示)。飞船可以转弯和推进(前向或逆向)，可以使用“多维空间”，并且能够射击。之后，随着展示 AI 技术的需求不断增加，我们将结合一些额外元素，比如其他不同的小飞船、不同的武器和宝物。之所以选择这个应用程序是因为它非常简单，而且各种 AI 方法在该程序中都可以很容易实现。

首先将介绍这个基本程序，包括类图以及对相关代码的清楚剖析。

图 3-1 所示是程序所用到的各种类的布局。它是一个非常扁平的结构，只有一个主要的基类 GameObj。游戏中的动态目标，如行星、炮弹、爆炸、宝物和飞船，都是 GameObj 的子类。这使得 GameSession 类(主要的游戏逻辑类)具有能够作用的 GameObj 类的完整列表。还有 3 个其他的主要文件：Aisteroids.cpp 是程序的主循环以及 GLUT(OpenGL Utility Toolkit)的初始化代码，而 utility.cpp 和 utility.h 文件则包含了一些全局函数。这些函数包括一些有用的数学函数、一些游戏相关的定义以及在 GLUT 平台下将文本在屏幕上进行显示的函数。

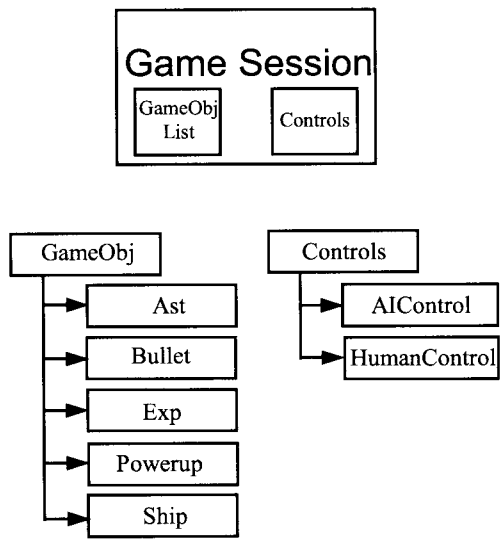


图 3-1　AIsteroids 的类结构

3.1 GameObj 类

从程序清单 3-1 可知，GameObj 类是非常直接易懂的。它对诸多元素进行了封装，包括对象设计、碰撞(包括对碰撞的检测以及在发生碰撞时需要运行的特殊代码)、基本的物理环境以及 Draw()和 Update()函数。Explode()函数负责为那些碰撞时会爆炸的对象类型产生爆炸。

注意对象类型的枚举。这里用位值枚举代替了直接的整型枚举，从而这些代码也可以将该对象类型用作碰撞标志。每个对象都必须注册它将碰撞的具体对象类型，并且这种位值表示允许一个对象注册与多个对象类型的碰撞。所有游戏对象的碰撞都通过简单的碰撞范围交集检测来处理。

另外，要注意在默认情况下，普通的 GameObj 类在碰撞时刻并不会点火、爆炸或执行任何特殊代码。该类的子类必须要重载那些成员函数以便完成各个动作。

程序清单 3-1 GameObj 类的 Header

```
class GameObj
{
public:
     //constructors/functions
     GameObj(float _size = 1);
     GameObj(const Point3f &_p, const float _angle, const Point3f &_v);
     virtual void Draw(){}
     virtual void Init();
     virtual void Update(float t);
     virtual bool IsColliding(GameObj *obj);
     virtual void DoCollision(GameObj *obj) {}
     virtual void Explode() {}

     //unit vector in facing direction
     Point3f UnitVectorFacing();
     Point3f UnitVectorVelocity();

     enum//collision flags/object types
     {
        OBJ_NONE      = 0x00000001,
        OBJ_ASTEROID = 0x00000010,
        OBJ_SHIP      = 0x00000100,
        OBJ_BULLET    = 0x00001000,
        OBJ_EXP       = 0x00010000,
        OBJ_POWERUP   = 0x00100000,
        OBJ_TARGET    = 0x01000000
     };
```

```
    //data
    Point3f         m_position;
    Point3f         m_axis;
    float           m_angle;
    Point3f         m_velocity;
    float           m_angVelocity;
    bool            m_active;
    float           m_size;
    Sphere3f        m_boundSphere;
    int             m_type;
    unsigned int    m_collisionFlags;
    int             m_lifeTimer;

};
```

3.2 GameObj 类的 Update()函数

程序清单 3-2 是基类的 Update()函数，它对基本物理参数 m_position 和 m_angle 进行更新，并使可选的 m_lifeTimer 值减小。后者是使游戏对象简单地持续一定时间，然后自动清除的一种一般性的方式。它可用于子弹、爆炸和宝物。

程序清单 3-2　基本游戏对象的 Update()函数

```
//--------------------
void GameObj::Update(float dt)
{
    m_velocity  += dt*m_accelleration;
    m_position  += dt*m_velocity;
    m_angle     += dt*m_angVelocity;
    m_angle      = CLAMPDIR180(m_angle);

    if(m_position.z() !=0.0f)
    {
       m_position.z() = 0.0f;
    }

    if(m_lifeTimer != NO_LIFE_TIMER)
    {
       m_lifeTimer -= dt;
       if(m_lifeTimer<0.0f)
          m_active=false;
    }
};
```

3.3 Ship 对象

Ship 对象是一个 GameObj 类，外加控制和发射炮弹的能力。程序清单 3-3 给出了该类的 header。一些函数构成了对小飞船的控制，而其他函数则负责发射炮弹和记录。整数 m_invincibilityTime 用于级别开始时无敌初始阶段或主飞船的再生。变量 m_shotPowerLevel 是一个影响射击力级别的宝物的累加器。如果要设计其他宝物类型，也很可能需要为它们提供结构累加器变量。Update()函数与基类只有一点点不同，Update()函数检查 m_thrust 是否为真，如果为真，则计算一个加速度，然后更新速度、位置和角度。如果还有剩余时间，该函数也对 m_invincibilityTime 进行更新。

程序清单 3-3　Ship 类的 Header

```
class Ship : public GameObj
{
public:
     //constructor/functions
     Ship();
     virtual void Draw();
     virtual void Init();
     virtual void Update(float t);
     virtual bool IsColliding(GameObj *obj);
     virtual void DoCollision(GameObj *obj);

     //ship controls
     void ThrustOn(    {m_thrust=true;}
     void ThrustOff()  {m_thrust=false;}
     void TurnLeft()   {m_angVelocity=180.0;}
     void TurnRight()  {m_angVelocity=-180.0;}
     void StopTurn()   {m_angVelocity=0.0;}
     void Stop();
     void Hyperspace();

     //Powerup Management
     virtual void GetPowerup(int powerupType);
     int GetShotLevel() {return m_shotPowerLevel;}
     int GetNumBullets(){return m_activeBulletCount;}
     void IncNumBullets(int num = 1){m_activeBulletCount+=num;}
     void MakeInvincible(float time){m_invincibilityTimer = time;}

     //bullet management
     virtual int MaxBullet();
     void TerminateBullet(){if(m_activeBulletCount > 0)
                              m_activeBulletCount--;};
     virtual void Shoot();
     virtual float GetClosestGunAngle(float angle);

     //data
```

```
    Control* m_control;
private:
    int      m_activeBulletCount;
    Point3f  m_accelleration;
    bool     m_thrust;
    bool     m_revThrust;
    int      m_shotPowerLevel;
    float    m_invincibilityTimer;
};
```

3.4 其他游戏对象

Exp(爆炸)和 Powerup 都是非常简单的对象，它们很容易实例化，并在持续一段时间后自动消失。然而，如果飞船接触到一个宝物，那么飞船的 GetPowerup()函数会被调用。行星没有最大的寿命限制，其唯一的附加逻辑是如果它们足够大，则被炮弹击中后会分裂。目标对象是用于调试的(除非因为其他原因而希望对它进行实现，比如自导引导弹)，而且仅仅是一个不存在使它自身显示为 X 的逻辑的游戏目标。

炮弹需要一个更深入的碰撞步骤，如程序清单 3-4 所示。

程序清单 3-4　炮弹专用的碰撞代码

```
void Bullet::DoCollision(GameObj *obj)
{
    //take both me and the other object out
    if(obj->m_active)
    {
        obj->Explode();
        obj->DoCollision(this);
    }

    m_active=false;
    if(m_parent)
    {
        Game.IncrementScore(ASTEROID_SCORE_VAL);
        m_parent->TerminateBullet();
    }
}
```

在这个简单的函数中，炮弹也增加得分，并调用其父类的 TerminateBullet()函数(取决于是否为该炮弹设置了飞船父类，因为炮弹可以自由实例化)，该函数只是减少飞船已经激活的射击数目。炮弹也将消灭与它碰撞的另一个对象。通用的碰撞系统只是出于优化的原因对碰撞中的第一个对象调用 Explode()和 DoCollision()函数。因此，由于需要两个对象都运行碰撞代码，因而对炮弹需要进行这种特殊的考虑。

3.5 GameSession 类

整个游戏的结构如程序清单 3-5 所示。大多数的类都是 public 类，因为无论怎样它都只是通过主要的游戏函数来进行访问。游戏被划分成少数几个高层状态：STATE_PLAY，STATE_PAUSE，STATE_NEXTWAVE 和 STATE_GAMEOVER。它们是基本的游戏流程状态并仅仅作为描绘和控制代码的修正器。对于这个示例程序来说，有两个实例化的 Control 类，即一个处理键盘事件的 HumanControl 类和一个 AIControl 类。其中 AIControl 类现在不起任何作用，但最终我们会为游戏设计 AI 代码。

程序清单 3-5 GameSession 类的 Header

```
typedef std::list<GameObj*> GameObjectList;
class GameSession
{
public:
     //constructor/functions
     GameSession();
     void Update(float dt);
     void Draw();
     void DrawLives();
     void Clip(Point3f &p);
     void PostGameObj(GameObj*obj)
          {m_activeObj.push_back(obj);}

     //game controls
     enum
     {
          CONTROL_THRUST_ON,
          CONTROL_THRUST_REVERSE,
          CONTROL_THRUST_OFF,
          CONTROL_RIGHT_ON,
          CONTROL_LEFT_ON,
          CONTROL_STOP_TURN,
          CONTROL_STOP,
          CONTROL_SHOOT,
          CONTROL_HYPERSPACE,
          CONTROL_PAUSE,
          CONTROL_AION,
          CONTROL_AIOFF
     };
     void UseControl(int control);

     //score functions
     void IncrementScore(int inc) {m_score += inc;}
     void ResetScore()            {m_score = 0;}

     //game related functions
```

```
    void StartGame();
    void StartNextWave();
    void LaunchAsteroidWave();
    void WaveOver();
    void GameOver();
    void KillShip(GameObj *ship);

    //data
      Ship*          m_mainShip;
      HumanControl*  m_humanControl;
      AIControl*     m_AIControl;

    bool   m_bonusUsed;
    int    m_screenW;
    int    m_screenH;
    int    m_spaceSize;
    float  m_respawnTimer;
    float  m_powerupTimer;
    int    m_state;
    int    m_score;
    int    m_numLives;
    int    m_waveNumber;
    int    m_numAsteroids;
    bool   m_AIOn;

    enum
    {
          STATE_PLAY,
          STATE_PAUSE,
          STATE_NEXTWAVE,
          STATE_GAMEOVER
    };

private:
    GameObjList m_activeObj;
};
```

游戏的动态对象列表存储在名为 m_activeObj 的 list 结构标准模板库(STL)中。这是一个简单程序，故在游戏中仅仅执行 new 和 delete 存储器的操作。然而，大多数的真实游戏都设法预先进行完全的内存分配(或许通过分配许多不同的 GameObj 结构，而仅在需要时才使用)，以防止内存碎片(memory fragmentation)。通过把所有的游戏对象都归入这种结构，GameSession 类的 Update()函数就变得一般化和简单化。对该函数的讨论将分 8 部分给出，从而更新中的每一部分都可以独立讨论。参见程序清单 3-6-1 到 3-6-8。

3.5.1　主逻辑与碰撞检测

程序清单 3-6-1 是更新循环的主要部分。它设置了一个 for 循环，从而在所有的游戏对象之间迭代，然后对每个对象运行其 Update()方法并调整它的位置到视见区(按照行星方

式，它也包含了该位置)。之后，该函数通过在对象间的循环并调用 IsColliding()函数来检测与其他对象的所有碰撞。这些计算通过下列方式来实现最优：

(1) 对象必须通过使它的 m_collisionFlags 变量不包含 GameObj::OBJ_NONE 位来对碰撞进行注册。

(2) 对象只对它注册过类型的对象进行碰撞检测。

(3) 对象不能与另一个不活动的对象进行碰撞。

(4) 对象不能与其自身碰撞。

程序清单 3-6-1 GameSession 类的更新循环 1：更新与碰撞检测

```
void GameSession::Update(float dt)
{
     GameObjectList::iterator list1;
     for(list1=m_activeObj.begin();list1!=m_activeObj.end();++list1)
     {
        //update logic and positions
        if((*list1)->m_active)
            {
            (*list1)->Update(dt);
            Clip((*list1)->m_position);
        }
        else continue;

        //check for collisions
        if((*list1)->m_collisionFlags != GameObj::OBJ_NONE)
        {
            GameObjectList::iterator list2;
            for(list2=m_activeObj.begin();
                list2!=m_activeObj.end();++list2)
            {
                //don't collide with yourself
                if(list1 == list2)
                    continue;


                if((*list2)->m_active  &&
                    ((*list1)->m_collisionFlags  &
                      (*list2)->m_type)  &&
                    (*list1)->IsColliding(*list2))
                {
                      (*list1)->Explode();
                      (*list1)->DoCollision((*list2));
                }
            }
        }
        if(list1==m_activeObj.end()) break;
     }//main for loop
}
```

3.5.2　对象清除

程序清单 3-6-2 给出了一些必要的代码，将那些 m_active 字段被设定为假的对象被简单地通过 remove_if STL 算子从列表中移除出去，然后擦除。检测不活动条件的算符(RemoveNotActive)也负责删除该对象占据的存储空间，而 erase 函数仅仅是把对象从列表中移除而已。

程序清单 3-6-2　GameSession 类的更新循环 2: 被杀死对象的清除

```
//get rid of inactive objects
GameObjectList::iterator end    = m_activeObj.end();
GameObjectList::iterator newEnd =
          remove_if(m_activeObj.begin(),
               m_activeObj.end(),RemoveNotActive);
if(newEnd != end)
     m_activeObj.erase(newEnd,end);
```

3.5.3　主飞船和宝物的产生

程序清单 3-6-3 和程序清单 3-6-4 是更新循环的简单部分，只是检测两个定时器：m_respawnTimer 和 m_powerupTimer。再生定时器用于主飞船被摧毁的场合，它使得在再生之前有一个很小的暂停，从而让选手有时间来意识到它已经爆炸了。在以后，当使用 AI 控制的主飞船时，我们或许需要减小这个值，因为 AI 并不需要这么一个停顿期。宝物定时器提供了每个宝物产生之间的暂停。如果这个暂停时间用完，那么游戏将以随机位置和速度产生一个宝物并把它添加到主对象列表里。

程序清单 3-6-3　GameSession 类的更新循环 3：主飞船的再生

```
//check for no main ship, respawn
     if(m_mainShip == NULL || m_respawnTimer>=0)
     {
          m_respawnTimer-=dt;
          if(m_respawnTimer<0.0f)
          {
              m_mainShip = new Ship;
              if(m_mainShip)
              {
                  PostGameObj(m_mainShip);
                  m_humanControl->SetShip(m_mainShip);
                  m_AIControl->SetShip(m_mainShip);
              }
          }
     }
```

程序清单 3-6-4 GameSession 类的更新循环 4：宝物的再生

```
//occasionally spawn a powerup
     m_powerupTimer -=dt;
     if(m_powerupTimer <0.0f)
     {
          m_powerupTimer = randflt()*6.0f + 4.0f;
          Powerup* pow = new Powerup;
          if(pow)
          {
              pow->m_position.x()= randFlt()*m_screenW;
              pow->m_position.y()= randFlt()*m_screenH;
              pow->m_position.z()= 0;
              pow->m_velocity.x()= randFlt()*40 - 20;
              pow->m_velocity.y()= randFlt()*40 - 20;
              pow->m_velocity.z()= 0;
              PostGameObj(pow);
          }
    }
```

3.5.4 奖励生命

程序清单 3-6-5 用于进行一个简单的分数检测，并每隔 10000 分时奖励玩家一条生命。这是一种非常直接的代码，并且在此类游戏中用得非常普遍。

程序清单 3-6-5 GameSession 类的更新循环 5：奖励生命

```
//check for additional life bonus each 10K points
if(m_score >= m_bonusScore)
{
     m_numLives++;
     m_bonusScore += BONUS_LIFE_SCORE;
}
```

3.5.5 级别和游戏的结束

下面的两个程序清单(3-6-6 和 3-6-7)用于检测两个重要的游戏条件：当前关卡的结束(没有行星供玩家来射击)和游戏的结束(没有生命)。每个条件调用一个函数，WaveOver() 或 GameOver()，它们设置一些临界标志并使游戏状态转移到 STATE_NEXTWAVE 或 STATE_GAMEOVER。

程序清单 3-6-6　GameSession 类的更新循环 6：关卡结束

```
//check for finished wave
     if(!m_numAsteroids)
     {
          m_waveNumber++;
          WaveOver();
     }
```

程序清单 3-6-7　GameSession 类的更新循环 7：游戏结束

```
//check for finished game, and reset
     if(!m_numLives)
          GameOver();
```

3.6 Control 类

为了给飞船发送命令，游戏需要使用Control类。其基类是一个概要结构，包括Update()、Init()以及一个指向被控飞船的 m_ship 指针。该类是人类控制系统(HumanControl 类)和AI(AIControl 类)的父类。目前，由于人类控制方案始终是相同的，故 HumanControl 类仅有一点点不同，即它不使用它的更新函数，而仅仅是程序传送给 GLUT 以执行键盘检测的全局回调函数的存放地。如果游戏更加复杂，可以为人类玩家设计一个基于状态的控制方案(或其他一些将系统功能进行划分的方案)，因此需要 Control 类的完整功能。在本书后面，当要实现各种不同的 AI 方法时，都将从设计具体的 AIControl 类开始，从而对各种 AI 方法的细节进行安排。

3.7 AI 系统钩子

在程序清单 3-6-8 中，GameSession 类检测 AI 系统是否开启，如果开启，则调用AIControl 类的 Update()函数。该更新函数在 AIControl.cpp 文件中是空的，因此 AI 系统不会有任何反应。再次强调，这仅仅是各种 AI 技术将来实现的一个框架。在后面，我们会为这个概要的 AIControl 类安排一些子类，由它们来运行各种技术的具体代码。

程序清单 3-6-8　GameSession 类的更新循环 8：AI 更新钩子

```
//update AI control, if turned on
 if(m_AIOn)
      m_AIControl->Update(dt);

//end of GameSession::Update()
```

加入到这个基类的只有一些调试数据字段，它们在本书中被用于开发该演示程序，并留下来作为各个额外调试信息的一个良好开头。必须从一开始就在设计中包含调试系统钩

子，因为我们将在开发过程中花费大量的时间来设法改变 AI 引擎，使得它简单地输出文本串或屏幕上的视觉调试元素。

存在两个更新函数：常规的 Update()函数和更新系统级数据对象的 UpdatePerceptions()函数。将这些函数独立出来，赋予了系统最大的灵活性，以及合理的组织性。图 3-2 是试验平台运行第 15 章的有限状态机(FSM)AI 系统时的屏幕截图。

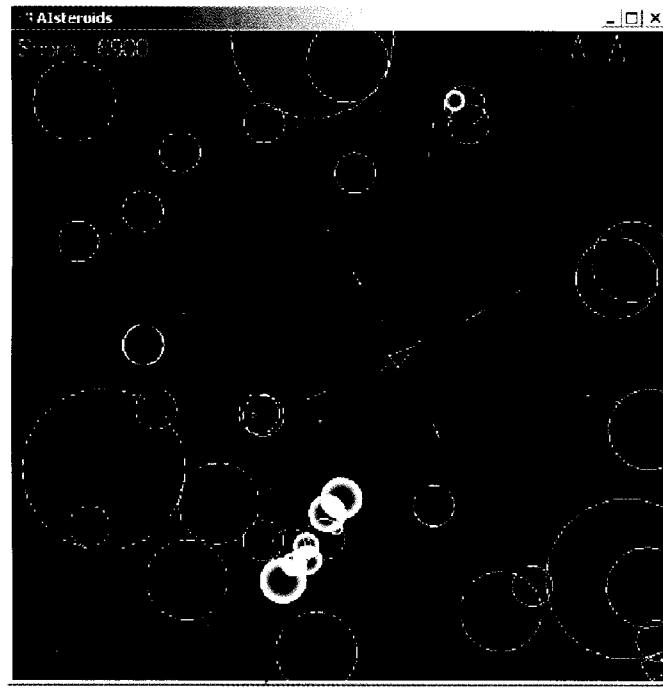


图 3-2 AIsteroids 的屏幕截图

3.8 游戏主循环

AIsteroids.cpp 是项目的主要游戏文件。它对 GLUT 进行初始化，并为游戏更新、游戏绘制和处理来自 Windows 或用户的所有输入(处理键盘命令的全局函数位于 HumanControl.cpp 文件中)设置回调函数指针。

3.9 小结

本章对第III部分和第IV部分中实现各种 AI 技术时将使用的试验平台应用程序进行了描述，并对整体类结构和代码中值得注意的部分进行了讨论。

- GameObj 类是主要的游戏对象类。它负责处理物理现象以及对象的绘制和更新。
- 游戏的当前对象包括行星、炮弹、爆炸、宝物、飞船以及一个调试目标对象。

- GameSession 类是单一游戏类。它处理运行游戏所需要的所有变量和结构。它是游戏的主要更新和绘制函数。它产生所有额外的游戏元素并对对象间的碰撞检测进行管理。
- Aisteroids.cpp 是主循环文件。它包括对 GLUT 的所有初始化以及运行游戏的所有回调函数。
- Control 类处理飞船对象的逻辑。该逻辑可以是一种 AI 技术的形式或者对于人类玩家来说是一种键盘功能。
- AIControl 类是 AI 与系统联系的分支点。通过使用一个具体的 AI 方法类(如第 15 章将要讨论的 FSMAIControl 类)对该类进行重载，我们可以对 CPU 控制的对手应用这个游戏应用程序。键盘控制仍然处于激活状态，但这只是为了方便应用程序作为一个试验平台，因为当 AI 系统运行时，我们仍然希望能够对游戏发送键盘事件。

第II部分 游戏类型

在接下来的几章中，将会对各种不同的游戏类型进行剖析，并讨论对每种游戏类型最适合的AI技术及其原因。本书将对各种游戏类型的下列几方面进行考虑：

- 需要某种形式AI的各类型游戏的元素。
- 各种游戏类型中通用的AI技术，包括最终产品的示例。各种技术的细节在本书后面几部分进行讨论。
- 真实世界中的示例。
- 值得注意的例外(如果有的话)。
- 需要改进的领域，或传统上缺失的领域。

再次强调，尽管不是包罗万象的，但下面几章内容将给出一个关于不同类型游戏的AI需求的很好的思想，以及AI程序员为了满足这些需求而经常使用的解决方案。

4 角色扮演类游戏

随着个人计算机变得越来越普及，大量游戏开始利用键盘内在的更复杂的控制方案，而不是操作杆和按钮。角色扮演类游戏(Role Playing Game，RPG)就是因为个人计算机而使其成为了一种新的游戏类型。以前，街机游戏是普遍的游戏类型，因为早期的家庭系统基本上只支持所有流行的街机游戏。为了获取更多利润，街机游戏的玩法通常被设计成很快就结束，这与花费大量时间和精力而设计的游戏很不相符。

最早的 RPG 游戏是基于文本的，或者是在 ASCII 码字符基础上加工而成。前者的例子有 Adventure 或 Wumpus，而后者则有 Rogue 和 NetHack 等。程序清单 4-1 是 NetHack 游戏的一部分代码，这里列出的函数是确定和定义来自 AI 控制的敌人的导弹攻击的一般方法。该函数考虑了不同的感知，根据基于特征和技能的判断来确定不同的情形，并做出最终的行为决策。随着图形 RPG 游戏的产生，其他 RPG 游戏便退出了历史舞台。更高性能的控制台(同时具有某种类型的存储系统，即玩家能够存储信息的卡片，或卡式磁带上的存储器)开始出现，并且控制台 RPG 游戏开始大量出现。控制台 RPG 主要是一些更具实时性的战斗游戏，因为控制台具有更多动作导向的输入控制器(尽管 Final Fantasy 系列或 Phantasy Star 游戏是基于回合制的)。并且由于控制台游戏的主要玩家一般都比计算机游戏玩家年轻，因此本质上这些游戏发展还不够成熟(主要角色通常在 15 岁左右，而不是像计算机 RPG 游戏里那样坚毅的武士)。现在，控制台 RPG 和计算机 RPG 的平台界限已经非常模糊，诸如 Diablo 游戏因为其简单、控制台式的行动导向玩法而成了计算机游戏；而控制台上新的在线持久 RPG 游戏也几乎与其个人计算机版本完全一样。

程序清单 4-1　RPG 游戏 NetHack 的部分 ASCII 开放源码

(摘自 NetHack GPL)

```
/* monster attempts ranged weapon attack against player */
Void thrwmu(mtmp)
struct monst *mtmp;
{
    struct obj *otmp, *mwep;
    xchar x, y;
    schar skill;
    int multishot;
    const char *onm;

    /* Rearranged beginning so monsters can use polearms not in a line */
```

```
if (mtmp->weapon_check == NEED_WEAPON || !MON_WEP(mtmp)) {
    mtmp->weapon_check = NEED_RANGED_WEAPON;
    /* mon_wield_item resets weapon_check as appropriate */
    if(mon_wield_item(mtmp) != 0) return;
}

/* Pick a weapon */
otmp = select_rwep(mtmp);
if (!otmp) return;

if (is_pole(otmp)) {
    int dam, hitv;

    if (dist2(mtmp->mx, mtmp->my, mtmp->mux, mtmp->muy) >
            POLE_LIM || !couldsee(mtmp->mx, mtmp->my))
    return; /* Out of range, or intervening wall */

    if (canseemon(mtmp)) {
    onm = xname(otmp);
    pline("%s thrusts %s.", Monnam(mtmp),
          obj_is_pname(otmp) ? the(onm) : an(onm));
    }

    dam = dmgval(otmp, &youmonst);
    hitv = 3 - distmin(u.ux,u.uy, mtmp->mx,mtmp->my);
    if (hitv < -4) hitv = -4;
    if (bigmonst(youmonst.data)) hitv++;
    hitv += 8 + otmp->spe;
    if (dam < 1) dam = 1;

    (void) thitu(hitv, dam, otmp, (char *)0);
    stop_occupation();
    return;
}

x = mtmp->mx;
y = mtmp->my;
/* If you are coming toward the monster, the monster
 * should try to soften you up with missiles. If you are
 * going away, you are probably hurt or running. Give
 * chase, but if you are getting too far away, throw.
 */
if (!lined_up(mtmp) || (URETREATING(x,y) &&
       rn2(BOLT_LIM - distmin(x,y,mtmp->mux,mtmp->muy))))
    return;

skill = objects[otmp->otyp].oc_skill;
mwep = MON_WEP(mtmp); /* wielded weapon */

/* Multishot calculations */
```

```
multishot = 1;
if ((ammo_and_launcher(otmp, mwep) || skill == P_DAGGER ||
    skill == -P_DART || skill == -P_SHURIKEN) && !mtmp->mconf) {
    /* Assumes lords are skilled, princes are expert */
    if (is_prince(mtmp->data)) multishot += 2;
    else if (is_lord(mtmp->data)) multishot++;

    switch (monsndx(mtmp->data)) {
    case PM_RANGER:
        multishot++;
        break;
    case PM_ROGUE:
        if (skill == P_DAGGER) multishot++;
        break;
    case PM_NINJA:
    case PM_SAMURAI:
        if (otmp->otyp == YA && mwep && mwep->otyp == YUMI) multishot++;
        break;
    default:
    break;
    }
    /* racial bonus */
    if ((is_elf(mtmp->data) && otmp->otyp == ELVEN_ARROW &&
        mwep && mwep->otyp == ELVEN_BOW) || (is_orc(mtmp->data) &&
        otmp->otyp == ORCISH_ARROW && mwep && mwep->otyp == ORCISH_BOW))
    multishot++;

    if ((long)multishot > otmp->quan)
            multishot = (int)otmp->quan;
    if (multishot < 1) multishot = 1;
    else multishot = rnd(multishot);
}

if (canseemon(mtmp))
    {
    char onmbuf[BUFSZ];

    if (multishot > 1)
    {
    /* "N arrows"; multishot > 1 implies otmp->quan > 1, so
        xname()'s result will already be pluralized */
    Sprintf(onmbuf, "%d %s", multishot, xname(otmp));
    onm = onmbuf;
    }
    else {
    /* "an arrow" */
    onm = singular(otmp, xname);
    onm = obj_is_pname(otmp) ? the(onm) : an(onm);
    }
    m_shot.s = ammo_and_launcher(otmp,mwep) ? TRUE : FALSE;
```

```
            pline("%s %s %s!", Monnam(mtmp),
              m_shot.s ? "shoots" : "throws", onm);
            m_shot.o = otmp->otyp;
        }
        else
        {
            m_shot.o = STRANGE_OBJECT;
                /* don't give multishot feedback */
        }

        m_shot.n = multishot;
        for (m_shot.i = 1; m_shot.i <= m_shot.n; m_shot.i++)
            m_throw(mtmp, mtmp->mx, mtmp->my, sgn(tbx), sgn(tby),
                distmin(mtmp->mx, mtmp->my, mtmp->mux, mtmp->muy), otmp);
                        m_shot.n = m_shot.i = 0;
        m_shot.o = STRANGE_OBJECT;
        m_shot.s = FALSE;

        nomul(0);
    }
```

RPG 游戏一般遵循一些简单的规则：从一无所有开始，为了财宝或金钱而执行任务(主要是杀死怪物和继续探险)，训练技能，并最终成为一个具有强大力量的人物，然后修正地球上的终极错误。有些游戏包含了一群冒险家，因此玩家实际上是在构造一整队角色。无论技术细节如何，游戏都是大同小异的：让玩家辨别主要的角色，并投入大量时间去构造这么个角色，最终完成该游戏。

大多数 RPG 游戏中充满敌人并持续敌对的世界或许看起来非常奇怪，但十几岁的青少年就不会有这样的想法。在某种程度上，年轻人更容易与那些孤独的、反对任何人的、受到普遍误解和攻击的角色发生联系。正是这点使得 RPG 游戏对那些年轻玩家具有非常大的吸引力。一小撮队员的加入极大地促进了大多数年轻人的小集团世界的形成。在那里，可以召集一群热情的朋友，并将“我与世界对抗”的战斗扩展到那些人之中。这并不是说年长的或更年轻的人就不能享受 RPG 游戏的乐趣，而是说明此类游戏很受欢迎的一个理论原因。

RPG 游戏非常依赖 AI，这主要是因为它们通常是非常庞大的游戏，具有很多种不同类型的玩法，并且每一个头衔都需要很多小时的游戏体验。同样，多变游戏元素的智能应该比大多数都要高(至少需要更多的脚本来描述)，因为人们花费在游戏中的总时间将使得任何的行为重复都变得更明显，并使得 AI 行为中的细微困扰显得更大。另外，特别是在家庭计算机中，用户至少需要大约 40 小时从 RPG 游戏中获取玩法体验。控制台则稍微少一些，通常是 20~40 小时。这在游戏玩家心中似乎是个确定的规则(游戏能够维持玩家兴趣的大概时间与关于能从成本中获取多大的玩法体验的营销教育的一个奇怪混合)，但也有例外，比如针对 PC 的 Baldur's Gate 游戏就有超过 100 小时的玩法体验。

由于人们对游戏玩法具有如此巨大的数量需求，因此游戏最好能够具有多种多样的玩法类型(比如谜题、战斗、某种类型的技能、不同类型的旅行等)或者主要战斗系统最好能够非常有趣并让人上瘾。Diablo 游戏属于后面这一类。其玩法非常具有重复性，但也很容

易让人上瘾。有人已经从理论上说明游戏唤醒了我们内在的“hunter-gatherer”本性，并且我们只是情不自禁地喜欢上了电脑游戏。

4.1 通用 AI 元素

4.1.1 敌人

RPG 游戏中最多的是敌人(enemy)。游戏需要几乎无止境的敌人，从而使得玩家有东西可以派遣，并获取经验值、金钱和其他一些强有力的新道具。以前的 RPG 游戏几乎毫无例外地使用一些可被描述为统计 AI 的东西，因为怪物的特征决定了关于它们的一切，比如它们使用的攻击手段、它们格斗的方式、它们的能力、当它们死时会出现什么财宝等。现在的游戏比以前更进了一步，并通常具有能够自我适应的敌人。这些敌人也使用一些较复杂的行为模式，包括逃跑、自我康复、包围玩家进行分组格斗并使用互补的攻击方法等。

由于在 RPG 游戏中敌人往往大量出现，因此具体构造的 AI 更多的是“人工”，而不是“智能”。过去的回合制 RPG 游戏(如 Bard's Tale、Phantasy Star 和 Chrono Trigger)，所谓实时战斗 RPG 游戏(如 Legend of Zelda、随后的 Ultima™游戏、Diablo 和 Terranigma)，以及近来相互融合的变种(如 Baldur's Gate 或 Icewind Dale，它们是实时游戏，但能够暂停，因此也可以是回合制游戏)，它们都非常依赖于充当组合容器(财富和经验值)和障碍物(通过设置一定数量击打值的“墙”，英雄只有摧毁它们才能通行)的敌人。

当然，这都是通过设计来实现的。当一个玩家花费 60 小时或更多的时间终于走入一个房间，并看到一个与他以前见过的敌人角色很相像的怪物拦在眼前时，他应该有如下三种想法之一：

- **我能打败这个家伙**。我知道他所使用的攻击、他大概的击打值以及我拥有的能够克制这个敌人的武器。
- **我想我能打败这个家伙**。他跟我以前对抗过的敌人很相像，但他有不同的颜色或特殊的名字，这使得他不同寻常并很有可能更先进。事实上，我相信他属于我见过的敌人类型，但我不确定他到底有多强。
- **我不能打败这个家伙**。他太强了，或者我不具备穿透他的装甲的武器。我之所以知道是因为我以前尝试过但失败了，或者游戏中的别人曾经警告过我。

这仅仅是使玩家沉浸到游戏之中并让他感觉到他是世界一部分的另一种方式而已，因为他能够凭经验了解敌人。如果一个低级的 Orc 突然拔出手榴弹(在近 50 次遭遇战中都是无用地助跑并使用迟钝的匕首)并对玩家实施攻击，那么玩家将会有受到欺骗的感觉。然而，如果允许玩家随时保存游戏，或游戏事实上就以很高的频率自动保存，这个基本步骤就有可能回避。在这种方式下，与一个特殊敌人的一次很不寻常的遭遇或许会杀死玩家，但如果他已经保存了游戏，他将不会损失任何的时间。是的，这将导致玩家出现更多的“保存，然后跑到拐角处杀死一个怪物，然后再保存”行为，但这将使设计者有更多的自由，从而将更多令人惊奇的元素添加到随机遭遇战中。

4.1.2 头目

头目(boss)是更大更复杂的游戏角色。它要么是类人的，要么是动物，当在每个关卡末端击败一群次要敌人后便会出现。它们通常相当于怪物领袖——怪物的国王。它们是特殊的、通常是打破以前所有规则的独特的敌人。我们希望玩家会因为这些家伙使用的力量、技能、武器等东西而感到惊讶。头目甚至被认为是RPG游戏世界的乐趣来源，并且一个好的头目能够补偿很多的游戏缺陷，不管是在一般玩法领域，还是仅仅为了能够在游戏世界中继续前进而必须经过的一段单调乏味的闯关。

因此，通常需要精心设计怪物头目，并使它们具有只有它们才有的特殊攻击和行为。怪物头目还通常以构思前进信息或纯粹恶言谩骂的形式来与玩家进行沟通。因此为它们设计的AI需要包含对话系统。Final Fantasy系列游戏的怪物头目具有非常巧妙的特殊编码，其遭遇战可能需要好几小时的真实时间，连同不同的搏斗和交谈阶段。开发人员对这些遭遇战进行了严格的调整，使得最初占据优势时按计划炮弹群发，以及随后敌人占据优势时，对其他敌人、特殊游戏事件或其他一些他们能够想到的东西进行脚本化中断。

另一个经实践证明可行的头目策略涉及到“到现在为止还不能杀死”的头目。这包括一个接近死亡边缘却奇迹般地逃脱并大叫“我还会回来的!”并且发誓下次更强大和更坏的头目。尽管有些陈旧，但这是简单角色发展过程的一种尝试，其坏人具有与玩家差不多的游戏历程。

有的游戏使用“隐藏头目(sub-boss)”，从而使得游戏中的怪物更具层次性，尽管它们仅仅是普通动物的较强者，就像Diablo系列中经常出现的独特动物一样。但就算是许多人认为是准RPG游戏的Diablo游戏，也有更多专门的动物头目来使用额外的对话、动画、拼写和武器效果以及特殊能力。

头目还包括玩家需要击败才能赢得游戏的最后动物(巫师、上帝、邪恶制造者)，也叫做终极头目(End Boss)。这个角色的确非常重要，许多很好的游戏都因为具有令人失望或滑稽可笑的终极头目而受到人们不好的评价。玩家需要执行许多他在游戏中学到的窍门，并增强他的技能以达到摧毁终极头目的能力，而且终极头目本身也需要能够做一些玩家在游戏中从来没有见过的东西。当然，从统计学观点看，终极头目应该更强，但它也应该具备一些超越典型的行为。这也是为什么称它为终极头目的首要原因。

4.1.3 非玩家角色

非玩家角色(Nonplayer Character，NPC)被定义为“游戏中非人类玩家的任何人”。然而，NPC这个术语通常是指游戏中人类能够通过除了战斗以外的任何方式相互作用的人。NPC是那些居住在市镇里的人，是玩家找到的能够为玩家提供“前面危险”重要线索的垂死的士兵，是玩家偶然遇到的给玩家钱并让玩家营救他女儿的老人。他们通常要么是一闪即逝，如涉及到某些任务的人；要么是信息倾销站，从而玩家能够在游戏的不同时刻与他们交流，并且他们或许知道现在游戏流程中“新”东西的一些额外信息。

无论如何，NPC通常都不是智能的，也不必是智能的。除了信息和故事进展以外，它们所提供的只是让游戏变得更具风味。然而，它们也是玩家与关于故事情节进程的信息进行交互的最大部分之一，同时也是游戏内置的帮助，即使得那些被阻塞或迷路的玩家返回

到接近游戏目标的状态上来。因此，许多游戏都曾尝试过处理 NPC 交谈的不同方式。有的游戏给玩家提供一些代表给 NPC 的问题的关键字(比如在 Ultima 游戏中)，有的则让玩家对代表对 NPC 不同态度的句子进行选择(如最近的 PS2 游戏 Baldur's Gate)。随着语法系统变得更好、更快以及更为广泛接受，这些系统将会继续得到发展。将来，玩家或许可以直接与一个通用的 AI NPC 进行交谈(通过说话)，而 NPC 则将通过对它的知识库进行索引或巧妙地构成语句来对我们提供广泛的响应。到那时，我们就做到了我们能够做的事情。

4.1.4　店员

店员(shopkeeper)是与玩家做交易(比如买卖装备、教玩家新的技巧等)的特殊 NPC。店员通常没有一般的 NPC 那么聪明，之所以着重对他们进行描述是因为他们通常具有扩展的接口，从而需要特殊的代码使他们看起来比较智能和便于使用。有时，店员是脚本化任务或游戏序列中的一部分，因为只有在游戏后期或某一项任务被完成之后他们才能变成店员。因此，一个店员可以具有是否喜欢玩家的观念，该观念将影响到他的态度、价格以及何时与玩家进行交易。有的游戏对其中的每个角色都安排有一个一般的魅力属性(或派生属性，其意思是鉴于第一印象、外表以及交谈能力，别人对角色的自然认识)、某种类型的描述角色所具有的优缺点数量的名望系统以及表示 NPC 能够注意到并对之响应的角色所做的具体事情的标志。

赋予无生命物体一些人类的特性，这是人类的一个自然趋势，这种趋势与我们处理该物体必须花费的时间直接相关，并与这些物体需要我们付出的成本有关。很少有人赋予他们的鞋子一些人类特性，但大多数人都为他们的汽车起名，知道它们的性别，知道如何区分它是否具有糟糕的一天，甚至当它运行不好的时候还会为它辩解。这两个对象(鞋子和汽车)基本上具有相同的功能：保护我们的身体免受旅行的折磨。但为什么如此不同？答案非常明显。根据我们 3 岁时就已经学会的简单程序，不需要做任何改变，在早上我们穿上鞋子，之后就忘记了它们，而且买一双新鞋并不需要信用支票。汽车则完全相反。同样的情况对店员 AI 来说也是一样的。如果在游戏中有一闪即逝的 NPC，则可以按自己对该行为、对话以及与玩家的交互所希望的进行适当处理。玩家不会有太多的期望，并且对于大多数事情也只是一面之缘。但对于店员，特别是玩家在游戏的大部分情况下都将用到的店员，则每个细微差别、回答以及动画框架都得仔细观察、记忆并进行人性化处理。游戏是否具有物品交换系统(它实际上将根据玩家的魅力属性高低，加上一个随机因素，来决定玩家能够享受到的一个小小折扣)？随着时间的流逝，人类玩家将开始设想一些复杂的规则，包括他要交易道具的顺序、时间、店员的心情以及许多其他不存在的因素。正是这种人类趋势使得游戏制作者可以避开游戏中很小的细节，因为人类会在本来不复杂的地方充满所有的复杂性。给我们的启发是店员不仅仅给玩家提供一个系统的接口，他们同时也增加了世界的丰富性并给玩家提供游戏中需要考虑的一些其他方面的东西。

4.1.5　队员

冒险队伍中的成员也是特殊的 NPC，除了他们随玩家旅行，并且要么是完全由玩家控制(在回合制 RPG 游戏中，或者其后的一些允许暂停从而有时间给出详细命令的游戏)，要么是具有一些 AI 代码。需要对这些基于 AI 的队员进行仔细编码，因为愚蠢的队员将很快

赶跑那些潜在的玩家。很多的实时战斗游戏使用简单的队伍 AI，因此玩家可以预测(并依靠)对抗中每个队员将采取的动作。在实时战斗 RPG 游戏中需要记住的另一个要素是路径搜索。在基于回合制的战斗系统中，队员仅仅是依附于玩家，或者是直接跟随在玩家左右(比如 Final Fantasy 游戏，或早期的 Bard's Tale 游戏)，但在实时游戏中，他们实际上必须要寻找路径来对玩家进行跟随。如果玩家处于半包围空间(例如在地牢中)，这将带来麻烦，因为他们不能通过某个队员来进行解救(该队员无法采用路径搜索器设法找到的冗长路线)。对玩家来说这是非常令人沮丧的事情，而且它将导致那些不正确的队员穿过地图上其他部分的大量怪物，并让他们跟随“帮”玩家的朋友跑进房间来参与战斗。在同样的情形下，智能的队员可能会说：“Hmm，我不能直接摆脱那个家伙来使用我的剑。但我包裹里有弓箭，而且我的箭术还不错，或许我将尝试那样攻击。”或者，“我不能直接避开，所以不能进行攻击。或许我应该帮被动物所伤的较弱伙伴一把，并顶替他上前线。”此类聪明(而不是盲目地进行路径搜索和脚本跟随)正是队员和需要玩家照顾的其他伙伴之间的差别。

如果冒险角色通常很糟糕，做错误的事情，或者经常被自己或玩家杀死，那么玩家将不会希望继续与他玩下去。Baldur's Gate 游戏(及其派生版本)甚至允许用户编辑简单的管理队员 AI 的脚本，从而用户更加能够控制这个至关重要的游戏元素。可以参见 4.2.1“脚本”一节。

加入一个脚本系统来编辑队伍 AI 需要进行仔细的权衡。如果使得它很容易使用，而不提供足够的复杂性和功能性，那么它并无价值。但如果该系统太过强大，那么它又彻底压制住了那些偶尔的游戏玩家，从而对大部分的游戏玩家来说也没有价值。许多运动类游戏采用的允许玩家调整游戏中 AI 的一个方法是，将行为的具体“趋势”由玩家可以设置的“滑块”(与变量关联的滚动条)来进行表示。对运动类游戏来说，这意味着玩家可以通过设置滑块到某个位置，使得篮球游戏中 AI 不会设法去抢球，也不会防守，并很善于投三分球。也可以使用一个相似的系统来使得更多偶然的游戏玩家能够进行 AI 编辑，而不需要编写任何脚本代码。甚至脚本系统中的一些复杂性，如确定 AI 魔术师角色何时施加特殊的咒语，也可以表示成与那个咒语相关的滑块。这的确转变成了许多基本的滑块，但再次强调，它比脚本文件更容易让大部分玩家接受。

4.2 有用的 AI 技术

4.2.1 脚本

大多数 RPG 游戏都大量使用脚本，因为这些游戏中的大多数都遵循一个非常具体的故事情节。脚本可应用于多种游戏结构，包括对话、游戏事件标志、特殊敌人或 NPC 行为、玩家如何与环境交互以及许多其他结构。之所以使用脚本是因为大多数 RPG 游戏都是线性的，或至多是分支线性的，因此，它们可与脚本化界面很好地结合。可以将游戏的某部分设计成几乎跟设计一样结束，将阻塞点和标志嵌入到脚本中，从而强迫玩家跟随游戏流程从 A 点到 B 点，即使他们在此期间首先到了 C、D、E 和 F 点。另外，许多 RPG 游戏的交谈特性也有助于这项技术。可以把脚本想象成一些基于整个故事中出现的多种事件的硬编

码数据。程序清单 4-2 是 Black Isle 游戏 Baldur's Gate 中的一小段脚本的一个示例。从中可以看到一个非常基本的攻击脚本，它简单地根据敌人离玩家的距离决定是否对敌人实施攻击，然后同时决定是否使用远程或混战武器。它要进行感知检测(计算范围)和感知调度(指定脚本多长时间进行重复)。它也具有一定的随机性，因为采用远程还是近距离作战是由一个随机数决定的(在 33%的时间中它选择混战，而在其他时间内则选择远程作战)。

程序清单 4-2　Baldur's Gate 中用户定义脚本的战士 AI 示例

```
IF
     // If my nearest enemy is not within 3

     !Range(NearestEnemyOf(Myself),3)

     // and is within 8

     Range(NearestEnemyOf(Myself),8)
THEN
     // 1/3 of the time

     RESPONSE #40
        // Equip my best melee weapon
        EquipMostDamagingMelee()
           // and attack my nearest enemy, checking every 60 ticks
           // to make sure he is still the nearest
        AttackReevalutate(NearestEnemyOf(Myself),60)

        // 2/3 of the time

     RESPONSE #80

        // Equip a ranged weapon

        EquipRanged()

        // and attack my nearest enemy, checking every 30 ticks
           // to make sure he is still the nearest

        AttackReevalutate(NearestEnemyOf(Myself),30)
END
```

4.2.2　有限状态机

有限状态机(Finite-State Machine，FSM)是游戏开发中大量使用的 AI 元素。正如 FSM 在任何游戏中都非常有用一样，它们在 RPG 游戏中也非常有用。它们允许开发人员将游戏划分成清晰的状态——在每个状态中特定角色将执行不同的任务——并将它们按离散代码块进行管理。因此，可以拥有一个 NPC，它第一次遇见玩家便给玩家一个任务(例如，遇见玩家之前的状态为 state_into，在给了玩家关于任务的信息后变成 state_quest)，当玩家完

成该任务之后，它便变成一个店员并以一定折扣卖给玩家东西以示奖励(state_shopkeep)。要注意，如果敌人近在咫尺，Baldur's Gate 中的脚本何时才可以应用。任何其他游戏状态都需要额外的脚本，或者它简单地跟随默认脚本(很可能是跟随在人类身边)。

通过拥有一个基于状态的系统，但通过脚本来进入和退出到那些状态，许多 RPG 游戏都掩饰了那些困难的状态转换。其他游戏则不然，如 Nintendo 经典的 The Legend of Zelda 游戏，它被划分成两个截然不同的状态：上流社会和下层社会的地牢。游戏的音乐将改变，角色本身也会有一点不同的表现(由于照明的原因)，并且如果玩家死了，游戏也会有一点不同(如果愿意，可继续呆在这个地牢中)，所有这些都是因为该基本状态的改变。

4.2.3 消息

由于 RPG 游戏世界中有如此众多的元素，实体之间的通信需求是非常高的，因此，在这类游戏类型中消息系统显得非常有用。信息可以在队员之间迅速和轻易地传递，从而方便了组队战斗或对话。钥匙，或其他游戏中使用的东西，可以使锁打开；当不合适的石墙被推倒时会导致一系列事件的发生。由于在 RPG 游戏中消息系统的绝对使用数量非常大，它能够真正地给用户带来实现上的灵活性和便利性。

需要注意一件事情，因为它打破了以某些形式表现出来的幻想，即游戏使用的“即时”消息。玩家的队伍在世界远端消灭了某些动物。于是他们把消息远距离传输到市镇上(凭借一个特殊的魔法道具)，从而每个回到市镇里的人都已经知道这场战斗，知道玩家赢了，知道玩家是个英雄。市镇上的人明显接收到了该消息并为此转变到了游戏状态的特殊行为。如果与玩家交谈的第一个人并不知道(除非玩家长途跋涉回家，并给每个人时间让他们自己去发现)，而且玩家必须要告诉他，这样是不是会有更好的反应？那时，那个家伙是不是会跑进街道并散布这个好消息？是的，把消息机制放到游戏之中，并用它来设置改变游戏行为的游戏标志。但不要使用过度，或通过允许游戏状态以不可能发生的方式即时改变来滥用该系统。如果该市镇的市长自身具有法力，能够通过他的水晶球看到所有发生的事情，那是另外一种不同的情况，并且需要这样进行描述。

4.3 示例

像 Wizardy、早期的 Ultimas、Phantasy Star、Might and Magic 以及 Bard's Tale 等经典游戏主要使用基于统计的敌人，很少有特殊的事件代码。它们通常也采用谜题系统(通过使用诸如钥匙、宝石或 Skull of Muldard 等)，且必须在合适的时间和在合适的地点找到并使用它。这与对系统标志进行编码非常相像，这些标志意味着游戏元素有权决定游戏进展的细节。

通常，这些游戏的玩法图应该包括一个市镇状态、一个旅行状态和一个战斗状态。这些状态对大多数该类型游戏的核心体验进行了分割，并且这些游戏之间的区别通常是整个游戏的图形质量、玩家如何与 NPC 交谈以及战斗界面。

最奇怪的是，如今一些大规模的多人在线 RPG 游戏(MMORPG)也使用这种游戏样式来产生人们可以游戏的庞大世界。当然，唯一增加的真实玩法是大量的玩家同时玩这个游戏，从而产生更多的玩家对玩家的交互。

更多现代 RPG 游戏，比如后来的 Final Fantasy 游戏、Neverwinter Nights、Baldur's Gate 和 System Shock 等，都使用了更加脚本化的事件。这些游戏具有一些基于属性的敌人，但在路途上也有很大一部分手工定制的遭遇战和环境，从而给玩家一个更具技巧性的玩法体验。直到最近在线 RPG 游戏才开始尝试这种策略(如 Final Fantasy 在线游戏)，因为在任何时候在有成千上万的人居住的世界中设计一个定制任务和遭遇战需要结合大量的工作。但是，由于有更高质量游戏内容的需求，游戏公司必须要提供这个功能。

4.4　例外

Bethesda 软件公司制作了优秀的 Elder Scrolls 系列 RPG 游戏(图 4-1 是该系列的第一个游戏 Elder Scrolls：Arena 的屏幕截图)，它宣称其为“开放的”，即玩家可以完成这个游戏并以一种非线性方式执行各种不同的任务。该游戏的确比其他任何的 RPG 游戏都提供了更大的自由度。游戏给了我们很大的自由，对我们接收到的任务不做任何时间限制(或很少)，因此我们可以到处收集任务并以任意次序执行。任务仍然主要是脚本化的(对不同角色和位置使用大量的任务类型来作为模板)，并且通常具有相当简单的特性，从而便于实现(尽管该系列中较新的游戏已经极大地改进了任务的种类和复杂性)。主任务仍然是线性的，并由脚本化的有独特的 NPC 的遭遇战来帮助实现，但它也允许玩家花时间去完成其他的副任务。

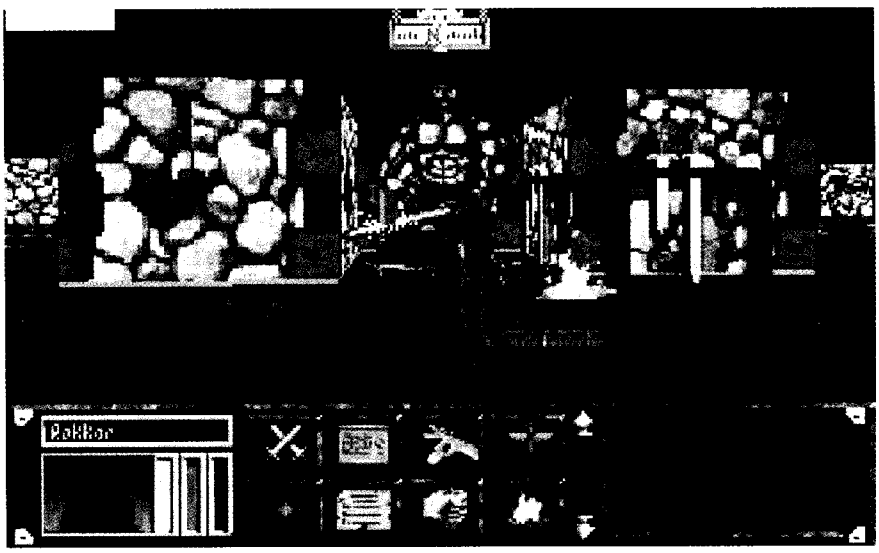


图 4-1　Elder Scrolls: Arena 屏幕截图

(© 1993，Bethesda Softworks LLC，ZeniMax Media 公司)

Neverwinter Nights 是最近另一个被认为是有巨大变化的游戏。通过允许玩家控制游戏中的一个角色并实际充当 Dungeon Master 角色(借用纸笔世界中的概念)，该游戏被认为是完全使用在计算机上的 Dungeons and Dragons(D&D)。就它成功的程度来说，在很多方面它真正展现的就是一般人难以设计得很好的游戏内容。补丁可以解决其中的一些问题，但如果游戏不是为长久寿命而设计的，那么该游戏也就毫无意义。因此，这的确需要改变，并且更好的模块将会在网络上出现。

4.5 需要改进的具体游戏元素

4.5.1 角色扮演不等于战斗

在多数 RPG 游戏中，从一开始，用在“角色扮演”上的大多数时间实际上都用在了屠杀上。这主要是受两方面的影响：真正的老式纸笔 RPG 游戏(比如 Dungeons and Dragons，以及比它更早的 Chainmail)就集中在这个方面；战斗相比实际拥有绘图、角色和戏剧场景等的故事来说更容易模仿和设计。

考虑这种情况：大多数 RPG 游戏中的非杀手类只在一小部分精心设计的环境中有用，而且设计人员设计这些环境只是为了验证这些类而已。小偷是有问题的更经典的类型之一，甚至在纸张游戏 D&D 中也是一样。如果允许小偷真正做他们做的事情，那么他们将会过于强大(正如真实生活一样，Mafia 比警官更加强大，至少在当地是这样)，因为他们不需要遵守其他人需要遵守的规则，所以游戏对他们进行了约束。小偷可以解除陷阱，也可以当扒手。但是，如果陷阱解除错误，他们一般会死亡；如果扒东西不成功，则通常将永远被监禁。从这里我们看不到一点点乐趣。也想想一般 MMORPG 游戏中大量的“你能从中选择的强大职业”。在 Ultima Online 游戏中，如果愿意可以当一名面包师。但遗憾的是，必须要花费数月的时间来玩这个游戏，才能成为 Master Baker，即一个真正的 King of Baking，但在走出市镇的那一刻就会马上被一个低级的战士用一把生锈的勺子给杀死。在如今的 MMORPG 游戏中，人们倾向于成为坦克(具有大量健康值和盔甲值的战士类型，或能够抵挡伤害的人墙)，或者投手(站在坦克后面，要么能以咒语破坏动物，要么能够修复坦克从而使其继续发挥威力的人)。专业类差不多都融入到这两种基本的群体中。

RPG 游戏世界中具有内在的大量强迫性玩法，但需要对设计那些不涉及屠杀和利用非致命技巧的游戏内容的方式进行富有意义的思考，而不仅仅是影响购买新式武器的价格。这里涉及到的任务并不简单，而且编写 AI 代码来支持这些新的任务类型将会更难。但毫无疑问，我们的 RPG 游戏会因此而变得更好。

4.5.2 语法机器

语法机器(Grammar Machine，GM)用于产生更好的会话引擎。在 RPG 游戏中，大量与其他角色的交互是通过会话来完成的，通常是以从一系列的响应中进行选择的方式，然后读取角色的脚本化响应。Ultima 游戏使用一个关键字系统，因此玩家可以说“小偷”，另

一个角色便会告诉玩家关于当地的小偷的事情，并在最后提到一个叫做 Blue 的人对他们进行控制。一个新的关键字“Blue”便会出现在列表中，并且可以以同样的方式寻求一些额外信息。老式的文本冒险类游戏实际上具有初步的能够处理不太复杂的句子的语法引擎。在现代 RPG 游戏中还没有设计出一种能够用于与 NPC 会话的完全功能的语法系统。随着越来越好的语音识别软件的出现，这种情况或许会有所改变。最终，RPG 游戏将使用这种系统代替那种缓慢且笨拙的文本界面，从而让用户真正地询问问题。到那时，我们作为 AI 程序员的工作将是去完全充实语法引擎，并用足够多的知识来填充文本数据库，以便负责任地回答用户提出的问题。

4.5.3　任务发生器

对于开发人员来说，真正的任务就是设计一个不会产生派生或重复内容的任务发生器。对于诸如 Holy Grail 等的大规模 RPG 游戏来说，一个任务发生器应该能够产生玩家可以应付的新任务，而不需要游戏设计者进行明确指定和脚本化。像 Everquest 之类昼夜不停的在线游戏，能够从一个能产生任意数量队员和任意技能级别的新奇挑战的系统中获得很大好处。到现在为止，只有一小部分游戏具有“随机”任务，并且它们通常都属于“fedex”任务领域。也就是说，去寻找这个家伙，从他那里获取一些东西，并把它带回给玩家。一个改进或许是在系统中创建即兴表演方式，使用模板来产生包括多个角色、位置、奖励和需要完成的不同行动的自定义任务(或一串相互连接的任务)。这些模板与基本的即兴表演方式和位置的数据库相连，以及与针对技能级别等的得分任务方式相连，使得 RPG 游戏具有真正独特的游戏体验(至少对于副任务交互来说)。游戏甚至可以知道玩家喜欢哪些任务(通过对被拒绝的任务类型，或从未完成的任务类型与成功以及重复类型的对比进行记录)，并针对特定玩家调整任务类型。另外，通过使即兴表演机器具有可扩展性，可以连续地添加游戏内容(针对在线游戏，或者通过修改、打补丁或扩展包来针对单个产品)，即兴表演系统会把它并入到游戏之中。

4.5.4　更好的队员 AI

能够以隐式或显式的方式扩展和修改的队伍 AI 是另一个需要关注的重要领域。早期的实时 RPG 游戏(比如图 4-2 描述的 Ultima 7 游戏)具有一些简单的队伍 AI，他们只是跟随玩家在地图上四处走动，并在战斗中设法对玩家提供帮助。Baldur's Gate 游戏对实时 RPG 游戏的队伍 AI 做出了很大的贡献，使其成为优先考虑的事项。在他们简单的脚本形式内部能够完成的调整级别是非常让人惊讶的，但它还应该能够做得更好。角色能够了解玩家让角色执行的各种动作，并让它们自动执行。

可以把这想象成简单的模仿学习。玩家是否经常撤退某一个角色(比如很弱的魔法师)？在手动操作两三次后，这个魔法师会自动地撤退。玩家是否在其健康值只剩三分之一的时候就饮用健康剂，而不是在战斗结束或从当前的危险中逃脱之后才饮用？角色应该能够感知到这些，并模仿这些简单的行为。

设想一下玩家的游戏体验是如何随着游戏进程而发展变化的，而不是在数小时的玩法中一次又一次地维持那些单调的行动。甚至可以向玩家给出学习过的行为列表，并允许玩家通过删除某些行为或改变这些行为的优先级来对这个列表进行编辑。

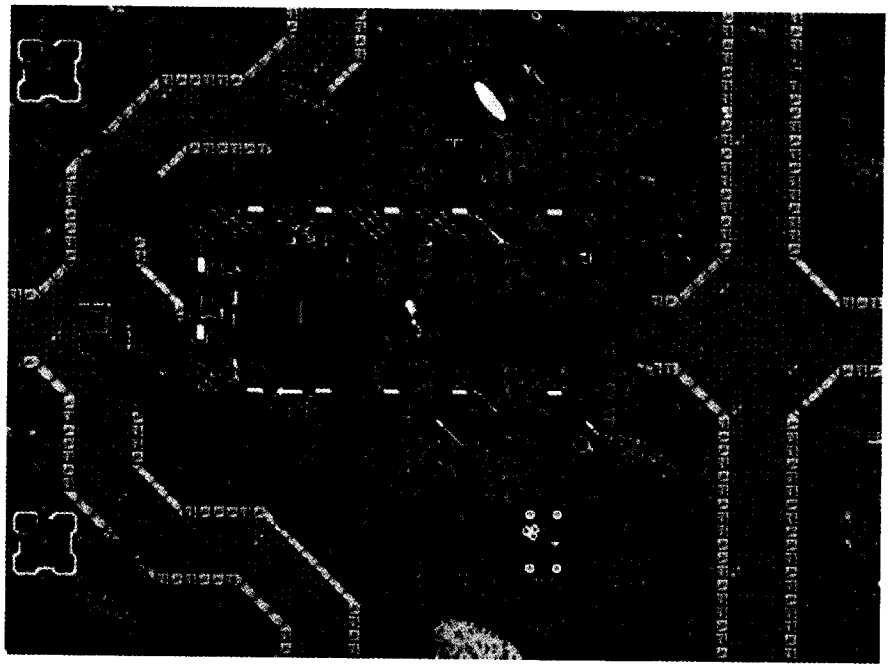


图 4-2 Ultima™ 7 屏幕截图

(© 2004，Electronic Arts Inc。Populous、SimCity、SimCity 2000、SimAnt、SimEarth、SimFarm、Dungeon Keeper、The Sims 和 Ultima 是 Electronic Arts Inc 在美国和/或其他国家中的商标或注册商标，版权所有)

4.5.5 更好的敌人

游戏中的敌人不应该是乌合之众，它们应该要实现如下的效果：成群的怪物转身向玩家前进，到了射程范围内便开始攻击；它们应该在多个阵线共同工作，并且使用计划和有利于它们的环境。它们应该设置埋伏，制造陷阱，找出玩家的弱点并设法充分利用，并做其他人类玩家会做的任何事情。当然，这是一个普遍的问题。如前所述，大多数 RPG 游戏中的敌人都被认为是相对愚笨无知的，因此玩家可以很快地杀死足够多的敌人，从而以一种让人感觉良好的速度提高自己的等级。问题是这样将接连产生一些非常单调的战斗，充斥着过度愚蠢的怪物。解决这个问题的一般方法是采用一些隐藏头目或适当脚本化和稍微积极的敌人，它们将使玩家觉得并不是全部动物都充斥着无意识的单调行动，并且不是所有的攻击都采取同样的方式。Dungeon Siege 游戏(图 4-3)和 Diablo 游戏使用这种技术相对而言比较成功，因为地图区域始终有一种本地类型的动物，并且该种动物中有一个相对较大且较强的动物来对它们进行领导。这个独特的动物将不跟任何的任务发生关联(尽管有些还是会有关联)，而只是在大量的士兵中提供一点多样性。

这些隐藏头目可以继续深入到真正成为统治部分游戏世界的小怪物头目，而不仅仅只是一般怪物的较强者。纸笔 RPG 游戏的重要贡献之一就是 Dungeon Master(DM)这个概念，即负责建立并运行游戏的人类玩家。隐藏头目可以是小的 DM，向它们的军队发布更好的

命令，并做领袖该做的事情。杀死这个头目之后，玩家将削弱该头目领导的所有动物的攻击，直到它们找到另一个领袖。

然而，关于 Dungeon Siege 游戏的一个题外话是该游戏自动为玩家做了太多的事情，而且在几乎没有用户输入的情况下游戏有时能够自己进行。如果这种自动行为能够被加以改进或调整(甚至只是设置一个滑块，让玩家可以选择他喜欢的自动级别)，那么游戏将会做得更好。

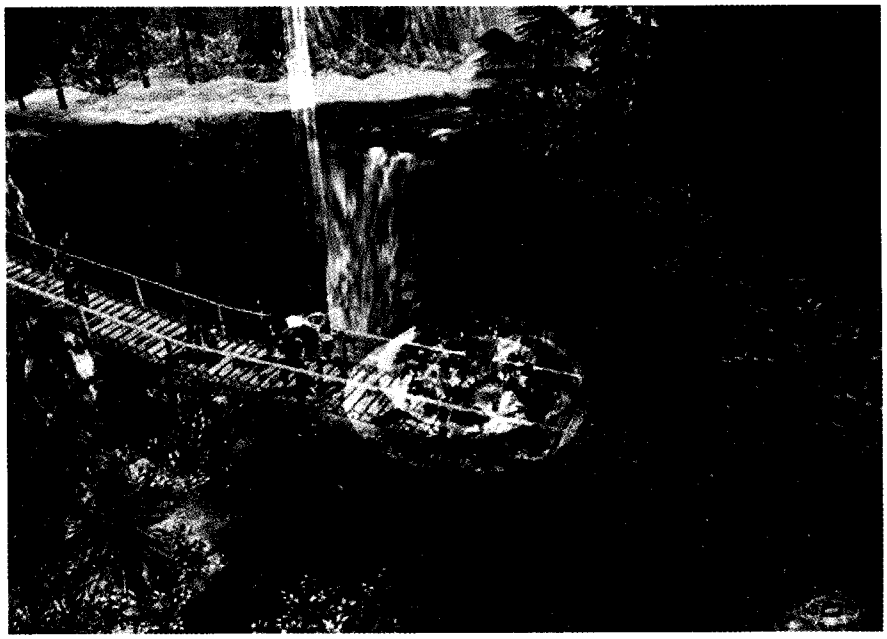


图 4-3　Dungeon Siege 屏幕截图

(©2002，Gas Powered Games Corp。版权所有。Gas Powered Games 和 Dungeon Siege 是 Gas Powered Games 公司的独家商标。未经允许，不得翻印)

4.5.6　完全真实的市镇

构成这些游戏的贸易和信息中心的市镇通常是非常萧条的，充斥着四处走动或在两个位置间移动的人们。这些市镇居民通常反复地说些相同的事情，看起来完全没有一点生气。很显然，这不够真实。通过使用一些简单的规则，以及一种数据驱动的方法来设计市镇，那么即使是很大的市镇也可以居住着一些人，他们或者工作，或者上学，或者去商店买杂货，或者做 RPG 游戏中可以做的所有事情。如果使用类似的系统，那也得让玩家更容易找到市镇中的人(这也是为什么大多数游戏都是让人们站在一个地方，从而用户知道在哪儿可以找到他们)。但这是一个可以解决的问题(或许玩家拥有一个重要的 NPC，根据时间可以在三个不同的地方之一找到他)，并且一个逼真市镇的整体效果将使得游戏世界更加有趣味性以及更让人有一种沉浸感。

有几种不同的方式可以用来实现此类的市镇。可以使用一个基于需求的系统，其中的每个 NPC 都会有许多需求，并将确定如何满足这些需求。随便举一个例子，市镇的某一部分包含有 100 个 NPC。每个 NPC 都有 3 个需求：食欲、商业和家庭。每个需求只有在 NPC

执行与特定需求相适应的任务时才能得到满足(例如，吃东西以满足食欲需求，贸易、训练、交谈等以满足商业需求，养育、赡养等以满足家庭需求)。于是，游戏可以使用一个“需求路径搜索”系统来给每个 NPC 提供关于如何满足其需求的信息。街道上将是忙碌的人们，来回走动、买面包、给栅栏喷油漆、照看小孩等。每个市民采取的行动由其最高需求来决定。实现这个系统的另一种方式是编写许多不同的脚本，每个脚本定义了一连串的动作，并将这些小的脚本分配给地图上的每个 NPC。第二种方法节省了大量的计算(因为不需要进行任何类型的规划，或需求跟踪)，但它不具有一般性(可以设计 100 个不同的地点让基于需求的 NPC 去满足他的食欲，并且 AI 将使用它们的全部；但在脚本系统中除了设计 100 个不同的地点之外，还得编写另外 100 个不同的脚本)。

4.6 小结

作为一种游戏类型，RPG 游戏已经存在很长一段时间，而且将继续存在下去。它们使得人们可以摆脱平凡生活，扮演另一个人或动物的角色。该游戏类型中的 AI 系统是非常复杂的，在整个游戏里具有许多不同的 AI 需求。

- 敌人和敌人头目有必要给玩家一些可以对抗的东西，并提供故事存在下去的动力。
- NPC 和店员给玩家提供了一个更加个性化的交互(不同于战斗)，并给世界(包括游戏产业)一个鲜活生命的感觉。
- 队员 AI 需要引起特别的注意，特别是在基于战斗的实时 RPG 游戏中。
- AI 脚本是在开发 RPG 游戏中使用的一个主要武器，但 FSM 和消息系统也是该类型游戏的主要成分。
- RPG 游戏需要改进的领域包括：能够更好会话的语法机器，适合于多变且持续长久玩法情形的任务发生器，对更好的敌人和队友 AI 的不懈追求，赋予玩家在游戏世界中更大的沉浸感的完全真实的市镇。

5 冒险类游戏

冒险类游戏和早期的计算机是天生的一对。在 20 世纪 70 年代晚期和 80 年代早期，冒险类游戏是沉闷的 PC 用作娱乐用途的最初游戏之一，而且那时 PC 才刚开始流行。

所谓基于文本的冒险类游戏(最初是 Dungeon，它最终演变成经典的 Zork 系列)是我们对该游戏类型的第一次体验，之所以这样命名是因为它们不具备任何的图形画面——仅仅对玩家所处房间进行文本描述并且玩家要如何走步全凭想象。玩家将命令输入解析器中，然后游戏或者以用户输入的行动所对应的结果进行响应，或者告诉用户它不理解他在说什么(如果用户输入的命令不属于游戏命令语言)。玩家从一个房间走到另一个房间，收集用于解谜的元素，解谜成功后将允许他访问另一个区域并继续游戏。

最后，人们开始对那些充满谜题的故事添加图画，这些故事包括类似 King's Quest 系列的游戏、LucasArts'®开创性的 Day of the Tentacle 和 Monkey Island 游戏，以及 Leisure Suit Larry 游戏。LucasArts 同时废弃了完全的文本解析器，而依靠一个高度简化的关键字和图标界面。

1993 年，一个叫 Cyan 的小公司发布了一款叫 Myst(《神秘岛》)的游戏。Myst 是一个冒险类游戏，并去掉了几乎整个的故事情节，只留下一个很优美的世界(它是首批 CD-ROM 游戏之一，并使用预设好的背景，这个背景与当时其他游戏使用的过分简化的实时 3D 游戏世界相比非常让人惊奇)以及大量需要解决的谜题。玩家永远不会死亡，但也没有任何帮助可以指引玩家完成这个游戏，它纯粹是一种试凑式的探险。尽管这看起来只是一个简单的假定，但 Myst 在当时却取得了巨大的胜利，并一直以来都作为最畅销的计算机游戏而备受赞扬(拥有超过一千两百万的拷贝)。它有 5 个后续版本以及无数试图遵循它的规则的相似游戏。

现在，经典的冒险类游戏几乎都已经消失了。似乎没人知道原因。Myst 游戏也许促成了该游戏类型巨大的销售量(冒险类游戏从来就不是很畅销的游戏)，但它也是造成缺乏新游戏的原因。人们开始将冒险类游戏的名字与缓慢的、偶然的赌博联系起来，它们仅仅是一堆的谜题，而抛弃了(或从来没听说过)早期游戏那些写得非常好的、丰富的故事情节。它们都被那些行动导向的冒险类游戏变种的即时满足性所吸引，而这些变种如今正开始占据该游戏类型。

本书将不会把注意力放在冒险类游戏的经典样式中(也叫“交互式虚构”)。这些游戏中内在 AI 元素的级别非常低。它们通常与基于状态的角色一起编码；大多数都只有静态的元素，甚至只有某几个游戏拥有的能够在房间之间移动的行动者。另外，由于人类可以以任意顺序解决多数游戏中的谜题，因此各角色的 AI 类似于实际编码中的标志数据库。

本书将集中讨论那些几乎占据该游戏类型的现代游戏。这些新的冒险类游戏通常都是第一人称射击/第三人称射击(FTPS)游戏类型的变种。这里的 FTPS 一般基于非战斗玩法情形，主要是探险游戏(如 Tomb Raider)以及近来的潜行游戏(stealth game)。潜行游戏涉及一个主要的英雄，他不能靠武力来扭转游戏的主要局势，但可以使用一些潜行和诡计等元素来摆脱并穿过守卫(比如最近的 Metal Gear 游戏或 Thief 系列游戏)。潜行游戏被证明是非常受欢迎的，这是由于它多变的玩法元素，以及高度紧张的感觉。之所以高度紧张是因为必须要想出一种替换直接使用武力的方法来通关并解决问题。这超越了 FTPS 的游戏根源，给我们带来一种持续解谜和强烈故事情节的感觉，而且是在一个实时游戏环境中，因此它们现在被认为是属于冒险类游戏。

另一个包含有一定战斗元素的变种叫做恐怖生存游戏(survival horror game)。诸如 Resident Evil 等的游戏仍然具有许多的战斗，而且主要是弹射攻击，但它们主要是具有许多驱使玩家在地图上四处走动的谜题元素的 3D 冒险类游戏。

5.1 通用 AI 元素

5.1.1 敌人 AI

在很大程度上，潜行游戏中的敌人趋向于由脚本化的运动序列或非常简单的规则来实现。玩家需要偷偷地躲过守卫及其他敌人，因此应该能够识别运动模式并确定如何利用这些模式。然而，一旦发现玩家到来，敌人的行为将变得非常敏捷，因此将非常棘手。守卫角色的注意力通常分为多个阶段(从“我是否听见什么东西？”到假装在他缓慢地向玩家的方向巡逻时没有听见玩家，却打开了枪的保险装置)，并执行诸如请求支援、捕获玩家等基本玩法行为。要记住在该游戏类型中对敌人行为有一个限制，即不希望敌人太过细心，否则玩家将面临让守卫发觉而引发的巨大复杂性，这也会让人类玩家觉得非常沮丧。

对其他类型的冒险类游戏来说，几乎可以使用任何事情。一些游戏像简单的 FTPS 游戏一样使用愚笨无知的猎人式敌人。另外的游戏则具有被约束在一定区域内的聪明敌人(比如 *Thief* 游戏)，因此玩家或许会被一个非常警惕的守卫捕捉到，但如果玩家能在合理的时间内逃离该守卫的地盘，那么玩家就不会引起整个世界的注意。

恐怖生存游戏使用非常简单的敌人 AI，通常是因为涉及的怪物是蛇神之类的东西。相对于探险和谜题交互来说，战斗界面通常没那么重要，因此敌人都是有点缓慢的，且动作也不是那么突然(依靠快速反射)。

5.1.2 非玩家角色

跟 RPG 游戏一样，NPC 是游戏世界中的非战斗居民。他们给玩家提供信息，或使得世界在视觉效果上更为真实。在这些角色上使用的 AI 是各式各样的，可以是一个能力级别和一个执行级别，也可以是一个静态的对话或行动，或者是一个更复杂的包括路径、目标和玩家参与的会话引擎的系统。这完全由游戏设计来决定。

5.1.3　协作元素

协作元素角色超出了 NPC 的领域。在 RPG 游戏中，他们将是玩家的队员。这些人直接给玩家提供帮助，通过指出玩家位置或帮助玩家对抗游戏中的动物，甚或是次要的主角色。后者主要出现在一些玩法中涉及玩家经常在不同游戏角色之间来回切换主要控制(以插曲式或基于任务的时间块的形式)的游戏之中。像这种的切换控制是降低游戏线性度的一种很重要的方式，并将行动分解成玩家容易处理的各块。

由于潜行游戏难于处理的特性，程序员必须要确保这种游戏类型中的 AI 助手不做任何招惹守卫的愚蠢事情，否则我们将会遭受失败。

5.1.4　感知系统

对于潜行游戏来说，AI 模型的复杂性蕴含在感知系统中。人们已经针对不同的感觉开发出了不同的技术——对感觉进行模仿但这种模仿又能很好地在计算机游戏中得到转化。

来自 LookingGlass™工作室的 Thief 游戏把潜行游戏带入了一个全新水平，其玩法的主要推动力是经常性的潜行、在隐蔽处的隐藏以及不进行寻找时挑选与封装相关的角色等。一个游戏行业的程序员在 2002 年的游戏开发大会上对 *Thief* 的感知系统进行了很好的分解。这篇文章可以在 http://www.gamasutra.com/gdc2003/features/20030307/ leonard_01.htm 上找到，标题是“*Building an AI Sensory System*(构造一个 AI 传感系统)”。如果计划要设计一个具有该复杂性的系统，那么强烈建议阅读这篇文献。另外，可以参考随书光盘中的额外链接和材料。

5.1.5　摄像机

大多数冒险类游戏都是 3D(一个显著的例外是二维[2D]Commandos 系列游戏)和第三人称的，因此我们再次看到了糟糕的摄像机布置所带来的问题。然而，由于该类型游戏的慢节奏，这通常是个相对容易解决的问题，而且影片式的摄像机剪辑连同精确的摄像机布置通常是标准规范。但游戏的某些部分可能需要一个自由形式的摄像机系统，因此需要引起程序员的注意。潜行游戏也经常需要一个拐角处的摄像机视角从而进行掩护并观察经过的守卫。这可以是一个当玩家蹲伏在角落里时就出现的算法式摄像机，或者摄像机的具体参数在关卡编辑器中针对特定的掩护位置进行设置。

5.2　有用的 AI 技术

5.2.1　有限状态机

潜行和探险式冒险类游戏的很多元素都非常有助于采用基于有限状态机(FSM)的 AI 系统。如果游戏是数字触发的，比如守卫有一个 yes 或 no 的警戒状态，或者是枚举状态(如中性的、苦恼的、警戒的、疯狂的、狂暴的等)，那么状态机就能起到非常好的作用。由于

状态机本身具有的特性，我们可以将部分 AI 设计得非常简单，而其他部分则具有更多的状态和更大的复杂性。因此，那些只有少数复杂 AI 任务并拥有大量很简单的 AI 任务的游戏应该使用这种系统。

5.2.2 脚本系统

一些冒险类游戏使用电影式十足的摄像机布置、许多游戏内置的对话以及体现关卡中其他地方已经解决了的谜题序列。脚本系统允许程序员(或设计者)对游戏的特殊部分进行额外的调整，而且这种技术很容易在这些游戏所采用的线性故事中使用。将引发脚本序列的触发事件和通过完成任务(从而设置一定的游戏状态标志)来拥有必须要“解锁”游戏后续部分的可信赖游戏机制相结合，将达到两全其美的效果。这使得游戏设计者可以在游戏中的很多地方让某件特殊的事情发生，同时仍然能够给玩家一种不受约束自由闲逛的感觉。

5.2.3 消息系统

由于谜题的事件驱动特性(推一下杠杆 A，门就打开；移动三个石头成一定的模式，隐藏的摄像机就开始工作等)，消息在这些游戏中可以大有作为。因此，游戏中分离的元素不需要相互接近就能通信(尽管标志的协调可以完全在感知程序里面完成)。潜行游戏先进的感知系统能够使用消息来确定感觉到的声音等，同时也给敌人守卫提供了一种不费力的方法来对他人进行警戒或寻求帮助。

5.2.4 模糊逻辑

由于潜行游戏感知系统的复杂特性，在处理传感数据时 AI 对手需要进行模糊决策。这也使得守卫状态对玩家更加宽容(如果没有跨过边界太多，就可以偷偷溜过去——就像推弹球格子，有些运动是合法的，但如果做得过度，则将使它倾斜)。通常，部分玩法是使守卫去处理一些情况，例如玩家引发的注意力不集中、牵制、埋伏以及其他类型的对敌人的藐视。这些类型的交互通常都进行脚本化，但也可以运用模糊逻辑来进行编码，从而允许守卫作为世界的一个模型，去处理由牵制而产生的不完备信息。因此，守卫在他领地上的模型是非常清楚的——他不能立即看到或听见任何值得怀疑的东西。于是，玩家往黑暗的角落里扔一块石头。他听见了，其怀疑级别也相应提高，他在其内部列表上增加了一个怀疑对象，并把他大部分的注意力都集中到这上面，因为这是他现在唯一关注的地方。玩家扔另一块石头，他变得更加怀疑。他大叫：“谁在那？”并拿起他的武器缓慢地向角落走去。我们应该能够理解。不管有多少目标，怀疑的消长由守卫关于世界的非常不清晰和稀少的理解来决定，而这些理解又是由他的感知来决定的。当然，玩家更难以解决此类系统；脚本化系统则通常容易暴露预期行动，因为观察守卫一会后就会注意到每隔两分钟，守卫会起床到阳台上去看看外面，这给了玩家足够多的时间进行移动。在实际中，大多数的这种模糊性在感知系统本身内都将得到更好的使用，而不是在决策结构中。带模糊转换逻辑的 FSM 比一个完全模糊逻辑的系统更容易通过编程实现。

5.3　示例

在经典冒险类游戏开始在大众中走下坡路之后，交叉游戏类型开始出现。Tomb Raider 就是早期的一个将射击类游戏与冒险类游戏交叉的成功的例子。其他更早的游戏包括 Alone in the Dark 和 Shadow Man，后者最终给我们带来了 Resident Evil 这个产生了大量基于恐怖的探险游戏(如 Silent Hill、美国 McGee 公司的 Alice in Wonderland 以及 Nightmare Creatures)。注意这些行动/冒险类游戏仍然包含了许多战斗。这是因为 AI 系统仍然从其 FTPS 游戏中进行了大量借鉴，而且设计者仅仅增加了探险和收集道具的挑战来包含更多的整体体验。

随着 AI 变得越来越好，以及感知系统变得复杂并使玩法更为深入，潜行游戏也随之出现，其最初的代表是 Thief、Deus Ex 和 Metal Gear Solid。这些游戏的娱乐性不在于杀死敌人，而是不让他们发现玩家。Commandos 是一款常见的 2D 潜行游戏，在这个游戏中，玩家的工作就是逐渐渗透到复杂的敌军基地上去并偷偷地从一个地点溜到另一个地点而不被人发现。这个游戏非常难，但做得相当不错。守卫的视线通过在地面上移动的锥面来实际体现，因此玩家可以更精确地安排运动时间以确保其秘密状态。

另一个非常著名的混合冒险类游戏是 Blade Runner，它号称具有真正的多种结局和故事情节，以及一个相当生动的世界。这意味着游戏中的 NPC 使用一些半自主的行为，在城市中穿梭以到达商店或工作地点等。但整体效果主要是装饰性的，因为与 NPC 的交互依然是基于状态和事件的。

尽管经典的冒险类游戏很少，但还没有完全消失。近年来这些游戏的一些重要例子包括 Full Throttle、Grim Fandango 和 Circle of Blood。这些游戏对旧的样式进行了扩展，具有更好的(也更棘手的)谜题、强大的图形显示以及更加多变的玩法元素(*Full Throttle* 甚至包括了一个摩托车战斗阶段)。在过去几年里，这些游戏使用的交互系统在复杂性上有升有降。在最初的文本冒险类游戏里，玩家几乎能输入任意命令，并且游戏解析器要么识别这个命令，要么说成是另外的东西。玩家最终能够学会解析器能够理解的命令。后来，在 LucasArts 的 SCUMM 系统中(它代表 Script Creation Utility for Maniac Mansion，是用于特定游戏的工具的例子，该特定游戏是一整套游戏的基础，因为 SCUMM 引擎最终在不少于 18 个游戏中得到了应用)，可能的命令以按钮的方式在图形界面上提供给玩家，并且玩家可以对屏幕上的不同元素应用这些命令。Full Throttle 则更为抽象，用一些描述玩家眼睛、嘴巴或正在使用的手的图标来表示应用到游戏对象的语境相关的命令。因此，如果应用嘴巴到 NPC 上，则表示要跟他说话；反之，如果应用嘴巴到啤酒上，则表示要喝掉它。由于这种人类输入的简单性便于与游戏进行交互，游戏中的 NPC 也在一定程度上变得更加简化。他们不能真正地与超出一定级别的玩家进行沟通，完全是因为玩家不再具有能够智能地响应的方法。如果 NPC 向玩家询问时间，玩家是用嘴巴图标还是用手图标去点击他呢？如果选择了错误的响应，并且 NPC 询问玩家发生了什么故障，那又如何呢？

5.4 需要改进的领域

5.4.1 潜行目标的附加类型

除了必须要规避模式化运动的经典潜行机制外，Deus Ex 游戏向玩家们提供了许多完成关键故事目标的不同方式。例如，为了通过一条特殊的门，玩家可以射杀该守卫并取下他的钥匙，然后还得与听到枪声后跑来的另外 4 个守卫进行战斗。也可以引起某些类型的牵制，然后利用玩家的剽窃技能来打开没有防卫的锁。或者，可以爬上一个通风井并找到一个不同的进入方式。甚至还可以找到一身守卫的制服并穿上它在守卫身旁走过。通过做这些事情，游戏设计者使得每一个遭遇战和地区都成了一个谜题。玩家需要对各种情形进行试验，从而揭示隐藏的玩法精华。玩家不必偷偷摸摸地走过某一特殊的走廊和打开一条特殊的门，这使得 Deus Ex 游戏的守卫 AI 更具可扩展性，而不会因为有这么多潜在的避开它们的方式而过多地进行脚本化。

5.4.2 传统冒险根源的回归

在早期，传统的交互式小说给计算机游戏玩家带来了一些最流行的游戏。很多经典的 LucasArts 和 Sierra 游戏都拥有许多忠心的跟随者，一直到现在。如今的探险游戏和动作导向游戏必须要结合该游戏类型的经典根源，并再次给冒险类游戏带来活力。

5.4.3 更好的 NPC 通信

现代冒险类游戏的内在非战斗本质非常有利于使一些额外的故事驱动元素作为游戏体验的一部分包括进来。在冒险类游戏中通过赋予 NPC 真正的语法系统，甚或是在整个较大的游戏故事内允许分支的故事情节，冒险所处的世界将变得更为真实，对于玩家来说更具人性化。当然，这需要在故事设计上做大量的工作，以补偿分支和一致性问题。

5.4.4 用户界面

当我们失去最初的文本冒险游戏的完全文本解析器时，我们也失去了与游戏内置角色进行丰富交互的能力。在采用图形界面之后，复杂性逐渐下降直到一些传统的冒险类游戏只剩下三四个基本能用于世界中的元素的命令。现在，在更多的动作导向的变种中，除了更好地给自己定位和在必要时候使用安静型武器或工具外几乎不存在任何的交互。

想象一下带有完全声音界面的 Sam and Max 游戏，或其他类型的一般界面，如果花时间去探究界面能力，在它们身上可以找到对游戏的更加丰富的连接。最终，一个新的界面或许将帮助冒险类游戏重新获得它们传统的深度，而不需要向计算机输入冗长的语句。

5.5　小结

冒险类游戏从其最初的根源开始一直在继续进化，这些根源更多的是一串串封装到故事中的谜题，并且不能实时进行。现代潜行游戏和更多动作导向的探险游戏是经典冒险类游戏的现代变种，它们将继续带给玩家挑战和可探险的新世界。

- 第一个冒险类游戏是基于文本的并需要用户向解析器输入命令。它们最终被图形冒险类游戏所取代，后者属于同样的类型游戏，但拥有一个图形化的用户界面。
- 现代冒险类游戏是 FTPS 类型游戏的变种，并强调诸如探险和潜行的非战斗局面。
- 潜行游戏中的敌人 AI 可以是基于模式的，因为游戏对象需要记录模式并规避冲突。在更具探险性的游戏中，敌人 AI 可以是更加多变的。
- 大多数冒险类游戏都具有大量的 NPC，以及给玩家提供信息或新道具的协作角色。这些智能体的 AI 级别有非常大的变化范围。
- 感知系统对于潜行游戏来说是非常重要的，因为克服守卫的感知就是游戏的目标。
- 对于冒险类游戏来说，摄像机 AI 通常是必要的，因为它们通常是在 3D 环境下进行游戏。
- FSM、脚本、模糊逻辑和消息 AI 系统在冒险类游戏中普遍使用。
- 新的潜行挑战(很可能通过在当前游戏配置中注入更多的智能敌人)是该类游戏中需要改进的领域之一。
- 需要向经典冒险类游戏的根源回归以帮助该游戏类型后代的复兴。
- 逐渐增加的 NPC 通信和故事分支将赋予冒险类游戏对玩家的额外个性化连接。
- 高级用户界面能够帮助恢复传统冒险类游戏和现代游戏更为丰富的交互级别。

6 即时策略游戏

即时策略(Real-Time Strategy，RTS)游戏使用的AI系统在游戏中是属于计算性非常强的类型，这是因为它们通常包括了大量需要协调的单元和需要导航以执行目标的技术树(technology tree)。它们也要占用部分CPU时间用于冲突检测和绘制程序，而这也要与大量单元进行竞争。尽管RTS游戏已经出现好多年(通常认为1990年为Sega® Genesis™控制台开发的*Herzog Zwei*游戏是第一款RTS游戏)，但这些游戏中的AI与优秀的人类玩家相比还相差很远。这是因为RTS游戏中的AI必须要处理巨大数量的对象，典型地只具有不完善的信息(就像战争中的迷雾)，还要非常关注一些细微动作，而且必须要在实时环境中运行。比较而言，AI被认为是专家级(或只是非常好)的大多数游戏都主要是回合制游戏，它们有完善的信息，大多数运动都具有全局结果，因此仅仅通过枚举就能获取比人类规划能力更强的性能。该类型游戏包括国际象棋等。因此，对AI性能来说，几乎RTS游戏的每个方面都被认为是非最优的，需要我们来克服这些问题。

6.1 通用AI元素

6.1.1 个体单元

RTS游戏中的真正玩家是游戏的管理者，即CPU或人类用户。每个玩家的目标都是为拥有社会的全部而战斗。然而，这并不意味着个体单元不需要操心。RTS游戏中的个体行为通常被认为是次要的，只是偶尔比用户给出的主要命令更重要。大多数这种局部智能都属于路径搜索、避障、集中攻击以及不能取胜时的撤退。在这种战术层级上安排多少智能是一个棘手的问题，需要由RTS游戏试图实现的微管理(micromanagement)数量来决定。单元具有的个体智能越多，玩家对军队中的各个单元进行的检查就越少。然而，对于具有较低个体AI的游戏来说，如果CPU对手对它的个体单元AI微管理太多(使得看起来具有更好的个体AI)，那么这将被认为是没有价值的，因为人类不可能快速或轻易地重复计算机所作的努力。它的一个简单例子是*Age of Empires*游戏中的射手行为。计算机将会派遣许多弱小单元，它们可以射击、撤退、再射击、再撤退。行为中这种非常简单的微管理使得这些较弱的单元变得更加强大，因为它们会拉长战线并将守卫的注意力分散到各个方向，这是一个人类很难实现(或至少是冗长乏味)的行为。通过这种简单的个体行为，也使得Age of Empires游戏不去更多地尝试通用的战略技术，比如设立一个混战战士的人墙以及把射手(或其他远程攻击者)放在它们后面以作支援，这是几乎所有人类玩家都做的事情。

6.1.2 雇佣个体单元

雇佣个体单元有时也叫做“佣工(peon)”(施工者和收集者)，它们是那些通常不参与战斗但充当玩家为建造他的军队而获取资源的雇员的单元。与其他个体单元非常相像，其AI级别须仔细调整到游戏追求的微管理级别。Age of Empires近来表明普遍不喜欢游戏AI这个领域，通过在建造一个资源相关的建筑物时让佣工自动地开始收集资源，并且通过“排队等候”农田而不是通过核对并手动重新栽种来使得食品收集更为容易。其他的通用技术包括：

- **命令队列**。在大多数RTS游戏中，界面允许玩家吩咐一个单元去接连执行多个动作。这是对该游戏类型的一个非常强大的改善，因为它允许聪明的玩家提前对它们的雇佣单元进行规划，并因此玩家能够确信他们的雇佣单元在游戏的更多战争导向点上将会非常繁忙。然而，界面仍然需要玩家来创建，因此每个个体单元的AI不需要过多的特例编码来使得佣工看起来更聪明。
- **自动撤退**。佣工单元很少进行战斗(或不擅于战斗)，因此大多数RTS游戏为这些单元设计了某种自动撤退AI。然而，通常这只是在远离敌人的攻击范围里面。它可以进行改进，如靠近建筑物以寻求保护，或跑向最近的军事单元(同时大叫“救命！”)。另外，注意到何时危险已经结束并回去继续工作将是另一个很受欢迎的改进。

6.1.3 指挥官与中级战略性元素

一些游戏直接使用“指挥官(commander)”来支持成组的单元(比如Total Annihilation游戏，除了一个超级单元之外，它使用其指挥官单元作为一个主要的施工者)，或者AI系统内在地使用指挥官来将单元分组到格斗元素中，并在一个更大战争的角度对它们进行控制。这是一个中等层级AI，因为它与简单个体行动(如射击或走到某处)相比需要更多，并且不是完全的高级策略(控制一个特殊的资源，或防卫基地)，而是处于两者之间。一个简单的例子是指挥官为一组单元选择一个新的目的地，但个体单元决定如何保持编队和利用地形特征以到达那儿。更复杂的例子可以是发布一个高级的指示去攻击3号玩家，然后指挥官层级指挥20个步兵从西面进行攻击，之后一组远程武器单元跟随而至，从南面发动坦克，在路上拔除掉几个可能会危害步兵的城堡。这种级别的RTS游戏AI通常非常少，主要是因为它非常棘手。通过直接设定目的地、行动以及与当地和战斗级别目标相符合的敌人目标，它使得协同的战略战斗元素(通常超过5个单元，可能多达30个，但再大的话则被认为是一支独立的军队)朝更高的效率和性能方向发展。

6.1.4 高层战略性AI

可以把这当成真实军队的一般情况进行考虑。按照这种级别的指示执行命令或计划可能需要许多单元，或需要移动系统的所有部分，并包含需要完成的许多不同级别AI的行动。该层级的感知通常建立在较低层级AI的反馈信息之上，以决定敌人所从事的事情。在所有的这些反馈情况下，高层AI将制订计划来处理那些在感知数据之下暴露出来的威胁。这样，策略级别将影响所有事情，包括单个士兵(作为较大组士兵的一部分，他们被指

挥官级别告知以移动来作为响应)和整个经济系统(这发生在要变换对单元的分配时，他们重新获得资源，从而侧重于将支持高层计划的特殊类型)。

高层 AI 经常具有多面性，因为它要对游戏的几个不同方面(防卫、进攻、调查、节约)间的资源分配进行管理，并因此代表了给定 RTS 游戏文化的大多数个性。1 号种族可能重视进攻并具有强大的组织体系。2 号种族可能非常谨慎和细心。将给定 AI 类型的专业单元和一些可调参数结合，可以仅从该级别 AI 中区分出 AI 对手种族的不同类型。

6.1.5　市镇构建

构建一个最初的总部，和为 AI 构建一个先进基地一样，本身就是个困难的问题。为了使保护建筑物变得容易，我们希望能将它们放置在一起。但同时又希望它们能稍微分散，从而得到更好的可见度并预防区域作用武器(area effect weapon)。在这两者间找到一个平衡，同时保持运行一个流畅的系统，是非常具有挑战性的问题。很多游戏使用硬规则来构建市镇(按难度级别进行分解)，开始阶段还好，但是可能不能够处理变化的世界条件，因此在游戏最后看起来非常愚蠢。决定放置关键建筑物的位置需要考虑许多不同的因素。经济建筑物需要放置在它们需要存储资源的附近，军事建筑物需要有清晰的出口通道并靠近前线(如果可能的话)。守卫建筑物需要最大化可见度效果并能够相互支援，以及守卫最大数量的其他单元。

6.1.6　本土生活

大多数 RTS 游戏也包含其游戏世界的本地元素。Warcraft 等游戏有四处活动的羊群，而 Age of Empires 游戏则使用本地动物群作为一种可搜集的资源。其他游戏视本地事物为一种危险，或甚至是一种宝物之类的资源。这些实体的 AI 通常是最小的，但有的游戏则给它们提供了一定程度的“聪明”。依靠这些元素在游戏中体现的特性(视其为资源或危险)，我们需要平衡这些元素的分布以及相关的随意性，否则玩家将得不到任何乐趣。使用随机地图的 Age of Empires 游戏有时也因为使狼太接近玩家的初始市镇而陷入窘境，并且如果这条狼出其不意地杀死一个或多个玩家的初始佣工，那么该随机元素能够极大地减小玩家的初始性能。

6.1.7　路径搜索

对于 RTS 游戏来说，路径搜索是 CPU 最为关心的问题之一。由于具有大量的单元，而且可以想象它们会被派遣到地图上的不同位置，因此游戏路径搜索系统必须正确地找到良好的路径，平衡那些找到这些路径所必需的 CPU 时间负荷，并使用其他最优化方法来使得路径搜索对于许多分离实体都切实可行。诸如编队、群聚技术以及跟随领袖型系统等都将大大提高每个单元路径搜索的速度。其他路径搜索关心的问题还包括友军单元阻碍路径、桥梁等特殊障碍以及用户构建的墙或关卡残骸等动态路径元素。

6.1.8　战术与战略支撑系统

许多 RTS 游戏正逐渐使用扩展的 AI 技术来使得游戏采用的动作更加聪明。这些先进的支持系统包括如下几个：

- **地形分析**。通过将地形划分成容易处理的各块并对每块的不同方面进行分解，AI能够收集对战略性决策有用的大量数据。由于能够为路径搜索系统识别和记录地形瓶颈和古怪地貌，因此探险者可以更容易(以及快速)地找到智能路径。AI 能够跟踪敌人的基地位置和资源并在他(以及另一个玩家)的防卫中找到突破口。这其中的大多数都是通过使用影响力地图来实现的，因为影响力地图正好非常适合于基于网格的地图特征，或专门描述地图中每个网格块的某方面数据。这种数据能够根据额外到来的侦察信息进行更新，但或许代价昂贵，因此要确保预算允许该级别的重复计算。一些 RTS 游戏具有一种特殊的多层模式，某种资源的大多数都在地图上集中，从而导致所有玩家为这种珍贵资源进行残酷战斗。人类玩家能够认识到这是在此类地图上打赢的唯一途径，但除非具体分析该特征地形，AI 对手是很不擅于了解该种地图存在的长期问题。在当地资源已经耗尽后，它们将动身前往更遥远的资源，而通常都被已经控制了该资源的人类玩家所打垮。
- **对手建模**。在具有不完善信息的游戏中，如 RTS 游戏(或纸牌游戏)，我们不能使用标准的 AI 对手假设，如“我的对手将采用与我类似的决策，因为我们都对游戏的状态空间使用相同的最优搜索算法。”原因在于在任意给定的时间点，AI 可能不了解其他玩家的能力，因此不具备对其对手进行预测的基础。通过观察，并注意到对手的身体能力(如看见一个 Dread Mage 或者听见龙的叫喊声)和对手的行为(对手始终从右面攻击基地，或者始终在其金矿附近构建一座城堡)，AI 能够建立起其对手的模型。保持这个模型尽可能更新非常重要，这样在对付其对手时，AI 才能够利用该模型来进行更多适当的决策。通过注意到哪些玩家在他们的军队中设置了专门单元，AI 能够为其对手构建一个相当精确的科技树并知道每个对手使用的其他技术或单元，从而能够为那些可能会用到它们的将来攻击作计划。通过记录玩家的行为趋势(玩家喜欢哪种类型单元，玩家的攻击时间间隔，玩家通常使用的防御类型等)，AI 能够更好地分配防卫以及构建正确的单元来应对来自对手的即将到来的挑战。本质上，这是人类军事将领所做的事情，也是一个古老谚语所说的“了解你的敌人”。
- **资源管理**。大多数 RTS 游戏(Myth 是一个显著例外)都有一个需要跟战斗一样进行照管(如果不是更多的话)的经济体。在进行资源管理时，必须要考虑金子和木材等原始资源以及单元和建筑物等二级资源。大多数游戏的 AI 通过给 AI 一个建造次序(一串需要接连建造的事情，将带来一个旺盛的经济体)来处理这种复杂的任务，这甚至也是人类玩家使用的一种技术。然而，这将导致 AI 行为具有很大的可预见性，因为有经验的人类玩家很快就能发现这个建造次序并从中获悉攻击的大概时间，而且当 AI 防卫在线赶来时他们可以利用其防卫上的漏洞。最好能再包含一个资源分配系统，它能意识到资源的不足并通过使用规划系统来对它们进行调整，从而对满足这些需要所必需的目标进行组织。通过使用基于需求的系统，AI 对手能够向某些单元或资源严重倾斜并更依靠地图类型和特性，而不是盲目遵循一个建造次序并对初始的首场大战役结果做出反应。甚至是使用建造次序的人类也会很快根据他们所看到的具体事情(通过他们的侦察，以地图资源或敌人活动的形式)对建造次序进行调整，因此他们才不会陷于盲目。

- **侦察**。大多数此类游戏都具有一定形式的“战争烟雾”，这是对两种事物进行视觉表示的一种机制：尚未勘查的地形和视线。为了弥补这种感知不足，玩家需要使用单元去对地图进行探测，去揭示边界或资源等地图特征，并找到敌人及其军队。这的确是一个非常艰巨的任务。RTS 游戏中的大多数 AI 对手很擅于探测地图，仅仅因为他们能够比大多数人类更有效地微管理一个侦察单元，但通过额外的侦察监视敌人的运动和营地这个观念却并不普遍。人类需要用它来了解 AI(或其他人类玩家)制造了什么样的威胁来对付他，并要注意到自从上一次侦察兵搜查之后该地区发生的任何变化，比如防卫建筑物的产生，或资源被其他玩家掏空。有些游戏处理该问题的一种方式是让 AI 控制的玩家在建造其建筑物时使用一种分散的方法。AI 玩家不需要记住每种东西放在哪里，因此可以设计出非常随机和分散的市镇，赋予 AI 系统最大可能数量的视线。于是，来自其他玩家的进犯军队能够确保进入这些前线建筑物之一的视线，从而尽早使系统对入侵进行警戒。但这也将导致 AI 建筑物有更大的损失，因为人类经过时将会确保拿下这些前线建筑物。一个更好的系统是大多数人类使用的更为复杂的围墙建筑物和放置护柱。
- **外交系统**。如今 RTS 游戏的 AI 尚未充分利用的领域之一是外交，它可定义成为取得胜利的不同玩家聚集起来一同工作。Age of Empires 游戏采用外交来表明“我们不相互残杀”并且可以共享地图的可见度信息；但它尚未进入更深的领域，比如支持盟军的运动，或分工(“你生成单元，我采矿和建造城堡”)，或简单地伺机攻击以更好地与盟军协调。人类玩家能够很好地管理这些外交任务，AI 系统也应该要如此。当然，这涉及到额外的 AI 工作和额外的用户界面工作，因为人类需要有与其 AI 盟军进行通信的方式，告知他们他正计划在 15 分钟内从南面进行攻击，或者他在第六地区需要帮助。

6.2 有用的 AI 技术

6.2.1 消息

由于游戏中具有如此多潜在的单元，对游戏状态变化或敌人事件进行轮询将在计算上变得非常浪费。相反，可以使用消息系统来快速轻易地对大量注册的单元广播这些事件和游戏标志。

6.2.2 有限状态机

千万不要忘记，有限状态机(FSM)在作为 RTS 世界一部分的不同 AI 任务中将非常有用。个体单元 AI、战略层级内的系统以及许多其他游戏元素都可以利用这种可靠的 FSM。其中，个体单元 AI 通常是作为基于堆栈的 FSM 来实现的，因此它们可以被临时中断然后很容易地恢复；而在战略层级内的系统中，一个城市建筑物可能是一个基本的 FSM，并且由一个被证明是可行的建造次序进行离线构造。

6.2.3 模糊状态机

RTS 游戏更高层级的战略性需求是属于少数几个游戏类型的一些问题，它们不大适合于一般基于状态机的解决方案。关于对手和世界的不完善信息占据优势与需要进行的微决策数量相结合，导致了 AI 对手通常具有多个操作方向且每个方向都是成功决策的游戏的出现。处理此类系统的更好系统是模糊状态机(Fuzzy-State Machine，FuSM)。FuSM 提供了状态机的结构和可再现性，同时又解决了 RTS 游戏决策的盲目操作特性。AI 或许不知道敌人拥有多少坦克，或者对手储备有多少黄金来购买额外的预备部队，但仍然需要设法朝着胜利方向前进。FuSM 允许此类的玩法决策，而不使用更为直接简单的方式，如作弊和赋予 AI 关于其对手的位置和军队构成等知识，以及基于某种随机性和游戏难度级别所采取的所谓“智能”的决策。AI 系统可以通过使用 FuSM 的并行特性来单独确定在任何给定时间内，需要在可能需要引起注意的命令的各方面花费多少努力。因此，AI 所展现出来的完全混合的行为将是非常多变和语境相关的，而且将不使用无所不知的作弊来对 AI 进行帮助。

6.2.4 层次化 AI

RTS 游戏具有多种同时又相互冲突的 AI 需求，比如需要从 A 点移动一支部队到 B 点，但在沿途发现有一个小的埋伏，并且部队成员遭受了攻击。被攻击的单元是否突然停下来并进行反击？还是整个部队停下来并确保问题得到解决？或者所有人都对此威胁不理睬并继续前进？答案取决于个体相对指挥官(或战略相对战术)AI 的数量，但也取决于这些不同层级间的接口以及个人对他人有多大的影响力。层次化系统为 RTS 游戏提供了一种形成高层目标同时在单元层级体现智能的方法，而且不会阻碍主 AI 系统对资源的需求。

6.2.5 规划

目标规划是 RTS 游戏 AI 的很大一部分。为了完成更高层的任务(例如，守卫阵营的左边以防止空袭)，必须把所有的前提任务都添加到 AI 的当前计划之中。因此，对于刚才提到的任务，AI 将需要做如下事情：

(1) 从技术树中获取所有的基础技术(例如，可能需要先构建守卫城堡，然后再构建防空城堡，或者需要一个通信建筑物，从而武器能够使用雷达来探测即将到来的飞机)。

(2) 确定必须要花费的资源单元(如果不足，则将产生一个二级目标来获取更多需要的资源)。然而，技术树导航只是规划的一个领域。具体的进攻或防御目标也需要通过规划来使其看起来更加智能。有人曾经做过研究，为了表现出真实的智能，即使是从一次威胁中逃跑这样简单的任务也需要一定级别的前向思考(而不仅仅是路径搜索)。

6.2.6 脚本

尽管通常不像其他游戏类型那样被广泛使用，脚本也在某些游戏中用于扩展故事元素，或者用于更严格地描述某些单元在某些条件下的行为。某些游戏(特别是最近发展到

```
(defrule
    (current-age == feudal-age)
    (soldier-count > 20 )
    (or   (players-current-age any-enemy >= castle-age)
        (players-population any-enemy >= 20))
=>
    (set-goal GOAL-FAST-ATTACK 0)
)

(defrule
    (current-age >= feudal-age)
    (soldier-count > 20 )
=>
    (set-goal GOAL-FAST-ATTACK 1)
)

(defrule
    (current-age == feudal-age)
    (goal GOAL-FAST-ATTACK 1)
    (timer-triggered TIMER-ATTACK)
    (soldier-count > 20 )
=>

    (set-strategic-number sn-percent-attack-soldiers 100)
    (attack-now)
    (set-strategic-number sn-number-defend-groups 0)
    (disable-timer TIMER-ATTACK)
    (enable-timer TIMER-ATTACK 30)
)
```

6.3 示例

Herzog Zwei 是 RTS 游戏的开山始祖，是一款真正的动作游戏，在游戏中为了得到更多的装备玩家必须要挣钱。该游戏不存在真正的路径搜索，敌人经常会被困住，并且玩家能够欺骗 AI 施工者单元，从而让它不可能进行抵抗。在很大程度上，Herzog 是使用一种非常简单的状态机进行编码的，其状态有 Get Money、Attack 和 Defend。

Westwood Studio®公司出品的 Dune：The Building of a Dynasty 游戏出现在两年之后，并开创了这种一直延续到现在还是主流的游戏规范。在该游戏中，玩家构建市镇、开采资源、扩展技术树以及与敌人战斗。游戏不具有最好的 AI，但在给定游戏的最小系统需求时可以理解成具有最好的 AI。Dune 通常只是使用一个初始的防御建造次序，之后是寻找玩家基地的阶段，然后对玩家进行攻击。它将不会真正重建它的防御(因为它们仅仅在开始阶段建造)，除了玩家基地朝向 Dune 基地的那一侧(不是真正的侧翼包围或试图找到缺陷)，它不会对其他任何地方进行攻击，而且它作弊的技巧非常糟糕(它似乎看不出 AI 已经用完了所有的金钱，并且它将互不相连地建造其建筑物，然而人类却不会这样)。

RTS 游戏的黄金时代包括 Command and Conquer 系列、Warcraft(魔兽争霸)、Starcraft(星际争霸)以及许多的副产品和模仿品。在这个时期 AI 继续向前推进，其中路径搜索进步最大，但这些游戏仍然受到困扰，即 AI 利用的都是人类玩家能够很快找到的东西。这主要是由于不具备使用诸如影响力地图或更好的规划算法等所需要的处理能力或存储空间。

更多现代游戏(比如 Age of Empires 系列、Empire Earth、Cossacks 等)建立在这些适度基础之上并设计出具有许多挑战和相当好 AI 对手的全功能游戏。尽管不断出现一些问题(比如编队破坏路径搜索，且几乎不具备外交 AI)，但这些游戏也能够给人类玩家值得付出的体验，而不通过作弊(在很大程度上)和利用。这些游戏的大多数都通过规划来确定目标和子目标。初始的建造次序依然非常普遍，这只是因为它们容易实现且它们影响难度级别的方式可调。

有的现代 RTS 游戏已经稍微改变了方向，Warcraft III、Command and Conquer: Generals 和 Age of Mythology 就是显著的例子。这些游戏开始强调优胜者或超级单元，而不再是成群愚笨无知的单元。这些优胜单元是顽强者，它们更加能干，但它们的构建或损失都要花费更大的代价。它们也使用更多数量的任务脚本，因此游戏有更多技术性的感觉，而不是早期 RTS 游戏那样的任务(在那里，玩家只是与越来越多的反对力量进行竞争)。

6.4　需要改进的领域

6.4.1　学习

RTS 游戏的 AI 经常反复地在同一个陷阱里被捕获。很容易在 Age of Empire 系列游戏中发现一个简单的例子(尽管它在所有的 RTS 游戏中是非常普遍的)，其计算机将反复让一到两个单元穿过一个城堡(该城堡将杀死它们)。毫无疑问，AI 应该考虑关于地图位置的成功行进信息(使用前面描述的影响力映射技术)，从而能够避免被注意到线路移动的聪明玩家所杀死。RTS 游戏的其他学习还应该包括对手建模，比如跟踪玩家的进攻方向、注意到玩家喜欢哪种类型单元或是跟踪对抗某个特定玩家的多个游戏的游戏策略。玩家是否使用较早的冲锋？玩家是否依赖需要许多某一资源的单元？他是否经常在难以防御的地方建造大量的充满危险的建筑物？他的攻击是否平衡？或者他只是建造了许多石头、许多纸张，但没有任何的剪刀？当玩家开始攻击一个遥远基地时，它需要多久才能响应？此类问题的答案存在于对那些允许智能 AI 系统对此类和更多问题进行响应的统计之中。采用此类信息并不意味着 AI 将逐渐变得无可匹敌；它只是意味着人类为了要取胜只有转变战术，从而强迫他去研究游戏复杂性中的其他领域。瓦解具体的玩家进攻机动的 AI 对手并不一定意味着 AI 本身是攻击性的，除非玩家设置的难度级别非常高。

6.4.2　确定 AI 元素何时受困

从一定程度上说，几乎每一个游戏中 AI 元素(从最低的佣工到整个一群坦克)都可能陷入一种它们完全不知道何去何从的局面。或许所有资源中心都已耗尽，佣工的军队已经没有钱去构建另外一个资源中心，并且佣工拥有一些煤炭却不知该用到何处。或者一群坦克被一个高空单元所追捕(并且不能抵抗)，但还是在接近总部的地方被拦截住，并陷入一个

路径搜索/逃跑的循环，使得这些坦克在试图逃跑时相互牵制进入循环，周而复始。这种令人讨厌的反馈循环将使 AI 元素看起来极其愚蠢，但这正是几乎每个 RTS 游戏所具有的某种形式的行为。对这种“失控”进行探测，并或者对偶然事件进行计划，或者采取某种紧急援助行为，对于提高这些游戏的智能是必不可少的。

这个二级类型是一个经典的问题，即玩家必须要杀死敌军中的所有单元才能取胜，但 AI 使一个佣工单元隐藏在地图某处的一棵树后面。这使得玩家必须要搜索一个半小时，直到他偶然发现了这个僵硬地坐在那无事可做的佣工。RTS 游戏中的 AI 应该能够认识到何时它已经被击败(大多数都这样，但即使是最好的 AI 有时也会变糊涂)并且投降。如果玩家希望穷追直至抓获最后一个佣工，那就让他这样做，但应该给机会给那些狂怒者和歇斯底里的玩家，让他们能够看到费尽千辛万苦赢取的“胜利!”屏幕，而不用花费整天的时间追捕一个白痴单元。

6.4.3 AI 助手

为了缓解游戏中人类玩家反复执行微管理任务的压力，我们需要对 AI 助手进行研究。在第 4 章讨论 RPG 游戏的队员时也提到，可以改进单元独自执行的“自动”行为。一个灵活系统应该能够添加新行为(如果游戏意识到玩家一直在执行一个特殊小行为)、排除不希望的行为并且以适度智能来执行各种动作。该系统将使得实际 RTS 游戏比当前“建造、攻击、建造、攻击”式的鼠标点击更好玩，而且后者中人类了解最好的建造次序并能使游戏以最快的速度取胜。是的，有时这正是一些人所希望的游戏。但现在，它看起来似乎是构建大多数 RTS 游戏的方式。

实际上，系统可以找到一些小的行为宏指令，然后，或者询问玩家是否需要帮助，或者只是接管该任务(可能通过类似于“它已被处理”的消息来告知玩家)。玩家可以选择他喜欢的宏指令帮助级别，其中级别 0 表示不需要帮助，级别 5 将会找出那些重复执行超过 5 次的行为，并且如果玩家取消它们超过一次，则会废除这些行为，在级别 10 中它将会识别任何重复执行超过两次的行为并从不废除这些规则。无论如何，我们都希望少量的宏指令“标志”能够出现在屏幕上(或某个快捷菜单中)，从而玩家可以随时取消所有他希望取消的东西。

6.4.4 对抗人物

Herzog Zwei 具有两个相反的 AI 人物(基于攻击或基于防御)，而且它是最早的 RTS 游戏之一(如果不是第一个的话)。与不同的 AI 人物进行对抗是一种完全不同的体验。想象一下不仅在 AI 难度级别上有变化，而且在其他特征上也有变化。我们在运动类游戏或格斗游戏中是这么做的，为什么在 RTS 游戏上就不这么做呢？通过使用资源分配系统来描述对特定单元的侧重，或在技术树的不同分支上进行特殊处理，我们能够设计出更加有滋味的对手。在开发阶段，不同的稳定人物可以进行调整并相互对抗，从而找出通向胜利的组合。这些人物甚至可以随着时间推移由一个单个的 AI 对手来替换，从而他可以从一个非常平衡的游戏开始，但在经过一次残酷的战斗损失后变得“疯狂”并使用一种更具攻击性的资源分配方案来杀死更多的单元，以示报复。这将使得在与其对抗时 AI 更好玩，并有可能继续深入到外交游戏，因此玩家可能需要重新考虑与某 AI 角色的结盟(此 AI 角色具有刺激

其盟军的趋势，或者是一个鲁莽者且将因为很小的入侵而变得愤怒)，如果他不遵守较大的协商好的战斗计划而不追捕某人，则将视他为一个障碍。

6.4.5　多战略少战术

AI 微管理带来了更好的单个单元的行为。然而，为了看起来更人性化，RTS 游戏需要更好的策略领导小组，而不是在速度或沉闷上胜过人类的单个单元智能。与类似于人类玩游戏方式的更好的规划算法和班(或指挥官)级别 AI 不同，大多数游戏都依赖于计算机在个体基础上快速微管理攻击单元的能力(或者在人类玩家被控制住时具有一些未表现出的单元 AI，这使其感觉起来像是微管理)。这使得 AI 可以做一些人类几乎不可能做到的事情，从而让他们感到沮丧并产生一种 AI 在作弊的感觉。或许应该限制 AI 在给定的时间范围内能够执行的微管理的数量，以模仿人类滚动、点击鼠标和敲打热键所需要的时间。无论怎样，要使得游戏中的 AI 看起来更人性化并最终在对抗时更具娱乐性，RTS 游戏中更好的策略系统还有很长的路要走。一个高质量的策略系统应该实现如下目标：

- 按类型对单元进行分组，然后使用组来支援其他的组，或采用正确反击类型的单元分组来应对特定威胁。现在，大多数由 AI 对手发动的战斗都是从 AI 产生一些基于共同工作非常好的脚本化组合的混合单元开始，并受 AI 所拥有的资源的影响，以及在较低程度上受他们期望从人类玩家中了解的单元类型的影响。这是一个好的开头，但大多数游戏中的策略 AI 也仅限于此。一旦该军队与人类军队实际接触，AI 将以一种更有效的方式来进行响应。该方式使用一个指挥官级别的 AI 决策，它以具有良好反击单元的敌人为目标并像人类那样随着战斗的进行而适时做出调整。
- 通过设置攻击路线来利用多个阵地并将其作为向即将进来的额外军队开放的支援路线。一旦战争开始，大多数 RTS 游戏都需要使用大量的个体单元。它们并不使用太多的攻击调度。把一支军队分开，并从两侧进行行进，是当一个前进的敌人把单元放置在他们不能很好保护的地方时所使用的一种方式。但这需要两个阵线保持同步，从而它们可以相当一致，否则我们所做的也就只是把部队分成两部分而已。
- 使用地形特征来构造最优的围墙建筑物。围墙建筑将良好的RTS游戏AI分离出来。有的游戏使用随机地图产生器来保持多玩家游戏的新鲜感，因此，对于设计高质量、有用且依然最大限度地利用地形特征的围墙来说，对一个专门围墙构建者的需求是极为重要的。
- 如果它们具有可预见性，则按时间表撤退；或者如果所有事情都土崩瓦解，则开始撤退。具有许多“敢死队”单元的战斗应该仅在下列情况下发生：如果有更大的动机，即用它们的牺牲来牵制敌人(去攻击另一条战线，或跑向一个特殊的资源，或其他事情)，如果该力量是确定为对抗一些牢固的敌人防御而设计的，或如果 AI 具有某些类型的“微管理”点(不让它做比人类更快的事情)并让他们在地图上做一些其他事情从而忽略必败之战。与仅仅挑选每一个单元并分配给他们一个总部的目的地相比较而言，撤退应该是个较好的行为。

- 设置一些埋伏的局面，或切断前来军队的撤退路线。人类玩家采用的一个通用策略是保持一个较大武力在前线的后方，然后用少数快速单元前进并从敌人的堑壕中吸引一部分武力并返回到等待的埋伏中。或者，人类会使用这些快速单元从敌人主基地的侧翼去引开相当数量的敌人防御力量，然后派遣较大武力到这个疏于保护的区域。无论哪一种方式，基本策略都是通过武力中的一部分来保护撤退路线的。如果必须要撤退，AI 也不必担心快速敌人单元会跟踪撤退路线并消灭试图逃跑的较慢单元。

6.5 小结

RTS 游戏给游戏玩家们提供了担当负责整个军队(包括补充军队的经济体)的将军的惊奇机会。由于存在巨大数量的单元和在整个地图上可能实时发生的行动，RTS 游戏中的 AI 面临着非常大的挑战。

- 个体单元 AI 赋予单元人物个性，而不会阻碍较高层级的 AI 系统。
- 需要对经济 AI 进行仔细调整，使得人类玩家不需要进行太多或太少的微管理。
- 指挥官级别和队级别 AI 给系统提供了越来越多的策略层，使得每层都简单且易于维持。
- 市镇构建 AI 是一个独特的挑战，它必须要考虑诸如保护、可见度和为了看起来智能而作的前向规划等因素。
- 由于存在巨大数量的单元和复杂的地形，因而路径搜索占据了 CPU 的大部分时间。但是，设计一个优秀的路径搜索器对于游戏的成功也是极为重要的。
- 对 RTS 游戏非常重要的 AI 支持系统包括地形分析、对手建模、资源管理、侦察和外交系统。每一个都体现了 RTS 游戏体验的一个重要方面。
- 由于需要发生的通信的高层特性，消息对于 RTS 游戏来说是一个非常重要的 AI 技术。
- RTS 游戏的 AI 系统必须要处理大量不完善信息，且一个团队必须要将其资源和注意力分解到许多方向。FuSM 就是对上述情况进行模仿的一种很好的方式。
- 层次化 AI 系统，以及规划算法和脚本系统，都是很多 RTS 游戏的 AI 引擎的关键元素。
- 学习，不管是直接学习还是通过间接方法(如影响力地图)进行学习，都能使 RTS 游戏中的 AI 更具适应性。
- 确定一个单元(或整个游戏元素)何时陷入困境是很多 RTS 游戏都还没有解决得很好的一个问题。
- 可以将 AI 助手用作是人类玩游戏时的一个层级，通过给玩家提供自动对他们进行接管的选项来帮助缓解微任务带来的压力。
- RTS 游戏中的对手很少体现出任何的个性，因此，人类玩家并不真正地跟其对手联系。
- RTS 游戏需要更多集中于策略性战斗元素而较少集中于个体单元的战术 AI。

7 第一人称/第三人称射击游戏

第一人称/第三人称射击(First-Person Shooter/Third-Person Shooter，FTPS)游戏是另一个主要的游戏类型，其 AI 工作包括行业内部和经典理论研究领域，且主要是由于 Id 软件公司的早期努力。Id 的多数游戏都极大增强了图形和网络编程能力并在用户扩展性方面打破了场地的限制。其他主要的游戏都是对它的模仿。许多 FTPS 游戏都包括了人们可用于增加级别、改变武器、编写新的 AI 元素，甚至是执行所谓的“总变换(total conversion)”(指整个游戏以某种方式已经有所改变)等的工具。已经开始出现一个完整的“mod”场景，而且人们可以从许多网站上得到关于修改他们最喜欢的游戏的信息并下载其他用户设计的 mod。

明确使用 AI 技术的一个 mod 叫做“机器人(bot)”。这是 FTPS 世界中对自主智能体的称呼，它能够在地图中穿行、寻找敌人、对敌人进行智能攻击并对伤害和宝物等做出响应。程序清单 7-1 是 Quake 游戏中一个机器人代码块的例子。由于他们在 mod 世界中的独立工作，一些机器人编写者从事的是游戏开发界中的正统工作。较好的例子是 Reaper Bot(早期最有名的机器人之一)的编写者 Steve Ploge，他也是 Unreal 游戏的 AI 程序员。很多级别编辑器也开始在 mod 社区中出现。开发 FTPS 游戏的公司在接受访问时，往往是首先向来访者展示公司独立完成的级别或修改，并从社会角度进行很好的回顾。

程序清单 7-1　QuakeC 中 AI 控制的机器人的用户定义脚本示例

```
void (float dist) ai_run = {

    local vector delta;
    local float axis;
    local float direct;
    local float ang_rint;
    local float ang_floor;
    local float ang_ceil;
    movedist = dist;
    if ( (self.enemy.health <= FALSE ) ){

        self.enemy = world;
```

```
    if ( (self.oldenemy.health > FALSE ) ) {

        self.enemy = self.oldenemy;
        HuntTarget ();

    } else {

        if ( self.movetarget ) {

            self.th_walk ();

        } else {

            self.th_stand ();

        }
        return ;

    }

}
self.show_hostile = (time + TRUE);
enemy_vis = visible (self.enemy);
if ( enemy_vis ) {

    self.search_time = (time + MOVETYPE_FLY);

}
if ( ((coop || deathmatch) && (self.search_time < time)) ) {

    if ( FindTarget () ) {

        return;

    }

}
enemy_infront = infront (self.enemy);
enemy_range = range (self.enemy);
enemy_yaw = vectoyaw ((self.enemy.origin - self.origin));
if ( (self.attack_state == AS_MISSILE) ) {
    ai_run_missile ();
    return ;

}
if ( (self.attack_state == AS_MELEE) ){

    ai_run_melee ();
    return ;
```

```
    }
    if ( CheckAnyAttack () ){

        return ;
    }
    if ( (self.attack_state == AS_SLIDING) ) {

        ai_run_slide ();
        return ;

    }
    movetogoal (dist);

};
```

由于这种可扩展性(和产品的稳定性)，Id 公司的一些游戏成为了 AI 理论研究的试验平台。许多不同的研究室都采用他们的游戏，对其代码作重大改进，然后在更接近于真实世界的情形(而不像以前在实验室使用的那样)以及更多真实时间约束条件下测试 AI 技术。从存储环境信息的新方式到更快的规划算法以及完善的规则推理系统都使用游戏进行了测试。

最近比较流行的另一类型 FTPS 游戏是分队战斗游戏(Squad Combat Game，SCG)。该 FTPS 游戏的主“角色”并不只是一个单个的人，而是为一个共同目标工作的一个分队(通常是 3 到 10 个人)。此类游戏从常规的 FTPS 游戏模式开始，叫做“夺旗帜(Capture the Flag)”(每个队都有一面旗帜，如果玩家能得到其他队的旗帜并返回其基地，同时又仍然拥有自己的旗帜，那么玩家所在的队就获得一分)。这个概念被扩展到成熟的军事分队模拟中。此类游戏的 AI 非常复杂，因为分队机动和协调是一个比简单 FTPS 游戏困难得多的问题。

7.1 通用 AI 元素

7.1.1 敌人

FTPS 游戏的主要推动力就是拥有敌人，而且是大量的敌人。同样，这些敌人的 AI 对于产品的寿命也是至关重要的。许多游戏吹捧它们具有“更好的敌人 AI”，但一使用便立即发现它们根本不堪一击。

其他 FTPS 游戏则使用所谓的“街机 AI”，它是老式街机游戏中的简单模式 AI。Doom 游戏和现代的 Serious Sam 游戏非常擅于使用这种技术。它们给玩家提供机会，让玩家可以简单地操作最大的枪并摧毁挡在路上的所有东西，这正是一些人想要的。还有其他一些游戏，如 Half-Life，尝试运用更多的脚本、智能和丰富的玩法体验，并同样取得了成功。

在敌人身上做的工作量(在游戏的单玩家部分)与我们所追求的玩法体验类型直接相关。然而更奇怪的是这个观念，即街机和脚本类型的 FTPS 游戏都很难做得很好。Doom 通过其愚笨无知的敌人、强大的级别设计和武器平衡取得了一个完美的平衡。它带来了无

数的抄袭和模仿，但几乎都没有它做得好。Half-Life 通过脚本化内容在 FTPS 游戏中完成了同样的事情，具有一个伟大的故事、大量手调局面和非玩家角色(NPC)行为，以及良好的氛围。这些努力使得大量游戏试图跟随它做同样的事情，但没人真正做得更好。

7.1.2 敌人头目

一些更加基于动作的 FTPS 游戏，如 Serious Sam，也与射击类游戏和 RPG 游戏一样包含了敌人头目。在关卡末端，玩家将与一个通常更大、更强大、具有特殊攻击和运动能力的敌人面对面相遇。但即使是像 Half-Life 这样复杂的游戏也拥有一些需要对付的真正大动物。这些动物一般非常顽强，但具有一些如果发现就可利用的缺点。有些游戏甚至需要玩家使用环境元素来杀死敌人头目。

7.1.3 死亡竞赛对手

FTPS 游戏所必需的 AI 对手分为两个基本的类别：常规怪物敌人和死亡竞赛机器人。怪物被认为是像野兽的动物或充其量是邪恶的类人杀手。另一方面，在死亡竞赛游戏中，机器人则试图极力模仿人类的行为和性能。有些机器人被设计成拙劣地模仿某些行为(例如仅使用某一特定武器并一直跳跃着的机器人)，但它们主要都是设法模仿那些优秀的、可靠的以及人类死亡竞赛中的杀伤力。

如果希望在产品中添加多人游戏部分，那么需要机器人 AI，使得玩家可以在没法联系别人或只是想练习一下的情况下拥有一个多人游戏的体验。与 FTPS 游戏中的常规敌人不同，这些机器人应该尽可能地跟人类一样聪明(当然有困难级别)，从而给玩家提供娱乐，并使玩家产生打通死亡竞赛环境的挑战性兴趣。

难度级别通常涉及调整机器人行为的不同方面，比如进攻性、多长时间它会撤退并装载健康宝物、适当的武器运用(或者机器人是否拥有一件它使用得最好的武器)以及机器人的目标的优秀程度。

其他逐渐用于机器人的行为还包括使用聊天消息来嘲弄被它杀死的玩家，或对另一个玩家的精湛射击表示称赞。尽管它还非常简单，但随着游戏的持续使用，其效果正变得越来越好。或许将来在 FTPS 游戏中我们会有对等的聊天机器人，它们看起来更具人性。

7.1.4 武器

FTPS 游戏中的武器范围非常广泛，从开创性的火箭筒到 Blood 游戏中使玩家可以远距离射杀其敌人的非常奇怪的“voodoo doll”。通过使用尖端的武器，可以跟踪极度的甜言蜜语，或必须要像热导引导弹那样进行操控，有时智能仅仅用于游戏所使用的某些武器。其他的武器智能问题包括不使用射击时其碎片会造成伤害的武器(用于机器人本身可能被该效应所伤害的时候)，或甚至是更奇怪的武器使用，如第一个 Quake 游戏中的电枪放电(如果将电枪射向池塘中的水，它将杀死池塘中的所有人，包括最初的电枪持有者，如果他也在水中的话)。甚至可以说在决斗中哪种武器对抗其他武器会有更好的表现，以及在给定玩家类型、量程、弹药数量的情况下应该挑选哪种武器，就是明确的智能测试。

7.1.5　协作智能体

当更复杂或故事驱动的 FTPS 游戏开始出现时就有的元素是机器人“助手”，或存在于某关卡上的 NPC 类型。当玩家与这些特殊的角色交互时(而不是去杀死他们)，他们将会向其提供帮助或一种新的武器等。有的此类角色非常复杂，在整个关卡中跟随在玩家左右，帮忙对付敌人并指出地图特征。

成功地使用了该元素的游戏有：Half-Life、Medal of Honor：Underground 以及许多其他游戏。与 RPG 游戏中的协作智能体一样，这里的协作智能体需要具有足够的“智能”，从而不会让玩家觉得是在照顾它；否则玩家将很快抛弃该智能体，或者对游戏感到沮丧。

7.1.6　分队成员

如果要设计一个基于分队战斗的游戏，那么需要花很大一部分时间在分队成员上。基于分队的机动具有很大的范围，可以很简单(前向行进与提供掩护两者交互跃进)，也可以非常复杂(分队中的一部分突然停止前进，拿下一个守卫，同时主要部队继续前进，消灭另一个不同的守卫，然后两部分在某点会合)。控制这些分队成员的 AI 必须是反应性的(如果有人朝自己开枪，不能因为有人告诉自己这么做就保持向一个地点跑去，而应该寻找掩护，寻找射击来源，然后运用一些聪明的方法，或向指挥官报告，或利用地形特征来安全到达目的地)，主动的(如果一颗手榴弹被扔进了我们的战壕，应该有人捡起并把它扔回去，或向它扑去)，以及可通信的(关于他们的成功和失败、他们正在遭受的减速、他们发现的额外信息等)。如果要设计一个具有较少军事性的游戏(例如这样一个游戏，玩家及其虚拟家庭必须要保卫他们的家庭免受外来攻击)，那也需要解决一些额外的个性问题，包括遭受攻击时保持镇定、受伤后的处理、恐慌和看到暴力等。这些都是职业士兵经过训练能够做得很好的事情，但如果发现 8 岁的小妹妹在大量激光扫射以及腿上有严重创伤的情况下还有很好的表现，那么人们可能会认为这非常不现实。

在完成所有这些之后，分队级别的 AI 系统还需要使这个团队更具能力，但又不能太强。这是实现游戏平衡的很好路线。如果这个分队太能干，玩家会感觉自己是个旁观者，但如果该分队很无能，玩家又会感觉像是被一群白痴围绕着。这正是需要进行玩法测试的原因，而且还需要进行很多测试。

7.1.7　路径搜索

尽管路径搜索是敌人和协作智能体两者的一部分，但这里把它单独列出是因为路径搜索是 FTPS 游戏中主要的 AI 系统之一。但在即时策略游戏(RTS 游戏)中，路径搜索仅仅包括地形管理，而 FTPS 游戏中的路径搜索则通常包括了使用游戏元素(比如升降机、远距传输器、控制杆等)和专门的运动技术(“火箭式跳跃”，如果操作不当将会受损害的水下通行顺序)。因此，路径搜索游戏通常使用专门关卡数据和定制的路径搜索“成本”的组合，其中后者有助于解决特殊的古怪运动状态。对动态障碍的局部路径搜索(或避障)用于处理即时区域问题。根据语境，避障可以是作为补充，或是完全重载一般的路径搜索系统。如果一个角色身陷绝境，并被某个其他玩家或环境元素牵制住，那么路径搜索系统也需要识别这种“受困”状态，并为该角色施加一个退出事件。自主的 AI 控制角色能够并将在地图

上找到一些非常棘手的位置并陷入进去，从而路径搜索系统将戏剧性地受它的困扰。通过完全摒弃偶然性(或接近完全摒弃)，可以使关卡设计者自由支配他们所希望的任意环境，同时给创造物一个格斗机会来对他们进行成功导航。

7.1.8 空间推理

RTS 游戏的 AI 系统需要使用地形分析来找到人类擅于定位的游戏世界中内在的特殊元素，同样，FTPS 游戏也需要模仿人类做出的关于游戏世界区域的此类判决。人类非常擅于查看环境并找出狙击手位置、阻塞点、良好的环境掩护等。然而，在实时环境中这是个非常难解决的问题(RTS 游戏能够使用一个简化的一般二维地图来进行此类判决)。因此，该问题通常被认为是关卡设计者必须要帮忙解决的另一个步骤，即用一些 AI 对手能够辨别并加以利用的助手数据来对地图区域进行标记。然而，在一定程度上能够自动执行该过程的系统已经被开发出来，并通常作为一个以某种便于使用的方式产生这种空间推理数据的预处理阶段。

7.2 有用的 AI 技术

7.2.1 有限状态机

有限状态机(FSM)在 AI 编程界中大量使用，在这里它又再次出现。FSM 可以是独占式的(如 Serious Sam)，或只用于非常简单的元素(像 Half-Life 中的那样)。另外，在这些游戏中多数敌人的预期寿命都非常短，通常不需要真正的前向规划。这些游戏的死亡竞赛 AI 包含了最小的状态，通常是沿着攻击、撤退、探险和获取宝物的路线。其他智能来自于特殊的导航系统、机器人的运动模型以及其他支撑程序。程序清单 7-2 是 Quake 2 游戏中 AI 的部分 FSM 代码。该函数用于判断某些 AI 状态(ai_run 和 ai_stand)是否需要转变到 ai_attack 状态。注意标有 JDC(John Carmack 的首字母)的注释行。也要注意//FIXME 的注释，它表示是最终发布的代码。另外，知道 John 是一个人类也有好处。(译者注：John Carmack 是一个顶尖的游戏程序设计师。)

程序清单 7-2 Quake 2 中的部分 AI 代码

(© Id Software，GPL 授权)

```
/*
=============
ai_checkattack

Decides if we're going to attack or do something else
used by ai_run and ai_stand
=============
*/
qboolean ai_checkattack (edict_t *self, float dist)
{
    vec3_t temp;
```

```
qboolean hesDeadJim;

// this causes monsters to run blindly to the combat point w/o firing
if (self->goalentity)
{
    if (self->monsterinfo.aiflags & AI_COMBAT_POINT)
        return false;

    if (self->monsterinfo.aiflags & AI_SOUND_TARGET)
    {
        if ((level.time - self->enemy->teleport_time) > 5.0)
        {
            if (self->goalentity == self->enemy)
                if (self->movetarget)
                    self->goalentity = self->movetarget;
                else
                    self->goalentity = NULL;
            self->monsterinfo.aiflags &= ~AI_SOUND_TARGET;
            if (self->monsterinfo.aiflags & AI_TEMP_STAND_GROUND)
                 self->monsterinfo.aiflags &=
                     ~(AI_STAND_GROUND | AI_TEMP_STAND_GROUND);
        }
        else
        {
            self->show_hostile = level.time + 1;
            return false;
        }
    }
}
enemy_vis = false;

// see if the enemy is dead
hesDeadJim = false;
if ((!self->enemy) || (!self->enemy->inuse))
{
    hesDeadJim = true;
}
else if (self->monsterinfo.aiflags & AI_MEDIC)
{
    if (self->enemy->health > 0)
    {
        hesDeadJim = true;
        self->monsterinfo.aiflags &= ~AI_MEDIC;
    }
}
else
{
    if (self->monsterinfo.aiflags & AI_BRUTAL)
    {
        if (self->enemy->health <= -80) hesDeadJim = true;
```

```
    }
    else
    {
        if (self->enemy->health <= 0) hesDeadJim = true;
    }
}

if (hesDeadJim)
{
    self->enemy = NULL;
    // FIXME: look all around for other targets
    if (self->oldenemy && self->oldenemy->health > 0)
    {
        self->enemy = self->oldenemy;
        self->oldenemy = NULL;
        HuntTarget (self);
    }
    else
    {
        if (self->movetarget)
        {
            self->goalentity = self->movetarget;
            self->monsterinfo.walk (self);
        }
        else
        {
            // we need the pausetime otherwise the stand code
            // will just revert to walking with no target and
            // the monsters will wonder around aimlessly trying
            // to hunt the world entity
            self->monsterinfo.pausetime = level.time + 100000000;
            self->monsterinfo.stand (self);
        }
        return true;
    }
}

self->show_hostile = level.time + 1;// wake up other monsters

// check knowledge of enemy
enemy_vis = visible(self, self->enemy);
if (enemy_vis)
{
    self->monsterinfo.search_time = level.time + 5;
    VectorCopy (self->enemy->s.origin, self->
                                            monsterinfo.last_sighting);
}

// look for other coop players here
// if (coop && self->monsterinfo.search_time < level.time)
```

```
    // {
    //    if (FindTarget (self))
    //        return true;
    // }

    enemy_infront = infront(self, self->enemy);
    enemy_range = range(self, self->enemy);
    VectorSubtract (self->enemy->s.origin, self->s.origin, temp);
    enemy_yaw = vectoyaw(temp);


    // JDC self->ideal_yaw = enemy_yaw;
    if (self->monsterinfo.attack_state == AS_MISSILE)
    {
        ai_run_missile (self);
        return true;
    }
    if (self->monsterinfo.attack_state == AS_MELEE)
    {
        ai_run_melee (self);
        return true;
    }

    // if enemy is not currently visible, we will never attack
    if (!enemy_vis)
        return false;

    return self->monsterinfo.checkattack (self);
}
```

7.2.2　模糊状态机

在这些游戏中也设计有模糊状态机。因为模糊变量的数量通常很少，故我们不会陷入危害模糊系统的组合计算增长问题。另外，来自 FTPS 对手必须要做出判断的输入状态很少像需要考虑的常规状态(如非模糊状态)那样脆弱。AI 控制的对手或许只有 23%的健康度，但拥有一件非常好的武器，并且在人类玩家的背面出现，没被玩家发现。这样，如果 AI 对手已经严重受伤，他是否应该开枪？答案很可能是开枪，但只有在通过对该智能体使用不同模糊输入组合来对系统进行考虑时才可以。此外，这仅仅在考虑要设计的敌人类型时才具相关性。从背面射杀人类并不是一个有趣的行为，除非我们在设计一个死亡竞赛对手。

该技术能够很好地发生作用也是源于此类游戏描绘其动画的方式。角色身体的上部和下部通常是几乎完全分离的。下半部设法播放与行进方向相适应的某一运行动画，而上半部用于瞄准、开火并更换武器。在较低层级有两个状态可能被激活的情况非常适合于模糊解决方案，一个角色可能向玩家射击，但同时又在获取健康宝物，则模糊状态系统就输出“一半射击，一半获取宝物”的解决方案。

7.2.3 消息系统

在更多死亡竞赛 FTPS 游戏中，玩法的推动力量可在一定程度上描述成“具有输入处理机的物理模型”(指游戏玩法非常接近于只是从人类获取输入，利用物理代码来让每个人到处活动，并用导弹武器来摧毁玩家)。因此，在游戏中使用消息系统是一个很好的想法，因为我们有一个始终运行的稳定的基本系统(物理模型)，并用事件来标记所发生的所有感兴趣事情(例如发射火箭，或玩家X进入了 23 号远距传输器)。另外，这些游戏中的多数都是多人游戏，并且很多都使用客户机/服务器模型，这也使得使用消息系统来构建 AI 是一个不错的想法。用户可以拥有一个很小的状态类型系统，具有事件引发的状态变化，并导致消息的进入或“内部”消息(例如，当接收到 IveBeenHitByRocket 消息时便进入被击中状态)。

消息在 SCG 游戏中也具有非常重要的意义，因为它们需要定期在队员之间来回传递信息，包括共享许多关于可见的威胁、位置、状态等。

7.2.4 脚本系统

一些现代 FTPS 游戏使用了很高层次的脚本。包含环境中元素、敌人、会话、玩家与事件的交互以及游戏内的剪辑场景的所有事物都全部(或部分)被这些游戏进行了脚本化。一般而言，脚本仅仅是用于告知故事情节，因此如果 FTPS 游戏具有很强的故事元素，那么这是可以采用的方式。然而，在更多强调动作的游戏中，唯一脚本化的元素很可能是剪辑场景或摄像机移动；如果游戏拥有这些脚本化元素，那么就算是头目类型的动物也将具有一个脚本化元素来给予它们更多程式化的攻击模式。

7.3 示例

老式的 FTPS 游戏都使用简单的 AI，如 Doom 和 Duke Nukem 3D 等。大多数敌人都放置在关卡中，从而使得路径搜索(如果存在)最小化，而且关卡本身的特性(有时称为 2.5D，因为渲染引擎只能处理正面图而不能处理堆叠的房间)也使得相当直接的运动和战斗机动成为可能。

之后，游戏走向完全三维(最初的游戏之一是 Descent)，并开始使用路径搜索系统来躲避。然而，AI 敌人的大脑仍然大部分是没有智能的(除了头目，但它们的顽强一般是处于其纯粹的生命值、风险指数以及很多次玩家与它们一起受困于一个小房间这个事实)，因为人们只希望去杀死事物。诸如 Hexen、Blood、Heretic 等游戏都是属于该类型游戏的很好例子。Heretic 同时也是早期赋予新规则强大界面的第三人称射击游戏之一。

在 FTPS 游戏的下一个发展阶段中，我们突然获得了对我们真实新嗜好的一个完全体验，即在线多人死亡竞赛。在此之前，只有那些在电脑公司(拥有局域网[LAN])工作或拥有好几台家用计算机并通过一个毫无意义的调制解调器将它们串接起来的足够幸运的人才可以感觉到这种模式极其惊人的好处。但最后，程序员通过互联网或拨号连接发现了获取相

当好的玩法的方式，并且每人都希望参与进去。游戏变得越来越好，Quake 和 Unreal 就是其中最好的两个。也是在这个时期，Id 公司为终端用户高度扩展了 Quake(后来 Unreal 也进行模仿)，并因此促进了死亡竞赛机器人的发展，这不断地改变 FTPS 游戏的 AI。人们开始了解在赋予一定的智能之后 FTPS 游戏中的敌人可以实现什么功能，并开始在单人游戏中寻求更具挑战性的敌人。这导致了所有成员都具有更高层次的 AI 复杂性。

如今，这些游戏的一个新变种正逐渐占据着人们的空闲时间。这就是分队战斗游戏，其中最好的一些是 Socom 和 Tom Clancy's Rainbow Six。这些游戏包含了所有常规 FTPS 游戏 AI 并涉及在对抗数组敌人的实时战斗任务中多个队员(如果是单人游戏且没有其他人类的帮忙)之间的协调。在这些游戏中，发送给队员的高层命令和现实需要执行以同步运行的战术 AI 之间有一个很好的平衡。

FTPS 游戏最后出现的一批几乎完全基于战争主题领域(除了我们长期喜好的续篇之外，包括 Unreal)。Battlefield：1942、Call of Duty 以及 Battlefield：Vietnam 都是非常流行的游戏，它们吸取了真实战争中的坚毅不屈的精神，同时看起来仍然非常好并玩得不错。战争博弈的纯粹论者并没有因为允许采用历史细节或武器细节而感到愉悦，但中层射手群则真正喜欢将更加真实的世界包含进来，以及包含许多战争 FTPS 游戏允许的所有交通工具类型，包括坦克、小船和飞机。

7.4　需要改进的领域

7.4.1　学习与对手建模

与其他任何游戏类型一样，FTPS 游戏中的 AI 行为也有漏洞并可加以利用。然而，由于 FTPS 游戏往往是以多人情形在线进行，因此这些漏洞能够更快被发现，而且人们会迅速传递关于该漏洞的知识。FTPS 游戏有变得非常重复的风险，仅仅是因为即使玩家改变了位置、敌人和武器，玩家也仍然能够正确地捕捉到它们并对它们进行射击。AI 敌人需要从游戏生命周期的角度对其对手的个人游戏风格有更多的反应。敌人应该跟踪各种各样的状况，并以此来影响他们的玩法样式，比如：

- **人类使用最多的武器**。很多人都有专门的喜好，或者是因为某种武器的杀伤力很高(例如在不同的 Quake 游戏中似乎人人都喜欢的火箭筒)，或者是因为他们与某种武器有着密切的关系并且已经熟练掌握了使用它的特殊技术(例如最初的 Quake 游戏中的钉枪，它可以在拐角处反弹，而且如果有时间找到一个好的位置进行射击，它将反弹到地图上的一般践踏区，这将是非常具有威胁性的；还比如在 Duke Nukem 3D 中人们找到邪恶的地方来放置地雷)。
- **人类使用的穿越地图的路线**。玩这些游戏的一个普遍方法是学习一条穿越地图的好路线。它将让玩家获取所有的主要宝物，同时又保持他在移动，从而不会让他感到无聊。AI 可以辨别这些路线，并且要么在路线上监视玩家，要么朝人类习惯采用的路线发射火箭等武器，从而强迫玩家改变其游戏。

- **人类接近总部时的战斗样式**。例如，如果人类玩家一直围绕左翼进行扫射，那么AI应该利用这点来躲避即将到来的炮火。
- **人类的玩家类型**。这主要跟人类在玩游戏的同时所采用的运动层次有关。它通常从高层运动(或Hunter类型)，到中层运动(或Patroller类型)，再到几乎不运动(即Camper类型)。还有许多其他的运动，但上述三种是主要的类型。通过了解关于玩家的此类信息，AI对手能够对如何寻找玩家进行调整。

7.4.2 个性

即便现在的机器人玩得再好，而且玩家并没有觉得他们在对他进行欺骗(至少没太过分)，但他们也还远不具备玩家在跟其他人类对抗时所感觉的那种个性。特别是玩家经常跟某人玩游戏时，能够对该对手的个性(他的攻击级别、他受攻击时有多慌乱、他是否扎营等)和该个性的范围(例如，通常对手都是有头脑的，但在游戏的最后三分钟，他变得狂暴和精神错乱)有所认识。机器人设计通常包括他们特别的武器和他们的整体级别。随着玩家对机器人细节的认识，更多个性将实际上导致更具沉浸感的环境。另外，事实上，AI将注意对这种行为的半随机变化，并可能摆脱玩家。

7.4.3 创造力

与人类对抗时，可以看到对方具有许多种使用他们发现的武器和环境的新的且独特的方式。一个具有非常可靠的物理模型(考虑到数学稳定性，不存在许多特例)的FTPS游戏，或者能够注意到人类玩家的轨迹并计算出他们如何到达那里(通过跳跃，然后发射一枚火箭，将玩家高速输送到另一块礁石上)；或者能够随机尝试不同的使用自身的方式，然后通过这些穿越的新方式来标记他们的内在模型。许多人利用跳跃或使用武器的冲力来在地图上四处弹跳，这使得他们很难被击中。

尽管真正的创造力可能超出了AI系统的范畴，但AI程序员应该能够通过AI创造出更加丰富的环境，而且其整体效果将是一个“能够真正很好地理解关卡”的机器人。此AI很受玩家喜爱，并且通常让他们以新颖的方式通关并通过奇怪的方法攻击他们的对手。

7.4.4 预测

在FTPS游戏中优秀的玩家一直采用“预测(anticipation)”这个概念。如果看到一个玩家走进一个房间，由于这里只有一条门，因此可以掌握好区域作用武器的开火时间，当玩家从门里走出来时便对他进行射击。

这将要求AI具有其他玩家的心智模型，并估计玩家需要多长时间才能进入房间，使用可让玩家首先进入房间的宝物，然后恢复原态。之后，AI便可以准备射击，或设置更人性化的埋伏，利用该时间估计来确定需要多长时间。这将是一个相当先进的招式，但如果人类玩家真正希望去实践一下在线游戏是什么样子，那么这就是他需要适应的对手类型。

关于这个的一个较小版本是设置埋伏，或通过推理得知另一个玩家会使用某个入口并躺在那等待他的到来，或通过吸引敌人的注意力，并逃跑，然后在某个AI事先侦察好的安全地点等着他的追来。

7.4.5　更好的会话引擎

现在，FTPS 游戏中 AI 会话的技术发展水平只是千篇一律的一种模式，即当 AI 被玩家杀死或玩家杀死它时它就大声呼叫。这很快就变得重复，而且几乎从不与语境相关或有趣味性。通过一个小小的语法和少许的语句引擎，AI 可以使用与语境更相关的呼叫，这实际上是通过引入一种真实性来吸引玩家的。

7.4.6　动机

现在 FTPS 游戏的 AI 机器人具有两个主要的动机：保持生存和把玩家杀死。有些游戏则不在乎他们自己是否活着。但人们并不是那样战斗的。他们会生气，有时是针对特殊人群。或者，他们会变得很慌乱，并暂时撤退直到他们平静下来。AI 系统需要模仿这种行为，在一定程度上，更真实地模仿他们的人类相似物。想象一下 AI 机器人请求与玩家暂时休战，并与其他人类玩家结盟，或者 AI 机器人不能忍受宿营者(坐在隐蔽的地点从远处狙击玩家的人)并专门追捕他们直至捕获。这些更带感情的行为类型，与一些较高层的口头输出相结合，或许正好使他们看起来更加人性化。

7.4.7　更好的分队 AI

大多数基于分队的游戏都依靠很简单的分队命令(掩护我、跟上来、呆在这儿等)。很明显，此类命令更容易编码，但使用它们也是因为运行一个分队所需的界面需要变得非常简单，从而可以在战斗中快速有效地使用。

更好的方式可以是采用语境相关的菜单。该菜单是当前情形下可能响应的集合，有点类似于足球比赛中的剧本。指挥官能够选择他希望使用的命令，分队将启动这个命令，并从这开始，指挥官可以指挥单个士兵去做一些不同的事情，或改变剧本。有了这个系统，设计者可以为任意给定的入侵设计大量的基本策略、按习惯定制的分队编队以及各单元所承担的行动类型。人类玩家可以通过引导某些士兵去做其他事情来改变这种规则，但这些剧本可用于快速地为每个士兵设计一个切实可行的计划。在每个游戏情形下呈现在玩家面前的不同类型解决方案可以是基于态度的(进攻性、防御性)、基于目标的(保全弹药、展开等)，或者甚至是基于时间的(极端谨慎、立刻就跑)。因此，人类玩家使用的命令类型将使得整个战斗有滋有味。他能够尝试不同的解决方案，从而找到他觉得最舒服的方式，以及给他带来更多胜利的编队类型，甚或是更有趣的游戏局面。

7.5　小结

FTPS 游戏包含一些非常不同的 AI 编程类型，从简单的动物到具有人格和类型的死亡竞赛机器人。该游戏类型根源里的愚笨无知敌人被几乎能够执行所有人类玩家能执行的事情的智能系统所取代。

- 早期的 FTPS 游戏通过使大多数游戏代码具有可访问性和可扩展性，设置在游戏中完成 AI 研究的阶段，这导致了用户所做的修改，或 mod。

- 死亡竞赛机器人是通过设计完全自主的智能体将另一种级别的AI带入到该游戏类型的一种 mod。该智能体探测关卡，追捕玩家，智能地使用武器和宝物，并通常像常规人类玩家那样行动。
- FTPS游戏中的常规敌人是那些在单人战役中设计的敌人，它们或是愚笨无知的街机式敌人，或是跟随敌人样式的更加脚本化的故事。
- FTPS游戏也需要死亡竞赛AI，从而使得那些没有连入Internet或只是想练习的人都可以在死亡竞赛环境下与别人对抗。
- 协作AI机器人通过在游戏的某些部分给玩家提供帮助，或通过除了战斗之外的某种方式与玩家交互，使得游戏具有了故事情节并对动作进行分解。
- 分队AI与那些需要在游戏中处于适当位置的系统有关，这些游戏中玩家控制着不止一个角色，而其他角色则需要由CPU控制。这些机器人需要很高的智能，但其能力需要精密地调整，以使得玩家感觉重要但又不是孤军作战。
- FTPS游戏中的路径搜索尤其需要谨慎对待，因为环境通常是完全3D的并且可以具有非常复杂的建筑物。它们同时也包括了大量的额外玩法元素，如梯子、升降机、远距传输器以及其他需要关注路径搜索的元素。
- 空间推理给AI控制的角色提供了找到关卡相关的关心区域的方式，这些区域比如狙击位置或者便于掩护和可见的地点。
- FTPS游戏使用FSM，但由于FTPS游戏的输入特性，它同时也使用FuSM。
- 消息在这种游戏类型中具有很重要的意义。常规的FTPS游戏能从中获利，这是由于其内在的事件驱动玩法(移动、射击、击中等)和基于服务器在线模式的特性。SCG也可使用消息系统在角色之间轻易地来回协调信息。
- 脚本被用于一些FTPS游戏之中，这些游戏追求更多的手艺感，而不是那种经典的“我们制定规则和大量的关卡”心理。
- 通过赋予我们的创作物更合适的学习和对手建模，我们不再将玩法分解成寻找最好的武器，并通过让玩家与行动融合在一起而对其反复使用。
- 对运动和攻击位置的创造性解决方案将使 AI 对手朝真正的死亡竞赛智能大大前进。对迫近事件的预测将通过保持对可能的未来的心智模型而允许AI角色建立一个直接、未经准备的埋伏。
- 更好的会话引擎可以使当今游戏中千篇一律的呼喊和嘲弄变得更加基于语境，从而更为现实和戏谑。
- 赋予AI对手改变动机的能力可能带来一些如临时休战这样的先进概念，或展现某些类型的情绪性暴怒。
- 多数分队游戏采用的AI都非常简单，但能够有助于一个语境相关的快速命令系统，而且这些系统将带来更好看的分队机动和人类对局面更快的控制。

8 平台游戏

平台游戏(platform game)是控制台世界经常使用的游戏(经典的 Donkey Kong 就是非常早的例子)，而且每年都在进步，直到大规模游戏 Ratchet and Clank 的出现。平台游戏是完美的游戏体验之一，并将以某种方式始终伴随着我们。

旧的平台游戏主要是 2D、单屏以及 Mario Bros 式的玩法。角色从屏幕底部开始，并必须以跳跃方式穿过敌人和环境(游戏因此而得名，即需要从一个平台跳到另一个平台)。在街机世界中它们非常流行，因为它提供了一种崭新类型的挑战：通过计时，实现经考验证明是可行的模式识别街机概念(在平台游戏以前，大多数街机游戏几乎完全都是关于模式的，要么是向玩家走来的具有敌人模式的射手，如 Galaga，要么是需要避免的简单敌人模式，如 Pac-Man 和 Frogger)。平台游戏仍然主要是使用模式化敌人(因为使用模式的技术原因永远不会过时)，但现在人们希望玩家能够在敌人之上实现精确的跳跃以及从礁石到礁石通关并进入最高层。

之后，这个概念被扩展至滚动平台游戏，它获得了人们巨大的喜爱。这种类型的游戏几乎与早期的平台游戏一样，但增加了连续世界的概念，在玩家不断前进时游戏也不断地滚动。因此，与玩家所要占领的单个屏幕不同，现在玩家拥有了要挑战的整个世界，但它只有在玩家晋级到下一关卡时才能展现在他面前。Super Mario Bros、Sonic the Hedgehog 和 Mega Man(参见图 8-1 的屏幕截图)都是此类中非常有影响力的游戏，它们都产生了许多的后续版本以及成百上千的仿效者。

1995 年，一个叫 crack.com 的公司发布了一款叫做 Abuse 的 PC 游戏，后来该公司发布了该产品的全部源代码。Abuse 是一个先进的二维可滚动表格(2D scroller)，完全支持网络多人游戏，并具有几乎与第一人称/第三人称射击游戏(FTPS 游戏)一样的游戏体验。程序清单 8-1 是该源代码的一个示例。Abuse 游戏中的敌人 AI 是用 LISP 语言编写的。读者可以注意到这种动物(这里是蚂蚁)AI 的基本结构是有限状态机(FSM)，它是作为具有不同状态的 select 声明而设计的。

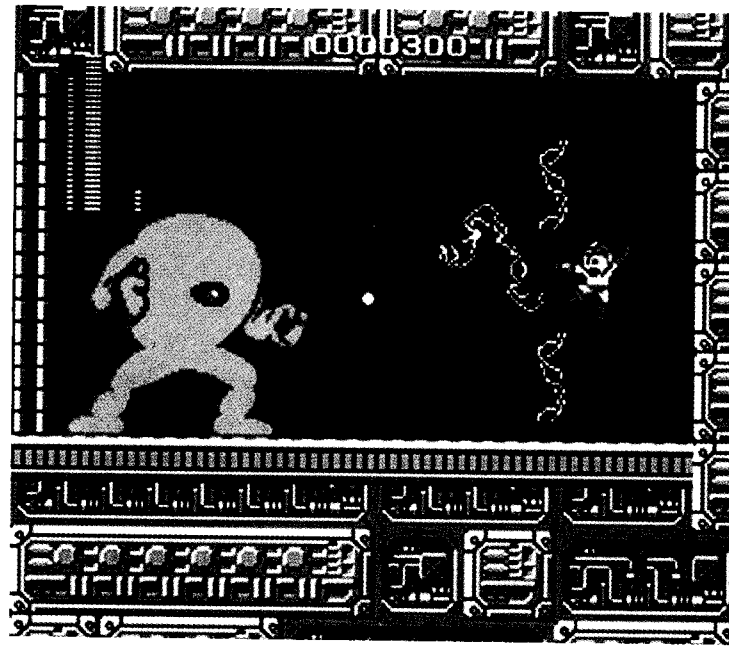


图 8-1 Mega Man 屏幕截图

(© Capcom Co。未经允许，不得翻印)

程序清单 8-1 Side Scroller Abuse 中敌人的 LISP 源代码示例

```
(defun ant_ai ()
     (push_char 30 20)
     (if (or (eq (state) flinch_up) (eq (state) flinch_down))
     (progn (next_picture) T)
     (prong

     (select (aistate)
          (0  (set_state hanging)
              (if (eq hide_flag 0)
              (set_aistate 15)
              (set_aistate 16)))

          (15 ;; hanging on the roof waiting for the main character
           (if (next_picture) T (set_state hanging))
           (if (if (eq (total_objects) 0);; no sensor, wait for guy
               (and (< (distx) 130) (< (y) (with_object (bg) (y))))
               (not (eq (with_object (get_object 0) (aistate)) 0)))
               (progn
                   (set_state fall_start)
                        (set_direction (toward))
                   (set_aistate 1))))

          (16 ;; hiding
           (set_state hiding)
           (if  (if (eq (total_objects) 0);; no sensor, wait for guy
```

```
      (and (< (distx) 130) (< (y) (with_object (bg) (y))))
     (not (eq (with_object (get_object 0) (aistate)) 0)))
      (progn
         (set_state fall_start)
              (set_direction (toward))
         (set_aistate 1))))

(1 ;; falling down
 (set_state falling)
 (scream_check)
 (if (blocked_down (move 0 0 0))
     (progn
         (set_state landing)
         (play_sound ALAND_SND 127 (x) (y))
         (set_aistate 9))))

(9 ;; landing /turn around(gerneal finish animation state)
 (if (next_picture) T
   (if (try_move 0 2)
   (progn
      (set_gravity 1)
      (set_aistate 1))
      (progn (set_state stopped)
      (go_state 2)))))  ;; running

(2 ;; running
 (scream_check)
 (if  (eq (random 20) 0) (setq need_to_dodge 1))
 (if  (not (ant_dodge))
   (if  (eq (facing) (toward))
   (progn
      (next_picture)
      (if (and (eq (random 5) 0) (< (distx) 180)
                               (< (disty) 100)
          (can_hit_player))
          (progn
            (set_state weapon_fire)
            (set_aistate 8)) ;; fire at player
            (if (and (< (distx)100)(> (distx) 10)
                 (eq (random 5) 0))
          (set_aistate 4) ;; wait for pounce

          (if (and (> (distx) 140)
           (not_ant_congestion)
           (not (will_fall_if_jump)))
           (set_aistate 6)

      (if  (> (direction) 0)
           (if (and (not_ant_congestion) (blocked_right
                                    (no_fall_move 1 0 0)))
```

```
            (set_direction -1))
          (if  (and (not_ant_congestion) (blocked_left
                                          (no_fall_move -1 0 0)))
              (set_direction 1)))))))
    (prong
      (set_direction (toward))
      (set_state turn_around)
      (set_aistate 9)))))

(4 ;; wait for pounce
 (if (ant_dodge) T
   (progn
     (set_state pounce_wait)
     (move 0 0 0)
     (if (> (state_time) (alien_wait_time))
     (progn
         (play_sound ASLASH_SND 127 (x) (y))
         (set_state stopped)
         (go_state 6))))))

(6 ;; jump
 (setq need_to_dodge 0)
 (if  (blocked_down (move (direction) -1 0))
    (progn
     (set_aistate 2))))

(8 ;; fire at player
 (if (ant_dodge) T
   (if  (eq (state) fire_wait)
   (if  (next_picture)
      T
     (progn
        (fire_at_player)
        (set_state stopped)
        (set_aistate 2)))
        (set_state fire_wait))))

(12 ;; jump to roof
 (setq need_to_dodge 0)
 (set_state jump_up)
 (set_yvel (+ (yvel) 1))
 (set_xacel 0)
 (let ((top (- (y) 31))
   (old_yvel (yvel))
   (new_top (+ (- (y) 31) (yvel))))
   (let ((y2 (car (cdr (see_dist (x) top (x) new_top)))))
     (try_move 0 (- y2 top) nil)
     (if (not (eq y2 new_top))
     (if (> old_yvel 0)
       (progn
```

```
            (set_state stopped)
            (set_aistate 2))
          (progn
            (set_state top_walk)
            (set_aistate 13)))))))

      (13 ;; roof walking
       (scream_check)
       (if  (or (and (< (y) (with_object (bg) (y)))
             (< (distx) 10) (eq (random 8) 0))
            (eq need_to_dodge 1)) ;; shooting at us, fall down
            (progn
               (set_gravity 1)
               (set_state run_jump)
               (go_state 6))
            (progn
              (if (not (eq (facing) (toward)))
                  ;; run toward player
              (set_direction (- 0 (direction))))
              (if (and (< (distx) 120) (eq (random 5) 0))
              (progn
                (set_state ceil_fire)
                (go_state 14))
          (let ((xspeed (if (> (direction) 0) (get_ability
                                                run_top_speed)
                 (- 0 (get_ability run_top_speed)))))
            (if(and(can_see (x)(- (y) 31)(+(x) xspeed)(- (y) 31) nil)
               (not (can_see (+ (x) xspeed) (- (y) 31)
                           (+ (x) xspeed) (- (y) 32) nil)))
               (progn
                  (set_x (+ (x) xspeed))
                  (if (not (next_picture))
                     (set_state top_walk)))
                  (set_aistate 1)))))))


      (14 ;; cieling shoot
       (if  (next_picture)
          T
        (progn
          (fire_at_player)
          (set_state top_walk)
          (set_aistate 13))))

    )))

  T)
```

1996年，Mario64游戏出现了，并给我们带来了完全的3D平台游戏。Mario64将滚动关卡带入一个完全真实的三维陆地领域，同时保留了更早的同类游戏的所有优点。该游戏

还是现代平台游戏度量自身的蓝本，并作为强大玩法、漂亮的摄影技巧和非常精湛的全面体验的模型。

如今，平台游戏主要是关于三件事情：探险(需要解决的东西藏在哪，如何到达那里)、解谜(或通过特殊的玩法，或通过游戏世界中找到的组合元素)、体格挑战(同步跳跃、执行特殊招式链、克服时间限制等)。此类游戏的设计者正持续推进新玩法机制、新挑战类型以及使该游戏类型更有趣和迷人的新方式的发展。

8.1 通用 AI 元素

8.1.1 敌人

大多数平台敌人都是非常简单的，具有基本的行为，因为在平台游戏中通常认为敌人跟障碍差不多。它们增加了探险挑战的难度(例如，通过把它放在无经验的玩家可能会跳到的精确位置，或然后强迫玩家执行另一个紧接着的跳跃)。这样，敌人的放置便成了设计者需要调整的另一个级别，因为它们找到了这个带来他们所追求的精确困难级别的设置。

然而，有的敌人是更一般的，或诡计多端，或极其熟练(例如 Golden Axe 游戏中的小蓝贼，他几乎不可能停止)。在 Oddworld 游戏中，很多敌人实际上是无敌的，至少直接攻击时是无法击败的。玩家必须找到某种使这些敌人失去能力的方式，通过影响环境或另一个角色，并因此间接地排除这个威胁。Oddworld 差不多是一个扩展的谜题游戏，每一个敌人都是一个谜题，玩家需要决定如何消灭它。

但一般来说，平台游戏更多的是关于身体上的挑战(跳跃、攀登等)，因此，敌人有时从属于次要位置。很多游戏也使用平台的敌人概念，其玩家走在像跳板(垫脚石)一样的巨大敌人的背上，但这并不意味着敌人必须要这样。他能够抵抗，并能够告诫玩家等。

8.1.2 敌人头目

现代平台游戏通常具有较大的、脚本化的、处于关卡末端的动物头目。多数游戏为怪物头目使用脚本模式(随着时间的流逝，玩家会逐渐学会)，并且另外，通常将强迫玩家去执行某些高级的跳跃挑战或展现其他游戏技术(把地面轰成一片片，从而玩家可用的登陆位置变少；或者用破坏性的炮火、尖桩或爆炸来暂时覆盖地面的大片区域)。像使用它们的所有游戏一样，敌人头目对平台游戏体验来说也是极为重要的。它们提供了对常规游戏玩法机制的突变，并提供了游戏节奏，同时它们通常的大尺寸和令人惊奇的能力有助于更有趣的游戏体验。

8.1.3 协作元素

一些游戏包括有一个支持角色，比如添加到 Mega Man 游戏中的狗助手 Rush。这个角色要么受用户的直接控制，要么自动地实现功能，需要的时候就提供帮助。在后一种情况下，AI 代码必须要控制这个角色，通常作为次要攻击、某种形式的宝物恢复或是增大玩法

的组合招式。因此，对于这些游戏智能体，AI通常不会过度复杂而且大部分都是对玩家所做的事情做出反应。在某些方面，玩家不希望有一个过于强大的助手，因为做太多事情的助手最终将使得玩家感觉自己不够重要。大约80%助手都是自主的(指它们在运行一小段脚本或对玩家做出反应)，它们的其他使用存在于对玩家发起的某类“动作”钥匙的响应中。“到这儿来”、“带我走”、或“到那儿去”都是玩家允许使用助手来进行受控的请求执行行动的例子。

8.1.4 摄像机

一旦平台游戏过渡到3D，它们就面临着许多游戏都面临的3D空间中关于精确定位和环境挑战的问题：**把摄像机放在哪才能最好地看见所有事情？**如今，随着更多动态环境和更快玩法的出现，这个问题变得更加显著。一些游戏运用其现代游戏控制台较高的图像处理能力试图对这个问题进行补救，通过使阻碍可见度的游戏元素变得透明，玩家可以透过它们看到动作。尽管这在某种程度上有所帮助，但它拉远了玩家与游戏体验之间的距离，让他觉得自己只是该动作的观察者，而不是主要的角色。智能的摄像机代码与关卡本身的紧密结合，能够用于设计一个具有良好的可见度同时维持与角色联系的摄像机系统。摄像机AI通常由一些不同的方法来设计：

- **从算法上将摄像机放置在主角色的后面并朝向他行进的方向(或其他矢量)**。这带来了可靠的摄像机运动，并且通过对摄像机的相对控制，允许人类玩家使用最少数量令人惊奇的运动(意思是，摄像机不会突然切向一个对玩家完全不同的角度，并因此影响使得玩家移动的控制方向)。该简单系统存在的问题是很难用它来解决特殊地形特征、动态敌人放置、快速移动(或在某一陌生方向上)角色的特殊招式等。实际上，算法解决方案仅能帮助解决一半问题。我们需要一个良好的一般解决方案，并出于玩法机制或关卡设计的原因，需要一个处理游戏可能面临的所有特例的方法。
- **指定为位置和方向而进行的关卡数据跟踪**。这种方法通常与前一种方法组合起来使用，它涉及到关卡设计者在地图中安放大量的摄像机路径。在地图里的某个特殊位置，摄像机通过指向这种地图数据能够知道自己应该在哪儿放置以及该放置的方向。这使得我们能够更好地使用受环境影响的摄像机物镜视角，并设计出一种让玩家感觉身临其境的生动的摄像机镜头。它也可以帮助用户在一个非常大或开阔的世界中判定游戏前进的方向。例如，在游戏中，可能会有一个非常深的洞穴，并具有很多玩家可以顺着它一直往下爬的平台。使用与这类似的摄像机系统，并将随着玩家到达各个阶段的边沿对摄像机位置进行调整，这样，摄像机将帮助玩家了解下一个平台的一般方向。
- **自由摄像机模式**。也即是“第一人称”模式，玩家直接控制摄像机的方向，并从主角色的视线方向进行处理。大多数游戏都包含了这种模式，因为其他两种模式都不能实现无所不包。甚至对那些几乎不需要自动摄像机分解的游戏，一些开发者也给玩家提供了这种选项，从而玩家可以偶尔暂停并欣赏游戏环境(或只是感觉具有更多的可控性)。

8.2 有用的AI技术

8.2.1 有限状态机

在平台游戏中状态机也是非常有用的。这些游戏有着非常简单的敌人，通常所有敌人都只有少数行为(或许除了头目，平台游戏中的敌人头目通常是真正基于状态或基于脚本的)。另外，这些行为通常很脆弱，即它们之间很少有灰色区域。例如，Maximo游戏中的盗尸者要么是在某个随机方向上缓慢地行走，要么是一看到玩家便立即向他发动攻击。

8.2.2 消息系统

大多数平台游戏的谜题式特性使得它非常有利于使用事件消息来告知敌人或环境元素关于游戏状态的变化，因为游戏需要像人类解决问题一样去查询未被发现的阶段，而这是一种非常浪费资源的方式。相反，只有在英雄发现了在房顶上的神奇绿色按钮并按下它之后，事件才被触发，从而打开阻碍全新洞穴的大门。

8.2.3 脚本系统

由于敌人头目的模式特性(更不用提那些正常的游戏敌人了)，脚本是设计这些元素AI的一种自然方式。平台游戏中的脚本允许对游戏某一部分的进程施加很好的控制，如头目遭遇战；或者给玩家提供信息的游戏内的影片序列。一些更复杂的平台游戏拥有一个游戏内的帮助角色，在第一个关卡里它跟随在玩家左右，并告诉玩家如何执行所有的走步以及主角色所能够支配的特殊能力。脚本允许玩家添加这个帮助角色的所有行动和对话，并将其与游戏的控制方案进行关联，从而帮助角色等待玩家执行这些走步，让他独立地进行探测，或是询问一些问题并让帮助角色重复某部分的脚本。

8.2.4 数据驱动系统

3D平台游戏的摄像机有时候非常复杂，因此如果不能找到一个合适算法，摄像机路线必须要在游戏的关卡编辑器中进行构建。在设计者往关卡上添加敌人时，通过了解不同类型动物的运动模式以及这些布置对人类在关卡中该区域通行时的影响，也可以对关卡进行调整。如果在设计者希望具有的各种挑战中加入足够多的事先考虑，以及关卡编辑器的限制和设计者为了调整关卡而需要施加的控制，那么这些游戏将变成是真正数据驱动的。

8.3 示例

经典平台游戏，比如Donkey Kong、Castlevania、Sonic the Hedgehog、MarioBros和Metroid，都是平台游戏中的出类拔萃者。Castlevania非常难，Sonic则非常快。Metroid中的主角色Samus则极其优秀。所有这些游戏都使用基于状态的敌人，而且经常是单状态敌人。通常，这些敌人采用简单的运动模式(比如在两个对象之间来回移动)，或者它们会“隐藏”起来直到玩家靠近，然后突然向玩家跳去。许多此类游戏都认为敌人只有接触到玩家

才能对他有所伤害，因此敌人很少使用攻击策略，而仅仅是与玩家碰撞，尽管有些的确具有一些投射物。

接下来的一代平台游戏有 Mario64(这是一个 3D 平台游戏，游戏中其他公司使用的很多技术都几乎是由 Nintendo 公司的主要游戏设计师 Shigeru Miyamoto 发明的)、Spyro the Dragon 和 Crash Bandicoot 等。由于 3D 世界中额外的运动复杂性，给向 3D 的过渡带来了新的挑战，但也带来了新的麻烦——蹩脚的摄像机系统。游戏继续使用 AI 设计中的多数早期样式，拥有模式化或脚本化的敌人，以及稍微复杂的头目。不幸的是，在平台游戏的 2D 和 3D 时代，许多平台游戏展示的都是其忸怩作态的新角色而不是玩法。我们面对的是急躁、稍微不好的心态和各种聪明的动物，并想方设法兜售那些至多是派生出来的游戏。然而对我们来说，幸运的是，该行业克服了这些小障碍。

如今，平台游戏跟以前一样强大，具有日益曲折的谜题、敌人 AI 以及关卡设计。像 Ratchet and Clank、Jak and Daxter 和 Super Mario Sunshine 等游戏继续推动着游戏技术的发展。有些游戏仍然使用简单的 FSM 和脚本化 AI，但必要时通过更智能的对手和聪明的伙伴对它进行了增强。这些现代游戏的摄像机系统尽管仍然存在问题，但正继续变得更好，且具有非常层次化的摄像机系统越来越接近于始终指向正确方向同时维持并增强游戏的整体感觉。

8.4 需要改进的领域

8.4.1 摄影技巧

和一些游戏的摄像机一样，有很少游戏在这个领域取得了完全的成功，部分是因为人们对摄像机有不同的喜好，部分是因为这的确是个非常困难的问题，尤其是当希望具有一个算法摄像机时。在某些方面，摄像机需要能够预测玩家的运动(甚或是更不可能的移动意图)并移动摄像机来向玩家指明那个方向的情况。该问题也是具体游戏相关的。能够跳很远的角色需要看得更远；参与大量战斗的角色需要对方位有所了解，从而成功击中附近可能反攻并具有很高准确度的敌人。将来，我们甚至可以有专门的外设(比如现在有的游戏使用的戴在头上的具有语音识别能力的麦克风)，并期望它能够跟踪某些运动以帮助摄像机。

8.4.2 帮助系统

有的平台游戏对于某些人来说非常难，或者某个给定位置的谜题能够牵制玩家很长时间。游戏进程中的这些慢下来的动作将很快毁灭游戏体验。如果游戏能够发现人类已经陷入困境并需要帮助，那么它应该能够提供一些暗示以使玩家继续前进。这可以是玩家能够开启或关闭的一个选项，因此那些希望亲自解决所有问题的顽固玩家不会因为失败而感到奇怪。但不经常玩游戏的玩家因为不知道需要绕开角落并使用一个无形的弹射器来越过这个深坑，在做了四小时无用功试图进行不可能的跳跃后，将会非常感激游戏所提供的帮助。这是一个专门的系统，但由于这些游戏的目标导向特性，它有可能拥有一个基于目标的帮助管理器。因此，玩法中的每个细小部分都能够跟踪人类试图解决游戏的原子部分所做的尝试，并注意到他们的失败。此外，游戏中后面同样类型的谜题能够更快地响应，因为游

戏已经传递了玩家对类似的较早的挑战存在困难这个信息。再次，此类系统应该能够进行难度设置(能够设置成开启或关闭，或者一定的帮助级别)，但在开始的“训练”级别或游戏使用的其他系统里应该默认把它打开。

8.5 小结

平台游戏在短短 10 年之间从很简单的事件发展到宏伟的逼真世界。即使是那些风景上的巨大变化，很多公司也成功地保持了娱乐样式的完整性，谨慎地坚持了该游戏类型的优点并将其对所有额外技术的影响降到了最低，这些技术用于具有智能控制和良好 AI 系统的玩法机制。

- 平台游戏中的大多数敌人都非常简单，具有模式化或简单的运动，从而使得在游戏中杀死敌人并没有身体上的挑战那么重要。
- 敌人头目通常是更大和更强的敌人，但通常也仍然进行了脚本化处理。诀窍是发现这些模式，然后利用它来对付动物并把它击败。
- 平台游戏中的协作元素更像是半智能的宝物，因为它们通常只是增强主角色的能力。
- 如果是 3D 游戏，那么摄像机系统对游戏的整体质量级别有着至关重要的作用，因为在这个更大和更开放的世界中在合适的时间看到合适的事情是极其复杂的。算法解决方案、在编辑器中设计的摄像机跟踪以及自由视角摄像机等技术都是处理该问题的典型方法。
- 由于 AI 敌人的简单特性，FSM 在这类游戏中使用得非常多。
- 由于谜题和交互的事件驱动特性，消息系统在该游戏类型中具有重要意义。
- 脚本有助于敌人模式化运动的设计，并给游戏内的影片事件提供了适应定制动画和声音序列的一种方法。
- 需要继续在摄影技巧上进行努力，以便提供给玩家一个获取关于事件的最好视角而又不牺牲控制的系统。
- 应该设计一个帮助系统来对那些在谜题或身体挑战上陷入困境的玩家提供提示或直接帮助(如果他们希望)，这将有助于那些受挫的玩家，但的确需要很大数量的 AI 来对它进行实现。

9 射击类游戏

“射击类游戏(shooter game)”是一种非常开放的游戏类型，包括了经典射击类游戏(静态和平面，或垂直滚动)和现代的变种，且它们都是使用轻武器来进行游戏。大多数此类游戏都为它们的敌人使用非常简单的 AI 或模式，并且任何一个给定的关卡都能找到这种模式(或 AI 弱点)并利用它来轻而易举地进入下一个关卡或敌人。有些射击类游戏给玩家提供足够多的敌人，从而即使玩家能找到这种模式，也难以幸免。简单的控制方案一般是使用陆地定律，人们通常不能在敌人密集的子弹中俯下身去找一个按钮。一个显著的例外是 Defender II: Stargate 游戏，它是一个经典的平面射击类游戏，具有不少于 7 种的控制：上下拨动操纵杆、推进、倒退(用于转向)、一个多维空间按钮(随机对你进行远距运输)、一个射击按钮、一个“不可见”按钮(一类无敌的防护物)和一个智能炸弹按钮(能够杀死屏幕上的所有敌人)。游戏非常难，而且控制方案的特性使得它更难。但这是一个巨大的成功并且持续为人们所喜爱。再次，如果游戏足够好，人们就会花时间去学习如何才能玩得更好。

射击类游戏源于街机游戏，但并没有很好地转化到个人计算机世界，尽管它们在不同的家庭控制台上有很好的表现。射击类游戏通常是关于我们的某个角色面临着巨大数量的敌人，而且它们以某种模式对他发动攻击。玩家需要尽可能多地杀死敌人，同时要躲避(或在某些轻型游戏中，闪避到掩护中去)敌人的子弹。在行进中，玩家获得宝物并与头目(通常是游戏中的大块头)进行战斗。

有趣的是，大量独立设计的射击类游戏都可以在互联网上找到并下载。许多人通过亲自开发一个某类型的 2D 射击类游戏来开始学习游戏编程。这是一种一个人就能独自完成并做得不错的游戏。程序清单 9-1 是 Wing 游戏的开放源代码中敌人 AI 的一个示例，Wing 的创造者(Adam Hiatt)开玩笑地把游戏名称说成是代表“Wing Is Not Galaga”的递推首字母。注意 Adam 的游戏对基于有限状态的 AI 系统进行了简单的实现，它编写了不同的行为(从 Attack_1 到 Attack_5)并使敌人在敌人之间以某些模式进行循环。

程序清单 9-1　Adam Hiatt 编写的 Wing 游戏中的 AI 代码示例

(经 GNU 授权)

```
//=============================================================
void EnemyTYPE :: UpdateAI ( int plane_x, int plane_y )
{
     EnemyNodeTYPE * scan = enemy_list;
   for (; scan != NULL; scan = scan -> next)
```

```
{
    if ( scan -> health <= 0 && scan->explode_stage ==
                                        ENEMY_EXPLODE_STAGES - 1 )
      DeleteNode ( scan );
    else
    {
        if ( scan -> attacking )
        {
            if ( (scan -> xpos >= plane_x && scan -> xpos < plane_x
                  + PLANE_WIDTH) ||
                  (scan -> xpos + EnemyWidths [scan->TypeOfEnemy] >=
                  plane_x && scan -> xpos + EnemyWidths [scan->
                  TypeOfEnemy] < plane_x + PLANE_WIDTH))
            {
                if(timer - scan -> TimeOfLastFired > BULLET_PAUSE &&
                    (plane_y > scan -> ypos + EnemyHeights [scan->
                    TypeOfEnemy] && timer- scan->TimeOfLastFired >=
                    BULLET_PAUSE))
                {
                    scan -> TimeOfLastFired = timer;
                    enemy_bullets.Fire (scan -> xpos, scan->ypos,
                                XBulletVelocities [scan->weapon],
                                    -(YBulletVelocities [scan->
                                    weapon]), scan->weapon );
                }
            }

            switch ( scan->state )
            {
                case ATTACKING_1 : Attack_1 ( scan );
                                        break;
                case ATTACKING_2 : Attack_2 ( scan );
                                        break;
                case ATTACKING_3 : Attack_3 ( scan,plane_x );
                                        break;
                case ATTACKING_4 : Attack_4 ( scan );
                                        break;
                case ATTACKING_5 : Attack_5 ( scan );
                                        break;
                case ATTACKING_6 : Attack_5 ( scan );
                                        break;
                default          :      break;
            }
            scan -> state_stage ++;
            if ( (scan -> ypos < -80 || scan -> ypos>SCREEN_HEIGHT) ||
                  (scan -> xpos + EnemyWidths[scan->TypeOfEnemy] < 0 ||
                   scan -> xpos > SCREEN_WIDTH ) )
            {
                scan -> attacking = false;
                num_enemies_attacking --;
```

```
            }
          }
        }
    }
}
//=============================================================
void EnemyTYPE :: Attack_1 ( EnemyNodeTYPE * enemy )
{
   if ((enemy->xpos >= SCREEN_WIDTH - 75 && enemy->dx > 0 )||
       (enemy->xpos <= 5 && enemy->dx < 0))
         enemy->dx = -(enemy->dx) ;
   else if ( enemy -> state_stage % 20 == 0 )
   {
      if( enemy->xpos < SCREEN_WIDTH / 2 )
      {
         if ( enemy -> xpos <= 160 && enemy -> dx < 0 )
            enemy->dx /= 2;
         else if ( enemy ->dx < 8 && enemy ->dx > -8 )
            enemy->dx *= 2;
         if ( enemy->dx == 0 )
            enemy-> dx = 1;
      }
      else
      {
           if ( enemy-> xpos >= SCREEN_WIDTH-160 && enemy-> dx > 0 )
              enemy->dx /= 2;
           else if ( enemy ->dx < 8 && enemy ->dx > -8 )
              enemy->dx *= 2;
           if ( enemy->dx == 0 )
              enemy-> dx = 1;
      }
   }
   enemy->ypos += enemy->dy;
   enemy->xpos += enemy->dx;
}
//=============================================================
void EnemyTYPE :: Attack_2 ( EnemyNodeTYPE * enemy )
{
   if ( enemy -> ypos == INIT_ENEMY_Y )
   {
   enemy -> dy = 4;
      if ( enemy -> xpos < SCREEN_WIDTH / 2 )
      enemy -> dx = 3;
      else
           enemy -> dx = -3;
   }

   if ( (enemy -> ypos) % 160 == 0)
        enemy->dx = -(enemy->dx);
```

```
    enemy->ypos += enemy->dy;
    enemy->xpos += enemy->dx;
}
//=====================================================================
void EnemyTYPE :: Attack_3 ( EnemyNodeTYPE * enemy, int plane_x )
{
    if ( enemy -> ypos == INIT_ENEMY_Y )
    {
        enemy -> dy = 6;
        if ( enemy -> xpos < SCREEN_WIDTH / 2 )
            enemy -> dx = 3;
        else
            enemy -> dx = -3;
    }
    else if ( enemy -> ypos > 175 )
    {
        if ( enemy -> dy == 6)
        {
            enemy -> dy = 4;
            if ( enemy -> xpos > plane_x )
                enemy -> dx = -10;
            else
                enemy -> dx = 10;
        }
        if ( enemy -> state_stage % 20 == 0 )
            enemy -> dx /= 2;
    }
    enemy->ypos += enemy->dy;
    enemy->xpos += enemy->dx;
}
//=====================================================================
void EnemyTYPE :: Attack_4 ( EnemyNodeTYPE * enemy )
{
    if ( enemy -> ypos == INIT_ENEMY_Y )
    {
        enemy -> dy = 4;
        if ( enemy -> xpos < SCREEN_WIDTH / 2 )
            enemy -> dx = 3;
        else
            enemy -> dx = -3;
    }

    if ( (enemy -> ypos) % 160 == 0)
        enemy->dx = -(enemy->dx);

    if ( enemy-> ypos > 0 )
    {
        if ( enemy -> state_stage % 40 == 0 )
        {
            enemy-> dx = rand() % 13;
```

```
            enemy-> dy = rand () %13;
        }

        if ( enemy->dx > 7 )
            enemy->dx = -rand ()%7;
        if ( enemy->dy > 7 )
            enemy->dy = -rand ()%7;
    }
    else
        enemy-> dy = 4 ;

    enemy->ypos += enemy->dy;
    enemy->xpos += enemy->dx;

}
//=================================================================
void EnemyTYPE :: Attack_5 ( EnemyNodeTYPE * enemy )
{
    if ( enemy -> ypos == INIT_ENEMY_Y )
    {
        enemy -> dy = 4;
        if ( enemy -> xpos < SCREEN_WIDTH / 2 )
            enemy -> dx = 3;
        else
            enemy -> dx = -3;
    }

    if ( (enemy -> ypos) % 160 == 0)
        enemy->dx = -(enemy->dx);

    if ( enemy-> ypos > 0 )
    {
        if ( enemy -> state_stage % 30 == 0 )
        {
            enemy-> dx = rand() % 13;
            enemy-> dy = rand () %13;
        }

        if ( enemy->dx > 6 )
            enemy->dx = -rand ()%6;
        if ( enemy->dy > 6 )
            enemy->dy = -rand ()%6;
    }
    else
        enemy-> dy = 3 ;

    if ( enemy->xpos + enemy->dx < 0 || enemy->xpos + enemy->dx +
         EnemyWidths [enemy->TypeOfEnemy] > SCREEN_WIDTH )
         enemy->dx = -(enemy->dx);
```

```
    enemy->xpos += enemy->dx;
    enemy->ypos += enemy->dy;
}
```

9.1 通用AI元素

9.1.1 敌人

射击类游戏中的敌人通常都很明确地进行了模式化，所以我们可以持续对模式有更多的了解，并深入到游戏之中。因此，这些游戏的AI通常根本不具有智能。轻武器游戏也是采用同样的基本机制：某些家伙的模式从隐藏事件中突现出来，且玩家必须要在被它们射杀前射杀它们。

然而，有的游戏的确偏离了这种基本模式，并使AI敌人很容易找到玩家或使用差不多是第一人称/第三人称射击游戏(FTPS游戏)中类似机器人的行为，使用相当好的“智能”来与人类对抗。然而，甚至是那些拥有先进敌人的游戏也通常使玩家处于某种类型的轨道中(即地图上的一条规定路径，之所以这样称呼是因为看起来玩家更像是在一辆依靠轨道的缓慢火车之中)。这使玩家受到了约束并允许“聪明”的对手躲避到屏幕之外从而逃脱玩家的攻击。轨道概念可用于传统的射击类游戏和轻武器游戏，它主要是控制游戏的节奏(这些轨道最开始产生于街机游戏之中，用来限制游戏中玩家以一定的速度晋级，同时给玩家一种游戏世界缓慢变化的观点)。

其他游戏使用较大的移动的动物(比如Jurassic Park: The Lost World中的恐龙)，它们偶尔会出现一些易受攻击的地方，这也正是玩家要射击的地方。这种行为基本都一样(目标向玩家发动攻击)，但该系统越来越多的屏幕上的运动使得游戏更好看和更有感觉。

9.1.2 敌人头目

与角色扮演游戏(RPG游戏)一样，射击类游戏中的头目也常常在每个关卡末端才能找到。由于其内在的相当重复的玩法，射击类游戏通常过分强调敌人头目。好的头目设计有时能够挽救，甚至是产生比一般的射击类游戏更好和更难忘的游戏体验。因此，头目的AI系统非常重要，并应该足够灵活以包含游戏中每个头目需要的任何类型的专门需求。需要注意的一件事是许多游戏对它们中敌人头目的运动和开火时间进行了完全脚本化。滚动射击类游戏中的头目通常都是非常巨大且全副武装的庞然大物，在各个方向喷射出各种形状和大小的子弹。它们通常分波进行攻击(在实现上，它们转化为状态)，先是阶段性的大规模攻击，接着是短暂的修整，接着是一阵盲目的枪炮扫射，然后全部又再次重复。头目是典型的刀枪不入的，除了某些关键位置之外(通常标成红色，或以某种形式发光)，可以是或不是基于状态的(因为它们有时被某类的保护性外壳掩盖着)。

在狂热的头目战斗中，很多滚动射击类游戏具有一些被主要玩家称为“安全区”的地方。它们是屏幕上一些很特殊的位置，在这些地方玩家可以坐下且不会被敌人的子弹击中，但仍然能够偶尔朝头目开枪。有的游戏接受这个概念，同时使得头目很难对付，而只能依

靠人类去找到一个安全地点；而其他游戏则采取另一种途径，增加一个临时的“自导引”射击来找出那些不动的玩家。

9.1.3　协作元素

有的射击类游戏包括了一个 AI 控制的雄蜂或某种助手对象，它或者是玩法技巧的主要部分(如 Gaires 游戏中的 TOZ)，或者变成一个一旦找到便能帮助玩家的宝物或武器(如 Gradius 游戏中的“Option”宝物)。这些元素通常都很简单，但这完全取决于游戏设计者。但无论如何，我们不希望雄蜂承担太多的工作。

9.2　有用的 AI 技术

9.2.1　有限状态机

在这种游戏类型中，状态机也仍然有用，主要是因为此类游戏中的大多数 AI 都具有简单和直接的特性。游戏本身的组织(基于关卡)从一个容易的阶段开始，接着是组合积累，然后是头目，它也有利于一个基于状态的体系结构。该游戏类型中的多数敌人都只具有一个状态，比如 Centipede 游戏中的主要动物，其 AI 只有一个简单的规则。它将向前移动，直到它击中了一个无辜中弹的旁观者。然后它将移到低处并反向。他拥有的其他的也是唯一的行为是，如果动物只剩下了一部分，那么它就加速。在现代 AI 编程中，这叫做应急行为。Centipede 中的元素与源于交互的最终行为相结合。那时，这才能算是一个好的游戏设计。

9.2.2　脚本系统

射击类游戏中的敌人头目通常属于不能移动的庞然大物且具有一到两个防护得很好的脆弱部位类型，但即使它们能够移动，也很可能仅仅是脚本化的事件。怪物头目很少对人类的行动做出反应，尽管它们可以缓慢地朝玩家前进，或向玩家扑来，或其他。相反，它们倾向于按模式出招同时吐出一波波的子弹和其他东西来伤害玩家。这些简单的一系列行为被教科书用于简单的脚本系统。

增加一个可以在脚本内进行随机分支的能力将给模式脚本带来一定程度的变化(因为各块将以某个随机的顺序执行)。脚本也将使得对具体敌人产物标记难度等级信息变得容易(因此在更难的游戏中将有更多敌人对玩家进行攻击，或从不同的角度和位置)，所以，同样的脚本可以用于简单、一般和困难三个难度级别。

9.2.3　数据驱动系统

射击类游戏一般的敌人 AI(如果遵从模式化的波动范型)对于所有数据驱动结构都是开放的。敌人运动和开火方式的基本类型可以通过代码来定义，然后设计者(或其他人)能够很容易地建立一个关于这些模式将何时何地出现在关卡中的数据库表格，或者实际上可以把它们放在某种关卡编辑器中，通过编辑器来产生这些表格。这样，设计者可以快速并容易地调整关卡中的敌人内容，而不需要程序员的帮助。当然，新的模式需要程序员的介入。

但如果需要，这也可以在编辑器中建立，通过给设计者提供更多的基本积木块来构建范围之外的行为模式。

9.3 例外

Zanac是1986年发布的一款8位任天堂娱乐系统(Nintendo Entertainment System，NES)游戏，它宣称具有“AI 代码的自动难度级别”，并考虑了人类的攻击模式和技能级别。然而，实际上，它只是检查少数的几个状态(比如玩家的射击率、玩家的击中百分比和玩家的存活时间)，然后调整敌人的数目、速度和进攻行为。如果玩家走得够远，杀死了每一个飞船，并使用一个涡轮按钮增强控制器，则它将花费系统大约10分钟的游戏时间用子弹来填充几乎整个屏幕。这是一个很重要的概念，它使得游戏的困难级别随着玩家的能力进行调整，是这样吗？不是。人类将通过不杀光所有人、射失目标和偶尔故意装成快死的样子等对它进行愚弄。所有这些都导致尝试这种困难级别方法的游戏的巨大失败。实际上，我们必须考虑人类的性能，必须要滤除恶意或奇怪的行为，这样系统才不会被愚弄成帮助AI来击败它自己。

9.4 示例

射击类游戏是最初真正的视频游戏之一。的确，Pong类型游戏已经统治游戏界好些年头。但在1978年出现了Space Invaders游戏，它被有些人看作是第一个真正的视频游戏——包括一个分数字段、若干生命以及射击时逐渐蔓延逼近的敌人。随着时间流逝，控制逐渐变得更为棘手，敌人模式更复杂，宝物也逐渐变得更精细和强大。

其他早期游戏(如Gradius、1943、Raiden和R-Type等)进一步对该类型进行了详细阐述：玩家必须要面对数量惊人的敌人，而且这些敌人会一直出现直到真正巨大的目标头目到来并在玩家路线上进行大肆屠杀。

在路线上，玩家可以获得许多宝物，而这些宝物将把玩家简单的飞船变成生产子弹的工厂。此类游戏中的敌人采用模式化的运动。一拨又一拨到来的敌机以来回反复的模式进行运动，比如各种各样的蜿蜒或环绕形状，或者如同足球比赛中的组合线路：直线移动至左侧，直到准备好与玩家拼抢，然后加速并向他发动攻击。

在随后的几年，射击类游戏的流行趋势开始衰减，并产生了轻武器游戏。诸如Duck Hunt、Wild Gunman(它甚至导致了第二部《Back to the Future》电影的产生)、House of the Dead、Time Crisis和Point Blank(参见图9-1的屏幕截图)等游戏都是该变种的主要例子。这些游戏在功能上与前面那些游戏基本相似，但具有一个不同的输入媒介。大多数仍然需要玩家以某种方式躲避敌人的火力，这可以通过命令玩家的屏幕角色躲藏在掩护物后面，或使玩家射击并四处移动角色(比如Cabal游戏)来实现。多数只是要求玩家最先射击。几乎所有这些游戏都包含有宝物，它将提供更强大的武器或更多的健康值等。

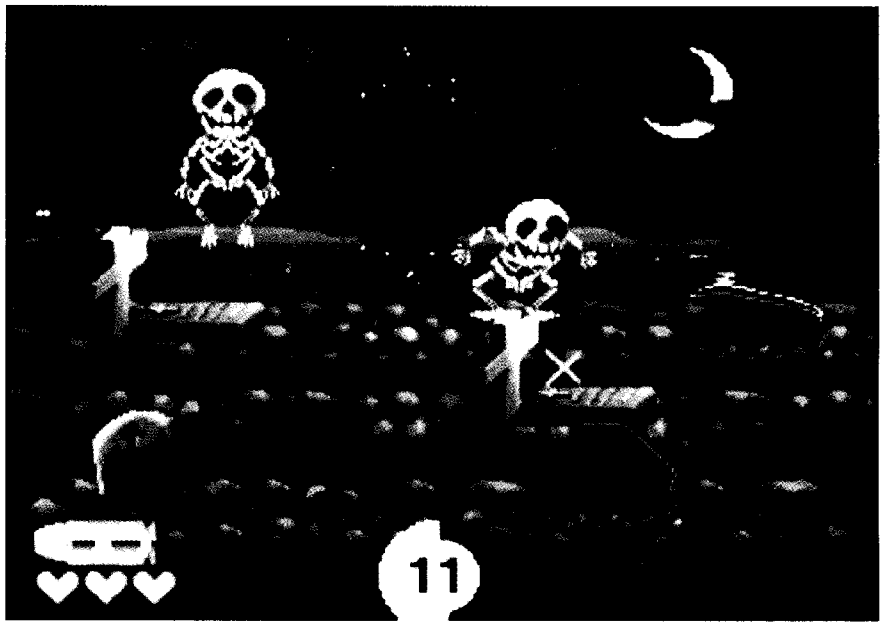


图 9-1　Point Blank 屏幕截图

(POINT BLANK® © 1994 Namco Ltd.，版权所有。经 Namco Holding Corp 允许)

街机平台上的某些射击类游戏试图通过使用一些不同寻常的控制方法来获取本类型之外的额外玩法。Robotron 游戏和 Smash TV 游戏使用两个操纵杆，从而玩家可以在一个方向移动并在另一个方向射击。Cabal 游戏和 Blood Bros 游戏则使用一个控制玩家和屏幕底部第三人称角色的武器瞄准的跟踪球。玩家必须在躲避敌人射击的同时向该角色瞄准。轻武器游戏遵循同样的趋势，其游戏使用不同的枪(比如自动武器、大的来复枪、手枪等)，或特制的枪(例如 Silent Scope 游戏，它使用一个小的 LCD 屏幕来模拟狙击手的视野；或 Brave Firefighters 游戏，当游戏中某处起火时，它让玩家控制一个灭火水龙带，从而可以用它来灭火)。

9.5　需要改进的领域

最近，这些游戏的地位和声誉有所降低，大概是因为人们已经多次尝试这些模式识别的老方法和寻找头目的弱点，这个概念已经逐渐消失。轻武器游戏使得它暂时回到该类型游戏，但最终增加的这点小的玩法也将逐渐变得厌倦。然而，很可能玩法能够保存下来，但应该编写具有真实 AI 程序的敌人。具有该类型 AI 的滚动射击类游戏将更像 FTPS 游戏的死亡竞赛，具有基本的射击类游戏玩法技巧和 FTPS 游戏的机器人对手。这可能是射击类游戏能够继续在 PC 上立足的方式，即设计一个射击死亡竞赛游戏，能够在线进行并可能具有更多同时参与的玩家。

9.6 小结

射击类游戏是很老的一种游戏类型，由于在玩法和内容上缺少创新，它开始变得陈旧无新意。轻武器变种暂时给了该游戏类型额外的激励，但射击类游戏需要一些新的东西来延续一个可行的类型。

- 射击类游戏中的敌人是模式化的；目标是要解决该模式以便进入更深层次的游戏。
- 人们通常认为敌人头目是一个乐趣来源，而且是射击类游戏中非常重要的元素。
- 协作元素通常是先进的宝物，它们包括了额外的玩法技巧。
- FSM 或数据驱动 AI 是用于射击类游戏的主要方法。AI 控制的敌人的简单特性，加上每个射击类游戏的每个关卡通常都是出现了的敌人的一个长期脚本化模式这个事实，非常有利于这两种方法。
- 不管是更复杂的 FSM，还是一个脚本系统，对于较大的敌人头目来说都可能非常有用。
- 真实 AI 技术的融合很可能使得该游戏类型更具生气，一个可能的趋势或许是设计除了在射击类游戏玩法世界之外，还能够在死亡竞赛游戏模式中与玩家对抗的由 AI 控制的机器人。

10 运动类游戏

运动类游戏(sports game)从一开始就一直伴随着我们，因为从技术上说，Pong 就是一个网球游戏。可即刻认知的玩法组合(每个人都知道怎么玩的游戏！)与肉搏战动作相结合，使运动类游戏具有其他类型游戏不可能具有的大量吸引力。再加上大量购买不断出现的各类运动游戏的狂热爱好者，该游戏类型已经成了能够抓住运动类游戏玩家的心的公司的赚钱事业。

在运动类游戏中 AI 正变得越来越重要。早期的运动类游戏几乎都像是动作游戏，在这些游戏中，玩家学习其他队展现出来的模式并利用这些模式来赢得游戏。回到 LED 足球游戏，在那里玩家可以轻易地得到触地得分，只需要在防守队员周围快速且不停地控制鼠标。

现在的运动类游戏希望计算机对手能够像在真实生活中那样进行比赛，有智能，有速度，还有些许风格。AI 对手射击非常精确或者具有“欺骗”对手状态的游戏正因其不公平的数字欺骗而受到指责，并很少有人购买。

大多数有竞争力的运动类游戏都是属于以下两种基本类型之一：

- **流畅的玩法运动**。它们是诸如英式足球、曲棍球或篮球等的运动，游戏很迅速，也很动态，并将持续较长时间(很少停顿或不停顿)。此类游戏的球场条件经常变化，该特性意味着就算是最简单的策略也需要引起十分的注意，以决定何时给定的播放没有奏效，并通过对下一组游戏条件进行响应来很好地恢复。基于状态的 AI 倾向于把这些类型的游戏进行分解，因为有很多“状态”都与其他状态相连接，其结果是蜘蛛网而不是一个很好的流程图。状态层级有助于解决这个问题，但工作层级的结构似乎很难权衡。
- **可重设的玩法运动**。它们是这样一种游戏，当一个设定好的事件或时间到来之后，游戏便停止且复位，比如足球和篮球。此类游戏中的 AI 团队因为能够很快重新开始而获得好处，因此能够基于它来设计 AI 系统的结构。此类游戏更有利于一个基于状态的系统，因为运动本身可以很好地划分成不同的游戏流程状态。

在运动类游戏的 AI 引擎上进行工作的好处之一是通常在产品开始之前游戏就已经完全设计好了。至少，用户试图模仿的基本游戏就是这样。如果要设计一个使用机器人和武器的篮球派生游戏，那么需要进行独立解决。但直接的运动模仿具有一个优势，即有大量的关于如何打赢一场比赛的信息，多年的研究和玩家统计能够支持这一点。

该优势同时也是它的一个缺点。无论玩家往哪边看，都有运动着的人。他们的吃、喝和呼吸都与游戏相关。他们了解所有的状态，遵从他们的团队，并对游戏和玩家充满热情。

他们最先购买运动类游戏。游戏的主要观众都具有关于该运动的大量熟悉知识，这的确将给开发者带来压力。如果是要进行纯粹的模仿，那么最好能把它做好。如果玩游戏的人所看到的玩家行为在真实生活中不可能出现，那他将会有所认识。游戏试图模仿的部分玩家或许是名人，他们的行动和性能级别是人们是否认同系统对他们的描绘的标志。

10.1 通用 AI 元素

10.1.1 教练或团队级别 AI

把教练或团队级别 AI 看成是在即时策略游戏(RTS)或国际象棋游戏中发现的策略性 AI。高层 AI 进行的决策包括调用哪个剧本，或替换一个玩家因为他身陷犯规困境而且希望把他放到最后一节等。如果运动类游戏的 AI 系统中没有这个级别，那么该队的玩法看起来很随机，并缺乏目的性。当然是这么一种情况。

团队级别玩家包括整个团队级别的决策，但可能也要处理稍微小些但仍然涉及不止一个玩家的任务(以协作的方式)，比如足球比赛中的传球或篮球比赛中为持球者开路。通常，系统中的该级别使用某种共享数据区(比如黑板系统，或一个团队单独类)，该共享数据区对该级别中的工作进行了封装，并提供了一个不同的其他游戏元素进行寻找的主要地点(当它们需要使用这些团队决策的时候)。

在对这部分运动类游戏的 AI 系统进行编码时的一个共同性错误是不对任务进行分解或在该级别上使用所有属性数据。在玩家级别时，大多数运动类游戏都经常使用属性，但在对团队级别进行编码时不应该有同样的想法。使用团队级别属性和整体目标，相同的系统同样能模仿特定团队进行比赛的不同方式，这在教练是更重要元素之一的游戏中尤其显得重要，比如大学的篮球比赛。玩家很棒但经验不足，因此教练要调用几乎所有的剧本和策略，并且两支大学球队可能进行完全不同类型的比赛，即使每个队的玩家具有相似的技能级别。

10.1.2 玩家级别 AI

在玩家级别，AI 决策关注那些仅涉及玩家的更个人化的战术行为：从一垒位置开始进行快速移动以争取获得空档，或者对其防卫者做一个假的出招从而执行团队级别 AI 给他的大目标。该层级产生的决策和行为也要仔细考虑玩家的个人属性，按照其真实生活中的对应者进行思考(如果有的话)。通过扰乱具有真实统计的 AI 行为，人类玩家会觉得他在与一个跟真实的运动玩家的技能级别相当的角色比赛。这样，运动类游戏的 AI 也包括了一个大的模仿元素，因为我们并不希望每个人都是超级英雄。相反，差劲的传球手应该丢掉更多的球，拙劣的防卫者应该放弃抵抗并让更具进攻性的玩家很好地完成任务。

一旦被分配了具体行为后，玩家级别 AI 实际上更像是两个系统：战术决策部分和动画选择(更多知识可参见第 23 章“分布式 AI 设计”)。作为一个例子，让我们看看在足球比赛中为传球而设法寻找空位的思考过程。

- 决策系统决定它希望得到一个空位。因此，确定假动作类型(基于属性、个人偏好和防卫配合)和运动方向(根据与其他玩家的距离、球场边界以及一般站位)。

- 动画选择过程将选取该行为数据(方向和类型信息)，并用它来确定玩家将使用的假动作的精确动画。动画层将考虑的其他因素有：玩家类型(大、小、快、卖弄或一些标志性招式)、玩家速度、方向的变化(小的变化只是使玩家旋转，大的变化则需要掉头类型的变换招式)、使同样动画不会一直播放的某个随机因素，以及取决于行为的其他因素。

有时，运动类游戏中的动画选择几乎是玩家执行每个动作的第二步骤。如今的大多数运动类游戏对游戏中的招式使用运动捕获(motion-captured)的动画。运动捕获提供了球星的标志性招式，并显示了次要身体运动的丰富性。这通常只能通过运动捕获技术来得到，因为它很难通过手动来很好地实现这些动画。对于某些招式(比如足球比赛中的“end zone dance”或篮球比赛中的“dunk”)，玩家需要大量类型的动画，因为它们变成了游戏中的嘲弄，允许反复提起那些特别棒的与对手的对抗。由于对一个给定的行为有如此大量可用的动画，系统必须能够使用各种条件，从巨大数量的可用动画中精确挑选出最符合语境的正确动画。一般的动画选择技术(如基于表格的系统)能够用于描述属性数据(和任何其他的决定因素)与各动作的不同动画之间的链接，这将极大地改善游戏的整体组织性并通过数据驱动该过程来限制代码的重复。

动画选择通常不作为 AI 系统的一个纯粹部分，因为人类玩家需要这个相同功能来执行游戏行为。即使这样，该过程还是需要一定级别的智能，因为它一般是一个高度语境敏感的判决(即在行为基础上的一个独特过程)；笼统的方法将很快使得游戏看起来更乏味或非常不合适。

10.1.3　路径搜索

要在运动类游戏的狂乱中找到良好的运动路径是件真正艰巨的任务。的确，屏幕上显示的角色数量是有限的，并且环境中通常没有静态障碍物(尽管不总是这样，比如在曲棍球和足球比赛中就有一个大网)，但动态障碍物(其他玩家和可能的裁判)几乎总在不停地运动，使得传统的路径规划非常缓慢和麻烦。必须要使用一种轻量级的 CUP 最优化的方法来使玩家像在真正比赛中那样相互之间四处移动。

大多数运动类游戏中的导航在选择路径时，也需要考虑比赛相关的信息。比如，在篮球比赛中，假定玩家所在的队在发动进攻，那么如果他能对持球者有所帮助的话就不要跑到其正前方。即使玩家能够相当有技巧性地避开他，他也打断了他的运动并甚至可能在他前面造成堵塞。这是我们不希望的。在足球比赛中，有更多关于路线的规则，而找到好的路径(或破坏它们)实际上是比赛的一项主要工作。

10.1.4　摄像机

现代运动类游戏的摄像机系统通常具有两个相互矛盾的目标：为了推动好的游戏玩法必须要以最好的方式来展现动作；使其看起来像是电视直播的运动类游戏。这两个目标集中于摄像机物镜视角的类型、剪辑以及可用于游戏中且保持游戏的可玩性的运动方式。这两个目标的平衡只能由具体游戏的设计来决定。如果是要争取得到“成为玩家”的经验，

那么可能要试验不同的摄像机物镜视角，而且这些视角差不多都是第一人称的或者严重向某一玩家的视角倾斜。如果是要设法让人类玩家感觉到“身临其境”，那么应该增大摄像机焦距，让人类对动作有一个更宽广的、整个球场的视界。类似于游戏设计类型的其他摄像机样式包括：be the coach、watch the game on TV(一个非常普遍的选择)、old school(在很多较老的游戏中使用的俯视、几乎是二维的视野 view)等。图 10-1 所示是使用这些样式的两个例子。

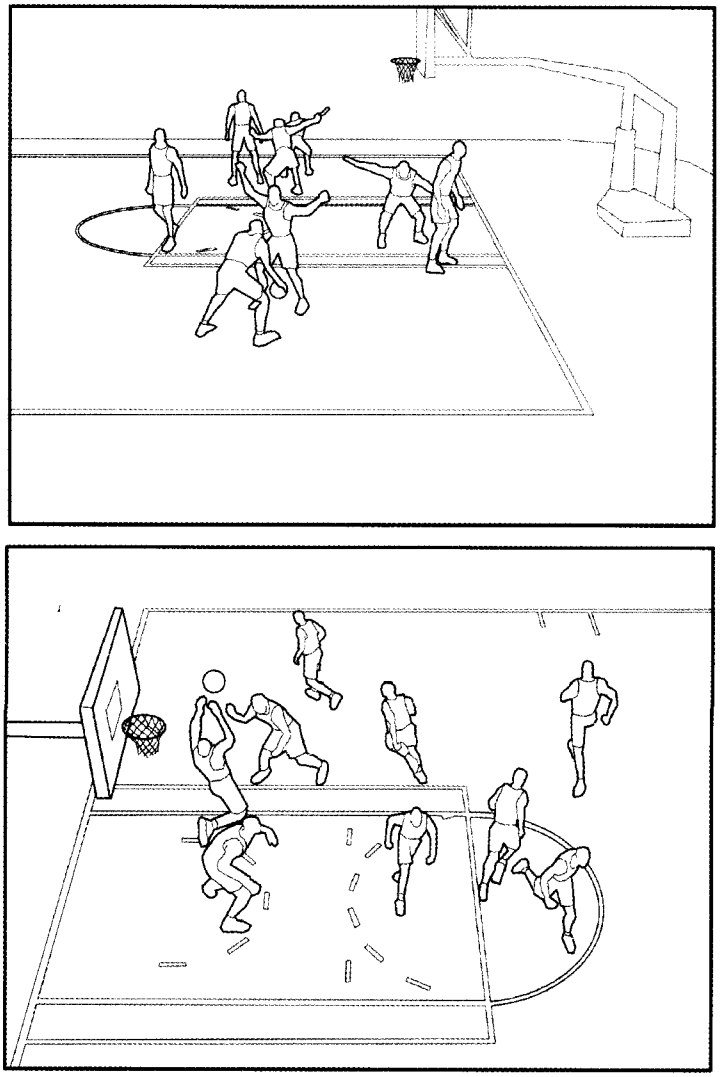


图 10-1 运动类游戏的不同摄像机样式能够影响玩法

10.1.5 混杂元素

混杂元素包括拉拉队队长、吉祥物、兼职教练、人群以及所有构成运动类游戏中次要角色的东西。尽管它们通常是非常简单的 AI，但这些元素能够通过向玩家提供世界中的活泼元素(而不管是否与他进行直接交互)，使得游戏看起来更加真实。

10.2　有用的 AI 技术

10.2.1　有限状态机与模糊状态机

属于“可重设玩法”类型的游戏比那些更加动态的游戏更容易适应纯粹的基于状态的 AI 模型。然而，所有游戏都遵循一个设定的游戏流程(甚至篮球游戏也有中圈跳球、发边线球、对战和罚球等状态)，并且该游戏流程级别中 AI 的结构是基于状态的。但在该游戏流程内的某些状态里面，教练和玩家必须做出的决策是非常不清楚的。确实，必须要在运动类游戏的几乎每个层级上都进行模糊决策，并且 FuSM 可以用于提供该类型的模糊决策。

另一种方式是在运动类游戏的感知层级上使用一定程度的模糊性。状态本身必须是清晰的，但对状态的激活则将带有一些模糊度。因此，关于是否有开阔的视野来实施一个绝好的射门的状态变量将需要在它的计算中带有一些模糊性，从而清晰的“击球射门”状态将只有在这个更加模糊的判决中才能被激活。

程序清单 10-1 包含了 Sony 公司的篮球游戏 NBA Shootout 2004(PS2)中的一些示例代码。这些代码展示了持球的 AI 玩家能够执行的部分高层行为状态(大约是 10%)。该系统采用一个层次化的 FSM 来进行设计。

程序清单 10-1　NBA Shootout 2004 中 FSM 行为示例

(Code © Sony Computer Entertainment America。未经允许，不得翻印)

```
//--------------------
//--------------------
//AlleyOop
//--------------------
//--------------------
void gAlleyOop::Update(AIJob* playerjob)
{
   playerjob->ShowGoalLabel("Alley Oop");
}
bool gAlleyOop::GetPriority(AIJob* playerjob)
{
   bool doTheOop = false;
   int shotDistanceType = playerjob-> m_pPhysic-> GetShotDistanceType();
   t_Player* oopPlayer = NULL;
   if((fmodf(GameTime::GetElapsedTime(),BP_ALLEY_OOP_INTERVAL) <
     GameTime::GetDeltaTime())&& Random.Get(BP_ALLEY_OOP_CHANCE) &&
     (( shotDistanceType == t_BallAI::distance_outside ) ||
       (shotDistanceType == t_BallAI::distance_three_point ) ) )
   {
      AlleyOopCoach.SetPasser( playerjob->m_Player );

      if((oopPlayer = AlleyOopCoach.FindAlleyOopReceiver())!= NULL)
      {
         if( oopPlayer->GetBallHandlerJob()->
               GetNumberOpponentsLineOfSightColumn( Basket.
```

```
                GetPosition(), BP_LINE_OF_SIGHT_WIDTH ) <= 1 )
            doTheOop = playerjob->m_Player->
                    GetBallPlayerSkill()->AlleyOop(oopPlayer);
        }
    }
    return doTheOop;
}
//--------------------
//--------------------
//LastDitchShot
//--------------------
//--------------------

void gLastDitchShot::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Last Ditch Shot");
    Team[playerjob->m_Player->team].ClearMiniPlay();
    ((BallHandlerJob*)(playerjob))->DoShootBall();
    return;
}

//--------------------

bool gLastDitchShot::GetPriority(AIJob* playerjob)
{
    if( Court.IsBehindBackboard(playerjob->m_Player) )
        return false;
    if(Team[playerjob->m_Player->team].m_humanOnMyTeam &&
    playerjob->m_Player->GetBallHandlerJob()->m_justReceivedBall)
        return false;

    // last ditch effect
    return( GameState.GameClock.GetTime() <= 2.0f ||
                GameState.ShotClock.GetTime() < 2.0f );
}
//--------------------
//--------------------
//FastBreak
//--------------------
//--------------------

void gFastBreak::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Fast Break");

    //try passing, it won't do it if it cannot
    ((BallHandlerJob*)(playerjob))->DoFastBreakPass();

    Vec3 basket = Basket.GetPosition();
    Vec3 target;
```

```
    target.x = (playerjob->m_pPhysic->position.x+basket.x)/2.0f;
    target.y = 0.0f;
    target.z = (playerjob->m_pPhysic->position.z+basket.z)/2.0f;

    playerjob->m_pPhysic->SetDestDirection
                    ( Basket.GetPlayerDirection(playerjob->m_Player) );
    playerjob->m_pPhysic->SetTargetPositionBallHandler( target );
    playerjob->m_pPhysic->SetCPUGotoAction( PHYS_TURBO );
}

//--------------------

bool gFastBreak::GetPriority(AIJob* playerjob)
{
    if( !GameState.isFastBreak )
        return false;

    if(playerjob->m_Player->GetPlayerSkill()->m_inCollision )
        return false;

    return true;
}
//--------------------
//--------------------
//LongHold
//--------------------
//--------------------

void gLongHold::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Long Hold");

    t_Player* passTo = playerjob->m_Player->m_pBestPassTo;
    int chance = (Basket.GetPlayerDistance(playerjob->m_Player) >
                FEET(15.0f) && playerjob->m_Player->
                m_pHasDefenderInPlace)? 90 :playerjob->m_Player->
                Personality->passes ;
    bool wouldPass = Random.Percent( chance );

    if( passTo != NULL && wouldPass)
    {
         GoalOffPass.Update(playerjob);
    }
    else
    {
         ((BallHandlerJob*)(playerjob))->DoJumpShot();
    }
}

//--------------------
```

```
bool gLongHold::GetPriority(AIJob* playerjob)
{
    //the point guard on the initial bring up
       //shouldn't be limited as much
    if(Rules.shotClock == LowmemGameRules::ON && playerjob->
          m_Player->position == POINT_GUARD &&
          GameState.ShotClock.GetTime() > 9.0f)
       return false;

    Time stillTime = 0.0f;
    stillTime = playerjob->m_Player->GetBallPlayerSkill()->
                m_ballHoldTimer.Get();

    Time decisionTime = lerp(playerjob->m_Player->Personality->
                             dribbles/100,3.0f,5.0f);
    if(playerjob->m_Player->m_isOut)
    {
       if ( GameState.period >= 3 &&
          GameState.GameClock.GetTime() < 60.0f)
          decisionTime = 60.0f;
       else
          decisionTime = lerp(playerjob->m_Player->Personality->
                              playsPerimeter/100,3.0f,6.0f);
    }

    if( Court.IsInKey( playerjob->m_Player ) )
       decisionTime = 1.5f;

    bool result = false;

    if ( stillTime > decisionTime )
    {
       dbgprintf( "Long hold timeout: decision - %f still - %f\n",
                  decisionTime, stillTime );

       result = true;
    }

    return result;
}

//--------------------
//--------------------
//OffPass
//--------------------
//--------------------

void gOffPass::Update(AIJob* playerjob)
{
```

```
    char msg[80];
    sprintf(msg,"Offense pass, chance:%d",chance);
    playerjob->ShowGoalLabel(msg);

    t_Player* m_passTo = playerjob->m_Player->m_pBestPassTo;
    //if invalid, try the team stuff
    if((!m_passTo || m_passTo == playerjob->m_Player))
         m_passTo = Team[playerjob->m_Player->
                              team].m_bestPlayerToShoot;
    if(!m_passTo || m_passTo == playerjob->m_Player)//failsafe
         m_passTo = playerjob->m_Player->
                GetClosestPlayerToPlayer(playerjob->m_Player->team);

    if(m_passTo && (((m_passTo==GameRules.LastPossession.player) &&
    (playerjob->m_Player->GetBallPlayerSkill()->
    m_ballHoldTimer.Get()>1.0f)) ||
    ( m_passTo != GameRules.LastPossession.player ) ) )
    {
      playerjob->m_Player->GetBallPlayerSkill()->PassBall(m_passTo);
         playerjob->m_Player->GetOffenseSkill()->
                   m_targetTimer.Clear();//go back to where ya from
    }
}

//--------------------

bool gOffPass::GetPriority(AIJob* playerjob)
{
    //if nobody to pass to...
    if(!playerjob->m_Player->m_pBestPassTo)
       return false;

    if(playerjob->m_Player == playerjob->m_Player->m_pBestPassTo)
       return false;

    chance =0;
    if(playerjob->m_Player->IsInsidePlayer())
    { //inside players
      if(Basket.GetPlayerDistance(playerjob->m_Player) <=
              FEET(2.0f))
           chance = 10;//basket is close
      else if(playerjob->m_Player == t_Team::m_pDoubledOffPlayer)
           chance = (playerjob->m_Player->position==CENTER)?
                       70:80;
      //double team
      else if(playerjob->m_Player->m_pHasDefenderInPlace)
      {
           if(playerjob->m_Player->GetPlayerSkill()->m_canDribble)
           {
                if(playerjob->m_Player->Ratings->insideShooting<75)
```

```
                    chance = (playerjob->m_Player->
                              position==CENTER)? 60:50;
                    //covered, can dribble, low inside shot
               else
                    chance = (playerjob->m_Player->
                              position==CENTER)? 20:40;
                    //covered, can dribble, high inside shot
          }
          else
               chance = (playerjob->m_Player->
                         position==CENTER)? 50:70;
               //covered, can't dribble
     }
     else
          chance = 10;//not covered (or dteamed, or really close)
}
else // outside players
{
     if(!playerjob->m_Player->GetPlayerSkill()->m_canDribble)
          chance = 100;//can't dribble
     else if(playerjob->m_Player->m_pHasDefenderInPlace)
          chance = 30;//covered
     else
     {
          if(!playerjob->m_Player->m_pHasDefenderInPlace)
               chance = 10;//wide open
          else
               chance = 30;//not covered, no lane
     }
}

//offset for longer holds, greater increase if
//you're inside or can't dribble
float modVal;
if(playerjob->m_Player->IsInsidePlayer() ||
   !playerjob->m_Player->GetPlayerSkill()->m_canDribble)
{
   modVal = GameTime::GetGoalDeltaTime();
}
else
{
   modVal = 0.1f;
}

float rem = fmodf(playerjob->m_Player->GetBallPlayerSkill()->
           m_ballHoldTimer.Get(), modVal);
int holdAdj = int(rem/GameTime::GetGoalDeltaTime());
chance += holdAdj;
//now check for tendencies
bool wouldI = Random.Percent( playerjob->m_Player->
```

```
                                        Personality->passes);

    return (wouldI && Random.Percent(chance));
}

//--------------------
//--------------------
//Dunk
//--------------------
//--------------------

void gDunk::Update(AIJob* playerjob)
{
    playerjob->ShowGoalLabel("Dunk");
    if(playerjob->m_Player->GetBallPlayerSkill()->DunkBall())
        playerjob->m_Player->Task.SetCPUSequence(TaskDoChargeMove);

}

//--------------------

bool gDunk::GetPriority(AIJob* playerjob)
{
    //don't try if you can't
    if(!playerjob->m_Player->m_canDunk)
        return false;
    //always dunk if you're wide open
    else if(playerjob->m_Player->m_laneCoverage <= 0.1f)
        return true;

    //otherwise, use personality
    return(Random.Percent(playerjob->m_Player->Personality->dunks));
}
```

10.2.2　数据驱动系统

由于具有大量的玩家和随时调用的剧本、大量的统计数据以及大量的动画，运动类游戏至少需要依靠一些数据驱动的 AI。另外，由于要实现更为真实的运动 AI 和在线游戏，数据驱动系统将使得对 AI 进行调整变得更容易，并用在线变化(它要么反映真实世界玩家的统计变化，要么是进一步的游戏平衡修饰)来对它进行更新。一般由数据驱动技术执行的部分事件如下：

- **剧本**。与为 AI 系统设计剧本不同，更好的系统是设计 AI 控制的玩家能够执行的自动行为，然后使用一个设计者能够用于将这些行为连成完整剧本的编辑器，从而为游戏中的球队设计剧本。这样，设计者能够试验不同的剧本并手动挑选出最好的几个(或各队在真实生活中最喜欢使用的那些)；AI 程序员则可以集中于额外行为，而不是设法调整硬编码的剧本。

- **动画挖掘**。通过详细给出(通过一个可视的编辑器或某种脚本工具)给定行为的最好动画的条件类型，设计者能够很快清楚地阐述对游戏内每个动作有意义的动画，并根据需要改变或扩展这些动画列表，而不需要进行任何的代码改变。
- **玩家统计**。在这一层级，玩家需要一些接近该层级的其真实生活对等物的统计数据，以及其他必须设计的游戏内部统计，从而能够以某种方式将各种属性与游戏模仿相关联。

10.2.3 消息系统

因为许多玩家之间需要进行相互通信，而且运动类游戏具有非常动态的环境，所以将消息系统引入运动类游戏的 AI 框架是非常有意义的。从两个玩家之间(或者是冲突事件)对游戏的协调，到对人类动作的注意，每件事情都可以通过消息系统来传递，而且 AI 只对它所感兴趣的消息进行响应，而不必连续监视整个运动场。不同层级的 AI 系统也能够使用相同的系统，因此物理层将响应冲突事件，而队层级则响应两个玩家之间的协调事件。

10.3 示例

早期运动类游戏，比如 Intellivision 和 Atari 上的 Football 和 Basketball，甚至不支持各队拥有完全数量的玩家。另外，它们也使用简化的 AI，这与必须要与之磋商的重要支持者非常相像。

随着程序员最终具有了让游戏接近比赛所必需的能力，运动类游戏开始真正地成为人们的焦点(即 NES 游戏系统)，尽管它们仍然处于较为原始的阶段。像 RBI Baseball、Tecmo Super Bowl、Ice Hockey 和 Double Dribble 等游戏至今仍受运动游戏迷的喜爱。这些游戏所采用的玩法非常简单，但已接近对实际比赛的模仿，并且我们终于开始了解统计学更重要的使用价值(而不是两个不相上下的球队彼此对抗)。

现在很多游戏，即便是具有很好的图形界面，也仍然采用了这个时期产生的大部分玩法规则。在某些方面，这使得运动类游戏在玩法进化上有些迟钝，但它具有让长期爱好者即刻参与大多数游戏的优势，因为他们对控制方案、整体游戏技巧和一般游戏策略都相当熟悉。在借用 Street Fighter 游戏的六按钮控制布局和特殊的操纵杆移动的格斗类游戏中也有相似的情形。

1990 年，几乎所有流行的游戏都出现了连续的周期性版本，现在已经到了 16 位和更高版本。随着游戏质量和领域的不断提高，并随着控制台开始使用更精密尖端的控制器，游戏赋予玩家对事件处理的更多控制。这意味着 AI 必须要跟进，从而增加了复杂度。

现在的运动类游戏是 AI 的奇迹，像 Madden、Sega's NBA、NFL 2K 系列以及 World Soccer 等不断出现的游戏都对各自运动进行了熟练的模仿，同时展现了玩家的个性并给游戏玩家带来了强烈的运动体验。这些游戏使用多种 AI 系统，包括调用剧本和对战术决定的复杂 FSM、基于某些因素选择正确动画的数据驱动系统、使得游戏角色像在真实生活中一样执行动作的熟练的仿真计算以及越来越多的使游戏更为真实和有趣的尝试。

10.4　需要改进的领域

10.4.1　学习

运动类游戏的 AI 继续成为开发利用的牺牲品，即使是最好的 AI 控制球队也会失败，因为人类能够做一些 AI 很难停下来的重复事情。如果 AI 能够通过明确地指向这种重复行为来进行补偿，那么它将迫使人类玩家或者改变他的游戏战术或者停止轻易得分。队友 AI 也可通过辨别人类喜欢采用的剧本从中进行学习，并且如果这种剧本再次发生，则会更好地进行支持。该类型的运动学习已经通过使用影响力地图(通过逐渐改变位置数据来反映更多的取胜局面)和统计学习(通过跟踪奏效或不奏效的行为，适当地进行调整)得到了实现。该系统不需要增加游戏的难度；它只是阻止毁坏 AI 系统整体性能的开发利用。最后，该系统将仅仅促使玩家更经常地改变其游戏计划，并且整体体验将更接近于真实比赛。当然，如果希望，相同的系统也可以用于增加难度，因为该系统能够很快学习那些人类难以停下来的事情，并偏向于选择此类行为(事实上，该系统在寻找对抗人类智能的行为)。

10.4.2　游戏平衡

运动类游戏经常遇到游戏平衡问题。某些运动相关的任务，比如篮球比赛中的防守，比其他任务更难执行(其原因是篮球比赛是一种快节奏的运动，防守动作是反应式的，因此总是比进攻要滞后)。我们怎么用它来支持人类(使该任务更具娱乐性)，而不破坏游戏的平衡(通过使它容易防守来阻止进攻)？随着这个问题继续发展，在具体问题具体分析的基础上，当 AI 程序员遇到需要基于手头的游戏及其娱乐因素的决策问题时，它将继续需要 AI 程序员做出努力。

在线游戏使得游戏平衡任务更为复杂。迄今为止，由于带宽限制和其他原因，在线游戏中就算是最快的连接也存在内在滞后。由于这个原因，运动类游戏中的高度反应性行为在视觉上就会受到困扰，并比 FTPS 等基于物理学的游戏(physics-based game)更加麻烦(后者具有非常简单的动画并能够使用物理学来预测角色并产生运动来填补滞后造成的空隙)。

因此，在在线运动类游戏中，游戏平衡问题与处理这种滞后的问题相关，尤其是在那种能相互摆脱同步或使用事件驱动的网络化游戏方法的游戏之中。如果玩家看到他接到了一个传球，但服务器认为他并没有，那么当该玩家突然不再持球时他会感到非常困惑。如果这种情况发生，则也应该将它忽略。但如果这是一个系统问题，游戏中的客户端不断地通过间歇动画、行为和位置来追赶服务器中的事实，那么这个游戏就不具有可玩性。

10.4.3　玩法创新

运动类游戏变得越来越相似，而且其游戏方式也日渐停滞。市场也驱使这个游戏行业中的高利润部门进行创新。甚至是 Madden 这个所有运动类游戏中最好和最成功的成员，在多年以来也没有进行任何的创新。Madden 球队已经改进了图形质量、展示和动画，并在界面上做了一些小小的变化。但游戏几乎与其最早的 Madden 足球游戏相同，玩法也一样。仅仅是更漂亮了。这是消费者所真正希望的吗？或者这是不是就是提供给消费者的东

西？当然，动机是不丢失任何市场份额(通过把人们吓走)，因为要么人们不会即刻喜欢游戏的玩法技巧或 AI 行为，要么不能足够快地学习。不管在营销学上怎么考虑，如果游戏体验足够好，人们将会购买该游戏并且事实上会花时间去学习一个新的界面或游戏技巧。当第一个篮球游戏出现时，没有人知道该怎么控制它，然而，消费者还是买了它。

在运动类游戏世界中存在大量创新空间，包括在玩法上和在竞争和协作 AI 上。我们必须努力向消费者提供一些新东西，以免该游戏类型开始变得陈旧。想象一下足球游戏中的 AI 系统，它在一群人中与玩家讨论事情并帮助玩家形成一个对抗别的球队的计划。想象一下一个讲解员 AI 系统，它进行一个电视式的慢动作，同时重点谈及屏幕上的播放和绘制事件。想象一下对这些游戏进行更直觉的语音控制，玩家可以“朝”某一玩家呼喊(通过头部运动跟踪或其他方法)并得到一个相应的响应。这些都是使该游戏类型保持新鲜和成长的东西。

10.5 小结

运动类游戏从不可思议的简单版本开始已经走过了很长一段路，第一个为家庭控制台设计的运动类游戏是在 1970 年。由于视觉和玩法变得更加真实，对高质量 AI 控制运动员的需求也变得更高。现在，运动类游戏是行业中最赚钱的游戏之一，并且花了钱的玩家在每一个元素上都要求有高质量。

- 两种主要的运动相关玩法是流畅和可重设游戏。流畅是指那些具有通常不停止玩法的游戏，具有非常动态的局面。而可重设游戏是那些周期性重启或停止动作的游戏，因此它更为线性。
- 运动类游戏的一般购买者通常都具有高层的运动相关知识，这意味着模仿类型的运动类游戏需要考虑更高层次的细节。
- 教练或团队级别的 AI 给系统提供了更为深远的决策过程，同时提供了在多个玩家之间协调动作的方法。
- 玩家级别 AI 系统通常比教练级别更加强调战术性，并通常包括决策和动画选择元素。
- 运动类游戏中的路径搜索涉及到更大数量的动态障碍物，并需要考虑根据具体游戏规则行进的方法。
- 动画挖掘系统对任意可能动作都包含大量动画的运动类游戏来说非常重要，因为系统需要有一种查询动画数据库和做出智能决策的快速方式。
- 混杂元素使世界不仅仅局限于游戏球场，并给玩家一种更强烈的身临其境的感觉。
- FSM 和 FuSM 在这些游戏中使用也很广泛。当使用这些技术的时候，游戏类型(流畅或可重启)有时也是一个要素，但由于所有运动类游戏的内在特性，某些程度的状态机在构造游戏时将非常有用。
- 数据驱动系统可以在玩家级别、剧本级别和动画级别减轻需要提出大量细节的压力。
- 消息有助于 AI 系统各层级之间的通信并提供了一种克服该极其动态的环境的快速方法。

- 学习有助于解决 AI 开发利用的问题，并能帮助玩家对系统进行学习。
- AI 系统需要扩展它们在需要强调游戏平衡和公平博弈的领域内的能力，因为往系统中添加过多的智能将给玩家提供更多的帮助，但是会破坏游戏平衡。
- 该游戏类型必须继续在玩法、对手和协作 AI 系统上进行创新，这样它才不会丧失元气。由于这些游戏的视觉效果几乎已经达到电视质量的程度，这变得更为重要，因为它们将不再是游戏的集中点。

11 赛车游戏

不管是从游戏玩法的角度，还是从 AI 的角度来看，赛车游戏(racing game)类型都是一个非常有意思的游戏类型。该游戏类型通常可划分成两个主要的类别，即车辆赛车游戏和专业赛车游戏。这两类具有相同的思路，至少在外表上都是对竞赛的物理模拟。

早期游戏，如 Pole Position(甚或是其更早版本，即 1974 年的 Atari 游戏 Gran Trak)，更多的是沿着动作类游戏的路线，因为那时硬件的处理能力不允许进行更多的仿真，而仅仅能给玩法系统提供娱乐。

是的，一些赛车游戏(甚至是一些现代赛车游戏)并没有认真对待它们的物理特性，但这正是视频游戏需要解决的东西，即坚持我们并不在意被限制住的真实领域，并抽象出我们所做的真实部分。因此，我们希望这些游戏能够大部分符合真实情况并得到处理，但同时我们也希望能够使汽车跃过 10 辆卡车并在着陆后还能照常行驶。这就像那些玩家一样，他们并不在意每次射击都要重装火箭筒，但如果一次只能携带三枚火箭，他们就会在意；他们希望背包中有一百枚火箭，而不管该负载是否已经超出了该角色能够携带的最大负载，更不用说跳跃的时候。

在早些时候，车辆赛车游戏出现了两个变种，而且其界限现在还在。它们通过摄像机的视角进行区分：第一/第三人称赛车游戏(如 OutRun 或 Stun Runner)和俯视游戏(RC Pro-AM 或 Mickey Thompson's Offroad Challenge)。俯视游戏往往偏离那些更具街机感觉的游戏，且具有很不切实际的物理特性；另一种类型则更忠实于其根源游戏。

专业赛车游戏很受人们喜爱，当时，它们包括了节奏感很强的赛车类运动。过去取得了一定成功的例子包括滑雪板、滑雪术、划船、冲浪、气垫船、越野自行车等。这些游戏需要通过具体运动相关的行为来增强传统的赛车 AI，例如实施诡计、处理未来、非传统的物理系统等。

最后一个子类型是手推车赛车游戏(由于 Mario Kart 游戏而变得流行，但从那时候起通过具有相当多的不同角色而取得了很大的成功)，它简化了游戏的操纵部分并增加了障碍物、陌生轨迹以及其他动作元素。

纯粹的车辆模仿是一项非常具有技术性的任务。需要相当复杂的数学解答能力来处理不同的悬浮系统、良好的多体碰撞处理机、能够适应不同道路条件的 AI 对手(尤其是在道路外的赛车游戏或在有下雨、加油和结冰危险的游戏中)，以及其他游戏可能带来的特殊关注点。

一些最好的赛车游戏已经成为新的游戏系统开始发布时其计算和图形能力的展示平台。这是因为这些游戏使用的物理模型和控制方案现在已经高度精炼，几乎不需要作任何

的调整，我们仅需要设计一个很好的图形引擎，抽象出一些高质量的汽车模型，然后就得到了一个做好的高质量的可以投放市场的游戏。

整体而言，纯粹赛车游戏的 AI 在这些年来已经变得非常先进，有许多非常棒的轨迹 AI 的例子，它们不需要作弊就能完成很有竞争力的工作。实际上，由于缺乏新意，赛车游戏类型已经开始不再那么流行。大量游戏涌现出来，它们主要的操纵模仿做得非常好，并且接近真实，以至于几乎不可能超越它们。该游戏类型需要有一个刺激因素来使它复兴。

Twisted Metal 游戏于 1995 年发布，从而产生了第一个真正的车辆战斗游戏(尽管其他具有汽车和武器的游戏发布得更早，但它们通常更像是卡通，比如 Mario Kart，或者只是普通的动作游戏，比如 SpyHunter，因此它们没有真正的操纵模仿，但它们的确对该游戏类型有很大影响)。Twisted Metal 游戏在当时是一个适度真实的操纵模仿，外加竞技式的关卡和额外赠送的一系列武器。人们并不计较它不够标准的图形质量和非常陌生的控制设置(因为附加玩法元素非常原始)，并且玩得非常尽兴。然而，这并不够，主要是因为单人体验必须要忍受糟糕的 AI(包括性能和难度级别)，而且当玩家仍与同一个对手对抗时该游戏玩法将非常重复(无意义地对坐在身边的伙伴唠叨不休，并当他被杀死时听到他的尖叫声，似乎对所有游戏都增加了重玩值)。其他游戏也相继出现，包括流行的 Interstate' 76，它增加了线性故事的概念并且人们对它的整体评价还不错。但它也受 Twisted Metal 游戏的重玩游戏和单人问题的困扰。此外，该类型游戏还需要更多改进。

近来，似乎开始有了转机。通过一步步深入，并对赛车游戏类型添加除了武器之外的复杂冒险和故事元素，这些游戏真正为赛车游戏打开了新的局面。Grand Theft Auto 出现在 1997 年，它是一个相当原始的俯视二维游戏，且具有非常简单的概念：提供一个逼真的城市，在城市中玩家可以执行包括驾驶在内的许多不同活动，从而像杀人犯那样生活。它将这个概念保持了很多年，但后来转移到一个十分壮观的完全真实的 3D 世界，具有真实的操纵模仿(或许是过多操纵模仿)，并具有所有人们能想象到的色情、暴力和摇滚乐。它也是一直以来最畅销的游戏之一；至今为止，该系列中的 4 个游戏已经有超过 2500 万的拷贝。开放玩法和成熟内容相结合被证明是非常受欢迎的。许多其他游戏后来利用了这种规则，因此全盛的车辆动作游戏类型再次崛起，而这正是赛车仿真和战斗游戏已经放弃的领域。这些游戏中的动作元素非常广泛，进入了冒险游戏或第一/第三人称射击(FTPS)游戏的领域，但主要玩法系统还是车辆，或至少到现在为止是这样。

11.1 通用 AI 元素

11.1.1 轨迹 AI

轨迹 AI(track AI)是一个在路面与车轮接触的地方保证 CPU 控制的汽车以很高的速度在跑道上(或城市街道)行驶同时遵守游戏规则的系统。通常，它是基于状态的系统，用不同的车辆状态来详细设计保持赛车手在轨道上的主要方式(多数可能是 OnTrack、OffTrack、WrongWay 和 Recovering，或其他相似状态)。每个车辆状态都有多种操纵和应用油门和刹车来最好地适应车辆所处的特殊状态的方式，并与车辆在轨道上的位置和与其他司机的关系相结合。作为指导方针，大多数游戏都使用物理学和“最优行进路线”(它或者是在轨迹

编辑器中设计的数据路径，或者是由一种称为“寻找最小曲率路径”技术来自动计算，如图 11-1 所示)的组合，这是对人类在轨道和道路上赛车时所使用的无形行进路线的模仿。另外，如果真正的最优位置已经被占用，那么还有最优的分支位置。

还要注意有些赛车游戏不是在道路上进行的，而是在水上(通过小船或喷射滑行装置)、雪山上(通过滑雪板)甚或是更奇异的地形上(如 Stun Runner 游戏中的管道和斜道)。因此，它们可能不会使用一个纯粹的最小曲率技术，因为其表面的动力学特性可能需要其他类型的最优机动。

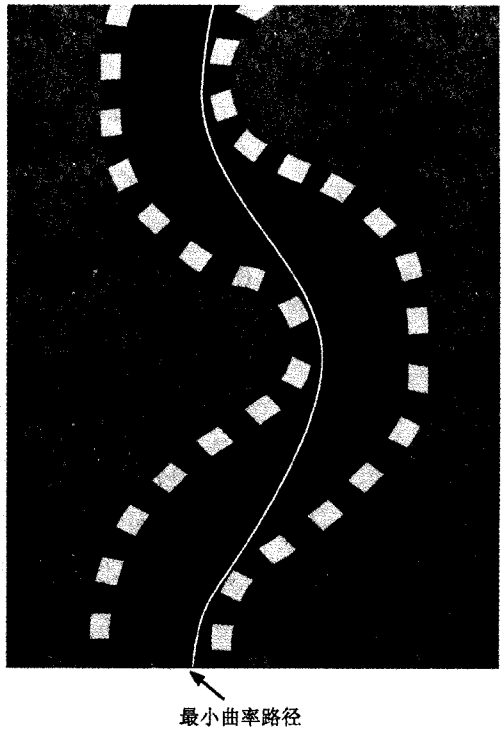


图 11-1　最小曲率路径的轨迹

11.1.2　交通

很多此类游戏都在功能城市上进行赛车，因此它们必须要与交通系统打交道，包括停止行进信号、公路系统以及使用它们的大量汽车。游戏中的交通一般看起来都很好，但很少对玩家的运动进行反应(实际上，这可能是专门设计的，因为玩家不希望每个人都给他让路)。

然而，一些游戏使用了非常复杂的交通系统，它们非常真实，拥有变化的车道、超越警车的汽车、正确使用交通灯和十字路口的情况等。这主要是 FSM 行为，通过大量的消息传递来确保不发生事故(除非一些吵闹者恰好碰上了 130km 的时速或其他)，并使用一些随机性来确保事情不会变得重复。

11.1.3 行人

自从赛车游戏开始以城市为背景，就一直需要对行人进行处理。不同的游戏对行人有不同的处理。Midtown Madness 游戏做得较好，它们简单地让行人随机地在路上走，如果有车靠近，他们便从道路中跳离。但一些游戏，如 Grand Theft Auto 或 Carmageddon，允许用户几乎可以撞倒他希望撞倒的任何人。行人需要设法避开道路，但机灵的暴力无赖会找到一条道路从而撞倒行人。实际上，Grand Theft Auto 游戏具有非常广泛的行人类型，他们都根据各自功能运行着不同的 AI。在大多数游戏中，该行为是基于状态的，或许还使用一些消息。

其他系统使用非常简单的群聚式行为，使关卡中的区域具有特殊的吸引力和排斥力(因此，某些店面可能吸引人们，他们将从窗户外看一会，然后朝下一个引起注意的地方走去；而死人可能是一个很强的排斥力，因此人们都尽量避开事故)。紧急状态(State of Emergency)在一些类似的系统中取得了成功。人群非常具有流动性并能够对大多数的动作做出很好的反应。

11.1.4 敌人与战斗

这是死亡竞赛机器人代码中的汽车对应物，因为有些游戏允许完全的战斗，可以是汽车对汽车、行人对汽车或任意的其他组合。这个代码需要将前面提到的赛车 AI 和 FTPS 游戏中的机器人 AI 相结合，包括人类层级的性能检测，即执行一些诸如使 AI 发动不起来和偶尔撞墙等事情，以确保玩家不会有被欺骗的感觉(或仅仅让他被一个无情的邪恶机器人追赶，如果这是游戏开发者的意图)。它也可能包括多辆汽车协同工作，如警车在不同的街道上切断逃跑的多条路线，或两辆车对玩家进行夹击从而使他不能转弯。

11.1.5 非玩家角色

非玩家角色(NPC)是游戏中必须要处理的非战斗人员，比如那些向玩家提供信息的人或者卖给玩家更好的汽车的人。与 RPG 游戏中的类似角色一样，这些角色通常具有脚本化的行为和对话，以便于他们的相遇。他们通常不是反应性的，因为大多数此类游戏都不具有高度发展的会话引擎(这并不是该游戏类型的关键，如果人们希望那样，那他们可以去玩 RPG 游戏)，因此大多数 NPC 都仅在非交互式的剪辑场景中进行操作。

11.1.6 其他竞争行为

有些赛车游戏也需要来自其 AI 对手的其他行为，比如在滑雪板游戏或摩托车越野赛游戏中施加诡计。这些系统需要脚本化的一连串招式(总体看起来非常不错)或者对物理学和时间选择有一个足够好的理解(从而可以选择能获得成功和流行的招式)。该类型的决策结构实际上更像是一个格斗类游戏，每个动作都与某个时间相关联(即该动作需要执行多长时间)，并且 AI 将根据动作的持续时间(通过考虑可以达到的速度和高度的简单物理学计算)、技能级别和个性来确定所要选择的动作。

11.2　有用的 AI 技术

11.2.1　有限状态机

赛车游戏具有相当简单的 AI 设计，大多数都由物理定律以及当前“比赛”的简单目标(最先到达终点线或捡起某个包裹并把它带回的同时又躲过其他玩家的攻击)来定义。另外，大多数经典赛车游戏的游戏流程的状态设计也非常直接(开始、赛车、出轨、追赶、调速、停在加油区)。FSM 在这里再次起到了非常重要的作用。

11.2.2　脚本系统

车辆动作游戏类型通常遵循某种故事情节(尽管有些是极其开放的)并可以与脚本范例很好地结合。另外，周围的行人和交通系统可以有助于脚本系统，其不同的运动模式被脚本化并与城市的街道设计相结合。有时，这仅仅是第一层，具有合适的可重载反应系统，从而以特殊方式影响这种脚本化行为。因此，如果让一帮人在商场里漫无目标地乱转、对商品进行付账、使用自动扶梯等，那么这将是一系列使各 AI 控制的玩家看起来都具有熟悉环境知识的很小脚本。但如果一辆汽车突然从窗户中撞出去，行人将不具备逃避行为，而必须推翻标准脚本，用一种狂热的冲撞来逃避被碾碎的可能。

11.2.3　消息系统

周围的交通和行人系统一般都使用消息系统来进行相互之间的谈话，并以某种在真实生活中发生的复杂方式对运动进行协调。当然也可以利用 FSM 来对该类型行为进行编码(即使使用消息系统，也可以用脚本或状态机来控制交通和行人的整体行为)，但如果面临的是大量的周围车辆和行人(并且希望他们单独或协调地对周期性事件或特定事件进行响应)，那很可能只有选择消息系统。

11.2.4　遗传算法

某些此类游戏具有庞大数量的汽车(比如，Gran Turismo 2 游戏有超过 500 辆的汽车)，每一辆都需要调整其操作和性能，因此有的公司通过使用一个简单的离线遗传算法来修改汽车的性能参数直至获取最优的结果，从而自动完成了该调整任务。之后，对这些结果进行存储并将它们直接用于实际玩法之中。这是此类技术的一个非常简单的作用(作为调整系统参数使其最优化的预处理器)，并且遗传算法在进行这些计算时所花费的时间要比程序员或设计师在游戏中使用试凑法花费的时间少得多。

11.3　示例

驾驶游戏从一开始就伴随着我们，其中最早的一批出现于 20 世纪 70 年代早期。这些早期的驾驶游戏只是保持车辆在两条滚动的线条之间运行。但在给定方向盘和油门的情况下，这种简单的表示就是所有需要考虑的东西。

驾驶游戏已经走过了很长一段路，与现在几乎是电影质量视觉效果的 Gran Turismo 游戏相比，较老的 Pole Position 和 SpyHunter 游戏看起来非常过时。另外，过去街机风格和快速松散的游戏玩法几乎迷失在如今更好赛车游戏中接近完美的操作和性能建模之中。然而，并不是 Crazy Taxi 之类的游戏缺乏真实性。Midtown Madness 游戏给我们带来了非常棒的城市交通；The Simpsons：Hit and Run 游戏成功地将游戏模型扩展至滑稽领域，并成功地保持其喜剧性；Interstate '76 游戏将流行样式和好的故事融入游戏之中；Carmageddon 游戏则实际上给我们提供风档刮水器用于清除血迹。

11.4 需要改进的领域

经典赛车模拟游戏已经非常成熟。如果设计的赛车模拟游戏并不具备一个构造得很好的可靠物理模型以及与其结合的精炼直觉的控制方案、超现实主义的视觉效果以及一些可以将游戏与其他已经具备上述特点的游戏相区别的方式，那么甚至不用将它上市。然而，将车辆赛车与玩法中的其他元素相结合的新变种仍然具有许多改进的空间。

11.4.1 除犯罪以外的其他感兴趣领域

要尽量推动游戏朝更主流(考虑到该游戏类型已经售出数百万游戏，很难想象主流是什么样子)、更能为父母所接受(并不是每个母亲都希望看到她的孩子为了钱而去充当妓女)的方向发展。视频游戏中的暴力的确有市场，但它不需要像 Grand Theft Auto 游戏那样走极端。

11.4.2 更多的智能 AI 敌人

想象一下自己正在被一群 AI 汽车追赶，但它们很少协同工作来设置路障并对自己进行拦截，相反，这成了布鲁斯兄弟(Blue Brothers)式的追逐赛，即最前一辆车被 40 辆警车追踪。这非常接近于该游戏类型的一般情况，但需要对对手使用更复杂的机动。只要给人类“罪犯”玩家一个警察扫描器，他就能稍微提前知道该路障并避开追捕。有的游戏在该领域中使用车间距离概念，但很少。

其他问题可以通过一些游戏跑道上的超车机动表现出来。在有些游戏中，AI 控制的车辆很少注意其他 AI 控制的车辆，它们确实在一定程度上对它们的速度和转弯进行调整，但 AI 车辆之间碰撞的调整是尽量减小它们对每辆车的影响，从而简化整个赛车模拟。因此，有的游戏中的 AI 车辆相互之间并不采用真正的超车动作——一辆车以一种从远处看起来非常好的方式把另一辆车撞翻到赛道之外，但却并没有接近。为什么不给每辆车一个更真实的 AI 比赛模型，从而让人类注意不到这种 AI 作弊呢？

11.4.3 永不落幕的游戏世界

车辆动作游戏目前还不适用于多人在线模型，但这对该游戏类型来说有很大的好处。想象一下基于 Autoduel 的游戏世界(1985 年由 Origin™ 出品，基于 Steve Jackson 的纸笔 RPG

游戏 Car Wars，在文明世界崩溃后有点像是 Mad Max)或 Grand Theft Auto 的游戏与多人在线模型相关的一切。此类故事世界的动态性非常有利于赛车的玩法技巧，因为赛车具有在线游戏所需要的大且开放的游戏世界。

问题在于简单的计算能力。解决车辆模拟中的复杂数学以及管理整个城市的交通 AI(而不是以玩家为中心的一个球形区域内的交通，如 Midtown Madness 游戏中使用的那样)并不能很好地与 Internet 的有限带宽性能相结合。在线游戏的这种不对称(在某些游戏类型中这点稍微能够忍受并且可以进行补偿)可能使得游戏不具有可玩性。我们应该研究能否突破这些限制，并将赛车风格的玩法带入到在线游戏中。

11.5　小结

赛车游戏始于 20 世纪 70 年代街机中非常简单的游戏，一直发展成为如今图形化和技术性健全的游戏。然而，质量的快速上升是以玩法创新为代价的，而且该游戏类型几乎已经停止了发展。额外的玩法元素与赛车游戏的现代融合极大地鼓舞了该游戏类型并赋予了它全新的生命力。

- 赛车游戏类型被普遍定义为使用某些赛车物理模型的游戏。
- 车辆赛车游戏包括了更普通的车辆类型：汽车、摩托车、F1 赛车手等。这些车辆既可以运行于公路上，也可以运行于公路外，并且拥有一个实际赛道，或在城市或其他场所进行。
- 专业赛车游戏涉及其他类型的竞赛，如喷射滑雪游戏、滑雪板游戏等。
- 车辆战斗游戏的产生增加了该游戏类型的玩法潜力，而且最终添加了冒险和动作元素，对车辆动作游戏进行了扩展。
- 轨迹 AI 是当比赛超出了物理系统的界限和游戏的规则时 CPU 控制的赛车手维持控制的系统。
- 对于在城市范围内进行的游戏来说，交通和行人系统极大地增强了城市的视觉效果和局面的现实性。
- 在使用超越竞赛的额外玩法元素的游戏中需要有战斗 AI。
- 如果游戏使用除了战斗或专门的经济体或信息之外的额外角色交互，那么就需要使用 NPC AI。
- 如果游戏在比赛中包含有施加诡计或其他的动作，那么其他竞争元素也需要 AI 操作。
- 由于大多数比赛场景的线性特性，FSM 在这种游戏类型中非常有用。
- 脚本非常有助于车辆动作游戏的故事情节以及交通和行人系统的特性。
- 消息将使得复杂比赛和交通 AI 系统中游戏元素之间的通信需求变得简单。
- 遗传算法有助于对较大的赛车游戏所需要的大量汽车的操纵和性能参数进行自动调整。
- 除了犯罪之外还需要为车辆动作游戏开发其他感兴趣的领域，这将继续推动游戏的大众吸引力并让母亲们对孩子玩游戏感到放心。

- 对手AI需要额外的智能，因为其穿越城市并在赛道上超车的能力仍然比不上人类。
- 在此游戏类型中，具有永不落幕游戏世界的游戏能够通过很多事情来对它进行扩展。

12 经典策略游戏

博弈论(Game Theory)可以粗略定义为："在处理交互时所进行的对人类行为的学习，该交互的结果取决于两个或多个对立的或至多是混合动机的人的策略"。事实上，1928 年，John von Neumann(约翰·冯·诺依曼)就通过研究纸牌游戏中的欺骗概念，发现了该分析对经济学有重大的影响，从而开创了这片领域。

在博弈论中，game 这个概念具有特殊的意义。与更普遍的娱乐导向定义不同，博弈论将 game 定义成"几个智能体通过采取某些行动来最大化其效用的任务，但其结果取决于所有玩家的行动"。通过发现此概括存在于不同类型的 game 之中，博弈论希望能够解释很多不同领域内的人类交互，从商业到战争，从西洋跳棋盘到人口过剩。

博弈论研究的一些经典 game 包括：门口的野蛮人(Barbarians at the Gate)、核战阴影(Mutually Assured Destruction)、囚徒困境(the Prisoner's Dilemma)和货物出门概不退换(Caveat Emptor)。它们都是数学上的构造，试图定义不同人类行为的所谓的支配性策略。

在冯·诺依曼的一些初期工作中，他给出了一个需要一个很大必要条件的非常重要的发现。该发现是：对于某些博弈，在给定策略和游戏内在效用的情况下，"理性"(指采取的最好行动)是可以从数学上进行计算的。必要条件是：博弈是所谓的"零和博弈"，即"一个玩家的取胜行为将直接导致另一个玩家的等价损失"。换句话说，这些游戏中有许多玩家参与一个完全竞争的系统，该系统中只有一个赢家。这并不是一个微不足道的要求。博弈论希望解决的许多社会中的非常重要的问题(比如处理自然资源使用的经济学，以及政治系统)，都不属于零和博弈。尽管博弈论也能够洞察这些另外类型的博弈，但它不能像在零和博弈的有限世界中那样定义具体博弈相关的理性。

冯·诺依曼的工作是每个早期 AI 研究者工作的基础，因为他们计划设计一些需要理性来完成复杂任务的程序。怎样最好地测试他们的设计，才能比寻找一些对世间问题的抽象描述更好，同时又能设法很好地适合于简洁数学模型，从而可以确保理性呢？零和博弈回答了这个问题，并且到现在为止它仍然还是所有 AI 问题中研究得最多的问题。

经典策略游戏，如国际象棋、西洋跳棋、井字棋和纸牌游戏等，都是零和博弈的例子。另外，像 Monopoly 游戏这样的非零和博弈(这里，两个人有可能形成一个联盟，以实现双方都从银行"赢得"钱)也可以通过把其中一个玩家看作是棋盘本身(在 Monopoly 游戏中是 Bank)来将其转化为零和博弈。这个虚构的玩家本质上损失了玩家所赢得的效用数量，因此关于零和博弈的所有形式假设和证据都可以使用。

几乎当计算机开始出现时，研究人员就开始利用计算机来构建能够玩这些游戏的“智能程序”。1950年，Alan Turing(因为图灵测试而名声大震)和Claude Shannon编写了几个最早的国际象棋程序，这仅仅是在世界上第一台电子计算机ENIAC出现后的第5年。两个人都提出，一个能够胜任这些游戏的程序概括了对一些需要(或体现)智能的事件的定义。

这带来了关于一般的AI问题的一个有趣的比喻。过去，如果一个任务很难由计算机来解决，人们就会说如果谁能设计一个程序来实现这个任务，那么这个程序就是智能的。但在经过多年的工作后，当某人最终发布了一个能够执行该任务的程序时，那些恶意批评者又宣称这仅仅是一种简单的强力搜索(或程序使用的计算机技术)，并不是真正的智能。因此，AI从来没有实际解决过任何问题。

由于多种原因，研究人员开始将目光转移到游戏中。游戏更加复杂，并比所谓的“玩具”问题更有助于真实世界情形，同时比诸如旅行商问题(The Traveling Salesman Problem)之类的大规模搜索判决或集成电路设计表示了更多的不确定性和激动人心的世界。经典策略游戏也对经典AI搜索技术的最优条件进行了人性化处理。这些游戏具有完备的信息(双方玩家都了解游戏世界中的所有事情)，它们的走步通常是全局有效的(而不是在一个很小的影响半球内)，并且通常是回合制，从而给计算机留出思考时间。策略游戏同时也是非常复杂的(状态空间很大)，因此需要智能的方法来找到理性的解决方案。当前，正是属性列表造就了一个好的计算机AI模拟。然而，由于这些游戏也增加了对手元素，因此它们产生了一个具有不确定性的元素问题，以及更明确的有指导的不确定性。无指导的不确定性可以是由骰子或其他相似方法引入的随机性，因此是无偏的且仅仅是部分游戏代价。但有指导的不确定性用于处理欺骗等事件，混合之后的策略表现为随机，并且不管什么原因都采用无理性的走步。

如果考虑以前提到的解决AI问题的最优条件，就很容易确定策略游戏的哪些部分对AI系统来说是薄弱的。封锁的国际象棋最后阶段(“封锁”涉及到棋盘中间的多个互锁棋子的状态，见图12-1)对于传统的AI系统来说是极其困难的。原因何在？在效果上这些走步不再是全局的。通过在另一侧使用牵制性的走步，从而让AI系统觉得某些事情要发生，我们可以突然将棋盘分成独立的几块并摆脱计算机。像这样的策略是Gary Kasparov击败许多计算机国际象棋程序的方式之一(当然也因为他是国际象棋历史上最好的玩家之一)。

时间限制将大多数理论研究从更传统娱乐形式的经典游戏程序中分离开来。当给定一个不切实际的无限时间请求时，将始终能够找到最优的解决方案。但当给定真实世界限制后，游戏程序总是具有某种形式的时间限制，并且我们必须对分配给我们的时间进行处理。当然，随着计算机速度的提高，当强力搜索变为可能时，我们将越来越接近这个最优点，甚至是给定适度的时间约束。但总是还有另外更复杂的游戏，它们将迫使AI研究者使用替换方法来更快地找到更好的解决方案，而不需要依赖整体的强力搜索。

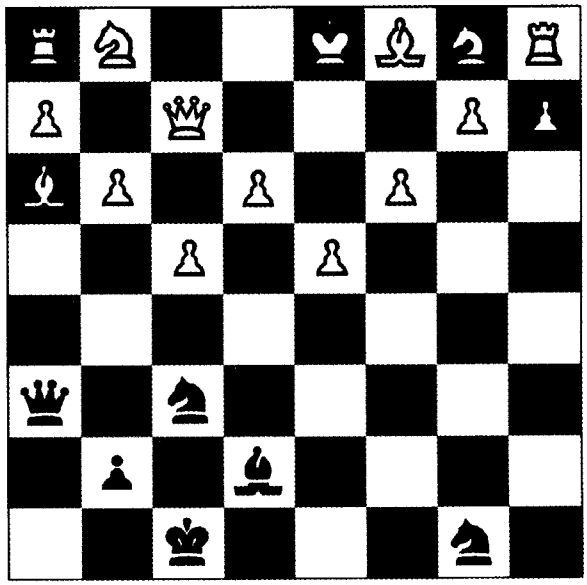


图 12-1　封锁的国际象棋游戏位置

AI 研究者已经“解决”了这些游戏中的几个，这意味着整个状态空间已经有了计划并能轻易地被当今的计算机搜索到，从而带来最优性能(对最开始走步的玩家来说是获胜，或者是平局)。已经解决的游戏有：井字棋、西洋跳棋、Connect Four、GoMoku 以及奥塞罗等。有的其他游戏正处于被解决的不同状态之中。国际象棋也正接近被解决。最高级别的国际象棋程序使用一个存储的“开放知识(opening book)”(经过几世纪来国际象棋大师们研究出来的为了获得好的表现而采用的一系列走步)，为瞬间中间游戏阶段采用一个智能的搜索技术，然后为最后阶段采用另一个存储走步的数据库。参见图 12-2，它给出了已经解决的和部分解决的游戏列表。

已经解决的
- Awari
- Connect Four
- GoMoku
- Hex(至 9×9)
- Nim
- Nine Men's Morris
- Three Men's Morris
- Qubic
- Tic-Tac-Toe

部分解决的
- Checkers(西洋跳棋)
- Chess(国际象棋)
- Go(已经到 4×4，游戏一般是 19×19)。
- Reversi

图 12-2　已经解决(全部或部分)的经典游戏

有的游戏具有非常庞大的状态空间(游戏 Go 具有一个大约 10400 个节点的游戏树，粗略地说，这个数字比宇宙中原子的数量还要大)，以至于几乎不能采用强力搜索方法。因此，要么需要在状态空间经过验证的部分内拥有一个非常智能的有指导的搜索程序，要么是需要一个在给定游戏规则情况下能够开发新颖解决方案的智能算法。无论哪种方式，都有一些经典定义的 AI 问题。程序清单 12-1 给出了国际象棋程序 Faile 的开放源代码中的 search()和 think()两个函数。该程序是由 Adrien M. Regimbald 编写的。本书所附光盘里有它的完整源程序，以及相应的网站链接，读者可以从中获取更多信息。Faile 是一款非常紧凑但又具有完全特征的 Alpha-Beta 搜索系统，它赋予了这个微小程序专家级的 AI 游戏能力。注意到该搜索功能使用了有限最优，因为它有时间限制，并将在剩下时间内它所了解的最优走步的基础上进行决策，甚至将根据时间对是否还需要继续搜索进行决策。更多细节将在本章后面讨论 Alpha-Beta 搜索时给出。

程序清单 12-1 Faile 程序中的 search()函数和 think()函数

(经 MIT 授权摘录)

```
long int search (int alpha, int beta, int depth, bool do_null)
{

   /* search the current node using alpha-beta with negamax search */

   move_s moves[MOVE_BUFF], h_move;
   int num_moves, i, j, ep_temp, extensions = 0, h_type;
   long int score = -INF, move_ordering[MOVE_BUFF],
            null_score = -INF, i_alpha,h_score;
   bool no_moves, legal_move;
   d_long temp_hash;

   /* before we do anything, see if we're out of time or we have input: */
   if (i_depth > mindepth && !(nodes & 4095))
   {
      if (rdifftime (rtime (), start_time) >= time_for_move)
      {
         /* see if our score has suddenly dropped, and if so,
               try to allocate some extra time: */
         if (allow_more_time && bad_root_score) {
         allow_more_time = FALSE;
            if (time_left > (5*time_for_move))
            {
               time_for_move *= 2;
            }
            else {
               time_exit = TRUE;
               return 0;
            }
         }
         else {
            time_exit = TRUE;
```

```
    return 0;
        }
    }
    #ifndef ANSI
    if (xb_mode && bioskey ()) {
        time_exit = TRUE;
        return 0;
    }
    #endif
}

/* check for a draw by repetition before continuing: */
if (is_draw ()) {
    return 0;
}

pv_length[ply] = ply;

/* see what info we can get from our hash table: */
h_score = chk_hash (alpha, beta, depth, &h_type, &h_move);
if (h_type != no_info) {
    switch (h_type){
        case  exact:
    return (h_score);
        case  u_bound:
    return (h_score);
        case l_bound:
    return (h_score);
        case  avoid_null:
    do_null = FALSE;
    break;
        default:
    break;
    }
}

temp_hash = cur_pos;
ep_temp = ep_square;
i_alpha = alpha;

/* perform check extensions if we haven't gone past maxdepth: */
if (in_check ())
{
    if (ply < maxdepth+1) extensions++;
}
/* if not in check, look into null moves: */
else
{
    /* conditions for null move:
        - not in check
```

```
        - we didn't just make a null move
        - we don't have a risk of zugzwang by being in the endgame
        - depth is >= R + 1
        what we do after null move:
        - if score is close to
             -mated, we're in danger, increase depth
        - if score is >= beta, we can get an early cutoff and exit */
    if (do_null && null_red && piece_count >= 5 &&
             depth >= null_red+1) {
        /* update the rep_history just so things don't get funky: */
        rep_history[game_ply++] = cur_pos;
        fifty++;

        xor (&cur_pos, color_h_values[0]);
        xor (&cur_pos, color_h_values[1]);
        xor (&cur_pos, ep_h_values[ep_square]);
        xor (&cur_pos, ep_h_values[0]);

        white_to_move ^= 1;
        ply++;
        ep_square = 0;
        null_score = -search (-beta, -beta+1,
                                  depth-null_red-1, FALSE);
        ep_square = ep_temp;
        ply--;
        white_to_move ^= 1;

        game_ply--;
        fifty--;

        xor (&cur_pos, color_h_values[0]);
        xor (&cur_pos, color_h_values[1]);
        xor (&cur_pos, ep_h_values[ep_square]);
        xor (&cur_pos, ep_h_values[0]);
        assert (cur_pos.x1 == compute_hash ().x1 &&
           cur_pos.x2 == compute_hash ().x2);
           /* check to see if we ran out of time: */
           if (time_exit)
        return 0;

        /* check to see if we can get a quick
              cutoff from our null move: */
        if (null_score >= beta)
    return beta;

        if (null_score < -INF+10*maxdepth)
    extensions++;
    }
}
```

```
/* try to find a stable position before passing
        the position to eval (): */
if (!(depth+extensions))
{
    captures = TRUE;
    score = qsearch (alpha, beta, maxdepth);
    captures = FALSE;
    return score;
}

num_moves = 0;
no_moves = TRUE;

/* generate and order moves: */
gen (&moves[0], &num_moves);
order_moves (&moves[0], &move_ordering[0], num_moves, &h_move);

/* loop through the moves at the current node: */
while (remove_one (&i, &move_ordering[0], num_moves)) {

    make (&moves[0], i);
    assert (cur_pos.x1 == compute_hash ().x1 &&
        cur_pos.x2 == compute_hash ().x2);
    ply++;
    legal_move = FALSE;

    /* go deeper if it's a legal move: */
    if (check_legal (&moves[0], i)) {
        nodes++;
        score = -search (-beta, -alpha, depth-1+extensions, TRUE);
        no_moves = FALSE;
        legal_move = TRUE;
    }

    ply--;
    unmake (&moves[0], i);
    ep_square = ep_temp;
    cur_pos = temp_hash;

    /* return if we've run out of time: */
    if (time_exit)  return 0;

    /* check our current score vs. alpha: */
    if (score > alpha && legal_move) {

        /* update the history heuristic since we have a cutoff: */
        history_h[moves[i].from][moves[i].target] += depth;

        /* try for an early cutoff: */
        if (score >= beta) {
```

```
            u_killers (moves[i], score);
            store_hash (i_alpha, depth, score, l_bound, moves[i]);
            return beta;
               }
               alpha = score;

               /* update the pv: */
               pv[ply][ply] = moves[i];
               for (j = ply+1; j < pv_length[ply+1]; j++)
            pv[ply][j] = pv[ply+1][j];
               pv_length[ply] = pv_length[ply+1];
            }

         }

         /* check for mate / stalemate: */
         if (no_moves) {
            if (in_check ()) {
               alpha = -INF+ply;
            }
            else {
               alpha = 0;
            }
         }
         else {
            /* check the 50 move rule if no mate situation
                     is on the board: */
            if (fifty > 100) {
               return 0;
            }
         }

         /* store our hash info: */
         if (alpha > i_alpha)
            store_hash (i_alpha, depth, alpha, exact, pv[ply][ply]);
         else
            store_hash (i_alpha, depth, alpha, u_bound, dummy);

         return alpha;

      }
      //--------------------
      move_s think (void) {

         /* Perform iterative deepening to go further in the search */

         move_s comp_move, temp_move;
         int ep_temp, i, j;
         long int elapsed;
```

```
/* see if we can get a book move: */
comp_move = book_move ();
if (is_valid_comp (comp_move)) {
    /* print out a pv line indicating a book move: */
    printf ("0 0 0 0 (Book move)\n");
    return (comp_move);
}

nodes = 0;
qnodes = 0;
allow_more_time = TRUE;

/* allocate our time for this move: */
time_for_move = allocate_time ();

/* clear the pv before a new search: */
for (i = 0; i < PV_BUFF; i++)
  for (j = 0; j < PV_BUFF; j++)
    pv[i][j] = dummy;

/* clear the history heuristic: */
memset (history_h, 0, sizeof (history_h));

/* clear the killer moves: */
for (i = 0; i < PV_BUFF; i++) {
  killer_scores[i] = -INF;
  killer_scores2[i] = -INF;
  killer1[i] = dummy;
  killer2[i] = dummy;
  killer3[i] = dummy;
}

for (i_depth = 1; i_depth <= maxdepth; i_depth++) {
  /* don't bother going deeper if we've
          already used 2/3 of our time, and we
     have finished our mindepth search, since
          we likely won't finish */
  elapsed = rdifftime (rtime (), start_time);
  if (elapsed > time_for_move*2.0/3.0 && i_depth > mindepth)
     break;

  ep_temp = ep_square;
  temp_move = search_root (-INF, INF, i_depth);
  ep_square = ep_temp;

  /* if we haven't aborted our search on time,
          set the computer's move
     and post our thinking: */
  if (!time_failure) {
     /* if our search score suddenly drops, and
```

```
                we ran out of time on the
            search, just use previous results */
        comp_move = temp_move;
        last_root_score = cur_score;
        /* if our PV is really short, try to get some
               of it from hash info
            (don't modify this if it is a mate / draw though): */
        if (pv_length[1] <= 2 && i_depth > 1 &&
                abs (cur_score) < (INF-100) &&
            result != stalemate && result != draw_by_fifty &&
            result != draw_by_rep)
            hash_to_pv (i_depth);
        if (post && i_depth >= mindepth)
            post_thinking (cur_score);
      }

      /* reset the killer scores (we can keep the
            moves for move ordering for now, but the
            scores may not be accurate at higher depths, so we need
         to reset them): */
      for (j = 0; j < PV_BUFF; j++) {
         killer_scores[j] = -INF;
         killer_scores2[j] = -INF;
      }

   }

   /* update our elapsed time_cushion: */
   if (moves_to_tc) {
     elapsed = rdifftime (rtime (), start_time);
     time_cushion += time_for_move-elapsed+inc;
   }

   return comp_move;

}
```

12.1 通用 AI 元素

12.1.1 对手 AI

根据定义，一个零和博弈必须有一个取胜或失败的对手。从娱乐角度来说，该对手必须是另一个人而且是在个性外表下按照规则进行游戏。对于大多数游戏，这种个性简单地由一个难度等级和每一个等级中与程序对抗足够多次来表示，人类将最终决定计算机控制的玩家将采取或不采取的走步。

12.1.2　AI 助手

国际象棋等消费者游戏通常包括一种 tutor(指导)模式，在该模式下计算机会带领玩家浏览许多的训练和教程以改进玩家的游戏。尽管有的游戏只以脚本教程方式提供了最小限度的指导内容，但其他游戏则实际上包含了一个智能帮助系统，它将发现玩家在游戏中的缺陷，然后指引他到脚本教程中去，或实时给出关于棋盘设置的建议。许多人购买国际象棋游戏产品都仅仅是因为这个特点，因为他们希望通过 AI 系统的指导和实践来学习或改进他们的游戏。其他像 Bridge 之类的游戏具有相当大或容易混淆的规则设置，它们也使用 AI 助手系统来传授游戏的基本策略。

12.2　有用的 AI 技术

12.2.1　有限状态机

大多数此类游戏都是相当线性的游戏(尽管有些只具有一个基本状态变化，即结束游戏)。游戏玩法可以分解成更小的部分(比如国际象棋游戏中可分为开始阶段、中间阶段和最后阶段)，它们很容易辨认，因此允许系统根据这些子状态在不同的 AI 方法之间切换。

12.2.2　Alpha-Beta 搜索

在需要搜索极小化极大树(minimax tree)的经典游戏中，Alpha-Beta 搜索几乎已经成为事实上的搜索标准。极小化极大树是一种专门设置的游戏状态树，树的各层都可由节点和与节点相联系的数值组成，其中节点表示玩家可以做出的选择，数值则代表相应节点与取胜节点的接近程度。图 12-3 是极小化极大树的一个简化示例。根据极小化极大树就可以得到算法，即在进行每次选择时，第一个对手选择使其数值分数最大的走步，而另一个玩家选择数值分数最小的走步。这是因为第一个玩家试图最大化他的分数，而第二个玩家试图最小化第一个玩家的分数。这种技术为此类游戏指出了一个最优的走步方向，但是也存在问题，即它假定第二个玩家是完全防御的。

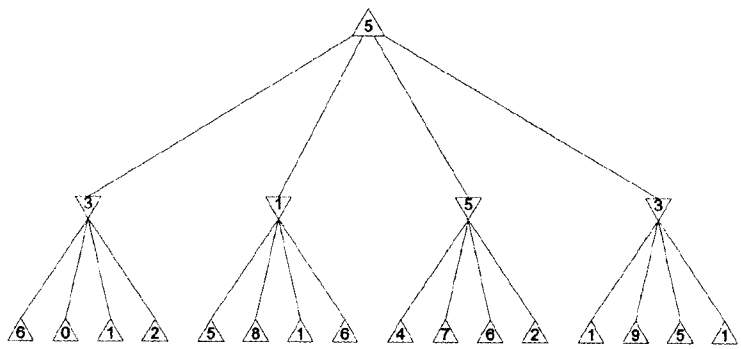


图 12-3　各玩家一个回合(或“来回”)的极小化极大树的一个简化示例

极小化极大方法可以扩展到那些包含一个纯偶然元素的游戏之中，如西洋双陆棋。该扩展称为 expectimax 树，它仅仅添加了在每个树节点都不能计算的纯最小值和最大值，从而引入了偶然节点(用于对游戏中引入的随机数值进行估计)。

完全极小化极大搜索存在的问题是它对整个树都进行了考虑。对于国际象棋，在任意给定的棋盘位置，如果那样做，则通常有 35 种合法走步。这意味着 1 层搜索有 35 个入口，2 层是 352 个，6 层就几乎是两百万个，10 层搜索(事实上每个玩家只能走 5 步)就超过 2*1015 个。进行尽可能深的搜索是非常重要的(普通人类玩家通常可以基于向前看 6 到 8 步来进行决策，而大师级玩家的决策有时可以向前看 10 到 20 步)，而且 Alpha-Beta 搜索允许我们安全地对整个树的分支进行裁剪，因此极大地减小了需要执行比较的数量，除非很不幸地将游戏树设置成最坏的情况(也即意味着完全不存在最优情况，而且这其实就是在执行常规的极小化极大搜索)。

12.2.3 神经网络

具有较大状态空间或相当奇怪的棋盘位置赋值的策略游戏(如 Go 游戏，其大多数的位置得分都涉及到一些非常深奥的东西，如“影响力”和“地域”等)，可以很好地将此类深奥知识存储在神经网络(Neural Network，NN)中。然而，此类数据结构非常难以训练，甚至也很难进行调试。之所以把它用于上述情形是因为没有其他方法可以真正解决这些问题。

12.2.4 遗传算法

遗传算法(Genetic Algorithm，GA)可以认为是另一种搜索类型，即“随机游动搜索(random walk search)”。这意味着使用某种形式的有指导的随机性来搜索状态空间，从而获得解决方案。在这种情况下，采用自然选择作为指导方针，并采用随机变异作为随机元素。我们将在本书第IV部分对该类算法进行详细讨论。

12.3 例外

大多数策略游戏的玩法都非常直接，没有预谋或偏袒。然而，有些人对他们的游戏做了技术修改，让它们具有了个性的外观，如 Digenetics™ 公司出品的 Checkers with an Attitude，该游戏采用多个神经网络来使这个西洋跳棋游戏变得非常优秀并更具个性。

12.4 示例

国际象棋计算机程序从计算机诞生开始就一直伴随着我们。最早的一个产生于在 20 世纪 50 年代早期，并到现在仍然是作为娱乐的最流行的经典游戏之一。早期的商业游戏，如 Sargon 等，并不具有多少智能，且运行速度也非常慢。现在，国际象棋游戏实际上已经

发展到了很高的水平，只需不到 30 美元，便可以到商店里去购买一个大师级的快速国际象棋程序来进行对抗。随着时间的流逝，有些公司试图将各种规则进行混合，但仍然维持同样的游戏，比如 1988 年的 Battle Chess 游戏，当玩家吃掉对手的棋子时它会显示一个动画的死亡序列。

12.5　需要改进的领域

12.5.1　创造力

遗传算法的扩展使用将使得该类型的 AI 对手逐步找到遗传算法所擅长的非直觉解决方案。另外，通过神经网络或遗传算法来确定启发函数，可以实现不同的基于启发函数的搜索，从而创造性地找到局部解决方案。

12.5.2　速度

速度一直都是游戏 AI 编程的一个最重要的因素，特别是在需要进行大量搜索的策略游戏之中更是如此。通过改进强力搜索方法，我们能够最终找到一个获取决策的智能方法，而不需要通过花费大量时间搜索大规模的状态树来得到最优的解决方案。或者，计算机速度变得足够快，使得最优搜索变得很容易，而且可以让 AI 运行于其他地方。

12.6　小结

经典策略游戏是最早使用理论 AI 技术来构建对手的游戏之一，因为它们是有指导的 AI 搜索方法的理想游戏。策略游戏已经向娱乐行业展示了对此类型问题使用真正的 AI 解决方案所带来的好处，并向我们提供了大多数的数据结构和方法论。

- 经典策略游戏被定义为具有完备信息的零和博弈，它们主要是全局走步，并且是基于回合制。
- 所编码的对手类型是以所设计的游戏类型为基础的：一个竞争对手需要最优的性能，但一个娱乐对手必须使用难度等级之类的东西。
- 娱乐策略游戏有时包括有 AI 助手，在游戏过程中用于教导并提供建议。
- FSM 仍可用于这些游戏，从而将状态空间分解成较小的块。
- Alpha-Beta 搜索是一种主要的对手建模方法，很多经典策略游戏使用它来在规划中考虑对手的走步。
- 遗传算法和神经网络能够以新的方式来推进有指导的搜索，或寻找非直觉的解决方案。
- 创造力是这些游戏中普遍缺乏的元素，这些游戏通常使用更多的强力搜索来寻找正确的响应。
- AI 系统的速度一直是此类游戏的关注点，因为它占据了游戏使用的 CPU 时间的很大一部分。

13 格斗类游戏

格斗类游戏(fighting game)过去是动作游戏和对手谜题游戏的一种奇怪混合体，特别是在街机游戏中。早期格斗游戏是具有健全人物和主要角色的简单横向卷轴(side-scrolling)游戏(有时被称为“搏击游戏(brawler)”或“剑魂(beat-em-up)”，如 Double Dragon、Bad Dudes 和 Final Fight 等)，它们更像是使用武术而不是射弹武器的水平滚动射击游戏。其他类型的早期格斗游戏包括拳击游戏(如 Nintendo 公司的 Punch Out 游戏)和摔跤竞赛(NES 游戏中的 Pro Wrestling)。所有这些游戏都非常流行，但格斗类游戏仍然只是另一种游戏类型。

然而，在 20 世纪 90 年代早期，Capcom 公司发布的一款《街霸 II》(Street Fighter 2：The World Warrior，SF2)游戏(见图 13-1 的屏幕截图)使格斗游戏类型达到了其流行的顶峰。《街霸 II》采用简单的搏击游戏规则，其全部体验就是格斗，通过连续技(combo)、阻碍技(block)、超必杀技(super move)以及在玩家看来是人与人对抗的动作(尽管更早的一个游戏 Karate Champ 在最开始做了部分事情，但《街霸 II》在所有方面都比它做得更好并取得了第一个真正的肉搏战格斗游戏的头衔)等而获得了巨大成功。商场将他们的其他机器腾空出来，全部换成《街霸 II》机器。各地的人们都来排队，在机器旁边建立起他们的住所，等着轮到他们上场。《街霸 II》完成的一件重要事情是向游戏世界重新提出了复杂游戏控制的概念。《街霸 II》的高级玩家所必需的特殊招式与游戏世界中以前出现过的都不一样，而且人们喜欢使用复杂的手部运动来战胜怪物组合，而这种运动是需要玩家几天或几星期的练习才能掌握的。

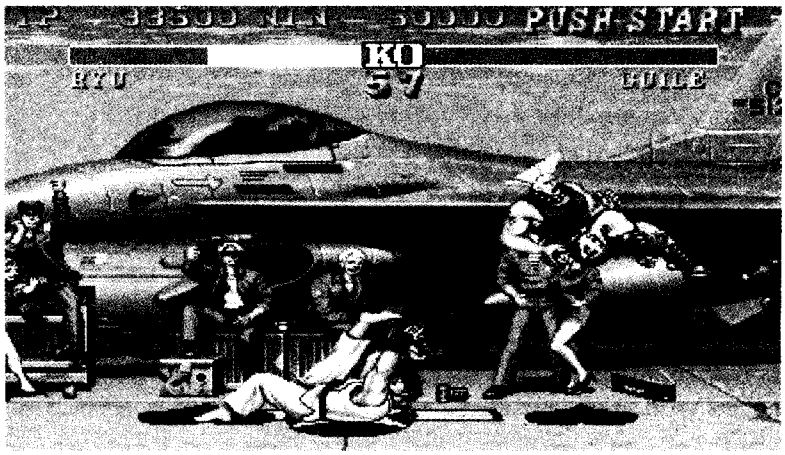


图 13-1 《街霸 II：世界武士》屏幕截图

(© Capcom Co.。未经允许，不得翻印)

该游戏被证明是非常流行的，人们通常认为它是 Super Nintendo 控制台最终在销量上追上 Sega Genesis 的一个主要原因，因为《街霸 II》的 Super Nintendo 版本是一个更好版本的游戏，其爱好者对它进行了疯狂抢购。

格斗类游戏与其他类型游戏一样，逐渐过渡到 3D，但并不是以同样的方式。诸如 Virtua Fighter 和 Tekken Tag Tournament(参见图 13-2 的屏幕截图)等游戏使用 3D 格斗方法为它们自身设计合适的环境；《街霸》系列继续停留在 2D 领域，但设计了更为深奥的玩法系统，而且由于摄像机问题和获胜所必需的快速玩法，该系统不能在 3D 环境中进行复制。这两种完全不同的格斗游戏世界依然存在。



图 13-2 Tekken Tag Tournament 屏幕截图

(TEKKEN TAG TOURNAMENT®，© 1994 1995 1996 1999 Namco Ltd.。版权所有。经 Namco Holding Corp 允许)

然而，摔跤游戏并不需要真正向 3D 过渡。大多数摔跤都包括缠斗(通过定义)，因此角色交互的类型更大，并且当角色使用不同的锁住和抱摔动作来回移动时，玩家可以通过发动招式和反击建立一个很深的招式链。角色纠缠在一起，因此不存在排列对手的问题，而且由于靠近摔跤选手，摄像机也可以更紧密地定位。

近年来，尽管格斗类游戏已经有所回落，但它们仍然到处存在并正侵蚀着其他类型的游戏，比如 Buffy the Vampire Slayer，它可被看作是半个格斗游戏和半个冒险游戏。这是所有游戏类型的趋势：开始非常成功，发展到成熟(和复杂性)时至多是增加一些额外的改进，暂时变得不流行，然后半绝望地与其他游戏类型融合以增加体验的内容和风味。

13.1 通用 AI 元素

13.1.1 敌人

格斗类游戏中的敌人使用一些迄今调整的最多和平衡的对手 AI 代码。大多数成功的格斗类游戏的最大卖点之一就是游戏是平衡的，即没有一个角色在本质上比其他角色更容

易取胜。有些角色可能很难或很容易控制，但经过练习，玩家可以使用它们之中的任一种来获得同样的致命效果。因此，对单个角色招式的精确控制，直至动画级别的单帧动画，都要加以练习。同样地，大多数此类游戏都使用能够一帧帧描述事件的某种脚本语言。这些事件包括：声音、打开或关闭防御或进攻冲突范围、在可能有分支的动画中标记得分(对于连续技)以及招式需要触发的其他东西。角色编剧将花费数月时间来研究游戏的平衡问题。

在一些格斗类游戏中，背景不仅仅是一个背景幕，还可能包含一些元素。它们可以用于战斗，或者隐藏在后面，或者需要被打碎从而获取某种宝物。这些游戏中的敌人也需要通过这些元素来智能地做出反应。例如，逼近主角色的敌人发现在他本身和他的对手之间有一个大木箱。他是使用一个避障系统来绕开它？还是将它捡起并把它扔到路外或仍向他的对手？他要跳过它吗？还是使用一记重拳将它击碎？如果要设计一个背景交互的格斗类游戏，那么这就是敌人需要完成的高级决策类型。

13.1.2　碰撞系统

角色级别的碰撞系统对格斗游戏类型来说也是极其重要的。每个角色典型地都具有许多的碰撞区域，而且对于每个给定的动画帧，它们的大小都可能改变，甚至在某些时段它们将失去碰撞能力。为了使游戏玩法更容易，碰撞从来不会真正地基于物理而是依靠一些调整好的数据，这些数据详细给出了每个玩家感觉到的回击数量、碰撞时播放的动画、产生的声音或效果、与招式相联系的恢复时间(即出了一招后完全不能出下一招之间的时间数量)以及游戏需要的许多其他数据值。

13.1.3　敌人头目

与 RPG 游戏和一些其他游戏类型一样，格斗类游戏在各关卡末端也使用敌人头目来将玩家引入一个更大和更具威胁的敌人。在 2D 搏击游戏中，它们有时是整个游戏中唯一难以忘却的敌人，这是与水平滚动射击游戏的另一个相似点。肉搏战格斗游戏传统上只具有一个头目，他是在玩家击败所有其他敌人后还需要对抗的家伙。这个角色传统上都很难被击败，其难度比游戏中其余敌人所设定的难度要高得多。

13.1.4　摄像机

与 3D 平台游戏一样，在 3D 格斗类游戏中，也会出现摄像机定位的问题。然而，由于快节奏特性和格斗游戏所使用的与摄像机相关的控制，3D 格斗游戏中的摄像机需要引起特殊注意；否则，它将会搞乱连续技，并由于方位问题造成招式丢失目标，而且通常使游戏非常混乱，从而毁灭整个格斗类游戏。

与平台游戏的另一个区别是玩家没有时间来使用一个自由视角的摄像机，因为他正参与一个近距离格斗。另外，由于潜在具有两个(或更多)人类玩家，从控制或可见度角度来说，自由视角的摄像机也不可行。因此，一个良好的算法摄像机或跟踪摄像机系统是必不可少的。

13.1.5 动作和冒险元素

格斗类游戏中一些类型交叉的变种使用了越来越多的动作游戏或冒险游戏成分。有的包含了大量的冒险游戏所做的探险和解谜。但有的也包含了平台游戏世界所具有的跳跃和攀登挑战。通过混合这些额外的游戏元素，开发者保持了格斗类游戏的活力，并发明了一些保持游戏新鲜感的新玩法体验组合。

13.2 有用的AI技术

13.2.1 有限状态机

格斗类游戏通常是基于状态的，其 AI 控制的对手发招，然后坐在那，或对碰撞做出响应。一个简单的有限状态机(FSM)能够协调大多数的格斗类游戏，并向开发者提供足够多的结构以增加复杂性但又不造成维护上的困难。通常 FSM 的结构存储在某种数据库中，从而便于应对在调整和测试时任意给定角色的状态图将急剧变化这个事实。

13.2.2 数据驱动系统

由于具有大量的角色、招式、阻挡技、投技(throw)和连续技，尤其是在给定调整级别和游戏所应具有的平衡的时候，用设计者可访问的脚本将基本格斗引擎进行划分是唯一可行的方式。通常，每个招式都被脚本化从而允许对攻击、防御、连续技分支、音效、碰撞时间和碰撞范围大小以及造成的损害等进行精确判断。碰撞系统通常非常简单(甚至最早的《街霸 II》游戏对每个敌人鬼怪都有很多的碰撞区域，具有分离的头部、手臂、身体和腿部等)，通过一些数据表格来详细描述当敌人区域被击中或被阻挡时应该播放的动画。另外的表格将通过列出招式和连续技的偏好值、玩家能够防御侵略的程度以及其他所有关于格斗者的事情，来描述每个格斗者的“个性”。

13.2.3 脚本系统

除了设计者需要严格控制格斗动画之外(因此，它们通常需要一个脚本来详细描述在每个招式中需要发生的所有事情)，格斗类游戏中的故事元素等也仍然非常普遍。这尤其发生在一些冒险式格斗游戏变种之中。可见，脚本系统在格斗类游戏中也经常使用。

脚本系统在游戏内的影片式场合中也非常有用。例如，当格斗开始时角色进入竞技场，或某玩家获胜后跳起了某种胜利舞蹈。非常复杂的招式(有时称为“超连续技(super combo)”等)可能也需要一定程度的脚本，因为超连续技通常是从其他招式中构建出来的，并以某一特定方式将它们联系起来。

13.3　示例

早期格斗类游戏都是些非常简单的事情。通常拥有一个拳击按钮，或者是一个拳打脚踢按钮。该领域内的游戏是卷轴游戏(或搏击游戏)，比如 Bad Dudes、Kung-Fu Master、Golden Axe 和 Ninja-Gaiden 等。敌人具有非常简单的 AI，通常只是试图包围玩家并使出它们武器库中具有的一些相当简单的招式或招式组合。卷轴格斗游戏有头目角色，但头目通常只是速度更快，或具有很多的生命值(hit point)，或拥有巨大的武器，它们基本上不会更智能。

之后，肉搏战格斗游戏开始出现，并变得非常流行，以至于出现了许多不同的游戏专利，如 Samurai Showdown、King of Fighters、Mortal Kombat 和 Street Fighter 系列。随着时间的发展，这些游戏的续篇将继续变得更好、更复杂且更具技术含量。

肉搏战格斗游戏中 AI 控制的敌人通常是完全有血有肉的好战对手，具有几乎所有人类的全部能力。AI 难度级别可通过操作器(在街机中)或玩家(在家庭控制台上)来设置，以符合任意用户的技能级别——从完全无能到几乎无可匹敌。这是可能的，因为在构建这些游戏的过程中，通过精细调整的输入窗口、动画帧计数和严格调整过的碰撞系统，游戏开发者允许整个系统根据粗略的难度级别和时间缩放比例(对于不同的超高速播放模式)来按比例增加或减小。脚本以及与各招式相联系的数据都能够内在处理变化的技能级别。

3D 搏击游戏也已经出现有很长一段时间，其最初的游戏是 Battle Area Toshinden(该游戏与许多人的第一个 Playstation 控制台一块出现)，所有游戏一起导致了如今的品牌游戏，如 Soul Calibur、Dead or Alive 和 Virtua Fighter。这些游戏都使用它们 2D 版本游戏的数据驱动 AI 系统，但也使用了扩展的摄像技巧，并且由于有些游戏使用了先进的地形，故它们甚至使用一定程度的路径搜索。

像 Buffy the Vampire Slayer(使用一种流行的授权和大量的探险挑战)、The Mark of Kri(与电影艺术的伟大融合)和 Viewtiful Joe(一个倒退的游戏，它采用现在的先进技术并与老式搏击游戏的核心部分相结合)等游戏都是在不同的其他游戏类型中使用大量格斗系统的例子。所有这些游戏都使用了一些在常规格斗游戏中使用的技术，以及大量在主流动作和冒险游戏中出现的 AI 挑战。

13.4　需要改进的领域

学习

格斗游戏与大多数的视频游戏相似，最终，人类将会找到一个弱点并反复对它进行利用，从而使得游戏对他自己来说变得太简单。甚至在《街霸 II》中就很明显，不断跳跃并反复进行猛烈的拳击就几乎总能击败非常困难的角色 Zangief。如果可怜的 Zangief 哪怕具有一点点的学习 AI，那他就能最终发现人类的进攻模式，并进行防范。学习系统也有助于开发一般情形，并且实际上如果人类重复一个单一的、非常强大的攻击并用计谋挫败对手的话，就可以通过 AI 提示来帮助保持游戏玩法的公平(至少是对抗计算机)。

如果某个攻击始终是成功击中，那么甚至可以用低难度级别的 AI 来帮助玩家调整其攻击模式。这样，尽管人类依然犯同样的错误，但格斗将变得更有趣味。

13.5 小结

格斗类游戏，不管是 2D 还是 3D 的，都给玩家提供了大多数其他游戏不能提供的一定程度的角色控制。它们对游戏高手和战术家(他们研究不同的阻挡和进攻系统从而找出各自的优势)都有很大的吸引力。

- 格斗类游戏开始于 2D 卷轴搏击游戏，且具有简单的控制和很少的策略。
- 肉搏战格斗游戏将该游戏类型与它生存所需要的玩法深度相结合，并在几乎一个时代里成为最流行的游戏类型。
- 格斗类游戏的敌人和敌人头目需要进行大量的调整，以保持游戏的平衡，因此在对它们编码时需要对此进行考虑。
- 格斗类游戏中使用的碰撞系统也非常复杂，需要具有比大多数游戏更高解析度的目标。
- 摄像机系统(对于 3D 格斗游戏)以及任何额外动作或冒险元素可能也需要 AI 代码。
- FSM 和脚本(或一些其他形式的数据驱动 AI)是设计格斗类游戏 AI 最普遍的方法。数据驱动格斗游戏非常重要，这是因为格斗游戏玩法的许多层级都需要具有大量的调整和设计者输入。
- 格斗类游戏中的学习能够帮助克服 AI 的开发使用，同时防止玩法变得重复。

14 著名的混杂游戏类型

尽管大多数游戏都属于前面各章中的类型，但还是有许多游戏要么很难进行归类，要么是独成一派。本章将强调这些游戏中最为著名的几个，并讨论在它们的设计中所用到的AI方法。

14.1 文明游戏

文明游戏是基于回合制的策略游戏。它是一个大的回合制策略游戏，有时在每个给定的回合里，都需要控制相当大数量的单元，并需要玩家进行许多的管理和调整工作。它几乎毫无例外地属于PC游戏类型(主要是出于界面的考虑)，仅有少数几个该类型的控制台游戏，较好的例子是Final Fantasy Tactics游戏和Advance Wars掌上游戏。该类游戏几乎都被一个人主持开发，即Sid Meier。他在1989年设计了热门游戏SimCity™(将在后面的“天神游戏”中讨论)的一个副产品，并在两年后成功地设计了一种全新的游戏类型，即《文明(Civilization)》。从那时起，出现了一个完整的系列，以及许多其他文明游戏。《文明》系列(图14-1和图14-2给出了从《文明》到《文明III》的发展)和最近的Alpha Centauri以及其他一些游戏都是重要的文明游戏，它们具有难以置信的深层策略、非常具有挑战性的AI系统、优秀的界面，并且几乎具有无限的重玩价值(replay value)。文明游戏的一些其他重要例子有：X-Com、Heroes of Might and Magic游戏和Master of Orion系列。

界面是回合制的，意味着玩家(人类和AI对手的混合体)轮流对他们的军队、城市等发布命令，然后观察该回合呈现出来的全部行动。该过程持续下去，来回进行，直到游戏结束。玩家能够控制所有的事情：发动哪场战争，建造哪些城市和市镇，进行哪种类型的研究，对哪些发明分配资源等。这些游戏能够持续很长时间，几个小时甚至是几天。但是，由于是一种回合制的机制，双方都有更多的时间来进行决策，因此也就出现了深层的玩法策略。在第1章“基本定义与概念”中讨论的有限最优概念在这里开始产生作用；更实时的AI系统面临的时间限制几乎都加在了这些游戏中AI控制的对手身上。人类并不愿意等待计算机来采取招式和决策，因此在人类执行他的回合时，大多数文明游戏的AI引擎都要进行大量的计算，这样，可以限制计算机对手回合所耗费的时间。

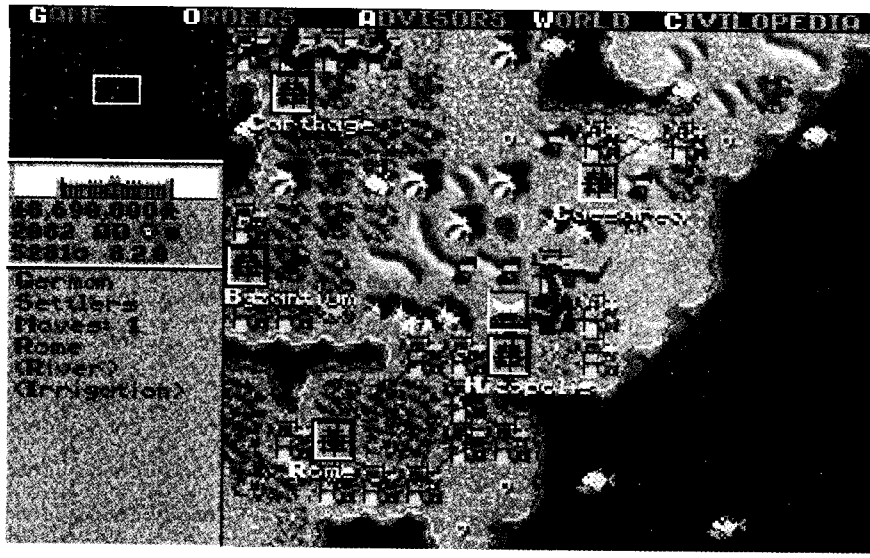


图 14-1 《文明》屏幕截图

(Sid Meier's Civilization® 和 Sid Meier's Civilization® III。经 Atari Interactive, Inc 允许。
© 2004，Atari Interactive, Inc。版权所有。未经允许，不得使用)

与 RTS 游戏不同，这里很少有基于单元的智能。几乎所有的决策都是策略性的，单个战斗单元之间(甚至是单元与被防护的城市之间)的冲突归结为基于单元兵力和防御数量的随机数。这导致了更多的模拟感，而不是单个战斗添加到 RTS 游戏类型上之后的动作元素。

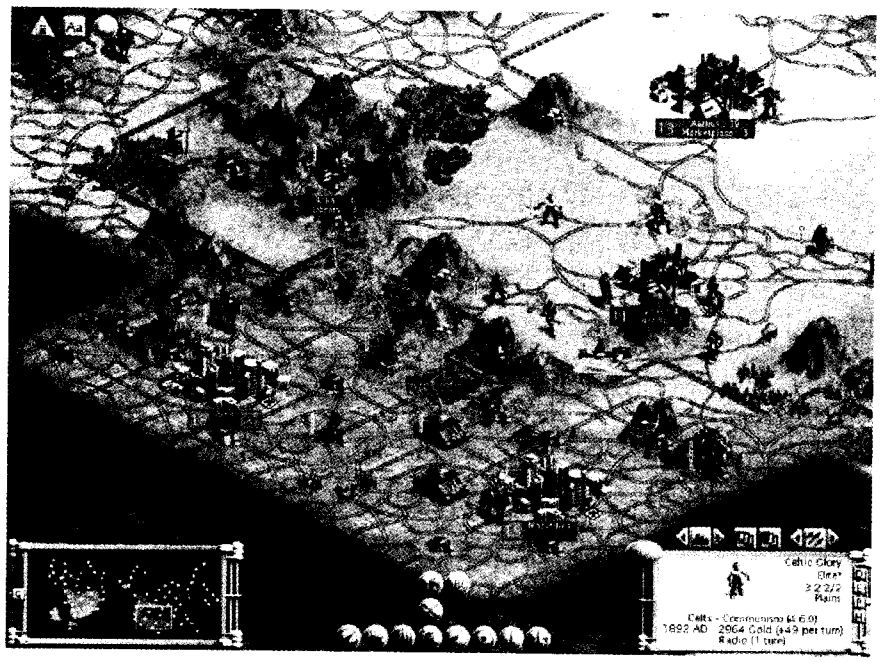


图 14-2 《文明 III》屏幕截图

(Sid Meier's Civilization® 和 Sid Meier's Civilization® III，经 Atari Interactive, Inc 允许。
© 2004，Atari Interactive, Inc。版权所有。未经允许，不得使用)

文明游戏中使用的典型 AI 系统如下：

- 与 RTS 游戏需要的 AI 方法同样类型的大多数方法，包括有限状态机(FSM)、模糊状态机(FuSM)、层次化 AI 系统、良好的路径搜索和消息系统。
- 文明游戏也借用了 RTS 游戏使用的大多数支持系统，包括地形分析、资源管理、城市规划技术和对手建模。
- 根据文明类型(以及许多类型的单元、技术、资源等)的数量，大量数据驱动的元素通常在这些游戏中也有所描述。
- 因为这些游戏通常具有扩展的技术树和巨大的游戏世界，因此需要鲁棒规划算法。程序清单 14-1 是 FreeCiv 游戏中 AI 代码的一个非常小的示例。FreeCiv 是《文明》游戏的一个开放源码娱乐游戏，具有大量的跟随者并已经被移植到许多平台游戏中。
- 先进的顾问与外交 AI 系统。许多此类游戏都需要完成大量的工作，而一些人会觉得做完所有的事情太无聊和厌烦，因此引入了顾问这个概念。这些 AI 角色可以通过被问起时提供建议来帮助玩家完成一些玩家觉得单调或混淆的部分游戏任务。实际上，在人类进行控制时该系统使用 AI 决策引擎来忽略游戏世界，然后告诉人类计算机将马上采取什么行动，来作为一个可以接受或抛弃的建议。在典型情况下，这些顾问是专门针对游戏的不同部分的，如贸易、研究或管理。这样，玩家只需要向这些顾问咨询他所希望的事情而忽略其他。外交系统也非常复杂。不同的群体可以结盟，领袖可能欺诈、撒弥天大谎或耿耿于怀。这些外交类型的心理状态在一个游戏中变化非常快，要满足所有人是不可能的，就跟在真实生活中一样。事实上，在最初的《文明》游戏中，是几乎不可能运行一个完全不流血的游戏的，文明游戏不可能生活在一个安详和繁荣的世界中，直到某人通过技术上的优势取胜。Sid 是否了解关于人类本质的一些东西呢？

程序清单 14-1　FreeCiv 中的 AI 代码示例

```
/**************************************************************************
   Buy and upgrade stuff!
**************************************************************************/
static void ai_spend_gold(struct player *pplayer)
{
   struct ai_choice bestchoice;
   int cached_limit = ai_gold_reserve(pplayer);

   /* Disband troops that are at home but don't serve a purpose. */
   city_list_iterate(pplayer->cities, pcity) {
      struct tile *ptile = map_get_tile(pcity->x, pcity->y);
      unit_list_iterate(ptile->units, punit) {
        if (((unit_types[punit->type].shield_cost > 0
              && pcity->shield_prod == 0)
             || unit_has_role(punit->type, L_EXPLORER))
            && pcity->id == punit->homecity
            && pcity->ai.urgency == 0
            && is_ground_unit(punit)) {
```

```
          struct packet_unit_request packet;
          packet.unit_id = punit->id;
          CITY_LOG(LOG_BUY, pcity,
                       "disbanding %s to increase production",
                   unit_name(punit->type));
          handle_unit_disband(pplayer, &packet);
        }
      } unit_list_iterate_end;
    } city_list_iterate_end;

    do {
        int limit = cached_limit; /* cached_limit is our gold reserve */
        struct city *pcity = NULL;
        bool expensive; /* don't buy when it costs x2 unless we must */
        int buycost;

        /* Find highest wanted item on the buy list */
        init_choice(&bestchoice);
        city_list_iterate(pplayer->cities, acity) {
        if (acity->anarchy != 0) continue;
        if (acity->ai.choice.want > bestchoice.want &&
                                        ai_fuzzy(pplayer, TRUE))
            {
          bestchoice.choice = acity->ai.choice.choice;
          bestchoice.want = acity->ai.choice.want;
          bestchoice.type = acity->ai.choice.type;
          pcity = acity;
        }
      } city_list_iterate_end;

      /* We found nothing, so we're done */
      if (bestchoice.want == 0)  break;

      /* Not dealing with this city a second time */
      pcity->ai.choice.want = 0;

      ASSERT_REAL_CHOICE_TYPE(bestchoice.type);

      /* Try upgrade units at danger location
            *  (high want is usually danger) */
      if (pcity->ai.danger > 1) {
        if (bestchoice.type == CT_BUILDING &&
              is_wonder(bestchoice.choice)) {
          CITY_LOG(LOG_BUY, pcity,
                     "Wonder being built in dangerous position! ");
        }else {
          /* If we have urgent want, spend more */
          int upgrade_limit = limit;
          if (pcity->ai.urgency > 1) {
              upgrade_limit = pplayer->ai.est_upkeep;
```

```
    }
    /* Upgrade only military units now */
    ai_upgrade_units(pcity, upgrade_limit, TRUE);
  }
}

/* Cost to complete production */
buycost = city_buy_cost(pcity);

if (buycost <= 0) {
    continue; /* Already completed */
}

if (bestchoice.type != CT_BUILDING
     && unit_type_flag(bestchoice.choice, F_CITIES)) {
    if (!city_got_effect(pcity, B_GRANARY)
        && pcity->size == 1
        && city_granary_size(pcity->size)
            > pcity->food_stock + pcity->food_surplus) {
            /* Don't build settlers in size 1
            * cities unless we grow next turn */
      continue;
    }
    else
    {
      if (city_list_size(&pplayer->cities) <= 8) {
          /* Make AI get gold for settlers early game */
          pplayer->ai.maxbuycost =
                                MAX(pplayer->ai.maxbuycost, buycost);
      } else if (city_list_size(&pplayer->cities) > 25) {
          /* Don't waste precious money buying settlers late game */
          continue;
      }
    }
}else {
    / * We are not a settler. Therefore we
            * increase the cash need we
      * balance our buy desire with to
            * keep cash at hand for emergencies
      * and for upgrades */
    limit *= 2;
}

/* It costs x2 to buy something with no shields contributed */
expensive = (pcity->shield_stock == 0)
            || (pplayer->economic.gold - buycost < limit);

if (bestchoice.type == CT_ATTACKER
    && buycost > unit_types[bestchoice.choice].build_cost * 2) {
    /* Too expensive for an offensive unit */
```

```
      continue;
    }

    if (!expensive && bestchoice.type != CT_BUILDING
        && (unit_type_flag(bestchoice.choice, F_TRADE_ROUTE)
        || unit_type_flag(bestchoice.choice, F_HELP_WONDER))
        && buycost < unit_types[bestchoice.choice].build_cost * 2) {
      /* We need more money for buying caravans. Increasing
         maxbuycost will increase taxes */
      pplayer->ai.maxbuycost = MAX(pplayer->ai.maxbuycost, buycost);
    }

    /* FIXME: Here Syela wanted some code to check if
     * pcity was doomed, and we should therefore attempt
     * to sell everything in it of non-military value */

    if (pplayer->economic.gold - pplayer->ai.est_upkeep >= buycost
        && (!expensive
        || (pcity->ai.grave_danger != 0
        && assess_defense(pcity) == 0)
        || (bestchoice.want > 200 && pcity->ai.urgency > 1))) {
      /* Buy stuff */
      CITY_LOG(LOG_BUY, pcity, "Crash buy of %s for %d (want %d)",
               bestchoice.type != CT_BUILDING ?
                     unit_name(bestchoice.choice)
               : get_improvement_name(bestchoice.choice), buycost,
               bestchoice.want);
      really_handle_city_buy(pplayer, pcity);
    } else if (pcity->ai.grave_danger != 0
               && bestchoice.type == CT_DEFENDER
               && assess_defense(pcity) == 0) {
      /* We have no gold but MUST have a defender */
      CITY_LOG(LOG_BUY, pcity,
               "must have %s but can't afford it (%d < %d)!",
          unit_name(bestchoice.choice),
               pplayer->economic.gold, buycost);
      try_to_sell_stuff(pplayer, pcity);
      if (pplayer->economic.gold - pplayer->ai.est_upkeep >=
            buycost) {
        CITY_LOG(LOG_BUY, pcity,
                      "now we can afford it (sold something)");
        really_handle_city_buy(pplayer, pcity);
      }
      if (buycost > pplayer->ai.maxbuycost) {
        /* Consequently we need to raise more money through taxes */
        pplayer->ai.maxbuycost =
                                MAX(pplayer->ai.maxbuycost, buycost);
      }
    }
  } while (TRUE);
```

```
  /* Civilian upgrades now */
  city_list_iterate(pplayer->cities, pcity) {
    ai_upgrade_units(pcity, cached_limit, FALSE);
  } city_list_iterate_end;

  if (pplayer->economic.gold + cached_limit <
          pplayer->ai.maxbuycost) {
    /* We have too much gold! Don't raise taxes */
    pplayer->ai.maxbuycost = 0;
  }

  freelog(LOG_BUY, "%s wants to keep %d in reserve (tax factor %d)",
          pplayer->name, cached_limit, pplayer->ai.maxbuycost);
}
#undef LOG_BUY

/**************************************************************************
  cities, build order and worker allocation stuff here..
**************************************************************************/
void ai_manage_cities(struct player *pplayer)
{
  int i;
  pplayer->ai.maxbuycost = 0;

  city_list_iterate(pplayer->cities, pcity)
    ai_manage_city(pplayer, pcity);
  city_list_iterate_end;

  ai_manage_buildings(pplayer);

  city_list_iterate(pplayer->cities, pcity)
    military_advisor_choose_build(pplayer, pcity,
                                  &pcity->ai.choice);
    /* note that m_a_c_b mungs the seamap, but we don't care */
    establish_city_distances(pplayer, pcity);
    /* in advmilitary for warmap */
    /* e_c_d doesn't even look at the seamap */
    /* determines downtown and distance_
    * to_wondercity, which a_c_c_b will need */
    contemplate_terrain_improvements(pcity);
    contemplate_new_city(pcity);
    /* while we have the warmap handy */
    /* seacost may have been munged if we found
    * a boat, but if we found a boat we don't rely on the seamap
    * being current since we will recalculate. - Syela */

  city_list_iterate_end;

  city_list_iterate(pplayer->cities, pcity)
```

```
        ai_city_choose_build(pplayer, pcity);
      city_list_iterate_end;

      ai_spend_gold(pplayer);

      /* use ai_gov_tech_hints: */
      for(i=0; i<MAX_NUM_TECH_LIST; i++) {
        struct ai_gov_tech_hint *hint = &ai_gov_tech_hints[i];

        if (hint->tech == A_LAST)
          break;
        if (get_invention(pplayer, hint->tech) != TECH_KNOWN) {
          pplayer->ai.tech_want[hint->tech] +=
          city_list_size(&pplayer->cities) * (hint->turns_factor *
                                num_unknown_techs_for_goal
                                (pplayer,
                                hint->tech) +
                                hint->const_factor);
          if (hint->get_first)
        break;
        } else {
          if (hint->done)
        break;
    }
  }
}
```

2003 年 10 月 28 日，Activision®发布了 Call to Power II 游戏的源代码，该游戏是主要《文明》游戏的一个分支。由于它具有的可扩展性程度，它受到了众多的爱好者的欢迎。它包含了非常强大的脚本系统(事实上，在它的源码发布之前，该游戏代码的许多实际漏洞就已经让聪明的游戏玩家用于设计基于脚本的工作区，并将它们在 Internet 上进行散布)。

14.2 天神游戏

另一种独特的且事实上被少数游戏运营商占据的游戏类型是天神游戏(god game)。之所以把它们称为“天神游戏”是因为玩家充当造物主、管理员以及改变整个游戏力量的角色，但又不直接控制游戏中的其他居民。在某些方面，这看起来更像是人工生命(alife)游戏的体验，但它具有更大的规模。人工生命游戏通常是只塑造一个创造物(或少数几个)，而且是通过直接训练和照顾它们来实现的。该游戏类型的两个创始人是 Will Wright 和 Peter Molyneux，他们设计和创造了最早的天神游戏。Wright 的游戏发布于 1987 年，叫做 SimCity™(图 14-4 和图 14-5 分别是 SimCity 和 SimCity 2000™的屏幕截图)。SimCity 是一个实时游戏，其玩家建造一个不断发展的城市并设法让 AI 控制的城市居民过得快乐和健康。1989 年，Molyneux 发布了 Populous™游戏(图 14-3 是其屏幕截图)，它通过把玩家塑造成超越陆地的上帝进一步深化了天神游戏的概念。

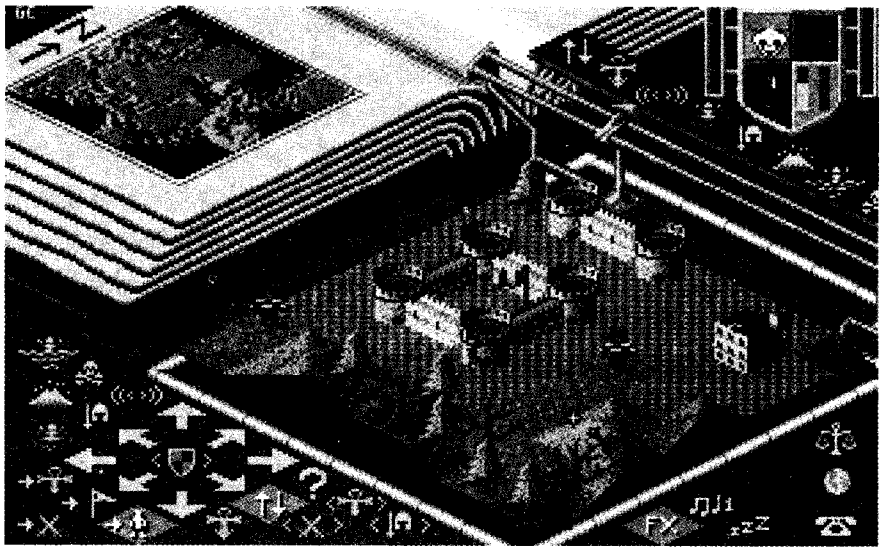


图 14-3　Populous 屏幕截图

(© 2004，Electronic Arts Inc。Populous、SimCity、SimCity 2000、SimAnt、SimEarth、SimFarm、Dungeon Keeper、The Sims 和 Ultima 是 Electronic Arts Inc 在美国和/或其他国家的商标或注册商标。版权所有)

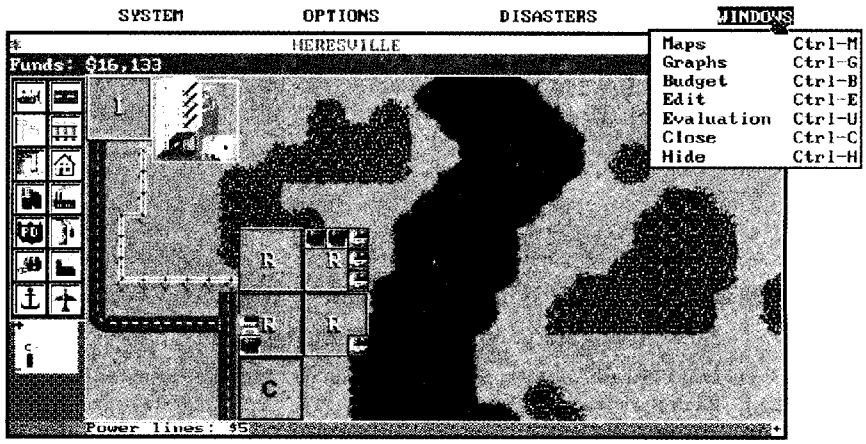


图 14-4　SimCity 屏幕截图

(© 2004，Electronic Arts Inc。Populous、SimCity、SimCity 2000、SimAnt、SimEarth、SimFarm、Dungeon Keeper、The Sims 和 Ultima 是 Electronic Arts Inc 在美国和/或其他国家的商标或注册商标。版权所有)

玩家可以创造或毁灭陆地上的元素，可以运用上帝级功力创造瘟疫或火山，并设法让陆地上的居民对他更为崇拜，从而赋予他更多的功力。随着时间流逝，Wright 和 Molyneux 都推出了其他的该类型的游戏，包括 Wright 阵营的 SimCity 变种(SimAnt™、SimEarth™和 SimFarm™等)以及 Molyneux 阵营的 Dungeon Keeper™和 Populous 2。他们两人现在都在从事持续向人工生命游戏发展的项目，这将在后面进行讨论。

此类游戏需要为上帝对手(如果有的话)设计大量的策略 AI。但在许多此类游戏中，尤其是在 SimCity 变种中，根本不存在策略 AI 系统。人类为他所在的一方提供所有的策略决策，而“对手”仅仅是毁灭力量，逐渐增加元素到需要玩家管理的模拟中，或经常性地通

过随机事故、耐力问题、逐渐增加的居民、资源以及系统需求等，来设法拆毁建筑物、城市等玩家试图建造的东西。

然而，所有这些游戏都具有通用的一种类型的 AI 系统，即所要神化的相当自主的角色，他们可以是人、蚂蚁或其他任何东西。他们将在指定的规则下居住和生活。通常，这些单个角色是随需要而带入游戏的，如每个生命需要 X 数量的食物、Y 数量的空间和 Z 数量的幸福度(或任意其他游戏中的等价物)。他们将四处徘徊，寻找能够满足这些需要的方式，并且如果已经正确建造了城市、世界或蚂蚁农田，他们将会找到。如果没有建造，他们会生气或离开，造成模拟的挫败。

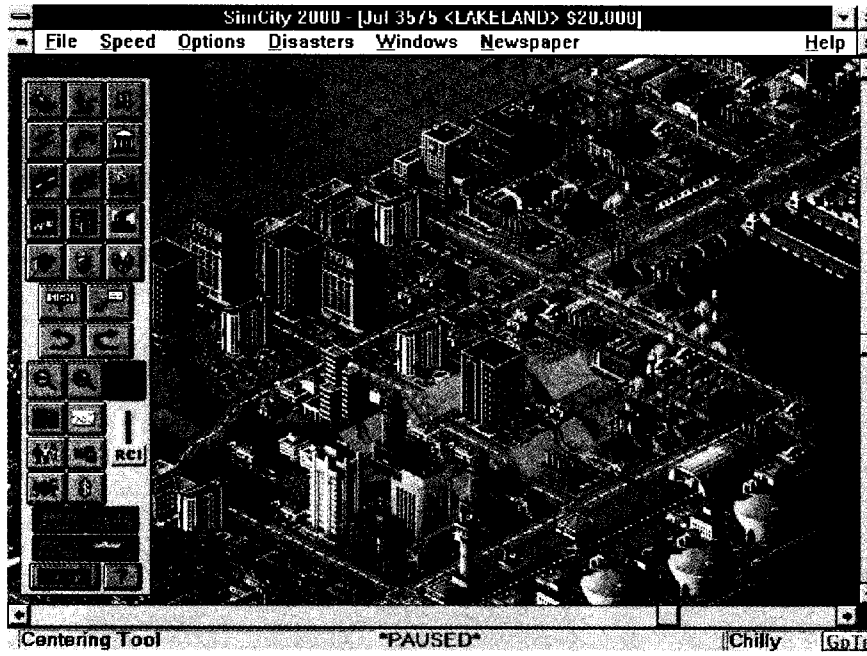


图 14-5 SimCity 2000 屏幕截图

(© 2004，Electronic Arts Inc。Populous、SimCity、SimCity 2000、SimAnt、SimEarth、SimFarm、Dungeon Keeper、The Sims 和 Ultima 是 Electronic Arts Inc 在美国和/或其他国家的商标或注册商标。版权所有)

在天神游戏中使用的典型 AI 系统有：

与文明游戏一样，该游戏类型也使用与 RTS 游戏相同的策略 AI 系统，但只有在存在一个与玩家竞争追随者或对世界的控制权的上帝对手时，才需要该类型的决策能力。

自主角色很可能使用一个基于状态的需求系统。在顶层，每个基本需求都与一个状态相联系，如 GetFood 或 GetAHouse，它们的激活需要饥饿或无家可归这样的感知。在每个状态下角色采取的行动将会给他们带来所需要的资源，并终结他们需要该资源的感知，从而改变他们的状态。一个非常平衡的该类型游戏几乎从来不会拥有一个不需要任何东西的自主角色，因此他会一直处于获取某些东西的状态，并一直忙碌下去。

“世界”的 AI 级别决定了玩家的市镇是否具有足够吸引力，从而有更多人将会出现在这里；或者触发一些随机事件来加大对玩家的挑战。这包括了所谓游戏的“规则”，它在大多数游戏中包括了物理定律和当玩家死亡时对魔法和再生的规定。然而，在天神游戏中，规则就是玩家要竞争的实际对手。因此，玩家必须要谨记这样的规则：“城市中的每个人都

必须要有 50 平方英尺的生活空间”和“每隔 300 个崇拜者就必须要建造另一个庙宇”，以防止失去对游戏的控制。

14.3　战争游戏

战争游戏与近来大量出现的战争主题 FTPS 游戏无关(如 Battlefield：1942 或 WW2Online)，它属于经典的回合制策略战争游戏，不通过对系统的间接控制来补充军队。这些游戏试图再现历史上的战斗，因此纸上谈兵的将军可以了解他们是否具有和那些职业将军一样的本能，或者比他们做得更好。这些游戏一直是瞄准机会的市场，甚至它们的原始形式，即那些非常复杂的棋盘游戏也是如此。Avalon Hill 是一个创造了大多数更知名的棋盘游戏的公司，大多数成功的计算机战争游戏都带有它的一些成分，或完全就是经典 Avalon Hill 游戏的翻版。

这些游戏需要比常规策略游戏具有更多的真实模拟，因为其整个卖点就是历史再创作，如果事情不是像真实生活中那样发生，游戏就不会被开始瞄准的哪怕是微小的机会市场所接受。因此，诸如地形穿越、视线计算、真实的天气模拟以及几乎对战斗各角度的统计建模等对于战争模拟的成功都具有极为重要的意义。

优秀战争游戏的一些例子包括：Combat Mission 游戏和 Airborne Assault 系列。程序清单 14-2 给出了 Wargamer：Napoleon 1813 游戏开放源码中的一个函数 buildObjective()。该游戏最初由 Empire® Interactive 在 1999 年发布，是对一些拿破仑最著名的战役的深层模拟，并成为了开放源码社区的一部分。该示例函数是 AI 使用的较高层级系统的一部分，用以确定将来的策略性计划。

程序清单 14-2　Wargamer: Napoleon 1813 中的 buildObjective()函数

(经 GNU 授权摘录)

```
bool AIC_ObjectiveCreator::buildObjective(const AIC_TownInfo& tInfo)
{
#ifdef DEBUG
   d_sData->logWin("Assigning units to %s", d_sData->campData()->
                                        getTownName(tInfo.town()));
#endif

   /*
    * Pass 1:
    *   build list of units and keep track of SPs removed from
    *   other objectives
    *
    *   Ony units that would not destroy an objective with
    *   a higher townImportance can be used
    */

   std::map<ITown, int, std::less<int> > otherObjectives;
   std::vector<TownInfluence::Unit> allocatedUnits;
```

```
    SPCount spNeeded = d_townInfluence.spNeeded();
    SPCount spAlloced = 0;
    SPCount spToAllocate = d_sData->rand(spNeeded,
                           d_townInfluence.spAvailable());

    TownInfluence::Unit infUnit;
    while((spAlloced < spToAllocate) &&
      d_townInfluence.pickAndRemove(&infUnit))
    {
      ASSERT(infUnit.cp() != NoCommandPosition);

      if(infUnit.cp()->isDead())
          continue;

      AIC_UnitRef aiUnit = d_units->getOrCreate(infUnit.cp());
      TownInfluence::Influence unitInfluence = infUnit.influence();
      // friendlyInfluence.influence(aiUnit.cp());
      float oldPriority = d_townInfluence.effectivePriority(aiUnit);
      if(unitInfluence >= oldPriority)
      {
          SPCount spCount = aiUnit.spCount();

#ifdef DEBUG
          d_sData->logWin("Picked %s [SP=%d, pri=%f / %f] ",
              (const char*) infUnit.cp()->getName(),
              (int) spCount,
              (float) unitInfluence,
              (float) oldPriority);
#endif

          /*
            * If it already has an objective
            * Then update the otherObjective list
            */

          AIC_Objective* oldObjective = aiUnit.objective();
          if(oldObjective)
          {
              ITown objTown = oldObjective->town();

              if (spAlloced > spNeeded)
              {

#ifdef DEBUG
                  d_sData->logWin("Not using %s from %s because we already
                  have enough SPs",
                      (const char*) infUnit.cp()->getName(),
                      (const char*) d_sData->campData()->
                                    getTownName(objTown));
#endif
```

```
                continue;
            }


            if (objTown != tInfo.town())
            {
                const AIC_TownInfo& objTownInf =
                            d_towns->find(objTown);
                if(objTownInf.importance() >= tInfo.importance())
                {
                    int* otherCount = 0;
                if(otherObjectives.find(objTown) ==
                    otherObjectives.end())
                {
                    otherCount = &otherObjectives[objTown];
                    *otherCount = oldObjective->spAllocated() -
                                  oldObjective->spNeeded();
                }
                else
                    otherCount = &otherObjectives[objTown];

                if(*otherCount >= spCount)
                    *otherCount -= spCount;
                else
                {
#ifdef DEBUG
                    d_sData->logWin("Can not use %s because it would
break objective at %s",
                         (const char*) infUnit.cp()->getName(),
                         (const char*) d_sData->campData()->
                                        getTownName(objTown));
#endif
                    continue;
                }
              }
            }
          }

          allocatedUnits.push_back(infUnit);
          spAlloced += spCount;
        }
      }

      if (spAlloced < spNeeded)
      {
#ifdef DEBUG
        d_sData->logWin("Can not be achieved without breaking more
        important objective");
#endif
        return false;
```

```
    }

    /*
     * Assign the allocated Units to objective
     */

    Writer lock(d_objectives);
    AIC_Objective* objective = d_objectives->
                               addOrUpdate(tInfo.town(),
                               tInfo.importance());
    ASSERT(objective != 0);
    if(objective == 0) //lint !e774 ... always true
       return false;

#ifdef DEBUG
    d_sData->logWin("Creating Objective %s", d_sData->campData()->

    getTownName(tInfo.town()));
    d_sData->logWin("There are %d objectives", (int)d_objectives->
    size());
#endif

    objective->spNeeded(spNeeded);

    for (std::vector<TownInfluence::Unit>::iterator it =
              allocatedUnits.begin();
         it != allocatedUnits.end();
         ++it)
    {
       const TownInfluence::Unit& infUnit = *it;

       AIC_UnitRef aiUnit = d_units->getOrCreate(infUnit.cp());
       TownInfluence::Influence unitInfluence = infUnit.influence();
       // friendlyInfluence.influence(aiUnit.cp());

#ifdef DEBUG
        d_sData->logWin("Adding %s",
           (const char*) infUnit.cp()->getName());
#endif

       // Remove unit from its existing Objective
       // Unless it is already attached to this one

       AIC_Objective* oldObjective = aiUnit.objective();

       if(oldObjective != objective)
       {
          if(oldObjective != 0)
          {
             // Remove Unit from Objective
```

```
                // If objective does not have enough SPs then
                // remove the objective

                removeUnit(infUnit.cp());
            }

            ASSERT(aiUnit.objective() == 0);

            // Add it to the objective table

            aiUnit.objective(objective);
            objective->addUnit(infUnit.cp());
        }

        // Set priority to a higher value to
        // reduce the problem of objectives being
        // created and destroyed too quickly.

        const float PriorityObjectiveIncrease = 1.5;
        aiUnit.priority(unitInfluence * PriorityObjectiveIncrease);
    }

#ifdef DEBUG
    if(d_objectiveDisplay)
        d_objectiveDisplay->update();
    if(campaign)
        campaign->repaintMap();
#endif

    return true;
}
```

战争游戏中使用的典型 AI 系统如下：

- 与文明游戏使用的策略 AI 相同级别的 AI，但在战争游戏中，AI 更集中于直接的战斗体验。
- 经常使用数据驱动系统，因为大多数此类游戏对参与玩家都有大量的战斗，以及对每件装备、战术单元和位置的大量详细的统计资料。
- 脚本系统也经常用于精确模仿不同寻常的或重大的战斗运动以及特定指挥官在特定战斗中使用的策略。

14.4　飞行模拟游戏

另一个有固定市场的游戏是飞行模拟游戏，它试图精确模仿特定飞机的领航并给玩家一个真实的驾驶员座舱视角以及在实际飞机中能够用到的所有控制。最流行的例子是 Microsoft Flight Simulator，它最开始出现于 1982 年，一直到现在还非常成功。尽管纯粹的飞行模拟游戏并不具备真正的 AI(玩家基本上是在与重力作战，并设法不发生碰撞)，但已

经发布有一些飞行模拟游戏的变种，它们试图使该类型游戏更具吸引力。它们之中最有名的一些是基于 Star Wars 的游戏，如 X-Wing 和 Tie Fighter。这两种游戏在飞行模拟游戏中都是较为简单的(仅有一部分的驾驶员座舱控制，并且由于是在外太空飞行，故没有失速和奇异的大气干扰)，但它们给玩家提供了足够的模拟，使玩家真正沉浸到 Star Wars 世界，并让更多人领略到比过去尝试的更好的飞行模拟体验。Wing Commander 系列也属于这种类型。其他游戏，如 Descent，把飞行模拟游戏带入到 FTPS 游戏世界，因为它有点像玩飞行车辆的死亡竞赛。Privateer 和 Freelancer 游戏则向飞行模拟游戏中添加一个完整的故事，并取得了不错的效果。在此类游戏中还有无数的基于战争的飞行模拟游戏，在这些游戏中，可以像在战争游戏中那样执行历史性的任务，但是是从相关飞机之一的驾驶员座舱的角度，因此更有针对个人的感觉。

飞行模拟游戏使用的典型 AI 系统如下：

- 纯粹飞行模拟游戏没有竞争 AI 元素，玩家所做的就是与自然力量进行战斗，主要是重力和空气动力学，以保持对飞船的控制。有些此类游戏的确具有某种形式的 AI 系统，可以教玩家如何驾驶飞船，但它通常都是表明不同飞船系统和性能的脚本化序列。程序清单 14-3 给出了开放源码的飞行模拟游戏 FlightGear 的主 AI 循环，它具有一些与玩家混战的简单 AI 元素。
- 动作导向的飞行模拟游戏在某方面与动作赛车游戏很相似，即它们都需要一些能够胜任操纵游戏中的车辆以及能够处理游戏引入的额外元素(如在战斗中使用宝物等)的 AI 系统。这些游戏也可能包括基于陆地的 AI 控制敌人，并需要超越简单车辆控制的额外功能。这些游戏更像是其他复杂的组合类型的游戏，并混合使用 FSM、消息和脚本。

程序清单 14-3 FlightGear 的主 AI 循环

(经 GNU 授权摘录)

```
void FGAIAircraft::Run(double dt) {

    FGAIAircraft::dt = dt;

    double turn_radius_ft;
    double turn_circum_ft;
    double speed_north_deg_sec;
    double speed_east_deg_sec;
    double ft_per_deg_lon;
    double ft_per_deg_lat;
    double dist_covered_ft;
    double alpha;

    // get size of a degree at this latitude
    ft_per_deg_lat = 366468.96 - 3717.12 *
                     cos(pos.lat()/SG_RADIANS_TO_DEGREES);
    ft_per_deg_lon = 365228.16 * cos(pos.lat() /
                          SG_RADIANS_TO_DEGREES);

    // adjust speed
```

```
double speed_diff = tgt_speed - speed;
if (fabs(speed_diff) > 0.2)
{
   if (speed_diff > 0.0) speed += performance->accel * dt;
   if (speed_diff < 0.0) speed -= performance->decel * dt;
}

// convert speed to degrees per second
speed_north_deg_sec = cos( hdg / SG_RADIANS_TO_DEGREES )
                      * speed * 1.686 / ft_per_deg_lat;
speed_east_deg_sec = sin( hdg / SG_RADIANS_TO_DEGREES )
                      * speed * 1.686 / ft_per_deg_lon;

// set new position
pos.setlat( pos.lat() + speed_north_deg_sec * dt);
pos.setlon( pos.lon() + speed_east_deg_sec * dt);

// adjust heading based on current bank angle
if (roll != 0.0)
{
   turn_radius_ft = 0.088362 * speed * speed
                    / tan( fabs(roll) / SG_RADIANS_TO_DEGREES );
   turn_circum_ft = SGD_2PI * turn_radius_ft;
   dist_covered_ft = speed * 1.686 * dt;
   alpha = dist_covered_ft / turn_circum_ft * 360.0;
   hdg += alpha * sign( roll );
   if ( hdg > 360.0 )  hdg -= 360.0;
   if ( hdg < 0.0)  hdg += 360.0;
}

// adjust target bank angle if heading lock engaged
if (hdg_lock)
{
   double bank_sense = 0.0;
   double diff = fabs(hdg - tgt_heading);
   if (diff > 180) diff = fabs(diff - 360);
   double sum = hdg + diff;
   if (sum > 360.0)  sum -= 360.0;
   if (fabs(sum - tgt_heading) < 1.0)
   {
      bank_sense = 1.0;
   } else {
      bank_sense = -1.0;
   }
   if (diff < 30)  tgt_roll = diff * bank_sense;
}

// adjust bank angle
double bank_diff = tgt_roll - roll;
if (fabs(bank_diff) > 0.2)
```

```
{
    if (bank_diff > 0.0) roll += 5.0 * dt;
    if (bank_diff < 0.0) roll -= 5.0 * dt;
}

// adjust altitude (meters) based on current vertical speed (fpm)
altitude += vs * 0.0166667 * dt * SG_FEET_TO_METER;
double altitude_ft = altitude * SG_METER_TO_FEET;

// find target vertical speed if altitude lock engaged
if (alt_lock)
{
    if (altitude_ft < tgt_altitude)
    {
        tgt_vs = tgt_altitude - altitude_ft;
        if (tgt_vs > performance->climb_rate)
          tgt_vs = performance->climb_rate;
    } else {
        tgt_vs = tgt_altitude - altitude_ft;
        if (tgt_vs < (-performance->descent_rate))
          tgt_vs = -performance->descent_rate;
    }
}

// adjust vertical speed
double vs_diff = tgt_vs - vs;
if (fabs(vs_diff) > 1.0)
{
    if (vs_diff > 0.0)
    {
        vs += 400.0 * dt;
        if (vs > tgt_vs)  vs = tgt_vs;
    } else {
        vs -= 300.0 * dt;
        if (vs < tgt_vs)  vs = tgt_vs;
    }
}

// match pitch angle to vertical speed
pitch = vs * 0.005;

//############################//
// do calculations for radar //
//############################//

// copy values from the AIManager
double user_latitude = manager->get_user_latitude();
double user_longitude = manager->get_user_longitude();
double user_altitude = manager->get_user_altitude();
double user_heading = manager->get_user_heading();
```

```
double user_pitch = manager->get_user_pitch();
double user_yaw = manager->get_user_yaw();
double user_speed = manager->get_user_speed();

// calculate range to target in feet and nautical miles
double lat_range = fabs(pos.lat() - user_latitude) *
                                ft_per_deg_lat;
double lon_range = fabs(pos.lon() - user_longitude) *
                                ft_per_deg_lon;
double range_ft = sqrt(lat_range*lat_range +
                                lon_range*lon_range );
range = range_ft / 6076.11549;

// calculate bearing to target
if (pos.lat() >= user_latitude) {
    bearing = atan2(lat_range, lon_range) * SG_RADIANS_TO_DEGREES;
        if (pos.lon() >= user_longitude) {
            bearing = 90.0 - bearing;
        } else {
            bearing = 270.0 + bearing;
        }
} else {
    bearing = atan2(lon_range, lat_range) * SG_RADIANS_TO_DEGREES;
        if (pos.lon() >= user_longitude) {
            bearing = 180.0 - bearing;
        } else {
            bearing = 180.0 + bearing;
        }
}

// calculate look left/right to target, without yaw correction
horiz_offset = bearing - user_heading;
if (horiz_offset > 180.0) horiz_offset -= 360.0;
if (horiz_offset < -180.0) horiz_offset += 360.0;

// calculate elevation to target
elevation = atan2( altitude_ft - user_altitude, range_ft )
                    * SG_RADIANS_TO_DEGREES;

// calculate look up/down to target
vert_offset = elevation + user_pitch;

/* this calculation needs to be fixed
// calculate range rate
double recip_bearing = bearing + 180.0;
if (recip_bearing > 360.0) recip_bearing -= 360.0;
double my_horiz_offset = recip_bearing - hdg;
if (my_horiz_offset > 180.0) my_horiz_offset -= 360.0;
if (my_horiz_offset < -180.0) my_horiz_offset += 360.0;
rdot =(-user_speed * cos(horiz_offset * SG_DEGREES_TO_RADIANS ))
```

```
                + (-speed * 1.686 * cos( my_horiz_offset *
                                        SG_DEGREES_TO_RADIANS ));
    */

    // now correct look left/right for yaw
    horiz_offset += user_yaw;

    // calculate values for radar display
    y_shift = range * cos( horiz_offset * SG_DEGREES_TO_RADIANS);
    x_shift = range * sin( horiz_offset * SG_DEGREES_TO_RADIANS);
    rotation = hdg - user_heading;
    if (rotation < 0.0)  rotation += 360.0;

}
```

14.5 音乐游戏

最近开发的一个流行的游戏类型是音乐游戏。在某些方面，此类游戏是与掌上游戏Simon 同等意义的视频游戏，它假定玩家重复一些逐渐变长的音乐和视觉模式序列。第一个音乐游戏是 1997 年发布的 PaRappa The Rapper，并且从那以后游戏就包括了从唱歌到弹奏不同乐器再到跳舞的所有事情。它们在很大程度上都遵循同样的 Simon 规范。这些游戏是真正的谜题游戏，但更加模式化，因此持续重玩该游戏的玩家能够一步步地向前推进。

尽管许多此类游戏都只是让玩家与实际的音乐音符对抗，但有些游戏也的确包含了试图战胜玩家的对手。甚至 PaRappa 也具有一个最后的自由式阶段来完成这个游戏。但是，这些对手涉及的 AI 至多是脚本化的。然而，被播放的脚本应该考虑玩家的游戏级别，从而对手才能接近该挑战。但实际的即兴音乐使用尝试人类所做事情的类型并在此基础上增加更多复杂性(就像人类在拥挤的会议室所做的那样)的 AI，而这些在此类游戏中还没有用到。

在音乐游戏中使用的典型 AI 系统如下：

- 脚本用于匹配 AI 控制角色的运动、与歌曲进行对话和建立故事元素。
- 数据驱动的游戏玩法。在该玩法中，可将一般的光照系统(或其他视觉系统)关联到音乐分析软件上，并包含大量的歌曲。它的例子有 Vib Ribbon、Frequency 和 Amplitude。
- 有些音乐游戏具有额外的元素，如 Rez(一款滚动射击游戏)和 Chu Chu Rocket(沿袭 Bomberman 的一类益智或聚会游戏)。这些游戏使用相当简单的基于状态的或脚本化的智能系统，它们也一样对音乐进行处理。

14.6 益智游戏

益智游戏是一种较小的、具有一定技巧的简单游戏，它们通常持续进行，但随着时间推移其难度也逐渐增加。它们通常具有非常简单的界面，对于如何进行游戏的描述则更为

简单。但是，由于这种简单性，它们也是最让人上瘾和广泛流传的游戏之一。据说 Nintendo Gameboy 成为一种全世界现象的主要原因就是由于一个叫做 Tetris(参见图 14-6)的小游戏，并且一直以来玩得最多的计算机游戏还是 Freecell(伴随着 Microsoft Windows 产生的一个纸牌游戏)。这些游戏并不需要占用我们太多的注意力或时间。我们可以玩 10 分钟游戏，然后把它关闭。正是这些游戏的特性使得玩家能够体验到少许的挑战，而不需要去参与一些情绪性或费时的事情。

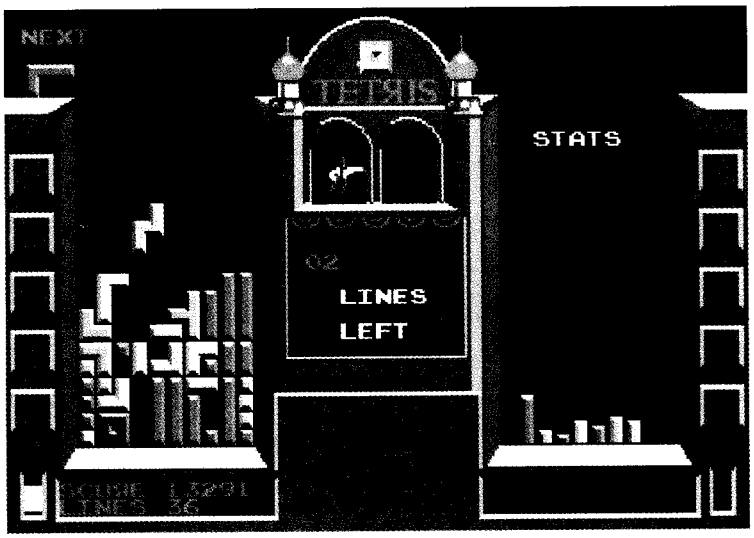


图 14-6　Tetris 屏幕截图

(Tetris®：Elorg 1987。未经允许，不得翻印)

这些游戏主要用于两个领域：在线游戏世界、移动电话和 PDA。在线游戏中益智游戏具有很重要的意义。我们可以用最少的资源(保持下载速度很低则更为理想)来编写一个益智游戏，并允许世界各地的人到网站来免费或接近免费地玩游戏。最小限度的游戏尺寸也有助于将它用在空间受限的移动电话和 PDA 中。人们希望当他们被堵在机场或等公交车时能够有可以娱乐的东西，而且大多数人已经拥有了这样的设备。一旦硬件可以支持，这就是一个自然的结合。不幸的是，大多数益智游戏并不真正使用 AI，玩法仅仅由简单模式或玩家必须要克服或解开的具体设置组成。然而，有的游戏也的确使用了 AI，例如 PopCap 公司的 Mummy Maze，尽管通常它只具有一个简单的基于状态的行为。

在益智游戏中使用的典型 AI 系统如非常简单的基于状态的行为(如果一个游戏具有 AI 元素)。

14.7　人工生命游戏

有人认为该类型不能算是游戏，但这种类型更像是基于宠物等的视频游戏。这类游戏并不多，但它们之中的某些使用了迄今为止我们所拥有的最先进的游戏 AI 编程技术。它们代表了外来 AI 技术在实时游戏体验中的最高点。人工生命游戏中的其他游戏不具备如此复杂的 AI，但代表了构建 AI 系统来最好地模仿传统困难元素的另一种方式。

该类游戏中的第一个游戏实际上是一个很小的电子装置，叫做 Tamagotchi，它们曾经在日本引发了巨大的狂热。它们基本上是基于液晶显示器(LCD)的小装置(钥匙链大小)，且其上面描绘有看起来很愚笨的动物。根据该动物拥有的一组需求，它要求被喂食，或被宠爱，或其他等。然后人类按下相应的按钮来给动物提供它所需要的东西。如果长时间内都没有执行正确的任务，动物会生气，甚或是死亡。如果正确做好了事情，它便会茁壮成长，并度过一个很长且完全的生命期，而且不断长大并会有一些微小的视觉区别，从而人们可以根据该区别来区分他们的宠物。尽管从游戏标准来看这是一个非常奇怪的概念，但它也是一个非常流行的游戏。

这些玩具最终导致游戏开发者利用这个假设来设计视频游戏。这样的例子有：Seaman(在该游戏中玩家是一条具有人的头脑的粗暴的鱼的管理者)、Monster Rancher(使用任意 CD 中的随机数据来创造独特的角色，之后训练该角色进行作战)和 Petz(纯粹的 Tamagotchi 式宠物)。与此具有相同主线的另一系列产品是由 Cyberlife 公司开发的 Creatures 系列。它们用于进化其游戏角色的实际系统使得这些游戏非常有名，然而其他游戏主要使用一些先进的模糊状态机(FuSM)，或只是对人类交互进行大量统计并把它们映射到一个较大的行为查找表中。Creatures 游戏已经走出了一条高技术含量的路线。它们使用先进的神经网络(NN)来模仿学习和情感，并使用基因系统来允许用户通过基因选择对动物进行交叉繁殖和进化。该产品几乎不能算是游戏，而更像是高技术含量的试验台，而且甚至是开发者也把它认为是一种技术演示。它们非常依赖 CPU 资源，并且由于具有非常多要学习的事情，故需要不断地运行。但从游戏 AI 的观点来看，它们给人的印象是非常深刻的。

其他类型的人工生命游戏力图从中得到更多真实的游戏体验，这包括 Wright 公司最新开发的一批游戏 The Sims™和 Molyneux 公司的 Black & White。

在 The Sims 游戏中，玩家正在控制的东西模拟的是人的生命。在游戏的开始阶段，玩家被提供一个 Sim，即具有许多需求的半自主角色。Sim 之所以是半自主的是因为他可以执行需求获取任务以保证生存(如果周围有食物而且他已经饿了，那么 Sim 将吃掉这些食物)，但是要真正变得优秀或发展，主要还是依靠人来照顾 Sim，让他更快和更有效率地执行他的职责，并鼓励进行额外的交互，特别是与其他 Sim 进行交互。游戏通过创造一个简单的 AI 范例 smart terrain 开辟了一片新天地。在这个概念中，智能体只有需要满足的基本需求(它足够聪明，能够在世界中到处走动，从而找到可以满足这些需求的东西)，并且具有一个模糊系统来使它具有一些偏好和初步的学习能力。但是该系统的真正智能是散布于陆地上的，嵌入进了居住在游戏世界中的所有对象。与 Sim 能够相互作用的每个游戏对象都包含了关于这种交互怎样才能发生和它能够给 Sim 提供什么东西(包括播放的动画)的所有信息。这样，新的道具可以随时添加到游戏世界之中，并能马上被 Sim 使用(考虑到支持游戏的扩展包的数量，这是显而易见的)。由于它们巨大的开放性，和由于非暴力带来的吸引力，以及游戏彻底的自定义和扩展能力，模拟游戏一直是最为畅销的游戏之一。

Black & White 则采用天神游戏的概念(具有一个小村庄的崇拜者，并且必须要照顾他们)，并添加了一个额外的元素：图腾动物。该角色由一个成熟的(根据游戏标准)AI 系统进行控制，包括动力学规则构建、决策树设计和简单神经网络(NN，称为感知器)的使用。该动物被认为是直接从用户身上学会大多数的行为，并且为了便于实现，游戏允许这些图腾以大量不同的方式进行学习，包括直接的命令、观察、反射以及直接从玩家进行反馈(玩家

可以拍打或抚摸该动物，以表示他做错了或做对了事)。通过允许该动物以这么多种方式进行学习，并影响他的信念和欲望，该动物的全部行为集具有很大的扩展性，因此，每个动物都是独特的。这也带来了比使用任意一种方法更快的学习效率。

在人工生命游戏中使用的典型 AI 系统如下：

- 大量使用 FuSM，因为它们很容易训练并提供了更多有指导的行为模式。
- 神经网络正逐渐被更多地研究和使用，随着开发者找到越来越好的方式来对它们进行训练并密切注意那些可能导致的非常错误的行为。
- 遗传算法正被使用在一些该种类型的游戏中，它便于训练程序，并帮助这一代游戏角色以多种方式进行进化。
- 人们也使用标准游戏 AI 技术(包括常规 FSM、消息和脚本)的一个可靠帮助系统。

第III部分 基本的AI引擎技术

在本书接下来的两部分中，将对现在游戏AI界中具体的AI技术进行详细讨论。

本部分将覆盖 AI 程序员所使用的较基本和通用的技术。对于每种方法，都将从下列各方面进行描述：

- 对技术的一个基本概述
- 对骨架代码进行解释
- 在AIsteroids试验平台上的示例实现
- 方法的优缺点
- 对范例一般实现的扩展
- 对技术进行优化
- 该方法如何适应第2章“AI引擎的基本组成与设计”中描述的8个设计上的考虑

各种技术的部分代码都将随着对它的解释在文中给出，并且多数代码以及项目和构成文件都可以在随书光盘中找到。

15 有限状态机

在游戏 AI 编程界中，没有任何一个数据结构比有限状态机(Finite-State Machine，FSM)使用得更多(除非把对开关状态的计数作为一种数据结构，但开关实际上也可以认为是状态机的一种简单形式)。这个简单而强大的组织工具帮助程序员将初始的问题分解为更容易处理的子问题，从而让程序员设计更加灵活和更加可伸缩的智能系统。即使读者没有使用过正式的 FSM 类或者功能，也很可能使用了该结构所遵从的原则，因为一般而言它是考虑软件问题的一种基本方式。因此，即使游戏对某个决策元素使用了其他的 AI 技术，也很可能在游戏中使用某种形式的状态系统。

15.1 FSM 概述

从本质上来说，一个“状态机”是一个包含 3 个因素的数据结构：该机器内在的所有状态、若干输入条件、作为状态之间连接性的转换函数。本书将对此稍作改变，将转换逻辑放到实际状态本身。因此，机器将存储这些状态，并更新当前状态，这需要一个恰当的转换。之后回到机器上来，从而激活该转换。图 15-1 给出了经典系统与本书使用的系统之间的区别。这样做是因为通过使状态成为简单的表示连接性的数据结构，可以防止机器成为储藏所有逻辑的仓库。相反，每个状态都是独立的模块，具有它的更新逻辑、转换逻辑以及进入和退出事件的特殊代码。

经典设计与本书将使用的设计的另一个区别是转换系统。在经典 FSM 方法中，转换是作为纯粹事件表示的，而且这些事件通常是某些类型的枚举列表，并能够被感知系统用来触发转换。之后每个状态将其转换注册到一个由输入输出对(例如，PLAYER_IN_RANGE 和 AttackState，或 SHOT_IN_HEAD 和 DeathState)组成的列表中。通过发送所有的状态到当前输入事件的机器中便可以实现转换检测，并且如果它对这个特殊的输入事件有响应的话，每个状态都将返回它的输出状态。

本书将改为给每个状态一个检测转换的函数(而不是一个检测注册过的转换的内部列表的基类函数，该转换源于输入触发类型)。这样，本书给出的骨架结构不仅能够执行经典的 FSM 系统任务，即通过创建一个输入类型的枚举然后测试它们之中是否有的已经在当前状态的转换函数中被触发，还能够在状态基础上允许更复杂的计算来确定状态的转换。本书给出的代码框架更加灵活，并为我们努力追求的代码模块化提供了可能。

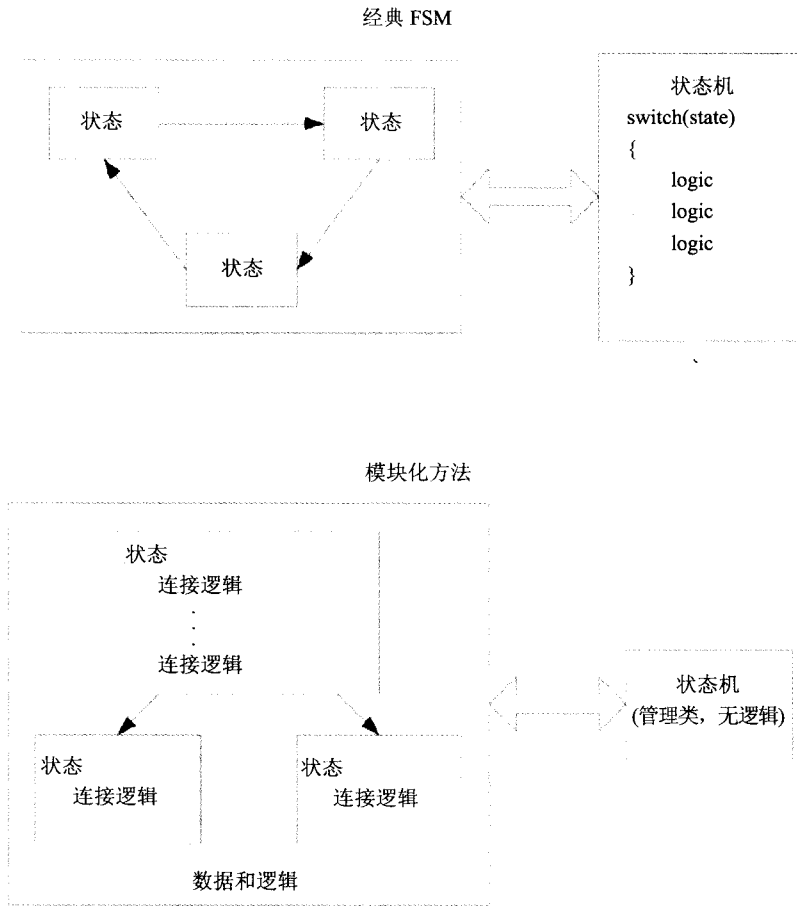


图 15-1　经典 FSM 与模块化 FSM 的执行流程比较

从经典的电子工程(从中我们首先借用了 FSM 的概念)来看，游戏中的大多数 FSM 都使用摩尔模型(Moore model)进行编码，这意味着我们将动作置于状态内部，而不是状态之间的转换(这是 Mealy 机器模型)。因此，在 Sit 状态中，角色播放一个坐的动画。在 Mealy 机器中，角色将在 Stand 状态和 Sit 状态之间的转换中播放坐的动画，而在 Sit 状态下除了等待转换到来外不执行任何动作。然而，通过放置一些智能代码，我们可以通过本章中的一般结构获得任何一种效果。

让我们看一个简单的例子，如图 15-2 所示。

图中给出的是 Blinky 的一个 FSM 框图，其中 Blinky 是 Pac-Man 中的一个 red ghost，是一个最直接追逐玩家的对象。所有的 ghost 都将从 Rise 状态开始其生命，因为它们通常位于迷宫的中央部分。在该状态下，ghost 会获取另一个身体(如果他还没有一个身体的话)，然后退出中央盒子。这样做会触发 FSM 转换到 Blinky 的主状态：ChasePlayer。它将保持在该状态之中直到玩家死亡或玩家吃了一个能量球(power pellet)。如果玩家死亡，Blinky 将转换到 MoveRandomly 状态。另外一种退出是转换到 RunFromPlayer 状态，这将导致 Blinky 逃跑，因为它因能量球而变得沮丧。当逃跑时，如果能量球用尽，它会回来继续追

逐玩家。如果被 *Pac-Man* 吃掉，它便转换到 Die 状态，同时变为一副眼珠子并回到迷宫中央。一旦进入中央，它便又转换到 Rise 状态，一切重新开始。

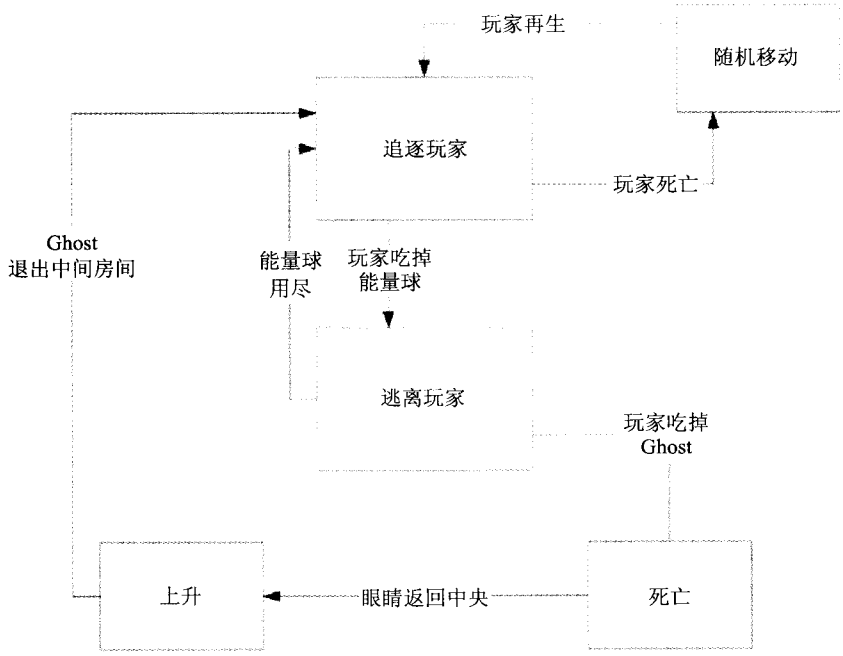


图 15-2　Pac-Man 中 red ghost 的简单 FSM 框图

因此，框图中给出了状态形成、输入条件和转换路线之间的清楚描述。另外，通过这样的划分，我们需要编码来实现整个 FSM 的原子行为。将 AI 系统的行为划分为原子单元是非常有用的，特别是在具有不同 AI 控制的角色，且这些角色之间只在某些少数方面不同或缺少某些特殊行为的时候。

Inky 的框图也是一样。Inky 是 Pac-Man 中的另一个 ghost，它不像 Blinky 那样具有攻击性，但根据不同条件能够在 RunFromPlayer、ChasePlayer 和 MoveRandomly 三种状态之间进行转换。它可能随机地在上述状态之间进行转换(完全不确定的行为)，或是基于与玩家的物理距离(避免或限制视线模拟)，或只是经常改变它的思想(因此它看起来是单一思想的，但反复无常)。当然，它仍然需要具有与 Blinky 一样的能量球和死亡逻辑，因为这是基本的 ghost 行为，而不是每个 ghost 的“个性”(根据它在迷宫内运动的定义)。

这种对 Blinky 状态的非常简单的 FSM 控制可以使用一个简单的转换声明，像程序清单 15-1 那样进行编码。事实上，许多游戏仍然为其简单的游戏元素使用此类自由形式的 FSM。然而，如果这不是 Pac-Man，而是 Madden Football，那么复杂性将急剧增加。可以想象，这种级别的组织结构难以满足要求，并过分复杂。由于取决于实现的顺序，转换优先级将变得越来越难以确定。放置该转换声明的函数将随着添加到函数中的状态的增多而变得日益庞大。本书使用的模块化系统将提供一个处理这些问题的形式组织模型。

程序清单 15-1　*Pac-Man* 的自由形式 FSM 实现

```
switch(m_currentState)
{
    case STATE_RISE:
    if(AtCenter())
         ExitCenter();
    else
         ChangeState(STATE_CHASEPLAYER);
    break;

    case STATE_DIE:
    if(!AtCenter())
         ChangeToEyesAndMoveBacktoCenter();
    else
         ChangeState(STATE_RISE);
    break;

    case STATE_RUNFROMPLAYER:
    if(!PoweredPacMan())
         ChangeState(STATE_CHASEPLAYER);
    else if(Eaten())
         ChangeState(STATE_DIE);
    else
         MoveAwayFromPacMan();
    break;

    case STATE_CHASEPLAYER:
    if(PoweredPacMan())
         ChageState(STATE_RUNFROMPLAYER);
    else if(!PacMan)
         ChangeState(STATE_MOVERANDOMLY);
    else
         MoveTowardsPacMan();
    break;

    case STATE_MOVERANDOMLY:
    if(PacMan)
         ChangeState(STATE_CHASEPLAYER);
    else
         MoveRandomly();
    break;

    default:
    PrintError("Bad Current State");
    break;

};
```

15.2　FSM 骨架代码

代码将通过以下类来实现：

- FSMState 类，系统中的基本状态。
- FSMMachine 类，放置作为状态机的所有状态和动作。
- FSMAIControl 类，放置状态机，以及游戏相关的代码，如感知数据。

下一节将对这些类进行更详细的阐述，并讨论 FSMAIControl 类的具体实现，和 AI 试验平台应用所需要的所有 FSMState。

15.2.1　FSMState 类

在设计一个状态系统时，最好把每个状态当作是世界中的唯一状态进行编码，状态之间相互不了解，或者状态机本身也不了解。这将带来非常模块化的状态，从而可以以任意的次序进行安排而不用考虑先决条件或未来需求。由于是最基本的，每个状态都应该具有下列功能：

- Enter()。当进入该状态时，该函数就会运行。它允许状态执行数据或变量的初始化。
- Exit()。该函数在离开状态时运行，并且主要作为一项清除任务，以及指向另外要运行的代码(在特定转换中希望发生的代码)。
- Update()。这是一个主要的函数，如果该状态是 FSM 的当前状态，那么该函数在 AI 的每个处理循环中都要被调用。
- Init()。对状态进行复位。
- CheckTransitions()。该函数用于处理决定状态结束的逻辑。该函数返回将运行状态的枚举值，如果不需要改变，则返回相同的状态。注意，决定逻辑状态转换的顺序变成了不同转换的优先级。因此，如果函数首先检测攻击状态的开关，然后检测躲避状态，那么这样的 AI 将会比相反检查顺序的 AI 更具攻击性。

这个类的骨架代码的 header 见程序清单 15-2。该类的复杂性已经保持在了最小值，因此该代码可以作为需要使用 FSM 来构建的任意系统的基础。这个类包含两个数据成员：m_type 和 m_parent。整个状态机以及根据被考虑的特殊状态而进行判断的局部代码都使用了类型域(type field)。对这些值的枚举存储在一个叫 FSM.h 的文件中，并且通常是空的，只包含默认的 FSM_STATE_NONE 值。当需要实际使用某个对象的代码时，便添加所有状态类型到这个枚举中，并从那里开始运行。单独的状态则使用父域(parent field)，从而它们能够通过它们的 Control 结构访问一个共享数据区域。

程序清单 15-2　状态的基类 Header

```
class FSMState:
{
public:
      //constructor/functions
      FSMState(int type=FSM_STATE_NONE,Control* parent=NULL)
        {m_type = type;m_parent = parent;}
```

```
        virtual void Enter()                  {}
        virtual void Exit()                   {}
        virtual void Update(int t)            {}
        virtual void Init()                   {}
        virtual void CheckTransitions(int t) {}

        //data
    Control* m_parent;
        int m_type;
};
```

15.2.2 FSMMachine 类

状态机类(它的 header 见程序清单 15-3)包括了 STL 向量中与机器相联系的所有状态。它还具有一个一般的 UpdateMachine()函数，该函数的实现由程序清单 15-4 给出。它还包括了向机器中添加状态和设置一个默认状态的函数。注意，状态机实际上源于状态类。这有利于实际上是一个完全不同状态机的状态。像状态类一样，机器类也具有一个类型域，它的类型在 FSM.h 文件的枚举中进行声明，目前基本上都是空的。

程序清单 15-3　FSMachine 类的 Header

```
class FSMMachine: public FSMState
{
    public:
      //constructor/functions
      FSMMachine(int type = FSM_MACH_NONE)
           {m_type = type;}
      virtual void UpdateMachine(int t);
      virtual void AddState(FSMState* state);
      virtual void SetDefaultState(FSMState* state)
           {m_defaultState = state;}
      virtual void SetGoalID(int goal) {m_goalID= goal;}
      virtual TransitionState(int goal);
      virtual Reset();
      //data
      int m_type;
    private:
      vector<FSMState*> m_states;
      FSMState* m_currentState;
      FSMState* m_defaultState;
      FSMState* m_goalState;
      FSMState* m_goalID;
};
```

程序清单 15-4　机器类中的 UpdateMachine()函数

```
void FSMMachine::UpdateMachine(int t)
{
     //don't do anything if you have no states
     if(m_states.size() == 0 )
          return;

     //don't do anything if there's no current
     //state, and no default state
     if(!m_currentState)
          m_currentState = m_defaultState;
     if(!m_currentState)
          return;

     //update current state, and check for a transition
     int oldStateID = m_currentState->m_type;
     m_goalID = m_currentState->CheckTransitions();

     //switch if there was a transition
     if(m_goalID != oldStateID)
     {
          if(TransitionState(m_goalID))
          {
               m_currentState->Exit();
               m_currentState = m_goalState;
               m_currentState->Enter();
          }
     }
     m_currentState->Update(t);
}
```

UpdateMachine()函数非常简单。它具有两个快速的优化：如果机器未被赋予任何状态，它会保证返回；如果没有当前状态集并且没有可依靠的默认状态，它也会返回。下一个程序块将调用当前状态的 Update()和 CheckTransition()函数，之后该程序块就已经事实上确定了该状态是否触发了一个转换。如果触发了转换，则 TransitionState()函数将查询机器的状态列表，以确定机器是否具有被请求的新状态，如果存在，则对系统要离开的状态调用 Exit()函数并对新状态调用 Enter()函数。

15.2.3　FSMAIControl 类

基本 FSM 系统的最后一部分(同时也是游戏相关代码的开始部分)是 Control 类(它在第 3 章“AIsteroids：AI 试验平台”中有简要的叙述)。该类是游戏内的主飞船的行为控制器，并且是人类控制和 AI 框架的主位置之间的分支点。因此，对于一个 AI 控制的飞船来说，我们将继承 AIControl 类并创造出子类 FSMAIControl 类(它的 header 可参见程序清单 15-5)。

程序清单 15-5　FSMAIControl 类的 Header

```
class FSMAIControl: public AIControl
{
   public:
     //constructor/functions
     FSMAIControl(Ship* ship = NULL);
     void Update(int t);
     void UpdatePerceptions(int t);
     void Init();

     //perception data
     //(public so that states can share it)
     GameObj*     m_nearestAsteroid;
     GameObj*     m_nearestPowerup;
     float        m_nearestAsteroidDist;
     float        m_nearestPowerupDist;
     Point3f      m_collidePt;
     bool         m_willCollide;
     bool         m_powerupNear;
     float        m_safetyRadius;

   private:
     //data
     FSMMachine* m_machine;
};
```

FSMAIControl 类包括一个标准的 Updata()函数，该函数对状态机进行更新，并运行 UpdatePerceptions()方法。该类也包括了游戏相关的黑板数据成员，它们将被机器中的所有状态所共享。如果这是一个更复杂的游戏，具有大量此类的全局数据成员(或多种简单和非常复杂的在某个特殊的感知级别需要大量管理的数据成员)，那么构建一个感知管理器类并且使 FSMAIController 类正好包含游戏中正确的感知管理器是个更好的办法。但是对于我们试验平台演示的简单需求来说，这就已经足够了。通过仅仅维持最小限度的数据成员列表，我们不必担心计算会花费太长时间，或费力地完成一个不实用的冗长的感知更新函数。

15.3　在试验平台上实现 FSM 控制的飞船

为了使用本方法开始 AIsterioids 程序，首先需要确定我们希望模仿的小行星游戏中飞船所展示出的行为的整体状态图。对于我们的应用场合来说，图 15-3 能够很好地实现这一目的。

如图 15-3 所述，AI 控制的 AIsteroids 飞船具有 5 个基本的状态：

- Approach 状态。将获得最靠近行星范围内的飞船。
- Attack 状态。将使飞船瞄准其射程范围内最近的行星，然后开火。
- Evade 状态。在一个碰撞路线中开始躲避一个行星。
- GetPowerup。设法拾取一定范围内的宝物。

- Idle。如果没有其他事件有效，则只是坐在那里。

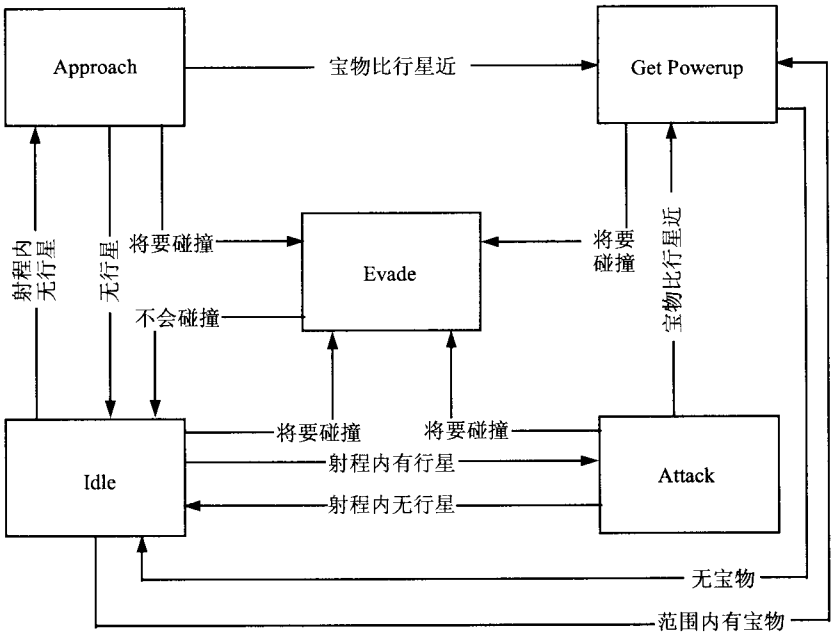


图 15-3　行星的 FSM 框图

游戏还需要下列条件以在这些状态之间建立起必要的逻辑联系：

- **射程内的行星**。需要进行一个简单的距离检测，以及跟踪最近的行星。
- **碰撞路线上的行星**。需要进行距离检测，以及一个轨迹横断相交。横断相交的代价相当昂贵，故只有行星处于距离检测内时才进行。
- **拾取范围内的宝物**。需要进行另外的距离检测，并跟踪最近的宝物。

另外，需要注意关于状态图的另一件事：每个状态都需要检测“行星位于碰撞路线上”的条件，然后转换到 Evade 状态。这里就引出了在每个状态上构造逻辑的一个内在缺陷，因为此类判断必须要在每个状态上都进行重复。但是，由于我们使用 Control 类的 UpdatePerceptions()函数作为一个全局数据位置(对于受该特殊 control 类影响的状态来说)，我们实际上是使用控制类来作为一个中央位置，这将使得该计算对整个状态机来说都是全局计算。这使得我们通过保持重复计算在最小限度(通过一个中央存储位置)以及使计算中的非重复部分仅仅在需要时才完成(通过在具体状态内进行逻辑和计算)充分利用了世界的优点。

15.4　示例实现

现在，我们将选取前面已经讨论了的 FSM 类并用它们来为我们的试验应用构造一个起作用的 AI 飞船。我们将首先建立 Control 类，然后为系统实现各个必备状态。

15.4.1　Control 类编码

FSM 模型的控制器类(程序清单 15-5 是该类的 header，程序清单 15-6 是重要函数的实现)包括状态机结构和该 AI 模型的全局数据成员。

该类的构造器通过对机器类进行实例化，然后添加各个必要状态的一个实例，从而构造了 FSM 的结构。构造器同时也设置了默认状态，它将用作机器的启动状态。

Update()方法非常简单，并确保了该类正在控制的飞船的存在性，并且如果这样，则更新感知和状态机。

有动作的地方就有 UpdatePerception()函数。最近的行星和宝物被记录下来，飞船与这些对象的距离也被确定，状态变量也被设置(m_willCollide 和 m_powerupNear)。这些感知使得单个状态中的所有转换检测变成是简单的比较，而不必单独对这些事情进行计算。该方法也增强了代码——更好或更快的方法可以在这里实现，而且其效果可以从状态中体现。

程序清单 15-6　FSMAIControl 类中函数的实现

```
//--------------------
FSMAIControl::FSMAIControl(Ship* ship):
AIControl(ship)
{
    //construct the state machine and add the necessary states
    m_machine = new FSMMachine(FSM_MACH_MAINSHIP,this);
    StateApproach* approach = new StateApproach(this);
    m_machine->AddState(approach);
    m_machine->AddState(new StateAttack(this));
    m_machine->AddState(new StateEvade(this));
    m_machine->AddState(new StateGetPowerup(this));
    m_machine->AddState(new StateIdle(this));
    m_machine->SetDefaultState(approach);
}

//--------------------
void FSMAIControl::Update(int t)
{
    if(!m_ship)
    {
        m_machine->Reset();
        return;
    }
    UpdatePerceptions(t);
    m_machine->UpdateMachine(t);
}

//--------------------
void FSMAIControl::UpdatePerceptions(int t)
{
    //store closest asteroid and powerup
    m_nearestAsteroid = Game.GetClosestGameObj
                        (m_ship,GameObj::OBJ_ASTEROID);
```

```
m_nearestPowerup = Game.GetClosestGameObj
                    (m_ship,GameObj::OBJ_POWERUP);

//asteroid collision determination
m_willCollide = false;

//small hysteresis on this value, to avoid
//boundary oscillation
if(m_willCollide)
   m_safetyRadius = 30.0f;
else
   m_safetyRadius = 15.0f;
if(m_nearestAsteroid)
{
   float speed = m_ship->m_velocity.Norm();
   m_nearestAsteroidDist = m_nearestAsteroid->
               m_position.Distance(m_ship->m_position);
   float dotVel;
   Point3f normDelta =  m_nearestAsteroid->m_position -
                        m_ship->m_position;
   normDelta.Normalize();
   float astSpeed = m_nearestAsteroid->
                       m_velocity.Norm();
   if(speed > astSpeed)
        dotVel = DOT(m_ship->UnitVectorVelocity()
                     ,normDelta);
   else
   {
        speed = astSpeed;
        dotVel= DOT(m_nearestAsteroid->
                UnitVectorVelocity(),-normDelta);
   }
   float spdAdj = LERP(speed/AI_MAX_SPEED_TRY
                       ,0.0f,50.0f)*dotVel;
   float adjSafetyRadius =  m_safetyRadius + spdAdj +
                            m_nearestAsteroid->m_size;

   //if you're too close, and I'm heading somewhat
   //towards you, flag a collision
   if(m_nearestAsteroidDist <= adjSafetyRadius
      && dotVel > 0)
        m_willCollide = true;
}

//powerup near determination
m_powerupNear = false;
if(m_nearestPowerup)
{
   m_nearestPowerupDist = m_nearestPowerup->m_position.
                          Distance(m_ship->m_position);
```

```
            if(m_nearestPowerupDist <= POWERUP_SCAN_DIST)
            {
                m_powerupNear = true;
            }
        }
    }
```

15.4.2　状态编码

下面 5 个程序清单(15-7~15-11)是对必要状态的实现。本书将对它们进行单独讨论，接着是相应的程序清单。

1. StateApproach

该状态的目的在于朝向最近的行星，然后向它推进。为了简单起见，该演示的 AI 系统并不试图处理游戏世界的环绕效果，因为那将需要更多的数学知识，而这并不是本书的焦点。

为了找到朝向最近行星的挺进角，Update()函数需要进行更多的计算，并且如果飞船速度过高，它还将增加一个“制动”矢量。这是为了防止 AI 控制的飞船因为速度过高而陷入困境。

在挺进角计算出来之后，该代码将使飞船转向正确的方向，或者如果飞船已经指向正确位置，则开启合适的推进器。与人类玩家相比，此类运动更为数字化，因此这看起来更像是机器人而不是人类。通过在转弯中使用推进器(这是大多数人类的方式)将使得运动看起来更自然，但这将使计算变得更复杂，而且本示例是专门为可读性编写的，而不是给出最优的实现。

CheckTransition()也是非常简单的，它轮流检测从该状态开始可能的 3 个转换：FSM_STATE_EVADE(如果希望进行碰撞)、FSM_STATE_GETPOWERUP(如果附近有一个powerup)、FSM_STATE_IDLE(如果没有可处理的行星)。

Exit()函数保证了在较大游戏世界中状态设置的任何系统都可以复位。这样，飞船的转弯和推进控制可以开启，也可以由此函数来关闭。

程序清单 15-7　StateApproach 类的函数

```
//--------------------
void StateApproach::Update(int t)
{
     //turn and then thrust towards closest asteroid
     FSMAIControl* parent = (FSMAIControl*)m_parent;
     GameObj* asteroid = parent->m_nearestAsteroid;
     Ship* ship = parent->m_ship;
     Point3f deltaPos = asteroid->m_position -
                        ship->m_position;
     deltaPos.Normalize();

     //add braking vec if you're going too fast
```

```
    float speed = ship->m_velocity.Norm();
    if(speed > AI_MAX_SPEED_TRY)
       deltaPos += -ship->UnitVectorVelocity();

    //DOT out my velocity
    Point3f shpUnitVel = ship->UnitVectorVelocity();
    float dotVel = DOT(shpUnitVel,deltaPos);
    float proj = 1-dotVel;
    deltaPos -= proj*shpUnitVel;
    deltaPos.Normalize();

    //find new direction, and head to it
    float newDir = CALCDIR(deltaPos);
    float angDelta = CLAMPDIR180(ship->m_angle - newDir);
    if(fabsf(angDelta) <2 || fabsf(angDelta)> 172)
    {
       //thrust
       ship->StopTurn();
       if(speed < AI_MAX_SPEED_TRY ||
          parent->m_nearestAsteroidDist > 40)
            fabsf(angDelta)<2? ship->ThrustOn() :
                          ship->ThrustReverse();
       else
            ship->ThrustOff();
    }
    else if(fabsf(angDelta)<=90)
    {
       //turn when facing forwards
       if(angDelta >0)
            ship->TurnRight();
       else
            ship->TurnLeft();
    }
    else
    {
       //turn when facing rear
       if(angDelta<0)
            ship->TurnRight();
       else
            ship->TurnLeft();
    }

    parent->m_target->m_position = asteroid->m_position;
    parent->m_targetDir = newDir;
    parent->m_debugTxt = "Approach";
}

//--------------------
int StateApproach::CheckTransitions()
{
```

```
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    if(parent->m_willCollide)
        return FSM_STATE_EVADE;

    if(parent->m_powerupNear&&(parent->m_nearestAsteroidDist
  >parent->m_nearestPowerupDist)&& parent->m_ship->
  GetShotLevel() < MAX_POWER_LEVEL)
        return FSM_STATE_GETPOWERUP;

    if(!parent->m_nearestAsteroid ||
      parent->m_nearestAsteroidDist < APPROACH_DIST)
        return FSM_STATE_IDLE;
    return FSM_STATE_APPROACH;
}

//--------------------
void StateApproach::Exit()
{
    if(((FSMAIControl*)m_parent)->m_ship)
    {
        ((FSMAIControl*)m_parent)->m_ship->ThrustOff();
        ((FSMAIControl*)m_parent)->m_ship->StopTurn();
    }
}
```

2. StateAttack

StateAttack 类将使飞船朝向最近的行星，然后启动大炮。该类可以通过调用飞船的 GetClosestGunAngle()函数解决多门大炮(通过获取宝物来给玩家提供多门大炮)的问题，这将给最近的大炮一个角度参数。

Update()函数计算最近行星的位置，但同时必须执行一些额外的计算以寻找行星的投射位置，从而找到一个超前角来命中运动中的行星。在找到该位置之后，它得到相应的角度，使飞船转弯，然后启动大炮。

该状态的 CheckTransitions() 函数与 StateApproach 状态的一样，具有 FSM_STATE_EVADE、FSM_STATE_GETPOWERUP 和 FSM_STATE_IDLE 三个分支。

该状态仅仅使飞船转弯，因此 Exit()函数仅需要考虑对相应的特殊标志进行复位。

程序清单 15-8　StateAttack 类的函数

```
//--------------------
void StateAttack::Update(int t)
{
   //turn towards closest asteroid's future position,
   //and then fire
   FSMAIControl* parent = (FSMAIControl*)m_parent;
   GameObj* asteroid = parent->m_nearestAsteroid;
   Ship* ship = parent->m_ship;
```

```
    Point3f futureAstPosition = asteroid->m_position;
    Point3f deltaPos = futureAstPosition - ship->m_position;
    float dist = deltaPos.Norm();
    float time = dist/BULLET_SPEED;
    futureAstPosition += time*asteroid->m_velocity;
    Point3f deltaFPos = futureAstPosition - ship->m_position;
    deltaFPos.Normalize();

    float newDir = CALCDIR(deltaFPos);
    float angDelta = CLAMPDIR180(ship->GetClosestGunAngle
                                 (newDir) - newDir);
    if(angDelta >1)
        ship->TurnRight();
    else if(angDelta < -1)
        ship->TurnLeft();
    else
    {
        ship->StopTurn();
        ship->Shoot();
    }

    parent->m_target->m_position = futureAstPosition;
    parent->m_targetDir = newDir;
    parent->m_debugTxt = "Attack";
}

//--------------------
int StateAttack::CheckTransitions()
{
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    if(parent->m_willCollide)
        return FSM_STATE_EVADE;

    if(parent->m_powerupNear && parent->m_nearestAsteroidDist
        >parent->m_nearestPowerupDist && parent->m_ship->
        GetShotLevel() < MAX_POWER_LEVEL)
          return FSM_STATE_GETPOWERUP;

    if(!parent->m_nearestAsteroid ||
        parent->m_nearestAsteroidDist > APPROACH_DIST)
          return FSM_STATE_IDLE;

    return FSM_STATE_ATTACK;
}

//--------------------
void StateAttack::Exit()
{
    if(((FSMAIControl*)m_parent)->m_ship)
        ((FSMAIControl*)m_parent)->m_ship->StopTurn();
}
```

3. StateEvade

StateEvade 是一个重要的状态，它设法通过执行推进机动和发射大炮清除道路来避免与行星的碰撞。

Update()函数用于计算一个包含飞船与行星之间路线的侧向法矢量的操纵矢量，并且如果是朝向行星，则添加一个制动矢量。之后，与 StateApproach 状态一样，Update()函数计算与该推力矢量之间的角度，使飞船转弯并在合适的时候向前推进，但在使用它的推进器时也将使飞船点火，有时这具有清除该区域的额外好处。

CheckTransition()函数仅需要检测一个状态，即 FSM_STATE_IDLE。我们可以直接检测面向其他状态的转换，但这并不需要。通过保持最小限度的状态连接，我们可以减少 CPU 运行状态机的需求(尤其是当转换判断比简单比较要复杂时)，并使得整个状态图更简单且易于将来的扩展(当希望向系统中插入更多状态时)。

StateEvade 状态的 Exit()函数与任何其他控制运动的状态一样，必须要复位被控制飞船的转弯和发动机状态。

程序清单 15-9　StateEvade 类的函数

```
//--------------------
void StateEvade::Update(int t)
{
    //evade by going to the quad opposite as the asteroid
    //is moving, add in a deflection,
    //and cancel out your movement
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    GameObj* asteroid = parent->m_nearestAsteroid;
    Ship* ship = parent->m_ship;
    Point3f vecSteer = CROSS(ship->m_position,asteroid->
                        m_position);
    Point3f vecBrake = ship->postion - asteroid->m_position;
    vecSteer += vecBrake;

    float newDir = CALCDIR(vecSteer);
    float angDelta = CLAMPDIR180(ship->m_angle - newDir);
    if(fabsf(angDelta) <5 || fabsf(angDelta)> 175)//thrust
    {
         ship->StopTurn();
         if(ship->m_velocity.Norm() < AI_MAX_SPEED_TRY ||
            parent->m_nearestAsteroidDist< 20 +asteroid->
            m_size)
              fabsf(angDelta)<5?
                   ship->ThrustOn() : ship->ThrustReverse();
         else
              ship->ThrustOff();

         //if I'm pointed right at the asteroid, shoot
         ship->Shoot();
```

```
    }
    else if(fabsf(angDelta)<=90)//turn front
    {
        if(angDelta >0)
            ship->TurnRight();
        else
            ship->TurnLeft();
    }
    else//turn rear
    {
        if(angDelta<0)
            ship->TurnRight();
        else
            ship->TurnLeft();
    }


    parent->m_target->m_position=asteroid->m_position;
    parent->m_targetDir = newDir;
    parent->m_debugTxt = "Evade";
}

//---------------------
int StateEvade::CheckTransitions()
{
    FSMAIControl* parent = (FSMAIControl*)m_parent;

    if(!parent->m_willCollide)
        return FSM_STATE_IDLE;

    return FSM_STATE_EVADE;
}

//---------------------
void StateEvade::Exit()
{
    if(((FSMAIControl*)m_parent)->m_ship)
    {
        ((FSMAIControl*)m_parent)->m_ship->ThrustOff();
        ((FSMAIControl*)m_parent)->m_ship->StopTurn();
    }
}
```

4. StateGetPowerup

该状态是识别宝物的位置，并试图接触到该宝物，以增强其战斗力。

该状态中的 Update()函数与 StateApproach 状态中的该函数非常相似，唯一需要的是更精确的接触，而不只是向一个大概的方向移动。因此，该状态必须要计算宝物的投射运动。

另外，也与接近状态一样，如果飞船速度太快，则施加一个制动因素，从而可以控制飞船的最大速度。之后，与其他状态一样，Update()计算一个新的方向，使飞船转弯，并让发动机点火。

CheckTransitions()用于判断从该状态出发的两个退出子句：FSM_STATE_EVADE 和 FSM_STATE_IDLE。

Exit()函数必须要对飞船的转弯和推进控制进行复位，以确保使它们处于一个中间位置。

程序清单 15-10　StateGetPowerup 类的函数

```
//--------------------
void StateGetPowerup::Update(int t)
{
     FSMAIControl* parent = (FSMAIControl*)m_parent;
     GameObj* powerup = parent->m_nearestPowerup;
     Ship* ship = parent->m_ship;

     //find future position of powerup
     Point3f futurePowPosition = powerup->m_position;
     Point3f deltaPos = futurePowPosition - ship->m_position;
     float dist = deltaPos.Norm();
     float speed = AI_MAX_SPEED_TRY;
     float time = dist/speed;
     futurePowPosition += time*powerup->m_velocity;
     Point3f deltaFPos = futurePowPosition - ship->m_position;
     deltaFPos.Normalize();

     //add braking vec if you're going too fast
     speed = ship->m_velocity.Norm();
     if(speed > AI_MAX_SPEED_TRY)
        deltaFPos += -ship->UnitVectorVelocity();

     //DOT out my velocity
     Point3f shpUnitVel = ship->UnitVectorVelocity();
     float dotVel = DOT(shpUnitVel,deltaFPos);
     float proj = 1-dotVel;
     deltaFPos -= proj*shpUnitVel;
     deltaFPos.Normalize();

     float newDir = CALCDIR(deltaFPos);
     float angDelta = CLAMPDIR180(ship->m_angle - newDir);
     if(fabsf(angDelta) <2 || fabsf(angDelta)> 177)//thrust
     {
        ship->StopTurn();
        if(speed < AI_MAX_SPEED_TRY ||
          parent->m_nearestPowerupDist > 20)
            fabsf(angDelta)<2?
            ship->ThrustOn() : ship->ThrustReverse();
        else
```

```
            ship->ThrustOff();
        }
        else if(fabsf(angDelta)<=90)//turn front
        {
            if(angDelta >0)
                ship->TurnRight();
            else
                ship->TurnLeft();
        }
        else//turn rear
        {
            if(angDelta<0)
                ship->TurnRight();
            else
                ship->TurnLeft();
        }


        parent->m_target->m_position = futurePowPosition;
        parent->m_targetDir = newDir;
        parent->m_debugTxt = "GetPowerup";
}

//--------------------
int StateGetPowerup::CheckTransitions()
{
        FSMAIControl* parent = (FSMAIControl*)m_parent;

        if(parent->m_willCollide)
            return FSM_STATE_EVADE;

        if(!parent->m_nearestPowerup || parent->
           m_nearestAsteroidDist < parent->m_nearestPowerupDist)
            return FSM_STATE_IDLE;

        return FSM_STATE_GETPOWERUP;
}

//--------------------
void StateGetPowerup::Exit()
{
        if(((FSMAIControl*)m_parent)->m_ship)
        {
            ((FSMAIControl*)m_parent)->m_ship->ThrustOff();
            ((FSMAIControl*)m_parent)->m_ship->StopTurn();
        }
}
```

5. StateIdle

最后一个必要的状态仅仅是一个捕获所有事情的状态，它纯粹是一个瞬时状态。本书这个简单演示中的状态机具有很少的状态，所以 StateIdle 状态与机器中的其他所有状态都具有连通性，但这并不是必要的。如果我们开始向该游戏中添加额外的行为(例如专门的攻击状态)，那么状态树中的这些状态会更加孤立。但是该游戏的简单性使该状态成为其他状态的一个通用的返回点，因此当飞船做完其他事情之后将始终退回到该状态。

该状态的 Update()函数除了在向屏幕显示调试信息时为调试系统提供一个使用标志外不执行任何其他操作。

由于在该游戏中存在空闲状态的基本特性，因此使用 CheckTransitions()函数来决定向游戏中其他状态的转换。

该状态不存在 Exit()函数，因为在更大游戏意义上它不做任何的改变。

程序清单 15-11　StateIdle 类的函数

```
//--------------------
void StateIdle::Update(int t)
{
    //Do nothing
    FSMAIControl* parent = (FSMAIControl*)m_parent;
    parent->m_debugTxt = "Idle";
}

//--------------------
int StateIdle::CheckTransitions()
{
    FSMAIControl* parent = (FSMAIControl*)m_parent;

    if(parent->m_willCollide)
        return FSM_STATE_EVADE;

    if(parent->m_nearestAsteroid)
    {
        if(parent->m_nearestAsteroidDist > APPROACH_DIST)
            return FSM_STATE_APPROACH;

        if(parent->m_nearestAsteroidDist <= APPROACH_DIST)
            return FSM_STATE_ATTACK;
    }

    if(parent->m_nearestPowerup)
        return FSM_STATE_GETPOWERUP;

    return FSM_STATE_IDLE;
}
```

15.5　使用该系统的 AI 的性能

利用这个简单的框架，AI 能够很好地进行行星游戏，而且能够偶尔达到两百万的分数。在 StateEvade 状态下附加的射击行为似乎是系统能够在随后的关卡中生存的关键，因为飞船几乎不停地在躲避巨大数量的行星。然而，仔细观察就会发现，很多事情都还存在改进的空间：

- **添加一些特殊状态**。获取第一个宝物能够显著提高 AI 的生存机会，因此这可作为一个优先状态。在没有很多行星时就明确地装满宝物将会非常有用，因为这将带着最多的武器进入下一个关卡。另外，如果人类恰好具有所有的宝物，然后坐在屏幕中间不断地旋转并开火，那么他能够持续玩这个游戏。在适当时间，AI 也可以使用这种“螺旋式死亡发展”攻击方式，比如当它被包围的时候。另一种状态是利用无可匹敌的特性，即 AI 飞船能够走捷径来获取宝物或当无敌时可以忽略躲避策略。
- **增强数学模型的复杂性**。这使得 AI 系统能够处理世界中的缠绕事件。现在，AI 的主要缺陷在于当世界中的事情缠绕在一起时它将失去焦点中心；在瞄准和避障中对此进行考虑将极大地增强 AI 飞船的生存能力。
- **飞船的弹药管理**。现在，飞船只是瞄准，然后开始点火。在弹药上不存在点火率，因此它往往是向目标发射一群炮弹。有时候这是有利的，当它向一个大行星发射一群炮弹时，残余的炮弹有时可以杀死行星分裂时的碎片。但当它发射了为它配备的全部炮弹时且必须等待它们在能够再次发射之前碰撞或死亡时就会遇到麻烦，使它暂时不具有防御性。
- **更好的适合攻击的飞船配置**。这意味着飞船不会经常丢失掉快速移动的目标。人类往往是移动到行星将最终到达的地点，然后在该位置停下并等待行星的到来。由于演示中的数学被故意设计得非常简单，该系统是直接朝行星运动。甚至是这么一种简单的方法也会因为世界的缠绕效应而出现问题。这种玩游戏的方法并不如人类方案看起来那么智能。
- **更好的躲避行为**。现在，飞船仅仅为避障采取简单的操纵行为(稍做修改，因为我们只能正向和逆向推进)。人类则使用一种复杂得多的避障判断，包括潜在碰撞下的射击(不对推进做任何调整)、注意到一群行星的到来并作为一个群体来进行躲避、在行星接近之前进行抢先定位或为了放慢行动的速度施加一个制动的停止动作。对球场进行简单的分析有助于 AI 更好地处理这些动作。通过了解地图中的哪部分具有较少集中的行星，它可以往“更多空间”的一般方向执行躲避战术，甚或先发制人建立一个低集中度的区域，以便使其自身得到一个更好的生存机会。

15.5.1　基于 FSM 系统的优势

FSM 可以很容易并直观地进行描述，尤其是在处理摩尔式机器时。我们在试验平台上的设计使用的是摩尔式状态机，其行动包含于状态之中(而不是在转换之中)，该设计关注

大多数人是如何考虑 AI 行为的。然而，甚至在本范例内，也可以对演示游戏进行多种方式的 FSM 编码，它们可以实现相似的性能。

从本章可知，FSM 也很容易实现。给定一个精心设计的状态图，状态机的结构事实上就已经确定。简单性是 FSM 最大的优势，因为该方法的特性使得它非常有助于将 AI 问题分解为具体的各块并定义各块之间的连接性。用不了多久，编写 FSM 结构对于大多数程序员来说都将成为一项机械式任务。

基于状态的系统容易进行添加，因为游戏流程非常确定并且状态之间的连接也非常具体。事实上，复制 FSM 图(如果非常大则是它的一部分)并随着用户对系统进行扩展而持续保持它的通用性。这将增强维持用户对整体 FSM 结构的了解并帮助用户找到需要连接但却没有连接的逻辑漏洞或领域。此类记录甚至可以通过往状态中插入特殊的调试代码来实现，从而游戏可以有效地将状态图写入到一个文件中并进行离线检查，寻找丢失或放错位置的变换。

FSM 方法也非常容易调试。状态机的确定性本质使得它容易(通常是这样)复制缺陷，并且 FSMMachine 类的集中本质也使得进行代码定位变得容易，从而当具体的 AI 角色或行为出现时可以进行捕捉。在该范例中视频调试也变得容易，因为对单个角色来说在屏幕上输出状态信息以及观察 AI 在空闲时进行决策都是微不足道的。此类信息在一定条件下也可以非常有用地编写成状态转换的一个日志。

最后，由于它们的非特殊性本质，FSM 系统能够用于所有问题，从屏幕之间简单的游戏流程，到最复杂的 NPC 对话。这种内在的通用目的性意味着在一定程度上，几乎每个游戏都将具有某种基于状态的元素。并不是说非常简单的状态系统也需要运行一个完全形式的架构，而是几乎每个游戏都将简单地使用某种形式的 FSM，因为它们可应用到如此广泛的不同游戏问题中。

15.5.2　基于 FSM 系统的劣势

FSM 系统的主要优势，即它们很容易实现，有时往往也是它们的最大劣势。当基于状态的系统不是从一开始就以一种静态框架进行设计，而更多的是使用基于 switch 和 case 的 FSM，混合更多的形式状态机时，则该项目会出现问题。程序员有时能够快速地对一个行为进行编码(在压碎的过程中，或在试验的瞬间)，然后不厌其烦地返回并在整个游戏结构中正确地对它进行重新实现。这将导致碎片式系统，其逻辑在各个方向和位置上分散，组织上也不完备，从而带来维护上的困难。

在项目过程中，FSM 系统会随着找到更多的特殊行为(例如本章开头提到的那些能够改进行星玩法的 FSM)而变得越来越复杂。尽管随着时间流逝设法改善 AI 系统看起来不错，但 FSM 往往不能很好地适应这种迭代式工作。转换数量随着新加入的状态数量而呈指数增加，状态图也随之变得极其复杂，因此转换的判断和行动的优先权变得难以解决。

基于状态的模型的另一个不足是状态震荡问题。当分离两个或更多状态的感知数据非常脆弱，即不存在交叠空间时就会发生状态震荡。例如，游戏中的一个动物(如图 15-4 所示)只有两个状态：Flee 和 Stand。Flee 是直接从离他小于四英尺的敌人中潜逃，而 Stand

只是让该动物站在那。现在，一个敌人角色进入了该场景，并站在离动物 3.99 英尺的地方。动物则进入他的 Flee 状态，但当他开始他的动画时，他的位置稍稍发生变化，并突然离敌人 4.001 英尺。因此他又转换到 Stand 状态。Stand 状态播放不同的动画，从而可能使他返回到接触状态，再次开始整个循环。这是一个非常具体和简化的例子，但从中可知如果不注意，状态系统的内在脆弱性可以导致类似的震荡状态。解决该问题的一些方法将在下面章节中给出。

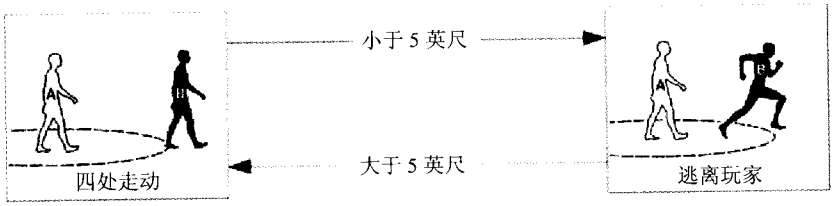


图 15-4　一般的基于状态的震荡问题

15.6　范例扩展

由于 FSM 的实现具有很大的开放性，为了克服 FSM 系统的缺点，这些年出现了很多有用的变种。在这里对这些扩展中较为有用的几个进行讨论。

15.6.1　层次化 FSM

有时，FSM 中给定的状态会非常复杂。在 Aisteroids 示例中，为了使躲避状态更加完美，需要把该状态变得非常复杂。可以编写一些特例代码来分离玩家被包围或一组行星正向玩家走来等情形。其他代码可以通过移动到更加开放的区域来支配碰撞，或者为即将到来的交通而直接扫平道路。在当前 evade 的 Update()方法中应该注意这些事情，但处理这些的一种更好的方法是使 evade 状态成为一个完全不同的状态机。在该状态机内部，可以反复处理这些威胁，并将代码划分成更容易处理的部分。因此，evade 状态机应该包含一些状态，首先处理被包围的情况，然后通过射击或躲避来处理临近的威胁，然后设法到达一个更安全的位置，从而能够完全退出 evade 状态。

该方法是一种向 FSM 系统增加复杂性而又不在更大的状态机中创建不恰当的连接的一种很好的方式。事实上，是在将状态分组到更加局部的范围里，并利用这些局部状态之间的相似性。像 AIsteroids 示例中的 FSMAIControl 结构所做的一样，通过在它们自己的状态机内分组相似的状态，包含这种新机器的“超状态(super state)”也能够具有通用功能和共享的数据成员。

或者，子状态也不一定是真正的状态。另一个通常使用的技术是使较大 FSM 中的状态包含许多子状态，然后随机地(或者根据感知触发器的一定组合)触发其中的一个等价子状态。这与在经典状态图上让两个或更多的状态具有相等的分支是一样的，但在选择哪个分支嵌入到状态更新方法时使用了一定的逻辑，而不是间接地通过感知次序优先级或其他一些迂回方式。

15.6.2 基于消息和事件的FSM

在某些游戏(或仅仅某些状态)中，转换将经常发生。如果是这样，而且如果游戏包含大量状态或确定转换的计算非常复杂，那么以轮询方式来检测转换在计算量上将变得代价昂贵。相反，使用消息作为触发器而不是轮询的FSM系统可以很容易实现。

状态机框架的整体结构应该要转换到使用此类系统。游戏(在某些方面主要通过Control类)必须要传递消息到状态机中，然后由状态机将它们分发到不同状态中去。FSMMachine::UpdateMachine()方法将成为状态机的消息井(message pump)，而且各状态的Check-Transitions()函数将成为处理它希望考虑的不同消息的转换声明。代码的其他部分通常保持不变。甚至是Enter()、Exit()和Update()等函数都可以被系统内自动发送的消息所触发。注意：可以实现组合系统，其每个状态都存储一个反映是轮询状态还是事件驱动状态的标志，并且UpdateMachine()函数能够对它作适当处理。

15.6.3 具有模糊转换的FSM

可以编写FSM使得不使用事件或其他类型的感知触发器来引起机器中的转换，而是使用模糊判决(比如简单比较或计算)来触发状态转换。由于本章中的框架已经按该方式编写，故该技术在实现上不需要做任何代码上的改变。事实上，本章前面提到的AIsteroids实现使用了该技术。如果它是使用FSM的较传统的方式编写的，那么所有的状态转换逻辑都应该在Control类中执行，并且每个状态的CheckTransition()方法将由输入事件触发。例如，在StateIdle状态中，CheckTransition()函数将检测附近是否存在一个行星，如果存在，则检测与它的距离，然后分配一个转换。经典设计的FSM将由Control类来进行存在性和距离检测工作，然后传递(或设置一个函数将检测的Boolean值)该输入类型ASTEROID_ASTERIOD_CLOSE_TO_PLAYER，然后idle类使用它来分配到攻击状态的转换。本示例中的转换仍然定义得很脆弱，但它们可以具有一个考虑斜坡相位的更模糊判决(从而它可以在设定的“反应时间”内注意到行星)，或者是某一设定的最小时间(从而飞船不会脱离出该状态，直到已经过了最小时间)，或可能需要的任意其他类型的计算。

通过采用这种更为灵活的方法来分配转换，该代码框架为其他分配转换的更丰富的方法打开了方便之门。它也使得所有权逻辑计算在状态本身的范围内进行，而不是在大的控制器类中(在其感知功能内执行所有的逻辑)。

15.6.4 基于堆栈的FSM

常规FSM设计的另一个变种是在状态机类中扩展m_currentState成员，成为一个堆栈数据结构。由于机器是从状态到状态进行转换，它通过将以前的状态推入堆栈中来保持一个对以前状态的历史。一旦一个状态被完全结束，它将从堆栈中弹出，从而下一个顶端状态就成为了当前状态。这使得角色具有一个有限形式的存储器，而且其任务可以被中断(通过来自另一角色的命令，或处理更紧迫的事件，如突然被击中)，但中断被处理之后，它们又返回到它们以前的工作。

在使用这种变种时必须要小心，当进入或退出当前状态时通过中断来清除所有的错误堆栈问题。因此，假定一个处于Patrol状态的AI控制角色因为被玩家狙击而发生中断并立

即转换到 Take Cover 状态。如果该角色被击中，那么在狙击危险清除后返回到 Patrol 状态的确没有任何意义。被 Take Cover 状态中断的 Patrol 状态事实上应该通过一种“置换”行为来进行标记，因为它替换 Patrol 状态成为堆栈中的顶端行为。这个新状态也应该设置一个退出行为，不管它是否受伤，从而 AI 能够到达某个能够更说得通的状态。这样，当角色从隐藏地出来时，他不会仅仅是盲目地开始再次巡逻，而是寻求帮助(当受伤时)或调查炮弹来源地。当然，如果这是希望游戏做的事情，则另当别论。

15.6.5　多重并发 FSM

同步或协同多个 FSM 的问题可以划分为两类：多个角色之间的 FSM、控制单个角色的多个 FSM。多角色协同通常由一个某种类型的管理器进行处理，该管理器是一个观测器，从上级给这些角色下达命令并设置复杂的场景作为各式各样的木偶戏。有的游戏智能地使用常规 FSM 系统来处理此类活动，在这些系统中只是简单地相互对抗、基于状态方式，但彼此之间并不了解。

稍微不同寻常的一种情形是角色内的多个 FSM 交互。这需要角色能够真正地同时做两件事情。这与 Robotron 中的 AI 角色一样简单和直接，它们中的一个 FSM 用于运动，另一个 FSM 用于射击(尽管在 Robotron 中这两个系统是完全独立的，更适于使用模糊状态机，可参见第 16 章“模糊状态机”)。这与即时策略(RTS)游戏的 AI 对手中彼此一块运行的一系列 FSM 一样复杂。

该对手需要对资源管理、研究、战斗等使用独立的决策状态机。这些 FSM 可以通过某种类型的一个观测器(甚至可能是另一个 FSM，即把其他 FSM 的输出作为转换条件的一个“通用”FSM)进行相互通信，通过一个共享的数据区域(像我们在 AIsteroids 上的 FSM 实现一样)或在状态和状态机之间传递消息和事件数据。在此类系统中需要注意的事情是当进行网络化编码或并行处理系统相遇时会出现问题。一个状态机可能会重写另一个不同的状态机需要的共享数据成员；两个状态机或许会相互形成一个反馈循环，导致发生震荡；由于处理定时问题或相似原因，某些计算的固有次序得不到保证。

15.6.6　数据驱动 FSM

向着更丰富定义 AI 行为集的推动力使得许多开发人员考虑设计他们的 FSM 系统以使得其建造过程主要由非程序员(可能是设计者和制作者)完成。这意味着不需要程序员太多的参与就可以将新的(或改进的)行为添加到系统之中，从而赋予项目中更多人塑造游戏玩法的能力。存在许多不同的方法来设计一个数据驱动的 FSM 系统。实现此目的的一些更普遍的方式是通过使用如下方法：

- 使用实际文本文件或在常规编码环境中的简单宏语言的脚本化 FSM。这是最容易设计的，但也需要设计者具有更多技术上的努力，尤其是因为脚本语言最终成为常规语言的一个子集(尽管有的是 Python、LISP，甚至是汇编代码方式的脚本语言，但大多数通常都是 C 语言的小版本)。脚本系统的简化版本可以单独由一般的比较运算符(>、<、==、!、=等)组成，并且脚本编辑器通过定义状态(使用预定义的变量和数值)之间的转化连接来建立状态机。宏语言比完全的语言解释器要稍微容易实现(除非是极其简单的语言)，并具有实际编码的优点，使得调试更加容易。它们

同时也具有编码上的不足：设计者必须要对游戏进行编译以运行他们的新脚本(以及明显需要公司购买编程环境的额外拷贝)，尽管这能够通过对这些宏文件使用现代的源码控制工具来进行补偿，并因此通过自动合并提供诸如多人工作于同一个文件等功能，以及设置未经允许不能改变的保护文件。

- 编写视频编辑器，从而允许设计者采用与使用标准FSM图对它们进行原型化以说明状态连接性和系统流程的相同方式建立FSM。此类系统非常便于设计者使用，但需要比其他系统投入更多的精力用于编码。必须要对常规游戏进行编写，从而在编辑器上显示状态、转换条件以及其他信息，从而设计者能够在该列表变长或变化时根据这些元素建造FSM图。除此之外，必须要在产品的整个生命周期内编写和维护编辑器本身(有些情况必须要超越产品的生命周期)。

15.6.7 惯性FSM

FSM的问题之一是状态震荡的概念(本章前面有详细的论述)。这是由于导致状态之间转换的事件的发生相当接近。篮球比赛中跟踪玩家是否具有一条到篮板的开放路线的感知就是一个例子。该感知应该这样设计：首先在玩家和篮板之间进行视线检测，然后再进行与其他队玩家的碰撞视线检测。如果经常执行这种检测(假定现在还没有进行优化，并在每帧上都进行检测)，那么就可以知道该玩家很容易在Stand状态和DriveToTheBasket状态之间急剧波动，因为随着其他玩家在球场上运动，碰撞视线可以在每帧上轻微变化。这正是必须要避免的一类行为，否则角色将看起来容易颤动。

克服这个的方法是向系统中引入“惯性”的概念。这仅仅意味着如果状态已经被激励，那么它将保持激励一定时间，或者一个新状态在它首次被唤起前必须要克服一定的惯性。这可以在两个层级实现：状态本身或唤起该状态的感知。

在状态层级，状态机本身能够跟踪当前状态并可能拥有最小的运行时间(改变的惯性，或所有能够认为是AI系统的单一思想的东西)，或者即将到来的状态在它实际上成为当前状态前需要请求提升为当前状态好几次(静态惯性，类似于某种环境认知或可称为反应时间的东西)。这样，感知可以尽量粗糙，并且状态机会对感知流进行采样以注意到感知变量的“趋势”(而不是单个数据变化的尖峰信号)，并用这来使状态发生变化。在状态层级，也可以在检测转换时使用时间函数；状态在机器中作为当前状态的时间越长，从它开始的转换就越有可能。仍然存在转换出界，它们只是随着时间发展而变得更加容易获取。

感知层级的惯性则完全相反。状态转换非常脆弱，但感知事件的激励则以它们代表系统中惯性的方式进行模仿。感知可以以多种唤起方式进行激励(反应时间)，需要一定程度的感知来唤起(敏感性)，在感知结束后继续保持激励(斜坡下降，或消失的敏感性)，或者是在它们本身被唤起前需要另外的感知来唤起，甚至是在最开始感知的数值是真实的时候(先决条件，或串联式激励)。

有时候更希望来自感知部分的惯性，因为感知可以共享为跨越许多不同状态的触发器，因此，将惯性并入一个单一的、通常使用的感知中可以在系统的很大一部分中阻止震荡。但是，来自状态部分的惯性更为一般并潜在地能够更快地实现。可以使用这两种方法的组合来非常简单地获取希望系统具有的精确平滑度(与反应性相对)。

最后，记住，如果 AI 系统需要极端的反应性(例如，在动作类游戏中，具有非常快速的赌博需求和即时的 AI 玩家反应)，那可能需要放弃这些类型的决策平滑技术，并反过来依赖动画引擎之类的东西来帮助平滑颤抖的角色。如果动画引擎具有构建在混合系统中的一定程度的惯性，或者是在动作改变的一刻或几刻时间内时不改变动画，那么 AI 系统能够有效地进行相当多的跳跃，并且不会损害游戏的整体表现。然而，最后，该层级的反应性很少是必需的，因为在六十分之一秒(或更少)反应的敌人通常并不认为是更智能的，而且最终不会给人类带来太多乐趣。现在有一个具有超过人类反应时间的怪物头目，而玩家必须要在它身上使用一个道具来降低它的性能，然而……

15.7　最优化

FSM 易于编码，并很可能是所有 AI 方法中最有效率的一种，因为它们从组织结构上和计算能力上将代码逻辑地划分成可易于处理的几块。但是，在算法(处理速度)和整个数据结构(存储器使用等)上都还存在优化的空间。一般的技术包括如下几个。

15.7.1　FSM 和感知的负荷平衡

负荷平衡是指分散那些随着时间过去需要完成的计算量，从而减轻处理器的即时负荷。可以把这看作是通过分期付款来购买东西：得到了对象，但还有逐渐增加的成本。在购买中，该成本是支付的利息。在我们的系统中，该成本是必须要为 AI 和感知系统设计时间调度系统或设计递增算法的一般管理费用。

负荷平衡通常由两种方式之一来进行处理(两种方法都可以工作在 AI 层级和感知层级)：使系统按一个设定的或预定的速度(如每秒两次，或每隔一秒等)运行，或使系统在给定更多时间时可递增得到更好的结果。许多路径搜索系统工作于第二种方式，它们最初只给出一个运动的粗略方向，然后随着花费在算法上的时间的增加得到越来越好的路径。沿着该路线的另一种系统是可中断的 FSM 系统，在一个设定的时间限制之后整个机器能够停下来，然后当它获得系统中的另一个时间片时再重新运行。

此类的计算复杂性并非对所有事情都是必要的，因为对大多数感知(我们在模仿人类的行为，而且人类自身的感知系统很少以超过每秒 60 帧的速度工作)和一般的 AI 决策系统(同样，人类很少以每秒 60 帧的速度改变他们的主意)来说，简单的时间调度就已经能够很好地工作。如果需要调度的事情的数量很大，分散所有计算的一个较好的方式是使用一个自动负荷平衡算法来设法使得处理过程中经常出现的尖峰最小化，同时系统程序员保持对更新调度的粗略控制。此类算法保持对计算时间的统计数据，并使用推断来预测不同游戏元素的未来需要，之后使用该数据来决定更新对象的次序，从而使处理过程变得平滑。

15.7.2　LOD AI 系统

LOD(Level of Detail，细节度)系统最初(现在仍然是)是 3D 图形程序员为了减轻需要执行的传输渲染工作量而引入的，通过使用由少数多边形和纹理构成的模型来显示遥远的物体，因为无论如何玩家都不会注意到他们之间的差别。在有些游戏中，玩家能够看得很远，

实际上是使用了某些 LOD 系统将游戏角色简化成了具有一定颜色的个别三角形。但由于距离很远，玩家不能进行分辨，而且渲染引擎也并不是始终都在运行，它将计算该角色通常使用的 2000 个多边形模型的所有事情。

同种思想也开始应用到 AI 工作中，因为我们现在正需要处理 CPU 消耗严重的 AI 程序，而且我们具有的仍然是游戏世界的一个有限玩家视角。因此，为什么不在玩家注意不到的情况下进行简化处理呢？不通过使用路径搜索系统来产生一条从 A 到 B 的真实路径，在世界另一部分中的人类角色将只是估计他需要多长时间才能到达 B 点，然后在该时间到来后简单地把他远距传输到那个位置(更好的方式是按块进行远距传输，从而尽量减小该行为可能弄糟事情的机会)。或者一个设法从人类玩家中逃脱的撤退角色只是在设定时间到来后就取回他的健康值，而不用实际去抓到健康宝物并使用它们。这看起来有点像是在作弊，而且如果过度使用则确实是作弊。但是，通过模仿(使用简单的估计而不是通常使用的代价昂贵的方法)事情随时间变化的效果，并确保人类不会撞上在错误 LOD 中的某人或在人类看不见之后 AI 马上使用 LOD，就可以减轻作弊的感觉。

正如图形界所反对的那样，LOD 系统在 AI 界中的问题是用于图形渲染的 LOD 系统大多是自动的。它们中的某些确实需要为 LOD 的每一个步骤设计特殊的技术，但其他将自动产生这些额外的细节度。然后，图形引擎只需要确定视线和离玩家的距离，从而使用正确的 LOD 来对角色进行显示。但在 AI 编程中，通常需要为每个 LOD 专门编写 LOD 用例，因此 LOD 系统只可用于那些能够有重大改善的游戏类型和玩法中。考虑一个具有到处走动和与环境交互的动态人群的游戏。在最靠近的 LOD 中，人群成员使用完整的避障、碰撞响应、使用面部表情和动画来进行相互交流、并产生像他们扔掉的垃圾那样的其他物体。在最远处的 LOD 中，他们只是单个的多边形，根本不存在碰撞，没有动画，并只是在城市设定的规定路线上进行移动，而且效果仍然可以非常不错。

15.7.3　共享数据结构

这是优化 FSM 计算速度的最基本和最强大的技术之一。FSM(在某个层级上)需要一个环境条件触发状态转换的系统，而且这些条件或许以某种方式被不同的状态共享，因此可以确保一个直接的加速，只要这些不同的条件不被每个状态重复计算，而是在一个状态共享的通用区域内进行计算。在 AIsteroids 演示中是通过状态的 CheckTransitions()函数来直接判断的，同时使其他的计算在 FSMAIControl 结构的 UpdatePerceptions()函数中完成。

有时该功能对于游戏引擎来说是非常基本的，以至于使用了整个的框架范例，即“黑板”。这给任意游戏对象都提供了向一个中央数据区域发布信息的形式上的方式，感兴趣的对象可以请求该信息或被赋予一个检查它们是否相关的事件消息。

15.8　设计上考虑的因素

在决定全力投入到基于状态的系统之前，我们应该考虑第 2 章“AI 引擎的基本组成与设计”中讨论的关于游戏的所有因素，并注意 FSM 能够很好模仿的系统类型。这些因素有：

解决方案的类型、智能体的反应能力、系统的真实性、游戏类型、游戏内容、游戏平台、开发限制和娱乐限制。

15.8.1　解决方案的类型

由于 FSM 具有通用特性，它可以适用于任何类型的解决方案，包括战略性的和战术性的。然而，显然，我们最熟悉的是状态类型的解决方案，因此应该注意到系统需要的解决方案越明确，提供该解决方案的状态也应该要越具体。或者，这意味着需要使用一个层次化的 FSM 来获取更多的特征。一般而言，如果游戏中状态的数量相对较少而且状态本身非常独立和离散，那么 FSM 能够真正显示其力量。一个系统包括 400 个状态，而且这些状态除了微小差别之外几乎相同，那么 FSM 结构将花费相当多的一般管理费用，并没什么好处。

15.8.2　智能体的反应能力

由于 FSM 处理模型具有简单的特性，可以通过调整 FSM 系统来向系统提供任意级别的智能体反应性。事实上，大多数 FSM 系统都运行得足够快，以至于决策稳定性成为建造 FSM 时的一个因素(在 FSM 的劣势一节中与状态震荡一起讨论)。FSM 做出转换决策的时间实际上是瞬间的，真正的代价在于感知计算。然而，这并非人类决策的过程(除了非常简单的 hardwired 行为，如反射动作或本能行为)。人类是深思熟虑的，有一定的反应时间，并受决策时他们所处环境的影响。当 AI 以很快的速度进行决策时，看起来像是机器人式的和抖动的。此类决策抖动可以在两个层级进行处理：状态机本身或感知层级。假设 FSM 进行所有的转换判断，且该判断是感知变化的结果，那么我们可以通过阻止感知抖动来阻止状态机的抖动。可以通过设计一些 2.2 节中“输入处理机与感知”部分或本章关于惯性 FSM 部分讨论的技术来对此进行处理。因此，AI 控制角色的反应性能够在 FSM 系统的许多层级中进行明确控制。

15.8.3　系统的真实性

除非涉及的 FSM 系统非常复杂并且系统期望的被模仿行为相当狭隘，基于 FSM 的决策系统往往是很不真实的。FSM 是静态的，并且除非拥有一个复杂的层次化系统来包含每一个可能的事件，否则它们将以某种方式进行反应，即可能事件的子集通过感知来展现。正是由于这个特性，它们仅能对游戏中提供给它们的状态变化进行响应。人类往往很善于找到 FSM 行为的 AI 模式并对玩家能够很快利用的“缺乏的”感知或状态进行定位。这或许正是游戏所需要的(例如，在射击游戏中对怪物头目进行编码时，该头目可能在战斗期间遵循状态的一个设定模式，并且找到这种模式是玩家战胜该头目的关键)。因此，FSM 行为模型通常用于更多的静态行为设置，或者目标是不变的反应路线的系统。

15.8.4　游戏类型

由于 FSM 缺乏问题相关的语境，故它们可用于每一种游戏类型。它们在那些感知可以在简单条件下计算以及具有唯一条件设置的游戏类型中发展得非常旺盛，因此其输入空间可以被系统划分成便于使用的状态。AIsteroids 这个演示程序事实上对 FSM 来说并不是

非常理想，因为每一波行为(攻击并获取宝物)的游戏玩法大部分都是相似的，并且行为的类型也非常相似(通常是转弯柄向某一目标推进)。然而，FSM可以以一种模块化的方式来构建，从而它们可用于游戏决策结构中的某一给定子集，而不影响AI引擎的其他部分。这意味着如果游戏具有一个特殊的状态导向性非常强的元素，那么就可以仅仅为该部分使用这种类型的范例。这通常是大多数游戏的情形，也是几乎每个游戏中都能够找到某种形式的FSM的原因之一。

15.8.5 游戏内容

游戏内容根据要设计的游戏而有所不同。游戏是否需要遵从状态驱动流程的决策元素？这种额外行为是否可以划分成以转换系统的某种方式连接的具体状态？如果是这样，那么可以使用FSM来对它进行控制。但如果不是，那么可能需要其他类型的控制结构来捕获由具体游戏内容设计的特殊系统的行为。本书的其他技术之一可能合适。

15.8.6 游戏平台

FSM也可以是平台无关的，因为它们没有很大的计算能力或存储区需求。由于这些较低的需求，老式的街机游戏经常使用一些非常依赖FSM的方法。事实上，一些很老的街机游戏为它们的AI对手使用真实的固态逻辑(或敌人运动的模式)，并在电子工程意义上使用FSM。

15.8.7 开发限制

FSM的开发和调试速度非常有利于它们在具有很强开发限制的游戏中使用。特别是在短期项目中，FSM通常不会受过多添加和调试的困扰(它们在长期运行中将会给FSM带来麻烦)。另外，只具有一个AI程序员(或少数几个)的小规模游戏也适合于使用FSM，当然，如果仅仅是一场比赛。有限数量的人记住开发中状态机的变化的结构和连接性要比更大的小组或及其分散的小组容易得多。额外的玩法元素能够比有些系统更容易地添加进FSM中，完全是因为如果能够将一个新状态完全整合到状态图中，那么就能够对系统进行编码以合并该变化。FSM系统能够很容易被新来的程序员所理解，它不像其他更外来的AI系统那样需要新员工一个扩展的学习曲线周期。基于状态模型的系统的质量保证一般也没那么痛苦——行为通常很容易进行复制且行为日志等也容易设计和使用。

15.8.8 娱乐限制

娱乐上的考虑，尤其是难度等级和游戏平衡，很容易由基于状态的系统进行处理。如果游戏玩法的难度等级在游戏过程中会发生改变，那么这种设置本身也需要由一个FSM来控制，该FSM将响应游戏中因为难度等级改变而发生的特殊事件。游戏平衡则更简单，因为系统需要一个状态来对任意给定的感知状态进行响应，实际上是执行一个rock-paper-scissors场景。因此，如果对手以Rock状态向玩家走来，玩家应该转换到Paper状态。很明显，这是假定玩家的FSM模型是工作在反应条件，而不是预测条件，但并没有规定传回到状态机中的感知不能使用预测方法进行计算。

15.9　小结

FSM 是游戏界的传输带。它们简单、强大、易于使用并能够应用到几乎所有的 AI 问题之中。然而，与传输带一样，由此而来的解决方案可以工作，但不够完美，难以扩展和修改，而且如果经常弯曲，那么就可能失败。

- 一个状态机被定义为一系列的状态，以及一个定义了给定条件下状态之间连接性的结构。
- 本书给出的 FSM 框架比大多数都更模块化，因为它将转换类型和转换逻辑装入单个状态之中。每个状态都是模块，因为它包含了它需要与其他状态交互的所有事情。这也允许具有比经典的输入事件方法更复杂的转换判断。
- 本书的 FSM 系统包括 3 个主类：FSMState、FSMMachine 和 FSMAIControl。
- 在 AIsteroids 试验平台上实现的 FSM 只使用 5 个状态(Approach、Attack、Evade、GetPowerup 和 Idle)来达到非常高的性能，或许稍微超越人类。
- 为了获取更高的性能而对试验平台进行的扩展包括状态的添加、处理环绕的更好的数学、弹药管理、更好的攻击和躲避机动。
- FSM 系统的优点在于它们容易射击、实现、扩展、维护和调试。它们也是一种一般问题的解决方法，并可应用到更宽广范围的 AI 问题。
- FSM 系统的弱点是组织上的非正式性、不能进行缩放以及状态震荡问题。
- 层次化 FSM 允许递增的复杂度，同时可以使得整个状态机通过分组保持一定程度的组织性。代码和数据也可以对这些状态进行局部共享，而不用打乱全局 FSM 结构。
- 基于消息的 FSM 非常适合于具有大量状态或零星转换事件的系统。该系统将广播转换信息，而不用单个状态去轮询转换触发器的感知系统。
- 基于堆栈的 FSM 变种使得状态可以被更多的紧迫活动所中断，并且之后通过一个状态堆栈的简单“存储器”进行返回。
- 多个 FSM 能够控制单个 AI 控制的角色的不同方面，并处理角色的决策问题的独立部分，但从组织观点来看仍然可以保持系统的简单性。
- 使用脚本或视频编辑器的数据驱动 FSM 是赋予设计者掌控角色的 AI 决策流程的一种重要方式，并且可提高设计速度以及增强产品的扩展性。
- 负荷平衡算法可以应用到FSM系统及其感知系统中，从而获得更稳定的CPU使用。
- LOD AI 系统可以显著地减少具有许多 AI 控制的角色或可能对人类玩家部分隐藏的世界的游戏中 CPU 的使用。
- 共享数据结构有助于减少 FSM 中不同状态转换逻辑中的重复条件计算。

16 模糊状态机

在上一章中，我们讨论了有限状态机，它涉及到不同状态之间的转换，且系统一次只处于一个当前状态。本章将讨论状态机的一个变种，即模糊状态机(Fuzzy-State Machine，FuSM)。

16.1 FuSM 概述

FuSM 建立在模糊逻辑的概念之上，一般定义为“被扩展来处理部分真值概念的传统(布尔)逻辑的超集”。应该注意，FuSM 建立在该概念之上，但并不代表实际的模糊逻辑系统。

部分真值是一个非常强大的概念。与常规的 FSM 不同，FuSM 在范围上不具有一般性。与 FSM 一样，FuSM 跟踪一系列可能的游戏状态。但不同的是，FSM 具有一个单一的当前状态，然后通过转换到一个不同的状态来响应输入事件，而 FuSM 可能同时具有多个状态，因此不存在转换。模糊系统中的每个状态都计算一个“激活水平(activation level)”，该激活水平决定了系统处于任意给定状态的程度。因此，系统的整体行为由当前被激活状态的贡献的组合来决定。

FuSM 仅仅对那些能够同时处于多个状态并且具有超越简单数字值(如开或关、关闭或打开、生存或死亡等)的系统有用。模糊数值用于描述部分开、几乎关闭和没有完全死亡等。另一种对此类数值类型进行量化的方法是使用一个归一系数(0.0 与 1.0 之间的数)来表示条件对各个端状态的隶属度(例如，0.0 表示完全关闭，1.0 表示完全开启)，尽管对于 FuSM 来说归一化并不是必须的。这是不必记住集合隶属度的所有权限制的一种简单方式，同时也确保了集合隶属度值之间比较的简单性。

关于什么是真正的 FuSM(在游戏 AI 领域)还存在一些混淆的看法，因为在同一类别中存在好几个 FSM 变种都被当作是 FuSM。这些变种(它们将在本章后面进行详细论述)包括：

- **具有转换优先级的 FSM**。在该模型中，必须对每个可用状态的激活水平(该模型仍然是一个 FSM，因此每个状态都有一系列可能的转换)进行计算，然后那个具有最高激活水平的状态获胜并成为新的当前状态。这是许多程序员使用模糊度概念来增强其决策状态机的方式，但真实情况是该系统仍然是一个 FSM，并且类似系统输出行为的可预测性仅仅比常规 FSM 稍微小一点。
- **概率 FSM**。在该形式的 FSM 中，从状态出发的各个转换都被赋予了一定的概率，因此 FSM 遍历更加不确定，从而具有较小的可预测性。这些概率能够随时间发生

变化，或者可以在 FSM 内部进行设置，使用多个 FSM 来对不同的概率集进行分组。概率 FSM 有时用于当某些转换具有许多等价输出状态的情形。例如，靠近一个敌人将使他转换到 3 个状态(具有等效值)之一：Punch、Kick、HeadButt。如果某一给定转换只有一个输出状态，FSM 则正常工作。但如果有多个状态，则对这多个状态进行概率分配(可以是同等选择的平均分配，或侧重于某些状态，或根据分支最近是否被采用或人类阻止使用某一走步等进行更复杂的判断)。

- **马尔可夫模型**。与概率 FSM 很相似，但其转换逻辑是完全基于概率的，因此马尔可夫模型可用于在耦合状态中推断变化。举一个非常简单的例子，假定有两个状态：Aim 和 FireWeapon。在该游戏中，这两个状态通常是完全连接的，因为当瞄准后就会发射。但是，假设要模仿一个更真实的枪炮模型，并有 2%的时间 Aim 状态将转换到 WeaponJam 状态。此类状态转换有时(在其他使用马尔可夫模型的领域中)称为“可靠性建模”。在该示例中，武器具有 98%的可靠性。马尔可夫模型主要用于此类统计建模，因为系统的假设之一就是下一个状态通过概率与当前状态发生联系。因此，马尔可夫模型在诸如风险评估(确定失败率)、赌博(寻找增加利润的方式)和工程学(确定制造中必需的公差，以确保产品的可靠性在可接受的水平)中非常有用。一个反应性的视频游戏可能具有某些属于该类的元素，但由于 AI 对手可能改变状态的主要原因是为了响应人类玩家的愚蠢行动，此类的状态预测很少成为主流。此类系统的一个有趣的用法是去实际模仿人类偶尔表现出来的“意外事件”——AI 对手能够偶尔摔倒、丢球或打到自己的脚。所有这些意外事件都可以在基本的跑步、持球或射门动作层级中处理，并能够在这些活动中的高度耦合动画中通过采用极其不太可能的分支来使其发生。这类真实行为是否符合游戏模拟，或对玩家是否具有娱乐性，完全取决于用户。
- **实际的模糊逻辑系统**。与流行的观念相反，FuSM 并不是真正的模糊逻辑系统。模糊逻辑事实上是一个过程，部分真值表示的规则通过它来进行组合和推理，从而得到决策。之所以设计模糊逻辑是因为许多真实世界问题并非总能表示为限定事件，而且真实世界的解决方案也并非总能表示为限定行动。模糊逻辑仅仅是常规逻辑的一种扩展，使得我们能够处理此类的规则集。游戏中实际模糊逻辑的最简单形式(它非常普通)是描述行为变化的简单“if…else”语句(或其等价物，通过一个数据表格或某种组合矩阵)。例如，语句“如果我的健康值低，而我的敌人的健康值高，那我应该逃跑”就是一个简单的模糊规则。它以一种模糊方式(低、高)对两个感知(我的健康值、我的敌人的健康值)进行比较，并赋予它一个行动(逃跑)。在过去几年中，该语句作为一个 if 语句在大量游戏中使用。这是一个实际模糊系统最简单、最小的形式。一个真正的模糊逻辑系统将包括许多一般的模糊指导方针，它们可处理“我的健康值”、“我的敌人的健康值”以及所有其他规则需要考虑的变量等的任意组合，其中这些规则将通过算法组合来提供一个响应动作。这往往是从模糊系统中获取结果的一种强大方式，但当存在许多模糊变量时会出现问题，这是因为会产生一个很快将难以管理的必要规则集尺寸，即“组合爆炸”问题。这可以通过采用一种叫做 Comb's Method 的统计方法进行处理，它能够简化所需要的规则集合，但同时也降低了精度。

FuSM(以及以前提到的相似变种)很快成为了游戏 AI 使用中的一种普遍方法。FSM 的可预测性正越来越不能满足人们的需要，并且许多游戏的整体内容正变得足够丰富从而确保了 FuSM 额外设计和实现复杂性的必要性。

FuSM 需要具有比 FSM 更多的前向思考。游戏问题必须真正地分解为问题允许的极其独立的元素。FSM 能够在 FuSM 系统范围内进行设计，通过计算数字激活水平并设计系统使得在状态执行中不存在交叠。有人在建立模糊系统时偶然(或通过无知)做到了这些。对许多问题情形来说以一种限定方式进行思考是很自然的事情，因此如果在游戏中很难想出一种方法来实现 FuSM，那很可能是因为不应该一开始就使用模糊方法。FuSM 不如 FSM 那样能够适用于一般范围问题。FuSM 是一类允许多个状态激活为当前状态并能够具有与游戏局势有利于每个状态的程度相当的一定激活水平的 FSM。

事实上，许多人都认为 FuSM 根本就不是真正的状态机(因为系统并不处于孤立状态)，而更像是模糊知识库，其中有多个断言能够同时部分为真。但是，通过对独立状态进行编码以利用这些断言，我们能够使用 FuSM 来实现需要此类机制的 AI 目标。

对 Robotron 游戏中 AI 控制敌人的决策系统进行编码是使用此类系统的一个简单例子。图 16-1 是一个简单 Robotron 玩家的 FSM 状态图。它有 3 个主状态(该游戏非常类似于 Asteroids 游戏，因此看起来很熟悉)：Approach、Evade 和 Attack。在一个严格的基于 FSM 的系统中，为了能够同时移动和射击，必须进行编码以使得 Approach 和 Evade 状态以特定方向开始运动，但当状态改变时也不停止运动。因此，当处于 Attack 状态时，玩家仍然能够从他上一次的运动状态中进行移动。这样做能够奏效，但不够简洁。Attack 状态必须要保持对向其他状态的转换进行检测，因此玩家在朝另一个方向射击时不会撞上敌人，也不会陷入远离所有敌人的角落。更好的方式是为游戏设计一个 FuSM。这样，玩家能够同时 Approach、Evade 和 Attack。

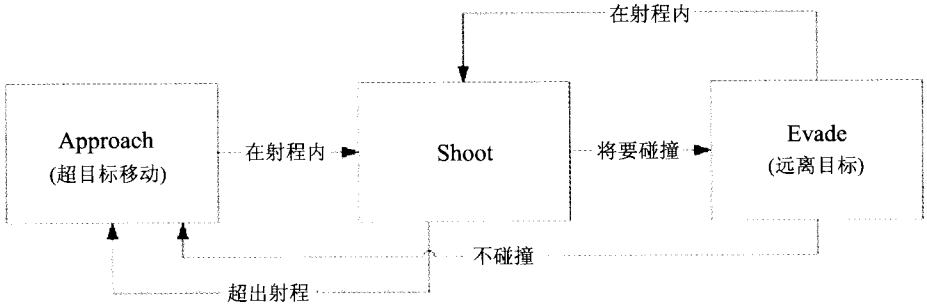


图 16-1　Robotron 玩家的 FSM 图

与 FSM 一样，FuSM 也可以以一种自由形式的方式进行编写。可以按照程序清单 16-1 那样更好地实现 Robotron 的 FSM 行为。从中可知，Robotron 玩家的 Update()函数使用 3 个不同的函数，且各函数如果满足一个条件便可以进行更新——如果读者以前编写过游戏就会见过类似的代码，因为它非常普遍。玩家类包括了处理行为不同方面的独立方法和确定是否使用各种方法的判断函数。它在这样相对简单的例子中可以有很好的表现，但在一个复杂系统(考虑在即时策略(RTS)游戏中，具有一个运行决策引擎的 FuSM，它将根据需要更新的每个独立决策系统的激活水平对它所具有的用于计算的时间进行划分)中，需要将

这种逻辑分解成不同的模块，使系统更具组织性和可读性，并能够由多个程序员同时进行处理。

程序清单 16-1　自由形式的 Robotron 游戏 FuSM 玩家的更新代码

```
void RobotronPlayer::Update(float dt)
{
    float urgency;
    if(CalculateApproachUrgency(urgency))
        Approach(dt,urgency);
    if(CalculateEvadeUrgency(urgency))
        Evade(dt, urgency);
    if(CalculateAttackUrgency(urgency))
        Attack(dt, urgency);
}
```

关于 Robotron 示例需要注意的另一件事是 Attack 状态不能完全模糊化。玩家要么射击，要么不射击，因为我们不可能部分启动激光器。其他状态则不同，比如运动就可以表示成在不移动和全速移动之间的一个平滑梯度。然而，Attack 状态的不可模糊化并不影响系统的其余部分，而且并不会使该方法失效。FuSM 能够很容易地通过以数字方式计算激活水平来加入更多的数字状态，系统将像其他状态一样响应该数字状态。

16.2　FuSM 骨架代码

与 FSM 一样，FuSM 的代码也将由 3 个主类来实现：

- **FuSMState 类**。它是基本的模糊状态。
- **FuSMMachine 类**。它是模糊状态机。
- **FuSMAIControl 类**。它是 AIControl 类，负责机器的工作，并存储游戏相关的信息和代码。

下面将对这些类进行充分讨论。

16.2.1　FuSMState 类

在大多数纯粹的设计层级上，FuSM 系统中的状态都是分离系统。每个状态都将使用感知变量(来自 Control 类，或更复杂和专用的感知系统)来确定激活水平(在本书中将以一个 0 到 1 之间的数字表示)，该激活水平用于测量所需状态怎样完全主动地对感知进行响应。在最简单的方式下，激活水平能够响应游戏中某个值的数量，如进攻性；激活水平为 0.0 意味着根本不具有进攻性，而 1.0 则表示完全沉浸于进攻在中。FuSM 状态的最小需求与 FSM 状态非常相似，它们有：

- **Enter()**。只要进入该状态，该函数就会始终运行。它使得状态可以执行数据或变量的初始化。
- **Exit()**。该函数在离开状态时运行，并主要作为一项清除任务，或用于需要运行的(或开始运行)任意希望在特定转换处发生的额外代码的地方(针对 Mealy 式状态机)。

- Update()。如果该状态是 FSM(针对 Moore 式状态机)中的当前状态，那么这是调用 AI 每个处理循环的主函数。
- Init()。该函数对状态进行初始化。
- CalculateActivation()。该函数用于确定状态的模糊激活水平。它返回一个数值，并将其作为 m_activationLevel 数据成员存储在状态之中。在本章后面将会看到，通过返回布尔值而非正常的归一值，我们可以模仿更多的数字状态(如试验平台上的攻击状态)。

程序清单 16-2 给出了该类的 header。再次，我们尽量全面地设计这个类，从而在游戏实现时能够具有最大的灵活性。从中可知，除了 m_activationLevel 数据成员外，它与 FSM 类非常相似。事实上，该数据成员可以并入 FSM 类中，并交替使用两种类型的状态来开发一个混合系统。

程序清单 16-2　FuSMState 类的 Header

```
class FuSMState
{
public:
   //constructor/functions
   FuSMState(int type = FUSM_STATE_NONE,
             Control* parent = NULL)
      {m_type = type;m_parent = parent;
       m_activationLevel = 0.0f;}
   virtual void Update(float dt){}
   virtual void Enter()          {}
   virtual void Exit()           {}
   virtual void Init()           {m_activationLevel = 0.0f;}
   virtual float CalculateActivation()
             {return m_activationLevel;}

   virtual CheckLowerBound(float lbound = 0.0f)
             {if(m_activationLevel < lbound)
              m_activationLevel = lbound;}
   virtual CheckUpperBound(float ubound = 1.0f)
             {if(m_activationLevel > ubound)
              m_activationLevel = ubound;}
   virtual CheckBounds(float lb = 0.0f,float ub = 1.0f)
             {CheckLowerBound(lb);CheckUpperBound(ub);}


   //data
   Control* m_parent;
   int      m_type;
   float    m_activationLevel;
};
```

该类具有 3 个边界检测函数，它们仅仅是对激活水平进行上下限检测。可以从状态中调用任意一个，或者如果希望具有一个完全粗略的激活水平，则不需要进行任何调用。

与常规FSM一样，该类也包括两个数据成员：m_type和m_parent。整个状态机和状态间的代码都可以使用类型域，以基于正在考虑的特殊状态来进行判断。这些数值的枚举存储在FSM.h文件中并且通常是空的，只包含默认的FSM_STATE_NONE值。当实际使用某事件的代码时，需要向该枚举中添加所有的状态类型，并从中开始运行。单个的状态使用父域，因此它们能够通过其Control结构访问一个共享数据区域。

16.2.2 FuSMMachine类

与相应的FSMMachine类一样，该类(其header见程序清单16-3)包含了机器需要跟踪的所有状态。为了查询，它也包含了一系列当前的活动状态。像FSMMachine类一样，模糊机是FuSMState类的一个子类，因此可以通过使一个特殊的模糊状态成为整个FuSM来构造一个层次化的FuSM。

程序清单16-3 FuSMMachine类的Header

```
class FuSMMachine: public FuSMState
{
public:
     //constructor/functions
     FuSMMachine(int type = FUSM_MACH_NONE,Control* parent = NULL);
     virtual void UpdateMachine(float dt);
     virtual void AddState(FuSMState* state);
     virtual bool IsActive(FuSMState* state);
     virtual void Reset();

     //data
     int m_type;
protected:
     std::vector<FuSMState*> m_states;
     std::vector<FuSMState*> m_activatedStates;
};
```

程序清单16-4给出的UpdateMachine()函数用于运行一般的模糊机。从中可知，系统非常简单：运行每个状态的CalculateActivation()函数，分离出活动状态，通过Exit()函数将所有非活动状态作为一个组退出，然后为所有活动状态调用Update()函数。尽管简单地为每个状态轮流调用退出或更新函数而不将状态存储为单独的矢量看起来非常吸引人，但这样做具有限制性。之所以需要按这种方式进行处理是因为某些非活动状态的Exit()函数可能重置某些活动状态已经开启的事件，或在更新过程中需要做出变化。

程序清单16-4 FuSMMachine::UpdateMachine()函数

```
void FuSMMachine::UpdateMachine(float dt)
{
   //don't do anything if you have no states
   if(m_states.size() == 0)
      return;
   //check for activations, and then update
   m_activatedStates.clear();
```

```
    std::vector<FuSMState*> nonActiveStates;
    for(int i =0;i<m_states.size();i++)
    {
         if(m_states[i]->CalculateActivation() > 0)
            m_activatedStates.push_back(m_states[i]);
         else
            nonActiveStates.push_back(m_states[i]);
    }

    //Exit all non active states for cleanup
    if(nonActiveStates.size() != 0)
    {
         for(int i =0;i<nonActiveStates.size();i++)
            nonActiveStates[i]->Exit();
    }

    //Update all activated states
    if(m_activatedStates.size() != 0)
    {
         for(int i =0;i<m_activatedStates.size();i++)
            m_activatedStates[i]->Update(dt);
    }
}
```

16.2.3 FuSMAIControl 类

最后，程序清单 16-5 给出了 FuSM 系统的控制类。它几乎与 FSM 的控制类一样，包含了运行系统所必需的全局数据成员和一个指向模糊机结构的指针。在更形式化的游戏中，一般具有许多全局数据成员或复杂的感知更新计算，其更好的方式是设计一个专门的感知系统(由控制类进行控制)，但通过 UpdatePerceptions()方法直接更新这个小的列表对于我们的测试应用来说已经很好了。

程序清单 16-5　FuSMAIControl 类的 Header

```
class FuSMAIControl: public AIControl
{
public:
     //constructor/functions
     FuSMAIControl(Ship* ship = NULL);
     void Update(float dt);
     void UpdatePerceptions(float dt);
     void Init();

     //perception data
     //(public so that states can share it)
     GameObj*   m_nearestAsteroid;
     GameObj*   m_nearestPowerup;
     float      m_nearestAsteroidDist;
     float      m_nearestPowerupDist;
```

```
        bool        m_willCollide;
        bool        m_powerupNear;
        float       m_safetyRadius;

    private:
        //data
        FuSMMachine* m_machine;
    };
```

16.3　在试验平台上实现 FuSM 控制的飞船

运行我们的 AIsteroids 主飞船所必需的 AI 系统并不适合于模糊系统，因为在执行一些状态的同时也必须要执行其他状态(必须要执行转弯来进行射击，但也要执行不同的转弯来进行推进)。因此，在试验平台示例中，设想我们拥有第二种类型的飞船，即 Saucer，它与我们的主飞船有很大的不同。Saucer 不需要转弯来进行推进。它飞行时没有重量，因此不会受惯性和低加速度的困扰。它可以在它希望的任何方向进行推进，并且具有缓冲器以保证飞行员的安全。由于具有这种令人吃惊的能力，它也装备了一种能够往任意方向发射的炮塔。另外，它也具有一个能够用来将物体拖向它自身的牵引光束(tractor beam)。

此类航行器具有独立的系统并且可以相对自由地对其决策的不同部分进行连接(运动几乎完全独立于攻击，并且抓取物体也进行了解耦)，因此对它进行管理是 FuSM 系统的主要内容。当给定几个基本的感知时，每个系统(枪炮、发动机、牵引光束)都能进行独立操作，也能同时进行操作。因此，我们的飞船不再使用状态系统，因为从一个状态到另一个状态都是发展的；相反，我们将在下述事实下进行操作，即每个独立活动都将控制其对飞船的整体行为是否有贡献。

16.4　示例实现

在下面各节中，将会介绍并详细描述实现 Saucer 所必需的类以及控制其行为的 FuSM。

16.4.1　添加 Saucer

Saucer 是游戏要实现的新飞船类型(参见程序清单 16-6 所示的 header)。从中可知，它是非常类似的，尽管 GetClosestGunAngle()方法指示返回通过的角度(passed in angle)，因为炮塔可以朝任意方向开火。

程序清单 16-6　Saucer 的 Header

```
class Saucer : public Ship
{
public:
    //constructor/functions
    Saucer(int size = 7);
```

```
        void Draw();
        void Init();

        //bullet management
        virtual void Shoot();
        virtual float GetClosestGunAngle(float angle)
        {return angle;}
};
```

16.4.2　其他的游戏修改

为了让 saucer 能够工作，需要引入几个其他的系统。基地飞船类将给出控制信号以处理牵引光束和 AG 推进器(antigravity，或非惯性驱动)。它也被赋予一个 AG 驱动方向的矢量 m_agNorm。该矢量可以通过两种不同的方式进行赋值：通过 AGThrust(vector)来开启该驱动，并设置方向为通过矢量的规格化值；或使用 AGThrustAccumulate(vector)来开启驱动，但之后将该矢量添加到 m_agNorm 变量中。然后像飞船对运动的更新方法所使用的那样对它进行规格化。这是系统模糊性的一个重要部分。每个需要运动的状态都使用 AGThrustAccumulate()方法来请求飞船运动，并通过将它与其激活水平相乘来缩放它将传递的矢量。这样做之后，一个具有很高激活水平的状态将比具有低激活水平的状态对飞船的运动方向有更多的贡献。之后，基类飞船的 Update 函数检测 AG 驱动是否已经开启，如果开启，则对飞船位置应用 m_agNorm 矢量，从而给它一个瞬时加速度和忽略惯性的能力。

对代码的另一个添加是新的 GameSession::ApplyForce()函数。该函数需要进行两次重载，第一次是作为一个对象类型的参数，即一个力矢量和使用该力的增量时间。它将贯穿游戏对象列表并将力添加进所有传递类型的对象中。第二次是 ApplyForce()方法，它采用一个对象类型、一条力的线路、力矢量和一个使用该力的增量时间。当它开始检测对象在应用该力之前是否与这个力的路线冲突时，我们将使用该方法来模拟牵引光束。

16.4.3　FuSM 系统

图 16-2 是 FuSM 的框图。与行星游戏的 FSM 设计不同，这里只有 4 个而不是 5 个状态。FSM 系统本质上是闭环的，且必须始终具有一个当前状态。在 FSM 设计中，Idle 状态是系统中所有其他状态的主要分支点，并作为上一次处理的状态。但是，FuSM 可以运行任意数量的状态(包括零个)，因此该状态并不是模糊系统所必需的。从图 16-2 可知，基本的状态包括如下几个：

- Approach。它将使飞船进入最近行星的射程。
- Attack。对 Saucer 来说，它仅仅是在最近行星方向上启动枪炮。飞船将向前发动武器并需要转弯且面向目标，但 Saucer 具有一个炮塔。
- Evade。它将通过监控飞船的速度来启动避免一个碰撞路线上的行星。
- GetPowerup。它将设法捡起所在范围内的宝物。然而，与飞船不同，Saucer 具有一个牵引光束，它能够用于抓取宝物。

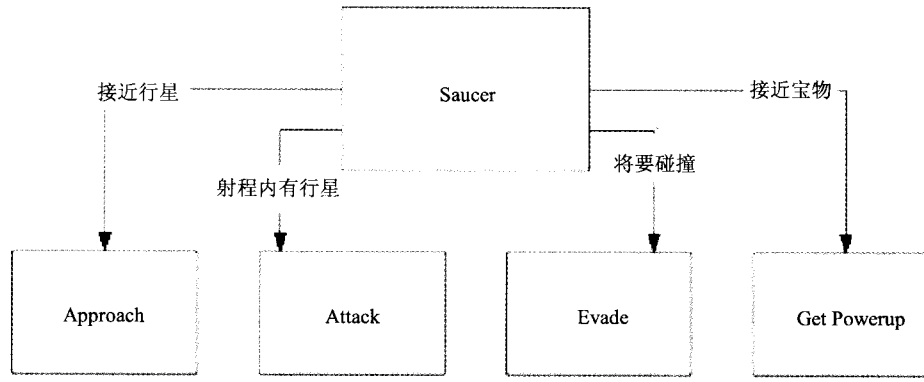


图 16-2　Asteroids 游戏的 FuSM 框图

FuSM 需要一些数据来计算每个状态的激活水平。这些数据有：

- 与最近行星的距离用于确定是否激活下列 3 个状态：Approach、Evade 和 Attack。行星越近，航行器将进行更多的躲藏和攻击；行星越远，激活的可处理行为也越多。
- 与最近宝物的距离。这影响到 GetPowerup 状态的激活。Saucer 离宝物越近，它越将设法得到宝物。

关于该系统有一些需要注意的事项。每个模糊状态都不存在关于其他状态的信息。每个状态只关心直接处理它本身的感知检测。在 FSM 设计中，几乎每个状态都需要监视 m_willCollide 字段，一旦它为真，则转换到躲避状态。减少在限定系统中存在的冗余状态的转换检测非常重要。由于 FSM 中的所有状态均具有一个优先次序，故在我们的行星 FSM 示例中的许多状态都是相互连接的。如果发现 FSM 正使用一个几乎完全连接的状态图，那么系统可能非常适合 FuSM。这并非是所有的情况，但如果游戏能够在任意状态之间进行移动，则很可能不存在太多的先决条件，且系统表现出来的是线性行为。

16.5　控制类编码

FuSM 模型的控制器类(程序清单 16-7 是它的 header，程序清单 16-8 是其中重要函数的实现)包括状态机结构和该 AI 模型的全局数据成员。

程序清单 16-7　FuSMAIControl 类的 Header

```
class FuSMAIControl: public AIControl
{
public:
     //constructor/functions
     FuSMAIControl(Ship* ship = NULL);
     void Update(float dt);
     void UpdatePerceptions(float dt);
     void Init();

     //perception data
```

```
    //(public so that states can share it)
    GameObj* m_nearestAsteroid;
    GameObj* m_nearestPowerup;
    float m_nearestAsteroidDist;
    float m_nearestPowerupDist;
    bool m_willCollide;
    float m_safetyRadius;

private:
    //data
    FuSMMachine* m_machine;
};
```

从感知的观点来看，模糊控制类是非常简单的。然而，这归功于打破了行星的规则(比如 Saucer 不具有惯性，且拥有炮塔和牵引光束)，而不是因为我们使用了 FuSM。在数学方式上它只是更容易使 Saucer 移动到或避免某个特殊位置，因为它不需要太担心它自身的速度。

FSM 的 AI 数据成员 m_powerupNear 不再是必需的，它更多的是一个 FSM 能够响应的事件触发器，但模糊系统使用与宝物的距离来直接与 GetPowerup 状态的激活水平相联系。

Update()方法与 FSM 设计中的完全一致。如果不存在要控制的飞船，则不运行该控制器。它只是对感知和模糊机器本身进行更新。

程序清单 16-8　FuSMAIControl 类中重要函数的实现

```
FuSMAIControl::FuSMAIControl(Ship* ship):
AIControl(ship)
{
    //construct the state machine and add the necessary states
    m_machine = new FuSMMachine(FUSM_MACH_SAUCER,this);
    m_machine->AddState(new FStateApproach(this));
    m_machine->AddState(new FStateAttack(this));
    m_machine->AddState(new FStateEvade(this));
    m_machine->AddState(new FStateGetPowerup(this));
}

//--------------------
void FuSMAIControl::Update(float dt)
{
    if(!m_ship)
    {
        m_machine->Reset();
        return;
    }

    UpdatePerceptions(dt);
    m_machine->UpdateMachine(dt);
}
```

```
//--------------------
void FuSMAIControl::UpdatePerceptions(float dt)
{
     if(m_willCollide)
        m_safetyRadius = 30.0f;
     else
        m_safetyRadius = 15.0f;

     //store closest asteroid and powerup
     m_nearestAsteroid = NULL;
     m_nearestPowerup = NULL;
     m_nearestAsteroid = Game.GetClosestGameObj(m_ship,
                                GameObj::OBJ_ASTEROID);
     if(m_ship->GetShotLevel() < MAX_SHOT_LEVEL)
        m_nearestPowerup = Game.GetClosestGameObj(m_ship,
                                GameObj::OBJ_POWERUP);

     //asteroid collision determination
     m_willCollide = false;
     if(m_nearestAsteroid)
     {
        m_nearestAsteroidDist = m_nearestAsteroid->
                  m_position.Distance(m_ship->m_position);
        float adjSafetyRadius = m_safetyRadius +
                                m_nearestAsteroid->m_size;

        //if you're too close,
        //flag a collision
        if(m_nearestAsteroidDist <= adjSafetyRadius )
           m_willCollide = true;
     }

     //powerup near determination
     if(m_nearestPowerup)
        m_nearestPowerupDist = m_nearestPowerup->
                  m_position.Distance(m_ship->m_position);
}
```

模糊状态编码

在下面各节中将分别讨论 4 个状态的实现(程序清单 16-9~程序清单 16-12)。

1. FStateApproach

该状态仅仅是计算到最近行星的矢量，并用它来作为 Saucer 无重力驱动的一个推进矢量。这里没有什么特殊的地方，无重力驱动只是像前面所讨论的那样通过直接影响位置而不是加速度来工作。

如果附近不存在行星，CalculateActivation()方法则返回一个零；否则它返回一个介于0.0f(当与行星的距离几乎为零时)和 1.0f(当距离等于或大于 FU_APPROACH_DIST 时)的规格化数值。CheckBounds()函数调用确保了活动值落在这个范围之内。

最后，Exit()函数停止该 AG 驱动，因为这是该状态处理的唯一模式。

程序清单 16-9　FstateApproach 状态的实现

```
//--------------------
void FStateApproach::Update(float dt)
{
   //turn and then thrust towards closest asteroid
   FuSMAIControl* parent = (FuSMAIControl*)m_parent;
   GameObj* asteroid = parent->m_nearestAsteroid;
   Ship* ship = parent->m_ship;
   Point3f deltaPos = asteroid->m_position -
                      ship->m_position;

   //move there
   ship->AGThrustAccumulate(deltaPos*m_activationLevel);

   parent->m_target->m_position = asteroid->m_position;
   parent->m_debugTxt = "Approach";
}

//--------------------
float FStateApproach::CalculateActivation()
{
   FuSMAIControl* parent = (FuSMAIControl*)m_parent;
   if(!parent->m_nearestAsteroid)
      m_activationLevel = 0.0f;
   else
      m_activationLevel = (parent->m_nearestAsteroidDist -
        parent->m_nearestAsteroid->m_size)/FU_APPROACH_DIST;
   CheckBounds();
   return m_activationLevel;
}

//--------------------
void FStateApproach::Exit()
{
   if(((FuSMAIControl*)m_parent)->m_ship)
      ((FuSMAIControl*)m_parent)->m_ship->StopAGThrust();
}
```

2. FstateAttack

该状态也比 FSM 中的要稍微简单一些。再次强调，Saucer 不需要像常规飞船那样转弯，因此它所需要做的所有工作就是计算前向角并开火。

该状态的激活函数是数字的，即 0 或 1，因为我们不能朝某物进行部分射击。当存在一个行星而且它处于射击范围之内时，该状态开启，否则它关闭。

该状态没有 Exit()方法，因为射击命令并不是一个开/关的套环命令。每次它只能射击一次。

程序清单 16-10　FstateAttack 状态的实现

```
//--------------------
void FStateAttack::Update(float dt)
{
    //turn towards closest asteroid's future position, and then fire
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    GameObj* asteroid = parent->m_nearestAsteroid;
    Ship* ship = parent->m_ship;

    Point3f futureAstPosition = asteroid->m_position;
    Point3f deltaPos = futureAstPosition - ship->m_position;
    float dist = deltaPos.Norm();
    float time = dist/BULLET_SPEED;
    futureAstPosition += time*asteroid->m_velocity;
    Point3f deltaFPos = futureAstPosition - ship->m_position;

    float newDir = CALCDIR(deltaFPos);
    ship->Shoot(newDir);

    parent->m_target->m_position = futureAstPosition;
    parent->m_debugTxt = "Attack";
}

//--------------------
float FStateAttack::CalculateActivation()
{
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    if(!parent->m_nearestAsteroid)
        m_activationLevel = 0.0f;
    else
        m_activationLevel = parent->m_nearestAsteroid &&
            parent->m_nearestAsteroidDist < FU_APPROACH_DIST;
    return m_activationLevel;
}
```

3. FstateEvade

该状态与其他运动状态完全一致。它计算一个距离最近行星的矢量并设置 AG 驱动以朝所在方向推进。

当逐步靠近最近的行星时，该状态的激活水平也逐步提高，从而模仿当接近碰撞时将变得更加专注于逃避这个事实。

与使用无重力系统的其他状态一样，当退出时它关闭 AG 发动机。

程序清单 16-11　FstateEvade 状态的实现

```
//--------------------
void FStateEvade::Update(float dt)
{
    //evade by going away from the closest asteroid
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    GameObj* asteroid = parent->m_nearestAsteroid;
    Ship* ship = parent->m_ship;
    Point3f vecBrake = ship->m_position - asteroid->m_position;

    ship->AGThrustAccumulate(vecBrake*m_activationLevel);
    parent->m_target->m_position = parent->
                        m_nearestAsteroid->m_position;
    parent->m_debugTxt = "Evade";
}

//--------------------
float FStateEvade::CalculateActivation()
{
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    if(!parent->m_nearestAsteroid)
        m_activationLevel = 0.0f;
    else
        m_activationLevel = 1.0f - (parent->
                            m_nearestAsteroidDist - parent->
                            m_nearestAsteroid->m_size)/
                            parent->m_safetyRadius;
    CheckBounds();
    return m_activationLevel;
}

//--------------------
void FStateEvade::Exit()
{
    if(((FuSMAIControl*)m_parent)->m_ship)
        ((FuSMAIControl*)m_parent)->m_ship->StopAGThrust();
}
```

4. FstateGetPowerup

与正常飞船不一样，saucer 装备了一个强大的牵引光束，当激活时能够向其自身拖拉宝物。它还可以接近宝物，并且接近的紧急程度将由该状态的激活水平来控制。该状态也将开启牵引光束来拖拽宝物。

激活计算方法与 FStateEvade 状态的非常相似，都是越靠近宝物，激活越强。因此，如果宝物在附近，Saucer 将尽最大努力把它捡起(通过它的机动)；否则，牵引光束将会完成大部分的工作。

由于该状态使用了牵引光束和 AG 发动机，故 Exit()方法需要关闭它们两个。

程序清单 16-12　FstateGetPowerup 状态的实现

```
//--------------------
void FStateGetPowerup::Update(float dt)
{
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    GameObj* powerup = parent->m_nearestPowerup;
    Ship* ship = parent->m_ship;

    Point3f deltaPos = powerup->m_position -
                       ship->m_position;

    ship->AGThrustAccumulate(deltaPos*m_activationLevel);
    ship->TractorBeamOn(-deltaPos);

    parent->m_target->m_position = powerup->m_position;
    parent->m_debugTxt = "GetPowerup";
}

//--------------------
float FStateGetPowerup::CalculateActivation()
{
    FuSMAIControl* parent = (FuSMAIControl*)m_parent;
    if(!parent->m_nearestPowerup)
        m_activationLevel = 0.0f;
    else
        m_activationLevel = 1.0f - (parent->
                    m_nearestPowerupDist - parent->
                    m_nearestPowerup->m_size)/
                    FU_POWERUP_SCAN_DIST;
    CheckBounds();
    return m_activationLevel;
}

//--------------------
void FStateGetPowerup::Exit()
{
    if(((FuSMAIControl*)m_parent)->m_ship)
    {
        ((FuSMAIControl*)m_parent)->m_ship->StopAGThrust();
        ((FuSMAIControl*)m_parent)->
                                m_ship->StopTractorBeam();
    }
}
```

16.6　使用该系统的 AI 的性能

由于适当放置了 FuSM 系统，并且 Saucer 必须遵守更加不严格的玩法规则，在试验平台游戏中几乎总能摧毁行星。只要玩家允许，它就会进行，并且在测试中它可以在连续数小时的游戏中保持生存。图 16-3 给出了工作的 Saucer。它仍然偶尔会死亡，并能够通过有助于 FSM 系统的同类改进来实现战无不胜。

- 增加数学模型的复杂性以使 AI 系统具有处理坐标缠绕的游戏世界的能力。现在，AI 的主要缺陷是当游戏世界中事件缠绕时它便失去了目标，因此在瞄准和避障中解决该问题将极大增强 AI 飞船的生存能力。甚至通过 Saucer 超越常规飞船的性能，这个缺陷也有所减弱，因为它不会像飞船那样漂浮在边界上。
- 飞船的弹药管理。现在，它只是瞄准，然后开始射击。枪炮不存在点火率，因此往往是朝目标发射一群子弹。这是相当有利的。当它向一个大行星发射一群炮弹时，残余的炮弹有时可以杀死行星分裂时的碎片。但当它发射完其所有配置的弹药时，飞船便会陷入困境，且在它能再次发射之前必须等待它们碰撞或过期，从而使其暂时失去防御能力。

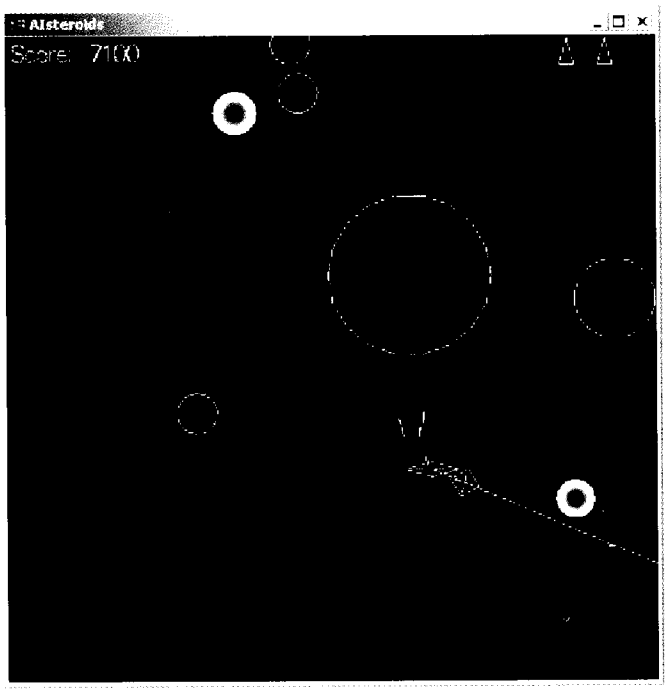


图 16-3　AIsteriods 试验平台上的 FuSM 实现

16.6.1　基于 FuSM 系统的优势

对于适当的问题来说 FuSM 很容易设计。如果 AI 情形涉及独立、并发的系统，那么该模型将有助于设计相互之间毫无关系的分离系统。因此，我们不需要对转换事件和 FSM 系统需要的状态间的链接进行设计。该模型根据用户为特定问题定义的一个缩放比例来激

活每个状态。通过简单地让激活计算函数返回数字值，FuSM 也可以将数字激活状态与更模糊的状态进行自由混合。

由于不存在转换，故 FuSM 系统的设计比 FSM 设计还简单。每个状态可以进行完全隔离设计，而仅仅由全局感知数据(存储于控制类中)来将系统粘合在一起。

对模糊系统进行扩展就像寻找能与系统自由结合的其他状态一样简单。在我们的行星示例中，可以添加另一个状态来以排斥光束(repulsive beam，牵引光束的对立物)的方式来帮助躲避。这可以从飞船中射出并使即将到来的行星偏转。几乎不需要费力就可以向 FuSM 中添加一个控制排斥光束使用的状态；可以通过复制 GetPowerup 状态并改变少数几行来影响最近的行星而不是宝物，并改变应用到礁石上的力的方向。

对模糊系统进行调试也非常简单。由于状态的非耦合特性，可以使任何暂时不需要关注的状态失效，然后集中精力于剩下的活动状态。通过禁能攻击状态，saucer 的躲避代码达到了最小限度。Saucer 将设法躲避礁石，但由于每次只考虑一个行星，它将总是被包围并被摧毁。为了扩展航行器的能力，需要设计并试验先进的躲避技术(可能包括适度的路径搜索或某种形式的影响力地图分析)，而不需要担心摧毁每件事物并为 Saucer 清除道路的有效攻击行为。

由于状态的非连接特性，FuSM 能够很好地进行缩放。需要处理的唯一问题是过度混杂的概念，它或许将导致整体上的非常普通或混乱的行为。假定我们的试验平台不仅具有当前的接近、躲避以及与飞船运动竞争的获取宝物行为，还具有设法以浮动基地入坞、使用某类运输门的机动、对其他友好 Saucer 的编队请求进行响应等的状态，甚至可能对蛀洞等紧急事件进行响应。最终，有许多的状态都可以影响 AG 驱动的推进方向，以至于飞船根本就不能移动。加入系统某一具体特性的状态越多，每个单独状态的贡献就变得越弱。这种变弱可以通过设法将状态按相似性进行分组来克服(例如，先前例子中运输门处理状态可被认为是一个不同种类的宝物，蛀洞处理状态则可划分到躲避那一组)。

模糊系统可以使得 AI 控制的智能体呈现出更大范围的行为个性。通过减少对 FU_APPROACH_DIST 的详细说明，当前的 FuSM 行星设计可以变得更具“攻击性”。通过提高躲避行为的优先级并提高宝物状态的整体激活水平，我们可以得到一个更具防御性的角色，当宝物出现时它甚至将变得贪婪。使用不同状态的重定义的 CalculateActivation() 方法可以编写不同的 Saucer，或者不同的 Saucer 可以使用一个数据驱动的、可访问一系列属性的界面来调整整体的混合行为，使它们展现出特殊的个性特征。

FSM 中的状态震荡问题在 FuSM 中不会出现。FuSM 实际上能够同时处于每一个状态，或不处于任何状态，因此不存在在状态之间来回转换的概念。然而，该问题被行为震荡的概念所取代，这将在下一节进行讨论。

16.6.2 基于 FuSM 系统的劣势

FuSM 并不是像 FSM 一样的通用问题求解器。FSM 模仿的是顺次接连发生的行为，它们代表一个考虑了反应性、主动任务分配和先决条件行动的循环发展系统。FuSM 更适

合于通过将小的无连接的行为进行混合而构造的复杂行为系统。这个混合概念是关键。FuSM 能够独特地处理行为梯度。这并不是所有游戏都需要的概念，因为在快速运动、低图像解析度、确定动画和技术资产的游戏世界中，细微的行为差别通常是不为人知的。将来，当面部动画和基于物理学的运动系统(它对运动的模仿是基于作用在人身上的力，而不是角色播放的手制或运动捕获动画)成为规范时，FuSM 将成为将全范围的情感和环境带入 AI 控制角色的必不可少的一部分。对于现在而言，纯粹的 FuSM 系统是对某些特定行为非常有用的合适的技术。

正如上一节所述，设计得不好的 FuSM 将会产生行为震荡。在我们的行星 Saucer 中，我们不需要对此担心，因为唯一可能相互对抗的状态是精确相反的，即 Approach 状态和 Evade 状态。然而，如果两个状态都取最大值，它们会相互抵消，从而飞船将保持不动。但如果在使用中 Approach 和 Evade 不采用相反的矢量，并且如果 Approach 设法变得比 Evade 能够允许的值更近，那么飞船将有非常古怪的行为：它可能盘旋移动或以某种循环对角 Z 字形移动。解决这个问题的方式正是采用 Saucer 的方式：模仿人体使用肌肉的行为，用辅助但相反的状态(它们完成工作并共同工作)来减少激活中的矛盾。

16.7　范例扩展

正如本章开头讨论的那样，FuSM 很容易造成误解。人们使用的所谓 FuSM 的不同方法有许多种。下面对这些扩展和变种中的一些较有用的方法进行介绍。

16.7.1　有限数量当前状态的 FuSM

人们或许需要一个具有一系列平滑激活梯度行为的系统，但只有一个或少数几个行为能够进行更新。FuSM 很容易扩展到将各状态的激活水平作为一个优先级函数，并且赢家(或具有最高优先级的状态)将成为唯一要更新的状态。由于是一个单个状态，该系统很像是第 15 章“有限状态机”中讨论的具有模糊转换的 FSM 变种。如果仍然允许存在多个当前状态，则可以考虑把该方法作为克服前面章节中讨论的弱化问题的一种方法。特殊的模糊状态可以有子类型，并且最高优先级子类型将会为该特殊子类型种类而成为赢家。在 AIsteroids 示例中，攻击就是一个子类型，还有运动和牵引光束。因此，Approach 和 Evade 将竞争以成为到达函数的唯一运动状态。这有助于弱化，但也对系统进行反模糊化，因为是在将额外混合元素放到整体行为中。也可以使用此类系统来作为关心计算成本的游戏的一种节约计算成本的优化方法。

16.7.2　作为角色支持系统的 FuSM

尽管完全模糊控制的角色相当少(看一看在最初的 AIsteroids 示例中我们为了让它适合使用 FuSM 打破了多少规则)，角色的某些部分却非常适合于使用该方法。面部表情系统将非常适合使用此类方案。每个状态都将是一个特殊的情感：高兴(撇嘴并半眯着眼)、悲伤(拱

起眉毛并耷拉着嘴)、疯狂(露牙、紧锁眉毛、睁大眼睛)等。每种情感都可激活到一个基于独立感知的水平，并且整个系统将与 AI 系统正在做的其他事情同时运行。

16.7.3　在较大 FSM 中作为单一状态的 FuSM

尽管并不是给定角色所使用的所有状态或行为都是独立或模糊的，但某些特殊部分却可能是这样。一个简单的例子就是运行正常状态机同时在地图上到处走动以获取道具并且与其他角色进行交互的角色。但当他站着不动时，一个模糊状态可能开始运行，将 3 种独立的行为进行混合：四处察看(他在该环境中的时间越短，他对该环境就越好奇)、不安的(他拥有的任务越多，或他等待得越久，或自从他上一次遭遇敌人的时间越短，他会变得越不安)、吹口哨(他感觉越安全，他站起走动时就会越喧闹)。该状态是 FSM 的当前状态，但它将运行任意或所有这些子状态以模仿该角色的站立行为。

16.7.4　层次化 FuSM

与 FSM 一样，FuSM 也可以很容易地进行层次化。为了便于实现，该骨架代码从 FuSMState 类继承得到了 FuSMMachine 类。然而，从设计上来看，这并不是最有用的概念。多个状态能够同时运行，因此除了组织性外，不需要将状态进行分组组合。如果要将这些变种方法进行组合，那这将更有用。可以使用一个 FuSM 来包含额外的使用以前提到的“有限数量当前状态”方法的 FuSM。每个子 FuSM 都将在其子类型中返回最高优先级的状态，然后所有这些最高优先级状态将在父 FuSM 中运行。

另外一种类型或许是这么一种 FSM，它的每个状态都是一个 FuSM。事实上，这是一个能够基于游戏事件或感知变化来转换其整个模糊状态系统的模糊系统。这是一个非常强大且具有通用用途的系统。

可以设想一个层次化 FSM，它的状态要么是 FuSM(对于更加动态和紧急的行为)，要么是常规 FSM(对于更加静态或对游戏事件半脚本化反应的行为)，它能够使程序员用最合适的系统来最好地适应游戏的特定状态。

16.7.5　数据驱动 FuSM

FuSM 系统通常比 FSM 系统要少一些数据驱动的设计，但它们也没有在很大程度上进行使用。数据驱动一个 FSM 通常意味着设计者可以采用某种方法(以脚本方式或通过某种视频界面)来建立状态并能够在状态之间呈现出转换连接性，并对转换分配条件。在 FuSM 中，控制信号发生了改变，因为设计者将决定添加哪些状态到总的机器(它将变成不同的元素，并被混合成终端行为)中，然后控制每个状态的激活计算，通过直接设定条件或简单的等式，或通过用限定符来影响一个标准计算(如调整状态的激活水平边界，或使用某个缩放因子)。此类数据可以在单个角色层级上进行调整，以从系统中获得不同的个性类型，或者在难度等级基础上调整，通过影响行为的选择来影响游戏的整体难度。

16.8　最优化

FuSM 可以同时运行许多不同的状态，因此在计算代价上将比 FSM 稍微昂贵。FuSM 不会对限定系统进行转换计算，但具有它们自身的激活计算代价。与 FSM 一样的最优化可以应用到模糊系统中，它们包括：负荷平衡、LOD 系统和共享数据。对这些技术的讨论可参见第 15 章。

16.9　设计上考虑的因素

FuSM 很适合于处理一些与 FSM 具有很大不同的 AI 问题。当决定采用基于 FuSM 的系统时，必须要对下列各项进行考虑：解决方案的类型、智能体的反应能力、系统的真实性、游戏类型、游戏平台、开发限制、娱乐限制。

16.9.1　解决方案的类型

FuSM 是另一种一般的问题解决工具，并能够用于设计大多数类型的解决方案类型。FuSM 有一点自相矛盾，因为它们既能够很好地用于高端(high-end)解决方案类型，又能很好地用于低端(low-end)解决方案类型。原因在于这两种类型都是联合一些元素来获取最终解决方案的有机解决方案。该领域的范围也没那么广泛，因此也不会倾向于影响 FuSM 的弱化问题。更程式化或脚本化的行为(类似于在路中间结束之类的行为)往往更适合于基于状态的系统，因为它们往往具有许多先决条件的活动并通常由感知来决定。RTS 游戏的高层决策系统将结合几个模糊状态(如侦察、资源搜集、外交、战斗以及防御等)的输出来确定其整体活动。更高层的决策处理过程将为每个领域分配一个顾问，然后混合这些顾问的建议来形成关于如何运行文明游戏的整体决策。低层或战术决策例子包括将即时命令或目标(“到这儿来”、“攻击这个单元”、“采集该资源”)与行为的二级状态(运动到其他单元以寻求支持、当不是一个战斗单元时避免战斗、当严重受伤时则逃跑等)进行混合。

16.9.2　智能体的反应能力

给定一个稀疏连接的状态结构，FuSM 通常比 FSM 更具反应性，因为这里不存在一个角色为了达到目标必须要通过的转换。但是，就简单的 FSM 或相互连接的 FSM 而言，这两种方法在成本上并无多少差别，并且系统中的每个状态都能够实现几乎任意级别的反应性。在本节关于惯性 FuSM 中描述的技术能够用于帮助调整游戏需要的智能体反应性级别。

16.9.3　系统的真实性

基于 FuSM 游戏的真实性具有很重要的意义，因为系统的最终行为是感知反应的一个连续曲线。这要比角色达到一定门限然后改变到其他状态更具真实性。通过调整系统的当前行为，而不是完全改变行为，一个设计得很好的 FuSM 将以一种真实的方式对感知变化

做出反应。大多数人都是通过稍微修正他们正在进行的行为来对一个新形势做出反应的(除非这个新形势是危及生命的或令人震惊的，即使在那种情况下，这个新的行为也是从人们已经在做的行为经过一定延时而开始的，而且FuSM也能够模仿此类的快速变化行为)。

16.9.4 游戏类型

作为一种非常通用的技术，FuSM能够以一种至少是有限的方式适用于任意类型的游戏。当作为一个主要的游戏范围AI框架时，FuSM则只针对有限的游戏类型。我们不会设法使用一个模糊状态系统来设计一个线性的、脚本化的游戏。但即使是在通常不需要此类问题解决方法的游戏中，也可能针对那类符合FuSM的模糊判断使用FuSM。例如，可以使用FuSM作为框架来编写游戏的感知系统。感知通常是独立的，并可以不用太多考虑其他感知而进行编码。感知具有任意的输出值(布尔值、连续浮点值、枚举类型等)这个事实非常适合于FuSM系统。执行此类工作的FuSM将使用不同的状态来表示各个感知，用状态的Update()方法计算感知值，并把激活水平当作是游戏需要更新感知的指示器。所有的二级感知计算，如反应时间、负荷平衡等都可以通过CalculateActivation()函数进行处理，尽管通过FuSMState类的特殊数据成员(它能够记录所有的调度系统)可以处理真实的时间调度更新，从而模糊机可以递减计时器或为更新的状态确定触发器。

16.9.5 游戏平台

FuSM的存储器和CPU需求是最小的，因此FuSM通常是不受平台约束的。然而，它们的确有助于更细微的行为，而这通常属于PC游戏的领域。是否使用它们通常更多是一个游戏设计关心的问题。

16.9.6 开发限制

如果要解决的AI问题属于FuSM能够处理得很好的那类情形，那么不存在比FuSM更好的设计方法。FSM很容易理解和设计，但FuSM也不是更难，而且它能够提供一个更丰富和更动态的产品。FuSM和FSM一样容易调试，因为即使它们具有很大范围的行为输出，它们也仍然是确定性的。

16.9.7 娱乐限制

调整难度设置、平衡具体的行为以及其他娱乐关注点一般都可以由FuSM来简单解决。它们可以在感知层级上以状态为基础进行调整，或是任意组合方式。有些行为可能具有与其他行为的增强效益(比如在AIsteroids设计中，攻击状态帮助单纯的躲避状态的能力)，并需要谨慎地进行某些调整，但通常单独状态可以分别进行调整。

16.10 小结

FuSM建立在简单的FSM系统之上，使得复杂行为(它们能够分解成分离且独立的动作)可以通过在不同的激活水平上对这些动作进行混合来进行构建。这种对FSM概念的强

大扩展使 FuSM 方法具有设计更大范围输出行为的能力，但增加了这种集合行为构建方式的必要条件。

- FuSM 的定义相当模糊，在真正的 FuSM 和其相似系统(如具有模糊转换的 FSM、概率 FSM、马尔可夫模型和实际的模糊逻辑系统等)上存在着混淆。
- FuSM 不使用一个单一的当前状态，而是具有任意数量的活动状态，当激活时，每个活动状态具有一个可变的激活水平。
- FuSM 中的一些状态可以具有数字激活水平，并且该系统某部分的这种反模糊化非常好，且不影响该整体方法。
- 本书讨论的 FuSM 框架建立在 3 个基类之上：FuSMState、FuSMMachine 和 FuSMAIControl。
- 最初的游戏并不能很好地符合 FuSM 模型，因此我们增加了一个新的飞船类，即 Saucer。它可以无重量飞行(没有惯性或加速度)，具有能够朝任意方向发射的炮塔，以及可以拽拉宝物到其自身的牵引光束。这使得游戏能够更理想地适应 FuSM 控制结构，因为 Saucer 主要使用独立的系统，大多数都具有可变的激活水平。
- 在 AIsteroids 试验平台上设计 FuSM 只需要 4 个状态：Approach、Attack、Evade 和 GetPowerup。它的状态实现要比 FSM 系统中的状态实现简单，并且感知计算也更简单，但这更多是因为 Saucer 打破了常规飞船所遵守的一些游戏规则，而不是 AI 技术上的转变。然而，Saucer 在性能上要比 FSM 设计高级，并且几乎能够无限期地进行游戏。
- 为了获取更好的性能而对 AIsteroids 游戏进行扩展，包括把缠绕的世界用攻击和躲避来表示，以及使用弹药管理程序。
- FuSM 系统的优势是容易设计(对于合适类型的问题)、实现、扩展、维护和调试。它们为更大范围的行为个性提供了可能，并不会产生 FSM 中的状态震荡问题。
- FuSM 系统的劣势是它们不像 FSM 那样是一个通用的解决方案系统，并且如果设计不当会产生行为震荡问题，但这很容易通过预先计划来克服。
- 可以编写具有有限数量当前状态的 FuSM 来调整希望在游戏中使用的模糊级别。在状态的子类型内部，可以有一个、少数几个或有限个当前状态。
- 作为角色支持系统的 FuSM 是添加模糊性的一种重要方式，它只有在复杂角色设计中需要时才采用，比如在一个面部表情系统中。
- 在较大 FSM 中作为一个单个状态的 FuSM 能够用于表示一个具有模糊行为判断的角色，但只限于一个较大的限定游戏状态。
- 层次化 FuSM 在它们最纯粹的形式中很少使用，因为它们没有多大意义，但当与其他状态机变种结合时，便能体现出它们真正的力量。
- 数据驱动 FuSM 包括设计者对角色可能使用的特殊状态的控制以及影响激活水平计算。
- FuSM 能够从常规 FSM 使用的优化技术中获益。

17 基于消息的系统

在现代游戏编程世界中，只有一种技术使用得比状态机还多，那就是消息(有时也称之为事件)。消息的概念是非常简单的。假设 A 和 B 是两个游戏实体，为了检测某些特定变化，我们并不是在每个时刻都让 A 对 B 进行检测，而是在 B 发生变化时，通过一个由 B 传送到 A 的“消息”来告知 A 这个变化。这意味着不需要浪费计算周期或代码空间(通过遍及游戏引擎的检测)来确定变化是否发生。游戏通过消息来告知实体它所感兴趣的事件，然后做它自己的事情，而不用对它担心，直到另一个消息到来。

17.1 消息概述

与本书讨论的许多其他技术不同，消息从本质上来说并不是一个决策结构。它更多的是一种通信技术，用在游戏中有助于组织性、最优化以及减轻游戏中不同对象和类之间的通信。消息作为一个二级系统，处于游戏的基本决策结构之下。它被用于那些该类型通信具有很高性价比的游戏中，并且随着游戏变得越来越复杂，该类型也在逐步发展。大多数现代游戏都能够因其引擎使用消息系统而获益。

AI 系统具有两个主要的特征使其能够很好地使用基于消息的通信：

- AI 控制的角色大多数都被设计为是“反应性”的，因为它们经常依靠外部的感知变化来影响角色的行为。这具有重要意义，因为我们正在对与游戏交互的人类玩家做出反应。但这也意味着 AI 系统将为感知的变化等待很长时间，或者为了确定这些感知变化而执行许多计算。AI 角色可以在大部分游戏玩法时间内都是完全不活动的，特别是当它们对于人类玩家是不可见的时候。在这些周期中花时间来执行计算都将是浪费的。
- AI 是游戏开发中非常高层的部分。在设计条件检测和 AI 系统的行为时，AI 程序员可能必须跟许多其他游戏系统(包括动画、角色和世界的物理性、游戏玩法、控制、声音等)进行通信。在执行引擎各部分间的通信时，如果不进行某种形式的抽象，游戏将陷入一种能够访问游戏引擎的所有其他领域的 AI 系统。尽管这将给予 AI 程序员许多能力(有利也有弊)，但这通常被认为是一种不好的编程方法，并将带来不可维护的系统，即几乎不能扩展、调试和理解。

消息是唯一可以克服这些问题的技术。它设计的是具有完全反应性的系统，因为系统只对事件消息做出响应。它也从代码中对数据进行了解耦，所以 AI 系统能够从游戏的其

他领域请求数据，且不必完全访问这些系统的基本类结构。它提供了一种在 AI 代码片断和更大的游戏之间移动数据和事件的重要方式，从而基本的 AI 系统能够发生变化，而不必记录从游戏的其他部分获取信息的整个过程。

本章将为一般的事件消息系统设计一个框架，其中该系统能够用于整个游戏或部分游戏。这个通用框架将由 3 个主要部分组成：消息对象、消息井、客户端句柄。图 17-1 是这种体系结构的直观表示。

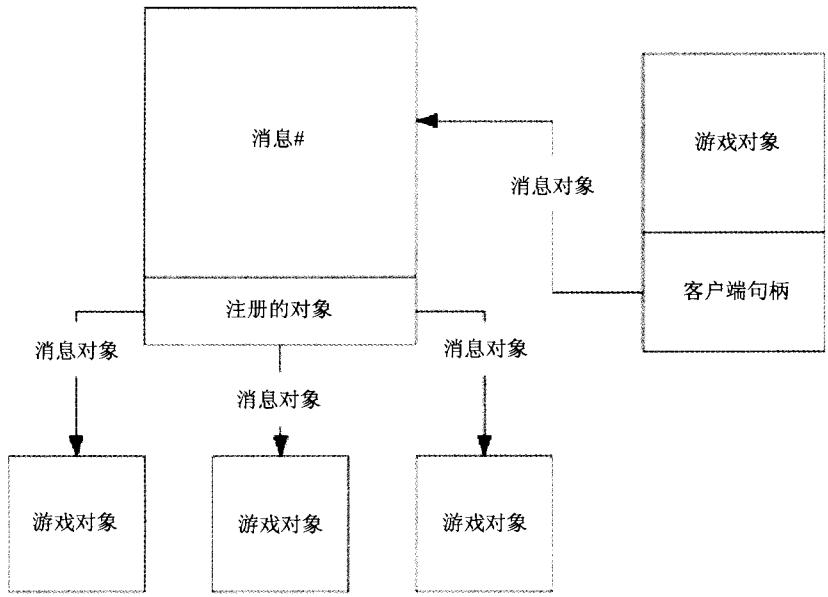


图 17-1　消息系统纵览

17.2　消息的骨架代码

本章将要介绍的一般消息系统使用下面这些基类进行设计：

- **Message 类**。存储消息的个体信息需求。
- **MessagePump 类**。是中枢消息路由器。
- **客户端句柄**。运行代码以容纳任意即将到来的消息。

在下面各节中，将对这些类进行充分讨论。

17.2.1　Message 对象

消息对象是存储消息的一般结构。程序清单 17-1 是消息对象的 header，从中可知，它仅仅包含少数几个数据字段：

- m_typeID 是消息的类型。
- m_fromID 是发送消息的对象的唯一 ID。这是消息的可选数据字段，因为消息是可以进行匿名发送的。

- m_toID 是消息要传送到的对象的 ID。这也是一个可选字段，因为消息并也可以具有那些为某个给定状态传送消息而登记的消息，因此消息本身不需要进行具体指定。
- m_timer 用于设置消息传送中的延时。
- m_delivered 被消息并用于标记已经处理过的消息，从而它们可以从队列中清除。
- DataMessage 也被包含在文件中并且它是一个简单的模板类，具有关于接收的数据类型的单一数据字段。可以用这个类来传递具有简单数据字段的消息，但如果要发送更复杂的数据，则需要设计额外类型的消息。无论如何，消息处理回调函数都会将进来的消息投射到它所知道的消息类型中(通过消息 ID 类型)并通过投射指针来访问这些数据。

程序清单 17-1　Message 类的 Header

```
class Message
{
public:
    //constructor/functions
    Message(int type = MESSAGE_DEFAULT){m_typeID =
                type;m_delivered = false;m_timer = 0.0f;}
    ~Message(){}

    //data
    int m_typeID;
    int m_fromID;
    int m_toID;
    float m_timer;
    bool m_delivered;
};

//simple template message class for simple data passing
template <typename T>
class DataMessage: public Message
{
public:
    DataMessage(int type, T data):Message(type){m_dataStorage = data;}
    ~DataMessage(){}

    //data member
    T m_dataStorage;
};
```

17.2.2　MessagePump 类

Messagepump 类是存储所有可能的消息类型的类，同时也是来回传送消息的中央位置。Messagepump 可以将延迟消息、广播消息与感兴趣的对象联系起来，并通常起到系统的邮局的作用。程序清单 17-2 给出了该类的 header，而程序清单 17-3 则是其重要实现。

根据 header 可知，我们将把 messagepump 类设计成一个单元素集合(singleton，它是一种软件设计模式，仅仅是指在整个游戏中该类只有一个实例)。在 header 末尾的#define 提供了对单元素集合类结构的简洁访问。

程序清单 17-2　MessagePump 类的 Header

```
typedef std::list<Message*> MessageList;
typedef std::map<int,MessageType*> MessageTypeMap;

class MessagePump
{
public:
     static inline MessagePump& Instance()
     {
          static MessagePump inst;
          return inst;
     }

     static void Update(float dt);
     static void AddMessageToSystem(int type);
     static int RegisterForMessage(int type, int objected,
                                   Callback& cBack);
     static void UnRegisterForMessage(int type, int objectID);
     static void SendMessage(Message* newMessage);

protected:
     MessagePump();
     MessagePump& operator= (const MessagePump&){}

private:
     static MessageTypeMap m_messageTypes;
     static MessageList m_messageQueue;
};

#define g_MessagePump MessagePump::Instance()
```

程序清单 17-2 给出了 pump 类中的一些重要函数。

Update()方法用于检测队列中的每个消息，并且如果它是一个延迟消息，则减小它的定时器，或者将消息传送到那个通过提供一个回调函数为该消息登记的实体。该函数然后从队列中清除所有已经传递了的消息。

AddMessageToSystem()用于向消息井的一系列可能消息中插入消息类型。它可以在任何时候执行，不管是在设计类还是得到一个新的对象(需要系统在新的消息类型上存储信息)。

RegisterForMessage()函数需要两个条件：系统中存在该消息类型，还不曾注册过该消息。如果这两个条件都满足，它将把所接收的特定消息类型添加到通知列表中。

UnRegisterForMessage()用于完成与上述相反的工作。它在所有为某一特定消息而进行的注册中循环，并从列表中将玩家移除。

程序清单 17-3　MessagePump 类的实现

```
//--------------------
void MessagePump::Update(float dt)
{
    if(m_messageQueue.size() == 0)
        return;

    //process messages
    MessageList::iterator msg;
    for(msg=m_messageQueue.begin();
        msg!=m_messageQueue.end();++msg)
    {
        if((*msg)->m_timer > 0)
        {
            //delayed message, decrement timer
            (*msg)->m_timer -= dt;
        }
        else
        {
            //check for registrations
            MessageTypeMap::iterator mType;
            mType = m_messageTypes.find((*msg)->m_typeID);
            if(mType == m_messageTypes.end())
                 continue;

            MessageRegList::iterator msgReg;
            for(msgReg=(*mType).second->
                 m_messageRegistrations.begin();
                 msgReg!=(*mType).second->
                 m_messageRegistrations.end();++msgReg)
            {
                 //deliver message by launching callback
                 if((*msgReg)->m_callBack)
                     (*msgReg)->m_callBack.function
                              ((*msgReg)->m_objectID,(*msg));
            }
            (*msg)->m_delivered = true;
        }
    }

    //remove all delivered messages from queue
    MessageList::iterator end = m_messageQueue.end();
    MessageList::iterator newEnd = std::remove_if
              (m_messageQueue.begin(),m_messageQueue.end(),
               RemoveIfDelivered);

    if(newEnd != end)
        m_messageQueue.erase(newEnd,end);
}
```

```
//--------------------
void MessagePump::AddMessageToSystem(int type)
{
    //ensure that this type isn't already in the system
    MessageTypeMap::iterator mType;
    mType = m_messageTypes.find(type);

    if(mType == m_messageTypes.end())
    {
        MessageType *newType = new MessageType;
        newType->m_typeID = type;
        m_messageTypes[type] = newType;
    }
}

//--------------------
void MessagePump::SendMessage(Message* newMessage)
{
    m_messageQueue.push_back(newMessage);
}

//--------------------
int MessagePump::RegisterForMessage(int type, int objectID,
                                    Callback& cBack)
{
    //only register once
    MessageTypeMap::iterator mType;
    mType = m_messageTypes.find(type);

    if(mType == m_messageTypes.end())
        return REGISTER_ERROR_MESSAGE_NOT_IN_SYSTEM;

    MessageRegList::iterator msgReg;
    for(msgReg=(*mType).second->
        m_messageRegistrations.begin();
        msgReg!=(*mType).second->
        m_messageRegistrations.end();++msgReg)
    {
         if((*msgReg)->m_objectID == objectID)
             return REGISTER_ERROR_ALREADY_REGISTERED;
    }
    //add new registration
    MessageReg* newRegistration = new MessageReg;
    newRegistration->m_callBack = cBack;
    newRegistration->m_objectID = objectID;
    (*mType).second->m_messageRegistrations.
                    push_back(newRegistration);
    return REGISTER_MESSAGE_OK;
```

```
}

//--------------------
void MessagePump::UnRegisterForMessage(int type, int objectID)
{
    //find entry
    MessageTypeMap::iterator mType;
    mType = m_messageTypes.find(type);

    if(mType == m_messageTypes.end())
        return;

    MessageRegList::iterator msgReg;
    for(msgReg=(*mType).second->
          m_messageRegistrations.begin();
          msgReg!=(*mType).second->
          m_messageRegistrations.end();++msgReg)
    {
          if((*msgReg)->m_objectID == objectID)
          {
              (*mType).second->
                 m_messageRegistrations.erase(msgReg);
              delete (*msgReg);
              //you can exit out here, there is only one
              //registration allowed per message type
              return;
          }
    }
}
```

17.3　客户端句柄

在本实现中，消息句柄函数将被当作回调函数来编写。回调函数是当消息被传送到某一特定游戏对象时，代表游戏对象希望运行的且用于响应消息的代码的函数。

另一种经常使用的一般消息处理系统是只具有一个 ProcessMessage()函数，该函数在本质上是一个较大的转换声明，具有响应任意接收消息类型的代码或函数调用。回调函数的使用提供了比这更灵活的系统，并避免了棘手的包罗万象的处理功能。

C++语言不允许直接使用成员函数作为回调函数，因此我们将使用一个普遍方法，即为要从中继承的回调对象使用虚拟回调类(用一个虚函数来表示回调方法的形式)。然后在传统的 C 语言回调函数中使用这些回调对象。程序清单 17-4 给出了虚拟回调函数的 header，以及使用该接口的一个例子。

程序清单 17-4　回调系统示例

```
class Callback
{
public:
     virtual void function(int pid, Message* msg);
};

class EvadeCallback : public Callback
{
     void function(int pid, Message* msg);
};
```

17.4　在 AIsteroids 试验平台上的示例实现

在本节中，我们将使用消息来重复 FSM 在我们的试验平台上所做的工作以执行所有的状态转换，同时使用该系统来影响其他的游戏变化，比如用一个状态来给出对飞船的命令。为了实现该目的，我们需要对最初的有限状态系统做一些改变，并为转换回调函数和消息函数加入新的代码。

必须注意 AIsteroids 试验平台并不是需要该技术的一种游戏。感知变化经常发生，从而状态转换也经常发生。很少有主飞船等待做某件事情的时间。游戏中只有极少数的对象需要一个简洁的通信或交互通道。游戏中的消息实际上增加了成本，并很可能稍微拖慢系统，因为随着状态的改变，它需要不停地注册和解除注册消息。这个实现只是为了给出该方法的实际应用，在游戏环境中该实现并不是消息的一个好的实现。

17.4.1　MessState 类

只需要对状态类本身做少许改变。由于该逻辑不再存在于每个单独状态中，故不再需要 CheckTransition()函数。相反，将由 Control 类来执行这种计算并发送正确的消息来启动转换回调函数。

此时，Enter()和 Exit()方法将负责为状态建立消息注册，以及所有其他的清除功能。这样，消息就被局限于特定状态，因为之后任意给定的状态将只对那些在状态内具有意义的消息做出响应。

Init()方法可用于向系统中添加状态需要的额外消息类型。这对于状态使用的自我管理消息非常有用。例如，Flee 状态将首先注意到敌人，等待半秒钟时间(模仿一个反应时间)，然后开始逃跑。可以把这作为两个状态(Notice，然后 Flee)，或者是 Flee 状态本身，进入该状态后将其自身置于一个等待模式(在该模式中角色的先前行为将不会发生改变)并向其自身发送一个延迟半秒的唤醒消息。当该消息恢复时，它将改变角色的等待状态，之后该角色将逃跑。

程序清单 17-5　MessState 类的 Header

```
class MessState
{
public:
    //constructor/functions
    MessState(int type = FSM_STATE_NONE,Control* parent = NULL)
                                        {m_type = type;m_parent=parent;}
    virtual void Enter()                {}
    virtual void Exit()                 {}
    virtual void Update(float dt)       {}
    virtual void Init()                 {}

    //data
    Control* m_parent;
    int m_type;
};
```

17.4.2　MessMachine 类

状态机本身与有限状态机相比仅有一个变化，即 UpdateMachine()函数。如程序清单 17-6 所示，该方法缺少了调用当前状态的 CheckTransitions()函数的这一行。该模型不会轮询转换的变化。相反，状态转换的消息将由单个状态发送到控制类中，而且控制类通过直接设置 MessMachine 的 m_goalID 成员来对这些消息进行响应。

程序清单 17-6　MessMachine 类的更新实现

```
void MessMachine::UpdateMachine(float dt)
{
    //don't do anything if you have no states
    if(m_states.size() == 0 )
        return;

    //don't do anything if there's no current
    //state, and no default state
    if(!m_currentState)
        m_currentState = m_defaultState;
    if(!m_currentState)
        return;

    //check for transitions, and then update
    int oldStateID = m_currentState->m_type;

    //switch if there was a transition
    if(m_goalID != oldStateID)
    {
        if(TransitionState(m_goalID))
        {
            m_currentState->Exit();
```

```
            m_currentState = m_goalState;
            m_currentState->Enter();
        }
    }
    m_currentState->Update(dt);

}
```

17.4.3 MessAIControl 类

程序清单 17-7 包含了 MessAIControl 类的 header，以及控制器用于响应改变机器状态改变的回调类。注意到它与常规 FSM 控制器十分相似。控制器间的真正差别在于它们的设计。程序清单 17-8 给出了消息控制器文件中的相关函数。

程序清单 17-7　MessAIControl 类的 Header

```
class ChangeStateCallback : public Callback
{
    void function(int pid, Message* msg);
};

class MessAIControl: public AIControl
{
public:
    //constructor/functions
    MessAIControl(Ship* ship = NULL);
    void Update(float dt);
    void UpdatePerceptions(float dt);
    void Init();
    void SetMachineGoalID(int state);

    //perception data
    //(public so that states can share it)
    GameObj* m_nearestAsteroid;
    GameObj* m_nearestPowerup;
    float m_nearestAsteroidDist;
    float m_nearestPowerupDist;
    bool m_willCollide;
    bool m_powerupNear;
    float m_safetyRadius;

private:
    //data
    MessMachine* m_machine;
    ChangeStateCallback m_changeStateCallback;
};
```

设计差别是显而易见的。构造器必须为游戏建立消息系统，因此它需要向消息井中添加所有可应用的消息类型。构造器也对那些变化的状态消息进行注册，因为控制器此时将通过响应状态的消息请求来促成状态机转换。

最大的变化是 UpdatePerceptions()方法。包含在 CheckTransitions()状态函数中的一些逻辑做了相应的改变。这的确不利于原来 FSM 系统所具有的模块化组织模型，并且这里给出的这些代码清楚地表明不能在游戏中使用该方法。尽管该函数在这个简单的试验应用程序中是足够简单的，但是即使是一个中等复杂的游戏也需要大量逻辑来产生为了执行所有转换所需要的消息。再次强调，该实现仅仅是给出使用中的代码，它并不保证这是一个好的游戏用例。

程序清单 17-8　MessAIControl 类的实现

```
//--------------------
MessAIControl::MessAIControl(Ship* ship):
AIControl(ship)
{
   //construct the state machine and add the necessary states
   m_machine = new MessMachine(FSM_MACH_MAINSHIP,this);
   MStateApproach* approach = new MStateApproach(this);
   m_machine->AddState(approach);
   m_machine->AddState(new MStateAttack(this));
   m_machine->AddState(new MStateEvade(this));
   m_machine->AddState(new MStateGetPowerup(this));
   m_machine->AddState(new MStateIdle(this));
   m_machine->SetDefaultState(approach);

   g_MessagePump.AddMessageToSystem(MESSAGE_WILL_COLLIDE);
   g_MessagePump.AddMessageToSystem(MESSAGE_NO_ASTEROIDS);
   g_MessagePump.AddMessageToSystem(MESSAGE_NO_POWERUPS);
   g_MessagePump.AddMessageToSystem(MESSAGE_ASTEROID_NEAR);
   g_MessagePump.AddMessageToSystem(MESSAGE_ASTEROID_FAR);
   g_MessagePump.AddMessageToSystem(MESSAGE_POWERUP_NEAR);
   g_MessagePump.AddMessageToSystem(MESSAGE_POWERUP_FAR);
   g_MessagePump.AddMessageToSystem(MESSAGE_CHANGE_STATE);

   g_MessagePump.RegisterForMessage(MESSAGE_CHANGE_STATE,
                    m_ship->m_ID,m_changeStateCallback);
}

//--------------------
void ChangeStateCallback::function(int pid, Message* msg)
{
   ChangeStateMessage* csMsg = (ChangeStateMessage*)msg;
   int newState = *((int*)(csMsg->m_data));
   ((MessAIControl*)Game.m_AIControl)->
                              SetMachineGoalID(newState);
}

//--------------------
```

```
void MessAIControl::SetMachineGoalID(int state)
{
    m_machine->SetGoalID(state);
}

//--------------------
void MessAIControl::Init()
{
    m_willCollide = false;
    m_powerupNear = false;
    m_nearestAsteroid = NULL;
    m_nearestPowerup = NULL;
    m_safetyRadius = 15.0f;

    m_target = new Target;
    m_target->m_size = 1;
    Game.PostGameObj(m_target);
}

//--------------------
void MessAIControl::Update(float dt)
{
    if(!m_ship)
    {
        m_machine->Reset();
        return;
    }

    UpdatePerceptions(dt);
    m_machine->UpdateMachine(dt);
}

//--------------------
void MessAIControl::UpdatePerceptions(float dt)
{
    if(m_willCollide)
        m_safetyRadius = 30.0f;
    else
        m_safetyRadius = 15.0f;

    //store closest asteroid and powerup
    m_nearestAsteroid = Game.
            GetClosestGameObj(m_ship,GameObj::OBJ_ASTEROID);
    if(m_ship->GetShotLevel() < MAX_SHOT_LEVEL)
        m_nearestPowerup = Game.
            GetClosestGameObj(m_ship,GameObj::OBJ_POWERUP);
    else
        m_nearestPowerup = NULL;

    //reset distance to a large bogus number
```

```
m_nearestAsteroidDist = 100000.0f;
m_nearestPowerupDist = 100000.0f;

//asteroid collision determination
m_willCollide = false;
if(m_nearestAsteroid)
{
    float speed = m_ship->m_velocity.Norm();
    m_nearestAsteroidDist = m_nearestAsteroid->
                m_position.Distance(m_ship->m_position);
    float dotVel;
    Point3f normDelta = m_nearestAsteroid->m_position -
                m_ship->m_position;
    normDelta.Normalize();
    float astSpeed = m_nearestAsteroid->
                                        m_velocity.Norm();
    if(speed > astSpeed)
         dotVel = DOT(m_ship->UnitVectorVelocity(),
                      normDelta);
    else
    {
         speed = astSpeed;
         dotVel = DOT(m_nearestAsteroid->
                      UnitVectorVelocity(),-normDelta);
    }
    float spdAdj = LERP(speed / AI_MAX_SPEED_TRY, 0.0f,
                        50.0f)*dotVel;
    float adjSafetyRadius = m_safetyRadius + spdAdj +
                            m_nearestAsteroid->m_size;

    //if you're too close, and I'm heading
    //somewhat towards you, flag a collision
    if(m_nearestAsteroidDist <= adjSafetyRadius &&
       dotVel > 0)
    {
       m_willCollide = true;
       Message* msg = new Message(MESSAGE_WILL_COLLIDE);
       g_MessagePump.SendMessage(msg);
    }
    else
    {
       Message* msg = new Message(MESSAGE_WONT_COLLIDE);
       g_MessagePump.SendMessage(msg);
    }

}
else
{
    Message* msg = new Message(MESSAGE_NO_ASTEROIDS);
    g_MessagePump.SendMessage(msg);
```

```
    }

    //powerup near determination
    m_powerupNear = false;
    if(m_nearestPowerup)
    {
        m_nearestPowerupDist = m_nearestPowerup->m_position.
                               Distance(m_ship->m_position);
        if(m_nearestPowerupDist <= POWERUP_SCAN_DIST)
            m_powerupNear = true;
    }
    else
    {
        Message* msg = new Message(MESSAGE_NO_POWERUPS);
        g_MessagePump.SendMessage(msg);
    }

    //arbitrate asteroid/powerup near messages
    if(m_powerupNear && m_nearestAsteroidDist >
                        m_nearestPowerupDist)
    {
        Message* msg = new Message(MESSAGE_POWERUP_NEAR);
        g_MessagePump.SendMessage(msg);
    }
    else if(m_nearestAsteroid)
    {
        if(m_nearestAsteroidDist > APPROACH_DIST)
        {
            Message* msg = new Message(MESSAGE_ASTEROID_FAR);
            g_MessagePump.SendMessage(msg);
        }
        else
        {
            Message* msg =new Message(MESSAGE_ASTEROID_NEAR);
            g_MessagePump.SendMessage(msg);
        }
    }
}
```

17.5　状态编码

单个状态本身几乎没有发生变化。本书将给出其中的一个 idle 状态，来阐述这种差别。程序清单 17-9 是 MStateIdle 的 Header(包含了所有需要的回调对象的声明)，程序清单 17-10 则是与常规 FSM 方法的设计差别。

程序清单 17-9　MStateIdle 的 Header

```
//callbacks for handling messages
class EvadeCallback : public Callback
{
   void function(int pid, Message* msg);
};
class ApproachCallback : public Callback
{
   void function(int pid, Message* msg);
};
class AttackCallback : public Callback
{
   void function(int pid, Message* msg);
};
class GetPowerupCallback : public Callback
{
   void function(int pid, Message* msg);
};

class MStateIdle : public MessState
{
public:
   //constructor/functions
   MStateIdle(Control* control):
              MessState(FSM_STATE_IDLE,control){}
   void Enter();
   void Exit();
   void Update(float dt);
   EvadeCallback m_evadeCallback;
   ApproachCallback m_approachCallback;
   AttackCallback m_attackCallback;
   GetPowerupCallback m_getPowerupCallback;

};
```

程序清单 17-10　MStateIdle 与常规 FSM 的设计差别

```
//--------------------
void MStateIdle::Enter()
{
   g_MessagePump.RegisterForMessage(MESSAGE_WILL_COLLIDE,
                          m_parentID,m_evadeCallback);
   g_MessagePump.RegisterForMessage(MESSAGE_ASTEROID_FAR,
                          m_parentID,m_approachCallback);
   g_MessagePump.RegisterForMessage(MESSAGE_ASTEROID_NEAR,
                          m_parentID,m_attackCallback);
   g_MessagePump.RegisterForMessage(MESSAGE_POWERUP_NEAR,
                          m_parentID,m_getPowerupCallback);
}
```

```
//--------------------
void MStateIdle::Exit()
{
    g_MessagePump.UnRegisterForMessage(MESSAGE_WILL_COLLIDE,
                                       m_parentID);
    g_MessagePump.UnRegisterForMessage(MESSAGE_ASTEROID_FAR,
                                       m_parentID);
    g_MessagePump.UnRegisterForMessage(MESSAGE_ASTEROID_NEAR,
                                       m_parentID);
    g_MessagePump.UnRegisterForMessage(MESSAGE_POWERUP_NEAR,
                                       m_parentID);
}

//--------------------
void EvadeCallback::function(int pid, Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
                (MESSAGE_CHANGE_STATE,MFSM_STATE_EVADE);
    newMsg->m_fromID = pid;
    g_MessagePump.SendMessage(newMsg);
}

//--------------------
void ApproachCallback::function(int pid,Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
                (MESSAGE_CHANGE_STATE, FSM_STATE_APPROACH);
    newMsg->m_fromID = pid;
    g_MessagePump.SendMessage(newMsg);
}

//--------------------
void AttackCallback::function(int pid,Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
                (MESSAGE_CHANGE_STATE, FSM_STATE_ATTACK);
    newMsg->m_fromID = pid;
    g_MessagePump.SendMessage(newMsg);
}

//--------------------
void GetPowerupCallback::function(int pid,Message* msg)
{
    DataMessage<int>* newMsg = new DataMessage<int>
                    (MESSAGE_CHANGE_STATE, FSM_STATE_GETPOWERUP);
    newMsg->m_fromID = pid;
    g_MessagePump.SendMessage(newMsg);
}
```

在程序清单 17-10 中，使用了 DateMessage 模板来发送具有附加数据的消息。DataMessage 包含了数据成员 m_dataStorage，该成员的类型在实例化中被传递到模板中。当该消息被传送时，接收回调函数将进来的消息指针投射到 DateMessage 结构的正确类型中，并能够访问该数据。更复杂的数据也将使用相同的系统。如果一个消息必须要传递多个数据字段，那么该消息只需通过一个包含必要数据字段的 struct 来进行设计。在发送时刻，struct 字段将被初始化为相关的值，而且接收器也只是将进来的消息指针转换到所使用的 DataMessage 类型中。

17.6　使用该系统的 AI 的性能

该系统的游戏性能几乎与常规 FSM 实现的游戏性能相当。除了转换产生的方法之外并无任何改变。然而，如前所述，实际的 CPU 性能似乎有所下降，因为我们在这里使用消息系统是相当浪费的。当某些特定消息进入或退出时，状态需要对它们进行注册和解除注册。

与常规 FSM 实现的不同之处是这里具有更多难以躲避的行星。这里给出了让状态机使用消息来取得状态的一个缺陷，即转换优先级。状态变化通常由为消息事件注册了的回调函数来触发，因此我们不再需要控制在某一单个游戏循环中发生的多个回调函数的优先级。系统只是根据最先到来的消息而改变状态。事实上，系统将为每个到来的消息改变状态(因为消息并在状态机之前进行更新)，所以起作用的唯一状态变化是队列中的“最后一个”状态变化消息。这可以在感知层级通过简单的优先级仲裁来确定，但此时又失去了消息系统最开始具有的逻辑解耦能力。仅仅能够发送最高优先级的消息，并且本质上系统将很费力地调用成员函数。较好的方式是使用一系列进入到机器中的状态改变请求，并根据它们来确定优先级。然而，该方法并不是很简洁，因为必须面对在状态层级上的集中性逻辑。解决该问题的真正方法是前面提到的设计一个变化状态队列，但在消息本身分配优先级数，并用这些优先级数进行仲裁。消息优先级将在 17.7.1“消息优先级”一节进行讨论。

17.6.1　消息系统的优势

实际上，消息系统可以优化客户端代码。在我们的游戏中，这意味着代表客户端的状态由于不必考虑转换判断而得到了优化。服务器(游戏中的控制器类)必须要执行必要的计算，而且状态只是等待直到它们从服务器中获得了正确的信号。我们并没有节省任何成本(代码从状态移动到控制器上)，因此也没有什么特别的东西。

但是，有限的试验平台实现也只是消息系统能够完成的一个开头而已。我们也能够用消息来运行一个碰撞系统，告知游戏对象它们已经发生了碰撞。当宝物被收集时，它可以向飞船发送消息，告知飞船结果，而不是使用飞船类为获取宝物而具有的转换声明。单个行星可以检测与主飞船的距离，当处于一定界限或碰撞即将发生时则发送消息，从而加速控制器类 UpdatePerception 方法的碰撞判断。当结合所有这些示例，以及游戏能够利用的通用消息系统的许多其他方式时，该技术的力量将开始体现。类层级通信的解耦解脱了程序员，使他们可以用消息系统来完成那些之前需要在游戏区域之间对类进行完全访问的事件。

大多数系统都是事件驱动的，因此可以通过把消息流记入日志来帮助调试系统。如果系统是为此而编写的，那甚至可以记录消息流并将其回放到系统中来直接复制漏洞或具体的游戏行为。

通过把消息系统作为与其他游戏部分通信的主要方法，可以以一种完全模块化的方式来对类进行编写。这将加速类的编译时间(因为它并不需要在各处都包含文件来访问其他类)并使得内部方法更容易随时间而改变，因为与外界的唯一接口就是通过消息来进行传递。消息如何确定以及如何从各个类发送消息并不重要，唯一重要的是它们被发送出去。

17.6.2　消息系统的劣势

消息系统具有很少的转移点，因为它们的简单和通俗易懂使得它们很容易应用，而且正是它们的特性(带通知的分布式计算，而不是集中式的轮询计算)使得它们较少占有 CPU 时间。事件驱动的体系结构通常具有额外的存储器区域，这是由于需要跟踪消息(以及其他附属数据)而造成的。但对于简单的消息系统来说(比如本章实现的消息系统)，不存在不能克服的问题。甚至对于那些需要巨大数量到来和延迟消息以及大量传递数据的系统来说，也可以对消息井使用负荷平衡技术，这将平滑系统的处理过程以及地址存储器所关心的问题。但即使是没有该级别的复杂性，一个简单的消息系统也能向游戏引擎提供消息的大部分优势。

然而，存在一个过多好处的概念。试图完全合并消息的系统或许会遇到特殊化的问题。如果游戏框架是完全事件驱动的，游戏的某些部分将很难执行它所需要的轮询功能，或者如果轮询对象数目很大，与消息系统内部轮询有关的成本将变得非常难以处理。监视 CPU 的使用以注意到这点是非常重要的，并且用户(及其游戏引擎)应该足够灵活以使得模块成为可能，该模块使用一种更优化的方法来获取它所需要的数据。

另一个缺陷是本章先前在分析该系统用于 AIsteroids 游戏中的性能时所讨论的问题，即消息优先级。如果不通过给消息一定的优先级来扩展系统，那么将失去对到来消息的大量控制(它们在正常的基于状态的系统中是不明显的)。

17.7　范例扩展

本章给出的消息系统是非常直接和简单的。它很容易理解、使用和对系统进行添加。可以改进系统的一些因素，包括消息优先级、消息仲裁以及自动和扩展的消息类型。

17.7.1　消息优先级

通过为消息分配一个优先级，消息井能够对消息进行分类，从而更重要的消息可以最先得到处理。这对于那些消息系统只具有有限的时间以及必须要把这些时间分配给最紧急的事件的游戏来说尤其有用。然而，这也带来一个问题，即消息饥荒(message starvation)。必须要谨慎处理那些具有低优先级的消息，以确保它们不会永远呆在队列的最底端，不断

地被较高优先级的消息推回到队列中。如果拥有一些不需注意但仅仅是撤退状态事件的自由消息，那不成问题。但在使用消息系统来保持网络更新的在线游戏中，消息饥荒将导致完全丧失游戏同步。

17.7.2　消息仲裁

另一个普遍使用的过程是仲裁。这是一类记账(bookkeeping)程序，通过在处理之前读取到来的消息列表来执行，并寻找诸如消息冗余、消息冲突(两个消息实际上彼此对消)、消息饥荒或其他任意需要系统在空闲时确定的问题。之后，仲裁器根据一些内建的规则来处理各个问题。消息冲突可能导致两个消息都从队列里移除。如果这两个消息彼此没有“完全”对消，它们可能被移除并需要运行一个小的清除代码片断。消息冗余将只是涉及到移除多余的消息，除非该冗余有特殊的意义，比如具有细微差别和更新数据元素的额外的消息(尽管是相同类型的消息)。在这种情况下，除了最后一个外它们都应该被扔掉。此类的消息处理非常广泛，包括了许多层级的优化和提炼。然而，仲裁系统也可以非常复杂(如果允许)，因此必须小心，使得一个干净简洁的基于消息的系统不会因为杂乱的仲裁阶段而变得让人费解。

17.7.3　自动和扩展的消息类型

消息类型可以简化消息处理过程，并使频繁或自动发生的任务更容易处理。其他消息类型包括由系统进行不同处理的特例消息。所使用的一些普遍的消息类型如下：

- **周期消息**。它们是一些以一定的常数时间周期发生的消息，如每秒一次或一分钟两次等。某些事件驱动性很强的游戏使用一种更新消息，它每隔一定游戏时间便自动发生，而且通过发信号来让游戏中的所有对象调用它们的 Update()方法。
- **调试消息**。消息可以嵌入到可被一个中枢调试系统(模仿断言的一个简单系统，但不仅仅是为了观察而在一个中心位置设置陷阱或忽略被断言的代码来停止程序的执行)感知的系统中。调试系统可以识别这些“隐藏”消息(因为游戏的其他部分并不知道或不关心它们的存在)和一般的游戏变量，从而向程序员提供为了确定和调整代码片断所需要的东西。可以编写这些消息，使得它们可以通过设置一个定义来进行编译，因此最终的代码并不会因调试信息和过程而变慢。
- **确认消息**。某些系统可能需要延迟消息(stalling message)。当游戏对象 A 向 B 发送一条消息时，A 在传送之前等待(或延迟)来自 B 的一个响应。在接收消息时将自动向消息发送者发送一个确认信号，这样这种行为才能够被接纳。
- **即时消息**。在现在的实现中，消息是存储在一个队列中并且是在每个游戏循环中处理一次。即时消息将跳过这一传统，并将立即送往其他游戏对象进行处理。当具有需要即时注意的重载消息，或者某些游戏状态要比其他状态更重要但在消息结构中没有优先级系统时，即时消息是非常必要的。

17.8　最优化

事件驱动的 AI 是相当优化的(与更传统的轮询模型相比)，但只针对包含一个更多事件友好环境的系统；否则，消息将增加管理费用和复杂性。

正如在 AIsteroids 试验平台中所看到的，一些游戏并不是很适合使用消息范例。尽管在试验平台上实现消息系统(其转换是基于事件的)或许是一个有问题的例子，但游戏的其他部分将因为消息而受益，并且之后的游戏扩展也可利用一个已经实例化的事件系统。

17.9　设计上考虑的因素

由于消息系统是一种游戏对象和代码部分之间的非常直接和有效的通信方式，故它几乎能够在所有游戏中使用。它们可以构建快速且分离的系统，使其具有集中来回传递数据和事件的方法，而不需要复杂的严重耦合的类结构。

17.9.1　解决方案的类型

消息通常更有助于那些更高阶类型的解决方案(在战略相对于战术的意义上)。这是因为战略思考更多的是关于协调朝向同一个大目标的多个元素(分离的游戏实体或某个游戏实体的不同方面)，而这需要在分离的游戏元素之间进行大量的通信。另一方面，战术解决方案类型更多的是处理响应某个给定命令的物理动作，或确定处理一个简单任务的最好方式。然而，甚至是在战术级别中，消息也能为对战略系统的战术反馈(为了增加的战略响应)提供一个好方法。因此，消息能够自上而下进行应用(通过提供一种有效的方法来在许多元素上分布战略计划信息)，也能够自下而上进行应用(通过提供方法使得许多独立游戏元素可以向系统提供反馈)。

17.9.2　智能体的反应能力

事件驱动的智能体几乎都是反应性的，因为它们只是等待某事的发生，从而进行响应。使用正确的感知系统，任意级别的反应性都能够通过基于消息的智能体通信来实现。

17.9.3　系统的真实性

需要注意事件驱动的游戏可能的事件列表不能太窄，否则 AI 控制的角色将会表现得非常具有可预测性和静态性。这并不是说事件驱动系统本身是可预测的。但是一些游戏几乎完全由感知事件来驱动角色的实际行为，而这将急剧减少角色的丰富性。通常，事件驱动的角色也需要使用其他的技术(如脚本)，通过独占的丰富行为来对重要游戏事件进行响应。这只是消息如何与另外的 AI 技术结合以提供额外的现实性或游戏深度的一个例子。

17.9.4　游戏类型和平台

游戏类型和平台几乎不涉及到消息方法。然而，消息特别适用于那些或者具有大量需要通信的游戏对象的游戏(例如 RTS 游戏，它拥有大量被命令且发送回来持续反馈的军队)，

或者是在游戏对象之间具有非常丰富的交互的游戏(例如足球游戏，其两个或更多玩家对彼此的相对位置做出响应，参与相互间的复杂冲突，并处理游戏中的其他因素)。消息系统的确需要额外的存储器，但通常通过简化代码库来进行补偿，而且甚至能够在很小和受限的平台上实现。

17.9.5　开发限制

开发限制实际上将使得一个团队更倾向于基于消息的系统。它提供了惊人的鲁棒性，并能加速需要在类和游戏对象间协作的游戏特征的实现。基于消息的系统(如果适当设计的话)通常可以更快地进行编译和构建，因为该方法坚持对类进行解耦。调试系统可以直接并入消息流(或在对象层级上)中并为解决潜在问题和为回顾而记录行为日志提供了许多访问点。

17.9.6　娱乐限制

调整难度设置、平衡具体的行为以及其他娱乐上的考虑通常都与消息系统的使用无关，因此通常不能算是问题。

17.10　小结

消息系统提供了多种类型的决策范例，使得游戏对象之间以及必须同步动作的分离代码片断之间的通信变得非常灵活。

- AI 角色通常是反应性的，这非常有助于采用事件驱动的方法。
- AI 系统通常是高层的，这意味着它们能够访问游戏引擎的许多分离部分，因此它们能够执行那些必须的判断和行为。消息提供了一个简洁的接口，使得不需要访问全局类便可以实现这种访问。
- 本章的简单消息系统从 3 个部分进行设计：消息井、消息对象和客户端句柄。
- 该方法中的客户端句柄将像回调对象那样进行编码，以获取比更集中性的方法(比如一个一般的 ProcessMessage()函数)更大的灵活性和组织性。
- 对试验平台进行了设计，使得所有转换都根据事件而发生。尽管这并不是一个理想的消息示例，但它的确体现了该技术的力量，并提供了一个消息是如何使用的清晰例子。
- 当大多数游戏都利用基于消息的系统时，该系统可以工作得最好。调试等其他二级系统也能够使用消息引擎来减轻这些任务。
- 消息的确需要用额外的存储器来存储消息及附加数据，但负荷平衡和仲裁系统可以对必需的 CPU 和存储器进行优化。
- 增加设置消息优先级的能力、允许消息仲裁、往系统中增加自动或扩展的消息类型都可以扩展消息的使用，使其超越它最简单的实现。

18 脚本系统

到目前为止，我们主要致力于在代码层次上构造游戏逻辑和行为。这将使得定制的 AI 系统和游戏产品非常匹配，而且我们能通过优化得到相当出色的性能。但是正如我们所知道的，这种模式需要程序员花费相当的精力来实现、调试和扩展 AI 系统。而由于游戏产品本身的特点，程序员不可能作为监管整个产品创意开发的人员。另外，给现有的程序员灌输有关创意思维是一件有相当难度的事。更为普遍的是，游戏中所包含的创意的层次太高，以至于无法在程序员人数有限的情况下按照游戏要求的质量来完成代码的编写。

18.1 脚本概述

脚本(scripting)是一种无需雇佣更多的程序员，但却可以充分高效完成 AI 内核设计的通用技术。脚本模式意味着采用简化的程序设计语言来创建 AI 元素、游戏逻辑和行为(尽管脚本工具可以变成可视化的)。RPG(角色扮演类游戏)中的会话树、游戏故事序列中的场景动画和不同角色的出场画面、复杂的拳击类游戏中每个动作的细节或者多股敌对势力联合攻击的方式等都可以采用基于脚本的 AI 系统来完成，而且当前很多游戏也是这么做的。

脚本语言的范围很广，可以相当简单(这种脚本语言有时也称作预处理语言，因为它们大多只是包含一些一般性规则，比如关键字、值和某些表达式)，也可以是完整的程序设计语言(比如 Neverwinter Nights 中采用 Java 作为脚本语言)。

为游戏产品设计一种脚本语言不是一件简单的事情。从本质上说，这是在设计另外一个新产品，它有自己特定的用户(在该情况下特定的用户是指游戏的脚本程序员和游戏模型的终端设计人员)，输入输出需求以及设计、实现和调试计划，这些与软件系统设计师试图设计的游戏是独立的。在设计脚本语言时需要仔细考虑许多技术上的和创意上的因素。

- **语言应支持的功能类型**。游戏是否需要事件的线性触发，或者脚本语言是否需要支持条件分支？脚本语言是否需要支持变量？从本质上说，设计师是在定义所设计脚本的复杂度。下面以垂直射击游戏为例进行说明。在该游戏中，玩家操纵一架飞船在预先定义好长度的游戏空间中飞行，同时攻击沿途敌方飞船，这些敌方飞船以各种模式飞行。为该游戏设计的简单脚本语言只需支持脚本来定义各种类型敌人的出现位置和支持脚本在适当时机修改初始参数即可，敌方飞船行为的细节可以留在代码中进行处理。在稍微复杂点的系统中支持在脚本中定义敌方飞船

的活动模式和对不同的敌方飞船赋予不同的模式值。更复杂的脚本则需完全驱动敌船的创建：脚本将用一组属性(包括体型、速度、耐力和护甲层次等)和行为类型(包括攻击方式、运动类型和属性等)来定义每一个敌船，这些属性和行为类型来源于游戏引擎中的概率列表。最后，所实现的系统也允许敌船能访问游戏玩家操纵的飞船的数据，并按照脚本定义好的序列做出反应。这将支持脚本程序员编写反应系统以完成敌船对游戏中玩家或其盟友的状态做出反应的功能。

- **是运行时引擎解释脚本还是把脚本预编译成某种形式的字节码然后执行？**解释执行的脚本的运行速度相对慢些，需要的资源也多些。但解释执行能带来更多的灵活性(比如，可以通过游戏在线控制台输入新脚本，然后其能被正在运行的引擎重新解释)，同时它的脚本更便于使用，因为它不需要中间的编译步骤。在游戏机硬件平台中的 CPU 计算能力没有较大幅度地超出解释执行的需求时，对于到底是解释执行还是预编译的抉择是必要的。
- **脚本的大小。**程序代码一般都编译成较小的程序块，而脚本代码则不是(特别是解释执行的脚本)。设计人员必须关心开发游戏的目标平台是否有足够的内存空间，以存储游戏中的所有脚本。如果需要从硬盘读入脚本流，那么还必须关心存储带宽是否足够。
- **脚本语言的使用者。**如果一般程序员使用脚本，那么脚本语言需要设计得相对复杂和完全些。脚本语言的最初用户——创意设计人员是否有技术背景？如果他们严重缺乏技术背景，那么脚本系统需要足够简单以方便使用，需要足够健壮以支持非崩溃的错误处理，还必须包含相当数量的调试钩子。在设计脚本系统时的一个通病是为没有技术背景的脚本程序员设计一个过于复杂和强大的脚本系统。他们开始往往为该系统的强大而兴奋不已，但最后都没能充分发挥系统的能力。一个更好的设计思路是在开始时设计一个简单易用的系统，在脚本程序员熟悉了该系统后再逐渐加入更多的高级特征。在公开脚本语言(如果需要，还可以公开相应的工具)以支持大家更好地为游戏建模时，必须注意的是：不管怎么样，那些尝试和使用脚本语言的程序员，其平均水平是没有多少编程经验的。如果脚本语言遵循和现存的程序设计语言类似的某些规则，那么在学习过程中他们将会少走很多弯路(这也是现在业界存在很多类 C 脚本语言的主要原因)。另外可以在游戏中加入一些示例脚本，以演示如何完全地使用语言提供的功能。

18.2　Alsteroids 测试平台中的脚本实现

这一节我们将讨论如何在游戏中使用两种不同的脚本语言。首先，我们将实现一种简单的配置脚本语言，其将支持在线的游戏变量绑定和基于脚本标记(token)的行为。接下来，我们将通过讨论 Lua 语言、讨论怎么在游戏中嵌入 Lua 语言以及在游戏软件代码中通过什么方式向 Lua 语言提供函数和变量，来论述一种更加通用的功能完备的脚本方式。

18.2.1　一种配置脚本语言

下面要讨论的第一个脚本语言相对简单，程序员可以通过简单的语法使用关键字和参数。这个脚本语言没有变量、作用域、参数传递或者其他的高级语言的特征，但是它还是包括一些简单语言要素以支持规则转换、触发器以及一些简单的行为。我们应该感谢这种语言，因为它提供了通过脚本配置和初始化变量的形式化方法。可以通过向脚本公开所需的标签、数据和触发器，然后编写脚本完成所需的设置任务。

我们为游戏 AIsteroids 完成的工作可以分为 4 部分：解析器、标记列表、在线游戏标记回调机制和实际的脚本文件。

- 解析器是一组代码，它首先完成脚本文件的装载，接着扫描应用标记，最后执行所找到的标记。
- 标记包括标记名(脚本程序员借助它调用标记)、Execute()函数(解释器每发现一个标记就执行该标记的 Execute()函数)。在执行标记时，首先在文件中扫描标记所含的附加参数，然后发送包含该参数的消息给游戏软件引擎。
- 在线游戏标记回调机制是一组用于对 Execute()函数所发的消息做出反应的回调函数。通过这些回调函数才能真正地把脚本文件中的数据绑定到实际的游戏变量中。
- 我们可以使用任何文本编辑器来完成脚本的编写。在脚本中所采用的唯一的语法是每一行必须以分号结束。我们所实现的示例脚本(参考程序清单 18-1 test.txt 脚本文件)采用了通用的语法“Token=parameter;”，但是“=”实际上是标记名的一部分，因此读者可以采用任何字符来代替它。文件顶部的注释行并不一定需要注释符“//”，它们仅仅是为了脚本的可读性。事实上，如果注释中包含了真正的标记，它也将被找出来并被解析。

程序清单 18-1　test.txt 脚本文件

```
//don't need to put anything in a certain order,
//the parser ignores whitespace after the = sign,
//all lines begin with a token and end in a semicolon,
//and tokens are case insensitive
//all values in the script Override the default values
//that are set up in the Init() functions.

PowerupScanDist= 150.0;
SeekPowerups= true;
MaxSpeed= 80.0;
ApproachDist= 180.0;
AttackDist= 260.0;
SafeRadius= 15.0;
```

在初始化系统时，设计师应在解析器中注册试图在脚本文件中使用的标记(参见程序清单 18-2)。从代码中可以看到，每个标记都包含一个 ID、一个内部变量 m_matchPos(供解析器在扫描标记时使用)和一个用于供解析器识别标记的名称字符串。Get 函数作为标准的查

找函数，用于从脚本文件中提取不同的参数类型。如果游戏中还存在其他应用得相对较广的参数类型(其他的数据类型，或者复杂的数据结构)，也可以在此加入新方法用于装载这些参数类型。结构 enum 中存放了游戏中将使用的所有标记的 ID。

程序清单 18-2　标记头信息

```
class Token
{
public:

    enum
    {
        TT_NONE,
        TT_POWERUPDIST,
        TT_POWERUPSEEK,
        TT_APPROACHDIST,
        TT_ATTACKDIST,
        TT_MAXSPEED,
        TT_SAFERADIUS,
    };

    //constructor/functions
    Token(int type = TT_NONE, char* name = "")
         { m_tokenAsStr=new char(MIN(strlen(name),MAX_TOKEN_LENGTH));
          m_tokenAsStr = name;m_tokenID = type;m_matchPos = 0;}
    ~Token(){}
    virtual void Execute(_iobuf* fileName) {}

    //Additional data acquisition
    float GetFloat(_iobuf* fileName);
    char GetChar(_iobuf* fileName);
    int GetInt(_iobuf* fileName);
    bool GetBool(_iobuf* fileName);
    void GetString(_iobuf* fileName, string& storageStr);

    //data
    int m_tokenID;
    int m_matchPos;
    char* m_tokenAsStr;
};
```

解析器(程序清单 18-3 是其头信息，程序清单 18-4 是具体实现的几个重要函数)非常地简单。事实上，它简单得和普通的文本解释器差不多。如程序清单 18-2 所示，解析器只有一个数据结构(私有数据成员)：指向标记列表的指针，该数据结构在实例化解析器时完成初始化。这有助于利用解析器完成整个游戏脚本的解释；可以改变标记列表并重新解析该文件或者解析另一文件。在采用该脚本系统时，不同的游戏状态和角色等级以及不相连的场景将会有不同的标记列表。

程序清单 18-3 解析器头信息

```
class Parser
{
public:
    Parser(TokenList *tList = NULL):m_tokenList(tList){}
    int CheckForToken(char currentChar);
    bool ParseFile(char* fileNameStr);
    void Reset();
    Void SetTokenList(TokenList *tList){m_tokenList = tList;}

protected:
    TokenList* m_tokenList;
};
```

解析器是以字符为单位进行解析的。它每次从文件中读取一个字符，然后和标记列表中所有的标记进行比较。如果该字符和某个标记中当前位置上的字符匹配，则该标记的 m_matchPos 变量增加 1，使其指向列表中该标记的下一个字符，一旦发现某个标记的所有字符都匹配完毕，则重置列表中所有标记的 m_matchPos 变量并返回该标记的 ID。对于每个列表中的标记，如果在字符匹配中出现了不匹配的现象，则把该标记的 m_matchPos 重置。我们需要特别注意下面两点：

- **扫描标记的大小写不敏感性**。该特点将减少对具体问题进行查错时的麻烦，特别是对于没有技术背景的人来说。同时，我们应该注意到 Token.cpp 文件中的 GetBool() 函数对不同的字符串输入直接给出 true 或者 false 的判断，这也方便没有编程经验的人使用。
- **标记名的蕴含性**。如果一个标记名是另外的标记名的子集，就会发生标记名的冲突。假如某个脚本中有两个标记：“Shout=”和“Out=”。很显然，在解析该脚本时，我们将遇到标记名的冲突，这是由于“Out=”在两个字符串中同时出现。因此，在解析该脚本文件时，只有一个标记会得到执行，具体是哪个标记决定于它们在标记列表中的顺序。可以通过扩展脚本系统来避免冲突引起的问题(比如在解析脚本时先顺序扫描列表，记录下需要执行的标记然后批量执行，或者其他合适的方式)，当然也可以在标记命名时就避免该问题(比如可以和程序员约定标记命名规范或者引入函数 RegisterToken()，通过该函数来注册标记和传递标记列表给解析器，因为我们知道解析器本身不存储标记列表，只存储列表指针)，一旦出现了标记名的冲突就会报错。

程序清单 18-4 解析器的实现代码

```
//--------------------
int Parser::CheckForToken(char currentChar)
{
    TokenList::iterator tListiterator;
    for (tListiterator = m_tokenList.begin();
         tListiterator != m_tokenList.end(); ++tListiterator)
    {
```

```
            Token* pToken = *tListiterator;
            if (tolower(currentChar) ==
                tolower(pToken->m_tokenAsStr[pToken->m_matchPos]))
            {
                // if the currentChar matches the requested
                // character of the current token,...
                    // increase the "match-position" counter
                pToken->m_matchPos++;

                if (pToken->m_matchPos == strlen(pToken->m_tokenAsStr))
                {
                    // if the counter equals the length of the current
                    // token, we found a token. Thus,...
                    // ...reset the counters of all the
                          // other tokens and...
                    Reset();
                    // ...return the token found
                    return pToken->m_tokenID;
                }
            }
            else
            {
                // if the currentChar does *not* match the requested
                // character of the current token,...
                // reset the corresponding counter
                pToken->m_matchPos = 0;
            }
        }
        return NO_TOKEN;
}

//--------------------
void ZeroPosition(Token* pToken)
{
    pToken->m_matchPos = 0;
}

//--------------------
void Parser::Reset()
{
    for_each(m_tokenList.begin(),m_tokenList.end(),ZeroPosition);
}

//--------------------
bool Parser::ParseFile(char* fileNameStr)
{
    FILE* pFile;
    if ((pFile = fopen(fileNameStr, "r")) == NULL)
    {
        return false;
```

```
    }

    char buffer;
    Reset();
    while (fread(&buffer, 1, 1, pFile) == 1)
    {
        int currentToken = CheckForToken(buffer);
        if(currentToken == Token::TT_NONE)
            continue;
        else
        {
            TokenList::iterator tListiterator;
            for (tListiterator = m_tokenList.begin();
                 tListiterator!=m_tokenList.end();++tListiterator)
            {
                 Token* pToken = *tListiterator;
                 if(pToken->m_tokenID == currentToken)
                      pToken->Execute(pFile);
            }
        }
    }

    fclose(pFile);
      return true;
}
```

一旦发现标记，该标记的 Execute()就会被执行(程序清单 18-5 中给出了两个标记的执行方法)，该方法负责从文件中查找附加的参数，创建包含这些参数的消息并发送给引擎。方法的实现可以完全按照程序员自己的意思，也可以使用其他的数据结构。示例中采用的单个参数可以扩展成多个参数，用逗号把它们分开，而标记也可以是状态标志。“if”标记的执行方法可以是新的解析周期完成附加条件的查找。“if”标记拥有自己的条件标记列表，并根据查找到的附加条件来解析这些条件标记。“then”标记则进入这组标记执行的后半周期，停止附加条件的查找，开始扫描和解析这些后继标记。这种复杂的、嵌套的标记模式可以读入多行脚本，并通过单个“if”标记来触发。由于该解析系统的通用性，也可以用它来解析简单的脚本。

程序清单 18-5 一些 Execute 方法的实现

```
//--------------------
void TokenSafeRadius::Execute(_iobuf* fileName)
{
    float safeRad = GetFloat(fileName);

    //send out message with data of incoming token
    DataMessage<float>* newMsg = new DataMessage<float>
                    (MESSAGE_TOKEN_SAFERAD,safeRad);
    g_MessagePump.SendMessage(newMsg);
```

```
}

//--------------------
void TokenPowerupSeek::Execute(_iobuf* fileName)
{
    bool seekPowerups = GetBool(fileName);

    //send out message with data of incoming token
    DataMessage<bool>* newMsg = new DataMessage<bool>
                    (MESSAGE_TOKEN_POWSEEK,seekPowerups);
    g_MessagePump.SendMessage(newMsg);

}
```

18.2.2　配置脚本系统的AI性能分析

上述的简单脚本系统直接建立在基于消息的系统的基础上，除了初始化一些变量，没有任何其他的功能，因此事实上它纯粹是基于消息方式的系统。在游戏装载时读取和解析小文件，对系统的性能带来的影响几乎可以忽略，就算是包含相当数量标记的配置文件带来的影响对整个游戏的性能影响也不是很大。

配置脚本语法的扩充

该配置脚本语言非常开放和通用。从技术角度说，设计人员可以为游戏编写任何所需的高级标记，可以构造带任意多参数的标记，或者也可以用“//”来表示注释并停止解析脚本行。一个更复杂的扩展标记是驱动解析器进入特殊解析模式的标记。在该模式下，解析器将根据条件选择性地查找后继标记。该扩展标记大体上类似于常规程序设计语言的块(block)语法类型。比如，一旦发现“if”标记，C 编译器就会继续查找左括号(该处左括号用于表示条件表达式块的开始)，如果没有找到，编译器将会报告语法错误。C 编译器会在该表达式后查找单条语句或者是用于表示新的表达式块开始的“{”。从技术角度说，也可以通过建立块结构类型使得所设计的脚本语言支持高级组织结构。当然，如果脚本程序员想利用简单语法外的其他语法规则进行编程，在上述简化解析器中可能会引入错误，而这些错误很难被该解析器捕捉并处理。

18.2.3　游戏中Lua的嵌入

在本节中，我们将把 Lua 嵌入到游戏环境中，以代替我们自己创建的脚本语言。我们首先对 Lua 语言进行简单介绍，然后把重点放在 Lua 语言在游戏中的嵌入和怎样在 Lua 语言和 C/C++编程环境中传递消息。

1. Lua 语言概述

Lua 语言是一种轻量级的程序设计语言，它是由巴西 Pontifical Catholic 大学计算机图形技术小组的一个团队开发的。除了可以作为独立的程序设计语言外，Lua 语言还被很多

游戏开发人员用作通用脚本语言或者系统扩展语言。使用和进入 Lua 领域的原因很多。设计人员可能没有时间编写、调试和维护自定义系统。另外，所要设计的游戏可能存在很多对扩展性有很高要求的地方。在一般情况下，扩展性通过脚本来实现，因此设计人员需要一种通用的脚本语言，其能应用到游戏尽可能多的地方。在某些设计团队中可能存在一些员工对 Lua 语言有所了解，因为 Lua 语言已经面世一段时间了，已经在很多著名游戏中被采用(Baldur's Gate 和 Grim Fandango 是其中比较著名的两个)。

Lua 语言正逐渐成为主流嵌入式脚本语言，它能取代很多先前出现的嵌入式语言(比如 Python)的地位，主要的原因有以下几点：

- 与先前的嵌入式语言相比，它执行得更快，需要更少的内存资源，而且更易上手。
- Lua 的语法更加结构化，而且包含动态类型。
- 执行方式多样化，Lua 既可以采用解释执行的方式，也可以编译成字节码后执行。
- Lua 采用包含垃圾回收器的内存自动管理机制。
- Lua 很方便一般程序员和没有技术背景的人员学习，这是由于其采用了比较自由的 Pascal 语法(游戏开发者 2003 年年会上出现了这么一种观点：有经验的程序员能在一两个小时内学会使用 Lua 语言，而没用技术背景的人在面临一个简单 Lua 任务时可能需要稍长的时间，但是也能比较容易地上手)。

但 Lua 语言成为主流的嵌入式语言最主要的原因是其能在很多方面和其他语言兼容。Lua 语言通过一组简单易用的 API 来和游戏本身的数据交换，而不是采用大量语言本身的语法特征。如此，Lua 语言可以作为创建游戏专用语言(game-specific languages)的工具。设计人员可以通过创建一组函数来有效地支持游戏开发人员或者脚本程序员设计用于在游戏中产生动作的游戏专用脚本。不仅 Lua 语言语法本身很好学，而且采用 Lua 编写出来的代码简洁明了。

2. Lua 语言的基本语法概念

本节中将给出 Lua 语言的基本概念。我们不准备详尽地解释 Lua 语言，而是简单地给出 Lua 语言的主要概念。如果有读者需要有关 Lua 语言程序开发方面的更多资料，可以登录 http://lua-users.org/wiki/TutorialDirectory，上面几乎涵盖了 Lua 语言的所有方面。通过这些指导材料读者可以很容易学会 Lua 语言，Lua 语言解释器可以独立运行，也可以由用户在提示符下敲入命令启动后运行。如果读者有高级语言的程序开发经验，学习 Lua 语言的语法相当容易。程序清单 18-6 中给出了一些 Lua 示例代码，这些代码只是用于作说明，没有实际的作用。下面将给出一些基本的语法概念：

- **非常简单的作用域定义**。所有的 Lua 语句都是全局的。唯一限制作用域的手段就是在代码的子模块中给变量赋予局部状态(该模块通过控制结构与函数的其他部分分开)。
- **动态类型**。Lua 语言不需要为变量声明类型。Lua 语言只支持 7 种不同的数据类型，分别是：空类型(nil)、布尔类型(boolean，nil 的布尔值为假，而数字 0 和“”的布尔值为真)、数值类型(number，Lua 语言中的所有数据都是浮点的)、字符串类型(string)、表格类型(table)、函数类型(function)、线程类型(thread)和用户数据类型

(userdata)(用户数据类型是一种为存储任意 C 指针而设计的特殊类型，本质上它是 void*变量)。程序员可以把这些数据类型混合到令人惊讶的程度，特别是铁杆的 C++程序员。

- **表格**。表格是 Lua 语言中形式非常自由的数据类型。和 LISP 中的 list 类似，程序员可以把任意类型的组合放入表格中，而且表格的成员也可以是表格。本质上，表格中的所有成员都以标准模板库(STL)中的 map 的形式存储，每个成员有一个关键字和一个数值。简单的表格(比如 table = {1,2,3})称作数值索引(numerically indexed)，这是由于它们的关键字有点类似数组的下标。一个相对的非数值索引的表格定义如 table = {name = "Bob", number = "5551212", hometown = "Somewhere"}。在该表中，可以通过关键字名来访问其内容，比如 table. name == "Bob"。表格同样可以包含函数，可以认为其是面向对象的方法。
- **控制结构**。Lua 中提供了一系列的标准控制块，包括 do 循环，repeat...until...，if...then...else...elseif 和 for 循环语法块。所有的控制结构(除了 repeat...until...)都需要以 end 作为结束符。
- **堆栈**。Lua 通过堆栈和 C 程序相互传递数值。尽管我们在这里采用了术语堆栈，该数据结构其实不是堆栈。一般而言，堆栈只能通过压入和弹出来完成操作。Lua 的堆栈更像是一组带索引的寄存器，通过这些寄存器完成在脚本和程序之间的通信和数据交换。一旦在 Lua 中调用了 C 函数，将会自动地创建一个独立的堆栈用于传递数据。堆栈的默认大小为 20，该值是通过 lua.h 文件中的 LUA_ MINSTACK 宏定义来指定的。一般地，默认大小的堆栈够用了，除非需要在堆栈中压入很多的数据或者需要在 C 程序和脚本之间传递复杂结构。另外，堆栈的大小可以通过 lua_checkstack()函数来扩展。除了一般的堆栈压入和弹出操作外，Lua 还支持插入、删除和取代指定单元的操作，以更好地支持堆栈随机访问(单元的位置可以通过一个整数来指定，如果该数为负数，表示该单元相对于堆栈顶部的位置，如果该数为正数，表示该单元相对于堆栈底部的位置)。程序清单 18-6 的最后部分展示了怎样通过 C 程序操作堆栈的数值。

程序清单 18-6　简单 Lua 语法示例

```
-examples declaring different types
varNumber  = 5
varFloat    = 5.5
varFunction = function(i) return i-1 end
varNumber = varFunction(56)
varTable  = {1,false,6,8,{12,"string",7.99}}
v1,v2,v3  = 12,"apple",-5.6

-examples of control structures
index = 1
do
   index = 5
```

```
   print("Index = "..index)--should print 5
end
print("Index = "..index)--should print 1
--===========
index = 1
while index < 5 do
   print("Been here "..index.." times")
end
--===========
num = 1
repeat
   print(num)
   num = num * 3
until num > 100
--===========
function min(a,b)
   local minimum
   minimum = a
   if b < a then
         minimum = b
   end
   return minimum
end
--===========
if x == 3
   print("X equals 3")
elseif x < 1
   print("X is not 1")
else
   if x > 0
         print("X is positive, and less than 1")
   else
         print("X is negative")
   end
end
--===========
for index 1,50,3 do
   print("Loop value ="..index)
end

varTable = {name="marvin",look="monkey",job="ceo"}
for key,value in varTable do
   print(key,value)
end

--examples of table usage
table = { 23,44.5,18, color="blue", name="luxor" }
print(table[1])--will print 23
print(table[color])--will print blue
```

```
--------------------


//examples of stack usage, C code
//As an example, if the stack starts as 10 20 30 40 50*
//(from bottom to top; the '*' marks the top
lua_pushnumber(L, 10);// --> 10*
lua_pushnumber(L, 20);// --> 10 20*
lua_pushnumber(L, 30);// --> 10 20 30*
lua_pushnumber(L, 40);// --> 10 20 30 40*
lua_pushnumber(L, 50);// --> 10 20 30 40 50*


lua_pushvalue(L, 3);// --> 10 20 30 40 50 30*
lua_pushvalue(L, -1);// --> 10 20 30 40 50 30 30*
lua_remove(L, -3);// --> 10 20 30 40 30 30*
lua_remove(L, 6);// --> 10 20 30 40 30*
lua_insert(L, 1);// --> 30 10 20 30 40*
lua_insert(L, -1);// --> 30 10 20 30 40* (no effect)
lua_replace(L, 2);// --> 30 40 20 30*
lua_settop(L, -3);// --> 30 40*
lua_settop(L, 6);//--> 30 40 nil nil nil nil*
```

3. Lua脚本的集成方案

在游戏中集成Lua脚本是一件很容易的事。只需在连接生成游戏可执行文件时，导入Lua库文件，并完成Lua解释器的实例化。此时，用户可以通过输入Lua命令的方式或者通过装载并解释脚本文件的方式来在游戏中使用Lua。程序清单18-7给出了启动解释器的C代码。在第二个lua_open()函数后出现的一系列函数(包括lua_baselibopen()等4个函数)完成解释器中常用逻辑(包括输入输出、高级字符串函数和数学函数等)的初始化，lua_settop()函数用于完成堆栈中数据的复位和初始化，在Lua库文件完成堆栈创建后，堆栈中的数据是随机的。整个解释器后半部分的主要工作是完成在Lua和游戏代码之间注册函数和数值。本书中的代码采用了Lua的一个简单的扩展库——LuaPlus Call Dispatcher来完成注册过程，它由Joshua Jensen编写完成。LuaPlus Call Dispatcher的头文件提供了良好的用于注册C++函数代码和数据结构的函数模板，而且用户不需要考虑这些函数和数据结构是全局的还是某个类的成员，甚或是虚函数。一般地，任何需要在Lua中注册的C函数必须是静态函数，函数类型声明为static int Function(lua_state* ls)，同时所有的参数和返回值都将传入堆栈中。这是我们采用该库的理由。这些函数称为粘合函数(glue function，它们作为所需实现的C++函数和Lua脚本之间的中间层存在。LuaPlusCD通过提供非常简单的代码模板为我们提供粘合函数和处理堆栈操作，这些操作主要用于在Lua和C代码之间传递参数和返回值。程序清单18-8给出了在Lua和C++之间注册变量和代码的示例。

程序清单 18-7 简单解释器启动代码

```
#include "luaPlusCD.h"
extern "C"
{
    #include "lua.h"
    #include "lualib.h"
}

//and this code must be in an actual function

m_luaState = lua_open();
lua_baselibopen(m_luaState);
lua_iolibopen(m_luaState);
lua_strlibopen(m_luaState);
lua_mathlibopen(m_luaState);
lua_settop(m_luaState,0);
```

程序清单 18-8 在 Lua 和 C++之间注册变量和函数的示例

```
//from C++ to Lua
//--------------------
//variable data
int integerVariable = 42;
char stringVariable[] = "doughnut";

lua_pushnumber(m_luaState,integerVariable);
lua_setglobal(m_luaState,"intVar");
lua_pushstring(m_luaState,stringVariable);
lua_setglobal(m_luaState,"strVar");
////////////////////////////////////
//static functions using barebones Lua
//function takes a number argument,
//and returns 3*the number and 4*number
static int MyCFunction(lua_state* L)
{
    int numArgs = lua_gettop(L);//should be one
    float arg[numArgs];
    int i;
    for(i=0;i< numArgs;i++)
        arg[i] = lua_isnumber(L,1);
    for(i=0;i<numArgs;i++)
    {
        lua_pushnumber(L,arg1*3.0f);
        lua_pushnumber(L,arg1*4.0f);
    }
    return 2*numArgs;//number of results
}
lua_register(m_luaState,"MyCFunction",MyCFunction);
//Lua script can then say:
```

```
// a,b = MyCFunction(25)
//with results: a==75, b==100
//...or...
// a,b,c,d = MyCFunction(4,5)
//with results: a==12,b==16,c==15,d==20
////////////////////////////////////

//regular functor examples using LuaPlusCD
//(example taken from author's website)
static int LS_LOG(lua_State* L)
{
   printf("In static function\n");
   return 0;
}

class Logger
{
public:
   int LS_LOGMEMBER(lua_State* L)
   {
      printf("In member function. Message:%s\n",
                                          lua_tostring(L,1));
      return 0;
   }

   virtual int LS_LOGVIRTUAL(lua_State* L)
   {
      printf("In virtual member function\n");
      return 0;
   }
};

lua_pushstring(L, "LOG");
lua_pushfunctorclosure(L, LS_LOG, 0);
lua_settable(L, LUA_GLOBALSINDEX);

Logger logger;
lua_pushstring(L, "LOGMEMBER");
lua_pushfunctorclosure(L, logger, Logger::LS_LOGMEMBER, 0);
lua_settable(L, LUA_GLOBALSINDEX);

lua_pushstring(L, "LOGVIRTUAL");
lua_pushfunctorclosure(L, logger, Logger::LS_LOGVIRTUAL, 0);
lua_settable(L, LUA_GLOBALSINDEX);

//and the package can also set up direct calls, which are much
//more natural to C programmers...
void LOG(const char* message)
{
   printf("In global function: %s\n", message);
```

```
}


class Logger
{
public:
    void LOGMEMBER(const char* message)
    {
        printf("In member function: %s\n", message);
    }
    virtual void LOGVIRTUAL(const char* message)
    {
        printf("In virtual member function: %s\n", message);
    }
};


lua_pushstring(L, "LOG");
lua_pushdirectclosure(L, LOG, 0);
lua_settable(L, LUA_GLOBALSINDEX);

Logger logger;
lua_pushstring(L, "LOGMEMBER");
lua_pushdirectclosure(L, logger, Logger::LOGMEMBER, 0);
lua_settable(L, LUA_GLOBALSINDEX);

lua_pushstring(L, "LOGVIRTUAL");
lua_pushdirectclosure(L, logger, Logger::LOGVIRTUAL, 0);
lua_settable(L, LUA_GLOBALSINDEX);
////////////////////////////////////

//from Lua to C++
//--------------------

//variables
int intVar;
char strVar[20];
lua_getglobal(m_luaState,"intVarName");
intVar = lua_tonumber(lua_gettop(m_luaState));
lua_getglobal(m_luaState,"strVarName");
strVar = lua_tostring(lua_gettop(m_luaState));
////////////////////////////////////

//functions
//Lua function looks like:
//    function multiply(x,y)
//        return x*y
//    end

//C code would require:
```

```
float x = 123.0f;
float y = 55.0f;
lua_getglobal(m_luaState,"multiply");
lua_pushnumber(m_luaState,x);
lua_pushnumber(m_luaState,y);
float result = lua_tonumber(lua_call(m_luaState,2,1),-1);
```

18.3 Lua 在 Alsteroids 测试平台中的实现

为了能在测试平台中完成 Lua 的运行，我们需要完成一些额外工作。这些工作是在第 17 章所介绍的基于消息的系统(messaging-based system)的基础上完成的。事实上，我们需要向 Lua 脚本系统提供必要的感官数据，而 Lua 将利用这些数据进行适当的计算得到当前飞船的状态机的状态。程序清单 18-9 给出了为了能在基于消息的系统中采用 Lua 脚本需要对代码做的改动，程序清单 18-10 给出了对 AI ship 进行控制的 Lua 脚本示例。值得注意的是，基于这么一个事实——在躲避状态下，如果玩家飞船和敌方的飞船在一条线上，那么可以完成射击任务，在第二个脚本中飞船只需躲避和靠近两个状态。

程序清单 18-9　为支持 Lua 脚本对 MessAIControl 的改动

```
#include "luaPlusCD.h"
extern "C"
{
#include "lualib.h"
}

//---------------------
MessAIControl::MessAIControl(Ship* ship):
AIControl(ship)
{
    g_MessagePump.AddMessageToSystem(MESSAGE_SHIP_TOTAL_STOP);
    g_MessagePump.AddMessageToSystem(MESSAGE_CHANGE_STATE);

    //construct the state machine and add the necessary states
    m_machine = new MessMachine(MFSM_MACH_MAINSHIP,this);
    m_machine->AddState(new MStateApproach(this));
    m_machine->AddState(new MStateAttack(this));
    m_machine->AddState(new MStateEvade(this));
    m_machine->AddState(new MStateGetPowerup(this));
    MStateIdle* idle = new MStateIdle(this);
    m_machine->AddState(idle);
    m_machine->SetDefaultState(idle);
    m_machine->Reset();
    m_messReceiver = new MessageReceiver;

    //default values
    m_safetyRadius = SAFETYRADUIS;
```

```
    m_powerupScanDist = POWERUP_SCAN_DIST;
    m_maxSpeed = MAI_MAX_SPEED_TRY/Game.m_timeScale;
    m_appDist = MAPPROACH_DIST;
    m_attDist = MATTACK_DIST;
    m_powerupSeek = true;

    m_luaState = lua_open();
    lua_baselibopen(m_luaState);
    lua_settop(m_luaState,0);//clear the stack

    //bind const values to lua variables
    lua_pushnumber(m_luaState,MAX_SHOT_LEVEL);
    lua_setglobal(m_luaState,"gvMaxShotPower");
    lua_pushnumber(m_luaState,MFSM_STATE_APPROACH);
    lua_setglobal(m_luaState,"gsSTATEAPPROACH");
    lua_pushnumber(m_luaState,MFSM_STATE_ATTACK);
    lua_setglobal(m_luaState,"gsSTATEATTACK");
    lua_pushnumber(m_luaState,MFSM_STATE_EVADE);
    lua_setglobal(m_luaState,"gsSTATEEVADE");
    lua_pushnumber(m_luaState,MFSM_STATE_GETPOWERUP);
    lua_setglobal(m_luaState,"gsSTATEGETPOWERUP");
    lua_pushnumber(m_luaState,MFSM_STATE_IDLE);
    lua_setglobal(m_luaState,"gsSTATEIDLE");

    //bind state change function for lua to use
    lua_pushstring(m_luaState,"ChangeState");
    lua_pushdirectclosure(m_luaState,*this,
       &MessAIControl::SetMachineGoalState,0);
    lua_settable(m_luaState,LUA_GLOBALSINDEX);
}
//--------------------
void MessAIControl::Update(float dt)
{
    if(!m_ship)
    {
       m_machine->Reset();
       return;
    }

    UpdatePerceptions(dt);
    //update exposed lua variables
    lua_pushnumber(m_luaState,m_nearestPowerupDist);
    lua_setglobal(m_luaState,"gvDistPowerup");

    lua_pushnumber(m_luaState,m_nearestAsteroidDist);
    lua_setglobal(m_luaState,"gvDistAsteroid");

    lua_pushboolean(m_luaState,m_willCollide);
    lua_setglobal(m_luaState,"gvWillCollide");
```

```
    lua_pushboolean(m_luaState,m_isPowerup);
    lua_setglobal(m_luaState,"gvIsPowerup");

    lua_pushboolean(m_luaState,m_isAsteroid);
    lua_setglobal(m_luaState,"gvIsAsteroid");

    lua_pushnumber(m_luaState,m_ship->GetShotLevel());
    lua_setglobal(m_luaState,"gvShotPower");

    //run lua script, which handles state transitions
    lua_dofile(m_luaState,"script1.lua");

    m_machine->UpdateMachine(dt);
}
```

程序清单 18-10　飞船控制 Lua 脚本示例

```
---Lua script for simple asteroids state Logic


    if gvWillCollide then
        ChangeState(gsSTATEEVADE)
    elseif gvIsPowerup and gvShotPower < gvMaxShotPower then
        ChangeState(gsSTATEGETPOWERUP)
    elseif gvIsAsteroid then
        if gvDistAsteroid < 200 then
            ChangeState(gsSTATEATTACK)
        else
            ChangeState(gsSTATEAPPROACH)
        end
    else
        ChangeState(gsSTATEIDLE)
    end

-----Another asteroids Lua script
    if gvWillCollide then
        ChangeState(gsSTATEEVADE)
    else
        ChangeState(gsSTATEAPPROACH)
    end
```

Lua 脚本用于处理状态机的状态转移。由于该测试平台通过调用 lua_dofile()函数来完成脚本的执行，所以我们可以在不关闭游戏的情况下改变 AI 行为。用户修改 script1.lua 文件并保存后，在下一个游戏节拍该脚本文件就会被载入并执行，脚本文件可以通过文本编辑器来修改。但是在真正的游戏中，我们一般不希望在游戏运行过程中经常性地访问磁盘。针对该问题，Lua 为我们提供了解决方案：可以先把 Lua 脚本载入到缓存中，然后执行，而不是通过访问磁盘文件的形式来执行。这并不妨碍我们快速改变脚本行为，我们可以通过对缓存进行重载并执行来完成。当然，另外也存在折中的方案：在产品研发和测试阶段，采用文件随机访问的形式；而在出售的最终产品中，采用缓存的形式。

在该Lua脚本中，把所有的状态转移逻辑都囊括在脚本的一个if...then...else子句里。在大型设计里，这看起来像设计上的退步，但在这里却是明智并且规范的设计。这是由于测试平台的状态机足够简单。事实上，在进行C++程序设计时，我们也是这么做的，而且程序运行得也更快。但是在大规模和复杂的游戏系统中，该方法显然不合适。

在实际游戏中，游戏设计者都希望在游戏端使用极其简单的有限自动机(FSM)，游戏的所有逻辑都通过数据来驱动。游戏端的状态机将主要包括状态列表、状态转移和动作所需的输入数据块(这些数据块对Lua脚本是可见的)。同时，游戏端代码还包括动作列表，动作列表包括所有的游戏世界中的行为和表现。而脚本文件也是按照游戏状态通过Lua函数的形式组织起来的。每个游戏状态的方法包括行为方法调用和状态迁移逻辑。

游戏引擎调用Lua脚本时，首先将更新Lua脚本中存放当前游戏状态名的全局变量，该变量用来关联处理该状态的函数。脚本程序员可以通过编写新函数并在全局状态表中加入其所编写的函数或状态名来完成游戏中新状态的引入，值得注意的是全局状态表对游戏端代码是可见的。游戏被加载(或者重载)时，全局状态表将会被实时地读取并创建简单状态机。程序清单18-11给出了一个简单的C++示例代码和相关的Lua脚本。

程序清单18-11 一种更好的Lua控制的状态迁移方案

```
//FSM Game code
LuaPerceptionExport();
UpdatePerceptions();

lua_pushnum(m_luaState,m_currentState);
lua_setglobal(m_luaState,"gCurrentState");
lua_doFile(m_luaState,"transitions.lua");

UpdateMachine();

-------------------------
--example.lua

--game state functions
function gsStateStand()
   --start/stop behaviors based on Perception data

   --check for transitions
   --would call a ChangeState() function, which would
   -- change the C++ m_currentState variable
end

function gsStateRun()
   --do run state
end

function gsStateSit()
   --do sit state
end
```

```
--global table of functions, C code can
--access this in order to find out the number of
--game states in the system, and their order
funcs = {gsStateStand,gsStateRun,gsStateSit}

--executes the current state function
funcs[gCurrentState]()
```

和上述系统类似的脚本系统只需要脚本程序员定义所有可能的状态，任何的增加和删除动作都和软件程序员没有关系。这样做的结果是把游戏分成两大部分：感知和行为在代码部分完成，而逻辑和参数(比如某些用于调节和平衡游戏角色的属性和数值)的配置在脚本中完成。脚本程序员可以任意地创建游戏状态以适应逻辑树的需求，他们对程序员的唯一需求就是程序员需给出游戏代码中得到的感知数据和行为列表。

18.4　Lua 脚本系统的 AI 性能分析

我们的脚本非常简短，因此脚本文件的执行所引起的性能损失和定时解释脚本所引起的性能损失是可以忽略的。该脚本系统能和 AIsteroids 很协调地运行。考虑到可以在游戏运行时编辑游戏逻辑，该脚本系统具有很强的调试性和修改性。

但是，上节给出的脚本系统并不太适合扩展应用于大型游戏。考虑在游戏中存在成千上百的状态和不同的角色时的情况。解释器需要不断地解释庞大的文件并不停地遍历 if 子句。这并不是我们想要的系统。性能将非常糟糕，调试也将是个噩梦，而系统的扩展几乎不可能。因此，我们需要分割系统并采用模块化的方式组织系统，尽可能早地细化系统。

为了使 Lua 能在更大型或者采用了多线程环境的游戏中更好地应用，我们需要引入更高级和有效的概念：多线程(threads)和协同例程(coroutines)。多线程技术引入了完全独立的多个 Lua 状态环境，而协同例程只是可以任意暂停和唤起的可重入函数。有了协同例程和良好的编程习惯，我们可以创建大量不同的脚本函数，用于执行游戏世界里不同的 AI 实体，而不需考虑某个脚本函数可能会空占所有的 CPU 资源。

18.5　脚本系统的优点

在 AI 引擎中使用脚本系统意味着非技术人员也能创建和扩展游戏逻辑、调试系统和 AI 行为，甚至完全改变整个 AI 系统(如果游戏引擎是完全由数据来驱动的)。脚本系统的优点主要包括快速原型开发(rapid prototyping)、更低的门槛、更快的调试速度、更好的用户扩展性和更广的适用范围。

18.5.1　快速原型开发

任何时候想把游戏的感知和行为部分抽象到更高的层次上(比如在决定给 Lua 脚本提供什么变量和函数用于和宿主代码交互时)，设计人员只需要从游戏中提取出其中最有意义

的概念。一旦这些概念提供给了脚本程序员，他们很快就可以通过脚本把其中的高级逻辑和行为序列表达出来，同时完成游戏设置和事件的在线调试。借助于开发脚本时的快速迭代周期(通过脚本的在线重载)，可以很容易地在游戏中加入新内容。

18.5.2　更低的门槛

借助于专用高级语言，可以消除采用真正程序设计语言进行开发时会遇到的障碍，因此如果愿意，将会有更多的人可以很容易地入门。可以想象，当人们发现汽车各部分所涉及的原理非常简单和直观时，那么将会有更多人在汽车出现问题后自己修理。通过设计并提供方便易用的脚本系统，我们可以吸引更多设计人员，甚至终端用户参与到脚本的扩展和改进中来。

18.5.3　更快的 AI 调试速度

脚本系统提供了比其他方式(比如代码修改、编译、链接和游戏重启)更多的调试 AI 行为的便利。同时脚本系统提供了开放的手段来支持更多的人同时进行调试工作。

18.5.4　更多的用户扩展手段

用于进行 AI 和游戏内容编码的脚本系统可以包含在产品发行版本中并提供给用户。很多公司在它们的游戏产品中包含了几乎所有的开发套件。特别地，该做法已经成为第一/三人称射击类游戏的标准做法。在这类游戏中，优秀的用户自定义游戏模式可以把游戏的热卖时间从几个星期或者几个月延长到几年。这些高度开放的游戏所包含的脚本语言(比如 QuakeC 或者 UnRealScript)的复杂度可与真正意义上的程序设计语言相媲美，脚本系统还允许终端用户自己控制、改变和创建游戏的各种效果。通过扩展使用脚本语言，终端用户可以自己创建第一/三人称射击类游戏，比如飞行类和赛车类游戏。

18.5.5　更广的适用范围

随着控制对象的数目不断增大，脚本系统的真正能力也不断凸显。利用数据驱动模式后，脚本系统所附带的开销将越来越少。尽管在不同的游戏系统中脚本的作用可能不同(在不同的游戏中，可能需要向脚本系统注册不同的函数列表供其使用，在脚本中也将有不同的相对应的函数类型来完成不同状态逻辑或者场景序列)，但是如果脚本平台足够开放和灵活，那么可以将其作为基础应用到这些不同类型的游戏中。

18.6　脚本系统的缺点

脚本系统确实也存在一些缺点，但一般这些缺点都可以通过认真筹划和仔细设计来克服，同时当前 PC 和游戏机所提供的巨大处理能力对克服这些缺点也很有帮助。在设计脚本系统时，我们需要考虑的问题主要有执行速度、调试难度、脚本作用、脚本和宿主代码的功能划分以及所需维护的系统数量。

18.6.1　执行速度

任何解释执行的程序比编译成本地机器代码的程序执行得慢。Lua 脚本可以预编译成字节码，这会稍微提高一点 Lua 脚本的执行速度，同时降低 Lua 脚本的可读性，如果设计人员不想用户看懂其所设计脚本的细节，那么字节码模式是个好主意。执行速度也是很多人致力于提高脚本和通用程序设计语言的集成度的主要原因，集成度越高，执行速度越快。在进行游戏设计时，一般会考虑不同部分的速度敏感度，需要比较快速的部分，我们一般用通用程序设计语言来完成，而对速度相对不敏感的部分，我们可以采用脚本语言来完成。值得一提的是，Lua 的执行速度比其他的脚本语言更快，这主要是归功于其简练的语法结构和清晰的编码风格。

18.6.2　调试难度

脚本系统被质疑的主要方面是调试。这里说的调试和前文的 AI 调试是不同的，主要是代码本身的调试，而不是 AI 系统各参数的调试。主要质疑可以分成两个独立的方面。第一个方面的问题是：在脚本语言(特别是用户自定义的脚本语言)的编程环境中没有提供通用程序设计语言都提供的调试工具链(包括调试器、配置器、断言、内建错误检查和编译时的语法检查等)，而为了调试脚本系统，用户需要在脚本代码中插入额外代码。第二个方面的问题是：利用脚本语言进行编程的人员一般不具备程序员的技术思维(这点是否准确？)，因此他们缺乏调试应具备的基本常识。这些技术主要包括：二进制代码的除错，通过打印语句在线检测变量，在脚本中加入错误检查，甚至于简单的逻辑技巧也不一定被创意人员所掌握。通常，脚本程序员需要不断地学习这些技术(就像程序员一样)。因此那些学习层次分明的语言更适合于脚本程序员。Lua、Python 和其他一些轻量而又完全的脚本语言提供了分明的学习层次。我们可以很快地学会编写简单脚本，同时学习更高级的语言要素。脚本程序员可以经过不断地学习，设计出高效的脚本。

18.6.3　脚本作用

游戏脚本曾经一度发挥着类似于好莱坞电影脚本的作用，用于描述大段游戏场景。比如下面提到的场景。当进入某房间后，玩家立刻丧失了游戏控制权，游戏场面的视角不断变化，跟着在门口出现了 3 个怪兽。它们按照既定位置排列，为首的怪兽将慢慢靠近玩家并痛骂他，然后告诉他它将怎么杀死他。此时，整个房间将弥漫着烟雾，同时前个游戏场景中得到的怪异帽子开始升起。所有这一切看起来很不错，至少是在前几次看到该场景时。但如果由于该场战斗难度较高造成玩家需要多次重试该游戏场景或者这些类似的场景多次出现时，玩家将变得非常地恼火。此时，他会认为他不是在玩游戏，而是被牵着鼻子经历一个个场景。有些游戏能很好地处理场景的取舍，因此它们得到了玩家的肯定。更多的游戏没能很好处理，因此玩家觉得有点像是傻坐着并经历漫长的、毫无交互的、无聊的折磨。真正有效的脚本不应该包含这些冗长的游戏场景序列。想象一下这样的场景，游戏中玩家和吧台的醉汉交谈，该醉汉打下饱嗝，跌倒并叫着“滚开，让我一个人呆着！”，然后他爬起来并坐回椅子上，这一过程不需要和玩家有任何的交互。如果玩家相继和该醉汉交谈了 3 次，那么他很快就可以知道该醉汉的行为是由脚本控制的。尽管该场景可能是由一小段

程序代码而不是脚本来控制，但它还是会受到玩家的谴责。脚本控制的内容越丰富，第一次出现时的效果越好。但在第二次出现时，所有这一切的美好都被破坏，玩家也就能很容易地发现是脚本在作祟。消除这样的脚本带来的影响的最好办法是不采用脚本。但如果某些游戏确实需要前面提到的这类游戏场景，则只要满足下列条件，脚本还是可以采用的：该脚本不过分渲染场景或者当玩家在游戏中遇到困难需要对某个战斗场面进行重试(此时脚本可能会多次执行)时，脚本不再在场景的渲染上对他们进行折磨。

18.6.4 宿主代码和脚本的功能划分

数据驱动 AI 系统的一个主要问题是怎么决定 AI 脚本的作用边界。人们一般都是基于状态转移进行脚本设计的。但这样做时，我们需要注意在系统稍微扩展后，就把整个状态机的定义和设计都交给了脚本程序员。从技术角度上说，在另加代码后，脚本程序员就能定义大部分 AI 行为。如果再加函数到脚本中，他们就能定义游戏的感知，包括用于计算的等式和更新频率等。问题是：如果在脚本中完成很少的功能，脚本给游戏带来的好处就不足以抵消其给游戏带来的开销，如果让脚本完成更多功能，可能会使脚本程序员有一种需要独立设计整个游戏的压力，而且所设计的游戏的执行速度也将只有原来的 3/4。这样做还带来了另外的风险——给脚本程序员如此多影响游戏逻辑的能力，而他们并不能起这么大的作用，除了不断地引入错误或者给出相互矛盾的逻辑并拖延项目的进度。怎么划分脚本和软件程序之间的逻辑功能是在设计脚本系统时需认真考虑的问题。问题的答案依赖于所设计的游戏的类型，依赖于预期脚本程序员完成什么样的工作，依赖于预期怎么安排和控制游戏内容。

18.6.5 需维护的系统数量

在决定编写基于脚本的 AI 系统后，意味着游戏系统设计师在完成游戏产品本身外还需要提供一个和游戏本身完全独立的产品。因此，系统设计师有两方面的工作要完成。一方面，需要完成游戏产品的开发，这部分工作的目标客户是游戏玩家，他们对游戏的视觉、感官和可玩性有着自己的需求和期望；另一方面，系统设计师需要设计一种轻量的程序设计语言产品，该产品有着完全不同的目标用户(尽管有时该产品也会提供给终端用户)，该用户对脚本系统应该如何工作也有着自己的要求。当决定把脚本引擎集成到 AI 系统中时，系统设计师必须站在脚本程序员的角度考虑问题，因为一个脚本系统的好坏在很大程度上取决于其是否更便于脚本程序员使用，以及其是否更便于系统设计师向脚本程序员传授脚本的使用方法。系统设计师至少应向脚本程序员提供一些脚本示例(这些示例需要编写得尽可能优质，这是因为其不仅会作为脚本程序员的学习指南，也可能会部分被他们修改后直接用在游戏脚本中)和一些常见问题的解答，这些问题覆盖从基本调试技术到简单函数方法的设计等各个方面。基本逻辑、函数组织方式、循环和数据的不同表示风格也需要包含在内。另外，没有编程背景的员工对脚本作用的期望也是日积月累的。系统设计师提供给他们利用脚本进行游戏修改的技术越多，他们期望用于修改游戏的脚本技术就越多，因此他们希望系统设计师在脚本语言中加入更多的语法要素。系统设计师必须为未来系统的扩展提供灵活性。但还应该记住脚本语言中提供的功能越多，带来的问题也越多。

18.7　范例扩展

脚本系统旋风式地出现了很多变种，所有这些变种的出现都很自然。因此，对脚本功能的扩展有很多的选择，采用哪种完全取决于系统设计师和脚本程序员的需求。脚本语言的高级概念有以下一些：自定义语言、内建调试工具、智能脚本 IDE、游戏脚本自动集成和自主修改脚本等。

18.7.1　自定义语言

游戏脚本语言设计的另外一种办法是经典的“Lex & Yacc”流程。该流程出现时主要用于定制编译器，但其同样可用于高效地创建自定义脚本语言。该过程相当地直观，首先，设计师需要在 Lex 工具的帮助下给出语法文件，用于确定所设计语言的各个词素的细节。Lex 允许设计师通过一种特殊的规则——上下文自由语法(context-free grammar)来建立语言的规则。上下文自由语法意味着这些语法规则可以包含通配符和嵌套定义。然后设计师在该语法文件上运行 Yacc(Yacc 全称 Yet Another Compiler Compiler)，它将生成用于解释所定义语言的 C 代码，该代码既包括解释字节码的解释器，也可以包括解释源码的解释器。游戏需要在游戏解释器中包含 Yacc。该过程有时被称为“即时编译”(Just-In-Time)技术，即时编译意味着脚本在将近执行前完成编译然后执行。通过使用该工具链，自定义的语言可以提供和通用编译器同样的灵活性和解释能力，而且完全不需要设计师为设计该语言的解释器而抓狂。通过该过程设计的脚本语言也可以使用复杂结构、不同的操作数和关键字。有了这些语法要素，接下来的游戏设计工作就能很好地开展了。

18.7.2　内建调试工具

作为调试手段，内建调试工具在复杂大型系统中具有举足轻重的作用。在脚本系统中直接嵌入调试函数或者游戏程序端嵌入即时调用系统将给脚本程序员调试游戏程序或者脚本中的错误提供极大的便利。简单工具，比如“监视”(watch)(一种对变量进行实时监控的技术)、断点语言(break statement)(一种用于在指定位置中断程序运行的技术，该位置称为断点)和脚本的单步调试(一种每触发一次只执行一行脚本的技术)，可以使得脚本的调试像常规程序的调试一样得到加速。另外，游戏在线调试中常见的调试功能可视化也是相当有意义的。脚本程序员应该能够在脚本中加入向游戏界面输出脚本状态的语句，输出的形式也可以是文本、图标或者其他任何可以描述脚本的信息类型。通过这些信息可以帮助脚本程序员调试脚本程序。所有这些内建调试手段在游戏产品发布时可以忽略，要么通过脚本注释手段注释掉，要么在游戏产品的脚本中直接删除掉。

18.7.3　智能脚本 IDE

在编码时，出现的愚蠢错误有时候很难发现，尤其是没有程序设计背景的人。他们往往会盯着“heroHitPoints = 0”这么一条判断语句很长时间，而不知道应该把此处的“=”改为“==”，还有的人不知道函数调用需要以大写字母开头。智能 IDE 或者某些文本编辑器可以提供实时语法检查(通过语法检查解释器)、关键字补齐(程序员只需要输入前面若干

字符，后面部分 IDE 会补上)和拼写检查等。这些技术可以减少程序员在编码时会犯的低级错误，并增加脚本程序的可读性和可用性。

18.7.4 游戏脚本自动集成

脚本系统为设计人员在游戏中增加新内容提供了便利。当然脚本系统在提供便利的同时，也给游戏软件设计增加了一定的工作量，软件程序员需要在游戏软件中为加入新内容提供支持。但是，正如前面章节表述的那样，设计人员可以通过工具来定义游戏中被脚本和游戏引擎同时使用的数据结构。如此，软件引擎同样可以读取这些数据并重建整个系统的对象集，而这一切不需额外的工作量来支持头信息的处理。如果在增加新内容时，有程序员参与，那么我们还有另外的办法，仅仅把脚本系统看作是程序员的反馈工具，就像请求系统一样。一旦需要一个新插件(不管采用该插件的意图是作为修改过的特殊文件还是脚本中的 RequestedWidget 命令中的一部分)，系统将把该插件加入到程序员能访问的特殊列表以确定脚本需要加入什么样的新功能。该系统允许脚本程序员申请一个新的功能并给出简要的描述，简要描述的目的是避免请求让人看不懂。在收到请求后，程序员才能实现该请求插件，并加入到游戏代码中。那么脚本下一次启动时，该请求的插件就能被系统识别为已完成，并继续执行。

18.7.5 自主修改脚本

自主修改脚本为脚本系统在 AI 游戏中的使用打开了大门。在原先的完全采用常规程序设计语言编写的游戏中，这一切是不可能的。我们没有理由由于脚本系统只能记录某些特定行为是否起作用就对其产生偏见。脚本系统可以加入或者删除特定规则作为游戏中某些事件的后果。类似的行为可以看作是机器学习，甚至更进一步。事实上，在 AI 领域存在一个分支，称为遗传程序设计(genetic programming)，专门负责研究上述现象。和遗传算法不同，遗传算法主要把生物遗传进化引入到算法本身中用于解决具体问题，遗传程序设计是借助遗传算法来更好地编写解决某个问题的程序。因此，该方法在设计空间上寻找更好地完成指定功能的脚本代码的设计方案，而不是寻找特定脚本的某个参数的理想值。借助遗传算法寻找最优脚本代码的主要问题是经常性地得到没有实际作用的脚本代码，这些代码根本不需要解释，更不用说是执行了，但是在有了更抽象的和更高层次的脚本系统后，我们就有可能完成脚本的产生、测试并最后得到能在 AI 系统中执行的更好的代码。显然，这是一条困难的而又费劲的路线，但最终我们将获得胜利。

18.8 优化

给定脚本语言的性能在很大程度上取决于设计人员怎么在游戏中使用它、语言所提供的功能和设计人员怎么组织整个系统。本章前面部分所实现的简单配置脚本系统几乎等同于文件解释器，该脚本对于游戏的运行基本上不起作用(这是由于在设计时配置脚本的主要目的是在游戏载入时设置变量和标志信息，而不是在游戏运行时设置)。如果想在该脚本语言的基础上开发实用的脚本系统，程序员需要相当强的文件处理技巧。我们不提倡这么做，

因为配置脚本语言甚至不具备最简单的脚本所应有的特点。Lua 是一种相当轻量的语言，而且相对于其他解释执行的语言，其执行速度还是非常快的，但是程序员依然不会采用Lua设计 AI 系统中的探路者，因为探路者需要在脚本和代码之间来回传递大量的搜索循环和数据。

那些只在脚本中对某些事件做出 AI 反应的脚本系统比那些试图在脚本中完成几乎所有 AI 计算的脚本系统要快一些，这是由于前者脚本中只包含需要的动作序列，而后者需要在脚本中完成计算，必然需要在脚本和代码之间传递额外的数据。

处理大型脚本系统时，设计人员将会遇到性能问题，这是由于这些脚本中包含了大量的逻辑和游戏代码端的函数调用。当发现 AI 需要超额的计算时间后，设计人员应该考虑采用线程和协同例程，但是要注意只有在占用大量 CPU 时间的任务是可重入时，也就是说它们可以通过游戏循环解决时，才能使用线程和协同例程等功能。

18.9　设计上考虑的因素

脚本系统在很多类型的游戏中应用，比如很多早期的游戏和现代游戏。早期游戏依赖大量的游戏模式占领市场，而现代游戏则更多地是依靠丰富详尽的内容取胜。脚本系统支持没有程序设计背景的人采用快速安全的方式设计游戏内容，这基本可以消除程序员设计游戏内容时带来的时间上和功能上的瓶颈。这些系统同时支持快速调试，因为它们支持在不修改游戏代码并重新编译的前提下完成游戏内容的修改。

18.9.1　解决方案的类型

脚本系统在底层 AI 应用中能很好地起作用，同样，它也能很好地应用于高级 AI 中。在底层 AI 应用中，脚本系统主要完成游戏行为的描述、动画的选择、游戏变量的简单设置，所有这些都将对游戏起到本质的影响，而在高层 AI 应用中，脚本系统主要完成大量游戏单元的策略选择和大量行为序列的描述。脚本系统比软件代码更适合高层 AI 的描述，脚本系统对游戏世界的描述更抽象，虽然有时脚本系统也需要处理一些游戏的底层部分，比如前面所述的游戏画面的选择。

18.9.2　智能体的反应能力

脚本系统能提供客户所需的不同层次的响应时间，虽然这还取决于客户所选用的脚本语言。如果某脚本系统描述的行为数量有限或者响应时间很长，那么该系统会被认为过度使用脚本。但是如果脚本系统仅仅用于描述基于状态的 AI 系统，那么智能体的响应时间应能满足系统的更新频率的要求。

18.9.3　系统的真实性

考虑游戏系统的真实性时，采用脚本系统实现的 AI 单元能针对当前游戏状态做出最真实和独特的反应。问题是脚本系统所描述的行为特点越丰富，所需要的属性就越多，因此脚本系统在游戏中的应用还是很有限的。如果所设计的游戏完全是状态驱动的反射系统，

那么游戏必然包括大量的属性脚本，使得游戏中的每个反应和动作更加真实和丰富。这样的系统要么看起来和其他部分格格不入，要么使得游戏其他部分的效果更加不能接受。因此，不得不使用大量的脚本属性使得游戏的每部分的效果都看起来不错。另外，玩家在某个地方不停地重复场景时，对于过分依赖脚本的系统，每次响应场景必须不同，这样才不会让玩家觉得游戏内容机械化。

18.9.4 游戏类型和平台

游戏类型和平台不是脚本系统需要关注的问题。脚本系统很适合包含大量 AI 实体和场景的游戏，因为这些游戏里存在太多的 AI 实体，而让程序员直接或者间接来平衡各个实体的关系是不实现的。如果游戏只是包含一个主要角色和一个敌人，并且每个实体只有3 个属性，那么设计人员可以只用软件代码来实现整个游戏，并很好地平衡各个角色的关系。当然，程序员也可以使用配置脚本系统来完成参数和属性的设置，如此调节参数时不需要重新编译游戏。脚本系统确实需要额外的存储资源，这主要是解释器和数据的开销，但可以通过简化基本代码来弥补，如此脚本系统也能适用于更小型和条件更苛刻的平台。除非系统进行了精心设计，在采用脚本系统时我们必须考虑性能要求是否能满足，毕竟脚本系统的运行速度比编译过的代码运行起来要慢。

18.9.5 开发限制

游戏开发上的限制是设计人员在做出是否在 AI 引擎中使用脚本系统的决定时需考虑的主要问题之一。设计师需要决定是否有时间设计脚本语言，是否有时间培训脚本程序员和是否能承担调试脚本和游戏代码所带来的附加工作量，另外，得到最终产品时所节省的时间不一定能弥补这些工作所带来的成本负担。

18.9.6 娱乐限制

不同游戏参数的协调性、不同游戏行为的平衡性和其他娱乐相关问题是一个团队选择脚本系统的真正理由。因为这些问题要么对于产品本身很重要，要么抽象层次太高，而不管是简单脚本语言还是功能完整的脚本语言，对于处理这些问题都是有帮助的。

18.10 小结

脚本系统为游戏开发人员提供了一种增加游戏内容的手段，该手段不需要以增加更多程序员为代价。脚本系统采用简化的程序设计语言来创建 AI 单元逻辑和行为。脚本语言可以是简单的基于标记的语言，也可以是功能完整的程序设计语言。

- 在设计脚本语言时，需要考虑脚本的功能、运行引擎解释脚本的方式、脚本文件的大小和语言的潜在用户。
- 配置脚本语言是一种简单的、用于解释用户定义标记的文本解释系统。
- 本章涉及的简单配置脚本包括 3 部分：标记、解释器和一组回调函数，这些回调函数和标记一一对应。

- 由于配置脚本语言本身的特点，我们可以对其进行任意扩展。但是由于配置脚本语言不具备健壮语言所应有的特点，很少有人会在配置脚本语言的基础上进行投入和开发。
- 通过把 Lua 作为脚本语言嵌入到系统中以取代自主开发的脚本语言是一种很好的选择，这种方法已经为越来越多的设计人员所采用。Lua 具有小型化、速度快和简单易学等特点，而且可与 C/C++很方便地集成。
- Lua 的主要语法要素包括：动态类型、垃圾回收、组相联函数调用表和用于在 Lua 和宿主语言之间传递参数的随机访问堆栈。
- 为了在游戏环境中集成 Lua，设计人员需要把包含头文件的函数库编译进游戏，实例化 lua_state 解释器，向解释器注册任何需要访问 Lua 脚本文件中数据的宿主函数，然后传递数据、命令和文件给解释器并执行。游戏结束时，关闭解释器。
- 脚本文件可以编译成字节码，这样执行的速度会快些。脚本文件也可以加密，这样就可以避免被人盗读。
- 在本章涉及到的测试平台中，在 Lua 脚本中实现所有的状态转移逻辑。在大型游戏设计时，比较鲁棒的实现方案是在 Lua 中实现整个状态机，而只在脚本和宿主代码之间传递游戏感知部分和用于对感知做出反应的注册行为。每个状态的功能块都是模块化和易维护的。
- 多线程和协同例程可支持跨游戏循环的大规模 Lua 脚本，可支持脚本运行的暂停和多线程环境下的运行。
- 脚本系统的优点包括快速原型开发、入门简单、调试方便和便于扩展。
- 脚本系统的缺点包括执行速度低、调试难度大、脚本空洞化、直接关系到脚本性能和学习难度的脚本功能划分不合适以及脚本设计带来了额外工作。这些缺点在很多时候可以克服。
- 对基本脚本语言的扩展有内建调试工具、智能 IDE、自动集成工具和自主修改脚本。
- 在对系统进行优化时，要考虑游戏系统的特殊性和脚本系统的特殊性。有些脚本系统在采用基于简单反射行为时的性能比基于模型计算反射时的性能好。对于可重入系统，多线程和协同例程可以提高性能。

19 基于位置的信息系统

与第 17 章“基于消息的系统”类似，本章将介绍一系列用于辅助全局 AI 决策引擎的 AI 技术。但和基于消息的系统侧重于通信技术不同，基于位置的信息(Location-Based Information，LBI)系统通过向智能引擎提供额外的信息来帮助提高引擎的能力。这些额外的信息是有关游戏世界的某种形式的数据，这些信息通过集中存储体结构(centralized bank)的形式给出。LBI 可以看作是一种特殊的感知数据类型，也可能包含在嵌入式逻辑或者更底层的 AI 中。

19.1 基于位置的信息系统概述

在本书中，有关 LBI 的讨论将会常规地分为 3 类(但它们并不相互排斥)：影响图(influence map)技术、智能地形(smart terrain)技术和地形分析(terrain analysis)技术。本章首先将分别简单介绍三类技术的基本概念，然后实现部分简单影响图技术。另外两类技术公认比较复杂，和具体的游戏也更加相关，完全实现这些技术已超出了本书介绍的范围，因此在本章中，没有完全实现这些技术，但还是针对本章的测试用例和实际的游戏大概介绍了智能地形和地形分析的实现方法。

19.1.1 影响图技术(IM)

影响图正慢慢成为一种在游戏软件中普遍使用的 AI 辅助技术。它通用的结构和开放的使用方法使它成为继有限自动机(FSM)之后最易实现的技术，适用于不同游戏 AI 问题。术语“影响图”意味着其数据结构是简单数组，数组的每个成员代表某特定位置的数据。从概念上看，IM 被认为是游戏世界的 2D 网格结构。该网格的分辨率(数组成员数量)主要取决于游戏所需存储信息的游戏空间的最小值。数据大小和影响精度的折中决定了网格的分辨率。因此，在大型游戏世界中，如果我们需要知道每平方英寸上的具体数据，IM 就需要很高的分辨率，也需要相当大的存储空间来存储这些数据。很多游戏通过采用多个 IM 来减少存储和查找开销，或者为不同的 AI 系统提供不同层次的游戏空间的分辨率。因此，即时策略类(RTS)游戏中都存在一个低分辨率的 IM(可以这么说，每个元素代表一个游戏屏幕)，该 IM 存储了每种资源的总量。游戏通过该 IM 来制定决定何时建立城镇以及城镇的发现方向的计划。一般地，玩家都希望把分基地建在资源更丰富以及更利于以后扩张的位置上。我们的示例游戏中另外利用了一个分辨率更高的 IM 用于保存每个网格节点上被消

灭的游戏单元，该IM的每个节点大概有4个基本节点组成。该图用于指导路径搜索引擎，避免后继单元进入高死亡率的区域。

对于采用大量3D技术的游戏则需要更加复杂的数据结构，其中游戏画面通过垂直画面层相互叠加而成。这些数据结构可以是分层IM或者导航mesh(navigation mesh)，通过这些数据结构来完成路径搜索。另外还有一种称为局部IM(local IM)的技术。比如，在即时策略(RTS)类游戏中敌对势力的战斗可能会在地图的任意位置开始，设计人员需要详尽的IM以匹配各方势力的表示，但是他们不希望占用太多的内存来达到他们的要求。因此，设计人员一般不采用全局IM技术，而是采用局部IM技术。系统在检测到战斗场景后，以战斗场景为中心，扩展一定的距离得到可与战斗相匹配的局部战斗IM。战斗中心的检测可以通过另外低分辨率的IM来得到，这些IM中保存了人口数量和战斗位置的变化过程。因此，在全局IM相当受限的前提下，局部信息仍然可以很详尽。

很多游戏采用一种称为“高层逼近”(high-level approach)的技术来完成游戏角色之间的交互。每个游戏角色通过查询某个冗长的列表来确定该交互是否必要。该技术的使用有点类似于冲突检测，但是冲突检测中AI系统对感知距离更加地敏感。如此，游戏中的所有角色看起来都像是处在比游戏画面更高的位置，俯视整个游戏世界并攫取感兴趣的信息，但这不是真实世界中人类的做法。人类只有有限的视野，只能对很小范围内的事件做出反应。我们在IM中保存玩家的位置信息，那么该IM将允许更低层次手段的交互行为。通过该IM限制每个游戏角色的视野范围更加的符合客观世界。该IM可以提供中心位置以供游戏实体来完成交互数据的查询，可以提供一种存储其他类型信息的平台，可以帮助降低游戏实体的耦合度，可以减少因游戏实体和数据的全局查询而附加的代码。

IM所具有的通用数据结构的特点可以帮助我们通过大量不同的手段来完成有关位置的特殊数据的构造，唯一受限的地方是设计人员的想象力和游戏的特殊性。事实上，该基本系统可以并且通常也是游戏的集中存储库，用于支持其他两种LBI技术与游戏其他部分完成通信。

19.1.2　智能地形技术(Smart Terrain)

智能地形技术在SIM游戏中开始流行，该术语由Will Wright提出，Will Wright是SIM的创始人。智能地形技术把游戏逻辑和行为等放入到对象内部，这些数据用于确定怎么利用游戏世界的特征以及其他游戏对象的方式。在SIM游戏中，游戏角色由不同的需求来驱动，这些需求可以通过和不同角色的交互得到满足。SIM程序员可以给不同的对象分配不同的属性，这些属性表示游戏角色的特殊需求。微波炉满足食物的需求，而床满足睡眠的需求。游戏角色遍历游戏世界，寻求其在某给定时刻没有得到满足的需求，它们通过监听来自它们周围的智能对象广播的消息来完成游戏世界的遍历。这些消息将和角色的不同对象中等待满足的需求类进行通信，而游戏角色可以“免费”地利用这些游戏对象满足自己的需求。当然，在这里我们只是大概给出了SIM游戏的工作过程，不过相信读者基本掌握了其中的要点。

智能地形技术把角色在使用特定对象时所需的交互激励、声音和其他的特殊数据都捆绑在一个对象中，因此该对象是完全自包含的。这样，在任何时候开发人员都可以加入新对象，并且只需要满足两个条件：新对象必须包含所有必要数据，游戏中的每个对象必须

是功能完全的对象，新对象能满足一个或者多个基本需求，这样游戏角色才会找到和使用该对象。

19.1.3　地形分析技术(Terrain Analysis，TA)

在 IM 中跟踪不同属性和统计数据的变化只是问题的一方面。我们还必须使用这些数据。TA 是使用 IM 技术的一系列方法，其利用 IM 技术向 AI 系统提供有关地图状态的策略信息，特别是随机生成的地图的策略信息，如果没有这些策略信息，就算是一级设计师也很难从该图上获得好处。甚至经过 TA 预处理的定制地图也需要在线地图分析，这是由于大多数游戏都存在一些动态改变的游戏单元。TA 技术最恰当的解释是针对 IM 数组的模式识别专用技术。设计人员通过对 IM 数组的遍历来确定战略和战术决策的主基调。任何在中等规模的 IM 上工作的 TA 技术都需要大量的计算资源，这是由于 TA 中存在的搜索过程需要大量模式匹配算法。为了克服该缺点，一些游戏采用非精确方法来确定模式，比如神经网络和模糊逻辑系统，通过这些技术，我们能在算法上完成 IM 数组的模式搜索。

19.2　各种技术的使用方法

IM 技术在 AI 游戏中使用得越来越多，特别是即时策略类游戏(其他类型的游戏纷纷跟进，比如角色扮演类游戏、动作类游戏和冒险类游戏)。IM 技术在游戏中的应用手段包括：占用数据(occupance data)、场地控制(ground control)、路径查找(pathfinding)、危险预警(danger signification)、初步的战场计划、简单地形分析和高级地形分析等。

19.2.1　占用数据

占用数据的作用是跟踪游戏中不同位置不同人口类型的变化过程。IM 技术的简单应用之一就是跟踪某区域内不同类型游戏对象的数量变化。设计人员可能需要跟踪所有战斗单元、专门资源的位置、重要请求条目或者其他在游戏的对象。简单占用数据可以在很多方面应用：障碍的躲避，游戏感知的粗略估计，势力发展方向的确定，以及其他一些需要局部人口数量快速访问的任务。

大家所熟悉的游戏《战争之雾》中的视界系统是占用 IM 技术的一个比较常见的应用，这样的视界系统几乎出现在所有的即时策略类游戏中。开始时，地图上的所有位置都充满着烟雾，因此玩家不得不在地图的所有地方探路，寻找资源和敌对城镇。探索过的地图块的烟雾将会被移除，玩家也就能看清其中的资源和敌对城镇的细节，但是如果玩家希望实时地观测某个位置上的活动，则必须在该位置的观测范围内安排自己的游戏单元。设计人员在设计游戏时，通过一些占用 IM 来保存所有单元的视界范围内的资源和活动。

19.2.2　场地控制

场地控制致力于游戏场地的实际影响的确定。尽管在游戏 AI 中术语影响图是定义比较粗略的数据结构，该术语所指的技术最早出现于热力学领域(用于表示热变化)和场分析

领域(比如地磁场)。这些等式也可以用于游戏设置，可以迅速定位哪个玩家控制地图的哪个部分。

场地控制的算法非常简单：首先清空整幅地图，根据各个玩家的控制程度给地图中的各个方块赋值，不同大小的数值表示不同的控制程度，一个玩家对应一个数值。如果游戏中只有两个玩家，那么可以对玩家A赋予正值，而对玩家B则赋予负值，如果游戏中的玩家较多，则赋值模式相对复杂些。然后，再次遍历整幅地图，把每个方块的数值和周围的各方块上的数值相加后得到新数值，为避免新数值溢出，把该数值减去某常量后作为该方块最终的数值。重复上述过程若干遍，直至影响图进入稳定状态。最后在只有两个玩家时，玩家A控制影响值为正的网格，而玩家B控制影响值为负的方块。该技术既可以用于度量全局控制情况，也可以用于度量局部控制情况。没有被直接控制的区域，其影响值将会受附近被直接控制的区域的数值的影响。整个游戏世界被分成了各个玩家的势力范围。我们可以通过计算各玩家控制的方块的数量以决定哪个玩家控制的面积最大，哪个是最大的赢家。

19.2.3　探路系统的辅助数据

在提供了某特定区域的信息后，虽然该区域地形奇特，但探路系统还是能帮助给出平滑地通过该区域的方案，方案的形式可以是直接给出捷径或者是对AI控制角色使用地图特性的支持，比如远程端口(teleport)或者梯形图(ladder)。探路者数据主要包括通行能力、与地形特征(比如山峰或者悬崖)和地形类型(比如陆地或者海洋)的相对位置、准备通过区域的控制玩家的信息等。很多游戏使用简单的势场类技术作为对基于节点的路径系统的补充，这些技术一般通过影响因子来实现，影响因子可在设计时设定或者程序运行时产生。通常这些技术可以帮助AI角色，使它们能避开危险区域，这些区域有成堆巡游的怪兽聚集一起。势场IM能不断地变化并适应不同的游戏条件，在游戏角色身上表现出来的行为就是它们能随着游戏的进程不停地学习。举个在游戏的例子——运动类游戏，比如冰上曲棍球游戏，在这些游戏中，小型偏移量IM场作为信息系统的补充而存在。在游戏启动的时候，所有偏移量都清零，此时机器控制的团队使用标准信息来完成定位。一旦人类玩家和该团队开始游戏对抗，该IM就开始为机器玩家跟踪人类滑行习惯位置和传球位置的修正值以寻找击败人类玩家的方法。如此人类玩家将被迫不断地改变玩游戏的方式，只有这些才能更好地得分，因为机器团队将不断地精确化人类玩家玩游戏的习惯方式。

19.2.4　危险预警

IM技术的另一个有效应用是跟踪某段时间内某区域内恶性事件的发生和变化过程。该数据能帮助修正AI行为，使AI角色不至于不停地做出相似举动并导致恶性事件不断地出现。以在即时策略类游戏这一章中提到的示例为例，AI角色在探路过程中会人为放置攻击塔，很多AI游戏都这么做。该塔将在后继游戏中杀死整列零星靠近的敌对单元。如果AI角色使用了危险预警系统，这些单元将在靠近该塔时停下来，因为AI角色将看到影响通过该区域的探路代价的危险数据，如果AI系统能派出攻击团队来调查该区域危险的来源

那就更好。再举个例子，比如第一/三人称(FTPS)射击类游戏中的自爆 AI 机器人，该机器人将通过 IM 表示的危险预警系统来侦测它埋伏或者说狙击的区域的变化，这样它才能避开那些危险区域并很容易地靠近敌人。

19.2.5　初步战场计划

详细的场地控制方法能用以快速指出战场中的各方感兴趣的区域。通过定位零值区域或者接近零值的区域，我们能很容易地发现各方可能会陈兵对抗的位置以及各方争夺的焦点区域。这将告诉我们冲突的位置和特定战场的前线位置。大型接近零值的区域可能是没有被任何一方控制的区域。通过了解自身主力位置，自身主力相对于敌对方主力的位置以及各方的规模，玩家可以确定攻击方向，确定胜算以及可能的代价，更加合理地分配新训练出的后继部队，更加清晰地协调各条战线。

19.2.6　简单战场分析

简单战场分析包括更多的数学测定技术，比如覆盖率(cover)(给定点给定任何角度的可攻击程度)，可见度(visibility)(某种程度上和覆盖率正好相反，但是还需考虑到视界焦点和视线问题)以及高度因子(height factor)(很多游戏允许导弹飞离地面很高并获得更高的可见度)。覆盖率最好的区域将变成狙击机器人集中区域，可见度较低的区域将成为地图中其他区域的暗门，可见度较高的区域将成为伏击的目标，但是前提是该区域的周围是高覆盖率的地形。

19.2.7　高级战场分析

通常，在即时策略(RTS)类游戏中，当 AI 角色和人类进行对抗时需要更高级的 TA 方法来使其变得更加智能。

寻找地图中优良的咽喉位置以及在运动和可见度极度受限的地方立足是常见的 IM 使用方式。通过扫描地图的这些特性，如果咽喉位置在两个或者更多的主要地图区域之间，AI 角色将在此设置伏击点，特别是如果该咽喉位置是个“完美”的咽喉位置，也就是说除了该位置，不存在另外的位置连接两大区域，而且其他玩家必须通过该位置才能赢得游戏。为了保卫某区域不受威胁有时需要在该区域周围筑墙，而在咽喉位置筑墙可以最大程度地减少筑墙代价。

IM 信息的另一个主要用途在于确定修筑城堡和防御城墙以及其他建筑的位置。修筑城堡需要一些事前计划。一方面，建筑位置必须要保持人群能持续地受到控制，这样做主要有两方面要考虑，第一是探路的需要，第二是保护群体的需要，建筑的密度太高将会带来更多的危险系数，很容易受到大型火炮类武器的攻击。另一方面，建筑位置应该可以最大化未来的发展潜力，建筑位置应该有利于抢占更多的资源，还需要考虑到原有建筑和新建筑之间的通路。最后，建筑位置还需要考虑建筑的可维持性，城堡不应该有很多的受攻击面，在城堡最薄弱的一面增加防御工事以强化城堡的防御能力。AI 系统会合理地利用不可通过的区域，在该区域上筑墙，因此可以减少受攻击面。人类通过筑墙来减慢 AI 敌人在

某区域的探路或者改变 AI 敌人的探路方向(图 19-1 给出了筑有复杂城墙的人造杀人区)。这些诡计同样可以被 AI 系统所采用以欺骗那些没有使用微操作来指挥自身力量的人类玩家。但是使用这些手段需要 AI 系统寻找到有利位置，否则这些 AI 行为将看起来非常愚蠢。

确定某个位置是否是重要区域(比如地图中包含大量资源的位置或者重要的战略点)是人类最擅长的。在考虑某地图布局时(参见图 19-2)，一位优秀的人类玩家能迅速地反应出需要控制区域 A，因为该点包含了最多的火力并且很好防御。而 AI 系统在这方面通常是相当弱智的，但是包含这类信息(火力分布、覆盖率信息以及咽喉位置)变化的 IM 能在一定程度上弥补 AI 系统的这个缺点。

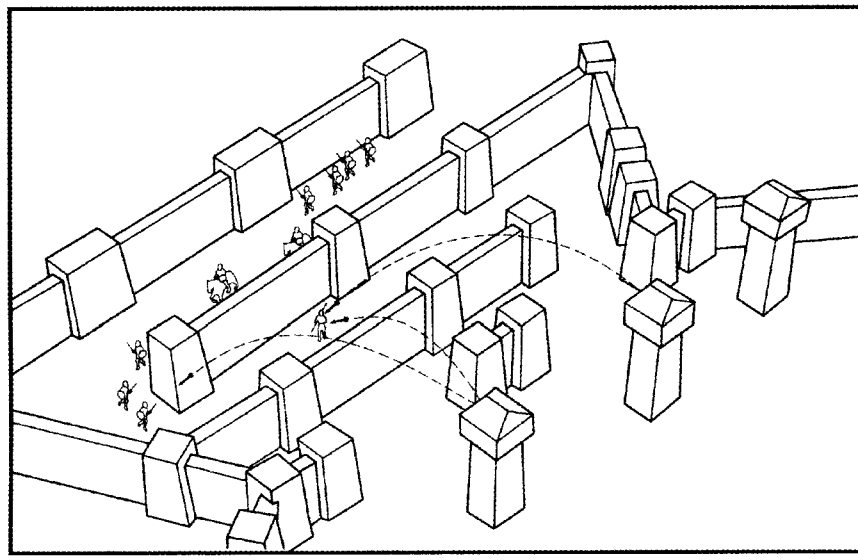


图 19-1　用于迷惑 AI 攻击者的迷阵墙

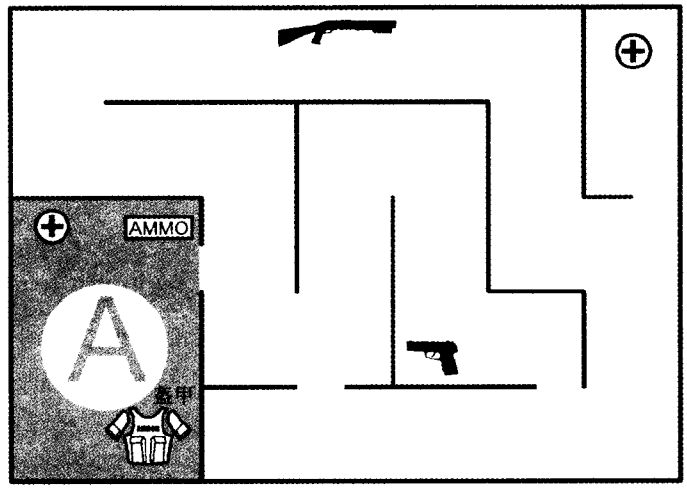


图 19-2　包含多种策略元素的地图示例

19.3　影响图框架代码及测试平台实现

本节将在测试平台上实现一些不同类型的基本 IM，目的很单纯，只是为了说明 IM 技术本身，这些实现对于基于该测试平台的 AI 子系统的决策不产生任何影响。同时，本节将展示收集信息并在 IM 中集中是多么简单，并通过调试系统图形化地输出这些信息，调试系统支持在游戏过程中抽取游戏网格以及其中的内容。在每个实现后面，我们将讨论怎样在测试平台中引入特殊方法来改进性能。

我们将给出 3 种简单类型 IM 的实现，并展示它们的不同用法。如下所示：

- **基于占用的 IM**。在该 IM 中，通过 IM 技术来完成某个特定游戏对象在游戏中位置轨迹的跟踪。
- **基于控制的 IM**。在该 IM 中，通过使用梯度来展示每个游戏对象的控制场地以及各游戏方怎样利用该概念。
- **逐位 IM**。在该 IM 中，把 IM 元素的数值拆分成逐位数据元件。

每个 IM 类型从基本 IM 类 InfluenceMap 继承得到(类的头声明参见程序清单 19-1，类的函数实现参见程序清单 19-2)。正如读者所看到的，基本类中最吸引人的是 IM 数组 m_map，它是一个整型数组(数组成员是 16 位无符号数)。如果读者需要数组具备很多或者更少的容量，读者完全可以按照自己的需要改变该数组的容量。甚至读者可以自定义数组的成员为结构体，但此时读者还需要相应地修改类的实现体。

程序清单 19-1　InfluenceMap 头信息

```
struct RegObj
{
   GameObj*  m_pObject;
   int       m_objSizeX;
   int       m_objSizeY;
   int       m_objType;
   Point3f   m_lastPosition;
   bool      m_stamped;
};

typedef std::list<RegObj*> RegObjectList;

class InfluenceMap
{
public:
   //constructor/functions
   InfluenceMap(int type):m_influenceType(type)
                     {m_drawGrid = false;m_drawInfluence = false;}
   ~InfluenceMap();
   virtual void Update(float dt) {}
   virtual void Draw();
   virtual void DrawTheGrid();
```

```
    virtual void DrawTheInfluence();
    virtual void Init(int sizeX, int sizeY, int wSizeX, int wSizeY);
    virtual void Reset();
    virtual void RegisterGameObj(GameObj* object);
    virtual void RemoveGameObj(GameObj* object);
    virtual void StampInfluenceShape(int* pMap,Point3f& location,
                                     int sizeX,int sizeY, int value);
    virtual void StampInfluenceGradient(int* pMap,Point3f& location,
                                        int initValue);
    int SumInfluenceShape(int* pMap,Point3f& location,
                          int sizeX,int sizeY);
    int GetInfluenceValue(int* pMap,Point3f& location);
    void SetType(int type) {m_influenceType = type;}
    void DrawGrid(bool on = true){m_drawGrid = on;}
    void DrawInfluence(bool on = true){m_drawInfluence = on;}
    int GetSizeX(){return m_dataSizeX;}
    int GetSizeY(){return m_dataSizeY;}

    //influence map types
    enum
    {
        IM_NONE,
        IM_OCCUPANCE,
        IM_CONTROL,
        IM_BITWISE
    };

protected:
    //data members
    int* m_map;
    RegObjectList m_registeredObjects;

    int m_dataSizeX;
    int m_dataSizeY;
    int m_numCels;
    int m_worldSizeX;
    int m_worldSizeY;
    float m_celResX;
    float m_celResY;
    int m_influenceType;

    bool m_drawGrid;
    bool m_drawInfluence;
};
```

该影响系统通过一个称为 registeredObjects 的列表维护注册游戏对象。游戏对象不需要担心更新它们在 IM 中存在的数据，因为系统本身完成了列表的维护工作，但程序员必须记住在这些对象消失的时候在 IM 系统列表中将相应的项清除掉。

StampInfluence()和 StampInfluenceGradient()这两个函数用于在 IM 中写入数值。完全版本仅仅在数组中写入数值块，数值的大小和位置通过参数传入，梯度版本则在数组的某个开始位置写入数据减少量的平方梯度。这些操作很简单，因此它们可以在基类中实现，但它们也可以在任何用户自定义的子类中进行重载以完成 IM 数组的自定义写。

GetInfluenceValue()函数是地图的存取器，SumInfluence()函数则用于完成特定外形的区域内各点的影响值的求和运算。值得注意的是，所有处理 IM 成员地图的函数的参数中都包含指向该数组的指针。这样有助于处理自定义 IM 类型，在处理这些类型的 IM 时需要引入临时地图以完成在全图上进行多步操作。

程序清单 19-2　InfluenceMap 类主要函数的实现

```
//--------------------
InfluenceMap::~InfluenceMap()
{
   if(m_registeredObjects.size() == 0)
      return;
   RegObjectList::iterator listObj;
   for(listObj=m_registeredObjects.begin();
      listObj!=m_registeredObjects.end();++listObj)
   {
      delete (*listObj);
   }
   m_registeredObjects.clear();
}
//--------------------
void InfluenceMap::Init(int sizeX, int sizeY, int wSizeX, int wSizeY)
{
   m_dataSizeX = sizeX;
   m_dataSizeY = sizeY;
   m_numCels = m_dataSizeX*m_dataSizeY;
   m_map = new int[m_numCels];

   //clear out the map
   memset(m_map,0,m_numCels*sizeof(int));

   m_worldSizeX = wSizeX;
   m_worldSizeY = wSizeY;
   m_celResX = m_worldSizeX / m_dataSizeX;
   m_celResY = m_worldSizeY / m_dataSizeY;
}

//--------------------
void RemoveAll(RegObj* object)
{
   delete object;
```

```
}

//--------------------
void InfluenceMap::Reset()
{
    //clear out the map
    memset(m_map,0,m_numCels*sizeof(int));

    //get rid off all the objects
    if(m_registeredObjects.size() == 0)
        return;
    for_each(m_registeredObjects.begin(),
             m_registeredObjects.end(),RemoveAll);
    m_registeredObjects.clear();
}

//--------------------
void InfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeY,sizeX;
    sizeX = sizeY = 1;
    RegObj* temp;
    temp = new RegObj;
    temp->m_pObject = object;
    temp->m_objSizeX = sizeX;
    temp->m_objSizeY = sizeY;
    temp->m_lastPosition = object->m_position;
    temp->m_stamped = false;
    m_registeredObjects.push_back(temp);
}

//--------------------
void InfluenceMap::RemoveGameObj(GameObj* object)
{
    if(m_registeredObjects.size() == 0)
        return;

    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        RegObj* temp = *listObj;
        if((*listObj)->m_pObject == object)
        {
            m_registeredObjects.erase(listObj);
            delete temp;
```

```
            return;
        }
    }

}

//--------------------
void InfluenceMap::StampInfluenceShape(int* pMap,Point3f& location,int
sizeX, int sizeY, int value)
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;

    int startX = gridX - sizeX/2;
    if(startX < 0) startX += m_dataSizeX;
    int startY = gridY - sizeY/2;
    if(startY < 0) startY += m_dataSizeY;
    for(int y = startY;y<startY + sizeY;y++)
    {
        for(int x = startX;x<startX + sizeX;x++)
        {
            pMap[(y%m_dataSizeY)*m_dataSizeY+(x%m_dataSizeX)]+=value;
        }
    }
}

//--------------------
int InfluenceMap::GetInfluenceValue(int* pMap,Point3f& location)
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;
    return pMap[gridX,gridY];
}

//--------------------
int InfluenceMap::SumInfluenceShape(int* pMap,Point3f& location,
                                    int sizeX,int sizeY)
{
    int sum = 0;
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;

    int startX = gridX - sizeX/2;
    if(startX < 0) startX += m_dataSizeX;
    int startY = gridY - sizeY/2;
    if(startY < 0) startY += m_dataSizeY;

    for(int y = startY;y<startY + sizeY;y++)
```

```
    {
        for(int x = startX;x<startX + sizeX;x++)
        {
            sum+=pMap[(y%m_dataSizeY)*m_dataSizeY+(x%m_dataSizeX)];
        }
    }
    return sum;
}

//--------------------
void InfluenceMap::StampInfluenceGradient(int* pMap,Point3f&
                                                    location, int initValue)
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;

    int stopDist = fabsf(initValue)*0.75f;//*(m_dataSizeX/32);
    int halfStopDist = stopDist / 2;
    int startX = gridX - halfStopDist;
    if(startX < 0) startX += m_dataSizeX;
    int startY = gridY - halfStopDist;
    if(startY < 0) startY += m_dataSizeY;

    for(int y = startY;y<startY + stopDist;y++)
    {
        for(int x = startX;x<startX + stopDist;x++)
        {
            int value;

            int distX = fabs(x - (startX + halfStopDist));
            int distY = fabs(y - (startY + halfStopDist));

            value = initValue*( halfStopDist -
                        MAX(distX,distY))/halfStopDist;
            pMap[(y%m_dataSizeY)*m_dataSizeY +
                      (x%m_dataSizeX)] += value;
        }
    }
}
```

19.3.1　占用影响图

在读者了解了基本系统后，接下来我们将继续给出前面提到的3种IM的实现。程序清单19-3和程序清单19-4中给出了OccupanceInfluenceMap类的头信息和实现体，该类实现的简单IM跟踪了IM中不同游戏对象人口数量的变化。

程序清单 19-3　OccupanceInfluenceMap 头信息

```
class OccupanceInfluenceMap:public InfluenceMap
{
public:
     //constructor/functions
     OccupanceInfluenceMap():InfluenceMap(IM_OCCUPANCE){}
     ~OccupanceInfluenceMap();
     virtual void Update(float dt);
     virtual void RegisterGameObj(GameObj* object);
     virtual void RemoveGameObj(GameObj* object);
     virtual void DrawTheInfluence();
};
```

在程序清单 19-4 中，Update()函数完成了主要工作。同时也可以看到 IM 数据的两种不同的处理方式。在 Update()方法中，memset()调用被注释掉，后面跟着段代码反标记(unstamp)原有位置并标记(stamp)新位置。在继续更新之前的反标记功能有点类似绘图系统中的“脏矩形”(dirty rectangle)技术，在该技术中我们只需重新绘制必要的单元而不是整个画面。当然，程序员也可以注释掉这段反标记代码，但同时需要在标记循环中撤销运动检查并注释掉 memset()调用。如果所设计的游戏世界很小并且需要编写大量的对象，比如我们设计的测试平台，更理性的方案是直接擦除整个 IM 数组并重新开始。但在具有大型游戏世界并且没有很多游戏对象或者具有静态 IM 数据(比如地形特征或者特别的 IM 标记)的游戏中，设计人员往往采用脏矩形方案，因此不需要更新缓冲区。我们必须注意到，由于该类使用了反标记过程，该类的 RemoveGameObj()方法也必须反标记删除的对象，不至于留下垃圾数据。

RegisterGameObj()函数同样用于设置对象影响值的大小。该函数更多地是算法过程，把近乎圆形的游戏对象转变成测试平台中的方形 IM 阴影，该过程被证实非常有效。在 DrawInfluence()函数中，我们应该注意到我们为每个 IM 数组元素绘制了灰色的多边形，该多边形的值在 IM 中存在 10 个对象时达到最大值。

程序清单 19-4　OccupanceInfluenceMap 类重要函数的实现

```
//--------------------
void OccupanceInfluenceMap::Update(float dt)
{
    //bail out if nobody to update
    if(m_registeredObjects.size() == 0)
        return;

    // clear out map
    // memset(m_map,0,m_numCels*sizeof(int));
    RegObjectList::iterator listObj;
    //unstamp old locations
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
```

```
            if((*listObj)->m_pObject->m_position ==
                (*listObj)->m_lastPosition)
                continue;
            if((*listObj)->m_stamped)
                StampInfluenceShape(m_map,(*listObj)->
                                    m_lastPosition,(*listObj)->
                                    m_objSizeX,(*listObj)->m_objSizeY, -1);
        }

        //stamp new locations
        for(listObj=m_registeredObjects.begin();
            listObj!=m_registeredObjects.end();++listObj)
        {
            if((*listObj)->m_pObject->m_position ==
                (*listObj)->m_lastPosition)
                continue;
            StampInfluenceShape(m_map,(*listObj)->
                                m_pObject->m_position,(*listObj)->
                                m_objSizeX,(*listObj)->
                                m_objSizeY, 1);
            (*listObj)->m_stamped = true;
            (*listObj)->m_lastPosition = (*listObj)->m_pObject->m_position;
        }
    }

    //--------------------
    void OccupanceInfluenceMap::RemoveGameObj(GameObj* object)
    {
        if(m_registeredObjects.size() == 0)
            return;

        RegObjectList::iterator listObj;
        for(listObj=m_registeredObjects.begin();
            listObj!=m_registeredObjects.end();++listObj)
        {
            RegObj* temp = *listObj;
            if((*listObj)->m_pObject == object)
            {
                if((*listObj)->m_stamped)
                    StampInfluenceShape(m_map,(*listObj)->
                                        m_lastPosition,(*listObj)->
                                        m_objSizeX,(*listObj)->
                                        m_objSizeY, -1);
                m_registeredObjects.erase(listObj);
                delete temp;
                return;
            }
        }
```

```
}

//--------------------
void OccupanceInfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeX,sizeY;
    if(object->m_size <4)
    {
        sizeX = m_dataSizeX/16;
        sizeY = m_dataSizeY/16;
    }
    else if(object->m_size<11)
    {
        sizeX = m_dataSizeX/10;
        sizeY = m_dataSizeY/10;
    }
    else if(object->m_size<33)
    {
        sizeX = m_dataSizeX/8;
        sizeY = m_dataSizeY/8;
    }
    else if(object->m_size <49)
    {
        sizeX = m_dataSizeX/5;
        sizeY = m_dataSizeX/5;
    }
    else if(object->m_size <65)
    {
        sizeX = m_dataSizeX/4;
        sizeY = m_dataSizeX/4;
    }
    else
    {
        sizeX = m_dataSizeX/3;
        sizeY = m_dataSizeX/3;
    }

    //set minimum size of 1 in each direction
    sizeX = MAX(1,sizeX);
    sizeY = MAX(1,sizeY);

    RegObj* temp;
    temp = new RegObj;
    temp->m_pObject = object;
    temp->m_objSizeX = sizeX;
    temp->m_objSizeY = sizeY;
    temp->m_lastPosition = object->m_position;
```

```
    temp->m_stamped = false;
    m_registeredObjects.push_back(temp);
}

//--------------------
void OccupanceInfluenceMap::DrawTheInfluence()
{
    glPushMatrix();
    glDisable(GL_LIGHTING);
    glTranslatef(0,0,0);
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
    for(int i=0;i<m_numCels;i++)
    {
        if(m_map[i])
        {
            int y = i / m_dataSizeY;
            int x = i - y*m_dataSizeY;
            float grayscale = m_map[i]/10.0f;
            glColor3f(grayscale,grayscale,grayscale);
            glBegin(GL_POLYGON);
            glVertex3f(x*m_celResX, y*m_celResY, 0);
            glVertex3f(x*m_celResX, y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,y*m_celResY, 0);
            glEnd();
        }
    }
    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
    glPopMatrix();
}
```

在使用该系统时，程序员首先需要实例化该地图，初始化该地图，按照需求初始化网格的分辨率和游戏世界的大小。程序员然后要注册所有需要在 IM 中跟踪的对象。在我们设计的测试平台中，所有对象都在 IM 中进行了注册。GameObj 类的另一个数据成员——布尔变量 m_influence 也加入到 IM 中，因此程序员可以通过该变量来决定某个特定的对象是否会影响系统数值。

19.3.2　占用 IM 测试平台的使用

图 19-3 所示为该系统测试平台的截屏图，该图主要用于调试系统。在 AIsteroids 测试平台中使用占用 IM 系统可以提高躲避状态，可以指导玩家更好地躲避拥挤地带。程序员也可以在游戏世界范围内安排静态占用环，然后躲避状态将会尝试保持，使得主飞船不至于太靠近游戏边界，在边界满世界快速飞行的小飞船将会吸引主飞船并将其撞毁。

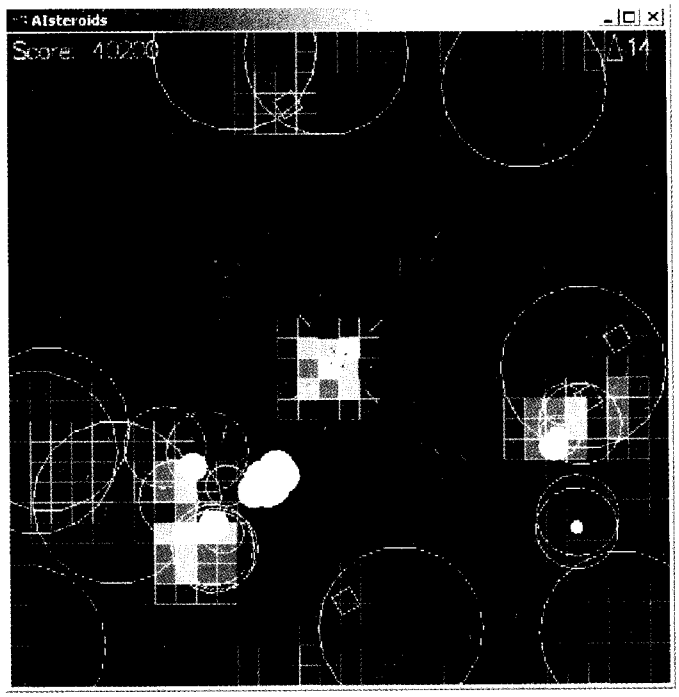


图 19-3　占用 IM 在 AIsteroids 测试平台中的使用图

在攻击状态时，主飞船也会检测占据地图并发射大量子弹，因此和主飞船在同一位置的多个小飞船将同时受到攻击。

19.3.3　控制影响图

接下来给出的系统用于记录游戏区域的控制问题。游戏对象将在地图中写入控制梯度，该梯度值的大小由对象的大小决定，除了一些特殊的对象，这些对象将直接赋予和大小没有关系的梯度值。主飞船和子弹的影响值为正数，而小飞船的影响值为负数。地图上所有的对象影响值累加在一起，因此哪个 IM 单元上的影响正值更大些表示那个单元存在更多的主飞船和子弹，相反地，哪个 IM 单元上的影响负值更大些表示那个单元存在更多的小飞船。程序清单 19-5 和程序清单 19-6 给出了 ControlInfluenceMap 类的头信息和实现体。

程序清单 19-5　ControlInfluenceMap 头信息

```
class ControlInfluenceMap:public InfluenceMap
{
public:
    //constructor/functions
    ControlInfluenceMap():InfluenceMap(IM_CONTROL){}
    ~ControlInfluenceMap();
    virtual void Update(float dt);
    virtual void RegisterGameObj(GameObj* object);
    virtual void DrawTheInfluence();

};
```

值得注意的是，在该类中没有影响值的反标记过程，在每次更新时，我们只是简单地清除 IM 数组。在 ControlInfluenceMap 类中使用特殊数据类型来表示注册的对象，只有 OT_FRIENDLY 和 OT_ENEMY 参与更新，OT_BULLET 只是 OT_FRIENDLY 类型的特例。通过使用该数据类型来决定在地图中是写入控制数据的正值还是负值。

DrawInfluence()函数用于进行 IM 数组单元彩色多边形的绘制，具体的颜色和形状根据每个位置的控制数值的大小和符号而定。

程序清单 19-6　ControlInfluenceMap 重要函数的实现

```
//--------------------
void ControlInfluenceMap::Update(float dt)
{
    //bail out if nobody to update
    if(m_registeredObjects.size() == 0)
        return;

    //clear out map
    memset(m_map,0,m_numCels*sizeof(int));

    //stamp obj locations
    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        //only care about "control" objects, not miscellaneous
        if((*listObj)->m_objType == OT_MISC)
            continue;

        if((*listObj)->m_objType == OT_FRIENDLY)
            StampInfluenceGradient(m_map,(*listObj)->
                                   m_pObject->m_position, 16);
        else if((*listObj)->m_objType == OT_BULLET)
            StampInfluenceGradient(m_map,(*listObj)->
                                   m_pObject->m_position, 8);
        else
            StampInfluenceGradient(m_map,(*listObj)->m_pObject->
                    m_position, -(((*listObj)->m_pObject->
                                   m_size)/2));
        (*listObj)->m_lastPosition = (*listObj)->m_pObject->
                                                    m_position;
    }
}

//--------------------
void ControlInfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeX,sizeY;
    sizeX = sizeY = 1;

    RegObj* temp;
```

```
    temp = new RegObj;
    temp->m_pObject = object;
    temp->m_objSizeX = sizeX;
    temp->m_objSizeY = sizeY;
    temp->m_lastPosition = object->m_position;
    temp->m_stamped = false;
    if(object->m_type == GameObj::OBJ_SHIP ||
       object->m_type == GameObj::OBJ_SAUCER)
        temp->m_objType = OT_FRIENDLY;
    else if(object->m_type == GameObj::OBJ_BULLET)
        temp->m_objType = OT_BULLET;
    else if(object->m_type == GameObj::OBJ_ASTEROID)
        temp->m_objType = OT_ENEMY;
    else
        temp->m_objType = OT_MISC;
    m_registeredObjects.push_back(temp);
}

//--------------------
void ControlInfluenceMap::DrawTheInfluence()
{
    glPushMatrix();
    glDisable(GL_LIGHTING);
    glTranslatef(0,0,0);
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
    for(int i=0;i<m_numCels;i++)
    {
        if(m_map[i])
        {
            int y = i / m_dataSizeY;
            int x = i - y*m_dataSizeY;
            float color = m_map[i]/16.0f;
            if(color > 0)
                glColor3f(0,0,color);
            else
                glColor3f(-color,0,0);
            glBegin(GL_POLYGON);
            glVertex3f(x*m_celResX,y*m_celResY,0);
            glVertex3f(x*m_celResX, y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,
                       y*m_celResY+m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,
                       y*m_celResY, 0);
            glEnd();
        }
    }
    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
    glPopMatrix();
}
```

19.3.4　控制 IM 测试平台的使用

图 19-4 给出了控制 IM 系统的测试平台的截图，该测试平台是为调试而设计的。跟踪 AIsteroids 测试平台的控制信息可以带来很多好处。

通过尽可能地停留在控制范围，处于躲避状态的主飞船看起来就更加智能，这样的主飞船具备了主动躲避能力，而不是一般游戏采用的被动躲避手段。另外这样的主飞船也提供了简单的路径搜索平台，通过在平台在 IM 数组上的执行可以找到清晰的路径。如果在控制位置考虑了速度，或者跟踪了控制轮廓在行进过程中的变化梯度并把该数据作为位置信息传递给标记函数，就能提供更多的躲避能力。和占用 IM 类似，我们可以在游戏世界的边沿设置一个小飞船控制的静态环，这样躲避状态的主飞船就不至于进入游戏世界的边沿。这种类型的 IM 数据可能会产生其他模糊效果，静态控制环可采用平滑梯度，这样不至于使主飞船越靠近边沿越厉害。

如果距离最近的能量提升瓶(powerup)在主飞船控制范围之内，状态 GetPowerup 的优先级将提高。主飞船将累计小飞船的控制数据，在该数据足够低的情况下，飞船将最大状态 GetPowerup 的优先级以取得能量提升瓶(因此在游戏中存活的小飞船很少的情况下，主飞船将通过“吃”能量提升瓶提升能力并开足火力，这将极大地增加其在下一波中的存活概率)。

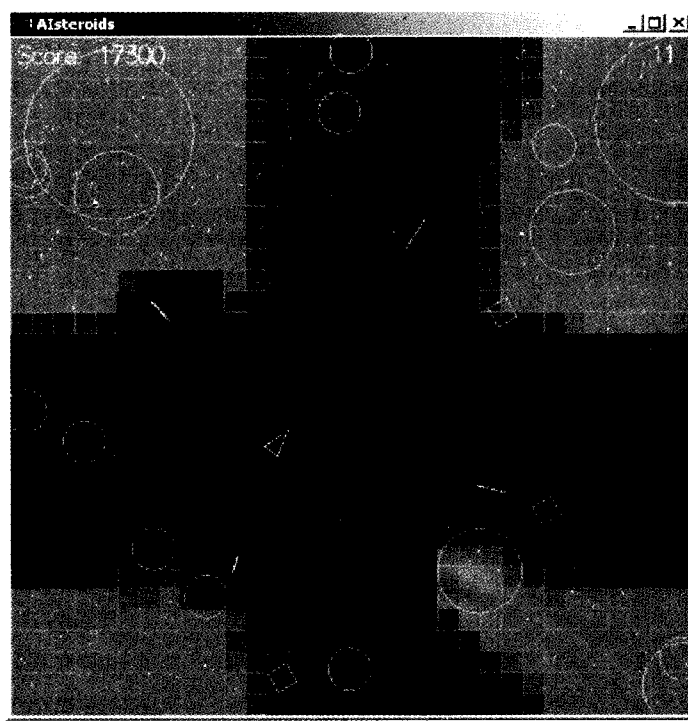


图 19-4　控制 IM 在 AIsteroids 测试平台中的使用图

19.3.5　逐位影响图

最后是一个简单的IM设计将展示如何通过布尔值的形式独立使用数组元素的每一位。IM数组的这种通用方法可使设计人员定制其在IM系统中需要跟踪的任何信息。在我们的测试平台应用中，我们将跟踪两部分内容：对象类型以及运动方向。每个数组元素的低 8 位对应对象类型，9~12 位上的不同数据表示对象是否在不同的主要方向上运动。这个系统的用法看起来有点随意，它仅仅用来演示该方法，并不说明一定要这么做。程序清单 19-7 和 19-8 分别给出了 BitwiseInfluenceMap 类的头信息和实现体。

程序清单 19-7　BitwiseInfluenceMap 头

```
class BitwiseInfluenceMap:public InfluenceMap
{
public:
    //constructor/functions
    BitwiseInfluenceMap():InfluenceMap(IM_BITWISE){}
    ~BitwiseInfluenceMap();
    virtual void Update(float dt);
    virtual void RegisterGameObj(GameObj* object);
    virtual void DrawTheInfluence();
    virtual void StampInfluenceShape(int* pMap,Point3f& location,
                    int sizeX,int sizeY, int value, bool undo = false);
    int GetVelocityDirectionMask(GameObj* object);
    int GetInfluenceType(int* pMap,Point3f& location);
    int GetInfluenceDirection(int* pMap,Point3f& location);
};
```

该类和其他的类基本相似，稍微有一点区别。这些不同点主要是为处理数据的逐位访问。在这里标记函数使用了逻辑操作，尽管当前实现不需要使用反标记(因为每次更新都清空地图)，标记还是具备了撤销标记的能力。

这里提到的调试函数 draw()和以前有所不同，因为它绘制出了每个 IM 元素三方面的数据，方块的底半部分表示方块中的对象类型，如果左上部分着色表示对象在左右运动，如果右上部分着色表示对象在上下运动。真正游戏的调试系统通常使用更说明问题的可视化调试手段，而不是简单的颜色，比如状态图标或者文本输出，但对于一个测试应用来说，颜色就足够了。

程序清单 19-8　BitwiseInfluenceMap 主要函数的实现

```
//--------------------
void BitwiseInfluenceMap::Update(float dt)
{
    //bail out if nobody to update
    if(m_registeredObjects.size()== 0)
        return;

    //clear out map
```

```
    memset(m_map,0,m_numCels*sizeof(int));

    //stamp new data
    RegObjectList::iterator listObj;
    for(listObj=m_registeredObjects.begin();
        listObj!=m_registeredObjects.end();++listObj)
    {
        RegObj* temp = *listObj;
        //have to update the bits, since you can
        //change direction continuously
        temp->m_objType = (char)temp->m_objType |
                          GetVelocityDirectionMask(temp->m_pObject);
        StampInfluenceShape(m_map,(*listObj)->m_pObject->
                    m_position,(*listObj)->m_objSizeX,(*listObj)->
                    m_objSizeY,(*listObj)->m_objType);
        (*listObj)->m_stamped = true;
        (*listObj)->m_lastPosition = (*listObj)->m_pObject->
                                                    m_position;
    }
}

//---------------------
void BitwiseInfluenceMap::RegisterGameObj(GameObj* object)
{
    int sizeX,sizeY;
    if(object->m_size <4)
    {
        sizeX = m_dataSizeX/16;
        sizeY = m_dataSizeY/16;
    }
    else if(object->m_size<11)
    {
        sizeX = m_dataSizeX/10;
        sizeY = m_dataSizeY/10;
    }
    else if(object->m_size<33)
    {
        sizeX = m_dataSizeX/8;
        sizeY = m_dataSizeY/8;
    }
    else if(object->m_size <49)
    {
        sizeX = m_dataSizeX/5;
        sizeY = m_dataSizeX/5;
    }
    else if(object->m_size <65)
    {
        sizeX = m_dataSizeX/4;
```

```
        sizeY = m_dataSizeX/4;
    }
    else
    {
        sizeX = m_dataSizeX/3;
        sizeY = m_dataSizeX/3;
    }

    //set minimum size of 1 in each direction
    sizeX = MAX(1,sizeX);
    sizeY = MAX(1,sizeY);

    RegObj* temp;
    temp = new RegObj;
    temp->m_objType = object->m_type;
    temp->m_objType |= GetVelocityDirectionMask(object);
    temp->m_pObject = object;
    temp->m_objSizeX = sizeX;
    temp->m_objSizeY = sizeY;
    temp->m_lastPosition = object->m_position;
    temp->m_stamped = false;
    m_registeredObjects.push_back(temp);
}

//--------------------
int BitwiseInfluenceMap::GetVelocityDirectionMask(GameObj* object)
{
    int velDir = 0;
    if(object->m_velocity.x() > 0)
    velDir |= DIR_RIGHT;
    else if (object->m_velocity.x() < 0)
    velDir |= DIR_LEFT;
    if(object->m_velocity.y() > 0)
    velDir |= DIR_UP;
    else if (object->m_velocity.y() < 0)
    velDir |= DIR_DOWN;
    return velDir<<8;
}

//--------------------
void BitwiseInfluenceMap::DrawTheInfluence()
{
    glPushMatrix();
    glDisable(GL_LIGHTING);
    glTranslatef(0,0,0);
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ONE);
    for(int i=0;i<m_numCels;i++)
```

```
{
    if(m_map[i])
    {
        int y = i / m_dataSizeY;
        int x = i - y*m_dataSizeY;
        //determine color for type
        Point3f color(0,0,0);
        for(int index = 0;index<8;index++)
        {
            int bitset = (m_map[i] & (1<<index));
            if(bitset)
                color += colorArray[index];
        }
        glColor3f(color.x(),color.y(),color.z());
        glBegin(GL_POLYGON);
        glVertex3f(x*m_celResX,y*m_celResY,0);
        glVertex3f(x*m_celResX,y*m_celResY+m_celResY*0.5f,0);
        glVertex3f(x*m_celResX+m_celResX,y*m_celResY+
                                        m_celResY*0.5f,0);
        glVertex3f(x*m_celResX+m_celResX,y*m_celResY,0);
        glEnd();

        color = Point3f(0,0,0);
        //get colors for direction
        int direction = m_map[i]>>8;
        if(direction & DIR_LEFT)
             color = colorArray[COLOR_SILVER];//left
        if(direction & DIR_RIGHT)
             color = colorArray[COLOR_PURPLE];//right
        glColor3f(color.x(),color.y(),color.z());
        glBegin(GL_POLYGON);
        glVertex3f(x*m_celResX,y*m_celResY+m_celResY*0.5f,0);
        glVertex3f(x*m_celResX,y*m_celResY+m_celResY,0);
        glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
                                            m_celResY,0);
        glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
                                          m_celResY*0.5f,0);
        glEnd();

        color = Point3f(0,0,0);
        if(direction & DIR_UP)
            color = colorArray[COLOR_OLIVE];//up
        if(direction & DIR_DOWN)
            color = colorArray[COLOR_TEAL];//down

        glColor3f(color.x(),color.y(),color.z());
        glBegin(GL_POLYGON);
        glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
```

```
                                                m_celResY*0.5f,0);
            glVertex3f(x*m_celResX+m_celResX*0.5f,y*m_celResY+
                                                m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,y*m_celResY+
                                                m_celResY,0);
            glVertex3f(x*m_celResX+m_celResX,y*m_celResY+
                                                m_celResY*0.5f,0);
            glEnd();
        }
    }
    glDisable(GL_BLEND);
    glEnable(GL_LIGHTING);
    glPopMatrix();
}

//--------------------
void BitwiseInfluenceMap::StampInfluenceShape(int* pMap,Point3f&
                        location,int sizeX, int sizeY, int value, bool
                        undo)
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;

    int startX = gridX - sizeX/2;
    if(startX < 0) startX += m_dataSizeX;
    int startY = gridY - sizeY/2;
    if(startY < 0) startY += m_dataSizeY;

    for(int y = startY;y<startY + sizeY;y++)
    {
        for(int x = startX;x<startX + sizeX;x++)
        {
            if(undo)
                pMap[(y%m_dataSizeY)*m_dataSizeY + (x%m_dataSizeX)]
                    &= ~value;
            else
                pMap[(y%m_dataSizeY)*m_dataSizeY + (x%m_dataSizeX)]
                    |= value;
        }
    }
}
//--------------------
int BitwiseInfluenceMap::GetInfluenceType(int* pMap,
                                                Point3f& location)
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;
    return pMap[gridX,gridY] & 0x0f;
}
```

```
//---------------------
int BitwiseInfluenceMap::GetInfluenceDirection(int* pMap,
                                                Point3f& location)
{
    int gridX = location.x()/ m_celResX;
    int gridY = location.y()/ m_celResY;
    return pMap[gridX,gridY] >> 8;
}
```

19.3.6　逐位 IM 测试平台的使用

图 19-5 给出了逐位系统以及其在测试平台游戏中的运行情况。只要任取例子中所跟踪的变量提供给主飞船的 AI 系统，都能获得好处。通过检测正在靠近的小飞船的 IM，并使用通用方向标识，主飞船能控制自身方向，更好地进行躲避。行进方向这个参数同样也可以用于指示是否有小飞船将会靠近，因为 Evade 状态的主飞船可以观测正离开主飞船的小飞船。正在逃逸和靠近的小飞船都可以使用在 IM 中记录的通用行进方向。人类很少直接飞向小飞船，因为他们知道这样很快就会被击落。他们通常从安全方向靠近，然后转向并射击。如果在该逐位系统中跟踪其他的变量，那么很多行为都将会被迫离开。

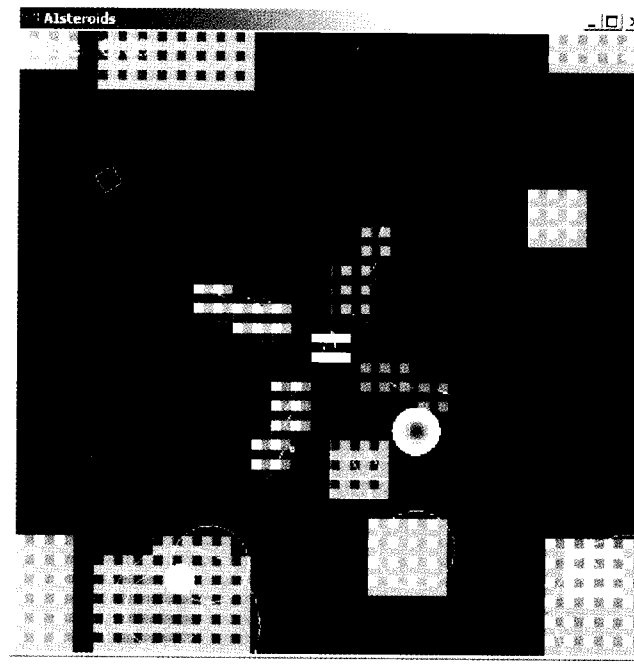


图 19-5　逐位 IM 在 AIsteroids 测试平台中的使用图

19.3.7　其他实现

前面我们所实现的 IM 示例仅仅是简单影响图示例的一部分。下面给出测试平台中的其他一些示例。

- IM 中跟踪的危险指数可使主飞船更有效地躲避，如果有小型慢速小飞船靠近，该指数较低，如果有大型快速小飞船靠近，该指数较高。这个示例有点类似于控制影响图，但更适合于躲避行为。
- 如果游戏中包含更多的能量提升瓶、敌对主飞船、环境对象(比如静止的小行星或者黑洞)，AI 系统就需要更复杂的状态来处理这些对象。一个复杂的 IM 既可用于更具针对性的不同路径搜索任务，也可用于为有复杂交互需求的对象提供控制信息，也可用于作为地图分析平台。
- 如果游戏世界变得更大或者其形状更加特别，IM 数组将是进行游戏对象搜索的好地方，这是由于探路者能获得比当前使用的简单“Closest asteroid or powerup”系统更好的目标。IM 会被打上连接数据的标签，因此搜索对象就能考虑覆盖边界，轮廓不规则的游戏世界也就能同规则的游戏世界一样被覆盖。
- IM 数组可以保持小段时间内(10 秒足够了)游戏对象的位置记录和运动方向。主飞船可以利用这些信息来避免进入快飞区，或者计算下一时刻小飞船的来向，然后攻击它。该系统只有在游戏中所剩小飞船不多或者只存在快速飞行的小飞船时才采用，主飞船也就不会为了设法找到游荡的小飞船而做出不必要的行动。
- 在测试平台中使用智能地形技术需要下面 4 个条件。
  - 需要对 GameObj 类进行扩展，扩展后可取名 SmartObj，在该类中包含了 Update()函数，该函数用于完成消息的广播，这些消息主要包括对象的类型和其他的一些信息，比如主飞船的位置、主飞船和主主飞船之间的距离和一些优先级数值等。每个 SmartObj 另外需要 Interact()函数，主飞船对象通过调用该函数来适当地和每个对象进行交互。对象的交互是上下文相关的，因此结果可能是射击或者躲避，而能量提升意味着靠近该对象。值得注意的是，对象不需要被删除，它可在游戏世界中静态创建，任何时候它都能在一定程度上有效。
  - 主飞船的新决策系统监听输入的消息，并根据当前主飞船状态选择主要对象进行交互，简单 FSM 就可以完成该决策系统的任务。基于其他一些因素，主飞船可能需要和其他对象进行交互(比如游戏中有 6 艘小飞船，那么主飞船需要分别调用这 6 艘小飞船的 Interact()函数，调用的结果可能不同)，因此主飞船需要保持一个敏感对象的列表，每次依次调用列表中所有对象的 Interact()方法。如果交互过程相当复杂，那么该智能对象的 Interact()方法可以采用 FSM 或者某种形式的脚本来设计。
  - 为了达到全面智能，每个对象都必须包含和主飞船交互时所需的所有必要的代码和数据。比如游戏中主飞船进行能量提升时播放的动画，该动画必须在与之交互的对象中存在。另外还有一些例子，比如声音效果、能量提升效果和其他一些游戏中需要的代码等。
  - 某些处理对象交互的游戏代码需要删除。因此，函数 GetPowerup()需要从主飞船类中删除，另外还有很多行为代码也需要从主飞船类中删除，移入智能对象的交互函数。这看起来有点倒退的感觉，但这样做可以降低游戏各部分的耦合度。我们应该知道在处理完上述工作后，一旦需要加入新的能量提升瓶、新的空间怪物和敌人，所有的工作就只是创建智能对象。

19.4　基于位置的信息系统的优点

LBI 系统是针对游戏的通用接口系统，任何位置数据都可以建立在该系统上。LBI 具有良好的直观性、编程方式和伸缩性。LBI 系统的调试相当简单，可视化反馈的使用也相当直观，这可从演示系统中看出来。

通常 IM 系统倾向于通过降低 AI 决策时所用数据的分辨率来简化感知的搜索空间，并通过共享知识库的形式供 AI 角色使用。因此主飞船自身不需要计算，只需简单查询 IM 系统，就能获得有关小飞船的信息，可在更短的时间内做出更智能的决策。

19.5　基于位置的信息系统的缺点

尽管如此，LBI 系统也有缺点。LBI 系统所需 IM 数组的代价很大，特别是需要大数据、大游戏空间和高分辨率的游戏更加明显。实现 IM 系统往往需要采用高明的技巧，为降低数据大小在不同情况下采用不同分辨率，只在需要时采用局部可重定位的高分辨 IM，而不是采用全局高分辨率 IM。

地形分析技术需要大量的计算资源，因为我们需要通过对数组的搜索来获得所需的信息。但几乎所有的模式匹配算法都很消耗计算资源并且错误率很高。正因为如此，到目前为止手写识别依然没有被广泛应用。

19.6　范例扩展

本章所实现的 LBI 系统非常基础。在游戏中使用本章提到的基本原则时，需要考虑的因素包括游戏的类型、IM 中数据的大小以及为在地图中查找适当模式所花的 CPU 时间。LBI 技术几乎适合所有的游戏类型。在第一/三人称射击(FTPS)类游戏中，这些技术可用于 king-of-the-hill-style 比赛的操作跟踪。即时策略(RTS)类游戏最适合采用这些技术，游戏中不同位置和不同任务的对象可以共享使用 IM，甚至经典的冒险类游戏都能使用 LBI 技术，我们可以跟踪玩家点击鼠标的位置。如果玩家不停地到处点击或者在同一位置反复点击，他极有可能遇到问题并且需要一定的场景帮助。

19.7　优化

LBI 技术需要在 IM 中进行大量猝发性存储操作，因此读写 IM 的问题有点类似于早期图形引擎上的问题。在进行 IM 操作时，我们总是修改数组中的部分数据，然后读取整个数组。因此很多针对图形的优化策略可以应用到 IM 优化中，比如我们在占用 IM 实现中涉及的脏矩形(dirty rectangle)技术正来自于早期的图形优化策略。在对象移动时，我们只需更

新 IM 中的部分单元，而不是所有单元。其他一些优化策略包括针对游戏平台数据总线的宽度设计数据单元的大小，提高数据传输速度以及 Cache 的命中率。

本章其他部分提到的优化策略，比如不同 IM 采用不同的分辨率以及只针对局部 IM 采用高分辨率等策略，可以帮助设计人员节省存储空间和计算资源。

地形分析技术的优化在不同的游戏中不尽相同，主要的影响因素有：地形分析技术的作用、IM 搜索范围、搜索模式类型和地形分析任务的更新频率。

19.8　设计上考虑的因素

LBI 通常在 AI 成分比较重的游戏中出现，比如即时策略类游戏、第一/三人称射击类游戏和角色扮演类游戏，因为这些游戏中的 AI 角色需要具备很多行为驱动类游戏中的 AI 角色所不需要的智能水平。LBI 具有开放的位置信息系统结构，支持被证实有效的查找方法对 LBI 中数据的利用。

19.8.1　解决方案的类型

LBI 可用于解决战术和战略类 AI 问题。在战术方面，IM 可以帮助搜索路径和动态规避。它可以给游戏角色的行为提供辅助提示，使得其看起来更加智能。在战略方面，AI 系统提供了模式匹配技术更好地利用地形，更好地规划大规模的战役和更好地修建城镇。智能地形系统中的对象更多的是战术，因为这些 AI 角色中没有加入很多战略智能技术。我们很难在 Sims 中看到 AI 角色计划的前瞻性。这些角色只会漫无目的地晃悠，等待附近对象的帮助。诚然，在 Sims 中这些角色也能工作挣钱，但是这都在游戏机制实现时定义好的，角色本身并不会因为需要购买某些物品而去挣钱。这是由于每个对象是一个“孤岛”，它们并不知道游戏中的其他对象，除非在游戏开发时设计好的。

19.8.2　智能体的反应能力

LBI 是一种辅助系统，因此能否提高智能体的反应能力是影响其能否被主 AI 系统所采用的主要因素。如果某个 LBI 系统能使得角色的反应更具前瞻性，那么就可以考虑使用该 LBI 系统。

19.8.3　系统的真实性

具有 IM 系统的游戏本身并不比其他游戏更加真实(人们通常具有强大的局部记忆能力，而不是通过标记来完成感知数据和记忆的定位；微波一般都是广播，而不是单纯传播到某个特定位置)，但是 IM 技术和地形分析技术的正确使用可以给游戏效果带来更多的真实性，使得游戏角色的决策更像人类。局部地图信息的利用使得 AI 角色的行为更加像人，而不像一般的机器人了解全局信息，这看起来有点像“作弊”。智能地形技术并不会提高系统的真实性，但是这些技术使得游戏环境更加丰富，新对象的加入更加方便。另外，它们完成了对对象和环境单元的抽象，而这都类似于人类的行为。

19.8.4 游戏类型和平台

通常前面提到的游戏类型——即时策略类游戏、第一/三人称射击类游戏和角色扮演类游戏都会采用 IM 系统和地形分析技术。尽管到目前智能地形在很少的章节中出现，但是其还是很具通用性的。任何需要严格交互的游戏，不管这些交互来自于对象还是环境，都能从智能对象系统中获得好处：智能对象给 AI 决策系统的编写和系统的扩展都带来了便利。在使用具有 LBI 的平台时唯一需要考虑的问题是 IM 数组的存储，但如果经过精心筹划和优化，这个问题可以克服。

19.8.5 开发限制

开发限制不是进行 LBI 系统设计时要考虑的问题。LBI 信息可以用于游戏的调试，IM 可以降低 AI 角色和游戏其他部分的耦合度，使得 AI 系统变得更加模块化，智能地形对象支持模块化实现，所有这些技术都具有较高的调试性和扩展性。

19.8.6 娱乐限制

调整参数的设置、平衡各种游戏行为和其他有关娱乐的问题都和 LBI 系统的使用无关，因此在决定是否采用 LBI 不需要考虑这些问题。

19.9 小结

基于位置的信息系统在提供多种决策模式的同时带来了很多好处：为位置专用数据的处理提供了灵活性，通过集中提供位置数据和交互处理逻辑降低了 AI 角色和游戏其他部分的耦合度。

- 本章覆盖了 LBI 系统中的 3 种主要技术：影响图、智能地形和地形分析技术。
- IM 是一种采用 2D 网格来表示游戏世界的通用数据结构。IM 中的数据，甚至 IM 本身完全独立于实现方法。
- 智能地形技术是一种有效利用游戏世界的地形和游戏对象分布特点的技术。该技术为其中的游戏角色提供了模块化和高扩展性的游戏世界，但它同时限制了所有游戏角色能执行交互的数量。
- 地形分析技术是一组利用地形数据完成有效模式的搜索以做出更好决策的方法。
- 本章提供的演示实现包括 4 部分：基本 IM、占用 IM、控制 IM 和逐位 IM。
- 除了本章中的演示实现，LBI 方法的实现可以采用很多其他方式。
- LBI 系统的优点包括实现和调试的简单性、通用接口和集中化的 AI 数据。
- LBI 系统的缺点包括 IM 数组所需的大容量存储空间以及复杂地形分析可能带来的高昂计算代价。
- IM 函数的优化可以采用在早期图形程序中经常使用的大数据量相邻数组的读写优化技术。

第Ⅳ部分 高级 AI 引擎技术

本书第III部分主要讲述了在商业游戏的 AI 编程中会用到的一些通用技术，而第Ⅳ部分将深入研究一些更为特殊的方法，这些方法来自于学术界，正逐渐在游戏开发中采用。

首先将研究两种特殊技术：遗传算法和神经网络。与第III部分内容的展开方式相同，我们首先给出所讲述的技术的主要内容，接着在 AIsteroids 测试平台中实现其框架代码，然后再深入讨论，给出该技术的优点、缺点、范例扩展和优化。

在阐述完遗传算法和神经网络后，我们将在第 22 章“其他技术备忘录”中讨论另外的高级 AI 技术，但不在测试平台中加以实现。这些技术都是将在 AI 游戏设计中迅速应用的前沿技术。我们将全面讨论这些技术，并给出每种技术的优缺点。同时，在 CD-ROM 中通过超链接的形式提供了很多这些技术的相关资源，而相关技术的来源和演示代码也可从 CD-ROM 中提供的书籍和论文中获得。

20 遗传算法

我们经常会遇到很难解决的 AI 问题，原因可能包括两个方面，一方面可能是因为问题的计算相当复杂和困难，另一方面可能是因为问题求解过程相当耗时。我们需要考虑太多的可能反应和太多的输入变量。尽管我们有可能找到问题的解决方案，但这需要我们进行反复的迭代，包括手工尝试。迭代过程相当烦琐，甚至可能需要成百上千轮次。举个例子，在调试游戏大卡车 4(Gran Turismo 4)中各车的性能参数时我们需要不断地进行物理仿真。假如游戏提供了 500 多辆汽车，每辆汽车的操控系统具有若干可调参数，那么几乎没有一家公司能完成这些参数的调试工作(至少在合理的时间和预算内没法完成)。假如游戏中各汽车的参数需要精确地模拟现实中的汽车性能，那么该问题将变得更加难以解决。

20.1 遗传算法概述

本章将讨论一种称为遗传算法(英文简写 GA)的 AI 技术，该算法的思想来源于生物进化学，该算法为我们提供了问题解决的新思路。我们将通过讨论自然进化模型给出算法的基本方法，然后演示如何把该模型应用在游戏中，最后将给出一个简单的 GA 类，并在 Alsteroids 测试平台中加以实现作为该算法的示例。

20.1.1 自然进化规律

GA 技术借用了自然界生物进化的主要思想来给出解决问题的算法。自然进化的主要过程包括以下几个要点：

- 为了维持物种的延续性，所有的生物都具备生殖能力。简单地讲，生殖就是通过一定规则创造新个体的过程。这些规则通过染色体的方式保存在 DNA 中，DNA 存在于每个细胞中，而大量细胞有机组合构成了生物。
- 染色体由基因序列构成，基因通过 4 种基本蛋白质排列而成，这 4 种蛋白质分别是：胸腺嘧啶、腺嘌呤、胞嘧啶和鸟嘌呤，通常用 T、A、C 和 G 分别代表它们。每个基因所包含的信息都代表着某种或某组生物特征。

- 当父代生物个体进行生殖时，他们的 DNA 分裂后传递给后代个体，因此每个后代的一半 DNA 来源于父代雄性个体，另一半来源于父代雌性个体，这个过程称为基因的交叉(crossover)和重组(recombination)。
- 基因的交叉将在后代个体中表现出新的混合特征。如果该特征适应自然条件，那么具有该特征的个体将得以存活、继续生殖并传递至少一半该特征相关的基因给下一代。如果继承了该特征的后代个体的体质较弱或者不适应自然条件，那么它将很难生存，或者不能繁殖后代。不能繁殖后代可能源于某些生物因素，比如没有生殖能力；也可能源于社会因素，比如找不到生殖配对对象。代复一代的发展，那些具有优良生物特征的个体不停地繁殖后代，而那些具有低劣生物特性的个体慢慢地被淘汰，导致整个生物群体向优质方向演化。通过一个称为适应度的变量来评估生物对环境的适应程度，适应度数值越大，个体表现出来的生物特征越适应环境，这些特征包括生存能力和生殖能力两方面。但对人类社会来说，一个取得巨大成功的男人在传种接代的竞争中依然有可能是个失败者，这主要是由于他把精力都花在了工作上，而没有时间来生小孩并传递其基因。因此生物基因的遗传和生物的进化是生存能力和生殖能力共同作用的结果。
- 偶尔进化过程也会出现点“瑕疵”，尽管有关该“瑕疵”是否是进化过程不可分割的一部分的争论还没有停止。后代个体的基因有时会出现变异，变异后的后代个体就有可能是一个全新的个体，其部分基因没有出现在父代个体中。该变异来源于基因复制过程中的故障，或者受精时的化学平衡的破坏，或者其他的一些原因，到目前我们依然没有弄明白其中的机理。我们把该基因上的变异称为基因突变或者基因变异(mutation)，变异后原基因位上的等位基因是随机的，因此该基因对生物个体特征的影响也是随机的。在多数情况下，基因突变将导致原有特征的阴性特征。在发生基因突变时，鸟儿的翅膀可能会变短而不能飞翔，树獭的头部可能会出现棕色斑点而造成其他树獭不愿和其交配，猴子可能会听到频率极低的声音而造成夜里打寒战时神经错乱。
- 有时基因突变会在后代获得某些特征的进化而使得个体更加适应环境或者更具生殖能力。这种突变得到的基因将通过生殖过程不断地传递给下一代。
- 因此，“适者生存”模式将不断地改变物种的特征集或者基因组，使得该物种的平均适应度最终达到理想值，也就是该物种达到最适应当前环境的状态。

20.1.2　游戏中的进化

那么进化能给游戏 AI 带来什么呢？我们发现该进化算法的实现能满足游戏中的某些需求，它可用于调试游戏行为、参数和其他一些手工很难调试的因素，比如游戏区域等。该过程被称为“进化搜索”，该过程同样也需要完成对给定游戏问题的搜索空间进行全面的搜索，我们通过对生物进化过程的模拟给出一种基于适应度进化的搜索算法。

该算法可分为实现代价完全不同的两部分：问题的进化和结果的使用。通常，在游戏中遗传算法得到的信息的使用可看作“黑盒操作”。这是具有“魔力”的盒子，其能针对某些特殊问题给出尽可能的最优解，或者说所能搜索到的问题空间中的最优解。问题的进化几乎是全部的工作，该部分工作在游戏产品的开发过程中完成。很少有游戏在发布版本中还包含遗传算法的进化部分的内容。当然也存在一些例外的游戏，比如某些具有机器学习模块的游戏。这些游戏的机器学习模块也处于严密监控之下，具有学习能力的AI行为和特征在设计的过程中都内建了机器学习模块，也就是说这些行为和特征的学习进化和游戏的其他逻辑不相关，因此学习和进化过程能得到很好的控制和隔离。考虑这些游戏中的AI角色的行为很难预测(学习过程的不可预测性)，我们允许其出现一定程度的非正常行为。

遗传算法的实现需要相当大的计算代价，该过程相当耗时(算法需要运行很多代的进化过程，而且各代群体的基数很大)，正因为如此，过去AI游戏很少选择遗传算法，也正因为如此，进化过程基本都是离线完成的。随着计算机性能的提高，遗传算法逐渐成为主流技术。

遗传算法属于随机搜索算法，该系列算法还包括模拟退火算法等，随机搜索算法的有效性取决于进化元素的随机变化和搜索方向正确选择的概率。这类算法的搜索过程需要不停地迭代(因为我们不知道随机元素是否会造成搜索过程步入歧途)，同样我们也不能专注于问题的某个特殊“解”(很多应用中会出现很多基因组，虽能提供良好的解，但这些解却并非最优解，这些解可用一个术语表示：“局部最优解”(local maximums)，我们需要进化元素的随机变化能避免搜索过程收敛于局部最优解)。

和很多的搜索系统不同，遗传算法和问题本身是不相关的，算法可以在很多不同的数据结构上工作，这样的工作模式使得遗传算法也可应用于变量类型混合的系统中。尽管在遗传算法中表示基因的最通用的办法是位串，设计人员依然可以采用其他的任何数据结构(包括数组、树等)，前提是利用该数据结构表示的个体特征是问题的完整的解并且我们可以设计出基于该数据结构的基因操作，比如交叉、变异等。

在利用遗传算法进行问题求解时，需要注意的是该算法并不能保证执行的性能以及能否找到问题的最优解。事实上，遗传算法有时会以最糟糕的形式执行，给出最糟糕的结果。这是由于在算法执行过程中我们经常性地引入随机因素。当算法实现不能给出所期望的结果时，设计人员需要调整基因结构、参数设置和基因操作等。尽管可以通过调整参数等来改进算法的性能，但还是有很多设计人员叫苦说遗传算法不适应他们遇到的问题。

20.1.3 遗传算法基本过程

采用遗传算法解决AI问题可分为以下3步：初始化、评估和新个体的产生。

1. 群体的初始化

群体的初始化一般采用随机初始化的方式，考虑到问题空间的某些特定信息时可赋予个体符合问题要求的数据。初始群体可以是任意大小，主要依据是实践经验和可用资源，比如进化过程可以多长，需要怎样的结果等。

2. 个体对问题空间适应度的评估

所有个体都需要通过执行特定的适应度函数来完成评估并返回结果数值(或者向量)来表示该个体的适应度。适应度函数的计算相当耗时，这也正是遗传算法计算代价高昂的主要原因。如果仅仅是针对单个体单游戏循环后的适应度计算，整个过程是相当快的。但在游戏中，群体中的所有个体都需要执行多个游戏循环后才能产生其适应度分值。假如每次适应度的测试需要个体在游戏中运行 5 分钟，而群体中个体的数量为 100，那么总需 8.33 个小时才能完成群体的一次进化，而典型的遗传算法需要执行成千上万次进化才能获得有效解。因此游戏仿真时间的可调性(通过较短的实际时间模拟较长的游戏时间)对于加速进化过程是有益的。但是一个经过加速的游戏过程和真实的游戏过程是不同的，比如物理冲突检测过程中可能会由于各帧之间的时间间隔 Δ 太大而丢失冲突，因此经过加速的游戏仿真确实可以得到一定的好处，但是其并没有真实时间下的游戏仿真来得有效。另一个相关技术是用遗传算法代替游戏中某些特定的决策过程，这样由于遗传算法离线执行得到最优结果使得决策过程相当地简单并对时间就不敏感了。

3. 新个体的产生

一旦群体中的所有个体完成了适应度的计算，部分个体将会被选择进行生殖。选择也是进化过程中很重要的一部分。如果只选择那些适应度最高的个体，可能会迅速收敛于局部最优解，因为太多的基因被排除掉。如果选择过程太随意，可能永远也没法取得最优解，因为这样可能引起进化过程太多的跳跃。在本章后面部分中将给出一些选择方法。一旦父代个体选择完毕，这些个体将生殖得到下一次进化的候选者。下一代的产生采用了 3 种普通方法：交叉(或者说性别生殖)、变异(或者说基因突变)和精英主义(elitism)(也就是说上一代个体中适应度最高的个体直接进入下一代，不经过生殖阶段，有点像克隆)。对 3 种方法混合方式的选择以及各种方法具体操作的确定都需要反复地实验和专业背景知识。本章将讨论很多不同的操作方式。

20.2　问题的表示

一般遗传算法都采用问题本身的语言来完成复制和进化的定义。在这里将设计一种和遗传学兼容的方式来表示问题，并给出新的基因操作定义来完成问题的抽象表示。

20.2.1 基因和基因组

在采用遗传算法进行问题求解时，首先需要确定基因的结构、基因的特征类型和特征数量等。在确定基因特征时需要确定特征的表示类型，是采用二值数据还是采用实数表示特征，如果是实数，它的范围又是多少？另外还需要确定的是问题中各个特征之间是否存在关系。

我们不应该忽视这些选择的重要性。在我们确定基因和基因组结构的同时，也就确定了问题的状态空间以及问题的精度。这好比我们在形式化某种语言时，其实也决定了通过该语言我们能得到的答案。因此假如我们在定义等位基因方向时只定义了 4 种被选方向，那么最后的解中也可能只是这 4 种方向。我们不必担心遗传算法的适用性，因为遗传算法不仅适用于离散类型数据，同样也适用于连续类型数据，只是作用于更大的搜索空间而已。

问题表示的选择过程是折中的过程。问题的搜索空间越大，所能得到的问题的解也越好，甚至于好到令人惊讶。然而，太大的问题搜索空间也有两个缺点：可能需要花更多的时间才能获得问题的解，或者通过遗传算法得到的最终解可能超过了期望值或者太过人性化。举个可能不太相关的例子，比如电影《指环王》的动画系统。由于该电影中存在太多的战争场面，动画设计师很难通过手工方式来完成所有场景的设计，因此他们采用了 AI 系统来控制战士。然而在通过 AI 系统创建人类、精灵和兽类(奥克斯)之间的大规模战争的场景动画时，由于奥克斯的数量占绝对优势，战斗一开始所有由 AI 系统驱动的人类和精灵都躲进树林中。这一切都归因于 AI 常识系统的默认定义太过理智，通过修改常识系统才能使 AI 驱动的战士具备原著中的战士所应有的意志和士气。遗传算法也经常会找到计算过程的漏洞而得到无效解。而如果遗传算法的问题搜索空间过大，就特别容易触发这样的问题。

位串是我们在基因串编码时经常采用的方法之一。比如游戏《吃豆先生》中的精灵的行动只有 4 种选择：向上移动、向下移动、向右移动和向左移动，每个游戏循环选择一种。精灵知道其当前路径方向上的状态，状态主要是指路径上存在什么物体，包括普通怪物、蓝色怪物、待清除的垃圾和墙。因此可以通过遗传算法来生成各种不同感知条件下精灵的行动。在该算法实现中，每个基因由位宽为 2 的位串构成，代表着可能的 4 种选择，基因组则可由一串基因组成，代表着精灵的反应序列。

另外还存在一种典型的基因类型，各基因是内容互斥，或者内容敏感的。这种基因的最常见的例子是经典的货郎担问题(Traveling Salesman Problem，TSP)。该问题的表述如下(见图 20-1)：对于给定的城市，每个城市只能访问一次，货郎如何依次访问这些城市才能使得所走的路最短。该问题的基因组显然是城市列表，但和前面所述的吃豆先生这个示例不同，基因组中每个基因的内容是互斥的，这是因为每个城市只能访问一次。从技术上讲，也可以采用和 TSP 类似的基因结构来定义示例游戏《吃豆先生》中的问题从而获得最短的吃豆路径。但如果我们这么做将面临一些问题，在精灵吃豆过程中怪物会随机出现在某些位置上，而精灵需要避开这些位置。因此在该例中我们一般采用第一种基因

来表示问题。事实上，我们应该知道这里的《吃豆先生》只是个用以说明遗传算法的问题表示的示例。由于问题相对简单，在商业游戏《吃豆先生》的 AI 系统的设计中，采用了比遗传算法更普遍的方法。一般地，遗传算法擅长于解决我们很难制定解决方案的问题。这些问题如果采用除遗传算法外的其他启发式算法，不是需要过多的计算资源，就是需要程序员花很多的时间对算法给出的数据进行判断取舍。我们将在本章“系统优点”这一节详细阐述该问题。

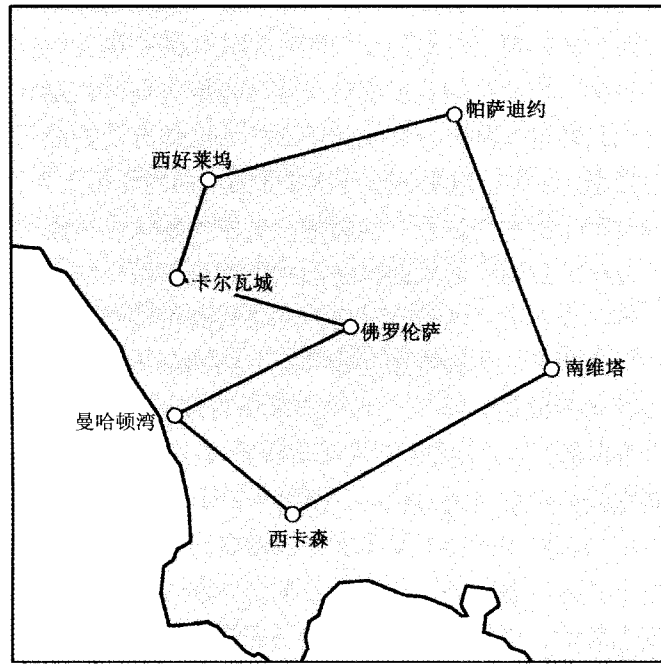


图 20-1　TSP 问题

另外可用于遗传算法基因组表示的数据结构还有数组和树。这两种结构都很容易处理，也都存在标准基因操作。这些基因组中的基因的数量既可以是固定的，也可以是可变的。最后我们需要注意的问题是：在进行基因组表示选择时应该考虑到所选的基因组表示既便于问题的全面表示，也便于更简单的基因操作。

20.2.2　适应度函数

在确定基因和基因组的格式后，我们需要给出对每个基因组进行性能评估的标准。适应度函数和问题领域是密切相关的。在游戏《吃豆先生》中，适应度函数可能需要考虑总分、吃豆速度、存活时间长度和蓝色怪物消灭数量等几个方面的因素。适应度函数需要综合考虑各因素，每个因素具有不同的权重，比如存活时间的权重大于吃豆速度和总分等。蓝色怪物消灭数量这个因素是额外的，因为我们在计算总分和存活时间这两个因素时已经算入了蓝色怪物消灭数量这个因素。各因素权重的不同意味着精灵的行动方式的不同。

适应度函数在遗传算法求解过程中相当关键，因此需要认真考虑其设计。适应度函数是我们搜索问题空间时的启发式函数。太多的参数和行为将弱化遗传算法的能力并延长算法的执行时间，太少的参数将使得遗传算法丢弃很多看起来不需要的个体而专注于某些使适应度局部最大化的个体。

完成基本适应度函数的设计后，通过执行该算法将得到适应度数值，该数值需要执行一定的变形运算以避免算法过早收敛(Premature Convergence，PMC)以及过早进入滞点(stagnation)。由于算法中个体数量过少而使得算法早期出现的异常个体传递其基因给后代个体并占绝对地位会造成 PMC 问题的出现。而如果群体中的很多个体具有相似的适应度，那么就很容易出现“滞点问题”。在该情况下个体之间的差异将达到极小值，至少要小于选择过程的敏感度。这是我们不愿意看到的，因为在该情况下算法不存在很大的选择压力。适应度数值的变形能使很多由基因组合带来的好处更加明显。下面给出了数值变形的几种通用方式。

(1) 求和截断

经过该种变形的适应度数值 $F' = F - (F\hat{} - c*\text{sigma})$。其中的 $F\hat{}$是适应度的平均值，sigma 是群体适应度的标准差，c 是合理的系数，其值介于 1 和 3 之间。如果变形的结果是负值，我们约定置 0。当然也可以简单地采用群体的适应度标准差来变形每个个体的适应度，但只有在群体中各个体差异较大时该方式才有效(比如模拟开始阶段)，而当模拟逐渐收敛个体之间的适应度差异较小时该方式几乎无效。

(2) 排序变形

在排序变形中，我们利用适应度排位取代适应度分数来评估个体的适应度。群体中适应度最低的个体的适应度分数为 1，适应度次低的个体的适应度分数为 2，而适应度最高的个体的适应度分数为群体的个体总数。值得注意的是采用该技术后的遗传算法需要更长的收敛时间。

(3) 共享变形

共享变形方式的出发点是促进基因变化，适应度分数相近的个体将被罚分。首先，该方法记录了每个基因在不同基因组中出现的次数以及基因组中存在共享基因的数目。然后，根据共享基因的数量对基因组进行分类，比如有 5 个共享基因的基因组归入第 5 组。最后，根据基因组集合中基因组的数量完成基因组的变形。

20.2.3 繁殖

遗传算法中包含了进化群体以及对该群体中各个体进行适应度评估的函数后，我们还需要给出进化过程所需的繁殖过程。当前业界存在两种主要的繁殖类型。第一种称为世代繁殖，在该种类型中上代个体在完成繁殖后退出进化群体，可以采用的方式有直接复制、交叉、变异或者直接取代。第二种称为稳态繁殖，在该类型中任何时候群体中通过交叉、变异等繁殖方式得到的个体数量有限，大部分保持不变。而稳态群体中个体的取代方式是另外一个课题，最常用的取代方式是最差者被淘汰，不过也存在其他的方式，比如随机淘

汰法、最大似然淘汰法和上代淘汰法等。我们把上代个体直接复制入下一代个体的技术称为精英主义，该技术可以帮助我们降低选择过程中最优个体误淘汰的概率。精英主义和适应度变形正好相反：它减少了基因之间的差异加快了进化的收敛过程，因此需要非常小心。太多的精英主义将造成最后解只会是局部最优而非全局最优，稳态繁殖实现不需要额外的精英主义过程。

其他选择方式包括以下几种：

(1) 轮盘赌选择法

基因组的轮盘赌选择法是按照适应度分数的比例进行的，适应度分数最大的个体被选中的概率最大。值得注意的是，高分个体可能会被多次选中，因为该个体在上轮被选中后还可以参加下轮的选择。但这都是在概率条件下进行的，因此适应度分值最高的个体依然有可能不会出现在下代群体中，这是为什么人们在使用该选择方式时配合使用精英主义的原因。

(2) 随机遍历抽样法

随机遍历抽样法是色环轮盘赌选择的简单说法。首先确定参加选择的个体数目，假如参加选择的个体为 n，则划分轮盘刻度时每个色环为 $1/n$ 圈，所以选中各色环的概率都是 $1/n$。该方式在保持不同的适应度数值和明显的基因差异等方面比普通轮盘更具优势。

(3) 锦标赛选择法

在该方式中，进化群体中的某些个体被随机抽出，适应度分数最高的进入下一代，然后选中的个体重新进入群体进行下一轮的选择。该选择过程可以反复迭代多次直到选定个体的数目达到要求。

在选定个体后，我们需要确定基因交叉的概率。模拟进化的过程，我们将用到交叉率这个参数，一般该参数在 0.7f 左右，如果觉得该数值不合适，可以修改它。交叉的过程如下：通过随机数生成器得到处于 0 和 1 之间的随机数，如果该数小于交叉率，那么使选中的两个个体完成交叉操作得到两个后代个体，否则不产生后代个体。在确定交叉操作时需要考虑到基因组采用的变量和结构类型以及良好的实验过程所需的交叉比重。

下面给出了一些二值变量上的交叉操作(见图 20-2)。

(1) 单点交叉

在基因组中随机选择一个位置，然后交换两父代个体中该位置后面的所有基因来产生后代个体。

(2) 多点交叉

在基因组中随机选择两个位置，然后交换两父代个体中处于两位置之间的基因来产生后代个体。

(3) 均匀交叉

均匀交叉又称“逐点交叉”，在交叉前先进行基因的变异检测，通过后再行交叉。

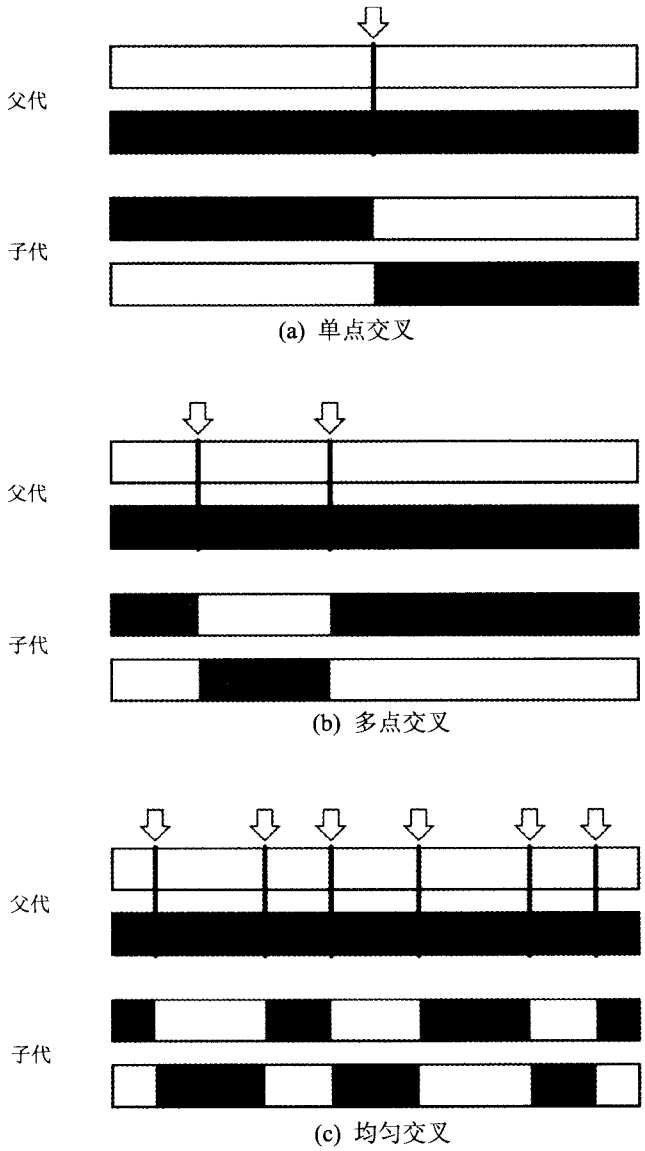


图 20-2　二值变量交叉操作

下面给出了一些连续数值变量上的交叉操作(见图 20-3)。

(1) 离散交叉

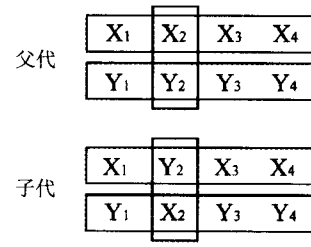
直接交换个体的变量数值。

(2) 中间交叉

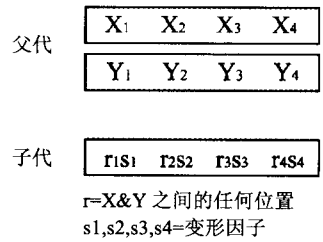
通过中间交换得到的后代个体的变量值处于两父代个体之间。计算公式如下：Offspring = Parent1val + Scale*(Parent2val – Parent1val)，其中系数 Scale 针对不同数值随机确定，其范围为(－d,1+d)。正常中间交叉时，d = 0，如果需子代个体变量值超过父代个体，那么 d > 0。

(3) 线交叉

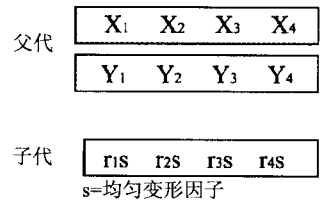
线交叉和中间交叉基本相同，只是不同的变量采用相同的变形。



(a) 离散交叉



(b) 中间交叉



(c) 线交叉

图 20-3　连续变量交叉操作

下面给出一些针对有序基因的交叉操作(见图 20-4)。

(1) 部分映射交叉

部分映射交叉有时又称“PMX”。该交叉操作的过程如下：首先从父代个体 1 的基因组中随机选择两个位置，确定该位置上的基因，再根据位置得到父代个体 2 的基因组上的基因，从而得到两组映射，比如图 20-4 中的 2<->5 和 3<->4，最后根据这两组映射完成交叉操作。在此过程中两父代基因组分别进行操作，比如图 20-4 中父代个体 1 的基因组 12345 根据映射交换后得到子代个体 1 的基因组 15432。

(2) 有序交叉

有序交叉操作的过程如下：首先在父代个体 1 中随机选择若干基因，确定它们的位置，然后根据位置在父代个体 2 中取得相应的基因，交换父代个体 1 的这些基因得到子代个体的基因组。

(3) 位置导向的交叉

位置导向的交叉操作和有序交叉操作有点类似，其过程如下：首先在父代个体 1 的基因组中随机选择若干基因，保持这些基因的位置相对不变，并复制到子代个体的基因组中，

该个体基因组中的剩余基因取自父代个体 2，在取得这些基因的过程中需要注意保持顺序不变而且保证基因不重复。

在完成基因交叉操作后，繁殖过程只差最后一步：变异。变异仅仅是对经过交叉得到的子代基因组进行一定的操作。基因变异的概率和实际的问题密切相关。根据相关文献([Bäck93], [MSV93])，位串问题的典型变异率为 0.0001f，而实数问题的典型变异率在 0.05f 和 0.2f 之间。而变异操作的类型和所采用的基因组结构有关。

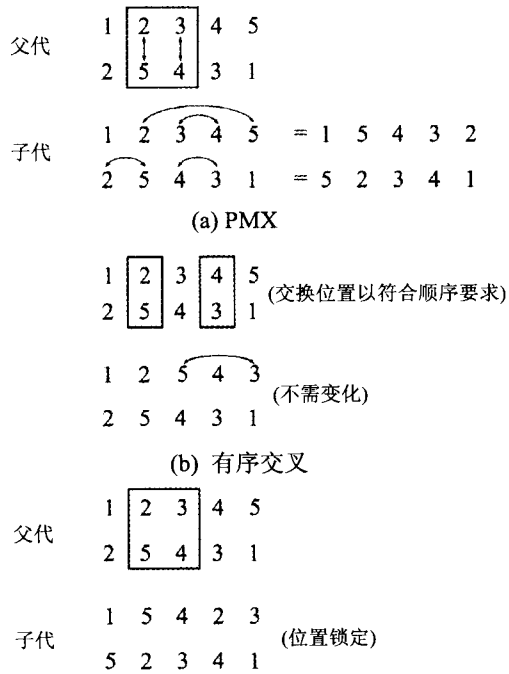


图 20-4　有序基因交叉操作

下面给出了在有序基因组结构中普遍采用的变异操作(见图 20-5)。

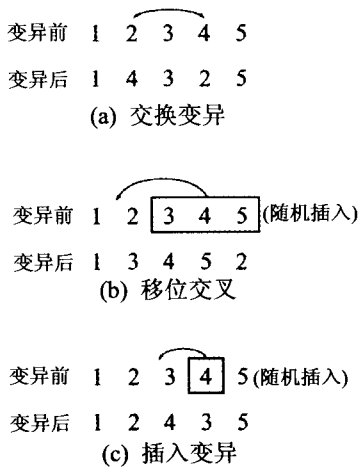


图 20-5　有序基因变异操作

(1) 交换变异

交换基因组中的两段基因。

(2) 移位变异

随机选择两个位置，整体移动两位置之间的基因。

(3) 插入变异

和移位变异类似，唯一的区别就是移动的基因只有一段。很多测试表明插入变异的效果比其他两种变异要好些。当然，读者也可能会遇到效果相反的情况。

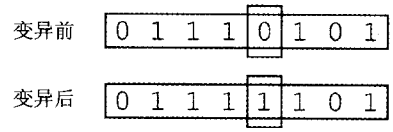
下面给出无序基因组的变异操作(见图 20-6)：

(1) 二进制数变异

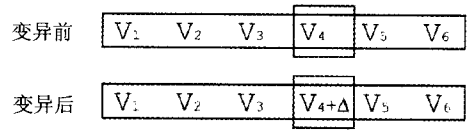
只需翻转基因组中的某一位。

(2) 实数变异

选择基因组中某段基因累加参数 Δ。Δ 的大小很难确定，步长太小虽然有效，但太费时。



(a) 二进制变异



(b) 实数变异

图 20-6　无序基因变异操作

20.3　Alsteroids 测试平台中遗传算法的实现

简单 Asteroids 几乎不存在需要通过遗传算法才能解决的问题。大部分决策问题可以归结为简单的数学问题，其他的问题则可以分解为一个个影响行为的状态。但为了说明遗传算法的实现过程，我们假设处于躲避状态的 AI 角色需要使用遗传算法。作为实现示例，我们利用遗传算法提高了 AI 主飞船的躲避能力。

该问题的基因设计比较简单，这主要是由于主飞船本身的移动模式比较简单。无论在任何时候，主飞船只能推进或者转弯。因此，我们所设计的基因组只包含两个字符：第一个字符表示推进的类型(前进、后退或者无操作)，第二个字符是 0 到 17 之间的无符号整数，指出当前状态主飞船应指的方向。在这里把以主飞船为中心的空间分成 18 个区域，每个区域占 20 度，用 18 个无符号整数分别表示。如果考虑到字符的表示范围，可能会出现非法数值，可以对该字符进行取余操作，这并不影响精度。

我们所设计的基因回答了问题“给定时刻主飞船的行动选择和方向选择”。在采用遗传算法进行该问题的求解前，我们需要完成问题状态的定义。由于本例中我们只完成主飞船躲避动作的 AI 控制，因此输入感知信息也只需包括躲避时要用到的信息。简单的躲避专用游戏状态只需考虑三方面的信息(所有的信息都和最近的小飞船和主飞船本身相关)：

- **两船的相对速度**。首先，我们定义了一个规格化 Δ 向量，该向量是所有小飞船和主飞船的规格化向量的汇总。小飞船和主飞船的相对速度通过累加两船速度向量在飞行方向上的分量得到(详见图 20-7)。该数值越大，两船靠近地越快。为了更便于表示，我们对该数进行了变形，使得变形后的数值落在 0 到 9 之间，这样就可以用 10 个碰撞状态来表示两船之间的相对速度。

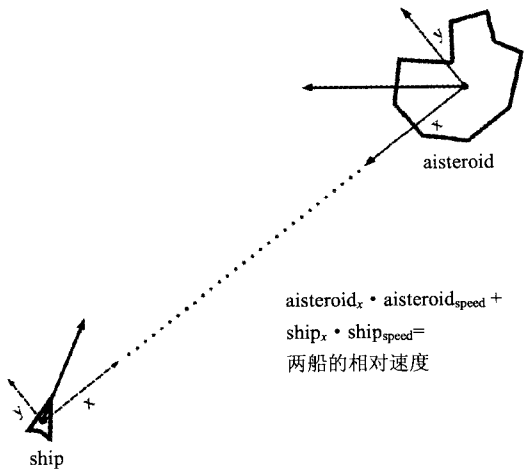


图 20-7　碰撞状态计算图

- **两船的飞行方向**。同样，我们也对该数值进行了量化，使得其可以用有限的几个状态表示。我们只是简单地计算向量所指的角度并变形，使得其数值落在 0~17 之间，共 18 个可能方向状态。
- **两船之间的相对距离**。最后需要知道两船之间的相对距离。只有当两船之间的距离相当近时，我们才启动躲避程序，因此我们也需要把相对距离量化成我们关心的几种可能的基本距离。我们利用需躲避的小飞船的长度来衡量两船之间的相对距离。假如主飞船和小飞船之间的相对距离为一个小飞船的长度，此时两船之间的距离设为 1，其他依次。只有当两船之间的距离为 4 或者更小时，飞船才开始关心相对距离，因此我们只需 4 个距离状态。

综上所述，我们定义了一组规则，给出了主飞船和最近小飞船的碰撞状态、方向状态和距离状态的定义。很明显主飞船和小飞船的状态空间为 520(10*18*4)。如果飞船知道自身处在这 520 种状态下所应有的反应时，其能很好地完成躲避任务。我们需要再次声明，主飞船的躲避系统最简单的实现方式是采用数学模型思想而不是本章给出的遗传算法。

接下来，我们将研究怎样利用遗传算法来解决上述问题。由于我们希望通过基因组直接表示问题规则，因此在基因组中包含 520 段基因。程序清单 20-1 给出了基因和基因组的头信息。

程序清单 20-1　基因和基因组头信息

```
class Gene
{
public:
    //methods
    Gene() {m_thrust = randint(0,2);m_sector = randint(0,NUM_SECTORS-
1);}
    Gene(int a, int d):m_thrust(a), m_sector(d){}
    bool operator==(const Gene &rhs) const {return (m_thrust ==
                    rhs.m_thrust) && (m_sector == rhs.m_sector);}
    bool operator!=(const Gene &rhs) const {return (m_thrust !=
                    rhs.m_thrust) || (m_sector != rhs.m_sector);}

    enum
    {
        THRUST_OFF,
        THRUST_FORWARD,
        THRUST_REVERSE
    };

    //data
    char m_thrust;
    char m_sector;
};

class Genome
{
public:
    //methods
    Genome():m_fitness(0){}
    Genome(const int num_genes):m_fitness(0)
                    {for(int i=0; i<num_genes;++i)
                        m_genes.push_back(Gene());}

    bool operator<(const Genome& rhs){return (m_fitness <
                                              rhs.m_fitness);}

    //data
    vector<Gene> m_genes;
    float m_fitness;
};
```

基因类和基因组类都非常简单。基因类中保存了我们在进化过程中所需的信息，而基因组类则是基因及其适应度分数的容器。在 Aisteroids 中，我们保存了解决碰撞问题的“计划”方案，形式是推进和转向角度动作序列。不管应用于什么类型，也不管设计中是采用位串还是实数来表示基因，基因类总作为定义遗传物质的类而存在。基因组类基本可以通用，而基因类作为底层类，需要针对不同的应用进行不同的实现。

我们将实现进化应用(Evolution Application，EA)，这是本书第一次编写可应用于商业游戏的规则系统。这部分工作主要包括 GameSession 类、HumanTestControl 类、AIControl 类、GAAIControl 类以及 GAMachine 类，其中的 GAMachine 类是所有游戏功能的容器类，也最为重要。

TestSession 类和 HumanTestControl 类仅仅是 EA 游戏端支持代码，负责处理输入信息、绘画功能和主游戏更新循环。测试器应用的控制操作包括标准的加速和减速按钮、单步按钮和复位按钮。应用中的会话也非常简单，其只负责不停地产生小飞船和主飞船，一旦出现碰撞现象就休眠这些飞船，而不是删除它们。最后在该代进化结束时(也就是所有的飞船进入休眠状态时)，复位所有的飞船并开始新一轮的进化。

进化算法的主题部分包含在 GAMachine 类中。程序清单 20-2 给出了其头。我们将在程序清单 20-3 到程序清单 20-11 中分别给出这些函数的实现体，然后依次对这些函数做些简单讨论。

有关 GAMachine 类的头信息，需要注意的是该类中包含了一些测试程序中用不到的函数。GAMachine 类中实现了两种选择类型、6 种交叉操作和 4 种变异操作。这些实现一方面向读者提供了可在 AI 编程中使用的工具函数，另一方面也为读者在测试平台中评估各种方案的优劣提供了便利。记住，遗传算法更多的是需要实践而非理论，确定各种基因操作的类型和关键参数(交叉率、变异率和精英主义总量)是遗传算法的难点。

程序清单 20-2 GAMachine 头

```
class GAMachine
{
public:
    GAMachine(GAAIControl* parent):m_parent(parent){}
    void SetupNextGeneration();
    void CreateStartPopulation();
    void Update(float dt);
    void UpdateFitness(int index);
    void Init();
    void Reset();
    void ApplyBehaviorRule(int index);
    bool WriteSolution();
    bool ReadSolution();

    //selection operators
    Genome& SelectRouletteWheel();
    Genome& SelectTournament();
    Genome& SelectRank();

    //crossover operators
    void CrossUniform(const vector<Gene> &parent1,
                      const vector<Gene> &parent2,
                      vector<Gene>&offspring1,
                      vector<Gene>&offspring2);
```

```
    void CrossSinglePoint(const vector<Gene> &parent1,
                          const vector<Gene> &parent2,
                          vector<Gene>&offspring1,
                          vector<Gene>&offspring2);
    void CrossMultiPoint(const vector<Gene> &parent1,
                         const vector<Gene> &parent2,
                         vector<Gene>&offspring1,
                         vector<Gene>&offspring2);
    //crossover operators - order based genes
    void CrossPMX(const vector<Gene> &parent1,
                  const vector<Gene> &parent2,
                  vector<Gene>&offspring1,
                  vector<Gene>&offspring2);
    void CrossOrderBased(const vector<Gene> &parent1,
                         const vector<Gene> &parent2,
                         vector<Gene>&offspring1,
                         vector<Gene>&offspring2);;
    void CrossPositionBased(const vector<Gene> &parent1,
                            const vector<Gene> &parent2,
                            vector<Gene>&offspring1,
                            vector<Gene>&offspring2);

    //mutation operators
    void MutateOffset(vector<Gene> &genes);
    //mutation operators - order based genes
    void MutateExchange(vector<Gene> &genes);
    void MutateDisplacement(vector<Gene> &genes);
    void MutateInsertion(vector<Gene> &genes);

    //elitism
    void CopyEliteInto(vector<Genome>&destination);

protected:
    GAAIControl* m_parent;
    //genetic data
    vector<Genome> m_genomes;
    int m_rankIndexLast;
    Genome m_bestGenome;
    int m_generation;
    float m_crossoverRate;
    float m_mutationRate;
    float m_offsetSize;
    float m_bestFitness;
    float m_totalFitness;
    int m_liveCount;
};
```

程序清单 20-3 GAMachine::Update()实现体

```
//--------------------
void GAMachine::Update(float dt)
{
    //find best out of the maximum tries, then start over
    if(m_generation > NUM_MAX_GENERATIONS)
    {
        WriteSolution();
        //reset
        CreateStartPopulation();
        Reset();
    }

    m_liveCount = 0;
    for (int shpNum=0; shpNum<POPULATION_SIZE; ++shpNum)
    {
        if(!Game.m_ships[shpNum]->m_active)
            continue;
        m_liveCount++;
        m_parent->UpdatePerceptions(dt,shpNum);
        ApplyBehaviorRule(shpNum);
        UpdateFitness(shpNum);
    }

    //if the generation is over...
    if(!m_liveCount)
        SetupNextGeneration();
}
```

Update()函数是遗传算法的主循环。函数首先检测是否完成指定轮次的进化，如果完成了指定轮次的进化，也就意味着整个模拟过程结束，此时系统将给出最优基因组并退出函数。如果需要该函数，可以给出最优的 10 个基因组，甚至整个基因组列表。我们可以略微改变参数或者选用不同的进化操作来获得不同的最优解，并对比这些最优解确定各方案的优劣。

如果模拟没有结束，那么系统将更新每艘飞船的感知数据(值得注意的是，在一般情况下控制器能自主更新，但在我们给出的遗传算法演示程序中，需要控制很多飞船，因此通过调用 GAAIControl::UpdatePerceptions()函数来完成感知的更新)并根据飞船的工作状态对其打分。如果游戏中不存在活跃飞船，该函数将调用 SetupNextGeneration()函数进入下一代进化。

程序清单 20-4 GAMachine::ApplyBehaviorRule()实现体

```
//--------------------
void GAMachine::ApplyBehaviorRule(int index)
{
    if(index < 0 || index > POPULATION_SIZE)
        return;
```

```
    Ship* ship = (Ship*)Game.m_ships[index];

    //not going to collide, just idle...
    if(m_parent->m_currentEvasionSituation == -1)
    {
        ship->ThrustOff();
        ship->StopTurn();
        return;
    }

    //thrust
    int thrustTp = m_genomes[index].
              M_genes[m_parent->m_currentEvasionSituation].m_thrust;
    ship->StopTurn();
    if(thrustTp == Gene::THRUST_FORWARD)
        ship->ThrustOn();
    else if(thrustTp == Gene::THRUST_REVERSE)
        ship->ThrustReverse();
    else
        ship->ThrustOff();

    //turn
    //-10 puts you in the middle of the sector
    float newDir = m_genomes[index].
                   m_genes[m_parent->m_currentEvasionSituation].
                                                    m_sector*20 -10;
    float angDelta = CLAMPDIR180(ship->m_angle - newDir);
    if(fabsf(angDelta)<=90)
    {
        if(angDelta >0)
            ship->TurnRight();
        else
            ship->TurnLeft();
    }
    else
    {
        if(angDelta<0)
            ship->TurnRight();
        else
            ship->TurnLeft();
    }
}
```

ApplyBehaviorRule()函数通过参数 m_currentEvasionSituation 传入特定飞船的当前躲避状态，对照飞船基因组中的正确规则确定飞船躲避行为：推进或者转到新方向。如果飞船不处于碰撞危险状态，传入的参数 m_currentEvasionSituation 约定为 - 1。如果飞船碰到该信息，它将停止转向和推进。

程序清单 20-5 GAMachine::UpdateFitness 实现体

```
//--------------------
void GAMachine::UpdateFitness(int index)
{
   Ship* ship = (Ship*)Game.m_ships[index];
   if(ship && ship->m_active)
   {
      //if I'm currently surviving a collision situation,
           //incr fitness
      if(m_currentEvasionSituation != -1)
         m_genomes[index].m_fitness++;
      m_liveCount++;
   }
}
```

测试平台中所采用的适应度是基于飞船进入躲避状态而没被消灭的次数的。测试平台通过检测某些感知信息来完成适应度的计算，这些信息包括飞船是否存活并且处于躲避状态，如果是，则增加该飞船的适应度数值。该测试平台没有采用适应度变形。在完成所有个体的基因组的更新和适应度的计算后才可能需要进行适应度变形。为了简单起见，测试平台可以采用排序变形。考虑到我们已经对基因组列表进行了分类，我们可以很容易地对基因组完成排序并用序号代替原来的适应度。在测试平台中采用该种变形方式有利于避免算法收敛于局部最优解。

程序清单 20-6 GAMachine::SetupNextGeneration()

```
//--------------------
void GAMachine::SetupNextGeneration()
{
   //next Generation
   vector<Genome> offspring;

   //sort the population (for scaling and elitism)
   sort(m_genomes.begin(), m_genomes.end());
   m_rankIndexLast = POPULATION_SIZE-1;

   //statistics
   m_totalFitness = 0.0f;
   for (int i=0; i<POPULATION_SIZE; ++i)
      m_totalFitness += m_genomes[i].m_fitness;
   m_bestFitness = m_genomes[POPULATION_SIZE - 1].m_fitness;

   CopyEliteInto(offspring);
   while (offspring.size() < POPULATION_SIZE)
   {
      //selection operator
      Genome parent1 = SelectRouletteWheel();
      Genome parent2 = SelectRouletteWheel();
```

```
            //crossover operator
            Genome offspring1, offspring2;
            CrossSinglePoint(parent1.m_genes,
               parent2.m_genes,
               offspring1.m_genes,
               offspring2.m_genes);

            //mutation operator
            MutateOffset(offspring1.m_genes);
            MutateOffset(offspring2.m_genes);

            //add to new population
            offspring.push_back(offspring1);
            offspring.push_back(offspring2);
        }

        //replace old generation with new
        m_genomes = offspring;

        for(i = 0;i<POPULATION_SIZE;i++)
            m_genomes[i].m_fitness = 0.0f;

        ++m_generation;

        //reactivate the ships
        for (int shpNum=0; shpNum<POPULATION_SIZE; ++shpNum)
        {
            //reset test ships to startup state
            Ship* ship = (Ship*)Game.m_ships[shpNum];
            ship->m_active = true;
            ship->m_velocity.x() = 0;
            ship->m_velocity.y() = 0;
            ship->m_velocity.z() = 0;
            ship->MakeInvincible(3.0f);
        }


    }
```

SetupNextGeneration()函数用于完成所有进化操作。该函数用于完成基因组分类、记录各代统计信息、利用精英主义和产生下一代等任务。在产生下一代中采用的操作主要有：轮盘选择、单点交叉和移位变异等。另外该函数还用于完成复位飞船的任务。

程序清单 20-7　GAMachine::CopyEliteInto()

```
//--------------------
#define NUM_ELITE 4
#define NUM_COPIES_ELITE 2
void GAMachine::CopyEliteInto(vector<Genome>&destination)
{
    int numberOfElite = NUM_ELITE;
```

```
    //copy the elite over to the supplied destination
    for (int i=numberOfElite; i>0; --i)
    {
        for(int j=0;j<NUM_COPIES_ELITE;++j)
            destination.push_back(m_genomes[(POPULATION_SIZE - 1) -
                                            numberOfElite]);
    }
}
```

CopyEliteInto()函数复制了上代群体中的顶级个体进入下代群体。这里说的复制是单纯的复制，不存在交叉和变异过程。当然，读者也可以根据自己的需要，引入低概率的变异或者是不引入新基因的变异，比如小位移量的移位变异。上述试验和修正同样也是遗传算法系统本身所需要的。

程序清单 20-8 GAMachine::SelectRouletteWheel()实现体

```
//--------------------
Genome& GAMachine::SelectRouletteWheel()
{
    float wedge = randflt() * m_totalFitness;
    float total = 0.0f;

    for (int i=0; i<POPULATION_SIZE; ++i)
    {
        total += m_genomes[i].m_fitness;
        if (total > wedge)
            return m_genomes[i];
    }
    return m_genomes[0];
}
```

SelectRouletteWheel()函数直接实现了轮盘赌选择算法。该算法的作用是按照适应度的比例确定生殖机会，也就是说适应度越高越有机会参与繁殖过程。但因为整个选择过程是建立在最开始的随机函数 randflt()之上的，所以有时得到的结果和预期的不太一样。换句话说，该算法有可能会漏掉适应度最高的个体，这也正是其需要精英主义过程的配合才能更好地选择的原因。对于某些问题，特别是进化群体较小的问题，随机遍历采样(Stochastic Universal Sampling，SUS)选择或者锦标赛选择更合适，理由如前所述。

程序清单 20-9 GAMachine::CrossUniform()实现体

```
//--------------------
void GAMachine::CrossUniform( const vector<Gene> &parent1,const
vector<Gene> &parent2,

vector<Gene>&offspring1,vector<Gene>&offspring2)
{
    if ( (randflt() > m_crossoverRate) || (parent1 == parent2))
    {
```

```
        offspring1 = parent1;
        offspring2 = parent2;
        return;
    }

    for (int gene=0; gene<GENOME_SIZE; ++gene)
    {
        if (randflt() < m_crossoverRate)
        {
            //switch the genes at this point
            offspring1.push_back(parent2[gene]);
            offspring2.push_back(parent1[gene]);

        }
        else
        {
            //just copy into offspring
            offspring1.push_back(parent1[gene]);
            offspring2.push_back(parent2[gene]);
        }
    }
}
```

CrossUniform()函数所实现的均匀交叉很简单。随机选择某基因位置，交换该位置前的所有基因，并直接复制剩余基因。和所有的交叉算法一样，均匀交叉算法在执行之前也需要完成两父代基因一致性的检查，如果检查出两父代基因组相同，算法直接返回。很显然，相同的父代个体将产生相同的子代，这也正是太多相同的进化个体将导致个体差异不明显的原因。

程序清单 20-10　GAMachine::MutateOffset()实现体

```
//--------------------
void GAMachine::MutateOffset(vector<Gene> &genes)
{

    for (int gene=0; gene<genes.size(); ++gene)
    {
        //check for thrust mutation
        if (randflt() < m_mutationRate)
        {
            genes[gene].m_thrust += (randint(0,1)?
                                    -m_offsetSize: m_offsetSize);
            //bounds check
            if(genes[gene].m_thrust > NUM_THRUST_STATES)
                genes[gene].m_thrust = 0;
            if(genes[gene].m_thrust < 0)
```

```
            genes[gene].m_thrust = NUM_THRUST_STATES;
        }

        //check for angle mutation
        if (randflt() < m_mutationRate)
        {
            genes[gene].m_sector += (randint(0,1)?
                                    -m_offsetSize: m_offsetSize);
            //bounds check
            if(genes[gene].m_sector > NUM_SECTORS)
                genes[gene].m_sector = 0;
            if(genes[gene].m_sector < 0)
                genes[gene].m_sector = NUM_SECTORS;
        }
    }
}
```

实数表示的基因组的移位变异就是简单地在原数的基础上加减偏移量，但我们事后需要检测变异得到的数据是否上溢或者下溢。事实上，我们只是希望通过数据的微小变化来找到更优解。位移量不能太大也不能太小：如果太大，算法可能会漏掉最优解；如果太小，算法可能没法从局部最优的“陷阱”中跳出来。因此，位移量的选择过程是折中的过程。另外，小位移量一般比大位移量要好些，但需要更长的收敛时间。

程序清单 20-11 GAAIControl::UpdatePerceptions()实现体

```
//--------------------
void GAAIControl::UpdatePerceptions(float dt,int index)
{
    Ship* ship = (Ship*)Game.m_ships[index];
    if(!ship)
        return;

    //determine current game evasion state
    int collisionState = -1;
    int directionState = -1;
    int distanceState = -1;

    //store closest asteroid
    m_nearestAsteroid = Game.GetClosestGameObj(ship,
                                        GameObj::OBJ_ASTEROID);

    //reset distance to a large bogus number
    m_nearestAsteroidDist = 100000.0f;
```

```
    if(m_nearestAsteroid)
    {
        Point3f normDelta = m_nearestAsteroid->m_position -
                            ship->m_position;
        normDelta.Normalize();

        //asteroid collision determination
        float speed = ship->m_velocity.Norm();
        m_nearestAsteroidDist = m_nearestAsteroid->
                            m_position.Distance(ship->m_position);
        float astSpeed = m_nearestAsteroid->m_velocity.Norm();
        float shpSpeedAdj = DOT(ship->UnitVectorVelocity(),normDelta)*speed;
        float astSpeedAdj = DOT(m_nearestAsteroid->
                              UnitVectorVelocity(),-normDelta)*astSpeed;
        speed = shpSpeedAdj+astSpeedAdj;
        speed = MIN(speed,m_maxSpeed);
        collisionState = (int)LERP(speed/m_maxSpeed,0.0f,9.0f);

        //direction determination
        directionState = GETSECTOR(normDelta);

        //distance determination
        distanceState = MIN((int)(m_nearestAsteroidDist/
                                  m_nearestAsteroid->m_size),4);
    }
    if(collisionState == -1)
        m_currentEvasionSituation = -1;
    else
        m_currentEvasionSituation=
          (collisionState*10)+(directionState*18)+distanceState;
}
```

UpdatePerceptions()函数的逻辑和先前控制器类的逻辑类似：负责决策过程中所需的感知数据的计算。在本例中该函数的主要工作是计算变量 m_currentEvasionSituation，该变量会在飞船选择适当躲避规则时用到。该变量计算的理论基础已经在本章的前面部分给出。

20.4　遗传算法在测试平台中的性能

尽管在 AIsteroids 程序中所实现的遗传算法的进化复杂度很低，但我们可以看到，经过仅仅 50 代的进化后飞船的躲避能力得到了很大的提高，而经过上千代的进化飞船的行为有点奇怪，但这些现象对于改进算法是有益的。图 20-8 给出了遗传算法在测试平台中的运行过程的截图。我们所实现的算法存在一些可以改进的地方，主要有如下几个方面：

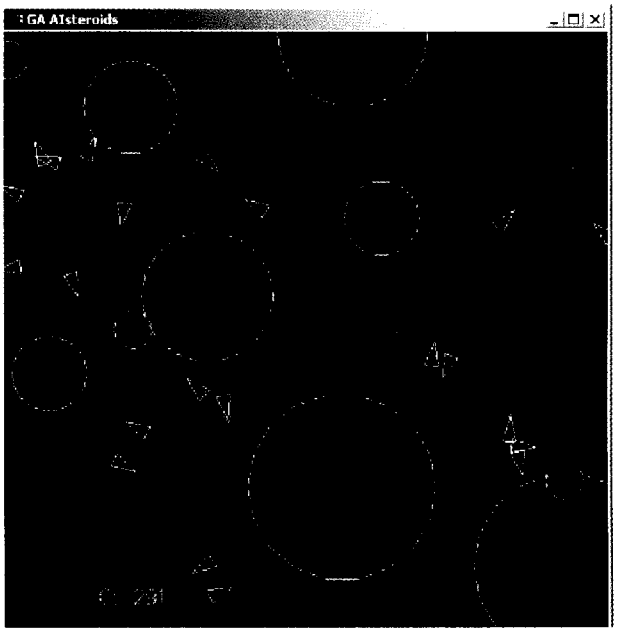


图 20-8　遗传算法在 AIsteroids 中的运行截图

- 在个体较少的情况下，算法的收敛速度太快。我们可以通过改变选择算法、增加个体数量和适当减少精英策略来改进该缺点。
- 我们需要对包含如此大量规则的规则集进行优化。这意味着我们需要执行几十万轮次的进化，甚至更多。我们可以采用实数来表示最优方向和推进力量的基因。这样就不需要对躲避状态进行进化搜索，只需通过用相对复杂的躲避算法来计算最优方向和推进。该方法有点类似简单的神经网络，不同的是神经网络是通过遗传算法来训练的。有关神经网络的内容我们将在下章展开。
- 测试应用应该是时间无关的，可以支持遗传算法以更快的速度执行。这其中将涉及到某些细节问题，比如需要保证变量 GameSession::m_timescale 包含在所有的计算时间内，包括在速度判断内。同样地，碰撞检测算法也应该能检测出两次检测算法运行过程中发生的碰撞，图 20-9 给出了该问题的形象说明。碰撞检测算法很难检测该类问题的主要因素是游戏节拍过大，也就是两次检测算法运行的时刻相差太大。在这段时间内，游戏对象的活动范围可以很大，可以从游戏空间的某个位置转到很远的某个位置，而在该运动过程中可能已经和其他对象相撞。解决该问题的主要办法是在算法中跟踪各对象的原有位置和新位置，根据跟踪结果判断新位置和原有位置之间的直线上是否发生碰撞。如果有其他对象出现在该直线上，则可以判断出发生了碰撞并休眠这两个对象。

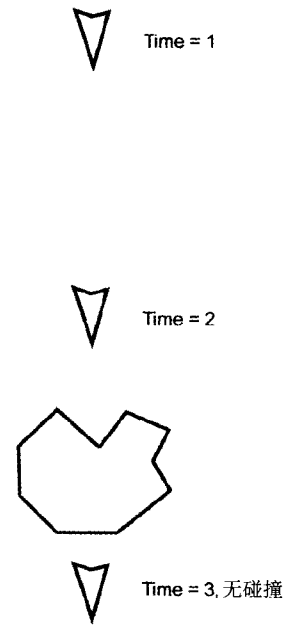


图 20-9　时间高度变形的游戏的问题

20.5　基于遗传算法的系统的优点

虽然遗传算法在 AI 游戏设计中不能扮演“万金油”的角色，但是其在 AI 游戏设计的某些方面确实很有优势，包括下面几点：

- 遗传算法适用于包含很多非线性相关参数的问题。参数关系越复杂，遗传算法的优势越明显。
- 遗传算法适用于从大量局部最优解中搜索出全局最优解的问题。这些问题的一个典型例子是本书前面提到的大量汽车物理参数的调试问题。该问题不是简单地尝试能解决的。
- 遗传算法适用于解中包含不连续输出的问题。输出完全连续的问题可以通过简单的函数调用来实现输入和输出的函数映射。输出半连续的问题可以通过状态机来解决，该状态机中可能存在某些在任何状态下都存在的行为，而为了完全覆盖该行为，在状态机实现时需要在每个状态下实现该行为。最后，各个状态看起来像是毫无联系的“孤岛”，而实际上各个状态包含部分相同的行为。不连续的行为是不光滑的，包含一个个没有关系的动作。
- 遗传算法可作为传统技术的补充。遗传算法令人难以置信的模块化，可以很容易地应用于大型 AI 系统。游戏中可能存在一些状态的决策可以采用遗传算法。如果能在游戏框架中引入遗传算法的支持手段并设计出合适的适应度函数，我们就不用坚持使用原有的那些烦琐而多错的方法，转而采用遗传算法。

- 遗传算法适用于计算代价很大的问题。对于需要大量计算资源的问题，找到合适的遗传算法解决方案意味着可以节省大量CPU时间。遗传算法方案可以看作“黑盒子”以取代其他方案所需的复杂数据结构，从而实现对游戏的优化。在抽象数学模型方面，遗传算法和神经网络有点类似。本质上，我们可以利用遗传算法完成非线性系统的建模，其中非线性系统的系数和外部参数通过基因组来表示。此时，适应度函数成了进化函数和其他数学函数的主要不同点。当然，只有当用遗传算法解决非线性问题的代价比其他方法小时，我们才会考虑使用。

另外遗传算法还有很多天生的优点。就算不知道怎么解决问题，我们也能很容易地通过遗传算法完成问题解决方案的建立并得到最终解。遗传算法是非常优化的算法，这意味着一般我们都能通过该算法来获得问题的解。最后，遗传算法能获得全局最优解，而不是仅仅限于局部最优解，这是因为遗传算法在很多候选解中并行地搜索。而很多传统的搜索算法或者数学算法一般只能找到问题的一个解，而不管该解是否是最优解。

20.6 基于遗传算法的系统的缺点

遗传算法也不是“免费的午餐”。和很多的AI系统一样，算法执行的时间越长，所获得的解越优。遗传算法的主要缺点包括：时间代价较大、算法性能随机性大、结果成败定义模糊、最优解不能保证、算法的扩展和参数调试难度大。

20.6.1 时间代价较大

通常为了获得最优解，算法需要迭代多次，甚至在有良好的基因组和完美的基因操作时也是一样的。另外，为了提高算法的性能我们经常需要对算法的方方面面都进行改进，包括基因的编码、变异和交叉等基因操作的设计等。这所有的过程都是很费时间的。当然对于那些适应度函数能通过数学算法简单地表示出来的问题，算法的过程可以得到比较明显的简化，我们不需要为了获得某个体的适应度而迭代多轮游戏循环。但这样的问题比较少。而在游戏产品中建立遗传算法并进行进化过程对游戏的性能是有很大影响的。我们不鼓励遗传算法的进化过程出现在玩家游戏的过程中，因为算法需要花很长的时间才能获得最优解。以前面说到的飞船的躲避能力为例，飞船通过进化获得较高的躲避能力需要很长时间，而在这段时间内飞船几乎没有躲避能力。正因为如此，遗传算法的执行都是在游戏产品发布之前离线完成的。

20.6.2 算法性能随机性大

由于存在很多不同的问题表示方法，存在很多不同的选择、交叉、变异和适应度变形算法；存在大量的辅助参数，比如个体数量和变异率等，以及适应度函数确定的主观性，遗传算法的可塑性达到了前所未有的高度。对于给定的问题，我们可能通过遗传算法获得很高的性能，但是前提是我们正确地完成了问题的表示，正确地给出了选择、交叉等算法

操作，正确确定了参数的值等。但是要正确地组合这些因素需要很长的时间和不断地实验。获得这些因素的正确组合的唯一办法是实验，这是由于遗传算法的实现具有因应用而异的特性。

20.6.3　结果成败定义模糊

很多时候我们发现自己的遗传算法实现无效，但却不知道原因，是变异操作引起算法没法收敛，抑或是算法收敛于低劣的局部最优解，如果是前者需要降低变异程度，如果是后者则需要提高变异程度。很多时候我们不得不不停地实验以期解决该问题。事实上，所实现的遗传算法不起作用可能是由于适应度函数和基因操作设计上的失误。由于遗传算法本身的特性，我们很难识别是代码设计上的问题还是其他的原因。因此，在实现遗传算法时我们必须很小心，避免出现上面提到的问题。

20.6.4　最优解不能保证

遗传算法使用了启发式技术，而启发式技术带有很多的随机性，因此我们不能保证得到最优解。确实可能存在既保证能获得最优解而又不会丧失遗传算法好处的方法，但寻找这样的方法的过程有点赌博的味道，其中存在太多的随机性。另外一个问题是很多时候我们自己都不清楚是否获得了最优解，因此也就不能确定修改算法的某些因素就能获得更好的结果。

20.6.5　参数调试和扩展难度大

一旦通过遗传算法解决了某个问题，特别是该问题的解决是经过了很长一段时间并对算法的实现进行了无数地修改的情况下，我们都不想对该问题做过多的修改，原因是害怕失去好不容易获得的成功。但游戏开发人员很难在设计中全面地预期到游戏 AI 的需求。很多时候，AI 参数的调试都是在游戏的最后阶段进行的，调试过程一般是这样的：游戏设计人员向玩家提供游戏的测试版，玩家试玩后给出反馈意见，最后设计人员根据反馈意见修改游戏的设置。对于那些基于代码的系统，参数的调试以及简单功能的扩展是很容易的，特别是所设计的 AI 系统是数据驱动的系统或者其他扩展性良好的系统。但对于基于遗传算法的系统，AI 参数的调试和游戏扩展就没那么简单了。参数的简单调试也许是可能的，这主要是通过对解进行不同的解释来实现的，比如在我们的测试平台中，当通过遗传算法得到飞船朝向的解后，可以对该解进行适当的修改而完成参数的调试。但在游戏中加入哪怕很小的新功能都需要重新实现遗传算法的整个过程，包括基因结构的设计和进化过程。正因为如此遗传算法只应用于游戏中很少需要修改的部分。

20.7　范例扩展

20.7.1　蚁群算法

蚂蚁群被发现是解决问题的高手。至少笔者个人是这么认为的。假如我们把一只蚂蚁孤零零地扔在一个新环境里，它除了等死，没有其他的选择。它会缓慢地探索所有的方向，

最后却毫无办法。但如果扔下的是一群蚂蚁，那情况就大大不同了。更进一步，如果扔下的是几十万只蚂蚁，那它们会集合起来，建立种群，寻找食物，抵御攻击，甚至消灭比邻的种群。蚁群是怎么做到这一切的？这来源于蚂蚁通过自然进化得来的集体智能。蚂蚁寻找食物的过程就体现了该智能。蚂蚁爬行时会在沿途释放少量称为信息素(pheromone)的特殊化学物质。该化学物质会引起其他蚂蚁的注意，蚂蚁根据信息素的强度选择前进的路径，信息素强度越大，该路径被选中的概率越大。蚂蚁之间通过信息素交流来选择最短路径，最后找到距离最近的食物。这听起来有点类似影响图。事实上，也可以通过 LBI 系统对这些信息进行编码以实现蚁群算法，但在实现过程中需要注意到：蚁群算法和遗传算法有点类似，也涉及到很多的随机性和基因重组问题。在这里我们借用集体智能的思想帮助我们改进遗传算法中的适应度函数。和原有的遗传算法类似，加入蚁群算法的遗传算法开始于大量的随机群体，区别是在寻找解的过程中新改进的遗传算法的成功更多地依靠于整个进化群体而不是原算法中的某些特殊个体。

20.7.2 协同进化

另外一个吸引人的扩展是协同和竞争的引入。如果系统中存在两种或者更多生物并能共存，而且它们的适应度能相互促进达到最大值时，我们称这些生物的关系是合作的。如果系统中的两种生物的适应度是此消彼长的，我们称这两种生物的关系是竞争的。在两种情况下，所有生物的进化过程都会加速，因为不同的生物群体会相互影响并产生进化合力推动进化[Hillis91]。该思想可以进一步扩展到整个社会群体的进化，有时我们又称这种进化为社会进化(societal evolution)。社会进化可用于开发即时策略类游戏，比如《世界魔兽》，在这类游戏中所有种群可能处于混战中，合作和竞争不停地变化。社会进化还可用于在游戏平衡系统中模拟现实物种之间的关系。

20.7.3 自适应遗传算法

普通遗传算法的性能取决于系统中操作类型和参数的选择。而这些参数和操作很难手工调整。研究人员设计了很多遗传算法变种在完成问题求解的过程中顺便进行参数和进化物质的进化[Bäck92]。因此，在进化过程中交叉率和变异率也是变化的。这样的系统有时能很好地工作，但并不适用于所有问题。有时它们收敛得太快，因此研究人员针对该问题给出了很多解决方法。

20.7.4 遗传程序设计

在该扩展中，基因中的遗传物质是程序代码本身。遗传程序设计的目的是寻找到更好地解决某个问题的代码序列，而不是确定某个代码实现中的参数的最优解。游戏代码序列的交叉和变异有点特殊，至少要保证代码在交叉和变异操作后仍然是合法的，难度较大。因此遗传程序设计是很少使用的。但对于数据驱动系统而且系统中的数据是表示行为的一组简单指令序列(或者脚本)，遗传程序设计可用于完成这些指令序列(或者脚本)的编写，也可以把设计人员提供的脚本序列作为初始群体进行进化，得到这些脚本序列变种，使得所设计的 AI 机器人更具个性。

20.8　设计上考虑的因素

遗传算法是一种能解决难度大或者计算代价大的问题的方法，同时它也能给出程序员不能给出但又有意义的解。通常遗传算法都是离线执行的，因为在大多数情况下进化过程是相当漫长的。在决定是否在游戏中采用遗传算法时，需要考虑以下几个方面：解决方案的类型、智能体的反应能力、系统的真实性、游戏类型、平台、开发限制、娱乐限制等。

20.8.1　解决方案的类型

复杂 AI 战略决策系统通常需要在游戏开发的过程中不停地修改功能集，需要在开发后期不停地调试游戏的可玩性和效果，因此在这种情况下通常不宜采用遗传算法。战术决策比战略决策更加地模块化，因此其通常会被独立出来并通过遗传算法加以解决。如果采用遗传算法来完成大型文明类(Civilization-style)游戏中的外交系统的设计，由于不同种群的外交系统的进化需要单独完成(采用社会进化时可能不用单独进行，但是难度较大)，而我们需要完成游戏中所有种群的进化，因此将耗费相当长的时间。但对于某些小而复杂的问题，我们就可以采用遗传算法来解决，比如游戏中城镇的修建，就可以通过遗传算法来找到最好的修建方式，即使得城镇和周围的地形相适应，又使得城镇的防御和设施的利用率最大化。

20.8.2　智能体的反应能力

可以把采用遗传算法解决的问题看作黑盒子，在游戏产品真正运行的过程中，该黑盒子的延时几乎可以忽略。因此遗传算法的引入不会对智能体的反应能力造成任何影响。

20.8.3　系统的真实性

在涉及到系统的真实性时，遗传算法可能会是个包袱。有时通过遗传算法得到的解太过优化。遗传算法给出的最终解综合考虑了所有可能碰到的情况并对 AI 角色的所有特征和行为进行了最优化组合，以至于采用遗传算法的 AI 角色的“智慧”似乎超过了人类。以第一/三人称射击类游戏(FTPS)为例，在这些游戏中玩家可通过武器的后坐力使自己上升到正常情况下到不了的地方，而由遗传算法驱动的机器人在感知到这一切后，就会不停地满地图利用该技巧升入空中，而从不落地。但不管怎样我们可以通过修改适应度函数来避免出现这样的问题。

20.8.4　游戏类型

几乎所有的游戏类型都可以在很多方面使用遗传算法：即时策略类游戏可以在修造建筑决策和防御某些特殊战术(比如 rushing：这是一种人类经常使用的战术，其主要思想是人类在游戏开始阶段生产大量攻击性单元，比如星际争霸中的小狗等，试图一举歼灭还处于发展阶段的敌人)时采用遗传算法；第一/三人称射击类游戏或者其他平台游戏可以通过遗传算法来更好的处理地图特性，赛车类游戏可以通过遗传算法获得更好的 AI 驱动的赛车手，射击类游戏可以通过遗传算法来协同进化所有的角色。

20.8.5 平台

在考虑是否采用遗传算法时平台不是主要考虑的问题，因为进化过程基本都是离线进行的。事实上，由于遗传算法“黑盒子”的执行几乎不需要时间，其可以帮助我们提高在CPU受限平台中的游戏的性能。

20.8.6 开发限制

在考虑是否采用遗传算法时我们需要特别考虑开发因素。遗传算法很难调试，我们可能需要花费额外时间来跟踪算法中出现的小问题。开发人员应该注意到遗传算法参数调试的过程需要大量的时间，也应该注意到进化的过程同样需要大量的时间。开发的过程中是否能提供那么多时间？开发团队中的其他程序员是否需要在开发的最后时刻修改可能危及遗传算法解的有效性的部分设计？最终的游戏产品是需要提供整个进化部分的实现还是仅仅需要进化得到的解？遗传算法的调试和反馈是否自始至终出现在产品的整个开发流程中？产品的开发流程是否支持遗传算法的快速调试？所有的这些问题不仅需要考虑开发团队的因素，也需要考虑游戏本身的因素。

20.8.7 娱乐限制

我们可以通过遗传算法来解决很多游戏相关的问题，比如游戏难度设置问题，可以在针对不同难度等级的 AI 系统中设计不同的适应度函数，针对难度要求高的游戏层次设计更好的适应度函数。参数调试和游戏平衡会更难些。遗传算法的真正优势体现在当游戏需要诡异或者多样的AI行为时其能给出所需的非常规解。

20.9 小结

遗传算法是一种用于解决或优化 AI 问题的令人着迷的方式。遗传算法的实现很容易建立，但却很难做到完美，因为存在大量的设置和用法。遗传算法能针对游戏环境找到新解，这是目前游戏设计的最高目标之一。

- 自然进化采用基因作为规则集。根据对环境的适应程度，一对生物体会被选中来完成生殖并传递它们的遗传物质进入下代个体。新个体在产生过程中将经历交叉和变异，这些过程可能提高个体对环境的适应度。
- 遗传算法一般都是离线执行，因为进化过程相当耗时而且需要执行很多代的进化才能获得有用的结果。
- 遗传算法是一种启发式搜索算法，通常也会被认为是强制搜索算法。
- 遗传算法不能保证性能和解的有效性。
- 基本的算法执行过程是这样的：首先随机初始化群体，计算群体中所有个体的适应度，然后选择合适的个体进行生殖，在生殖过程中引入随机变异，最后得到新一代个体。迭代上述过程直到个体的适应度达到可接受的水平。
- 在利用遗传算法解决问题时，利用基因和基因组结构表示问题的解。

- 适应度函数是遗传算法是否优化的主要因素。适应度函数得到的数值可以直接使用，也可以在变形后使用，以避免数据扎堆。
- 繁殖过程可以为系统剔出低劣基因并提升优秀基因提供物质基础。同时，它还通过交叉来帮助搜索优化解，通过变异来帮助避免收敛于局部解。当前存在很多类型的选择、交叉和变异算法。
- 在测试平台中实现遗传算法需要首先创建用于进化过程的应用程序，然后创建处理主要算法的类 GAAIControl。
- 遗传算法在解决包含大量非线性相关的参数、包含很多局部最大值或者输出非连续的问题时很有优势。遗传算法不仅可用于辅助传统技术，还可用于优化计算代价很大的决策系统。
- 遗传算法的进化过程相当耗时，而且算法性能随机性很大，不一定能获得问题的最优解。算法不具备成败判断能力，因此很难进行调整和调试。
- 遗传算法的扩展变形包括：蚁群算法、协同进化、自适应遗传算法和遗传程序设计。

21 神经网络

神经网络(NN，也称为人工神经网络，因为它源于实际的大脑)是计算机学家在 AI 中结合生物学知识的产物，它在某种程度上与遗传算法类似。遗传算法利用最适合的技术得到问题可能的解，而神经网络对问题的求解方法则基于大脑组织和功能上的原理。虽然它并不是对大脑的实际模拟，但它确实为我们提供了对输入数据进行模式匹配和趋势预测的直观方式。

21.1 自然中的神经网络

动物大脑基本上是称为神经元的神经细胞之间的大型互连。对于地球上的某些高等生物而言，“大型互连”这个词过于简单，人脑大约由 100 亿个神经元组成，大象则是我们的 10 倍。每个神经元都有到其他神经元的一系列连接，包括向内和向外的连接(人脑每个神经元大约有 10000 个连接)。向内的连接称为树突，向外的连接称为轴突(见图 21-1)。严格地说，神经元之间并不是直接连接的，而是一个神经元的树突与其他神经元的轴突之间非常靠近(通常约为 0.01 微米)，它们之间的空间称为突触间隙，或突触。神经元具有电气性(虽然它们的导电率、总电荷、电容和其他特性在某种程度上由内部化学特性决定)。

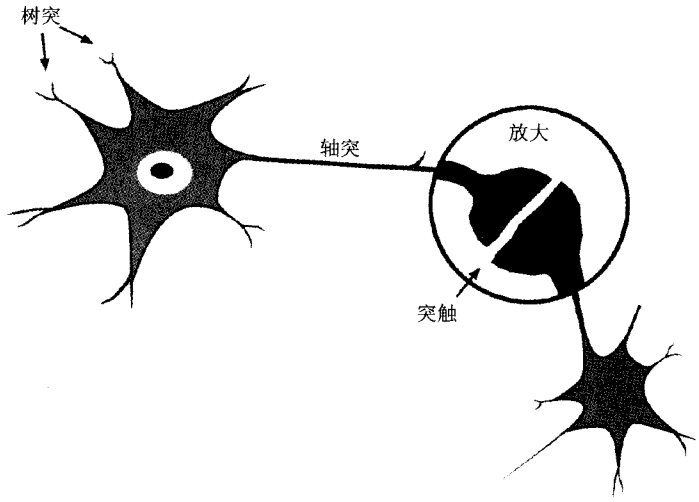


图 21-1 神经元的各部分

一种对单个神经元内部机理十分简化的描述是，单个神经元的大量树突从附近的轴突得到电荷，在神经元内部像电容器一样累积电荷。如果累积的电荷量过多(超过某个上限)，则由轴突释放电能(总是以一定的量释放，类似计算机中位的开关状态)，称为动作电位，由此传递到其他神经细胞树突。如果某个神经元放电频率过高，则会引起它内部微小的生物学变化(例如沿轴突和树突电阻下降，突触的电敏感度增强，甚至各点之间神经纤维的大小发生变化)，引起放电电位的下降。实际上，神经元已经“学会”通常需要放电并且放电的电阻会逐渐减小，而不是等待整个电荷建立起来。相反的效果也可能发生，即某个特定的神经元几乎从不放电，因此逐渐萎缩。虽然这明显给人们一种通过期望进行学习的生物学概念，但也在细胞水平建立了模式识别的概念。

神经元之间相互作用的另一个概念是表现(exhibition)和抑制(inhibition)。如果一个神经元终止了到达其他神经元的电荷，则对其他神经元而言它是抑制的，反之则是表现的。这与细胞内部的突触变化不同，因为它不是针对特定连接的。对于抑制性的神经元，所有进入它的连接都会被抑制，而每个进入该细胞的突触则会萎缩。

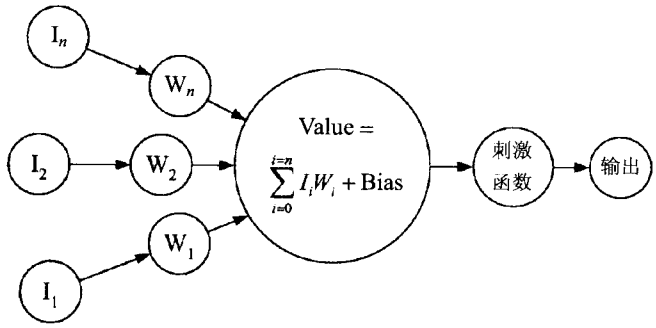
从本质上讲，动物大脑的工作机制是，接收输入、识别输入的模式、基于这些模式做出决策，这正是我们要在软件中用神经网络进行模拟的。

我们还试图利用大脑中的连接在求解问题时表现出的并行性。人脑大约以 100Hz 的频率运算，远远不及现代计算机的速度。但是计算机一次处理一条指令(对于多处理器系统是几条指令)，人脑一次能执行数百万条指令。由于大脑以符号化方式存储知识和解决问题，我们可以利用大脑多种层次的效率，从而完成大量的并行处理。显然，除非采用并行处理的 CPU，否则我们就无法模拟人脑实际的并行性。人们希望利用编码到神经网络中的许多关联并行层次，而其他方式很难或无法找到。

21.2　人工神经网络概述

图 21-2 显示了部分人工神经元和基本神经网络的概况。注意与一个神经元关联的值是各个输入值乘以其连接权值求和之后，再加上神经元的偏置值。偏置值是指网络中神经元所具有的抑制或表现效果。神经元上的“轴突”是由神经元的输出值表示的，有可能还要经过一个活动函数的作用，这稍后将在“神经网络的使用”一节中介绍。

在概况图中，圆圈是神经元(在讨论人工神经网络时，有时也称之为节点)，它们之间的连线表示神经元之间的连接。图中第一列的节点是输入层(input layer)的一部分。节点代表网络入口点；外部未分类输入由此进入神经网。第二列包括隐含层(hidden layer)，代表网络的内部数据存储。这些节点很有用，它们为网络增长提供了空间，同时使网络具备处理更大模式变化的能力。隐含层可能包括一组节点(如图 21-2 所示)，或多组节点，根据处理要求可以达到任意复杂度。一种特例是没有隐含层，输入直接映射到输出，称之为感知机(perceptron)。这些属于非常低级的神经网络，但仍可以用于一些线性模式识别。第三列叫做输出层(output layer)，它对应于网络对输入的实际分类。



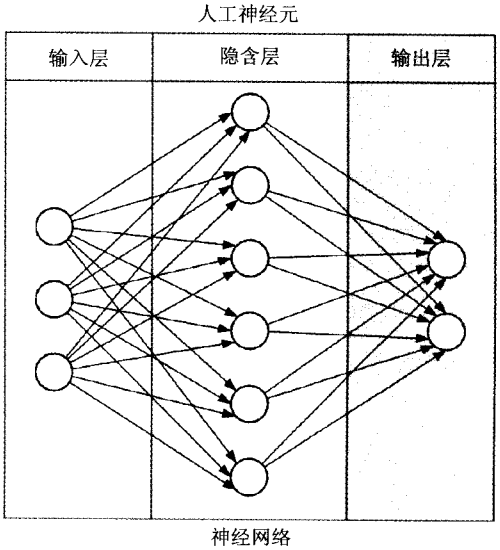


图 21-2 人工神经元和基本神经网络

还要注意图中的实际连接本身。每个连接都关联一个值和一个方向。这个值表示连接的权值，相当于生物学上两个神经元之间关联的强度。至于方向，图 21-2 中的神经网络是一个前馈(Feed Forward，FF)网络，因为每一层在网络中仅向前传播。另一类神经网络则没有这个限制，通常称为递归网络(recurrent network)。在这类网络中，信息可以从输入传到输出，然后再返回，系统内允许反馈。为了实现上述情况，递归网络含有一系列状态变量，因而比前馈网络略微复杂些。在游戏设计中，程序员几乎全部使用前馈系统，因为它们易于理解，调试起来更直观，并且运行成本较低(因为反馈阶段需要数据在网络上运行多次)。递归网络在技术上比前馈系统能力更强，但前馈网络的生成十分容易，因而递归网络的许多好处可以通过运行多个前馈网络而获得。

神经网络的最后一个性质是其连接的数量。图 21-2 中表示的是一个全连接的神经网络，因为每个节点都与下一层的各个节点相连接。如果某些节点由于种种原因而不符合这一规则，那么这类神经网络也被称为稀疏(sparsely)相连的。虽然建立稀疏相连的节点并不

比建立更为常见的全连接网络复杂多少，但它们很可能会影响系统的性能，好的神经网络能够确定不必要的连接并相应地调整连接的权值。因此，稀疏相连的神经网络并不常用。

神经网络已成功应用到许多不同的行业中。早期大规模成功应用的一个案例是美国邮政系统，该系统利用密集训练的神经网络对邮件地址进行手写识别。其他用途还包括预测天气、判断信用卡诈骗、语音识别、诊断疾病或其他健康问题、人工视觉技术，甚至包括过滤色情网站或其他图形材料。

游戏需要解决的问题与其他行业类似，因此神经网络在游戏中也用于某些模式识别或预测问题。只要我们可以找到系统中的一个模式，逻辑上说就能够利用它直接对系统做出决策，确定系统中对方所做的决策，或利用先前存储的数据预测将要发生的事件。以上三点在游戏 AI 领域都很有用。最基本的是，神经网络可以被训练成黑匣子，用于潜在的动态选择这类高开销操作(例如篮球运动员当前的灌篮动作、他的技术、周围的球员分布、游戏的难度级别等)，这与利用模式直接做出决策大致相似。利用合适的神经网络进行模式识别可以作为人物建模系统的基础，通过预测玩家接下来的动作，游戏 AI 可以占据玩家的上风。最后，虽然在游戏设计中并不常见，但可以利用神经网络“存储”信息，在游戏期间让学习元素持续运行，从而使神经网络潜在地学会自适应技术。这之所以不常见是因为其具有不可预测性，以及不稳定性，这是神经网络学习算法的本质所决定的。有些系统允许这种情况发生，但严格限制学习的范围，以尽量减小进入游戏世界的随机元素。如果游戏人物突然表现出所谓的灾难性忘却，并且在有效接受网络中存储的完整关系之后，无法执行任何任务，那么“黑与白”这个游戏将很难成功。

21.3　神经网络的使用

在游戏中实现神经网络系统的基本步骤是设置网络、用特定数据作为输入训练网络、然后在游戏输入上实际使用它。第一步，为游戏问题设计神经网络结构，需要考虑以下几个因素：结构、学习和训练数据。

21.3.1　结构

结构指的是要构造的神经网的类型(前馈，递归)和组织(节点的数目，隐含层的数目)。多数人使用前馈网络，因为在某种程度上反馈网络可以建立到前馈网络中，而且更加经济，性能更高。

想要神经网络分类或模式匹配的变量数决定了神经网络的输入节点数。一个神经网络可能只含一个输入，其作用相当于询问“这是什么，或者这该如何处理？”但也可能需要对一些信息做出决策。尽量减少输入的数目，将其控制在必不可少的输入上，因为这里增加的任何额外元素都会显著扩大神经网络搜索的空间。我们需要系统完成的任务，基本上是找到一个将每个输入联系在一起的模式。因此在两个输入的情况下，神经网络只需找到一条“线”连接这两个输入，但如果有 12 个输入，那么相当于要求神经网络找到刚好适合

数据点的最近似的“十二面体”，这并不容易做到。注意，代表基于简单变量组合或计算的抽象变量，往往更适合神经网络。因此，在我们的测试平台上，一个称为“危险”的变量可能会比代表几个靠近的行星位置、速度等的一串输入更好。

在神经网络中关于节点数只有一个基本规则：所处理的节点数越少越好。再次说明，神经网络中包含的节点越多，神经网络在寻找合适的解时需要搜索的空间就越大。

对于需要多少个隐含节点实际上并无定论(虽然对于游戏人工智能遇到的大多数问题而言，一个隐含层似乎比较合适)。通常的做法是采用适中的隐含节点数(输入节点数的两倍)，然后适当增减节点并比较性能的变化，直到性能基本不再发生变化。许多资料给出了隐含节点数的指导规则，甚至包括“永远成立”的规则。但这类信息大部分是无用的，主要因为这些资料是基于输入和输出节点数确定这些规则的，并没有考虑到诸如训练样本的大小、所求解函数的复杂度或者输出中噪声(方差)的大小这类重要因素。

输出节点数与需要从神经网络得到的输出数相等。如果我们想建立一个系统，用来告诉我们是否可以见到游戏英雄？那么我们的神经网络仅需两个输出节点：是与否。如果要建立一个能够识别全部数字的字符识别系统，则需要 10 个输出节点，每个对应 0~9 中的一个数字。

注意每个输出不一定只有两种可能；它可以是激活的连续值。因此，输出神经元可以是“左转”和“右转”，神经元的激活水平会告诉我们转弯的程度。光滑激活是通过适当的激活函数得到的。常见的一些激活函数包括阶跃函数、双曲正切 S 形函数、对数 S 形函数和高斯函数。

对输出整形并不是在给定神经元的最终值上使用激活函数的唯一原因。对隐含节点使用激活函数也是出于完全不同的原因。神经网络最强大的能力之一就是封装一个将输入映射到输出的非线性函数。然而，只有神经网络本身能够代表一个非线性函数，它才能实现上述功能。如果没有隐含层，那么感知机只能找到输入和输出之间的线性关系。但给感知机增加一个隐含层是不够的，我们还必须用非线性激活函数作用在节点上，给网络连接提供一个非线性元素。除了多项式函数以外，几乎任何非线性函数都可以使用。对于反向传播学习(稍后将讨论)，激活函数必须是可微的，如果函数是有界的则更好，这也是一般的激活函数的选择标准。

21.3.2 学习机制

在建立了神经网络之后，需要确定如何训练网络。神经网络的学习类型主要有两种：有监督学习和无监督学习。有监督学习包含由输入输出对构成的训练数据。将数据输入神经网络，如果网络输出和训练数据给出的期望输出之间存在差异，则相应地调整网络权值。训练持续到达到一定水平的精度为止。这种方法称为反向传播(backpropagation)，因为调整网络参数的方式是从后至前。有监督学习的另一种形式称为强化(reinforcement)学习。在这类系统中，算法并不包含期望的输出，但网络执行良好时会得到奖励(或其行为得到增强)。有些实现还在系统表现不佳时加上了惩罚，但这通常会矫枉过正。

无监督学习是在统计上查看输出并相应调整权值。其中的一种技术称为扰动学习(perturbation learning)，它与学术上称为模拟退火(simulated annealing)的人工智能技术非常相似。在扰动中对神经网络进行测试，然后微调某些值，之后再试一次。如果得到更好的性能，那么就重复这一过程；否则就回到上次的网络设置。另一种颇为常见的技术就是用遗传算法调整神经网络的权值。二者之间的关系确实体现了各自的优点：神经网络确定了输入和输出间的模式，而遗传算法则用于一系列参数的优化，以最大化某些适应值函数。

21.3.3　创建训练数据

现在已经建立了神经网络，并且也知道如何训练这一网络。如果选择使用有监督学习，那么接下来的任务就是获取在训练神经网络时所用的测试数据。有下面几种方法。

为了训练系统完成某种任务，可以实际记录人在完成这项任务时的做法，然后基于人的行为创建一批测试用例。这类训练数据非常好，因为还可以用它将人类的表现建立到 AI 中去(因为可以利用人类该做而没有去做、甚至做错的情况所对应的数据点)，这会令您的 AI 系统看上去并不像机器人那般机械，这是使用静态算法所无法达到的效果。但是这种方法非常耗时，并且人工智能系统所具备的能力受限于被模拟的人所具有的能力。

另一种方式是编写一个单独的程序生成合理的输入情况，并让人判断将会得到何种输出。这种方法很适合二元或离散输出值(虽然对于涉及的人而言同样非常耗时)，但对于实数或连续型输出却不适用。可以生成随机输入输出对，并检查其有效性，只有在成功时才存储它们。也可以使用关于这个问题的某种专家知识来生成一些训练数据点。考虑到开始使用神经网络的原因就在于不具备这类数据，因此这可能比较困难。

所需训练样本的数目取决于目标中的噪声指标和所学函数的复杂度，但是作为起点，训练样本至少是输入单元的 10 倍比较合理。这对高度复合的函数而言或许不够。对于分类问题，最小类别中的样本数也至少应该是输入单元数的几倍。训练集的最佳大小应争取是输入数的 10 倍。

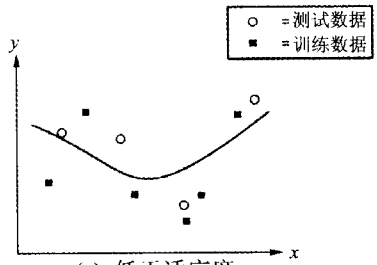
21.4　神经网络活动

我们应了解神经网络的模式识别能力，真正理解它的工作机理。充分认识其中的过程还有助于调试或完善神经网络的性能。神经网络适合完成的两个基本(或类似)任务是回归和分类。图 21-3 是一个回归的例子；图 21-4 是几个分类的例子。

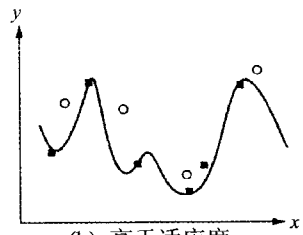
回归定义为在容许的偏差范围内寻找适合所有数据点的函数。例如要建立一个神经网络使游戏中的 AI 敌人横跨一步以躲避玩家的子弹。输入敌人面对的方向、玩家的位置以及敌人的位置。假定子弹将直接从玩家飞向敌人，那么神经网络将确定敌人的运动向量。这个例子中神经网络实际学习的是什么呢？如果要用算法解决这个问题，那么需要计算二者间的向量，然后用该向量与敌人面对方向上的单位矢量做点乘，得到玩家需要从当前面对的方向转过的角度。但是向左转还是向右转？需要再执行一次同样的运算，但这次用的

是垂直于敌人面对方向上的单位向量。如果把这些数学运算组合成一个大的函数，那么这个函数就是为了解决该问题而需要神经网络学习的函数。神经网络实质上就是学习如何正确执行点乘和比较操作。

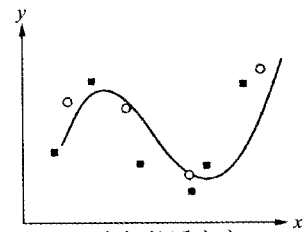
读者如果进一步看这个例子，可以从中看出网络的结构要求。想象一下，神经网络的全部输出只是输入的一个线性函数。例如游戏中有一个光滑的山坡。山上的敌人知道他沿着此山行进的距离，而想要知道他当前的高度。若用神经网络解决这一难题，便能很快找到答案，不用隐含节点就可以解决。这是因为要找的函数是山的斜率，它是输入的线性函数。但如果游戏中的山和一座真实的山一样，在山谷、高地、低洼等地形处斜率不断变化，那么神经网络需要计算一个非线性方程(相当于某种形式的复杂傅立叶变换或类似函数)去近似高度“函数”。为了做到这一点，将需要大量的隐含层，用非线性激活函数存储这类信息。



(a) 低于适应度



(b) 高于适应度



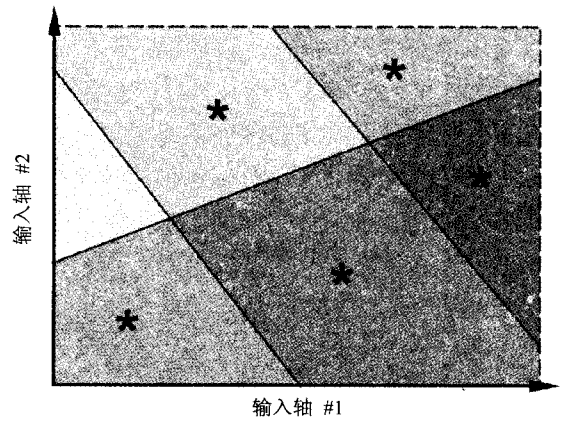
(c) 良好的适应度

图 21-3 回归示例

上面可视化的另一好处是，有助于估计或调试对神经网络训练的程度。在图 21-3(c)中，可以看到曲线与数据的趋势很符合。这个神经网络训练的程度刚好合适。它能够很好

地确定数据的本质，不会欠拟合(函数过于光滑，从而忽视数据中的关键变化趋势)或过拟合(回归函数不够光滑，将噪声数据也考虑在内，因此会得到意外的结果)。如果想建立平稳运行的神经网络，但又不想把大量时间花费在为此进行的实验上，那么明确地认识到网络中发生的情况是第一步。

神经网络得以发展的另一种类似的任务是分类。如果有一堆纽扣，并要求把它们按颜色分开，那么就会把它们放到代表每种颜色的堆中。换句话说，将会对其进行分类。当给神经网络输入时，要求网络将输入分类到输出节点所代表的堆的编号中。但神经网络处理的是整个搜索空间，而不是每个对象(给神经网络两个输入，不是给它两个不同的数，而是给它两个输入轴)。它并非将对象分成堆。输出节点代表的是输入可能性状态空间中的分割线。用这种方式直观显示一个分类神经网络十分有用。把每个输入作为图中的一个理论轴，每个输出作为一条线(或一个面、一个超面，取决于系统的维度)，将不同输入分离到不同的类别中。图 21-4 所示分别是 2 输入和 3 输入的例子。



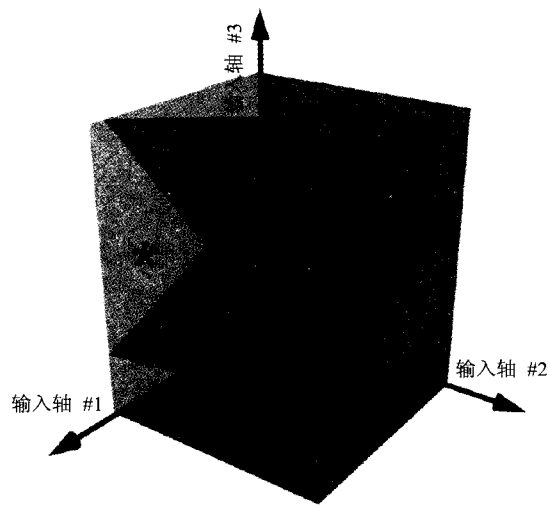


图 21-4　具有两输入和三输入搜索空间的分类例子

当以这种方式想象神经网络时，就很容易理解神经网络的一系列行为了。添加不必要的输入节点会增加神经网络解决问题的难度，使搜索空间随维度的指数增长。额外的输出节点使分类更严格，但也会使区分工作复杂得多。虽然上面的描述是神经网络内部机制的简化，但它有助于对效果做出描述。

21.5　在 Alsteroids 测试平台上实现神经网络

首先必须确定在测试平台应用中我们想要神经网络完成何种任务。在这个简单的游戏中，虽然并没有什么问题非要用神经网络来解决不可，但我们一定能够提出一个适当的难题，以说明神经网络适于处理此类问题。本章中我们再次以小飞船碰撞避免问题为例。为此，我们首先通过记录游戏玩家的动作来创建必要的训练数据。然后用这个数据来训练神经网络。最后，重新启动游戏，加载训练过的神经网络数据，并用它来进行正确的规避动作。

在这个例子中，我们要用的神经网络十分简单，由一个 8 神经元的隐含层和 4 个输入节点以及 3 个输出节点构成。

我们使用如下的输入：

- 两个输入，分别是飞船和最近小飞船间向量的 X 分量和 Y 分量。
- 这两个对象一起移动的速度，它的确定方法是考虑每个对象的移动速度，并找到沿连接另一对象方向上的速度分量。
- 飞船的移动方向，它为神经网络提供了参照系，以确定与其他输入的相互关系。

系统提供的输出只是用于飞船控制的简单布尔值，用于决定飞船是否推进、左转或者右转。

神经网络系统的实现由 4 个主要部分组成：

- **Neuron 是神经网络的基本单元**。神经元结构存储了几个数据域，包括每个进入神经元的连接的权值、输出值以及在训练中根据期望输出计算得到的误差梯度。
- **NLayer 是构成神经网络中特定一层的神经元的集合**。在这个层次上，可以对原有神经元执行各种操作，例如传播(使输入向前通过网络)、反向传播(反向沿着网络在每个神经元处计算误差梯度)、连接权值的最速下降调整。各种不同的激活函数也是在这个层次上定义的。
- **NeuralNet 类网络的主要接口**。实际运行和训练网络所需的函数都在这里。
- **NNAIControl 是神经网中将要用到的 Controller 类**。它提供了神经网络在游戏中的特定用途。我们的控制器将用于处理几种不同的控制“模式”，即网络的训练，与之对应的是训练完毕之后实际使用网络完成 AI 任务。

21.5.1　NeuralNet 类

为了在游戏中使用神经网络，我们需要构造实际的网络结构，用数据集训练网络，然后用网络来决定如何处理新数据。

程序清单 21-1 是 NeuralNet 类的定义；程序清单 21-2 是它的实现。

程序清单 21-1　NeuralNet 的定义

```
class NeuralNet
{

public:

    NeuralNet(int nIns,int nOuts,int nHiddenLays,int
nNodesinHiddenLays);
    void Init();

    //access methods
    void Use(vector<float> &inputs,vector<float> &outputs);
    void Train(vector<float> &inputs,vector<float> &outputs);
    float GetError() {return m_error;}
    void WriteWeights();
    void ReadWeights();
protected:
    //internal functions
    void AddLayer(int nNeurons,int nInputs,int type);
    void SetInputs(vector<float>& inputs);
    void FindError(vector<float>& outputs);
    void Propagate();
    void BackPropagate();

    //data
    vector<NLayer> m_layers;
    NLayer* m_inputLayer;
    NLayer* m_outputLayer;

    float m_learningRate;
    float m_momentum;
    float m_error;

    int m_nInputs;
    int m_nOutputs;
    int m_nLayers;
    int m_nHiddenNodesperLayer;
    int m_actType;
    int m_outputActType;
};
```

Init()是网络的初始化函数。它通过反复调用 AddLayer()实例化每层神经元，从而建立起网络的内部结构。建立的系统既可以处理只有一个输入和输出层(感知机)的简单网络，也可以处理一般的多层神经网络。

Propagate()函数将输入引入网络，并沿网络向前推进。BackPropagate()的操作则相反，求出最终输出的误差并沿网络从最后一层到第一层找到正确的误差梯度。

Train()和 Use()是实际使用神经网络时的两个主要函数。在训练网络时，用需要训练的输入输出对调用 Train()函数。它会将输入沿着网络传播，根据期望输出找到误差，并将误

差反向传播。Use()函数假设网络已经训练完毕。它只是将输入作用在网络上并返回网络的输出。

FindError()在训练中根据给定的输出确定网络的输出误差。它利用激活函数的微分，确定每个输出神经元的误差梯度，这些误差梯度将用于把必要的改变反向传播给网络的连接权值，以便权值达到最优。

程序清单 21-2 NeuralNet 的实现

```
//--------------------
void NeuralNet::Init()
{
    m_inputLayer = NULL;
    m_outputLayer = NULL;
    m_actType = ACT_BIPOLAR;
    m_outputActType= ACT_LOGISTIC;
    m_momentum = 0.9f;
    m_learningRate = 0.1f;

    //error check
    if(m_nLayers<2)
        return;

    //clear out the layers, incase you're restarting the net
    m_layers.clear();

    //input layer
    AddLayer(m_nInputs, 1, NLT_INPUT);

    if(m_nLayers > 2)//multilayer network
    {
        //first hidden layer connect back to inputs
        AddLayer(m_nHiddenNodesperLayer, m_nInputs, NLT_HIDDEN);

        //any other hidden layers connect to other hidden outputs
        //-3 since the first layer was the inputs,
        //the second (connected to inputs) was initialized above,
        //and the last one (connect to outputs) will be initialized
        //below
        for (int i=0; i<m_nLayers-3; ++i)
            AddLayer(m_nHiddenNodesperLayer, m_nHiddenNodesperLayer,
                     NLT_HIDDEN);

        //the output layer also connects to hidden outputs
        AddLayer(m_nOutputs, m_nHiddenNodesperLayer, NLT_OUTPUT);
    }
    else//perceptron
    {
        //output layer connects to inputs
        AddLayer(m_nOutputs, m_nInputs, NLT_OUTPUT);
    }
```

```
    m_inputLayer = &m_layers[0];
    m_outputLayer= &m_layers[m_nLayers-1];
}

//--------------------
void NeuralNet::Propagate()
{
    for (int i=0; i<m_nLayers-1; ++i)
    {
        int type = (m_layers[i+1].m_type == NLT_OUTPUT)?
                        m_outputActType : m_actType;
        m_layers[i].Propagate(type,m_layers[i+1]);
    }
}

//--------------------
void NeuralNet::BackPropagate()
{
    //backprop the error
    for (int i=m_nLayers-1; i>0; --i)
        m_layers[i].BackPropagate(m_actType,m_layers[i-1]);

    //adjust the weights
    for (i=1; i<m_nLayers; i++)
        m_layers[i].AdjustWeights(m_layers[i-1],
                                  m_learningRate,m_momentum);
}

//--------------------
void NeuralNet::Train(vector<float> &inputs,vector<float> &outputs)
{
    SetInputs(inputs);
    Propagate();
    FindError(outputs);
    BackPropagate();
}

//--------------------
void NeuralNet::Use(vector<float> &inputs,vector<float> &outputs)
{
    SetInputs(inputs);
    Propagate();
    outputs.clear();

    //return the net outputs
    for(int i =0;i< m_outputLayer->m_neurons.size();++i)
        outputs.push_back(m_outputLayer->m_neurons[i]->m_output);
}
```

```
//--------------------
void NeuralNet::SetInputs(vector<float>& inputs)
{
    int numNeurons = m_inputLayer->m_neurons.size();
    for (int i = 0; i<numNeurons; ++i)
        m_inputLayer->m_neurons[i]->m_output = inputs[i];
}
//--------------------
void NeuralNet::FindError(vector<float>& outputs)
{
    m_error = 0;
    int numNeurons = m_outputLayer->m_neurons.size();
    for (int i=0; i<numNeurons; ++i)
    {
        float outputVal = m_outputLayer->m_neurons[i]->m_output;
        float error = outputs[i]-outputVal;
        switch(m_actType)
        {
        case ACT_TANH:
            m_outputLayer->m_neurons[i]->m_error = m_outputLayer->
                                        InvTanh(outputVal)*error;
            break;
                case ACT_BIPOLAR:
            m_outputLayer->m_neurons[i]->m_error = m_outputLayer->
                                    InvBipolarSigmoid(outputVal)*error;
            break;

        case ACT_LOGISTIC:
        default:
            m_outputLayer->m_neurons[i]->m_error = m_outputLayer->
                                    InvLogistic(outputVal)*error;
            break;
        }
        //error calculation for the entire net
        m_error += 0.5*error*error;
    }
}
```

21.5.2 NLayer 类

因为神经网络的主要操作是在从一层到另一层的连接上，它是系统的主要部分。程序清单 21-3 是 NLayer 类的定义，程序清单 21-4 是它的实现。

程序清单 21-3 NLayer 的定义

```
class NLayer
{
public:
    NLayer(int nNeurons, int nInputs, int type = NLT_INPUT);
```

```
    void Propagate(int type, NLayer& nextLayer);
    void BackPropagate(int type, NLayer& nextLayer);
    void AdjustWeights(NLayer& inputs,float lrate = 0.1f,
                       float momentum = 0.9f);

    //activation functions
    float ActLogistic(float value);
    float ActStep(float value);
    float ActTanh(float value);
    float ActBipolarSigmoid(float value);
    void ActSoftmax(NLayer& outputs);

    //derivative functions for backprop
    float DerLogistic(float value);
    float DerTanh(float value);
    float DerBipolarSigmoid(float value);

    //data
    vector<Neuron*> m_neurons;
    int m_type;
    float m_threshold;
};
```

程序清单 21-4　NLayer 的主要实现

```
//--------------------
void NLayer::Propagate(int type, NLayer& nextLayer)
{
    int weightIndex;
    int numNeurons = nextLayer.m_neurons.size();
    for (int i=0; i<numNeurons; ++i)
    {
        weightIndex = 0;
        float value = 0.0f;

        int numWeights = m_neurons.size();
        for (int j=0; j<numWeights; ++j)
        {
            //sum the (weights * inputs), the inputs
            //are the outputs of the prop layer
            value += nextLayer.m_neurons[i]->m_weights[j] *
                     m_neurons[j]->m_output;
        }

        //add in the bias (always has an input of -1)
        value+=nextLayer.m_neurons[i]->m_weights[numWeights]*-1.0f;

        //store the outputs, but run activation first
        switch(type)
        {
```

```
            case ACT_STEP:
                nextLayer.m_neurons[i]->m_output = ActStep(value);
                break;
            case ACT_TANH:
                nextLayer.m_neurons[i]->m_output = ActTanh(value);
                break;
            case ACT_LOGISTIC:
            nextLayer.m_neurons[i]->m_output = ActLogistic(value);
                break;
            case ACT_BIPOLAR:
                nextLayer.m_neurons[i]->m_output =
                                                ActBipolarSigmoid(value);
                break;
            case ACT_LINEAR:
            default:
                nextLayer.m_neurons[i]->m_output = value;
                break;
        }
    }
    //if you wanted to run the Softmax activation function, you
    //would do it here, since it needs all the output values
    //if you pushed all the outputs into a vector, you could...
    //uncomment the following line to use SoftMax activation
    //outputs = ActSoftmax(outputs);
    //and then put the outputs back into the correct spots

    return;
}

//--------------------
void NLayer::BackPropagate(int type, NLayer &nextLayer)
{
    float outputVal, error;
    int numNeurons = nextLayer.m_neurons.size();
    for (int i=0; i<numNeurons; ++i)
    {
        outputVal = nextLayer.m_neurons[i]->m_output;
        error = 0;
        for (int j=0; j<m_neurons.size(); ++j)
            error+=m_neurons[j]->m_weights[i]*m_neurons[j]->m_error;
        switch(type)
        {
            case ACT_TANH:
                nextLayer.m_neurons[i]->m_error =
                                        DerTanh(outputVal)*error;
                break;
            case ACT_LOGISTIC:
                nextLayer.m_neurons[i]->m_error =
                                    DerLogistic(outputVal)*error;
                break;
```

```
            case ACT_BIPOLAR:
                nextLayer.m_neurons[i]->m_error =
                                DerBipolarSigmoid(outputVal)*error;
                break;
            case ACT_LINEAR:
            default:
                nextLayer.m_neurons[i]->m_error = outputVal*error;
                    break;
        }
    }
}

//--------------------
void NLayer::AdjustWeights(NLayer& inputs,float lrate,
                                float momentum)
{
    for (int i=0; i<m_neurons.size(); ++i)
    {
        int numWeights = m_neurons[i]->m_weights.size();
        for (int j=0; j<numWeights; ++j)
        {
            //bias weight always uses -1 output value
            float output = (j==numWeights-1)? -1 :
                                inputs.m_neurons[j]->m_output;
            float error = m_neurons[i]->m_error;
            float delta = momentum*m_neurons[i]->m_lastDelta[j] +
                        (1-momentum)*lrate * error * output;
            m_neurons[i]->m_weights[j] += delta;
            m_neurons[i]->m_lastDelta[j] = delta;
        }
    }
}
```

该类包含激活函数及其微分。此外，每一层包含其组成神经元的链表，以及 m_type 域(它是输入层、隐含层还是输出层?)、阈值(一般设置为 1.0f，对于简单的阶跃激活函数，这个值表示神经元必须进行累积以点火的输出值，对于 S 形函数这个值表示的是函数增益，它对应于输出图中 S 形的光滑度：很小的值对应直线，很大的值代表阶跃函数的形状)。

Propagate()是对网级相应函数的层级扩展。它对层内神经元依次循环执行标准神经网络公式：将神经元的全部输入求和，乘以相应的连接权值，然后代入指定的激活函数运行。

BackPropagate()也是该操作在层级特定的延续。它累加每个神经元上的总权值，在通过激活函数的微分运行输出之后，将总权值乘以输出值从而计算出梯度。系统已提供了几个激活函数。标准对数函数得到的值位于 0 和 1 之间。tanh 和 bipolar sigmoid 函数得到的值都位于 - 1 到 1 之间。线性函数相当于没有激活函数，即输出并未缩放。

AdjustWeights()在权值上执行最速下降方法，因为我们已经计算了我们所要的增量的梯度。最速下降是一种贪婪算法，也就是说它很容易终止于局部极小值 minima，所以用这种方法一定要谨慎。因此，在权值调整中我们用动量，它意味着更频繁的调整，以便出现

大的变化，因为先前的变化具有更高的优先级。这有助于减轻最速下降的局部最小问题，但它却使训练变慢，所以需要调整动量值。

21.5.3 NNAIControl 类

NNAIControl 类作为神经网络技术的 AI 控制器。该类包含网络本身和特定技术用途的代码，将其和 AIsteroids 游戏联系在一起。正如读者在定义中(程序清单 21-5；程序清单 21-6 给出了一些重要函数的实现)可以看到的，该类存储了全部常见的控制器信息(感知机数据和更新方法，以及 FSMAIControl 类继承的方法，因此还可以处理 AI 飞船的状态)，还包含用于训练和使用神经网络的所有数据和功能。

程序清单 21-5 NNAIControl 类头的定义

```
class NNAIControl: public FSMAIControl
{
public:
    //constructor/functions
    NNAIControl(Ship* ship = NULL);
    ~NNAIControl();
    void Update(float dt);
    void UpdatePerceptions(float dt);
    void Init();
    void Reset();
    void GetNetOutput();
    void TrainNetAndSave();
    void ReTrainNetAndSave();

    //perception data
    float m_powerupScanDist;

    //network output variables
    bool m_shouldThrust;
    bool m_shouldTurnLeft;
    bool m_shouldTurnRight;

private:
    int m_numIterationsToTrain;
    int m_numSavedTrainingSets;
    float m_maximumAllowedError;

    //network input variables
    float m_speedMovingTogether;
    Point3f m_nearestAsteroidDelta;
    float m_shipMovingDirection;

    //net, used for training and for actual usage in game
    NeuralNet* m_net;
    vector<float> m_inputs;
    vector<float> m_outputs;
```

```
    int m_numInputs;
    int m_numOutputs;
    int m_numHiddenLayers;
    int m_numHiddenNodes;
    int m_netMode;
};
```

根据我们是将控制器实例化为训练模式、重训练模式还是一般的“使用”模式，该类的构造函数会自动完成所需的任务。在训练模式中，网络由训练函数实例化，并在结束执行之后关闭。通常的游戏使用模式会立即实例化网络，因为游戏会马上会使用网络以避免潜在的障碍。

在通常的训练模式中，并没有真正的 AI 在运行，因为训练需要用到玩家的实际输入。正如读者在 Update()函数中可以看到的，当 m_willCollide 感知机为真时，NNAIControl 结构存储了将作为网络输入输出的变量。当收集了 1000 个数据集后，Update()将实例化并用数据训练网络，最后保存网络权值以便以后重用。

重训练模式的工作机制是从文件中加载保存的输入与输出训练数据并训练网络，而不是从游戏退出。当想尝试不同的网络设计时(如调整隐含层的数目或节点数、使用不同的激活函数或改变训练迭代的次数等)，重训练模式很有用。当然，如果想改变输入或输出的数目，那么需要用通常的 NM_TRAIN 模式重新捕获新的训练数据。

使用模式实际是通过运行一个有限状态机(FSM)来控制飞船的。一种稍作调整的躲避状态(新的类 StateNNEvade 中的 Update()函数见程序清单 21-7)，然后利用控制器神经网络的输出决定在即将碰撞的情况下如何躲避。NNAIController::Update()函数的作用是决定网络的输出，检查碰撞感知并在必要的情况下更新网络输出。GetNetOutput()通过网络运行值，得到当前输出并将输出转换回布尔值。读者可能会问，为什么不让网络直接输出布尔值？这是因为使用模拟值更容易确定误差梯度信息，这将有助于我们更好地、更快地训练网络。此外，可以从网络中确定我们想要的推广程度。如果输出是 0.4f，那么可能有一些系统仍为正输出；游戏也可能设置了将会发生的第二个动作，或者调整基本动作以考虑在当前输入数据情况下较低的网络输出。相反的情况是想在做出动作之前得到很高的输出，但是同样地，如果网络的输出是模拟值而不是单纯的数字，那么做出这类决策会容易得多。

程序清单 21-6　NNAIControl 的函数实现

```
//--------------------
NNAIControl::NNAIControl(Ship* ship):
FSMAIControl(ship)
{
    m_net = NULL;

    Init();

    if(m_netMode == NM_USE)
    {
        m_net = new NeuralNet(m_numInputs,m_numOutputs,
                              m_numHiddenLayers,m_numHiddenNodes);
```

```
        m_net->ReadWeights();
    }
    else if (m_netMode == NM_RETRAIN)
    {
        m_numSavedTrainingSets = 1000;
        ReTrainNetAndSave();
    }

}

//--------------------
void NNAIControl::Update(float dt)
{
    Ship* ship = Game.m_mainShip;
    if(!ship)
    {
        m_machine->Reset();
        return;
    }

    switch(m_netMode)
    {
        case NM_TRAIN:
            UpdatePerceptions(dt);
            if(m_willCollide)
            {
                //write test data to file
                FILE* pFile;
                if ((pFile =fopen("NNtrainingdata.txt","a"))== NULL)
                    return;

                fprintf(pFile,"%f %f %f %f ",
                            m_nearestAsteroidDelta.x(),
                            m_nearestAsteroidDelta.y(),
                            m_speedMovingTogether,
                      m_shipMovingDirection);
                fprintf(pFile,"%d %d %d ",ship->IsThrustOn(),
                  ship->IsTurningRight(),ship->IsTurningLeft());

                m_numSavedTrainingSets++;
                m_inputs.push_back(m_nearestAsteroidDelta.x());
                m_inputs.push_back(m_nearestAsteroidDelta.y());
                m_inputs.push_back(m_speedMovingTogether);
                m_inputs.push_back(m_shipMovingDirection);
                m_outputs.push_back(ship->IsThrustOn());
                m_outputs.push_back(ship->IsTurningRight());
                m_outputs.push_back(ship->IsTurningLeft());

                fclose(pFile);
            }
```

```
            if(m_numSavedTrainingSets==NUM_TRAINING_SETS_TO_AQUIRE)
            {
                TrainNetAndSave();
                Game.GameOver();
            }
            break;

        case NM_RETRAIN:
            Game.GameOver();
            break;

        case NM_USE:
        default:
            UpdatePerceptions(dt);
            if(m_willCollide)
                GetNetOutput();
            m_machine->UpdateMachine(dt);
            break;
    }
}
//--------------------
void NNAIControl::TrainNetAndSave()
{
    m_net = new NeuralNet(m_numInputs, m_numOutputs,
                          m_numHiddenLayers, m_numHiddenNodes);

    vector<float> tempIns;
    vector<float> tempOuts;
    for(int i =0;i< m_numIterationsToTrain;++i)
    {
        for(int j = 0;j< m_numSavedTrainingSets; ++j)
        {
            tempIns.clear();
            tempOuts.clear();
            //get training set inputs
            for(int k = 0;k<numInputs;++k)
                tempIns.push_back(m_inputs[k+j*numInputs]);
            //get training set outputs
            for(k = 0;k<numOutputs;++k)
                tempOuts.push_back(m_outputs[k+j*numOutputs]);

            m_net->Train(tempIns,tempOuts);
        }
        float totalError = m_net->GetError();
        if(totalError < m_maximumAllowedError)
        {
            //save out net and exit
            m_net->WriteWeights();
            return;
```

```
        }
    }
}

//--------------------
void NNAIControl::ReTrainNetAndSave()
{
    FILE* pFile;
    if ((pFile = fopen("NNtrainingdata.txt","r")) == NULL)
        return;
    m_net = new NeuralNet(m_numInputs,m_numOutputs,
                          m_numHiddenLayers,m_numHiddenNodes);

    vector<float> tempIns;
    vector<float> tempOuts;
    for(int i =0;i< m_numIterationsToTrain;++i)
    {
        for(int j = 0;j< m_numSavedTrainingSets; ++j)
        {
            tempIns.clear();
            tempOuts.clear();
            //get training set inputs
            for(int k = 0;k<m_numInputs;++k)
            {
                float temp;
                fscanf(pFile,"%f ",&temp);
                tempIns.push_back(temp);
            }
            //get training set outputs
            for(k = 0;k<m_numOutputs;++k)
            {
                float temp;
                fscanf(pFile,"%f ",&temp);
                tempOuts.push_back(temp);
            }

            m_net->Train(tempIns,tempOuts);
        }
        float totalError = m_net->GetError();
        if(i> 100 && totalError < m_maximumAllowedError)
        {
            //save out net and exit
            m_net->WriteWeights();
            return;
        }
    }
}

//--------------------
void NNAIControl::GetNetOutput()
```

```
{
    //clear out temp storage
    m_inputs.clear();
    m_outputs.clear();

    //set up inputs
    m_inputs.push_back(m_nearestAsteroidDelta.x());
    m_inputs.push_back(m_nearestAsteroidDelta.y());
    m_inputs.push_back(m_speedMovingTogether);
    m_inputs.push_back(m_shipMovingDirection);

    //get output values
    m_net->Use(m_inputs,m_outputs);
    m_shouldThrust = m_outputs[0] > BOOL_THRESHOLD;
    m_shouldTurnRight = m_outputs[1] > BOOL_THRESHOLD;
    m_shouldTurnLeft = m_outputs[2] > BOOL_THRESHOLD;
}
```

程序清单 21-7　StateNNEvade::Update()方法

```
//--------------------
void StateNNEvade::Update(float dt)
{
    NNAIControl* parent = (NNAIControl*)m_parent;
    Ship* ship = parent->m_ship;

    if(parent->m_shouldThrust)//thrust
        ship->ThrustOn();
    else
        ship->ThrustOff();

    if(parent->m_shouldTurnRight)
        ship->TurnRight();
    else if(parent->m_shouldTurnLeft)
        ship->TurnLeft();
    else
        ship->StopTurn();

    parent->m_debugTxt = "Evade";
}
```

21.6　测试平台的性能

用这些参数和设置训练网络虽然较慢，但还是很成功的。成功主要基于捕捉到良好的躲避数据，这是大多数神经网络系统面临的问题。一旦提供了正确的数据，网络就可以利用许多相同的技术来躲避碰撞。

用最大可能的训练数据集训练神经网络可以得到最好的结果，尤其是在网络需要学习极其复杂的任务的情况下。不过，如果训练集(无论什么原因)不可能规模很大或具有足够的多样性，那么可能就需要提防数据的过拟合问题。过拟合的神经网络不能很好地推广，因为它过于严格地匹配了输入模式，而不再具有准确包含误差数据点的灵活性。克服上述问题的一种方法称为早终止(early stopping)，使得网络避免过拟合数据。这种技术是当曲线在推广和误差之间达到平衡时停止训练。人们想要得到的是一个准确的神经网络，在大多数情况下能够做出正确的决策，但如果输入变量不是很好仍然希望它能够智能地“猜测”出来。找到最佳的停止训练的时机是另一个棘手的问题，这需要根据经验和实验来解决。大多数系统的做法是监视网络输出的误差，当误差下降了很长一段时间并开始上升时停止训练。然而这并不是绝对和快捷的规则：网络可能只是找到了脱离局部极小点的方法，而并未找到克服过拟合导致的性能下降问题的方法。

21.7 优化

优化神经网络通常是优化训练阶段以得到更好的效果，因为大部分神经网络是离线使用的，并且利用已有的训练网络在游戏中做出决策速度很快。为加快算法的训练速度，应尽量排除不必要的输入(或将输入加入到更复杂的计算中)或隐含节点。此外，在可以接受的误差容限内进行实验，因为即使最大误差容限发生很小的降低，也可以节省成千上万次训练迭代。

神经网络系统使用中其他层次的优化是，减少用于构造可行网络设计的时间，以及创建高效、相关的训练数据。对于任何实用的神经网络而言，这两项任务都很困难，并且会占据程序员的大量时间。然而对过程的这一步进行优化，需要理解神经网络的工作机制并明确接下来的具体任务。总之，对将要用神经网络建模的关系了解得越多，在挑选合适的网络输入和选择所需的最小输出上就会做得越好，并且训练数据也会越好。如果这个过程令人感到困扰，那么可以参照下面的一些基本注意事项：

- 如果网络很容易受到局部极小值的干扰，出现误差时处于稳定状态，但仍高于所期望的水平，那么可能是因为所用的训练集太少，或者设置的隐含层可能太小(如果隐含层没有足够的神经元，那么网络在搜索最优解时就不具有足够的自由度)。
- 如果训练不太稳定(即误差频繁出现，似乎不会稳定下来或逐渐减小)，那么可能是因为设置了过多的隐含层神经元，网络得到了过大的试验空间。
- 正如前面介绍过的，当训练集太少时可能会发生过拟合问题，因为即使很简单的网络也可以存储有限数据的大量信息。另一点是，当在数据上进行了过多的训练迭代时可能会发生过拟合问题，即训练的时间过长。针对每个训练集尝试一下减少迭代次数。
- 如果使用了大量含有过多噪声的训练集，或者训练迭代次数不够，那么可能会出现欠拟合问题。欠拟合的神经网络只具有推广性，几乎没有准确性。如果数据噪声很严重，由于网络很难把噪声从真实数据中过滤出去，因而可能会使问题进一

步恶化。找到一种方式扩展这些训练集，使得真实数据点和噪声数据的差别显现出来，有助于解决这一问题。

- 如果误差在值之间摆动，那么可能是因为设置的学习速率过高，或者设置的动量可能过大。梯度下降是一种贪婪算法，如果步长(在这种情况下是学习速率)设置得太大，那么算法的性能会比较差。可能的解决方法包括减小学习速率(这可能有帮助，但也有可能大大延长训练时间)，动态改变学习速率(若网络的误差下降，则缓慢地增大学习速率；若误差上升，则减小速率)，甚至用开销更大的方法，如牛顿法(包括求误差的二阶导以精确地找到最近极小点)。

21.8　基于神经网络的系统的优点

神经网络是寻找输入条件之间抽象关系的一种好方法。它很适合以实用和优化的方式存储复杂知识。神经网络还具有以下优点：

- 神经网络可以提取相当复杂的数学函数解决方法。这些数学函数基本上近似到神经网络的权值中去，所以当在游戏中使用网络时，基本上节省了 CPU 执行实际数学函数的代价。已经在数学上证明，至少具有一个隐含层和非线性激活函数的神经网络，可以准确地描述几乎任何紧致状态集上的有限维向量函数。
- 神经网络具有很强的从非线性或不精确数据中获取信息的能力。它们能以人很难甚至无法理解的方式概括出数据之间的关联关系。一个训练良好、设计合理的神经网络所具有的概括能力甚至超过了人类专家。
- 一旦确定了合适的网络设计，训练所需的 CPU 时间只是试错法所需时间的很少一部分。
- 人们使得神经网络具有一定的“意义”。神经网络组织数据和知识的方式容易控制，因而对人而言很有吸引力，它比某些更复杂的方法容易调试或实验，例如模糊逻辑系统。

21.9　基于神经网络的系统的缺点

神经网络很适合用来解决某些问题。但它也不是免费的午餐。如何确定训练网络通常就是它的代价。这就转移到了另一个问题上。现在需要我们做的不是如何解决问题(这可能是一个指数复杂度的问题)，而是搞清楚如何训练神经网络使它能够解决问题(已被证明这是一个指数复杂度的问题)。其他问题还包括：

- 神经网络不是万能的，无用的输入会导致无用的输出。如果将任意的、庞大的甚至虚假的数据输入网络，那么网络很可能会找到它们之间的一些相关因素。但这并不表示想要那些相关因素的输出。事实上，神经网络很容易学习错误的东西。找到正确输入和训练数据的最大困难，在于排除我们不想让网络学习的那些输入和训练集关系。而通常在网络训练完毕之后才能发现这些不良关系，这时网络已经学习了我们不想要的抽象。可能只有到那时才会意识到网络行为不正常的原因。

但是这些相关因素可能非常不明显，以至于根本无法找到它们，因此只能用单纯的试错法解决问题。

- 神经网络是一个数学黑盒，因此很难甚至无法调试。一旦训练完毕，神经网络中的权值数据就是无法理解的。它不像决策树结构的节点那样具有直观含义，我们无法根据这些权值确定网络中发生的情况。网络中的信息以高度并行或高效的方式分布在连接上，而不是有序地存在于某种类型的层次结构中，甚至也不取决于训练的顺序。通常调试神经网络时，需要从头开始调整训练前的参数或数据，然后重新训练。
- 全部输入域必须都是数值型的。模糊值或者由表达式来表示可能更好的那些输入，不能用神经网络建模。在这种情况下可能更适合用层次式系统，用神经网络处理更直观的元素，而用一种不同开销的结构(如决策树，或简单有限状态机)处理特殊或定义不十分明确的情况。
- 神经网络实现起来比较困难，这是由于需要确定大量因素，但在确定这些因素的最佳方式上却没有相应的指导原则或规则。这些因素包括网络结构、输入输出的选择、激活函数、学习速率、训练数据问题等。此外网络对某些随机因素十分敏感，例如权值的初始化或冗余的输入。
- 过拟合，或噪声学习损失了网络的推广能力，必须用前面提到的技术加以克服。
- 神经网络有时会受到称为灾难性忘却现象的影响。当几乎完全训练好的网络接收额外的训练数据时，会彻底破坏前面的学习效果，就会发生这种情况。后期可能会从测试或焦点组得到反馈，从而表明需要增加神经网络的功能，对这种情况必须小心处理，除非给自己留出充裕的时间去处理网络变化带来的问题。
- 训练，特别是大规模搜索空间中的复杂学习情况，可能需要大量的训练数据和 CPU 时间。如果在质量测试中由于游戏 AI 的神经网络部分引起了缺陷，重新训练网络可能代价巨大，所以在决定用神经网络系统时一定要考虑到这一点。
- 神经网络可扩展性不好。大于 1000 个节点的神经网络很少见并且不太稳定。虽然其中的原因还不十分清楚，但是维度灾难(curse of dimensionality，有时这样称呼)似乎使得这些大型网络的学习能力受到破坏，即在搜索空间中运动的自由度太大，以至于网络可能不停地循环改变其权值，永远无法得出所要的解。幸运的是，在游戏中可能用到的神经网络几乎都没有理由达到如此大的规模。

21.10 范例扩展

本章所用的前向传播、反向传播训练神经网络决不是当前仅有的网络类型。网络类型数目繁多，并且每种类型的网络都是为了在特定的应用中达到一定的性能而特别设计的。有些模型实际上并不太适用于游戏设计，但是知道它们的存在仍很重要，这样才有利于未来的进一步开发。其中大部分来自学术领域或商用领域，这也是神经网络发展近 40 年来一直所围绕的领域。下面列出了一些其他类型的网络或对上述网络的扩充。注意，这里列举的并不涵盖全部类型。

21.10.1　其他类型的神经网络

(1) 简单递归网络。这些基本上是普通多层神经网络的变体。在这种模式下，网络的隐含层也反向连接到输入层，每个连接的权值为 1。固定的反向连接导致基本上保持了隐含层先前值的一个副本(因为在应用学习规则之前，网络沿连接传播)。因此，该网络能够保持一种状态，使得它可以完成诸如序列预测这类任务，这是超出标准多层网络的能力之外的。

(2) Hopfield 网络。这是为了模仿大脑的联想记忆而设计的网络，这些网络允许存储整个“模式”，然后由系统进行回忆。与大脑一样，如果系统不同部分之间的某些连接发生失效或破坏，回忆成功的可能性还是很高。这类结构使用完全相连的神经元，其中每一个只能存储布尔值。这类系统中没有专门的输入和输出神经元。反之，输入发生在全部神经元上，然后让其传播直到达到稳定状态，此时所有神经元的状态都被视为系统的输出。这些对于模式识别很有用，尤其是对模式的记忆。如果将一系列图像存储在一个 Hopfield 网络中，然后输入其中的一个图像，但损坏该图像的某些部分，网络通常能够确定输入的是哪张图像。对于 Hopfield 网络，能够直接计算出存储信息所需的节点数目以及其他附加信息，这是 Hopfield 网络所具有的额外特点。一般神经网络的节点数目在某种程度上不容易确定，与此不同的是，Hopfield 网络中的节点只是模式矩阵的分布式存储，所以它的节点数只是所要吸收信息量的函数。

(3) 机器委员会(Committee of machines)。在这种技术中，用同样的数据训练多个网络，但每个网络的初始化不同。然后在使用过程中，输入数据让全部网络运行，通过对每个单独网络的输出进行统计，所有网络对最终输出进行“投票”。统计上表明这有助于缓和神经网络存在的问题。然而它比普通网络在初始化时具有更大的开销。

(4) 自组织图(SOM)。对分类(SOM 用户通常称之为聚类)任务很有用，SOM 含有两个网络层：一个输入层和一个竞争层。连接到竞争层中任何一个神经元的全部输入的总和称为输入空间中的参考向量(reference vector)。SOM 实质上包含一系列输入向量，这些向量由竞争层中的一组神经元表示，通常排列成神经元的二维网格。SOM 采用一种称为竞争学习的无监督学习形式，因此得出层的名字。当把一种新的输入模式引入网络时，第一步是要找出具有最接近新模式的参考向量的竞争神经元。然后挑选“胜出”的神经元并将其作为网络中权值改变的焦点。不但这个神经元的权值发生改变，而且其邻域(定义为以胜出神经元为中心，一定网格距离内覆盖的所有神经元)内的神经元的权值也随它们与焦点神经元的距离按比例改变。邻域的大小随时间逐渐减小，因此当训练结束时，邻域的大小为零。这种模型的效果是，输入数据的组织在网络中成为分组的，因此最相似的输入将会靠近在一起。这类图的主要用途是直观表示大量高维输入中的关系和类别。在游戏设计中，它对于角色建模可能很有用，可以把行为的多种维度考虑在内，给 AI 系统提供更好的玩家想要的角色画面。

21.10.2　神经网络学习的其他类型

(1) 强化学习。本章介绍的反向传播技术是一种有监督学习，有人称之为老师指导下的学习(learning with a teacher)，因为为网络提供了目标输出值。强化学习还涉及监督，但

只是在评估上有帮助，因此被称为评论下的学习(learning with a critic)。当网络由于输入信息而输出一些值时，监督者判断结果是否良好。监督者可以由人类专家担当，也可以通过一个接收网络外部(环境，或者与网络有交互关系的其他实体)广播的额外输入信号提供给网络，这样可以使网络基本上无监督地运行。

(2) 无监督学习。这些技术使网络可以完全依靠自身进行训练。它不需要手工输入训练数据或目标输出。无监督学习的例子包括利用遗传算法找到最佳的神经网络连接权值(这一技术更像强化学习，即由遗传算法适应度函数完成评论任务)，扰动学习(迭代并随机地调整和测试权值以提高性能)，和竞争技术(包含多个神经元通过在网络中调整其权值来“竞争”学习权利；胜者通常基于提供给网络的输入而不是声明的输出)。无监督学习的另一分支涉及的是输出无法预先得知的问题。在这种问题中，网络的主要工作是分类、聚类、找出关系，否则就压缩特定区域的输入数据。SOM 就是一个例子。

21.11 设计上考虑的因素

神经网络绝不是一种“一刀切”的人工智能技术，尤其是在游戏设计中，其调试和扩展上的不灵活性，使之难以针对游戏进行调优。注意第 2 章“AI 引擎：基本组成和设计”中介绍的游戏引擎设计问题：解决方案的类型、智能体的反应能力、系统的真实性、类型、平台、开发限制以及娱乐限制。

21.11.1 解决方案的类型

对于(以非线性、黑盒形式)将输入映射到输出的简单模块化系统，神经网络非常适合。正因为如此，它们本质上更倾向于作为战术性的，而不是高层战略性的方案。训练一个神经网络以处理即时战略(RTS)游戏中的外交活动并不合适，因为这类任务实在太庞大、太复杂，需要太多的调整。但可以用它来决定在棒球中采用哪个动作来接球。这里我们要处理一个非常原子性的任务，有具体的输入(打过来的球、球员的方位以及他的技能)，以及具体的输出(每个可能的接球动作)。用一般技术处理映射这些元素的逻辑比较耗 CPU 时间或构造起来比较复杂。而一个小型神经网络消耗的 CPU 时间则少得多，并且如果游戏中不增加额外的接球动作，那么训练完成的网络就不需改变(在这种情况下，传统游戏逻辑可能总要改变)。

21.11.2 智能体的反应能力

神经网络可以优化 CPU 密集型的计算，所以它实际上可能有助于提高游戏智能体的反应速度。

21.11.3 系统的真实性

在许多系统中，神经网络有助于使系统更真实，主要因为它们是一般的模式匹配系统，也就是说它们不是针对特例的系统，例如 FSM 或只针对特定情况做出反应的脚本实体。因此，它们可能对某些情况做出错误的反应，但仍以一种似乎正确的方式，因为不管什么原

因模式仍然成立。人们经常遇到相同的问题，主要是因为我们的大脑也使用类似的一般情况模式匹配。因此看到人们不时撞在玻璃门上，或者扶住他们认为是墙的地方防止自己摔倒。这类行为是否可以用在游戏世界中，取决于动作的背景、所设计游戏的类型以及面向的玩家。主要面向成年玩家的喜剧冒险类游戏，可能需要捕捉这类小细节，以增加游戏的真实度和幽默性。但是面向儿童的纯粹的动作类游戏，则可能把这类细节看作“愚蠢 AI”的“失误”。

21.11.4　游戏类型和平台

游戏类型和平台并不是神经网络真正的限制因素。它们是真正的模块化技术，当有特别要用到其分类或预测的功能时，它们会比较有用。

21.11.5　开发限制

这是使用神经网络时需要考虑的主要因素。正如本章其他部分所讨论的那样，神经网络不但需要设计时间和人力上的前期投入，而且需要收集训练数据。神经网络的实际训练还需要大量时间，在开发后期还要处理调优问题。神经网络的在线学习(在游戏运行中学习)甚至需要时间和设计上的更大投入。如果因为 AI 系统中具有适应性的神经网络使得动作和游戏事件处于动态变化中，从而测试人员很难(甚至不可能)再现导致游戏崩溃的事件，那么可能会很难完成游戏的调试工作。再加上单纯对包含大量自适应和发展元素的游戏运行系统进行测试，就明显比按字典序测试静态内容要花费大得多的精力。

21.11.6　娱乐限制

本章已经讨论了神经网络驱动的系统在调整、优化以及完善上面临的困难。正因为如此，在游戏设计中，神经网络确实只适用于那些做一次后就让其自己运行、黑盒式的解决方案，而这正是神经网络所擅长的。

21.12　小结

神经网络是另一种可以帮助人们解决难题的人工智能技术，尤其对于非线性或非直观性质的问题，可以使用测试数据，也可以采用某种方式的无监督学习对系统进行训练。

- 大脑的工作机制是，称为神经元的脑细胞通过从轴突到树突的突触彼此传输电脉冲。
- 人工神经元具有一系列输入(每个被赋予一个权值)，内部偏置值，以及一个可选的激活函数及其输出值。
- 神经网络是以某种特定形式连接起来的神经元系统。通常的结构是一系列输入节点，然后是一系列隐含节点，随后是一系列输出节点。前馈网络只有单向连接。递归网络在两个方向上都是完全连接的。还有其他类型的系统。
- 当使用神经网络时，必须确定网络的结构，选择一种学习类型，并创建训练数据。然后可以实现神经网络，不断调整这一实现以优化结果。

- 神经网络方法的优点在于能提取和压缩复杂的数学函数，它们的推广能力强，在游戏中能快速使用，以及它们的运算对大多数人“具有意义”这一事实。
- 神经网络的不足在于它们实现和调试起来困难，对训练的敏感度高，训练和测试所需投入的时间长，以及它们通常对大规模问题的扩展性不好。
- 神经网络的扩展包括递归网络、Hopfield 网络、自组织图以及机器委员会。除有监督学习之外，其他学习技术包括强化学习和无监督学习。

22 其他技术备忘录

本章将讨论其他一些在游戏 AI 编程领域具有前景的人工智能技术。在剖析每种技术时，我们将分为以下几个部分：方法概述、使用中的一般注意事项、优缺点以及设计中需要考虑的问题。

22.1 人工生命

科学一直在寻找一些所谓“普适原则”的规律，也就是十分普遍和基础的规则或真理，以至于它们根本无法再细分，用它们来理解自然现象以及用这些公式去预测结果从来不会出错。当牛顿给出适合描述万物运动规律的数学方程，“证明”笛卡尔的事物时钟宇宙观(上帝时钟般设置了万物，包括星球、植物、动物以及任何非人类的事物)时，他给每个科学家一种整体的美好感觉。此后我们发现了大量牛顿理论不适用的情况，因此科学家再次寻求普适原则，这次是在无生命的世界中。现在我们可以如此近距离地观察生物组织的行为(甚至是亚原子粒子的行为)，因而我们对几乎所有系统所体现的多样化和复杂性更加感到惊奇。但对于无生命系统，通过将物理系统解构为可以隔离并研究的各个部分，我们可以将事物分解，提炼更深层的知识。然而这种方法却不适用于研究生命。生命的本质不允许这种解构，所以我们感到无所适从。生命科学家们对先前工作的某些例子也感到无所适从，因为基本的生命系统不复存在，取而代之的是 10 亿多年前开始的复杂进化。这就像一个从未学习过英语的其他种族的人通过阅读纽约时报来尝试学习写作一样。发现任何普适原则的机会都是非常有限的。因此，或许我们应该尝试构建我们自己的非常单纯的“生命模拟”，从而更多地了解生命的一般规律。

人工生命(artificial life 或 alife)是这样一个研究领域，它试图从虚拟计算机环境或其他人工手段重新创建生物现象，以期更好地了解自然生命。这个名称实际上涵盖了计算机科学与工程学科，虽然它们确实具有某些共同点。人工生命的一个主要原则是，生命只是一个反复遵循某些简单规则的自然突现属性。突现属性是指当生物超出其组成部分的能力时，表现出的一种特质或行为。

在游戏设计中，我们也搜索突现行为和游戏运行情况，这促使人们在游戏环境的限制下寻求获得新乐趣的方式时，研究或试图运用人工生命原则。

22.1.1　人工生命在游戏中的用途

运用了人工生命的一些热门游戏(这里所说的热门是指广受评论，两个游戏在商业上都称不上热门)包括第 14 章“著名的混杂游戏类型”中介绍的“黑与白”和“造物计划”。这两个游戏都含有运用了简单规则的生命(“黑与白”中玩家的图腾动物会盯着玩家看，并在满足它自身需要的同时讨好玩家；“造物计划”实际上对整个化学和遗传学系统进行建模，试图模拟“建造”一个完整的生命 Norns)，这些规则以有趣的方式组合，动态地制作出游戏作者并未特地编程实现的行为。

22.1.2　人工生命学科

涉及人工生命的部分学科包括：元胞自动机、自组织行为和种群、遗传算法、机器人学。

1. 元胞自动机

元胞自动机(Cellular Automata，CA)是一组算法，它用极其简单的规则表现惊人数量的复杂行为。其中最有名的 CA 之一是 Conway 的“生命游戏”，它是利用元胞的二维集合，其中每个可以是填充的，也可以是空缺的，并且每个都有 8 个邻居。然后将 4 个简单的规则应用到游戏中的元胞上，就可以看到大量复杂的行为。

应用在填充元胞上的“规则”：

- 只有一个邻居或没有邻居的元胞会因孤独而死。
- 有 4 个或 4 个以上邻居的元胞会因拥挤而死。
- 有两个或三个邻居的元胞可以生存。

应用在空缺元胞上的“规则”：

- 有 3 个邻居的元胞将因为出生而被填充。

Conway 元胞自动机中的某些构造甚至已被证明能够进行数学运算。其他的可以从自然中再生出各种结构，它们具有不可思议的细节，包括厂房结构、贝壳、珊瑚。Mirek Wojtowicz 编写的免费程序 Mcell，可以展示大量的 CA 行为。它具有一般的图形界面，可以在运行中构造和观察 CA，图 22-1 所示是它的一些输出。

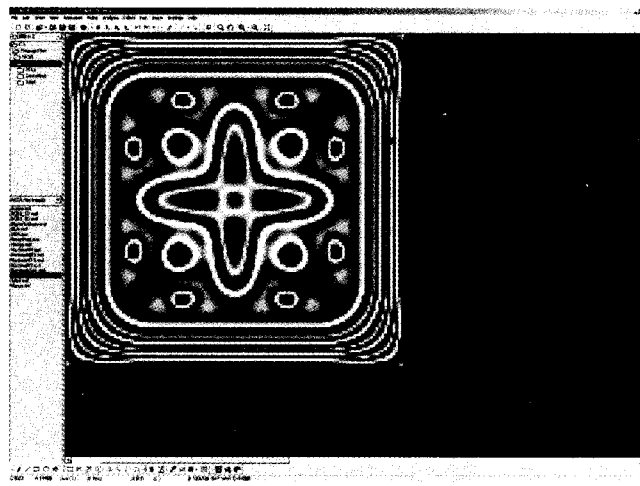


图 22-1　Mcell 程序的一些输出

CA 行为模式可以表现在宏观和微观层次。从微观上讲，这些模式可以用来模拟霉菌的生长、阿米巴的传播，从宏观上讲，它们可以用来发现交通堵塞或城市建设的趋势。

2. 自组织行为和种群

许多生物群体(如鱼、蚂蚁、鸟)可以在群体内快速容易地组织它们的行动，似乎群体成员具有一种统一的认识。对群体生物如何完成这种动作的研究，启发我们设计了用简单规则复制这类行为的算法。其中最著名的是 Craig Reynolds 的 Boid 研究[Reynolds87]，他描述了如何用几个关键概念(即分离以避免拥挤、向群体平均对齐、群体的位置内聚性)，达到对多种种群运动的惊人模拟程度。含有成群动物或人群的游戏几乎普遍使用了 Reynolds 的算法模拟他们的运动。这一领域还有其他一些有用的算法已经应用到游戏中，包括蚁群和蜂群研究(这两种动物都构造了十分复杂的结构，它们共同工作，具有一种与它们简单的智能不相称的组织)，经济学(货币理论表明经济似乎能够构造自身，或者崩溃或者并非基于游戏开发人员开始发现的稍微隐含的规则，例如大型多人在线 RPG[MMORPG])。

3. 遗传算法

遗传算法有时归属到人工生命中，虽然许多研究者认为这种分类是错误的。当认为遗传算法是一种很抽象的系统，遗传算法试图建模的事物(通过操纵基因获得进化)几乎完全不像我们所知的算法所用的简化版本，应该把遗传算法想象成计算机科学借鉴进化思想而设计出的有趣切线。

4. 机器人学

虽然机器人学研究的主要内容是制造系统，使其在人类无法胜任的环境下工作，可以说有些机器人学家正尝试制造人工生命，因此是物理的人工生命。他们并不是想要了解自然界，而是试图创造自己的自然界。在模拟生命的过程中，他们理解了自然界解决问题的方法以及自然界用其自身方法解决的问题，这是一个极具周期性的科学思考的例子。对这一问题不同研究者走向了相反的方向。有的致力于制造十分简单的机器人，但却可以与其他简单机器人交流，以建立母巢意志社区。其他的致力于设计人工训练的机器人，使之具有类似人类的行为，包括情绪反应、个人空间感以及个性发展。

22.1.3　优点

- **突现行为**。人工生命是我们目前半一致地建立突现情况的最佳方法之一。根据定义，行为的脚本化程度越高，或者行为序列越多，遇到的突现情况就越少。反之，突现行为最可能出现在开放式游戏(意味着游戏允许玩家和 AI 人物执行多种活动)中，简单行动可以以许多不同的方式结合，产生丰富的不同的最终行为。

- **行为重用**。人工生命技术要求开发人员用构造块设计游戏，将游戏不断分解，直到可以被表达为只有通过多次迭代才具有意义的简单规则。事实上，大部分人工生命游戏创作都容易编码，但需要很长时间来调优。

22.1.4　缺点

- **突现行为**。突现在游戏产业是一把巨大的双刃剑。人工生命能够凭空创造出可靠逼真的游戏效果。但是这种创造也许永远不会出现，使人们处于困境，没有真正的游戏可言。突现行为也许没有那么精彩，或者对大多数人而言过于困难。当您以自由的行为方式，不考虑任何结果时，既开启了神奇的大门，也开启了平凡的大门。
- **调优问题**。万一对游戏进行调优破坏了突现行为怎么办？游戏参数或者游戏运行系统发生的微小变化，很容易破坏您游戏中最引人注目的部分。事实上，有时修改一个程序错误会引起一连串事件的发生，可能使游戏运行情况变差。这种情况可以发生在任何游戏中，但是有时(多做一些工作)可以找出问题代码中的有利配置，将这些配置纳入无错误版本的代码中，从而尽量减小上述问题的影响。但是这种做法没有任何保证，因为所处理的是几个以非直观方式一起发生作用的因素。

22.1.5　游戏设计可以开发的领域

- 进一步利用更先进的种群技术，用于城市的人群等。
- 利用 Conway“生命游戏”的规则还可以模拟其他类型的运动，例如单细胞有机体的探索爬行、藤本植物的蔓延。
- 在 MMORPG 中的生物上运用人工生命技术，创造实际的生态世界，而不是用脚本怪兽随机生成点。

22.2　规划算法

规划算法的定义是：在行动之前决定一种行动，特别是利用与问题有关的更大范围内的知识，将行动链接在一起，以便得到更长远的解决办法。现实生活中一个鲜明的例子是毫无规划可言，但却很少出错的生物——普通苍蝇。虽然它的脑部很小，但仍能够以几乎滑稽的效率避开人们典型的追杀。但苍蝇不能，也决不会看到一扇关着的窗户，然后告诉自己“嗯，最好去找开着的门。”正因为这个简单的事实，所以通常窗台上的死苍蝇比房子中的其他地方要多。对于苍蝇而言，窗户就相当于游戏 AI 中不良相连的寻径节点，“如果我仍然一味尝试，我应该能走出去……”

概念上讲，大多数规划算法都遵循某些简单的形式：

- 将 AI 的处理能力分解为独立的操作符。
- 设计 AI 人物，以及游戏环境，使之可以被表示为状态集中的一个成员。

- 构造一棵表示状态间转换连接的树(列出引起这些转换的操作符)，或者将规则嵌入到每个状态中，详细表示出哪些操作符是可用的。然后 AI 通过在它当前状态的副本上应用这些转换操作符，并测试得出最佳动作以使其获得想要的行为，从而在局部工作内存中形成规划。

因此，规划包括了解人们所处的状态、想要处于什么状态，然后找到一串操作符使其从当前状态到达最终状态。在寻径问题中，操作符都是动作类型(身体跑动或走路以及包括像乘火车、使用远程运输这类动作)，这种情况下的状态是对寻径网络进行定义的图中的路径节点，它们构成一棵树，然后用 A*算法在这棵树上规划出一条路径。

采用某种方式，标准的决策问题模式，例如有穷状态机(FSM)和所有其他问题，都可看作一种预处理规划，它是对规划过程的一种优化。给出游戏的明确表示，以及一系列的低级操作符，规划算法应该能以合理的方式找到影响游戏状态所需的最佳行为。但由于规划算法代价可能很大，因而人们历来使用“硬编码”规划(对我们的情况而言，可以是状态机、脚本等)，一般情况下能够得到正确的行为。在更复杂的游戏环境中，我们设置的行为模式总有失效的时候，这些也是需要研究的地方。

22.2.1　当前在游戏中的使用状况

多数游戏已经使用了某些形式的规划算法，例如用于寻径的 A*搜索算法形式。寻径系统存储了广泛的游戏世界信息，使得 AI 控制的对象知道如何从 A 点走到 B 点。

某些游戏，特别是即时战略(RTS)游戏，使用同一系统执行其他规划任务。例如 RTS 游戏中的 AI 种族看到某个敌人的巡航部队——激光船。它现在知道敌人可以建造那些部队，为了针对这种船防卫它的海岸线，这个种族需要自己的激光船，或者一种称为反射塔的防御结构。通过一棵描述搜索任何给定技能或结构所需前提条件的技术树，AI 可以有效地生成一条“路径”，从它当前在技术树中的位置到为了建造两种所需部队之一而要到达的技术树中的位置。它甚至可以决定哪种防御方式“更近”(或更经济，或当前更适合的任何其他度量)，并选用那条路径。另外，当注意到敌人已经具有某种技术时，通过核对到达该技术的路径上所有必需具备的部队，AI 可以更新其针对敌人的技术树模型。

规划算法在游戏中的使用刚刚开始得到重视，主要是由于对 RTS 游戏的高级策略性思考。一些较早题材的游戏具有大量的先进策略，例如战争游戏，但大部分都是历史题材的战争游戏，遵循一种模仿历史战场的半脚本化模式，这通常要比对拿破仑建模并希望游戏中的他也会像历史上那样作战效果要好。

规划最终被视为一种主要的人类特征，然而高级 AI 系统越来越多地用到规划，具有更高的智能并且更像人类。例如在 FTPS 中，赋予敌人预期(anticipation)能力会使之更加逼真。预期是规划的另一种形式。采用这种技术后，AI 敌人可以看到玩家进入房间。通过使用规划算法，他会猜测玩家将在房间里做什么事。如果房间只有一个入口，里面有一个重要的宝物，规划算法运行得到的规划可能是得到那个宝物，然后再从同一扇门出来。敌人不但十分清楚玩家会从那扇门出来，而且由于他进行了行动规划，因此甚至可以估计出玩

家多久后会出现在门口。敌人可以进行非常有效的伏击。另一个规划方案可能是看到玩家的装备比较差，因此去追捕玩家。但敌人还检查玩家潜在的“规划”，并注意到在玩家前进的方向上有更好的装备。如果敌人健康值不佳，但仅仅因为具备火力上的优势而追捕玩家，那么他可能更聪明地不再追击而去寻找健康宝物，因为他知道玩家会忙着取装备宝物，从而自己有一些时间。

这是明显的 AI 行为，需要用于玩家面对高级对手的游戏环境中。在一个长的、充满故事性的游戏中，在玩家刚刚走进一间房子时，他肯定不想让任何第一人称射击/第三人称射击(FTPS)敌人破门而入，因为这样看起来并不好玩。但在一个难度级别很高的死亡之战设置中，玩家几乎都期望这类行为(因为他自己也是这样考虑问题的)，如果敌人不这么做，玩家还会嘲笑游戏引擎。下一个层次是让 AI 敌人预期玩家对自己的预期。因此如果敌人进入一个没有其他出口的房间，并且注意到玩家紧随其后，他可能从门里面向最佳伏击方位发射一枚弹药，或者干脆在房间里等候，直到玩家放弃在门外的伏击。这种实现可能代价较高，但可以看到这种概念，好处在于敌人不但可以猜测玩家的行动，而且当他感觉到玩家也在猜测他的行动时，放弃他原本的行动计划，从而表现得更高级，更像人类的做法。

通常，即使规划并未执行寻径，游戏也会用 A*搜索算法去寻径(因为大部分游戏都已经采用高效、典型的负载平衡 A*版本)。这通常是好的，尤其因为大部分计算机游戏并没有大量操作符或智能体状态。然而如果发现规划算法影响了游戏速度，应该考虑一些更为优化的搜索技术，例如手段-结局分析(MEA)。MEA 结合了树的向前搜索和向后搜索，并针对规划算法尽可能减小不必要的搜索。

另一种常见的规划优化称为修补重计算(patch recalculation)，“失败”的规划(例如由于某些游戏事件的发生导致某一步失效)并不是令整个规划作废，而是经过某个函数作用后，最终得出一个规划步骤，弥补前面遭到破坏的规划，从而修补其中的漏洞。只有当整个规划具有足够的长度，能够断定不需要丢弃整个规划而从头开始，这种方法才起作用。但对于冗长或计算代价较高的规划，这种方法提供了一种方式，使规划保持最新状态而不必每次都从零开始。

极小极大算法是另一种规划算法，它的思想是认为对手不会放过和自己作对的每一个机会。虽然极小极大算法大部分用于国际象棋这种棋盘类游戏，但是在某些回合制 RPG 游戏或者文明系列游戏中使用可能也会有好处，可以把作为这类游戏标准的脚本化、重复战斗序列替换掉，用基本的极小极大方法基于敌人和玩家的能力执行简单规划。特定的作战环节仍可以脚本化，但多数战役不会感到太单调且一成不变。规划器还可以考虑加入一些强化学习元素，例如不允许玩家采用相似的、非常有效的作战方式(通过有效的防御块)脱逃掉，从而给玩家提出更高的挑战。

22.2.2　优点

- 规划算法提供了更为前瞻性的智能行为。在现实生活中做出决策时很少不需要三思而行。事实上可以说，只要不是最基本的放松活动，都需要进行一些规划。即使在戴着摩托车头盔和手套时伸手去抓一下鼻子也需要规划。

- 规划是一个通用方法并且可以提供独立于数据的解决方案。因此，RTS 游戏中的同一寻径搜索算法还可以帮助 AI 按正确的顺序对技术进行研究，设置必要的命令以对敌人发起大规模进攻，并在建立其基地时注意到防止随着游戏的执行出现空间紧缺的情况。
- 与大多数通用算法一样，规划也可以层次式地实现。可以将规划系统分层，使每一层比较容易执行相应的规划，从而优化整体规划的代价。相应的例子是用高层规划器做出“建设强大军队，然后攻打下一座城”的规划。用下一层规划器分别针对“建设强大军队”和“攻打下一座城”做出较低层的规划。重复这一过程，直到做出足够底层的规划，这时的规划包括给每个牵涉到的人物发布行动命令。系统每一层刚好能够使用所需的足够细节以简化这一层次的规划器设计，但仍能给出有意义的规划。

22.2.3　缺点

- 如果尝试了不必要的冗长规划，那么会造成较高的计算代价。大部分游戏(甚至是战略游戏)很少要求其 AI 事先做出过长的规划。进行过长的规划代价很高，因为玩家的行为很难预测，因此长规划很少成功。规划的深度需要在规划的速度和灵活性与短期规划以避免出错之间仔细平衡。对于较长或代价较高的规划，可以利用修补重计算弥补消耗的时间。
- 如果规划过于一致或者适应新情况的时间太长，那么规划可能会使 AI 看上去比较迟钝或不够灵敏。当然这要受到人们所设计的游戏的限制：大规模文明类游戏可能比大多数游戏需要更多的规划，但它们也往往基于回合制，因而不会被认为“迟钝”。

22.2.4　游戏设计可以开发的领域

当创建需要多个步骤来实现目标的策略 AI 系统时，可以使用规划算法。前面提到的 RTS 任务是首选。但是动作类游戏也可以利用一些简单的规划。

- FTPS 对手可以用预期设置埋伏与陷阱。
- AI 司机可以规划更复杂的赛车动作，以一种更逼真的方式超过它的对手。不像可能欺骗的策略性“加速”，AI 司机可以在关键处做假动作，然后基于其他车辆的位置、剩余时间等因素规划动作，以便在关键时刻超越对手。
- 格斗游戏可以像拳击手那样规划组合动作：拳击手知道如果自己策略性地放弃防卫、使自己暴露在对手特定的一拳之下，那么他的对手将有可能遭受自己更具杀伤力的组合拳。
- 足球游戏可以利用规划，生成迷惑玩家的动作，或者利用比赛剩余时间上的优势。

22.3　产生式系统

产生式系统有时简称为专家系统——现在就可以使用一个原始版本。这是因为产生式系统基本上是基于规则的系统，它力求涵盖某一特定领域内的专家知识。产生式系统最简单的例子是用 AI 引擎中的条件 if...then 分支做出决策。在 AI 发展的早期，研究者极力创建通用计算机智能，他们相信借助合理而有力的逻辑规则，自己可以解决一切问题。

这一趋势持续到 1969 年，当时 Alan Newell 和 Herbert Simon 发表了通用问题求解(General Problem Solver，GPS)理论[Newell61]，其中给出了一个基本的规则集合，在某种程度上基于他们所认为的人类思维方式，一种称为手段-结局分析的过程。算法所需的全部就是对所要达到目标的声明，以及问题“规则”的一个集合。虽然人们发现 GPS 对于定义足够明确的简单问题和棋类问题而言很适用，但是没多久人们就发现它绝对不是通用问题的求解器。然而它确实引入了用动作作为操作符来转换当前世界状态这一概念。产生式系统就是从他们的理论上发展起来的。

有趣的是产生式系统却被用于与 GPS 用途(一般性)恰好相反的问题。这些系统现在用于存储极其具体的问题的专家知识。第一个专家系统用于解读质谱，此后这些系统又用于诊断特定的疾病，给出抵押税的建议。这种方法在游戏中很常见，大量代码用于存储曲棍球、吃豆、滚动跳跃等游戏所需的专家规则。然而，一个完整的产生式算法组织更有条理，分为 4 个部分：全局数据库、产生式规则、规则/情境匹配器、冲突消解函数(用于规则碰撞的情况)。全局数据库代表系统对周围环境当前所了解的全部事实。产生式规则是作为操作符以转换当前环境的实际 if...then 语句。匹配器一个函数，用于决定下次在数据库上使用哪个操作符以进一步接近目标。最简单的匹配器可以是一个简单的对数据库进行搜索、用规则 if 语句和当前世界状态进行比较的函数。但是专门的算法明显比原始版本要快得多。当多个规则同时与数据库匹配上时，就进行冲突消解。大多数消解模式很简单，甚至比较随意。

值得注意的一点是，传统上产生式系统只使用称为向前链接推理(即它们只能向前执行逻辑推理，例如：如果我身上起火，那么我应该跳湖)，但现代的扩充也允许使用向后链接推理(我刚跳进湖里，因此我可能身上起火了)。

在实际应用中，可以用产生式系统编写一般的游戏逻辑，用它作为规划系统(因为它们可以解决多步骤任务中的操作定序问题)，甚至可以用它作为记忆和学习单元(通过允许在全局数据库中新增和取消数据)。

真正的产生式系统在主流游戏中还不常见，但是学术前沿项目利用游戏来改善产生式系统。Soar 是一个由 Alan Newell、John Laird 和 Paul Rosenbloom(与 GPS 理论中谈到的 Newell 是同一个人)发起的项目，作为 Newell 认知理论的测试平台，从 1983 年就开始在学术界使用。Soar 提供了一个开源的 ANSI C 通用产生式系统，供认知科学家以及任何感兴趣的人使用。图 22-2 是 Soar 系统的高层概况。在与国防先进研究项目局(DARPA)合作完成了一些开发智能空战体的 Soar 工作后，John Laird 开始实验用 Soar 作为一种手段来提高 AI 在商业游戏中的性能。他在密歇根大学人工智能实验室的小组已成功地开发了 Soar 与雷神之槌 2 和天旋地转 3 的接口，并且为每个游戏设计了能力强、非脚本化的对手。利用

一个包含 700 多条规则的系统，这个小组创作了一个雷神电脑对手，它可以穿梭于任意游戏级别水平，利用一切武器和级别元素(如弹跳垫和个人物品)，为高水平玩家提出了挑战。此外，系统还执行规划，因此可以预见玩家的行为，生成用户水准的运动路线去尽可能多地收集宝物，并且进行智能化的伏击和攻击行为。

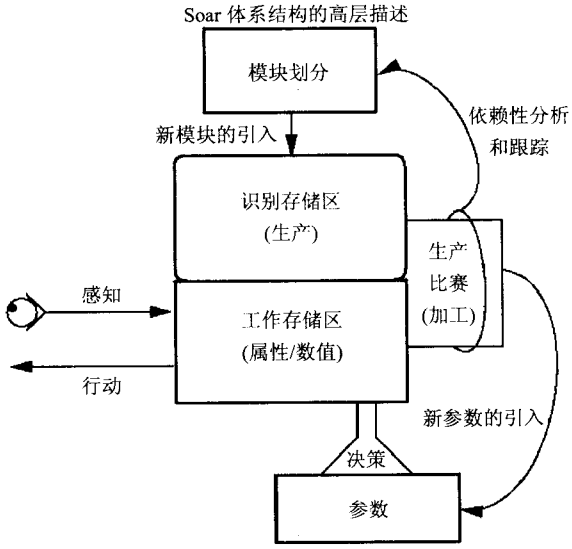


图 22-2　Soar 的系统结构图

22.3.1　优点

- **一般算法**。和规划算法一样，产生式系统的决策是独立于数据的，因此游戏的不同部分可以用产生式系统与基于规则的决策一起提供不同的系统。这可以是在独立的数据库上，不同的规则集上，或共享的任何组合。
- **研究**。在产生式系统上已经进行了大量的研究。已经设计了像 RETE 和 TRETE (RETE 的一种无状态变体)这类快速匹配算法，大大加快了产生式系统的条件检查。
- **目标指导的**。产生式系统总是目标指导的，意味着它们选取一个总体目标并找出实现该目标的一种方式。这比纯粹的反应行为体现出更高的智能性。
- **高度活跃**。这些系统可以十分活跃，利用一个良好的规则集合，它们可以提供与人类表现非常相似的效果。

22.3.2　缺点

也像规划器一样，产生式系统的计算代价可能较高，尤其是对于具有大型规则集或非任意匹配冲突消解的游戏。如果游戏必须找出匹配，并且必须执行大量计算以便在匹配中做出仲裁，那么使用这类系统的代价可能较高。

22.3.3　游戏设计可以开发的领域

用高度数据驱动的方式实现一个产生式系统是可行的，因此新的规则、感知等这类实体可以被添加到游戏世界中，只要把它们加入游戏的数据文件中去就可以实现这一点。给

定一个新的产生式规则集，产生式系统可以执行同样的算法。像这样的系统是高度可扩展的并且具有无限的重用性。

22.4　决策树

决策树是重新组织常用代码结构，使之更灵活、功能更强的另一种方式。这种方法不是编写数页 if...then 语句，您可以将每个语句实现为树上的一个节点，构造这样一棵树，从而可以对树而不是一堆嵌套的 if 进行遍历。图 22-3 是决策树结构的一个例子，表示的是 Joust 对手运行所需的 AI。树从根节点开始，也可以标为“问题”节点。这棵树要回答的问题是什么？在这个例子中是“我应当做什么？”注意我们的示意图是二叉决策树(BDT)，因为任何给定问题节点的答案都是布尔值，对或错的决策。特别的优化算法，甚至对任意决策类型的树不可用的插入和删除后重组方法对二叉树都是可用的(因为它们基本上都是红黑树结构)。

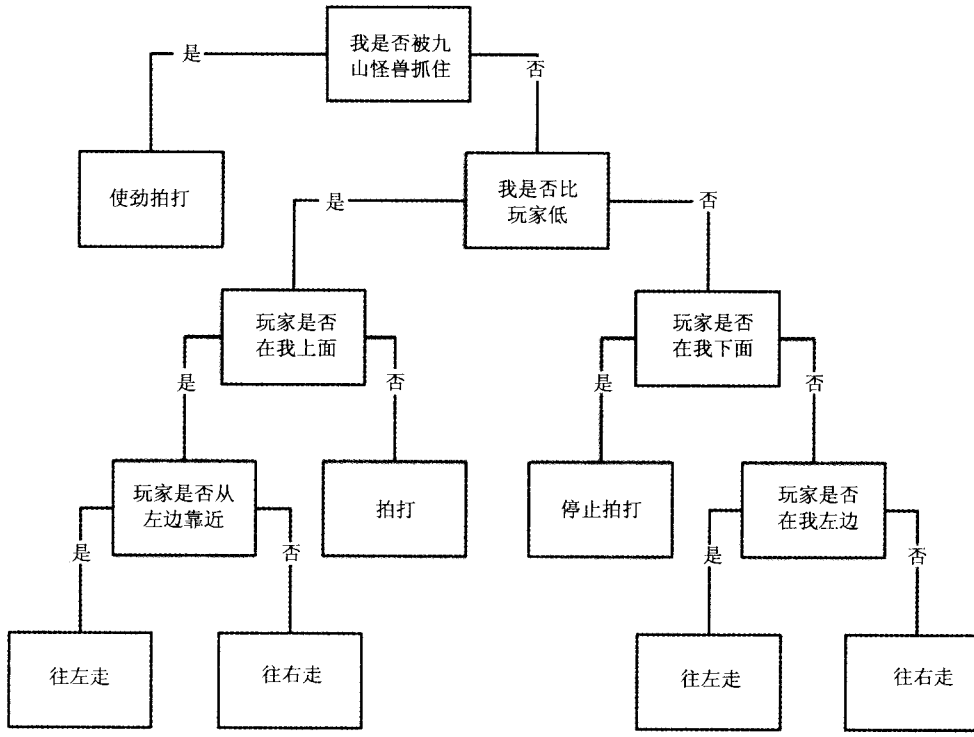


图 22-3　Joust 的决策树结构

决策树有两种常用的分支：分类树和回归树，它们都属于统计方法，采用算法手段通过使用一组训练数据从而构造决策树。两种方法都可以被认为是一种“穷人的神经网络”，它们尝试将输入推广到输出。这里有必要指出使用决策树和神经网络的不同之处：

- 神经网络黑盒系统，我们无法看出并理解其内部权值的意义，而一个完全“训练”的决策树很直观并容易理解。

- 决策树只能在(典型的)二元结果上处理严格的比较。正因为如此，它们的预测能力比较有限且僵化。神经网络则恰恰相反，它们善于很好地处理噪声密布的数据或存在空缺以及具有奇异跃变的数据。
- 决策树的输出总是离散值，如果使用了正确的激活函数，那么神经网络的输出可以是连续值。
- 决策树一次考虑一个变量，这称为单元(monothetic)算法。多个变量单独存在时的影响微小，而这些变量组合后会对行为产生重要影响，对于这种情况它无法处理。神经网络则被认为是多元的，因为它同时考虑多个变量。这是使神经网络难于处理的原因之一(表现在一个神经网络可能会找到多元关系，但这些关系并不是人们想要在测试数据中发现的，从而对其学习产生负面影响)，但这种性质也正是神经网络在决策树不适用的领域如此有效的原因。
- 通常神经网络的精确性远远高于基于树的方法。在一个传统例子数据集上，基于树的方法产生的相对误差统计检验值可能是反传神经网络的 10 倍以上。

分类树处理的是有类别输入变量的 BDT，而回归树处理的是连续型输入变量。BDT 允许不同变量类型的组合，但大多数用到这种方法的游戏 AI 问题通常会二者选一。分类任务可能包括在 FTPS 游戏中尝试确定玩家的表现更像哪种“类型”的选手(攻击手、狙击手、纯防御型选手、狂暴型选手等)，然后让 AI 执行为了应付那种类型的选手而特别优化的代码。回归任务多属于预测型任务，同一 AI 系统可能会预测它察觉到的玩家所遇到的游戏难度，然后如果这种难度过高或过低，那么它会调整其 AI 行为。

22.4.1　优点

- 决策树容易使用和理解。这使得它们很适合作为“基本通过”型 AI 系统，可以根据例子数据，找出程序员可能没有发现的逻辑联系。它还以简单的逻辑规则形式提供这些信息。这种可读性使得在必要时可以对其进行手工调试。
- 人们对其性能以及如何改进其功能进行了大量的研究。决策树是统计界及其派生的数据挖掘技术的一个重要部分。因为这两个领域都具有广阔的应用，决策树已经经过全球智囊团的剖析和重新组织。有许多实验性和真正的算法用于设计、训练、调试、校正以及优化决策树。
- 游戏 AI 具有足够的限制性，使得决策树在实际中比神经网络更合理。神经网络额外的建模能力通常不是必要的，决策树的可读性对最终调试和改进工作(二者都是典型的游戏开发环节，用神经网络几乎是不可能完成的)很有好处。此外，树状结构额外的复杂性有时可以用层次式决策树加以解决。树中的任何给定节点都可以是另外一整棵树，只要子树能够求解出对/错决策值就可以。

22.4.2　缺点

- BDT 往往比较脆弱，因为它们处理的是明显的状态，而且状态之间的边界是硬编码的。因此，和大量 if...then 语句一样，它们的扩展性一般不好，而且它们一旦达到某种复杂程度就会难以维护和扩充。

- 决策树的大小在精度和大小之间是直接的反相关，因此如果需要具体的输出，那么应使用神经网络(或其他方法)，因为高清晰度的输出会大幅度增加树的大小。
- 决策树往往缺乏策略性，这使得神经网络这类更具突现性的系统很受关注。但很多游戏并不寻求策略，它们寻求的是以可扩展的方式添加内容。
- 在非二叉树中，没有这样一种普遍共识，即树中多个孩子引入的额外复杂性会带来某些好处，考虑到构造和使用这样一棵树所花费的额外努力。针对特定游戏环境中的某个具体情况可以建立想要的结构，但对于需要在全 AI 系统针对各种任务使用的一般方案而言，标准框架更加可取。

22.4.3　游戏设计可以开发的领域

- 使用分类树进行简单的人物建模，因为人类的反应将比较有限，所以想让 AI 具有广泛的类别。
- 通过让数据驱动树，可以向树中插入或删除节点，以及调整二元检查参数，用它作为一种存储或学习形式。“黑与白”以此来记录玩家角色对周围环境的高级“思维”，为他提供了经验和训练，能在任何时候决定采取何种行动。这棵树在游戏进行中可能会发生巨大变化。
- 在游戏中使用一个通用 BDT 系统，能使几个二元决策具有组织和结构，通过用数据驱动这些定义，允许设计者以他们认为合适的方式做出上述二元决策。在篮球游戏中，可以用一棵树决定哪一方赢得了争球。同一系统也可用于从类似“我能越过防守上篮吗？”、“对防守做的假动作成功了没有？”甚至“模拟游戏中 X 组是否击败了 Y 组？”这样的二元决策中提供用户结果。因此，设计者可以对感知系统的一部分进行细微调整，为他们提供另一个参与游戏设计的维度。

22.5　模糊逻辑

我们在第 16 章中介绍了模糊状态机(FuSM)。但是，真正的模糊逻辑是一个复杂得多的系统，它有自己的一套逻辑证明和方法。模糊逻辑是布尔逻辑的扩展，由加州大学伯克利分校的 Lotfi Zadeh 博士在 1965 年提出[Zadeh65]，用于处理部分正确性的概念：完全正确和完全错误之间的某个值。Zadeh 最初用这个概念对他在处理自然语言时遇到的不确定性进行建模。

真正的模糊逻辑在非游戏应用中的例子都很少，更不用说用于游戏了。直到最近模糊逻辑才开始慢慢得到应用。据报道索尼公司的 PalmTop 使用了基于模糊逻辑的决策树对手写日本文字进行分类。模糊逻辑的另一个实现于 1993 年用在三菱的原型车上，他们展示了一个能够学习司机日常驾驶习惯的车内安全系统。该原型车内置了雷达系统，能够感觉到接近的障碍。该系统会尝试判断司机是否能对即将遇到的威胁做出反应，如果司机没能及时反应，系统将会控制汽车以避免碰撞。

真正的模糊逻辑允许人们在方程或规则上完全使用模糊值进行计算，例如：

```
if Health is low AND WeaponStrength is lame OR Bravery is meek, then
Camping is high
```

上述公式中有 4 个模糊变量：Health、WeaponStrength、Bravery 和 Camping。Camping 是输出变量，其余的是输入变量。与每个变量相关联的还有模糊隶属函数或模糊子集方法，用于确定相对模糊值：low、lame、meek、high。这些函数专门用于在变量范围内定量衡量相对隶属度。因此，low 的范围可以从 0.0 到 1.0，取决于 Health 的值。为了确定上述语句的正确性，把每个隶属度函数应用到它的关联变量上，以确定正确的程度。然后这些正确性值再由规则中定义的 AND 和 OR 操作符进行运算，它们在模糊逻辑中的定义如下：

```
Crisp: truth (x and y) = Fuzzy: minimum (truth(x), truth(y))
Crisp: truth (x or y) = Fuzzy: maximum (truth(x), truth(y))
```

由于在给定这些模糊参数之后，这类规则能够得出逻辑真值，因此它们可以用于“模糊逻辑产生式系统”，这种系统遵循普通产生式系统的所有规则，但它的全部推理则使用模糊逻辑。一般的推理过程分为 3 步(也可以分为 4 步)[Kant97]：

(1) 在模糊化的过程中，用定义在输入变量上的隶属度函数作用在它们的实际值上，以确定每个规则前提的正确程度。

(2) 在推理的作用下，计算每个规则的前提正确性值，并作用到每个规则的结论部分。这导致为每个规则的每个输出变量都分配了一个模糊子集。通常只有 MIN 或 PRODUCT 用作推理规则。在 MIN 推理中，输出隶属度函数在一定高度被剪掉，依据的是规则前提计算出的正确度(模糊逻辑 AND)。而在 PRODUCT 推理中，输出隶属函数根据规则前提计算出的正确度进行缩放。

(3) 在合成的作用下，分配给每个输出变量的全部模糊子集组合在一起，为每个输出变量形成一个模糊子集。通常仍使用 MAX 或 SUM。在 MAX 合成中，组合输出模糊子集是通过在推理规则(模糊逻辑 OR)分配给一个变量的全部模糊子集中逐点选取最大值而构造出来的。而在 SUM 合成中，组合输出模糊子集是通过在推理规则分配给输出变量的全部模糊子集上逐点求和而构造出来的。

(4) 最后是(可选的)去模糊化，用于将模糊输出集转变为一个明确的数值。去模糊化的方法有很多种(至少 30 种)。两种比较常见的技术是重心法和最大化法。对于重心法，明确的输出变量值是通过为模糊值找出隶属度函数的重心对应的变量值而计算出来的。对于最大化法，当模糊子集达到其最大正确性值时，其中的一个变量值作为输出变量的明确值。

这类系统用在模式识别、财政系统以及涉及密集数据分析的领域。通常只有在需要大量真实性的系统中才用到它们，因为在现实世界中几乎没有什么是黑白分明、完全一致并且精确定位的。另一方面，在游戏中这些情况都会发生，所以额外的运算对于大部分游戏而言并不是必须的。

模糊逻辑认识上的一个普遍误区是，以概率值的某种思考方式对模糊值进行思考。但这是两种不同的思考方式。能够说明两种思考方式不同之处的一个例子是“可饮用水”对象，即“液体”对象的一个子集。如果给两个杯子，分别标着“90%可饮用概率”和“90%可饮用模糊隶属度”，那么喝哪一杯？答案是 90%可饮用模糊隶属度。因为它对应的 90%表示的是“几乎完全”属于可饮用液体集合，而标着概率 90%的杯子 10 次有 1 次是有毒

的。实际上，考虑到杯子标签上对“可饮用”的定义，标着模糊值的杯子里可能是酒或者很酸的果汁，或者其他被认为不是完全“可饮用”的饮料。概率处理的是统计意义上某事可能正确的几率。模糊集隶属度讨论的是某事已经正确到什么程度了。

22.5.1　优点

对通常的布尔逻辑进行扩充以包含定义更为松散的变量，从而对于这类值的争议仍能在数学上得到证明。

22.5.2　缺点

计算代价大，包含大量模糊向量的模糊系统，可能会受规则开销的影响。已经出现了不同的方法用于克服这一点(例如 Combs 方法)，但仍存在问题。

22.5.3　游戏设计可以开发的领域

- 涉及大量未知或部分已知信息的游戏问题，如角色建模，对于模糊逻辑的使用很有意义。在典型的战略游戏中，例如 RTS、文明类游戏，甚至扑克游戏，角色建模很重要，如果致力于真正智能地解决问题。如果在设计游戏时并不让系统完全获得玩家的游戏数据，从而不会使系统出现作弊行为，那么 AI 系统将不得不依靠感知数据来建立玩家的模型以做出反应。这种信息很可能极为粗略，发现过程也没有一定顺序，甚至含有虚假内容(玩家为了欺骗 AI)，因此模糊系统很可能是解决这个问题的最佳方式。
- 对于在线或多人游戏，角色建模可能实际被游戏用作“助手”AI 实体，如果玩家要去洗手间或接听电话，那么它可以临时代替玩家玩游戏。这个特性与暂停类似，但不会打断游戏中的其他玩家。模糊系统会尝试模仿人们玩游戏的方式，当人们不在的时候继续用这种方式进行游戏。

22.6　小结

- 人工生命技术尝试用生物学中的经验从简单规则创建突现行为。
- 人工生命可以创建突现的重用行为，但它同时也具有不可预测性和脆弱性。
- 属于种群技术、其他类型的有机运动以及生态建设的人工生命技术在未来的游戏设计中可以考虑使用。
- 规划算法尝试在玩家开始之前用一个问题的附加信息来决定该做什么，希望不会出现局部错误。
- 规划已经采用 A*算法或某些变体以寻径形式用于很多游戏中。
- 规划可以为 AI 系统提供大量额外的智能，并且足够一般化以用于游戏引擎的不同部分。它们可能很耗 CPU，如果使用不当会造成 AI 反应迟钝。
- 规划在游戏中可以用于预期玩家的行动，更强意图性的 AI 动作会导致组合行为。
- 产生式系统是处理特定领域大量专家知识的通用方法。

- 产生式技术的特点是通用、已被深入研究、目标指导以及高度活跃。它可能很耗 CPU。
- 利用产生式系统组织游戏中基于规则的逻辑代码，用数据驱动产生式系统以最大程度地增加游戏的可扩展性，二者对游戏设计都很有帮助。
- 决策树是组织 if...then 结构的另一种方式，用二元分离方法优化结构和变量检查的数目。
- 分类树和回归树都与神经网相似，但也有许多重要的差异，包括可读性、易于处理的数据类型、输出类型、变量的考虑以及准确性。
- 它们比某些分类系统更容易调试，已被深入研究，很容易实现。它们可能比较脆弱、扩展性不好并且只能输出离散值。
- 游戏可以利用决策树进行简单的角色建模，像存储数据那样存储记忆，并提供一般的数据驱动系统，以确定二进制感知数据。
- 模糊逻辑是扩充普通逻辑以包含部分正确性值的一种方法。
- 它允许游戏使用定义更为松散的变量，同时依赖数学上已被证明的方法。模糊逻辑的 CPU 代价可能较高。
- 角色建模和助手 AI 系统可能是在游戏中使用模糊逻辑系统的方式。

第Ⅴ部分 AI 实战游戏开发

虽然每个游戏的 AI 引擎是不同的，但是在开发过程中还是存在一些普遍关注的问题。在本书的第Ⅴ部分，我们将专题阐述 AI 开发者在系统实现时应该关注的开发和娱乐方面的问题。

第 23 章“分层式 AI 设计”将介绍用来对游戏的 AI 系统进行分层化设计的方法，该方法把 AI 系统分成若干相互独立的游戏层，故称为分层式 AI。

第 24 章“AI 开发中普遍关心的问题”将提出一些 AI 游戏开发中不同层次上的人们普遍关注的问题：设计、娱乐以及最终产品。

第 25 章“调试”将讨论软件调试和参数调试的问题。这些问题都是 AI 程序设计的重要部分。另外，我们还将介绍一种基于 MFC 的运行时调试工具。通过引入该工具我们能发现任何游戏的程序实现和参数设置问题。

第 26 章“总结和展望”将以若干简要结论结束本书，并展望 AI 游戏的未来。

23 分层式 AI 设计

不管开发什么类型的游戏，AI 引擎的实现总包含着若干特定目标。设计和实现完整的 AI 系统并不是一件轻松的事情。我们将在本章中讨论适用于更大规模游戏的 AI 引擎设计模式，该模式称为分层式 AI 设计。它将帮助我们把 AI 系统分成若干更便于管理的子系统。我们将讨论该模式的各个部分，并在此过程中给出大量示例。最后将采用不同方法划分经典游戏《超级玛莉》。对于现代 AI 引擎，这些方法都是可实现的。

23.1 基本回顾

在第 2 章“AI 引擎的基本组成与设计”中，我们详细分析了 AI 引擎的各部件(主要是决策系统、感知系统和导航系统)。我们已经涉及了游戏中主要类型的编码技术，因此现在我们能更正确地阐述不同的却能使 AI 引擎各个部分正常工作的 AI 方法，以及能更好地判断哪些部分能协同工作。

大多数人认为游戏开发的第一原则是使游戏程序尽可能简单和清晰。设计和实现完整的 AI 引擎不是一件简单的事情，过分工程化的思想会扼杀程序员大脑中处于主导地位的经验和思想。分层式方法的目的是简化整个 AI 的实现和维持过程，该方法把 AI 任务划分成若干可彼此独立工作并且可有机组合成丰富游戏内容的组件，这些组件或者说分层系统都不是很复杂。

一个现实生活中的示例

为便于描述分层式设计技术，我们将先给出一个现实生活中的例子。假如某人坐在书桌边。突然隔壁房间有人喊他，然后他站起来，走到隔壁房间去看看谁在叫他。让我们中断刚才发生的事情，并通过游戏动画的形式重现该过程，为了表述方便，我们假设该 AI 角色名叫 Brian。

(1) Brian 坐在桌边并从事着某项名为 DoingWork 的工作，某时刻其接收到以游戏事件形式出现的输入(他的名字被某个人喊到)或者某感知变量发生变化(比如 DistanceToNearestDisturbance 或者 BeingCalled 等)，具体形式取决于感知系统的建立方式。

(2) Brian 的决策系统判定新来的感知数据非常重要，必须做出合适的反应。

(3) 当Brian的状态转移到(或者被赋予)称作Investigate的新状态时，Brian将获得一个新目标位置(隔壁房间)，这个位置要么是通过声音位置猜测算法获得，要么是通过作弊手段获得。

(4) Brian执行路径搜索算法来获得到达目标的路径，该算法会给出一系列路径节点，顺着这些路径节点可以到达目标位置。

(5) Brian的移动代码包含到达目的地的路径序列表、不同的路径节点上的动画以及如何更真实地播放这些动画的控制程序。

(6) 接着Brian准备进入最近的路径节点，但是书桌却阻碍了它。此时障碍躲避系统发挥作用，Brian从书桌旁边绕过并进入下一路径节点，障碍躲避系统可帮助Brian躲避任何移动物体。然后Brian打开门并朝声音方向走去。

(7) 当Brian离开房间后，Brian想到房间门还开着，而钱包还在桌上，因此他回来关了门，然后继续前进。

(8) 移动10步之后，看到Bob，认出那就是自己听到的声音。

(9) 新输入感知(Bob叫Brian的声音)改变了Brian的状态，从Investigate转变成ActCynical，在状态ActCynical下，Brian问："你到底想要干什么？"

整个游戏过程就这么进行，但有点值得我们注意，该过程最先描述的是"我听到有人喊我的名字，然后起身来看看是谁在叫我。"然而，通过AI角色实现该过程以及AI角色在该过程中所表现出来的一系列判断和智能行为水平比开头那句话所暗示的过程要复杂得多。这就是AI系统，每个任务都由许多子任务互相配合而构成。

23.2 分层式层结构

分层式AI设计是本书对某种技术的定义，该技术参考现实世界分层实现AI行为，并把它应用于AI引擎的组织与实现中。我们有很多研究该技术的理由：该技术使AI引擎结构更加清晰以及更便于引擎代码的理解、扩展和维护。由于其具备模块化的结构，分层式AI技术便于其他智能技术在系统中的应用，特别是更便于多次出现的技术的复用。另外，我们不必通过庞大的"AIPlayer.cpp"文件来实现AI系统，该文件中存储了大量处理AI角色在不同环境下的反应的实现代码。相反，我们把智能划分成多个不同层次：

- **感知和事件层**。对输入的信息进行过滤和分配，不同的感知数据传递给不同的行为层单元处理。
- **行为层**。具体描述如何执行给定动作。
- **动画层**。判断哪组动画更符合当前游戏状态。
- **运动层**。处理探路、碰撞和躲避等行为。
- **短期决策层**。AI实体短视距上的智能处理层，主要关注实体本身。
- **长期决策层**。AI实体开阔视距上的智能处理层，比如制定计划或者团队配合等问题。
- **基于位置的信息层**。包括来自于影响图、智能地形或者类似结构的信息。

23.2.1　重现前述示例

我们将通过逐步跟踪前述示例的形式来介绍系统的主要层次逻辑并在各层次中合理地分配系统任务(依照前面的惯例我们把其中的主要 AI 角色叫做 Brian)。

(1) Brian 被别人叫喊。假设游戏采用基于事件的系统，因此 Brian 将接到一个来自于叫它名字的游戏实体的消息。

(2) 假设该游戏采用分层有限状态机(FSM)(包含状态栈，使用方式同存储器)作为主要决策模型。此处提到的有限状态机同 15 章中提到的有限状态机。在该例的初始阶段 Brian 的状态是“DoingWork”，在该状态下等待消息“MyNameIsCalled”。在获得该消息后，Brian 的状态将发生转移并进入状态 Investigate，同时把状态“DoingWork”压入堆栈中。这是一个短期智能示例，中断行为 DoingWork 是由于个人感知的变化而不再执行，并且整个行为过程处于更高层次状态机的状态 Afternoon 中。

(3) 在状态 Investigate 中，状态机根据声音的来源计算调查目标的位置。整个计算逻辑完全包含在状态 Investigate 的行为中，因此属于行为层。

(4) Brian 使用位于导航层的探路者来产生到达目标位置的动作序列。

(5) 在确定移动方向和动作序列后，运动层被激活并开始移动。接下来 AI 系统首先要做的就是利用动画层来选择正确的动画。

(6) 我们开始播放移动动画后，另外一个感知器却提示我们将与某移动障碍物相撞(不包括和墙壁类似的固定障碍物，这些固定障碍物可以通过路径搜索算法避过)，所以动作层将调用动态躲避系统来避开这些移动障碍物。

(7) 当 Brian 打算离开时，其钱包发过来一个消息告诉 Brian：钱包落在桌上了(可以通过智能对象系统来实现钱包，假如 Brian 距离钱包一定距离后，钱包就会发送消息给 Brian，因为我们约定 Brian 不希望丢失钱包，而 Brian 和钱包之间距离的测量则是在更高层状态机的 Afternoon 状态完成的)，Brian 被逼暂时放弃当前目标而先关注钱包问题，此时长期决策层将进行一定检查并发现：钱包在办公室里，因此关门离开后它仍然是安全的(用一个简单计划算法来实现)。Brian 调用实现在行动层的关门动作，该动作再去播放位于动画层的关门动画。当状态被弹出堆栈后，Brian 将返回到上一状态。

(8) 感知系统在视觉上认出 Bob(一旦 Brian 到达一定距离之内，Bob 和 Brian 就相互看见)，同时发出在状态 Investigate 注册的消息“See Friend”了。处于状态 Investigate 的 Brian 收到该消息后，开始检测 Bob 的位置是否接近被调查目标，如果答案肯定，那么就确定是 Bob 在叫。该推理逻辑是行为层的一部分，也是状态 Investigate 触发的动作的一部分。

(9) 此时状态机从状态 Investigate 转移到新状态。该状态的动作，简单地说，就是播放处于动画层的 Brian 见到 Bob 时的动画。

相信大家都理解了其中的基本流程，接下来我们将讨论采用这些层次结构的理由，并针对每层给出其主要决策任务以及实现这些决策的技术，最后给出其他一些使用这些技术的示例。

23.2.2 感知和事件层

把感知计算独立出来的理由已经在第2章中给出。创建集中感知处理模块有利于优化AI计算。该模块的计算结果可以在同一游戏循环中多次使用，可避免冗余计算，并便于在游戏开发和调试过程中对某重要感知变量进行跟踪，因为该变量只在一个地方出现。

感知层的智能主要以各感知的反应时间和阈值的形式存在。所有的感知判断都从动作中分离出来，分离后的动作不需要关心感知的变化形式。因此，和某给定感知相关的所有行为，不管是激活还是状态转移，都能从内嵌于感知层智能中的感知更新方式中得到好处。

集中感知系统能在基于消息的系统中良好工作。当某感知触发(也就是说感知发生了变化)时，集中感知系统会发送和该感知相关的消息给特定玩家或者在整个系统中广播更加普适的事件。为了具体化感知的属性数据，在感知系统中需要包含一些AI计算。假如我挥动大剑砍向两个不同的游戏角色，其中一个角色的反应迟钝，它可能处于极度的惊讶中；而另一个(反应比较快速)就可能很容易地躲避我的砍杀，甚至反击我；角色的不同反应速度和行为取决于角色本身的设计。不同角色所做的反应是不同的，原因很简单，一个角色的感知系统感受到“大剑砍过来”这一动作，而另一个角色没有。

23.2.3 行为层

在多数游戏中，每个行为更应该看成角色的一个状态，然后通过一系列状态构造出更加复杂的动作。因此行为层一般定义了系统的状态或者游戏中将用到的原子动作。即使在角色始终处于某游戏状态(比如“Fight to the Death”)的格斗类游戏中，我们也应该为角色游戏结束时的动作设计行为层逻辑，尽管该逻辑仅完成动画的开始和停止操作。

不管游戏采用什么实现技术，行为层逻辑主要涉及状态机的状态转移以及各个状态下的行为。程序清单23-1给出了若干格斗类游戏动作的伪码，该例的这些动作需要大量动画序列来展示动作的效果。

程序清单23-1 格斗类游戏行为BigPunch的伪码

```
Begin BigPunch
    ForInit
    {
        DoSound(GRUNT_BIG)
        UseAnim(rand(NUM_BIG_PUNCHES)+ANIM_BIG_PUNCH_FIRST)
    }
    ForFrames
    {
        1 AllowCombo(off)
        2..5 TimeScale(1.6)
        6 OffenseCollisionSphere(1,on)
        7..9 SpawnParticle(FORCE_LINES)
        10 DoSound(AIR_SNAO)
        11..16 TimeScale(0.8)
        17 OffenseCollisionSphere(1,off)
```

```
        18 TimeScale(1.0)
        19..25 AllowCombo(on)
    }
End BigPunch
```

在该伪码中我们看到一些行为比如声音、动画、粒子效应和各种各样游戏标志开关的创建。由于这是一个格斗类游戏，每帧动画都有可能和某些代码或事件关联，因为格斗类游戏参数的调试和平衡需要非常高的精确度。程序清单 23-1 中的行为包括了多种类型的标志和事件，比如支持某些动作的暂停(为了支持组合动作的出现)、角色局部时间变形(便于调试动态效果的动画数据)、大声发笑时间、粒子效应和碰撞范围的切换(这样在动画描述时只需关注对手是否击中)。注意到该例假设所有格斗动画有相同数量的帧，在采用通用行为来尝试和平衡游戏性时，这点是非常重要的，但是很明显，如果我们愿意采用编写适用于任意帧数的动画代码，那么就没必要采用相同的帧。取而代之，系统会跟踪动画帧的数量以及脚本所能访问的最大帧数以决定每帧动画的停留时间。这样，每组动画的帧数是相对的，只要各“段”动画的持续时间相同即可，这样整个挥击动画的帧数就可以根据动画创作人员的喜欢任意确定。

在前面提到的示例中，当状态机的状态是 Investigate 时，AI 系统就会调用 Enter()方法，该方法根据 MyNameIsCalled 消息(消息中包含叫声的大约角度和音量)计算目标位置，并根据计算结果确定 Brian 将要调查的精确位置。与感知层一样，角色的属性将影响角色在行为层中的动作，比如两个不同角色在完成同一类型的行为 Jump()时会检查自己的属性来决定要跳多高。

对很多游戏来说，行为层是在程序代码中实现的，特别是对于 AI 角色的行为数量有限的游戏。但如果情况相反，数据驱动模式将是首选。如果能用脚本或者其他数据驱动游戏性方式来实现角色的行为代码(比如前面提到的格斗类游戏)，这将对系统的设计和实现很有好处，同时意味着游戏角色行为的设计可由设计人员完成，也减少了程序员的负担。该层包含的内容越丰富，角色展现出的个性与智能就越是计算无关，这样做并不是由于脚本不能包含计算过程或者计算速度慢，而是由于脚本主要用于保存感官类 AI 数据。通过设计人员提供的感官数据在脚本中的应用，游戏行为变得更真实。

23.2.4　动画层

NES 推出的第一款名为《超级玛莉》的游戏只用 8KB 的存储空间来保存图像数据，这些数据包括背景动画以及所有角色动画。随着游戏变得越来越复杂，给定游戏角色的动画数量飙升。某些画面质量很高的第三人称游戏仅仅主要角色的动画数据就达到 5MB，甚至更多。在很多游戏中，选择合适动画来播放也显得相当繁琐，尤其是那些画面质量很高的游戏，比如格斗类游戏和运动类游戏。格斗类游戏的角色可能具有数十种特殊的动作方式。运动类游戏依靠运动捕获的动画来模拟玩家行为，一个简单的动作可能需要 100 组或者更多动画连接而成(比如篮球游戏中的投篮动作、棒球游戏中的热身运动以及足球游戏的球门区庆祝动作等)。

随着游戏复杂度的提高，不仅动画数量增多，处理动画的游戏逻辑和计算复杂度也相应增加。某些游戏动画的组合计算是比较简单的，比如球门区庆祝动画，这些动画和游戏

性基本没有关系，唯一目的是提高游戏的视觉效果。而有些游戏动画需要较多的计算逻辑，比如足球游戏的接传球动画。处理该动画的代码需要考虑很多因素：球员的移动方向和速度、足球能经过的区域、传球的方向和角度、接球员的技能特点(比如速度慢跟不上球或者左脚无法接球等)和拿球后跑动方向等。在确定接球动画序列后，代码需要检测这些动画以确定球员将在哪一帧接触球或者确定球员是否要跟着球跑动一段时间后才能接住来球。在该例中，我们还需要考虑其他一些因素，比如球员需要为了避免和其他靠近的队员相撞而跃起，比如为了能追上球而拼命逃脱防守队员的围堵，比如在和其他球员碰撞后重新计算状态并试图重新拿球。在篮球类游戏中，几乎每个球员都是接球者，也都可能是防守者。在该类游戏的实现前没有明确处理动画的计算过程，那么整个AI系统将受到影响。

在实现包含很多动画资源或者繁重计算任务(这些计算任务可能包含很多需要调整和平衡的参数)的游戏时，我们应该首先考虑采用数据驱动模式。通常动画选择系统都是表格驱动的，比如图 23-1 给出的类似于数据库文件的表格文件，该文件用于确定篮球游戏的layup动画的选择。

动画名称	投篮类型	靠近角度	开始速度	球框
LayupUnder2ft	hard	efg	stand,walk	right
LayupBaseLtStand	norm	abc	stand,walk	front
LayupBaseRtStand	norm	fgh	stand,walk	front
LayupCtStand	norm	def	stand,walk	front
LayupCornerLtStand	norm	bcd	stand,walk	front
LayupCornerRtStand	norm	efg	stand,walk	front
LayupUnderStand	norm	efg	stand,walk	right
LayupBaseLtJump	norm	abc	all	left
LayupBaseRtJump	norm	fgh	all	right
LayupCtJump	norm	def	all	front
LayupCornerLtJump	flash	bcd	all	left
LayupCornerRtJump	flash	efg	all	right
LayupBaseLtRun	norm	abc	run	left
LayupBaseRtRun	norm	gh	run	right
LayupCtRun	norm	def	Run	front
LayupCornerLtRun	hard	bcd	Run	left
LayupCornerRtRun	hard	efg	Run	right
LayupUnder	norm	abc	Run	back
LayupRev	flash	abc	run	back
LayupRevTrick	flash	gh	run	back

图 23-1 篮球铺叠动画选择表

这张表格包含许多 layup 动画。每组动画都有一组参数，这些参数用于确定决策结构或者模式如何使用对应的动画。该模式涉及到4个参数：投篮类型(动作难度、技能率层次)、靠近角度(该参数可以是“a”到“h”的任意字母组合，“a”表示左边)、球员速度和上次

接球方向。从技术上讲，最后一个参数的数据类型可以和第二个参数一样，但由于该参数只需 4 种情况，因此为了提高游戏的执行速度我们没那么做。采用表格形式的第二个理由是为了节省游戏在动画选择上花费的计算时间(第一个理由是节省编程时间)。从技术上讲，可以把 layup 动画都播放一遍，然后根据算法确定所有参数，但是该过程过于浪费时间，所以采用表格来代替。而表格可以由算法来产生。动画表格工具系统可以通过输入的模式和动画集来产生该表格。我们可以在最终文件上进行任意的重写动作，这可以为设计人员减少很多工作量。

如今多动画通道技术应用普遍。双通道意味着角色的动画可以通过两组通道完成，上层通道负责角色的移动，而下层通道负责角色的其他动作，比如射击或者扔球。三通道或者更多的通道可便于角色身体各部分的分离控制或者其他辅助对象的控制(游戏角色不是真人，可以是“三头六臂”)。多动画通道技术会增加动画选择系统的复杂度，但可以通过嵌套表格的形式来优化系统实现。

和行为层类似，我们也可以利用脚本系统来实现动画层逻辑，因为动画层的逻辑和行为层的逻辑存在很多的相似性。我们可以在 Update()函数中逐帧控制动画效果和事件的加载。

23.2.5　运动层

正如在第 2 章中所阐述的，导航任务是 AI 游戏引擎设计的另一重点。路径搜索(主要形式是计划，有关方面我们在 22 章“其他技术备忘录”中已经提到)以及分支躲避是决定角色行为的主要因素。和其他层一样，运动层也是独立的，运动层逻辑会被 AI 引擎的其他部分复用(特别是需要地图移动的行为)，因此我们不能在行为逻辑中嵌入太多感知方面的逻辑。

运动层的实现技术已经在第 2 章中讨论过。由于该层与机器人学有关，有关路径搜索和障碍躲避的算法和资源可以很方便地从学术界获得。在互联网上随便搜索一下路径搜索就能获得成千上万条结果。

我们应该在该层嵌入什么逻辑和功能呢？答案是应该在该层试着建立角色的个性和属性模型。在赋予路径搜索系统和障碍躲避系统更多的基于属性的行为模式后，不同类型的角色将会有不同的探路和躲避模式，而不是采用固定的探路和躲避模式。在碰到障碍物时，体型较大的角色可能会绕过或者等障碍物先过去，而不像体型较小的个体一样直接闪过。不同类型的角色在选择路径时也可能不同。喜欢跳跃的角色可能会选择那些需要跨越沟渠的道路，具有翻墙能力的角色可能会选择直接翻墙而过，而聪明的角色可以找到靠近玩家的捷径。该系统的另一扩展是针对不同的游戏场景或者角色设计不同的障碍躲避手段。如果游戏角色的体型和超人一样，它会绕开垃圾？或者捡起垃圾并扔入外太空？或者疯了似得把垃圾踢掉？如果某角色可以跳跃 20 英尺，却选择绕开 6 英尺的箱子，这样的 AI 系统看起来非常愚蠢。

所有这些任务都可以封装在运动层中，系统其他部分不需关心具体实现。分层式 AI 的总体目标就是每层分管各自工作，而不影响其他层。从技术上说，障碍躲避系统的确有可能会使 AI 角色绕过能量提升宝物，但如果没有障碍躲避系统，AI 角色将一头撞墙而死，那就更没可能吃到该宝物了。

23.2.6 短期决策层(ST)

在 ST 层，决策所涉及的内容和特殊角色相关，这主要是由它们的属性、当前感知状态或者过去的经验决定的。一旦角色处于将死状态，它会和人类一样不顾一切地获得血瓶或者逃跑。它会利用武器的特殊性或者假如士气很低时拼命躲进树林里。我们把该层划分出来以支持具有其他个性或者属性的对象重用该层中的 AI 行为。在 ST 层中建立行为模型可能需要解决一些棘手的问题：在 ST 层中 AI 行为越多，角色移动时就越具个性。如果用 ST 层为大规模部队的队列建模，而三分之一的战士在行进过程中思念着自己的女友，整个队列将很不整齐。当然，ST 系统可以基于状态实现，强制某些关键时刻队伍的一致性，而在其他时刻角色的行为可以自由些。而事实上，大多 ST 决策系统确实是基于状态的系统，主要原因是基于状态的系统更加直接。

23.2.7 长期决策层(LT)

和 ST 决策相比，LT 决策不仅限于关注某个游戏角色，更多的是考虑大量的游戏角色，甚至有可能是游戏中的所有角色。LT 层考虑更多的是决策类型问题，考虑问题的计划、协调和时序关系。当然，并不是所有游戏需要考虑这些因素。但是就算是最简单的单人版游戏《铁手套》(Gauntlet)也包含一个基本 LT 判断：时间的流逝将带来角色健康值的降低。不管在 ST 层中为角色设定了什么样的近期目标，如果角色的健康值太低，其必然会抛开短期目标转而追求活命。而多人版《铁手套》需要更多的 LT 功能，因为团队合作会比单枪匹马能消灭更多的敌人。

和 ST 层一样，LT 系统的主要实现方式也是基于状态的。一个优秀的策略可分为在不同时期执行的子策略。象棋游戏的策略可分为开局策略、中盘策略以及残局策略；即时策略类游戏的决策也有发展期决策、一定数量的探索期决策以及最后的火并期决策。计划对 LT 系统来说非常重要——几乎定义了整个 LT 决策系统，在制定计划时需要更全面地观测问题，通过计划可以制定更全面的决策，包括对未来状态的前瞻。LT 系统有时也需借助于模糊技术、模糊状态系统或者完全模糊的逻辑系统。模糊状态系统可很好地用于即时策略类游戏中的多目标规划问题，比如进攻、防御、资源采集以及研究等目标的平衡和规划。完全模糊的逻辑系统可用于分析游戏前期侦查得到的有关战场的蛛丝马迹来生成赢得游戏的计划。

23.2.8 基于位置的信息层

很多时候，由于系统环境中存在太多对象，很难在给定时间内支持所有对象对周围环境的扫描以寻找感兴趣的对象并记录相关的各种感知数据，因此我们借助于表格实现相关问题。LBI 系统创建了有关游戏世界的集中去耦合的数据结构供各游戏角色访问。智能地形和对象向游戏对象广播其存在使得原本复杂的环境和对象之间的交互得以简化和优化。影响图支持有关游戏世界的丰富信息的存储、分类、计算和分析。LBI 层是一个相对独立的系统，我们可以把它看作创建 AI 引擎的大型黑板结构。LBI 层可以帮助 LT 决策层分析地形选招 AI 防御的弱点，识别地图上军事要塞的位置(那些包含大量的敌方足迹和尸体的地方)或者发现第一/三人称射击类游戏中的伏击点位置。LBI 层给出的小心附近敌人的提示

(可能通过声音的方式)可以改变 ST 层的决策。路径搜索系统可以利用影响图技术来避免角色进入地图中的死亡之地或者某些人为陷阱。

最后一种 LBI 层在系统中的使用方式是在游戏世界里直接嵌入触发器，这些触发器由某些游戏状态触发或者由邻近的触发器触发。这些触发器的操作可以是引起事件、发送消息或者执行游戏脚本，这些脚本将引起一系列事件的发生。通过触发器可使得相关区域具有“智能”，这可以简化 AI 系统的其他部分。触发器的布局主要通过某些游戏关编辑器完成，但也不排除使用其他系统，比如在游戏布局中或者位置相关的简单文本脚本中，尽管触发器在这些场合的应用方式不是很友好。

23.3　BROOKS 包容式体系结构

分层式 AI 结构在一定程度上类似于第 1 章提到的 Brooks 包容式体系结构。Brooks 的研究目标是对低智能怪物进行建模，并期望所设计的机器人有朝一日具有昆虫同等水平的鲁棒性，这里提到的鲁棒性并不是智能，而是在遇到危险因素时所表现出来的可靠性。而分层结构则更进一步，赋予了 AI 系统昆虫不具备的策略计划和智能。包容式体系结构的设计目标是智能体在数量和类型动态减损的情况下完成所制定的目标，而这正是我们 AI 引擎设计的目标之一。游戏系统和 AI 控制的角色不仅应该具有和人类玩家一样的处理能力，也应该具备故障自我恢复能力。

另外，Brooks 关于“机器人的智能应先达到昆虫的水平，然后继续提高”的理念在游戏界是非常不必要的。家蝇的智能水平只需达到能引起人类的注意就可以了。即时策略游戏中的蚁群只需具备发展、防御和偶尔的攻击等行为所必要的智能即可。我们设计 AI 的目的不是实现 AI 的最终目标，而是使游戏角色具备一定的智能。大部分人常常会为自己能在半小时内捆死在边上嗡嗡直叫的家蝇而自豪和欣慰，但却从来不会为自己没能杀死脑袋只有斑点那么大的动物而感到羞辱。家蝇通过惊人和有效的行为引起人类的注意并积聚能量，但最后却不是玩家的对手，这是完美 AI 游戏的深刻教训。

23.4　游戏层次分解

23.4.1　目标

本部分首先把一个非常盛行的商业游戏按照分层式 AI 思想分成若干层，并具体说明各层的实现方式。注意，本部分的目的不是描述如何实现各游戏层使得整个系统更加高效或者更有娱乐价值，而是展示如何从各个角度深入思考游戏 AI 问题，帮助读者真正理解不同 AI 状态下各种解决手段的意义。因此我们将着力于“最重要的”AI 技术，也就是说那些对玩家影响最大的 AI，我们将局限于游戏的灵魂部分，而不去考虑那些只为提高游戏效果而设计的动画和行为类型。

然后，我们将谈论如何为游戏创建 AI 控制角色。在该部分中，我们不会改变敌人的行为或者影响游戏性。我们的主要目的是创建能很好地完成游戏任务的 AI 系统，而不是

特别关心该角色的行为是不是和人类类似。该示例并不是为了说明其是实现 AI 玩家的最好方式，而是为了帮助设计人员拓展在给出有关给定输入输出条件下的领域视野。

23.4.2 分层式超级玛莉

从技术上讲，游戏 Pitfall 是最早采用卷轴游戏技术的游戏，但由于该游戏没有充分利用卷轴机制，因此通常认为《超级玛莉》是最先采用卷轴平台的游戏。《超级玛莉》是一款非常经典的游戏。该游戏首次实现了“奖命”(1-UP)机制，“奖命”机制通过玩家吃宝物的方式实现，另外玩家还可以通过吃某类宝物直接进入隐藏关。该游戏共有 32 组正常关和 20 组隐藏关。游戏总共包括 32KB 代码和 8KB 图像数据。这 32KB 代码部分并不全是代码，其中包含一些用于构造游戏关的数据信息。由于存储空间受限，几乎所有游戏元素在设计时都尽可能简化。在游戏实现时，通过帽子来消除游戏提供发型动画的需求，通过胡子来避免游戏提供嘴型动画的需求等，所有这些都是为了减少游戏的存储需求。

在后面描述中将涉及以下几种怪物，下面进行简单介绍：

- **正常怪物**。这些怪物可被玛莉踩扁，比如蘑菇(又称 Goomba)。
- **各种库巴(Koopa)**。这些怪物的头可以被玛莉踩缩进去，并被玛莉用来攻击其他敌人，但缩头后的库巴也会伤害玛莉自己，并且碰到墙壁后会反弹。后面几关还会出现长了翅膀的库巴，消灭这些库巴需要三发子弹，通过第一发子弹把它们打成普通库巴。
- **铁锤(Hammer Brother)**。是一种特殊的库巴。比普通的库巴高，而且成双成对出现。它们跳得跟玛莉差不多高，并以抛物线状向玛莉投掷成打的旋转铁锤。
- **Lakitu**。是一种只在某些游戏关出现的腾云驾雾的特殊的库巴。它们会主动和玛莉保持合适距离，并扔出一些踩不死的小库巴。Lakitu 是游戏中除了铁锤兄弟外智能水平最高的怪物。
- **库霸王**。每关的终极头目，是一只体型较大的库巴。在游戏中它跳上跳下，喷出大量火焰。偶尔它也会像铁锤兄弟一样扔掷铁锤。和其他怪物一样，它也可被玛莉的火球或者火山烧死，前者的条件是玛莉处于火力加强态，后者的条件是首先炸毁吊桥。

23.4.3 AI 怪物的实现

商业发行的《超级玛莉》中几乎没有采用任何 AI 技术。所有怪物都是通过算法控制的，只能按照既定模式移动和攻击。大部分怪物的行为是简单地朝前移动，碰到墙后朝反方向继续前进；也有一部分怪物对玛莉的位置敏感，比如 Lakitu、铁锤兄弟以及库霸王，但它们也只会针对玛莉的位置改变移动方向而已。总的说来，玛莉能预测到所有怪物的移动行为，因此，这些怪物在优秀的玩家眼里只不过是一些很好躲避的障碍，他们能很轻松地通关。

1. 感知和事件层

游戏中玛莉的行为模式是受限的。在移动方面，它只能前进、后退、跳跃以及奔跑。在那些“海基”游戏关中，玛莉的控制方式也是一样的，只不过跳跃意味着往上游，唯一

的区别是重力因素。如果在陆路或者阶梯上跳跃，重力因素将使玛莉很快就落回地面。而在水中移动时，如果没有按住跳跃键，那么玛莉很快就会落入水底。奔跑模式只在陆地或者阶梯上有效。玛莉的状态主要有 4 种：正常体型态、体型增大态、火力加强态以及不死态。

由于玛莉的行为模式有限，游戏感知系统很容易快速而全面地跟踪玛莉的行为。AI 系统可能用到的其他感知数据还包括：当前画面的左边界(由于在卷轴类游戏中，游戏以一个个卷轴场景的方式出现，卷轴不断往右卷，不能回卷，因此有必要记录左边界以确定能回退的范围)、怪物和玛莉之间的距离以及其他一些复杂数据结构，比如怪物当前所处的阶梯，通过该数据可以预测玛莉可能的来向和路径。

这些感知数据基本都是有关玛莉的，因此可以集中存储并供各 AI 怪物共享。

2. 行为层

玛莉的行为是受限的，而大多数怪物更加如此。这些怪物的行为非常简单：要么播放动画，要么移动。少数怪物还能投掷炸弹。

《超级玛莉》行为层的行为主要有 3 种：PlayAnimation(参数包括动画名、动画开始帧以及时间范围)、Move(参数包括位移量以及左右方向)和 SpawnProjectile(参数包括炸弹类型、速度以及轨迹类型：抛物状还是直线状)。由于行为层的函数数量少而且功能简单，我们没有理由不直接采用程序代码实现并应用于 FSM(游戏中 ST 层采用 FSM 实现)。我们可以通过继承来创建很多怪物的特殊动作，比如组合动画(食人草的动画就是上升动画和咀嚼动画的组合)、特殊类型的 Move(直线、绕圈以及反弹等)以及投弹攻击(弹的类型包括铁锤、火球以及子弹等)。这些继承行为可以通过代码直接实现或者通过简单的脚本系统实现指定行为的参数设置(第 18 章“脚本系统”中提到的配置脚本即可)。

3. 动画层

《超级玛莉》中的动画不需要使用现代动画选择例程。这些动画最多只有两帧，而怪物就在两帧所定义的位置之间来回移动。有些动画只有一帧并通过怪物映射来表示移动。因此我们很幸运，不需要在动画层花费大量时间。

4. 运动层

《超级玛莉》中怪物的行为都可以通过设置参数衍生得到，因此运动层的功能稀少。我们需要赋予怪物有限的路径搜索能力使得它们能从一个阶梯跳到另一个，并可以更好地攻击玛莉。同样，具有飞行能力的怪物也需要一定程度的路径搜索能力。游戏中的怪物基本都是局部化的，它们被限制于游戏的某个区域，因此路径网络也只需局部化，不需要整关互连。当 Lakitu 处于云层上时，具有特殊的路径网络，该网络就是云层移动的轨迹，一般为直线。由于系统所实现的移动都是离散化的，因此可利用势场技术使移动过程看起来更加光滑和有机。

游戏环境并不是完全静态的，玛莉可以撞碎能够得到的所有砖块。因此路径搜索数据是和砖块状态相关的，如果怪物希望这些数据来指导跳动，那么每次玛莉撞碎某些砖块后相应数据就必须更新。

5. ST 决策层

考虑到怪物所具备的行为和感知非常有限，各种怪物所需的决策也相当简单。以游戏中的花朵为例，从代码角度考虑，花朵从花盆中绽放的过程就是播放动画的过程。花朵绽放存在时刻表，只有当时刻表显示此时应该绽放并且玛莉没有站在花盆上时，该过程才会启动。假如我们放松某些怪物的时间差限制，那么游戏难度将变得非常高，这是由于原来的那些按时行动的怪物的动作无法预测，任何时刻玛莉都有可能因为某些怪物的意外出现而死亡。

这些 AI 怪物的动作集非常简单(通常只包含 Walk 和 TurnAround，最复杂的怪物也只具备移动、跳跃以及有限的投弹能力)，因此可以通过 FSM 实现 ST 决策层，而且大部分怪物只有两种状态，少数三种。

另一种描述这些动作的方法是脚本。程序清单 23-2 给出了一些该假想系统的简单类 C 风格的脚本。由于这些脚本非常简单，脚本编写工具甚至可以在稍微大型点的游戏关编辑器里实现。该工具将支持我们构造脚本，并把游戏中的怪物和脚本关联起来。程序清单中的最后一个脚本是前面脚本的升级版，该脚本使用了更多的感知数据来对可怜的玛莉构成更大的威胁。该脚本试图躲避跳过来的玛莉以及伤害值很高的火球，并试图避免怪物落入悬崖。同时，该脚本试图避免怪物扎堆以此来减少玛莉的容身空间。受该脚本控制的怪物是真正凶残的怪物。

程序清单 23-2　《超级玛莉》中怪物的行为脚本示例

```
//Simple Guy
Update:
   WalkForward;
End;

OnWallCollision:
   TurnAround;
End;
//--------------------
//Hammer Brothers
Update:
   If(MarioHeight > MyHeight)
      JumpUp;
   Elseif(MarioHeight<MyHeight)
      JumpDown;
   FaceMario;
   SpawnThrownHammer;
End;

//-Advanced Script--------------------
//Simple Guy
Update:
      If(MarioIsJumping)
      {
```

```
            //returns if Mario will land to my left or right
            dir = CalcMarioLandingSpot;
            //FindBestOffset finds the best spot right next to
            //where he'll land, so that he won't squish me on
            //the way down
            dir = FindBestOffset(dir);
            If(dir != MyDir)
               TurnAround;
            //make sure enemies don't pile up, spread them out
            //so its harder to land safely
            WalkForwardNoCrowding;
         }
         else
      {
         if(!FireBallNear)
         {
            FaceMario;
            If(!NearEdge)
               WalkForwardNoCrowding;
            Else
               TurnAround;
         }
         else
            DodgeFireBall;
      }
   End;
```

6. LT 决策层

按照定义，通常 LT 决策层负责处理长期决策问题。但由于《超级玛莉》游戏的特殊性，LT 决策层考虑更多的是 LT 系统的其他用法：协调怪物之间的行为实现更大的目标。《超级玛莉》的决策层的主要作用是协调各怪物之间的动作来堵玛莉的路。LT 系统将监测当前游戏画面中的能量提升宝物和隐藏块并派专职怪物把守，而其他怪物也将守在玛莉必须经过的位置上。LT 决策层可通过和 ST 决策层 FSM 并行工作的 FSM 来实现或者通过 ST 决策层的上层 FSM 来实现。从本质上讲，LT 决策层给每个怪物分配一定的任务，而每个怪物的 ST 决策层负责选择更好的工作序列来完成该任务。我们假设所有的怪物都是 LT 决策层的“马前卒”，只要某种条件满足，任何怪物都会完成 LT 分配的任务。这种假设在很多支持个性化角色的游戏中是不成立的，比如战斗类游戏以及格斗类游戏，如双龙(Double Dragon)。这些游戏中每个角色都有士气或者勇气值，如果该值太低，它们就不会听从 LT 决策层的吩咐并逃跑。但在《超级玛莉》中不是这样的，所有的怪物都是为了荣誉而活着，它们的唯一目标是阻止玛莉营救公主。

7. 基于位置的信息层

假如我们创建的 AI 系统可以多次学习，那么影响图技术就能跟踪玛莉在每关中的通行踪迹的统计量，该统计量记录了玛莉最喜欢并尝试获得的能量提升宝物，该信息可以改变专职把守的怪物的分布以最大程度地伤害玛莉。当然，这类信息是统计数据，因此需要

玛莉多次重复同一关游戏。该系统的最大优点是经过学习后适用于任何地图，因为该系统抽取了玛莉移动的策略系统而针对特定游戏关的行为，比如跳吃某个位置的能量提升宝物等具体信息。同样该系统会随时间自我调整，比如玛莉根据怪物位置调整通过方式后，这些怪物也会相应地调整位置。

23.4.4 AI玩家的实现

在对AI玩家进行游戏层次的划分中，我们假设怪物采用《超级玛莉》中的标准“AI”实现。本部分将设计AI控制的类似于玛莉的角色托尼，其具备和人类玩家控制的玛莉相同的能力。我们可以在游戏中同时实现电脑控制的角色和人类控制的角色，它们同时闯关，这样可以比试是电脑玩家还是人类玩家获得最高分和最多的能量提升宝物。

1. 感知和事件层

托尼的输入数据的总量是有限的。当它看到某些怪物时，感知层就把这些怪物加入到列表m_nearbyEnemies中，并跟踪其类型和位置，目的是预测移动轨迹。同时托尼还需要保存有关附近能量提升宝物的变量列表(包括已知却隐藏的)。

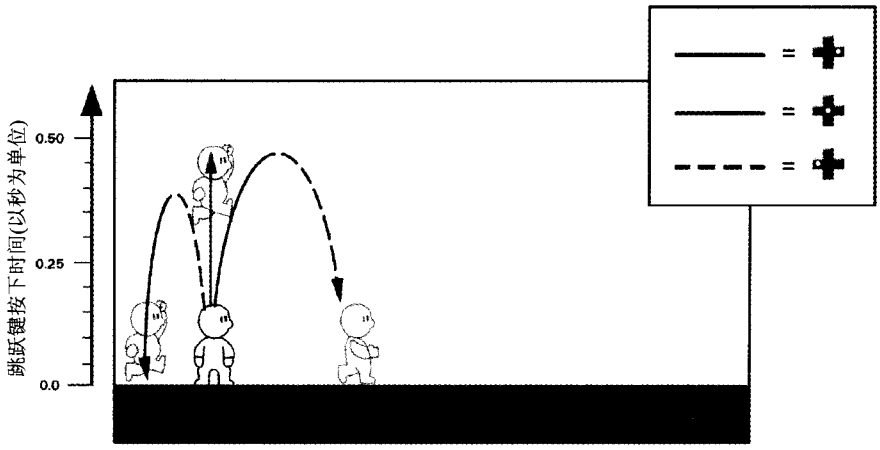
2. 行为层

托尼具备若干种行为能力。它能走、跑、跳跃以及射击(如果有子弹)。除了跳跃，所有其他行为可以通过简单的代码实现。在《超级玛莉》中，对玛莉来说游戏的精髓在于跳跃。《超级玛莉》是最先提出受控跳跃的游戏。玩家可以按住跳跃按钮来获得高度更大的跳跃，也可以通过按住方向键来获得角度更大的跳跃。这些简单的附加功能使得托尼AI的实现更加复杂。

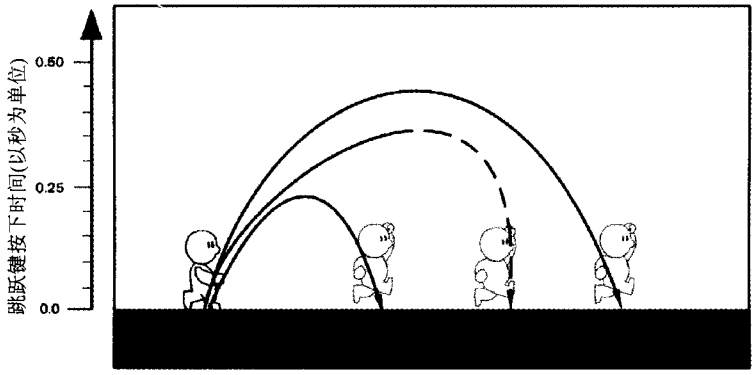
受控跳跃可以通过两种行为组合而成：空中移动和上跳。图23-2给出了受控跳跃的基本游戏性机制。当玩家按住跳跃按钮时，游戏系统通过计算按下按钮瞬间的冲量、跳跃的高度和根据方向键得到的跳跃方向(正值表示前进、负值表示后退)来生成跳跃角度。为了创建有效的AI托尼，我们需要使其具备该机制下的跳跃能力。

受控跳跃的实现方式可以参照任何复杂的游戏行为。我们需要确定和跳跃相关的操作杆输入，并根据不同的输入把跳跃划分成不同类型的跳跃，包括LongestJumpPossible、JumpToSingleBlockDirectlyAboveMe以及其他很多的跳跃类型。我们甚至可以针对不同的游戏区域为托尼的跳跃编写专用代码。该方法是可行的，因为游戏中托尼所经过的区域类型有限，我们只需要设计有限的专用代码。我们可以创建50(该数据没有特殊含义)种跳跃类型，然后创建针对这些跳跃的选择系统，选择的依据是当前位置和期望位置的向量差Δ。我们甚至可以通过训练神经网络或者遗传算法来获得选择规则。

进一步说，我们可以通过训练神经网络或者遗传算法来确定正确的操作杆输入以忽略特定的跳跃向量。该过程可能需要一段时间，因为我们使用了很多条规则(比如我们提到的50)，而每条规则可能需要10~30步。假设每次跳跃需要一秒钟，而每秒30帧，因此如果每条规则为30步，我们就可以完全控制整个跳跃过程。当然这些数据是可以减少的，我们可以通过实验来决定需要多少条跳跃规则以及每条规则需要多少步。



(a) 站立后跳



(b) 全速跑后跳

图 23-2　托尼的跳跃机制

3. 动画层

和游戏怪物一样，玛莉也不需要动画选择系统。玩家移动的最大变数是跳跃，不同的跳跃需要不同的动画类型。由人类控制的玛莉的跳跃是不可预测的。而 AI 托尼却不同，在其跳跃之前游戏可以知道跳跃的类型和幅度，因此可以为其绑定特定的跳跃动作使得游戏效果更酷。假如托尼有 20 种不同的跳跃动画，也就是每个方向各有 10 种类型的跳跃，此时可以通过非常简单的动画选择系统根据跳跃类型选择播放不同类型的动画，然后通过某种随机检测机制来混合所播放的动画，使动画看起来不属于某类型两组中的任何一组。动画选择的另一用处是状态转变动画的播放，当玩家突然停止前进时，系统能正确播放“刹车”动画，或者在连跳时，系统能正确播放跳跃组合。

4. 运动层

尽管游戏 2D 的特性使得路径搜索系统得以简化，我们依然需要为赋予托尼移动能力而进行路径搜索系统的设计。路径主要由不同阶梯之间的跳跃组成。我们甚至可以预处理必要的阶梯跳跃类型并编入路径节点以优化执行过程的计算代码。

路径搜索机制可以通过某类势场实现。该势场每段阶梯保存着跳入下段阶梯所需的力量向量。如果下一阶梯段是地面，那么相关的向量就是“行走”向量或是“跳跃”向量。同时具有“行走”和“跳跃”向量的阶梯段则是阶梯的边缘段，或者可以通过多种方式到达的阶梯段。为了吃到能量提升宝物或进入更好的阶梯，需要在势场图上回溯搜索(从目标位置到当前位置的搜索)以获得到达目标的移动向量。

障碍躲避机制也有很多实现方式，取决于游戏关中托尼的目标。如果托尼不介意跳过这些怪物，那么这些怪物就是单纯的障碍物，托尼可以选择其他方式通过画面而不杀死它们。如果托尼比较嗜血，那么它会注意到所有的怪物，如果该怪物是可杀的，那么他就会选择跳跃运动来踩死它或者选择射击行为射死它。

如果采用势场方法实现托尼，那么怪物就会发出负势场以便于托尼在怪物靠近时自动跳跃的实现。一般托尼在靠近怪物之前就跳跃，因此势场必须考虑托尼跳跃点和怪物位置之间的偏移量。采用这种工作方式的运动层不需要关心怪物所处的水平，因为其只影响托尼在阶梯之间的跳跃动作。尽管如此，我们必须非常小心地避免托尼自动跳过怪物并落入洞穴而死。

5. ST 决策层

ST 决策层只处理托尼当前所处的画面。获得能量提升宝物、为了最小化危险而按序踩杀怪物、朝下一游戏画面行走等都将在 ST 决策层处理。简单的 FSM(只需包含少量状态，比如 EliminateCreatures 和 GetPowerups)即可完成 ST 层的任务，特别是当我们把大量的智能逻辑放在其他层中。

6. LT 决策层

LT 决策包括时间性决策(比如游戏结束时间的限制迫使托尼选择捷径进入下一关)，或者为了获得更多的金币，或者增加生命数而选择特殊的路径。托尼经常需要撞击某些位置特殊的砖块来获得墙后的宝物，因此它必须在这方面具有特殊能力。在《超级玛莉》中有这么一个场景：玛丽首先撞掉某些砖块，然后通过撞出的洞进入上一层阶梯，并在此基础上撞掉再上一层阶梯的两三块砖块，然后跳过陷阱进入另一边，碰出爬藤并顺着该藤进入 warp zone。进入 warp zone 这个简单任务需要上述的计划过程。尽管该过程可以通过计算在游戏中动态生成，但这需要消耗相当多的计算资源。因此，由于我们没有通过随机方式构造游戏关，也就是说什么位置会出现什么问题事先是知道的，我们可以通过预计算的方式来进入隐藏点，通过脚本来确定进入该隐藏点最先需要撞掉的砖块的位置以及后继的动作。

7. 基于位置的信息层

很多时候托尼向前行进时没有看到怪物，因此它选择跳入后继阶梯，由于该跳跃带动卷轴向前滚动使得它碰到了原本看不到的怪物，就这样托尼被杀了。假如我们在游戏各关中实现了所有可以容身的空间的影响图，使用占用数据来存储某段时间内所有位置上的怪物数量。这样，我们就可以通过该占用数据来记录这段时间内怪物是否在指定的位置出现过，而不管这些怪物当前是依然处于该位置还是处于慢慢离开状态。在采用了这类系统后，托尼在跳跃之间可以检测是否存在这样的怪物：其近段时间内曾经在该阶梯中出现过而此时却回到画面的右边。托尼甚至可以通过观测若干帧内该数值的衰变情况来判断怪物的移动方向和速度。类似的系统能避免移动怪物给托尼造成意外死亡，却不能避免一次性怪物给托尼带来的意外死亡，因为它们直到在画面中出现后才会移动。

智能地形系统肯定也能给托尼带来益处。通过智能地形系统，托尼可以知道能量提升宝物的位置，甚至知道获得这些宝物的方式。不同层次的单元(比如花朵管道、大跳垫等)也可通过智能地形系统提供给托尼，因此它可以“闭着眼睛”使用它们。

23.5　小结

分层 AI 设计是一种 AI 任务划分方式，它不仅支持游戏功能的划分，也可以作为独立平台为 AI 控制角色增加个性。它在保证游戏代码重用性的同时丰富了游戏行为。

- 分层 AI 设计支持不同 AI 问题的可控划分。
- 分层 AI 设计方法的主要层次包括感知层、行为层、动画层、运动层、ST 决策层、LT 决策层以及基于位置的信息层。
- 分层设计对 Brook 包容式体系结构的改进包括高级决策层和处理多 AI 实体的协同层。
- 本章详细剖析了《超级玛莉》，改进和提供了 AI 怪物的智能水平，创建了 AI 控制的角色托尼。
- 本章给出了很多不同的分层技术，并不是说所有的经典游戏划分技术都可同时应用甚至兼容，只是要让读者知道存在多种《超级玛莉》中 AI 怪物和 AI 玩家的分层实现方式。

24 AI 开发中普遍关心的问题

本章将主要阐述在 AI 引擎开发中人们普遍关心的问题。这些问题包括设计方面的、娱乐方面的以及产品方面的问题，本章旨在为读者提供对 AI 开发中出现的问题的分析，这些问题可能会影响 AI 引擎的实现和 AI 控制实体的实现。

24.1 有关设计的问题

本书将深入研究 AI 引擎设计和实现中应该考虑的问题。这些问题不仅涉及到游戏 AI 系统设计的方向，也包括了系统具体实现的细节。

- **数据驱动系统设计时需考虑的问题**。在设计数据和逻辑分离的引擎时需要注意的地方。
- **“一根筋”(One-Track Mind，OTM)综合症**。这类毛病是指 AI 引擎设计人员在确定采用某单一 AI 技术后固执地将其应用于解决所有 AI 问题。
- **多细节层次(一种 3D 动画技术，Level of Detail，LOD)AI**。一些对优化多细节层次技术有益的 AI 引擎实现思想。
- **支持 AI**。在游戏工程中设计人员可能忽略的 AI 技术。
- **通用 AI 设计思想**。在 AI 引擎设计中可以尝试的通用思想。
- **通用 AI 实现思想**。在 AI 系统开发和实现时应该时刻铭记的观念和原则。

24.1.1 数据驱动系统设计时需考虑的问题

随着游戏内容层次的不断提高，必须在 AI 系统设计和实现上投入越来越多的人手。设计人员需要在 AI 系统中定义越来越多的内容，因此游戏 AI 逻辑的大小和复杂度不断增加。通常游戏 AI 行为和游戏代码存在着紧耦合关系，游戏代码需要访问动画系统、处理游戏声效、连接通信系统(比如消息系统)，因此非专业人士很难在系统中加入新的功能。正因为 AI 系统和游戏代码之间的紧耦合关系，在数据驱动系统的设计过程中也会涉及到 AI 系统的逻辑。在脚本 AI 引擎中，角色的基本行为是通过关键字和函数的形式提供给程

序员的。在采用可视化系统设计状态机或者其他结构时，设计人员通过互连不同的行为节点(这些节点代表不同的游戏系统)来构成整个游戏行为。

一种游戏数据驱动实现层次的选择办法是自顶而下地尝试：首先尝试在系统的最高层次实现数据驱动，然后监测数据驱动系统的利用情况，如果设计人员总是要求程序员在下层逻辑中加入数据驱动单元，那么我们就应该在该层实现数据驱动系统，这样根据需求不停地迭代，最后得到数据驱动合适的实现层次。这种思想来源于很多现代 3D 游戏的数据驱动图形生成器。

3D 游戏生成器是一种通过大量可重用的底层图元进行图形生成的软件系统。这些图元可以来源于不同的图层：最底层的是多边形，通过多边形的组合构成模型，然后通过模型的组合构成画面。在各层次的生成器管线里，所有的对象都通过下层对象来构造，同层的对象之间相互独立。在采用数据驱动进行图形生成时，“全硬代码”到“全数据驱动”的转变过程也是从系统的最顶层到最低层的过程。一些早期的 3D 生成器就采用这样的脚本。软件 POV-Ray(一款最早出现于 20 世纪 80 年代的画图引擎)就是最好的例子。POV-Ray 中包括很多图元(超环面、球面以及其他一些可以通过数学等式表示的图形)。通过 POV-Ray 来生成图形的主要工作是编写并执行画面脚本。年轻的图形程序员首先需要尝试生成些小示例来学习 POV-Ray，然后才能更好地利用其生成游戏画面。但是我们不能通过 POV-Ray 来定义模型，因为其只支持数学等式。一些图形，比如 Crash Bandicoot，就很难通过数学模型表示，因此我们需要使用更低层次的生成器，比如 mesh 生成器。在复杂图形的建模过程中将用到一组和 POV-Ray 不同的图元(多边形图元，包括三角形和四边形等)，在这些图元的基础上通过 mesh 生成器而非数学生成器来完成图形的生成。这是当前 3D 建模的主流技术。而三角形等多边形本身不需要使用数据驱动的生成器来生成，因为它们的层次已经足够低，不需要进一步地抽象了。

在游戏 AI 中，和图元作用相对应的基本单元是动画，通过动画的组合来构成角色的行为，然后通过角色行为来构成角色策略。根据前面所述的自顶向下的步骤，设计数据驱动系统的第一步是在游戏策略层上完成系统策略的编码，从而完成在策略层上的数据驱动系统的设计。当前几乎所有游戏的数据驱动系统都是在游戏策略层上完成的。我们通过使用脚本(通常选择)或者表格来构造系统，在该系统的基础上为玩家属性和行为以及场景过渡决策等进行游戏规则的建立、状态机的设计以及模糊参数的设置。

如果发现设计人员不断地要求加入新行为，那么我们就需要为设计人员提供定义行为的技术支持。这需要我们给出能衍生出所有行为的基本单元集，然后由设计人员通过基本单元集完成行为的建模。通常这些基本单元包括客观世界的变化过程(移动)、某些物理过程(动画)和游戏事件(信息的发送、声音的加载或者图像效果)。如果我们在动画中通过使数据变换来实现角色的移动，而不是与 Legend of Zelda 以及 Quake 一样通过动画背景下的角色滑动来实现角色移动，那么该移动也属于动画范畴。甚至我们确实需要通过滑行来实现角色移动，其滑行速度也可以内建于背景动画。因此，我们可仅仅通过动画和事件(省去了移动)来表示游戏策略等。我们可以定义在不同的动画间进行切换的状态机，偶尔通过脚本或者表格触发一个游戏事件来建立 AI 的策略层。

如果发现上述支持还不够，很多游戏，比如射击类游戏，需要设计人员不停地对动画进行修改，那么我们就需要为设计人员自己设计动画提供支持，通常我们不会这么做，因为动画的制作涉及到很多美工问题。动画的数据驱动需求导致了更低层基本单元的出现，我们称这些基本单元为动画的“帧”。帧意味着动画过程某微小时间内角色物理位置和运动的瞬态或者快照。射击类游戏设计人员可以通过参数和事件选择特定的帧，也可以通过混合播放不同动画的帧序列来完成动画的混合。

数据驱动设计也不是万能的，它不可能改进所有的游戏和系统。随着数据驱动层次的降低，组织层次需要上升。定义所有可能性需要的数据大小随着数据驱动层次的下降而增大，数据将急剧膨胀，伴随着也将出现其他的问题。对于更适合采用代码直接解决的这类问题，我们应该避免采用数据驱动。当动画系统完全是数据驱动的，但所有的动画除了某个脚本外基本使用相同的脚本时，该数据驱动系统完全是种浪费，不管从性能还是从数据大小角度考虑我们都应该采用代码直接解决该问题。

我们仅支持在必要时采用数据驱动设计，而且使用时要注意遵循以下原则：通过创建尽可能简单且可重用的基本单元来生成复杂对象。如果创建这些基本单元有问题，可以选择更低和更复杂的层次，也许数据驱动系统就能更好地工作。

24.1.2　“一根筋”(OTM)综合症

AI 程序员经常犯的错误是想到一种专用技术后就试图将其应用于他们所遇到的所有问题，而不管该技术和所遇到的问题是否相关。AI 技术的主要问题之一是，它们对内容非常敏感，只能解决特定类型的问题，只针对特定场合的输入和游戏条件。就算中等复杂度的游戏，也不存在这样的 AI 技术，使我们可以清晰地、可伸缩地以及容易地解决所有的问题。

AI 程序员，特别是刚入门的 AI 程序员很容易掉入“状态机是万能的”这样的陷阱里，主要的原因是大多数人发现理解状态机很容易而且他们也喜欢状态机划分问题的方式。然后，他们就迷失了方向，成了状态机的铁杆拥护者。FSM 在游戏界已经存在很长一段时间了。我们确实可以通过状态机解决 80%的问题，特别是对于那些不介意每周工作 100 小时分析包含在大量状态转移语句和定义自由的优先级系统中的让人看不懂的状态机逻辑的程序员，他们又一次忘记了状态机的扩展性不是很好，加入越多的状态，状态机的维护就越困难。FSM 就像是铁锤，没有铁锤没法建房子，但是建房子不能仅靠铁锤。

AI 引擎在设计时不应该只采用某技术。我们可以通过 FSM 来实现游戏中的基本状态的转移(比如游戏片头动画、游戏介绍、游戏过程和游戏结束等状态的转移)，通过 FuSM 来实现主要的短期决策层，通过简单的 planner(一种人工智能语言)来实现路径搜索和长期决策层，通过脚本系统来实现数据驱动动画的选择以及通过配置脚本系统来实现决策层。游戏的感知系统可以活跃于后台，并不停地给其他的所有系统传递消息。听起来很复杂吧？但如果把整个系统划分成一个个模块，那么我们所需创建的就是简单的、可扩展的、模块化的系统。我们可以通过模块划分和有针对性的技术来解决所有 AI 游戏设计中遇到的问题，包括游戏开发最后阶段因集中测试而出现的游戏创意和调试危机问题。当遇到某些完全没有预计到的问题或者某些不可解决的问题时，该系统也足够灵活，可以在不破坏其他部分的前提下引入新的模块，而那些大型的完全基于 FSM 或规则的系统是没法做到的。

24.1.3　多细节层次(LOD)AI

AI 系统通常都受限于资源。查阅历年游戏开发者大会的论文，我们发现 AI 所需的 CPU 资源占整个游戏 CPU 资源的比重逐渐增加，从开始 2%左右到 20 世纪 90 年代中期的 10%左右。当然有些游戏，比如 turn-based game，会明显低于该数值，但这里提到的是各种游戏的平均值。

和图形系统类似，减少 AI 系统对处理器资源的需求的一种有效方法就是使用 LOD AI 技术。不同的细节层次主要取决于 AI 角色和玩家的距离。下面给出几种典型的细节层次：

- **非游戏画面中且距离很远层**。处于该层的 AI 角色对于玩家来说完全是不存在的。
- **非游戏画面中但距离不远层**。玩家虽然看不到处于该层的 AI 角色，但却能通过它们经过时的关门声等方式来判断它们的位置。很多游戏不采用这些判断依据，但它们可以感知到是否有 AI 角色正在靠近。
- **甚远距离层**。处于该层的 AI 角色可表示成一到两个像素。
- **远距离层**。处于该层的 AI 角色的颜色、外形可能会被玩家看到，但它的具体细节却不能被玩家看到。如果该 AI 角色是生物，玩家能分辨出它是人还是怪兽，却不能分辨出它是哪个人或者哪个怪兽；如果该 AI 角色是汽车，玩家能分辨出它是卡车还是小汽车，却不能分辨出它的牌子。
- **中等距离层**。处于该层的 AI 角色处在玩家的有效视野范围内，它的清晰程度更多地取决于游戏视角以及其他一些非距离因素，比如周围是否存在浓雾以及浓雾的位置、浓度等。该层大概距离玩家 40 到 70 码。
- **近距离层**。处于中等距离层和交互层之间的 LOD 称为近距离层。
- **交互层**。处于该距离的 AI 角色通常正以某种形式和玩家进行交互。

当 AI 角色的位置在不同的层之间变化时，我们可以使用很多处理方案，其中之一就是针对不同的细节层次执行不同的 AI 例程。如果游戏是通过脚本系统的方式实现 AI，那么就可以在 AI 引擎中内嵌 LOD 检测功能，然后针对不同的 LOD 执行不同的脚本或者脚本中的不同逻辑。当周围存在人类时，一般 AI 角色需要执行动态障碍躲避逻辑，而当它不处于游戏画面中或者处于甚远距离层时则不需要执行该逻辑，前提是当人类玩家回来并靠近时 AI 玩家能保证自己不会进入奇怪位置(比如和人类玩家重叠的位置)。在实现时 AI 系统可以不管角色所处的细节层次，只需简单调用 DoAvoidance()函数，在该函数逻辑中通过查询 LOD 系统决定是否真正发起躲避操作。该方案所带来的问题和图形系统类似：程序员需要编写不同版本的脚本、代码或者数据库。总之，为了在系统中引入 LOD 技术，我们需要花费数倍的实现代价和调试代价。

另外一种 LOD 处理方案涉及到 AI 引擎的更新频率，也就是说在不同的层次中 Update()函数的调用频率是不同的。当 AI 角色处于中间距离层时，AI 决策的更新频率非常快，可以达到每秒 10~30 次。而对于不在游戏画面中的角色，更新频率可能降到每秒 2~5 次，甚至更少。如果设计人员觉得必要，一些非必需行为(所谓的“橱窗装饰”，因为这些内容除了增加画面的视觉效果，没有其他任何作用)完全可以不必更新。在制定 Update()函数中各角色的更新时刻表时，我们应该考虑所有时刻 Update()函数的负载平衡问题。假如存在大量某种类型的角色，它们都处于 LOD 技术的第 3 层次，并且更新周期都为 15 个游戏循环。

在保证更新周期不变的前提下，错开这些角色的更新时机，每个游戏循环更新若干个角色，在 15 个游戏循环中完成所有该类型角色的更新，这样我们就平衡了 Update()函数在每个循环的负载。上述过程当然是有条件的：各角色的创建时间是不同的，只有当创建时间不同时我们才能在不改变角色更新周期的前提下平衡 Update()的负载。

下面通过示例具体阐述 LOD 技术的含义，该示例是某 3D 第一/第三人称射击类游戏中的 NPC 角色。NPC 是个科学家，其在脚本的控制下在 3 个不同的工作站之间穿梭。当 NPC 所处的工作站不同时，AI 系统会给出不同的动画以表示不同的工作。该 NPC 对游戏的操作没有影响，而仅仅影响游戏的感官效果，细节层次对其行为的影响如下。

(1) 不在游戏画面层

当 NPC 不在游戏画面时，他的 AI 系统不执行，因为他根本不起任何作用。如果他逐渐靠近玩家并且实验室门的隔音效果不是很好，那么玩家能偶尔听到 NPC 发出的声音。

(2) 甚远距离层

当 NPC 处于该层时，我们只需将他处理成静止不动状态。NPC 和玩家之间的视距非常大，其移动的效果在游戏画面中的表现仅仅是几个像素的移动，为降低实现代价，我们可以忽略其移动。

(3) 远距离层

当 NPC 处于该层时，我们只需使其偶尔在设备之间直线滑动(没有动画、路径搜索和躲避，仅仅是来回滑动)。处于此层的 NPC 对游戏画面有一定的影响，我们需要表现出来。如果 NPC 不是科学家，而是杂技演员，并假设他在表演侧手翻和空翻，那么其表现出来的行动就不仅仅是滑动了。因此，我们得使用不同的技术来处理此时的 NPC。

(4) 中等距离层

当 NPC 处于该层时，其常规 AI 系统将开启。因为 NPC 非常特殊，它的 AI 系统不是很耗计算时间。NPC 的设计给了我们一定的启发：在设计非必要游戏角色时尽量不要分配高级 AI 任务，比如 NPC 不包含任何感知系统以及路径搜索系统。

再举个例子，比如即时战略游戏中的敌对文明。该类游戏有一些不同：玩家不限于某个位置，可以自由发展，而且人类玩家不能通过观察了解整个游戏的进展态势，这是因为这些游戏，比如《战争之雾》，在设计时强制任何时刻玩家只能看到部分游戏世界的情况。该类游戏的 LOD 划分主要包括以下内容。

(1) 非游戏画面中且距离很远层

策略 AI 的优化方式主要有两种：一种是在任何情况下策略 AI 都持续运行，但不同 LOD 的更新频率不同；而在另一种优化方式下，非游戏画面中且距离很远层的策略 AI 的更新是受限的，也就是说一般不更新其策略 AI，只有在发生某主要事件(比如 AI 进入当前科技树的后继主要级别)后，策略 AI 才会在短暂时间内完成更新。战术决策和行动的优化方式比策略 AI 简单得多。值得注意的是，游戏《战争之雾》中迷雾下的角色的 LOD 处理方式和非画面中的角色一样。

该层角色的移动可以选择两种不同的方式：一种是以预定速度匀速移动，另一种是时间节点上的瞬移。该层游戏单元之间的碰撞也可以简化，甚至忽略，但我们必须确保这些单元出现在画面时不会重叠在一起。

该层角色的很多行动，比如战斗、修建筑或者采矿都可以通过统计模型完成，这样游戏就不需要在后台模拟执行非画面中的战斗场景，也不需要在后台模拟执行非画面中玩家的采矿动作，而通过定时器定时更新玩家的资源数。

(2) 非游戏画面中且距离很近层

由于“近”意味着可能马上就会碰到，因此该 LOD 中 AI 系统除了有关画面单元外的所有单元应该近乎完全工作。不处于迷雾下的单元战斗时应该发出声音。而动画选择显然可以被忽略。

(3) 甚远距离层至近距离层

对于 AI 系统来说，这些 LOD 几乎是一样的。即时策略类游戏的画面内容比其他任何游戏更加局部化，而且游戏的视角也很狭窄(Myst 使用更加通用的系统)。基于这两个因素，我们不需要了解全貌，只需和其他游戏一样把这些 LOD 上的角色看作是斑点。

(4) 交互层

处于该层的 AI 系统完全开启并且更新频率也处于最高水平以更好地支持智能决策。

24.1.4　支持 AI

有时我们需要在 AI 引擎中加入和游戏的主要部分没多大关系的内容。游戏的其他部分可从这些 AI 技术中获得好处。这些 AI 辅助系统包括：

- **用户接口**。游戏可能包含智能库存系统，该库存系统在安排对象分布时应该最大化空间或者最大化关键对象的存取效率。我们可能会通过鼠标移动或点击来完成游戏的操作(每种规则移动对应一个函数并可能引发一系列操作)。这些系统可以很容易地通过神经网络的离线训练完成。在文明类游戏中，用户接口的另外一个主要用途是“顾问”，和 AI 控制的对手类似，该“顾问”内建专用 AI 分析单元，监测玩家的游戏过程，当玩家在游戏性上面临选择时，给出人性化的建议，比如研究或者外交等选择建议。
- **调试游戏参数**。当 AI 系统包含大量参数时，我们应该自问一下：我们是否可以编写程序完成该系统的自动复制或者至少不断地执行？我们能否通过某种标准化的变形来完成参数好坏的衡量？我们没法确定的参数和错误的参数设置之间是否存在关系？如果我们的回答都是肯定的，那么我们的系统可以通过 AI 方法来调试。遗传算法非常适用于参数调试，特别是这些变量状态可很容易地变换到某种遗传表示。但由于很多参数和游戏性的设置相关，我们不得不按照游戏性设置的不同划分游戏状态，在不同的游戏状态下通过遗传算法进行参数调试，或者在参数调试时少量参数通过遗传算法而其他的参数则通过手工的方式调试。这些参数调试方法可应用于物理仿真或者死亡模式机器人的状态机转移参数的调试。
- **自动化测试**。随着游戏复杂度的增加，通过覆盖测试所有因素的可能性组合来发现软件中潜在的错误变得越来越困难。我们可以通过一种其他软件方向用了很多年的测试方法来测试游戏软件，该测试方法基于自动化测试系统。在游戏设计的同时完成自动化测试辅助模块的设计，我们就可以通过自动化工具完成部分游戏错误的调试，使得我们有更多时间进行游戏性参数的调试。自动化测试软件的应用前提是待测软件的输入输出系统是通用、开放以及“可欺骗”的(“可欺骗”的

意思是待测软件可以接收其他程序的输出作为输入来模拟人类的输入)。游戏软件具备上述特征，因为游戏软件的人机接口非常简单，主要是键盘、手柄以及鼠标，我们能很容易地模拟这些设备的输出并输入到游戏软件里。当前业界存在若干种测试类型。边界测试主要测试系统在输入数据处于系统处理能力的边界时是否会出现错误。随机测试则通过输入完全随机的数据来测试系统。智能测试通过模拟真人游戏行为来测试游戏软件，一般会在关键点结合其他两种测试手段。因此智能测试手段非常适合于赛车类游戏的测试，但在采用智能测试对赛车类游戏进行测试的过程中也需在关键时刻结合其他的测试手段，比如当赛车处在桥面上并且周围存在其他赛车时，测试系统可能会随机发送赛车操作命令来检测赛车模型在不同的操控和碰撞条件下的鲁棒性。在采用自动化测试系统时，我们需要考虑是采用模块测试还是采用集成测试。模块测试用于完成某些功能模块的测试，在模块实例化之前完成除错操作，避免影响其他模块。所有这些测试系统通过其他 AI 技术来实现。任何技术都可用于实现测试系统，包括遗传算法、部分随机测试等，甚至可以是 AI 控制的对手(前提是该 AI 控制的对手能输出控制数据与游戏的其他部分进行交互)。通过这些技术，我们可以测试游戏软件中存在于输入边界上或者某些随机时刻的潜在错误。

24.1.5　通用 AI 设计思想

设计 AI 引擎时，设计人员就好比站在海边(只不过这里的海包容的是无数的可能性)，或者设计人员就好比处于开发图腾柱的次高端的附属物。AI 系统需要几乎所有其他系统向其提供钩子来完成理性决策并使它们执行得更快。在处理包含大量功能集的 AI 系统时，我们需要考虑以下几点：

- 设计过程应该提倡头脑风暴。一旦问题摆出来或者明确了 AI 引擎的设计要求后，我们应花尽可能多的时间尝试所有能想到的办法来解决这些问题。当然，我们不应该只是想想，而应该在思考后将其记录下来。头脑风暴意味着让思维的火花尽可能地碰撞，激发大脑活力，并提出大量可能方案。稀奇古怪的想法并非毫无作用或者说浪费时间。因为很多时候那些愚蠢的想法是完美想法的基础。
- 经过头脑风暴得到一些想法后，应该和其他 AI 同事一起逐个谈论这些想法。我们不应该放过任何“愚蠢”的想法，而应该把所有的想法提到桌面上谈论并分类。仔细分析那些“愚蠢”的想法以确定这些想法是真正愚蠢的想法还是“深藏金蛋”的想法。在设计过程中跳出任务(比如 AI 引擎设计)的约束，思考其他问题可以帮助我们发现“最优方案”中可能存在的漏洞和模糊点，也只有多想想和任务无关的问题才使大脑不至于陷入思维定势，给出漏洞百出的方案。
- 如果有时间，可以尝试在实验室提供的环境中实现该 AI 问题的小规模原型系统，比如在本书中给出的 AIsteroids 测试平台中实现原型系统。这样我们就能在几小时或者几天的时间内确定采用何种高级技术并完成其实现并提交给测试人员，而不至于花费数以周计的时间来确定该技术是否适用。另外通常我们会在实现过程发现设计时期没有考虑到的问题。我们不必为此而感到自己是个失败的设计师。AI 问题中包含太多的变数，我们不必期望能预测所有可能性。我们可以在设计阶段

给出问题 80%的解决方案，并对代码进行深入研究后在原型实现阶段给出剩余的20%。在一般的工作模式下，我们将花费数月时间反复设计和修改，试图发现尽可能多的问题(可能会另外发现 10%)，但在编写代码的时候我们依然会发现设计上的漏洞(剩余的 10%)，因此也依然需要花费时间来分析和改进方案。对比该工作模式和一般的工作模式，我们发现其可以帮助节省大量时间和精力。

- 最后我们应尽可能地开放。不要以为其他人的想法都和我们自己的想法对着来。和人沟通时，特别当对方是程序员时，我们应该多想想模糊逻辑课上学到的东西。自己对并不意味着别人是错的，反之亦然。我们可能对了 50%，别人也可能对了50%，总的说来两人都是对的。在和人交流的过程中，我们应该允许理性处于模糊状态。我们不应该通过语义纠缠来证明自己是对的，而应该综合各方观点得到更好的解决方案。

24.2 有关娱乐的问题

和其他软件不同，游戏软件有两大目标：实现承诺的功能和给玩家提供娱乐体验。这两个目标虽然不冲突，但也很难同时满足。因此游戏的设计过程是两大目标折中的过程。这部分内容包括以下几点：

- **趣味性因素**。娱乐性因素是指在游戏 AI 设计中需要考虑的趣味方面的因素。
- **随机感**。随机感是有关游戏随机行为的问题，该问题对游戏性有特别大的影响。
- **游戏难度**。游戏难度的设置的确是个娱乐问题，但其中也牵扯到游戏设计和 AI 设计的问题。
- **一些使 AI 系统看起来很愚蠢的问题**。

24.2.1 所有重要的趣味性因素

当不同的人问别人对某个新游戏的评价时，他想得到的答案是不同的：对于 AI 程序员来说，他想知道的是别人对 AI 智能水平的评价；而对于迟钝的玩家来说，他想知道的是和同类游戏相比其游戏性(主要是指操控性)怎么样。

而当典型的游戏发烧友问别人对某个新游戏的评价时，他想得到的答案和游戏的趣味性相关，比如：相当漂亮，该游戏通过大量超空间纹理渲染构成猴子模型！但再问一下，真的有趣吗？

那么何种因素才使游戏变得有趣？心理学家有很多理论。一种理论说某些简单任务使得游戏有趣，这些任务的完成需要快速的视觉识别技能和原始的狩猎天性。人类天生对运动敏感，远胜于对外形和颜色。视频游戏给我们的大脑带来了极大的视觉变换上的冲击，“我们从非洲高原走出来”后逐渐丧失了感受这种冲击的机会，取而代之的是法国油画带给我们的静态视觉感受。另外一种理论是经典的条件反射理论，按照条件反射理论：任何不断重复的行为加上周期性的正反馈将引起人类大脑生理上的变化，使得人类不由自主地重复该行为。另外还有一种理论说任何人喜欢表现自己的优点，却羞于表现自己的缺点，因此在完成他们自己擅长的任务时最开心。

因此，趣味性强的游戏需要带给玩家一定的优势，但不能太多，因为游戏过程还是需要一定挑战的。那么怎么才能使我们的 AI 系统尽可能地做到这些呢？我们需要维持系统的反应速度以提供适量的剧情和反应能力。我们应该保证游戏难度不是很大(因为如果觉得没机会获胜，人类很容易会放弃，而为了面子，他们可能会说："我只是不想赢而已！")，但我们也不应该使游戏太简单，因为趣味性的魅力在于挑战能力极限。而每个玩家的能力极限是不同的。这也正是 AI 难度自适应技术的驱动力之一。这种系统通过监督玩家的游戏情况，判断玩家的水平，然后调整智能体的 AI 水平。

尽管自适应元素是解决游戏难度设计上的"圣杯"，但依然存在一些需要解决的问题(特别是要注意为那些能力较低的人降低游戏难度)。

然而有朝一日当我们能很好地处理难度等级问题时，特意开发的自适应系统将被取代。其中一种取代方案是建立人类玩家的能力模型，并尝试查明伪反作用因素。在正确调整这些因素后，玩家不应再能发现时好时坏的行为模式，否则将导致玩家对 AI 对手的压倒性优势。同时，为了让人类具备少许优势，应该保持玩家处于"40%健康"的状态，这意味着玩家在游戏中会碰到麻烦，但依然可以牢牢控制游戏。

趣味性的另一元素是新颖性。新颖的游戏将激发我们尝试的欲望，而那些陈旧的游戏激发不了我们的欲望。在游戏《防御者》刚出来的时候，玩家能忍受其糟糕的难度等级设置，就是因为该游戏的新颖性。而现在任何人想推出相似的游戏，必定遭受失败，因为这样的游戏已经不再新颖。现存的游戏基本上都是包含难度控制模式的非平衡类游戏，而不存在单纯依靠 AI 和纹理图形的游戏。有人宣称游戏《荣誉勋章》的 AI 敌人是独特的，这些 AI 敌人会捡起玩家扔过来的手雷并扔回去。其实该创意很简单，只要在脚本系统中加入少量代码就可完成。但在此之前没人想到这么做，因此该创意为《荣誉勋章》赢得了市场。

当前游戏实体所表现出来的 AI 不仅仅是难度等级和创意问题，还存在很多其他因素，包括游戏性机制、控制模式、能量提升总量和时间限制等，这些因素也会对 AI 系统的成败产生影响。在设计 AI 系统时需要考虑很多的因素，而趣味性是最重要的，如果游戏的趣味性不好，AI 系统再智能，游戏也不会有任何市场。

24.2.2　随机感

几乎所有游戏的 AI 系统和操控系统都在一定程度上使用了随机性事件。使用随机性事件的出发点是提高游戏的重复可玩性，该属性用于评价玩家在"游戏通关"或者水平进步后是否还有重新玩该游戏的欲望。有关资料证明当前存在两种类型的游戏，其重复可玩性最好：游戏性固定且支持多玩家的游戏(比如《雷神之锤》和《象棋》)、游戏性固定且采用平衡随机性的游戏(比如《俄罗斯方块》和《扑克》)。不管哪种类型的游戏都必须提供良好、固定的游戏感受。

平衡随机性意味着游戏性的某些元素是随机的，但不会对游戏的输出产生很大影响。在《俄罗斯方块》中，不管积木以何种顺序掉下，只要玩家保持冷静的头脑，都能坚持很

长时间。甚至在《扑克》中，不管牌运有多差，优秀的玩家都有机会打出好牌并赢得游戏。而具有非平衡随机的游戏让人觉得赢得游戏的唯一途径是靠运气。这会让玩家觉得他没法控制游戏，他所取得的成果有可能在任何时候被破坏。在游戏中使用了过多非平衡随机性的游戏必定会有灭顶之灾。

非平衡随机行为一般是通过调用 randNorm()函数引入游戏的，该函数的调用将返回一个范围为 0.0f 到 1.0f 的随机浮点数。当通过该函数的返回值来控制 AI 行为时，非平衡随机行为就在游戏中起作用了。这种随机行为产生方式是错误的理由很简单。人类本能地不愿接受统计意义上的随机性。当被问到“抛了 30 次硬币，每次都是字面朝上，那么下次再抛时会是哪一面朝上？”时，大多数人会回答“花面！我确定！”，尽管该面在前 30 次中没有一次出现。概率和正常人脑最格格不入，这也正是为什么很多彩票机构能赚到钱的原因。如果大多数人能意识到被闪电击中的概率比中普通的州立彩票的概率要大 600 倍，他们就会吃光每周五花在彩票上的 50 美元。

那么怎样才能保证游戏中的随机性是平衡的呢？答案很简单，舍弃随机行为。比如在某 AI 决策函数的编码中只在 70%的情况下需要做出反应。如果我们使用表达式“randNorm()< 0.7f”，从统计学上说，我们已经解决了该问题，该表达式只在 70%的情况下返回真。但对于短时间运行的简单游戏它也可能全部返回假。这样的情况虽然不多，但从统计学上说是完全可能的。而这对游戏又意味着什么呢？非平衡随机性。我们需要创建一系列平衡的输出，以保证我们输出结果和理论上的随机结果更为接近。同样旨在获得平衡随机的输出，我们需要在游戏开始阶段创建游戏所需的数据串，数据串的长度大致等于平均条件下游戏所需的数据量。假如游戏中某函数预期调用 20 次，为了获得平衡随机的输出，在游戏开始阶段我们将产生 20 个布尔数，其中 14 个数为真值，位置随机，然后把这些数压入堆栈中。每次需要调用该函数时，弹出布尔数决定调用行为。我们也可以对其进行某些统计学上的变换。在游戏过程中该函数可能调用 18 次或者 23 次(如果游戏时间太长，可以在游戏的过程中产生数量合适的布尔数)，我们需要对初始数据进行一定的改变，故函数的返回值可能处于 65~75%之间。由于统计数据是通过一系列“随机”结果获得的，因此必须保证所给出的这些统计数据和期望非常接近。抛开对“真正随机”的依赖得到的结果才不至于使玩家有上当受骗的感觉。

24.2.3 一些令 AI 系统看上去非常愚蠢的因素

- 使用机枪“规则”的普通敌人。这些规则是：躲避第一枪，不首先开枪，通过示踪弹暴露自身位置并等敌人靠近后疯狂扫射。如果利用得好，这些规则是好规则；而如果断章取义地利用或者滥用，这些规则却是愚蠢的规则。AI 敌人是应该经常性躲避射击，但不能把机枪当作消防龙头。在近距离遭遇战时，应该提高射击的准确度，而不是习惯性地扫射却打不中对手，这会使玩家觉得 AI 系统很愚蠢。
- 糟糕的探路系统。没有任何其他东西能比糟糕的探路系统更能显示机器的“愚蠢”。包含糟糕的探路系统的 AI 角色会使 AI 系统看起来非常愚蠢。没有良好的探路系统，AI 角色会选择糟糕的前进路线并且不会躲避移动障碍物。没有良好的探路系

统，高速移动的 AI 角色会围绕着某点选择却不知道怎么靠近。没有良好的探路系统，位置接近的 AI 角色会拥挤到一起动弹不得。所有这些现象将使得 AI 系统看起来相当愚蠢。另外，没有良好的探路系统，AI 角色不得不费劲地修墙等。

- 敌人在知道人类玩家拿着链炮躲在角落里的情况下却非常平静地在角落边上漫步，这样的敌人使得 AI 系统看起来非常愚蠢。智能敌人是不允许出现这样的自杀性行为。那么智能敌人应该怎么做呢？简单地离开？不，他应该跃出并朝玩家所在的位置扔颗手雷。
- 那些看到地上 35 具同伴尸体却不去想“周围是否存在狙击手或者死亡之城”，反而踏着同伴尸体巡逻的 AI 敌人使得 AI 系统看上去非常愚蠢。这些敌人对周围环境的变化不敏感，确实有些游戏在设计时会隐藏尸体，不允许其他角色通过尸体了解周围环境，但在进行游戏设计时我们必须保证处理该问题的一致性，要么隐藏尸体，要么使 AI 敌人能对显示的敌人尸体有所反应。

24.3　有关产品的问题

在设计 AI 系统时，我们还需要考虑一些和产品相关的问题。游戏设计团队越来越庞大，游戏也越来越庞大而且复杂。

- **保持 AI 行为的一致性**。美工领导在考虑整个游戏产品感官效果时需综合考虑各美工的设计质量和一致性，在进行游戏 AI 系统的设计时也需要保证整个 AI 系统效果的一致性。
- **提前思考游戏参数的调试问题**。在创建游戏 AI 系统的开始阶段就思考参数的调试问题将使游戏的整个开发过程比较顺利。
- **预防 AI 系统的未知行为**。通过假设 AI 系统的某些未知行为并做出相应的处理将使得 AI 角色看起来更加智能。
- **设计人员使用工具的方式是不同的**。设计人员不是程序员，我们不应该把他们看成程序员。AI 工具在提交给设计人员前需要考虑某些问题。

24.3.1　保持 AI 行为的一致性

任何人在 1~3 关获得娱乐并能很容易地通关后，就会继续玩第 4 关和第 5 关以感受不同难度所带来的不同刺激。但当他们发现这些关和前几关几乎没有延续性而且游戏性机制极差时，他们就会停止游戏。出现这种情况的主要原因是游戏开发团队中多人参加了 AI 系统的开发，但他们相互之间缺少沟通。然而该问题最近有下降的趋势，因为 AI 系统在游戏中越来越重要，开发过程中花费越来越多的时间进行 AI 系统的协调和改进。越来越多的 AI 任务是在经过谈论后再划分并由不同的程序员完成的。

解决 AI 设计一致性问题的办法之一是通过“任务说明书”的方式构造游戏。对于任何给定的游戏，应该对游戏性和 AI 系统的目标有清晰而简单的描述。运动类游戏的目标可以是“提供有趣快速的篮球比赛，通过统计学方法模拟角色的移动、投篮能力和比赛过程中的叫喊，但提供快速街机风格的运动系统以及超顶级的特殊运动和防御机会。”而格斗

类游戏的目标可能包含“创建空手道模拟环境，在该环境下玩家可以应用复杂的组合行动和反击行动，而无需等待前一操作完成后才能继续下一操作”。这样，就算有很多程序员来共同完成 AI 系统的不同部分，他们依然知道游戏的目标并按照设计人员的要求实现系统。他们不会产生有关游戏设计方式和编码方式上的奇怪想法。

24.3.2 提前思考游戏参数的调试问题

AI 系统参数的调试是游戏开发过程中很重要的一步。关注下暴雪公司的游戏：《魔兽》、《星际争霸》以及他们新出的在线游戏《魔兽世界》。几乎所有暴雪推出的游戏的开发过程都会经历一个和其他很多游戏整个开发过程一样长的 beta 版测试期。尽管这些游戏的游戏性比现有的其他游戏要优秀得多，尽管这些游戏都已经推向了市场，他们依然会不停地调试游戏性平衡问题，追求完美的游戏效果。为什么呢？主要原因是他们希望推出尽可能好的游戏，另外还有一些次要原因：游戏发烧友的需求以及暴雪自身对创新的追求。是的，这样做会花费他们很多的金钱，但也正因为这些投资，他们的产品卖得异常火爆。其他一些公司，比如 Square 和 Nintendo 也模仿暴雪投入重资测试游戏平衡性，不停地进行参数调试，直到没有可调试的内容为止。

为了更便于对该层次完美性的追求，AI 系统需要支持游戏参数的快速调试以及游戏平衡性的调试。数据驱动 AI 在该方向上迈出了重大一步，支持流水化的参数调试。程序员可以从大量的参数调试以及其他一些数据驱动问题，比如不同游戏关中敌人的布局、敌人对玩家位置和状态的特殊反应中解脱出来，使得参数调试过程得以加速。另外，当该过程变得快速而简单时，设计人员就更愿意花更多时间进行调试，这是一个良性循环的过程。

另一个需要注意的问题是不要让幻数出现在 AI 决策系统中。比如游戏中出现这么一条语句：if(m_nearestEnemy<45)，我们应该注意其中的“45”，应该保证在这里使用的 45 是经过精心设计而不是随手确定的，如果不能保证，我们应该用一个参数代替它，并在游戏调试过程中完成该参数的选择。第 25 章“调试”详细说明了 Widget 系统，该系统支持这类参数的调试。这类系统在 AI 系统中是必要的，我们可以依靠它们完成游戏性和平衡性的调试。

24.3.3 预防 AI 系统的未知行为

我们需要时刻警惕那些无知行为对 AI 系统的影响。就算我们自己认为 AI 系统不会出现无知行为，可能性还是存在的。如果不能确定没有未知行为，那么肯定存在未知行为。软件系统本身都有可能发现并利用我们不小心留下的后门，更不要说人类玩家了，任何人都会有自己玩游戏的方式，这样发现后门的概率将会大大增加。我们应该注意该问题并消除游戏中可能存在的后门。定时器行为是最早用来处理未知行为的技术，但简单地退出会使 AI 行为看起来很愚蠢。我们应该习惯于为 AI 行为设计合适的处理方式而不是简单地退出。该思想也可用于完成数据的检测。固定模式的“检测器”可以帮助我们节省大量开发时间，该检测器在数据从命令行输入或者在游戏装载时(可以在游戏发布时去掉)完成对 AI

数据的扫描，以确定这些数据是否存在不一致性、完全错误、破坏路径节点网络、丢失或者重复单元等问题。这些代码是可以省略的，但如果没有这些代码，我们有可能会为了调试某些小问题而花费数周时间，比如由于设计人员误操作输入文件造成 AI 系统的异常或者由于脚本中的某字节而造成版本控制的崩溃。

24.3.4　注意设计人员工具使用方式的差异性

提供给设计人员使用的 AI 工具需要特别注意。我们不需向设计人员提供游戏所有的逻辑和功能性内容，而只需提供足够他们使用的逻辑即可，这样可简化工具的设计和使用。那么需要提供逻辑表达式吗？只需考虑“与”、“或”、“非”和“异或”等逻辑。逻辑技巧并非一些程序员的强项，更何况是刚刚开始游戏业工作的员工。这不是对设计人员的侮辱，他们从事着游戏业中最困难的工作。他们需要在该年完成游戏设计这一伟大工作后，在来年继续改进该游戏。每个人认为自己是设计人员，就像每个人认为自己会唱歌一样，我们需要照顾到不同人的不同习惯。另外，不要在命令行中包含太多参数设置问题(如果可以，通过 Export 导入这些参数或者编写批处理文件来完成参数的设置)。在工具和脚本系统中提供经过多次验证的示例函数。最后，我们应该虚心接受设计人员在 GUI 和功能上提出的质疑并做出相应的改进。

24.4　小结

现代 AI 游戏引擎是相当复杂的软件系统，在实现时需要考虑很多方面的问题。本章特别关注设计、娱乐和产品等方面的问题。

- 在进行 AI 引擎设计时需要考虑的问题包括：数据驱动问题、OTM 综合症、LOD AI、支持 AI 和其他一些 AI 设计思想。
- 娱乐相关的问题包括：趣味性因素、随机感、难度设置和一些使 AI 系统看起来愚蠢的问题。
- 产品相关问题包括：AI 表现的一致性、游戏的参数调试、不可预测问题的避免以及设计人员使用工具的方式。

25 调　试

本章将阐述 AI 游戏开发中的另一部分——调试，涉及到调试过程中普遍关心的问题以及针对可能出现的程序错误编写有效代码的几种方法。最后本章给出了一个名为 Widget 的通过 Windows MFC 实现的运行时调试工具。

25.1 AI 系统的通用调试

由于 AI 引擎的固有特性，调试是一件累赘的事情。AI 总是包含在许多游戏系统中，负责连接控制、物理、声音、游戏性机制和输入输出系统。很多时候程序错误是基于 AI 系统出现的，但却隐藏在 AI 所连接系统的代码中，直到 AI 系统开始执行该段游戏代码程序错误才表现出来。一旦出现问题，我们不仅仅要通知其他同事需要修改他们的代码，而且还要备份一个可以重现该问题的测试用例或者直接把它们放在工作空间中单步调试。这样，我们就能节省在发送邮件和等待过程中浪费的很多时间和精力。

采用第 23 章提到的分层式 AI 设计的一大好处是对各个层次断点设置的支持，把每个子系统模块分成一个个断点，以便于找出代码中出现问题的地方。所以，程序员只需调试自己负责的子系统，而不必跟踪整个系统或者关系复杂的类。

25.2 可视化调试

可视化调试是利用从游戏运行过程中获得的有关当前系统状态的可视化信息来调试程序。游戏运行时，通过输出 AI 角色的信息来发现问题，这些信息的输出形式包括相关 AI 角色的当前状态、意图、移动方向和思考过程的文本，甚至可以是游戏中影响图的变化情况。游戏 AI 系统从这些调试信息中获得的好处比其他系统要多，好处如下。

25.2.1 提供各种信息

可视化调试包括向游戏界面和其他可视化辅助程序输出文本。可以用线标出每个 AI 角色感兴趣的目标，或者突出显示导航跟踪的方法来发现导航系统中的错误。影响图数据的可视化表示有助于调试游戏(实际上，我们并不想关心正常游戏角色的绘制，而只关心异常角色的影响数据)，这是一种看待游戏的抽象数据组织的简化方法。

25.2.2 有助于调试

在游戏的每个阶段，必须给正在发生的事情一个恰当的可视化表示方法，这样才能保证所发生的事情正是我们所想的。如果我们正在编写一个与视线等特殊感知有关的代码，往游戏中加入可视化系统，既便于观察到 AI 确定视界的所有踪迹，便于准确地知道视界什么时候开始、在什么时候结束。然后深入游戏长时间观察系统的边界，确保所做即所想，但要保证我们将逐渐找到对该系统的感觉。这对于辅助特征参数如反应时间来说尤其重要。往游戏中添加指示器，然后监测几百次，这样有助于找出在行为和感知中建立空穴时的错误数学反馈，也有助于系统的调试，使游戏性感觉变得更快和更简单。

25.2.3 时序信息

我们一般很难使那些只在单个游戏循环出现的错误复现在调试工具中，尤其是那些主游戏更新循环是基于时间(而不是基于帧)的游戏，调试过程根本无法设置恒定的增量。如果游戏时钟采用处理机的系统时钟，用断点来暂停和单步调试代码会有比较大的时间增量，因为在调试时，时间仍然是继续流逝的。若采用恒定增量时间来调试，这个问题将被最小化。假若不能如此，那么在游戏高速运行时，我们可以根据可视化调试信息建立同屏信息来发现问题。值得注意的是，因为游戏会因为太多的文本而可能慢下来，所以在重现错误的过程中又会出现新的问题。

25.2.4 监视状态转变

当采用基于状态的决策结构时，我们可以根据 AI 系统输出到游戏中的可视化状态信息来监测奇怪状态的变化。使游戏把状态信息直接输出在角色附近或者它的上方，这样我们可以很轻松地把数据和角色联系起来。其他有效的状态信息包括继承状态、有效的统计数值，比如角色处于该状态的时间和暂态信息。

25.2.5 有助于控制台调试

在 PC 上运行控制台调试程序，一般先在 Windows 或 Linux 环境下开发，然后上传可执行文件到测试控制台。由于是远程调试，不能够使用一些通用的调试技巧，因此在控制台屏幕上显示文本和图片成为最主要的调试手段。

25.2.6 调试脚本语言

除非特地为脚本语言写一个完整的调试系统(没有多少游戏允许以这种进度调试游戏)，否则要从游戏技巧中发现错误，测试人员会觉得孤立无助。常用的技巧之一就是利用脚本编辑器把专门的调试命令嵌入到脚本中；也可以根据 AI 行为脚本设置同屏文本。

25.2.7 双功能影响图

游戏中的影响图系统也可以当作一个可视化调试工具来使用。通过增加每个影响单元的空间(如果空间有剩余)或者完全取代系统(如果所测试游戏的其他部分不需要影响图)来

输出专门的调试信息是该技术的一贯用法。我们可以输出飞行时的地形分析、各个 AI 单元的躲避代码。游戏运行时使系统输出影响图(IM)的内容，所有与确定位置相关的信息就这样可视化地表示出来。该位置可在影响图中设置。

25.3 Widget

在编写特殊行为感知的代码时，我们希望这些数值在游戏运行过程中不仅是可视的，还可以修改。Widget 就是在这种理念的基础上提出并实现的，该工具能可以很方便地集成 Windows 系统下的游戏，也可以很容易地移植到其他非 MFC 系统中。

Widget 支持在游戏中设置各种各样的“控制旋钮”。游戏运行时会出现一个称为 Widget Bank 的小窗口，其中存储了所有的 Widget。在 Widge Bank 中，我们可以实时修改与 Widget 相关的变量值。

25.3.1 实现

Widget 的实现代码非常简单，只需通过若干基本规则完成启动和运行。整个系统由 Max Loeb 完成，Max Loeb 也是本书的审阅人之一。系统的几个基本部分如下：

- WidgetBank。处于 Widget 分级结构的最高层，是 Widget 窗口群的聚集。
- Widget Group。处于 Widget 分级结构的第二层。每个 Widget 是 Widget Group 的一部分；没有任何 Widget 直接从属于 WidgetBank。Widget Group 也可包含若干子 Widget Group。
- Widget 类。Widget 一般有以下几种类型：基本按钮(引发一定事件操作的类型)、小圆按钮(用于选择两个标记设置)、二值按钮(布尔值表示的开或关的按钮)、卷轴和监视器。
- EventHandler 类。支持在 Widget 类中调用回调函数。

在 Widget 类中有两个基本函数：update 和 draw，使得 Widget 有公共的父类。Widget bank 和其他类可以通过这两个基本函数调用所有的 Widget。

WidgetBank 类(程序清单 25-1 给出了其头文件)是整个系统的主要窗口。类中使用了很多 MFC 函数。

程序清单 25-1　WidgetBank 的头文件

```
/////////////////////////////////////////////////////////////////////////
***
* WidgetBank: This is the window that houses all the Widget windows, ie.
*           camera Widget, bone Widget, light Widget, and so on.
*
/////////////////////////////////////////////////////////////////////////
*/
class WidgetBank : public CWnd
{
public:
    // constructors
```

```
    WidgetBank();
    virtual ~WidgetBank();

    // member methods
    BOOL Init();
    void RedrawWidgets();
    void UpdateWidgetBankSize();
    void Update();

    // Widget creation methods
    int GetHeight();
    Group* AddGroup( char * label );
    afx_msg UINT OnNcHitTest(CPoint point);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);

    // member variables
    Group * myWidgets[MAX_NUM_WIDGETS];

    int m_numWidgets;
    int m_totalWidgetHeight;
    CRect m_ClientSize ;

    DECLARE_DYNCREATE(WidgetBank)

    DECLARE_MESSAGE_MAP()

private:
    int m_id;
public:
    afx_msg void OnNcDestroy();
    virtual BOOL CreateEx(DWORD dwExStyle, LPCTSTR lpszClassName,
LPCTSTR
                          lpszWindowName, DWORD dwStyle,const RECT&
rect,
                          CWnd* pParentWnd,UINT nID,LPVOID
lpParam=NULL);
    afx_msg void OnVScroll(UINT nSBCode,UINT nPos,CScrollBar*
pScrollBar);
};
```

Widget Group 是 Widget 一种分层排列的组织方法。初始化时所有群都是最小的。单击标签打开一个群，然后才能打开包含独立 Widget 的库。而使用时只有那些希望在任何时间看到的 Widget 才会打开它们的群。该类有点复杂，因为群是 Widget 组织函数存在的主要地方。在程序清单 25-2 的头文件中，正如读者所看到的一样，主要功能就是往库中添加描述各种类型的 Widget、更新窗口动作、调整窗口大小等。

程序清单 25-2 Group 的头文件

```
/*********************************************************************
* Group:  A Group represents an entry in the Widget bank, and
*         can house other groups or Widget. It contains a
*         header which can be expanded/contracted to show/hide
*         its child groups or child Widget, which are added with
*         subsequent calls to AddScrubber, AddOnOff, etc.
*
*********************************************************************/
class Group : public CWnd {
public:
    // constructors
    Group( char * label, CWnd * pWin, int pos, int height, int width,
           const int level );
    virtual ~Group();

    // member methods
    virtual void Update();
    void OnClickHeader();

    bool IsExpanded(){ return m_status; }
    int GetHeight();
    int GetClientHeight();
    int GetPrevPos(){ return m_prevPos; }
    void SetPrevPos( int prevPos ){ m_prevPos = prevPos; }
    void SetWidgetBank( WidgetBank * wb ){ m_WidgetBank = wb; }

    Group * AddGroupWidget( char * label );

    ScrubberWidget<int> * AddScrubber( char * name, int & val );
    ScrubberWidget<float> * AddScrubber( char * name, float & val );
    ScrubberWidget<unsigned char> * AddScrubber( char * name, unsigned
                                                 char & val );

    OnOffButton * AddOnOff( char * name, bool & a, int ID1 = 0,
                            EventHandler * h = 0 );

    void AddWatcher( char * caption, float & val );
    void AddWatcher( char * caption, int & val );

    void AddText( char * caption );
    RadioButton * AddRadio( char * groupName, char * leftName, char *
                            rightName, int & val, int id1, int id2,
                            EventHandler * h = 0 );
    BasicButton * AddBasicButton( char * filename, int id,
                                  EventHandler * h = NULL );
```

```
    int Draw( int y_pos );

    // MFC Overrides
    DECLARE_MESSAGE_MAP()
    afx_msg void OnNcDestroy();
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);

    // member variables
private:
    Group * m_childGroups[ MAX_CHILD_GROUPS ];
    Widget * m_childWidgets[ MAX_CHILD_WIDGETS ];
    WidgetBank * m_WidgetBank;        // a pointer to the parent
Widgetbank
    CButton m_header;
    COLORREF m_color;                 // the color of this Widget
    int m_top;                        // position of the top of this
Widget
                                      // in Widget bank
    int m_prevPos;                    // used for positioning groups when
                                      // drawing
    bool m_status;                    // is this Widget currently
expanded?
    unsigned int m_numChildGroups;    // number of subgroups contained in
                                      // this group
    unsigned int m_numChildWidgets;   // number of child Widget in this
                                      // group
    int m_level;                      // how many levels deep is this
nested?
};
```

EventHandler是一个基本回调类，包含一个称为UIEvent()的虚函数。为了使用Widget的event handler功能，必须建立EventHandler类的一个子类，在子类中实例化复制函数，然后重载UIEvent()函数使其成为子类的回调函数。要确保EventHandler子类指针指向父类，这样回调函数才能访问到需要的东西。建立Widget按钮时，给按钮分配一个ID。按钮就会按时调用UIEvent()函数，取得按钮的ID。那些重载函数就可以根据ID来确定要进行什么操作。

接下来我们介绍不同类型的Widget。在介绍完每一种类型后都会给出实现该类型的程序示例文件。

25.3.2 BasicButton

BasicButton是Widget中最简单的按钮，可以用此按钮上的标签引发回调事件。程序清单25-3列出了实现该按钮的头文件。从中可以看出，该类型的按钮实际上仅仅是MFC中CButton的一个封装。

程序清单 25-3 BasicButton 的头文件

```
class BasicButton : public Widget
{
public:
    BasicButton( EventHandler * eventHandler = 0 );
    ~BasicButton(void);
    void Create( char * label, int id, CWnd* pWin, int pos );
    void Draw();
    EventHandler * m_eventHandler;

    DECLARE_MESSAGE_MAP()
protected:
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

private:
    CButton m_button;
};
```

25.3.3 Watcher

Watcher 是一个能显示变量值的模板 Widget(尽管现在只适用于 int 和 float 的变量类型)，我们不能直接从 Widget 本身修改变量值。Watcher 用于不断更新变量，而不用于在 Widget 处修改变量，因为这样的话，游戏很快就会把改变覆盖掉。

程序清单 25-4 实现 Watcher 的头文件

```
template <class T>
class Watcher: public Widget
{
public:
    Watcher( int & watch);
    Watcher( float & watch);
    ~Watcher(void);
    void Create( CString label, CRect r, CWnd* pWin );
    void Draw();
    void Update();

private:
    CStatic m_label;
    CStatic m_watch;
    T & m_val;
    int m_frameCount;       // frame counter
    int m_updateInterval;  // update every this many frames
};
```

25.3.4 RadioButton

RadioButton 是标准的 Windows 控件，用于完成对象的选择。目前的实现只支持两个选择项，但是可很容易地扩展到任意数量的选择项。它的头文件在程序清单 25-5 中。

程序清单 25-5 实现 RadioButton 的头文件

```
class RadioButton : public Widget
{
public:
    // constructors
    RadioButton( char * groupName, char * leftName,
        char * rightName, int & val,
        CWnd * pWin, int yPos,
        int id1, int id2,
        EventHandler * h );
    ~RadioButton(void);
    void Draw();

protected:
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

private:
    // member variables
    CButton m_GroupButton;
    CButton m_LeftButton;
    CButton m_RightButton;

    int & m_val;

    EventHandler * m_eventHandler;
};
```

25.3.5 OnOffButton

这种类型的 Widget 是一种布尔值的二值元件。可以用 Windows 控件的“选择框”类型，也可以用标准的按钮。程序清单 25-6 列出了头文件的基本信息。

程序清单 25-6 实现 OnOffButton 的头文件

```
class OnOffButton : public Widget {
public:
    // constructors
    OnOffButton( bool & state, EventHandler * eventHandler = 0 );

    // member methods
    void SetStyle( int style );
    void SetCheck( bool checked );
    int GetCheck();
    void Draw();
    CButton m_button;

    DECLARE_MESSAGE_MAP()
protected:
```

```
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

private:
    // member variables
    bool & myState;

    EventHandler * m_eventHandler;
};
```

25.3.6 ScrubberWidget

Scrubber是一个非常有用的Widget控件。它可以访问显示在Widget中的float、int和char类型的变量值。我们通过单击Widget就可以在设置好的最大值和最小值之间任意拖动。我们也可以设置刷新速度(慢、正常或者非常慢)。程序清单25-7给出了其头文件。

程序清单25-7 实现ScrubberWidget的头文件

```
template <class T>
class ScrubberWidget : public Widget {
public:
    // constructors
    ScrubberWidget(T & var );
    ~ScrubberWidget();

    // member methods
    void Refresh();
    void Draw();
    void OnEditKillFocus();
    void Create( char * label, CWnd* pWin, int pos, SCRUB_SPEED speed =
                 REGULAR_SPEED );
    void SetMin( T min ){ myHoverButton->m_minValue = min; }
    void SetMax( T max ){ myHoverButton->m_maxValue = max; }
    void SetMinMax( T min, T max ){myHoverButton->SetMinMax( min, max
);}

    // Overrides
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);

    // member variables
    CEdit myEdit;
    HoverButton@T: * myHoverButton;

    T &scrubVar;      // the value being changed

    DECLARE_MESSAGE_MAP()
    afx_msg void OnNcDestroy();
};
```

25.3.7 程序集成

Widget 的使用过程可分成以下简单几步：

(1) 把 WidgtsBank.h 文件包含到准备使用 Widget 的类中。

(2) 然后，往类中添加一个被称为 AddWidgets()的函数。如果该类是“主类”，那么该函数需要 WidgetBank 指针作为参数。如果是“从类”，那么该函数需要 Widget Group 指针作为参数。

(3) 不管是添加 WidgetBank、Group，还是添加 Widget，都需要重载 AddWidgets()函数，使用方法同程序清单 25-8 中的示例。

(4) 指定如何更新 WidgetBank，如要获得完全更新的 Widget，那么每帧都必须调用 update()函数。

程序清单 25-8 Widget 使用指南示例

```
// Our car's EventHandler class. Note that it must
// be created with a pointer to the car so that
// we can interact with it in our UIEvent's switch
// statement. Alternatively, we could also
// simply derive our car class directly from
// an EventHandler, and remove the need for
// a Car pointer.
class CarEventHandler : public EventHandler
{
public:
    CarEventHandler( Car * car ){ m_car = car; }
    virtual void UIEvent ( WPARAM id );
private:
    Car * m_car;
};

void CarEventHandler::UIEvent(WPARAM id )
{
    switch (id)
    {
    case Car::IGNITION_KEY:
        m_car->StartCar();
        break;
    case Car::WIPERS_CONTROL:
        m_car->StartWipers();
        break;
    case Car::AIR_COND:
        m_car->ToggleAirCond();
        break;
    }
}

class RacingGame
{
```

```
public:
    RacingGame(){};
    void AddWidgets( WidgetBank wb );

private:
    Car    m_car;
    Track  m_track;
};

class Car
{
public:
    enum {
        IGNITION_KEY,
        WIPERS_CONTROL,
        AIR_COND
    };
    void AddWidgets( Group * g );
    void StartCar(){ m_engine.Start() }
    void StartWipers(){ m_wipersOnOff = true; }
    void ToggleAirCond( m_air ? m_air = FALSE : m_air = TRUE );

private:
    Engine m_engine;
    bool m_lightsOnOff;
    bool m_transmission;
    bool m_air;
    bool m_wipersOnOff;

    CarEventHandler m_eventHandler;
};

/*****************************************************************************
* Name:  AddWidgets
*
* Info:  A typical AddWidgets function for a fictitious racing
*        game. Because the RacingGame object is high level, it
*        will be adding Groups directly to the Widget bank. Actual
*        Widgets will be added to these groups by the AddWidget
*        functions of lesser, individual components of the game.
*
*  Args: wb - A pointer to the WidgetBank, which is the top-level
*        parent of all Widgets and Groups. Again, you don't add
*        Widget
*        directly to the Widget bank—you only add Groups.
*
*****************************************************************************/
void RacingGame::AddWidgets( WidgetBank * wb )
{
    Group * g;
```

```
    // Add our first group to the Widget bank. AddGroup() returns a
    // pointer to the group it created. You can either use this pointer
    // to add Widget now, or, preferably, pass it to the AddWidgets()
    // function of a lower-level contained class.
    g = wb->AddGroup("Car Properties");

    // Now that we have our group, we'll pass it to the AddWidgets
    // function of our car class object, which is a member of a
    // RacingGame object.
    m_car.AddWidgets( g );

    // That takes care of the car's Widget, so let's add Widget for
    // the race track. We'll make another group, and reassign our
    // group pointer to it
    g = wb->AddGroup("Track Properties");

    // Again, we pass the newly assigned pointer to the AddWidget()
    // function of a lower level class, this time a RaceTrack object.
    m_track.AddWidgets( g );
}


/*****************************************************************
*  Name: AddWidgets
*
*  Info: A typical AddWidgets function for a fictitious car class
*        to demonstrate the use of Widget. This example only
*        covers a Car object, but remember that you have to write
*        an AddWidgets function for any class that you want
*        to have Widget. From here you might write AddWidgets
*        functions for your racetrack class, your environmental
*        class, your AI classes, etc.
*
*  Args: wb - A pointer to a Group. We use a Group pointer
*        to add the actual Widgets to our application.
*
*****************************************************************/
void Car::AddWidgets( Group * g )
{
    // We'll need a pointer to a group. We'll call it pg, for "parent
    // group"—be careful not to confuse it with the group pointer
    // that is being passed into this function.
    Group * pg;

    // Our car class contains an engine object. Let's give it its own
    // Widget group. Groups can contain other groups, which gives
    // you a lot of flexibility to organize your Widget.
    pg = AddGroup("Engine Properties");
```

```
    // Our engine class has its own AddWidgets function, so we'll
    // pass it our new group pointer

    m_engine.AddWidgets( pg );

    // Our car class has some member variables that would
    // be fun to control while the game runs. We'll hook up
    // some Widget to them now, using the group pointer
    // that was passed in

    // let's start by adding a Widget to control the on/off state
    // of the car's headlights
    g->AddOnOff( "Headlights", m_lightsOnOff );

    // it would be nice to monitor the car's fuel gauge--we'll
    // add a Watcher Widget.
    g->AddWatcher( "Fuel Level", m_fuel);

    // We want to tune the car's mass as it drives around, so
    // we'll attach a ScrubberWidget. We're going to catch the
    // ScrubberWidget pointer that this function returns, so that
    // we can change a setting
    ScrubberWidget * sw;
    sw = g->AddScrubber( "Mass", m_mass );

    // We don't want negative or absurdly huge values for the mass of
    // this car during scrubbing, so we'll set some limits on the value
    // using the pointer we got back from the AddScrubber function
    sw->SetMinMax( 0, 10000);

    // Car objects can have automatic or manual transmissions. We'll
use
    // a radio button, which allows you to have a caption for the
overall
    // control, as well as each actual button.
    g->AddRadio( "Transmission:", "Automatic", "Manual", m_transmission
);

    // For our final Widget, we'll add a button that starts the car's
    // engine and other systems. Because we want to attach some
    // functionality to this button(it won't do anything if we don't),
we
    // pass in an EventHandler object that we've written for Car
Objects.
    // We also pass in an enum name for the button. This enum value
will
    // become the id number of the Widget. When the button is actually
    // pressed by a user, its id number is passed to the EventHandler's
    // UIEvent function, and is used in a switch statement to call that
    // Widget's particular code.
```

```
    g->AddBasicButton("Start Car", IGNITION_KEY, m_eventHandler );

    // We'll add a few more Widget that use the same EventHandler.
    g->AddBasicButton("Windshield Wipers", WIPERS_CONTROL,
                      m_eventHandler);

    g->AddOnOff("Air Conditioning", m_air, AIR_COND, m_eventHandler );

}
```

如果我们继续往下看，就会发现在游戏中使用 Widget 可以加快游戏调试进度。为了获得更多好处，我们必须对 Widget 系统进行一定的扩展：

- 用#ifdef__DEBUG 预编译命令来封装所有的 AddWidgets()函数，或者在工程中采用条件属性代码来编译。游戏发布时再把条件属性的预编译开关全部移除。
- 采用 BasicButton 来保存或编写文本文件，该文件中包含所有 Widget 的值。我们可在文件中序列化所有 Widget 的变量值，游戏开始时，通过输入文件中的值初始化所有变量。采用该方法后，我们就不必花费整个小时来调整某个数值，然后为调整游戏中的初始值而不得不把所有数值都写在文件中。但是在发布游戏之前，我们必须把所有的初始值移出文件，也就是把该文件当作配置脚本来使用。

25.4 小结

AI 系统的调试是一项非常繁重的工作，因为它与其他游戏系统都紧密相连，调试 AI 系统是专注于发现案例代码或数据重现中的错误，该过程相当复杂。本章阐述了调试游戏 AI 引擎时 AI 开发者必须关注的几个问题。

- AI 通用调试方法可应用于其他非 AI 开发者的代码中，通过使用分层式 AI 设计方法可以减少该调试工作量。
- 可视化调试提供的多样化信息有助于游戏的调试，其能提供时间信息、监视状态变换。这种方法在控制台开发中尤其有效，有利于调试脚本语言，能够和游戏中使用的影响图系统很好地结合。
- 本章介绍的 Widget 库为用户提供了一个通用平台，该平台主要用于输出变量到简单用户接口。在游戏运行时，该接口既支持数值的监视，也支持变量数值的修改。

26 总结与展望

26.1 AI 引擎设计总结

本书已经阐述了许多相关背景与知识，希望读者掌握如何通过使用 CD-ROM 中的一些框架代码，并加上自己的努力和创造性来实现 AI 系统。我们涉及到了很多知识，从简单的到相当复杂的，从理论的到实践的，并通过完整的实例讨论了 AI 引擎的设计。

设计引擎时，我们可以对 AI 任务进行划分。划分后的 AI 系统包括以下几个层：

- 感知和事件层
- 行为层
- 动画层
- 运动层
- ST 决策层
- LT 决策层
- 基于位置的信息层

然后实现每层的逻辑时，在考虑采用什么决策技术时我们不能忽略以下 8 个方面的因素：

- 解决方案的类型
- 智能体的反应能力
- 系统的真实性
- 游戏类型
- 内容相关需求
- 游戏平台
- 开发限制
- 娱乐限制

26.2 AI 游戏的未来展望

对具有更高智能水平的 AI 对手的追求仍然在继续。尽管越来越多的人开始玩在线游戏，但还有很多人只玩单人游戏，而不玩在线游戏。这些人是游戏玩家的主体，他们需要越来越复杂的游戏，甚至希望游戏 AI 角色能智能到足够和其对抗的程度。

在游戏公众眼里中，AI 变得越来越重要。游戏评审的大部分时间都花在 AI 的利弊评估上。最近 10 年可以说完全是游戏图形学的天下，也涌现了很多研究成果，比如大规模多边形计算、逼真的纹理和光线处理技术，视觉效果完全达到电影标准的画面技术。一股推动力正进入并推动游戏 AI 系统的发展。游戏的 AI 系统越来越复杂，也越来越具有创造性，出现了模仿人类玩家的游戏风格和反应(主要是学习能力和对手模型)的敌人，也出现了为游戏问题(接口、紧急行为、创新性)提供创造性解决方案的 AI 对手，甚至出现了具有人类心情或者情感的对手。

以前的游戏缺乏个性化。对手之间几乎没有差别，就算有区别，也纯粹是统计学上的差别(如敌人 A 比 B 强大，但是 B 比 A 跑得快)。主要原因是以前的对手都是通过硬编码(以一种非常特殊的基于代码的方式编写)实现的，出发点是游戏平衡性和编写时间问题。相反，现代游戏的 AI 对手是在学习系统的基础上实现的，只具备有关游戏世界的基本知识，这些对手通常根据自身所处的环境来做出反应，并且它们将永远处于类似环境中。Black & White 采用了和上述模式类似的系统(尽管前面的描述非常简单)，几乎没有任何两个主要游戏角色会表现出相同的特性，甚至在玩家不变的情况下也如此。角色的个性取决于相当多的因素，以至于新行为和个性的出现也是不可避免的。因此最后得到的游戏角色具有更多的个性，系统的智能水平也更加令人满意。同样，玩游戏比看与游戏内容相同的书更有意思和体现智能。由于玩游戏是交互的，能唤起了我们的本能，把生命赋给我们所碰到的角色和单元，简单地说，就是把我们自己当成整个过程中的一部分。我们能和世界交互、改变世界，然后成为世界的一部分。但是看书仅仅是接收一个故事，不能回答我们的所有问题，不能提供给我们从另一个角度看问题的机会，由于作者的写作方式以及其表述故事的能力，其所写出的故事与故事本身存在一定的差距。这也是为什么完全由脚本实现的 AI 系统不能满足玩家，因为如果游戏受到脚本编写者自身的限制，往游戏中添加内容丰富的脚本编写人员的经验并不能代表所有人的经验。

一般来说，AI 也会导致游戏的变化，从而需要大幅度调整以适应这些变化。从短期看，这些变化主要发生在人机接口方面。很多游戏开始把语音命令包含进来，这些语音命令从一个磁头组或麦克风获得。而将来某一天，游戏完全可能需要完整的语音识别系统和类型转变能力。同时，游戏出现了在线化趋势，这使得游戏在运行起来后就可能会永不停止。在线游戏中的 AI 角色或者 NPC 将因此获得长时间学习和个性化的机会。

总有一天，我们会拥有完全智能的游戏系统，这些游戏中的角色能针对人类玩家的不同水平表现出不同的风格、创造性、个性以及智能水平。那将是一个多么有趣的世界啊！

附录 有关 CD-ROM 的说明

本书附带的 CD-ROM 包括了本书涉及到的所有源代码以及演示程序，同时还包括了其他一些有益的资料。而且，读者也可以登录 Charles River 的网站获得有关本书的最新、最全的资料，网址是 www.charlesriver.com。

1. 内容

- **源代码**。所有的源代码都在相关章节的子目录中。所有的演示程序都通过 Microsoft Visual C++ 6.0 编译，并且编译得到的二进制文件可在相应的目录中获得。
- **图片**。本书中的所有图片包括在该目录中，其文件名同书中出现的图名。
- **有用超链接**。本目录给出了一些网络资源的超链接，包括通用资源和专用资源。这些链接可分为几大类：ALife、模糊逻辑、通用 AI 网址、遗传算法、基于位置的信息、神经网络、脚本、各种 AI 链接、游戏源代码和各种游戏 AI 问题。
- **库**。该目录中包括了本书演示代码实现时用到的两个最新库：OpenGL 的一个封装库 GLUT 和 Lua 语言。当然，读者可以从网上下载到更新的版本，但本书的示例只验证了光盘中提供的版本。

2. 系统配置需求

所有的演示程序都通过若干类型机器的测试，包括 Pentium 4 3.2GHz (台式机) +GeForce 3(显卡)和 Pentium 3 1GHz(笔记本)+ GeForce Go(显卡)。由于 GLUT 在所有 Windows 操作系统(Me、2000 和 XP)中都能很好地工作，因此这些示例应该在该环境下编译执行。所有的程序都是在 Microsoft Visual C++ 6.0 下编写和编译的，由于程序中用到了 GLUT，因此读者需要同时安装 GLUT 和 OpenGL。

游戏开发经典丛书

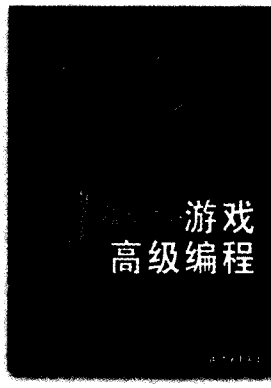
新手，可以从这里快速入门；老手，可以从这里稳固提升！



.NET游戏编程入门经典——C#篇
ISBN 7-302-12104-4



.NET游戏编程入门经典——VB.NET篇
ISBN 7-302-13590-8



Java游戏高级编程
ISBN 7-302-11083-2



3D游戏编程入门经典
ISBN 7-302-12167-2



3D游戏开发大全
ISBN 7-302-11914-7



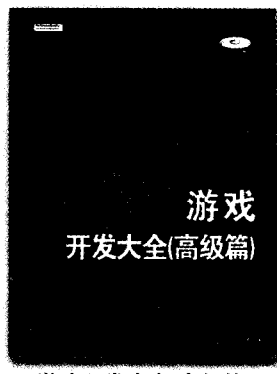
游戏测试精通
ISBN 978-7-302-16098-4



GPU Gems精粹2
ISBN 978-7-302-13943-0



AI游戏引擎程序设计
ISBN 978-7-302-16312-1



3D游戏开发大全(高级篇)
ISBN 978-7-302-16258-2

即将出版		
英文书号	英文书名	中文书名
1548504587	Ultimate Game Programming With DirectX	DirectX游戏开发终极指南