

本书是资深 Lua 游戏开发工程师 10 余年工作经验和智慧的结晶, Lua 语言创始人亲自作序推荐, 是 Lua 游戏开发领域最具实践意义和代表性的著作之一。它不仅详细讲解了在游戏开发中使用 Lua 的各种技术细节、方法技巧和最佳实践, 而且讲解了如何使用 Lua 作为主要工具将游戏设计转化为代码实现的过程。此外, 它还重点阐述了 Lua 语言的核心要素。最重要的是, 本书包含大量精心设计的案例, 并附赠了完整的源代码, 可操作性极强。

全书一共 15 章: 第 1~3 章简单地介绍了 Lua 语言的特性、授权, 以及在游戏开发中的强大用途; 第 4~5 章详细讲解了 Lua 语言的基本语法和核心要素; 第 6~7 章讲解了 Lua 与 C/C++ 程序的整合以及 C++ 的交互相关的技术细节; 第 8~9 章介绍了开发前需要做的准备工作, 以及如何设计 Lua 版本的实现; 第 10 章讲解了如何使用 Lua 来处理游戏数据; 第 11 章讲解了 Lua 驱动的 GUI; 第 12 章详细讲解了两个完整的游戏开发案例; 第 13 章结合实例讲解了如何使用 Lua 定义和控制 AI; 第 14 章展示了 Lua 在图形绘制和图像处理方面的强大功能; 第 15 章探讨了 Lua 与多媒体、Lua 脚本的调试、Lua 应用的资源管理以及 Lua 代码的发布等内容。

Paul Schuytema and Mark Manynen: Game Development with Lua (ISBN 978-1-58450-404-7).

Copyright © 2005 by CHARLES RIVER MEDIA, INC., a part of Cengage Learning.

Original edition published by Cengage Learning. All Rights reserved.

China Machine Press is authorized by Cengage 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.

Cengage Learning Asia Pte. Ltd.

5 Shenton Way, # 01-01 UIC Building, Singapore 068808

本书原版由圣智学习出版公司出版。版权所有, 盗印必究。

本书中文简体字翻译版由圣智学习出版公司授权机械工业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)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。

本书封面贴有 Cengage Learning 防伪标签, 无标签者不得销售。

封底无防伪标均为盗版

版权所有, 侵权必究

本书法律顾问 北京市展达律师事务所

本书版权登记号: 图字: 01-2012-7564

### 图书在版编目(CIP)数据

Lua 游戏开发实践指南/(美)斯库特玛(Schuytema, P.), (美)马尼恩(Manynen, M.)著; 田剑译. —北京: 机械工业出版社, 2013.1

(华章程序员书库)

书名原文: Game Development with Lua

ISBN 978-7-111-40335-7

I. L… II. 斯… ②马… ③田… III. 游戏程序—程序设计—指南 IV. TP311.5

中国版本图书馆 CIP 数据核字 (2012) 第 265407 号

机械工业出版社 (北京市西城区百万庄大街 22 号 邮政编码 100037)

责任编辑: 马超

北京市荣盛彩色印刷有限公司印刷

2013 年 1 月第 1 版第 1 次印刷

186mm × 240mm · 16.25 印张

标准书号: ISBN 978-7-111-40335-7

ISBN 978-7-89433-699-6 (光盘)

定价: 59.00 元 (附光盘)

凡购本书, 如有缺页、倒页、脱页, 由本社发行部调换

客服热线: (010) 88378991; 88361066

购书热线: (010) 68326294; 88379649; 68995259

投稿热线: (010) 88379604

读者信箱: hzsj@hzbook.com

# 译者序

在程序开发领域里，各种编程语言层出不穷，在不同的场合选择合适的工具才能事半功倍。对于现在的游戏开发来说，已经不是那种小作坊的工作模式，也不是从零开始一步步打造自己作品的时代了。目前市场上的大型游戏往往需要数十人甚至上百人的团队至少经过数月甚至几年的艰苦开发才能完成。这样的开发规模没有合理的团队分工协作、科学的工程管理和强有力的开发工具是难以成功的。

游戏开发是一个创意性的工作，需要通过快速开发原型、测试和修改来验证游戏性。因此，需要一个具有良好兼容性、简单而高效的编程语言来帮助游戏设计师完成他们的工作。本书介绍的 Lua 就是这样一种语言，它借助 C/C++ 等底层语言可以无限扩展，而脚本语言的特性又让它十分适合快速原型开发和迭代。近年来，许多大型游戏都采用了 Lua 作为自己的嵌入式脚本语言，以此来实现可配置性和可扩展性。

本书的作者 Paul Schuytema 和 Mark Manyen 作为从事游戏行业多年的技术专家，由浅入深、循序渐进地为大家展示了如何使用 Lua 开发令人激动的游戏。本书从最简单的 Hello World 到复杂的人工智能和路径搜索，使用了大量的例子为初学者详细解释了 Lua 语言的方方面面，并带领大家从游戏设计开始逐步实现游戏的快速原型，展示了完整的游戏开发流程。

本书的翻译经历了三个多月时间，非常感谢华章公司的编辑们在翻译过程中的支持和帮助。由于译者的水平和时间有限，错误和不当之处在所难免，敬请广大读者批评指正。

# 序

脚本的概念在程序中十分重要，在游戏开发领域，它更是决定性的。脚本语言让程序员可以区分游戏开发的“硬核部分”和“软体部分”。“硬核部分”一般对计算性能要求很高，在开发过程中变更较少、重用性很高。图形引擎和人工智能模块是其中的代表。这些模块最适合使用 C 或者 C++ 这样的语言开发，可以提供更好的性能。“软体部分”控制“硬核部分”来创建最后的图形和大量的物体。这个部分更适合使用 Lua 这样的脚本语言开发，可以为程序员在尝试、测试和改变游戏代码上提供更多的灵活性。

Lua 作为脚本语言的诞生是因为两个特别的行业需求，它们都和游戏相关。应用程序需要灵活的数据描述语言和简单的行为描述脚本语言（例如，数据验证），这些就是 Lua 最初的目标：可移植性、小巧、无限制。它必须是可移植的，因为目标客户的计算机是多样化的（MS-DOS、Windows 3.0、IBM AIX 及许多其他平台）。它还必须小巧，不能让程序因为使用它而变得庞大，因为一些目标机器空间是很有限的。同时，以往的经验告诉我们，编程语言不能限制程序员，因为在开发程序时，往往会出现我们意想不到的需求和用法。因此，从第一个版本开始，Lua 就因为它自身的简单性而表现出了优异的性能。后来，因为这种简单性被证明是十分有价值的，所以我们将它加到了 Lua 的目标中。

Lua 在 Tecgraf 一举成功之后，用在了许多其他项目中，因此我们向全世界开放了它。这是个成功的决定，Lua 的免费公开，让我们获得了国际化的顾问小组，语言本身也从获得了长足的发展。

1996 年年末，卢卡斯艺术公司的 Bret Mogilefsky 在 Dobb 博士的网站上读到一篇关于 Lua 的论文后，决定在他开发的游戏采用它。几个月后，他在游戏开发大会中公开了 Lua 相关的信息。不久以后，许多游戏公司就开始使用 Lua 了。Lua 使用率的快速增加令我们十分惊讶，因为我们从来没想到它能成为游戏开发的一种语言。它曾经在很多图形相关的项目中使用过，但是从未染指游戏领域。不过现在看来，Lua 进入游戏开发领域也是合情合理的。

Lua 所有的优点对于游戏开发都是很重要的：简单性、可移植性以及运行效率高，从

此以后，我们开始关注游戏开发者，陆续为 Lua 增加了许多新的特性，如为游戏开发提供了协同例程（coroutine）的功能。尽管 Lua 的功能不会局限在游戏开发上，但是在 Lua 未来的发展中，必着重考虑游戏开发。

非常欢迎本书成为 Lua 系列书籍的新成员。本书重点关注游戏开发，强烈推荐给有志于开发专业游戏的读者。我们期待本书能在这个不断发展的领域中更好地传播 Lua 语言。

——Lua 语言创始人之一 Roberto Ierusalimsky





# 前言

## Lua 游戏开发

游戏开发是一个激动人心的过程，创造出让玩家花费数小时并乐在其中的游戏，给人带来的成就是任何事情都无法比拟的。然而，这个创造的过程正在变得越来越难。那种奋战几个晚上或者几周就能单枪匹马设计出热门游戏的日子已经一去不复返，现在的游戏往往需要数十人的开发团队工作很多个月甚至几年才能完成。就算是那些可以从网上下载的最简单的“休闲游戏”也通常是由专业开发者组成的团队开发数月的成果。

尽管游戏开发的规模不断增大，但始终有一个不变的追求——测试、更新、调整，以及快速验证游戏性的能力，通常这个部分是设计和开发过程的核心。采用一种像 Lua 这样的脚本语言以及内核级别的语言（如 C++）可以帮助用户开发专业的游戏，并且还能让开发者和设计师快速实现设计想法、测试游戏功能。

## 适用读者

本书适合三类读者：

**游戏程序员。**程序员在开发团队中负责实现 Lua 和 C++ 之间的接口，并且通常还要编写部分或者全部的游戏脚本。本书将告诉程序员如何将 Lua 和 LuaGlue 的功能集成到游戏开发项目中。对于程序员来说，最重要的是，使用 Lua 可以在游戏开发的过程中节省很多时间和精力，因为许多游戏的功能可以由设计师和脚本程序员来实现。

**游戏设计师。**通常，游戏设计师会采用像 Lua 这样的脚本语言在运行环境中来实现部分设计。本书可以作为 Lua 语言的初级读本，为设计师打下坚实的技能基础，从而去构建真实的游戏世界。同时，本书还可以激发设计师的灵感，使用 Lua 开发可以帮助他们使用工具快速开发原型，快速实现并且进行创意的验证。

**业余游戏开发者。**学习如何开发你自己的游戏是富有成就感和具有挑战性的。游戏行业鼓舞了许多这样的业余爱好者，通过自己的项目来学习更多关于游戏和开发的知识，这样的付出很值得。本书展示了经验丰富的业余游戏开发者如何在他们的项目中使用 Lua，还提供一个已有的框架以便入门并深入学习 Lua，进而快速开发没有任何 C++ 代码的游戏

(提供完整的控制台和游戏测试环境)。

## 本书主要内容

在本书中，有一篇关于 Lua 的简介，包括历史背景和脚本编程两方面。此外，读者还将学会如何链接 Lua API 来扩展 C++ 功能。

建立了一定的知识基础后，本书将带领读者使用 Lua 脚本语言开发一个游戏的“快速原型”。这个游戏会为读者展示使用 C++ 功能和 Lua 脚本的方法，例如：

- 存储和载入游戏数据
- 创建模块化的、灵活的 GUI 系统
- 用 Lua 脚本管理游戏的实时事件
- 使用 Lua 定义和控制游戏的 AI (人工智能)

## 系统要求

- P450 或者更好的处理器
- Windows 2000/XP
- 32MB RAM
- 演示程序要求：
  - DirectX 9 (包含在 CD-ROM 中)
  - DX 兼容的 3D 视频加速器
- DirectX SDK:
  - 操作系统: Microsoft Windows (r) 98, Windows Millennium Edition (Windows Me), 或者 Windows 2000, Windows Server 2003, Windows XP
  - 约 65MB 可用硬盘空间 (安装完成后可以删除安装文件。剩下的 DirectX 文件约占 18MB 硬盘空间, 如果安装了更早版本的 DirectX, 可能使用空间会有所不同。DirectX 9.0 会覆盖之前的版本。)
- Lua:
  - 兼容大部分可以编译 C 语言的系统
- Ogg Vorbis:
  - Microsoft Windows 95, 98, Me, NT, 2000 或者 XP
  - 声卡
  - 处理器: Pentium 200MHz 以上

## 致 谢

没有 Nick Carlson 提供的优秀脚本，这本书很难顺利完成，他和我一起完成了书中的例子和大量脚本，他才刚刚开始大学生涯，未来一定前途无量。同样感谢 Chris Listello 为《Take Away》的美术设计和封面设计所做的工作。还要感谢 Roberto Ierusakimschy 为本书作序。感谢所有 Lua Tecgraf 小组的成员，为我们创造了这样一款功能强大且性能优越的脚本语言。最后感谢 Charles River 小组：感谢 Jenifer Niles 的支持，感谢技术编辑们帮助我们修改了文字，感谢所有制作小组的成员，你们的帮助成就了这个令人自豪的项目。



# 目 录

译者序

序

前言

致谢

## 第1章 游戏开发入门 ..... 1

1.1 越来越复杂的开发过程 ..... 1

1.2 更好的开发方式 ..... 2

1.3 为什么使用 Lua ..... 3

1.4 本章小结 ..... 4

## 第2章 脚本语言 ..... 5

2.1 脚本语言简介 ..... 5

2.2 Lua 简介 ..... 6

2.2.1 Lua 的历史 ..... 7

2.2.2 Lua 授权 ..... 7

2.3 本章小结 ..... 8

## 第3章 游戏开发世界的 Lua 语言 ..... 10

3.1 脚本语言和游戏 ..... 10

3.2 游戏项目中的 Lua ..... 11

3.2.1 游戏界面 ..... 11

3.2.2 管理游戏数据 ..... 12

3.2.3 事件处理 ..... 14

3.2.4 保存和读取游戏状态 ..... 14

3.2.5 人工智能 ..... 15

3.2.6 快速构建原型 ..... 16

3.3 本章小结 ..... 16

## 第4章 Lua 入门 ..... 17

4.1 使用 Lua 控制台 ..... 17

4.2 Lua 基础 ..... 19

4.3 变量 ..... 21

4.3.1 nil ..... 21

4.3.2 Boolean ..... 21

4.3.3 string ..... 22

4.3.4 Number ..... 22

4.3.5 table ..... 23

4.3.6 局部变量和全局变量 ..... 23

4.4 运算符 ..... 24

4.4.1 算术运算符 ..... 24

4.4.2 关系运算符 ..... 24

4.4.3 逻辑运算符 ..... 25

4.5 控制结构 ..... 26

4.5.1 if ..... 27

4.5.2 while 和 repeat ..... 27

4.5.3 for ..... 28

4.5.4 break ..... 29

4.6 本章小结 ..... 29

## 第5章 深入学习 Lua ..... 30

5.1 函数 ..... 30

5.1.1 单一参数 ..... 31

5.1.2 多个参数 ..... 31

5.1.3 返回值 ..... 32

5.2 标准库 ..... 34

5.2.1 assert(myValue)() ..... 34

5.2.2	dofile(filename) .....	35	6.2.3	命令处理 .....	51
5.2.3	math.floor() .....	36	6.2.4	退出程序 .....	52
5.2.4	math.random() .....	36	6.2.5	cLua 对象和 LuaLib .....	52
5.2.5	math.min() .....	37	6.2.6	使用 cLua 的例子 .....	53
5.3	字符处理 .....	38	6.2.7	LuaGlue 函数的优点 .....	55
5.3.1	类型转换 .....	38	6.2.8	LuaGlue 函数: 参数和 返回值 .....	55
5.3.2	string.char(n1,n2,...) ...	38	6.3	本章小结 .....	56
5.3.3	string.len(myString) .....	38	第7章	Lua 与 C++ 的交互 .....	57
5.3.4	string.sub(myString, start,end) .....	39	7.1	重新审视 LuaGlue 函数 .....	57
5.3.5	string.format() .....	39	7.2	C++ 代码和 Lua 的交互 .....	58
5.3.6	string.find(sourceString, findString) .....	40	7.3	事件驱动的编程 .....	58
5.3.7	字符和格式 .....	40	7.3.1	示例事件 .....	58
5.4	table 数据结构 .....	42	7.3.2	事件的参数 .....	59
5.4.1	table.getn(myTable) .....	43	7.4	错误处理 .....	60
5.4.2	table.insert(myTable, position,value) .....	43	7.5	本章小结 .....	61
5.4.3	table.remove(myTable, position) .....	44	第8章	开发准备 .....	62
5.4.4	table 引用 .....	44	8.1	Visual C++ 6.0 工作区 .....	62
5.4.5	多维 table .....	44	8.2	DirectX 基础 .....	63
5.4.6	pairs() .....	45	8.3	LuaGUI 简介 .....	65
5.5	I/O 基础 .....	46	8.3.1	启动 GUI .....	66
5.6	本章小结 .....	47	8.3.2	界面 .....	66
第6章	Lua 与 C/C++ 程序的 整合 .....	48	8.3.3	界面控件 .....	66
6.1	初期设计要点 .....	48	8.3.4	事件 .....	67
6.1.1	Lua 环境 .....	48	8.3.5	与 GUI 系统相关的 LuaGlue 函数 .....	67
6.1.2	LuaGlue 函数 .....	49	8.3.6	Shell 程序的扩展 .....	68
6.2	基本实现方式 .....	49	8.4	调试窗口 .....	69
6.2.1	创建 Lua 运行环境 .....	50	8.5	Windows 注册表 .....	69
6.2.2	添加 LuaGlue 函数 .....	51	8.6	本章小结 .....	70
			第9章	设计 Lua 版本的实现 .....	71
			9.1	游戏设计原则 .....	71
			9.1.1	什么是游戏 .....	71

9.1.2 了解玩家的想法 .....	72	11.5.4 主菜单界面 .....	125
9.2 基础库设定 .....	73	11.5.5 Controls 界面 .....	130
9.3 设计文档 .....	78	11.5.6 InGame 界面 .....	132
9.4 Lua 编程规范 .....	81	11.6 本章小结 .....	135
9.5 本章小结 .....	83	<b>第12章 Lua 游戏编程</b> .....	136
<b>第10章 使用 Lua 处理游戏数据</b> .....	84	12.1 游戏主循环 .....	136
10.1 简单的游戏数据 .....	84	12.2 井字棋 .....	137
10.1.1 太空飞船的例子 .....	85	12.2.1 游戏的初始化 .....	138
10.1.2 《Take Away》的玩家 飞船 .....	88	12.2.2 游戏回合处理 .....	139
10.1.3 敌舰数据 .....	89	12.2.3 模拟游戏回合 .....	147
10.1.4 补给箱数据 .....	91	12.3 《Take Away》游戏的实现原理 .....	147
10.2 大数据集 .....	92	12.3.1 InGame .....	147
10.2.1 表单型数据 .....	93	12.3.2 使用计时器 .....	152
10.2.2 Lua 格式的数据文件 .....	95	12.3.3 玩家操作 .....	154
10.3 使用 Lua 保存游戏数据 .....	96	12.3.4 子弹运动 .....	156
10.3.1 案例1——《Frontrunner》 .....	106	12.3.5 飞船移动 .....	158
10.3.2 案例2——健身大亨 .....	107	12.3.6 绘制活动的物体 .....	161
10.4 本章小结 .....	108	12.4 本章小结 .....	163
<b>第11章 Lua 驱动的 GUI</b> .....	110	<b>第13章 使用 Lua 定义和 控制 AI</b> .....	164
11.1 GUI 系统概要 .....	110	13.1 智能的体现 .....	164
11.2 GUI 的 C++ 类 .....	111	13.2 21 点游戏 .....	165
11.2.1 GUI 控件: Sprite .....	112	13.3 井字棋 .....	170
11.2.2 GUI 控件: TextField .....	113	13.4 《Take Away》游戏的实现 .....	175
11.2.3 GUI 控件: Button .....	113	13.4.1 掠夺舰 .....	175
11.2.4 界面 .....	114	13.4.2 攻击舰 .....	176
11.2.5 GUI 管理器 .....	115	13.4.3 冲击舰 .....	176
11.3 GUI LuaGlue 函数 .....	116	13.4.4 混合舰 .....	177
11.4 进一步的说明 .....	118	13.4.5 控制飞行方向 .....	178
11.5 Lua 游戏界面 .....	119	13.4.6 碰撞检测 .....	179
11.5.1 界面设计原则 .....	119	13.5 其他 AI 的例子 .....	183
11.5.2 快速创建界面 .....	120	13.5.1 静态追踪 .....	183
11.5.3 载入界面 .....	121	13.5.2 近距离追踪 .....	185

9.1.2 了解玩家的想法 .....	72	11.5.4 主菜单界面 .....	125
9.2 基础库设定 .....	73	11.5.5 Controls 界面 .....	130
9.3 设计文档 .....	78	11.5.6 InGame 界面 .....	132
9.4 Lua 编程规范 .....	81	11.6 本章小结 .....	135
9.5 本章小结 .....	83	<b>第12章 Lua 游戏编程</b> .....	136
<b>第10章 使用 Lua 处理游戏数据</b> .....	84	12.1 游戏主循环 .....	136
10.1 简单的游戏数据 .....	84	12.2 井字棋 .....	137
10.1.1 太空飞船的例子 .....	85	12.2.1 游戏的初始化 .....	138
10.1.2 《Take Away》的玩家 飞船 .....	88	12.2.2 游戏回合处理 .....	139
10.1.3 敌舰数据 .....	89	12.2.3 模拟游戏回合 .....	147
10.1.4 补给箱数据 .....	91	12.3 《Take Away》游戏的实现原理 .....	147
10.2 大数据集 .....	92	12.3.1 InGame .....	147
10.2.1 表单型数据 .....	93	12.3.2 使用计时器 .....	152
10.2.2 Lua 格式的数据文件 .....	95	12.3.3 玩家操作 .....	154
10.3 使用 Lua 保存游戏数据 .....	96	12.3.4 子弹运动 .....	156
10.3.1 案例1——《Frontrunner》 .....	106	12.3.5 飞船移动 .....	158
10.3.2 案例2——健身大亨 .....	107	12.3.6 绘制活动的物体 .....	161
10.4 本章小结 .....	108	12.4 本章小结 .....	163
<b>第11章 Lua 驱动的 GUI</b> .....	110	<b>第13章 使用 Lua 定义和 控制 AI</b> .....	164
11.1 GUI 系统概要 .....	110	13.1 智能的体现 .....	164
11.2 GUI 的 C++ 类 .....	111	13.2 21 点游戏 .....	165
11.2.1 GUI 控件: Sprite .....	112	13.3 井字棋 .....	170
11.2.2 GUI 控件: TextField .....	113	13.4 《Take Away》游戏的实现 .....	175
11.2.3 GUI 控件: Button .....	113	13.4.1 掠夺舰 .....	175
11.2.4 界面 .....	114	13.4.2 攻击舰 .....	176
11.2.5 GUI 管理器 .....	115	13.4.3 冲击舰 .....	176
11.3 GUI LuaGlue 函数 .....	116	13.4.4 混合舰 .....	177
11.4 进一步的说明 .....	118	13.4.5 控制飞行方向 .....	178
11.5 Lua 游戏界面 .....	119	13.4.6 碰撞检测 .....	179
11.5.1 界面设计原则 .....	119	13.5 其他 AI 的例子 .....	183
11.5.2 快速创建界面 .....	120	13.5.1 静态追踪 .....	183
11.5.3 载入界面 .....	121	13.5.2 近距离追踪 .....	185

## ■ 为什么选择 Lua

如果你是一名游戏爱好者，一定体验过第一次玩自己的游戏时的那种快乐，还有当看到伙伴玩你的游戏时，发自内心的喜悦和兴奋的那种满足感。

这种感觉对于资深的游戏开发者来说也是一样，我们同样在乎我们开发的游戏，没有什么比看到别人快乐地游戏更让人感到激动的了，因为其中饱含着爱、血汗、眼泪和真心付出。

许多年前，大部分游戏是开发者在车库和地下室、利用周末或业余时间开发的。现在若制作能够在当地电子市场售卖的游戏，则需要许多专业的开发者分工协作。

复杂度逐渐增长导致了专业的分工。游戏美术设计人员负责制作 2D 或 3D 动画以及静态模型，程序员实现网络编程、人工智能（AI）和 3D 渲染。在这种专业的分工下，想要保持过去那种灵活并富有创造性的游戏开发过程越来越难。

开发团队规模的不断增长,以及游戏复杂度的不断提高,使得不同游戏系统之间的依赖性也在提高。这些依赖性则直接导致了开发周期延长,游戏设计想法无法验证,创新和游戏灵感也不得不由于开发周期紧张而有所限制。

若干年前，我得到一个机会拜访了一家知名的游戏工作室，他们当时正在开发一款第三人称冒险射击类游戏。这个游戏看起来很“酷”，3D 场景绚丽多彩，与环境的交互也显



得十分真实。

我有幸看到了游戏制作的整个过程。首先，3D 美术设计师用建模软件创建了一个游戏场景，把这些模型导入一个内部工具中，让设计师设置各种触发区域，当玩家角色或者 AI 控制的敌人进入该区域时触发特定的游戏事件。然后，设计师会坐下来和程序员交流每一个触发区域，告诉他们期待发生的结果，程序员做好各种笔记，接着花许多天时间来实现这些代码。完成之后，设计师会检验成果，并提出一些修改意见，然后整个过程再不断重复。

尽管结果是可靠的，但可以想象这个过程非常艰苦，不仅耗时而且呆板。我知道一定有更好的方式，不过在那个时候我对脚本语言还一无所知。

## 1.2 更好的开发方式

更好的方式是使用中层脚本语言来构建项目，它可以帮助游戏设计师把握整个开发的交互过程，让程序员去做大量更基础的工作。

从游戏开发者的角度看，脚本语言可以帮助用户很容易地返回游戏开发过程。也许需要几个小时来构建一个“干净”的游戏项目，但脚本语言可以帮助用户快速做出修改并且立刻看到游戏的效果。游戏设计师可以独立于程序员尝试新想法，游戏美术设计师可以创建图形界面把游戏流程和功能组合到一起。

脚本语言存在于由软件工程师编写并编译后的代码之上，通常是在运行时编译，是一种方便设计师或者程序员处理和控制的简单语言。

为什么要使用脚本语言呢？对于资深从业人员或者业余开发者来说，这都是一个值得关注的问题。从游戏设计师的角度来说，使用脚本语言开发游戏可以很清楚地界定底层代码和游戏玩法代码。通常，在引入了脚本语言的项目中，底层模块交给像 C++ 这样的核心语言，诸如界面交互、数据管理、人工智能和事件处理等，一般使用脚本语言实现。这种职责的划分可以让用户的游戏更加稳定，并且使得并行开发成为可能。

脚本语言还能使开发团队中的非技术成员参与到核心开发过程中。界面美术师不仅只制作界面素材，还能独立于程序员编写让界面运行在游戏中的脚本框架。要实现这些想法，设计师则可以不麻烦程序员，而直接着手 AI、数据处理或者创建场景脚本。

相对于 C++ 这样的底层语言，脚本语言本身是易学易用的。由于 Lua 语言无须关心复

杂的内存管理、对象渲染或者 TCP/IP 网络通信，所以十分容易上手并投入实际开发。学习脚本语言不需要花费很长时间，开发者可以在几个小时内就掌握它的语法和功能。

在本书中，我们将自下而上地学习如何使用 Lua 语言并利用 C++ 的功能来创建一个完整的游戏。在这个过程中，会特别向读者展示脚本语言是如何优化开发过程的，不管是资深开发者还是业余爱好者，都将会有所收获。

脚本语言有很多，那我们为什么要选择 Lua？纵观整个脚本语言领域，我们可以看到有 Perl、Tcl、Ruby、Forth、Python、Java 和 Lua。尽管所有的脚本语言在特定领域都有自己的一席之地，但在游戏开发的世界中，Python 和 Lua 是非常适合的（因为它们可以直接调用 C++ 的功能）。

许多商业游戏已经成功地使用了 Python 和 Lua，因为它们都有很强的兼容性，所以可以与编译后且基于 C++ 技术的模块协同工作，而且还能扩展。如果读者有机会询问一下程序员对于这两种语言的看法，通常会有非常不同的观点，还可能有激烈的争论，就像体育中的“德比”赛事那样（想象一下芝加哥小熊对白袜，纽约喷气机对巨人，纽约大都会对洋基）。其实，这两种语言都是游戏开发领域中非常出色的工具。

### 1.3 为什么使用 Lua

对于游戏开发而言，Lua 是较好的选择，其设计的核心目标是可扩展性，因此在最初设计时就考虑到要能够集成在大型应用中。因为有了这样的设计目标，所以非常容易在应用程序中加入 Lua 脚本。Lua 的易集成的特性还使得 Lua 可以很方便地与父程序通信。游戏程序员都希望脚本语言能够简单地实现游戏设计，在这方面，Lua 也能够胜任。

Lua 免费、小巧、快速且易移植。所有的游戏开发者和游戏公司都喜欢“免费”的工具。通常讲，一分钱一分货，但是对于 Lua 来说，它完全超出你的预期。Lua 采用了非常灵活的发布协议，它有极少的源代码，运行轨迹十分紧凑，在编译时间和运行时内存占用上都有很好的性能表现。

要说 Lua 最让人惊喜的地方，应该是它的执行速度。对于任何脚本语言的技术方案，游戏开发者的第一反应就是：“脚本语言太慢了，帧率一定不会很理想。”但这个说法对 Lua 是不成立的，事实上，我们还没有看到任何一个项目因为 Lua 的使用而造成瓶颈。最后，游戏开发界正在迎接一次新的硬件周期，我们将要学习如何使用一组新的平台。因为

Lua 的易移植性，当我们的技术储备转移到新的平台时，至少有一部分是不会过时的。

Lua 是非常容易学习的语言。不需要了解很高级的编程概念（如对象和继承），大部分具有计算机学习背景的人都可以在短期内掌握它的基础知识并且马上投入正式的工作。如果团队成员熟悉其他的语言，那么 Lua 可以轻松上手，很适合那些非程序员背景的团队人员，它们也能对游戏功能和美术部分进行修改或创建。

在我们公司，最近刚刚发布了第 13 款使用 Lua 开发的游戏。我们的团队虽然很小，但是也有程序员、美术师和设计师的标准构成。当我们开始一个新的项目时，首先要确定项目的技术需求（什么是我们还没有而需要去实现的？）和设计需要的功能。程序员可以负责技术设计部分，专注于那些他们擅长的技术难题。同时，设计师和美术师可以马上开始着手界面流程和核心游戏功能的设计工作。通常，美术师（包括 2D 和 3D）还会花一部分时间确定游戏的视觉需求。与此同时，3 位熟悉 Lua 的设计师开始构建游戏基础、游戏数据及核心游戏系统。他们甚至都不需要等着程序员，如果需要什么功能一般就直接先用 Lua 代替。这样我们就可以进行非常高效率的游戏开发，因为团队的每个人都能从一开始就“热火朝天”干起来。

有一个项目要特别提起，在我们为 2004 年美国大选开发选举游戏模拟器的時候，我们已经可以用 Lua 开发出完整的游戏原型来验证 AI 和游戏流程，然后再回过头来重新用 C++ 实现那些核心的部分。快速的原型开发可以让一个开发者就能够完成设计和开发环节最有价值的部分，这也是业界少有的高效率。

## 1.4 本章小结

游戏开发是一个令人兴奋的产业，人们一旦有了灵感就想马上开始行动。Lua 赋予了我们这样的能力，让我们可以快速实现游戏核心概念，设计并测试界面，并且方便地管理大量的运行时数据，而这一切都不需要读者有很深厚的技术背景。

在以后的章节中，我们将要学习如何在游戏项目中使用 Lua，并学习语言本身的特性。然后，我们将开发一个完整的游戏，一起经历从开始到完工的整个过程。完成了这些，我们就可以很好地掌握这种灵活、小巧的开发语言，并切身感受到它给你的游戏和游戏开发体验所带来的不同。

# 脚本语言

## 本章要点

### ■ 脚本语言简介

### ■ Lua 简介

虽然计算机可以做很多事情，从生成报表到模拟经营属于你自己的主题公园，但是计算机自己不会思考，它需要接受系统化的指令来工作。大部分用户通过像 Word 文字处理器或者电子表格工具这样的应用程序来为计算机指派任务。软件工程师则使用像 C++ 这样的底层编程语言让计算机工作（通常为普通用户开发应用程序）。而我们所说的脚本语言，存在于操作便捷的应用程序和开发软件的底层编程语言这二者之间。

## 2.1 脚本语言简介

脚本语言可以方便地与计算机底层功能交互。这体现在它常常被当做批处理命令工具，即发送一系列重复的指令给命令处理器的工具。所以早期的脚本语言常常称做批处理语言或者作业控制语言。

一个熟悉的例子就是 MS-DOS 时期的老的 \*.bat 文件，这种批处理文件就是简单的文本文件，它包含一系列顺序执行的 DOS 命令。该语言本身就是 DOS 命令集合，通过进一步扩展成为一种伪脚本（参考下面的示例）。

```
copy g:\whitehouserun\working\wnr.exe  
copy g:\whitehouserun\working\wnrd.exe  
ren *.txt *.lua  
copy *.lua g:\luabank\whitehouserun
```

计算机语言用于解决一些特定的问题，从系统控制级别的 C 和 C++ 到人工智能处理语言（如 LISP）。脚本语言通常拥有一些共同点，他们一般用在快速开发中（低成本、高效率），并采用接近自然语言的语法，对于非程序员背景的人更易于书写和阅读，这样有

一定基础的用户就可以在没有程序员的帮助下编写和使用脚本语言。脚本语言在调用其他底层语言开发的模块方面十分出色。

脚本文件都是在载入时解释和编译（不是预编译，而是在调用时才处理）。以 Lua 为例，它只有在载入时才被编译成二进制形式并存在于内存中，直到被释放。

在软件开发（特别是游戏开发）领域，结合使用脚本语言和底层语言可以让开发者更好地控制运行环境，使得在开发过程中，在运行环境上的修改和测试都拥有更大的灵活性。

## 2.2 Lua 简介

Lua 和传统的脚本语言不同，它是一种易整合语言（glue language）。一般的脚本语言用于控制执行重复的任务，而易整合语言可以让使用者把其他语言开发的功能整合在一起。这样就让脚本程序员有了更大的发挥空间，而不仅仅局限于执行命令。程序员可以使用这种脚本在底层语言开发的功能模块基础上创建新的命令。本书将探讨如何使用 Lua 来整合 C++ 的与游戏相关的一些功能，如 GUI、AI、数据等。

Lua 本身是一种简单而又强大的编程语言，它可以让脚本程序员完成大量的处理。这种语言拥有很强大的字符处理和数学运算能力、灵活的数据类型（很快就可以熟悉），以及定义函数的功能。但是如果没有整合其他环境的组件的“魔力”，这些基础的特性也就丧失了。（没错，你可以在命令行下执行 Lua 脚本并查看运行结果，但除了学习语言本身，对于游戏开发来说，命令行式的输出是没有实际意义的。）学习其他编程语言经典的第一课是如何输出“hello world”，Lua 版本的方法参见代码清单 2.1。

代码清单 2.1 用 Lua 编写的“hello world”程序

---

```
--Lua's 'hello world'
myString = 'hello world'
print(myString)
```

---

Lua 非常适合作为更强大的底层编程语言的搭档，如 C++。Lua 能让游戏开发者快速建立游戏原型甚至是完整的游戏。游戏开发者可以在没有程序员帮忙的情况下构建整个图形界面。它还可以用来管理游戏进度文件的保存和载入，而且很容易阅读和调试。在游戏开发领域，Lua 能帮助开发者构建一个高效并且方便验证游戏想法的环境。

按照开发 Lua 的团队的描述, Lua 是一个可以集成在应用程序中的“语言引擎”。它本身是一种编程语言, 并且还提供了很多可以和应用程序交换数据的 API (应用编程接口)。另外, Lua 还能够通过整合 C++ 的模块来进行功能的扩展(这个就是我们之前所说的“整合”功能)。和程序开发语言(如 C++) 配合使用时, Lua 也可以用来作为特定项目的框架语言。这种易扩展性使 Lua 非常适合作为游戏开发的环境。

作为独立的编程语言(在运行窗口中执行), Lua 功能很有限, 只能用做教学工具。(我们会在接下来的章节中使用控制台学习该语言。) Lua 只有集成在其他语言中才能发挥它的价值。它的实现非常简单, 仅仅通过一些 LuaGlue 函数就可以和底层语言通信, 在用户自定义 LuaGlue 函数的基础上, 它还可以进一步被扩展, 甚至成为一种新的编程语言。

### 2.2.1 Lua 的历史

Lua 在葡萄牙语中是“月亮”的意思, 1993 年由巴西的 Pontifical Catholic University 开发。该语言是由一个来自计算机图形技术组织(Tecgraf)的团队(Roberto Ierusalimsky、Waldemar Celes 和 Luiz Henrique de Figueiredo)开发, 并作为自由软件发行。Lua 开发小组的目标是开发一种小巧、高效并且能够很好地和 C 语言一起工作的编程语言。在脚本语言领域, Lua 是最快、最高效的脚本语言之一, 因此它有资格作为游戏开发的备选方案。Lua 的内核小于 120KB (Python 的内核大约 860KB, Perl 的内核大约 1.1MB), 当编译和集成到游戏开发系统中时非常小巧。Lua 通常比 Python 这种流行的游戏开发脚本语言运行更快速, 完整的性能测试报告可以在计算机编程语言实战性能测试网站中找到 (<http://shootout.alioth.debian.org/>)。

计算机图形技术组织(Tecgraf)成立于 1987 年, 致力于开发和维护用于技术和科技领域的计算机图形和用户界面。除了 Lua, Tecgraf 小组还开发了 IUP (一种开发用户界面的系统)、CanvasDraw (跨平台的图形库)、TWF (一种用于 Web 页面的图形文件格式) 和其他一些系统。读者可以访问相关网站获取更多信息。

### 2.2.2 Lua 授权

Lua 是免费的开源软件, 可以免费用于科研及商业应用。关于开源软件的更多信息可访问 [www.opensource.org](http://www.opensource.org)。

对于游戏开发专业人员，授权费对于开发技术的选择影响很大。通常，对于一个项目，游戏引擎的预算会超过 50 万美元，中间件技术会花费 5 千美元 ~ 5 万美元不等。因此，开源软件对于开发团队来说是很具有吸引力的。

开源软件因为本身不盈利，所以它们的代码通常有很多 bug，又由于没有太多注释，因此难以理解。另外，没人为技术支持付钱，所以也谈不上什么技术支持。

Lua 则避免了这些问题，它小巧，实现简单（而且还在维护），代码简洁、清晰。Lua 的开发团队是由具有计算机工程背景的专家组成，并且一直在关注着它的升级。和其他开源项目不同，设计 Lua 旨在项目中扩展功能，而不是在 API 级别，因此它的内核一直很稳定。

Lua 授权的精神在于用户可以在任何情况下免费使用，并且不需要取得版权所有者的许可。如果用户想知道更多 Lua 授权的信息，那么可以访问 [www.lua.org/license.html](http://www.lua.org/license.html)，

完整的 Lua 5.0 授权如下（CD-ROM 中也提供了该授权）：

Lua 5.0 License

Copyright ©2003-2004 Tecgraf, PUC-Rio.

*Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:*

*The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.*

*THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.*

## 2.3 本章小结

脚本语言最初只是一种简单的工具，目的是为了资深用户在简单文本文件中能够批



量执行常用命令。现在许多脚本语言，如 Python、Ruby 和 Lua，可以提供像计算机编程语言一样强大而灵活的功能。最近几年，脚本语言已经“走”到了游戏开发业界的前端，作为一个可行的中间件产品，它可以有效地提升开发小组的工作效率以及游戏项目的性能。来自巴西的计算机图形技术组织（Tecgraf）的团队开发的 Lua，由于它的小巧、高速以及与 C 和 C++ 良好的兼容性，成为非常适合游戏开发的工具。Lua 是免费的开源语言并且有良好的技术支持，再加上不断增长的忠实用户群，让它成为专业开发者及业余游戏开发爱好者的不错选择。





## 第3章

游戏开发世界的 Lua 语言

# 游戏开发世界的 Lua 语言

### 本章要点

- 脚本语言和游戏
- 游戏项目中的 Lua

现实中的游戏开发常常面临两种互相矛盾的压力，一方面需要测试和验证新想法，另一方面又需要快速开发并且按时交付。如果适当地在项目中引入脚本语言，可以让资深工程师充分发挥他们的能力，开发出优秀的游戏系统和模块。

## 3.1 脚本语言和游戏

脚本语言可以让美术师直接开始界面设计，让设计师和初级程序员（脚本语言是一种让新手快速进入游戏开发的很好的方式）立即着手游戏流程和逻辑的开发，让关卡设计师能迅速掌控游戏环境和游戏体验。

脚本语言不是非常高效——它们没有原生代码的运行效率，因此不适合作为开发高性能需求处理的工具。但易整合语言能够利用原生语言编写的模块扩展功能，比如 Lua，可以作为控制机制来调用原生代码编写的高性能处理组件。（Lua 是运行效率最高的脚本语言之一，因此大部分性能方面的问题都可以不用担心。）C 函数可以利用自己高性能的特点，并且整合到 Lua 中，让脚本程序员可以利用这些功能。

这种处理的一个例子是在游戏世界中放置一个 3D 模型的功能。渲染系统完全由 C++ 开发，但 Lua 可以调用 C++ 来创建一个特定模型的实体对象，并且设置其在场景中的位置，然后 Lua 还可以为这个 3D 模型指定动画。Lua 并不处理任何实时的复杂运算来改变该模型，而只是告诉底层渲染系统什么时候该做什么。在下面的例子中（参见代码清单 3.1），AddEnvironmentObject() 是一个 LuaGlue 函数——它可以直接调用 C++ 方法并且让 Lua 能够控制底层的 3D 渲染功能。这种函数由 C++ 程序员编写，把底层的功能提供给

脚本程序员和设计师。

代码清单 3.1 使用 Lua 脚本在场景中放置 3D 模型

```
--Lua script to add room, table, and human models
--to a 3D runtime environment
envmID = AddEnvironmentObject("casino_02.mlg",0,0,0,0,0)
tableID = AddEnvironmentObject("poker_table_normal_02_5card.mlg",-
2,0,1.5,0,0,0)
if bodyID1 ~= nil or headID1 ~= nil then
    PositionChildEntity(tableID, bodyID1, "anchor_0")
end
-- This sets the color of the Ambient light in the scene
AddLight(LIGHT_AMBIENT, 30, 30, 30)
-- This Adds a directional light source to the scene
AddLight(LIGHT_DIRECTIONAL, 1.0, -1.0, 1.0, 255, 255, 255)
-- set initial camera look at
StartCamera(0,0,.556, 0,3,-5, 0.03,1.5,3,20,.5,2,0.001)
```

## 3.2 游戏项目中的 Lua

把脚本语言集成到游戏项目中可以提升团队的开发效率，并且可以很好地扩展原生编译语言的能力。Lua 在游戏开发的许多基础领域中都表现得很出色。

在游戏开发团队中，可能有许多成员都使用 Lua 来完成他们的工作。程序员负责将 Lua 整合到游戏开发环境中，通常，他们会需要编写一些 Lua 代码。游戏设计师是脚本语言的主要使用者，因为他们和上层的游戏设计和数据直接打交道。美术师也会经常使用 Lua，进行诸如界面布局、设计和 3D 场景中各种模型的摆放之类的工作。

Lua 是非常强大的工具，可以用来完成下面这些工作：

- 编辑游戏的用户界面
- 定义、存储和管理基础游戏数据
- 管理实时游戏事件
- 创建和维护开发者友好的游戏存储和载入系统
- 编写游戏的人工智能系统
- 创建功能原型，可以之后用高性能语言移植

### 3.2.1 游戏界面

游戏界面是玩家和你的游戏进行交互的媒介。游戏界面是一个游戏最基本的部分，因

为它负责所有和玩家的交互工作。因为它的重要性，所以需要能够高品质、高性能地运行，并且能够被测试和持续改善，以最大化地提升用户体验。

Lua 可以让界面设计师迅速创建所有主要的界面元素——进行界面布局、管理用户输入并且输出游戏数据。利用游戏程序员开发的一部分核心的界面控件和控制函数，界面设计师不仅能负责界面的美术观感还能控制游戏的交互。这不仅节省了程序员的时间，而且给了美术设计师更大的创作空间，同时还能为界面设计的测试节省出许多时间。代码清单 3.2 展示了使用 Lua 创建文本 GUI 控件的方法。

代码清单 3.2 使用 Lua 创建文本 GUI 控件

---

```
--Text object example
CreateItem(200,"TextField")
SetItemPosition(200, 300, 20, 200, 28)
SetFont(200, "Arial", 16)
ItemCommand(200, "SetColor", 255,255,255,255)
ItemCommand(200, "SetString", "I am a text object")
```

---

### 3.2.2 管理游戏数据

管理游戏数据对于游戏开发者一直都是一种挑战。游戏数据定义了游戏世界中所有对象的参数和特性，如脉冲枪升级费用、气垫船的行驶速度等。对于数据密集型游戏来说，开发者常常利用电子表格工具来输入和保存数据，然后创建解析工具将其转换成游戏中可以使用的格式。这种方法一般用在数据密集型游戏中，如角色扮演类游戏（NPC 或者非游戏玩家角色的信息以表格结构形式存储）或者策略类游戏（角色单位信息保存在数据表中）。

Lua 可以让这个存储系统更为简单，它以 Lua 文件作为存储介质让程序使用相同的数据。通过创建一个简单的数据管理系统，变量和类型可以定义在 Lua 中，然后可以很容易地读取。因为不用关心整个数据处理过程，所以游戏设计师可以按照需要修改、增加和缩减游戏数据，而不需要程序员的协助。因为 Lua 语法的特性和在脚本中添加注释的能力，数据文件是易读的。如果需要转换工具，也非常容易开发，把 Lua 数据输出到 Lua 文件，然后在运行时载入。

Lua 本身并没有可以直接访问外部数据库的能力，但可以用 C++ 开发访问数据库的组件，然后再利用 LuaGlue 函数整合该组件来达到目的。

在代码清单 3.3 中，我们可以看到如何使用 Lua 示例文件来直接保存游戏数据。在这

个例子中, 通过使用 LuaGlue 函数, 向由核心代码管理的数据结构中写入数据。代码清单 3.3 的输出可参照图 3.1。

代码清单 3.3 使用 Lua 保存美国总统选举模拟游戏 (《Frontrunner》) 的数据

```
-- Auto-generated LUA campaign data file
-- File created on: 07/28/04 12:02:08
AddCampaign( newID)
SetCash( newID, 39000000)
AddPerson( newID, CANDIDATE, "John Kerry", 70, 85, 100, SELF)
AddPerson( newID, RUNNING_MATE, "John Edwards", 65, 80, 100,
RUNNING_MATE)
SetCampaignData( newID, "Portrait", "ui_kerry.bmp")
SetCampaignData( newID, "RunningMatePortrait", "ui_edwards.bmp")
SetCampaignData( newID, "Name", "Democratic Party")
SetPersonLocation( newID, CANDIDATE, 22)
SetPersonLocation( newID, RUNNING_MATE, 34)
SetPersonHomeState( newID, CANDIDATE, 22)
SetPersonHomeState( newID, RUNNING_MATE, 34)
SetPersonFatigueValue( newID, CANDIDATE, 1)
SetCampaignColor( newID, 0, 200, 0, 255)
SetCampaignData( newID, "ColorName", "blue")
SetCampaignThinkTime( newID, 2)
SetCampaignData( newID, "LastAction", "Rest")
SetCampaignData( newID, "UniOrg", "3")
SetUniversalOrgLevel( newID, 3)
```



图 3.1 游戏《Frontrunner》的截图, Lua 数据的实时显示

### 3.2.3 事件处理

通常，游戏中的绝大部分重要的处理都是由事件驱动的，要么是来自游戏角色，要么是来自游戏中那些交互的代理。这些事件可以是简单的，如用户按下了 <W> 键，也可以是复杂的，如两个游戏实体同时来到了某个地方。

事件驱动的编程，对于熟悉了 Windows 开发（事件是 Windows GUI 操作的基础）的用户来说不是什么陌生的技术。在 C++ 精心开发的事件系统中，使用 Lua 来接收和处理这些事件，用户可以在游戏内部运行机制、高级 Lua 函数和用户输入之间创建清晰的反馈流程。一个简单的例子是，获取键盘输入然后向 Lua 事件处理器发送该事件，同时将该输入值显示出来。这个概念将一直贯穿于本书之中——事件会返回事件类型 ID 和驱动事件的物体 ID，参照代码清单 3.4。

代码清单 3.4 获取按键事件的例子

---

```
SetEventHandler("MainMenuEvent")
function MainMenuEvent(id, eventCode)

    if eventCode == GUI_EVENT_BUTTON_UP then
        if ID == OK_BUTTON then
            AddTextToConsole("OK button pressed")
            PlaySound("button_click.wav")
        end
    end
end
end
```

---

### 3.2.4 保存和读取游戏状态

保存和读取玩家的数据是游戏开发项目中最具有挑战的事情之一。因为玩家有时需要暂时离开游戏，所以需要一种保存游戏进度的方法。玩家还需要在某种新的尝试和挑战前保存游戏状态，这样万一尝试失败了还可以恢复游戏进度。同样，在开发和测试过程中，开发者会经常需要读取特定的游戏状态来验证游戏功能或者确认 bug 是否已修复。

如果使用 Lua 保存用户的核心游戏数据，那么就可以用 Lua 作为保存和读取当前游戏状态的系统。在 Lua 中，游戏进度文件是简单的可执行的 Lua 代码的文本文件。程序员或者设计师可以通过进度文件来获取当前游戏的状态（也可以修改），读取游戏进度就和执行 Lua 脚本一样简单。利用 Lua 标准的输入/输出函数，编写一个函数来保存游戏数据到可执行的 Lua 脚本中是最直接的方法。这个系统的优点是可以让设计师在开发过程中，根据

游戏数据的增长和删减来编辑并修改这个函数（不需要特别的读取函数）。在开发的最后阶段，还可以利用脚本编译函数来为游戏数据加密。

### 3.2.5 人工智能

人工智能（AI）在如今的游戏中非常关键——玩家需要精明的、有挑战性的对手，感觉就像真人一样。游戏开发者明白真实世界的 AI 不是说完全模仿人如何玩和反应，而是为玩家创造这种感觉。多年来，开发者一直在争论计算机对手“作弊”（计算机对手可以比玩家访问更多的游戏数据）的优缺点。这种争论最好用在别的时间和地方。不过，几乎所有的开发者都同意，比起 AI 模拟，AI 行为的玩家感知更为重要。

就开发 AI 判定来说，Lua 是一种非常高效的工具。有许多人工智能组件，如路径寻找，最好留给底层语言来实现。路径寻找（计算机控制的物体在虚拟世界中的路径寻找）是一个数据运算量很大的工作，计算机需要反复测试可能的路径来寻找最短或者最直接的路径。（路径寻找最好整合到上层的 LuaGlue 函数中以便控制相关参数，但还是会在后面的章节中介绍一种 Lua 的实现。）另一个例子是用最大最小值方法实现的移动判定，一般被用在计算机象棋游戏中，预测之后几步的移动并尝试计算出最优的移动步骤。一般来说，“能思考的函数”都需要大量的数学计算，如导航树或者尝试错误法运算最好都留给底层代码。依赖有限的数据集和参数的人工智能才更适合 Lua 的特点。Lua 的优点是设计师可以编写简单的模型来试错，并快速验证和迭代想法而不需要麻烦程序员。想要利用 Lua 高效率实现 AI，需要很仔细地设计函数（C 函数），让 Lua 脚本可以访问和交互游戏数据。使用 Lua 作为事件管理系统同样可以让 AI 设计师能应对游戏开发中的变更，开发出灵活反应的 AI 系统。

在代码清单 3.5 中，用户可以看到使用 Lua 来评估可能的竞选旅行目的地，评估的依据是这个州的选举人票数量和这个州的支持率。

代码清单 3.5 评估虚拟的美国总统候选人应该去哪个州的 Lua 人工智能函数

```
--AI candidate travel
StateTravTable = {}
stateCount = 1
--build a set of potential 'travel to' states
while stateCount < 6 do
    pickState = math.random(1,51)
    --make sure there are enough electoral votes to make it worthwhile
    if GetStateData(pickState,'ElectoralVotes') > 12 then
```

```
StateTravTable[stateCount] = pickState
stateCount = stateCount + 1
end
end
--pick the state out of the list with the lowest support
lowSupport = 100
targetState = 1
for indx = 1,5 do
    if TalleySupport(StateTravTable[indx], Campaign) < lowSupport then
        lowSupport = TalleySupport(StateTravTable[indx], Campaign)
        targetState = StateTravTable[indx]
    end
end
end
--issue the travel order
Travel(Campaign, Person, targetState)
SetCampaignData(Campaign, "LastAction", "Travel")
```

### 3.2.6 快速构建原型

商业化的游戏必须为玩家提供高性能的体验，掉帧和处理延迟的现象在如今竞争激烈的市场中是绝不允许的。程序概要分析（Profiling）是行之有效的确定性能瓶颈的方法，但必须在所有功能都开发完成并正确工作的前提下才能进行。原型开发和性能改善一般是不能同时进行的。

Lua 是构建可移植的核心游戏功能原型的不错工具。因为 Lua 可以整合原生语言开发的组件，所以移植单独的 Lua 函数到 C++，对于其他 Lua 函数来说是不需要变更的。它可以让设计师在构建原型时，如果碰到有高性能需求的函数就可以让程序员用底层语言来实现。因为程序的算法结构已经有了，所以 C 函数和 LuaGlue 调用都可以高性能、无缝地集成在项目中。

### 3.3 本章小结

在游戏开发领域，Lua 和 C++ 是一个功能十分强大的组合。对于 GUI 开发、事件管理、数据保存和获取、游戏进度保存和载入以及人工智能这些游戏开发工作，需要能够快速地构建原型、测试和迭代设计，而这些正是 Lua 所擅长的。在核心底层语言开发的项目中，频繁的变更会导致系统的不稳定，并让开发者不能专注于更重要的开发任务。使用 Lua 环境分担这些工作，可以让设计师和程序员快速而安全地使用脚本来设计、测试和实现游戏功能。

# Lua 入门

## 本章要点

- 使用 Lua 控制台
- Lua 基础
- 变量
- 运算符
- 控制结构

本章将向用户简单介绍 Lua 控制台，这个小程序可以让用户直接地运行 Lua 脚本（也可以运行大一些脚本文件）。有了它，用户就能开始学习 Lua 脚本语言核心的功能（和学习别的编程语言类似），并了解到如何在游戏开发中利用这个工具。

## 4.1 使用 Lua 控制台

本书附带的光盘中有 Lua 控制台，用户可以用它输入 Lua 命令并得到处理结果。同时，它还能载入和执行 Lua 脚本文件。在下一章中，用户可以学习如何创建控制台，但这一章中，我们只用它作为学习 Lua 语言的工具。


Lua 控制台程序看起来非常像 Microsoft Windows 中的 MS-DOS 命令程序。后者用来处理 DOS 命令（或者运行批处理文件，DOS 版的脚本文件），而这个控制台用来处理 Lua 命令。

Lua 命令是一些简单的语句，在环境中可以立刻被处理。可以参照下面简单的 Lua 命令：

```
myValue = 7
print("hello")
myTable = {1,2,3,4}
```

Lua 环境可以立刻处理这些命令。打开 Lua 控制台输入这些命令，在每个命令后按下



 键，可以看到 print 命令会产生输出，而其他的命令仅仅是被处理了。

在控制台中，用户可以使用标准的方式单击和选中文本，按下 <Enter> 键来复制内容到剪贴板，然后单击鼠标右键选择“粘贴”命令进行粘贴。

Lua 脚本是包含一系列 Lua 命令的简单脚本（扩展名为 .lua 的文本文件）。Lua 不关注格式、制表符、换行符或者其他让脚本文件更具可读性的转换符号（不过 Lua 还是需要命令之间的空格符号，用来区别变量和函数调用）。

简单的 Lua 脚本：

```
for indx = 1,4 do
    Print('line: ', indx)
end
```

也可以写成：

```
for indx=1,4 do print('line: ',indx) end
```

格式化后的脚本更容易阅读，但并不改变其运行结果。

单个命令或者一系列命令组成的脚本文件，在 Lua 中我们把它称做“代码块”（Chunk），代码块可以很小（如单个 print 命令），也有大到几兆字节的（如大型的保存游戏功能或者数据定义脚本）。

尽管可以在 Lua 控制台中使用单个命令或者单行输入，不过我们一般都执行 Lua 脚本。之后将为大家举例说明如何在 Lua 控制台中运行脚本，然后再开始学习 Lua 语法和常用的命令。

在 CD-ROM 中，读者可以找到代码清单 4.1 的代码，它保存在本章的第一个文件中。



代码清单 4.1 ch4\_1.lua

```
-- Game Development with Lua
-- by Paul Schuytema and Mark Manyen
-- Published by Charles River Media
-- Lua script example
-- Listing 4.1
-- Hello World
myString = 'Hello World'
print(myString)
```

为了运行该脚本，需要将它复制到本地文件夹（最好创建一个文件夹来保存所有在学

习过程中要用到的脚本，如 c:\lua\_scripts)。

参照图 4.1 在控制台中运行下列命令。



图 4.1 在控制台中执行脚本文件

dofile 命令用来立即执行脚本。参数是字符串，指定文件的名字和路径。我们使用“\\”代替单斜线，因为单斜线是用来告诉 Lua 运行环境它后面是特殊符号（如双引号、换行符等）。

使用 dofile 命令可以在命令行中运行任何脚本（只要不是那些包含了用 C++ 写的 LuaGlue 函数的就可以）。如果脚本有什么错误的话，那么命令行会给出错误信息以便调试。

从本章一直到下一章，最好亲手创建一些脚本来学习接下来的例子，并使用 dofile 命令来执行它们（这个比起在命令行中手动输入可轻松多了）。

## 4.2 Lua 基础

Lua 是一门简单的编程语言，它的优势在于可以整合 C++ 的模块来扩展自身的功能。不过作为入门书籍，我们仅在本章中介绍该编程语言的一些基础语法知识。

### 语言定义

在 Lua 语言中，标识符有很大的灵活性（变量和函数名），不过用户不能以数字作为起始符，也要避免下画线（\_）接大写字母，因为这种格式是为 Lua 自身保留的，如 \_Start。

### Lua 的保留关键字

Lua 有一些不能作为标识符的保留关键字，见表 4.1。

表 4.1 Lua 保留关键字

and	local
break	nil
do	not
else	or
elseif	repeat
end	return
false	then
for	true
function	until
if	while
in	

在第9章“设计 Lua 版本的实现”中，我们将讨论使用 Lua 样式指南来让用户的脚本更加容易阅读和保持一致性，不过目前，建议用户使用下面的格式和命名规则来定义变量、常量和函数名：

- 常量用全大写和下画线，例如：MY\_CONSTANT
- 变量第一个字母小写，例如：myVariable
- 全局变量第一个字母用小写 g 表示，例如：gMyGlobal
- 函数名第一个字母大写，例如：function MyFunction()

在 Lua 中，用户可以在一段文字前加两个减号（--）来标记该行为注释，还可以使用块注释，参照下面的示例：

```
-- this is a comment in Lua that is on its own line
myValue = 7 --you can also add a comment to a line of script
--[

function Counting()
    for indx = 1,50000 do
        print(indx, "+", indx + 1, "=", indx + (indx + 1))
    end
end
--]]
```

## 4.3 变量

在 Lua 中，变量不需要在使用前声明，这个稍微有些争议。因为不需要声明，所以可以在任何地方引入需要的变量。形式主义者会抱怨这个会导致糟糕的程序代码，因为不是显式声明，所以很难追踪变量，或者是在另一个函数中用了相同名字的变量而造成数据的混乱。因此，底线是我们必须十分小心地使用变量，要保证变量是可追踪的，编程语言本身可不会帮我们。同样不需要指定变量的类型（string、number 等），因为变量的类型取决于用户给它赋的值。这种方式的优点是有很大的灵活性，但如果不小心处理变量就会给程序调试带来困难。

创建、分类和给变量赋值可以参考下面的代码：

```
myValue = 7
```

这段代码创建了一个名为 myValue 的变量，并给它赋值为数字 7。可以使用 type 函数来判断变量的类型。在控制台试试下面的代码：

```
Ready> myValue = 7
Ready> print(type(myValue))
number
Ready> myValue = 'hello'
Ready> print(type(myValue))
string
```

可以看到，通过简单的赋值就可以更改变量的类型。Lua 中有 5 种变量类型：nil、Boolean、string、Number 和 table。

### 4.3.1 nil

nil 是一个简单类型，用来表示这个变量还没有被赋值（参考下例）。如果给一个变量赋值为 nil，那么实际上表示删除该变量的意思。

```
myValue = nil --this deletes the variable
local myValue --this creates a local variable with an initial nil
value
```

### 4.3.2 Boolean

Boolean 类型的变量只有两种值：true 和 false。Boolean 变量在条件表达式中非常有用，

参考下面的例子：

```
myValue = true --creates a boolean variable with a value of true
```

### 4.3.3 string

字符串 (string) 在 Lua 中相对简单，在下一章中我们会介绍一些非常实用的字符串处理函数（实际上快速的字符串操作是 Lua 出色的特性之一）。Lua 字符串可以小到一个字，也可以包含百万字符以上。

```
myValue = "hello world" --a string variable
```

#### 特殊字符串

Lua 提供了一些特殊字符串，见表 4.2。

表 4.2 Lua 字符串

\a	响铃	\v	垂直制表符
\b	退格	\\	反斜杠
\f	换页符	\"	双引号
\n	换行符	\'	单引号
\r	换行符	\[	左方括号
\t	制表符	\]	右方括号

需要注意的是，Lua 会根据上下文在合理的情况下进行数字和字符之间的转换。在控制台测试如下代码：

```
Ready> print("8" + 8)
16
Ready> print("8 + 8")
8 + 8
Ready> print("hello world" + 8)
ERROR:[string "?"]:1: attempt to perform arithmetic on a string value
```

### 4.3.4 Number

Number 在 Lua 中是双精度浮点数。Lua 没有整数类型（由于小于  $1e14$  的数值不存在取整误差，所以不需要整数类型）。数字可以用下列方式表达：

- myNumber = 7
- myNumber = 0.765

- `myNumber = 7.65e8` (表示  $7.65 \times 10^8$ , 或者 765 000 000)
- `myNumber = 7.65e-2` (表示  $7.65 \times 10^{-2}$ , 或者 0.0765)

### 4.3.5 table

table 在 Lua 中是最强大也最容易造成困扰的数据类型。在后面的章节中,我们会详细讨论 table 的细节和在脚本中使用这个数据类型。但是作为入门,我们可以把它当做数组来使用。在控制台测试如下代码:

```
Ready> myTable = {2,4,6,8,10}
Ready> print(myTable[3])
6
Ready> myTable[6] = 12
Ready> print(myTable[6])
12
```

在这个例子中, table 相关的函数看起来就像保存了许多变量的数组。我们使用中括号和索引来获取 table 中的值。之后我们会再详细介绍 table。

### 4.3.6 局部变量和全局变量

Lua 变量默认是全局的,也就是说,变量的值在整个会话(Session)中是保持不变的,除非脚本改变了它。尽管这个特性给脚本程序员带来了许多方便,不过也会造成一些困扰,特别是在一个有很多脚本和函数的游戏项目中。

当使用全局变量时,变量名前加一个 `g` 字母(参照语言定义小节)会更加明确(使脚本之后更加容易调试)。当然,尽量使用局部变量会更好。

定义局部变量可以给它设定一个初始值,也可以不用。参考下面的示例:

```
local myValue -- the variable is declared and has a value of nil
local myValue2 = 3 -- the variable has an initial value of 3
```

变量的有效范围(变量存在的范围)取决于声明变量的位置。参考下面的示例:

```
function MyFunction()
  local myX = 7 --this will be destroyed when the function is done
  if myX < 10 then
    local myY = "hello world" --this will be destroyed
    --when this code block is done
    print(myY) -- prints "hello world"
  end
  print(myY) --print nil, because the variable above is destroyed
end
```

局部变量可以让用户的代码更加规范，特别是项目越来越大、脚本越来越多的时候（用户可以不用辛苦地追踪一个藏在某个隐秘角落的被错误赋值的全局变量）。在控制结构中（参考 4.5 节），局部变量非常适合作为那种常用的计数器，如在 for 循环中。

## 4.4 运算符

运算符是特殊的符号，可以让两个值（如变量）得出运算结果。算术运算符可以得出计算结果，关系运算符可以得出 Boolean（true 或者 false）的结果。

### 4.4.1 算术运算符

Lua 支持下列标准的算术运算：

- 加，例如： $a + b = c$
- 减，例如： $a - b = c$
- 乘，例如： $a * h = c$
- 除，例如： $a / b = c$

### 4.4.2 关系运算符

用户可以使用下列标准的关系运算符来比较值或者表达式：

```
if a == b then -- equal to
    print('a is equal to b')
end
if a ~= b then -- not equal to
    print('a is not equal to b')
end
if a < b then -- less than
    print('a is less than b')
end
if a > b then -- greater than
    print('a is greater than b')
end
if a <= b then -- less than or equal to
    print('a is less than or equal to b')
end
if a >= b then -- greater than or equal to
    print('a is greater than or equal to b')
end
```

如果用户使用关系运算符来比较两个 table，只有两个 table 是同一个对象的时候才能得到预期的结果，因为变量只是 table 对象的引用（就像指针一样），不能直接比较存在于 table 中的值，例如：

```
tableA = {1, 2, 3}
tableB = {1, 2, 3}
if tableA == tableB then
    print("The tables are the same")
else
    print("The tables are not the same")
end
```

在前面的例子中，我们可以看到结果 “The tables are not the same”，因为这两个 table 是完全不同的结构。

```
tableA = {1, 2, 3}
tableB = tableA
if tableA == tableB then
    print("The tables are the same")
else
    print("The tables are not the same")
end
```

在第二个例子中，我们可以得到结果 “The tables are the same”，因为 tableB 指向了同样的对象 tableA。需要注意的是，用户可以按照期待的方式检查 table 中的值。如果我们在第一个例子中添加如下判断：

```
if tableA[2] == tableB[2] then
    print("The values are the same")
end
```

那么我们可以看到 table 中的值确实是相等的。

#### 4.4.3 逻辑运算符

逻辑运算符测试两个参数并返回相互关联的结果。在 Lua 中，逻辑运算符使用小写字母。

and 运算符比较两个参数时，如果第一个参数是 false，则返回 false；否则返回第二个参数的值。

```
a = 5
b = 10
c = 20
```



```
if (a < 10) and (b < 20) then
    print('this returns true -- which is the value of the second
    argument')
end
if (a > c) and (b < 20) then
    Print('this returns false -- which is the value of the first
    argument')
end
```

or 运算符和 and 正好相反。如果第一个参数不是 false，则返回第一个参数的值。如果第二个参数是 true，则返回第二个参数的值。

```
a = 5
b = 10
c = 20
if (a < 10) or (b < 20) then
    print('this returns true -- which is the value of the first argument')
end
if (a > c) or (b < 20) then
    print('this returns true -- which is the value of the first argument')
end
if (a > c) or (b < 5) then
    print('this returns false -- which is the value of the second argument')
end
```

not 表达式返回 true 或者 false。在 Lua 中，false 和 nil 会被逻辑运算符都当成 false，其他值为 true。not 运算符返回参数的相反值（参考图 4.2）。



图 4.2 控制台其中的一些 not 运算符的例子

## 4.5 控制结构

Lua 包含了一小部分重要的控制结构，可以让用户在脚本中有充分的选择自由。所有的控制结构都以 end 作为结束标记。

尽管没有必要在控制结构中添加缩进，不过在嵌套的控制结构中添加清晰的缩进是一个非常好的习惯。在稍后的章节中，读者会看到游戏脚本中大量的嵌套式控制结构，特别是当用户在编写人工智能函数的时候。

#### 4.5.1 if

if 语句很常见，如果用户熟悉其他的编程语言或者脚本语言，肯定见过很多次了。if 语句可以判断一个参数，如果是 true 的话，那么程序块就会被执行。例如：

```
myValue = 7
if myValue < 10 then
    print('myValue is less than ten. ')
end
if (myValue > 5) and (myValue < 10) then
    print('myValue is between five and ten. ')
end
```

还可以使用 else 和另一个程序块来扩展 if 语句的功能。如果 if 语句得到 false 的结果，那么被 else-end 括起来的程序块将被执行。例如：

```
myValue = 20
if myValue == 21 then
    print('the value is 21')
else
    print('the value is NOT 21')
end
```

此外，用户还可以使用 elseif 关键字来添加一系列的条件式。在 AI 脚本中，这个关键字非常有用，因为 Lua 不支持 case 句式。例如：

```
myValue = 17
if myValue < 6 then
    print('myValue is between zero and five. ')
elseif myValue < 11 then
    print('myValue is between six and ten. ')
elseif myValue < 16 then
    print('myValue is between eleven and fifteen. ')
elseif myValue < 21 then
    print('myValue is between sixteen and twenty. ')
else
    print('myValue is greater than twenty. ')
end
```

#### 4.5.2 while 和 repeat

while 和 repeat（参考下一个例子）控制结构非常相似，它们都可以循环执行一段脚本

直到满足某个条件。while 控制结构首先判断一个参数，如果条件为 true，那么程序块会被执行（也有可能永远不被执行）。repeat 控制结构则是在最后判断参数，这就保证了该程序块至少会被执行一次。

while 控制结构使用 do 关键字，和 if 控制结构的 then 关键字一样，用来标记有条件的程序块的开始，参考下面的示例：

```
indx = 1
while indx < 10 do
    print("loop pass: ", indx)
    indx = indx + 1
end
```

在这个例子中，如果 indx 初始值大于或等于 10，那么脚本就不会被执行。

在 repeat 控制结构中，repeat 关键字用来标记程序块的开始，until 标记程序块的结束。控制结构的参数跟在 until 关键字之后，例如：

```
indx = 1
repeat
    print("loop pass: ", indx)
    indx = indx + 1
until indx > 10
```

在这个例子中，如果 indx 初始值为 1000，那么程序块仍然会被执行一次。当用户考虑该使用哪个控制结构时，需要首先明确是否程序块至少被执行一次，如果是这种情况，就应该使用 repeat 控制结构。

需要特别注意条件参数，避免返回预期外的 true 结果，要不然就会碰到死循环的情况，从而不得不强制关闭程序。

### 4.5.3 for

Lua 提供了两种 for 控制结构（数字型和通用型），这里我们先解释数字型的控制结构，在下一章关于 table 的进阶教程中再学习另一种。

for 控制结构可以让用户根据表达式的值有限次地执行一段脚本。一个简单的示例如下：

```
for indx = 1,10 do
    print(indx)
end
```

在 for 关键字之后，需要提供变量值的范围，遍历这个范围内每个值的同时，程序块会被执行。do 关键字标记程序块的开始，end 标记程序块的结束。

用户可以在第三个参数中定义“step”的值。例如：

```
for indx = 10,1, -1 do --this counts backwards
    print(indx)
end
for indx = 1,100, 2 do --this counts forwards by 2s
    print(indx)
end
```

在使用 for 控制结构时，用户需要注意以下几点。首先，循环次数只是在第一次执行时确定，因此，就算用户更改了参数的值也不会影响最终循环的次数。其次，循环结构中的变量是局部变量，一旦循环结束就会被清除。如果想保存它们的值，那么必须使用全局变量或者更高级别的局部变量。

#### 4.5.4 break

break 语句可以从循环控制结构中强制退出。用户不能在循环外使用它，而且它必须在程序块的最后（通常是 if-then 语句）。参考下面的示例：

```
for indx = 1,100 do
    if indx == 52 then
        print('52--ouch! ')
        break --the last line of the block, breaks the for loop
    end
    print('the value is ', indx)
end
print('this is the line that will be executed after the break')
```

### 4.6 本章小结

Lua 是简单、明确的脚本语言，使用了很多其他编程语言中常见的控制结构和句式，如果用户熟悉其他编程语言，那么使用 Lua 时会感觉非常容易。

Lua 控制台是测试短小的 Lua 代码块（chunks）和语句的非常好的工具，但同时它也可以执行 Lua 脚本文件（之后，用户会在游戏项目中执行 Lua 脚本）。

在本章中，我们学习了 Lua 基本的语法和核心的语句。在下一章中，我们会继续巩固这些知识并进一步深入学习 Lua，为之后真正的游戏项目开发做好准备。

## 第 5 章

7-27429208-0 第 5 章 深入学习 Lua

# 深入学习 Lua

### 本章要点

- 函数
- 标准库
- 字符处理
- table 数据结构
- I/O 基础

在上一章中，我们简单学习了 Lua 的基本语法并在控制台测试了一些简单的例子。这一章会进一步介绍 Lua 编程语言的基础知识。你会了解更多高级的特性和标准函数，为将来开发令人兴奋的 Lua 游戏打下坚实的基础。

## 5.1 函数

函数是划分游戏脚本功能的主要工具。它是通过标识符（事实上是一个变量）来调用的 Lua 代码块，可以执行某种处理、返回数值，或者二者都有。

简单的函数定义如下例：

```
function Wow()  
    print(" ")  
    print("Wow, that was awesome!")  
    print(" ")  
end
```



本章所有的例程都可以在 CD-ROM 中的 ch5\_1.lua 文件中找到。如果在控制台运行该脚本，就可以在命令行中简单地执行这些函数（参见图 5.1）。

函数的定义以 `function` 关键字开始，后面是函数名称，然后是传递给函数的参数列表。在上面的例子中，没有参数传给函数，但我们仍然用 `()` 来表示一个空的参数列表。函数的定义以 `end` 关键字结尾。



图 5.1 控制台可以让你载入包含函数定义的文件，然后在命令行中运行这些函数

在载入脚本文件时，文件中的函数并没有被执行（如同 `dofile` 命令那样），仅仅是被载入内存中并和函数名变量关联起来。

### 5.1.1 单一参数

我们来看看下面这个只有一个参数的例子：

```
function SetName(myString)
    print(" ")
    print("Your name is:", myString)
    print(" ")
end
```

在这个函数中，参数 `myString` 传递给了函数，并在函数中使用。函数中的参数是局部变量，在调用结束时被回收。

### 5.1.2 多个参数

你也可以传递多个参数，但之间用逗号隔开，例如：

```
function MyInfo(myName, myAge)
    print(" ")
    print("Your name is:", myName)
    print("Your age is:", myAge)
    print(" ")
end
```

Lua 还有一个强大的功能，就是可以定义不定长的参数列表。使用 `(...)` 代替参数列表，Lua 会创建一个局部的名字为 `arg` 的 table，保存所有调用时传递的参数，以及参数个数（通过 `arg.n` 获取）。对于这种函数最好添加清楚的注释，因为在游戏开发过程中这个函数可能会被别人修改或调用，所以很容易忘记需要多少个参数。参考下面的例程，检查

HowMany()函数的输出结果:

```
function HowMany(...)
    if arg.n > 0 then
        for indx = 1, arg.n do
            local myString = string.format("%s%d", "Argument ", indx,
                ":")
            print(myString, arg[indx])
        end
    else
        print("No variables entered.")
    end
end
```

在这个函数中,我们可以传递任意多个有效的参数。arg 表中保存了所有的值, arg.n 中保存了参数的数量值。使用这个特性时,我们通常还会定义一些必要的参数,并结合可选的不定长参数列表。参考下面的示例:

```
function Multiply(val1, val2, ...)
    --the default is one
    local myString
    if arg.n == 0 then
        myString = string.format("%d%s%d%s%d", val1, " * ", val2, " = ",
            val1 * val2)
    else
        local val3 = val1 * val2 * arg[1]
        myString = string.format("%d%s%d%s%d%s", val1, " * ", val2, " * ",
            arg[1], " = ", val3)
    end
    print(myString)
end
```

在例程的函数中,前两个参数是必须的,会用来做简单的乘法运算。如果有第三个参数,则会继续相乘,之后的参数将忽略。

### 5.1.3 返回值

使用函数进行独立的处理,可以将处理结果返回到被调用的脚本。函数使用 return 关键字并跟上数值(通常为变量名)来返回结果。参考下面的示例:

```
function TimesTwo(myValue)
    myValue = myValue * 2
    return myValue
end
```

可以使用有返回值的函数作为表达式的参数,例如:

```
a = 24 + TimesTwo(12)
print(a)
```

函数还可以返回多个结果，只需用逗号分隔开。参考下面的示例，其中 ThreeDice() 函数输出的结果可以参见图 5.2。

```
function ThreeDice()
    d1 = math.random(1,6)
    d2 = math.random(1,6)
    d3 = math.random(1,6)
    myTotal = d1 + d2 + d3
    return d1, d2, d3, myTotal
end
```

A screenshot of a Windows command prompt window titled "C:\lua\_scripts\luaconsole.exe". The window shows the following text:

```
Ready>
Ready> print<ThreeDice()>
1      3      5
Ready> print<ThreeDice()>
1      6      8
Ready> print<ThreeDice()>
3      1      5
Ready> print<ThreeDice()>
3      4      8
Ready> a,b,c,d = ThreeDice()
Ready> print(a,b,c,d)
4      4      3      12
Ready> print(c)
4
Ready> _
```

图 5.2 控制台中 ThreeDice() 函数的输出结果

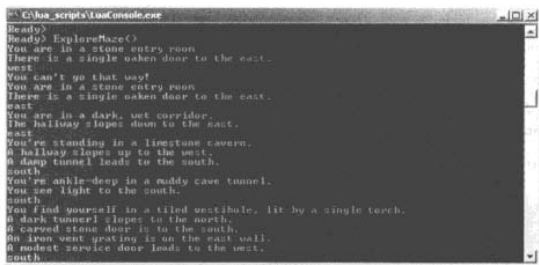
Lua 另一个有趣的特性是使用 return 去调用另一个函数。这个特性有趣的地方（相比于在函数体中调用）在于 Lua 把 return 语句作为一个函数的物理结束，运行结束后从栈中被删除。

这个特性可以让你编写一系列的函数调用，并降低发生栈溢出错误的风险。从原理上来讲，Lua 使用栈（stack）来维护各种变量、数值等，当它们被函数返回时，会被 pop 出栈，然后在调用的地方接着被使用。如果使用简单退出函数的方式并在存在嵌套的函数中调用，就有可能会占满有限的数据空间，从而导致 Lua 命令解释器崩溃。参考示例，可使用 dofile 命令来读取 CD-ROM 中 ch5\_2.lua 脚本，并在控制台输入 ExploreMaze()。

在如图 5.3 所示的老校舍迷宫探险的例子中，我们使用函数本身作为迷宫的不同地点，



并使用 `return` 让玩家到达新的地方。因为每个函数都在 `return` 之后从栈中被清除，所以我们不会遇到任何数据溢出的危险，或者进入无休止的循环嵌套函数调用陷阱中，而导致程序崩溃。



```
C:\lua_scripts\luaconsole.exe
Ready>
Ready> ExploreMaze()
You are in a stone entry room.
There is a single oaken door to the east.
west
You can't go that way!
You are in a stone entry room.
There is a single oaken door to the east.
east
You are in a dark, wet corridor.
The hallway slopes down to the east.
east
You're standing in a limestone cavern.
A hallway slopes up to the west.
A damp tunnel leads to the south.
south
You're ankle-deep in a muddy cave tunnel.
You see light to the south.
south
You find yourself in a tiled vestibule, lit by a single torch.
A dark tunnel slopes to the north.
A carved stone door is to the south.
An iron vent grating is on the east wall.
A modest service door leads to the west.
south
```

图 5.3 老校舍迷宫探险的输出结果

## 5.2 标准库

Lua 提供了大量标准函数库，可以帮助你完成许多复杂的处理而不需要编写额外的代码。Lua 5.0 版本的标准函数库的详细资料可以参考在线文档 [www.lua.org/manual/5.0/](http://www.lua.org/manual/5.0/)。

在本章剩余的部分中，我们将学习一些常用的标准函数，让你的游戏开发过程变得更加轻松。

### 5.2.1 `assert(myValue)()`

`assert` 函数可以让你像处理函数一样运行编译后的 Lua 代码块（chunk）。传入指向编译后的代码的变量，然后立刻被执行。你可以使用 `loadstring` 或者 `loadfile` 函数载入并编译脚本。

在游戏开发中，虽然 `loadfile` 并不常用（因为可以用更简便的方式载入文件），但是 `loadstring` 和 `assert` 函数经常配合使用。你可以用 `loadstring` 函数将编写好的代码块（chunk）存到 string 中，然后再用 `assert` 函数执行它。

你也可以使用 `loadstring` 函数直接执行代码块，如图 5.4 例程所示，它不提供像 `assert` 函数那样的报错功能，但可以作为一种“简单粗放”的运行脚本的方式。参考下面的

示例:

```
myString = "math.max(7,8,9,10)"  
loadstring(myString)()
```



图 5.4 这个例子是将 Lua 脚本存入 string, 然后再使用 loadstring 和 assert 函数执行它

### 5.2.2 dofile(filename)

前面我们已经使用过这个函数了, 它的功能是载入并立刻执行 Lua 脚本文件。通常我们用它来载入定义函数的文件以备调用, 除此之外还可以用来载入数据文件 (如游戏存档, 稍后章节中会提到) 或者想立即执行的 Lua 代码。参考下面的示例:

```
dofile('scripts/runtime_functions.lua');
```

通常, 一个游戏项目有若干子目录用来存放游戏素材 (如贴图、脚本、音效等)。dofile 函数会把程序的执行目录作为当前目录, 因此要运行脚本目录中的文件, 需要在文件名的开始加 “scripts/”。

在后面的章节中, 我们将要学习两个自定义函数, 一个是执行用户界面的 RunGUI() 函数, 另一个是 RunScript() 函数, 和 dofile 比较类似, 不过它还可以执行已经编译好的二进制的 Lua 文件。

#### 数学运算函数

Lua 提供函数级别的可以调用 C 标准库的数学运算函数。大部分是简单的 LuaGlue 接口, 它们调用了 C 标准库。这些函数实际上是存放在一个叫做 “math” 的表中, 你可以使用标识符来使用这些功能, 标识符请参考表 5.1。

表 5.1 标识符

math. abs	math. max
math. acos	math. min
math. asin	math. mod
math. atan	math. pow
math. atan2	math. rad
math. ceil	math. sin
math. cos	math. sqrt
math. deg	math. tan
math. exp	math. frexp
math. floor	math. ldexp
math. log	math. random
math. log10	math. randomseed

表 5.1 中还包括 math. pi, 用来保存 pi 值。

下一小节中还会给人家介绍一些在游戏开发中常用的函数。

### 5.2.3 math.floor()

floor 函数用来向下取整 (Lua 中没有浮点数或者整数的概念), 该函数只是舍去小数部分。如果想四舍五入取整一个数字, 那么可以先给它加上 0.5, 然后再向下取整。参考下面的示例:

```
a = 5.125
b = 5.75
a = a + 0.5
b = b + 0.5
a = math.floor(a) -- a will equal 5
b = math.floor(b) -- b will equal 6
```

### 5.2.4 math.random()

在游戏开发过程中, 随机函数随处可见。math.random() 函数随机生成一个 0~1 之间的伪随机数 (和其他编程语言类似)。Lua 函数更有用的是可以传入最大值和最小值, 这样就可以随机生成这个范围内的数字, 参考下面的示例:

```
-- 6-sided dice  
myDie = math.random(1,6)
```

当程序开始的时候，最好给随机数种子设定一个唯一的值，这样就可以得到更好的随机数值。这个处理在 StartGui.lua 文件中，在初始化游戏时被运行（参考 7.2 节）。最简单的方式是使用 os.date 函数，它可以产生唯一的值，例如：

```
\math.randomseed(os.date("%d/%m/%S"))
```

为了调试时，可能需要把这个随机数种子设定为整数，这样就可以保证每次运行游戏时都能得到相同的数值。

### 5.2.5 math.min()

在游戏开发过程中，经常需要确定一个集合中的最大值或者最小值（如英雄角色的最高属性，或者最多选票）。math.min 函数和 math.max 函数可以为你提供这样的功能。它们可以接受若干数字类型的参数，math.min 函数返回最小值，math.max 函数返回最大值。

在 Lua 中，由于大部分数据都存在于 Table 中，所以使用函数处理起来会有一些挑战，因为你需要把 Table 中的所有数据都加到函数的参数中。不过还好，我们可以创建字符串然后利用 loadstring 函数来解决这个问题，参考下面的示例：

```
function GetMin(theTable)  
    myString = 'myValue = math.min(  
        for index,value in ipairs(theTable) do  
            myString = string.format("%s%d%s", myString, value, ',')  
        end  
        --remove final comma  
        myString = string.sub(myString, 1, string.len(myString) - 1)  
        --add final )  
        myString = string.format('%s%s', myString, ')')  
        --run the chunk  
        loadstring(myString)()  
        return myValue  
    end
```

该函数和 GetMax(myTable) 函数的代码可以在 CD-ROM 中找到。它们接收 table 作为参数，并返回 table 中的最大或最小值。



## 5.3 字符处理

Lua 最强大的特性之一就是它的字符处理能力。Lua 具有可扩展的模式匹配功能, 以及许多实用的字符处理函数。在本章的这个小节中, 我们将会介绍许多在游戏开发中常用的函数。完整的字符处理函数的资料可以参考 Lua 在线文档。



本节中的例程可以在 CD-ROM 的 ch5\_small\_examples.lua 文件中找到。

### 5.3.1 类型转换

在游戏开发中, 字符和数字之间常常需要转换。要把字符转换成数字, 可以使用 `tonumber()` 函数, 例如:

```
myString = "1234"
myNumber = tonumber(myString)
print(myNumber + 2) -- this will display 1236
```

还可以使用 `tostring()` 函数把数字转换成字符:

```
myNumber = 1234
myString = tostring(myNumber)
print(type(myString)) -- you will see "string"
```

### 5.3.2 string.char(n1, n2, ...)

`string.char` 函数根据 ASCII 编码返回传入参数对应的字符。这个函数不是很常用, 但在 Lua 游戏保存文件中插入一个换行符的时候非常有用, 它可以让文件更便于阅读。例如:

```
myFile:write(string.char(10)) -- writes out a linefeed to the open
file
```

### 5.3.3 string.len(myString)

通常情况下, 知道字符串的长度是非常有用的, 该函数可以告诉你这个信息, 即返回传入参数的字符数。例如:

```
myString = "1234"
print(string.len(myString)) -- will print 4
```

### 5.3.4 string.sub(myString, start, end)

string.sub 函数返回指定字符串的子串。start 参数指定子串的开始位置，end 指定子串的结束位置。例如：

```
myString = "hello world"
newString = string.sub(myString, 1, 5)
print(newString) -- this will print "hello"
```

也可以指定 start 参数为负数，这种情况下，子串的位置从字符串的最后开始计算（如果 start 为 -5，那么会返回字符串的最后 5 位）。例如：

```
myString = "hello world"
newString = string.sub(myString, -5, 10)
print(newString) -- this will print "world"
```

end 参数可以省略，这时函数会返回从 start 到字符串末尾的子串。通过这种方式可以获取指定字符的后缀，例如：

```
myString = "hello world"
newString = string.sub(myString, -5)
print(newString) -- this will print "world"
```

### 5.3.5 string.format()

string.format 函数可以让你格式化输出指定字符串。在输出字符串到 GUI 界面时，这个函数很常用。我们可以用这个函数来连接字符串（Lua 不能简单地连接两个字符串），例如：

```
string1 = "hello"
string2 = "world"
for index = 1, 3 do
    string1 = string.format("%s%s", string1, string2)
end
print(string1) -- prints "helloworldworldworld"
```

在上面的例子中，string.format 函数的第一个参数用来指定字符串的格式。由于%s 表示字符串（%d 表示数字），所以%s%s 表示连接两个字符串。

string.format 函数的另一个主要用途是根据参数格式化输出复杂的字符串。例如：

```
myName = "Fred"
myStr = 16
myString = string.format("%s%s%d%s", myName, 's strength is ', myStr, ".")
print(myString)
```

在这个例子中, %s %s %d %s 格式为函数指定了参数的位置, 因此可以得到结果“Fred's strength is 16.”。这时, 符合定义的参数会按照其指定格式输出结果。我们还可以改变函数格式来得到相同的结果, 例如:

```
myString = string.format("%s's strength is %d.", myName, myStr)
```

这个例子中我们把格式符和固定字符串组合在一起并得到相同的结果。

string.format 函数还有其他一些用途, 如输出指定位数的数字。参考下面的示例:

```
myHealth = 17.34556
myString = string.format("%.2f%s", myHealth, "% of health remaining.")
print(myHealth) -- prints '17.34556'
print(myString) -- prints '17.35% of health remaining.'
```

这个例子中, %.2f 表示以两位小数的格式输出 myHealth 的值。

### 5.3.6 string.find(sourceString, findString)

string.find 该函数会在 sourceString 中查找第一个符合 findString 字符的位置。如果找到了该目标字符则返回它的开始和结束位置; 如果没有找到则返回 nil, 例如:

```
myString = "My name is John Smith."
sStart, sEnd = string.find(myString, "John")
print(sStart, sEnd) -- prints '12 15'
```

### 5.3.7 字符和格式

Lua 强大的字符处理函数支持格式化功能。在前面的例子中, 我们看到了格式在 string.format 函数中的使用。格式是一种模板, 让 Lua 可以从字符串中过滤出有意义的结果。参考下面的示例:

```
myString = "The price is $17.50."
filter = "$%d%.%d%"
print(string.sub(myString, string.find(myString, filter)))
```

在这个例子中, filter 参数指定了我们需要查找内容的格式。我们要查找的结果包含美

元符号和小数，数字可以是任意的数字。例程的结果是“\$17.50”。

使用大写字母可以得到相反的格式，如% d 表示所有数字，% D 则表示非数字。

加%前缀可以让特殊符号（例如（）. % + \_ \* ? [ ^ \$ ]）也能用在格式中，如%%代表百分比符号。

Lua 支持下列字符作为格式化输出符号：

.	所有字符
%a	字母
%c	控制符
%d	数字
%l	小写字母
%p	标点符号
%s	空格符号
%u	大写字母
%w	字母数字
%x	十六进制数
%z	用0表示的字符

### 1. string.gsub(sourceString, pattern, replacementString)

string.gsub 函数返回一个字符串，sourceString 字符中满足 pattern 格式的字符都会被替换成 replacementString 参数的值。参考下面的示例：

```
myString = 'My name is John Smith. My phone is 555-3257.'
newString = string.gsub(myString, "%d", "***")
print(newString) -- returns 'My name is John Smith. My phone is ***-
****.'
```

使用这种方式还可以更新电话号码中的区号，例如：

```
custData = "(309) 555-1234"
custData = string.gsub(custData, "%(%d%d%d%)", "(781)")
print(custData) -- prints "(781) 555-1234"
```

可在函数的最后加一个可选参数，用于指定替换的次数，例如：

```
myString = 'happy, hello, hone, hot, hudson'
myString = string.gsub(myString, "h[a+]", "An H word!", 2)
print(myString)
```



在这个例子中，我们查找以 h 开头的字符，%a+ 表示任意长度的字母，并在遇到空格或者标点符号时为止。最后的参数 2 表示只替换最先找到的两个字符串。

## 2. string.gfind(sourceString, pattern)

string.gfind 函数遍历一个字符串，一旦查找到符合指定格式的字符串就返回该子串。

参考下面的示例：

```
myString = "This is my rather long string."
print(myString)
counter = 1
for myWord in string.gfind(myString, "%a+") do
    print(string.format("Word #d: %s", counter, myWord))
    counter = counter + 1
end
```

该例程使用 for 循环控制结构遍历源字符串，%a+ 匹配独立的单词（在解析游戏数据时非常有用）。运行结果如图 5.5 所示。



图 5.5 控制台输出例子中 string.gfind 的结果

## 5.4 table 数据结构

在上一章中，我们了解到 table 非常实用，它可以用在不同的场合。最常用的方式是把它当做其他编程语言的数组。参考下面的示例：

```
myTable = {}
for index = 1,100 do
    myTable[index] = math.random(1,1000)
end
```

在这个例子中，我们使用 for 循环控制结构创建包含 100 个元素的 table。可给每一个元素随机赋 1 ~ 1000 之间的值。我们可以通过 myTable[x] 访问任意元素，x 表示索引。

### 5.4.1 table.getn(myTable)

Lua 提供了很多实用的内建函数来操作 table。首先，table.getn() 返回 table 中元素的个数。如果运行下面的示例：

```
print(table.getn(myTable))
```

我们会得到 100。通常，我们使用 table 来保存游戏数据，但用户不知道表中有多少元素，通过这个函数用户可以得到元素个数这个值，并用来获取 table 中的每个值。参考下面的示例：

```
for index = 1, table.getn(myTable) do
    print(myTable[index])
end
table.sort(myTable)
```

这个简单的函数遍历了整个 table，并从小到大重新排列。还可以添加函数名作为另一个参数，然后通过该函数得到比较的结果（true 或者 false），并根据它来排序。参考下面的示例：

```
function Sort(theTable, direction)
    if direction ~= 1 then
        table.sort(theTable)
    else
        function Reverse(a, b)
            if a < b then
                return false
            else
                return true
            end
        end
        table.sort(theTable, Reverse)
    end
end
```

该函数在 ch5\_3.lua 文件中。

该函数的参数为 table，然后像 table.sort() 一样排序。如果添加第二个可选参数 1，那么该 table 会以相反的方向排序（降序）。

### 5.4.2 table.insert(myTable, position, value)

table.insert 函数在 table 中插入一个新的值，位置参数是可选的，如果没设定，会添加

新的值到 table 末尾，如果指定了该值，则插入到指定的位置。

下面的例子实现插入 hello 到 table 的第 25 个元素的位置，并重新索引。

```
table.insert(myTable, 25, 'hello')
```

### 5.4.3 table.remove(myTable, position)

table.remove 函数从指定 table 中删除并返回一个元素，必要时重新索引 table。如果没有指定 position 的值，则默认删除 table 的最后一个元素，例如：

```
print(table.remove(myTable, 25))
```

如果我们在前面的例子中运行这段代码，将删除 hello 元素，并重新索引 table 为 100 个元素。因为函数会返回被删除的元素，所以输出结果为 hello。

### 5.4.4 table 引用

table 不仅能使用数字索引，还可以使用其他的值作为索引值。参考下面的示例：

```
myData = {}  
myData.name = 'Thardwick'  
myData.class = 'Barbarian'  
myData.str = math.random(3,18)  
myData.dex = math.random(3,18)
```

在这个例子中，我们使用名字作为索引值，它们可以作为获取 table 中值的关键字。该 table 同样还能使用数字索引，我们还能添加下面的数据：

```
myData[1] = 17  
myData[2] = 34  
myData[3] = 24
```

这个方法给了我们很大的灵活性，让 table 不仅保存了很多属性值，还可以保存数组，并通过索引来访问。

### 5.4.5 多维 table

在 Lua 中创建多维 table 非常容易。实际上，在之后的章节中，用户将学习如何使用多维 table 保存和载入游戏进度。可以把多维 table 当做 table 中的 table，可以使用多个关键字访问。参考下面的示例：

```
widget = {}  
widget.name = {}  
widget.cost = {}  
widget.name[1] = "Can opener"  
widget.cost[1] = "$12.75"  
widget.name[2] = "Scissors"  
widget.cost[2] = "$8.99"
```

在这个例子中，我们创建了一个空 table，叫做 widget。然后设定两个属性——name 和 cost，同样是空 table。定义好数据结构后，可以开始添加数据。当引用 widget.cost[1] 时，我们引用了 cost 属性，它指向一个 table，索引是对应该 table 的索引。如果调用 table.getn(widget.cost)，那么会得到 2 这个值，表示该表有两个元素。

### 5.4.6 pairs()

pairs() 函数可以遍历 table 中的每一个元素。参考下面的示例：

```
myNames = {"Fred", "Ethel", "Lucy", "Ricky", "Rockey", "Betsy", "Bill"}  
for index, value in pairs(myNames) do  
    print(index, value)  
end
```

在 for 循环控制结构中，pairs() 函数遍历整个 table（即使不知道长度），并返回索引值和每一个元素的值（参考图 5.6）。



图 5.6 pairs() 例程在控制台中的输出

在下面的例子中，用户可以在 for 循环控制结构中用更传统的方式实现同样的功能，即使不知道 table 的大小：

```
for index = 1, table.getn(myNames) do  
    print(index, myNames[index])  
end
```

`pairs()` 函数在遍历非数字索引的 `table` 时非常有用，参考下面的示例：

```
myData = {}  
myData.name = "Billy"  
myData.interest = "Wind surfing"  
myData.quote = "Cold out, eh?"  
myData.shoesize = 11  
for index, value in pairs(myData) do  
    print(index, value)  
end
```

## 5.5 I/O 基础

在本书中，我们将使用 Lua 来保存和载入游戏的关键信息。在后面的章节中，我们会学习如何完成这个任务，使用 Lua 开发保存/载入系统的最大好处是根本不需要解析游戏数据，Lua 可以帮我们做好它，因为所有的游戏数据会以 Lua 脚本文件的形式表示。

也就是说，我们还需要学习如何输出游戏数据到文件中，以生成这些 Lua 脚本文件。我们需要学习一些关于 Lua 文件输出的功能。

首先，我们需要打开一个文件来输出数据，可以使用 `io.open()` 函数，如下：

```
myFile = io.open("test_data.lua", "w")
```

`io.open()` 函数有两个参数：文件名和输出方式。`w` 代表写模式，如果文件不存在则创建一个文件，同时写入时会覆盖之前的数据。作为游戏进度的保存，我们一般会使用这种方式，因为我们每次都需要保存完整的游戏状态。我们也许会用追加模式，它会保留原来的数据，仅仅在文件的结尾添加新的数据。

如果在打开或创建文件时发生错误，那么返回值是 `nil`。我们在写入文件前，可以通过这个结果来判断是否正常打开文件。参考下面的示例，打开文件并写入若干行数据：

```
myFile = io.open("test_data.lua", "w")  
if myFile == nil then  
    myFile:write("-- Test lua file")  
    myFile:write(string.char(10))  
    myFile:write(string.char(10))  
    myFile:write(string.format("%s%s", "-- File created on: ",  
                                os.date()))  
    myFile:write(string.char(10))  
    myFile:write(string.char(10))  
    myFile:write("print('\\hello world!\\')")  
    io.close(myFile)  
end
```

在这个例子中，我们打开一个文件并使用 `write()` 函数写入数据。`string.char()` 函数写入换行符，这样可以得到便于阅读的结果。`\` 符号可以让我们在 `hello world` 这行添加引号。最后，使用 `io.close()` 函数关闭文件。如果运行该脚本，那么可以生成一个 Lua 文件，可以在控制台使用 `dofile` 命令运行该文件。

## 5.6 本章小结

在本章中，我们学习了一些 Lua 的基础知识，并通过一些函数的学习进一步加深了对字符串和 `table` 数据结构的理解。本章的目的是掌握 Lua 的基础特性，以便让你在之后几章的游戏项目的例子中开始使用 Lua。

我们还学习了如何定义函数、传入参数，并使用 `return` 返回结果。接着我们进一步了解了 Lua 如何处理字符和用 `table` 处理数据。最后，还初步接触了 Lua 内建的 I/O 特性。所有的这些语言特性都将在稍后几章的游戏开发中用到。

首先在控制台中练习本章中出现的例子，然后就可以考虑使用 C++ 来学习如何在独立的应用程序中引入 Lua 并开始体会它带来的好处。



## 第 6 章

# Lua 与 C/C++ 程序的整合

### 本章要点

- 初期设计要点
- 基本实现方式

从 C++ 程序员的观点来看, Lua 像一个“黑盒子”, 为一些服务处理命令和调用。Lua 通常作为最上层接口直接和程序使用者或者游戏玩家打交道, 在核心程序处理之前接受并响应输入。因为 Lua 处在底层代码和用户之间的位置, 所以在设计时会更加注重与 C/C++ 的通信和集成。

## 6.1 初期设计要点

在游戏项目的初期设计阶段, 技术人员应该尽可能地确定每一个可以使用 Lua 的地方。确定这些使用场景是关键的设计目标, 它可以让项目进展更加顺利。实际上, 更多的场景使用 Lua 会收到更好的效果, 因为移除比后期再往里添加更容易。举个例子, 假如有个接口是让 Lua 代码可以直接访问 PC 扬声器, 之后设计变更取消了这个接口, 那就只需要删掉或者忽略它就好。没有程序员或者设计师会抱怨可以控制的资源太多了。

### 6.1.1 Lua 环境

Lua 环境由所有可操作的数据构成, 如编译好的函数、变量以及其他运行时内存。这些数据保存在一个称做 `lua_State` 的结构中。所有 Lua 应用程序都要求至少有一个 `lua_State`, 如果需要还可以有多个 (如需要为两个不同的系统保存不同的数据时)。对于我们来说, Lua 环境是用来发送和接收数据的地方, 它利用栈 (Lua Stack) 来达到该目的。Lua 栈不同于系统栈, 它只能通过 Lua 的 API 函数访问。

Lua 在运行时创建变量并将它们保存在环境中。Lua 语言支持很多类型, 我们主要讨论

用在 C++ 代码中的三个（字符串、数字和函数）。像表（table）和用户数据这样复杂的数据结构我们最好还是只在 Lua 代码中使用。

### 6.1.2 LuaGlue 函数

Lua 可以让程序员开发在 Lua 脚本中调用 C++ 函数的接口。我们把这些接口称做 LuaGlue 函数，因为它们可以在 Lua 环境中整合 C++ 的功能。LuaGlue 函数看起来和一般的 Lua 函数没什么区别，但它们可以访问任何 C++ 程序员开发的模块，是读取和修改 C++ 数据、调用 C++ 函数的主要工具。Lua API 提供了函数让 C++ 代码也可以直接调用 Lua 函数，还提供了方法可以传递字符和长文字给 Lua 解释器。也就是说，C++ 代码和 Lua 脚本之间的交互是双向的，如图 6.1 所示。

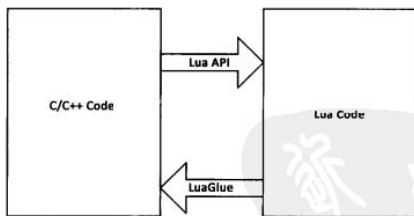


图 6.1 C/C++ 和 Lua 的交互

## 6.2 基本实现方式

Lua 很强大的一个特性就是它可以很容易地嵌入到其他程序中。Lua 的设计者提供了一种简单的接口让用户的程序可以整合 Lua。其中一个例子就是控制台程序，它读取键盘输入并将字符串传递给 Lua 运行环境。完整的源代码可以在随书附送 CD-ROM 的 Lua 文件夹中找到。代码清单 6.1 列出了部分控制台的功能描述。



代码清单 6.1 Lua 控制台的部分代码

```
// Simple Lua console with limited vocabulary
// include the standard system headers.
#include <stdio.h>
```



```
#include <string.h>
/*
** Include the Lua headers.
** Note that they are 'C' language headers.
** Because we are a C++ program, we need to let the C++
** compiler know that the referenced prototypes and data
** will not have C++ namespace processing.
*/
extern "C" {
#include <lua.h>
#include <lualib.h>
#include <lauxlib.h>
}
/*
** This is an example of a LuaGlue function
** The function will be called from the Lua Environment
** Again, since Lua is written in ANSI C, we need to
** turn off the C++ namespace function (extern "C")
*/
extern "C" int _Version(lua_State *L)
{
    puts("This is Version 1.0 of the Console Program");
    puts(LUA_VERSION);
    puts(LUA_COPYRIGHT);
    puts(LUA_AUTHORS);
    return 0;
}
/*
** This is where we hook up our LuaGlue functions.
** The program will register each of the defined functions
** and allow them to be called from Lua.
*/
static luaL_Reg ConsoleGlue[] = {
    {"Version",      _Version},
    {NULL, NULL}
};
char gpCommandBuffer[254];
const char *GetCommand(void)
{
    printf("Ready> ");
    return gets(gpCommandBuffer);
    puts("\n");
}
```

### 6.2.1 创建 Lua 运行环境

第一个任务是创建 Lua 运行环境，可以使用 Lua API 的 `lua_open()` 函数（参考代码清

单 6.2)。

代码清单 6.2 Lua 运行环境的初始化

---

```
lua_State *pLuaState = lua_open();
luaopen_base(pLuaState);
lua_olibopen(pLuaState);
lua_strlibopen(pLuaState);
lua_mathlibopen(pLuaState);
lua_dblibopen(pLuaState);
```

---

函数向 lua\_State 结构返回了一个指向 Lua 运行环境的指针。这个指针很重要，需要传给大部分 Lua API 函数。创建好 lua\_State 结构后，应用程序需要的 Lua 程序库就被初始化好了。在上面的例子中，我们初始化了所有 Lua 5.0.2 版本的程序库，这样可以让 Lua 脚本访问所有的函数，有些程序库可能不会被使用，可以忽略它。

## 6.2.2 添加 LuaGlue 函数

接下来，程序需要注册 LuaGlue 函数，让 C++ 代码可以被 Lua 脚本访问。在代码清单 6.3 中，我们提供了一个简单的函数，用以输出程序和 Lua 的版本信息。这里使用循环的方式注册函数，并且使用了一个简单的数据结构以方便管理和调试 LuaGlue 函数。

代码清单 6.3 LuaGlue 函数初始化

---

```
for(int i=0; ConsoleGlue[i].name; i++)
{
    lua_register(pLuaState,
        ConsoleGlue[i].name, ConsoleGlue[i].func);
}
```

---

## 6.2.3 命令处理

剩下的就是循环处理来自键盘的输入，并将输入字符传递给 Lua 运行环境。代码清单 6.4 所示的 Lua API 函数 luaL\_loadbuffer 载入输入的字符、进行语法检测并且准备好代码以便执行。如果所有的检测都通过并且可以执行，那么函数会返回 0。字符如果被成功载入，就可以使用 Lua API 函数 lua\_pcall 来执行它。如果运行结果正常，就返回 0。如果有错误，则输出 Lua 栈的错误信息，该信息可以使用函数 luaL\_checkstring 来读取。例程输出了“ERROR:”和 Lua 栈的错误信息，方便用户调试。

代码清单 6.4 控制台处理循环

```
const char *pCommand = GetCommand();
while(strcmp(pCommand, "QUIT") != 0)
{
    // send command to the Lua Environment
    if (luaL_loadbuffer(pLuaState, pCommand,
        strlen(pCommand), NULL) == 0)
    {
        if(!lua_pcall(pLuaState, 0, LUA_MULTRET, 0))
        {
            // error on running the command
            printf("ERROR:%s\n",
                luaL_checkstring(pLuaState, -1));
        }
    }
    else
    {
        //error loading the command
        printf("ERROR:%s\n",
            luaL_checkstring(pLuaState, -1));
    }
    // get next command
    pCommand = GetCommand();
}
```

## 6.2.4 退出程序

当用户输入“QUIT”时，处理循环结束，lua\_State 被释放，如下所示：

```
lua_close(pLuaState);
```

这个简单的例子说明了运行 Lua 最基本的要求。在这个例子中，我们只是简单地在 Visual C++ 程序中包含了所有相关的代码并且直接访问 Lua API。尽管对于小的项目这样做是可以的，但更好的方式是抽象出简单的 Lua 程序库，让它更便于整合 C++ 代码，并且可以用在多个项目中。我们接下来就要创建这个程序库，并修改控制台程序来使用它。

## 6.2.5 cLua 对象和 LuaLib

把 Lua 源程序编译成程序库可以让用户的项目更加简洁清晰，让代码的修改更加集中化，也避免了因为 Lua 代码的 bug 修复而改动用户自己的脚本，或者每次需要构建应

用程序时重新编译所有的代码。大部分的程序实现只需要用到很小的一部分 Lua API 函数, 这些函数可以通过类 (class) 封装起来, 隐藏 Lua 需要的初始化和关闭处理。cLua 类满足了这些需求, 并且让用户可以根据需要访问原生的 API。cLua 类的定义可参考代码清单 6.5。

代码清单 6.5 cLua 类的头文件

```
struct lua_State;
#define LuaGlue extern "C" in:
extern "C" {
    typedef int (*LuaFunctionType)(struct lua_State *pLuaState);
};
class cLua
{
public:
    cLua();
    virtual ~cLua();
    bool        RunScript(const char *pFilename);
    bool        RunString(const char *pCommand);
    const char *GetErrorString(void);
    bool        AddFunction(const char *pFunctionName,
                           LuaFunctionType pFunction);
    const char *GetStringArgument(int num,
                                   const char *pDefault=NULL);
    double      GetNumberArgument(int num,
                                   double dDefault=0.0);
    void        PushString(const char *pString);
    void        PushNumber(double value);
private:
    lua_State   *m_pScriptContext;
};
```

构造函数 (constructor) 负责 Lua 运行环境的初始化, 析构函数 (destructor) 用来关闭它。类的 RunString 和 RunFile 方法用来执行包含 Lua 代码的字符串和文件。其他的方法会在具体介绍 LuaGlue 函数、传递参数和返回值时进一步说明。

我们创建了一个包含必要的 Lua 源代码和 cLua 类的代码的程序库 (创建程序库的项目文件在 CD-ROM 的 C++ Code/Lua 文件夹中)。它可以让我们在不同的应用程序中使用相同的代码, 并且让程序的更新和 bug 的修改都只在这个项目中, 然后同步到关联项目中去。该程序库也会在本书的其他例程中被使用。



### 6.2.6 使用 cLua 的例子

更新后使用 cLua 对象的控制台程序见代码清单 6.6, 可以看到它更加简单和轻巧。

代码清单 6.6 更新后的控制台程序

```
// Simple Lua console with limited vocabulary
// Using cLua object and Lua library
// include the standard system headers.
#include <stdio.h>
#include <string.h>
#include <cLua.h>
LuaGlue _Version(lua_State "L")
{
    puts("This is Version 2.0 of the Console Program");
    return 0;
}
char gpCommandBuffer[254];
const char *GetCommand(void)
{
    printf("Ready> ");
    return gets(gpCommandBuffer);
    puts("\n");
}
void main(void)
{
    // print the banner.
    puts("Lua Console (c) 2004 Charles River Media");
    puts("Enter Lua commands at the prompt");
    puts("\'QUIT\' to exit\n\n");
    cLua *pLua = new cLua;
    pLua->AddFunction("Version", _Version);
    // process commands
    const char *pCommand = GetCommand();
    while(stricmp(pCommand, "QUIT") != 0)
    {
        // pass the string to cLua
        if(!pLua->RunString(pCommand))
        {
            printf("ERROR:%s\n",
                pLua->GetErrorString());
        }
        // get next command
        pCommand = GetCommand();
    }
    delete pLua;
}
```

可以看到，例程中不再需要包含 Lua 头文件（除了 cLua.h），并且 LuaGlue 函数的定义也因为使用了 LuaGlue 类型被简化了。创建 cLua 对象代替了初始化 Lua 环境的代码，并且删除了终止 Lua 环境的代码。cLua::AddFunction 的调用添加了 LuaGlue 函数 Version。

对于大量需要注册函数的情况，使用前面例子中的 table 结构的方式更加容易。调用 `cLua::RunString` 函数代替了载入和运行 Lua 缓存，让代码更易理解。

### 6.2.7 LuaGlue 函数的优点

LuaGlue 函数是一种 C++ 函数，按照一定规则开发，可以直接被 Lua 脚本调用。这是 Lua 脚本和 C++ 代码通信的主要方式。当需要调用 C++ 函数时，就通过 LuaGlue 函数来完成操作。对于 Lua 程序员来说很幸运的是，这种函数就和 Lua 函数基本一样。有个很有趣的技巧是可以用 Lua 编写需要的函数，当发现性能瓶颈时再换成 C++ 实现。因为程序调用几乎是一样的，仅仅需要移除原来的 Lua 函数（或者重命名），因此可以开始体会到 LuaGlue 函数的优点。

### 6.2.8 LuaGlue 函数：参数和返回值

让 LuaGlue 函数变得更有一点是其具有向函数传递参数的能力。Lua 通过 Lua 栈来做到这一点。Lua 支持的所有数据类型都可以作为参数传给 LuaGlue 函数。Lua 中的数字等同于 C++ 中的双精度浮点数（double），字符等同于空值终止字符串（null-terminated string）。Lua table 也可以作为 LuaGlue 函数的参数，不过使用 Lua ANSI C API 操作表是一件很麻烦的事，也没有必要。

Lua 还支持一种被称做用户数据（user data）的类型，一般不使用，但是也可以作为参数。cLua 类有获取 LuaGlue 函数的参数的方法。cLua::GetStringArgument 方法返回指向空值终止字符串的指针或者 NULL 表示参数错误。cLua::GetNumberArgument 方法返回双精度浮点数或者 0.0 表示没有参数。两个方法都有一个参数用来指定要获取的参数位置，参数从 1 开始，表示参数列表的第一个参数，后继参数逐级递增 1。

返回值也使用 Lua 栈处理。向函数调用返回数值时，就把该值存于 Lua 栈中，并且在 C++ 返回语句中指定返回值个数。注意，这里的返回值个数是指 C++ 向 Lua 运行环境返回值的个数。Lua 函数可以返回多个值，这个特性十分有用，LuaGlue 函数也支持。很重要的一点，返回值的个数一定要和添加到 Lua 栈中的返回值个数一致。调用 `cLua::PushString` 函数或者 `clua::PushNumber` 函数可以向 Lua 栈中存入返回值。

最好定义一个局部变量来保存参数个数，并设初始值为 1，每当从 cLua 对象读取了一个参数后就加 1。另外还建议每次都返回相同个数的返回值，不够就用默认值代替，这是

为了避免遇到 nil 值的错误。在后面的章节中，我们会继续讨论错误处理。参考代码清单 6.7。

代码清单 6.7 取得和返回数值的示例

```
extern cLua *pLua;
LuaGlue _SwapExt(lua_State *L)
{
    int argNum = 1;
    const char *fileName =
        pLua->GetStringArgument(argNum++);
    const char *newExt =
        pLua->GetStringArgument(argNum++);
    char fname[_MAX_FNAME];
    char ext[_MAX_EXT];
    _splitpath(fileName, NULL, NULL, fname, ext);
    std::string sRet = fname;
    if(newExt[0] != '\0')
    {
        sRet += ".";
        sRet += newExt;
    }
    lua->PushString(sRet.c_str());
    return 1;
}
```

函数首先从 Lua 运行环境中获取参数。这个例子中，参数是两个 char 字符串，使用 cLua::GetStringArgument 函数获取。完成了字符串的操作后，调用 cLua::PushString 方法存入处理后的结果，并且返回 1（返回值的个数）。

## 6.3 本章小结

Lua 的设计者创造了一个灵活、易移植的系统，从而使得它可以很容易地集成到 C 程序中。本章的例子向大家展示了如何整合 Lua 和 C++ 程序，并且隐藏了一些复杂、难以理解和维护的代码。这些仅仅是进入 Lua 游戏开发世界的第一步。

在下一章中，我们会学习一个 C++ Lua 项目的技术设计，并且使用一些代码风格让 Lua 脚本更加易读。

## Lua 与 C++ 的交互

### 本章要点

- 重新审视 LuaGlue 函数
- C++ 代码和 Lua 的交互
- 事件驱动的编程
- 错误处理

虽然 Lua 本身是功能十分丰富的运行环境，但是如果 Lua 脚本不能与 C++ 代码交互，就做不了实际的工作。在本章中，我们要继续开发在第 6 章中已经出现过的基础框架，并且要实现一个可以让 Lua 和 C++ 交互的系统。

### 7.1 重新审视 LuaGlue 函数

在第 6 章中，我们初步了解了 LuaGlue 函数的概念，这个重要的内容是构建稳定的交互系统的基础。LuaGlue 函数是让 Lua 脚本访问 C++ 函数的接口。C++ 程序员为 Lua 程序员开发的 API 促成了这门脚本语言强大的能力，C++ 程序员可以提供所有类型的数据和功能给 Lua 程序员，并且让程序接口易于调试。对资源的控制和访问在这个系统中是很容易调整的。

我们将会探讨 LuaGlue 函数一个关键的特性，即它看起来就像普通的 Lua 函数一样。要做到这点，可以先使用 Lua 开发函数，再改编成 LuaGlue 函数，并注释掉 Lua 函数。关键的处理可以在 Lua 的快速开发环境中测试和完善，之后再将它们移植到运行效率更好的 C++ 环境。

还要注意的一点是，LuaGlue 函数应该比标准 Lua 函数有更详细的文档说明，因为脚本程序员可以通过查看 Lua 函数的脚本来知道需要传递什么参数，但没有直接的方法可以知道 LuaGlue 函数的相关信息。



有一个好办法是创建一个文档，里面记录着所有的 LuaGlue 函数以及它们的参数信息，甚至还包括在每个项目中的示例。这个文档会成为脚本程序员非常重要的参考工具，特别是当他们过了很久之后再次要使用这些函数的时候。

## 7.2 C++ 代码和 Lua 的交互

LuaGlue 函数给 Lua 脚本提供了一种调用 C++ 代码的方法，但是 C++ 代码该如何调用 Lua 脚本呢？Lua 提供了一个系统，让 C 代码可以直接调用 Lua 函数，不过这种方法很笨拙，它依赖于 Lua 脚本中定义的特定名字，不是很灵活。本书中的例子会使用一种事件驱动的模式来与 Lua 代码交互。这种系统更加便于修改和扩展，并且让 Lua 的交互处理更加集中。下面是 Lua 可以处理的一些事件：

- 键盘输入和鼠标事件
- 特定的游戏事件（触发器被按下、游戏角色死亡等）
- GUI 相关的事件（按下按钮）
- 计时器超时

## 7.3 事件驱动的编程

如果要使用事件驱动的系统与 Lua 交互，那么 C 程序需要知道被调用的 Lua 函数，并向 Lua 传递事件。我们提供了一个 LuaGlue 函数来设定 Lua 事件处理程序的名字，然后 Lua 调用它来开始事件流。通过这种方式，Lua 程序员可以控制事件处理函数的名字和位置。还可以让它更容易扩展，一致认为事件驱动的程序可以增强系统可扩展性。

### 7.3.1 示例事件

为了解释这个概念，参考下面的例子：

```
EVENT_SAMPLE = 1000
RegisterEvent('EventHandler')
function EventHandler(id, ...)
    if id == EVENT_SAMPLE then
        print('Sample Event!')
    end
end
end
```

如果运行上面的脚本，C++ 代码就可以向代码发送事件并且让 Lua 来处理。RegisterEvent 函数是 C++ 代码编写的 LuaGlue 函数，保存之后使用的事件处理程序名，该函数的代码参考代码清单 7.1。

代码清单 7.1 RegisterEvent 函数和 FireEvent 函数的例子

```
#define EVENT_SAMPLE 1000
std::string g_strEventHandler = "";
extern "C" int _RegisterEvent(lua_State *L)
{
    g_strEventHandler = g_pLua->GetStringArgument(1, "");
}
void FireEvent(int id)
{
    if(g_strEventHandler != "")
    {
        char buf[254];
        sprintf(buf, "%s(%d)", g_strEventHandler, id);
        lua_dostring(buf);
    }
}
```

注意，这里只是为了编写一段简单、短小的脚本，忽略了 Lua 系统的一些健壮性要求。这个方法现在是有问题的，稍后我们会增加错误处理，让程序更加健壮、稳定。

### 7.3.2 事件的参数

可以使用 Lua 的变量参数系统为每个事件设定不同的参数，并传给相应的事件处理程序。在前面的例子中，EventHandler 函数的第一个参数是事件 id，第二个参数是“...”。这表明以“...”开头的所有的参数都会保存到名称为 args 的局部数组中。可参考下面的 Lua 代码：

```
EVENT_SAMPLE = 1000
function EventHandler(id, ...)
    print(string.format("id = %d\n", id))
    for i = 1, arg.n do
        print(string.format('arg[%d] = ', i))
        print(arg[i])
        print('\n');
    end
    if id == EVENT_SAMPLE then
        print('Sample Event!')
    end
end
EventHandler(EVENT_SAMPLE, 100, 'next arg', 'lastarg')
```

如果在控制台程序中运行上面的脚本，会输出和下面的例子一样的结果：

```
id = 1000
arg[1] = 100
arg[2] = "next arg"
arg[3] = "lastarg"
```

发送事件的 C++ 函数也需要修改，用来处理包含参数列表的字符串。见代码清单 7.2。

代码清单 7.2 传递参数的例子，C++

---

```
void FireEvent(int id, const char *args)
{
    if(g_strEventHandler != "")
    {
        char buf[254];
        if(args)
            sprintf(buf, "%s(%d,%s)", g_strEventHandler, id, args);
        else
            sprintf(buf, "%s(%d)", g_strEventHandler, id);
        lua_dostring(buf);
    }
}
```

---

这个系统需要注意的地方在于，Lua 代码和 C++ 代码需要对事件 id 的值和传递的参数含义保持一致。这点付出是值得的，它可以让函数更容易使用。

## 7.4 错误处理

在开发中，脚本错误是非常普遍的。如果没有工具可以让 Lua 程序员知道是什么原因导致了程序异常终止的话，那么完成项目开发是一件不可能的事情。

标准的 Lua 错误处理系统会在控制台输出错误消息。（可以在控制台程序里试试：输入一些错误代码看看会有什么提示信息。）但如果不是在文本控制台环境（如游戏）下的话，这种方法就不行了。一种解决方式就是在游戏中提供一个控制台用来查看错误消息，它还可以用来输出变量和运行函数。不过这种方法对于最终用户的使用来说是不能接受的。

我们会在之后的章节中添加这个功能，只在调试版本中提供调试控制台。我们决定处理错误的方式是在 Lua 中捕捉它们，然后再抛出 C++ 异常。异常处理程序会打开一个标准的对话框输出 Lua 提供的消息然后再关闭程序。从 C 调用 Lua 时，采用了一种“受保护的调用”的方式，等同于前面例子中的 lua\_dostring 类型的调用。Lua 不再使用它正常的错误

处理系统，而是返回错误代码和字符串给主调函数。error 对象的类可参考下面的示例：

```
class CError
{
public:
    CError();
    ~CError();
    virtual
    void AppendLocation(const char *fileName, int lineNum);
    void AppendMessage(const char *formatString, ...);
    void Report();
private:
    std::list<std::string> m_lstMessages;
};
```

当 Lua 返回错误时，会创建 CError 对象，返回的错误字符串会追加错误发生地点的相关信息。主程序包含在一个大的“try...catch”程序块中，在 catch 中会调用 CError Report 方法（显示对话框），然后结束程序。这个方法高效地捕捉所有的 Lua 错误（或者用户想抛出的错误）并报告它们。

## 7.5 本章小结

在本章中，扩展了在第 6 章接触到的 LuaGlue 函数的概念，并且开始学习用事件驱动的方式让 C++ 程序和 Lua 脚本环境交互。这种双向交互的方式在游戏开发中非常重要，因为为用户交互（例如玩家使用鼠标、通过按键的交互，或者输入文字）是游戏体验的基础。

学习如何开发自己的 LuaGlue 函数，让我们能从不同的角度来解决游戏开发中可能遇到的问题。LuaGlue 函数让我们可以编写在 Lua 中调用的 C++ 函数，从而获得编译为原生机器代码带来的性能优势。Lua 语言本身允许我们编写脚本级别的易于开发、测试和修改的函数。还可以使用 Lua 开发原型函数，之后再转换成 C++ 函数，并通过 LuaGlue 函数来调用。

利用本章的知识点，我们就能够使用 Lua 来驱动游戏的界面、图形、音效和游戏数据，开发出稳定的系统。

## 第8章

# 开发准备

### 本章要点

- Visual C++ 6.0 工作区
- DirectX 基础
- LuaGUI 简介
- 调试窗口
- Windows 注册表

Lua 是一种嵌入式语言，这意味着要发挥它的作用，需要依赖能够给它提供应用环境和程序框架的程序。本章将主要讨论程序框架，也称做 shell，我们会在本书的所有示例，当然也包括之后几章要开发的示例中使用它。

## 8.1 Visual C++ 6.0 工作区

本书中所有的示例都包含在 Visual C++ 6.0 工作区（Workspace）中。Visual C++ 6.0 工作区如图 8.1 所示。

工作区包含所有的示例和用到的程序库。Base 项目是之后章节的游戏基础库。Chapter8 是本章会讨论的项目。两个控制台示例的源代码分别放在 Console1 和 Console2 文件中。DX9 库包含所有与 DirectX 相关的代码，为其他示例提供了 DirectX 的功能。lua 项目包含 Lua 5.0.2 版本的文件（来自 [www.lua.org](http://www.lua.org)）和 cLua 类的文件。LuaGUI、Take Away 和 TicTacToe 项目会在之后的章节中说明。

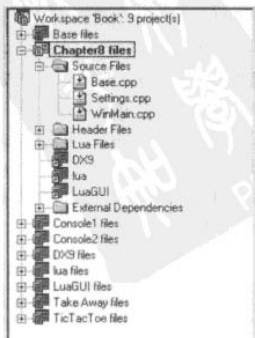


图 8.1 Visual C++ 6.0 工作区


## 8.2 DirectX 基础

DirectX 是由微软公司开发的一系列 API，它让程序员可以编程访问计算机的硬件资源，而不需要知道具体的硬件细节和驱动程序（如显卡、游戏手柄或者声卡）如何运行。DirectX 通过一个中间层将共通的硬件命令转换成和具体硬件相对应的命令来完成这些工作。

对于用户来说，DirectX 让游戏程序员可以在一个标准的平台中开发程序，把硬件的运行细节交给第三方厂商（如显卡生产商）的驱动来控制。对于本章的示例，我们不需要深入了解 DirectX 的细节，如果想要学习 DirectX 的更多信息可以参考互联网上的文档和技术指南，如微软 DirectX 开发者中心 [www.msdn.microsoft.com/directx](http://www.msdn.microsoft.com/directx)。

本章的例子是一个简单的 shell，用来实现 DirectX 的功能并提供 Lua 环境和基于 Lua 的 GUI 系统。shell 程序启动 DirectX 图形和音频功能，使用 Windows 的标准鼠标、键盘输入，还支持游戏手柄和其他的输入设备，实现了 DirectX 输入。DX9 标准库对于任何 DirectX 项目来说都是十分合适的选择。

### 构建 DirectX Shell 程序

第一个示例使用 DirectX 库、cLua 库和 LuaGUI 库来开发一个简单的游戏 shell 程序，见  代码清单 8.1。这个基础的 shell 会用在本书后面章节的示例中。该 shell 程序的 C++ 代码在本书附赠光盘的“Chapter8”项目中。示例引用的代码是 Base.cpp 文件和 CBase::Init 函数。

代码清单 8.1 shell 初始化

```
bool CBase::Init(HINSTANCE hInstance, const char *szClass,
                const char *szCaption, WNDPROC WindowProc)
{
    m_lpDX = new DXContext(hInstance, szClass, szCaption, WindowProc);
    if (!m_lpDX->Init(800, 600, 16, 2, TRUE)) {
        delete m_lpDX;
        return false;
    }
    m_pSettings = new Settings;
    m_pSettings->Init("Chapter8");
    for(int i=0; MyGlue[i].name; i++)
    {
```

```
m_lua.AddFunction(MyGlue[i].name, MyGlue[i].func);
}
m_hConsole = CWinConsole::StartConsole(hInstance, &m_lua);
if(m_hConsole)
{
    int x = m_pSettings->GetInteger("DebugWinX");
    int y = m_pSettings->GetInteger("DebugWinY");
    int w = m_pSettings->GetInteger("DebugWinW");
    int h = m_pSettings->GetInteger("DebugWinH");
    if(w == 0 || h == 0)
    {
        RECT r;
        GetWindowRect(m_hConsole, &r);
        w = r.right - r.left;
        h = r.bottom - r.top;
    }

    SetWindowPos(m_hConsole, HWND_BOTTOM, x, y, w, h, 0);
}
m_pGUIManager = new CGUIManager;
m_pGUIManager->Init(&m_lua);
return true;
}
```

该函数完成了下列处理：

- 新建 DXContext (DX9 标准库)
- 设置 DirectX Graphics 像素为  $800 \times 600 \times 16$ ，两个备用缓存
- 创建 Settings 接口
- 注册特定于应用的 LuaGlue 函数
- 初始化调试窗口
- 启动 LuaGUI 系统

### 1. 启动 DirectX Shell 组件

shell 的初始化从启动负责显示的 DirectX Graphics 组件开始。程序在 Base.cpp 中调用，实际的处理代码在工作区的 DX9 项目中。代码在微软公司提供的例子基础上做了一些修改，可以让程序运行在窗口或者任意屏幕尺寸的全屏模式下。DirectX Audio 组件也会启动，这样就可以访问相应的 DirectX API 的音频部分。Windows 消息系统提供了鼠标和键盘功能，以后还可以启动 DirectX input 来支持手柄和其他输入设备。DirectX 还提供了 GUI 系统需要的两个对象：子画面 (Sprite) 和字体。这些对象原来都包含在微软公司提供的示例中。

## 2. 创建 Settings 接口

创建 CSettings 对象，并告诉它在 Windows 注册表的什么地方存储信息。更多关于 CSettings 和 Windows 注册表的信息可以查看 8.5 节。

## 3. 初始化 Lua 环境

在前面控制台的示例中，我们已经知道如何使用 cLua 对象来启动 Lua 环境。在启动 GUI 系统之前一定要先初始化 Lua 环境并注册游戏相关的 LuaGlue 函数，因为 GUI 系统需要 cLua 对象来正确运行，并需要那些与游戏功能相关的函数。cLua 对象的代码在 Lua 项目的 cLua.cpp 文件中，本书通篇使用的 Lua 发布版 5.0.2 的源代码也包含在这个项目中。

## 4. 创建 Lua 调试窗口

调试窗口只是简单地把前面控制台的例子迁移到窗口中，并在这里打开。窗口的位置和尺寸信息通过 CSettings 对象保存在 Windows 注册表中。有关窗口的详细信息可阅读后续章节。

## 5. 启动 GUI 系统

定义了所有 GUI 操作需要的 LuaGlue 函数的 GUI 系统已启动了。在处理过程中，Lua 脚本会执行。（LuaGUI 系统的具体细节可参考第 11 章。）初始化完成后，GUI 系统就可以使用了。GUI 标准库在 LuaGUI 项目中，之后的章节会详细讨论这个项目。

## 6. 终止程序

当终止程序时，需要关闭在 CBase::Init 函数中启动的系统。该步骤的实现在 CBase 的析构函数中。下列代码关闭了调试窗口和所有的 DirectX 系统：

```
CWinConsole::StopConsole();  
if(m_lpDX)  
{  
    m_lpDX->Cleanup();  
    delete m_lpDX;  
}
```

# 8.3 LuaGUI 简介

所有的例子都是基于 DirectX 基础库开发的，该基础库支持游戏系统的核心 LuaGUI 标



准库。我们将从子画面（sprite）和 GUI\_EVENT\_KEYPRESS 事件开始介绍 LuaGUI 的基础知识。

### 8.3.1 启动 GUI

主程序启动 GUI 系统是通过创建 CGUIManager 对象和调用它的 Init 方法访问 cLua 对象实现的。CGUIManager 定义在工作区的 LuaGUI 项目中。GUIManager.h 文件包含了类的定义，该文件保存在 CD-ROM 的“C++ Code/includes”文件夹中。Init 方法初始化 GUI 所有相关的 LuaGlue 函数，并执行一个特殊脚本 StartGUI.lua。该脚本负责 Lua 需要的初始化处理，还可以用来定义用户需要的常量值和 Lua 的通用工具类函数。它还调用 LuaGlue 函数 RunGUI 来启动显示给用户的第一个界面。RunGUI 是 GUI 系统很大的一部分，它负责界面之间的迁移。RegisterEvent 函数告诉 GUI 系统事件处理 Lua 函数的名字并启动事件流。GUI 系统内部会保存每次运行程序时创建的界面，通过 RunGUI 来重新启动这些界面而不是重新执行相关脚本。

### 8.3.2 界面

在讨论 GUI 系统的过程中，我们不断提到“界面”和“界面对象”，它们是指所有 RunGUI 命令启动的界面。界面和界面脚本/事件处理程序是一一对应的关系。我们可以把界面当做游戏中的一个画面。一旦创建界面，重新显示的时候就不需要再次载入相关资源。在 C++ 代码中，界面被定义成 CUserInterface 对象。类的定义参考 LuaGUI 项目的 UserInterface.h 文件。

### 8.3.3 界面控件

GUI 控件（如按钮）是组成界面的可视化组件，它们是 C++ 程序中的预定义对象，可以被 Lua 代码任意创建和使用。在后面的章节中会定义许多 GUI 控件。我们不会涵盖所有可能的控件，因为它没有上限的。常用的控件有按钮、子画面（sprite）和列表，这些控件以及其他的控件都可以添加到 GUI 系统中。

GUI 控件都继承于 CGUIObject 类，定义在 LuaGUI 项目中。因为所有控件都基于 CGUIObject，所以拥有一些共通的基本功能。它们都有独立的子画面和字体。子画面是最简单的例子，因为基类本身包含了显示子画面的所有处理。文本对象可以使用子画面作为背

景,并用基类的字体显示。这种 CGUIObject 的实现依赖于 DirectX 抽象的子画面和字体类。如果底层的图形 shell 程序有了改变,那么新的系统也需要支持这些类,以便让 GUI 系统正常工作。

### 8.3.4 事件

第 7 章讨论了事件驱动编程和系统。现在开发的 GUI 系统就是第一个采用这些概念的系统。C++ 程序会触发事件, Lua 程序响应事件或者忽略。在本章的例子中,事件处理程序只响应唯一一个事件: GUI\_KEY\_PRESS。当事件触发时,脚本会调用 LuaGlue 函数 QuitProgram。具体的程序可参考代码清单 8.2。

代码清单 8.2 Lua 脚本例子

---

```
StartGUI.lua:
-- define constant values for all scripts
-- Standard LuaGUI event codes
GUI_KEY_PRESS = 4

RunGUI("MainMenu.lua")
-- END of StartGUI.lua

MainMenu.lua:
RegisterEvent('EventHandler')
function EventHandler(id, _)
    if id == GUI_KEY_PRESS then
        QuitProgram()
    end
end
end
-- END of MainMenu.lua
```

---

StartGUI.lua 文件是 GUI 脚本系统的入口。用它来定义包含 GUI\_KEY\_PRESS 事件代码的变量。这个代码可以让 Lua 程序员在程序中使用更易懂的名字,并且在 C++ 程序由于某种原因改变了事件的值的时候(一般不会),让我们可以很容易做出相对应的改变。然后让 GUI 系统执行 MainMenu.lua 脚本。该脚本创建了一个简单的界面,只响应唯一的事件(GUI\_KEY\_PRESS),响应的结果是关闭程序。

### 8.3.5 与 GUI 系统相关的 LuaGlue 函数

我们现在开始介绍与 GUI 系统相关的一些 LuaGlue 函数。随着我们介绍 GUI 系统的其他功能,这个列表会逐渐增长。

### (1) RunGUI ( “interface. lua” )

RunGUI() 函数执行传入的 Lua 代码文件，指示 GUI 系统创建一个新的界面，或者重新启动之前的界面。这个界面会变成当前界面。这个 Lua 文件应当通过 LuaGlue 函数 RegisterEvent 定义并注册事件处理程序。

### (2) RegisterEvent ( “eventFunction” )

RegisterEvent() 函数告诉 GUI 系统调用相应函数来响应当前界面发出的 GUI 事件。每个界面都需要事件处理程序来处理相关事件。

## 8.3.6 Shell 程序的扩展

这个基础库被设计成易于扩展的。扩展性的一个例子是 CBase::Init 方法（参见代码清单 8.1）中的循环，其中注册了特定于程序的 LuaGlue 函数。所有添加到 MyGlue 数组的函数都会在 Lua 中自动注册。本章的例子中，只有一个新的 LuaGlue 函数，但是它很重要。

### QuitProgram ( )

该函数用来退出应用程序。在主程序而不是 LuaGlue 标准库中定义该函数可以让每个程序根据自己的需要控制退出。QuitProgram 函数的代码在 Base. cpp 文件中，向 Windows 发送消息退出程序。该消息在 WinMain. cpp 文件中处理。

#### 扩展的思想

基础库可以用不同的方式扩展，例如：

- 1) 3D 功能。
- 2) 更多的 LuaGUI 控件。
- 3) DirectInput 功能以支持手柄。
- 4) 资源缓存系统。
- 5) 全屏模式和窗口图形模式的切换。
- 6) 动画系统。
- 7) 粒子效果系统。
- 8) 物理引擎。
- 9) 其他游戏功能扩展。

## 8.4 调试窗口

在第7章中，提到允许用户访问控制台是一个很好的想法。在开发游戏基础 shell 程序时，会包括用于调试的控制台，在这个 shell 程序基础上开发的 debug 版本的游戏都可以使用它。前面章节中的文本控制台应用程序和调试控制台的区别在于前者是基于文本的，而后者是一个具有输入/输出功能的窗口程序。调试窗口仅仅在 debug 版本中可用，因为不希望玩家使用这个强大的功能。

调试窗口是个小型的独立窗口程序，它有自己的窗口、消息处理程序和控件。用户甚至可以单击关闭按钮关闭它。主程序通过实例化 CWinConsole 对象创建调试窗口。只有使用常用的 Lua 文本才能直接把调试窗口连接到主程序，使用 CWinConsole 方法在窗口中插入文本和 print LuaGlue 函数的实现。在窗口的底部有一个输入控件，剩下的部分用来显示 Lua 和 C++ 程序的消息。输入区域检测到 <Enter> 键按下时，会向窗口发送输入的命令，输出区域会显示 Lua 或者主程序向对象传递的信息。需要注意的是，控制台处理错误不同于主 C++ 程序，它不会产生 C++ 异常，只是简单地在输出区域显示 Lua 的错误信息。

在查看 Lua 变量值时，调试窗口的作用十分明显，另外一个强大的功能是可以调用 Lua 函数显示变量，甚至使用 Lua 文件 I/O 标准库转储 (Dump) 它们。一个调试的技巧是可以使用 print 语句来标记调试进度。print 语句生成的字符串会显示在调试窗口中，在运行时，用户甚至还可以立即修改 Lua 函数或者更改 Lua 变量的值来看看对主程序会产生什么样的影响。

调试窗口的源代码在 DX9 项目的 WinConsole.cpp 文件中。

## 8.5 Windows 注册表

Windows 提供了一个中心存储库来保存与程序相关的数据，这些数据在程序运行时必须保存，这就是“注册表”。在基础 shell 程序中，想使用一种简单的获取注册表中数据的方式。注册表通过“键” (key) 来保存不同的数据，程序可以通过“键”来读取和写入值。每个程序都在注册表中创建自己的区域来保存这些“键”。微软公司提供的 API 不是很难，但是最好能统一和标准化访问数据的函数。我们会创建一个被称做 CSettings 的类负

责所有和注册表相关的处理。为了简单起见，该类只提供了两种保存数据的方法，它支持字符和整数。参考代码清单 8.3。

代码清单 8.3 CSettings 类

---

```
class Settings
{
public:
    Settings();
    virtual ~Settings();
    void Init(const char *baseKey);

    int GetInteger(const char *key);
    std::string GetString(const char *key);

    void SetInteger(const char *key, int value);
    void SetString(const char *key, const char *value);

protected:
    std::string m_baseKey;
};
```

---

在创建 CSettings 对象时，需要调用 Init 方法来告诉对象在 Windows 注册表中的键的位置。初始化完成后，可以调用相关的 Get 和 Set 方法来获取和保存数据。CSettings 类的完整源代码在“Chapter8”项目的 Settings.cpp 文件中。

## 8.6 本章小结

在本章中，开发了第一个基于 DirectX 并集成了 Lua 的应用程序。基础库建立在 Microsoft DirectX API、Windows 注册表、GUI 系统、调试控制台和 Lua 之上，它会用在所有的示例中，也是开发自己项目的一个很好的起点。DX9 标准库可以用来简化开发，它隐藏了 DirectX 内部的处理。我们还介绍了 LuaGUI 标准库的一些重要概念，以便为之后的详细介绍做好准备。前面章节中的文本控制台程序升级为调试控制台不仅可以显示错误消息，还能在运行时输入代码进行调试。本章最后介绍了如何在 Windows 注册表中存储数据。

## 设计 Lua 版本的实现

## 本章要点

- 游戏设计原则
- 基础库设定
- 设计文档
- Lua 编程规范

在正式编写游戏脚本之前，我们需要准备好一个完善的基础库。首先决定开发的游戏类型，并以此确定需要在开发过程中创建的文件和脚本。有了这个基础，我们就可以开始编写设计文档，并为第一个使用 Lua 开发的游戏制定蓝图。

## 9.1 游戏设计原则

在考虑游戏脚本的细节之前，我们应当花些时间思考一下游戏的含义。“游戏”是一种娱乐体验，引导玩家（通过规则和事件）收获某种结果（通常是游戏胜利、失败或者平局）。

客观来说，游戏不一定是有趣的（如那些训练军人作战的战争模拟类游戏），但在我们（游戏开发者）看来，为玩家创造非凡的游戏体验是我们真心的追求。

### 9.1.1 什么是游戏

根据过去十多年的游戏开发经验，我们给出的比较正式的定义如下：

游戏是一种包含一系列操作的娱乐活动,有一定的游戏规则,并设定了结束条件。游戏规则定义了游戏行为的结构和前后关系,并为玩家创造有挑战性的环境。玩家的行动、决策、选择和变化是游戏流程的一部分,或者是游戏旅程的一部分。游戏的成功与否取决于它的品质、挑战性、兴奋点和趣味性,而不仅仅局限于是否达成游戏结束条件。

再进一步解释，玩家通过游戏中的行动参与游戏，它可以是操作手柄躲避敌人或者掷色子移动“大富翁”（Monopoly）棋盘上的棋子。这些操作都受到游戏规则的限制。因此，在“大富翁”游戏中，我们不能任意移动棋子，而必须遵守规则掷色子。

游戏规则限制了玩家的操作，这是我们创造虚拟世界，并让玩家融入其中的一种机制。游戏规则决定了游戏的完整性和可玩性，因此规则本身应该是合理的，能够让玩家在游戏世界中一定程度上掌握自己的命运。

这里我们触及了游戏的关键：在互动的游戏中，玩家应该能够控制游戏的进程。他们应该知道游戏的目的，不论是射击外星人、躲避障碍还是移动物体。

在开发游戏时，必须时刻提醒自己游戏是为玩家开发的。他们的体验是最重要的，而不是坦克模型做得有多好或者爆炸效果有多逼真。我们所做的一切都要从玩家的角度出发，展示的一切都要以游戏性和引导玩家完成游戏为核心，而不应该浪费时间在那些和游戏性不相关的部分上。

### 9.1.2 了解玩家的想法

在设计游戏规则和让玩家挑战之前，了解一些玩家的心理是十分重要的。有一些基本的原则：首先，玩家都不希望失败，他们喜欢获胜的感觉。更确切地说，比起很容易的获胜，他们更喜欢有挑战性的，享受那种从失败的教训中获得的最终胜利。这种感觉是我们渴望获得成功的一种本性，也是成功时收获的那种兴奋与激动。

因此，我们设定游戏规则就是要创造出一种机制，让玩家可以不断尝试和挑战，通过努力收获令人激动的胜利。

说点有意思的题外话（会成为我们游戏项目的关键），在早期开发街机游戏的时候，阿塔利（Atari）的开发者制作了一份游戏设计“宝典”，是关于如何让那些街机玩家上瘾的设计元素。其中的一个设计方式就是没有终点的游戏（如爆破彗星 Asteroids 这款游戏）：游戏永远不会结束，除非玩家失败。

表面上看起来这是个相当愚蠢的设计，但真正关键的部分在于设计师必须让玩家能感受到，如果他可以在某个环节提高一些，或者尝试不同的玩法，他就可以在下一局中得到更好的结果。就是这种挑战自我的感觉让玩家想不断地尝试，花费大量的硬币在无止境的、没有胜利条件的游戏中。

另外一个有关玩家心理的关键元素：玩家喜欢游戏过程中有许多成就提示，比起游戏

结束时只有一个胜利的提示，游戏过程中时常出现的小成就更让人欣喜。这和旅行是相似的，玩家不只是为了一个结局来玩游戏，他们更注重享受游戏过程中的体验。

有了这些想法，我们就来设计一种游戏体验，让玩家可以在挑战最高分和最终通关的游戏过程中，收获不同的成就。这种设计可以看成是“嵌套式游戏”。以很受欢迎的街机游戏 Star Wars（第一款使用数字化语音的游戏）来说，在游戏中，需要玩许多小游戏：射击 TIE 飞机、攻击炮塔、战壕追逐，最后摧毁死星。每一关都包含了很多挑战和小成就，但它们都围绕着镇压游戏中的叛乱这个最终目的。在 PC 游戏中，那些动作游戏都是由一系列关卡组成：在每个关卡中的获胜作为小成就推动着游戏的进程，直到最终通关。

最后，很重要的一点是，游戏设计者要知道玩家在游戏时的感知是有限的。因此，完整的游戏体验应该和玩家看到的、听到的保持一致。只有注意到了这点才能合理地设计游戏的画面和音乐（包括声效和背景音乐），以此增强玩家的游戏体验，而不是让他们感到困惑。

在设计游戏时，不仅要以玩家体验为核心，并要以此为依据来判断设计。同时，还需要考虑一些技术因素，如目标平台、进度安排、团队组成和游戏的哪些组件适合用 C++，以及哪些部分适合用 Lua 脚本。

## 9.2 基础库设定

在开发集成了 Lua 的游戏项目时，我们首先要确定一些边界条件：什么部分使用 Lua 脚本，什么模块使用传统的 C++ 开发。

这个确定过程十分简单。因为 C++ 可以实现的且很好的一些功能 Lua 是不能做的，如画面的渲染。在做进一步的工作之前，我们需要有一个明确的目标。

首先，创建一个开发范围的文档：列举出我们要开发游戏的基本功能。根据这份文档我们就可以知道之前问题的答案，并创建一份设计文档作为开始正式编程前的技术蓝图。

在本书中，我们将要开发一款动作射击类的街机游戏，它在一定程度上继承了老的游戏 Rip Off，这里称做《Take Away》（参考图 9.1）。我们从概要开始，用一两句话描述这个游戏：

在《Take Away》的游戏中，玩家会控制一艘小飞船，对抗一伙偷箱子的敌人。玩家可以飞行、会轰击敌人，并尽可能让自己生存更久。当箱子都被偷走时，游戏结束。

概要设计包含了我们游戏最初的想法，并规定了基本的玩法：轰击敌人和生存。有了



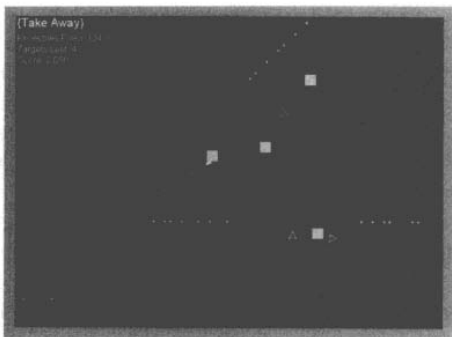


图 9.1 《Take Away》运行时的截图

概要设计，我们就可以开始编写开发范围文档。当然，对于这样一个简单的游戏，开发范围文档是简洁明了的。对于大型的商业游戏来说，也许得用三四页文档来列出全部的开发内容。如果还有更长的开发范围文档，就要考虑是不是写了过多细节内容。

我们游戏的开发范围文档见表 9.1。

表 9.1 《Take Away》的开发范围文档

游戏名	Take Away
平台	PC
玩家数	单一玩家
类型	动作/街机
游戏概要	在《Take Away》的游戏中，玩家会控制一艘小飞船，对抗一伙偷箱子的敌人。玩家可以飞行、会袭击敌人，并尽可能让自己生存更久。当箱子都被偷走时，游戏结束
游戏目标	目标是积累尽可能多的分数。玩家要保护 8 个箱子不被敌人移动到屏幕外，当箱子都被偷走时，游戏结束
游戏特性	<p>屏幕区域是整个游戏场景，玩家不能移动到屏幕外</p> <p>玩家控制飞船，可以加速、转向和射击（游戏中没有重力，因此移动不会停止）</p> <p>当最后一个箱子被拖到屏幕外时游戏结束</p> <p>玩家击败敌人后会得到分数</p> <p>敌人会尝试偷窃箱子并攻击玩家，甚至撞击玩家</p> <p>游戏速度会随着时间的增长而加快</p> <p>玩家可以在任何时候保存游戏</p>

有了这个开发范围文档，我们就可以考虑如何使用 Lua 来实现它。通常可以从用户使用感受的角度出发，来考虑界面元素和想要呈现给用户的东西。

一般游戏开始都是启动画面或者载入画面。该画面可以让用户知道正在载入的游戏，并在等待的过程中提供一些绚丽的效果。在我们的例子中，游戏的载入是瞬间完成的，但我们仍然需要一个载入画面来吸引玩家。

在表格中列出接口画面，这样就可以一目了然。参考表 9.2 的例子。

表 9.2 《Take Away》的接口画面

Lua 文件	调用位置	下一步处理	描述
StartGUI.lua	executable	GUI_Loading.lua	这是初始化游戏的脚本文件，设定常量，调用载入文件
LuaSupport.lua	StartGui.lua		这个文件包括所有与游戏相关（除了界面）的函数
GUI_Loading.lua	StartGUI.lua	GUI_MainMenu.lua	载入画面会显示几秒游戏 logo 并打开主界面

StartGUI.lua 这个脚本我们在前面的章节中提到过。它被主程序调用，用来初始化游戏，并启动程序流程。

StartGUI.lua 脚本的末尾如下所示：

```
-- This initializes the game at startup.
math.randomseed(os.date("%d/%m/%S"))

-- Load in the support functions
dofile("Scripts\\LuaSupport.lua")

--Start in the ingame screen
RunGUI("Loading.lua")
```

首先，我们使用 `os.date()` 函数初始化随机函数生成器，在脚本开始时得到一个新的随机函数序列。然后，载入 `LuaSupport.lua` 文件，包含了所有非界面相关的游戏函数。最后，调用第 8 章学习过的 `LuaGlue` 函数 `RunGUI` 打开第一个游戏画面。

表 9.2 中的另一个文件 `LuaSupport.lua` 用来定义所有在游戏中使用的与界面不相关的函数（与界面相关的函数单独存放在一个文件中）。通常，这个文件会逐渐变成项目中最大的一个脚本文件，当载入该文件时，Lua 代码会被编译并存入内存，在游戏中就可以根据需要调用。随着项目规模逐渐增大，把这个文件分离成许多小文件也是一个很好的方法，如 `AI_functions.lua`、`Save_game.lua` 等。

目前我们知道了项目中的 3 个 lua 脚本，回过头来再看看开发范围文档，我们可以考

虑在游戏开发过程中还应该添加什么脚本，毕竟还有很多需要展示给玩家的信息，比如下面的内容：

- 游戏开始
- 显示游戏子画面
- 显示高分榜
- 控制推进器
- 控制敌方飞船
- 管理存储容器
- 游戏数据的追踪
- 游戏保存

通过这个列表，我们可以创建一个 Lua 脚本的工作表格，用来有效地管理我们的游戏。脚本列表参考表 9.3。图 9.2 展示了《Take Away》游戏中由不同界面和 lua 脚本组成的流程。

表 9.3 《Take Away》中使用的 Lua 脚本

Lua 文件	调用位置	下一步处理	描 述
StartGUI.lua	executable	GUI_Loading.lua	这是初始化游戏的脚本文件，设定常量，调用载入文件
LuaSupport.lua	StartGui.lua		这个文件包括所有与游戏相关（除了界面）的函数
GUI_Loading.lua	StartGUI.lua	GUI_MainMenu.lua	载入画面会显示几秒游戏 logo 然后打开主界面
GUI_MainMenu.lua	GUI_Loading.lua	GUI_InGame.lua	主界面显示高分榜、允许玩家开始游戏或者退出游戏，还可以保存和载入游戏进度
GUI_InGame.lua	GUI_MainMenu.lua	GUI_EndGame.lua/ GUI_Escape.lua	InGame 界面在游戏过程中显示。玩家可以按下 <Esc> 键调出 Escape 界面。游戏结束时显示 End Game 界面，该界面还可以显示玩家的当前分数
GUI_EndGame.lua	GUI_InGame.lua	GUI_InGame.lua	该界面告诉玩家游戏结束，显示最终分数，并让用户返回主界面
GUI_Escape.lua	GUI_InGame.lua	GUI_InGame.lua	该界面暂停游戏，用户可以返回、退出或者保存游戏
GUI_KeySelect.lua	GUI_MainMenu.lua	GUI_MainMenu.lua	该界面用于匹配用户的按键
Text.lua	StartGui.lua		用于保存项目中的文本字符

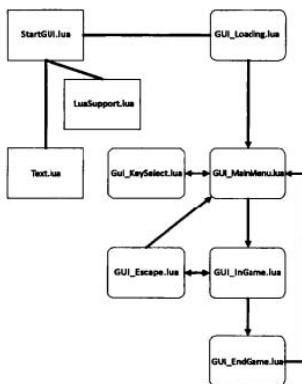


图 9.2 《Take Away》游戏中由不同界面和 Lua 脚本组成的流程

当你开始把自己的想法写到开发范围文档，填入脚本列表时，你就开始走上了游戏设计的漫漫长路。当然，大的项目可能需要更多的 Lua 脚本和更详细的设计文档，不过对于我们的第一个游戏《Take Away》来说，这些设计文档已经足够了。

理想状况下，设计文档应该是贯穿游戏开发过程的蓝图。它应该包含游戏的技术平台，还应该列出游戏的基本流程。

当把所有的设计都汇集到一起时，要考虑以下几方面：首先，文档的阅读对象是谁？是你还是游戏开发团队，无论是谁，使用文档的一定是最终开发游戏的人。这不是一份商业计划书，因此不必浪费时间在优美的软文、无意义的图片和其他那些改变文档意义的东西上，要记住设计文档只是帮助你开发游戏的。

明确了这点后，我们就可以把所有的设计要素编写成设计文档。这份文档是简单明确的，没有任何多余的东西，但它涵盖了游戏的所有方面，并展示给我们一个如何制作游戏的清晰蓝图。

很重要的一点是，设计文档并不包含游戏中详细任务的细节，它仅仅从不同角度描述了游戏的功能。诸如界面设计、战舰的样子、AI 函数或者存档功能等细节工作，就可以交给程序员、设计师和美术师，让他们充分发挥自己的想象力。

## 9.3 设计文档



可以花点时间阅读设计文档（保存于 CD-ROM 的 Chapters/Chapter9/takeawaydesign.doc 文件中），大致了解下一章中我们开发的第一款 Lua 游戏的设计蓝图。下面是设计文档的部分章节。

设计文档——Take Away

上次修改时间：2002/14/05

编辑者：Paul Schuytema

### 1. 游戏概念

在《Take Away》的游戏中，玩家会控制一艘小飞船，对抗一伙偷箱子的敌人。玩家可以飞行、会轰击敌人，并尽可能让自己生存更久。当箱子都被偷走时，游戏结束。

### 2. 环境

《Take Away》看起来会像复古的街机游戏，像素风格的画面，黑色背景。

游戏概念这一节的内容直接引用了开发范围文档中那段简短的描述。环境小节让我们确定游戏的视觉效果。对于复杂的游戏，会有更具体的描述，不过对于《Take Away》这个小游戏来说，我们只是简单描绘了关键的要素：像素画风和大胆的色彩。

### 3. 技术需求

《Take Away》是 PC 平台的游戏，使用 DirectX 9 渲染。GUI 和子画面对象使用 C++ 开发，游戏主流程使用 Lua。

C++：渲染、音效、输入、Lua 实现、LuaGlue 函数、基础 GUI 控件。

Lua：事件处理、GUI 设计、子画面控制、游戏数据、游戏保存和载入、游戏玩法、计分系统、AI。

游戏的运动物体（如飞船）和界面控件使用 2D 图像。所有的图像都是 24 位 BMP 文件。

需要的 GUI 控件对象：子画面、显示标签、文本输入框、按钮、计时器、键盘输入、鼠标输入。

在技术需求小节中，我们列举了所有与游戏相关的技术要点，明确了什么语言处理什

么任务、目标系统、需要响应的输入。这节为开发团队的工作打下了基础，可以帮助项目经理选择合适的人承担对应的工作。

#### 4. 基本玩法

游戏开始时，在屏幕中间有 8 个“箱子”，玩家的飞船出现在屏幕正中间。玩家必须保护这些箱子，因为敌人的飞船会不断出现并试图偷走箱子。当最后一个箱子被拖出屏幕时，游戏结束。玩家可以调整飞船方向并向前推进。游戏没有重力，飞船会一直移动直到碰到屏幕边缘反弹，或者转向后向相反方向推进。

敌人的飞船会从屏幕外的随机位置进入屏幕（有不同类型的敌人），冲向玩家的飞船或者箱子。玩家可以朝敌人开火，击中一次，敌人就会被摧毁。敌人也会向玩家开火。

如果玩家和敌人相撞，那么双方都被摧毁。

当玩家被摧毁后，会在屏幕中间重新生成（不管是被击中或者撞毁）。

随着时间的流逝，敌人的飞船会越来越快，游戏会更有挑战性。

游戏结束时，玩家可以看到他的得分。如果分数足够高，就会被保存到前 10 位的高分榜中。

在基本玩法这小节中，我们介绍了游戏的玩法。读者会发现我们并没有提到界面或者菜单，或者玩家单击“开始”按钮后会发生什么。对于一些游戏来说，这个小节可能有十几页。这是游戏体验的蓝图，也是 Lua 脚本程序员开始实现游戏玩法的起点，以及技术小组理解他们工作内容的重要参考。

#### 5. 玩家操作

玩家的操作可以和 KeySelect 界面的按键对应。当进入正式游戏画面时，按键才能使用。

飞船左转（默认 [ 键）：左转 45°。

飞船右转（默认 ] 键）：右转 45°。

向前推进（默认空格键）：朝飞船目前的方向推进。

开火（默认 <P> 键）：飞船发射一颗子弹。

退出菜单（<Esc> 键）：调用 GUI\_Escape。

玩家操作小节列出了在游戏里玩家可以交互的所有操作。脚本程序员可以根据这些信息选择适当的按键、控制器或者按键事件，并编写相应的处理代码。

## 6. 游戏对象

游戏应该提供初始化函数用来设置所有的游戏参数，如速度、得分、位图名等，这样可以为后续的优化工作提供方便。

所有的飞船对象（玩家的飞船和4种敌舰）都是基于一张64×64像素的位图，每种飞船有8个方向的图像。

玩家飞船（浅蓝）：响应玩家的控制信息。

敌舰A（深蓝）：忽略玩家，直接去偷箱子。当移动到箱子上时会拖走箱子并飞出屏幕，除非被击中（箱子掉落）。最大速度和玩家一样。50分。

敌舰B（浅绿）：向玩家的方向飞行并攻击玩家。最大速度和玩家一样。100分。

敌舰C（红）：向玩家的方向飞行，撞击玩家（不射击）。最大速度比玩家快。200分。

敌舰D（黄）：向玩家或者箱子飞行（更近的那个），可以攻击玩家，也可以偷箱子。最大速度和玩家一样。300分。

箱子：16×16像素单色位图。游戏开始时有8个箱子。当一个偷箱子的敌人被消灭后，箱子停留在掉落处。

子弹：16×16像素单色位图。子弹以匀速朝着发射的方向飞行，飞出屏幕时消除。

在游戏对象这节中，我们列出了游戏中所有的物件，并描述了它们的属性。这个重要的小节在大型角色扮演游戏或者策略游戏中常常会超过100页。这节的目的在于描述对象的完整细节，避免在开发时造成混淆。

## 7. 游戏界面

Loading：显示一段时间的游戏图标，进度条显示进度，启动主界面。

Main：显示高分榜、关闭窗口、设定按键（KeySelect）、玩游戏（InGame）或者载入游戏。载入按键只有在有存档时显示（存档显示上次保存的日期）。只能保存一个存档。

KeySelect：可以让玩家为4个功能键设定任意按键，功能键可还原为默认键，设定好后可以返回Main界面。

InGame：实际游戏画面，玩家可以一直看到他的分数，按<Esc>键出现Escape界面，游戏失败显示EndGame界面。

Escape：暂停游戏画面，可以让玩家保存进度，返回游戏，或者退回Main界面。

EndGame：该画面报告玩家的最终得分，并让玩家知道自己是否进入了高分榜。可以返回Main界面。

游戏界面这一节包含了每个界面的功能描述、可以看到的界面和子界面。有时这节会独立出来成为“功能描述”文档。在这个部分，我们要提供足够的细节，让脚本程序员可以准确地开发各个游戏界面。

要把游戏设计文档看成是开发团队的蓝图，不管成员只有你还是专业的开发者团队。在开发过程中，设计文档可能会有变动，但是它是一个基础，在正式开发游戏之前需要创建一份包括游戏所有方面描述的文档。

一定要记住这是一份“开发文档”，它应该只包含那些重要的东西。它不是给发行商看的“推销文档”，不要写那些冗余的无关紧要的话。这份文档是开发团队用来有效协作，创造非凡游戏体验的工具。

## 9.4 Lua 编程规范

因为 Lua 是非常灵活的语言，所以很容易上手开始编程。虽然入门容易，但是随着脚本复杂度的提高，用户需要有意识地去保持代码的有序性。

Lua 可以被用于任何文本编辑器，用户也可以选择专业的程序编辑器，如 Zeus，如图 9.3 所示（CD-ROM 里有一个 Zeus）。使用专业的编辑器可以更方便地格式化和结构化 Lua 脚本，不过脚本是否易于阅读和理解还是取决于用户。特别是在经过了很长时间后再回过头来阅读代码时，如果最开始代码写得很草率、大意，而且没有添加适当的注释，就很可能浪费很多时间去弄明白某个函数做了什么。

改善这种情况的一种补救方式是养成写清晰注释的好习惯。给每个函数都添加头注释以解释函数的功能，每次添加变量、返回值或者复杂处理时都添加注释。现在添加注释可以以为以后调试和修改 bug 节省很多时间。

还有一种更有效的方法是保持代码风格的一致性。如果能在每个脚本、每个函数、每个项目上保持风格的一致性，就可以自信且快速地阅读和审阅之前的代码。在团队开发环境中，遵循特定的标准风格来编写代码也是非常重要的，这样可以至少在不同实现方式上最低限度地保持命名和程序结构的一致性。

我们提供了一个 Lua 编程规范（HTML 格式），保存在 CD-ROM 的 Chapters/Chapter9/Lua\_style.htm 文件中。读者可以用这份文档作为 Lua 代码的规范或者在它的基础上进一步修改，创建自己的规范。接下来我们花些时间来说明一下这种编程规范的要点。







图 9.3 正在编辑 Lua 脚本的 Zeuss 程序编辑器

在之前的章节中，我们讨论过了变量的命名规范，现在来看看别的一些规范。首先，在每个脚本的开始要添加头注释，提供一些这个文件的基本信息，如下：

```
=====
-- (c) copyright 2005, Lantern Learning, LLC
-- All Rights Reserved. U.S.A.
=====
-- filename:  GUI_Escape.lua
-- author:    Chris Listello
-- created:   April 4, 2004
-- descrip:   Escape key InGame menu
=====
```

这些信息可以作为文件的功能介绍，以便之后查阅，也可以帮助其他的开发者阅读代码。此外，我们还应该给每一个函数添加标准的注释，参考下面的示例：

```
=====
-- function:  CreateMultiText
-- author:    Paul Schuytema
-- created:   April 1, 2004
-- descrip:   Creates a multiline text object
-- returns:   nothing (process)
=====
function CreateMultiText(localID, x, y, lines, charWIDTH)
```

有时，用户会很惊讶自己忘记代码的速度，不管当初起了多好的名字，这段注释可以帮用户找回相关的记忆。注释也是一个很好的工具，可以在用户快速查看很长的脚本时，

很自然地只看这些一段一段的注释，从而提高效率。

在第 10 章中，我们会通过游戏项目中界面的相关代码进一步了解编程规范。目前只需要快速查看 CD-ROM 中附带的相关规范，尽快熟悉这些将用在脚本中的标准，并开始思考要如何定制更适用于自己项目的脚本规范。

CD-ROM 中包含了本章的文件夹，里面保存了在开发过程中提到的所有脚本。到此，我们已经为《Take Away》游戏的开发打下了一个坚实的基础。



## 9.5 本章小结

在本章中，思考游戏项目前，我们先讨论了一些关键的设计假设，然后列出了开发《Take Away》游戏需要的 Lua 脚本。在这个过程中，我们起草并完成了游戏设计文档，它将成为脚本开发的基本蓝图。

在下一章中，我们要讨论如何使用 Lua 管理和保存实时的游戏数据。



## 第 10 章

# 使用 Lua 处理游戏数据

### 本章要点

- 简单的游戏数据
- 大数据集
- 使用 Lua 保存游戏数据

Lua 在游戏开发中的一个很强大的特性是它能够作为保存、载入游戏数据的主要工具。所有的游戏都和数据相关，在脚本层处理数据，意味着游戏设计师和脚本程序员可以独立于底层的核心代码，访问所有的游戏数据并且设计符合游戏需要的数据存储方式。Lua 还可以被用于保存游戏进度、游戏开始时的数据载入，以及玩家载入游戏进度。

在本章中，我们将要了解如何使用 Lua 变量和表来存放运行时数据，以及如何输出数据保存游戏进度。

### 10.1 简单的游戏数据

几乎所有的游戏都有很重要的数据组件。一些简单的经典游戏，如《Space Invader》（太空侵略者<sup>①</sup>）、《Asteroids》（爆破彗星<sup>②</sup>），在运行时只有很少的数据，但也包含了基本的信息。更多的现代游戏，如《Doom3》（毁灭战士 3）或者《Neverwinter Nights》（无冬之夜），在运行时会有大量的数据产生。

有些数据在 C++ 领域是十分清楚的。想象一下 3D 第一人称射击游戏，在每一帧的图形绘制中，计算决定了屏幕上显示的場景以及用来表示动态物体的数据，如那些玩家射击的怪物和玩家自己控制的 3D 角色，这些物体都是由成百上千个点和三角形组成，这些就

① 《Space Invader》是 1978 年由日本太东公司发行的街机游戏，由西角友宏设计。

② 《Asteroids》是 Atari 公司在 1979 年发行的一款街机游戏。它是街机黄金年代里一个最受欢迎且极具影响力的游戏。——译者注

是游戏需要处理、转换和显示的数据。

这些游戏场景、3D 物体、在显存上的位图渲染等数据最好使用渲染相关的代码，交给游戏程序员用主开发语言来管理。但对于我们之前提到的 3D 第一人称射击游戏来说，一些实时的数据是非常适合使用 Lua 环境的。游戏角色包含一些基本属性，如速度、剩余的子弹数、正在使用的枪械、装备的子弹类型。所有的这些都是关系到游戏体验的重要数据，比绘图都更贴近用户的体验，是更高层的。这些就是需要转移到 Lua 环境中，使用脚本管理的数据。

### 10.1.1 太空飞船的例子

让我们回到之前《Asteroids》游戏的例子。希望读者还记得这款经典游戏，玩家控制一个小飞船可以在屏幕四周飞行。玩家的目标是发射炮弹摧毁周围飞过的小行星，可参考图 10.1。



图 10.1 类似经典游戏 Asteroids 的屏幕截图

我们需要跟踪一些关于飞船的基本运行时数据：位置、方向和移动。位置信息很明显，在 2D 游戏中，x-y 坐标决定了更新屏幕时需要绘制飞船的位置。方向是表示飞船朝向的简单数值（单位为度数）。在《Take Away》游戏中，我们限定了飞船的 8 个角度，每个相差 45°。因此，在我们的游戏中，方向是 1~8 的数字。

移动的数据有些特殊，需要特别处理（特别是《Take Away》游戏中飞船的移动机制类似于《Asteroids》）。在《Asteroids》游戏里，玩家按下推进键，会给飞船一个推动力。

因为游戏没有重力和摩擦力的设定，所以一旦在一个方向施加了推动力，飞船就会一直朝那个方向飞行（直到撞着小行星）。当飞船飞出屏幕边缘后会从另一边出现。

在游戏中，每当飞船在屏幕中被刷新时，不管是否施加了推动力还是调整了方向，都会在某个方向进行移动。我们可以把这个移动看成是类似位置坐标的 x-y 值，区别在于它只是绘制飞船时的位移值。



在绘制飞船图像时，还需要一个数据：绘制飞船后部的推进器的火焰。因此，在使用 Lua 设定飞船的数据时，代码可能看起来像下面例子这样（参考 CD-ROM 中的 ch10\_1.lua 代码文件）：

```
myShip = {}
myShip.Xposition = 100
myShip.Yposition = 100
myShip.Rotation = 0
myShip.Xthrust = 0
myShip.Ythrust = 0
myShip.ShowThrust = false
```

开始先创建 myShip 表，因为要用它存放多个数值，然后设定并初始化属性关键字（在我们的例子中，游戏场景是 200 × 200 像素，飞船在屏幕中间）。我们使用 Lua 环境中这个简单并且强大的数据结构作为保存飞船属性的容器，然后再使用下面的函数处理位移数据并绘制飞船：

```
function: UpdateShip(theShip)
    theShip.Xposition = theShip.Xposition + myShip.Xthrust
    theShip.Yposition = theShip.Yposition + myShip.Ythrust
    DrawShip(theShip.Xposition, theShip.Yposition)
    if myShip.ShowThrust then
        DrawThrustImage()
    end
end
```

在这个例子中，使用推进器的数据来更新位置坐标，然后在新的位置绘制飞船。接着检测推进器是否正在使用，如果是就绘制图形（从飞船后面喷出的火焰）。

利用这个非常简单的方法，我们建立了一个保存和管理飞船数据的机制。此外还应该有事，即在推进键按下时，根据飞船的方向来更改 Xthrust 和 Ythrust 的值。

思考一下这个例子，如果要跟踪多个飞船，比如你有一个舰队或者多个敌人的情况，该如何用 Lua 来处理？

看下面的示例：

```
myShip = {}  
yValue = 60  
for index = 1,5 do  
    myShip[index] = {}  
    myShip[index].Xposition = 100  
    myShip[index].Yposition = yValue  
    yValue = yValue + 20  
    myShip[index].Rotation = 0  
    myShip[index].Xthrust = 0  
    myShip[index].Ythrust = 0  
    myShip[index].ShowThrust = false  
end
```

在这个例子中，我们利用 Lua 多维表的特性初始化了 5 艘飞船。在 for 循环中，为每艘飞船设定了偏移值（yValue），让它们排列整齐。还可以改变传入的数值，利用前面的函数来更新飞船的绘图：

```
UpdateShip(myShip[2])
```

在这个调用中，我们传入了 5 艘飞船中的第 2 艘的属性 table，在函数中引用的属性关键字仍然可以正常工作。

如果使用这种数据结构来维护游戏中相同敌人的数据的话，那么就可以利用 Lua 标准的表操作函数来管理对象的数量。例如，在我们 5 艘飞船中，如果第 3 艘被摧毁，就可以使用下面的代码来移除它：

```
table.remove(myShip, 3)
```

这个调用会删除第 3 艘飞船的整个子表，并重新索引 myShip 主表。

如果要在表中添加飞船，可以使用 table.insert() 函数，但必须保证插入的是正确的类型，因为 myShip 表中的每个元素都是一个表。要做到这点，我们需要插入表类型数据，参考下面的示例：

```
table.insert(myShip, {})
```

这段代码会在 myShip 表的末尾插入空表，然后再为这张表设定适当的初始值。参考下列代码：

```
lastEntry = table.getn(myShip)  
myShip[lastEntry].Xposition = 100  
myShip[lastEntry].Yposition = 100  
myShip[lastEntry].Rotation = 0  
myShip[lastEntry].Xthrust = 100  
myShip[lastEntry].Ythrust = 0  
myShip[lastEntry].ShowThrust = false
```

同样，你也可以像下面这样：

```
table.insert(myShip, {Xposition = 100, Yposition = 100})
```

当然，你可能想填满整张表，这种方式尽管不易阅读，但是可以在一个函数调用里完成任务。

### 10.1.2 《Take Away》的玩家飞船

我们已经介绍了游戏《Asteriods》的实时数据，现在来看看《Take Away》这个游戏。在这个游戏中，我们操作一个飞船，飞行方式和《Asteriods》类似。游戏开始的时候，需要初始化玩家飞船的核心数据，代码在 LuaSupport.lua 文件的 initialSetup() 函数中。部分代码可参看代码清单 10.1。

代码清单 10.1 InitialSetup() 函数

---

```
function InitialSetup()
    --Sets up key data for the game
    --Initial player values
    myRotation = 1 --Player's rotation (#)
    myX = 390      --Player's x coordinate (#)
    myY = 290      --Player's y coordinate (#)
    myXThrust = 0  --Player's thrust along the x-axis (#)
    myYThrust = 0  --Player's thrust along the y-axis (#)
    alive = "yes"  --Player's life status ("yes" or "no")
    --Initial limits
    respawnInterval = 20 --Number respawnCounter must reach to respawn
    player (#)
    --Initial setting of counters
    respawnCounter = 0 --Player's death period (#)

    score = 0          --Player's score (#)
    timeCounter = 0     --Passage of time (#)
    targetDoneCounter = 0 --Targets stolen by enemies (#)
    --Preferred game speed
    refreshRate = .1 --Seconds between timer expirations (#)
    --Initial GUI setup goes here
    --
    --
end
```

---

在这个函数中，我们使用基本变量保存玩家的飞船数据（相对于表）。在函数开始部分给这些变量设定了默认的初始值。另外，还设定了其他游戏需要的主数据变量。

在 Lua 脚本中，创建这些在游戏开始时调用的初始化函数是一个很好的做法。把所有

初始化核心数据的处理都放在一个函数中，可以让用户很容易地在需要的时候调用函数，重置游戏初始状态。另外，这种“一站式”服务般的函数还能让用户方便地调整数据，测试不同的参数下游戏的表现。

函数的最后部分还没实现，在下一章中等我们熟悉了不同的 GUI 控件，并且能够在屏幕上绘制图形后，再做这部分工作。

### 10.1.3 敌舰数据

我们已经完成了飞船的核心数据的设定，现在需要为敌舰以及那些需要保护的补给箱设定数据。因为几种敌舰的属性只是稍有区别，所以可以像上个例子那样把它们定义在 table 中。

先来看看初始化敌舰的函数，参考下面的示例：

```
function EnemyInit()

    enemyCount = 5 --Number of enemies in the game
    myEnemies = {} --Creates myEnemies table
    --Creates table, one entry per potential enemy
    for indx = 1,enemyCount do
        --creates a table to hold the data for each enemy
        myEnemies[indx] = {}
        --now initialize the enemy
        EnemyRespawn(indx)
    end
end
```

这个函数设定了游戏中会出现多少种敌舰，并初始化了存储数据的表，没有设定值，只是调用了 EnemyRespawn() 函数填充数据。

EnemyRespawn() 函数负责设定敌舰数据，我们把它独立出来是为了重新生成所有敌舰的时候不用再初始化 table。也可以用这个函数一次只重新生成单个敌舰（如替换被击毁的）。函数的代码可参考代码清单 10.2。

代码清单 10.2 EnemyRespawn() 函数

```
function EnemyRespawn(indx)
    --Fills/refills myEnemies table according to indx
    --indx is the myEnemies table index assigned to the enemy
    --Initial values
    myEnemies[indx].XTHRUST = 0
    myEnemies[indx].YTHRUST = 0
    myEnemies[indx].ROT = math.random(1,8)
    myEnemies[indx].ID = 14+indx --Starts IDs at 15
```



```
myEnemies[indx].E_TOW = 'no'
myEnemies[indx].FIRE = 0
--Randomly selects side of screen to enter from
entrySide = math.random(1,4)
if entrySide == 1 then --Left
    myEnemies[indx].EX = math.random(-40,-20)
    myEnemies[indx].EY = math.random(-40,620)
elseif entrySide == 2 then --Right
    myEnemies[indx].EX = math.random(800,820)
    myEnemies[indx].EY = math.random(-40,620)
elseif entrySide == 3 then --Top
    myEnemies[indx].EX = math.random(-40,820)
    myEnemies[indx].EY = math.random(-40,-20)
else --Bottom
    myEnemies[indx].EX = math.random(-40,820)
    myEnemies[indx].EY = math.random(600,620)
end
--Determines enemy's thrust, reaction, and firing abilities based on
time
--Reaction time decreases as REACT decreases (must be at least 1)
--Maximum thrust increases as MAX increases
--enemyFireInterval (#) compared to FIRE to determine enemy shooting
(see EnemyFacing(indx) function)
if (timeCounter >= 0) and (timeCounter < 100) then
    myEnemies[indx].REACT = 0
    myEnemies[indx].MAX = 5
    enemyFireInterval = 9
elseif (timeCounter >= 100) and (timeCounter < 200) then
    myEnemies[indx].REACT = 4
    myEnemies[indx].MAX = 6
    enemyFireInterval = 8
elseif (timeCounter >= 200) and (timeCounter < 300) then
    myEnemies[indx].REACT = 3
    myEnemies[indx].MAX = 7
    enemyFireInterval = 7
elseif (timeCounter >= 300) and (timeCounter < 400) then
    myEnemies[indx].REACT = 2
    myEnemies[indx].MAX = 8
    enemyFireInterval = 8
elseif (timeCounter >= 400) then
    myEnemies[indx].REACT = 1
    myEnemies[indx].MAX = 9
    enemyFireInterval = 5
end
--Randomly selects the AI type
myEnemies[indx].TYPE = math.random(1,4)
--create the sprite and place here
--
--
end
```

函数开始先设定了敌舰的基本数值，并且选择了随机的方向。ID 键用于游戏的 GUI 系统（下章会详细解释），这个唯一的 ID 让我们可以定位并且操作用来表示某个敌舰的 sprite。

下一步是决定敌舰从屏幕的什么地方出现。游戏场景是 600 × 800 像素的屏幕区域，我们在屏幕外生成敌舰，让它们按照自己的方式进入屏幕。随机选择屏幕的一边，并设定一个 x 和 y 坐标的随机范围，让它们在下一帧更新时显示在屏幕上（如果在屏幕外就不显示）。

接下来需要为敌舰设定一些行为参数，第 12 章会进一步说明。简单来说，就是随着游戏进程难度会逐渐增加。函数设定了不同进程下敌舰的属性。

最后，就像游戏设计文档提到的，游戏中有 4 种不同的敌舰，我们随机选择一种。完成了这些处理后，就要初始化 GUI sprite 来显示敌舰，在第 11 章中会详细说明。

#### 10.1.4 补给箱数据

最后需要设定的数据就是那些需要我们保护的箱子。在游戏开始的时候要创建 8 个箱子，环形排列在屏幕中间（玩家出现在正中间）。我们使用下面的初始化函数，可参考代码清单 10.3。

代码清单 10.3 创建补给箱的函数

```
function TargetInit()
--Creates and fills myTargets table
    myTargets = {}
    targetCount = 8 --Number of targets in the game
    startID = 2000
    startX = 360
    startY = 260
    --Creates/fills table with initial values
    for indx = 1, targetCount + 1 do
        myTargets[indx] = {}
        myTargets[indx].T_ID = startID
        startID = startID + 1
        myTargets[indx].T_TOW = 'no'
        myTargets[indx].T_X = startX
        startX = startX + 30
        if startX > 430 then
            startX = 360
        end
        myTargets[indx].T_Y = startY
        if indx > 3 then
            if indx < 7 then
                startY = 290
            end
        end
    end
end
```

```
        else
            startY = 320
        end
    end
end

--now delete the middle target, so the player can spawn in
table.remove(myTargets, 5)

--Creates targets (as sprites) and places them
--
--
end
```

函数开始部分设定了箱子的基本数据，然后使用了一对 if 语句设定 x 和 y 坐标值，一共 9 种组合。我们用了 9 宫格的排列，这是最简单的一种方式。创建好后再删除第 5 个箱子（中间的那个），得到一个环形的排列，这样就可以在屏幕中间生成玩家的飞船，如图 10.2 所示。

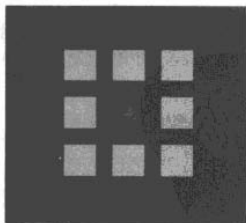


图 10.2 《Take Away》游戏中补给箱的初始化

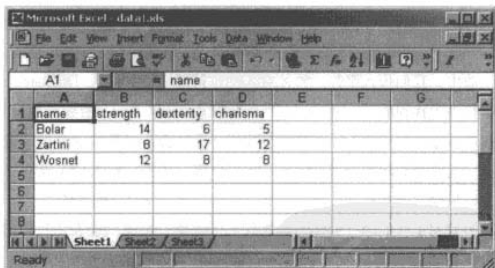
## 10.2 大数据集

我们已经看到了怎样利用 Lua 的简单数据结构来保存游戏的运行时数据。对于像《Take Away》这样简单的游戏来说，这个是最基本的方法。但对于大型游戏该如何处理呢？通常一些游戏大作或者角色扮演游戏都需要载入大量的数据，对于这种情况，Lua 也可以成为非常出色的开发工具。

## 10.2.1 表单型数据

一般大型的数据集需要开发者利用像 Microsoft Excel 这样的外部工具来保存数据。想象一下追踪 1000 个 NPC（非玩家控制角色）的名字和状态，在这种情况下，表单就是非常实用的工具了。采用这种方法时，程序员需要开发工具来解析 Excel 文件并将数据载入游戏。

我们可以使用 Lua 来完成这个工作。如果把表单保存为 CSV 文件（数值用逗号隔开的文件），就可以很容易地使用 Lua 解析并保存到表格中。思考图 10.3 这个表单的示例。



The screenshot shows a Microsoft Excel window titled 'data.xls'. The active sheet is 'Sheet1'. The table has 7 columns (A-G) and 8 rows (1-8). The data is as follows:

	A	B	C	D	E	F	G
1	name	strength	dexterity	charisma			
2	Bolar	14	6	5			
3	Zartini	8	17	12			
4	Wosnet	12	8	8			
5							
6							
7							
8							

图 10.3 Excel 格式的简单的角色数据表单

在这个例子中，第一行是键名，剩下的行是数值。我们可以创建下面的 Lua 函数载入该 CSV 文件并保存到表中（参考 ch10\_2.lua 代码文件）：



```
function GetLines(fileName)
    indx = 0
    myLines = {}
    for line in io.lines(string.format("%s%s", "c:/lua_scripts/",
    fileName)) do
        indx = indx + 1
        myLines[indx] = line
    end
    return indx, myLines --returns number of lines and line table
end
```

参数是文件名（例子里是固定的路径，用户可以任意变更）。函数返回文件的行数和一个表，每条数据包含了该行的完整字符。

然后再创建一个函数解析每行的字符，并创建表格保存该行的所有数据，参考下面的示例：

```
function GetValues(myString)
    num = 0
    values = {}
    if myString ~= nil then
        while string.find(myString, ",") ~= nil do
            i, j = string.find(myString, ",")
            num = num + 1
            values[num] = string.sub(myString, 1, j-1)
            myString = string.sub(myString, j+1, string.len(myString))
        end
        num = num + 1
        values[num] = myString
    end
    return num, values
end
```

在这个函数中，我们传入字符（刚才载入的每行的完整字符）。调用 `string.find()` 函数定位在这个字符串中找到的下一个逗号，然后再调用 `string.sub()` 函数截取逗号前的数值。之后移除该数值和逗号继续解析，直到没有数值。函数会返回获取的数值个数以及包含了所有数值的表。这两个简单的处理步骤可以让用户解析几乎所有的 CSV 文件，不论简单还是复杂的文件，只要它遵守标准，第一行是属性名，之后是对应的数值。

下面的函数调用了我们刚才创建的函数载入了表单，生成二维表并打印出结果。用户可以在控制台运行该示例，载入 `ch10_2.lua` 代码文件（通过 `dofile()` 方法），运行 `LoadCharacters()` 函数。

```
function LoadCharacters()
    myCharacters = {}
    numLines, allLines = GetLines("data1.csv")
    --load labels (the first line)
    count, myLabels = GetValues(allLines[1])
    --ignore line 1, it's got the labels
    for indx = 2, numLines do
        count, charHold = GetValues(allLines[indx])
        myCharacters[indx-1] = {}
        for indx2 = 1, count do
            myCharacters[indx-1][indx2] = charHold[indx2]
        end
    end
    --now print them
    for indx = 1, 3 do
        for indx2 = 1, table.getn(myLabels) do
            print(myLabels[indx2], myCharacters[indx][indx2])
        end
    end
end
```

该函数把属性名读入一个表，数值读入另一个二维表，最后打印出每个角色和它们对应的属性。可以看出，通过 Lua 和电子表单工具这种方式，保存、载入和管理游戏数据是多么简单。

针对易读性和灵活性这两个需求，我们有两种处理方式。第一种方法就像前面例子中那样，属性名和数值分别用两个表存储，使用相同的索引。这种方式可以让用户在 Lua 中根据表单宽度创建数据存储表，然后使用数字索引来获取对应数值（如用 `myCharacters [charNum] [fieldNum]` 取得角色的力量值，这里 `fieldNum` 代表力量值的索引）。这种方法有最好的灵活性，不过代码比较难以理解。

另一种方法是在 Lua 中定义固定的属性名，在数据结构发生变化时会带来一些麻烦，不过可以让代码易读并容易调试。我们可以用类似 `myCharacters [charNum] . strength` 的方式获取角色的力量值。

学习了如何利用电子表单这样的外部工具保存海量数据，我们就可以在游戏中载入并使用这些数据。这种方式非常适合有大量数据需要管理，并且数据之间还有关联的情况，比如我们想查看所有的数据并做一些整体的处理时，可以利用 Excel 的功能对数据进行排序、批量处理和公式运算。如果要处理 800 个以上的 NPC 角色，我们可能会采用这种方法，因为可以很容易地排序和调整数值。

## 10.2.2 Lua 格式的数据文件

有时数据之间并不需要互相关联，单独查看反而更好（例如，在“大亨”游戏中需要为物体设定的交易参数）。这种情况下，直接在 Lua 脚本中设定数值比利用外部工具更加方便。使用 Lua 初始化数据，可以直接得到初始结果，而不用进行任何解析和辅助的文件处理。参考下面的示例：

```
pokerPlayers = {}
index = 1
pokerPlayers[index] = {}
pokerPlayers[index].Name = "Ralph Hollywood"
pokerPlayers[index].Gender = "Male"
pokerPlayers[index].Model = "poker_player_02.mlm"
pokerPlayers[index].Skin = "poker_player_02_blue.bmp"
pokerPlayers[index].Anim = "sit_idle_male"
pokerPlayers[index].CurCash = 550
pokerPlayers[index].Aggressive = 1
```

在这个例子中，我们创建了一个 `pokerPlayers` 的表，用来保存游戏 Texas Hold'Em 锦标

赛中角色相关的数据，然后设定索引值为 1，之后使用 `pokerPlayers[index] = {}` 语句创建表格。

该语句创建了一个子表，然后就可以利用易读的属性名为玩家设定数值。从这些数据中可以看到，使用电子表单没有什么必要，因为它们之间是没有关系的，这些属性只是些字符串，用来表示角色名、模型名、待机动画等。在 Lua 中创建并处理这些数据是最简单的方法，在需要载入数据的时候只需要运行这个文件即可。

通过这种方式载入数据，我们就可以在运行时直接改变数值。下面的语句从现金中移除当前的赌注：

```
pokerPlayers[curPlayer].CurCash = pokerPlayers[curPlayer].CurCash -  
curBet
```

如果想重置数据（比如在新游戏开始的时候），我们只需要重新载入这个 Lua 脚本。对于大数据集的处理，可以使用 `ipairs` 遍历函数，参考下面的示例：

```
function PrintAll(t)  
  for i,v in ipairs(t) do  
    print("Player #:", i)  
    print(v.Name)  
  
    print(v.Gender)  
    print(v.Model)  
    print(v.Skin)  
    print(v.Anim)  
  end  
end
```

如果传入 `pokerPlayers` 二维表到这个函数，它会遍历主表的所有元素并返回指定索引的 `index (i)` 和 `value (v)`。这时，得到的数值本身也是一个表，可以通过属性名获取对应的数值。

### 10.3 使用 Lua 保存游戏数据

学习了使用 Lua 保存和修改游戏数据之后，接下来我们看看如何保存和载入游戏进度。实际上，载入游戏进度是非常简单的，只需要把之前数据保存为 Lua 文件。

一般游戏开发者会把数据保存为内部格式然后再直接载入。如果采用 C++ 那样的底层数据访问方式，可以很好地完成任务，但是，我们通常希望保存的数据是可读的（至少在

开发阶段)。这时候,程序员需要开发解析器来处理文件的读取,我们不得不把他从其他的任务调过来处理保存和载入游戏数据的功能。

别忘了,我们是 Lua 脚本开发者,是可以利用 Lua 内建的 I/O 功能来处理数据输出的。我们只需确保输出的数据是采用了合适的语法,这样就可以使用 `dofile()` 载入数据了。

下面的例子是第 5 章 I/O 示例的简单修改。

```
myFile = io.open("c:\\lua_scripts\\test_savegame.lua", "w")
if myFile -- nil then
    myFile:write("--- Test Lua SaveGame file")
    myFile:write(string.char(10))
    myFile:write(string.char(10))
    myFile:write(string.format("%s%s", "-- File created on: ",
    os.date()))
    myFile:write(string.char(10))
    myFile:write(string.char(10))
    myFile:write("myValue = 10")
    io.close(myFile)
end
```

如果运行这段脚本,会生成下面这样的 Lua 文件:

```
-- Test Lua SaveGame file

-- File created on: 02/23/05 15:06:27

myValue = 10
```

这个文件很简单,合法的 Lua 文件就可以使用 `dofile()` 函数载入。我们需要做的是让这个文件更有用,能够在运行时捕获游戏的变量值。我们回过头来看看在《Take Away》游戏中与飞船相关的一些变量。变量的初始化如下所示:

```
--Initial player values
myRotation = 1
myX = 390      --Player's x-coordinate (#)
myY = 290      --Player's y-coordinate (#)
myXThrust = 0  --Player's thrust along the x-axis (#)
myYThrust = 0  --Player's thrust along the y-axis (#)
alive = "yes"  --Player's life status ('yes' or 'no')
```

假设这些变量就是这个游戏的环境设定,我们可以写一个简单的函数在任何时候都可以把它们保存为 Lua 文件。函数的代码可参考代码清单 10.4。



## 代码清单 10.4 将变量值存入 Lua 文件的例子

```
function WorldSave()
    myFile = io.open('c:\\lua_scripts\\sample_savegame.lua', "w")
    if myFile == nil then
        myFile:write('-- Lua SaveGame file for limited Take Away data')
        myFile:write(string.char(10))
        myFile:write(string.char(10))
        myFile:write(string.format("%s%s", "-- File created on: ",
            os.date()))
        myFile:write(string.char(10))
        myFile:write(string.char(10))
        myFile:write("-- ship data values")
        myFile:write(string.char(10))
        myString = string.format("%s%d", "myRotation = ", myRotation)
        myFile:write(myString)
        myFile:write(string.char(10))
        myString = string.format("%s%d", "myX = ", myX)
        myFile:write(myString)
        myFile:write(string.char(10))
        myString = string.format("%s%d", "myY = ", myY)
        myFile:write(myString)
        myFile:write(string.char(10))
        myString = string.format("%s%d", "myXThrust = ", myXThrust)
        myFile:write(myString)
        myFile:write(string.char(10))
        myString = string.format("%s%d", "myYThrust = ", myYThrust)
        myFile:write(myString)
        myFile:write(string.char(10))
        myString = string.format("%s%s", "alive = \"", alive, "\"")
        myFile:write(myString)
        myFile:write(string.char(10))
        io.close(myFile)
    end
end
```

函数很简洁，只是将所有需要保存的变量值存入了文件。这个方法的关键点是创建合法的 Lua 语句的字符串，可以在运行时取得变量值。我们使用 `string.format()` 函数来做这件事。注意函数末尾部分，这里是处理字符型变量的方法。为了生成合法的 Lua 文件，在字符串中必须使用转义字符（\）输出引号。

函数的输出如下所示：

```
-- Lua SaveGame file for limited Take Away data

-- File created on: 02/24/05 16:30:58

-- ship data values
```

```
myRotation = 1
myX = 390
myY = 290
myXThrust = 0
myYThrust = 0
alive = "yes"
```

使用 `dofile()` 函数执行该脚本后，会根据脚本初始化变量，结果和执行初始化函数一样。通过这种方式，可以很容易地读取和编辑游戏存档，以便测试游戏中的不同情况。

现在我们用这个方法开发游戏保存函数，以保存《Take Away》游戏中的所有数据。因为函数很长（参考代码清单 10.5 ~ 代码清单 10.9），所以我们会在一个小章节中专门说明。

在函数的第一部分，初始化文件并写入 session 信息（是可读的信息并包含时间戳）。然后，输出所有定义了游戏当前状态的非表结构数据。我们用临时变量 `myValue` 保存数值，并使用 `string.format()` 函数和 `write` 命令输出结果。

代码清单 10.5 游戏保存函数（第 1 部分）

```
function SaveGame()
--Saves the current game
    gSavedGameDate = os.date("%m/%d/%Y %I:%M%p")
    --Creates the pathway for the file
    local fileName = "Take_Away_Saved_Game"
    --Writes the current game to the specified location
    myFile = io.open(string.format("%s%s", "SaveGames\\", fileName,
        ".lua"), "w")
    if myFile == nil then --File exists
        myFile:write("-- Take Away save game file");
        myFile:write(string.char(10))
        myFile:write(string.char(10))
        myFile:write(string.format("%s", "-- File created on: ",
            os.date()));
        myFile:write(string.char(10))
        myFile:write(string.char(10))
        myFile:write("--Initial player constants")
        myFile:write(string.char(10))
        myValue = myRotation
        myFile:write(string.format("%s%d", "myRotation = ", myValue))
        myFile:write(string.char(10))
        myValue = myX
        myFile:write(string.format("%s%d", "myX = ", myValue))
        myFile:write(string.char(10))
        myValue = myY
        myFile:write(string.format("%s%d", "myY = ", myValue))
        myFile:write(string.char(10))
```

```
myValue = myXThrust
myFile:write(string.format('%s%d', "myXThrust = ",myValue))
myFile:write(string.char (10))
myValue = myYThrust
myFile:write(string.format('%s%d', "myYThrust = ",myValue))
myFile:write(string.char (10))
myValue = alive
myFile:write(string.format('%s%s%s', "alive = ", string.char
(34), myValue, string.char (34)))
myFile:write(string.char (10))
myFile:write(string.char (10))
myFile:write(string.char (10))
myFile:write("---Initial limits")
myFile:write(string.char (10))
myValue = respawnInterval
myFile:write(string.format('%s%d', "respawnInterval = ",myValue))
myFile:write(string.char (10))
myFile:write(string.char (10))
myFile:write("---Initial setting of counters")
myFile:write(string.char (10))
myValue = respawnCounter
myFile:write(string.format('%s%d', "respawnCounter = ",myValue))
myFile:write(string.char (10))
myValue = score
myFile:write(string.format('%s%d', "score = ",myValue))
myFile:write(string.char (10))
myValue = timeCounter
myFile:write(string.format('%s%d', "timeCounter = ",myValue))
myFile:write(string.char (10))
myValue = targetDoneCounter
myFile:write(string.format('%s%d', "targetDoneCounter =
",myValue))
myFile:write(string.char (10))
myFile:write(string.char (10))
myFile:write("---Preferred game speed")
myFile:write(string.char (10))
myValue = refreshRate
myFile:write(string.format('%s%.2f', "refreshRate = ",myValue))
myFile:write(string.char (10))
myFile:write(string.char (10))
```

函数的下一部分输出 myEnemies 表，创建 Lua 脚本并输出表的所有 key 和 value。因为是显式输出数值，所以这个部分的输出文件很大，不过文件很易读。

#### 代码清单 10.6 游戏保存函数 (第 2 部分)

```
myFile:write("---Enemy information")
myFile:write(string.char (10))
myValue = enemyCount
myFile:write(string.format('%s%d', "enemyCount = ",myValue))
myFile:write(string.char (10))
```

```
myValue = enemyFireInterval
myFile:write(string.format("%s%d", "enemyFireInterval = ", myValue))
myFile:write(string.char (10))
for indx = 1, enemyCount do
    myValue = myEnemies[indx].ID
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myEnemies[",
            indx, ".ID = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myEnemies[", indx,
            ".ID = nil"))
    end
    myFile:write(string.char (10))
    myValue = myEnemies[indx].XTHRAUST
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myEnemies[",
            indx, ".XTHRAUST = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myEnemies[", indx,
            ".XTHRAUST = nil"))
    end
    myFile:write(string.char (10))
    myValue = myEnemies[indx].YTHRAUST
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myEnemies[",
            indx, ".YTHRAUST = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myEnemies[", indx,
            ".YTHRAUST = nil"))
    end
    myFile:write(string.char (10))
    myValue = myEnemies[indx].EX
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myEnemies[",
            indx, ".EX = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myEnemies[", indx,
            ".EX = nil"))
    end
    myFile:write(string.char (10))
    myValue = myEnemies[indx].EY
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myEnemies[",
            indx, ".EY = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myEnemies[", indx,
            ".EY = nil"))
    end
    myFile:write(string.char (10))
```

```
myValue = myEnemies[indx].ROT
if type(myValue) == "number" then
    myFile:write(string.format("%s%d%s%d", "myEnemies[",
        indx, ".ROT = ", myValue))
elseif type(myValue) == "nil" then
    myFile:write(string.format("%s%d%s", "myEnemies[", indx,
        "].ROT = nil"))
end
myFile:write(string.char (10))
myValue = myEnemies[indx].FIRE
if type(myValue) == "number" then
    myFile:write(string.format("%s%d%s%d", "myEnemies[",
        indx, ".FIRE = ", myValue))
elseif type(myValue) == "nil" then
    myFile:write(string.format("%s%d%s", "myEnemies[", indx,
        "].FIRE = nil"))
end
myFile:write(string.char (10))
myValue = myEnemies[indx].E_TOW
if type(myValue) == "number" then
    myFile:write(string.format("%s%d%s%d", "myEnemies[",
        indx, ".E_TOW = ", myValue))
elseif type(myValue) == "string" then
    myFile:write(string.format("%s%d%s%s%s",
        "myEnemies[",
        indx, ".E_TOW = ", string.char (34), myValue,
        string.char (34)))
end
myFile:write(string.char (10))
myValue = myEnemies[indx].MAX
myFile:write(string.format("%s%d%s%d", "myEnemies[", indx,
    "].MAX = ", myValue))
myFile:write(string.char (10))
myValue = myEnemies[indx].REACT
myFile:write(string.format("%s%d%s%d", "myEnemies[", indx,
    "].REACT = ", myValue))
myFile:write(string.char (10))
myValue = myEnemies[indx].TYPE
myFile:write(string.format("%s%d%s%d", "myEnemies[", indx,
    "].TYPE = ", myValue))
myFile:write(string.char (10))
end
myFile:write(string.char (10))
myFile:write(string.char (10))
```

接下来的部分采用和输出 myEnemies 表相同的方式，输出 myTargets 表中所有相关的数据。

## 代码清单 10.7 游戏保存函数 (第 3 部分)

```
myFile:write("--Target information")
myFile:write(string.char (10))
myValue = targetCount
myFile:write(string.format("%s%d", "targetCount = ",myValue))
myFile:write(string.char (10))
for indx = 1,targetCount do
    myValue = myTargets[indx].T_ID
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myTargets[",
            indx, "].T_ID = ", myValue))
    elseif type(myValue) == 'nil' then
        myFile:write(string.format("%s%d%s", "myTargets[", indx,
            "].T_ID = nil"))
    end
    myFile:write(string.char (10))
    myValue = myTargets[indx].T_X
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myTargets[",
            indx, "].T_X = ", myValue))
    elseif type(myValue) == 'nil' then
        myFile:write(string.format("%s%d%s", "myTargets[", indx,
            "].T_X = nil"))
    end
    myFile:write(string.char (10))
    myValue = myTargets[indx].T_Y
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myTargets[",
            indx, "].T_Y = ", myValue))
    elseif type(myValue) == 'nil' then
        myFile:write(string.format("%s%d%s", "myTargets[", :indx,
            "].T_Y = nil"))
    end
    myFile:write(string.char (10))
    myValue = myTargets[indx].T_TOW
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d%s", "myTargets[",
            indx, "].T_TOW = ", myValue))
    elseif type(myValue) == 'string' then
        myFile:write(string.format("%s%d%s%s%s%s",
            "myTargets[",
            indx, "].T_TOW = ", string.char (34), myValue,
            string.char (34)))
    end
    myFile:write(string.char (10))
end
myFile:write(string.char (10))
myFile:write(string.char (10))
```

下面是 myProjectiles 表, 输出所有的 key 和 value。当用户创建这样的数据保存函数时, 可以先写好输出表的 key/value 的主循环, 在填入具体内容前复制并粘贴。还可以更进一步, 创建一个通用函数。这样得先创建一个保存表名和对应数据集的机制, 就像之前我们处理 CSV 文件的函数那样。

#### 代码清单 10.8 游戏保存函数 (第4部分)

```
myFile:write("--Projectile information")
myFile:write(string.char (10))
myValue = pCount
myFile:write(string.format("%s%d", "pCount = ", myValue))
myFile:write(string.char (10))
myValue = pIndx
myFile:write(string.format("%s%d", "pIndx = ", myValue))
myFile:write(string.char (10))
myValue = playerProjectiles
myFile:write(string.format("%s%d", "playerProjectiles = ", myValue))
myFile:write(string.char (10))
for indx = 1, pCount do
    myValue = myProjectiles[indx].PROJ_ID
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myProjectiles[",
            indx, ".PROJ_ID = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myProjectiles[",
            indx, ".PROJ_ID = nil"))
    end
    myFile:write(string.char (10))
    myValue = myProjectiles[indx].PROJ_XTH
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myProjectiles[",
            indx, ".PROJ_XTH = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myProjectiles[",
            indx, ".PROJ_XTH = nil"))
    end
    myFile:write(string.char (10))
    myValue = myProjectiles[indx].PROJ_YTH
    if type(myValue) == "number" then
        myFile:write(string.format("%s%d%s%d", "myProjectiles[",
            indx, ".PROJ_YTH = ", myValue))
    elseif type(myValue) == "nil" then
        myFile:write(string.format("%s%d%s", "myProjectiles[",
            indx, ".PROJ_YTH = nil"))
    end
    myFile:write(string.char (10))
    myValue = myProjectiles[indx].PROJ_X
```

```

if type(myValue) == "number" then
    myFile:write(string.format("%s%d%s", "myProjectiles[",
        indx, "].PROJ_X = ", myValue))
elseif type(myValue) == "nil" then
    myFile:write(string.format("%s%d%s", "myProjectiles[",
        indx, "].PROJ_X = nil"))
end
myFile:write(string.char (10))
myValue = myProjectiles[indx].PROJ_Y
if type(myValue) == "number" then
    myFile:write(string.format("%s%d%s", "myProjectiles[",
        indx, "].PROJ_Y = ", myValue))
elseif type(myValue) == "nil" then
    myFile:write(string.format("%s%d%s", "myProjectiles[",
        indx, "].PROJ_Y = nil"))
end
myFile:write(string.char (10))
myValue = myProjectiles[indx].PROJ_SHIP
if type(myValue) == "number" then
    myFile:write(string.format("%s%d%s", "myProjectiles[",
        indx, "].PROJ_SHIP = ", myValue))
elseif type(myValue) == "string" then
    myFile:write(string.format("%s%d%s%s%s",
        "myProjectiles[", indx, "].PROJ_SHIP = ", string.char
        (34), myValue, string.char (34)))
end
myFile:write(string.char (10))
end
myFile:write(string.char (10))
myFile:write(string.char (10))

```

最后一部分是处理和分数相关的 GUI 控件，这样只要文件被载入，就可以显示出对应的状态。

#### 代码清单 10.9 游戏保存函数（第 5 部分）

```

myFile:write("--Initial GUI setup")
myFile:write(string.char (10))
myFile:write(string.format("%s%s%s%s%s%s%s",
    "ItemCommand(GUI_INGAME + 201, ", string.char (34), "SetString",
    string.char (34), ",", string.char (34), "Projectiles Fired: ",
    CommaFormatBigInteger(playerProjectiles), string.char (34), ")"))
myFile:write(string.char (10))
myFile:write(string.format("%s%s%s%s%s%s%s",
    "ItemCommand(GUI_INGAME + 202, ", string.char (34),
    "SetString",
    string.char (34), ",", string.char (34), "Targets Lost: ",
    CommaFormatBigInteger(targetDoneCounter), string.char (34), ")"))
myFile:write(string.char (10))
myFile:write(string.format("%s%s%s%s%s%s%s",

```



```

ItemCommand(GUI_INGAME + 203, ", ", string.char (34), "SetString",
string.char (34), ", ", string.char (34), "Score: ",
CommaFormatBigInteger(score), string.char (34), ")")
myFile:write(string.char (10))
if targetDoneCounter == targetCount then
    myFile:write(string.format("%s%s%s%s%s%s%s%s",
        "ItemCommand(GUI_INGAME + 204, ", string.char (34),
        "SetString", string.char (34), ", ", string.char (34), "Your
        Final Score Is ", CommaFormatBigInteger(score), string.char
        (34), ")"))
    else
        myFile:write(string.format("%s%s%s%s%s%s%s%s",
            "ItemCommand(GUI_INGAME + 204, ", string.char (34),
            "SetString", string.char (34), ", ", string.char (34),
            string.char (34), ")"))
    end
    myFile:write(string.char (10))
    myFile:write(string.char (10))
    io.close(myFile)
end
end
end

```

尽管函数很长，不过使用的方法很简单，也很容易维护（比如在数据发生变化的时候）。文件的开始部分很简单：输出基本的玩家的飞船数据（也保存了一些之后需要在界面部分使用的全局变量）。然后，我们写了一个 for 循环输出游戏中所有飞船的数据。这个处理生成了很长的游戏存档文件，不过很容易阅读。之后是所有的游戏目标数据，最后是炮弹的数据（占游戏存档的大部分内容）。

使用 dofile() 载入该文件后，可以恢复游戏的存档状态，包括所有物体的位置、推进方向和所有的炮弹。

采用这种方式保存游戏存档的另一个好处是可以很直观、高效地审阅游戏数据。还可以很容易地分辨出不合法的数据，这在游戏开发过程中很有用。

用户也可以建立特定的游戏存档文件（只需编辑这个函数输出的 Lua 文件）来为游戏测试定制不同的状态。

还可以利用这种方式为不同的剧情开发初始模板，和游戏存档一样，只是为剧情定制一些特殊的初始游戏状态。当玩家选择了某个剧情开始游戏时，只需要载入相应的初始存档文件。

### 10.3.1 案例 1——《Fronrunner》

在开发《Fronrunner》游戏时（参考图 10.4），我们需要保存游戏进度的功能。游戏

中的所有主要数据都保存在程序员用 C 开发的数据结构中。我们通过一些 LuaGlue 函数访问它们，一切都能顺利运行，直到保存游戏的时候碰到问题了。



图 10.4 《Frontrunner》游戏

保存和载入游戏是程序员的工作，因为数据保存在他们开发的结构中。起初，这套系统运转顺利，但当游戏越来越复杂时开始出现问题。随着开发的逐渐进行，我们需要越来越多的 Lua 变量来保存游戏状态（如哪些竞选活动是由玩家控制、竞选辩论进行了几轮等）。

程序开发团队不得不创建特殊的数据类型（还有 LuaGlue 函数）让我们在保存游戏前和载入游戏进度后能够访问和设定 Lua 变量值。结果这逐渐变成了一个冗余的系统，存档文件也是二进制的，无法调试。当保存或者载入游戏进度出错时，我们不得不让程序员停下手头的工作来帮我们检查哪里出了问题。

这种调试基本游戏功能的方法很浪费时间，我们不得不重新思考该如何保存游戏数据。于是，我们开始尝试使用 Lua 作为主要的保存和载入游戏进度的系统。

### 10.3.2 案例2——健身大亨

在健身大亨（Fitness Tycoon）这个游戏中，参见图 10.5，我们需要一个稳定的存档系

统可以支持游戏原型开发，还能够应付游戏规模增长的需求。另外，还要求能够方便脚本程序员检查数据和调试系统。有了前面项目的经验（《Frontrunner》游戏的经验），我们决定使用 Lua 开发存档系统。



图 10.5 健身大亨

游戏的 save 函数输出的是 Lua 脚本文件，可以被任何的编辑器打开。我们还添加了关键部分的注释帮助审阅和调试文件。编辑保存/载入功能也仅仅只需要修改一个 Lua 函数，因为载入游戏进度的工作 Lua 本身就能完成。

这样做还有一些预期外的好处。因为游戏存档文件是易于理解的 Lua 脚本，所以我们可以编辑文件创建一些特殊的游戏状态来更高效地测试游戏。例如要测试支付日功能，我们可以修改存档文件让日期变为支付日的前一天，然后载入这些数据进行测试。

## 10.4 本章小结

在本章中，我们详细介绍了如何使用 Lua 作为游戏运行时数据的主要存储和读取工具。保存和载入游戏进度是游戏开发过程中最枯燥和花时间的工作。通常，游戏数据在开发中都需要更改，程序开发团队经常会忽略存档系统的更新请求，因为这会影响一些更重要的

开发任务。把这部分工作转移到 Lua 这边可以让脚本程序员和设计师来管理游戏数据存储，并随时保持最新状态，设计师也不用操心该如何解析数据，这些 Lua 自己可以处理，而且自由更改数据还可以方便测试各种不同游戏状态。

Lua 既可以维护简单的数据，也可以通过扩展，如外部的电子表格文件或者自己创建的数据文件，来管理更复杂的大规模数据。

然后，我们学习了如何使用 Lua 收集游戏数据并保存到 Lua 文件，并且使用 Lua 内建的函数载入、解析数据。

学习了这些方法，我们就掌握了《Take Away》游戏的数据结构。在下一章中，我们会开始接触 GUI 系统，并从底层代码和 Lua 脚本两方面学习如何实现用户界面。



## 第 11 章

Chapter 11: Lua-Driven GUIs

# Lua 驱动的 GUI

### 本章要点


- GUI 系统概要
- GUI 的 C++ 类
- GUI LuaGlue 函数
- 进一步的说明
- Lua 游戏界面

Lua 在游戏开发中的一个主要用途是创建和控制 GUI。本章会详细介绍一个全功能的环境，用来创建和管理 GUI 控件，帮助我们开发各种简单或者复杂的界面。除此之外，还会介绍《Take Away》游戏的界面，以及它们的功能和交互。

## 11.1 GUI 系统概要

第 8 章已经接触过 GUI 系统的一些内容。用户界面是指图形控件的集合，并包含可以响应用户交互的 Lua 代码。在示例的 GUI 系统中，有 3 种图形控件可以使用，并采用事件驱动系统来控制界面。

### 3 种 GUI 控件

 GUI 系统中实现了 Sprite、4 种状态的按钮和文本控件。Sprite 是静态的 2D 图形控件，按钮是响应鼠标位置和按钮状态的 Sprite 集合，文本控件是按指定字体显示文本的控件。为了支持这 3 种控件，游戏 Shell 脚本需要支持 Sprite 和按照指定字体绘制文本。Sprite 对象是 CSprite；字体对象是 CFont。支持这些控件的代码在 CD-ROM 的 DX9 项目下。这 3 个控件仅仅是可以实现的最基本的 GUI 控件。

## 11.2 GUI 的 C++ 类

GUI 的代码主要是由 C++ 实现。Lua 用来控制和排版 GUI 控件，主要的处理在 C++ 代码中。CGUIObject 类是所有 GUI 控件的基类。该类的定义参考代码清单 11.1。

代码清单 11.1 CGUIObject 类

```
class CGUIObject
{
public:
    CGUIObject();
    ~CGUIObject();
    virtual const char *GetObjectTypeName();
    virtual bool Update(float fSecsElapsed,
                        CUserInterface *pParent);
    virtual bool Render(void);
    virtual int ItemCommand(const char *pCommand);
    virtual void SetPosition(float fx, float fy,
                             float fw, float fh,
                             float z);
    virtual bool SetFont(char *pFontName,
                         float normalSize);
    virtual bool KeyHit(int ascii);
    void SetID(unsigned int id);
    unsigned int GetID(void);
    RECT GetScreenRect();
    virtual bool StealMouse(int x, int y);
    void Enable(bool bDisable,
                bool bKeepDrawing);
    bool isDisabled();
    bool isDrawing();
};
```

### (1) 方法: Update

该方法用来通知对象时间的变化，如果在渲染循环外有需要的处理，那么可以在这里完成。

### (2) 方法: Render

该方法在渲染循环中被调用，绘制 GUI 控件。基本的处理是检测是否有 Sprite 载入，如果有就在屏幕上绘制。代码的实现使用了 DX9 Framework。

### (3) 方法: ItemCommand

该方法在 LuaGlue 函数中被调用，负责控件相关的处理。一般不做什么处理，不过在

许多应用程序中会用到这种方法。

**(4) 方法: `SetPosition`**

该方法设置控件在屏幕上的显示位置。所有的绘图工作都要根据这个位置进行。

**(5) 方法: `SetFont`**

该方法为控件设置字体。主要用来设置 `TextField` 控件的字体。

**(6) 方法: `KeyHit`**

该方法用来设置和控件关联的按键。如果控件使用了某个按键就返回 `true`，然后停止按键处理，否则返回 `false`（默认状态）。

**(7) 方法: `SetID` 和 `GetID`**

界面中的每个控件都有唯一的 ID。这个 ID 在 Lua 代码创建控件时被设置。在 C++ 代码中也可以取得该 ID。

**(8) 方法: `GetScreenRect`**

该方法返回控件的屏幕区域。

**(9) 方法: `StealsMouse`**

该方法用来检测是否响应在某区域的鼠标事件。

**(10) 方法: `Enable`**

该方法用来设置控件的可用状态。不可用状态的控件仍然能被显示或者隐藏在屏幕上。C++ 代码（和 `LuaGlue` 函数）可以使用 `isDisabled` 和 `isDrawing` 方法来获取可用状态和绘图状态。

因为所有的 GUI 控件都继承自这个类，所以它们拥有相同的功能。例如，它们都有背景 `sprite` 和显示文本的字体。

我们开发的控件中大部分的功能都直接来自基类。只有按钮控件实现了很多自己的特殊功能。

### 11.2.1 GUI 控件: `Sprite`

`Sprite` 控件的所有功能都在基类（`CGUIObject`）中实现了。实现这个子类是为了 `GetObjectTypeName` 方法能够返回适当的值，未来的扩展（如选择转、拉伸或者窗口配置）也会容易些。类的定义参考下面的示例：

```
class CGUISprite : public CGUIObject
{
public:
    CGUISprite();
    ~CGUISprite();
    virtual const char *GetObjectTypeName();
};
```

### 11.2.2 GUI 控件: TextField

TextField 控件在渲染时, 会在指定位置保存并绘制文本。类的定义参考下面的示例:

```
class CTextField : public CGUIObject
{
public:
    CTextField();
    ~CTextField();
    virtual const char *GetObjectTypeName();
    virtual int ItemCommand(const char *pCommand);
    virtual bool Render(void);
};
```

TextField 控件重写了 ItemCommand 方法。该方法设置、清除或者返回字符串的内容。该方法会被 LuaGlue 函数调用, 稍后的章节中会说明。Render 方法也被重写了, 因此, 在基类的 Render 方法被调用后, 绘制控件的文本。

### 11.2.3 GUI 控件: Button

Button 类是 sprite 的特殊版本。两者最大的区别在于, Button 会根据鼠标的位置选择显示 3 种图像中的一种。Button 的状态分为正常、悬停和按下。该控件会生成事件让 Lua 处理按钮状态的变化。类的定义参考下面的示例:

```
class CButton : public CGUIObject
{
public:
    CButton ();
    ~CButton ();
    virtual const char *GetObjectTypeName();
    virtual int ItemCommand(const char *pCommand);
    virtual bool Update(float fSecsElapsed,
        CUserInterface *pParent);
    virtual bool Render(void);
};
```



注意：Button 类重写了 Update 方法。该方法根据鼠标的位置和状态更新控件的状态。Button 有 3 种 sprite，每个状态对应一个，在创建时或者通过 ItemCommand 方法设置。控件可以生成 GUI\_EVENT\_BUTTON\_UP 和 GUI\_EVENT\_BUTTON\_DOWN 事件。用户按下该按钮时生成“down”事件，松开时生成“up”事件。“up”事件表明按钮被单击。

### 11.2.4 界面

正如前面所提到的，所有的控件都被放置在界面上。同一时间只有一个“激活”（active）状态的界面，它被显示、更新并向事件处理器发出输入事件。类的定义可参考代码清单 11.2。

代码清单 11.2 CUserInterface 类

```
class CUserInterface
{
public:
    CUserInterface();
    virtual ~CUserInterface();
    bool Init(const char *fname);
    bool Update(float fSecElapsed);
    bool Render(void);
    void AddGUIObject(unsigned int id, CGUIObject *pObject);
    void DeleteGUIObject(unsigned int id);
    void SetGUIObjectPosition(unsigned int id,
                              float fx, float fy,
                              float fw, float fh);
    void SetGUIObjectFont(unsigned int id,
                          char *fontName, int fontSize);
    int ItemCommand(unsigned int id, const char *pCommand);
    void EnableObject(unsigned int id,
                     bool bDisable, bool bKeepDrawing);
    void SetEventHandler(const char *pHandlerName);
    const char *GetEventHandler(void) {return m_pEventHandlerName;}
    bool KeyHit(int ascii);
    void StartTimer(float fTime);
};
```

该类的主要功能是维护 CGUIObject 对象列表，并提供访问各个 GUI 控件的统一接口。《Take Away》游戏的所有与 GUI 系统相关的方法可参考表 11.1。

表 11.1 《Take Away》游戏中与 GUI 系统相关的方法

方 法	描 述
Init	当创建新的界面时，调用该方法初始化界面并执行 Lua 脚本。该脚本创建所有的 GUI 控件，并且定义界面的事件处理函数。具体做法会在下一小节中说明

(续)

方 法	描 述
Update	该方法用来通知界面时间的变化。界面会调用所有 GUI 控件的 Update 方法, 更新每个控件
Render	界面本身不做任何渲染的工作, 在这里调用列表中所有 GUI 控件的 Render 方法
AddGUIObject	添加新的 GUI 控件
DeleteGUIObject	删除已有的 GUI 控件
SetGUIObjectPosition	该方法找到引用的 GUI 控件并设置在屏幕上的显示位置
ItemCommand	该方法找到传入的控件并调用该控件的 ItemCommand 方法
EnableObject	根据 ID 设置控件的可用状态
SetEventHandler	获取界面的 Lua 事件处理器名称
GetEventHandler	设置界面的 Lua 事件处理器名称
KeyHit	该方法用来通知界面用户按下了某个按键, 界面会通知所有的 GUI 控件, 如果没有控件和它关联, 会向界面的事件处理器发送 GUI_KEY_PRESS 事件
StartTimer	每个界面都有提供给 Lua 使用的计时器。当调用该方法时, 计时器开始计时, 经过了设定时间后, 向事件处理器发送 GUI_TIMER_EXPIRED 事件, 之后计时器变成非激活状态

## 11.2.5 GUI 管理器

GUI 管理器是 C++ 代码访问整个 GUI 系统的接口。它负责维护所有的界面和 LuaGlue 函数。CGUIManager 类是单件类 (singleton), 同时只能有一个对象。我们可以通过这个类访问整个 GUI 系统。类的定义参考代码清单 11.3。

代码清单 11.3 CGUIManager 类

```

class CGUIManager
{
private:
    static CGUIManager *m_pInstance;
public:
    static CGUIManager *GetInstance();
public:
    CGUIManager();
    ~CGUIManager();
    bool Init(cLua *pContext);
    bool StartGUI(const char *pFilename);
    bool Update(float fSecsElapsed);
    bool Render();
    void SendEvent(int iEventCode, int id,
        float arg1 = 0.0f, float arg2 = 0.0f,
        float arg3 = 0.0f, float arg4 = 0.0f);
    bool KeyHit(int ascii);
};

```

### (1) 静态方法: GetInstance

该方法让我们可以在 C++ 代码的任何地方获取 GUI 管理器对象的指针。私有指针在对象的构造函数和析构函数中更新, 在任何地方调用 `CGuiManager::GetInstance()` 可以获取该指针。这个特性让 GUI 控件可以访问整个系统, 并且让主程序可以很容易地在初始化后调用它的方法。

### (2) 方法: Init

`Init` 方法启动系统并且可以访问主程序的 Lua 环境。系统初始化完成后, 会执行 `StartGUI.lua` 脚本来启动 Lua 端的初始化处理。

### (3) 方法: Update 和 Render

`Update` 方法和 `Render` 方法与之前的类相似。调用当前界面中所有控件的对应方法。

### (4) 方法: SendEvent

发送事件到界面的事件处理器。

### (5) 方法: KeyHit

发送键盘输入到当前界面。

## 11.3 GUI LuaGlue 函数

控制 GUI 系统的 LuaGlue 函数和前面的 C++ 代码紧密关联。GUI 的 Lua 组件在系统启动时通过执行 `StartGUI.lua` 脚本初始化。该脚本至少需要启动一个界面作为初始界面。下面是 GUI 系统 Lua 端的 LuaGlue 函数。

### (1) RunGUI (filename)

该函数创建新的界面或者返回之前创建好的界面。传入的是创建界面控件和定义事件处理器的 Lua 脚本。

### (2) SetEventHandler (name)

该函数设定界面的 Lua 事件处理函数。设定的函数负责处理界面产生的所有事件, 确保每个界面都有唯一的事件处理函数, 防止界面调用错误的处理函数。GUI 事件处理函数的参数如下:

```
function SampleEventHandler(itemID, eventCode, param1, param2, param3,
param4)
    -- event handler code here
end
```

### (3) CreateItem (id, type, item specifics)

该函数用来创建界面控件。Lua 程序员为每个控件设定唯一的 ID。传入控件类型 (sprite, text field 或者 button), 和该类型控件的参数。如下所示:

■ Sprite: 图像的文件名

■ TextField: 无

■ Button: 3 种状态对应的图像文件名, 即 normal、hover、pressed

### (4) DeleteItem (id)

彻底删除界面中的控件。

### (5) SetItemPosition (id, x, y, w, h)

设定 GUI 控件的位置。

### (6) ItemCommand (id, command, ...)

该函数传递 command 给指定控件, 参数取决于控件类型和 command 类型。控件对应的 command 如下:

■ Sprite: 无

■ TextField

- ① SetString: 设定下一个参数为显示字符。
- ② AddString: 在当前显示字符后添加字符。
- ③ GetString: 返回当前显示字符。
- ④ SetColor: 设定显示字符的颜色。

■ Button: 无

### (7) SetFont (id, fontname, fontsize)

该函数设定控件字体。只适用于 TextField 控件, 也可以用于其他的扩展控件, 如 list 或者 edit box。

### (8) EnableObject (id, enable, draw)

该函数设定控件的可用状态和绘制状态。禁用的控件可以设置为可绘制但是没有用户响应, 或者设定为不可绘制。

### (9) StartTimer (time)

该函数启动 GUI 的计时器。到期后发送 GUI\_TIMER\_EXPIRED 事件到界面事件处理函数, 并禁用计时器 (可以在事件处理函数中重新开始计时)。

## GUI 事件

可以发送给界面的所有 GUI 事件如下所示：

```
GUI_EVENT_BUTTON_UP  
GUI_EVENT_BUTTON_DOWN  
GUI_KEY_PRESS  
GUI_ENTER_INTERFACE  
GUI_REENTER_INTERFACE  
GUI_TIMER_EXPIRED
```

### (1) GUI\_EVENT\_BUTTON\_UP

用户单击 GUI 按钮时发送该事件。参数 id 为发送事件的按钮 id。

### (2) GUI\_EVENT\_BUTTON\_DOWN

用户按下 GUI 按钮尚未松开时发送该事件。参数 id 为发送事件的按钮 id。

### (3) GUI\_KEY\_PRESS

用户按下键盘按键后发送该事件。参数 id 为按键的 ASCII 编码。

### (4) GUI\_ENTER\_INTERFACE

新界面发出的第一个事件。告诉 Lua 代码这是界面第一次运行，需要做一些初始化处理。

### (5) GUI\_REENTER\_INTERFACE

界面在非激活状态又被激活后发送该事件。

### (6) GUI\_TIMER\_EXPIRED

StartTimer 函数传入的时间到期后发送该事件。

## 11.4 进一步的说明

我们的测试游戏中提供的控件仅仅是开始，用这种方式可以开发用户能想象到的所有控件。用户还可以把控件组合成一个全新的控件。一些可能的控件可参考表 11.2。

表 11.2 可能的 GUI 控件

GUI 控件	说 明
Scrolling List	由 TextField 控件组成的列表，从上到下排列。按钮可以上下滚动列表
Text Edit/Entry	接收键盘输入以编辑 TextField 中的字符串

(续)

GUI 控件	说 明
Text Banner	横向滚动显示文本的 TextField
Pie Chart	根据数据显示饼图的控件
Progress Meter	显示处理进度的条形控件
Radio Buttons	单选按钮, 只能设定其中一个为“选中”状态
Check Boxes	复选按钮 (有“on”和“off”两种状态), 通过鼠标单击改变开关状态
Rotated Sprites	使用 DX9 rotation 编辑当前的 sprite 控件
Animated Sprites	多个 sprite 控件组成, draw 函数绘制其中的一个, 在特定的时间更新指定的 sprite
3D	使用 DX9 创建新的控件显示 3D 场景或者简单的对象

创建新的控件需要继承基类 CGUIObject 并且支持 ItemCommand 消息。以 TextField 控件为例, 当创建好新的类后, 需要添加代码在 LuaGlue 函数 CreateItem 中以创建控件对象。一旦完成, 用户就可以在 Lua 程序中使用该控件了。

## 11.5 Lua 游戏界面

我们已经完成了在 C++ 游戏项目中整合 Lua 界面系统的工作, 现在将学习如何使用 Lua 组装界面。第一部是设计, 这个已经完成。看看之前的设计, 可以发现我们已经规划好了 Lua 实现, 并且准备好了之后游戏中需要开发的脚本框架。

不只是案例游戏《Take Away》的开发, 在今后的开发中用户也可以采用这种方法: 设计 Lua 实现并且创建将来会在项目中使用的脚本文件的框架, 这样可以让你的脚本更专注, 而且和设计保持一致。

完成了游戏开发后, 你一定想要继续“打磨”, 但在开发阶段, 在提交完整实现给美术师之前, 让组件先运行起来通常是个很好的做法。开发者称之为原型开发, 而我们更倾向于叫它“快速实现”, 因为这不只是开发一个原型, 也是快速构建游戏的结构和流程, 验证了我们的想法后, 再强化和完善它。

### 11.5.1 界面设计原则

界面是我们用来和游戏玩家交互的图形化工具。界面是玩家在游戏中看到、听到、点击和切换的物体。有很多书籍介绍界面设计和讨论用户对不同界面的感受, 但这些不在本书的讨论范围中。一些值得阅读的书籍包括 Joel Spolsky 的《User Interface Design for Pro-

grammers》和 Jacob Neilson 的《Designing Web Usability》（主题是 Web 设计，不过对于游戏同样值得参考）。下面是一些界面设计的原则：

1) **保持界面的简洁**：不要设计复杂的、有大量按钮的界面。保持界面的简洁可以让游戏玩家的注意力集中在更重要的部分。

2) **设计舒适的视觉体验**：开发让人感觉舒服和专业的界面。记住，界面可向玩家传递开发者在游戏设计方面的品味和感觉，不要使用太多的颜色和花哨的设计。

3) **只显示重要的信息**：如果提供给玩家无限子弹的枪，就不要显示子弹数量。界面仅显示游戏中会变化且对玩家有意义的信息。

4) **提供有意义的反馈**：确保提供有用的数据，让玩家知道游戏的进程。这个信息可以是得分、血量、子弹数（这种有意义的的数据），保证通过合理的方式告诉玩家游戏的当前状态。

### 11.5.2 快速创建界面

开始编写脚本后，就不要停滞在界面的艺术细节上了。利用一些工具可以快速地创建界面（如果你听过 Programmer art 这个名字就明白是什么意思了）。我们使用 Microsoft Visio（如图 11.1 所示）创建界面模型，当然其他工具也没问题（甚至是 word 文档）。找到可以让用户快速创建全屏界面布局的工具，将其保存为图片文件，然后截出按钮和 sprite 组件。

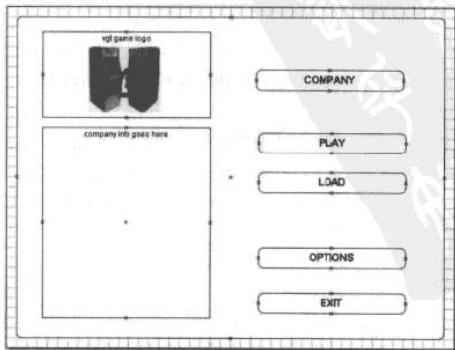


图 11.1 在 Visio 中快速创建界面布局

快速创建界面后，可以列出所有界面的流程。这个过程可以让用户测试界面的流程，检查显示的信息，看看是否遗漏了重要的部分，这些工作会花费几个小时。

### 11.5.3 载入界面

大部分游戏的开始界面是版权页或者载入界面，在《Take Away》游戏中我们也会这样做。通常，游戏数据和资源在显示载入界面时加载入内存。《Take Away》是很轻量的小游戏，几乎没有游戏载入时间，但这个屏幕可以让游戏看起来更美观。

这个界面我们不准做得很复杂，它会显示背景图像、版权声明和进度条（只是显示一段时间文字）。

在 GUI 脚本中，我们有初始化控件和为子 GUI 控件调用 `dofile()` 的处理，由于载入界面不需要这些，所以这些部分是空的。之前，我们在脚本中添加了 `SetEventHandler`（“LoadingMenuEvent”）函数设定事件处理器和事件的处理结构。之后我们会了解如何捕获和响应事件，现在只需要初始化 GUI 控件。

这个过程中第一个需要注意的地方是 `StartGUI.lua` 文件，该文件初始化游戏常量。在这个文件中为所有游戏会使用的 GUI 界面设定 ID 常量。前面提到过，GUI 在 Lua 脚本中是独立的单元——GUI 文件中的 `sprite` 可以设定和其他 GUI 文件中的控件相同的 ID，不会互相影响。不过，为了更好地组织文件，我们会为每个界面创建唯一的 GUI ID 常量，每个界面都预留了 1000 个 ID 给自己的控件（对于子界面很有用，之后会说明）。《Take Away》游戏的 GUI 常量定义如下：

```
-- Sub-interface constants
GUI_INGAME = 0
GUI_RUNTIME_SPRITES = 1000
GUI_MAIN_MENU = 2000
GUI_ESCAPE = 3000
GUI_KEY_SELECT = 4000
GUI_LOADING = 5000
GUI_END_GAME = 6000
```

`StartGUI.lua` 脚本最后调用载入界面并启动游戏，代码如下：

```
--Start with the loading screen
RunGUI("GUI_Loading.lua")
```

现在来看 `GUI_Loading.lua` 文件，首先设定背景图像：

```
CreateItem(GUI_LOADING + 100, "Sprite", "ui_bg_loading.bmp")
SetItemPosition(GUI_LOADING + 100, 0, 0, 800, 600)
```



通过这些函数，我们创建了 GUI 控件，使用 StartGUI.lua 中设定的常量，创建了 Sprite 控件，并设定了背景图 ui\_bg\_loading.bmp。之后利用 SetItemPosition() 函数设定了控件的位置 (0, 0) (屏幕左上角)，尺寸 800 像素 × 600 像素 (全屏)。

系统中实现的 GUI 控件会按照 ID 从小到大的顺序绘制。我们给 sprite 控件 ID 设置了 100 的范围，因为它们经常被用作背景图片和其他界面元素。要在全屏图像前面绘制 sprite，需要设定更大的 ID。

### 1. 文本列表

载入界面的其余部分是文本控件，我们来看看如何设定。文本控件的设定很简单，参考下面的脚本：

```
CreateItem(myID, "TextField")
SetItemPosition(myID, 335, 20, 0, 0)
SetFont(myID, "Arial", 16)
ItemCommand(myID, "SetColor", 169, 20, 231, 255)
ItemCommand(myID, "SetString", "Hello World!")
```

在这个例子中，我们首先创建了 TextField 控件，ID 为变量 myID，然后设定位置 (TextField 的大小取决于字体的大小，因此 SetItemPosition() 中的尺寸参数没有实际意义)，接着设定字体 (这是系统字体，因此在运行游戏的系统上需要安装该字体，并且可以在字体控制面板中找到)。

下一步是设定颜色——简单的 RGBA 值。前 3 个数字是 RGB 颜色值 (0 ~ 255)，第 4 个数字是文本的 alpha 透明度，255 代表不透明。

最后根据传入参数设定实际显示的文本。我们可以直接设定传入的文本，不过如果需要对游戏进行本地化工作，或者之后需要大量修改字符，可以把它们交给对应的 Lua 脚本处理 (在 Lua 版本的设计文档中已经设定好了)。

在我们的游戏中，使用 Text.lua 脚本载入一张数据表，表中包含了需要显示的文本。因此，我们首先需要创建一个函数，如下所示，可以很容易地取得字符串：

```
function GetText(localID)
    return textTable[localID]
end
```

该函数简单地从表中取得数据。如果需要改变表内容或者操作字符数据，可以在这个函数中完成，避免了在其他脚本中寻找 textTable 的引用。

在脚本中开始初始化 textTable，然后使用相同的界面对象的 GUI ID 往表中添加内容

(在 StartGUI.lua 脚本中已经载入了该数据)。载入界面需要下面的内容：

```
--define empty table
textTable = {}
--Loading text
textTable[GUI_LOADING + 200] = "Version 1.0"
textTable[GUI_LOADING + 201] = "Copyright 2005, Charles River Media"
```

在这里，表的 Key 是基于 GUI 界面对象的，会生成许多未使用的 Key。不过请记住，Lua 不会根据一个范围的值来创建表，而是根据设定的单个值来创建表内容，因此我们不必担心这个方法会牺牲任何表的存储空间。

首先为界面创建两个文本控件：

```
--200s Text items
CreateItem(GUI_LOADING + 200,"TextField")
SetItemPosition(GUI_LOADING - 200, 335, 20, 0, 0)
SetFont(GUI_LOADING + 200, "Arial", 16)
ItemCommand(GUI_LOADING + 200, "SetColor", 169,20,231,255)
ItemCommand(GUI_LOADING + 200, "SetString", GetText(GUI_LOADING + 200))

CreateItem(GUI_LOADING + 201,"TextField")
SetItemPosition(GUI_LOADING - 201, 220, 555, 0, 0)
SetFont(GUI_LOADING + 201, "Arial", 16)
ItemCommand(GUI_LOADING + 201, "SetColor", 169,20,231,255)
ItemCommand(GUI_LOADING + 201, "SetString", GetText(GUI_LOADING + 201))
```

最后一个文本控件是进度条，定义如下：

```
loadString = "."
CreateItem(GUI_LOADING + 202,"TextField")
SetItemPosition(GUI_LOADING + 202, 212, 500, 0, 0)
SetFont(GUI_LOADING + 202, "Arial", 48)
ItemCommand(GUI_LOADING + 202, "SetColor", 177,174,255,255)
ItemCommand(GUI_LOADING + 202, "SetString", loadString)
```

为该控件设定普通的字符串，不过在初始阶段我们使用 loadString 变量（我们会检测字符串的长度以决定进度条需要移动多远）。为了让程序运行，需要看看界面文件中的事件处理程序。

## 2. 载入界面的事件处理

载入界面中需要处理的唯一的事件是计时器事件，在设定事件处理器前，我们使用 StarTimer (0.25) (LuaGlue 函数) 开启计时器。该函数为界面开启 GUI 计时器，时间到后发出 GUI\_TIMER\_EXPIRED 事件（例子中是 0.25s）。事件处理代码如下所示：

```
function LoadingMenuEvent(id, eventCode)
    if eventCode == GUI_TIMER_EXPIRED then
        if string.len(loadString) < 21 then
            loadString = string.format("%s%s", loadString, ".")
            ItemCommand(GUI_LOADING + 202, "SetString", loadString)
            StartTimer(.15)
        else
            RunGUI("GUI_MainMenu.lua")
        end
    end
end
end -- event handler
```

处理函数中，首先检测 loadString 的长度是否小于 21（显示 20 个或者更少的句号）。如果小于，继续等待并且在 loadString 后添加句号，然后更新 TextField。最后开始另一个计时器，会重新执行该事件处理函数。

一旦进度条足够长，string.len() 函数将跳出 if 控制结构进入 else 部分，脚本会运行主界面 GUI。

### 3. 载入对象

在这个例子中，我们实现了一个不显示实际进度的进度条。还可以使用相同的方法在其他数据密集型的游戏中载入需要的数据，参考下面的例子，这是另外一个游戏：

```
if periodCount < 21 then
    if periodCount == 1 then
        LoadSounds()
    end
    if periodCount == 5 then
        AddImageToCache('female_thin.bmp')
        AddImageToCache('female_normal.bmp')
        AddImageToCache('female_heavy.bmp')
        AddImageToCache('male_thin.bmp')
        AddImageToCache('male_normal.bmp')
        AddImageToCache('male_heavy.bmp')
    end
    if periodCount == 10 then
        AddModelToCache('female_heavy.mlm')
    end
    if periodCount == 15 then
        AddModelToCache('female_normal.mlm')
    end
    loadString = string.format("%s%s", loadString, ".")
    ItemCommand(GUI_LOADING + 202, "SetString", loadString)
    StartTimer(.15)
else
    RunGUI("GUI_MainMenu.lua")
end
```

上面的例子在进度条的不同时间段进行了重要的业务处理，如将音乐、贴图和模型载入内存。

如果开发自己的 LuaGlue 函数，就可以使用 Lua 控制游戏载入，在这个事件循环中，预缓存、载入游戏数据（底层数据类型）或者通过 `dofile()` 函数执行 Lua 脚本。

### 11.5.4 主菜单界面

游戏《Take Away》的主菜单界面，如图 11.2 所示，是玩家进入游戏前的等待区域。在主界面中，我们可以看到积分榜、开始新游戏、载入游戏进度、重新设置控制按键或者退出游戏返回 Windows 桌面等内容。



图 11.2 《Take Away》主菜单界面

《Take Away》主菜单界面包含了背景图片、一些 TextField（显示积分榜和存档日期），以及 4 个按钮。背景图片和 TextField 的设定和载入界面相同，这里不再说明。需要注意的是，因为文本控件的内容来自游戏数据，所以并不从文本表中获取值，而是在更新函数中设定（参考下面的章节）。

#### (1) 按钮

这个主菜单界面的新控件是按钮。《Take Away》的“开始新游戏”（New Game）按钮定义如下：

```
--300s Buttons
-- to start a new game
CreateItem(GUI_MAIN_MENU + 300, "Button", "uib_newgame_up.bmp",
"uib_newgame_hv.bmp", "uib_newgame_dn.bmp")
SetItemPosition(GUI_MAIN_MENU + 300, 452, 123, 293, 67)
```

我们创建了 GUI button 控件，并且设定了 3 个位图，分别对应 up、down 和 hover 状态，如图 11.3 所示，然后设定按钮的位置和大小。



图 11.3 “开始新游戏”按钮的 up、down 和 hover 状态

注意：在游戏例子中，我们尽量保持控件和原图片大小一致，否则会因为 DirectX 的自动拉伸功能使图片失真。

为了在脚本中使用按钮，我们捕获 GUI\_EVENT\_BUTTON\_UP 事件代码。该事件通过变量 ID 传递 GUI 对象的 ID。可以根据变量的值来判断哪个按钮被按下，然后进行相应处理。“开始新游戏”按钮的按下处理代码如下：

```
if eventCode == GUI_EVENT_BUTTON_UP then
    if id == GUI_MAIN_MENU + 300 then
        PlaySound(1, "button.wav")
        gFromEscape = "no"
        RunGUI("GUI_InGame.lua")
    end
end
```

可以看到例子中检测了对象的 ID，如果匹配则执行那 3 行 Lua 脚本。这里播放了按钮单击时的简单音效，设定了全局变量（会在 InGame 界面中使用），并执行 GUI\_InGame.lua 脚本，开始游戏。

## (2) 进入界面

在主菜单界面事件处理器的前两个事件如下：

```
GUI_ENTER_INTERFACE  
GUI_REENTER_INTERFACE
```

在第一次载入界面时发出这些事件。第一个事件在程序运行过程中，首次进入界面时发出。REENTER 事件在之后重新进入该界面时发生。区分这两个事件是为了让我们可以分别处理只运行一次的和每次都运行的初始化处理。

在《Take Away》游戏中，所有的调用的函数都定义在 LuaSupport.lua 文件，该文件在程序开始时载入（这些函数是在内存中编译好的）。

在首次运行某界面时（GUI\_ENTER\_INTERFACE），会做如下处理：

```
LoadSettings()  
SortScoreLists()  
UpdateMainMenu()
```

第一个函数的代码如下：

```
function LoadSettings()  
--Loads the current setting and high scores  
--Creates the pathway for the file  
local fileName = io.open("SaveGames\\Take_Away_Saved_Settings.lua",  
"r")  
--Checks for the file's existence  
if fileName ~= nil then  
--Reads the file, closes it, and begins the action  
dofile("SaveGames\\Take_Away_Saved_Settings.lua")  
fileName:close()  
else  
gSavedGameData = ""  
gThrustKey = 32  
gShootKey = 112  
gTurnLeftKey = 91  
gTurnRightKey = 93  
gHighScoreNum = 0  
myHighScoresDate = {}  
myHighScoresAmount = {}  
EnableObject(GUI_MAIN_MENU + 301, 0, 0)  
end  
end
```

该函数载入保存游戏设定的文件。首先检测文件是否存在，因为在玩家第一次玩游戏之前是没有设定文件的。如果文件存在，就使用 dofile() 函数载入并运行它。如果不存在，

就为变量设定默认的初始值。

第二个函数处理积分榜的显示：

```
function SortScoreLists()
  if gHighScoreNum > 0 then
    for indx = 1, gHighScoreNum do
      for i = gHighScoreNum, indx+1, -1 do
        if myHighScoresAmount[i] > myHighScoresAmount[i-1] then
          --swap numbers
          t = myHighScoresAmount[i]
          myHighScoresAmount[i] = myHighScoresAmount[i-1]
          myHighScoresAmount[i-1] = t
          --swap dates
          t = myHighScoresDate[i]
          myHighScoresDate[i] = myHighScoresDate[i-1]
          myHighScoresDate[i-1] = t
        end
      end
    end
  end
end
```

该函数对 myHighScoresAmount 表以及关联的 myHighScoresDate 表排序。如果只排序一个表，可以使用 table.sort() 函数，因为这里要保持日期和分数的关联，所以需要手动排序。

最后，使用下面的函数更新主界面的 GUI 控件：

```
function UpdateMainMenu()
  --This updates the Date and Score portions of the Main Menu
  for indx = 1,10 do
    if myHighScoresDate[indx] == nil then
      ItemCommand(GUI_MAIN_MENU + 200 + indx, "SetString", "")
      ItemCommand(GUI_MAIN_MENU + 210 + indx, "SetString", "")
    else
      ItemCommand(GUI_MAIN_MENU + 200 + indx, "SetString",
        myHighScoresDate[indx])
      ItemCommand(GUI_MAIN_MENU + 210 + indx, "SetString",
        CommaFormatBigInteger(myHighScoresAmount[indx]))
    end
    ItemCommand(GUI_MAIN_MENU + 200, "SetString", gSavedGameDate)
    if gSavedGameDate == "" then
      EnableObject(GUI_MAIN_MENU + 301, 0, 0)
    else
      EnableObject(GUI_MAIN_MENU + 301, 1, 1)
    end
  end
end
```

该函数遍历 myHighScoresDate 表（总共 10 个数据，不管有没有值）。如果包含值，就生成 TextField，并设定日期和分数。最后，在“载入游戏”按钮下方显示最后保存游戏的日期。

对比“开始新游戏”和“保存游戏”（Saved Game）按钮的事件脚本，会发现只有一点不同：“保存游戏”按钮调用 InGame 界面后调用 LoadGame()。首先载入 InGame 界面是因为飞船、敌舰、补给箱和炮弹是属于 InGame 界面的一部分。载入界面实际上是设定 InGame 界面为激活状态，尽管当前界面还是游戏主界面。

LoadGame() 函数的代码如下：

```
function LoadGame()
--Loads a saved game
--Creates the pathway for the file
local fileName = io.open({'SaveGames\\Take_Away_Saved_Game.lua'},
"r")
--Checks for the file's existence
if fileName == nil then
--Clears all projectiles
for indx = 1,pCount do
if myProjectiles[indx].PROJ_ID == nil then
DeleteItem(myProjectiles[indx].PROJ_ID)
end
end
--Clears all targets
for indx = 1,targetCount do
if myTargets[indx].T_ID == nil then
DeleteItem(myTargets[indx].T_ID)
end
end
--Reads the file, closes it, and begins the action
dofile("SaveGames\\Take_Away_Saved_Game.lua")
fileName:close()
StartTimer(refreshRate)
end
end
```

这个函数非常简单，首先检测游戏存档是否存在，如果存在，清除所有炮弹和目标数据（这样做是防止在游戏运行状态时开始新游戏，那样的话界面中的目标和炮弹会影响新游戏的初始化）。清除工作完成后，通过 dofile() 函数读取存档文件（InGame 界面已经初始化了所有的数据，游戏存档文件只是更改这些值）载入新游戏数据。最后启动 InGame 计时器，它是控制游戏运行的主要机制。

“Controls”按钮可以载入 GUI\_KeySelect.lua 文件，后面会介绍由这个按钮打开的 Con-



trols 界面。

“退出”(Exit)按钮调用 SaveSettings() 函数(参考 LuaSupport.lua 文件)保存所有游戏设定。该函数类似于前一章的 SaveGame() 函数,这里就不再详细说明,只需要看一下源文件就可以知道是如何保存游戏数据的(如分数和按钮设定)。

最后,“退出”按钮调用 LuaGlue 函数 QuitProgram(), 关闭游戏 Shell 和 Lua 环境,并退出游戏。

### 11.5.5 Controls 界面

Controls 界面允许玩家设定控制游戏的 4 个按键(左转、右转、前进和开火)。玩家通过单击对应的控制设定并输入键盘按键来重新设定控制键。界面在 GUI\_KeySelect.lua 脚本中定义,如图 11.4 所示。



图 11.4 Controls 界面

做法和之前一样,创建背景图像,设定按钮和 TextField。为 TextField 设定文本内容时,我们执行下面的函数:

```
ItemCommand(GUI_KEY_SELECT + 200, "SetString",
ASCIIToString(gThrustKey))
```

该函数在 LuaSupport.lua 文件中,把 ASCII 编码处理成我们可以理解的字符,如下

所示:

```
function ASCIIToString(id)
    if id == 8 then
        myString = "Backspace"

        elseif id == 9 then
            myString = "Tab"
        elseif id == 13 then
            myString = "Enter"
        elseif id == 32 then --Space Bar
            myString = "Space Bar"
        else
            myString = string.char(id)
        end
    end
    return myString
end
```

该函数使用 `string.char()` 函数返回字符给 `TextField`。我们加入了一些检测,一些特殊的按键(如空格键、回车键)不直接使用 `string.char()` 函数处理,而是返回更容易理解的字符串来表明所按的键。

接着来看看事件处理,这里使用退格键返回主界面。其他按键都可以用来设定控制键。我们创建一个模式来负责该处理,定义为 `keySelect` 变量,事件处理脚本如下:

```
if id == GUI_KEY_SELECT + 301 then
    PlaySound(1, "button.wav")
    keySelect = 1
    HighlightText(GUI_KEY_SELECT + 200)
end
```

脚本通过为每个单击的按钮设定 `keySelect` 值来设定 `mode`,然后执行 `HighlightText()` 函数,传入关联的 `TextFiled` 控件 ID。函数代码如下:

```
function HighlightText(id)
    for indx = GUI_KEY_SELECT + 200, GUI_KEY_SELECT + 200 + 3 do
        ItemCommand(indx, "SetColor", 176,173,254,255)
    end
    if (id > GUI_KEY_SELECT + 199) and (id < GUI_KEY_SELECT + 204) then
        ItemCommand(id, "SetColor", 255,0,0,255)
    end
end
```

该函数首先设定界面上所有 `TextField` 控件的默认颜色,然后将传入的控件 ID 对应的 `TextFiled` 控件设为红色。代码检测了控件的 ID 范围,不在范围内的就设定为原来的颜色。

在进入界面或者重新打开界面时，可以调用 `HighlightText(0)` 重置所有的 `TextField` 为默认值。

最后，事件处理函数根据当前的 `keySelect` 模式，获取按键值并执行下面的脚本：

```
elseif keySelect == 1 then
    --Thrust
    if (gShootKey == id) and (gTurnLeftKey == id) and
        (gTurnRightKey == id) and (27 == id) then
        gThrustKey = id
        ItemCommand(GUI_KEY_SELECT + 200, "SetString",
            ASCIIToString(id))
    end
    keySelect = 0
    HighlightText(0)
```

代码会检测是否设定了重复的按键，如果没有，就设定该值并更新 `TextField`，重置 `keySelect` 模式为 0（表示没有进入设定状态），最后恢复 `TextField` 的默认颜色。

### 11.5.6 InGame 界面

下一章我们会详细介绍 `InGame` 界面的处理，包括游戏流程和人工智能。这个小节只涉及界面控件部分。

`InGame` 界面的核心部分是一些 `TextField` 用来显示玩家的分数、发射子弹数以及失去了多少补给箱。

除了核心的游戏流程元素，该界面最让人感兴趣的是菜单子页面。在设计游戏界面时，一些界面可以很明确地为全屏，如我们游戏中的主界面。这些界面是完全独立的。

通常，在实时的游戏模式下，需要在不同的时间显示不同的界面，还要保持操作的舒适性（不是那种接收所有输入的界面）。如果想在主窗口中显示野蛮人战士的数量，同时还能单击界面上的其他按钮和一些与数量显示无关的操作。这种情况下，我们就会用到子界面，它允许在一个父界面下显示并操作多个子界面。一个子界面的例子如图 11.5 所示。

子界面是基于 `GUI` 有限的 `ID` 范围和关联的事件处理器。在前面的章节中，我们了解到 `ID` 在 `StartGUI.lua` 中被设定成了常量，这些值是基数，界面中所有的控件 `ID` 都是根据这个数字生成，注意不要设定 1000 个以上的 `ID`。我们可以使用下面的函数来控制 `GUI` 的显示（参考 `LuaSupport.lua` 代码文件）：

```

function ClearGUI(id)
    for indx = 1,1000 do
        EnableObject(id + indx, 0, 0);
    end
end
function RestoreGUI(id)
    for indx = 1,1000 do
        EnableObject(id + indx, 1, 1);
    end
end
end

```

这些函数简单地遍历了所有可能的 ID 值，并调用 LuaGlue 函数 EnableObject() 激活或者禁用整个界面（没有使用的 ID 会被忽略）。函数需要传入要处理的 ID 基数常量。控件的激活状态被设为 0 时不会显示，也不接收输入。

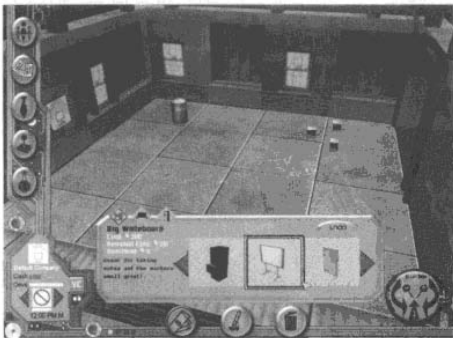


图 11.5 子界面的例子：一个购买物品的小界面，这时 InGame 界面仍然处于活动状态

在 GUI\_InGame.lua 文件的最上面，会看到下面的语句：

```
dofile("Scripts\\GUI_Escape.lua")
```

该语句载入 Escape 界面（暂停游戏，可以保存、返回或者退出游戏）。同时，InGame 界面的 GUI\_ENTER\_INTERFACE 事件会调用 InitialSetup() 函数，在它的末尾，代码如下：

```

ClearGUI(GUI_ESCAPE)
gEscapeOn = 0

```

该函数会清除 Escape 界面。用户会注意到我们设定全局变量 gEscapeOn 为 0，这个值表明 Escape 界面是否正在显示。

再来看一下 InGame 事件处理函数的开始部分，有下面的初始结构：

```
if EscapeEvent(id, eventCode) ~= 1 then
end -- escape event
```

该 if 语句包括了整个 InGame 的事件处理函数，它首先执行函数 EscapeEvent() 并判断结果，然后确定是否执行 InGame 的事件处理脚本。

在 GUI\_Escape.lua 中，有一个函数很像事件处理函数（但我们不会使用 SetEventHandler() 函数设定它为事件处理函数）。在 Escape 的事件处理函数开始部分，设定 result 变量为 0，如果接收到输入就设定 result 为 1。该值会传给 InGame 事件处理函数，说明输入是否被处理。

通过这种嵌套的事件处理（在 12 个子界面中都采用了这个方式），可以保证其他界面运行的同时，输入也能被正确的界面处理。

可以看到 InGame 事件处理函数响应 <Esc> 按键，如下所示：

```
if id == 27 then --Esc
    if gEscapeOn == 0 then
        RestoreGUI(GUI_ESCAPE)
        gEscapeOn = 1
    end
end
```

如果 Escape 界面还处于未显示状态，它会被显示，并设定全局变量 gEscapeOn。相反，在 Escape 事件处理函数中，如果按下 <Escape> 键，会执行下面的脚本：

```
if id == 27 then --Esc
    if gEscapeOn == 1 then
        ClearGUI(GUI_ESCAPE)
        gEscapeOn = 0
        StartTimer(refreshRate)
        result = 1
    end
end
```

脚本清除界面，设定 mode 变量，并返回 1，这时 InGame 事件处理函数会忽略剩下的处理。还请注意 StartTimer() 函数，在 Escape 菜单中暂停后，它会重新开始游戏。

## 11.6 本章小结

在本章中，我们学习了如何使用 Lua 在 GUI 系统中构建界面控件。通过控件命令和 LuaGlue 函数的关联，可以在 Lua 脚本中创建和控制 GUI 控件并响应界面事件。

然后介绍了《Take Away》游戏中的主要界面，看到了这些界面是如何在游戏开发过程中逐步完成的。下一步是介绍游戏逻辑和游戏事件系统，这样我们就可以在 Lua 环境中创造游戏体验了。



## 第 12 章

——Lua 游戏编程

# Lua 游戏编程

### 本章要点

- 游戏主循环
- 井字棋
- 《Take Away》

在前面章节中，我们使用 LuaGlue 函数开发了 GUI 系统，并学习了如何处理界面事件。这是一些主要的工作，但对于游戏开发来说还不够，它们仅仅是开发游戏的一个基础。

在这一章中，我们要学习使用 Lua 开发整个游戏的逻辑部分，其中包括一个回合制游戏和即时游戏《Take Away》。主要内容是主控制系统，也称做游戏主循环，用来控制游戏从开始到结束的整个流程。我们将学习如何处理事件以及驱动游戏进程的时间触发器，并进一步深入逐渐完善的《Take Away》游戏底层细节部分。

## 12.1 游戏主循环

到目前为止，我们已经有了《Take Away》游戏的静态实现，包括界面和控件的初始化，但还没有游戏的概念。抛开游戏在语义上的定义，我们需要归纳出最基本的元素，帮助我们了解游戏开发的结构。

想一想这个最简单的游戏：井字棋（tic-tac-toe）。尽管游戏的设计限制了游戏时间大概只有 30s，但还是一个有意思的例子。在这个游戏里，可以看到两个游戏的核心元素：每个玩家每个回合可以画一个 O 或者 X 符号，当某个玩家将 3 个同样的符号连成一线或者格子被填满时（和局）游戏结束。

这个游戏有回合的概念，还有胜利条件。胜利条件是让游戏结束的必须满足的参数。在《Take Away》游戏中，我们遵循了老式街机游戏无止境的游戏设计风格，只有玩家的失误才能结束游戏（玩家的成就是游戏结束时的分数）。游戏可以一直进行，直到所有的

补给箱都被移出屏幕。

真正的挑战是在 Lua 脚本的上下文中定义游戏的流程。每个游戏都需要某种程度的游戏主控制循环或者每回合、每个时间片（像《Take Away》这样的即时游戏）必须完成的处理。首先，我们来看看井字棋这个游戏。

## 12.2 井字棋

井字棋是回合制游戏：一方玩家先移动，然后另一方移动，以此类推，直到游戏结束。继续分解这个过程可以得到每个回合的具体处理，如下所示：

- 玩家在  $3 \times 3$  的棋盘上放置自己的符号（X 或者 O）。
- 如果玩家的 3 个符号连成一线（横、竖、斜线）则获胜。
- 获胜一方的 3 个相连的符号会标上连接线表示胜利。
- 如果玩家下满了整个棋盘还没有出现 3 个相连的符号，则判定为和局。

当然，假设棋盘在开局时是空的。我们需要创建一个表示棋盘的变量，以便在任何时候都可以追踪棋局状态（如图 12.1 所示）。通过这个数据，我们可以绘制游戏的当前状态。

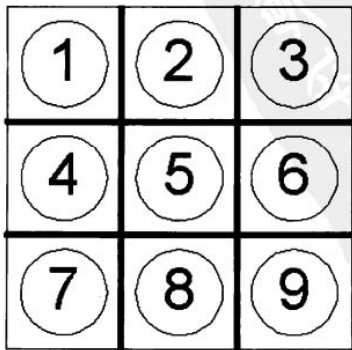


图 12.1 为井字棋游戏的每个格子设定的编号



如果想创建 Lua 数据结构来匹配图 12.1 的棋盘，可以使用下面的表：

```
myBoard = {0,0,0,0,0,0,0,0,0}
```

这段代码表示游戏开始时的空棋盘。每个游戏开始时，都需要恢复游戏的初始设定，这样随着每回合的移动，游戏会更接近胜利条件。

### 12.2.1 游戏的初始化



在井字棋游戏的例子中，我们会预先生成棋盘中的所有可能的图形元素（因为是有限的），之后再根据棋盘的状态来设定显示或者不显示。在 CD-ROM 的 GUI\_InGmae.lua 文件中，可以看到下列代码：

```
--board
CreateItem(100, "Sprite", "board.bmp")
SetItemPosition(100, 250, 150, 300, 300)

--X pieces
EX = 110
CreateItem(EX + 1, "Sprite", "piece_x.bmp")
SetItemPosition(EX + 1, 270, 170, 60, 60)
EnableObject(EX + 1, 0, 0)

CreateItem(EX + 2, "Sprite", "piece_x.bmp")
SetItemPosition(EX + 2, 370, 170, 60, 60)
EnableObject(EX + 2, 0, 0)
```

我们在每个格子上都创建了 X 和 O 符号，以及用来表示胜利的线。通常，这种做法有些多余，不过在这种小游戏里是没有问题的。然后，使用 EnableObject() 函数控制图形的渲染。在 EnableObject() 函数中，传入的第一个参数是 GUI 控件 ID，然后是控制渲染和输入的标记值，如下所示：

```
EnableObject(EX + 2, 0, 0)
```

如果第二个参数是 1，则表明需要在屏幕上显示，如果第三个参数是 1，则表明可以接收用户输入。

之后可以创建一个函数来初始化游戏和所有的 GUI 控件，如代码清单 12.1 所示。

代码清单 12.1 初始化游戏和 GUI 控件的函数

```
function InitGame()
    --turn off graphics
    --first, the pieces
    for indx = 1,9 do
```

```
    EnableObject(EX + indx, 0, 0)
    EnableObject(OH + indx, 0, 0)
end
--the bars
for indx = 1,3 do
    EnableObject(H_BAR + indx, 0, 0)
    EnableObject(V_BAR + indx, 0, 0)
end
EnableObject(D_BAR + 1, 0, 0)
EnableObject(D_BAR + 2, 0, 0)
--the text
for indx = 1,5 do
    EnableObject(GUI_INGAME + 200 + (indx * 10), 0, 0)
end
--set up the game data
myBoard = {0,0,0,0,0,0,0,0}
theWinner = -1
--set up for the first turn
if math.random(1,100) > 49 then
    curTurn = EX
    ItemCommand(GUI_INGAME + 250, "SetString", "X's turn: left click
to place")
else
    ItemCommand(GUI_INGAME + 250, "SetString", "O's turn: left click
to place")
    curTurn = OH
end
EnableObject(GUI_INGAME + 250, 1, 1)
end
```

该函数首先处理了所有界面上定义的控件，并设定了它们的初始值，然后初始化棋盘和 theWinner 变量（表明游戏状态是游戏中、和局或者某一方获胜），最后，函数随机决定先手的一方。函数执行后（在 GUI\_InGame.lua 脚本的 GUI\_ENTER\_INTERFACE 事件中启动）游戏就可以开始了。游戏结束时，再调用该函数重置游戏。

## 12.2.2 游戏回合处理

完成了游戏的设定后，现在来看一下如何处理游戏的回合。回合开始时，只有一方可以行动，一旦他在棋盘上放下棋子，游戏系统会判定游戏状态，决定是否结束游戏或者轮到下一个玩家。“在棋盘上放置棋子”这句话是关键，因为它可以用来判断回合是否结束。在例子中，玩家使用鼠标单击棋盘——游戏会触发 GUI\_MOUSE\_BUTTON\_UP 事件。事件的处理如下：

```
if eventCode == GUI_MOUSE_BUTTON_UP then

    if theWinner == -1 then
        MakeMove()
    end
end
```

我们使用 if 语句判定 theWinner 变量值是 -1 时, 表明游戏正在进行, 需要处理玩家的鼠标点击事件。通过这个简单的事件, 可以控制游戏回合及处理结果。代码在 MakeMove() 函数中, 如代码清单 12.2 所示。

代码清单 12.2 MakeMove() 函数

```
function MakeMove()
    thePos = GetBoardLocation(GetMousePosition())
    if thePos == -1 then
        --turn on the space
        EnableObject(curTurn + thePos, 1, 1)
        --change text

    if curTurn == EX then
        ItemCommand(GUI_INGAME + 250, "SetString", "O's turn: left
        click to place")
    else
        ItemCommand(GUI_INGAME + 250, "SetString", "X's turn: left
        click to place")
    end
    --now update the table
    myBoard[thePos] = curTurn
    --check for win
    theWinner = -1
    theWinner, slashID = WinCheck()
    if theWinner == -1 then
        --no winner
        --now swap turns
        if curTurn == OH then
            curTurn = EX
        else
            curTurn = OH
        end
    elseif theWinner == 0 then
        --cat's game
        EnableObject(GUI_INGAME + 210, 1, 1)
        EnableObject(GUI_INGAME + 240, 1, 1)
        EnableObject(GUI_INGAME + 250, 0, 0)
    else
        --we have a winner
        EnableObject(GUI_INGAME + 250, 0, 0)
        EnableObject(GUI_INGAME + 240, 1, 1)
        EnableObject(slashID, 1, 1)
        if theWinner == EX then
            EnableObject(GUI_INGAME + 220, 1, 1)
        else
            EnableObject(GUI_INGAME + 230, 1, 1)
        end
    end
end
end
end
end
```

读者可以仔细阅读一下该函数，它包含了井字棋游戏的核心部分，可以帮助我们学习之后的《Take Away》游戏逻辑。首先，判断玩家点击了棋盘的什么位置，参考下面的代码：

```
thePos = GetBoardLocation(GetMousePosition())
```

GetMousePosition() 函数是 C++ 编写的 LuaGlue 函数（第 13 章中会详细介绍），该函数返回当前鼠标的 x 和 y 坐标。我们将结果直接传给 GetBoardLocation() 函数，它会返回玩家点击的棋盘位置（棋盘外的位置返回 -1）。函数如代码清单 12.3 所示。

代码清单 12.3 GetBoardLocation() 函数

```
function GetBoardLocation(myX, myY)
    --set default response (not valid)
    myPos = -1
    --modify values to make it easier to comprehend
    myX = myX - 250
    myY = myY - 150
    --now check for the click area
    if (myX > 0) and (myX < 100) and (myY > 0) and (myY < 100) then
        myPos = 1
    end
    if (myX > 100) and (myX < 200) and (myY > 0) and (myY < 100) then
        myPos = 2
    end
    if (myX > 200) and (myX < 300) and (myY > 0) and (myY < 100) then
        myPos = 3
    end
    if (myX > 0) and (myX < 100) and (myY > 100) and (myY < 200) then
        myPos = 4
    end
    if (myX > 100) and (myX < 200) and (myY > 100) and (myY < 200) then
        myPos = 5
    end
    if (myX > 200) and (myX < 300) and (myY > 100) and (myY < 200) then
        myPos = 6
    end
    if (myX > 0) and (myX < 100) and (myY > 200) and (myY < 300) then
        myPos = 7
    end
    if (myX > 100) and (myX < 200) and (myY > 200) and (myY < 300) then
        myPos = 8
    end
    if (myX > 200) and (myX < 300) and (myY > 200) and (myY < 300) then
        myPos = 9
    end
    if myPos == -1 then
        --if the click is legal, see if the space is occupied
        if myBoard[myPos] == 0 then
```

```

        myPos = -1
    end
end
return myPos
end

```

该函数根据棋盘上 9 个格子的屏幕位置来为 myPos 变量赋值，在函数结束时返回该变量。如果没有在格子范围内，则返回 -1。

检测了所有位置后，函数接着检测存储当前棋盘状态的 myBoard 表，判断当前位置是否已经有棋子，如果有则返回 -1。

如果当前位置可以放置棋子，则设定该位置的当前玩家的棋子状态为开启，如下所示：

```
EnableObject(curTurn + thePos, 1, 1)
```

我们使用 curTurn 变量，因为有两套 GUI 控件（EX 和 OH），并且设定了和表索引位置对应的 GUI 控件，因此，我们可以在调用函数时使用这些值在指定位置为玩家激活 GUI 控件。

除了改变屏幕显示的内容，下一步的处理是更新棋盘状态，参考下面的代码：

```
myBoard[thePos] = curTurn
```

这个处理是为了在当前回合，在棋盘的指定位置设定值（EX 和 OH）。

下一个任务是判定更新后的棋盘的状态，使用下列语句：

```
theWinner, slashID = WinCheck()
```

WinCheck() 函数返回棋盘状态（在 theWinner 变量中）以及胜利提示线的 ID。函数代码如代码清单 12.4 所示。

代码清单 12.4 WinCheck() 函数

```

function WinCheck()
    --set default for on-going
    theGame = -1
    theID = -1
    --first, check for cat's game
    openSpace = false
    for indx = 1,9 do
        if myBoard[indx] == 0 then
            openSpace = true
        end
    end
end

```

```
if openSpace == false then
    --no open spaces, so cat's game
    theGame = 0
end

--now check for a win
--row1
if myBoard[1] == curTurn then
    if myBoard[2] == curTurn then
        if myBoard [3] == curTurn then
            theGame = curTurn
            theID = H_BAR + 1
        end
    end
end

--row2
if myBoard[4] == curTurn then
    if myBoard[5] == curTurn then
        if myBoard [6] == curTurn then
            theGame = curTurn
            theID = H_BAR + 2
        end
    end
end

--row3
if myBoard[7] == curTurn then
    if myBoard[8] == curTurn then
        if myBoard [9] == curTurn then
            theGame = curTurn
            theID = H_BAR + 3
        end
    end
end

--col1
if myBoard[1] == curTurn then
    if myBoard[4] == curTurn then
        if myBoard [7] == curTurn then
            theGame = curTurn
            theID = V_BAR + 1
        end
    end
end

--col2
if myBoard[2] == curTurn then
    if myBoard[5] == curTurn then
        if myBoard [8] == curTurn then
            theGame = curTurn
```

```

        theID = V_BAR + 2
    end
end
end
--col3
if myBoard[3] == curTurn then
    if myBoard[6] == curTurn then
        if myBoard[9] == curTurn then
            theGame = curTurn
            theID = V_BAR + 3
        end
    end
end
end
--diag1
if myBoard[1] == curTurn then
    if myBoard[5] == curTurn then
        if myBoard[9] == curTurn then
            theGame = curTurn
            theID = D_BAR + 1
        end
    end
end
end
--diag2
if myBoard[3] == curTurn then
    if myBoard[5] == curTurn then
        if myBoard[7] == curTurn then
            theGame = curTurn
            theID = D_BAR + 2
        end
    end
end
end
return theGame, theID
end

```

该函数首先检测棋盘所有的位置是否都放置了棋子，如果是则有可能是和局，设定 theWinner 值为 0。

然后检测所有可能的胜利的棋盘位置（总共 8 种），看当前玩家是否有 3 个棋子连成一线，如图 12.2 所示。如果 3 个同样的棋子连成一线，设定 theGame 变量为 curPlayer 变量的值，theID 设定为对应的胜利提示线 ID。最后返回这两个变量。

### 在 C 语言中处理 WinCheck 函数

在井字棋游戏中，把某些函数写成 LuaGlue 函数是一种很好的方式，可以在一些情况下用不同的方式实现函数。LuaGlue 函数的代码在另一个包含初始化 LuaGlue 函数的脚本



图 12.2 O 玩家获胜，有 3 个棋子在斜线方向连成一线，在棋子的上面显示连接线表明胜利

中。该函数在程序初始化时被调用，方便以后的扩展。

井字棋游戏的 LuaGlue 函数 WinCheck() 利用了含有 9 个整数的数组来表示棋盘，和我们在 Lua 中的做法相同。索引 0、1、2 是最上排的格子；3、4、5 是中间的格子；6、7、8 是最下排的格子。0 值表示“可用”或者是空的，其他值表示玩家 ID。有了这个数据结构后，可以定义所有的获胜条件的组合，如下所示：

```
const int winningPositions[8][3] =  
{  
    {0, 1, 2},  
    {3, 4, 5},  
    {6, 7, 8},  
    {0, 3, 6},  
    {1, 4, 7},  
    {2, 5, 8},  
    {0, 4, 8},  
    {2, 4, 6}  
};
```

棋盘和获胜条件设定好后，我们可以开始编写 LuaGlue 函数 WinCheck()。如果没有获胜并且棋盘还有空位置，函数返回 -1。如果返回值是 0 的话，说明是和局，没有获胜玩家



也没有空位置。如果有胜利玩家，返回胜利玩家的棋盘值。代码如下：

```
extern "C" int TTT_WinCheck(lua_State *L)
{
    cLua *lua = g_pBase->GetLua();
    int winner = -1;
    int argNum = 1;
    int board[9];
    for(int i=0; i<9; i++)
    {
        board[i] = (int) lua->GetNumberArgument(argNum++, 0);
    }
}
```

首先取得 cLua 对象，并且获取棋盘状态，将传入的前 9 个参数设定到内部 board 数组。

```
for(i=0; i<9; i++)
{
    if(board[winningPositions[i][0]]==board[winningPositions[i][1]] &&
        board[winningPositions[i][0]] == board[winningPositions
            [i][2]])
    {
        winner = board[winningPositions[i][0]];
    }
}
if(winner == -1)
{
    //didn't find a winner, check for draw
    bool bDraw = true;
    for(i=0; i<9; i++)
    {
        if(board[i])
            bDraw = false;
    }
    if(bDraw)
        winner = 0;
}
```

检测所有的获胜条件组合，看看是否有玩家满足。如果满足条件，则返回该玩家 ID。如果没有，则检测棋盘是否还有空位置，如果有空位置（0 表示空位）则游戏继续。确定了所有的返回值后，最后返回这些值并退出函数，代码如下：

```
lua->PushNumber(winner);
return 1;
}
```

在游戏开发的过程中，可以验证什么情况下使用 Lua，什么时候使用 C++ 的 LuaGlue 函数。在 WinCheck() 函数这个例子中，两种方式都适用。虽然使用 C++ 运行速度更快，但是在井字棋这样的回合制游戏中性能不是问题。可以从这个角度思考一下，如果要动态

显示胜利提示线，你会选择 LuaGlue 函数（使用 C 语言）还是 Lua 脚本？

### 12.2.3 模拟游戏回合

调用 WinCheck() 函数后，如果没有获胜方（theWinner 值为 -1），就进入另一个玩家的回合。如果游戏和局，则显示对应的提示信息；如果玩家获胜，也同样显示适当的信息。

如果 theWinner 等于 -1（继续游戏），下一个鼠标输入会处理另一个玩家的移动。如果游戏结束，GUI\_MOUSE\_BUTTON\_UP 事件的判断条件会忽略之后的输入。我们可以通过空格键重置游戏，代码如下：

```
if eventCode == GUI_KEY_PRESS then
    if id == 32 then --space bar
        if theWinner == -1 then
            InitGame()
        end
    end
end
end
```

在这个例子中，我们接触了回合制游戏的游戏主循环，并通过玩家的输入来改变回合。接下来看一看《Take Away》游戏，学习如何处理即时游戏的主逻辑。

## 12.3 《Take Away》游戏的实现原理

《Take Away》是即时游戏，不管玩家是否有操作，游戏都会继续。我们利用计时器和 StartTimer() 函数控制游戏流程。StartTimer() 函数在 GUI\_InGame.lua 函数中被调用，会生成 GUI\_TIMER\_EXPIRED 事件。在后面的章节中，我们会学习游戏的主循环。

### 12.3.1 InGame

尽管看上去很简洁，但 GUI\_InGame.lua 是游戏中最复杂的脚本。它非常依赖于 Lua-Support.lua 中定义的函数。为了更好地理解《Take Away》游戏的机制，我们会说明在一个典型的游戏进程中，游戏逻辑的每一个处理步骤。有了这个想法，我们可以忽略其他代码直接来看事件处理函数，在这里找到程序运行的方式。

GUI\_InGame.lua 脚本第一次执行时，我们创建一个表来初始化游戏，参考下面的脚本：

```
world = {}
```

这个  $40 \times 30$  的表将  $800 \times 600$  像素的屏幕分成若干  $20 \times 20$  像素的正方形。在显示游戏的虚拟边界时会使用该表。同时, 注意下面脚本中的事件处理函数。

第11章讨论过, 在检测了子界面后, GUI\_REENTER\_INTERFACE/GUI\_ENTER\_INTERFACE 事件调用的第一个函数是 InitialSetup(), 参考代码清单 12.5。

代码清单 12.5 GUI\_REENTER\_INTERFACE/GUI\_ENTER\_INTERFACE 函数

```
function InitialSetup()
    --Initial player constants
    myRotation = 1 --Player's rotation (#)
    myX = 390      --Player's x-coordinate (#)
    myY = 290      --Player's y-coordinate (#)
    myXThrust = 0  --Player's thrust along the x-axis (#)
    myYThrust = 0  --Player's thrust along the y-axis (#)
    alive = 'yes'  --Player's life status ('yes' or 'no')
    --Initial limits
    respawnInterval = 20 --Number respawnCounter must reach to respawn
    player (#)
    --Initial setting of counters
    respawnCounter = 0 --Player's death period (#)
    score = 0         --Player's score (#)
    timeCounter = 0   --Passage of time (#)
    targetDoneCounter = 0 --Targets stolen by enemies (#)
    --Preferred game speed
    refreshRate = .1 --Seconds between timer expirations (#)
    --Initial GUI setup
    ItemCommand(GUI_INGAME + 201, "SetString", GetText(GUI_INGAME + 201))
    ItemCommand(GUI_INGAME + 202, "SetString", GetText(GUI_INGAME + 202))
    ItemCommand(GUI_INGAME + 203, "SetString", GetText(GUI_INGAME + 203))
    ItemCommand(GUI_INGAME + 204, "SetString", "0")
    ItemCommand(GUI_INGAME + 205, "SetString", "0")
    ItemCommand(GUI_INGAME + 206, "SetString", "0")
    ClearGUI(GUI_ESCAPE)
    gEscapeOn = 0
    masterCellID = 100000 --Constant used for the IDs of the world's
    boundaries
end
```

该函数中所有的控件都需要在每次游戏重新开始时重置为原始值。标记了“初始化玩家常量”的变量是关于玩家飞船的。“上限”值是发送事件的计时器（这里的事件是重新生成玩家飞船）。

“计数器”是在游戏过程中随着不同事件的发生而增长的值。例如, timeCounter 和 respawnCounter 是基于时间的, 而其他的如 score 和 targetDoneCounter 是根据游戏中发生的特定事件而增长。“选定游戏速度”表示游戏计时器更新频率, 以秒为单位。因为大部分游戏功能和 GUI\_TIMER\_EXPIRED 事件相关, 所以 refreshRate 值是游戏流畅运行的关键。最

后,“GUI 初始设定”部分的处理是恢复 TextField 为默认值。

回过头来看看时间处理函数的处理顺序, MakeWorld() 函数的调用紧接着 InitialSetUp() 函数。Makeworld() 函数使用 world 表和 DrawCell() 函数绘制《Take Away》游戏的粉色边框,代码如下:

```
function MakeWorld()
    for x = 1,40 do
        for y = 1,30 do
            if world[y][x] == 1 then
                DrawCell(x,y)
            end
        end
    end
end
function DrawCell(x,y)
    --x is the x-coordinate
    --y is the y-coordinate
    CreateItem(masterCellID, 'Sprite', 'box1.bmp')
    SetItemPosition(masterCellID, (x-1) * 20, (y-1) * 20, 20, 20)
    masterCellID = masterCellID + 1
end
```

绘制完边框后, InGame 界面调用 EncmyInit() 函数创建若干敌舰。不同种类敌舰的数据保存在 myEnemies 表中。这张表中集中保存了所有敌舰状态的数据。该函数是界面使用的若干脚本之一, 保存在 LuaSupport.lua 文件中, 代码如下:

```
function EnemyInit()
    enemyCount = 5 --Number of enemies in the game
    myEnemies = {} --Creates myEnemies table
    --Creates table, one entry per potential enemy
    for indx = 1,enemyCount do
        --Creates a table to hold the data for each enemy
        myEnemies[indx] = {}
        --Now initialize the enemy
        EnemyRespawn(indx)
    end
end
```

和 MakeWorld() 函数一样, EnemyInit() 函数在 for 循环中调用函数。其中 EnemyRespawn(indx) 函数根据基本数据生成 myEnemies 表。创建完成后, myEnemies 表是整个游戏脚本中最重要的部分。为了继续填充这个表, 有必要来看一下 EnemyRespawn(indx) 函数, 它保存在 LuaSupport.lua 文件中。为了方便理解, 我们分段说明, 首先是“初始化”, 如下所示:

```

function EnemyRespawn(indx)
--Fills/refills myEnemies table according to indx
--indx is the myEnemies table index assigned to the enemy
--Initial values
myEnemies[indx].XTHRUST = 0 --Thrust along the x-axis (#)
myEnemies[indx].YTHRUST = 0 --Thrust along the y-axis (#)
myEnemies[indx].ROT = math.random(1,8) --Rotation of enemy ship (#)
myEnemies[indx].ID = GUI_RUNTIME_SPRITES + indx + 100 --Starts GUI
identification at 101 (#)
myEnemies[indx].E_TOW = 'no' --Towing flag (target ID # or 'no')
myEnemies[indx].FIRE = 0 --Projectile firing time interval (#)

```

上面列出的值用来定义敌舰的不同参数。XTHRUST、YTHRUST 和 ROT 用来处理敌舰的加速度和方向，ID 对应飞船的 GUI 标识。E\_TOW 表示敌人当前是否拖着补给箱。如果敌舰没有拖着任何补给箱，那么该值表示为“no”，否则设定为补给箱的 GUI ID。最后，FIRE 值表示敌舰 AI 在 EnemyFacing (indx) 函数中等待开火的时间。该值和 enemyFireInterval 比较的结果决定适当的开火间隔。

接下来的脚本是确定场景边界，敌舰会从边界出现。使用 Lua 的 math.random 函数，设定 entrySide 的值为 1~4 之间随机生成的整数。然后将该变量的值作为屏幕的四边，设定敌舰的初始 x 和 y 坐标，在对应位置创建敌舰。这样飞船就会从可见区域外飞入屏幕中的不同位置。为了避免冗余，这里只列出一部分代码，如下所示：

```

entrySide = math.random(1,4)
--if entrySide == 1 then --Left
-- myEnemies[indx].EX = math.random(-40,-20) --X-coordinate (#)
-- myEnemies[indx].EY = math.random(-40,620) --Y-coordinate (#)

```

在 EnemyRespawn (indx) 函数的第二部分，我们定义了敌舰的速度、灵活性和射击能力。这些属性会根据游戏的进行逐渐调整，即根据游戏计时器超时后 timeCounter 增长而变化。默认的 refreshRate 是 0.1s，需要注意 timeCounter 表示在这个频率下计时器超时的次数。举例来说，如果 timeCounter 值为 100 的话，在当前设定下表示游戏时间经过了 10s，如下所示：

```

if (timeCounter >= 0) and (timeCounter < 100) then
myEnemies[indx].REACT = 5 --Reaction time interval (#)
myEnemies[indx].MAX = 5 --Maximum thrust (#)
enemyFireInterval = 9
elseif (timeCounter >= 100) and (timeCounter < 200) then
myEnemies[indx].REACT = 4 --Reaction time interval (#)
myEnemies[indx].MAX = 6 --Maximum thrust (#)
enemyFireInterval = 8

```

当 EnemyRespawn (indx) 函数被调用来重新创建被“摧毁”的敌舰时, 会根据当前游戏的经过时间来确定飞船适当的 REACT 和 MAX 参数值。enemyFireInterval 值会影响游戏中所有的敌舰。REACT 值表示在 AI 处理时, 需要等待重生的超时次数。越小的值表示重生的等待间隔越短。MAX 值是敌舰的最大推进力, 越大的值表示飞行的速度越快。

函数的最后一步是确定敌舰的 TYPE 值 (参考下面的例子)。该参数决定了飞船的 AI 类型。在我们这个简单的游戏中, 总共有 4 种 AI 类型, 每种都是独特的设计。根据随机决定的 TYPE 值在适当位置创建飞船的 sprite 对象, 并使用与 EX 和 EY 类似的方法设定位置。这里我们可以发现 TYPE 为 3 的飞船有速度上的优势。每种飞船的特点会在后面 EnemyFacing (indx) 函数中详细说明。

```
myEnemies[indx].TYPE = math.random(1,4) --AI type (1,2,3, or 4)
if myEnemies[indx].TYPE == 1 then
    CreateItem(myEnemies[indx].ID, "Sprite", "e1_ship1.bmp")
elseif myEnemies[indx].TYPE == 2 then
    CreateItem(myEnemies[indx].ID, "Sprite", "e2_ship1.bmp")
elseif myEnemies[indx].TYPE == 3 then
    myEnemies[indx].MAX = 10
    CreateItem(myEnemies[indx].ID, "Sprite", "e3_ship1.bmp")
elseif myEnemies[indx].TYPE == 4 then
    CreateItem(myEnemies[indx].ID, "Sprite", "e4_ship1.bmp")
end
SetItemPosition(myEnemies[indx].ID, myEnemies[indx].EX,
myEnemies[indx].EY, 20, 20)
end
```

事件处理函数还调用了 TargetInit()。该函数的作用和 EnemyInit() 类似, 都用来创建保存特定对象数据的表。在游戏中, TargetInit() 函数是创建补给箱数据的。补给箱是普通的 sprite 对象分散在屏幕中, 一般是固定的, 可以被敌舰拖走。所有的参数保存在 myTargets 表中, 每个补给箱放置在适当的位置 (参考脚本 LuaSupport.lua), 如代码清单 12.6 所示。

代码清单 12.6 TargetInit() 函数

```
function TargetInit()
    targetCount = 8 --Number of targets in the game
    myTargets = {} --Creates myTargets table
    startID = GUI_RUNTIME_SPRITES + 200 --Starts GUI identification at
    200 (#)
    startX = 360
    startY = 260
    --Creates/fills table with initial values
    for indx = 1, targetCount + 1 do
        myTargets[indx] = {}
        myTargets[indx].T_ID = startID
        startID = startID + 1
    end
end
```

```

myTargets[indx].T_TOW = "no" --Towing flag (enemy ID #, "no", or
"done")
myTargets[indx].T_X = startX --x-coordinate (#)
startX = startX + 30
if startX > 420 then
    startX = 360
end
myTargets[indx].T_Y = startY --y-coordinate (#)
if indx > 2 then
    if indx < 6 then
        startY = 290
    else
        startY = 320
    end
end
end
end
--now delete the middle target, so the player can spawn in
table.remove(myTargets, 5)
--Creates targets (as sprites) and places them
for indx = 1,targetCount do
    CreateItem(myTargets[indx].T_ID, "Sprite", "box2.jpg")
    SetItemPosition(myTargets[indx].T_ID, myTargets[indx].T_X,
myTargets[indx].T_Y, 20, 20)
end
end
end

```

在这个结构中，根据 T\_ID、T\_TOW、T\_X、T\_Y 参数创建了 9 个补给箱（脚本中表示为 targets）。在敌舰和这里的所有参数中，只有 T\_TOW 是变量，补给箱的 T\_TOW 表示拖箱子的敌舰 GUI ID 或者箱子的拖动状态（no 或者 done）。补给箱 9 个一组出现在屏幕中间，不过只有 8 个是有效的，定义在 targetCount 变量中，因此，在游戏开始时，要使用 table.remove 删除最中间的箱子，剩下其余 8 个。

InGame 界面调用的最后一个函数是 ProjectileInit()。和其他初始化函数相比，它有一些不同之处。pCount 值表示游戏中同时可以显示的子弹数。因为设定了上线，所以省去了保存无限子弹的数据的麻烦。pIndx 是每个子弹对应的索引。当 pIndx 到达最大值 pCount 时，重设为 0，所以 pIndx 是循环的。有必要保存 pCount 和 pIndx 值，因为游戏中子弹数量是个常量，但索引值是循环的。playerProjectiles 计数器保存了玩家飞船一局中发射的子弹数。除了这些参数，myProjectiles 表中的 PROJ\_SHIP 参数是新的，该值保存发射子弹的玩家飞船或者敌舰索引。所有的部分包含在 FireProjectile (ship, xThrust, yThrust) 函数中，后面会详细说明。

### 12.3.2 使用计时器

初始化之后，InGame 界面的时间处理函数会调用 StartTimer() 开始游戏。计时器是

《Take Away》游戏的核心部分，因为它控制着游戏的主循环。每次超时，计时器会刷新屏幕并调用一些重要的函数（在 C++ 中，一般通过系统计时器或者根据屏幕刷新频率进行该处理）。

默认 refreshRate 为 0.1s，计时器会在瞬间超时，可以让玩家有非常流畅的游戏体验。有了这个概念，我们来看看 GUI\_InGame.lua 脚本的 GUI\_TIMER\_EXPIRED 第一部分，如下所示：

```
if eventCode == GUI_TIMER_EXPIRED then
    if gEscapeOn == 0 then
        if alive == "yes" then
            --Refreshes the player's ship image
            DrawShip(myRotation, myXThrust, myYThrust)
        else
            --Respawns the player after a certain period
            respawnCounter = respawnCounter + 1
            if respawnCounter == respawnInterval then
                myRotation = 1
                myXThrust = 0
                myYThrust = 0
                myX = 390
                myY = 290
                respawnCounter = 0
                alive = "yes"
            end
        end
    end
end
```

每次 GUI\_TIMER\_EXPIRED 事件发生时，我们都检测玩家飞船是否“活着”。如果 alive 值为 yes，调用函数 DrawShip (myRot, x, y)。该函数根据玩家的方向和推进力参数刷新玩家飞船的图像，本章的后面部分会详细说明。如果 alive 值为 no，respawnCounter 计数器增加 1。当 respawnCounter 值等于 respawnInterval 值（默认为 20）时，重置玩家飞船的参数。然后，玩家飞船在屏幕中间重新生成，alive 设定为 yes。

事件处理函数的第二部分增加 timeCounter 变量的值，如下所示。之前提到过，该值和时间相关的事件有关，如在游戏中生成的速度越来越快的敌舰。

```
--Directs each enemy based on its own REACT setting
timeCounter = timeCounter + 1
for indx = 1, enemyCount do
    if (timeCounter/myEnemies[indx].REACT) ==
        (math.floor(timeCounter/myEnemies[indx].REACT)) then
        EnemyFacing(indx)
    end
end
```



在该事件中, timeCounter 除以 REACT 值用来决定敌舰反应时间, 整除时, 调用 EnemyFacing (indx) 让敌舰开始朝目标移动。剩下部分是各种函数调用和重置计时器, 如下所示:

```
--Updates entities
DrawProjectile()
DrawEnemyShip()
DrawTargets()
--Checks for various collisions
EnemyHitCheck()
EnemyTowCheck()
end
--Resets the timer
StartTimer(refreshRate)
end
end --Escape Sub-GUI
end
```

所有的函数会在稍后详细说明, 为了避免出现疑惑, 这里简单说明一下它们的处理。DrawProjectile()、DrawEnemyShip() 和 DrawTarget() 函数会根据敌舰的位置更新对应的 sprites 并检测是否碰到场景边界。EnemyHitCheck() 检测玩家炮弹和敌舰的碰撞, 如有必要则删除敌舰。EnemyTowCheck() 用来检测敌舰和补给箱的碰撞, 如有必要则创建拖拽关系。最后, 重置 GUI 计时器。

到目前为止, 我们已经说明了《Take Away》游戏的主逻辑, 虽然还有更多的细节需要解释, 但我们已经有了即时游戏主逻辑的概念。

### 12.3.3 玩家操作

回到 GUI\_InGame.lua 脚本, 我们来看一看玩家在游戏操作。在 GUI\_REENTER\_INTERFACE/GUI\_ENTER\_INTERFACE 事件后, 会看到 GUI\_KEY\_PRESS 事件的脚本。这里的 ID 是键盘按键对应的 ASCII 编码, 该事件的第一部分是第 11 章说明过的 <Escape> 键功能。接下来可以看到下面的代码:

```
if id == gTurnLeftKey then --Default [
--Turns player's ship counterclockwise
myRotation = myRotation - 1
if myRotation < 1 then
myRotation = 8
end
end
```

gTurnLeftKey (在 Controls 界面中设定) 减少 myRotation 值, 以逆时针旋转玩家的飞

船。当 myRotation 为 0 时，再设定为 8，以保持始终指向基本方位（北、东北、东、东南等）。这些方向限定了表示玩家飞船的 8 张位图。改变 myRotation 的值可以影响 DrawShip (myRot, x, y) 函数绘制飞船的方式。该函数根据 gTurnRightKey 值顺时针旋转飞船，让玩家可以自由控制方向。gThrustKey 根据玩家飞船的方向调节 myXThrust 和 myYThrust 的值，范围为 -5 ~ 5。这个范围保证了游戏的挑战性和节奏感。此外，gShootKey 功能定义如下：

```
if id == gShootKey then --Default p
    --Fires a projectile from the player
    if alive == 'yes' then
        FireProjectile('player', myXThrust, myYThrust)
    end
end
end
```

当 gShootKey 值对应的键被按下时，该脚本检测玩家的飞船是否在屏幕上，并调用 FireProjectile (ship, xThrust, yThrust) 函数。结束了操作部分的脚本，接下来说明代码清单 12.7 的函数（在 LuaSupport.lua 文件中）。

代码清单 12.7 FireProjectile (ship, xThrust, yThrust) 函数

```
function FireProjectile(ship, xThrust, yThrust)
    --ship can either be 'player' or the myEnemies index of the enemy
    --xThrust is the projectile's thrust along the x-axis
    --yThrust is the projectile's thrust along the y-axis
    pIndx = pIndx + 1
    myProjectiles[pIndx].PROJ_ID = GUI_RUNTIME_SPRITES + pIndx + 299 -
        Starts IDs at 300 (not including offset)
    if ship == 'player' then
        --Player's projectile
        playerProjectiles = playerProjectiles + 1
        myProjectiles[pIndx].PROJ_SHIP = 'player'
        myProjectiles[pIndx].PROJ_X = myX
        myProjectiles[pIndx].PROJ_Y = myY
        Rot = myRotation
        CreateItem(myProjectiles[pIndx].PROJ_ID, "Sprite", "box2.jpg")
    else
        --Enemy's projectile
        myProjectiles[pIndx].PROJ_SHIP = ship
        myProjectiles[pIndx].PROJ_X = myEnemies[ship].EX +
        myEnemies[ship].XTHRUST
        myProjectiles[pIndx].PROJ_Y = myEnemies[ship].EY +
        myEnemies[ship].YTHRUST
        Rot = myEnemies[ship].ROT
        CreateItem(myProjectiles[pIndx].PROJ_ID, "Sprite", "box3.jpg")
    end
end
```

FireProjectile (ship, xThrust, yThrust) 函数创建子弹的 sprite 图像，并设定在屏幕中，根据飞船参数计算速度和方向。后面的函数是关于子弹的碰撞检测，这里只说明一下创建

过程。在函数的开始部分，增加 pIndx 的值为 myProjectiles 表生成新的索引。然后，根据飞船的值确定发射子弹的飞船。之后会将对应飞船的参数添加到 myProjectiles 表中。方向相关值的处理如下所示：

```
--Set the projectile's position and thrust
if Rot == 1 then --Up
    myProjectiles[pIndx].PROJ_X = myProjectiles[pIndx].PROJ_X + 8
    myProjectiles[pIndx].PROJ_Y = myProjectiles[pIndx].PROJ_Y
    myProjectiles[pIndx].PROJ_XTH = xThrust
    myProjectiles[pIndx].PROJ_YTH = yThrust - 10
end
```

最后，在游戏场景相关飞船的位置绘制各自的图像。只要游戏在进行，界面会被更新以显示玩家发射的子弹数。当 pIndx 达到 pCount 值时，会重置为 0，如下所示：

```
--Update the display
SetItemPosition(myProjectiles[pIndx].PROJ_ID,
myProjectiles[pIndx].PROJ_XTH, myProjectiles[pIndx].PROJ_YTH, 4, 4)
if targetDoneCounter ~= targetCount then
    ItemCommand(GUI_INGAME + 204, "SetString",
    CommaFormatBigInteger(playerProjectiles))
end
if pIndx == pCount then
    pIndx = 0
end
end
```

尽管每当子弹发射的时候都会进行上面的处理，但是循环的特性让处理器可以更加集中地处理与游戏玩法相关的问题。当 pIndx 达到 50 时，通过“重写”myProjectiles 表的值，在不破坏有限子弹系统的前提下，我们可以让游戏看起来有无限的子弹。

### 12.3.4 子弹运动

我们已经生成了子弹，还必须逐帧控制它的运动。重新来看 GUI\_InGame.lua 脚本的 GUI\_TIMER\_EXPIRED 部分，会发现 DrawProjectile() 函数在每次事件中都被调用了。该函数（在 LuaSupport.lua 文件中）一直在刷新显示所有的子弹图片，是一个 for 循环处理，如代码清单 12.8 所示。

代码清单 12.8 DrawProjectile() 函数

```
function DrawProjectile()
    for indx = 1, pCount do
        if myProjectiles[indx].PROJ_ID ~= nil then
            DeleteItem(myProjectiles[indx].PROJ_ID)
            if myProjectiles[indx].PROJ_SHIP == "player" then
```

```
--Player's projectile
CreateItem(myProjectiles[indx].PROJ_ID, "Sprite",
"box2.jpg")
else
--Enemy's projectile
CreateItem(myProjectiles[indx].PROJ_ID, "Sprite",
"box3.jpg")
end
myProjectiles[indx].PROJ_X = myProjectiles[indx].PROJ_X +
myProjectiles[indx].PROJ_XTH
myProjectiles[indx].PROJ_Y = myProjectiles[indx].PROJ_Y +
myProjectiles[indx].PROJ_YTH
SetItemPosition(myProjectiles[indx].PROJ_ID,
myProjectiles[indx].PROJ_X, myProjectiles[indx].PROJ_Y, 4, 4)
```

DrawProjectile() 函数遍历 myProjectiles 表, 并检测游戏中的所有子弹。它删除所有的子弹再更新图像位置来表示子弹的位移, 如图 12.3 所示。该处理出现在所有的绘图函数中。为什么在每一帧需要不停创建并删除这些图像而不是改变它们的位置呢? 答案在游戏载入过程中。

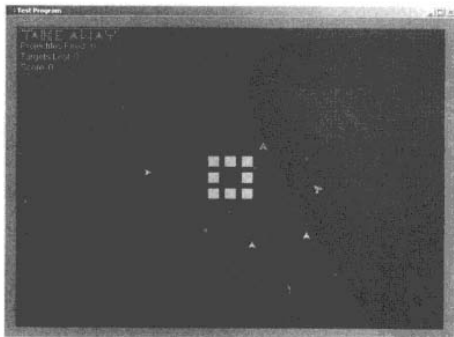


图 12.3 在《Take Away》游戏实时画面中, 可以看到敌舰、子弹和储物箱

持续的创建和删除这些 sprite 可以在游戏中, 根据表中的 ID 参数简单地激活实体, 如果载入的游戏数据中子弹的 PROJ\_ID 值为 nil, 就不会被绘制。如果是一个合法的数字, 就会自动调用 DrawProjectile() 函数创建并显示该子弹。

继续函数的最后一部分，这里检测了游戏边界和子弹的碰撞。当子弹碰到了边界时，删除子弹的 sprite 并清除它对应的数据，如下所示：

```
--Deletes projectiles when they pass the world's boundaries
if (myProjectiles[indx].PROJ_X > 780) or
   (myProjectiles[indx].PROJ_X < 20) or
   (myProjectiles[indx].PROJ_Y > 580) or
   (myProjectiles[indx].PROJ_Y < 20) then
    DeleteItem(myProjectiles[indx].PROJ_ID)
    myProjectiles[indx].PROJ_X = nil
    myProjectiles[indx].PROJ_Y = nil
    myProjectiles[indx].PROJ_XTH = nil
    myProjectiles[indx].PROJ_YTH = nil
    myProjectiles[indx].PROJ_ID = nil
    myProjectiles[indx].PROJ_SHIP = nil
end
end
end
end
end
```

这里只做简单的碰撞检测，通过计算判断子弹的任意部分是否穿过了边界。根据这里的判断，可以删除对应的子弹。这种碰撞检测对于子弹来说是有效的，因为它只需要删除碰撞体。

### 12.3.5 飞船移动

《Take Away》游戏中的飞船，会以不同方式碰到障碍物；碰到墙时会反弹。在说明决定正确反弹方向的 CollisionCheck (ship, x, y) 函数之前，有必要先看一下它的一个组件——GetTravelDirection (xThrust, yThrust) 函数（在 LuaSupport.lua 文件中），如代码清单 12.9 所示。

代码清单 12.9 GetTravelDirection (xThrust, yThrust) 函数

```
function GetTravelDirection(xThrust, yThrust)
--xThrust is a ship's thrust along the x-axis
--yThrust is a ship's thrust along the y-axis
direction = 1
if math.abs(xThrust) >= math.abs(yThrust) then
--Left/right
lrPercent = (math.abs(yThrust)/math.abs(xThrust)) * 100
if lrPercent < 30 then
if xThrust < 0 then
direction = 7 --Left
else
direction = 3 --Right
end
end
```

```
    else
        if xThrust < 0 then
            if yThrust < 0 then
                direction = 8 --Up/left
            else
                direction = 6 --Down/left
            end
        else
            if yThrust < 0 then
                direction = 2 --Up/right
            else
                direction = 4 --Down/right
            end
        end
    end
end
else
```

GetTravelDirection (xThrust, yThrust) 函数使用了一系列 if-then 语句, 并根据 xThrust 和 yThrust 的值, 判断飞船的当前位置。计算 xThrust 和 yThrust 的比率 lrPercent 或者 udPercent (未显示), 根据这个百分比来推测飞船的方向。direction 值和飞船的 8 个方向的位图相关, 函数最后返回 direction 值。

确定飞船的移动方向后, 我们可以来处理它和场景边界的碰撞检测了。计算碰撞时飞船的前进方向可以确定适当的反弹轨迹。全部的处理在 LuaSupport.lua 脚本中的 CollisionCheck (ship, x, y) 函数中。我们分段说明, 参考代码清单 12.10。

代码清单 12.10 CollisionCheck (ship, x, y) 函数

```
function CollisionCheck(ship, x, y)
--ship can either be 'player' or the myEnemies index of the enemy
--x is the ship's x-coordinate
--y is the ship's y-coordinate
collision = NO
if ship == 'player' then
    --Player
    if (y <= 20) then
        myY = 25
        collision = HORIZONTAL
    elseif (y >= 560) then
        myY = 555
        collision = HORIZONTAL
    elseif (x <= 20) then
        myX = 25
        collision = VERTICAL
    elseif (x >= 760) then
        myX = 755
        collision = VERTICAL
    end
end
```

```

travelDir = GetTravelDirection (myXThrust, myYThrust)
--Set local variables to global values
XThrust = myXThrust
YThrust = myYThrust
else

```

首先，函数根据 ship 值（等于 player 或者 myEnemies 表的索引值）判断是玩家飞船还是敌舰。然后，比较飞船位置和场景边界并且确定碰撞的类型，HORIZONTAL 表示碰到了左右的墙壁（屏幕的上、下边界），VERTICAL 表示碰到了上下的墙壁（屏幕的左、右边界）。

需要注意的是，敌舰和玩家飞船的活动边界并不相同。在脚本中你会发现敌舰的边界多了 60 以上的像素空间（有 40 像素在屏幕外）。这些多余的空间可以根据游戏的需要让敌舰消失在屏幕外，又可以很快地让它们返回游戏。这种方法让游戏环境有了逻辑顺序并且让游戏流程保持在一个最佳状态。

函数还更新飞船的坐标，在弹出墙壁前和墙壁分离。如果 gThrustKey 持续按下，这个推力可以让飞船逐渐向边界前进。这段代码最后部分设定 travelDir、Xthrust 和 Ythrust 变量的值与飞船数据一致。

函数的第二部分根据 collision 和 travelDir 的值改变 Xthrust 和 Ythrust 的值，代码如下：

```

if collision > NO then
--Set thrusts based on travel direction and collision
  if collision == HORIZONTAL then
    if travelDir == 1 then
      YThrust = YThrust * -1
    end

```

该部分循环处理 travelDir 的 8 种可能的值以及 VERTICAL 碰撞。该方法全面负责《Take Away》游戏的碰撞并同时保持逻辑上的简洁性，代码如下所示：

```

--Set global variables equal to local values
if ship == "player" then
  myXThrust = XThrust
  myYThrust = YThrust
else

```

最后，CollisionCheck (ship, x, y) 函数更新全局变量的值，如果是玩家飞船，则更新 myXThrust 和 myYThrust 变量，如果是敌舰，则更新 myEnemies 表中的对应属性。

以上就是边界碰撞检测，接下来可以看看 3 个绘图函数：DrawShip (myRot, x, y)、DrawEnemyShip() 和 DrawTargets()。它们都使用了和 DrawProjectile() 函数相同的结构，这里我们只说明它们不同的部分。

### 12.3.6 绘制活动的物体

DrawShip (myRot, x, y) 函数和 DrawProjectile() 函数的不同之处在于, 它调用了 CollisionCheck (ship, x, y) 函数和 CasualtyCheck() 函数 (另一个碰撞检测函数, 稍后会说明)。在更新函数中调用这些函数, 可以保证所有重要的碰撞检测可以在 0.1s 的时间内独立进行。

因为 DrawEnemyShip() 函数 (在 LuaSupport.lua 文件中) 需要检测和操作 myEnemies 表, 所以看起来会比 DrawShip (myRot, x, y) 函数复杂一些, 不过大体结构还是一样。不同的部分是它包含在一个 for 循环内, 并根据 ROT 值处理 XTHRUST 和 YTHRUST 参数生成 x 和 y 的推力值, 如下所示:

```
--Determine enemy's new thrust
newThrust = math.random(0,2)
if myEnemies[indx].ROT == 1 then --Up
    myEnemies[indx].YTHRUST = myEnemies[indx].YTHRUST -
    newThrust
end
if myEnemies[indx].ROT == 2 then --Up/right
    myEnemies[indx].YTHRUST = myEnemies[indx].YTHRUST -
    newThrust
    myEnemies[indx].XTHRUST = myEnemies[indx].XTHRUST +
    newThrust
end
end
```

之后要确保推力值没有超过 MAX 上限。通过这种方式, 来保证游戏的难度和节奏。在这部分处理之后, 有如下代码:

```
--Checks for collisions and displays the new enemy ship
shipName = string.format("%s%d%s%d%s", "e",
myEnemies[indx].TYPE, "_ship", myEnemies[indx].ROT, ".bmp")
myEnemies[indx].EX = myEnemies[indx].EX +
myEnemies[indx].XTHRUST
myEnemies[indx].EY = myEnemies[indx].EY +
myEnemies[indx].YTHRUST
CollisionCheck(indx, myEnemies[indx].EX, myEnemies[indx].EY)
CreateItem(myEnemies[indx].ID, "Sprite", shipName)
SetItemPosition(myEnemies[indx].ID, myEnemies[indx].EX,
myEnemies[indx].EY, 20, 20)
end
end
end
```

最值得注意的是, 在函数中, 表系统的功能是相当强大的。通过在一张表中保存敌舰



所有的数据，我们可以构建位图文件名、检测碰撞，以及创建和删除飞船并设定它们的位置。通过表结构为不同实体做上述一系列的处理是很容易的事情。

DrawTargets() 函数的结构也同其他绘图函数一样。它包含两个大的 for 循环。根据 T\_TOW 参数判断了箱子的拖动状态后，函数会集中处理那些被敌舰拖动的补给箱（没有被拖动的箱子使用函数的最后几行代码重新绘制）。考虑了拖动箱子的敌舰方向，可以计算箱子的位置，如下所示：

```
--Position target behind enemy
if myEnemies[i].ROT == 1 then --Up
    myTargets[indx].T_X = myEnemies[i].EX
    myTargets[indx].T_Y = myEnemies[i].EY + 20
end
if myEnemies[i].ROT == 2 then --Up/right
    myTargets[indx].T_X = myEnemies[i].EX - 20
    myTargets[indx].T_Y = myEnemies[i].EY + 20
end
end
```

通过适当地调整 myTargets 值，创建补给箱的图像并设定位置，让它们总是显示在敌舰的后面。这里我们使用 DrawProjectile() 函数相同的结构，如下所示：

```
if (myTargets[indx].T_X > 800) or (myTargets[indx].T_X < -20) or
(myTargets[indx].T_Y > 800) or (myTargets[indx].T_Y < -20) then
    DeleteItem(myTargets[indx].T_ID)
    myTargets[indx].T_X = nil
    myTargets[indx].T_Y = nil
    myTargets[indx].T_ID = nil
    myTargets[indx].T_TOW = "cone"
    myEnemies[i].E_TOW = "no"
```

注意：这里的 T\_TOW 和 E\_TOW 参数需要分别设为 done 和 no，表示敌舰成功地“偷”走了补给箱并且现在可以拖动新的补给箱。

其他注意点会在 EnemyHitCheck()、EnemyTowCheck() 和 CasualtyCheck() 函数的说明中提到。函数的最后一部分是处理最后一个被“偷”走的补给箱，这种情况下结束游戏，代码如下：

```
--Update stolen target display and/or display game over string
targetDoneCounter = targetDoneCounter + 1
ItemCommand(GUI_INGAME + 205, "SetString",
CommaFormatBigInteger(targetDoneCounter))
if targetDoneCounter == targetCount then
    for indx = 1,pCount do
        if myProjectiles[indx].PROJ_ID ~= nil then
            DeleteItem(myProjectiles[indx].PROJ_ID)
        end
    end
    DeleteItem(GUI_RUNTIME_SPRITES + 100)
    RunGUI("GUI_EndGame.lua")
end
end
```

该部分增加 `targetDoneCounter` 计数器，它会和补给箱总数（`targetCount`）进行比较，如果相等，则表示所有的补给箱都被“偷”走了，这时清除所有的子弹和玩家的飞船，直到重新开始游戏。最后执行 `GUI_EndGame.lua` 脚本。

## 12.4 本章小结

我们完整地说明了通过 GUI 计时器控制的即时游戏《Take Away》的流程。本章还介绍了使用事件驱动系统的回合制游戏井字棋的流程。利用这里的方法，可以思考一下如何控制即时战略游戏或者更复杂的回合制游戏的流程。

有了这些知识，我们接下来要学习使用 Lua 开发由计算机控制的对手，也就是人工智能。在下一章中，还会继续以井字棋和《Take Away》游戏为例，并结合其他的例子来共同说明如何使用 Lua 控制由计算机控制的对手。



## 第 13 章

# 使用 Lua 定义和控制 AI

### 本章要点

- 智能的体现
- 21 点游戏
- 井字棋
- 《Take Away》
- 其他 AI 例子
- 有限状态机
- 路径寻找

在前一章中，我们开发了一个简单的桌面游戏。通过编写脚本我们可以控制飞船，在被敌人包围的游戏世界中穿行。怎样让这个游戏世界更有挑战性呢？在我们的游戏中，计算机控制的敌人尝试着冲击你，抢夺你的补给箱并且攻击你的飞船。控制这些敌人的就是 AI——通过程序代码开发的人工智能（AI）。

人工智能是指那些计算机表现出“思考”的情况。众所周知，计算机是不会思考的，至少现在还不能，但是计算机很擅长处理数据并且根据这些数据做出响应。如果响应正确，那么看起来就像计算机“思考”。

在本章中，我们会学习使用 Lua 开发 AI，不仅在《Take Away》游戏中，在其他例子中也会讨论，内容涵盖了路径寻找和有限状态机等主题。

### 13.1 智能的体现

人工智能以不同的形式存在于游戏中，因为游戏的种类很多。AI 可以用在不同场合，例如，在棋类游戏中用来确定下一步最佳移动，在扑克游戏中决定如何下注，21 点游戏中庄家是否发牌，在碎片散落的战场寻找最优路径等。

在本章中，我们会介绍一些人工智能的基本要点，并结合本书中的游戏例子说明它们是如何工作的。我们还会通过一些例子介绍路径寻找和卡牌游戏中的决策选择。学习了这些要点，你会更好地理解该如何使用 Lua 在游戏中控制计算机对手。这些基础知识可以帮助你为自己的游戏开发 AI 系统。

在游戏开发中需要清楚一个关键的概念，AI 不等同于智能，它只是智能的一种表现。对于玩家来说，AI 通常是提升游戏挑战性的机制（也可为玩家控制的复杂物体提供自动操作）。

当游戏开发者开发一个系统来控制敌人，为玩家提供一些挑战时，玩家对敌人的感知是最重要的。计算机控制的敌人不需要评估游戏中所有的风险来确定最佳的行动方案，它只需要让玩家觉得这些行为背后是有一定逻辑的，会让玩家感受到挑战并不断做出反应。

关于 AI 作弊——让计算机对手可以在游戏中获取比玩家更多的信息（如准确的行走轨迹或者知道迷宫中门和钥匙的所在地），支持者和反对者争论了很多年。我们对此的观点是这根本不重要：玩家的游戏体验才是核心。如果作弊可以提供更多的、有趣的、令人兴奋的对手而又不会影响游戏的平衡性，那就是可行的。毕竟这些都是关于智能的体现，而且你也希望你的敌人和你面临的挑战都是足够智能的，而且难度也可以接受，可能会有些挣扎，但是可以克服。

## 13.2 21 点游戏

卡牌游戏是很适合用 Lua 脚本开发的人工智能游戏类型之一。卡牌游戏处理的是随机的、有限的牌，并且还有清楚的规则（这手牌好于另外一手牌），Lua 很擅长这一类处理。

首先，来看一个简单的例子：21 点游戏。在 CD-ROM 的本章资源中，有一个小的 21 点游戏，可以和庄家玩牌。在学习 AI 之前，先来看看怎么设定牌。首先创建下列常量，这可以让后面的代码可读性更强。

```
--blackjack constants
SPADE = 1
DIAMOND = 2
CLUB = 3
HEART = 4

ACE = 1
JACK = 11
QUEEN = 12
KING = 13
```

然后，在 Lua 表中创建一副虚拟的牌，如下所示：

```
function CreateDeck()
    curDeckLocation = 1
    myDeck = {}
    cardNum = 1
    for suit = SPADE, HEART do
        for card = ACE, KING do
            myDeck[cardNum] = {}
            myDeck[cardNum].suit = suit
            myDeck[cardNum].card = card
            cardNum = cardNum + 1
        end
    end
end
```

该函数为每种可能的牌创建了实例（也可以创建多副牌）。下一步是洗牌，如下所示：

```
function ShuffleDeck()
    shuffleValue = 1000
    for indx = 1, shuffleValue do
        card1 = math.random(1,52)
        card2 = math.random(1,52)
        hold = {}
        hold = myDeck[card1]
        myDeck[card1] = myDeck[card2]
        myDeck[card2] = hold
    end
end
```

该函数循环 1000 次，每次从一副牌中随机抽取两张牌并交换位置，结果可以得到一副足够随机的牌，这样每次从上面拿到的牌都是随机分布的，如图 13.1 所示。

最后，初始化玩家和庄家的手牌，使用如下所示的函数：

```
function InitHands()
    theHand = {}
    theHand[DEALER] = {}
    theHand[HUMAN] = {}
end
```

该函数创建了二维表来保存玩家和计算机对手（即庄家）的牌，所有的牌都用 my-Deck 表的索引来表示。然后可以用下面的函数来绘制扑克牌：

```
function GetCard(player)
    if curDeckLocation > 52 then
        print("Out of cards! Start new game!")
    else
        table.insert(theHand[player], curDeckLocation)
        DrawHand(player) -- draws the card on the screen
        curDeckLocation = curDeckLocation + 1
    end
end
```

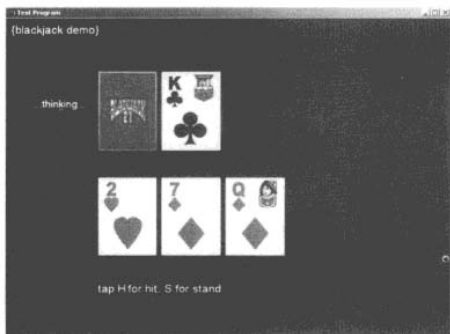


图 13.1 我们做的 21 点游戏，控制庄家和玩家对战

```

end
end

```

我们使用 `curDeckLocation` 变量来保存牌的位置，发牌时增加变量值。`DrawHand()` 函数在屏幕上绘制牌，函数很简单，可以自己查看代码。

有了这个机制，我们可以继续看看这两个更有趣的函数：评估手牌和让计算机庄家“思考”。在 21 点游戏中，手牌根据 A 的数量有不同的估值方式。这里使用代码清单 13.1 的函数得到当前手牌的估值。

代码清单 13.1 CheckHandValue 函数

```

function CheckHandValue(player)
    myValue = 0
    numAces = 0
    for indx = 1, table.getn(theHand[player]) do
        curCardValue = myDeck[theHand[player][indx]].card
        if curCardValue > 10 then
            curCardValue = 10
        end
        if curCardValue == ACE then
            --we have an ace
            numAces = numAces + 1
        else
            myValue = myValue + curCardValue
        end
    end
end

```

```
end
end
if numAces > 0 then
    --deal with one ace
    if numAces == 1 then
        if myValue + 11 > 21 then
            myValue = myValue + 1
        else
            myValue = myValue + 11
        end
    end
    --deal with two aces
    if numAces == 2 then
        if myValue + 12 > 21 then
            myValue = myValue + 2
        else
            myValue = myValue + 12
        end
    end
    --deal with three aces
    if numAces == 3 then
        if myValue + 13 > 21 then
            myValue = myValue + 3
        else
            myValue = myValue + 13
        end
    end
    --deal with four aces
    if numAces == 4 then
        if myValue + 14 > 21 then
            myValue = myValue + 4
        else
            myValue = myValue + 14
        end
    end
end
return myValue
end
```

在该函数中，我们遍历了玩家手中的牌，如果是“人头牌”（扑克中的 J、Q、K）就设为 10；如果是 A，就不计入总点数，单独统计有多少个 A。最后看看手牌总点数。函数判断了 A 的所有可能，虽然有些啰嗦，但是说明了该如何根据当前情况自动处理 A 牌的值。

在玩家游戏循环中按下 <H> 键（基于 GUI\_KEY\_PRESS 事件，在 GUI\_InGame.lua 文件中），使用该函数确定拿了另一张牌后是否“爆牌”（即超过 21 点）。

玩家拿好牌后，轮到庄家。在拉斯维加斯的玩法中，庄家一方的第一张牌是扣底牌，

牌点是需要统计的。玩家一旦按下 <S> 键后，庄家开始“思考”，处理过程如下面的函数所示：

```
function Think(player)
    result = THINKING
    if CheckHandValue(player) < HOUSE_STAND then
        GetCard(player)
    else
        result = DONE_THINKING
    end
    if CheckHandValue(player) > 21 then
        result = BUST
    end
    return result
end
```

这是一个非常简单的计算机 AI 的例子——庄家必须停牌的点数定义在常量 HOUSE\_STAND 中（设定为 17 点——在赌场，庄家一般在 17 点及以上点必须停牌）。庄家对比手牌点数做出选择，这时有 3 种结果：尚在思考、完成或者“爆牌”。

我们使用计时器函数来处理庄家“思考”的循环，代码如下：

```
if eventCode == GUI_TIMER_EXPIRED then
    progress = Think(DEALER)
    if progress == BUST then
        EnableObject(GUI_INGAME + 205, 0, 0)
        gameState = DEALER_DONE
        DrawHand(DEALER)
        ItemCommand(MESSAGE, "SetString", "Dealer busts... YOU WIN!!")
    elseif progress == THINKING then
        StartTimer(1.5)
    else
        --dealer is done...process results
        EnableObject(GUI_INGAME + 205, 0, 0)
        gameState = DEALER_DONE
        DrawHand(DEALER)
        if CheckHandValue(HUMAN) > CheckHandValue(DEALER) then
            theScore = tostring(CheckHandValue(HUMAN))
            ItemCommand(MESSAGE, "SetString",
                string.format("%s%s", "You win the hand with a ",
                    theScore, "1"))
        else
            theScore = tostring(CheckHandValue(DEALER))
            ItemCommand(MESSAGE, "SetString",
                string.format("%s%s", "Dealer takes the hand with a
                    score of ", theScore, "."))
        end
        EnableObject(GUI_INGAME + 202, 1, 1)
    end
end
```



```
end  
end
```

这段代码调用了 Think() 函数, 然后根据返回值报告结果。如果结果是“尚在思考”, 计时器开始并重新执行游戏循环。

在上面的例子中, 我们可以看到使用一个简单的 AI 就可以达到拉斯维加斯赌场的水平, 因为都遵循了同样的停牌、要牌规则。

### 13.3 井字棋

现在回到前一章提到的井字棋游戏, 看看该如何创建由计算机控制的对手。这次使用 C++ 来开发 LuaGlue 函数, 决定玩家的移动。

首先, 查看在双人对战的井字棋中要整合计算机对手需要做些什么。在只有玩家对战的游戏中, 我们使用下面的函数来获取棋子的位置:

```
thePos = GetBoardLocation(GetMousePosition())
```

该函数返回 1~9, 分别表示棋子的位置, -1 表示没有可以移动的位置。之后, MakeMove() 函数负责绘制棋子, 检测胜负和显示提示信息。在理想情况下, 计算机对手可以像之前的例子一样使用一个函数来确定移动。这样的话, 我们就可以直接在 MakeMove() 函数中调用, 保留尽可能多的代码。

LuaGlue 函数会是下面的样子:

```
thePos = GetMove(myBoard[1],myBoard[2],myBoard[3],myBoard[4],  
myBoard[5],myBoard[6],myBoard[7],myBoard[8],myBoard[9],OH)
```

在这个函数中, 我们传入棋盘和当前玩家(我们必须手动传入表, 因为 C 不能直接处理 Lua 表类型的参数), 如图 13.2 所示。

在井字棋的例子中, 计算机控制 O 棋子, 玩家控制 X 棋子。游戏初始化时, 随机决定先手。接着初始化脚本, 加入下列代码:

```
if eventCode == GUI_ENTER_INTERFACE then  
    InitGame()  
    if curTurn == OH then  
        StartTimer(1.5)  
    end  
end  
end
```



图 13.2 在前一章的井字棋游戏中加入了 LuaGLite 函数可以让我们和计算机对战

该脚本检测当前玩家是否为 OH 玩家，如果是则开始计时器事件。事件处理函数如代码清单 13.2 所示。

代码清单 13.2 处理计时器事件的脚本

```
if eventCode == GUI_TIMER_EXPIRED then
    MakeMove()
end

This is really all we need to change in our GUI_InGame.lua logic. Now the
revised MakeMove() function looks like this:
function MakeMove()

    if curTurn == EX then
        --X human player
        thePos = GetBoardLocation(GetMousePosition())
    else
        --O AI player
        thePos = GetMove(myBoard[1],myBoard[2],myBoard[3],myBoard[4],
            myBoard[5],myBoard[6],myBoard[7],myBoard[8],myBoard[9],OH)
    end
    if thePos == -1 then
        --turn on the space
        EnableObject(curTurn + thePos, 1, 1)
        --change text
        if curTurn == EX then
            ItemCommand(GUI_INGAME + 250, "SetString", "O's turn: AI
            thinking")
        end
    end
end
```

```

else
    ItemCommand(GUI_INGAME + 250, "SetString", "X's turn: left
        click to place")
end
--now update the table
myBoard[thePos] = curTurn
--check for win
theWinner = -1
theWinner, slashID = WinCheck()
if theWinner == -1 then
    --no winner
    --now swap turns
    if curTurn == OH then
        curTurn = EX
    else
        curTurn = OH
    end
elseif theWinner == 0 then
    --cat's game
    EnableObject(GUI_INGAME + 210, 1, 1)
    EnableObject(GUI_INGAME + 240, 1, 1)
    EnableObject(GUI_INGAME + 250, 0, 0)
else
    --we have a winner
    EnableObject(GUI_INGAME + 250, 0, 0)
    EnableObject(GUI_INGAME + 240, 1, 1)
    EnableObject(slashID, 1, 1)
    if theWinner == EX then
        EnableObject(GUI_INGAME + 220, 1, 1)
    else
        EnableObject(GUI_INGAME + 230, 1, 1)
    end
end
end
end
if (curTurn == OH) and (theWinner == -1) then
    StartTimer(1.5)
end
end
end

```

前面的函数和双人游戏版本中使用的函数基本相同，除了开头部分的检测。如果玩家是由人控制的（EX），就调用 GetBoardLocation() 函数；如果是由计算机控制的，就调用 GetMove() 函数。

在函数的最后，看当前玩家（我们已经轮换）是否是计算机玩家，并且游戏是否在进行。如果是的话，调用计时器函数（这样做是为了让人觉得计算机花了些时间来“思考”，显得更自然些），触发下一个 MakeMove() 函数。

## LuaGlue 函数 GetMove( )

AI 移动寻找有很多方式，有简单的也有复杂的。由于井字棋是个简单的游戏，所以基于规则的简单粗放的方式是最适合的。这个实现也能让读者很容易地改进函数性能。下面来看看用 C 语言编写的函数：

```
extern "C" int TTT_GetMove(lua_State *L)
{
    cLua *lua = g_pBase->GetLua();
    int move = -1;
    int argNum = 1;
    int board[9];
    for(int i=0; i<9; i++)
    {
        board[i] = (int) lua->GetNumberArgument(argNum++, 0);
    }
    int sidetomove = (int) lua->GetNumberArgument(argNum++, 0);
```

前面的函数我们创建了棋盘，通过取得的 9 个数值计算出结果。这个函数需要知道是谁在移动，这样我们就可以为他寻找最佳移动。然后，我们需要生成一个包含合法移动的列表，如下列脚本所示：

```
std::list<int> legalMoves;
for(i=0; i<9; i++)
{
    if(board[i] == 0)
        legalMoves.push_back(i);
}
```

所有的合法移动都放在 STL 列表中，等待下一步的处理。检测棋盘的每个位置，如果是空的，就加入合法移动列表。处理完后，使用下列脚本寻找最佳移动：

```
int bestVal = 0;
std::list<int>::iterator it = legalMoves.begin();
while(it != legalMoves.end())
{
    // look at each winning move and count the
    // spaces (with this move made) that are in it
    for(i=0; i<8; i++)
    {
        int val = 0;
        if((*it) == winningPositions[i][0]) ||
           (board[winningPositions[i][0]] == sidetomove))
            ++val;
    }
}
```

```

    if ((*it) == winningPositions[i][1]) ||
        (board[winningPositions[i][1]] == sidetomove))
        ++val;
    if ((*it) == winningPositions[i][2]) ||
        (board[winningPositions[i][2]] == sidetomove))
        ++val;
    if ((board[winningPositions[i][0]] != sidetomove) &&
        (board[winningPositions[i][0]] != 0))
        val = -1;
    if ((board[winningPositions[i][1]] != sidetomove) &&
        (board[winningPositions[i][1]] != 0))
        val = -1;
    if ((board[winningPositions[i][2]] != sidetomove) &&
        (board[winningPositions[i][2]] != 0))
        val = -1;

    if (val > bestVal)
    {
        move = (*it);
        bestVal = val;
    }
}

++it;
}

```

每个合法移动都被评估并且打分，以此确定最佳移动。计算方法是该移动可以匹配多少获胜组合。如果因为某个位置被计算机占据而不能满足获胜条件，则设为负值。生成了所有分数后，会和目前最高分进行比较，如果更高，就把当前分数设为最高分。这个方法称做“求最大值算法”，在很多地方都很有用。所有组合都被检测后，就可以进行下一步了，看看是否“没有可以移动的位置”，如下所示：

```

if (move == -1)
{
    move = (*legalMoves.begin());
}

lua->PushNumber(move+1); // adjust for 1 base in Lua code
return 1;
}

```

如果没有找到合适的移动方式，就返回列表的第一个移动位置。这个函数只是个开始，有很多可以改进的地方。一个建议是可以考虑加入某种防御检测：检测是否有移动可以阻止对方的胜利，这样可以得到更高的分数。

## 13.4 《Take Away》游戏的实现

现在回到《Take Away》这个游戏，看一看它的多种控制方式，如图 13.3 所示，控制飞船朝不同敌人的方向飞行。

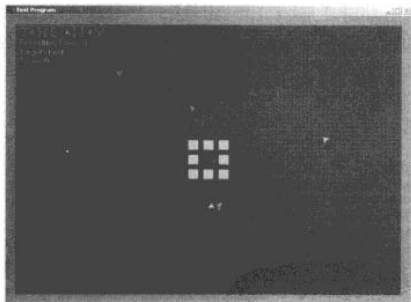


图 13.3 在《Take Away》中我们处理 4 种不同的 AI 行为

敌舰的 AI 在 `EnemyFacing()` 函数中，`GUI_TIMER_EXPIRED` 事件发生时根据敌舰的 `REACT` 设定来调用。总体来看，这个函数是游戏里最复杂的函数之一。`EnemyFacing()` 函数可以简单地分为 4 部分，每个部分负责一种 AI。每种 AI 都使用 `EnemyFacing()` 函数定位特定的目的地（被称为“目标”），然后调用 `SetTravelDirection (indx, tX, tY)` 调整方向，指向目标。在《Take Away》中，我们建立了 4 种完全不同的敌舰：掠夺舰（Box Grabbers）、攻击舰（Shooters）、冲击舰（Rammers）和混合舰（Hybrids）。下面分别介绍。

### 13.4.1 掠夺舰

```
function EnemyFacing(indx)
--indx is the myEnemies table index assigned to the enemy
  if myEnemies[indx].ID == nil then
    if myEnemies[indx].TYPE == 1 then --Box grabbers
      for i = 1, targetCount do
        if myEnemies[indx].E_TOW == 'no' then --Enemy not towing
          if myTargets[i].T_TOW == 'no' then --target free
            SetTravelDirection(indx, myTargets[i].T_X + 10,
```

```
        myTargets[i].T_Y + 10)
    end
end
end
```

如前面的脚本所示，掠夺舰的目标是拖走补给箱。它并不关心自身的安危，只是加速飞向没有被拖走的补给箱，然后把它拖出屏幕（如此往复）。通过查找 myTargets 表寻找有效补给箱，找到后，利用 T\_X 值、T\_Y 值和补给箱一半的尺寸来确定箱子的中心位置。之后，调用 SetTravelDirection (indx, tX, tY) 函数，将箱子的中心坐标设定为目标位置 (tX 和 tY)。

### 13.4.2 攻击舰

```
elseif myEnemies[indx].TYPE == 2 then --Shooters
    SetTravelDirection(indx, myX + 10, myY + 10)
    --Enemy shooting script (see EnemyRespawn(indx) function)
    myEnemies[indx].FIRE = myEnemies[indx].FIRE + 1
    if myEnemies[indx].FIRE == enemyFireInterval then
        FireProjectile(indx, myEnemies[indx].XTHRUST,
            myEnemies[indx].YTHRUST)
        myEnemies[indx].FIRE = 0
    end
end
```

攻击舰的唯一目标是通过发射无限的炮弹消灭玩家。调用 SetTravelDirection (indx, tX, tY) 函数设定有效的玩家飞船坐标为目标。攻击舰的脚本特别的地方在于它能控制敌舰的射击。每次调用 EnemyFacing() 函数，攻击舰的 FIRE 参数就会增加，当 FIRE 参数达到 enemyFireInterval 时调用 FireProjectile (ship, xThrust, yThrust) 函数，然后重置 FIRE 参数。系统可以通过 enemyFireInterval 和敌舰的 REACT 参数来调整射击的固定间隔，这可以让 AI 随着游戏的进行逐渐调整难度，就像设计文档中提到的那样。

### 13.4.3 冲击舰

```
elseif myEnemies[indx].TYPE == 3 then --Rammers
    SetTravelDirection(indx, myX + 10, myY + 10)
```

这段简单的 AI 或许让人觉得冲击舰是游戏中最弱的敌人。毕竟，从这段代码看来，冲击舰就像是去掉了射击功能的攻击舰。不过，冲击舰的优点是它的最大速度 (MAX 值)。由于具有其他飞船两倍的速度，所以冲击舰在 EnemyFacing() 函数中是非常具有“挑战性”

的敌舰。

### 13.4.4 混合舰

混合舰可以像掠夺舰一样拖箱子，也可以像攻击舰一样发射炮弹。混合舰到底是拖箱子还是攻击玩家取决于它们之间的距离。基本逻辑是比较与玩家之间的距离和与最近箱子之间的距离，然后选择最近的一个。确定之后调用 SetTravelDirection (indx, tX, tY) 设定对应目标的位置，如下所示：

```
elseif myEnemies[indx].TYPE == 4 then --Hybrids
    if myEnemies[indx].E_TOW == "no" then --Enemy not towing
        --Determine distance from player
        playerDistance = math.sqrt(((myEnemies[indx].EX + 10) -
            (myX + 10))^2 + ((myEnemies[indx].EY + 10) - (myY +
            10))^2)
        targetDistance = 10000
```

首先，我们使用勾股定理计算混合舰中心到玩家飞船中心的距离 (playerDistance)，然后给 targetDistance 设定一个很大的值。这样做的原因可以参考接下来的代码：

```
for i = 1,targetCount do
    if myTargets[i].T_TOW == "no" then --Target free
        --Determine distance from closest target
        tempTargetDistance = math.sqrt(((
            myEnemies[indx].EX + 10) - (myTargets[i].T_X +
            10))^2 + ((myEnemies[indx].EY + 10) -
            (myTargets[i].T_Y + 10))^2)
        if tempTargetDistance < targetDistance then
            targetDistance = tempTargetDistance
```

接下来，在 for 循环中遍历所有的补给箱，计算和混合舰的距离 (tempTargetDistance)。如果当前距离 (tempTargetDistance) 小于之前的最近距离 (targetDistance)，则设定当前距离为最近距离。开始为 targetDistance 设定的大数值可以保证找到更近的距离，这样就可以比较到最近箱子的距离和到玩家飞船的距离了，如下所示：

```
--Evaluate for the closest option
if playerDistance < targetDistance then
    tX = myX + 10
    tY = myY + 10
    --Enemy shooting script
    myEnemies[indx].FIRE =
    myEnemies[indx].FIRE + 1
    if myEnemies[indx].FIRE ==
    enemyFireInterval then
```



```

        FireProjectile(indx,
            myEnemies[indx].XTHRUST,
            myEnemies[indx].YTHRUST)
        myEnemies[indx].FIRE = 0
    end
else
    tX = myTargets[i].T_X + 10
    tY = myTargets[i].T_Y + 10
end
end
else
    tX = myX + 10
    tY = myY + 10
end
end
SetTravelDirection(indx, tX, tY)
end
end
end
end
end

```

如果玩家飞船的距离更近，混合舰就设定目标为玩家飞船并且执行攻击脚本。如果补给箱更近，就把它作为目标而忽略玩家飞船。虽然脚本第一眼看上去比较复杂，实际上只是一个比较简单的根据参数决定选择的系统。

### 13.4.5 控制飞行方向

学习了 EnemyFacing (indx) 函数后，《Take Away》游戏的人工智能部分只剩下一个需要研究的函数。这个函数是之前提到过的 SetTravelDirection (indx, tX, tY)。虽然之前出现了很多次，但是我们还没有说明它的原理和内部结构。SetTravelDirection (indx, tX, tY) 专门用来更新敌舰的 ROT 值，通过敌舰坐标和它的目标间的数学运算。DrawEnemyShip() 函数使用 myEnemies 表的 ROT 值更新 XTHRUST 和 YTHRUST 值让敌舰攻击并追击目标。根据两个物体的中心 x 坐标的比较，该函数可以分为 3 个部分。第一部分是当目标的 x 坐标小于敌舰的 x 坐标的情况，如下所示：

```

function SetTravelDirection(indx, tX, tY)
--indx is the myEnemies table index assigned to the enemy
--tX is the X coordinate of the goal (target or player's ship)
--tY is the Y coordinate of the goal (target or player's ship)
    if tX < myEnemies[indx].EX then
        if tY < myEnemies[indx].EY then
            if (myEnemies[indx].EY - tY) > 150 then
                if (myEnemies[indx].EX - tX) > 150 then

```

```

        myEnemies[indx].ROT = 8 --Up/left
    else
        myEnemies[indx].ROT = 1 --Up
    end
elseif (myEnemies[indx].EY - tY) > 30 then
    myEnemies[indx].ROT = 8 --Up/left
else
    myEnemies[indx].ROT = 7 --Left
end
elseif tY == myEnemies[indx].EY then
    myEnemies[indx].ROT = 7 --Left
elseif tY > myEnemies[indx].EY then
    if (tY - myEnemies[indx].EY) > 150 then
        if (myEnemies[indx].EX - tX) > 150 then
            myEnemies[indx].ROT = 6 --Down/left
        else
            myEnemies[indx].ROT = 5 --Down
        end
    elseif (tY - myEnemies[indx].EY) > 30 then
        myEnemies[indx].ROT = 6 --Down/left
    else
        myEnemies[indx].ROT = 7 --Left
    end
end
end
end

```

小点的部分是 x 坐标相等时的处理，如下所示：

```

elseif tX == myEnemies[indx].EX then
    if tY <= myEnemies[indx].EY then
        myEnemies[indx].ROT = 1 --Up
    else
        myEnemies[indx].ROT = 5 --Down
    end
end

```

最后一个部分是目标的 x 坐标大于敌舰的 x 坐标时的处理，因为这部分的结构和第一部分类似，所以这里就不再说。阅读了第一部分的概要后，很快地查看一遍第三部分的处理就可以明白它的内容。

### 13.4.6 碰撞检测

现在我们来了解一下 InGame 处理的最后一个主要部分：碰撞检测。处理不同物体间碰撞的 3 个函数分别是 EnemyTowCheck()、EnemyHitCheck() 和 CasualtyCheck()。这些 LuaSupport.lua 函数是最复杂的，也是 InGame 中最重要的部分。尽管各有不同，不过它们结构是相似的。我们首先解释 EnemyTowCheck() 函数的碰撞检测（在 LuaSupport.lua 脚本中，

如下所示)。该函数检测掠夺舰或者混合舰是否碰到了没有被拖动的箱子，碰到后就拖走它。

```
function EnemyTowCheck()
  for i = 1, enemyCount do
    if myEnemies[i].ID == nil then
      j = 1
      while ((myEnemies[i].E_TOW == "no") and (j < (targetCount + 1))) do
        if (myTargets[j].T_TOW == "no") and ((myEnemies[i].TYPE == 1) or (myEnemies[i].TYPE == 4)) then --Enemies are box grabbers or hybrids & the target is free for towing
```

函数体包含在 for 循环内，遍历了游戏中的所有敌舰。确定了敌舰是在游戏中后，我们使用一个 while 语句处理所有的补给箱。当前敌舰的 E\_TOW 状态也是 while 语句的判断条件，这两个条件（E\_TOW 状态和游戏中的箱子数）保证了在循环中可以处理所有的补给箱，同时排除被拖动的箱子。while 控制结构在这里很重要，因为在循环中 E\_TOW 的状态可能会发生改变。在 for 循环中，E\_TOW 状态只会判断一次（在循环的开始），就可能出现数据不一致的情况。参考下面的函数片段，可以帮助我们解释这个问题：

```
    if ((myEnemies[i].EX + 20) < myTargets[j].T_X) or
       (myEnemies[i].EX > (myTargets[j].T_X + 20)) or
       ((myEnemies[i].EY + 20) < myTargets[j].T_Y) or
       (myEnemies[i].EY > (myTargets[j].T_Y + 20)) then
      --No collision
    else
      --Collision
      --Indexes the ID of each entity into the
      appropriate portion of the other's table
      myTargets[j].T_TOW = myEnemies[i].ID
      myEnemies[i].E_TOW = myTargets[j].T_ID
    end
    j = j + 1
  end
end
end
end
end
```

通过计算两个物体是否处在合理位置来进行碰撞检测。如果发生碰撞，箱子和敌舰的 ID 会被保存为对方的拖动状态参数。最后，存储箱的索引增加，进行下一步处理。大体来说，函数的基本结构如下：对于每个敌舰，对所有可拖动箱子进行碰撞检测。如果碰撞发生，碰撞物体的 E\_TOW 和 T\_TOW 值设定为碰撞发生的双方，然后终止该 while 循环，对

下一个敌舰继续该 for 循环的碰撞检测。

EnemyHitCheck() 函数中（参考 LuaSupport.lua 文件）也可以看到这个简单的结构。因为两个函数十分相似，所以这里只关注玩家炮弹和敌舰的碰撞检测，如下所示：

```
--Collision
if myEnemies[i].E_TOW == 'no' then --Enemy
    was towing
    for k = 1,targetCount do
        if myTargets[k].T_TOW ==
            myEnemies[i].ID then
            --Makes the target available for
            towing again
            myTargets[k].T_TOW = 'no'
        end
    end
end
end
```

碰撞发生时，检测敌舰是否拖着储物箱，如果是则让敌舰丢弃它——停留在最后的位置。需要注意的是，T\_TOW 参数被设为了 no，因为箱子还没有被拖到屏幕外，所以不是 done 状态。最后，处理碰撞的两个物体，如代码清单 13.3 所示。

代码清单 13.3 删除敌舰并清除表

```
--Deletes the enemy and clears its table

indexes
DeleteItem(myEnemies[i].ID)
myEnemies[i].XTHRUST = nil
myEnemies[i].YTHRUST = nil
myEnemies[i].ROT = nil
myEnemies[i].EX = nil
myEnemies[i].EY = nil
myEnemies[i].ID = nil
myEnemies[i].E_TOW = 'no'
myEnemies[i].FIRE = nil
--Respawns the enemy if the game is not over
if targetDoneCounter ~= targetCount then
    UpdateScore(myEnemies[i].TYPE)
    EnemyRespawn(i)
end
--Deletes the projectile and clears its table

indexes
DeleteItem(myProjectiles[j].PROJ_ID)
myProjectiles[j].PROJ_X = nil
myProjectiles[j].PROJ_Y = nil
myProjectiles[j].PROJ_XTH = nil
myProjectiles[j].PROJ_YTH = nil
```

```

        myProjectiles[j].PROJ_ID = nil
        myProjectiles[j].PROJ_SHIP = nil
    end
end
end
j = j + 1
end
end
end
end
end
end

```

炮弹和敌舰的图像都被删除，它们相关联的表数据也被设置为 nil。和 T\_TOW 一样，E\_TOW 参数也需要重置为有效状态。另外，积分榜的数据也需要更新（参考 GUI\_InGame.lua 文件中的 UpdateScore (enemyType) 函数）。如果游戏还没有结束，调用 EnemyRespawn (indx) 函数，参数为刚刚掉的敌舰索引。这个系统保证了游戏中敌舰数量的一致性和处理器的平衡（因为同时只有少量的敌舰数据需要存储）。

与碰撞检测相关的最后一个函数是 CasualtyCheck()，它用来检测玩家飞船和敌舰或者炮弹之间的碰撞。不同于 EnemyHitCheck() 和 EnemyTowCheck() 函数在 GUI\_TIMER\_EXPIRED 事件发生时调用，该函数用在 DrawShip (myRot, x, y) 函数中，只在玩家飞船存在的时候调用。函数分成两个部分，第一部分处理飞船间的碰撞，第二部分负责炮弹相关的碰撞。因为 2/3 的飞船间的碰撞检测代码结构和 EnemyHitCheck() 类似，第二部分又和第一部分关系紧密，所以这里我们只讨论第一部分的少量代码，如下所示：

```

function CasualtyCheck()
    --Collisions with enemies
    i = 1
    while ((alive == "yes") and (i < (enemyCount + 1))) do
        if myEnemies[i].ID ~= nil then
            if ((myEnemies[i].EX + 20) < myX) or (myEnemies[i].EX > (myX
                + 20)) or ((myEnemies[i].EY + 20) < myY) or (myEnemies[i].EY
                > (myY + 20)) then
                --No collision
            else
                --Collision
                --Delete the player's ship
                DeleteItem(GUI_RUNTIME_SPRITES + 100)
                alive = "no"
            end
        end
        i = i + 1
    end
end

```

这里需要注意的部分是 while 语句的条件，特别是 alive 值的判断，它是必要的，因为这个值可能在循环中被改变。因此，如果玩家飞船被删掉，循环就终止，此时敌舰就不需要再检测和玩家飞船间的碰撞。函数的第二部分是类似的，只需要快速浏览就可以明白它

的处理过程。

这里只是概要地说明了多物体间的碰撞检测。有了 InGame 最后这一部分的逻辑,《Take Away》游戏的体验就完整了。我们通过学习这些增强游戏性的函数,现在对于它们之间的协作也更容易理解了。使用游戏事件、GUI 界面、函数、表、全局和局部变量以及 Lua 语言提供的工具,我们开发了一个具有挑战性、娱乐性及启发性的交互环境。希望学习到的这些技能能够成为用户自己开发游戏时的有效工具。

## 13.5 其他 AI 的例子

现在我们花点时间来看一看《Take Away》加入的一些变化内容。这些变化保存在 CD-ROM 的第 13 章的 AI Sandbox 文件夹下。可执行文件和《Take Away》一样,只是脚本有些变化。例子包括了 8 种不同的 AI 行为,通过改变脚本,可以看到 GUI\_MainMenu.lua 的按钮事件如何选择脚本。

### 13.5.1 静态追踪

该例子演示了敌舰如何追踪静态的目标。这里我们使用和《Take Away》游戏中敌舰追踪补给箱类似的机制。例子中最重要的是 StationaryTrackSetTravelDirection() 函数。该函数根据目标位置设定飞船方向,如代码清单 13.4 所示。

代码清单 13.4 StationaryTrackSetTravelDirection() 函数

```
function StationaryTrackSetTravelDirection(indx, tX, tY)
--indx is the myEnemies table index assigned to the enemy
--tX is the X-coordinate of the goal (target or player's ship)
--tY is the Y-coordinate of the goal (target or player's ship)
    if tX < myEnemies[indx].EX then
        if tY < myEnemies[indx].EY then
            if (myEnemies[indx].EY - tY) > 150 then
                if (myEnemies[indx].EX - tX) > 150 then
                    myEnemies[indx].ROT = 8 --Up/left
                else
                    myEnemies[indx].ROT = 1 --Up
                end
            elseif (myEnemies[indx].EY - tY) > 30 then
                myEnemies[indx].ROT = 8 --Up/left
            else
                myEnemies[indx].ROT = 7 --Left
            end
        end
    end
```

```

    elseif tY == myEnemies[indx].EY then
        myEnemies[indx].ROT = 7 --Left
    elseif tY > myEnemies[indx].EY then
        if (tY - myEnemies[indx].EY) > 150 then
            if (myEnemies[indx].EX - tX) > 150 then
                myEnemies[indx].ROT = 6 --Down/left
            else
                myEnemies[indx].ROT = 5 --Down
            end
        elseif (tY - myEnemies[indx].EY) > 30 then
            myEnemies[indx].ROT = 6 --Down/left
        else
            myEnemies[indx].ROT = 7 --Left
        end
    end
elseif tX == myEnemies[indx].EX then
    if tY <= myEnemies[indx].EY then
        myEnemies[indx].ROT = 1 --Up
    else
        myEnemies[indx].ROT = 5 --Down
    end
elseif tX > myEnemies[indx].EX then
    if tY < myEnemies[indx].EY then
        if (myEnemies[indx].EY - tY) > 150 then
            if (tX - myEnemies[indx].EX) > 150 then
                myEnemies[indx].ROT = 2 --Up/right
            else
                myEnemies[indx].ROT = 1 --Up
            end
        elseif (myEnemies[indx].EY - tY) > 30 then
            myEnemies[indx].ROT = 2 --Up/right
        else
            myEnemies[indx].ROT = 3 --Right
        end
    elseif tY == myEnemies[indx].EY then
        myEnemies[indx].ROT = 3 --Right
    elseif tY > myEnemies[indx].EY then
        if (tY - myEnemies[indx].EY) > 150 then
            if (tX - myEnemies[indx].EX) > 150 then
                myEnemies[indx].ROT = 4 --Down/right
            else
                myEnemies[indx].ROT = 5 --Down
            end
        elseif (tY - myEnemies[indx].EY) > 30 then
            myEnemies[indx].ROT = 4 --Down/right
        else
            myEnemies[indx].ROT = 3 --Right
        end
    end
end
end
end
end

```

该函数使用穷举的方法检测目标出现在哪个 45°角区域, 然后设定飞船的方向。下一章中, 我们会给出不同的解决方式。

### 13.5.2 近距离追踪

在这个例子中, 我们在游戏场景中随机放置了两个补给箱和一个敌舰 (《Take Away》中的掠夺舰)。按下 <1> 键让敌舰追踪距当前位置最近的箱子, 按 <2> 键追踪最远的箱子。如何区别距离远近的算法在 ProximityTrackEnemyFacing() 函数中, 如代码清单 13.5 所示。

代码清单 13.5 ProximityTrackEnemyFacing() 函数

```
function ProximityTrackEnemyFacing(indx)
--indx is the myEnemies table index assigned to the enemy
if (myEnemies[indx].E_TOW == "no") and (desiredTarget == "none") then
--Enemy not towing and seeking target
if desiredTarget == "set" then --Initial run
--Determine distance from targets
targetDistance = {}
for i = 1, targetCount do
if myTargets[i].T_TOW == "no" then --Target free
targetDistance[i] = math.sqrt(((myEnemies[indx].EX +
10) - (myTargets[i].T_X + 10))^2 +
((myEnemies[indx].EY + 10) - (myTargets[i].T_Y +
10))^2)
end
end
if (targetDistance[1] == nil) and (targetDistance[2] == nil)
then
--Evaluate for the distance options
if targetDistance[1] < targetDistance[2] then
nearIndx = 1
farIndx = 2
else
nearIndx = 2
farIndx = 1
end
end
if desiredTarget == "near" then
tX = myTargets[nearIndx].T_X + 10
tY = myTargets[nearIndx].T_Y + 10
desiredTarget = "set"
elseif desiredTarget == "far" then
tX = myTargets[farIndx].T_X + 10
tY = myTargets[farIndx].T_Y + 10
desiredTarget = "set"
end
ProximityTrackSetTravelDirection(indx, tX, tY)
```



```

        end
    else
        ProximityTrackSetTravelDirection(indx, tX, tY)
    end
end
end
end

```

该函数依赖 `desiredTarget` 变量，它包含 3 种状态。最开始设为 “none”，表示不追踪任何目标。敌舰简单地在游戏场景中沿直线飞行。通过按键改变状态 “near” 和 “far”。

我们使用代数运算取得和目标物体间的距离，然后比较 `targetDistance [index]` 距离值，以决定哪个坐标更近。根据当前的状态，载入最近或者最远目标的 `tX` 和 `tY` 值。

### 13.5.3 动态追踪

在这个例子中，我们引入了玩家控制的飞船。玩家可以转向、加速并移动飞船，它是敌舰的目标。这里使用和静态追踪相同的方式确定方向，但由于这次目标是移动的，所以敌舰会根据玩家的飞船位置调整方向。具体的处理在 `MovingTrackSetTravelDirection()` 函数中，它根据在每个时间点都可能改变的目标位置调整方向。

### 13.5.4 预判型追踪

在空中格斗时，预测飞机的移动可以帮助玩家更准确地射击。恐怕没有人比飞行员更擅长在飞行中进行几何运算（大部分时候是靠感觉和经验），预测子弹飞行时间里敌人飞机可能的位置。

在这个例子中，我们给敌舰增加了这种能力，如图 13.4 所示。我们在前面例子的基础上，在玩家控制的飞船前进的路线上，让敌舰预测一个提前位置。

这个算法在 `AnticipationTrackSetTravelDirection()` 函数中。该函数和前一节的函数基本相同，只是在开始部分有增加的运算，如代码清单 13.6 所示。

代码清单 13.6 `AnticipationTrackSetTravelDirection()` 函数

```

local enemyDistance = math.sqrt(((myX + 10) - (myEnemies[indx].EX +
10))^2 + ((myY + 10) - (myEnemies[indx].EY + 10))^2)
local enemyRate = math.sqrt(((myEnemies[indx].XTHRUST)^2 +
((myEnemies[indx].YTHRUST)^2))
local time = enemyDistance / enemyRate

local playerRate = math.sqrt(((myXThrust)^2) + ((myYThrust)^2))
local playerDistance = playerRate * time

```

```
if playerDistance > 25 then
    playerDistance = 25
end
local playerXVector = math.sqrt(((playerDistance)^2) --
((myYThrust)^2))
local playerYVector = math.sqrt(((playerDistance)^2) --
((myXThrust)^2))
local dir = GetTravelDirection(myXThrust, myYThrust)
if dir == 1 then --Up
    tY = tY - (playerDistance)
elseif dir == 2 then --Up/right
    tX = tX + (playerXVector)
    tY = tY - (playerYVector)
elseif dir == 3 then --Right
    tX = tX + (playerDistance)
elseif dir == 4 then --Down/right
    tX = tX + (playerXVector)
    tY = tY + (playerYVector)
elseif dir == 5 then --Down
    tY = tY + (playerDistance)
elseif dir == 6 then --Down/left
    tX = tX - (playerXVector)
    tY = tY + (playerYVector)
elseif dir == 7 then --Left
    tX = tX - (playerDistance)
elseif dir == 8 then --Up/left
    tX = tX - (playerXVector)
    tY = tY - (playerYVector)
end
SetItemPosition(GUI_RUNTIME_SPRITES + 150, tX, tY, 4, 4)
```



图 13.4 在这个示例中，由计算机控制的飞船可以预测玩家控制的飞船的移动

增加的脚本运用了基本的几何运算,根据方向、速度和敌舰的距离,确定玩家飞船移动方向的前方的点。脚本动态更改 tX 和 tY 的值来生成移动的目标物体坐标,控制敌舰的移动。

在前面的例子中,躲避敌舰很容易,这也是《Take Away》游戏中所有的逻辑。但在这里,你会发现躲开敌人的撞击是非常困难的,因为敌人知道你的移动方式,程序中速度的更新(受 StartTimer() 函数控制)也比我们想象得快。通过这个例子可以看到,一个简单的调整会给既有的游戏带来更智能的表现。

### 13.5.5 炮塔攻击

在这个例子中,我们创建了一个炮塔,会旋转方向瞄准最近的敌人(可以是计算机控制的或者是玩家控制的飞船),并朝它开火。炮塔调整方向的方式和敌舰追踪动态目标的方式相同(还可以增强它的智能,使用前一小节的方法让炮塔可以预测目标的移动)。

炮塔使用 TurretFireTurretFacing() 函数选择最近的敌人,如代码清单 13.7 所示。

代码清单 13.7 TurretFireTurretFacing() 函数

```
function TurretFireTurretFacing()  
  --Determine distance from player  
  playerDistance = math.sqrt(((turX + 10) - (myX + 10))^2 + ((turY +  
  10) - (myY + 10))^2)  
  enemyDistance = 10000  
  for i = 1,enemyCount do  
    if myEnemies[i].ID ~= nil then --Enemy exists  
      --Determine distance from closest enemy  
      tempEnemyDistance = math.sqrt(((turX + 10) - (myEnemies[i].EX  
      + 10))^2 + ((turY + 10) - (myEnemies[i].EY + 10))^2)  
      if tempEnemyDistance < enemyDistance then  
        enemyDistance = tempEnemyDistance  
      end  
      --Evaluate for the closest option  
      if playerDistance < enemyDistance then  
        tX = myX + 10  
        tY = myY + 10  
      else  
        tX = myEnemies[i].EX + 10  
        tY = myEnemies[i].EY + 10  
      end  
    end  
  else  
    tX = myX + 10
```

```

        tY = myY + 10
    end
    turFireCounter = turFireCounter + 1
    if turFireCounter == 5 then
        TurretFireFireProjectile(0, 0)
        turFireCounter = 0
    end
end
TurretFireSetTravelDirection("turret", tX, tY)
end

```

该函数使用简单的公式  $a^2 + b^2 = c^2$  来计算玩家（炮塔）到所有游戏中活动敌人间的距离，然后设定最近的目标位置并使用 `TurretFireSetTravelDirection()` 函数计算方向。读者会发现，计算方向的函数和之前敌舰使用的函数是相同的，在函数的开始都有一个简单的检测。

### 13.5.6 躲避攻击

这个例子中，敌舰会躲避攻击。如果敌人没有开火，会和前面的例子一样的移动。如果炮弹接近，它就会调转方向躲避。这个处理在函数 `AvoidFireEnemyFacing()` 中，如代码清单 13.8 所示。

代码清单 13.8 `AvoidFireEnemyFacing()` 函数

```

function AvoidFireEnemyFacing(indx)
--indx is the myEnemies table index assigned to the enemy
--Determine distance from projectiles
projectileDistance = 100
tX = myX + 10
tY = myY + 10
state = "none"
for i = 1, pCount do
    if myProjectiles[i].PROJ_ID == nil then --Projectile exists
        --Determine distance from closest target
        tempProjectileDistance = math.sqrt(((myEnemies[indx].EX + 10)
        - (myProjectiles[i].PROJ_X + 2))^2 + ((myEnemies[indx].EY +
        10) - (myProjectiles[i].PROJ_Y + 2))^2)
        if tempProjectileDistance < projectileDistance then
            projectileDistance = tempProjectileDistance
            tX = myProjectiles[i].PROJ_X + 2
            tY = myProjectiles[i].PROJ_Y + 2
            state = "avoid"
        end
    end
end
AvoidFireSetTravelDirection(indx, state, tX, tY)
end

```

projectileDistance 变量设定了敌舰对炮弹的响应范围。我们使用和前面例子相同的距离计算。如果距离小于预警距离，设定敌舰的状态为“avoid”。

在 AvoidFireSetTravelDirection() 函数中，我们加入了下列代码处理这种情况：

```
if state == "none" then
    myEnemies[indx].ROT = Ret
else
    myEnemies[indx].ROT = Rot + 4
    if myEnemies[indx].ROT > 8 then
        dif = myEnemies[indx].ROT - 8
        myEnemies[indx].ROT = 0 + dif
    end
end
end
```

这段代码检测了 state 值，如果状态是“avoid”，当前目标会设定和正常前进方向相反的方向，这样下一帧就会开始躲避炮弹。因为炮弹需要一段时间移动 100 像素的“预警范围”，所以敌舰最终会躲开炮弹。

### 13.5.7 防御性射击

这个例子和前面小节的基本相同，区别在于敌舰会尝试移动到保护墙壁后（没有使用路径寻找——这个之后会提到）。如果对方开火，它会尝试躲避炮弹并回到保护墙后。这个处理在函数 ProtectionFireEnemyFacing() 中，如下所示：

```
function ProtectionFireEnemyFacing(indx)
    --indx is the myEnemies table index assigned to the enemy
    --Determine distance from projectiles
    projectileDistance = 100
    tX = 100
    tY = 300
    state = "protection"
    for i = 1, pCount do
        if myProjectiles[i].PROJ_ID == nil then --Projectile exists
            --Determine distance from closest target
            tempProjectileDistance = math.sqrt(((myEnemies[indx].EX + 10)
            - (myProjectiles[i].PROJ_X + 2))^2 + ((myEnemies[indx].EY +
            10) - (myProjectiles[i].PROJ_Y + 2))^2)
            if tempProjectileDistance < projectileDistance then
                projectileDistance = tempProjectileDistance
                tX = myProjectiles[i].PROJ_X + 2
                tY = myProjectiles[i].PROJ_Y + 2
                state = "avoid"
            end
        end
    end
    end
    ProtectionFireSetTravelDirection(indx, state, tX, tY)
end
```

默认的 tX 和 tY 值是墙后的坐标。该例子可以扩展为动态值，如基地或者其他类似的位置。敌舰会尝试躲避攻击，否则就回到基地。

### 13.5.8 攻击伤害

这个例子引入了伤害的概念，在敌舰的数据表中添加了 myEnemies[indx].DAMAGE 键。当敌舰被击中时，会受到一定伤害。参考代码清单 13.9 中的 DamageFireEnemyFacing() 函数，敌舰会根据条件设定自己的状态。

代码清单 13.9 DamageFireEnemyFacing() 函数

```
function DamageFireEnemyFacing(indx)
--indx is the myEnemies table index assigned to the enemy
--Determine distance from projectiles
projectileDistance = 100
tX = myX + 10
tY = myY + 10
state = "none"
if (myEnemies[indx].DAMAGE >= 3) and (myEnemies[indx].DAMAGE < 6)
then
    print("Now in Avoidance Mode")
    for i = 1, pCount do
        if myProjectiles[i].PROJ_ID ~= nil then --Projectile exists
            --Determine distance from closest target
            tempProjectileDistance = math.sqrt(((myEnemies[indx].EX +
            10) - (myProjectiles[i].PROJ_X + 2))^2 +
            ((myEnemies[indx].EY + 10) - (myProjectiles[i].PROJ_Y +
            2))^2)
            if tempProjectileDistance < projectileDistance then
                projectileDistance = tempProjectileDistance
                tX = myProjectiles[i].PROJ_X + 2
                tY = myProjectiles[i].PROJ_Y + 2
                state = "avoid"
            end
        end
    end
elseif (myEnemies[indx].DAMAGE >= 6) then
    tX = 100
    tY = 300
    state = "protection"
    print("Now in Protection Mode")
end
end
DamageFireSetTravelDirection(indx, state, tX, tY)
end
```

在这个例子中，敌舰完全不在乎“呼啸而过”的炮弹，直到第一次被击中——会躲开一些炮弹，不过还是会被击中。受到了一定程度的伤害后，它会躲进保护墙，如图 13.5 所示。

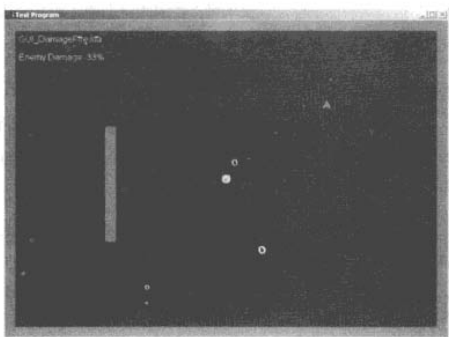


图 13.5 在这个例子中，敌舰在受到一定伤害后会躲进保护墙

接下来的章节中读者会学习到一种有趣方法去控制电脑玩家内部状态。到时候可能会用到这里的例子，看看如何利用那种方法管理这些 AI 行为。

## 13.6 有限状态机

有限状态机是人工智能领域一个很强大的工具。这是一个“虚拟的机器”，可以让程序代码从一种状态迁移到另一种状态。考虑一下自身，尽管你是一个非常复杂度的生物体，但从某种意义上来说，你就像是一个有限状态机。情绪控制着你的行为，有快乐、悲伤、愤怒、害羞、热情等。每种情绪都影响着你的表现，愤怒时的行为一定和害羞时不一样。某种特定的事件可以改变你的状态。想象一下你陷入思考、沉思状态，阅读这本书，光脚在室内行走，然后突然脚趾撞到了桌脚——你的状态立刻就发生了改变，会做出不同的反应。

这个和计算机领域的有限状态机十分相似。这是一种记录当前状态的非常简单的机制，它控制着某种状态下的处理，以及从一种状态变换到另一种状态的条件。可以回想一下一些普通的即时战略（RTS）游戏，AI 控制的军事单位应该有多种状态：原地待命、追击目标、区域巡逻以及逃跑。有限状态机可以让你控制这些单位（和所有独立的单位）的

状态，并且在游戏循环中能够正确处理对应的行为。

尽管有限状态机编程可以作为一本完整的书的主题，但这里我们只介绍如何使用 Lua 开发一个简单的有限状态机。参考下面的函数：

```
function ProcessState(currentState, stateTable)
    if stateTable[currentState] ~= nil then
        return stateTable[currentState]()
    end
    return "Error"
end
```

这几行代码包含了一个完整的有限状态机。再看看下列 4 个函数：

```
function HoldFunction()
    --unit holds position
    if EnemyNearby() then
        return "pursue"
    else
        return "hold"
    end
end

function PursueFunction()
    if DistanceToEnemy() > 10 then
        return "to_base"
    elseif DistanceToEnemy() > 5 then
        return "pursue"
    else
        return "attack"
    end
end

function AttackFunction()
    if EnemyHealth() > 0 then
        return "attack"
    else
        return "to_base"
    end
end

function ToBaseFunction()
    if DistanceToBase() > 0 then
        return "to_base"
    else
        return "hold"
    end
end
```

这些函数说明了我们想象中的军事单位可能采取的行动：原地待命、追击目标、区域巡逻或者返回基地。在函数内部，可以编写脚本定义在每种状态下的行为，这里使用了一



些想象中的函数，我们需要控制的是状态变化的条件。

接下来创建一个表（表示一个单位），如下所示：

```
Unit = {}  
Unit.CurrentState = "hold"  
Unit ["hold"] = HoldFunction  
Unit ["pursue"] = PursueFunction  
Unit ["attack"] = AttackFunction  
Unit ["to_base"] = ToBaseFunction
```

这个表中，表示单位不同状态的字符是表的键。对应的值是函数名，有限状态机会调用它们。

现在可以在主游戏循环发出下列命令来控制军事单位的行动（以及可能的状态改变）：

```
Unit.CurrentState = ProcessState(Unit.CurrentState, Unit)
```

这一个函数会让军事单位执行 `CurrentState()` 函数（当前状态是“原地待命”，对应的是 `HoldFunction()` 函数），然后，执行完成后会返回新的状态（可能是“原地待命”或者“追击目标”）并设定当前状态为该值。通过这种方法，可以在游戏循环中设定每个单元的状态，它们是独立的，还包括自己的流程，控制这些行动和状态变换只需要简单的几行代码即可。

## 13.7 路径寻找

在模拟空间移动的游戏（如即时战略游戏、战争游戏或者动作游戏）中，最重要的部分是确定由计算机控制的物体从点 A 到点 B 的最佳路径。这种处理被称为路径寻找。这些年来，提出了很多解决这个问题方法，其中最广泛使用的是 A\*（读作 A 星）算法。有许多这种算法的优化，这里只介绍一种 Lua 实现的基本方法。在这个基础上，你可以开发适合自己游戏的寻路系统。



该例子可以参考 CD-ROM 中第 13 章的 PathFinding 文件夹。可执行文件和《Take Away》游戏相同，脚本和图像是不同的。

### 13.7.1 算法概要

为了便于说明，参考图 13.6 中的网格图像。和我们在《Take Away》游戏中绘制的边

界，以及前面的 AI 例子中的保护墙一样，这些网格可以称为元像素格或者节点（其他的算法书一般把它称为节点）。

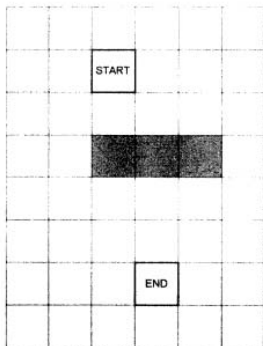


图 13.6 用网格标注开始和目标（结束）节点的小例子，开始和目标之间加了一个障碍

这个算法由两个列表组成：一个开放列表和一个闭合列表。开放列表包含所有需要判断的节点，它可能是最后生成的最佳路径中的节点。闭合列表则包含了最佳路径的节点。

在这个例子中，一个节点包含了下列信息：

- 1) 在游戏场景中的位置：2D 场景中的  $x$  和  $y$  值。
- 2) 父节点：到达该节点的上一个节点。
- 3) 节点的  $G$  值：从星节点到该节点的移动距离。
- 4) 节点的  $H$  值：从该节点到目标节点的预测距离。

下面是该算法的具体步骤：

- 1) 在开放列表中添加开始节点（当前节点）。
- 2) 查看所有和星节点相邻的可以到达的节点（如墙节点是不能到达的）。添加这些节点到开放列表中，设定它们的父节点为星节点。
- 3) 从开放列表中移除星节点，将其添加到闭合列表中。
- 4) 开始循环。
- 5) 查找开放列表中的所有节点，找出  $F$  值（ $G$  值 +  $H$  值）最小的节点。

- 6) 从开放列表中删除该节点, 将其添加到闭合列表中, 设定它为当前节点。
- 7) 检测当前节点的所有相邻节点, 忽略不能到达的节点和闭合列表中的节点。
- 8) 如果节点在开放列表中, 检测当前的 G 值, 如果它比开放列表中节点的值小, 就改变它的父节点, 如果大的话就忽略。
- 9) 当前节点为目标节点时结束循环。
- 10) 现在遍历闭合列表, 从目标节点开始, 父节点组成的链表就是我们要找的路径。
- 11) 有了寻路算法, 现在来看看 Lua 的实现。

### 13.7.2 路径寻找示例

运行路径寻找示例, 可以看到由粉红方块组成的迷宫, 如图 13.7 所示。该模式可以设定开始位置, 单击鼠标右键设定开始节点。

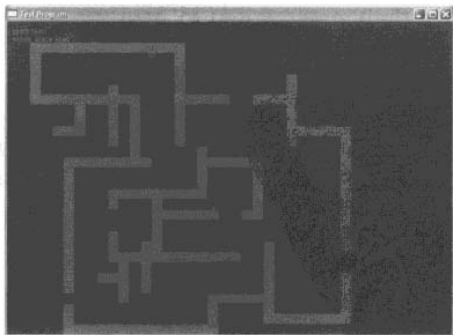


图 13.7 迷宫是测试 A\* Lua 实现的一个很好的例子

按下空格键切换模式, 开始设定目标节点。在迷宫中再次单击鼠标右键设置目标位置。程序开始计算最佳路径, 并用灰色方块标记, 如图 13.8 所示。按 < Enter > 键可以重置。

查看 LuaSupport.lua 脚本, 可以看到如何使用一个大的二维列表定义游戏场景。你可以修改 0 和 1 的值创建自己的迷宫或者进行寻路测试。

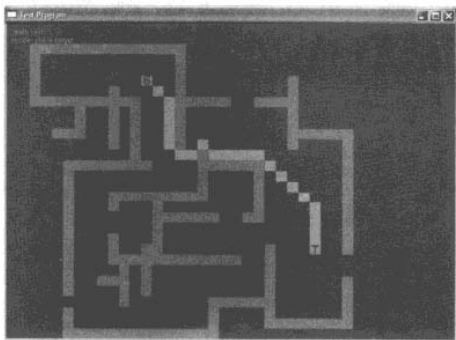


图 13.8 设置了目标节点后，程序会生成最佳路径

### 13.7.3 Lua 实现

在我们的 Lua 实现中，使用了 3 个表，都定义在 `InitWorld()` 函数中：

```
open = {}  
closed = {}  
path = {}
```

开放列表和闭合列表保存节点，路径表保存直接的结果：最佳路径的 x 和 y 坐标。

当玩家设定了开始位置后（通过鼠标右键单击），我们使用下面的函数在开放列表中保存节点：

```
function DefineStart(myX, myY)  
    startX = myX  
    startY = myY  
    --add start location to first item in open list  
    open[1] = {}  
    open[1].X = startX  
    open[1].Y = startY  
    open[1].Parent = 0  
    open[1].G = 0  
    open[1].H = 0  
end
```

因为这是第一个节点，所以不需要计算 G、H 或者 F 的值，它也没有父节点。

当玩家使用鼠标右键单击设置目标节点后，执行下面的函数，开始路径寻找计算：

```
function PlaceTarget(myX, myY)
    RefreshWorld()
    DrawStart(startX, startY)
    DefineStart(startX, startY)
    myX = math.floor(myX/20)
    myY = math.floor(myY/20)
    if (myX < maxX-1) and (myX > 1) and (myY < maxY-1) and (myY > 1) then
        if world[myY][myX] == 1 then
            DefineTarget(myY, myX)
            DrawTarget(targetX, targetY)
            --is the line below needed?
            --open[1].H = 10*(math.abs(startX-targetX) + math.abs(startY-
            targetY))
            repeat
                targetFound = FindPath()
            until targetFound
            BuildPath()
            DrawPath(path)
        end
    end
end
```

函数的第一部分一目了然，转换屏幕坐标为游戏场景的坐标、重新绘制游戏场景并且准备好开始寻找路径。函数的核心部分是循环控制结构，执行 FindPath() 函数，直到 targetFound 被设为 true。之后，通过闭合列表生成路径，并绘制在屏幕上。

接下来看一看 FindPath() 函数，如代码清单 13.10 所示。

代码清单 13.10 FindPath() 函数

```
function FindPath()
    targetFound = false
    openIndex = FindLowestF(open)
    closedIndex = table.getn(closed) + 1
    table.setn(closed, closedIndex)
    closed[closedIndex] = {}
    closed[closedIndex].X = open[openIndex].X
    closed[closedIndex].Y = open[openIndex].Y
    closed[closedIndex].Parent = open[openIndex].Parent
    closed[closedIndex].G = open[openIndex].G
    closed[closedIndex].H = open[openIndex].H
    --Dump()
    --now check out the squares around it
    curX = closed[closedIndex].X
    curY = closed[closedIndex].Y
    --square1
```

```
if not targetFound then
    myX = curX - 1
    myY = curY - 1
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
--square2
if not targetFound then
    myX = curX - 1
    myY = curY
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
--square3
if not targetFound then
    myX = curX - 1
    myY = curY + 1
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
--square4
if not targetFound then
    myX = curX
    myY = curY - 1
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
--square5
if not targetFound then
    myX = curX
    myY = curY + 1
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
--square6
```

```

if not targetFound then
    myX = curX + 1
    myY = curY - 1
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
end
--square7
if not targetFound then
    myX = curX + 1
    myY = curY
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
end
--square8
if not targetFound then
    myX = curX + 1
    myY = curY + 1
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
end
--Dump()
table.remove(open, openIndx)
return targetFound
end

```

该函数首先在开放列表中寻找 F 值最小的节点。我们调用下面的函数进行该查找：

```

function FindLowestF(nodeTable)
    count = table.getn(nodeTable)
    minF = 1000000
    minFIndex = 0
    if count > 0 then
        for indx = 1, count do
            curF = nodeTable[indx].G + nodeTable[indx].H
            if curF <= minF then
                minF = curF
                minFIndex = indx
            end
        end
    end
    return minFIndex
end

```

这个简单的函数遍历表中的所有节点，比较它们的F值（节点的G值和H值之和）和目前最小的F值。查找完所有的节点后，可以返回最小F值的节点索引值。

有了该节点的索引，FindPath()函数将其添加到闭合列表。该节点即为“当前节点”。

读者会注意到代码中的 Dump() 函数被注释掉了，调用该函数可以输出查询的节点，帮助读者了解算法是如何工作的。

然后，我们继续检测当前节点周围的8个节点，第一个节点的代码如下：

```
--square1
if not targetFound then
    myX = curX - 1
    myY = curY - 1
    if (myX > 0) and (myY > 0) and (world[myX][myY] ~= 1) then
        --in bounds and in play
        --does it exist?
        ProcessNode(myX, myY)
    end
end
end
```

通过使用 myX 和 myY 变量设定要查找的节点的值，然后检测是否在边界内，位置是否有效，如果是，则调用代码清单 13.11 中的函数处理该节点。

代码清单 13.11 ProcessNode() 函数

---

```
function ProcessNode(newX, newY)
    targetFound = false
    exists = AlreadyExists(newX, newY)
    if exists == -1 then
        --it's a new open node
        --is it the target???
        if (newX == targetX) and (newY == targetY) then
            --target is found
            targetFound = true
            print("target found")
        else
            --it's a new open node
            NewOpenEntry(newX, newY)
            print("new entry")
        end
    else
        --the node is already in the open table
        print("exists!")
        existingG = open[exists].G
        curValue = 0
        parentG = closed[closedIdx].G
    end
end
```



```

    if (open[exists].X == closed[closedIndx].X) or (open[exists].Y ==
        closed[closedIndx].Y) then
        --NOT diagonal
        curValue = 10
    else
        curValue = 14
    end
    newG = parentG + curValue
    if newG < existingG then
        --open[exists].Parent = openIndx
        open[exists].Parent = closedIndx
    end
end
return targetFound
end

```

首先调用 `AlreadyExists()` 函数检测节点是否存在。如果函数不存在，就继续检测它是否是目标节点（如果是，那么处理结束）。如果不是目标节点，就使用下列函数将其添加到开放列表中：

```

function NewOpenEntry(newX, newY)
    myIndx = table.getn(open) + 1
    table.setn(open, myIndx)
    open[myIndx] = {}
    open[myIndx].X = newX
    open[myIndx].Y = newY
    open[myIndx].Parent = closedIndx
    open[myIndx].G = FindG(open, myIndx)
    open[myIndx].H = FindH(open[myIndx].X, open[myIndx].Y, targetX, targetY)
end

```

该函数比较简单，除了最后一行计算了该节点的 G 值和 H 值。计算 G 值，使用下列函数：

```

function FindG(nodeTable, node)
    parentG = closed[nodeTable[node].Parent].G
    if (nodeTable[node].X == closed[nodeTable[node].Parent].X) or
        (nodeTable[node].Y == closed[nodeTable[node].Parent].Y) then
        --NOT diagonal
        curValue = 10
    else
        curValue = 14
    end
    myG = parentG + curValue
    return myG
end

```

G 值的计算是将父节点的 G 值与当前节点到父节点的距离相加。这里只做简单的计

算，垂直或者水平相邻的节点距离设为 10，斜线距离设为 14。

这个函数会根据不同地表（如道路、空地或者碎石地）的移动消耗来进行相应的修改。

H 值的计算，使用下列函数：

```
function FindH(curX,curY,tarX,tarY)
    --manhattan distance
    myH = 10*(math.abs(curX-tarX) + math.abs(curY-tarY))
    return myH
end
```

该函数计算当前节点到目标节点间的“曼哈顿距离”。曼哈顿距离是一种简单的计算两点间折线距离的方式，就像在曼哈顿街区行走时一样，你没办法走斜线。这是当前节点到目标节点在这种情况下的预测距离（没有考虑地表障碍），对于我们的处理来说，只需要做这样的计算。

回到 ProcessNode() 函数，这里先判断开放列表是否存在该节点，如果存在，就判断 G 值是否有更小的值。如果有更小的值，就改变开放列表中的父节点，如果没有就忽略。

再看 FindPath() 函数，我们对所有相邻节点做该处理，当目标节点要加入开放列表时退出该函数。函数的最后，从列表中移除并返回当前节点。循环该处理直到找到目标节点。

处理结束后，我们得到了由节点组成的闭合列表，其中部分节点是我们最终路径里的节点，另一部分不是。要区分这些节点，需要对闭合列表做如下处理：

```
function BuildPath()
    --once the path is found, this will build the path table
    --load the first node, the target
    count = 1
    path[count] = {}
    path[count].X = targetX
    path[count].Y = targetY
    --load in the last node in the closed table
    count = count + 1
    path[count] = {}
    pathIndex = table.getn(closed)
    path[count].X = closed[pathIndex].X
    path[count].Y = closed[pathIndex].Y
    newPathIndex = closed[pathIndex].Parent
    --now walk through closed table
    while newPathIndex ~= 1 do
        count = count + 1
        path[count] = {}
    end
```

```

    path[count].X = closed[newPathIdx].X
    path[count].Y = closed[newPathIdx].Y
    oldPathIdx = newPathIdx
    newPathIdx = closed[oldPathIdx].Parent
end
path = ReverseTable(path)
--FinalDump()
end

```

该函数遍历闭合列表中的所有节点，根据父节点查找节点，并且添加 x 和 y 坐标到路径列表。当 newPathIdx 等于 1 时，添加开始节点到路径列表，完成处理。这时我们得到了一系列从后往前的坐标值，它们代表从开始到结束的最佳路径。在绘制路径前的最后一步是使用下面的函数颠倒路径表：

```

function ReverseTable(myTable)
    hold = {}
    endCount = table.getn(myTable)
    for idx = 1, table.getn(myTable) do
        hold[idx] = myTable[endCount]
        endCount = endCount - 1
    end
    return hold
end

```

该函数反向表中的所有元素（因为是通用函数，所以其他场合也适用）。在这里，我们得到了一个从开始位置到目标位置的路径表，可以使用下面的函数绘制路径：

```

function DrawPath(myPath)
    --x is the down coordinate
    --y is the across coordinate
    if table.getn(myPath) > 0 then
        for idx = 1, table.getn(myPath) do
            CreateItem(masterCellID, "Sprite", "box_path.bmp")
            SetItemPosition(masterCellID, (myPath[idx].Y * 20) +
                worldOffset, (myPath[idx].X * 20) + worldOffset, 20, 20)
            masterCellID = masterCellID + 1
        end
    end
    --now draw the end as the target
    DrawTarget(myPath[table.getn(myPath)].X, myPath[table.getn
        (myPath)].Y)
end

```

尽管这个例子使用范围有限，但我们还是构建了一个足够通用的方式，可以处理开始和目标位置（想象一下坦克的当前位置和在游戏地图上单击鼠标右键）。得到的结果是一系列的坐标值，表明到终点的各个路径位置。例如，当在地图上移动物体时，可以在任何

移动函数中使用这些数据。当玩家在物体移动过程中重新点击了目标位置时，我们还可以“废弃”这些数据，重新计算。还可以在表中保存多个目标位置——路径点。你可以改变起始位置的当前目标，并且移除路径点列表中最顶部的节点重新创建新的移动顺序。

现在已经有了坚实的理论基础，接下来可以使用 Lua 在任意环境中进行最佳路径的寻找。

## 13.8 本章小结

本章的示例程序只是使用 Lua 实现人工智能的一些最基本的例子。在我们工作室开发的游戏中，使用 Lua 开发的 AI 游戏包括：虚拟美国总统选举游戏、德州扑克、健身中心、冰壶游戏和迷你高尔夫球。Lua 是 AI 的主要开发语言，不管是独立使用还是配合 LuaGlue 函数，都能让你快速开发原型、测试并修改 AI 算法，这是其他需要编译的语言很难做到的。

希望这章的内容可以帮助你为自己的游戏开发出聪明的计算机对手。在下一章中，我们会继续深入复杂的 GUI 系统和 Sprite 类，让你掌握更多的图形配置和 Lua 技巧。



## 第 14 章

Introduction of the book's content

# Lua 和图像


### 本章要点

- 运行绘图示例
- 线性移动
- 碰撞检测
- 2D 粒子系统

在前一章中，我们说明了《Take Away》这个示例游戏，并且使用了一些比较原始的 sprite 图形来显示飞船、补给箱等物体。尽管 Lua 不是用于高性能绘图的工具，但是通过使用一些 LuaGlue 函数，它可以成为一个十分强大的 2D 图形控制系统。

在本章中，我们特别针对业余开发者，扩展了《Take Away》游戏的代码，增加了更多的图形选项，并且提供了一些工具可以用来开发更加复杂而稳定的 2D 游戏。

## 14.1 运行绘图示例

 在 CD-ROM 的第 14 章的文件夹下，可以找到 graphics\_demos 文件夹。示例的可执行文件和《Take Away》游戏一样。执行该文件可以运行独立的示例脚本，我们会在稍后讨论。这些示例从最基础的部分开始，然后逐渐变复杂。在学习它们的过程中，可以思考一下哪些部分可以作为一个函数或者功能加入到自己的 Lua 游戏中去。

想要运行示例程序，可单击文件名右边的按钮。退出时，按 <Esc> 键返回主菜单。

### 14.1.1 指纹示例

第一个例子没有用到新的 LuaGlue 函数，而是在我们接触过的函数基础上，开发了一个很有趣的指纹扫描的动画图像。在 Textures 目录下，可以看到这个动画有 17 帧。我们希望扫描仪可以旋转，这样看起来更像扫描杆在指纹图像上面上下移动，如图 14.1 所示。



图 14.1 计时器事件触发后，上下扫描指纹示例的动态图像

我们使用 `StartTimer()` 函数控制图像，它会在界面第一次载入时调用。接下来一起看看 `GUI_TIMER_EXPIRED` 事件中的控制循环，如下所示：

```
if eventCode == GUI_TIMER_EXPIRED then
    if printDir == POSITIVE then
        printCounter = printCounter + 1
        if printCounter > 17 then -- max number of 2d pics, so
            reverse direction
            printDir = NEGATIVE
            printCounter = 16
        end
    else
        printCounter = printCounter - 1
        if printCounter < 1 then -- min number of 2d pics, so
            reverse direction
            printDir = POSITIVE
            printCounter = 2
        end
    end

    AnimatePrint()
end
```

该循环控制两个变量。第一个变量是 `printDir`，它表示扫描仪是否需要上下移动。该变量值决定我们应该增加还是减少第二个变量——`printCounter`，它保存了当前需要绘制的动画帧的值。

之后使用下面的函数绘制图像，在前面控制循环的最后调用该函数，函数代码如下所示：

```
function AnimatePrint()
    local pic = "ui_tp_01.bmp"
    if printCounter < 10 then
        pic = string.format("%s%d%s", "ui_tp_0", printCounter, ".bmp")
    else
        pic = string.format("%s%d%s", "ui_tp_", printCounter, ".bmp")
    end
    CreateItem(GUI_INGAME + 101, "Sprite", pic)
    SetItemPosition(GUI_INGAME + 101, 264, 120, 272, 360)
    StartTimer(.09)
end
```

该函数调用 `string.format()` 函数生成当前帧对应图像的文件名。然后，我们创建一个 `sprite` 并设置当前图像。因为是在一个已经存在的 GUI 控件上调用 `CreateItem()` 函数，所以只是替换数据而不用生成新的 `Sprite` GUI 控件。该函数调用了 `StartTimer()` 函数，用来回到控制循环。

### 14.1.2 爆炸示例

在这个示例中，点击屏幕任意位置可以看到爆炸动画。爆炸动画共有 13 帧，图片是 TGA 文件（扩展名为 `tga`），包含 `alpha` 通道，能看到爆炸图像重叠。

如果在游戏中使用这样的爆炸动画，要确保爆炸图像的 ID 大于屏幕上的其他控件，这样才能看到爆炸，因为 GUI 系统先绘制 ID 小的控件。

该示例使用的控制机制和之前的指纹示例相同，但它所有的代码都包含在 `GUI_TIMER_EXPIRED` 事件代码中。对于这个动画来说，它不是循环的，只会播放一次。在控制循环中，我们总是会触发计时器，即便没有发生爆炸。这里只发生一次爆炸是因为在鼠标事件处理器中设定了帧计数器，如下所示：

```
if eventCode == GUI_MOUSE_BUTTON_UP then
    if id == LEFT then
        if explosionFrame == 0 then
            tarX, tarY = GetMousePosition()
            explosionFrame = 1
        end
    end
end
end
```

你会发现这是一个新的事件，我们用到了一个新的 LuaGlue 函数：GetMousePosition()。

## 1. 鼠标按键事件

为了触发爆炸，我们需要捕获鼠标点击事件，它和按钮控件没有关系。当鼠标按键改变了按下状态时，会触发 GUI\_MOUSE\_BUTTON\_DOWN 事件和 GUI\_MOUSE\_BUTTON\_UP 事件。鼠标的左、右键都支持，返回的 ID 值表示哪个鼠标键被按下。

## 2. GetMousePosition 函数

C++ 代码使用 Windows 消息系统跟踪鼠标的位置。当鼠标移动时，Windows 发消息给应用程序。C++ 代码跟踪这些移动并且保存位置信息以供内部使用。为了让 Lua 代码访问这些信息，需要编写新的 LuaGlue 函数 GetMousePosition()，并且返回 x 和 y 值。由于 Lua 可以返回多个参数，所以很容易实现。GetMousePosition() 利用了这种特性并返回 x 和 y 的值。Windows 消息控制相关的代码（在 WinMain.cpp 文件中）如下：

```
case WM_LBUTTONDOWN:
    g_mouseButtons |= MOUSE_LBUTTONDOWN;
    CGUIManager::GetInstance()->NotifyMouseButtonDown(1);
    break;
case WM_LBUTTONUP:
    CGUIManager::GetInstance()->NotifyMouseButtonUp(1);

    g_mouseButtons &= ~MOUSE_LBUTTONDOWN;
    break;
case WM_RBUTTONDOWN:
    CGUIManager::GetInstance()->NotifyMouseButtonDown(2);
    g_mouseButtons |= MOUSE_RBUTTONDOWN;
    break;
case WM_RBUTTONUP:
    CGUIManager::GetInstance()->NotifyMouseButtonUp(2);
    g_mouseButtons &= ~MOUSE_RBUTTONDOWN;
    break;
case WM_MOUSEMOVE:
    g_mousePoint.x = LOWORD(lParam);
    g_mousePoint.y = HIWORD(lParam);
    break;
```

消息控制器跟踪了鼠标的状态改变和所有移动信息。g\_mouseButtons 和 g\_mousePoint 变量会同步更新，在程序的很多地方被使用。这些代码还调用 GUI 系统来通知鼠标状态的改变。GUI 系统之后会给 Lua 事件控制器发送新的事件，1 表示鼠标左键；2 表示鼠标右键。为了让 Lua 代码访问该值，需要编写新的 LuaGlue 函数 GetMousePosition()，并且返回 x 和 y 值。由于 Lua 可以返回多个参数，所以很容易实现。GetMousePosition() 利用了这



种特性并返回 x 和 y 的值。GUIManager.cpp 中的代码如下：

```
extern int g_mouseButtons;  
extern POINT g_mousePoint;  
extern 'C' int GUI_GetMousePosition(lua_State *L)  
{  
    cLua *lua = CGUIManager::GetInstance()->GetLuaContext();  
    lua->PushNumber(g_mousePoint.x);  
    lua->PushNumber(g_mousePoint.y);  
    return 2;  
}
```

注意这里的 LuaGlue 函数返回“2”，表示在 Lua 栈有两个返回值。

### 3. 图像旋转

在《Take Away》游戏中，对于飞船和敌舰的旋转我们使用了一个靠得住但较笨的方法：用不同的图片表示不同的方向。这个可以正常工作，但限制了只能有 8 个方向，45°角的变化。这曾经是许多 RTS 游戏采用的方式。现在我们使用 DirectX，它把所有物体都当成 3D 贴图，可以利用这种高级特性来开发我们的系统，如图 14.2 所示。

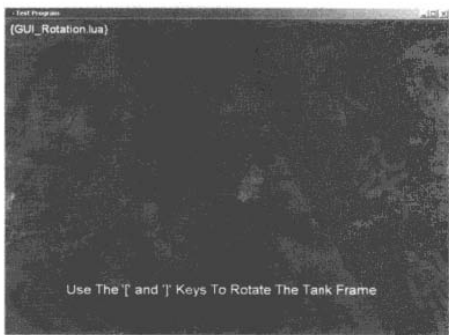


图 14.2 利用了 DirectX 9 的特性可以让我们的图像在屏幕上旋转得更加平滑

在这个例子中，你可以按下 < [ > 和 < [ < ] > 键旋转图像的方向。示例的核心代码在按键事件控制器中，如下所示：

```
if id == 91 then --[
    tankRot = tankRot + .1
    if tankRot > 6.2 then
        tankRot = 0
    end
    ItemCommand(tankID, "SetRotation", tankRot)
end
```

当用户按下 < [ > 键 (ASCII 代码值 91) 后, 就将旋转值 tankRot 增加 0.1。SetRotation 控件命令负责实际的图像旋转。因为旋转单位是弧度, 所以重置为  $2\pi$  和 0。

#### 4. 图像旋转 LuaGlue 函数

旋转命令是图像控件的 ItemCommand 函数。CGUISprite 类 (GUISprite.cpp 文件) 被扩展为支持 ItemCommand 函数, 并且 CSprite 类 (DXSprite.cpp 文件) 这个 DX9 类被修改为追踪旋转角度并且使用标准的定点旋转公式绘制图像。

GUI 图形对象通过普通的 GUI 处理获取 ItemCommand()。命令处理器从 Lua 栈中获取旋转值, 并且设定 DXSprite 对象。下一次绘制图形时 (下一帧), 会显示新的角度。GUISprite.cpp 中的代码如下:

```
int CGUISprite::ObjectCommand(const char *pCommand)
{
    cLua *L = CGUIManager::GetInstance()->GetLuaContext();
    int retVal = 0;
    if(strcmp(pCommand, "SetRotation") == 0)
    {
        // rotate this arg rot (in radians)
        float rot = (float) L->GetNumberArgument(3);
        m_pTexture->SetRotation(-1, -1, rot);
    }
    return retVal;
}
```

当位置或者角度变化时, DXSprite 对象设定标记值, 这样下次更新时可以计算新的位置。DXSprite 类的 UpdatePosition 方法 (参考 DXSprite.cpp 文件) 如代码清单 14.1 所示。

代码清单 14.1 DXSprite 类的 UpdatePosition 方法

```
void Sprite::UpdatePosition(void)
{
    if(m_vb)
```

```

{
    SpriteVertex* pVertices;
    m_vb->Lock(0, 0, (void **)&pVertices, 0);
    pVertices[0].diffuse = pVertices[1].diffuse =
    pVertices[2].diffuse = pVertices[3].diffuse =
        D3DCOLOR_ARGB(255,255,255,255);
    pVertices[0].rhw = pVertices[1].rhw =
    pVertices[2].rhw = pVertices[3].rhw = 1.0f;
    pVertices[0].u0      = 0.0f;
    pVertices[0].v0      = 0.0f;
    pVertices[1].u0      = 1.0f;
    pVertices[1].v0      = 0.0f;
    pVertices[2].u0      = 0.0f;
    pVertices[2].v0      = 1.0f;
    pVertices[3].u0      = 1.0f;
    pVertices[3].v0      = 1.0f;
    pVertices[0].x        = m_x - 0.5f;
    pVertices[0].y        = m_y - 0.5f;
    pVertices[1].x        = m_x + m_width - 0.5f;
    pVertices[1].y        = m_y - 0.5f;
    pVertices[2].x        = m_x - 0.5f;
    pVertices[2].y        = m_y + m_height - 0.5f;
    pVertices[3].x        = m_x + m_width - 0.5f;
    pVertices[3].y        = m_y + m_height - 0.5f;
    if (m_rot != 0)
    { // apply rotation of m_rot radians
        m_centerX = m_x + m_width/2;
        m_centerY = m_y + m_height/2;
        float sn = sinf(m_rot);
        float cs = cosf(m_rot);
        float x = m_x - m_centerX;
        float y = m_y - m_centerY;
        pVertices[0].x = (x * cs) - (y * sn) + m_centerX;
        pVertices[0].y = (x * sn) + (y * cs) + m_centerY;
        x = m_x + m_width - m_centerX;
        y = m_y - m_centerY;
        pVertices[1].x = (x * cs) - (y * sn) + m_centerX;
        pVertices[1].y = (x * sn) + (y * cs) + m_centerY;
        x = m_x - m_centerX;
        y = m_y + m_height - m_centerY;
        pVertices[2].x = (x * cs) - (y * sn) + m_centerX;
        pVertices[2].y = (x * sn) + (y * cs) + m_centerY;
        x = m_x + m_width - m_centerX;
        y = m_y + m_height - m_centerY;
        pVertices[3].x = (x * cs) - (y * sn) + m_centerX;
        pVertices[3].y = (x * sn) + (y * cs) + m_centerY;
    }
    pVertices[0].z      = m_z - 0.5f;
    pVertices[1].z      = m_z - 0.5f;
    pVertices[2].z      = m_z - 0.5f;
}

```

```
pVertices[3].z = m_z - 0.5f;
m_boundingRect.top = (long) ( __min ( __min (pVertices[0].y,
pVertices[1].y), __min (pVertices[2].y, pVertices[3].y)));
m_boundingRect.left = (long) ( __min ( __min (pVertices[0].x,
pVertices[1].x), __min (pVertices[2].x, pVertices[3].x)));
m_boundingRect.bottom = (long) ( __max ( __max (pVertices[0].y,
pVertices[1].y), __max (pVertices[2].y, pVertices[3].y)));
m_boundingRect.right = (long) ( __max ( __max (pVertices[0].x,
pVertices[1].x), __max (pVertices[2].x, pVertices[3].x)));
m_vb->Unlock();
}
}
```

当设定了“脏标记”时就运行这段程序。它会检测图像的 DirectX 顶点缓冲区是否存在，如果缓冲区不存在，就不做处理。另外，顶点缓存会被锁住，这样组成图像的顶点可以被访问。首先，我们设定图像为未旋转位置，然后检测旋转值是否不为 0，如果存在旋转值，每个顶点会以图像中心为轴旋转  $m\_rot$  弧度。最后，根据图像的旋转生成新的边框矩形。

## 14.2 线性移动

在这个移动的例子中，我们使用了和《Take Away》游戏的敌舰移动不同的方式。在那个游戏中，敌舰会锁定目标、调整方向、使用推进力向目标飞行。其结果，特别是对于移动的目标来说，是一个调整的循环，距离目标越来越近，但不是一个直接的路径。

在这个示例中，我们直线移动图像到地球上通过点击设定的目标位置。该方法需用到了简单的直角三角形算法，我们会使用在许多场合都很有用的函数来计算它。这里需要确定朝向目标的角度，然后想象一条连接坦克和目标的线，根据当前时间和速度确定  $x$  和  $y$  值，然后向目标移动。具体处理参考 `MovementApproachTarget()` 函数，如代码清单 14.2 所示。

代码清单 14.2 `MovementApproachTarget()` 函数

```
function MovementApproachTarget(curX, curY, tarX, tarY, moveSpeed)
--first, figure out the sides of the triangle:
a = curY - tarY
b = curX - tarX
c = math.sqrt((a*a)+(b*b))
--now, find the angle to the target:
angle = math.acos(b/c)
```

```

--now, determine the presence of negatives:
if a > 0 then
    yNeg = -1
    if b > 0 then
        xNeg = -1
    elseif b == 0 then
        xNeg = 0
    else
        xNeg = -1
    end
elseif a == 0 then
    yNeg = 1
    if b > 0 then
        xNeg = -1
    else
        xNeg = -1
    end
else
    yNeg = 1
    if b > 0 then
        xNeg = -1
    elseif b == 0 then
        xNeg = 0
    else
        xNeg = -1
    end
end
--now, determine the x and y movement:
movX = math.floor(moveSpeed * math.cos(angle) * xNeg)
movY = math.floor(moveSpeed * math.sin(angle) * yNeg)
--now, update your current position:
curX = curX + movX
curY = curY + movY
tankX = curX
tankY = curY
SetItemPosition(tankID, tankX, tankY, 78, 34)
end

```

函数首先要确定到目标的距离，不是曼哈顿距离——那种“阶梯状”路线的计算方法，而是直线距离。得到正确的角度和直角三角形的斜边长度，如图 14.3 所示。

得到距离后，就可以使用下面的代码计算角度：

```
angle = math.acos(b/c)
```

下一步是处理可能的例外值，因为在游戏中，屏幕左上角是 (0, 0)，因此，当向右下移动时，x 和 y 会增加（和一般的笛卡儿网格法不同）。

处理了这些例外坐标后，我们使用下面的代码计算下一个更新周期的新位置 x 和 y 值：

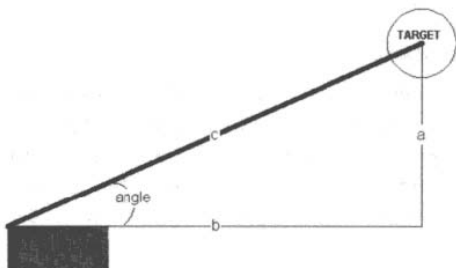


图 14.3 在图中我们可以很容易地确定  $a$  和  $b$ ，并通过简单的数学计算得到  $c$  和到目标的角度

```
movX = math.floor(moveSpeed * math.cos(angle) * xNeg)
```

这里根据 `moveSpeed`（在一个更新循环内沿  $c$  方向移动的距离）和坐标系统的调整系数（`xNeg`）计算出了向目标移动的  $\Delta X$ 。我们使用 `math.floor()` 函数舍去小数部分（还可以加 0.5 来取整），因为这里只关心整数部分。最后，使用下面的脚本设定图像的新位置：

```
SetItemPosition(tankID, tankX, tankY, 78, 34)
```

和其他的示例一样，这个例子也是通过 `GUI_TIMER_EXPIRED` 事件控制，如下所示：

```
if eventCode == GUI_TIMER_EXPIRED then
    if ((tarX ~= nil) and (tarY ~= nil)) then
        MovementApproachTarget(tankX, tankY, tarX, tarY, tankSpeed)
        hitTable = HitTest(tarX, tarY)
        for index,value in ipairs(hitTable) do
            if value == tankID then
                tarX = nil
                tarY = nil
            end
        end
        StartTimer(.1)
    end
end
```

判断是否到达目标位置，使用另一个 LuaGlue 函数——`HitTest()`。这个函数是 Lua 开发的游戏系统中新引入的两个碰撞检测函数之一。`HitTest()` 返回值是一张表，它包含了所有与传入位置发生碰撞的 GUI 对象。然后使用 `ipairs()` 遍历该表，看看是否有图像和目标

点碰撞，如果有，设定 tarX 和 tarY 为 nil，表示处理完毕。

### 14.2.1 GetCollisions 函数

因为子画面（sprite）不仅仅是作为静态图片来使用，所以需要检测它们之间的碰撞。该处理应该使用 C++ 代码来完成，因为所有的信息都在 C++ 程序中。子画面的旋转让函数需要一些技巧来处理特殊形状的子画面。传入的子画面可以和多个子画面发生碰撞，因此需要一张表来返回所有发生碰撞的子画面。只要熟悉了代码，建立表并且返回是很简单的。下面的 LuaGlue 函数调用碰撞检测函数并且使用标准模板库（参考 GUIManager.cpp）创建了表。

```
extern "C" int GUI_GetCollisions(lua_State *L)
{
    cLua *lua = CGUIManager::GetInstance()->GetLuaContext();
    unsigned int id = (unsigned int) lua->GetNumberArgument(1);
    // create the return table on the stack
    lua_newtable (L);
    CUIterInterface *pUI = CGUIManager::GetInstance()->GetCurrentUI();
    unsigned int i = 1;
    if(pUI)
    {
        std::list<unsigned int> list = pUI->SpriteCollision(id);
        std::list<unsigned int>::iterator it = list.begin();
        while(it != list.end())
        {
            lua_pushnumber(L, i);
            lua_pushnumber(L, (*it));
            lua_settable (L, -3);
            ++i;
            ++it;
        }
    }
    // now add the number of entries in the table as member "n" (t.n)
    lua_pushstring(L, "n");
    lua_pushnumber(L, i-1);
    lua_settable (L, -3);
    return 1;
}
```

该表需要在 Lua 栈中生成，索引和值也需要保存在 Lua 栈中。调用 Lua API 后，表会存在栈顶，并且插入表数据。n 属性也被插入表中，它包含了表中数据的个数（不包含 n 属性本身）。在这个例子中，表中包含了成员 table [1] 到 table [table.n]，这样便于 Lua 程序员编写循环，处理返回的所有 ID 值。下面的例子展示了处理返回列表的基本循环：

```

T = GetCollisions(1)
for indx = 1,T.n do
    -- Process collision of sprite ID 1 and sprite ID T[indx]
end

```

实际生成碰撞列表的函数在 UserInterface.CPP 文件中。它使用下面的函数来检测点是否在矩形中（参考 GUIManager.CPP）：

```

bool InRect(int x, int y, const RECT &r)
{
    if((x < r.left) || (x > r.right))
        return false;
    if((y < r.top) || (y > r.bottom))
        return false;
    return true;
}

```

如果 x 坐标小于矩形的左边界或者大于矩形的右边界，那么该点就不在矩形内部。如果 y 坐标小于矩形顶部坐标或者大于底部坐标，那么该点也不在矩形内部。如果函数运行到了最后部分，那么说明该点是在矩形内部。现在我们使用代码清单 14.3 的代码检测子画面（参考 UserInterface.CPP）。

代码清单 14.3 代码检测子画面

```

std::list<unsigned int> CUserInterface::SpriteCollision(unsigned int id)
{
    std::list<unsigned int> retVal;
    CGUISprite *pSprite = (CGUISprite *)FindObject(id);
    if(pSprite && (strcmp(pSprite->GetObjectTypeName(),
        kpSpriteName)==0))
    {
        // the passed GUI object is indeed a sprite
        RECT rOne = pSprite->GetBoundingRect();
        std::map<unsigned int,CGUIObject *>::iterator it =
            m_mapObjects.begin();
        while(it != m_mapObjects.end())
        {
            bool bBoom = false;
            if(((*it).first != id) &&
                strcmp((*it).second->GetObjectTypeName(),
                    kpSpriteName)==0)
            {
                CGUISprite *pTarget = (CGUISprite *) (*it).second;
                RECT rTwo = pTarget->GetBoundingRect();
                // see if the rects collide
                if(InRect(rTwo.right, rTwo.top, rOne))

```



```

    {
        bBoom = true;
    }
    else
    {
        if(InRect(rTwo.right, rTwo.bottom, rOne))
        {
            bBoom = true;
        }
        else
        {
            if(InRect(rTwo.left, rTwo.top, rOne))
            {
                bBoom = true;
            }
            else
            {
                if(InRect(rTwo.left, rTwo.bottom,
                           rOne))
                {
                    bBoom = true;
                }
            }
        }
    }
}
}
if(bBoom)
{
    retVal.push_back((*it).first);
}
it++;
}
}
return retVal;
}

```

至此，界面上的所有控件都被检测了，所有的子画面也进行了碰撞检测。如果子画面的任意一角在传入的子画面内部，该子画面的 ID 会被加入到碰撞列表。所有的控件检测完毕后，返回该列表。

### 14.2.2 HitTest 函数

这个检测和 LuaGlue 函数 GetCollisions() 类似，不过它会检测所有的 GUI 控件而不只是一个点（通常是调用 GetMousePosition() 函数获得的鼠标指针位置）。所有碰到传入点的 GUI 控件都会在返回的列表中。该表和 GetCollisions() 函数返回的列表相似。

### 14.2.3 进一步的说明

要想开发 RTS 游戏那样的单位移动系统，需要用到这里的碰撞检测和之前一章学习的路径寻找示例。路径表顶部的值作为当前目标，到达该位置后，将它移出路径表，然后设定下一个表顶部的值作为新的目标。

## 14.3 碰撞检测

下一个例子是两个子画面间的碰撞检测。在这个例子中，我们可以像前面的例子那样移动坦克，游戏场景同样是随机生成，会有一些随机分布的小屋，代码如下：

```
function HutInit()
    hutCount = 3 --Number of huts in the game
    myHuts = {} --Creates myHuts table
    --Creates table, one entry per potential hut
    for indx = 1, hutCount do
        --Creates a table to hold the data for each hut
        myHuts[indx] = {}
        myHuts[indx].ID = GUI_INGAME + 100 + indx --GUI identification
        number (#)
        myHuts[indx].X = math.random(0,760) --X coordinate (#)
        myHuts[indx].Y = math.random(0,520) --Y coordinate (#)
        if (myHuts[indx].X > 325) and (myHuts[indx].X < 500) then
            myHuts[indx].X = 500
        end
        myHuts[indx].DAMAGE = 0
        CreateItem(myHuts[indx].ID, "Sprite", "hut1.bmp")
        SetItemPosition(myHuts[indx].ID, myHuts[indx].X, myHuts[indx].Y,
            40, 80)
    end
end
```

该表创建了一个随机分布的小屋，函数根据 hutCount 的值进行循环处理，如图 14.4 所示。

在 GUI\_TIMER\_EXPIRED 事件处理器中的处理和前面的例子一样，但是事件处理函数的最后调用了 CollisionCheck() 函数。下面的函数用来检测坦克和小屋之间的碰撞：

```
function CollisionCheck()
    for indx = 1, hutCount do
        if myHuts[indx].DAMAGE ~= 1 then
            hitTable = GetCollisions(myHuts[indx].ID)
            for index, value in ipairs(hitTable) do
                if value == tankID then
                    myHuts[indx].DAMAGE = 1
                end
            end
        end
    end
end
```

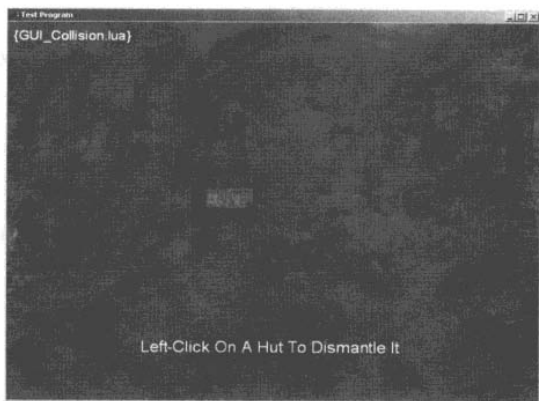


图 14.4 坦克已经碰到了小屋，LuaGlue 函数 CollisionCheck() 检测到了该碰撞

```

        SetTexture(myHuts[indx].ID, 'hut3.bmp')
    end
end
end
end
end

```

在该函数中，我们首先遍历了所有的小屋并且使用它们的 GUI ID 调用 GetCollision() 函数，然后检查返回的列表（使用 ipairs() 遍历）中是否包含坦克。如果包含，则改变小屋的损害状态并且调用 LuaGlue 函数 SetTexture()，小屋的图片变成受损状态。

也可以使用 SetTexture() 函数，它比指纹扫描和爆炸示例中的做法更有效率。

### 14.3.1 LuaGlue 函数 SetTexture

为了让 Lua 代码可以改变子画面显示的图像，SetTexture 方法需要包装成 LuaGlue 函数。在前面的例子中，调用了该函数，改变已经被创建并且载入图片的子画面（参考

GUIManager.CPP);

```
extern "C" int GUI_SetTexture(lua_State *L)
{
    luaL *lua = CGUIManager::GetInstance()->GetLuaContext();
    unsigned int id = (unsigned int) lua->GetNumberArgument(1);
    char *texName = (char *) lua->GetStringArgument(2);
    CUserInterface *pUI = CGUIManager::GetInstance()->GetCurrentUI();
    if(pUI)
    {
        CGUIObject *pObject = pUI->FindObject(id);
        if(pObject)
        {
            pObject->SetTexture(texName, false);
        }
    }
    return 0;
}
```

这个 LuaGlue 函数查找传入的控件并且调用 SetTexture 方法，如代码清单 14.4 所示。  
注意：这里用的是 GUIObject 父类，不是 GUISprite 类，因为要显示的图片数据是父类的一部分（参考 GUIObject.CPP）。

代码清单 14.4 SetTexture 方法

```
bool CGUIObject::SetTexture(char *pTextureName, bool bSetPos)
{
    if(!pTextureName)
        return true;
    float rotx=0.0f, roty=0.0f, rot=0.0f;
    float oldx=0.0f, oldy=0.0f, oldz=0.0f, oldw=0.0f, oldh=0.0f;
    if(m_pTexture)
    {
        m_pTexture->GetRotation(rotx, roty, rot);
        m_pTexture->GetPosition(oldx, oldy, oldz);
        m_pTexture->GetDimensions(oldw, oldh);
        SAFE_DELETE(m_pTexture);
    }
    m_pTexture = new Sprite();
    m_pTexture->SetTexture(pTextureName);
    if(bSetPos)
    {
        m_pTexture->SetPosition(0, 0, 0);
        int w, h;
        DXContext::GetInstance()->GetScreenDimensions(w, h);
        m_pTexture->SetDimensions((float) w, (float) h);
    }
    else
    {

```

```

// reset the location and dims
float w, h;
w = m_rScreen.right - m_rScreen.left;
h = m_rScreen.bottom - m_rScreen.top;
m_pTexture->SetPosition(olddx, oldy, oldz);
m_pTexture->SetDimensions(olddw, oldh);
m_pTexture->SetRotation(rotx, roty, rot);
}
return true;
}

```

因为需要替换成新的图片，因此，如果该对象当前已经有了图片数据，则保存它的旋转和位置信息，并删除图片。接下来生成新的子画面并且载入新的图片文件。如果用户需要在默认位置显示，则设为全屏。如果不需要，则设定原子画面的位置、尺寸和旋转。

### 14.3.2 坦克示例

graphics\_demo 的最后一个示例包含了前面所有示例的功能。玩家可以移动坦克，这次我们添加了一个炮塔，它会跟着坦克旋转，并且可以使用括号键控制。

第一个需要注意的事情是坦克会转向目标。这个行为的处理在 LuaSupport.lua 文件的函数中，如代码清单 14.5 所示。

代码清单 14.5 让坦克朝向目标

```

function FaceTarget(object, curX, curY, curRot, tarX, tarY)
--first, figure out the sides of the triangle:
a = curY - tarY
b = curX - tarX
c = math.sqrt((a*a)+(b*b))
--now, find the angle to the target:
angle = math.acos(b/c)
--now, determine the presence of negatives:
if a > 0 then
    if b > 0 then
        angle = math.pi - angle -- Upper-Left
        print('Upper-Left')
    elseif b == 0 then
        angle = 1.5 * math.pi -- Directly Up
        print('Directly Up')
    else
        angle = math.pi - angle -- Upper-Right
        print('Upper-Right')
    end
end
elseif a == 0 then
    if b > 0 then

```

```
        angle = math.pi          -- Directly Left
        print("Directly Left")
    else
        angle = 2 * math.pi      -- Directly Right
        print("Directly Right")
    end
else
    if b > 0 then
        angle = math.pi - angle  -- Lower-Left
        print("Lower-Left")
    elseif b == 0 then
        angle = .5 * math.pi      -- Directly Down
        print("Directly Down")
    else
        angle = math.pi - angle  -- Lower-Right
        print("Lower-Right")
    end
end
end
if curRot > angle then
    angleDif = curRot - angle
    if angleDif < math.pi then
        angleDif = "ccwise"
    else
        angleDif = "cwise"
    end
else
    angleDif = angle - curRot
    if angleDif < math.pi then
        angleDif = "cwise"
    else
        angleDif = "ccwise"
    end
end
if angleDif == "ccwise" then
    curRot = curRot - .1
    if curRot < .1 then
        curRot = 6.2
    end
else
    curRot = curRot + .1
    if curRot > 6.2 then
        curRot = 0
    end
end
end
if object == "tank" then
    tankRot = curRot
    ItemCommand(tankID, "SetRotation", tankRot)
    if angleDif == "ccwise" then
        turRot = turRot - .1
        if turRot < 0 then
```

```

        turRot = 6.2
    end
else
    turRot = turRot + .1
    if turRot > 6.2 then
        turRot = 0
    end
end
ItemCommand(turID, 'SetRotation', turRot)
end
if (curRot < (angle + .1)) and (curRot > (angle - .1)) then
    orientation = "done"
else
    orientation = "processing"
end
end
end

```

读者会发现这个函数和前面章节中向目标移动的例子相似。这里，我们需要得到目标的角度，然后确定哪种角度和当前的旋转相关。接下来，调用 `ItemCommand()` 函数，使用 `SetRotation` 命令设定增加的角度。因为是先旋转再移动，所以我们使用状态变量 `orientation` 让控制循环知道坦克运行的状态。GUI\_TIMER\_EXPIRED 事件中的游戏循环要更加复杂，每个更新循环都进行了大量处理，参考代码清单 14.6。

代码清单 14.6 GUI\_TIMER\_EXPIRED 事件中的游戏循环

```

if eventCode == GUI_TIMER_EXPIRED then
    if ((tarX == nil) and (tarY == nil)) then
        if orientation == "done" then
            ApproachTarget("tank", tankX, tankY, tarX, tarY,
                tankSpeed)
            hitTable = HitTest(tarX, tarY)
            for index,value in ipairs(hitTable) do
                if value == tankID then
                    tarX = nil
                    tarY = nil
                end
            end
        end
    else
        FaceTarget("tank", tankX, tankY, tankRot, tarX, tarY)
    end
end
for indx = 1,pCount do
    if myProjectiles[indx].ID == nil then
        ApproachTarget(indx, myProjectiles[indx].X,
            myProjectiles[indx].Y, myProjectiles[indx].TARX,
            myProjectiles[indx].TARY, proSpeed)
    end
end
end

```

```
HutHit()
if explosionFrame > 0 then
    if explosionFrame < 10 then
        pic = string.format("%s%d%s", "kaboom_0",
            explosionFrame, ".tga")
    else
        pic = string.format("%s%d%s", "kaboom_",
            explosionFrame, ".tga")
    end
    CreateItem(GUI_INGAME + 105, "Sprite", pic)
    SetItemPosition(GUI_INGAME + 105, myHuts[burningHut].X+1,
        myHuts[burningHut].Y+21, 38, 38)
    explosionFrame = explosionFrame + 1
    if explosionFrame > 13 then
        DeleteItem(GUI_INGAME + 105)
        explosionFrame = 0
    end
end
end
StartTimer(.1)
end
```

在这个大的游戏循环中，首先进行状态检测，看看坦克是在转向还是移动，并且调用对应函数（ApproachTarget() 或者 FaceTarget() 函数）。然后，遍历当前炮弹列表，使用相同的 ApproachTarget() 函数处理炮弹的移动。接着，调用 HitHut() 函数检测炮弹是否击中小屋。最后，处理当前爆炸的动画。游戏截图如图 14.5 所示。



图 14.5 在这个示例中，我们加入了所有的元素，同时处理坦克和炮塔的旋转，并且开火摧毁小屋




这种方式的游戏循环处理在实时游戏中是相当典型的。如果采用之前章节提到的有限状态机，循环中的处理还可以更加简洁。

这个示例的另一个有趣的试验是将旋转和移动放在一起处理，这样坦克就可以一边转向一边移动。为了看起来更真实，还可以根据当前坦克方向和目标物体的位置关系实时调整方向，避免出现只向一个方向旋转的情况。

## 14.4 2D 粒子系统

粒子系统用来模拟很多自然现象，从烟、火焰到云的移动（如图 14.6 所示）。3D 粒子系统在 3D 游戏中已经使用了很多年，但在那之前，2D 游戏中就已经在使用这种系统了。

粒子系统使用了“发射器”的概念，它是空间中的一个点并且包含一定行为的参数。这个发射器会生成粒子，它是一种图形元素并包含了一些行为参数。随着时间的流逝，发射器会生成新的粒子并且移动之前的粒子、改变大小并且遵循一定的轨迹直到生命周期结束，被粒子控制函数从游戏场景中删除。

 2D 粒子系统的简单示例在 CD-ROM 中第 14 章的 2D\_particle 文件夹下。玩家点击屏幕后，该示例会创建一个小的烟雾系统（使用了爆炸示例的一个图形元素）。

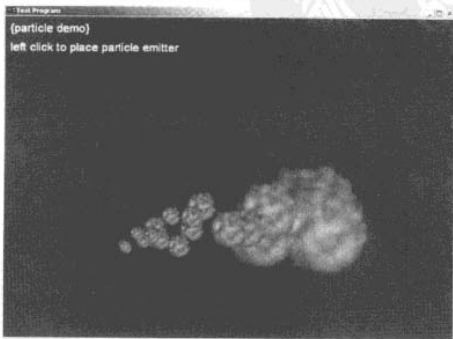


图 14.6 实时冒出黑烟的粒子系统示例

该粒子系统使用下列表的参数定义：

```
partSystem = {}
partSystem.Texture = "kaboom_11.tga"
partSystem.BirthRate = .1
partSystem.AvgLife = 2
partSystem.LifeVariance = 1
partSystem.X = 200
partSystem.XVariance = 10
partSystem.Y = 200
partSystem.YVariance = 10
partSystem.XDrift = 10
partSystem.XDriftVariance = 6
partSystem.YDrift = -5
partSystem.YDriftVariance = 6
partSystem.Scale = 25
partSystem.ScaleVariance = 4
partSystem.ScaleChange = 1.08
partSystem.ScaleChangeVariance = .1
partSystem.BaseID = 1000

partSystem.Life = 20
```

这些参数的原理是，每个粒子创建时使用默认的参数，增加或者减去 0 到差值之间的随机数值，这样就可以生成不同的粒子。

当玩家点击屏幕时，鼠标的 x 和 y 坐标被存入粒子表，初始化粒子系统（清空粒子表，将计数器设为 0），并且设定计时器的初始时间为 timeSlice。当计时器超时，调用粒子控制函数，如代码清单 14.7 所示。

代码清单 14.7 主粒子控制函数

```
function ProcessParticles()
    --increment cumulative counter
    curTime = curTime + timeSlice
    if curTime < partSystem.Life then
        --new particle born
        if EvenDivide(curTime, partSystem.BirthRate) then
            CreateNewParticle()
        end
    end
    removeMe = nil
    --apply particle changes
    for index,value in ipairs(particles) do
        --increment age
        value.Age = value.Age + timeSlice
        --process drift
        myValue = math.random(1, partSystem.XDriftVariance)
        if math.random(1,100) > 50 then
```

```

        value.X = value.X + partSystem.XDrift + myValue
    else
        value.X = value.X + partSystem.XDrift - myValue
    end
    myValue = math.random(1, partSystem.YDriftVariance)
    if math.random(1,100) > 50 then
        value.Y = value.Y + partSystem.YDrift + myValue
    else
        value.Y = value.Y + partSystem.YDrift - myValue
    end
    --process scale change
    myValue = math.random(0, partSystem.ScaleChangeVariance * 10)
    if math.random(1,100) > 50 then
        value.Scale = (value.Scale + (myValue/10)) *
            partSystem.ScaleChange
    else
        value.Scale = (value.Scale - (myValue/10)) *
            partSystem.ScaleChange
    end
    --render particle
    if value.Age < value.Life then
        CreateItem(value.ID, "Sprite", partSystem.Texture)
        SetItemPosition(value.ID, value.X, value.Y, value.Scale,
            value.Scale)
    else
        DeleteItem(value.ID)
        removeMe = index
    end
end
end
--now remove old particles
if removeMe ~= nil then
    table.remove(particles, removeMe)
end
if table.getn(particles) > 0 then
    StartTimer(timeSlice)
end
end
end

```

该函数，在 GUI\_TIMER\_EXPIRED 事件中被调用，首先更新累积计时器，然后检测整个系统周期是否结束。如果没有，则通过模糊的布尔函数检测累积计时器是否能被生成率整除。如果可以，则生成新的粒子，如下所示：

```

function CreateNewParticle()
    tempPart = {}
    tempPart.Age = 0
    idCounter = idCounter + 1
    myValue = math.random(1, partSystem.LifeVariance * 10)
    if math.random(1,100) > 50 then
        tempPart.Life = partSystem.AvgLife + (myValue/10)
    end
end

```

```
else
    tempPart.Life = partSystem.AvgLife - (myValue/10)
end
myValue = math.random(1, partSystem.XVariance)
if math.random(1,100) > 50 then
    tempPart.X = partSystem.X + myValue
else
    tempPart.X = partSystem.X - myValue
end
myValue = math.random(1, partSystem.YVariance)
if math.random(1,100) > 50 then
    tempPart.Y = partSystem.Y + myValue
else
    tempPart.Y = partSystem.Y - myValue
end
myValue = math.random(1, partSystem.ScaleVariance)
if math.random(1,100) > 50 then
    tempPart.Scale = partSystem.Scale + myValue
else
    tempPart.Scale = partSystem.Scale - myValue
end
tempPart.ID = partSystem.BaseID + idCounter
table.insert(particles, tempPart)
print("new particle created")
end
```

该处理首先遍历了用来跟踪所有独立粒子信息的临时表，并且在这里计算每个参数的变化值，然后在粒子表（保存了所有的活跃状态的粒子）的最后插入临时表中生成的新数据。

回到 ProcessParticles() 函数，使用 ipairs() 函数遍历所有活跃状态的粒子，并动态调整它们的值。接着，检测粒子的存活时间，如果大于生命周期则删除该粒子（每个周期删除一个）；否则就在屏幕上显示粒子图像。最后，从粒子主表中删除最后一个被 removeMe 变量标记的粒子。

这个简单的系统很容易被扩展和优化。例如，可以使用快速的粒子系统来代替动画，模拟爆炸，还可以使用长周期的粒子系统来表示燃烧的废墟等。在这个示例中，粒子从左向右移动，也可以让它们向任意方向移动——甚至从一个点扩散或者收缩到中心点。在系统中添加额外的行为也是比较简单的。考虑一下该如何在粒子系统中添加随机旋转子画面的运动。

### 3D 侧边栏

本章主要讨论了示例程序中的子画面图像处理。除此之外，Lua 也是控制 3D 图形的很强大的工具。在我们开发的游戏 Magic Lantern 中，我们创建了一个 3D 系统为开发 3D 游戏提供了很多灵活性。具体的示例如图 14.7 所示。

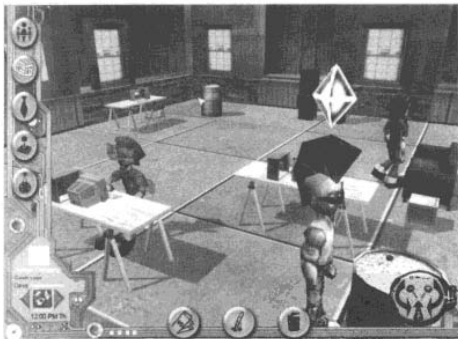


图 14.7 在这个屏幕截图中，可以看到使用 Lua 控制的 3D 游戏，不仅仅是 GUI 控件，还可以管理屏幕上的 3D 模型和动画

我们使用了内部系统来控制、渲染和处理 3D 模型动画，3D 环境的控制部分则使用了 Lua。下列函数是我们的核心 LuaGlue 函数库，它可以让脚本程序员来控制动态和静态的 3D 资源。

- 1) StartCamera(): 在 3D 场景中设定摄像机的位置和视角。
- 2) SlewCamera(): 用适当的加速度和减速度滑动摄像机到新的位置和视角。
- 3) AddEnvironmentObject(): 在场景中添加 3D 模型。还可以获取它的 ID，用来实时改变它的行为。
- 4) AddLight(): 在 3D 场景中添加环境光或者定向光源。
- 5) AttachParticleEffect(): 添加 3D 粒子效果（也定义在 Lua 中）到指定节点或者实体。

6) `AttachChildEntity()`: 在实体的节点上添加 3D 模型。如果父模型被移除, 那么子模型也会被移除。

7) `PositionChildEntity()`: 调整节点的子 3D 模型的位置, 如果既存实体被移除, 那么该子模型也会被移除。

8) `TintEntity()`: 用指定的环境色给实体染色。

9) `SetEntityAnimation()`: 为实体设定动画。

10) `StartAnimSequence()`: 开始实体的动画。

该类表只是列举出了一部分通过 `LuaGlue` 函数控制的 3D 系统的例子。通过这些非常有限的 3D 控制函数, 结合功能强大的脚本语言和事件系统, 我们在多款发行的游戏中已经能够管理和控制动态的 3D 资源了。

## 14.5 本章小结

在本章中, 我们学习了大量的例子, 它们使用 `Lua` 控制 2D 图形显示。尽管目前 3D 已经在游戏世界占据了主导地位, 但是 2D 同样是个不能忽略的游戏开发平台, 它也是个非常好的学习核心游戏开发过程的环境。

通过《Take Away》游戏和前面两章中的例子, 读者应该已经掌握了开发游戏的基础。如果不想从头开发自己的 `LuaGlue` 函数, 《Take Away》游戏以及它的脚本可以给你提供一个稳定的工具库, 用来开发诸如桌面游戏、通过有限状态机控制的具有人工智能敌人的即时战略游戏等。

如果想进一步深入了解并且开发自己的 `LuaGlue` 函数, 可以尝试创建一个能够利用 3D 模型作为界面元素的系统。这个系统可以锻炼用户使用 `Lua` 脚本开发游戏的各个方面的能力。

不管采用哪种方式, 现在你都有了可以让自己的想象力得以实现的工具。在本书的最后一部分, 我们将讨论如何调试 `Lua` 程序、如何使用编译过的脚本, 以及如何发布游戏。

## 第 15 章

# 最后说明

### 本章要点

- 添加音效和音乐
- 使用编辑器
- 调试 Lua 脚本
- 资源管理
- 发布 Lua 代码
- 许可证
- 进一步的说明

通过整本书的学习，我们加深了对 Lua 脚本语言的理解，它可以帮助我们打造令人兴奋和着迷的游戏。我们先介绍了脚本语言，然后学习了 Lua 语言的特性并讲解了示例游戏《Take Away》。不管从现有的代码，还是从你自己开发项目的角度来说，我们都已经基本掌握了 Lua 这个工具的全部内容。

尽管已经花了很多工夫，但是在开始自己的探索之前，还有一些课题需要说明。在本章中，我们会学习如何在游戏中添加音效和音乐（并在 Lua 中控制播放）、调试脚本、发布脚本以及之后可以选择的方向等。

### 15.1 添加音效和音乐

音效和音乐是游戏中很重要的一部分，它可以让游戏体验更完整。有很多复杂的系统可以为游戏添加音效和多音轨音乐，这里只介绍最基本的功能。

首先，来看看如何在界面添加简单的音效。在大部分游戏程序中，当单击按钮时可以听到单击的音效或者声音提示。因为已经捕获了单击事件，所以添加音效非常容易。参考下列《Take Away》游戏中的例子：

```
if id == GUI_MAIN_MENU + 300 then
    PlaySound(1, "button.wav")
    RunGUI("GUI_InGame.lua")
end
```

在这个例子中，参数 1 是设定给音效的 ID 值。在我们的程序中，Lua 音效会在播放结束时自动停止，如果想手动停止，可以使用下面的函数，这里需要音效的 ID 值：

```
StopSound(1)
```

### 15.1.1 LuaGlue 函数 PlaySound

音效是 WAV 格式的播放一次的声音文件。我们使用 DirectSound 来载入和播放音效。当开始播放时，DX9 库载入 DirectSound 系统。负责 DirectSound 内部处理的代码来自于 Microsoft 的示例库。文件是 dsutil.cpp 和 dsutil.h。音效相关的 LuaGlue 函数如下：

```
extern "C" int _PlaySound(lua_State *L)
{
    int argNum = 1;
    cLua *lua = g_pBase->GetLua();

    int soundID = lua->GetNumberArgument(argNum++);
    const char *soundName = lua->GetStringArgument(argNum++);
    if(soundName && (soundID != 0))
    {
        DXContext::GetInstance()->PlaySound(soundID, soundName);
    }
    return 0;
}
```

LuaGlue 函数 PlaySound() 创建了一个对象，很像 GUI ID。可以方便用户之后引用该对象，并且传入要播放的 WAV 文件。当音效播放结束或者调用 LuaGlue 函数 StopSound() 时停止播放。

```
extern "C" int _StopSound(lua_State *L)
{
    int argNum = 1;
    cLua *lua = g_pBase->GetLua();
    int soundID = lua->GetNumberArgument(argNum++);
    if(soundID != 0)
    {
        DXContext::GetInstance()->StopSound(soundID);
    }
    return 0;
}
```



Lua 代码可以调用 LuaGlue 函数 `StopSound()` 来停止正在播放的音效，需要指定 ID。

### 15.1.2 音乐

音乐功能由 ogg/vorbis 库提供。ogg/vorbis 是一个免费的类库，专利权也是共享的，它可以将音乐压缩成类似于其他需要收费的格式。关于格式的更多信息可以参考网站 [www.vorbis.com/](http://www.vorbis.com/)。类库包含在既存的 DirectSound 功能中。Music 相关的 LuaGlue 函数如下：

```
extern "C" int _PlayMusic(lua_State *L)
{
    int argNum = 1;
    cLua *lua = g_pBase->GetLua();
    int musicID = lua->GetNumberArgument(argNum++);
    const char *musicName = lua->GetStringArgument(argNum++);
    if(musicName && (musicID != 0))
    {
        DXContext::GetInstance()->PlayMusic(musicID, musicName);
    }

    return 0;
}
```

`PlayMusic` 类似于 `PlaySound`，区别在于输入为 OGG 文件。参考下面的例子：

```
extern "C" int _StopMusic(lua_State *L)
{
    int argNum = 1;
    cLua *lua = g_pBase->GetLua();
    int musicID = lua->GetNumberArgument(argNum++);
    if(musicID != 0)
    {
        DXContext::GetInstance()->StopMusic(musicID);
    }

    return 0;
}
```

调用 LuaGlue 函数 `StopMusic()` 并传入音乐的 ID，可以停止播放。

每个开发游戏的人都有自己的建议可以提供给别人。这些建议可以归纳成两点：组织好你的所有资源；保证游戏的核心玩法。

## 15.2 使用编辑器

Lua 脚本是文本文件，可以在任何支持文本文件的程序中进行编辑，如 Microsoft Word



### 15.3.1 通用原则

作为使用 Lua 开发游戏的人来说，我们的目标和其他开发者一样：创造有趣、吸引人的游戏体验。我们经常听到“有趣、吸引人”这样的口号，但不常听到却很重要的目标：“开发没有 bug 的游戏。”

bug 是游戏程序代码中的错误。在游戏中，bug 可能会让程序崩溃或者造成游戏数据混乱，让游戏逻辑无法正常运行。开发者的底线是想要交付没有错误的游戏，因为常常崩溃的程序或者错误逻辑导致的数据混乱会完全毁掉游戏体验。一个质量很棒的游戏可以让玩家沉浸其中，而 bug 会毁掉这种体验，让玩家感觉是在操作计算机程序而不是控制飞船在浩瀚的宇宙中穿行。

修复游戏中错误的过程称为“调试”，我们常常听到程序员说“我除掉了一个 bug”。讨论游戏如何调试的时候，游戏开发者分成两个阵营。一方觉得这个事情就像去看牙医，不好玩，但是必须要做；另一方则把这当成探案的机会，仔细地分析每个线索并修复所有的 bug。

调试的第一步是找到 bug。通常，开始测试的时候会发现大量的 bug 藏在代码的各个角落。当发现 bug 时，需要做一些记录。如果是一个人的小作坊，可能就写在一张纸上。如果是大型开发小组的成员，则需要利用一些可以列出所有 bug 状态的 bug 跟踪文档或者系统。

有很多缺陷跟踪系统可以提供给小工作室或者大公司使用。在内部，我们想尽可能让事情简单化，所以我们选择了 Mantis 系统，它是一个免费的基于 Web 的系统（[www.mantisbt.org](http://www.mantisbt.org)），如图 15.2 所示。

当然，在记录 bug 之前还需要做的事情是，运用科学的方法重现问题。如果 bug 不能再现的话是很难修复的，所以这是你的第一步。如果是和多个开发者协同工作（或者很幸运，小组有测试人员），需要养成习惯为每个找到的 bug 填写清楚的“重现步骤”。确定了什么情况下会产生这个 bug 是调试和修复时最好的提示。

当找到 bug，并且确定了再现的条件后（这也是使用 Lua 保存游戏数据的另一个重要原因，你可以在错误发生前创建一个游戏存档，这样就可以很方便地再现错误），接下来应该简化该问题。通常，一个完整的游戏同时会进行大量处理，就算在简单的《Take Away》游戏中，我们也同时跟踪了敌舰、炮弹和目标。

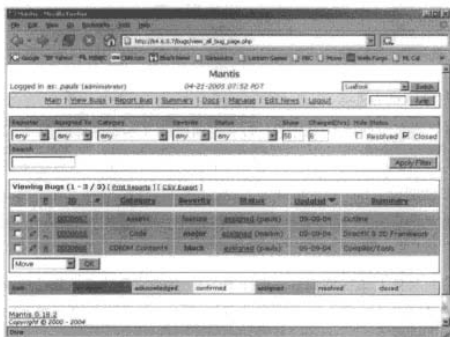


图 15.2 Mantis 障碍追踪系统是免费、稳定的 Web 系统

你的工作是尽可能剔除额外的处理让自己可以专注于问题本身。使用块注释是一个很好的方法，因为这样可以轻易地排除大段的代码。经过一些排查工作后就可以大致确定有问题的代码部分，然后再修复它。当然，你会发现很多拼写错误的 bug（如变量大小写不一致）或者某些明显的逻辑错误。这些错误会占据所有 bug 的 70% ~ 80%，只用花 5% 的时间就可以修复。而那些深藏在脚本中的逻辑 bug 或者数据错误会花掉你大部分的时间和精力。

打好基础后，就可以准备好深入程序内部寻找 bug。最重要的事情之一是你需要了解你在程序流程的哪一步，变量值应该是什么样的。在 IDE 中，可以设定程序断点、单步调试代码并在任意时候查看变量值。但在 Lua 中，我们需要多做一些事情，因为没有 IDE 的支持。下面给出了一些建议可以让你跟踪脚本的运行并且查看变量值。有了这些信息就可以帮助你更容易地找到错误的原因并修复它们。

尽管 Lua 没有提供全功能的调试环境，但是它也有自己的优点。我们可以很容易地修改并且重新执行脚本而不用编译。这种特性可以让我们快速且高效地使用试错法。我们还可以利用运行时命令行的特性（参考下一小节）来实时编辑、查看和改变游戏的数值。

### 15.3.2 调用 DoFile 函数

验证 Lua 脚本是否合法的最简单直接的方法就是调用函数 `dofile()`。在解析和编译脚

本时，如果发现错误，Lua 解释器就会返回找到的第一个错误（非逻辑错误）的相关信息。

### 15.3.3 Lua 错误消息

Lua 命令解释器实时处理 Lua 代码。如果遇到错误（不能处理的代码），就会生成错误消息并发送给 C++ 控制程序。在本书的代码中，我们会将错误消息显示在调试窗口（参考下一小节）以便阅读。Lua 会尽可能地发送该错误的详细信息，如文件名、发生错误的行数（所以说使用能显示行数的编辑器很重要）。Lua 还能显示编译过的脚本发生错误的行数。

这些错误消息是修复错误的很重要的工具，但需要注意的是，发生在第 165 行的错误并不代表错误的实际发生位置，也可能是在其他地方，只是因为你在第 165 行将其他地方定义的变量传给了某个函数而导致了该错误。

### 15.3.4 使用实时调试窗口

读者会发现本书中使用的可执行程序（不包括开始部分介绍的控制台程序）启动时会打开两个窗口——其中一个是调试窗口。它是用户调试 Lua 的最有用的工具，因为它是一个 Lua 命令行解释器，和游戏同时运行在相同的 Lua 环境下。用户可以像之前章节介绍的独立的控制台一样使用它，非常实用。在游戏运行的时候，调试窗口可以访问游戏中的所有变量和函数。如果想查看正在运行的游戏的变量值，可以使用下面的脚本：

```
print(curDistance)
```

用户也可以在脚本中使用 `print()` 函数，函数会在调试窗口中输出结果。如果想确定游戏运行时是否调用了某个函数，可以像下面例子这样添加 `print()` 函数：

```
function HardMath(myValue)
    print("running HardMath()")
    return myValue * 2
end
```

下面是一些使用调试窗口的方法：

#### (1) 移动界面控件，预览界面布局

```
SetItemPosition(GUI_MAIN_MENU + 200, 498, 292, 0, 0)
```

- (2) 可以在任意时候运行 GUI

```
RunGUI("GUI_MainMenu.lua")
```

- (3) 在程序正在运行的时候, 可以重新加载刚刚修改过的脚本

```
dofile("Scripts\\GUI_DamageFire.Lua")
```

- (4) 直接运行游戏函数

```
ClearGUI(GUI_INGAME)
```

- (5) 输出 Lua 数学函数计算结果

```
print(math.cos(math.pi/3))
```

- (6) 查看表数据 (帮助用户确定参数的设定)

```
print(myEnemies[1].X)
```

- (7) 确定函数的返回值

```
var1 = GetState()  
print (var1)
```

- (8) 通过运行事件处理函数, 模拟 GUI 事件

```
InGameEvent(gThrustKey, GUI_KEY_PRESS)
```

(9) 用户可以在调试窗口中复制/粘贴想调试的函数中的代码, 这样就可以单步调试函数。添加一些输出语句就可以逐步地检测代码的运行。

(10) 还可以使用内建的调试函数来检测 Lua 栈中的函数。对于脚本程序员不是很有用, 但在修复那些很复杂的错误时, 可以使用这个函数并在 C++ 程序员 (可以分析内存位置) 的帮助下找到解决问题的方法。

```
print(debug.traceback())
```

### 15.3.5 使用文本框

另一个简单的工具是在主游戏界面添加一些文本框, 默认不显示。设定 GUI\_KEY\_PRESS 事件来控制这些控件的显示, 并输出想查看的值, 如下所示:

```

if eventCode == GUI_KEY_PRESS then
    if id == 118 then -- v key
        if gDebugVisible == false then
            EnableObject(100, 1, 1)
            EnableObject(101, 1, 1)
            ItemCommand(100, "SetString", string.format("%s%d",
                "Current Thurst value: ", curThurst))
            ItemCommand(101, "SetString", string.format("%s%d",
                "Current Rotation value: ", curRot))
            gDebugVisible = true
        else
            EnableObject(100, 1, 1)
            EnableObject(101, 1, 1)
            gDebugVisible = false
        end
    end
end
end

```

有了这个“调试监视器”，可以控制是否显示，并在游戏进行过程中随时查看变量值。

### 15.3.6 使用文件输出

当需要调试高速数据处理或者大数据量处理时，在调试窗口中输出信息的做法不是很有效。

可以选择在 LuaSupport.lua 文件中使用下面的 3 个函数创建一个调试文件输出系统：

```

function StartDebugOutput()
    debugFile = io.open("debug_data.txt", "w")
    if debugFile ~= nil then
        debugFile:write("-- Auto-generated debug report")
        debugFile:write(string.char(10))
        debugFile:write(string.char(10))
        debugFile:write(string.format("%s%s", "-- File created on: ",
            os.date()))
        debugFile:write(string.char(10))
        debugFile:write(string.char(10))
    end
end

```

第一个函数打开文件并输出头信息，在游戏初始化的时候调用这个函数（可以放在 StartGUI.lua 中，在执行 LuaSupport 脚本之后）。处理完毕后需要关闭文件，在退出程序前调用下面的函数：

```

function EndDebugOutput()
    io.close(debugFile)
end

```

最后，使用下面的函数来输出想查看的信息：

```
function DebugMsg(myString)
    if debugFile ~= nil then
        debugFile:write(myString)
        debugFile:write(string.char (10))
    end
end
```

这个函数可以接收字符型参数并整行输出到文件中。注意，可以使用 `string.format()` 函数来格式化字符和数字的输出。该函数可以在程序的任意位置调用，最好在输出信息中添加一些标识符，帮助用户了解输出数值的意义和它的位置，如下所示：

```
DebugMsg(string.format("%s%d", "function: CalculateRange(), curRange: ",
    curRange))
```

这个脚本会输出追踪的函数名和变量值。如果每隔 0.1s 就调用一次该函数，在调试窗口会得到大量的数据，不过这样可以检查文件变量的值在哪里出现了问题。

如果经常更新调试文件，标记程序的进程，那么当出现错误时，如出现程序崩溃或者无限循环的情况，这个工具会非常有用。在程序崩溃时，可以打开调试文件来查看程序是在什么地方发生了错误。在发生无限循环时，可以按下 <Alt + F4> 组合键终止程序，然后同样查看该文件来确定问题发生的位置。

## 15.4 资源管理

资源是指在游戏开发过程中产生的有价值的东西，如源代码、2D 和 3D 美术资源、工具和音乐等。在大型游戏中，需要管理大量的文件。由于这些文件会经常被更新，所以保证游戏中使用最新和最好的资源是一个挑战。当开发团队增加人手的时候，资源也许很快就会失去控制。因此，还需要考虑资源运行时的组织方式。你是直接发布给用户一堆文件，还是通过某种方式来组织文件？文件的命名是怎样的？它们保存在磁盘的什么位置？游戏程序是如何找到这些资源的？

### 15.4.1 资源的组织

资源是游戏的组成部分，包括 Lua 脚本、C++ 头文件和代码文件、界面位图、音效和声音文件等。不仅游戏在运行时需要访问某些资源，开发团队在游戏开发的过程中也需要。



### 源代码控制

在开发过程中，资源需要保持最新状态并且可以被所有开发成员访问。提供这种功能的工具称做源代码控制工具或者配置管理工具。管理资源文件的一个很出色的工具是 CVS (Concurrent Versions System, [www.cvshome.org/](http://www.cvshome.org/))。其他的常用工具还包括微软公司的 SourceSafe (<http://msdn.microsoft.com/vstudio/previous/ssafe/>) 和 Perforce ([www.perforce.com/](http://www.perforce.com/))。CVS 的优点是开源 (免费) 和跨平台，因此在互联网上被广泛使用。如果在 Windows 平台下使用 CVS，可以选择插件工具 TortoiseCVS ([www.tortoisecvs.org/](http://www.tortoisecvs.org/))，它是 Windows 桌面的一个插件，使用很方便。

不管选择什么工具，都能得到很多好处。可以使用这些系统来保存和跟踪所有的资源，如源代码、Lua 脚本、美术原画、3D 模型等。还可以通过从版本库中获取所有最新的资源来确保生成最新的发布版本。这些系统让我们能够有效地组织运行时数据。本书的所有例子都使用了简单的目录结构。游戏根目录的主文件是游戏程序。也有其他的文件，不过程序一般不使用它们。其他的实时资源都保存在相应文件夹下，文件夹有 Scripts、Textures、Sounds 和 Music。如果资源列表中包含 3D 模型，还应该有个 Models 文件夹。用户可以将所有文件打包成压缩文件，如 ZIP 文件，并修改 C++ 代码来读取压缩包。这样可以减少运行时的磁盘读取，并且在文件系统比较弱或者没有文件系统的环境中安装程序。

#### 15.4.2 运行时的文件夹

当然，如果游戏对文件系统不清楚，所有的资源组织都是无效的。示例代码默认使用了前面小节提到的目录结构。C++ 代码在 Scripts 文件夹下读取 Lua 代码、在 Textures 文件夹下读取位图文件、在 Sounds 文件夹下读取声音文件 (.wav 文件)、在 Music 文件夹下读取音乐文件 (.ogg 文件)。

### 15.5 发布 Lua 代码

你可以直接发布开发过程中使用的文本形式的 Lua 代码。这对游戏修改社区 (“modding” community) 的人来说非常“酷”，这种发布形式可以让最终用户拥有最大的访问权限。如果你不想用户修改你的 Lua 代码或者不想别人知道系统的内部秘密，就需要别的方式来发布。最终的文本文件可以通过 Lua 作者提供的标准发布版本中的程序来执行。这个

程序称做 luac.exe, 把 Lua 文本文件“编译”成二进制的文件。发布编译后的 Lua 代码可以作作弊者或者黑客更难知道你的游戏是如何运行的, 采用了什么技术。

### Lua 编译器

luac.exe 的源代码在 lua-5.0.2\src\luac 文件夹的原始发布包里。示例工作空间有一个 Visual C++ 项目来编译程序。程序是一个命令行应用, 可以在命令提示环境 (类似于 DOS 提示符) 接收指令参数。一个典型的编译的命令如下:

```
Luac -o StartGUI.lub StartGUI.lua
```

这一行代码会编译 StartGUI.lua 脚本并在 StartGUI.lub 文件中生成二进制数据。可以使用批处理文件来简化处理。批处理文件中包含一些命令脚本, 文件扩展名为 .bat 便于 Windows 识别。你可以在文件中为每个想编译的 Lua 脚本添加编译指令 (像之前例子那样)。做好批处理文件后, 只需要在 Windows 桌面上双击该文件就可以编译所有的脚本。

### LuaGlue 函数 RunScript

Lua 可以读取不同地方和格式的代码。在本书所有的例子中, Lua 代码都保存在 Scripts 文件夹下。这些脚本是标准的扩展名为 .lua 的文本文件, 或者扩展名为 .lub 的 (使用 luac.exe) 编译后的二进制文件。LuaGlue 函数 RunScript 用来确定代码的位置并执行。RunScript 是 Lua 的 do\_file 函数的替代函数。它在 Scripts 文件夹下寻找传入的脚本文件, 如果没找到就在当前文件夹下继续查找; 如果还是没找到就重复上述步骤查找 .lub 文件。

使用不同的扩展名来区别文本文件和二进制文件是很方便的。文本形式的 Lua 代码使用 .lua 扩展名, 二进制形式的 Lua 代码使用 .lub 扩展名。Lua 解析器并不关心你传给它的是文本文件还是二进制文件, 也不关心扩展名是什么, 但扩展名可以让游戏开发者一眼就分辨出文件的类型。cLua 对象的 RunScript 成员调用帮助函数 findScript() 定位对应的 Lua 脚本文件并执行, 代码如下 (参考 cLua.cpp):

```
static std::string findScript(const char *pFname)
{
    FILE *fTest;
    char drive[_MAX_DRIVE];
    char dir[_MAX_DIR];
    char fname[_MAX_FNAME];
    char ext[_MAX_EXT];
    _splitpath( pFname, drive, dir, fname, ext );
    std::string strTestFile = (std::string) drive + dir +
        "Scripts\\" + fname + ".LUB";
```

```
fTest = fopen(strTestFile.c_str(), "r");
if(fTest == NULL)
{
    strTestFile = (std::string) drive + dir +
        "Scripts\\" + fname + ".LUA";
    fTest = fopen(strTestFile.c_str(), "r");
}
if(fTest == NULL)
{
    strTestFile = (std::string) drive + dir + fname + ".LUB";
    fTest = fopen(strTestFile.c_str(), "r");
}
if(fTest == NULL)
{
    strTestFile = (std::string) drive + dir + fname + ".LUA";
    fTest = fopen(strTestFile.c_str(), "r");
}
if(fTest != NULL)
{
    fclose(fTest);
}
return strTestFile;
}
```

## 15.6 许可证

Lua 是自由软件，可以作为非盈利性质或者盈利性质的软件自由使用，并且不需要得到巴西 PUC-Rio 的许可。Lua 是开源的软件，遵守 MIT 许可条例。许可的具体内容如下：

Copyright © 1994—2004 Tecgraf, PUC-Rio.

被授权人有权利使用、复制、修改、合并、出版发行、散布、再授权及贩售软件及软件的副本。

在软件和软件的所有副本中都必须包含版权声明和许可声明。

该软件不包括任何明示或者暗示的保证，包括是否侵权或者其他的保证。在任何情况下作者或者版权所有人不承担任何因为该软件的使用而产生的索赔、损失或者责任。

包含了该许可证文档（CD-ROM 的 License 文件夹下）的文件应该包含在软件的发布中。Lua 小组还建议，但不是强制要求，使用该软件的网站应该在某处显示蓝色的 Lua 标识。

如果使用 Lua 开发，那么将你的项目添加到 Lua 项目列表（[www.lua.org/uses.html](http://www.lua.org/uses.html)）

是一个很好的主意，可以让每个人都知道你做的事情。

## 15.7 进一步的说明

到目前为止，你已经掌握了 Lua 脚本语言，并且和我们一起学习了第一个游戏的开发，现在是时候讨论一下以后的计划了。想提高开发能力或对 Lua 感兴趣的话，接下来应该做些什么呢？

一个非常好的建议是加入 Lua 社区。可以到 Lua 的官方网站（[www.lua.org](http://www.lua.org)）看看，如阅读文档、开发向导和技术讨论。投入其中，并且提出问题并分享你的收获。尽管 Lua 已经存在了很多年，但在游戏业界中的广泛使用仅仅是刚开始，对你来说，这是一个很好的机会成为技术潮流的一部分。

另一个不错的建议是在本书提供的《Take Away》游戏基础上开发自己的游戏。仔细阅读 LuaGlue 文档就会发现，几乎所有的 LuaGlue 函数都不只是针对《Take Away》这一个游戏的——它们都是开发 Lua 游戏通用的函数。这意味着你已经有了一个简单的游戏开发环境和开发自己的游戏需要的所有工具，就算不会 C++ 编程也没问题。

考虑一下你想要开发的游戏——可以是简单的，但要比《Take Away》再复杂一些。看看手里的工具和示例程序，从碰撞检测、图像旋转，到路径寻找——使用这些工具可以开发什么样的游戏呢？通过学习那种使用 Lua 控制所有的游戏玩法的完整项目，可以让你收获很多 Lua 的开发经验，以及游戏开发的各个方面，如界面设计、游戏设计和音效/音乐编程等。

有了自己的游戏项目后，可以去网络，看看是否有能够合作的项目（[www.garagegames.com](http://www.garagegames.com) 是个很棒的协作项目的网站）已经在使用 Lua 开发或者适合使用 Lua 的实现方式。和协作小组一起工作可以学到团队分工、为脚本编写文档，共同开发一个大家期望的游戏。如果找不到在线的协作项目，那么可以自己发起一个。

如果你已经在开发小组中工作，可以和同事们分享一下 Lua 语言，让大家了解它的特性、优点和缺点，看看是否可以成为未来开发项目中使用的工具。

在内部，我们从来没有打算成为 Lua 的专家，但在开发了自己的功能有限并且难以维护的脚本语言后，我们发现 Lua 可以满足我们对通用脚本语言的需求。我们已经使用 Lua 开发了超过 17 款游戏，从每个项目中都学到了很多（写这本书的过程也收获颇丰）。它是

我们游戏开发工具库中很强大的一个，所以我建议那些有脚本语言需求的开发小组至少都应该调查一下 Lua 可以做什么。

最后，你还可以进一步学习 Lua 语言，学习在应用中如何使用。Lua 并不是为游戏开发的脚本语言（不过越来越适合游戏开发，这也是 Lua 未来发展的核心部分），有越来越多的程序可以使用 Lua 开发。在 Lua 和 Tecgraf 的网站上，可以看到 Lua 在许多领域的应用，甚至是像 PHP 一样作为 HTML 的预处理语言。

Lua 还有其他方面的特性，如在本书中没有提到的字符处理中的模式系统（Patterning System），还有协同例程（Co-routines）。可参考 Lua 文档和其他书籍（可以在 [www.lua.org](http://www.lua.org) 网站上看到 Lua 发明者之一 Roberto Ierusalimsky 所著的“the essential Programming in Lua”一书）来获取更多关于这种小巧而出色的语言的相关信息。

## 15.8 本章小结

至此，我们一起完成了这段美好的学习旅程，现在到了你收获所学、扩展技能，并使用 Lua 开发自己的出色游戏的时候了。我们从概要开始，到编写第一个函数方法，再到复杂迷宫里的路径寻找，在这个过程中学到了很多。

我们已经掌握了在 C++ 项目中集成 Lua 的技术，以及使用脚本语言开发 LuaGlue 函数方法。还学习了如何开发事件驱动的系统，并在此之上创建 GUI 系统。

我们还学习了如何进行简单的游戏设计，以及如何使用 Lua 作为主要工具将设计转化为实现。另外，还了解了很多工具和方法，如人工智能、距离矢量、粒子系统等。同时，还已经掌握了相关技术和工具来开发自己的游戏。

我们希望这是一个享受的过程——我们自己当然是这样。非常高兴能和人家一起分享 Lua 技术，期待能通过这本书逐渐扩展 Lua 社区，因为随着社区的成长，协作开发创新游戏体验的机会也会增加。所以，行动吧，用 Lua 来创造伟大的游戏！