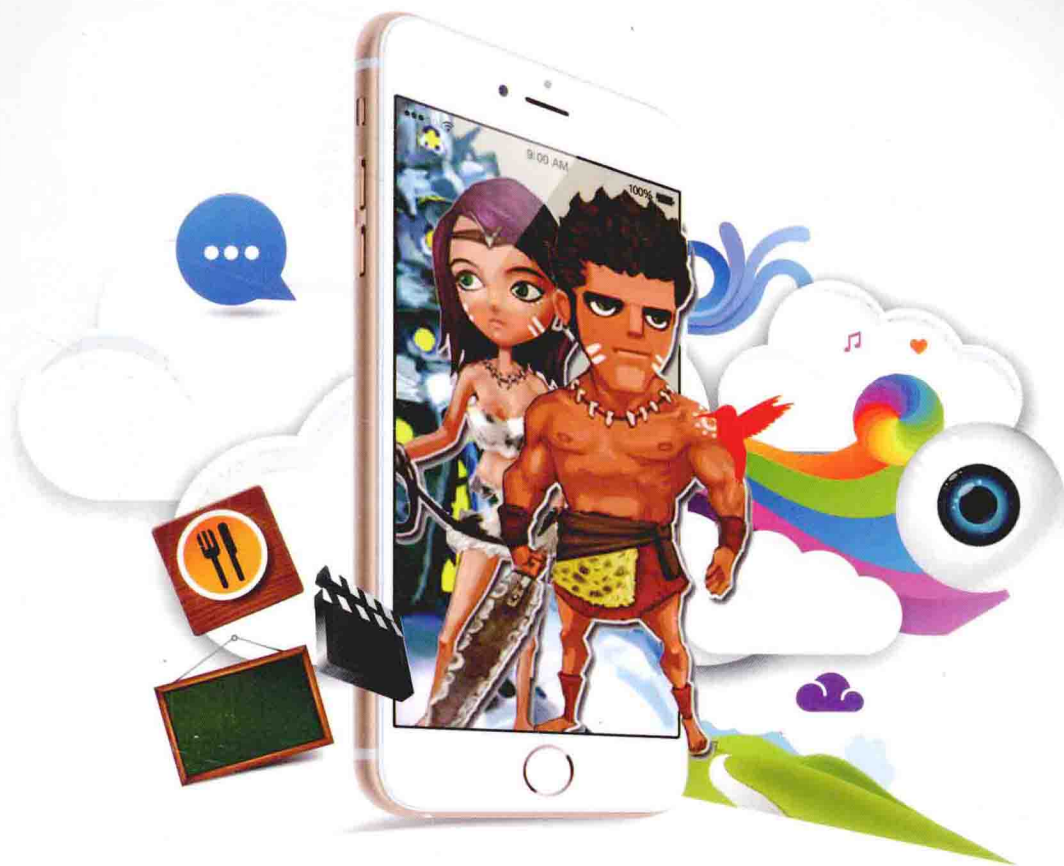


Unity

开发人员的首选



- 亚马逊、京东、当当网络书店排名第一的Unity游戏开发原创精品
- 基于Unity 4.6版本全新改写，Unity3D\2D游戏开发核心技术完全公开
- 热点游戏实例教学，资深Unity开发工程师的经验分享

Unity 3D\2D

手机游戏开发

第2版
金玺曾 编著

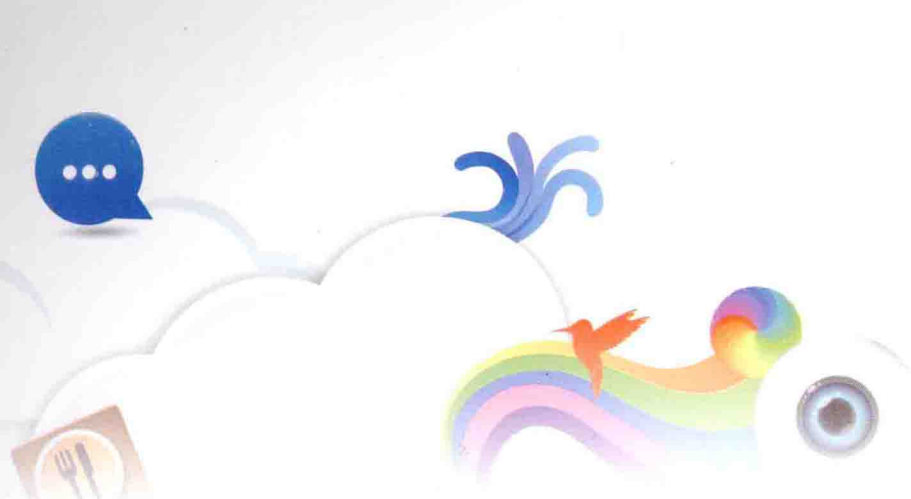


使用C#开发

提供全书实例源代码与素材下载



清华大学出版社



Unity

开发人员的首选

第二版更有料、更深入、更完整，除了仍保留通篇实例的特色外，全书还做了大量的更新和改进，新增了如编辑器扩展，新UI，2D游戏示例，以及使用Python脚本批量编译Android工程等很多内容，几乎重写了读者关注的网络部分。全书内容通俗易懂，高效实用，分享了更多Unity游戏开发中的宝贵经验，相信新版本一定会给您带来更多惊喜。



清华大学出版社数字出版网站

WQBook 书文局泉

www.wqbook.com

ISBN 978-7-302-37990-4



9 787302 379904 >

定价：69.00元

Unity 3D\2D

手机游戏开发

第2版
金玺曾 编著



清华大学出版社
北京

内 容 简 介

本书以实例教学为主线,循序渐进地介绍了 Unity 在游戏开发方面的不同功能。第 1 章,由零开始,引导读者熟悉 Unity 编辑器的各个功能模块,这部分内容对 Unity 程序员、美工和策划都有帮助。第 2~4 章是 3 个不同特色的 3D 游戏实例,让读者对 Unity 游戏开发有一个较全面的认识。第 5 章是一个 2D 游戏实例,全方位地介绍了 Unity 在 2D 游戏方面的应用。第 6 章和第 7 章,重点介绍了 Unity 在网络方面的应用。第 8~10 章介绍了如何将 Unity 游戏移植到网页、iOS 和 Android 平台。另外,本书最后附有 C# 语言的快速教程,帮助缺乏程序开发基础的读者快速入门,同时也包括 Unity 编辑器菜单栏的中英文对照表供读者查阅。

本书还提供了所有实例的源代码与素材文件,供读者上机练习使用,读者可从网上下载本书资源文件。

本书适用于广大游戏开发人员、游戏开发爱好者、软件培训机构,以及计算机专业的学生等。

本书封面贴有清华大学出版社防伪标签,无标签者不得销售。

版权所有,侵权必究。侵权举报电话:010-62782989 13701121933

图书在版编目(CIP)数据

Unity3D\2D 手机游戏开发 / 金玺曾编著. — 2 版. — 北京:清华大学出版社, 2014
ISBN 978-7-302-37990-4

I. ①U… II. ①金… III. ①移动电话机—游戏程序—程序设计 IV. ①TN929.53

中国版本图书馆 CIP 数据核字 (2014) 第 216058 号

责任编辑:王金柱

封面设计:王 翔

责任校对:闫秀华

责任印制:杨 艳

出版发行:清华大学出版社

网 址: <http://www.tup.com.cn>, <http://www.wqbook.com>

地 址:北京清华大学学研大厦 A 座 邮 编:100084

社总机:010-62770175 邮 购:010-62786544

投稿与读者服务:010-62776969, c-service@tup.tsinghua.edu.cn

质 量 反 馈:010-62772015, zhiliang@tup.tsinghua.edu.cn

印 装 者:三河市中晟雅豪印务有限公司

经 销:全国新华书店

开 本:190mm×260mm 印 张:26.5 字 数:678 千字

版 次:2013 年 8 月第 1 版 2014 年 11 月第 2 版 印 次:2014 年 11 月第 1 次印刷

印 数:1~3500

定 价:69.00 元

产品编号:057735-01



本书第一版读者的赞誉

(摘自亚马逊、京东、当当网络书店)

关于Unity3D的书我基本上都是通过当当买的，仅仅就内容来说，看了之后不后悔就是这本吧。

——青竹工作室

本人使用Unity3D进行开发有一年左右的时间了。也买过几本Unity3D的书。我先对本书进行一下评价。1.本书老少皆宜。无论你是初学，还是已经工作一段时间，你都能从本书中或多或少地获得一些技术知识要点，这点很难得。2.本书大体翻了一下，没有糊弄，不像其他的书一样，要不就是通篇的代码，让人看着一个劲打瞌睡，要么就是通篇大图，让人以为在看连环画，总体来说，这本书代码量适中，内容也清晰。3.整本书主要以几个案例来讲解，大体看了一下，基本Unity3D最擅长制作的方面都有涉及，特别是服务器交互那块，如果作者没有非常丰富的经验，估计也写不出来这么全面的书，虽然我没有读完，但我觉得，受益将会匪浅。我作为读者，感谢作者能出版这本书，无私地把一些技术要点奉送，希望今后能出版更多这类技术的书籍，分享技术，造福程序猿！

——霸气的月牛

Unity实战好书。整本书是以游戏实战开发为主线的，实际操作的内容比较多，很好。

——T***T

这本书讲解深入浅出，让人受益匪浅。书中图片挺多的，让人很容易接受。对手机游戏开发讲解算是很到位的，让我在短短的时间内对它有比较全面系统的认识。里面的程序都列出来，很容易重现和在电脑上操作，有很好的指导作用。等读完这本书，我相信我就有能开发一般休闲游戏的能力了。书本纸张有一丁点发黄，不过总体看着还可以，是值得购买的一本书。

——Xuehua549919020

内容很丰富，实例很精辟，很实用。只需要一步一步跟着书做实例就好。看了80页代码还没遇到bug。真是没白买，这两天学到很多。

——amaverick

真正Unity3D游戏开发专家的力作。看过几本Unity3D开发的书，我觉得这本是有经验与最实用的。首先作者通篇通过热点范例教学，如第一人称射击游戏、太空射击游戏、塔防游戏等，其中的内容效果都来自作者自身开发的游戏，如《野人大作战》（大家可以在网络上搜到这款游戏），不像很多书，代码一大堆，浪费了很多版面，本书代码适中，并加入了必须的注解；第二是图片适中，不像有的书，全是大图，占用了许多版面，结果没有效果；第三是经验之作，比如关于网络部分的讲解，绝对是很多书不多涉及的，内容极其有效，这块内容作者真是毫无保留，让游戏开发者少走很多弯路，一看就明白怎么回事了，绝对核心，价值所在，没有开发经验的只是看帮助文件写书的，肯定没有这种效果。学习了，非常受用，一本有价值的好书。想要开发手游的你一定要看这本书。

——minus911

这本书就是U3D的经典书籍，推荐！

——fardece

适合有一定经验的人提高，也适合快速入门。

——小强Robert

作者各方面都能有独到的自己的见解，确实Unity书籍中的独一无二。

——昏晃

非常好的书！例子详实，资源和代码质量都很高，没有错误，没有废话，全是干货！！

——ZKY

作者开发过作品并发布，所以显然比一些自己都不知道没搞过的人强。由浅及深，重要功能大多数都有涉及，比如网络方面，讲得比官方手册清楚，并额外介绍了TCP/IP方式的连接侦听，以及protocol buffer的使用方法。技术点的笔墨刚刚好，不像一些书，啰嗦但却说不清楚。反正挺适合我的……

只是这个书要点是讲开发，指望当手册或者翻查的就不用考虑这本书了，买官方手册去吧……

——faintx

前言

Unity, 也称Unity 3D, 是近几年非常流行的一个3D游戏开发引擎, 它的特点是跨平台能力强, 支持PC、Mac、Linux、网页、iOS、Android等几乎所有的平台, 移植便捷, 3D图形性能出众, 同时也支持2D功能, 为众多游戏开发者所喜爱。在手机平台, Unity几乎成为3D、2D游戏开发的标准工具。

游戏开发是一项复杂的工作, 本书在编写过程中十分注重与实际开发相结合, 全书以实例为基础, 使读者在较短的时间内能快速掌握Unity的各种工具和开发技巧, 并应用于实践当中。

本书在内容结构上, 可以分为几个部分, 第1章是入门部分, 从总体上对Unity编辑器的各个功能模块进行了介绍; 第二部分通过4个实例, 包括太空射击游戏、第一人称射击游戏、塔防游戏和一个2D捕鱼游戏, 使读者对Unity游戏开发有一个较全面的认识, 能有开发一般休闲游戏的能力; 第三部分重点介绍了Unity在HTTP和TCP/IP网络通信方面的应用; 第四部分专门介绍了如何将Unity游戏移植到网页、iOS和Android平台。另外, 本书还附有C#的入门教程和Unity编辑器菜单的中英文对照介绍。

本书为第2版, 下面是各章节的内容概要及与第一版的区别。

第1章介绍了Unity编辑器的各个功能模块, 与第一版相比, 合并了原来的第1章和第5章, 修改了Shader部分的内容。

第2章是一个太空射击游戏教程, 这是一个入门级的教程, 从如何创建一个脚本, 到一个完整的游戏有较为细致的介绍。与第一版相比, 改进了一些细节, 同时增加了内容, 包括用鼠标控制飞机(适用于手机), 创建精准的碰撞模型, 自动创建Prefab和代码优化等内容。

第3章是一个第一人称射击游戏教程, 涉及人工智能寻路、动画控制、摄像机控制等内容。与第一版相比, 改进了UI和小地图部分的内容。

第4章是一个塔防游戏教程, 介绍了创建更为复杂的关卡, 导入由Excel创建的数据等。与第一版本相比, 本章有很多全新的内容, 如动画的使用, 创建自定义的编辑器, 使用NGUI创建UI界面等。

第5章是全新的一章, 介绍了Unity在2D游戏方面的应用, 包括创建Sprite, 动画的播放和一个较为完整的2D捕鱼游戏实例。



第6章介绍了Unity在HTTP网络通信方面的应用，还涉及了PHP和MySQL的基础应用，使Unity游戏可以与Web服务器进行通信，上传得分记录等。与第一版相比，去掉了一些冗余的内容，使用JSON实现网络消息收发。

第7章是一个完整的、基于TCP/IP协议的聊天实例，在这一章中介绍了使用Unity创建聊天客户端，并使用.NET开发环境创建聊天服务器端。与第一版相比，本章的变化很大，增加了入门教程，优化了网络使用的步骤，方法更灵活也更富有技巧。

第8章介绍了如何将Unity游戏转为Unity网页游戏和Flash游戏，重点介绍了面向不同网页平台的一些专门技术要点，如何编写Flash插件等。与第一版相比，增加了AssetBundle部分的内容。

第9章介绍了如何将Unity游戏移植到iOS平台，从如何申请iOS平台开发资格，到测试、发布iOS游戏都有详细的介绍，最后着重介绍了如何在Xcode开发环境中开发Unity插件，实现Game Center和内消费功能。

第10章介绍了如何将Unity游戏移植到Android平台，并详细介绍了几种为Unity开发Android插件的方法。与第一版相比，改进了插件制作流程，增加了使用脚本自动编译游戏的教程。

本书还提供了所有实例的源代码与素材文件，供读者上机练习使用，读者可从网上下载本书资源文件。下载地址：<http://pan.baidu.com/s/1c0kvOMs>

本书的读者主要是游戏开发程序员和Unity爱好者，同时也适合游戏策划和美工使用。

对于本书的完成，要特别感谢王金柱编辑给予的帮助和指导，感谢我的妻子在深夜帮助我校对书稿，还要感谢我的儿子给我莫大的精神支持。

金玺曾

2014年7月13日



目 录

第 1 章	Unity 入门.....	1
1.1	Unity 简介.....	2
1.2	运行 Unity.....	3
1.2.1	Unity 的版本.....	3
1.2.2	安装 Unity.....	3
1.2.3	在线激活 Unity.....	4
1.2.4	运行示例工程.....	4
1.2.5	安装 Visual Studio.....	6
1.3	创建一个“Hello World”程序.....	7
1.4	调试程序.....	10
1.4.1	显示 Log.....	10
1.4.2	设置断点.....	10
1.5	光照.....	11
1.5.1	光源类型.....	11
1.5.2	环境光与雾.....	12
1.5.3	Lightmapping.....	13
1.5.4	Light Probe.....	15
1.6	Terrain.....	17
1.7	Skybox.....	20
1.8	粒子.....	22
1.9	物理.....	25
1.10	自定义 Shader.....	28
1.10.1	自定义字体.....	28
1.10.2	创建 Shader.....	29
1.11	游戏资源.....	32
1.11.1	贴图.....	32
1.11.2	3ds Max 静态模型导出.....	33
1.11.3	3ds Max 动画导出.....	35
1.11.4	Maya 模型导出.....	35
1.12	Unity 动画系统.....	36

1.13 美术资源的优化	40
小结	40
第2章 太空射击游戏	41
2.1 浅谈游戏开发	42
2.1.1 开始一个游戏项目	42
2.1.2 阶段性成果	42
2.1.3 策划	42
2.1.4 编写脚本	43
2.1.5 美术	43
2.1.6 QA 测试	43
2.1.7 发布游戏	44
2.2 游戏策划	44
2.2.1 游戏介绍	44
2.2.2 游戏 UI	44
2.2.3 主角	44
2.2.4 游戏操作	45
2.2.5 敌人	45
2.3 导入美术资源	45
2.4 创建场景	46
2.4.1 创建火星背景	46
2.4.2 设置摄像机和灯光	49
2.5 创建主角	51
2.5.1 创建脚本	51
2.5.2 控制飞船移动	52
2.5.3 创建子弹	54
2.5.4 创建子弹 Prefab	55
2.5.5 发射子弹	56
2.6 创建敌人	58
2.7 物理碰撞	59
2.7.1 添加碰撞体	59
2.7.2 触发碰撞	60
2.8 高级敌人	63
2.8.1 创建敌人	63
2.8.2 发射子弹	64
2.9 声音与特效	67
2.10 敌人生成器	69
2.11 游戏管理器	71
2.12 标题界面	76

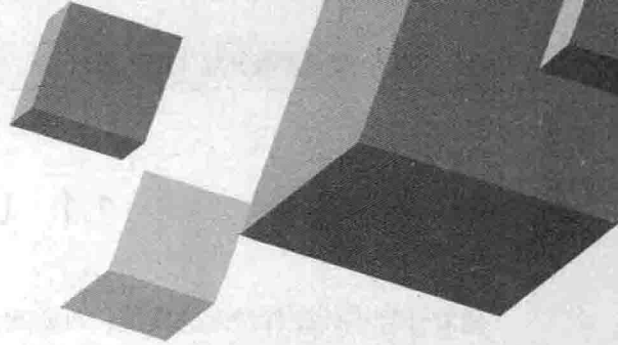
2.13 用鼠标控制主角	77
2.14 精确的碰撞检测	80
2.15 自动创建 Prefab	81
2.16 发布游戏	83
2.17 代码优化	86
小结	88
第3章 第一人称射击游戏	89
3.1 策划	90
3.1.1 游戏介绍	90
3.1.2 UI 界面	90
3.1.3 主角	90
3.1.4 敌人	90
3.2 游戏场景	90
3.3 主角	91
3.3.1 角色控制器	91
3.3.2 摄像机	94
3.3.3 武器	95
3.4 敌人	96
3.4.1 寻路	96
3.4.2 设置动画	99
3.4.3 行为	101
3.5 UI 界面	104
3.6 交互	109
3.6.1 主角的射击	109
3.6.2 敌人的进攻与死亡	112
3.7 出生点	114
3.8 小地图	117
小结	120
第4章 塔防游戏	121
4.1 策划	122
4.1.1 场景	122
4.1.2 摄像机	122
4.1.3 胜负判定	122
4.1.4 敌人	122
4.1.5 防守单位	122
4.1.6 UI 界面	123
4.2 游戏场景	123

4.3	制作 UI	125
4.4	创建游戏管理器	128
4.5	摄像机	132
4.6	路点	135
4.7	敌人	139
4.8	敌人生成器	143
4.8.1	在 Excel 中设置敌人	144
4.8.2	创建敌人生成器	147
4.8.3	遍历敌人	152
4.9	防守单位	152
4.10	生命条	162
4.11	地图编辑器	164
4.11.1	创建一个自定义窗口	165
4.11.2	绘制多边形	166
4.11.3	在 Inspector 窗口添加自定义 UI 控件	167
	小结	169
第 5 章	2D 游戏	170
5.1	Unity 2D 系统简介	171
5.2	创建 Sprite	171
5.2.1	使用 SpriteEditor 创建 Sprite	171
5.2.2	使用 SpritePacker 创建 Sprite	174
5.2.3	图层排序	175
5.3	动画制作	176
5.3.1	序列帧动画	176
5.3.2	使用脚本实现序列帧动画	177
5.3.3	骨骼动画	179
5.4	2D 物理	181
5.5	捕鱼游戏	185
5.5.1	游戏玩法	185
5.5.2	准备 2D 资源	186
5.5.3	创建鱼	187
5.5.4	创建鱼群生成器	189
5.5.5	创建大炮	191
5.5.6	物理碰撞	193
5.6	2D 材质	195
5.6.1	修改默认材质	195
5.6.2	自定义的黑白效果材质	196
	小结	196

第 6 章 与 Web 服务器的交互	197
6.1 建立服务器	198
6.1.1 安装 Apache	198
6.1.2 安装 MySQL	200
6.1.3 安装 PHP	202
6.1.4 显示 PHP 信息	204
6.1.5 调试 PHP 代码	205
6.2 WWW 基本应用	207
6.2.1 HTTP 协议	207
6.2.2 GET 请求	208
6.2.3 POST 请求	209
6.2.4 上传下载图片	211
6.2.5 下载声音文件	213
6.3 分数排行榜	213
6.3.1 创建数据库	214
6.3.2 创建 PHP 脚本	215
6.3.3 上传下载分数	217
6.4 MD5 验证	220
小结	221
第 7 章 基于 TCP/IP 协议的聊天实例	222
7.1 TCP/IP 开发简介	223
7.2 一个简单的网络程序	225
7.3 网络引擎	227
7.3.1 数据包	228
7.3.2 逻辑处理	234
7.3.3 网络功能	237
7.3.4 创建聊天协议	243
7.4 聊天客户端	244
7.5 聊天服务器端	247
7.6 Protobuf 简介	250
小结	255
第 8 章 用 Unity 创建网页游戏	256
8.1 网页游戏简介	257
8.2 Unity Web 游戏	257
8.2.1 Streaming 关卡	257
8.2.2 上传游戏到 Kongregate	260
8.2.3 与网页通信	263

8.2.4	在网页上记录积分	265
8.2.5	自定义网页模板	265
8.2.6	自定义启动画面	270
8.3	Flash 游戏	271
8.3.1	软件安装	271
8.3.2	导出 Flash 游戏	271
8.3.3	调试 Flash 游戏	272
8.3.4	从 Flash 工程读取 Unity 导出的 Flash 游戏	273
8.3.5	在 Unity 内调用 AS3 代码	279
8.3.6	Flash 版本的太空射击游戏	282
8.4	AssetBundle	285
8.4.1	打包资源	286
8.4.2	下载资源	287
8.4.3	批量打包 AssetBundle	290
8.4.4	安全策略	292
小结		293
第9章	将 Unity 游戏移植到 iOS 平台	294
9.1	iOS 简介	295
9.2	软件安装	295
9.3	申请开发权限	295
9.4	设置 iOS 开发环境	296
9.5	测试 iOS 游戏	299
9.6	发布 iOS 游戏	301
9.6.1	申请发布证书	302
9.6.2	创建新应用	302
9.6.3	提交审核	303
9.7	集成 Game Center	305
9.7.1	Xcode 到 Unity	305
9.7.2	设置高分榜和成就	311
9.7.3	实现 Game Center 功能	312
9.8	集成内消费系统	322
9.8.1	设置内消费	322
9.8.2	实现内消费	323
9.9	本地存储位置	330
小结		330
第10章	将 Unity 游戏移植到 Android 平台	331
10.1	Android 简介	332

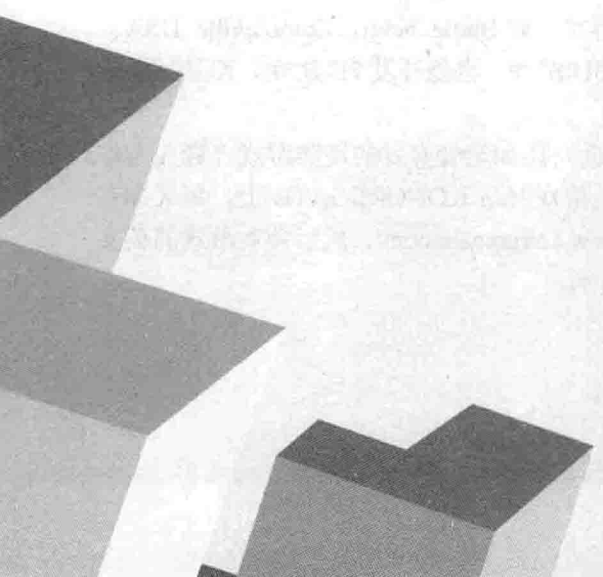
10.2 软件安装	332
10.3 运行 Android 游戏	333
10.3.1 设置 Android 手机	333
10.3.2 安装驱动程序	334
10.3.3 设置 Android 游戏工程	335
10.3.4 测试 Android 游戏	338
10.3.5 发布 Android 游戏	338
10.4 触屏操作	340
10.5 从 eclipse 到 Unity	345
10.5.1 创建 jar 文件	345
10.5.2 导入 jar 到 Unity	349
10.5.3 使用 LogCat 查看 Log	350
10.6 从 Unity 到 Eclipse	351
10.6.1 导出 eclipse 工程	351
10.6.2 设置导出的 eclipse 工程	353
10.6.3 发布程序	355
10.7 使用脚本编译游戏	356
10.7.1 使用脚本输出 Android 工程	356
10.7.2 使用脚本编译 Android 工程	360
小结	361
附录 A C#语言	362
A.1 C#基础	363
A.2 面向对象编程	373
A.3 字符串	381
A.4 数组和排序	383
A.5 I/O 操作	388
A.6 委托	393
小结	397
附录 B 特殊文件夹	398
附录 C Unity 编辑器菜单中英文对照	400



第 1 章

Unity 入门

本章主要介绍如何安装并运行 Unity，编写最简单的脚本，在 Unity 中创建光源、地形、粒子等美术资源，以及如何从 3ds Max、Maya 中导出美术资源等。



1.1 Unity 简介

随着计算机软硬件技术的发展,对游戏品质的要求越来越高,技术上的研发也变得越来越困难,一些有实力的公司开发了自己的技术,推出了不同的游戏引擎,使开发者可以重用已有的技术,集中精力在游戏的逻辑和设计上,很大程度提高了生产效率。

Unity (也称 Unity 3D) 是一套包括图形、声音、物理等功能的游戏引擎,提供了一个强大的关卡编辑器,支持大部分主流 3D 软件格式,使用 C#或 JavaScript 等高级语言实现脚本功能,使开发者无需了解底层复杂的技术,快速地开发出具有高性能、高品质的游戏产品。

Unity 是跨平台的 3D 游戏引擎,支持的平台包括 PC、Mac、Linux、Web、iOS、Android、Xbox360、Play Station3 等大部分主流游戏平台,还可以将游戏直接导出为 Flash 格式放到网页上,如图 1-1 所示。很多时候,可以选择在 PC 平台开发和测试,然后只需要很少的改动,即可将游戏移植到其他平台。

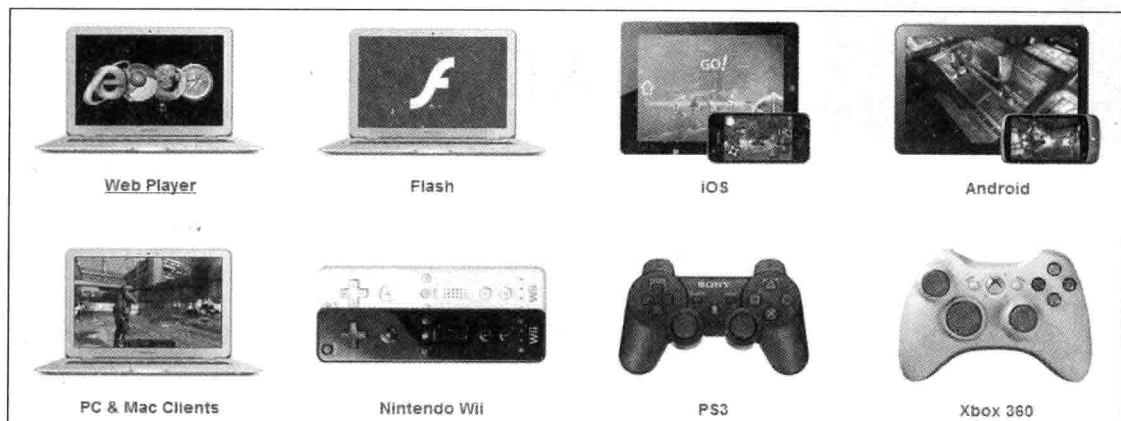


图 1-1 Unity 支持的游戏平台

Unity 是一个成熟的游戏引擎,其能力是毋庸置疑的,随着 iOS、Android 手机的大量普及和 3D 网页游戏的兴起,Unity 因其强大的功能、良好的可移植性,在手机和网页平台得到了广泛地应用和传播。

在手机移动市场可以找到大量使用 Unity 开发的游戏,如 Battle heart、Zombievillle USA、AirAttack HD、Samurai II: Dojo 等。Unity 不但能开发 3D 游戏,也能开发 2D 游戏,而且画面效果出众。

使用 Unity 开发的游戏可以方便地发布到网页上面,比如笔者开发的塔防游戏“野人大作战”,除了在 iPhone 和 iPad 上发布,也发布到了在线游戏网站 KONGREGATE 上,每天都有很多玩家在玩这个游戏。如果有兴趣,可访问 <http://www.kongregate.com/>,然后搜索游戏的英文名 Wild Defense,就可以玩到这个游戏了,如图 1-2 所示。

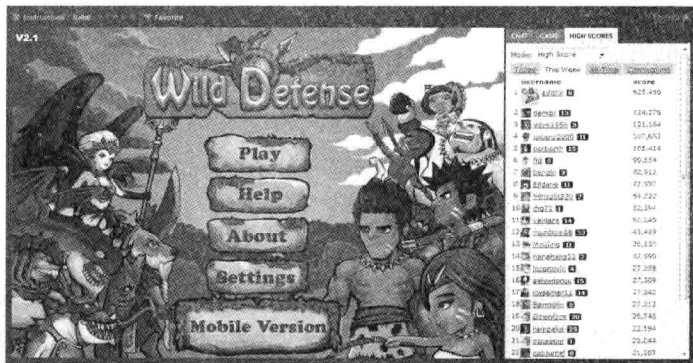


图 1-2 网页版野人大作战

Unity 内置了 Raknet 网络引擎，如果只是需要制作玩家与玩家之间直接连接的网络游戏，使用这些功能就够了。

经常会有人问，Unity 能做 MMO 游戏吗？答案是肯定的。但 Unity 只能完成客户端部分，服务器部分通常是由 C++、Java，或者 PHP 之类的技术完成的。

在 Unity 中，可以使用标准的 Socket 功能完成长连接网络通信，也可以使用 Unity 提供的 WWW 功能提交 HTTP 网络请求。

Asset Store 是 Unity 官方在线商店的名称，里面主要是出售 Unity 的插件或美术资源。严格来说这并不能算是 Unity 的一个功能，但它确实已经成为 Unity 的一部分，很多 Unity 的插件已经成为 Unity 开发中的必要组成部分，如 NGUI 等。

1.2 运行 Unity

本节介绍如何安装和运行 Unity 示例工程及安装 Visual Studio，为使用 Unity 开发游戏创建一个工程环境。

1.2.1 Unity 的版本

Unity 提供了基础版和专业版两个版本，专业版相对于基础版有更多的高级功能，比如实时阴影效果、屏幕特效等。

在 PC 和 Mac 平台上，基础版是完全免费的，但针对 Flash、iOS、Android 等平台则要收取授权费用。到 Unity 的在线商店 <https://store.unity3d.com/> 可以了解到详细的价格情况。

1.2.2 安装 Unity

在 Unity 的官方网站 <http://unity3d.com/unity/download/> 可以免费下载 Unity，包括 PC 版和 Mac 版，这是完整的安装包，包括专业版和针对 Flash、iOS、Android 等平台的全部功能。下载完 Unity 后，运行安装程序，按提示安装即可。

1.2.3 在线激活 Unity

第一次运行 Unity 会提示在 Enter your serial number 处输入 Unity 的序列号, 如图 1-3 所示。对于没有序列号的用户, 可以选择【Activate the free version of Unity】使用免费的基础版, 如图 1-4 所示。

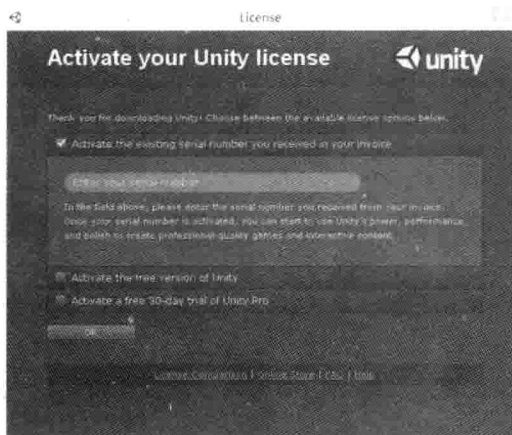


图 1-3 输入序列号

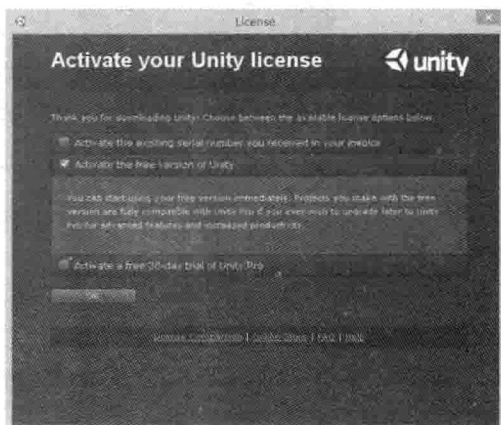


图 1-4 选择基础版

在正式使用免费版之前, 需要输入 Unity 的用户名和密码。如果还没有账户, 选择【Create Account】即可创建一个新的账户, 如图 1-5 所示。

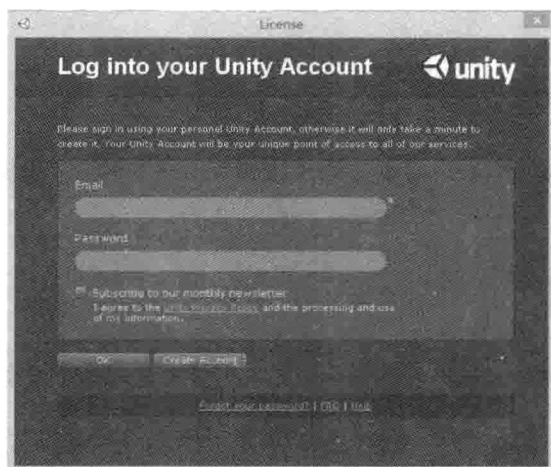


图 1-5 登录账户

1.2.4 运行示例工程

第一次启动 Unity, 会打开 Unity 的工程向导对话框, 如图 1-6 所示。选择【Open Other】, 浏览路径到示例工程存放的位置, 默认存放在“我的文档”目录下的 Unity Projects\4-0_AngryBots 中, 当在浏览器中看到 Assets 这个文件夹时, 选择【选择文件夹】即可打开示例工程。



图 1-6 工程向导对话框

打开示例工程后，会看到 Unity 的编辑器界面，几乎所有的开发工作都要在这个编辑器中完成。在编辑器的视图中，开发者可以像操作 3D 图形软件一样设置游戏场景，编辑游戏对象，所有的游戏资源（包括模型、贴图、脚本等）都需要导入到编辑器中才能使用。Unity 编辑器中包括 Hierarchy、Project、Inspector、Scene、Game 几个窗口，如图 1-7 所示。

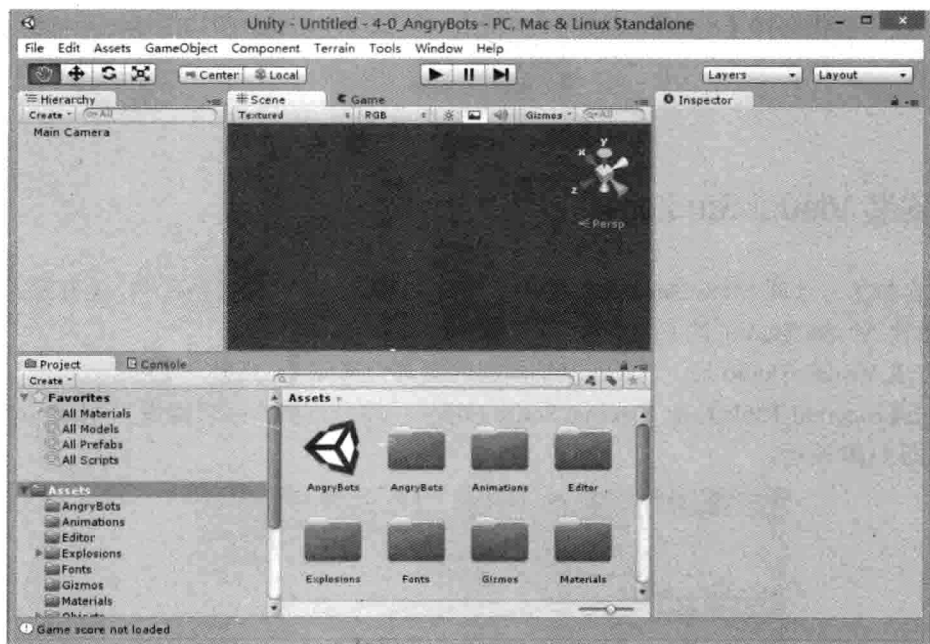


图 1-7 编辑器界面

在菜单栏选择【File】→【Open Scene】，选择 AngryBots.unity，打开示例的关卡文件。现在，在 Scene 窗口中可以看到游戏的场景，如图 1-8 所示。

关卡即是 Unity 中的 Scene 场景文件，Unity 最后打包的游戏所包含的内容，即是所有与关卡相关联的资源 Resources 路径下的资源。



图 1-8 示例游戏场景

为了能够在 Scene 窗口中浏览场景, 有一些改变场景视图角度的快捷键需要知道:

- 按鼠标中键平移视图。
- 按鼠标左键+Alt 键旋转视图。
- 按鼠标右键+Alt 键或滑动鼠标滑轮推拉视图。
- 按 F 键可以快速锁定选中的目标。

最后, 在工具栏选择【播放】(如图 1-9 所示), 即可在 Game 窗口中运行示例游戏。



图 1-9 选择【播放】

1.2.5 安装 Visual Studio

Unity 自带了一个叫 MonoDeveloper 的工具, 用来书写代码。如果是在 PC 上开发, 建议同时也安装微软 Visual Studio 的 C# 部分, 可代替 MonoDeveloper。

安装完成 Visual Studio 后, 在 Unity 编辑器的菜单栏选择【Edit】→【Preferences】打开设置窗口, 选择【External Tools】, 在 External Script Editor 中将外部脚本编辑器设为 Microsoft Visual Studio, 如图 1-10 所示。

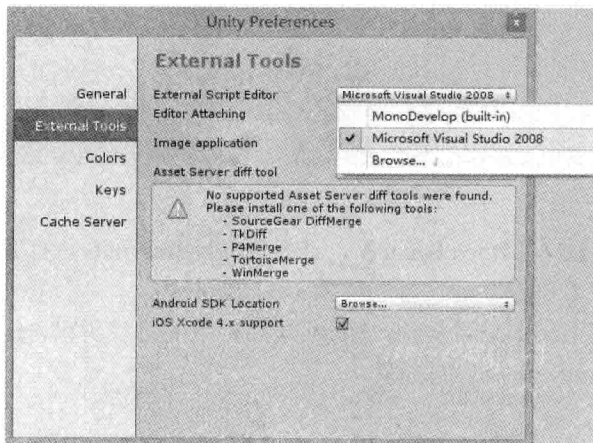


图 1-10 设置 Visual Studio

1.3 创建一个“Hello World”程序

Unity 的底层是使用 C++ 开发，对于 Unity 的开发者，只允许使用脚本进行开发，降低了开发难度。Unity 支持的脚本包括 C#、Javascript 和 Boo，比较流行的是 C# 和 JavaScript，本书的范例全部都是使用 C#。

所有的 Unity 脚本都继承自 MonoBehaviour 这个类，它没有 Main 函数入口，采用了事件触发的模式，提供了比如 Start、Update 这样的函数，根据不同的事件响应不同的函数。具体内容可以搜索帮助文件中 MonoBehaviour 的相关内容。

接下来，我们将使用 Unity 完成一个“Hello World”程序，并将它创建为一个标准的 Windows 可执行程序，步骤如下：

- 步骤 01** 启动 Unity，在菜单栏选择【File】→【New Project】打开工程向导窗口，选择【Browse】确定新工程的保存路径，然后选择【Create】创建一个新的工程，如图 1-11 所示。

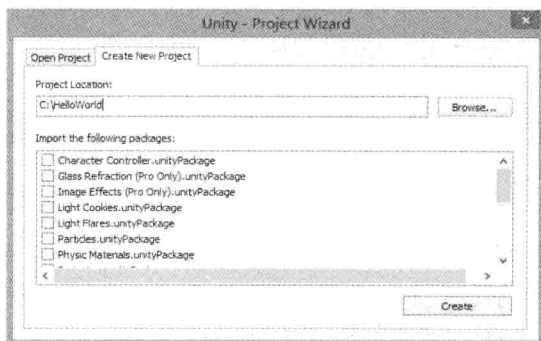


图 1-11 创建新工程

提示 Unity 只允许在空的文件夹内创建新工程。

- 步骤 02** 创建新工程后，在 Project 窗口选择【Assets】，然后按右键选择【Create】→【C# Script】创建一个新的 C# 脚本，将脚本命名为 Hello World，如图 1-12 所示。

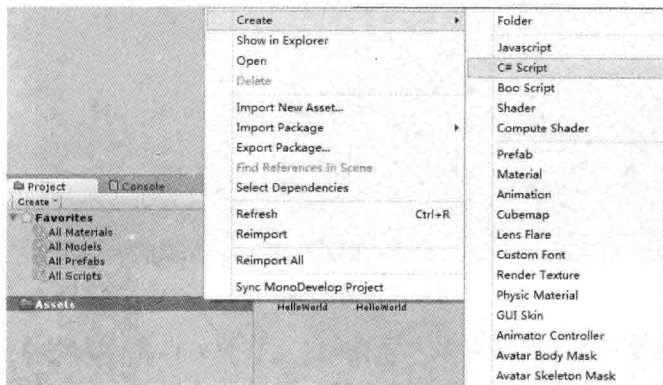


图 1-12 创建新脚本

步骤 03 双击 Hello World 脚本文件将其打开, 会发现里面已经被自动填充了一些基本代码。我们这里的任务是希望在屏幕上显示 “Hello World” 几个字, 添加如下代码:

```
using UnityEngine;
using System.Collections;

public class HelloWorld : MonoBehaviour {
    // Use this for initialization
    void Start () {
    }
    // Update is called once per frame
    void Update () {
    }
    void OnGUI()
    {
        // 改变字符的大小
        GUI.skin.label.fontSize = 100;
        // 输出文字
        GUI.Label( new Rect( 10, 10, Screen.width, Screen.height ), "Hello World" );
    }
}
```

Hello World 是一个类的名称, 它继承自 MonoBehaviour, 注意这个类在 Unity 中不能使用关键字 new 创建, 因此也没有构造函数。Start 函数从字面即可看出是开始的意思, 可以简单把它理解为一个初始化函数, Update 函数也是一个事件触发函数, 它在每一帧都会被执行, OnGUI 函数专门用来绘制 UI 界面。

步骤 04 回到 Unity, 在 Hierarchy 窗口内选择 Main Camera 选中摄像机, 在菜单栏选择【Component】→【Scripts】→【Hello World】将脚本指定给摄像机。运行游戏, 即可看到 “Hello World” 几个字显示在屏幕上, 如图 1-13 所示。



图 1-13 运行游戏

步骤 05 现在, 需要将前面的劳动成果保存一下。在菜单栏选择【File】→【Save Scene As】将当前关卡保存在 Asset 目录内, 命名为 Hello World。可以看到, 我们一共创建了两个文件: 一个是脚本文件; 另一个是关卡文件, 如图 1-14 所示。



图 1-14 脚本和关卡文件

步骤 06 确定前面保存的关卡处于打开状态，在菜单栏选择【File】→【Build Settings】打开 Build Settings 窗口，如图 1-15 所示。选择【Add Current】将当前关卡添加到 Scenes In Build 下面的框中（也可以直接将关卡文件拖入框中），只有将关卡添加到这里，它才能被集成到最后创建的游戏。

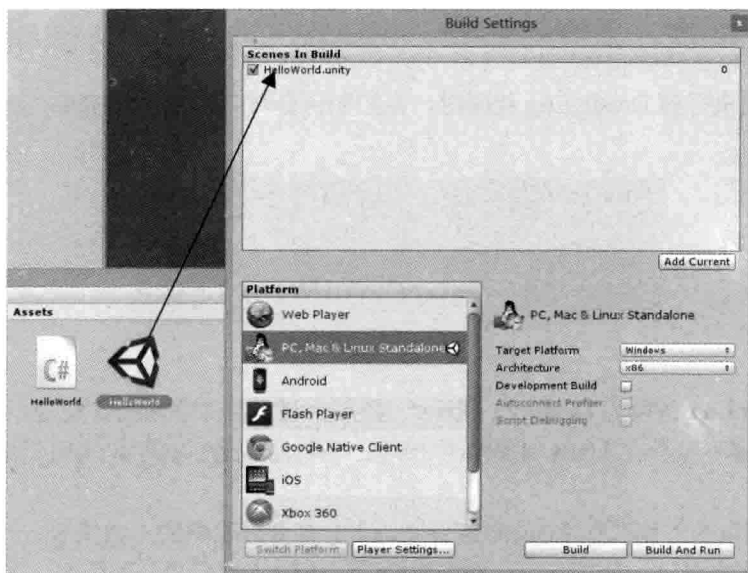


图 1-15 添加关卡

步骤 07 最后还需要进行很多设置，这里我们将只设置游戏的名字。在 Build Settings 窗口选择【PlayerSettings】，在 Inspector 窗口将 Product Name 设为“Hello World”，如图 1-16 所示。



图 1-16 设置游戏名字

步骤 08 在 Build Settings 窗口选择【Build】，然后选择保存路径即可将程序编译成独立运行的标准 Windows 程序。

本节的示例文件保存在资源文件目录\chapter01_Hello World 中。

1.4 调试程序

游戏开发中出现错误是正常的, 调试程序发现错误非常重要。本节将介绍调试程序的几种常用方式。

1.4.1 显示 Log

在 Unity 编辑器下方有一个 Console 窗口, 用来显示控制台信息, 如果程序出现错误, 这里会用红色的字体显示出错误的位置和原因, 我们也可以在程序中添加输出到控制台的代码来显示一些调试结果:

```
Debug.Log("Hello, world");
```

运行程序, 当执行到 Debug.Log 代码时, 在控制台会对应显示出 “Hello, world” 信息, 如图 1-17 所示。

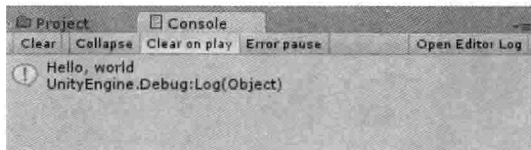


图 1-17 显示调试信息

如果将 Debug.Log 替换为 Debug.LogError, 控制台显示的文字将呈红色显示。

这些 Log 内容不仅会在 Unity 编辑器中出现, 当将游戏运行在手机上时仍然可以通过工具实时查看。

在 Console 窗口的右侧选择【Open Editor Log】会打开编辑器的 Log 文档, 一个比较实用的功能是, 当创建出游戏后, 在这个 Log 文档中会显示出游戏的资源分配情况, 如图 1-18 所示。

Textures	587.1 kb	11.2%
Meshes	42.0 kb	0.8%
Animations	1.8 kb	0.0%
Sounds	2.1 mb	40.8%
Shaders	0.0 kb	0.0%
Other Assets	5.0 kb	0.1%
Levels	11.1 kb	0.2%
Scripts	11.6 kb	0.2%
Included DLLs	2.4 mb	46.3%
File headers	16.4 kb	0.3%
Complete size	5.1 mb	100.0%

图 1-18 Log 中保存的信息

1.4.2 设置断点

Unity 自带的 Mono 脚本编辑器提供了断点调试功能, 使用的方法如下:

步骤 01 使用 MonoDevelop 作为默认脚本编辑器。在 Project 窗口中右键选择【Sync MonoDevelop Project】, 打开 MonoDevelop 编辑器。

步骤 02 在代码中按 F9 键设置断点。

步骤 03 在 MonoDevelop 的菜单栏选择 **【Run】→【Attach to Process】**，选择 Unity Editor 作为调试对象，然后选择 **【Attach】**，如图 1-19 所示。

步骤 04 在 Unity 编辑器中运行游戏，当运行到断点时游戏会自动暂停，这时可以在 MonoDevelop 中查看调试信息，如图 1-20 所示。之后，需要按 F5 键越过当前断点才能继续执行后面的代码。

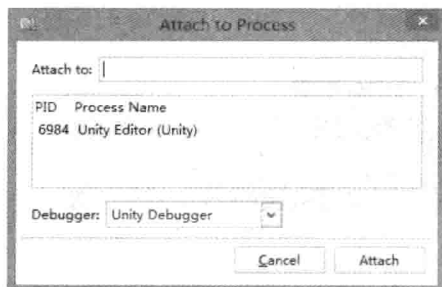


图 1-19 选择 Unity Editor 作为调试对象

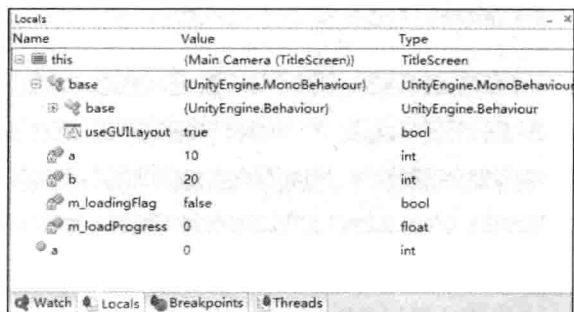


图 1-20 利用断点调试

1.5 光照

在 3D 游戏中，光是一项重要的组成元素，一个漂亮的 3D 场景如果没有光影效果的烘托将暗淡无光。因此，Unity 提供了多种光影解决方案，下面将逐一介绍。

1.5.1 光源类型

Unity 一共提供了 4 种光源，不同光源的主要区别在于照明的范围不同。在 Unity 菜单栏选择 **【GameObject】→【Create Other】**，即可创建这些灯光，包括 Directional Light（方向光）、Point Light（点光源）、Spot Light（聚光灯）、Area Light（范围光）。

Directional Light 就像是一个太阳，光线会从一个方向照亮整个场景，在 Forward Rendering 模式下，只有方向光可以显示实时阴影，如图 1-21（左）所示。

Point Light 像室内的灯泡，从一个点向周围发射光线，光线逐渐衰减，如图 1-21（右）所示。

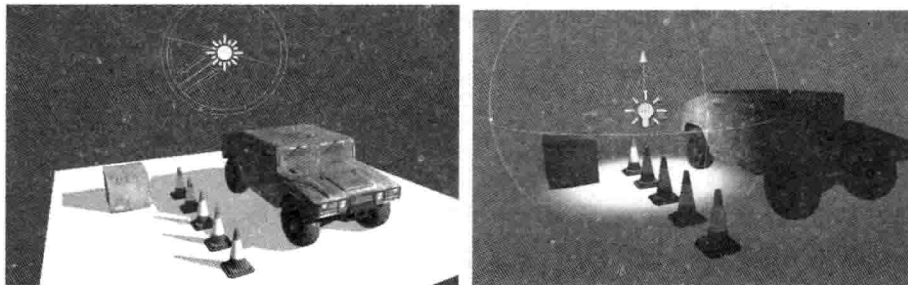


图 1-21 方向光和点光源

Spot Light 就像是舞台上的聚光灯, 当需要光线按某个方向照射, 并有一定范围限制, 那就考虑使用 Spot Light, 如图 1-22 (左) 所示。

Area Light 只有在 Pro 版中才能使用, 它通过一个矩形范围向一个方向发射光线, 只能被用来烘焙 Lightmap, 如图 1-22 (右) 所示。

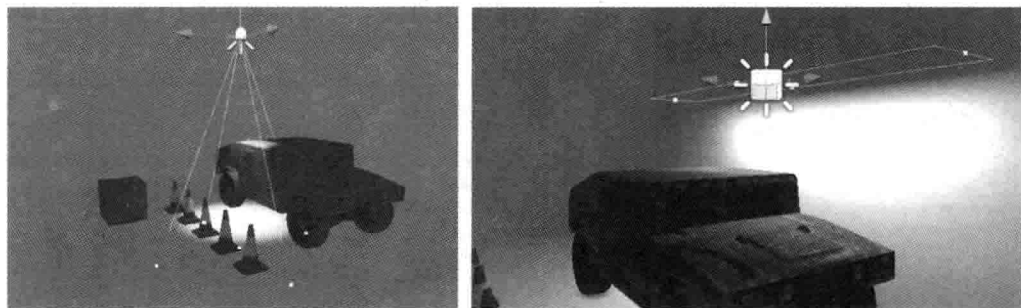


图 1-22 聚光灯和范围灯

这几种光源都可以在 Inspector 窗口进行设置, 如图 1-23 所示。

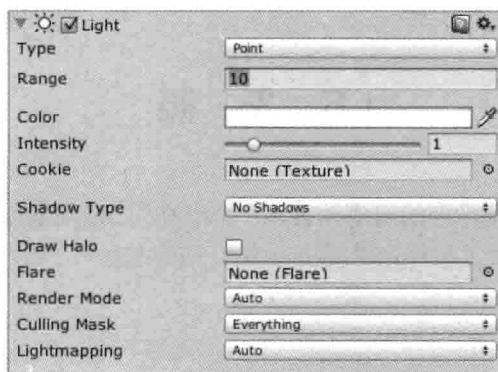


图 1-23 设置光源

其中 Range 决定光的影响范围, Color 决定光的颜色, Intensity 决定光的亮度, Shadow Type 决定是否使用阴影。

Render Mode 是一个重要的选项, 当设为 Important 时其渲染将达到像素质量, 设为 Not Important 则总是一个顶点光, 但可以获得更好的性能。

如果希望光线只用来照明场景中的部分模型, 可通过设置 Culling Mask 控制其影响对象。

Lightmapping 可设为 RealtimeOnly 或 BakedOnly, 这将使光源仅能用于实时照明或烘焙 Lightmap。

1.5.2 环境光与雾

环境光是 Unity 提供的一种特殊光源, 它没有范围和方向的概念, 会整体地改变场景亮度。环境光在场景中是一直存在的, 在菜单栏选择【Edit】→【Render Settings】, 然后在 Inspector 窗口调节 Ambient Light 的颜色即决定了环境光的亮度和颜色。

在这里选中 Fog 还可以开启雾效功能, 通过设置 Fog Color 改变雾的颜色, 设置 Fog Density

改变雾的强度，如图 1-24 所示。

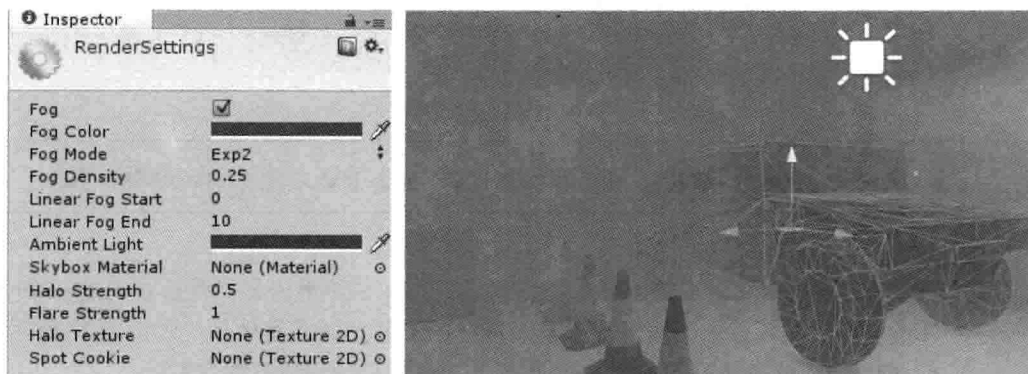


图 1-24 设置环境光和雾



提示

雾效功能对性能会造成一定影响，在硬件性能较差的平台要谨慎使用这个功能。

1.5.3 Lightmapping

当游戏场景包含了大量的多边形时，实时光源和阴影对游戏性能的影响会很大。这时更适用使用 Lightmapping 技术，将光线效果预渲染成贴图使用到多边形上模拟光影效果。这种方式不用担心光源数量和阴影对性能带来的开销，即使是使用基础版的 Unity，仍然可以使用这种方式获得高质量的光影效果。

下面将通过一个简单的示例来说明如何使用 Lightmapping 技术：

步骤 01 打开资源文件目录下的 chapter01_Lightmap 工程，打开 lightmap_start.unity 场景文件。在这个场景中，预先提供了一些用于测试的模型和预设的光源，如图 1-25 所示。

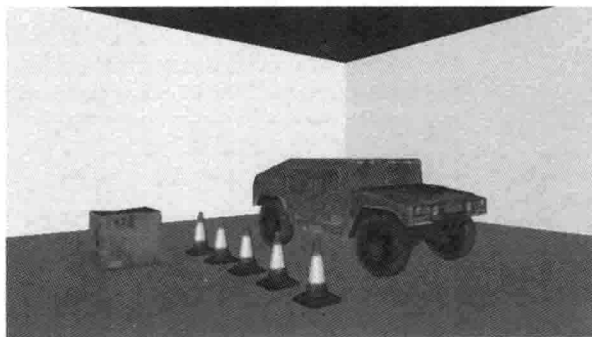


图 1-25 用于测试灯光贴图的场景

步骤 02 选择场景中的模型，在 Inspector 窗口右上方选中 Static 选项，这表示该模型是一个静态多边形模型（在游戏中不会动的模型），只有选中了这个选项的模型才能参与 Lightmapping 计算，如图 1-26 所示。

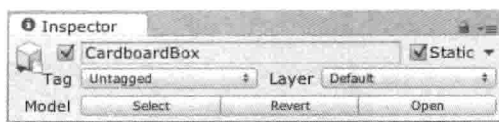


图 1-26 设为静态

提示 使用灯光贴图的模型必须有第二套 UV，这套 UV 不能有 UV 重叠的地方。

步骤 03 创建一个 Spot Light 置于场景上方向下照射，并使用阴影，然后再创建一个 Area Light 置于场景当中，适当地调节光源参数使其达到满意效果，如图 1-27 所示。

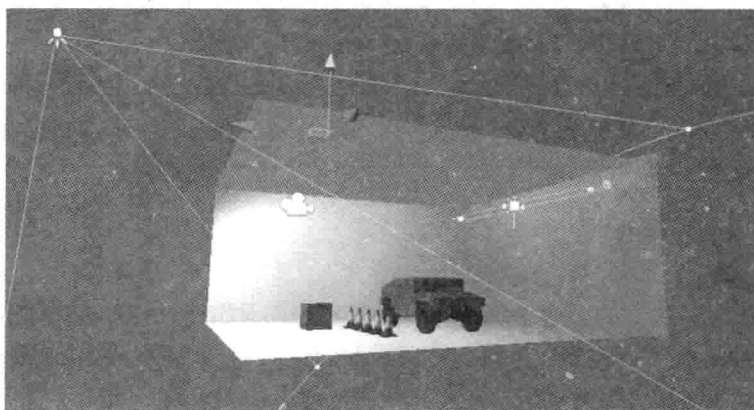


图 1-27 设置光源

步骤 04 在菜单栏选择【Window】→【Lightmapping】打开 Lightmapping 窗口。选择 Bake 进行烘焙设置，如图 1-28 所示。

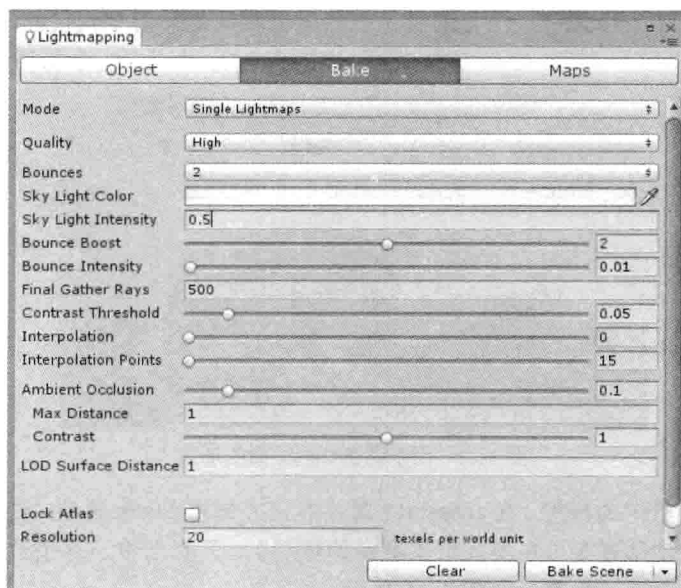


图 1-28 烘焙设置

- Quality 决定烘焙的质量, High 为最高, 但计算时间会比较长。
- Bounces 决定光子的计算级别, 当其大于 0 时, 可以通过设置 Final Gather Rays 等选项获得高质量的光能传递运算效果。
- Ambient Occlusion 会使模型交界处产生阴影过渡效果。
- Resolution 的数值决定场景中的每个单位面积将使用多少个贴图像素, 这个值越大, 贴图的尺寸越大, 计算时间也越长。

步骤 05 选择 Bake Scene 开始烘焙计算, 烘焙完成后, 最后的效果可参考图 1-29 所示。

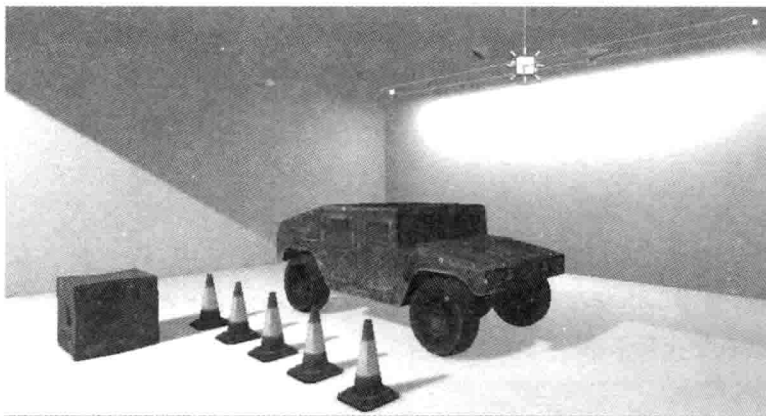


图 1-29 使用 Lightmap 的场景

在烘焙过程中, 编辑器右下角会出现一个烘焙计算进度条, 如果想中途放弃, 按 Esc 键即可。所有光影贴图都会自动存放在当前场景的存放路径中, 选择 Clear 即可清除贴图。本节最终的示例场景保存在当前工程的 lightmap_finish.unity 场景文件中。

1.5.4 Light Probe

Lightmapping 技术虽然可以使静态场景拥有无以伦比的光影效果, 但它无法影响到场景中动态的模型, 这可能会导致出现这样的情况, 场景中的静态模型看起来非常真实, 但那些运动中的模型, 比如角色, 相比较会显得非常不真实并与场景中的光线无法融合在一起。

Unity 提供了一个叫 Light Probe 的功能可以很好地解决上述问题, Light Probe 可以将场景中的光影信息存储在不同的 Probe 中, 我们需要手工摆放这些 Probe 的位置, 光影信息越是丰富的地方就越需要更多的 Probe, 它们将对场景中 Lightmap 的光影信息进行采样, 场景中运动的模型将参考这些 Probe 的位置模拟出与静态场景类似的光影效果。

下面我们将继续前面完成的工程, 为场景添加 Light Probe 功能。

步骤 01 在 Unity 中打开前面完成的 lightmap_finish.unity 场景, 将其另存为一个新的场景。

步骤 02 选择汽车模型, 取消选择 Static 选项, 使其成为一个动态模型, 它将不再受 Lightmap 影响。然后在它的 Mesh Renderer 组件中选 Use Light Probes, 使其接收来自 Light Probe 的光影信息, 如图 1-30 所示。

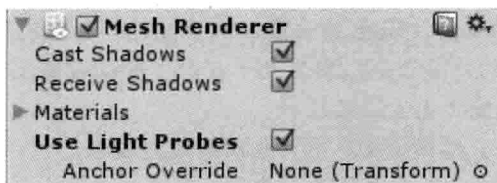


图 1-30 使用 Light Probe

- 步骤 03** 在场景中选择地面 (或任意模型), 然后在菜单栏选择【Component】→【Rendering】→【Light Probe Group】为其添加一个 Light Probe Group 组件。
- 步骤 04** 在 Inspector 窗口找到 Light Probe Group 组件, 选择【Add Probe】创建一个 Probe, 如图 1-31 所示。



图 1-31 创建 Probe

- 步骤 05** Probe 就像一个球体, 按 Ctrl+D 组合键可以快速地复制它, 将其摆放在场景中形成网络采集光影信息, 如图 1-32 所示。

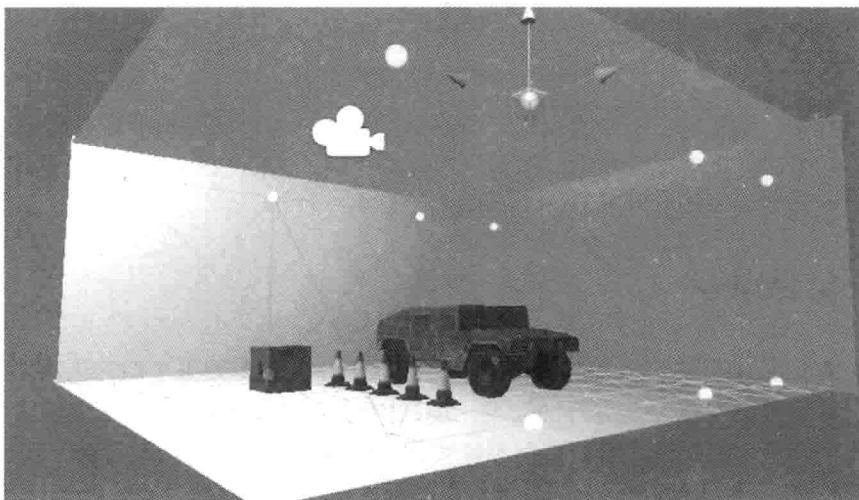


图 1-32 摆放 Probe

**提示**

每个 Light Probe 都会消耗一定的内存, 在实际项目中, 应当根据场景中不同位置的重要程度和光影变化程度决定 Light Probe 的分布密度。

- 步骤 06** 为了使 Light Probe 效果更明显, 可以根据需要改变光源设置, 使其对比度更强烈一些。
- 步骤 07** 重新为场景烘焙 Lightmap, 然后在 Lightmapping 窗口的 Maps 中找到 Light Probes, 为其指定当前场景中的 Light Probe, 如图 1-33 所示。

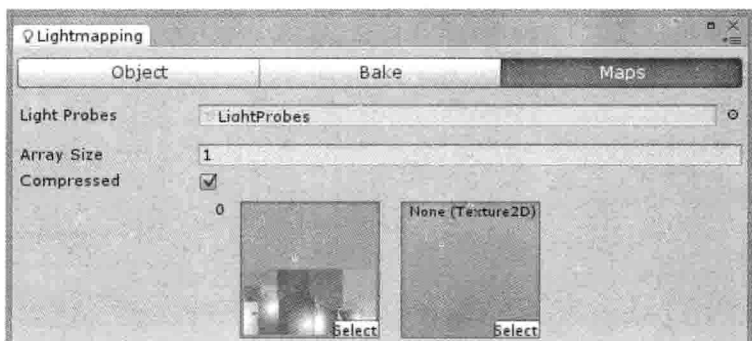


图 1-33 指定 Light Probe

确定将场景中的光源全部设置为 **BakedOnly**, 这样场景中的汽车模型将不会受到任何实时光照影响, 尽管如此, 将其移动到场景中的不同位置, 它还是会像使用了 **Lightmap** 一样产生与场景近似的光影效果, 如图 1-34 所示。

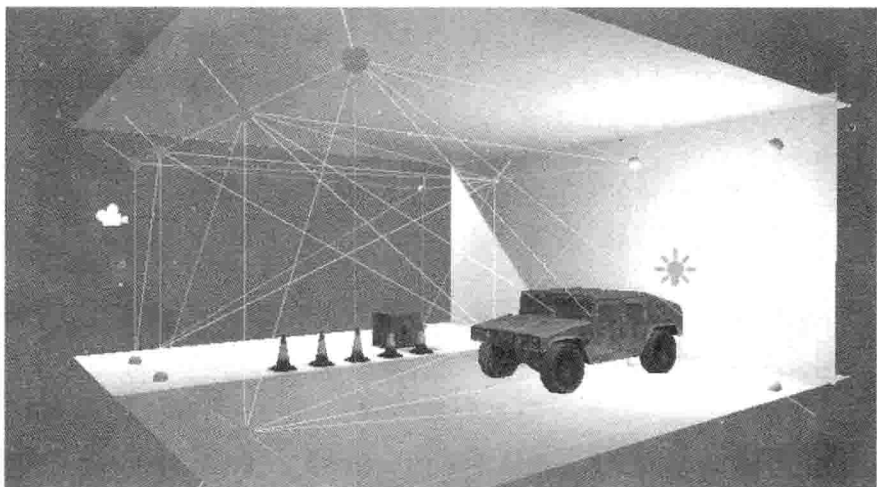


图 1-34 Light Probe 效果

本节最终的示例场景保存在当前工程的 `lightmap_lightprobe.unity` 场景文件中。

1.6 Terrain

Terrain (地形) 是 Unity 提供的一个地形系统, 主要用来表现庞大的室外地形, 特别适合表现自然的环境。下面将通过一个示例说明 **Terrain** 的应用。

- 步骤 01** 新建一个 Unity 工程, 在 Project 窗口单击右键, 选择 **【Import Package】** → **【Terrain Assets】**, 然后选择 Import 导入 Unity 提供的 Terrain 模型、贴图素材, 我们将使用 Unity 提供的这些素材完成一个地形效果。
- 步骤 02** 在菜单栏选择 **【GameObject】** → **【Create Other】** → **【Terrain】** 创建一个基本的 Terrain。然后在 Inspector 窗口选择 Terrain 设置选项, 如图 1-35 所示。默认的 Terrain 非常大, 这里将 Terrain Width 和 Terrain Length 设为 500, 缩小 Terrain 尺寸。将 Heightmap

Resolution 设为 257, 降低其精度。



图 1-35 设置 Terrain

步骤 03 在 Inspector 窗口选择 Raise 工具, 设置 Brush Size 改变笔刷大小, Opacity 改变笔刷力度, 然后在 Terrain 上绘制拉起表面, 若同时按 Shift 键则会将表面压下。使用 Paint Height 工具可以直接绘制指定高度。使用 Smooth Height 工具可以光滑 Terrain 表面, 如图 1-36 所示。

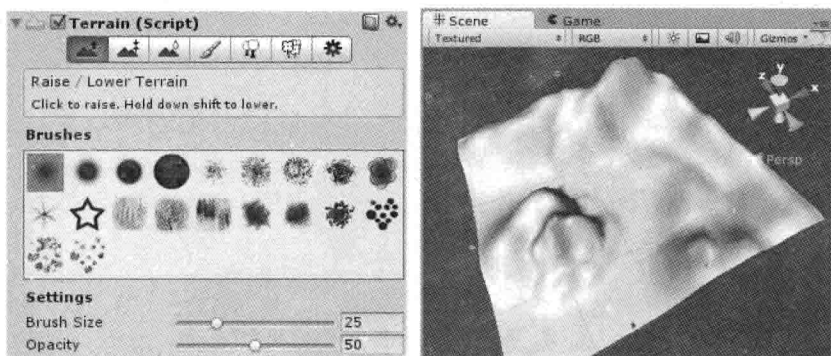


图 1-36 改变地形

步骤 04 选择 Paint Texture 工具, 选择【Edit Textures】→【Add Texture】打开编辑窗口, 为 Terrain 添加贴图, 注意在 Tile Size 中设置贴图的尺寸。这个操作可以反复执行多次添加多张贴图。最后在 Textures 中选择需要的贴图, 将贴图画到 Terrain 上面, 如图 1-37 所示。

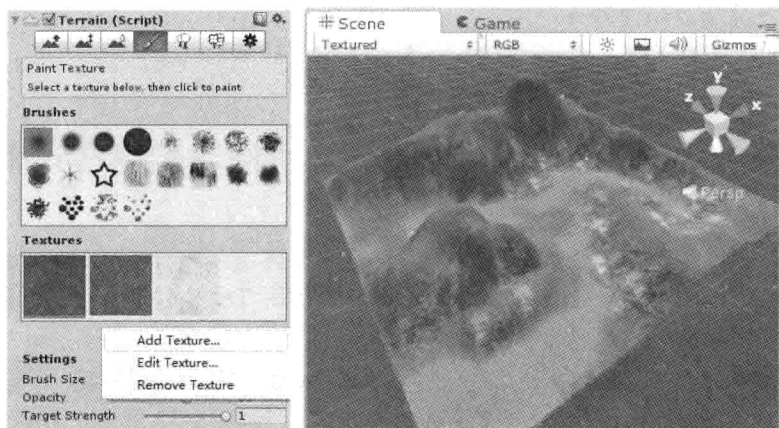


图 1-37 绘制贴图

步骤 05 选择 Place Trees 工具，选择【Edit Trees】→【Add Tree】添加树模型，这个操作可以执行多次加入多个模型。在 Trees 中选择需要的模型，将其绘制到 Terrain 上面，如图 1-38 所示。

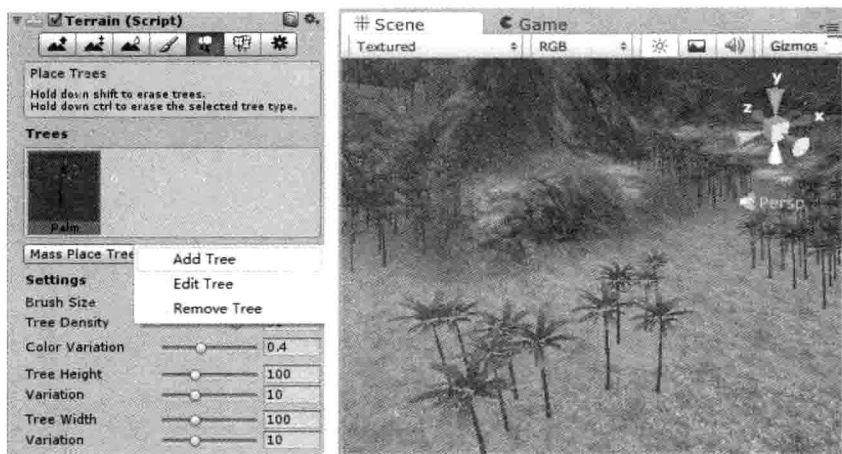


图 1-38 绘制树

步骤 06 选择 Paint Details 工具，选择【Edit Details】→【Add Grass Texture】添加草贴图（贴图一定要有 Alpha），选择【Add Detail Mesh】添加细节模型（如石头等），这个操作可以反复执行多次。最后在 Details 中选择需要的草贴图或细节模型，将其绘制到 Terrain 上面，如图 1-39 所示。

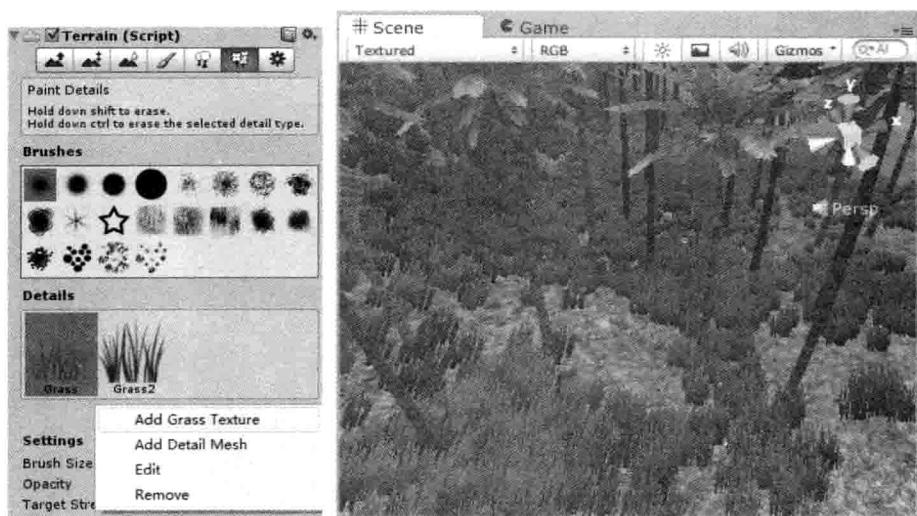


图 1-39 绘制草

步骤 07 Terrain 同普通的模型一样，可以使用 Lightmapping 模拟光影效果，添加了 Lightmap 的 Terrain 将会看上去更加生动，最终效果如图 1-40 所示。



图 1-40 最终效果

本节的示例工程文件保存在资源文件目录 chapter01_Terrain 中。

1.7 Skybox

在前面的 Terrain 例子中,虽然完成了一个漂亮的地面,但还缺少天空。在 Unity 中,可以使用一种叫作 Skybox 的技术来表现天空的效果,具体操作如下:

- 步骤 01** 继续前面的 Terrain 工程。在 Project 窗口单击右键,选择【Import Package】→【Skyboxes】导入 Unity 提供的 Skybox 素材。
- 步骤 02** 在 Project 窗口单击右键,选择【Create】→【Material】创建一个材质,将其类型设为 Skybox,如图 1-41 所示。

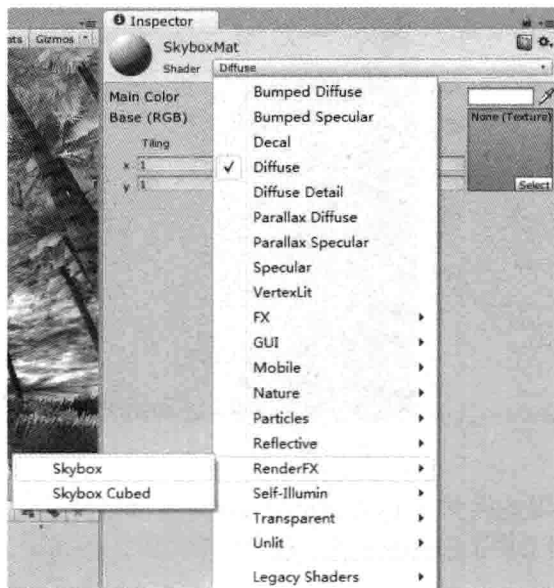


图 1-41 创建 Skybox 材质

- 步骤 03** 从 Unity 提供的 Skybox 素材中选择 6 张 Skybox 贴图,分别指定到 Skybox 材质的

Front (前)、Back (后)、Left (左)、Right (右)、Up (上)、Down (下)。
如图 1-42 所示。



图 1-42 指定贴图

步骤 04 在场景中选择 Main Camera 摄像机，在菜单栏选择【Component】→【Rendering】→【Skybox】为其添加 Skybox 组件，将 Clear Flags 设为 Skybox，将前面制作的 Skybox 材质拖动到 Custom Skybox 中，如图 1-43 所示。



图 1-43 指定 Skybox 材质

现在，已经完成了 Skybox 的制作，本节的示例工程文件保存在 Terrain 工程的 TerrainSkybox.unity 场景中，最终效果如图 1-44 所示。



图 1-44 天空效果

1.8 粒 子

Unity 中的粒子功能非常强大，它可以用来表现游戏中的魔法、云、烟火或其他特殊效果。本节将使用粒子完成一个气泡效果，基本操作如下：

- 步骤 01** 新建 Unity 工程。在 Project 窗口单击右键，选择【Import Package】→【Particles】，然后选择 Import 导入 Unity 提供的粒子素材。
- 步骤 02** 在菜单栏选择【GameObject】→【Create Other】→【Particle System】创建一个粒子发射器。一个粒子发射器包括很多模块，不同的模块具有不同的功能，默认只有少量模块是被激活的，如图 1-45 所示。

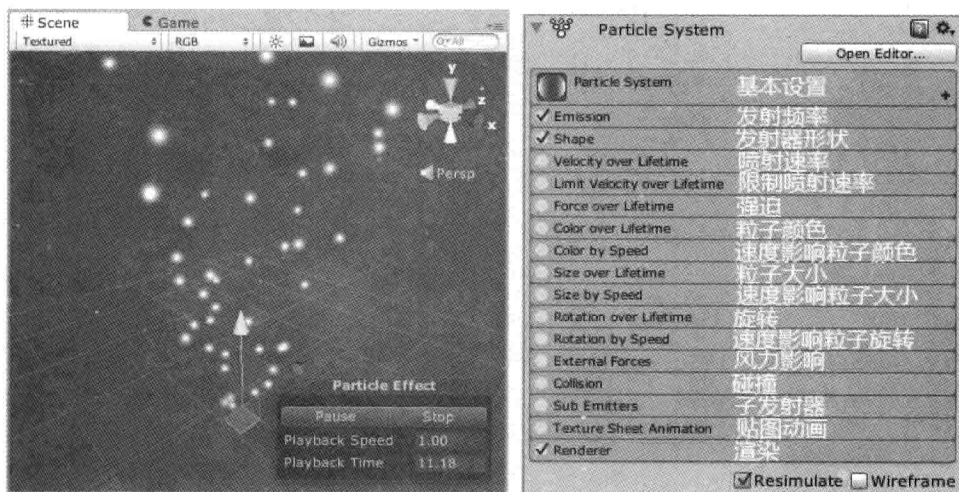


图 1-45 粒子发射器的各个模块

- 步骤 03** 在 Particle System 中设置 Start Lifetime 为 30，增加粒子的存活时间。设置 Start Speed 为 3，降低粒子运动速度。设置 Start Size 为 6，增加粒子的大小。设置 Max Particles 为 100，减少粒子的最大数量，如图 1-46 所示。

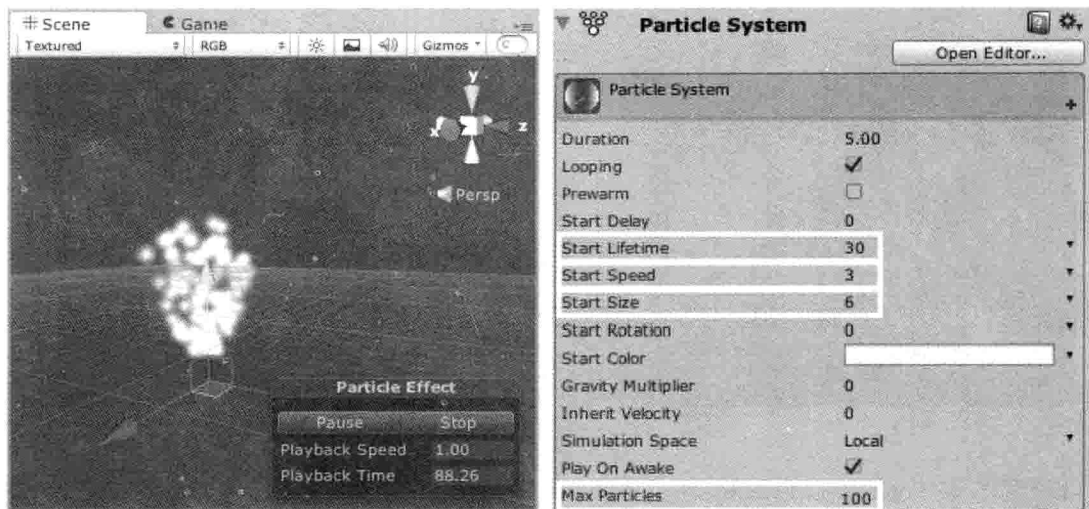


图 1-46 设置粒子初始值

步骤 04 选择 Unity 提供的 SoapBubble 材质，将其拖动到 Renderer 模块的 Material 中，如图 1-47 所示。

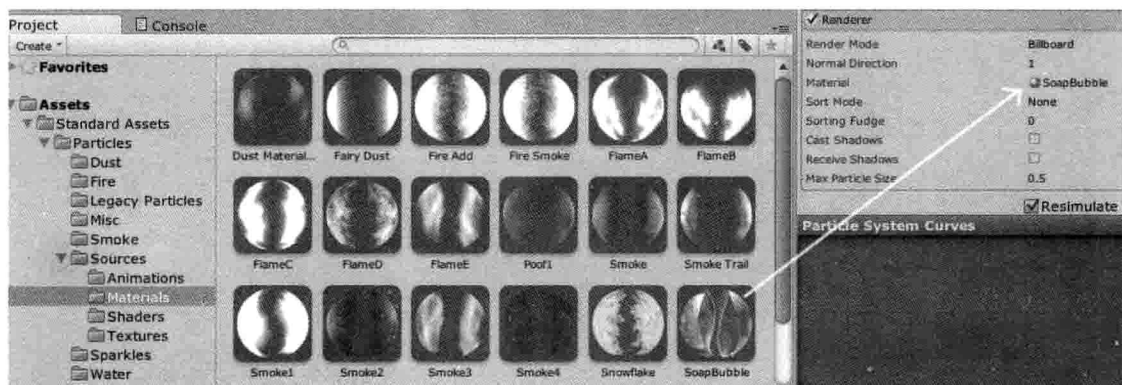


图 1-47 指定粒子材质

步骤 05 在 Emission 模块中将 Rate 设为 1，降低发射频率，如图 1-48 所示。

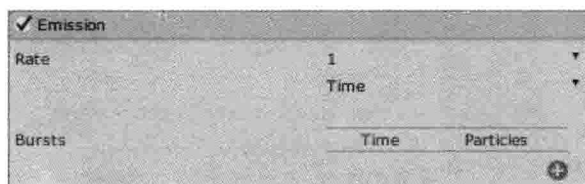


图 1-48 降低发射频率

步骤 06 在 Shape 模块中将 Shape 设为 Box，改变粒子发射器形状，如图 1-49 所示。

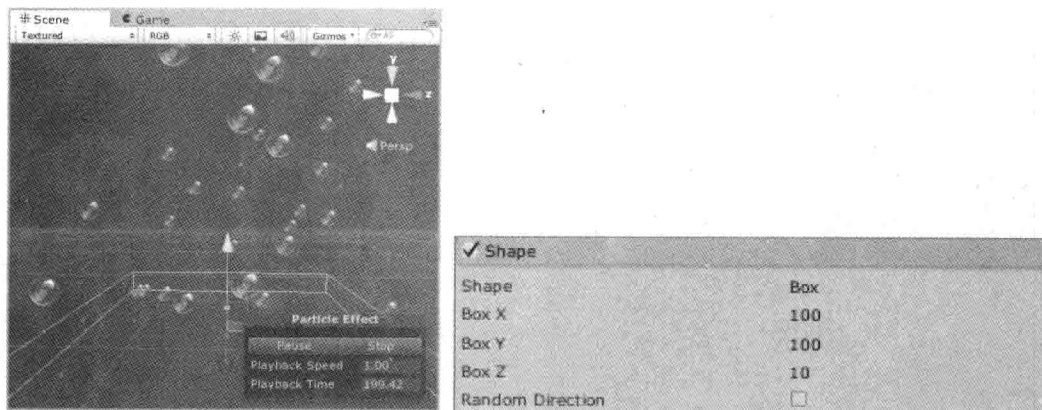


图 1-49 改变粒子发射器形状

步骤 07 在 Force Over Lifetime 模块中选择右边的小三角, 在弹出的子菜单中选择【Random Between Two Constants】, 然后将 Y 设为 0.5 和 1, 并选中 Randomize 使粒子的运动呈现一个随机的加速过程, 如图 1-50 所示。

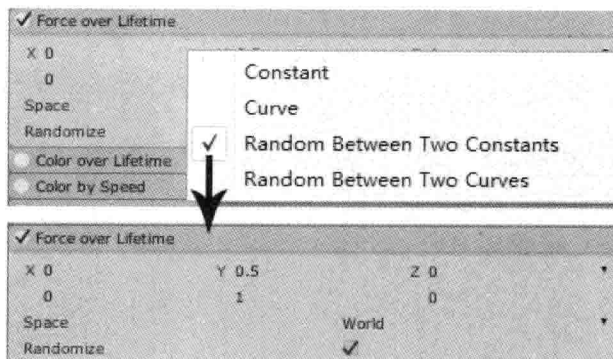


图 1-50 随机速度变化

步骤 08 激活 Size by Speed 模块。选择右边的小三角, 在弹出的子菜单中选择【Random Between Two Constants】, 然后将 Size 设为 0.3 和 2, 现在粒子的大小将随着运动速度的变化而变化, 如图 1-51 所示。

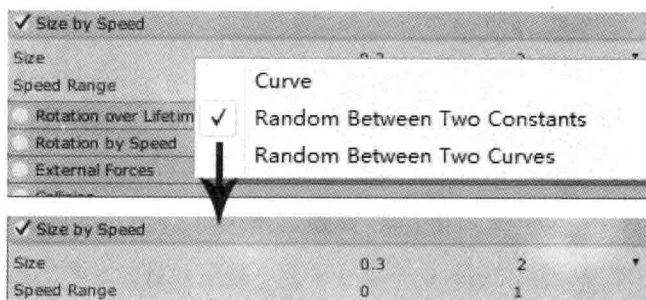


图 1-51 粒子大小变化

步骤 09 激活 Color Over Time 模块。双击色板打开 Gradient Editor 窗口。在这个窗口有一个色板, 色板从左至右表示粒子的生命周期。在色板上面的方块控制粒子透明度变

化, 下面的方块控制颜色变化。将色板上上面两边的两个方块的 Alpha 设为 0, 然后在中间单击再加两个方块, 将 Alpha 设为 1。现在, 粒子将会半透明地慢慢出现, 最后逐渐消失, 如图 1-52 所示。

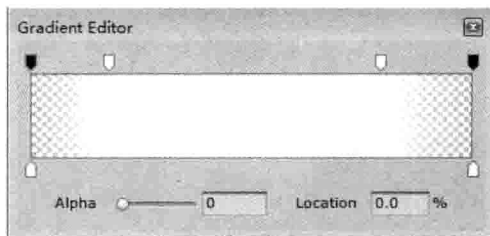


图 1-52 粒子透明度变化

最后的效果如图 1-53 所示, 粒子犹如水中的气泡, 缓缓升起, 渐渐消失。本节的示例工程文件保存在资源文件目录\chapter01_Particle 中。

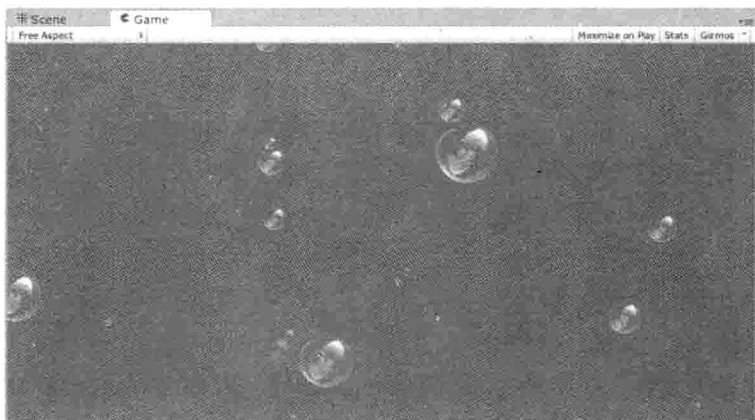


图 1-53 气泡效果

1.9 物 理

Unity 内部集成了 NVIDIA PhysX 物理引擎, 可以用来模拟刚体运动、布料等物理效果, 比如我们可以在 FPS 游戏中使用刚体碰撞模拟角色与场景之间的碰撞, 使角色不能够从墙中穿过去。此外, 物理功能还包括射线、触发器等, 都非常有用。Unity 的物理模拟还可以分层, 指定只有某些 Layer (层) 中的物体才会发生物理效果等。

下面是一个简单的示例, 我们将在一个“坡”上放置带有物理属性的“箱子”, 因为受重力影响, 它们将沿着坡路翻滚着滚下去, 并彼此产生碰撞。

步骤 01 打开资源文件目录下的 chapter01_Physic 工程, 打开 physic_start.unity 场景。在这个场景中, 预先提供了一个用于碰撞的地面模型和一个箱子模型, 如图 1-54 所示。

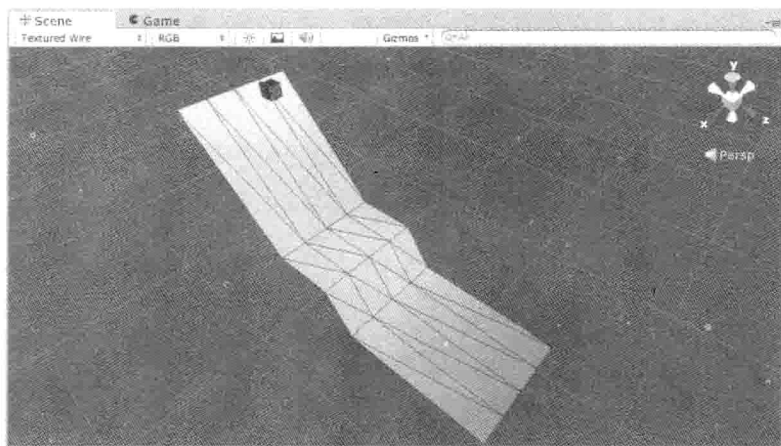


图 1-54 模型

步骤 02 选择地面模型，在菜单栏选择【Component】→【Mesh Collider】，为地面模型增加一个多边形碰撞体组件，使其具有基于多边形形状的物理碰撞功能，如图 1-55 所示。

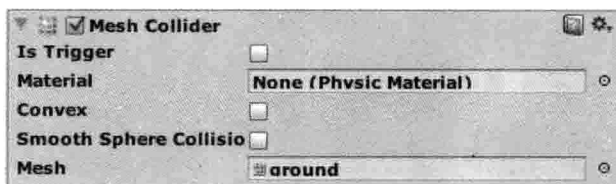


图 1-55 多边形碰撞体组件

步骤 03 选择箱子模型，在菜单栏选择【Component】→【Box Collider】，为其增加一个立方体碰撞组件，设置 Center 的值调整碰撞体的中心位置，设置 Size 的值调整碰撞体的大小。现在，箱子模型将具有基于立方体形状的物理碰撞功能，如图 1-56 所示。

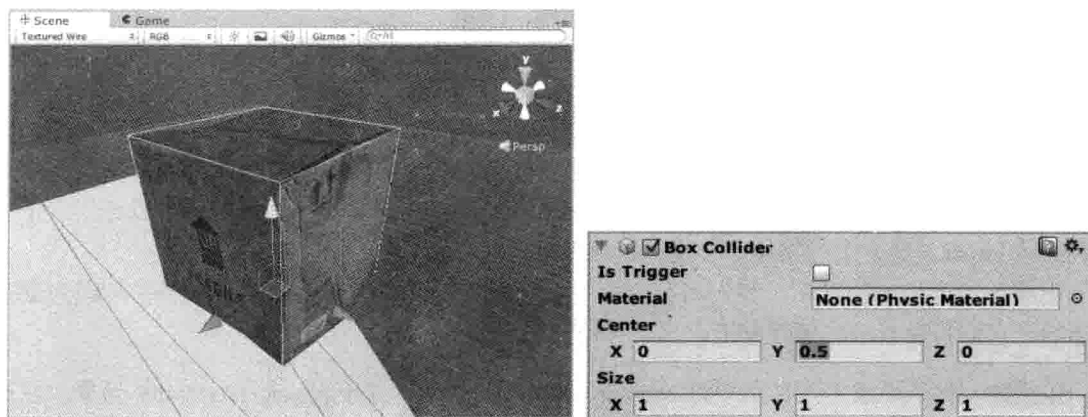


图 1-56 立方体碰撞组件

步骤 04 确定仍然选择箱子模型，在菜单栏选择【Component】→【Rigidbody】，为其添加一个刚体组件，默认 Use Gravity 为选中状态表示受重力影响，如图 1-57 所示。

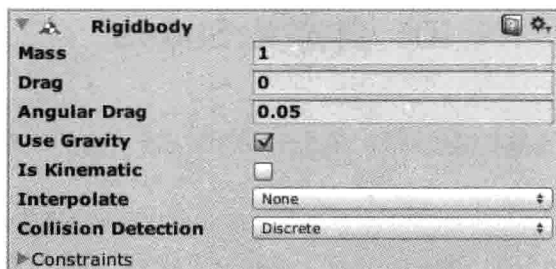


图 1-57 刚体组件

运行游戏，会发现箱子模型受重力影响从空中掉落到地面模型上翻滚落下，不过其翻滚的力度并不是很强，接下来需要改变它的物理属性使其翻滚更有力一些。

步骤 05 在 Project 窗口单击右键，选择【Create】→【Physic Material】创建一个物理材质，将 Bounciness 设为 0.9，增加弹跳力。选择箱子模型，将这个物理材质拖动到其 Box Collider 组件的 Material 中，如图 1-58 所示。运行游戏，会发现箱子模型的翻滚力度加强了。

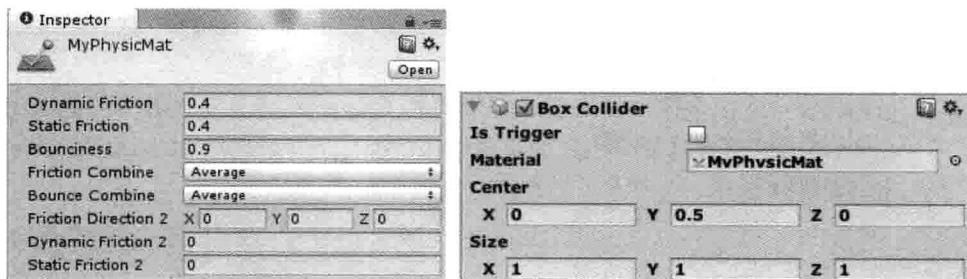


图 1-58 指定物理材质

步骤 06 按 Ctrl+D 组合键复制若干个箱子模型，将它们摆放到不同位置。运行程序，这些箱子模型将逐个掉落到地面上向下翻滚，彼此间可能还会产生多次碰撞，如图 1-59 所示。本节的示例文件保存在当前工程的 physic_finish.unity 场景中。

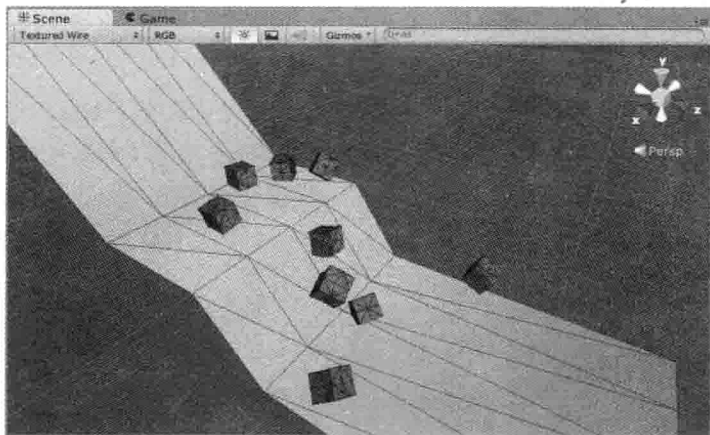


图 1-59 翻滚的箱子

1.10 自定义 Shader

Shader 是 3D 游戏中的重要组成部分,它能够对 3D 对象表面的纹理和特性进行处理,表现出各种材质效果。

Unity 提供的内置 Shader 其类型已经非常丰富,并提供原码下载。在 Unity 中,Shader 和 Materials (材质)是密不可分的,Shader 提供了各种材质属性,并且支持 CG 语言,而 Materials 则像是 Shader 的实例,可以调节 Shader 的属性并指定给模型。

下面将进行一个示例,我们首先创建一个自定义的 3D 文字字体,然后再创建一个自定义的 3D 文字 Shader。

1.10.1 自定义字体

- 步骤 01** 新建 Unity 工程,在 Windows/fonts 目录内复制 COOPBL.TTF 字体 (或任意其他字体) 到 Unity 的工程目录内。
- 步骤 02** 在 Project 窗口选择导入字体,在菜单栏选择 **【GameObject】** → **【Create Other】** → **【3D Text】** 在场景中创建一个 3D 文字,如图 1-60 所示。
- 步骤 03** 仍然选择导入的字体,在 Inspector 窗口将 Character 设为 ASCII default set,选择 Apply 确定。然后选择右上方的齿轮按钮,在其子菜单选择 **【Create Editable Copy】** 为当前字体创建一个可编辑的副本,如图 1-61 所示。

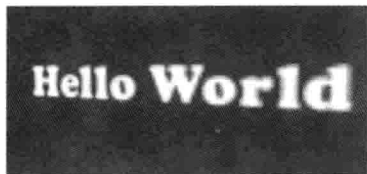


图 1-60 3D 文字

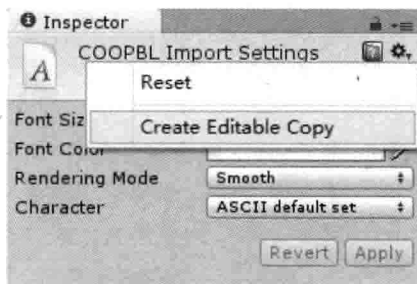


图 1-61 创建可编辑副本

- 步骤 04** 创建出的字体副本有 3 个文件,其中.png 文件是一张文字图片,我们可以使用 2D 图像处理软件如 Photoshop 修饰这张图片,使其更具有艺术效果,如图 1-62 所示。

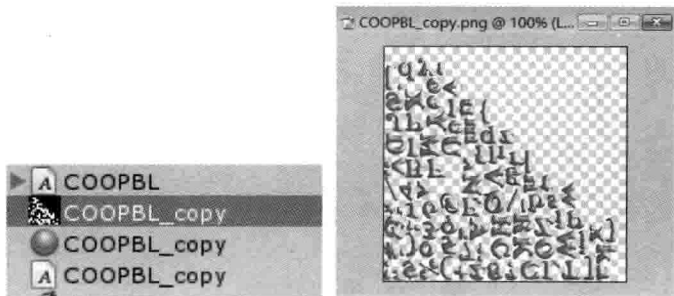


图 1-62 创建可编辑副本

步骤 05 在 Unity 中, 确定字体副本的图片仍处于选择状态, 在 Inspector 窗口将它的 Format 设为 ARGB 32 bit, 如图 1-63 所示。

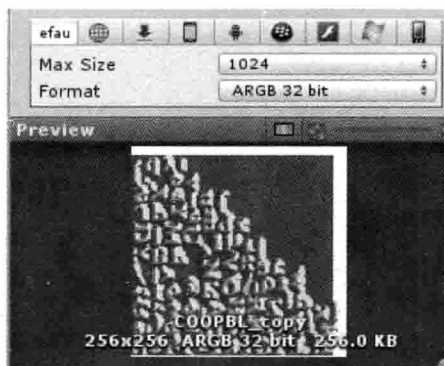


图 1-63 修改图片格式

步骤 06 在场景中选择 3D 文字, 将创建的文字副本指定给 3D 文字替换掉原来的字体, 如图 1-64 所示。

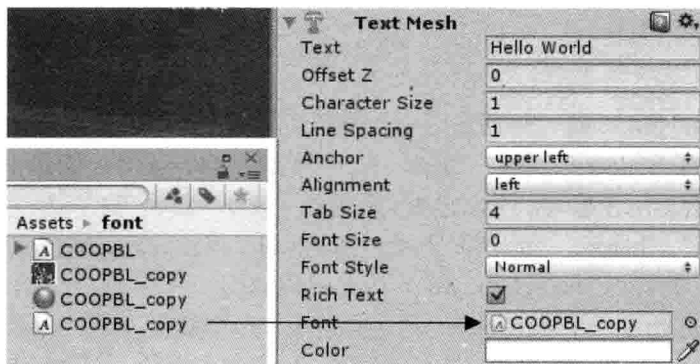


图 1-64 指定新的字体

虽然替换了字体, 但 3D 文字的显示却没有任何变化, 这是因为默认的文字 Shader 并不支持自定义的文字图片。

1.10.2 创建 Shader

接下来要创建一个 Shader, 它可以支持艺术字图片, 并确保 3D 文字不受光照影响, 且总是显示在 3D 模型前面。本示例将通过修改 Unity 内置的 Shader 实现需要的效果。

步骤 01 在 Project 窗口单击右键, 选择【Create】→【Shader】创建一个新的 Shader, 双击打开这个 Shader 文件, 它里面包括了一些基本的 Shader 代码。

```
Shader "Custom/NewShader" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
    }
}
```

```

SubShader {
    Tags { "RenderType"="Opaque" }
    LOD 200

    CGPROGRAM
    #pragma surface surf Lambert
    sampler2D _MainTex;
    struct Input {
        float2 uv_MainTex;
    };
    void surf (Input IN, inout SurfaceOutput o) {
        half4 c = tex2D (_MainTex, IN.uv_MainTex);
        o.Albedo = c.rgb;
        o.Alpha = c.a;
    }
    ENDCG
}

FallBack "Diffuse"
}

```

Shader "Custom/NewShader" 语句中双引号的部分定义了这个 Shader 在 Unity 中的名称。

Properties 定义了 Shader 的属性, 当前只有一个属性 `_MainTex`, "Base (RGB)" 是默认的属性名称, 这个名称会显示在编辑器中, 它的实际属性是 2D, 表示 2D 贴图, "white" 是 2D 贴图的默认名称, "white"、"black"、"gray" 或 "bump" 都是 Unity 内置的默认贴图名称。这部分具体内容可以参考 Unity 帮助文档 ShaderLab syntax: Properties 部分。

每个 Shader 都是由若干个 SubShader 的定义组成, 至少要有一个, 当读入 Shader 的时候, 会从第一个 SubShader 开始检查是否支持当前硬件, 如果都不支持, 则调用 FallBack "Diffuse", 使用一个默认的 Shader。

SubShader 中的 Tags 是用来告诉渲染器如何渲染这个 SubShader, 它的值都是由 = 号连接成对的字符串, 关键字与值对应的形式。当前的 RenderType 的值为 Opaque, 表示这是一个不透明的 Shader。这部分的具体内容可以参考 Unity 帮助文档的 ShaderLab syntax: SubShader Tags 部分。

SubShader 中的 LOD 表示当前 SubShader 所支持的功能细节级别, 200 表示 Diffuse 级别。我们可以在 C# 代码中调用 `Shader.maximumLOD` 设置一个值, 如果当前 SubShader 的 LOD 值大于所设置的值, 那么这个 SubShader 将不能使用。设置它主要是考虑不同的硬件对 Shader 的支持情况, 具体可以参考 Unity 帮助文档 Shader Level of Detail 部分。

标识符 CGPROGRAM 到 ENDCG 之间的部分是使用 CG 语言进行编程的部分, CG 语言是标准的 Shader 语言, 掌握它可以实现几乎任何类型的 Shader, 不过它也比较复杂, 有兴趣可以找专门的书籍来学习一下。

步骤 02 首先我们要把 Shader 的名字改掉, 这里将 "Custom/NewShader" 改为 "Custom/My3DFont"。

步骤 03 修改 Shader 的代码如下:


```

Shader "Custom/My3DFont" {
    Properties {
        _MainTex ("Base (RGB)", 2D) = "white" {}
        // 添加色彩属性,主要用来提供 alpha 控制
        _Color ("Main Color", Color) = (1,1,1,1)
    }
    SubShader {
        Tags { "RenderType"="Transparent"
               "IgnoreProjector"="True"
               "Queue"="Transparent" }
        LOD 200
        Pass {
            // 在材质中添加对色彩属性的支持
            Material {
                Diffuse [_Color]
            }
            Blend SrcAlpha OneMinusSrcAlpha //alpha 融合方式
            Lighting Off // 不受光源影响
            Cull Off // 双面显示
            ZTest Always // 总显示在最前面
            ZWrite Off // 不把像素写到 Depth 通道中
            Fog { Mode Off } // 不受雾的影响

            SetTexture [_MainTex] {
                constantColor [_Color]
                combine texture * constant }
        }
    }
    FallBack "Diffuse"
}

```

这里只添加了很少的代码,但取得了非常不错的效果。我们增加了一个 Color 属性,它提供了色彩功能,默认值是纯白色。除了改变颜色,Shader 有了 Alpha,可以表现半透明效果。

RenderType 现在被设为 Transparent,表示这是一个有透明效果的 Shader。

IgnoreProjector 设为 true,这个 Shader 将不受 Projector (用来模拟投影或之类效果的投射器)影响。

Queue 设置 Shader 的渲染顺序为 Transparent,默认的渲染顺序依次为 Background, Geometry, AlphaTest, Transparent, Overlay。

Pass 部分可以使 Shader 渲染一次,在其中我们可以设置多种渲染选项。

SetTexture [_MainTex] 为 Shader 设置贴图,combine 使贴图和颜色属性融合。

步骤 04 选择字体副本的材质,将 Shader 设为 My3DFont,场景中的 3D 文字也会相应产生变化,如图 1-65 所示。

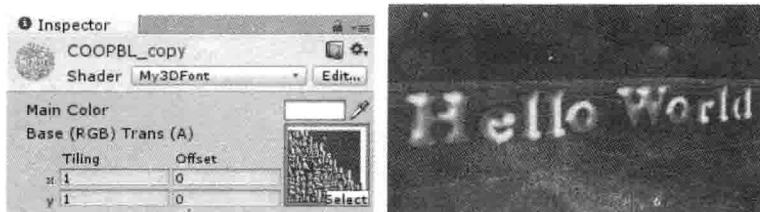


图 1-65 指定新的 Shader

现在, 我们可以在场景中随意放些其他模型, 3D 文字总是显示在最前面, 它不会受光照影响, 同时我们可以为这个 3D 文字指定 Alpha 效果。

本例只是对 Shader 非常粗浅的尝试, 最后的示例工程文件保存在资源文件目录 chapter01_3DFontShader 中。

1.11 游戏资源

虽然游戏的逻辑需要靠代码实现, 但如果没有画面和声音表现, 恐怕不会有人去玩这个游戏。Unity 中美术的资源主要包括 3D 模型、动画和贴图, 同时也支持如 Wave、MP3、Ogg 等音效格式, 导入这些资源的方式是一样的, 只要将它们复制粘贴到 Unity 工程路径内即可, 开发者可以自定义路径结构管理资源, 就像在 Windows 资源管理器上操作一样。

Unity 支持多种 3D 模型文件格式, 如 3ds Max、Maya 等。大部分情况, 可以将 3D 模型从 3D 软件中导出为 FBX 格式到 Unity 中使用。

并不是所有导入到 Unity 工程中的资源都会被使用到游戏中, 这些资源一定要与关卡文件相关才会被加载到游戏中。除此之外, 还有两种方式可以动态地加载资源到游戏中: 一种是将资源制作成 AssetBundles 上传到服务器, 动态地下载到游戏中; 另一种是将资源复制到 Unity 工程中名字为 Resources 的文件夹内, 无论是否真的在游戏中使用了它们, 这些文件都会被打包到游戏中。我们可以通过资源的名称, 动态地读取资源, 这种方式更近似于传统的 IO 读取方式。

在 Unity 中还可以创建一种叫 Prefab 的文件, 可以将它理解为一种配置。开发者可以将模型、动画、脚本、物理等各种资源整合到一起, 做成一个 Prefab 文件, 随时可以重新运用到游戏中的各个部分。

1.11.1 贴图

无论是 2D 游戏还是 3D 游戏, 都需要使用大量的图片资源。Unity 支持 PSD、TIFF、JPEG、TGA、PNG、GIF、BMP、IFF、PICT 格式的图片。大部分情况, 推荐使用 PNG 格式的图片, 它的容量更小且有不错的品质。

对于作为模型材质使用的图片, 其大小必须是 2 的 N 次方, 如 16×16、32×32、128×128 等, 通常会将其 Texture Type 设为默认的 Texture 类型, 将 Format 设为 Compressed 模式进行压缩。在不同平台, 压缩的方式可能是不同的, 可以通过 Unity 提供的预览功能查看压缩模式和图片压缩后的大小, 如图 1-66 所示。

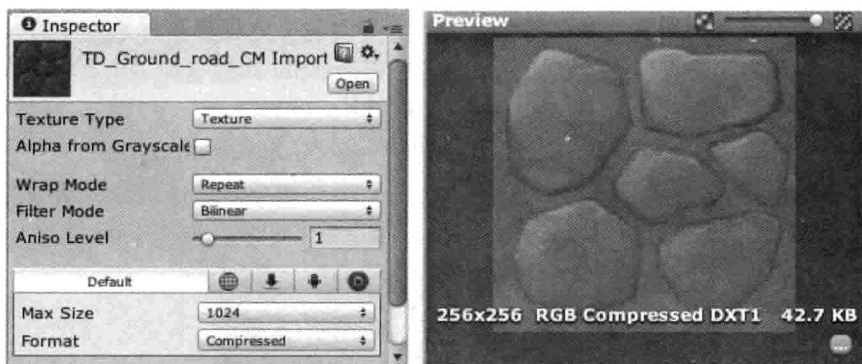


图 1-66 将材质类型设置为 Texture

如果图片将作为 UI 使用, 需要将 Texture Type 设为 GUI。值得注意的是 Format 的设置, 如果使用的图片大小恰好是 2 的 N 次方, 虽然也可以将其设为 Compressed 模式进行压缩, 但画面质量可能会受到影响。

对于那些大小非 2 的 N 次方的图片, 即使将其设为 Compressed, 在手机平台也不会得到任何压缩, 这种情况, 可将其设为 16bits 试试, 如果图像在 16bits 模式下显得很糟糕, 那只能将 Format 设为 32bits, 但图片容量会变得很大。

1.11.2 3ds Max 静态模型导出

3ds Max 是最流行的 3D 建模、动画软件, 可以使用它来完成 Unity 游戏中的模型或动画, 最后将模型或动画导出为 FBX 格式到 Unity 中使用。3ds Max 静态 (没有动画) 模型的制作和导出流程, 可以遵循下列步骤和规范:

步骤 01 在 3ds Max 菜单栏选择【Customize】→【Units Setup】, 将单位设为 Meters, 然后选择【System Unit Setup】, 将 1 Unit 设为 1 Centimeters, 如图 1-67 所示。

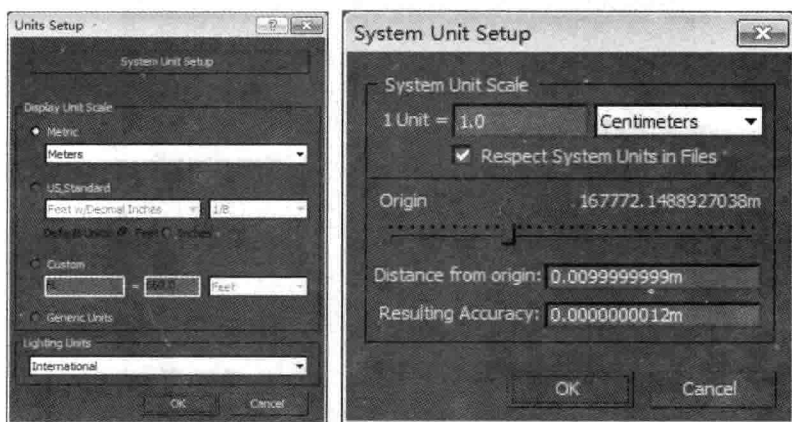


图 1-67 设置 3ds Max 单位

步骤 02 完成模型、贴图的制作, 确定模型的正面面向 Front 视窗。如果需要在 Unity 中对模型使用 Lightmap, 一定要给模型制作第 2 套 UV。

- 步骤 03** 如果没有特别需要,通常将模型的底边中心对齐到世界坐标原点(0,0,0)的位置。方法是确定模型处于选择状态,在 Hierarchy 面板选择 Affect Pivot Only,将模型轴心点对齐到世界坐标原点(0,0,0)的位置。
- 步骤 04** 在 Utilities 面板选择 Reset XForm 将模型坐标信息初始化。
- 步骤 05** 在 Modify 窗口单击右键,选择 Collapse All 将模型修改信息全部塌陷。
- 步骤 06** 按 M 键打开材质编辑器,确定材质名与贴图名一致,如图 1-68 所示。

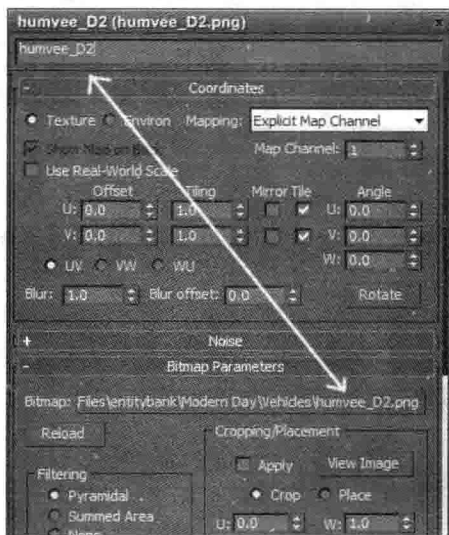


图 1-68 保持材质名称与贴图名称一致

- 步骤 07** 选中要导出的模型,在菜单栏选择【File】→【Export】→【Export Selected】,选择 FBX 格式,打开导出设置窗口,可保持大部分默认选项,取消选择 Animation,确定单位设为 Centimeters 且 Y 轴向上,选择【OK】将模型导出,如图 1-69 所示。

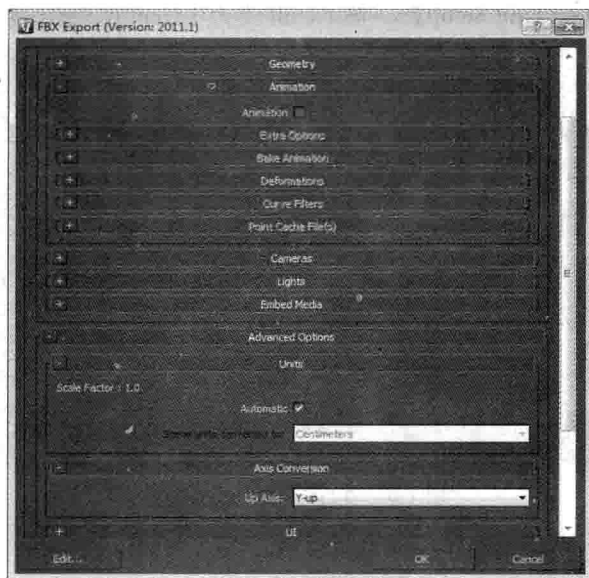


图 1-69 导出设置

步骤 08 将导出的模型和贴图复制粘贴到 Unity 工程路径 Assets 文件夹内的某个位置即可导入到 Unity 工程中。

1.11.3 3ds Max 动画导出

动画模型是指那些绑定了骨骼并可以动画的模型，其模型和动画通常需要分别导出，动画模型的创建流程可以先参考前一节步骤 1~6，然后还需要：

步骤 01 使用 Skin 绑定模型。

步骤 02 创建一个 Helper 物体（如 Point）放到场景中的任意位置，这么做的目的是为了使得导出的模型和动画的层级结构一致。

步骤 03 选择模型（仅导出动画时不需要选择模型）、骨骼和 Helper 物体，在菜单栏选择【File】→【Export】→【Export Selected】打开导出设置窗口，注意要选中 Animation 才能导出绑定和动画信息，其他设置与导出静态模型基本相同。

模型文件可以与动画文件分开导出，但模型文件中的骨骼与层级关系一定要与动画文件一致。仅导出动画的时候，不需要选择模型，只需要选择骨骼和 Helper 物体导出即可。

动画文件的命名需要按“模型名@动画名”这样的格式命名，比如模型命名为 Player，动画文件即可命名为 Player@idle、Player@walk 等。

1.11.4 Maya 模型导出

Maya 也是一款非常流行且功能强大的 3D 动画软件，它的内部坐标系统与 Unity 一样都是 Y 轴向上，非常适合完成 Unity 游戏的模型工作，下面是一个基本的工作流程参考：

步骤 01 在 Maya 的菜单栏选择【Window】→【Settings/Preference】→【Preferences】，将单位设为 meter，然后选择【Save】保存退出，如图 1-70 所示。

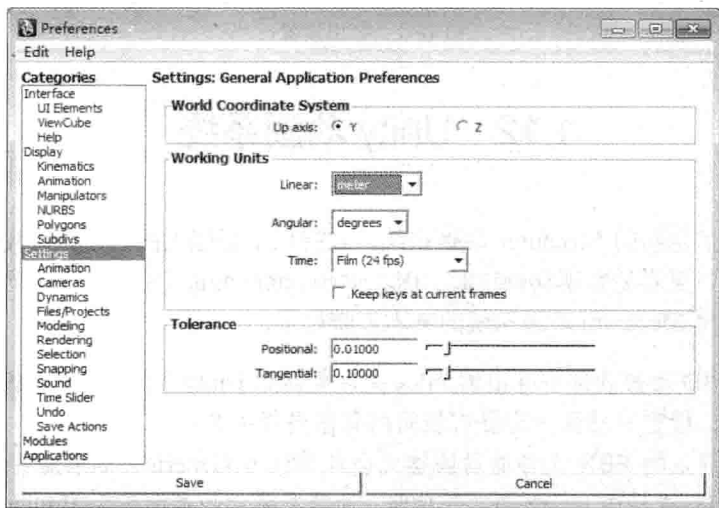


图 1-70 设置单位

- 步骤 02** 完成模型、贴图的制作, 确定模型的正面面向 Front 视窗。如果需要在 Unity 中对模型使用 Lightmap, 一定要给模型制作第 2 套 UV。
- 步骤 03** 如果没有特别需要, 通常将模型的底边中心对齐到世界坐标原点 (0, 0, 0) 的位置。方法是选择模型, 按 Insert 键, 然后按住 X 键将模型轴心点对齐到世界坐标原点 (0, 0, 0) 的位置。
- 步骤 04** 在菜单栏选择【Modify】→【Freeze Transformations】将坐标信息归零。
- 步骤 05** 在菜单栏选择【Edit】→【Delete All by Type】→【History】将历史记录清空。
- 步骤 06** 选择需要导出的模型, 在菜单栏选择【File】→【Export Selection】, 将导出格式设为 FBX, 选择 Export 打开设置窗口。如果不需要导出动画, 可以取消选中 Animation, 然后将 Units 设为 Centimeters, 最后选择【Export】导出, 如图 1-71 所示。

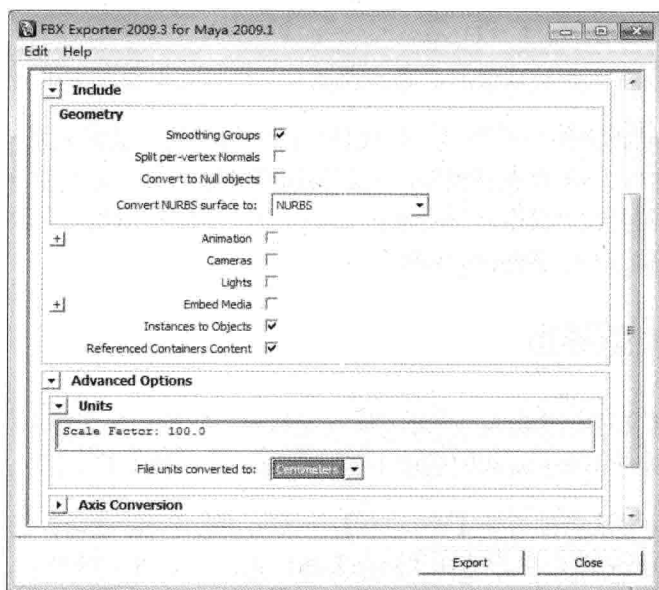


图 1-71 导出设置

1.12 Unity 动画系统

Unity4.0 引入了全新的 Mecanim 动画系统, 它提供了更强大的功能, 使用一个叫状态机的系统控制动画逻辑, 更容易实现动画过渡、IK、动画 retargeting (将同一个动画使用到不同的模型上) 等功能。使用 Mecanim 动画系统的基本步骤如下:

- 步骤 01** 将从 3D 动画软件中导出的 FBX 文件复制到 Unity 工程中。一个模型可以拥有多个动画, 模型与动画一定要有相同的骨骼层级关系。
- 步骤 02** 默认导入的 FBX 文件的动画格式会自动设为 Generic。如果需要使用 Mecanim 提供的 IK 或动画 retargeting 等功能, 还需要将动画类型设为 Humanoid, 这是专门针对两足人类动作的一种动画系统, Mecanim 提供的大部分高级功能均只针对这种

动画类型，如图 1-72 所示。

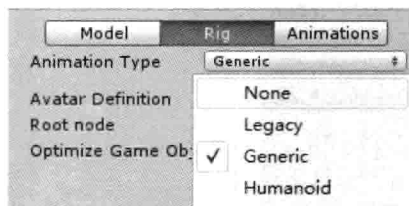


图 1-72 设置动画循环

Legacy 模式是 Unity 在 4.0 版本之前的一种标准动画模式，它的功能相对较弱，但使用起来更简单，本书塔防游戏教程中使用到了这种模式。

步骤 03 当将带有动画的 FBX 文件导入 Unity 工程后，如果需要循环播放该动画，只需要选中 Loop Time 即可使其成为一个循环播放的动画，如图 1-73 所示。

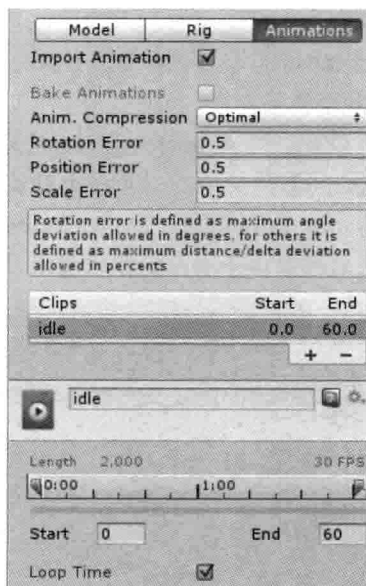


图 1-73 设置动画类型

在 Unity 中，像这种手工设置动画循环之类的事情，都可以使用代码批量完成，在本书后面的实例中会使用到这类方法。

步骤 04 当动画导入后，在 Project 窗口展开动画文件层级，选择动画，在 Inspector 窗口预览动画，如图 1-74 所示。如果动画文件本身没有模型，只需要将模型文件拖放到预览窗口即可。

步骤 05 在 Project 窗口单击右键，选择【Create】→【Animator Controller】创建一个动画控制器（如果是选择 Legacy 动画模式，不需要创建 Animator Controller）。

步骤 06 将包含绑定信息的模型制作成一个 Prefab，在 Animator 组件的 Controller 中为其指定动画控制器，如果需要使用脚本控制模型位置，取消选择 Apply Root Motion 选项，如图 1-75 所示。



图 1-74 预览动画

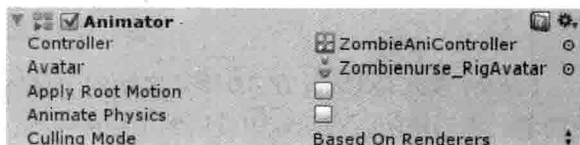


图 1-75 指定动画控制器

- 步骤 07** 确定动画控制器处于选择状态，在菜单栏选择【Window】→【Animator】打开 Animator 窗口。
- 步骤 08** 如果需要分层动画，比如角色的上半身和下半身分别播放不同的动作，选择左上方 Layers 上的 + 号添加动画层。
- 步骤 09** 将与当前模型相关的动画拖入 Animator 窗口（注意这是与不同的动画层对应的）。
- 步骤 10** 右键选择【Set As Default】使选中的动画成为默认初始动画。
- 步骤 11** 分别选择不同的动画，单击右键选择【Make Transition】使动画之间产生过渡，由哪个动画过渡到哪个动画取决于游戏的逻辑需求，如图 1-76 所示。

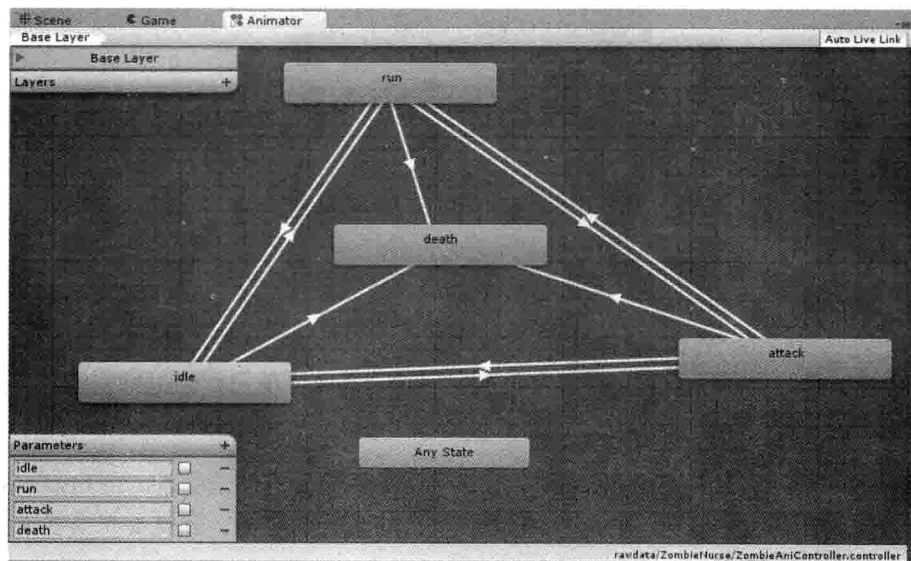


图 1-76 设置动画过渡

现在播放动画，动画会自动从默认动画一直播放到设置的最后一个动画，但游戏中的动画播放往往是由逻辑或操作控制的，比如按一下鼠标左键，播放某个动画。默认的动画过渡是使用时间控制，我们也可以按条件过渡动画，并使用代码控制。

- 步骤 12** 在 Animator 窗口有一个 Parameters 选项，选择 + 号即可创建 Vector、Float、Int

和 `bool` 类型的数值，每个数值还有一个名字。比如我们希望从一个叫 `idle` 的动画过渡到另一个叫 `run` 的动画，这时可以创建一个 `bool` 类型的值，命名为 `idle`，它默认的状态是 `false`。

步骤 13 选择 `idle` 动画到 `run` 动画之间的过渡线，在 `Conditions` 中将默认的 `Exit Time` 改为 `run`，如图 1-77 所示。

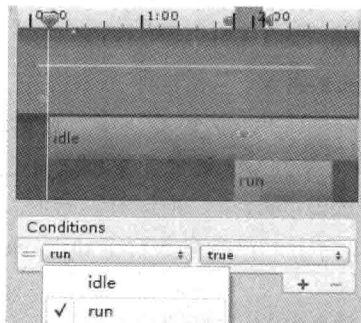


图 1-77 设置动画过渡条件

步骤 14 在控制动画播放的代码中，首先要获得 `Animator` 组件，然后通过 `SetBool` 将 `run` 的值设为 `true`，即从 `idle` 动画过渡到 `run` 动画，示例代码如下：

```
Animator m_ani;
void Start () {
    // 获得动画组件
    m_ani = this.GetComponent<Animator>();
}
void Update () {
    // 获取当前动画状态
    AnimatorStateInfo stateInfo= m_ani.GetCurrentAnimatorStateInfo(0);
    // 如果状态处于 idle
    if (stateInfo.nameHash == Animator.StringToHash("Base Layer.idle"))
    {
        // 如果按鼠标左键, 播放 run 动画
        if (Input.GetMouseButtonUp(0))
        {
            m_ani.SetBool("run", true);
        }
    }
    else
        m_ani.SetBool("idle", false);
}
```

Unity 还提供了一个 `Animator Override Controller`，它可以继承其他 `Animator Controller` 的设置但使用不同的动画，这样就不用重新设置动画的逻辑关系了。

1.13 美术资源的优化

美术资源的使用会对游戏的性能造成很大影响,下面列出了一些需要注意的地方,以供参考。

(1) 模型顶点的数量会影响 GPU 的性能。通常,在手机平台上,模型的顶点数量控制在 100 000 个以内为佳。在 PC 平台上,模型的顶点数量可控制在几百万个以内。

(2) 减少模型 UV 接缝和硬边的数量。

(3) 场景中模型的数量会影响到 CPU 的性能,所以要尽可能减少场景中的模型数量,或者将使用相同 Material 的多个模型合并到一起(如果模型之间是使用的不同的 Material,合并没有任何意义),这样会减少 draw calls 的数量。当运行游戏时,在 Game 窗口选中 Stats 会看到 draw calls 的统计。通常,在手机平台上,控制在数百个 draw calls 内为佳,在 PC 平台上可以控制在几千个以内。

(4) 尽可能减少角色模型骨骼的数量。

(5) 避免在 Unity 内使用 IK 动画。

(6) 减少 Material 的数量。通常,一个模型至少需要一张贴图,如果可能,可以将多张贴图拼成一张贴图,这样多个模型可以共享同一个 Material。

(7) 尽可能压缩贴图,如果不能压缩,则尽可能将贴图设为 16 位而不是 32 位。

(8) 尽可能为贴图使用 Generate Mip Maps 功能,除非贴图总是 1:1 渲染显示,比如 UI 或 2D 游戏。

(9) 首选使用 Mobile 或 Unlit 的 Shader,它们同样可以很好地工作在 PC 平台。

(10) 将不需要显示的模型隐藏,可以减少 CPU 的工作。

(11) 雾会对性能造成较大影响。

(12) 尽可能减少像素灯光、阴影、反射的使用,这些功能会导致模型被渲染多次,加重 CPU 的负担。

(13) 尽可能使用 Lightmap 而不是用实时光照亮场景。

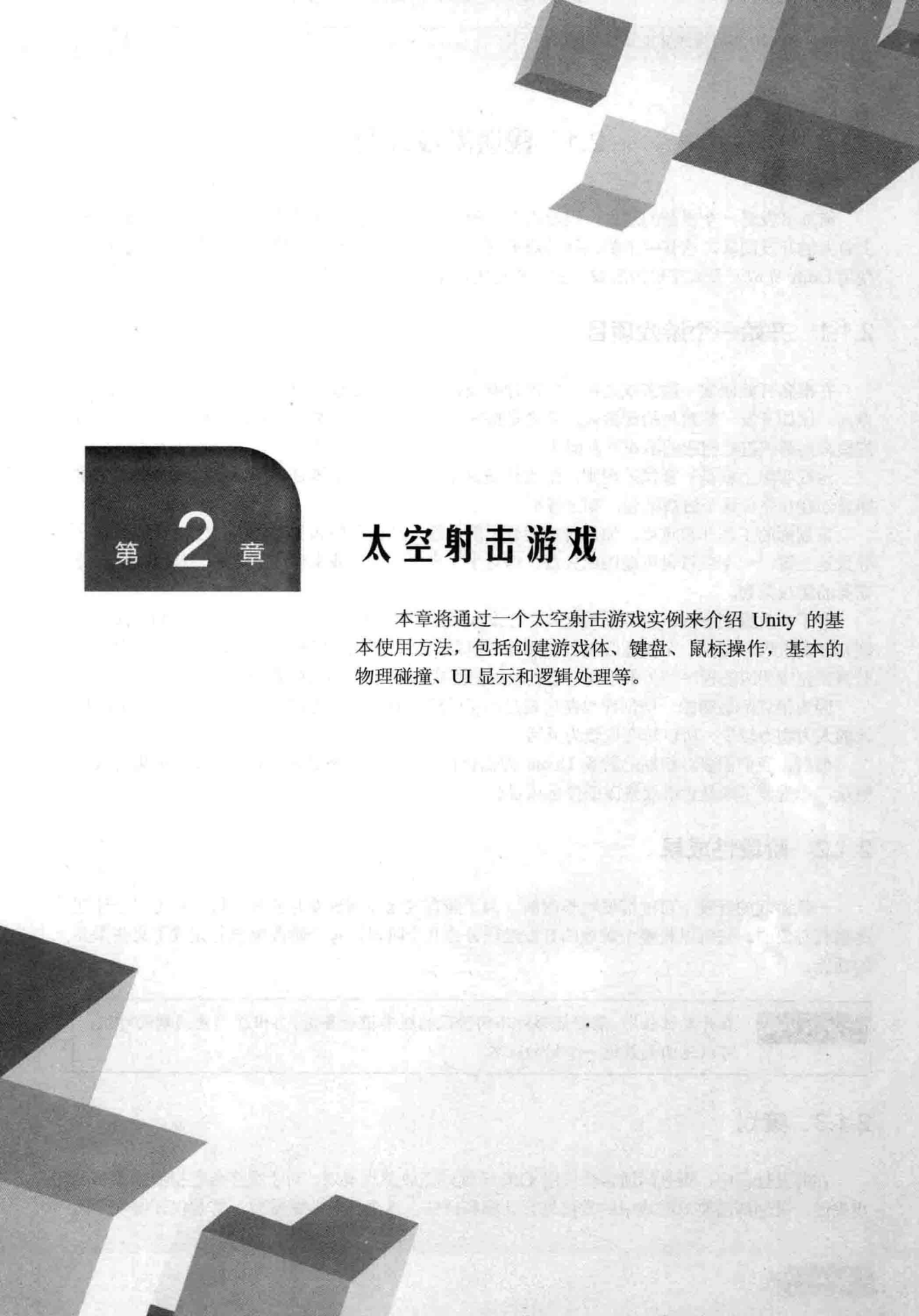
(14) 小心使用实时阴影,它会对性能造成较大影响。

(15) 在手机平台上,带有 Alpha 效果的 Shader 会对性能造成较大影响。

小 结

本章首先介绍了如何安装和激活 Unity,并在 PC 机上演示了一个“Hello World”程序;之后逐步介绍了各个模块的基本使用方法,包括如何使用光照系统,创建 Lightmap 和 Light Probe 等,如何创建 Terrain 和 Skybox,通过示例说明粒子、物理和 Shader 的基本应用;最后还介绍了在 3D 动画软件导出模型、动画到 Unity 的流程和规范。

本书主要是以实例的方式讲解不同的应用技术,如果希望了解 Unity 功能的全部细节,最好的方法是查看 Unity 的帮助文档。



第 2 章

太空射击游戏

本章将通过一个太空射击游戏实例来介绍 Unity 的基本使用方法，包括创建游戏体、键盘、鼠标操作，基本的物理碰撞、UI 显示和逻辑处理等。

2.1 浅谈游戏开发

游戏开发是一个复杂的过程,很难由个人独立完成。一款游戏的背后,通常有一个数人至上百人的开发团队,这其中主要包括游戏策划、程序员、美术制作、项目管理人员等。在我们使用 Unity 完成一个太空射击游戏之前,要先从宏观上了解一下游戏开发的各个环节。

2.1.1 开始一个游戏项目

在准备开始研发一款游戏之前,大部分游戏公司会对市场进行调研,然后确立项目的开发方向,比如开发一款角色扮演游戏,或者是格斗游戏等。之后,游戏公司需要根据项目的预期规模来招募或组建相应的游戏开发团队。

游戏实际上就是一套娱乐规则,游戏开发首先需要的是一份策划,游戏策划师的主要工作即是创建和完善这个游戏策划,制定规则。

策划师的工作非常重要,如果游戏策划出现问题,那么程序和美术的工作也会受到影响,导致返工等,一个项目也可能因此失败,但对于一个原创的游戏来说,很难从一开始就有一份完美的游戏策划。

对于一款原创游戏,为了回避风险,在投入力量开发之前,可以考虑先进入 Prototype (原型) 开发阶段。这是一个快速尝试各种想法的阶段,策划师需要明确游戏的核心游戏规则,同时需要完成相应的程序和美术工作,通过较少的游戏内容,体现出游戏的核心乐趣。

因为策划师最初的一些创作思路可能是不正确的,开发可能出现反复,但因为这个阶段投入的人力物力较少,所以转变也较为灵活。

最后,我们需要对初期的游戏 Demo 做出评估,游戏的乐趣是否达到了预期,如果游戏不好玩,那就要考虑是否继续修改或停止项目。

2.1.2 阶段性成果

一款游戏的开发,可能需要很长时间,为了能有效地控制进度并监督质量,制定一个计划是很有必要的。它可以将整个游戏的开发过程分为几个阶段,每个阶段标志着完成了某些重要的功能。



提示

在开发过程中,最好能够对不同阶段的版本进行备份,当出现严重问题的时候,可以返回到最近一个好的版本。

2.1.3 策划

在开发过程中,策划师通常会使用 Office 软件完成游戏策划,对于很多角色扮演或策略游戏来说,策划师需要对游戏中的数值进行计算和评估,这个环节非常重要,它是保证游戏平衡

性的关键。

策划师还需要花费大量时间完成关卡编辑之类的工作。游戏规则本身并不能直接带来乐趣，只有通过具体的编排才能将规则有趣地反映出来。对于 Unity 游戏来说，关卡编辑工作主要都是在 Unity 的编辑器中完成的，所以策划师也同样需要熟悉 Unity 的基本操作。

2.1.4 编写脚本

Unity 程序员的主要工作就是编写脚本。Unity 支持多种不同的脚本语言，其中 C#语言的使用最为广泛，如果你还不知道 C#，可以参考本书的附录 A 快速地掌握它。

在 Unity 中，每个游戏中的物体都可以称为是一个游戏体（Game Object），实际上，一个 Unity 游戏，就是由不同的游戏体组成的。

Unity 中的游戏体可以拥有多个组件（Component）。组件可以是一个脚本，一个模型，一个物理碰撞体，一张贴图，一个粒子发射器，或是一个声音播放器。有了这些组件，游戏体就有了相应的功能，程序员可以通过编写脚本控制游戏体及它所拥有的组件，从而实现游戏的逻辑。

Unity 不但能开发单人游戏，也能开发多人游戏。它内置了 RakNet（一个网络游戏开发包），适用于快速开发多人游戏。对于现在流行的“弱联网”非实时互动游戏，Unity 提供了 HTTP 网络通信功能，可以方便地与 PHP 或 .NET 服务器实现网络通信。对于大型的网络游戏，开发者可以使用 C#编写基于 .NET 的 Socket 客户端程序与使用 C++、C#或 Java 开发的服务器端实现网络通信。

2.1.5 美术

对于 3D 游戏来说，美术团队的人数往往是非常多的。美术的职位比较多，包括原画设计、UI 界面设计、3D 角色模型师、3D 场景模型师、3D 动画师等。

美术的工作非常重要。现在，每天都有新游戏投入市场，一款游戏如果没有较好的画面，它将很容易被忽略。

在一款游戏当中，美术工作决非仅是艺术创造，同时还包含大量的技术环节。在手机平台上，内存相对比较小，如果不注意控制美术资源的总量，则可能会造成严重的内存问题，导致程序无法启动或经常崩溃。所以，在注重画面效果的同时，还要注意如何优化美术资源，在相对节约的情况下表现出最好的美术效果。

Unity 支持几乎所有高端的 3D 动画软件，如 3ds Max、Maya、LightWave 等，美术人员可以按需求选择自己的 3D 动画软件，将制作的模型和动画导出为 FBX 格式供 Unity 使用，在本书后面的章节中，会对这部分内容专门介绍。

2.1.6 QA 测试

为了保证游戏质量，需要安排 QA 人员对游戏进行全面的测试。QA 的主要工作是找出游戏中的 BUG，这些 BUG 可能是程序造成的，也可能是关卡编辑错误造成的，还可能是美术资源

的问题造成的。

我们最好能够对 BUG 进行分级 (比如 A、B、C) 并跟踪修改记录, 优先去修改较为严重的 BUG。

因为修改 BUG 本身有时会造成出现新的 BUG, 所以在完成一个版本的修改后, 仍需要对整个游戏进行较全面的测试。

2.1.7 发布游戏

使用 Unity 开发的手机游戏, 可以发布到苹果的 App Store, Google 的 Google Play, Amazon 的 Kindle Fire 等在线商店, 这些都是全球性的商店, 用户可以在上面使用信用卡付费或免费下载数字产品。此外, Android 在国内也有很多本土在线商店, 如腾讯无线等。

网页平台仍是目前最大的一个游戏平台。Unity 有自己的网页游戏格式, 可以使用户在网页上直接体验 Unity 游戏, 但用户必须安装 Unity 的网页插件, 目前这个插件的使用还并不是很广泛。

Unity 从版本 4.0 开始正式支持 Flash 平台, 几乎所有浏览器上面都安装有 Flash 插件, 所以将游戏以 Flash 格式发布到网页上是个不错的主意。目前国内很多公司都在发展以 Flash 游戏为主的开放平台, 任何开发者都可以将产品放到上面运营, 进而得到丰厚的回报。

2.2 游戏策划

就像前面讨论的, 在开始一款游戏之前我们需要一份游戏策划, 这个太空射击游戏也不例外。本节将准备一份游戏策划, 当然, 这是个非常简单的游戏, 但我们可以从中了解到很多 Unity 游戏开发的基本功能。

2.2.1 游戏介绍

在游戏中, 主角和敌人是不同的太空飞行器。游戏开始后, 主角会迎着敌方的火力前进。消灭敌人会取得分数, 游戏没有尽头, 如果主角战败, 则游戏结束。

2.2.2 游戏 UI

屏幕上会显示主角的装甲以及得分。如果游戏结束, 屏幕上将会显示“游戏结束”, 同时还会显示出“再试一次”按钮。

2.2.3 主角

主角拥有 3 级装甲, 被敌人击中或撞击 1 次, 损失 1 级装甲, 当装甲为 0 时, 游戏结束。

2.2.4 游戏操作

本游戏将在 PC 平台上开发，按键盘上的 W、S、A、D 或上、下、左、右键控制主角上下左右飞行，按空格或鼠标左键射击。



提示

如果将这个游戏移植到手机平台，则可能要改变操作或自定义虚拟操作键。

2.2.5 敌人

游戏中只有初级敌人和高级敌人两种：

- (1) 初级敌人：装甲较弱，以撞击主角为主，沿弧线飞行。
- (2) 高级敌人：装甲较强，可以发射子弹，直线飞行。

2.3 导入美术资源

在本书的附带资源文件中包括用于这个太空射击游戏的 3D 模型和贴图，Unity 支持多种格式的 3D 模型和贴图，比较常用的是 FBX 格式的模型和 PNG 格式的贴图。下面，我们看一下如何将它们导入到 Unity 工程中。

步骤 01 在资源文件目录 rawdata 复制 airplane 文件夹，这个文件夹内包括所有游戏需要的模型和贴图文件，如图 2-1 所示。

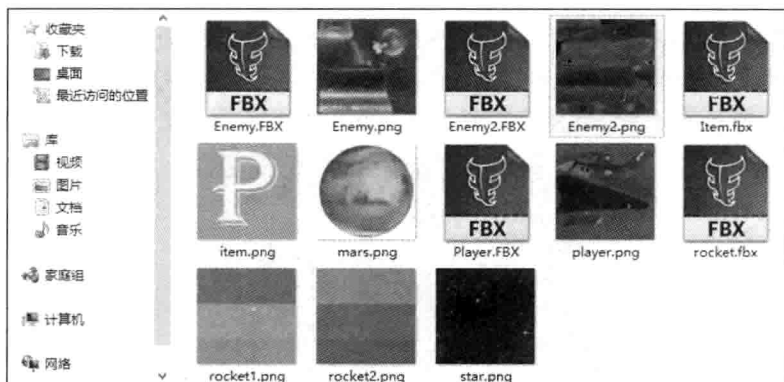


图 2-1 模型和贴图资源

步骤 02 创建一个新的 Unity 工程，在 Project 窗口选择 Assets，然后单击右键，选择【Show in Explorer】，将前面复制的 airplane 文件夹粘贴到 Assets 文件夹内，返回 Unity，会发现模型和贴图已经被导入当前的 Unity 工程中，如图 2-2 所示。

Project 窗口是一个浏览器窗口，主要负责资源管理，它与硬盘上游戏工程下的 Assets 文件夹是相对应的，我们可以直接通过复制粘贴的方式将外部资源导入到 Unity 中，或者在 Project

窗口单击右键, 选择【Import New Asset】也可以将资源导入。

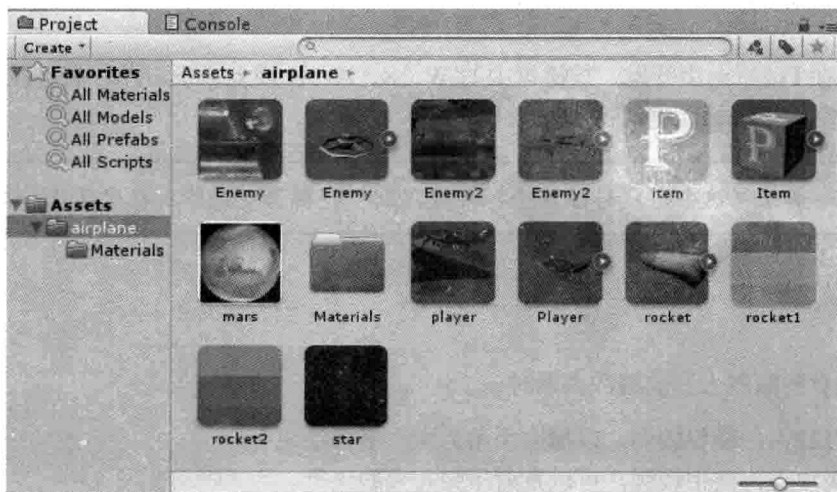


图 2-2 导入模型和贴图

2.4 创建场景

游戏是发生在太空中, 背景是一颗巨大的火星和浩瀚的星空。在这部分, 我们将介绍如何创建材质球, 并为星空完成一个 UV 动画。

2.4.1 创建火星背景

步骤 01 在菜单栏选择【File】→【New Scene】创建一个新的场景。

步骤 02 在菜单栏选择【File】→【Save Scene As】将场景另存为 level1.unity, 如图 2-3 所示。接下来这个游戏的大部分工作都将在这个场景中完成。



图 2-3 新建场景

步骤 03 在菜单栏选择【GameObject】→【Create Other】→【Plane】创建一个平面体作为火星背景模型。

步骤 04 在 Project 窗口单击右键选择【Create】→【Material】创建一个材质球, 将其命名为 Background, 选择【Select】指定 mars.png 作为贴图, 如图 2-4 所示。

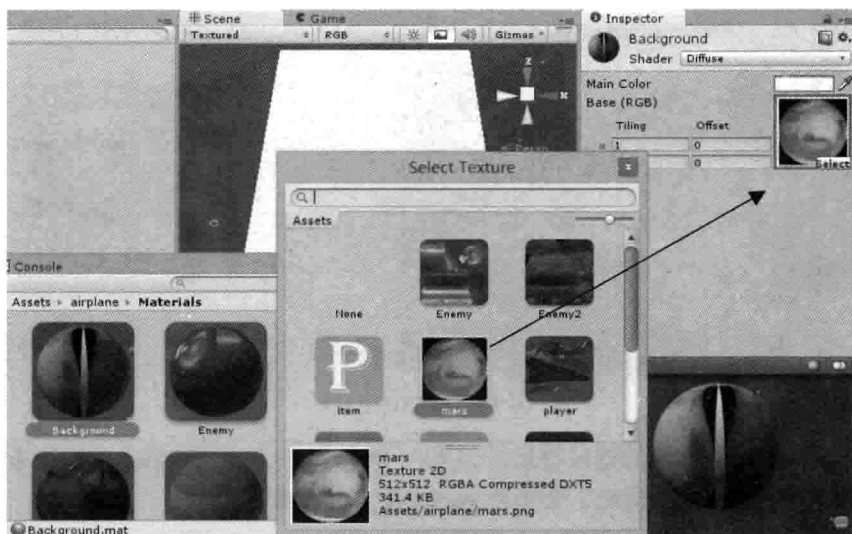


图 2-4 指定贴图

当前指定贴图的方式比较直接，但是当贴图较多时便难以查找。另一种指定贴图的方式是确定材质球处于选择状态，在 Project 窗口浏览到目标贴图，直接将其拖至材质球的贴图框内。

步骤 05 在 Scene 窗口选择火星背景模型，在 Inspector 窗口找到 Materials 下面的 Element 0，选择右边的小圆圈按钮，指定 Background 材质球，如图 2-5 所示。

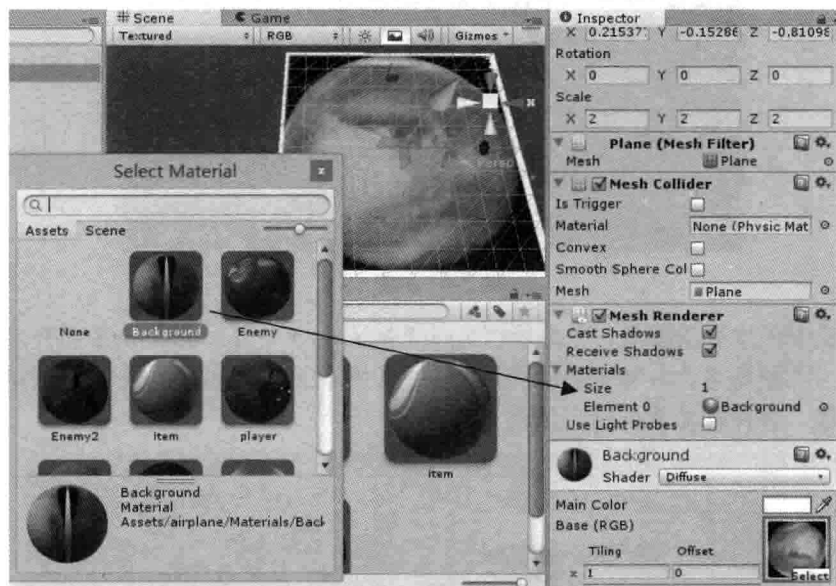


图 2-5 指定材质

也可以直接将 Background 材质球拖至 Scene 窗口的火星背景模型上为其指定材质。在 Unity 中，几乎所有的关联操作都可以通过拖动的方式完成。

步骤 06 我们会发现火星四周黑色的边框很难看，选择 Background 材质球，将 Shader 设为【Transparent】→【Cutout】→【Diffuse】，材质将带有 Alpha 信息，如图 2-6 所示。

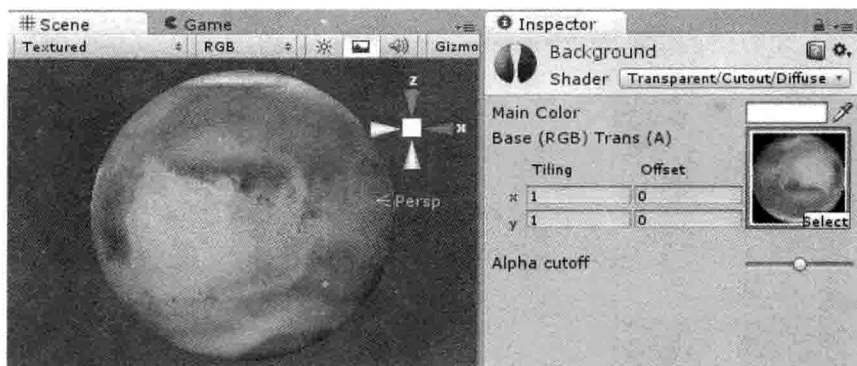


图 2-6 透明材质

这是一个没有半透明效果的透明材质, 如果将 Shader 设为【Transparent】→【Diffuse】可以获得半透明效果, 但在某些情况下会出现 Alpha 乱序的现象。

接下来, 我们将在火星后面创建一个星空背景, 创建星空与创建火星的方式相似, 在此基础上我们将为星空制作一个 UV 动画, 使画面看起来在缓缓向前移动。

步骤 01 创建另一个平面体, 将其放大一些, 置于火星下面作为星空的背景, 然后为其创建一个材质球, 并指定 star.png 作为其贴图, 如图 2-7 所示。

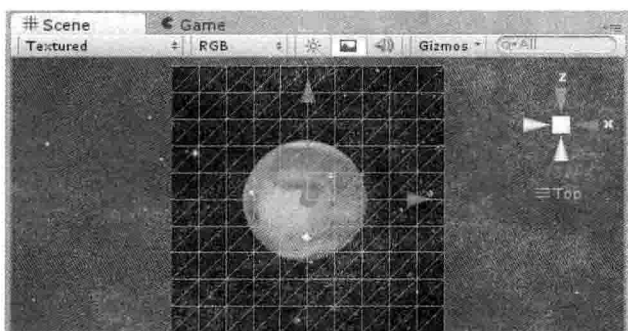


图 2-7 星空

步骤 02 选择星空背景模型, 在菜单栏选择【Window】→【Animation】打开动画窗口, 然后在模型名称右侧的空白框处单击, 选择【Create New Clip】为其创建一个新的动画, 并保存在 Assets 目录内, 如图 2-8 所示。

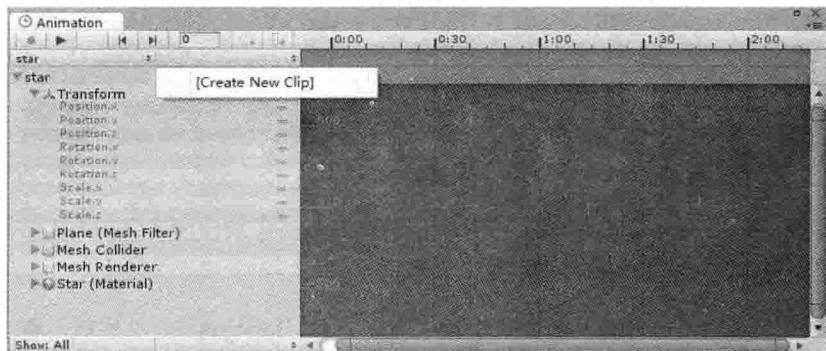


图 2-8 创建新动画

步骤 03 在动画窗口左侧的列表是当前游戏体可以用于动画的属性，找到（Material）将其展开，然后在 Main Tex.offset.v 右侧选择【Add Curve】为 UV 设置动画曲线，如图 2-9 所示。

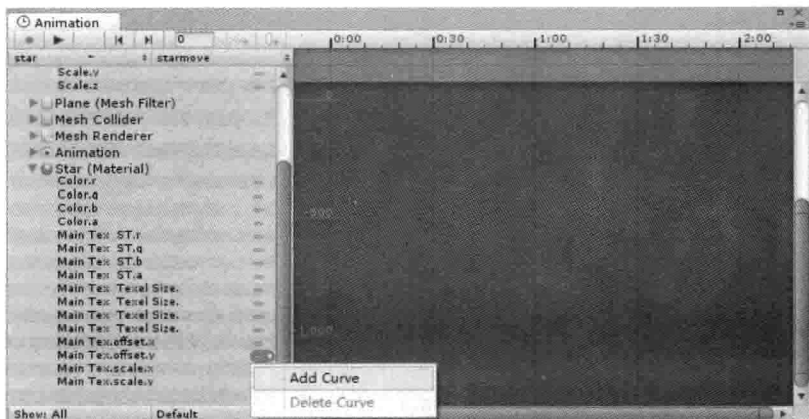


图 2-9 创建动画曲线

步骤 04 在添加动画曲线后，会发现动画窗口左上方的录制按钮被自动激活，这表明现在可以开始录制动画了。将时间轴前进至 30 帧左右，将 Main Tex.offset.v 的值设为 -1，在动画窗口下方选择【Loop】将动画设为循环播放方式，如图 2-10 所示。

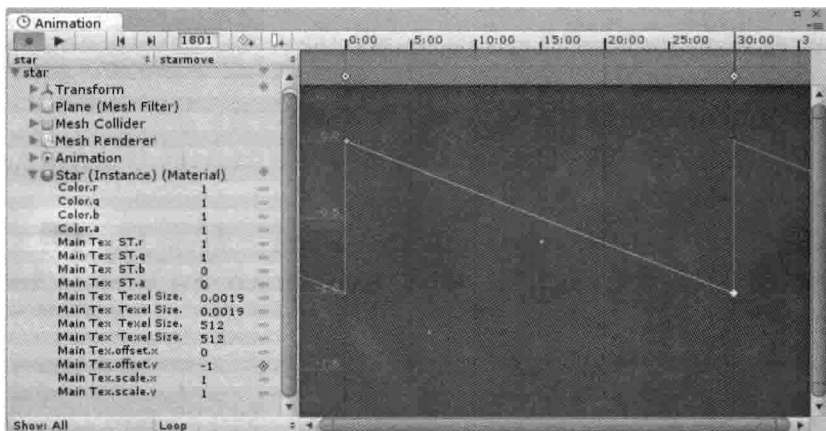


图 2-10 设置动画播放方式

在制作动画的过程中，可以使用动画窗口内的播放功能预览动画，当动画达到满意的效果时关闭动画窗口即可。

2.4.2 设置摄像机和灯光

摄像机是用来展示游戏世界的窗口。在一个游戏场景中，允许同时有多个摄像机，摄像机与其他游戏体一样，可以移动、旋转，用脚本控制等。

在这个太空射击游戏中，摄像机的工作很简单，就是从上向下展望火星就行了。

- 步骤 01** 在 Scene 窗口调整视图角度。
- 步骤 02** 在 Hierarchy 窗口选择 Main Camera，这是场景中默认的摄像机。在菜单栏选择 **【GameObject】→【Align With View】** 使摄像机视角与当前视图一致，如图 2-11 所示。

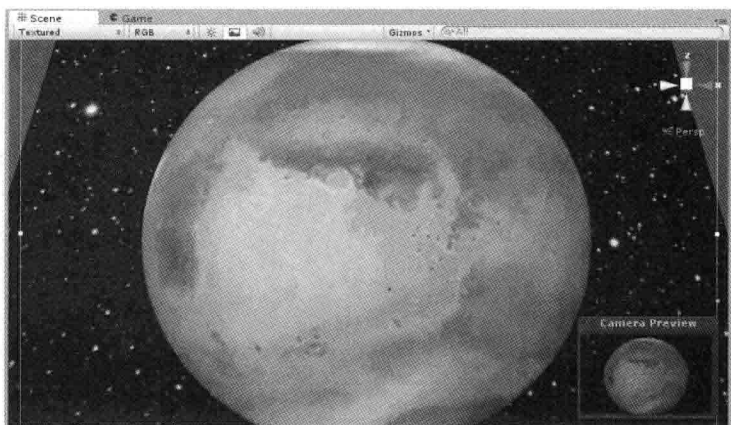


图 2-11 调整视图角度

Hierarchy 窗口是一个名称列表，并可以按子父层级关系排列显示，它们与 Scene 窗口中出现的游戏体是一一对应的。

- 步骤 03** 现在，如果我们运行游戏，可能会发现在 Game 窗口中的画面亮度与 Scene 窗口不一致。在 Scene 窗口上方单击“太阳”按钮，Scene 窗口将使用真实的灯光信息，因为这时场景中还没有灯光，所以画面是比较暗的。
- 步骤 04** 在菜单栏选择 **【Edit】→【RenderSettings】**，然后在 Inspector 窗口选择 Ambient Light，改变它的颜色，适当增加场景亮度。
- 步骤 05** 在菜单栏选择 **【GameObject】→【Create Other】→【Point Light】** 创建一个点光源，将其置于火星模型上方，然后调节其 Range 的值改变灯光范围，调节 Intensity 的值改变灯光强度，如图 2-12 所示。

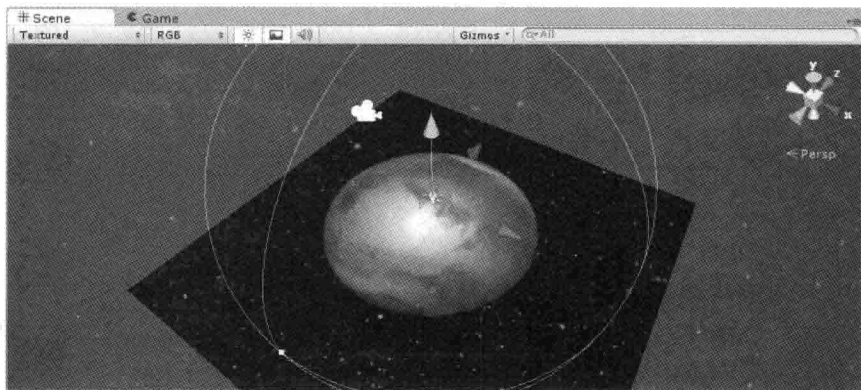


图 2-12 调节灯光

2.5 创建主角

这个游戏的主角是一艘太空飞船，我们将使用一个飞船模型作为主角的游戏体，并赋予它一个脚本，控制它的运动。脚本是实现游戏逻辑的核心，它本身并不能独立运行，它必须作为某个游戏体的组件才能运行。

2.5.1 创建脚本

步骤 01 在 Project 窗口找到 Player.fbx，将其拖动到 Hierarchy 窗口创建飞船模型的游戏体，然后在 Inspector 窗口将它的 Y 轴坐标设为 0，并旋转 180°，如图 2-13 所示。



图 2-13 创建飞船游戏体

步骤 02 在 Project 窗口选择 Assets，单击右键选择【Create】→【Folder】创建一个文件夹，并命名为 Scripts，我们可以将所有创建的脚本都放入到这个文件夹中。

步骤 03 选择 Scripts 文件夹，单击右键选择【Create】→【C# Script】创建一个 C# 脚本，命名为 Player。

步骤 04 双击 Player.cs 脚本将其打开，会发现 Unity 已经自动创建了一些基本代码：

```
using UnityEngine;
using System.Collections;

public class Player : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```


}

Player 是类的名字,这个名字必须与脚本的文件名一致,它默认继承自 MonoBehaviour,只有继承自 MonoBehaviour 的类才能作为 Unity 脚本组件使用。

步骤 05 选择主角的飞船模型游戏体,然后在菜单栏选择【Component】→【Scripts】→【Player】将脚本指定给主角游戏体作为组件。

步骤 06 默认新创建的脚本都会出现在菜单栏【Component】→【Scripts】下面,为了便于管理脚本,我们也可以自定义脚本在菜单栏中的位置。在 Player 类前面添加.AddComponentMenu 属性:

```
[AddComponentMenu("MyGame/Player")]
public class Player : MonoBehaviour
```

步骤 07 在菜单栏选择【Component】,会发现 Player 脚本出现在自定义的 MyGame 子菜单中,如图 2-14 所示。

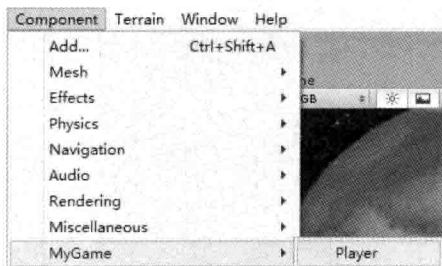


图 2-14 自定义脚本在菜单中的位置

2.5.2 控制飞船移动

我们将使用键盘上的上、下、左、右键来控制飞船的行动。

步骤 01 在 Player 类里添加一个控制飞行速度的属性:

```
public float m_speed = 1;
```

步骤 02 在 Player 类里面,默认有一个 Update 函数,这个函数在程序运行时,每帧都会被调用,我们将在这个函数中添加键盘操作的代码:

```
// Update is called once per frame
void Update () {

    // 纵向移动距离
    float movev=0;

    // 水平移动距离
    float moveh=0;
```

```

// 按上键
if ( Input.GetKey( KeyCode.UpArrow ) )
{
    movev -= m_speed*Time.deltaTime;
}

// 按下键
if ( Input.GetKey( KeyCode.DownArrow ) )
{
    movev += m_speed * Time.deltaTime;
}

// 按左键
if ( Input.GetKey( KeyCode.LeftArrow ) )
{
    moveh += m_speed * Time.deltaTime;
}

// 按右键
if ( Input.GetKey( KeyCode.RightArrow ) )
{
    moveh -= m_speed * Time.deltaTime;
}

// 移动
this.transform.Translate( new Vector3( moveh, 0, movev ) );
}

```

Input 是一个包装了输入功能的类，它包括几乎所有的键盘、鼠标或触控操作函数。

Time.deltaTime 表示每帧的经过时间，那些需要每帧做增减变动的数值都需要乘上 Time.deltaTime。在我们这个例子中，速度与 Time.deltaTime 相乘，表示每帧移动 N 米距离。

this.transform 调用的是游戏体的 Transform 组件，Transform 组件提供的主要功能都是和移动、旋转、缩放游戏体有关的。我们调用了 Translate 函数移动游戏体，其中有一个 Vector3 类型的参数，用来表示 x、y、z 三个方向上的移动距离。

步骤 03 在 Update 函数中每帧都去调用 this.transform 组件，会造成一定的效率问题，我们可以在对象初始化时只调用一次并将其保存起来。Start 函数会在对象被实例化时自动调用一次，类似构造函数，可以在这里做一些初始化工作，如下所示：

```

protected Transform m_transform;

// Use this for initialization
void Start () {

```

```
m_transform = this.transform;
}
```

**提示**

MonoBehaviour 的派生类不能使用构造函数初始化。

步骤 04 在 Update 函数中修改控制移动的代码，将 this.transform 替换为 this.m_transform，这样程序就不用每帧去查找 Transform 组件，提高了程序的运行效率，如下所示：

```
this.m_transform.Translate( new Vector3( moveh, 0, movev ) );
```

步骤 05 运行游戏，应当可以控制飞船移动了，但我们可能会觉得飞船的移动速度不够快。退出游戏运行模式，选择主角飞船游戏体，在 Inspector 窗口找到 Player 脚本组件，将 Speed 的值加大至 3，如图 2-15 所示。

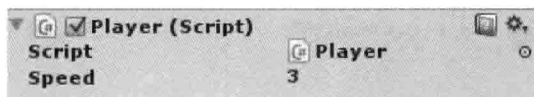


图 2-15 脚本组件

Speed 的值与 Player 脚本内的 m_speed 的值是对应的，这样我们只需要在场景中修改 Speed 的值就可以改变飞船的移动速度，而不用每次都去修改原始的代码。

**提示**

只有 public 类型的属性才能在编辑器窗口实例化。

步骤 06 再次运行游戏，飞船的移动速度明显提高了。

2.5.3 创建子弹

这是一个射击游戏，飞船还需要发射子弹，我们先创建它的游戏体和脚本。

步骤 01 在 Project 窗口找到 rocket.fbx 模型文件，将其拖动到 Hierarchy 窗口创建子弹的游戏体，如图 2-16 所示。



图 2-16 创建子弹游戏体

步骤 02 创建 Rocket.cs 脚本，将其指定给子弹游戏体。

步骤 03 在 Rocket 类中添加 3 个属性，分别控制子弹的飞行速度、生存时间和威力，如下所示：

```
[AddComponentMenu("MyGame/Rocket")]
public class Rocket : MonoBehaviour {

    // 子弹飞行速度
    public float m_speed = 10;

    // 生存时间
    public float m_liveTime = 1;

    // 威力
    public float m_power = 1.0f;

    protected Transform m_trasform;

    // Use this for initialization
    void Start () {
        m_trasform = this.transform;
        Destroy(this.gameObject, m_liveTime);
    }
    ...
}
```

步骤 04 在 Rocket 的 Update 函数中添加如下代码：

```
// Update is called once per frame
void Update () {
    m_trasform.Translate( new Vector3( 0, 0, -m_speed * Time.deltaTime ) );
}
```

注意在 Start 函数中，我们已经调用了 Destroy 函数，其中填入了 m_liveTime 作为参数，表示在一定时间后销毁自身。

步骤 05 运行游戏，子弹将飞速地前进，一定时间后会自动消失。

2.5.4 创建子弹 Prefab

前面我们已经创建了子弹的游戏体，但游戏中的子弹应当是玩家操作发射的，并可以发射很多子弹。对于需要重复使用的游戏体，我们需要将其制作成 Prefab。

Prefab 在 Unity 中可以理解为可重用的游戏体。当我们创建了一个游戏体，为它设置了脚本、参数等，我们可以将其保存为 Prefab，然后在任何时间、场景，甚至在其他 Unity 游戏中重复使用。下面，我们将会把子弹游戏体制作成一个 Prefab。

步骤 01 在 Project 窗口的 Assets 目录内创建一个名为 prefabs 的文件夹用于保存 prefab, 然后单击右键选择【Create】→【Prefabs】, 创建一个空的 Prefab, 将其命名为 Rocket, 如图 2-17 所示。

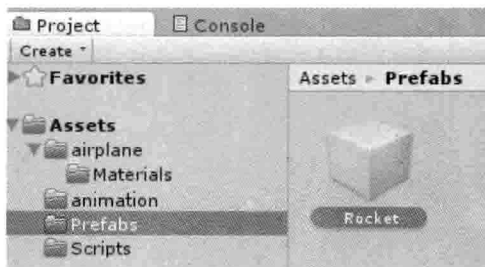


图 2-17 创建 Prefab

步骤 02 在 Hierarchy 窗口选择前面创建的子弹游戏体, 将其拖动到刚刚创建的 prefab 上面, Prefab 的制作就完成了, 如图 2-18 所示。

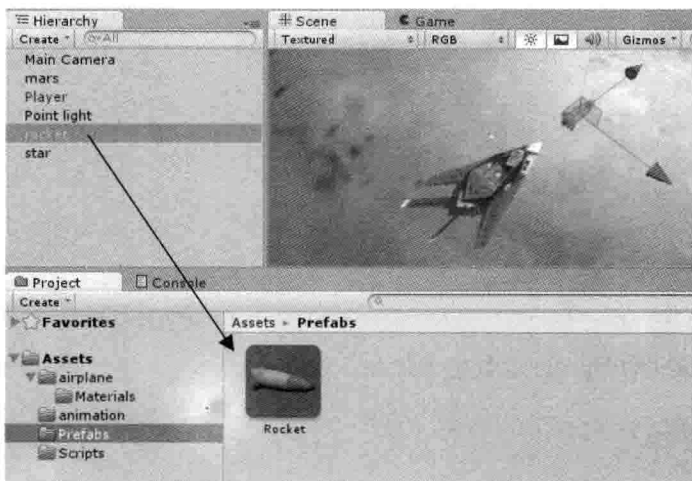


图 2-18 制作 Prefab

步骤 03 场景中的原始子弹游戏体已经不再需要了, 可以按 Delete 键删除它。

2.5.5 发射子弹

接下来, 我们将使用 Instantiate 函数动态的创建子弹游戏体。

步骤 01 首先, 我们要将飞船和子弹的 Prefab 关联起来。打开 Player.cs 脚本, 添加一个 Transform 属性, 它将指向子弹的 Prefab, 如下所示:

```
public Transform m_rocket;
```

步骤 02 回到 Unity 编辑器, 选择 Player 游戏体, 在 Inspector 窗口找到 Player 脚本组件, 会发现多增加了一个 Rocket 选项, 选择子弹的 Prefab, 将其拖动到 Rocket 选项上, 如图 2-19 所示。



图 2-19 关联 Prefab

步骤 03 打开 Player.cs 脚本，在 Update 函数的最下面添加如下代码发射子弹：

```
void Update () {
    ...
    // 按空格键或鼠标左键发射子弹
    if ( Input.GetKey(KeyCode.Space) || Input.GetMouseButton(0) )
    {
        Instantiate( m_rocket, m_transform.position, m_transform.rotation );
    }
}
```



提示

Unity 的游戏体只能使用 Instantiate 函数实例化，不能使用 new。

步骤 04 运行游戏，按空格键或鼠标左键可以发射子弹，但现在的发射速度太快，我们再接再厉修改一下代码，先添加一个控制发射频率的属性：

```
float m_rocketRate = 0;
```

步骤 05 在 Update 函数中修改代码，每隔 0.1 秒发射一次子弹，如下所示：

```
void Update () {
    ...
    m_rocketRate -= Time.deltaTime;

    if ( m_rocketRate <= 0 )
    {
        m_rocketRate = 0.1f;

        if ( Input.GetKey( KeyCode.Space ) || Input.GetMouseButton(0) )
        {
            Instantiate( m_rocket, m_transform.position, m_transform.rotation );
        }
    }
}
```

2.6 创建敌人

创建敌人的方式与创建主角类似, 不过敌人的行为需要由计算机来控制, 它将从上方迎着主角缓慢飞出, 并左右来回移动。

步骤 01 创建 Enemy.cs 脚本, 添加代码如下:

```
[AddComponentMenu("MyGame/Enemy")]
public class Enemy : MonoBehaviour {
    // 移动速度
    public float m_speed = 1;
    // 旋转速度
    protected float m_rotSpeed = 30;

    protected Transform m_transform;

    // Use this for initialization
    void Start () {
        m_transform = this.transform;
    }

    // Update is called once per frame
    void Update () {
        UpdateMove();
    }

    protected void UpdateMove()
    {
        // 左右移动
        float rx = Mathf.Sin(Time.time) * Time.deltaTime;

        // 前进
        m_transform.Translate(new Vector3(rx, 0, -m_speed * Time.deltaTime));
    }
}
```

UpdateMove 函数用来执行敌人的移动, 使用了 Sin 函数使数值在-1~1 之间循环变化, 敌人在向前飞行的同时, 还将左右迂回移动。

步骤 02 在 Project 窗口找到 Enemy.fbx 文件, 为其创建 Prefab, 并指定 Enemy 脚本作为其组件。

步骤 03 将敌人的 Prefab 从 Project 窗口拖动到场景中, 然后按 Ctrl+D 组合键复制多个敌人并摆放在不同位置。运行游戏, 敌人将向前沿弧线来回缓缓移动, 如图 2-20 所示。

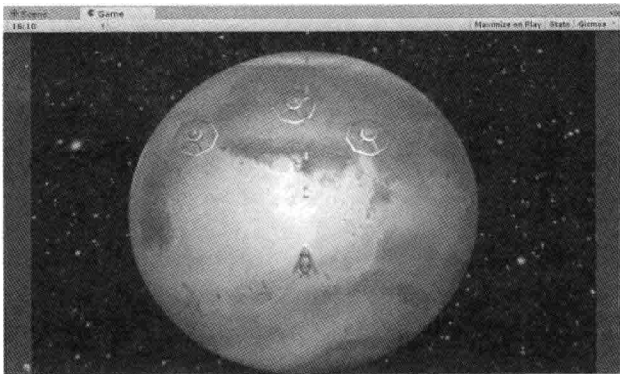


图 2-20 敌人

2.7 物理碰撞

现在的游戏虽然有主角、敌人，也可以发射子弹，但主角和敌人却没有任何交互。接下来我们将分别给主角、子弹和敌人添加碰撞体，并添加触发碰撞的代码，使其在碰撞的时候产生交互。

2.7.1 添加碰撞体

步骤 01 我们先给敌人添加碰撞体。在场景中选择任意一个敌人，在菜单栏选择【Component】→【Physics】→【Box Collider】为敌人添加一个立方体碰撞体组件，然后在 Inspector 窗口找到【Box Collider】组件，选中【Is Trigger】，使其具有触发作用，如图 2-21 所示。

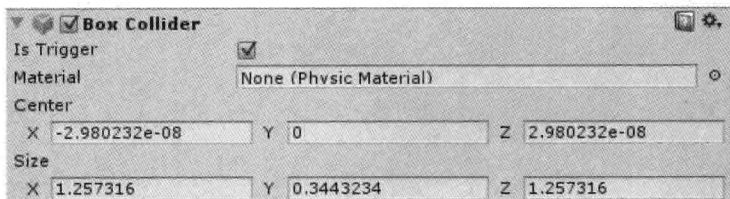


图 2-21 立方体碰撞体选项

步骤 02 接着在菜单栏选择【Component】→【Physics】→【Rigidbody】为敌人添加一个刚体组件，所有需要参与物理计算的游戏体都需要有一个刚体组件才能正常工作。在 Inspector 窗口找到【Rigidbody】组件，取消选择【Use Gravity】去掉重力影响，选中【Is Kinematic】使游戏体的运动不受物理模拟影响，如图 2-22 所示。

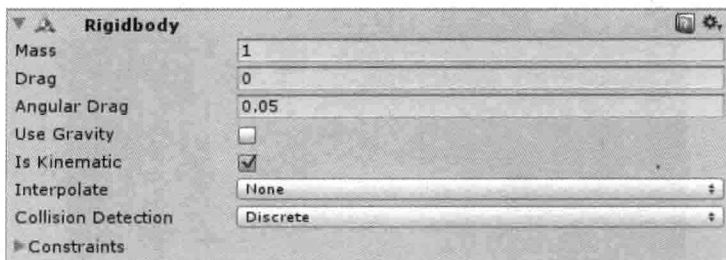


图 2-22 刚体选项

步骤 03 现在,我们已经为场景中的一个敌人添加并设置了相关的物理组件,确定其处于选择状态,在 Inspector 窗口的右下角选择【Apply】,如图 2-23 所示,原始敌人的 Prefab 及场景中的其他敌人会自动更新到当前敌人的设置。

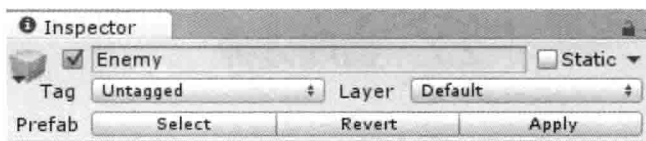


图 2-23 Prefab 选项



提示

如果是在 Project 窗口修改原始敌人 Prefab 的设置,场景中的敌人游戏体会自动更新与 Prefab 一致。

步骤 04 参考前面的步骤为主角和子弹添加物理组件,如图 2-24 所示。

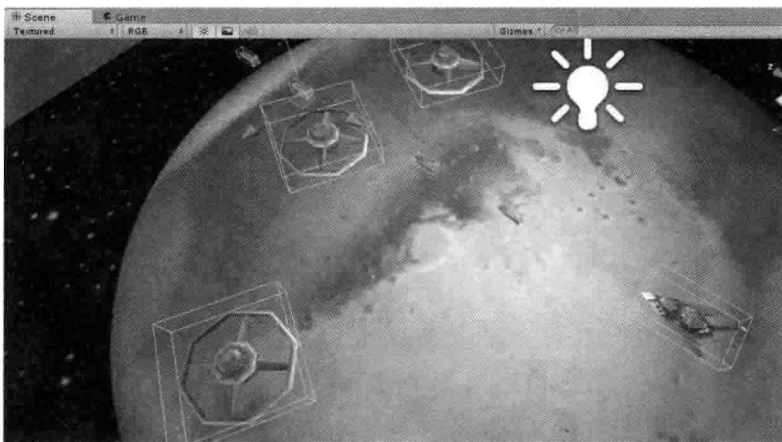


图 2-24 立方体碰撞框

2.7.2 触发碰撞

我们已经为主角、子弹和敌人添加了碰撞体,但还是看不到任何互动效果,我们将要为它们分别指定一个 Tag 标识,然后添加触发碰撞事件的代码。

步骤 01 在菜单栏选择【Edit】→【Project Settings】→【Tags】,在 Size 中输入 2,创建 2

个新的 Tag，名为 PlayerRocket 和 Enemy，如图 2-25 所示。

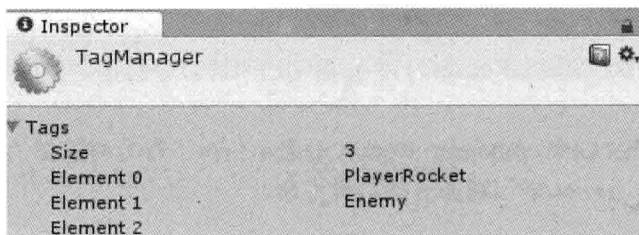


图 2-25 添加 Tag

步骤 02 在 Project 窗口选择敌人的 Prefab，在 Inspector 窗口设置 Tag 为 Enemy，如图 2-26 所示。

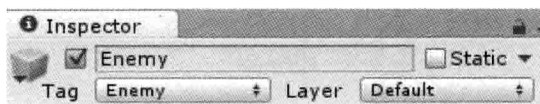


图 2-26 设置敌人的 Tag

步骤 03 选择子弹的 Prefab，设置它的 Tag 为 PlayerRocket。

步骤 04 选择主角，设置它的 Tag 为 Player，这个 Tag 是 Unity 预设在工程内的。

步骤 05 打开 Enemy.cs 脚本，添加一个生命属性：

```
public float m_life = 10;
```

步骤 06 在 Enemy 脚本中添加一个 OnTriggerEnter 函数，这是一个派生自 MonoBehaviour 的函数，在碰撞体互相接触时会被触发，如下所示：

```
void OnTriggerEnter(Collider other)
{
    if (other.tag.CompareTo("PlayerRocket") == 0)
    {
        Rocket rocket = other.GetComponent<Rocket>();
        if (rocket != null)
        {
            m_life -= rocket.m_power;

            if (m_life <= 0)
            {
                Destroy(this.gameObject);
            }
        }
    }
    else if (other.tag.CompareTo("Player") == 0)
    {
        m_life = 0;
        Destroy(this.gameObject);
    }
}
```

```

    }
}

```

`other.tag.CompareTo("PlayerRocket") == 0` 语句用来比较字符串, 判断遇到的碰撞体是否是主角的子弹。

`Rocket rocket = other.GetComponent<Rocket>()` 语句获得了对方碰撞体的 `Rocket` 脚本组件。

`m_life -= rocket.m_power` 语句减去一些自身生命, 当生命为 0 时, 使用 `Destroy` 函数销毁自身。

`other.tag.CompareTo("Player") == 0` 语句用来判断是否与主角相撞, 如果是则直接销毁自身。运行游戏, 敌人已经可以被主角发射的子弹消灭, 如果它与主角相撞, 则直接销毁。

步骤 07 打开 `Rocket.cs` 脚本, 也添加一个 `OnTriggerEnter` 函数, 它的作用很简单, 如果子弹撞击到敌人, 销毁自身。

```

void OnTriggerEnter(Collider other)
{
    if (other.tag.CompareTo("Enemy") != 0)
        return;

    Destroy(this.gameObject);
}

```

步骤 08 打开 `Player.cs` 脚本, 添加一个生命属性, 然后也添加一个 `OnTriggerEnter` 函数, 主角飞船与任何己方子弹以外的碰撞体相撞都会损失一点生命, 生命为零时销毁自身, 如下所示:

```

public float m_life = 3;

void OnTriggerEnter(Collider other)
{
    if (other.tag.CompareTo("PlayerRocket") != 0)
    {
        m_life -= 1;

        if (m_life <= 0)
        {
            Destroy(this.gameObject);
        }
    }
}

```

运行游戏, 我们可以发射子弹消灭敌人, 主角也可以被敌人撞击损坏, 游戏已经有点模样了, 但如果敌人不能被消灭则会一直存活下去, 解决的方法很简单, 只需要在屏幕下方放一个空的碰撞体, 当敌人与它碰撞时使它消失即可。

步骤 09 创建一个空游戏体，为它添加并指定一个名为“bound”的 tag 和物理组件，将它置于屏幕下方，如图 2-27 所示。

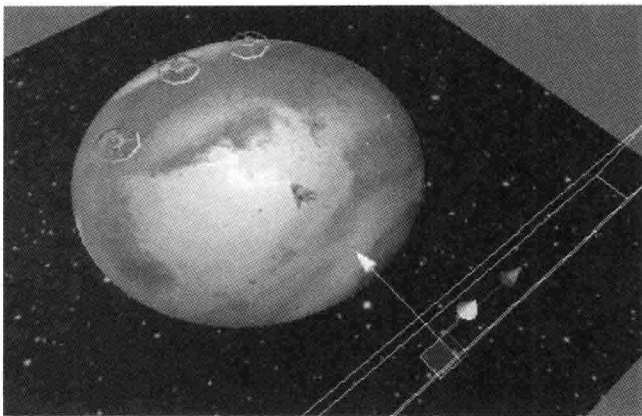


图 2-27 设置边界

步骤 10 修改 Enemy.cs 的 OnTriggerEnter 函数。

```
if (other.tag.CompareTo("bound") == 0)
{
    m_life = 0;
    Destroy(this.gameObject);
}
```

现在，敌人在飞出屏幕外后会自行销毁。

2.8 高级敌人

2.8.1 创建敌人

只是一种敌人太单调了，我们将再添加一种新的敌人，它将继承原有敌人的大部分功能，同时又增加一些新的功能。

步骤 01 在 Project 窗口找到 Enemy2.fbx 模型文件，为其创建 Prefab。



提示

可以在 Project 窗口直接将 FBX 模型拖动到空的 Prefab 文件上创建敌人的 Prefab。

步骤 02 打开 Enemy.cs 脚本，将 protected void UpdateMove() 改为 protected virtual void UpdateMove()，这里添加了一个 virtual，使其成为一个虚方法，这样我们可以在派生类中重写这个方法。

步骤 03 创建 SuperEnemy.cs 脚本，使它的类继承 Enemy，并重写 UpdateMove 方法，新敌

人的移动只是简单的缓缓向前, 如下所示:

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("MyGame/SuperEnemy")]
public class SuperEnemy : Enemy {

    protected override void UpdateMove()
    {
        // 前进
        m_transform.Translate( new Vector3( 0, 0, -m_speed * Time.deltaTime ) );
    }
}
```

在 SuperEnemy 这个类中, 只有 UpdateMove() 一个方法, 这个方法前面使用了 override 标识符, 表示这是一个重写的方法。虽然 SuperEnemy 这个类没有其他东西, 但它继承了 Enemy 类的其他全部功能。

- 步骤 04** 选择新敌人的 prefab, 将 SuperEnemy 脚本指定给它作为组件。
- 步骤 05** 在 Inspector 窗口设置新敌人的 SuperEnemy 组件, 增加生命至 50。
- 步骤 06** 为新敌人的 prefab 添加 Rigidbody 和 Box Collider 组件, 并参考原有敌人的设置进行设置。因为这个新敌人的模型比较大, 用立方体碰撞体会显得很不够精确, 我们可以将碰撞体缩小一些, 如图 2-28 所示。如果我们希望使用精度较高的碰撞体, 最好在 3D 软件中将其建模出来, 然后将其设为 Mesh 碰撞体。
- 步骤 07** 将新敌人的 Tag 设为 Enemy。

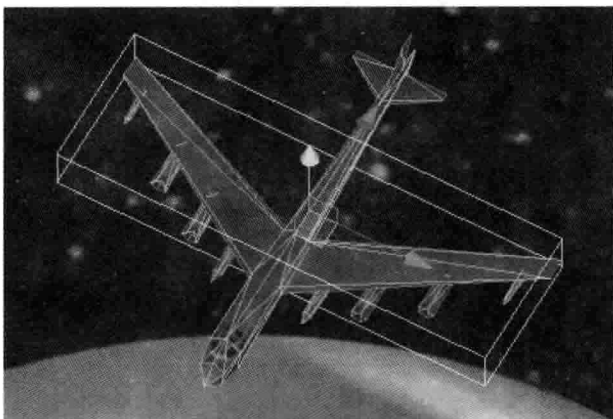


图 2-28 设置 box 碰撞体

2.8.2 发射子弹

将新敌人放到场景中, 运行游戏, 会发现新敌人缓缓向前, 但不会做其他事情。我们接下

来会为其添加一点新功能，使其可以向主角发射子弹，更有威胁。

- 步骤 01** 使用 rocket.fbx 创建一个新的子弹 Prefab，命名为 EnemyRocket，再为其创建一个新的材质，使用 rocket2.png 作为贴图，使敌人的子弹看上去与主角的不同一些。
- 步骤 02** 为敌人子弹添加 Rigidbody 和 Box Collider 组件并正确设置。
- 步骤 03** 为敌人子弹新建一个名为 EnemyRocket 的 Tag。
- 步骤 04** 创建 EnemyRocket.cs 脚本，它将继承 Rocket 类的大部分功能，我们只需要略修改一下 OnTriggerEnter 方法，使其只能与主角飞船发生碰撞，如下所示：

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("MyGame/EnemyRocket")]
public class EnemyRocket : Rocket
{
    void OnTriggerEnter( Collider other )
    {
        if ( other.tag.CompareTo("Player") != 0 )
            return;

        Destroy( this.gameObject );
    }
}
```

- 步骤 05** 将 EnemyRocket 脚本指定给敌人子弹的 Prefab。
- 步骤 06** 在 Inspector 窗口设置 EnemyRocket 组件，降低敌人子弹的移动速度，增加生存时间，如图 2-29 所示。

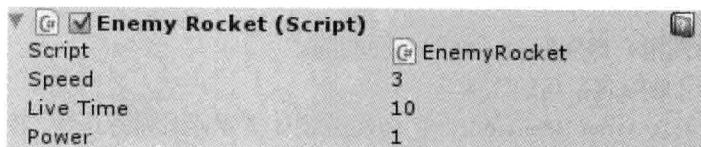


图 2-29 设置敌人子弹

- 步骤 07** 接下来关联新敌人和子弹，打开 SuperEnemy.cs 脚本，修改代码如下：

```
public class SuperEnemy : Enemy {

    public Transform m_rocket;

    protected float m_fireTimer = 2;

    protected Transform m_player;

    void Awake()
```



```

{
    GameObject obj=GameObject.FindGameObjectWithTag("Player");
    if ( obj!=null )
    {
        m_player=obj.transform;
    }
}

protected override void UpdateMove()
{
    m_fireTimer -= Time.deltaTime;
    if (m_fireTimer <= 0)
    {
        m_fireTimer = 2;

        if ( m_player!=null )
        {
            Vector3 relativePos = m_transform.position-m_player.position;
            Instantiate( m_rocket, m_transform.position, Quaternion.LookRotation
(relativePos) );
        }
    }

    // 前进
    m_transform.Translate(new Vector3(0, 0, -m_speed * Time.deltaTime));
}
}

```

m_fireTimer 属性用来控制发射子弹的时间间隔。

m_player 属性用来指向主角的飞船。

Awake 方法继承自 MonoBehaviour, 它会在游戏体实例化时执行一次, 并先于 Start 方法。我们在这里使用 FindGameObjectWithTag 函数获得主角的游戏体实例。

Quaternion.LookRotation 使子弹在初始化时朝向主角的方向。

步骤 08 选择新敌人 Prefab, 在 Inspector 窗口选择 Rocket 属性, 与敌人子弹的 Prefab 关联, 如图 2-30 所示。

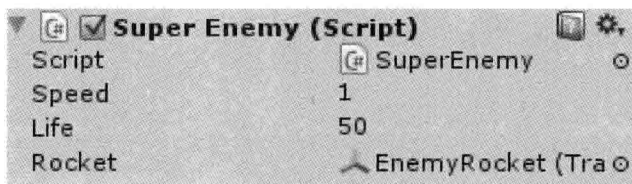


图 2-30 为敌人添加子弹

将新敌人放入场景中, 运行游戏, 新敌人会向主角发射子弹, 如图 2-31 所示。



图 2-31 新敌人的子弹

2.9 声音与特效

现在的游戏是无声的，我们将添加几个简单的音效，并增加一个爆炸效果。

步骤 01 在 Project 窗口单击右键，选择【Import Package】→【Custom Package】，然后到资源文件目录 packages 浏览 Unity 包文件，选择 ShootingFX.unitypackage，将其打开，选择【Import】导入到当前工程中，如图 2-32 所示。在这个包中包含了几个音效文件和一个爆炸特效 Prefab，它们都会被导入到 Assets 目录的 FX 文件夹下。

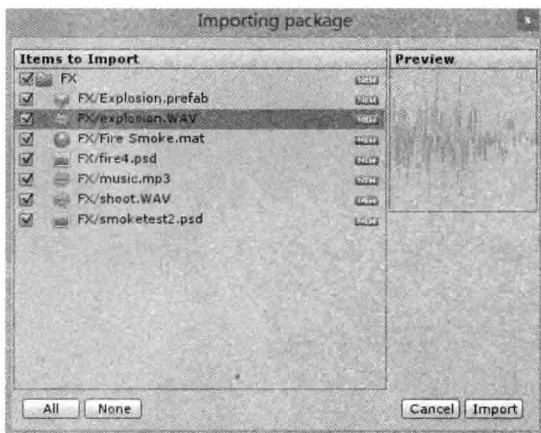


图 2-32 导入音效素材

.unitypackage 文件是 Unity 专用的资源包，在 Project 窗口选择资源，如脚本、模型、声音等任何资源，然后单击右键选择【Export Package】，可以将所选资源导出为.unitypackage 格式的包，然后重用到其他 Unity 工程中。

步骤 02 选择主角飞船游戏体，在菜单栏选择【Component】→【Audio】→【Audio Source】为主角添加一个 Audio Source 组件，凡是需要发声的游戏体，必须有这个组件。

步骤 03 打开 Player.cs 脚本, 添加并修改代码如下:

```
// 声音
public AudioClip m_shootClip;

// 声音源
protected AudioSource m_audio;

// 爆炸特效
public Transform m_explosionFX;

// Use this for initialization
void Start () {

    m_transform = this.transform;

    m_audio = this.audio;
}
```

m_shootClip 属性是射击的声音, 在后面会将它和音效文件关联。

m_audio 属性是声音源组件, 用于播放声音, 在 Start 函数中将其指向实际的声音源组件。

步骤 04 在 Player 脚本的 Update 函数中添加播放声音的代码:

```
if ( Input.GetKey( KeyCode.Space ) || Input.GetMouseButton(0) )
{
    Instantiate( m_rocket, m_transform.position, m_transform.rotation );

    // 播放射击声音
    m_audio.PlayOneShot(m_shootClip);
}
```

步骤 05 在 Player 脚本的 OnTriggerEnter 函数中添加创建爆炸特效的代码:

```
if (m_life <= 0)
{
    // 爆炸特效
    Instantiate(m_explosionFX, m_transform.position, Quaternion.identity);

    Destroy(this.gameObject);
}
```

步骤 06 选择主角游戏体, 在 Project 窗口的 FX 文件夹下分别找到 shoot.WAV 音效文件和 Explosion.prefab 爆炸特效文件, 在 Player 组件中将其分别与 m_shootClip 和 m_explosionFX 属性关联, 如图 2-33 所示。

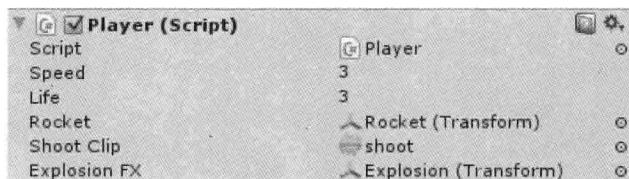


图 2-33 关联爆炸

步骤 07 选择爆炸特效的 Prefab，为其添加一个 Audio Source 组件，然后在 FX 文件夹下找到 explosion.WAV 文件，将其指定到 Audio Source 的 Audio Clip 中，因为默认 Play On Awake 选项是处于选中状态，所以当爆炸特效被实例化后，会自动播放爆炸的声音，如图 2-34 所示。

运行游戏，可以听到射击声音，当主角死亡，会看到爆炸效果，并能听到爆炸声音，如图 2-35 所示。我们也需要为敌人添加同样的爆炸效果和声音，重复前面的步骤即可，这里就不再赘述。

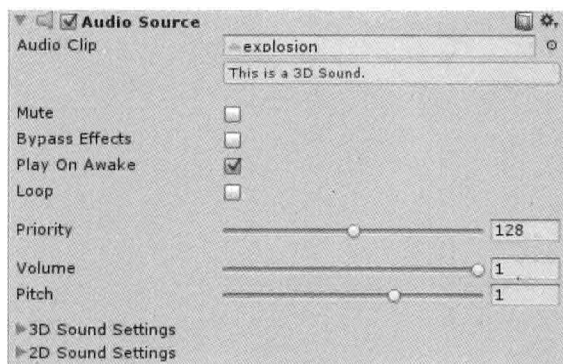


图 2-34 添加爆炸声音

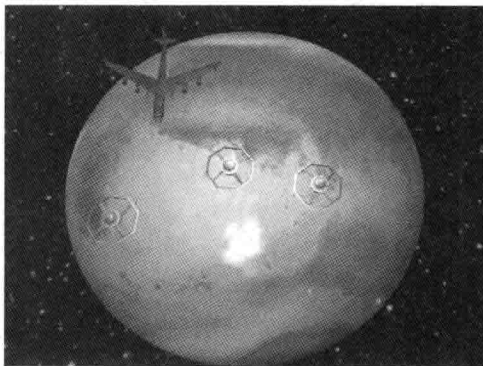


图 2-35 爆炸效果

2.10 敌人生成器

在现在的游戏中，只是随意放了几个敌人，将其消灭后就没有其他敌人了。我们需要创建一个敌人生成器，使其不停地制造新的敌人，这样游戏才能一直玩下去。

步骤 01 创建 EnemySpawn.cs 脚本：

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("MyGame/EnemySpawn")]
public class EnemySpawn : MonoBehaviour
{
    // 敌人的 Prefab
    public Transform m_enemy;
```



```

// 生成敌人的时间间隔
protected float m_timer = 5;

protected Transform m_transform;

// Use this for initialization
void Start () {

    m_transform = this.transform;
}

// Update is called once per frame
void Update () {

    m_timer -= Time.deltaTime;
    if (m_timer <= 0)
    {
        m_timer = Random.value * 15.0f;
        if (m_timer < 5)
            m_timer = 5;

        Instantiate(m_enemy, m_transform.position, Quaternion.identity);
    }
}
}

```

m_enemy 属性指向敌人的 Prefab，我们当前只有两种敌人。

m_timer 属性用来计算生成敌人的时间间隔。

Random.value 会生成 0.0~1.0 之间的一个随机数，在这个例子中，将会在 5~15 秒之间生成一个新的敌人。

步骤 02 在菜单栏选择 **【GameObject】** → **【Create Empty】** 创建一个空的游戏体作为敌人生成器，注意这个空游戏体是看不到的，但它确实存在，可以在 Hierarchy 窗口通过名称选择它。

步骤 03 将 EnemySpawn 脚本指定给敌人生成器游戏体作为组件。

步骤 04 将敌人生成器制作为两个 Prefab，一个 Prefab 的 m_enemy 属性与普通敌人的 Prefab 相关联，另一个与高级敌人相关联。

我们虽然不需要敌人生成器在游戏时显示出来，但如果在场景中也不能显示则不利于摆放它，所幸我们可以让它在场景中以图标的方式显示，在游戏时则看不到它。

步骤 05 在 Project 窗口的根目录创建一个名为 Gizmos 的文件夹，注意这个文件夹的名字必须为 Gizmos。

步骤 06 在 Project 窗口找到图片素材 item.png，将其复制到 Gizmos 的文件夹中，这个图片将作为敌人生成器的图标使用，实际上我们可以使用任何图片。

步骤 07 打开 EnemySpawn.cs 脚本，添加用来显示图标的函数：

```
void OnDrawGizmos ()
{
    Gizmos.DrawIcon ( transform.position, "item.png", true );
}
```

步骤 08 将敌人生成器摆放到场景上方，我们会看到敌人生成器的图标，但在游戏时它们是不显示的。运行游戏，屏幕上方会涌现大量敌人，如图 2-36 所示。

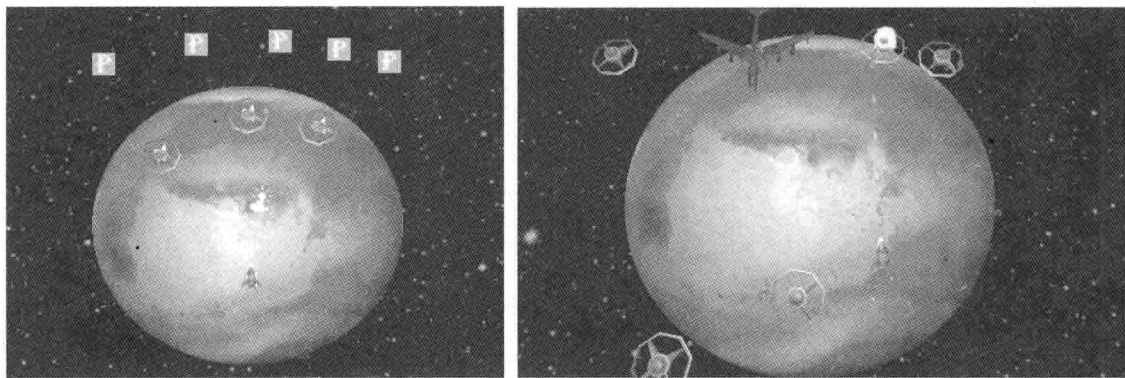


图 2-36 敌人生成器不断生成敌人

2.11 游戏管理器

现在的游戏中还缺少显示游戏信息的 UI 和游戏失败的状态提示，我们将创建一个游戏管理器来处理这些东西。

步骤 01 创建 GameManager.cs 脚本：

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("MyGame/GameManager")]
public class GameManager : MonoBehaviour {

    public static GameManager Instance;

    //得分
    public int m_score = 0;

    //纪录
```

```
public static int m_hiscore = 0;

//主角
protected Player m_player;

// 背景音乐
public AudioClip m_musicClip;

// 声音源
protected AudioSource m_Audio;

void Awake()
{
    Instance = this;
}

// Use this for initialization
void Start () {

    m_Audio = this.audio;

    // 获取主角
    GameObject obj = GameObject.FindGameObjectWithTag("Player");
    if (obj != null)
    {
        m_player = obj.GetComponent<Player>();
    }

}

// Update is called once per frame
void Update () {

    // 循环播放背景音乐
    if (!m_Audio.isPlaying)
    {
        m_Audio.clip = m_musicClip;
        m_Audio.Play();
    }

    // 暂停游戏
    if (Time.timeScale > 0 && Input.GetKeyDown(KeyCode.Escape))
    {

```



```

        Time.timeScale = 0;
    }
}

void OnGUI()
{
    // 游戏暂停
    if (Time.timeScale == 0)
    {
        // 继续游戏按钮
        if (GUI.Button(new Rect(Screen.width * 0.5f - 50, Screen.height * 0.4f, 100, 30),
"继续游戏"))
        {
            Time.timeScale = 1;
        }

        // 退出游戏按钮
        if (GUI.Button(new Rect(Screen.width * 0.5f - 50, Screen.height * 0.6f, 100, 30),
"退出游戏"))
        {
            // 退出游戏
            Application.Quit();
        }
    }

    int life = 0;
    if (m_player != null)
    {
        // 获得主角的生命值
        life = (int)m_player.m_life;
    }
    else // game over
    {
        // 放大字体
        GUI.skin.label.fontSize = 50;

        // 显示游戏失败
        GUI.skin.label.alignment = TextAnchor.LowerCenter;
        GUI.Label(new Rect(0, Screen.height * 0.2f, Screen.width, 60), "游戏失败");

        GUI.skin.label.fontSize = 20;

        // 显示按钮
    }
}

```

```

        if (GUI.Button(new Rect(Screen.width * 0.5f - 50, Screen.height * 0.5f, 100, 30),
"再试一次"))
        {
            // 读取当前关卡
            Application.LoadLevel(Application.loadedLevelName);
        }
    }

    GUI.skin.label.fontSize = 15;

    // 显示主角生命
    GUI.Label(new Rect(5, 5, 100, 30), "装甲 " + life);

    // 显示最高分
    GUI.skin.label.alignment = TextAnchor.LowerCenter;
    GUI.Label(new Rect(0, 5, Screen.width, 30), "纪录 " + m_hiscore);

    // 显示当前得分
    GUI.Label(new Rect(0, 25, Screen.width, 30), "得分 " + m_score);

}

// 增加分数
public void AddScore( int point )
{
    m_score += point;

    // 更新高分纪录
    if (m_hiscore < m_score)
        m_hiscore = m_score;
}
}

```

Instance 是一个静态的句柄，在 Awake 函数中指向自身实例，这样我们就可以在其他类中使用 GameManager。Instance 的形式直接取得 GameManager 实例。注意，GameManager 只能有一个实例。

m_score 和 m_hiscore 属性分别表示当前得分和最高得分纪录，m_hiscore 是一个静态属性，它会一直存在，并不会因为读取关卡而重新初始化。

在 Start 函数中获得主角实例的方式与在 SuperEnemy 类中的方式近似，只是这一次获得的是主角的 Player 脚本组件。

在 Update 函数中，我们循环播放背景音乐，注意播放方式与前面播放射击音效的区别。另外，如果按 Esc 键，将设 Time.timeScale 的值为 0，这时游戏会暂停。

OnGUI 是一个特殊的函数，专门用来画 UI 界面。其中 Rect 函数用来表示 UI 的出现位置和

大小, Label 函数用来书写文字, Button 函数用来显示按钮。

当 Time.timeScale 为 0 时, 屏幕将显示两个按钮, 一个用于继续游戏, 另一个用于退出游戏。Application.Quit 用来退出当前程序。

Application.LoadLevel 用来读取下一个关卡, 这里是重新读取当前关卡。在读取下一个关卡的时候, 当前关卡的游戏体都会被销毁, 如果希望可以保存一些游戏数据, 可以将其设置为静态的属性。

步骤 02 创建空游戏体作为游戏管理器, 为其指定 Audio Source 组件, 然后指定 GameManager.cs 作为其脚本组件。

步骤 03 在资源文件中找到 music.mp3, 把它与游戏管理器的 m_musicClip 属性相关联。

步骤 04 选择 music.mp3, 在 Inspector 窗口取消选中 3D Sound, 使其成为一个 2D 声音, 它的音量将不会受 3D 空间距离影响。默认在场景摄像机上有一个 Audio Listener 组件, 3D 声音的音量取决于声音源与摄像机的距离。

步骤 05 打开 Enemy.cs, 为它添加一个分数属性, 不同敌人的分数可以在 Inspector 窗口中作不同的设置。

```
public int m_point = 10;
```

步骤 06 在 Enemy 的 OnTriggerEnter 函数中添加 GameManager.Instance.AddScore(m_point) 语句, 当敌人被消灭时我们会获得一定分数:

```
if (m_life <= 0)
{
    GameManager.Instance.AddScore(m_point);

    Instantiate(m_explosionFX, m_transform.position, Quaternion.identity);
    Destroy(this.gameObject, 0.1f);
}
```

运行游戏, 最后的效果如图 2-37 所示。

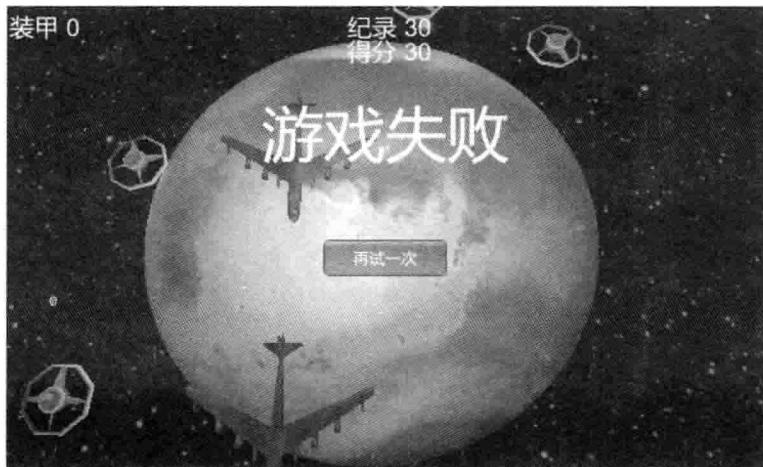


图 2-37 游戏 UI

如果 UI 上的中文显示为乱码, 使用 Visual Studio 打开 GameManager.cs 脚本, 在菜单栏选择【Save】→【Advanced Save Options】, 然后将其保存为 UTF-8 格式即可, 如图 2-38 所示。

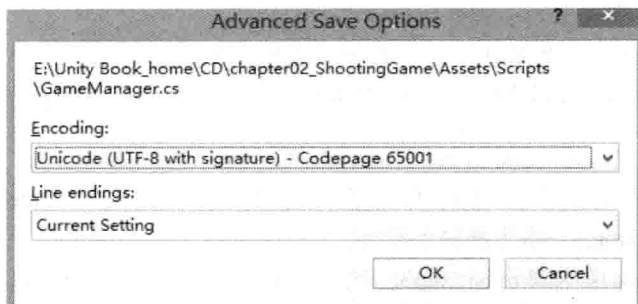


图 2-38 将脚本存储为 UTF-8 格式

2.12 标题界面

游戏中是不是还缺少一个标题画面? 现在的游戏中只有一个关卡, 我们将为它添加另一个关卡, 显示一个简单的标题画面。

步骤 01 在菜单栏选择【File】→【New Scene】创建一个新关卡, 命名为 start 并保存。

步骤 02 创建 TitleScreen.cs 脚本, 代码如下:

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("MyGame/TitleScreen")]
public class TitleScreen : MonoBehaviour
{
    void OnGUI()
    {
        // 文字大小
        GUI.skin.label.fontSize = 48;

        // UI 中心对齐
        GUI.skin.label.alignment = TextAnchor.LowerCenter;

        // 显示标题
        GUI.Label(new Rect(0, 30, Screen.width, 100), "太空大战");

        // 开始游戏按钮
        if (GUI.Button(new Rect(Screen.width * 0.5f - 100, Screen.height * 0.7f, 200, 30),
            "开始游戏"))
        {
```

```
// 开始读取下一关
Application.LoadLevel("level1");
}
}
```

这里的代码很普通，只是显示一个标题和一个按钮，使用 `Application.LoadLevel` 读取关卡的名字。

步骤 03 将 `TitleScenen.cs` 脚本指定给场景中的摄像机作为组件。

步骤 04 在 `Project` 窗口选择图片 `mars.png`，然后在菜单栏选择 **【GameObject】→【Create Other】→【GUI Texture】** 将这张图作为标题画面的 2D 背景，如图 2-39 所示。

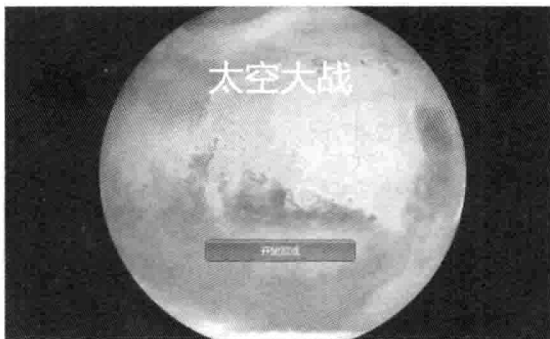


图 2-39 标题画面

2.13 用鼠标控制主角

至今为止，我们的游戏在 PC 上运行一切正常，因为手机不方便使用键盘操作，如果移植到手机上便会遇到问题。接下来，我们将为主角增加鼠标操作的功能，当将游戏移植到手机上后，鼠标操作会自动转换为触屏操作。

步骤 01 因为是 3D 游戏，鼠标的位置只是屏幕坐标，当需要获取到相应的 3D 坐标时通常需要一个参照物，我们将创建一个平面并将它放置在与主角飞船 Y 轴坐标一致的位置作为参照物。在菜单栏选择 **【GameObject】→【Create Other】→【Quad】** 创建一个平面物体。将其的坐标设为 “0, 0, 0”，并旋转 X 轴 90°，然后使用缩放工具将它尽可能拉大，使其能铺满整个屏幕。

步骤 02 默认的平面物体有一个 **【Mesh Collider】** 组件，它会使用模型的三角面进行碰撞计算。在 `Inspector` 面板选中 **【Is Trigger】**，使 `Mesh Collider` 仅作为触发器使用。因为我们只需要平面物体参考位置，并不需要显示它，所以可以取消选中 **【Mesh Renderer】**，如图 2-40 所示。

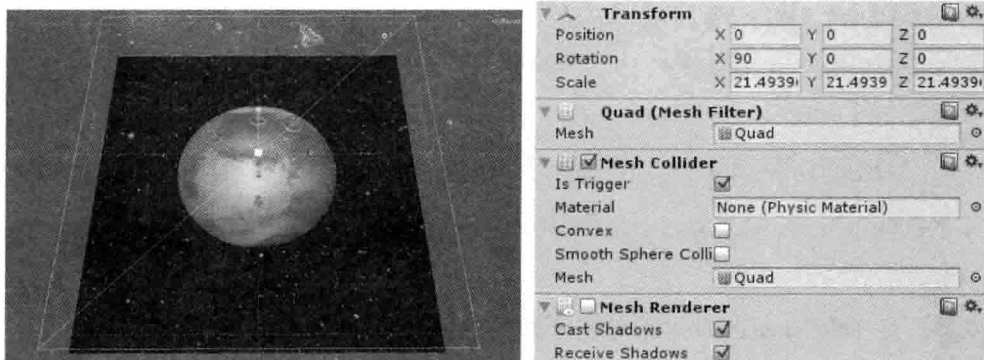


图 2-40 设置平面物体

步骤 03 因为场景中有很多物体都有碰撞体，为了区分，这里需要新建一个层，将平面物体放到这个层中，后面的鼠标操作将产生射线，只与这个层中的平面物体碰撞。先选择 Layer，然后选择 Add Layer，添加一个名为 plane 的层，并将平台物体的层由 Default 改为 plane，如图 2-41 所示。

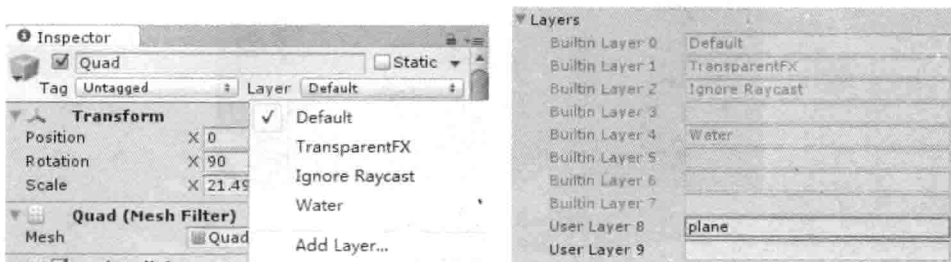


图 2-41 设置层

步骤 04 打开脚本 Player.cs，添加两个新的属性，一个用于定位主角的目标位置，即鼠标单击的位置，另一个用于设置鼠标射线的碰撞层。

```
// 目标位置
protected Vector3 m_targetPos;
// 鼠标射线碰撞层
public LayerMask m_inputMask;
```

步骤 05 在场景中选择主角，将【Input Mask】设置为 plane，这样在接下来的操作中，鼠标单击产生的射线将只与这个层内的物体发生碰撞，如图 2-42 所示。

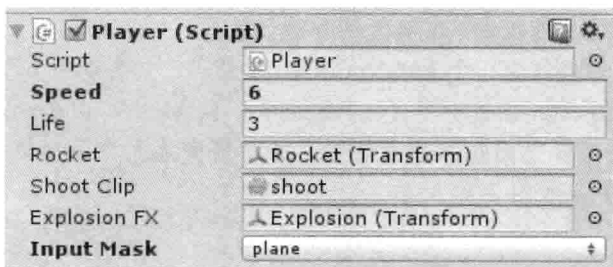


图 2-42 设置层 Mask

步骤 06 打开脚本 Player.cs, 在 Start 函数中将 m_targetPos 的值初始化为当前主角的位置, 然后添加一个 MoveTo 函数, 代码如下所示:

```
void Start () {

    m_transform = this.transform;
    m_audio = this.audio;
    // 初始化目标位置
    m_targetPos = this.m_transform.position;
}

void MoveTo()
{
    if (Input.GetMouseButton(0))
    {
        // 获得鼠标屏幕位置
        Vector3 ms = Input.mousePosition;
        // 将屏幕位置转为射线
        Ray ray = Camera.main.ScreenPointToRay(ms);
        // 用来记录射线碰撞信息
        RaycastHit hitinfo;
        // 产生射线
        bool iscast = Physics.Raycast(ray, out hitinfo, 1000, m_inputMask);
        if (iscast)
        {
            // 如果射中目标, 记录射线碰撞点
            m_targetPos = hitinfo.point;
        }
    }

    // 使用 Vector3 提供的 MoveTowards 函数, 获得朝目标移动的位置
    Vector3 pos = Vector3.MoveTowards(this.m_transform.position, m_targetPos, m_speed
* Time.deltaTime);
    // 更新当前位置
    this.m_transform.position = pos;
}
```

使用 Input.mousePosition 可以获得当前鼠标的屏幕位置, 然后使用 ScreenPointToRay 函数将其转化为一条射线, 使用 Physics.Raycast 函数射出射线, 将碰撞结果返回到 RaycastHit 对象中。Vector3 提供了很多种移动、旋转物体的方法, 这里使用了它提供的 MoveTowards 函数, 计算出从当前位置朝目标位置移动, 返回一个新的位置。

注意, 在 Physics.Raycast 这个函数中, 我们引用了 m_inputMask 参数, 射线只会与 m_inputMask 指定的层中的物体发生碰撞。实际上, 我们完全可以通过层的名称来指定碰撞层, 下面的代码作用与之前是完全一样的, 但省去了在编辑器中指定层的步骤, 如下所示:

```

LayerMask mask =new LayerMask();
mask.value = (int)Mathf.Pow(2.0f, (float)LayerMask.NameToLayer("plane"));
bool iscast = Physics.Raycast(ray, out hitinfo, 1000, mask);

```

最后不要忘了将 MoveTo 函数放到 Update 函数中执行。

现在运行游戏, 主角将会按鼠标的位置移动, 如果将这个游戏移植到手机上, 即是按照手指的位置移动。

2.14 精确的碰撞检测

在前面的制作过程中, 飞机与子弹之间的碰撞都是使用比较简单的 Box 碰撞检测, 但实际飞机和子弹的形状都比较复杂, 这种检测结果会显得很不够精确, 看上去就像是子弹还没碰到飞机, 但已经打中了。

为了能得到更加准确的碰撞检测, 可以直接使用模型本身进行碰撞检测, 但因为模型的多边形数量较高, 这样做对游戏性能会产生很大影响。比较好的做法是使用一个与模型形状近似的碰撞模型, 但尽可能简单, 专门用来检测碰撞, 具体步骤如下:

步骤 01 使用 3D 软件, 比如 3ds Max 或 Maya, 为游戏中使用的模型 Enemy2 创建一个碰撞模型, 如图 2-43 所示。注意, 这个工作通常是由 3D 美术工作者来完成。

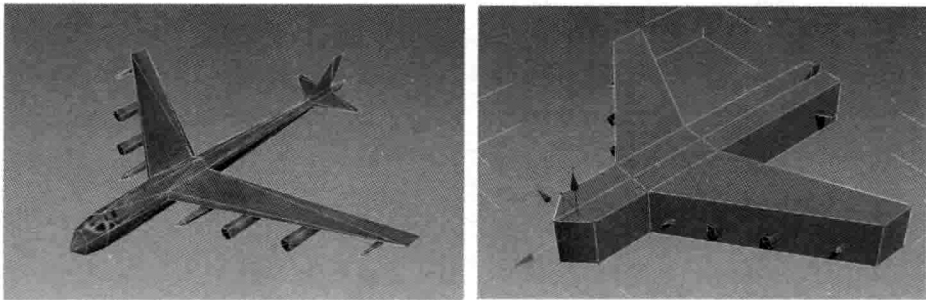


图 2-43 为 Enemy2 创建碰撞模型

步骤 02 为碰撞模型命名, 这里命名为 col, 同时选中游戏模型和碰撞模型, 将它们一起导出为 FBX 文件, 命名为 Enemy2b, 然后再导入到 Unity 中。

步骤 03 参考本章前面的内容, 为 Enemy2b 制作为敌人的 Prefab。注意, 它有两个子物体, 一个名为 col, 即是碰撞模型, 另一个才是游戏中显示的模型。选择碰撞模型, 在菜单栏选择【Component】→【Physics】→【Mesh Collider】为它添加一个 Mesh 碰撞体。因为我们在游戏中并不想看到碰撞模型, 所以将【Mesh Renderer】取消选中, 最后的设置如图 2-44 所示。

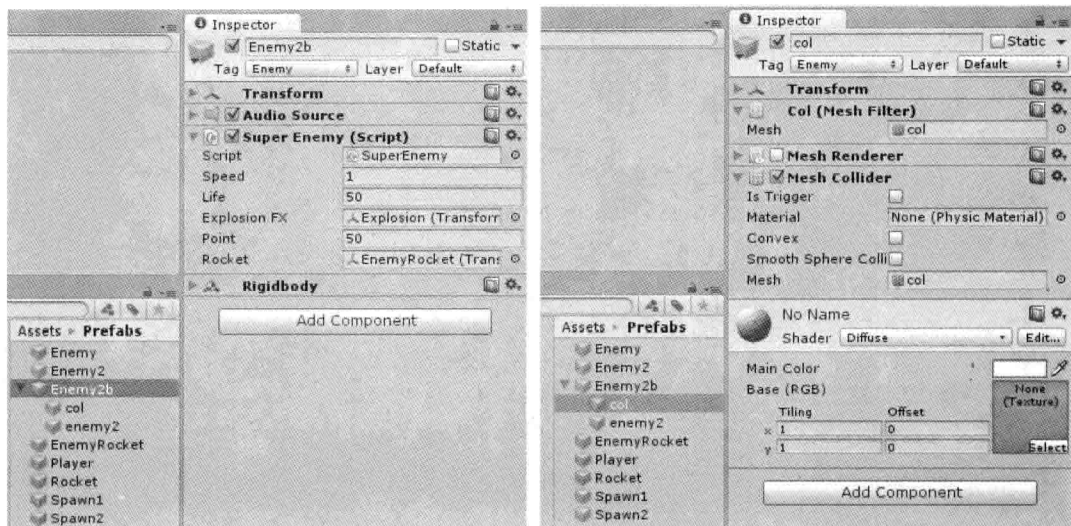


图 2-44 设置 Prefab

将 Enemy2b 替换掉前面创建的 Enemy2，会发现碰撞更加精准了。

2.15 自动创建 Prefab

经过前面的步骤，我们会发现 Unity 提供了可视化工具，在编辑器中甚至不用编写代码便可以自由装配 Prefab，快速实现游戏中需要的某些功能。但这种方式也有缺点，当一个 Prefab 比较复杂时，重新装配或修改 Prefab 需要较多的工作。比如我们要再创建 10 个敌人，就需要手动重复前面的步骤 10 次，工作量很大，也容易出错。

实际上，所有在 Editor 中的手动操作都可以使用代码完成。这样，只需要编写一次代码，很多重复的工作就省掉了。以 Enemy2b 的 Prefab 制作为例，我们将完全使用代码来创建它。

步骤 01 首先，我们要编写一个脚本，它的作用是在当模型被导入到 Unity 工程中时自动将这个模型创建为 Prefab。在工程目录内创建一个名为 Editor 的文件夹，在里面创建脚本 ProcessModel.cs。这个脚本是一个编辑器脚本，它在游戏运行时没有作用，只能在 Editor 文件夹中创建。

步骤 02 编辑脚本 ProcessModel.cs，添加 using UnityEditor，这样就能使用 Unity 的编辑器相关 API。这里定义的 ProcessModel 类继承自 AssetPostprocessor，它专门用来处理资源导入。OnPostprocessModel 方法在导入模型时会被自动调用，其中的参数 GameObject 即是导入的模型。我们在 ProcessModel 方法中将 Enemy2b.fbx 直接制作为一个 Prefab，并命名为 Enemy2c.Prefab，代码如下：

```
using UnityEngine;
using UnityEditor;
using System.Collections;
```

```

class ProcessModel : AssetPostprocessor
{
    void OnPostprocessModel( GameObject input )
    {
        // 只处理名为"Enemy2b"的模型
        if (!input.name.Contains("Enemy2b"))
            return;
        // 设置导入模型的 tag
        input.tag = "Enemy";

        // 查找碰撞模型
        foreach (Transform obj in input.GetComponentsInChildren<Transform>())
        {
            if (obj.name.CompareTo("col") == 0)
            {
                // 取消碰撞模型的显示
                MeshRenderer r = obj.GetComponent<MeshRenderer>();
                r.enabled = false;

                // 添加 Mesh 碰撞体
                obj.gameObject.AddComponent<MeshCollider>();

                // 设置碰撞体的 tag
                obj.tag = "Enemy";
            }
        }

        // 设置刚体
        Rigidbody rigid = input.AddComponent<Rigidbody>();
        rigid.useGravity = false;
        rigid.isKinematic = true;

        // 取得导入模型的相关信息
        ModelImporter importer = assetImporter as ModelImporter;

        // 从工程中将该模型读出来
        GameObject tar = (GameObject)AssetDatabase.LoadAssetAtPath(importer.assetPath,
            typeof(GameObject));

        // 将这个模型创建为 Prefab
        GameObject prefab = (GameObject)PrefabUtility.CreatePrefab
            ("Assets/Prefabs/Enemy2c.Prefab", tar );

        // 为 Prefab 添加声音组件
    }
}

```



```

prefab.AddComponent();

// 获得子弹的 Prefab
GameObject rocket =(GameObject)AssetDatabase.LoadAssetAtPath
("Assets/Prefabs/EnemyRocket.Prefab", typeof(GameObject));

// 获得爆炸效果的 Prefab
GameObject fx =(GameObject)AssetDatabase.LoadAssetAtPath
("Assets/FX/Explosion.Prefab", typeof(GameObject));

// 为 Prefab 添加脚本
SuperEnemy enemy=prefab.AddComponent<SuperEnemy>();
// 设置脚本的默认参数
enemy.m_life = 50;
enemy.m_point = 50;
enemy.m_rocket= rocket.transform;
enemy.m_explosionFX = fx.transform;
}
}

```

重新导入 Enemy2b.fbx, ProcessModel.cs 会被自动调用, 并自动创建 Enemy2c.Prefab, 它的所有功能和设置均与前面手工设置的 Prefab 相同。

2.16 发布游戏

最后, 我们将把游戏打包成标准的.exe 文件, 这个游戏就完成了。

步骤 01 在菜单栏选择【Edit】→【Project Settings】→【Player】, 然后设置游戏的名称、公司名和图标, 如图 2-45 所示, 无论游戏发布在哪一个平台上, 这几个选项都是一样的。

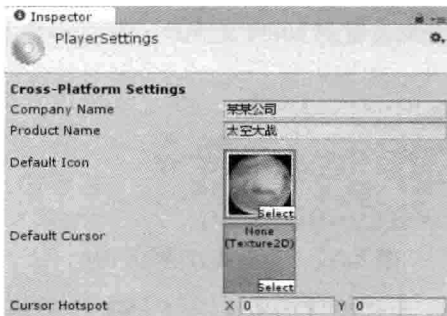


图 2-45 设置游戏名称和默认图标

步骤 02 在【Resolution】选项组中设置游戏窗口的大小。如果选中【Run In Background】, 即使游戏窗口失去焦点, 它也会在后台运行。如果在【Display Resolution Dialog】

中选择【Disabled】，游戏在每次启动时则不会显示出一个用于设置显示分辨率的窗口，如图 2-46 所示。

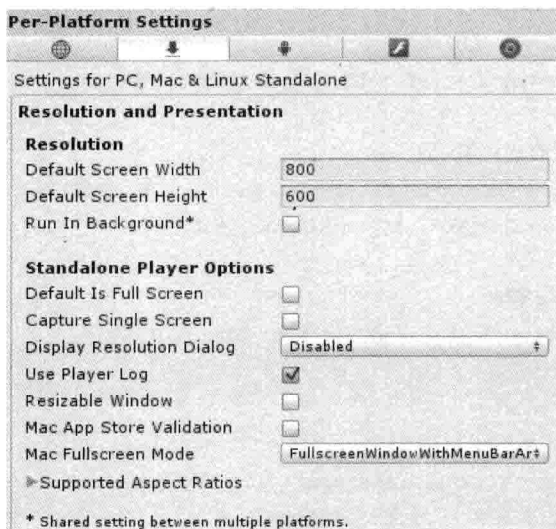


图 2-46 设置游戏窗口大小

步骤 03 在 Icon 选项组中可以设置不同大小的图标，如果不指定，它们将自动调用 Default Icon 中设置的图标并自动缩放，但自动缩放的图标可能会有锯齿，如图 2-47 所示。

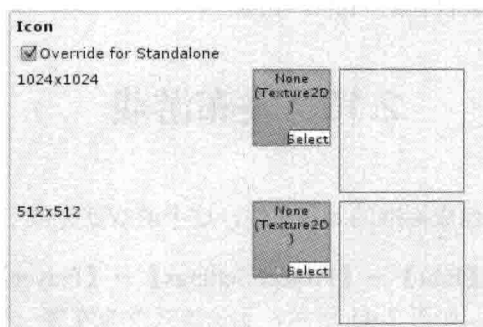


图 2-47 设置图标

步骤 04 默认启动 Unity 游戏会看到 Unity 的商标，在 Splash Image 中指定一张图片即可以显示自定义的图片，如图 2-48 所示。这个功能只有在 Unity Pro 版中才能使用。



图 2-48 设置游戏启动画面图片

步骤 05 在 Other Settings 中设置【Rendering Path】，大部分情况我们可以选择默认的【Forward】。【Static Batching】是 Unity Pro 版才有的功能，它会将静态模型整合，根据资源使用方式不同，有时会使游戏尺寸变大，有时会使变小，如图 2-49 所示。

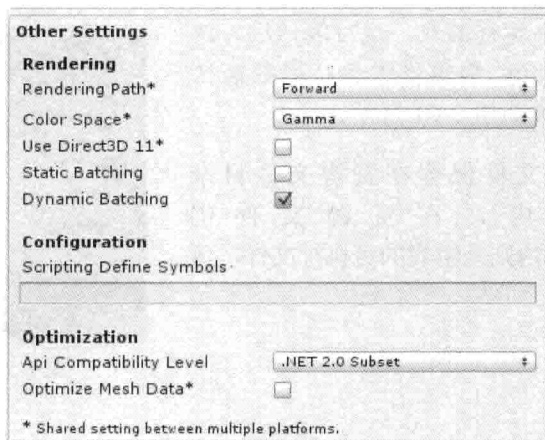


图 2-49 其他设置

在【Rendering Path】中可以设置 3 种渲染模式：

- **【Deferred Lighting】**有最真实的灯光效果，可以不受数量限制地使用灯光和实时阴影，所有的灯光都支持 per-pixel（用于计算 normal maps），该模式当前不支持抗锯齿和半透明显示。目前只有 Unity Pro 才支持这种模式，需要显卡支持 Shader Model 3.0 或更高，不支持手机平台。
- **Forward Rendering** 以 Shader 为基础，只能用一个方向光显示实时阴影。在菜单栏选择 **【Edit】→【Project Settings】→【Quality】**，在 Pixel Light Count 里可以设置 per-pixel 灯光的最大数量。如果将灯光的 Render Mode 设为 Not Important，它将总是一个 per-vertex 或 SH 灯光；如果设为 Important 则总是一个 per-pixel 灯光。
- **Vertex Lit** 是灯光效果最简单的渲染模式，不支持实时阴影，不支持 normal maps，但有最好的性能，适合在一些较差的硬件上使用。

步骤 06 打开 Build Settings 窗口，从 Project 窗口将 .unity 关卡文件拖动到 Build Settings 窗口中，注意要将 start.unity 放在第一个位置，游戏启动后才会先进入标题画面，如图 2-50 所示，然后选择 **【Build】** 将游戏编译为 .exe 格式的标准 Windows 执行程序。

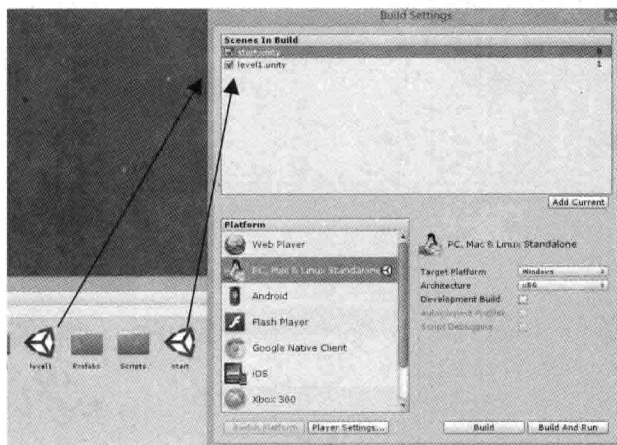


图 2-50 添加关卡

步骤 07 双击.exe 文件运行游戏, 运行编译后的游戏会比在 Unity 编辑器中运行帧数高很多, 最终效果如图 2-51 所示。

本游戏的最终工程文件保存在资源文件目录 \chapter02_ShootingGame 内, 在资源文件的 \builds\shooting game 目录内则是游戏的可执行文件, 可以直接运行游戏。

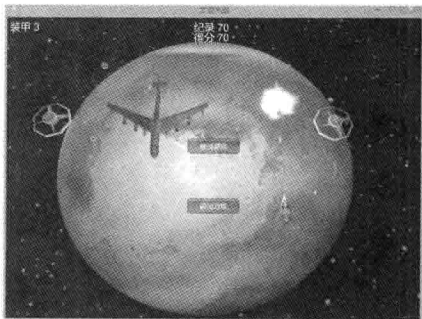


图 2-51 运行发布的游戏

2.17 代码优化

通常创建和销毁游戏实例的代码比较影响游戏的性能, 因为每次都要重新分配内存, 当销毁的时候还需要进行垃圾回收。

在 Unity 中创建一个游戏体实例是使用 Instantiate 函数, 因此, 应当尽可能避免频繁使用这个函数。在这个太空射击游戏实例中, 我们使用这个函数来创建子弹, 因为子弹的创建频率很高而且经常用到, 一个比较好的优化办法是在游戏开始之前一次创建多个子弹, 将它们放到看不到的位置隐藏起来, 当需要时再放到需要的位置显示出来。又或者采用只创建不销毁的策略, 并需要销毁的实例放到看不到的位置隐藏, 需要时再用。这些功能最好通过一个专门的类来管理实现, 下面是一个简单的示例:

```
public class CachedObject : MonoBehaviour
{
    // 是否在使用中
    public bool inUse { set; get; }
    // 资源
    protected GameObject _asset = null;
    // 资源中包括的粒子
    protected ParticleSystem[] _ps;

    // 创建一个缓存资源
    public static CachedObject Create(string assetname, GameObject prefab)
    {
        GameObject go = new GameObject();
        go.name = "cache_" + assetname;
        CachedObject co = go.AddComponent<CachedObject>();
        co.Init(assetname, prefab);
        return co;
    }
}
```

```

// 缓存一个游戏体
void Init(string assetname, GameObject prefab)
{
    inUse = false;
    // 创建资源并移动到看不到的地方
    Vector3 pos = new Vector3(10000, 10000, 10000);
    _asset = (GameObject)GameObject.Instantiate(prefab, pos, Quaternion.identity);
    _ps = _asset.GetComponentInChildren<ParticleSystem>();
    // 隐藏这个资源
    _asset.SetActive(false);
}

// 在游戏中创建资源 (将隐藏的资源显示出来)
public GameObject CreateAsset(Vector3 pos, Vector3 euler )
{
    if (_asset == null)
    {
        return null;
    }
    inUse = true;
    this.transform.position = pos;
    this.transform.eulerAngles = euler;
    // 显示资源
    _asset.SetActive(true);
    // 显示资源中的粒子
    foreach (ParticleSystem ps in _ps)
    {
        ps.Play(true);
    }

    return _asset;
}
/// <summary>
/// 清除缓存资源 (不是真的清除 只是隐藏了)
/// </summary>
public GameObject DestroyAsset()
{
    if (_asset == null)
    {
        return null;
    }
    inUse = false;
    this.transform.position = new Vector3(10000, 100000, 10000);
    this.transform.eulerAngles = Vector3.zero;
}

```

```

        // 隐藏
        _asset.SetActive(false);
        // 停止粒子运动
        foreach (ParticleSystem ps in _ps)
        {
            ps.Stop(true);
        }
        return _asset;
    }
}

```

使用这个类的方式如下:

```

// 初始化载入资源
CachedObject cached = CachedObject.Create("assetname", someprefab);
// 在需要的时候显示缓存资源
cached.CreateAsset(Vector3.zero, Vector3.zero);
// 在不需要的时候隐藏缓存资源
cached.DestroyAsset();

```

除了创建资源, Mono 的垃圾回收也是影响性能的原因之一。在 C#语言中, 没有关键字 **Delete**, 这意味着我们不需要显示的删除对象, 系统会自动获取到不再使用的对象, 将它们清除, 这个过程可能会造成性能下降。因为游戏中通常需要比较稳定的帧数, 一个折中的办法是定时强制进行垃圾回收操作:

```

if (Time.frameCount % 30 == 0) // 每 30 帧强制清理一次
{
    System.GC.Collect();
}

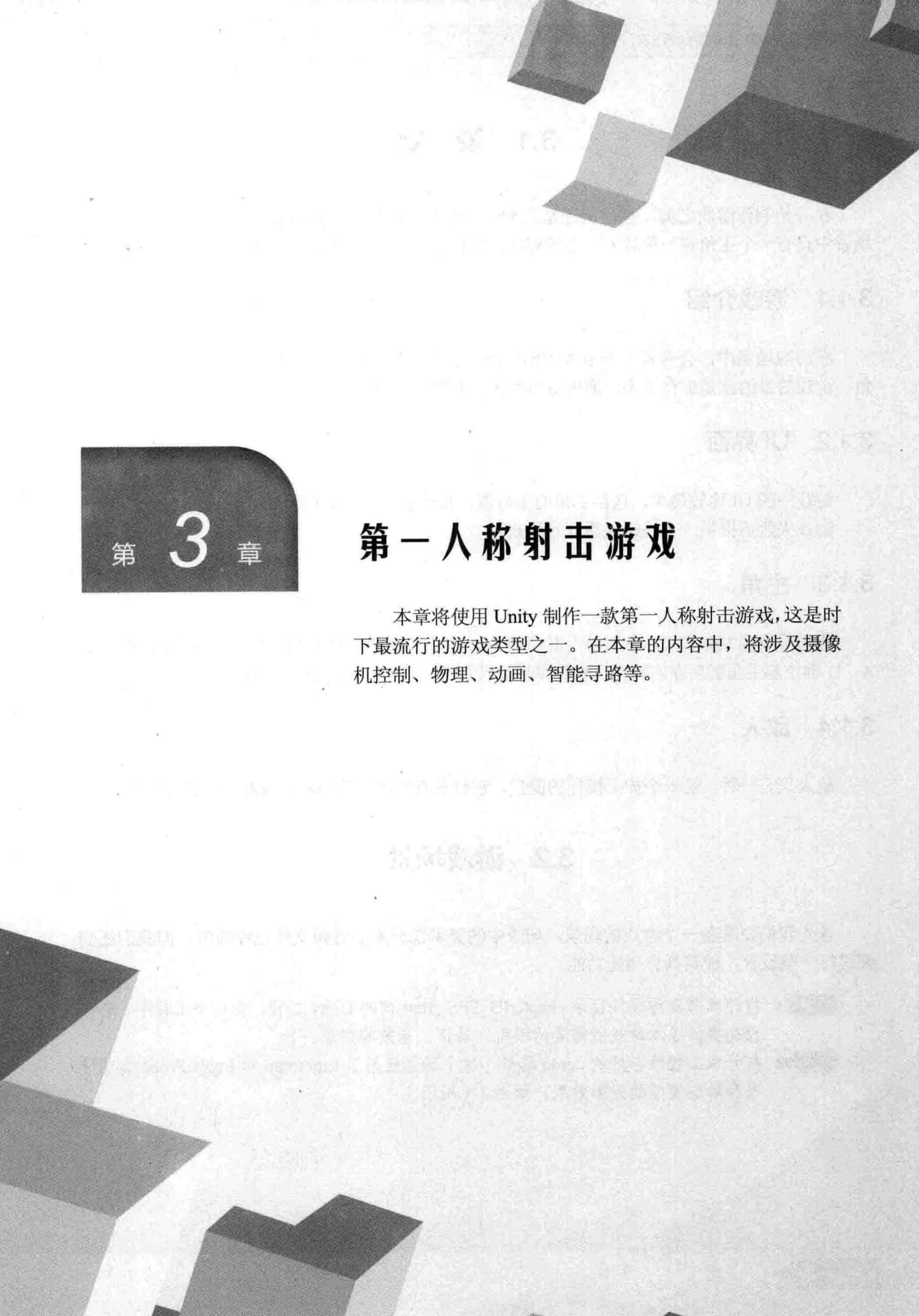
```

小 结

本章通过一个太空射击游戏实例, 一步步介绍了制作 Unity 游戏的很多基础内容, 包括 Unity 编辑器的基本使用、如何管理资源、编写脚本、添加组件、使用物理功能等, 最后介绍了略深入一些的内容, 包括创建精准的碰撞模型、自动创建 Prefab 等。

在完成本章的教程后, 我强烈建议初学者打开 Unity 3D 的帮助文档, 对应查找教程中用到的 API 说明, 相信会有很大的收获。

如果与专业的商业游戏比较, 这个太空射击游戏还比较简单, 但我们已经知道足够的内容, 可以为它添加更多游戏要素和细节, 如加强武器威力的道具, 各种不同行为的敌人, 更完善的游戏流程等, 而这只取决于我们的想象力。



第 3 章

第一人称射击游戏

本章将使用 Unity 制作一款第一人称射击游戏,这是时下最流行的游戏类型之一。在本章的内容中,将涉及摄像机控制、物理、动画、智能寻路等。

3.1 策 划

在开始制作游戏之前,按惯例还是先准备一份游戏策划。我们将要制作的游戏并不复杂,场景中只有一个主角和一种敌人,尽管如此,它已经具备了第一人称射击游戏的基本要素。

3.1.1 游戏介绍

在游戏场景中,会有若干个敌人的出生点,它会定时生成一些敌人,它们会寻找并攻击主角。游戏的目的就是生存下去,消灭更多敌人,获取更多分数。

3.1.2 UI 界面

游戏中的 UI 比较简单,包括主角的生命值、弹药数量、得分和瞄准星。
游戏失败后提供一个按钮重新开始游戏。

3.1.3 主角

我们看不到主脚本人,在屏幕上看到的是一支端在胸前的 M16 机关枪。按键盘的 W、S、A、D 键控制主角前后左右行动,移动鼠标旋转视角,单击鼠标左键进行射击。

3.1.4 敌人

敌人只有一种,是一个护士模样的僵尸,它将具有智能寻路功能,躲避障碍物并攻击主角。

3.2 游戏场景

首先我们要准备一个游戏的场景,场景中的美术部分本书资源文件已经提供,但我们还需要进行一些设置,使其具有碰撞功能。

- 步骤 01** 打开本书资源文件目录 chapter03_FPS_Start 内的 Unity 工程,在这个工程中,已经预先提供了本游戏所需要的模型、动画、音效等资源。
- 步骤 02** 打开本工程准备好的 demo 场景,这个场景使用了 Lightmap 和 Light Probe 表现静态和动态模型的光影效果,如图 3-1 所示。

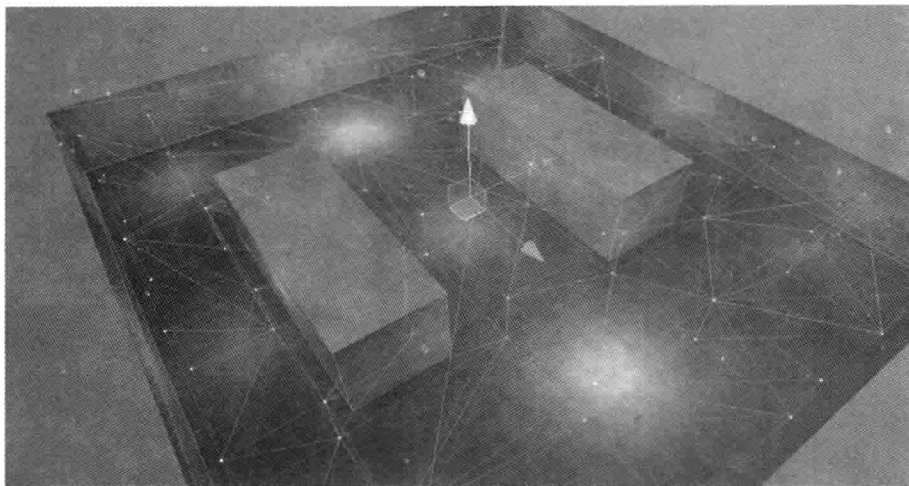


图 3-1 游戏场景

步骤 03 选择三个场景模型，如图 3-2 所示，然后在菜单栏选择【Component】→【Physics】→【Mesh Collider】为其添加多边形碰撞体组件。现在，场景模型既可以用于显示，也可以用于物理碰撞。在实际项目中，模型通常比较复杂，这时就需要做两组模型，一组用于显示，模型有较高的质量，另一组专门用于物理碰撞，为了提高性能，碰撞模型相对做得比较简单，在第 2 章我们已经使用过这种技术。

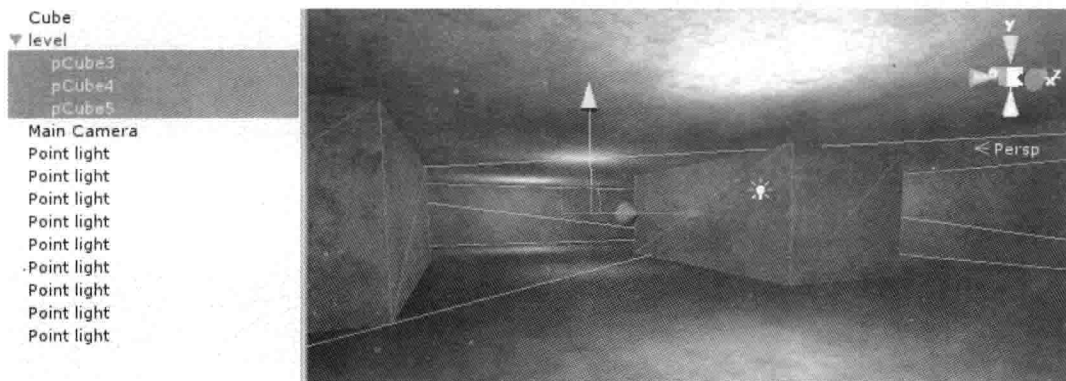


图 3-2 游戏场景

3.3 主 角

因为是第一人称射击游戏，所以主角是不可见的，但在屏幕上可以看到主角手中拿的枪，尽管如此，我们还是需要为主角创建碰撞体并控制其移动。

3.3.1 角色控制器

步骤 01 在菜单栏选择【GameObject】→【Create Empty】创建一个空的游戏体，将它的 Tag

设为 Player, 它便是我们的主角。

步骤 02 在菜单栏选择【Component】→【Physics】→【Character Controller】为主角添加一个角色控制器组件, 使用这个组件提供的功能, 我们将可以实现在控制主角移动的同时与场景的碰撞产生交互, 比如在行走时不会穿到墙里面去。

步骤 03 为主角再添加一个 Rigidbody 组件, 取消选择【Use Gravity】去掉重力模拟, 并选中【Is Kinematic】使其不受物理演算影响, 这样我们才能使用脚本控制其移动。在 Character Controller 组件中需要调整碰撞体的位置和大小, 如图 3-3 所示。

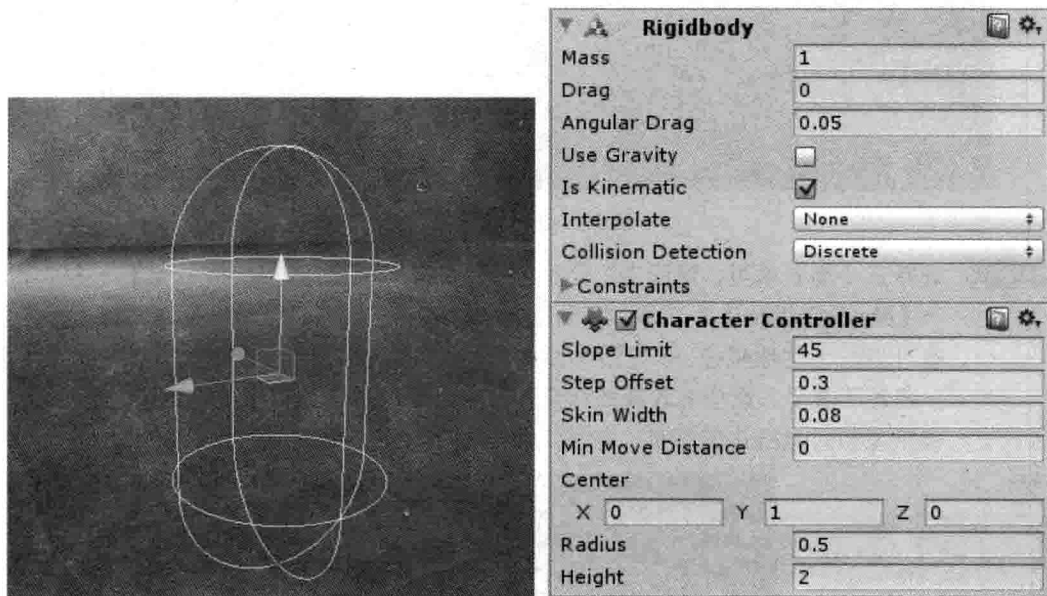


图 3-3 设置组件

步骤 04 创建脚本 Player.cs:

```
using UnityEngine;
using System.Collections;

public class Player : MonoBehaviour {
    public Transform m_transform;
    // 角色控制器组件
    CharacterController m_ch;

    // 角色移动速度
    float m_movSpeed = 3.0f;

    // 重力
    float m_gravity = 2.0f;

    // 生命值
    public int m_life = 5;
```

```
void Start () {
    m_transform = this.transform;
    // 获取角色控制器组件
    m_ch = this.GetComponent<CharacterController>();
}

void Update () {
    // 如果生命为0, 什么也不做
    if (m_life <= 0)
        return;
    Control();
}

void Control()
{
    // 定义3个值控制移动
    float xm = 0, ym = 0, zm = 0;

    // 重力运动
    ym -= m_gravity*Time.deltaTime;

    // 上下左右运动
    if (Input.GetKey(KeyCode.W)){
        zm += m_movSpeed * Time.deltaTime;
    }
    else if (Input.GetKey(KeyCode.S)){
        zm -= m_movSpeed * Time.deltaTime;
    }

    if (Input.GetKey(KeyCode.A)){
        xm -= m_movSpeed * Time.deltaTime;
    }
    else if (Input.GetKey(KeyCode.D)){
        xm += m_movSpeed * Time.deltaTime;
    }

    // 使用角色控制器提供的 Move 函数进行移动 它会自动检测碰撞
    m_ch.Move( m_transform.TransformDirection(new Vector3(xm, ym, zm)) );
}

// 在编辑器中为主角显示一个图标
void OnDrawGizmos()
{
    Gizmos.DrawIcon(this.transform.position, "Spawn.tif");
}
```

```
}
```

这里的代码主要是控制主角前后左右移动。

在 Start 函数中,我们先获取了 CharacterController 组件,然后在 Control 函数中,通过键盘操作获得 X 和 Y 方向上的移动距离,最后使用 CharacterController 组件提供的 Move 移动主角。使用 CharacterController 提供的功能进行移动,在移动的同时会自动计算移动体与场景之间的碰撞。

在 OnDrawGizmos 函数中,用一个图标来显示主角,主要是为了观察方便。

将 Player.cs 脚本指定给主角的游戏体。运行游戏,按 W、S、A、D 键可以控制主角前后左右移动,但可能不易观察到,这是因为摄像机还没有与主角的游戏体关联起来。

3.3.2 摄像机

接下来,我们将在 Player.cs 脚本中添加一些代码,使摄像机伴随着主角移动。

步骤 01 打开 Player.cs,添加用于控制摄像机的属性:

```
// 摄像机 Transform
Transform m_camTransform;

// 摄像机旋转角度
Vector3 m_camRot;

// 摄像机高度(即表示主角的身高)
float m_camHeight = 1.4f;
```

步骤 02 修改 Start 函数,初始化摄像机的位置和旋转角度,并锁定鼠标:

```
void Start () {

    m_transform = this.transform;
    // 获取角色控制器组件
    m_ch = this.GetComponent<CharacterController>();

    // 获取摄像机
    m_camTransform = Camera.main.transform;

    // 设置摄像机初始位置
    Vector3 pos = m_transform.position;
    pos.y += m_camHeight;
    m_camTransform.position = pos;
    // 设置摄像机的旋转方向与主角一致
    m_camTransform.rotation = m_transform.rotation;
    m_camRot = m_camTransform.eulerAngles;
```

```
//锁定鼠标
Screen.lockCursor = true;
}
```

步骤 03 在 Control 函数中旋转摄像机，在移动主角后使摄像机位置与主角位置保持一致：

```
//获取鼠标移动距离
float rh = Input.GetAxis("Mouse X");
float rv = Input.GetAxis("Mouse Y");

// 旋转摄像机
m_camRot.x -= rv;
m_camRot.y += rh;
m_camTransform.eulerAngles = m_camRot;

// 使主角的面向方向与摄像机一致
Vector3 camrot = m_camTransform.eulerAngles;
camrot.x = 0; camrot.z = 0;
m_transform.eulerAngles = camrot;

// 操作主角移动,代码略...

// 使摄像机的位置与主角一致
Vector3 pos = m_transform.position;
pos.y += m_camHeight;
m_camTransform.position = pos;
```

我们通过控制鼠标旋转摄像机方向，使主角跟随摄像机的 Y 轴旋转方向，在移动主角的时候，又使摄像机跟随主角运动。运行游戏，已经可以自由地在场景中行走。

3.3.3 武器

接下来，我们将把武器绑定到摄像机上，使其随着主角移动。

- 步骤 01** 将摄像机的位置和旋转角度都设为 0。
- 步骤 02** 将摄像机的 Clipping Planes/ Near 设为 0.1，使其可以看到更近处的物体。
- 步骤 03** 在 Project 窗口的 Prefabs 文件夹中找到 M16.Prefab，这是预设好的武器 Prefab，它只是个简单的枪支模型，将其拖入场景。
- 步骤 04** 在 Hierarchy 窗口选择武器的 Prefab（名字是 M16），设置位置和旋转都为 0，并置于摄像机层级下方，使其成为摄像机的子物体。然后选择武器 Prefab 下一级层级中的枪模型 weapon，调整它的位置和角度，并在 Camera Preview 中预览效果，直到对效果满意为止，如图 3-4 所示。

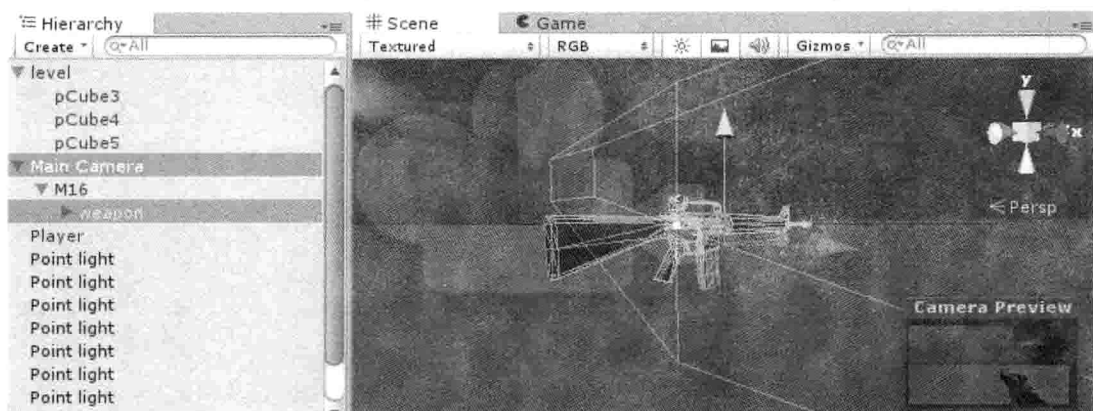


图 3-4 设置武器位置

步骤 05 运行游戏，效果如图 3-5 所示。

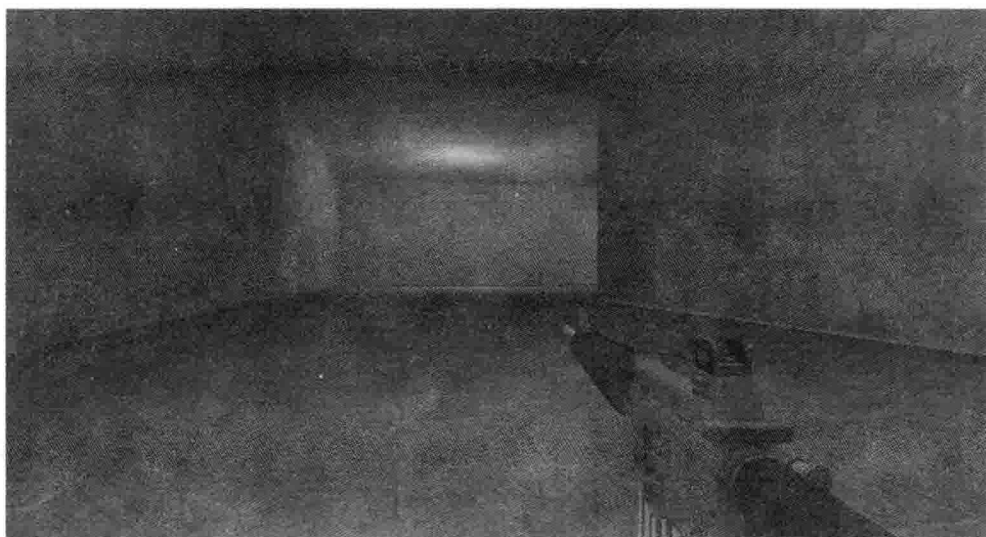


图 3-5 第一人称视角

3.4 敌人

3.4.1 寻路

在很多游戏中，敌人经常要在复杂的地形中追着主角跑，因为场景中存在很多障碍物，敌人的 AI 要足够聪明，才能找出到达目标点的最近道路，且避开障碍物。写一个完善的寻路算法是比较有挑战性的，特别是在复杂的 3D 场景中，好在 Unity Pro 版提供了一个非常实用的寻路功能，只需要较少的代码即可实现复杂的寻路功能。

Unity 的寻路系统分为两部分：一部分是对场景进行设置，使其满足寻路算法的需求；另一部分是设置寻路者。

步骤 01 选择场景模型（名为 level），然后单击 Inspector 窗口选项 Static 旁边的小三角按钮显示出下拉菜单，确定其中【Navigation Static】被选中，Unity 将知道这样的模型用于寻路计算，如图 3-6 所示。

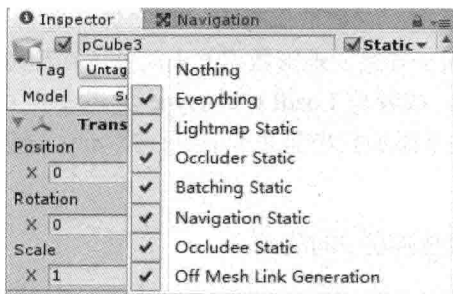


图 3-6 第一人称视角

步骤 02 在菜单栏选择【Window】→【Navigation】，打开 Navigation 窗口，如图 3-7 所示。

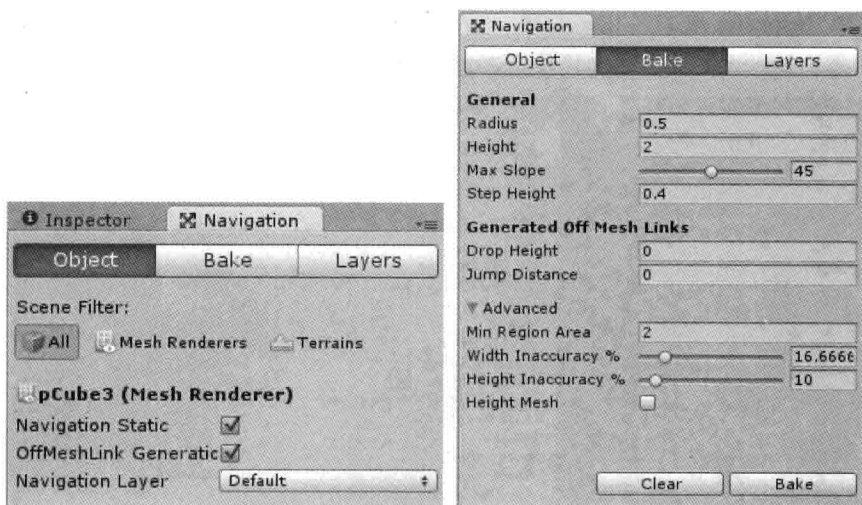


图 3-7 寻路窗口

【Object】主要用来定义模型。选择模型，如果取消选中【Navigation Static】，该模型即不会被用作计算。在脚本中调用静态函数 `GameObjectUtility.SetStaticEditorFlags` 也可以调置这个选项。

【Back】主要是定义地形对寻路的影响。

- Radius 和 Height 可以理解为寻路者的半径和高度。
- Max Slope 是最大坡度，超过这个坡度寻路者则无法通过。
- Step Height 是楼梯的最大高度，超过这个高度寻路者则无法通过。
- Drop Height 表示寻路者可以跳落的高度极限。
- Jump Distance 表示寻路者的跳跃距离极限。

步骤 03 在 Navigation 窗口设置好选项后，选择 Bake 对地形进行计算。选择 Clear 会清除计算结果。

调用静态函数 `NavMeshBuilder.BuildNavMesh` 也可以执行 Bake 计算。Unity 编辑器中的很多功能都可以在脚本中找到相应的函数来执行,有时为了方便制作,可以把一些手动操作写在脚本中执行,然后加载在菜单中。

关于 Bake 出来的寻路数据,它会被保存为一个文件,并与当前关卡相关联,这一点与 Lightmap 很像,不同的是,用于寻路的地形数据并不能被用在动态添加的关卡中。在读取关卡的时候,如果使用 `LoadLevel`,或者是 `LoadLevelAsync` 异步读取关卡都没有问题,但如果使用 `LoadLevelAdditive` 将读取的关卡添加到当前关卡中,目前 Unity 是无法正常加载用于寻路的地形数据的。

接下来设置寻路者,也就是游戏中的敌人。

步骤 04 在当前工程 Assets/Prefabs 内找到 `Zombie.prefab`, 将其拖入场景,它是一个僵尸模型,将作为游戏中的敌人。

步骤 05 在菜单栏选择【Component】→【Nav Mesh Agent】将寻路组件指定给敌人。然后在 Inspector 窗口可以进行进一步的设置,Radius 和 Height 表示寻路者的半径和高度,Speed 是最大运动速度,Angular Speed 是最大旋转速度,如图 3-8 所示。

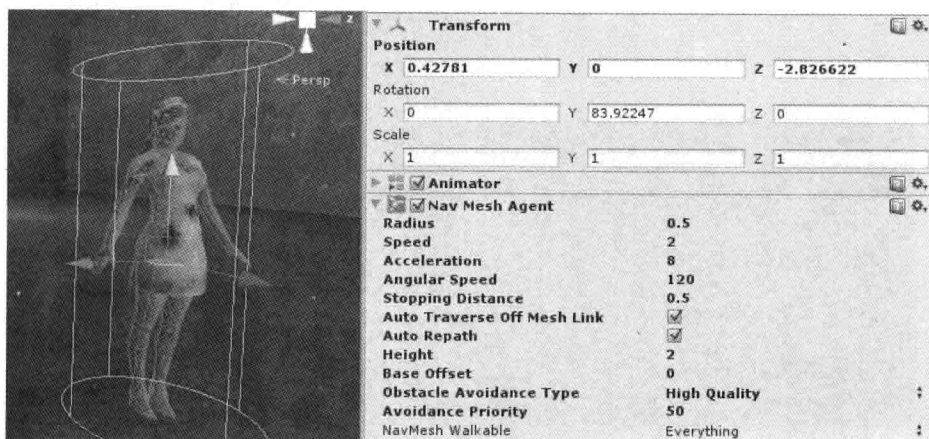


图 3-8 设置敌人的寻路

步骤 06 创建脚本 `Enemy.cs`, 添加代码如下:

```
using UnityEngine;
using System.Collections;

public class Enemy : MonoBehaviour {
    Transform m_transform;

    // 主角
    Player m_player;

    // 寻路组件
    NavMeshAgent m_agent;
```

```

//移动速度
float m_movSpeed =2.5f;

void Start () {
    m_transform = this.transform;

    // 获得主角
    m_player = GameObject.FindGameObjectWithTag("Player").GetComponent<Player>();

    // 获得寻路组件
    m_agent = GetComponent<NavMeshAgent>();
    m_agent.speed = m_movSpeed; // 设定寻路器的行走速度
    // 设置寻路目标
    m_agent.SetDestination(m_player.m_transform.position);
}
}

```

这是敌人的脚本。在 Start 函数中获得 NavMeshAgent 组件，然后调用 SetDestination 函数设置一个目标点，即可自动追击主角。如果希望自行控制移动，只能使用 NavMeshAgent.CalculatePath 计算出路径，然后按路径节点移动。

步骤 07 将脚本 Enemy.cs 指定给敌人。运行游戏，敌人会找出最短路径朝主角的位置前进，并会躲开障碍物。

3.4.2 设置动画

在前面，我们创建了可以自动寻路的敌人角色，接下来将为其增加动画效果。敌人共有 4 种动画，对应其状态，包括待机、行走、攻击和死亡。在本示例中，敌人的动画已经预先导入 Unity 工程并进行了基本设置。

步骤 01 在场景中选择敌人（或选择它的 Prefab），默认它有一个 Animator 组件，并在 Controller 中已经预设了一个 Animator Controller。取消选择【Apply Root Motion】选项，强迫使用脚本控制游戏体的位置而不是通过动画，如图 3-9 所示。

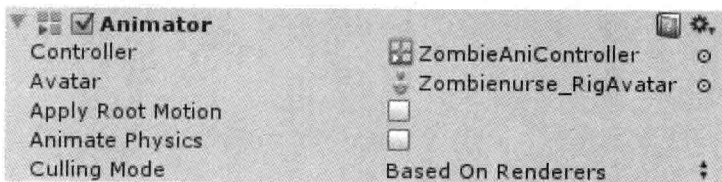


图 3-9 Animator 组件

步骤 02 在菜单栏选择【Window】→【Animator】打开 Animator 窗口，Animator Controller 的信息都显示在这里。在这个窗口中能看到敌人的全部 4 个动画，双击动画图标即可在 Project 窗口找到原始动画资源。选择 Parameters 旁边的 +，然后在子菜单中

选择【Bool】，创建4个Bool类型数值，名称分别为 idle、run、attack、death，我们将会使其与动画过渡关联，并在脚本中控制它们，如图 3-10 所示。

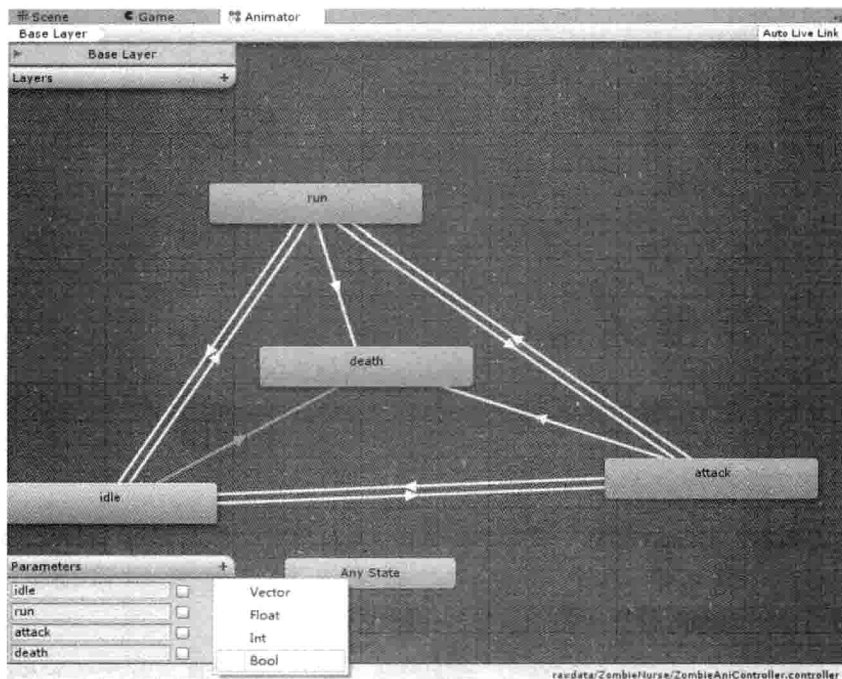


图 3-10 Animator 窗口

步骤 03 在当前工程中，动画之间已预设好动画过渡，不同动画之间过渡是用连接线表示的，默认情况下动画之间通过播放时间自动过渡，我们需要使其受脚本控制。选择连线，比如从待机动画到跑步动画，在 Conditions 中将动画过渡条件设为 run，当 Bool 值 run 为 true 时即从待机动画过渡到跑步动画，如图 3-11 所示。

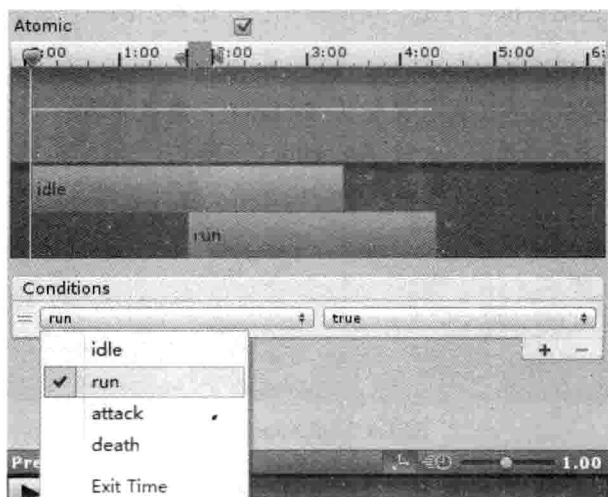


图 3-11 设置动画过渡

步骤 04 重复步骤 3 为每个动画过渡设置条件。

3.4.3 行为

敌人的行为与动画的状态紧密关联，我们将修改敌人的脚本，在不同的动画状态使敌人的行为也发生改变。

步骤 01 打开 Enemy.cs 脚本，添加动画组件等属性，更新代码如下：

```
// Transform 组件
Transform m_transform;

// 动画组件
Animator m_ani;

// 寻路组件
NavMeshAgent m_agent;

// 主角
Player m_player;

// 角色移动速度
float m_movSpeed = 0.5f;

// 角色旋转速度
float m_rotSpeed = 30;

// 计时器
float m_timer=2;

// 生命值
int m_life = 15;

void Start () {
    m_transform = this.transform;
    // 获取动画组件
    m_ani = this.GetComponent<Animator>();
    // 获得主角
    m_player = GameObject.FindGameObjectWithTag("Player").GetComponent<Player>();
    m_agent = GetComponent<NavMeshAgent>();
    m_agent.speed = m_movSpeed; // 指定寻路器的行走速度
    // 设置寻路目标
    m_agent.SetDestination(m_player.m_transform.position);
}
```


`m_rotSpeed` 用于控制旋转速度, 当敌人进攻主角时, 它将始终旋转到面向主角的角度。`m_timer` 用来计算时间间隔, 比如待机一定时间, 每隔一定时间更新寻路。`m_life` 是敌人的生命值。

步骤 02 添加 `RotateTo` 函数, 它的作用是使敌人始终旋转到面向主角的角度:

```
// 转向目标点
void RotateTo()
{
    // 获取目标方向
    Vector3 targetdir = m_player.m_transform.position - m_transform.position;
    // 计算出新方向
    Vector3 newDir = Vector3.RotateTowards(transform.forward, targetdir,
m_rotSpeed * Time.deltaTime, 0.0f);
    // 旋转至新方向
    m_transform.rotation = Quaternion.LookRotation(newDir);
}
```

`Vector3.RotateTowards` 是一个实用的函数, 通过目标点和自身的位置, 计算出转向目标点的角度。

步骤 03 修改 `Update` 函数:

```
void Update () {
    // 如果主角生命为 0, 什么也不做
    if (m_player.m_life <= 0)
        return;

    // 获取当前动画状态
    AnimatorStateInfo stateInfo = m_ani.GetCurrentAnimatorStateInfo(0);

    // 如果处于待机状态
    if (stateInfo.nameHash == Animator.StringToHash("Base Layer.idle")
&& !m_ani.IsInTransition(0))
    {
        m_ani.SetBool("idle", false);

        // 待机一定时间
        m_timer -= Time.deltaTime;
        if (m_timer > 0)
            return;

        // 如果距离主角小于 1.5 米, 进入攻击动画状态
        if (Vector3.Distance(m_transform.position, m_player.m_transform.position) <
1.5f)
```



```

        {
            m_ani.SetBool("attack", true);
        }
    else
    {
        // 重置定时器
        m_timer=1;

        // 设置寻路目标点
        m_agent.SetDestination(m_player. m_transform.position);

        // 进入跑步动画状态
        m_ani.SetBool("run", true);
    }
}

// 如果处于跑步状态
if (stateInfo.nameHash == Animator.StringToHash("Base Layer.run")
&& !m_ani.IsInTransition(0))
{
    m_ani.SetBool("run", false);

    // 每隔 1 秒重新定位主角的位置
    m_timer -= Time.deltaTime;
    if (m_timer < 0)
    {
        m_agent.SetDestination(m_player. m_transform.position);

        m_timer = 1;
    }

    // 如果距离主角小于 1.5 米, 向主角攻击
    if (Vector3.Distance(m_transform.position, m_player. m_transform.position) <=
1.5f)
    {
        // 停止寻路
        m_agent.Stop();
        // 进入攻击状态
        m_ani.SetBool("attack", true);
    }
}

// 如果处于攻击状态
if (stateInfo.nameHash == Animator.StringToHash("Base Layer.attack"))

```

```

    && !m_ani.IsInTransition(0))
    {
        // 面向主角
        RotateTo();
        m_ani.SetBool("attack", false);

        // 如果动画播完, 重新进入待机状态
        if (stateInfo.normalizedTime >= 1.0f)
        {
            m_ani.SetBool("idle", true);

            // 重置计时器
            m_timer = 2;
        }
    }
}

```

在 Update 函数中, 首先获得了一个 AnimatorStateInfo 对象, 它保存着动画的状态, 敌人包括待机、跑步、攻击、死亡 4 种状态, 我们根据不同的状态处理不同的逻辑。无论哪种状态, 都使用了 IsInTransition 判断是否是过渡状态, 如果是, 什么也不做。

SetBool 函数有两个参数: 第一个是动画的名称, 我们前面在 Animator 窗口中定义了它; 第二个是布尔值, 如果是 true, 则播放相应动画, false 则停止该动画。如果我们的程序是一个状态机结构, 可以在进入某个状态时将相应动画的值设为 true, 离开该状态时设为 false。

默认敌人处于待机状态, 并播放待机动画, 我们使用了一个计时器, 当待机时间超过 2 秒并距离主角 1.5 米以内, 则播放攻击动画, 进入攻击状态, 否则进入跑步状态。

在跑步状态中, 使用计时器每间隔 1 秒更新一次主角的位置进行寻路, 并始终追击主角, 当距离主角 1.5 米以内, 停止寻路, 播放攻击动画进入攻击状态。

在攻击状态中, 如果攻击动画播完则回到待机状态。

运行游戏, 敌人在不同的状态下会播放相应的动作, 当距离主角较近时, 则会攻击主角。

3.5 UI 界面

在继续改进主角和敌人的脚本之前, 我们需要创建一个游戏管理器来管理游戏中的事件和 UI 界面显示。

在前一章, 我们使用 OnGUI 制作了一些简单的 UI, 使用 OnGUI 在编写脚本的同时无法预览 UI 的效果。这一次我们将使用另一种方式, 直接在场景中创建 2D 贴图表示 UI 界面, 这种方式比较简单, 不过目前在实际的项目中更多的是使用 NGUI 插件制作 UI, 在后面的章节中会专门介绍。

步骤 01 在 Project 窗口找到预设的 UI 贴图文件, 如图 3-12 所示

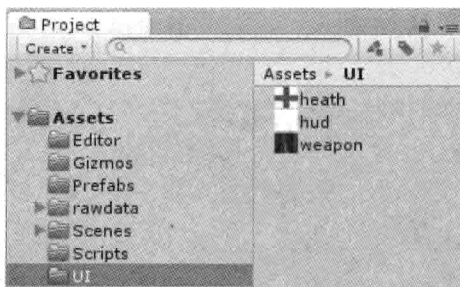


图 3-12 找到预设的贴图文件

步骤 02 选择贴图文件，在菜单栏选择【GameObject】→【Create Other】→【GUI Texture】创建 2D 贴图，然后在 Inspector 窗口移动贴图的位置，在 Game 窗口可以直接预览贴图效果。默认贴图是中心对齐，如果将 Pixel Inset 的 X 和 Y 设为 0，即为左下对齐，如果将 X 和 Y 设为 Width 和 Height 的负值，则为右上对齐，可根据需要调整，如图 3-13 所示。



图 3-13 UI 贴图效果

步骤 03 接下来创建文字。到 Windows 的 Font 文件夹中找一款自己喜欢的字体，将其复制到 Unity 工程内，确定其处于选中状态，在菜单栏选择【GameObject】→【Create Other】→【GUI Text】创建游戏文字，并调整位置、大小等。文字的常见设置包括在 Text 中输入预设的文字，在 Anchor 中选择文字的轴心点，在 Alignment 中选择文字对齐方式，在 Font Size 中设置文字的大小，如图 3-14 所示。

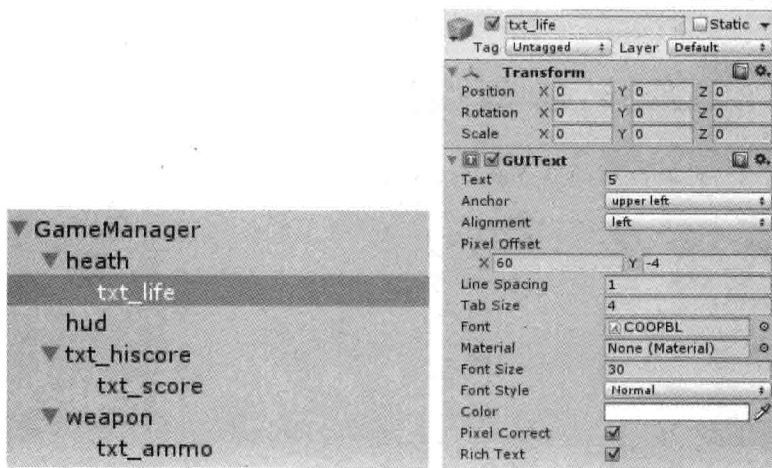


图 3-14 设置文字

最后的 UI 界面效果如图 3-15 所示。

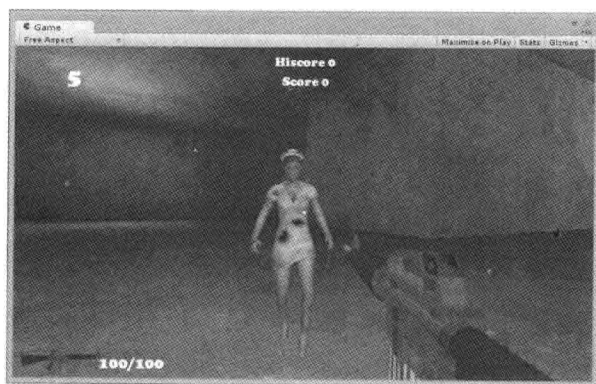


图 3-15 UI 界面效果

步骤 04 创建一个空的游戏体并命名为 `GameManager`，将其坐标设为 0，然后将所有 2D 贴图和文字都作为其子物体。因为我们需要在游戏中动态改变文字的内容，所以需要知道文字的名字，这里设 `txt_ammo` 为弹药数量的名字，`txt_hiscore` 为纪录，`txt_life` 为生命，`txt_score` 为得分。注意 UI 元素之间的子父物体关系，主要是为了使 UI 元素之间无论在何种分辨率下保持同样的相对位置，如图 3-16 所示。

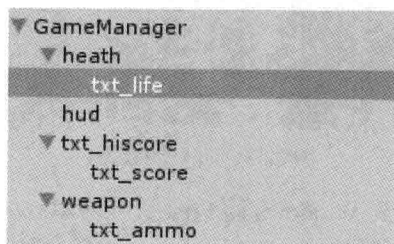


图 3-16 创建空游戏体

步骤 05 创建脚本 `GameManager.cs`，将其指定给游戏体 `GameManager`。

```
using UnityEngine;
using System.Collections;

public class GameManager : MonoBehaviour {

    public static GameManager Instance = null;

    // 游戏得分
    int m_score = 0;

    // 游戏最高得分
    static int m_hiscore = 0;

    // 弹药数量
    int m_ammo = 100;
```



```

// 游戏主角
Player m_player;

// UI 文字
GUIText txt_ammo;
GUIText txt_hiscore;
GUIText txt_life;
GUIText txt_score;

// Use this for initialization
void Start () {

    Instance = this;

    // 获得主角
    m_player = GameObject.FindGameObjectWithTag("Player").GetComponent<Player>();

    // 获得设置的 UI 文字
    foreach (Transform t in this.transform.GetComponentsInChildren<Transform>())
    {
        if (t.name.CompareTo("txt_ammo") == 0)
        {
            txt_ammo = t.GetComponent<GUIText>();
        }
        else if (t.name.CompareTo("txt_hiscore") == 0)
        {
            txt_hiscore = t.GetComponent<GUIText>();
            txt_hiscore.text = "High Score " + m_hiscore;
        }
        else if (t.name.CompareTo("txt_life") == 0)
        {
            txt_life = t.GetComponent<GUIText>();
        }
        else if (t.name.CompareTo("txt_score") == 0)
        {
            txt_score = t.GetComponent<GUIText>();
        }
    }
}
}

```

这里的代码主要是通过 `GetComponentsInChildren` 获得 `GameManager` 的所有子物体, 并通过子物体的名字获取到 UI 文字组件。

步骤 06 添加函数, 用于更新 UI 界面信息。

```

// 更新分数
public void SetScore(int score)
{
    m_score += score;

    if (m_score > m_hiscore)
        m_hiscore = m_score;
    // <color=yellow> 分数 </color> 使用标记改变分数的颜色
    txt_score.text = "Score <color=yellow>" + m_score + "</color>";
    txt_hiscore.text = "High Score " + m_hiscore;
}

// 更新弹药
public void SetAmmo(int ammo)
{
    m_ammo -= ammo;

    // 如果弹药为负数, 重新填弹
    if (m_ammo <= 0)
        m_ammo = 100 - m_ammo;

    txt_ammo.text = m_ammo.ToString() + "/100";
}

// 更新生命
public void SetLife(int life)
{
    txt_life.text = life.ToString();
}

```

步骤 07 如果主角的生命为零, 则游戏失败。这里用 OnGUI 提供一个简单的按钮重新开始游戏, 添加代码如下:

```

void OnGUI()
{
    if (m_player.m_life <= 0){
        // 居中显示文字
        GUI.skin.label.alignment = TextAnchor.MiddleCenter;

        // 改变文字大小
        GUI.skin.label.fontSize = 40;

        // 显示 Game Over 文字

```



```

        GUI.Label(new Rect(0, 0, Screen.width, Screen.height), "Game Over");

        // 设置按钮文字大小
        GUI.skin.label.fontSize = 30;
        // 显示重新游戏按钮
        if ( GUI.Button( new Rect( Screen.width*0.5f-150,Screen.height*0.75f,
300,40),"Try again"))
        {
            // 重新读入当前关卡
            Application.LoadLevel(Application.loadedLevelName);
        }
    }
}

```

3.6 交互

当前的敌人虽然会攻击主角，但并没有造成实际伤害，主角暂时也不能攻击敌人。下面我们将分别为主角和敌人添加处理逻辑的代码，使其具备攻击对方的能力。

3.6.1 主角的射击

步骤 01 打开脚本 `Player.cs`，添加 `OnDamage` 函数，该函数的作用是减少主角的生命，并更新 UI 界面的显示，当生命小于零时，取消鼠标锁定。

```

public void OnDamage(int damage)
{
    m_life -= damage;

    // 更新 UI
    GameManager.Instance.SetLife(m_life);

    // 如果生命为 0，取消鼠标锁定
    if (m_life <= 0)
        Screen.lockCursor = false;
}

```

步骤 02 在 `Player.cs` 中添加几个新的属性：

```

// 枪口 transform
Transform m_muzzlepoint;

// 射击时，射线能射到的碰撞层
public LayerMask m_layer;

```

```

// 射中目标后的粒子效果
public Transform m_fx;

// 射击音效
public AudioClip m_audio;

// 射击间隔时间计时器
float m_shootTimer = 0;

```

步骤 03 在 Player.cs 的 Start 函数中添加下面的代码, 获取枪口的位置。武器模型的枪口有一个叫 muzzlepoint 的空游戏体, 它是在 3D 建模软件中加进去的, 用来标识枪口的位置。注意, 查找它的时候, 因为它处于较深的层级, 层级之间的名字要使用 “/” 来分隔。此外, 也可以使用 GameObject.GetComponentInChildren 一次获得所有的子物体, 然后通过名字查找。

```
m_muzzlepoint = m_camTransform.FindChild("M16/weapon/muzzlepoint").transform;
```

步骤 04 在 Player.cs 的 Update 函数中添加下面的代码实现射击功能。这里主要是使用 Physics.Raycast 射出一条射线, 如果射线与敌人相碰撞, 则使敌人减少一定生命。

```

// 更新射击间隔时间
m_shootTimer -= Time.deltaTime;
// 鼠标左键射击
if (Input.GetMouseButton(0) && m_shootTimer<=0)
{
    m_shootTimer = 0.1f;
    //射击音效
    this.audio.PlayOneShot(m_audio);
    // 减少弹药, 更新弹药 UI
    GameManager.Instance.SetAmmo(1);

    // RaycastHit 用来保存射线的探测结果
    RaycastHit info;

    // 从 muzzlepoint 的位置, 向摄像机面向的正方向射出一根射线
    // 射线只能与 m_layer 所指定的层碰撞
    bool hit = Physics.Raycast(m_muzzlepoint.position,
m_camTransform.TransformDirection(Vector3.forward), out info, 100, m_layer);
    if (hit)
    {
        // 如果射中了 Tag 为 enemy 的游戏体
        if ( info.transform.Tag.CompareTo("enemy")==0 )
        {
            Enemy enemy = info.transform.GetComponent<Enemy>();

```

```

        // 敌人减少生命
        enemy.OnDamage (1);
    }
    // 在射中的地方释放一个粒子效果
    Instantiate(m_fx, info.point, info.transform.rotation);
}
}

```

步骤 05 添加两个碰撞层, enemy 和 level, 将 enemy 层指定给敌人, level 层指定给场景模型。然后再创建一个 Tag, 命名为 enemy, 指定给敌人, 如图 3-17 所示。

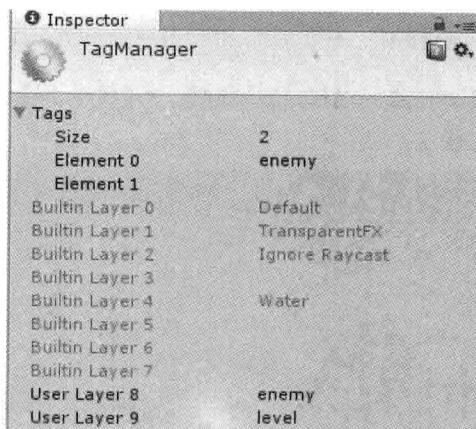


图 3-17 添加碰撞层

步骤 06 在场景中选择主角的游戏体, 为其添加 Audio Source 组件。然后在 Player 脚本组件中将 Layer 设为 enemy 和 level, 这样主角射击时, 其射线可以击中敌人和场景。在 Project 窗口找到 FX.Prefab, 将其作为射击时击中目标的特效, 在 rawdata\Sound Pack 中找到 shot.WAV 作为射击的音效, 如图 3-18 所示。

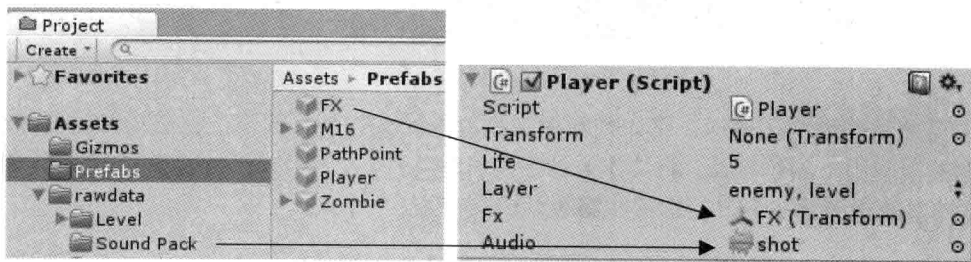


图 3-18 设置 Player

步骤 07 创建一个新的脚本 AutoDestroy.cs, 将其指定给射击特效 FX.Prefab, 这个脚本的作用是在一定时间内自动销毁游戏体。

```

using UnityEngine;
using System.Collections;

```

```

public class AutoDestroy : MonoBehaviour {

    public float m_timer = 1.0f;

    void Start () {
        Destroy(this.gameObject, timer);
    }

}

```

3.6.2 敌人的进攻与死亡

接下来我们继续修改敌人的脚本，主要是添加攻击逻辑和死亡状态。

步骤01 选择敌人，在菜单栏选择【Component】→【Physics】→【Capsule Collider】为其添加碰撞体，如图 3-19 所示。

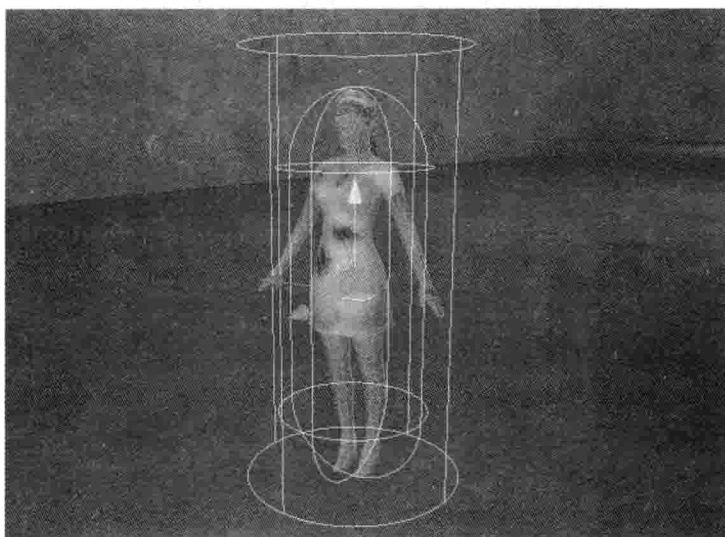


图 3-19 添加碰撞体

如果需要精确地测试碰撞，比如判断射击到敌人的头或脚等部位，在建模时需要专门创建一个低面的模型用于测试碰撞，将其作为骨骼的子物体导出，在 Unity 中将其显示功能关闭，设为 Mesh 类型的碰撞体即可，其过程在第 2 章中有介绍。

步骤02 打开 enemy.cs 脚本，添加 OnDamage 函数更新敌人的伤害，当生命值为零时，敌人进入死亡状态。

```

public void OnDamage (int damage)
{
    m_life -= damage;

    // 如果生命为 0，进入死亡状态
}

```



```

        if (m_life <= 0)
        {
            m_ani.SetBool("death", true);
        }
    }
}

```

步骤 03 在 Update 函数中添加死亡状态, 当敌人的死亡动画播完, 更新 UI 界面并销毁自身。

```

        if (stateInfo.nameHash == Animator.StringToHash("Base Layer.death") &&
            !m_ani.IsInTransition(0))
        {
            // 当播放完成死亡动画
            if (stateInfo.normalizedTime >= 1.0f)
            {
                // 加分
                GameManager.Instance.SetScore(100);

                // 销毁自身
                Destroy(this.gameObject);
            }
        }
    }
}

```

步骤 04 在 Update 函数中修改攻击状态, 执行主角的 OnDamage 函数更新主角的生命值。

```

        if (stateInfo.nameHash == Animator.StringToHash("Base Layer.attack")
            && !m_ani.IsInTransition(0))
        {
            // 始终面向主角
            RotateTo();

            m_ani.SetBool("attack", false);
            // 当完成攻击动画
            if (stateInfo.normalizedTime >= 1.0f)
            {
                // 进入待机状态
                m_ani.SetBool("idle", true);
                // 重置计时器
                m_timer = 2;

                // 更新主角的生命
                m_player.OnDamage(1);
            }
        }
    }
}

```

运行游戏, 如果主角距离敌人过近, 则有被消灭的可能。单击鼠标左键, 可以向敌人开火消灭敌人, 在子弹击中的地方还会有粒子效果出现, 如图 3-20 所示。

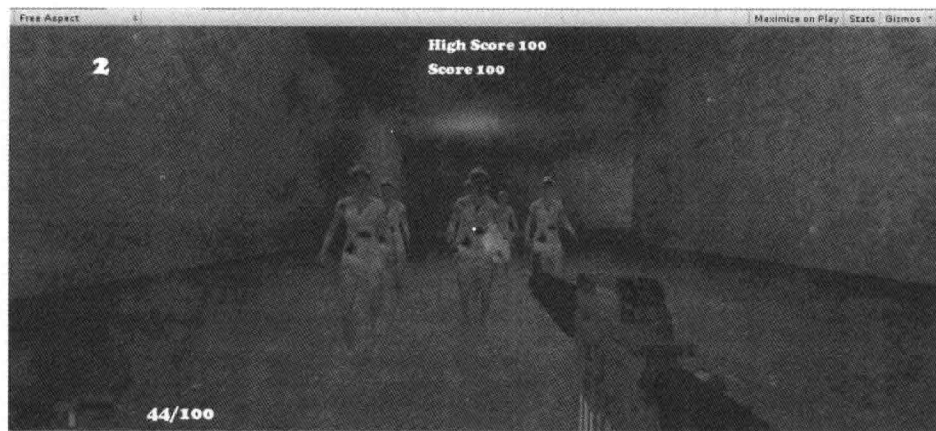


图 3-20 射击效果

3.7 出生点

接下来, 我们需要为敌人制作出生点, 使其在场景中的不同位置每隔一定时间生成一定数量的敌人。这个出生点的制作与在上一章完成的出生点类似, 但需要略加一些其他功能。为了能够控制敌人的数量, 每个出生点需要能清楚自己生成了多少敌人, 当达到最大值时, 则暂停生成敌人。当敌人被消灭时, 则需要通知出生点更新生成数量。

步骤 01 创建脚本 EnemySpawn.cs:

```
using UnityEngine;
using System.Collections;

public class EnemySpawn : MonoBehaviour
{
    // 敌人的 Prefab
    public Transform m_enemy;

    // 生成的敌人数量
    public int m_enemyCount = 0;

    // 敌人的最大生成数量
    public int m_maxEnemy = 3;

    // 生成敌人的时间间隔
    public float m_timer = 0;

    protected Transform m_transform;
    void Start () {
        m_transform = this.transform;
    }
}
```



```

    }

    void Update () {

        // 如果生成敌人的数量达到最大值, 停止生成敌人
        if (m_enemyCount >= m_maxEnemy)
            return;

        // 每间隔一定时间
        m_timer -= Time.deltaTime;
        if (m_timer <= 0)
        {
            // 得到下一轮生成敌人的间隔时间, 最大 15 秒, 最小 5 秒
            m_timer = Random.value * 15.0f;
            if (m_timer < 5)
                m_timer = 5;

            // 生成敌人
            Transform obj=(Transform)Instantiate(m_enemy, m_transform.position,
            Quaternion.identity);

            // 获取敌人的脚本
            Enemy enemy = obj.GetComponent<Enemy>();

            // 初始化敌人
            enemy.Init(this);
        }
    }

    void OnDrawGizmos ()
    {
        Gizmos.DrawIcon (transform.position, "item.png", true);
    }
}

```

在 Update 脚本中, 当生成敌人后, 我们取得了敌人的脚本组件, 然后执行了一个 Init 函数完成敌人的初始化。

步骤 02 打开敌人的脚本 Enemy.cs, 添加一个 EnemySpawn 属性, 然后添加 Init 函数, 取得 EnemySpawn 的实例, 并更新其生成敌人的数量。

```

// 生成点
protected EnemySpawn m_spawn;

// 初始化敌人

```

```

public void Init(EnemySpawn spawn)
{
    m_spawn = spawn;
    // 该敌人的出生点增加计数
    m_spawn.m_enemyCount++;
}

```

步骤 03 在 Enemy.cs 的 Update 函数中更新死亡状态的代码, 通知其出生点更新生成敌人的数量。

```

// Update 函数中的死亡状态
if (stateInfo.nameHash == Animator.StringToHash("Base Layer.death")
&& !m_ani.IsInTransition(0))
{
    if (stateInfo.normalizedTime >= 1.0f)
    {
        //更新敌人计数
        m_spawn.m_enemyCount--;

        // 加分
        GameManager.Instance.SetScore(100);

        // 销毁
        Destroy(this.gameObject);
    }
}

```

步骤 04 创建一个空游戏体, 指定 EnemySpawn.cs 脚本, 并关联敌人的 Prefab, 最后的场景设置如图 3-21 所示。

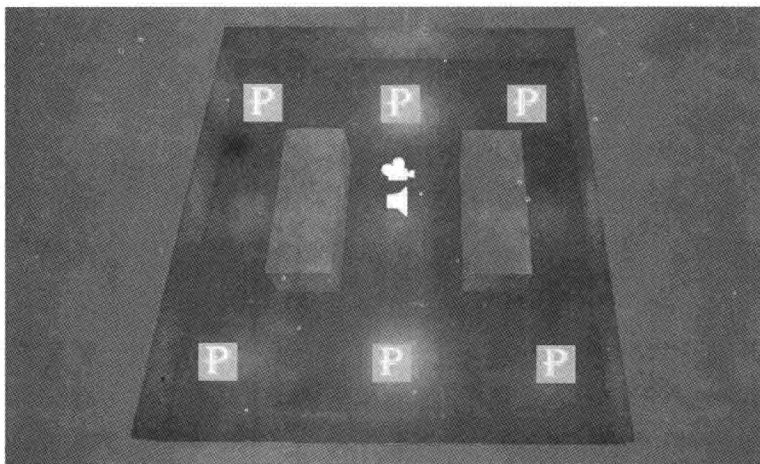


图 3-21 设置多个敌人出生点

3.8 小地图

现在的游戏已经有模有样，但因为四面八方都是敌人，我们在游戏的时候可能搞不清敌人都在哪里，最后我们将为游戏添加一个小地图，用来查看敌人的位置。

步骤 01 在菜单栏选择【GameObject】→【Create Other】→【Camera】创建一个新的摄像机，它将作为小地图的专用摄像机。调整它的位置，使其在场景上方垂直向下，然后将其设置为 Orthographic，取消透视并调整 Size 的值改变视图大小，设置 Viewport Rect 改变摄像机显示区域的位置和大小，如图 3-22 所示。

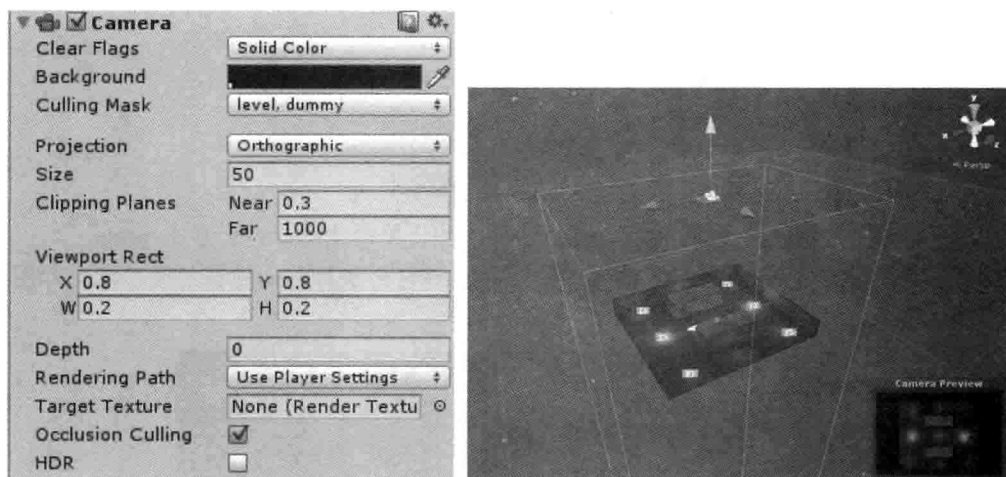


图 3-22 设置小地图摄像机

步骤 02 运行游戏，屏幕的右上方即会出现一个小地图，但根本看不清里面的东西，可能位置也不正确，如图 3-23 所示。

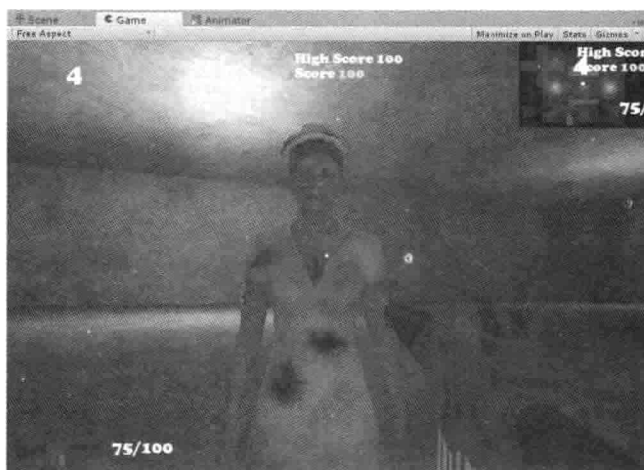


图 3-23 小地图

现在的小地图摄像机与正常摄像机的显示是一样的，只不过它是从上向下看。接下来，我

们要做的是使小地图摄像机与主摄像机只专注显示自己需要的东西, 并通过脚本使小地图摄像机的视图无论在何种分辨率下永远是一个正方形。

- 步骤 03** 创建一个球体, 命名为 dummy, 将其材质设为红色 Self-Illumin/Diffuse, 它将作为敌人的“代替体”, 只能显示在小地图之中, 并不能在主摄像机视图显示出来。将球体的 Sphere Collider 去掉, 如图 3-24 所示, 我们只需要它的显示功能。

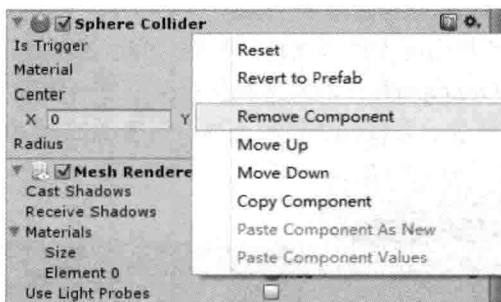


图 3-24 删除球体的碰撞组件

- 步骤 04** 创建一个新的 Layer, 命名为 dummy, 并设置球体的 Layer 为 dummy。
- 步骤 05** 将球体置于敌人 Prefab 的层级之下, 这样它会随着敌人的移动而移动, 如图 3-25 所示。

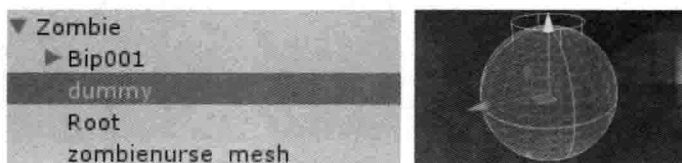


图 3-25 关联球体和敌人

- 步骤 06** 选择主摄像机, 取消显示 dummy 层, 球体在主摄像机视图中将不会被显示出来, 如图 3-26 所示。



图 3-26 在主摄像机视图中隐藏球体

- 步骤 07** 选择小地图摄像机, 使其只显示 level 和 dummy 层, 这样在小地图中只能看到场景和球体, 如图 3-27 所示。

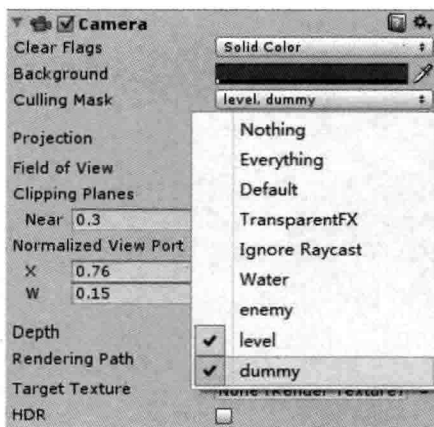


图 3-27 在小地图摄像机视图中只显示球体和场景

步骤 08 使用相同的方法为主角也创建一个“代替体”，可以为其指定与敌人不同的颜色。

步骤 09 创建脚本 MiniCamera.cs，将其指定给小地图摄像机，添加代码如下：

```
using UnityEngine;
using System.Collections;

[AddComponentMenu("Game/MiniCamera")]
public class MiniCamera : MonoBehaviour {
    // Use this for initialization
    void Start () {
        // 获得屏幕分辨率比例
        float ratio = (float)Screen.width / (float)Screen.height;
        // 使摄像机视图永远是一个正方形，rect 的前两个参数表示 XY 位置，后两个参数是 XY 大小
        this.camera.rect = new Rect((1 - 0.2f), (1 - 0.2f * ratio), 0.2f, 0.2f * ratio);
    }
}
```

这里的代码很简短，主要是使小地图摄像机的视图永远是一个正方形。

步骤 10 因为主摄像机上已经有一个 Audio Listener，同一个场景中只允许存在一个该组件，所以要取消选中（或删除）小地图摄像机的【Audio Listener】，如图 3-28 所示。

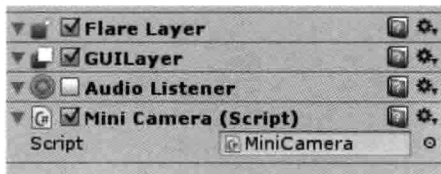


图 3-28 取消选中【Audio Listener】

运行游戏，最后的效果如图 3-29 所示。如果希望继续改进小地图的显示，还可以为场景专门制作一个用于小地图显示模型。

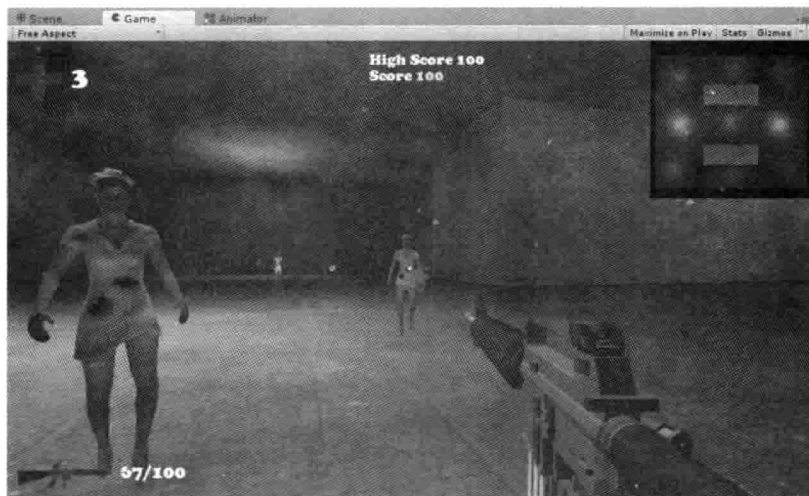


图 3-29 最后的小地图效果

本章的示例工程文件保存在资源文件目录 `chapter03_FPS` 中。在资源文件目录 `build\fps` 下保存有已经编译好的版本，可以直接运行游戏。

小 结

本章完成了一个完整的第一人称射击游戏实例，在这个过程中，我们了解了如何控制带有物理功能的角色移动，并使摄像机伴随移动。游戏中的射击判定使用了基于射线的物理碰撞，还涉及了 Mecanim 动画系统，寻路功能等。本章最后为游戏创建了一个小地图，使用了分层显示的功能。

第 4 章

塔防游戏

本章将使用 Unity 完成一款塔防游戏。我们将使用自定义的编辑器创建场景，创建路点引导敌人行动，动态读取资源，动画播放，还将涉及摄像机控制，读取 XML 文件，使用 Unity 4.6 的新 UI 系统完成 UI 界面等。

4.1 策 划

移动平台上的塔防游戏已经多得数不胜数,笔者也曾经开发过一款叫做《野人大作战》的塔防游戏(英文名为 Wild Defense)。塔防游戏的基本玩法比较类似,在场景中我方有一个基地,敌人从场景的另一侧出发,沿着相对固定的路线攻打基地。我方可以在地图上布置防守单位,攻击前来进攻的敌人,防止他们闯入基地。

本章我们也将制作一款塔防游戏,它将具备塔防游戏的最基本要素。

4.1.1 场景

塔防游戏的场景有些固定的模式,它由一个二维的单元格组成,每个格子的用途可能都不同:

- 专用于摆放防守单位的格子。
- 专用于敌人通过的格子。
- 即无法摆放防守单位,也不允许敌人通过的格子。

4.1.2 摄像机

摄像机始终由上至下俯视游戏场景,按住鼠标左键并移动可以移动摄像机的位置。

4.1.3 胜负判定

我方基地有 10 点生命值,敌人攻入基地一次减少一点生命值,当生命值为零,游戏失败。

敌人以波数的形式向我方基地进攻,每波由若干个敌人组成。在这个实例中,一关有 10 波,当成功击退敌人 10 波的进攻则游戏胜利。

4.1.4 敌人

敌人有两种,一种在陆地上行走,另一种可以飞行。每打倒一个敌人会奖励一些铜钱,用来购买新的防守单位。

4.1.5 防守单位

塔防游戏会有多种类型的防守单位,为了使本篇教程尽可能简单,我们只完成两种类型的防守单位,一种是近战类型,另一种是远程,每造一个防守单位需要消耗相应数量的铜钱。

4.1.6 UI 界面

游戏中的 UI 包括防守单位的按钮，敌人的进攻波数，基地的生命值和铜钱数量。
当防守单位攻击敌人时，在敌人的头上需要显示一个生命条表示剩余的生命值。
当游戏失败或胜利后显示一个按钮重新开始。

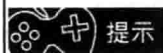
4.2 游戏场景

塔防游戏的场景通常比较简单，就像一个棋盘格，可以在上面摆放防守单位，并专门留给敌人一条通道。在这个实例中，我们使用数组来表示场景中的格子，每个格子有几种状态，分别为允许摆放防守单位的格子、专门供敌人前进的格子和无作用的格子。

步骤 01 打开本书资源文件目录 chapter04_TD_Start 内的 Unity 工程，在这个工程中，已经预先提供了本游戏所需要的模型、动画、音效等资源。此外，这个工程中还包括了笔者专门为这个教程编写的编辑器扩展工具，用来创建场景，在本章后面将会介绍如何创建一个这样的工具。首先我们简单看一下如何使用它，在菜单栏选择【Tools】→【Tile Creator】打开窗口，【Tile object Name】是场景模型的名称，【tile Size】表示每一个格子的大小，x tile count 和 z tile count 表示 x、z 轴方向上格子的数量，确定后，选择【Create Tile Object】按钮创建场景模型。如图 4-1 所示。



图 4-1 编辑器扩展窗口



提示

在【TileCreator】窗口内创建的场景模型，它的 Mesh 默认被保存在工程路径 Assets/TileEditor/Mesh 下，创建相同名称的模型，后创建的会覆盖掉前面创建的模型。

步骤 02 现在创建出的场景模型只是一个平面物体，还没有材质，我们将为其指定两种材质。选择场景模型，在 Mesh Renderer 中为其指定两个材质球，如图 4-2 所示。

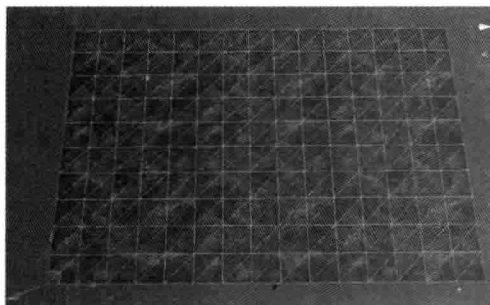
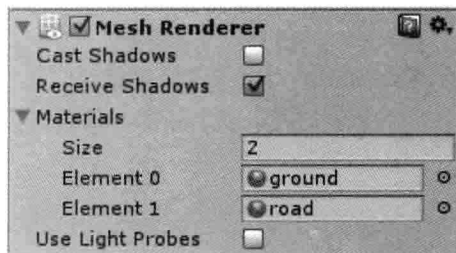


图 4-2 指定材质

步骤 03 新建一个名为 tile 的 Layer，将场景模型的 Layer 指定为 tile，然后在 Tile Object 组件内指定 Tile Layer 为 tile。选中【Edit】进入编辑模式，在【Material ID】中选择材质编号，然后即可为场景模型绘制材质，如图 4-3 所示。

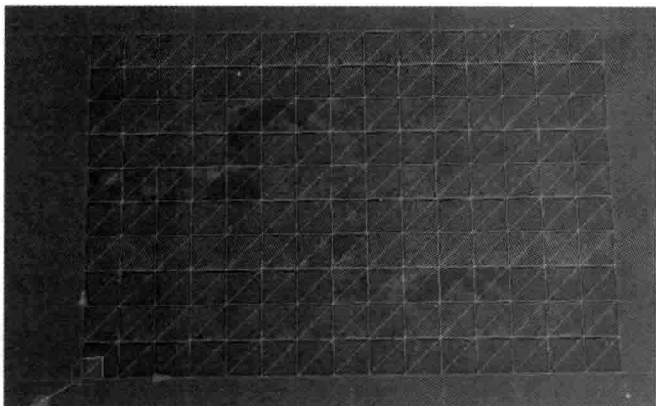


图 4-3 绘制材质

步骤 04 选择 Data，在【Data ID】中选择数值。在这个游戏中，我们定义 0 表示可以摆放防守单位，1 表示专用于敌人通过，2 表示无效格子，即不能摆放防守单位，也不能供敌人通过。选择相应的数值，直接在场景中将数值绘制到格子上面，如图 4-4 所示。

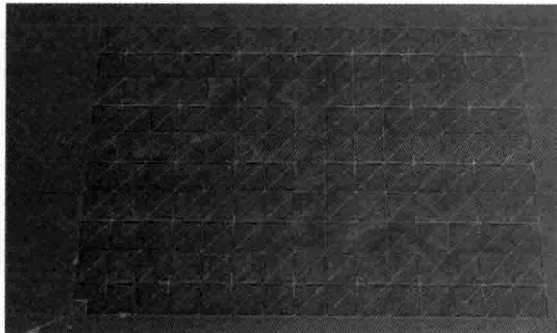
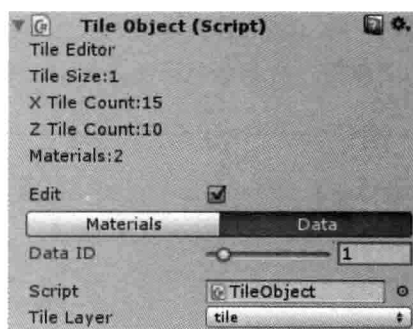


图 4-4 绘制数值



提示

TileObject.cs 这个脚本是编辑器扩展功能的一部分，它有一个 int 数组成员，数组中的每个数值与场景内的格子一一对应。绘制数值的过程即是修改数组中数值的过程，默认数组内的每个值都为零。

步骤 05 在工程路径 Assets/Rawdata/level/中提供了少量场景模型，我们可以根据自己的喜好摆放到场景中，也可以使用 **Lightmapping** 技术为场景提供灯光效果，最后的塔防场景如图 4-5 所示，道路最左边是敌人出发的位置，最右边的房子是敌人的进攻目标。

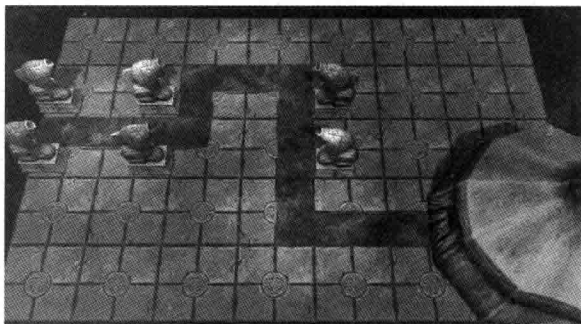


图 4-5 塔防游戏场景

4.3 制作 UI

从 Unity 4.6 开始，Unity 推出了一套新的 UI 系统，功能非常强大，接下来，我们将使用这套全新的 UI 系统创建 UI 界面。

步骤 01 在工程路径 Assets/GUI/内存放有所有的 UI 图片，选择所有的图片，然后在【Inspector】窗口将 Texture Type 设为 Sprite 类型，如图 4-6 所示。Unity 新 UI 系统只能使用 Sprite 类型的图片，关于更多 Sprite 设置的细节可以参考本书 2D 游戏部分的介绍。

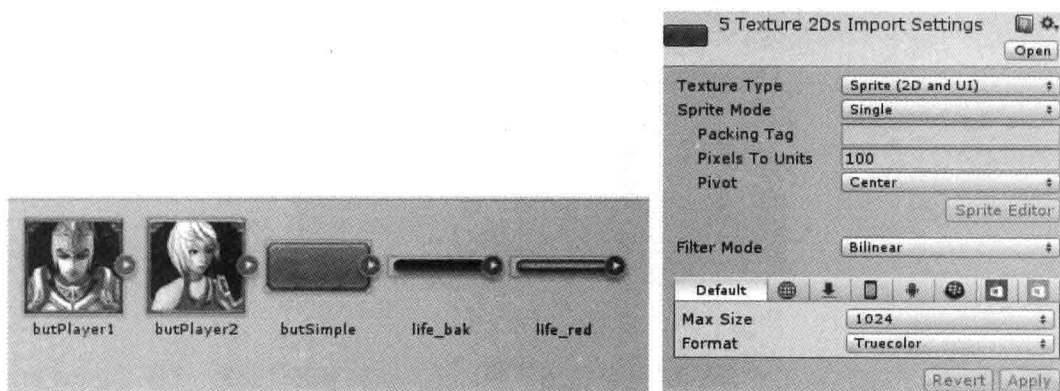


图 4-6 创建 Sprite

步骤 02 首先,我们先创建几个 UI 文字控件。在菜单栏选择【GameObject】→【UI】→【Text】创建文字,在创建文字物体的同时,还会自动创建 Canvas 和 EventSystem 物体。Canvas 会自动作为文字控件的父物体,所有的 UI 控件都需要放到 Canvas 层级下。EventSystem 物体上有很多 UI 事件脚本,用来管理和响应 UI 事件,如图 4-7 所示。

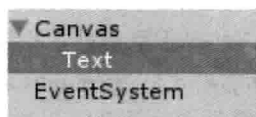


图 4-7 创建基础的 UI 控件

步骤 03 接下来我们设置文字的位置。在编辑器的上方,选择 UI 编辑按钮,然后选择文字控件即可移动它的位置,如图 4-8 所示。

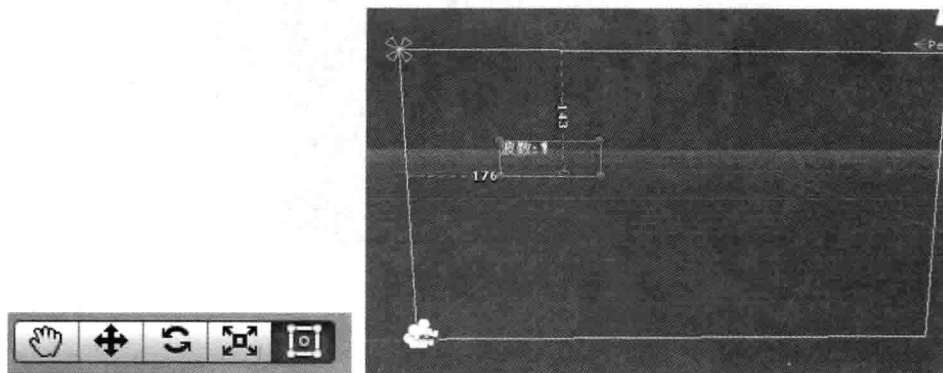


图 4-8 移动控件

步骤 04 在不同的分辨率下对齐 UI 的位置一直是件很麻烦的事情,不过使用 Unity 的新 UI 系统,一切将变得非常简单。选择前面创建的文本 UI 控件,在【Inspector】窗口的 Rect Transform 中可以快速设置控件的对齐方式,比如左对齐,右对齐等,如图 4-9 所示。

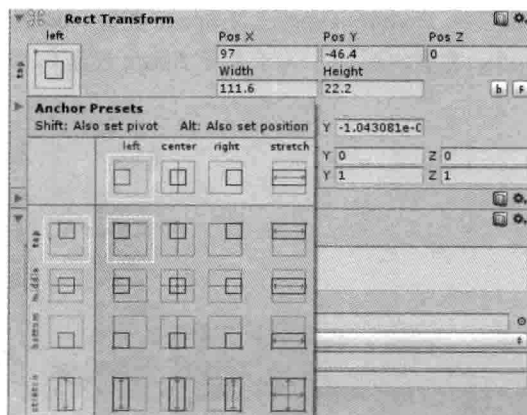


图 4-9 对齐控件

步骤 05 除了对齐,我们还需要 UI 控件根据不同的屏幕分辨率进行缩放。选择 Canvas 物体,然后在菜单栏选择【Component】→【Layout】→【Reference Resolution】,在 Resolution

中设置默认的分辨率，UI 控件将以这个分辨率为基础，在不同的分辨率下进行缩放适配，如图 4-10 所示。

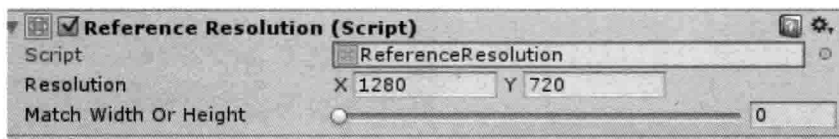


图 4-10 适配分辨率

步骤 06 这里一共需要创建 3 个不同的文字控件，分别用来显示敌人进攻的波数、铜钱和生命值。在【Inspector】窗口我们可以设置文字的内容、字体、大小和颜色等等，我们也可以使用 Unity 的 Rich Text 功能，在文本中添加标记语言，使文字的色彩不同，如图 4-11 所示。关于 Rich Text 的语法，在 Unity 的帮助文档中搜索“rich text”即可找到相关详细说明

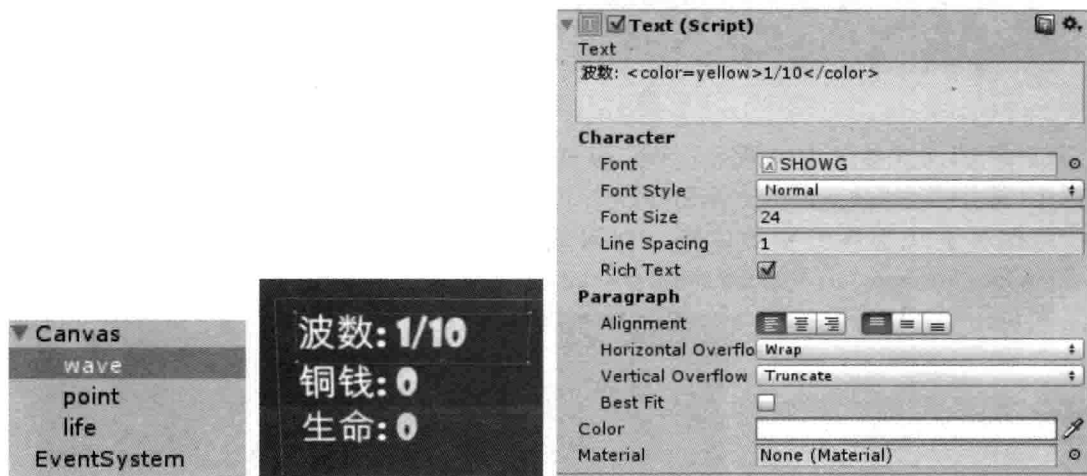


图 4-11 设置文本

步骤 07 现在很多游戏都给文字配上了描边，那么我们也加一个吧，选择文字，在菜单栏选择【Component】→【UI】→【Effects】→【Outline】，如图 4-12 所示。



图 4-12 描边字

步骤 08 最后我们要创建几个按钮，包括创建防守单位的按钮和重新游戏的按钮。在菜单栏选择【GameObject】→【UI】→【Button】即可在场景中创建一个新的按钮控件，默认按钮控件下面还有一个文字控件用来显示按钮的名称。在【Inspector】窗口找到 Image 组件下的 Source Image 设置按钮的图片，选择下面的 Set Native Size 可以

使按钮的大小与图片的尺寸快速适配。在 Button 组件中, 将 Normal Color 的 Alpha 值设为 255 使图片完全不透明。如图 4-13 所示。

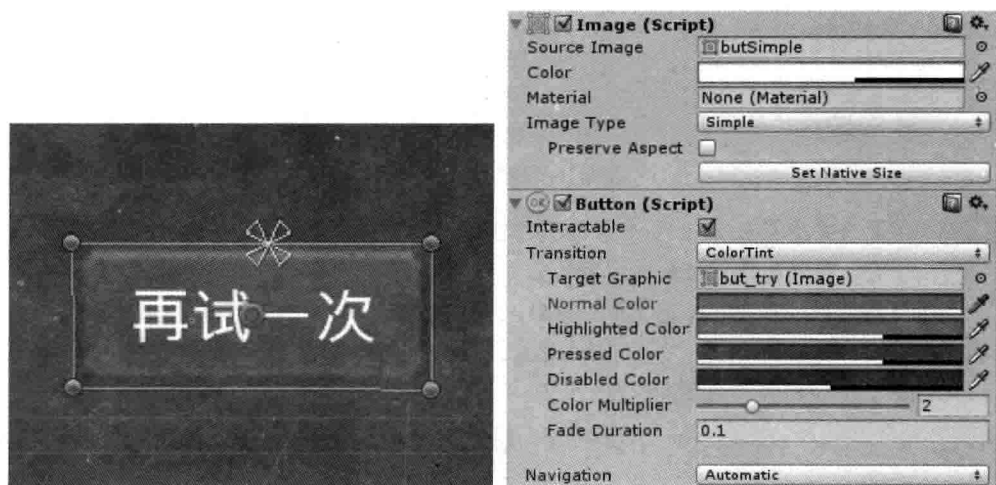


图 4-13 创建按钮

最后的 UI 效果如图 4-14 所示。注意 UI 控件的名字, 我们在后面需要通过名字来查找 UI 控件。UI 控件的层级关系也比较重要, 因为控件均位于 Canvas 或其他层级之下, 所以它们的位置只是相对于父物体的相对位置, 如果通过脚本去修改控件的位置, 通常是修改 `transform.localPosition`, 而不是 `transform.position`。

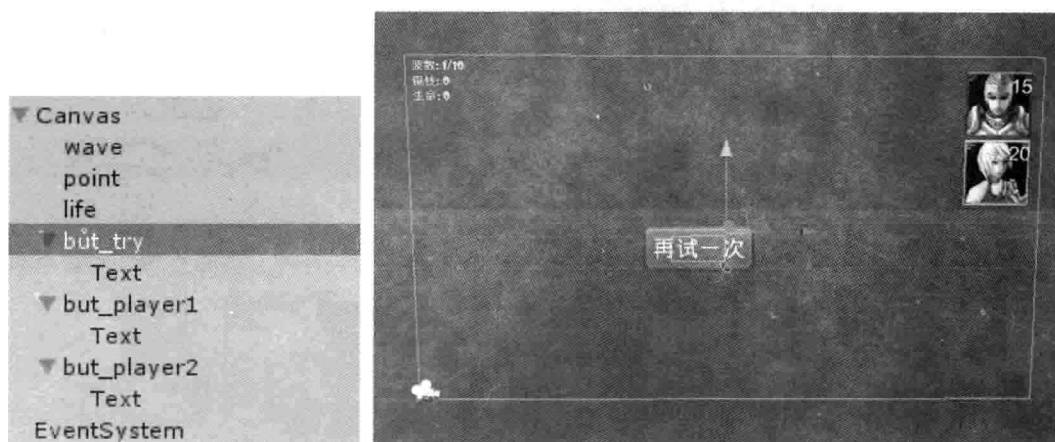


图 4-14 创建的 UI

完成 UI 设置后, 可以将 UI 保存成 Prefab。

4.4 创建游戏管理器

在前一节, 我们创建了 UI, 但还没有功能, 接下来我们将创建一个游戏管理器, 它主要用来管理 UI, 处理鼠标输入和逻辑事件等。

创建脚本 GameManager.cs, 指定到 Canvas 物体上, 添加代码如下:

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;
using UnityEngine.UI;          // 注意 UI 控件命名空间的引用
using UnityEngine.Events;      // 注意 UI 事件命名空间的引用
using UnityEngine.EventSystems; // 注意 UI 事件命名空间的引用

public class GameManager : MonoBehaviour {
    public static GameManager Instance;

    // 波数
    public int m_wave = 1;
    public int m_waveMax = 10;
    // 生命
    public int m_life = 10;
    // 铜钱数量
    public int m_point = 50;
    // 文字控件
    Text m_txt_wave;
    Text m_txt_life;
    Text m_txt_point;
    // 重新游戏按钮控件
    Button m_but_try;
    // 当前是否选中的创建防守单位的按钮
    bool m_isSelectedButton=false;

    void Awake()
    {
        Instance = this;
    }

    void Start () {

        // 定义 UnityAction, 在 OnButCreateDefenderDown 函数中响应按钮按下事件
        UnityAction<BaseEventData> downAction =
            new UnityAction<BaseEventData>(OnButCreateDefenderDown);
        // 定义 UnityAction, 在 OnButCreateDefenderDown 函数中响应按钮抬起事件
        UnityAction<BaseEventData> upAction =
            new UnityAction<BaseEventData>(OnButCreateDefenderUp);

        // 定义按钮按下事件 Entry
        EventTrigger.Entry down = new EventTrigger.Entry();
        down.eventID = EventTriggerType.PointerDown;
```

```

down.callback.AddListener(downAction);

// 定义按钮抬起事件 Entry
EventTrigger.Entry up = new EventTrigger.Entry();
up.eventID = EventTriggerType.PointerUp;
up.callback.AddListener(upAction);

// 查找所有子物体, 根据名称获取 UI 控件
foreach (Transform t in this.GetComponentsInChildren<Transform>())
{
    if (t.name.CompareTo("wave") == 0) //找到文字控件"波数"
    {
        m_txt_wave = t.GetComponent<Text>();
        SetWave(1);
    }
    else if (t.name.CompareTo("life") == 0) //找到文字控件"生命"
    {
        m_txt_life = t.GetComponent<Text>();
        m_txt_life.text =
            string.Format("生命: <color=yellow>{0}</color>", m_life);
    }
    else if (t.name.CompareTo("point") == 0) //找到文字控件"分数"
    {
        m_txt_point = t.GetComponent<Text>();
        m_txt_point.text =
            string.Format("铜钱: <color=yellow>{0}</color>", m_point);
    }
    else if (t.name.CompareTo("but_try") == 0) //找到按钮控件"重新游戏"
    {
        m_but_try = t.GetComponent<Button>();
        // 添加按钮单击函数回调
        m_but_try.onClick.AddListener( OnButRetry );
        // 隐藏重新游戏按钮
        m_but_try.gameObject.SetActive(false);
    }
    else if (t.name.Contains("but_player")) //找到按钮控件"创建防守单位"
    {
        // 给创建防守单位按钮添加 EventTrigger, 并添加前面定义的按钮事件
        EventTrigger trigger = t.gameObject.AddComponent<EventTrigger>();
        trigger.delegates = new List<EventTrigger.Entry>();
        trigger.delegates.Add(down);
        trigger.delegates.Add(up);
    }
}
}

```



```

    }

    // 更新文字控件"波数"
    public void SetWave(int wave)
    {
        m_wave = wave;
        m_txt_wave.text =
            string.Format("波数: <color=yellow>{0}/{1}</color>", m_wave, m_waveMax);
    }

    // 更新文字控件"生命"
    public void SetDamage(int damage)
    {
        m_life -= damage;
        if (m_life <= 0) {
            m_life = 0;
            m_but_try.gameObject.SetActive(true); //如果生命值为0,则显示重新开始游戏按钮
        }
        m_txt_life.text = string.Format("生命: <color=yellow>{0}</color>", m_life);
    }

    // 更新文字控件"铜钱"
    public bool SetPoint(int point)
    {
        if (m_point + point < 0) // 铜钱数量不可能小于0
            return false;
        m_point += point;
        m_txt_point.text = string.Format("铜钱: <color=yellow>{0}</color>", m_point);

        return true;
    }

    // 响应重新游戏按钮
    void OnButRetry(GameObject go)
    {
        Application.LoadLevel(Application.loadedLevelName);
    }

    // 按下"创建防守单位按钮"
    void OnButCreateDefenderDown(BaseEventData data)
    {
        m_isSelectedButton = true;
    }

    // 抬起"创建防守单位按钮"

```

```

void OnButCreateDefenderUp(BaseEventData data)
{
    // 暂时先不做什么,但可以打印一下当前按钮的名字
    GameObject go = data.selectedObject;
    Debug.Log(go.name);
}
}

```

在 **Start** 函数中,我们首先定义了按钮的事件,然后通过查找所有子物体的名称找到相应的 UI 控件进行初始化处理,对于 **Text** 文字控件,赋予初始的波数、生命和铜钱数值。

因为在游戏中开始时我们不希望看到重新游戏按钮,我们调用 `gameObject.SetActive(false)` 方法将该按钮隐藏,当游戏结束时再显示该按钮重新游戏。

对于创建防守单位的按钮,我们分别定义了按下和抬起两个事件,当按下按钮的时候,获得要创建的对象,抬起按钮时在选定位置创建防守单位,不过当前这些按钮事件的回调函数是空的,还没做什么事情。

4.5 摄像机

因为游戏的场景可能会比较大,我们需要移动摄像机才能观察到场景的各个部分。接下来我们将为摄像机添加脚本,在移动鼠标的时候可以移动摄像机。

步骤 01 在为摄像机创建脚本前我们需要先创建一个空游戏体作为摄像机观察的目标点,并为其创建脚本 `CameraPoint.cs`,它只有很少的代码。

```

using UnityEngine;
using System.Collections;

public class CameraPoint : MonoBehaviour
{
    public static CameraPoint Instance = null;

    void Awake()
    {
        Instance = this;
    }

    void OnDrawGizmos()
    {
        Gizmos.DrawIcon(transform.position, "CameraPoint.tif");
    }
}

```

步骤 02 创建脚本 `GameCamera.cs`,并将其指定给场景中的摄像机。


```
using UnityEngine;
using System.Collections;

public class GameCamera : MonoBehaviour {

    public static GameCamera Inst = null;

    // 摄像机距离地面的距离
    protected float m_distance = 15;

    // 摄像机的角度
    protected Vector3 m_rot = new Vector3(-55, 180, 0);

    // 摄像机的移动速度
    protected float m_moveSpeed = 60;

    // 摄像机的移动值
    protected float m_vx = 0;
    protected float m_vy = 0;

    // Transform 组件
    protected Transform m_transform;

    // 摄像机的焦点
    protected Transform m_cameraPoint;

    void Awake()
    {
        Inst = this;
        m_transform = this.transform;
    }

    // Use this for initialization
    void Start()
    {
        // 获得摄像机的焦点
        m_cameraPoint = CameraPoint.Instance.transform;

        Follow();
    }

    // 在 Update 之后执行
    void LateUpdate()
    {
```

```

        Follow();
    }

    // 摄像机对齐到焦点的位置和角度
    void Follow()
    {
        m_transform.position = m_cameraPoint.position;
        m_transform.eulerAngles = m_rot;
        m_transform.Translate(0, 0, m_distance);

        this.transform.LookAt(m_cameraPoint);
    }

    // 控制摄像机移动
    public void Control(bool mouse, float mx, float my)
    {
        if (!mouse)
            return;

        m_cameraPoint.Translate(-mx, 0, -my);
    }
}

```

在这个脚本的 Start 函数中, 我们首先获得了前面创建的 CameraPoint, 它将作为摄像机目标点的参考。

在 Follow 函数中, 摄像机会按预设的旋转和距离始终跟随 CameraPoint 目标点。

LateUpdate 函数和 Update 函数的作用一样, 不同的是它始终会在执行完 Update 后执行, 我们在这个函数中调用 Follow 函数, 确保在所有的操作完成后再移动摄像机。

Control 函数的作用是移动 CameraPoint 目标点, 因为摄像机的角度和位置始终跟随这个目标点, 所以也会随着目标点的移动而移动。

步骤 03 打开 GameManager.cs 脚本, 在 Update 函数中添加代码如下:

```

void Update () {
    // 如果选中创建士兵的按钮则取消摄像机操作
    if (m_isSelectedButton)
        return;

    // 按下鼠标操作
    bool press=Input.GetMouseButton(0);

    // 获得鼠标移动距离
    float mx = Input.GetAxis("Mouse X");
    float my = Input.GetAxis("Mouse Y");
}

```

```
// 移动摄像机
GameCamera.Inst.Control(press, mx, my);
}
```

这段代码的作用是获取鼠标操作的各种信息，并传递给摄像机，现在运行游戏，已经可以移动摄像机了。

4.6 路 点

在前一章，我们使用 Unity 提供的寻路功能实现敌人的行动，但在塔防游戏中，敌人通常不需要智能寻路，而是按照一条预设的路线行动。接下来我们将为敌人创建一条前进路线，这条路线是预设的，敌人将从游戏场景的左侧沿着通道一直走到右侧。

步骤 01 路线是由若干个路点组成，首先为路点创建脚本 PathNode.cs:

```
using UnityEngine;
using System.Collections;

public class PathNode : MonoBehaviour {
    // 父路点
    public PathNode m_parent;
    // 子路点
    public PathNode m_next;

    // 设置子节点
    public void SetNext(PathNode node)
    {
        if (m_next != null)
            m_next.m_parent = null;
        m_next = node;
        node.m_parent = this;
    }
    // 显示路点图标
    void OnDrawGizmos()
    {
        Gizmos.DrawIcon(this.transform.position, "Node.tif");
    }
}
```

在游戏中，敌人将从一个路点到达另一个路点，即到达当前路点的子路点。在 PathNode 脚本中，主要是通过 SetNext 函数设置它的子路点。

接下来我们将创建路点并为每个路点设置子路点, 为了设置方便, 我们将添加一个菜单功能, 加速设置路点的操作。

步骤 02 在 Project 窗口内的 Assets 目录下创建一个名为 Editor 的文件夹, 名称是特定的, 不能改变, 所有需要在编辑状态下执行的脚本都应当被存放到这里。在 Editor 文件夹内创建脚本 PathTool.cs, 它将提供一个自定义的菜单, 帮助我们设置路点, 代码如下:

```
using UnityEngine;
using UnityEditor;
using System.Collections;

public class PathTool : ScriptableObject
{
    // 父路点
    static PathNode m_parent=null;

    [MenuItem("PathTool/Create PathNode")]
    static void CreatePathNode()
    {
        // 创建一个新的路点
        GameObject go = new GameObject();
        go.AddComponent<PathNode>();
        go.name = "pathnode";
        // 设置 tag
        go.tag = "pathnode";
        // 使该路点处于选择状态
        Selection.activeTransform = go.transform;
    }

    // 菜单 SetParent, 用来设置父路点, 快捷键 Ctrl+Q
    [MenuItem("PathTool/Set Parent %q")]
    static void SetParent()
    {
        // 如果没有选中任何物体, 或选择物体数量大于 1, 返回
        if (!Selection.activeGameObject ||
            Selection.GetTransforms(SelectionMode.Unfiltered).Length>1)
            return;
        // 如果选中的物体 Tag 设为 pathnode
        if (Selection.activeGameObject.tag.CompareTo("pathnode") == 0)
        {
            // 保存当前选中的路点
            m_parent = Selection.activeGameObject.GetComponent<PathNode>();
        }
    }
}
```

```

    }

    // 菜单 SetNextChild, 用来设置子路点, 快捷键 Ctrl+W
    [MenuItem("PathTool/Set Next %w")]
    static void SetNextChild()
    {
        if (!Selection.activeGameObject || m_parent==null ||
            Selection.GetTransforms(SelectionMode.Unfiltered).Length>1)
            return;
        if (Selection.activeGameObject.tag.CompareTo("pathnode") == 0)
        {
            // 设置子路点
            m_parent.SetNext( Selection.activeGameObject.GetComponent<PathNode>() );
            m_parent = null;
        }
    }
}

```

这里的代码只有在编辑状态才能被执行, 注意 PathTool 继承自 ScriptableObject, 并在最前面引用了 UnityEditor。所有在这里使用的属性和函数均为 static 类型。

Selection 是专用在编辑模式下的一个静态类, 通过它可以获取到当前选择的物体。

[MenuItem("PathTool/Set Parent %q")]属性将在菜单中添加名为 PathTool 的自定义菜单, 并包括子菜单 Set Parent, 快捷键为 Ctrl+q。菜单 Set Parent 执行的功能即是 SetParent 函数中的功能, 将当前选中的节点作为父路点。

相应的 SetNextChild 函数将当前选中的路点作为父路点的子路点。

步骤 03 在菜单栏选择【PathTool】→【Creat PathNode】创建路点。

步骤 04 复制若干个路点沿着道路摆放。按快捷键 Ctrl+Q 将其设为父路点, 然后选择下一个路点按 Ctrl+W 设为子路点, 再按 Ctrl+Q 将它设为父路点, 再选择子路点, 反复这个操作, 直到将所有路点设置完毕, 注意, 最后一个路点没有子路点。最后的效果如图 4-15 所示。



图 4-15 设置路点

虽然我们设置好路点,但还无法在场景中清楚地观察路点之间的联系,接下来我们将在 GameManager.cs 中添加代码,使路点之间产生一条连线。

步骤 05 打开脚本 GameManager.cs,添加两个属性如下。m_debug 是一个开关,控制是否显示路点之间的连线,m_PathNodes 是一个 ArrayList,它用来保存所有的路点。

```
public bool m_debug = false;
public ArrayList m_PathNodes;
```

步骤 06 继续在 GameManager.cs 中添加函数 BuildPath,并在 Start 函数中调用它,它的作用是将所有场景中的路点装入 m_PathNodes。

```
[ContextMenu("BuildPath")]
void BuildPath()
{
    m_PathNodes = new ArrayList();
    // 查找所有 tag 为 pathnode 的 GameObject
    GameObject[] objs = GameObject.FindGameObjectsWithTag("pathnode");
    for (int i = 0; i < objs.Length; i++)
    {
        PathNode node = objs[i].GetComponent<PathNode>();
        m_PathNodes.Add(node);
    }
}
```

步骤 07 继续在 GameManager.cs 中添加函数 OnDrawGizmos,它的作用是当 m_debug 属性为真时显示路点之间的连线。

```
void OnDrawGizmos()
{
    if (!m_debug || m_PathNodes==null )
        return;
    Gizmos.color = Color.blue;
    foreach (PathNode node in m_PathNodes)
    {
        if (node.m_next != null)
        { //画线
            Gizmos.DrawLine(node.transform.position,
                             node.m_next.transform.position);
        }
    }
}
```

步骤 08 选择 UI Root 上的 Game Manager 脚本组件,设置 m_debug 属性为真,选择右上方的齿轮按钮,在子菜单中选择【BuildPath】,这是我们自定义的菜单,如图 4-16 所示。



图 4-16 自定义的 BuildPath 选项

步骤 09 选择 BuildPath 后将在场景中看到路点之间的连线，如图 4-17 所示。



图 4-17 路点之间的连线

4.7 敌人

敌人一共有两种，一种在陆地上前进，另一种则会飞行。我们将先创建前一种，然后继承它的大部分属性和函数，略加修改完成另一种。

步骤 01 创建敌人的脚本 Enemy.cs:

```
public class Enemy : MonoBehaviour {

    // 当前路点
    public PathNode m_currentNode;
```

```

// 生命
public int m_life = 15;
public int m_maxlife = 15;

// 移动速度
public float m_speed = 2;

// Update is called once per frame
void Update () {
    RotateTo();
    MoveTo();
}

// 旋转向目标
public void RotateTo()
{
    // 获得当前 y 轴方向
    float current= this.transform.eulerAngles.y;
    //使用 LookAt 方向转向目标, 因此获得了最终角度
    this.transform.LookAt(m_currentNode.transform);
    // 使用 Mathf.MoveTowardsAngle 方法获得转由当前 y 轴转到最终角度的中间值
    float next=Mathf.MoveTowardsAngle(current, this.transform.eulerAngles.y,
                                       120 * Time.deltaTime);
    this.transform.eulerAngles = new Vector3(0, next, 0);
}

// 向目标方向移动
public void MoveTo()
{
    Vector3 pos1 = this.transform.position;
    Vector3 pos2 = m_currentNode.transform.position;
    float dist = Vector2.Distance(new Vector2(pos1.x,pos1.z),new Vector2(pos2.x,
pos2.z));
    if (dist < 1.0f)
    {
        if (m_currentNode.m_next == null) // 如果到达我方基地
        {
            GameManager.Instance.SetDamage(1);
            DestroyMe();
        }
        else
            m_currentNode = m_currentNode.m_next;
    }
    this.transform.Translate(new Vector3(0, 0, m_speed * Time.deltaTime));
}

```

```

    }

    public void DestroyMe()
    {
        Destroy(this.gameObject);
    }
}

```

在这个脚本中，定义了敌人的一些基本属性，如生命值、移动速度、类型等，它有一个路点属性作为出发点。

RotateTo 函数使敌人始终转向目标路点(注意在后面创建防守单位时也创建了一个 RotateTo 函数，不过是使用的不同方式)，MoveTo 函数则使其沿着当前方向前进，当距离目标路点较近时，将该路点作为当前路点，再向下一个路点前进。注意这里计算敌人与子路点的距离时没有计算 Y 轴。当敌人走到最后的路点，即是到达我方基地，销毁自身，并使基地减少一点生命值。

步骤 02 在【Project】窗口下的 Rawdata/enemy/Resources 文件夹中找到 boar@skin.FBX 模型文件，拖入场景放到通道左侧。这是个野猪模型，它将作为陆地上的敌人。将 Enemy.cs 脚本指定给它，并设置起始路点，如图 4-18 所示。

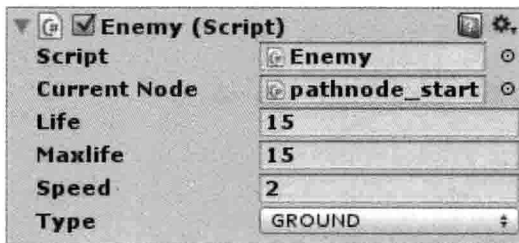


图 4-18 敌人组件

运行游戏，敌人会从起始点出发，沿着路点，一路前进到达我方基地，然后消失，我方基地将损失一点生命值。

这时我们会发现，野猪模型没有任何动画，看上去很生硬。在前一章我们使用了 Mecanim 动画系统播放动画效果，这种方式是一种基于状态的动画系统，功能强大，也比较复杂，使用起来略麻烦一些。如果不是追求特别复杂的动画效果，我们可以使用 Unity 中原始的动画播放功能。

步骤 03 带有动画的模型，默认被导入到 Unity 中时会被自动设为 Generic，这是一种适配 Mecanim 动画系统的模式，我们要将它转为 Legacy 模型，然后在 Animation 组件中指定动画，当前的野猪模型只有一个跑步动画，文件名是 bird@run，@后面的 run 即是动画的名称，这是 Unity 对动画文件定义的一种命名规则，最后的设置如图 4-19 所示。

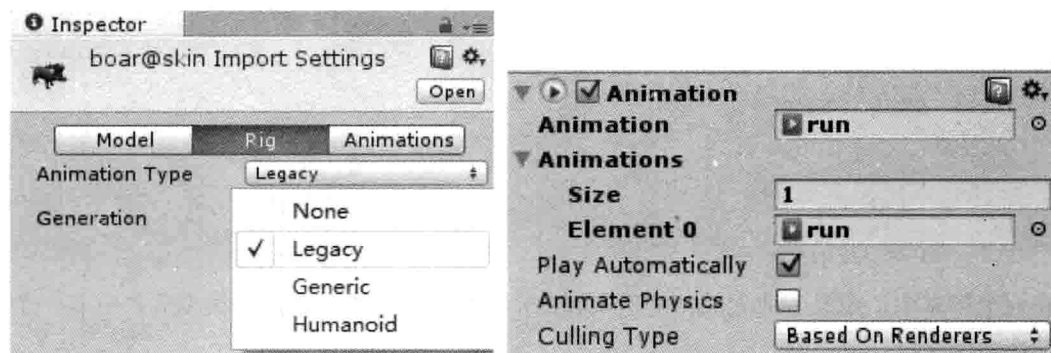


图 4-19 添加动画

再次运行游戏，即可看到模型在前进中播放了跑动的动画。

这里的步骤，每次导入模型后都需要手工修改动画模式比较麻烦，我们可以在 Editor 文件夹里添加一个脚本，并继承自 AssetPostprocessor，即可通过回调函数预处理导入的模型，使它自动转为 Legacy 模式，代码如下：

```
using UnityEngine;
using UnityEditor;
using System.Collections;

public class ProcessModel : AssetPostprocessor {
    // 预处理模型回调函数
    void OnPreprocessModel()
    {
        // 如果模型名称包括@
        if (assetPath.Contains("@"))
        {
            // 获取模型
            ModelImporter model = assetImporter as ModelImporter;
            // 更改它的动画模式
            model.animationType = ModelImporterAnimationType.Legacy;
        }
    }
}
```

步骤 04 接下来创建另一个飞行敌人的脚本 AirEnemy.cs，它继承了 Enemy 脚本的大部分功能，只添加一个 Fly 函数，作用是当高度小于 2 时向上飞行。

```
using UnityEngine;
using System.Collections;

public class AirEnemy : Enemy {
    void Update () {
        RotateTo();
    }
}
```

```

        MoveTo();
        Fly();
    }
    public void Fly()
    {
        float flyspeed = 0;
        if (this.transform.position.y < 2.0f)
        {
            flyspeed = 1.0f;
        }
        this.transform.Translate(new Vector3(0, flyspeed * Time.deltaTime, 0));
    }
}

```

步骤 05 在 Project 窗口下的 Rawdata/enemy/Resources 文件夹中找到 bird@skin.FBX 模型，拖入场景放到通道左侧并指定动画。这是个飞鸟模型，它将作为空中的敌人。将 AirEnemy.cs 脚本指定给它，并设置起始路点。运行游戏，效果如图 4-20 所示。



图 4-20 沿着路点前进的敌人

当前我们看到了敌人的画面效果，接下来一种选择是将敌人制作成 Prefab 直接使用，但如果有很多敌人，设置起来很麻烦。另一种选择是直接通过代码读取模型并指定脚本和动画，当数据量增大时，这么做的好处非常明显，只需要写一次代码，便不用再关心模型、动画和脚本等组件的设置。本章将采用第二种方法。

4.8 敌人生成器

塔防游戏的敌人通常是成批出现，一波接着一波，因为敌人的数量众多，所以需要有一个生成器按预先设置的顺序生成不同的敌人。为了提高工作效率，我们将在 Excel 中设置每一波出现的敌人，然后将其导出为 XML 格式，敌人生成器会读取这个 XML 文件，按其设置生成敌人。

实际上,我们也可以在 Unity 的【Inspector】窗口直接设置配置的数值,好处是调试比较方便,但查阅设置比较困难,多人合作开发也会带来一些管理上的问题。

需要注意的是,直接读取 XML 并不是最好的配置方式,还有一些更复杂的做法,比如使用脚本(推荐 Python)读取 Excel,将内容存为游戏可以解析的文件,也可以在客户端创建内置数据库管理数据等。

4.8.1 在 Excel 中设置敌人

我们先在 Excel 中设置数据,之后会将其导入 Unity。

步骤 01 首先要创建一个用于定义 XML 格式的数据源,使用任意文本编辑器输入如下文本,并保存为 template.xml (文件可任意命名)。

```
<?xml version="1.0" encoding="utf-8"?>
<ROOT>

<!-- content -->
  <table wave="" enemyname="" level="" wait="" />
  <table wave="" enemyname="" level="" wait="" />
</ROOT>
```

这是一个简单的 XML 文件,里面没有什么内容,但定义了我们需要的格式: wave 表示第几波, enemyname 是敌人的名字,这个名字需要在 Unity 中与敌人关联, level 表示敌人的等级, wait 表示生成这个敌人需要等待的时间。

注意我们定义了两列一样的 table,这是必须的,只有这样在 Excel 中才能批量映射数据元素。

步骤 02 启动 Excel,这里使用的版本是 Excel2010。

步骤 03 在 Excel 工具栏右键选择【自定义功能区】(或者在工具栏选择【文件】→【选项】),如图 4-21 所示。

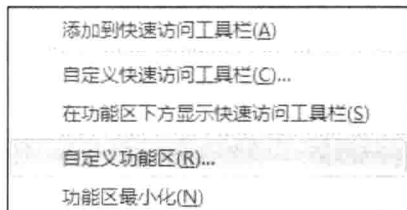


图 4-21 自定义功能区

步骤 04 在自定义功能区中选中【开发工具】选项 (Office2007 需要在【常用】选项中选中【在功能区显示“开发工具”选项卡】),如图 4-22 所示

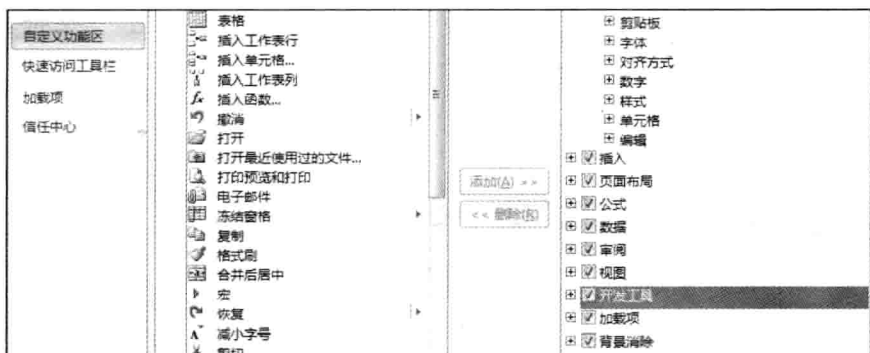


图 4-22 显示“开发工具”选项卡

步骤 05 在开发工具选项卡中选择【源】，如图 4-23 所示。

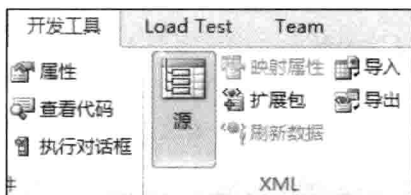


图 4-23 源

步骤 06 在 XML 源窗口的下方选择【XML 映射】，如图 4-24 所示。

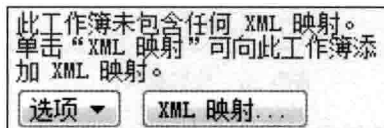


图 4-24 XML 映射

步骤 07 选择【添加】打开之前准备的 template.xml，如图 4-25 所示。

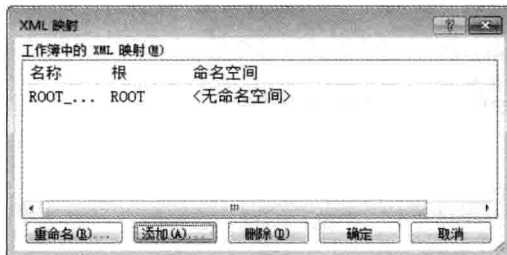


图 4-25 添加 XML 映射数据源

步骤 08 在 Excel 中添加文本，名称与 XML 数据源一致，如图 4-26 所示。

wave	enemyname	level	wait

图 4-26 添加文本

步骤 09 在 XML 源中依次选择 table 下面的 wave、enemyname、level、wait 并拖曳至相应的文本表格中, 如图 4-27 所示。

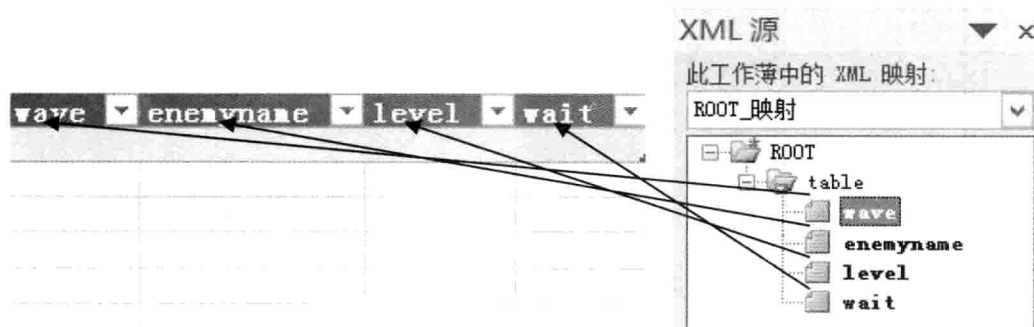


图 4-27 映射

步骤 10 在 enemyname 中设置的敌人名字需要与 Unity 的敌人名字完全一样, 为了避免输入错误, 可以通过设置数据有效性的方法来避免。选中 enemyname 下的单元格, 在【数据】选项卡中选择【数据有效性】, 如图 4-28 所示。



图 4-28 数据有效性

步骤 11 在【允许】中选择【序列】, 然后在【来源】中输入敌人的所有名字, 每个名字都用逗号隔开, 这样在单元格中只能使用预先定义的名字, 如图 4-29 所示。



图 4-29 设置来源

步骤 12 输入相应的数据, 如图 4-30 所示。因为本示例是教程的原因, 数据都比较简单。通常实际的项目中, 每个敌人需要一个索引, 在另一张表中通过索引查看敌人的具体数值, 如生命力、移动速度、价值、模型信息等。

wave	enemynam	level	wait
1	boar	1	3
1	boar	1	3
2	boar	2	4
2	bird	2	4
3	boar	2	3
3	boar	3	3
4	bird	3	5
4	bird	3	4
5	boar	4	4
5	bird	4	4
6	boar	4	4
6	bird	4	5
7	bird	5	4
7	bird	5	3
8	boar	5	4
8	boar	5	4
9	bird	6	4
9	boar	6	5
10	boar	7	4
10	bird	7	4

图 4-30 数值

步骤 13 在表格上右键选择【XML】→【导出】导出 XML 文件，本示例保存为 enemy.xml，如图 4-31 所示。

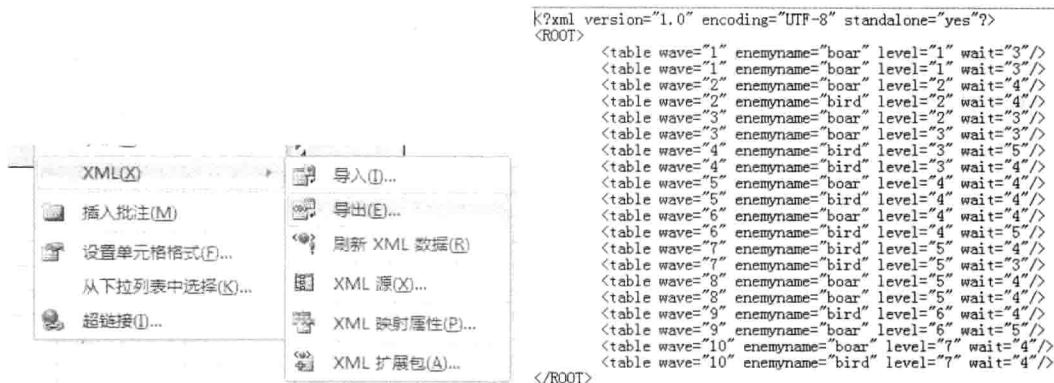


图 4-31 导出 XML 文件

4.8.2 创建敌人生成器

敌人生成器的主要功能分为两个部分，一部分是读入 XML 定义的敌人数据，并将其存入到一个 List 中，另一部分是从 List 中读出敌人的数据并将其创建出来。

步骤 01 回到 Unity 中，创建 EnemySpawner.cs，首先在 EnemySpawner 类中定义一个类 SpawnData，用来帮助映射 XML 文件中敌人的相关数据。

```

using UnityEngine;
using System.Collections;
using System.Collections.Generic;
public class EnemySpawner : MonoBehaviour {

```

```
// XML 数据
public class SpawnData
{
    // 波数
    public int wave = 1;
    public string enemyname = "";
    public int level = 1;
    public float wait = 1.0f;
}
}
```

步骤 02 为 EnemySpawner 定义属性，xmldata 将指向我们需要读入的 XML 文件，而读入后的数据将保存在 m_spawnlist 中。

```
// 起始路点
public PathNode m_startNode;

// 存储敌人出场顺序的 XML
public TextAsset xmldata;

// 保存所有的从 XML 读取的数据
List<SpawnData> m_spawnlist;

// 距离下一个敌人出场的时间
float m_timer = 0;

// 出场敌人的序列号
int m_index = 0;

// Use this for initialization
void Start () {
    // 读取 XML
    ReadXML();

    // 获取初始敌人
    SpawnData data = m_spawnlist[m_index];
    m_timer = data.wait;
}

// 读取 XML
void ReadXML()
{
    m_spawnlist = new List<SpawnData>();

    XMLParser xmlparse = new XMLParser();
```



```

XMLNode node = xmlparse.Parse(xmldata.text);

XMLNodeList list = node.GetNodeList("ROOT>0>table");
for (int i = 0; i < list.Count; i++)
{
    string wave = node.GetValue("ROOT>0>table>" + i + ">@wave");
    string enemyname = node.GetValue("ROOT>0>table>" + i + ">@enemyname");
    string level = node.GetValue("ROOT>0>table>" + i + ">@level");
    string wait = node.GetValue("ROOT>0>table>" + i + ">@wait");

    SpawnData data = new SpawnData();
    data.wave = int.Parse(wave);
    data.enemyname = enemyname;
    data.level = int.Parse(level);
    data.wait = float.Parse(wait);

    m_spawnlist.Add(data);
}
}

```

ReadXML 函数将 XML 中的数据读出，并存入到 m_spawnlist 中。

这里我们并没有使用 .NET 中提供的标准 XML 功能读取 XML 文件，而是使用了 Unity 社区中的开源脚本，而且是 .js 格式的，它已经预先放入当前工程，保存在工程路径 Plugins/XMLParser/ 内。这个脚本的发布网址是 <http://dev.grumpyferret.com/unity/>，在这里可以获得最新的版本。

不使用 .NET 提供的 XML 功能是因为它会使游戏的体积变得较大，而且有些功能无法应用到网页版中，Unity 官方不建议使用它，在 Unity 安装目录 Editor\Data\Documentation\Documentation\Images>manual 下有一个文件 Mono.Xml.zip，这是 Unity 官方提供的一个精简版的 XML 解析脚本。

此外，除了 XML，还可以直接使用 Excel 另存为 CSV 格式，使用恰当的方式解析，都是不错的选择。在序列化数据的时候，最好使类的属性或变量名称与配置文件中的名称一致，然后创建一个类专门用来自动序列化配置文件，在本书附录 A 中 A.2.6 节介绍了该方法。

步骤 03 为 EnemySpawner 脚本中添加生成敌人的函数：

```

void Update () {
    SpawnEnemy();
}

// 每隔一定时间生成一个敌人
void SpawnEnemy()
{
    // 如果已经生成所有敌人
    if (m_index >= m_spawnlist.Count)
        return;
}

```

```

// 更新时间, 等待下一个敌人
m_timer -= Time.deltaTime;
if (m_timer > 0)
    return;

// 获取下一个敌人的数据
SpawnData data = m_spawnlist[m_index];
// 如果下一个敌人是下一波, 需要等待前一波的敌人全部消亡
if (GameManager.Instance.m_wave < data.wave )
{
    if ( GameManager.Instance.m_EnemyList.Count > 0 )
        return;
    else
        GameManager.Instance.SetWave(data.wave); // 更新 wave 数值
}

// 更新敌人序列号
m_index++;
if (m_index < m_spawnlist.Count)
    m_timer = m_spawnlist[m_index].wait; // 更新等待的时间

// 读取敌人的模型
GameObject enemymodel = Resources.Load<GameObject>(data.enemyname + "@skin");
// 读取敌人的动画
GameObject enemyani = Resources.Load<GameObject>(data.enemyname + "@run");
// 实例化敌人的模型 并转向第一个路点
Vector3 dir = m_startNode.transform.position - this.transform.position;
GameObject enmeyObj = (GameObject)Instantiate(enemymodel,
                                                this.transform.position,
                                                Quaternion.LookRotation(dir));
enmeyObj.animation.AddClip(enemyani.animation.GetClip("run"), "run");

// 设置循环播放动画
enmeyObj.animation["run"].wrapMode = WrapMode.Loop;
// 播放跑步动画
enmeyObj.animation.CrossFade("run");
// 添加 Enemy 角本
Enemy enemy = enmeyObj.AddComponent<Enemy>();

// 设置敌人的出发路点
enemy.m_currentNode = m_startNode;

// 设置敌人数值, 本示例只是简单的根据波数增加敌人的生命

```

```

        enemy.m_life = data.level * 3;
        enemy.m_maxlife = data.level * 3;
    }

    void OnDrawGizmos()
    {
        Gizmos.DrawIcon(transform.position, "spawner.tif");
    }

```

生成敌人的主要功能都在 `SpawnEnemy` 函数中，每次生成敌人需要间隔一定时间，然后在 `m_enemylist` 中获得下一个敌人的数据。

我们通过 `Resources.Load` 函数直接读入模型文件，这很像 IO 操作，但被读入的模型只能存放在命名为 `Resources` 的文件夹内，除了模型，任何 `Resources` 文件夹内的资源都能被直接读入。要注意的是无论我们是否使用了存放在 `Resources` 文件夹中的资源，它们都会被打包到最终游戏中。

我们不但读取了敌人的模型，还读取了它的动画，通过 `animation.GetClip` 可以根据名称获取模型的动画，`animation.AddClip` 则可以用来为当前模型添加动画。设置动画的 `WrapMode` 可以使动画循环播放或只播放一次。使用这些功能，我们可以不必设置 `Prefab`，直接从原始文件中读取模型和动画文件。

在读取并设置好敌人的模型后，我们使用 `AddComponent<Enemy>` 函数将敌人的脚本动态的添加到模型上面，并初始化数值。使用这种方式，可以完全用代码创建实例，避免很多编辑器中的手工操作。如果是新创建一个实例，则可以像下面这样：

```

GameObject go = new GameObject();
go.AddComponent<SomeScript>();

```

步骤 04 创建一个空游戏体作为敌人生成器放置到场景通道左侧，为其指定 `EnemySpawner.cs` 脚本。在 `m_startNode` 中设置起始路点，在 `m_xmldata` 中关联 XML 文件，如图 4-32 所示。

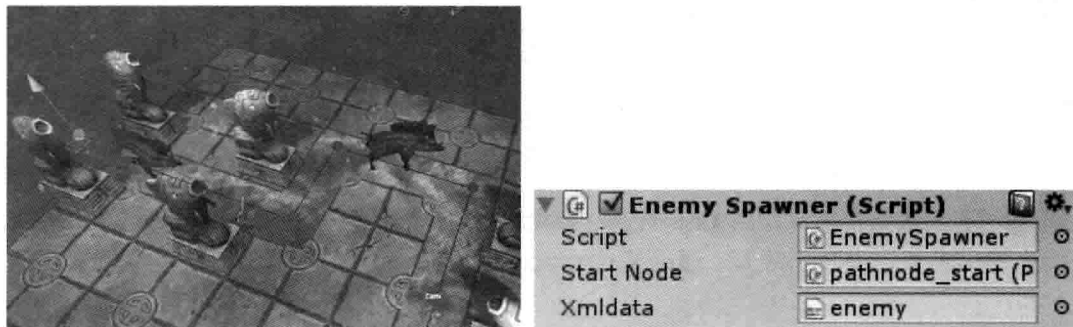


图 4-32 设置敌人 `prefab` 和起始路点

运行游戏，敌人会按照 Excel 表格中的设计逐个生成。

4.8.3 遍历敌人

现在,游戏中有很多敌人,为了能方便地遍历游戏中的所有敌人,查看它们的情况,我们可以准备一个容器,将所有生成的敌人都装进去。

步骤 01 打开 GameManager.cs 脚本,添加一个 List 属性用来存放所有的敌人。

```
using System.Collections.Generic;

public List<Enemy> m_EnemyList = new List<Enemy>();
```

步骤 02 打开 Enemy.cs 脚本,在 Start 和 DestroyMe 函数中分别更新 List 中的敌人。添加一个 SetDamage 函数更新敌人生命值,当生命值为零时销毁自身并增加些铜钱。

```
void Start () {
    GameManager.Instance.m_EnemyList.Add(this);
}

public void DestroyMe()
{
    GameManager.Instance.m_EnemyList.Remove(this);
    Destroy(this.gameObject);
}

public void SetDamage(int damage)
{
    m_life -= damage;
    if (m_life <= 0)
    {
        m_life=0;
        // 每消灭一个敌人增加一些铜钱
        GameManager.Instance.SetPoint(5);
        DestroyMe();
    }
}
```

现在所有的敌人都被保存到 List 当中,当我们创建出防守单位后,他们可以通过遍历 List 中的敌人查找敌人,并攻击敌人,通过 SetDamage 函数更新敌人的生命值。

4.9 防守单位

本游戏中有两种防守单位,一种是近战类型,另一种是远程类型。我们先创建近战类型的防守单位,然后再通过它派生出远程类型的防守单位。

步骤 01 创建防守单位的脚本 Defender.cs:

```

using UnityEngine;
using System.Collections;

public class Defender : MonoBehaviour {
    // 格子状态
    public enum TileStatus
    {
        LOCK = 0,    // 不能在上面做任何事
        ROAD=1,      // 专用于敌人行走
        FREE=2,      // 专用于创建防守单位的格子
    }

    // 攻击范围
    public float m_attackArea = 2.0f;
    // 攻击力
    public int m_power = 1;
    // 目标敌人
    protected Enemy m_targetEnemy;
    // 防守单位的模型
    protected GameObject m_model;
    // 防守单位的动画
    protected Animation m_ani;

    public delegate void VoidDelegate();
    // 用一个委托指向不同行为（在实际项目中最好使用状态机来管理不同行为）
    public event VoidDelegate m_action;
    public void SetAction(VoidDelegate act) { m_action = act; }

    // 静态工厂函数 创建防守单位实例
    public static T Create<T>( Vector3 pos, Vector3 angle ) where T : Defender
    {
        GameObject go = new GameObject();
        go.transform.position = pos;
        go.transform.eulerAngles = angle;
        go.name = "defender";
        T d = go.AddComponent<T>();
        d.Init(); // 初始化

        // 将自己所占格子的信息设为占用
        TileObject.get.setData(d.transform.position.x, d.transform.position.z,
                                (int)TileStatus.LOCK);

        return d;
    }
}

```



```

}

// 初始化数值
protected virtual void Init()
{
    m_attackArea = 2.0f; //在实际项目中, 数值通常会从配置文件中读取
    m_power = 2;
    CreateModel("swordman");
    m_action = Idle;
}

// 创建模型
protected void CreateModel(string myname)
{
    //读取模型和动画资源
    GameObject model = Resources.Load<GameObject>(myname + "@skin");
    GameObject ani_run = Resources.Load<GameObject>(myname + "@attack");
    GameObject ani_idle = Resources.Load<GameObject>(myname + "@idle");
    // 创建模型
    m_model = (GameObject)Instantiate(model, transform.position,
                                     transform.rotation);
    m_model.transform.parent = this.transform;
    m_ani = m_model.animation;
    m_ani.AddClip(ani_run.animation.GetClip("attack"), "attack");
    m_ani.AddClip(ani_idle.animation.GetClip("idle"), "idle");
    //ClampForever 动画会停留在最后一帧
    m_ani["attack"].wrapMode = WrapMode.ClampForever;
    m_ani["idle"].wrapMode = WrapMode.Loop;
    m_ani.CrossFade("idle");
}

// Update is called once per frame
void Update () {
    // 更新当前行为
    if (m_action != null)
        m_action();
}

// 待机状态
protected void Idle()
{
    if ( !m_ani.IsPlaying("idle")) // 如果没有播放待机动画则播放待机动画
        m_ani.CrossFade("idle");
    FindEnemy(); // 查找敌人
    if (m_targetEnemy != null) //如果找到敌人则进入攻击状态

```

```

    {
        m_action = Attack;
    }
}

protected virtual void Attack()
{
    bool ok = RotateTo(); // 转向敌人
    if (ok)
    {
        if (!m_ani.IsPlaying("attack")) // 如果没有播放攻击动画则播放攻击动画
        {
            m_ani.CrossFade("attack");

        }
        else if (m_ani["attack"].time > m_ani["attack"].length * 0.5f)
        {
            // 当动画播到一半的时候进入 OnAttack 状态
            m_action = OnAttack;
        }
    }
}

protected virtual void OnAttack()
{
    if (m_targetEnemy != null)
        m_targetEnemy.SetDamage(m_power); // 减少敌人生命值
    m_targetEnemy = null;
    if (m_ani["attack"].time >= m_ani["attack"].length) // 播完动画回到待机状态
    {
        m_action = Idle;
        m_ani.Stop("attack");
    }
}

// 转向敌人
public bool RotateTo()
{
    if (m_targetEnemy == null)
        return true;
    // 是否转向目标
    bool ok = false;

    // 获得敌人的位置

```

```

Vector3 targetpos = m_targetEnemy.transform.position;
targetpos.y = this.transform.position.y;

// 获得目标方向
Vector3 targetDir = targetpos - transform.position;
// 向目标方向旋转, 返回新的方向
Vector3 rot_delta = Vector3.RotateTowards(this.transform.forward, targetDir,
                                           20.0f * Time.deltaTime, 0.0F);

// 转向这个方向
Quaternion targetrotation = Quaternion.LookRotation(rot_delta);

// 获得目标方向与当前方向的差值
float angle = Vector3.Angle(targetDir, transform.forward);
// 如果
if (angle < 1.0f)
{
    // 已转向目标
    ok = true;
}
transform.rotation = targetrotation;
return ok;
}

// 查找最近敌人
void FindEnemy()
{
    if (m_targetEnemy != null)
        return;

    float mindist = -1;
    foreach (Enemy enemy in GameManager.Instance.m_EnemyList)
    {
        if (enemy.m_life == 0)
            continue;

        Vector3 pos1 = this.transform.position; pos1.y = 0;
        Vector3 pos2 = enemy.transform.position; pos2.y = 0;
        // 获得目标距离自己的距离
        float dist = Vector3.Distance(pos1, pos2);
        // 如果目标不在攻击范围内
        if (dist > m_attackArea)
            continue;
        // 寻找距离最近的
        if (mindist < 0 || mindist > dist)

```



```

    {
        // 更新目标
        m_targetEnemy = enemy;
        mindist = dist;
    }
}
}
}

```

Create 函数是一个静态函数，我们可以使用它直接在代码中创建防守单位的游戏体，而无需创建 Prefab。这个函数使用了模版，在这个例子中我们可以添加所有继承自 Defender 的派生类到 Create 函数中创建的游戏体上。

TileStatus 是一个枚举，用来表示场景中格子的状态，在 Create 函数中创建防守单位的时候，我们会更新格子的状态，使原本空闲的格子变为占有状态，这样便不能在这个格子中创建新的防守单位。

Init 函数是一个初始化函数，初始化了一些数值。注意这是一个虚函数，当我们派生出其它类后，可以重载 Init 函数，赋予不一样的数值。在实际的项目中，我们也可以将所有的防守单位数值记录到配置文件中，创建不同的角色，读入不同的数值即可。

CreateModel 函数创建了所有的模型和动画。我们使用 Resources.Load 函数读入了所有的模型和动画资源，将它们实例化，然后使用 AddClip 函数将动画添加到模型上，并设置动画的 WrapMode 改变循环模式，Loop 表示循环，ClampForever 只播放一次动画。在 ClampForever 模式下，当动画播到最后一帧，动画会停在最后一帧，在这种模式下，当前动画可以使用 CrossFade 函数平滑过渡到另一个动画，使用 Once 模式也是播放一次动画，但不能得到很好的动画过渡效果。

FindEnemy 函数会遍历所有的敌人，找出处于攻击范围内的敌人，并选择距离自己最近的。委托 m_action 在这里的作用类似函数指针，我们将它指向不同的函数，切换不同的行为。

步骤 02 接下来创建另一个有远程射击能力的防守单位，创建一个新的角本 Archer.cs:

```

using UnityEngine;
using System.Collections;

// 远程防守单位
public class Archer : Defender {

    // 模型边框
    Bounds bounds;

    // 初始化数值
    protected override void Init()
    {
        m_attackArea = 5.0f;
        m_power = 1;
    }
}

```

```

        CreateModel("archer");
        SetAction(Idle);
        // 获得模型的边框
        bounds = m_model.GetComponentInChildren<SkinnedMeshRenderer>().bounds;
    }

    protected override void Attack()
    {
        bool ok = RotateTo();
        if (ok)
        {
            if (!m_ani.IsPlaying("attack"))
            {
                m_ani["attack"].time = 0;
                m_ani.CrossFade("attack");
            }
            else if (m_ani["attack"].time > m_ani["attack"].length * 0.5f)
            {
                SetAction(OnAttack);
                if (m_targetEnemy != null)
                {
                    // 将弓箭的初始位置设置到自身中心偏上并向前0.5个单位
                    Vector3 pos = this.transform.TransformPoint(0, bounds.center.y + 0.5f,
                                                                0.5f);

                    // 创建弓箭
                    Arrow.Create(m_targetEnemy.transform, pos, (Enemy enemy) =>
                    {
                        enemy.SetDamage(m_power);
                        m_targetEnemy = null;
                    });
                }
            }
        }
    }

    // 向敌人攻击
    protected override void OnAttack()
    {
        if (m_ani["attack"].time >= m_ani["attack"].length)
        {
            //m_action = Idle;
            SetAction(Idle);
            m_ani.Stop("attack");
        }
    }

```



```

    }
}

```

Archer 类继承自 Defender 类，重载了 Init、Attack 和 OnAttack 函数，赋予不同的数值并做出不同的攻击行为。因为是该类表现为远程攻击，所以在攻击的时候创建了一个弓箭实例，弓箭的角本我们将在下一步创建。

在 Init 函数中，我们获取了模型的 Bounds 属性，它通常用来表示模型的边框大小。在创建弓箭的时候，我们使用了 TransformPoint 函数按当前位置偏移，将弓箭的位置设于模型中心偏上并向前 0.5 个单位。如果需要更准确的射击位置，可以在 3D 建模软件中指定一个攻击点，然后在读取模型时根据名称找到这个点即可。

接下来我们将创建弓箭的角本。

步骤 03 创建弓箭脚本 Arrow.cs:

```

using UnityEngine;
using System.Collections;

public class Arrow : MonoBehaviour {
    // 当打击到目标时执行的动作
    System.Action<Enemy> onAttack;
    // 目标对象
    Transform m_target;
    // 目标对象模型的边框
    Bounds m_targetCenter;
    // 静态函数 创建弓箭
    public static void Create( Transform target, Vector3 pos, System.Action<Enemy>
onAttack )
    {
        // 创建新的游戏体
        GameObject go = new GameObject();
        // 添加弓箭脚本组件
        Arrow ar = go.AddComponent<Arrow>();
        // 设置弓箭的目标
        ar.m_target = target;
        // 获得目标模型的边框
        ar.m_targetCenter =
            target.GetComponentInChildren<SkinnedMeshRenderer>().bounds;
        // 取得 Action
        ar.onAttack = onAttack;
        // 初始化位置
        ar.transform.position = pos;
        ar.transform.LookAt(target);

        // 读取弓箭模型
    }
}

```

```

        GameObject arrowmodel = Resources.Load<GameObject>("arrow");
        // 实例化弓箭模型
        GameObject arrowObj = (GameObject)Instantiate(arrowmodel,
ar.transform.position, ar.transform.rotation);
        // 将弓箭模型设为当前游戏体的子物体
        arrowObj.transform.parent = go.transform;
        // 3 秒之后销毁
        Destroy(go, 3.0f);
    }

    void Update()
    {
        // 瞄准目标
        if (m_target != null)
            this.transform.LookAt(m_targetCenter.center);

        // 向目标前进
        this.transform.Translate(new Vector3(0, 0, 10 * Time.deltaTime));
        if (m_target != null)
        {
            // 检查是否打击到目标
            if (Vector3.Distance(this.transform.position, m_targetCenter.center) <
0.5f)
            {
                // 通知弓箭发射者
                onAttack(m_target.GetComponent<Enemy>());
                Destroy(this.gameObject);
            }
        }
    }
}

```

弓箭类的功能很简单, 创建一个弓箭模型, 然后朝目标点前进, 当距离目标点小于 0.5 个单位时, 触发 Action 通知弓箭发射者已经打击到目标, 然后销毁自己。

最后, 我们要修改 GameManager.cs 脚本, 在按钮事件中添加创建防守单位的代码。

步骤 04 打开 GameManager.cs 脚本, 添加一个 Layer Mask 属性, 与格子地面的 Layer 相关联, 修改 OnButCreateDefenderUp 函数, 在抬起鼠标的时候用射线测试是否与格子地面碰撞, 如果符合条件则创建防守单位, 代码如下:

```

// 按下"创建防守单位按钮"
// 响应创建防守单位按钮
void OnButCreateDefenderUp( BaseEventData data )
{
    GameObject go = data.selectedObject;

```

```

// 创建射线
Ray ray = Camera.main.ScreenPointToRay(Input.mousePosition);
RaycastHit hitinfo;
// 计算是否与地面相碰撞
if (Physics.Raycast(ray, out hitinfo, 1000, m_groundlayer))
{
    Debug.Log(go.name);
    // 如果选中的是一个空闲的格子
    if (TileObject.get.getData(hitinfo.point.x, hitinfo.point.z) ==
        (int)Defender.TileStatus.FREE)
    {
        // 获得选中的位置
        Vector3 pos = new Vector3(hitinfo.point.x, 0, hitinfo.point.z);
        // 使位置位于格子中心
        pos.x = (int)pos.x + TileObject.get.tileSize * 0.5f;
        pos.z = (int)pos.z + TileObject.get.tileSize * 0.5f;

        // 简单根据按钮名称创建不同的防守单位
        if (go.name.Contains("1")) // 创建近战单位
        {
            if (SetPoint(-15)) // 需要消耗 15 个铜钱
                Defender.Create<Defender>(pos, new Vector3(0, 180, 0));
        }
        else // 创建远程单位
        {
            if (SetPoint(-20)) // 需要消耗 20 个铜钱
                Defender.Create<Archer>(pos, new Vector3(0, 180, 0));
        }
    }
}
m_isSelectedButton = false;
}

```

OnButCreateDefenderUp 函数是之前我们在 Start 函数中定义的 UI 回调函数，我们首先使用参数 BaseEventData 获得当前按钮物体，这里简单通过按钮物体的名字判断选中了哪个按钮。在实际项目中，对于比较复杂的情况，可以在创建按钮的时候给每个按钮使用 gameObject.AddComponent 添加一个专用的脚本，在这个脚本中可以记录各种信息，当单击这个按钮后，在回调函数中可以通过 gameObject.GetComponent 获得该脚本，获取相应的数据。

运行游戏，按一下右侧的按钮，然后拖拉鼠标到场景中松开鼠标，即可创建一个相应的防守单位，如图 4-33 所示。

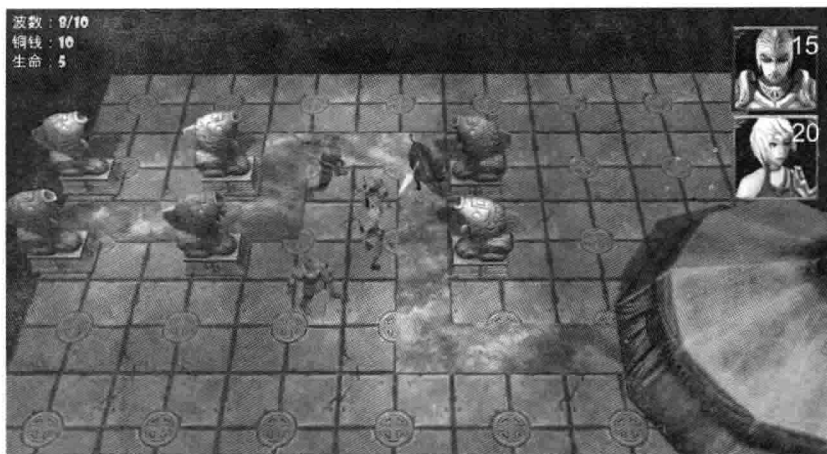


图 4-33 防守单位

4.10 生命条

敌人在受到攻击的时候,我们并不知道它受到了多少伤害,为了能够显示它的剩余生命值,我们需要为它制作一个生命条,需要显示在敌人的身体上方。因为这是一个 3D 游戏,所以我们将使用 3D UI 功能创建这个生命条。

步骤 01 在菜单栏选择【GameObject】→【UI】→【Canvas】在场景中创建一个新的 Canvas,并命名为 Canvas3D,将它的 Render Mode 设为 World Space,使这个 UI 成为一个 3D UI,如图 4-34 所示。

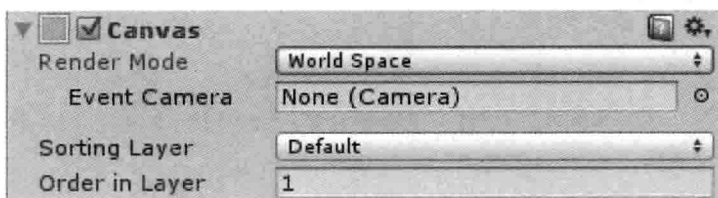


图 4-34 设置为 3D UI

步骤 02 在菜单栏选择【GameObject】→【UI】→【Slider】创建一个滑动条控件,在 Source Image 中指定生命条的背景,因为生命条并不需要滑块,所以将 Handle Slide Area 隐藏或删除,如图 4-35 所示。



图 4-35 生命条背景图

步骤 03 默认 Silder 层级下的 Fill 即是生命条的前景，在 Source Image 中设置前景图片，在 Image Type 中选择 Filled，Fill Method 选择 Horizontal，如图 4-36 所示。

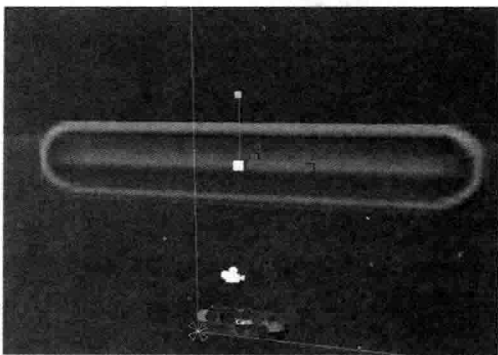
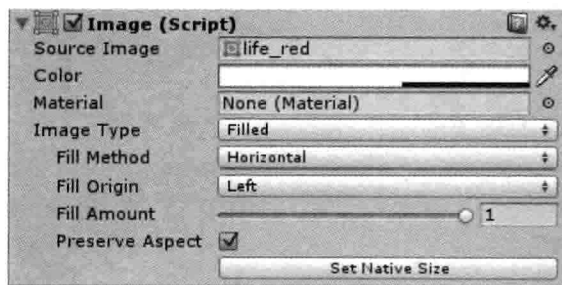


图 4-36 生命条设置

将生命条 Slider 存为 Prefab 保存在 Resource 目录下，并命名为 ui_lifobar，然后删除场景中的 Slider，仅保留 Canvas3D 物体。

步骤 04 打开 Enemy.cs 脚本，添加创建、更新、删除生命条的脚本如下：

```
// 添加生命条相关成员
Transform m_lifobarObj;
UnityEngine.UI.Slider m_lifobar; //slider 控件

// 在 Start 函数中添加创建生命条的代码
void Start ()
{
    // 继续添加代码如下...

    // 在场景中找到 Canvas3D 节点
    GameObject uirroot3d = GameObject.Find("Canvas3D");
    // 读取生命条 prefab
    GameObject prefab = (GameObject)Resources.Load("ui_lifobar");
    // 创建生命条
    GameObject lifebarmobj = (GameObject)Instantiate(prefab);
    m_lifobarObj = lifebarmobj.transform;

    //非常重要，将生命条作为 Canvas3D 的子节点，
    m_lifobarObj.parent = uirroot3d.transform;
    // 缩小生命条以适应场景中角色的大小
    m_lifobarObj.localScale = new Vector3(0.02f, 0.02f, 0.02f);
    m_lifobar = m_lifobarObj.GetComponent<UnityEngine.UI.Slider>();
    // 更新生命条位置等
    UpdateLifobar();
}

// 更新生命条状态的函数，放到 Update 中执行
```



```

void UpdateLifebar()
{
    // 更新生命条
    m_lifebar.value = (float)m_life / (float)m_maxlife;
    // 更新位置
    Vector3 lifebarpos = this.transform.position;
    lifebarpos.y += 2.0f;
    m_lifebarObj.transform.position = lifebarpos;
    // 更新角度
    m_lifebarObj.transform.eulerAngles = Camera.main.transform.eulerAngles;
}
// 在 DestroyMe 函数中添加删除生命条的代码
public void DestroyMe()
{
    // ...
    // 删除生命条
    Destroy(m_lifebarObj.gameObject);
}

```

运行游戏, 在敌人的上方会出现一个生命条, 当敌人受到攻击生命值下降时, 生命条状态会改变, 如图 4-37 所示。在实际的项目中, 如果需要精确的放置生命条在角色头上的位置, 可以在创建 3D 模型的时候专门创建一个节点用来参考生命条的位置。

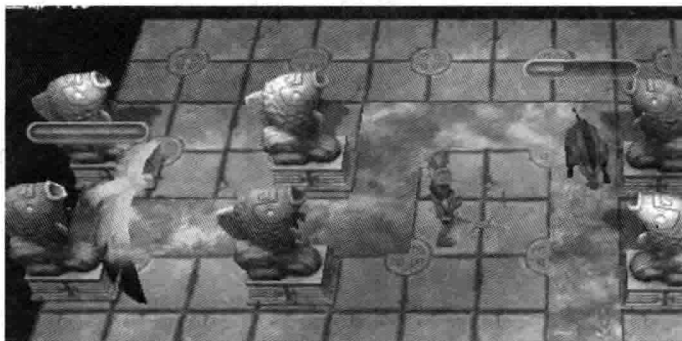


图 4-37 生命条

这个塔防游戏到这里就结束了, 它还非常简陋, 但具备了塔防游戏的基本要素, 如果添加更多的细节和更好的画面, 相信它可以变成一款不错的游戏。

本章的最终示例工程保存在资源文件目录 chapter04_TD 中, 包括 Excel 文件。在资源文件目录 builds\td 中保存有编译好的游戏, 可以直接运行。

4.11 地图编辑器

Unity 的编辑器功能非常强大, 它允许程序员根据需求写出自定义的编辑器, 像在本章的开

始，我们使用了一个自定义的地图编辑器来设置场景的贴图。接下来，我们将介绍如何创建一个这样的编辑器。

4.11.1 创建一个自定义窗口

应用到编辑器的脚本要在前面引用到 `UnityEditor`，并保存在 `Editor` 文件夹内，如下所示：

```
using UnityEditor;
```

创建编辑器窗口的类要继承自 `UnityEditor.EditorWindow`，如下所示：

```
public class TileCreator : UnityEditor.EditorWindow
```

下面的代码在编辑器菜单中添加了一个叫做 `TileCreator` 的菜单，它的功能即是创建一个最基本的窗口，该窗口将应用到 `TileCreator` 这个类中的功能。

```
[MenuItem("Tools/TileCreator")]
static void Create () {
    EditorWindow.GetWindow(typeof(TileCreator));
}
```

我们需要在 `TileCreator` 这个类中添加一个 `OnGUI` 函数，所有的窗口控件都要在该函数中创建，注意这里使用的 UI 函数均来自 `GUILayout` 和 `EditorGUILayout`，使用它们的好处是 UI 会自动排版，我们不必关心 UI 的具体坐标位置，但这种做法效率较低，一般只用于编辑器，不适合在游戏中使用。最后的 UI 如图 4-38 所示。

```
void OnGUI()
{
    // 显示窗口名称
    GUILayout.Label("Tile Creator", EditorStyles.boldLabel);
    // 模型名称
    tileName = EditorGUILayout.TextField("Tile Object Name", tileName);
    // 四边形大小
    polySize = EditorGUILayout.IntSlider("tile size", polySize, 1, 20);
    // x z 轴四边形数量
    xCount = EditorGUILayout.IntSlider("x tile count", xCount, 1, 100);
    zCount = EditorGUILayout.IntSlider("z tile count", zCount, 1, 100);
    // 按钮
    if (GUILayout.Button("Create Tile Object"))
    {
        // 执行 createMesh 函数创建自定义的多边形
        createMesh();
    }
}
```

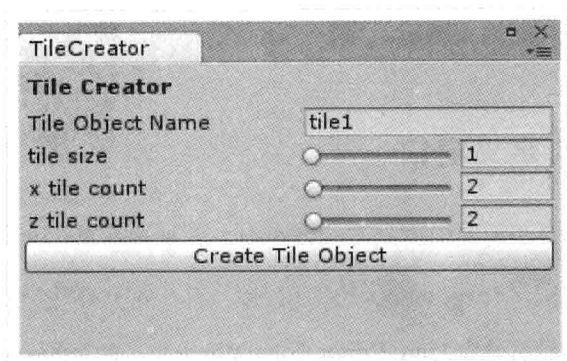


图 4-38 使用 OnGUI 创建的 UI 界面

4.11.2 绘制多边形

在 `TileCreator` 这个类中, `createMesh` 函数负责创建多边形, 实际上, 我们不只可以在编辑器内创建多边形, 也可以在游戏中动态地创建、修改多边形, 下面将简单介绍一下这个过程。

一个多边形由至少 3 个顶点组成, 按顺序画出 3 个顶点, 即形成一个三角面, 通常我们看到的四边面是由两个三角面组成, 两个三角面可以共用 4 个顶点, 因此一个四边的多边形可以只由 4 个顶点组成。

在准备绘制多边形模型之前, 还需要准备以下数据。

顶点: `Vector3` 数组, 每个值是一个顶点的位置。

法线: `Vector3` 数组, 每个值对应一个顶点的法线方向。

UV: `Vector2` 数组, 每个值对应一个顶点的 UV 坐标位置, 一个多边形可以有多个 UV。

三角形: 一个 `int` 数组, 每个值用来记录三角面的 1 个顶点编号, 实际就是顶点数组的长度位置, 连续的三个编号就是画出一个三角面的顶点顺序。尽管一个四方面可以由 4 个顶点组成, 但绘制一个三角面仍需要遍历三个顶点, 因此一个四方面至少需要 6 个三角形编号, 它们中有些可能是指向同一个顶点。

当准备好以上数据后, 下面的代码即可以按指定的数据创建出多边形模型:

```
// 创建 Mesh
Mesh mesh = new Mesh();
// Mesh 的名字
mesh.name = tileName;
// 为 Mesh 指定顶点数组
mesh.vertices = vertices;
// 为 Mesh 指定法线数组
mesh.normals = normals;
// 为 Mesh 指定 UV 数组
mesh.uv = newUV;
// 一个多边形如何使用多张贴图, 即需要多个 subMesh
mesh.subMeshCount = 1;
// 指定三角形数组
```

```

mesh.SetTriangles(newTriangles, 0);
// 计算多边形边界大小
mesh.RecalculateBounds();
// 创建多边形资源
AssetDatabase.CreateAsset(mesh,
                           "Assets/TileEditor/Mesh/" + mesh.name + ".asset");
// 将创建的 Mesh 保存在本地
AssetDatabase.SaveAssets();

```

详细的代码，请参考示例工程中的脚本 TileCreator.cs。

4.11.3 在 Inspector 窗口添加自定义 UI 控件

在 Unity 的编辑器中，当选中一个游戏体后，我们即可以在【Inspector】窗口设置它的详细属性，默认【Inspector】窗口中的选项都是预定的，Unity 提供了 API 可以扩展【Inspector】窗口中的 UI 控件。

以本示例的地图编辑器为例，为了扩展 TileObject 这个类的【Inspector】窗口，我们创建了一个类叫做 TileEditor，继承自 Editor，因为它是一个编辑器脚本，所以必须放到 Editor 文件夹中，其中一些主要代码如下：

```

[CustomEditor(typeof(TileObject))]
public class TileEditor : Editor {

```

为了能获得 TileObject 的相关属性或变量，在 TileEditor 中引用了一个 TileObject 对象，并在初始化时指向关键字 target，代码如下：

```

protected TileObject tileObject;

void OnEnable()
{
    // 获得 target
    tileObject = (TileObject)target;
}

```

接下来，我们需要在【Inspector】窗口绘制自定义的 UI，而绘制 UI 的工作一定要在 OnInspectorGUI 函数中完成，代码如下：

```

public override void OnInspectorGUI()
{
    // 获得 Tile Object 的 Mesh Renderer 数量
    int maxcount = tileObject.GetComponent<MeshRenderer>().sharedMaterials.Length;
    // 显示文字
    GUILayout.Label("Tile Editor");
    // 显示格子大小
    EditorGUILayout.LabelField("Tile Size:" + tileObject.tileSize);
}

```

```

EditorGUILayout.LabelField("X Tile Count:" + tileObject.xTileCount);
EditorGUILayout.LabelField("Z Tile Count:" + tileObject.zTileCount);
// 显示材质数量
EditorGUILayout.LabelField("Materials:" + maxcount);
// UI 空一行
EditorGUILayout.Separator();

// 创建一个开关, 名字叫 Edit, 后面的值 editMode 是一个 bool, 表示是否选中
editMode = EditorGUILayout.Toggle("Edit", editMode);
// 定义工具栏按钮的名称 Materials 和 Data
string[] editModeStr = {"Materials", "Data"};
// 创建一个工具栏, editID 表示当前按钮的 id, 后面是一个 string 数组表示按钮的名称
editID = GUILayout.Toolbar(editID, editModeStr);

// 如果 editID 的值为 0, 显示一个滑动条, 名称为 Material ID
if (editID == 0)
{
    matID = EditorGUILayout.IntSlider("Material ID", matID, 0, maxcount - 1);
}
// 如果 editID 的值为 1, 显示一个滑动条, 名称为 Data ID, 取值范围 0-9
else if (editID == 1)
{
    tileObject.dataID = EditorGUILayout.IntSlider("Data ID",
                                                    tileObject.dataID, 0, 9);
}
// 显示一个分隔符
EditorGUILayout.Separator();
// 绘制默认的窗口选项
DrawDefaultInspector();
}

```

最后的 UI 如图 4-39 所示, 其中 Script 和 Tile Layer 是 TileObject 脚本中的选项, 其他则是在 TileEditor 这个类中额外添加的。

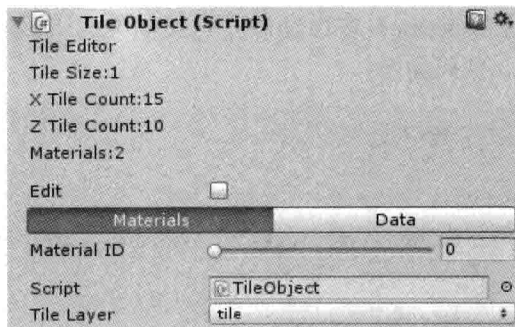


图 4-39 自定义【Inspector】窗口

在本章的地图编辑器中，我们可以进入一种特殊状态，可以在场景中绘制贴图，而不会因为按住鼠标选到其他东西，要实现这样的功能，需要在 `OnSceneGUI` 函数中取消编辑器的选择功能，代码如下：

```
public void OnSceneGUI()
{
    if (editMode) // 如果是编辑模式
    {
        // 取消编辑器的选择功能
        HandleUtility.AddDefaultControl(
            GUIUtility.GetControlID(FocusType.Passive));

        // 获取编辑器 Input 事件
        Event e = Event.current;

        // 如果单击鼠标左键一次
        if (e.isMouse && e.button == 0 && e.clickCount == 1)
        {
            //...
        }
    }
    else
    {
        // 恢复编辑器的选择功能
        HandleUtility.Repaint();
    }
}
```

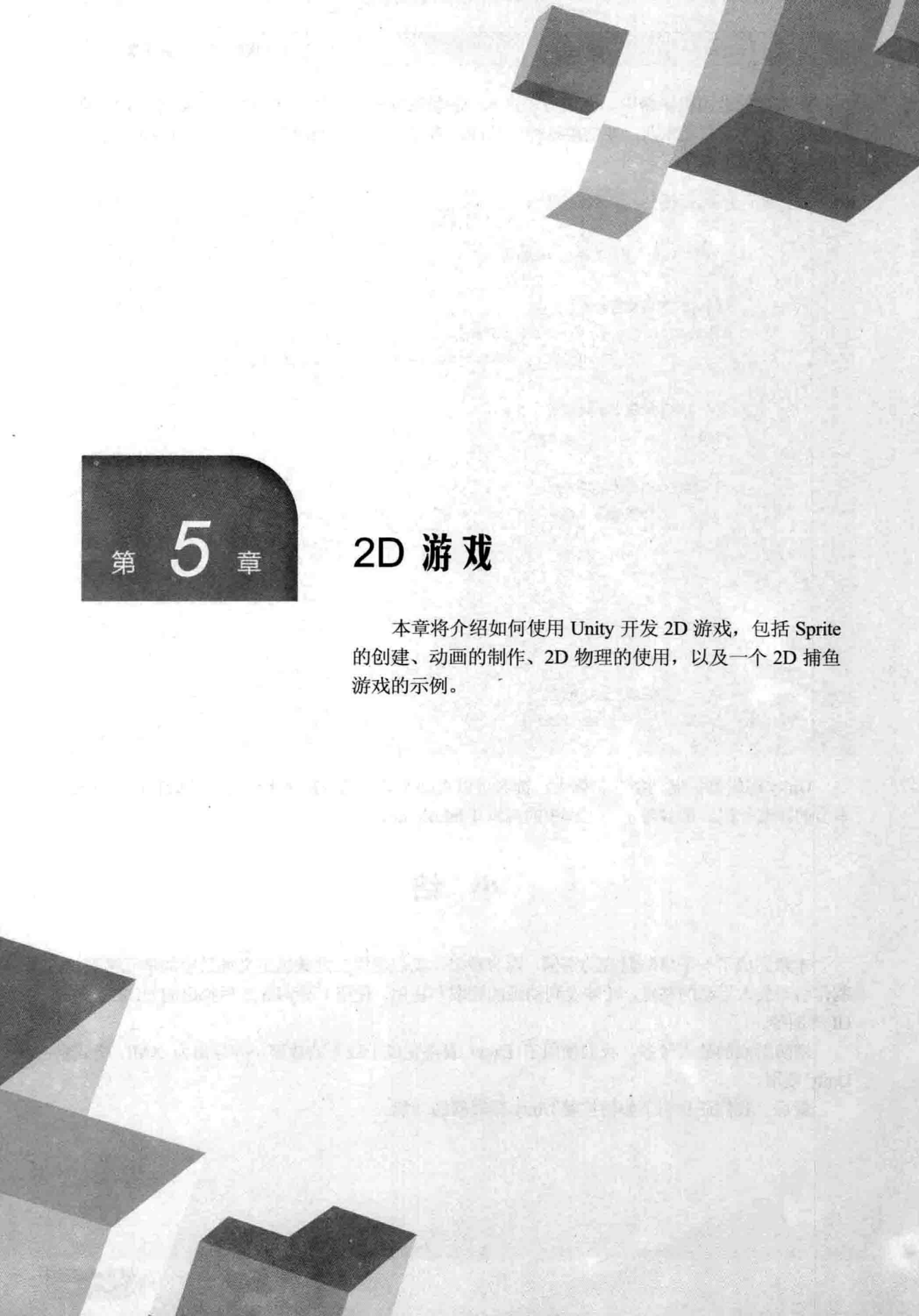
Unity 编辑器扩展功能非常强大，读者可以查阅帮助文档获得更多信息，帮助自己的开发。本节的详细代码，请参考示例工程中的脚本 `TileEditor.cs`。

小 结

本章完成了一个塔防游戏的实例，内容较多，我们使用二维数组定义场景中的单元格数据，制作引导敌人行动的路点，还涉及到动画的读取和使用，使用 Unity4.6 之后推出的 UI 功能完成 UI 界面等。

塔防游戏的敌人较多，我们使用了 Excel 表格完成了敌人的数据，并导出为 XML 格式供 Unity 使用。

最后，我们还介绍了如何扩展 Unity 编辑器的功能。



第 5 章

2D 游戏

本章将介绍如何使用 Unity 开发 2D 游戏，包括 Sprite 的创建、动画的制作、2D 物理的使用，以及一个 2D 捕鱼游戏的示例。

5.1 Unity 2D 系统简介

Unity 或者叫 Unity 3D，直觉上是一个制作 3D 游戏的游戏引擎，但事实上，Unity 也一直被很多公司用于 2D 游戏的制作中，早期比如《Battle Heart》（战斗之心）、《Bad Piggies》（捣蛋猪）等，最近暴雪刚刚推出的 2D 卡牌游戏大作《Heart stone》（炉石传说）也是用的 Unity。在 Unity 的 Asset Store（Unity 官方在线商店）中有很多 Unity 2D 游戏制作的插件，在 Unity4.x 版本之前，很多 2D 游戏都是用这些插件制作的。Unity 升级版本到 4.x 后，终于发布了内置的 2D 游戏制作功能，核心模块包括 Sprite（精灵）的创建、动画的制作和 2D 专用的物理模块，现在，Unity 不但是一个 3D 的游戏引擎，也是一个专业的 2D 游戏引擎了。

5.2 创建 Sprite

2D 游戏的图像部分主要是图片的处理，通常称图片为 Sprite（精灵）。为了提高效率，2D 游戏会将动画帧或不同的小图片拼成一张大图，在游戏运行的时候，再将这张大图的某一部分读出来作为 Sprite 显示在屏幕上。

使用 Unity 制作 Sprite 有两种选择，可以将不同的图片在图像软件中拼接为一张大图，然后导入到 Unity 中，使用 Sprite Editor 工具将这张大图分割成若干个 Sprite 使用。另一种做法是直接将原始图片导入到 Unity 中，使用 Unity 的 Sprite Packer 工具将这些零散的图片打包，再从中读取 Sprite 使用。接下来，将分别介绍两种创建 Sprite 的方式及动画的制作。

5.2.1 使用 SpriteEditor 创建 Sprite

步骤 01 启动 Unity 编辑器，在菜单栏选择【Edit】→【Project Settings】→【Editor】打开编辑器设置窗口，将 Default Behavior Mode 设为 2D，Unity 编辑器将会进入 2D 编辑模式，在这种模式下，导入到工程中的图片默认会被自动设为 Sprite 格式。将 Sprite Packer 设为 Always Enabled，这时在任何情况下都能启动 Sprite Packer，如图 5-1 所示。

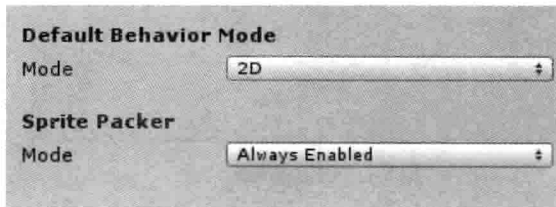


图 5-1 将编辑器设置为 2D 模式

步骤 02 在资源文件目录“rawdata/2d”下面找到图片 envtile.png，将它导入到 Unity 工程中。这是一张由若干个小图组成的图片，如图 5-2 所示。

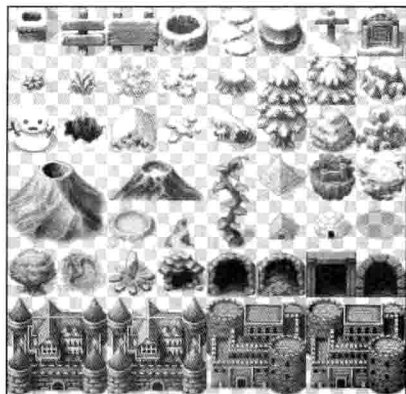


图 5-2 导入的 2D 图片

步骤 03 在 Project 窗口选择导入的图片, 在 Inspector 窗口将 Sprite Mode 设为 Multiple, 这种模式可以将一张大图分割为若干个 Sprite, 如图 5-3 所示。



图 5-3 设置 Sprite

步骤 04 在 Inspector 窗口选择 Sprite Editor 打开编辑窗口, 选择左上角的 Slice, 默认的切割方式为 Automatic (自动的), 将它改为 Grid, 然后在 Pixel Size 中输入每个 Sprite 的尺寸大小, 本例为 32×32 , 默认 Sprite 的轴点为 Center (中心), 选择【Slice】开始分割图片, 如图 5-4 所示。

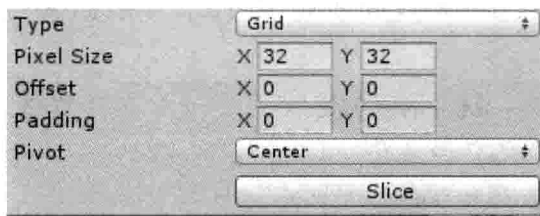


图 5-4 设置 Sprite 尺寸

步骤 05 分割好图片后, 会发现有些 Sprite 的尺寸要大于 32×32 , 可以用鼠标直接选择分割出来的 Sprite, 将无用的删除, 选择需要放大尺寸的 Sprite, 在右下角的 Sprite 窗口中重新设置尺寸, 最后选择【Apply】完成修改, 如图 5-5 所示。

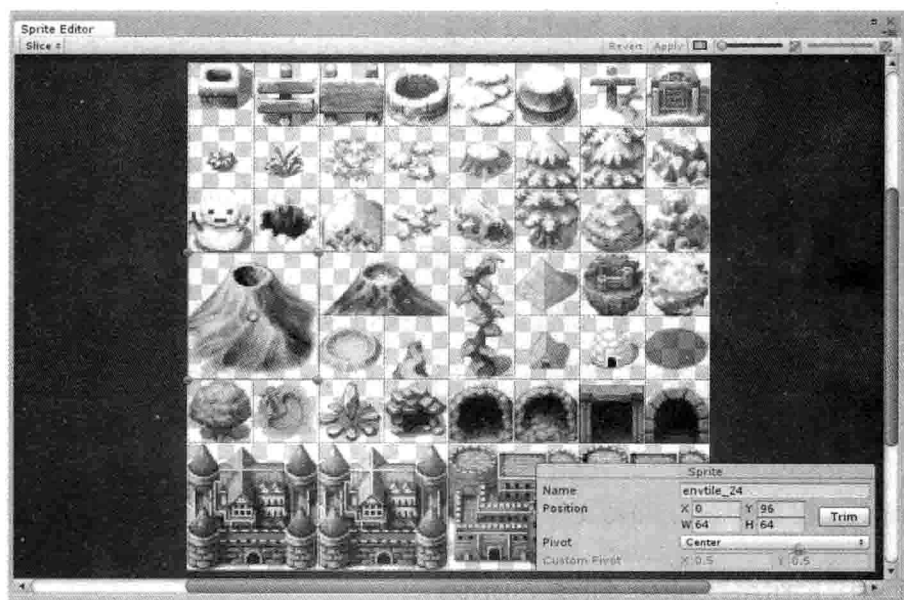


图 5-5 设置单个 Sprite 尺寸

步骤 06 在菜单栏选择【GameObject】→【Create Other】→【Sprite】在场景中创建一个空的 Sprite 实例，在 Inspector 窗口选择 Sprite 选项，指定贴图，如图 5-6 所示。



图 5-6 创建 Sprite 实例

步骤 07 在场景中复制多个 Sprite，并指定不同的贴图，最后的示例效果图如图 5-7 所示。

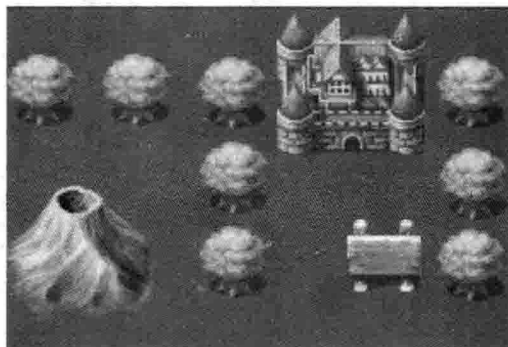


图 5-7 2D Sprite 示例

尽管我们在场景中创建了很多 Sprite，但如果在 Game 窗口选择 Stats 查看运行信息，会发

现 Draw Calls 只有一个, 效率很高, 如图 5-8 所示。

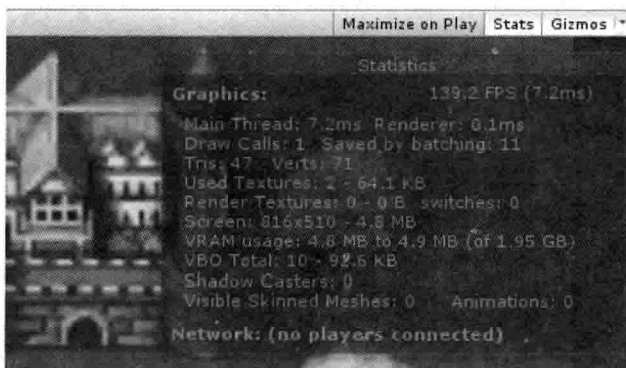


图 5-8 查看 Draw Calls

本节的示例工程文件保存在资源文件目录 chapter05_tileSprite 中。

5.2.2 使用 SpritePacker 创建 Sprite

手动将琐碎的图片拼凑到一张大图上是件很麻烦的事情, 特别是如果需要添加或删除某个图片元素, 每次都要重新整理图片和在 Unity 中的设置。Unity 提供的另一个 Sprite 工具 SpritePacker 可以动态地将零散的图片自动合成, 功能强大, 省去了手动设置的麻烦。下面用一个简单的例子介绍这个工具的使用。

步骤 01 首先要确认在 Editor 设置中将 Sprite Packer 设为 Always Enabled。在资源文件目录 rawdata\2d 复制 player_n.png 等共 8 张图片到 Unity 工程目录内。

步骤 02 在 Project 窗口按住 Shift 键, 同时选中导入的多张图片, 将 Packing Tag 设为 player, 这个 Tag 用来标记哪些图片将会被一同打包。默认图片的轴心点在中心位置, 选择【Custom】允许使用一个自定义的位置, 如图 5-9 所示。

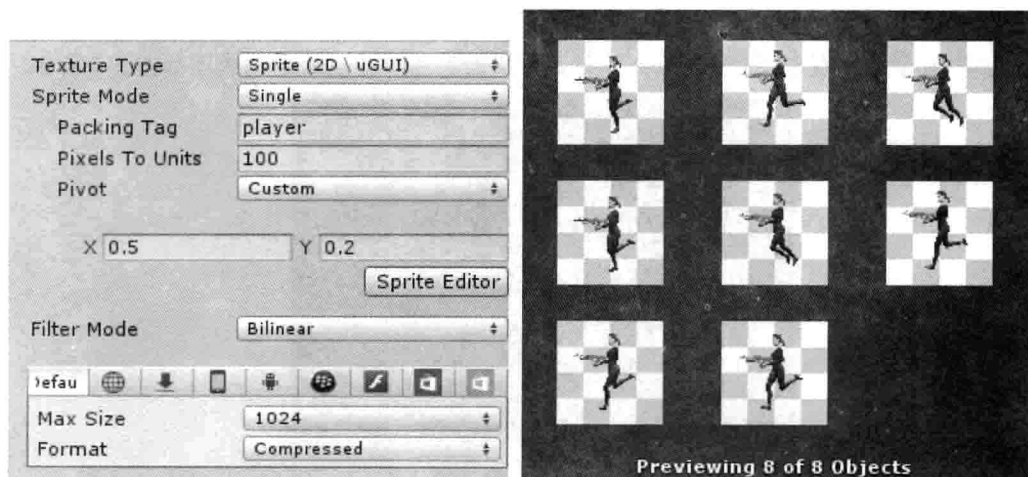


图 5-9 设置 Tag 和轴心点

步骤 03 在菜单栏选择【Window】→【Sprite Packer】, 打开 Sprite Packer 窗口, 选择【Pack】

打包图片，8 张 256×256 的图片被合为 1 张 1024×512 的图片，如图 5-10 所示。注意现在在窗口上方有一个 **player** 选项，它即是我们之前设置的 **Tag** 标签，如果设置有多个 **Tag**，这里则会出现多个选项，我们可以从中选择相应的选项，查看打包后的效果。

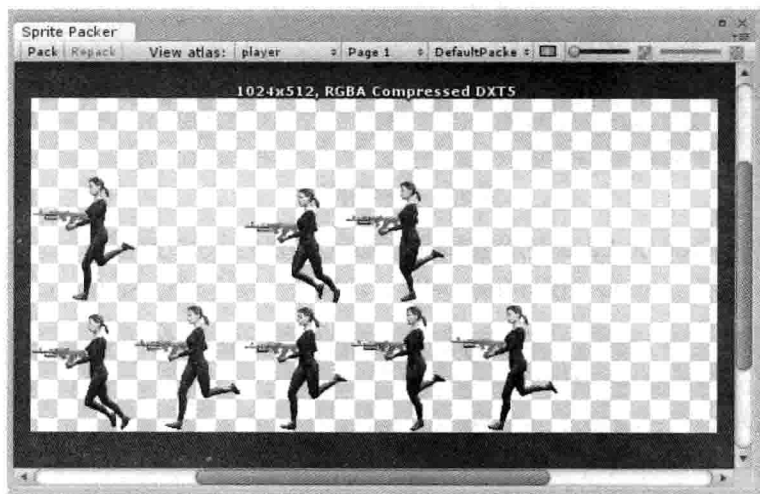


图 5-10 打包图片

前面我们在 Editor 中设置过 Sprite Packer 的模式，Always Enabled 允许在任何情况下使用 Sprite Packer，Enabled For Builds 只有在导出游戏的时候才会使用 Sprite Packer，Disabled 则取消使用 Sprite Packer 的功能，如图 5-11 所示。也就是说，在游戏运行或导出游戏的时候，Unity 会自动为我们打包图片，因此，在 Project 窗口，我们是找不到打包之后的图片（也称作 atlas）资源的。

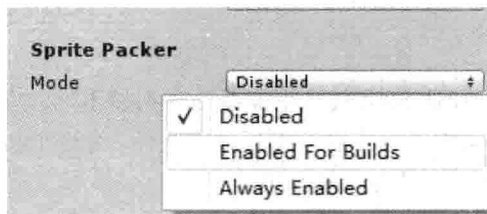


图 5-11 打包图片

本节的示例工程文件保存在资源文件目录 chapter05_spritePacker 中。

5.2.3 图层排序

2D 游戏中的图片之间没有深度关系，都处在一个平面上，但在实际游戏中通常仍需要为它们排序，按视觉上的空间先后顺序显示。在 Sprite 的 **Sorting Layer** 中可以为不同的 Sprite 创建不同的层，每个层有一个名称，层的顺序即是 Sprite 的显示顺序，值越大越显示在前面，在同一个层中的 Sprite 可以通过 **Order in Layer** 来排列显示顺序，如图 5-12 所示。

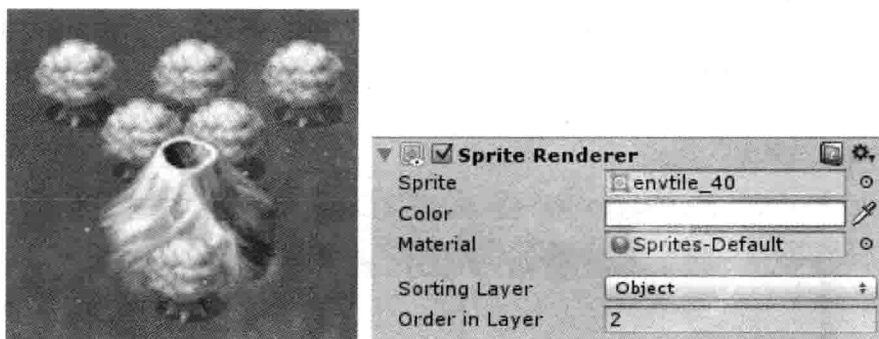


图 5-12 图层排序

Sorting Layer 和 Order in Layer 的值在脚本中都可以获取, 通过脚本可以动态地改变图层的顺序, 下面的代码改变了 Sprite 的层和排序顺序, 注意改变层是通过 Sorting Layer 的名称。

```
SpriteRenderer r = this.GetComponent<SpriteRenderer>();
r.sortingLayerName = "someLayer";
r.sortingOrder = 100;
```

5.3 动画制作

5.3.1 序列帧动画

传统的 2D 动画主要是由若干张表现有连续动作的图片组成, 在 Unity 中制作这种类型的 2D 动画非常简单, 可以将动画序列帧直接保存为动画文件, 使用起来和普通的 3D 动画一样, 使用方式如下。

步骤 01 在 Project 窗口, 按住 Shift 键选中需要的序列帧图片, 将它们拖向 Hierarchy 窗口, 这时 Unity 会自动弹出一个窗口保存动画文件, 为这个动画文件命名, 选择【保存】将动画文件保存在工程内, 如图 5-13 所示。



图 5-13 保存序列帧动画文件

步骤 02 上一步在创建动画的同时，也在场景中创建了一个 Sprite 文件。运行游戏，即可看到 2D 的序列帧动画效果，如图 5-14 所示。

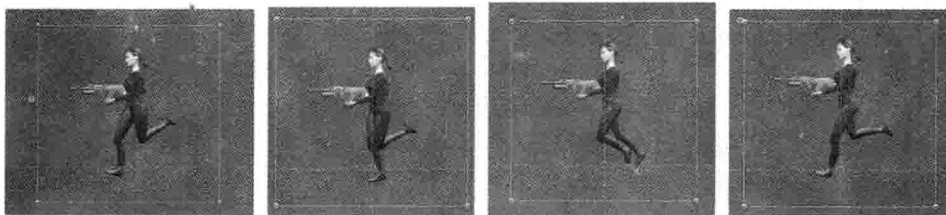


图 5-14 序列帧动画

步骤 03 在 Project 窗口多出了两个文件，如图 5-15 所示，player@run 是之前保存的动画文件，player0028 是自动创建的动画控制器。另外，在场景中创建的 Sprite 会被自动添加一个 Animator 组件。双击动画控制器文件打开 Animator 窗口可以像设置 3D 动画一样设置 2D 的动画。

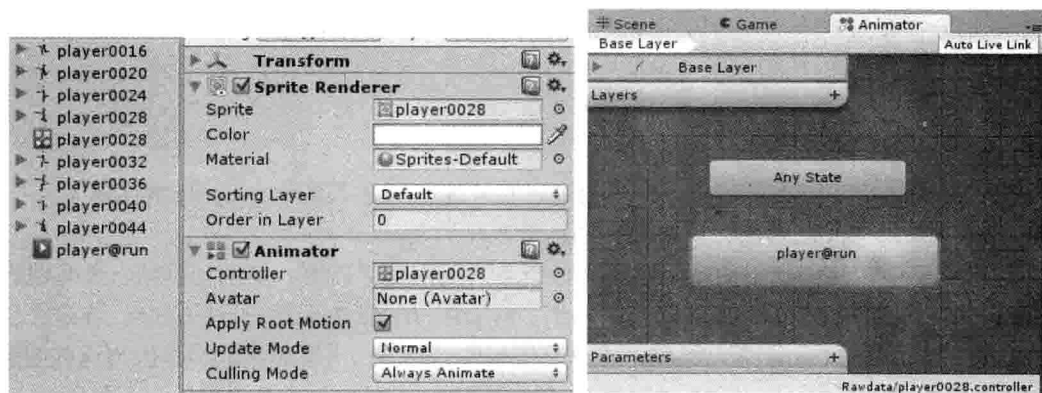


图 5-15 序列帧动画

5.3.2 使用脚本实现序列帧动画

因为 2D 的序列帧动画只是将图片一张张按顺序显示出来，因此使用脚本实现序列帧动画比较简单，如果是批量生产，比如从配置文件中读取资源数据，使用脚本实现动画效果效率也更高一些，下面是一个示例。

步骤 01 新建一个工程，复制资源文件目录 rawdata/2d 下的 player0016.png 等贴图文件到当前工程的 Resources 目录内。

步骤 02 在 Unity 的菜单栏选择【GameObject】→【Create Empty】创建一个空的游戏体，为这个游戏体添加一个脚本，命名为 SpriteAnimator。

步骤 03 在 SpriteAnimator.cs 脚本中添加代码如下：

```
using UnityEngine;
using System.Collections;

/// Sprite 动画播放器
```

```

public class SpriteAnimator : MonoBehaviour {

    /// Sprite 渲染器
    protected SpriteRenderer m_sprite;
    /// Sprite 动画帧
    protected Sprite[] m_clips;
    /// 动画计时器 (默认每隔 0.1 秒更新一帧)
    protected float timer = 0.1f;
    /// 贴图文件名前缀
    public string m_sprite_name = "player00";
    /// 贴图文件起始帧数字
    public int m_start_frame = 0;
    /// 动画的帧数
    public int m_frame_count = 8;
    /// 跳帧数量
    public int m_offset_frame = 4;
    /// 当前的帧数
    protected int m_frame = 0;
}

```

SpriteRenderer 是 Sprite 的渲染器, 只有通过它才能在屏幕上显示出 Sprite 的图像。

m_clips 是一个 Sprite 数组, 用来保存动画所需要的 Sprite。

在这个例子中, 我们使用的动画贴图文件命名都采用数字后缀, 表示动画帧, 并且跳帧, 如 “player0016, player0020, player0024……”。m_start_frame 表示第一帧的数字, 在这个例子中是指 16, m_offset_frame 表示跳帧, 在这个例子中是指 4。出现跳帧数字是因为这些图像都是通过 3D 软件渲染而成, 在渲染的时候进行了跳帧设置, 否则这个动画需要几十张图片来表现。

步骤 04 在 Start 函数中添加代码如下:

```

void Start () {
    /// 初始化动画帧数组
    m_clips = new Sprite[m_frame_count];
    for (int i = 0; i < m_frame_count; i++)
    {
        /// 当前帧
        int currentframe = m_start_frame + i * m_offset_frame;
        /// 计算贴图名称
        string spritename = string.Format(m_sprite_name + "{0:D2}", currentframe);
        /// 从 Resources 目录读取贴图
        Texture2D texture = (Texture2D)Resources.Load(spritename);
        /// 由贴图创建出 Sprite
        m_clips[i] = Sprite.Create(texture,
                                   new Rect(0, 0, texture.width, texture.height),
                                   new Vector2(0.5f, 0.2f));
    }
}

```



```

    }
    // 为当前 GameObject 添加一个 Sprite 渲染器
    m_sprite = this.gameObject.AddComponent<SpriteRenderer>();
    // 设置第 1 帧的 Sprite
    m_sprite.sprite = m_clips[m_frame];
}

```

在 Start 函数中, 首先创建了一个 Sprite 数组 m_clips, 这个数组用来保存动画帧图像。我们从 Resources 目录内读取贴图文件, 然后使用 Sprite.Create 创建 Sprite, 并放到 m_clips 数组中。注意, 在创建 Sprite 的时候, 需要设置 Rect 矩形面积, 我们可以只取得原始贴图的一部分图像, 当前的例子我们获取了整个图像。

步骤 05 在 Update 函数中添加代码如下:

```

void Update () {
    // 更新时间
    timer-=Time.deltaTime;
    if ( timer<=0 )
    {
        timer = 0.1f;
        // 更新帧
        m_frame++;
        if (m_frame >= m_frame_count)
            m_frame = 0;
        // 更新 Sprite 动画
        m_sprite.sprite = m_clips[m_frame];
    }
}

```

这里的代码每隔 0.1 秒, 更新到下一帧 Sprite, 因此产生了动画效果。如果动画已经播放到最后一帧, 则回到第 1 帧。

现在运行游戏, 可以看到与 5.3.1 节类似的动画效果, 不过存放在 Resources 中的 Sprite 不能使用 Sprite Packer 工具。本节的示例工程文件保存在资源文件目录 chapter05_scriptanimation 中。

5.3.3 骨骼动画

如果需要表现细腻的序列帧动画效果, 需要很多图片, 相当消耗资源, 因此游戏中的序列帧动画往往是一定程度跳帧的, 但跳帧后动画流畅度会大幅下降。现在也有很多 2D 游戏, 将 2D 的角色拆分为若干个 Sprite, 用骨骼绑定起来, 用制作 3D 动画的方式制作 2D 动画, 在更加节约资源的同时又加强了动画的流畅性, 下面是一个示例。

步骤 01 新建一个工程, 复制资源文件目录 rawdata/2d/下的 warrior.png 到当前工程目录内, 这是一个角色图片, 角色的不同位置是分开的, 如图 5-16 所示。



图 5-16 原始图片资源

- 步骤 02** 将角色图片的 Sprite Mode 设为 Multiple, 打开 Sprite Editor, 选择左上角的【Slice】, 选择【Automatic】模式, 然后选择【Slice】自动切分图片, 选择每个切分部分, 移动上面的圆点设置轴心位置, 如图 5-17 所示。

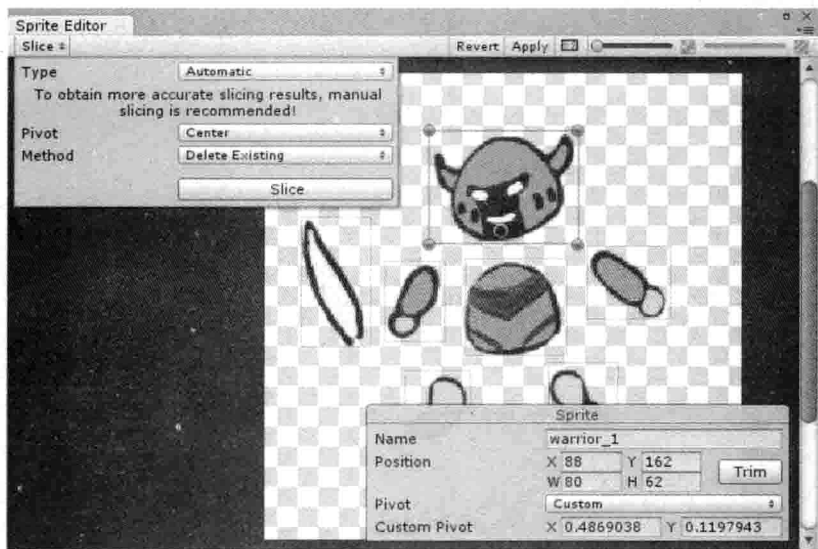


图 5-17 自动切分图片

- 步骤 03** 在场景中新建一个空的 GameObject, 将角色不同位置的图片放到它的层级下面作为子物体, 拼接成完成的角色形像, 注意不同位置的层级关系, 如图 5-18 所示。

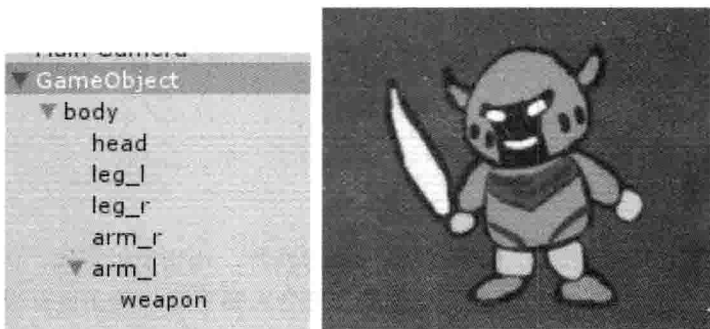


图 5-18 拼接角色

- 步骤 04** 确定当前选择是角色的最顶级物体, 在菜单栏选择【Window】→【Animation】打开动画编辑窗口, 选择左上方的录制按钮开始为角色录制动画, 滑动时间轴到不同

时间,然后在场景中位移或旋转角色的不同部分,即可完成动画的制作,如图 5-19 所示。



提示

Ctrl+C 组合键和 Ctrl+V 组合键可以复制、粘贴关键帧。

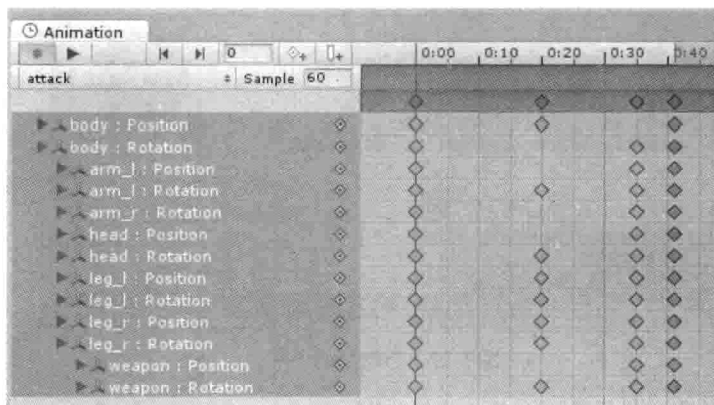


图 5-19 动画帧

最后的动画效果如图 5-20 所示,本节的示例工程文件保存在资源文件目录 chapter05_boneanimation 中。



图 5-20 动画

5.4 2D 物理

针对 2D 系统,Unity 提供了专门的 2D 物理功能,下面是一个示例,移动鼠标滑动一个球体,使它向相反方向弹出去,在场景中有一些方块,如果球体弹到方块,则会将方块撞飞。

- 步骤 01** 新建一个 Unity 的 2D 工程,复制资源文件目录 rawdata/2d/中的 box.png 图片到当前工程中。
- 步骤 02** 将 box.png 的 Sprite Mode 设为 Multiple,然后自动分割,最后选择【Apply】确定,如图 5-21 所示。

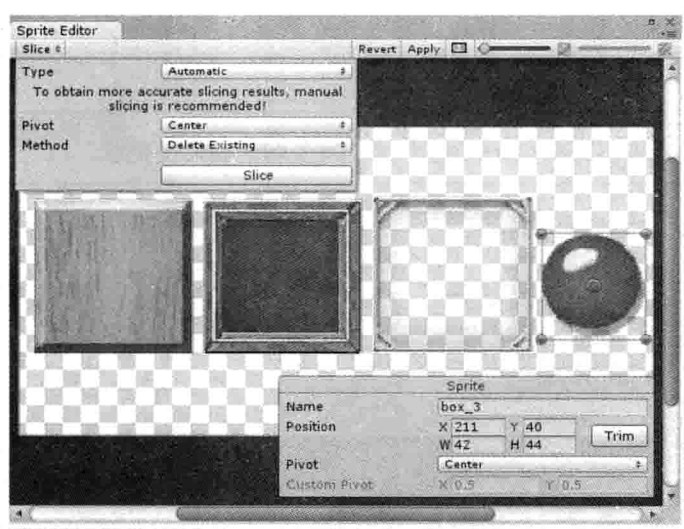


图 5-21 自动分割图片

步骤 03 这一步将制作场景的地面。在菜单栏选择【GameObject】→【Create Other】→【Sprite】创建一个 Sprite 并指定贴图，然后在菜单栏选择【Component】→【Physics 2D】→【Rigidbody 2D】添加一个 2D 刚体组件，并将 Is Kinematic 选中使其不受重力影响。在菜单栏选择【Component】→【Physics 2D】→【Box Collider 2D】添加一个 2D 矩形碰撞体组件，最后将 Sprite 复制多个组成地面，如图 5-22 所示。

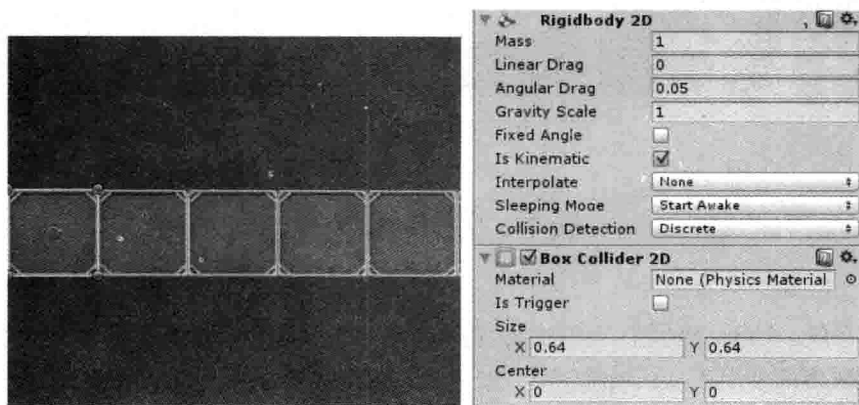


图 5-22 设置地面

步骤 04 重复步骤 2 创建不同的 Sprite 作为物理碰撞的方块，唯一与步骤 2 不同的是不要将 Is Kinematic 选中，如图 5-23 所示。

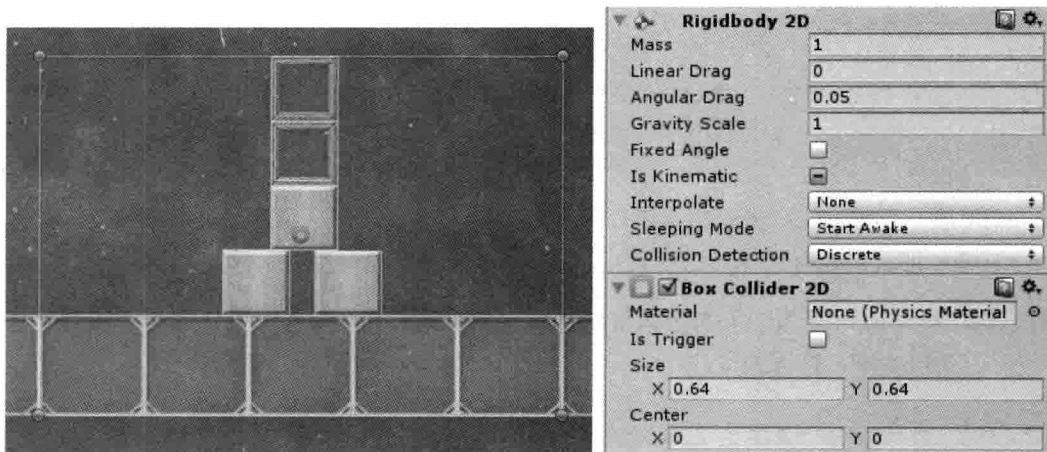


图 5-23 设置方块

步骤 05 创建一个 Sprite 作为球，添加 RigidBody 2D 组件，将 Is Kinematic 选中，我们将在后面使用脚本打开这个选项。在菜单栏选择【Component】→【Physics 2D】→【Circle Collider 2D】添加一个 2D 圆形碰撞体组件，最后的效果如图 5-24 所示。

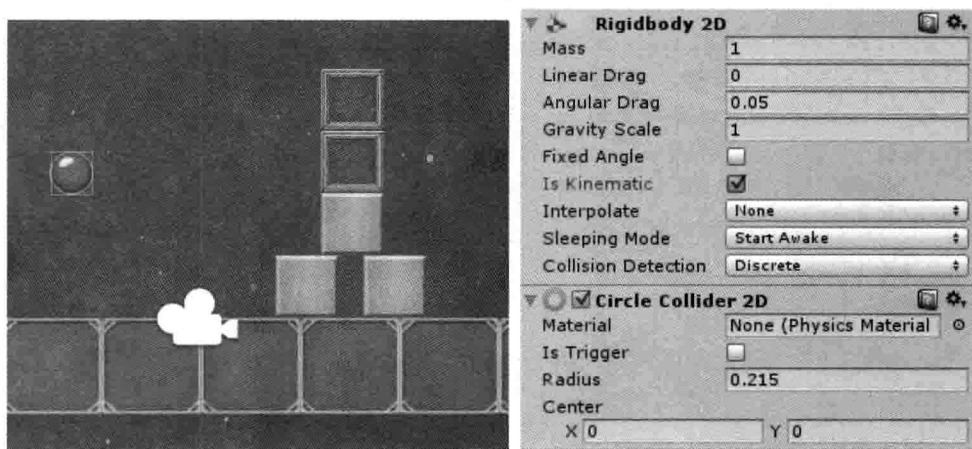


图 5-24 设置球

步骤 06 选中球的 Sprite，添加脚本 Ball.cs，添加代码如下：

```
using UnityEngine;
using System.Collections;

public class Ball : MonoBehaviour {
    /// 是否碰到球
    bool m_isHit = false;
    /// 碰到球的位置
    Vector3 m_startPos;
    /// 球的渲染器
```



```

SpriteRenderer m_spriteRenderer;
void Start () {
    m_spriteRenderer = this.GetComponent<SpriteRenderer>();
}
}

```

m_isHit 是一个布尔值, 记录鼠标操作是否碰到球。

m_startPos 记录鼠标碰到球时的位置。

步骤 07 添加 IsHit 函数, 判断当前鼠标位置 (在手机上就是手指的位置) 是否碰到了球体的 Sprite, 代码如下:

```

bool IsHit()
{
    m_isHit = false;
    // 获得鼠标位置
    Vector3 ms = Input.mousePosition;
    // 将鼠标位置由屏幕坐标转为世界坐标
    ms = Camera.main.ScreenToWorldPoint(ms);
    // 获得球的位置
    Vector3 pos = this.transform.position;
    // 获得球 Sprite 的宽和高 (注意宽和高不是图片像素值的宽和高)
    float w = m_spriteRenderer.bounds.extents.x;
    float h = m_spriteRenderer.bounds.extents.y;
    // 判断鼠标的位置是否是在 Sprite 的矩形范围内
    if (ms.x > pos.x - w && ms.x < pos.x + w &&
        ms.y > pos.y - h && ms.y < pos.y + h )
    {
        m_isHit = true;
        return true;
    }
    return m_isHit;
}
}

```

步骤 08 在 Update 函数中添加代码如下, 先通过按住鼠标记录鼠标起始位置, 再通过释放鼠标记录结束位置, 通过两个位置算出矢量方向, 给球加一个力, 发射出去。

```

void Update () {
    // 如果点击鼠标左键并且碰到球
    if (Input.GetMouseButtonDown(0) && IsHit() )
    {
        // 记录起始位置
        m_startPos = Input.mousePosition;
    }
}

```

```

//当松开鼠标
if (Input.GetMouseButtonUp(0) && m_isHit)
{
    Vector3 endPos = Input.mousePosition;
    // 计算球的运动方向
    Vector3 v = (m_startPos - endPos) * 3.0f;
    // 使 isKinematic 为 false
    this.rigidbody2D.isKinematic = false;
    // 向球加一个力
    this.rigidbody2D.AddForce(v);
}
}

```

运行游戏，选中球向后滑动鼠标，释放鼠标后会将球射出，如果碰到方块，会将方块弹飞，如图 5-25 所示。本节的示例工程文件保存在资源文件目录 chapter05_physics2d 中。

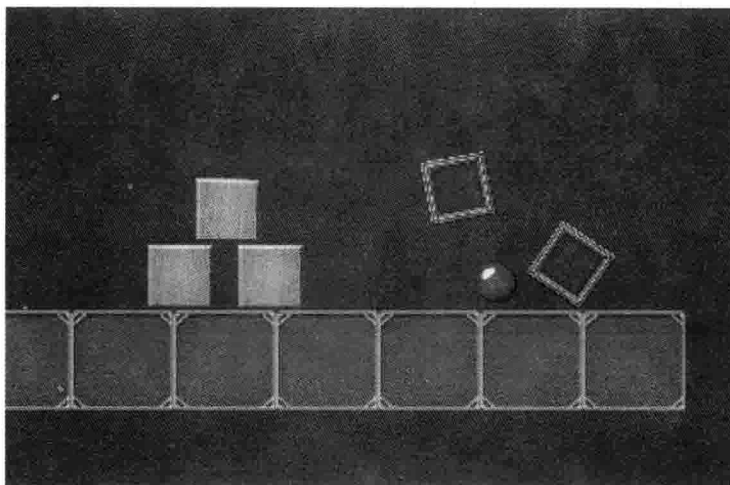


图 5-25 物理碰撞

5.5 捕鱼游戏

5.5.1 游戏玩法

接下来，我们将使用 Unity 的 2D 功能完成一个相对完整的 2D 游戏实例。游戏的玩法比较简单，在屏幕上有很多不同的鱼来回游动，我们的任务则是操作屏幕下方的一门大炮向鱼群开火，将鱼消灭（或者称捕获）。

5.5.2 准备 2D 资源

步骤 01 新建一个工程,复制资源文件目录 rawdata\2d\fishgame 下的所有图片到当前工程目录内,这里包括背景、鱼、大炮、特效等图片,如图 5-26 所示。

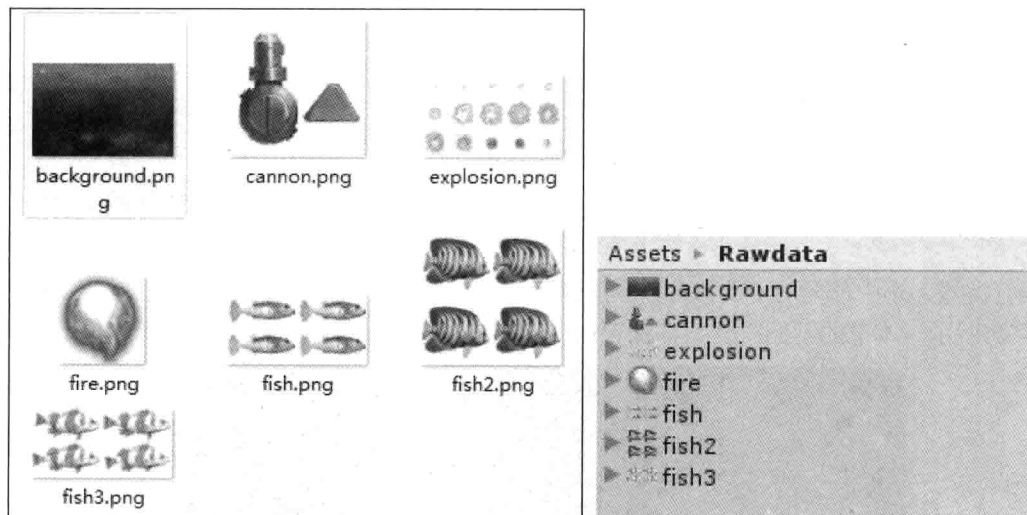


图 5-26 图片资源

步骤 02 将鱼、大炮和爆炸效果的 Sprite 设为 Multiple,使用 Sprite Editor 进行切割,注意大炮的 Sprite 需要将轴心点设置到炮的圆轴中心,如图 5-27 所示。

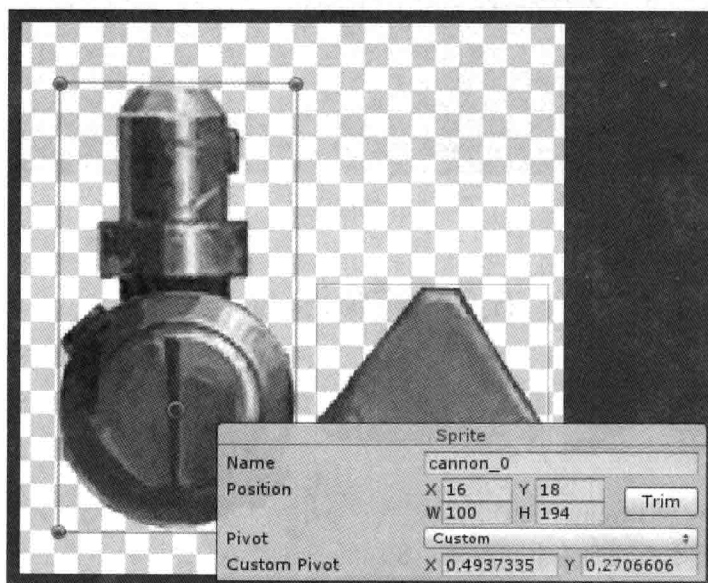


图 5-27 设置大炮的轴心点

步骤 03 将背景和大炮的 Sprite 放到场景中,在菜单栏选择【Edit】→【Project Settings】→【Tags And Layers】,在 Sorting Layers 创建 3 个新的层。然后选择背景和大炮的

Sprite，将背景设到 background 层，将大炮设到 weapon 层，如图 5-28 所示。

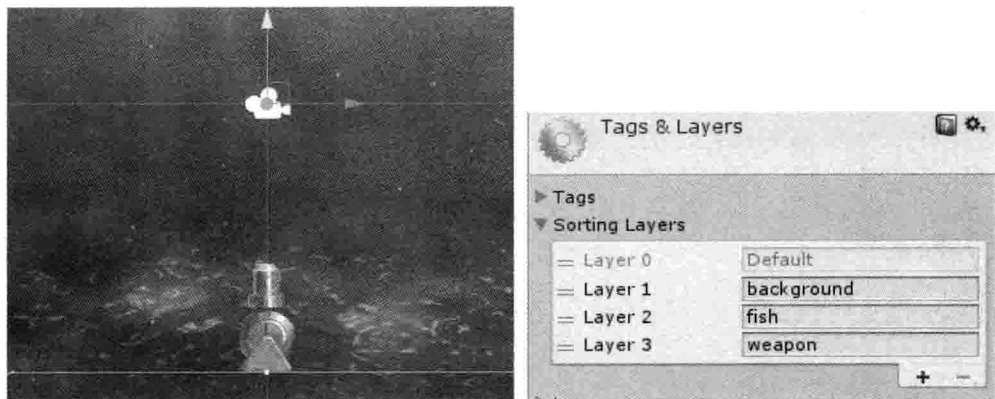


图 5-28 设置背景和层

步骤 04 同时选择鱼的切割 Sprite，将其拖放到场景中，这时会自动创建序列帧动画。使用相同的方式为所有的鱼和爆炸制作动画，如图 5-29 所示。

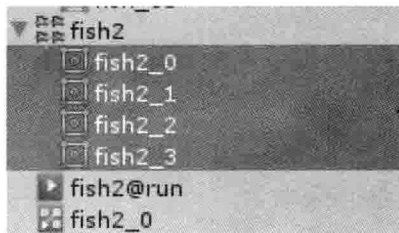


图 5-29 创建动画

步骤 05 为开火、带有动画的鱼和爆炸 Sprite 制作 Prefab，并存放到 Resources 文件夹内，如图 5-30 所示。将鱼 Sprite 的 Sorting Layer 设为 fish，将开火和爆炸设为 weapon。创建完成 Prefab 后，可以删除场景中的这些 Sprite。

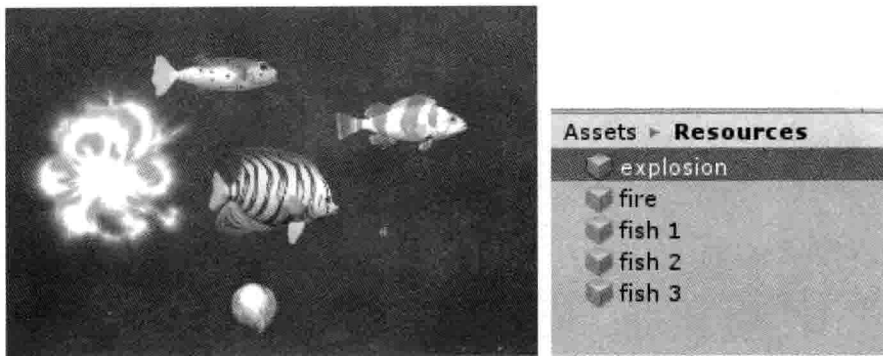


图 5-30 制作 Prefab

5.5.3 创建鱼

在准备好美术资源后，接下来我们将使用脚本创建一条会游来游去的鱼。

步骤 01 创建脚本 Fish.cs, 代码如下:

```

using UnityEngine;
using System.Collections;

public class Fish : MonoBehaviour {
    /// 移动速度
    protected float m_moveSpeed = 2.0f;
    /// 生命值
    protected float m_life = 10;
    public enum Target
    {
        Left=-1,
        Right=1
    }
    /// 移动目标 (方向)
    public Target m_target = Target.Left;
    /// 目标位置
    protected Vector3 m_targetPosition;
}

```

`m_moveSpeed` 是鱼的移动速度; `m_target` 用来记录当前的移动方向是向屏幕左方还是向右方; `m_targetPosition` 用来记录移动的目标位置。

步骤 02 添加 `SetTarget` 函数, 用来设置当前的移动目标位置。添加 `UpdatePosition` 函数用来更新移动位置, 代码如下:

```

void Start () {
    SetTarget();
}

void Update () {
    UpdatePosition ();
}

/// 更新位置
void UpdatePosition()
{
    /// 向目标位置移动
    Vector3 pos = Vector3.MoveTowards(this.transform.position, m_targetPosition,
                                      m_moveSpeed*Time.deltaTime);

    /// 如果到达目标位置则改变目标
    if (Vector3.Distance(pos, m_targetPosition) < 0.1f)
    {
        m_target = m_target==Target.Left ? Target.Right : Target.Left;
        SetTarget();
    }
}

```



```

        this.transform.position = pos;
    }

    // 设置目标位置
    void SetTarget()
    {
        // 随机值
        float rand = Random.value;
        // 设置 Sprite 翻转方向
        Vector3 scale = this.transform.localScale;
        scale.x = Mathf.Abs(scale.x) * (m_target == Target.Right ? 1 : -1);
        this.transform.localScale = scale;
        // 取得摄像机 z 轴位置
        float cameraz = Camera.main.transform.position.z;
        // 设置目标位置
        m_targetPosition = Camera.main.ViewportToWorldPoint(
            new Vector3((int)m_target, 1 * rand, - cameraz));
    }

```

步骤 03 为了看一下脚本的效果，可以将 Resources 下鱼的 Prefab 拖入场景，指定 Fish.cs 脚本。运行游戏，则会看到鱼在场景中来回游动。

5.5.4 创建鱼群生成器

步骤 01 继续修改 Fish.cs，添加代码如下：

```

public delegate void VoidDelegate( Fish fish );
public VoidDelegate OnDeath;
/// 创建一个 Fish 实例
public static Fish Create(GameObject prefab, Target target, Vector3 pos )
{
    GameObject go = (GameObject)Instantiate(prefab, pos, Quaternion.identity);
    Fish fish = go.AddComponent<Fish>();
    fish.Init(target);
    return fish;
}
void Init(Target target)
{
    m_target = target;
}
void SetDamage( int damage )
{
    m_life -= damage;
    if (m_life <= 0)

```

```

    {
        OnDeath(this);
        Destroy(this.gameObject);
    }
}

```

OnDeath 是一个事件, 当鱼死亡的时候, 则会触发这个事件; Create 函数是一个静态函数, 使用它可以快速地通过代码创建鱼的实例; SetDamage 函数处理当鱼受到伤害时, 以减少生命值。

步骤 02 创建脚本 FishSpawn.cs, 它用来生成鱼群, 代码如下:

```

using UnityEngine;
using System.Collections;

public class FishSpawn : MonoBehaviour {
    /// 生成计时器
    public float timer = 0;
    /// 最大生成数量
    public int max_fish = 30;
    /// 当前鱼的数量
    public int fish_count = 0;
    void Update () {
        timer -= Time.deltaTime;
        if (timer <= 0)
        {
            // 重新计时
            timer = 2.0f;
            // 如果鱼的数量达到最大数量则返回
            if (fish_count >= max_fish)
                return;

            // 随机 1、2、3 产生不同的鱼
            int index = 1 + (int)(Random.value * 3.0f);
            if (index > 3)
                index = 3;
            // 更新鱼的数量
            fish_count++;
            // 读取鱼的 prefab
            GameObject fishprefab = (GameObject)Resources.Load("fish " + index);
            // 取得摄像机 z 轴位置
            float cameraz = Camera.main.transform.position.z;
            // 鱼的初始随机位置
            Vector3 randpos = new Vector3( Random.value, Random.value, - cameraz );
            randpos = Camera.main.ViewportToWorldPoint(randpos);
            // 鱼的随机初始方向

```

```

        Fish.Target target =
            Random.value > 0.5f ? Fish.Target.Right : Fish.Target.Left;
        // 创建鱼的实例
        Fish f = Fish.Create(fishprefab, target, randpos);
        // 注册鱼的死亡消息
        f.OnDeath+=OnDeath;
    }
}

void OnDeath( Fish fish )
{
    // 更新鱼的数量
    fish_count--;
}
}

```

这里的代码主要集中在 Update 函数中，每隔 2 秒创建一个鱼的实例，在创建的过程中，使用随机数随机鱼的初始位置、方向和鱼的 Prefab。

步骤 03 在菜单栏选择【GameObject】→【Create Empty】创建一个空游戏体，指定 FishSpawn.cs 脚本。运行游戏，会不断出现新的鱼游来游去，如图 5-31 所示。

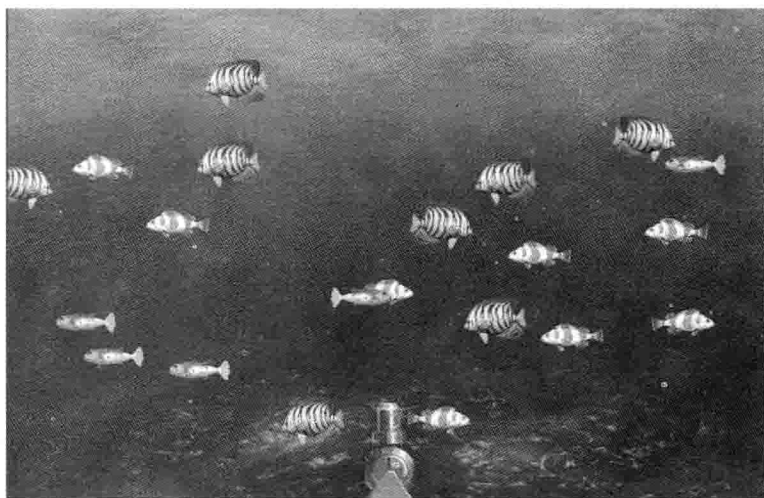


图 5-31 鱼群

5.5.5 创建大炮

现在，游戏里已经有足够的鱼供我们捕猎了，接下来将实现大炮的功能向鱼群开火。

步骤 01 在创建大炮之前，先要完成子弹的脚本。创建脚本 Fire.cs，添加代码如下：

```

using UnityEngine;
using System.Collections;

```

```

public class Fire : MonoBehaviour {
    // 移动速度
    float m_moveSpeed = 10.0f;
    // 创建子弹
    public static Fire Create( Vector3 pos, Vector3 angle )
    {
        // 读取子弹 Sprite prefab
        GameObject prefab = Resources.Load<GameObject>("fire");
        // 创建子弹 Sprite 实例
        GameObject fireSprite = (GameObject)Instantiate(prefab,
                                                         pos, Quaternion.Euler(angle));
        Fire f = fireSprite.AddComponent<Fire>();
        // 2 秒后销毁自己
        Destroy(fireSprite, 2.0f);
        return f;
    }
    // Update is called once per frame
    void Update () {
        // 更新位置
        this.transform.Translate(new Vector3(0, m_moveSpeed * Time.deltaTime, 0));
    }
}

```

Create 函数将创建一个子弹的实例，并初始化子弹的位置和旋转角度。
在 Update 函数中，持续更新子弹的位置，沿着自身 Y 轴移动。

步骤 02 创建脚本 Canon.cs，添加代码如下：

```

using UnityEngine;
using System.Collections;

public class Cannon : MonoBehaviour {
    // 射击计时器
    float m_shootTimer = 0;
    void Update () {

        m_shootTimer -= Time.deltaTime;
        UpdateInput();
    }

    void UpdateInput()
    {
        // 获得鼠标位置
        Vector3 ms = Input.mousePosition;
        ms = Camera.main.ScreenToWorldPoint(ms);
    }
}

```

```

// 大炮的位置
Vector3 mypos = this.transform.position;
// 按鼠标左键开火
if (Input.GetMouseButton(0))
{
    // 计算鼠标位置与大炮位置之间的角度
    Vector2 targetDir = ms - mypos;
    float angle = Vector2.Angle(targetDir, Vector3.up);
    if (ms.x > mypos.x)
        angle = -angle;
    this.transform.eulerAngles = new Vector3(0, 0, angle);
    if (m_shootTimer <= 0)
    {
        m_shootTimer = 0.1f;
        // 开火, 创建子弹实例
        Fire.Create(this.transform.TransformPoint(0, 1, 0),
                    new Vector3(0, 0, angle));
    }
}
}
}

```

在 `UpdateInput` 函数中, 计算出大炮到鼠标位置的角度, 旋转大炮, 并发射子弹。运行游戏, 按鼠标左键, 发射子弹, 如图 5-32 所示。

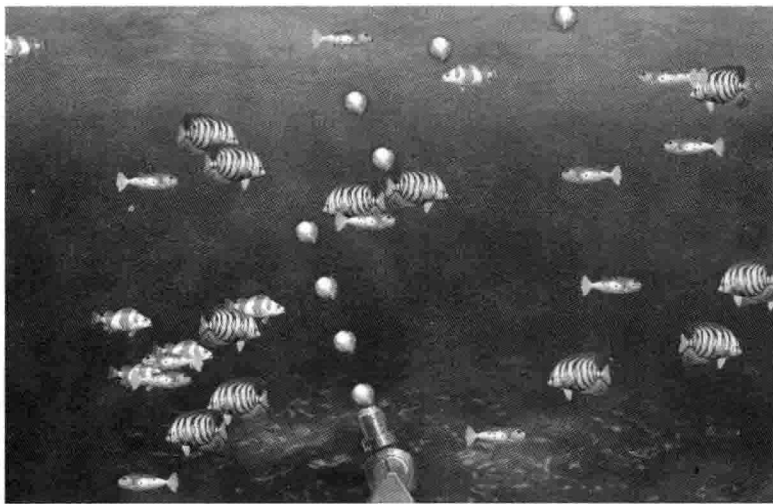


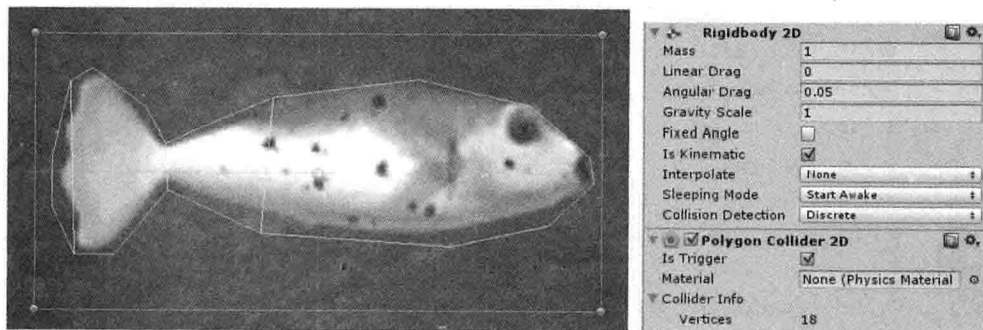
图 5-32 发射子弹

5.5.6 物理碰撞

现在虽然大炮可以发射子弹, 但还不能消灭鱼。接下来我们将添加碰撞功能, 使子弹可以

碰撞到鱼。

步骤 01 在 Resources 文件夹中选择鱼的 Prefab, 为它们添加 Rigidbody 2D 和 Polygon Collider 2D 组件, 分别将 Is Kinematic 和 Is Trigger 选中, 按住 Shift 键, 可以改变多边形碰撞体顶点位置或增加顶点, 按住 Ctrl 键可以删除顶点, 如图 5-33 所示。



图题 5-33 设置鱼的碰撞

步骤 02 重复步骤 1 对子弹 Prefab 做同样的处理。

步骤 03 修改 Fire.cs 脚本, 添加处理碰撞的代码。

```
void OnTriggerEnter2D(Collider2D other)
{
    Fish f = other.GetComponent<Fish>();
    if (f == null)
        return;
    else
        f.SetDamage(1);
    Destroy(this.gameObject);
}
```

运行游戏, 子弹现在可以打到鱼了。

步骤 04 修改 Fish.cs 脚本, 在 SetDamage 函数中添加代码创建爆炸效果, 当鱼被消灭后, 会产生一个爆炸效果。

```
public void SetDamage( int damage )
{
    m_life -= damage;
    if (m_life <= 0)
    {
        // 读取爆炸动画 prefab
        GameObject prefab = Resources.Load<GameObject>("explosion");
        // 创建爆炸效果实例
        GameObject explosion = (GameObject)Instantiate(
            prefab, this.transform.position, this.transform.rotation);
        // 1 秒后自动删除爆炸效果
```

```
        Destroy(explosion, 1.0f);

        OnDeath(this);
        Destroy(this.gameObject);
    }
}
```

运行游戏，效果如图 5-34 所示。

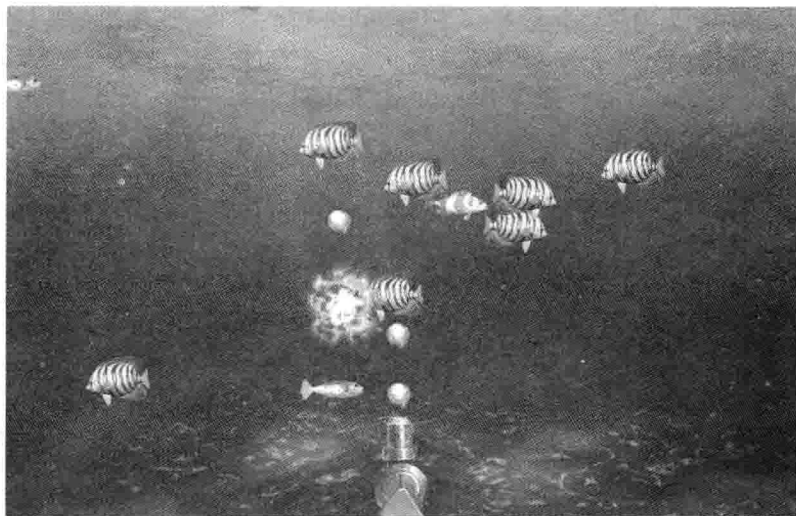


图 5-34 2D 的爆炸效果

这个示例到这里就结束了，如果希望深入做下去，还可以添加不同的大炮、不同的鱼、得分系统等。本节的示例工程文件保存在资源文件目录 `chapter05_fish2d`，在目录 `builds\fish2d` 中是编译好的游戏，可以直接运行。

5.6 2D 材质

5.6.1 修改默认材质

尽管 `Sprite` 多用于 2D 游戏（或者 UI），但它实际上也是个 3D 的简单模型，有自己的材质和 `Shader`。下面的代码获取了 `SpriteRenderer` 的 `material`，并修改了默认的颜色，使 `Sprite` 的颜色变为红色。在很多 2D 游戏中，当敌人受到攻击则会变为红色闪烁，在 Unity 中动画 `Sprite` 材质的颜色可以很容易实现这种效果。

```
SpriteRenderer render = this.GetComponent<SpriteRenderer>();
render.material.color = new Color(1, 0, 0, 1);
```

5.6.2 自定义的黑白效果材质

2D 游戏中一种常见的效果是将图片变为黑白色, 为了实现这种效果, 我们需要略修改默认的 Sprite 材质, 步骤如下。

- 步骤 01** 在 <http://unity3d.com/unity/download/archive/> 或 Unity wiki 站点下载最新的 Unity 官方 shader 原始文件, 当前最新的文件名为 builtin_shaders-4.5.2.zip。
- 步骤 02** 解压 shader 压缩包, 找到 Sprites-Default.shader 这个文件并导入到 Unity 工程中。
- 步骤 03** 双击 Sprites-Default.shader 修改它的代码, 先修改第一行, 将 Shader 的名字改掉, 这里改为 Sprite/Gray, 然后修改 fixed4 frag(v2f IN) 函数, 添加两行代码, 使输出的颜色变成灰度显示, 如下所示:

```
Shader "Sprites/Gray"

fixed4* frag(v2f IN) : SV_Target
{
    fixed4 c = tex2D(_MainTex, IN.texcoord) * IN.color;
    c.rgb *= c.a;
    // 添加的代码
    float gray = dot(c.xyz, float3(0.299, 0.587, 0.114));
    c.xyz = float3(gray, gray, gray);
    return c;
}
```

- 步骤 04** 创建一个新的 Material 指定给 Sprite 并使用自定义的黑白效果 Shader, 即可使图片变为黑白色。我们也可以通过代码动态地改变 shader 实现该效果, 代码如下:

```
SpriteRenderer render = this.GetComponent<SpriteRenderer>();
render.material.shader = Shader.Find("Sprites/Gray");
```

小 结

本章介绍了 Unity 在 2D 游戏上的应用, 包括创建 Sprite, 不同的动画方式, 2D 物理的应用, 最后还通过一个较为完整的实例, 介绍了如何创建一个 2D 捕鱼游戏。尽管我们在这里是开发 2D 游戏, 但因为 Unity 天生是一个 3D 游戏引擎, 实际上我们也可以在 3D 的模式下使用 2D 功能, 创造出 3D、2D 混合的图形效果。

第 6 章

与 Web 服务器的交互

Unity 提供了一个叫 WWW 的功能, 可以使 Unity 游戏访问 Web 服务器, 上传、下载数据等。本章将介绍如何快速创建一个 Web 服务器, 使 Unity 游戏与 Web 服务器交互, 实现如游戏分数排行榜等功能。

6.1 建立服务器

在本章,我们可以把 Unity 游戏想象为一个网站的前台,它可以使用一个叫 WWW 的功能与 Web 服务器进行通信,下载服务器数据到 Unity 游戏中,也可以上传游戏数据到服务器上,并保存在数据库中。WWW 是非常实用的功能,可以轻松实现具有网络功能的游戏,使在线用户获得非实时的交互功能。

创建 Web 服务器要比完成一个供游戏使用的服务器程序容易得多,我们只需要安装一些服务器端软件。接下来,本节将演示如何使用 Apache、PHP 和 MySQL 在 Windows 上建立一个基本的 Web 服务器,这些软件都是免费的,并可以运行在 Linux 上。

6.1.1 安装 Apache

Apache 是著名的 Web 服务器软件,由于其跨平台 and 安全性被广泛使用,下面是安装步骤:

- 步骤 01** 在本书的资源文件目录找到/software/httpd-2.2.22-win32-x86-openssl-0.9.8t.msi 文件,双击按提示安装(也可以访问网址 <http://httpd.apache.org/download.cgi> 免费下载)。
- 步骤 02** 在任务栏单击右键选择 Apache 的图标,选择【Open Apache Monitor】打开 Apache Service Monitor 窗口,如图 6-1 所示。

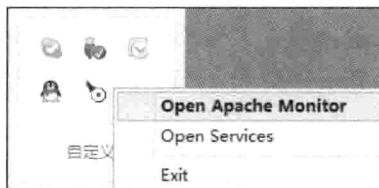


图 6-1 打开 Apache Service Monitor

- 步骤 03** 在 Apache Service Monitor 窗口中, Start、Stop、Restart 分别用来启动、停止、重新启动 Apache 服务,如果 Start 呈灰色显示,表示 Apache 服务已经成功启动,如图 6-2 所示。

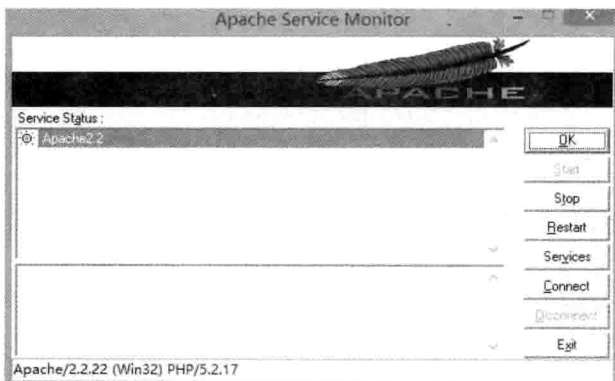


图 6-2 启动 Apache 服务

步骤 04 确定 Apache 服务已经成功启动, 打开浏览器, 输入服务器 IP 地址 (如果是在本机测试, 输出 127.0.0.1 即可), 然后会进入默认的网站页面, 如图 6-3 所示。

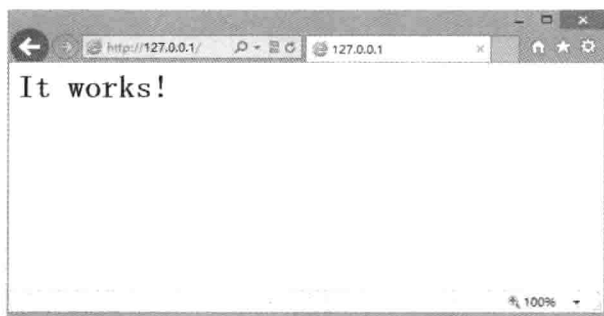


图 6-3 默认的网页页面

在 Windows 上安装 Apache 的过程中, 经常会遇到不能启动 Apache 服务的情况, 一个比较常见的原因是 80 端口被占用, 如图 6-4 所示。

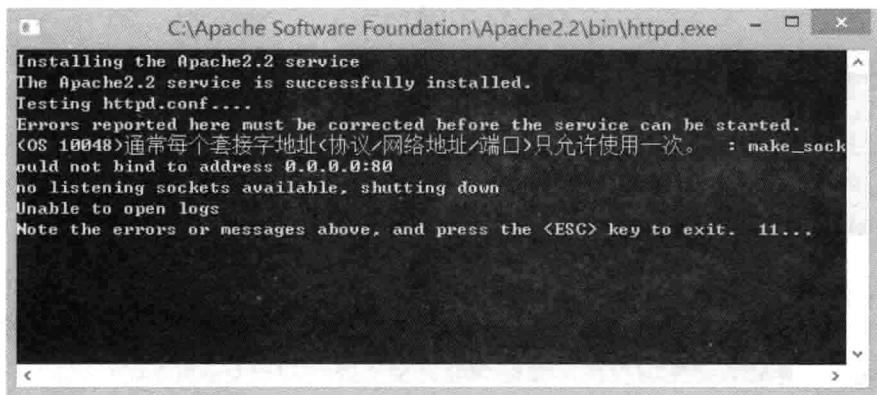


图 6-4 80 端口被占用

这种情况, 只要关掉占用 80 端口的软件即可:

步骤 01 打开命令提示符窗口, 输入 `netstat -a -n -o`, 然后找到使用 80 端口的 PID, 如图 6-5 所示, PID 为 4724 的程序占用了 80 端口。

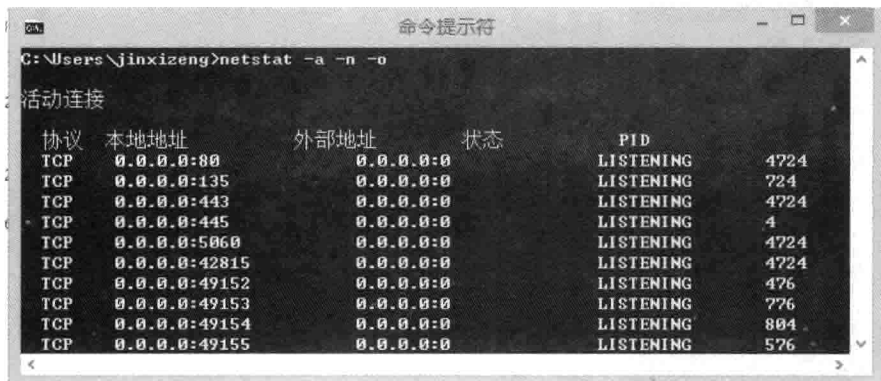


图 6-5 占用 80 端口的程序

步骤 02 按 Ctrl+Alt+Delete 组合键打开任务管理器, 根据 PID 找到占用 80 端口的程序, 将其关掉, 如图 6-6 所示, 然后重启服务即可。

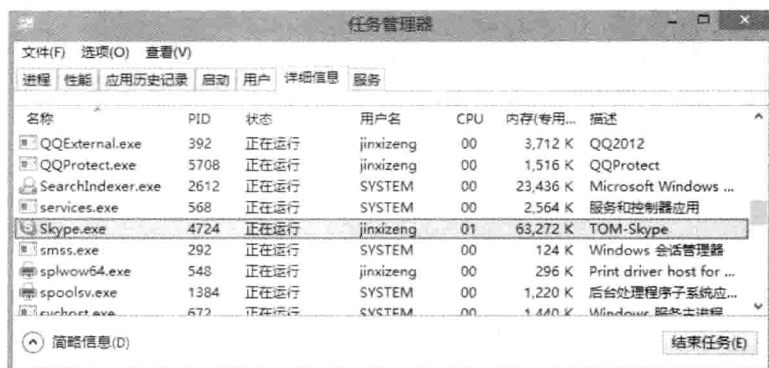


图 6-6 关掉占用 80 窗口的程序

最后, 不要忘记在防火墙中加入 Apache 安装目录 bin 下面的 httpd.exe, 使其通过防火墙。

6.1.2 安装 MySQL

MySQL 是一个关系型数据库管理系统, 由于其体积小、速度快、总体拥有成本低, 很多中小型网站的开发都选择了 MySQL 作为数据库, 使用它搭配 PHP 和 Apache 可组成良好的开发环境。下面是 MySQL 的安装步骤:

步骤 01 在本书的资源文件目录找到/software/mysql-installer-community-5.5.28.3.exe 文件, 双击开始安装(也可以访问网址 <http://www.mysql.com/downloads/mysql/> 免费下载)。

步骤 02 在选择安装类型的时候, 选择 Full 完全安装, 如图 6-7 所示。

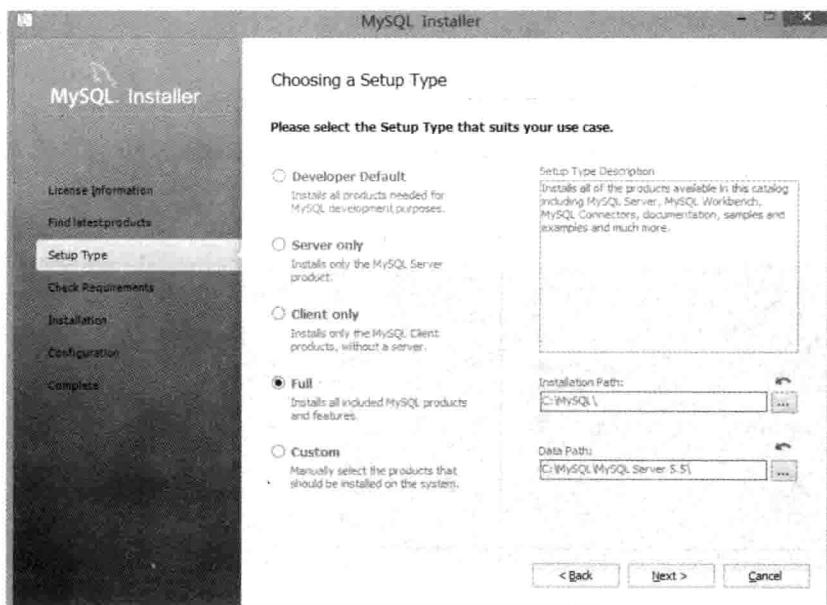


图 6-7 完全安装

步骤 03 如果是安装在本机，可以在选择服务器设置类型时选择 Development Machine，如图 6-8 所示。

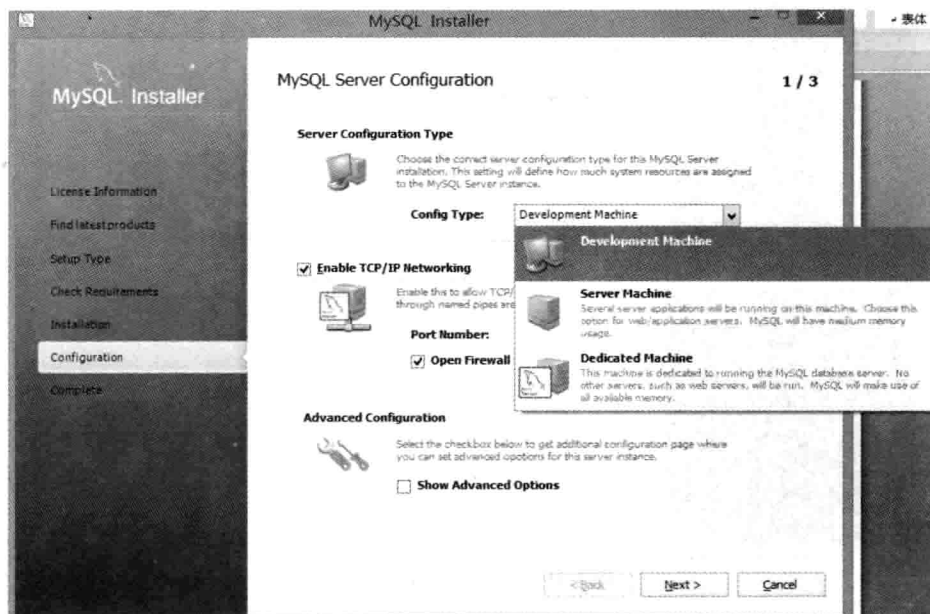


图 6-8 选择服务器类型

步骤 04 设置管理员密码，这个密码后面会经常用到，一定要记住，如图 6-9 所示。

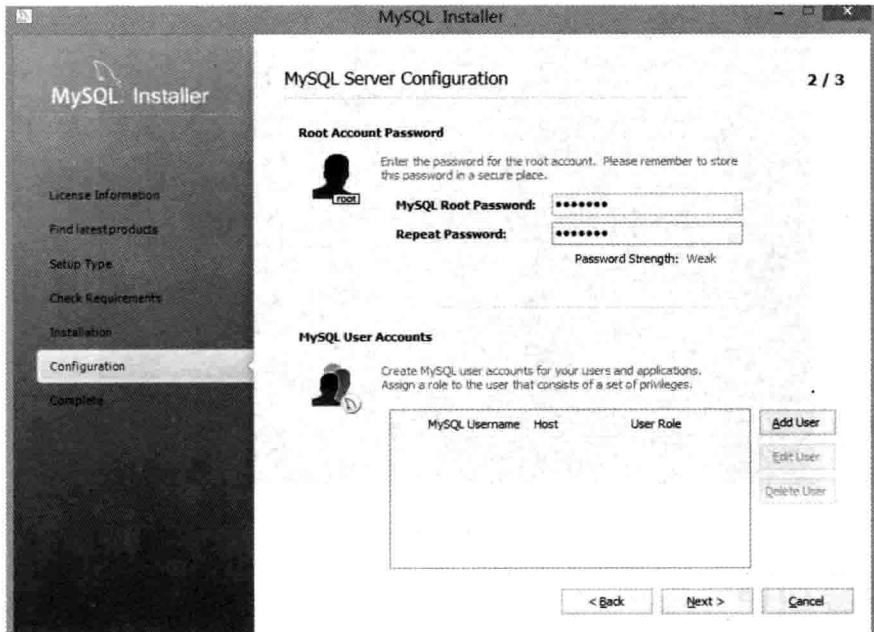


图 6-9 设置管理员密码

步骤 05 选中 Start the MySQL Server at System Startup 选项，启动 MySQL 服务，如图 6-10 所示。

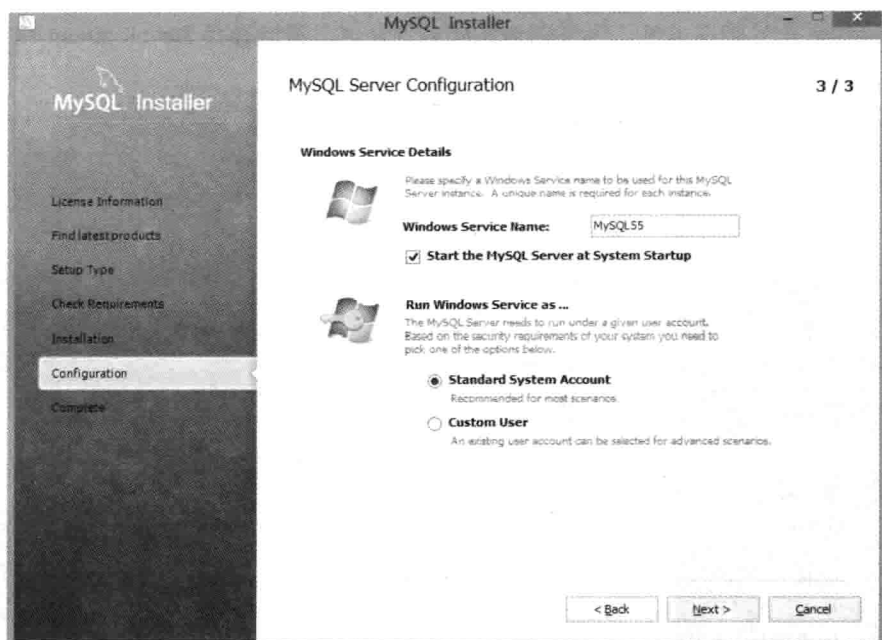


图 6-10 启动 MySQL 服务

步骤 06 安装完成后，可以先简单试一下数据库功能。在开始菜单选择 MySQL 5.5 Command Line Client 启动 MySQL 命令行窗口，输入管理员密码，进入 MySQL 命令行模式，输入 SQL 命令“show databases;”显示出默认的数据库，如图 6-11 所示。

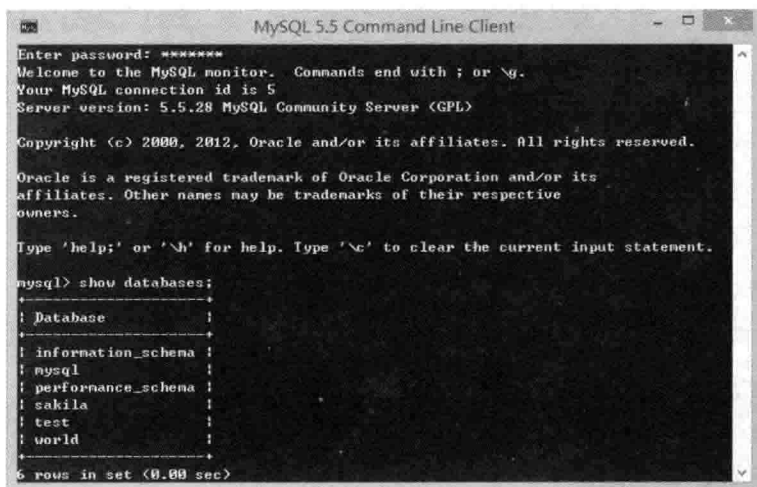


图 6-11 显示默认数据库

6.1.3 安装 PHP

PHP 是英文超文本预处理语言 Hypertext Preprocessor 的缩写，它是一种 HTML 内嵌式语言，运行在服务器端，语法与 C 语言类似，被广泛地运用。安装 PHP 的步骤如下：

步骤 01 在本书的资源文件目录找到/software/php-5.2.17-Win32-VC6-x86.zip 文件（也可以

访问网址 <http://www.php.net/downloads.php> 免费下载），将其解压至 C 盘根目录，并将文件夹名称改为 php，如图 6-12 所示。

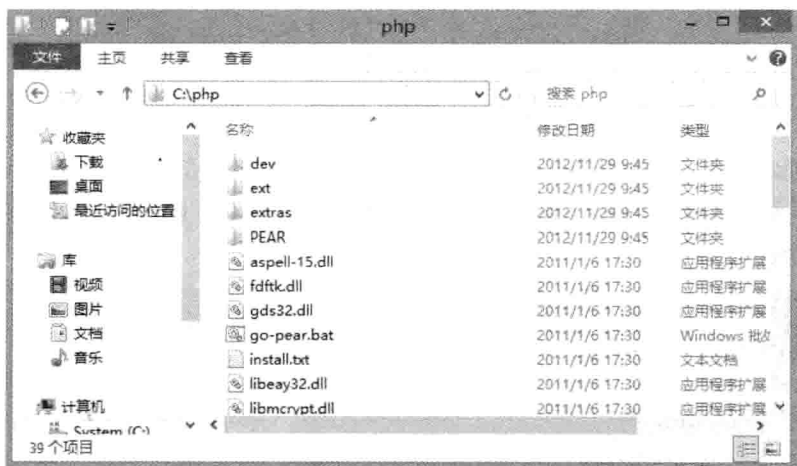


图 6-12 php 文件目录

步骤 02 在 php 文件夹内找到 php.ini-recommended 文件，将其备份，然后更名为 php.ini。

步骤 03 使用文件编辑器打开 php.ini 文件，找到 `extension_dir = "."`，在这里设置 PHP 扩展路径，如下所示：

```
; Directory in which the loadable extensions (modules) reside.
extension_dir = "c:/php/ext"
```

步骤 04 在 php.ini 文件中继续查找，找到 `;extension=php_mysql.dll` 和 `;extension=php_mysqli.dll`，分别将前面的 “;” 号去掉启用 MySQL 扩展，然后保存文件，如下所示：

```
extension=php_mysql.dll
extension=php_mysqli.dll
;extension=php_oci8.dll
```

步骤 05 在 Apache 的安装目录/Apache2.2/conf 内找到 httpd.conf 文件，使用文本编辑器将其打开。

步骤 06 在 httpd.conf 文件末端添加如下脚本，需要注意的是，脚本内容需要对应当前安装的 PHP 或 Apache 版本：

```
LoadModule php5_module "c:/php/php5apache2_2.dll"
AddHandler application/x-httpd-php .php

# configure the path to php.ini
PHPIniDir "c:/php/"
<FilesMatch \.php$>
    SetHandler application/x-httpd-php
</FilesMatch>
```

步骤 07 网站根目录默认是存放在 Apache 安装目录内的 htdocs 内，在 httpd.conf 文件内查

找到 DocumentRoot, 可以自定义网站的默认位置, 如下所示:

```
DocumentRoot "D:/web"
```

步骤 08 在 httpd.conf 文件内继续查找 Directory, 设置自定义的网站, 如下所示:

```
<Directory "D:/web">
```

步骤 09 网站的默认启动页是 index.html, 在 httpd.conf 文件内查找“<IfModule dir_module>”, 添加 DirectoryIndex index.php, 使 index.php 文件也可以作为网站的启动页面, 如下所示:

```
# DirectoryIndex: sets the file that Apache will serve if a directory
# is requested.
#
<IfModule dir_module>
    DirectoryIndex index.php
    DirectoryIndex index.html
</IfModule>
```

步骤 10 在 PHP 安装目录找到 libmysql.dll 文件, 将其复制到 C:\Windows\System32 下。注意, 如果是 64 位 Windows 操作系统需要将其复制到 C:\Windows\SysWOW64。

步骤 11 重新启动 Apache 服务。

6.1.4 显示 PHP 信息

我们可以使用任何文本编辑器编写 PHP 脚本, 各种专门的 PHP 代码编辑器也非常多。在本书中, 我们将使用免费的 Eclipse for PHP Developers 编写 PHP 脚本, 有兴趣的读者也可以查找相关资料, 选择其他工具。

接下来, 我们将创建一个简单的 PHP 工程, 在一个网页页面上显示出当前安装的 PHP 相关信息。

步骤 01 在本书的资源文件目录找到/software/eclipse-php-luna-M5-win32-x86_64.zip 文件 (也可以访问网址 <http://www.eclipse.org/downloads/index-developer.php> 免费下载), 将其解压, 双击“eclipse.exe”启动程序。

步骤 02 启动 eclipse 的时候要求选择 PHP 工程的目录, 将其设置在网站的根目录。

步骤 03 在 eclipse 的菜单栏选择【File】→【New】→【PHP Project】创建一个新的 PHP 工程。

步骤 04 在工作区单击右键选择【New】→【PHP File】打开新窗口, 输入文件名, 创建 test.php 文件。

步骤 05 在 test.php 中输入代码 phpinfo(); 这个函数的作用是在网页页面上显示出当前安装的 PHP 设置信息:

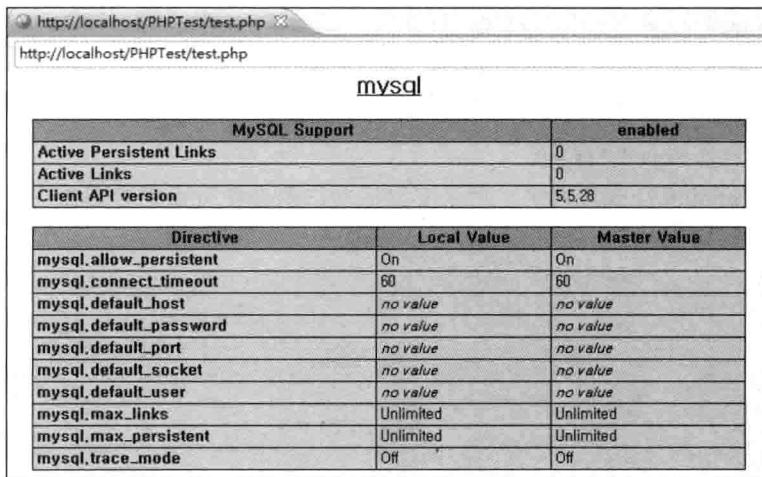
```
<?php
```

```
phpinfo();
```

```
?>
```

步骤 06 在工作区选择 test.php 文件，在菜单栏选择【Run】→【Run As】→【PHP Web Page】显示出网页页面。

现在，在网页页面上将会显示出相应的 PHP 设置信息，如图 6-13 所示。在这个页面上还会显示出 MySQL 的相应信息，如果找不到，说明 PHP 或 MySQL 的设置出现了问题，这时将不能正常使用 MySQL 数据库。



MySQL Support		enabled
Active Persistent Links		0
Active Links		0
Client API version		5.5.28

Directive	Local Value	Master Value
mysql.allow_persistent	On	On
mysql.connect_timeout	60	60
mysql.default_host	no value	no value
mysql.default_password	no value	no value
mysql.default_port	no value	no value
mysql.default_socket	no value	no value
mysql.default_user	no value	no value
mysql.max_links	Unlimited	Unlimited
mysql.max_persistent	Unlimited	Unlimited
mysql.trace_mode	Off	Off

图 6-13 PHP 设置信息

本节的示例文件保存在资源文件目录/chapter08_PHPTest。

6.1.5 调试 PHP 代码

PHP 是弱类型语言，这也意味着调试将变得困难，因此好的调试方法和工具变得非常重要。Zend Studio Web Debugger 是一个 PHP 调试插件，我们将通过它来调试 PHP 代码。

- 步骤 01** 在本书的资源文件目录找到/software/ZendDebugger-20110410-cygwin_nt-i386.zip 文件（也可以访问网址 <http://www.zend.com/en/products/studio/downloads> 注册后下载），将其解压。
- 步骤 02** 在解压后的目录下有一个 readme.txt 文件是安装说明。我们需要根据当前 PHP 版本选择相应的文件。将 dummy.php 复制到网站根目录。
- 步骤 03** 选择 5_2_x_comp 文件夹内的 dll，将 ZendDebugger.dll 复制到 PHP 安装目录。另一个 5_2_x_nts_comp 文件夹是针对 non-thread safe 版本的 PHP。
- 步骤 04** 在 PHP 安装目录打开 php.ini 文件并添加：

```
[Zend]
zend_extension_ts          = "c:/php/ZendDebugger.dll"
zend_debugger.allow_hosts  = localhost,127.0.0.1
```

```
zend_debugger.expose_remotely = always
```

zend_extension_ts 指向 ZendDebugger.dll 文件的存储位置。

zend_debugger.allow_hosts 指向网站的 IP。



提示

如果是使用 non-thread safe 版本的 PHP, 需要将 zend_extension_ts 改为 zend_extension。

在开发过程中, 为了可以看到相关的错误提示信息, 还可以继续修改 php.ini:

```
- display_errors = On           [Security]

- error_reporting = E_ALL       [Code Cleanliness, Security(?)]
```

步骤 05 重新启动 Apache 服务。

步骤 06 运行 phpinfo(); 查看网页, 如果安装成功, 则会在页面中看到相应的 Debug 信息, 如图 6-14 所示。

PHP API	20041225
PHP Extension	20060613
Zend Extension	220060519
Debug Build	no
Thread Safety	enabled
Zend Memory Manager	enabled
IPv6 Support	enabled
Registered PHP Streams	php, file, data, http, ftp, compress, zlib
Registered Stream Socket Transports	tcp, udp
Registered Stream Filters	convert.iconv.*, string.rot13, string.toupper, string.tolower, string.strip_tags, convert.*, consumed, zlib.*

This program makes use of the Zend Scripting Language Engine:
 Zend Engine v2.2.0. Copyright (c) 1998-2010 Zend Technologies
 with Zend Debugger v5.2.15. Copyright (c) 1999-2008, by Zend Technologies

Powered By

图 6-14 显示 Debug 信息

在下面的 PHP 代码中, 存在很多错误, 在正确安装好 PHP 调试器后, 在菜单栏选择【Run】→【Debug】调试这段代码, 在 Console 窗口则会提示错误的位置和原因, 如图 6-15 所示。

```
<?php

$n=$n1;

add( $n, 1 );

function add( $a,$b )
{
    return $a+b;
```

}

?>

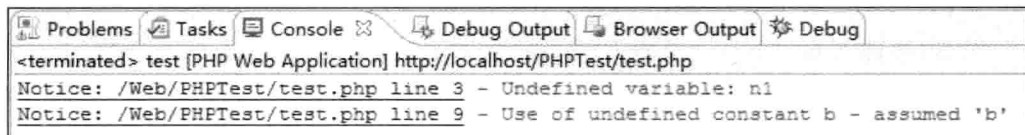


图 6-15 错误信息

6.2 WWW 基本应用

本节将演示一个简单的示例，从 Unity 程序向 Web 服务器发送数据，Web 服务器收到数据后向 Unity 程序再返回数据。

6.2.1 HTTP 协议

Unity 的 WWW 是基于 HTTP 协议的网络传输功能，HTTP (hypertext transport protocol) 协议即超文本协议，它规定了万维网数据通信的规则，它是客户端/服务器模式，客户端和服务端都必须支持 HTTP。HTTP 协议的一个重要特点是每次连接只处理一个请求，当服务器处理完客户端的请求即断开连接，节省传输时间。

使用 HTTP 协议传输数据有多种方式，Unity 的 WWW 主要支持其中的 GET 和 POST 方式。GET 方式会将请求附加在 URL 后，POST 方式则是通过 FORM (表单) 的形式提交。GET 方式最多只能传输 1024 个字节，POST 方式理论上则没有限制。从安全角度来看 POST 比 GET 方式安全性更高，所以在实际使用中可以更多选择 POST 方式。

下面我们先创建一个简单的 UI 界面，提供两个按钮，分别用于使用 GET 和 POST 方式向服务器提交数据。

步骤 01 新建 Unity 工程，创建脚本 WebManager.cs，将其指定给场景中的任意游戏体。

步骤 02 在 WebManager.cs 中添加一个 m_info 属性和 OnGUI 函数显示 UI:

```
string m_info = "Nothing";

void OnGUI() {

    GUI.BeginGroup(new Rect(Screen.width * 0.5f - 100, Screen.height * 0.5f - 100, 500,
200), "");

    GUI.Label(new Rect(10, 10, 400, 30), m_info);
    // 创建按钮
    if (GUI.Button(new Rect(10, 50, 150, 30), "Get Data")) {
    }
}
```

```
// 创建按钮
if (GUI.Button(new Rect(10, 100, 150, 30), "Post Data")){
}

GUI.EndGroup();
}
```

运行程序,在窗口中会出现两个按钮并显示“Nothing”,如图 6-16 所示。我们将使用 Get Data 和 Post Data 按钮分别通过 GET 和 POST 方式向 Web 服务器发送数据,然后服务器返回数据,传递给 m_info 属性显示在屏幕上。



图 6-16 两个按钮

6.2.2 GET 请求

接下来,我们使用 GET 方式向服务器提交数据,包括一个用户名和一个密码,服务器收到后会返回一个字符串。

步骤 01 在 WebManager.cs 脚本中添加一个 IGetData()函数,注意函数的返回类型是迭代器 IEnumerator,并可以通过协程调用。

```
IEnumerator
IEnumerator IGetData()
{
    // 使用 Get 方式访问 Http 地址
    WWW www = new WWW("http://127.0.0.1/test.php?username=get&password=12345");
    // 等待服务器的响应
    yield return www;
    // 如果出现错误
    if (www.error != null)
    {
        m_info = www.error;
        yield return null;
    }
    // 获得服务器的响应文本
    m_info = www.text;
}
```

在这个函数中,我们首先创建了一个 WWW 实例,使其向指定的 IP 地址发送 GET 请求,跟随在 IP 地址后面的? 用于附加数据,这里我们发送了两个 GET 数据,一个是 username,另

一个是 password，它们的值分别是 get 和 12345。

WWW 实例将在后台运行，yield return www 会等待 Web 服务器的反应。

如果 WWW 实例的 error 属性不为空，Web 服务器返回的数据则会保存在 WWW 实例的 text 属性当中。

步骤 02 在 OnGUI 函数中添加代码执行 IGetData 函数：

```
if (GUI.Button(new Rect(10, 50, 150, 30), "Get Data"))
{
    StartCoroutine(IGetData());
}
```

步骤 03 接下来，我们要创建一个 PHP 脚本响应 WWW 的 GET 请求。新建 PHP 工程，在 Web 服务器根目录创建 test.php，并添加代码：

```
<?php

if ( isset($_GET['username']) && isset($_GET['password']) )
{
    echo 'username is ' . $_GET['username'] . ' and password is ' . $_GET['password'];
}
else
    echo "error!";

?>
```

这是一段 PHP 代码，isset 函数用来判断是否收到相应的 GET 请求，如果收到了，则使用 echo 函数输出结果，并将其返回到 Unity 程序中。



提示

在 PHP 中，连接两个字符串是使用“.”而不是“+”。

在 Unity 中运行程序，单击 Get Data 按钮，会收到服务器返回的数据，结果如图 6-17 所示。



图 6-17 收到服务器返回的数据

6.2.3 POST 请求

使用 POST 提交数据的方式与 GET 类似，但我们会把字符串转为 byte 数组。

步骤 01 在 WebManager.cs 脚本中添加一个 IPostData() 函数:

```
IEnumerator IPostData()
{
    System.Collections.Hashtable headers = new System.Collections.Hashtable();
    headers.Add("Content-Type", "application/x-www-form-urlencoded");
    // 将要发送的 Post 文本
    string data = "username=post&password=6789";
    // 将文本转为 byte 数组
    byte[] bs = System.Text.UTF8Encoding.UTF8.GetBytes(data);
    // 向 Http 服务器提交 Post 数据
    WWW www = new WWW("http://127.0.0.1/test.php", bs, headers);
    // 等待服务器的响应
    yield return www;

    if (www.error != null)
    {
        m_info = www.error;
        yield return null;
    }

    m_info = www.text;
}
```

与 GET 不同的是,在保存数据的字符串中,最前面没有? 符号,但仍使用&符号连接数据,最后我们将字符串转为一个 byte 数组。headers 是一个 Hashtable,它由键、值对应,这里我们用它来保存 HTTP 报头。

步骤 02 在 OnGUI 函数中添加代码执行 IPostData 函数:

```
if (GUI.Button(new Rect(10, 100, 150, 30), "Post Data"))
{
    StartCoroutine(IPostData());
}
```

步骤 03 修改 PHP 脚本,添加 POST 请求的响应:

```
<?php

if ( isset($_GET['username']) && isset($_GET['password']) )
    echo 'username is ' . $_GET['username'] . ' and password is ' . $_GET['password'];
else if ( isset($_POST['username']) && isset($_POST['password']) )
    echo 'username is ' . $_POST['username'] . ' and password is ' . $_POST['password'];
else
    echo 'error';
```

??>

在 Unity 中运行程序, 单击 Post Data 按钮, 会收到服务器返回的数据, 结果如图 6-18 所示。



图 6-18 收到服务器返回的数据

6.2.4 上传下载图片

Unity 的 WWW 不但能上传、下载文本形式的数据, 还可以上传、下载图片, 不过在传输过程中, 图片的信息仍需要转为文本格式。

步骤 01 在 WebManager.cs 脚本中添加两个 Texture2D 属性用于保存图片信息:

```
public Texture2D m_uploadImage;

protected Texture2D m_downloadTexture;
```

步骤 02 在当前工程中导入任意一张图片与 m_uploadImage 属性相关联, 我们将把这张图片上传到服务器, 再将其下载回来赋予 m_downloadTexture。

步骤 03 在 WebManager.cs 脚本中添加一个 IRequestPNG () 函数:

```
IEnumerator IRequestPNG()
{
    // 将图片的字节读入 byte 数组
    byte[] bs = m_uploadImage.EncodeToPNG();
    // 创建 WWWForm
    WWWForm form = new WWWForm();
    // 添加二进制的 Post 数据
    form.AddBinaryData("picture", bs, "screenshot", "image/png");
    // 向服务器提交二进制的 Post 数据
    WWW www = new WWW("http://127.0.0.1/Test.php", form);
    // 等待服务器响应
    yield return www;

    if (www.error != null)
    {
        m_info = www.error;
        yield return null;
    }
    // 收到服务器返回的图片数据
```

```
m_downloadTexture = www.texture;
}
```

在这段代码中, 我们使用 `EncodeToPNG` 函数将图片转出为 `byte` 数组, 使用 `WWWForm` 的方式上传到 Web 服务器上, 与之前不同的是, 这一次是上传的 `PNG` 格式的图片。我们将在服务器端直接返回图片的文本信息, 将其指定给 `m_downloadTexture`。

步骤 04 在 `OnGUI` 函数中添加如下代码, 用于显示下载的图片并提交图片:

```
if ( m_downloadTexture!=null )
    GUI.DrawTexture(new Rect(0, 0, m_downloadTexture.width,
                                m_downloadTexture.height),
                    m_downloadTexture);

if (GUI.Button(new Rect(10, 150, 150, 30), "Request Image"))
{
    StartCoroutine(IRequestPNG());
}
```

步骤 05 修改 PHP 文件:

```
<?php
if ( isset($_GET['username']) && isset($_GET['password']) )
    echo 'username is ' . $_GET['username'] . ' and password is ' . $_GET['password'];
else if ( isset($_POST['username']) && isset($_POST['password']) )
    echo 'username is ' . $_POST['username'] . ' and password is ' . $_POST['password'];
else if ( isset($_FILES['picture']))
    echo file_get_contents($_FILES['picture']['tmp_name']);
else
    echo 'error';
?>
```

因为在 Unity 中上传的是一张图片, 所以我们使用 `$_FILES` 来获得图片, 这是一个 PHP 数组, 其中 `'tmp_name'` 是保存临时文件的位置, 我们访问数组的这个位置即可获得图片。最后使用 `file_get_contents` 将文件转为文本发回给 Unity 程序。

在 Unity 中运行程序, 单击 `Request Image` 按钮, 会收到服务器返回的图片并显示在屏幕上, 结果如图 6-19 所示。



图 6-19 收到服务器返回的图片

6.2.5 下载声音文件

使用 WWW 功能，除了能够下载图片，还能下载声音，方法与下载图片类似，下面是一个简单的示例：

步骤 01 在网站的根目录放置一个声音文件，比如文件的名称叫 music.wav。

步骤 02 在 WebManager.cs 脚本中添加一个 m_downloadClip 属性和 DownloadSound() 函数：

```
protected AudioClip m_downloadClip;

IEnumerator DownloadSound()
{
    // 下载一个声音文件
    WWW www = new WWW("http://127.0.0.1/music.wav");

    yield return www;

    if (www.error != null)
    {
        m_info = www.error;
        yield return null;
    }
    // 获得服务器返回的声音文件
    m_downloadClip = www.GetAudioClip(false);
    // 播放声音
    audio.PlayOneShot(m_downloadClip);
}
```

步骤 03 在 Start 函数中执行 DownloadSound() 函数：

```
void Start () {
    StartCoroutine(DownloadSound());
}
```

步骤 04 为当前游戏体添加一个 Audio Source 组件，运行程序，下载完成声音后，即会听到播放的声音。

本节的示例文件保存在资源文件目录 chapter06_WebTest，其中还包括一个 test.php 和用于测试的 music.wav 文件，需要被放置到网站的根目录下。

6.3 分数排行榜

本节将综合运用本章所涉及的内容完成一个分数排行榜。我们可以在 Unity 中向服务器发送

用户名和得分并存入数据库,也可以将数据库中的得分按排行榜的形式下载到 Unity 中。在向服务器上传数据的时候,我们会使用标准的 Post 方式,服务器收到数据后,会将结果按照 JSON 的格式发回给 Unity,而我们将在 Unity 中使用一个叫 MiniJSON 的插件解析 JSON 格式的文本。

6.3.1 创建数据库

首先,我们要在 MySQL 数据库中建立一个简单的数据库,用来保存用户名和得分,使用 MySQL 提供的 MySQL Workbench 工具只需几个步骤即可完成这个工作。

- 步骤 01** 确定完整安装了 MySQL,启动 MySQL Workbench,它是一个图形化界面的数据库工具软件。
- 步骤 02** 在菜单栏选择【File】→【New Model】创建一个新的数据库模型。默认的数据库模型叫 mydb,双击 mydb 名称,在下面的 Name 中输入数据库的名称,这里命名为 myscoresdb,如图 6-20 所示。

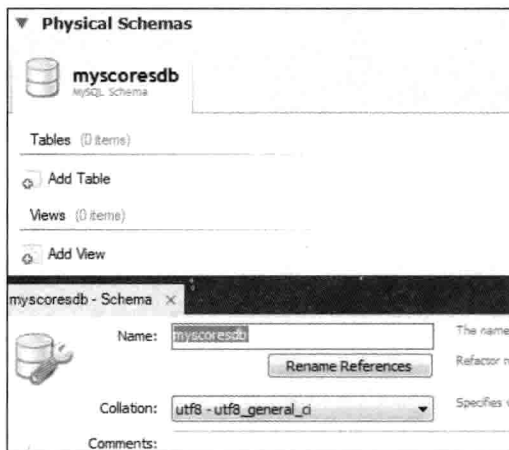


图 6-20 创建数据库

- 步骤 03** 选择【Add Table】创建一个数据表并命名为 Hiscores,添加 3 条数据,分别是 id、name 和 score,其中 id 是主键, name 和 score 表示用户名和分数,如图 6-21 所示。

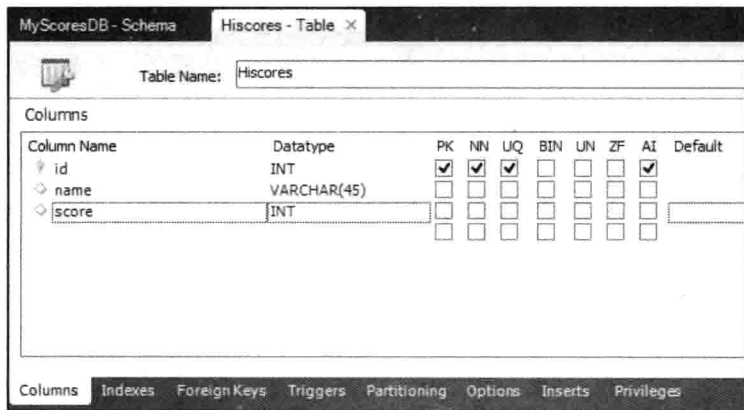


图 6-21 创建高分数据表

步骤 04 在菜单栏选择【Database】→【Forward Engineer】打开新窗口。

步骤 05 输入 MySQL 数据库的 IP 地址和用户账号，如果是在本地，则保持不变即可，最后需要输入 MySQL 管理员密码，即可将前面创建的数据库模型导入真正的 MySQL 数据库中。

步骤 06 为了验证是否创建正确，在菜单栏选择【Database】→【Connect to Database】连接到数据库，然后输入 SQL 语句，按 Ctrl+Enter 组合键执行语句，会看到数据表的描述，如图 6-22 所示。

```
use myscoresdb;
describe hiscores;
```

	Field	Type	Null	Key	Default	Extra
▶	id	int(11)	NO	PRI	NULL	auto_increment
	name	varchar(45)	YES		NULL	
	score	int(11)	YES		NULL	

图 6-22 显示数据表内容

6.3.2 创建 PHP 脚本

我们需要创建两个 PHP 脚本：一个用来上传用户名和分数；另一个用来下载分数并将其排序后发送给 Unity。

步骤 01 创建 UploadScore.php 脚本，在这里我们首先要读入来自 Unity 的 username 和 score 数据，然后打开数据库，最后使用 SQL 语句将数据插入到数据库中，代码如下：

```
<?php

// 读入用户名和分数
$UserID=$_POST["username"];           // user name
$hiscore=$_POST["score"];             // user score

// 连接数据库
$myData=mysqli_connect( "localhost" ,"root" ,"123456" );
if ( mysqli_connect_errno() )
{
    echo mysqli_connect_error();
    return;
}

// 校验用户名是否合法(防止 SQL 注入)
$UserID=mysqli_real_escape_string($myData,$UserID);

// 选择数据库
mysqli_query($myData,"set names utf8") ;
```

```

mysqli_select_db( $myData , "myscoresdb" );

// 插入新数据
$sql="insert into hiscores value( NULL, '$UserID','$hiscore')";
mysqli_query($myData,$sql);

//关闭数据库
mysqli_close($myData);

echo 'upload '.$UserID.':'.$hiscore;

?>

```

步骤 02 创建 DownloadScores.php 脚本，我们将从数据库中查询分数最高的 20 个记录，然后在循环语句中将用户名和得分发送给 Unity，代码如下：

```

<?php

// 连接数据库
$myData=mysqli_connect( "localhost" ,"root" ,"123456" );
if ( mysqli_connect_errno() )
{
    echo mysqli_connect_error();
    return;
}

// 选择数据库
mysqli_query($myData,"set names utf8" );
mysqli_select_db( $myData , "myscoresdb" );

// 查询
$sql = "SELECT * FROM hiscores ORDER by score DESC LIMIT 20 ";

$result = mysqli_query($myData,$sql) or die("<br>SQL error!<br/>");
$num_results = mysqli_num_rows($result);

// 准备发送数据到 Unity
$arr =array();
// 将查询结果写入到 JSON 格式的数组中
for($i = 0; $i < $num_results; $i++)
{
    $row = mysqli_fetch_array($result ,MYSQLI_ASSOC);

```

```

        $id=$row['id'];
        $name=$row['username'];
        $score=$row['score'];
        $arr[$id]['id']=$id;
        $arr[$id]['username']=$name;
        $arr[$id]['score']=$score;
    }

    mysqli_free_result($result);

    // 关闭数据库
    mysqli_close($myData);

    // 发送 JSON 格式的数据
    echo json_encode($arr);
?>

```

6.3.3 上传下载分数

在 Unity 中，我们将通过简单的 UI 实现两个功能：一个是将用户名和得分上传；另一个是下载得分排名前 20 的用户名和得分。实际上，我们是通过 Unity 的 WWW 功能调用相应的 PHP 脚本更新数据库的内容，并反馈到 Unity 中。

步骤 01 在资源文件目录打开 Chapter06_HighScore_Start 工程，在这个 Unity 工程中，已经包括了一些简单的 UI 代码，包括上传、下载的按钮和一个排行榜 UI，如图 6-23 所示，还包括一个 MiniJSON.cs 文件，它专门用来解析 JSON 格式的数据。

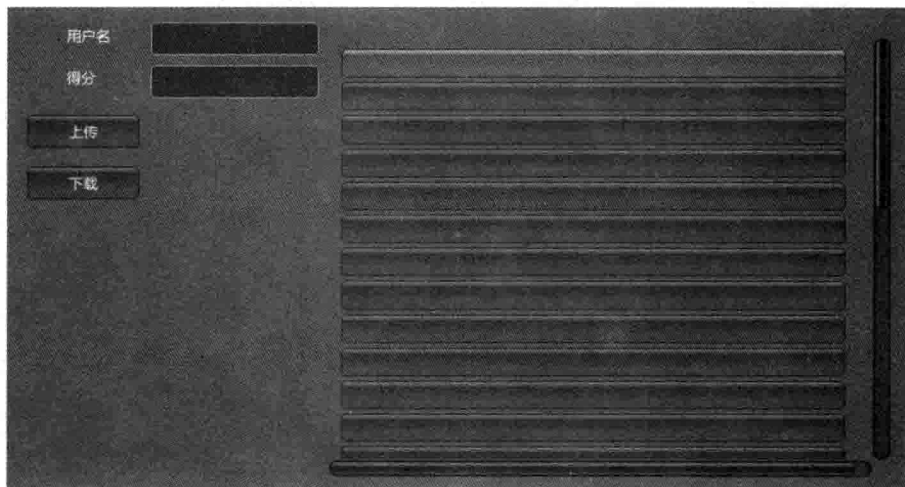


图 6-23 上传下载按钮和一个分数排行榜的 UI

步骤 02 打开 HiScoreApp.cs 脚本，添加 UploadScore 函数上传分数，这里使用的是 Post 方式：

```
IEnumerator UploadScore( string name, string score)
{
    WWWForm form = new WWWForm();
    form.AddField("username", name); // 传入用户名
    form.AddField("score", score);   // 传入分数

    WWW www = new WWW("127.0.0.1/UploadScore.php", form );

    yield return www;

    if (www.error != null)
    {
        Debug.LogError(www.error);
    }
}
```

步骤 03 在上传分数按钮中执行 UploadScore 函数:

```
// 上传分数
if (GUI.Button(m_uploadBut, "上传"))
{
    StartCoroutine(UploadScore(m_name, m_score));
    m_name = string.Empty; //清空 UI 中的输入
    m_score = string.Empty; //清空 UI 中的输入
}
```

步骤 04 定义一个 UserData 类, 用来保存服务器返回的数据, 添加 DownloadScores 函数下载服务器返回的数据, 使用 MiniJSON 提供的功能将其解析。

```
// 定义一个类, 它的字段名称一定要与服务器返回的 JSON 格式的数据的键名一致
public class UserData
{
    public int id;
    public string username;
    public int score;
}

IEnumerator DownloadScores(string name, string score)
{
    WWW www = new WWW("127.0.0.1/DownloadScores.php");

    yield return www;

    if (www.error != null)
    {
```



```

        Debug.LogError(www.error);
    }
    else
    {
        // 将服务器返回的数据使用 MiniJSON 解析为一个键值对应的 Dictionary
        var dict = MiniJSON.Json.Deserialize(www.text) as
                                                    Dictionary<string, object>;

        // 将 Dictionary 中所有的值取出, 并解析为一个 UserData 对象
        int index = 0;
        foreach (object v in dict.Values)
        {
            UserData user = new UserData(); ;
            MiniJSON.Json.ToObject(user, v);

            m_hiscores[index] = user.username + ":" + user.score;
            index++;
        }
    }
}

```

步骤 05 在上传分数按钮中执行 DownloadScores 函数:

```

// 下载分数
if (GUI.Button(m_downLoadBut, "下载"))
{
    StartCoroutine(DownloadScores(m_name, m_score));
}

```

运行程序, 输入用户和分数, 然后单击【上传】按钮即可将用户名和分数上传到数据库中, 单击【下载】按钮, 即可将分数排名前 20 的记录下载下来并显示在排行榜中, 如图 6-24 所示。

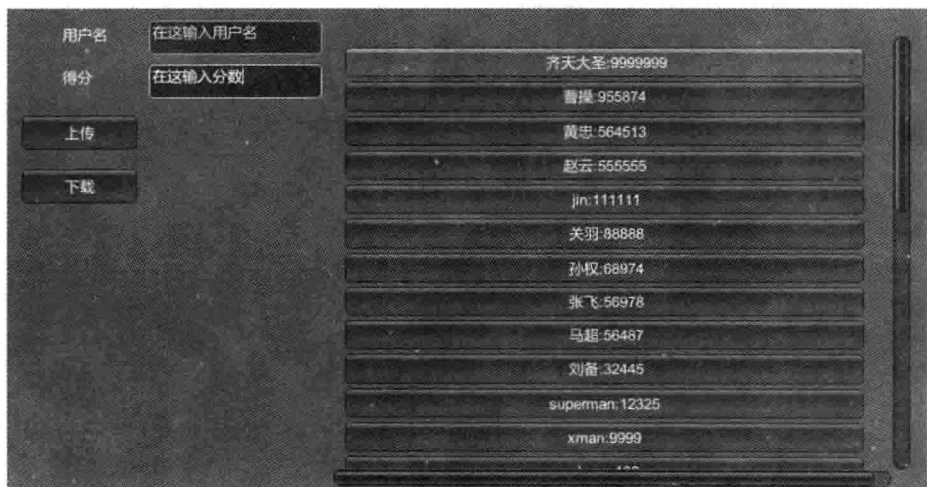


图 6-24 分数排行榜

本节的最终示例文件保存在资源文件目录 chapter06_HighScore, 其中还包括 PHP 脚本和 MySQL 数据库模型文件。

注意, 本章中的 WWW 示例基本都使用了 yield 方式, 如果不希望使用这种方式, 可以通过查询 WWW 对象的 isDone 的值, 获知 WWW 请求是否结束。

6.4 MD5 验证

数据经网络传输后会变得非常不安全, 最简单有效的解决方案是给数据加一个密钥, 使用 MD5 算法算出校验码, 服务器收到数据和校验码后再进行比较校验码是否正确, 以此来判断数据是否被修改过。PHP 生成的 MD5 校验码默认为 32 位的字符串, 而 C#默认是 16 位的字节数组, 需要略加修改, 转为 32 个字节的字符串, 代码如下:

```
public static string Md5Sum(string strToEncrypt)
{
    // 将需要加密的字符串转为 byte 数组
    byte[] bs = UTF8Encoding.UTF8.GetBytes(strToEncrypt);
    // 创建 md5 对象
    System.Security.Cryptography.MD5 md5;
    md5=System.Security.Cryptography.MD5CryptoServiceProvider.Create();
    // 生成 16 位的二进制校验码
    byte[] hashBytes = md5.ComputeHash(bs);

    // 转为 32 位字符串
    string hashString = "";
    for (int i = 0; i < hashBytes.Length; i++)
    {
        hashString += System.Convert.ToString(hashBytes[i], 16).PadLeft(2, '0');
    }

    return hashString.PadLeft(32, '0');
}
```

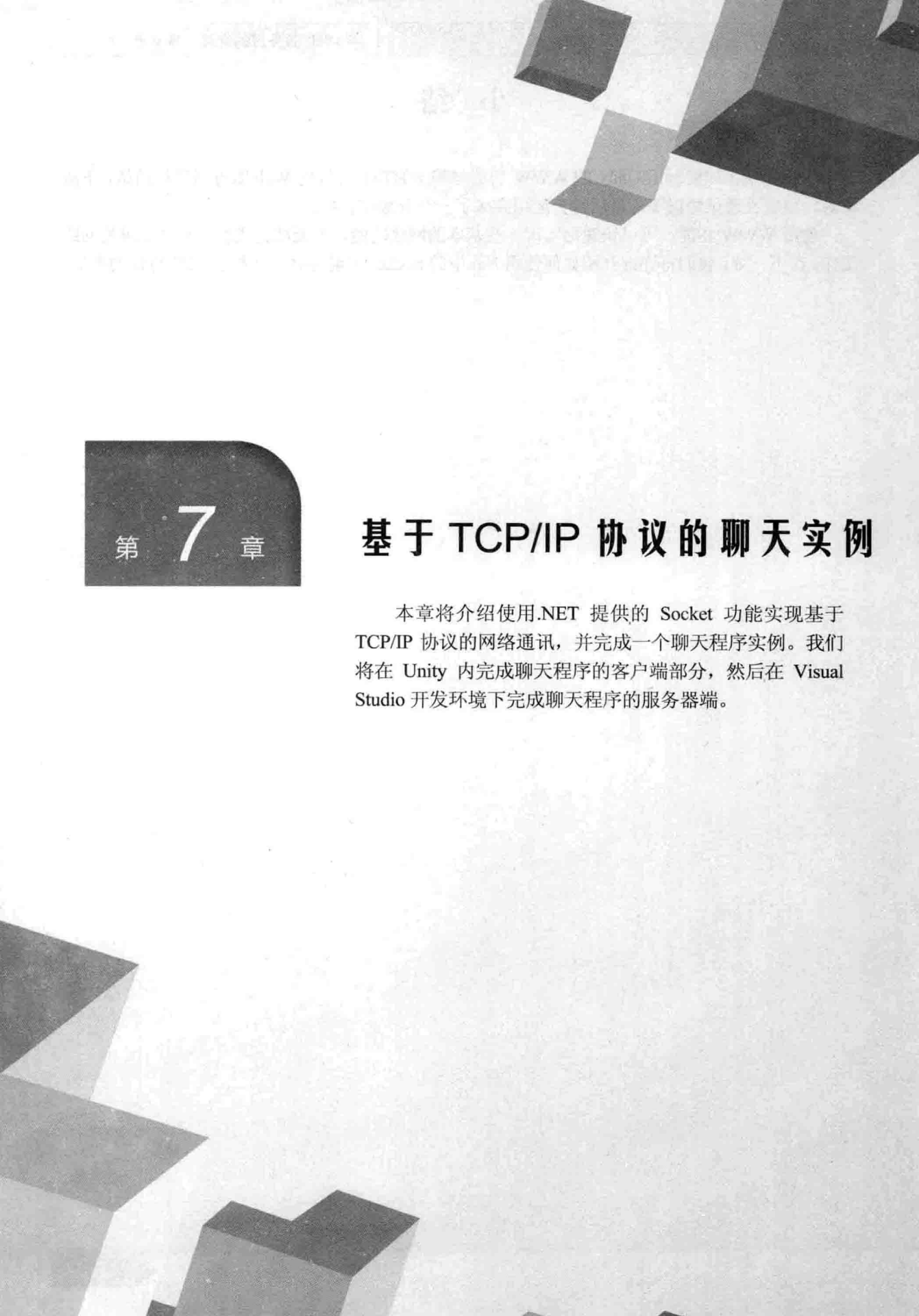
使用这个 MD5 函数非常简单, 在下面的代码示例中, 数据是包含有 “hello,world” 的一个字符串, 密钥为 123, 使用 Md5Sum 算出 32 位的校验码字符串。

```
string data = "hello,world";
string key = "123";
Md5Sum ( data + key ); // 返回 a3886a8079dd5fa34995a6434b72932a
```

小 结

本章介绍了如何运用 Unity 的 WWW 功能与基于 HTTP 协议的 Web 服务器进行通信, 下载资源, 自定义通讯协议等, 最后综合运用完成了一个分数排行榜。

使用 WWW 功能, 可以快速地实现一些基本的网络功能, 但无法完成有实时性要求的网络工作。在下一章, 我们将详细介绍如何使用 .Net 中的 Socket 实现与服务器基于 TCP 协议的通信。



第 7 章

基于 TCP/IP 协议的聊天实例

本章将介绍使用 .NET 提供的 Socket 功能实现基于 TCP/IP 协议的网络通讯，并完成一个聊天程序实例。我们将在 Unity 内完成聊天程序的客户端部分，然后在 Visual Studio 开发环境下完成聊天程序的服务器端。

7.1 TCP/IP 开发简介

在第 6 章,我们介绍了如何使用 Unity 的 WWW 功能与基于 HTTP 协议的 Web 服务器进行网络通信,这种通信方式容易实现,并可以用于动态下载资源,但缺点也很明显,它无法满足实时交互的网络需求。

TCP/IP 是 Transmission Control Protocol/Internet Protocol 的缩写,意思是传输控制/因特网协议,又名网络通讯协议,它是国际通用的基本网络协议,有着广泛的使用基础。从分层协议来说,它由 4 个层次组成:网络接口层、网络层、传输层和应用层。

通俗的理解 TCP/IP, TCP 负责监督网络传输,如果有问题就发出信号要求重传,保证数据可以安全的到达目的地, IP 即是指因特网分配给每台计算机的地址。

本章我们将使用 .NET 提供的 Socket 功能实现基于 TCP/IP 协议的网络功能,并完成一个聊天程序,这是一个相对复杂的工程,需要分别创建客户端和服务端

客户端是指运行在本地计算机上的程序,这部分我们将在 Unity 内完成,它的任务是向服务器端发出连接请求,当成功连接到服务器后,它可以接收服务器发来的数据,也可以向服务器发送数据。

服务器端是指运行在远程计算机上的程序,它的任务是监听来自客户端的连接请求,一旦建立连接,它可以收到客户端发来的数据,也可以向任何一个客户端发送数据。在客户端/服务器端模式下,任何客户端之间的通讯都需要经过服务器端。

客户端和服务端之间传输数据,需要将数据写入到一个流中,也就是将不同的数据存储在一个由字节组成的数组中,发送给对方。客户端和服务端需要使用相同的规则创建这个字节数组,这种规则通常也称作协议,一种最原始的做法,即是将数据按顺序序列化到字节数组中,客户端和服务端都使用约定好的相同顺序将数据从字节数组中取出。下面是一个简单的例子,将一个数字转为 byte 数组,然后再将这个 byte 数组转为 32 位的整型数字。

```
public byte[] WriteInt(int number)
{
    // 将一个 32 位的 int 整型数字转为 4 个字节的 byte 数组
    byte[] bs = System.BitConverter.GetBytes(number);
    return bs;
}

public int ReadInt(byte[] bs)
{
    // 将 byte 数组转为 32 位整型数字
    int number = System.BitConverter.ToInt32(bs, 0);
    return number;
}
```

不同的数据,字节长度不同,比如 32 位的整型是 4 个字节,64 位整型即是 8 个字节,短整型是 2 个字节,浮点数通常也是 4 个字节。需要注意的是,如果要在数据中传入字符串,因为

字符串的长度是不固定的,通常可以先发送一个整型数字表示字符串的长度,再跟着发送字符串。解析的时候会先读入字符串的长度,再根据这个长度解析字符串。

这种按顺序序列化数据到字节数组中的方法必须保证发送和接收方的处理方式高度一致,比较容易出错。实际上,C#已经提供了自动序列化对象的功能,不过如果使用这个功能,必须是客户端和服务端均使用C#语言。下面是一段示例代码,我们将 Player 对象序列化到一个 Stream 中,然后再将它反序列化到另一个 Player 对象中。

```
using System;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

// 定义一个类用于序列化,注意必须添加 Serializable 指明它可以序列化
[Serializable]
public class Player
{
    public int id;
    public string name;
    public int life;
}

static void Main(string[] args)
{
    // 初始化这个对象
    Player player = new Player();
    player.id = 1;
    player.name = "hero";
    player.life = 1000;

    using (MemoryStream stream = new MemoryStream())
    {
        // 创建序列化类
        BinaryFormatter bf = new BinaryFormatter();
        // 将 player 序列化到 stream 中
        bf.Serialize(stream, player);
        stream.Seek(0, SeekOrigin.Begin);
        // 将 stream 中的二进制数据反序列化到 player2
        Player player2 = (Player)bf.Deserialize(stream);
        // 打印 player2 的结果
        Console.WriteLine("{0},{1},{2}",
                           player.id, player2.name, player2.life);
    }
}
```

更进一步,我们可以将数据序列化为一些标准的格式,如 XML、Json 等,序列化和反序列化都有标准的方式,可以适用各种语言开发,保证可移植性,但缺点是 XML 和 Json 都基于文

本, 额外的标记字符也比较多, 浪费存储空间。

现在还有一种流行的序列化方式, 是使用 Google 的 Protobuf 作为传输协议。Protobuf 存储的序列化流也基于二进制, 序列化效率相当不错。我们可以定义一份由客户端和服务端共享的 Protobuf 数据协议, 它支持很多种语言, 因此即使客户端和服务端使用不同的语言, 比如 C# 和 C++, 但它们仍然可以使用相同的序列化方式互相沟通。Protobuf 现在已经被广泛地应用到很多网络游戏中, 在本章的最后, 我们会介绍一下如何在 Unity 中使用它。

当客户端或服务端接收到来自远程的数据, 我们将会把数据传入一个队列, 然后有一个专门的处理逻辑的线程处理队列中的数据, 根据不同的事件消息做出不同的逻辑反应。

7.2 一个简单的网络程序

在我们开始写一个聊天程序之前, 先来开发一个最简单的网络程序, 看看一些最基本的内容。

步骤 01 启动 Visual Studio, 新建一个 C# 的 Console 工程作为服务器工程。注意, 我们使用的是 .NET Framework 2.0, 因为 Unity 可能不支持高版本的 .NET 功能。

步骤 02 添加服务器代码如下:

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Sockets;

namespace SimpleServer
{
    class Program
    {
        static void Main(string[] args)
        {
            // 服务器 ip 地址
            string ip = "127.0.0.1";
            // 服务器端口
            int port = 8000;

            try
            {
                // 获得终端地址
                IPEndPoint ipe = new IPEndPoint(IPAddress.Parse(ip), port);
                // 创建 Socket
                Socket listener = new Socket(AddressFamily.InterNetwork,
                                                SocketType.Stream, ProtocolType.Tcp);
                // 将 Socket 绑定到终端地址上
```

```

        listener.Bind(ipe);
        // 开始监听 最大允许处理 1000 个连接
        listener.Listen(1000);
        Console.WriteLine("开始监听");
        // 开始接受客户端请求 程序在这里会阻住
        Socket mySocket=listener.Accept();
        byte[] bs = new byte[1024];
        int n = mySocket.Receive(bs);
        // 将客户端发来的数据返回给客户端
        mySocket.Send(bs);
        // 关闭与客户端的连接
        mySocket.Close();
    }
    catch (Exception e)
    {
        Console.WriteLine(e.Message);
    }
}
}
}

```

我们首先创建了一个 **Socket**，并将它绑定到服务器的地址和端口上，然后开始监听客户端的连接，在接受客户端的连接后，将客户端发来的数据存到一个 **byte** 数组中，并返回给客户端。为了简单，这里的示例使用的全部是同步的网络处理方式，程序会在等待客户端连接的时候堵塞，直到客户端连接后才会继续后面的程序。在本章后面的聊天实例中，我们将会使用异步的方式，程序则不会产生堵塞。

步骤 03 接下来，创建一个客户端工程，代码如下：

```

using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace SimpleClient
{
    class Program
    {
        static void Main(string[] args)
        {
            // 服务器 ip 地址
            string ip = "127.0.0.1";
            // 服务器端口
            int port = 8000;

```

```

try
{
    // 获得终端
    IPEndPoint ipe = new IPEndPoint(IPAddress.Parse(ip), port);
    // 创建 Socket
    Socket client = new Socket(ipe.AddressFamily,
                              SocketType.Stream, ProtocolType.Tcp);
    // 开始连接服务器, 程序在这里会阻住, 直接连接成功或失败
    client.Connect(ipe);
    Console.WriteLine("连接到服务器");
    // 向服务器发送数据
    string data = "hello,world";
    byte[] bs=UTF8Encoding.UTF8.GetBytes(data);
    client.Send(bs);
    // 接收到服务器的数据
    byte[] rev = new byte[256];
    // 接收到服务器返回的数据
    client.Receive(rev);
    Console.WriteLine(UTF8Encoding.UTF8.GetString(rev));
    // 关闭连接
    client.Close();
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
}
}

```

客户端的很多代码与服务器类似, 不同的是接受客户端连接的步骤变成了连接服务器的请求, 因为是同步处理, 程序在连接服务器时也会堵塞, 直到收到服务器的响应。

分别编译客户端和服务端, 先运行服务器, 再运行客户端, 客户端会向服务器发送字符串“hello,world”, 服务器收到后会把这个字符串返回给客户端, 客户端将它打印出来。最后的工程文件保存在资源文件目录 chapter07_Chat/BasicNetwork 下。

7.3 网络引擎

因为在客户端和服务端中所使用的很多功能是共享的, 所以我们可以先完成一个通用的网络引擎, 实现最基础的网络功能, 然后将这个引擎分别使用到聊天客户端和服务端中。

7.3.1 数据包

在传输网络数据的时候,接收方一次收到的数据长度可能是不确定的,比如客户端发送了 100 个字节给服务器,服务器有可能一次收到 100 个字节,也可能先收到 20 个,再收到 80 个。为了知道到底一个数据的长度是多少,我们将首先创建一个类,用于管理序列化的数据流,序列化、反序列化对象。

步骤 01 启动 Visual Studio,新建一个 C# 的 Class Library 工程。注意,我们使用的是 .NET Framework 2.0,因为 Unity 可能不支持高版本的 .NET 功能,如图 7-1 所示。

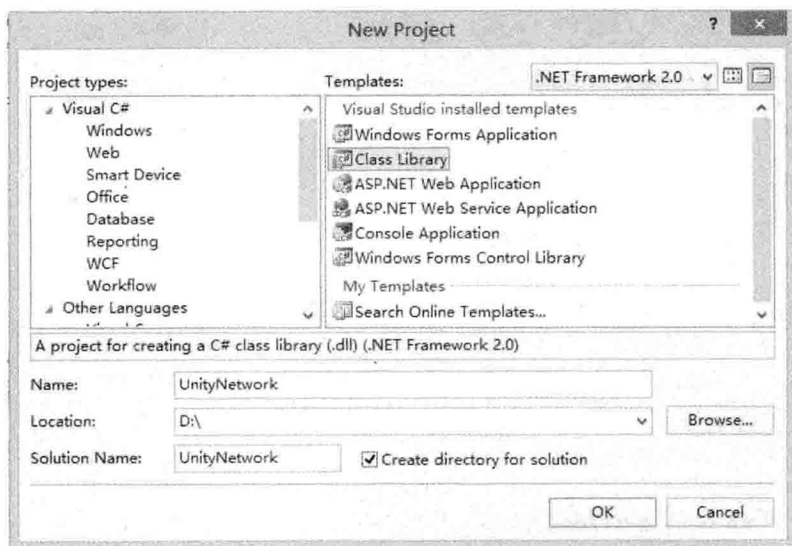


图 7-1 新建工程

步骤 02 创建 NetPacket.cs,这个类提供的功能主要包括两部分:一部分是将序列化的数据写入,并加入 4 个字节作为数据的“头”;另一部分是从 byte 数组的前 4 个字节解析数据长度,再读取相应长度的数据。首先我们定义一些属性。

```
using System;
using System.Collections.Generic;
using System.Net.Sockets;
using System.Text;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

namespace UnityNetwork
{
    public class NetPacket
    {
        // int32 占 4 个字节
        public const int INT32_LEN = 4;
```



```

// 头 占 4 个字节
public const int headerLength = 4;
// 身体 最大 512 个字节
public const int max_body_length = 512;
// 总长
public int Length
{
    get { return headerLength + bodyLength; }
}
// 当前数据体长
public int bodyLength { get; set; }

// byte 数组
public byte[] bytes { get; set; }

// 发送这个数据包的 socket
public Socket socket;
// 从网络上读到的数据长度
public int readLength { get; set; }

// 构造函数
public NetPacket()
{
    readLength = 0;
    bodyLength = 0;
    bytes = new byte[headerLength + max_body_length];
}
// 重置参数
public void Reset()
{
    readLength = 0;
    bodyLength = 0;
}

```

NetPacket 这个类的作用即是将在网络上传输的数据全部序列化存储到一个字节数组 **bytes** 中，这里要注意数据长度，比如一个 32 位的整型占 4 个字节。

我们定义了 **headerLength** 为 4 个字节，它指的是字节数组 **bytes** 中“头”的长度，即 32 位整型，占 4 个字节，它的值实际上就是最后 **bodyLength** 的长度。

bodyLength 是指数据的长度，**headerLength** 加上 **bodyLength** 的长度才是传输到网络上的数据的总长度。

readLength 表示的是网络传输过程中已读取的字节长度。前面提到，在网络传输的过程中，有时会一次收到所有数据，有时则需要分几次收到，**readLength** 即用来表示已经读取的数据长度，我们会一直等待读取的长度与 **bodyLength** 相同，确认收到一次数据。

步骤 03 创建两个函数，分别用于序列化和反序列化。这里使用的方法仅适用服务器和客户端均使用 C#。如果是使用 Protobuf 或其他协议，也可以使用类似的方式将序列化和反序列化的过程封装在函数中。

```
// 序列化对象
public byte[] Serialize<T>(T t)
{
    using (MemoryStream stream = new MemoryStream())
    {
        try
        {
            // 创建序列化类
            BinaryFormatter bf = new BinaryFormatter();
            // 序列化到 stream 中
            bf.Serialize(stream, t);
            stream.Seek(0, SeekOrigin.Begin);
            return stream.ToArray();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
            return null;
        }
    }
}

// 反序列化对象
public T Deserialize<T>(byte[] bs)
{
    using (MemoryStream stream = new MemoryStream(bs))
    {
        try
        {
            BinaryFormatter bf = new BinaryFormatter();
            T t = (T)bf.Deserialize(stream);
            return t;
        }
        catch (Exception ex)
        {
            Console.WriteLine("Deserialize:"+ex.Message);
            return default(T);
        }
    }
}
```

步骤 04 下面的函数主要用来写数据，将不同类型的序列化数据写到 byte 数组中。

```
// 开始写入数据
public void BeginWrite( string msg )
{
    // 初始化体长为 0
    bodyLenght = 0;
    WriteString(msg);
}

// 写整型
public void WriteInt(int number)
{
    if (bodyLenght + INT32_LEN > max_body_length)
        return;

    byte[] bs = System.BitConverter.GetBytes(number);
    bs.CopyTo(bytes, headerLength + bodyLenght);
    bodyLenght += INT32_LEN;
}

// 写字符串
public void WriteString(string str)
{
    int len = System.Text.Encoding.UTF8.GetByteCount(str);
    this.WriteInt(len);
    if (bodyLenght + len > max_body_length)
        return;

    System.Text.Encoding.UTF8.GetBytes(str, 0, str.Length, bytes,
                                         headerLength + bodyLenght);
    bodyLenght += len;
}

// 写入 byte 数组
public void WriteStream(byte[] bs)
{
    WriteInt(bs.Length);
    if (bodyLenght + bs.Length > max_body_length)
        return;

    // 压入数据流
    bs.CopyTo(bytes, headerLength + bodyLenght);
}
```

```

        bodyLenght += bs.Length;
    }

    // 直接写入一个序列化的对象
    public void WriteObject<T>(T t)
    {
        byte[] bs = Serialize<T>(t);
        WriteStream(bs);
    }

    // 将数据长度转为 4 个字节存到 byte 数组的最前面
    public void EncodeHeader()
    {
        byte[] bs = System.BitConverter.GetBytes(bodyLenght);

        bs.CopyTo(bytes, 0);
    }

```

BeginWrite 的作用是在字节数组中写入一个消息标识符, 即标识当前数据包的作用, 这个函数要最先调用。当将所有数据写入后, 最后不要忘记使用 **EncodeHeader** 函数将数据的长度写到 **byte** 数组最前面的 4 个字节中。

步骤 05 下面的函数主要用来读取、反序列化数据, 这里的功能与写数据的方式是一一对应的。

```

// 开始读取
public void BeginRead(out string msg)
{
    bodyLenght = 0;
    ReadString(out msg);
}

// 读 int
public void ReadInt(out int number)
{
    number = 0;
    if (bodyLenght + INT32_LEN > max_body_length)
        return;
    number = System.BitConverter.ToInt32(bytes, headerLength + bodyLenght);
    bodyLenght += INT32_LEN;
}

// 读取一个字符串
public void ReadString(out string str)
{

```



```

        str = "";
        int len = 0;
        ReadInt(out len);
        if (bodyLenght + len > max_body_length)
            return;
        str = Encoding.UTF8.GetString(bytes, headerLength + bodyLenght, (int)len);
        bodyLenght += len;
    }

    // 读取 byte 数组
    public byte[] ReadStream()
    {
        int size = 0;
        ReadInt(out size);
        if (bodyLenght + INT32_LEN > max_body_length)
            return null;
        byte[] bs = new byte[size];
        for (int i = 0; i < size; i++)
        {
            bs[i] = bytes[headerLength + bodyLenght + i];
        }
        bodyLenght += size;
        return bs;
    }

    // 直接将读取的 byte 数组反序列化
    public T ReadObject<T>()
    {
        byte[] bs = ReadStream();
        if (bs == null)
        {
            return default(T);
        }
        return Deserialize<T>(bs);
    }

    // 由 byte 数组最前面 4 个字节得到数据的长度
    public void DecodeHeader()
    {
        bodyLenght = System.BitConverter.ToInt32(bytes, 0);
    }

```

DecodeHeader 的功能是从 **byte** 数组最前面的 4 个字节中解析出数据的长度，并保存给 **bodyLenght**。当在网络上接收数据的时候，我们先接收数据包的最前面 4 个字节，然后使用

DecodeHeader 即可算出后面的数据长度, 再继续接收后面的数据。

7.3.2 逻辑处理

创建 NetworkManager.cs, 这个类的作用是处理逻辑, 主要包括两部分: 一部分是注册网络消息, 使每个消息标识符与一个回调函数相关联; 另一部分是将数据包入队并分发给相应的回调函数, 代码如下:

```
using System.Collections;
using System.Collections.Generic;
using System.Text;
namespace UnityNetwork
{
    public class NetworkManager
    {
        // 逻辑处理使用一个独立线程, 与网络的线程分开
        System.Threading.Thread myThread;
        // 代理回调函数
        public delegate void OnReceive( NetPacket packet );
        // 每个消息对应一个 OnReceive 函数
        public Dictionary<string, OnReceive> handlers;
        // 存储网络数据的队列
        private Queue Packets = new System.Collections.Queue();

        public NetworkManager()
        {
            handlers = new Dictionary<string, OnReceive>();
            // 注册连接成功, 丢失连接消息
            AddHandler("OnAccepted", OnAccepted);
            AddHandler("OnConnected", OnConnected);
            AddHandler("OnConnectFailed", OnConnectFailed);
            AddHandler("OnLost", OnLost);
        }

        // 注册消息
        public void AddHandler(string msgid, OnReceive handler)
        {
            handlers.Add(msgid, handler);
        }

        // 数据包入队
        public void AddPacket( NetPacket packet )
        {
            lock (Packets)
```

```
        {
            Packets.Enqueue(packet);
        }
    }

    // 数据包出队
    public NetPacket GetPacket()
    {
        lock (Packets)
        {
            if (Packets.Count == 0)
                return null;
            return (NetPacket)Packets.Dequeue();
        }
    }

    // 开始执行另一个线程处理逻辑
    public void StartThreadUpdate()
    {
        // 为逻辑部分建立新的线程
        myThread = new System.Threading.Thread(
            new System.Threading.ThreadStart(ThreadUpdate));
        myThread.Start();
    }

    // 逻辑线程
    protected void ThreadUpdate()
    {
        while (true)
        {
            // 为了节约 cpu, 每次循环暂停 30 帧
            System.Threading.Thread.Sleep(30);
            Update();
        }
    }

    // 处理数据包, 更新逻辑
    public void Update()
    {
        NetPacket packet = null;
        for (packet = GetPacket(); packet != null; )
        {
            string msg = "";
            // 获得消息标识符
            packet.BeginRead(out msg);
        }
    }
}
```

```

        OnReceive handler = null;
        if (handlers.TryGetValue(msg, out handler))
        {
            // 根据消息标识符找到相应的 OnReceive 代理函数
            if (handler != null)
                handler(packet);
        }
        packet = null;
    }
}

// 处理服务器接受客户端的连接
public virtual void OnAccepted(NePacket packet)
{
}

// 处理客户端取得与服务器的连接
public virtual void OnConnected(NePacket packet)
{
}

// 处理客户端取得与服务器连接失败
public virtual void OnConnectFailed(NePacket packet)
{
}

// 处理丢失连接
public virtual void OnLost(NePacket packet)
{
}
}
}

```

AddHandler 的功能是注册网络消息，实际上是将字符串形式的消息标识符和回调函数以键值对应的方式存入到一个 **Dictionary** 中。我们这里的回调函数是使用了一个代理函数 **OnReceive**，在应用的时候，只需要定义一个与 **OnReceive** 函数返回值、参数一致的函数，使用 **AddHandler** 传入该函数名即可。

AddPacket 函数将数据包入队，然后在 **Update** 函数中使用 **GetPacket** 获得数据包。因为网络和逻辑处理有可能是在不同的线程中，所以在入队出队的时候使用了 **lock** 防止多线程带来的问题。

StartThreadUpdate 函数会开启一个新的线程运行 **ThreadUpdate** 函数，它的内部是一个循环，会一直检查是否有新的数据包入队，有则处理。在 **Unity** 中使用 **NetworkManager** 这个类的时候，我们不会使用多线程，而是直接在 **Unity** 的循环函数中调用 **NetworkManager** 对象的 **Update** 函数实现循环。

7.3.3 网络功能

下面将创建一个类封装 TCP/IP 协议的网络功能,因为客户端和服务端有很多网络功能类似,比如接收和发送数据,所以这里并没有将客户端和服务端写为两个不同的类,而是将它们的功能封装在同一个类中。

步骤 01 创建 TCPPeer.cs, 首先定义它的构造函数。

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net;
using System.Net.Sockets;

namespace UnityNetwork
{
    public class TCPPeer
    {
        // 是否是服务器
        public bool isServer { set; get; }
        // 使用的 socket
        public Socket socket;
        // 网络管理器
        NetworkManager networkMgr;

        public TCPPeer ( NetworkManager netMgr )
        {
            networkMgr = netMgr;
        }

        // 添加内部消息
        private void AddInternalPacket(string msg, Socket sk )
        {
            // 通知丢失连接
            NetPacket p = new NetPacket();
            p.socket = sk;
            p.BeginWrite(msg);
            networkMgr.AddPacket(p);
        }
    }
}
```

isServer 用来判断 TCPPeer 实例是客户端还是服务器。如果是客户端, Socket 则用于连接服务器, 否则用于监听来自客户端的连接。在实例化 TCPPeer 的时候, 在构造函数中需要传入

NetworkManager 对象, 这样 TCPPeer 可以将从网络上收到的数据传送到 NetworkManager 对象中进行逻辑处理。

AddInternalPacket 函数用于向 NetworkManager 对象添加一些内部消息, 这类消息的数据并不是从网络上得到的, 比如连接失败, 连接成功之类。

步骤 02 接下来添加服务器的功能。创建 Listen 和 ListenCallback 函数, 用于监听来自客户端的连接。与本章第 2 节使用的方法不同, 我们现在使用的全部是异步方式, 效率更高。

```
// 作为服务器, 开始监听
public void Listen( string ip, int port, int backlog=1000 )
{
    isServer = true;
    // ip 地址
    IPEndPoint ipe = new IPEndPoint(IPAddress.Parse(ip), port);
    // 创建 socket
    socket = new Socket( AddressFamily.InterNetwork,
                        SocketType.Stream, ProtocolType.Tcp);
    try
    {
        // 将 socket 绑定到地址上
        socket.Bind(ipe);
        // 开始监听
        socket.Listen(backlog);
        // 开始异步接受连接
        socket.BeginAccept(new System.AsyncCallback(ListenCallback), socket);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

// 异步接受一个新的连接
void ListenCallback(System.IAsyncResult ar)
{
    // 取得服务器 socket
    Socket listener = (Socket)ar.AsyncState;
    try
    {
        // 获得客户端的 socket
        Socket client = listener.EndAccept(ar);

        // 通知服务器接受一个新的连接
    }
}
```



```

        AddInternalPacket("OnAccepted", client);
        // 创建接收数据的数据包
        NetPacket packet = new NetPacket();
        packet.socket = client;
        // 开始接收从来自客户端的数据
        client.BeginReceive(packet.bytes, 0, NetPacket.headerLength, SocketFlags.
None, new System.AsyncCallback(ReceiveHeader), packet);
    }
    catch (System.Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
    // 继续接受其他连接
    listener.BeginAccept(new System.AsyncCallback(ListenCallback), listener);
}

```

Listen 函数创建一个 Socket 异步监听来自远程的连接, ListenCallback 是异步监听的回调函数, 如果连接成功, 取得远程客户端的 Socket, 开始异步接收客户端发来的数据。

步骤 03 创建 Connect 和 ConnectionCallback 函数, 它们是用客户端的函数。Connect 负责向远程服务器发起异步连接, ConnectionCallback 是异步连接的回调函数, 处理连接结果, 如果连接成功, 则开始异步接收来自服务器的数据, 否则向消息队列发送连接失败的消息。

```

// 作为客户端, 开始连接服务器
public void Connect( string ip, int port )
{
    isServer = false;
    IPEndPoint ipe = new IPEndPoint(IPAddress.Parse(ip), port);
    try
    {
        socket = new Socket(AddressFamily.InterNetwork,
                            SocketType.Stream, ProtocolType.Tcp);

        // 开始连接
        socket.BeginConnect(ipe, new System.AsyncCallback(ConnectionCallback),
                            socket);
    }
    catch (System.Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

// 异步连接回调

```

```

void ConnectionCallback(System.IAsyncResult ar)
{
    Socket client = (Socket)ar.AsyncState;
    try
    {
        // 与服务器取得连接
        client.EndConnect(ar);
        // 通知已经成功连接到服务器
        AddInternalPacket("OnConnected", client);
        // 开始异步接收服务器信息
        NetPacket packet = new NetPacket();
        packet.socket = client;
        client.BeginReceive(packet.bytes, 0, NetPacket.headerLength,
            SocketFlags.None, new System.AsyncCallback(ReceiveHeader), packet);
    }
    catch (System.Exception ex)
    {
        Console.WriteLine(ex.Message);
        AddInternalPacket("OnConnectFailed", client);
    }
}

```

步骤 04 无论是创建用于监听的服务器 Socket 还是用于发起连接的客户端 Socket, 最后都会进入接收数据状态。接收数据主要是通过 ReceiveHeader 和 ReceiveBody 两个函数。

```

// 接收消息头
void ReceiveHeader(System.IAsyncResult ar)
{
    NetPacket packet = (NetPacket)ar.AsyncState;
    try
    {
        // 返回网络上接收的数据长度
        int read = packet.socket.EndReceive(ar);
        // 已断开连接
        if (read < 1)
        {
            // 通知丢失连接
            AddInternalPacket("OnLost", packet.socket);
            return;
        }

        packet.readLength += read;
        // 消息头必须读满 4 个字节
    }
}

```

```

        if (packet.readLength < NetPacket.headerLength)
        {
            packet.socket.BeginReceive(packet.bytes,
                packet.readLength, // 存储偏移已读入的长度
                NetPacket.headerLength - packet.readLength, // 这次只读入剩余的数据
                SocketFlags.None,
                new System.AsyncCallback(ReceiveHeader),
                packet);
        }
        else
        {
            // 获得消息体长度
            packet.DecodeHeader();
            packet.readLength = 0;
            // 开始读取消息体
            packet.socket.BeginReceive(packet.bytes,
                NetPacket.headerLength,
                packet.bodyLength,
                SocketFlags.None,
                new System.AsyncCallback(ReceiveBody),
                packet);
        }
    }
}
catch (System.Exception ex)
{
    Console.WriteLine("ReceiveHeader:" + ex.Message);
}
}

// 接收体消息
void ReceiveBody(System.IAsyncResult ar)
{
    NetPacket packet = (NetPacket)ar.AsyncState;
    try
    {
        // 返回网络上接收的数据长度
        int read = packet.socket.EndReceive(ar);
        // 已断开连接
        if (read < 1)
        {
            // 通知丢失连接
            AddInternalPacket("OnLost", packet.socket);
            return;
        }
    }
}

```

```

        packet.readLength += read;
        // 消息体必须读满指定的长度
        if ( packet.readLength < packet.bodyLength )
        {
            packet.socket.BeginReceive(packet.bytes,
                NetPacket.headerLength + packet.readLength,
                packet.bodyLength - packet.readLength,
                SocketFlags.None,
                new System.AsyncCallback(ReceiveBody),
                packet);
        }
        else
        {
            // 将消息传入到逻辑处理队列
            networkMgr.AddPacket(packet);

            // 下一个读取
            packet.Reset();
            packet.socket.BeginReceive(packet.bytes,
                0,
                NetPacket.headerLength,
                SocketFlags.None,
                new System.AsyncCallback(ReceiveHeader),
                packet);
        }
    }
}
catch (System.Exception ex)
{
    Console.WriteLine("ReceiveBody:" + ex.Message);
}
}

```

在 `ReceiveHeader` 函数中, 我们的目标是接收到 4 个字节, 它是一个 `int` 值, 表示数据体的长度。如果只接收到 0 个字节, 表示已经与服务器断开连接。因为一次接收的数据有可能小于 4 个字节, 这种情况则继续接收, 直到读满 4 个字节。

`ReceiveBody` 函数的实现方式与 `ReceiveHeader` 类似, 当 `ReceiveHeader` 读满 4 个字节后, 我们就知道了数据的实际长度, 在 `ReceiveBody` 中完成对数据的接收, 并将数据传入消息队列, 然后调用 `ReceiveHeader` 继续新一轮数据接收。

步骤 05 `Socket` 除了能接收数据, 还需要向远程连接发送数据。创建 `Send` 函数异步发送数据, `SendCallback` 是异步发送数据的回调函数。


```

// 向远程发送消息
public void Send( Socket sk, NetPacket packet )
{
    NetworkStream ns;
    lock (sk)
    {
        ns = new NetworkStream(sk);
        if (ns.CanWrite)
        {
            try
            {
                ns.BeginWrite( packet.bytes, 0, packet.Length,
                               new System.AsyncCallback(SendCallback), ns);
            }
            catch (System.Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}

//发送回调
private void SendCallback(System.IAsyncResult ar)
{
    NetworkStream ns = (NetworkStream)ar.AsyncState;
    try
    {
        ns.EndWrite(ar);
        ns.Flush();
        ns.Close();
    }
    catch (System.Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

```

在发送数据的过程中，使用到了 `NetworkStream` 这个类，它专门用来处理 TCP 连接的数据流，这里主要使用它将数据异步发送出去。

7.3.4 创建聊天协议

最后，我们将添加一个用于聊天的协议，它包括一个用户名和聊天内容。


```

namespace Chat
{
    [System.Serializable]
    public class ChatProto
    {
        // 用户名
        public string userName;
        // 聊天内容
        public string chatMsg;
    }
}

```

在 Visual Studio 中编译工程, 在当前工程的 bin 目录内会出现一个 UnityNetwork.dll 文件, 我们将把它使用到聊天客户端和服务端中。

7.4 聊天客户端

聊天客户端将在 Unity 中完成。在 Unity 中有两种方式使用前面完成的网络引擎: 一种方法是直接将 UnityNetwork.dll 复制到 Unity 工程 Assets 目录内的任何地方; 另一种方式是直接将原始代码文件复制到 Unity 工程内。本例中将采用第一种方式。

步骤 01 新建一个 Unity 工程。

步骤 02 将前面创建的 UnityNetwork.dll 复制到当前工程内。

步骤 03 创建脚本 ChatManager.cs, ChatManager 类继承自 NetworkManager, 它的内部包括一个 TCPPeer 对象, 我们使用它与服务器建立连接并收发数据, 因为一切都在前面的网络引擎部分完成了, 这里只是简单的应用, 代码非常简单。

```

using UnityEngine;
using System.Collections;
using UnityEngine;

public class ChatManager : NetworkManager {

    TCPPeer client;
    // Use this for initialization
    public void Start () {
        // 连接到服务器
        client = new TCPPeer(this);
        client.Connect("127.0.0.1", 10001);
    }
    // 发送聊天消息
    public void Send( NetPacket packet )

```

```

    {
        client.Send(client.socket, packet);
    }
    // 处理丢失连接
    public override void OnLost (NetPacket packet)
    {
        Debug.Log ("丢失与服务器的连接");
    }
    // 处理客户端取得与服务器的连接
    public override void OnConnected (NetPacket packet)
    {
        Debug.Log ("成功连接到服务器");
    }
    // 处理客户端与服务器连接失败
    public override void OnConnectFailed (NetPacket packet)
    {
        Debug.Log ("连接服务器失败, 请退出");
    }
}

```

步骤 04 创建脚本 ChatClient.cs, 并将其指定给场景中的游戏体。这是一个标准的 Unity 脚本, 它包括一个简单的输入框 UI 用于输入聊天信息, 还包括一个按钮发送聊天信息。

```

using UnityEngine;
using System.Collections;
using UnityNetwork;

public class ChatClient : MonoBehaviour {
    ChatManager clientPeer;
    // 收到的聊天消息
    public string revString = "";
    // 输入的聊天消息
    protected string inputString = "";

    // Use this for initialization
    void Start () {
        // 在这里创建 ChatManager 实例
        clientPeer = new ChatManager();
        // 注册一个聊天消息
        clientPeer.AddHandler("chat", OnChat);
        clientPeer.Start();
    }
}

```

```

// Update is called once per frame
void Update () {
    // unity 中处理逻辑使用的是单线程
    clientPeer.Update();
}

void OnGUI()
{
    // 显示收到的聊天记录
    GUI.Label(new Rect(5, 5, 200, 30), revString);

    // 输入聊在消息
    inputString = GUI.TextField(new Rect(Screen.width * 0.5f - 200,
        Screen.height * 0.5f - 20, 400, 40), inputString);

    // 发送聊天消息
    if ( GUI.Button( new Rect( Screen.width*0.5f-100,Screen.height*0.65f,200,30),
        "发送消息") )
    {
        // 向服务器发送聊天消息
        SendChat();
    }
}

// 发送聊天消息
void SendChat()
{
    // 聊天数据包
    Chat.ChatProto proto = new Chat.ChatProto();
    proto.userName = "客户端";
    proto.chatMsg =inputString;
    NetPacket p = new NetPacket();
    p.BeginWrite("chat");
    p.WriteObject<Chat.ChatProto>(proto);
    p.EncodeHeader();
    clientPeer.Send(p);
    //清空_inputString
    inputString = "";
}

// 处理聊天消息
public void OnChat(NetPacket packet)
{
    Debug.Log("收到服务器的消息");
    Chat.ChatProto proto = packet.ReadObject<Chat.ChatProto>();
}

```

```

        revString = proto.userName + ":" + proto.chatMsg;
    }
}

```

在 Start 函数中，我们注册了一个 chat 消息，并在 OnChat 函数中处理这个消息。

运行程序，将显示一些简单的 UI，因为我们还没有创建服务器端，所以会收到连接服务器失败的消息，如图 7-2 所示。

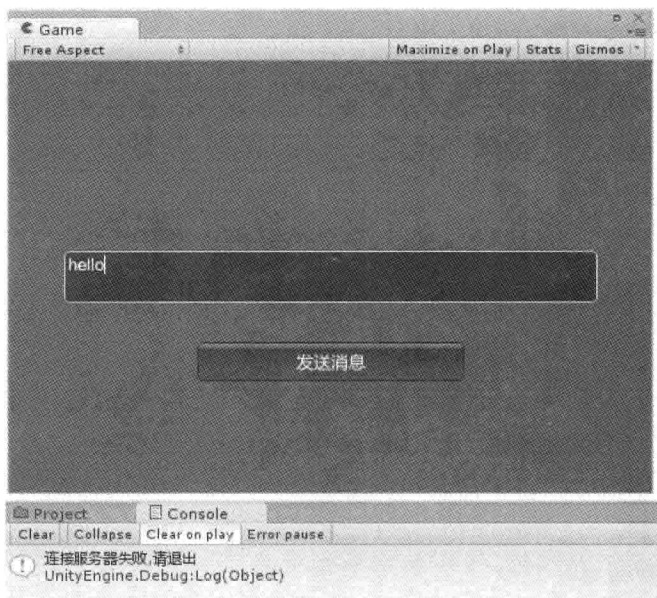


图 7-2 聊天客户端界面

7.5 聊天服务器端

用于聊天的服务器端将在 Visual Studio 中创建，它是一个控制台程序，只需要负责收发数据、逻辑处理等，不需要显示任何图形界面。

- 步骤 01** 在前面创建的 UnityNetwork 工程中添加一个控制台工程，命名为 ChatServer，如图 7-3 所示。
- 步骤 02** 在 ChatServer 工程上单击右键，在弹出的快捷菜单中选择【Add Reference】，然后选择网络引擎的 UnityNetwork.dll，如图 7-4 所示。
- 步骤 03** 创建一个 ChatServer 类，继承自 NetworkManager，它将作为聊天服务器。

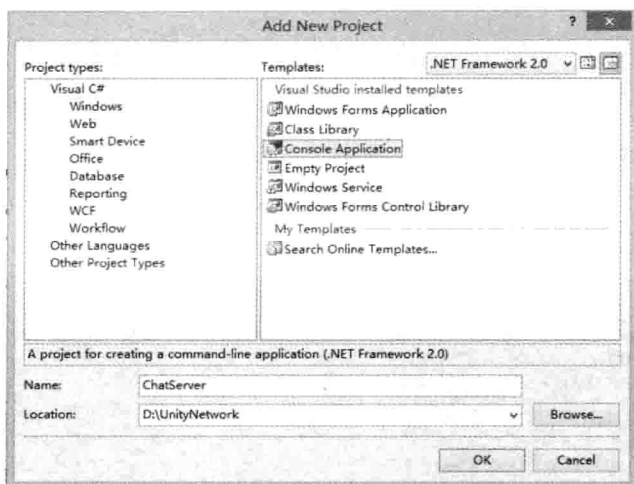


图 7-3 添加控制台工程

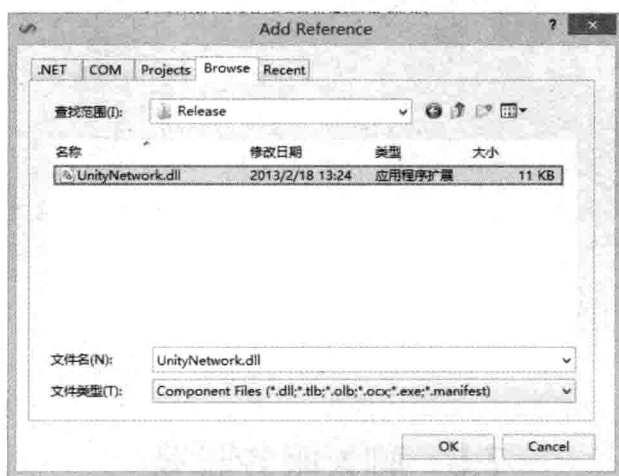


图 7-4 添加 dll

在 ChatServer 中, 当收到 OnAccepted 连接成功的消息时, 我们使用一个 List 保存接受的客户端连接, 以便在后面可以找到任何一个连接的客户端。

在 StartServer 函数中, 与客户端一样, 我们注册了一个 chat 消息用于聊天, 并在 OnChat 函数中处理这个消息, 将收到的聊天消息转发给其他所有客户端。

在启动服务器后, 我们调用了 StartThreadUpdate 函数, 在一个独立的线程中处理消息队列。完整的聊天服务器端代码如下:

```
using System;
using System.Collections.Generic;
using System.Net;
using System.Net.Sockets;
using UnityNetwork;

namespace ChatServer
```



```

{
    class Program
    {
        static void Main(string[] args)
        {
            ChatServer server = new ChatServer();
            // 启动服务器, 绑定到本地 ip 地址, 使用端口 10001
            server.StartServer("127.0.0.1", 10001);
        }

        public class ChatServer : NetworkManager
        {
            // 保存所有的客户端连接
            List<Socket> peerList;
            // 服务器
            TCPPeer server;

            public ChatServer()
            {
                // 创建一个列表保存每个客户端的 Socket
                peerList = new List<Socket>();
            }
            // 启动服务器
            public void StartServer( string ip , int port )
            {
                // 注册事件, 本例中只有一个聊天消息
                AddHandler("chat", OnChat);
                server = new TCPPeer(this);
                server.Listen(ip, port);
                // 启动另一个线程处理消息队列
                this.StartThreadUpdate();
                Console.WriteLine("启动聊天服务器");
            }

            // 处理服务器接受客户端的连接
            public override void OnAccepted(NetPacket packet)
            {
                Console.WriteLine("接受新的连接");
                peerList.Add(packet.socket);
            }

            // 处理丢失连接
            public override void OnLost(NetPacket packet)
            {

```

```

        Console.WriteLine("丢失连接");
        peerList.Remove(packet.socket);
    }

    // 处理聊天消息
    public void OnChat(NetPacket packet)
    {
        // 在服务器上显示聊天内容
        Chat.ChatProto proto = packet.ReadObject<Chat.ChatProto>();
        if ( proto!=null )
            Console.WriteLine(proto.userName + ":" + proto.chatMsg);

        packet.BeginWrite("chat");
        packet.WriteObject<Chat.ChatProto>(proto);
        packet.EncodeHeader();

        // 将消息转发给所有聊天客户端
        foreach (Socket sk in peerList)
        {
            server.Send(sk, packet);
        }
    }
}
}
}

```

运行服务器端，然后启动多个客户端，在客户端之间即可进行聊天。客户端每次发出的聊天消息都会被群发给其他所有客户端，如果希望只发送给指定的客户端，则需要为每个客户端指定一个标识符辨别身份。

本示例最后的工程文件保存在资源文件目录 chapter07_Chat 中，在资源文件目录 builds/chat 中提供了已经编译好的服务器和客户端的可执行文件。

7.6 Protobuf 简介

在聊天实例中，我们使用的协议序列化方式只适用于客户端与服务器均使用 C# 语言。本章最后将额外介绍一个第三方的工具——Google 公司推出的 Protobuf，它可以支持多种语言，如 C++、Java 等，只要协议是相同的，使用不同语言的 Protobuf 通信完全没问题。

使用 Protobuf，我们需要先定义一份通信协议，然后即可在程序中将其自动序列化为 byte 数组，也可以将 byte 数组反序列化。下面我们将介绍一下 Protobuf 针对 Unity 的基本应用。

步骤 01 Google 官方的 Protobuf 暂时不支持 C# 语言，好在有很多开发者为 Protobuf 开发了支持 C# (.NET) 的插件，打开网址 <http://code.google.com/p/protobuf-net/>，这是 .NET

版本的主页。

步骤 02 下载 `protobuf-net_r668.zip` (本书资源文件目录 `Software` 内也提供了这个压缩包), 将其解压, 然后将路径 `Full\unity` 内的所有文件复制到 `Unity` 工程目录内, 这里建议复制到 `Plugins` 目录内。

步骤 03 使用任意文本编辑器创建协议文件, 本例将文件命名为 `chatapp.proto`, 然后定义协议如下:

```
package ChatAPP;

message User {
    required string name = 1;
    required string chat = 2;
    optional string email = 3;
}

message Chat {
    repeated User user = 1;
}
```

这里我们可以将 `package` 看作是名称域, `message` 看作是 `class`, `User` 有 3 个成员, 用户名、聊天内容和 E-mail。

标识符 `required` 表示这是个必须的数据, `optional` 表示这是可选的。

在 `Chat` 中, 只有一个成员是 `User`, 它的前面有一个标识符 `repeated`, 表示一个 `Chat` 可以拥有多个 `User`。

步骤 04 在 `protobuf-net_r668` 压缩包内有一个名为 `ProtoGen` 的文件夹, 将 `chatapp.proto` 复制到这个文件夹。

步骤 05 在 `Windows` 开始菜单输入 `CMD`, 打开控制台窗口, 将路径设置到 `ProtoGen`, 然后输入下面的脚本, 转出文件 `chatapp.cs`, 将这个文件复制到客户端和服务端。

```
protogen -i:chatapp.proto -o:chatapp.cs
```

注意, 因为每次修改 `.proto` 文件都要重新生成新的 `.cs` 文件, 为了方便, 可以将生成 `.cs` 文件的脚本写在一个 `.cmd` 文本文件中, 这样每次操作只需要选择这个文件即可, 如图 7-5 所示。

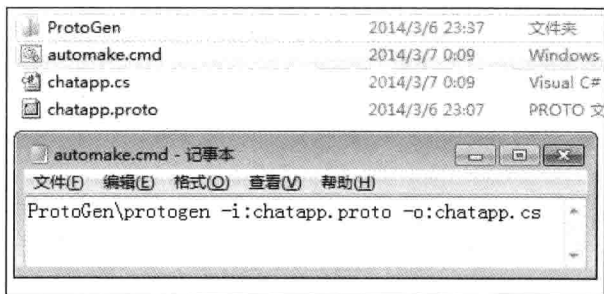


图 7-5 使用 `cmd` 脚本生成 `.cs` 文件

步骤 06 在 C# 中序列化 User 的方法如下:

```
ChatAPP.User user = new ChatAPP.User();
user.name = "用户甲";
user.chat = "hello,world";

using (System.IO.MemoryStream stream = new System.IO.MemoryStream())
{
    ProtoBuf.Serializer.Serialize<ChatAPP.User>(stream, user);
    Byte[] bs = stream.ToArray();
}
```

步骤 07 从一个 byte 数组中解析 User 的方法如下:

```
// bs 是一个 byte 数组, 它应当是从网络或文件中得到的
using (System.IO.MemoryStream stream = new System.IO.MemoryStream(bs))
{
    ChatAPP.User user = ProtoBuf.Serializer.Deserialize<ChatAPP.User>(stream);
}
```

步骤 08 repeated 类型的数据比较特殊, 它在 C# 中实际是一个数组:

```
ChatApp.Chat chat;
chat.user.Add(user);
```

虽然使用 Protobuf 很简单, 但当将它使用到 IOS 平台会遇到点麻烦, .Net 版本的 Protobuf 用到了 C# 的反射功能, 这个功能当前在苹果 iOS 系统中是不能使用的, 好在 .Net 版本的 Protobuf 提供了一个方法解决这个问题。这里以前面创建的 chatapp.proto 这个文件为例, 操作步骤如下:

步骤 01 在 Visual Studios 中创建一个 C# 的 Class Library 工程, 将前面得到的 chatapp.cs 文件添加到这个工程。

步骤 02 在工程中选择 Add Reference 将 Protobuf 提供的 protobuf-net.dll 添加到当前工程, 如图 7-6 所示。



图 7-6 添加 Protobuf 的 dll 文件

步骤 03 编译当前工程，在输出目录生成一个叫 ProtoLib.dll 的文件，这个文件的名称并不重要，如图 7-7 所示。

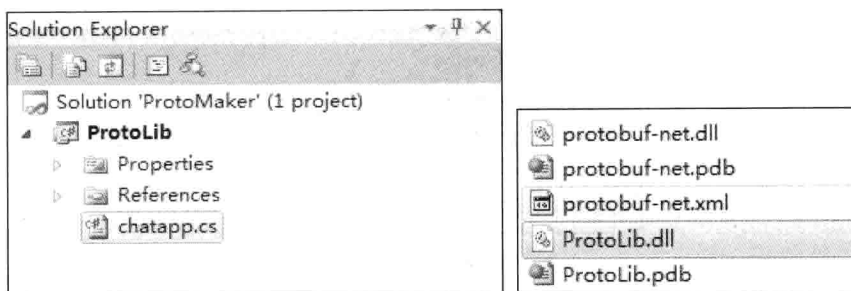


图 7-7 生成一个 ProtoLib.dll 文件

步骤 04 添加一个新的 C# 控制台工程，将 protobuf-net.dll 和上一步生成的 ProtoLib.dll 添加到这个工程中。添加代码如下：

```
using System;
using System.Collections.Generic;

namespace ProtoBuilder
{
    class Program
    {
        static void Main(string[] args)
        {
            ProtoBuf.Meta.RuntimeTypeModel model =
                ProtoBuf.Meta.RuntimeTypeModel.Create();
            model.Add(typeof(ChatAPP.Chat), true);
            model.Add(typeof(ChatAPP.User), true);
            model.Compile("ChatSerializer", "ChatSerializer.dll");
        }
    }
}
```

这里的代码首先创建了一个 RuntimeTypeModel 对象，然后将 proto 文件中定义的类都添加到该对象中，最后的参数 ChatSerializer 是类的名称域，ChatSerializer.dll 是生成的文件名。

步骤 05 编译当前工程，在输出目录找到 ProtoBuilder.exe 文件，双击运行它，生成 ChatSerializer.dll 文件，如图 7-8 所示。将 ChatSerializer.dll 和 ProtoLib.dll 这两个 dll 文件都复制到 Unity 工程中。注意，在 Unity 工程中已经不需要 chatapp.cs 这个文件，dll 文件中已经包括了它。

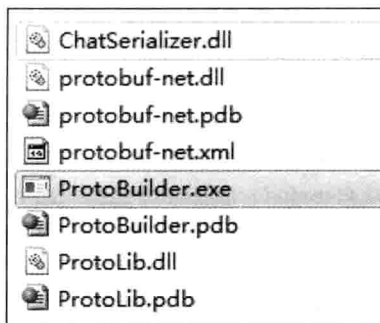


图 7-8 生成 dll 文件

从最初生成 Protobuf 的.cs 文件,到反复编译 dll 文件,再将它们复制到 Unity 工程中是件非常麻烦的事,在实际的项目中,最好编写一些批处理脚本来完成这些事情。

步骤 06 在 Unity 的 C#脚本中序列化 User,注意与之前代码的不同。

```
ChatAPP.User user = new ChatAPP.User();
user.name = "用户甲";
user.chat = "hello,world";
byte[] bs;
using (System.IO.MemoryStream stream = new System.IO.MemoryStream())
{
    ProtoBuf.Meta.RuntimeTypeModel model = ChatSerializer.Create();
    model.Serialize(stream, user);
    bs = stream.ToArray();
}
```

步骤 07 反序列化的代码如下:

```
ChatAPP.User user2;
using (System.IO.MemoryStream stream = new System.IO.MemoryStream(bs))
{
    ProtoBuf.Meta.RuntimeTypeModel model = ChatSerializer.Create();
    user2= (ChatAPP.User)model.Deserialize(stream,
                                           null, typeof(ChatAPP.User));
}
```

每次序列化、反序列化写这么多代码很麻烦,我们可以将它们写为泛型的函数,代码如下:

```
// 序列化
public static byte[] ProtoRuntimeSerialize<T>(T t)
{
    byte[] bs;
    using (System.IO.MemoryStream stream = new System.IO.MemoryStream())
    {
        ProtoBuf.Meta.RuntimeTypeModel model = ChatSerializer.Create();
        model.Serialize(stream, t);
    }
}
```

```

        bs = stream.ToArray();
    }
    return bs;
}
// 反序列化
public static T ProtoRuntimeDeserialize<T>(byte[] bs)
{
    T t = default(T);
    using (System.IO.MemoryStream stream = new System.IO.MemoryStream(bs))
    {
        ProtoBuf.Meta.RuntimeTypeModel model = ChatSerializer.Create();
        t = (T)model.Deserialize(stream, null, typeof(T));
    }
    return t;
}

```

现在，只需要简单调用这些函数就可以完成所有的序列化和反序列化工作了。

```

ChatAPP.User user = new ChatAPP.User();
user.name = "用户甲";
user.chat = "hello,world";

// 序列化
byte[] bs = ProtoRuntimeSerialize<ChatAPP.User>(user);
ChatAPP.User user2;
// 反序列化
user2 = ProtoRuntimeDeserialize<ChatAPP.User>(bs);

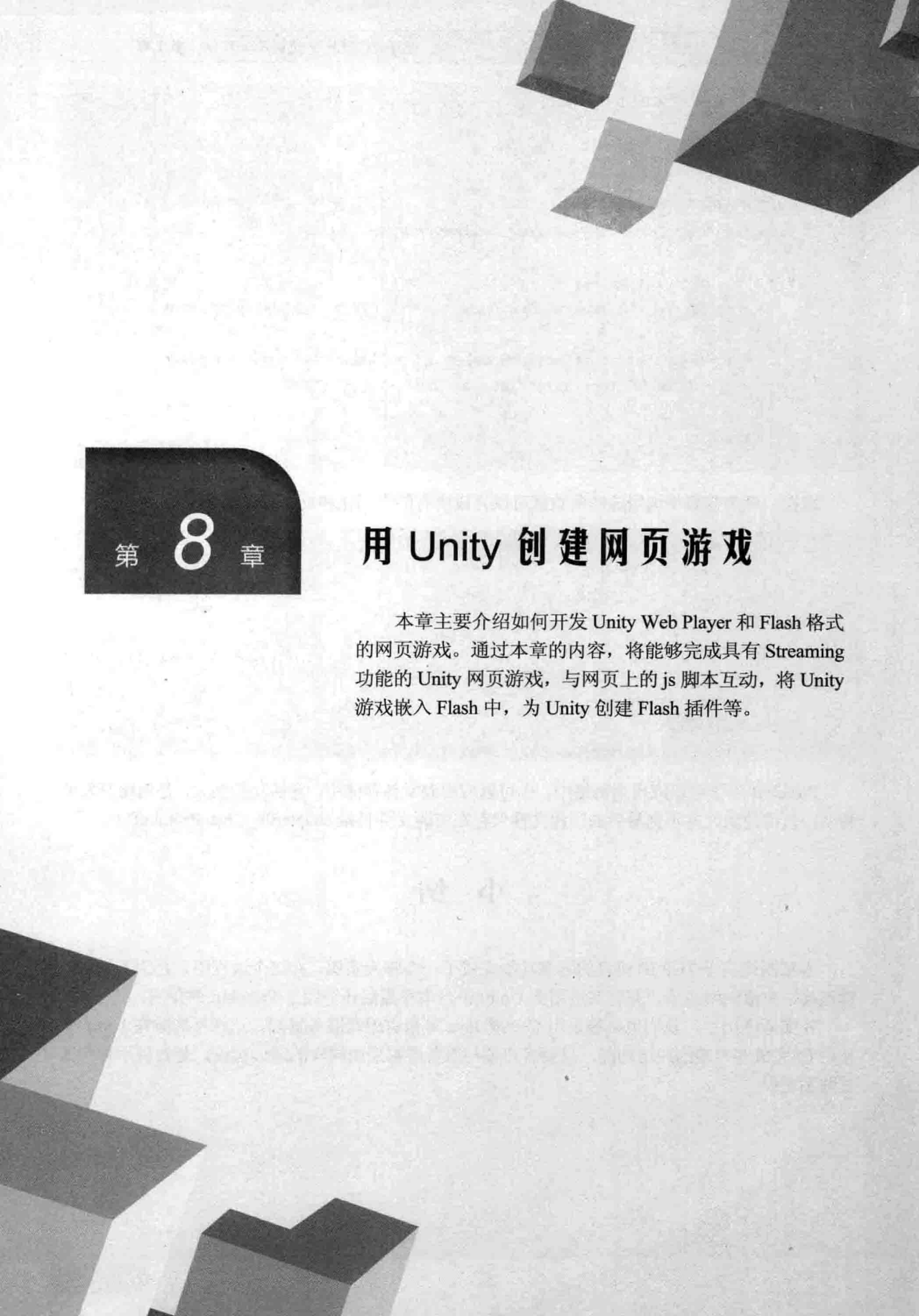
```

Protobuf 除了可以应用到网络中，也可以应用到文件存储中，它的功能强大，是网络开发的利器，值得使用。本示例最后的工程文件保存在资源文件目录 `chapter08_Chat_Protobuf` 中。

小 结

本章围绕基于 TCP/IP 协议的网络功能完成了一个聊天实例，在这个过程中，介绍了如何创建连接，完成网络通信，并将其应用到 Unity 中，本章最后还介绍了 Protobuf 的使用。

在实际应用中，我们也可能采用 C++ 或 Java 等语言编写服务器端，这都不影响在 Unity 中使用 C# 完成客户端的网络功能，只要客户端与服务器端采用同样的通信协议，是否使用相同语言并不重要。



第 8 章

用 Unity 创建网页游戏

本章主要介绍如何开发 Unity Web Player 和 Flash 格式的网页游戏。通过本章的内容，将能够完成具有 Streaming 功能的 Unity 网页游戏，与网页上的 js 脚本互动，将 Unity 游戏嵌入 Flash 中，为 Unity 创建 Flash 插件等。

8.1 网页游戏简介

随着网络的发展,越来越多的游戏可以直接运行在网页上。使用 Unity,可以将游戏发布为专门的 Unity 网页游戏格式,也可以发布为 Flash 格式,无论哪种格式,都可以运行在几乎所有的浏览器内。

Unity Web Player 是 Unity 游戏专用的网页游戏格式,不过用户需要安装 Unity 专用的网页插件才能运行游戏。Unity 官方发布的数据显示,已经有超过 1 亿的用户安装了 Unity Web Player 插件。

Flash 格式的网页游戏是最流行的网页游戏,全世界几乎所有的系统上都安装有 Flash 插件。Flash 游戏通常要通过 Flash 或 Flash Builder 来开发。使用 Unity,我们也可以开发 Flash 游戏,将 3D 游戏发布为 Flash 格式,这意味着可以在网页上立刻体验到用 Unity 开发的游戏,而不需要再安装额外的插件。

8.2 Unity Web 游戏

我们将继续第 2 章的太空射击游戏,将其发布为 Unity Web Player 格式,并上传到 Kongregate 网站上运营,同时会集成 Kongregate 的高分排行榜功能,让不同的用户比较分数。

8.2.1 Streaming 关卡

对于客户端游戏,用户必须拥有全部的游戏安装包才能开始游戏,而网页游戏则不需要。当用户在网页上打开游戏,实际上也是在下载游戏,这很像打开网页读取页面上的图片和文字等,用户不必等整个游戏下载完成,而是仅下载一部分即可开始游戏,在游戏中同时下载另一部分。

大部分 Unity 游戏都是由很多个关卡组成的,在网页游戏中,我们可以让用户一次只下载一个关卡即可开始游戏。如果一个游戏由 10 个关卡组成,每个关卡平均有 1MB,虽然整个游戏有 10MB,但用户只需要下载 1MB 左右的关卡就可以开始游戏了。

在第 2 章的太空射击游戏中一共有两个关卡,一个用于显示标题,另一个是实际游戏的关卡。在网页版中,我们将先读入标题关卡,然后再在标题关卡读入游戏关卡,读取关卡的时间根据网速和关卡容量的大小而定,但我们必须保证下一个关卡已经被完全下载才能开始读取关卡。

步骤 01 在资源文件目录打开 chapter02_ShootingGame 工程,这是在第 2 章完成的太空射击游戏工程。

步骤 02 在菜单栏选择【File】→【Build Settings】打开 Build Settings 窗口,选择【Web Player】,然后选择【Switch Platform】将当前工程转为 Web Player 工程。

步骤 03 打开 TitleScreen.cs 脚本, 修改代码:

```

using UnityEngine;
using System.Collections;

[AddComponentMenu("MyGame/TitleScreen")]
public class TitleScreen : MonoBehaviour
{
    // 下载标识符
    bool m_loadingFlag = false;

    // 下载完成比例
    float m_loadProgress = 0;

    // Update is called once per frame
    void Update()
    {
        //开始读取下一关
        if (m_loadingFlag)
        {
            //获取关卡下载完成比例
            m_loadProgress = Application.GetStreamProgressForLevel("level1");

            // 如果关卡已经下载完成
            if (Application.CanStreamedLevelBeLoaded("level1"))
            {
                // 读取下一关
                Application.LoadLevel("level1");
            }
        }
    }

    void OnGUI()
    {
        // 显示标题
        GUI.skin.label.fontSize = 48;
        GUI.skin.label.alignment = TextAnchor.LowerCenter;
        GUI.Label(new Rect(0, 30, Screen.width, 100), "SPACE WAR");

        // 显示下载完成比例
        if (m_loadingFlag)
        {
            GUI.skin.label.fontSize = 30;

```



```

        // 显示下载比例
        GUI.Label( new Rect( 0, Screen.height * 0.5f - 15, Screen.width, 40 ), "Loading
" +(int) (m_loadProgress*100)+"%");

        // 锁定开始游戏按钮
        GUI.enabled = false;
    }

    // 开始游戏按钮
    if (GUI.Button(new Rect(Screen.width * 0.5f - 100, Screen.height * 0.7f, 200, 30),
"Start Game"))
    {
        // 开始读取下一关
        m_loadingFlag = true;
    }

    GUI.enabled = true;
}
}

```

`m_loadingFlag` 属性用来标识是否开始读取下一关，当单击开始游戏按钮时，这个标识被设为 `true`。

`GetStreamProgressForLevel` 函数用来获得关卡的下载完成度，0 为最小值，1 表示下载完成，在这里我们将其值返回给 `m_loadProgress` 属性。

`CanStreamedLevelBeLoaded` 函数用来判断是否可以打开下一个关卡，当返回值为 `true` 时，表示关卡下载完成，这时才可以打开下一个关卡。

在下载关卡的同时，为了使玩家清楚下载的完成情况，我们在 `OnGUI` 中设了一段文字，表示下载的完成比例，如图 8-1 所示。



图 8-1 显示下载完成比例

步骤 04 打开 `Build Settings` 窗口，选中 `Streamed`，游戏才可以分段下载。

步骤 05 在 `Resolution and Presentation` 中设置网页游戏窗口的大小，如图 8-2 所示。

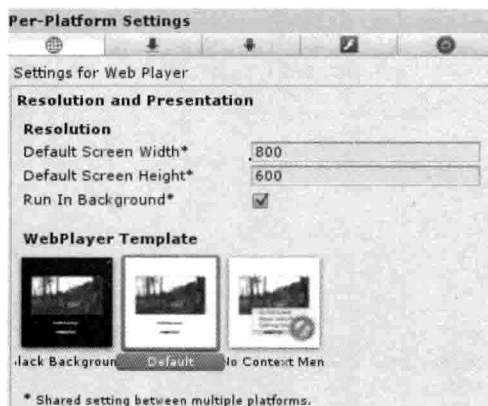


图 8-2 设置窗口大小

步骤 06 在 Other Settings 中将 First Streamed Level 设为 0，如图 8-3 所示。

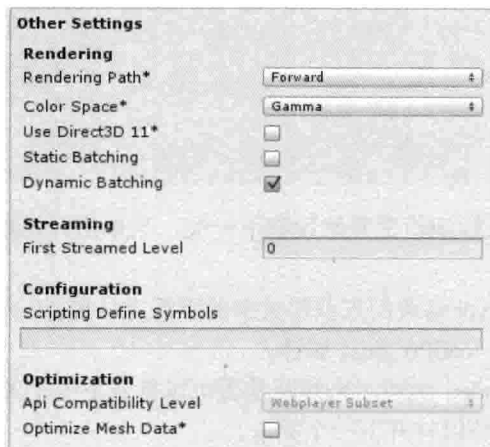


图 8-3 其他设置

8.2.2 上传游戏到 Kongregate

发布 Unity Web Player 网页游戏的最快速途径莫过于将游戏上传到 Kongregate (<http://www.kongregate.com/>) 网站上，那里为 Unity 网页游戏设置了专区，上面的大部分用户都安装过 Unity Web Player 插件。接下来，我们就看一下如何将游戏发布到 Kongregate。

步骤 01 在 Kongregate 网站上免费注册一个账号，然后在页面上选择【GAMES】，再选择【UPLOAD A GAME】，如图 8-4 所示。

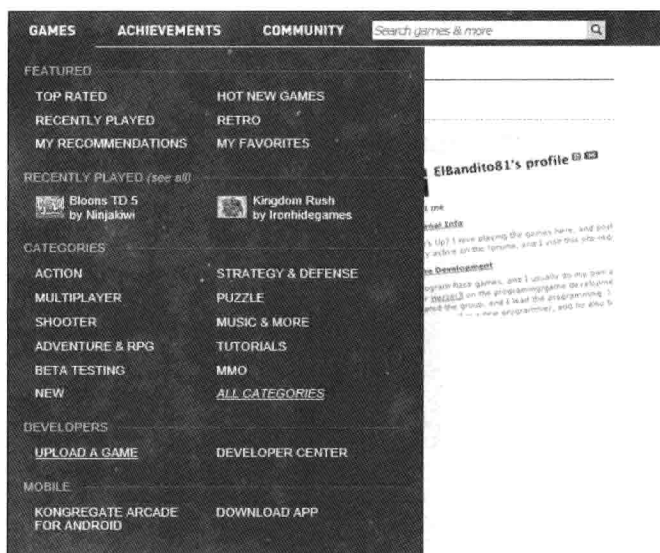


图 8-4 选择上传游戏

步骤 02 在接下来的页面填写游戏名称和相关信息，然后选择【Continue】，如图 8-5 所示。

 A screenshot of the 'Upload your game to share with the world!' form on Kongregate. The form is titled 'Upload your game to share with the world!' and has a progress indicator showing 'STEP: 1 Game Info'. The form includes the following fields:

- Title:** A text input field containing 'Space War'.
- Target Device:** Two radio button options: 'Computers (Playable with a keyboard & mouse)' (selected) and 'Mobile device (Lower power processor, touchscreen)'.
- Game Description:** A text area with the placeholder text 'This is just a game for test.' and a 'formatting guide' link.
- Game Instructions:** A text area with the placeholder text 'Use Up/Down/Left/Right Arrow control the ship, left mouse to fire.'
- Category:** A dropdown menu currently showing 'Shooter'.
- Collaborators:** A text area with the placeholder text 'Add up to three users who collaborated with you on this game.'
- API Callback URL:** A text input field.

 A 'Continue' button is located at the bottom right of the form.

图 8-5 游戏信息

步骤 03 选择【浏览】，选择需要上传的.unity3d 文件，然后设置游戏窗口的尺寸，并上传一张 250 × 200 像素的游戏图标，如图 8-6 所示。



图 8-6 上传游戏

步骤 04 在 Statistics API 中选择【Add a statistic】添加统计信息。在 Statistic name 中填入统计信息名称，这个名称会用到 Unity 中，与最高得分关联。将统计类型设为 Max Type，表示数值大的分数排在前列。选中【Display in leaderboards】会显示排行榜，最后选择【Save】保存，如图 8-7 所示。

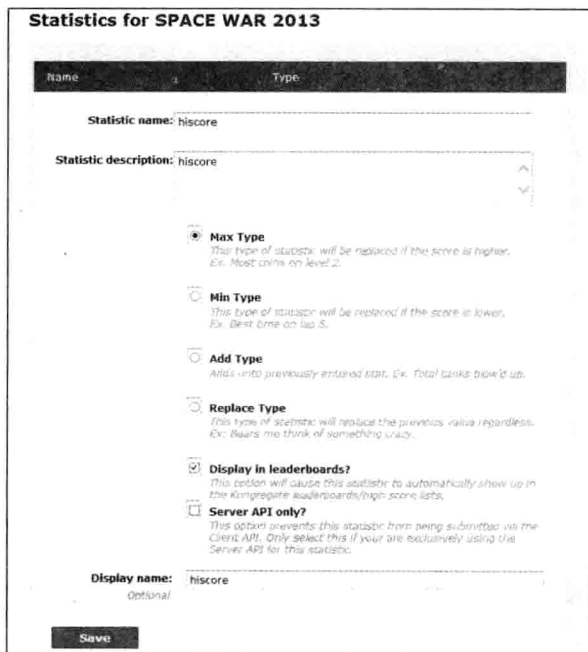


图 8-7 设置统计信息

步骤 05 在 Tags 中选择【Add a tag】可以为游戏添加更多与游戏相关的关键字，利于玩家在搜索时更容易找到游戏。

步骤 06 最后选择页面最下方的【Upload】上传游戏。

步骤 07 当游戏上传完成后会进入预览页面，我们可以在这里先试玩一下游戏，如图 8-8 所

示。如果觉得游戏没有问题，选择【publish】发布游戏，否则选择【edit】继续修改前面的信息，或者选择【Upload a New version】上传一个新版本。

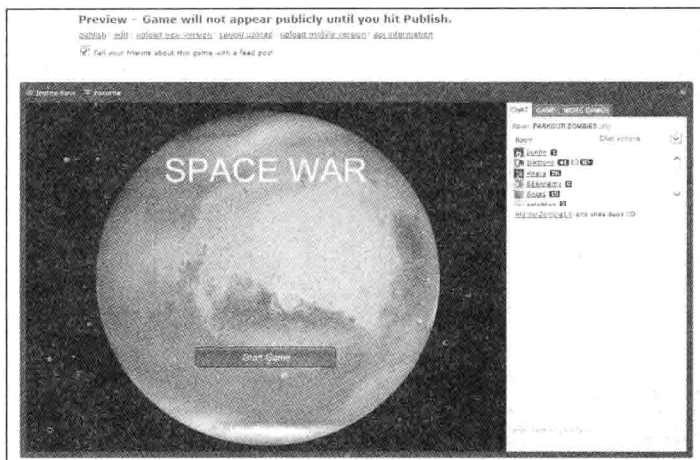


图 8-8 在 Kongregate 的网页上预览游戏

8.2.3 与网页通信

Kongregate 网站的一个特色是可以为每一个游戏设置一个高分榜，然后让玩家去比较分数和排名。前面我们在 Statistics API 中已经设置了一个最高分的排行榜，接下来我们将在游戏中添加代码使 Unity 网页游戏与 Kongregate 页面上的 javascript 脚本通信，上传分数。

步骤 01 新建 Kongregate.cs 脚本：

```
using UnityEngine;
using System.Collections;

public class Kongregate : MonoBehaviour
{
    static bool m_isinit = false;

    // 初始化
    public static void Init()
    {
        #if UNITY_WEBPLAYER

            if (m_isinit)
                return;

            m_isinit = true;

            string javaCode = @"
                // Load the API
```



```

        var kongregate;
        kongregateAPI.loadAPI(onComplete);

        // Callback function
        function onComplete(){
            // Set the global kongregate API object
            kongregate = kongregateAPI.getAPI();
        }
    ";
    Application.ExternalEval(javaCode);
#endif
}

// 上传分数或其他统计数据
public static void Submit(string stat, int score)
{
#if UNITY_WEBPLAYER

    if (!m_isinit)
        return;

    Application.ExternalCall("kongregate.stats.submit", stat, score);
#endif
}
}

```

`m_isinit` 属性用来标识 Kongregate API 是否已经被初始化, Kongregate 初始化代码只能被执行一次, 否则会出现错误。

`Init` 函数用来初始化 Kongregate 的 API, 其中 `javaCode` 字符串是运行在网页中的 javascript 代码, 用来获取 Kongregate 的 API, 我们使用 Unity 的 `Application.ExternalEval` 执行这段代码, 其效果与在网页上调用这段 javascript 代码完全一样。

`Application.ExternalCall` 是另一种调用网页脚本的形式, 其中 `kongregate.stats.submit` 是 Kongregate API 中的函数, 用来上传分数。在这里我们可以传入两个参数, 一个是统计信息的名称, 另一个是它的值。

`UNITY_WEBPLAYER` 是 Unity 预设的预处理标识符, 判断当前的版本是否是 Unity Web Player 版本, 这里的代码将只为 Unity 网页版使用。

步骤 02 打开 `TitleScreen.cs` 脚本, 在 `Start` 函数中初始化 Kongregate API。

```

void Start()
{
    Kongregate.Init();
}

```

步骤 03 打开 Player.cs 脚本, 添加一个 OnDisable 函数, 它将在 Player 对象被销毁时自动调用。我们在这个函数中上传当前游戏的得分:

```
void OnDisable()
{
    Kongregate.Submit("hiscore", GameManager.Instance.m_score);
}
```

步骤 04 重新 Build 并上传游戏, 运行一次游戏, 在上传记录后刷新网页, 如果没有错误, 页面将会发生变化, 在游戏窗口旁边会显示出一个 HIGH SCORES 排行榜。

8.2.4 在网页上记录积分

网页游戏不支持一般的 I/O 操作, 也就是说我们不能自定义一个文件保存在硬盘上存储网页游戏的记录, 好在 Unity 提供了一个叫 PlayerPrefs 的类, 用来保存游戏的预设项, 在网页游戏中, 也可以用来保存网页游戏的记录, 针对每个不同网址的网页游戏, 其存储文件的大小必须在 1MB 以内。

步骤 01 打开 TitleScreen.cs, 在 Start 函数中添加 PlayerPrefs.GetInt 函数获得最高分的记录, 如果记录不存在, 默认值为 0:

```
void Start()
{
    Kongregate.Init();

    GameManager.m_hiscore = PlayerPrefs.GetInt("space war score", 0);
}
```

步骤 02 打开 Player.cs, 在 OnDisable 函数中添加 PlayerPrefs.SetInt 函数保存当前的最高得分记录:

```
void OnDisable()
{
    Kongregate.Submit("hiscore", GameManager.Instance.m_score);

    PlayerPrefs.SetInt("space war score", GameManager.m_hiscore);
}
```

步骤 03 重新 build 并上传游戏, 运行一次游戏并刷新网页, 会发现上一次游戏保存的最高分被保存了下来。

8.2.5 自定义网页模板

默认的 Unity 网页游戏画面只是显示在一个空白的网页上。实际上, Unity 提供了 3 种预设

的网页模板, 分别是白、黑背景和一个限制使用鼠标右键的模板, 在实际应用中, 我们通常需要修改默认的网页页面, 如图 8-9 所示。

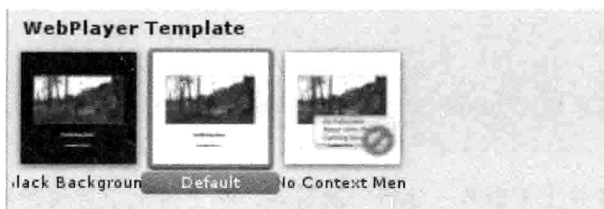


图 8-9 预设的网页模板

我们可以定制自定义的网页模板, 使默认的页面更符合需求, 这样就不用每次创建游戏都要去修改网页了。下面是一个例子:

- 步骤 01** 在游戏工程目录 Assets 文件夹下建一个名为 WebPlayerTemplates 的文件夹, 这个名称是特定的, 不能修改。
- 步骤 02** 在 WebPlayerTemplates 文件夹下面再创建一个任意名称的文件夹, 比如叫 MyWebGame, 这个文件夹即表示一个模板, 文件夹的名称即是模板的名称, 文件夹内用来保存网页上所需要的各种类型文件。
- 步骤 03** 使用如 Dreamweaver 之类的软件创建一个 index.html 文件置于模板文件夹内, 默认的.html 文件主要包括 head 和 body 两个部分。
- 步骤 04** 在 head 中添加读取 Unity 的脚本:

```
<script type="text/javascript" src="%UNITY_UNITYOBJECT_URL%"></script>
<script type="text/javascript">
<!--
function GetUnity() {
    if (typeof unityObject != "undefined") {
        return unityObject.getObjectById("unityPlayer");
    }
    return null;
}
if (typeof unityObject != "undefined") {
    unityObject.embedUnity("unityPlayer",          "%UNITY_WEB_PATH%",          %UNITY_WIDTH%,
%UNITY_HEIGHT%);
}
-->
</script>
```

%UNITY_UNITYOBJECT_URL% 是一个模板标识符, 在实际的网页中会指向类似 http://webplayer.unity3d.com/download_webplayer-3.x/3.0/uo/UnityObject.js 这样的地址去下载 UnityObject.js 文件, 但如果是在本地运行 Unity 网页游戏, 有时可能会找不到 UnityObject.js 这个文件。我们可以先从 Unity 官方服务器上手动下载 UnityObject.js 文件, 然后将其置于模板文件夹内, 它会与.html 文件一起发布。

修改代码如下，从本地调用 UnityObject.js:

```
<script type="text/javascript" src="UnityObject.js"></script>
```

步骤 05 在 body 中添加如下脚本显示游戏画面，在实际的项目中，我们可以将其置于网页页面的某个位置。

```
<div id="unityPlayer">
```

```
</div>
```

步骤 06 运行 Unity 网页游戏的前提是用户必须安装有 Unity Web Player 插件，如果用户还没有安装插件，在网页上将不能显示出游戏画面，这时我们需要将游戏显示画面替换为提醒用户安装插件的画面，只需要添加一个名为 missing 的 div 即可，如下所示：

```
<div id="unityPlayer">
  <div class="missing">
    <a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now!">
    </a>
  </div>
</div>
```

默认可以让用户直接到 Unity 官网上下载插件，也可以将 <http://unity3d.com/webplayer/> 替换为我们自己的网站地址去下载插件。

步骤 07 下面是一个完整的模板示例代码，其中标记 style 中的内容多数是和 div 网页布局相关的。

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/
xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>

<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title> %UNITY_WEB_NAME%</title>

<script type="text/javascript" src="UnityObject.js"></script>
<script type="text/javascript">
<!--
function GetUnity() {
if (typeof unityObject != "undefined") {
return unityObject.getObjectById("unityPlayer");
}
}
```

```

return null;
}

if (typeof unityObject != "undefined") {
unityObject.embedUnity("unityPlayer", "%UNITY_WEB_PATH%", %UNITY_WIDTH%, %UNITY_HEIGHT%);
}
-->
</script>

<style type="text/css">
<!--
body {
font-family: Helvetica, Verdana, Arial, sans-serif;
background-color: #066;
color: black;
text-align: center;
}
a:link, a:visited {
color: #000;
}
a:active, a:hover {
color: #666;
}
p.header {
font-size: small;
}
p.header span {
font-weight: bold;
}
p.footer {
font-size: x-small;
}
div.content {
margin: auto;
width: %UNITY_WIDTH%px;
}
div.missing {
margin: auto;
position: relative;
top: 50%;
width: 193px;
}
div.missing a {
height: 63px;

```



```

position: relative;
top: -31px;
}
div.missing img {
border-width: 0px;
}
div#unityPlayer {
cursor: default;
height: %UNITY_HEIGHT%px;
width: %UNITY_WIDTH%px;
}
-->
</style>
</head>

<body>
<p class="header">%UNITY_WEB_NAME%</p>%UNITY_BETA_WARNING%
<div class="content">
<div id="unityPlayer">
<div class="missing">
<a href="http://unity3d.com/webplayer/" title="Unity Web Player. Install now!">
</a>
</div>
</div>
</div>
</div>
<p class="footer">&laquo; created with <a href="http://unity3d.com/unity/" title="Go to
unity3d.com">Unity</a> &raquo;</p>

</body>
</html>

```

步骤 08 在模板文件夹内放入一张大小为 128×128 , 格式为 .png 的图片, 并命名为 thumbnail 作为模板的图标, 如图 8-10 所示。



图 8-10 自定义模板图标

最后, 使用模板编译网页游戏, 其页面与默认页面略有不同, 我们可以在这个基础上自由修改模板, 定制需要的样式。

8.2.6 自定义启动画面

在启动 Unity 网页游戏的时候,默认会显示出 Unity 的图标,同时会显示下载进度,当最初的下载完成,才会进入第一个关卡,如图 8-11 所示。



图 8-11 默认的 Unity 网页游戏启动画面

我们可以自定义 Unity 网页游戏的启动画面,但前提是游戏必须是用 Unity Pro 版本创建的。

步骤 01 打开前面创建的网页模板.html 文件,修改 embedUnity 部分的代码如下:

```
var params = {
    backgroundcolor: "FFFFFF",
    bordercolor: "000000",
    textcolor: "FFFFFF",
    logoimage: "MyLogo.png",
    progressbarimage: "MyProgressBar.png",
    progressframeimage: "MyProgressFrame.png",
    disableContextMenu: true
};

unityObject.embedUnity("unityPlayer", "%UNITY_WEB_PATH%", %UNITY_WIDTH%, %UNITY_HEIGHT%,
params);
```

logoimage 作为启动时的背景图片; Progressbarimage 是一张进度条图片; Progressframeimage 是进度条后面的背景框,它的大小最好与进度条图片一致; disableContextMenu 决定是否屏蔽鼠标右键功能,设为 true 即不能使用右键快捷菜单。

步骤 02 在网页模板文件夹中分别添加背景和进度条图片,名称一定要与模板中定义的一致,如图 8-12 所示。

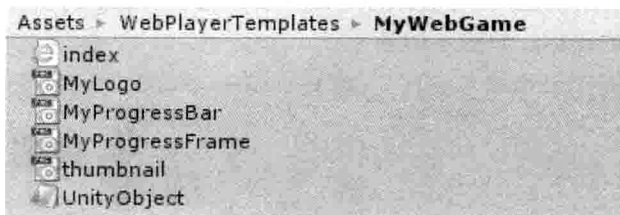


图 8-12 自定义图片

步骤 03 重新编译游戏,自定义的启动页面如图 8-13 所示。

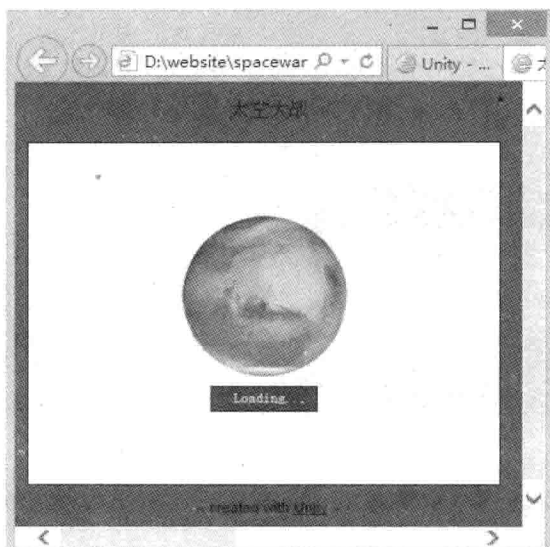


图 8-13 自定义网页游戏启动页面

关于网页的页面，不一定都需要在模板中定义和修改，也可以直接修改最终的.html 文件，方法与修改模板是一样的，Unity 也支持.php 文件，最终的选择应当根据实际情况。本节的示例工程文件保存在资源文件目录\chapter08_ShootingGameWeb 中。

8.3 Flash 游戏

Unity 从版本 4.0 开始支持 Flash 平台，这是一个巨大的更新，可以使 Unity 游戏扩展到更大的市场范围供用户体验。

8.3.1 软件安装

为了能够运行 Unity 创建的 Flash 游戏，首先要确定系统中安装了 Adobe Flash Player 11.2 或更新版本的插件。

在从 Unity 导出 Flash 游戏的时候，还需要安装 JRE (Java SE Runtime Environment)，可以到 <http://www.oracle.com/technetwork/java/javase/install-windows-141940.html> 免费下载。

最后，编写 Flash 脚本，需要安装编写 Action Script3 代码的编辑器，如 Flash Builder、PowerFlasher FDT、FlashDeveloper 等。

8.3.2 导出 Flash 游戏

将 Unity 游戏导出为 Flash 非常简单，只需要几个步骤：

步骤 01 在菜单栏选择【File】→【Build Settings】打开 Build Settings 窗口，选择【Flash Player】，然后选择【Switch Platform】将当前工程转为 Flash 工程，如图 8-14 所示。



图 8-14 转为 Flash 工程

- 步骤 02** 在 Build Settings 窗口中设置 Target Player，这个设置对应的是 Flash 插件的版本，通常保持默认即可。
- 步骤 03** 如果游戏还处于测试阶段，建议选中 Development Build，这将使导出 Flash 的过程快上很多倍，但游戏将不会压缩，不适合最终发布。
- 步骤 04** 最后选择【Build】导出 Flash 游戏，会得到 4 个文件：

- .swf 文件是 Flash 游戏文件。
- .html 文件是一个用来显示 Flash 游戏的网页文件，在实际项目中通常要修改这个文件。
- swfobject.js 用来在网页上检查是否安装有 Flash 插件并整合 .swf 文件到网页中。
- embeddingapi.swc 用来在 Flash 工程中读取 Unity 导出的 Flash 游戏。

需要注意的是，Unity 的一些功能并不支持 Flash，这可能会导致在导出 Flash 的时候产生错误，不过随着 Unity 版本的升级，情况会得到逐渐改善。

.NET 语言中的一些特殊类或函数可能无法转为 AS3 代码，这种情况也会导致错误。在导出 Flash 后，到游戏工程目录下的 \Temp\StagingArea\Data\ConvertedDotNetCode 中可以找到被转换的 .net 脚本，它们现在都保存为 .as 格式，这时可以根据提示查找出现错误的原因。

8.3.3 调试 Flash 游戏

运行 Unity 导出的 Flash 游戏经常会遇到一些不会在 Unity 编辑器中出现的错误，为了能查找到出现问题的原因，我们需要到 Flash 的 Log 文件中查看日志。

Unity 的 Debug.Log 函数可以在控制台输出调试信息，同时也可以保存到 Flash 的 Log 文件中，使用 Flash Log 文件的方法如下：

- 步骤 01** 为了能够调试 Flash 游戏，我们需要安装 Debug 版本的 Flash Player，到 Adobe 的官方网站 <http://www.adobe.com/support/flashplayer/downloads.html> 可免费下载它。
- 步骤 02** 创建一个 mm.cfg 文件，并将其保存在相应目录内：

```
Macintosh OS X    /Library/Application Support/Macromedia/mm.cfg
XP                C:\Documents and Settings\username\mm.cfg
```

```
Windows Vista/Win7    C:\Users\username\mm.cfg
Linux                  /home/username/mm.cfg
```

步骤 03 在 mm.cfg 内输入:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

步骤 04 运行 Flash 游戏, 在相应的目录内找到 Log 文件:

```
Macintosh OS X    /Users/username/Library/Preferences/Macromedia/Flash Player/Logs/
XP    C:\Documents and Settings\username\Application Data\Macromedia\Flash Player\Logs
Windows Vista/Win7    C:\Users\username\AppData\Roaming\Macromedia\Flash Player\Logs
Linux    /home/username/.macromedia/Flash_Player/Logs/
```

8.3.4 从 Flash 工程读取 Unity 导出的 Flash 游戏

Unity 导出的 Flash 游戏可以被导入到另一个 Flash 工程中, 下面是一个简单的实例:

步骤 01 创建一个 Unity 游戏工程, 或者用 Unity 打开资源文件目录下的 chapter08_Embedding 工程, 将其导出为 Flash。

步骤 02 使用 Flash Builder 创建一个 Action Script 工程, 将从 Unity 中导出的.swf 和 embeddingapi.swc 文件复制粘贴到 Action Script 工程中, 然后再从资源文件目录 /rawdata/2d 下复制粘贴 fish.png 到工程中, 这是一张鱼的图片, 也可以使用其他任何图片, 如图 8-15 所示。

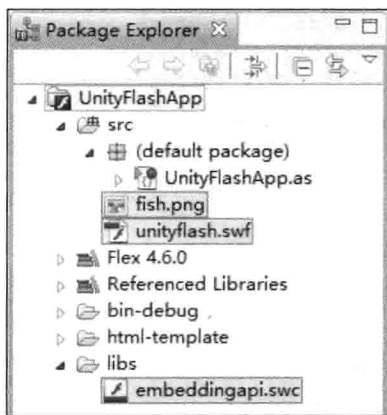


图 8-15 在 Flash 工程中导入 Unity Flash 文件

步骤 03 在 Flash Builder 菜单栏选择【Project】→【Properties】, 选择 ActionScript Build Path, 在 Library path 中选择【Add SWC】, 然后选择 embeddingapi.swc 文件, 如图 8-16 所示。

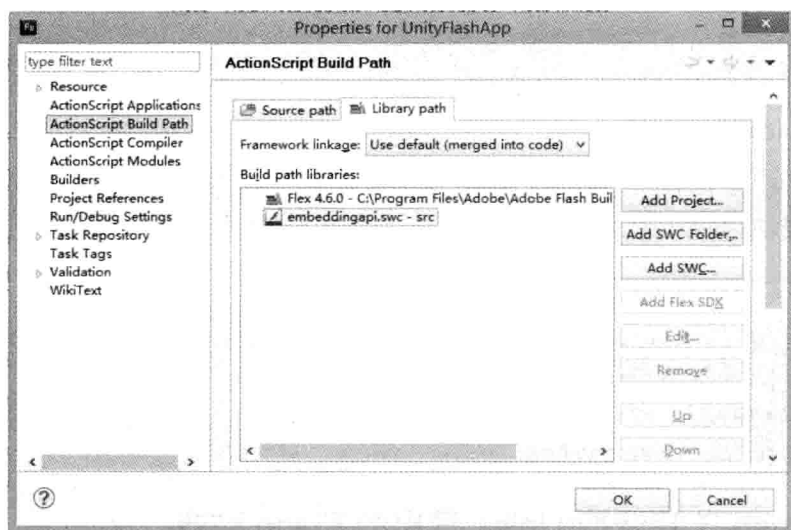


图 8-16 使用.swc 文件

embeddingapi.swc 是从 Unity 中导出的 Flash 库文件, 包括一些在 Flash 工程中调用 Unity 的 API。

- 步骤 04** 在 Package Explorer 窗口的 html-template 中选择 index.template.html 文件, 单击右键选择【Open With】→【Text Editor】, 找到和 params 相关的语句, 添加 params.wmode="direct";语句, 只有将 wmode 设为 direct 才能调用 Unity 的资源, 如图 8-17 所示。

```
var params = {};
params.quality = "high";
params.bgcolor = "${bgcolor}";
params.allowscriptaccess = "sameDomain";
params.allowfullscreen = "true";
params.wmode = "direct";
```

图 8-17 设置 wmode

- 步骤 05** 打开.as 文件, 添加和修改代码如下:

```
package
{
    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.net.URLRequest;
    import com.unity.IUnityContent;
    import com.unity.IUnityContentHost;
    import com.unity.UnityContentLoader;
    import com.unity.UnityLoaderParams;
    import flash.events.ProgressEvent;
```

```

public class UnityFlashApp extends Sprite implements IUnityContentHost
{
    protected var fishloader:Loader;
    protected var unityLoader:UnityContentLoader;

    public function UnityFlashApp()
    {
        // 读取 Flash 2D 图片
        fishloader = new Loader();
        var request:URLRequest=new URLRequest("fish.png");
        fishloader.load(request);
        fishloader.contentLoaderInfo.addEventListener(Event.COMPLETE,
onContentLoaded);

        // 读取 Unity 3D 资源
        LoadUnityContent();
    }

    protected function LoadUnityContent():void
    {
        var params:UnityLoaderParams = new UnityLoaderParams(false, 800, 600,
false);
        unityLoader = new UnityContentLoader("unityflash.swf", this, params,
false);
        unityLoader.contentLoaderInfo.addEventListener(ProgressEvent.PROGRESS,
onUnityContentLoaderProgress);
        unityLoader.contentLoaderInfo.addEventListener(Event.COMPLETE,
onUnityContentLoaderComplete);
        unityLoader.loadUnity();
    }

    private function onUnityContentLoaderProgress(event:ProgressEvent):void
    {
        //Respond to load progress
        //trace( "Loading: "+ event.bytesLoaded/event.bytesTotal );
    }

    private function onUnityContentLoaderComplete(event:Event):void
    {
        addChild(unityLoader);
        unityLoader.unityContent.setContentHost(this);
    }

    public function unityInitStart():void

```

```

    {
        //Unity engine started
    }

    public function unityInitComplete():void
    {
        unityLoader.unityContent.sendMessage("Avatar", "ChangeSpeed",
{responder:this});
    }

    private function onContentLoaded( event:Event ):void
    {
        this.addChild(fishloader);
    }
}
}

```

在这段代码中, 首先使用标准的 AS3 语句读入 fish.png 图片, 并将它显示在屏幕上。然后创建一个 UnityContentLoader, 读入 Unity 创建的 unityflash.swf。

unityInitComplete 是一个回调函数, 当第一个 Unity 场景被加载完成后, 这个函数会被触发, 在这里我们使用 sendMessage 函数向 Unity 发一个消息, 接收者是 Unity Flash 中的一个名叫 Avatar 的游戏体实例, 并调用它的 ChangeSpeed 函数, 在这个示例中, ChangeSpeed 函数会改变 3D 角色的旋转速度。

按 F11 键调试游戏, 背景是 Unity 的 3D 角色, 前景是使用 AS3 读入的一条鱼的图片, 3D 的画面会永远显示在 Flash 的 2D 画面之后, 如图 8-18 所示。

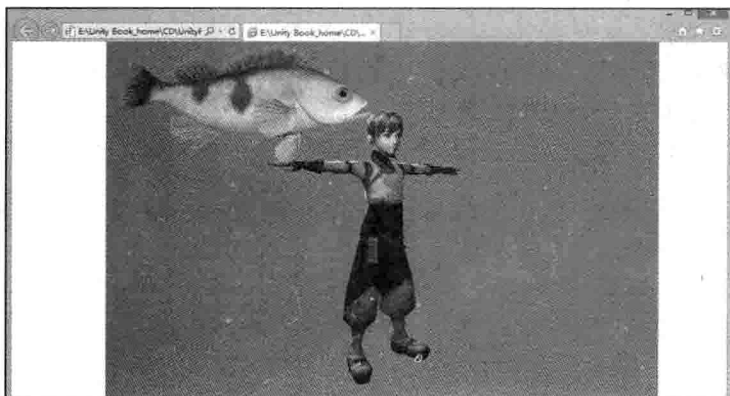


图 8-18 嵌入 Flash 工程中的 Unity Flash

本示例工程文件保存在资源文件目录/chapter08_Embedding 和/chapter08_UnityFlashProject 内, 其中 chapter08_UnityFlashProject 是一个 Flash Builder 工程。

现在有一个问题是, Unity 导出的.swf 文件与当前 Flash 工程的.swf 文件是两个独立文件, 但很多网站只允许上传一个单独的.swf 文件, 我们可以使用 Flash 提供的 Embed 标识符将 Unity 的.swf 文件与当前 Flash 工程的.swf 文件合二为一。使用这种方法, 也可以自定义 Unity Flash 的

启动画面。

修改.as 文件如下：

```
package
{
    import flash.display.Loader;
    import flash.display.Sprite;
    import flash.display.DisplayObject;
    import flash.events.Event;
    import flash.net.URLRequest;
    import com.unity.IUnityContent;
    import com.unity.IUnityContentHost;
    import com.unity.UnityContentLoader;
    import com.unity.UnityLoaderParams;
    import flash.events.ProgressEvent;
    import mx.core.BitmapAsset;

    [SWF(width='800', height='600', backgroundColor='#FFFFFF', frameRate='60')]
    public class UnityFlashApp extends Sprite implements IUnityContentHost
    {
        [Embed(source="unityflash.swf", mimeType="application/octet-stream")]
        private static const unityflash : Class;

        [Embed(source="fish.png")]
        private static const fish : Class;

        protected var unityContent : IUnityContent;
        protected var unityLoader:Loader;
        private var childSWF : DisplayObject;

        protected var fishloader:BitmapAsset;

        public function UnityFlashApp()
        {
            // 读取 Flash 2D 图片
            fishloader = new fish() as BitmapAsset;
            this.addChild(fishloader);

            // 读取 Unity 3D 资源
            LoadUnityContent();
        }

        protected function LoadUnityContent():void
        {
```



```

        unityLoader = new Loader();
        unityLoader.contentLoaderInfo.addEventListener(Event.COMPLETE,
onUnityContentLoaderComplete);
        unityLoader.loadBytes(new unityflash());
    }

    private function onUnityContentLoaderProgress(event:ProgressEvent):void
    {
        //Respond to load progress
        //trace( "Loading: "+ event.bytesLoaded/event.bytesTotal );
    }

    private function onUnityContentLoaderComplete(event:Event):void
    {
        unityLoader.contentLoaderInfo.removeEventListener(Event.COMPLETE,
onUnityContentLoaderComplete);
        childSWF = addChild(unityLoader.content);
        childSWF.visible=false;
        unityContent = unityLoader.content as IUnityContent;
        unityContent.setContentHost(this);
    }

    public function unityInitStart():void
    {
        //Unity engine started
    }

    public function unityInitComplete():void
    {
        unityContent.sendMessage("Avatar", "ChangeSpeed", {responder:this});

        childSWF.visible=true;
    }
}

```

这段代码与之前的代码相比有很大变化, 主要是现在所有资源都整合到一个.swf 文件中, 读取资源的方式发生了改变。最后使用了一个 visible 设置来隐藏默认 Unity 的读取画面, 当 Unity 资源被完全读入后再将其显示出来。

本示例工程文件保存在资源文件目录/chapter08_UnityFlashEmbed 内, 它也是一个 Flash Builder 工程。

8.3.5 在 Unity 内调用 AS3 代码

很多时候,可能需要在 Unity 内调用 Flash 的 AS3 代码,比如 Flash 不支持 Unity 的 Socket,这样就只能在 Flash 中完成关于 Socket 工作,然后将代码拿到 Unity 中来使用,不过在 Unity 编辑器中并不能预览 AS3 代码的效果,只有等游戏被导出为 Flash 后才能实现 AS3 的功能,所以在开发中最好用 Unity 本身的代码暂时代替 AS3 的代码,下面是一个简单的示例:

步骤 01 在 Flash 编辑器中新建一个.as 脚本,这里命名为 CallFlash.cs。

步骤 02 在 CallFlash.cs 中输入如下代码:

```
package
{
    public class CallFlash
    {
        public static var _instance : CallFlash;

        public var _id : int;
        public var _name : String;
        public var _flag : Boolean;

        public function Init() : void
        {
            _id=100;
            _name="Flash";
            _flag=true;

            trace("Init Flash Function");
        }

        public static function GetCallFlash() : CallFlash
        {
            if ( _instance==null )
            {
                _instance = new CallFlash();
            }

            return _instance;
        }

        public static function aStaticFunctionWithParams(a : int) : void
        {
            GetCallFlash()._id = a;
        }
    }
}
```

```

        public static function aStaticFunctionWithReturnType() : String
        {
            return "Call Flash ActionScript";
        }
    }
}

```

在这个类中, `_id`、`_name`、`_flag` 分别是 3 个不同类型的属性, `_instance` 是实例的静态句柄; `Init` 函数用来做一些初始化工作; `GetCallFlash` 函数将返回一个 `CallFlash` 实例的句柄。

`aStaticFunctionWithParams` 是一个静态函数, 为 `_id` 赋值。

`aStaticFunctionWithReturnType` 是一个带返回类型的静态函数, 返回一个字符串。

步骤 03 新建一个 Unity 工程, 在 `Assets` 文件夹内创建一个名为 `ActionScript` 的文件夹, 并将 `CallFlash.as` 脚本放到这个文件夹内。



提示

保存 `.as` 文件的文件夹名称必须是 `ActionScript`。

步骤 04 在 Unity 内新建一个 `CallFlash.cs` 脚本, 输入代码如下:

```

using UnityEngine;
using System.Collections;

[NotRenamed]
[NotConverted]
public class CallFlash{

    [NotRenamed]
    public static CallFlash _instance;

    [NotRenamed]
    public int _id;

    [NotRenamed]
    public string _name;

    [NotRenamed]
    public bool _flag;

    [NotRenamed]
    public void Init()
    {
        _id=100;
        _name="Flash";
    }
}

```

```

        _flag=true;

        Debug.Log("Init Unity Function");
    }

    [NotRenamed]
    public static CallFlash GetCallFlash()
    {
        if ( _instance==null )
        {
            _instance = new CallFlash();
        }

        return _instance;
    }

    [NotRenamed]
    public static void aStaticFunctionWithParams( int a )
    {
        GetCallFlash()._id = a;
    }

    [NotRenamed]
    public static string aStaticFunctionWithReturnType()
    {
        return "Call Unity C#";
    }
}

```

可以注意到，这个类的结构与 Flash 版本的完全一致，只是换成了 C# 的语法。在这个类中有两个特别的属性，分别是 NotRenamed 和 NotConverted，NotRenamed 是为了防止名称被改变，NotConverted 是为了阻止在导出 Flash 的时候将 .cs 代码转为 .as 代码。

下面，我们将调用 ClassFlash 这个类的实例，当在 Unity 编辑器或非 Flash 平台中的时候，我们调用的是 C# 的代码，当导出为 Flash 的时候，会调用 AS3 的代码。

步骤 05 新建一个 AppManager.cs 脚本，输入代码如下：

```

using UnityEngine;
using System.Collections;

public class AppManager : MonoBehaviour {

    string m_data;

    // Use this for initialization

```

```

void Start () {

    CallFlash flashas = CallFlash.GetCallFlash();
    //CallFlash flashas= new CallFlash();
    flashas.Init();

    CallFlash.aStaticFunctionWithParams(10);
    m_data = CallFlash.aStaticFunctionWithReturnType();

}

void OnGUI()
{
    GUI.Label( new Rect(10, 10, 200, 40), m_data );
}
}

```

在这段代码中，我们创建了一个 CallFlash 的实例，将其初始化，并在 UI 中显示 aStaticFunctionWithReturnType 函数返回给 m_data 的字符。将 AppManager.cs 指定给摄像机，运行程序，屏幕上的显示结果如图 8-19 所示。

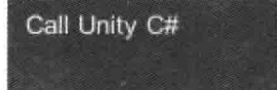


图 8-19 在 Unity 编辑器中的执行结果

步骤 06 无须做任何改动，导出为 Flash 并运行，结果如图 8-20 所示。

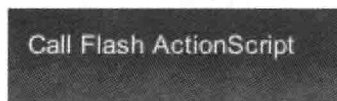


图 8-20 在 Flash 中的执行结果



提示 本节的示例文件将保存在资源文件目录\chapter08_UnityFlashPlugin 内。

8.3.6 Flash 版本的太空射击游戏

本节我们将把前面创建的 Unity 网页版太空射击游戏转为 Flash 工程并导出为 Flash 上传到 Kongregate 上。我们将使用 AS3 调用 Kongregate API，然后将这个脚本导入到 Unity 中，使 Unity 导出的 Flash 也能像 Unity 网页游戏一样在 Kongregate 上传分数。

步骤 01 使用 Flash 编辑器创建脚本 Kongregate.as:

```
package
```

```

{
    import flash.display.Loader;
    import flash.display.LoaderInfo;
    import flash.display.Sprite;
    import flash.events.Event;
    import flash.net.URLRequest;
    import flash.system.Security;
    import com.unity.UnityNative;

    public class Kongregate
    {
        // FlashVars API 路径
        static var paramObj:Object;

        // API 路径. "shadow" API 将帮助本地测试.
        static var apiPath:String;

        // Kongregate API 句柄
        static var _kongregate:*;

        // 初始化
        public static function Init():void
        {
            paramObj=LoaderInfo(UnityNative.stage.loaderInfo).parameters;

            apiPath = paramObj.kongregate_api_path ||
                'http://www.kongregate.com/flash/API_AS3_Local.swf';

            // 允许 API 与 SWF 通信
            Security.allowDomain(apiPath);

            // 读取 Kongregate API
            var request:URLRequest = new URLRequest(apiPath);
            var loader:Loader = new Loader();
            loader.contentLoaderInfo.addEventListener(Event.COMPLETE, loadComplete);
            loader.load(request);
            UnityNative.stage.addChild(loader);
        }

        // 完成 API 的下载
        static function loadComplete(event:Event):void
        {
            // 保存 Kongregate API
            _kongregate = event.target.content;
        }
    }
}

```



```

        // 连接服务器
        _kongregate.services.connect();
    }

    // 提交统计数据
    public static function Submit( stat: String , score: int ) : void
    {
        if ( _kongregate==null )
            return;

        _kongregate.stats.submit(stat,score);
    }
}
}

```

这段代码在 Flash 编辑器中并不能直接编译通过,原因是导入了 com.unity.UnityNative 这个包,这个包在 Flash 中是看不到的,但稍后我们会将 Kongregate.as 导入到 Unity 中,那时就没问题了。

Init 函数主要是从 Kongregate 服务器上获得 API,并在 loadComplete 函数中将 API 赋予 _kongregate 属性。

Submit 函数负责上传分数。

步骤 02 打开资源文件目录/chapter08_ShootingGameWeb 下的工程,这是 Unity 网页版的太空射击游戏的工程文件。

步骤 03 将 Kongregate.as 复制粘贴到 ActionScript 文件夹中。

步骤 04 修改 Kongregate.cs,使之与 Flash 脚本相对应:

```

using UnityEngine;
using System.Collections;

[NotRenamed]
[NotConverted]
public class Kongregate
{
    static bool _isini =false;

    // 初始化
    [NotRenamed]
    public static void Init()
    {
#if UNITY_WEBPLAYER

        if ( _isini)

```

```

        return;

        _isini = true;

        string javaCode = @"
            // Load the API
            var kongregate;
            kongregateAPI.loadAPI(onComplete);

            // Callback function
            function onComplete(){
            // Set the global kongregate API object
            kongregate = kongregateAPI.getAPI();
            }
        ";

        Application.ExternalEval(javaCode);
    #endif

    }

    // 上传分数或其他统计数据
    [NotRenamed]
    public static void Submit(string stat, int score)
    {
    #if UNITY_WEBPLAYER
        if (!_isini)
            return;

        Application.ExternalCall("kongregate.stats.submit", stat, score);
    #endif
    }
}

```

实际上, Kongregate.cs 中的大部分代码在 Flash 中都不会被执行, 它变成了一个“壳”, 最终在 Flash 中执行的是 Kongregate.as。

最后, 导出 Flash 游戏上传到 Kongregate 上预览效果, 本节的示例工程文件保存在资源文件目录\chapter08_ShootingGameFlash 中。

8.4 AssetBundle

Unity 的 Flash 版本并不支持 Unity 网页版的 Streaming, 如果要实现分流下载游戏, 需要使

用其他方法。

Unity Pro 版提供了一个叫 AssetBundle 的功能, 能够使游戏动态地从服务器下载资源(请参考本书第 6 章创建一个 Web 服务器), 这种技能可应用于桌面、手机、网页等各个版本, 游戏的客户端将变得很小, 但游戏必须始终处于联网状态。

下面, 我们将修改网页版的太空射击游戏, 使其在游戏初期下载一部分资源, 然后在游戏中去下载其他的资源。

8.4.1 打包资源

第一步, 我们先要将需要动态下载的资源打包上传到 Web 服务器。为了简单一些, 在这里我们只打包和主角飞船相关的资源。

步骤 01 使用 Unity 打开资源文件目录下的 chapter08_ShootingGameWeb 工程。

步骤 02 打包 AssetBundle 资源需要通过编辑器脚本编程(Editor Scripting)来实现。在 Project 窗口的 Assets 文件夹中新建一个名为 Editor 的文件夹, 这是存放编辑器脚本的专用文件夹。

步骤 03 在 Editor 文件夹中新建 ExportAssetBundles.cs 脚本:

```
using UnityEngine;
using UnityEditor;

public class ExportAssetBundles
{
    [MenuItem("Assets/Build AssetBundle From Selection - Track dependencies")]
    static void ExportResource()
    {
        // 显示保存窗口
        string path = EditorUtility.SaveFilePanel("Save Resource", "", "New Resource",
"unity3d");
        if (path.Length != 0)
        {
            // 从选择对象中创建资源。
            Object[] selection = Selection.GetFiltered(typeof(Object), SelectionMode.
DeepAssets);
            BuildPipeline.BuildAssetBundle(selection.activeObject, selection, path,
BuildAssetBundleOptions.CollectDependencies | BuildAssetBundleOptions.CompleteAssets,
BuildTarget.WebPlayer);
            Selection.objects = selection;
        }
    }

    [MenuItem("Assets/Build AssetBundle From Selection - No dependency tracking")]
    static void ExportResourceNoTrack()
```

```

{
    // 显示保存窗口
    string path = EditorUtility.SaveFilePanel("Save Resource", "", "New Resource",
"unity3d");
    if (path.Length != 0)
    {
        //从选择对象中创建资源.
        BuildPipeline.BuildAssetBundle(Selection.activeObject, Selection.objects,
path);
    }
}
}

```

MenuItem 将在 Project 窗口的右键快捷菜单中添加两个选项, 分别是导出选中的资源和导出选中的相关的资源, 如图 8-21 所示。

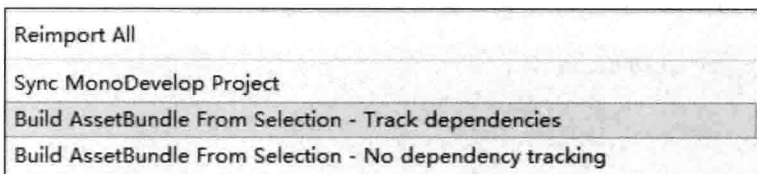


图 8-21 AssetBundle 选项

EditorUtility.SaveFilePanel 方法用于显示保存窗口。

BuildPipeline.BuildAssetBundle 方法用于打包并保存资源, 注意最后一个参数 BuildTarget, 默认为 WebPlayer, 它决定了 AssetBundle 对应的平台, 如果需要将 AssetBundle 使用到 Flash、IOS 或 Android, 这个参数也要对应修改。



提示

Unity 为不同平台创建的 AssetBundle 不能互相通用。

- 步骤 04** 将主角飞船制作成一个 Prefab, 命名为 PlayerPrefab, 这个名称很重要, 将来可能需要用名称来查找需要的资源。
- 步骤 05** 在 Project 窗口选择主角飞船的 Prefab, 然后单击右键选择【Assets/Build AssetBundle From Selection - Track dependencies】将主角飞船的 Prefab 及相关资源打包保存为 player.unity3d, 这个文件需要上传到服务器以供下载。
- 步骤 06** 在游戏场景中删除预设的主角飞船 Prefab。运行游戏, 游戏会直接 Game Over, 因为场景中的主角已经被删除了。

8.4.2 下载资源

现在, 游戏已经不能正常运行, 因为预设在场景中的主角飞船 Prefab 被删除了, 即使在 Project 窗口中仍然有这个 Prefab, 但毕竟它没有被使用到场景中, 也不会被集成到最后的游戏中。接下来, 我们将从 Web 服务器下载主角飞船的 Prefab 及相关资源, 动态地将其在游戏中创建出来。

步骤 01 修改 GameManager.cs 脚本, 添加一个 m_isloaded 属性判断是否已经下载主角:

```
// 是否下载主角
bool m_isloaded = false;
```

步骤 02 修改 OnGUI 函数中判断游戏结束的条件, 只有将主角下载之后才能触发游戏结束:

```
if (m_player != null )
{
    // 获得主角的生命值
    life = (int)m_player.m_life;
}
else if (m_isloaded) // game over
{
    ...
}
```

步骤 03 添加 DownloadPlayer 函数, 在这里进行真正的下载:

```
IEnumerator DownloadPlayer()
{
    GameObject obj = null;

    // 检查缓存是否可用
    float time = Time.time;
    while (!Caching.ready)
    {
        if (Time.time - time > 3.0f)
        {
            Debug.Log("Caching Not ready");
            break;
        }
        yield return null;
    }

    // 开始从服务器下载
    WWW www = WWW.LoadFromCacheOrDownload("http://127.0.0.1/player.unity3d", 1);

    // 等待下载完成
    yield return www;

    // 出现错误
    if (www.error != null)
    {
        throw new System.Exception("WWW download had an error:" + www.error);
    }
}
```



```

// 获得 AssetBundle
AssetBundle bundle = www.assetBundle;

obj=(GameObject) Instantiate(bundle.mainAsset,new
                             Vector3(0,0,-3),new Quaternion(0,-180,0,1) );

// 获得 Player 脚本
m_player = obj.GetComponent<Player>();

bundle.Unload(false);

m_isloaded = true;
}

```

在这段代码中，首先检查下载的缓存是否可用，为了防止进入死循环，设定一个等待时间。

WWW.LoadFromCacheOrDownload 方法用于连接 Web 服务器，同时也会检查本地是否有缓存的资源可用。它的第一个参数是网址和资源名称，第二个参数是版本号，默认本地缓存已有的资源将不会再次下载，但如果版本更新，还是会重新下载。

www.assetBundle 将返回下载的 **AssetBundle** 资源，使用 **Instantiate** 函数即可为这个资源动态创建实例。

当前，我们使用的 **bundle.mainAsset** 方法获取了全部下载的资源，如果只想取出这个资源包中的一部分，可以使用 **bundle.Load("PlayerPrefab")** 方法，直接读取资源的名称。

最后，使用 **bundle.Unload** 方法将资源卸载，释放内存。

在实际的项目中，有时可能只想检查资源是否下载，而不必马上读取使用它，这时可以使用 **Caching.IsVersionCached** 函数来检查，它有两个参数，第一个参数是资源的网络地址，第二个参数则是资源的版本号。

步骤 04 在 **Awake** 函数中添加 **StartCoroutine** 函数，执行 **DownloadPlayer** 函数下载资源：

```

void Awake()
{
    Instance = this;

    StartCoroutine(DownloadPlayer());
}

```

现在运行游戏，游戏已经回复正常。为了便于本地测试，我们可以使游戏在编辑器中只从本地读入资源，修改代码如下：

```

IEnumerator DownloadPlayer()
{
    GameObject obj = null;

    #if UNITY_EDITOR

```

```

obj = (GameObject)Resources.LoadAssetAtPath("Assets\\Prefabs\\PlayerPrefab.prefab",
                                             typeof(GameObject));

Instantiate(obj);
yield return null;

#else

    // 检查缓存是否可用
    float time = Time.time;
    while (!Caching.ready)
    {
        ...
    }
#endif

```

Unity 还提供了一种异步读取资源的方式, 这样就不会因为读取资源而影响游戏进程, 如下所示:

```

AssetBundle bundle = www.assetBundle;

//异步读取
AssetBundleRequest request =
    bundle.LoadAsync("PlayerPrefab", typeof(GameObject));

// 等待读取结束
yield return request;

//获得游戏体
obj = (GameObject)Instantiate(request.asset as GameObject, new Vector3(0, 0, -3), new
Quaternion(0, -180, 0, 1));

```

本节的示例工程文件保存在资源文件目录\chapter08_ShootingGameWebAssets 中, 如果要测试从服务器下载资源, 需要在本地建设一个 Web 服务器。

8.4.3 批量打包 AssetBundle

在前面的示例中, 我们创建了一个对话框将选中的资源打包为 AssetBundle, 但当我们的项目不断膨胀, 就会发现这种方式效率非常低下。实际上, Unity 提供了丰富的 API, 我们可以直接通过资源的路径找到所需要的资源, 将它们批量打包到指定位置。下面是一段代码示例, SearchDir 函数将会搜索所有当前工程 rawdata 目录内的.fbx 文件, BuildForAndroid 函数将它们打包为一个 AssetBundle。

```

using UnityEngine;
using System.Collections;

```

```

using UnityEditor; // 这是一个 Editor 脚本, 必须保存在 Editor 文件夹内

public class AssetsBuilder {
    // 工程内的资源路径
    public static string assetPath = Application.dataPath + "/rawdata/";
    // AssetBundle 的输出路径
    public static string outputPath = Application.dataPath + "../assetsbundle/";
    // AssetBundle 名称
    public const string bundlename = "cardbundle.unity3d";
    // 用来保存所有的资源路径名称
    public static System.Collections.Generic.List<string> filelist =
        new System.Collections.Generic.List<string>();

    [MenuItem("BundleBuilder/BuildForAndroid")] //将功能添加到菜单
    static void BuildForAndroid()
    {
        // 如果输出路径不存在则创建相应目录
        if (!System.IO.Directory.Exists(outputPath))
        {
            System.IO.Directory.CreateDirectory(outputPath);
        }

        filelist = new System.Collections.Generic.List<string>();
        // 查找并保存所有的资源路径
        SearchDir(assetPath);

        System.Collections.Generic.List<Object> objlist
            = new System.Collections.Generic.List<Object>();
        // 遍历所有的资源并读取它们
        foreach (string file in filelist)
        {
            Object obj = AssetDatabase.LoadAssetAtPath(file, typeof(Object));

            objlist.Add(obj);
        }

        Object[] objs = objlist.ToArray();
        // 打包 AssetBundle
        BuildPipeline.BuildAssetBundle(objs[0], objs, outputPath + bundlename,
            BuildAssetBundleOptions.CollectDependencies |
            BuildAssetBundleOptions.CompleteAssets, BuildTarget.Android);
    }
}
// 搜索所有子目录, 查找 .fbx 格式的文件

```

```

public static void SearchDir(string path)
{
    string[] files = System.IO.Directory.GetFiles(assetPath, "*.fbx");
    foreach (string f in files)
    {
        // 去掉路径中的 "Assets/"
        int n = f.IndexOf("Assets/");
        string s = f.Substring(n, f.Length - n);
        // 保存资源的路径
        filelist.Add(s);
    }
    // 继续搜索当前路径下的子路径
    string[] subdir = System.IO.Directory.GetDirectories(path);
    foreach (string dir in subdir)
    {
        // 递归
        SearchDir(dir);
    }
}
}

```

对于很多项目,可能需要创建若干个 **AssetBundle** 资源包,这种情况可以通过配置文件将需要打包的资源写在一个列表里,比如写在 XML 文件中,然后将其批量导出到指定的位置。

8.4.4 安全策略

无论是 Unity Web Player 或是 Flash 版的游戏,当它需要通过网络访问非同一个域名下的服务器,比如游戏是布置在 <http://mysite/webgame.unity3d>,然后游戏需要上传分数到 <http://myscores/highscore.php>,这时必须放置一个叫的 **crossdomain.xml** 文件在 <http://myscores> 网站根目录下,文件存为 ASCII 格式,其内容为:

```

<?xml version="1.0"?>
<cross-domain-policy>
<allow-access-from domain="*" />
</cross-domain-policy>

```

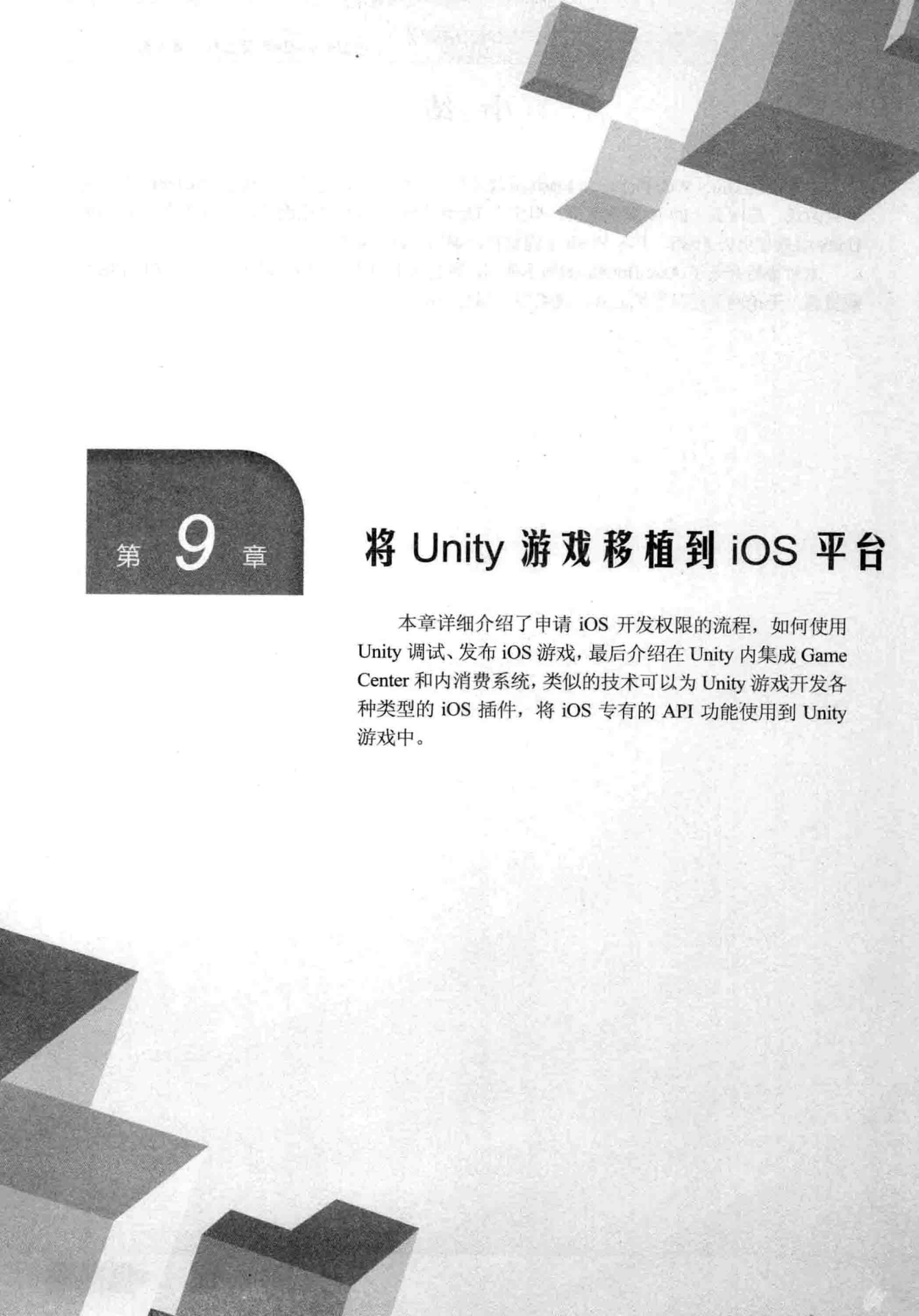
现在,域名为 <http://myscores> 的网站可以与任何域名下的 Unity 网页游戏进行通信。

使用 **crossdomain.xml** 文件的目的是主要是从安全性考虑,为了防止网页游戏在运行的时候偷偷去访问一些不该访问的地方。

小 结

本章围绕 Unity Web Player 和 Flash 游戏进行了详细介绍，包括如何构建 Streaming 的 Unity 网页游戏，与网页上的 js 脚本通信，自定义 Unity 网页游戏启动画面等，同时还介绍了如何将 Unity 游戏导出为 Flash，并与 Flash 工程整合，编写 Flash 插件等。

本章最后介绍了 AssetBundle 的基本使用，通过这个功能，我们可以在游戏的同时动态地下载资源，无论网页还是手机游戏，这类功能都是必不可少的。



第 9 章

将 Unity 游戏移植到 iOS 平台

本章详细介绍了申请 iOS 开发权限的流程，如何使用 Unity 调试、发布 iOS 游戏，最后介绍在 Unity 内集成 Game Center 和内消费系统，类似的技术可以为 Unity 游戏开发各种类型的 iOS 插件，将 iOS 专有的 API 功能使用到 Unity 游戏中。

9.1 iOS 简介

iOS (iPhone Operation System) 是苹果公司开发的手机操作系统, 主要安装在 iPhone 和 iPad 设备上。

开发 iOS 游戏或应用, 首先需要到苹果公司的官方网站申请 iOS 开发权限, 理论上只能在苹果公司的 Mac 计算机上开发。

iOS 游戏可以发布到苹果公司的 App Store 中, 这是发布 iOS 游戏的唯一合法途径。

开发 iOS 应用或游戏主要是使用 Objective-C 语言, 这是苹果公司专有的计算机语言, 语法比较特别, 好在使用 Unity 开发 iOS 游戏并不需要对 Objective-C 有非常深入的了解。

9.2 软件安装

开发 iOS 游戏, 首先要准备一台 Mac 计算机, 然后到苹果公司开发者网站 <https://developer.apple.com/devcenter/ios/index.action> 或 Mac 电脑上的 App Store 中免费下载 Xcode 软件, 将其安装到 Mac 上。

Xcode 是开发苹果软件的必备工具, 最新版只支持 Lion 操作系统, 而 Snow Leopard 操作系统, 要下载对应的老版本。

安装好 Xcode 后, 再安装 Mac 版的 Unity, 在 Mac 上安装 Unity 的过程与 PC 版基本一样, 这里将不再赘述。

9.3 申请开发权限

只是安装 Xcode 和 Unity 还不能在 iOS 设备上真机测试 Unity 游戏, 需要到苹果公司的开发者网站申请一个开发账号, 不过这需要每年使用信用卡支付 99 美金。申请的大体流程如下:

步骤 01 到苹果公司 iOS 开发者网站选择 Register 注册一个苹果公司账号。

步骤 02 使用注册好的苹果公司账号登录, 选择 Join the iOS Developer Program 申请加入 iOS 开发, 如图 9-1 所示。然后, 在新页面选择 Enroll Now 进行登记, 根据提示, 填入相应的信息。



提示

一定要填入真实的 E-mail 地址和信用卡信息。



图 9-1 申请 iOS 开发

步骤 03 目前, 针对开发年费, 苹果公司并不支持国内的在线支付, 所以当填写完信息后, 最后的页面可能会出现 Apple Online Store is unavailable (苹果公司在线商店不可用) 的提示, 这个页面上出现的信息都很重要, 包括各种 ID 和传真号码等, 需要保存下来。之后下载这个页面上提供的 purchaseform.pdf 文件, 将其填写好后传真给苹果公司。

步骤 04 苹果公司收到传真后, 如无问题会扣除信用卡中的 99 美金作为开发年费, 再发回一个 activation code (激活码) 到注册时留下的邮箱中。最后, 按邮件中的说明激活, 即可在 App Store 中发布自己的游戏了。

9.4 设置 iOS 开发环境

有了 iOS 开发资格, 还需要在苹果公司的开发者网站上完成各项配置, 并与 Mac 电脑和测试机关联。设置步骤如下:

步骤 01 到苹果公司 iOS 开发者网站使用苹果公司账号登录, 在页面上找到 iOS Developer Program 一栏, 选择 iOS Provisioning Portal 进入新的页面, 在网页左侧有 Home (首页)、Certificates (执照证书)、Devices (设备)、App IDs (软件的 ID)、Provisioning (配置)、Distribution (发布) 几个选项, 如图 9-2 所示。

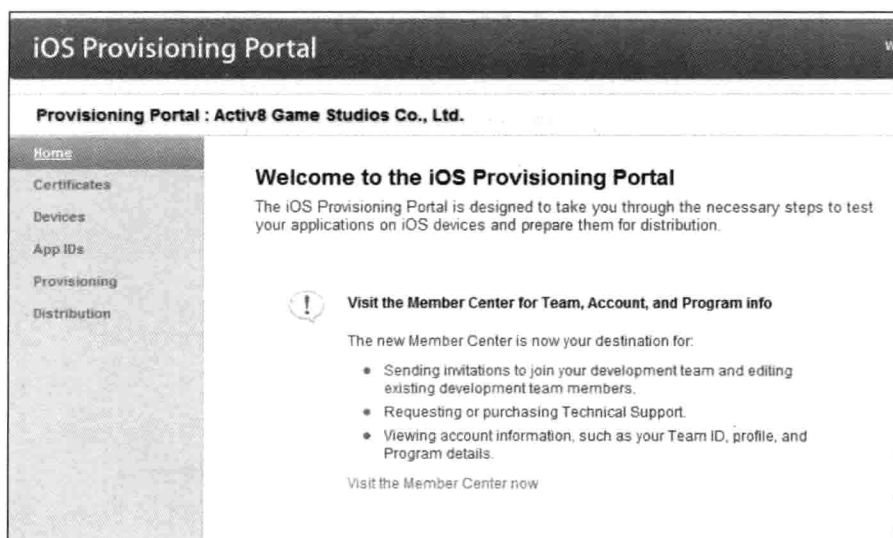


图 9-2 iOS Provisioning Portal 页面

步骤 02 选择 Certificates 进入新页面。选择 Request Certificate 来请求一个新的证书，如图 9-3 所示。之后出现的页面将说明如何申请证书。



图 9-3 申请开发证书



提示

证书有两种：一种用于开发；一种用于发布最终产品。默认是在 Development 页面下申请用于开发的证书。

接下来按说明申请开发证书即可。

步骤 03 在 Mac 电脑上打开 Application 文件夹，打开 Utilities 文件夹，运行 Keychain Access。

步骤 04 在 Keychain 菜单栏选择【Certificate Assistant】→【Request a Certificate from a Certificate Authority】，打开 Certificate Information 窗口，输入 Email 和名字，选择 Saved to disk，如图 9-4 所示。最后，选择 Continue（下一步）保存一个 Certificate Signing Request（CSR）文件到硬盘上。

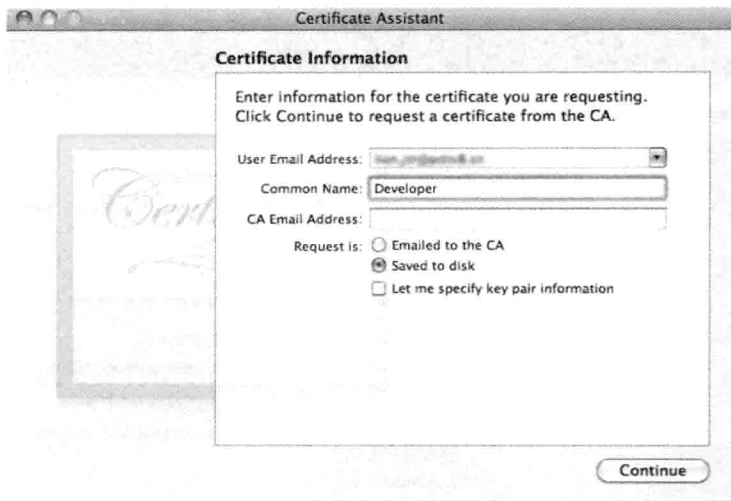


图 9-4 填写信息

步骤 05 回到网页上的 Certificates 页面，确定选择 Development 标签，选择“浏览”上传 Certificate Signing Request (CSR) 文件。然后选择 Submit 提交并创建证书。

步骤 06 仍回到 Certificates 页面，刷新一下，会看到新建证书旁边有一个 Download 选项，如图 9-5 所示。双击即可下载这个证书，下载完成后，双击下载的文件，将其装入 Mac 电脑中。



图 9-5 下载证书

步骤 07 将 iOS 设备连接到 Mac 电脑上，打开 Xcode，在菜单栏选择【Window】→【Organizer】，选择窗口上方的 Devices 标签，会看到 iOS 设备的相关信息。选择 Use For Development 使其成为开发机，然后复制 Identifier 后面一个 40 位长的字符串，它是 iOS 设备的 ID，如图 9-6 所示。

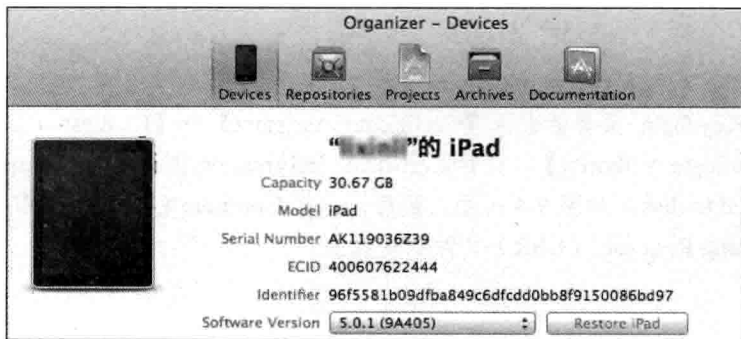


图 9-6 获得 iOS 设备的 ID

- 步骤 08** 回到前面的网页上，选择 Devices 打开新页面。选择 Add Devices 选项添加用于测试的 iOS 设备，会出现一个新的页面，如图 9-7 所示。在 Device Name 中输入 iOS 设备的名字，这个名字可以是任意的。然后输入或粘贴 iOS 设备的 ID。

图 9-7 输入 iOS 设备的名字和 ID

- 步骤 09** 在网页上选择 App IDs，选择 New App ID，在 Bundle Identifier 内填写应用的 APP ID，注意一定要按照 com.XXX.XXX 这样的格式填写，比如 com.company.gamename。如果希望这个 ID 是所有游戏通用的，也可以将 Bundle Identifier 设为*。
- 步骤 10** 在网页页面上选择 Provisioning，选择 New Profile 建立新档案，然后在出现的新页面中输入 Profile 的名字，并与前面创建的 Certificates、App ID 和 iOS 设备关联，这些设备都是在步骤 8 中添加的。最后选择 Submit 提交创建一个新的 Provisioning Profile。
- 步骤 11** 确定 iOS 设备已与 Mac 电脑连接。在网页上选择前面创建的 Provisioning Profile 旁边的 Download，如图 9-8 所示，下载这个 Provisioning Profile 到 Mac 电脑上，双击下载的文件则会将其安装到 Xcode 和 iOS 设备中。至此，一些常规设置就结束了。

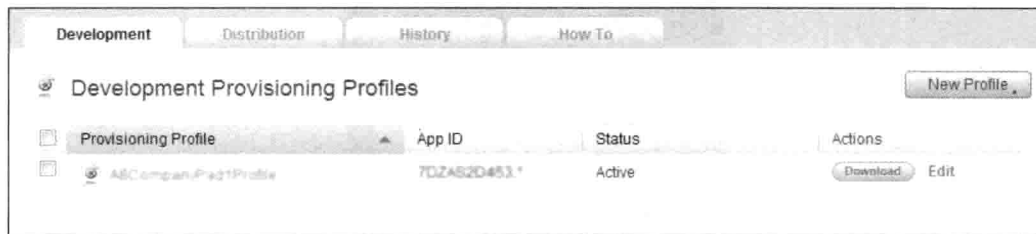


图 9-8 下载 Provisioning Profile

9.5 测试 iOS 游戏

完成了以上繁复的设置后，现在终于能够做点事情了。本节将把一个在 PC 上开发的 Unity 程序移植到 iOS 平台，并在 iOS 测试机上跑起来。

- 步骤 01** 在资源文件目录下找到 chapter09_iOS_Start 工程，这是在 PC 上开发的一个 Unity 工程。可以先运行看一下，屏幕上会显示几个简单的按钮，包括显示 Game Center 每按一次加一分，上传分数/成就到 Game Center 和内消费购买按钮，还包括分数、成就和点数的简单 UI，另外还有一个 3D 模型，它没有任何作用，只是为了使画面丰富一些，如图 9-9 所示。这些按钮暂时还不包括任何功能，我们将在后面逐一

实现这些按钮中的相应功能。

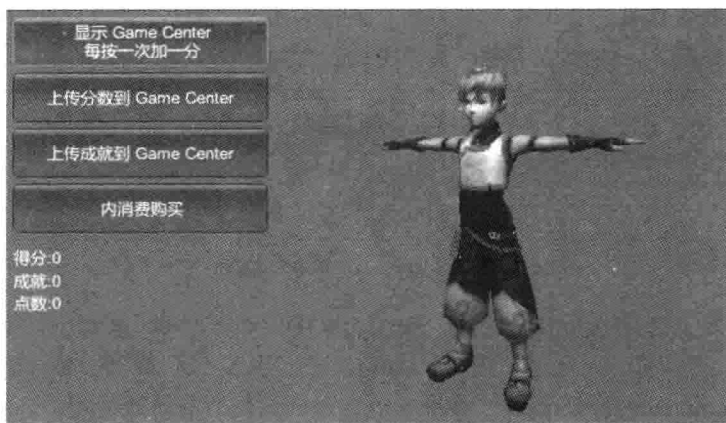


图 9-9 简单的 UI 显示

- 步骤 02** 我们无法在 Windows 上使用 Unity 开发 iOS 游戏, 将这个工程复制到安装有开发权限的 Mac 电脑上, 使用 Mac 版的 Unity 打开这个工程。
- 步骤 03** 在菜单栏选择【File】→【Build Settings】, 在 Platform 中选择 iOS, 选择 Switch Platform 将工程转为 iOS 工程, 如图 9-10 所示。



图 9-10 转换为 iOS 平台

- 步骤 04** 选择 Player Settings, 在 Inspector 窗口进行 iOS 平台的设置。
- 步骤 05** 在 Resolution and Presentation 选项组中设置屏幕旋转方向, Portrait 表示纵向, Landscape 表示横向, 这里选择 AutoRotation, 并且选中 Landscape Right 和 Landscape Left, 屏幕将会根据设备旋转方向自动调整方向, 但只能横向旋转, 如图 9-11 所示。

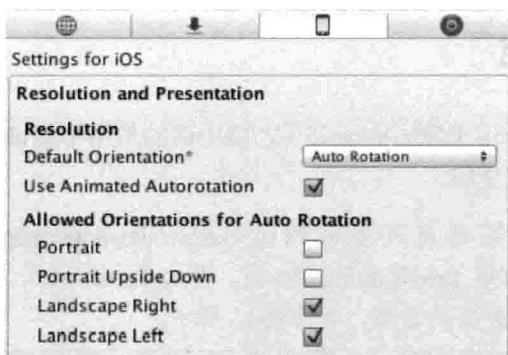


图 9-11 设置屏幕显示方向

- 步骤 06** 在 Other Settings 选项组中设置 Bundle Identifier, 这个 ID 需要与在网页上设置的 Bundle ID 一致, 格式通常是 com.XXX.XXX。
- 步骤 07** 在 Splash Image 选项组中可以自定义启动画面。
- 步骤 08** 在 Other Settings 选项组中设置 Target Device 为 iPhone 或 iPad, 也可以选择 iPhone+iPad 两种设备通用, 但这种情况要确定游戏的 UI 会自动适配不同的分辨率, 如图 9-12 所示。

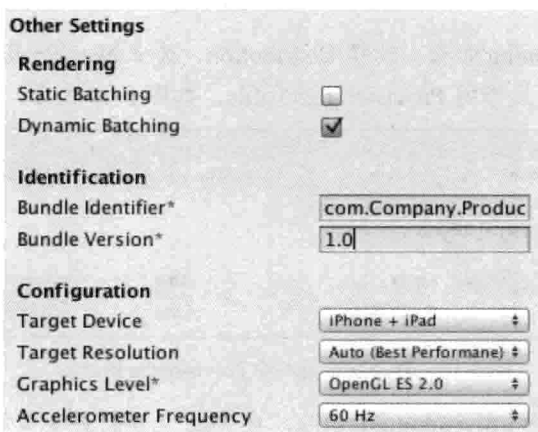


图 9-12 设置目标设备

- 步骤 09** 确定 iOS 设备与 Mac 电脑处于连接状态, 选择 Build And Run, 然后选择一个路径保存。片刻之后, Xcode 会自动启动, 如果弹出对话框, 提示有 codesign wants to sign using key “XXX” in your keychain 这样的信息, 选择 Always Allow 选项即可。最后, 如无错误, 程序将会运行在 iOS 设备上, 画面和操作与在 PC 或 Mac 上几乎是一样的。

9.6 发布 iOS 游戏

虽然我们只通过几个简单的步骤就可以在 iOS 设备上运行 Unity 程序, 但如果想在 App Store 中发布 Unity 游戏, 还需要一些其他的设置。

9.6.1 申请发布证书

发布游戏需要申请一个用于发布的证书（之前申请的是用于测试的证书），这个证书只需要申请一次，将来可以反复使用。

步骤 01 再次进入苹果公司开发者网站 <https://developer.apple.com/devcenter/ios/index.action>，进入 iOS Provisioning Portal，进入 Certificates，选择 Distribution，按提示创建一个用于发布的证书，如图 9-13 所示。



图 9-13 创建发布证书

步骤 02 进入 Provisioning 页面，选择 Distribution，选择 New Profile，按提示创建一个用于在 App Store 发布的 Provisioning Profile，如图 9-14 所示。

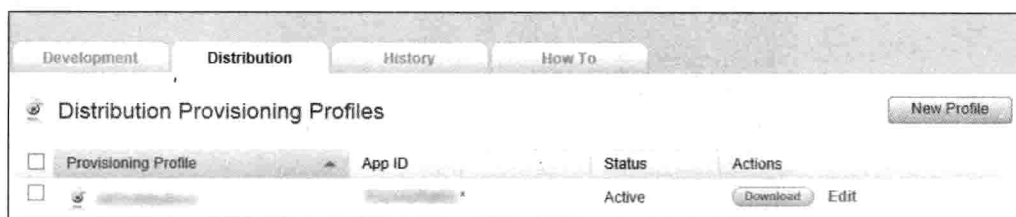


图 9-14 创建发布版本的 Provisioning Profile

步骤 03 下载新创建的 Provisioning Profile，在 Mac 上运行并安装它。

9.6.2 创建新应用

虽然有了发布证书，但我们还需要到苹果公司的官方网站 iTunes Connect 去创建新应用并进行相关设置，一旦我们准备好要发布的游戏，就要经常访问这里了。

步骤 01 登录 <https://itunesconnect.apple.com>，这是 App Store 的后台管理网站，在这里可以发布、设置游戏相关信息，也可以查看财务报表或下载量等，如图 9-15 所示。

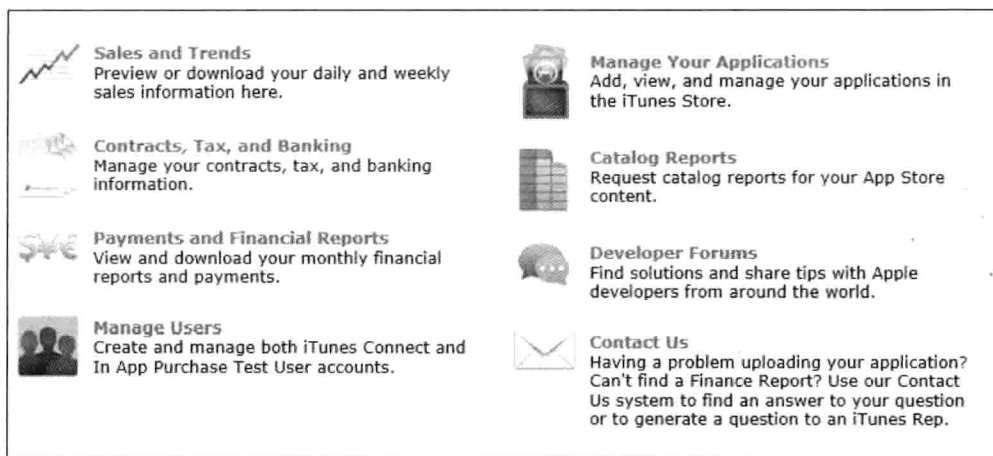


图 9-15 App Store 后台管理页面

步骤 02 选择 **Manage Your Applications** 进入应用管理页面。选择 **Add New App** 创建一个新的应用，然后按提示设置应用的名称、说明和截图等，比较重要的是 **Bundle ID** 的设置，它必须与 Unity 中设置的 **Bundle Identifier** 一致，如图 9-16 所示。

图 9-16 创建新应用

步骤 03 当设置好所有的选项后，在设置页面的上方选择 **Ready to Submit**，通知苹果公司我们已经设置好了，随时可以提交。这一步非常重要，没有这个操作，在 Mac 上将不能提交所发布的版本。

9.6.3 提交审核

现在，我们需要在 Unity 和 Xcode 中重新 Build 游戏，生成一个用于发布的版本并提交给苹果公司进行审核。

步骤 01 在 Unity 的 Build Settings 窗口使用 Build 生成 Xcode 工程，注意这里是用 Build 而

不是 Build And Run。

- 步骤 02** 使用 Xcode 打开 Build 好的 Xcode 工程, 选择 Build Settings, 找到 Code Signing Identity, 将 Release 下的 Any iOS SDK 设置为发布版本的 Provisioning Profile, 如图 9-17 所示。

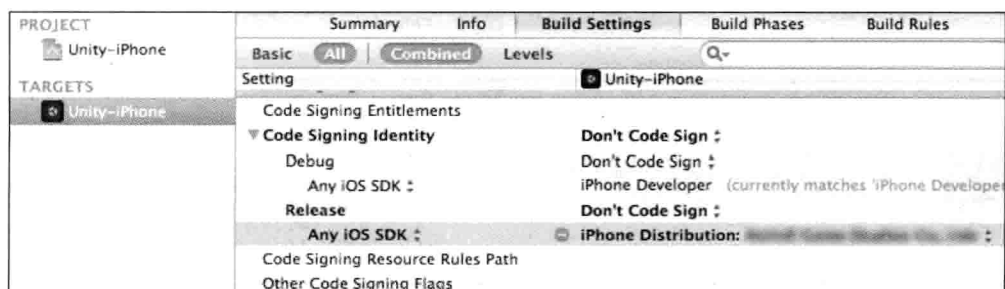


图 9-17 设置发布

- 步骤 03** 选择工具栏上的 Unity-iPhone, 选择 Edit Scheme, 如图 9-18 所示。

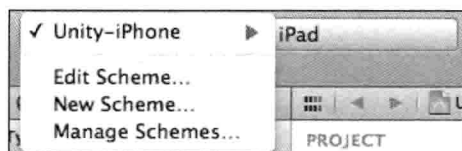


图 9-18 设置

- 步骤 04** 选择 Archive, 在 Build Configuration 中选择 Release, 如图 9-19 所示。

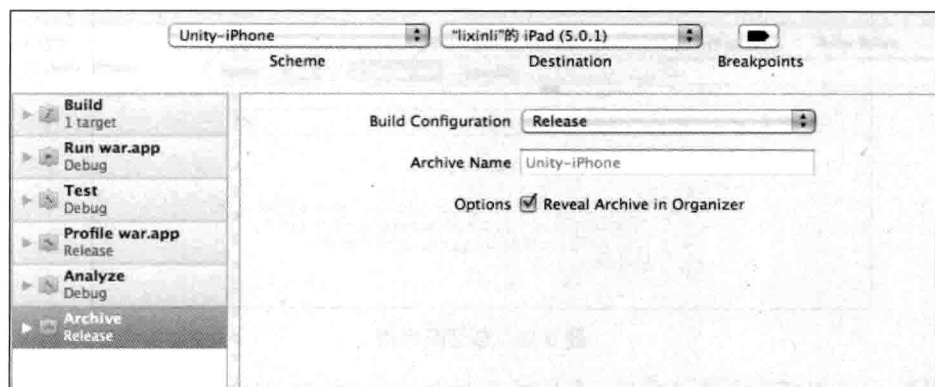


图 9-19 设置 Archive 为 Release 版本

- 步骤 05** 在 Xcode 菜单栏选择【File】→【Project Settings】打开新窗口, 在 Derived Data Location 中选择 Custom, 然后自定义一个用于输出的路径保存输出文件, 如图 9-20 所示。

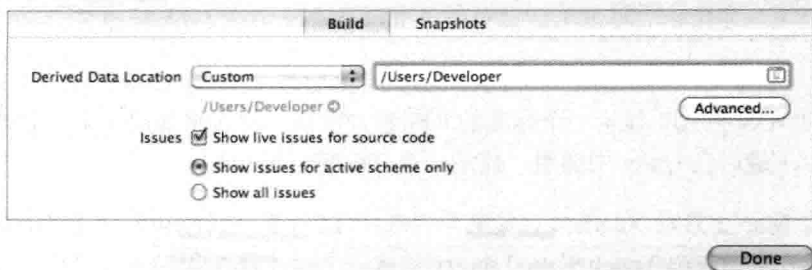


图 9-20 设置自定义输出路径

- 步骤 06** 在资源列表中选择 Info.plist，设置 Localization native development region 为 China，这个选项会影响产品在 App Store 里的语言类型显示，设为 China 后，游戏的语言类型即显示为中文，如图 9-21 所示。

	Key	Type	Value
Unity-iPhone			
1 target, iOS SDK 5.0			
SystemConfiguration.framework			
CoreLocation.framework			
MediaPlayer.framework			
Default.png			
Icon.png			
Classes			
Libraries			
Info.plist	Localization native development region	String	China
Frameworks	Executable file	String	\$(EXECUTABLE_NAME)
	Icon file	String	Icon.png
	Bundle identifier	String	com.companyname.\${PRODUCT_NAME}
	InfoDictionary version	String	6.0
	Bundle name	String	\$(PRODUCT_NAME)
	Bundle OS Type code	String	APPL
	Application requires iPhone environment	Boolean	YES
	Bundle version	String	2.2

图 9-21 设置语言类型

- 步骤 07** 在 Xcode 菜单栏选择【Product】→【Archive】对当前工程进行 Build，之后会自动弹出 Organizer 窗口，选择 Submit，按提示提交游戏到苹果公司等待审核。

9.7 集成 Game Center

虽然理论上我们可以不写一行 Objective-C 代码就能够发布一款使用 Unity 制作的 iOS 游戏，但问题是当我们需要集成一些 iOS 平台专有的功能，比如 Game Center 或内消费（游戏内消费）功能，我们就不得不与 Objective-C 打交道，在 Xcode 中为 Unity 编写插件，使 Unity 游戏也能使用到上述功能。

9.7.1 Xcode 到 Unity

在 Xcode 中编写 Unity 插件通常需要几个步骤：

- 步骤 01** 在 Xcode 中使用 Objective-C 语言调用 iOS 的 API 实现所需要的功能。
- 步骤 02** 在 Xcode 中使用 C 语言语法来包装使用 Objective-C 写的代码。
- 步骤 03** 将在 Xcode 中完成的代码导出为一个 .a 文件，然后复制到 Unity 工程中的 Plugins\iOS 目录内（或者直接将源代码文件复制过去也可以）。
- 步骤 04** 在 Unity 中使用“DllImport”导入在 Xcode 中完成的函数并用 C# 函数进行包装，

在实际使用中直接调用这些包装着 iOS 插件功能的 C# 函数即可。

为了便于理解,本节将完成一个简单的示例说明以上步骤的具体实现。我们将使用 Objective-C 调用 iOS 的 API 显示一个标准的 iOS 警告窗口,然后将其用一个 C 语言函数包装,导出到 Unity 中,最后在 Unity 中调用,显示一个 iOS 警告窗口。

步骤 01 在 Mac 上启动 Xcode,在其菜单栏选择【File】→【New】→【New Project】,然后选择 Cocoa Touch Static Library 创建一个静态库工程,它最终将生成一个.a 文件供 Unity 使用,如图 9-22 所示。

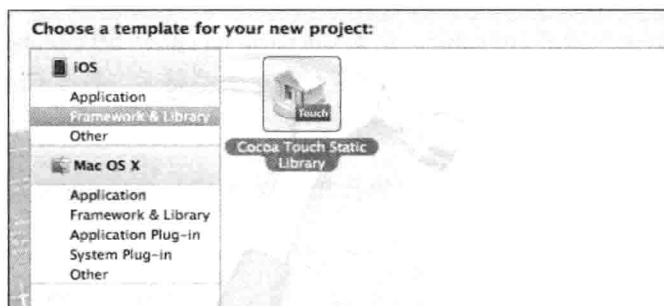


图 9-22 在 Xcode 中创建静态库工程

步骤 02 输入工程名 UGameManager,取消 Include Unit Tests 和 Use Automatic Reference Counting 选项,如图 9-23 所示。创建工程后,在工程中默认会有一个 UGameManager.h 和 UGameManager.m 文件。

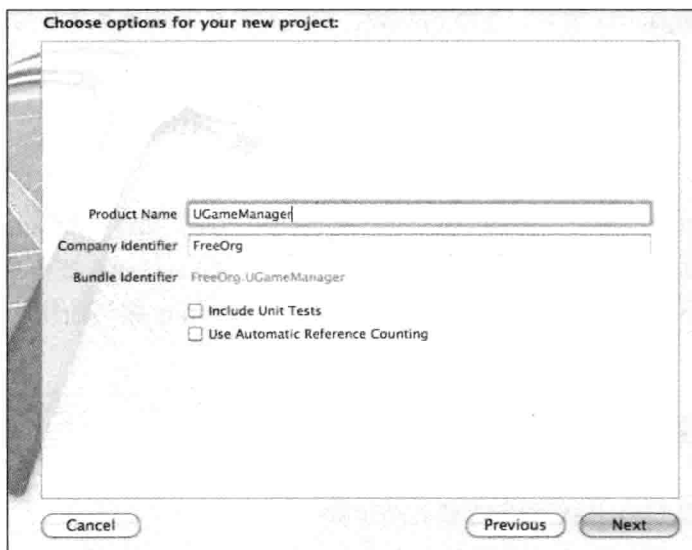


图 9-23 输入工程名



提示

选中 Use Automatic Reference Counting,将不能使用 Objective-C 的 release 函数手动释放内存,而是由程序自动管理。

步骤 03 打开 UGameManager.h, 输入代码:

```
#import <Foundation/Foundation.h>

@interface UGameManager : NSObject

// 获取 UGameManager 对象的实例
+ (UGameManager*) instance;

// Objective-C 函数, 显示一个警告窗口
- (void) ShowWarningBox:(NSString*) strTitle: (NSString*) strText;

// *****
// c 函数, 将被导出到 Unity 中
// *****

// 初始化, 创建一个 UGameManager 实例
void _startup(void);

//显示一个警告框
void _showWarningBox(char* strTitle ,char* strText);

@end
```

Objective-C 的.h 文件与 C 语言的.h 文件类似, 用来声明类或函数等, 可以被其他文件引用。这里我们声明了一个叫 UGameManager 的类, 注意@符号, Objective-C 所有的类都继承自 NSObject。

初次接触 Objective-C 可能会对函数前面的“+”和“-”感到不解, 这其实只是一种特殊的语法, “+”表示静态, 类似 C 语言的 static, “-”则是非 static。

Objective-C 的另一个特别之处是函数的参数, 每个参数之间用“:”号分隔而不是用“,”号, 参数也不需要写在一个括号内。

步骤 04 打开 UGameManager.m, 输入代码:

```
#import "UGameManager.h"
#import <UIKit/UIKit.h>

@implementation UGameManager

static UGameManager* gameMgr=nil;

// 初始化
- (id)init
{
    self = [super init];
```

```

        if (self) {
            // Initialization code here.
        }

        return self;
    }

    // 销毁
    - (void) dealloc
    {
        [super dealloc];
    }

    // 获得 UGameManager 对象实例
    + (UGameManager*) instance
    {
        if (gameMgr==nil)
        {
            gameMgr=[[UGameManager alloc]init];
        }

        return gameMgr;
    }

    // 显示警告窗口
    - (void) ShowWarningBox:(NSString*) strTitle: (NSString*) strText
    {
        UIAlertView* alertview=[ [UIAlertView alloc] initWithTitle:strTitle message:strText
        delegate:nil cancelButtonTitle:@"OK" otherButtonTitles:nil];
        [alertview show];

        //不能在 Use Automatic reference Counting 模式下使用
        [alertview release];
    }

    // *****
    // c 函数, 将被导出到 Unity 中
    // *****
    // 初始化, 内部执行了创建 UGameManager 对象实例的函数
    void _startup()
    {
        [UGameManager instance];
    }

```



```
// 显示警告窗口, 内部执行了创建 Objective-C 的版本
void _showWarningBox(char* strTitle ,char* strText)
{
    [ gameMgr ShowWarningBox:[NSString stringWithUTF8String: strTitle]:[NSString
stringWithUTF8String: strText] ];
}

@end
```

UGameManager.m 是实现文件, 其中 init 函数类似于默认构造函数, 会在实例化的时候自动执行。

Instance 函数是一个静态函数, 当 UGameManager 实例不存在的时候返回一个新创建的实例到 gameMgr。

在 ShowWarningBox 函数中我们创建了一个 UIAlertView, 它将显示出一个 iOS 警告窗口。

在最后, 我们使用标准的 C 语言语法创建了函数 _startup 和 _showWarningBox, 并在这两个函数内调用了 Objective-C 的相应函数。

步骤 05 在 Xcode 菜单栏选择 **【File】→【Project Settings】**, 在 Derived Data Location 中选择 Project-relative, 这将使输出的 .a 文件保存在当前工程目录内。

步骤 06 在 Xcode 菜单栏选择 **【Product】→【Build For】→【Build For Archiving】** 导出 .a 文件, 然后到当前工程的 Derived Data/UGameManager/Build/Products/Release-iphones 目录找到 libUGameManager.a 文件。

接下来的工作将在 Unity 中进行。

步骤 01 在 Unity 工程的 Assets 目录下创建 Plugins\iOS 目录, 将 libUGameManager.a 复制到 iOS 文件夹中, 如图 9-24 所示。这个目录结构是特定的, 不能随意改变。

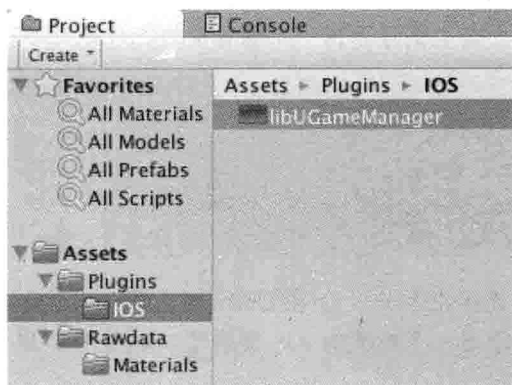


图 9-24 创建 iOS 插件目录

步骤 02 在 Unity 中创建脚本 UnityiOSKit.cs, 我们将在这个脚本导入 .a 文件中的 C 函数, 代码如下:

```
using UnityEngine;
```

```

using System.Collections;
using System.Runtime.InteropServices;

public static class UnityiOSKit{

    //导入_startup 函数
    [DllImport ("__Internal")]
    private static extern void _startup();

    //导入_showWarningBox 函数
    [DllImport ("__Internal")]
    private static extern void _showWarningBox(string strTitle ,string strText);

    // C#版本的初始化函数, 内部执行的是_startup()
    static public void Start()
    {
        if ( Application.platform==RuntimePlatform.IPhonePlayer)
            _startup();
    }

    // C#版本的显示警告窗口函数, 内部执行的是_showWarningBox
    static public void ShowWarningBox(string strTitle ,string strText)
    {
        if ( Application.platform==RuntimePlatform.IPhonePlayer)
            _showWarningBox( strTitle , strText);
    }
}

```

步骤 03 打开脚本 iOSAPP.cs, 在 Start 函数中执行 ShowWarningBox 函数, 当程序在 iOS 设备上启动后, 会立刻显示出一个警告窗口, 代码如下:

```

void Start () {

    UnityiOSKit.Start();
    UnityiOSKit.ShowWarningBox("Hello","World");
}

```

步骤 04 确定 iOS 设备连接在 Mac 上, 在 Unity 内选择 Build And Run, 程序将运行在 iOS 设备上, 并显示出一个标准的 iOS 警告窗口, 如图 9-25 所示。



图 9-25 显示 iOS 警告窗口

在 Mac 上编写脚本会遇到一个问题，就是不能正确地显示中文，因为 Mac 版的 Mono Developer 编辑器无法将脚本保存为所需要的 UTF-8 格式，所以如果发现程序中的中文都变成了乱码，那也是正常的。解决方案是使用 Mac 版的 Ultra Edit 编辑脚本，它可以正确地保留中文信息。除此以外，我们还要确定当前的文字 UI 使用了正确的中文字体，在本示例中，初始的工程中已经预设好了中文字体。

本节的示例工程文件保存在资源文件目录 chapter09_TestiOSPlugin 内，其中 UGameManager 目录内是 Xcode 工程，UnityiOSProject 目录内则是最后的 Unity 工程。

9.7.2 设置高分榜和成就

所有的 iOS 游戏都可以集成 App Store 专用的游戏分数排行榜和成就功能，分数和成就的记录都保存在苹果公司的服务器上，所以使用这些功能需要保持联网状态。接下来，我们要做的是调用 iOS 的 API 将分数和成就上传到服务器上，但在这之前，我们首先要到苹果公司的网站 <https://itunesconnect.apple.com> 去设置一番。

步骤 01 登录 itunesconnect 网站后，选择 Manage Your Applications，选择游戏，选择 Manage Game Center，如图 9-26 所示。



图 9-26 设置 Game Center

步骤 02 选择 Leaderboard 添加分数排行榜，然后根据提示设置。这里要注意 Leaderboard ID 的填写，它必须是独一无二的，不能与其他游戏 ID 重名，我们将在游戏中通过这个 ID 关联相应的排行榜，如图 9-27 所示。

Leaderboard Reference Name: 分数排行榜

Leaderboard ID: com.company.game.leaderboard

Score Format Type: Integer

Sort Order: ☐ Low to High ☒ High to Low

Score Range (Optional): -9223372036854775000 To 9223372036854775000

图 9-27 设置高分榜

步骤 03 选择 Achievements 添加成就，我们可以添加任意个成就，当完成一个成就，就可以获得一定的成就值 (Point Value)，注意 Achievement ID 的填写，与高分榜一样，这个 ID 必须是独一无二的，需要在游戏中使用这个 ID 与其关联，如图 9-28 所示。

Achievement Reference Name: 游戏成就

Achievement ID: com.yourcompany.yourgame.sample

Point Value: 100

440 of 1000 Points Remaining

Hidden: ☒ Yes ☐ No

Achievable More Than Once: ☒ Yes ☐ No

图 9-28 设置成就

步骤 04 在编辑游戏说明的页面找到 Game Center 选项，选中高分榜和成就，这些操作必须在执行 Ready to Submit 之前进行。提交游戏后，高分榜和成就将交由苹果公司方审核。

9.7.3 实现 Game Center 功能

在网站上设置好分数和成就后，我们就可以在代码中通过调用 iOS 的 API 上传分数和成就了。

步骤 01 在 Mac 上启动 Xcode，打开 9.7.1 小节完成的工程。

步骤 02 打开 UGameManager.h，修改代码如下：

```
#import <Foundation/Foundation.h>
#import <GameKit/GameKit.h>

@protocol UGameManagerDelegate <NSObject>
@optional
- (void) processGameCenterAuth: (NSError*) error;
@end

@interface UGameManager : NSObject
```

```

{
    // 代理
    id <UGameManagerDelegate, NSObject> delegate;

    // Unity 的 view controller
    id unityviewController;

    // Game Center 是否可用
    BOOL enableGameCenter;
}

@property (nonatomic, assign) id <UGameManagerDelegate> delegate;
@property (nonatomic, retain) id unityviewController;
@property (nonatomic) BOOL enableGameCenter;

+ (UGameManager*) instance;

// 显示警告窗口
- (void) ShowWarningBox:(NSString*) strTitle: (NSString*) strText;

- (void) registerForAuthenticationNotification;
- (void) authenticationChanged;

// 判断 Game Center 是否可用
+ (BOOL) isGameCenterAvailable;

// 认证本地用户
- (void) authenticateLocalPlayer;

// 上传积分
- (void) reportScore: (NSString*)identifier hiscore: (int64_t) score;

// 上传成就
- (void) reportAchievementIdentifier: (NSString*) identifier percentComplete: (float)
percent;

// 显示排行榜
- (void) showLeaderboard: (NSString*)leaderboard;

// 显示成就
- (void) showAchievements;

// export c function
void _startup(void);

```



```

void _showWarningBox(char* strTitle ,char* strText);
bool _isGameCenterAvailable(void);
void _authenticateLocalPlayer(void);
void _reportScore( char* identifier, int score );
void _reportAchievement( char* identifier ,float percent );
void _showLeaderboard(char* leaderboard);
void _showAchievements(void);

@end

```

可以发现,头文件中增加了很多与 Game Center 相关的函数,它们主要是用来认证当前用户是否已登录 Game Center,上传积分与成就,显示排行榜和成就界面的。另外,使用 Game Center,必须导入与之相关的 GameKit/GameKit.h 头文件。

步骤 03 打开 UGameManager.m, 在原有代码的基础上添加如下代码:

```

// 与头件@property 相对应
@synthesize delegate;
@synthesize unityviewController;
@synthesize enableGameCenter;

// 判断 Game Center 是否可用
+ (BOOL) isGameCenterAvailable
{
    // 检查 GKLocalPlayer API
    Class gcClass = (NSClassFromString(@"GKLocalPlayer"));

    // 检查当前 iOS 系统是否是在 4.1 以上
    NSString *reqSysVer = @"4.1";
    NSString *currSysVer = [[UIDevice currentDevice] systemVersion];
    BOOL osVersionSupported = ([currSysVer compare:reqSysVer options:NSNumericSearch] !=
    NSOrderedAscending);

    return (gcClass && osVersionSupported);
}

// 认证本地用户
- (void) authenticateLocalPlayer
{
    if([GKLocalPlayer localPlayer].authenticated == NO)
    {
        [[GKLocalPlayer localPlayer] authenticateWithCompletionHandler:^(NSError *error)
        {
            [self callDelegateOnMainThread: @selector(processGameCenterAuth:)
            withArg: NULL error: error];
        }];
    }
}

```

```

        }};
    }
    else {

        enableGameCenter=YES;
    }
}

// 如果认证失败
- (void) callDelegateOnMainThread: (SEL) selector withArg: (id) arg error: (NSError*) err
{
    dispatch_async(dispatch_get_main_queue(), ^(void)
    {
        [self callDelegate: selector withArg: arg error: err];
    });
}

- (void) callDelegate: (SEL) selector withArg: (id) arg error: (NSError*) err
{
    assert([NSThread isMainThread]);
    if([delegate respondsToSelector: selector])
    {
        if(arg != NULL)
        {
            [delegate performSelector: selector withObject: arg withObject: err];
        }
        else
        {
            [delegate performSelector: selector withObject: err];
        }
    }
    else
    {
        NSLog(@"Missed Method");
    }
}

// 处理 Game Center 认证结果
- (void) processGameCenterAuth: (NSError*) error
{
    if(error == NULL)
        enableGameCenter=YES;
    else
        enableGameCenter=NO;
}

```

```

    }

    // 如果 Game Center 用户变化,发出通知
    - (void) registerForAuthenticationNotification
    {
        NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
        [nc addObserver: self
             selector:@selector(authenticationChanged)
             name:GKPlayerAuthenticationDidChangeNotificationName
             object:nil];
    }

    //处理通知结果
    - (void) authenticationChanged
    {
        if ([GKLocalPlayer localPlayer].isAuthenticated)
            // 如果登录 Game Center...
        else
            // 否则...
    }

```

通常会先使用 `isGameCenterAvailable` 来判断 Game Center 是否可用, 如果可用, 接着使用 `authenticateLocalPlayer` 认证 Game Center 用户; 如果失败了, 将调用 `callDelegateOnMainThread` 来处理失败结果。

步骤 04 在 `UGameManager.m` 中添加上传分数、成就和显示分数、成就界面的代码为:

```

// 上传分数
- (void) reportScore: (NSString*)identifier hiscore: (int64_t) score
{
    if ( score<0 || !enableGameCenter)
        return;

    GKScore *scoreReporter = [[[GKScore alloc] initWithCategory:identifier] autorelease];
    scoreReporter.value = score;

    [scoreReporter reportScoreWithCompletionHandler:nil];
}

// 上传成就, 最大值 100 (完成成就)
- (void) reportAchievementIdentifier: (NSString*) identifier percentComplete: (float)
percent
{
    if ( percent<0 || !enableGameCenter)
        return;
}

```



```

        GKAchievement *achievement = [[[GKAchievement alloc] initWithIdentifier: identifier]
autorelease];
        if (achievement)
        {
            achievement.percentComplete = percent;
            [achievement reportAchievementWithCompletionHandler:nil];
        }
    }

    // 显示分数排行榜
    - (void) showLeaderboard: (NSString*)leaderboard
    {
        if ( !enableGameCenter) //如果 Game Center 不可用,弹出一个警告窗口
        {
            UIAlertView* alertview=[[ UIAlertView alloc] initWithTitle:@"Game Center
Disabled" message:@"Sign in with the Game Center application to view leaderboards" delegate:nil
cancelButtonText:@"OK" otherButtonTitles:nil];
            [alertview show];

            [alertview release];
            return;
        }

        GKLeaderboardViewController *leaderboardController = [[GKLeaderboardViewController
alloc] init];
        if (leaderboardController != NULL)
        {
            leaderboardController.category = leaderboard;
            leaderboardController.timeScope = GKLeaderboardTimeScopeAllTime;
            leaderboardController.leaderboardDelegate = unityviewController;

            [unityviewController presentModalViewController: leaderboardController animated:
YES];
        }
    }

    // 显示成就窗口
    - (void) showAchievements
    {
        if ( !enableGameCenter) //如果 Game Center 不可用,弹出一个警告窗口
        {
            UIAlertView* alertview=[[ UIAlertView alloc] initWithTitle:@"Game Center

```

```

Disabled" message:@"Sign in with the Game Center application to view achievements" delegate:nil
 cancelButtonTitle:@"OK" otherButtonTitles:nil];

[alertView show];

[alertView release];
return;
}

GKAchievementViewController *achievements = [[GKAchievementViewController alloc]
init];
if (achievements != NULL)
{
    achievements.achievementDelegate = unityviewController;
    [unityviewController presentModalViewController: achievements animated: YES];
}
}

```

步骤 05 在 Xcode 中编译当前工程,然后将编译的.a 和在 UGameManager.h 文件复制到 Unity 的 iOS 目录下。

将 UGameManager.h 也复制到 Unity 中是因为 unityviewController 这个属性,它是 Unity 的显示控制器,但我们没有在当前代码中获取到 Unity 的显示控制器对象,这里要使用一点技巧。当 Unity 将游戏导出为 Xcode 工程后,会自动调用一些预设的代码,这其中就包括获取 Unity 显示控制器的代码,如果我们将此代码文件也复制到 iOS 目录内,即可替换默认的预设代码,实现我们自己的目的。

步骤 06 在 Mac 计算机上选择 Macintosh HD,选择 Applications,找到 Unity 执行文件,单击右键选择【Show Package Contents】,如图 9-29 所示。

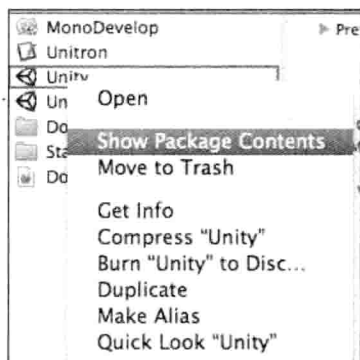


图 9-29 显示 Unity 资源

步骤 07 在 Contents 目录中打开 PlaybackEngines/iPhonePlayer/iPhone-Trampoline/Classes 目录,找到 iPhone_View.mm 文件,将其复制到 Unity 的 iOS 目录内。

步骤 08 打开 iPhone_View.mm,添加头文件:

```
#import "UGameManager.h"
```


步骤 09 在 iPhone_View.mm 中查找 CreateViewHierarchy 函数, 在 _viewController.view = _view 语句后添加获取 Unity 显示控制器的代码:

```
_viewController = [[UnityViewController alloc] init];
_viewController.wantsFullScreenLayout = TRUE;
_viewController.view = _view;

// 在这里添加代码, 创建 UGameManager 实例, 并获得 Unity 的显示控制器
[UGameManager instance].unityviewController=_viewController;
```

步骤 10 在 iPhone_View.mm 中查找 @implementation UnityViewController, 在对应的 @end 之前添加两个用于关闭 Game Center 窗口的回调函数, 如果不添加这两个函数, 关闭 Game Center 窗口将可能会导致程序崩溃, 代码如下:

```
// 当关闭分数排行榜
- (void) leaderboardViewControllerDidFinish: (GKLeaderboardViewController
*)viewController
{
    [self dismissModalViewControllerAnimated:YES];

    [viewController release];
}

// 当关闭成就界面
- (void) achievementViewControllerDidFinish: (GKAchievementViewController
*)viewController
{
    [self dismissModalViewControllerAnimated:YES];

    [viewController release];
}
@end
```

步骤 11 在 Unity 中打开 UnityiOSKit.cs, 我们将在这个脚本中添加代码导入前面创建的 Objective-C 代码:

```
[DllImport ("__Internal")]
private static extern bool _isGameCenterAvailable();

[DllImport ("__Internal")]
private static extern void _authenticateLocalPlayer();

[DllImport ("__Internal")]
private static extern void _reportScore(string identifier,int hiscore );
```

```

[DllImport ("__Internal")]
private static extern void _reportAchievement(string identifier ,float percent);

[DllImport("__Internal")]
private static extern void _showLeaderboard(string leaderboard);

[DllImport ("__Internal")]
private static extern void _showAchievements();

// Game Center
static public bool IsGameCenterAvailable()
{
    if ( Application.platform==RuntimePlatform.IPhonePlayer)
        return _isGameCenterAvailable();

    return false;
}

static public void AuthenticateLocalPlayer()
{
    if ( Application.platform==RuntimePlatform.IPhonePlayer)
        _authenticateLocalPlayer();
}

static public void ReportScore( string identifier, int hiscore )
{
    if ( Application.platform==RuntimePlatform.IPhonePlayer)
        _reportScore(identifier,hiscore);
}

static public void ReportAchievement(string identifier ,float percent)
{
    if ( Application.platform==RuntimePlatform.IPhonePlayer)
        _reportAchievement(identifier,percent);
}

static public void ShowLeaderboard(string identifier)
{
    if (Application.platform == RuntimePlatform.IPhonePlayer)
        _showLeaderboard(identifier);
}

static public void ShowAchievements()
{

```

```

        if ( Application.platform==RuntimePlatform.IPhonePlayer)
            _showAchievements();
    }

```

步骤 12 在 Unity 中打开 iOSAPP.cs, 我们将在各个按钮中添加代码显示 Game Center 界面, 上传分数和成就的功能:

```

void Start () {

    UnityiOSKit.Start();
    UnityiOSKit.ShowWarningBox("Hello","World");

    // 检查 Game Center 是否可用
    if ( UnityiOSKit.IsGameCenterAvailable())
    {
        // 认证 Game Center 用户
        UnityiOSKit.AuthenticateLocalPlayer();
    }
}

if (GUI.Button( m_showGameCenterPos, "显示 Game Center\n每按一次加一分") )
{
    // 显示成就界面
    UnityiOSKit.ShowAchievements();

    // 显示排行分数排行榜, 需要填写排行榜 ID
    //UnityiOSKit.ShowLeaderboard("com.xxx.xxx.xxx");
    m_score++;
    m_achiv++;
}

if (GUI.Button(m_uploadGameCenterPos, "上传分数到 Game Center"))
{
    // 上传分数, 需要填写排行榜 ID
    UnityiOSKit.ReportScore("com.xxx.xxx.xxx",m_score);
}

if (GUI.Button(m_uploadGameCenterPos2, "上传成就到 Game Center"))
{
    // 上传成就 最高 100 点, 完成成就
    UnityiOSKit.ReportAchievement("com.xxx.xxx.xxx",m_achiv);
}

```

在 iOS 设备上测试程序, 效果如图 9-30 所示。



图 9-30 显示 Game Center 界面

本节的示例工程文件保存在资源文件目录 chapter09_GameCenter, 其中 UGameManager 目录内是 Xcode 工程文件, UnityiOSProject 目录内是最后的 Unity 工程文件。

9.8 集成内消费系统

游戏内消费是近些年一种热门的商业模式, 游戏本身是免费的, 但在游戏中提供很多虚拟服务, 可以通过游戏内消费去换取。

9.8.1 设置内消费

与 Game Center 一样, 首先我们需要到苹果公司的 <https://itunesconnect.apple.com> 网站上去设置一番。

- 步骤 01** 登录 itunesconnect 网站后, 选择 Manage Your Applications, 选择游戏, 然后选择 Manage In-App Purchases 进入内消费设置页面。
- 步骤 02** 在内消费设置页面中选择 Create New, 创建一个新的内消费。
- 步骤 03** 内消费有多种类型, 我们这里选择 Consumable 形式, 这种模式可反复消费, 比如虚拟货币, 消耗没有后仍可以再次购买。如果是永久性的解锁一个功能, 比如解锁关卡, 更适合 Non-Consumable 形式, 用户只需要购买一次, 可永久享有一次购买的成果。
- 步骤 04** 每个内消费都对应一个 ID, 形式类似 com.xxx.xxx.xxx, 这和 Game Center 中创建的 ID 是一样的, 并不能与其他 ID 重复, 此外还要在这里决定内消费的价格, 最低 0.99 美金消费一次。
- 步骤 05** 设置完内消费的内容后进入编辑游戏说明的页面, 找到 In-App Purchases 选项, 确定选中设置好的内消费, 这些操作必须在执行 Ready to Submit 之前进行, 在提交游戏后, 苹果公司会对内消费的设置和操作进行审核。

步骤 06 因为我们在测试内消费的时候并不想真的去消费,这时要使用到一个用于测试的账号,在 itunesconnect 网站首页上选择 Manage Users 即可进行设置。

9.8.2 实现内消费

本节将继续 9.7 节的内容,在 Xcode 和 Unity 工程中添加内消费部分的代码。

步骤 01 在 Mac 上使用 Xcode,打开 9.7 节完成的 Xcode 工程。

步骤 02 打开 UGameManager.h,添加内消费功能的头文件:

```
#import <StoreKit/StoreKit.h>
```

步骤 03 为 UGameManager 添加内消费交易的代理,并增加一个判断是否正在交易的属性:

```
@interface UGameManager : NSObject<SKPaymentTransactionObserver>
{
    // ...
    // 用来判断是否正在交易中
    BOOL isProcessPayments;
}

@property (nonatomic) BOOL isProcessPayments;
```

步骤 04 为 UGameManager.h 中继续添加内消费功能的函数声明:

```
// Objective-C
// 初始化内消费
- (void) iniStoreKit;

// 是否可以内消费
- (BOOL) canProcessPayments;

// 购买产品
- (void) purchaseItem:(NSString*) identifier;

// 结束交易
- (void) completeTransaction: (SKPaymentTransaction *)transaction;

// 重置交易
- (void) restoreTransaction: (SKPaymentTransaction *)transaction;

// 交易失败
- (void) failedTransaction: (SKPaymentTransaction *)transaction;

// 记录交易
- (void) recordTransaction: (SKPaymentTransaction *)transaction;
```



```
// 提供产品
- (void) provideContent: (NSString*)identifier;

// 最后在 Unity 中只需要调用这 4 个 C 函数
void _iniStore(void);
BOOL _canProcessPayments(void);
BOOL _isProcessPayments(void);
void _purchaseItem( char* identifier);
```

步骤 05 打开 UGameManager.m 中添加内消费功能的实现代码:

```
@synthesize isProcessPayments;

// 初始化内消费
-(void) iniStoreKit
{
    [[SKPaymentQueue defaultQueue] addTransactionObserver:self];
}

// 判断是否可以内消费
-(BOOL) canProcessPayments
{
    if ([SKPaymentQueue canMakePayments])
        return YES;
    else
        return NO;
}

// 输入内消费商品 ID, 向服务器发送购买请求
-(void)purchaseItem:(NSString*) identifier
{
    isProcessPayments=YES;
    SKPayment *payment = [SKPayment paymentWithProductIdentifier:identifier];
    [[SKPaymentQueue defaultQueue] addPayment:payment];
}

// 内消费回调函数, 分别返回购买完成, 购买失败和重置交易几种状态
- (void)paymentQueue:(SKPaymentQueue *)queue updatedTransactions:(NSArray
*)transactions
{
    for (SKPaymentTransaction *transaction in transactions)
    {
```

```

switch (transaction.transactionState)
{
    case SKPaymentTransactionStatePurchased:
        [self completeTransaction:transaction];
        break;
    case SKPaymentTransactionStateFailed:
        [self failedTransaction:transaction];
        break;
    case SKPaymentTransactionStateRestored:
        [self restoreTransaction:transaction];
    default:
        break;
}
}
}

// 购买完成状态
- (void) completeTransaction: (SKPaymentTransaction *)transaction
{
    isProcessPayments=NO;

    // 执行记录交易
    [self recordTransaction: transaction];

    // 购买成功, 执行交付商品
    [self provideContent: transaction.payment.productId];

    // 执行结束交易
    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
}

// 重置交易状态
- (void) restoreTransaction: (SKPaymentTransaction *)transaction
{
    isProcessPayments=NO;

    [self recordTransaction: transaction];
    [self provideContent: transaction.originalTransaction.payment.productId];
    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
}

```

```

// 购买失败状态
- (void) failedTransaction: (SKPaymentTransaction *)transaction
{
    isProcessPayments=NO;

    if (transaction.error.code != SKErrorPaymentCancelled)
    {
        //购买出现错误并通知 Unity

        UnitySendMessage("UGameObject","iOSIAPCallback",[transaction.error.localizedDescription UTF8String]);
    }
    else{
        //处理用户取消了购买
    }

    [[SKPaymentQueue defaultQueue] finishTransaction: transaction];
}

// 记录交易
- (void) recordTransaction: (SKPaymentTransaction *)transaction
{
    //...
}

// 购买成功后通知 Unity 提供商品
- (void) provideContent: (NSString*) identifier
{
    UnitySendMessage("UGameObject"," iOSIAPCallback ",[identifier UTF8String]);
}

```

当我们需要通知 Unity 交易成功或失败时, 使用了一个叫 `UnitySendMessage` 的函数, 其中第一个参数是指 Unity 中游戏体的名字, 第二个参数是游戏体脚本中函数的名字, 第三个参数是一个字符串, 如果购买成功, 则将商品的 ID 写入这个字符串发送给 Unity 的 `iOSIAPCallback` 函数, 根据 ID 提供虚拟商品。

步骤 06 在 Xcode 中重新编译.a 文件, 复制到 Unity 工程的 iOS 目录内。

接下来的工作回到 Unity 中。

步骤 01 在 Mac 上使用 Unity, 打开 9.7 节完成的 Unity 工程。

步骤 02 打开脚本 `UnityiOSKit.cs`, 添加代码导入内消费功能:


```

[DllImport ("__Internal")]
private static extern void _iniStore();

[DllImport ("__Internal")]
private static extern bool _canProcessPayments();

[DllImport ("__Internal")]
private static extern bool _isProcessPayments();

[DllImport ("__Internal")]
private static extern void _purchaseItem( string identifier);

// 增加一个初始化函数,一次性初始化 Game Center 和内消费的功能
static public void Init()
{
    UnityiOSKit.Start();

    // 初始化 Game Center
    if (UnityiOSKit.IsGameCenterAvailable())
        UnityiOSKit.AuthenticateLocalPlayer();

    // 初始化内消费
    UnityiOSKit.IniIAPStore();
}

// 初始化内消费功能
static public void IniIAPStore()
{
    if ( Application.platform==RuntimePlatform.IPhonePlayer)
        _iniStore();
}

// 是否可以处理内消费
static public bool CanProcessPayments()
{
    if ( Application.platform==RuntimePlatform.IPhonePlayer)
        return _canProcessPayments();

    return false;
}

// 是否正在处理交易
static public bool IsProcessPayments()

```

```

    {
        if ( Application.platform==RuntimePlatform.IPhonePlayer)
            return _isProcessPayments();

        return false;
    }

    // 向服务器发出请求购买内消费商品
    static public void PurchaseItem(string identifier)
    {
        if ( Application.platform==RuntimePlatform.IPhonePlayer)
        {
            _purchaseItem( identifier );
        }
    }
}

```

步骤 03 打开脚本 iOSAPP.cs, 添加请求购买内消费商品的功能:

```

void Start () {

    // 将游戏体的名称改为 UGameObject, 内消费结束后会向这个游戏体发出消息
    name="UGameObject";

    UnityiOSKit.Init();
}

// 选择这个按钮发起购买请求
if (GUI.Button(m_IAPPos, "内消费购买")){
    UnityiOSKit.PurchaseItem("com.xxx.xxx");
}

// 购买结束
void iOSIAPCallback(string msg)
{
    // 如果商品 ID 为 com.xxx.xxx, 表示此商品购买成功
    if (msg.CompareTo("com.xxx.xxx") == 0)
    {
        // 更新商品
        m_point+=100;

        // 显示对话框表示购买成功
        UnityiOSKit.ShowWarningBox("谢谢","购买成功!");
    }
    else // 显示对话框表示交易失败
        UnityiOSKit.ShowWarningBox("购买未能成功!", msg);
}

```


步骤 04 在 Unity 中使用 Build 导出 Xcode 工程, 然后使用 Xcode 打开这个工程, 选择 Build Phases, 选择  添加 StoreKit.framework, 这是针对内消费功能需要的模块, 如图 9-31 所示。



图 9-31 添加 StoreKit.framework

步骤 05 在 Xcode 中编译程序, 在 iOS 设备上测试, 最后效果如图 9-32 所示。



图 9-32 测试内消费购买

本节的示例工程文件保存在资源文件目录 chapter09_IAP, 其中 UGameManager 是 Xcode 工程, UnityiOSProject 是 Unity 工程。

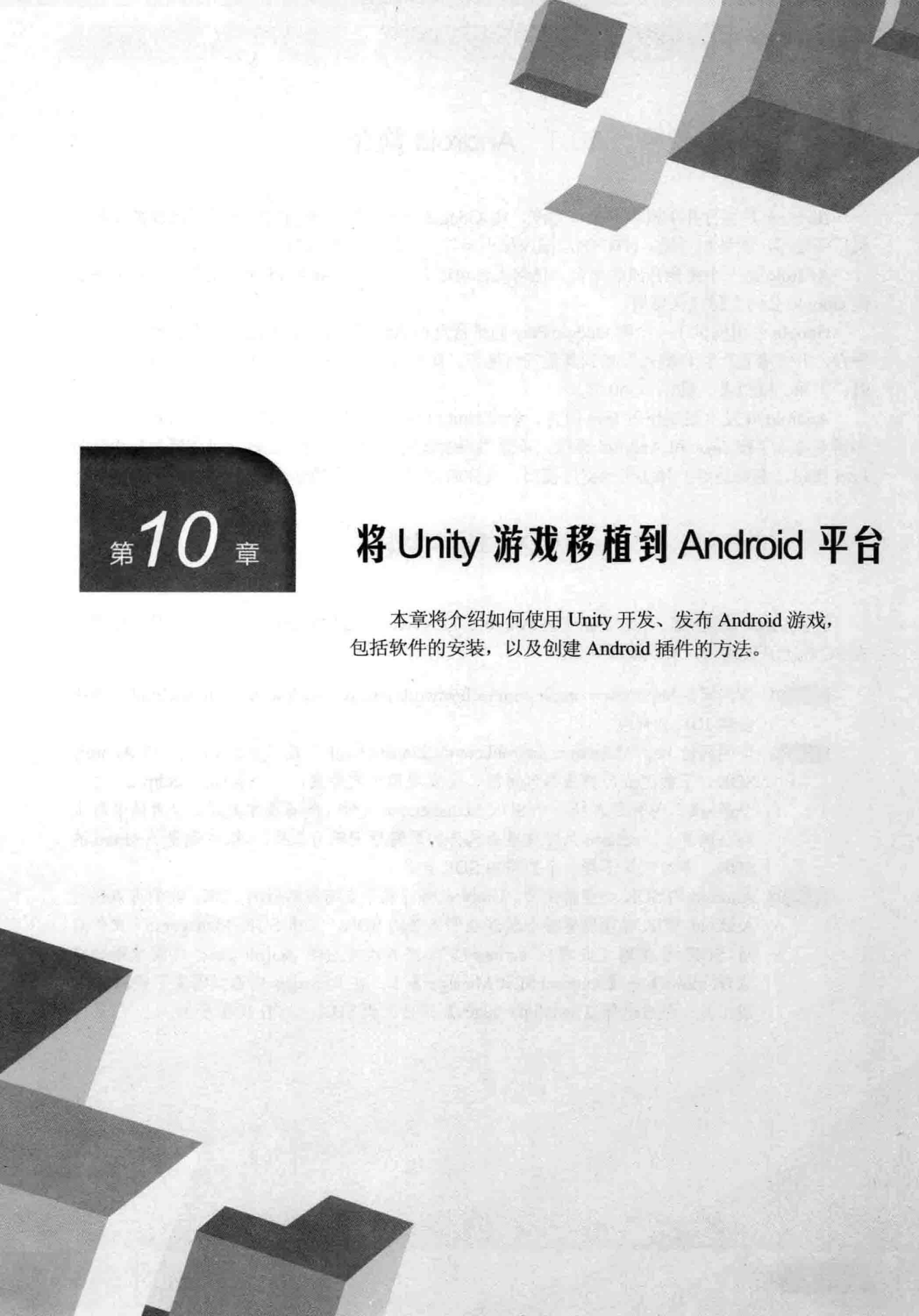
9.9 本地存储位置

如果我们需要将游戏记录保存在 iOS 设备本地, 只能将其保存在 iOS 设备上一个叫 Documents 的文件目录内, 比如我们要保存一个叫 sav.dat 的文件, 其位置即是:

```
// 在不同平台, Application.persistentDataPath 指向的位置不同  
string dir=Application.persistentDataPath+"/Documents/sav.dat";
```

小 结

本章详细介绍了 iOS 平台的开发资格申请流程, 同时也介绍了如何配置开发环境, 测试、发布使用 Unity 开发的 iOS 游戏。本章最后通过 Game Center 和内消费功能的实例介绍了如何为 Unity 编写 iOS 插件, 使 Unity 游戏与 iOS 平台更好地结合在一起。



第 10 章

将 Unity 游戏移植到 Android 平台

本章将介绍如何使用 Unity 开发、发布 Android 游戏，包括软件的安装，以及创建 Android 插件的方法。

10.1 Android 简介

Android 是运行在手机上的操作系统, 由 Google 公司研发, 因为其良好的开放性被很多手机厂商使用, 国外如三星、HTC 等, 国内如小米等, 市场占有率很高。

Android 是一个完全开放的平台, 任何人都可以开发、发布 Android 应用或游戏, 而不需要向 Google 公司支付任何费用。

Google 公司提供了一个叫 Google Play 的平台发布 Android 游戏, 不过这并不是唯一的发布平台, 开发者也可以将游戏发布到其他任何地方, 国内发布 Android 游戏的平台非常多, 比如 91、当乐、豌豆荚、腾讯、360 等。

Android 开发主要是使用 Java 语言。使用 Unity 可以开发出高品质的 Android 游戏, 而几乎不需要深入了解 Java 和 Android 系统。不过当将游戏发布到相应的平台时, 仍需要编写少量的 Java 代码, 主要是用于对接消费支付接口, 具体的方式本章都会相应介绍。

10.2 软件安装

开发 Android 游戏, 一定要先安装 Android 的 SDK, 同时也要安装一些开发工具。这里以在 PC 机上开发为例, 安装步骤如下:

步骤 01 访问网址 <http://www.oracle.com/technetwork/java/javase/downloads/index.html> 下载并安装 JDK 软件包。

步骤 02 访问网址 <http://developer.android.com/sdk/index.html> 下载 Windows 版本的 Android SDK。下载完成后将压缩包解压, 会发现两个文件夹, 一个名称是 eclipse, 另一个是 sdk, 另外还包括一个 SDK Manager.exe 文件 (随着版本更新, 文件的名称也许会改变)。eclipse 文件夹里面包含的是编写代码的工具, sdk 中则是 Android 的 SDK, 不过它并不是一个完整的 SDK 包。

步骤 03 Android 的 SDK 会经常更新, Unity 也有可能不支持最新版的 SDK, 我们可以通过 Android SDK 管理器直接下载到需要的 SDK。双击 SDK Manager.exe 文件启动 SDK 管理器 (或者在 eclipse 文件夹内双击启动 eclipse.exe, 在菜单栏选择 **【Windows】** → **【Android SDK Manager】**)。在 Packages 中选择需要下载的 SDK 或工具, 然后选择 **【Install packages】** 即可下载 SDK, 如图 10-1 所示。

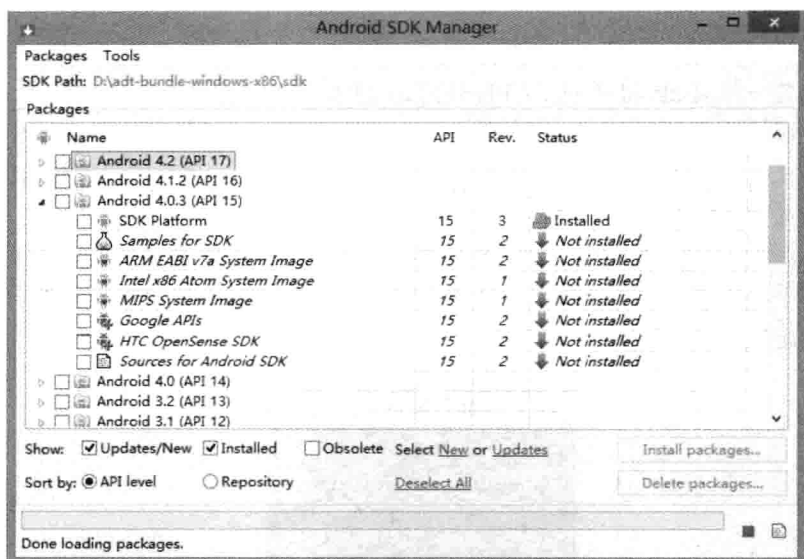


图 10-1 下载 SDK

步骤 04 启动 Unity，在菜单栏选择【Edit】→【Preferences】，选择【External Tools】→【Android SDK Location】指定 Android SDK 的位置，如图 10-2 所示。

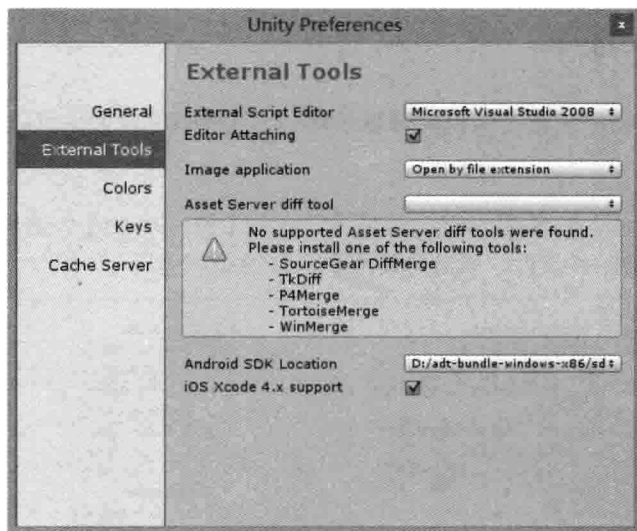


图 10-2 指定 Android SDK 的位置

10.3 运行 Android 游戏

10.3.1 设置 Android 手机

几乎任何品牌的 Android 手机都可以用来测试开发中的游戏，但需要一些简单的设置。不同

品牌的手机其设置过程有可能略有不同,下面以一款 HTC 手机为例,设置步骤如下:

- 步骤 01** 准备一部 Android 手机, CPU 最好是 ARMv7 架构的。
- 步骤 02** 将手机的驱动程序安装在计算机上。有一些手机的驱动程序带有如 Sync 之类的工具,运行这些工具有时会使 Unity 找不到测试手机,如果遇到这样的问题,建议将其关闭或卸载。
- 步骤 03** 在手机上选择【USB 调试】启动调试模式,为了方便调试,选择【保持唤醒状态】使手机在充电时不会休眠,如图 10-3 所示。注意,不同型号的手机设置 USB 调试的选项可能不同。

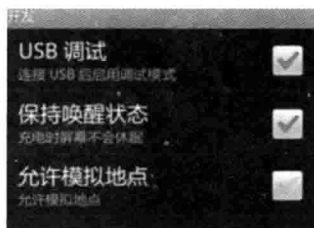


图 10-3 设置调试模式

- 步骤 04** 将手机连接到计算机,有些老式手机一定要选择“充电”模式。

10.3.2 安装驱动程序

如果 Android 手机或平板没有随机附带驱动程序,这种情况我们也可以使用 Google 提供的驱动程序,方法如下:

- 步骤 01** 启动 Android SDK 管理器,选择【Google USB Driver】,将驱动程序下载,它被保存在 SDK 目录内,稍后会用到它,如图 10-4 所示。

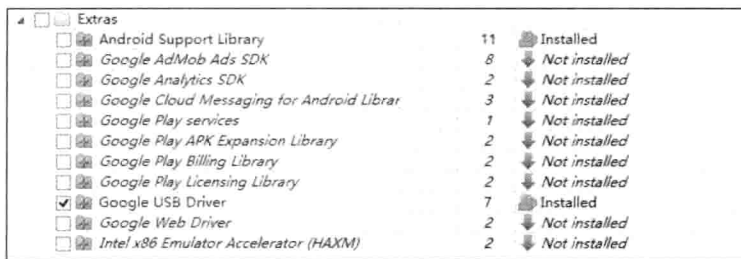


图 10-4 下载 USB 驱动

- 步骤 02** 将 Android 设备连接到计算机,在设备管理器中找到该设备,选择为这个设备更新驱动程序,并指定驱动的存储位置,如图 10-5 所示。

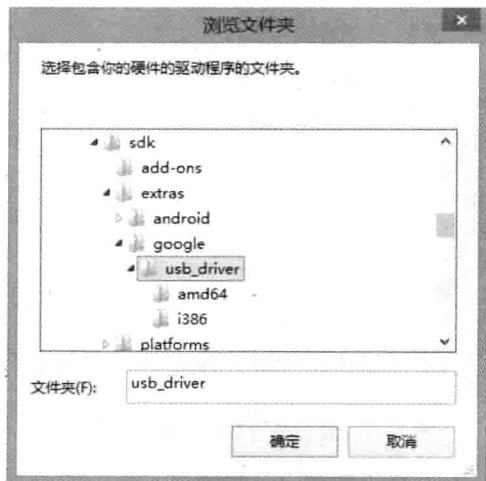


图 10-5 选择 USB 驱动的存储位置

如果从 Google 下载的驱动程序也不适合当前手机，可以使用一些如手机助手之类的软件，自动为手机安装驱动程序。

10.3.3 设置 Android 游戏工程

在 Unity 内开发 Android 游戏与开发 PC 游戏没有什么不同，只是最后我们需要将游戏工程从当前平台转换到 Android 平台，然后做一些适当的设置，就可以将游戏运行在 Android 手机上了。

接下来，我们将把本书第 2 章的太空射击游戏由 PC 游戏转为 Android 游戏，并输出到手机上进行测试，步骤如下：

- 步骤 01** 启动 Unity 并打开资源文件目录内的 chapter02_ShootingGame 工程。在菜单栏选择【File】→【Build Settings】打开 Build Settings 窗口，在 Platform 中选择 Android，选择【Switch Platform】将当前开发平台转为 Android 平台，如图 10-6 所示。

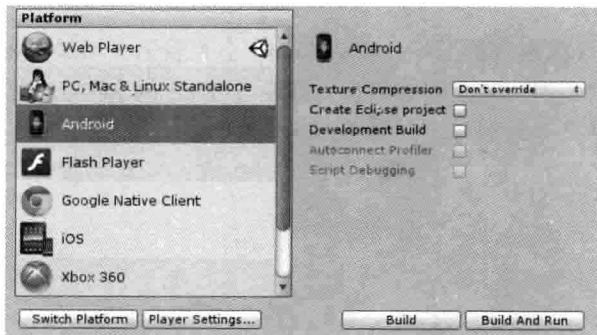


图 10-6 切换到 Android 平台

- 步骤 02** 在 Build Settings 窗口选择【Player Settings】（或在菜单栏选择【Edit】→【Project Settings】→【Player】），Inspector 窗口会显示出创建 Android 游戏的相关设置。
- 步骤 03** 在 Company Name 和 Product Name 中分别设置开发公司和游戏的名字，在 Default

Icon 中设置游戏的图标, 如图 10-7 所示。



图 10-7 设置公司名称和游戏名称

步骤 04 在 Default Orientation 中设置游戏的横竖屏显示, 【Portrait】表示竖屏, 【Landscape】表示横屏, 选择 【Auto Rotation】可以自动旋转屏幕方向。这里我们设置为竖屏并自动旋转, 如图 10-8 所示。

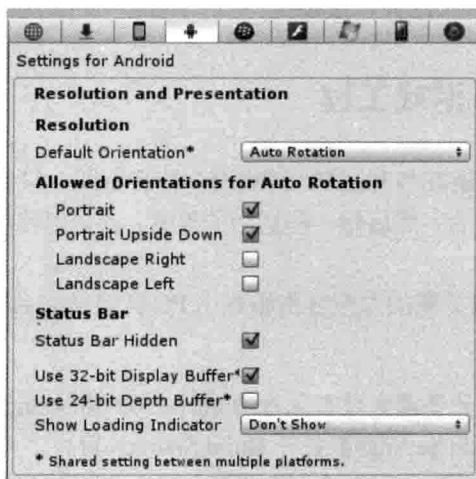


图 10-8 设备屏幕旋转方向

步骤 05 在 Icon 中设置游戏的图标。默认 Unity 会为我们自动适配各种大小不同的图标, 但自动缩放图片可能会有锯齿, 选中 【Override for Android】, 可以指定已经准备好的各种尺寸的图标, 如图 10-9 所示。

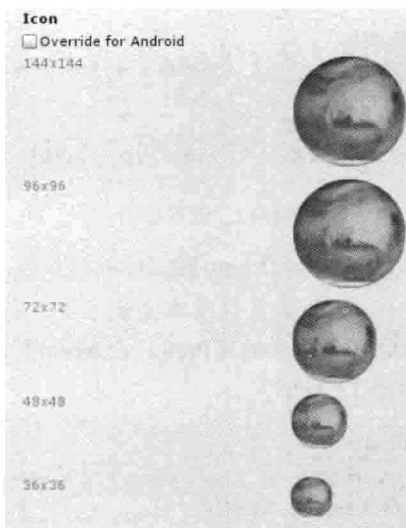


图 10-9 图标

- 步骤 06** 默认启动 Unity 游戏时将会看到 Unity 的 Logo，在 Splash Image 中指定一张图片即可替换默认的启动画面，在 Splash scaling 中可以设置启动图片的缩放模式，如图 10-10 所示。

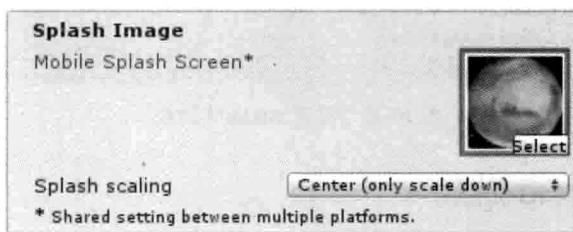


图 10-10 设置启动画面

- 步骤 07** 在 Other Settings 中有很多选项需要设置。Static Batching 会将静态模型合并提高效率，根据游戏中 3D 模型不同的制作方式，有时会使游戏的包变小或变大，如图 10-11 所示。

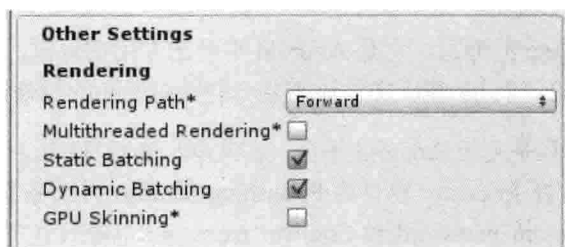


图 10-11 其他设置

- 步骤 08** 设置 Bundle Identifier，它是游戏的标识，格式是 com.company.appname。在 Bundle Version 中输入版本号，在 Minimum API Level 中设置游戏所支持的最小 API 版本，比如设置为 Android 2.3.3，那么安装有 Android 2.3.3 或更新版本的手机都可以运行这个游戏，如图 10-12 所示。

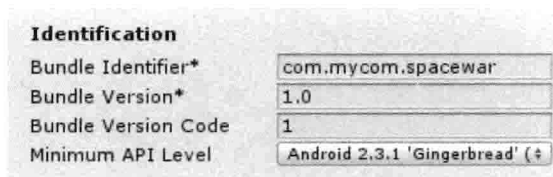


图 10-12 设置标识

步骤 09 在 Device Filter 中只有【ARMv7 only】选项,也就是说 Unity 游戏不能运行在 ARMv6 架构的老式手机上(这类手机已经基本淘汰了)。在 Install Location 中提供了几种游戏安装方式,通常保持默认的【Prefer External】即可,游戏会被优先安装到外部存储卡中,如图 10-13 所示。



图 10-13 设置 Android 游戏

10.3.4 测试 Android 游戏

将手机连接到计算机,然后在 Unity 的菜单栏选择【File】→【Build & Run】,选择 APK 文件的保存路径,略等片刻,游戏将运行在 Android 手机上。

10.3.5 发布 Android 游戏

APK 是 AndroidPackage 的缩写,它是 Android 平台的专用文件包,我们的游戏最后将会编译成为 APK 格式发布出去。发布 APK 文件还需要一些针对发布的额外设置,步骤如下:

步骤 01 确定当前工程是处于 Android 平台,在菜单栏选择【Edit】→【Project Settings】→【Player】,在 Inspector 窗口的 Publishing Settings 内选择【Create New Keystore】,然后在 Keystore password 和 Confirm password 中输入自定义的密码(密码一定要记下来),选择【Browse Keystore】设置.keystore 文件的保存位置,如图 10-14 所示。

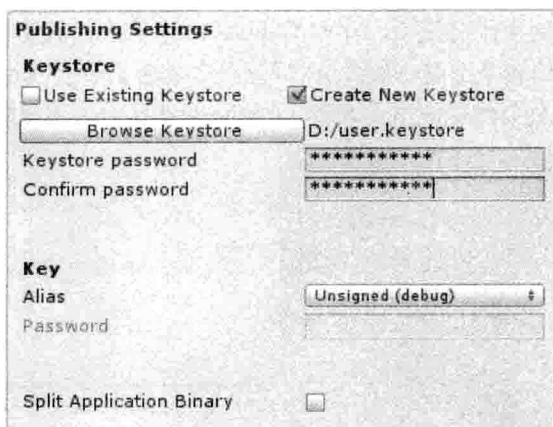


图 10-14 发布设置

步骤 02 设置 Alias，选择【Unsigned(debug)】，选择【Create a new key】打开创建 keystore 文件的窗口，如图 10-15 所示。

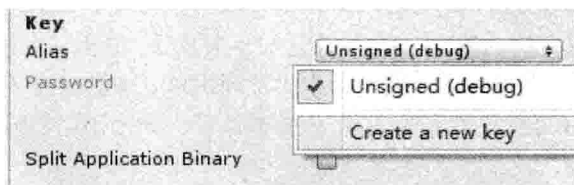


图 10-15 发布设置

步骤 03 在 Alias 中输入游戏的别名，在 Password 和 Confirm 中输入密码（密码一定要记下来），在 Validity(years)中输入游戏的使用年限，在 Organization 中输入公司名称，最后选择【Create key】创建.keystore 文件，如图 10-16 所示。

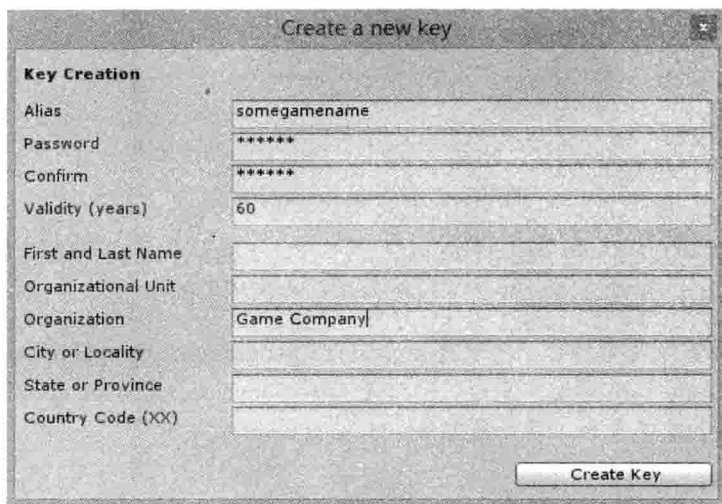


图 10-16 创建.keystore 文件

步骤 04 选择【Use Existing Keystore】，选择【Browse Keystore】打开之前保存的.keystore 文件并在下面输入密码。在 Alias 中选择对应的游戏别名并输入密码。有些发布平

台对安装包的大小有限制,比如 Google Play 商店,就需要使用 Split Application Binary 功能将游戏分包,如图 10-17 所示,命名中带有_single 的是一个独立的安装包,其他两个文件是使用了 Split Application Binary 创建的安装包。



图 10-17 使用.keystore 文件

步骤 05 在菜单栏选择【File】→【Build Settings】,然后在 Build Settings 窗口选择【Build】创建用于发布的 APK 文件。如果这一步失败了,原因通常是没有正确指定 Android SDK 或 SDK 的版本。如果是提示找不到 JDK,请尝试在环境变量的用户变量中添加 JAVA_HOME,并指定 JDK 的安装路径,如图 10-18 所示。

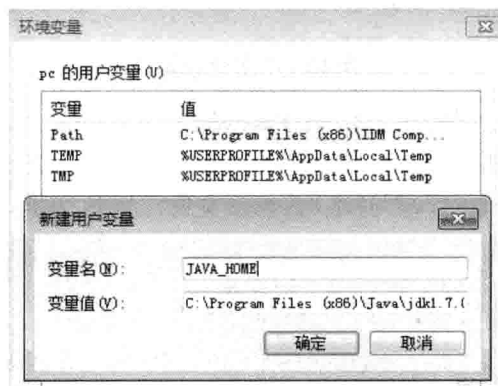


图 10-18 设置 JAVA 环境变量

这个 APK 文件可以发布到任何 Android 运营平台,下面列举了几个比较大的运营平台,国内的运营平台非常多,开发者通过网络可以很容易查找到相关的信息。

- Google 的官方发布平台,网址是 <https://play.google.com/apps/publish/>;
- 亚马逊 Kindle Fire 的发布平台,网址是 <https://developer.amazon.com/welcome.html>;
- 腾讯无线的发布平台,网址是 <http://dev.qq.com/>。

10.4 触屏操作

当将 Unity 游戏运行到 iOS 或 Android 设备上时,桌面系统中的鼠标左键操作可以自动变为

手机屏幕上的触屏操作，但鼠标操作无法实现一些特有的触屏操作，比如多点触屏等。在 Unity 的 Input 类中，除了包括桌面系统的各种输入功能，也包括专门针对手机触屏的各种功能，下面将通过一个简单的例子来说明如何实现触屏、多点触屏等。本节的示例同样适用于 iOS 平台。

- 步骤 01** 打开资源文件目录 chapter10_Android_touch_Start 下的 Unity 工程，在这个工程的场景中只有一个角色模型，我们将使用触屏操作在 Android 设备上移动摄像机，并使用多点触屏缩放摄像机视图。
- 步骤 02** 打开脚本 AndroidTouch.cs，这是预先创建好的脚本，但是里面几乎是空白的，什么也没有，我们将在这个脚本中添加触屏操作的代码，控制摄像机的运动。
- 步骤 03** 首先添加一个 m_screenpos 属性，并在 Start 函数中启动多点触屏。我们使用 m_screenpos 记录手指第一次触屏的位置，然后再根据手指离开屏幕时的位置判断手指的划动方向。

```
public class AndroidTouch : MonoBehaviour {

    // 记录手指触屏的位置
    Vector2 m_screenpos = new Vector2();
    // 摄像机移动速度
    float m_speed = 0.1f;

    void Start () {
        // 允许多点触屏
        Input.multiTouchEnabled = true;
    }
}
```

- 步骤 04** 在 AndroidTouch.cs 中添加函数 MobileInput，使用 Input.touchCount 判断有几个手指触摸到屏幕，当只有一个手指触屏时，我们可以通过 Began 和 Moved 状态判断手指是否刚刚触屏或是移动中，当手指移动的时候，我们按其移动的方向和距离移动摄像机，代码如下：

```
void MobileInput()
{
    if (Input.touchCount <= 0)
        return;

    if (Input.touchCount == 1) // 1 个手指触摸屏幕
    {
        // 开始触屏
        if (Input.touches[0].phase == TouchPhase.Began)
        {
            // 记录手指触屏的位置
            m_screenpos = Input.touches[0].position;
        }
        // 手指移动
    }
}
```

```

else if (Input.touches[0].phase == TouchPhase.Moved)
{
    //移动摄像机
    Camera.main.transform.Translate(
        new Vector3(Input.touches[0].deltaPosition.x * m_speed,
            Input.touches[0].deltaPosition.y * m_speed, 0));
}

// 手指离开屏幕
if (Input.touches[0].phase == TouchPhase.Ended &&
    Input.touches[0].phase != TouchPhase.Canceled)
{
    Vector2 pos = Input.touches[0].position;

    // 手指水平移动
    if (Mathf.Abs(m_screenpos.x - pos.x) > Mathf.Abs(m_screenpos.y - pos.y))
    {
        if (m_screenpos.x > pos.x){
            //手指向左划动
        }
        else{
            //手指向右划动
        }
    }
    else // 手指垂直移动
    {
        if (m_screenpos.y > pos.y){
            //手指向下划动
        }
        else{
            //手指向上划动
        }
    }
}
}
}

```

当手指离开屏幕，我们根据最初触屏的位置和离开时的位置判断手指的移动方向，不过当前的代码并没有做什么实际的事情。在著名游戏 *Temple Run*（也是用 Unity 制作）中，就是用这个操作来决定主角拐弯时的方向和跳跃等。

步骤 05 当有多个手指触屏，我们通过判断手指的划动距离缩放摄像机视图，继续前面的代码，添加代码如下：

```

else if ( Input.touchCount >1 )
{

```

```
// 记录两个手指的位置
Vector2 finger1 = new Vector2();
Vector2 finger2 = new Vector2();

// 记录两个手指的移动距离
Vector2 mov1 = new Vector2();
Vector2 mov2 = new Vector2();

for (int i=0; i<2; i++ )
{
    // 获得手指 1 和 2 的触控信息
    Touch touch = Input.touches[i];

    if (touch.phase == TouchPhase.Ended )
        break;
    // 如果手指处于移动状态
    if ( touch.phase == TouchPhase.Moved )
    {
        float mov = 0;
        if (i == 0)
        {
            // 获得手指 1 的位置和移动
            finger1 = touch.position;
            mov1 = touch.deltaPosition;
        }
        else
        {
            // 获得手指 2 的位置和移动
            finger2 = touch.position;
            mov2 = touch.deltaPosition;
            // 获得手指移动距离
            if (finger1.x > finger2.x){
                mov = mov1.x;
            }
            else{
                mov = mov2.x;
            }

            if (finger1.y > finger2.y){
                mov+= mov1.y;
            }
            else{
                mov+= mov2.y;
            }
        }
    }
}
```



```
// 根据手指移动距离改变摄像机位置
Camera.main.transform.Translate(0, 0, mov * m_speed);
    }
}
}
```

步骤 06 在 Update 函数中执行 MobileInput 函数, 然后在 Android 设备上测试程序, 上下左右划动手指将会移动摄像机, 用两个手指推拉移动将会缩放摄像机视图。

步骤 07 由于触屏操作的代码不能在桌面系统上使用, 为了在桌面系统上预览效果, 我们通常还需要添加一些针对桌面系统的代码:

```
void DesktopInput()
{
    // 记录鼠标左键的移动距离
    float mx = Input.GetAxis("Mouse X");
    float my = Input.GetAxis("Mouse Y");

    if ( mx!= 0 || my !=0 )
    {
        //鼠标左键
        if (Input.GetMouseButton(0))
        {
            Camera.main.transform.Translate(new Vector3(mx, my, 0));
        }
    }
}
```

这个版本的桌面系统操作只是使用鼠标左键简单地移动摄像机, 并不能模拟多点触屏的操作, 不过我们可以使用别的操作代替, 如用鼠标滑轮操作实现缩放摄像机视图。

步骤 08 在 Update 函数中修改代码。UNITY_EDITOR、UNITY_IOS、UNITY_ANDROID 是 Unity 预设的预处理标识符, 当程序运行在相对应的平台, 相应的预处理标识符即为真。这里当程序运行在 iOS 或 Android 平台且不是运行在 Editor 内, 就调用函数 MobileInput 中的触屏操作, 否则调用桌面系统鼠标操作:

```
void Update () {
    // 按手机上的返回键退出当前程序
    if (Input.GetKeyUp(KeyCode.Escape))
        Application.Quit();
    #if !UNITY_EDITOR && ( UNITY_IOS || UNITY_ANDROID )
        MobileInput();
    #else
        DesktopInput();
    #endif
}
```

```
}
```

本节的示例工程文件保存在资源文件目录 chapter10_Android_touch。

10.5 从 eclipse 到 Unity

与 iOS 平台一样,理论上可以使用 Unity 完成一款 Android 游戏且不写一行 Android 平台相关代码,但如果我们需要调用 Android 平台专有的 API,那么就不得不接触一些 Android 平台相关的东西。

因为 Android 是一个开放的平台,因此各种类型的 Android 在线商店也特别多。每个平台都有自己的 SDK,如果想发布游戏到该平台则需要引用其 SDK 做一些对接工作,大部分情况都需要接入相应平台的消费功能。

大部分 Android 在线商店提供的 SDK 都是 Java 代码,使用 Unity 是无法直接与这些代码沟通的,因此我们将使用类似在 iOS 平台的做法为 Unity 编写 Android 平台的插件,不过不同平台的对接做法可能不同,因此没有一个统一的解决方案。

Unity 与 Android API 沟通的方法有很多种,本章将介绍比较常用的几种方法。我们可以在 eclipse 中编写好针对 Android 平台的功能,然后将其制作成库文件使用到 Unity 中;也可以将 Unity 工程导出为 Android 工程,然后在 eclipse 中继续修改游戏的内容。听起来前一种做法似乎更为习惯一些,就像在 iOS 平台那样,但不幸的是,前一种做法并不是总能有效地解决问题。

在 Unity 中可以直接使用 Java 库文件,其格式为.jar。首先我们要使用 eclipse 创建一个库文件工程,其 Activity (Android 的显示功能都来自 Activity) 必须继承自 Unity 的 Activity,然后输出.jar 格式的库文件.jar 文件必须被复制到 Unity 的指定工程目录 Plugins/Android 内,并需要重新定义 Unity 的 AndroidManifest.xml。下面我们将在 eclipse 中创建一个标准的 Android 对话框,将其输出为.jar 文件并使用到 Unity 中。

10.5.1 创建.jar 文件

首先我们将在 eclipse 中创建.jar 文件。

- 步骤 01** 在下载 Android SDK 文件包中找到 eclipse.exe, 双击启动 eclipse 编辑器。
- 步骤 02** 在 eclipse 菜单栏选择【File】→【New】→【Android Application Project】, 打开 Android 工程创建向导。
- 步骤 03** 注意 Package Name 的填写, 这个名称一定要与 Unity 工程中的 Bundle Identifier 一致, 我们这里用的名称是 com.project.helloworld, 如图 10-19 所示。

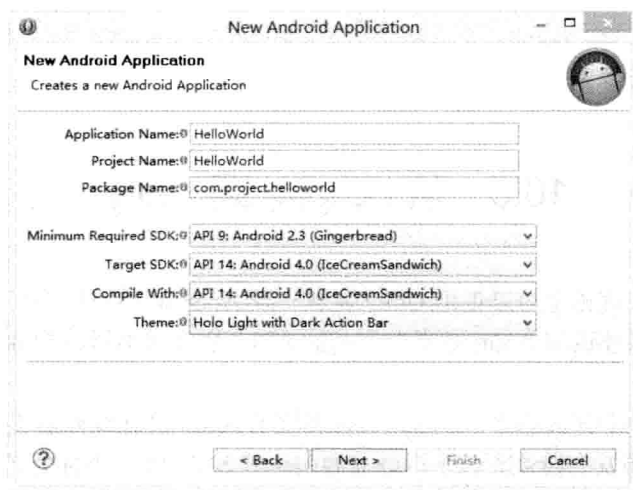


图 10-19 设置工程名称

步骤 04 选中【Mark this project as a library】使当前工程变成一个库工程，如图 10-20 所示。

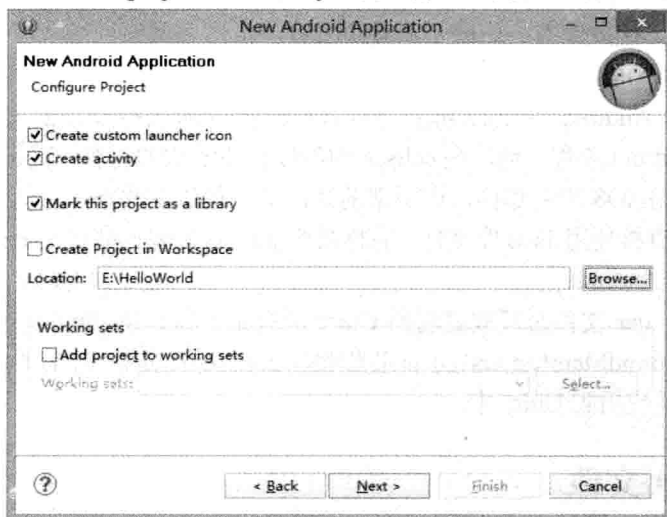


图 10-20 设置库工程

步骤 05 选择 BlankActivity 创建一个空白的 Activity，如图 10-21 所示。



图 10-21 创建空白的 Activity

步骤 06 确定 Activity 的名称，这个名称将来会在 Unity 中引用到，这里为默认的 MainActivity，如图 10-22 所示。

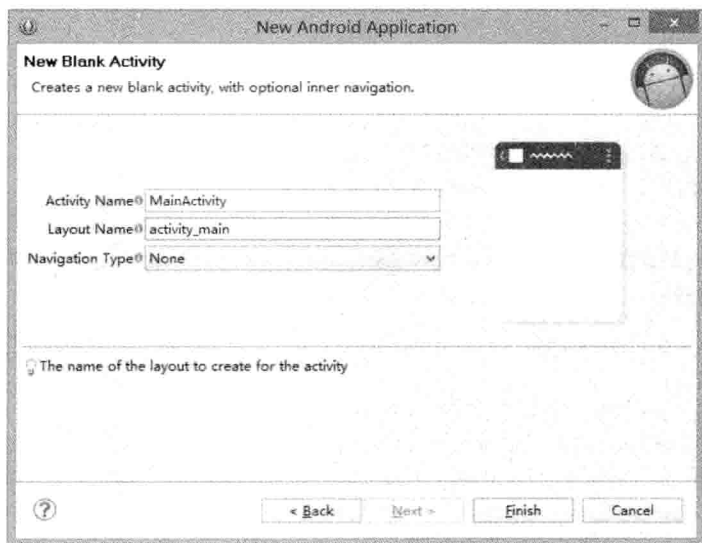


图 10-22 确定 Activity 的名称

步骤 07 创建好工程后，首先要引用 Unity 针对 Android 平台准备的一个 jar 库文件。在 eclipse 的菜单栏选择【Project】→【Properties】，然后选择 Java Build Path，选择 Add External JARs，浏览到 Unity 安装目录 Editor\Data\PlaybackEngines\androidplayer\bin，选择 classes.jar，将其添加到 eclipse 工程中，如图 10-23 所示。

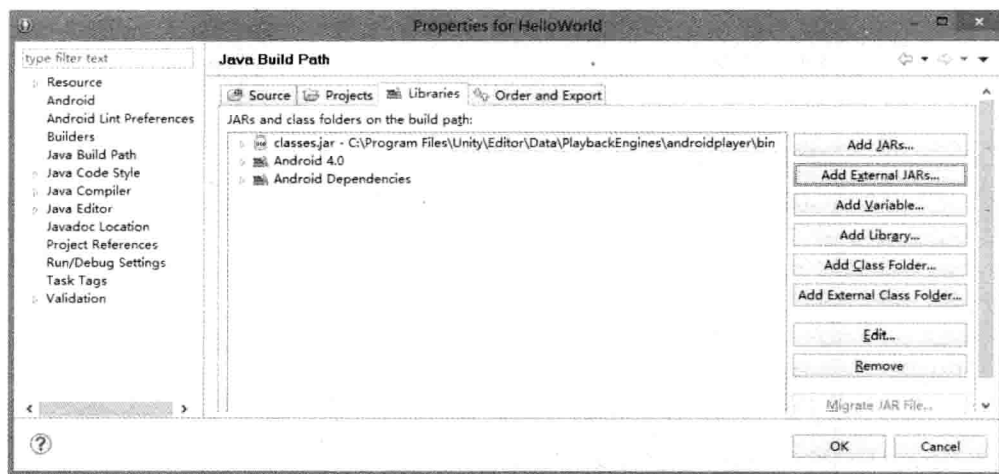


图 10-23 添加 Unity 的 classes.jar 到 eclipse 工程

步骤 08 打开 MainActivity.java, 修改代码如下:

```
package com.project.helloworld;

import android.os.Bundle;
import android.app.Activity;
import android.app.AlertDialog;

import com.unity3d.player.UnityPlayer;
import com.unity3d.player.UnityPlayerActivity;

// MainActivity 继承自 UnityPlayerActivity
public class MainActivity extends UnityPlayerActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
    }

    // 准备在 Unity 中调用的函数, 有两个参数
    protected void HelloWorld(final String title, final String content)
    {
        runOnUiThread(new Runnable() {
            public void run() {
                MakeDialog(title, content);
            }
        });
    }

    // 显示对话框
```



```

public void MakeDialog(String title, String content)
{
    AlertDialog.Builder builder = new AlertDialog.Builder(MainActivity.this);

    builder.setTitle(title)
        .setMessage(content)
        .setCancelable(false)
        .setPositiveButton("OK", null);

    builder.show();
}
}

```

首先,我们导入了 Unity 的 UnityPlayerActivity 包,使 MainActivity 继承自 UnityPlayerActivity。MakeDialog 函数用于显示一个标准的 Android 对话框。

HelloWorld 函数是准备在 Unity 中调用的一个函数,我们在这个函数中执行了 MakeDialog 函数显示对话框。注意,这里使用了 runOnUiThread 处理 UI 线程,如果不这么做,在显示 Android 对话框的时候程序可能会崩溃。

步骤 09 在菜单栏选择【Project】→【Clean】,然后在工程目录的 bin 里面找到 helloworld.jar, eclipse 中的工作到这里就完成了。

10.5.2 导入.jar 到 Unity

接下来,我们要将前面创建的 helloworld.jar 导入 Unity 工程中。

- 步骤 01** 打开资源文件目录 chapter10_Android_Plugin_Start 下的 Unity 工程,在这个工程的场景中只有一个预设的按钮,我们要做的就是按一下它,然后显示 Android 的对话框。
- 步骤 02** 将前面创建的 helloworld.jar 复制到 Unity 工程下的 Plugins/Android 目录内,同 iOS 平台一样,这个目录不能任意改变。
- 步骤 03** 在 Unity 的安装目录 Editor\Data\PlaybackEngines\androidplayer 内复制 AndroidManifest.xml 文件到 Unity 工程的 Plugins/Android 目录内。
- 步骤 04** 打开 AndroidManifest.xml 文件,修改 Activity 的名称使其与 eclipse 工程中的 Activity 一致:

```

找到 <activity android:name=" com.unity3d.player.UnityPlayerProxyActivity"
将其替换为 <activity android:name=" com.project.helloworld.MainActivity"

```

- 步骤 05** 打开预设的脚本 AndroidAPP.cs,这个脚本里现在几乎什么也没有,添加代码如下:

```

// Android 的 Activity
private AndroidJavaObject activity;

```

```

void Start () {
    // 获得Android Activity
    AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
    activity = jc.GetStatic<AndroidJavaObject>("currentActivity");
}

void OnGUI()
{
    GUI.skin=m_skin;
    if (GUI.Button(m_showAndroidDialog, "显示 android 对话框"))
    {
        string[] args=new string[2];
        args[0]="Hello";
        args[1]="World";
        activity.Call("HelloWorld", args);
    }
}

```

代码非常简单, 在 Start 函数中我们获取了 Android 的 Activity, 在 OnGUI 的按钮中, 我们通过前面获得的 Activity 访问在 eclipse 中创建的 HelloWorld 函数, 并通过个 string 数组传递给 HelloWorld 函数两个参数。

步骤 06 确定当前 Unity 工程的 Bundle Identifier 与 eclipse 工程中设置的 Package Name 一致, 这里为 com.project.helloworld。

编译并在 Android 设备上运行程序, 效果如图 10-24 所示。



图 10-24 Android 真机测试效果

本节的示例工程保存在资源文件目录 chapter10_Android_E2U 内, 其中 HelloWorld 目录内是 eclipse 工程, UnityProject 目录内是 Unity 工程。

10.5.3 使用 LogCat 查看 Log

当程序运行在 Android 手机上时, 确定手机的 USB 线与电脑处于连接状态, 启动 eclipse 编

编辑器，在菜单栏选择【Window】→【Show View】→【Other】，然后选择 LogCat 打开 LogCat 窗口。

在 LogCat 窗口默认可以监视当前 Android 设备上所有输出的 log 内容，在 LogCat 窗口的过滤框中输入“tag:unity”则可以只显示 Unity 程序的 Log，如图 10-25 所示。

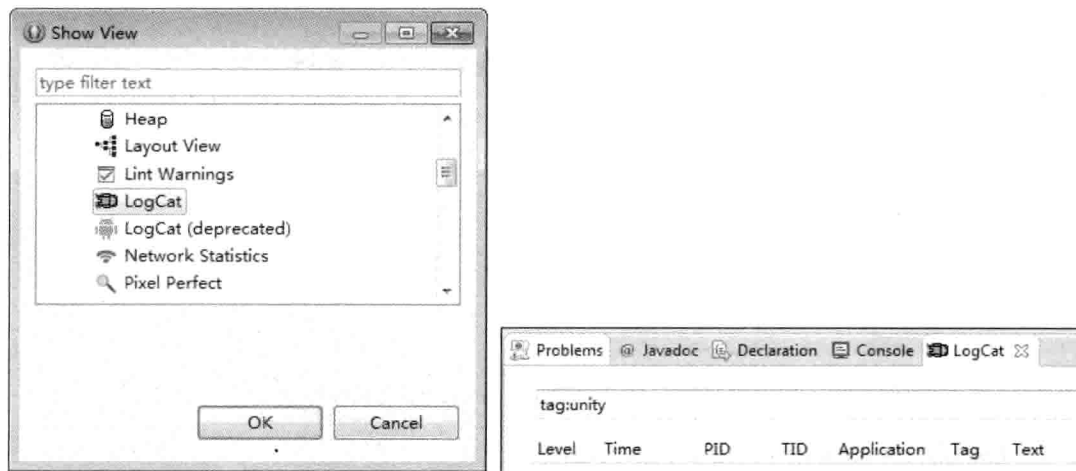


图 10-25 在 LogCat 窗口查看 Log

10.6 从 Unity 到 Eclipse

在下面的示例中，我们仍将在程序中显示一个 Android 对话框，但这次是将 Unity 工程导出到 eclipse 工程中。eclipse 与 Unity 工程之间的通信我们将使用 Unity 的 UnitySendMessage 函数完成，这个做法与在 iOS 平台是一样的。

10.6.1 导出 eclipse 工程

步骤 01 打开资源文件目录 chapter10_Android_Plugin_Start 内的 Unity 工程。

步骤 02 打开脚本 AndroidAPP.cs，添加代码如下：

```
// Android 的 Activity
private AndroidJavaObject activity;

// Use this for initialization
void Start () {

    // 当前游戏体的名字
    this.name = "AndroidManager";

    // 获得 Android Activity
    AndroidJavaClass jc = new AndroidJavaClass("com.unity3d.player.UnityPlayer");
```

```

        activity = jc.GetStatic<AndroidJavaObject>("currentActivity");
    }

    void OnGUI()
    {
        GUI.skin=m_skin;
        if (GUI.Button(m_showAndroidDialog, "显示 android 对话框"))
        {
            string[] args=new string[2];
            args[0]="Hello";
            args[1]="World";
            activity.Call("HelloWorld", args);
        }
    }

    // 改变摄像机背景颜色为红色,将在 eclipse 中使用 SendMessage 执行
    void AndroidCallBack()
    {
        Camera.main.backgroundColor = new Color(1.0f, 0, 0);
    }

```

这里的代码与前一节示例中的代码相似,增加了一个 `AndroidCallBack` 函数,它的作用是改变摄像机背景的颜色,我们将在 `eclipse` 中使用 `SendMessage` 执行这个函数。

在 `Start` 函数中,我们将当前游戏体的名字改为 `AndroidManager`,在 `eclipse` 工程中,我们将查找这个名字传送消息。

步骤 03 将当前工程的 `Bundle Identifier` 设为 `com.project.helloworld`,稍后在 `eclipse` 工程中设置的 `Package Name` 一定要与它相同。

步骤 04 在 `Unity` 菜单栏选择【File】→【Build Settings】,确定当前工程已转为 `Android` 平台,选中 `Google Android Project`,然后选择 `Export` 将当前工程导出为 `Android` 工程,如图 10-26 所示。

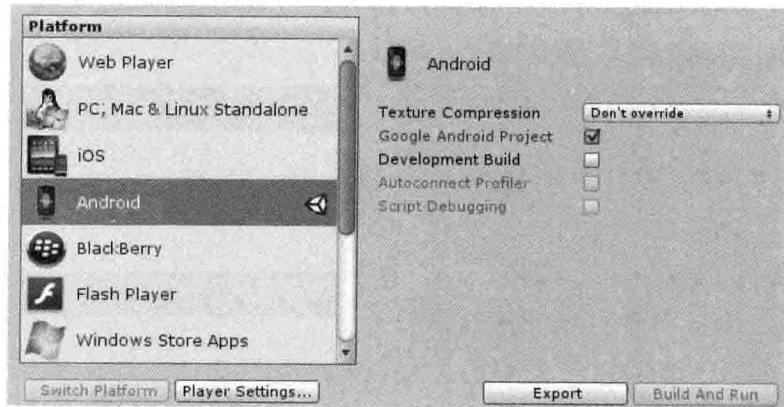


图 10-26 导出为 `Android` 工程

10.6.2 设置导出的 eclipse 工程

后面的工作将在 eclipse 中完成。实际上，从 Unity 中导出的 eclipse 工程可以直接在 eclipse 中编译使用，接下来我们要在导出的 eclipse 工程中添加少量代码。

步骤 01 在 eclipse 的菜单栏选择【File】→【Import】，然后选择【Existing Android Code Into Workspace】导入前面从 Unity 中导出的 eclipse 工程，如图 10-27 所示。

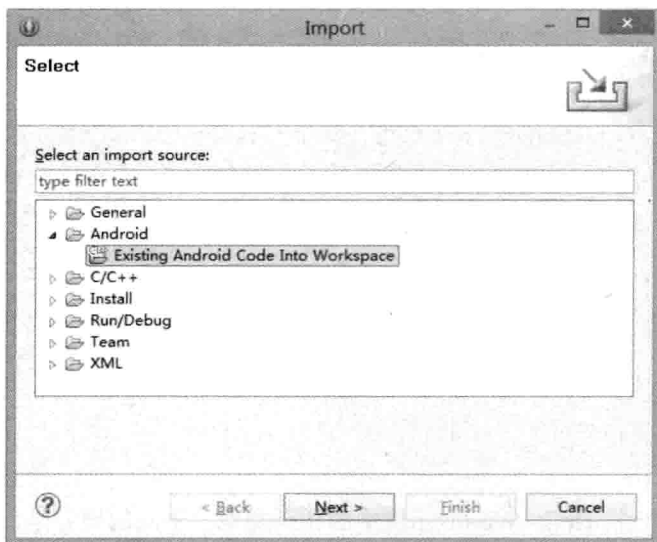


图 10-27 导入 Android 工程

步骤 02 打开 UnityPlayerNativeActivity.java 文件，如图 10-28 所示。注意，随着 Unity 版本的升级变化，这个文件名也可能会产生变化。



图 10-28 java 代码文件

步骤 03 在 UnityPlayerNativeActivity 类中添加两个函数，用于显示 Android 对话框。

```
// 在 Unity 中调用的函数
protected void HelloWorld(final String title, final String content)
{
    runOnUiThread(new Runnable() {
        public void run() {
            MakeDialog(title, content);
        }
    });
}
```



```

    }

    // 显示对话框
    public void MakeDialog(String title, String content)
    {
        AlertDialog.Builder builder =
            new AlertDialog.Builder(UnityPlayerNativeActivity.this);

        builder.setTitle(title)
            .setMessage(content)
            .setCancelable(false)
            .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int which) {
                    UnityPlayer.UnitySendMessage("AndroidManager",
                                                "AndroidCallBack", "");
                }
            })
        ;

        builder.show();
    }
}

```

这里的代码与前一个 Hello World 示例中的代码类似,不同的地方是,在单击对话框中的 OK 按钮后,使用 UnitySendMessage 向 Unity 的游戏体 AndroidManager 发送了一个消息,执行 AndroidCallBack 函数,使摄像机背景变为红色。

步骤 04 确定手机与计算机处于连接状态,在菜单栏选择【Run】→【Run】,选择 Android Application,如图 10-29 所示,在 Android 设备上测试程序。



图 10-29 测试 Android 程序

在 Android 设备上运行程序,效果与之前的示例类似,但在单击 OK 按钮后,摄像机背景会变为红色。

10.6.3 发布程序

将 Unity 工程导入到 eclipse 后, 只能在 eclipse 中发布 APK 程序。实际上, 在这里设置比在 Unity 中要简单一些, 步骤如下:

- 步骤 01** 选择要发布的工程, 单击右键选择【Android Tools】→【Export Signed Application Package】, 在导出窗口选择【Create new keystore】, 输入密码并将 keystore 文件保存到指定位置, 如图 10-30 所示。

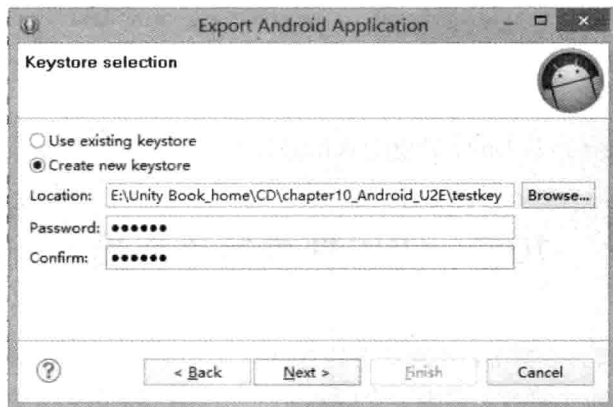


图 10-30 创建 keystore

- 步骤 02** 在这里填写 keystore 信息, 注意 Validity(years)通常设为 50 年以上, 如图 10-31 所示。

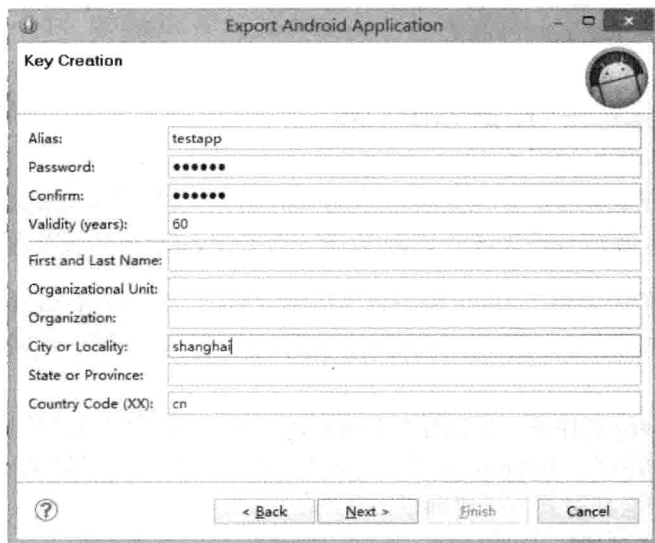


图 10-31 填写 keystore 信息

- 步骤 03** 选择路径导出用于发布的 APK 文件, 如图 10-32 所示。

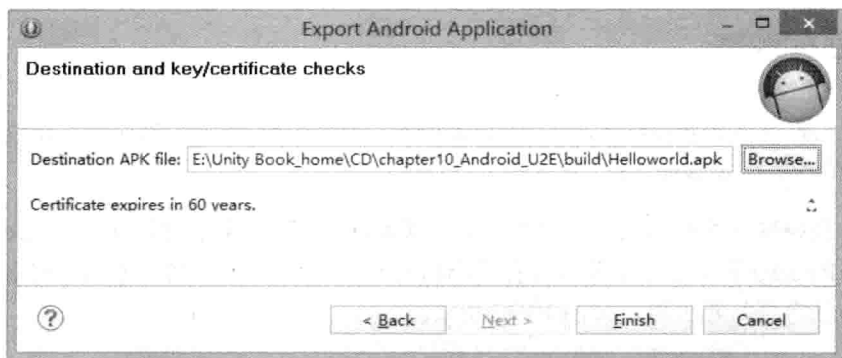


图 10-32 导出 APK 文件

本节的示例工程文件保存在资源文件目录 chapter10_Android_U2E 内, 其中 UnityProject 是 Unity 工程, export project 是从 Unity 导出的 eclipse 工程。

10.7 使用脚本编译游戏

Android 平台的运营商非常多, 不同平台的支付系统也都不一样, 当将游戏发布到这些平台上的时候, 几乎需要为每个平台专门做一个版本, 而每次游戏更新, 又需要重新为每个平台做一个版本, 这些重复、繁琐的工作几乎会占掉游戏发布后的大部分开发资源。

本节将会使用 Python 脚本结合 Unity 的命令行功能输出 Android 工程, 这样当需要创建很多不同版本的时候, 只需要简单修改代码, 即可完成所有原本需要手动操作的工作。本示例中的内容并不仅仅是针对 Android 平台, 当面向其他平台时, 我们同样也可以使用类似的方法将一些重复的设置工作通过脚本自动完成。

10.7.1 使用脚本输出 Android 工程

步骤 01 打开资源文件目录的 chapter10_Android_U2E 工程, 这是本章 10.6 节完成的示例工程文件, 我们将修改这个 Unity 工程, 使用 Python 脚本批处理输出 Android 工程。

默认情况下, 我们必须使用 Unity 编辑器中提供的 UI 选项才能将当前游戏工程输出为相应平台的可执行文件。接下来, 我们将不使用 UI 提供的功能, 直接通过 C#脚本控制整个输出流程。

步骤 02 创建 Editor 文件夹, 添加脚本 GameBuilder.cs, 它的作用是通过脚本直接将当前工程输出为一个 Android 工程, 通过类似的方式我们也可以通过脚本直接创建 IOS 或 PC 平台的游戏, 代码如下:

```
using UnityEngine;
using UnityEditor;
using System.Collections;

class GameBuilder{
```

```

// 输出路径
private const string BuildPath = "../export project/";

[MenuItem("GameBuilder/BuildForAndroid")]
public static void BuildForAndroid()
{
    // 如果不是 android 平台, 转为 android 平台
    if (EditorUserBuildSettings.activeBuildTarget != BuildTarget.Android)
    {
        EditorUserBuildSettings.SwitchActiveBuildTarget (BuildTarget.Android);
    }

    // 允许读写 SD 卡
    PlayerSettings.Android.forceSDCardPermission = true;

    // 设置 keystore 信息
    PlayerSettings.Android.keystoreName = "user.keystore";
    PlayerSettings.Android.keystorePass = "123456";
    PlayerSettings.Android.keyaliasName = "u2e demo";
    PlayerSettings.Android.keyaliasPass = "123456";

    // 允许调试 开发 外部修改
    BuildOptions options = BuildOptions.AllowDebugging | BuildOptions.Development
        | BuildOptions.AcceptExternalModificationsToPlayer;

    // 添加一个叫 Debug 的宏
    PlayerSettings.SetScriptingDefineSymbolsForGroup (BuildTargetGroup.Android,
        "Debug");

    // 设置标识符
    PlayerSettings.bundleIdentifier = "com.game.test";

    // 添加场景
    string[] scenes ={
        "Assets/demo.unity",
    };

    // 检查输出路径
    if (!System.IO.Directory.Exists (BuildPath))
    {
        System.IO.Directory.CreateDirectory (BuildPath);
    }
}

```

```
// 输出!
BuildPipeline.BuildPlayer(scenes, BuildPath, BuildTarget.Android, options);
}
}
```

这里的代码在 Unity 编辑器的菜单中增加了一个 BuildForAndroid 选项, 现在如果在菜单栏选择【GameBuilder】→【BuildForAndroid】, 即可直接将当前工程输出为 Android 工程到指定位置。

字符串 BuildPath 指向的路径即是工程的输出路径, 要记住这个位置, 后面需要到这里查找输出文件。

PlayerSettings 是一个静态类, 它的成员都是和输出游戏的设置有关。我们在 PlayerSettings.SetScriptingDefineSymbolsForGroup 中添加了一个名为“Debug”的预处理名称, 如果我们需要在代码中使用预处理, 即可在这里定义, 一个字符串中可以包括多个预处理的名称, 不同的名称需要用“;”分号隔开。注意, 在当前的实例中我们并没有使用到预处理, 这里只是为了说明如何设置它。

BuildOptions 中提供了很多输出游戏选项, 我们当前使用了几个选项, 有允许调试、采用开发模式、允许在外部修改。注意, 如果是希望直接从 Unity 输出 APK 文件, 就不要使用 BuildOptions.AcceptExternalModificationsToPlayer 选项。

BuildPipeline.BuildPlayer 会将当前的工程输出为相应平台的游戏, 它的第一个参数是一个 string 数组, 必须在 string 数组中包括所有需要输出的 Unity 场景文件名称, 它的第二个参数是输出的路径, 第三个参数是输出的平台, 最后一个参数对应的是输出选项。

步骤 03 退出 Unity, 我们将使用命令行在不需要启动 Unity 编辑器的情况下调用 GameBuilder.BuildForAndroid 函数。这里将使用 Python 作为脚本语言执行命令行, 为了执行和编写 Python 脚本, 需要到 Python 的官方网站 <https://www.python.org> 免费下载 Python 的安装包。在本书的资源文件目录 Software 中也提供了它的安装文件和相关文档。

步骤 04 启动 Python 编辑器, 编写 Python 代码如下:

```
import os, shutil

# 定义一个函数, 从 Unity 中输出 Android 工程文件
def BuildAndroid(unityPath, projectPath, logPath):
    # 更改当前路径位置
    os.chdir('F:/chapter10_Android_U2E_batch')
    # 如果当前工程已经存在, 删除这个工程
    if os.path.exists('./export project/chapter10_Android'):
        shutil.rmtree('./export project/chapter10_Android')

    # 将 Unity 可执行文件添加到环境变量路径
    os.putenv("path", unityPath);

    # unity 命令行
```



```

        command = 'Unity.exe -quit -batchmode -projectPath {0} -logFile {1} -executeMethod
GameBuilder.BuildForAndroid'.format(projectPath,logPath)

        # 执行命令行
        os.system(command)

        # Unity 路径
        unitypath='C:/Program Files (x86)/Unity/Editor/'
        # Unity 工程路径
        projectpath='F:/chapter10_Android_U2E_batch/UnityProject'
        # Log 存放位置
        logpath='F:/chapter10_Android_U2E_batch/UnityProject/buildlog.txt'

        # 执行 BuildAndroid 函数
        BuildAndroid(unitypath, projectpath, logpath)

```

def 是 Python 语言的关键字，这里使用它定义了一个名为 BuildAndroid 的函数，并带有 3 个参数，分别是 Unity 的路径、当前 Unity 工程的路径和输出 Log 文件的路径。

BuildAndroid 实际上只有一个功能，即执行一条命令行语句。语句中的 Unity.exe 是 Unity 的可执行文件，-quit 表示运行完语句后退出 Unity，-batchmode 表示进入批处理模式，-projectPath 指明当前的 Unity 工程路径，-logFile 指明输出的 Log 文件路径。

最后执行 BuildAndroid 函数，Unity 会将当前工程输出为一个 Android 工程到指定的位置。

步骤 05 在 Python 编辑器中按 F5 键执行脚本，略等片刻，Unity 会将当前工程输出到指定的位置。如果出现错误，可以打开 Log 文件查看问题所在。打开输出的 Android 工程，使用 Eclipse 的 import 导入这个工程。

步骤 06 在 Eclipse 中打开 UnityPlayerNativeActivity.java 文件，将本章 10.6.3 节步骤 6 创建的 HelloWorld 和 MakeDialog 函数复制粘贴到 UnityPlayerNativeActivity 类中。确定手机已经与 PC 机连接，在 Eclipse 的菜单栏选择【Run】→【Run As】→【Android Application】，将游戏输出到手机上运行。

因为每次从 Unity 中输出的 Android 工程都是一个新的工程，这样我们会丢失掉之前完成的很多设置和工作。我们可以将修改过的文件备份到硬盘中的其他位置，每次从 Unity 重新输出 Android 工程后，使用脚本将备份的文件拷贝回输出的工程中。

步骤 07 将 Eclipse 工程中的\src\com\game\test\UnityPlayerNativeActivity.java 文件复制到硬盘中的其他地方，在前面创建的 Python 脚本 os.system(command)语句后添加如下语句，它的作用是将备份的 UnityPlayerNativeActivity.java 文件复制回输出的 Android 工程中。

```

#备份文件位置
source = 'F:/chapter10_Android_U2E_batch/UnityPlayerNativeActivity.java'
#目标位置
target = 'F:/chapter10_Android_U2E_batch/export project/chapter10_Android/'

```

```
src/com/game/test/UnityPlayerNativeActivity.java'
#执行复制
shutil.copy(source,target)
```

现在, 无论何时需要更新版本, 只需要运行一次 Python 脚本, 即可将 Unity 工程输出到指定位置, 并应用到之前对 Android 工程所做的修改和设置。

10.7.2 使用脚本编译 Android 工程

接下来, 我们将对 Python 脚本再做修改, 添加少量代码, 直接在 Python 脚本中完成 Android 工程的编译工作, 输出 APK 文件。

步骤 01 到网址 <http://ant.apache.org/bindownload.cgi> 免费下载 ant, 它专门用来编译 java 程序。也可以直接使用本书资源文件目录内提供的文件 (software/apache-ant-1.9.4-bin.zip)。

步骤 02 将 Android SDK 安装路径/tools 目录添加到环境变量 path 中, 将 Ant 安装路径/bin 目录也添加到环境变量 path 中, 如图 10-33 所示。

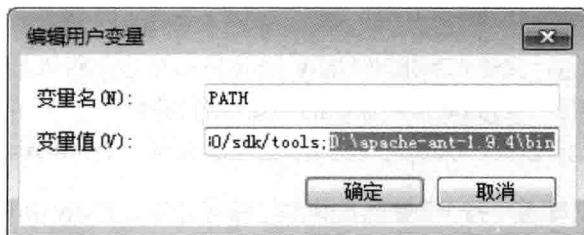


图 10-33 添加路径环境变量

步骤 03 在 Windows【开始】菜单中输入 cmd 打开控制台窗口, 输入 android -h 查看 android 批处理命令帮助, 输入 ant -h 查看 ant 的批处理命令帮助。

步骤 04 继续修改 Python 脚本, 在 BuildAndroid 函数的最后添加代码如下:

```
# 更新当前路径到输出的 android 工程路径
os.chdir('../export project/chapter10_Android')
# 检查是否已经存在 build.xml 文件
if os.path.exists('../build.xml'):
    os.remove('../build.xml')
# 添加 android 和 ant 批处理环境变量

os.putenv("path", 'D:/adt-bundle-windows-x86_64-20131030/sdk/tools;D:/apache-ant-1.9.4/bin')

# 更新 android 工程
os.system('android update project --target 5 --path ./ --name mygame')

# 添加数字签名
with open('local.properties', 'a') as f:
```

```
f.write('key.store=../../testkey\n')
f.write('key.alias=testapp\n')
f.write('key.store.password=123456\n')
f.write('key.alias.password=123456\n')

os.system('ant clean')
# 创建 apk
os.system('ant release')
```

使用 ant 编译 Java 程序, 需要创建一个 build.xml 文件, 它记录了用于编译的各种配置信息, 执行 android update 语句会帮助我们自动创建这个文件: -target 是指 android SDK 的目标 SDK ID, 在控制台窗口输入 android list targets 可以对该 ID 进行查阅; --path 是指当前工程的路径; --name 为工程的名称。

执行完 android update 语句后默认会在工程中创建一个 local.properties 文件, 我们使用 IO 操作打开这个文件, 并在这个文件最后添加数字签名信息: key.store 为签名文件路径; key.alias 为签名名称; key.store.password 为签名密码; key.alias.password 为别名密码。

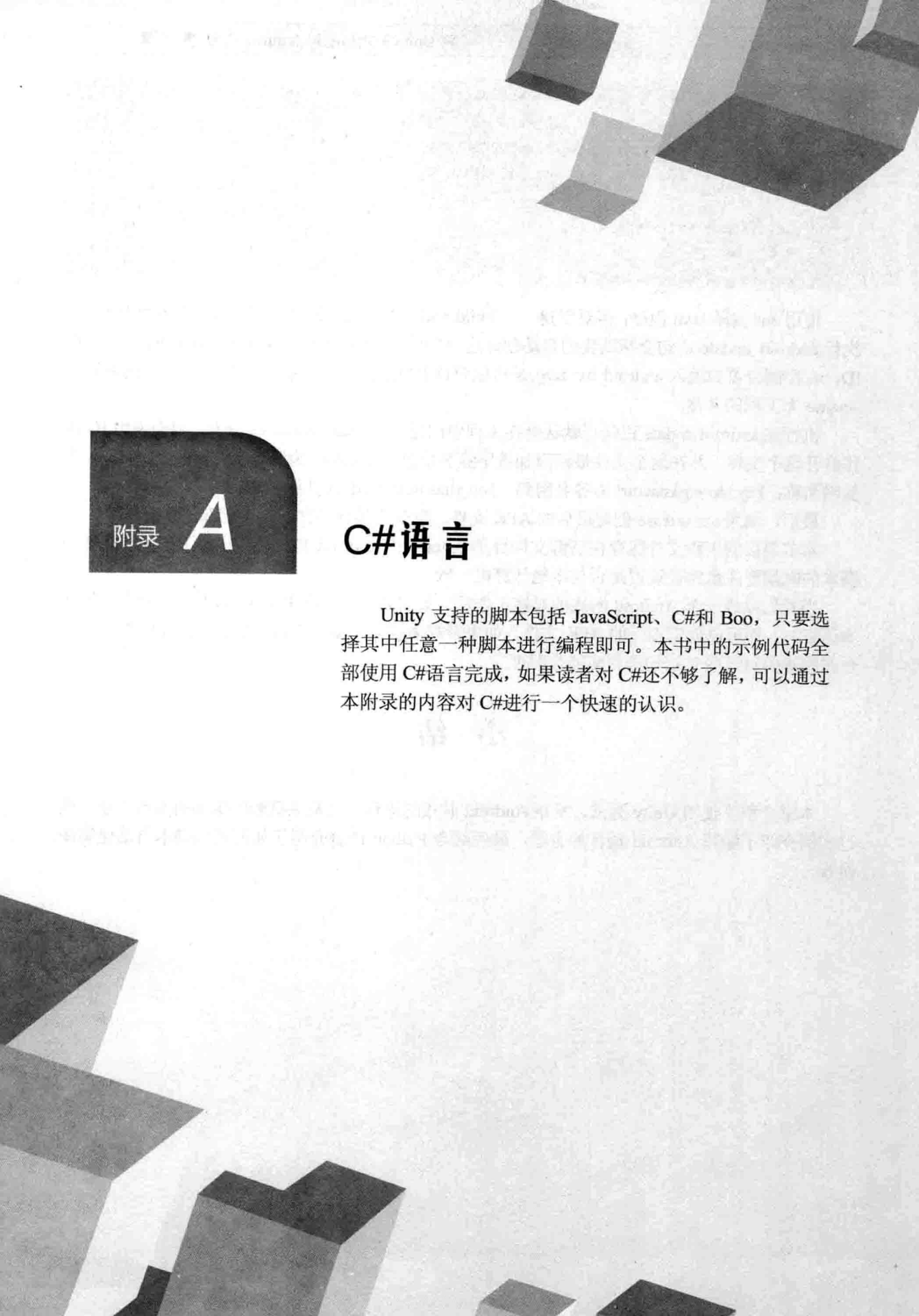
最后, 调用 ant release 创建最后的 APK 文件, 默认它被保存在当前工程 bin 目录下。

本节最后的工程文件保存在资源文件目录 chapter10_Android_U2E_batch 内, 在使用 Python 脚本的时候要注意路径设置是否与本地计算机一致。

当我们完成一个 Android 游戏的对接工作后, 只需要运行一次 Python 脚本 (或使用其他脚本语言), 即可以创建相应的 APK 文件。如果游戏需要面向多个平台, 那么只需要针对不同平台在脚本中写一些 if 语句作相应处理即可。

小 结

本章介绍了使用 Unity 测试、发布 Android 游戏的流程, 以及手机触屏操作的实现方法, 通过实例介绍了编写 Android 插件的方法, 最后结合 Python 语言介绍了如何使用脚本自动化编译游戏。



附录

A

C#语言

Unity 支持的脚本包括 JavaScript、C#和 Boo，只要选择其中任意一种脚本进行编程即可。本书中的示例代码全部使用 C#语言完成，如果读者对 C#还不够了解，可以通过本附录的内容对 C#进行一个快速的认识。

A.1 C#基础

A.1.1 C#简介

在 Unity 内编程, 首选使用 C#脚本, 它是一门面向对象语言, 所有的事物都可以看作是对象。在 C#中, 万物皆是类, 绝不允许有一个独立于类的函数或变量。同时, 它有着和 C/C++ 类似的语法, 和 Java 类似的垃圾回收机制, 简单、易用。

Unity 使用的 C#和微软.NET 平台下的 C#很像但又不完全一样。Unity 内的 C#运行于 Mono 虚拟机, 它是一个开源软件平台, 以微软的.NET 开发框架为基础(.NET 是微软为 Windows 操作系统提供的应用程序编程接口), 能够实现跨平台开发。或者可以这么理解, 因为 Mono 提供了与.NET 差不多的功能, 在 Unity 内使用 C#不但能调用 Unity 引擎本身的功能, 还能调用.NET 平台中提供的大部分功能。

A.1.2 运行控制台程序

接下来, 我们将集中精力在 C#语言本身。为了能够快速调试 C#代码, 我们先将 Unity 放到一边, 使用 Visual Studio 直接创建控制台程序调试代码, 创建控制台程序的步骤如下:

步骤 01 启动 Visual Studio, 在菜单栏选择【File】→【New】→【Project】, 打开 New Project 窗口。选择 Console Application 创建一个控制台程序, 如图 A-1 所示。

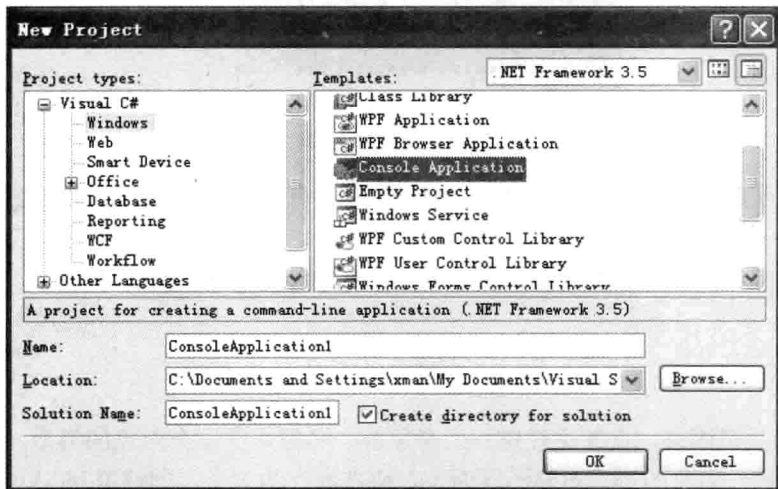


图 A-1 创建控制台程序

步骤 02 创建一个控制台程序后默认会自动创建一个 Program.cs 文件, 里面已经包括了一些基本的代码。在控制台程序中, Main 函数是程序的入口, 我们将使用 Console.WriteLine 在控制台窗口输出“Hello,world”几个字。完整的代码如下:


```

using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // 输出 Hello, world
            Console.WriteLine("Hello, world");

            // 输入任意键退出
            Console.ReadKey();
        }
    }
}

```

步骤 03 在菜单栏选择【Debug】→【Start Debugging】（或按 F5 键）运行程序，如图 A-2 所示。

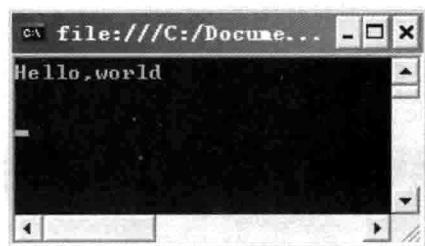


图 A-2 运行控制台程序

可以看到，创建控制台程序非常简单，本附录中所有的 C# 代码，都可以放到一个控制台程序中进行调试。

本节的示例文件保存在资源文件目录\appendixA_C#\helloworld 中。

A.1.3 类型

C# 是一种强类型语言，在使用任意一个对象前，必须声明这个对象的类型，如整型、浮点型、字符串类型等。对象的类型对编译器而言是所占内存的大小，如整型 `int` 占 4 个字节等。

C# 的类型分为两大类：值类型和引用类型。两者的主要区别是值在内存中的存储方式不同。值类型的实例通常是在栈上静态分配的，引用类型的对象则总是在进程堆中动态分配。

值类型包括内置类型（用关键字 `int`、`char`、`float`、`bool` 等声明），结构（用关键字 `struct` 声明）和枚举（用关键字 `enum` 声明）。

引用类型包括类（用关键字 `class` 声明）和委托（用关键字 `delegate` 声明）。

A.1.4 内置类型

所有的值类型隐性派生于 `System.ValueType`，其中内置类型是最基本的类型。下面的表 A-1 详细说明了这些类型的大小和取值范围。

表 A-1 内置类型

类型	大小/字节	.NET 类型	说明
byte	1	Byte	无符号，值 0~255
char	2	Char	Unicode 字符
bool	1	Boolean	true 或者 false
sbyte	1	SByte	有符号，值 -128~127
short	2	Int16	有符号，值 -32 768~32 767
ushort	2	UInt16	无符号，值 0~65535
int	4	Int32	有符号整数，值 -2 147 483 648~2 147 483 647
uint	4	UInt32	无符号整数，值 0~4 294 967 295
float	4	Single	浮点数，7 位有效数字
double	8	Double	双精度浮点数，15~16 位有效数字
decimal	12	Decimal	固定精度，值为最大 28 位加小数点
long	8	Int64	有符号整数，值 -9 223 372 036 854 775 808~9 223 372 036 854 775 807
ulong	8	UInt64	无符号整数，值 0 ~ 0xffffffffffffff

通常，选用什么样的整数类型取决于值的大小需求，现在的计算机内存都比较大，大多数情况可以选择 `int` 类型。对于浮点型，C# 默认的浮点型为 `double` 类型，但大部分情况用 `float` 就可以了，`float` 类型的赋值，后面要跟一个小写字母 `f`，如下所示：

```
float somevar = 0.1f;
```

内置类型，可以隐式或显式地转换为另一种类型。隐式转换是自动进行的，如从 `short` (2 字节) 转为 `int` (4 字节)，这个转换不会丢失任何信息。而反方向转换，需要显式地转换，因为 `int` 的值可能会高于 `short` 的最大值，这时信息则有可能会丢失，如下所示：

```
short x=10;
int y = x; // 隐式转换
x = y; // 错误,不能编译
x =(short) y; // 显式转换
```

A.1.5 标识符

标识符可以用来表示类型、方法、变量、常量、对象的名称。标识符必须以字母或下划线开头，并区分大小写，如下所示：

```
int abc;    // int 表示整数类型, abc 是标识符
float ABc;
```

```
bool _abc;
float 2Abc; //错误!不能编译
```

A.1.6 语句和表达式

一条完整的程序指令称为语句，一个完整的程序就是由若干条语句组成，每条语句必须以分号为结尾。

能够计算出值的语句称为表达式，这是一种最基本的语句，通常使用等于号“=”进行赋值。对于普通的语句，程序会按顺序执行它们，如下所示：

```
int first;           // 语句1, 声明一个整数类型
first = 100;         // 语句2, first 的值为 100
int second = first;  // 语句3, 声明一个整数类型 second, 值为 100
```

A.1.7 变量和常量

创建一个变量就是声明一个类型，可以在任何时候改变其赋值。注意，在使用一个变量之前，一定要为其赋值，否则将出错，如下所示：

```
int a;               //声明 a
int b = a;           //错语, 不能编译, 因为 a 在使用前没有被赋值
float c = 0.0f;       // c 的值为 0
c = 50.1f;           // c 的值为 50.1
```

常量是一种值固定的变量。在实际编程中，有些值是不需要变动的，如 π ，它的值永远是 3.1415926，为了防止不小心改变它的值，可将其设为常量。

声明一个常量只需要在类型前面增加关键字 **const**，并在声明时为其赋值，之后再也不能改变它的值，如下所示：

```
const float PI = 3.141f;
PI = 500; //不能编译
```

A.1.8 枚举

枚举是一种独特的值类型，可以把它看作是一个常量列表。在下面的代码中，有 3 种不同的水果，每种水果有一个常量值：

```
enum FRUIT
{
    Apple=0,
    Banana, //值为 1
    Cherry, //值为 2
}
```

```
FRUIT fruit = FRUIT.Apple;// 当前的水果类型为苹果
```

也可以使用常量代替枚举完成前面的工作，但代码之间会缺少联系，如下所示：

```
const int Apple = 0;
const int Banana = 1;
const int Cherry = 2;

int fruit = Apple;
```

A.1.9 数学操作符

C#中常用的数学操作符有 5 个，分别为+（加）、-（减）、*（乘）、/（除）和%（模），其中加、减、乘、除与数学中的用法一样，模操作符用于获取余数，具体用法如下所示：

```
int a = 2, b = 3, c=0;
float d=0;

c = a + b;           // c 的值为 5
c++;                // c 的值为 6
c = c - a;           // c 的值为 4
c = a * b;           // c 的值为 6
c--;                // c 的值为 5
c*=2;               // c 的值为 10

d = b / a;           //d 的值为 1, 注意, 因为 a 和 b 为整数, 所以返回值也是整数, 去掉了小数点后面的值
d = (float)b / (float)a; // d 的值为 1.5

d = a % b;           // d 的值为 2, 当被除数大于除数, 返回除数的值
d = b % a;           // d 的值为 1, 3/2 的余数为 1
```

任何数与 10 进行模计算得到的余数，即是这个数字的最后一位数，用这个方法，可以通过模计算获得一个数字的任意一位数，如下所示：

```
int number = 4321;
int v=0;
v = number % 10;           // 返回 1
v = number / 10 % 10;      // 返回 2
v = number / 100 % 10;     // 返回 3
v = number / 1000 % 10;    // 返回 4
```

A.1.10 关系操作符

关系操作符包括==、!=、>、>=、<、<=，作用是比较两个值的大小关系，经常和条件语句 if 一同使用，然后返回布尔值（true 或 false），如下所示：

```

int a = 50, b = 100;
if ( a == b )    // 判断 a 和 b 是否相等,此例返回 false
if ( a != b )    // 判断 a 和 b 是否不相等, 此例返回 true
if ( a > b )      // 判断 a 是否大于 b, 此例返回 false
if ( a >= b )     // 判断 a 是否大于或等于 b, 此例返回 false
if ( a < b )      // 判断 a 是否小于 b, 此例返回 true

```

A.1.11 逻辑操作符

逻辑操作符包括&&(与)、||(或)和!(非), 主要用于拥有多个表达式的复合条件语句, 只有返回值为 true 时才会执行条件语句后面{}中的代码, 具体用法如下所示:

```

int a = 50, b = 100;
if (a == 50 && b == 50) { }    //值为 false, 只有两个表达式的值都为真才能返回 true 值
if (a == 50 || b == 50) { }    //值为 true, 只要其中任意一个表达式的值为真即返回 true 值
if (!(b == 50)) { }            //值为 false, 如果表达式的值为 true, 加上!后返回值即为 false

```

A.1.12 操作符优先级

计算多个操作符的值时, 顺序是从左到右, 先乘除后加减, 但可以使用括号改变顺序, 如下所示:

```

int a = 5, b = 1, c = 10;
int d = (a + b) * 10; // d 的值为 60
int e = a + b * 10;   // e 的值为 15

```

A.1.13 方法

对于普通的语句, C#是按顺序执行, 从开始执行到最后, 但大部分情况, 在程序中都需要执行分支语句和循环语句。

分支语句分为无条件分支和有条件分支。无条件分支通过调用方法(也称作函数)来实现, 一个方法内包括若干条语句, 当程序遇到方法时, 会先处理方法中的语句, 然后再按顺序调用其他语句, 如下所示:

```

// 定义一个方法, 返回值类型为浮点型, 标识符是 Add, 带两个浮点型参数
float Add(float a, float b)
{
    return a + b;
}

//...
float number1 = 5;           //执行语句 1
float number2 = 10;          //执行语句 2

```



```
// 语句 3,先执行方法 Add 内的语句,再将返回值赋给 number3
float number3 = Add(number1, number2);

Console.WriteLine(number3); // 输出 number3,值为 15
```

如果方法的参数是内置类型,则在方法内部引用的是参数的副本,方法内的运算不会影响到参数本身。如果希望在方法内部改变内置类型参数的值,则需要添加 `ref` 或 `out` 标识符,如下所示:

```
// 定义一个方法,参数前面添加 ref 标识符,表示引用类型
void Change(ref int a)
{
    a += 10;
}

int a = 0;
Change(ref a);
Console.Write( a ); // 因为是引用类型, a 的值现在是 10
```

我们也可以将 `ref` 替换为 `out`,则我们必须在方法内部首先初始化引用的参数使用它。

A.1.14 条件分支语句

条件分支语句通过条件语句创建,会使用到 `if`、`else`、`switch` 等关键字,其中 `if` 语句是最常用的条件语句,当表达式的值为 `true` 时会执行 `if` 后面 `{ }` 内的语句,如下所示:

```
int a = 1, b = 2, c = 0;

// 如果 a 大于 b, c 的值等于 a, 否则等于 b
if (a > b)
    c = a;
else
    c = b;
```

`switch` 语句的作用和 `if` 类似,但可读性更佳,下面是一个完整示例:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        // 定义枚举
```

```

enum FRUIT
{
    Apple = 0,
    Banana,
    Cherry,
}

static FRUIT fruit = FRUIT.Apple;

static void Main(string[] args)
{
    switch (fruit)
    {
        case FRUIT.Apple: // 如果 fruit 的值等于 FRUIT.Apple
            Console.WriteLine("apple");
            break; // 使用 break 退出

        case FRUIT.Banana:
            Console.WriteLine("Banana");
            break;

        case FRUIT.Cherry:
            Console.WriteLine("Cherry");
            break;
    }

    // 以上 switch 语句等同于
    if (fruit == FRUIT.Apple)
        Console.WriteLine("apple");
    else if (fruit == FRUIT.Banana)
        Console.WriteLine("Banana");
    else if (fruit == FRUIT.Cherry)
        Console.WriteLine("Cherry");

    Console.ReadKey(); // 按任意键退出
}
}

```

本节的示例文件保存在资源文件目录\appendixA_C#\condition\中。

A.1.15 循环语句

while 语句是一个条件循环语句，每次循环都会返回一个逻辑表达式的值，只有表达式的值为 **false** 时才会退出 **while** 循环，所以在实际使用中要避免程序陷入死循环，如下所示：

```
int n = 0;
while (n < 10) //如果n的值小于10,则会一直循环{}中的语句
{
    n++; //当n的值为10时退出 while 循环
}
```

for 语句与 while 类似,都是条件循环语句,不同的是,for 语句可以更好的控制循环次数,如下所示:

```
for ( int i = 0; i < 10; i++) //i 的初始值为 0,当 i 小于 10 时会继续循环,每次循环 i 自加 1
{
    //循环 10 次
}
```

很多时候,需要在循环中直接退出或不再执行剩下的语句继续下一个循环,这时可以使用 break 或 continue 改变循环状态,如下所示:

```
int n = 10;
while (true)
{
    if (n == 5) // 当n等于5时使用 break 直接退出循环
        break;
    n--;
}

for ( int i = 0; i < 10; i++)
{
    if (i == 5) // 当i等于5时,将跳过后面的代码直接到下一个循环
        continue;
    //其他代码...
}
```

A.1.16 三元操作符

三元操作符的作用和 if 语句类似,如果条件表达式返回值为 true,则执行表达式 1,否则执行表达式 2,如下所示:

```
//条件表达式 ? 表达式 1 : 表达式 2

int a = 50, b = 100;

int max = a > b ? a : b;    //max 的值为 100
```

A.1.17 预处理

使用预处理,可以使程序编译或不编译某些代码,与 C++的预处理相比,C#的预处理不支持宏。

使用预处理首先要使用`#define` 定义一个预处理标识符,它必须被定义到 `using` 语句之前,然后可以在任何地方使用`#if`、`#else`、`#endif` 等关键字定义预处理影响的代码,如下所示:

```
#define DEBUG // 定义预处理标识符 DEBUG,必须声明在 using 关键字之前

using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {

#if TEST
            // 因为未定义预处理标识符 TEST,这里的代码不会被编译
            // do something...
#elif DEBUG
            // 定义了 DEBUG,这里的代码将会被编译
            Console.WriteLine("DEBUG");
#else
            //这里的代码不会出现在最终程序中,不会被编译
            // do something...
#endif

#if TEST || DEBUG
            // 只要定义了预处理标识符 TEST 或 DEBUG 中的任意一个,即可编译这里
            Console.WriteLine("TEST || DEBUG");
#endif

#if !TEST
            // 如果没有定义预处理标识符 TEST,编译这里
            Console.WriteLine("!TEST");
#endif

            // 输入任意键退出
            Console.ReadKey();
        }
    }
}
```



```

    }
}

```

本节的示例文件保存在资源文件目录\appendixA_C#\preprocessor\中。

A.2 面向对象编程

A.2.1 类

类是引用类型，代表不同的数据对象，它可以是具体或抽象的，如游戏中的主角、敌人等。定义一个类需要使用关键字 `class` 声明，并给它一个名字，在一对大括号 `{}` 中定义类的属性和方法等，如下所示：

```

//定义类
public class SomeClass
{
    int data1=10; // 定义属性
    int data2=5;

    // 定义方法
    public int Add()
    {
        return data1+ data2;
    }
}

```

C#使用 `new` 创建类的对象，然后这个对象可以调用公有的成员属性或方法，如下所示：

```

SomeClass sc = new SomeClass(); //创建一个对象
int n=sc.Add();                // 调用 Add 方法

```

在使用 `new` 创建对象的同时，默认会调用类的构造函数，定义构造函数需要声明一个与类同名的方法，并且没有返回类型，如下所示：

```

public class SomeClass
{
    //定义构造函数
    public SomeClass()
    {
        Console.WriteLine("SomeClass");
    }
}

```



```
//...
//创建一个对象,输出"SomeClass"
SomeClass sc = new SomeClass();
```

C#提供了垃圾回收器，在程序执行到对象的作用域以外时，对象会自动被系统垃圾回收，因此不需要显示地通过 delete 销毁。



提示

在 Unity 中，继承自 MonoBehaviour 的类不能使用 new 创建，同时也不能使用构造函数。

A.2.2 this 关键字

在每个类的方法中都可以使用 this 关键字指向对象的属性和其他方法，默认它可以被省略，但使用它可以避免一些名字歧义。在下面的代码示例中，如果一个方法中的参数名与类的成员名一样，这时可以使用 this 关键字来区分：

```
public class SomeClass
{
    int data = 0;

    public void SomeMethod(int data)
    {
        this.data = data;
    }
}
```

A.2.3 封装

面向对象编程的核心是封装、继承和多态。

C#使用访问修饰符 public、protected、internal、protected internal、private 决定类或类成员的可见性，尽可能封装内部实现。5 种修饰符意义如表 A-2 所示。

表 A-2 访问修饰符

访问修饰符	意义
public	完全公开，可以在类内部或外部任何地方访问
protected	同一名字空间内的派生类可以访问
internal	同一名字空间内的任何地方可以访问
protected internal	满足 protected 或 internal 的条件可以访问
private	仅在类内部可以访问

**提示**

在当前版本的 Unity 中，C#并不支持名字空间（namespaces）。

如在定义类的时候，在它的成员属性、方法前面加上修饰符 **public**，即表示它的成员属性、方法是公有的，完全可见的，否则只能在类内部访问，如果不加任何修饰符，则默认为 **private**，如下所示：

```
//在 class 前加上 public,表示这是一个公有类
public class SomeClass
{
    public int data1=0;    // 可以在任何地方访问
    int data2 = 0;        // 没有修饰符,默认为 private,只能在类内部访问

    // 公有方法,可以在任何地方访问
    public int Add()
    {
        return data1 + data2;
    }

    // 私有方法,只能在类内部访问
    private void CallIt()
    {
        //...
    }
}

//...
SomeClass sc = new SomeClass(); //创建一个对象
sc.data1 = 10; //OK
sc.Add();      // OK
sc.data2 = 10; //错误,私有属性,不能在类外部调用
sc.CallIt() ;  //错误,私有方法,不能在类外部调用
```

在 C#中，可以通过 **set** 和 **get** 方式将属性设为“可写”或“只读”。**set** 方式有一个隐藏参数 **value**，它即是指向属性的参数。只使用 **get** 方式的属性，它将是一个只读属性，只能被访问，不能改变它的值，如下所示：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    public class Player
```

```

    {
        private string m_name = ""; //私有, 不能直接访问
        public string Name
        {
            set { m_name = value; } //通过访问 Name 属性改变 m_name 的值
            get { return m_name; } //通过访问 Name 属性获得 m_name 的值
        }

        private int m_life = 100; //私有, 不能直接访问
        public int Life
        {
            get { return m_life; } //通过访问 Life 属性获得 m_life 的值
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Player player = new Player();
            player.Name = "player1"; //OK
            //player.Life = 10;      //错误, Life 是只读属性

            Console.WriteLine(player.Name); // 输出 Name
            Console.WriteLine(player.Life); // 输出 Life 的值 100

            // 输入任意键退出
            Console.ReadKey();
        }
    }
}

```

本节的示例文件保存在资源文件目录\appendixA_C#\property\中。

A.2.4 继承与多态

使用继承功能, 可以创建一个派生类继承基类的代码, 并只需重写需要改动的地方, 这样做除了利于代码的重用, 从逻辑上看起来也更合理。实际上, 所有 C#类都是从 System.Object 派生出来的。

创建派生类只需要在类的名字后面加一个“:”号, 后面再跟基类的名称, 如下所示:

```

// 定义一个叫 Enemy 的基类
public class Enemy{

```

```

//构造函数
public Enemy(){
    Console.WriteLine("enemy contructor");
}

public void UpdateAI(){
    Console.WriteLine("update enemy ai");
}
}

// 派生类 Boss 继承基类 Enemy,
public class Boss:Enemy{

    //构造函数
    public Boss(){
        Console.WriteLine("boss contructor");
    }
}

//...
Boss boss = new Boss(); // 先执行 Enemy 的构造函数,然后再执行 Boss 的构造函数
boss.UpdateAI();        // 调用 Enemy 的方法 UpdateAI

```

只是通过继承使两个类变得一样没有太大意义,很多时候,需要在派生类中重写基类的某些方法,这么做需要在基类中将方法标记为 **virtual**,然后在派生类的方法定义中使用关键字 **override** 代替基类的方法,完整的示例如下:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    // 定义一个叫 Enemy 的基类
    public class Enemy
    {
        //构造函数
        public Enemy()
        {
            Console.WriteLine("enemy contructor");
        }

        // 声明为虚方法
        public virtual void UpdateAI()

```



```

    {
        // Enemy 的 AI
        Console.WriteLine("update enemy ai");
    }
}

// 派生类 Boss 继承自基类 Enemy,
public class Boss : Enemy
{
    //构造函数
    public Boss()
    {
        Console.WriteLine("boss contructor");
    }

    //添加 override, 代替基类的方法
    public override void UpdateAI()
    {
        // Boss 的 AI
        Console.WriteLine("update boss ai");
    }
}

class Program
{
    static void Main(string[] args)
    {
        Enemy[] enemies = new Enemy[2]; //创建一个数组, 包括两个 Enemy 基类
        enemies[0] = new Enemy();        //创建一个 Enemy, 执行 Enemy 的构造函数
        enemies[1] = new Boss(); //创建一个 Boss, 先执行 Enemy 的构造函数, 再执行 Boss 的

        for (int i = 0; i < 2; i++)
        {
            // enemies[0]会调用 Enemy 类的 UpdateAI
            // enemies[1]会调用 Boss 类的 UpdateAI
            enemies[i].UpdateAI();
        }

        // 输入任意键退出
        Console.ReadKey();
    }
}
}

```


编译程序，输出结果如图 A-3 所示。

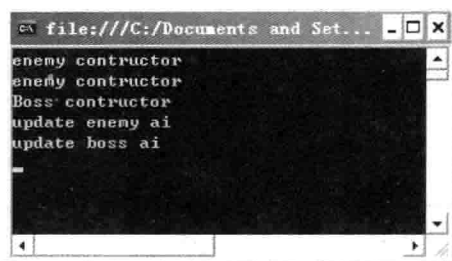


图 A-3 程序输出结果

本节的示例文件保存在资源文件目录\appendixA_C#\inherit\中。

A.2.5 静态成员

在定义类的成员属性或方法时加上 `static`，即表示它是一个静态成员，静态成员不能被类的对象引用，它的值会被所有对象共享，如下所示：

```
using System;
using System.Collections.Generic;
using System.Text;
namespace ConsoleApplication1
{
    public class Player
    {
        public static int count = 0; //静态成员,用于统计 Player 对象的数量

        public Player()
        {
            count++; // 每创建一个 Player 对象,count 自加一次
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Player player1 = new Player(); //创建一个 Player 对象
            Console.WriteLine(Player.count); // 输出 1,有 1 个 Player

            Player player2 = new Player(); // 又创建一个 Player 对象
            Console.WriteLine(Player.count); // 输出 2,有 2 个 Player

            // n = player2.count; 错误用法,静态成员不能被对象直接调用
        }
    }
}
```

```
// 输入任意键退出
Console.ReadKey();

}

}
```

**提示**

不能在静态方法中调用非静态的属性或方法。

本节的示例文件保存在资源文件目录\appendixA_C#\staticmember\中。

A.2.6 通过字段名称序列化对象

C#提供的泛型类似 C++中的模板，下面是一个实例，结合泛型和反射功能，可以使我们通过字符串查找类中的变量成员，并赋值。这个功能非常有用，比如当我们从数据库或 XML 文件中读取数据的时候，即可以通过字段名称赋值，实现自动序列化。代码如下：

```
using System;
using System.Collections.Generic;

namespace genericsample
{
    class Program
    {
        static void Main(string[] args)
        {
            // 创建一个 Player 对象
            Player p = new Player();

            // 通过字段名称给 Player 对象赋值
            SetObjectValue<Player>(p, "name", "goodman");
            SetObjectValue<Player>(p, "id", 100);

            Console.WriteLine(string.Format("name:{0},id:{1}", p.name, p.id));
            Console.ReadLine();
        }

        // 一个 Player 类，它包括名称为 name 和 id 两个成员
        public class Player
        {
            public string name = string.Empty;
            public int id = 0 ;
        }
    }
}
```

```
// 一个使用了泛型的函数, <T>是泛型参数, 它可以是任何类型的对象
public static void SetObjectValue<T>(T obj, string fieldname, object value)
{
    // 通过字段名称获得对象的成员变量
    System.Reflection.FieldInfo info = obj.GetType().GetField(fieldname);

    if (info != null)
    {
        // 给成员变量赋值
        info.SetValue(obj, value);
    }
}
}
```

本节的示例文件保存在资源文件目录\appendixA_C#\genericsample\中。

A.3 字符串

C#字符串是使用 **string** 关键字声明的一个字符数组, 它也是一个对象, 封装了所有字符串操作, 如比较、插入、删除、查找等。下面是一个完整的程序示例, 示范了常用的 **string** 用法:

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // 创建一个字符串
            string str1 = "apple orange banana";
            Console.WriteLine("str1:"+str1);

            // 创建另一个字符串
            string str2 = str1+" peach";
            Console.WriteLine("str2:" + str2);

            // 比较两个字符串是否一致
```

```

if (String.Compare(str1, str2) == 0)
{
    Console.WriteLine("str1 和 str2 两个字符串一致");
}
else
{
    Console.WriteLine("str1 和 str2 两个字符串不一致");
}

// 从第 0 个字节开始查找空格的位置
int n = str1.IndexOf(' ', 0);
Console.WriteLine("str1 第一个空格在第{0}个字节", n);

// 删除第 1 个空格之后的所有字符
str2=str1.Remove(n);
Console.WriteLine("删除 str1 第一个空格后的所有字符:"+str2);

// 将所有空格替换为-
str2 = str1.Replace(' ', '-');
Console.WriteLine("将 str1 所有空格替换为-:"+str2);

// 在第一个空格之后插入@@@
str2 = str1.Insert(n, " peach");
Console.WriteLine("在 str1 第一个空格后插入 peach:" + str2);

// 取第一个空格后的 6 个字符
str2 = str1.Substring(n+1, 6);
Console.WriteLine("取 str1 第一个空格后的 6 个字符:" + str2);

// 以空格为标识符,将字符串分解为若干个新的字符串
char[] chars={' '};
string[] strs = str1.Split(chars);

Console.WriteLine("以空格为标识符,将 str1 分解为:");
for ( int i=0; i<strs.Length; i++ )
    Console.WriteLine(i+": "+strs[i]);

// 输入任意键退出
Console.ReadKey();
}
}
}

```

运行程序, 输出结果如图 A-4 所示。

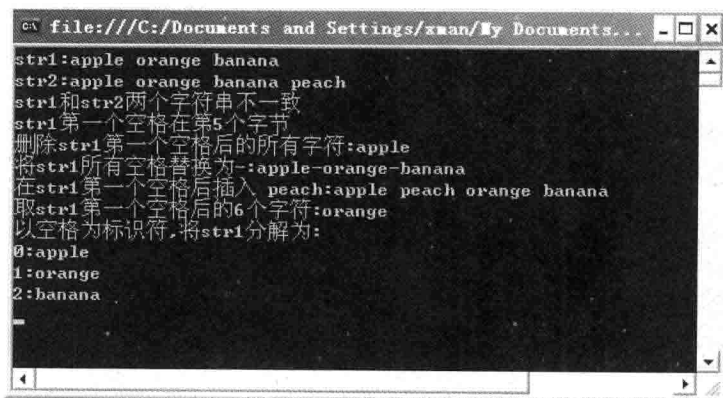


图 A-4 程序输出

本节的示例文件保存在资源文件目录\appendixA_C#\stringsample\中。

A.4 数组和排序

A.4.1 创建数组

数组就像是一个容器，将一连串相同类型的数据按顺序保存起来。数组可以是一维、多维或交错的。数组中的数值类型默认值为零，而引用类型的默认值为 `null`。数组的索引从零开始，具有 `n` 个元素的数组的索引是从 0 到 `n-1`，如下所示：

```

// 声明一个用于存储 int 类型的一维数组并赋值
int[] array1 = new int[2];
array1[0]=1;
array1[1]=9;

// 声明一个数组的同时并给出了 5 个数值
int[] array2 = new int[] { 1, 9, 5, 7, 3 };

// 另一种声明方式
int[] array3 = { 1, 2, 3, 4, 5, 6 };

// 声明二维数组，注意[]括号内的逗号
int[,] multiArray= new int[2, 3];
multiArray[0,0]=19;
multiArray[0,1]=5;
//...

// 声明二维数组的同时给出数值
int[,] multiArray2 = { { 1, 2, 3 }, { 2, 3, 4 } };

```


A.4.2 遍历数组

数组是从抽象基类型 `array` 派生的引用类型。由于此类型实现了 `IEnumerable`，因此可以对 C# 中的所有数组使用 `foreach` 迭代，如下所示：

```
int[] array = new int[] { 1, 2, 3, 4, 5 };

// 使用 for 遍历数组
for (int i = 0; i < 5; i++)
{
    Console.WriteLine( array [i] );
}

// 使用 foreach 遍历, 等同于使用 for
foreach ( int n in array )
{
    Console.WriteLine(n);
}
```

A.4.3 插入、删除、查找

如果在数组中经常需要进行插入或删除操作, 使用 `System.Collections.ArrayList` 会更加方便, 它会自动管理数组的大小。创建 `ArrayList` 和创建一个普通对象一样, 它可以使用 `Add` 方法存储多个数据, 使用 `Remove` 删除指定的元素, 如下所示:

```
// 声明两个对象
Player p1 = new Player();
Player p2 = new Player();

// 使用 new 创建 ArrayList
ArrayList array = new ArrayList();

// 添加元素
array.Add(p1);
array.Add(p2);

// 遍历所有元素
for (int i = 0; i < array.Count; i++)
{
    Player p=(Player)array[i];
}

// 清除元素
```

```
array.Remove(p1);

// 清除所有元素
array.Clear();
```

在获取 `ArrayList` 元素的时候，需要指明元素的类型，使用的时候比较麻烦。C#中的 `System.Collections.Generic.List` 功能和 `ArrayList` 接近，但它是泛型实现，只需要在创建 `List` 的时候声明类型即可，示例代码如下：

```
static void Main(string[] args)
{
    // 随机数
    Random rand = new Random();
    // 元素为 int 类型
    List<int> list = new List<int>();
    for (int i = 0; i < 10; i++)
    {
        // 生成 100 以内的随机数添加到 List 中
        int n = rand.Next(100);
        list.Add(n);
    }

    // 遍历 List
    foreach (int i in list)
    {
        Console.WriteLine(i);
    }
}
```

在数组中查找也是常见的操作，在 `List` 中提供了一个 `Find` 函数，可以实现最基本的查找需求，代码如下：

```
class Player
{
    public int id { set; get; }
}

List<Player> list = new List<Player>();
// 此处省略... 在 list 中插入很多 Player 对象

// 在 List 中查找 id 等于 1 的 Player 对象，此处 x 指 List 中的 Player 对象
list.Find((x) => x.id == 1);
```

A.4.4 排序

排序是比较常见的数组操作,使用 List 对元素排序非常简单,如果元素是内置类型,只需要调用一个 sort 函数,默认会从小到大进行排序。

```
list.Sort();
```

很多时候,元素不是内置类型,可能是一个对象,通过比较对象的某个成员进行排序,这时需要通过继承 ICompare 接口创建自定义比较函数完成排序操作,示例代码如下:

```
// 定义一个 Player 类
class Player
{
    public int id;
    public int life;

    public Player(int iId, int iLife)
    {
        this.id = iId;
        this.life = iLife;
    }
}

// 比较
public class PlayerCompare : IComparer<Player>
{
    // 比较 Player 的 id 大小
    int IComparer<Player>.Compare(Player a, Player b)
    {
        return a.id.CompareTo(b.id);
    }
}
```

对象排序仍然是调用 sort 方法,但需要加入用于比较的函数,示例代码如下:

```
// 创建一个 List
List<Player> player_list = new List<Player>();
for (int i = 0; i < 10; i++)
{
    int n = rand.Next(100); // 生成随机数
    int n2 = rand.Next(100); // 生成随机数
    Player p = new Player(n, n2);
    // 在 List 中添加创建的 Player 实例
    player_list.Add(p);
}
```

```
// 对 List 进行排序
player_list.Sort( new PlayerCompare());

foreach (Player p in player_list)
{
    Console.WriteLine( "player id:" + p.id); // 打印出排过序的 Player id
}
```

在前面的示例，我们通过 Player 的 id 的值进行比较排序，如果我们还希望按 life 值的大小进行排序，可以定义一个枚举，根据不同的枚举类型进行不同的比较，代码如下：

```
public class PlayerCompare : IComparer<Player>
{
    // 定义比较类型
    public enum CompareType
    {
        id,
        life
    }
    CompareType compareType;
    public PlayerCompare(CompareType ct)
    {
        compareType = ct;
    }
    int IComparer<Player>.Compare(Player a, Player b)
    {
        if ( compareType == CompareType.id )
            return a.id.CompareTo(b.id);
        else if (compareType == CompareType.life)
            return a.life.CompareTo(b.life);

        return 0;
    }
}
```

这一次，在排序的时候需要指定比较的类型，比如按 life 值大小排序：

```
player_list.Sort( new PlayerCompare(PlayerCompare.CompareType.life));
```

虽然现在可以按对象中特定的值排序，但还有一种略复杂的情况，比如我们希望按 id 优先排序，但如果 id 的值相同，则再按 life 的值排序决定先后顺序，这种情况我们可以在一个循环中依次按类型比较，如果 id 的值不相同，比较结束，否则进入下一轮按 life 的值比较，代码如下：

```
public class PlayerCompare : IComparer<Player>
{
```

```

// 将所有排序类型按先后顺序加入一个 List 中
private List<CompareType> compareList = new List<CompareType>();
// 定义比较类型
public enum CompareType
{
    id,
    life
}
public PlayerCompare()
{
    compareList.Add(CompareType.id);
    compareList.Add(CompareType.life);
}
int IComparer<Player>.Compare(Player a, Player b)
{
    int result = 0;
    foreach (CompareType ct in compareList)
    {
        if (ct == CompareType.id)
            result = a.id.CompareTo(b.id);
        else if (ct == CompareType.life)
            result = a.life.CompareTo(b.life);

        // 如果比较不相等则停止后面的比较
        if (result != 0)
            break;
    }
    return result;
}
}

```

本节的示例文件保存在资源文件目录\appendixA_C#\ListApp\中。

A.5 I/O 操作

I/O 操作主要是针对计算机文件的创建和修改。在游戏中,最常见的功能是将游戏记录写入到一个文件中,并可以随时读取,这个过程主要包括创建文件,读写文件和删除文件。

A.5.1 写文件

写文件即是把数据存入到一个文件的过程。在下面的代码示例中,首先用 **string** 字符串定义了文件的存储路径和文件的名称,然后创建一个 **string** 数组保存两条数据,最后使用

IO.File.WriteAllLines 方法将数据保存到文件中。

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // 存储文件的路径和文件名
            string file = "d:\\test.dat";

            string[] data=new string[2];
            data[0] = "第一条信息";
            data[1] = "第二条信息";

            // 将两条数据写入 D 盘的 test.dat 文件中
            System.IO.File.WriteAllLines(file, data);

            // 输入任意键退出
            Console.ReadKey();
        }
    }
}
```

运行程序，然后在 D 盘下找到 test.dat 文件，用文本编辑器打开，会看到写入的字符。本节的示例文件保存在资源文件目录\appendixA_C#\writefile\中。

A.5.2 读文件

读取文件的过程通常要参考写文件的过程，也就是要按照写文件时的文本格式将数据读出来。在下面的示例中，首先要使用 Exists 方法判断文件是否存在，然后将数据读入到 string 数组中，最后在控制台显示出来，如图 A-5 所示。

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
```

```

{
    static void Main(string[] args)
    {
        // 存储文件的路径和文件名
        string file = "d:\\test.dat";

        // 判断 test.dat 文件是否存在
        if (System.IO.File.Exists(file))
        {
            // 用于存储读入的数据
            string[] data = new string[2];

            // 将 C 盘的 test.dat 文件中的数据读出
            data = System.IO.File.ReadAllLines(file);

            // 打印数据
            Console.WriteLine(data[0]);
            Console.WriteLine(data[1]);
        }

        // 输入任意键退出
        Console.ReadKey();
    }
}

```

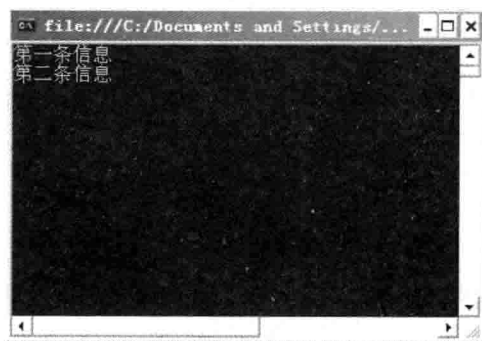


图 A-5 显示读入的数据

本节的示例文件保存在资源文件目录\appendixA_C#\readfile\中。

A.5.3 删除文件

删除文件只需要使用 `IO.File.Delete` 方法即可，如下所示：

```
// 存储文件的路径和文件名
```

```
string file = "d:\\test.dat";

// 判断 test.dat 文件是否存在
if (System.IO.File.Exists(file))
{
    // 删除文件
    System.IO.File.Delete(file);
}
```

A.5.4 读写 bytes

无论是 int、float 或是其他数据类型，都可以将其转换为 byte（字节）存入到数组中，然后写入到文件里，读取回来时再转为正确的数据类型使用，如下所示：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    class Program
    {
        static void Main(string[] args)
        {
            // 存储文件的路径和文件名
            string file = "d:\\test.dat";

            // 需要存储的数据
            int data1 = 100;
            float data2 = 0.1f;

            // 记录数据的位置
            int index = 0;

            // 用于存储所有数据的 byte 数组
            byte[] bytes = new byte[256];

            // 将 data1 转为 byte 数组
            byte[] b = BitConverter.GetBytes(data1);

            // 将 data1 数据存入 bytes 数组中
            b.CopyTo(bytes, index);

            // 一个 int 类型占 4 个 byte
```

```

        index += 4;

        // 将 data1 转为 byte 数组
        b = BitConverter.GetBytes(data2);

        // 将 data2 数据存入 bytes 数组中
        b.CopyTo(bytes, index);

        // 一个 float 类型占 4 个 byte
        index += 4;

        // 将数据存储到 d:\\test.dat
        System.IO.File.WriteAllBytes(file, bytes);

        // 判断 test.dat 文件是否存在
        if (System.IO.File.Exists(file))
        {
            bytes = new byte[256];

            // 读入数据
            bytes=System.IO.File.ReadAllBytes(file);

            // 取出头 4 个字节作为 int
            int readdata1 = BitConverter.ToInt32(bytes, 0);

            // 再取出 4 个字节作为 float
            float readdata2 = BitConverter.ToSingle(bytes, 4);

            // 打印数据
            Console.WriteLine(readdata1);
            Console.WriteLine(readdata2);
        }

        // 输入任意键退出
        Console.ReadKey();
    }
}

```

本节的示例文件保存在资源文件目录\appendixA_C#\iobytes\中。

A.6 委托

A.6.1 委托与事件

在 C# 中, 委托就像 C 语言中的函数指针, 可以将函数作为参数传递, 比较典型的应用实例如创建回调函数。

委托的使用, 首先要使用 `delegate` 关键字声明一个委托函数, 并定义它的返回值和参数列表。在实际使用的时候, 首先要定义一个委托实例, 可以称作事件, 并通过 `+` 号将事件与具体的函数相关联, 最后当触发事件, 则会通知所有相关联的函数。下面是一个示例, 在一个按钮类中创建一个响应按钮的委托事件:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    /// <summary>
    /// 定义一个按钮类
    /// </summary>
    public class Button
    {
        // 按钮的 ID
        public int ID {get;set;}

        // 按钮是否处于点击状态
        public bool isClick {get;set;}

        // 定义一个委托类型
        public delegate void OnClick(Button button);

        // 定义一个委托事件
        public event OnClick onClick;

        public Button( int id )
        {
            ID = id;
            isClick = false;
        }

        // 更新按钮, 如果处于点击状态, 则触发 onClick 委托事件
        public void Update()
```



```

        {
            if (isClick)
            {
                onClick(this);
            }
        }
    }
}

class Program
{
    static void Main()
    {
        List<Button> buttonlist = new List<Button>();
        for (int i = 0; i < 10; i++) // 创建 10 个按钮
        {
            Button button = new Button(i);
            buttonlist.Add(button);

            // 设置按钮点击事件的回调函数
            button.onClick += OnButtonClick;

        }

        // 将一个按钮的点击状态设为 true
        buttonlist[4].isClick = true;

        // 更新所有按钮, 如果按钮处于点击状态, 将产生回调
        foreach (Button button in buttonlist)
        {
            button.Update();
        }

        // 输入任意键退出
        Console.ReadKey();
    }
    // 按钮事件的回调函数
    static void OnButtonClick(Button button)
    {
        // 输出
        Console.WriteLine("Button ID:" + button.ID);    // 输出按钮 ID
    }
}

```

本节的示例文件保存在资源文件目录\appendixA_C#\delegatesample\中。

A.6.2 泛型委托

C#的委托支持泛型参数，也就是说委托可以对应任何类型的参数列表，如下所示：

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    public class ImplementingClass
    {
        // 定义委托调用的方法
        public float WhichIsBig(int a, float b)
        {
            if (a > b)
                return a;
            else
                return b;
        }
    }

    class Program
    {
        // 定义一个泛型委托,它的参数可以是任意类型
        public delegate float CallDelegate<T, S>(T t, S s);

        static void Main(string[] args)
        {
            //实例化委托
            CallDelegate<int, float> delegateBig = new CallDelegate<int, float>(
                new ImplementingClass().WhichIsBig);

            //调用委托方法,执行 ImplementingClass WhichIsBig
            float big = delegateBig(10, 20.5f);

            // 输出最大值 20.5
            Console.WriteLine(big);

            // 输入任意键退出
            Console.ReadKey();
        }
    }
}
```

```

    }
}
}

```

本节的示例文件保存在资源文件目录\appendixA_C#\delegategeneric\中。

A.6.3 Action

C#提供了一个叫 **Action** 的关键字可以用来实现和 **delegate** 差不多的功能,但使用起来更加方便。在下面的示例中,有一个叫作 **NetworkManager** 的类,它可能会用来实现一些网络功能,它有一个函数叫作 **SendMessage**,用来向远程发送消息,在使用这个函数的时候,我们给它一个 **Action** 参数,当 **NetworkManager** 得知消息已发送到远程并得到反馈,则触发这个 **Action** 参数,即类似完成回调函数的作用,代码如下:

```

using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication1
{
    public class NetworkManager
    {
        // 一个队列,存储带有 string 参数的回调函数
        Queue<Action<string>> Callbacks;

        public NetworkManager()
        {
            Callbacks = new Queue<Action<string>>();
        }
        // 定义一个 Action
        public void SendMessage( Action<string> callback )
        {
            // 伪码,向远程计算机发送数据,将回调函数放到队列中
            Callbacks.Enqueue(callback);
        }

        public void Run()
        {
            // 伪码,收到远程返回的信息
            string getData="hello,world";

            // 从队列中获取回调函数
            Action<string> callback = Callbacks.Dequeue();
            callback(getData);
        }
    }
}

```

```

    }
}

class Program
{
    static void Main(string[] args)
    {
        NetworkManager networkMgr = new NetworkManager();

        // 发送消息
        networkMgr.SendMessage((string data) =>
        {
            // 获得远程的消息后回调
            Callback(data);
        });

        // 获得远程的消息
        networkMgr.Run();

        Console.ReadLine();
    }

    // 回调函数
    static void Callback(string data)
    {
        Console.WriteLine(data);
    }
}
}

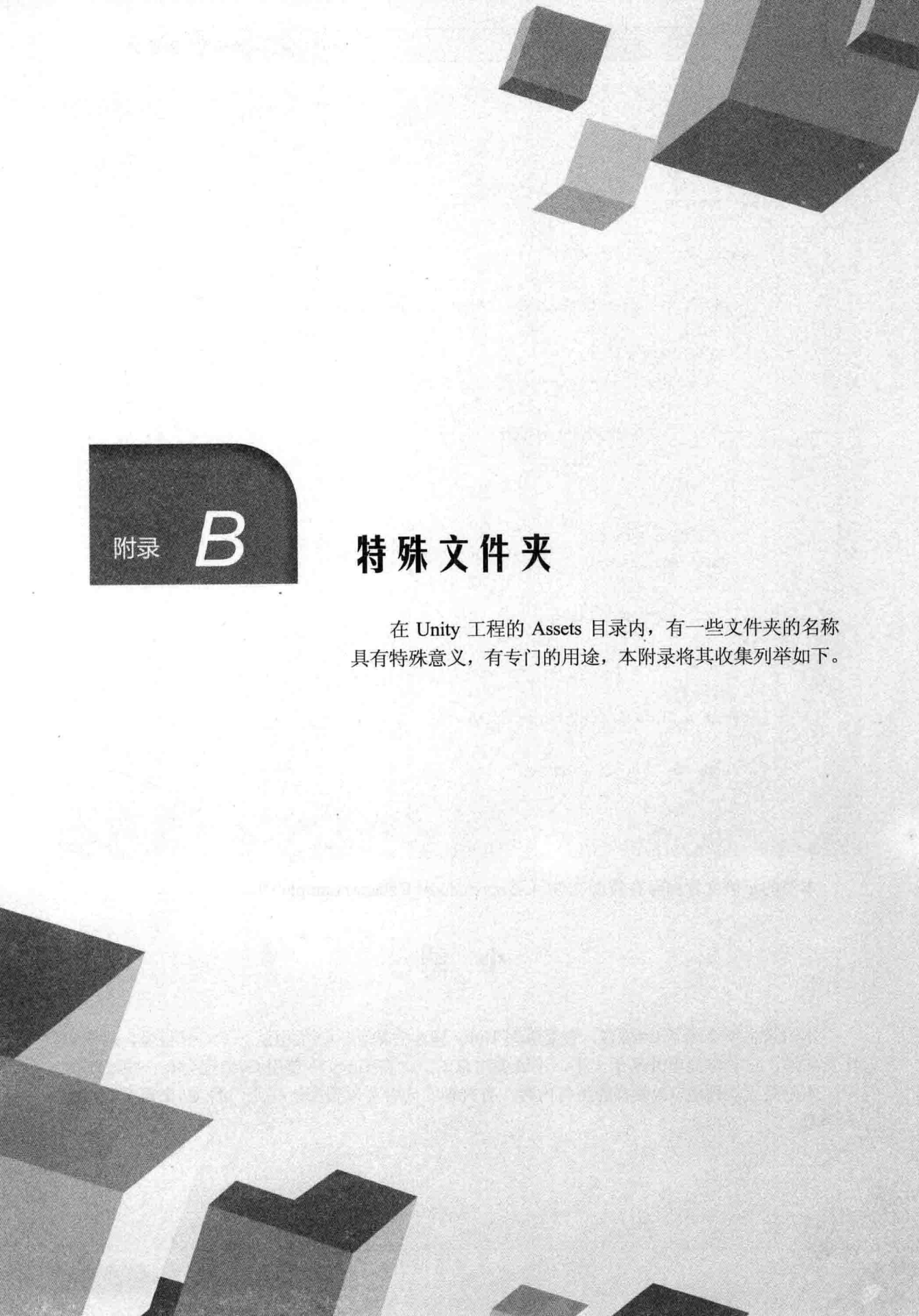
```

本节的示例文件保存在资源文件目录\appendixA_C#\actionsample\中。

小 结

本附录主要介绍了 C#语言，它是编写 Unity 脚本的基础。C#的语法与 C/C++很像，对于程序员来说，它非常简单并易于上手，不过要注意的是，在 Unity 中使用 C#编程会有一些限制。

本附录无法覆盖 C#语言的所有内容，有兴趣的读者可以查阅一些专门的 C#书籍或资料深入学习。



附录 *B*

特殊文件夹

在 Unity 工程的 Assets 目录内，有一些文件夹的名称具有特殊意义，有专门的用途，本附录将其收集列举如下。

ActionScript

存放 Flash 的 ActionScript 脚本。当游戏被导出为 Flash 格式的时候，这里的脚本会自动替换指定的 C#脚本。

Editor

存放编辑器脚本。

Gizmos

通常是存放 TIF 格式的图片，在 OnDrawGizmos 函数内使用 Gizmos.DrawIcon 将其画为图标在场景中显示。

Plugins/Android

存放 Android 插件，包括.jar 或 XML 文件等。

Plugins/iOS

存放 iOS 插件，包括.a 或.m、.mm 文件等。

Resources

存放使用 Resources.Load()动态读取的资源，它们可以是图片、模型等不同类型的资源。

Standard Assets

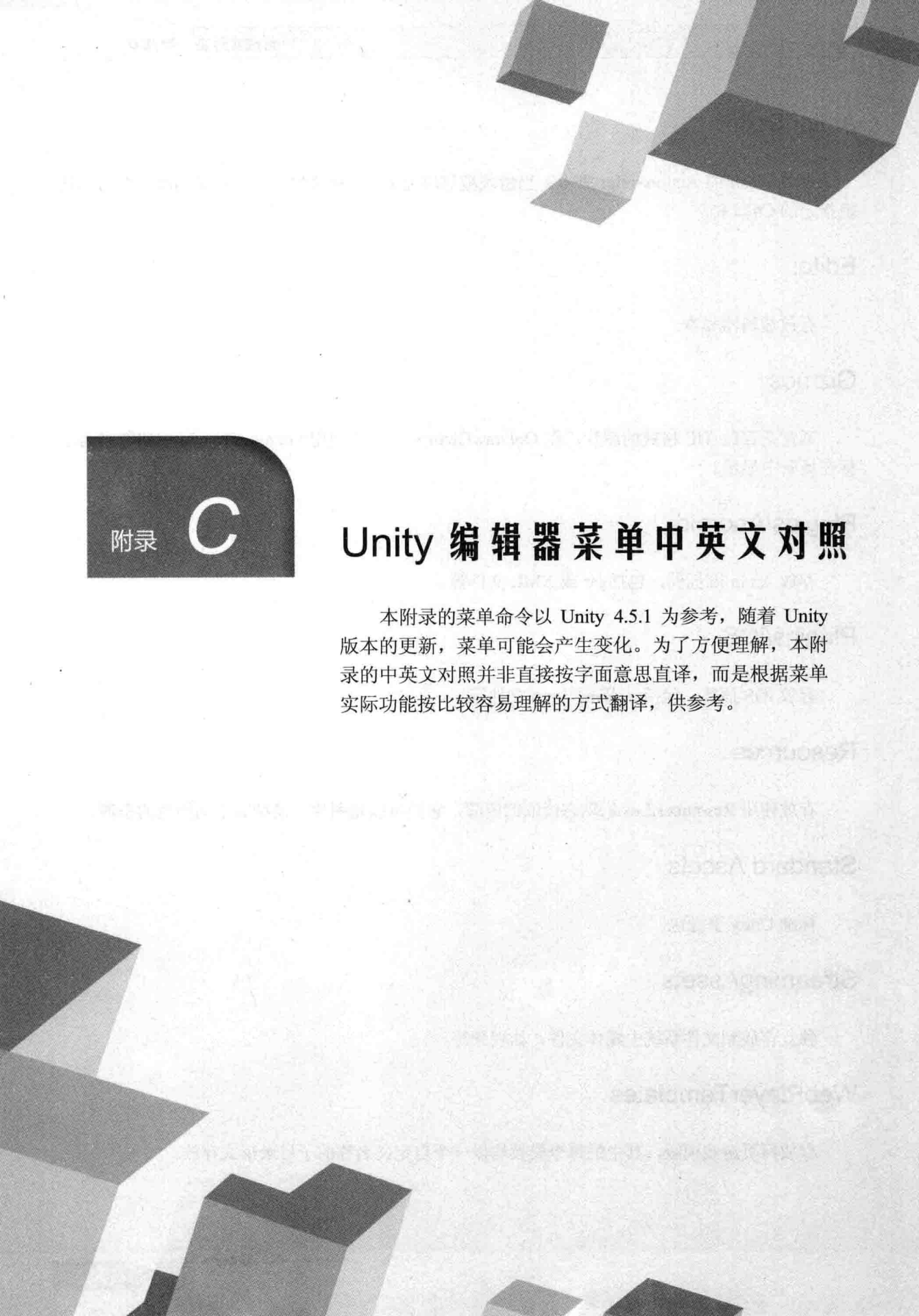
标准 Unity 资源包。

StreamingAssets

独立存放到文件系统中媒体文件，如视频等。

WebPlayerTemplates

存放网页游戏模版，其中的每个模板均以一个自定义名称的子目录形式存放。



附录

C

Unity 编辑器菜单中英文对照

本附录的菜单命令以 Unity 4.5.1 为参考，随着 Unity 版本的更新，菜单可能会产生变化。为了方便理解，本附录的中英文对照并非直接按字面意思直译，而是根据菜单实际功能按比较容易理解的方式翻译，供参考。

File: 文件

New Scene: 新场景
 Open Scene: 打开场景
 Save Scene: 保存当前场景
 Save Scene as: 另存当前场景
 New Project: 新建工程
 Open Project: 打开工程
 Save Project: 保存工程



如果是非正常退出 Unity，又没有使用该选项保存设置，很可能会丢失编辑器内的设置和修改。

Build Settings: 输出设置
 Build & Run: 创建游戏并运行
 Exit: 退出编辑器

Edit: 编辑

Undo: 返回前一个操作
 Redo: 取消返回操作
 Cut: 剪切
 Copy: 复制
 Paste: 粘贴
 Duplicate: 复制为另一个
 Delete: 删除
 Frame Selected: 将选中物体在视图中最大化显示
 Lock View to Selected: 锁定视图到选中物体
 Find: 查找
 Select All: 选择全部
 Preference: 首选项
 Module: 更新、下载模块
 Play: 播放游戏
 Pause: 暂停运行中的游戏
 Step: 单步运行
 Selection: 选择
 Project Settings: 工程设置
 Input: 键盘鼠标的输入配置

- Tags and Layers: 标记和层
- Audio: 音效
- Time: 时间

**提示**

修改 Time Scale 的值会改变游戏的运行速度，降低这个值，则会使游戏看起来像慢动作一样。

- Player: 输出游戏设置
- Physics: 物理
- Physics 2D: 2D 物理
- Quality: 输出质量

**提示**

这里的设置主要是影响游戏的表现品质和性能，在发布最后的游戏之前，请务必到这里设置，如阴影、灯光、骨骼影响等。

- Graphics: 图像
- Network: 网络
- Editor: 编辑器

**提示**

在多人合作的 Unity 项目中，如果使用了 SVN 之类的版本管理软件，最好将 Version Control 设为 Visible Meta Files，这时会在工程目录内看到与每个资源对应的 Meta 数据文件。将 Asset Serialization 设为 Force Text，这样版本管理软件才能有效地对比文件变化。

- Script Execution Order: 脚本执行顺序

Render Settings: 渲染设置

Network Emulation: 网络模拟

Graphics Emulation: 图像模拟

Snap Settings: 捕捉设置

Assets: 资源

Create: 创建

- Folder: 文件夹
- JavaScript: JavaScript 脚本
- C# Script: C#脚本
- Boo Script: Boo 脚本
- Shader: Shader
- Compute Shader: 运行在正常渲染通道之外的 Shader

- Prefab: 预置文件
- Material: 材质
- Cubemap: 主要用来制作反射效果的贴图
- Lens Flare: 镜头光晕
- Render Texture: 使用摄像机实时渲染、更新的贴图
- Animator Controller: 动画控制器 (针对 Mecanim 动画)
- Animation: 动画控制器 (针对老式的动画)
- Animator Override Controller: 动画控制器重载 (继承某个动画控制器的配置)
- Avatar Mask: 角色动画遮罩, 用来打开或关闭角色身体上的某些动画细节
- Physic Material: 物理配置材质
- Physics 2D Material: 2D 物理配置材质
- GUI Skin: GUI 配置文件
- Custom Font: 自定义字体

Show in Explorer: 打开资源管理器

Open: 打开

Delete: 删除

Import New Asset: 导入原始资源

Import Package: 导入.unity 格式的资源包

Export Package: 导出选中的资源为.unity 格式的资源包

Find References In Scene: 选中场景中所有与当前选择 Prefab 相关联的物体

Select Dependencies: 选中当前场景中所选物体的 Prefab

Refresh: 刷新

Reimport: 重新导入所选资源

Reimport All: 重新导入全部资源

Sync MonoDevelop Project: 同步 MonoDevelop 工程

GameObject: 游戏体

Create Empty: 创建空物体

Create Empty Child: 创建当前物体的子空物体

Create Other: 创建其他的

- Particle System: 粒子系统
- Camera: 摄像机
- GUI Text: GUI 文字
- GUI Texture: GUI 贴图
- 3D Text: 3D 的文字
- Directional Light: 方向光
- Point Light: 点光
- Spotlight: 聚光灯

- Area Light: 区域光
- Cube: 立方体
- Sphere: 球体
- Capsule: 胶囊体
- Cylinder: 圆柱体
- Plane: 平面
- Quad: 四方面 (与平面形状一样, 但只有 4 个顶点)
- Sprite: 2D 精灵图片
- Cloth: 布料
- Audio Reverb Zone: 混响音效区域 (模拟不同场景中的音效效果)
- Terrain: 地形
- Ragdoll: 角色碰撞物理
- Tree: 树
- Wind Zone: 风力影响区域

Center On Children: 使父物体与子物体对齐, 同时子物体坐标归零

Make Parent: 使先选中的一个物体成为另一个物体的父物体

Clear Parent: 使选中的子物体脱离父物体

Apply Changes to Prefab: 更新当前物体的 Prefab

Break Prefab Instance: 断开当前物体与 Prefab 的关联

Move to View: 移动选中物体到视图中心

Align with View: 使物体与视图方向对齐, 通常用来对齐摄像机到当前视图角度

Align View to Selected: 使当前视图与所选物体位置、角度一致

Component: 组件

Add: 添加组件

Mesh: 多边形

- Mesh Filter: 将多边形数据传输到渲染器
- Text Mesh: 文字多边形
- Mesh Renderer: 多边形渲染器

Effects: 特效

- Particle System: 粒子系统
- Trail Renderer: 轨迹效果
- Line Renderer: 线段效果
- Lens Flare: 镜头光晕效果
- Halo: 光环效果
- Projector: 投射器 (将 2D 图片投射到 3D 表面, 比如假阴影)
- Legacy Particles: 老式粒子

Physics: 物理

- Rigidbody: 刚体
- Character Controller: 角色控制器
- Box Collider: 立方体碰撞
- Sphere Collider: 球形碰撞
- Capsule Collider: 胶囊形碰撞
- Mesh Collider: 多边形碰撞
- Wheel Collider: 轮子碰撞
- Terrain Collider: 地面碰撞
- Interactive Cloth: 交互布料模拟
- Skinned Cloth: 绑定的布料模拟
- Cloth Renderer: 布料渲染器
- Hinge Joint: 铰链关节
- Fixed Joint: 固定关节
- Spring Joint: 弹簧关节
- Character Joint: 角色关节
- Configurable Joint: 可配置关节
- Constant Force: 持续的力

Physics 2D: 2D 物理

- Rigidbody 2D: 2D 刚体
- Circle Collider 2D: 2D 圆形碰撞
- Box Collider 2D: 2D 方形碰撞
- Edge Collider 2D: 2D 边碰撞
- Polygon Collider 2D: 2D 多边形碰撞
- Spring Joint 2D: 2D 弹簧关节
- Distance Joint 2D: 2D 距离关节
- Hinge Joint 2D: 2D 铰链关节
- Slider Joint 2D: 2D 滑动关节

Navigation: 人工智能寻路

- Nav Mesh Agent: 寻路体
- Off Mesh Link: 寻路地形连接 (用于不连续的地形)
- Nav Mesh Obstacle: 寻路地形中的阻挡物

Audio: 音效

- Audio Listener: 收听器 (一个场景只能有一个)
- Audio Source: 音源播放器
- Audio Reverb Zone: 混响音效区域 (模拟不同场景中的音效效果)
- Audio Low Pass Filter: 音效低通滤波器
- Audio High Pass Filter: 音效高通滤波器
- Audio Echo Filter: 音效回声滤波器
- Audio Distortion Filter: 音效扭曲滤波器

- Audio Reverb Filter: 音效混响滤波器
- Audio Chorus Filter: 音效合声滤波器

Rendering: 渲染

- Camera: 摄像机
- Skybox: 天空盒
- Flare Layer: 光晕层
- GUI Layer: UI 层
- Light: 灯光
- Light Probe Group: 灯光 Probe 组
- Occlusion Area: 遮挡区域
- Occlusion Portal: 遮挡入口
- LODGroup: LOD 组
- Sprite Renderer: 精灵渲染
- GUITexture: UI 贴图
- GUIText: UI 文字

Miscellaneous: 杂项

- Animator: 动画 (针对 Mecanim 动画)
- Animation: 动画 (针对老式 Unity 动画)
- Network View: 网络 (针对 Unity 内置的基于 Raknet 的网络功能)
- Wind Zone: 风力影响区域

Scripts: 开发者创建的脚本

Window: 窗口

Next Window: 关于 Unity

Previous Window: 管理授权

Layouts: 布局

Scene: 场景

Inspector: 查看 (详细信息)

Hierarchy: 层级

Project: 工程

Animation: 动画编辑器 (针对老式 Unity 动画)

Profiler: 性能分析器

Asset Store: Unity 插件、资源在线商店

Version Control: 版本控制

Animator: 动画编辑器 (针对 Mecanim 动画)

Sprite Editor: 精灵 (2D 图片) 编辑器

Sprite Packer: 精灵打包器

Lightmapping: 光照贴图

Occlusion Culling: 环境遮挡（用于性能优化，使计算忽略被其他 3D 模型挡住的 3D 模型）

Navigation: 人工智能寻路

Console: 显示 Log 内容的控制台

Help: 帮助

About Unity: 关于 Unity

Manage License: 管理授权

Unity Manual: Unity 基本使用说明文档

Scripting Reference: 脚本参考文档

Unity Forum: Unity 技术论坛

Unity Answers: Unity 疑难解答

Unity Feedback: Unity 反馈

Welcome Screen: 欢迎屏幕

Check for Updates: 检查更新

Release Notes: 版本说明

Report a Bug: 提交 Bug