

Cocos2d-x 3.X 手游开发实例详解

于浩洋 著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书以 Cocos2d-x V3.0 系列版本为框架,通过实例讲解手机游戏的开发知识和方法,着重通过实例提高读者的开发动手能力。涉及的内容包括:环境搭建、交叉编译以及打包部署;Cocos2d-x 数据类型和基本概念的使用,如场景、导演、精灵等;使用 Cocos2d-x 创建用户界面,如文本、菜单、滚动框等基本控件,以及一些扩展控件的使用;使用 Cocos2d-x 创建动作,比如让精灵移动、跳跃、旋转,以及如何创建按顺序或同时进行的动作;使用 Cocos2d-x 播放、停止、暂停、继续播放背景音乐和音效;Cocos2d-x 使用的触摸事件机制;在游戏中存储数据的几种方式;使用 HTTP、Sockets、WebSockets 进行网络编程;在游戏中使用物理引擎 Box2D、使用瓷砖地图、使用 CocosBuilder 制作游戏界面等;Cocos2d-x 中的纹理和动画。最后通过两个完整的游戏开发实例讲解使用 Cocos2d-x 开发游戏的全过程。

本书实例丰富,代码完备,知识点清晰简洁。本书所有实例均提供完整代码下载,书后附有代码清单概要,非常方便读者查找使用。此外作者还将 Cocos2d-x 3.X 各版本间的区别用实例做了详细说明,且将在博客不断更新。

本书适合对 Cocos2d-x 感兴趣及有志于学习和从事移动平台游戏开发的读者阅读参考。

未经许可,不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有,侵权必究。

图书在版编目(CIP)数据

Cocos2d-x 3.X 手游开发实例详解 / 于浩洋著. —北京:电子工业出版社, 2014.9
ISBN 978-7-121-23998-4

I. ①C… II. ①于… III. ①移动电话机—游戏程序—程序设计 ②便携式计算机—游戏程序—程序设计 IV. ①TN929.53 ②TP368.32

中国版本图书馆 CIP 数据核字(2014)第 179534 号

责任编辑:董 英

印 刷:北京中新伟业印刷有限公司

装 订:河北省三河市路通装订厂

出版发行:电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本:787×980 1/16 印张:22.25 字数:410 千字

版 次:2014 年 9 月第 1 版

印 次:2014 年 9 月第 1 次印刷

印 次:3000 册 定价:59.00 元

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

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

服务热线:(010) 88258888。

前言

记得第一次接触移动互联网是在 2008 年左右，当时互联网应用相当普及，市场竞争进入红海，绝大多数人认为移动互联将成为下一个增长点，是很大的蓝海，社会将进入 3G 时代。大到三大运营商和一线互联网公司，小到小公司和个人开发者都想进入移动互联，从 3G 大蛋糕中分到一块利益。转眼几年过去，正如当时预料的那样，移动互联已经深入人们的生活当中，4G 也悄悄来临。在移动互联的发展过程中，苹果公司起到了不可磨灭的作用，因为它发布了能承载移动互联功能的新概念智能手机 iPhone。于是乎，诞生了一大批移动应用和移动游戏的开发商和个人开发者。其中以《疯狂的小鸟》为代表的第一批手机游戏进入人们的视线并大获成功，之后手机游戏以比雨后春笋还疯狂很多倍的速度成长起来。各种类型的游戏出现在市场中，例如《我叫 MT》《大掌门》《找你妹》及腾讯的天天游戏系列。

在手机游戏快速成长的过程中，手机游戏开发框架也跟着发展起来，其中最著名的就是 Cocos2d 家族，Cocos2d-iphone、Cocos2d-js、Cocos2d-java 等，其中 Cocos2d-iphone 使用数最多。但是 Cocos2d-iphone 是用 Objective-C 开发的，只能用于开发 iPhone 平台的游戏，如果想发布到 Android 平台，就要使用另外一种语言，几乎重写所有代码。于是，中国有位叫王哲的“大神”，使用 C++ 重新实现了一遍 Cocos2d-iphone 的 API，使用 C++ 开发的游戏不仅效率高，也能发布到 iOS、Android、in、Linux 等多个平台，真正实现了编写一次、多平台运行的目的。

Cocos2d-x 3.0 系列统一修改了类和接口（比如把类前面的 CC 去掉），使代码更规范、更美观。3.0 系列重写了纹理渲染方式，不再会产生成倍消耗内存的情况，使游戏运行效率更高、更稳定。本书以 Cocos2d-x 3.0 版本为基础，记录整理了笔者在学习和使用 Cocos2d-x 中遇到的问题和常用功能，以实践为主，着重提高读者的动手开发能力，让读者从入门手游的菜鸟成为游戏开发界的大神。

本书的主要内容如下。

第 1 章介绍 Cocos2d-x 的优缺点和目录结构，本书案例需要的环境搭建、交叉编译以及打包部署。

第 2 章讲解了 Cocos2d-x 中常用的数据类型和方法及一些宏定义。

第 3 章讲解了 Cocos2d-x 的基本概念，即基础节点 Node、相机、导演、场景、布景、精灵。

接着由浅入深，第 4 章讲解了如何使用 Cocos2d-x 创建用户界面，如文本、菜单、滚动框等基本控件，以及一些扩展控件，如取色器、计步器、复选框等。

之后笔者用很多篇幅在第 5 章讲解了使用 Cocos2d-x 创建动作的方式，因为动作在游戏中是必不可少的重中之重，内容包括移动、跳跃、旋转等简单动作，以及按顺序执行或同时执行的复杂动作。

音乐在游戏中也不可或缺，好的音效给人耳目一新的感觉，本书第 6 章讲解了使用 Cocos2d-x 播放、停止、暂停、继续播放背景音乐和音效。

第 7 章介绍了瓷砖地图相关的内容，包括瓷砖地图的概念、如何制作瓷砖地图，如何在游戏使用、操作瓷砖地图等。

第 8 章讲解了 Cocos2d-x 中的事件机制，包括触摸事件、鼠标事件、键盘事件，以触摸事件为主要内容。智能手机大多数都是触摸屏，要监听用户的触摸行为，然后调用相应的行为动作，Cocos2d-x 提供了一套触摸事件机制，并且 3.0 系统重写了该机制，使监听用户触摸行为更加方便快捷，同时 Cocos2d-x 还支持多点触摸机制，增加玩家与手机的交互乐趣。

除此之外，进行游戏时还产生很多数据，这些数据有时需要保存下来以便下次使用，所以第 9 章介绍了多种保存数据的方式，比如 UserDefaults、Plist、本地数据库 SQLite 等。

当开发网络游戏时，客户端要不断地跟服务端进行数据交换，第 10 章讲解了如何使用 HTTP、Sockets、WebSockets 进行网络编程。

第 11 章介绍了物理引擎，它可以使游戏更加接近现实世界，给玩家更真实的体验。

第 12 章讲解了纹理和动画，介绍了纹理的渲染方式并提供了几个提高渲染效率的方法。动画部分讲解了如何在游戏中播放帧动画、plist 动画和骨骼动画。书中还穿插介绍了一些工具的使用方法，比如使用 bitmap font generator 制作自定义字体，使用 Tiled Map Editor 制作编辑瓷砖地图，使用 SQLiteStudio 管理 SQLite 数据库，使用 CocoStudio 制作编辑 UI、动画、场景等。

第 13 章讲解了休闲游戏 2048 的完整开发过程，核心内容包括使用 CocoStudio 制作 UI 界面，UI 界面与逻辑代码的交互，使用 UserDefaults 存取游戏数据，数字方块移动、合并的逻辑等。

第 14 章讲解了水浒卡牌游戏的完整开发过程，挑选了其中几个典型的功能进行了详细讲解，包括登录界面、公共菜单、游戏首页、英雄列表、地图、推图和战斗界面等。本章内容对于游戏开发人员具有很高的参考价值。

本书所有实例代码在附录 A 中有详细说明，读者可到博文视点网站下载，地址：www.broadview.com.cn/23998。

附录 B 详细介绍了 Cocos2d-x 3.X 主要版本间的区别，用实例详细说明各个功能点的使用方法。

最后衷心感谢读者的支持，如果有问题或想法可以到博客上留言交流，我的博客是 <http://blog.watchtech.net>。

于浩洋

目 录

第 1 章 准备	1
1.1 Cocos2d-x 简介	1
1.2 Cocos2d-x 架构和目录结构	6
1.3 环境搭建	7
1.3.1 Windows 开发环境搭建	7
1.3.2 Mac 开发环境搭建	11
1.3.3 创建新项目	12
1.3.4 在 Android 上调试项目	14
1.3.5 打包 APK 文件	22
第 2 章 Cocos2d-x 常用数据和方法	23
2.1 C++数据类型	23
2.2 Cocos2d-x 封装的数据类型	24
2.2.1 布尔型 Bool 的使用	24
2.2.2 整型 Integer 的使用	25
2.2.3 浮点型 Double、Float 的使用	25
2.2.4 字符串 String 的使用	25
2.2.5 数组 Array 的使用	26
2.2.6 点 Point 的使用	27
2.2.7 尺寸 Size 的使用	28
2.2.8 矩形 Rect 的使用	29
2.2.9 字典 Dictionary 的使用	31
2.3 常用宏定义	32

2.3.1	数学相关宏的使用	32
2.3.2	断言宏 CCAsset 的使用	33
2.3.3	数组遍历宏 CCARRAY_FOREACH 和 CCARRAY_FOREACH_REVERSE 的使用	33
2.3.4	字典遍历宏 CCDICT_FOREACH 的使用	35
2.3.5	对象创建方法宏 CREATE_FUNC 的使用	36
2.3.6	属性定义宏 CC_PROPERTY 的使用	37
2.3.7	命名空间宏	39
2.4	Cocos2d-x 中的坐标和坐标系	39
2.4.1	OpenGL 坐标系和屏幕坐标系	39
2.4.2	锚点和位置的使用	40
2.4.3	节点坐标系和世界坐标系的相互转换	42
第 3 章	Cocos2d-x 核心概念	45
3.1	基础节点	46
3.1.1	Node 简介	46
3.1.2	Node 应用举例之移除节点	47
3.2	相机	47
3.2.1	相机简介	47
3.2.2	使用 CCCamera 循环缩放点	48
3.3	导演	49
3.3.1	导演 Director 简介	49
3.3.2	Director 常用功能举例	50
3.4	场景	51
3.4.1	场景定义	51
3.4.2	创建显示战斗场景	52
3.4.3	动态切换多个场景	53
3.5	布景	56
3.5.1	布景定义	56
3.5.2	使用 Layer 模拟 Windows Phone 主界面	57
3.6	精灵	59

第 4 章 Cocos2d-x 用户界面	61
4.1 文本渲染	61
4.1.1 制作 fnt 格式字体	62
4.1.2 使用 LabelBMFont 显示文本	66
4.1.3 使用 LabelTTF 显示文本	67
4.1.4 使用 LabelAtlas 显示文本	70
4.2 菜单	71
4.2.1 菜单和菜单项的简单使用	72
4.2.2 使用菜单制作游戏菜单功能	74
4.3 滚动框	80
4.3.1 使用 ScrollView 显示多页内容	80
4.3.2 监听 ScrollView 的滚动和缩放事件	82
4.3.3 使用 TableView 展示多页内容	83
4.3.4 触摸 TableView 里的菜单来滚动 TableView	86
4.4 扩展控件	87
4.4.1 滑动条控件 ControlSlider	87
4.4.2 开关控件 ControlSwitch	90
4.4.3 取色器控件 ControlColourPicker	92
4.4.4 电位计控件 ControlPotentiometer	94
4.4.5 步进器控件 ControlStepper	95
4.4.6 按钮控件 ControlButton	96
4.4.7 Scale9Sprite	99
4.5 使用编辑框制作用户登录界面	100
第 5 章 Cocos2d-x 动作	105
5.1 动作分类	105
5.2 瞬时动作	106
5.2.1 使用 FlipX/FlipY 实现 X/Y 翻转	106
5.2.2 使用 Hide、Show 实现隐藏和显示	108
5.3 延时动作	109
5.3.1 使用 MoveTo 或者 MoveBy 实现移动	109

5.3.2	使用 RotateTo 和 RotateBy 实现旋转	110
5.3.3	使用 JumpTo 和 JumpBy 实现跳跃	113
5.3.4	使用 ScaleTo 和 ScaleBy 实现缩放	114
5.3.5	使用 SkewTo 和 SkewBy 实现倾斜变形	115
5.3.6	使用 CardinalSplineBy 和 CardinalSplineTo 实现曲线运动	117
5.3.7	使用 FadeIn 和 FadeOut 实现渐隐渐出	119
5.4	联合动作	120
5.4.1	按先后顺序执行动作	120
5.4.2	同时执行动作	121
5.4.3	逆向执行动作	122
5.4.4	多次重复执行动作	123
5.4.5	延时执行动作	124
第 6 章	音频处理	125
6.1	音频处理类 SimpleAudioEngine	125
6.2	添加控制背景音乐	126
6.2.1	播放背景音乐并调整音量	126
6.2.2	停止播放背景音乐	128
6.2.3	暂停播放背景音乐	128
6.2.4	继续播放背景音乐	128
6.3	添加控制音乐效果	129
6.3.1	播放音乐	129
6.3.2	停止播放音乐	131
6.3.3	暂停播放音乐	131
6.3.4	继续播放音乐	131
6.3.5	停止、暂停、继续播放所有音乐	132
6.4	Cocos2d-x 支持的音频格式	133
第 7 章	Cocos2d-x 瓷砖地图	135
7.1	什么是瓷砖地图	135
7.2	使用 Tiled 制作瓷砖地图	137
7.2.1	安装 Tiled	137

7.2.2	制作地图	138
7.3	在游戏中使用瓷砖地图	140
7.3.1	使用 TMXTiledMap 把瓷砖地图加载到游戏中	140
7.3.2	拖曳 TMX 地图	141
7.3.3	在 TMX 地图中添加并移动精灵	142
7.3.4	读写 TMX 地图中的图层和瓷砖	143
第 8 章	Cocos2d-x 中的事件机制	146
8.1	事件和事件调度	147
8.2	触摸事件	149
8.2.1	单点触摸事件的类和方法	149
8.2.2	单击屏幕移动精灵	150
8.2.3	拖动精灵移动	152
8.2.4	修改监听器的优先级	154
8.2.5	多点触摸事件	156
8.2.6	使用多点触摸实现缩放	157
8.3	鼠标事件	159
8.4	键盘事件	159
8.4.1	键盘事件介绍	159
8.4.2	实例：把键盘输入内容显示在屏幕中	160
8.5	加速计	161
8.5.1	加速计介绍	161
8.5.2	实例：利用加速计控制小球移动	162
第 9 章	Cocos2d-x 本地数据存储	165
9.1	使用 UserDefaults 存储数据	165
9.1.1	UserDefaults 介绍	165
9.1.2	使用 UserDefaults 存储修改数据	167
9.2	文件	169
9.2.1	文件处理类 FileUtils	169
9.2.2	判断文件是否存在	169
9.2.3	设置文件别名	170

9.2.4	获取文件完整路径	172
9.2.5	设置文件搜索路径	174
9.2.6	根据分辨率调用不同的资源	175
9.2.7	向文件中写入数据	176
9.2.8	从文件中读取数据	179
9.2.9	把数据写入 plist 文件	180
9.2.10	从 plist 文件读取数据	182
9.3	SQLite 存储	183
9.3.1	SQLite 简介	183
9.3.2	可视化管理工具 SQLiteStudio	185
9.3.3	使用 SQLiteStudio 添加数据库	186
9.3.4	使用 SQLiteStudio 添加表和数据	188
9.3.5	使用 C 语言接口操作 SQLite 数据库	190
9.3.6	不使用回调查询 SQLite 数据库	194
第 10 章	网络编程	197
10.1	HTTP 实现网络通信	198
10.1.1	HTTP 通信简介及常用类	198
10.1.2	GET 方式通信	200
10.1.3	POST 方式通信	203
10.2	Socket 实现网络通信	204
10.2.1	Socket 简介	204
10.2.2	在 Cocos2d-x 中使用 Socket	205
10.3	WebSocket 实现网络通信	209
10.3.1	WebSocket 简介	209
10.3.2	在 Cocos2d-x 中使用 WebSocket	210
第 11 章	物理引擎 Box2D	215
11.1	Box2D 简介	216
11.2	创建 Box2D 的 HelloWorld 项目	217
11.2.1	创建一个世界	217
11.2.2	创建一个地面物体	217

11.2.3	创建一个动态物体	219
11.2.4	模拟 (Box2D 的) 世界	219
11.2.5	清理工作	221
11.3	世界 b2World	221
11.3.1	b2World 简介	221
11.3.2	世界常用功能	222
11.4	物体 b2Body	225
11.4.1	b2Body 简介	225
11.4.2	物体定义	226
11.4.3	创建物体	228
11.4.4	使用物体	229
11.5	固定装置 b2FixtureDef	231
11.5.1	b2FixtureDef 简介	231
11.5.2	创建 b2FixtureDef	231
11.6	关节	234
11.6.1	关节简介	234
11.6.2	关节定义	235
11.6.3	创建关节	235
11.6.4	关节类型和使用关节	236
11.7	接触	242
11.7.1	接触简介	242
11.7.2	接触监听器	243
11.7.3	接触筛选	244
第 12 章	纹理和动画	246
12.1	渲染和修改纹理	246
12.1.1	纹理类 Texture2D	247
12.1.2	Cocos2d-x 支持的纹理格式	249
12.1.3	Cocos2d-x 支持的最大纹理尺寸	249
12.1.4	使用 RenderTexture 保存截屏	250
12.1.5	图片抗锯齿处理方式	251

12.1.6	使用图片缓存	253
12.1.7	制作游戏加载场景	254
12.1.8	使用 TexturePacker 制作 Sprite Sheet	256
12.2	动画	258
12.2.1	帧动画	258
12.2.2	使用帧动画实现英雄打斗	259
12.2.3	Sprite Sheet 动画	266
12.2.4	骨骼动画	268
12.2.5	使用 CocoStudio 制作骨骼动画	269
12.2.6	在项目中调用 CocoStudio 制作的骨骼动画	273
第 13 章	使用 Cocos2d-x 制作 2048 休闲游戏	275
13.1	准备工作	275
13.2	使用 CocoStudio 制作 UI 界面	276
13.3	编写逻辑代码	279
13.3.1	把 UI 界面添加到游戏界面中	279
13.3.2	添加获取分数控件并设置分数	281
13.3.3	添加数字方块类	281
13.3.4	初始化游戏数据	282
13.3.5	添加按钮功能	284
13.3.6	添加事件监听	285
13.3.7	实现方块上下左右移动	287
13.3.8	添加新的数字块	290
13.3.9	判断游戏是否结束	291
13.3.10	添加游戏介绍界面	293
第 14 章	使用 Cocos2d-x 制作水浒卡牌游戏	294
14.1	准备工作	294
14.2	定义游戏数据结构和存储单例	295
14.3	添加登录界面	299
14.4	添加游戏主场景	305
14.5	添加游戏首页	307

14.6	添加英雄卡牌列表界面	311
14.7	添加战斗流程	314
14.8	添加战斗界面	319
14.9	终结	323
附录 A	实例代码清单说明	324
附录 B	Cocos2d-x 3.X 主要版本间的区别	328



第 1 章

准 备

本章介绍 Cocos2d-x 3.0 的优缺点、应用场景、环境搭建，以及它使用的核心概念，包括导演、场景、层、精灵等。

1.1 Cocos2d-x 简介

Cocos2d-x 是一个支持多平台的 2D 手机游戏引擎，使用 C++ 开发，基于 OpenGL ES，采用 cocos2d-iphone 的架构和语法。并且 Cocos2d-x 是一个开源框架，在 MIT 许可证下发布。Cocos2d-x 具有以下多个优点。

1. 免费开源

Cocos2d-x 使用了最宽松的 MIT 开源协议，MIT 许可证之名源自麻省理工学院（Massachusetts Institute of Technology, MIT），又称「X 条款」（X License）或「X11 条款」（X11 License）。

MIT 许可证规定被授权人有权利使用、复制、修改、合并、出版发行、散布、再授权及贩售软体及软体的副本。被授权人可根据程式的需要修改授权条款为适当的内容。在软件和软件的所有副本中都必须包含版权声明和许可声明。此授权条款并非属 copyleft 的自由软体授权条款，允许在自由/开放源码软体或非自由软体（proprietary software）所使用。MIT 条款可与其他授权条款并存。另外，MIT 条款也是自由软体基金会（FSF）所认可的自由软体授权条款，与 GPL 相容。在放心开发的同时，还最大限度地保护开发者的技术投入。由此看出 Cocos2d-x 是一款使用起来非常自由的开源软件。

2. 易学易用的 API 风格

Cocos2d-x 采用 cocos2d-iphone 的架构和语法，属于 cocos2d 家族一员，它提供的 API 与其他家族成员几乎一样，会其他框架的开发人员可以很容易转移到 Cocos2d-x。

Cocos2d-x 周边有很多工具用来提高工作效率，比如 CocosBuilder、Tiled Map Editor。CocosBuilder 是一个可视化的场景编辑器，使用它可以快速地创建一个游戏场景，所见即所得地调整场景中精灵、菜单的位置，避免重复地修改代码，编译运行，再修改再编译，提高效率。

Tiled Map Editor 是一款瓷砖地图编辑器，可以快速地构建一个游戏地图，在塔防等游戏的地图制作中经常用到。

除此之外，Cocos2d-x 提供详细的 API 文档，以及很好的参考书籍，还有专门的交流社区，你可以利用这些资源快速高效地开发游戏。并且这些 API 采用统一的风格，易学易用。下面介绍下 Cocos2d-x API 主要的几个风格。

（1）使用静态方法 `create()` 创建实例对象，例如创建一个精灵对象：

```
Sprite* _sprite = Sprite::create("Monster.png");
```


但有些类通过 `getInstance` 方法获取对象的实例，例如：

```
Size size = Director::getInstance()->getWinSize();
```

(2) 函数名采用 `doSomething()` 的格式，第一个单词为动词，第二个单词为名词，比如 `replaceScene(Scene*)`，`getTexture()`。另外有时也会用到 `doWithResource()` 的格式，比如 `initWithTexture(Texture*)`，`initWithFilename(const char*)`。

(3) 回调函数一般采用“onAction”格式，或者是“onActionConditionTriggers”，例如：

```
class Layer
{
public:
    virtual void onEnter();
    virtual void onExit();
    virtual void onEnterTransitionDidFinish();
}
```

但有些回调函数由于历史问题没有遵循该方式，比如 `TouchBegan`，`TouchMoved` 等。

(4) 使用 `setProperty` 和 `getProperty` 来设置和获取对象的属性。比如设置一个精灵的位置：

```
_sprite->setPosition(Point(0,0))
```

而获取一个精灵的位置要这样写：

```
_sprite->getPosition();
```

3. 稳定可靠，成功案例多

根据开源社区的保守统计，基于 Cocos2d-x 开发的游戏全球范围内已经突破一亿安装量。

很多流行的游戏都是用 Cocos2d-x 开发的，如《捕鱼达人 2》、《龙之力量》、《Warring States》等。网龙、空中网、Haypi、TinyCo、人人游戏、4399、热酷和五分钟等国内外领先的游戏开发公司，都在使用 Cocos2d-x 开发手机游戏。

4. 活跃的社区支持

Cocos2d-x 有一个全职的团队在维护和发展，缺陷可以被及时发现并修复。并且很多大公司如 Zynga、Intel、Google 的工程师也参与到该社区，促进 Cocos2d-x 的发展与完善。表 1-1 展示了 Cocos2d-x 的主要贡献者和维护者。

表 1-1 Cocos2d-x 的主要贡献者和维护者

语 言	平 台	贡献者和维护者	地 址
C++ Famework	iOS	Team-X	中国 厦门
	Android		
	WinXP/7/8		
	WP8	Microsoft	Redmond,us
	BlackBerry	RIM	Waterloo,Canada
	Qt	Nokia	中国 上海
	MeeGo	Intel	German
	Marmalade	Marmalade	SF,US
	WebOS	Bntaniu	Ukraine
	Mac OS X	NetDragon	中国 福州
Lua Binding		NetDragon	中国 福州
		YuLei Yao	中国 成都
Javascript Binding		Zynga	SF,US

5. 跨多种移动平台

Cocos2d-x 本身用 C++开发，具有跨多种移动平台的特性，最大限度减少程序员重复开发的工作，提高效率。表 1-2 展示了目前 Cocos2d-x 对各种平台的支持情况。表 1-2 中，“o”代表 ok，可以放心地正常使用；“i”表示可以使用但已经不被维护、弃用或者尚未整合到标准库中；“w”代表正在开发中。

6. 脚本支持

Cocos2d-x 支持脚本绑定，工程师可以使用 JavaScript、Lua 脚本开发 Cocos2d-x 游戏，将进一步降低开发门槛，广大 JavaScript 工程师也可以方便地使用 Cocos2d-x。

基于 JavaScript 绑定版开发的游戏，未来可以实现平滑过渡至 HTML 5。

表 1-2 Cocos2d-x 跨平台支持情况

平台分类	平 台	C++	Lua	JavaScript
移动平台	iOS	o	o	o
	Android	o	o	o
	WindowsPhone8	o		
	Bada	i		
	BlackBerry	o		
	MeeGo	i		
	Marmalade	o		
桌面平台	win32	o	o	o
	Linux	o	o	
	Win8 Metro	o		
	Mac OS X	o	o	
	Native Client	o	o	

使用 JavaScript 开发游戏比用 C++开发更快更容易，不用考虑指针、引用计数、内存泄露等，并且可以实现真正的跨平台，既能在各种移动平台也能在 Web 上运行。JavaScript Binding（图 1-1）可以把 JavaScript 程序解析成 C，然后编译成本地应用，也可以直接发布成网页游戏。

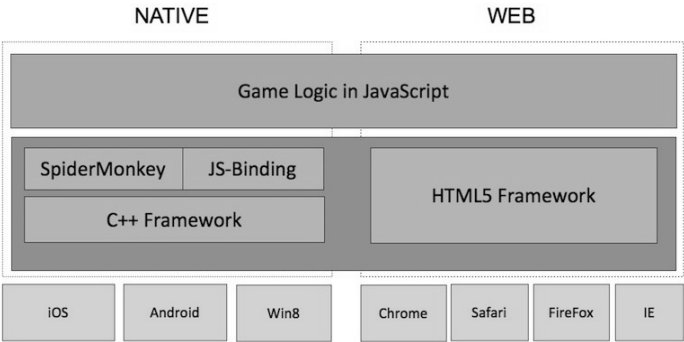


图 1-1 JavaScript Binding 原理图

1.2 Cocos2d-x 架构和目录结构

Cocos2d-x 框架包括图形处理、音频、物理引擎、脚本支持几个模块，图 1-2 详细描述了 Cocos2d-x 的架构。

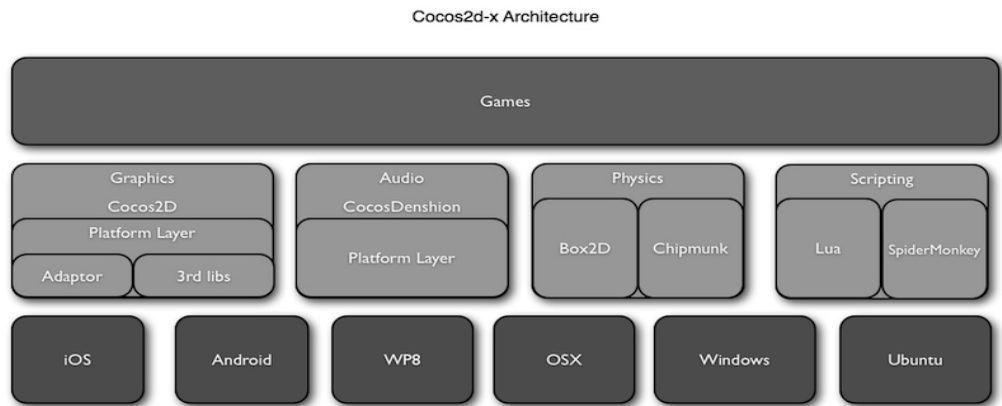


图 1-2 Cocos2d-x 的架构

下载 Cocos2d-x 源码，解压后就可以看到它的目录结构，对它的目录结构的详细介绍可参见表 1-3。

表 1-3 Cocos2d-x 的目录结构

目 录	目录解释
build	该文件夹包含了 Cocos2d-x 3.0 的工程文件，比如 cocos2d_libs.xcodeproj 是 xcode 项目，cocos2d-win32.vc2012.sln 是 Visual Studio 2012 项目
cocos	几乎所有的内容都在该目录中，其中 2d 是 Cocos2d-x 的源码，audio 包含音频支持库，network 包含网络支持库，physics 包含物理引擎库，ui 包含 Cocos2d-x 的 UI 支持，scripting 目录包含了 Lua 官方引擎和 JavaScript 脚本引擎 SpiderMonkey
docs	Cocos2d-x 的文档，可以下载 doxygen，然后用 doxygen 打开 doxygen.config 文件，就可以生成本地 API 文档，不连接网的情况下也能查看

续表

目 录	目录解释
extensions	如果用到图形用户界面（GUI）控件等特性，就需要使用该扩展功能，它的命名空间为 cocos2d::extension
external	该文件夹包含了外部引用库，如物理引擎库 box2d、chipmunk、网络支持 curl、本地数据库 sqlite3 等
licenses	Cocos2d-x 依赖于很多其他开源项目，它们的使用条款都在这里
tests	Cocos2d-x 示例都在这里，其中 cpp-tests 介绍了所有类的使用方法。Lua 和 JavaScript 的示例也在该目录中
templates	在不同 IDE 和平台上新建 Cocos2d-x 项目的模板
tools	这里存放了绑定 C++到 Lua 和 JavaScript 的脚本
setup.py	用来配置 Cocos2d-x 环境变量

1.3 环境搭建

1.3.1 Windows 开发环境搭建

本节介绍如何搭建 Windows 开发环境。系统为 Windows7 32 位，编译工具为 Visual Studio 2012。

1. 下载安装 Visual Studio 2012

到微软官方下载 Visual Studio 2012 的安装程序，地址为 <http://www.microsoft.com/visualstudio/chs/downloads#d-2012-express>。这是一个漫长的过程，下载完成后以管理员权限运行该软件，就能看到安装界面（图 1-3）。



图 1-3 Visual Studio 2012 安装界面

修改安装路径, 选择同意条款, 单击下一步按钮, 在出现的界面中选择全部, 单击安装按钮。接下来就又是漫长的安装过程。

安装好之后运行程序, 第一次启动时会让你设置开发语言, 这里选择 C++ 语言作为开发语言。选择之后 Visual Studio 2012 会用几分钟时间进行环境配置。配置完后就进入开发界面了 (图 1-4)。

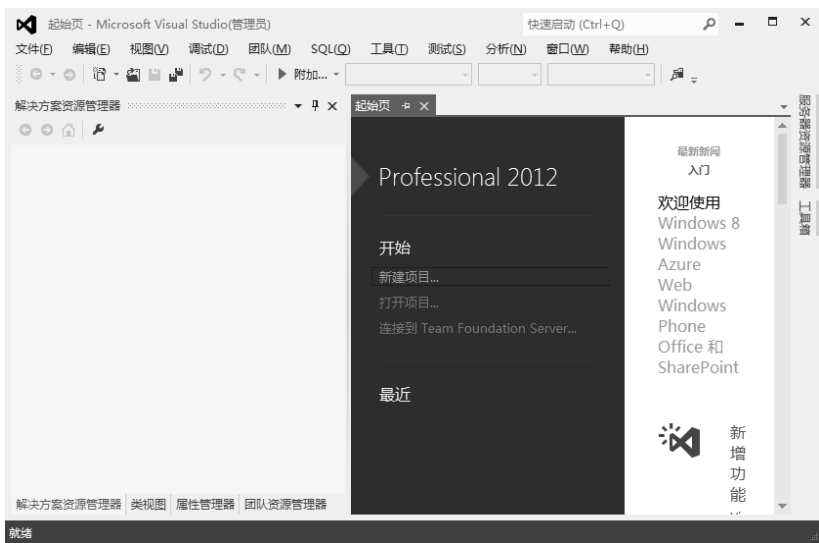


图 1-4 Visual Studio 2012 界面

2. 下载安装最新版本 Cocos2d-x

到 Cocos2d-x 官方网址下载 Cocos2d-x 最新版，地址是 <http://www.Cocos2d-x.org/projects/Cocos2d-x/wiki/Download>。笔者下载的是 3.0 版本。解压下载的文件 `cocos2d-x-3.0.zip` 到合适的位置（可以是任何地方，当然最好是让你的文件分类整洁一些）。

进入文件夹 `cocos2d-x-3.0`，双击打开工程项目 `cocos2d-win32.vc2012.sln`，就可以用 Visual Studio 2012 打开 Cocos2d-x 项目了（图 1-5）。

3. 运行 cpp-tests

在 `cpp-tessts` 项目上单击右键，选择“设为启动项目”，按 `Ctrl+F5` 组合键运行项目，第一次运行因为要链接、编译所有项目，所以会有点慢。当你在输出窗口看到“生成：成功 6 个，失败 0 个，最新 0 个，跳过 0 个”时，说明编译就成功了，这时就能看到 Cocos2d-x 提供的示例程序了（图 1-6）。

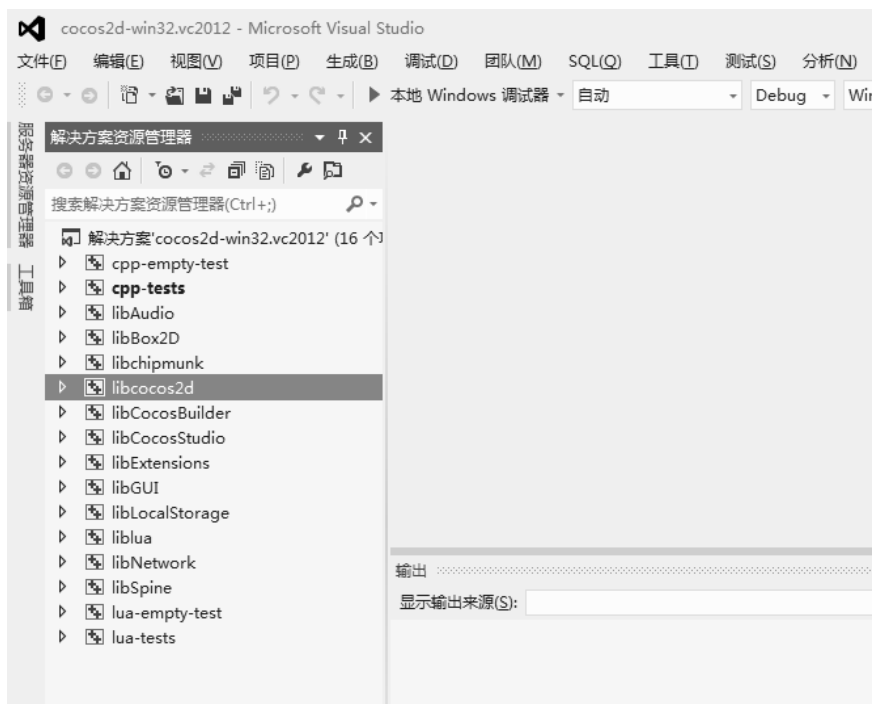


图 1-5 Cocos2d-x 项目

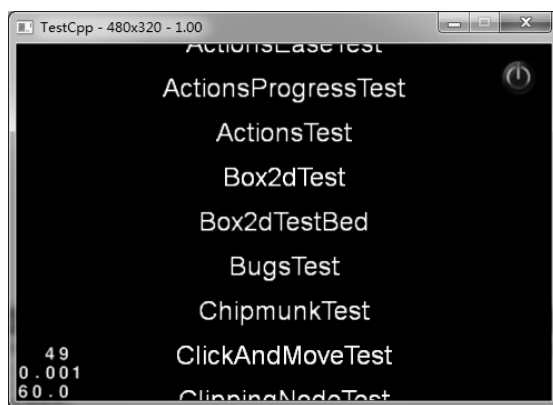


图 1-6 Cocos2d-x 示例程序演示

1.3.2 Mac 开发环境搭建

本节讲解 Mac 下开发环境的搭建，操作系统需为 Mac OS X 10.8+，编译工具为 Xcode 4.6.2+。

(1) 下载安装 Xcode。

(2) 下载 Cocos2d-x 并解压，进入 build 目录，双击 cocos2d_tests.xcodeproj，使用 Xcode 打开项目。

(3) 在下拉列表中选择“cpp-tests iOS”（图 1-7）。

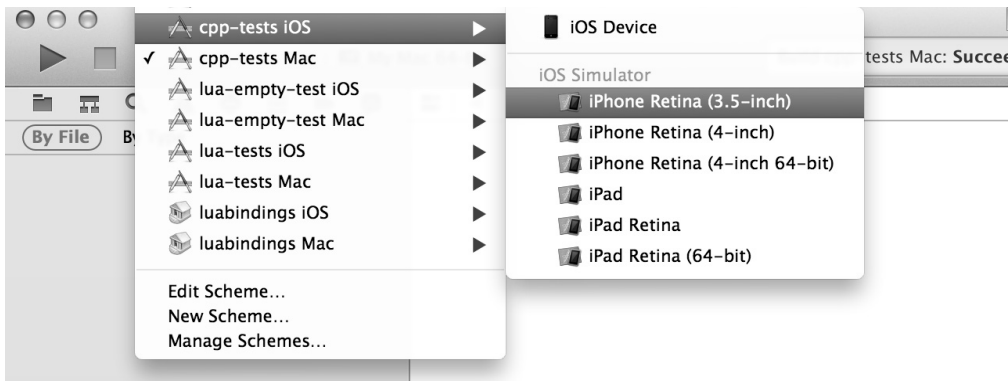


图 1-7 下拉列表

(4) 按组合键 Cmd+R，运行程序，就能在 iOS 的模拟器中看到 cpp-tests（图 1-8）。

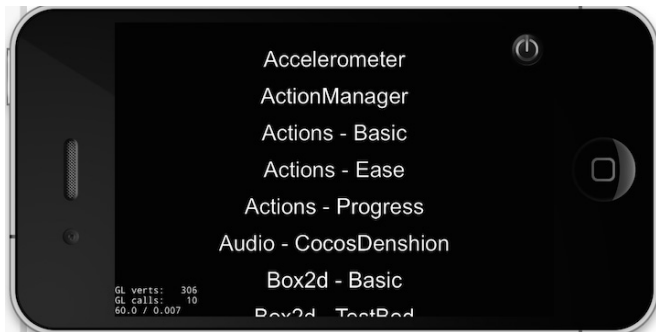


图 1-8 iOS 模拟器

1.3.3 创建新项目

本节介绍如何在 Windows 7 平台使用 cocos console 创建一个新项目。Mac 和 Linux 系统平台类似，主要是把运行命令的“命令提示符”换成“终端”等。

1. 安装 Python

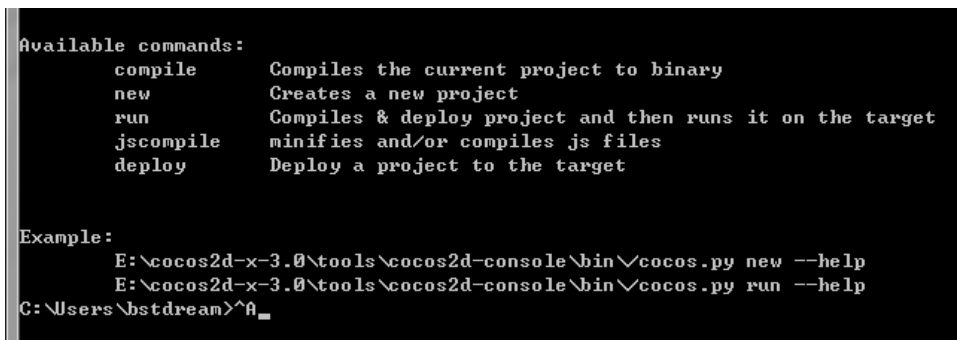
Cocos2d-x 使用 Python 编写的脚步文件创建新项目，所有需要安装 Python。到 Python 官网（<https://www.python.org/>）下载 Python 并安装，然后在环境变量里设置 path，添加 Python 所在目录。

2. 配置环境

打开命令提示符，运行 Cocos2d-x 3.0 根目录下的 setup.py:

```
python setup.py
```

该脚本配置 cocos console 路径和 Android 平台，本节直接敲回车跳过 Android 配置。然后重启命令提示符，输入 cocos，看到如图 1-9 所示内容，说明配置成功。



```
Available commands:
  compile      Compiles the current project to binary
  new          Creates a new project
  run          Compiles & deploy project and then runs it on the target
  jscompile    minifies and/or compiles js files
  deploy       Deploy a project to the target

Example:
  E:\cocos2d-x-3.0\tools\cocos2d-console\bin\cocos.py new --help
  E:\cocos2d-x-3.0\tools\cocos2d-console\bin\cocos.py run --help
C:\Users\bstdream>^H_
```

图 1-9 配置成功

3. 配置环境

在控制台输入下面的命令，

```
cocos new HelloWorld -p com.Hello.HelloWorld -l cpp -d E:\
```

new 命令中各参数解释如下。

- ◎ HelloWorld: 是项目名称，替换成自己的项目名称。
- ◎ -p com.Hello.game: android 项目的包名。
- ◎ -l cpp: 使用的开发语言，可以是 cpp、lua、js。
- ◎ -d E:\: 放项目的路径。

创建成功后就能看到如图 1-10 所示的文件夹结果。

名称	修改日期	类型
Classes	2014/4/25 19:40	文件夹
cocos2d	2014/5/13 0:05	文件夹
proj.android	2014/4/25 19:40	文件夹
proj.ios_mac	2014/5/13 0:05	文件夹
proj.linux	2014/4/25 19:40	文件夹
proj.win32	2014/5/13 0:05	文件夹
proj.wp8-xaml	2014/4/25 19:40	文件夹
Resources	2014/4/25 19:40	文件夹
.cocos-project.json	2014/5/13 0:05	JSON 文件
CMakeLists.txt	2014/5/13 0:05	文本文档

图 1-10 文件夹结果

4. 运行程序

运行程序有两种方式，一种是使用 Visual Studio 2012 打开 proj.win32 目录下的工程项目，按 F5 键运行程序（图 1-11）。

第二种方式是使用 cocos console。在命令行中输入命令：

```
cocos run -s E:\HelloWorld -p win32
```

在 run 命令中各选项含义如下。

- ◎ -s: 指定项目所在文件路径，可以是绝对路径或相对路径。
- ◎ -p: 指定在什么平台上运行程序，可以是 iOS、Android、Win32、Mac 和 Linux。

可以使用 `cocos run --help` 命令查看 `run` 的详细使用方式。

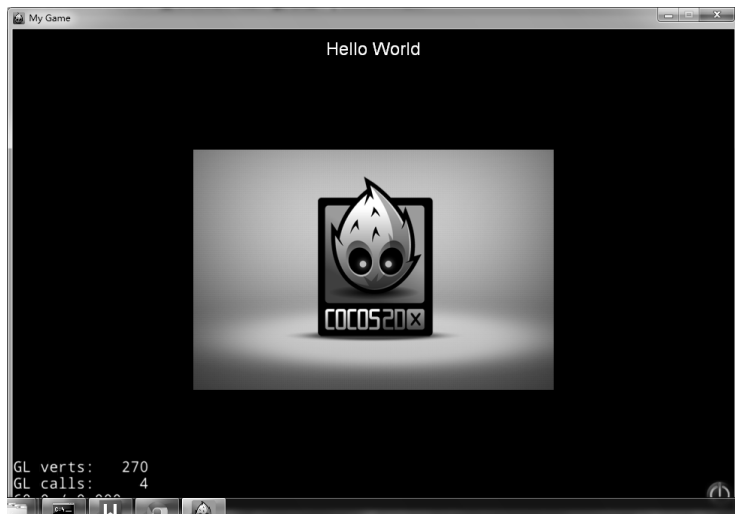


图 1-11 运行程序

1.3.4 在 Android 上调试项目

本节讲解如何在 Android 上运行调试项目，平台是 Win32，Mac 和 Linux 具有类似操作。

1. 安装 JDK

打开命令提示符，输入 `java -version`，如果出现如图 1-12 所示内容，说明已经安装 JDK。

```
java version "1.7.0_21"  
Java(TM) SE Runtime Environment (build 1.7.0_21-b11)  
Java HotSpot(TM) Client VM (build 23.21-b01, mixed mode, sharing)
```

图 1-12 已经安装 JDK

如果是其他错误提示，需要下载安装 JDK，网址是 <http://www.oracle.com/technetwork/java/javase/downloads/index.html>。

2. 下载 Android SDK

Google 官方提供了开发 Android 应用程序的集成式 IDE “ADT Bundle”。ADT Bundle 集成了所有开发应用的工具和配置，解决了大部分新手通过 Eclipse 来配置 Android 开发环境的复杂问题。它包括这些东西：

- ◎ 一个集成了 ADT 的 Eclipse。
- ◎ Android SDK 管理工具。
- ◎ Android 平台工具。
- ◎ 最新的 SDK。
- ◎ Android 模拟器。

到 <http://developer.android.com/sdk/index.html> 下载相应的版本，本书使用的是 Win32 32 位版本。下载完成后，直接解压压缩包，可以看到两个文件夹 *sdk* 和 *eclipse*。

打开 Eclipse，单击工具栏里的 Android SDK Manager，可以下载需要的 Android SDK（图 1-13）。

3. 下载 NDK

NDK 提供了一系列的工具，帮助开发者快速开发 C（或 C++）的动态库，并能自动将 so 和 Java 应用一起打包成 APK。另外，NDK 也集成了交叉编译器，并提供了相应的 mk 文件隔离平台、CPU、API 等差异，开发人员只需要简单修改 mk 文件（指出“哪些文件需要编译”、“编译特性要求”等），就可以创建出 so。

到 <https://developer.android.com/tools/sdk/ndk/index.html> 下载对应的 NDK，本书使用的是 Windows 32-bit 版，下载完成后解压即可。

4. 下载 Ant

Apache Ant 是一个将软件编译、测试、部署等步骤联系在一起加以自动化的一个工具，常用于 Java 环境中的软件开发。由 Apache 软件基金会所提供。可以到 <http://ant.apache.org/bindownload.cgi> 下载，下载完成后直接解压即可。

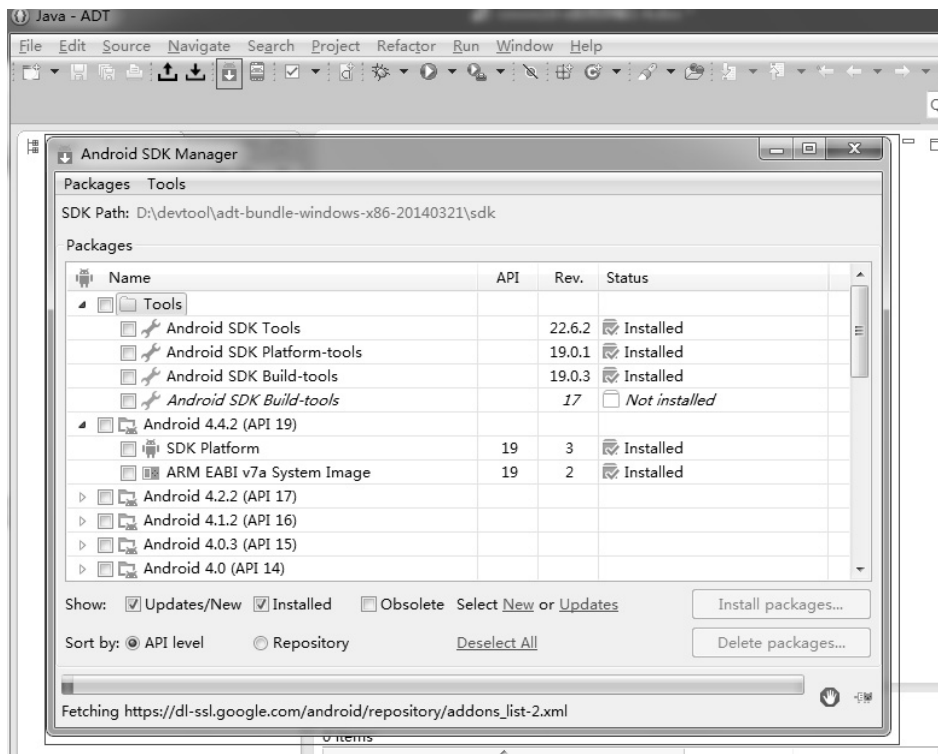


图 1-13 下载需要的 SPK

5. 配置环境

完成上面的步骤后，在“命令提示符”中进入 Cocos2d-x 目录，运行 python setup.py，看到如图 1-14 所示界面。

根据图中所示，NDK_ROOT、ANDROID_SDK_ROOT 已经配置好，但 ANT_ROOT 还没有，把上面 Ant 的解压路径输入命令提示符中、按回车。

6. 安装 Cygwin

由于编译 Cocos2d-x 项目要在 Linux 环境下，所以要在 Windows 平台下安装 Cygwin。Cygwin 是一个在 Windows 平台上运行的类 UNIX 模拟环境。

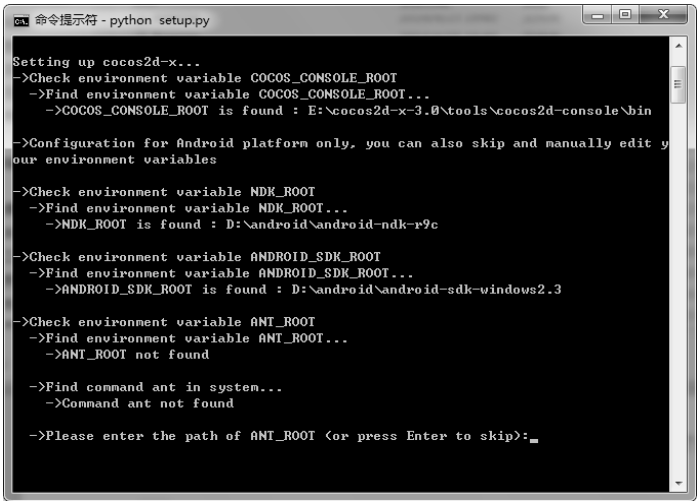


图 1-14 运行 python setup.py

首先到 Cygwin 官网 (<http://www.cygwin.com/>) 下载安装程序，运行 setup.exe 文件。第一次安装时，需要联网下载所需要的安装文件，下载的过程有点漫长，所以最好把下载的文件离线保存到本地，再次安装时，可以直接指定从本地保存的文件安装。

本书需要的文件有 autoconf2.1、automake1.10、binutils、gcc-core、gcc-g++、gcc4-core、gcc4-g++、gdb、pcrc、pcrc-devel、gawk、make。务必选中 dev 下的 make 文件，如果嫌麻烦就全装，将默认改成安装（图 1-15）。

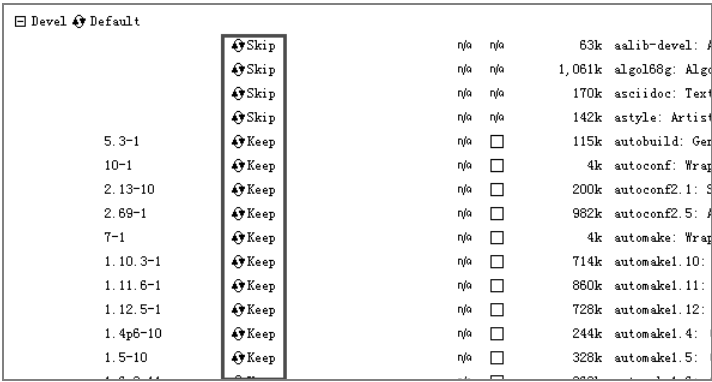


图 1-15 需要安装的文件

`skip` 是不安装的意思，`keep` 是保持也就是安装过的意思，如果你想安装，就单击一下 `skip`，让它变成版本号就是安装了。

安装完成以后，在命令行中进入 `cygwin` 目录，并执行 `cygwin.bat`，如果不是用 `Administrator` 账号登录的系统，那么会在 `cygwin\home\` 文件夹中生成一个以登录名命名的新文件夹。

修改新生成文件夹中的 “.bash_profile” 文件，用 `sublime` 等文本编辑器打开，在最后增加 `NDK` 的安装路径。

```
NDK_ROOT=/cygdrive/d/android/android-ndk-r9c
export NDK_ROOT
```

然后保存关闭。

最后添加系统变量 `Path`，`C:\cygwin\bin`。

7. 编译生成

完成上面的步骤后，就可以编译生成项目了，打开 `Cygwin Terminal`，进入上面创建的项目的 `Android` 工程目录中：

```
cd /cygdrive/e/HelloWorld/proj.android
```

然后运行 `build_native.py` 项目：

```
python build_native.py
```

接下来就是等待编译，编译结束后，如果看到如图 1-16 所示结果，表示已经编译成功。倒数第二行表示已经成功地生成了 `.so` 文件。

8. 在模拟器中运行

打开安装好的 `ADT Bundle`，依次选择菜单中的 `File→New→Project`。在弹出的对话框中找到 `Android/Android Project from Existing Code`（图 1-17）。

单击 `Next` 按钮，在弹出的对话框中选择创建的 `HelloWorld` 项目（图 1-18）。

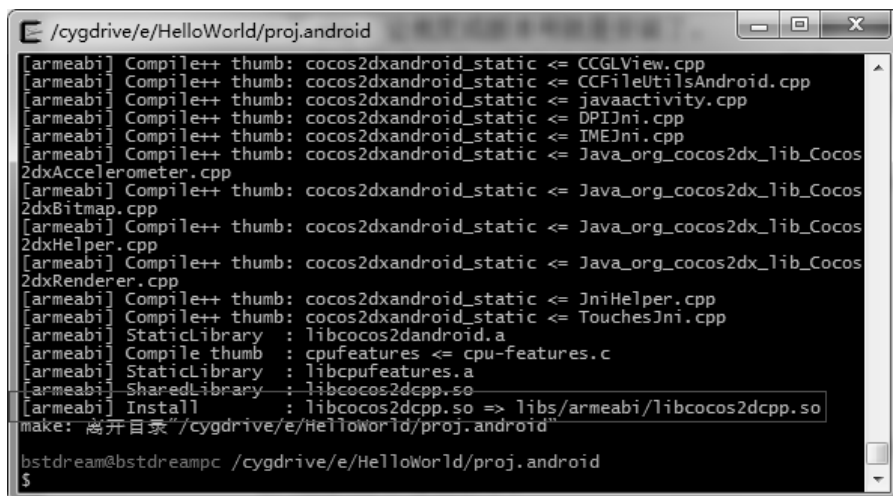


图 1-16 编译成功

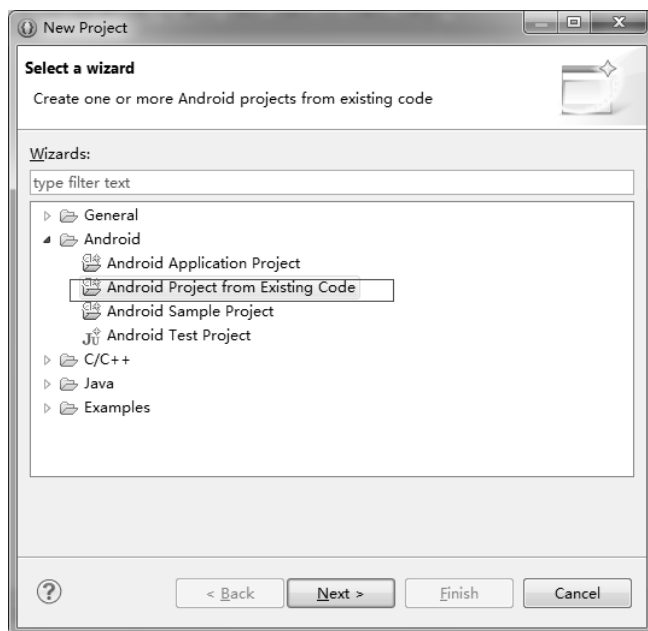


图 1-17 新建项目

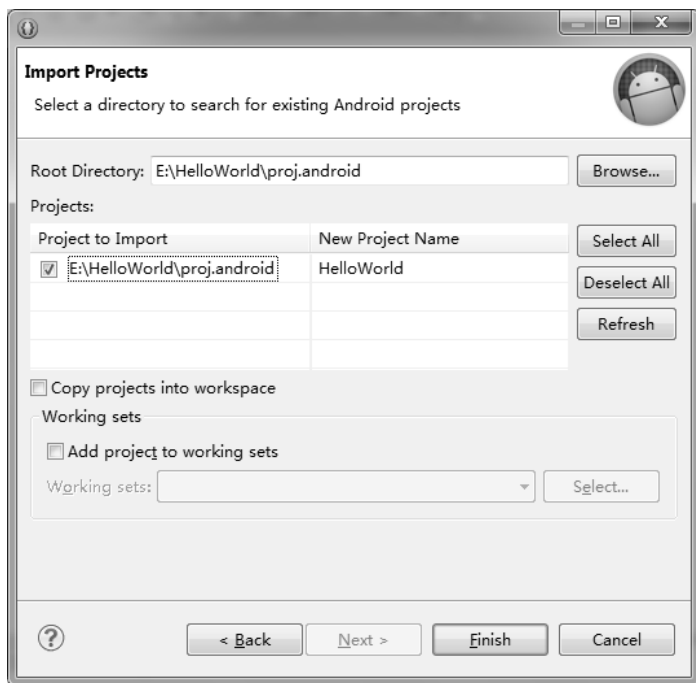


图 1-18 选择项目

单击 Finish 按钮即可导入项目。这时 ADT 会提示错误，这是因为缺少 cocos2d 库。找到 cocos2dx\platform\android\java\src 文件夹，把里面的 org 文件夹整个复制到项目中。然后刷新项目，错误提示就会消失。

在运行程序前，需要先创建一个虚拟机，有两种方式创建虚拟机（图 1-19）。

在弹出的窗口中单击右侧的 New，弹出创建对话框，选择后输入相应的选项（图 1-20）。然后单击 OK 按钮，成功创建虚拟机。

在项目上右击，选择“Run As→Android Application”，等待模拟器启动，成功后就可以看到经典的 HelloWorld 了。

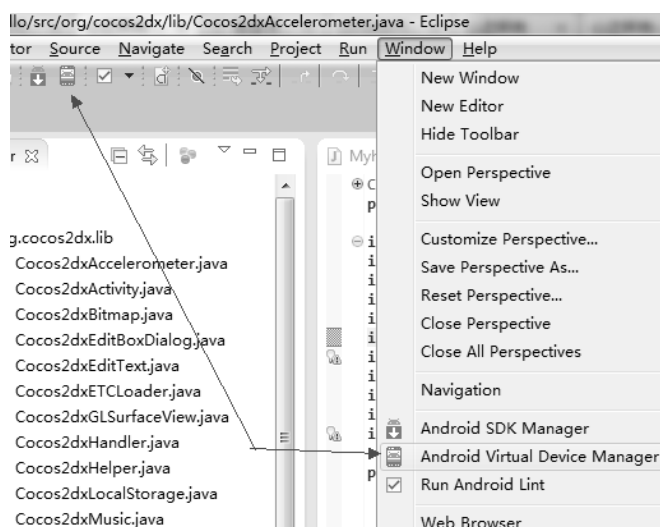


图 1-19 创建虚拟机

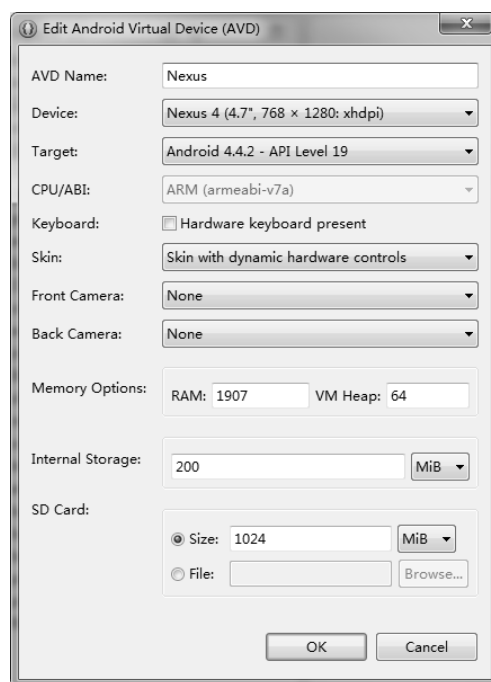


图 1-20 输入项目

1.3.5 打包 APK 文件

在模拟器或真机上运行的主要目的是进行调试，上传到应用商店时都要打包成 APK 文件，本节就讲解下如何打包 Android 安装包。

(1) 选择菜单栏中的“File→Export”，在弹出的对话框中选择“Android/Export Android Application”（图 1-21）。

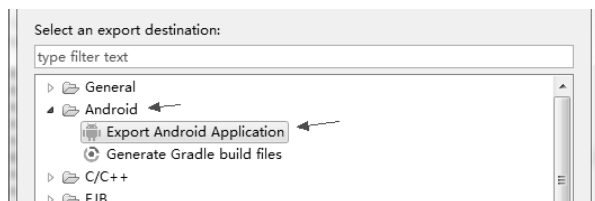


图 1-21 选择打包目标

(2) 双击后，在弹出窗口中输入游戏名称，单击 Next 按钮。接下来需要填入认证文件和密码。如果是第一次，需要选择“Create New keystore”，在弹出的对话框中就是输入认证文件的信息（图 1-22）。



图 1-22 认证文件信息

(3) 选择保存路径，然后单击 Finish 按钮即可生成 Android 安装文件。

2

第 2 章

Cocos2d-x 常用数据和方法

本章介绍 Cocos2d-x 经常用到的数据类型、方法、宏定义，了解了这些，可以更加快速地学习 Cocos2d-x，减少代码量，提高工作效率。

2.1 C++数据类型

由于 Cocos2d-x 是用 C++开发的，所以 C++中的类型都可以在 Cocos2d-x 中使用。C++中主要的类型有：

- ◎ 整型 int，比如 3、5、10。
- ◎ 浮点型 float、double，比如 122.342、1.000000。

- ◎ 字符型 `char`，比如 ‘a’。
- ◎ 字符串 `string`，比如 “string”。
- ◎ 布尔型 (`bool`)，即 `true` 和 `false`。
- ◎ 数组，比如包含三个整数的数组，`int intArr[3]`。

除此之外，指针在 Cocos2d-x 中也随处可见。

2.2 Cocos2d-x 封装的数据类型

虽然有 C++ 的数据类型可以使用，但这些数据的功能不够用，或者不符合 Cocos2d-x 的使用方式，所以它自己提供了一些封装的数据类型。本节示例完整代码请参考代码清单 2-2。

2.2.1 布尔型 Bool 的使用

`Bool` 是对布尔型 (`bool`) 的封装。它使用 `create` 创建对象，使用 `getValue` 获取 `bool` 值。下面的示例先创建一个 `Bool` 对象，再判断值的真假，输出相应内容。

```
Bool *b1 = Bool::create(true);
if(b1->getValue()){
    log("bool = true");
}else
{
    log("bool = true ");
}
```

由于 `b1` 的值为 `true`，所以输出 `bool = true`。注意，想要在项目输出窗口看到输出，需要以调试（按 F5 键）模式运行程序，而不是直接运行（按 Ctrl+F5 组合键）。

2.2.2 整型 Integer 的使用

Integer 是对整型数的封装，它类似 Bool，使用 create 创建对象，使用 getValue 获取值。

```
Integer *myint = Integer::create(20);
log("myint = %d",myint->getValue());
```

调试程序，在输出窗口会看到 myint = 20。

2.2.3 浮点型 Double、Float 的使用

Float 和 Double 是对浮点数的封装，它使用 create 方法创建对象，使用 getValue 方法获取值。

```
Float *myFloat = Float::create(1.0);
Double *myDouble = Double::create(2.0);
log("myFloat = %f;\n myDouble = %f",myFloat->getValue(),myDouble->getValue());
```

调试程序，在输出窗口会看到如下结果：

```
Cocos2d: myFloat = 1.000000;
myDouble = 2.000000
```

2.2.4 字符串 String 的使用

字符串是程序开发中经常需要处理的数据类型，并且字符串的处理比上面几种数据类型都要复杂。Cocos2d-x 中使用 String 对字符串进行处理。

创建 String 字符串的方法：可以由 C++ 标准库里的字符串创建，可以从文件中获取，可以像 printf 函数一样创建格式化的字符串。String 还能转成其他类型的数据，比如，通过 getCString 方法转化成 C 语言中的字符串，通过 boolValue 方法转化成 bool 型数据，通过 intValue、floatValue、doubleValue 转化成整型、单精度浮点型、双精度浮点型。下面的示例展示了 String 的常用方法。

```

//由C++标准库里的字符串创建 String
String *str1 = String::create("Cocos2d-x");
//创建格式化的字符串
String *str2 = String::createWithFormat("I love %s",str1->
getCString());
//打印出两个字符串, 并进行比较
if(str1->isEqual(str2)){
    log("字符串 %s 与字符串 %s 相同",str1->getCString(),str2->
getCString());
}else{
    log("字符串 %s 与字符串 %s 不相同",str1->getCString(),str2->
getCString());
}

```

运行程序看到如下输出结果:

```
Cocos2d: 字符串 Cocos2d-x 与字符串 I love Cocos2d-x 不相同
```

2.2.5 数组 Array 的使用

Array 是对数组的封装, C++标准数组就是相同类型数据的有序组合。例如 `int a[10]` 就定义了一个包含有 10 个整数的整型数组 `a`。但是 Array 中的元素是 Object, 可以存放不同类型的数据。Array 同时封装了一些常用方法以便更高效地使用数组。这些方法有创建数组, 查找、添加、删除、替换数组元素, 比较数组等。

下面这段程序演示了常用的对数组操作的方式。

```

//你可以直接创建一个不含任何元素的数组
Array *heroArr= Array::create();
//给该数组添加数组元素
heroArr->addObject(String::create("撼地神牛")); //添加英雄名称
heroArr->addObject(Integer::create(10)); //添加英雄等级
heroArr->addObject(Integer::create(2000)); //添加血量
heroArr->addObject(Float::create(200.5)); //添加攻击力
//打印出英雄的名称, 等级, 血量, 攻击力。
log("\n 英雄名是:%s,\n 等级是:%d,\n 血量是:%d,\n 攻击力是:%f\n",
    ((String *)heroArr->objectAtIndex(0))->getCString(),
    ((Integer *)heroArr->objectAtIndex(1))->getValue(),

```



```
((Integer *)heroArr->objectAtIndex(2))->getValue(),
((Float *)heroArr->objectAtIndex(3))->getValue());
```

输出结果为：

```
英雄名是:撼地神牛,
等级是:10,
血量是:2000,
攻击力是:200.500000
```

2.2.6 点 Point 的使用

Point 用来表示一个点，可以通过 Point (float x, float y) 快速创建一个点对象（注意 p 宏已经被放弃使用）。

Point 类也提供了很多方法，方便我们实现与点相关的一些操作。比如计算经过原点和该点的向量与 x 轴正方向或另一个向量的夹角；计算两个点的距离；计算两个点的中点。并且 Point 重写了+、-、*、/、=操作符，可直接对点进行加、减、乘、除、赋值。接下来就演示 Point 的常用使用方式。

```
//生成两个点
Point point1 = Point(10,10);
Point point2 = Point(60,60);
Point point3;
//点1与x轴的夹角
log("\n 点 1(%f,%f) 与 x 轴的夹角为:%f",point1.x,point1.y,point1.
getAngle());
//两个点的夹角
log("\n 点 1(%f,%f) 与点 2(%f,%f) 的夹角为:%f"
,point1.x,point1.y,point2.x,point2.y,point1.getAngle(point2));
//两个点的距离
log("\n 点 1(%f,%f) 与点 2(%f,%f) 的距离为:%f",
point1.x,point1.y,point2.x,point2.y,point1.getDistance(point2));
//两个点相加
point3 = point1 + point2;
log("\n(%f,%f)+(%f,%f)=(%f,%f)",point1.x,point1.y,point2.x,point2.
y,point3.x,point3.y);
//两个点相减
```

```

    point3 = point1-point2;
    log("\n(%f,%f)-(%f,%f)=(%f,%f)",point1.x,point1.y,point2.x,point2.y,point3.x,point3.y);
    //除法
    point3 = point1/2;
    log("\n(%f,%f)/2=(%f,%f)",point1.x,point1.y,point3.x,point3.y);
    //乘法
    point3 = point1*2;
    log("\n(%f,%f)+(%f,%f)=(%f,%f)",point1.x,point1.y,point2.x,point2.y, point3.x,point3.y);

```

输出结果为：

```

点 1 (10.000000,10.000000) 与 x 轴的夹角为:0.785398
Cocos2d:
点 1 (10.000000,10.000000) 与点 2 (60.000000,60.000000) 的夹角为:0.000000
Cocos2d:
点 1 (10.000000,10.000000) 与点 2 (60.000000,60.000000) 的距离为:70.710678
Cocos2d:
(10.000000,10.000000)+(60.000000,60.000000)=(70.000000,70.000000)
Cocos2d:
(10.000000,10.000000)-(60.000000,60.000000)=(-50.000000,-50.000000)
Cocos2d:
(10.000000,10.000000)/2=(5.000000,5.000000)
Cocos2d:
(10.000000,10.000000)+(60.000000,60.000000)=(20.000000,20.000000)

```

2.2.7 尺寸 Size 的使用

Size 用来表示物体尺寸的大小，它具有两个参数，既高度和宽度。Size 对象也能进行加、减、乘、除等数学运算。使用 Size 方法创建一个 Size 对象，使用点运算加 width、height 获取 Size 的宽度和高度。最常用的地方就是获取游戏窗口屏幕的大小：

```
Size visibleSize = Director::getInstance()->getVisibleSize();
```

下面展示如何使用 Size。

```
//使用 SizeMake 创建两个 Size
```

```

Size size1 = Size(10, 20);
Size size2 = Size(50, 60);
Size size3;
//两个Size相加
size3 = size1 + size2;
log("size(%f,%f)+size(%f,%f)=size(%f,%f)",size1.width,size1.
height, size2.width,size2.height,size3.width,size3.height);
//两个Size相减
size3 = size1 - size2;
log("size(%f,%f) - size(%f,%f)=size(%f,%f)",size1.width,size1.
height, size2.width,size2.height,size3.width,size3.height);
//Size乘法
size3 = size1*10;
log("size(%f,%f)*10=size(%f,%f)",size1.width,size1.height,
size3.width, size3.height);
//Size除法
size3 = size1/10;
log("size(%f,%f)/10=size(%f,%f)",size1.width,size1.height,
size3.width, size3.height);

```

输出结果为：

```

Cocos2d:      size(10.000000,20.000000)+size(50.000000,60.000000)=size
(60.000000, 80.000000)
Cocos2d: size(10.000000,20.000000) - size(50.000000,60.000000)=size
(-40.000000,-40.000000)
Cocos2d: size(10.000000,20.000000)*10=size(100.000000,200.000000)
Cocos2d: size(10.000000,20.000000)/10=size(1.000000,2.000000)

```

2.2.8 矩形 Rect 的使用

Rect 与 Size 有很多相似之处。它表示在屏幕某一位置的矩形区域。所以 Rect 除了有宽度和高度外，还有位置的概念。一般使用 Rect 创建一个 Rect 对象（注意 CCRectMake 已经废弃不用）。Rect 需要 4 个参数，左下角的 x 坐标、 y 坐标，宽度和高度。

Rect 提供了一些常用方法用来获取有关 Rect 的信息。如可以使用 containsPoint

(const Point & point) 判断某个点是否在 Rect 定义的矩形区域内。也可以获得矩形 4 个定点和中点的坐标。下面展示如何用代码实现。

```
//Rect 测试
//生成一个坐标为 10, 20, 宽为 50, 高为 30 的矩形区域
Rect rect = Rect(10, 20, 50, 30);
//生成两个点
Point point1 = Point(15,25);
Point point2 = Point(100,100);
if (rect.containsPoint(point1)) {
    log("rect 包含点 point1\n");
}else{
    log("rect 不包含点 point1\n");
}
if (rect.containsPoint(point2)) {
    log("rect 包含点 point2\n");
}else{
    log("rect 不包含点 point2\n");
}
//获取 rect 矩形区域最左、右、上、下、中间的坐标点
float maxX = rect.getMaxX();
float minX = rect.getMinX();
float maxY = rect.getMaxY();
float minY = rect.getMinY();
float midX = rect.getMidX();
float midY = rect.getMidY();
log("rect 的左下角坐标为 (%f,%f)\n 左上角坐标为 (%f,%f)\n 右下角坐标为 (%f,%f)\n 右上角坐标为 (%f,%f)\n 中点坐标为 (%f,%f)\n",minX,minY,minX,maxY,maxX,minY,maxX,maxY,midX,midY);
```

运行程序，得到结果为：

```
rect 包含点 point1
rect 不包含点 point2
rect 的左下角坐标为 (10.000000,20.000000)
左上角坐标为 (10.000000,50.000000)
右下角坐标为 (60.000000,20.000000)
右上角坐标为 (60.000000,50.000000)
中点坐标为 (35.000000,35.000000)
```

2.2.9 字典 Dictionary 的使用

Dictionary 是不同数据的无序组合，利用哈希表算法来进行 Object 管理，其中的元素是通过哈希表进行存储的，哈希表算法放在 uthash.h 中。

Dictionary 通过键值对管理数据，每一个键（即 Key）对应一个值（即 Value），且 Key 是唯一不重复的。字典中元素主要也是通过键（Key）查找、添加、修改、删除对应的值。并且，只有 Object 及其子类的指针才能添加到 Dictionary 中。下面举例说明它的用法。

```
//创建一个字典，该字典会自动释放
Dictionary* pDict = Dictionary::create();
//插入对象到字典中
String* pValue1 = String::create("100");
String* pValue2 = String::create("120");
Integer* pValue3 = Integer::create(200);
pDict->setObject(pValue1, "key1");
pDict->setObject(pValue2, "key2");
pDict->setObject(pValue3, "key3");
//删除 key2 键值对
pDict->removeObjectForKey("key2");
//根据键 key 获取相应的值
String* pStr1 = (String*)pDict->objectForKey("key1");
log("{ key1: %s }", pStr1->getCString());
Integer* pInteger = (Integer*)pDict->objectForKey("key3");
log("{ key3: %d }", pInteger->getValue());
```

输出结果为：

```
{ key1: 100 }
{ key3: 200 }
```

注意：

当使用 setObject 为字典对象添加元素时，如果字典未包含该键值对，就把它添加到字典中，如果已经包含该键值对，它就会把原来的元素从字典中删除，然后把新的元素添加到字典中。

当使用 `objectForKey` 获取值时，要进行强制类型转换。如果事先不知道该对象的类型，可以使用 `dynamic_cast<SomeType*>` 对值进行检查：

```
String*    pStr2    =    dynamic_cast<String*>(pDict->objectForKey
("key1"));
if(pStr2 != NULL) {
    // 对 pStr2 进行操作
}
```

2.3 常用宏定义

宏定义是 C 提供的三种预处理功能的一种，另外两种是文件包含、条件编译。C 语言使用 `#define` 定义一个宏，它用来将一个标识符定义为一个字符串，该标识符被称为宏名，被定义的字符串称为替换文本。宏定义可以减少需要编写的代码量，也能方便模块化管理，所以 Cocos2d-x 也把一些常用的方法定义成宏，便于提高开发效率。本节示例完整代码请参考代码清单 2-3。

2.3.1 数学相关宏的使用

Cocos2d-x 提供了一些与数学相关的宏定义，例如生成随机数、角度转换等，下面示例列举几个常用的宏定义。

```
//CCRANDOM_MINUS1_1 返回一个-1 到 1 之间的浮点数
log("CCRANDOM_MINUS1_1()=%f",CCRANDOM_MINUS1_1());

//CCRANDOM_0_1 返回一个 0 到 1 之间的浮点数
log("CCRANDOM_0_1()=%f",CCRANDOM_0_1());

//CC_DEGREES_TO_RADIANS 角度转弧度
log("CC_DEGREES_TO_RADIANS(30)=%f",CC_DEGREES_TO_RADIANS(30));
//CC_RADIANS_TO_DEGREES 弧度转角度
log("CC_RADIANS_TO_DEGREES(180)=%f",CC_RADIANS_TO_DEGREES(180));
//CC_SWAP(x, y, type) 交换两个值
int x = 10;
int y = 20;
```

```
log("交换前 x=%d,y=%d",x,y);
CC_SWAP(x, y, int);
log("交换后 x=%d,y=%d",x,y);
```

运行程序，得到的结果是：

```
Cocos2d: CCRANDOM_MINUS1_1()=-0.999984
Cocos2d: CCRANDOM_0_1()=0.131538
Cocos2d: CC_DEGREES_TO_RADIANS(30)=0.523599
Cocos2d: CC_RADIANS_TO_DEGREES(180)=10313.240234
Cocos2d: 交换前 x=10,y=20,
Cocos2d: 交换后 x=20,y=10
```

2.3.2 断言宏 CCAsset 的使用

CCAssert 是语句中断宏，它的声明在 ccMacros.h 中，结构如下：

```
#define CCASSERT(cond,
    msg
)
#define CCAssert    CCASSERT
```

其中，cond 是断言表达式，msg 是错误提示信息。当 cond 为真时，接着运行下面的代码；如果不为真，则显示字符串 msg 信息，并且程序在该点中断，类似在该点设置断点。例如，我们可以在一个函数里加上 CCAsset 来判断需要的变量是否符合要求，如果不符合要求就终止程序并显示出错的地方。我们在程序中添加如下代码，运行后就会出现错误提示。

```
Point *point=NULL;
CCAssert(point!=NULL,"sdfasdf");
```

2.3.3 数组遍历宏 CCARRAY_FOREACH 和 CCARRAY_FOREACH_REVERSE 的使用

这两个宏用来遍历数组，其中 CCARRAY_FOREACH 为正向遍历，CCARRAY_FOREACH_REVERSE 为逆向遍历。CCARRAY_FOREACH 和 CCARRAY_FOREACH_REVERSE 的声明在 CCArray.h 中，结构如下：

```
#define CCARRAY_FOREACH ( __array__,  
    __object__  
)  
#define CCARRAY_FOREACH_REVERSE ( __array__,  
    __object__  
)
```

其中__array__为需要遍历的数组，__object__为 Object 对象，在程序里要对该 Object 进行强制类型转换，使用方法如下所示：

```
//先创建一个 1 到 10 的整型数组  
Array *eachArr=Array::create();  
eachArr->addObject(Integer::create(1));  
eachArr->addObject(Integer::create(2));  
eachArr->addObject(Integer::create(3));  
eachArr->addObject(Integer::create(4));  
eachArr->addObject(Integer::create(5));  
eachArr->addObject(Integer::create(6));  
eachArr->addObject(Integer::create(7));  
eachArr->addObject(Integer::create(8));  
eachArr->addObject(Integer::create(9));  
Object *objectItem;  
//使用 CCARRAY_FOREACH 正向遍历数组  
CCARRAY_FOREACH(eachArr,objectItem ){  
    Integer *intItem = (Integer *)objectItem;  
    log("CCARRAY_FOREACH :%d",intItem->getValue());  
}  
//使用 CCARRAY_FOREACH_REVERSE 逆向遍历数组  
CCARRAY_FOREACH_REVERSE(eachArr,objectItem ){  
    Integer *intItem = (Integer *)objectItem;  
    log("CCARRAY_FOREACH_REVERSE :%d",intItem->getValue());  
}
```

运行程序，在控制台可以看到下面的结果：

```
Cocos2d: CCARRAY_FOREACH :1  
Cocos2d: CCARRAY_FOREACH :2  
Cocos2d: CCARRAY_FOREACH :3  
Cocos2d: CCARRAY_FOREACH :4  
Cocos2d: CCARRAY_FOREACH :5
```



```

Cocos2d: CCARRAY_FOREACH :6
Cocos2d: CCARRAY_FOREACH :7
Cocos2d: CCARRAY_FOREACH :8
Cocos2d: CCARRAY_FOREACH :9
Cocos2d: CCARRAY_FOREACH_REVERSE :9
Cocos2d: CCARRAY_FOREACH_REVERSE :8
Cocos2d: CCARRAY_FOREACH_REVERSE :7
Cocos2d: CCARRAY_FOREACH_REVERSE :6
Cocos2d: CCARRAY_FOREACH_REVERSE :5
Cocos2d: CCARRAY_FOREACH_REVERSE :4
Cocos2d: CCARRAY_FOREACH_REVERSE :3
Cocos2d: CCARRAY_FOREACH_REVERSE :2
Cocos2d: CCARRAY_FOREACH_REVERSE :1

```

2.3.4 字典遍历宏 CCDICT_FOREACH 的使用

CCDICT_FOREACH 同 CCARRAY_FOREACH 类似，只是它遍历的对象是字典类型的数据。CCDICT_FOREACH 的声明在 CCDictionary.h 中，结构如下：

```

#define CCDICT_FOREACH ( __dict__,
    __el__
)

```

其中 __dict__ 为需要遍历的数组，__el__ 为 DictElement 对象。它与 CCARRAY_FOREACH 不同的是，在程序里面不需要对第二个参数进行强制类型转换，因为它已经是确定的类型了。可以把 2.2.9 节的示例代码修改下，用 CCDICT_FOREACH 遍历输出：

```

//创建一个字典，该字典会自动释放
Dictionary* pDict = Dictionary::create();
//插入对象到字典中
String* pValue1 = String::create("100");
String* pValue2 = String::create("120");
String* pValue3 = String::create("200");
pDict->setObject(pValue1, "key1");
pDict->setObject(pValue2, "key2");
pDict->setObject(pValue3, "key3");
DictElement *element = NULL;

```

```
//使用 CCDICT_FOREACH 遍历输出
CCDICTIONARY_FOREACH(pDict,element){
    String *value = (String *)element->getObject();
    log("%s:%s",element->getStrKey(),value->getCString());
}
```

运行程序，在控制台可以看到下面的结果：

```
Cocos2d: {key1:100}
Cocos2d: {key2:120}
Cocos2d: {key3:200}
```

2.3.5 对象创建方法宏 CREATE_FUNC 的使用

CREATE_FUNC 自动生成一个默认的静态 create 方法。CREATE_FUNC 的声明在 CCPlatformMacros.h 中，它的定义如下：

```
#define CREATE_FUNC (__TYPE__)
static __TYPE__* create() {
{
    __TYPE__ *pRet = new __TYPE__();
    if (pRet && pRet->init()) {
    {
        pRet->autorelease();
        return pRet;
    }
    else {
    {
        delete pRet;
        pRet = NULL;
        return NULL;
    }
    }
}
```

我们可以学习这种生成对象的方法，先用构造函数分配空间，之后立即用 init 方法初始化，并且由初始化结果确定能否返回一个可用的对象。在定义特定参数的 create 方法时也应当这样。关于 CREATE_FUNC，在向导创建的 HelloWorld 程序中有相应使用方式，这里就不举例说明了。

2.3.6 属性定义宏 CC_PROPERTY 的使用

CC_PROPERTY 用来声明一个 protected 变量。CC_PROPERTY 的声明在 CCPlatformMacros.h 中，结构如下：

```
#define CC_PROPERTY (varType,
    varName,
    funName
)
```

其中 varType 是变量类型，varName 是变量名称，funName 是指“get + funName”是变量的 getter，“set + funName”是变量的 setter。

使用 CC_PROPERTY 声明的变量可以使用 getter 获取变量的值，使用 setter 设置变量的值。但是在使用 getter 或者 setter 之前需要重构它们，因为它们都是虚函数。比如我们定义一个英雄类，用 CC_PROPERTY 定义三个属性，以及属性的 getter 和 setter，Hero.h 中的主要代码为：

```
class Hero: public CCSprite
{
public:
    //使用 CC_PROPERTY 定义三个属性
    CC_PROPERTY(int, _heroName, HeroName);
    CC_PROPERTY(int, _heroDegre, HeroDegre);
    CC_PROPERTY(int, _heroHp, HeroHp);
    CREATE_FUNC(Hero);
};
```

Hero.cpp 中的主要代码为：

```
//设置英雄名
void Hero::setHeroName(String* pName){
    _heroName = pName;
}
//获取英雄名
String* Hero::getHeroName(){
    return _heroName;
}
```

```
//设置英雄等级
void Hero::setHeroDegre(int degre){
    _heroDegre = degre;
}
//获取英雄等级
int Hero::getHeroDegre(){
    return _heroDegre;
}
//设置英雄血量
void Hero::setHeroHp(int hp){
    _heroHp = hp;
}
//获取英雄血量
int Hero::getHeroHp(){
    return _heroHp;
}
```

在 HelloWorldScene.cpp 的 init 方法里添加一个英雄，并设置英雄属性：

```
//测试 CC_PROPERTY 生成属性
Hero *hero = Hero::create();
//使用 setter 设置属性
hero->setHeroName(String::create("guanyu"));
hero->setHeroDegre(10);
hero->setHeroHp(1000);
//使用 getter 获取属性
String *name = hero->getHeroName();
int degre = hero->getHeroDegre();
int hp = hero->getHeroHp();
log("英雄的名字是 %s, 等级是 %d, 初始血量是 %d", name->getCString(),
degre, hp);
```

运行程序，看到下面的输出结果：

```
Cocos2d: 英雄的名字是 guanyu, 等级是 10, 初始血量是 1000
```

定义属性的宏除了有 CC_PROPERTY 外，还有 CC_PROPERTY_PASS_BY_REF、CC_PROPERTY_READONLY、CC_PROPERTY_READONLY_PASS_BY_REF。它们

的作用和使用方法类似 CC_PROPERTY，主要有以下区别。

- ◎ CC_PROPERTY_PASS_BY_REF 通过 getfunName 返回的值是引用。
- ◎ CC_PROPERTY_READONLY 定义的是只读属性，只能获取，不能改变，即只有 getfunName 方法，没有 setfunName 方法。
- ◎ CC_PROPERTY_READONLY_PASS_BY_REF 同 CC_PROPERTY_READONLY 类似，但返回的值是引用。

2.3.7 命名空间宏

Cocos2d-x 提供了几个命名空间的宏，使用这些宏可以快速定义命名空间。表 2-1 列举了常用的命名空间宏。

表 2-1 命名空间宏

宏	作用	同等作用的 C++ 语句
USING_NS_CC	定义 Cocos2d 命名空间	using namespace cocos2d;
USING_NS_CC_EXT	定义 Cocos2d 扩展功能命名空间	using namespace cocos2d::extension
NS_CC_BEGIN	Cocos2d 命名空间开始	namespace cocos2d {
NS_CC_END	Cocos2d 命名空间结束	}
NS_CC_EXT_BEGIN	Cocos2d 扩展功能命名空间开始	namespace cocos2d { namespace extension {
NS_CC_EXT_END	Cocos2d 扩展功能命名空间结束	}}

2.4 Cocos2d-x 中的坐标和坐标系

上文已经讲到 Cocos2d-x 中点的概念，点就是基于坐标和坐标系存在的。下面就讲讲 Cocos2d-x 中的坐标系和坐标。

2.4.1 OpenGL 坐标系和屏幕坐标系

Cocos2d-x 里包含多种坐标系，最基本的是屏幕坐标系和 OpenGL 坐标系，接下

来讲解这两个坐标系。

1. OpenGL 坐标系

Cocos2d-x 使用的是 OpenGL 坐标系，原点在屏幕左下角， x 轴向右， y 轴向上（图 2-1 所示）。

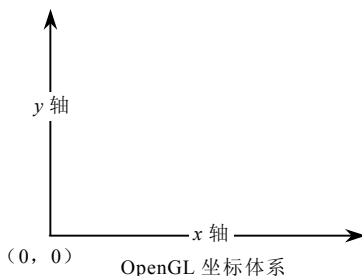


图 2-1 OpenGL 坐标系

2. 屏幕坐标系

屏幕坐标系默认原点在左上角， x 轴朝右， y 轴朝下（如图 2-2 所示）。iOS 的屏幕触摸事件 Touch 传入的位置信息使用的是该坐标系。因此在 cocos2d 中对触摸事件做出响应前需要首先把触摸点转化到 GL 坐标系。Cocos2d-x 已经自动帮我们转换了。

2.4.2 锚点和位置的使用

在 OpenGL 坐标体系中有两个非常重要的参数，即锚点和位置。

锚点是位置的参考点，即在设置某一个物件（如 Sprite、Layer）的位置时它的参照位置，数据为（0.0-1.0）之间，默认是（0.5,0.5），也就是说其默认参照位置是该物件的中心。如果把某一物体放到屏幕的中心，对于默认 anchor 来说，就是把该物体的中心放到屏幕中心。取值在 0 到 1 之间的好处就是，锚点不会和具体物体的大小耦合，也即不用关注物体大小，而应取其对应比率，如果把锚点改成（0,0），进行定位时，则以图片左下角作为参照点把物体放置在指定的点上。下面举例演示

锚点的作用，完整代码请参考代码清单 2-4。

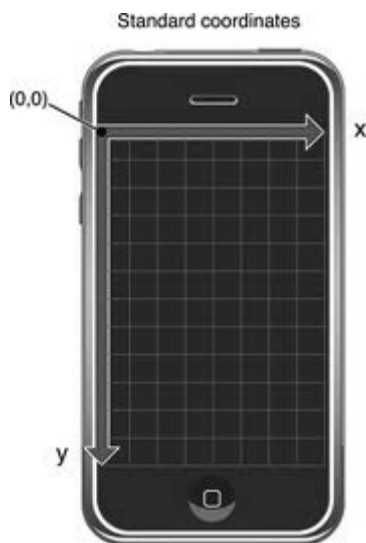


图 2-2 屏幕坐标系

用向导新建一个项目，打开 HelloWorldScene.cpp，把关闭按钮菜单项的位置设为 (0, 0) 点：

```
closeItem->setPosition(Point::ZERO);
```

由于锚点默认是物体中心点，所以只会显示 1/4（图 2-3）。

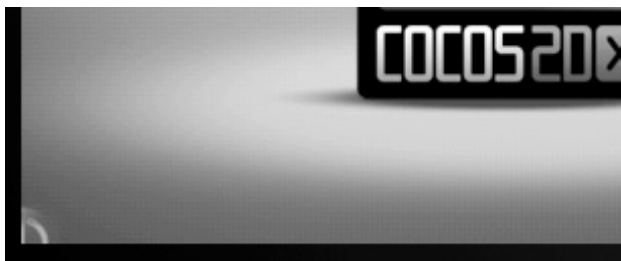


图 2-3 锚点为(0.5,0.5)，关闭按钮只显示 1/4

如果把锚点设置为 (0, 0)：

```
closeItem->setAnchorPoint(Point::ZERO);
```

运行程序就会看到关闭按钮全部显示（图 2-4 所示）。



图 2-4 锚点为(0,0)，关闭按钮全部显示

2.4.3 节点坐标系和世界坐标系的相互转换

节点坐标系和世界坐标系在 Cocos2d-x 中都属于 OpenGL 坐标系，只是它们的坐标原点不同。

1. 世界坐标系

也叫作绝对坐标系，Cocos2d-x 中的元素具有父子关系的层级结构，我们通过 `setPosition` 方法设定元素的位置使用的是相对于其父节点的节点坐标系而非世界坐标系。最后在绘制屏幕的时候 `cocos2d` 会把这些元素的本地坐标映射成世界坐标系坐标。

2. 节点坐标系

游戏画面中每一个对象都有一个自己的坐标系，这些坐标系就叫作节点坐标系。每个物体都有自己的坐标系，当物体移动或改变方向时，和该物体关联的坐标系将随之移动或改变方向，坐标原点是父级对象的左下角， x 轴向右， y 轴向上。

节点坐标和世界坐标可以通过 Cocos2d-x 提供的 4 个函数相互转化。其中，`convertToWorldSpace` 方法把基于当前 `node` 的节点坐标系下的坐标转换到世界坐标系中的坐标，坐标原点在左下角；`convertToWorldSpaceAR` 同 `convertToWorldSpace` 功能一样，但坐标原点是节点的锚点。`convertToNodeSpace` 方法可以把世界坐标的

坐标转换到当前 node 的本地坐标系中的坐标；convertToNodeSpaceAR 同 convertToNodeSpace 功能一样，但坐标原点是节点的锚点。下面分析两个坐标的关系和相互转化。

先在场景中添加一个精灵，称其为 sprite1，sprite1 的属性有：宽 300，高 200，锚点 (0.5, 0.5)，位置 (240, 150)。

再添加一个精灵，称之为 sprite2，sprite2 的属性有：宽 100，高 60，锚点 (0.5, 0.5)，位置 (60, 50)。

经过作图分析，sprite2 的位置 (60, 50) 相对于 sprite1 的左下角 (90, 50) 的节点坐标为 (-30, 0)，相对于 sprite1 锚点 (240, 150) 的节点坐标为 (-180, -100)；sprite2 的位置 (60, 50) 相对于 sprite1 的左下角 (90, 50) 的世界坐标为 (150, 100)，相对于 sprite1 锚点 (240, 150) 的世界坐标为 (300, 200)，使用代码验证上面的结论是否正确。

```
//世界坐标和节点坐标测试
//添加精灵 1
Sprite *sprite1 = Sprite::create("red.png");
sprite1->setPosition(Point(240,150));
addChild(sprite1);
//添加精灵 2
Sprite *sprite2 = Sprite::create("blue.png");
addChild(sprite2);
sprite2->setPosition(Point(60,50));
//获取 layer2 的节点坐标
Point point1 = sprite1->convertToNodeSpace(sprite2->getPosition());
Point point2 = sprite1->convertToNodeSpaceAR(sprite2->getPosition());
//获取 layer2 的世界坐标
Point point3 = sprite1->convertToWorldSpace(sprite2->getPosition());
Point point4 = sprite1->convertToWorldSpaceAR(sprite2-> getPosition
());

log("sprite2's node coordinate(%f,%f)",point1.x,point1.y);
log("sprite2's node coordinate(%f,%f)",point2.x,point2.y);
log("sprite2's world coordinate(%f,%f)",point3.x,point3.y);
log("sprite2's world coordinate(%f,%f)",point4.x,point4.y);
```

运行程序可以看到运行结果如图 2-5 所示。



图 2-5 节点坐标和世界坐标的关系

控制台的输出与上文分析的结果是一样的。

```
Cocos2d: sprite2's node coordinate(-30.000000,0.000000)
Cocos2d: sprite2's node coordinate(-180.000000,-100.000000)
Cocos2d: sprite2's world coordinate(150.000000,100.000000)
Cocos2d: sprite2's world coordinate(300.000000,200.000000)
```

3

第 3 章

Cocos2d-x 核心概念

Cocos2d-x 作为一个游戏框架，提供了便于开发游戏的一些核心概念，这些核心组成了游戏的基本骨架，也是开发过程经常使用到的理念。在 Cocos2d-x 中，游戏就相当于一部电影，整个电影的制作过程由导演统一管理安排，电影的内容由相机拍摄展示，电影由一个又一个不同的场景组成，在镜头中一次只能展示一个场景。一个场景由多个布景组成，演员在场景里表演动作。Cocos2d-x 游戏具有相似的流程，导演类控制游戏运行、管理游戏资源，每一个镜头都是一个场景，场景由不同的布景组成，每个布景又包含多个精灵，精灵就相当于电影里的人物角色。下面介绍这些核心概念的实例应用。

3.1 基础节点

3.1.1 Node 简介

Cocos2d-x 的核心类的实例都是一个节点，这些类是以树状结构继承的（参考 C++ 的继承），它们的基类都是 Node。

Node 主要特征有，可以包含其他 Node 节点，可以设置定期回调函数，可以执行动作。

我们可以继承 Node 制作自己需求的子类，在继承时一般要做的事情包括重写 init 方法初始化节点对象，创建回调函数，重写 draw 函数渲染节点。

Node 继承自 Ref 类，其继承关系如图 3-1 所示。

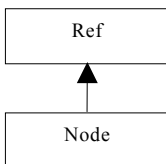


图 3-1 Node 类继承图

注意：尽量把游戏场景中用到的对象（例如精灵、菜单等）在 init 方法中创建好。如果不想让玩家在进入场景时就看到，可以把这些节点的 setVisible 属性先设置为 false；在后面需要使用这些预分配的对象时，就可以直接把 visible 设置为 true，这样做能提高游戏的性能。这种预先分配对象的策略，也叫作 lazy loading。

3.1.2 Node 应用举例之移除节点

Node 常用的地方就是，当需要对一个节点操作且不确定或没必要知道这个节点的确切类型时（如布景或者精灵等），就可以用 Node 对象获取该节点。这个实例把 helloworld 程序的关闭游戏功能，改为移除 Hello World 字样。实例代码如下，完整代码请看代码清单 3-1-2。

（1）使用 Visual Studio 2010 在解决方案中添加项目 3-1-2。

（2）打开项目 NodeTest 中的 HelloWorldScene.cpp，在 HelloWorld::init 方法中为 label 设置一个标记：

```
pLabel->setTag(12); //设置 HelloWorld 标签的 tag 标记，以便获取该对象
```

（3）把 menuCloseCallback 回调函数中的代码替换成如下代码：

```
do
{
    Node *pNode = this->getChildByTag(12); //获取 HelloWorld 标签
    CC_BREAK_IF(! pNode); //判断 pNode 是否存在，如果不存在跳出循环
    pNode->removeFromParentAndCleanup(true); //移除标签
} while (0);
```

（4）运行程序，当单击关闭按钮时，HelloWorld 这几个字就会消失。

3.2 相机

3.2.1 相机简介

Cocos2d-x 引擎里的相机使用 CCCamera 类实现，每个节点（Node）都需要使用 CCCamera，有了相机，节点才会被渲染成大家可以看到的东西。当节点发生缩放、旋转和位置变化等时，都需要覆盖 CCCamera，让这个节点通过 CCCamera 重新渲染。注意，CCCamera 的概念在 3.x 中已经被移除，但还可以使用，因为很多开发者还在使用 2.X 系列，所以这里也提一下。

CCCamera 继承自 Ref 类，其继承关系如图 3-2 所示。

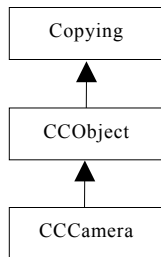


图 3-2 Camera 类继承图

3.2.2 使用 CCCamera 循环缩放点

这个实例展示了使用 CCCamera 缩放、旋转节点的功能。通过 CCCamera 的 setEyeXYZ 方法实现，setEyeXYZ 函数通过设置视角的 x 、 y 、 z 轴缩放旋转节点。实例代码如下，完整代码请看代码清单 3-2-2。

```
//创建精灵对象
Sprite* pSprite = Sprite::create("jinwei.png");
CC_BREAK_IF(! pSprite);
//设置精灵位置
pSprite->setPosition(Point(size.width/2, size.height/2));
pSprite->setTag(10);
//把精灵添加到布景中
this->addChild(pSprite, 0);
m_z = 0;
scheduleUpdate();
```

这段代码在视图区添加一个精灵对象，并通过 scheduleUpdate(); 设置一个定时器，定期刷新屏幕，刷新通过执行类的 update 方法实现，所以我们需要添加 update 方法，如下所示：

```
Node *sprite;
CCCamera *cam;
if(zoom==0){
    if(m_z<=200){
        m_z += 5;
    }else{
```

```

        zoom=1;
    }
}
}else{
    if(m_z>0){
        m_z -= 5;
    }else{
        zoom=0;
    }
}
}
sprite = getChildByTag(10);
cam = sprite->getCamera();
cam->setEyeXYZ(0, 0, m_z);

```

通过 `getChildByTag` 方法获取上面添加的精灵，使用 `getCamera` 方法获取该精灵的相机，然后使用相机的 `setEyeXYZ` 方法对精灵进行循环播放，效果如图 3-3 所示。

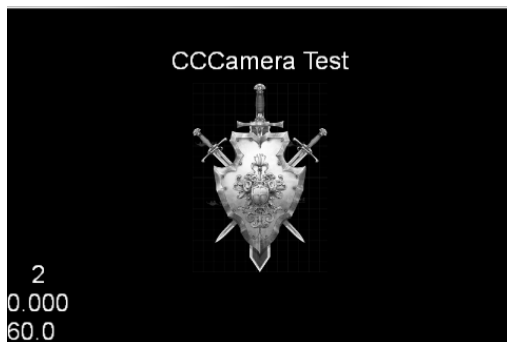


图 3-3 使用 CCCamera 循环缩放节点

3.3 导演

3.3.1 导演 Director 简介

Cocos2d-x 引擎里的相机使用 `Director` 类实现，导演对象是流程的总指挥，它负责游戏全过程的场景切换。且整个游戏里只有一个导演，游戏开始和结束时都需要调用 `Director` 的方法完成游戏初始化或者销毁的工作。

Director 继承自 Ref 类，其继承关系如图 3-4 所示。

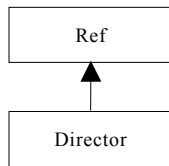


图 3-4 Director 类继承图

3.3.2 Director 常用功能举例

获取 Director 对象：

```
Director *pDirector = Director::getInstance();
```

打开关闭 FPS：

```
pDirector->setDisplayStats(true); 参数为 true 打开 FPS，为 false 关闭 FPS
```

设置刷新频率：

```
pDirector->setAnimationInterval(1.0 / 60);
```

运行某一场景：

```
pDirector->runWithScene(pScene); //pScene 为 Scene 对象
```

退出游戏：

```
pDirector->end();
```

暂停游戏：

```
pDirector->pause();
```

当暂停游戏时，仍然会绘制游戏画面，但停止所有动作、效果和定时器，且屏幕刷新率降到 4 FPS，结省电和内存等资源。

返回游戏:

```
pDirector->resume();
```

获取窗口大小:

```
pDirector->getWinSize();
```

设置游戏窗口大小:

在 main.cpp 里可以看到下面的代码:

```
EGLView* eglView = EGLView::getInstance();  
eglView->setFrameSize(480, 320);
```

其中 `setFrameSize` 就是设置游戏窗口大小, 比如需要设置成 960×640 像素的大小, 就可以用下面这句代码实现:

```
eglView->setFrameSize(960, 640);
```

3.4 场景

3.4.1 场景定义

Cocos2d-x 引擎里的场景使用 `Scene` 类实现, `Scene` 中存放需要渲染的布景、人物角色和菜单等, 这些对象可以作为一个整体, 一起渲染, 一起销毁, 一起被场景切换使用。

`Scene` 承担的作用是一个容器的功能。游戏开发时把多个需要渲染的对象放在 `Scene` 里面统一管理, 包括创建、销毁和场景切换等, 且同一时间只能运行一个场景。

`Scene` 常用的方法就是 `create`, 用来创建场景对象。`Scene` 继承自 `Node` 类, 其继承关系如图 3-5 所示。

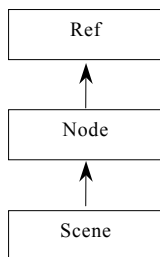


图 3-5 Scene 类继承图

3.4.2 创建显示战斗场景

该实例展示了如何在游戏中创建一个战斗场景类，并把该场景加载到游戏中。步骤和核心代码如下。完整代码请查看代码清单 3-4-2。

(1) 创建项目 3-4-2。

(2) 把战斗场景图片 FightScene.png 添加到项目里。

(3) 添加 FightScene.h 和 FightScene.cpp 到项目中，在头文件 FightScene.h 中，我们定义 FightScene 类，FightScene 类继承自 Layer，并声明初始化方法 init 和生成 FightScene 对象的静态方法 scene。代码如下：

```

#ifndef __FIGHT_SCENE_H__
#define __FIGHT_SCENE_H__
#include "cocos2d.h"
class FightScene : public cocos2d::Layer
{
public:
    virtual bool init();//初始化函数
    static cocos2d::Scene* scene();//生成 Fight 对象的静态方法
    CREATE_FUNC(FightScene );//生成 create 方法
};
#endif // __FIGHT_SCENE_H__
  
```

(4) 在源文件 `FightScene.cpp` 中,我们先实现 `scene` 静态方法,它返回一个包含 `FightScene` 对象的场景,代码如下:

```
Scene* FightScene::scene()
{
    Scene * scene = NULL;
    do
    {
        scene = Scene::create();//创建场景
        CC_BREAK_IF(! scene);
        FightScene *layer = FightScene::create();//创建 Fight 对象
        CC_BREAK_IF(! layer);
        scene->addChild(layer);
    } while (0);
    return scene;
}
```

然后在初始化函数 `init` 里为 `Fight` 对象加入场景的背景图片,并放在场景中间,代码如下所示:

```
CC_BREAK_IF(! Layer::init());
Size size = Director::getInstance()->getWinSize();
//创建战斗场景背景
Sprite* pSprite = Sprite::create("FightScene.png");
CC_BREAK_IF(! pSprite)
//把精灵对象放在场景中间
pSprite->setPosition(Point(size.width/2, size.height/2));
//添加精灵对象到 Fight 布景里
this->addChild(pSprite, 0);
```

(5) 运行程序,就能看到游戏的战斗背景画面了,如图 3-6 所示。

3.4.3 动态切换多个场景

该实例在 3.4.2 实例的基础上添加而成,演示了如何进行场景切换,并在切换时添加动态效果。当进行游戏后,会看到进入一个村落,村落中有一个“进入战斗”的标记,单击该标记就能进入战斗场景。步骤和核心代码如下所示。完整代码请查看代码清单 3-4-3。



图 3-6 创建战斗场景效果图

(1) 按照上一节所示方法添加村落场景，头文件为 CityScene.h，源文件为 CityScene.cpp。

(2) 在 AppDelegate.cpp 中把初始化场景改成村落场景。

(3) 打开 CityScene.cpp 文件，在村落场景中添加“进入战斗”标记，我们使用图片菜单来实现该标签，更多关于菜单的使用请查看后继章节。我们在 init 方法中添加背景图片代码的后面添加如下代码：

```
/* 创建一个 MenuItemImage 对象 */
MenuItemImage* fightItem=MenuItemImage::create(
    "goFightScene.png",NULL,this,menu_selector(CityScene::goFightScene));
/* 创建一个 Menu 对象，需要 Item 对象作为参数 */
Menu* menu = Menu::create(fightItem, NULL);
menu->setPosition(Point(size.width/2, size.height/2));
/* 最后将 Menu 菜单添加到场景中 */
this->addChild(menu);
```

运行程序就能看到村落场景，如图 3-7 所示。



图 3-7 创建村落场景效果图

(4) 接下来我们添加单击菜单后回调函数 `goFightScene`。

在 `CityScene.h` 中添加 `goFightScene` 函数声明：

```
void goFightScene(Ref* pSender);
```

在 `CityScene.cpp` 中添加 `goFightScene` 函数实现：

```
void CityScene::goFightScene(Ref* pSender){
    Scene *fightScene = FightScene::scene();
    Director::getInstance()->replaceScene(fightScene);
}
```

这里我们先生成战斗场景对象，然后用 `Director` 的函数把当前场景替换成战斗场景。

注意：回调函数必须加参数 `Ref* pSender`，否则就会报错。

(5) 这时单击“进入战斗”，就会切换到战斗场景，但切换很快，如果有个动态效果交互就会很好，下面我们就添加一些切换效果。我们把 `goFightScene` 函数内的代码换成如下代码：

```
Scene *fightScene = FightScene::scene();
TransitionTurnOffTiles* transition
    =TransitionTurnOffTiles::create(1.0, fightScene);
Director::getInstance()->replaceScene(transition);
```

运行程序，再单击“进入游戏”，就以瓷砖效果切换到战斗。

(6) 按上面提示的方法，为战斗场景添加返回到村落功能。场景切换时我们使用 `TransitionFlipX`，即按 `x` 轴旋转的方式进行切换，更多渐变效果请查看第 6 章的内容。

3.5 布景

3.5.1 布景定义

Cocos2d-x 引擎里的布景使用 `Layer` 类实现，每个游戏场景中都可以有很多布景，每个布景负责各自的任務，例如专门负责显示背景、专门负责显示道具或者专门负责显示人物角色等。在每个布景上面可以放置不同的元素，包括文本、精灵和菜单等。通过布景以及布景与布景之间的组合关系，我们就能够让游戏显示出各种各样的界面了。

`Layer` 继承自 `Node` 类，其继承关系如图 3-8 所示。

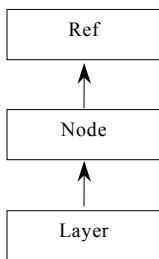


图 3-8 Layer 类继承图

Cocos2d-x 基于 `Layer` 又提供了几个布景类，这些类除了具有 `Layer` 所有的功能外，也增加了更多的功能，它们的简介和对比如下所示。

- ◎ LayerRGBA 使用纯色做背景的 Layer，它能设置透明度和背景颜色，背景色为 RGB 格式，如 Color3B(100, 100, 0)，意思为 R=100, G=100, B=0。
- ◎ LayerColor 同 LayerRGBA 一样，可设置透明度和背景颜色。
- ◎ LayerGradient 是 LayerColor 的子类，主要功能是设置背景为渐变色。
- ◎ LayerMultiplex 能够包含一到多个对象，但同一时刻只激活一个对象。

3.5.2 使用 Layer 模拟 Windows Phone 主界面

微软推出智能操作系统 Windows Phone 8，它的 UI 风格被命名为 Metro 设计，不管 Windows Phone 8 如何，Metro 设计确实给人眼前一亮的感觉，这一节我们就用 LayerColor 实现 Metro 界面。Metro 界面由一个个正方形色块组成，这些正方形分小、中、大三种。该实例中，窗口大小为 320，所以小正方形的宽高为 70，中正方形的宽高为 148，大正方形的宽高为 304，它们的间距为 8（单位均为像素）。

步骤和核心代码如下所示。完整代码请查看代码清单 3-5-2。

（1）新建项目 3-5-2，在 main.cpp 里把窗口大小调整为 320×480。

（2）在 init 方法里，使用 LayerColor 生成一个蓝色小正方形对象：

```
LayerColor *smallLayer1 = LayerColor::create(Color4B(45,137,240,255),
70,70);
```

这里使用 LayerColor 的 create 方法生成一个带背景色的布景，参数分别为颜色值、宽度、高度。c4 用来生成 RGBA 颜色，参数分别为 R 值、G 值、B 值、透明度。

然后设置该正方形的位置并添加到场景里：

```
smallLayer1->setPosition(Point(8,8));
addChild(smallLayer1);
```

同理，再创建 3 个 LayerColor 对象，使它们整齐地排成一行。

（3）使用 LayerGradient 添加一个带渐变色的布景。

首先创建一个 LayerGradient 对象：

```
LayerGradient *bigLayer =LayerGradient::create(Color4B(228, 208, 0, 255), Color4B(220,62, 17, 255));
```

该方法的第一个参数为起始颜色，第二个参数为结束颜色，渐变方向默认为从上到下。另外，LayerGradient 为我们提供了一些方法来改变一些属性值。

- ◎ setStartColor: 设置起始颜色值。
- ◎ setEndColor: 设置结束颜色值。
- ◎ setStartOpacity: 设置起始透明度。
- ◎ setEndOpacity: 设置结束透明度。
- ◎ setVector: 设置渐变方向。

之后把 bigLayer 添加到场景中：

```
bigLayer->setContentSize(Size(304, 148));
bigLayer->setPosition(Point(8,242));
addChild(bigLayer);
```

(4) 使用同样的方法添加其他正方形，运行项目，效果如图 3-9 所示。

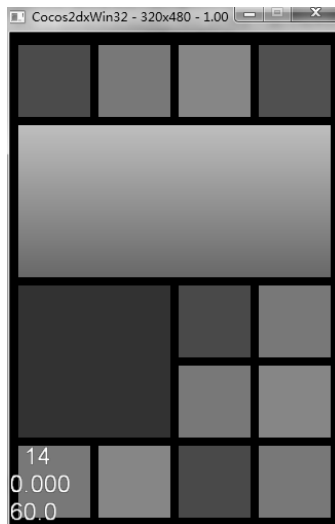


图 3-9 模拟 Windows Phone 主界面效果图

3.6 精灵

Cocos2d-x 引擎里的精灵使用 **Sprite** 类实现，精灵是整个游戏开发处理的主要对象，天上的飞机、地上的坦克、玩家控制人物等都是精灵。甚至随机飘过的一片云、飞过的一只鸟也是精灵。技术上讲，精灵是一个可以不断变化的图片。这些变化包括位置变化、旋转、放大、缩小和运动等。

Sprite 继承自 **Node** 和 **TextureProtocol** 类，其继承关系如图 3-10 所示。

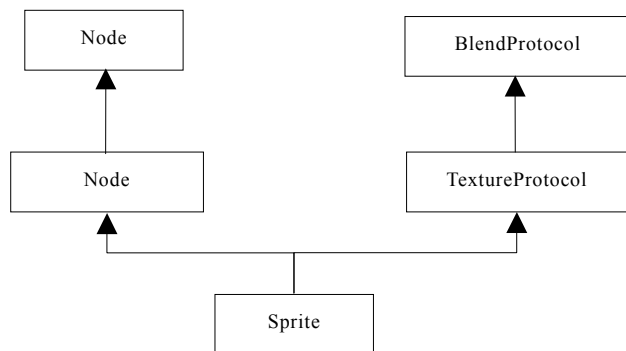


图 3-10 Sprite 类继承图

精灵最常用初始化方法有下面两种：

```
//通过图片来获取
Sprite* sprite = Sprite::create("hero.png");
//通过纹理获取
Texture2D* cache = TextureCache::getInstance()->addImage("hero.png");
Sprite* sprite = Sprite::spriteWithTexture(cache);
```

Cocos2d-x 提供另外一个类 **SpriteBatchNode**，该类用来批量绘制精灵，提高精灵绘制的效率，需要绘制的精灵越多，效果越明显。它能提高绘制效率的原因是：Cocos2d-x 使用 **opengl es** 绘制图片，每次绘制都要走 **open-draw-close** 流程；而 **SpriteBatchNode** 把所有精灵一起绘制，不需要单个绘制每个精灵，流程就变成了

open-draw-draw-...-close，避免了多次 open 和 close 的过程，所以提高了绘制效率。

注意：当使用 SpriteBatchNode 时，该对象的所有子节点都必须和它使用同一张图片，简单说就是要把所有精灵放在一个图片上面。



第 4 章

Cocos2d-x 用户界面

本章介绍 Cocos2d-x 中的图形用户界面，包括文本渲染、菜单按钮、滚动窗口和扩展控件。这些图形用户界面是游戏开发中最常用到的，能够提高视觉效应，增强交互体验，提高工作效率。例如使用 Cocos2d-x 提供的多种 label 显示不同样式的文字，菜单几乎包揽了所有点击按钮的作用，滚动窗口可以显示多屏内容。总之，用好 Cocos2d-x 的用户界面，您会得到不小的收获。

4.1 文本渲染

Cocos2d-x 提供多种 label 显示文字，可以使用系统字体，也可以自定义字体，最主要的有三个：LabelBMFont、LabelTTF 和 LabelAtlas。下面就介绍如何自定义

字体，并使用 Cocos2d-x 提供的文本类把这些文字显示出来。

4.1.1 制作 fnt 格式字体

本节介绍如何在 Windows 系统中使用 Bitmap Font Generator 制作 fnt 格式字体。fnt 格式字体就是位图字体，它包括一个图片文件 (*.png) 和一个字体坐标文件 (*.fnt)。

1. 下载并安装软件

到网站 <http://www.angelcode.com/products/bmfont/> 下载 Bitmap Font Generator 安装软件，笔者下载的是 v1.13 版本。下载完成后双击 install_bmfont_1.13.exe 开始安装，如图 4-1 所示。

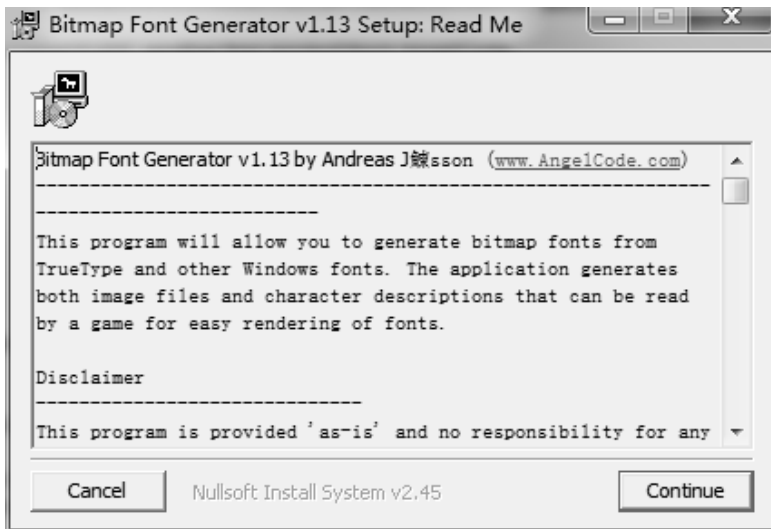


图 4-1 安装 Bitmap Font Generator

接下来就是不停地单击 Next 按钮或者 Continue 按钮。提示安装成功后运行软件，就能看到初始界面如图 4-2 所示。

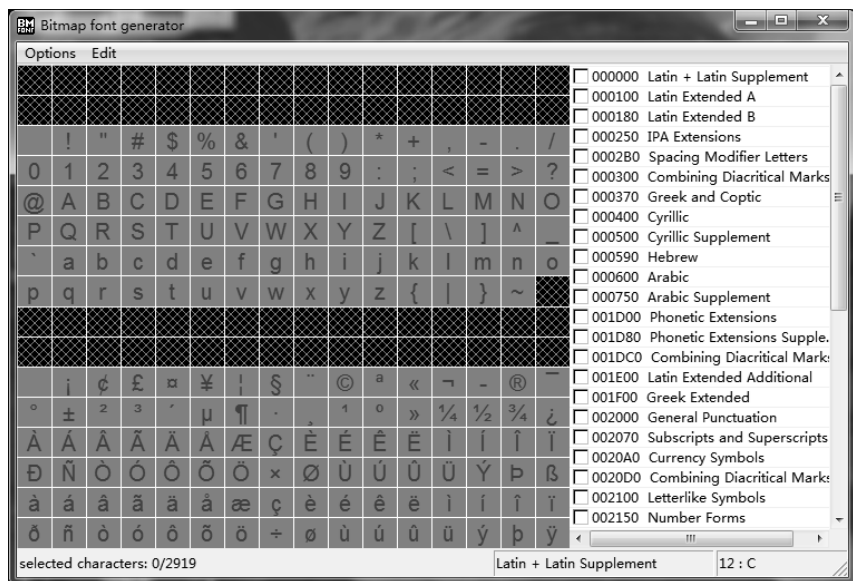


图 4-2 Bitmap Font Generator 主界面

2. Bitmap Font Generator 功能介绍

在图 4-2 中，右边的列表是笔者的字体库，各种不同的字体，左边的就是单独的字体。注意：为了符号等输入，请勾选右边列表的第一项 **Latin+Latin Supplement**。Bitmap Font Generator 提供了很多选项和设置让你制作优美的字体。

(1) Option->FontSetting，用来设定需要导出的字体。Font 可以设置字体，笔者选择微软雅黑；Charset 设置保存的文件格式；Size 可以设定字体大小，最好需要多大就设定多大；Height 可以设定字体的拉伸高度，保持默认的 100%（如图 4-3 所示）。

(2) Edit->Clear all chars in font，清空所有已选择的字符。

(3) Edit->Selecting text from file，选择一个文件，里面包含想要生成字体的文字。

(4) Option->ExportOptions，设定导出的样式（图 4-4）。在该设置界面需要注意几点。

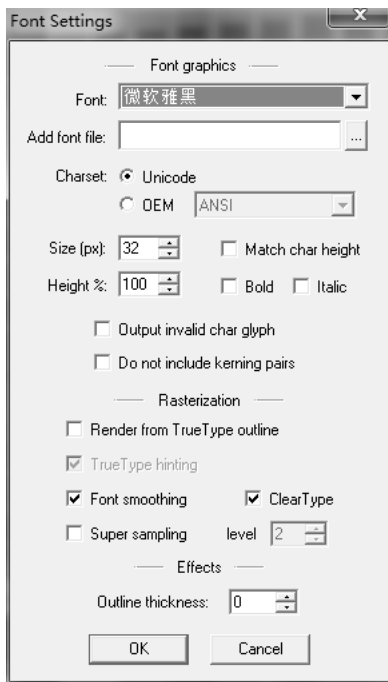


图 4-3 输出字体设置界面

- ◎ **Padding**，设置文字的内边框，或者理解为文字的周边留空要多大，做后期样式时这个属性很重要，需要预留空间来给描边、发光等特效使用。比如我预计我的样式要加一个 3px 的边框，然后在下面有 2px 的投影效果，所以 padding 可以设定为：3px 5px 5px 3px。
- ◎ **BitDepth**，必须 32 位，否则没有透明层。
- ◎ **Presets**，字体初始化的预设颜色通道设定，也就是说字体的初始颜色设定是什么样的，建议都用白色字，可以直接设定为 **White text with alpha**，即白色字透明底。
- ◎ **Font descript**，字体描述文件，可以使用 text 或者 xml，也就是 fnt 文件格式。
- ◎ **Textures**，纹理图片格式，选择 png。

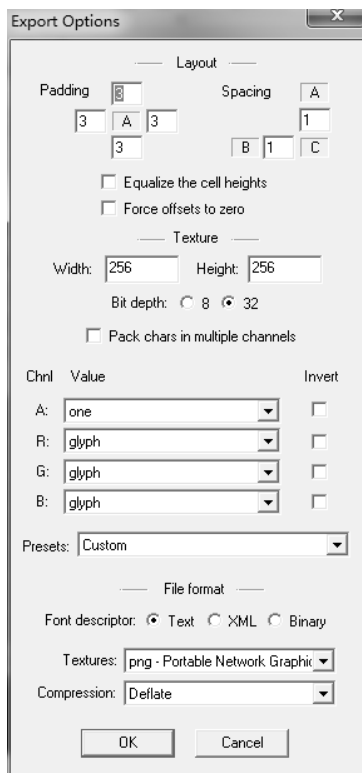


图 4-4 ExportOptions 设置页面

(5) Options->save bitmap font as..., 保存选择的字符为 fnt 格式。

3. 从 txt 文件创建字体

上面介绍了 Bitmap Font Generator 的安装和功能界面, 接下来我们从一个 txt 文本文件创建自己需要的 fnt 字体。相关设置在上文中已经设置好, 这一步就不赘述。

- (1) 新建一个 bitfont.txt 文件, 输入“我爱玩游戏”, 保存为 Unicode 格式。
- (2) 选择菜单 Edit->Selecting text from file, 把 bitfont.txt 文件导入到软件中。
- (3) 选择菜单 Options->save bitmap font as..., 把文件保存为 bitfont, 确定后就

能看到生成的两个文件 `bitfont.fnt` 和 `bitfont.pnt`。

4. 修饰美化字体

用 photoshop 处理导出的 png 文件，修饰美化图片里的字体。如果在系统安装了一些艺术字体，然后再加上后期处理，就可以创造出神奇的字体。

4.1.2 使用 LabelBMFont 显示文本

LabelBMFont 使用位图来显示文字。在使用 LabelBMFont 之前，需要添加好字体文件，包括一个图片文件 (*.png) 和一个字体坐标文件 (*.fnt)。当 LabelBMFont 渲染文字时，把图片加载到内存中，然后只改变图片坐标，就是以占用更多内存为代价加快标签的更新。它继承自 Node、LabelProtocol、BlendProtocol，如图 4-5 所示。

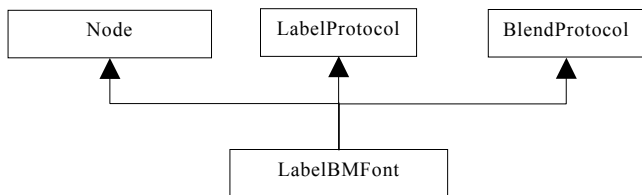


图 4-5 LabelBMFont 类继承图

LabelBMFont 具有下面这些优缺点。

- ◎ 不能设置字体大小，但可以用 `scale` 属性进行缩放来调整大小。
- ◎ 每个字母都可以当成一个精灵，进行旋转、缩放，执行动作。
- ◎ 可以当菜单项使用。

下面展示用 LabelBMFont 在屏幕中显示“我爱玩游戏”，并依次对每个字设置单独的样式。详细代码请看代码清单 4-1。

首先把上一节制作的位图字体添加到项目的 Resources 中，然后在 `init` 方法里创建 LabelBMFont 对象，并添加到场景中，放置在屏幕中间。


```
//生成 LabelBMFont 对象
LabelBMFont *bmfont = LabelBMFont::create(" 我 爱 玩 游 戏 ",
"bitfont.fnt");
//显示在屏幕中间
bmfont->setPosition(Point(visibleSize.width/2,visibleSize.height/2));
addChild(bmfont);
```

通过 getChildByTag 获取单个字母，并强制类型装化成 Sprite。

```
//获取每个字，并转化成精灵
Sprite *zhong = (Sprite *) bmfont->getChildByTag(0);
Sprite *guo = (Sprite *)bmfont->getChildByTag(4);
```

下面就可以单独对每个字母进行操作了。先把第一个字母放大一倍，然后让其不停旋转。

```
//放大第一个字母
zhong->setScale(2);
//让第一个字母不停的旋转
zhong->runAction(RepeatForever::create(RotateBy::create(0.4,
180)));
```

对第二个字母，让其向右移动 100 像素，再返回到原来的位置，之后不停地移动返回。

```
//让第二个字母移动一段距离后再返回
MoveBy *move = MoveBy::create(1, Point(100,0));
Sequence * moveSe = Sequence::create(move,move->reverse(), NULL);
guo->runAction(RepeatForever::create(moveSe));
```

运行程序，查看运行效果，如图 4-6 所示。

4.1.3 使用 LabelTTF 显示文本

LabelTTF 支持系统的字体，也支持自己添加的 ttf 字体，也可以使用 fnt 格式的字体。但是使用 LabelTTF 渲染文本很慢，所以尽量使用另外两种文本渲染类。但 LabelTTF 可以设置字体大小。



图 4-6 使用 LabelBMFont 显示文本

LabelTTF 继承自 Node、LabelProtocol、BlendProtocol，如图 4-7 所示。

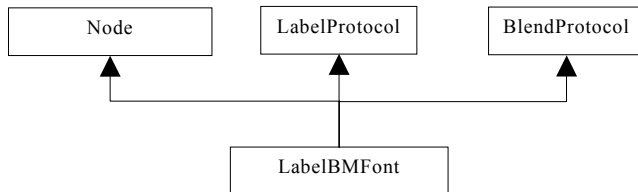


图 4-7 LabelTTF 类继承图

下面展示用 LabelTTF 在屏幕中显示“我爱玩游戏”，并设置字设的样式。详细代码请看代码清单 4-1。

首先把 arialn.ttf 字体添加到项目的 Resources 中，然后在 init 方法里创建 LabelTTF 对象，并添加到场景中。使用 create 方法生成一个对象，字体为自定义的 arialn，大小为 30：

```
LabelTTF *labelTtf = LabelTTF::create("中国", " bitfont ", 30);
```

使用 setColor 设置字体颜色为红色：

```
labelTtf->setColor(Color3B(255, 0, 0));
```

使用 setDimensions 设置文字显示区域的大小：

```
labelTtf->setDimensions(Size(300, 60));
```

使用 `setVerticalAlignment` 设置垂直对齐方式为上对齐，垂直对齐方式有上、中、下对齐方式，对应的参数值为 `Label::VAlignment::TOP`、`Label::VAlignment::MIDDLE`、`Label::VAlignment::BOTTOM`。

```
labelTtf->setVerticalAlignment(Label::VAlignment::TOP);
```

使用 `setHorizontalAlignment` 设置水平对齐方式为左对齐，水平对齐方式有左、中、右对齐方式，对应的参数值为 `Label::VAlignment::LEFT`、`Label::VAlignment::MIDDLE`、`Label::VAlignment::RIGHT`。

```
labelTtf->setHorizontalAlignment(Label::HAlignment::LEFT);
```

使用 `setFontSize` 设置字体大小为 36：

```
labelTtf->setFontSize(36);
```

把改文本标签添加到场景中，并定位：

```
labelTtf->setPosition(Point(visibleSize.width/2,100));  
addChild(labelTtf);
```

运行程序，就能看到在游戏界面中添加了“我爱玩游戏”五个红色文字，如图 4-8 所示。



图 4-8 使用 LabelTTF 显示文本

4.1.4 使用 LabelAtlas 显示文本

LabelAtlas 使用图片显示文字，所以就可以给文字添加各种效果。

LabelTTF 继承自 AtlasNode 和 LabelProtocol，如图 4-9 所示。

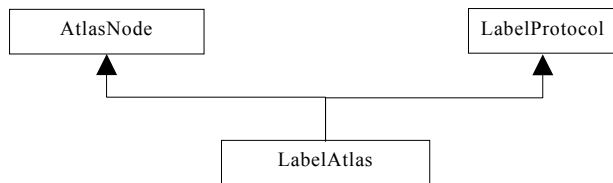


图 4-9 LabelAtlas 类继承图

LabelAtlas 提供静态函数 create 创建 LabelAtlas 对象。create 方法声明如下。

```
static LabelAtlas* create (  const char *   string,
                           const char *   charMapFile,
                           unsigned int    itemWidth,
                           unsigned int    itemHeight,
                           unsigned int    startCharMap )
```

参数说明如下。

- ◎ string: 要显示的文本字体。
- ◎ charMapFile: 关联的字符集的图片。
- ◎ itemWidth: 每一个字符图片占用的宽度，单位为 px。
- ◎ itemHeight: 每个字符图片占用的高度，单位为 px。
- ◎ startCharMap: 图片开始的字符的 ASCII 码。

使用该函数时，LabelAtlas 关联图片中的字符要按 ASCII 码的顺序排列。startCharMap 指定了开始字符的 ASCII 码，并且根据 itemWidth 和 itemHeight 来顺序排列各个字符。在显示字符时，将 string 参数的值进行遍历，每次遍历获取当前字符相应的 ASCII 码，接着根据开始字符的 ASCII 码及每个字符的高度和宽度匹配该字符在图片中的位置，然后显示出来。所以当使用 LabelAtlas 时，字符的 ASCII 码要连续，并且按 ASCII 码顺序排列，这样局限很大。这也决定了使用 LabelAtlas

显示中文很麻烦，所以一般用 LabelAtlas 显示字母或数字。

下面展示用 LabelAtlas 在屏幕中显示一窜数字。详细代码请看代码清单 4-1。

首先把 word.png 添加到项目中，该图片包含大写字母 A 到 Z，每个字符的高度是 20，宽度是 10，第一个字符的 ASCII 码是 65。

创建 LabelAtlas 对象，添加到画面中。

```
LabelAtlas *atlas = LabelAtlas::create("LOVE", "word.png", 10, 20, 65);  
atlas->setPosition(Point(visibleSize.width/2, visibleSize.height/2+  
50));  
addChild(atlas);
```

运行程序，可以在界面中看到“LOVE”字样，如图 4-10 所示。



图 4-10 使用 LabelAtlas 显示“LOVE”

4.2 菜单

菜单一般用来响应单击事件，当单击一个菜单时你可能想开始游戏、结束游戏、移动任务或者更换装备等。

4.2.1 菜单和菜单项的简单使用

Menu 类更多承担的是一个容器的作用，不会关联任何回调方法。一个菜单往往包含一到多个菜单项，每个菜单项都有不同的作用。

Menu 继承自 **Layer**，如图 4-11 所示。

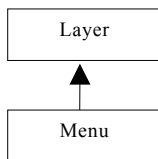


图 4-11 Menu 类继承图

菜单项需要添加到菜单 **Menu** 中才能添加到游戏界面中。根据不同的展现形式，把菜单项分成 6 种。

- ◎ **MenuItemLabel**: 用来展示文本菜单项，它可以把上节讲到的文本标签制作成一个菜单。
- ◎ **MenuItemAtlasFont**: 功能类似 **MenuItemLabel**，不同的是它不需要基于已有的 **label** 对象，而是直接使用文字做参数，用法同 **LabelAtlas** 相似。
- ◎ **MenuItemFont**: 也是直接使用文字展示菜单项，用法同 **labelTTTF**。
- ◎ **MenuItemSprite**: 把一个精灵做成菜单。
- ◎ **MenuItemImage**: 使用图片制作菜单。
- ◎ **MenuItemToggle**: 切换两个菜单项。

接下来简单演示如何在游戏中使用菜单，完整代码请看代码清单 4-2-1。

首先添加一个回调函数 **Roate**，当单击某个菜单时，该菜单会旋转 360°。

```
void HelloWorld::Roate(Ref *pSender){
    Node *node = (Node *)pSender;
    node->runAction(RotateBy::create(0.5, 360));
}
```

创建一个菜单 **Menu**，并添加到游戏界面中：

```
Menu* menu = Menu::create();
menu->setPosition(Point::ZERO);
this->addChild(menu, 1);
```

使用 **MenuItemLabel** 创建菜单项，并添加到菜单 **menu** 中：

```
LabelTTF *label = LabelTTF::create("开始游戏", "Arial", 20);
MenuItemLabel *itemLabel = MenuItemLabel::create(label,
CC_CALLBACK_1 (HelloWorld::Roate,this));
itemLabel->setPosition(Point(visibleSize.width/2, 30));
menu->addChild(itemLabel);
```

使用 **MenuItemAtlasFont** 创建菜单项，并添加到菜单 **menu** 中：

```
MenuItemAtlasFont *itemAtlasFont = MenuItemAtlasFont::create("LOVE",
"word.png", 10, 20, 65, CC_CALLBACK_1 (HelloWorld::Roate,this));
itemAtlasFont->setPosition(Point(visibleSize.width/2, 55));
menu->addChild(itemAtlasFont);
```

使用 **MenuItemFont** 创建菜单项，并添加到菜单 **menu** 中：

```
MenuItemFont *itemfont = MenuItemFont::create("开始游戏", CC_CALLBACK_1
(HelloWorld::Roate,this));
itemfont->setPosition(Point(visibleSize.width/2, 90));
menu->addChild(itemfont);
```

使用 **MenuItemSprite** 创建菜单项，并添加到菜单 **menu** 中：

```
Sprite *NormalSprite = Sprite::create("CloseNormal.png");
Sprite *SelectedSprite = Sprite::create("CloseSelected.png");
MenuItemSprite *itemSprite = MenuItemSprite::create(NormalSprite,
SelectedSprite, CC_CALLBACK_1 (HelloWorld::Roate,this));
itemSprite->setPosition(Point(visibleSize.width/2, 140));
menu->addChild(itemSprite);
```

使用 **MenuItemImage** 创建菜单项，并添加到菜单 **menu** 中：

```
MenuItemImage *itemImage = MenuItemImage::create("CloseNormal.png",
"CloseSelected.png", CC_CALLBACK_1 (HelloWorld::Roate,this));
itemImage->setPosition(Point(visibleSize.width/2, 200));
menu->addChild(itemImage);
```

使用 MenuItemToggle 创建菜单项，并添加到菜单 menu 中：

```
MenuItemToggle* itemtoggle = MenuItemToggle::createWithCallback  
(CC_CALLBACK_1(HelloWorld::Roate, this),  
MenuItemFont::create("开始"),  
MenuItemFont::create("结束"),  
NULL);  
itemtoggle->setPosition(Point(visibleSize.width/2, 250));  
menu->addChild(itemtoggle);
```

运行程序看到界面中有多个菜单，单击一个菜单，该菜单就会选择一圈，回到开始位置，如图 4-12 所示。



图 4-12 菜单项的简单使用

4.2.2 使用菜单制作游戏菜单功能

该实例基于 3.4.3 节的实例，在主场场景中添加游戏设置菜单，单击该菜单后，进入游戏设置界面，如图 4-13 所示。下面讲解如何制作该菜单功能，完整项目请查看代码清单 4-2-2。



图 4-13 菜单

(1) 新建一个游戏设置菜单类 `GameMenu`。用语句宏 `CREATE_FUNC` 定义 `GameMenu` 类的无参 `create` 方法。并且定义一个静态方法 `scene` 用来创建游戏菜单场景。

```
Scene* GameMenu::scene()
{
    Scene *scene = Scene::create();
    GameMenu *layer = GameMenu::create();
    scene->addChild(layer);
    return scene;
}
```

(2) 重写 `init` 方法，当使用 `create` 方法创建 `GameMenu` 对象时，会调用该方法初始化对象。在 `init` 方法中，添加菜单。

```
LayerColor *layer = LayerColor::create(Color4B(200, 200, 200, 200));
addChild(layer);
//把图片添加到纹理中
Texture2D *navTexture =
TextureCache::getInstance()->addImage("nav.png");
```

使用 `MenuItemSprite` 创建返回菜单，并为它注册单击回调函数 `goBack`：

```
Sprite *gobackNormal = Sprite::createWithTexture(navTexture, Rect(0, 0, 142, 100));
Sprite *gobackSelected = Sprite::createWithTexture(navTexture, Rect(0, 100, 142, 100));
Sprite *gobackDisabled = Sprite::createWithTexture(navTexture, Rect(0, 200, 142, 100));
MenuItemSprite *goback = MenuItemSprite::create(gobackNormal, gobackSelected, gobackDisabled, CC_CALLBACK_0(GameMenu::goBack, this));
goback->setPosition(Point(0, 0));
goback->setAnchorPoint(Point(0, 0));
```

使用 MenuItemSprite 创建商店菜单，并为它注册单击回调函数 MenuSelected:

```
Sprite *shopkNormal = Sprite::createWithTexture(navTexture, Rect(142*1, 0, 142, 100));
Sprite *shopSelected = Sprite::createWithTexture(navTexture, Rect(142*1, 100, 142, 100));
Sprite *shopDisabled = Sprite::createWithTexture(navTexture, Rect(142*1, 200, 142, 100));
MenuItemSprite *shop = MenuItemSprite::create(shopkNormal, shopSelected, shopDisabled, CC_CALLBACK_1(GameMenu::MenuSelected, this));
shop->setPosition(Point(142*1, 0));
shop->setAnchorPoint(Point(0, 0));
shop->setTag(SHOPITEM);
```

使用 MenuItemSprite 创建背包菜单，并为它注册单击回调函数 MenuSelected:

```
Sprite *warehousekNormal = Sprite::createWithTexture(navTexture, Rect(142*2, 0, 142, 100));
Sprite *warehouseSelected = Sprite::createWithTexture(navTexture, Rect(142*2, 100, 142, 100));
Sprite *warehouseDisabled = Sprite::createWithTexture(navTexture, Rect(142*2, 200, 142, 100));
MenuItemSprite *warehouse = MenuItemSprite::create(warehousekNormal, warehouseSelected, warehouseDisabled, CC_CALLBACK_1(GameMenu::MenuSelected, this));
warehouse->setPosition(Point(142*2, 0));
warehouse->setAnchorPoint(Point(0, 0));
```

使用 MenuItemSprite 创建英雄菜单，并为它注册单击回调函数 MenuSelected:

```

    Sprite *heroNormal = Sprite::createWithTexture(navTexture, Rect(142*3,
0, 142, 100));
    Sprite *heroSelected = Sprite::createWithTexture(navTexture,
Rect(142*3, 100, 142, 100));
    Sprite *heroDisabled = Sprite::createWithTexture(navTexture,
Rect(142*3, 200, 142, 100));
    MenuItemSprite *hero = MenuItemSprite::create(heroNormal,
heroSelected, heroDisabled, CC_CALLBACK_1(GameMenu::MenuSelected, this));
    hero->setPosition(Point(142*3, 0));
    hero->setAnchorPoint(Point(0,0));

```

使用 **MenuItemSprite** 创建强化菜单，并为它注册单击回调函数 **MenuSelected**:

```

    Sprite *forgeNormal = Sprite::createWithTexture(navTexture,
Rect(142*4, 0, 142, 100));
    Sprite *forgeSelected = Sprite::createWithTexture(navTexture,
Rect(142*4, 100, 142, 100));
    Sprite *forgeDisabled = Sprite::createWithTexture(navTexture,
Rect(142*4, 200, 142, 100));
    MenuItemSprite *forge = MenuItemSprite::create(forgeNormal,
forgeSelected, forgeDisabled, CC_CALLBACK_1(GameMenu::MenuSelected, this));
    forge->setPosition(Point(142*4, 0));
    forge->setAnchorPoint(Point(0,0));

```

使用 **MenuItemSprite** 创建地图菜单，并为它注册单击回调函数 **MenuSelected**:

```

    Sprite *mapNormal = Sprite::createWithTexture(navTexture, Rect(142*5,
0, 142, 100));
    Sprite *mapSelected = Sprite::createWithTexture(navTexture,
Rect(142*5, 100, 142, 100));
    Sprite *mapDisabled = Sprite::createWithTexture(navTexture,
Rect(142*5, 200, 142, 100));
    MenuItemSprite *map = MenuItemSprite::create(mapNormal, mapSelected,
mapDisabled, CC_CALLBACK_1(GameMenu::MenuSelected, this));
    map->setPosition(Point(142*5, 0));
    map->setAnchorPoint(Point(0,0));

```

使用 **MenuItemSprite** 创建任务菜单，并为它注册单击回调函数 **MenuSelected**:

```

    Sprite *taskNormal = Sprite::createWithTexture(navTexture, Rect(142*6,
0, 142, 100));
    Sprite *taskSelected = Sprite::createWithTexture(navTexture,
Rect(142*6, 100, 142, 100));
    Sprite *taskDisabled = Sprite::createWithTexture(navTexture,
Rect(142*6, 200, 142, 100));
    MenuItemSprite *task = MenuItemSprite::create(taskNormal,
taskSelected, taskDisabled, CC_CALLBACK_1(GameMenu::MenuSelected, this));
    task->setPosition(Point(142*6, 0));
    task->setAnchorPoint(Point(0,0));

```

使用 **MenuItemSprite** 创建设置菜单，并为它注册单击回调函数 **MenuSelected**:

```

    Sprite *setupNormal = Sprite::createWithTexture(navTexture,
Rect(142*7, 0, 142, 100));
    Sprite *setupSelected = Sprite::createWithTexture(navTexture,
Rect(142*7, 100, 142, 100));
    Sprite *setupDisabled = Sprite::createWithTexture(navTexture,
Rect(142*7, 200, 142, 100));
    MenuItemSprite *setup = MenuItemSprite::create(setupNormal,
setupSelected, setupDisabled, CC_CALLBACK_1(GameMenu::MenuSelected, this));
    setup->setPosition(Point(142*7, 0));
    setup->setAnchorPoint(Point(0,0));

```

将创建的菜单添加到场景中:

```

    Menu *navMenu = Menu::create(goback, shop, warehouse, hero, forge, map,
task, setup, NULL);
    navMenu->setAnchorPoint(Point(0,0));
    navMenu->setPosition(Point(0, size.height-100));
    navMenu->setTag(MENUNAV);
    addChild(navMenu);

```

(3) 添加回调函数 **goBack**:

```

void GameMenu::goBack() {
    Director::getInstance()->popScene();
}

```

(4) 添加回调函数 `MenuSelected`。由于有一个当前状态，所以当单击其他菜单时，处于当前状态的菜单需要还原为默认样式的菜单。所以添加一个私有的类成员变量 `menuItem`，用来记录当前选中的菜单。

```
void GameMenu::MenuSelected(Ref* pSender) {
    if(menuItem) {
        //先把当前选中菜单还原为默认样式的菜单
        menuItem->setEnabled(true);
    }
    //获取点击的菜单
    menuItem = (MenuItemSprite *) pSender;
    //把点击的菜单设置为选中状态
    menuItem->setEnabled(false);
}
```

(5) 在主城场景中添加“游戏设置”菜单：

```
LabelTTF *gameSet = LabelTTF::create("游戏设置", "Arial", 18);
gameSet->setColor(ccc3(255, 0, 0));
MenuItemLabel *itemfont = MenuItemLabel::create(gameSet,
CC_CALLBACK_1 (CityScene::goGameMenu,this));

itemfont->setPosition(Point(0, 0));
```

(6) 添加回调函数 `goGameMenu`：

```
void CityScene::goGameMenu(Ref* pSender) {

    Scene *gameMenu = GameMenu::scene();
    TransitionFade * TranScene = TransitionFade::create(0.2,
gameMenu);
    Director::getInstance()->pushScene(TranScene);
}
```

运行程序看到画面中有了“游戏设置”按钮（如图 4-14 所示），单击该按钮就进入游戏设置菜单场景了（如图 4-13 所示）。



图 4-14 在画面中添加游戏设置菜单

4.3 滚动框

当内容太多，一页无法显示全时，就需要用到滚动框。Cocos2d-x 为我们提供了 ScrollView、TableView 等多个控件来完成该功能。

4.3.1 使用 ScrollView 显示多页内容

ScrollView 是一个很灵活的滚动控件，里面的滚动内容不受限制，并且可以设定滚动区域，可以是全屏，也可以是某一块区域，并且可以监听滚动和缩放事件。下面就演示如何使用 ScrollView 显示一个大的背景图，完整代码请看代码清单 4-3-1。

注意：ScrollView 的定义位于扩展空间中，所以如果要使用，需要先引用 cocos-ext.h 头文件，即：

```
#include "cocos-ext.h"
```

再声明扩展命名空间，即：

```
USING_NS_CC_EXT;
```

(1) 首先创建一个 ScrollView 对象和一个容器，该容器可以是 Layer 或者 Node 的对象，用来存放 ScrollView 中的内容。我们为容器添加一个大的图片，该图片大小为 1136×640 像素。

```
ScrollView *scrollview = ScrollView::create();
Layer *container = Layer::create();
Sprite *bg = Sprite::create("fight.jpg");
bg->setAnchorPoint(Point::ZERO);
bg->setPosition(Point::ZERO);
container->addChild(bg);
```

(2) 设置 ScrollView 的显示区域，为了使 ScrollView 显示更灵活，ScrollView 提供了一个设置显示区域的方法，使用该方法设置显示区域大小为 480×320 像素，容器的大小为 1136×640 像素。

```
scrollview->setContentSize(Size(480, 320));
container->setContentSize(Size(1136, 640));
scrollview->setContainer(container);
```

(3) 把 ScrollView 添加到游戏场景中，并放在屏幕中：

```
addChild(scrollview);
scrollview->setPosition(Point(100, 80));
```

(4) 运行程序，就可以拖动图片查看隐藏的部分，如图 4-15 所示。



图 4-15 使用 ScrollView 显多页内容

4.3.2 监听 ScrollView 的滚动和缩放事件

要监听 ScrollView 的滚动和缩放事件，需要继承 ScrollViewDelegate 类，并实现两个事件监听函数，scrollViewDidScroll 用来监听滚动事件，scrollViewDidZoom 用来监听缩放事件。下面我们就基于上节提供的实例的基础上，监听滚动和缩放事件，完整代码请看代码清单 4-3-1。

(1) 在头文件中以 public 方式继承 ScrollViewDelegate 类：

```
class HelloWorld : public cocos2d::Layer, public ScrollViewDelegate{
    ...
}
```

(2) 在头文件中声明 scrollViewDidScroll 和 scrollViewDidZoom：

```
virtual void scrollViewDidScroll(ScrollView* view);
virtual void scrollViewDidZoom(ScrollView* view);
```

(3) 在执行文件中重写上面两个函数：

```
void HelloWorld::scrollViewDidScroll(ScrollView* view){
    log("offset
x=%f,y=%f",view->getContentOffset().x,view->getContentOffset().y);
}
void HelloWorld::scrollViewDidZoom(ScrollView* view){
    log("scale = %f",view->getScale());
}
```

(4) 在 init 方法里设置 ScrollView 在代理：

```
scrollview->setDelegate(this);
```

(5) 运行程序，拖动、缩放图片，就能看到控制台的输出结果如下：

```
offset x=-85.851181,y=-99.833252
offset x=-86.864380,y=-101.842751
...
offset x=-95.848022,y=-107.820580
offset x=-96.861214,y=-108.816879
```


4.3.3 使用 TableView 展示多页内容

使用 ScrollView 可以展示多页内容，但是滚动时太死板，没有那种弹性的感觉，本节我们就用 TableView 来展示多页内容，并提供很好的滚动操作。TableView 继承自 ScrollView。下面展示如何使用 TableView 制作一个简单背包，完整代码请看 4-3-3。

(1) 在 init 方法里创建一个 TableView 对象，宽为 780 像素，高为 365 像素：

```
TableView* tableView = TableView::create(this, Size(780, 365));
```

(2) 设置表格的滚动方向为 ScrollView::Direction::VERTICAL，即垂直方向。Cocos2d-x 一共提供 4 个滚动方向，参考表 4-1。

```
tableView->setDirection(ScrollView::Direction::VERTICAL);
```

表 4-1 TableView 滚动方向

滚动方式	使用方法	解释说明
BOTH	ScrollView::Direction::BOTH	水平、垂直两个方向滚动
HORIZONTAL	ScrollView::Direction::HORIZONTAL	水平滚动
VERTICAL	ScrollView::Direction::VERTICAL	垂直滚动
NONE	ScrollView::Direction::NONE	无滚动

(3) 设置单元格的渲染顺序为从上到下，Cocos2d-x 共提供两种渲染顺序，参考表 4-2。

```
tableView->setVerticalFillOrder(TableView::VerticalFillOrder::TOP_DOWN);
```

表 4-2 TableView 单元格渲染方向

渲染顺序	使用方法	解释说明
TOP_DOWN	TableView::VerticalFillOrder::TOP_DOWN	从上向下渲染单元格
BOTTOM_UP	TableView::VerticalFillOrder::BOTTOM_UP	从下向上渲染单元格

(4) 把 tableView 添加到场景中，并定位：

```
addChild(tableView);
tableView->setTag(100);
```

```
tableView->setAnchorPoint(Point(0, 0));
tableView->setPosition(Point(53, 137));
```

(5) 就要知道如何来渲染数据了，TableView 的使用方式类似 cocoa 框架的 UITableView，用到 TableView 的类需要继承 TableViewDataSource，然后声明几个函数。

- ◎ **tableViewCellTouched**: 触摸单元格时，会调用该函数。
- ◎ **tableViewCellSizeForIndex**: 根据 index 设定每一个单元格的大小。
- ◎ **tableViewCellAtIndex**: 根据 index 为单元格设置内容。
- ◎ **numberOfCellsInTableView**: 返回单元格的个数。

```
class HelloWorld :public Layer,public TableViewDataSource
{
    MenuItemImage *curEquipment;
public:
    ...
    virtual void tableViewCellTouched(TableView* table, TableViewCell*
cell);
    virtual Size tableViewCellSizeForIndex(TableView *table, unsigned int
idx);
    virtual TableViewCell* tableViewCellAtIndex(TableView *table,
unsigned int idx);
    virtual unsigned int numberOfCellsInTableView(TableView *table);
    void equipSelect(Ref* pSender);
    ...
};
```

(6) 实现上面声明的函数，首先设定有 20 行数据，这个值可以变，根据具体需求变更：

```
unsigned int HelloWorld::numberOfCellsInTableView(TableView *table)
{
    return 20;
}
```

再设置每一行的大小：

```
Size HelloWorld::tableCellSizeForIndex(Tableview *table, unsigned int
idx)
{
    return Size(780, 136);
}
```

然后在行中添加数据，Tableview 的每一行都是一个单元 TableViewCell，可以继承 TableViewCell 实现自定义的单元格，这样就能实现各种形式或样式的单元格，这里就用自带的 TableViewCell，为每个单元格添加四个并列的图片菜单，单击菜单时触发一个回调函数。

```
TableViewCell *cell = new TableViewCell();
cell->autorelease();
Menu * menu = Menu::create();
menu->setPosition(Point::ZERO);
for (int i =0 ; i<4; i++) {
    MenuItemImage *item = MenuItemImage::create("equipment.png",
"equipmentSelect.png",
"equipmentSelect.png",CC_CALLBACK_1(HelloWorld::equipSelect,this));
    item->setAnchorPoint(Point::ZERO);
    item->setPosition(Point(190*i+40, 15));
    menu->addChild(item);
}
cell->addChild(menu,-1);
return cell;
```

(7) 实现回调函数 equipSelect。当单击一个装备后，会换成另外一个图片。

```
void HelloWorld::equipSelect(Ref* pSender){
    if (curEquipment) {
        curEquipment->setEnabled(true);
    }
    curEquipment = (MenuItemImage*)pSender;
    curEquipment->setEnabled(false);
}
```

到此，Tableview 的使用已经基本完成，运行程序，我们就能看到效果，如图 4-16 所示。Tableview 也可以监听滚动和缩放事件，使用方式同 ScrollView。



图 4-16 TableView 制作背包

4.3.4 触摸 TableView 里的菜单来滚动 TableView

在 4.3.3 节中，介绍了如何使用 TableView 制作一个简单的背包，但是当你想查看更多的内容时，手指必须放在 TableView 没有菜单的地方才能滚动表格，当手指拖动菜单时，表格不会滚动。这一节就来解决该问题，当拖动菜单时也能使表格滚动。

先来分析下为什么会产生这种情况。在 Cocos2d-x 中菜单对触摸事件的响应级别是最高的，当触摸 TableView 里的菜单时，菜单先截获触摸事件，然后自己消化掉。这样一来，TableView 就接收不到事件，所以不会滚动。

既然知道了原因，解决方法就是把 TableView 的事件响应级别高于菜单的响应级别。这样有两种方式，一种是降低菜单的相应级别，另一种是提高 TableView 的响应级别。这里采用第一种方式。本实例基于 4.3.3 节，完整代码请查看 4-3-4。

(1) 添加一个自定义类菜单类 CustomMenu，该类继承自 Menu。

(2) 重构 registerWithTouchDispatcher，在该方法中修改事件响应级别，默认级别为 0，值越大响应级别越小，所以设定 CustomMenu 的级别为 1 即可。

`addTargetedDelegate` 方法的第二个参数设定响应级别。

```
void CustomMenu::registerWithTouchDispatcher() {
    Director::getInstance()->getTouchDispatcher()->
addTargetedDelegate (this, 1, false);
}
```

(3) 在 `HelloWorldScene.cpp` 中引入 `CustomMenu` 的头文件:

```
#include "CustomMenu.h"
```

(4) 在 `tableCellAtIndex` 方法中修改菜单生成代码为:

```
CustomMenu *menu = CustomMenu::create();
```

(5) 运行程序, 效果如图 4-16 所示, 但当拖动菜单时, `TableView` 也可以滚动。

4.4 扩展控件

本节讲解下 Cocos2d-x 提供的扩展控件的使用方式。

4.4.1 滑动条控件 `ControlSlider`

滑动条控件常用来设置一个连续值的调整, 比如音量调整等。`ControlSlider` 继承自 `Control` 类。下面就介绍 `ControlSlider` 的使用方式, 完整代码请查看 4-4-1。

(1) 创建一个 `ControlSlider` 对象, Cocos2d-x 提供了一个 `create` 函数, 原型为 `ControlSlider::create(const char *bgFile, const char *progressFile, const char *thumbFile)`, 其中 `bgFile` 为滑动条背景, `progressFile` 为滑动条进度, `thumbFile` 为滑动条的拖曳图标。代码为:

```
ControlSlider *slider = ControlSlider::create("sliderTrack.png",
"sliderProgress.png", "sliderThumb.png");
```

(2) 设置滑动条的最大值和最小值:

```
slider->setMinimumValue(0.0f);
slider->setMaximumValue(5.0f);
```

(3) 设置 slider 的位置，并添加到场景中：

```
slider->setAnchorPoint(Point(0.5f, 1.0f));
slider->setPosition(Point(screenSize.width / 2.0f, screenSize.height
/ 2.0f + 16));
slider->setTag(1);
addChild(slider);
```

(4) 同理，创建另外一个滑动条 **restrictSlider**，并添加到游戏场景中，但是设置该滑动条可使用的最大值和最小值：

```
restrictSlider->setMaximumAllowedValue(4.0f);
restrictSlider->setMinimumAllowedValue(1.5f);
```

(5) 添加一个文本 **label**，用来显示当前值的变化：

```
_displayValueLabel = LabelTTF::create("Move the slider thumb!\nThe
lower slider is restricted." , "Marker Felt", 32);
_displayValueLabel->retain();
_displayValueLabel->setAnchorPoint(Point(0.5f, -1.0f));
_displayValueLabel->setPosition(Point(screenSize.width / 1.7f,
screenSize.height / 2.0f));
addChild(_displayValueLabel);
```

(6) 为两个滑动条添加值变化事件监听器 **valueChanged**，当值发生变化时就会触发该函数：

```
slider->addTargetWithActionForControlEvents(this,
cccontrol_selector (HelloWorld::valueChanged),
Control::EventType::VALUE_CHANGED);
restrictSlider->addTargetWithActionForControlEvents(this,
cccontrol_selector (HelloWorld::valueChanged),
Control::EventType::VALUE_CHANGED);
```

addTargetWithActionForControlEvents 函数继承自 **Control** 类，作用是给一个对象注册某种事件的监听器。它的原型是：

```
void addTargetWithActionForControlEvents ( Ref * target,
Handler action,
EventType controlEvent
)
```

其中 **target** 是监听事件的对象，**action** 是事件监听器，**controlEvent** 是事件类型。

这里监听的是 **VALUE_CHANGED** 事件，Cocos2d-x 的控件类也提供了其他事件，类似 Cocoa 框架的事件定义，详细类型请参考表 4-3。

表 4-3 Control 事件类型表

事件名称	事件解释
TOUCH_DOWN	手指按下事件
DRAG_INSIDE	手指在控件内部拖曳事件
DRAG_OUTSIDE	手指在控件外部拖曳事件
DRAG_ENTER	手指从控件外部滑动进入控件内部事件
DRAG_EXIT	手指从控件内部滑动进入控件外部事件
TOUCH_UP_INSIDE	手指在控件内部松开事件
TOUCH_UP_OUTSIDE	手指在控件外部松开事件
TOUCH_CANCEL	取消当前触摸事件的系统事件
VALUE_CHANGED	值发生变化时触发的控件

(7) 实现 **valueChanged** 方法，当滑动条的值变化时，用 **_displayValueLabel** 显示变化后的值：

```
void HelloWorld::valueChanged(Ref *sender, Control::EventType
controlEvent)
{
    ControlSlider* pSlider = (ControlSlider*)sender;
    // Change value of label.
    if(pSlider->getTag() == 1)
        _displayValueLabel->setString(String::createWithFormat
("Upper slider value = %.02f", pSlider->getValue())->getCString());
    if(pSlider->getTag() == 2)
        _displayValueLabel->setString(String::createWithFormat
("Lower slider value = %.02f", pSlider->getValue())->getCString());
}
```

(8) 运行程序，可以看到结果如图 4-17 所示，拖动圆形图标，就能看到值的变化，且拖动下面那个滑动条时，只能在 1.5~4 之间取值。

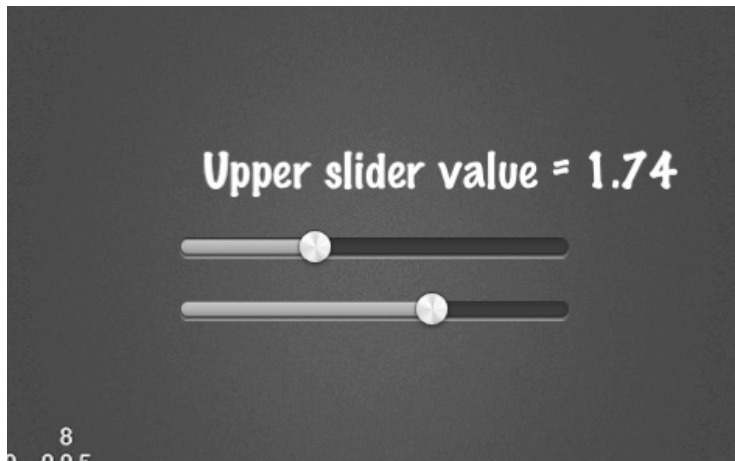


图 4-17 ControlSlider 使用示例

4.4.2 开关控件 ControlSwitch

开关在手机中是经常用到的功能之一，比如启用/关闭蓝牙，启用/关闭 WIFI。Cocos2d-x 也提供了一个开关控件 ControlSwitch。ControlSwitch 继承自 Control 类。下面就讲解如何使用 ControlSwitch 控件，详细代码请看代码清单 4-4-2。

(1) 创建一个 ControlSwitch 对象，并添加到游戏画面中：

```
ControlSwitch *switchControl = ControlSwitch::create
(
    Sprite::create("switch-mask.png"),
    Sprite::create("switch-on.png"),
    Sprite::create("switch-off.png"),
    Sprite::create("switch-thumb.png"),
    LabelTTF::create("On", "Arial-BoldMT", 16),
    LabelTTF::create("Off", "Arial-BoldMT", 16)
);
switchControl->setPosition(Point(300, screenSize.height/2));
addChild(switchControl);
```

该 create 方法的原型为：

```
ControlSwitch::create(Sprite *maskSprite, Sprite *onSprite, Sprite
```



```
*offSprite, Sprite *thumbSprite, LabelTTF *onLabel, LabelTTF *offLabel)
```

`maskSprite` 是开关的背景, `onSprite` 是开关打开时的状态, `offSprite` 是开关关闭时的状态, `thumbSprite` 表示当前状态的图标, `onLabel` 是开关打开时显示的文本, `offLabel` 是开关关闭时的状态。

(2) 添加一个 `LabelTTF` 文本标签 `_displayValueLabel`, 用来显示当前开关的状态。我们把该标签声明成类的私有成员, 便于在多个函数中使用, 并给该文本标签加一个背景, 方便查看:

```
Scale9Sprite *background = Scale9Sprite::create("buttonBackground.png");
background->setContentSize(Size(80, 50));
background->setPosition(Point(200, screenSize.height/2));
addChild(background);
_displayValueLabel = LabelTTF::create("#color", "Marker Felt", 30);
_displayValueLabel->retain();
_displayValueLabel->setPosition(background->getPosition());
addChild(_displayValueLabel);
```

(3) 为开关设置值变化监听函数 `valueChanged`, 当开关的值发生变化时会触发该函数:

```
switchControl->addTargetWithActionForControlEvents(this, cccontrol_selector(HelloWorld::valueChanged), Control::EventType::VALUE_CHANGED);
```

(4) 实现 `valueChanged` 函数, 先获取 `ControlSwitch` 对象, 然后用 `isON` 函数获取当前状态, 如果开关控件是 `on` 状态, 返回 `true`, 否则返回 `false`:

```
ControlSwitch* pSwitch = (ControlSwitch*)sender;
if (pSwitch->isOn())
{
    _displayValueLabel->setString("On");
}
else
{
    _displayValueLabel->setString("Off");
}
```

(5) 运行程序, 就能看到开关控件, 单击该开关, 会切换状态, 并且文本标签

同时显示当前状态，如图 4-18 所示。

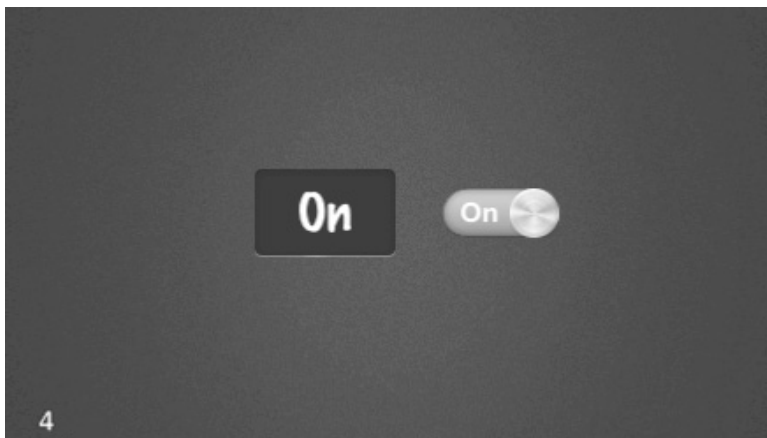


图 4-18 ControlSwitch 控件使用示例

4.4.3 取色器控件 ControlColourPicker

使用过 Photoshop 的同学们可能知道有一个拾色器，可以可视化地获取自己想要颜色的色值。Cocos2d-x 也提供了一个类似的控件 ControlColourPicker。ControlColourPicker 继承自 Control 类。下面就讲解如何使用 ControlColourPicker 控件，详细代码请看代码清单 4-4-3。

(1) 创建 ControlColourPicker 对象：

```
ControlColourPicker *colourPicker = ControlColourPicker::create();
```

该 create 方法创建一个取色器对象，在创建对象时，会使用 SpriteSheet 把一组图片添加到内存中，如果提示错误，请查看该控件源码修改图片地址，或者添加图片。

(2) 使用 setColor 函数设置 ControlColourPicker 的初始值，并添加到场景中：

```
colourPicker->setColor(Color3B(37, 46, 252));  
colourPicker->setPosition(Point (200, screenSize.height/2));  
addChild(colourPicker);
```

(3) 添加一个 LabelTTF 文本标签_colorLabel，用来显示当前颜色：

```
_colorLabel = LabelTTF::create("#color", "Marker Felt", 30);
_colorLabel->retain();
_colorLabel->setPosition(Point(350, screenSize.height/2));
addChild(_colorLabel);
```

(4) 为该控件注册事件监听器 valueChanged，当颜色值发生变化时触发该监听器：

```
colourPicker->addTargetWithActionForControlEvents(this, cccontrol_
selector(HelloWorld::valueChanged), Control::EventType::VALUE_CHANGED);
```

(5) 实现监听器 valueChanged，当调用该函数时，文本标签显示当前颜色值：

```
ControlColourPicker* pPicker = (ControlColourPicker*)sender;
_colorLabel->setString(String::createWithFormat("#%02X%02X%02X",
pPicker->getColor().r, pPicker->getColor().g, pPicker->getColor().b)->
getCString());
```

getColor 函数获取当前颜色值，是继承者 LayerRGBA 的函数。颜色值格式为 Color3B，它包括 R、G、B 三个值，分别通过.r、.g、.b 获取。

(6) 运行程序就能看到 ControlColourPicker 的真容了，如图 4-19 所示。

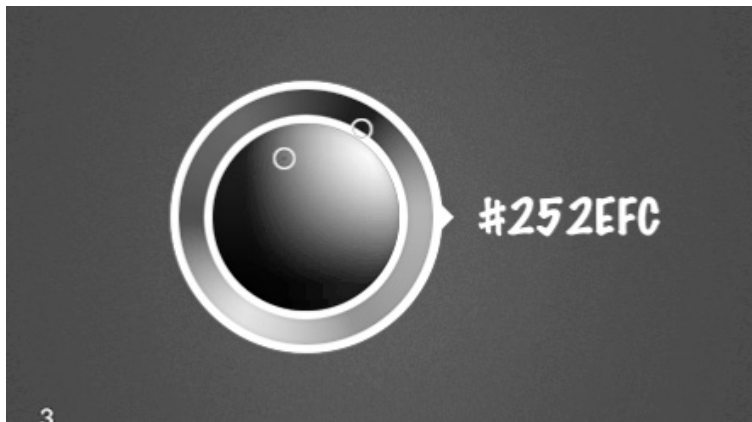


图 4-19 ControlColourPicker 控件使用示例

4.4.4 电位计控件 ControlPotentiometer

电位计控件通过拉动可旋转的按钮，从而改变所代表的值，但它更像圆形音量按钮。**ControlPotentiometer**继承自**Control**类。下面就讲解如何使用**ControlPotentiometer**控件。详细代码请看代码清单 4-4-4。

(1) 创建 **ControlPotentiometer** 对象，并添加到场景中：

```
ControlPotentiometer *potentiometer = ControlPotentiometer::create(
    "potentiometerTrack.png",
    "potentiometerProgress.png",
    "potentiometerButton.png");
    potentiometer->setPosition(Point(200, screenSize.height/2));
    addChild(potentiometer);
```

使用 **create** 方法创建一个电位计控件对象。该 **create** 方法原型是：

```
static ControlPotentiometer * create (const char *backgroundFile,
const char *progressFile, const char *thumbFile)
```

backgroundFile 是背景图片，**progressFile** 是当前值的图片，**thumbFile** 是触摸控制图片。

(2) 创建一个文本标签，用来显示当前值：

```
_displayValueLabel = LabelTTF::create("#color", "Marker Felt", 30);
_displayValueLabel->retain();
_displayValueLabel->setPosition(Point(300, screenSize.height/2));
addChild(_displayValueLabel);
```

(3) 为该控件添加事件监听器 **valueChanged**：

```
potentiometer->addTargetWithActionForControlEvents(this, cccontrol_
selector(HelloWorld::valueChanged), Control::EventType::VALUE_CHANGED);
```

(4) 实现 **valueChanged**，当改变电位计的值时，实时显示当前值的变化：

```
ControlPotentiometer* pControl = (ControlPotentiometer*) sender;
_displayValueLabel->setString(String::createWithFormat("%.02f",
pControl->getValue())->getCString());
```

(5) 运行程序即可看到运行效果如图 4-20 所示。

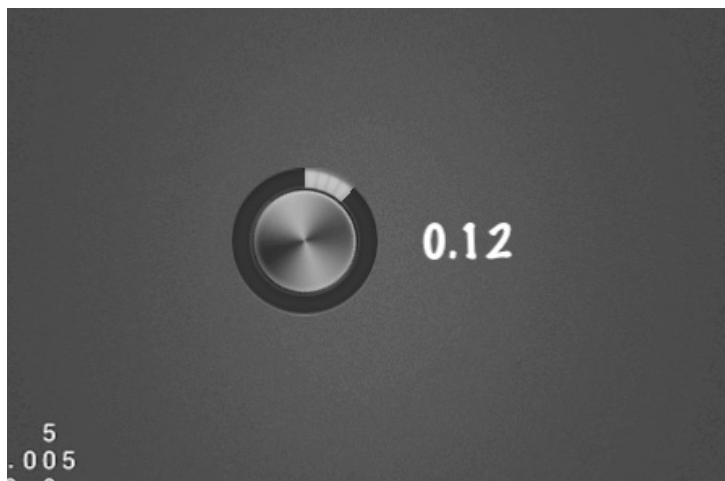


图 4-20 ControlPotentiometer 控件使用示例

4.4.5 步进器控件 ControlStepper

步进器控件 **ControlStepper** 类似购物网站中购物车里增加或者减少商品数量的功能，当单击加号时，数值就会加一，当单击减号时，数值就会减一。**ControlStepper** 继承自 **Control** 类。下面就讲解如何使用 **ControlStepper** 控件。详细代码请看代码清单 4-4-5。

(1) 创建 **ControlStepper** 对象，并添加到场景中：

```
Sprite *minusSprite      = Sprite::create("stepper-minus.png");
Sprite *plusSprite       = Sprite::create("stepper-plus.png");
ControlStepper *stepper  = ControlStepper::create(minusSprite,
plusSprite);;
stepper->setPosition(Point(200, screenSize.height/2));
addChild(stepper);
```

使用 **create** 方法创建一个步进器控件对象。该 **create** 方法原型是：

```
static ControlStepper * create (Sprite *minusSprite, Sprite
*plusSprite)
```

其中 **minusSprite** 是减号精灵，**plusSprite** 是加号精灵。

(2) 创建一个文本标签，用来显示当前值：

```
_displayValueLabel = LabelTTF::create("#color", "Marker Felt", 30);
_displayValueLabel->retain();
_displayValueLabel->setPosition(Point(300, screenSize.height/2));
addChild(_displayValueLabel);
```

(3) 为该控件添加事件监听器 `valueChanged`：

```
stepper->addTargetWithActionForControlEvents(this, cccontrol_
selector (HelloWorld::valueChanged), Control::EventType::VALUE_CHANGED);
```

(4) 实现 `valueChanged`，当改变电位计的值时，实时显示当前值的变化：

```
ControlStepper* pControl = (ControlStepper*)sender;
_displayValueLabel-> setString(String::createWithFormat("%0.02f",
(float)pControl->getValue())-> getCString());
```

(5) 运行程序即可看到运行效果如图 4-21 所示。

4.4.6 按钮控件 `ControlButton`

按钮控件是最简单也是最常用的控件，它提供了比菜单更灵活的使用方式，并且可以对上文提到的事件设置不同的响应方式。`ControlButton` 继承自 `Control` 类。下面就讲解如何使用 `ControlButton` 控件。详细代码请看代码清单 4-4-5。

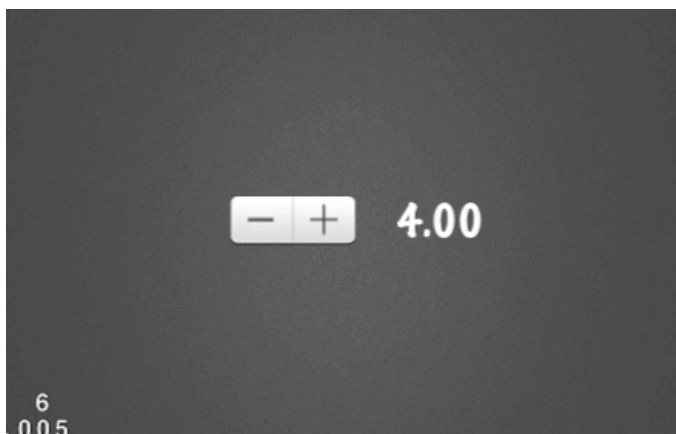


图 4-21 `ControlStepper` 控件使用示例

(1) 创建两个精灵 `backgroundButton` 和 `backgroundHighlightedButton`，分别表示按钮默认和被单击时的背景：

```
Scale9Sprite *backgroundButton =
Scale9Sprite::create("button.png");
Scale9Sprite *backgroundHighlightedButton = Scale9Sprite::create
("buttonHighlighted.png");
```

(2) 创建按钮显示的文本：

```
LabelTTF *titleButton = LabelTTF::create("Touch Me!", "Marker Felt",
30);
titleButton->setColor(Color3B(159, 168, 176));
```

(3) 使用 `create` 方法创建按钮对象，并添加到游戏场景中：

```
ControlButton *controlButton = ControlButton::create(titleButton,
backgroundButton);
controlButton->setAnchorPoint(Point(0.5f, 1));
controlButton->setPosition(Point(screenSize.width / 2.0f,
screenSize.height / 2.0f));
addChild(controlButton, 1);
```

使用 `create` 方法创建一个按钮控件对象。该 `create` 方法原型是：

```
static ControlButton* create(Node * label,
Scale9Sprite * backgroundSprite
)
```

`label` 表示按钮显示时的文本，`backgroundSprite` 是默认状态下按钮的背景。Cocos2d-x 还提供了其他 `create` 方法，这里就不举例说明。

(4) 使用 `setBackgroundSpriteForState` 设置按钮被触摸时的背景，用 `setTitleColorForState` 设置触摸时文字的样式：

```
controlButton->setBackgroundSpriteForState(backgroundHighlighted
Button, Control::State::HIGH_LIGHTED);
controlButton->setTitleColorForState(Color3B::WHITE,
Control::State::HIGH_LIGHTED);
```

这两个函数的第二个参数都是控件的状态，Cocos2d-x 的控件状态有 4 中，详细请看表 4-4。

表 4-4 控件状态表

控件状态	控件状态说明
NORMAL	默认状态，控件能单击但没被单击或高亮
HIGH_LIGHTED	高亮状态，当在控件内部按下、拖曳或从外部拖曳进控件内部会激活该状态
DISABLED	控件被禁止响应事件
SELECTED	控件被选中事件

(5) 添加一个文本标签，用来显示控件的变化：

```
_displayValueLabel = LabelTTF::create("", "Marker Felt", 30);
_displayValueLabel->setAnchorPoint(Point(0.5f, -1));
_displayValueLabel->setPosition(Point(screenSize.width / 2.0f,
screenSize.height / 2.0f));
addChild(_displayValueLabel, 1);
```

(6) 为控件的事件添加事件监听器：

```
controlButton->addTargetWithActionForControlEvents(this, cccontrol_
selector(HelloWorld::touchDownAction), Control::EventType::TOUCH_DOWN);
controlButton->addTargetWithActionForControlEvents(this,
cccontrol_selector(HelloWorld::touchDragInsideAction), Control:: EventType::
DRAG_INSIDE);
controlButton->addTargetWithActionForControlEvents(this,
cccontrol_selector(HelloWorld::touchDragOutsideAction), Control:: EventType::
DRAG_OUTSIDE);
controlButton->addTargetWithActionForControlEvents(this,
cccontrol_selector(HelloWorld::touchDragEnterAction), Control:: EventType::
DRAG_ENTER);
controlButton->addTargetWithActionForControlEvents(this,
cccontrol_selector(HelloWorld::touchDragExitAction), Control:: EventType::
DRAG_EXIT);
controlButton->addTargetWithActionForControlEvents(this,
cccontrol_selector(HelloWorld::touchUpInsideAction), Control::EventType::
TOUCH_UP_INSIDE);
controlButton->addTargetWithActionForControlEvents (this,
cccontrol_selector (HelloWorld::touchUpOutsideAction), Control:: EventType::
TOUCH_UP_OUTSIDE);
controlButton->addTargetWithActionForControlEvents(this, cccontrol_
selector(HelloWorld::touchCancelAction),
```



```
Control::EventType::TOUCH_CANCEL);
```

(7) 实现上面 TOUCH_DOWN 事件监听器，其他方法依此类推：

```
void HelloWorld::touchDownAction(Ref *senderz, Control::EventType
controlEvent)
{
    _displayValueLabel->setString(String::createWithFormat("Touch
Down")->getCString());
}
```

(8) 运行程序，测试按钮的点击状态如图 4-22 所示。



图 4-22 按钮控件使用示例

4.4.7 Scale9Sprite

在设计 UI 界面时，经常遇到对同样的设计进行不等比缩放，这样就出现压缩变形的情况。Scale9Sprite 针对这种情况提供了一种解决方式，它允许你对图片的某个地方进行缩放，而其他地方保持原样，从而保持不变形。

Scale9Sprite 把精灵分成 3×3 共 9 个区域，你可以定义从边框向内部多少尺寸的内部不进行缩放，例如当有个圆角按钮时，可以指定圆角不被缩放，如图 4-23 所示。

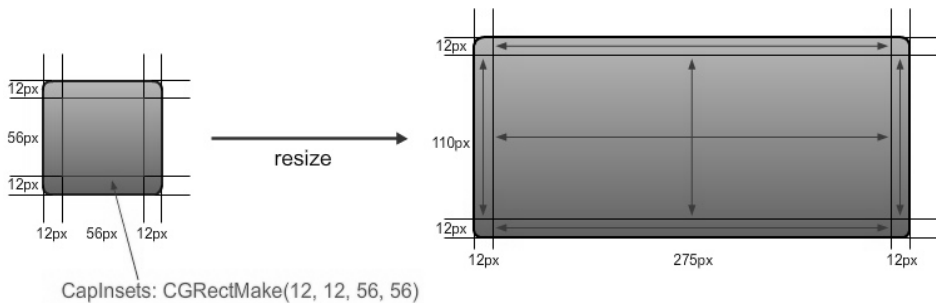


图 4-23 指定圆角不被缩放

下面的代码可以添加一个 `Scale9Sprite` 对象。

```
cocos2d::extension::Scale9Sprite *ribbon = cocos2d::extension::
Scale9Sprite::create("ribbon.png", RectMake(12, 12, 56, 56));
ribbon->setContentSize(Size(screenSize.width, 57));
addChild(ribbon);
```

在这段代码中定义了一个矩形，该矩形把精灵分成 9 个区域，从而确定哪些可以缩放，哪些不可以缩放。该例中定义了 4 个角不被缩放，左边和右边只有高度可以变，上边和下边只有宽度可以变，中间部分高度和宽度都可以变。

4.5 使用编辑框制作用户登录界面

编辑框是游戏中经常用到的功能，比如注册和登录时需要输入用户名和密码，Cocos2d-x 使用 `EditBox` 添加编辑框，`EditBox` 继承自 `ControlButton`，下面就介绍如何用 `EditBox` 制作一个简单的登录界面。详细代码请参考代码清单 4-5。

(1) 创建一个编辑框对象 `_editName`，用来输入姓名：

```
_editName = EditBox::create(editBoxSize, Scale9Sprite::create(
"green_edit.png"));
```

`create` 的原型为：

```
static EditBox* create ( const Size & size,
Scale9Sprite * pNormal9SpriteBg,
```

```
Scale9Sprite * pPressed9SpriteBg = NULL,
Scale9Sprite * pDisabled9SpriteBg = NULL
)
```

其中 `size` 为编辑框大小，`pNormal9SpriteBg` 是编辑框的默认背景，`pPressed9SpriteBg` 是编辑框被选中的状态，`pDisabled9SpriteBg` 是编辑框被禁止输入时的状态。后面两个参数默认为 `NULL`。

(2) 设置 `_editName` 的提示文字和提示文字的颜色。当在输入框内输入内容时，该提示文字会消失。

```
_editName->setPlaceholder("Name:");
_editName->setPlaceholderFontColor(Color3B::WHITE);
```

(3) 设置编辑框 `_editName` 内容的最大长度为 8:

```
_editName->setMaxLength(8);
```

(4) 设置结束输入时键盘的类型:

```
_editName->setReturnType(TextBox::KeyboardReturnType::DONE);
```

(5) 设置事件代理为当前对象:

```
_editName->setDelegate(this);
```

(6) 设置输入框文字的大小和颜色，并添加到游戏场景中:

```
_editName->setPosition(Point(screenSize.width/2,200));
_editName->setFontSize(25);
_editName->setFontColor(Color3B::RED);
addChild(_editName);
```

(7) 同理在场景中添加另外一个输入框 `_editPassword`，用来输入密码。密码输入框的内容不能用明文显示，必须用点代替。

```
_editPassword->setInputFlag(TextBox::InputFlag::PASSWORD);
```

`setInputFlag` 用来设置文字的显示形式。Cocos2d-x 提供了 5 种形式，详细情形请参见表 4-5。

表 4-5 输入框内容显示形式

输入框内容显示形式	解 释
PASSWORD	无论任何时候，文字都要隐藏显示，常用点代替文字
SENSITIVE	敏感文字，不能把内容保存到字典或表中
INITIAL_CAPS_WORD	文字每个单词的第一个字母大写
INITIAL_CAPS_SENTENCE	每一句话的第一个文字大写
INITIAL_CAPS_ALL_CHARACTERS	自动使所有字母大写

(8) 设置密码输入框的文字不能有换行符。Cocos2d-x 还提供了其他几种输入内容的控制，参考表 4-6。

```
_editPassword->setInputMode(EditBox::InputMode::SINGLE_LINE);
```

表 4-6 输入内容控制

输入内容名称	解 释
ANY	可以输入任何内容
EMAIL_ADDRESS	只允许输入电子邮件地址
NUMERIC	只允许输入数字
PHONE_NUMBER	只允许输入手机号码
URL	只允许输入网址
DECIMAL	只允许输入小数
SINGLE_LINE	只允许输入单行内容

(9) 在场景添加一个文本标签，用来显示内容变化：

```
_displayValueLabel = LabelTTF::create("", "Marker Felt", 30);
_displayValueLabel->setAnchorPoint(Point(0.5f, -1));
_displayValueLabel->setPosition(Point(screenSize.width / 2.0f,
screenSize.height-100));
addChild(_displayValueLabel, 1);
```

(10) 给输入框添加事件代理，事件代理函数在 `EditBoxDelegate` 中定义，所以自己定义的类要继承 `EditBoxDelegate`。其中包括 4 个处理函数。

◎ `editBoxEditingDidBegin`：当输入框聚焦时触发。

- ◎ **editBoxEditingDidEnd**: 当输入框失去焦点时触发。
- ◎ **editBoxTextChanged**: 当输入框内容发生变化时触发。
- ◎ **editBoxReturn**: 当按键盘的完成键时触发。

(11) 实现 **editBoxEditingDidBegin** 方法, 当一个输入框获得焦点时, 用文本标签显示是哪个文本框:

```
if (_editName == editBox)
{
    _displayValueLabel->setString("Name EditBox DidBegin !");
}
else if (_editPassword == editBox)
{
    _displayValueLabel->setString("Password EditBox DidBegin !");
}
```

(12) 实现 **editBoxEditingDidEnd** 方法, 当一个输入框失去焦点时, 用文本标签显示是哪个文本框:

```
if (_editName == editBox)
{
    _displayValueLabel->setString("Name EditBox DidEnd !");
}
else if (_editPassword == editBox)
{
    _displayValueLabel->setString("Password EditBox DidEnd !");
}
```

(13) 实现 **editBoxTextChanged** 方法, 当输入框的内容发生变化时会触发该函数, 并用文本标签显示变化后的内容:

```
if (_editName == editBox)
{
    _displayValueLabel->setString(String::createWithFormat(
        "EditBox Name TextChanged, text: %s ", text.c_str())->getCString());
}
else if (_editPassword == editBox)
{
    _displayValueLabel->setString(String::createWithFormat
```

```
("EditBox Password TextChanged, text: %s ",text.c_str()->getCString());  
}
```

(14) 实现 `editBoxReturn` 方法，当单击虚拟键盘的“完成”键时触发，用文本标签显示是哪个输入框：

```
if (_editName == editBox)  
{  
    _displayValueLabel->setString("Name EditBox return !");  
}  
else if (_editPassword == editBox)  
{  
    _displayValueLabel->setString("Password EditBox return !");  
}
```

(15) 运行程序，输入文字，体验下效果吧，如图 4-24 所示。

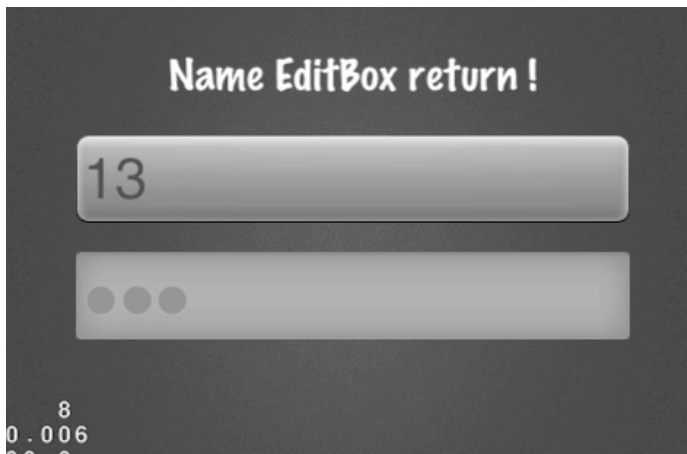


图 4-24 用输入框制作简单的登录界面

5

第 5 章

Cocos2d-x 动作

动作在游戏中是最常用、最普遍的元素。几乎任何一个游戏界面都会涉及动作、动画。本章就介绍 Cocos2d-x 中与动作相关的实例，包括动作分类、动作回调、动作同时进行、动作一直进行等。

5.1 动作分类

Cocos2d-x 中动作的基类是 `Action`，所有的动作都继承自该类。`Action` 提供了一些动作共有的方法，如下所示。

◎ `virtual Action * reverse ()`：表示动作逆向执行。

◎ virtual bool isDone (void) const: 动作是否结束。

◎ virtual void stop (void): 停止动作。

根据动作执行需要的时间, 分为瞬时动作和延时动作。

◎ 瞬时动作的主要特点是动作的执行不需要花费时间, 马上就能完成动作的执行。这些动作的基类是 **ActionInstant**, 比如 **Flix**、**Flixy**、**Hide**、**Show** 等。

◎ 延时动作的主要特点是动作的执行需要花费时间, 这些动作的基类是 **ActionInterval**, 比如 **MoveTo**、**JumpTo**、**TintTo**、**RotateBy** 等。

5.2 瞬时动作

瞬时动作就是瞬时完成的动作, 不需要设定执行时间。本节介绍如何使用 Cocos2d-x 常用的几个瞬时动作。

5.2.1 使用 FlipX/FlipY 实现 X/Y 翻转

FlipX 是水平翻转 180°, FlipY 是垂直翻转 180°。接下来讲解下如何使用 FlipX 和 FlipY, 详细代码请查看代码清单 5-2。

(1) 在场景中添加一个角色, 并放置在屏幕中间:

```
hero = Sprite::create("hero.png");
hero->setPosition(Point(screenSize.width/2, screenSize.height/2));
addChild(hero);
```

(2) 创建两个菜单 itemFlipX 和 itemFlipY, 分别用来触发点击回调函数 testFlipX 和 testFlipY:

```
MenuItemFont *itemFlipX = MenuItemFont::create("FlipX", CC_CALLBACK_1
(HelloWorld::testFlipX, this));
    itemFlipX->setPosition(Point(100, 50));
    MenuItemFont *itemFlipY = MenuItemFont::create("FlipY",
CC_CALLBACK_1 (HelloWorld::testFlipY, this));
    itemFlipY->setPosition(Point(200, 50));
```



```
Menu *actionMenu = Menu::create(itemFlipX,itemFlipY,NULL);
actionMenu->setPosition(Point(0,0));
addChild(actionMenu);
```

(3) 实现回调函数 `testFlipX`。在该函数中，使用 `isFlipX` 函数判断对象是否水平翻转了，如果没有（`isFlipX` 返回 `false`），就水平翻转对象，如果是（`isFlipX` 返回 `true`），就翻转成原来的样子。

```
if (hero->isFlipX()) {
    hero->runAction(FlipX::create(false));
}else{
    hero->runAction(FlipX::create(true));
}
```

这里使用了 `FlipX` 的 `create` 方法创建对象，`create` 方法的原型为：

```
static FlipX* create(bool x)
```

参数 `x` 为布尔型，如果参数为 `true`，则水平翻转，如果为 `false`，则为图片本身形状。

同理，添加实现 `testFlipY` 的代码，作用和 `testFlipX` 相似，只是水平翻转变成垂直翻转。

(4) 运行程序，单击 `FlipX` 和 `FlipY` 菜单，就能看到翻转效果，如图 5-1 所示。



图 5-1 `FlipX` 和 `FlipY` 示例

水平和垂直翻转的另外一种实现方式是调用精灵对象的 `setFlipX` 方法和 `setFlipY` 方法。`setFlipX` 的原型为：

```
void setFlipX (bool flippedX);
```

参数 `flippedX` 为布尔型，如果参数为 `true`，则水平翻转，如果为 `false`，则为图片本身形状。比如上面 `testFlipX` 的实现可以改成为：

```
if (hero->isFlipX()) {
    hero->setFlipX(false);
}else{
    hero-> setFlipX (true);
}
```

同理，`setFlipY` 跟 `setFlipX` 的使用方式一样，只是水平翻转变成垂直翻转。

5.2.2 使用 Hide、Show 实现隐藏和显示

看到 `Hide` 和 `Show`，就能知道它们是用来显示和隐藏对象的。这节示例基于 5.2.1 节，在它的基础上添加 `Hide` 和 `Show` 的功能。

(1) 创建两个菜单 `itemHide` 和 `itemShow`，分别用来触发单击回调函数 `testHide` 和 `testShow`。

(2) 实现 `testHide` 方法，当调用该方法时，隐藏对象：

```
hero->runAction(Hide::create());
```

这里先用 `create` 方法创建 `Hide` 对象，然后让精灵运行该行动，该 `create` 方法没有参数。

(3) 同理，实现 `testShow` 方法：

```
hero->runAction>Show::create());
```

(4) 运行程序，当单击 `Hide` 菜单时隐藏精灵对象，当单击 `Show` 菜单时显示精灵对象。

5.3 延时动作

顾名思义，延时动作就是在规定的时间内完成的动作。也就是说，创建延时动作时，要设定一个完成时间。Cocos2d-x 提供了很多延时动作，下面就讲解几个常用动作的使用方法。

5.3.1 使用 MoveTo 或者 MoveBy 实现移动

MoveTo 和 MoveBy 用来在屏幕中移动一个精灵，MoveTo 是把精灵移动到某个位置，MoveBy 是移动精灵一段距离。下面介绍如何在游戏场景中移动精灵。完整代码请查看代码清单 5-3。

(1) 添加两个精灵到场景中：

```
hero1 = Sprite::create("hero.png");
hero1->setPosition(Point(screenSize.width/4,
screenSize.height/2));
addChild(hero1);
hero2 = Sprite::create("hero2.png");
hero2->setPosition(Point(screenSize.width/4*3,
screenSize.height/2));
addChild(hero2);
```

(2) 在场景中添加 Move 菜单，当单击该菜单时调用 testMove 函数，在该函数里实现移动精灵的功能：

```
void HelloWorld::testMove(Ref* sender)
{
    hero1->setPosition(Point(screenSize.width/4,
screenSize.height/2));
    hero2->setPosition(Point(screenSize.width/4*3,
screenSize.height/2));
    //MoveTo 测试
    MoveTo *mt = MoveTo::create(0.2, Point(screenSize.width/4*3,
screenSize.height/2));
```

```
hero1->runAction(mt);  
//MoveBy 测试  
MoveBy *my = MoveBy::create(0.2, Point(-screenSize.width/2,0));  
hero2->runAction(my);  
}
```

该函数首先把两个精灵归位到原来的位置，这样可以多次查看移动效果。然后使用 `create` 方法创建 `MoveTo` 对象，`create` 方法的原型为：

```
static MoveTo * create (float duration, const Point &position);
```

参数解释如下。

- ◎ **duration**: 移动需要的时间。
- ◎ **position**: 是移动到的点，也就是目的地。

让 `hero1` 执行 `MoveTo` 动作。

同理，创建 `MoveBy` 对象，`create` 方法的原型为：

```
static MoveBy * create (float duration, const Point &deltaPosition)
```

参数解释如下。

- ◎ **duration**: 移动需要的时间。
- ◎ **position**: 移动的距离，不是移动的目的，比如该值为 (10,-10)，初始位置是 (50,50)，移动后的位置就是 (60,40)。

让 `hero2` 执行 `MoveBy` 动作。

(3) 运行程序，单击 `Move` 菜单查看效果，如图 5-2 所示。

5.3.2 使用 `RotateTo` 和 `RotateBy` 实现旋转

`RotateTo` 和 `RotateBy` 用来旋转一个精灵或者其他对象，同移动精灵相似，`RotateTo` 是把精灵旋转到某个角度，`RotateBy` 是把精灵旋转多少角度。下面介绍如何在游戏场景中旋转精灵。完整代码请查看清单 5-3。

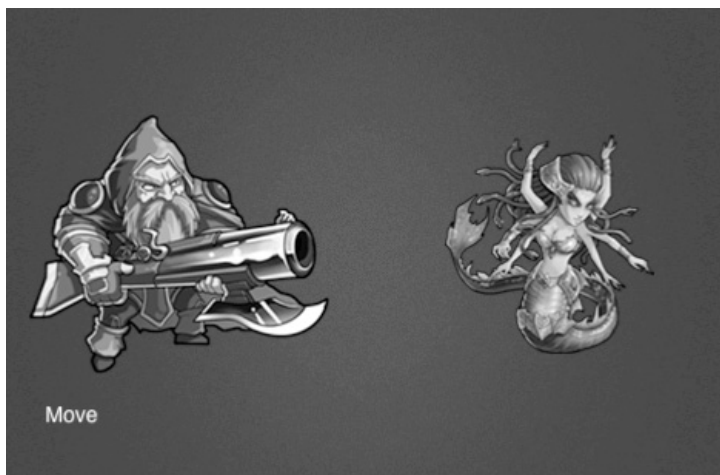


图 5-2 MoveTo 和 MoveBy 测试

(1) 添加旋转菜单 Rotate，当单击该菜单，触发 testRotate 回调函数：

```
LabelTTF *rotateLabel= LabelTTF::create("Rotate", "Helvetica", 14);
MenuItemLabel *itemRotate = MenuItemLabel::create(rotateLabel,
CC_CALLBACK_1 (HelloWorld::testRotate, this));
itemRotate->setPosition(Point(100, 50));
actionMenu->addChild(itemRotate);
```

(2) 实现 testRotate 函数，创建 RotateTo 对象 rt，并让 hero1 执行该动作；创建 RotateBy 对象 rb，并让 hero2 执行该动作：

```
RotateTo *rt = RotateTo::create(0.2, 100,100);
RotateBy *rb = RotateBy::create(0.2, 50, 100);

hero1->runAction(rt);
hero2->runAction(rb);
```

使用 create 方法创建 RotateTo 对象，方法原型为：

```
static RotateTo * create (float fDuration, float fDeltaAngleX, float
fDeltaAngleY)
```

参数解释如下。

◎ fDuration：动作执行时间。

◎ **fDeltaAngleX**: 沿 X 轴旋转的角度, X 轴是所看到平面的水平方向。

◎ **fDeltaAngleY**: 沿 Y 轴旋转的角度, Y 轴是所看到平面的垂直方向。

该例子中的旋转可以用另外一个 **create** 方法创建, 原型为:

```
static RotateTo * create (float fDuration, float fDeltaAngle)
```

参数解释如下。

◎ **fDuration**: 动作执行时间。

◎ **fDeltaAngle**: 旋转的角度, 同时沿 X 、 Y 轴旋转指定的角度。

即:

```
RotateTo *rt = RotateTo::create(0.2,100);
```

同理, **RotateBy** 的创建方法跟 **RotateTo** 类似, 需要注意的是, **RotateBy** 是旋转的角度, 基于现有的角度再旋转。

(3) 运行程序, 单击 **Rotate**, 就可以看到精灵的旋转效果, 只有一个旋转一次, 是用 **RotateTo** 实现的, 另一个不停旋转, 是用 **RotateBy** 实现的, 如图 5-3 所示。

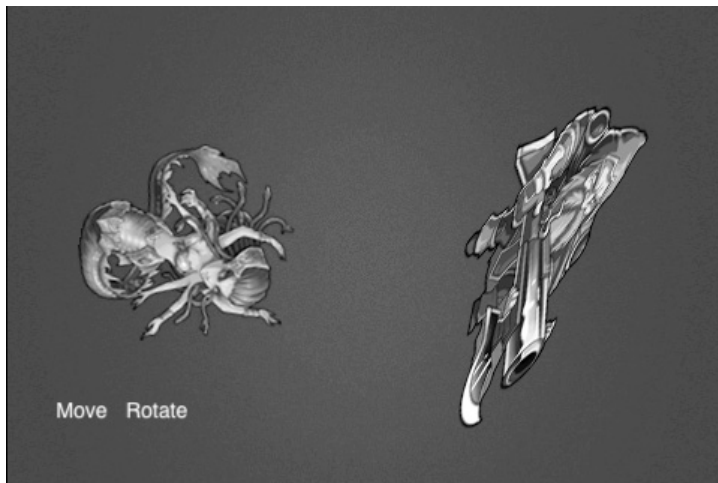


图 5-3 RotateTo 和 RotateBy 示例

5.3.3 使用 JumpTo 和 JumpBy 实现跳跃

JumpTo 和 JumpBy 使精灵或者其他对象跳越, JumpTo 让精灵跳跃到某个地方, JumpBy 让精灵跳跃多远距离。下面介绍如何在游戏场景中使精灵跳跃。完整代码请查看清单 5-3。

(1) 添加跳跃菜单 Jump, 当单击该菜单, 触发 testJump 回调函数:

```
LabelTTF *jumpLabel= LabelTTF::create("Jump", "Helvetica", 14);
MenuItemLabel *jumpRotate = MenuItemLabel::create(jumpLabel,
CC_CALLBACK_1(HelloWorld::testJump, this));
jumpRotate->setPosition(Point(150, 50));
actionMenu->addChild(jumpRotate);
```

(2) 实现 testJump 函数, 创建 JumpTo 对象 jt, 并让 hero1 执行该动作, 创建 JumpBy 对象 jb, 并让 hero2 执行该动作:

```
JumpTo *jt = JumpTo::create(1, Point(screenSize.width/4*3, screenSize.
height/2), 100, 5);
JumpBy *jb = JumpBy::create(1, Point(-screenSize.width/2,0), 120, 2);

hero1->runAction(jt);
hero2->runAction(jb);
```

这里使用了 create 方法创建 JumpTo 对象, 方法原型为:

```
static JumpTo * create (float duration, const Point &position, float
height, int jumps)
```

参数解释如下。

- ◎ fDuration: 动作执行时间。
- ◎ position: 跳跃到的位置, 设置该点后, 精灵会跳跃到该点。
- ◎ height: 每次跳跃的高度。
- ◎ Jumps: 跳跃的次数, 就是从当前点到目的点之间要跳跃多少次, 每次跳跃距离为总距离除以跳跃的次数。

同理, JumpBy 的创建方法跟 JumpTo 类似, 需要注意的是, JumpBy 里的 position

是跳跃的距离，需要基于目前所在位置计算跳跃后的位置。

(3) 运行程序，单击 **Jump**，就可以看到精灵的跳跃效果，如图 5-4 所示。

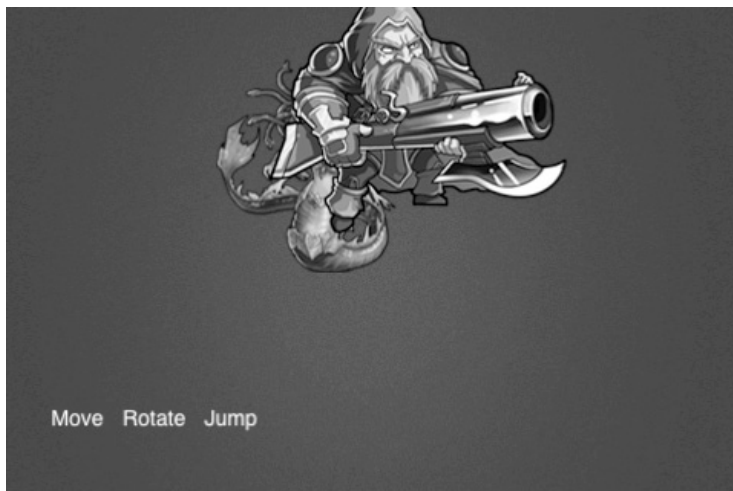


图 5-4 JumpTo 和 JumpBy 实例演示

5.3.4 使用 ScaleTo 和 ScaleBy 实现缩放

ScaleTo 和 ScaleBy 可以缩放精灵，ScaleTo 让精灵缩放到多少倍；ScaleBy 让精灵缩放多少倍，要在缩放前的基础上计算缩放后的倍数。下面介绍如何在游戏场景中使精灵缩放。完整代码请查看清单 5-3。

(1) 添加缩放菜单 **Scale**，当单击该菜单，触发 testScale 回调函数：

```
LabelTTF *scaleLabel= LabelTTF::create("Scale", "Helvetica", 14);
MenuItemLabel *scaleRotate = MenuItemLabel::create(scaleLabel,
CC_CALLBACK_1(HelloWorld::testScale, this));
scaleRotate->setPosition(Point(200, 50));
actionMenu->addChild(scaleRotate);
```

(2) 实现 testScale 函数，创建 ScaleTo 对象 st，并让 hero1 执行该动作，使 hero1 放大到原来的 1.5 倍；创建 ScaleBy 对象 sb，并让 hero2 执行该动作，使 hero2 每次都缩小 0.6。


```
ScaleTo *st = ScaleTo::create(1,1.5,1.5);
ScaleBy *sb = ScaleBy::create(1,0.6);
hero1->runAction(st);
hero2->runAction(sb);
```

这里使用了 `create` 方法创建 `ScaleTo` 对象，方法原型为：

```
static ScaleTo * create (float duration, float sx, float sy)
```

参数解释如下。

- ◎ **duration**: 动作执行时间。
- ◎ **sx**: 浮点数，*X* 轴方向的缩放倍数。
- ◎ **sy**: 浮点数，*Y* 轴方向的缩放倍数。

该例子中的缩放可以用另外一个 `create` 方法创建，原型为：

```
static ScaleTo * create (float duration, float s)
```

参数解释如下。

- ◎ **duration**: 动作执行时间。
- ◎ **s**: 缩放倍数，*X* 轴和 *Y* 轴同时缩放相同的倍数。

既：

```
ScaleTo *st = ScaleTo::create(1,1.5);
```

同理，`ScaleBy` 的创建方法跟 `ScaleTo` 类似。

(3) 运行程序，单击 `Scale` 菜单，就可以看到精灵的缩放的效果菜单，多次单击 `Scale` 菜单，`hero1` 不再改变，`hero2` 不停地缩放，如图 5-5 所示。

5.3.5 使用 `SkewTo` 和 `SkewBy` 实现倾斜变形

`SkewTo` 和 `SkewBy` 可以使精灵倾斜变形，`SkewTo` 让精灵倾斜到设定的角度，`SkewBy` 让精灵在当前的基础上再倾斜设定的角度。下面介绍如何在游戏场景中使精灵倾斜变形。完整代码请查看清单 5-3。

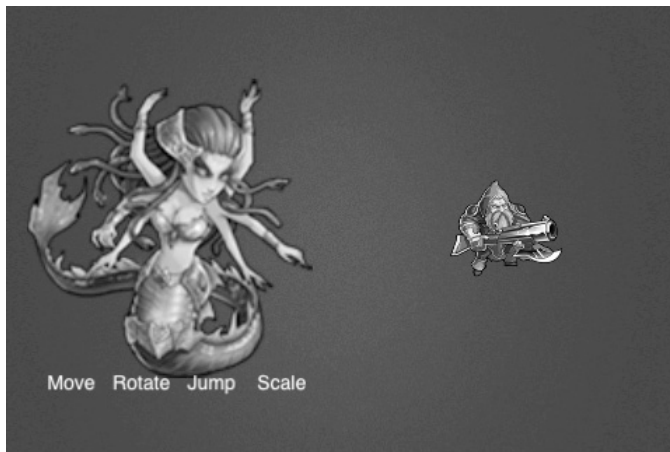


图 5-5 ScaleTo 和 ScaleBy 实例演示

(1) 添加缩放菜单 Skew，当单击该菜单，触发 testSkew 回调函数：

```
LabelTTF *skewLabel= LabelTTF::create("Skew", "Helvetica", 14);
MenuItemLabel *itemSkew = MenuItemLabel::create(skewLabel,
CC_CALLBACK_1(HelloWorld::testSkew, this));
itemSkew->setPosition(Point(250, 50));
actionMenu->addChild(itemSkew);
```

(2) 实现 testSkew 函数，创建 SkewTo 对象 st，并让 hero1 执行该动作，使 hero1 沿 Y 轴变形 60；创建 SkewBy 对象 sb，并让 hero2 执行该动作，使 hero2 每次沿 X 轴变形 30。

```
SkewTo *st = SkewTo::create(1, 0,60);
SkewBy *sb = SkewBy::create(1, -30,0);

hero1->runAction(st);
hero2->runAction(sb);
```

这里使用了 create 方法创建 SkewTo 对象，方法原型为：

```
static SkewTo* create(float t, float sx, float sy);
```

参数解释如下。

◎ t：动作执行时间。

- ◎ **sx**: 浮点数, 建议取值范围-90 到 90, 沿 X 轴方向倾斜, 当 **sx** 为负数时, X 轴上面的部分向左偏移, X 轴下面的部分向右偏移; 当 **sx** 为正数时, X 轴上面的部分向右偏移, X 轴下面的部分向左偏移。
- ◎ **sy**: 浮点数, 建议取值范围-90 到 90, 沿 Y 轴方向倾斜, 当 **sy** 为负数时, Y 轴左半部分向上, Y 轴右半部分向下, X 下面的部分向右偏移; 当 **sy** 为正数时, Y 轴左半部分向下, Y 轴右半部分向上。

同理, **SkewBy** 的创建方法跟 **SkewTo** 类似。

(3) 运行程序, 单击 **Skew** 菜单, 就可以看到精灵的变形效果了, 多次单击 **Skew** 菜单, **hero1** 不再改变, **hero2** 不停地变形, 如图 5-6 所示。

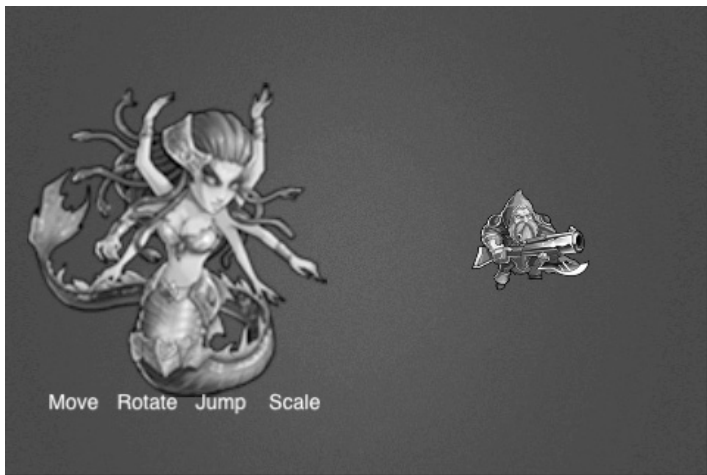


图 5-6 SkewTo 和 SkewBy 实例演示

5.3.6 使用 CardinalSplineBy 和 CardinalSplineTo 实现曲线运动

CardinalSplineBy 和 **CardinalSplineTo** 是样条曲线动作, 它们使用 **create** 方法创建对象, **CardinalSplineTo** 的 **create** 方法原型为:

```
static CardinalSplineTo* create(float duration, PointArray *
    points, float tension );
```

参数解释如下。

- ◎ **duration**: 动作执行时间。
- ◎ **Points**: 控制点类表, 就是在曲线运动过程中, 每次到达的点, 其中第一个为曲线开始点, 最后一个为曲线结束点。
- ◎ **tension**: 松紧程度。**tension==1** 时, 样条线是分段直线。**tension<1** 向外松弛弯曲, **tension>1** 向内缩紧弯曲。

CardinalSplineBy 与 **CardinalSplineTo** 有类似的创建方法, 区别是 **CardinalSplineTo** 中控制点列表的第一个点是起始点; **CardinalSplineBy** 中的起始点是当前所在点, 列表中第一个点作为曲线运动中的一个控制点。下面通过实例演示这两个曲线的用法和区别。完整代码请查看清单 5-3。

(1) 添加曲线运动菜单 **CardinalSpline**, 当单击该菜单, 触发 **testCardinalSplineLabel** 回调函数:

```
LabelTTF *CardinalSplineLabel= LabelTTF::create("CardinalSpline",
"Helvetica", 14);
MenuItemLabel *itemCardinalSpline = MenuItemLabel::create
(CardinalSplineLabel,CC_CALLBACK_1(HelloWorld::testCardinalSplineLabel,
this));
itemCardinalSpline->setPosition(Point(320, 50));
actionMenu->addChild(itemCardinalSpline);
```

(2) 实现 **testCardinalSplineLabel** 函数, 在函数中先创建一个控制点列表 **PointArray** 对象, 然后给该对象添加几个控制点。再用生成的控制点生成 **CardinalSplineBy** 和 **CardinalSplineTo** 对象。前者 **tension** 为 15, 向内缩紧弯曲; 后者 **tension** 为 0, 控制点与控制点之间直线移动。

```
auto s = Director::getInstance()->getWinSize();
auto array = PointArray::create(20);
array->addControlPoint(Point(0, 0));
array->addControlPoint(Point(s.width/2-30,0));
array->addControlPoint(Point(s.width/2-30,s.height-80));
array->addControlPoint(Point(0,s.height-80));
array->addControlPoint(Point(0,0));
//with high tension (tension==1)
```

```

auto *cst = CardinalSplineTo::create(3,array,15);
hero1->runAction(cst);
//with no tension (tension==0)
auto csb = CardinalSplineBy::create(3,array,0);
hero2->runAction(csb);

```

(3) 运行程序，单击 **CardinalSpline**，就可以看到精灵沿着设定的曲线运动，并且 hero1 是从点 (0,0) 开始运动，而 hero2 是从当前节点开始运动，如图 5-7 所示。

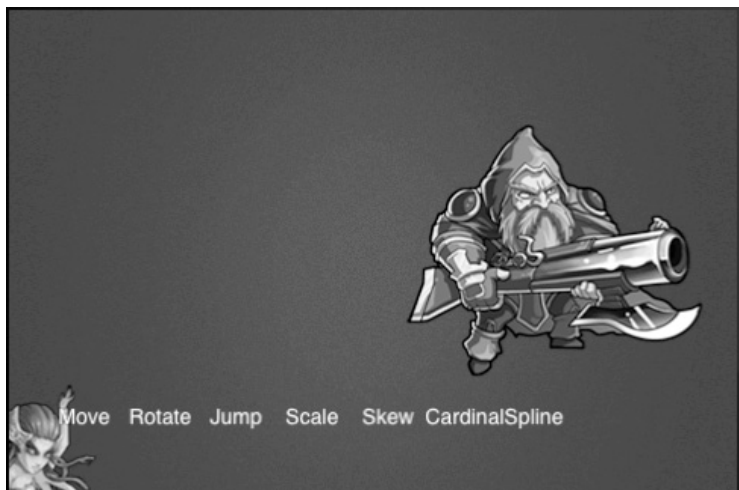


图 5-7 CardinalSplineBy 和 CardinalSplineTo 实例演示

5.3.7 使用 FadeIn 和 FadeOut 实现渐隐渐出

FadeIn 和 FadeOut 是渐隐渐出动作，FadeOut 是逐渐隐藏，FadeIn 是逐渐显示，它们使用 create 方法创建对象，FadeOut 的 create 方法原型为：

```
static FadeOut * create (float duration)
```

参数解释如下。

◎ **duration**：动作执行时间。

FadeIn 具有同样的创建方法，下面通过实例演示 FadeIn 和 FadeOut 的用法和区别。完整代码请查看清单 5-3。

(1) 添加显示隐藏菜单 **CardinalSpline**，当单击该菜单，触发 **testFade** 回调函数：

```
LabelTTF *fadeLabel= LabelTTF::create("Fade", "Helvetica", 14);
MenuItemLabel *itemFade = MenuItemLabel::create(fadeLabel,
CC_CALLBACK_1(HelloWorld::testFade, this));
itemFade->setPosition(Point(390, 50));
actionMenu->addChild(itemFade);
```

(2) 实现 **testFade** 函数，先把 **hero2** 隐藏掉，然后创建 **FadeIn** 对象 **fi** 和 **FadeOut** 对象，再显示 **hero2**，隐藏 **hero1**。

```
hero2->setVisible(false);
auto fi = FadeIn::create(2);
auto fo = FadeOut::create(2);
hero1->runAction(fo);
hero2->runAction(fi);
```

(3) 运行程序，单击 **Fade** 菜单，就可以看到 **hero1** 逐渐隐藏，**hero2** 逐渐显示。

5.4 联合动作

上面讲的都是相互独立执行的动作，下面讲如何把这些独立的动作关联起来，组合成多种多样的关联动作，例如按顺序执行、同时执行、逆向执行等。

5.4.1 按先后顺序执行动作

游戏中经常遇到按一定序列执行的动作，比如先移动后攻击，或者先旋转再跳动，本节就介绍如何用 **Sequence** 实现序列动作。**Sequence** 同样使用 **create** 静态方法创建对象。**create** 方法的原型为：

```
static Sequence* create(FiniteTimeAction * action1, ... );
```

参数可以是 1 到多个数量不定的动作对象，参数以 **NULL** 结尾，表示序列动作结束。下面通过一个简单的实例演示如何使用 **Sequence**，完整代码请查看清单 5-3。

(1) 添加序列动作菜单 **Sequence**，当单击该菜单，触发 **testSequence** 回调函数：

```
LabelTTF *sequenceLabel= LabelTTF::create("sequence", "Helvetica",
14);
MenuItemLabel *itemSequence = MenuItemLabel::create(sequenceLabel,
CC_CALLBACK_1(HelloWorld::testSequence, this));
itemSequence->setPosition(Point(50, 25));
```

(2) 实现 **testSequence** 函数，先创建一个跳跃动作，再创建一个旋转动作，然后用 **sequence** 把两个关联起来：

```
//跳跃
JumpTo *jt = JumpTo::create(1, Point(screenSize.width/4*3, screenSize.
height/2), 100, 5);
//旋转
RotateTo *rt = RotateTo::create( 0.2, 100, 100);
Sequence * simpleSequence = Sequence::create(jt,rt,NULL);
hero1->runAction(simpleSequence);
```

(3) 运行程序，单击 **Sequence**，就可以看到 **hero1** 先跳动到某一点，再旋转。

5.4.2 同时执行动作

游戏开发中也经常遇到多个动作同时进行的情况，比如边移动边攻击，本节就介绍如何用 **Spawn**，实现动作的同时进行。**Spawn** 使用 **create** 静态方法创建对象。**create** 方法的原型为：

```
static Spawn* create(FiniteTimeAction *action1, ... );
```

参数解释：参数可以是 1 到多个数量不定的动作对象，参数以 **NULL** 结尾，表示同时进行的动作结束。

下面通过一个简单的实例演示下如何使用 **Spawn**，完整代码请查看清单 5-3。

(1) 添加序列动作菜单 **Spawn**，当单击该菜单，触发 **testSpawn** 回调函数：

```
LabelTTF *spawnLabel= LabelTTF::create("Spawn", "Helvetica", 14);
MenuItemLabel *itemSpawn = MenuItemLabel::create(spawnLabel,
CC_CALLBACK_1(HelloWorld::testSpawn, this));
itemSpawn->setPosition(Point(120, 25));
```

```
actionMenu->addChild(itemSpawn);
```

(2) 实现 `testSpawn` 函数，先创建一个缩放动作，再创建一个旋转动作，然后用 `Spawn` 把两个关联起来：

```
//缩放
ScaleBy *sc = ScaleBy::create(0.5,3);
//旋转
RotateBy *rt = RotateBy::create(0.5, 300, 300);
//使用 Spawn
Spawn * simpleSpawn = Spawn::create(sc,rt,NULL);
hero1->runAction(simpleSpawn);
```

(3) 运行程序，单击 `Spawn`，就可以看到 `hero1` 同时放大旋转。使用 `Spawn` 可以创建很多意想不到的效果。

5.4.3 逆向执行动作

游戏开发中经常遇到逆向执行动作的情况，也就是动作按原来的路径从结束到开始执行一次，本节就介绍如何用动作类的 `reverse` 方法快捷生成原始动作的逆向动作。下面通过一个简单的实例演示如何使用 `reverse`，完整代码请查看清单 5-3。

(1) 添加序列动作菜单 `reverse`，当单击该菜单，触发 `testReverse` 回调函数：

```
LabelTTF *reverseLabel= LabelTTF::create("Reverse", "Helvetica", 14);
MenuItemLabel *itemReverse = MenuItemLabel::create(reverseLabel,
CC_CALLBACK_1(HelloWorld::testReverse, this));
itemReverse->setPosition(Point(180, 25));
actionMenu->addChild(itemReverse);
```

(2) 实现 `testReverse` 函数，在 5.4.2 中，实现了让精灵同时放大和旋转，这里在 5.4.2 的基础上让精灵按照原来的路径逆向缩小、旋转到初始状态。

先用 `reverse` 函数创建逆向动作，并同时执行：

```
Spawn * reverseSpawn = Spawn::create(sc->reverse(),rt->reverse(),
NULL);
```

然后让精灵先放大旋转，再缩小旋转：


```
Sequence *sequence = Sequence::create(simpleSpawn,reverseSpawn,
NULL);
hero1->runAction(sequence);
```

(3) 运行程序，单击 **Reverse**，就可以看到 hero1 先放大旋转，再按原来的路径反向缩小旋转。

5.4.4 多次重复执行动作

当游戏中需要某个精灵多次执行同一个动作时，可以使用 5.4.1 中的 **Sequence** 创建顺序执行的动作模拟重复动作，但是如果要重复的次数是 50，甚至成百上千，**Sequence** 的参数就要有相同的个数，这是多么烦琐又耗时的工作啊！好消息是 Cocos2d-x 提供了创建重复动作的类 **Repeat**。**Repeat** 使用 **create** 静态方法创建对象。**create** 方法的原型为：

```
static Repeat * create (FiniteTimeAction *action, unsigned int times)
```

参数解释如下。

- ◎ **action**：重复执行的动作对象。
- ◎ **times**：动作执行的次数。

下面通过实例演示 **Repeat** 的使用方法。完整代码请查看清单 5-3。

(1) 添加序列动作菜单 **Repeat**，当单击该菜单，触发 **testRepeat** 回调函数：

```
LabelTTF *labelRepeat= LabelTTF::create("Repeat", "Helvetica", 14);
MenuItemLabel *itemRepeat = MenuItemLabel::create(labelRepeat,
CC_CALLBACK_1(HelloWorld::testRepeat, this));
itemRepeat->setPosition(Point(240, 25));
actionMenu->addChild(itemRepeat);
```

(2) 实现 **testRepeat** 函数，让精灵重复执行 5.4.3 中生成的动作 **Sequence**：

```
Repeat *repeat = Repeat::create(sequence,6);
hero1->runAction(repeat);
```

(3) 运行程序，单击 **testRepeat**，就可以看到 hero1 重复执行放大旋转、缩小旋转的动作，共重复执行 6 次。

5.4.5 延时执行动作

有时需要延后一段时间执行某个动作，Cocos2d-x 给我们提供了一个类 **DelayTime**，用来实现等待一段时间后执行某个动作。

DelayTime 使用 **create** 静态方法创建对象。**create** 方法的原型为：

```
static DelayTime * create (float d);
```

参数解释如下。

◎ **d**：动作延时执行的时间。

下面通过实例演示 **DelayTime** 的使用方法。完整代码请查看清单 5-3。

(1) 添加延时动作演示菜单 **Delay**，当点击该菜单，触发 **testDelay** 回调函数。

```
LabelTTF *labelDelay= LabelTTF::create("Delay", "Helvetica", 14);
MenuItemLabel *itemDelay = MenuItemLabel::create(labelDelay,
CC_CALLBACK_1 (HelloWorld::testDelay, this));
itemDelay->setPosition(Point(300, 25));
actionMenu->addChild(itemDelay);
```

(2) 实现 **testDelay** 函数，在 5.4.1 小节实现的跳动和旋转动作之间停顿 1 秒：

```
//跳跃
JumpTo *jt = JumpTo::create(1, Point(screenSize.width/4*3, screenSize.
height/2), 100, 5);
//旋转
RotateTo *rt = RotateTo::create(0.2, 100, 100);
DelayTime *dt = DelayTime::create(1);
//先跳动，然后停留 2s，接着旋转
Sequence * simpleSequence = Sequence::create(jt,dt,rt,NULL);
hero1->runAction(simpleSequence);
```

(3) 运行程序，单击 **testDelay**，就可以看到 **hero1** 先跳动，然后停顿 1 秒，之后开始旋转。



第 6 章

音频处理

游戏离不开音乐，好的音乐会给游戏增枝添叶，让玩家在体验操作和视觉快感的同时获得听觉上的享受。本章介绍 Cocos2d-x 对音频的处理方法，教读者如何在游戏里添加、控制音乐。

6.1 音频处理类 SimpleAudioEngine

Cocos2d-x 集成了 SimpleAudioEngine 类来处理音频，所以需要在执行文件顶部引入 SimpleAudioEngine 头文件，并声明命名空间 CocosDenshion，即：

```
#include <SimpleAudioEngine.h>
using namespace CocosDenshion;
```

SimpleAudioEngine 类通过调用静态方法 getInstance 生成 SimpleAudioEngine 对象，当第一次调用时，会使用 new 生成对象，该方法原型为：

```
static SimpleAudioEngine * getInstance ();
```

得到 SimpleAudioEngine 对象后就可以调用该对象提供的方法对音乐进行播放、暂停、停止、继续等操作。下面具体讲解如何控制音频。

6.2 添加控制背景音乐

背景音乐就是游戏进行时不停播放的音乐，并且同一时间只能播放一个背景音乐，其他音乐效果都是很快结束的，比如单击某个按钮时的声音，本节讲解如何控制背景音乐。

6.2.1 播放背景音乐并调整音量

Cocos2d-x 使用 SimpleAudioEngine 类的 playBackgroundMusic 方法播放背景音乐，playBackgroundMusic 方法原型是：

```
virtual void playBackgroundMusic(const char * pszFilePath, bool bLoop = false)
```

参数解释如下。

- ◎ **PszFilePath**: 需要播放的音乐文件，比如 bg.mp3。
- ◎ **bLoop**: 背景音乐是否循环播放，默认值是 false，即不循环播放，如果设置成 true，就会循环播放。

Cocos2d-x 使用 setBackgroundMusicVolume 来设置音量大小，原型是：

```
virtual void setBackgroundMusicVolume (float volume)
```

参数解释如下。

- ◎ **Volume**: 浮点数，音量大小，取值范围 0.0 到 1.0。

下面用实例演示如何使用该方法，完整代码请查看代码清单 6-2。

(1) 创建项目 6-2，在项目 **Resource** 目录中添加需要的音频文件。

(2) 宏定义背景音乐变量 **MUSIC_FILE**，根据不同的平台定义不同的背景音乐，这样做的原因是不同平台支持不同的音频格式。

```
#if (CC_TARGET_PLATFORM == CC_PLATFORM_WIN32)
    #define MUSIC_FILE "music.mid"
#elif (CC_TARGET_PLATFORM == CC_PLATFORM_BLACKBERRY || CC_TARGET_PLATFORM == CC_PLATFORM_LINUX )
    #define MUSIC_FILE "background.ogg"
#else
    #define MUSIC_FILE "background.mp3"
#endif // CC_PLATFORM_WIN32
```

(3) 在初始化函数 **init** 里预加载背景音乐到内存中，当播放音乐时就可以马上播放，而不会因为加载而有延迟。预加载背景音乐使用函数 **preloadBackgroundMusic**，并设置背景音乐的音量为最大 1。

```
SimpleAudioEngine::getInstance()->preloadBackgroundMusic(MUSIC_FILE);
SimpleAudioEngine::getInstance()->setBackgroundMusicVolume(1);
```

(4) 添加 **Play** 菜单，单击该菜单调用回调函数 **playbgMusic**：

```
LabelTTF *lblPlay= LabelTTF::create("Play", "Helvetica", 14);
MenuItemLabel *itemPlay = MenuItemLabel::create(lblPlay,
CC_CALLBACK_1 (HelloWorld::playbgMusic, this));
itemPlay->setPosition(Point(100, 50));
actionMenu->addChild(itemPlay);
```

(5) 在 **playbgMusic** 里使用 **playBackgroundMusic** 函数播放背景音乐，并循环播放：

```
void HelloWorld::playbgMusic(Object* pSender)
{
    SimpleAudioEngine::getInstance()->playBackgroundMusic(MUSIC_FILE,
```

```
true);  
}
```

(6) 运行程序，单击界面上的 Play 菜单，背景音乐就开始循环播放。

6.2.2 停止播放背景音乐

Cocos2d-x 使用 `stopBackgroundMusic` 来停止音乐，原型是：

```
virtual void stopBackgroundMusic(bool bReleaseData = false)
```

参数解释如下。

◎ `bReleaseData`：当停止时，是否清除背景音乐数据，默认是 `false`。

用下面的方式调用该方法停止播放音乐，完整代码请查看代码清单 6-2。

```
SimpleAudioEngine::getInstance()->stopBackgroundMusic();
```

6.2.3 暂停播放背景音乐

Cocos2d-x 使用 `pauseBackgroundMusic` 来停止音乐，原型是：

```
virtual void pauseBackgroundMusic()
```

用下面的方式调用该方法暂停播放背景音乐，完整代码请查看代码清单 6-2。

```
SimpleAudioEngine::getInstance()->pauseBackgroundMusic();
```

6.2.4 继续播放背景音乐

Cocos2d-x 使用 `resumeBackgroundMusic` 来停止音乐，原型是：

```
virtual void resumeBackgroundMusic()
```

用下面的方式调用该方法继续播放背景音乐，完整代码请查看代码清单 6-2。

```
SimpleAudioEngine::getInstance()->resumeBackgroundMusic();
```

6.3 添加控制音乐效果

游戏中除了背景音乐，还有一些其他音乐效果，比如单击某个按钮时的声音、获得金币时的声音、人物升级时的声音等。与背景音乐不同，在同一时间可以播放多个音乐。添加控制这些音乐效果的方式与控制背景音乐类似，下面详细讲解。

6.3.1 播放音乐

Cocos2d-x 使用 SimpleAudioEngine 类的 playEffect 方法播放背景音乐，playEffect 方法原型是：

```
virtual unsigned int playEffect (const char *pszFilePath, bool bLoop=  
false, float pitch=1.0f, float pan=0.0f, float gain=1.0f)
```

参数解释如下。

- ◎ pszFilePath: 需要播放的音乐文件，比如 bg.mp3。
- ◎ bLoop: 是否循环播放音乐，默认值是 false，即不循环播放，如果设置成 true，就会循环播放。
- ◎ pitch: 频率，即播放速度的快或慢，正常值为 1.0。这也将影响播放需要的时间。
- ◎ pan: 立体声效果，取值范围-1~1，当值为-1 时，只有左声道播放，同理，如果值为 1，只有右声道播放。
- ◎ gain: 音量，取值范围 0~1，默认为 1，为 0 时表示静音。

playEffect 返回一个无符号整形，用来标记被播放的音乐。

同样 Cocos2d-x 使用 setEffectsVolume 来设置音量大小，传递参数和使用方法同 setBackgroundMusicVolume 一样。

下面用实例演示如何使用该方法，完整代码请查看代码清单 6-2。

(1) 宏定义要播放音乐的变量 `EFFECT_FILE`，根据不同的平台定义不同格式的音乐文件，这样做的原因是不同平台支持不同的音频格式。其中 **Android** 平台只支持 `.ogg` 格式的音频文件。

```
#if (CC_TARGET_PLATFORM == CC_PLATFORM_ANDROID)
    #define EFFECT_FILE      "effect2.ogg"
#elif (CC_TARGET_PLATFORM == CC_PLATFORM_MARMALADE)
    #define EFFECT_FILE      "effect1.raw"
#else
    #define EFFECT_FILE "effect1.wav"
#endif // CC_PLATFORM_ANDROID
```

(2) 在初始化函数 `init` 里预加载音乐文件到内存中，当播放音乐时就可以马上播放，而不会因为加载而有延迟。预加载音乐使用函数 `preloadEffect`，并设置音乐的音量为 1。

```
SimpleAudioEngine::getInstance()->preloadEffect( EFFECT_FILE );
SimpleAudioEngine::getInstance()->setEffectsVolume(1);
```

(3) 添加 **Play** 菜单，单击该菜单调用回调函数 `playeffectMusic`。

```
LabelTTF *lbleffectPlay= LabelTTF::create("Play", "Helvetica", 14);
MenuItemLabel *itemeffectPlay = MenuItemLabel::create
(lbleffectPlay, CC_CALLBACK_1(HelloWorld::playeffectMusic, this));
itemeffectPlay->setPosition(Point(100, 260));
actionMenu->addChild(itemeffectPlay);
```

(4) 在 `playeffectMusic` 里使用 `playEffect` 函数播放音乐，参数都按默认值。

```
void HelloWorld::playeffectMusic(Object* pSender)
{
    _soundId = SimpleAudioEngine::getInstance()->playEffect (EFFECT_
FILE);
}
```

同理，添加循环播放音乐的功能。


```
_soundId = SimpleAudioEngine::getInstance()->playEffect (EFFECT_FILE,
true);
```

(5) 运行程序，单击界面上的 Play 菜单，背景音乐就开始播放。

6.3.2 停止播放音乐

Cocos2d-x 使用 stopEffect 来停止音乐，stopEffect 函数原型是：

```
virtual void stopEffect (unsigned int nSoundId)
```

参数解释如下。

◎ nSoundId: playEffect 的返回值。

用下面的方式调用该方法停止播放音乐，完整代码请查看代码清单 6-2。

```
SimpleAudioEngine::getInstance()->stopEffect (_soundId);
```

6.3.3 暂停播放音乐

Cocos2d-x 使用 pauseEffect 来停止播放音乐，原型是：

```
virtual void pauseEffect(unsigned int nSoundId)
```

参数解释如下。

◎ nSoundId: playEffect 的返回值。

用下面的方式调用该方法暂停播放音乐，完整代码请查看代码清单 6-2。

```
SimpleAudioEngine::getInstance()->pauseEffect (_soundId);
```

6.3.4 继续播放音乐

Cocos2d-x 使用 resumeEffect 来停止音乐，原型是：

```
virtual void resumeEffect(unsigned int nSoundId) ;
```

参数解释如下。

◎ nSoundId: playEffect 的返回值。

用下面的方式调用该方法继续播放音乐，完整代码请查看代码清单 6-2。

```
SimpleAudioEngine::getInstance()->resumeEffect(_soundId);
```

6.3.5 停止、暂停、继续播放所有音乐

当游戏同时有多个音乐在播放时，怎么能停止、暂停、继续播放所有音乐呢？Cocos2d-x 给我们提供了实现的方法。

停止播放所有音乐使用 stopAllEffects 函数，即：

```
virtual void stopAllEffects()
```

暂停播放所有音乐使用 pauseAllEffects 函数，即：

```
virtual void pauseAllEffects ()
```

继续播放所有音乐使用 resumeAllEffects 函数，即：

```
virtual void resumeAllEffects ()。
```

下面演示如何使用这些方法。完整代码请查看代码清单 6-2。

(1) 添加三个菜单 stopAll、pauseAll 和 resumeAll，分别调用函数 stopAllMusic、pauseAllMusic 和 resumeAllMusic。

(2) 实现 stopAllMusic 函数，停止播放所有音乐。

```
void HelloWorld::stopAllMusic(Object* pSender)
{
    SimpleAudioEngine::getInstance()->stopAllEffects();
}
```

(3) 实现 pauseAllMusic 函数，暂停播放所有音乐。

```
void HelloWorld::pauseAllMusic(Object* pSender)
{
    SimpleAudioEngine::getInstance()->pauseAllEffects();
}
```

(4) 实现 `resumeAllMusic` 函数，继续播放所有音乐。

```
void HelloWorld::resumeAllMusic(Object* pSender)
{
    SimpleAudioEngine::getInstance()->resumeAllEffects();
}
```

(5) 运行程序，播放音乐，单击添加的按钮，就能测试停止、暂停、继续播放所有音乐的功能。

6.4 Cocos2d-x 支持的音频格式

Cocos2d-x 使用 CocosDenshion 处理音频，在不同平台下支持不同的音频格式。并且在多数平台下，Cocos2d-x 使用不同的 SDK API 播放背景音乐和音效，所以对它们支持的音频格式也略有不同。

对背景音乐格式的支持见表 6-1。

表 6-1 对背景音乐格式的支持

平 台	支持的音频格式
Android	android.media.MediaPlayer 支持的音频格式
IOS	推荐 MP3 and CAF
Windows	.mid, .wav, 不支持 MP3
Marmalade	.mp3
Blackberry	.ogg

对音乐效果格式的支持见表 6-2。

表 6-2 对音乐效果格式的支持

平 台	支持的音频格式
Android	最好是.ogg，也支持.wav
IOS	cocos2d-iphone 中 CocosDenshion 支持的格式，推荐.caf
Windows	.mid, .wav
Marmalade	只支持 raw PCM 格式
Blackberry	.wav

7

第 7 章

Cocos2d-x 瓷砖地图

这一章要讲解如何在游戏中使用瓷砖地图，瓷砖地图流行运用于塔防、休闲小游戏中，比如“超级玛丽”。瓷砖地图就是由正方形或长方形的图片组成的游戏地图。

本章会讲解如何使用 Cocos2d-x 显示、移动、放大、缩小，操作瓷砖地图。

7.1 什么是瓷砖地图

瓷砖地图来源于房屋地板装修，房屋地板由一块块正方形或者长方形的瓷砖组成，这样就能用很小的瓷砖组合成美观漂亮的地板。同理游戏中的瓷砖地图就是由

多个图片组成的游戏世界，这些小图片就被称为瓷砖。

瓷砖地图被分为“90°角瓷砖地图”和“斜 45°角瓷砖地图”。“90°角瓷砖地图”呈现出来的是平面游戏地图（如图 7-1 所示），其中的瓷砖是正方形或者长方形的。“斜 45°角瓷砖地图”可以使用 2D 图形来表现 3D 场景，使用简单的图片和工具就能创造出具有空间感、更加真实的游戏世界，这也是它比较流行的原因之一（如图 7-2 所示），其中的瓷砖是菱形的。

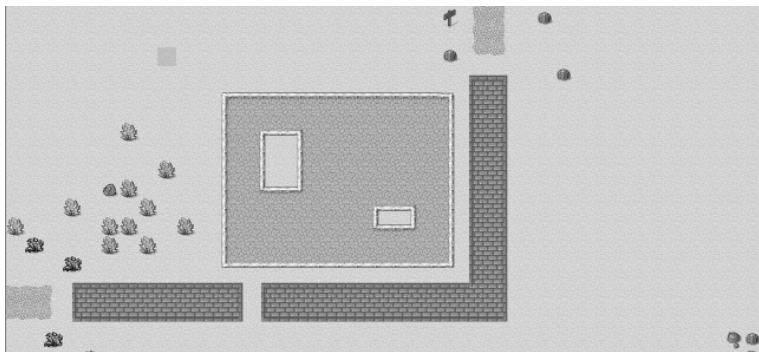


图 7-1 90°角瓷砖地图

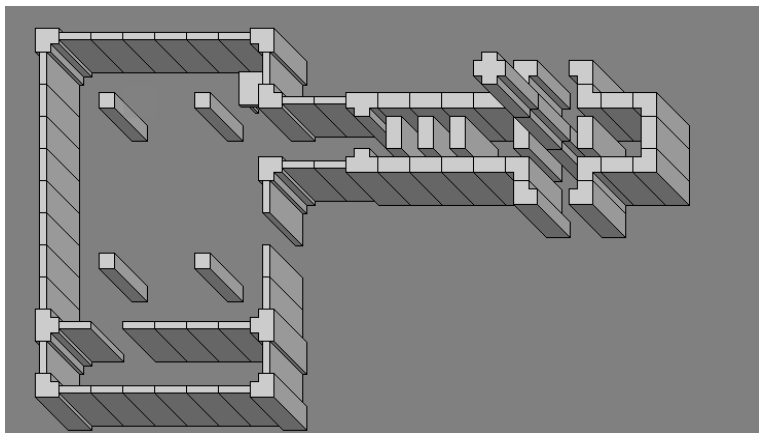


图 7-2 斜 45°角瓷砖地图

瓷砖地图按层级区分，在上面的图层会遮挡下面的图层。并且每块瓷砖都有唯一的标记，即 GID，Cocos2d-x 可以获取瓷砖的 GID，也可以根据 GID 获取瓷砖。

瓷砖地图的坐标与 Cocos2d-x 的坐标有所不同，瓷砖地图以左上角为坐标原点，向右为 X 轴正向，像下为 Y 轴正向。而 Cocos2d-x 的坐标是以左下角为坐标原点。

7.2 使用 Tiled 制作瓷砖地图

上一节介绍了瓷砖地图的基本概念，这节讲解如何制作瓷砖地图。本书使用 Tiled Map Edit 制作瓷砖地图，Tiled Map Edit 简称 Tiled，是一款开源免费的地图编辑软件。

7.2.1 安装 Tiled

安装 Tiled 跟安装其他常用软件类似。

(1) 我们先到 Tiled 官网下载适合的安装包，下载地址是 <http://www.mapeditor.org/download.html>，笔者下载的是 Tiled 0.9.1 for Windows。读者需要下载相应平台的安装包。

(2) 双击安装包，按默认单击“下一步”按钮即可安装完成，是不是很简单？接下来就看下 Tiled 具体长什么样吧，如图 7-3 所示。

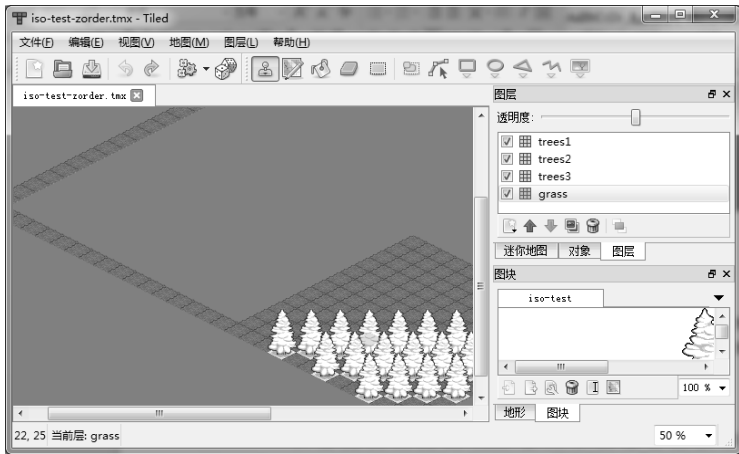


图 7-3 Tiled Map Edit 界面展示

7.2.2 制作地图

上一节把 Tiled 软件安装成功，下面就讲解如何用 Tiled 制作一个塔防地图，最终效果如图 7-4 所示。

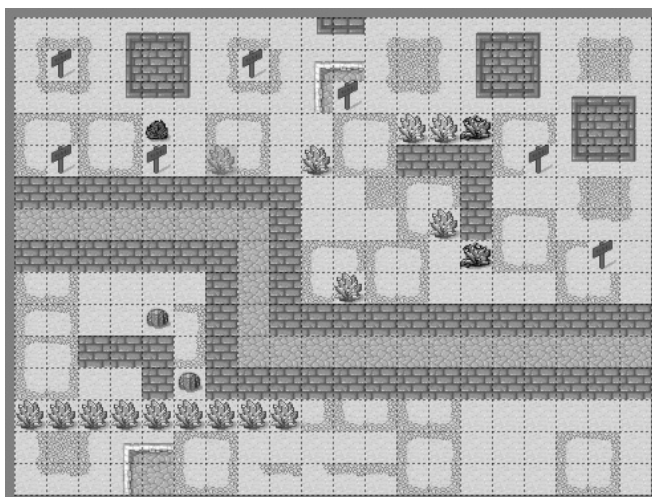


图 7-4 使用 Tiled 制作的塔防地图

(1) 打开 Tiled 地图编辑器，选择菜单栏里的“文件→新建文件”（或者使用快捷键 CTRL+N）新建一张地图，地图方向选择“正常”，代表垂直 90°视角（45°代表 45°角瓷砖地图）。地图大小宽度 20 块，高度 15 块。块大小为 32×32 代表每个图块的尺寸，单位是像素，如图 7-5 所示，然后单击“确定”按钮。



图 7-5 用 Tiled 创建新地图

(2) 新建完地图后, 需要准备一张图片素材用来绘制地面和墙壁。假设已经准备好了素材, 名字为 `tafang.png` (如图 7-6 所示), 图片里包含了制作地图需要的素材, 比如道路、墙壁、石头、植物等。选择菜单栏的“地图→新图块”, 在弹出的对话框里的“图像”位置选择 `tafang.png` 所在位置, 边距和间距修改成 `1px`, 其他保持默认, 单击“确定”按钮。这样就把图片素材添加到 Tiled 里了。

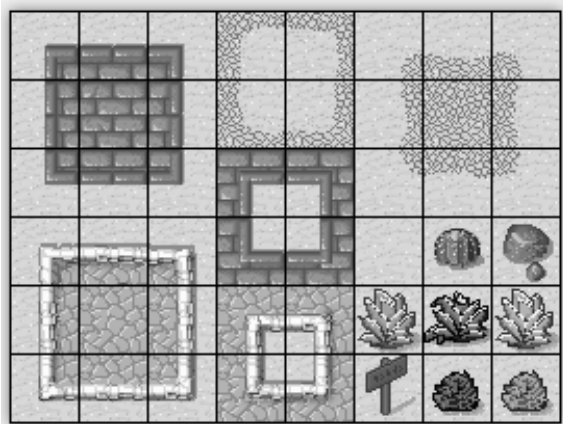


图 7-6 Tiled 创建塔防地图素材

(3) 在 Tiled 软件窗口右上角, 修改图层名为 `bg`。用鼠标选择右下角图块里的任一单元块, 然后再单击左边窗口的图块就能编辑地图了, 当然要想制作好的地图, 需要先设计好, 然后根据设计把图片素材填充到瓷砖地图中。

(4) 选择菜单栏里的“图层→添加图层”, 就会在 Tiled 右上角的图层窗口里添加一个新图层, 我们把新图层名改为 `tree`。新图层会在最上面, 也就是会遮盖其他图层内容。选择 `tree` 图层, 在上面添加一行树, 就会发现图层 `tree` 把图层 `bg` 的内容遮盖了。最下面图层的级别为 `0`, 其他图层依次往上递增。

(5) 在菜单栏里选择“保存”, 保存为 `tafang.tmx`。

这样我们就制作了一个塔防地图。在制作地图时需要注意下面几点。

- ◎ Tiled 编辑器支持在同一个块层使用多个图块文理, 但 Cocos2d-x 引擎的 `TMXTileMap` 类仅支持一个块层对应一个纹理。所以我们需要尽量将图块纹理分类及合并, 在每个块层上仅用一个纹理图来绘制。

- ◎ **TMXTiledMap** 类不允许任何一层中没有任何元素，否则解析时会报错，解决办法是：要么将图层删掉，要么在图层上放置一个透明的图块，让用户无法察觉。
- ◎ 是否将图块导出到外部图块（tileset），这一点需要看游戏本身的需要。如果你的外部图块中给图块配置了大量信息，比如怪物的属性和物品的功能等，这些对每个地图都是通用的，这时导出成外部图块就省去了重复的复制工作。但由于 Tiled 编辑器不支持加密，外部图块的数据会被其他人看到。所以对于正式发布的游戏，建议将影响游戏平衡性的数值配置在外部图块中。

7.3 在游戏中使用瓷砖地图

上两节介绍了瓷砖地图的概念以及如何用 Tiled 制作瓷砖地图，本节讲解如何使用 Cocos2d-x 操作瓷砖地图。

7.3.1 使用 TMXTiledMap 把瓷砖地图加载到游戏中

要想操作瓷砖地图，首先需要把地图加载到游戏中。Cocos2d-x 提供了一个类 **TMXTiledMap**，用来解析和渲染 TMX 地图，通常使用该类的静态方法 **create** 创建对象，然后使用 **addChild** 函数把 **tmx** 对象添加到场景中即可。**create** 原型为：

```
static TMXTiledMap * create (const std::string &tmxFile)
```

参数解释如下。

- ◎ **tmxFile**：需要加载的 **tmx** 文件名。

下面就讲解如何把前面制作的 **tmx** 瓷砖地图应用到游戏中，完整代码请查看 7-3。

（1）创建项目 7-3，把前面制作的 **tafang.tmx** 复制到 **Resources** 文件夹中，注意要把 **tafang.png** 图片一起复制到 **Resources** 中。

(2) 在初始化函数 `init` 中, 创建 `TMXTiledMap` 对象, 并添加到场景中:

```
auto map = TMXTiledMap::create("tafang.tmx");  
addChild(map);
```

(3) 按 `F5` 键运行程序, 就能看到已经把地图渲染到游戏中了, 如图 7-7 所示。



图 7-7 在游戏中渲染 `tmx` 地图

7.3.2 拖曳 `TMX` 地图

上节虽然把 `tmx` 地图渲染到游戏里了, 但是 `tmx` 地图的大小超过了屏幕尺寸, 只能显示部分地图, 其他地方都被隐藏起来了。那么如果能拖曳地图, 就能查看被隐藏起来的地图, 本节就讲解如何实现拖曳 `TMX` 地图。

移动 `TMX` 地图的原理是, 当手指或鼠标在屏幕上移动时, 获取移动的大小和方向, 然后把 `TMX` 地图按照移动的方向和大小重新设定位置。注意, 这里可能要用到与触摸事件相关的内容, 如果了解触摸事件的详情, 请看相关章节。下面演示如何用代码实现, 完整代码请查看代码清单 7-3。

(1) 添加 `onTouchesMoved` 事件监听函数, 当手指或鼠标在屏幕上移动时, 会触发该函数。在该函数里, 获取手指移动的大小和方向, 以及 `TMX` 地图当前的位置, 然后再移动 `TMX` 地图。

```
void HelloWorld::onTouchesMoved(const std::vector<Touch*>& touches,
Event *event)
{
    auto touch = touches[0];
    auto diff = touch->getDelta();
    auto node = getChildByTag(10);
    auto currentPos = node->getPosition();
    node->setPosition(currentPos + diff);
}
```

(2) 在 `init` 函数里注册监听器，当监听到 `onTouchesMoved` 事件时，就调用 `onTouchesMoved` 函数。

```
auto listener = EventListenerTouchAllAtOnce::create();
listener->onTouchesMoved = CC_CALLBACK_2(HelloWorld::onTouchesMoved,
this);
_eventDispatcher->addEventListenerWithSceneGraphPriority(listener,
this);
```

(3) 运行程序，用鼠标拖曳地图，就能看到被隐藏的地图了。

7.3.3 在 TMX 地图中添加并移动精灵

TMX 可以被渲染到游戏中，那么能不能添加精灵等对象到 TMX 地图中呢？答案是肯定的，TMX 地图可以像层、精灵一样通过 `addChild` 添加其他对象。

下面就讲解如何添加精灵到 TMX 地图中，并在地图中移动精灵，完整代码请参考代码清单 7-3。

(1) 创建精灵，并添加到 TMX 地图中，设置精灵的初始位置是图层 `tree` 里那排树最左边。注意添加精灵时的层级为 0，这样精灵会在图层 `bg` 上面，在图层 `tree` 下面。

```
enemy = Sprite::create("chong.png");
map->addChild(enemy,0);
enemy->retain();
int mapWidth = map->getMapSize().width * map->getTileSize().width;
enemy->setPosition(CC_POINT_PIXELS_TO_POINTS(Point(0,map->
getTileSize().width*2)));
```

```
enemy->setAnchorPoint(Point(0,0));
```

(2) 然后让精灵沿 X 轴移动 350px，再回到原来的位置，这样不停往返运动。

```
auto move = MoveBy::create(10, Point(350,0));
auto back = move->reverse();
auto seq = Sequence::create(move, back, NULL);
enemy->runAction( RepeatForever::create(seq) );
```

(3) 运行程序，就能看到小虫在地图中来回运动，运动时遮挡图层 bg，又被图层 tree 遮挡，如图 7-8 所示。



图 7-8 TMX 地图中添加移动精灵

7.3.4 读写 TMX 地图中的图层和瓷砖

上一节展示了可以向瓷砖地图中添加对象。除此，Cocos2d-x 还可以获取地图某一位置的瓷砖，并可以移动、旋转、移除该处的瓷砖。如果想获取某个位置的瓷砖，应该先获取地图中的图层，再通过图层获取那个位置的瓷砖。Cocos2d-x 使用 `getLayer` 获取图层，原型为：

```
TMXLayer* getLayer(const std::string & layerName) const
```

参数解释如下。

◎ **layerName**: 需要获取图层的名称。

返回值是 **TMXLayer** 对象, **TMXLayer** 是 Cocos2d-x 对 TMX 地图图层的封装, 通过 **TMXLayer** 封装好的方法可以获取图层的名称、获取某位置的瓷砖 (Tiled)、移除某位置的瓷砖等, 其中获取某位置瓷砖的方法是 **getTileAt**, 原型为:

```
Sprite * getTileAt (const Point &tileCoordinate);
```

参数解释如下。

◎ **tileCoordinate**: 点坐标, 比如 **Point(1,1)**, 代表以左上角为中心, 第二行第二列的瓷砖。

该函数返回一个精灵 (**Sprite** 对象), 该对象已经被添加到 **TMXLayer**, 不需要再添加。并且该精灵就像普通精灵对象一样, 可以旋转、缩放、倾斜、设置透明、设置颜色等。当然也可以通过下面的方式移除该精灵:

```
layer->removeChild(sprite, cleanup);
```

或者:

```
layer->removeTileAt(Point(x,y));
```

下面通过实例演示如何读写 **TMX** 地图中的图层和瓷砖, 并操作获取的瓷砖, 完整代码请查看代码清单 7-3。

(1) 获取 **tafang.tmx** 地图中的名为 **tree** 的图层:

```
auto layer = map->getLayer("tree");
```

(2) 在图层 **tree** 中的 **Point(9,7)**处, 有一个路标, 我们获取该瓷砖:

```
auto tile0 = layer->getTileAt(Point(9,7));
tile0->setAnchorPoint( Point(0.5f, 0.5f) );
```

(3) 生成一个序列动作, 选择 360 度, 放大 5 倍, 隐藏, 显示, 缩小到原始大小, 执行回调函数 **removeSprite**, 把该瓷砖从地图中删除:

```
auto rotate = RotateBy::create(2, 360);
auto scale = ScaleBy::create(2, 5);
```

```

auto opacity = FadeOut::create(2);
auto fadein = FadeIn::create(2);
auto scaleback = ScaleTo::create(1,1);
auto finish = CallFuncN::create(CC_CALLBACK_1(HelloWorld::
removeSprite, this));
auto seq0 = Sequence::create(rotate, scale, opacity, fadein,
scaleback, finish, NULL);
tile0->runAction(seq0);

```

(4) 实现删除瓷砖的回调函数 `removeSprite`:

```

void HelloWorld::removeSprite(Node* sender)
{
    auto p = ((Node*)sender)->getParent();
    if (p)
    {
        p->removeChild((Node*)sender, true);
    }
}

```

(5) 运行程序，在窗口中间的位置，你就能看到路标瓷砖执行设定的序列动作后从地图中删除，如图 7-9 所示。



图 7-9 读写瓷砖地图



第 8 章

Cocos2d-x 中的事件机制

游戏同漫画和视频最不一样的地方是，游戏充满交互；玩家需要单击屏幕控制游戏向前发展，需要输入用户名和密码才能登录游戏等。所以与玩家的交互是一款游戏不可缺少的功能。

Cocos2d-x 的人机交互采用事件机制，包括触摸事件、鼠标事件、键盘事件、Acceleration 事件和自定义事件。Cocos2d-x 3.0 采用了全新的事件处理方式，让响应事件的方式更简单明了。本章就来介绍如何使用 Cocos2d-x 处理这些事件。

8.1 事件和事件调度

Cocos2d-x 中的人机交互是通过事件机制实现的，框架本身提供了 4 种事件：EventTouch、EventMouse、EventKeyboard、EventAcceleration，它们都继承自事件类 Event。

其中，EventTouch 代表触摸事件，EventMouse 代表鼠标事件，EventKeyboard 代表键盘事件，EventAcceleration 代表重力感应事件。这些事件在下面几节会分别详细讲解。

当产生一个事件时，可以有多个对象在监听该事件，所以有优先级（Priority）。优先级越高（Priority 值越小），事件响应越靠前。

通过 Event 对象，可以获取跟事件相关的一些信息，比如当前事件的对象、事件类型等，它们通过下面几个函数获取。

1. getCurrentTarget

```
Node* getCurrentTarget();
```

获取事件的当前对象，并作为返回值返回。注意只有事件监听器与 Node 对象相关联才能使用该函数；如果事件监听器被设定为固定的 priority 值，会返回 0。

2. getType

获取事件类型，并返回，无参数。

如果产生了某个事件，怎么接收和响应这些事件呢？Cocos2d-x 使用了事件调度机制，并提供了类 EventDispatcher 来实现调度。EventDispatcher 的主要功能是添加和移除事件监听函数，常用到的函数有下面几个。

1. addEventListener

```
void addEventListener(EventListener * listener)
```

添加一个事件监听器，当调度事件时，添加进去的操作会被放到队列最后，也就是最后接收到事件。

参数 `listener` 是要添加的事件监听器。

2. `addEventListenerWithFixedPriority`

```
void addEventListenerWithFixedPriority(EventListener * listener,
int fixedPriority )
```

给特定事件添加事件监听器，响应优先级是一个固定的值。值越小，优先级越高，响应越早。

参数 `listener` 是特定事件的事件监听器，`fixedPriority` 是监听器的优先级，注意该值不能为 0。

3. `addEventListenerWithSceneGraphPriority`

```
void addEventListenerWithSceneGraphPriority(EventListener * listener,
Node * node)
```

给特定事件添加事件监听器，响应优先级跟场景图形优先级一样。

参数 `listener` 是特定事件的事件监听器，`node` 对象的渲染顺序就是添加的事件监听器的响应优先级，该优先级一般为 0。

4. `removeAllEventListeners`

```
void removeAllEventListeners()
```

该函数的作用是移除所有的事件监听器。

5. `removeEventListener`

```
void removeEventListener(EventListener * listener)
```

移除指定的事件监听器，参数 `listener` 是要移除的监听器。

6. removeEventListeners

```
void removeEventListeners(EventListener::Type listenerType)
```

移除所有具有相同类型的监听器。比如移除所有 `EventListenerTouchOneByOne` 类型的监听器。

Cocos2d-x 通过事件和事件调度机制，能够简单方便地实现玩家与游戏的互动，增强游戏的趣味性、好玩度。

8.2 触摸事件

`EventTouch` 表示触摸事件，是玩家触摸屏幕时产生的事件。触摸事件分 4 种，它们是 `BEGAN`、`MOVED`、`ENDED` 和 `CANCELLED`（表 8-1）。

表 8-1 触摸事件分类

触摸事件分类名	触摸事件简介
BEGAN	玩家触摸屏幕时最先触发的事件
MOVED	玩家手指按在屏幕上并滑动时触发
ENDED	玩家结束触摸时触发
CANCELLED	取消触摸时触发

8.2.1 单点触摸事件的类和方法

如果给事件注册事件监听器，某个事件触发时，就会执行相应的动作。Cocos2d-x 提供了类 `EventListenerTouchOneByOne`，该类用来生成触摸事件监听器，接收的事件为单点触摸，`EventListenerTouchOneByOne` 使用静态函数 `create` 生成对象：

```
static EventListenerTouchOneByOne * create ();
```

`EventListenerTouchOneByOne` 类有几个重要的方法和属性，用来设置触摸事件的监听方式和响应事件的方法。

常用方法有：

```
void setSwallowTouches(bool needSwallow)
```

参数解释如下。

◎ **needSwallow**: true 或者 false。

该方法用来设置触摸事件是否向下传递。如果 **needSwallow** 为 **true**，就会吞并触摸事件，不向下级传递事件；如果为 **false**，不会吞并触摸事件，会向下级传递事件。

常用属性有：

```
std::function< bool(Touch *, Event *)> onTouchBegan
```

onTouchBegan 是一个回调函数，当触发 **TouchBegan** 事件时，就会调用该属性指向的函数，执行某些操作或动作。

```
std::function< bool(Touch *, Event *)> onTouchMoved
```

onTouchMoved 是一个回调函数，当触发 **TouchMoved** 事件时，就会调用该属性指向的函数，执行某些操作或动作。

```
std::function< bool(Touch *, Event *)> onTouchEnded
```

onTouchEnded 是一个回调函数，当触发 **TouchEnded** 事件时，就会调用该属性指向的函数，执行某些操作或动作。

```
std::function< bool(Touch *, Event *)> onTouchCancelled
```

onTouchCancelled 是一个回调函数，当触发 **TouchCancelled** 事件时，就会调用该属性指向的函数，执行某些操作或动作。

接下来通过具体的例子来讲解触摸事件相关的各种情况。

8.2.2 单击屏幕移动精灵

本节通过一个小小的实例，讲解如何让一个对象监听触摸事件，效果是当单击屏幕时，场景中的精灵会移动到单击的位置。实例用到了前面介绍的几个方法和类，完整代码请查看代码清单 8-1。

(1) 在 `init` 初始化函数中为场景添加一个精灵，单击屏幕时，移动该精灵。

```
sprite1 = Sprite::create("CyanSquare.png");
sprite1->setPosition(origin+Point(size.width/2, size.height/2) +
Point (-80, 80));
addChild(sprite1, 10);
```

(2) 创建 `EventListenerTouchOneByOne` 对象。

```
auto listener = EventListenerTouchOneByOne::create();
```

(3) 给 `listener` 的 `onTouchBegan` 注册回调函数 `onTouchBegan`，为 `onTouchEnded` 注册回调函数 `onTouchEnded`。

```
listener->onTouchBegan = CC_CALLBACK_2(HelloWorld::onTouchBegan,
this);
```

(4) 在函数 `onTouchBegan` 中返回 `true`。如果返回值为 `true`，会执行 `onTouchMove`、`onTouchEnded` 中的代码。如果返回值为 `false`，不会执行 `nTouchMove`、`onTouchEnded` 中的代码。

```
bool HelloWorld::onTouchBegan(Touch* touch, Event *event)
{
    return true;
}
```

(5) 当触摸结束时，会调用 `onTouchEnded` 函数，我们在该函数中把精灵旋转移到结束触摸的位置。

```
void HelloWorld::onTouchEnded(Touch* touch, Event *event)
{
    auto location = touch->getLocation();
    sprite1->stopAllActions();
    sprite1->runAction( MoveTo::create(1, Point(location.x, location.
y) ) );
    float o = location.x - sprite1->getPosition().x;
    float a = location.y - sprite1->getPosition().y;
    float at = (float) CC_RADIANS_TO_DEGREES( atanf( o/a ) );
    if( a < 0 )
    {
        if( o < 0 )
```

```
        at = 180 + fabs(at);  
    else  
        at = 180 - fabs(at);  
    }  
  
    spritel->runAction( RotateTo::create(1, at) );  
}
```

(6) 调用 `_eventDispatcher` 对象的 `addEventListenerWithSceneGraphPriority` 方法，把监听事件的对象和事件触发的函数关联起来。

```
_eventDispatcher->addEventListenerWithSceneGraphPriority(listener,  
this);
```

(7) 运行程序，单击屏幕中的一点，可以看到精灵边旋转，边移动到单击的地方，如图 8-1 所示。

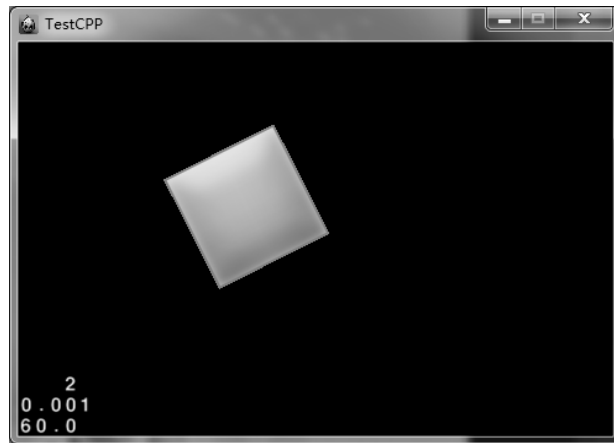


图 8-1 单击屏幕移动精灵

8.2.3 拖动精灵移动

上节讲解了单击屏幕控制精灵，本节讲解让精灵本身监听触摸事件。当手指在屏幕中滑动时，精灵会跟着手指移动；方法和流程同 8.1.3，不同点是控制精灵移动的，完整代码请查看代码清单 8-1。

(1) 在 `init` 初始化函数中为场景添加精灵 2，当拖动该精灵时，精灵会跟着手指移动。

```
auto sprite2 = Sprite::create("MagentaSquare.png");
sprite2->setPosition(origin+Point(size.width/2, size.height/2));
addChild(sprite2, 20);
```

(2) 生成事件监听器，并使用函数 `setSwallowTouches` 设置事件向下传递。

```
auto listener2 = EventListenerTouchOneByOne::create();
listener2->setSwallowTouches(false);
```

(3) 给监听器 `listener2` 注册 `onTouchBegan` 函数，在该函数中先通过参数 `event` 获取当前监听事件的对象：

```
auto target = static_cast<Sprite*>(event->getCurrentTarget());
```

获取开始触摸点的位置，并转变成以 `target` 为参考系的 Node 坐标：

```
Point locationInNode = target->convertToNodeSpace(touch->getLocation());
```

获取 `target` 的大小，然后计算出 `target` 所在区域：

```
Size s = target->getContentSize();
Rect rect = Rect(0, 0, s.width, s.height);
```

判断开始触摸点是不是在精灵 `sprite2` 内部，如果不在，返回 `false`，不执行 `onTouchMoved` 和 `onTouchEnded` 指向的函数。反之，如果在 `sprite2` 内部，就返回 `true`，继续向下执行。

```
if (rect.containsPoint(locationInNode))
{
    log("sprite began... x = %f, y = %f", locationInNode.x,
locationInNode.y);
    target->setOpacity(180);
    return true;
}
return false;
```

(4) 给监听器 `listener2` 注册 `onTouchMoved` 函数，当手指按着屏幕并移动时会触发该函数。

```
listener2->onTouchMoved = [](Touch* touch, Event* event){  
    auto target = static_cast<Sprite*>(event->getCurrentTarget());  
    target->setPosition(target->getPosition()+touch->getDelta());  
};
```

(5) 使用事件调度对象 `_eventDispatcher` 为 `listener2` 分配优先级。

```
_eventDispatcher->addEventListenerWithSceneGraphPriority(listener2,  
sprite2);
```

(6) 运行程序，拖曳红色方块，红色方块就会跟着移动，如图 8-2 所示。

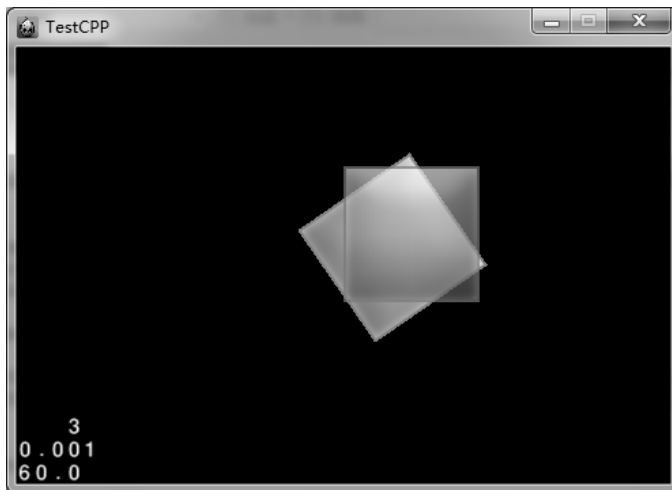


图 8-2 拖曳精灵移动

8.2.4 修改监听器的优先级

如果一个场景中包含多个 `Node` 对象，每个对象都关联一个事件监听器，并且需要设置这些监听器的响应顺序，那么就要手动改变这些监听器的 `priority` 值。本节通过一个实例讲解如何使用 `addEventListenerWithFixedPriority` 函数改变监听器的优先级。

(1) 在项目中添加头文件 `ChangePriority.h` 和执行文件 `ChangePriority.cpp`，用来生成一个类 `ChangePriority`，该类继承自 `Sprite`，有两个私有成员变量 `_listener` 和

`_fixedPriority`，分别代表事件监听器和响应等级。

(2) 添加 `setPriority` 函数，用来设置 `_fixedPriority` 的值：

```
void ChangePriority::setPriority(int fixedPriority)
{
    _fixedPriority = fixedPriority;
};
```

(3) 重写 `onEnter` 函数，在该函数中生成一个事件监听器。执行的动作是，当单击对象时，对象变成红色，当触摸结束时，对象变成白色。

```
listener->onTouchBegan = [=](Touch* touch, Event* event){

    Point locationInNode = this->convertToNodeSpace(touch->
getLocation());
    Size s = this->getContentSize();
    Rect rect = Rect(0, 0, s.width, s.height);

    if (rect.containsPoint(locationInNode))
    {
        this->setColor(Color3B::RED);
        return true;
    }
    return false;
};

listener->onTouchEnded = [=](Touch* touch, Event* event){
    this->setColor(Color3B::WHITE);
};
```

(4) 为生成的 `listener` 添加优先级：

```
_eventDispatcher->addEventListenerWithFixedPriority(listener,
_fixedPriority);
```

(5) 重写 `onExit` 函数，移除监听器 `listener`：

```
_eventDispatcher->removeEventListener(_listener);
```

(6) 在 `HelloWorldScene.cpp` 里创建三个 `ChangePriority` 对象，并设置各自的优先级，它们的优先级分别为 30、20、10。

```
auto sprite1 = ChangePriority::create();  
sprite1->setPriority(30);  
auto sprite2 = ChangePriority::create();  
sprite2->setPriority(20);  
auto sprite3 = ChangePriority::create();  
sprite3->setPriority(10);
```

(7) 运行程序，单击精灵 1 和精灵 2 重合的部分，精灵 2 会产生颜色变化。同理单击精灵 2 和精灵 3 重合的部分，精灵 3 会产生颜色变化，如图 8-3 所示。

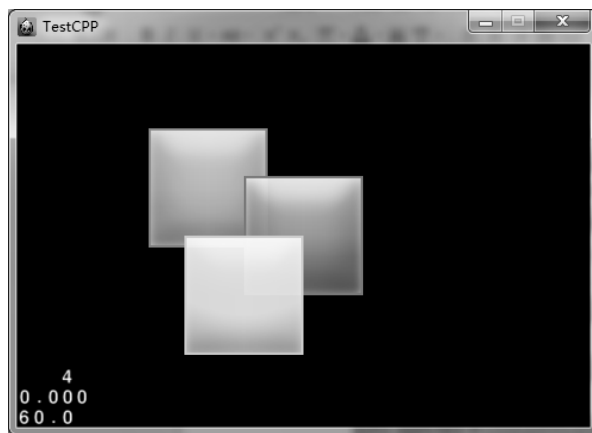


图 8-3 改变 Node 对象的 priority 的值

8.2.5 多点触摸事件

上文讲到的都是单点触摸，Cocos2d-x 也支持多点触摸事件，使用类 `EventListenerTouchAllAtOnce` 来处理多点触摸事件，它也使用静态函数 `create` 生成对象：

```
static EventListenerTouchAllAtOnce* create ();
```

同 `EventListenerTouchOneByOne` 只有一个触摸事件不同，`EventListenerTouchAllAtOnce` 同时会接收多个事件，所以它的属性与 `EventListenerTouchOneByOne` 略有变化。

常用属性如下。

```
std::function<void(const std::vector<Touch*>&, Event*)>
onTouchesBegan
```

`onTouchesBegan` 是一个回调函数，当多点触摸开始时，就会调用该属性指向的函数，执行某些操作或动作。

```
std::function<void(const std::vector<Touch*>&, Event*)>
onTouchesMoved
```

`onTouchesMoved` 是一个回调函数，当多个手指在屏幕上移动时，就会调用该属性指向的函数，执行某些操作或动作。

```
std::function<void(const std::vector<Touch*>&, Event*)>
onTouchesEnded
```

`onTouchesEnded` 是一个回调函数，当结束多点触摸时，就会调用该属性指向的函数，执行某些操作或动作。

```
std::function<void(const std::vector<Touch*>&, Event*)>
onTouchesCancelled
```

`onTouchesCancelled` 是一个回调函数，当取消多点触摸时，就会调用该属性指向的函数，执行某些操作或动作。

下一节通过一个例子来讲解如何使用 `EventListenerTouchAllAtOnce` 及其常用属性。

8.2.6 使用多点触摸实现缩放

开发过 iOS 或 Android 应用的读者应该知道这两个平台都提供手势操作，使用方便快捷，但 Cocos2d-x 没提供这些手势操作。本章使用多点触摸的方式模拟 iOS 的缩放手势，原理就是随时处理两点之间的距离与上一时刻两点之间的距离来获取缩放比。

下面是主要代码，完整代码请查看代码清单 `MutiTouchScale`。

(1) 在场景中添加一个精灵，用来演示缩放效果：

```
map = Sprite::create("FightScene.png");
map->setPosition(Point(screenSize.width/2, screenSize.height/2));
```

```
addChild(map);
```

(2) 创建事件监听器，并注册回调函数：

```
auto listener = EventListenerTouchAllAtOnce::create();
listener->onTouchesBegan = CC_CALLBACK_2(HelloWorld::
onTouchesBegan, this);
listener->onTouchesMoved = CC_CALLBACK_2(HelloWorld::
onTouchesMoved, this);
listener->onTouchesEnded = CC_CALLBACK_2(HelloWorld::
onTouchesEnded, this);
_eventDispatcher->addEventListenerWithSceneGraphPriority
(listener, this);
```

(3) 在 `onTouchesMoved` 函数中，先获取两个触摸点：

```
Touch *pTouch1 = touches.at(0);
Touch *pTouch2 = touches.at(1);
```

然后计算两个手指之间的距离：

```
lastDistance = ccpDistance( pTouch1->getLocation(), pTouch1->
getLocation() );
```

获取手指刚开始移动时的距离，`distance` 默认为 0，当 `distance` 为 0 时，表示刚开始移动，把两点的距离作为初始距离：

```
if(distance == 0){
    distance = lastDistance;
}
```

根据两点距离比值设置精灵的缩放大小：

```
map->setScale(scale * lastDistance/distance );
```

(4) 当触摸结束时，如果放大比率大于 2，把放大比率设置为 2。同样，如果缩小比率小于 0.5，也要把缩小比率设置为 0.5。这些动作在 `onTouchesEnded` 函数中实现：

```
if(map->getScale()>2){
    map->setScale(2);
}else if(map->getScale()<0.5){
    map->setScale(0.5);
}
```

```
}  
scale = map->getScale();
```

(5) 运行程序，用两个手指在屏幕上捏合或者扩大，能同时看到开始添加的精灵也跟着放大或者缩小。当图片放大超过一倍时，松开手指，图片会返回到一倍；当缩小到一半时，松开手指，图片会返回缩小到一半。

8.3 鼠标事件

MouseEvent 表示鼠标事件，是玩家使用鼠标时产生的事件。鼠标事件分 5 种，分别是 MOUSE_NONE、MOUSE_DOWN、MOUSE_UP、MOUSE_MOVE、MOUSE_SCROLL，如表 8-2 所示。

表 8-2 鼠标事件分类

鼠标事件分类名	鼠标事件简介
MOUSE_NONE	没有鼠标操作时触发的事件
MOUSE_DOWN	按下鼠标时触发的事件
MOUSE_UP	松开鼠标时触发的事件
MOUSE_MOVE	移动鼠标时触发的事件
MOUSE_SCROLL	滚动鼠标滚轮时触发的事件

由于本书介绍的是手机游戏开发，手机中没有鼠标事件，所以对此不进行详细说明。

8.4 键盘事件

8.4.1 键盘事件介绍

EventKeyboard 表示键盘事件，是玩家使用键盘或者虚拟键盘时产生的事件。键盘上的每一个按键都对应一个键盘事件。比如 KEY_3 代表数字 3，KEY_CAPITAL_K 代表大写 K，KEY_K 代表小写 k。

类似鼠标事件，Cocos2d-x 也提供了处理键盘事件的类 `EventListenerKeyboard`，使用静态函数 `create` 创建对象：

```
auto listener = EventListenerKeyboard::create();
```

`EventListenerKeyboard` 对象有两个常用属性，用来注册响应键盘事件的方法。

```
std::function<void(EventKeyboard::KeyCode, Event*)> onKeyPressed;
```

当按下键盘时会调用 `onKeyPressed` 指定的回调函数。

```
std::function<void(EventKeyboard::KeyCode, Event*)> onKeyReleased;
```

当释放键盘时会调用 `onKeyReleased` 指定的回调函数。

下面讲解如何使用类 `EventListenerKeyboard` 来处理键盘事件。

8.4.2 实例：把键盘输入内容显示在屏幕中

本节通过一个示例讲解如何使用键盘事件。该示例在场景中放一个 `LabelTTF`，用来显示按下键盘的编码，完整代码请查看代码清单 8-4-1。

(1) 在场景中间添加一个 `label`，用来显示按键的编码：

```
label = LabelTTF::create("Press Keyboard", "Arial", 28);
addChild(label, 0);
label->setPosition(Point(visibleSize.width/2,
visibleSize.height/2));
```

(2) 创建 `EventListenerKeyboard` 对象，并给 `onKeyPressed` 和 `onKeyReleased` 指定回调函数：

```
auto listener = EventListenerKeyboard::create();
listener->onKeyPressed = CC_CALLBACK_2(HelloWorld::onKeyPressed,
this);
listener->onKeyReleased = CC_CALLBACK_2(HelloWorld::onKeyReleased,
this);
_eventDispatcher->addEventListenerWithSceneGraphPriority
(listener, this);
```

(3) 实现 `onKeyPressed` 函数，在该函数中把按下键的代码显示在 `label` 中。注

意，不能直接使用 `label->setString(str->getCString());`。

```
String *str = String::createWithFormat("KeyPressedCode %d ", keyCode);
std::string stdStr = str->getCString();
label->setString(stdStr);
```

(4) 实现 `onKeyReleased` 函数，在该函数中把释放键的代码显示在 `label` 中。

```
String *str = String::createWithFormat("KeyReleasedCode %d ",
keyCode);
std::string stdStr = str->getCString();
label->setString(stdStr);
```

(5) 运行程序，按键盘中的任意一个键，当按下是场景中间显示“`KeyPressedCode`+数字”；松开键盘时，场景中间显示“`KeyReleasedCode`+数字”，如图 8-4 所示。

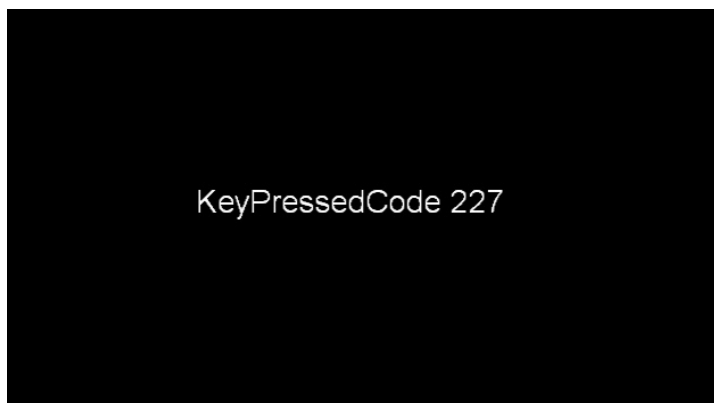


图 8-4 键盘事件示例

8.5 加速计

8.5.1 加速计介绍

`EventAcceleration` 表示加速计事件，现在的手机几乎都具有加速计功能。加速计在游戏中也得到广泛应用，比如赛车类游戏、跑酷类游戏，这些游戏利用加速计控制小车或者人物的前进方向。iOS 设备或大部分设备上的加速计都是支持三维立

体感应的,将 iPhone 屏幕向上放到桌上,那么 z 轴上的力就是 $-1g$,如果竖着放(home 键在下)那么 y 轴上的力就是 $-1g$; g 是单位,我们在应用中调用的一般都是纯粹的浮点数字,甩一甩或者摇一下,作用在某一个方向上的力就会突然加大。力度不同,值也不同,一般情况下绝对值会在 1.5 左右;Cocos2d-x 要使用加速计,必须先启动移动设备的加速计功能,即:

```
Device::setAccelerometerEnabled(true);
```

Cocos2d-x 定义了加速计类 Acceleration, 即:

```
class Acceleration
{
public:
    double x;
    double y;
    double z;

    double timestamp;

    Acceleration(): x(0), y(0), z(0), timestamp(0) {}
};
```

其中 x 、 y 、 z 分别表示 x 轴、 y 轴、 z 轴上的力,有正负之分,代表上下左右的不同倾斜。

Cocos2d-x 提供了类 EventListenerAcceleration 来处理加速计事件。EventListenerAcceleration 没有相应的属性指定监听函数,但在创建对象时通过设定参数设定。使用 create 方法创建对象,原型为:

```
static EventListenerAcceleration* create(std::function<void
(Acceleration*, Event*)> callback);
```

参数是回调函数。下节用一个实例来演示如何处理加速计事件。

8.5.2 实例：利用加速计控制小球移动

本节利用加速计控制小球在屏幕中移动,讲解如何使用 Cocos2d-x 处理加速计事件。完整代码请看代码清单 8-5-1。

(1) 在初始化函数中开启加速计功能:

```
Device::setAccelerometerEnabled(true);
```

(2) 添加要移动的精灵小球, 初始位置在屏幕中间, 并把精灵小球声明为类的私有变量, 方便其他函数调用:

```
_ball = Sprite::create("ball.jpg");
_ball->setPosition(Point(visibleSize.width/2.0,visibleSize.height/
2.0));
addChild(_ball);
```

(3) 使用 `create` 生成 `EventListenerAcceleration` 对象 `listener`, 注册静态函数 `onAcceleration`, 并为 `listener` 分发事件:

```
auto listener = EventListenerAcceleration::create(CC_CALLBACK_2
(HelloWorld::onAcceleration, this));
_eventDispatcher->addEventListenerWithSceneGraphPriority(listener,
this);
```

(4) 实现 `onAcceleration` 函数, 先获取小球大小和当前所在位置:

```
auto ballSize = _ball->getContentSize();
auto positionNow = _ball->getPosition();
```

(5) 由于 Cocos2d-x 使用的是 `opengl` 坐标, 与系统的 UI 坐标不一样, 所以要把 `opengl` 坐标转化成 UI 坐标:

```
auto positionTemp = pDir->convertToUI(positionNow);
```

(6) `onAcceleration` 的第一个参数 `acc` 是 `Acceleration` 类型, 根据上面讲的, 可以通过 `acc` 获取当前加速计的方向和力度, 然后计算出变化后的 UI 坐标:

```
positionTemp.x += acc->x * 9.81f;
positionTemp.y += acc->y * 9.81f;
```

(7) 计算出 UI 坐标后, 还需要转化成 `opengl` 坐标, 用来设置小球的位置:

```
auto positionNext = pDir->convertToGL(positionTemp);
```

(8) 为了不让小球超出屏幕边境, 需要修正位置。当计算后的位置的 x 坐标小于小球半径时, x 坐标设置成小球半径, 当 x 轴坐标大于屏幕宽度减去小球半径时, 把 x 轴坐标设置为屏幕宽度减去小球半径。为了实现该功能并便于应用, 我们定义

一个宏：

```
#define FIX_POS(_pos, _min, _max) \  
    if (_pos < _min) \  
        _pos = _min; \  
    else if (_pos > _max) \  
        _pos = _max; \  

```

然后利用宏对 x 坐标进行修正：

```
FIX_POS(positionNext.x, (ballSize.width / 2.0), (visibleSize.width -  
ballSize.width / 2.0));
```

同理对 y 坐标进行修正：

```
FIX_POS(positionNext.y, (ballSize.height / 2.0), (visibleSize.height  
- ballSize.height / 2.0));
```

(9) 修正好坐标后，把小球定位到那个位置：

```
_ball->setPosition(positionNext);
```

(10) 运行程序，部署到移动设备上，前后左右倾斜设备，就能看到小球向倾斜的方向移动，如图 8-5 所示。

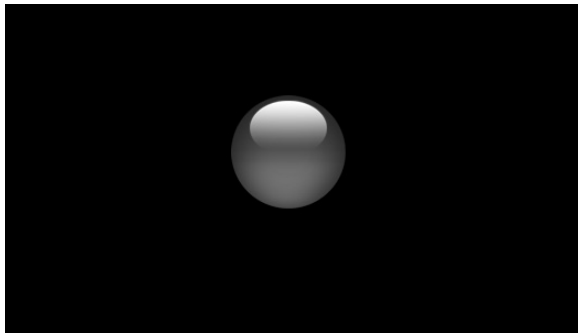


图 8-5 利用加速计控制小球移动



第 9 章

Cocos2d-x 本地数据存储

任何游戏都离不开数据。简单的游戏数据小，逻辑简单；复杂的游戏，数据多，逻辑复杂。产生了数据，就要存储，本章介绍 Cocos2d-x 如何进行本地数据存储，包括自带的小型数据库 UserDefaults、文件存储和 SQLite 数据库。

9.1 使用 UserDefaults 存储数据

9.1.1 UserDefaults 介绍

UserDefaults 采用 XML 格式存储数据，也就是键值对的形式，数据存放在名为 UserDefaults.xml 的文件中。

当存入数据时，如果文件 UserDefault.xml 不存在，会自动创建，存在后就可以进行增删改查操作。

Cocos2d-x 使用单例模型创建 UserDefault 对象：

```
UserDefault::getInstance()
```

UserDefault 可以存放 5 种数据类型，即字符串 (string)、整型 (integer)、单精度浮点型 (float)、双精度浮点型 (double)、布尔型 (bool)。使用 get、set 方法查找、设置数据。

UserDefault 对象设置数据的方法如下。

1. 设置 bool 型数据

```
void setBoolForKey(const char* pKey, bool value);
```

2. 设置整型数据

```
void setIntegerForKey(const char* pKey, int value);
```

3. 设置单精度浮点型数据

```
void setFloatForKey(const char* pKey, float value);
```

4. 设置双精度浮点型数据

```
void setDoubleForKey(const char* pKey, double value);
```

5. 设置字符串型数据

```
void setStringForKey(const char* pKey, const std::string & value);
```

UserDefault 对象查询数据的方法如下。

1. 查询 bool 型数据

```
bool getBoolForKey(const char* pKey, bool defaultValue = false);
```

2. 查询整型数据

```
int    getIntegerForKey(const char* pKey, int defaultValue = 0);
```

3. 查询单精度浮点型数据

```
float  getFloatForKey(const char* pKey, float defaultValue=0.0f);
```

4. 查询双精度浮点型数据

```
double getDoubleForKey(const char* pKey, double defaultValue=0.0);
```

5. 查询字符串型数据

```
std::string  getStringForKey(const char* pKey, const std::string &
defaultValue = "");
```

9.1.2 使用 UserDefaults 存储修改数据

上节介绍了 UserDefaults 的概念和常用存储修改方法，本节通过一个实例演示如何使用这些方法存储、修改、查询数据。完整代码请查看代码清单 9-1。

(1) 新建项目，在初始化函数中使用 `setStringForKey` 函数添加一个键为 `string`、值为 `value` 的数据：

```
UserDefaults::getInstance()->setStringForKey("string", "value");
```

同理添加其他 4 种数据类型的数据：

```
UserDefaults::getInstance()->setIntegerForKey("integer", 10);
UserDefaults::getInstance()->setFloatForKey("float", 2.3f);
UserDefaults::getInstance()->setDoubleForKey("double", 2.4);
UserDefaults::getInstance()->setBoolForKey("bool", true);
```

(2) 使用 `getStringForKey` 函数获取在步骤 (1) 中添加的字符串数据 `string`：

```
std::string  ret    =  UserDefaults::getInstance()->getStringForKey
("string");
```

同理使用其他 `get` 函数获取其他类型数据：

```
double d = UserDefaults::getInstance()->getDoubleForKey("double");
int i = UserDefaults::getInstance()->getIntegerForKey("integer");
float f = UserDefaults::getInstance()->getFloatForKey("float");
bool b = UserDefaults::getInstance()->getBoolForKey("bool");
```

(3) 在场景中添加一个 label，用来显示获得的 string 值：

```
auto *lblStr2 = LabelTTF::create(String::createWithFormat("string is
%s", ret.c_str())-&gtgetCString(), "Arial", 28);
addChild(lblStr2, 0);
lblStr2-&gtsetPosition( Point(100, 20) );
```

同理，再添加 4 个 label，显示其他 4 个数据。这 5 个 label 竖直对齐。

(4) 使用 setStringForKey 更改字符串 string 的值：

```
UserDefaults::getInstance()->setStringForKey("string", "value2");
```

同理，更改其他 4 个数据的值：

```
UserDefaults::getInstance()->setIntegerForKey("integer", 11);
UserDefaults::getInstance()->setFloatForKey("float", 2.5f);
UserDefaults::getInstance()->setDoubleForKey("double", 2.6);
UserDefaults::getInstance()->setBoolForKey("bool", false);
```

(5) 调用 UserDefaults 对象的 flush 方法把修改后的数据保存到 xml 文件中：

```
UserDefaults::getInstance()->flush();
```

(6) 参考上面的方法获取、显示更改后的数据。

(7) 运行程序，可以看到初始数据和更改后的数据，如图 9-1 所示。

init value	changed value
bool is true	bool is false
float is 2.300000	float is 2.500000
int is 10	int is 11
double is 2.400000	double is 2.600000
string is value1	string is value2

图 9-1 UserDefaults 存储数据

9.2 文件

UserDefault 用来存放简单少量的数据，很方便快捷，但是如果数据量很大时，使用 UserDefault 就不是最好的选择了。Cocos2d-x 提供了专门处理文件的相关类，使用这些类和提供的方法能够设置资源搜索路径、判断文件是否存在、读取写入数据等。

9.2.1 文件处理类 FileUtils

Cocos2d-x 提供了专业处理文件的类 FileUtils，FileUtils 也是单例模型，使用静态方法 getInstance 获取对象：

```
auto sharedFileUtils = FileUtils::getInstance();
```

FileUtils 是文件处理的基类，它派生出 4 个子类，如图 9-2 所示，FileUtilsAndroid 提供处理 Android 平台文件的方法，FileUtilsApple 提供处理 iOS 平台文件的方法，FileUtilsLinux 提供处理 Linux 平台文件的方法，FileUtilsWin32 提供处理 Windows 平台文件的方法。当使用 FileUtils 的方法时会根据平台不同调用不同的实现方式。

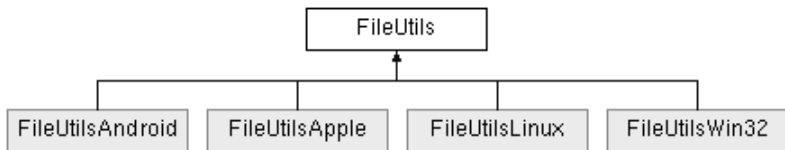


图 9-2 FileUtils 及其子类

9.2.2 判断文件是否存在

使用 FileUtils 的 isFileExist 方法判断某个文件是否存在。isFileExist 的完整声明是：

```
bool FileUtilsWin32::isFileExist(const std::string& strFilePath)
const
```

参数 `strFilePath` 指文件路径，可以是相对路径，也可以是绝对路径。当路径以 `/` 开头时表示该路径是绝对路径，如果不是以 `/` 开头，表示该路径是相对路径。当 `strFilePath` 为相对路径时，将会在默认根路径中搜索文件。

下面通过一个简单示例讲解如何利用 `isFileExist` 方法判断文件是否存在，并把结果显示在场景中。完整代码请查看代码清单 9-2。

(1) 在项目 `Resources` 文件夹中新建文件夹 `image`，在 `image` 文件夹里放一个图片文件 `bg.png`。

(2) 利用 `isFileExist` 判断文件 “`HelloWorld.png`” 是否存在：

```
bool isExist = sharedFileUtils->isFileExist("HelloWorld.png");
```

(3) 添加一个 `LabelTTF`，当图片 `HelloWorld.png` 存在时显示 “`HelloWorld.png exists`”，否则显示 “`HelloWorld.png doesn't exist`”。

```
LabelTTF* pTTF = LabelTTF::create(isExist ? "HelloWorld.png exists" :  
"HelloWorld.png doesn't exist", "Arial", 20);  
pTTF->setPosition(Point(10, s.height-30));  
pTTF->setAnchorPoint(Point::ZERO);  
this->addChild(pTTF);
```

(4) 同理判断 `image/bg.png` 和 `bg.png` 是否存在并用 `LabelTTF` 显示。

(5) 运行程序，可以看到判断结果，如图 9-3 所示。

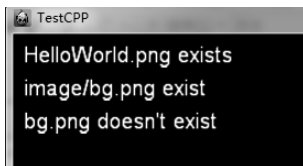


图 9-3 判断文件是否存在

9.2.3 设置文件别名

设置文件别名的意思就是代码里使用一个文件名，但实际使用的是另外一个具有不同名称的文件，Cocos2d-x 通过 `FileUtils` 类的 `setFilenameLookupDictionary` 实现

该功能：

```
virtual void setFilenameLookupDictionary(const ValueMap&
filenameLookupDict);
```

参数 filenameLookupDict 是 ValueMap 类型，ValueMap 是字典类型，定义如下：

```
typedef std::unordered_map<std::string, Value> ValueMap;
```

比如参数为 dic，并做这样的赋值 dict["ball"] = Value("image/ball.jpg")，那么当使用 ball 时，调用的则是 ball.jpg。

下面讲解如何使用 setFilenameLookupDictionary 函数，完整代码请看代码清单 9-2。

(1) 创建 ValueMap 类型 dict，设定 image/ball.jpg 的别名为 ball：

```
ValueMap dict;
dict["ball"] = Value("image/ball.jpg");
```

(2) 调用 setFilenameLookupDictionary 方法：

```
sharedFileUtils->setFilenameLookupDictionary(dict);
```

(3) 使用 ball 作为参数创建一个精灵，并添加到场景中：

```
Sprite *ball = Sprite::create("ball");
ball->setPosition(Point(30,s.height/2.0));
addChild(ball);
```

(4) 运行程序，看到图片 ball.jpg 被添加到场景中（如图 9-4 所示），说明文件名替换成功。Cocos2d-x 中所有使用文件名的地方都可以用该方式去调用另外一个名称的文件。

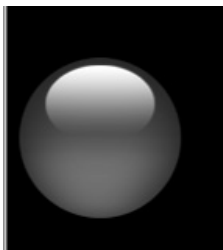


图 9-4 setFilenameLookupDictionary 设置文件别名

9.2.4 获取文件完整路径

文件的完整路径也可以看作文件的绝对路径，FileUtils 提供方法 `fullPathForFilename` 获取文件完整路径：

```
virtual std::string fullPathForFilename(const std::string &filename);
```

参数 `filename` 为文件名，返回该文件的完整路径。

在执行该函数时，首先会从 `filenameLookup` 字典中得到新的文件名，如果在字典中不能找到新的文件名，就用参数传递进来的原始名称；然后，它会尝试使用 `FileUtils` 搜索规则搜索文件名的完整路径。文件搜索基于解析目录和搜索路径的数组元素的顺序。

例如，使用 `setSearchPaths` 设置了两个搜索路径向量(`"/mnt/sdcard/"`, `"internal_dir/"`)，其中 `internal_dir` 是相对于 `Resources/`；使用 `setSearchResolutionsOrder` 设置三个解析元素(`"resources-ipadhd/"`, `"resources-ipad/"`, `"resources-iphonehd/"`)，并且我们在 `fileLookup` 字典中把 `sprite.png` 映射成 `sprite.pvr.gz`。首先，它会使用 `sprite.pvr.gz` 替换 `sprite.png`，然后搜索文件 `sprite.pvr.gz` 如下：

```
/mnt/sdcard/resources-ipadhd/sprite.pvr.gz (if not found, search next)
/mnt/sdcard/resources-ipad/sprite.pvr.gz (if not found, search next)
/mnt/sdcard/resources-iphonehd/sprite.pvr.gz (if not found, search next)
/mnt/sdcard/sprite.pvr.gz (if not found, search next)
internal_dir/resources-ipadhd/sprite.pvr.gz (if not found, search next)
internal_dir/resources-ipad/sprite.pvr.gz (if not found, search next)
internal_dir/resources-iphonehd/sprite.pvr.gz (if not found, search next)
internal_dir/sprite.pvr.gz (if not found, return "sprite.png")
```

如果文件名包含相对路径，如 `gameScene/uiLayer/sprite.png`，且 `fileLookup` 字典把 `gameScene/uiLayer/sprite.png` 映射为 `gameScene/uiLayer/sprite.pvr.gz`，那么搜索顺序为：

```

/mnt/sdcard/gamescene/ui layer/resources-ipadhd/sprite.pvr.gz
(if not found, search next)
/mnt/sdcard/gamescene/ui layer/resources-ipad/sprite.pvr.gz
(if not found, search next)
/mnt/sdcard/gamescene/ui layer/resources-iphonehd/sprite.pvr.gz
(if not found, search next)
/mnt/sdcard/gamescene/ui layer/sprite.pvr.gz
(if not found, search next)
internal_dir/gamescene/ui layer/resources-ipadhd/sprite.pvr.gz
(if not found, search next)
internal_dir/gamescene/ui layer/resources-ipad/sprite.pvr.gz
(if not found, search next)
internal_dir/gamescene/ui layer/resources-iphonehd/sprite.pvr.gz
(if not found, search next)
internal_dir/gamescene/ui layer/sprite.pvr.gz
(if not found, return "gamescene/ui layer/sprite.png")

```

下面通过一个实例讲解如何使用 `fullPathForFilename` 获得文件完整路径。

(1) 结合 9.2.4 获取 `HelloWorld.png` 的完整路径：

```

std::string ret = sharedFileUtils->fullPathForFilename("HelloWorld.
png");

```

(2) 添加一个 `LabelTTF`，显示 `HelloWorld.png` 的完整路径：

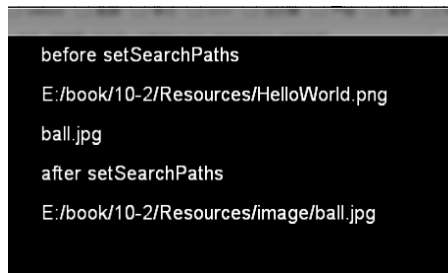
```

pTTF = LabelTTF::create(ret.c_str(), "Arial", 20);
this->addChild(pTTF);

```

(3) 同理，获取文件 `ball.jpg` 的完整路径，并使用 `LabelTTF` 显示在场景中。

(4) 运行程序，查看运行结果，如图 9-5 所示。运行结果：输出了 `HelloWorld.png` 的完整路径，但只输出了 `ball.jpg` 的文件名。这说明 Cocos2d-x 检索文件时，不会递归检索所以文件夹查找文件。那么，是否可以直接让 Cocos2d-x 搜索指定文件夹下的文件呢？答案是肯定的，下节将讲解如何设置搜索路径。



```
before setSearchPaths
E:/book/10-2/Resources>HelloWorld.png
ball.jpg
after setSearchPaths
E:/book/10-2/Resources/image/ball.jpg
```

图 9-5 获取文件完整路径

9.2.5 设置文件搜索路径

本节讲解如何使用 Cocos2d-x 获取和设置搜索路径。获取路径使用 `getSearchPaths` 函数，即：

```
virtual const std::vector<std::string>& getSearchPaths() const;
```

该函数返回所有搜索路径的数组。

设置搜索路径使用 `setSearchPaths` 函数，即：

```
virtual void setSearchPaths(const std::vector<std::string>&
searchPaths);
```

形参 `searchPaths` 为包含所有搜索路径的数组。

下面通过一个实例讲解如何使用这两个函数，完整代码请查看代码清单 9-2。

(1) 设置路径前需要先清空缓存路径：

```
sharedFileUtils->purgeCachedEntries();
```

(2) 使用 `getSearchPaths` 获取默认的搜索路径：

```
_defaultSearchPathArray = sharedFileUtils->getSearchPaths();
std::vector<std::string> searchPaths = _defaultSearchPathArray;
```

(3) 把 `image` 路径添加到 `searchPaths` 数组中：

```
searchPaths.insert(searchPaths.begin(), "image");
```

(4) 使用 `setSearchPaths` 把 `searchPaths` 设置成搜索路径：

```
sharedFileUtils->setSearchPaths(searchPaths);
```

(5) 获取 ball.jpg 的完整路径，并使用 LabelTTF 显示在场景中：

```
ret = sharedFileUtils->fullPathForFilename("ball.jpg");
pTTF = LabelTTF::create(ret.c_str(), "Arial", 20);
```

(6) 运行程序，可以看到 ball.jpg 的完整路径，如图 9-5 所示。

9.2.6 根据分辨率调用不同的资源

移动设备各种各样，屏幕尺寸和分辨率也是五花八门，如果为所有分辨率公用同一套图片资源，会产生变形、图片显示不全、出现黑边、图片质量太差等问题，造成游戏质量的下降。针对这种问题的一种解决方式就是根据不同的分辨率引入不同的资源。FileUtils 类的 setSearchResolutionsOrder 可以实现这种解决方式，可以用它设置搜索资源的顺序：

```
virtual void setSearchResolutionsOrder(const std::vector<std::string>
& searchResolutionsOrder);
```

参数 searchResolutionsOrder 是包含资源路径的数组。

下面通过一个实例讲解如何使用 setSearchResolutionsOrder 为不同的分辨率设置资源路径。

(1) 按照 9.2.5 节所示，设置搜索路径为 image。

(2) 先获取分辨率资源文件搜索路径数组 _defaultResolutionsOrderArray：

```
_defaultResolutionsOrderArray = sharedFileUtils-> getSearchResolutions
Order();
std::vector<std::string> resolutionsOrder = _defaultResolutions
OrderArray;
```

(3) 根据不同分辨率把用到的搜索路径添加到 resolutionsOrder 中：

```
int width = (int)s.width;
int height = (int)s.height;
if(width==1024&&height==768){ //ipad
    resolutionsOrder.insert(resolutionsOrder.begin(),
```

```

"resources-ipad");
    }else if(width==2048&&height==1536){//ipadhd
        resolutionsOrder.insert(resolutionsOrder.begin(),
"resources-ipadhd");
    }else if(width==1136&&height==640){//iphonehd
        resolutionsOrder.insert(resolutionsOrder.begin(),
"resources-iphonehd");
    }else{//iphone
        resolutionsOrder.insert(resolutionsOrder.begin(),
"resources-iphone");
    }
}

```

(4) 使用 setSearchResolutionsOrder 设置资源路径:

```
sharedFileUtils->setSearchResolutionsOrder(resolutionsOrder);
```

(5) 获取 icon.png 的完整路径, 并使用 LabelTTF 在场景中显示出来:

```

ret = sharedFileUtils->fullPathForFilename("icon.png");
pTTF = LabelTTF::create(ret.c_str(), "Arial", 20);
addChild(pTTF);

```

(6) 运行程序, 可以看到输出结果是.../Resources/image/resources-iphone/icon.png。因为窗口大小是 900×640 像素, resolutionsOrder 里的元素是 resources-iphone。

9.2.7 向文件中写入数据

Cocos2d-x 从文件读取数据或者向文件写入数据时, 文件所在位置必须在可写路径中。FileUtils 类提供函数 getWritablePath 来获取可以读取文件数据的路径:

```
virtual std::string getWritablePath() const = 0;
```

该函数返回可读写文件所在目录的路径。

下面通过一个实例讲解如何使用 getWritablePath 和其他函数向文本文件写入数据。

(1) 获取可读写目录的路径:

```
std::string writablePath = sharedFileUtils->getWritablePath();
```

(2) 在可写目录下添加文件 `hello.txt`，并在控制台打印出该文件的绝对路径：

```
std::string fileName = writablePath+"hello.txt";
log(fileName.c_str());
```

(3) 生成一个字符数组，并使用 `fopen` 打开 `hello.txt` 文件：

```
char szBuf[100] = "Hello Cocos2d-x!";
FILE* fp = fopen(fileName.c_str(), "wb");
```

`fopen` 用来打开一个文件，包含在 `stdio` 库中，是 C/C++ 标准库里打开文件的函数，当文件顺利打开后，返回指向该流的文件指针。如果文件打开失败则返回 `NULL`，并把错误代码存在 `errno` 中，原型为：

```
FILE * fopen(const char * path, const char * mode);
```

参数说明如下。

- ◎ **path**：要打开的文件名。
- ◎ **mode**：打开文件的模式，表 9-1 展示了所有打开的模式。

表 9-1 文件打开模式

名 称	解 释
r	以只读方式打开文件，该文件必须存在
r+	以可读写方式打开文件，该文件必须存在
rb+	读写打开一个二进制文件，允许读数据
rt+	读写打开一个文本文件，允许读和写
w	打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件
w+	打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件
a	以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留（EOF 符保留）
a+	以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留（原来的 EOF 符不保留）
wb	只写打开或新建一个二进制文件，只允许写数据
wb+	读写打开或建立一个二进制文件，允许读和写
wt+	读写打开或者建立一个文本文件，允许读写

续表

名 称	解 释
at+	读写打开一个文本文件，允许读或在文本末追加数据
ab+	读写打开一个二进制文件，允许读或在文件末追加数据

(4) 判断文件是否打开成功，如果打开成功使用 `fwrite` 写入数据，如果写入失败，使用 `CCASSERT` 中断程序，最后关闭文件。

```
if (fp)
{
    size_t ret = fwrite(szBuf, 1, strlen(szBuf), fp);
    CCASSERT(ret != 0, "fwrite function returned zero value");
    fclose(fp);
}
```

`fwrite` 是 C/C++ 标准库里向文件写入数据的函数，函数原型是：

```
size_t fwrite(const void* buffer, size_t size, size_t count, FILE* stream);
```

返回值为实际写入的数据块数目

参数如下。

- ◎ `buffer`：是一个指针，对 `fwrite` 来说，是要获取数据的地址；
- ◎ `size`：要写入内容的单字节数；
- ◎ `count`：要进行写入 `size` 字节的数据项的个数；
- ◎ `stream`：目标文件指针；

`fwrite` 以二进制形式对文件进行操作，不局限于文本文件。

(5) 运行程序，查看控制台，打印出了文件的绝对路径 `E:/book/10-2/proj.win32/Debug.win32/hello.txt`。按照该路径找到 `hello.txt` 文件，打开该文件，看到 `Hello Cocos2d-x!`成功写入文件中。

9.2.8 从文件中读取数据

本节讲解如何从一个文件中读取数据，并在控制台中显示，完整代码请查看代码清单 9-2。

(1) 获取可写文件目录的路径，并添加到搜索路径中：

```
std::string writePath = sharedFileUtils->getWritablePath();
sharedFileUtils->addSearchPath(writePath);
```

(2) 获取 `hello.txt` 的完整路径，并在控制台打印出来：

```
std::string fullPath = sharedFileUtils->fullPathForFilename ("hello.txt");
log("hello.txt path = %s", fullPath.c_str());
```

(3) 使用 `fopen` 打开文件，如何打开成功，使用 `fread` 从文件中读取数据。在控制台打印出来，最后关闭打开的文件。

```
fp = fopen(fullPath.c_str(), "rb");
if (fp)
{
    char szReadBuf[100] = {0};
    int read = fread(szReadBuf, 1, strlen(szBuf), fp);
    if (read > 0)
        log("The content of hello.txt from writable path: %s",
szReadBuf);
    fclose(fp);
}
```

(4) 运行程序，查看控制台，可以看到下面的输出，说明文件读取成功。

```
hello.txt path = E:/book/10-2/proj.win32/Debug.win32/hello.txt
The content of hello.txt from writable path: Hello Cocos2d-x!
```

Cocos2d-x 也封装了从文件中读取数据的函数，可以使用 `getStringFromFile` 从文件中读取字符串，函数原型是：

```
virtual std::string getStringFromFile(const std::string& filename);
```

返回值为从文件中读取的 `string` 类型的数据。

参数如下。

◎ **Filename:** 需要读取的文件名。

上面的第 3 步可以使用下面这段代码代替：

```
std::string data = sharedFileUtils->getStringFromFile(fullPath);
if (data.length>0)
{
    log("The content of hello.txt by getStringFromFile: %s", data.
c_str());
}
```

(5) 运行程序，在控制台看到下面的输出，说明读取文件成功。

```
The content of hello.txt by getStringFromFile: Hello Cocos2d-x!
```

9.2.9 把数据写入 plist 文件

plist 文件是一种特殊的 xml 文件，Cocos2d-x 可以向.plist 写入数据，plist 中的键值对及层级关系主要通过数组（Array）和字典（Dictionary）实现。下面讲解如何写入数据，完整代码请查看代码清单 9-2。

(1) 创建一个字典 root，使用字符串为字典设置键值对：

```
auto root = Dictionary::create();
root->setObject(String::create("string element value"), "string
element key");
```

(2) 创建一个数组 arrayInDict，之后会把该数组添加到字典 root 中：

```
auto arrayInDict = Array::create();
```

(3) 创建一个字典 dictInArray，并添加两个数据，最后把 dictInArray 添加到数组 arrayInDict 中：

```
auto dictInArray = Dictionary::create();
dictInArray->setObject(String::create("string in dictInArray
value 0"), "string in dictInArray key 0");
dictInArray->setObject(String::create("string in dictInArray
value 1"), "string in dictInArray key 1");
```

```
arrayInDict->addObject(dictInArray);
```

(4) 在数组 `arrayInDict` 中添加一个 `string` 对象:

```
arrayInDict->addObject(String::create("string in array"));
```

(5) 在创建一个数组 `arrayInArray`, 给数组添加两个 `string` 数据, 然后把 `arrayInArray` 添加到 `arrayInDict` 中:

```
auto arrayInArray = Array::create();
arrayInArray->addObject(String::create("string 0 in arrayInArray"));
arrayInArray->addObject(String::create("string 1 in arrayInArray"));
arrayInDict->addObject(arrayInArray);
```

(6) 把数组 `arrayInDict` 添加到 `root` 中:

```
root->setObject(arrayInDict, "array");
```

(7) 创建一个字典 `dictInDict`, 为 `dictInDict` 添加一个 `string` 数据, 然后把 `dictInDict` 添加到 `root` 中:

```
auto dictInDict = Dictionary::create();
dictInDict->setObject(String::create("string in dictInDict value"),
"string in dictInDict key");
root->setObject(dictInDict, "dictInDict");
```

(8) 获取可写文件路径, 生成 `plist` 文件的绝对路径:

```
std::string writablePathPlist = FileUtils::getInstance()-> getWritablePath();
std::string fullPathPlist = writablePathPlist + "text.plist";
```

(9) 使用 `writeToFile` 把字典 `root` 写入到文件 `text.plist` 文件中, 并在控制台打印出来:

```
if(root->writeToFile(fullPathPlist.c_str()))
    log("see the plist file at %s", fullPathPlist.c_str());
else
    log("write plist file failed");
```

(10) 运行程序, 查看控制台, 看到下面的输出说明写入成功。

```
see the plist file at E:/book/10-2/proj.win32/Debug.win32/text.plist
```

我们根据上面的路径找到 `text.plist` 文件，用文本编辑器打开该文件，可以看到成功写入了，如图 9-6 所示。

```
<plist version="1.0">
  <dict>
    <key>string element key</key>
    <string>string element value</string>
    <key>dictInDict</key>
    <dict>
      <key>string in dictInDict key</key>
      <string>string in dictInDict value</string>
    </dict>
    <key>array</key>
    <array>
      <dict>
        <key>string in dictInArray key 0</key>
        <string>string in dictInArray value 0</string>
        <key>string in dictInArray key 1</key>
        <string>string in dictInArray value 1</string>
      </dict>
      <string>string in array</string>
    </array>
  </dict>
</plist>
```

图 9-6 向 plist 文件写入数据

9.2.10 从 plist 文件读取数据

Cocos2d-x 主要使用 `Dictionary` 类对 `.plist` 和 `.xml` 文件进行解析。`Dictionary` 使用函数 `createWithContentsOfFile` 从文件中读取数据，原型为：

```
__Dictionary* __Dictionary::createWithContentsOfFile(const char
*pFileName);
```

返回值：把文件中的数据以字典类型返回。

参数：读取的文件名。

下面就讲解如何使用 `Dictionary` 从 9.2.7 节创建的 `text.plist` 文件中读取数据，在控制台打印出键 `"string in dictInArray key 0"` 对应的值，完整代码请查看代码清单 9-2。

(1) 获取 `text.plist` 的绝对路径，并使用 `createWithContentsOfFile` 函数把数据读取到字典 `rootDict` 中。

```
std::string fullPathList = sharedFileUtils->fullPathForFilename
("text.plist");
```

```
Dictionary* rootDict = Dictionary::createWithContentsOfFile
(fullPathList. c_str());
```

(2) 查看 text.plist 文件，可以看到键"string in dictInArray key 0"对应的值在一个数组中的字典里，使用要先获取该数组：

```
Array* arrayDict = dynamic_cast<Array*>(rootDict->objectForKey
("array"));
```

(3) 获取数组 arrayDict 中的字典：

```
Dictionary* dictArray = dynamic_cast<Dictionary*>(arrayDict->
objectAtIndex(0));
```

(4) 从字典 dictArray 中获得键"string in dictInArray key 0"对应的值：

```
String* str = dynamic_cast<String*>(dictArray->objectForKey("string
in dictInArray key 0"));
```

(5) 在控制台打印出得到的值：

```
log("value for \"string in dictInArray key 0\" is \"%s\"",str->
getCString ());
```

(6) 运行程序，查看控制台的输出，可以看到下面的输出，与 text.plist 文件中的值进行对比，说明获取成功。

```
value for "string in dictInArray key 0" is "string in dictInArray value
0"
```

9.3 SQLite 存储

9.3.1 SQLite 简介

SQLite 是一款轻型的数据库，是遵守 ACID 的关系型数据库管理系统，它占用资源非常低，只需要几百 KB 的内存就够了，主要用于嵌入式系统中，并且支持 Windows/Linux/ Unix 等主流操作系统。它具有以下特点。

(1) 一致性的文件格式，便于各平台之间的移植。

SQLite 的官方文档中解释到，我们不要将 SQLite 与 Oracle 或 PostgreSQL 进行比较，而是应该将它看作 fopen 和 fwrite。与我们自定义格式的数据文件相比，SQLite 不仅提供了很好的移植性，如大端小端、32/64 位等平台相关问题，而且还提供了数据访问的高效性，如基于某些信息建立索引，从而提高访问或排序该类数据的性能，SQLite 提供的事务功能也是在操作普通文件时无法有效保证的。

(2) 无须安装和管理配置。

SQLite 本身并不需要任何初始化配置文件，也没有安装和卸载的过程，当然也不存在服务器实例的启动和停止。在使用的过程中，无须创建用户和划分权限。在系统出现灾难时，如电源问题、主机问题等，对于 SQLite 而言，不需要做任何操作。

(3) SQLite 是储存在单一磁盘文件中的完整数据库。

SQLite 的数据库被存放在文件系统的单一磁盘文件内，只要有权限便可随意访问和复制，这样带来的主要好处是便于携带和共享。其他的数据库引擎，基本都会将数据库存放在一个磁盘目录下，然后由该目录下的一组文件构成该数据库的数据文件。尽管我们可以直接访问这些文件，但是我们的程序却无法操作它们，只有数据库实例进程才可以做到。这样的好处是带来了更高的安全性和更好的性能，但是也付出了安装和维护复杂的代价。

(4) SQLite 非常小，大致 13 万行 C 代码，4.43MB；它独立，没有额外依赖，并且具有简单、易用的 API，便于在嵌入式或移动设备上的应用。

(5) 在大部分普通数据库操作中，SQLite 比一些流行的数据库都要快。

(6) SQLite 包含 TCL 绑定，同时通过 Wrapper 支持其他语言的绑定，支持的开发语言包括 C、PHP、Perl、Java、C#、Python、Ruby 等。

(7) 代码开源，可以用于任何用途，包括出售它，且具有良好的注释，有着 90% 以上的测试覆盖率。

综上所述，SQLite 的主要优势在于灵巧、快速和可靠性高。SQLite 的设计者们为了达到这一目标，在功能上做出了很多关键性的取舍。与此同时，也失去了一些对 RDBMS 关键性功能的支持，如高并发、细粒度访问控制（如行级锁）、丰富的内

置函数、存储过程和复杂的 SQL 语句等。正是因为这些功能的牺牲才换来了简单，而简单又换来了高效性和高可靠性。

SQLite 存放的数据是无类型的，可以保存任何类型的数据到用户所想要保存的任何表的任何列中，无论这列声明的数据类型是什么（除了字段类型为“Integer Primary Key”的情况），并且对于 SQLite 来说对字段不指定类型是完全有效的。

虽然 SQLite 允许忽略数据类型，但是仍然建议用户在 Create Table 语句中指定数据类型。因为数据类型对于用户和其他的程序员交流，或者准备换掉数据库引擎时能起到一个提示或帮助的作用。SQLite 支持常见的数据类型，比如 VARCHAR、TEXT、INTEGER、FLOAT、BOOLEAN、BLOB、TIMESTAMP、NUMERIC、VARYING CHARACTER 等。

SQLite 虽然很小巧，但是支持的 SQL 语句不逊色于其他开源数据库，几乎所有的标准 SQL 语句都能支持。

这里简单介绍 SQLite 的基本概念、特点和常用场景，如果想详细了解 SQLite 的功能和用法，请查阅相关书籍或资料。

9.3.2 可视化管理工具 SQLiteStudio

现在有很多款 SQLite 可视化管理工具，这些工具让我们简单快捷地操作管理 SQLite 数据库。本书讲解的一款工具是 SQLiteStudio。SQLiteStudio 具有以下优点。

- (1) 开源免费，遵循 GPLv2 协议。
- (2) 单个可执行文件，无须安装卸载，双击即可使用。
- (3) 界面直观简单，可配置界面颜色、字体，图标等；功能全面强大，支持 SQLite3 和 SQLite2 的所有特性。
- (4) 跨平台，可运行在 Windows 9x/2k/XP/2003/Vista/7/Linux/MacOS X/Solaris/FreeBSD 等平台上。
- (5) 多国语言，目前有 9 种语言版本，包括中文。
- (6) 支持导出数据格式：SQL statements、CSV、HTML、XML、PDF、JSON、

dBase。

- (7) 支持从多种格式的文件导入数据，包括 CSV、dBase 和自定义文本文件。
- (8) 提供很多功能插件，比如格式化代码、历史查询等功能。
- (9) 支持 utf8 编码。

综上所述，SQLiteStudio 是一款不错的、功能齐全、操作简单、开源免费的 SQLite 可视化管理工具。

SQLiteStudio 的下载地址是 <http://sqlitestudio.pl/?act=download>。下载完成后直接双击文件即可运行，如图 9-7 所示。

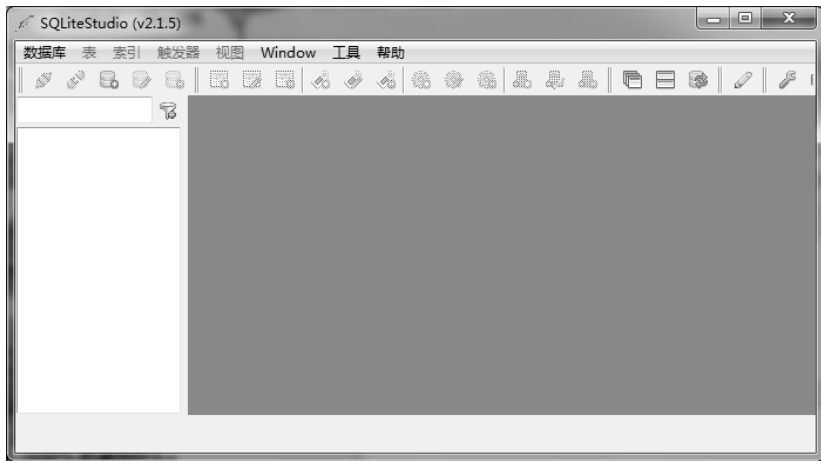


图 9-7 SQLiteStudio 初始界面

在窗口界面上移动鼠标，可以随意查看。

9.3.3 使用 SQLiteStudio 添加数据库

本节讲解如何使用 SQLite 可视化管理工具 SQLiteStudio 添加数据库。

- (1) 在桌面（可以是硬盘的任意位置）创建一个 hello.txt 文本文件，把后缀改成 sqlite，这样就创建了一个 sqlite 数据库 hello.sqlite。

(2) 打开 SQLiteStudio，把鼠标移动到工具栏左边起第三个位置的工具上，显示“添加数据库”，单击该按钮，在弹出的对话框里选择刚创建的 hello.sqlite 数据库，如图 9-8 所示，选择后就能把 hello.sqlite 数据库添加到工具里了。



图 9-8 添加数据库到 SQLiteStudio 中

(3) 虽然把 hello.sqlite 数据库添加进 SQLiteStudio 了，但是窗口左边数据库列表中显示为灰色，不能做任何操作，这是因为 SQLiteStudio 还没连接上数据库。单击工具栏左边的第一个按钮，连接成功后，hello.sqlite 变成激活状态，并多出两个子内容，即表和视图。这样就把数据库成功添加到 SQLiteStudio 中了，如图 9-9 所示。

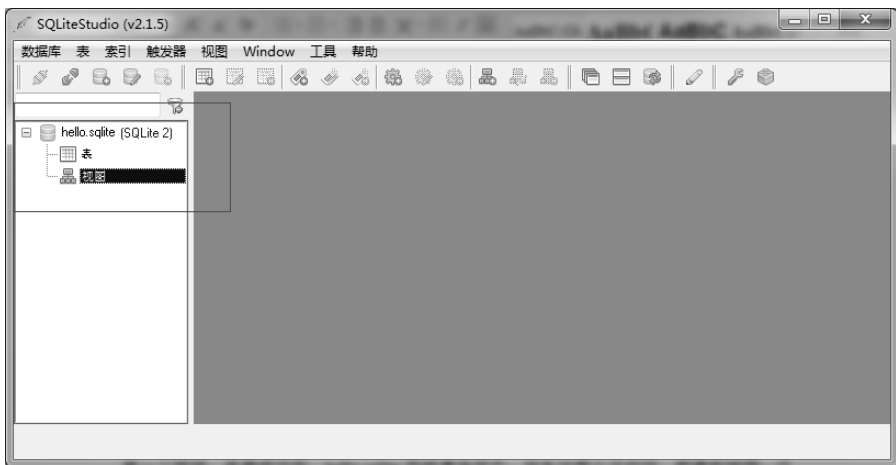


图 9-9 成功添加数据库

9.3.4 使用 SQLiteStudio 添加表和数据

本节讲解如何使用 SQLiteStudio 为数据库创建表、添加字段、增删改查数据等。

(1) 在 hello.sqlite 数据库下面的表上单击鼠标右键，选择“新建表”命令，如图 9-10 所示。

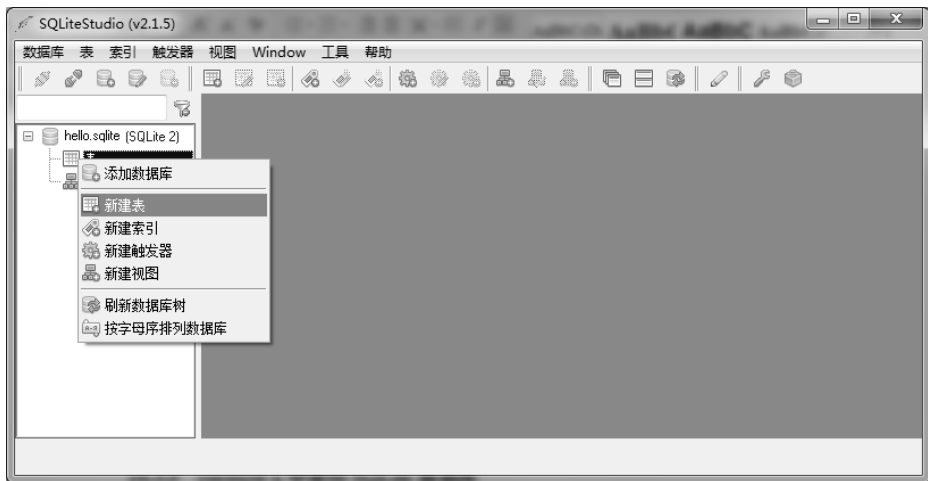


图 9-10 选择“新建表”命令

(2) 在弹出窗口的表名中输入“user”，然后单击右边的“添加列”按钮，会弹出一个“添加列”对话框。在“列名”里输入“name”，数据类型选择“VARCHAR”，单击“添加”按钮。这样就为表 user 添加了一个名为 name、类型为 VARCHAR 的字段，如图 9-11 所示。

(3) 同理，再为表添加 level、hp、mp、coin 四个字段。添加完成后，单击“新建表”对话框中的“创建”按钮，这样就在数据库里添加了表 user，如图 9-12 所示。

(4) 双击表 user，右边的编辑窗格就会出现该表的信息，选项卡“结构”显示了该表的字段和类型，选项卡“数据”显示了该表中的数据，选项卡“索引”显示了该表中的索引，选项卡“触发器”显示了该表中的触发器，选项卡“DLL”显示了创建该表中的 SQL 语句，如图 9-13 所示。



图 9-11 添加字段

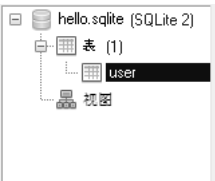


图 9-12 添加表完成

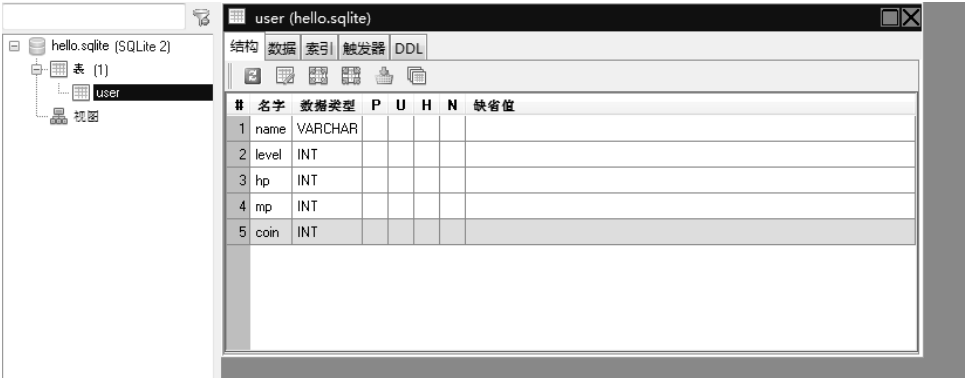


图 9-13 双击表 user

(5) 接下来为表添加一行数据，选择“数据”选项卡，单击下面红色加号按钮，就会添加一行空数据。单击空行中的单元格，输入数据后，再单击红色对号按钮，就能为表添加一行数据，如图 9-14 所示。



图 9-14 添加一行表数据

SQLiteStudio 还提供许多其他功能，比如删除数据、删除表、修改表结构或数据等。本书就不详细列举，读者可以去一一探索。

9.3.5 使用 C 语言接口操作 SQLite 数据库

了解了 SQLite 数据库，本节讲解如何在 Cocos2d-x 中使用 SQLite 数据库。

由于 Cocos2d-x 使用 C/C++ 语言，所以 Cocos2d-x 中使用 SQLite 的 C 语言实现。SQLite 最常用到的数据类型是 `sqlite3 *` 类型，从数据库打开开始，这个类型的变量就代表了读者要操作的数据库，直到数据库关闭。

SQLite 提供了几个常用的方法来操作数据库。

1. 打开数据库链接方法 `sqlite3_open`

```
int sqlite3_open( const char *filename,  sqlite3 **ppDb );
```

该函数用来打开数据库，参数如下。

◎ **filename**: 要打开的数据库名。文件名不需要一定存在，如果此文件不存在，

sqlite 会自动建立它。如果它存在，就尝试把它当数据库文件来打开。

- ◎ **ppDb:** sqlite3 *类型，用来指向打开的数据库。

返回值表示打开数据库是否成功，如果是 SQLITE_OK 则表示打开成功；否则表示打开不成功。

2. 关闭数据库链接方法 sqlite3_close

```
int sqlite3_close(sqlite3 *ppDb);
```

sqlite3_close 用来关闭数据库连接。

3. 执行 sql 语句 sqlite3_exec

```
int sqlite3_exec(sqlite3*, const char *sql, sqlite3_callback, void *,
char **errmsg );
```

sqlite3_exec 用来执行一条 sql 语句，参数如下。

- ◎ **sqlite3*:** open 函数得到的指针，代表已经打开的数据库。
- ◎ **sql:** 一条 sql 语句，以/0 结尾。
- ◎ **sqlite3_callback:** 回调函数，当这条语句执行之后，sqlite3 会去调用读者提供的这个函数。
- ◎ **void *:** void * 是读者所提供的指针，读者可以传递任何一个指针参数到这里，这个参数最终会传到回调函数里面，如果不需要传递指针给回调函数，可以填 NULL。
- ◎ **char **errmsg:** 错误信息。注意是指针的指针。sqlite3 里面有很多固定的错误信息。执行 sqlite3_exec 之后，执行失败时可以查阅这个指针（直接 printf(“%s/n”, errmsg) 得到一串字符串信息，这串信息告诉错在什么地方。sqlite3_exec 函数通过修改传入的指针的指针，把提供的指针指向错误提示信息，这样 sqlite3_exec 函数外面就可以通过这个 char* 得到具体错误提示。

这些参数中，`sqlite3_callback` 和 `void *` 的位置都可以是 `NULL`。`NULL` 表示不需要回调函数。比如进行 `insert` 或 `delete` 操作时，就没有必要使用回调函数。但当进行 `select` 操作时，就要使用回调函数，因为 `sqlite3` 把数据查出来，要通过回调函数告诉你查出了什么数据。

`sqlite3_exec` 的回调函数必须定义成下面所示的函数类型：

```
typedef int (*sqlite3_callback)(void*,int,char**, char**);
```

比如：

```
int funCallBack( void * para, int n_column, char ** column_value, char
** column_name )
{
    //编写自己的代码
}
```

`funCallBack` 的形参解释如下。

- ◎ **para:** `sqlite3_exec` 里传入的 `void *` 参数，通过 `para` 参数，可以传入一些特殊的指针（比如类指针、结构指针），在函数体内进行强制类型转换（参数是 `void*` 类型，必须进行强制类型转换才能得到需要的类型），然后操作这些数据。
- ◎ **n_column:** 一条记录有多少个字段。
- ◎ **column_value:** 保存查出来的数据，它实际上是个 1 维数组，每一个元素都是一个 `char *` 值，是一个字段内容（用字符串表示，以 `/0` 结尾）。

column_name: 与 `column_value` 对应，表示字段对应的字段名称。

下面通过一个实例演示如何操作 SQLite 数据库，完整代码请查看代码清单 9-3。

(1) 在 `sqlite` 官网下载 `sqlite` 的 C 语言源代码，下载地址 <http://www.sqlite.org/download.html>。

(2) 解压下载的压缩包，把 `sqlite3.c` 和 `sqlite3.h` 两个文件复制到项目 `Classes` 目录中，然后在 `visual studio` 中导入这两个文件。这样就可以使用 `sqlite` 数据库了，无须安装、配置。

(3) 在 HelloWorldScene.cpp 文件的顶部引入 sqlite 头文件:

```
#include "sqlite3.h"
```

(4) 先定义多次用的 4 个变量:

```
sqlite3 *pDB = NULL; //数据库指针
char * errMsg = NULL; //错误信息
std::string sqlstr; //SQL 指令
int result; //sqlite3_exec 返回值
```

(5) 使用 sqlite3_open 打开数据库 hero.db, 如果 hero.db 存在, 直接打开, 如果不存在, 先创建再打开。如果打开失败, 在控制台输出打开失败的错误代码和错误原因。

```
result = sqlite3_open("hero.db", &pDB);
if( result != SQLITE_OK )
    log( "打开数据库失败, 错误码:%d , 错误原因:%s\n" , result, errMsg );
```

(6) 创建表 hero, 包括两个字段, 一个是 ID, 整型且自增, 另一个是 name, 长度为 32 的 nvarchar 类型。

```
result=sqlite3_exec( pDB, "create table hero( ID integer primary key
autoincrement, name nvarchar(32) ) " , NULL, NULL, &errMsg );
if( result != SQLITE_OK )
    log( "创建表失败, 错误码:%d , 错误原因:%s\n" , result, errMsg );
```

(7) 删除表中的数据, 如果删除失败, 打印出错误代码和错误原因。

```
sqlstr=" delete from hero";
result = sqlite3_exec( pDB, sqlstr.c_str() , NULL, NULL, &errMsg );
if(result != SQLITE_OK )
    log( "删除记录失败, 错误码:%d , 错误原因:%s\n" , result, errMsg );
```

(8) 向表中插入 3 条数据:

```
sqlstr=" insert into hero( name ) values ( 'zhanshi' ) ";
result = sqlite3_exec( pDB, sqlstr.c_str() , NULL, NULL, &errMsg );
if(result != SQLITE_OK )
    log( "插入记录失败, 错误码:%d , 错误原因:%s\n" , result, errMsg );
```

(9) 查询表中数据, 查询成功后调用函数 callback:

```
sqlstr = "select name from hero";
result = sqlite3_exec(pDB,sqlstr.c_str(),callback,NULL,&errMsg);
```

(10) 实现 **callback** 函数，打印出查询到的数据：

```
int callback(void* ,int nCount,char** pValue,char** pName)
{
    std::string str;
    for(int i=0;i<nCount;i++)
    {
        log("hero %s:%s",pName[i],pValue[i]);
    }
    return 0;
}
```

(11) 关闭数据库：

```
sqlite3_close(pDB);
```

(12) 按 F5 调试程序，查看控制台，能看到打印出了插入的 3 条数据如下：

```
hero name:zhanshi
hero name:fashi
hero name:gongjishou
```

9.3.6 不使用回调查询 SQLite 数据库

9.3.5 使用了回调函数处理查询的结果，本节讲解另外一种查询数据库的方式，该方式可以直接查询而不需要回调函数，该方式通过方法 `sqlite3_get_table` 实现。`sqlite3_get_table` 的原型为：

```
int sqlite3_get_table(sqlite3*, const char *sql, char ***resultp, int
*nrow, int *ncolumn, char **errmsg );
```

参数解释如下。

- ◎ 第 1 个参数指向打开的数据库。
- ◎ 第 2 个参数是 sql 语句，跟 `sqlite3_exec` 里的 sql 是一样的。是一个很普通的以 `/0` 结尾的 `char *` 字符串。

- ◎ 第 3 个参数是查询结果，它依然一维数组。它内存布局是：第一行是字段名称，后面紧接着该字段的值。
- ◎ 第 4 个参数是查询出多少条记录（即查出多少行）。
- ◎ 第 5 个参数是多少个字段（多少列）。
- ◎ 第 6 个参数是错误信息。

当调用 `sqlite3_get_table` 方法后，无论查询是否成功，都要使用 `sqlite3_free_table` 释放掉查询结果。

下面讲解如何通过 `sqlite3_get_table` 获取数据，完整代码请查看代码清单 9-3。

把 9.3.5 中代码第（9）步，也就是查询数据库的代码换成下一行代码：

```
tableresult = sqlite3_get_table( db, "select name from hero", &dbResult,
&nRow, &nColumn, &errmsg );
```

如果查询成功，数据保存在 `dbResult` 中，注意 `dbResult` 是一维数据，第一行是字段名称，后面紧接着该字段的值。下面对 `dbResult` 进行解析。

```
if( SQLITE_OK == tableresult )
{
    index = nColumn; //前面说过 dbResult 前面第一行数据是字段名称，从
nColumn 索引开始才是真正的数据

    for( i = 0; i < nRow ; i++ )
    {
        for( j = 0 ; j < nColumn; j++ )
        {

            log("hero %s:%s",dbResult[j],dbResult [index] );
            ++index; // dbResult 的字段值是连续的，从第 0 索引到第
nColumn - 1 索引都是字段名称，从第 nColumn 索引开始，后面都是字段值，它把一个二维
的表（传统的行列表示法）用一个扁平的形式来表示

        }
    }
}
```

解析完后释放 dbResult:

```
sqlite3_free_table( dbResult );
```

运行程序，查看控制台，输出了插入的 3 个数据，说明查询成功。

```
hero name:zhanshi  
hero name:fashi  
hero name:gongjishou
```



第 10 章

网络编程

随着 3G/4G 网络的普及和手机游戏的发展，越来越多的网络游戏进入玩家的视线，所谓网络游戏，简单说就是用手机上网玩的游戏，最初的手机网游是 WAP 游戏，之后具有图形界面的手机网络游戏迅速发展起来。开发手机网游离不开网络通信，Cocos2d-x 提供了三种网络通信方式，分别是使用 HTTP 协议、Socket 协议和 WebSocket。本节就详细讲解如何使用 Cocos2d-x 进行网络通信。

10.1 HTTP 实现网络通信

10.1.1 HTTP 通信简介及常用类

HTTP 全称简单文本传输协议，是互联网广泛使用的通信协议，短连接形式，常用于 B/S 架构中。HTTP 有 3 种数据提交方式，分别是 GET、POST、PUT。简单来说，GET 适用于从服务器获取小的数据，POST 适合向服务器提交比较大的数据。

Cocos2d-x 封装了 3 个类来处理 HTTP 请求，HttpRequest、HttpClient 和 HttpResponse。它们在命名空间 cocos2d::network 中定义，所以使用时要先声明该命名空间。

```
using namespace cocos2d::network;
```

1. HttpRequest

HttpRequest 是一种数据类型，它提供了一些方法用来定义或获取 HTTP 请求的参数。常用方法包括下面几种。

设置请求连接：

```
void setUrl(const char * url);
```

设置请求类型：

```
void setRequestType(Type type)
```

Type 是 Cocos2d-x 定义的一个枚举类型，包括 5 种类型。

```
enum class Type
{
    GET,
    POST,
    PUT,
    DELETE,
    UNKNOWN,
};
```

设置请求完成后的回调函数：

```
void setResponseCallback(Ref * pTarget, SEL_HttpResponse pSelector )
```

设置请求的数据，参数 `buffer` 是提交的数据，`len` 是请求数据的长度：

```
void setRequestData(const char* buffer, unsigned int len)
```

为请求设置一个标记：

```
void setTag(const char* tag)
```

格式为字符串，可以通过 `HttpResponse->getHttpRequest->getTag()` 获取该标记。标记是 HTTP 请求的可选设置，不是必须设置的。

设置请求头信息，比如 `Content-Type` 等：

```
void setHeaders(std::vector<std::string> pHeaders)
```

上面的设置函数都有一个对应的 `get` 函数用来获取设置的信息，比如 `getUrl` 用来获取 `setUrl` 设置的请求连接。

2. HttpClient

`HttpClient` 使用单例模式，用来处理异步 HTTP 请求。如果在请求里设置了回调函数，当请求完成后，就会在主线程中调用该回调函数。它使用静态方法 `getInstance` 获取：

```
HttpClient* httpHandle = HttpClient::getInstance();
```

创建了 `HttpClient` 对象后，调用 `send` 函数发送请求，参数是定义好的 `HttpRequest` 对象：

```
void send(HttpRequest * request);
```

`HttpClient` 也提供了一些其他函数，比如设置请求超时的时间：

```
void setTimeoutForConnect(int value)
```

设置下载超时的时间：

```
void setTimeoutForConnect(int value)
```

3. HttpResponse

HttpResponse 也是一种数据格式，包含请求返回的数据等信息。使用 HttpResponse 提供的方法可以获取这些数据。

获取引发该请求的 HttpRequest 对象：

```
HttpRequest* getHttpRequest()
```

获取请求返回的数据：

```
std::vector<char>* getResponseData()
```

获取请求状态码：

```
long getResponseCode()
```

如果返回 200，说明请求成功。

下面就通过实例讲解如何使用 HTTP 服务器进行数据通信，由于 HTTP 请求通常使用 GET 和 POST 就可以完成所有功能，很少用到其他方式，所以本书只介绍 GET 和 POST 两种方式。

10.1.2 GET 方式通信

GET 用来获取信息，而且是安全的和幂等的。并且 GET 提交的数据很小，最多只能是 1024 字节。本节通过一个例子讲解如何使用 GET 进行通信，完整代码请查看代码清单 10-1。

(1) 在 HelloWorldScene.h 头文件中引入 HttpClient.h、HttpRequest.h、HttpResponse.h，并声明命名空间 cocos2d::network：

```
#include "cocos2d\cocos\network\HttpClient.h"
#include "cocos2d\cocos\network\HttpRequest.h"
#include "cocos2d\cocos\network\HttpResponse.h"
using namespace cocos2d::network;
```

(2) 在 init 函数中添加一个菜单，单击时调用 onMenuGetClicked 函数，在 onMenuGetClicked 函数中进行 GET 请求。

(3) 发送请求前, 先生成 `HttpRequest` 对象:

```
HttpRequest* request = new HttpRequest();
```

设置请求连接:

```
request->setUrl("http://httpbin.org/get");
```

设置请求类型为 GET:

```
request->setRequestType(HttpRequest::Type::GET);
```

设置请求完成后的回调函数为 `onHttpRequestCompleted`:

```
request->setResponseCallback(this,
httpresponse_selector(HelloWorld::onHttpRequestCompleted));
```

设置该请求的 tag:

```
request->setTag("GET test3");
```

生成 `HttpClient` 对象, 并发送请求:

```
HttpClient::getInstance()->send(request);
```

最后释放 `request`:

```
request->release();
```

(4) 到这一步, 请求已经发出, 接着我们实现回调函数 `onHttpRequestCompleted`, 用来处理请求结果。

首先通过回调函数中的参数 `response` 获取请求的标记, 如果能够获取, 说明请求成功发出。然后在控制台里输出:

```
if (0 != strlen(response->getHttpRequest()->getTag()))
{
    log("%s completed", response->getHttpRequest()->getTag());
}
```

使用 `getResponseCode` 函数获取请求状态码, 当状态码为 200 时表示请求成功, 否则不成功。把请求结果和请求 tag 通过 label 在屏幕中输出:

```
int statusCode = response->getResponseCode();
```

```
char statusString[64] = {};
sprintf(statusString, "HTTP Status Code: %d, tag = %s", statusCode,
response->getHttpRequest()->getTag());
_labelStatusCode->setString(statusString);
```

也可以直接通过 **HttpResponse** 对象提供的函数 **isSucceed** 判断请求是否成功：

```
if (!response->isSucceed())
{
    log("response failed");
    return;
}
```

使用 **getResponseData** 获取请求返回的数据：

```
std::vector<char> *buffer = response->getResponseData();
```

获取数据后，就可以对数据 **buffer** 进行相应处理，本书在控制台里输出这些数据：

```
std::vector<char> *buffer = response->getResponseData();
std::string buf(buffer->begin(),buffer->end());
log("Http Test, dump data:%s",buf.c_str());
```

(5) 运行程序，单击场景中的 **Test Get** 菜单，查看屏幕中间显示的请求状态，如图 10-1 所示。查看控制台，可以看到下面的输出，这说明请求成功。

```
Http Test, dump data:{
  "args": {},
  "headers": {
    "Connection": "close",
    "Accept": "*/*",
    "X-Request-Id": "7da33ac5-0e68-4258-8fb2-9d6201005aac",
    "Host": "httpbin.org"
  },
  "url": "http://httpbin.org/get",
  "origin": "123.117.52.91"
}
```




图 10-1 HTTP 请求

10.1.3 POST 方式通信

POST 用来向服务器提交数据，可能会修改服务器上的资源，且可以提交大量数据。本节在 10.1.2 的基础上修改，完整代码请查看代码清单 10-1。

(1) 添加 “Test Post” 菜单，当单击时调用函数 onMenuPostClicked。

(2) 在函数 onMenuPostClicked 里发送 POST 请求。请求过程跟 GET 几乎一样，只是要把请求类型设置成 POST：

```
request->setRequestType(HttpRequest::Type::POST);
```

也可选择性设置 HTTP 请求头的 Content-Type：

```
std::vector<std::string> headers;
headers.push_back("Content-Type: application/json; charset=utf-8");
request->setHeaders(headers);
```

(3) 使用 setRequestData 函数设置提交的数据：

```
const char* postData = "visitor=cocos2d&TestSuite=Extensions Test/NetworkTest";
request->setRequestData(postData, strlen(postData));
```

(4) 设置请求的 tag 为 POST test1：

```
request->setTag("POST test1");
```

(5) 运行程序，单击“Test Post”菜单，在屏幕中查看请求和对应的状态码。查看控制台，可以看到下面的输出。

```
Http Test, dump data:{
  "url": "http://httpbin.org/post",
  "data": "visitor=cocos2d&TestSuite=Extensions Test/NetworkTest",
  "origin": "123.117.52.91",
  "headers": {
    "Content-Type": "application/json; charset=utf-8",
    "Host": "httpbin.org",
    "X-Request-Id": "faacbf5a-1f9e-4232-b6bf-000e9768fcea",
    "Content-Length": "53",
    "Connection": "close",
    "Accept": "*/*"
  },
  "json": null,
  "files": {},
  "args": {},
  "form": {}
}
```

10.2 Socket 实现网络通信

10.2.1 Socket 简介

Socket 是一种常连接方式，通常也称作套接字，用于描述 IP 地址和端口，是一个通信链的句柄，常用于 C/S 架构中。如果 Internet 上的主机运行某种联网服务，该服务会打开一个 Socket，并绑定到一个端口上，不同的端口对应于不同的服务。Socket 连接过程一般分 3 个步骤：服务器监听、客户端请求、连接确认。

- ◎ 服务器监听：是服务器端套接字，并不定位具体的客户端套接字，而是处于等待连接的状态，实时监控网络状态。
- ◎ 客户端请求：是指由客户端的套接字提出连接请求，要连接的目标是服务

器端的套接字。为此，客户端的套接字必须首先描述它要连接的服务器的套接字，指出服务器端套接字的地址和端口号，然后就向服务器端套接字提出连接请求。

- ◎ 连接确认：是指当服务器端套接字监听到或者说接收到客户端套接字的连接请求时，它就响应客户端套接字的请求，建立一个新的线程，把服务器端套接字的描述发给客户端，一旦客户端确认了此描述，连接就建立好了。而服务器端套接字继续处于监听状态，继续接收其他客户端套接字的连接请求。

Socket 类型有两种：流式 Socket（SOCK_STREAM）和数据报式 Socket（SOCK_DGRAM）。流式 Socket 是一种面向连接的 Socket，针对面向连接的 TCP 服务应用；数据报式 Socket 是一种无连接的 Socket，对应于无连接的 UDP 服务应用。

本书对 Socket 只进行简单介绍，如果想要详细了解，请查阅相关书籍。

10.2.2 在 Cocos2d-x 中使用 Socket

了解了 Socket 的基本概念，这一节讲解如何在 Cocos2d-x 中使用 Socket。要使用 Socket：首先需要继承 Socket 代理类 SIODelegate，并实现 SIODelegate 的 4 个虚函数。

当打开 Socket 连接时会调用以下函数：

```
virtual void onConnect(cocos2d::network::SIOClient* client);
```

当接收到数据的时候会调用以下函数：

```
virtual void onMessage(SIOClient* client, const std::string& data) = 0;
```

当 Socket 连接关闭时调用以下函数：

```
virtual void onClose(SIOClient* client) = 0;
```

当建立连接出现错误时会调用以下函数：

```
virtual void onError(SIOClient* client, const std::string& data) = 0;
```

除了 SIODelegate, Cocos2d-x 使用类 SIOClient 控制 Socket 从打开到关闭连接的整个流程。它提供了下面这些函数。

setTag 为 Socket 连接设置一个标记, 用来区分不同的连接:

```
inline void setTag(const char* tag)
```

关闭 Socket 连接, 完成后会调用上面提到的代理函数 onClose:

```
void disconnect();
```

发送消息到 socket.io 服务器:

```
void send(std::string s);
```

委托类处理 socket.io 事件:

```
void emit(std::string eventName, std::string args);
```

注册 socket.io 事件回调函数, 参数 SIOEvent e 使用 CC_CALLBACK2 (&Base::function, this) 传递:

```
void on(const std::string& eventName, SIOEvent e);
```

此外 Cocos2d-x 还提供一个类 SocketIO, SocketIO 使用单例模型, 是一个容器类, 主要用来生成一个 SIOClient 对象。再使用 SIOClient 对象控制 Socket 连接的整个流程。

下面通过一个实例讲解如何使用这些方法, 完整代码请查看代码清单 10-2。

(1) 创建项目, 添加 Open SocketIO Client 菜单, 单击调用回调函数 onMenuSIOClientClicked。

(2) 在 onMenuSIOClientClicked 函数内, 打开 Socket 连接:

```
_sioClient = SocketIO::connect(*this, "ws://channon.us:3000");
```

为该连接设置一个标记:

```
_sioClient->setTag("Test Client");
```

注册两个事件 testevent 和 echotest, 并添加对应的回调函数:

```
_sioClient->on("testevent", CC_CALLBACK_2(HelloWorld::testevent,
```

```
this));
_sioClient->on("echotest", CC_CALLBACK_2 (HelloWorld::echotest,
this));
```

(3) 实现函数 `testevent`，在控制台输出数据：

```
void HelloWorld::testevent(SIOClient *client, const std::string& data)
{
    log("HelloWorld::testevent called with data: %s", data.c_str());
}
```

同理实现 `echotest` 函数：

```
void HelloWorld::echotest(SIOClient *client, const std::string& data)
{
    log("HelloWorld::echotest called with data: %s", data.c_str());
}
```

(4) 添加 `Send Test Message` 菜单，单击调用 `onMenuTestMessageClicked` 函数，向服务器发送消息：

```
if(_sioClient != NULL) _sioClient->send("Hello Socket.IO!");
```

(5) 添加 `Send Test Event` 菜单，单击调用 `onMenuTestEventClicked` 函数，使用 `emit` 函数触发 `echotest` 事件：

```
if(_sioClient != NULL) _sioClient->emit("echotest", "[{\"name\": \"myname\", \"type\": \"mytype\"}]");
```

(6) 添加菜单 `Disconnect Socket`，调用 `onMenuTestClientDisconnectClicked` 函数，关闭 `Socket` 连接：

```
if(_sioClient != NULL) _sioClient->disconnect();
```

(7) 在 `HelloWorld.h` 中让 `HelloWorld` 继承 `cocos2d::network::SocketIO::SIODelegate`，然后把 `SIODelegate` 的 4 个方法添加成 `HelloWorld` 的公有成员函数。

`onConnect` 函数在控制台输出 `HelloWorld::onConnect called`：

```
void HelloWorld::onConnect(network::SIOClient* client)
{
    log("HelloWorld::onConnect called");
}
```

`onMessage` 在控制台输出 `onMessage` 标记和接收到的数据：

```
void HelloWorld::onMessage(network::SIOClient* client, const std::string& data)
{
    log("HelloWorld::onMessage received: %s", data.c_str());
}
```

同理在 `onClose` 中打印出 `HelloWorld::onClose called`：

```
void HelloWorld::onClose(network::SIOClient* client)
{
    log("HelloWorld::onClose called");
};
```

在 `onError` 中打印出错误信息：

```
void HelloWorld::onError(network::SIOClient* client, const std::string& data)
{
    log("HelloWorld::onError received: %s", data.c_str());
}
```

(8) 运行程序，在场景中单击 `Open SocketIO Client`，查看控制台信息，打印出下面的信息：

```
HelloWorld::onConnect called
HelloWorld::onMessage received: welcome!
HelloWorld::onMessage received: {"name":"myname","type":"mytype"}
HelloWorld::testevent called with data: {"name":"testevent","args":
[{"name":"myname","type":"mytype"}]}
```

这说明打开连接时先调用了委托函数 `onConnect`，之后接收到从服务器传递过来的两条信息，调用了两次委托函数 `onMessage`。之后服务器又发送一个 `testevent` 事件，触发 `testevent` 函数。

再单击 `Send Test Message` 菜单，查看控制台输出：

```
HelloWorld::onMessage received: Hello Socket.IO!
```

说明服务器接收到发送的消息，并把消息返回给客户端，调用了委托函数

onMessage。

单击 Send Test Event 菜单，查看控制台输出：

```
HelloWorld::echotest called with data:
{"name":"echotest","args": [{"name":"myname","type":"mytype"}]}
```

说明触发了事件 echotest，调用回调函数 echotest。

最后单击菜单 Disconnect Socket，控制台输出 HelloWorld::onClose called。成功关闭 Socket 连接，调换委托函数 onClose。

10.3 WebSocket 实现网络通信

10.3.1 WebSocket 简介

WebSocket 是 HTML 5 中定义的新协议，实现了浏览器与服务器之间的全双工通信。在实现 WebSocket 连线过程中，浏览器和服务器只需要做一个握手动作，即浏览器发出 WebSocket 连线请求，服务器接收到请求后发出回应，这个过程被称为“握手”（handshaking）。连接成功后，浏览器和服务器之间就形成了一条快速通道，直接进行数据传送。

传统的 http 协议仅能实现单向的通信，为了达到即时通信（real-time）的目的，常使用轮询（polling）的方式。轮询是指在特定的时间间隔（如每 1 秒），浏览器向服务器发出 HTTP 请求，然后服务器向客服端的浏览器返回最新的数据。这种传统的模式的缺点就是浏览器需要不断地向服务器发出请求（request），然而 HTTP 请求的 header 是非常长的，需要传输的数据可能只是一个很小的值，这样会浪费大量带宽。另一种比较新的技术是 Comet，使用 AJAX 实现，虽然可达到全双工通信，但依然需要发出 HTTP 请求。

相对上面两种方式，WebSocket 具有明显的优势，一是互相沟通的 Header 很小，大概只有 2 Bytes，二是服务器可以主动传送数据给客户端，实现全双工通信。

10.3.2 在 Cocos2d-x 中使用 WebSocket

本节介绍如何在 Cocos2d-x 中使用 WebSocket 实现客户端和服务端之间的通信。类似 Socket，要使用 WebSocket，首先需要继承 WebSocket 代理类 Delegate，并实现 Delegate 的 4 个虚函数。

当打开 WebSocket 连接时会调用以下函数：

```
virtual void onOpen(WebSocket* ws) = 0;
```

当接收到数据的时候会调用以下函数：

```
virtual void onMessage(WebSocket* ws, const Data& data) = 0;
```

参数 data 是 Cocos2d-x 为消息定义的结构体 Data：

```
struct Data
{
    Data():bytes(nullptr), len(0), issued(0), isBinary(false){}
    char* bytes;
    ssize_t len, issued;
    bool isBinary;
};
```

当 WebSocket 连接关闭时调用以下函数：

```
virtual void onClose(WebSocket* ws) = 0;
```

当建立连接时出现错误时会调用以下函数：

```
virtual void onError(WebSocket* ws, const ErrorCode& error) = 0;
```

第二个参数 error 是 Cocos2d-x 定义的枚举类型 ErrorCode。

```
enum class ErrorCode
{
    TIME_OUT,
    CONNECTION_FAILURE,
    UNKNOWN,
};
```

除了 Delegate，Cocos2d-x 使用类 WebSocket 控制 WebSocket 从打开到关闭连接的整个流程。它提供了下面这些函数。

`init` 是 `WebSocket` 的初始化函数，必须在使用 `new` 创建 `WebSocket` 对象后马上调用该函数。参数 `delegate` 是从 `websocket` 接收数据的代理类，`url` 是 `websocket` 服务器地址。

```
bool init(const Delegate& delegate,
          const std::string& url,
          const std::vector<std::string>* protocols = nullptr);
```

关闭 `WebSocket` 连接，完成后会调用上面提到的代理函数 `onClose`。

```
void close();
```

发送字符串到 `websocket` 服务器：

```
void send(const std::string& message);
```

发送二进制数据到 `websocket` 服务器：

```
void send(const unsigned char* binaryMsg, unsigned int len);
```

获取链接状态：

```
State getReadyState();
```

返回值是枚举类型 `State`，`State` 定义了 4 种状态：

```
enum class State
{
    CONNECTING,
    OPEN,
    CLOSING,
    CLOSED,
};
```

下面就通过实例讲解如何使用这些类和方法，完整代码请查看代码清单 10-3。

(1) 添加 `Open WebSocket` 菜单，单击调用函数 `onMenuoOpenClicked`，使用 `new` 创建 3 个 `WebSocket` 对象：

```
_wsiSendText = new network::WebSocket();
_wsiSendBinary = new network::WebSocket();
_wsiError = new network::WebSocket();
```

(2) 创建 `WebSocket` 对象后使用 `WebSocket` 的 `init` 方法建立 `WebSocket` 连接，

前两个地址可以正常连接，最后一个地址不能正常连接：

```
_wsiSendText->init(*this, "ws://echo.websocket.org");
_wsiSendBinary->init(*this, "ws://echo.websocket.org");
_wsiError->init(*this, "ws://invalid.url.com");
```

(3) 添加 Send Text 菜单，单击调用函数 onMenuSendTextClicked，先输出进入该函数的标记：

```
log("HelloWorld::onMenuSendTextClicked");
```

使用 getReadyState 方法获取连接状态，如果状态为 OPEN 向服务器发送字符串消息：

```
if (_wsiSendText->getReadyState() == network::WebSocket::State::
OPEN)
{
    _sendTextStatus->setString("Send Text WS is waiting...");
    _wsiSendText->send("Hello WebSocket, I'm a text message.");
}
```

(4) 添加 Send Binary 菜单，单击调用 onMenuSendBinaryClicked 函数，如果连接状态为 OPEN，使用 send 函数发送二进制数据：

```
log("HelloWorld::onMenuSendBinaryClicked");
if(_wsiSendBinary->getReadyState() == network::WebSocket::State::
OPEN)
{
    _sendBinaryStatus->setString("Send Binary WS is waiting...");
    char buf[] = "Hello WebSocket,\0 I'm\0 a\0 binary\0 message\0.";
    _wsiSendBinary->send((unsigned char*)buf, sizeof(buf));
}
```

(5) 添加 Close WebSocket 菜单，单击调用回调函数 onMenuCloseClicked，关闭 WebSocket 连接：

```
if (_wsiSendText)
    _wsiSendText->close();
if (_wsiSendBinary)
    _wsiSendBinary->close();
if (_wsiError)
```

```
_wsError->close();
```

(6)实现 `WebSocket::Delegate` 的 4 个代理函数。在 `HelloWorld.h` 中让 `HelloWorld` 继承 `cocos2d::network::WebSocket::Delegate`。然后把 `Delegate` 的 4 个方法添加成 `HelloWorld` 的公有成员函数。

`onOpen` 函数在控制台输出连接打开信息:

```
void HelloWorld::onOpen(network::WebSocket* ws)
{
    log("HelloWorld::Websocket (%p) opened", ws);
}
```

`onMessage` 先输出消息来自于哪条 `WebSocket` 连接:

```
log("HelloWorld::onMessage Websocket (%p)", ws);
```

然后判断接收到的数据是文本还是二进制数据, 采用不同的方法输出。

同理在 `onClose` 中打印出 `HelloWorld::onClose called`:

```
void HelloWorld::onClose(network::WebSocket* ws)
{
    log("HelloWorld::websocket instance (%p) closed.", ws);
};
```

在 `onError` 中打印出错误信息:

```
void HelloWorld::onError(network::WebSocket* ws, const network::
WebSocket:: ErrorCode& error)
{
    log("HelloWorld::websocket(%p)Error was fired, error code: %d",ws,
error);
}
```

(7) 运行程序, 如图 10-2 所示, 单击菜单 `Open WebSocket`, 查看控制台输出:

```
HelloWorld::Websocket (04066228) opened
HelloWorld::Websocket (04066138) opened
```

调用两次 `onOpen`, 说明打开了两个 `WebSocket`, 与 `ws://echo.websocket.org` 建立了连接:

```
HelloWorld::websocket(04066360)Error was fired, error code: 1  
HelloWorld::websocket instance (04066360) closed.
```

先调用 `onError`，再调用 `onClose`，说明与 `ws://invalid.url.com` 建立连接失败。

单击 **Send Text** 菜单，查看控制台输出：

```
HelloWorld::onMessage Websocket (04066138)  
HelloWorld::response text msg: Hello WebSocket, I'm a text message.,  
-842150450
```

调用 `onMessage` 函数，向服务器发送 `Hello WebSocket, I'm a text message.`后，又从服务器返回该信息。同理单击 **Send Binary** 菜单，也能在控制台看到接收到的数据：

```
HelloWorld::onMessage Websocket (04066228)  
HelloWorld::response bin msg: Hello WebSocket,'\0' I'm'\0' a'\0'  
binary'\0' message'\0'.''\0', -842150450
```

单击菜单 **Close WebSocket**，查看控制台输出：

```
HelloWorld::websocket instance (04066138) closed.  
HelloWorld::websocket instance (04066228) closed.
```

调用 `onClose` 函数，关闭成功。



图 10-2 WebSocket

11

第 11 章

物理引擎 Box2D

游戏是动态的，游戏里的可视元素绝大多数是运动的，一些简单的运动可以使用物理公式通过编程或编写脚本来实现，比如加速、减速运动可以使用“牛顿定律”实现。但是遇到比较复杂的物体碰撞、滚动、滑动或者弹跳的时候，通过编程的方法就比较困难了，比如赛车类游戏里的碰撞。

物理引擎出现之前，游戏中的“物理效果”只会按照预先设定好的规定执行。当物理引擎引入游戏中后，物体只需遵行物理参数，模拟真实世界中物体运动的规律来运动。这样一来，运动就是多样性的，由玩家决定运动方式。

本节讲解如何在游戏中使用物理引擎 Box2D 实现比较复杂的运动。包括 Box2D 简介，核心概念，如何使用 Box2D 实现碰撞、滚动等动作。

11.1 Box2D 简介

Box2D 是一个广泛应用于 2D 游戏的物理引擎，可以使游戏中的物体按照现实世界中物体的运动方式运动，让游戏更加真实，更加具有交互性，从而提高游戏的质量。该引擎通过用户设定的参数（如重力、密度、摩擦等）计算力度、方向、角度等。

Box2D 提供多种语言的实现，包括 C++、Flash、Java、C#、Python 五种版本。Box2D 包含了 8 种核心概念，本节只做简单介绍，下文会讲解如何使用这些概念。

- ◎ **刚体 (rigid body):** 一块十分坚硬的物质，它上面的任何两点之间的距离都是完全不变的。Box2D 使用物体 (body) 来代替刚体。
- ◎ **固定装置 (fixture):** 为物体绑定形状，添加材料属性，比如密度、摩擦、弹性等。
- ◎ **约束 (constraint):** 一个约束 (constraint) 就是消除物体自由度的物理连接。在 2D 模型中，一个物体有 3 个自由度。如果我们把一个物体钉在墙上（像摆锤那样），那我们就把它约束到了墙上。这样，此物体就只能绕着这个钉子旋转，所以这个约束消除了它 2 个自由度。
- ◎ **接触约束 (contact constraint):** 一个防止刚体穿透，以及用于模拟摩擦 (friction) 和恢复 (restitution) 的特殊约束。接触约束会自动被 Box2D 创建，不必人为创建。
- ◎ **关节 (joint):** 关节是一种用于把两个或多个物体固定到一起的约束。Box2D 支持的关节类型有旋转、棱柱、距离等。关节可以支持限制 (limits) 和马达 (motors)。
- ◎ **关节限制 (joint limit):** 一个关节限制 (joint limit) 限定了一个关节的运动范围。例如人类的胳膊肘只能做某一范围角度的运动。
- ◎ **关节马达 (joint motor):** 一个关节马达能依照关节的自由度来驱动所连接

的物体。例如，你可以使用一个马达来驱动一个肘的旋转。

◎ **世界 (world):** 一个物理世界就是物体、形状和约束相互作用的集合。

11.2 创建 Box2D 的 HelloWorld 项目

学习一项编程都从经典的 HelloWorld 项目开始，本节带领读者创建一个 HelloWorld 程序，了解使用 Box2D 编程的流程和方式。完整代码请查看代码清单 11-2。

该程序创建一个大的地面盒子和一个小的动态盒，但没有可见图像，只能在控制台看到文本输出。

11.2.1 创建一个世界

每个 Box2D 程序都要首选创建一个世界对象 (b2World object)。世界对象管理 Box2D 中的内存、物体和模拟等。你可以在堆或栈中创建 b2World 对象。

首先定义一个重力矢量，可以设置重力朝向侧面：

```
b2Vec2 gravity(0.0f, -10.0f);
```

接着我们创建世界对象：

```
b2World world(gravity);
```

现在我们就有了自己的物理世界。

11.2.2 创建一个地面物体

物体通常由以下步骤来创建。

- (1) 使用位置 (position)、阻尼 (damping) 等定义一个物体。
- (2) 使用世界对象创建物体。

(3) 使用几何结构、摩擦、密度等定义形状。

(4) 在物体上创建形状。

按照上面的步骤, 第一步, 创建地面体要先创建一个物体定义 (body definition), 通过物体定义指定地面体的初始位置:

```
b2BodyDef groundBodyDef;
groundBodyDef.position.Set(0.0f, -10.0f);
```

第二步, 将物体定义传给世界对象来创建物体。世界对象并不会保存物体定义的引用, 它会把数据粘贴到 **b2Body** 结构中。默认情况下创建的物体是静态的。

```
b2Body* groundBody = world.CreateBody(&groundBodyDef);
```

第三步, 我们定义一个多边形, 再使用 **SetAsBox** 简捷地把物体形状规定为一个矩形形状, 矩形的中点就位于父物体的原点上。

```
b2PolygonShape groundBox;
groundBox.SetAsBox(50.0f, 10.0f);
```

其中, **SetAsBox** 函数接收了半个宽度和半个高度, 这样物体形状就是 100 个单位宽 (x 轴) 以及 20 个单位高 (y 轴) 的矩形。**Box2D** 使用米、千克和秒来作为单位。

第四步, 我们在物体上创建物体多边形, 以完成静态物体。这里我们使用了物体材料属性的默认值, 不用再创建物体材料定义, 直接把形状传递给 **b2Body**。第二个参数是物体的密度, 单位是千克每平方米。静态物体的质量为 0, 所以这里传递 0 进去。

```
groundBody->CreateFixture(&groundBox, 0.0f);
```

当使用材料确定物体的形状时, 形状的坐标就成为物体的本地坐标。所以当物体移动时, 形状也会移动。材料的转变继承自父类物体, 不能独立于父类物体而存在, 所以不能在物体上移动或修改形状。原因很简单, 形状可以变的物体不是刚性体, 但是 **Box2D** 是一个基于刚性体的引擎, 很多设定都是基于刚性体的, 如果违反了这一原则, 很多东西都会被打破。

11.2.3 创建一个动态物体

现在已经创建了一个静态物体,我们可以使用同样的方法来创建一个动态物体。除了尺寸之外的主要区别是,我们必须为动态物体设置质量性质。

(1) 用 `CreateBody` 创建物体,默认情况下物体是静态的,所以需要设定 `b2BodyType` 来使物体成为动态的:

```
b2BodyDef bodyDef;
bodyDef.type = b2_dynamicBody;
bodyDef.position.Set(0.0f, 4.0f);
b2Body* body = world.CreateBody(&bodyDef);
```

注意: 当想要物体在力的作用下移动时,必须设定物体的类型为 `b2_dynamicBody`。

(2) 创建并添加一个多边形形状到物体上:

```
b2PolygonShape shapeDef;
shapeDef.SetAsBox(1.0f, 1.0f);
```

(3) 定义 `b2FixtureDef`, 密度设置为 `1.0f`。默认密度为 0, 质量为 0, 生成的就静态物体。设置密度为非 0, 物体就有质量, 创建的为动态物体。阻尼设置为 `0.3`。

```
b2FixtureDef fixtureDef;
fixtureDef.shape = &dynamicBox;
fixtureDef.density = 1.0f;
fixtureDef.friction = 0.3f;
```

(4) 使用定义好的 `b2FixtureDef` 创建物体,这会自动更新物体的质量。也可以给 `b2FixtureDef` 设置其他属性,这些都会影响到物体的质量。

```
body->CreateFixture(&fixtureDef);
```

这就是初始化过程,现在我们已经准备好开始模拟了。

11.2.4 模拟 (Box2D 的) 世界

我们已经创建了一个地面物体和一个动态物体。只需要再考虑几个问题,剩下

的就交给牛顿去处理吧。

Box2D 使用了一个叫作积分器 (integrator) 的计算算法, 积分器 (integrator) 在离散的时间点模拟物理方程式, 它将与游戏动画循环一同运行。所以我们需要为 Box2D 选取一个时间步, 通常来说游戏物理引擎需要至少 60Hz 的速度, 也就是 1/60 的时间步。你可以使用更大的时间步, 但是你必须更加小心地为你的世界调整定义。我们也不喜欢时间步变化得太大, 变化的时间步会产生变化的结果, 在调试时这会给我们带来很大的困难, 所以不要把时间步关联到帧频 (除非你真的必须这样做)。直接设置一个定值就可以了:

```
float32 timeStep = 1.0f / 60.0f;
```

除了积分器之外, Box2D 中还有约束求解器 (constraint solver)。约束求解器用于解决模拟中的所有约束, 一次一个。单个的约束会被完美地求解, 然而当我们求解一个约束的时候, 我们就会稍微耽误另一个。要得到良好的解, 我们需要迭代所有约束多次。

约束求解器 (constraint solver) 包括两个层面: 速度和位置。在速度层面, 约束求解器计算作用在物体上的力, 以便使物体正确移动; 在位置层面, 约束求解调整物体的位置, 来减少重叠和关节脱落。每个层面都有自己的迭代次数。此外, 如果误差很小, 位置可以提前退出迭代。

建议速度迭代次数是 10 次, 位置迭代次数是 3 次。你可以按自己的需要去调整这个数, 但要记得它是速度与质量之间的平衡。更少的迭代会增加性能并降低精度, 同样地, 更多的迭代会减少性能但提高模拟质量。这是我们选择的迭代次数:

```
int32 velocityIterations = 6;  
int32 positionIterations = 2;
```

注意: 时间步和迭代数是完全无关的。一个迭代并不是一个子步。一次迭代就是在时间步之中的单次遍历所有约束, 你可以在单个时间步内多次遍历约束。

现在我们可以开始模拟循环了, 在游戏中模拟循环应该并入游戏循环。每次循环都应该调用 `b2World::Step`, 通常调用一次就够了, 这取决于帧频以及物理时间步。

这个 Hello World 程序设计得非常简单, 所以它没有图形输出。由于完全没有输出, 代码会打印出动态物体的位置以及旋转角度。Ok, 这就是模拟 1 秒内 60 个

时间步的循环:

```
for (int32 i = 0; i < 60; ++i)
{
    world.Step(timeStep, velocityIterations, positionIterations);
    b2Vec2 position = body->GetPosition();
    float32 angle = body->GetAngle();
    printf("%.2f %.2f %.2f\n", position.x, position.y, angle);
}
```

输出显示了动态物体向下掉落到地面上的过程, 输出应该是这样的:

```
0.00 4.00 0.00
0.00 3.99 0.00
0.00 3.98 0.00
...
0.00 1.25 0.00
0.00 1.13 0.00
0.00 1.01 0.00
```

11.2.5 清理工作

当一个世界对象超出它的作用域, 或通过指针将其删除 (`delete`) 时, 所有物体和关节的内存都会被释放。并且应该将物体、形状或关节的指针都清零, 因为它们已经无效了。

11.3 世界 b2World

11.3.1 b2World 简介

b2World 类包含着物体和关节。它管理着模拟的方方面面, 并允许异步查询 (就像 AABB 查询)。与 Box2D 的大部分交互都将通过 b2World 对象来完成。

创建世界非常简单, 只要先定义一个重力即可:

```
b2World* world = new b2World(gravity);
```

删除世界对象使用 C++ 的 `delete`:

```
delete world;
```

11.3.2 世界常用功能

世界可以创建和摧毁物体和关节，它提供了相关的工厂类。世界还具有其他管理和驱动功能。

1. 驱动模拟

世界类能够驱动模拟。需要指定一个时间步、一个速度迭代次数和一个位置迭代次数。

```
float timeStep = 1.0f / 60.f;
int velocityIterations = 8;
int positionIterations = 1;
myWorld->Step(timeStep, velocityIterations, positionIterations);
```

时间步完成之后，就可以调查物体和关节的信息了。最常见情况是需要获取物体的位置这样才能更新角色的位置并渲染它们。我们可以在游戏循环的任何地方执行时间步，但你应该先明白事情发生的顺序。例如，如果想要在一帧中得到新物体的碰撞结果，必须在时间步之前创建物体。

时间步最好是一个固定的值。使用大一些的时间步能在低帧率的情况下提升性能。但通常情况下应该使用一个不大于 1/30 秒的时间步。1/60 的时间步通常会呈现一个高质量的模拟。

迭代次数控制了约束求解器会遍历多少次世界中的接触和关节。迭代次数越多，模拟效果越好，但不要使用小频率大迭代数。60Hz 和 10 次迭代远好于 30Hz 和 20 次迭代。

执行 `step` 后，应该清除作用在物体上的任何力。可以使用 `b2World` 的 `ClearForces` 函数清除：

```
myWorld->ClearForces();
```

2. 扫描世界

世界是物体和关节的容器。你可以获取世界中所有物体和关节并遍历它们。例如，下面这段代码会唤醒世界中的所有物体：

```
for (b2Body* b = myWorld->GetBodyList(); b; b = b->GetNext())
{
    b->SetAwake(true);
}
```

需要注意，真正编码时会遇到许多问题，例如，下面的代码是错误的：

```
for (b2Body* b = myWorld->GetBodyList(); b; b = b->GetNext())
{
    GameActor* myActor = (GameActor*)b->GetUserData();
    if (myActor->IsDead())
    {
        myWorld->DestroyBody(b); // ERROR: now GetNext returns garbage.
    }
}
```

在一个物体被摧毁之前一切都很顺利。一旦一个物体被摧毁了，它的 `next` 指针就变得非法，所以 `b2Body::GetNext()` 就会返回垃圾。解决方法是在摧毁之前复制 `next` 指针。

```
b2Body* node = myWorld->GetBodyList();
while (node)
{
    b2Body* b = node;
    node = node->GetNext();

    GameActor* myActor = (GameActor*)b->GetUserData();
    if (myActor->IsDead())
    {
        myWorld->DestroyBody(b);
    }
}
```

这能安全地摧毁当前物体。然而，你可能想要调用一个游戏的函数来摧毁多个物体，这时需要十分小心。解决方案取决于具体应用，但在此给出一种方法：

```

b2Body* node = myWorld->GetBodyList();
while (node)
{
    b2Body* b = node;
    node = node->GetNext();

    GameActor* myActor = (GameActor*)b->GetUserData();
    if (myActor->IsDead())
    {
        bool otherBodiesDestroyed = GameCrazyBodyDestroyer(b);
        if (otherBodiesDestroyed)
        {
            node = myWorld->GetBodyList();
        }
    }
}

```

很明显要保证这个能正确工作，GameCrazyBodyDestroyer 对它都摧毁了什么必须要诚实。

3. AABB 查询

有时需要求出一个区域内的所有形状。b2World 类为此使用了 broad-phase 数据结构，提供了一个 $\log(N)$ 的快速方法。你提供一个世界坐标的 AABB，而 b2World 会返回一个所有大概相交于此 AABB 的形状之数组。这不是精确的，因为函数实际上返回那些 AABB 与规定之 AABB 相交的形状。例如，下面的代码找到所有大概与指定 AABB 相交的形状并唤醒所有关联的物体。

```

class MyQueryCallback : public b2QueryCallback
{
public:
    bool ReportFixture(b2Fixture* fixture)
    {
        b2Body* body = fixture->GetBody();
        body->SetAwake(true);

        // Return true to continue the query.
        return true;
    }
}

```

```

    }
};
...
MyQueryCallback callback;
b2AABB aabb;
aabb.lowerBound.Set(-1.0f, -1.0f);
aabb.upperBound.Set(1.0f, 1.0f);
myWorld->Query(&callback, aabb);

```

b2World 还能做许多其他事情，请读者按上面的步骤自己摸索使用。

11.4 物体 b2Body

11.4.1 b2Body 简介

物体具有位置和速度。可以应用力、扭矩和冲量到物体。物体可以是静态的或动态的，下面列出物体的几种类型。

1. b2_staticBody

静态物体永远不会移动，并且不会与其他静态物体发生碰撞。Box2D 内部，质量和逆质量的值为零，用户可以手动移动静态物体。静态物体的速度为零，静态物体不会与其他静态或运动物体发生碰撞。

2. b2_kinematicBody

运动物体在模拟中根据它的速度移动，运动物体不会对力做出反应，它们可以被用户手动移动。但是通常情况下，通过设置速度来移动一个运动物体。运动物体好像有无限的质量，但是在 Box2D 中它们的质量为零。运动物体不会与其他静态或运动物体发生碰撞。

3. b2_dynamicBody

动态物体被完全模拟，用户可以手动移动动态物体。但通常动态物体根据作用

力运动。动态物体可以和所有类型的物体碰撞。动态物体具有有限且不为零的质量，如果尝试将动态物体的质量设置为零，那么它的质量会自动变成一千克，且不会再转动。

物体是形状的主干，物体携带形状在世界中运动。在 Box2D 中物体总是刚体，这意味着同一刚体上的两个形状永远不会相对移动。

通常你会保存所有你所创建的物体的指针，这样你就能查询物体的位置，并在图形实体中更新它的位置。另外在不需要它们的时候也需要通过它们的指针摧毁它们。

11.4.2 物体定义

在创建物体之前你需要创建一个物体定义（b2BodyDef）。你可以把物体定义创建在栈上，也可以在你的游戏数据结构中保存它们。

Box2D 会从物体定义中复制出数据，它不会保存到物体定义的指针。这意味着可以循环使用一个物体定义去创建多个物体。让我们看一些物体定义的关键成员。

1. 物体类型

正如上一节所说，物体有三种类型：静态、运动和动态。在创建物体时就应该指定物体的类型。

```
bodyDef.type = b2_dynamicBody;
```

设置物体类型是强制性的。

2. 位置和角度

物体定义提供了一个在创建时初始化位置的机会，这要比在世界原点创建物体而后移动到某个位置具有更好的性能。

一个物体上主要有两个关键的点。其中一个是一个物体的原点，形状和关节都相对于物体的原点而被附加。另一个点是物体的质心。质心取决于物体上形状的质量分

配，或显式地由 `b2MassData` 设置。Box2D 内部的许多计算都要使用物体的质心，例如 `b2Body` 会存储质心的线速度。

当你构造物体定义的时候，可能你并不知道质心在哪里，因此你会指定物体的原点。你可能也会以弧度指定物体的角度，角度并不受质心位置的影响。如果随后你改变了物体的质量性质，那么质心也会随之移动，但是原点以及物体上的形状和关节都不会改变。

```
bodyDef.position.Set(0.0f, 2.0f); // the body's origin position.
bodyDef.angle = 0.25f * b2_pi; // the body's angle in radians.
```

3. 阻尼

阻尼用于减小物体在世界中的速率。阻尼与摩擦是不同的，因为摩擦仅在物体有接触的时候才会发生，而阻尼的模拟要比摩擦方便多了。然而，阻尼并不能取代摩擦，往往这两个效果需要同时使用。

阻尼参数的范围可以在 0 到无穷之间，0 的就是没有阻尼，无穷就是满阻尼。通常来说，阻尼的值应该在 0 到 0.1 之间，笔者通常不使用线性阻尼，因为它会使物体看起来发飘。

```
bodyDef.linearDamping = 0.0f;
bodyDef.angularDamping = 0.01f;
```

阻尼相似于稳定性与性能，阻尼值较小的时候阻尼效应几乎不依赖于时间步，而阻尼值较大的时候，阻尼效应将随着时间步而变化。如果使用固定的时间步（推荐）这就不是问题了。

4. 休眠参数

模拟物体的成本是高昂的，所以如果物体更少，那模拟的效果就能更好。当一个物体停止了运动时，我们喜欢停止去模拟它。

当 Box2D 确定一个物体（或一组物体）已经停止移动时，物体就会进入休眠状态，消耗很小的 CPU 开销。如果一个醒着的物体接触到了一个休眠中的物体，那么

休眠中的物体就会醒来。当物体上的关节或触点被摧毁的时候，它们同样会醒来。也可以手动地唤醒物体。

通过物体定义，可以指定一个物体是否可以休眠，或者创建一个休眠的物体。

```
bodyDef.allowSleep = true;  
bodyDef.awake = false;
```

5. 子弹

有的时候，在一个时间步内可能会有大量的刚体同时运动。如果一个物理引擎没有处理好大幅度运动的问题，就可能会看见一些物体错误地穿过了彼此。这种效果被称为隧道效应（tunneling）。

默认情况下，Box2D 会通过连续碰撞检测（CCD）来防止动态物体穿越静态物体，这是通过从形状的旧位置到新位置的扫描来完成的。引擎会查找扫描中的新碰撞，并为这些碰撞计算碰撞时间（TOI）。物体会先被移动到它们的第一个 TOI，然后一直模拟到原时间步的结束。如果有必要这个步骤会重复执行。

一般 CCD 不会应用于动态物体之间，这是为了保持性能。在一些游戏环境中你需要在动态物体上也使用 CCD，譬如，你可能想用一颗高速的子弹去射击薄壁。没有 CCD，子弹就可能会隧穿薄壁。

高速移动的物体在 Box2D 被称为子弹（bullet），你需要按照游戏的设计来决定哪些物体是子弹。如果你决定一个物体应该按照子弹去处理，使用下面的设置：

```
bodyDef.bullet = true;
```

子弹开关只影响动态物体。

CCD 的成本是昂贵的，所以你可能不希望所有运动物体都成为子弹。所以 Box2D 默认只在动态物体和静态物体之间使用 CCD，这是防止物体逃脱游戏世界的一个有效方法。然而，有时一些高速移动的物体需要一直使用 CCD。

11.4.3 创建物体

物体的创建和摧毁是由世界类提供的物体工厂来完成的。这使得世界可以通过

一个高效的分配器来创建物体，并且把物体加入世界数据结构中。

```
b2Body* dynamicBody = myWorld->CreateBody(&bodyDef);
... do stuff ...
myWorld->DestroyBody(dynamicBody);
dynamicBody = NULL;
```

注意：永远不要使用 `new` 或 `malloc` 来创建物体，否则世界不会知道这个物体的存在，并且物体也不会被适当地初始化。

Box2D 并不保存物体定义的引用，也不保存其任何数据（除了用户数据指针），所以你可以创建临时的物体定义，并复用同样的物体定义。

Box2D 允许你通过删除 `b2World` 对象来摧毁物体，它会为你做所有的清理工作。然而，你必须小心地处理那些已失效的物体指针。

当物体销毁时，形状和关节也会被销毁。

11.4.4 使用物体

在创建完一个物体之后，可以对它进行许多操作。其中包括设置质量、访问其位置和速度、施加力，以及转换点和向量。

1. 质量数据

物体有质量、质点和转动惯量。静态物体的质量和转动惯量都为零。当物体有一个固定的旋转时，转动惯量也是零。

通常情况当给物体添加一个形状时，会自动计算物体的质量。但也可以在运行时设定物体的质量：

```
void SetMassData(const b2MassData* data);
```

当直接设定一个物体的质量后，有可能又希望物体根据形状和材质自动计算质量，这时就可以这样做：

```
void ResetMassData();
```

通过以下这些函数可以获得物体的质量数据：

```
float32 GetMass() const;
float32 GetInertia() const;
const b2Vec2& GetLocalCenter() const;
void GetMassData(b2MassData* data) const;
```

2. 状态信息

物体的状态含有多个方面，通过这些函数可以访问这些状态数据：

```
void SetType(b2BodyType type);
b2BodyType GetType();

void SetBullet(bool flag);
bool IsBullet() const;

void SetSleepingAllowed(bool flag);
bool IsSleepingAllowed() const;

void SetAwake(bool flag);
bool IsAwake() const;

void SetActive(bool flag);
bool IsActive() const;

void SetFixedRotation(bool flag);
bool IsFixedRotation() const;
```

3. 位置和速度

可以访问一个物体的位置和角度，这在渲染相关游戏角色时很常用。也可以设置位置，尽管这不怎么常用。

```
bool SetTransform(const b2Vec2& position, float32 angle);
const b2Transform& GetTransform() const;
const b2Vec2& GetPosition() const;
float32 GetAngle() const;
```

可以访问世界坐标的质心。许多 Box2D 内部的模拟都使用质心，然而，通常不必访问它。取而代之，你一般应该关心物体变换。

```
const b2Vec2& GetWorldCenter() const;
const b2Vec2& GetLocalCenter() const;
```

可以访问线速度与角速度，线速度是对于质心所言的。因此，如果物体的质量改变时，线速度也会改变。

11.5 固定装置 b2FixtureDef

11.5.1 b2FixtureDef 简介

由于形状不能与物体关联起来，所以 Box2D 提供了 b2Fixture 类把形状绑定到物体上。一个物体可以有一个或多个固定装置。具有多个固定装置的物体被称作组合体 (*compound body*)。

固定装置包含下面这些信息。

- ◎ 一个单一的形状。
- ◎ 广义代理。
- ◎ 密度、摩擦和恢复原状。
- ◎ 碰撞过滤标记。
- ◎ 指向父级物体的指针。
- ◎ 用户数据。
- ◎ 传感器标记。

11.5.2 创建 b2FixtureDef

要创建一个固定装置，需要先初始化一个固定装置定义，然后把该定义传递给

父级物体：

```
b2FixtureDef fixtureDef;
fixtureDef.shape = &myShape;
fixtureDef.density = 1.0f;
b2Fixture* myFixture = myBody->CreateFixture(&fixtureDef);
```

也可以销毁物体上的固定装置：

```
myBody->DestroyFixture(myFixture);
```

创建固定装置时会定义物体的材料属性。

1. 密度

密度用来计算物体的质量，密度可以为 0 或者负值。通常情况下你应该为你的所有物体定义类似的密度，这将提供游戏性能。

当设置物体密度时，物体的质量不好自动变化，需要手动更新一下：

```
fixture->SetDensity(5.0f);
body->ResetMassData();
```

2. 摩擦

摩擦用来实现物体之间相互滑动。Box2D 支持静态摩擦和动态摩擦，但都使用同样的参数进行设置。摩擦在 Box2D 中会被正确地模拟，并且摩擦力的强度与正交力（称之为库仑摩擦）成正比。摩擦参数经常会设置在 0 到 1 之间，0 意味着没有摩擦，1 会产生强摩擦。当计算两个形状之间的摩擦时，Box2D 必须联合两个形状的摩擦参数，这是通过以下公式完成的：

```
float32 friction;
friction = sqrtf(fixtureA->friction * fixtureB->friction);
```

3. 恢复

恢复可以使对象弹起，想象一下，在桌面上方丢下一个小球。恢复的值通常设置在 0 到 1 之间，0 的意思是小球不会弹起，这称为非弹性碰撞；1 的意思是小球的

速度会得到精确的反射，这称为完全弹性碰撞。恢复是通过这样的公式计算的：

```
float32 restitution;
restitution = b2Max(fixtureA->restitution, fixtureB->restitution);
```

当一个形状发生多碰撞时，恢复会被近似地模拟。这是因为 Box2D 使用了迭代求解器。当冲撞速度很小时，Box2D 也会使用非弹性碰撞，这是为了防止抖动。

4. 筛选

碰撞筛选是一个防止某些形状发生碰撞的系统。譬如说，你创造了一个骑自行车的角色。你希望自行车与地形之间有碰撞，角色与地形有碰撞，但你不希望角色和自行车之间发生碰撞（因为它们必须重叠）。Box2D 通过种群和组支持了这样的碰撞筛选。

Box2D 支持 16 个种群，对于任何一个形状都可以指定它属于哪个种群。还可以指定这个形状可以和其他哪些种群发生碰撞。可以在一个多人游戏中指定玩家之间不会碰撞，怪物之间也不会碰撞，但是玩家和怪物会发生碰撞。这是通过掩码来完成的，例如：

```
playerFixtureDef.filter.categoryBits = 0x0002;
monsterFixtureDef.filter.categoryBits = 0x0004;
playerFixtureDef.filter.maskBits = 0x0004;
monsterFixtureDef.filter.maskBits = 0x0002;
```

下面是发生碰撞的规则：

```
uint16 catA = fixtureA.filter.categoryBits;
uint16 maskA = fixtureA.filter.maskBits;
uint16 catB = fixtureB.filter.categoryBits;
uint16 maskB = fixtureB.filter.maskBits;

if ((catA & maskB) != 0 && (catB & maskA) != 0)
{
    // fixtures can collide
}
```

碰撞组可以让你指定一个整数的组索引。你可以让同一个组的所有形状总是相

互碰撞（正索引）或永远不碰撞（负索引）。组索引通常用于一些以某种方式关联的事物，就像自行车的那些部件。在下面的例子中，`shape1` 和 `shape2` 总是碰撞，而 `shape3` 和 `shape4` 永远不会碰撞：

```
shape1Def.filter.groupIndex = 2;
shape2Def.filter.groupIndex = 2;
shape3Def.filter.groupIndex = -8;
shape4Def.filter.groupIndex = -8;
```

不同组索引之间形状的碰撞会按照种群和掩码来筛选。换句话说，组筛选比种群筛选有更高的优先权。

在 **Box2D** 中的其他碰撞筛选如下：

- ◎ 静态物体只能与动态物体发生碰撞。
- ◎ 永动物体只能与动态物体发生碰撞。
- ◎ 同一个物体上的形状之间永远不会发生碰撞。
- ◎ 可以有选择地启用或禁止由关节连接之物体上的形状之间是否碰撞。

11.6 关节

11.6.1 关节简介

关节的作用是把物体约束到世界，或约束到其他物体上。在游戏中的典型例子是木偶、跷跷板和滑轮。关节可以用许多种不同的方法结合起来，创造出有趣的运动。

有些关节提供了限制（`limit`），以便你控制运动范围。有些关节还提供了马达（`motor`），它可以以指定的速度驱动关节，直到你指定了更大的力或扭矩。

关节马达有许多不同的用途。可以使用关节来控制位置，只要提供一个与目标之距离成正比例的关节速度即可。还可以模拟关节摩擦：将关节速度置零，并且提

供一个小的、但有效的最大力或扭矩；那么马达就会努力保持关节不动，直到负载变得过大。

11.6.2 关节定义

各种关节类型都派生自 `b2JointDef`。所有关节都连接两个不同的物体，其中一个可能是静态物体。但是静态物体之间的关节是没有任何效果的，如果定义，也只会占用内存。

可以为任何一种关节指定用户数据。还可以提供一个标记，用于预防相连的物体发生碰撞。实际上，这是默认行为，你可以设置布尔值 `collideConnected` 来允许相连的物体碰撞。

很多关节定义需要提供一些几何数据。一个关节常常需要一个锚点（`anchor point`）来定义，这是固定于相接物体中的点。在 `Box2D` 中这些点需要在局部坐标系中指定，这样，即便当前物体的变化违反了关约束，关节还是可以被指定——在游戏存取进度时这经常会发生。另外，有些关节定义需要默认的物体之间的相对角度，这样才能通过关节限制或固定的相对角来正确地约束旋转。

初始化几何数据可能有些乏味。所以很多关节提供了初始化函数，消除了大部分工作。然而，这些初始化函数通常只应用于原型，在产品代码中应该直接地定义几何数据。这能使关节行为更加稳固。其余的关节定义数据依赖于关节的类型，下面我们来介绍它们。

11.6.3 创建关节

关节是通过世界的工厂方法来创建和摧毁的。

注意：不要试图在栈上创建物体或关节，也不要使用 `new` 或 `malloc` 在堆上创建。物体以及关节必须要通过 `b2World` 类的方法来创建或摧毁。

这是一个关于旋转关节生命期的例子：

```
b2RevoluteJointDef jointDef;
```

```

jointDef.bodyA = myBodyA;
jointDef.bodyB = myBodyB;
jointDef.anchorPoint = myBodyA->GetCenterPosition();
b2RevoluteJoint* joint = (b2RevoluteJoint*)myWorld->CreateJoint
(&jointDef);
... do stuff ...
myWorld->DestroyJoint(joint);
joint = NULL;

```

摧毁之后将指针清零总是一个好的方式。如果你试图使用它，程序也会以可控的方式崩溃。

关节的生命期并不简单。

注意：当物体被摧毁时其上的关节也会摧毁。

11.6.4 关节类型和使用关节

在许多模拟中，关节被创建之后便不再被访问了。然而，关节中包含着很多有用的数据，使你可以创建出丰富的模拟。

首先，你可以在关节上得到物体、锚点及用户数据。

```

b2Body* GetBodyA();
b2Body* GetBodyB();
b2Vec2 GetAnchorA();
b2Vec2 GetAnchorB();
void* GetUserData();

```

所有的关节都有反作用力和反扭矩，反作用力应用于 **body2** 的锚点之上。可以用反作用力来折断关节，或引发其他游戏事件。这些函数可能需要做一些计算，所以不要在不需要的时候调用它们：

```

b2Vec2 GetReactionForce();
float32 GetReactionTorque();

```

下面讲解下关节的类型和具体使用方式。

1. 距离关节

距离关节是最简单的关节之一，它描述了两个物体上的两个点之间的距离应该是常量。当指定一个距离关节时，两个物体必须已在应有的位置上。随后，指定两个世界坐标中的锚点。第一个锚点连接到物体 1，第二个锚点连接到物体 2。这些点隐含了距离约束的长度，如图 11-1 所示。

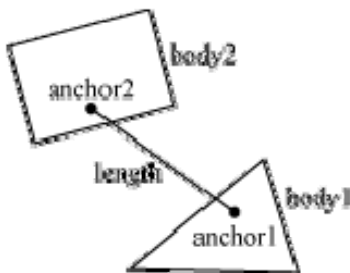


图 11-1 距离关节

这是一个距离关节定义的例子，在此我们允许了碰撞。

```
b2DistanceJointDef jointDef;
jointDef.Initialize(myBodyA,      myBodyB,      worldAnchorOnBodyA,
worldAnchorOnBodyB);
jointDef.collideConnected = true;
```

2. 旋转关节

一个旋转关节会强制两个物体共享一个锚点，即所谓铰接点。旋转关节只有一个自由度：两个物体的相对旋转。这称之为关节角，如图 11-2 所示。

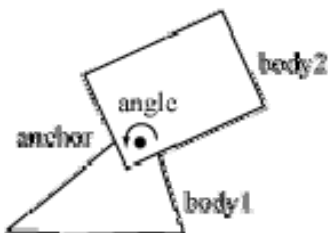


图 11-2 旋转关节

要指定一个旋转关节，需要提供两个物体以及一个世界坐标的锚点。初始化函数会假定物体已经在应有位置了。

在此例中，两个物体被旋转关节连接于第一个物体之质心。

```
b2RevoluteJointDef jointDef;
jointDef.Initialize(bodyA, bodyB, myBodyA->GetWorldCenter());
jointDef.lowerAngle = -0.5f * b2_pi; // -90 degrees
jointDef.upperAngle = 0.25f * b2_pi; // 45 degrees
jointDef.enableLimit = true;
jointDef.maxMotorTorque = 10.0f;
jointDef.motorSpeed = 0.0f;
jointDef.enableMotor = true;
```

可以通过下面这些方法获得关节的角度、速度、马达等：

```
float32 GetJointAngle() const;
float32 GetJointSpeed() const;
float32 GetMotorTorque() const;
```

可以更新马达的参数：

```
void SetMotorSpeed(float32 speed);
void SetMaxMotorTorque(float32 torque);
```

可以使用关节马达来跟踪仪想要的关节角度：

```
. Game Loop Begin ...
float32 angleError = myJoint->GetJointAngle() - angleTarget;
float32 gain = 0.1f;
myJoint->SetMotorSpeed(-gain * angleError);
... Game Loop End ...
```

通常来讲增益参数不应过大，否则关节可能会变得不稳定。

3. 棱柱关节

棱柱关节（prismatic joint）允许两个物体沿指定轴相对移动，它会阻止相对旋转。因此，移动关节只有一个自由度，如图 11-3 所示。

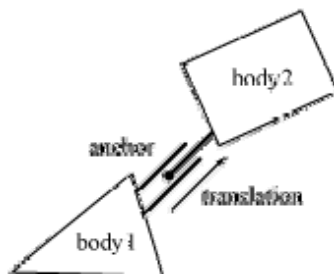


图 11-3 棱柱关节

移动关节的定义有些类似于旋转关节；只是转动角度换成了平移，扭矩换成了力。以这样的类比，我们来看一个带有关节限制以及马达摩擦的移动关节定义：

```
b2PrismaticJointDef jointDef;
b2Vec2 worldAxis(1.0f, 0.0f);
jointDef.Initialize(myBodyA, myBodyB, myBodyA->GetWorldCenter(),
worldAxis);
jointDef.lowerTranslation = -5.0f;
jointDef.upperTranslation = 2.5f;
jointDef.enableLimit = true;
jointDef.maxMotorForce = 1.0f;
jointDef.motorSpeed = 0.0f;
jointDef.enableMotor = true;
```

使用棱柱关节类似使用旋转关节，下面是相应的成员函数：

```
float32 GetJointTranslation() const;
float32 GetJointSpeed() const;
float32 GetMotorForce() const;
void SetMotorSpeed(float32 speed);
void SetMotorForce(float32 force);
```

4. 滑轮关节

滑轮关节用于创建理想的滑轮，它将两个物体接地（ground）并连接到彼此。这样，当一个物体升起时，另一个物体就会下降。滑轮的绳子长度取决于初始时的状态。

```
length1 + length2 == constant
```

还可以提供一个系数（ratio）来模拟滑轮组，这会使滑轮一侧的运动比另一侧要快。同时，一侧的约束力也比另一侧要小。你也可以用这个来模拟机械杠杆（mechanical leverage）。

```
length1 + ratio * length2 == constant
```

举个例子，如果系数是 2，那么 length1 的变化会是 length2 的两倍。另外连接 body1 的绳子的约束力将会是连接 body2 绳子的一半，如图 11-4 所示。

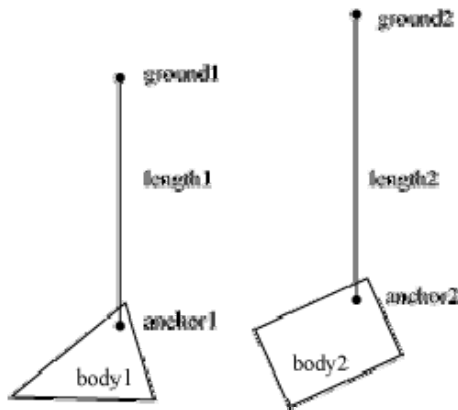


图 11-4 滑轮关节

当滑轮的一侧完全展开时，另一侧的绳子长度为零，这可能会出问题。此时，约束方程将变得很糟糕。因此，滑轮关节约束了每一侧的最大长度。另外出于游戏原因你可能也希望控制这个最大长度。最大长度能提高稳定性，以及提供更多的控制。

```
b2Vec2 anchor1 = myBody1->GetWorldCenter();
b2Vec2 anchor2 = myBody2->GetWorldCenter();
b2Vec2 groundAnchor1(p1.x, p1.y + 10.0f);
b2Vec2 groundAnchor2(p2.x, p2.y + 12.0f);
float32 ratio = 1.0f;
b2PulleyJointDef jointDef;
jointDef.Initialize(myBody1, myBody2, groundAnchor1, groundAnchor2,
anchor1, anchor2, ratio);
```

使用下面的方法可以获取当前的长度：

```
float32 GetLengthA() const;
float32 GetLengthB() const;
```

5. 齿轮关节

如果想要创建复杂的机械装置，可能需要齿轮。原则上，在 Box2D 中可以用复杂的形状来模拟轮齿，但这并不十分高效，而且这样的工作可能有些乏味。另外，还得小心地排列齿轮，保证轮齿能平稳地啮合。Box2D 提供了一个创建齿轮的更简单的方法：齿轮关节，如图 11-5 所示。

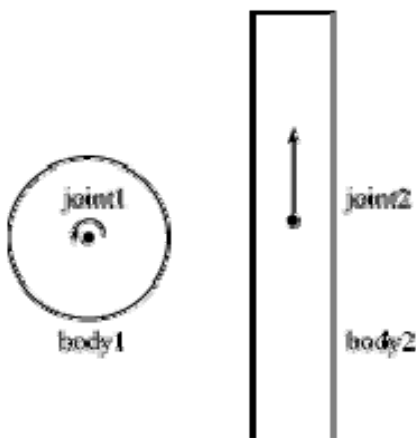


图 11-5 齿轮关节

类似于滑轮的系数，你可以指定一个齿轮系数（ratio），齿轮系数可以为负。另外值得注意的是，当一个是旋转关节（有角度的）而另一个是移动关节（平移）时，齿轮系数是长度或长度分之一。

```
coordinate1 + ratio * coordinate2 == constant
```

下面定义了一个齿轮关节，myBodyA 和 myBodyB 可以是任何两个物体，只要不是同一个物体。

```
b2GearJointDef jointDef;
jointDef.bodyA = myBodyA;
jointDef.bodyB = myBodyB;
jointDef.joint1 = myRevoluteJoint;
```

```
jointDef.joint2 = myPrismaticJoint;  
jointDef.ratio = 2.0f * b2_pi / myLength;
```

注意，齿轮关节依赖于两个其他关节，这是脆弱的：当其他关节被删除了会发生什么？

注意：齿轮关节总应该先于旋转或移动关节被删除，否则代码将会由于齿轮关节中的无效关节指针而导致崩溃。另外齿轮关节也应该在任何相关物体被删除之前删除。

11.7 接触

11.7.1 接触简介

接触（contact）是由 Box2D 创建的用于管理形状间碰撞的对象。接触有不同的种类，它们派生自 `b2Contact`，用于管理不同类型形状之间的接触。例如，有管理多边形之间碰撞的类，有管理圆形之间碰撞的类。通常这对你并不重要，笔者只是想或许你愿意了解一些。

这里是 Box2D 中的一些与碰撞有关的术语。

1. 触点（contact point）

两个形状相互接触的点。实际上当物体的表面相接触时可能会有一定接触区域，在 Box2D 则近似地以少数点来接触。

2. 接触向量（contact normal）

从 `shape1` 指向 `shape2` 的单位向量。

3. 接触分隔（contact separation）

分隔相反于穿透，当形状相重叠时，分隔为负。可能以后的 Box2D 版本中会以

正隔离来创建触点，所以当有触点的报告时你可能会检查符号。

4. 法向力 (normal force)

Box2D 使用了一个迭代接触求解器，并会以触点保存结果。你可以安全地使用法向力来判断碰撞强度。例如，可以使用这个力来引发破碎，或者播放碰撞的声音。

5. 切向力 (tangent force)

它是接触求解器关于摩擦力的估计量。

6. 接触标识 (contact ids)

Box2D 会试图利用一个时间步中的触点压力 (contact force) 结果来推测下一个时间步中的情况。接触标识用于匹配跨越时间步的触点，它包含了几何特征索引以便区分触点。

当两个形状的 AABB 重叠时，接触就被创建了。有时碰撞筛选会阻止接触的建立，有时尽管碰撞已筛选了 Box2D 还是须要创建一个接触，这种情况下它会使用 `b2NullContact` 来防止碰撞的发生。当 AABB 不再重叠之后接触会被摧毁。

也许你会皱起眉头，为了没有发生实际碰撞的形状（只是它们的 AABB）却创建了接触。好吧，的确是这样的，这是一个“鸡或蛋”的问题。我们并不知道是否需要一个接触，除非我们创建一个接触去分析碰撞。如果形状之间没有发生碰撞，我们需要正确地删除接触，或者，我们可以一直等到 AABB 不再重叠。Box2D 选择了后面这个方法。

11.7.2 接触监听器

通过实现 `b2ContactListener` 就可以接受接触数据。接触监听器支持 4 种事件：`begin`、`end`、`pre-solve` 和 `post-solve`。

```
class MyContactListener : public b2ContactListener
{
public:
```

```

void BeginContact(b2Contact* contact)
{ /* handle begin event */ }

void EndContact(b2Contact* contact)
{ /* handle end event */ }

void PreSolve(b2Contact* contact, const b2Manifold* oldManifold)
{ /* handle pre-solve event */ }
void PostSolve(b2Contact* contact, const b2ContactImpulse* impulse)
{ /* handle post-solve event */ }
};

```

注意：不要保存 `b2ContactListener` 中触点的引用，取而代之，用深复制将触点数据保存到你自己的缓冲区中。

11.7.3 接触筛选

通常，你不希望游戏中的所有物体都发生碰撞。例如，你可能会创建一个只有某些角色才能通过的门。这称之为接触筛选，因为一些交互被筛选出了。

通过实现 `b2ContactFilter` 类，Box2D 允许定制接触筛选。这个类需要一个 `ShouldCollide` 函数，用于接收两个 `b2Shape` 的指针，如果应该碰撞那么就返回 `true`。

```

bool b2ContactFilter::ShouldCollide(b2Fixture* fixtureA, b2Fixture*
fixtureB)
{
    const b2Filter& filterA = fixtureA->GetFilterData();
    const b2Filter& filterB = fixtureB->GetFilterData();

    if (filterA.groupIndex == filterB.groupIndex && filterA.
groupIndex != 0)
    {
        return filterA.groupIndex > 0;
    }

    bool collide = (filterA.maskBits & filterB.categoryBits) != 0 &&
        (filterA.categoryBits & filterB.maskBits) != 0;
}

```

```
    return collide;  
}
```

在运行的时候也可以创建接触筛选的实例，并使用 `b2World::SetContactFilter` 进行注册：

```
MyContactFilter filter;  
world->SetContactFilter(&filter);
```

12

第 12 章

纹理和动画

本章讲解纹理和动画的相关内容。纹理包括渲染纹理、纹理格式和大小，纹理缓存和一些性能优化的常用技巧。动画包括如何播放帧动画、骨骼动画和在 Cocos2d-x 中播放 Flash 动画。

12.1 渲染和修改纹理

Cocos2d-x 基于 OpenGL，在 OpenGL 中图片、文本和原始数据都是以纹理的形式存在的。

12.1.1 纹理类 Texture2D

在游戏中，当创建一个精灵对象时，加载的图片就以纹理的形式存放在内存中。Cocos2d-x 使用 Texture2D 类获取和处理纹理对象。

```
Sprite *imgMipMap = Sprite::create("Images/logo.png");
Texture2D texture = imgMipMap->getTexture();
```

Texture2D 不仅能使用图片，也能使用文本，原始数据创建 OpenGL 2D 纹理。根据创建 Texture2D 对象的方式不同，纹理的实际图片区域会比纹理的像素尺寸小，例如 "contentSize" 不等于 (pixelsWide, pixelsHigh), (maxS, maxT) 不等于 (1.0, 1.0)，所以使用时要注意这一点。

Texture2D 类提供了一些方法用来设置、获取纹理的属性。获取纹理的大小可以使用 getContentSize:

```
Texture->getContentSize();
```

使用 getName 获取纹理名称:

```
Texture->getName();
```

setDefaultAlphaPixelFormat 方法可以更改默认的纹理像素格式。Cocos2d 默认的纹理像素格式是 32 位颜色深度，如果把它们作为 16 位颜色深度的纹理来加载，内存消耗也就可以减少一半。并且这还会带来渲染效率的提升。下面是设置 16 位颜色深度的方法。

```
Texture2D::setDefaultAlphaPixelFormat(Texture2D::PixelFormat::RGB565);
```

但是需要注意，纹理像素格式的改变会影响后面加载的所有纹理。因此，如果你想后面加载纹理使用不同的像素格式，必须再调用此方法，并且重新设置一遍像素格式。另外，如果你的 Texture2D 设置的像素格式与图片本身的像素格式不匹配，就会导致显示严重失真。

Texture2D 提供了下面这些像素格式。

- ◎ Texture2D::PixelFormat::RGBA8888: 默认格式，生成 32 位纹理。
- ◎ Texture2D::PixelFormat::RGB888: 生成 24 位纹理。

- ◎ Texture2D::PixelFormat::RGBA4444: 生成 16 位纹理。
- ◎ Texture2D::PixelFormat::RGB5A1: 生成 16 位纹理。
- ◎ Texture2D::PixelFormat::RGB565: 生成 16 位纹理，无 alpha。
- ◎ Texture2D::PixelFormat::A8: 生成 8 位纹理，当仅需要一种颜色时可以使用这种格式。

在 16 位的纹理中，RGB565 可以获得最佳颜色质量，因为 16 位全部用来显示颜色，总共有 65536 个颜色值，但是图片必须是矩形的，并且没有透明像素。所以 RGB565 格式比较适合背景图片和一些矩形的用户控件。

RGB5A1 格式使用一位颜色来表示 alpha 通道，因此图片可以拥有透明区域。它只能表示 32768 种可用颜色值。使用 RGB5A1 时，图片要么只能全部是透明像素，或者全部是不透明的像素。因为只使用一位表示 alpha 通道，只有两种透明状态。但是你可以使用 fade in/out 动作来改变纹理的 opacity 属性。

如果图片包含有半透明的区域，那么 RGBA4444 格式很有用。它允许每一个像素值有 127 个 alpha 值，因此透明效率与 RGBA8888 格式的纹理差别不是很大。但是，由于颜色总量减少至 4096，所以，RGBA4444 是 16 位图片格式里面颜色质量最差的。

Cocos2d-x 2.X 版本中的纹理加载分为两个阶段，首先从图片文件中创建一个 UIImage 对象，再使用创建好的 UIImage 对象来创建 Texture2D 对象。所以，当一个纹理被加载的时候，它会消耗两倍于它本身内存占用的内存大小。当连续地加载纹理时，会一下子占用很多内存资源，所有不要连续地加载很多图片。但是 Cocos2d-x 3.0 重构了纹理渲染过程，不会再过量消耗内存。这是 3.X 系列核心性能改进之一。

另外需要注意的是，图片文件大小和纹理内存占用是两码事，图片文件大多是压缩过的，它们被使用时必须先解压缩，然后才能会 GPU 所处理，变成我们熟知的纹理。一个 2048×2048 像素的 png 图片，采用 32 位颜色深度编码，那么它在磁盘上占用空间只有 2MB。但是，如果变成纹理，它将消耗 16MB 的内存！

Texture2D 还具有很多其他功能，我们后面会讲解一些常用的方法。

12.1.2 Cocos2d-x 支持的纹理格式

不同的平台支持不同的纹理格式，表 12-1 整理了 Cocos2d-x 支持的纹理格式。

表 12-1 Cocos2d-x 支持的纹理格式

格式	是否支持 HAS ALPHA	支持的 GPU	示例设备	progress
JPEG	否	所有	所有	支持
PNG	否	所有	所有	支持
WebP	是	所有	所有	支持
PVRTC	是	PowerVR GPUs	所有 iOS 设备, Nexus S, Kindle fire	支持
ETC1	否	所有 GLES2.0 设备	大多数 Android 设备	支持
ATITC	是	Qualcomm Adreno GPUs	Nexus One	TBD
S3TC	是	Nvidia GPUs	Motorola Xoom	TBD
3DC	是	ATI & Nvidia GPUs	未知	TBD

12.1.3 Cocos2d-x 支持的最大纹理尺寸

理论上讲，Cocos2d-x 支持任意尺寸的纹理图片，但实际上支持的最大尺寸受不同平台的约束，表 12-2 列出了不同平台的模拟器中支持的最大尺寸。

表 12-2 不同平台的模拟器中支持的最大尺寸

平 台	最大尺寸（单位像素）
win32	2048×2048
Android	4096×4096
iPhone3	1024×1024
iPhone3GS	2048×2048
iPhone4	2048×2048

在真机上，也有不同的限制，例如 G3 (Hero) 支持 1024×1024, iPhone4 支持 2048×2048。所以，对于跨平台开发者，要限制纹理大小在 1024×1024 以内，这是对大多数设备的最低限制。也可以使用下面的代码获取该平台支持的最大纹理大小（模拟器中可能不起作用）。

```
GLint m_maxTextureSize = 0;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &m_maxTextureSize);
```

12.1.4 使用 RenderTexture 保存截屏

RenderTexture 是一个通用的渲染对象，用它来渲染一个对象，一般遵循下面几个步骤。

(1) 使用 **create** 方法构造一个 **RenderTexture** 对象：

```
auto renderTexture = RenderTexture::create(getWinSize().width,
getWinSize().height, Texture2D::PixelFormat::RGBA8888);
```

(2) 使用 **begin** 或者 **beginWithClear** 开启 **renderTexture**：

```
renderTexture->beginWithClear(0.0f, 0.0f, 0.0f, 0.0f);
```

(3) 使用游戏用的场景或其他对象访问 **renderTexture**：

```
getRunningScene()->visit();
```

(4) 调用 **end** 结束 **renderTexture**：

```
renderTexture->end();
```

下面就提供一个完整的方法，功能是保存游戏截图到本地硬盘。

```
void Director::saveScreenshot(const std::string& fileName, const std:::
function<void(const std::string&)>& callback){
    Image::Format format;
    if(std::string::npos != fileName.find_last_of("."))
    {
        auto extension = fileName.substr(fileName.find_last_of("."),
fileName.length());
        if (!extension.compare(".png")) {
            format = Image::Format::PNG;
        } else if(!extension.compare(".jpg")) {
            format = Image::Format::JPG;
        } else{
            CCLOG("cocos2d: the image can only be saved as JPG or PNG
```



```

format");
        return;
    }
    } else {
        CCLOG("cocos2d: the image can only be saved as JPG or PNG
format");
        return ;
    }

    auto renderTexture = RenderTexture::create(getWinSize().width,
getWinSize().height, Texture2D::PixelFormat::RGBA8888);
    renderTexture->beginWithClear(0.0f, 0.0f, 0.0f, 0.0f);
    getRunningScene()->visit();
    renderTexture->end();
    renderTexture->saveToFile(fileName , format);
    auto fullPath = FileUtils::getInstance()->getWritablePath() +
fileName;
    auto scheduleCallback = [&fullPath,callback](float dt){
        callback(fullPath);
    };
    auto _schedule = getRunningScene()->getScheduler();
    _schedule->schedule(scheduleCallback, this, 0.0f,0,0.0f, false,
"screenshot");
}

```

12.1.5 图片抗锯齿处理方式

当图片放大或者缩小时会出现锯齿，本节介绍如何尽量弱化这些锯齿，完整代码请看代码清单 12-1-5。当图片放大时，可以使用 Texture2D 的 `setAntiAliasTexParameters` 方法处理抗锯齿。原理是，当图片放大时使用相邻的四个像素进行混合运算，从而消除锯齿。但是会让图片产生模糊的感觉。

我们向场景中添加两个精灵，放大 8 倍，对其中一个的纹理调用 `setAntiAliasTexParameters` 方法。

```

auto people2 = people->create("grossinis_sister1.png");

```

```
people2->setScale(8);  
people2->setPosition(visibleSize.width*2/3,visibleSize.  
height/2);  
addChild(people2);  
people2->getTexture()->setAntiAliasTexParameters();
```

查看图 12-1，左边没有进行抗锯齿处理，右边进行了处理，没有了锯齿，但有点模糊。



图 12-1 抗锯齿效果对比

当图片缩小时，使用 minmap 效果会更好，如图 12-2 所示。

```
auto *imgMap = Sprite::create("logo-mipmap.pvr");  
if( imgMap )  
{  
    imgMap->setPosition(ccp(visibleSize.width/2.0f-100,visibleSize.  
height/2.0f));  
    addChild(imgMap);  
  
    // support mipmap filtering  
    TexParams texParams = { GL_LINEAR_MIPMAP_LINEAR, GL_LINEAR,  
GL_CLAMP_TO_EDGE, GL_CLAMP_TO_EDGE };  
    imgMap->getTexture()->setTexParameters(&texParams);  
}
```

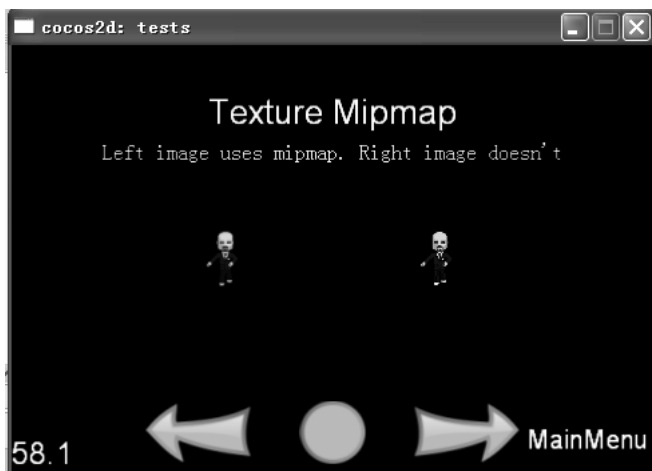


图 12-2 使用 minmap 效果

12.1.6 使用图片缓存

Cocos2d-x 中的纹理会被缓存在内存中，方便下次快速调用。当使用图片创建精灵时，这个图片也会被缓存起来。缓存的图片不能改变大小、颜色和纹理区域。加载纹理使用单例模型，纹理一旦被加载完，后面使用时会调用纹理的引用，这样可以减少使用 GPU 和 CPU 的使用量。

Cocos2d-x 使用 TextureCache 把纹理缓存到内存中：

```
auto texture0 = Director::getInstance()->getTextureCache()->addImage(
    "Images/grossini_dance_atlas.png");
auto img0 = Sprite::createWithTexture(texture0);
addChild(img0);
```

或者使用 SpriteFrameCache 批量添加精灵帧，然后就可以使用缓存的纹理创建精灵对象了：

```
SpriteFrameCache* cache = SpriteFrameCache::getInstance();
cache->addSpriteFramesWithFile("animations/grossini.plist",
    "animations/ grossini.png");
m_pSprite1 = Sprite::createWithSpriteFrameName("grossini_dance_01.
    png");
```

```
m_pSprite1->setPosition( Point( s.width/2-80, s.height/2) );
```

有三种方式加载纹理到 `SpriteFrameCache`:

- ◎ 一个.plist 文件（特殊的 XML）。
- ◎ 一个.plist 文件和一张图片，如上面使用的。
- ◎ 一个 `SpriteFrame`。

在使用之前最好预加载所有图片到缓存区。具体怎么加载有开发者具体实现，比较常用的一种方式异步加载，即在开始游戏前提供一个加载场景，在场景中添加一个进度条，用来显示图片加载进度。这样做最大的好处在于，游戏体验会表现得非常平滑，而且不需要再担心资源的加载和卸载问题了。

12.1.7 制作游戏加载场景

本节讲解如何制作游戏加载场景，加载场景要完成两个基本功能，一个是在后台异步加载图片到缓存，另一个是把加载进度显示在场景中，完整代码请查看代码清单 12-1-7。在该例子中加载 20 个图片，当加载完成后显示“all loaded”，如图 12-3 所示。

（1）在场景中添加一个进度条：

```
auto loadBg = Sprite::create("sliderTrack.png");//进程条的底图
loadBg->setPosition(Point(visibleSize.width/2,visibleSize.
height/2));
this->addChild(loadBg,1);

loadProgress = ProgressTimer::create(Sprite::create
("sliderProgress.png"));//创建一个进程条
loadProgress->setBarChangeRate(Point(1,0));//设置进程条的变化速率
loadProgress->setType(ProgressTimer::Type::BAR);//设置进程条的类型
loadProgress->setMidpoint(Point(0,1));//设置进度的运动方向
loadProgress->setPosition(Point(visibleSize.width/2,visibleSize.
height/2));
loadProgress->setPercentage(0.0f);//设置初始值为 0
this->addChild(loadProgress,2);
```

(2) 加载 20 个图片，当图片加载完成后调用回调函数：

```
Director::getInstance()->getTextureCache()->addImageAsync
("HelloWorld.png", CC_CALLBACK_1 (HelloWorld::loadingCallBack, this));
Director::getInstance()->getTextureCache()->addImageAsync
("grossini.png", CC_CALLBACK_1 (HelloWorld::loadingCallBack, this));
Director::getInstance()->getTextureCache()->addImageAsync
("grossini_dance_01.png", CC_CALLBACK_1 (HelloWorld::loadingCallBack,
this));
Director::getInstance()->getTextureCache()->addImageAsync
("grossini_dance_02.png", CC_CALLBACK_1 (HelloWorld::loadingCallBack,
this));
...
```

(3) 调用回调函数 `loadingCallBack`，先计算加载进度：

```
float newPercent = 100 - ((float)_numberOfSprites - (float)_numberOf
Loaded Sprites)/((float)_numberOfSprites/100); //计算进度条当前的百分比
```

(4) 更新进度条：

```
loadProgress->setPercentage(newPercent); //更新进度条
```

(5) 判断是否加载完，如果加载完更改加载状态：

```
//图片加载完成后
if(_numberOfLoadedSprites == _numberOfSprites)
{
    loadLabel->setString("all loaded");
}
```



图 12-3 加载完成

12.1.8 使用 TexturePacker 制作 Sprite Sheet

前面的章节已经提到优化游戏的两个着手点，一是减少资源所占内存，二是减少纹理加载次数。所以要把多个图片合并到一个大图里，这张大图就叫作 **sprite sheet**。图片格式采用 **PVR**，PVR 在运行速度和内存消耗方面都要比 **PNG** 格式要快和小。一般情况下 PVR 消耗的内存比 PNG 消耗的内存小 25% 左右。PVR 文件有三种格式，优先选择 **pvr.ccz** 格式。它是专门为 Cocos2d 和 TP 设计的。在 TP 里面，这是它生成的最小的 PVR 文件，而且 **pvr.ccz** 格式比其他任何文件格式的加载速度都要快。

本节讲解如何使用 TexturePacker 制作 **pvr.ccz** 格式文件。

1. 下载、安装 TexturePacker

下载地址为 <http://www.codeandweb.com/texturepacker>，安装 TexturePacker 就像安装其他软件一样简单方便。安装好后运行，第一次运行时会提示输入秘钥，我们选择免费版，如图 12-4 所示。



图 12-4 TexturePacker 运行界面

在该窗口的左边部分有一些配置选项，使用这些选项可以配置 Sprite Sheet 的大小、布局、输出格式等。下面介绍常用的几个选项。

- ◎ **Data Format:** 导出的 spritesheet 在那种情况下使用。
- ◎ **Data file:** 导出的.plist 的文件位置。
- ◎ **Texture format:** 导出的大图片的格式。
- ◎ **Texture file:** 导出的图片的位置。
- ◎ **Image format:** 纹理的像素格式，具体每个格式代表的意思可以查看 12.1.1 节内容。
- ◎ **Max size:** 设置纹理图片的最大尺寸。
- ◎ **Fixed size:** 指定纹理大小，不会根据图片多少进行尺寸调整。
- ◎ **Scale:** 设置 Sprite Sheet 比原始尺寸大或者小。

你不用担心怎么设置才比较好，很多设置采用默认值就完全没问题了，只需要指定下生成文件的保存位置。

2. 添加资源

单击工具栏上的 Add Folder，选择图片所在目录，TexturePacker 就会自动布局加入的图片。用到的图片在右边显示，当单击中间纹理集中的一个图片时，该图片周围会出现虚框，表示当前被选择的图片；把鼠标放到图片上面，也能显示当前图片的信息，比如大小等。

并且，当把新的图片放到指定的文件夹下或者移除图片时，TexturePacker 会自动从新布局，生成新的 Sprite Sheet。

3. 发布 Sprite Sheet

在 Texture format 选项中选择 Pvr.ccz 格式；单击 Texture file 后面的文件夹图形，在弹出的对话框中选择保存路径，名称命名为 res.pvr.ccz。然后，TexturePack 会自动为我们在 Data file 那里生成相应的 plist 文件路径。通常它们在同一目录下。

注意保存文件名使用.pvr.ccz，而不是.png，原因上文已经说明。然后单击工具栏中的 publish，这时就成功发布了 Sprite Sheet。

4. 在 Cocos2d-x 中使用 Sprite Sheet。

Cocos2d-x 使用 SpriteFrameCache 把 Sprite Sheet 添加到缓存中：

```
SpriteFrameCache::getInstance()->addSpriteFramesWithFile
("res.pvr.ccz.plist", "res.pvr.ccz");
```

我们用添加的纹理创建一个精灵：

```
auto nonencryptedSprite = Sprite::createWithSpriteFrameName("Icon.png");
```

“icon.png”是合并前单个图片的名称。

12.2 动画

12.2.1 帧动画

动画就是一个接一个的静态动作按一定顺序连续播放的结果，Cocos2d-x 中的帧动画也具有相同原理。把一个精灵的动作拆分成多个静态图片，然后再使用该精灵连续播放这些图片。这样，精灵就具有了跑动、打斗等效果。

Cocos2d-x 中播放帧动画需要几个类协作完成，这些常用类包括下面几个。

- ◎ **SpriteFrame**：精灵帧信息，序列帧动画依靠多个精灵帧信息来显示相应的纹理图像，一个精灵帧信息包含了所使用的纹理，对应纹理块的位置以及纹理块是否经过旋转和偏移，这些信息可以取得对应纹理中正确的纹理块区域作为精灵帧显示的图像。下面这句代码常用来创建精灵帧信息。

```
Texture2D *heroTexture=TextureCache::getInstance()->addImage ("hero.
png");
SpriteFrame *frame0=SpriteFrame::createWithTexture(heroTexture, Rect
(32*0, 32*1, 32, 32));
```


- ◎ **SpriteFrameCache:** SpriteFrameCache 把多个 SpriteFrame 缓存在内存中，它通过字典的方式存储单个 SpriteFrame。key 为帧的名字，值为 SpriteFrame。通过帧名获取 SpriteFrame，当没有时，返回 NULL。SpriteFrameCache 一般用来处理 plist 文件，该文件对应一张包含多个精灵的大图，并指定了每个独立的精灵在这张“大图”里面的位置和大小。下面是该类的常用使用方法。

下面是使用 SpriteFrameCache 的代码示例：

```
SpriteFrameCache* cache = SpriteFrameCache::getInstance();
cache->addSpriteFramesWithFile("animations/grossini.plist",
"animations/ grossini.png");
m_pSprite1 = Sprite::createWithSpriteFrameName("grossini_ dance_01.
png");
m_pSprite1->setPosition( Point( s.width/2-80, s.height/2) );
```

如果 plist 文件跟对应的 png 图片在同一目录下，且名字相同，上面第一句可简写成：

```
cache->addSpriteFramesWithFile("animations/grossini.plist");
```

- ◎ **AnimationFrame:** 帧动画单帧信息，它存储了对应的精灵帧信息。可通过 Animation 的 getFrames 函数获得。
- ◎ **Animation:** 帧动画信息，它存储了所有的单帧信息，可以对单帧信息进行管理。一般通过 create 或者 createWithSpriteFrames 方法创建。
- ◎ **Animate:** 帧动画处理类，它是真正完成动画表演的类。

创建帧动画的流程是先创建动画帧信息（SpriteFrame），然后由动画帧信息生成动画信息（Animation），最后由动画信息创建出序列帧动画（Animate）供精灵演示。

接下来，我们使用一个英雄打斗案例来演示精灵和帧动画的使用方法。

12.2.2 使用帧动画实现英雄打斗

该案例在 3.4.2 节的基础上添加相应功能，3.4.2 节的案例加载了一个战斗场景，

本节在该战斗场景中实现两个英雄相互打斗的效果。

当场景加载完成后，添加英雄 A 和英雄 B，为诉说方便起见，左边的英雄为英雄 A，右边的英雄为英雄 B。两个英雄相对站立在屏幕两边，然后英雄 A 攻击英雄 B，与此同时，英雄 B 进行格挡。英雄 A 攻击结束后，英雄 B 开始攻击英雄 A，英雄 A 进行格挡。以此过程，不停打斗下去。这就是本节要实现的打斗效果。

我们需要建一个 Hero 英雄类，简单分析一下，Hero 类应该包括这些功能：Hero 类能生成一个英雄对象，把英雄展示到场景中，能获取到英雄站立时的动作，能获取到英雄格挡的动作，能获取到英雄攻击的动作。由于在攻击时被攻击都有一个格挡的动作，所以要把攻击动作分成攻击前动作和攻击收回动作。由于每个英雄不确定哪一帧是最后的攻击动作，所以还要记录下每个英雄对象攻击动作的帧数。

步骤和核心代码如下所示。完整代码请查看代码清单 12-2-2。

(1) 新建一个项目 13-2-2，按照 3.4.2 所示方法，添加打斗场景。

(2) 把两个英雄的 png 图片添加到资源目录里，每个图片包含一个英雄的呼吸动作、格挡动作、攻击动作。我们把英雄的每个连续动作拆分成多个独立静态的图片，每一个动作都是帧动画的一帧。

(3) 在项目中添加 Hero 类头文件 Hero.h 和源文件 Hero.cpp，Hero 以公有的方式继承自 Sprite。

```
class Hero :public cocos2d::Sprite {
}
```

(4) 在 Hero.h 中添加整型成员变量 fightNum，记录最后攻击帧所在位置。

```
int fightNum = 0; 记录最后攻击帧是第几帧
```

(5) 添加构造函数，该构造函数包含一个整型参数，用来保存最后攻击帧所在位置。

在头文件中添加构造函数和生成函数的声明，如下所示：

```
//Hero 构造函数，fightNum 为最后攻击帧所在位置
Hero(int fightNum);
```

在源文件中分别实现两个函数：

```
//Hero 构造函数，保存最后攻击帧所在位置
Hero::Hero(int fightNum){
    this->fightNum = fightNum;
}
```

(6) 添加一个静态方法 **create**，便于生成英雄对象。

在头文件中添加 **create** 方法的声明：

```
static Hero* create(const char * heroPic,int fightNum);
```

在源文件中添加 **create** 方法的实现：

```
Hero* Hero::create(const char * heroPic,int fightNum){
    //调用构造函数
    Hero *hero = new Hero(fightNum);
    //调用父级的 initWithFile 方法初始化 hero 对象，如果初始化成功，返回 hero
    对象，如果初始化失败，返回 NULL
    if (hero&&hero->initWithFile(heroPic, Rect(0, 0, 237.5, 191))) {
        hero->autorelease();
        return hero;
    }else{
        delete hero;
        hero = NULL;
        return NULL;
    }
}
```

到此，我们就可以添加一个英雄到场景中了。打开 **FightScene.cpp**，在 **init** 方法里添加如下代码，保存运行，就能在模拟器里看到场景中多了一个静态的英雄，效果图如图 12-5 所示。

```
Size size = Director::getInstance()->getWinSize();//获取屏幕尺寸
Hero *pHero = Hero::create("niutouzhanshi.png",8);//创建英雄对象
addChild(pHero);//把英雄对象添加到场景中
pHero->setPosition(Point(120,150));
pHero->setTag(1);
```



图 12-5 在战斗场景中添加静态的英雄

(7) 已经能把英雄添加到场景中了，但英雄站立时是静态的，如果站立时能有个呼吸的动作，效果就更好。看下添加的图片 `niutouzhanshi.png`，前 4 幅图组成一个完整的呼吸动作，所以可以用这 4 幅图做一个帧动画来实现该功能。

我们为 `Hero` 类添加一个方法 `getStandAni`，用来获取英雄对象站立时的帧动画。首先定义 4 个 `SpriteFrame`，保存每个精灵帧的信息。这里使用到 `getTexture` 获取 `Hero` 对象的纹理图。

```
//声明 4 幅帧动画
SpriteFrame *frame0,*frame1,*frame2,*frame3;
//用纹理创建帧动画信息，第二个参数表示显示区域的 x,y,width,height
frame0=SpriteFrame::createWithTexture(this->getTexture(),
Rect(237.5*0,0,237.5, 191));
frame1=SpriteFrame::createWithTexture(this->getTexture(),
Rect(237.5*1,0,237.5, 191));
frame2=SpriteFrame::createWithTexture(this->getTexture(),
Rect(237.5*2,0,237.5, 191));
frame3=SpriteFrame::createWithTexture(this->getTexture(),
Rect(237.5*3,0,237.5, 191));
```

然后创建一个数组，把四个精灵帧信息保存在数组中：

```
Array *animFrames=Array::create();
```

```
animFrames->addObject(frame0);
animFrames->addObject(frame1);
animFrames->addObject(frame2);
animFrames->addObject(frame3);
```

再使用该精灵帧信息数组创建一个帧动画：

```
Animation *animation=Animation::createWithSpriteFrames(animFrames);
```

之后需要设置每帧播放的时间间隔，及生成帧动画处理类。**注意：一定要设置每帧播放的时间间隔，否则会报错。**

```
//根据动画模板创建动画
animation->setDelayPerUnit(0.2f);
animFrames->release();
Animate *animate=Animate::create(animation);
return animate;
```

我们在 FightScene.h 中，调用 getStandAni 方法，获取呼吸帧动画，然后让英雄播放该动画，就能实现英雄站立时有呼吸的动作了：

```
pHero->runAction(RepeatForever::create( pHero->getStandAni()));
```

(8) 按同样的方法，在场景中添加另外一个英雄。位置与第一个英雄相对，即把 X 坐标设置为 size.width-120。面向方向相反，使用 setFlipX 设置水平旋转。

```
mHero->setPosition(Point(size.width-120,150));
mHero->setFlipX(180);
mHero->setTag(2);
```

(9) 上面在场景中添加的两个英雄只能“深情相望”，但我们想要的是“互动干戈”，接下来就介绍下如何实现英雄 A 攻击英雄 B。

再次查看英雄 A 对应的图片资源 niutouzhanshi.png，图片里第二排能组成一个攻击动作，所以就用这 8 幅图片做一个帧动画来。第 5 幅图片是攻击动作的最后一幅，下面 3 幅是攻击收回动作。由于在攻击动作的最后一幅，英雄 B 要格挡一下，所以要把攻击帧动画分成攻击动画和攻击收回动画，这两个动画分别通过 getFightAni 和 getFightBackAni 获取。在构造函数里，我们已经传递进来一个整数，并存储为攻击最后一帧所在位置。两个函数的实现可以参考呼吸动画的实现方式，

但由于不知道攻击动作由几帧组成，所以要改成循环的方式实现。

```
Array *animFrames=Array::create();
//使用循环方式生成动画帧信息数组
for (int i=0; i<fightNum; i++) {
    animFrames->addObject(SpriteFrame::createWithTexture(this->getTexture(), RectMake(237.5*i,191,237.5, 191)));
}
//根据动画帧信息数组 Animation 对象
Animation *animation=Animation::createWithSpriteFrames(animFrames);
//根据动画模板创建动画
animation->setDelayPerUnit(0.1f);
Animate *animate=Animate::create(animation);
return animate;
```

攻击收回动作帧动画的产生与攻击动画基本一致，只是循环参数改成从 `fightNum` 到数值 8。现在已经有了攻击动画，就可以实现英雄 A 攻击英雄 B 的动作了。但需要把英雄 A 拆开的攻击动作连贯起来，`Sequence` 是生成按顺序执行动作的方法（详情请查看第 5 章），可以用来连贯两个动作。先定义三个变量，分别保存站立动作、攻击动作、攻击收回动作：

```
Animate *fstandAni = pHero ->getStandAni();
Animate *ffightAni = pHero->getFightAni();
Animate *ffightBackAni = pHero ->getFightBackAni();
```

由于，牛头人战士是近战英雄，需要跑到对面英雄前进行攻击，所以先要执行一个水平位移的功能，攻击完成后再返回到原来的位置，水平位移使用 `MoveBy` 生成：

```
MoveBy *move = MoveBy::create(0,p(size.width-280,0));
```

使用 `Sequence` 把这几个动作按顺序关联起来：

```
Sequence *firstSe = Sequence::create(fstandAni,move,ffightAni,
ffightBackAni, move->reverse(), NULL);
```

然后让英雄 A 播放该动作：

```
pHero->runAction(RepeatForever::create(firstSe));
```

运行项目，在模拟器中就能看到英雄 A 不停地攻击英雄 B。

(10) 在英雄 A 攻击英雄 B 的同时英雄 B 要有个格挡的动作，这个动作也在英雄 B 对应的图片中。首先给 Hero 类添加一个获取格挡动作的方法 `getProtectAni`，该方法返回一个格挡帧动画。实现方法同 `getStandAni`，但资源图片里的格挡动作只有一帧，所以只需生成一个精灵帧信息即可。由于帧的个数太少，需要把每帧播放时间设置长一点，否则会闪一下，很快消息。

```
SpriteFrame *frame0;
frame0=SpriteFrame::createWithTexture(this->getTexture(),
Rect(237.5*4, 191,237.5, 191));
Array *animFrames=Array::create();
animFrames->addObject(frame0);
Animation *animation=Animation::createWithSpriteFrames(animFrames);
//根据动画模板创建动画
animation->setDelayPerUnit(0.3f);
Animate *animate=Animate::create(animation)
```

(11) 现在通过 `getProtectAni` 可以获取英雄的格挡动作了，那么如何实现在英雄 A 攻击的同时，英雄 B 进行格挡呢？一个方法是按时间设置，我们可以计算出站立到攻击的时间，然后在该时间点让英雄 B 执行格挡动作。如果有很多英雄角色且每个英雄的攻击动作时间都不一样时，计算维护这么多时间就是让人头大的事。所以，我们可以使用回调的方式来实现攻击和格挡的动作同时进行。回调使用 `CallFunc` 来生成，更多关于回调函数参考第 5 章。

首先在 `FightScene` 中添加格挡动作回调函数 `protect`：

```
void FightScene::protect() {
    secondHero->runAction(secondHero->getProtectAni());
}
```

然后修改上面生成的序列动作 `firstSe`，把回调函数 `protect` 添加进去：

```
Sequence *firstSe = Sequence::create(fstandAni,move,ffightAni,
CallFunc::create(this, callfunc_selector(FightScene::protect)),
ffightBackAni, move->reverse(), NULL);
```

然后让英雄 A 播放该动作：

```
pHero->runAction(RepeatForever::create(firstSe));
```

运行项目，就能看到英雄 A 不停地攻击英雄 B，同时英雄 B 会格挡英雄 A 的攻击，效果如图 12-6 所示。



图 12-6 英雄 A 打击英雄 B

12.2.3 Sprite Sheet 动画

12.2.2 节展示的帧动画是从一系列图片创建的动画，虽然这种方式简单易懂，但是在实际的游戏开发中很少用到这种方式。Sprite Sheet 动画是最常用的创建 2D 动画的方式。

使用 Sprite Sheet 动画可以减少文件读取次数和纹理渲染次数。本节就演示如何创建 Sprite Sheet 动画。

有多种创建方式，常用的一种方式是通过 .png 和 .plist 创建。该方式使用 SpriteFrameCache 把 Sprite Sheet 添加到缓存中：

```
SpriteFrameCache * cache = SpriteFrameCache::getInstance();  
cache->addSpriteFramesWithFile("jiuyinbaiguzhua.plist",  
"jiuyinbaiguzhua.png");
```

当添加加载完成后，就可以使用这些 sprite frame 创建一个精灵对象：


```

    auto m_pSprite1 = Sprite::createWithSpriteFrameName ("jiuyinbaigu
zhua_1.png");
    m_pSprite1->setPosition(visibleSize.width/2.0f,
visibleSize.height/ 2.0f);
    addChild(m_pSprite1);

```

然后创建一个 **Vector** 对象，对象的类型为 **SpriteFrame***，通过一个循环把添加到场景中的 **SpriteFrame** 添加到该 **Vector** 对象中：

```

Vector<SpriteFrame*> animFrames(18);

char str[100] = {0};

for(int i = 1; i <= 18; i++)
{
    sprintf(str, "jiuyinbaiguzhua_%d.png", i);
    SpriteFrame* frame = cache->getSpriteFrameByName( str );
    animFrames.pushBack(frame);
}

```

然后根据 **animFrames** 创建 **animation**：

```

Animation* animation = Animation::createWithSpriteFrames(animFrames,
0.3f);
m_pSprite1->runAction(RepeatForever::create(Animate::create
(animation) ) );

```

运行程序，看到创建中不停播放添加的 **Sprite Sheet** 动画，如图 12-7 所示。

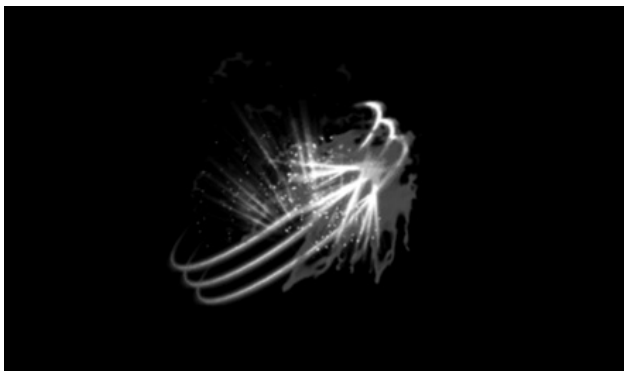


图 12-7 Sprite Sheet 动画效果

12.2.4 骨骼动画

创建 Sprite Sheet 动画快捷又简单，通常会选择这种方式。但是当游戏中需要很多动画时，就会占用很多内存。这时就不得不减少动画；如果不想减少，只能在很低 FPS 下播放动画，这时，动画就会显得不顺畅。骨骼动画在这种情况下应运而生。

在 Cocos2d 中，骨骼动画分为两个部分，一部分用来呈现外观，也被称作皮肤；另一部分用来控制动作，被称作骨架，图 12-8 有助于帮助我们理解骨骼动画。

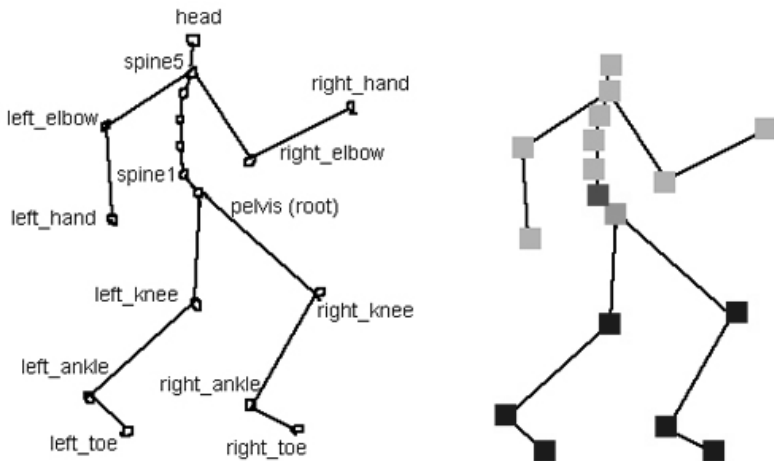


图 12-8 骨骼动画

骨骼动画由相互关联的骨骼组成，每个骨骼状态的变动都会影响其他的骨骼。通过骨头的不同变化组合，就可以得到不同的造型。

骨骼动画相对 Sprite Sheet 动画具有这些优点：

- ◎ 动画更精准，更真实，并可通程序动态控制。
- ◎ 动画各部分采用拼接方式，占用位图 / 内存资源少。
- ◎ 骨骼显示对象与骨骼的逻辑分离，可在不影响动画播放的情况下动态更换。

制作骨骼动画时，前期设置比较麻烦，但设置好后就能方便调用了。

从实现的角度，骨骼动画是 `KeyFrameAnimation` 的集合，而 `KeyFrameAnimation` 又包含多个关键帧，通过帧的编号和一个 `Transformation` 能定义一个关键帧，`Transformation` 是一个类，包含了 2D 变换，比如平移、选择、缩放等。

12.2.5 使用 CocoStudio 制作骨骼动画

制作骨骼动画的工具有多种，`CocosBuilder` 只能在 OS X 系统中运行，所以本书就介绍如何在 Windows 平台下使用 `CocoStudio` 制作骨骼动画。

`CocoStudio` 是触控提供的官方编辑器，能用来制作动画、UI、场景和数据。本节使用的版本是 1.4.0.1，针对 `Cocos2d-x 3.0`。

打开 `CocoStudio`，选择 `Animation Editor`。然后按 `Ctrl+N` 组合键创建一个新项目。在弹出的对话框中填入项目名称和保存路径，如图 12-9 所示。

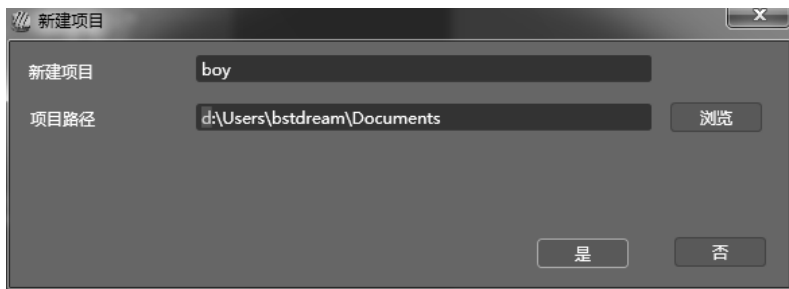


图 12-9 创建一个新项目

制作动画需要素材，找到窗口最右侧资源选项卡，单击文件夹图标，选择准备好的资源文件，这样就把素材添加到编辑器里了，如图 12-10 所示。

然后把英雄的图片拖入渲染区，窗口中间的对象结果标签里列出了被渲染的图片，双击名称能进入编辑状态，修改资源的名称；也能上下拖动来改变图片的层级，越靠上层级越高，在显示时会把层级低的对象遮盖住。

选择“属性”选项卡（图 12-11），坐标项确定图片位置，旋转项对图片进行旋转，缩放项对图片进行缩放，倾斜用来设置图片倾斜。

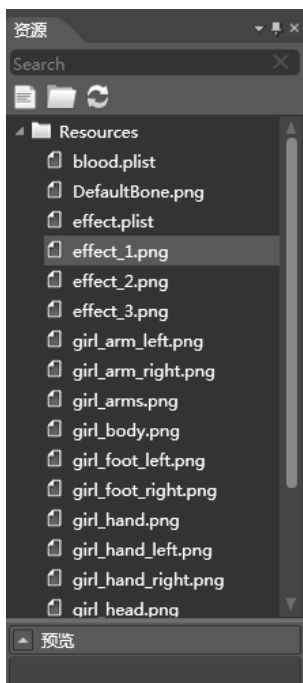


图 12-10 添加资源



图 12-11 设置属性

利用这些功能制作出英雄的体态，如图 12-12 所示。

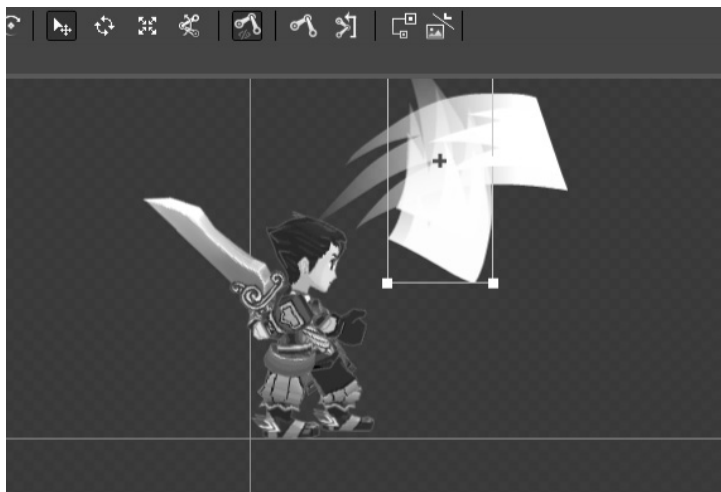


图 12-12 英雄的体态

接下来，为体态添加骨骼。工具栏最后几个图标有显示（隐藏）骨骼、创建（停止创建）骨骼等。下面为头部添加一个骨骼。

- （1）在渲染区选择头部，再单击 （快捷键 Alt+K）开始创建骨骼。
- （2）用鼠标单击头像的旋转点，该点在脖子附近，然后向头顶拖动鼠标。
- （3）单击停止创建骨骼按钮 （快捷键 Alt+K）停止创建骨骼。
- （4）选中头部，鼠标右击，在弹出的选项中选择“绑定到骨骼”（快捷键 Alt+I）。

（5）这样就完成了一个部分的骨骼，按同样的方法给其他身体部位也绑定骨骼动画。但是骨骼动画中某个部位的变动会影响其他部位，比如身体动会带动腿，腿又带动脚，我们可以为这些联动部位绑定父子关系以完成联动。当父节点变动时，它的子节点会跟着变动。

（6）选中腿部骨骼，在骨骼上点击右键，选中绑定父关系（快捷键 Alt+P），然后再选中下身体的骨骼，就完成了骨骼的父子绑定，现在移动下身体，就能看到，身体带动了腿脚一起运动。同理把为英雄其他的骨骼绑定父子关系，最后如图 12-13 所示。

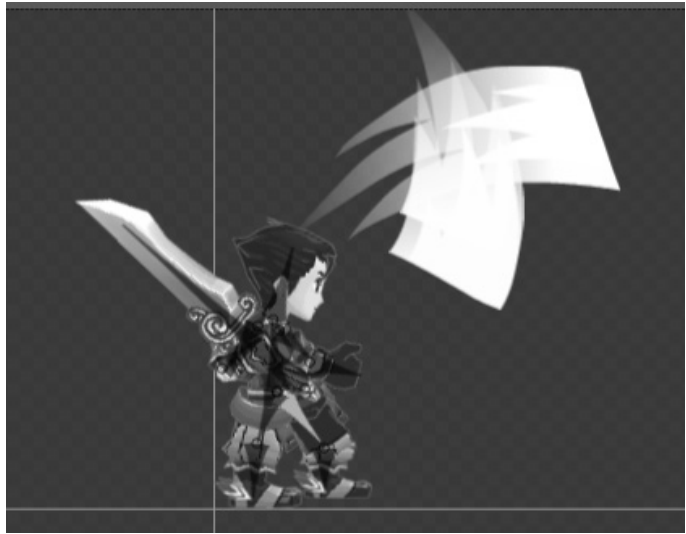


图 12-13 添加骨骼效果

接下来就开始制作动画，制作过程类似制作 Flash 的帧动画。选择工具栏最左边的 ICON  形体模式，切换到动画模式，窗口下边就会出现动画编辑区域，如图 12-14 所示。



图 12-14 动画编辑区域

下面我们就做一个头部摇晃的动作。

(1) 右键动作列表区域，在弹出的菜单中选择添加动画，然后双击刚添加的动

画，重命名为 shakeHead。

(2) 选择头部骨骼，动画帧区域相应的骨骼动画也会被选中。然后拖动时间轴到 5，设置旋转为 20；再拖到 10，设置旋转为 0；再拖到 15，设置选置旋转为-20。点击动画帧区域的左上角，渲染区就能预览到制作的动画。

(3) 同样的方式为英雄添加其他几个动作，如站立动画、跑步动画、攻击动画等。

(4) 接下来添加帧事件，帧事件的意思就是当动画播放到该帧时触发一个事件。在程序中为该事件绑定事件监听器，当播放到该帧时，就会触发该事件监听器。在动画帧区域选择骨骼 Layer17，再选择第三个关键帧，在属性窗口找到“帧事件”，输入事件 fight，如图 12-15 所示。



图 12-15 添加帧事件

制作好动画就要导出，按快捷键 Ctrl+E 打开导出对话框，在对话框选择默认配置，点击导出。

12.2.6 在项目中调用 CocoStudio 制作的骨骼动画

本节演示如何在程序中调用 12.2.5 导出的骨骼动画。

创建一个项目，把刚才导出的文件复制到项目的 Resource 目录下。由于创建的

项目没有把 CocoStudio 引入到项目中，所以需要先引入 CocoStudio。

(1) 右击解决方案→添加→现有项目，进入目录 cocos2d\cocos\editor-support\cocostudio\proj.win32 中，选择 libCocosStudio，确定，添加 libCocosStudio 库到项目中。使用同样的方式添加 libExtensions 和 libGUI 到解决方案中。

(2) 右击项目→引用→添加新引用→勾选 libCocosStudio、libExtensions 和 libGUI。

(3) 右击项目→属性→C/C++→常规→附加包含目录→编辑，添加\$(EngineRoot)cocos\editor-support，\$(EngineRoot)extensions；

经过上述设置，使用时在文件顶部引入头文件和命名空间：

```
#include "cocostudio/CocoStudio.h"
using namespace cocostudio;
```

然调用下面的代码就可以了：

```
ArmatureDataManager::getInstance()->addArmatureFileInfo( "Hero0.png", "Hero0.plist" , "Hero.ExportJson");
    Armature *armature = Armature ::create("Hero" );
    armature->getAnimation()->play( "run" );
    addChild(armature);
```


13

第 13 章

使用 Cocos2d-x 制作 2048 休闲游戏

本章带领读者完成 2048 游戏的制作，2048 是最近比较流行的一款益智类休闲游戏，也是经典的数字合成游戏。

13.1 准备工作

在开发之前要准备好制作环境的搭建，这些都在前面的章节中有详细讲解，这里只是总结一下。

- (1) 操作系统使用 Windows 7。
- (2) 框架使用 Cocos2d-x 3.0 正式版。
- (3) 开发工具使用 Visual Studio 2012 专业版。
- (4) UI 编辑器使用 CocoStudio V1.4.0.1 + Cocos2D-X 3.0。
- (5) 切图工具使用 PhotoShop。

先用 PhotoShop 切出准备需要用到的 UI 资源。切图过程就不用详细介绍，不是本书重点。

13.2 使用 CocoStudio 制作 UI 界面

(1) 打开 CocoStudio，在启动界面我们选择 UI Editor，按快捷键 Ctrl+N 新建一个项目，命名为 **game**，保存在 E:\UI 中，如图 13-1 所示。

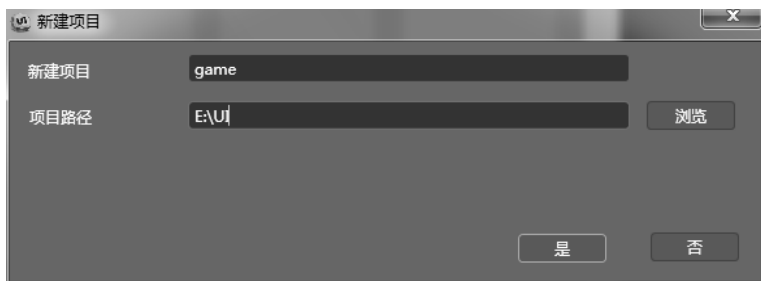


图 13-1 新建项目

(2) 创建项目成功后，选择资源面板中的添加文件功能，把准备好的图片资源添加到项目中，如图 13-2 所示。

(3) 选择工具栏里的画布，在下拉选择中选择自定义，然后在弹出窗口中修改宽为 640，高为 960，确定。这样调整好了画布大小。

(4) 把资源面板里的 **bg.png** 拖到属性面板中特性一栏中的文件项，如图 13-3 所示。

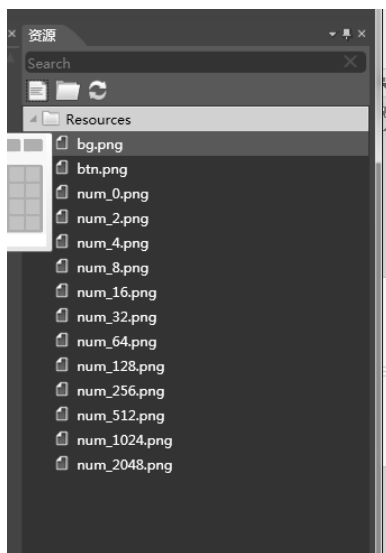


图 13-2 添加资源

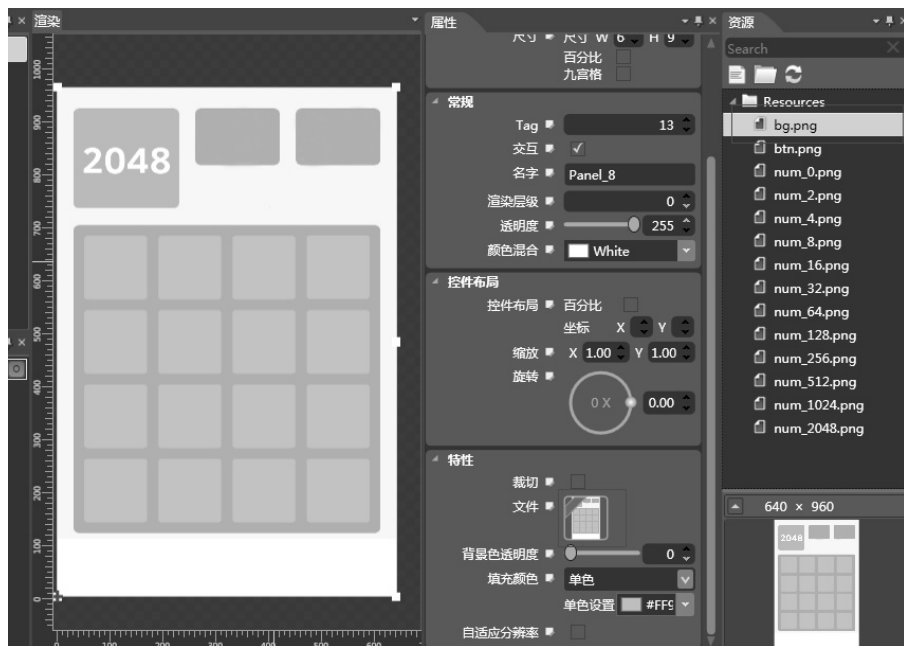


图 13-3 拖动背景图片

(5) 从最左边的控件栏中选择文本框，拖到渲染面板中和合适位置，在属性面板的特性区调整字号为 40，文本为 Score。同理再添加一个文本框 High Score。这两个文本框显示的文字是静态的，不会发生变化。

(6) 游戏中需要显示当前分数和最高分数，会根据游戏的进行发生变化，所以按上述步骤再添加两个文本框，分别在属性面板中常规区的名字修改为 score、highscore，如图 13-4 所示。



图 13-4 添加文本框

(7) 再添加两个按钮，代表从新开始和退出功能。在左边控件列表中选择按钮 ，并拖到渲染区，调整位置。把资源面板中的 btn.png 拖到属性面板，特性功能区的资源项的三个框中，第一个表示正常状态，第二个表示按下时的状态，第三个表示不可单击的状态，接着把文本改成 Restart，大小改成 40，名称改成 restart。同理添加退出按钮，如图 13-5 所示。

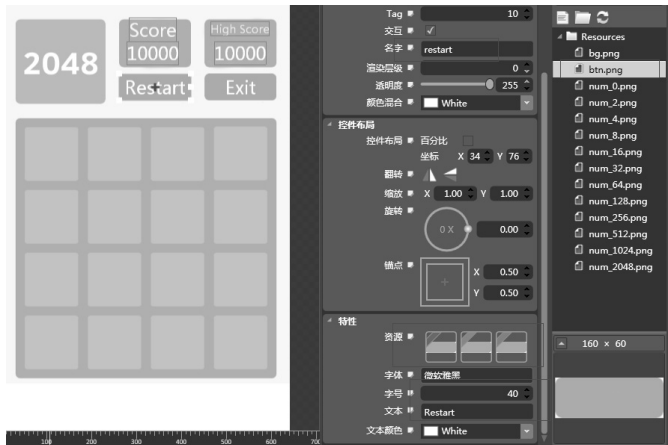


图 13-5 添加按钮

(8) 这时 UI 制作已经基本完成。最后还要添加一个容器，用来添加数字方块，位置和大小调整跟中间灰色大方框一样，名字改成 `mainpanel`。到此 UI 已经制作完成。

(9) 按快捷键 `Ctrl+E`，在弹出的对话框中选择“导出全部大图”单选项，如图 13-6 所示，单击“确定”按钮。

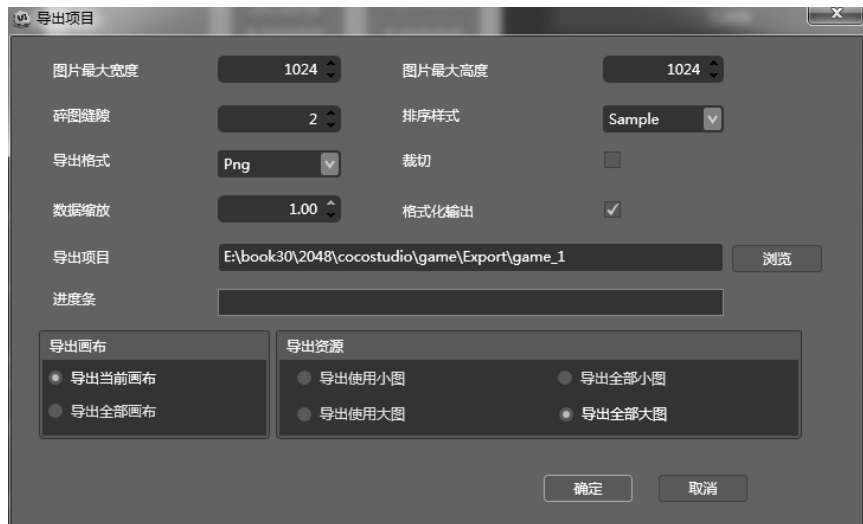


图 13-6 导出图片

13.3 编写逻辑代码

上节把 UI 界面制作成功，本节开始编写逻辑代码。

13.3.1 把 UI 界面添加到游戏界面中

(1) 新建游戏项目，把 13.2 节导出的内容复制到项目的 `Resource` 目录下。在项目添加一个类 `GameLayer`，这个类用来完成该游戏几乎所有功能。添加完成后，在 `AppDelegate.cpp` 中引入 `GameLayer` 头文件。

```
#include "GameLayer.h"
```

(2) 游戏开始后加载该场景:

```
auto scene = GameLayer::createScene();  
    glview->setDesignResolutionSize(640,960,ResolutionPolicy::  
SHOW_ALL);  
    // run  
    director->runWithScene(scene);
```

(3) 打开 GameLayer.cpp, 在 init 方法中加入 UI 界面:

```
//加载游戏界面  
auto UI = cocostudio::GUIReader::getInstance()-> widgetFromFile  
("game_1.ExportJson");  
this->addChild(UI);
```

(4) 按 F5 调试程序, 运行成后, 就能看到游戏的整体界面, 如图 13-7 所示。

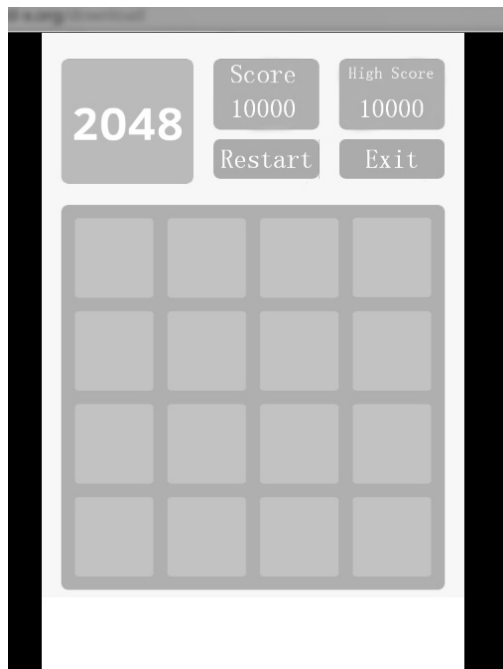


图 13-7 游戏整体界面

13.3.2 添加获取分数控件并设置分数

上面提到当前分和最高分根据游戏的进行而变化，所以要获取 UI 中的两个控件。IScore 为当前分数，highScore 为最高分，由于当前分数要在其他方法中用到，所以声明为成员变量：

```
IScore = (Text*)Helper::seekWidgetByName(UI, "score");
Text * highscore = (Text*)Helper::seekWidgetByName(UI,
"highscore");
```

游戏初始时当前分为 0，所以设置 IScore 的文本为 0：

```
IScore->setText(String::createWithFormat("%d",0)->getCString());
```

最高分使用 UserDefaults 保存：

```
int high=UserDefaults::getInstance()->getIntegerForKey ("HighScore",
0);
highscore->setText(String::createWithFormat("%d",high)->
getCString());
```

13.3.3 添加数字方块类

我们把数字方块做成一个类，数字方块有两个显示属性要变，背景图和显示数字。为了控制显示什么样的背景和数字，我们为每个方块定义了一个等级属性和等级对应的数字，所以要声明 4 个变量：

```
int level;
Sprite *background;
Label *label;
static const int nums[16];
```

Level 的初始值为 0，nums 是一个静态数组：

```
const int Cell::nums[16]={0,2,4,8,16,32,64,128,256,512, 1024,2048,
4096, 8192,16384,32768};
```

background 和 label 在 init 方法中添加到方块类中：

```

        auto cache=SpriteFrameCache::getInstance();
        this->background=Sprite::createWithSpriteFrame(cache->
spriteFrameByName("num_0.png"));
        this->background->setPosition(Point::ZERO);
        this->addChild(background);

        this->label=Label::create(String::createWithFormat
("%d",Cell::nums[level])->getCString(),"Arial",40);
        this->label->setPosition(Point::ZERO);
        //this->label->setString(String::createWithFormat("%d",
Cell::nums[level])->getCString());
        this->addChild(label,1);

```

然后再添加一个 `setLevel` 方法，用来改变方块的属性：

```

void Cell::setLevel(int l){
    auto cache=SpriteFrameCache::getInstance();
    this->level=l;
    this->background->setDisplayFrame(cache->spriteFrameByName
(String::createWithFormat("num_%d.png",Cell::nums[level])->getCString(
)));
    this->label->setString(String::createWithFormat("%d",Cell::nums
[level])->getCString());
}

```

13.3.4 初始化游戏数据

回到 `GameLayer` 类，添加 `gameInit` 方法用来初始化游戏数据。主要功能是生成 16 个 0 级别的数字块，并随机生成两个数字块 2 和 4。

(1) 生成 16 个 0 级别的数字块，默认要隐藏：

```

//初始化砖块
for(int i=0;i<4;i++){
    for(int j=0;j<4;j++){
        auto cell=Cell::create();
        //cell->setAnchorPoint(Point::ZERO);
        cell->setPosition(Point(RC_CONVERT_TO_XY(j),RC_

```



```

CONVERT_TO_XY(i));
    cell->setVisible(false);
    mainpanel->addChild(cell,1);
    tables[i][j]=cell;

}
}

```

(2) 使用 C++ 的随机函数生成两组下标:

```

//C++11 的随机数产生方式
default_random_engine e(time(NULL));
//这里是设定产生的随机数的范围, 这里是 0 到 3
uniform_int_distribution<unsigned> u(0,3);
int row1=u(e);
int col1=u(e);
int row2=u(e);
int col2=u(e);
//这个循环是保证两个砖块的坐标不会重复
do{
    row2=u(e);
    col2=u(e);
}while(row1==row2&&col1==col2);

```

(3) 添加第一个下标的方块, 它的数字也是随机生成:

```

//添加第一个砖块
auto cell11=tables[row1][col1];
int isFour=e()%10;
if(isFour==0){
    cell11->setLevel(2);
    cell11->setVisible(true);
}else{
    cell11->setLevel(1);
    cell11->setVisible(true);
}

```

(4) 同理添加第二个方块。这样初始化函数已经添加成功。在 `init` 函数中调用 `gameInit`。运行程序, 看到初始两个数字块, 如图 13-8 所示。

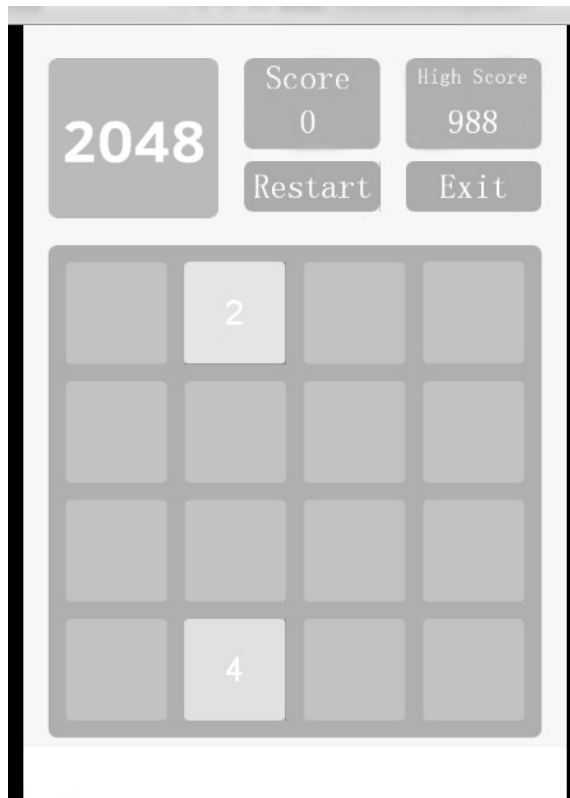


图 13-8 初始两个数字块

13.3.5 添加按钮功能

该游戏有两个按钮，功能分别是重新开始游戏和退出游戏。

(1) 要从 UI 界面中获取两个按钮：

```
Button* restart = (Button*)Helper::seekWidgetByName(UI, "restart");  
Button* exit = (Button*)Helper::seekWidgetByName(UI, "exit");
```

(2) 为两个按钮注册回调函数：

```
//添加退出和重新开始按钮  
exit->addTouchEventListener(this,
```

```

toucheventselector(GameLayer:: gameExit));
    restart->addEventListener(this,
    toucheventselector(GameLayer:: gameRestart));

```

(3) 实现 gameExit 函数，退出游戏：

```

void GameLayer::gameExit(Ref* sender, TouchEventType type){
    #if(CC_TARGET_PLATFORM==CC_PLATFORM_WP8||CC_TARGET_
PLATFORM==CC_ PLATFORM_WINRT)
        MessageBox("You pressed the close button. Windows Store Apps
do not implement a close button.", "Alert");
        return;
    #endif
        Director::getInstance()->end();

    #if(CC_TARGET_PLATFORM==CC_PLATFORM_IOS)
        exit(0);
    #endif
}

```

(4) 实现从新开始游戏函数 gameRestart:

```

void GameLayer::gameRestart(Ref* sender, TouchEventType type){
    Director::getInstance()->replaceScene(GameLayer::
createScene());
}

```

(5) 运行程序，测试两个按钮的功能。

13.3.6 添加事件监听

(1) 当手指滑动时，要监听到滑动事件，所以在 init 方法中加入静态代码：

```

//添加监听器
auto listener=EventListenerTouchOneByOne::create();
listener->setSwallowTouches(true);
listener->onTouchBegan=CC_CALLBACK_2(GameLayer::onTouchBegan,
this);
listener->onTouchMoved=CC_CALLBACK_2(GameLayer::onTouchMoved,
this);

```

```
listener->onTouchEnded=CC_CALLBACK_2(GameLayer::onTouchEnded,
this);
_eventDispatcher->addEventListenerWithSceneGraphPriority(listener,
this);
```

(2) 由于要判断手指滑动的方向, 所以要记录开始触摸时的坐标 `touchDown`:

```
bool GameLayer::onTouchBegan(Touch *touch, Event *unused_event){
    this->touchDown=touch->getLocationInView();
    this->touchDown=Director::getInstance()->convertToGL
(this->touchDown);
    return true;
}
```

(3) 在 `onTouchEnded` 函数中得到结束触摸时的坐标。并声明一个变量 `hasMoved`, 标志方块是否移动:

```
Point touchUp=touch->getLocationInView();
touchUp=Director::getInstance()->convertToGL(touchUp);
```

(4) 手指滑动方向的原理是, 如果沿 X 轴方向移动的距离比沿 Y 轴移动的方向大, 则表示是左右滑动 (反之则是上下移动), 结束时的 x 坐标减去开始时的 x 坐标, 如果是正数, 表示向右滑动, 反之向左滑动。

```
if(touchUp.getDistance(touchDown)>50){
    //判断上下还是左右
    if(abs(touchUp.x-touchDown.x)>abs(touchUp.y-touchDown.y)){
        //左右滑动
        if(touchUp.x-touchDown.x>0){
            //向右
            log("toRight");
            hasMoved=moveToRight();
        }else{
            //向左
            log("toLeft");
            hasMoved=moveToLeft();
        }
    }else{
        //上下滑动
        if(touchUp.y-touchDown.y>0){
            //向上
```

```

        log("toTop");
        hasMoved=moveToTop();
    }else{
        //向下
        log("toDown");
        hasMoved=moveToDown();
    }
}

```

13.3.7 实现方块上下左右移动

上一节判断了是向哪个方向滑动并调用相应的方法。本节以向下滑动的实现为例讲解如何实现滑动后数字块的合并。

移动时先将相同数字的方块合并。合并原理类似冒泡排序，过程查看流程图 13-9。

根据上面的流程图，写出实现代码：

```

for(int col=0;col<4;col++){
    for(int row=0;row<4;row++){
        //遍历的每一次获得的方块
        auto tiled=tables[row][col];
        //找到不为空的方块
        if(tiled->level!=0){
            int k=row+1;
            //看这一列有没有等级和这个方块等级相同的
            while(k<4){
                auto nextTiled=tables[k][col];
                if(nextTiled->level!=0){
                    if(tiled->level==nextTiled->level){
                        //找到等级和这个砖块等级相同的就把他们合并
                        tiled->setLevel(nextTiled->level+1);
                        nextTiled->setLevel(0);
                        nextTiled->setVisible(false);
                        GameLayer::score+=Cell::nums[tiled->level];
                    }
                    this->lScore->setText(String::createWithFormat("%d",GameLayer::score)->getCString());
                }
                k++;
            }
        }
    }
}

```

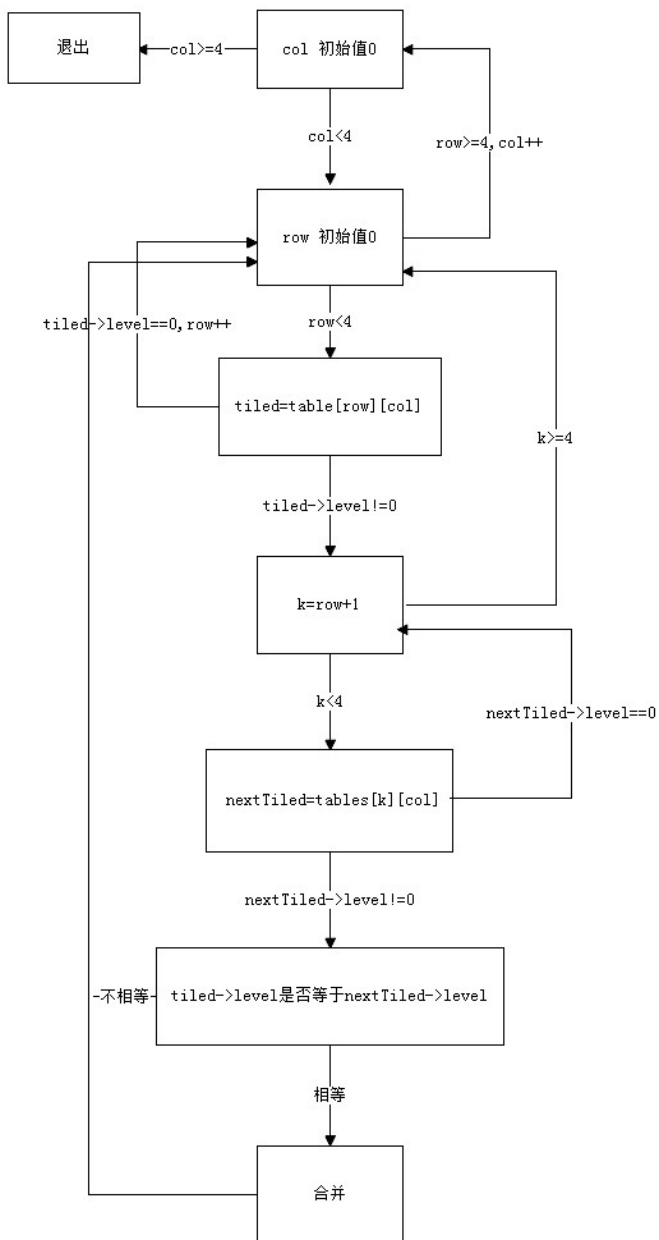


图 13-9 合并流程图

```

        hasMoved=true;
    }//if
    k=4;
} //if
k++;
} //while
} //if
}
}

```

合并数字后，要把有数的方块显示出来：

```

for(int col=0;col<4;col++){
    for(int row=0;row<4;row++){
        //遍历每一次的砖块
        auto tiled=tables[row][col];
        //找到空格子
        if(tiled->level==0){
            int k=row+1;
            while(k<4){
                auto nextTiled=tables[k][col];
                if(nextTiled->level!=0){
                    //将不为空的格子移到这里
                    tiled->setLevel(nextTiled->level);
                    nextTiled->setLevel(0);
                    tiled->setVisible(true);
                    nextTiled->setVisible(false);
                    hasMoved=true;
                    k=4;
                }
                k++;
            }
        }
    }
}
}
}

```

这样就实现了向下移动的代码。同理实现向上、左、右的逻辑代码。

13.3.8 添加新的数字块

当移动后要，要在随机的位置添加新的数字块。所以在 `onTouchEnded` 中添加代码：

```
if (hasMoved) {
    addCells();
}
```

`addCells` 完成数字块的添加功能。先随机到一个坐标：

```
//获取两个随机坐标
default_random_engine e(time(NULL));
uniform_int_distribution<unsigned> u(0,3);
int row=0;
int col=0;
do{
    row=u(e);
    col=u(e);
}while (tables[row][col]->level!=0);
```

在随机到的坐标处添加数字块：

```
//添加砖块
auto cell=tables[row][col];
int isFour=e()%10;
if (isFour==0){
    cell->setLevel(2);
    cell->setVisible(true);
}else{
    cell->setLevel(1);
    cell->setVisible(true);
}

cell->setScale(0.5,0.5);
cell->runAction(ScaleTo::create(0.1f,1.0f));
```


13.3.9 判断游戏是否结束

当移动一次后，要判断游戏是否结束，所以添加一个函数 `isOver` 来判断游戏是否结束。

判断逻辑就是，对 16 个数字块进行遍历，当有空的格子时没有结束，再判断相邻格式数字是否相等，如果相等游戏没有结束，否则就介绍。

```
bool GameLayer::isOver() {
    for(int row=0;row<4;row++){
        for(int col=0;col<4;col++){
            //判断是否存在空格子
            if (tables[row][col]->level==0) {
                //有空格子肯定不会 OVER
                return false;
            }
            //判断周围格子, 如果存在相等的数字则不 OVER
            //上
            int c=col;
            int r=row+1;
            if (r!=-1&&r!=4) {
                if (tables[row][col]->level==tables[r][c]->level) {
                    return false;
                }
            }
            //左
            c=col-1;
            r=row;
            if (c!=-1&&c!=4) {
                if (tables[row][col]->level==tables[r][c]->level) {
                    return false;
                }
            }
            //右
            c=col+1;
            r=row;
            if (c!=-1&&c!=4) {
```

```

        if (tables[row][col]->level==tables[r][c]->level) {
            return false;
        }
    }
    //下
    c=col;
    r=row-1;
    if (r!=-1&& r!=4) {
        if (tables[row][col]->level==tables[r][c]->level) {
            return false;
        }
    }
}
return true;
}

```

将该函数添加到 `onTouchEnded` 中，如果游戏结束，判断当前分数是否比已有最高分数大，如果大，就把现有分数保存为最大分数，然后切换到游戏介绍场景：

```

if (isOver()) {
    //存放分数
    int high=UserDefault::getInstance()->getIntegerForKey("HighScore",
0);

    if (GameLayer::score>high) {
        UserDefault::getInstance()->setIntegerForKey("HighScore",
        GameLayer::score);
        UserDefault::getInstance()->flush();
    }
    GameLayer::score=0;
    //切换画面
    Director::getInstance()->replaceScene
(TransitionSlideInB::create(1.0f,Over::createScene()));
}

```

13.3.10 添加游戏介绍界面

创建类 Over，在界面上添加一个GameOver 的文本标签：

```
auto label=Label::create("GAME OVER","Arial",50);
label->setPosition(Point(size.width/2,size.height/2+30));
this->addChild(label);
```

再添加一个菜单，从新开始游戏：

```
auto mLabel=Label::create("Restart","Arial",30);
auto uiRestart=MenuItemLabel::create(mLabel,CC_CALLBACK_1(Over::
restartMenu,this));
uiRestart->setPosition(Point(size.width/2,size.height/2-10));
auto menu=Menu::create(uiRestart,NULL);
addChild(menu);
```

这样，一个简单的 2048 游戏已经制作成功。运行体验一下吧，如图 13-10 所示。

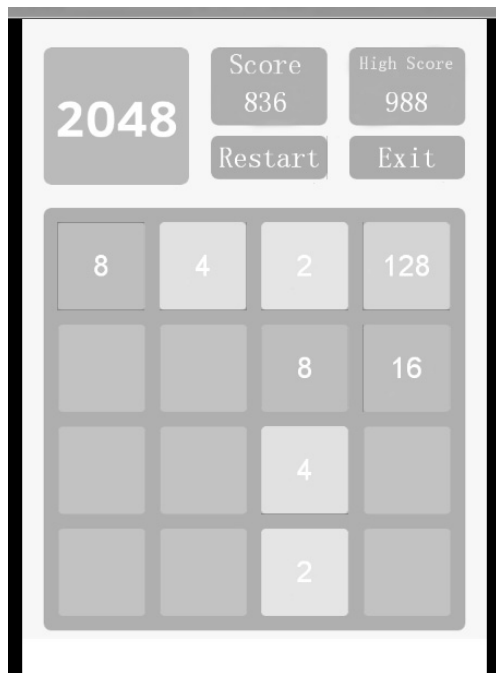


图 13-10 游戏制作成功

14

第 14 章

使用 Cocos2d-x 制作 水浒卡牌游戏

卡牌一直都是玩家青睐的游戏，它具有完整的升级养成系统，有炫酷的战斗动画，有各种特色活动。大家对比较经典的卡牌游戏都耳熟能详了，比如我叫 MT、大掌门、刀塔传奇等。本章就带领读者完成一款卡牌游戏的制作。制作一款卡牌游戏是一个很浩大的工程，这里讲解的是一款功能完整的单机卡牌游戏。

14.1 准备工作

在开发之前要准备好开发环境，这些都在前面的章节中有详细讲解，这里只介

绍下本案例的环境。

- ◎ 操作系统使用 Windows 7。
- ◎ 框架使用 Cocos2d-x 3.0 正式版。
- ◎ 开发工具使用 Visual Studio 2012 专业版。
- ◎ UI 编辑器使用 CocoStudio V1.4.0.1 + Cocos2D-X 3.0。
- ◎ Photoshop 等作图工具。

先用 Photoshop 等准备好需要用到的图片资源，包括游戏 UI、英雄图片、英雄头像、技能图片、技能图片等。准备过程就不用详细介绍，不是本书重点。接下来就进入本章的重点内容。

14.2 定义游戏数据结构和存储单例

该游戏中包含多个实体，每个都有固定的属性，所以用结构体的形式定义这些实体。定义放在 GameData.h 文件中。

首先，卡牌游戏中最重要的当然是卡牌英雄了。一个卡牌英雄的属性包括名称、技能、血量、攻击、防御等信息。详细的定义和属性解释如下。

```
typedef struct
{
    int type;//类型
    string textureName;//图片
    string name;//名称
    string exname;//绰号
    string starname;
    string skillsname;//技能
    int index;//编号
    int pos;//位置
    int hp;//卡牌的血
    int ap;//卡牌攻击
    int totalhp;//总的血量
    int totalap;//总的攻击
```

```

int defend;//物防
int magicdefend;//魔防
int totalmagicdefend;//总魔防
int totaldefend;//总的防御
int attackid;//攻击方式
int skillsid;//技能
int level;//等级
int exlevel;//品质
int xp;//经验
int card;//黑 1, 红 2, 梅 3, 方 4
int number;//2 j11 Q12 k13 A14 王 14,大王 15
int swordid;//武器 id;
int baowuid;//宝贝;
int magic;//是否魔法攻击
ZhuangBeiType zhuang[6];//装备
string des;//简介
}HeroType;

```

除了卡牌英雄，还有一个重中之重的是装备实体 `ZhuangBeiType`。

```

typedef struct
{
    int type;//类型
    int index;//编号
    int level;//等级
    int exlevel;
    int defender;//物防
    int magicdefender;//魔防
    int ap;//攻击
    int hp;//血量
    int star;//星级
    string name;//名称
    string textureName;//图片
    int user;//英雄编号
    int money;//价格
}ZhuangBeiType;

```

除了这两个，还定义了武器类型 `SwordType`、衣服类型 `ClothType`、头盔类型 `HeadType`、坐骑类型 `HorseType`、饰品类型 `JieZhiType`、宝物类型 `BaowuType`。在此只举例列出武器类型。其他类型查看代码 `GameData.h`。

```
typedef struct
{
    int ap;//攻击
    int type;//类型
    int level;//等级
    int star;//星级
    string textureName;//纹理
    string name;//名称
    int index;//编号
}SwordType;
```

GameData 中除了定义数据结构外, 该游戏的数据存储使用单例模式, **GameData** 也负责游戏进行中数据的提取和存放。本节实现 **GameData** 基础功能。具体数据操作功能在相应章节中添加到 **GameData** 类中。

首先通过静态方法 `shareGameData` 获取 **GameData** 实例:

```
GameData * GameData::shareGameData() {
    if(!game){
        game=new GameData();
    }
    return game;
}
```

在构造函数中初始化成员变量, 并从 **UserDefault** 中获取当前等级、经验、钱和金币:

```
mlevel=UserDefault::getInstance()->getIntegerForKey("level",1);
mxp=UserDefault::getInstance()->getIntegerForKey("xp",1);
mMoney=UserDefault::getInstance()->getIntegerForKey("money",
20000);
mCoin=UserDefault::getInstance()->getIntegerForKey("coin",200);
```

还要从 `myzhuangbei.json` 文件中获取当前装备。把该文件中的数据读取出来:

```
CSJson::Reader reader;
    ssize_t  nSize = 0;
    string path=CCFileUtils::sharedFileUtils()->getWritablePath();
    path.append("myzhuangbei.json");
    char* json_document=(char*)CCFileUtils::sharedFileUtils()->getFileData
(path.c_str(), "rb",&nSize);
```

还要获取装备类型:

```
map<int,ZhuangBeiType> ZhuangTypes=Config::sharedConfig()->
getZhuangBei();
```

Config 类用来统一获取游戏配置的配置类 getZhuangBei 的实现方法如下:

```
CSJson::Value Headroot;
CSJson::Value heads;
CSJson::Reader reader;
ssize_t nSize = 0;
string path =CCFileUtils::sharedFileUtils()->fullPathForFilename
("cloth.json");
char* json_document=(char*)CCFileUtils::sharedFileUtils()->
getFileData(path.c_str(), "rb",&nSize);
//printf("head%s",json_document);

if(reader.parse(json_document, Headroot))
{
    heads=Headroot["datas"];

    for (int i=0; i<heads.size(); i++)
    {
        ZhuangBeiType headType;
        int type= heads[i]["type"].asInt();
        headType.type =type;
        headType.name =heads[i]["name"].asString();
        headType.textureName =heads[i]["picture"].asString();
        headType.defender = heads[i]["defend"].asInt();
        headType.magicdefender = heads[i]["magic"].asInt();
        headType.star = heads[i]["star"].asInt();
        headType.hp = heads[i]["hp"].asInt();
        headType.ap = heads[i]["ap"].asInt();
        m_ZhuangTypes.insert(map<int,ZhuangBeiType>::value_type
        (type,headType));
    }
}
```

获取装备类型后, 解析 json_document, 把 json 数据存到 root 中。如果解析成功, 遍历装备和属性叠加。


```

for (int i=0; i<size; ++i)
{
    ZhuangBeiType zhuang;
    int level=root[i]["level"].asInt();//等级
    int type=root[i]["type"].asInt();//类型
    zhuang.exlevel=root[i]["exlevel"].asInt();//经验
    zhuang.index=root[i]["index"].asInt();//索引
    zhuang.type=type;
    zhuang.level=level;
    zhuang.ap=0;
    zhuang.hp=0;
    zhuang.user=root[i]["user"].asInt();
    zhuang.magicdefender=0;
    zhuang.defender=0;
    map<int,ZhuangBeiType>::iterator iter = ZhuangTypes.find
(type);

    if(iter!= ZhuangTypes.end())
    {
        ZhuangBeiType headtype=iter->second;
        zhuang.star=headtype.star;//星级
        zhuang.name=headtype.name;
        zhuang.hp=headtype.hp+headtype.hp*level/10;
        zhuang.ap=headtype.ap+headtype.ap*level/10;
        zhuang.textureName=headtype.textureName;//图片
        zhuang.defender=headtype.defender+headtype.defender*
level/10;
        zhuang.magicdefender=headtype.magicdefender+headtype.
magicdefender*level/10;
        m_ZhuangBeis.push_back(zhuang);
    }
}

```

关于 GameData 存储数据的其他方法，在相应章节中再详细讲解。

14.3 添加登录界面

由于是单机版卡牌游戏，要添加的登录界面其实更多的功能是一个开场动画，

介绍游戏的故事背景。在登录界面里采用玩家和 NPC 对话的形式，当单击屏幕时，切换到下一段对话。

首先，我们添加一个类 **LoginScene**，使用该类实现当前登录界面。在登录游戏时先判断是不是第一次登录，如果是第一次登录，就加载该登录界面，如果不是第一次，直接进入游戏主界面。前面的章节讲过少量数据可以使用 **UserDefault** 快速存储，所以这里就使用 **UserDefault** 存储是否是第一次登录游戏。

打开 **AppDelegate.cpp** 文件，然后加入这段代码获取是不是第一次登录：

```
bool bfirst=UserDefault::sharedUserDefault()->getBoolForKey("first",
false);
```

第一次登录时 **bfirst** 的值是 **false**，然后加个判断，如果 **bfirst** 为 **false**，设置 **first** 值为 **true**，加载登录场景；如果 **bfirst** 为 **true**，就直接进入游戏主界面，游戏主界面为类 **SceneGame**，下一节会详细讲解。

```
if(!bfirst)
{
    UserDefault::sharedUserDefault()->setBoolForKey("first",
true);
    UserDefault::sharedUserDefault()->flush();
    Scene *pScene=LoginScene::scene();
    // run
    director->runWithScene(pScene);
}
else
{
    Scene *pScene = new SceneGame(0);
    director->runWithScene(pScene);
}
```

在 **LoginScene** 初始化中，首先添加登录场景背景和对话框：

```
/* 背景 */
Sprite* bgSp = Sprite::create("storybg.png");
bgSp->setPosition(ccp(winSize.width / 2, winSize.height / 2));
...
this->addChild(bgSp,0,0);
/* 对话框 */
```

```
Sprite* chatBg = Sprite::create("chatbg.png");
chatBg->setPosition(ccp(winSize.width/2, chatBg->getContentSize()
().height / 2));
this->addChild(chatBg,1,TAG_CHAT_BG);
```

玩家与 NPC 的对话内容以 json 的格式存放在文件 story.json 中, 该文件的格式和 content 如下:

```
{
  "datas": [
    {
      "name": "我",
      "msg": "当今世界, 金钱利益主宰, 空虚寂寞无聊。还是水浒兄弟好,
时空机器带我穿越到水浒时代",
      "picture": "car108.png"
    },
    ...
    {
      "name": "我",
      "msg": "加入公孙胜队伍, 获得公孙胜卡牌",
      "picture": "car108.png"
    }
  ]
}
```

我们使用 jsoncpp 库来解析处理 json 格式的数据, 读者可以到 jsoncpp 的官网 <http://jsoncpp.sourceforge.net/> 了解更多内容。我们创建一个函数 readJson, 用来解析对话内容。先定义一个成员变量 root, 用来存放读取的数据:

```
CSJson::Value root;
```

然后把 story.json 中的内容读取到 root 中:

```
CSJson::Reader reader;
ssize_t nSize = 0;
string path = CCFileUtils::sharedFileUtils()->fullPathForFilename
("story.json");
char* json_document=(char*)CCFileUtils::sharedFileUtils()->
getFileData(path.c_str(), "rb",&nSize);
reader.parse(json_document, root);
```

数据读取之后要显示出来，所以再添加一个 `showNextMsg` 函数，用来显示下一段对话。首先添加一个整型成员变量 `m_iCurMsgIndex`，用来记录当前是第几段对话。

获取对话的总个数：

```
CSJson::Value datas;
    datas=root["datas"];
    int size = datas.size();
```

如果当前对话索引大于总的对话个数就退出函数：

```
if(m_iCurMsgIndex > size) {
    return;
}
```

如果当前对话索引小于总的对话个数，显示当前对话内容。首先获取当前对话的名称，内容和图片：

```
string name = datas[m_iCurMsgIndex]["name"].asString();
    string msg= datas[m_iCurMsgIndex]["msg"].asString();
    string strpic=datas[m_iCurMsgIndex]["picture"].asString();
```

把对话的角色（即玩家或 NPC）显示到场景中：

```
LabelTTF* nameLab = LabelTTF::create(name.c_str(),"Arial", 32);
    nameLab->setPosition(ccp(bgSize.width * 0.2f, bgColor.height *
0.9f));
    nameLab->setColor(ccRED);
    chatBG->addChild(nameLab,1);
```

然后把对话内容显示到场景中：

```
LabelTTF* msgLab = LabelTTF::create(msg.c_str(),"Arial", 32,Size(300,
0), kCCTextAlignmentLeft);
    msgLab->setPosition(ccp(bgSize.width/2-120,    bgColor.height *
0.4f));
    msgLab->setColor(ccBLUE);
    chatBG->addChild(msgLab,1);
```

最后当前对话索引自增：

```
m_iCurMsgIndex++;
```

如果当前对话索引等于总的对话个数时，我们要把默认的卡片英雄以 json 的格式保存到 myhero.json 文件中。先声明一个英雄对象：

```
CSJson::Value person;
```

一个英雄包含等级、品介、类型、经验、位置、索引和 6 个装备：

```
person["level"] = 1;//等级
    person["exlevel"] = 0;//品介
    person["type"] = 3;//类型
    person["xp"] = 0;//经验
    person["pos"]=1;//位置
    person["index"]=0;//卡牌索引
    person["zhuang0"]=300000;//头盔
    person["zhuang1"]=400000;//盔甲
    person["zhuang2"]=500000;//武器
    person["zhuang3"]=600000;//宝物
    person["zhuang4"]=700000;//鞋子
    person["zhuang5"]=800000;//坐骑
```

我们生成一个对象 CSJson::Value root;用来存储多个英雄：

```
CSJson::Value root;
root.append(person);
```

同理，再添加一个卡牌英雄。然后把 root 写入到文件 myhero.json 文件中：

```
CSJson::FastWriter fw ;
    std::string json_file = fw.write(root);
    FILE* file = fopen(path.c_str(), "wb");
    if (file)
    {
        fputs(json_file.c_str(),file);
        fclose(file);
    }
```

保存好卡牌英雄后，在为英雄添加几个装备，装备信息也是以 json 的格式保存在 myzhuangbei.json 文件中。该格式中，一个装备单元包括这些信息：装备等级、装备品介、装备类型、装备索引和装备在哪个英雄身上。

```
CSJson::Value root;
    CSJson::Value person;
```

```

    person["level"] = 1; //等级
    person["exlevel"] = 0; //品介
    person["type"] = 301; //类型
    person["index"] = 0; //卡牌索引
    person["user"] = -1; //哪个英雄装备
    root.append(person);

```

同理再添加几个装备到 root 中，然后写入文件 myzhuangbei.json 中：

```

CSJson::FastWriter fw ;
    std::string json_file = fw.write(root);
    FILE* file;
    file = fopen(path.c_str(), "wb");
    if (file)
    {
        fputs(json_file.c_str(), file);
        fclose(file);
    }

```

到此，登录场景的功能函数都已经添加完毕，然后给当前场景注册单击事件，当单击屏幕时切换到下一段对话，也就是调用函数 showNextMsg：

```

auto listener = EventListenerTouchOneByOne::create();
    listener->onTouchBegan = CC_CALLBACK_2(LoginScene::onTouchBegan,
this);
    listener->onTouchMoved = CC_CALLBACK_2(LoginScene::onTouchMoved,
this);
    listener->onTouchEnded = CC_CALLBACK_2(LoginScene::onTouchEnded,
this);
    _eventDispatcher->addEventListenerWithSceneGraphPriority(listener,
this);

```

在触摸结束时调用 showNextMsg 函数：

```

void LoginScene::onTouchEnded(Touch* touch, Event *event)
{
    showNextMsg();
}

```

一个完整的登录界面开发完成，运行程序，可以看到如图 14-1 所示登录界面。



图 14-1 登录界面

14.4 添加游戏主场景

在本游戏中，所有的功能界面都是一个 Layer，不同的 Layer 添加到游戏主场景中就组成不同的功能界面。游戏主场景是玩家一直操作的界面，也是整个游戏的中心。玩家要在主场景中加载不同的功能界面。该主场景通过 SceneGame 类实现。

游戏主场景包括游戏名称和 6 个菜单，分别是首页、组队、武将、战争、图鉴、商店，如图 14-2 所示。



图 14-2 主场景和主菜单

在初始化主场景中，先添加背景图和游戏名称，这两个都是图片，详细内容看代码清单。然后添加 6 个菜单：

```
MenuItemImage *item =MenuItemImage::create("home1.png","home2.png",
this, menu_selector(SceneGame::menuPauseCallback));
MenuItemImage *item1 =MenuItemImage::create("group1.png",
"group2.png",this, menu_selector(SceneGame::menuPauseCallback));
MenuItemImage *item2 =MenuItemImage::create("hero1.png",
"hero2.png",this, menu_selector(SceneGame::menuPauseCallback));
MenuItemImage *item3 =MenuItemImage::create("war1.png",
"war2.png",this, menu_selector(SceneGame::menuPauseCallback));
MenuItemImage *item4 =MenuItemImage::create("tujian1.png",
"tujian2.png",this, menu_selector(SceneGame::menuPauseCallback));
MenuItemImage *item5 =MenuItemImage::create("store1.png",
"store2.png",this, menu_selector(SceneGame::menuPauseCallback));
```

并给这些菜单设置 TAG：

```
item->setTag(0);
item1->setTag(1);
item2->setTag(2);
item3->setTag(3);
item4->setTag(4);
item5->setTag(5);
```

最后把菜单项添加到场景中：

```
CCMenu *menu =CCMenu::create(item,item1,item2,item3,item4,item5,NULL);
menu->alignItemsHorizontally();
menu->setPosition(ccp(size.width/2,item->getContentSize().
height/2));
menu->setTag(888);
this->addChild(menu,1000);
```

当单击这 6 个菜单项时，执行回调函数 `menuPauseCallback`，该函数的功能就是根据单击的菜单项在游戏场景中添加不同的功能 Layer。

首先获取单击菜单的 tag：

```
MenuItemImage *item=(MenuItemImage*)sender;
int tag=item->getTag();
```


在添加功能界面时，先要移除所有的界面：

```
removeChildByTag (HOMEGAG);
    removeChildByTag (HEROGAG);
    removeChildByTag (WARGAG);
    removeChildByTag (FIGHTGAG);
    removeChildByTag (HONGDONGGAG);
removeChildByTag (STOREGAG);
```

然后判断 tag，比如 tag 值为 0，就加载游戏首页：

```
if (tag==0)
{

    mHudLayer = new HomeLayer();
    mHudLayer->setPosition(ccp(0, 0));
    mHudLayer->autorelease();
    this->addChild(mHudLayer,1,HOMEGAG);
}
```

tag 为其他值就加载相应的功能界面。后面会依次把所有功能界面都加上。

14.5 添加游戏首页

上一节已经提到游戏主首页，当登录界面播放结束后默认进入的就是首页。

本游戏首页包含的信息和功能按钮有队伍等级、经验进度条、元宝数、铜币数、装备按钮、英雄按钮、登录奖励、升级奖励等信息，如图 14-3 所示。

游戏首页通过 HomeLayer 类实现。

首先添加队伍等级，队伍等级通过两个元素组成，一个是等级的背景，一个是显示等级的 Label，等级背景通过 ImageView 添加到游戏中，文本标签通过 Text 添加到游戏中。添加过程就不详细叙述。有一点要注意的是通过单例模式获取队伍等级。

```
mlevel=GameData::shareGameData()->mlevel;
```



图 14-3 游戏主界面

然后添加经验进度条，先添加进度条背景：

```
ImageView *timebg=ImageView::create();  
timebg->loadTexture("xp2.png")
```

接着需要使用 CCProgressTimer 实现进度条的效果。创建一个进度条对象：

```
Sprite*sp=Sprite::create("xp1.png");  
healthBar=CCProgressTimer::create(sp);
```

设置 healthBar 的类型为 kCCProgressTimerTypeBar：

```
healthBar->setType(kCCProgressTimerTypeBar);
```

设置中点和变化率：

```
healthBar->setMidpoint(ccp(0,0.5));  
healthBar->setBarChangeRate(ccp(1,0));
```

设置进度条百分比：

```
healthBar->setPercentage(((float)mxp/(float)totalxp) *100);
```

接着添加金钱和金币，详细添加过程请参考代码清单，这里就不详细讲解。下面讲解如何添加装备按钮。装备按钮通过 Button 来实现。先创建一个 Button：

```
Button* = Button::create("yinxiong1.png", "yinxiong2.png", "");  
textButton1->setPosition(ccp(textButton1->getContentSize().width/2
```

```
,winSize.height/2));
```

设置 `textButton1` 的 `tag` 并添加到游戏中：

```
textButton1->setTag(12);
ul->addChild(textButton1);
```

为 `textButton1` 注册单击事件，回调函数为 `menuCallback`：

```
textButton1->addEventListener(this, toucheventselector
(HomeLayer::menuCallback));
```

实现 `menuCallback`，先获取单击 `button` 的 `tag`。然后在 `TOUCH_EVENT_ENDED` 事件中处理单击事件。如果 `tag` 的值为 12，即单击英雄按钮，就会在游戏场景中添加英雄列表。英雄列表在下一节中详细讲解。

```
void HomeLayer::menuCallback(Ref *pSender, TouchEventType type)
{
    Button *btn=(Button*)pSender;
    HeroType hero;
    int tag=btn->getTag();
    switch (type)
    {
        case TOUCH_EVENT_BEGAN:
            break;

        case TOUCH_EVENT_MOVED:
            break;

        case TOUCH_EVENT_ENDED:
            .....
            if(tag==12)
            {
                HeroListLayer *list=new HeroListLayer();
                list->init(hero,-1);
                list->autorelease();
                addChild(list,10);
            }
            .....
            break;
        case TOUCH_EVENT_CANCELED:
```

```

        break;

        default:
            break;
    }

}

```

同理，添加其他 3 个功能按钮。

当玩家的等级升高后，要实时更新首页上的等级和经验进度条信息。所以在初始化函数中添加升级事件的监听器：

```

CCNotificationCenter::sharedNotificationCenter()->addObserver(this,
callfuncO_selector(HomeLayer::healthBarLogic),MSG_LEVEL_UPDATE, NULL);

```

在 `healthBarLogic` 中完成的功能更新等级、进度条、金钱和金币。

获取升级后的等级和经验：

```

int level=GameData::shareGameData()->mlevel;
int xp=GameData::shareGameData()->mxp;
int totalxp=getexp(1.1,level);
if(xp>=totalxp)
{
    mxp-=totalxp;
    xp-=totalxp;
    level++;
    mlevel++;
}

```

获取等级和经验后更新等级和进度条：

```

CCString *lelstr=CCString::createWithFormat("%d",level);
levellabel->setText(lelstr->getCString());
healthBar->setPercentage(((float)xp/(float)totalxp)*100);

```

更新金钱：

```

int money=GameData::shareGameData()->mMoney;
CCString *strMoney=CCString::createWithFormat("%d",money);
moneyLabel->setString(strMoney->getCString())

```

更新金币:

```
int coin=GameData::shareGameData()->mCoin;
    CCString *strCoin=CCString::createWithFormat("%d",coin);
    coinLabel->setString(strCoin->getCString());
```

到此, 游戏的首页已经添加完成。

14.6 添加英雄卡牌列表界面

在 14.5 节讲到在主页单击英雄按钮会出现英雄列表界面, 本节详细讲解如何实现英雄列表界面, 如图 14-4 所示。



图 14-4 英雄列表界面

英雄列表界面通过 HeroListLayer 实现, 每一个英雄包括多个元素: 英雄头像、英雄名、英雄等级、血量、攻击力、升级按钮、上阵按钮。

如果有很多英雄在一屏显示不完时, 使用 ScrollView 来滚动查看其他英雄。

首先创建一个 ScrollView, 设置大小、位置并添加到游戏场景中:

```
ui::ScrollView* scrollView = ui::ScrollView::create();
    //scrollView->setTouchEnabled(true);
    scrollView->setBounceEnabled(true);
    scrollView->setSize(Size(winSize.width,600));
```

```
//Size backgroundSize =Size(500,200);
scrollView->setPosition(ccp(0,winSize.height/2-380));
m_pLayer->addChild(scrollView);
```

然后通过 **GameData** 获取英雄列表。获取英雄的详细实现方式请查看代码清单 **GameData.cpp** 文件：

```
vector<HeroType> Heros=GameData::shareGameData()->getHeros();
```

每个英雄单元的宽和高都是固定的，跟英雄单元的背景一样大。而 **scrollView** 的内容高度等于每个单元的高度乘以英雄的个数。所以 **scrollView** 的内容高度计算方式如下：

```
ImageView* imageView = ImageView::create();
imageView->loadTexture("cellbg.png");
float innerWidth = scrollView->getInnerContainerSize().width;
float innerHeight = scrollView->getInnerContainerSize().height +
imageView->getSize().height*Heros.size();
scrollView->setInnerContainerSize(Size(innerWidth, innerHeight));
```

然后通过一个 **for** 循环变量英雄并添加到 **scrollView** 中：

```
for(int i=0;i<Heros.size();i++)
{
    ....
}
```

在上面的 **for** 循环体内添加单个英雄。首先把背景添加进去，位置设置为从上向下排列：

```
ImageView* imageView = ImageView::create();
imageView->loadTexture("cellbg.png");

int imageHeight=imageView->getSize().height;
imageView->setPosition(ccp(innerWidth /2,scrollView->
getInnerContainerSize().height-imageHeight*j-imageHeight/2));
scrollView->addChild(imageView);
```

下面添加英雄头像，先获取英雄头像的图片资源：

```
char strImage[32];
sprintf(strImage,"head%d.png",hero.type);
```

然后通过 `ImageView` 添加到 `scrollView` 中：

```
ImageView* imagehead= ImageView::create();
    imagehead->loadTexture(strImage);
    imagehead->setPosition(ccp(120,scrollView->getInnerContainerSize()
.height-imageHeight*j-imageHeight/2));
    scrollView->addChild(imagehead);
```

安装同样的方式，使用合适的 `ui` 把英雄名、等级、血量、攻击力加入到游戏当中。

每个英雄单元都有一个按钮，该按钮的功能会跟着当前英雄的情况而变化。

```
Button* button = Button::create();
    button->setTouchEnabled(true);
    if (mindex==-1) //升级
    {
        button->loadTextures("heroup1.png", "heroup2.png", "");
        button->addTouchEventListner(this, toucheventselector
(HeroListLayer::menuPauseCallback));
    }
    else if (mindex>0) //换将
    {
        button->loadTextures("changhero1.png", "changhero2.png",
        "");
        button->addTouchEventListner(this, toucheventselector
(HeroListLayer::menuCallback));
    }
    else if (mindex==-2) //英雄上阵
    {
        button->loadTextures("changhero1.png", "changhero2.png",
        "");
        button->addTouchEventListner(this, toucheventselector
(HeroListLayer::menuAddhero));
    }
    .....
    scrollView->addChild(button);
```

到此，首页的查看英雄列表的功能添加完成，其他功能点的完整实现请查看代码清单。这里就不赘述。

14.7 添加战斗流程

战斗是卡牌游戏的灵魂，有了喜欢的英雄，给英雄装备一套好的装备，这时不拿出来用多没意思啊！战斗让你的英雄久经沙场，变得更强大。该游戏中的战斗流程是，单击战争菜单，首先进入世界地图，如图 14-5 所示，单击世界地图上的某个势力范围，进入该势力范围的推图，再单击其中一个推图进入战斗场景。



图 14-5 世界地图

世界地图通过 `GameMapLayer` 实现。在 `GameMapLayer` 初始化中，添加一个 `ScrollView`，用来滚动显示隐藏的地图：

```
ui::ScrollView* dragPanel = ui::ScrollView::create();
```

设置 `dragPanel` 的大小为窗口大小：

```
Size widgetSize = Director::getInstance()->getWinSize();  
dragPanel->setSize(widgetSize);
```

设置 `dragPanel` 的内容大小为 640×1920 ：

```
Size backgroundSize = Size(640,1920);  
dragPanel->setInnerContainerSize(backgroundSize);
```


设置位置添加到场景中：

```
dragPanel->setPosition(ccp((widgetSize.width - backgroundSize.width)
/ 2, (widgetSize.height - backgroundSize.height) / 2 +
(backgroundSize.height - dragPanel-> getContentSize().height) / 2));

m_pLayer->addChild(dragPanel);
```

世界地图是用 CocosStudio 制作的 UI，这里省略详细的制作过程，因为上一个实例已经详细讲解了 CocosStudio 怎制作 UI。我们把 UI 添加到 dragPanel 中：

```
Layout* map_root = dynamic_cast<Layout*>(GUIReader::shareReader()->
widgetFromFile("GameMapLayer_1.json"));
dragPanel->addChild(map_root);
```

然后定义一个数组，代表势力范围的名称：

```
string str[21] = {"yananfu","shijiachun","shimiao","yezhulin",
"changzhou","hunxian","erlongsan","qingzhou","jiangzhou","liangsan","z
hujiashuang","beijing","chengtou","dongpingfu","dongjing","liangsan","
jizhou","shuzhou","hangzhou","guanfu"};
```

通过一个 for 循环遍历 UI 地图中的 Button，并为 Button 添加名称和回调函数：

```
for (unsigned int i = 0; i<20; ++i)
{
    Button* button = (Button*)(map_root->getChildByTag(i));
    button->setTitleFontSize(32);
    button->setTitleColor(Color3B(255,0,0));
    button->setTitleText(str[i].c_str());
    button->addEventListener(this,toucheventselector
(GameMapLayer::buttonTouchEvent));
}
```

在回调函数 buttonTouchEvent 中，先获取单击 Button 的 tag 和现在的等级：

```
void GameMapLayer::buttonTouchEvent(Ref *pSender, TouchEventType
type)
{
```

```

Size winSize =Director::getInstance()->getWinSize();
Sprite *bg=Sprite::create("tipbg.png");
Button* pHome=(Button*)pSender;
int tag=pHome->getTag();
int level=GameData::shareGameData()->mlevel;
switch (type)
{
    case TOUCH_EVENT_BEGAN:
        break;

    case TOUCH_EVENT_MOVED:
        break;

    case TOUCH_EVENT_ENDED:
        ....进行操作....
        break;
    case TOUCH_EVENT_CANCELED:
        break;

    default:
        break;
}
}

```

然后在 TOUCH_EVENT_ENDED 中判断等级是否大于等于 tag*5-10，如果是就进入推图场景：

```

if(level>=tag*5-10)
{
    FightLayer *info=new FightLayer(tag);
    addChild(info);
}

```

这时，运行程序，单击世界地图中最上面的势力，进入推图场景，如图 14-6 所示。

推图场景在 FightLayer 类中实现，滚动框使用 TableView。前面章节详解讲解过 TableView 的使用方式，这里就不再赘述。



图 14-6 推图场景

在 `FightLayer` 初始化函数中，首先获取推图数据列表，我们创建一个 `readJson` 方法，用来加载数据。推图数据存放在 `map_(level).json` 文件中，括号中的 `level` 为玩家等级。`readJson` 的实现方法如下：

```
void FightLayer::readJson(){
    CSJson::Reader reader;
    CSJson::Value root;
    ssize_t nSize = 0;
    char str[32];
    sprintf(str, "map_%d.json", mlevel);
    string path = CCFileUtils::sharedFileUtils()->fullPathForFilename
(str);
    map_document=(char*)CCFileUtils::sharedFileUtils()->getFileData
(path.c_str(), "rb",&nSize);
    if(reader.parse(map_document, root)){
        datas=root["data"];
    }
}
```

加载完数据后，要在场景中添加一个 `TableView`，大小 `660×600`，纵向滚动，单单元格从上往下渲染：

```
TableView *tableView=TableView::create(this, Size(660,600));
tableView->setDirection(TableView::Direction::VERTICAL);
```

```

        tableView->setPosition(ccp(winSize.width/2,winSize.height/2));
        tableView->setDelegate(this);
        tableView->setVerticalFillOrder(TableView::VerticalFillOrder::
TOP_DOWN);
        this->addChild(tableView,1);

```

在 `numberOfCellsInTableView` 代理函数中返回数据个数：

```

ssize_t FightLayer::numberOfCellsInTableView(TableView *table){
    return datas.size();
}

```

在 `tableCellAtIndex` 中填充每个单元格。先获取列表中的数据：

```

string tile=datas[idx]["name"].asString();
int card=datas[idx]["card"].asInt();
int sword=datas[idx]["sword"].asInt();
int xp=datas[idx]["xp"].asInt();
int money=datas[idx]["coin"].asInt();

```

使用 `LabelTTF` 把名称显示到单元格里：

```

LabelTTF *pLabel = LabelTTF::create(tile.c_str(), "Arial", 32.0);
pLabel->setColor(ccRED);
pLabel->setPosition(ccp(winSize.width/2,120));
pLabel->setTag(400);
cell->addChild(pLabel);

```

同理，把其他元素也加到单元格中。

当单击单元格时，会转到相应的战斗场景。单击单元格的逻辑在 `tableCellTouched` 中处理。

先获取单击对应的战斗：

```

int tag=cell->getIdx();
CSJson::Value data=datas[tag];

```

然后创建战斗界面，并添加到运行中的场景中：

```

Scene *pScene =Director::getInstance()->getRunningScene();
BattleScene *layer=new BattleScene();
layer->init(data);

```

```
layer->autorelease();
pScene->addChild(layer,10)
```

这样，推图场景就制作完成了。

14.8 添加战斗界面

在 14.7 节完成后，选择一个推图，单击进入战斗场景，如图 14-7 所示。



图 14-7 战斗场景

战斗逻辑在 `BattleScene` 类中实现。在初始化 `BattleScene` 时，先创建 3 个数组类型成员变量 `ene_Arr`、`deadArr`、`RoleArr`，分别用来存放敌人、死亡和上阵英雄。

```
ene_Arr=CCArray::create();
ene_Arr->retain();
deadArr=CCArray::create();
deadArr->retain();
RoleArr=CCArray::create();
RoleArr->retain();
```

然后调用函数 `initRole` 初始化上阵英雄，并把英雄添加到场景中。先通过单例获取所有英雄列表：

```
vector<HeroType> Heros=GameData::shareGameData()->getHeros();
```

接着对英雄列表进行遍历，在遍历过程中，判断英雄是否上阵，如果上阵就把英雄添加到场景中，由于 `HeroType` 只是一个数据结构，只能用来存放数据，所以要创建一个类 `role`，该类具有 `HeroType` 的属性，又具有精灵的特点，把上阵英雄添加到场景中的代码如下：

```
for (int i=0; i<Heros.size(); i++)
{
    HeroType hero=Heros[i];
    if(hero.pos>0)
    {
        role * ro=new role();
        ro->init(hero,this);

        ro->setPosition(ccp(110+210*((ro->pos-1)%3),winSize.height/
2-210*((ro->pos-1)/3)-100));
        ro->autorelease();
        this->addChild(ro);
        RoleArr->addObject(ro);
    }
}
```

同样的道理，添加 `initAttackData` 方法，在游戏初始化时使用该方法添加敌方阵营到游戏场景中。`initAttackData` 的具体实现方式请查考代码清单。

然后在设定一个定时器 `schedule`，每隔一秒执行一次 `BeginFight` 函数，该函数用来实现互相交替攻击。

在 `BeginFight` 函数体中，先判断双方阵营是否还有英雄在游戏场景中。如果都没有，就表示游戏结束：

```
if (RoleArr->count()<=0||ene_Arr->count()<=0)
    bGameOver=true;
```

如果游戏结束，在判断是胜利还是失败，然后退出：

```
if (bGameOver)
{
    if (RoleArr->count()>0)
        msucess=true;
    unschedule(schedule_selector(BattleScene::BeginFight));
```

```

        return;
    }

```

如果游戏没有结束，获取当前攻击的卡牌英雄，并预先设定一个 `bool` 类型变量 `bTurn` 来实现交替攻击。这段交替攻击逻辑实现如下：

```

role *r=getAttackRole(roleIndex);
    if(r&&bTurn)
    {
        r->attack();
        attack(r,7);
        bTurn=false;
    }
    else
    {
        role *enemy=getEnemyRole(enemyIndex);
        if(enemy)
        {
            enemy->attackRole(RoleArr);
            enemyattack(enemy,1);
        }
        else
        {
            begintime=-1;
        }
        bTurn=true;
        roleIndex++;
        enemyIndex++;

        if(enemyIndex>12)
        {
            enemyIndex=7;
            roleIndex=1;
        }
        if(roleIndex>6)
            roleIndex=1;
    }

```

当英雄攻击敌人时，首先执行 `role` 类的 `attack` 方法，`attack` 的功能是选择缩放攻击者并播放音效。同理，敌方攻击英雄时的方法 `attackRole` 也有同样的效果。

上段代码的 **attack** 方法用来确定攻击对象，并实施攻击，技能有 5 种，单体攻击技能、全体攻击技能、攻击前排技能、攻击后排技能和攻击竖排技能。

当技能 id 小于 100 时为单体攻击技能，在攻击之前要首先确定攻击对象，优先攻击前排对面，如果前排对面没敌方，敌方 index 增加。

如果技能 id 介于 100 和 200 之间为全体攻击技能，对敌方进行全体攻击：

```
CCARRAY_FOREACH(ene_Arr, object)
{
    role *r=(role*)object;
    int rd=CCRANDOM_0_1()*39+60;
    int ap=a->mhero.totalap*rd/100.0-r->mhero.
totalmagicdefend;
    r->hurt(skills,ap,from,r->getPosition());
}
```

如果技能 id 介于 200 和 300 之间为前排攻击技能，对敌方前排进行攻击：

```
bool bhead=false;
CCARRAY_FOREACH(ene_Arr, object)
{
    role *r=(role*)object;
    int rd=CCRANDOM_0_1()*59+120;
    int ap=a->mhero.totalap*rd/100.0-r->mhero.totaldefend;
    if(r->pos<=9)
    {
        bhead=true;
        r->hurt(skills,ap,from,r->getPosition());
    }
    else if(!bhead)
    {
        r->hurt(skills,ap,from,r->getPosition());
    }
}
```

如果技能 id 介于 300 和 400 之间为后排攻击技能，对敌方后排进行攻击 k,这时要判断后排是否有敌人，如果后排有人攻击后排；如果后排没人则攻击前排。如果技能 id 介于 400 到 500 之间为竖排攻击技能。这两种的实现方式请查看代码清单。

14.9 终结

本章讲解了卡牌游戏的登录界面、公共菜单、游戏首页、英雄列表、地图、推图和战斗界面。这些内容只是一款卡牌游戏众多功能中典型的几个功能。幸运的是本书涉及的这款卡牌游戏功能是很完善的，只是限于篇幅和时间，不能完整地详细讲解出来。希望读者拿到完整代码后，自己跑起来，看懂代码结构和功能，最后再单独实现一遍。



附录 A

实例代码清单说明

代码清单	实例说明
2-2	Cocos2d-x 基本数据类型的使用方式，比如 String、Array、Point、Size、Rect、Dictionary、Bool
2-3	Cocos2d-x 中常用宏定义的使用方式，比如生成随机数宏 CCRANDOM_MINUS1_1、CCRANDOM_0_1，角度转换宏 CC_DEGREES_TO_RADIANS、CC_RADIANS_TO_DEGREES，两值交换宏 CC_SWAP，数组遍历宏 CCARRAY_FOREACH 和 CCARRAY_FOREACH_REVERSE，字典遍历宏 CCDICT_FOREACH，属性定义宏 CC_PROPERTY
2-4	锚点的使用方式，节点坐标和世界坐标的相互转换
3-1-2	从场景中移除 Node 节点
3-2-2	使用 Camera 缩放、旋转节点
3-4-2	使用 Scene 创建一个战斗场景

续表

代码清单	实例说明
3-4-3	多个场景相互切换的方式
3-5-2	Layer、LayerColor 的使用方式
4-1	1. 使用 LabelBMFont 显示文本, 获取单个字并设置样式 2. 使用 LabelTTF 显示文本并对文本设置样式 3. 使用 LabelAtlas 在场景中显示自定义字体
4-2-1	介绍 6 种菜单项的区别和使用方式, 在游戏中添加菜单并设置单击菜单后的回调函数
4-2-2	使用菜单制作游戏菜单功能
4-3-1	在游戏中使用 ScrollView 显示多页内容, 监听 ScrollView 的滚动和缩放事件
4-3-3	通过制作一个简单背包讲解如何使用 TableView, 设置 TableView 大小、方向、渲染顺序、数据源。自定义单元格 TableViewCell
4-3-4	当 TableView 里添加菜单时, 如何区分处理是点击菜单还是滑动 TableView
4-4-1	在游戏中添加滑动条控件 ControlSlider, 设置滑动条的样式、最大(小)值。为 ControlSlider 的不同事件设置监听函数
4-4-2	在游戏中添加开关控件 ControlSwitch, 设置开关控件的样式和状态变化时的监听函数
4-4-3	在游戏中添加取色器控件 ControlColourPicker, 设置控件的样式和值变化时的监听函数, 获取选择的颜色值
4-4-4	在游戏中添加电位计控件 ControlPotentiometer, 设置控件的样式和值变化时的监听函数
4-4-5	在游戏中添加步进器控件 ControlStepper, 设置控件的样式和值变化时的监听函数
4-4-6	在游戏中添加按钮控件 ControlButtonr, 讲解了按钮的几种状态, 为按钮添加事件监听函数
4-5	通过制作一个用户登陆界面讲解编辑框的使用方式, 包括设置编辑框的显示形式、输入内容, 注册 4 种事件的代理函数
5-2	瞬时动作举例
5-3	延时动作举例, 如移动、跳跃、旋转、缩放、倾斜变形、曲线运动等。把简单动作联合起来, 实现按顺序播放动作、同时播放动作、逆向播放动作、重复播放动作等
6-2	介绍 Cocos2d-x 对背景音乐和音效的处理方法, 主要内容包括播放、停止、暂停、恢复等操作
7-3	在游戏中使用瓷砖地图, 包括添加地图, 拖拽地图, 在地图中添加移动精灵, 读写 TMX 地图中的图层和瓷砖

续表

代码清单	实例说明
8-2	通过三个例子讲解 Cocos2d-x 中的屏幕触摸事件。 1. 点击屏幕, 移动精灵到点击位置 2. 在屏幕中滑动手指, 控制精灵跟随手指移动 3. 修改触摸事件的优先级
MutiTouchScale	使用多点触摸实现缩放图片
8-4-1	把键盘输入内容显示在屏幕中, 讲解了如何使用 Cocos2d-x 的键盘事件
8-5-1	利用加速计控制小球移动
9-1	使用 UserDefaults 存储修改数据
9-2	1. 判断文件是否存在 2. 设置文件别名 3. 获取文件完整路径 4. 设置文件搜索路径 5. 根据分辨率调用不同的资源 6. 向文件中写入数据 7. 从文件中读取数据 8. 把数据写入 plist 文件 9. 从 plist 文件读取数据
9-3	1. 使用 c 语言接口操作 SQLite 数据库 2. 不使用回调查询 SQLite 数据库
10-1	讲解使用 http 实现网络通信, 包括: 1. GET 方式通信 2. POST 方式通信
10-2	使用 Socket 实现网络通信
10-3	使用 WebSocket 实现网络通信
11-2	演示使用 Box2D 编程的流程和方式
12-1-5	图片抗锯齿处理方式
12-1-7	制作游戏加载场景, 进入游戏前先把所有图片资源加载到内存中

续表

代码清单	实例说明
12-2-2	本例通过实现英雄打斗讲解帧动画的使用方式
12-2-3	讲解如何使用 Sprite Sheet 动画
12-2-5	使用 CocoStudio 制作骨骼动画，并在 Cocos2d-x 项目中播放
2048	2048 休闲游戏的完整开发过程
card	功能完整的水浒卡牌游戏开发详解



附录 B

Cocos2d-x 3.X 主要版本间的区别

Cocos2d-x 版本升级很快，并且每个版本都有很多地方有改变优化。特别是 3.X 和 2.X 之间，它们从底层架构到 API 形式都有翻天覆地的变化。下面从本书截稿时的最新版本开始介绍 Cocos2d-x 3.X 主要版本之间的区别。

Cocos2d-x V3.2

由于 `Node::enumerateChildren()` 使用了 `std::regex`，导致所以 Xcode6 beta3 中使用，在 iOS 上我们使用包括 64 位库文件的 fat library。但是 Xcode 5.0 或更低版本不支持这种方式。所以使用 V3.2 进行开发时用到的编译工具要满足以下要求：

- ◎ iOS 或 Mac 编译需要 Xcode 5.1 或以上版本。
- ◎ Linux 编译需要 gcc 4.9 或以上版本。
- ◎ Android 编译需要 NDK R9D 以上版本。
- ◎ Windows (win32)编译需要 Visual Studio 2012 或以上版本。
- ◎ Windows Phone 8 编译需要 Visual Studio 2012 或以上版本。

除了编译工具发生变化外, V3.2 版本加入了下面这些新特性:

- ◎ 新的 3D 动画节点 Animation3D/Animate3D。
- ◎ 支持 fbx-conv 工具生成 Sprite3D 支持的二进制格式。
- ◎ 支持游戏手柄。
- ◎ 支持快速瓦片地图。
- ◎ 加入 `utils::captureScreen` 方法用于截屏。
- ◎ Physics body 支持缩放和旋转。
- ◎ 加入 `Node::enumerateChildren` 和 `utils::findChildren` 方法, 且支持 C++ 11 的正则表达式。
- ◎ 加入 `Node::setNormalizedPosition` 方法, Node 的位置像素会根据它的父节点的尺寸大小计算。

下面简略介绍下这些特性, 详细使用方法请到作者博客查看。

1. 添加 3D 动画

Sprite3D 和 Animation3D 用来创建 3D 精灵和 3D 动画, 使用方式如下:

```
auto sprite3d = Sprite3D::create("filename.c3b");
addChild(sprite3d);
auto animation3d = Animation3D("filename.c3b");
auto animate3d = Animate3D::create(animation3d); sprite3d->runAction
(RepeatForever::create(animate));
```

并且同时提供了一个工具 **fbx-conv**，用来生成 Sprite3D 支持的二进制格式或文本格式。在 Mac OS X 中使用方式：

```
$ cd COCOS2DX_ROOT/tools/fbx-conv/mac $ ./fbx-conv [-a|-b|-t] FBXFile
```

Windows 环境下的使用方式：

```
cd COCOS2DX_ROOT/tools/fbx-conv/windows fbx-conv [-a|-b|-t] FBXFile
```

fbx-conv 的可选参数如下。

- ⊙ **-a**：输出文本和二进制格式。
- ⊙ **-b**：输出二进制格式。
- ⊙ **-t**：输出文本格式。

2. 游戏手柄

V3.2 加入游戏手柄的功能，它支持的手柄类型包括：

- ⊙ 标准的 Android 手柄
- ⊙ Amazon tv
- ⊙ OUYA
- ⊙ Moga
- ⊙ Nibiru
- ⊙ 标准的 iOS 手柄

简单使用方式如下：

```
//注册事件监听器
auto listener = EventListenerController::create(); listner->onKeyDown
= ... .. eventDispatcher->addEventListenerWithSceneGraphPriority
(listener, this);
//连接控制器
Controller::startDiscoveryController();
//处理 key down/ key up 事件
void GameControllerTest::onKeyDown(Controller *controller, int
```



```
keyCode, Event *event)
{
    switch (keyCode)
    {
        case Controller::Key::BUTTON_A: ... break;
        ...
    }
}
```

3. 添加 Node::enumerateChildren 方法

该方法用于枚举一个 Node 的子节点。它支持 C++ 11 的正则表达式。使用方法如下：

```
//找到名字为"nameToFind+数字"格式的节点
node->enumerateChildren("nameToFind[[:digit:]]+", [](Node* node) ->
bool { ... return false; });
```

4. 添加 Node::setNormalizedPosition 方法

用该方法可以使用 0~1 之间的值设置节点 position(x,y)。当节点有父节点时可以使用该方法。节点的像素位置计算方式如下：

```
void setNormalizedPosition(Vec2 pos)
{
    Size s = getParent()->getContentSize(); _position = pos * s;
}
```

Cocos2d-x V3.1

使用 V3.1 开发的游戏的运行平台要求如下：

- ◎ Android 2.3 或以上版本。
- ◎ iOS 5.0 或以上版本。
- ◎ OS X 10.7 或以上版本。
- ◎ Windows 7 或以上版本。

- ◎ Windows Phone 8 或以上版本。
- ◎ Linux Ubuntu 12.04 或以上版本。

编译环境要求如下：

- ◎ iOS 或 Mac 编译需要 Xcode 4.6 或以上版本。
- ◎ Linux 编译需要 gcc 4.7 或以上版本。
- ◎ Android 编译需要 NDK R9 以上版本。
- ◎ Windows (win32)编译需要 Visual Studio 2012 或以上版本。
- ◎ Windows Phone 8 编译需要 Visual Studio 2012 或以上版本。

V3.1 加入了下面这些亮点特性：

- ◎ 渲染 3D 模型的节点 Sprite3D。
- ◎ 重构了着色器系统。
- ◎ 新的统一的数学库。
- ◎ 添加节点 ui::VideoPlayer，可以用来直接播放视频。

下面简略介绍下这些特性，详细使用方法请到作者博客查看。

1. 添加 Sprite3D

Sprite3D 用来渲染 3D 模型。自从 Cocos2d-x 使用 4×4 转换矩阵后，可以在 x 、 y 、 z 三个方向缩放旋转 Sprite3D 节点。Sprite3D 的使用方式如下：

```
// v3.1 只支持.obj 格式
auto sprite3d = Sprite3D::create("mymodel.obj");

// 如何在.obj 中没有定义图片资源,可以用这种方式重写:
auto sprite3d = Sprite3D::create("mymodel.obj", "texture.png");

// 把 Sprite3D 添加到场景中
scene->addChild(sprite3d);
sprite3d->setRotation3D(Vec3(x, y, z));
```

2. 重构了着色器系统

为了支持 Sprite3D, Cocos2d-x 重构了着色器系统。于是乎，我们有了一个更好用、更强大的着色器系统，该着色器系统能够用在 2D 和 3D 模型中。简单使用方式如下：

```
auto glprogram = GLProgram::create(...);
auto glprogramstate = GLProgramState::create( glprogram );
//设置 Vec2 格式 (支持 Int, Texture, Vec3, Vec4, Mat4,)
glprogramstate->setUniformVec2("u_my_uniform", Vec2(x,y));
//使用回调函数进行设置
glprogramstate->setUniformCallback("u_my_uniform",
[](Uniform*uniform){
    // do something
});
//设置属性
glprogramstate->setVertexAttribPointer("a_my_attrib", 4, GL_FLOAT,
GL_FALSE, 0, vertex);
//使用回调函数设置属性
glprogramstate->setVertexAttribCallback("a_my_attrib",
[](VertexAttrib*attrib){
    // do something
});
```

3. 新的统一的数学库

Cocos2d-x V1.0 只支持 2D 特性，所以只使用 2×3 转变矩阵就可以。在 Cocos2d-x V2.0 中开始支持 OpenGL ES 2.0，所以可以使用 Kazmath 数学库，但是 Kazmath 只在框架内部使用，开发者并不能通过 API 的形式调用 Kazmath。在 V3.1 中添加了 Sprite3D，这就要求暴露更多 Kazmath API 给开发者使用。这就导致了一个问题，即部分代码使用 2D 数学库，部分代码使用 Kazmath 数学库，另外一部分代码使用 ad-hoc 数学库。于是，Cocos2d-x 统一使用 Gameplay3D 数学库，并在 Gameplay3D 的基础上做了一些修改。简单使用方式如下。

```
// vector2
Vec2 vec2(10,20);
Vec2 other2(30,40);
```

```
auto ret = vec2.cross(other2);
auto ret2 = vec2 + other2;
auto ret3 = vec3 * scalar;
node->setPosition(vec2);
// vector3
Vec3 vec3(30,40,50);
node->setPosition3D(vec3);
// vector4
Vec4 vec4 = Vec4::ZERO;
// Matrix 4x4
auto identity = Mat4::IDENTITY;
node->setNodeToParentTransform(identity);
```

4. 添加节点 UIVideoPlayer

使用 UIVideoPlayer，我们可以在 Cocos2d-x 中直接播放视频。VideoPlayer 在 experimental 命名空间下，只支持 iOS 和 Android。简单使用方式如下。

```
auto videoPlayer = cocos2d::experimental::ui::VideoPlayer::create();
videoPlayer->setContentSize(Size(x,y));
videoPlayer->setURL("http://example.com/video.mp4");
//也可以播放本地视频
//videoPlayer->setFileName("filepath/video.mp4");
videoPlayer->play();
```

Cocos2d-x V3.0

使用 V3.0 开发的游戏的运行平台要求如下：

- ◎ Android 2.3 或以上版本。
- ◎ iOS 5.0 或以上版本。
- ◎ OS X 10.7 或以上版本。
- ◎ Windows 7 或以上版本。
- ◎ 暂不支持 Windows Phone 8。

- ◎ Linux Ubuntu 12.04 或以上版本。

编译环境要求如下：

- ◎ iOS 或 Mac 编译需要 Xcode 4.6 或以上版本。
- ◎ Linux 编译需要 gcc 4.7 或以上版本。
- ◎ Android 编译需要 NDK R9 以上版本。
- ◎ Windows (win32)编译需要 Visual Studio 2012 或以上版本。
- ◎ Windows Phone 8 编译需要 Visual Studio 2012 或以上版本。
- ◎ V3.0 加入了下面这些亮点特性：
- ◎ 使用 C++ 11 的语法和最佳实践取代了 Objective-C 的语法。
- ◎ 优化了 Labels。
- ◎ 优化了渲染系统，比 V2.2 快很多。
- ◎ 新的事件分发机制。
- ◎ 集成了物理系统。
- ◎ 新的 UI 系统。
- ◎ JavaScript 远程调试。
- ◎ 支持远程控制台。
- ◎ 使用 Cocos 控制台命令创建和运行项目。
- ◎ 重构 Image，能够及时释放内存，并统一了 API。
- ◎ 自动生成 Lua bindings，添加了 LuaJavaBridge 和 LuaObjcBridge。
- ◎ 数据类型变化。

cocos2d::Map<> 取代 CCDictionary，cocos2d::Vector<> 取代 CCArray，
cocos2d::Value 取代 CCBBool、CCFloat 和 CCDouble。

下面简略介绍下这些特性，详细使用方法请到作者博客查看。

1. 引入 C++ 11 特性

C++ 11 有 `std::function` 的功能，所以可以使用 `std::function<void()>` 创建 `CallFunc`，使用 `std::function<void(Node*)>` 创建 `CallFuncN`，`CallFuncND` 和 `CallFuncO` 被移除。`MenuItem` 支持 `std::function<void(Node*)>` 做回调函数。如下面介绍了多种创建 `CallFunc` 的方式：

```
// in v2.1
CCCallFunc *action1 = CCallFunc::create( this, callfunc_selector
( MyClass::callback_0 ) );

// in v3.0 (short version)
auto action1 = CallFunc::create( CC_CALLBACK_0(MyClass::callback_0,
this));
auto action2 = CallFunc::create(CC_CALLBACK_0(MyClass::callback_1,
this, additional_parameters));

// in v3.0 (long version)
auto action1 = CallFunc::create( std::bind( &MyClass::callback_0,
this));
auto action2 = CallFunc::create( std::bind( &MyClass::callback_1, this,
additional_parameters));

// in v3.0 you can also use lambdas or any other "Function" object
auto action1 = CallFunc::create(
    [&]() {
        auto s = Director::sharedDirector()->getWinSize();
        auto label = LabelTTF::create("called:lambda
callback", "Marker Felt", 16);
        label->setPosition(ccp( s.width/4*1,s.height/2-40));
        this->addChild(label);
    } );
```

重新定义了一些常量和枚举类型变量，如表 B-1 所示。

表 B-1 V2.X 与 V3.0 的常量和枚举类型

V2.X	V3.0
kTypeValue	Type::VALUE

续表

V2.X	V3.0
kCCTexture2DPixelFormat_RGBA8888	Texture2D::PixelFormat::RGBA8888
kCCDirectorProjectionCustom	Director::Projection::CUSTOM
ccGREEN	Color3B::GREEN
CCPointZero	Point::ZERO
CCSizeZero	Size::ZERO

2. 删除 Objective-C 语法

去掉 C++ 函数和方法的 CC 前缀，举例如表 B-2 所示。

表 B-1 V2.X 与 V3.0 的函数和方法举例

V2.X	V3.0
CCSprite	Sprite
CCDirector	Director
ccDrawPoint()	DrawPrimitives::drawPoint()
ccDrawCircle()	DrawPrimitives::drawCircle()
ccGLBlendFunc()	GL::blendFunc()
ccGLBindTexture2D()	GL::bindTexture2D()

clone() 代替 copy() 创建制动释放对象。

```
// V2.1
CCMoveBy *action = (CCMoveBy*) move->copy();
action->autorelease();

// V3.0
// 不需要手动调用 autorelease, 不需要转换类型
auto action = move->clone();
```

所有的单例使用 getInstance() 和 destroyInstance() 创建和销毁，举例如表 B-3 所示。

表 B-3 V2.X 与 V3.0 的 `etInstance()`和 `destroyInstance()`举例

V2.X	V3.0
<code>CCDirector->sharedDirector()</code>	<code>Director->getInstance()</code>
<code>CCDirector->endDirector()</code>	<code>Director->destroyInstance()</code>

3. 重构渲染系统

新的渲染系统具有下面这些特点。

- ◎ 从场景图画中独立出来，`draw()`函数会向 `Renderer` 发送一个 `RenderCommand` 命令，`Renderer` 负责绘制队列中的 `RenderCommand`。
- ◎ `quadCommands` 会自动被批量处理。
- ◎ `CustomCommand` 对象允许程序使用自定义的 `OpenGL` 代码
- ◎ 全局显示顺序。

4. 优化了 `LabelTTF` / `LabelBMFont` / `LabelAtlas`

新的 `Label` 会代替 `LabelTTF`, `LabelBMFont` 和 `LabelAtlas`，这带来的好处是：

- ◎ 使用统一的 API 创建 `LabelTTF`、`LabelBMFont` 和 `LabelAtlas`。
- ◎ 使用 `freetype` 为 `Label` 创建纹理，这样可以保证不同的平台上，`Label` 的显示效果一样。
- ◎ 缓存纹理提高性能。

5. 新的事件分发机制

所有的事件分发函数都被移除，如 `EventDispatcher`、`TouchDispatcher`、`KeypadDispatcher`、`KeyboardDispatcher`、`AccelerometerDispatcher`。新 `EventDispatcher` 包括下面这些特性：

- ◎ 基于渲染队列分发事件。
- ◎ 所有事件都被 `EventDispatcher` 分发。
- ◎ 使用 `EventDispatcher` 分发自定义事件。

6. 更改全局变量和全局方法。

Cocos2d-x V3.0 移除了 ccTypes.h 中定义的结构体的 CC 前缀，把全局函数改成静态成员函数，把全局变量改成静态全局变量，如表 B-4 所示。

表 B-4 V2.X 与 V3.0 的全局变量和全局方法

V2.X	V3.0
ccColor3B	Color3B
ccColor4B	Color4B
ccColor4F	Color4F
ccVertex2F	Vertex2F
ccVertex3F	Vertex3F
ccTex2F	Tex2F
ccPointSprite	PointSprite
ccQuad2	Quad2
ccQuad3	Quad3
ccV2F_C4B_T2F	V2F_C4B_T2F
ccV2F_C4F_T2F	V2F_C4F_T2F
ccV3F_C4B_T2F	V3F_C4B_T2F
ccV2F_C4B_T2F_Triangle	V2F_C4B_T2F_Triangle
ccV2F_C4B_T2F_Quad	V2F_C4B_T2F_Quad
ccV3F_C4B_T2F_Quad	V3F_C4B_T2F_Quad
ccV2F_C4F_T2F_Quad	V2F_C4F_T2F_Quad
ccBlendFunc	BlendFunc
ccT2F_Quad	T2F_Quad
ccAnimationFrameData	AnimationFrameData

全局函数的改变包括：

```
// in v2.1
ccColor3B color3B = ccc3(0, 0, 0);
ccc3BEqual(color3B, ccc3(1, 1, 1));
ccColor4B color4B = ccc4(0, 0, 0, 0);
ccColor4F color4F = ccc4f(0, 0, 0, 0);
```

```

color4F = ccc4FFFromccc3B(color3B);
color4F = ccc4FFFromccc4B(color4B);
ccc4FEqual(color4F, ccc4F(1, 1, 1, 1));
color4B = ccc4BFromccc4F(color4F);

color3B = ccWHITE;

// in v3.0
Color3B color3B = Color3B(0, 0, 0);
color3B.equals(Color3B(1, 1, 1));
Color4B color4B = Color4B(0, 0, 0, 0);
Color4F color4F = Color4F(0, 0, 0, 0);
color4F = Color4F(color3B);
color4F = Color4F(color4B);
color4F.equals(Color4F(1, 1, 1, 1));
color4B = Color4B(color4F);

color3B = Color3B::WHITE;

```

7. Lua 绑定做了很多改到

在 Cocos2d-x V3.0 中，只需要为一个模块写一个 ini 文件，不需要再写一堆 .pkg 文件。

简单的使用 `ScriptHandlerMgr` 注册和接触注册 Lua 函数。如：

```

ScriptHandlerMgr::getInstance():registerScriptHandler(menuItem,
luafunction, cc.HANDLERTYPE_MENU_CLICKED)

```

不同的类被绑定到不同的模块，不再是绑定到一个全局模块。其中：

- ◎ `cocos2d`、`cocos2d::extension`、`CocosDenshion` 和 `cocosbulde` 中的类对应 `cc` 模块。
- ◎ `cocos2d::ui` 对应 `ccui` 模块。
- ◎ `spine` 对应 `sp` 模块。
- ◎ `cocostudio` 对应 `ccs` 模块。
- ◎ 所有关于 `openGl` 的方法和变量对应 `gl` 模块。