

• 游戏开发经典丛书 •

(英) Daniel Schuller 著

张磊 李苏军 译

精通C# 游戏编程

C# Game Programming For Serious Game Creation

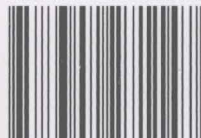
C# Game Programming For Serious Game Creation

即使经验丰富的游戏开发人员，有时也难以将自己的设想转变成为一个优秀的游戏。可用的编程语言、库和生产方法如此之多，使得开发过程变得令人生畏，得到的游戏代码也很容易复杂而不可靠。

本书通过引导读者创建一个基本的游戏，展示了如何使用C#和OpenGL一步步地开发出简单、整洁而可靠的代码。C#是一种高级编程语言，而OpenGL是业界显示图形最常用的方法。本书概述了创建优秀游戏项目时采用的方法和库，讨论了如何使用这些库和创建自己的库，最后帮助读者创建自己的射击类游戏。书中还提供了关于如何实现自己的游戏想法的提示和信息，以及可以采用的代码库，从而帮助读者将自己的游戏想法从概念变为现实。

本书配套资料中附有书中会用到的所有源代码、游戏资源以及有用的游戏开发网站和图形开发网站的链接。

本书配套资料可从<http://www.tupwk.com.cn/downpage>中下载。



• 游戏开发经典丛书 •

精通 C#游戏编程

(英) Daniel Schuller 著

张 磊 李苏军 译

清华大学出版社

北 京

内容提要

本书是教育部高职高专规划教材,是根据《高职高专教育高等数学课程教学基本要求》,并参考《全国各类成人高等学校专科起点本科班招生复习考试大纲(非师范类)》编写的.全书分上、下两册,本书为下册,包括向量代数与空间解析几何、多元函数微分学、多元函数积分学、无穷级数、微分方程等5章,书末附有行列式简介、数学实验、习题答案与提示等

本书将教材与辅导融为一体,一书两用.每章末设“学习指导”,例题、习题丰富,重点内容滚动复习,便于自学.适当拓宽知识面,扩大了适应性,可为继续深造学习“专升本”打下基础.

本书主要适用于工科类高职高专各专业,也可供经管类专业使用,还可作为“专升本”及学历文凭考试的教材或参考书

图书在版编目(CIP)数据

高等数学.下册/同济大学等编. —北京:高等教育出版社,2001.7 (2002 重印)

教育部高职高专规划教材

ISBN 7-04-009957-8

I. 高... II. 同... III. 高等数学-高等学校:技术学校-教材 IV. 013

中国版本图书馆 CIP 数据核字 (2001) 第26171号

高等数学 下册
同济大学 天津大学 编
浙江大学 重庆大学

出版发行 高等教育出版社
社 址 北京市东城区沙滩后街 55 号
邮政编码 100009
传 真 010-64014048

购书热线 010-64054588
免费咨询 800-810-0598
网 址 <http://www.hep.edu.cn>
<http://www.hep.com.cn>

经 销 新华书店北京发行所
印 刷 高等教育出版社印刷厂

开 本 787×1092 1/16
印 张 17.5
字 数 430 000

版 次 2001 年 7 月第 1 版
印 次 2002 年 9 月第 3 次印刷
定 价 26.50 元(含光盘)

本书如有缺页、倒页、脱页等质量问题,请到所购图书销售部门联系调换。

版权所有 侵权必究

作者简介

Daniel Schuller 生于英国，是一名计算机游戏开发人员，曾在美国、新加坡和日本工作和生活，目前在英国工作。他在 PC、Xbox 360 和 PlayStation 3 上发布过游戏。**Schuller** 为 Sony、Ubisoft、Naughty Dog、RedBull 和 Wizards of the Coast 开发过游戏，并运营着一个游戏开发网站，网址为 <http://www.godpatterns.com>。除了开发计算机游戏，**Schuller** 还在学习日文，并对人工智能、认知科学和游戏在教育中的作用非常感兴趣。

致 谢

非常感谢每个对本书出版做出过贡献的人，以及书中所用库和工具的开发者们。感谢策划编辑 Jenny Davidson，她使本书的表达语言更加简洁，并确保我可以及时地完成本书。感谢技术编辑 Jim Perry，他帮助我找出了一些代码错误，并给我提供了很有帮助的建议和意见。最后，感谢我的家人和同事对我的支持和鼓励。

前 言

希望本书能帮助你开发游戏。

每个人都有出色的游戏构思，但是从最初的构思到完成一个项目所应采取的方法并不明朗。众多的编程语言、库和生产方法让人望而生畏。即使经验丰富的游戏开发人员有时候也无法实现自己最初的设想。如果没有优秀可靠的架构，游戏代码可能变得非常复杂，把开发人员淹没其中。代码越复杂，修改和继续开发游戏的难度就越大。

通过开发两个简单的游戏，本书介绍了如何编写简单、整洁、可靠的代码。这些游戏使用 C#编程语言和 OpenGL 创建。C#是一个先进的高级编程语言，所以编写代码更快，需要避免的程序缺陷更少。OpenGL 几乎可以说是游戏业在显示图形时使用的标准方式。读完本书后，你将拥有一个优秀的代码库来进行开发和扩展，以实现自己的想法。

本书的第 I 部分将概述创建优秀游戏的方法和库。第 II 部分将介绍如何使用这些库和如何创建自己的可重用游戏库。你还将学到如何开发简单的卷轴射击游戏，并了解在实现自己的游戏想法时可以采纳的建议和提示。本书的配套资料中包含了开始开发游戏所需的一切。书中的每个代码段在资料中都有完整的源代码和程序。此外，资料中还提供了一些简单的游戏资源，以及指向有价值的游戏开发网站和图形网站的一些链接。本书配套资料可以从 <http://www.tupwk.com.cn/downpage> 中下载。

目 录

第 I 部分 背景知识

| | |
|--------------------|----|
| 第 1 章 C#的历史 | 3 |
| 1.1 C#基础 | 3 |
| 1.2 小结 | 14 |
| 第 2 章 OpenGL 简介 | 15 |
| 2.1 OpenGL 的架构 | 16 |
| 2.1.1 顶点: 3D 图形的基础 | 16 |
| 2.1.2 流水线 | 17 |
| 2.2 变化中的 OpenGL | 19 |
| 2.2.1 OpenGL ES | 19 |
| 2.2.2 WebGL | 19 |
| 2.3 OpenGL 和图形卡 | 20 |
| 2.4 Tao 框架 | 21 |
| 2.5 小结 | 23 |
| 第 3 章 现代方法 | 25 |
| 3.1 实效编程 | 25 |
| 3.1.1 游戏编程中的陷阱 | 25 |
| 3.1.2 KISS | 26 |
| 3.1.3 DRY | 26 |
| 3.1.4 源代码控制 | 30 |
| 3.1.5 单元测试 | 32 |
| 3.2 小结 | 37 |

第 II 部分 实 现

| | |
|------------------------------------|----|
| 第 4 章 设置 | 41 |
| 4.1 Visual Studio Express——C# | |
| 可以使用的免费 IDE | 41 |
| 4.1.1 Hello World 程序 | 42 |
| 4.1.2 关于 Visual Studio Express 的提示 | 44 |
| 4.2 Subversion | 50 |
| 4.2.1 获取 | 51 |
| 4.2.2 安装 | 51 |
| 4.2.3 创建源代码控制库 | 51 |
| 4.2.4 添加到库中 | 52 |
| 4.2.5 历史记录 | 56 |
| 4.2.6 扩展 Hello World | 56 |
| 4.3 Tao | 58 |
| 4.4 NUnit | 58 |
| 4.4.1 在项目中使用 NUnit | 59 |
| 4.4.2 运行测试 | 61 |
| 4.4.3 示例项目 | 63 |
| 4.5 小结 | 66 |
| 第 5 章 游戏循环和图形 | 67 |
| 5.1 游戏的工作方式 | 67 |
| 5.2 使用 C#实现一个快速的游戏循环 | 68 |

| | | | | | |
|-------|------------------------|-----|-------|---------------------|-----|
| 5.3 | 图形 | 76 | 7.5.2 | 对批(batch)绘制方法执行性能分析 | 141 |
| 5.3.1 | 全屏模式 | 79 | 7.6 | 小结 | 141 |
| 5.3.2 | 渲染 | 79 | 第8章 | 游戏数学 | 143 |
| 5.4 | 小结 | 84 | 8.1 | 三角函数 | 143 |
| 第6章 | 游戏结构 | 87 | 8.1.1 | 绘制图形 | 143 |
| 6.1 | 游戏对象的基本模式 | 87 | 8.1.2 | 使用三角函数实现特殊效果 | 147 |
| 6.2 | 处理游戏状态 | 88 | 8.2 | 向量 | 150 |
| 6.3 | 游戏状态演示 | 93 | 8.2.1 | 向量的定义 | 150 |
| 6.4 | 使用投影设置场景 | 95 | 8.2.2 | 长度操作 | 151 |
| 6.4.1 | 字体大小和 OpenGL 视口大小 | 95 | 8.2.3 | 向量的相等性 | 152 |
| 6.4.2 | 宽高比 | 96 | 8.2.4 | 向量加法、减法和乘法 | 153 |
| 6.4.3 | 投影矩阵 | 97 | 8.2.5 | 法向量 | 157 |
| 6.4.4 | 2D 图形 | 97 | 8.2.6 | 点积运算 | 159 |
| 6.5 | 精灵 | 100 | 8.2.7 | 叉积运算 | 162 |
| 6.5.1 | 定位精灵 | 103 | 8.2.8 | 关于向量结构的最后一点内容 | 163 |
| 6.5.2 | 使用四方形管理纹理 | 104 | 8.3 | 二维相交 | 164 |
| 6.5.3 | 纹理精灵 | 109 | 8.3.1 | 圆 | 164 |
| 6.5.4 | alpha 混合精灵 | 111 | 8.3.2 | 矩形 | 169 |
| 6.5.5 | 颜色调制精灵 | 113 | 8.4 | 补间 | 172 |
| 6.5.6 | Sprite 类和 Render 类 | 113 | 8.4.1 | 补间概述 | 172 |
| 6.5.7 | 使用 Sprite 类 | 119 | 8.4.2 | Tween 类 | 173 |
| 第7章 | 渲染文本 | 121 | 8.4.3 | 使用补间 | 176 |
| 7.1 | 字体纹理 | 121 | 8.5 | 矩阵 | 178 |
| 7.2 | 字体数据 | 124 | 8.5.1 | 矩阵的定义 | 178 |
| 7.2.1 | 解析字体数据 | 125 | 8.5.2 | 单位矩阵 | 179 |
| 7.2.2 | 使用 CharacterData | 126 | 8.5.3 | 矩阵乘法和矩阵与向量的乘法 | 181 |
| 7.3 | 渲染文本 | 129 | 8.5.4 | 平移和缩放 | 182 |
| 7.3.1 | 计算 FPS | 130 | 8.5.5 | 旋转 | 183 |
| 7.3.2 | 垂直同步和帧率 | 132 | 8.5.6 | 求逆矩阵 | 184 |
| 7.3.3 | 性能分析 | 133 | 8.5.7 | 对精灵执行矩阵操作 | 185 |
| 7.4 | 优化 Text 类 | 133 | 8.5.8 | 修改精灵来使用矩阵 | 187 |
| 7.5 | 使用 glDrawArrays 进行快速渲染 | 138 | 8.5.9 | 优化 | 189 |
| 7.5.1 | 修改渲染器 | 140 | | | |

| | | | |
|--|-----|-----------------------------|-----|
| 第 9 章 创建游戏引擎 | 191 | 第 11 章 创建自己的游戏 | 323 |
| 9.1 新的游戏引擎项目 | 191 | 11.1 项目管理 | 323 |
| 9.2 扩展游戏引擎 | 194 | 11.2 显示方法 | 325 |
| 9.2.1 在项目中使用游戏引擎 | 194 | 11.2.1 2D 游戏 | 325 |
| 9.2.2 多个纹理 | 202 | 11.2.2 3D 游戏 | 325 |
| 9.3 添加声音支持 | 205 | 11.3 游戏类型 | 328 |
| 9.3.1 创建声音文件 | 205 | 11.3.1 文字类游戏 | 328 |
| 9.3.2 开发 SoundManager | 206 | 11.3.2 益智游戏 | 330 |
| 9.4 改进输入 | 215 | 11.3.3 第一人称射击游戏 | 332 |
| 9.4.1 包装游戏控制器 | 215 | 11.3.4 策略游戏 | 333 |
| 9.4.2 添加更好的鼠标支持 | 229 | 11.3.5 角色扮演游戏 | 334 |
| 9.4.3 添加键盘支持 | 236 | 11.3.6 平台游戏 | 339 |
| 第 10 章 创建一个简单的卷轴射击 游戏 | 241 | 11.4 结束语 | 341 |
| 10.1 一个简单的游戏 | 241 | 附录 A 推荐阅读材料 | 343 |
| 10.2 第一遍实现 | 242 | | |
| 10.2.1 开始菜单的状态 | 246 | | |
| 10.2.2 游戏主体状态 | 257 | | |
| 10.2.3 游戏结束状态 | 260 | | |
| 10.3 开发游戏主体 | 263 | | |
| 10.3.1 移动玩家角色 | 263 | | |
| 10.3.2 使用滚动背景模拟移动 | 268 | | |
| 10.3.3 添加一些简单的敌人 | 271 | | |
| 10.3.4 添加简单的武器 | 279 | | |
| 10.3.5 伤害和爆炸 | 288 | | |
| 10.3.6 管理爆炸和敌人 | 295 | | |
| 10.3.7 定义关卡 | 301 | | |
| 10.3.8 敌人的移动 | 304 | | |
| 10.3.9 敌人攻击 | 315 | | |
| 10.4 继续迭代 | 319 | | |

第 I 部分

背景知识

第 1 章 C#的历史

第 2 章 OpenGL 简介

第 3 章 现代方法

第 1 章

C#的历史

C#是一种面向对象的现代语言，由 Anders Hejlsberg 带领的 Microsoft 团队开发。公共语言运行时(Common Language Runtime, CLR)是运行 C#的虚拟机。许多语言都运行在 CLR 上,这意味着在 Windows PC、Xbox 360 和 Zune 上都可以编译和使用它们。CLR 由 Microsoft 开发并拥有,但是它也有一个开源版本,叫做 Mono。通过 Mono, C#程序可以运行在 Mac、Linux, 或者其他能够编译 Mono 的系统上。

游戏业中一般使用C++,但是这并不代表程序员想这么做。C++是一种庞大的语言,有许多地方会让粗心的程序员掉入陷阱,也有许多根本不曾定义的地方。C#比 C++友好得多,用来编写游戏的话,也会使程序员的工作有趣得多。

1.1 C#基础

2000 年 7 月在 Orlando 举行的 Microsoft Professional Developers Conference 会议上第一次公布了 C#。在当时, C#还有点像 Microsoft 版本的 Java,但是很快它就被开发成了一种具有自己显著特点的语言。C#最近的更新颇具创新性,让开发人员倍感兴奋。

C#和 Java 等现代语言编写的程序运行在虚拟机上。而由 C++和 C 等早一些的语言编写的程序要直接运行在特定计算机的硬件上。运行程序的硬件叫做中央处理单元(Central Processing Unit, CPU),这是计算机的“大脑”。现代 Mac 和 PC 一般都采用 x86 CPU, Xbox 360 采用 Xenon CPU, PS3 则采用 Cell CPU。这些 CPU 彼此之间都存在一些差异,但是它们的基本用途是相同的:运行程序员编写的程序。程序就是一条条的指令。特定 CPU 能够理解的指令叫做机器码。一个 CPU 不太可能理解其他类型的 CPU 的机器码。

编译器把使用 C++、C 或其他语言编写的、人类可读的源代码编译成机器码。例如,

4 第 I 部分 背景知识

PlayStation 3 的 C++ 编译器将源代码编译成 Cell CPU 可以运行的机器码。要在 Xbox 360 的 Xenon CPU 上运行相同的 C++ 代码，必须使用 Xbox 360 的 C++ 编译器重新进行编译。C# 的工作方式与此不同。C# 编译得到的汇编语言叫做公共中间语言(Common Intermediate Language, CIL)。CIL 运行在一个虚拟机上，以生成特定系统的机器码。在 PlayStation 3 中，这意味着要运行 C# 代码，需要有一个把 CIL 代码转变为 Cell CPU 可以理解的机器码的虚拟机。图 1-1 显示了使用虚拟机的语言和直接编译为机器码的语言之间的这种区别。

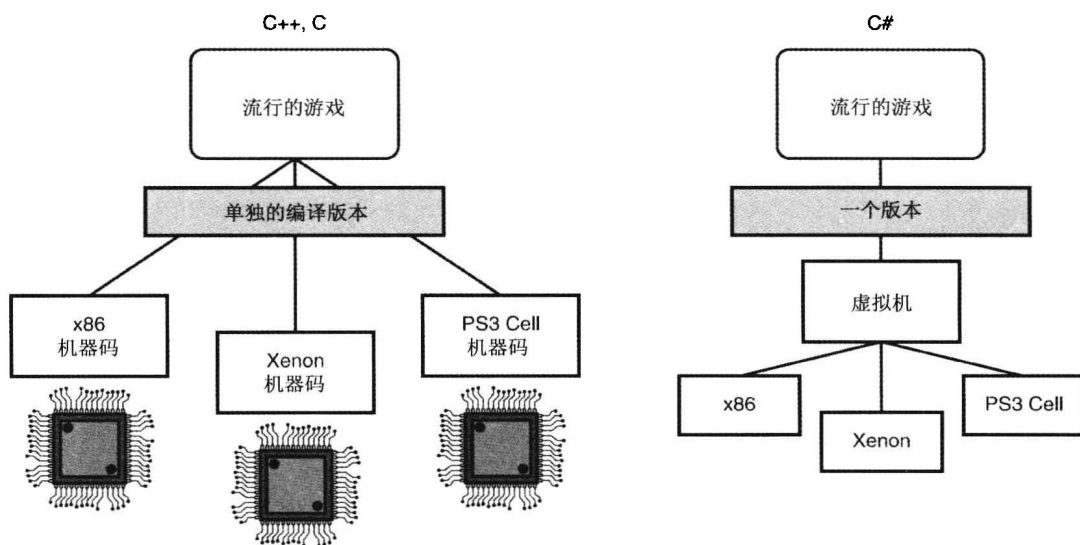


图 1-1 虚拟机和直接编译的代码

C# 的虚拟机 CLR 有许多优点。最显著的就是，只需要编写代码一次，而后代码就可以在每个支持 CLR 的 CPU 上运行。使用 C# 时，系统不太容易崩溃，因为虚拟机把系统隔离开了。在错误被发送到硬件之前，虚拟机就会捕获它们。内存的分配和释放也会自动完成。只要程序语言会被编译成 CLR，就可以混用它们，诸如 F#、IronLisp、IronRuby、IronPython 和 IronScheme 等语言可以用在一个程序中。游戏程序员可以使用高级的、适合编写 AI 的语言来编写 AI，而使用与 C 类似的过程语言编写图形处理代码。

C# 的版本

C# 定期更新功能和特性。即使程序员以前使用 C# 编写过程序，也可能会不知道 C# 在最近的更新中添加的所有新特性。

1. C# 2.0

C# 2.0 添加了泛型和匿名委托。使用示例最容易解释这些新特性。

泛型

泛型是编写代码来使用具有通用属性的通用对象，而不是使用具体对象的一种方法。可以对类、方法、接口和结构使用泛型，以定义它们使用的数据类型。泛型代码使用的数

据类型在编译时指定,这意味着程序员不需要编写代码来检查是否使用了正确的数据类型。这样做的好处是,由于不需要编写测试代码,代码更加简洁,而且由于减少了在运行时发生的类型不匹配错误,代码也更加安全。泛型代码的运行速度一般较快,因为不需要执行很多数据类型的强制转换。

下面的代码没有使用泛型:

```
ArrayList _list = new ArrayList();
_list.Add(1.3); // boxing converts value type to a reference type
_list.Add(1.0);

//Unboxing Converts reference type to a value type.
object objectAtPositionZero = _list[0];
double valueOne = (double) objectAtPositionZero;
double valueTwo = (double) _list[1];
```

这段代码创建了一个列表,并添加了两个十进制数字。C#将每个事物都看作对象,数字也不例外。C#中的全部对象都从 `object` 类继承。对象存储在 `ArrayList` 中。上面的代码添加了数字,但是 `ArrayList` 不知道它们是数字。对于 `ArrayList` 来说,它们就是对象, `ArrayList` 只能识别对象。把数字传递给 `ArrayList.Add` 时,它们被转换为 `object` 类型,然后添加到 `ArrayList` 的对象集合中。

每次从列表中取出对象时,必须将它从对象重新转换为原来的类型。类型转换的效率很低,而且使代码变得杂乱而难以阅读,所以非泛型代码很难处理。泛型代码可以解决这种问题。为了理解它们如何解决问题,必须理解引用类型和值类型的区别。

C#有两类宽泛的类型:引用类型和值类型。值类型是基本类型,例如 `int`、`float`、`double`、`bool`、`string` 等。引用类型是程序员使用关键字 `class` 定义的大型数据结构,例如 `Player`、`Creature`、`Spaceship` 等。使用关键字 `struct` 也可以自定义值类型。

```
// An example reference type
public class SpaceShip
{
    string _name;
    int _thrust;
}
// An example value type
public struct Point
{
    int _x;
    int _y;
}
```

值类型直接把数据存储到内存中。引用类型间接地把数据存储到内存中。在内存中,引用类型的地址是另外一个内存地址,数据实际存储在这个地址。初看起来,这可能有点奇怪,但是熟悉了这些概念后就会感到这其实很简单。两种类型在内存中的区别如图 1-2 所示。

6 第 I 部分 背景知识

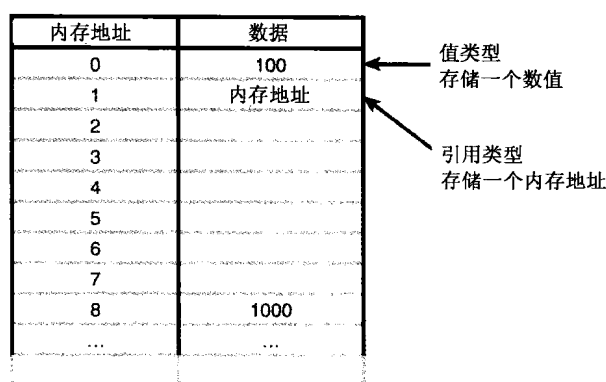


图 1-2 内存中的值类型和引用类型

如果把计算机的内存视为一个巨大的图书馆，其中都是书架，创建一个值类型就是在书架上加入几本书。创建引用类型则更像是在书架上放入一张纸条，指出书存放在图书馆的哪个地方。

下面是值类型的一个例子。

```
int i = 100;
```

如果在内存中查看 *i* 的存储位置，就会得到数字 100。在源代码中我们使用了变量名 *i*，在汇编代码中 *i* 是内存地址。所以这行代码将值 100 放到下一个可用的内存地址，并把该内存地址叫做 *i*。

```
int i = 100;
int j = i; // value type so copied into i
Console.WriteLine(i); // 100
Console.WriteLine(j); // 100
i = 30;
Console.WriteLine(i); // 30
Console.WriteLine(j); // 100
```

在这段代码中，值类型 *i* 被赋给 *j*，意味着它的数据将被复制。内存地址 *j* 中存储的数据将与 *i* 存储的数据完全相同。

下一个示例与此示例类似，但是使用了引用类型。首先，需要通过定义新类来创建一个引用类型。这个类的作用只是存储数字，就像是我们自己创建的 `int` 类型。这个类型将被命名为 `Int`，使用大写 `I` 来与 C# 内置的 `int` 类型进行区分。

```
class Int()
{
    int _value;
    public Int(int value)
    {
        _value = value;
    }
}
```

```
public void SetValue(int value)
{
    _value = value;
}
public override string ToString()
{
    return _value.ToString();
}
}

Int i = new Int(100);
Int j = i; // reference type so reference copied into i
Console.WriteLine(i); // 100
Console.WriteLine(j); // 100
i.SetValue(30);
Console.WriteLine(i); // 30
Console.WriteLine(j); // 30 <- the shocking difference
```

在这里，当 *i* 改变时，*j* 也会改变。这是因为 *j* 本身存储没有数据的副本，它只是具有与 *i* 相同的数据引用。当底层数据改变时，无论使用 *i* 还是使用 *j* 访问数据，总是会返回相同的结果。

再次以图书馆为例。在图书馆的索引系统中查找想要的图书，得到了两个书架位置。第一个位置 *i* 没有存放书，而只有一张纸条。检查位置 *j* 时，发现了类似的一张纸条。这两张纸条都指向了另外一个位置。无论采纳哪张纸条的指示，都可以得到要寻找的图书。书只有一本，但是引用却有两个。

装箱(boxing)是将值类型转换为引用类型的过程。取消装箱(unboxing)是将新的引用类型转换回原来的值类型的过程。在 C# 的第 1 版中，即使对象列表中只包含值类型，在把这些对象添加到列表中时也需要将它们转换为引用类型。引入泛型后，情况发生了变化。下面的示例代码使用了泛型功能，比较它们与前面使用 `ArrayList` 的示例之间的区别。

```
List<float> _list = new List<float>();
_list.Add(1);
_list.Add(5.6f);
float valueOne = _list[0];
float valueTwo = _list[1];
```

这段示例代码创建了一个由 `float` 类型的成员组成的列表。`float` 是值类型。在这里不需要像前面那样通过强制转换对 `float` 执行装箱和取消装箱操作，代码直接就可以完成工作。泛型列表知道自己存储的类型，所以不需要将值类型转换为对象类型。由于不需要总是在类型之间进行转换，所以得到的代码更加高效。

泛型对于创建可重用的数据结构(例如列表)十分有用。

匿名委托

在 C# 中，委托是一种把一个函数作为另一个函数的参数进行传递的方法。当需要对许

8 第 I 部分 背景知识

多不同的列表采取重复性的操作时，这种特性就非常实用。下面用一个函数迭代列表，用另一个函数对列表的每个成员执行操作。示例如下：

```
void DoToList(List<Item> list, SomeDelegate action)
{
    foreach(Item item in list)
    {
        action(item);
    }
}
```

匿名委托允许在 DoToList 函数调用中直接写函数，而不需要声明或预定义该函数。由于没有预定义，该函数没有名称，所以叫做匿名函数。

```
// A quick way to destroy a list of items
DoToList(_itemList, delegate(Item item) { item.Destroy(); });
```

这个示例演示了如何迭代列表，并对每个成员应用匿名函数。示例中的匿名函数调用每个成员的 Destroy 方法。这的确很方便。匿名函数还有另外一个技巧。观察下面的代码：

```
int SumArrayOfValues(int[] array)
{
    int sum = 0;
    Array.ForEach(
        array,
        delegate(int value)
        {
            sum += value;
        }
    );
    return sum;
}
```

这被叫做闭包。它为每次循环迭代创建一个委托，并使用相同的变量 *sum*。变量 *sum* 甚至可以超出作用域，但是匿名委托仍然将保持对它的引用。它将成为一个只能被封闭它的函数访问的私有变量。闭包在函数式程序语言中有着大量应用。

2. C# 3.0

C# 3.0 不满足于做少量改进，所以引入了极具创新性的新功能，可以显著减少样板代码的数量。最大的变化就是引入了 LINQ 系统。

LINQ

LINQ 是 Language-Integrated Query(语言集成查询)的缩写。它与处理数据结构的 SQL(结构化查询语言，一种用于从数据库中检索并操作数据的语言)有些类似。下面通过示例简单地了解一下 LINQ 的用法。

[illegible]

10 第 I 部分 背景知识

```

        orderby m.Name() descending
        select m;

foreach (Monster m in query)
{
    m.LevelUp();
}

```

运行这个查询时，它检索所有可以升级的怪物，然后按名称排序该列表。LINQ 不只用于处理集合，也可以用于处理 XML 和数据库。LINQ 还可以处理 C# 3.0 中新增的另外一个特性：lambda 函数。

lambda 函数

还记得匿名委托吗？lambda 函数是编写匿名委托的一种更方便的方法。下面这个示例删除了玩家物品栏中的全部物品：

```
_itemList.ForEach(delegate(Item x) { x.Destroy(); });
```

如果使用 lambda 函数，写法更加简洁：

```
_itemList.ForEach(x => x.Destroy());
```

较少的样板代码也使得代码更加易读。

对象初始化器

对象初始化器使对象的构造更加灵活，使构造函数不再那么冗长。下面这个示例没有使用对象初始化器。

```
Spaceship s = new Spaceship();
s.Health = 30;
```

使用对象初始化器，写法更加简洁：

```
Spaceship s = new Spaceship { Health = 30 };
```

一些 LINQ 查询中采取了这种做法。使用对象初始化器后，就没有必要编写多个只设置少量字段的构造函数了。

集合初始化器

使用集合初始化器设置列表是一件很轻松的工作。同样，为了展示改进方法的优点，首先给出原来的方式：

```

class Orc
{
    List<Item> _items = new List<Item>();

    public Orc()
    {
        _items.Add(new Item('axe'));
    }
}

```

```
        _items.Add(new Item('pet_hamster'));
        _items.Add(new Item('skull_helm'));
    }
}
```

以这种方式初始化列表需要将一定数量的设置代码放到构造函数或者单独的初始化方法中。而使用集合初始化器时，代码更加简洁：

```
class Orc
{
    List<Item> _items = new List<Item>
    {
        new Item('axe'),
        new Item('pet_hamster'),
        new Item('skull_helm')
    };
}
```

在本例中，根本没有用到构造函数，创建对象时会自动完成对它的初始设置。

隐含类型局部变量和匿名类型

如果在代码中需要同时初始化多个泛型对象，那么处理起来会有些麻烦。类型推断在此时可以提供帮助。

```
List<List<Orc>> _orcSquads = new List<List<Orc>>();
```

在这段代码中，我们需要重复输入相同的内容，也就是说，如果代码发生改变，就需要修改大量的代码。隐含类型局部变量可以帮助我们摆脱这种困境。

```
var _orcSquads = new List<List<Orc>>();
```

当想要让编译器自己确定正确的类型时，需要使用关键字 **var**。在我们的示例中这很有用，因为需要维护的代码量变少了。关键字 **var** 也用于表示匿名类型。如果在创建一个类型时没有为其指定预定义的类或结构，该类型就是匿名类型。创建匿名类型的语法与集合初始化器的语法有些类似。

```
var fullName = new { FirstName = 'John', LastName = 'Doe' };
```

现在可以像使用引用类型一样使用 **fullName** 对象。它有两个只读字段，分别是 **FirstName** 和 **LastName**。当只有一个函数，但是想要返回两个值时，可以把两个值放到匿名类型中返回。匿名类型更常见的用法是存储从 LINQ 查询返回的数据。

3. C# 4.0

C# 4.0 是最新的 C# 版本，其中最主要的改进是增添了动态类型。C# 是一种静态类型语言。静态类型是指各个对象的类型在编译时就已经确定。在定义类时会为其定义字段和方法，这些字段和方法在整个程序中不会改变。而在动态类型中，对象和类更加灵活——函

12 第 I 部分 背景知识

数和字段可以被添加和删除。**IronPython** 和 **IronRuby** 都是运行在 CLR 之上的动态语言。

动态对象

动态对象使用一个新的关键字 **dynamic**。下面的示例展示了其用法。假设有几个不相关的类同时实现了一个叫做 **GetName** 的方法。

```
void PrintName(dynamic obj)
{
    System.Console.WriteLine(obj.GetName());
}
PrintName(new User('Bob')) // Bob
PrintName(new Monster('Axeface')) // Axeface
```

当调用函数时，如果对象是动态的，则将在运行时查找该函数。这叫做鸭子类型(duck typing)。该查找操作通过反射完成，或者如果对象实现了接口 **IDynamicObject**，它将调用方法 **GetMetaObject** 在内部处理查找操作。鸭子类型的处理速度相对较慢，因为它需要时间来原因确定函数是否存在。传统的 C# 一般在编译时确定一下函数或者字段是否存在。

下面的代码在 C# 4.0 下可以编译通过。

```
public dynamic GetDynamicObject()
{
    return new object();
}
dynamic testObj = GetDynamicObject();
testObj.ICanCallAnyFunctionILike();
```

方法 **ICanCallAnyFunctionILike** 可能并不存在，所以当程序执行到这一部分代码时，将会抛出一个异常。

要调用动态对象的任何方法或者访问其任意成员，会要求该对象在运行时进行查找，以确认该成员或者方法确实存在。这意味着如果代码中存在拼写错误，编译过程中并不能捕获该错误，只有运行代码时才可以。在 C# 中，如果没有找到动态对象的方法或字段，将抛出 **RuntimeBinderException** 类型的异常。

如果游戏中需要使用大量动态脚本，或者如果想要像在 **PrintName** 示例中那样利用鸭子类型，**dynamic** 关键字十分有用。由于每次调用 **GetName** 方法时都必须查找该函数，所以 **PrintName** 的速度较慢。在某些情况中这种速度上的损失可能无关紧要，但最好还是留意这类问题。

可选参数和命名参数

C# 4.0 开始支持在方法中使用可选参数。可选参数是 C++ 的一个标准特性，其工作方式如下：

```
class Player
{
    // Knock the player backwards
    public void KnockBack(int knockBackDamage = 0)
```



```

    {
        // code
    }
}
Player p = new Player();
p.KnockBack(); // knocks the player back and by default deals no damage
p.KnockBack(10); // knocks the player back and gives 10 damage

```

可选参数允许程序员为方法的参数指定默认值。命名参数允许以一种优雅的方式使用多个可选参数。

```

enum HairColor
{
    Blonde,
    Brunette,
    Red,
    Black
}
enum Sex
{
    Male,
    Female
}
class Baby
{
    public string Name { get; set; }
    public Sex Sex { get; set; }
    public HairColor HairColor { get; set; }
}
class Player
{
    public Baby HaveBaby(string babyName, Sex sex = Sex.Female, HairColor
hairColor = HairColor.Black)
    {
        return new Baby()
        {
            Name = babyName,
            Sex = sex,
            HairColor = hairColor
        };
    }
}
Player player = new Player();
player.HaveBaby('Bob', Sex.Male);
player.HaveBaby('Alice');

```

在这个示例中，**Player** 类中有允许玩家生孩子的代码。**HaveBaby** 方法接受一个名字和

14 第 I 部分 背景知识

其他一些可选参数，然后返回一个新的 **Baby** 对象。**HaveBaby** 方法要求必须提供一个名字，但是 **sex** 和 **hairColor** 参数是可选的。下一个示例显示了如何设置头发颜色，其中 **sex** 参数采用了默认值 **female**。

```
player.HaveBaby('Jane', hairColor: HairColor.Blonde)
```

命名参数使程序员可以只设置希望设置的参数。

1.2 小结

C#是一门现代的、具有垃圾回收功能的面向对象编程语言，自从 2000 年 6 月在 Orlando 举办的 Microsoft Professional Developers Conference 第一次把它公之于众，C#已经有了长足的发展。C#运行在一个叫做公共语言运行时(CLR)的虚拟机上。CLR 可以运行在同一个程序中混合使用的多种不同的语言。

C#是一门不断发展的语言，随着时间的推移，仍在不断添加更多的功能和特性。C#有 4 个主版本，每个版本中都引入了新的、实用的功能。这些功能通常使程序更易于编写和更加高效，所以熟悉改进的地方十分重要。C#的最新版本是 C# 4.0，于 2010 年 4 月发布。

第 2 章

OpenGL 简介

每台计算机都有专门处理图形的硬件，它们控制着屏幕上显示的内容。OpenGL 向这种硬件发出命令，告诉它们执行什么操作。计算机游戏或者其他任意软件借助制造商提供的设备驱动程序，使用 OpenGL 向图形硬件发出命令，如图 2-1 所示。

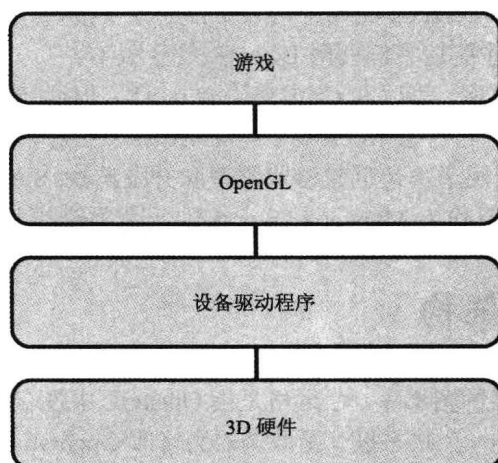


图 2-1 OpenGL 的典型应用

OpenGL(Open Graphics Library, 开放图形库)是游戏开发商使用最早、最流行的图形库之一。OpenGL 是 Silicon Graphics 公司(SGI)在 1992 年开发的，但是直到 1997 的 GLQuake 中采用了这种图形库以后，游戏开发商才真正对它产生了兴趣。GameCube、Wii、PlayStation 3 和 iPhone 都把 OpenGL 作为它们的图形库的基础。

除了 OpenGL 之外，还有一个选择是 Microsoft 的 DirectX。DirectX 由更多的库组成，

包括声音和输入，所以把 OpenGL 与 DirectX 中的 Direct3D 库进行比较更加合适。DirectX 的最新版本是 DirectX 11。Xbox 360 使用的是 DirectX 9.0。DirectX 10 和 DirectX 11 只能在安装了 Windows Vista 或 Windows 7 操作系统的计算机上使用。

Direct3D 和 OpenGL 的功能集基本相同。现在游戏引擎(例如 Unreal)通常构建一个抽象层，允许用户根据情况在 OpenGL 和 Direct3D 之间切换，如图 2-2 所示。当开发跨平台的游戏，例如需要把游戏发布到 PlayStation 3 和 Xbox 360 上时，必须提供这种抽象。Xbox 360 必须使用 Direct3D 调用，而 PS3 必须使用 OpenGL 调用。

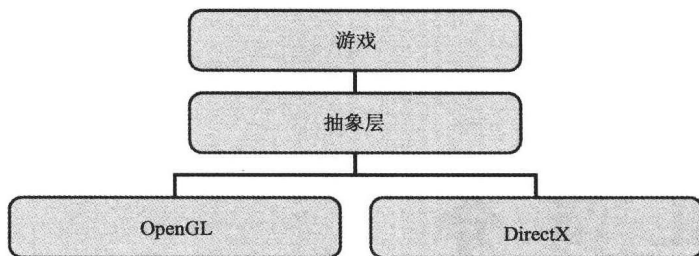


图 2-2 使用抽象层

DirectX 和 OpenGL 都是非常优秀的图形库。DirectX 工作在 Microsoft 平台上，而 OpenGL 可以应用到更加广泛的平台上。DirectX 更新得更快，意味着它可以利用最新的图形功能。OpenGL 的更新较慢，最新的图形功能只能通过一个不太方便的扩展机制使用。OpenGL 这种缓慢的更新有一个明显的好处，即接口很少发生改变，所以多年前编写的代码仍然可以使用最新的 OpenGL 版本。DirectX 的每个新版本都会改变接口，所以无法维持兼容性，必须调整和修改较老的代码才能使它们利用 DirectX 的新版本。

通过 Managed DirectX 库，可以在 C# 中使用 DirectX，但遗憾的是，官方不再支持或更新这些库。Managed DirectX 已经被 Microsoft 的 XNA 游戏创建库替代。XNA 使用 DirectX，但它是一个更高层的框架，用于快速创建游戏原型和开发游戏。SlimDX 是一个针对 DirectX 开发的独立的 C# API，可以作为 Managed Directx 的一个不错的替代品。

2.1 OpenGL 的架构

OpenGL 是一个 C 风格的图形库。C 风格是指 OpenGL 中没有类或对象，而只有大量的函数。OpenGL 在内部就是一个状态机。函数调用会修改 OpenGL 的内部状态，这又会影响 OpenGL 的行为和在屏幕上渲染多边形的方式。OpenGL 是状态机这个事实会带来一些问题，例如在代码中无意间修改了其他某个部分的状态，从而导致出现 bug。因此，留意哪些状态发生改变十分重要。

2.1.1 顶点：3D 图形的基础

OpenGL 中的基本单元是顶点。简单来说，顶点就是空间中的一个点。在这些点上可以附加其他信息，如它如何映射到纹理，是否具有一定的重量或颜色等，但是最重要的信息

是它的位置。

游戏会把大量的时间用于发送 OpenGL 顶点，或者告诉 OpenGL 以特定的方式移动顶点等。游戏可能首先告诉 OpenGL 它将要发送的全部顶点会构成三角形。此时，OpenGL 将使用线把它收到的每三个顶点连接起来以创建一个多边形，然后可能还会使用纹理或颜色填充多边形的表面。

现代图形硬件非常擅长处理大量的顶点，把它们构成多边形，然后渲染到屏幕上。这个从顶点到屏幕的过程叫做流水线(pipeline)。流水线负责定位顶点和提供光照，以及进行投影变换。这个过程将把收到的 3D 数据变换为 2D 数据，以便将其显示到屏幕上。投影变换听起来有点复杂，但是几个世纪以来，画家和艺术家们一直在利用这种变换，在平展的画布上绘制出他们周围的世界。

就算现代的 2D 游戏也是使用顶点创建的。2D 精灵(sprite)由两个三角形构成，这两个三角形形成了一个四边形(quad)。在给四边形提供纹理后，它就成为了精灵。2D 游戏使用一种特殊的投影变换，忽略了顶点中的所有 3D 数据，因为 2D 游戏中并不需要这些数据。

2.1.2 流水线

流水线的流程如图 2-3 所示。这个流水线叫做固定功能流水线(fixed function pipeline)。当程序员为这种流水线提供输入后，所有的功能已被固定，无法修改。例如，在像素处理阶段很容易向屏幕上添加蓝色调，但是在固定功能流水线中，程序员无法直接控制这个阶段。

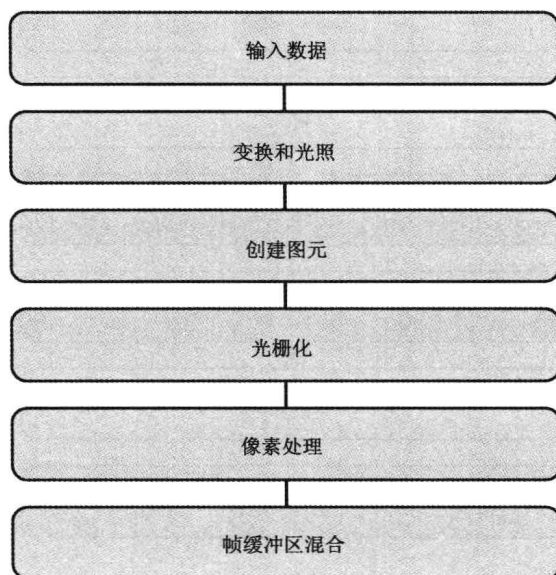


图 2-3 基本的固定功能流水线

固定功能流水线包含 6 个主要的阶段。这些阶段几乎都可以并行应用。如果硬件支持的话，每个顶点可以同时经过同一个阶段。由于这个原因，图形卡的处理速度比 CPU 的处理速度要快得多。

- 输入阶段。输入以顶点的形式给出，顶点的属性包括位置、颜色和纹理数据。
- 变换和光照。根据当前视图变换顶点。这包括将顶点的位置由 3D 空间(也叫做世界空间)改为 2D 空间(也叫做屏幕空间)。如果有些顶点对最终的显示没有影响，可以在这个阶段移除它们。在这个阶段也可以对每个顶点执行光照计算。
- 创建图元。这是使用顶点信息创建多边形的过程，需要根据 OpenGL 的状态把顶点连接到一起。游戏经常使用三角形或者三角带作为图元。
- 光栅化。这是将多边形转换为像素(也叫做片段)的过程。
- 像素处理。像素处理也叫做片段处理。这个阶段将测试像素，以确定是否把它们绘制到帧缓冲区中。
- 帧缓冲区。帧缓冲区是一块内存，代表特定的帧将显示在屏幕上的部分。混合设置将决定像素如何与已经绘制的部分进行混合。

在过去的几年中，已经可以使用程序控制流水线了。将程序上传到图形卡以后，它们将替换固定功能流水线的默认阶段。重写流水线的特定部分的小程序叫做着色器(shader)。

图 2-4 显示了可编程流水线的阶段。这里删除了变换和光照阶段，添加了顶点和几何着色器。像素处理阶段也可以像像素着色器那样进行编程。通过流水线的每个像素都会被应用像素着色器。像素着色器不能添加顶点，但是可以修改位置、颜色和纹理位置等属性。

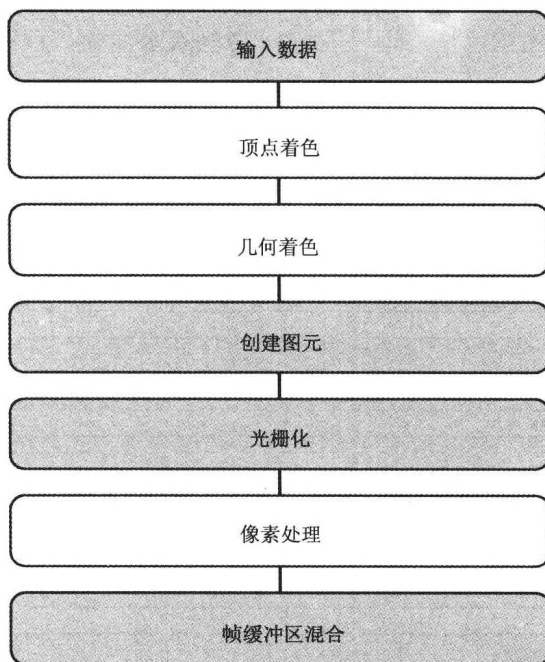


图 2-4

与像素着色器和顶点着色器相比，几何着色器加入的时间较晚。几何着色器以一个完整的图元(如线、点或三角形组成的带)作为输入，而顶点着色器针对每个图元都会运行。几何着色器可以创建新的顶点信息、点、线，甚至图元。几何着色器还可用于创建点精灵和动态镶嵌，以及实现其他一些效果。点精灵可以快速渲染大量精灵，这种技术经常用于

粒子系统，可创建类似于火焰或者烟雾效果。动态镶嵌可以在几何形状中添加更多的多边形。这种技术可以用来增加包含多边形数较少的游戏模型的平滑度，并在摄像机放大模型时展现更多的细节。

像素着色器会应用到被发送到帧缓冲区的每个像素。像素着色器用于创建凹凸贴图效果、镜面高光和逐像素光照。凹凸贴图效果可以给表面增加额外的高度信息。凹凸贴图通常包含一个图片，其中每个像素代表一个标准的法向量，该法向量描述了如何扭曲表面。像素着色器可以使用凹凸贴图来为模型提供一个更有趣的纹理，使其看上去有一定的深度感。逐像素光照用于替换 OpenGL 的默认的光照公式。OpenGL 的光照公式可计算出每个顶点的光照，然后为该顶点提供一种合适的颜色。逐像素光照使用更精确的光照模型，为每个像素单独应用光照。镜面高光是模型中非常亮、反射大量光的区域。

2.2 变化中的 OpenGL

现在是学习 OpenGL 的大好时机。OpenGL 的当前版本非常适合学习。这个版本中包含了大量易于使用的函数，可以完成各种各样与图形处理有关的工作。可以把 OpenGL 的当前版本想象成一个带有辅助轮的自行车，刚开始时借助它们学习怎么骑车，熟练之后就把它取下来。OpenGL 的下一个版本更像是一个性能强大的摩托车，没有必要的东西都被移除了，剩下的就是不受限制的原始力量。对于有经验的 OpenGL 程序员，其好处自不必言，但是初学者却容易对其望而生畏。

可以使用的 OpenGL 版本要取决于系统中安装的图形卡驱动程序。几乎每台计算机都支持 OpenGL 2.1，而近期生产的图形卡会支持 OpenGL 3.x。

2.2.1 OpenGL ES

OpenGL ES 是用于嵌入式系统的 OpenGL 的新版本。它与 OpenGL 的近期版本相似，但是功能集更加受限。它用于高端手机，例如 Android、BlackBerry 和 iPhone。OpenGL ES 也用于军用硬件，例如战机上的平视显示器。

OpenGL ES 支持可编程流水线，也支持着色器。

2.2.2 WebGL

WebGL 目前仍处于开发中，这个版本的 OpenGL 是专为在 Web 上使用而设计的。使用 WebGL 的浏览器必须支持 HTML 5 canvas 标签。就现在而言，还无法知道它会取得多大的成功。以前人们曾有过在 Web 上使用 3D 的尝试，但是都失败了。例如，VRML(Virtual Reality Modeling Language, 虚拟现实建模语言)与 HTML 类似，但是允许用户创建 3D 世界，它在学术界受到了不少关注，却从没有吸引过普通的用户。

WebGL 有一些强有力的支持者，许多大公司都加入了 WebGL 工作组，如 Google、Apple 和 Mozilla，而且 WebGL 还有一些让人印象非常深刻的演示。就目前来看，WebGL 的性能还是不错的，在成熟以后很可能可以与 Flash 一较高低。

2.3 OpenGL 和图形卡

OpenGL 是一个允许程序员发送指令到图形卡的库。图形卡是一种专用于显示 3D 数据的硬件,由很多标准组件构成,包括帧缓冲区、纹理内存和 GPU。GPU 是图形处理单元(Graphics Processing Unit)的缩写,它控制着如何处理顶点并把它们显示到屏幕上。CPU 向 GPU 发送指令和数据,描述每一帧应该怎样显示到屏幕上。纹理内存通常是一块较大的内存,用于存储游戏所需的大量纹理。帧缓冲区是内存中的一块区域,存储下一帧中将显示到屏幕上的图像。现代的图形卡通常有多个 GPU,每个 GPU 上都有许多着色器处理单元来执行大规模的并行着色器操作。分布式应用程序(如模拟蛋白质折叠的 Folding@home)和世界各地的数十万台计算机都利用了这一特点。

第一个获得流行的 3D 图形卡是 3dfx Voodoo 1。这是早期的一个图形卡,有 2MB 的纹理内存和 2MB 的帧缓冲区,并且使用 PCI 总线,时钟速度为 135MHz。早期的一些游戏使用它来加速执行,例如《古墓丽影(Tomb Raider)》、《Descent II》、《雷神之锤(Quake)》以及《雷神之锤 2(Quake 2)》的演示,从而运行得更加流畅,并且可显示更多的细节。Voodoo 1 使用一个标准的 PCI 总线,允许 CPU 以最高 533MB/s 的速度向图形卡发送数据。现代的图形卡已经从 PCI 转向使用 AGP(Accelerated Graphics Port,加速图形端口),其最高数据发送速度为 2GB/s,后来又转向使用 PCI Express。当前的这一代 PCI Express 卡的最高数据发送速度为 8GB/s。

好像每个月都会有新的图形卡问世,每个新图形卡都比之前的图形卡更快。在编写本书时,最快的图形卡可能是 ATI Radeon HD 5970。它有两个 GPU,每个 GPU 都有 1600 个着色器处理器。它的时钟速度为 725MHz,每秒可以处理 4.64 万亿次浮点运算。

大多数现代图形卡都有专门执行新操作的特殊硬件。这种硬件通过使用扩展提供给 OpenGL。当收到一个标识新扩展的字符串时,OpenGL 能够展示驱动程序和图形卡中的功能。例如,ATI Radeon HD 5970 有两个 GPU,这种情况很少见。为了能够充分利用两个 GPU,需要使用一些新的扩展方法,如 AMD_gpu_association。这个扩展允许用户在两个 GPU 之间分配任务。如果多家供应商实现了相同的扩展,那么扩展字符串的某个位置会有字母 EXT。有时候,控制 OpenGL 规范的架构评审委员会可能会把某个扩展的状态提升为官方扩展,此时,扩展字符串中将包含字母 ARB,所有的供应商都必须支持该扩展。

着色器——图形卡上的程序

着色器(shader)这个词带有一定的误导性。最初的着色器程序主要用于处理组成每个多边形表面的像素,以便为模型着色。但是随着时间推移,着色器程序的功能已经被扩展,现在还可以修改顶点属性、创建新顶点,甚至完成一般的操作。

着色器与运行在 CPU 上的普通程序具有不同的工作方式。着色器程序在大量的元件上同时执行,这意味着着色器程序是大规模并行程序,而运行在 CPU 上的程序一般则是串行运行的,一次只有一个实例运行。着色器程序非常适合对构成 3D 世界的像素和顶点的集合执行操作。

目前共有 3 类着色器,分别是顶点着色器、几何着色器和像素着色器,每种着色器都只能执行特定的操作。顶点着色器处理顶点,像素着色器处理像素,几何着色器处理图元。

为了降低复杂性，并使硬件制造商可以更高效地进行优化，所有这些着色器都被一种叫统一着色器(unified shader)的着色器替代了。

着色器通过运行在图形卡上的特殊语言进行编程。目前，这些语言比 C++ 低级得多(更别提 C# 了)。OpenGL 有一门叫做 GLSL(OpenGL Shading Language, OpenGL 着色语言)的着色语言，它与 C 语言有些类似，但是有大量用于处理向量和矩阵的特殊操作。DirectX 也有自己的着色语言，叫做 HLSL(High Level Shading Language, 高级着色语言)。两种语言十分类似。让人更加困惑的是，除了这两种语言以外，还有一种叫做 Cg 的语言，它由 Microsoft 和 Nvidia 开发，与 HLSL 有些类似。

游戏中的着色器非常适合创建需要进行大量计算的特效，如视差贴图的光照。当前的技术使得着色器程序几乎不能用于其他用途。本书主要介绍游戏编程，由于可编程流水线是一个很大的主题，所以不会讨论该技术。如果对此主题感兴趣，可以参阅附录 A 部分，那里介绍了几本非常好的书籍。

着色器的一种趋势是用于一般性的并行编程任务，而不只是处理图形。例如，Nvidia PhysX 库允许在 GPU 而非 CPU 上完成物理计算，从而得到更佳的性能。PhysX 是用另外一种叫做 CUDA 的着色器语言编写的，但是 CUDA 与其他着色器语言有一些不同，这种语言不怎么关注像素和顶点，而是更关注一般目的的并行编程。假设在游戏中要模拟一个城市，并且有一个非常新奇的并行算法可以更新城市中的全部居民，那么这种计算在 GPU 上可以更快地执行，而 CPU 就可以被解放出来，执行其他任务。CUDA 通常用于科学研究项目，因为这是利用强大的计算能力的一种廉价的方式。使用 CUDA 的应用程序包括量子化学计算、心肌模拟和黑洞建模。

2.4 Tao 框架

Tao 框架是 C# 使用 OpenGL 库的一种方式。Tao 包装了许多 C 库(见表 2-1)，并使得在 C# 中使用这些函数变得很简单。Tao 中还绑定了 Mono，所以也可以用在 Linux 和 Mac 中。

通过 Tao，C# 不只可以使用 OpenGL，还可以使用其他一些有用的库。

表 2-1 Tao 框架中包含的库

| 库 | 用 途 |
|----------------------|--|
| Tao.OpenAl | OpenAL 是一个强大的音频库 |
| Tao.OpenGl | OpenGL 是我们将要使用的图形库 |
| Tao.Sdl | SDL(Simple DirectMedia Layer)，一个构建在 OpenGL 之上的 2D 库 |
| Tao.Platform.Windows | 支持通过 Windows.Forms 使用 OpenGL |
| Tao.PhysFs | 一个 I/O 的包装器，支持游戏资源的存档文件，如.zip 文件 |
| Tao.FreeGlut | OpenGL 实用程序工具包(OpenGL Utility Toolkit)是一组包装器，用于设置 OpenGL 程序和一些绘图例程 |
| Tao.Ode | Open Dynamics Engine 是在游戏中使用的一个实时物理引擎 |
| Tao.Glfw | OpenGL Framework 是一个可以在多个平台上使用的轻量级的包装器类 |

(续表)

| 库 | 用 途 |
|--------------|--|
| Tao.DevIL | DevIL 是将各种图片类型(bmp、tif、gif、png 等)加载到 OpenGL 的出色的工具 |
| Tao.Cg | Cg 是一种高级着色语言 |
| Tao.Lua | Lua 是游戏业最常用的脚本语言之一 |
| Tao.FreeType | 字体包 |
| Tao.FFmpeg | 主要用于播放视频 |

OpenAL 代表开放音频库(Open Audio Library)，是一个强大的开源库。《生化奇兵(BioShock)》、《雷神之锤 4(Quake 4)》、《毁灭战士 3(Doom III)》和《虚幻(Unreal)》等游戏都使用了这个音频库。它采用 OpenGL 作为模型，具有相同的状态机风格的设计和扩展方法。

SDL(Simple DirectMedia Layer)是一个跨平台的库，支持输入、声音和图形。SDL 在游戏开发商中非常流行，在独立或者开源游戏中使用得尤其多。使用 SDL 开发的最著名的开源游戏之一是 *FreeCiv*，它是《文明(Civilization)》的一个联机版本。多数 Linux 游戏端口中也使用了 SDL。

PhysFs 初看起来可能是一个物理库，但是实际上却是一个小型的 IO 库。它可以将全部游戏资源打包为一个较大的二进制文件，或者几个小的二进制文件。许多商业游戏都有类似的系统，例如《毁灭战士(Doom)》的 wad 系统或《雷神之锤(Quake)》的 pak 系统。它可以使游戏在发布后的修改和更新变得更加简单。

FreeGLUT 是 OpenGL 实用程序工具包的免费版本。这个库中的函数可以让用户马上就能够使用 OpenGL。它还有从键盘和鼠标接受输入的方法，以及绘制各种基本形状的方法，例如球形、立方形，甚至茶壶形(这个茶壶在计算机图形学中非常有名，它是由 Martin Newell 在犹他大学求学期间进行建模的。茶壶是一个非常复杂的表面，所以在测试新的图形技术时非常有用。动画电影《玩具总动员》中就有个典型的茶壶模型，DirectX 甚至有自己的茶壶创建方法 D3DXCreateTeapot()。在讲授 OpenGL 时经常用到 FreeGLUT，但是它的功能很有限，很少用于真正的项目。

ODE(Open Dynamics Engine)是一个可以用在多个平台上的物理引擎，可以完成碰撞检测和刚体模拟。PC 上的第一人称射击游戏《潜行者(S.T.A.L.K.E.R)》中就使用了 ODE。Glfw 是可以通过 Tao 使用的第三个可移植的 OpenGL 包装器。Glfw 代表 OpenGL 框架(OpenGL framework)，它的目的是扩展 GLUT 提供的功能。如果不想使用 SDL，但又确实想使用框架来访问 OpenGL，就可以考虑使用 Glfw。

DevIL(Developer's Image Library)是一个从磁盘加载纹理到 OpenGL 中的库。DevIL 与 OpenGL 有些类似，因为它也是一个状态机，并且有类似的方法名称。DevIL 是跨平台的，支持多种(43 种)不同的图片格式。Cg 是本章前面提到的一种着色器语言。通过使用 Tao.Cg，可以从文本文件或字符串中加载着色器程序，进行处理，然后在 OpenGL 中使用。

Lua 可能是游戏开发中最流行的脚本语言。它是一种小型的、易于嵌入的语言，表达力非常强。使用 Tao.Lua 可以在脚本和 C#程序之间传递函数和数据。Tao.FreeType 是一个基

本的字体包，可以将 FreeType 类型的字体转换成一幅位图。它的接口简单易用。

Tao 提供的最后一个库是 FFmpeg，这个名称由 MPEG(一个视频标准)和 FF(Fast Forward, 快进)组成。它提供了一种播放视频的方式。如果要在游戏中使用过场动画，FFmpeg 是一个不错的选择。

Tao 提供的所有库都是完全开源的。其中的多数库都可以免费用在商业项目中，但还是有必要阅读许可证中列出的具体说明。Tao 是一个出色的程序包，刚开始涉足游戏的开发商可以把它作为一个起点。对每个库的介绍不在本书的讨论范围之内，我们将只关注其中最重要的那些库。从第 5 章开始，我们将使用 OpenGL 和 Tao.Platform.Windows 库。第 6 章将讨论 DevIL。第 9 章将讨论使用 OpenAL 播放声音，以及使用 SDL 处理手柄输入。每个库都很有用，所以很有必要花些时间研究每个感兴趣的库。

2.5 小结

OpenGL 和 DirectX3D 是业界使用的两个主要的图形库。这些图形库是与底层的图形硬件进行通信的标准方式。图形硬件通常包含几个标准部分，在把 3D 顶点信息转换为屏幕上显示的 2D 帧时非常高效。这种从 3D 顶点到 2D 帧的转换被称为图形流水线。图形流水线分为两种：不可以编程的固定流水线和可以编程的流水线，后者允许通过着色器程序控制流水线的特定阶段。

Tao 框架是一个实用库的集合，其中包括 OpenGL。C# 可以通过 Tao 框架使用 OpenGL 编写游戏。Tao 框架还包含其他几个对游戏开发很有帮助的库。本书中将结合使用 OpenAL、DevIL 和 SDL 来开发一个简单的横向卷轴射击游戏。

第 3 章

现 代 方 法

软件开发中存在一些问题，例如游戏中有大量 bug、蓝屏、随机崩溃、程序不响应等。建筑师和土木工程师则不会面对这些问题。金字塔已经稳稳地屹立了四千多年。一幢建筑建好以后，可以经受风吹雨打，但是建筑行业的存在时间要比软件行业的存在时间久得多。

在过去几年中，出现了许多值得注意的新的软件开发方法，它们尝试改进软件的开发过程以及软件产品的最终质量。本章将介绍程序员可以利用的一些比较实用的方法。

3.1 实效编程

实效编程(pragmatic programming)是指在预定时间内满意地完成程序的能力。这需要首先理解对最终程序有什么要求，然后尽快编写程序的一个基本框架版本。这个框架版本将被不断修改，直到达到最初的要求。改动可能很多，甚至可能需要重写程序中的整段代码，但是没有关系。最重要的是程序可以基本运行。

在游戏开发中，有一个可以运行的程序总是令人满意的，因为这样的话，如果期限提前，或者开发人员被要求提供一个演示，他们就不至于手忙脚乱，而可以立即展示一个可以工作的程序。而如果是独立工作，就可以通过向他人展示程序来获得反馈。反馈可以让开发人员更好地理解哪些地方做得较好，哪些地方还有待改进。

3.1.1 游戏编程中的陷阱

许多开发人员一开始计划编写一个游戏，但是突然发现他们编写的是一个框架(或者叫

游戏引擎), 这个框架适合编写各类游戏。但由于开发人员自己从来没有编写出游戏, 就不能陷入框架陷阱, 不断地重新开发框架, 添加最新的功能, 却从来没有实际进入游戏创建部分。

实效编程可以帮助开发人员绕开框架陷阱。它明确地禁止开发人员开发框架, 创建可以工作的游戏是首要的任务。

实效编程有两个主要的编码原则: KISS 和 DRY。这是两个出色的开发工具, 可以改进开发人员的编码能力。

3.1.2 KISS

KISS 代表 **Keep It Simple Stupid**。也就是说, 尽量不要让游戏编程变得复杂。

“要是将所有的敌人和子弹都放在一个四叉树中的话, 我的《太空入侵者》游戏会更加高效。所以我最好开始创建一个四叉树……”。

如果想要制作游戏, 这是一种错误的思考方法。如果想要学习四叉树, 这是一种不错的思考方法。考虑对自己来说哪一种目的更重要, 然后把重点放在那个目的上。如果想开发游戏, 应让游戏尽快可以工作, 然后再考虑是否需要额外的复杂性。

3.1.3 DRY

DRY 代表 **Don't Repeat Yourself**。

如果到处看到重复的代码段, 那么这些代码就应该被合并起来。代码重复的位置越少, 在发生改动时需要修改的位置就越少。再次以“太空入侵者”游戏为例。

```
class Invader
{
    ...
    int _health = 10;
    bool _dead = false;
    void OnShot()
    {
        _health--;
        if (_health <= 0)
        {
            _dead = true;
        }
    }
    ...
}

class FlyingSaucer
{
    ...
```

```
int _health = 10;
void OnShot()
{
    _health--;
    if (_health <= 0)
    {
        _dead = true;
    }
} ...
}

class Player
{
    ...
    int _health = 10;
    void OnShot()
    {
        _health--;
        if (_health <= 0)
        {
            _dead = true;
        }
    }
    ...
}
```

在上面示例中，重复的代码出现在了每个类中。这意味着在发生改动时，需要修改 3 个不同的地方。此时编码人员需要做更多的工作，而且由于代码更多，理解起来也更加困难。同时，代码越多，出现 bug 的可能性也就越大。

接下来需要向《太空入侵者》游戏中的生物添加护盾。

```
class Invader
{
    ...
    int _health = 10;
    bool _dead = false;
    int _shieldPower = 2;
    void OnShot()
    {
        if (_shieldPower > 0)
        {
            _shieldPower--;
            return;
        }
    }
}
```

```
    }  
    _health--;  
    if (_health <= 0)  
    {  
        _dead = true;  
    }  
}  
...  
}  
class FlyingSaucer  
{  
    ...  
    int _health = 10;  
    bool _dead = false;  
    int _shieldPower = 2;  
    void OnShot()  
    {  
        if (_shieldPower > 0)  
        {  
            _shieldPower--;  
            return;  
        }  
        _health--;  
        if (_health <= 0)  
        {  
            _dead = true;  
        }  
    } ...  
}  
class Player  
{  
    ...  
    int _health = 10;  
    bool _dead = false;  
    int _shieldPower = 2;  
    void OnShot()  
    {  
        if (_shieldPower > 0)  
        {  
            _shieldPower--;  
            return;  
        }  
    }  
}
```



```
    _health--;  
    if (_health <= 0)  
    {  
        _dead = true;  
    }  
}  
...  
}
```

上面的代码变得更加难以维护。如果将重复的代码重构到一个父类中，代码将清晰得多。

```
class Spacecraft  
{  
    int _health = 10;  
    bool _dead = false;  
    int _shieldPower = 2;  
    void OnShot()  
    {  
        if (_shieldPower > 0)  
        {  
            _shieldPower--;  
            return;  
        }  
        _health--;  
        if (_health <= 0)  
        {  
            _dead = true;  
        }  
    }  
}  
class Invader : Spacecraft  
{  
}  
class FlyingSaucer : Spacecraft  
{  
}  
class Player : Spacecraft  
{  
}
```

把代码放到一个位置后，在将来修改代码的工作就更加轻松。

C++

C++从根本上不支持 DRY。在 C#中，有一个用于定义类和方法的文件(.cs 文件)。在 C++中，有一个定义类和所有方法原型的头文件。另外一个扩展名为.cpp 的文件再次定义了函数，这一次包含了函数体。这些文件如下所示。

```
// Header File Player.h
class Player
{
    void TeleportTo(Vector position);
};
// CPP File Player.cpp
void Player::TeleportTo(Vector position)
{
    // code
}
```

如果想要添加返回值，报告传送是否成功，就必须修改两个位置的代码。这不符合 DRY 原则。在 C++中，类实现定义在一个文件中，类接口定义在另一个文件中，上面的代码段显示了代码的这种重复性。几乎所有的 C++程序都会使用类，所有的类都要求重复的实现。因此，C++的工作方式已经决定了它不符合 DRY 原则。Java 和 C#的开发时间都晚于 C++，并且受 C++的影响极大(采用了 C++的大量语法和特性)，但是两种语言都决定放弃使用头文件。

只要记住 DRY 原则即可，你可以写出更好的程序。

如果想要阅读关于实效编程的更多介绍，推荐阅读 Andrew Hunt 和 David Thomas 编著的 *The Pragmatic Programmer*。详细内容请参见附录 A。

3.1.4 源代码控制

源代码就是在编程时输入到编辑器中的内容。编译源代码时将产生机器码。源代码由人类用户阅读，机器码由计算机阅读。计算机使用机器码执行程序。源代码控制是一个程序，负责记录对源代码做出的全部更改(见图 3-1)。

知道代码的更改和开发过程是很有帮助的。源代码控制可以让程序员不受限制地执行“撤消”操作。当发生错误时，总是可以回到以前的源代码版本。当开发新功能，但是无意中破坏了代码时，这种功能十分有用。如果没有源代码控制，就不得不痛苦地回忆做出的每个更改。通过使用源代码控制，可以比较当前的版本和之前的版本，并立即看到改变，如图 3-2 所示。

```
class Creature
{
    string _name;

    public string GetName()
    {
        return _name;
    }
}

class Creature
{
    string _name;
    int _health = 50;
    int _power = 10;

    public string GetName()
    {
        return _name;
    }

    public void Attack(Creature otherCreature)
    {
        // todo: Write code
    }
}

class Creature
{
    string _name;
    int _health = 50;
    int _power = 10;

    public string GetName()
    {
        return _name;
    }

    public void TakeDamage(int amount)
    {
        _health -= amount;
    }

    public void Attack(Creature otherCreature)
    {
        otherCreature.TakeDamage(_power);
    }
}
```

版本 1

版本 2

版本 3

图 3-1 对源代码做出的更改

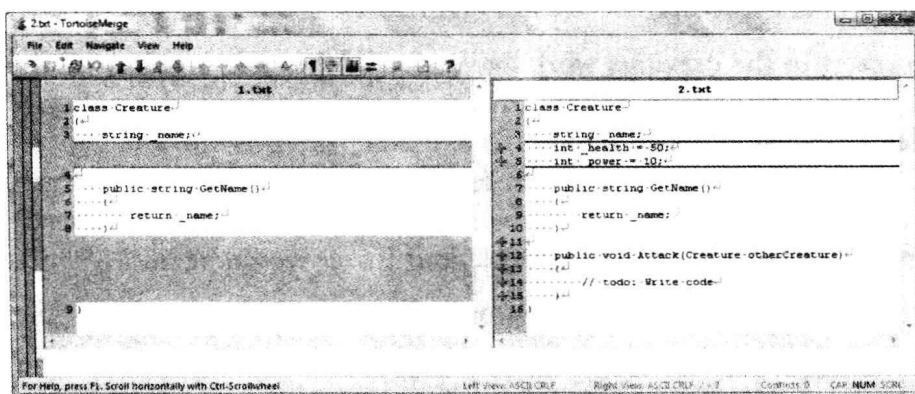


图 3-2 比较对源代码做出的更改

也可以返回到之前可以工作的代码版本，并删除添加的所有错误代码。返回就是指将

当前的代码文件替换为这些文件的之前版本的过程。就像是次时间旅行！在厨房中切胡萝卜时可能不小心把手指切破了。要是人也有源控制功能，就没有问题。只需要切换回到 15 分钟以前的自己，手指就会完好无损。

源代码控制也非常适合备份。它将所有的代码存储在一个位置，即源代码库。这个位置可能是计算机上的本地目录(c:\source_control)，或者是可以通过 Internet 访问的世界上任意位置的服务器。即使房间被烧毁了，源代码也是安全的。

源代码管理系统允许创建分支。这是一种将程序划分为不同版本的方法。例如，你的国际象棋游戏做的很不错，但是觉得它要是有一个第一人称射击模式会更好。添加这种模式需要修改大量代码，而且你可能会面临失败。使用源代码管理时就不必担心。你可以创建一个分支，所有新的代码修改都可以被放到一个新的分支库中。这里的修改不会影响主代码库，也叫做主干。如果没有成功地创建 ChessFPS，则可以切换回到主干，把失败的分支丢到一边。如果成功，之后可以把分支和主干合并在一起。

源代码控制对于独立的开发人员十分有用，但是其主要用途之一是使一个程序员团队同时工作在一个代码库上。源代码控制允许任意一个程序员修改代码，而其他任意程序员可以更新并看到这些修改。要与几个朋友一起开发游戏，源代码控制是必不可少的。

使用源代码控制系统

希望现在你已经明白了源代码控制系统有多么优秀，也知道自己应该使用它。本书的代码使用的源代码控制系统是 Subversion，但是了解关于其他源代码控制系统的信息是有益无害的。

- CVS——主要用在较早的项目中。最好不要使用这个系统，因为 Subversion 包含它的全部功能，还提供了更多的功能。
- Visual Source Safe——尽量不要使用这种系统。Microsoft 开发了这种系统，但是即使 Microsoft 自己在内部也不使用它。Visual Source Safe 在稳定性上存在问题。
- Subversion——优秀而且免费的开源源代码控制系统，可以与 Windows 很好地集成在一起。我的首选源代码控制系统。
- Perforce——许多游戏公司都使用这种系统，它包含非常好的合并工具，可很好地处理大量二进制数据(如图片和模型)。价格不菲。
- Git——Git 是一个相当新的系统，用作 Linux 内核的源代码控制系统。它不依赖于中央服务器，速度很快，而且可以扩展。目前这种系统还主要基于命令行，不过有一些 GUI 工具。
- Mercurial——也是一个新成员，主要基于命令行，同样也有一些 GUI 工具。它的目标是实现高性能、可扩展性、分散性，以及完全的分布式协同开发。Git 和 Mercurial 的目标类似。

如果想要站在版本控制(revision control)的最前沿，那么就去研究 Git 或 Mercurial。如果想要使用简单易用的工具，则推荐使用 Subversion。

3.1.5 单元测试

当程序员编写新功能时，通常会写一段代码来测试新功能。单元测试是把这些代码片

段合并在一起的一种灵巧的方式。而后，每次编译代码时，它们就可以运行。如果某个之前可以运行测试突然失败，那么很明显某个地方出错了。单元测试通常是很小的代码段，只测试一种功能。

在单元测试中，首先编写测试、然后编写代码是很常见的。通过示例很容易理解这一点。假设我们正在开发一个游戏，需要使用一个类来表示玩家。玩家应该有一个生命值，当创建一个玩家时，他的生命值应该大于 0。相应的代码为：

```
class PlayerTests
{
    bool TestPlayerIsAliveWhenBorn()
    {
        Player p = new Player();
        if (p.Health > 0)
        {
            return true; // pass the test
        }
        return false; // fail the test
    }
}
```

现在这段代码还无法编译，因为还没有创建玩家类。不过这个测试的用途应该很明显。创建的玩家的生命值应该大于 0，否则测试将失败。接下来就创建该玩家。

```
class Player
{
    public int Health { get; set; }
}
```

现在可以使用单元测试软件运行所有的测试。测试将会失败，因为默认情况下，整数被初始化为 0。为了通过测试，必须修改代码。

```
class Player
{
    public int Health { get; set; }
    Player()
    {
        Health = 10;
    }
}
```

再次运行单元测试软件，测试将通过。现在知道了这段代码在测试条件下可以工作，下面再添加两个测试。

```
class PlayerTests
{
    bool TestPlayerIsHurtWhenHit()
```

34 第I部分 背景知识

```

    {
        Player p = new Player();
        int oldHealth = p.Health;
        p.OnHit();
        if (p.Health < oldHealth)
        {
            return true;
        }
        return false;
    }
    bool TestPlayerIsDeadWhenHasNoHealth()
    {
        Player p = new Player();
        p.Health = 0;
        if ( p.IsDead() )
        {
            return true;
        }
        return false;
    }
}

```

这里又添加了两个测试，每个测试需要一个新的玩家函数。

```

class Player
{
    public int Health { get; set; }
    public bool IsDead()
    {
        return false;
    }
    public void OnHit()
    {
    }
    public Player()
    {
        Health = 10;
    }
}

```

单元测试的一般方法是编写测试，更新代码，然后运行测试。如果测试失败，就修改代码，直到通过测试。现在如果在更新后的代码上运行新测试，它们都会失败。接下来我们将解决这个问题。

```

class Player

```

如果发现了一个 **bug**，也不用担心，这是故意加到那里的。运行测试，它们应该都可以通过。代码得到了确认，但是测试不会覆盖全部代码。例如，代码中有一个 **TestPlayerIsDeadWhenHasNoHealth**，但是没有“测试生命值大于 0 时玩家是否还活着”的方法。这个方法的实现给读者留作练习。

在单元测试中，首要任务是编写一个重现 bug 的测试。编写出这个测试后，就可以修复代码，并通过测试证明这一点。下面的测试可以消除这种 bug：

```
class PlayerTests
{
    bool TestPlayerShouldBeDead()
    {
        Player p = new Player();
        p.Health = 2; // give him low health
        // Now hit him a lot, to reproduce the bug description
        p.OnHit();
        p.OnHit();
        p.OnHit();
        p.OnHit();
        p.OnHit();
        if ( p.IsDead() )
        {
```

```
        return true;
    }
    return false;
}
}
```

运行测试时，它会失败。现在有了一个重现 bug 的单元测试，必须将代码修改为通过这个测试。看上去死亡函数是修复 bug 的一个好位置。

```
class Player
{
    bool IsDead()
    {
        return Health <= 0;
    }
    ...
}
```

如果玩家的生命值为 0 或更低，就会死亡。这样所有的单元测试就都会通过，而我们现在可以确定这个 bug 不会在我们不知道的情况下突然出现。

1. 测试驱动的开发

编写测试，然后编写框架代码，最后使代码通过测试，这种方法叫做测试驱动的开发 (Test Driven Development)。以这种方式开发代码可以减少 bug 的数量，并且开发人员可以确信代码实现了应有的功能。如果在开发一个大型项目，这一点十分重要。测试驱动的开发还鼓励类尽量模块化和自包含。这是因为单元测试应该较小，所以不能为了运行一个简单的测试而在一个单元测试中编写 50 个类，然后设置它们。正因如此，要避免类之间存在大量无用的耦合。

单元测试的另外一个主要的优点是重构。例如，如果游戏中的物品栏系统在很大程度上需要重新设计，借助单元测试就可以确定自己正确地重新设计了系统。

通过单元测试，程序员可以自信地声称自己的代码是安全的。但不是什么都能测试，测试图形就十分困难，而对于库(例如 OpenGL)，你只能假定它们会正确地工作。

2. C#中的单元测试

用于 C#的最好的单元测试软件是 NUnit(见图 3-3)。它很简单，也易于使用。

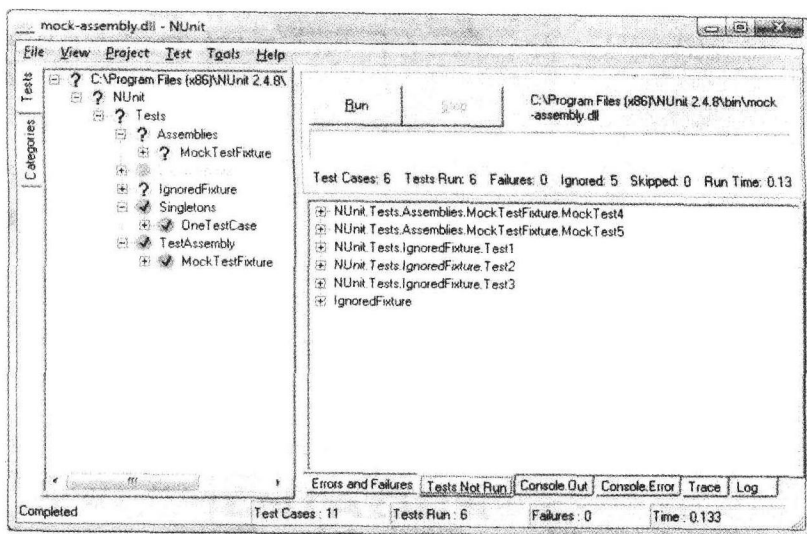


图 3-3 在 NUnit 中运行测试

在后面的章节中，我们将下载 NUnit，并讲解其安装过程。本书的所有代码都经过了单元测试，但是为了简洁起见，大多数单元测试代码都没有包含在代码示例中。

3.2 小结

编写高质量的软件很难，编写没有 bug 的软件很难，而且按时完成项目也很难(有时候甚至无法按时完成项目)。现代编程方法和实践尝试改进软件开发过程。

实效编程是指在预定时间内满意地完成程序的能力。这是一种开发方法，重点是尽快创建程序的最小版本，然后迭代地开发程序，直到程序符合最初的设想。实效编程的两个主要指导原则分别是 DRY 和 KISS。DRY 代表 Don't Repeat Yourself; KISS 代表 Keep It Simple Stupid。结合在一起，这两个指导原则意味着应该避免代码或功能重复，而且应该避免使程序过度复杂。

源代码控制用于使代码保持安全，并使多个开发人员可以同时工作在同一个项目上。单元测试是编写小测试来确定代码符合预期的一种实践。这些小测试帮助描述代码库的使用方法，并且也用于捕获在对代码做出较大更改时可能引入的 bug。这些工具和指导原则可以使软件开发过程变得更加简单，并且可以使最终产品更加优雅和健壮。

第 II 部分

实 现

第 4 章 设置

第 5 章 游戏循环和图形

第 6 章 游戏结构

第 7 章 渲染文本

第 8 章 游戏数学

第 9 章 创建游戏引擎

第 10 章 创建一个简单的卷轴射击游戏

第 11 章 创建自己的游戏

第 4 章

设 置

本章将介绍如何开始使用开发游戏所需的工具和库。这里介绍的所有工具都是免费的，你很快就会有一个完全可以工作的 C# IDE，一个可以保持所有文件安全、同时记录程序中的全部改动的源代码控制系统，以及一种对代码执行单元测试的方法。读者可以从配套光盘的 App 目录中找到本章介绍的所有工具的安装程序。

4.1 Visual Studio Express——C#可以使用的免费 IDE

Visual Studio Express 是 Microsoft 为 C# 提供的一个免费的 IDE。通过 Visual Studio 编写 C# 程序十分容易，简单地按下一个按钮就可以编译和运行代码。编写 C# 程序还有其他方法，例如，在记事本中编写全部程序代码，然后在命令行中使用编译器来运行程序。另外，还有其他许多 IDE 可以编辑 C# 代码，但是用于编辑 C# 代码的免费工具中最好的就是 Visual Studio Express。

Visual Studio Express 可以从 Microsoft 的网站(<http://www.microsoft.com/express/vcsharp/>)上下载。本书的光盘中也提供了该软件。

按照如图 4-1 所示的安装向导完成安装工作。向导会询问是否安装 Silverlight 运行时(本书中不会使用 Silverlight，所以不需要安装)。向导完成并重启计算机以后，Visual Studio Express 就安装完成了。现在就可以开始进行一些 C# 编程了。

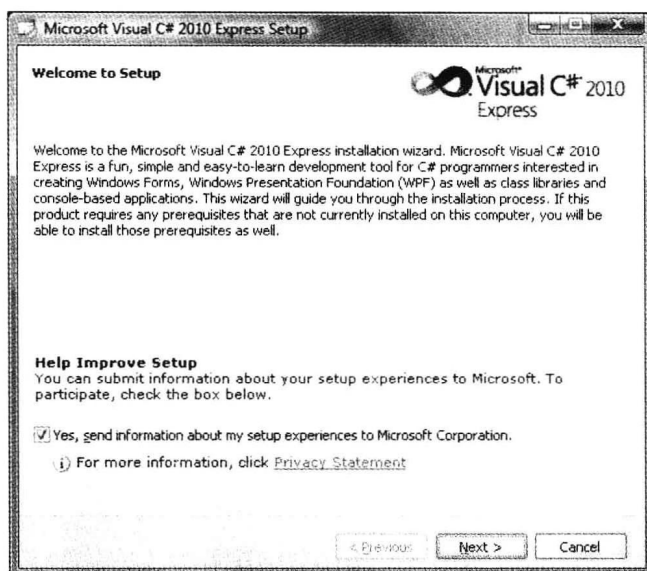


图 4-1 Visual Studio Express 的安装向导

4.1.1 Hello World 程序

创建一个快速的 Hello World 程序可以演示 Visual Studio Express 的特性。Start 菜单中应该包含了 Visual Studio Express 的快捷方式，所以可以从 Start 菜单中启动它。

在 Visual Studio Express 中，有解决方案和项目之分。解决方案可以包含任意数量的项目。每个项目都是代码文件的集合，一个项目可以使用另外一个项目的代码。通过选择 File | New Project 命令，可以创建新项目。弹出的对话框如图 4-2 所示。

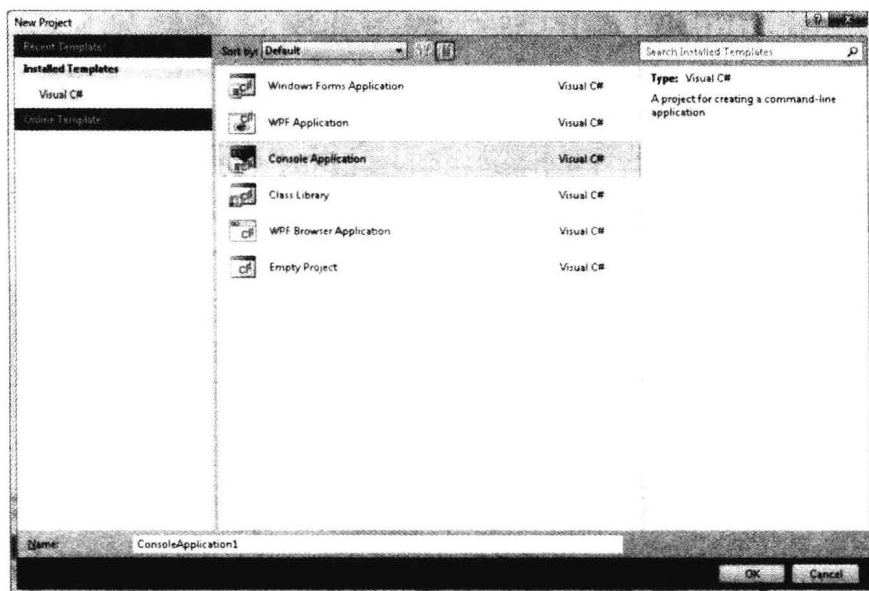


图 4-2 创建新项目

该对话框中列出了多种项目类型供用户选择。Windows Forms Application 这种类型的项目使用了窗体库。窗体是一个.NET 术语,用于描述几乎所有 Windows GUI 程序。Windows Console Application 用于基于文本的项目。Class Library 是由其他项目使用的代码,它们不能单独执行。

Hello World 程序一般是基于文本的,所以很适合使用 Windows Console Application。在 New Project 对话框中还需要为项目命名,默认的名称为 ConsoleApplication1。更具描述性的名称(如 HelloWorld)可以使项目的用途更加明显。单击 OK 按钮将创建项目。

图 4-3 显示了新项目。应用程序的主要部分是 Program.cs,这是项目中唯一的代码文件。在右侧是一个名为 Solution Explorer 的子窗口。Solution Explorer 中包含由解决方案中的全部项目构成的一个树。树的顶层显示了刚才创建的 HelloWorld 解决方案。这个树的唯一一个子节点是一个也叫做 HelloWorld 的项目。解决方案和项目可以具有相同的名称。

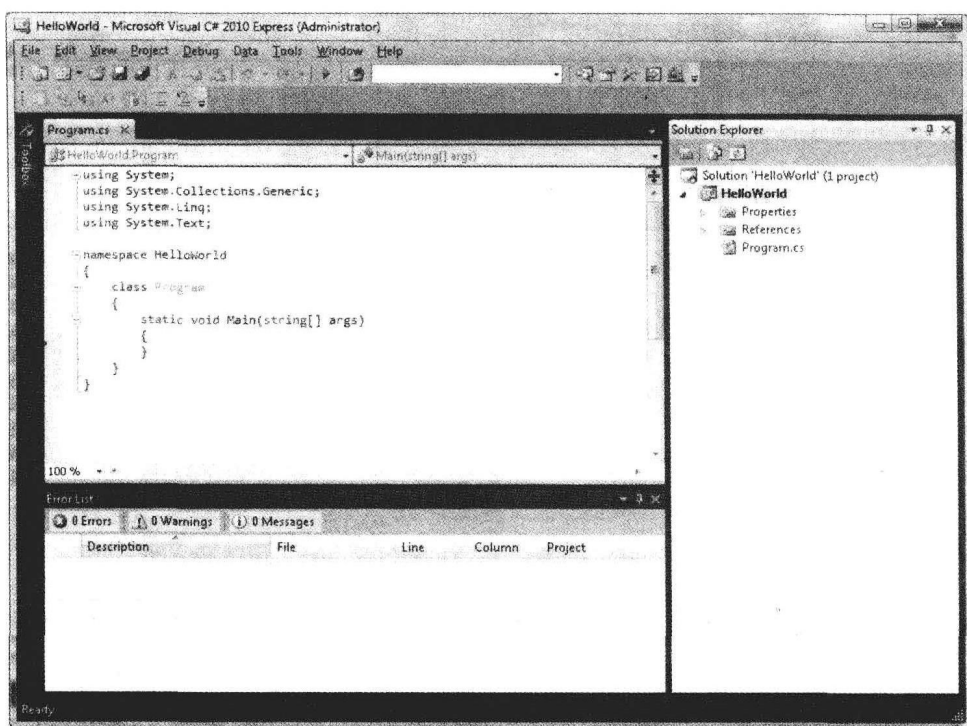


图 4-3 开始一个全新的项目

在 Visual Studio Express 窗口的最顶部是一个工具栏。工具栏中包含一个绿色的小箭头。单击该按钮可以编译项目并执行代码。现在单击该按钮。

此时将显示一个控制台窗口,因为程序中现在还没有代码,该窗口将迅速消失。C#程序通过调用 Program 类的静态 Main 方法开始执行。通过在这里编写代码,使程序输出 Hello World。

```
namespace HelloWorld
{
    class Program
```

```
{  
    static void Main(string[] args)  
    {  
        System.Console.WriteLine("Hello World");  
        System.Console.ReadKey();  
    }  
}
```

第一行代码调用一个函数，向系统控制台输出 **Hello World**。第二行代码等待用户在系统控制台中按下某个键。如果没有第二行，程序将运行并显示 **Hello World**，然后由于没有其他任务，所以会立即关闭。

单击绿色箭头，然后开始享受 **Hello World** 带给你的乐趣吧(见图 4-4)。

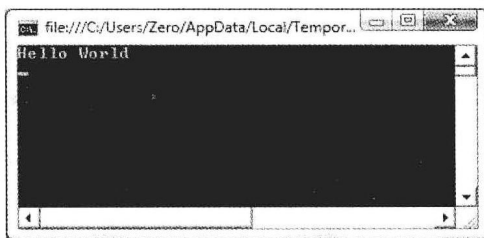


图 4-4 Hello World 程序

结果表明一切正常。保存代码很重要，以便可以在以后使用它。选择 **File | Save All** 命令。弹出的对话框要求指定文件的存储位置。默认的存储位置为 **My Documents** 文件夹中的 **Visual Studio 2010** 文件夹下。这个位置没问题。

4.1.2 关于 Visual Studio Express 的提示

如果人工输入 **Hello World** 程序，你会发现 **Visual Studio** 通过自动完成功能和下拉列表来显示名称空间、类和合适的变量，以帮助你减少输入的代码量。这是 **Visual Studio** 提供的比较明显的帮助。**Visual Studio** 还会在一些不太容易注意到的地方为程序员提供帮助，下面将一一阐述。

1. 自动重构和代码生成

重构是软件开发中的一个术语，指的是重新安排或者重新编写当前应用程序的一些部分，使代码更加清晰、简洁和高效，但是代码的功能不会改变。代码生成功能使 **Visual Studio** 替程序员编写一些代码。**Visual Studio** 中提供了许多辅助工具，并将它们划分为两个类别，分别是用于修改代码的 **Refactor** 和用于自动添加新代码的 **Generate**。自动重构分为很多种，下面列出的 3 种是我自己最常用到的。

重命名

名称空间、类、成员、字段和变量的名称都应该具有描述性，这样可以帮助程序员理解它们的用途。下面的示例代码的命名风格就不太好。


```
class GameThing
{
    int _health = 10;
    int _var2 = 1;

    public void HealthUpdate(int v)
    {
        _health = _health - Math.Max(0, v - _var2);
    }
}
```

除非游戏中的每个对象都会有生命值, 否则 **GameThing** 不是一个好名称。将这个类命名为 **GameCreature** 会更好一些。假设现在已经完成了一个大型游戏一半的开发工作。**GameThing** 类在多个不同的文件中出现了几百次。如果想手动重命名 **GameThing** 类, 可以手动编辑那几百个文件, 或者可以小心地使用 Visual Studio 的 Find and Replace 工具(这个代码库中也有很多使用类似名称的完全无关的类(例如 **GameThingManager**), Find and Replace 工具可能会无意中修改它们)。只需鼠标单击两次, Visual Studio 的 refactor 函数可以安全地重命名 **GameThing** 类。

右击 **GameThing**, 打开如图 4-5 所示的快捷菜单。选择 **Refactor | Rename** 命令。

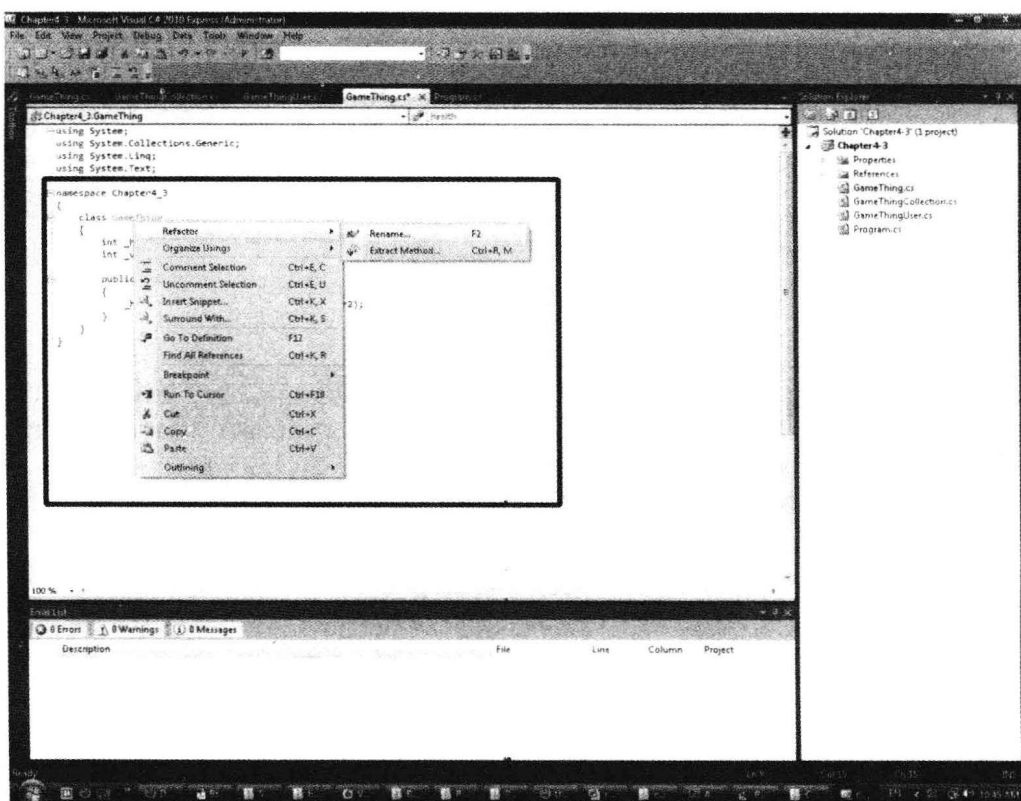


图 4-5 选择 Refactor|Rename 命令

此时将弹出一个新的对话框，在这里可以输入新名称，如图 4-6 所示。

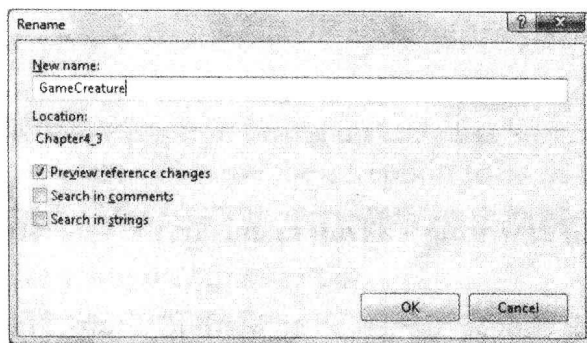


图 4-6 输入一个新的、更具描述性的名称

这将会重命名出现在代码中任意位置的该类。对话框中还提供了重命名注释和字符串中的类名的选项。对函数和变量名可以执行相同的过程，得到的结果如下所示。

```
class GameCreature
{
    int _health    10;
    int _armor     1;

    public void TakeDamage(int damage)
    {
        _health    _health - Math.Max(0, damage - _armor);
    }
}
```

现在代码就容易理解多了。

创建函数

编写一段代码时，经常需要用到还没有编写的函数。Visual Studio 中的生成工具允许编写代码，就好像函数确定存在一样，然后自动创建该函数。下面以 Player 类的更新循环为例。

```
class Player
{
    public void Update(float timeSinceLastFrame)
    {

    }
}
```

在更新循环中，需要更新玩家动画。这个任务最好在一个单独的函数中完成，该新函数如下所示。

```
class Player
{
    public void Update(float timeSinceLastFrame)
    {
        UpdateAnimation(timeSinceLastFrame);
    }
}
```

这个函数还不存在，但是不必自己手动编写这个函数，生成函数会完成这项工作(见图4-7)。

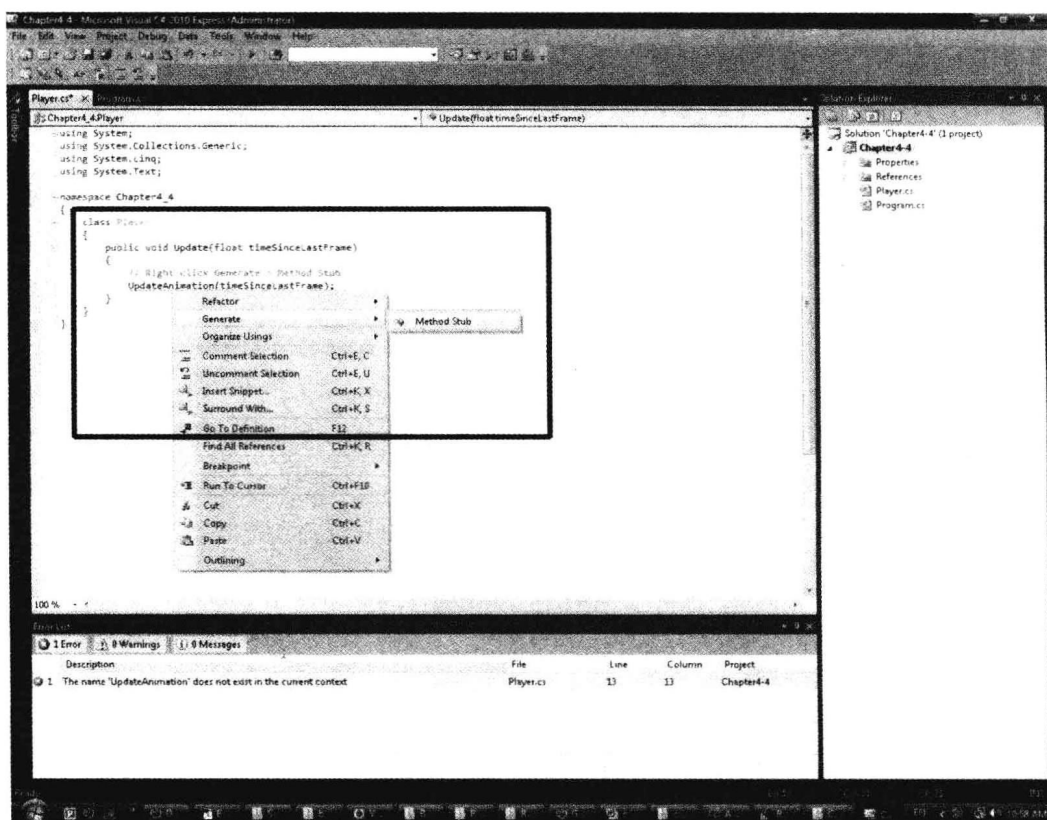


图 4-7 用于创建函数的 Generate 菜单

右击函数名，然后选择 **Generate|Method Stub** 命令。在 **Player** 类中将创建下面的代码。

```
private void UpdateAnimation(float timeSinceLastFrame)
{
    throw new NotImplementedException();
}
```

这个过程中不需要手动录入代码。函数中有一个异常，指出还没有编写代码。在程序

中调用这个函数会导致抛出该异常。在为函数编写代码后，可以删除该异常。

划分代码块

在大型项目中，经常会发现某个部分变得很臃肿。公共对象(如玩家)的更新循环可能会包含几百行代码。如果发生这种情况，将这部分代码分解成几个单独的函数是个好主意。假设游戏世界是循环更新的。

```
class World
{
    bool _playerHasWonGame = false;
    public void Update()
    {
        Entity player = FindEntity("Player");

        UpdateGameCreatures();
        UpdatePlayer(player);
        UpdateEnvironment();
        UpdateEffects();

        Entity goldenEagle = FindEntity("GoldenEagle");

        if (player.Inventory.Contains(goldenEagle))
        {
            _playerHasWonGame = true;
            ChangeGameState("PlayerWinState");
        }
    }

    // additional code
}
```

这个循环并不太差，但是最后一段代码不够整洁，可以把它移动到一个单独的函数中。为了自动完成这个任务，选择从 `Entity goldenEagle` 直到 `if` 语句的结束大括号的代码部分，然后右击鼠标。在弹出的快捷菜单中。选择 **Refactor|Extract Method** 命令，如图 4-8 所示。在提示为方法命名时，输入 **CheckForGameOver**。选择的代码将被移除并放到一个新函数中，它使用的任何变量将作为新创建的函数的参数传入。重构后的代码如下所示：

```
bool _playerHasWonGame = false;
public void Update()
{
    Entity player = FindEntity("Player");

    UpdateGameCreatures();
    UpdatePlayer(player);
```

```
UpdateEnvironment();  
UpdateEffects();  
  
CheckForGameOver(player);  
}  
  
private void CheckForGameOver(Entity player)  
{  
    Entity goldenEagle = FindEntity("GoldenEagle");  
  
    if (player.Inventory.Contains(goldenEagle))  
    {  
        _playerHasWonGame = true;  
        ChangeGameState("PlayerWinState");  
    }  
}
```

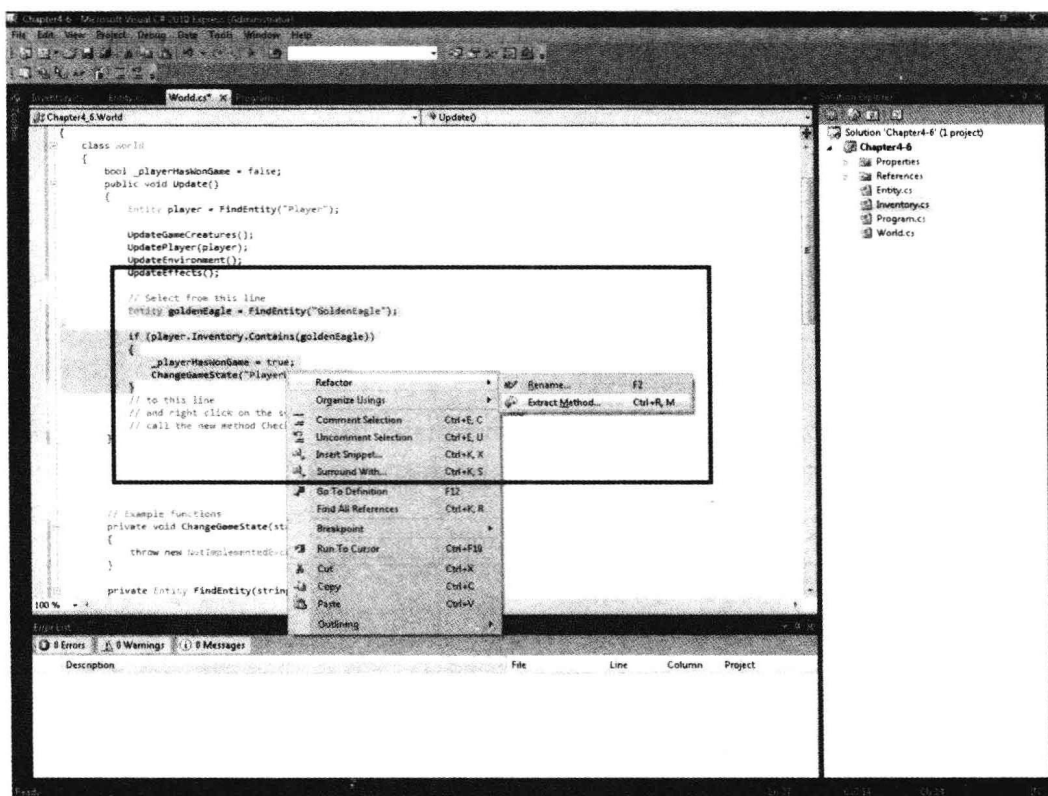


图 4-8 用于提取函数的 Refactor 菜单

现在世界更新函数更短、更易读。

研究和熟悉重构函数很有必要，因为它们可以节省大量开发时间。

快捷键

Visual Studio 和 Visual Studio Express 提供了许多有用的快捷键。表 4-1 列出了最有用的一部分快捷键。一些快捷键是大多数文本编辑器通用的。

表 4-1 Visual Studio 中的快捷键

| 组 合 键 | 作 用 |
|-------------------|---|
| Ctrl+A | 全选 |
| Ctrl+空格键 | 强制打开自动完成框 |
| Shift+Home | 选择从当前位置到行首的代码 |
| Shift+End | 选择从当前位置到行尾的代码 |
| Shift+← | 使用箭头键选择左边最邻近的字符 |
| Shift+→ | 使用箭头键选择右边最邻近的字符 |
| Ctrl+} | 如果光标邻近大括号字符, 该组合键将找到相应的结束或者开始大括号 |
| Ctrl+K, 然后 Ctrl+F | 这将正确地缩进所有选中的文本; 如果从 Internet 上复制了某个示例的代码, 结果格式变得比较混乱, 这些组合键十分有用 |
| Tab | 如果选中了某些代码, 然后下 Tab 键, 选中的代码将在被清除后缩进一个制表位 |
| Shift+Tab | 如果选中了某些代码, 然后按下了 Shift+Tab 组合键, 选中的代码将向前缩进一个制表位 |
| Ctrl+U | 所选的文本将被改为小写 |
| Ctrl+Shift+U | 所选的文本将被改为大写 |
| Alt+鼠标左键拖选 | 可以垂直选择文本; 选中的是字符列而不是字符行 |
| F5 | 生成并执行项目 |
| F12 | 如果光标在某个方法或者类名上, 则会把光标跳到定义方法或类的位置 |
| Ctrl+Shift+B | 生成当前项目 |
| Ctrl+K, Ctrl+C | 注释掉当前选中的代码 |
| Ctrl+K, Ctrl+U | 取消对当前选中代码的注释 |
| Ctrl+F | 打开 Find 对话框 |
| Ctrl+Shift+F | 打开 Find 对话框, 但是这一次将搜索全部项目, 而不是当前的代码文件 |

4.2 Subversion

Subversion 是一个源代码控制系统, 用于保护源代码安全。Subversion 通常被缩写为 SVN。在 Windows 上安装 SVN 最简单的方式是使用 TortoiseSVN。TortoiseSVN 可以集成到 Windows 的快捷菜单中, 使得管理源代码控制操作变得很简单。

4.2.1 获取

TortoiseSVN 的官方网站是 <http://tortoisesvn.net/>, 该软件最新的版本可以从 <http://tortoisesvn.net/downloads/> 上下载。本书的配套光盘上也提供了该软件。TortoiseSVN 有 x86 和 x64 两个版本, 分别针对 32 位计算机和 64 位计算机。

4.2.2 安装

安装操作很简单, 只需双击图标并接受所有默认选项。这会把程序安装到 Program Files 文件目录中。安装完成后, 需要重启系统。

系统重启后, 在桌面上右击鼠标。在弹出的快捷菜单中增加了 3 项, 如图 4-9 所示。这些菜单项用于管理源代码。

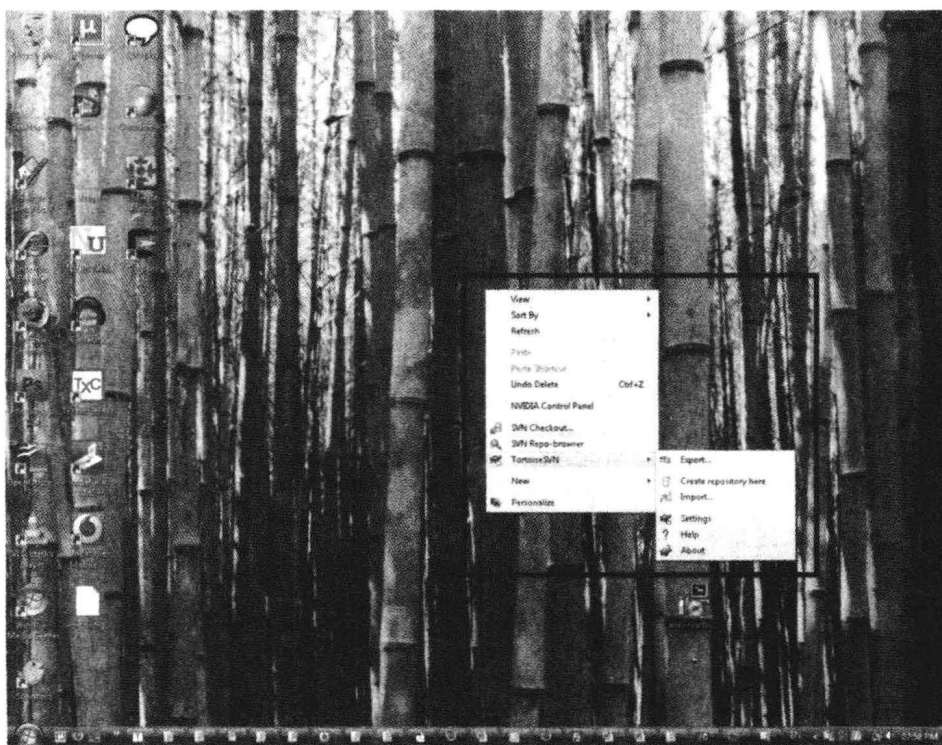


图 4-9 SVN 快捷菜单

4.2.3 创建源代码控制库

源代码控制库是存储所有的代码和数据的地方。我们将在硬盘上创建这个库。选择一个容易备份的位置是一个很好的选择。像 My Documents 就是一个不错的位置。可以把库看作一个保险箱, 它将照管好编写的所有代码。

确定了库的位置后, 再创建一个新目录。所有的源代码控制文件都将放在这个目录下。可以随意命名该目录。这里将其命名为 MyCode。打开文件夹并右击鼠标。从弹出的快捷菜单中选择 Tortoise SVN 命令。这会显示出更多的一些菜单项, 如图 4-10 所示。选择 Create

Repository here 命令。

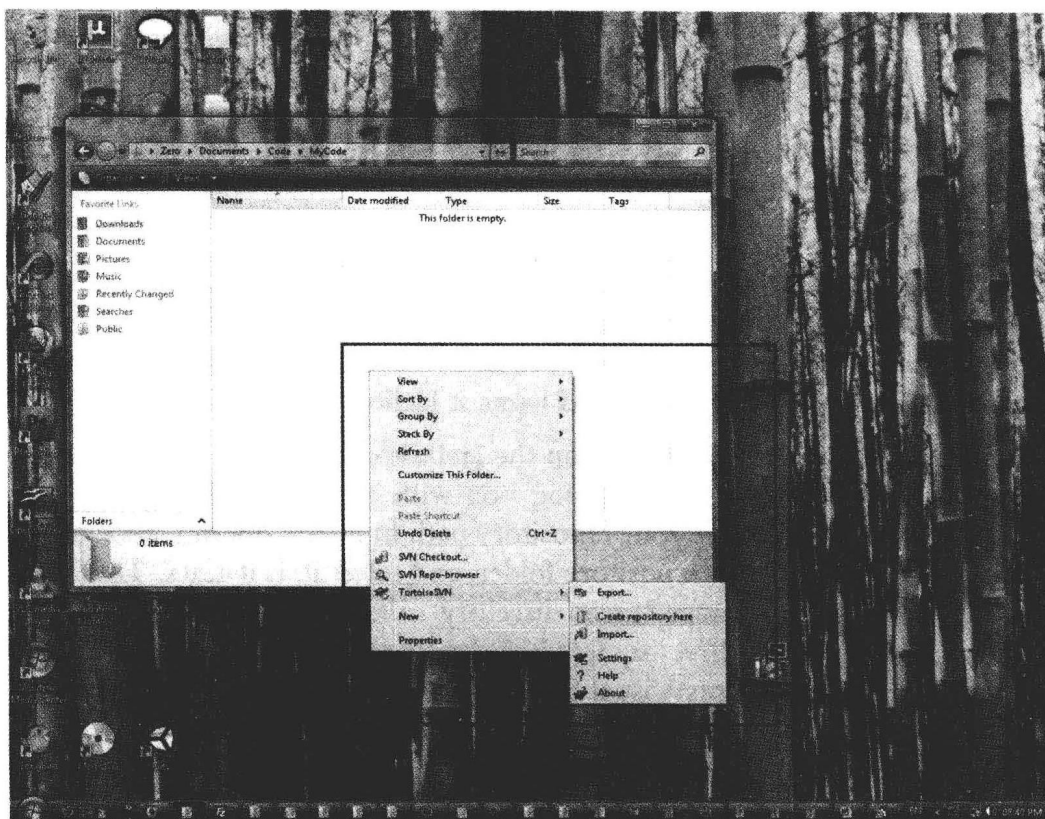


图 4-10 使用快捷菜单创建库

目录中将出现一些新文件，如图 4-11 所示。

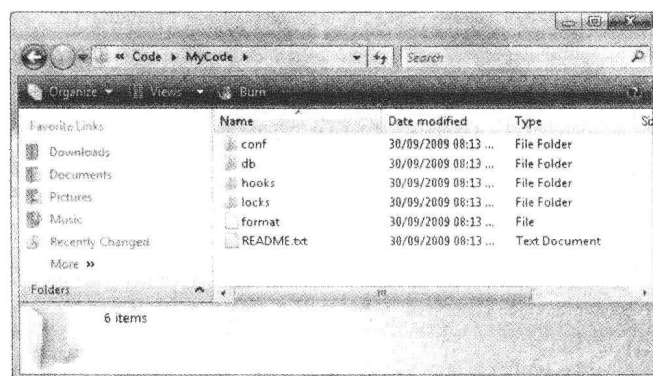


图 4-11 库中的文件

4.2.4 添加到库中

还记得 Hello World 项目吗？这些代码很重要，应该放到源代码控制系统中，以便保证

其安全性。如果将项目保存到默认目录,在 Windows 7 和 Windows Vista 上将会是 C:\User\YourUserName\Documents\Visual Studio 2010\Projects\,在 Windows XP 上将会是 C:\Documents and Settings\YourUserName\My Documents\Visual Studio 2010\Projects。

现在,打开源代码控制库所在的目录,像以前一样浏览库。右击最右边的窗口,从弹出的快捷菜单中选择 Create Folder 命令,并将新文件夹命名为 HelloWorld。

在项目目录中,右击 HelloWorld 文件夹,在弹出的快捷菜单中选择 SVN Checkout 命令。打开得到对话框中包含一个 URL of Repository 文本框,其中应该包含了库的路径。如果没有,则单击文件夹图标,浏览到库文件夹,然后选中该文件夹。单击 OK 按钮。对话框将会改变,显示一个名为 HelloWorld 的目录。选中该目录。在 Windows 7 和 Windows Vista 上,Checkout Directory 文本框中应该显示 C:\Users\YourUserName\Documents\Visual Studio 2010\Projects\HelloWorld,在 Windows XP 上,该文本框中应该显示 C:\Documents and Settings\YourUserName\My Documents\Visual Studio 2010\Projects\HelloWorld。

现在,库还是空的,其中只有一个刚才创建的空的 Hello World 目录。下一步是将这个空目录(check out)到与 Hello World 项目相同的位置。在登出窗口中,单击 OK 按钮。这将把库登出到与当前代码相同的位置。图 4-12 显示了 Hello World 目录的内部结构,以及发生的改动。

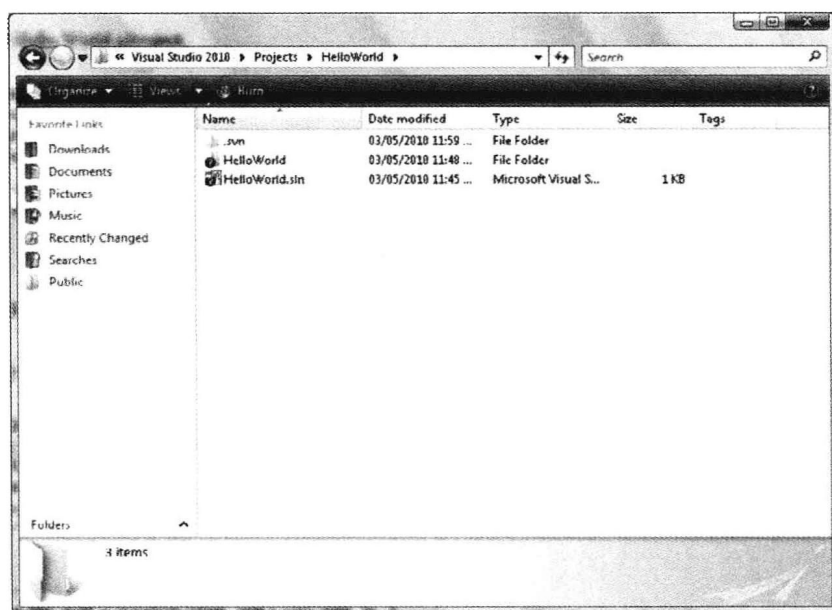


图 4-12 登出的 HelloWorld 项目

所有的文件现在都有一个小问号。这表明文件还没有被添加到源代码控制库中,但是它们位于登出的目录中。还有一个隐藏的文件夹.svn。这个隐藏的文件夹包含 SVN 用于管理自身的信息。

下一步是将所有重要的代码文件添加到 HelloWorld 库中。HelloWorld.sln 很重要,这个文件说明了如何设置解决方案及包含项目。

右击 HelloWorld.sln,在弹出的快捷菜单中选择 TortoiseSVN|Add 命令(见图 4-13),然

后刷新目录，HelloWorld.sln 旁边将显示一个加号，如图 4-14 所示。HelloWorld.sln 已经被添加到了库中。首先添加其余文件。项目目录是一个叫做 HelloWorld 的目录。右击该目录，然后在弹出的快捷菜单中选择 Add 命令，与之前对 HelloWorld.sln 采取的操作相同。

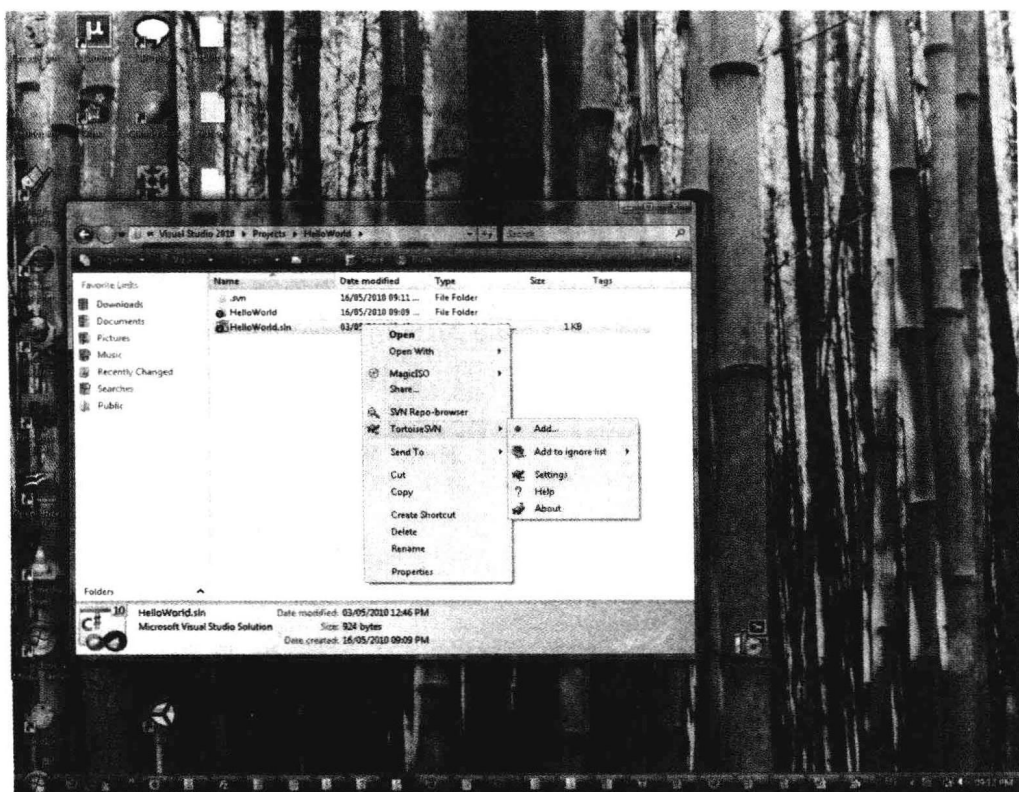


图 4-13 添加到库中



图 4-14 加号表明文件被添加到了源代码控制中

这将显示目录中包含的所有文件的列表。取消对所有的文件的选择，然后输入下面的内容：

```
HelloWorld/  
HelloWorld/HelloWorld.csproj  
HelloWorld/Program.cs  
HelloWorld/Properties  
HelloWorld/Properties/AssemblyInfo.cs
```

这是我们将要编辑的文件。其他文件将被自动生成，所以没有必要把它们添加到源代码控制系统中。选中文件后，就可以提交它们了。提交操作会复制本地登出目录中的新文件，然后把它们添加到源代码库中。

进行登入(check in)的最简单的方法是打开项目文件夹，右击 HelloWorld 文件夹，然后选择 SVN Commit。

这将打开一个对话框，其中列出了所有要提交的文件。对话框中还包含一个备注框。添加备注，描述做出了哪些更改，是一个很好的习惯。这里添加了一个新项目，所以可以添加这样一个备注：First commit of hello world project。单击 Commit 按钮，代码将被提交，如图 4-15 所示。

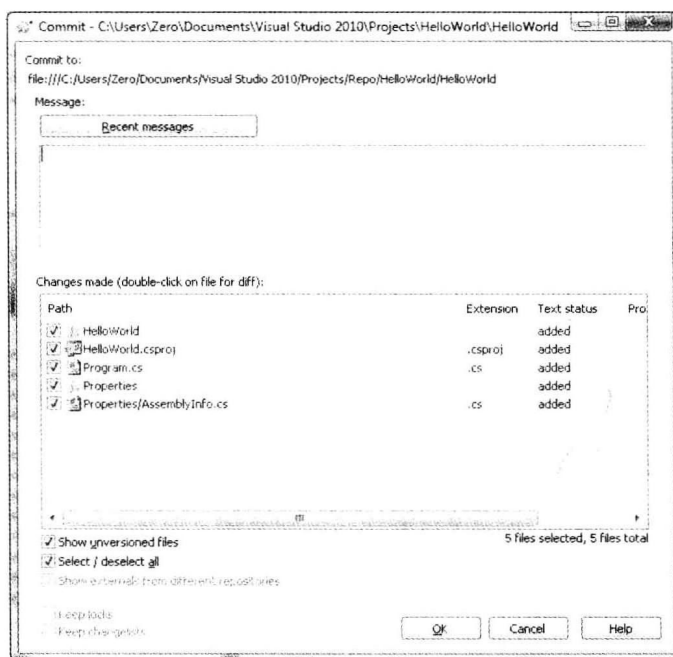


图 4-15 提交源代码

代码现在安全地被存储到库中。接下来就测试这些代码。删除 HelloWorld 文件夹，或者如果你对此操作抱有疑虑，将其重命名为 HelloWorld。然后右击在弹出的快捷菜单中选择 SVN Checkout 命令。HelloWorld 项目应该已被选中。单击 OK 按钮。注意出现了一个新目录 HelloWorld。进入该目录，并双击全新的 HelloWorld.sln 图标。这将启动 Visual Studio。

单击绿色箭头将正确地编译并运行 Hello World 程序。

当在一个团队中工作时，每个人都可以登出项目并进行处理。

4.2.5 历史记录

源代码控制系统的优点之一是回过头来查看项目，了解它的演化过程。右击 Hello World 项目目录，然后在弹出的快捷菜单中选择 SVN Show Log 命令，这将打开历史记录对话框。对话框中显示了每次提交以及提交的备注(见图 4-16)。

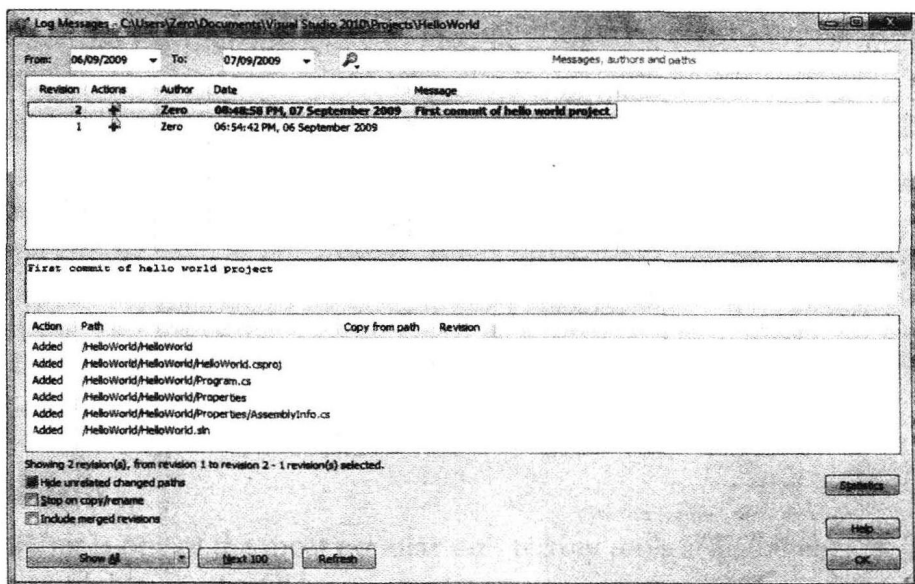


图 4-16 查看历史记录

到目前为止，历史记录视图只显示了两次提交。第一次提交是创建项目时，第二次提交是添加新文件时。可自由尝试选择历史记录选项。单击 statistics 按钮可以显示关于提交的图和信息，以及谁提交了什么。在团队中工作时，这种功能非常有用。

4.2.6 扩展 Hello World

打开 Visual Studio，编辑 Hello World 程序，使其如下所示。

```
public static void Main(string[] args)
{
    System.Console.WriteLine("Let's make some games.'');
    System.Console.ReadKey();
}
```

程序开发得很不错。我们不希望丢失这些更改，所以最好多做一次提交。返回到 Hello World 项目目录，再次选择 SVN Commit。接下来，将会显示修改后的 Program.cs。可添加一个帮助理解的备注，例如 Changed Hello world text。现在代码和源代码控制系统都是最

新的。

在这次提交后，就可以引入 SVN 的另外一个特性了。通过使用 SVN Show Log，再次打开历史记录窗口。单击最近一次提交，即版本号为 3 的提交。在第三个对话框窗口中，可以看到修改后的/HelloWorld/HelloWorld/Program.cs 文件。右击该文件，在弹出的快捷菜单中选择 Show Changes 命令，如图 4-17 所示。

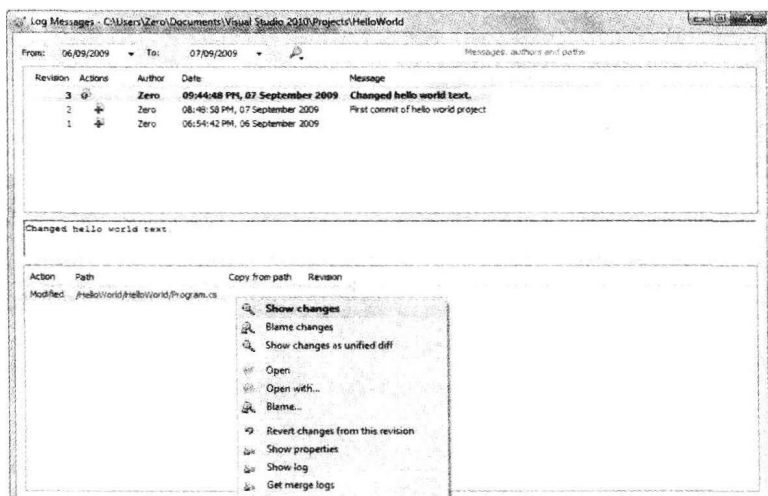


图 4-17 比较更改的方法

这将打开一个叫做 TortoiseMerge 的新程序。TortoiseMerge 将同时显示两个文件，分别是当前提交和前一次提交，如图 4-18 所示。在前一次提交中，Hello World 被线划去，并用红色突出显示，这表明这一部分代码已被修改，文本已被删除。在当前提交中，可以看到它被改为了 Let's make some games。

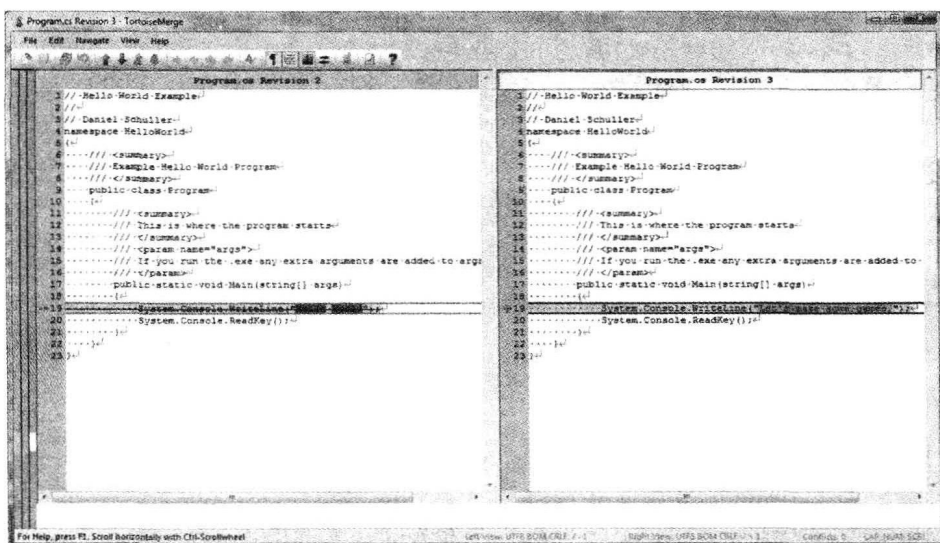


图 4-18 使用合并工具进行比较

对于查看代码的修改过程，这个合并工具非常实用。

4.3 Tao

Tao 允许 C#使用 OpenGL 的功能。安装并把 Tao 添加到游戏项目中非常简单。Tao 的安装程序可以从 sourceforge(<http://sourceforge.net/projects/taoframework>)上下载，本书的配套光盘中也提供了该安装程序。OpenGL 中添加新特性后，Tao 也会被更新，以添加这些新特性。

安装程序完成运行后，不需要采取其他操作。现在就可以在 C#中使用 Tao 提供的库了。

Tao 框架是一个编码项目，这一点与其他项目没什么区别，所以也使用源代码控制系统。事实上，它也使用 SVN。如果想要使用最新的、最前延的 Tao 框架版本，可浏览库，然后右击任意文件夹或者桌面，从弹出的快捷菜单中选择 SVN Repo-browser 命令。在提示后面输入 <https://taoframework.svn.sourceforge.net/svnroot/taoframework/trunk>。短暂等待后，将会显示 Tao 框架的最新代码，如图 4-19 所示。为了登出最新的副本，可采取相同的步骤，但是这一次选择 SVN Checkout 命令而不是 SVN Repo-browser 命令。

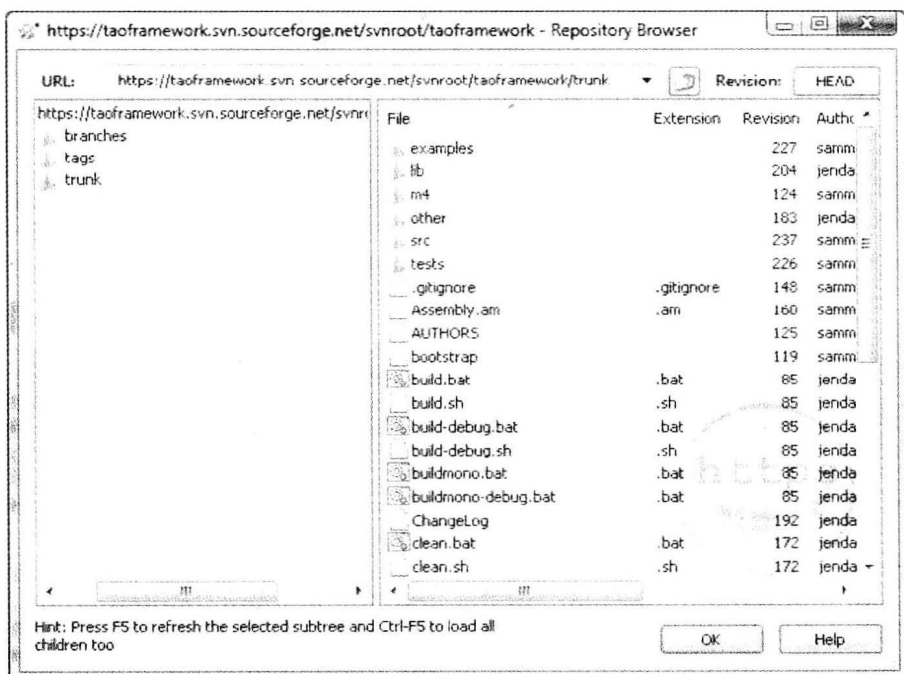


图 4-19 使用 SVN 浏览 Tao 框架

4.4 NUnit

NUnit 是 C#可以使用的最流行的单元测试工具之一。本书的配套光盘中提供了其安装

程序,如果想要使用最新版本,可以访问 <http://www.nunit.com/>。运行安装程序,在看到提示后选择典型安装。

4.4.1 在项目中使用 NUnit

NUnit 使用起来非常简单,但是在编写测试之前,需要有一个项目。在 Visual Studio 中,选择 File|New Project 命令启动一个新项目,如图 4-20 所示。同样,选择 Console Application 项目,将其命名为 PlayerTest。

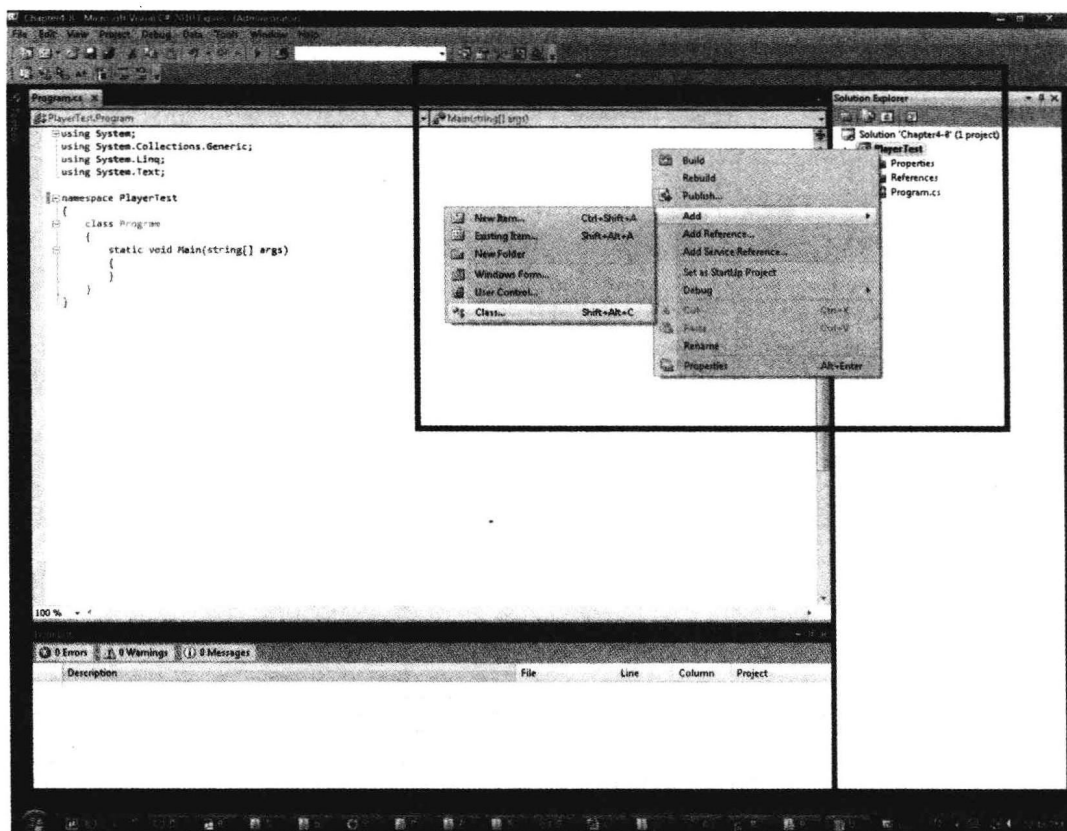


图 4-20 在 Visual Studio 中添加一个类

这将创建一个包含文件 Program.cs 的默认项目。右击该项目,在弹出的快捷菜单中选择 Add|Class 命令,如图 4-20 所示。它将作为 Player 类。

这将打开一个对话框,其中包含许多可供选择的预设类类型,如图 4-21 所示。这里只是需要一个类,所以选择 Class,并将其命名为 Player.cs。此外,还需要一个用于测试的类。采用同样的方式添加另外一个类,将其命名为 TestPlayer.cs。

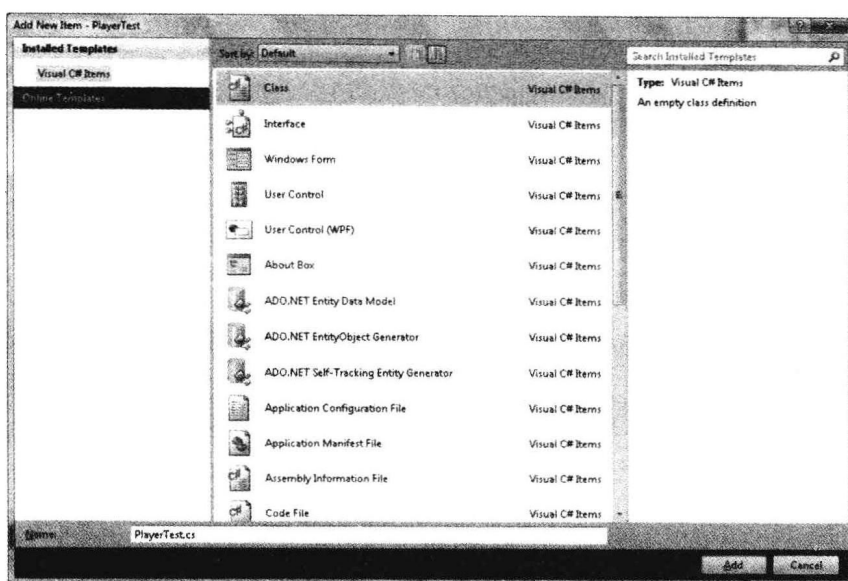


图 4-21 在 Visual Studio 中命名类

为了使用 NUnit, 需要把它作为引用添加到项目中。在 Solution Explorer 中, 右击 References 节点, 然后在弹出的快捷菜单中选择 Add Reference 命令, 如图 4-22 所示。

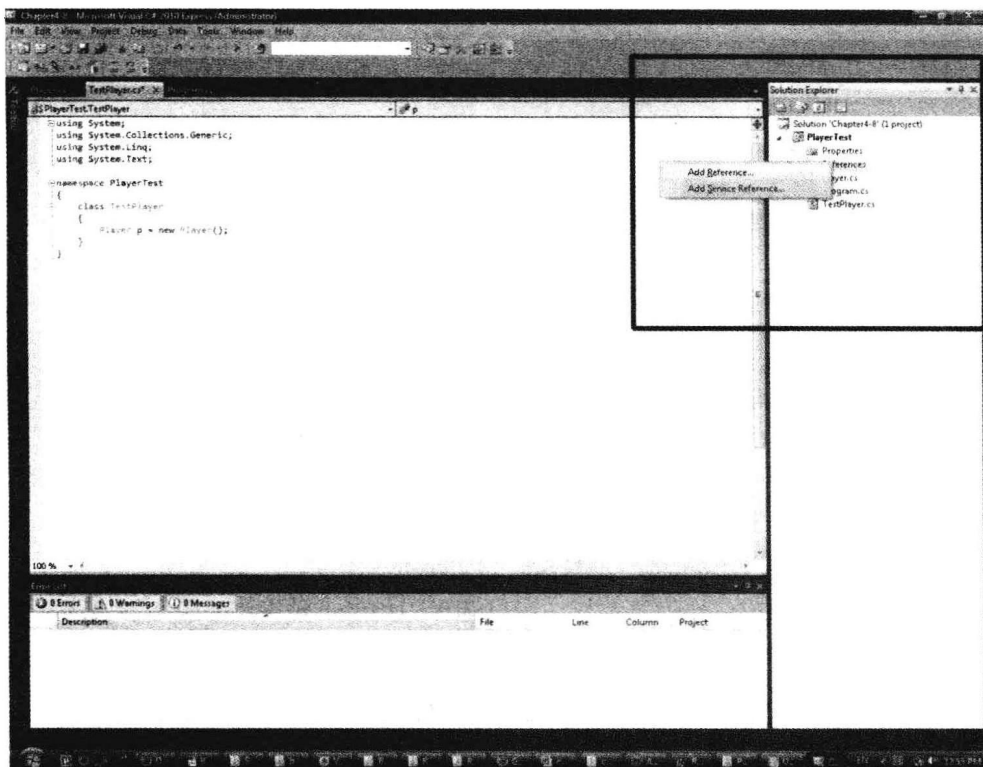


图 4-22 添加一个引用

这将显示一个非常大的引用列表。需要添加的引用是 `JUnit.Framework`。添加引用后，可以通过添加一个 `using` 语句来使用它。在文件顶部的 `TestPlayer` 中，有许多以 `using` 开头的语句。为了使用 `JUnit`，需要添加额外的一行：`using JUnit.Framework;`。然后就可以添加一个简单的测试。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using JUnit.Framework;

namespace PlayerTest
{
    [TestFixture]
    public class TestPlayer
    {
        [Test]
        public void BasicTest()
        {
            Assert.That(true);
        }
    }
}
```

在这个代码段中，首先要注意的是属性。在 C# 中，属性就是一条元数据，也就是可以通过程序访问的类、函数或变量的描述性信息。属性将被放在方括号中。在这里，类的属性为 `TestFixture` 类型。在 `JUnit` 术语中，测试装置(test fixture)是一组相关的单元测试。`TestFixture` 用于告诉 `JUnit` 类 `TestPlayer` 中将有多个测试。

第二个是 `BasicTest` 函数的属性。这是一个 `test` 属性，意味着函数 `BasicTest` 是一个需要运行的测试。编写 `Player` 的代码时，需要向 `TestPlayer` 类中添加更多这种小测试函数。

单击绿色箭头或者按下 `F5` 键来编译代码。一个控制台窗口将快速显示并消失。就现在而言，这种行为没有问题。

4.4.2 运行测试

`JUnit` 有一个专门用于运行测试的 GUI 程序。从 `Start` 菜单中可以找到 `JUnit GUI`。运行以后的 `JUnit GUI` 如图 4-23 所示。

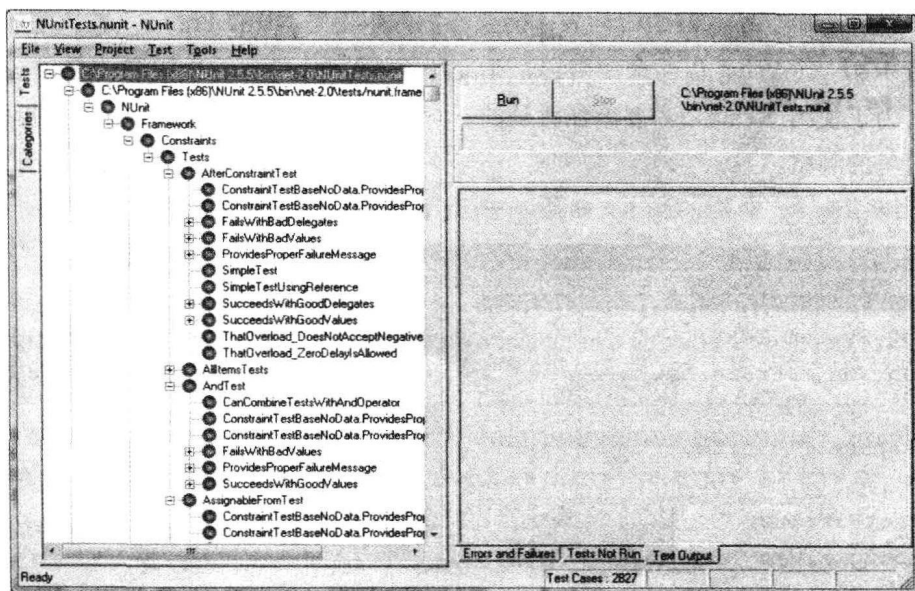


图 4-23 启动后的 NUnit GUI

NUnit 处理编译后的代码。每次单击绿色箭头或按下 F5 键时, Visual Studio 就会编译代码。编译后的代码存储在项目目录中(通常是 `Projects\PlayerTest\PlayerTest\bin(Debug/Release)`)。项目目录默认存储在 My Documents 下的 Visual Studio 2010 目录中。

在这个目录中可以找到一个 exe 文件, 这是代码的编译版本。目录中还有一个 nunit.framework.dll 文件, 这是前面包含的 NUnit 引用。如果没有这个 dll, exe 文件将无法运行。

为了运行测试, 在 NUnit GUI 中选择 File|Open Project 命令。找到发布目录中的编译代码。单击 `PlaerTest.exe` 文件。这将会加载前面在 NUnit 中编写的测试, 如图 4-24 所示。

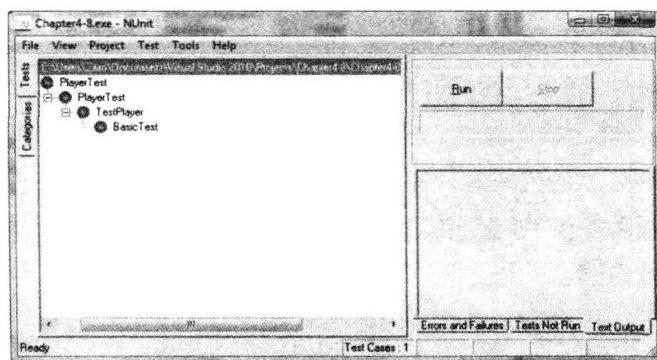


图 4-24 加载项目的 NUnit GUI

单击 Run 按钮, 项目树将变成绿色, BasicTest 旁边将出现一个绿色对勾。这意味着测试正确运行。如果修改代码使测试失败, 绿色小圆圈会变成红色小圆圈。试着修改代码, 使其失败, 然后再次运行测试。

4.4.3 示例项目

游戏中的玩家类需要表达这个概念：如果玩家吃掉一个蘑菇，他将变大。假设所有的玩家一开始都很小。这就是要执行的第一个测试。

```
namespace PlayerTest
{
    [TestFixture]
    public class TestPlayer
    {
        [Test]
        public void BasicTest()
        {
            Assert.That(true);
        }

        [Test]
        public void StartsLifeSmall()
        {
            Player player = new Player();
            Assert.False(player.IsEnlarged());
        }
    }
}
```

Assert 类是 **NUnit** 的一部分，并且是编写单元测试类的主力。函数 **IsEnlarged** 现在还不存在。使用重构工具时， **Visual Studio** 可以自动创建这个函数。这将在 **Player** 类中得到如下代码。

```
class Player
{
    internal bool IsEnlarged()
    {
        throw new NotImplementedException();
    }
}
```

现在在 **NUnit** 中运行单元测试将导致失败，产生的错误消息如下：

```
PlayerTest.TestPlayer.StartsLifeSmall:
System.NotImplementedException : The method or operation is not
implemented.
```

测试失败的原因是 **Player** 类中还有一个方法未曾定义。通过添加下面的功能可以解决

这个问题。

```
class Player
{
    bool _enlarged = false;
    internal bool IsEnlarged()
    {
        return _enlarged;
    }
}
```

现在就将通过测试。生成项目，再次运行 NUnit，项目旁边将显示一个绿色的圆圈。下一个测试是吃蘑菇测试。

```
namespace PlayerTest
{
    [TestFixture]
    public class TestPlayer
    {
        [Test]
        public void BasicTest()
        {
            Assert.That(true);
        }

        [Test]
        public void StartsLifeSmall()
        {
            Player player = new Player();
            Assert.False(player.IsEnlarged());
        }

        [Test]
        public void MushroomEnlargesPlayer()
        {
            Player player = new Player();
            player.Eat("mushroom");
            Assert.True(player.IsEnlarged());
        }
    }
}
```

这个新测试与前一个测试非常类似。生成的测试可以很好地描述项目。新接触代码的人可以阅读与某个函数相关的全部测试，从而比较准确地理解该函数的功能。

与之前的测试类似，这个测试中添加了新功能。**Player** 类现在还没有 **Eat** 方法。与前例相同，**Visual Studio** 的重构工具可以快速生成所需的方法。

```
class Player
{
    bool _enlarged = false;
    internal bool IsEnlarged()
    {
        return _enlarged;
    }

    internal void Eat(string p)
    {
        throw new NotImplementedException();
    }
}
```

如果现在运行 **NUnit**，新测试将会失败。使用如下代码可以通过测试。

```
class Player
{
    bool _enlarged = false;
    internal bool IsEnlarged()
    {
        return _enlarged;
    }

    internal void Eat(string thingToEat)
    {
        if (thingToEat == "mushroom")
        {
            _enlarged = true;
        }
    }
}
```

所有的测试都已通过，证明代码正确且符合预期。关于单元测试的内容基本上就是这些。这种技术不是特别复杂，但是可以让程序员相信代码的正确性，并且使它们可以在不破坏功能的情况下修改代码。

4.5 小结

Visual Studio Express 是开发 C#程序的最好的方法之一，它是免费的，但是特性却非常丰富。Visual Studio Express 是一个完整的 IDE，支持编写、编译和调试 C#程序。它提供了许多有用的功能来重构已有代码和生成新代码，并且还有许多可以加速代码编写速度的快捷键。源代码控制是一种记录代码中的所有更改，并将源代码放在一个位置的方法。SVN 是一个优秀的源代码控制程序，它有一个易于使用的 Windows 包装器，叫做 TortoiseSVN。

对小段代码进行单元测试可以确认程序的各段代码可以正确工作。NUnit 是用于 C#的一个单元测试程序，提供了为代码编写测试的接口。编写测试后，NUnit 中有一个以可视化方式显示并运行全部测试的程序，它会在通过的测试旁边显示一个绿色对勾，在失败的测试旁边显示一个红色的叉号。这 3 个程序是进行 C#开发的一个极佳的起点。

第 5 章

游戏循环和图形

计算机游戏分为许多类型，有抽象的益智游戏，如《俄罗斯方块(Tetris)》；有回合制策略游戏，如《文明(Civilization)》；也有快节奏的第一人称射击游戏，如《半条命(Half-Life)》。所有这些游戏，乃至所有的计算机游戏，都是以相同的方式编程的。

5.1 游戏的工作方式

游戏与玩家进行通信的最重要的方式是通过电视屏幕或者计算机显示器。在游戏中经常听到帧率(frame-rate)这个词。较好的帧率应该保持在 30 帧/s~60 帧/s 之间。那么在进行游戏编程时，帧率到底指的是什么？

帧是指两次屏幕更新之间相隔的时间。计算机程序负责至少 30 帧/s 使用新信息更新屏幕。计算机运行得非常快，所以以这种速度更新屏幕没有问题。

计算机需要多久的时间来更新每个帧？如果最低要求是 30 帧/s，那么每帧需要用 33ms。计算机有 33ms 的时间“思索”应该在下一帧中显示哪些内容。计算机可以在 33ms 内完成大量的计算，比大多数人在一周的时间内完成的计算量还多。

所有的游戏都有一个游戏循环，这是游戏中最主要的循环。在游戏运行期间，将不断地重复调用这个循环。游戏循环有 3 个主要的阶段：它是获取输入(如游戏手柄或键盘)的状态，更新游戏世界的状态，最后是更新屏幕上的所有像素。

游戏循环详解

游戏代码不只负责更新屏幕上的图像，还有其他两个非常重要的任务。它必须获取用户输入，例如“如果用户按下 B 按键，角色将会跳跃”；还必须更新游戏世界，例如“如

果玩家杀死了大 boss，则显示致谢界面。”

所有游戏循环都有类似的模式。

```
while (true)
{
    // Find out what state keyboard and joypads are in
    UpdateInput();
    // Handle input, update the game world, move characters etc
    Process();
    // Draw the current state of the game to the screen
    Render();
}
```

这就是游戏循环的 3 个主要阶段：更新用户输入，更新游戏世界，然后告诉图形卡渲染哪些内容。

5.2 使用 C#实现一个快速的游戏循环

打开 Visual Studio。Visual Studio 默认假定程序员将开发一个由事件驱动的软件，而不是一个持续执行代码的软件(例如游戏)。事件驱动的程序响应操作系统事件或者用户输入事件，并执行相应的代码。使用一个可以持续更新游戏世界的状态的主循环来编写游戏更加简单。需要对默认的代码做一些修改，以便实现一个快速的游戏循环。这个循环应该尽可能多地运行，这一点非常重要，所以主循环的 C#代码使用了一些 C 函数来确保它是最快的游戏循环。

新建一个 Windows Form Application，并将项目命名为 GameLoop。一个包含如下代码的 Program.cs 文件将自动生成：

```
namespace GameLoop
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
    }
}
```


运行这段代码将只显示一个标准的 Windows 窗体。现在，它还是一个事件驱动的应用程序。如果它持续地执行，则应有如下所示的程序：

```
namespace GameLoop
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new Form1());
        }
        static void GameLoop()
        {
            // GameCode goes here
            // GetInput
            // Process
            // Render
        }
    }
}
```

在每一帧中都应该调用 `GameLoop` 函数。为此，需要创建一个新类。在 Solution Explorer 选项卡中右击 `GameLoop` 项目，然后在弹出的快捷菜单中选择 `Add|Class` 命令。此时将提示输入类的名称，将该类命名为 `FastLoop`。`FastLoop` 类将由 `Program` 类使用。它会强制每帧调用一次 `GameLoop` 函数。在编写 `GameLoop` 之前，让我们先看看它的使用方式。

```
static class Program
{
    static FastLoop _fastLoop = new FastLoop(GameLoop);
    /// <summary>
    /// The main entry point for the application.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new Form1());
    }
}
```

```

    }
    static void GameLoop()
    {
        // GameCode goes here
        // GetInput
        // Process
        // Render
    }
}

```

FastLoop 类只接受一个参数，即函数 **GameLoop**。转换项目时需要该函数，以便项目可以持续执行。

打开 **FastLoop.cs**。在 **Program** 类中，**FastLoop** 构造函数接受对 **GameLoop** 函数的引用。因此 **FastLoop** 类必须定义它的构造函数，以便让它接受一个函数作为参数。

```

public class FastLoop
{
    public delegate void LoopCallback();
    public FastLoop(LoopCallback callback)
    {
    }
}

```

在 C# 中，通过使用定义了函数签名(函数的返回值和接受的参数)的委托，可以把函数存储为变量。这里使用了一个委托 **LoopCallback** 来定义没有返回值或者参数的函数类型。**FastLoop** 有一个接受一个回调的构造函数。回调函数将在每一帧中调用，以便允许游戏更新自身。

所有 C# 窗体程序都有一个叫做 **Application** 的静态类，该类代表程序和程序的设置。它也可以用来修改一个程序，使其可以实时使用。

程序有大量需要处理的事件。窗体可能最大化，也可能被关闭了。**Application** 是处理所有这些事件的代码部分。当没有要处理的事件时，它会调用 **Application.Idle**，意味着它将进入空闲状态。当 **Application** 不忙时，应该在这段空闲时间更新游戏逻辑。为了使用 **Application** 代码，需要在文件的顶部附近添加 **using System.Windows.Forms**。

```

using System.Windows.Forms;
namespace GameLoop
{
    public class FastLoop
    {
        public delegate void LoopCallback();
        LoopCallback _callback;

        public FastLoop(LoopCallback callback)
    }
}

```

```

    {
        _callback = callback;
        Application.Idle += new EventHandler(OnApplicationEnterIdle);
    }
    void OnApplicationEnterIdle(object sender, EventArgs e)
    {
    }
}
}

```

这里，游戏循环回调被存储在成员变量 `_callback` 中供以后使用。`Application.Idle` 事件的处理程序为 `OnApplicationEnterIdle`。当应用程序开始空闲时，该处理程序就会被调用。下一部分稍微有些棘手。代码需要知道 `Application` 是否仍然处于空闲状态，如果是，它需要持续调用循环回调。阅读下面的代码，了解回调是如何被调用的。

```

void OnApplicationEnterIdle(object sender, EventArgs e)
{
    while (IsAppStillIdle())
    {
        _callback();
    }
}
private bool IsAppStillIdle()
{
    Message msg;
    return !PeekMessage(out msg, IntPtr.Zero, 0, 0, 0);
}

```

除非 `IsAppStillIdle` 返回 `false`，否则代码会不断地调用回调。`IsAppStillIdle` 使用一个新的 `PeekMessage` 函数来检查应用程序是否有需要处理的重要事件。如果应用程序有需要处理的重要消息，那么在返回到游戏循环之前必须处理这些消息。

为了检查应用程序是否空闲，需要检查 Windows 的事件队列。Windows 有一个庞大的事件队列。移动窗口、最小化窗口或者按下键盘上的一个键，都会在事件队列中加入一个事件。许多不同的动作都会产生事件。所有的窗体都需要处理自己的事件。如果应用程序的事件队列中有一些事件，那么就需要退出空闲状态并处理这些事件。

使用一些 Windows C 函数检查事件队列是最容易的方法。在 C# 中，这被叫做 `InterOp`，是 `InterOperation` (互操作) 的缩写。在 C# 程序中使用一些 C 函数实际上很简单。为了查看事件队列的内容，我们将使用一个 C 函数 `PeekMessage`。这里只是想“窥视”一下队列，看其中是不是有事件。`PeekMessage` 的 C 定义如下所示。

```

BOOL PeekMessage(
    LPMSG lpMsg,
    HWND hWnd,

```

```

        UINT wMsgFilterMin,
        UINT wMsgFilterMax,
        UINT wRemoveMsg
    );

```

根据文档的描述可以知道：

- 如果存在需要处理的消息，则返回值为非零。
- 如果不存在需要处理的消息，则返回值为零。

这是判断应用程序中是否有等待处理的事件的绝佳函数。

理解这个 C 函数的全部细节并不重要。重要的是知道如何从 C# 中调用该函数。为此，需要将函数的参数从 C 类型修改为对应的 C# 类型。

第一个参数 `lpMsg` 是一个消息类型。C# 中没有提供这个类型，所以需要导入它。第二个参数是一个 **Windows** 句柄，它是对窗体的一个引用。最后三个参数都是无符号整型数，这是标准的 C# 类型。下面给出 C 的消息结构，它就是 `PeekMessage` 的第一个参数。

```

typedef struct {
    HWND hwnd;
    UINT message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD time;
    POINT pt;
} MSG, *PMSG;

```

结构的成员不太重要。将这个 C 类型导入 C# 的方法如下。首先在 `FastLoop.cs` 的顶部包含 `using System.Runtime.InteropServices;`。这个库中包含了用于导入 C 类型和结构的实用函数。

```

using System.Runtime.InteropServices;
using System.Windows.Forms;
namespace GameLoop
{
    [StructLayout(LayoutKind.Sequential)]
    public struct Message
    {
        public IntPtr hWnd;
        public Int32 msg;
        public IntPtr wParam;
        public IntPtr lParam;
        public uint time;
        public System.Drawing.Point p;
    }
}

```

添加这个 C 结构只需要将 C 类型和 C# 类型正确地对应起来。特性 `[StructLayout`

(LayoutKind.Sequential)]告诉 C#按照编写结构的方式在内存中布局这个结构。如果没有这个特性, C#可能会尝试重新调整结构, 使其高效地使用内存。C 则预期结构按照编写方式在内存中布局。

现在导入消息类型, 接下来还需要导入 `PeekMessage` 函数。

```
public class FastLoop
{
    [System.Security.SuppressUnmanagedCodeSecurity]
    [DllImport("User32.dll", CharSet = CharSet.Auto)]
    public static extern bool PeekMessage(
        out Message msg,
        IntPtr hWnd,
        uint messageFilterMin,
        uint messageFilterMax,
        uint flags);
}
```

第一个特性[System.Security.SuppressUnmanagedCodeSecurity]表示: “我们在调用 C 这个非托管语言, 所以不要进行任何托管安全检查”。第二个特性[DllImport("User32.dll", CharSet = charSet.Auto)]引用包含要导入的 C 函数的 DLL 文件。User32.dll 是使用 C 与 Windows 操作系统进行交互的主要的文件之一。PeekMessage 函数将填充 Message 结构, 所以必须能够写入该结构。第一个参数前面的 out 关键字用于实现该功能。其他的参数则可以忽略, 因为不会向它们传递有用的信息。

在定义 PeekMessage 后, 可以使用它来确定应用程序的事件队列中是否有等待处理的事件。

```
private bool IsAppStillIdle()
{
    Message msg;
    return !PeekMessage(out msg, IntPtr.Zero, 0, 0, 0);
}
```

正确定义 IsAppStillIdle 后, 程序就有了一个极快的游戏循环。在将来制作的任何游戏项目中都可以重用 FastLoop。

为了测试游戏循环确实可以工作, 可打开 Project.cs, 添加下面的代码行。

```
static void GameLoop()
{
    // GameCode goes here
    // GetInput
    // Process
    // Render
    System.Console.WriteLine("loop");
}
```

```
}
```

每次调用游戏循环后，它将向控制台输出“loop”一词。为了查看控制台，可选择 Debug|Windows|Output 命令，然后将弹出另外一个窗口。运行应用程序，在输出窗口中，词“loop”将在屏幕中不断地向下滚动。

精确的时间调配

为了使对象流畅地运动，知道两帧之间经过了多长时间非常重要。然后可以使用这个时间使动画独立于帧率。游戏应该在所有计算机上以相同的速度运行，不能让游戏角色在 CPU 速度更快的计算机上跑得更快。

计算机游戏中的时间调配非常重要。帧之间的时间必须精确且具有高分辨率，否则游戏将显得动动停停。为了得到最佳的时间调配函数，需要使用一些 C 代码。比起前面使用 InterOp 来查看 Windows 消息，这一点其实没有那么困难。帧之间的时间通常叫做 `elapsedTime` 或 `delta` 时间，有时候简写为 `dt`，它可以精确到 1/10 秒。

创建一个新类，将其命名为 `PreciseTimer`。这个类并不大，所以下面列出了它的全部代码。阅读这段代码，了解该类的作用。

```
using System.Runtime.InteropServices;
namespace GameLoop
{
    public class PreciseTimer
    {
        [System.Security.SuppressUnmanagedCodeSecurity]
        [DllImport("kernel32")]
        private static extern bool QueryPerformanceFrequency(ref long
PerformanceFrequency);
        [System.Security.SuppressUnmanagedCodeSecurity]
        [DllImport("kernel32")]
        private static extern bool QueryPerformanceCounter(ref long
PerformanceCount);
        long _ticksPerSecond = 0;
        long _previousElapsedTime = 0;
        public PreciseTimer()
        {
            QueryPerformanceFrequency(ref _ticksPerSecond);
            GetElapsedTime(); // Get rid of first rubbish result
        }
        public double GetElapsedTime()
        {
            long time = 0;
```

```

        QueryPerformanceCounter(ref time);
        double elapsedTime = (double)(time - _previousElapsedTime) /
(double)_ticksPerSecond;
        _previousElapsedTime = time;
        return elapsedTime;
    }
}
}

```

QueryPerformanceFrequency 函数用于获取高分辨率性能计数器的频率。多数现代硬件都有一个高分辨率计时器，这个函数用于获取计时器增长的频率。**QueryPerformanceCounter** 函数用于获取高分辨率性能计数器的当前值。可以结合使用这两个函数来确定最后一帧用了多长时间。

GetElapsedTime 应该每帧调用一次，它会记录时间。帧之间经过的时间非常重要，所以应该把它整合到游戏循环中。打开 **Program.cs** 文件，修改游戏循环，使其只接受一个参数。

```

static void GameLoop(double elapsedTime)
{
    // GameCode goes here
    // GetInput
    // Process
    // Render
    System.Console.WriteLine("loop");
}

```

还需要修改 **FastLoop.cs**。委托必须接受帧之间经过的时间作为参数，而且还需要添加一个 **PreciseTimer** 作为其成员。

```

public class FastLoop
{
    PreciseTimer _timer = new PreciseTimer();
    public delegate void LoopCallback(double elapsedTime);
}

```

之后，定时器将每帧调用一次，帧之间经过的时间将被传递给 **FastLoop** 的回调。

```

void OnApplicationEnterIdle(object sender, EventArgs e)
{
    while (IsAppStillIdle())
    {
        _callback(_timer.GetElapsedTime());
    }
}

```

现在就可以使用这个游戏循环来使游戏流畅地进行。在本章结束时，这个游戏循环将

用于平滑地旋转 3D 三角形，这是 OpenGL 编程中的 Hello World 应用程序。

5.3 图形

现在有了可以工作的游戏循环，下一步是在屏幕上显示一些东西。可以使用 `FastLoop.cs` 和 `GameLoop.cs` 继续开发当前的项目，也可以创建一个新项目，并修改其代码，使其包含一个快速游戏循环。为了显示图像，必须在项目中包含 Tao 库。如果还没有安装 Tao 框架，现在就是一个进行安装的好机会。

在 Visual Studio 中，找到 Solution Explorer 窗口。其中应该只包含一个项目。展开该项目，可以看到 References 图标。右击该图标，然后在弹出的快捷菜单中选择 Add Reference 命令。

这将打开引用对话框。单击 Browser 选项卡，找到 Tao 框架的安装目录。在 Windows 7 和 Windows Vista 上，这个路径是 `C:\Program Files (x86)\TaoFramework\bin`。在 Windows XP 上，这个路径是 `C:\Program Files\TaoFramework\bin`。在找到正确的目录后，得到的界面如图 5-1 所示。选择 `Tao.OpenGL.dll` 和 `Tao.Platform.Windows.dll`，然后单击 OK 按钮。Tao 框架默认提供一个叫做 `SimpleOpenGLControl` 的控件，允许 OpenGL 在 Windows 窗体中进行渲染。为了启用该控件，双击 Solution Explorer 中的窗体。这样将会打开如图 5-2 所示的窗体设计器。

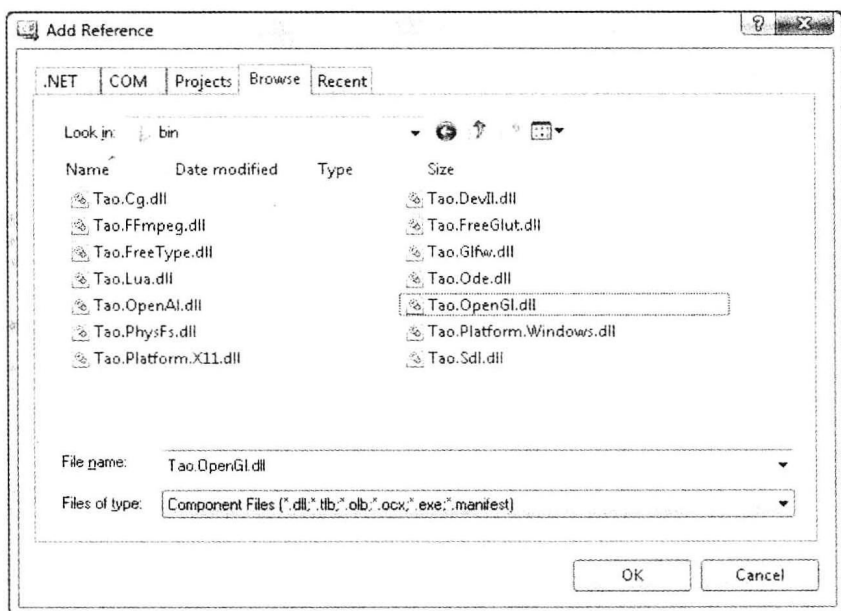


图 5-1 引用列表

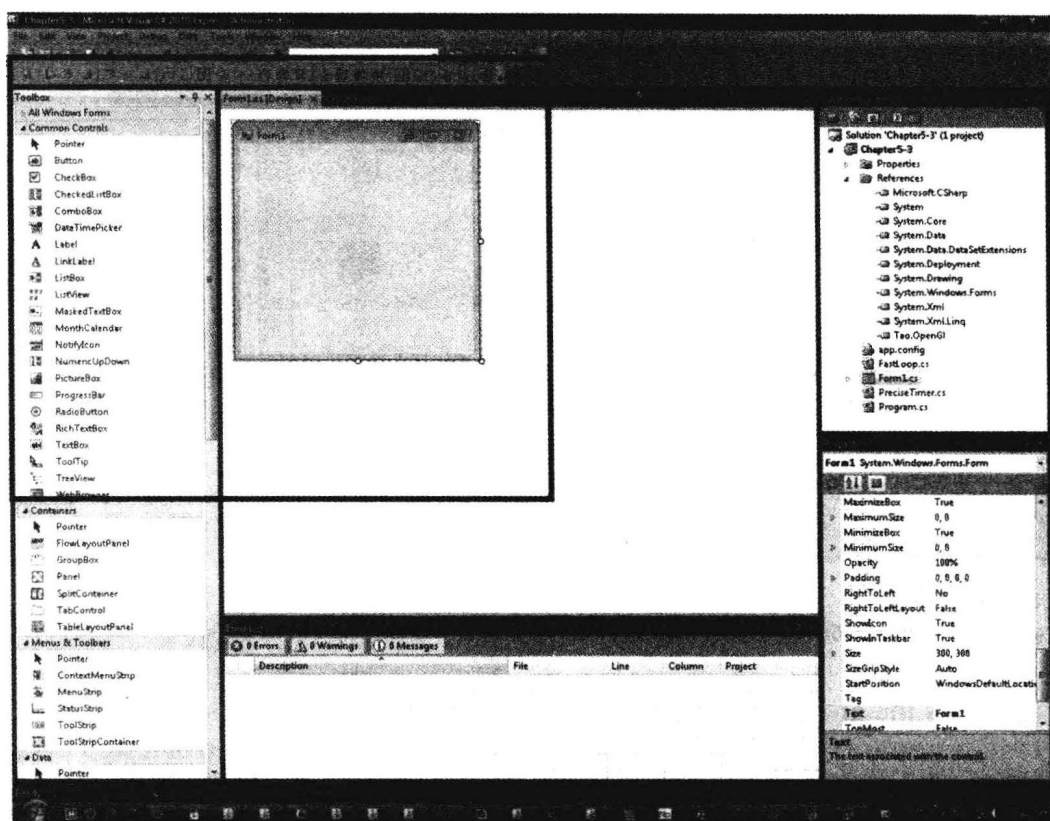


图 5-2 窗体设计器

右击工具栏面板，在弹出的快捷菜单中选择 Choose Items 命令，打开的对话框如图 5-3 所示。

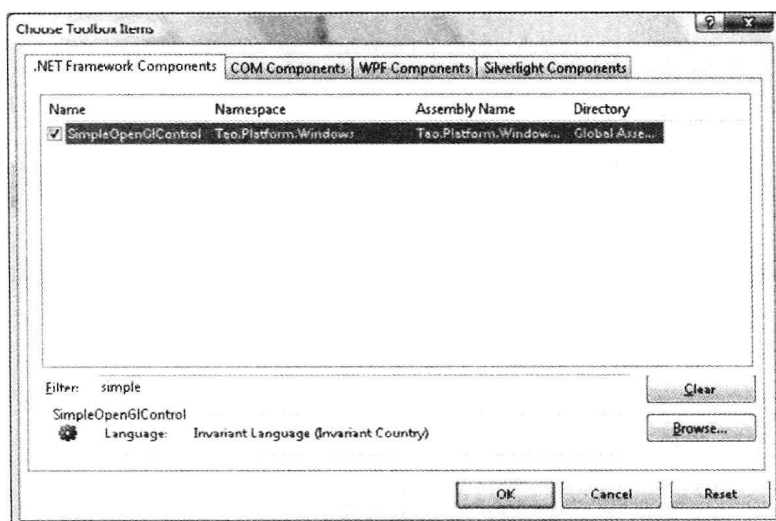


图 5-3 Choose Toolbox Items 对话框

Tao 安装程序在对话框中添加了一个 SimpleOpenGLControl 选项。如果没有 Simple OpenGLControl, 则单击 Browse 按钮, 找到 Tao 框架的二进制文件目录(用于添加引用的同一个目录), 选择 Tao.Platform.Windows.dll 文件。如图 5-3 所示, 选中复选框, 然后单击 OK 按钮。现在就在 Toolbox 中添加了一个新控件 SimpleOpenGLControl。在窗体设计器中, 将这个新控件从 Toolbox 中拖动到窗体上。现在的窗体如图 5-4 所示。

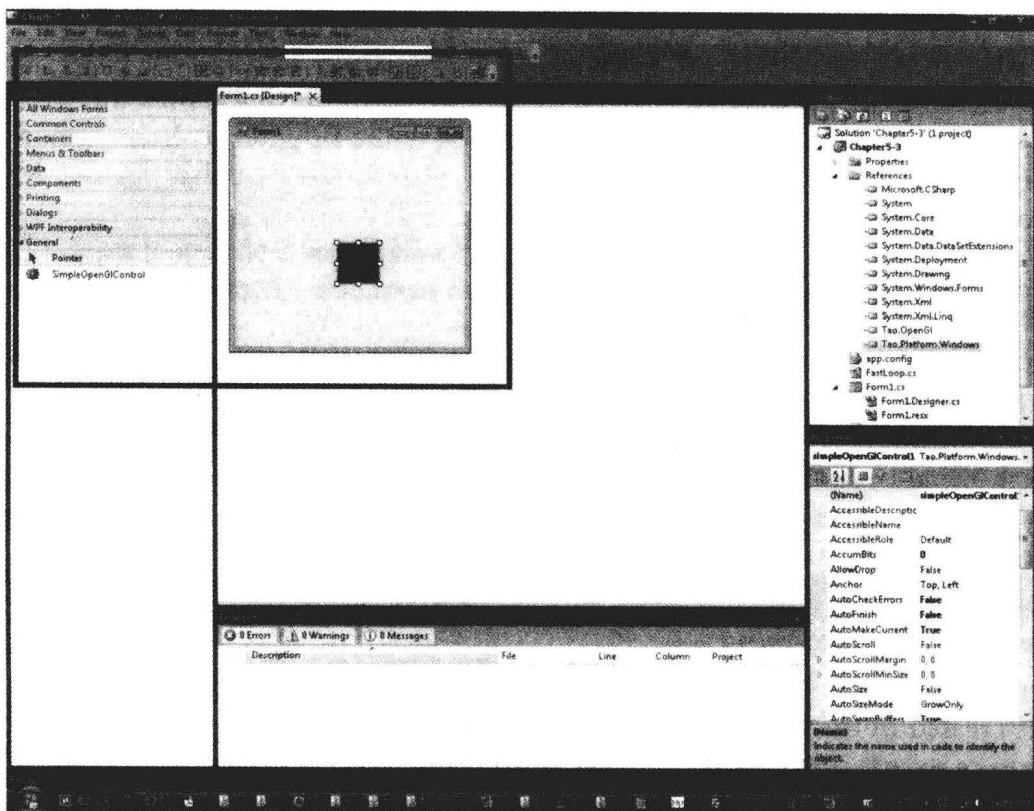


图 5-4 添加 SimpleOpenGLControl

窗体中的黑色小窗口是 OpenGL 进行渲染的地方。现在这个窗口还有点小。为使它与窗体具有相同的大小, 右击该控件, 然后在弹出的快捷菜单中选择 Properties 命令。在打开的 Properties 窗口中, 找到属性 Dock, 将其设置为 Fill。这会使 OpenGLControl 填充控件中的所有空间。

如果单击绿色箭头按钮, 现在会显示一个错误, 指出 “No device or rendering context available.”。这是因为 OpenGL 控件还没有被初始化。

在代码视图中打开窗体, 添加下面的代码。

```
public Form1()
{
    InitializeComponent();
    simpleOpenGLControl1.InitializeContexts();
}
```

运行程序后，将创建一个带有黑色屏幕的窗体。这就是在 C# 中使用 OpenGL 的结果。变量 `_openGLControl` 比 `simpleOpenGLControl` 更易读，可以使用重构工具完成重命名工作。

5.3.1 全屏模式

大多数游戏允许玩家在全屏模式下玩游戏。添加这个选项的代码十分容易，如下所示。

```
bool _fullscreen = true;
public Form1()
{
    InitializeComponent();
    _openGLControl.InitializeContexts();
    if (_fullscreen)
    {
        FormBorderStyle = FormBorderStyle.None;
        WindowState = FormWindowState.Maximized;
    }
}
```

边框样式设置窗口外的部分，包括菜单栏和边框。去除这些部分后，窗体的大小将直接反映 OpenGL 控件的大小。窗体可以处于 3 个窗口状态：`Normal`、`Minimized` 和 `Maximized`。`Maximized` 可以提供我们需要的全屏模式。全屏模式适合玩游戏，但是在开发游戏时，窗口模式要更好一些。如果程序位于一个窗口中，那么在游戏运行的同时就可以使用调试器。因此，可能最好将 `_fullscreen` 设置为 `false`。

5.3.2 渲染

为了制作游戏，需要学习如何让 OpenGL 在屏幕上进行绘制。在学习 OpenGL 时，享受乐趣、保持好奇并尝试 API 是十分重要的。不要害怕尝试会破坏代码。OpenGL 的网站上提供了所有 OpenGL 函数的文档，网址为 <http://www.opengl.org/sdk/docs/man/>。这是研究 OpenGL 库的一个很好的网站。

1. 清除背景

使用 OpenGL 渲染图像首先要求使用一个游戏循环。之前我们曾在 `Program.cs` 类中创建游戏，但是这一次将在 `Form.cs` 中创建游戏循环，以便它可以访问 `openGLControl`。

```
using Tao.OpenGl;
namespace StartingGraphics
{
    public partial class Form1 : Form
    {
        FastLoop _fastLoop;
        bool _fullscreen = false;
```

```
public Form1()
{
    _fastLoop = new FastLoop(GameLoop);
    InitializeComponent();
    _openGLControl.InitializeContexts();
    if (_fullscreen)
    {
        FormBorderStyle = FormBorderStyle.None;
        WindowState = FormWindowState.Maximized;
    }
}

void GameLoop(double elapsedTime)
{
    _openGLControl.Refresh();
}
}
```

这个游戏循环的代码与之前编写的代码十分类似。记得添加 `using Tao.OpenGL;` 语句，否则游戏循环将无法使用 OpenGL 库。

`GameLoop` 将在每帧中调用。现在它只是刷新 OpenGL 控件。刷新调用告诉 OpenGL 控件更新其图像。运行这个程序将显示一个黑色屏幕，但是现在已经建立了游戏循环，可以改变屏幕了。

```
void GameLoop(double elapsedTime)
{
    Gl.glClearColor(1.0f, 0.0f, 0.0f, 1.0f);
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    Gl.glFinish();
    _openGLControl.Refresh();
}
```

游戏循环中新添加了3行代码。第一行告诉 OpenGL 使用哪一种颜色清除屏幕。OpenGL 使用4个值表示颜色：红、绿、蓝和 `alpha`。`alpha` 决定颜色的透明度：1 表示完全不透明，0 表示完全透明。每个值的取值范围为 0~1。这里将红色设为 1，将绿色和蓝色设为 0，将 `alpha` 也设为 1。得到的结果是亮红色。清除色的 `alpha` 将被忽略。清除色只需要设置一次，但是为了将全部新代码放在一块，现在在每一帧中都设置了清除色。

第二行代码发出清除命令。这个函数使用前一行设置的清除色来清除屏幕。最后一行的 `glFinish` 告诉 OpenGL 对这一帧的处理已经完成，并让 OpenGL 确认所有命令都得到了执行。

运行这个程序后，窗口将显示新的亮红色背景。自由进行试验，使自己熟悉这个库。尝试交换命令或者修改颜色。例如，查看下面这段代码，在运行它之前猜测它将完成什么功能。

```
Random r = new Random();
Gl.glClearColor((float)r.NextDouble(), (float)r.NextDouble(), (float)
r.NextDouble(), 1.0f);
```

2. 顶点

顶点是组成虚拟世界的基础。最基本的顶点包含位置信息，即二维世界中的 x 和 y ，三维世界中的 x 、 y 和 z 。

使用立即模式在 OpenGL 中引入顶点很容易。立即模式是告诉图形卡绘制什么内容的一种方式。即使什么都没有改变，每帧中仍然都需要发送这些命令。这不是进行 OpenGL 编程最简单的方法，但却是最容易学习的方法。

下面首先绘制一个点。

```
void GameLoop(double elapsedTime)
{
    Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    Gl.glBegin(Gl.GL_POINTS);
    {
        Gl.glVertex3d(0, 0, 0);
    }
    Gl.glEnd();
    Gl.glFinish();
    _openGLControl.Refresh();
}
```

这里有 3 个新的 OpenGL 命令。glBegin 告诉图形卡你想要绘制某些东西，它接受的参数描述了要绘制的对象。这里传入的值为 GL_POINTS，告诉 OpenGL 将任何顶点都渲染为点（而不是三角形或者四边形）。

glBegin 的后面必须有一个 glEnd。紧跟在 glBegin 之后则是一个大括号。这个大括号提供了缩进，使得在哪里放置 glEnd 变得更加明显，但并不是必须使用该大括号。在大括号之间是一个以 3d 结尾的 glVertex 命令。3 意味着它将使用 3 个维度(x , y , z)来确定位置。d 意味着位置的类型为 double。

在开始和结束调用之间可以有任意多的顶点。当前的顶点在位置(0, 0, 0)处绘制，也就是屏幕的正中央。当前的场景是根据 OpenGL 的默认设置建立的。图 5-5 显示的立方体描述了 OpenGL 场景的默认设置。可以在这个立方体内的任意位置渲染点，但是如果任意一个坐标值小于-1 或者大于 1，点将位于场景之外，因而是不可见的。可以认为这个场景的摄像机位于 z 轴上的位置(0, 0, 1)，朝向-1 的方向。

位置(0, 0, 1)通常叫做场景的原点。绘制顶点后，关闭大括号，然后进行结束调用。运行程序后，可以在屏幕的中央看到一个白色的像素。这就是刚才绘制的点。现在这个点很小，但是在 OpenGL 中很容易调整绘制的点的大小。在 glBegin 语句之前添加 Gl.glPointSize(5.0f);。现在就更容易看到这个点了。

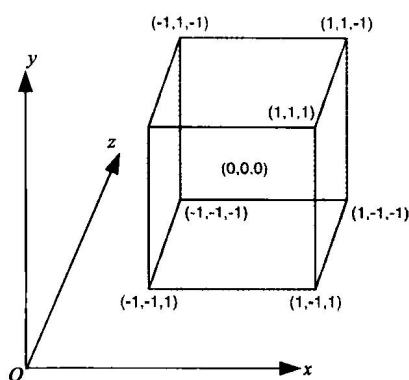


图 5-5 默认的 OpenGL 场景

3. 三角形

OpenGL 支持许多种由顶点构成的图元类型。大部分游戏都使用三角形带来表示游戏中的对象。FPS 或者对打游戏中的角色都只不过是一个顶点的列表，这些顶点可以连接起来构成一个三角形列表。

```
Gl.glBegin(Gl.GL_POINTS);
{
    Gl.glVertex3d(-0.5, 0, 0);
    Gl.glVertex3d(0.5, 0, 0);
    Gl.glVertex3d(0, 0.5, 0);
}
Gl.glEnd();
```

运行上面的代码片段，它将绘制三角形的 3 个顶点。为了实际绘制一个完整的三角形，传入 `glBegin` 的参数需要由 `GL_POINTS` 改为 `GL_TRIANGLES`。在修改代码后重新运行程序，可以看到如图 5-6 所示的白色三角形。

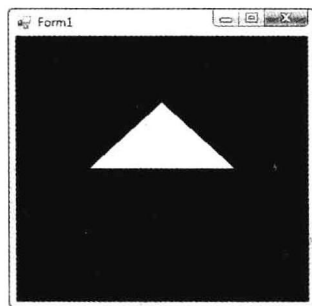


图 5-6 渲染得到的三角形

`GL_TRIANGLES` 使用读取的 3 个顶点绘制一个三角形。加载网格时，更常见的做法是使用 `GL_TRIANGLE_STRIP`。`GL_TRIANGLE` 需要 3 个顶点来绘制三角形。`GL_TRIANGLE_STRIP` 与之类似。第一个三角形需要 3 个顶点，但是之后的每个三角形只需要另外一个顶

点。两者的区别如图 5-7 所示。

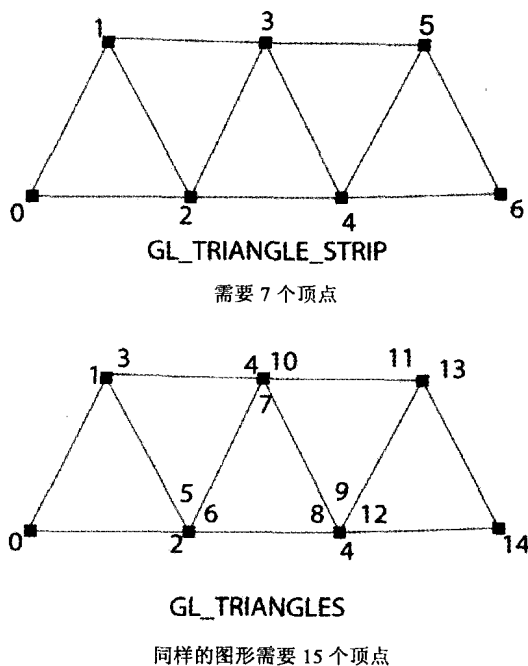


图 5-7 `GL_TRIANGLE` 和 `GL_TRIANGLE_STRIP`

4. 上色和旋转三角形

3D 编程世界的 Hello World 程序就是旋转一个各顶点分别为红色、绿色和蓝色的三角形。这可以演示所有基本的图形功能都正常工作，有经验的图形程序员可以轻松地以此为基础进行开发。

顶点可以存储大量不同的信息。现在，这里的 3 个顶点只包含位置信息。通过在立即模式下的每个顶点调用之前设置颜色信息，可以添加颜色数据。

```
Gl.glBegin(Gl.GL_TRIANGLE_STRIP);
{
    Gl.glColor3d(1.0, 0.0, 0.0);
    Gl.glVertex3d(-0.5, 0, 0);
    Gl.glColor3d(0.0, 1.0, 0.0);
    Gl.glVertex3d(0.5, 0, 0);
    Gl.glColor3d(0.0, 0.0, 1.0);
    Gl.glVertex3d(0, 0.5, 0);
}
Gl.glEnd();
```

运行这段代码将得到一个 3 个角分别为红色、绿色和蓝色的三角形。在三角形的表面上，颜色彼此混合，渗入到对方之中。默认情况下，OpenGL 将对顶点之间的颜色进行插

值。实际中，这意味着可以为三角形的每个顶点提供不同的颜色，然后被渲染的三角形的每个像素将根据与每个顶点的距离进行上色。基本的光照系统获取每个顶点的光照信息，然后利用这种插值来确定如何为表面添加阴影。

现在只剩下旋转三角形了。旋转三角形有两种方法：移动全部顶点，或者移动整个场景。移动整个场景更容易一些，因为 OpenGL 提供了一个用于旋转的函数。

```
Gl.glRotated(10 * elapsedTime, 0, 1, 0);
Gl.glBegin(Gl.GL_TRIANGLE_STRIP);
{
    Gl.glColor4d(1.0, 0.0, 0.0, 0.5);
    Gl.glVertex3d(-0.5, 0, 0);
    Gl.glColor3d(0.0, 1.0, 0.0);
    Gl.glVertex3d(0.5, 0, 0);
    Gl.glColor3d(0.0, 0.0, 1.0);
    Gl.glVertex3d(0, 0.5, 0);
}
Gl.glEnd();
```

这里有几个需要简单解释的地方。glRotate 接受 4 个参数。第一个参数是需要旋转的角度，单位为度，后面的 3 个参数是旋转时围绕的轴。旋转效果是累加的，也就是说，如果两次调用 glRotated，两次旋转都将被应用。在这里的示例中，更新调用将被不断地调用，所以旋转的度数将持续增加，三角形就呈现出旋转效果。旋转函数在旋转 360° 以后将重新计算，所以旋转 361° 相当于旋转 1°。

接下来的 3 个参数描述的是轴。这 3 个参数实际上是一个归一化向量。本书后面的部分将介绍向量。现在只需要认为向量是一条单位长度的线。这里的线在 Y 轴上为 1，所以是一条沿着 Y 轴的线。假设这条线通过三角形的中心，从三角形的上顶点连接到底边的中点。旋转这条线将会旋转整个三角形。思考一下，如果使用参数(1, 0, 0)和(1, 0, 1)旋转三角形会是什么效果，然后修改程序，看看自己的想法是否正确。

第一个参数是旋转角度的度数。将它与占用时间相乘是为了保证移动不受时间的影响，并使动画更加流畅。如果程序的帧率为 60fps，那么经过的帧时间将在 0.1s 左右。这意味着将三角形旋转 10° 的话需要 1s。如果其他人在一台更慢的计算机上运行这个程序，它只能以 30fps 的速度运行，那么每一帧需要 0.2s 的时间才能完成。每秒的帧数更少，但是在两台计算机上旋转三角形所需的时间是相同的。如果没有将旋转的度数乘以帧之间经过的时间，那么在每秒运行 60 帧的那台计算机上，程序的速度将加快一倍。

5.4 小结

大多数游戏都有一个中心循环，负责检查玩家输入、更新游戏世界，然后更新屏幕。

默认情况下，C#没有这样的一个中心循环，所以需要使用一些 C 函数来创建一个快速游戏循环，用于创建游戏。还需要使用其他 C 函数来访问时间分配信息，这样每个帧的时间都可以确定。这个时间可以确保即使运行在速度不同的计算机上，游戏也可以流畅地运行。

Tao 框架库有一个叫做 SimpleOpenGLControl 的控件。通过这个控件，在 Windows 窗体中使用 OpenGL 变得很简单。旋转的三角形经常作为 OpenGL 中的 Hello World 程序。三角形由 3 个顶点组成，每个顶点可以具有不同的颜色。如果三角形的顶点具有不同的颜色，三角形的像素将根据与每个顶点的距离确定。OpenGL 有一个 glRotated 函数，用于处理最终旋转三角形的任务。

第 6 章

游 戏 结 构

现在,进行渲染的基础部分已经可以开始工作,是时候回过头来重新看看游戏架构了。即使很小的游戏(例如 Pong),通常也需要大量的代码。Pong 是一个很简单游戏,玩家左右移动挡板,尝试使弹球越过对手的挡板。这个 1987 年发布的游戏是早期最流行的游戏之一,所以有无数人不断地重新实现它。SourceForge(<http://www.sourceforge.net>)上包含了几十万个开源项目,其中就包含了不少开源的 Pong 克隆,最简单的版本都有超过 1 000 行代码。作为一名程序员,有一些策略来管理这种复杂程度是十分重要的。

大型游戏的复杂程度比 Pong 高几个数量级。据 Kelly Brock 说“模拟人生”(The Sims)游戏中有一个函数包含超过了 32 000 行代码。这是在 Software Development for Games 邮件列表的一次讨论中贴出的(对于雄心勃勃的游戏程序员,这是一个非常好的邮件列表,可以注册加入这个网站: <http://list.midnightryder.com/listinfo.cgi/sweng-gamedev-midnightryder.com>)。如果期限太紧,压力太大,开发出的游戏代码将会变得庞大而无法管理。熟悉一些基本的架构技术可以帮助代码变得更加清晰,函数的功能更加独立。

6.1 游戏对象的基本模式

大部分游戏对象至少需要两个函数:(1)更新函数,对象在这里处理动画或其他随时间改变的事物;(2)渲染函数,对象使用这个函数在屏幕上绘制自身。在 C#中通过接口将很容易使用这两个函数。

```
public interface IGameObject
{
    void Update(double elapsedTime);
```

```
void Render();  
}
```

现在，希望在游戏中创建的任何对象都可以继承这个接口。IGameObject 中的 I 用于在代码中表示这是一个接口。设计游戏结构时，代码可以引用通用的 IGameObject，而不必担心游戏对象实际上是什么。现在，我们可以立即开始使用游戏对象来描述游戏状态。

6.2 处理游戏状态

游戏通常被分成多个不同的状态。在最顶层可能是公司的启动界面，然后是标题菜单，其子菜单中包含了声音等选项和启动游戏的选项，甚至还可能有其他一些选项。下面所示为编写这类游戏状态的一种简单的方式。

```
enum GameState  
{  
    CompanySplash,  
    TitleMenu,  
    PlayingGame,  
    SettingsMenu,  
}  
  
GameState _currentState = GameState.CompanySplash;  
  
public void Update(double elapsedTime)  
{  
    switch (_currentState)  
    {  
        case GameState.CompanySplash:  
        {  
            // Update the starting splash screen  
        } break;  
        case GameState.SettingsMenu:  
        {  
            // Update the settings menu  
        } break;  
        case GameState.PlayingGame:  
        {  
            // Update the game  
        } break;  
        case GameState.TitleMenu:  
        {  
            // Update title menu  
        } break;  
        default:  
        {
```

```
        // Error invalid state
    } break;
}
}
```

这里有一些事情需要注意。代码很长，很容易变得过于复杂而难以理解。它也违反了 DRY 原则。要添加新状态，比如一个致谢屏幕，必须在游戏状态枚举中添加一个新项，然后把这个新项添加到 switch 语句中。扩展代码时，需要修改的位置越少越好。

通过使用 `IGameObject` 接口，可以将庞大的 switch 语句替换为一个更好的系统。在 `IGameObject` 中，每个游戏状态都是一个游戏对象。

为了理解这个示例，最好使用一个游戏循环启动一个新项目。完成这个操作后，就需要创建一个新类，为每个游戏状态实现 `IGameObject` 接口。

```
namespace GameStructure
{
    class SplashScreenState : IGameObject
    {
    }
}
```

这里创建了一个继承 `IGameObject` 接口的启动界面类。上面的代码是无效的，因为它没有实现任何 `IGameObject` 方法。可以使用一个重构方法来自动创建方法，而不必手动输入它们。右击 `IGameObject` 文本，弹出的快捷菜单如图 6-1 所示。

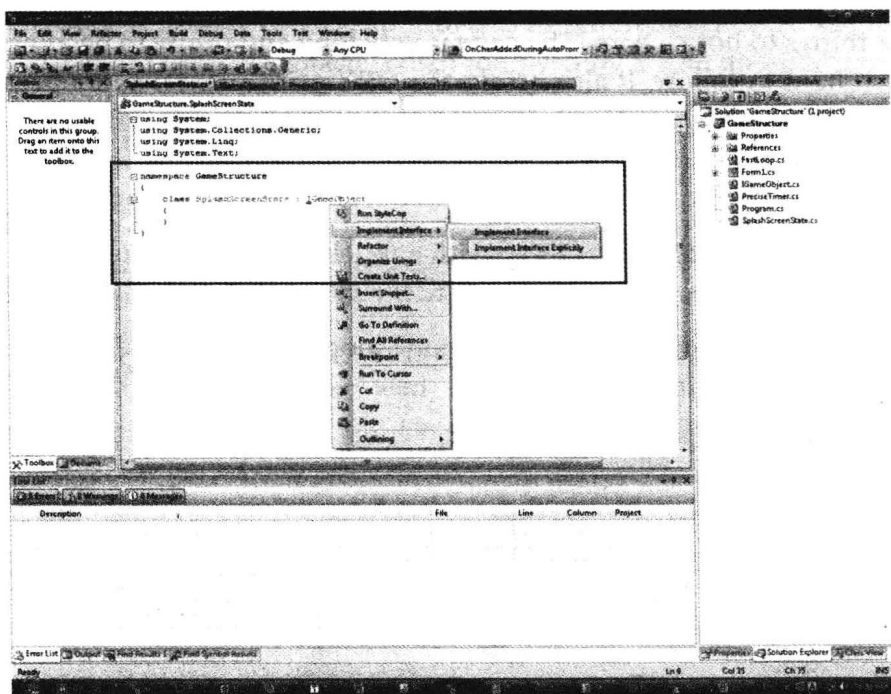


图 6-1 实现一个接口

有两个选项。**Implement Interface Explicitly** 完成的功能与 **Implement Interface** 相同，但是不创建 **Render()** 方法，而是创建 **IGameObject.Render()**，从而明确地显示方法来自什么地方。重构工具实现 **IGameObject** 接口后的代码如下。

```
class SplashScreenState : IGameObject
{
    #region IGameObject Members

    public void Update(double elapsedTime)
    {
        throw new NotImplementedException();
    }
    public void Render()
    {
        throw new NotImplementedException();
    }

    #endregion
}
```

我们现在还不会实现启动界面功能，因此可以删除异常，以简化代码的测试。类似于下面的代码更容易处理。

```
public void Update(double elapsedTime)
{
    System.Console.WriteLine("Updating Splash");
}

public void Render()
{
    System.Console.WriteLine("Rendering Splash");
}
```

这只是一个状态，其他的状态可以按照类似的方式创建。你可以自己创建一些状态来进行实践。

创建好状态后，处理状态的代码十分简单。为了与其余的代码一致，应该创建 **TitleMenuState**。它不需要有任何复杂的功能，只要能够在渲染和更新函数中打印出自己的名称就可以了。如果仍然不明白如何创建这个类，可以参考本书配套光盘中的代码。

下面的示例说明了如何在 **Form.cs** 文件中使用状态。

```
StateSystem _system = new StateSystem();
public Form1()
{
    // Add all the states that will be used.
```

```

_system.AddState("splash", new SplashScreenState(_system));
_system.AddState("title_menu", new TitleMenuState());
// Select the start state
_system.ChangeState("splash");

```

可以创建状态，并把它们添加到状态系统中，这些状态必须要有一个名称以便标识它们。然后就可以调用 **ChangeState** 并传入状态名称来选择状态。状态系统将管理当前活动状态的更新和渲染。有时候，某个状态可能要修改活动状态。例如，启动界面通常显示一幅图片或者一个动画，然后将状态修改为标题屏幕。要修改启动界面状态，就必须引用状态系统。

```

class SplashScreenState : IGameObject
{
    StateSystem _system;
    public SplashScreenState(StateSystem system)
    {
        _system = system;
    }

    #region IGameObject Members

    public void Update(double elapsedTime)
    {
        // Wait so many seconds then call _system.ChangeState("title_menu")
        System.Console.WriteLine("Updating Splash");
    }

    public void Render()
    {
        System.Console.WriteLine("Rendering Splash");
    }

    #endregion
}

```

然后在创建状态时，可以把 **StateSystem** 传入到构造函数中。

```

_system.AddState("splash", new SplashScreenState(_system));
class StateSystem
{
    Dictionary<string, IGameObject> _stateStore = new Dictionary<string,
IGameObject>();
    IGameObject _currentState = null;
    public void Update(double elapsedTime)

```

```

    {
    if (_currentState == null)
    {
        return; // nothing to update
    }
    _currentState.Update(elapsedTime);
    }

    public void Render()
    {
        if (_currentState == null)
        {
            return; // nothing to render
        }
        _currentState.Render();
    }

    public void AddState(string stateId, IGameObject state)
    {
        System.Diagnostics.Debug.Assert( Exists(stateId) == false );
        _stateStore.Add(stateId, state);
    }

    public void ChangeState(string stateId)
    {
        System.Diagnostics.Debug.Assert(Exists(stateId));
        _currentState = _stateStore[stateId];
    }

    public bool Exists(string stateId)
    {
        return _stateStore.ContainsKey(stateId);
    }
}

```

StateSystem 类非常适合单元测试，可以试着在 NUnit 中编写一些测试。测试应该是简短的代码片段，只检查程序的一个部分，如下所示。

```

[TestFixture]
public class Test_StateSystem
{
    [Test]
    public void TestAddedStateExists()
    {

```



```

    StateSystem stateSystem = new StateSystem();
    stateSystem.AddState("splash", new SplashScreenState
        (stateSystem));

    // Does the added function now exist?
    Assert.IsTrue(stateSystem.Exists("splash"));
}

```

测试的其余部分可以在本书的配套光盘中找到, 位置为 Code\Chapter 6\Chapter6-2\Test_StateSystem.cs。试着编写自己的测试, 然后将它们与光盘中的测试进行比较。

6.3 游戏状态演示

现在已经创建了 `StateSystem`, 接下来通过一个简短的演示来展示其实际应用。我们可以首先实现启动界面状态。由于只能绘制旋转的三角形, 这个界面不会太华丽, 但是对于演示来说是足够了。首先实现一个标题启动屏幕。

```

class SplashScreenState : IGameObject
{
    StateSystem _system;
    double _delayInSeconds = 3;

    public SplashScreenState(StateSystem system)
    {
        _system = system;
    }

    #region IGameObject Members

    public void Update(double elapsedTime)
    {
        _delayInSeconds -= elapsedTime;
        if (_delayInSeconds <= 0)
        {
            _delayInSeconds = 3;
            _system.ChangeState("title_menu");
        }
    }

    public void Render()
    {
        Gl.glClearColor(1, 1, 1, 1);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    }
}

```

```

        Gl.glFinish();
    }

    #endregion
}

```

这段代码让启动界面状态等待 3s，然后将状态修改为标题菜单。启动界面菜单在活动时会将屏幕渲染为白色。在介绍了 2D 渲染和精灵以后，我们才能够创建更有趣的演示。

```

class TitleMenuState : IGameObject
{
    double _currentRotation = 0;
    #region IGameObject Members

    public void Update(double elapsedTime)
    {
        _currentRotation = 10 * elapsedTime;
    }

    public void Render()
    {
        Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        Gl.glPointSize(5.0f);

        Gl.glRotated(_currentRotation, 0, 1, 0);
        Gl.glBegin(Gl.GL_TRIANGLE_STRIP);
        {
            Gl.glColor4d(1.0, 0.0, 0.0, 0.5);
            Gl.glVertex3d(-0.5, 0, 0);
            Gl.glColor3d(0.0, 1.0, 0.0);
            Gl.glVertex3d(0.5, 0, 0);
            Gl.glColor3d(0.0, 0.0, 1.0);
            Gl.glVertex3d(0, 0.5, 0);
        }
        Gl.glEnd();
        Gl.glFinish();
    }

    #endregion
}

```

还是前面的旋转的三角形。前面的示例中已经把这些状态加载到了状态系统中。剩下

的唯一一项任务是在 Form.cs 中调用 Update 和 Render 函数。

```
private void GameLoop(double elapsedTime)
{
    _system.Update(elapsedTime);
    _system.Render();
    _openGLControl.Refresh();
}
```

代码就是这些。现在运行代码，在显示一个白色屏幕 3s 后，将显示一个旋转的三角形。这证明了状态系统工作得很好，可以使用这个状态系统来分解代码。接下来从架构的高级概念中抽身出来，回到渲染的细节。

6.4 使用投影设置场景

到现在为止，我们还没有显式地设置 OpenGL 场景，而一直使用默认的设置。通过手动设置场景，可以更细致地控制游戏的显示方式。

6.4.1 字体大小和 OpenGL 视口大小

窗体的大小现在采用的是 Visual Studio 确定的默认大小。真正的游戏需要指定窗口的大小。在当前设置中，尝试启用全屏模式。结果如图 6-2 所示。



图 6-2 全屏模式的问题

程序成功地占据了整个屏幕，但是三角形仍然在左下角渲染。产生这种问题的原因是有两个不同的尺寸需要考虑。一个是窗体的尺寸，在全屏模式下它发生了变化。另一个是 OpenGL 的视口的尺寸，它没有任何变化，所以仍然以相同大小渲染三角形。

为了解决这个问题，必须知道窗体何时发生了变化，并把这种变化通知给 OpenGL。窗体的大小可以通过多种方式获知。图 6-3 显示了窗体的 Size 和 ClientSize 的差值。大小

包含框架和标题栏等。我们只关心绘制 OpenGL 图形的窗体的大小。这可以通过查询窗体的 `ClientSize` 值获知。

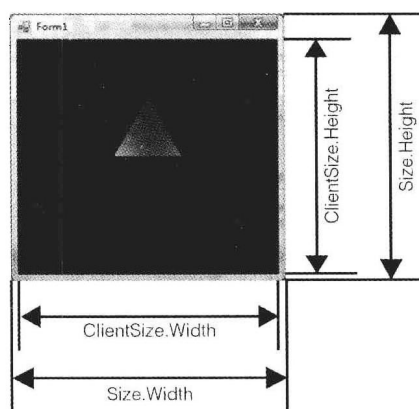


图 6-3 窗体的 `Size` 和 `ClientSize`

在窗体中可以看到可重写的方法调用 `OnClientSizeChanged`，这是更新 OpenGL 视口大小的绝佳位置。在 `Form.cs` 中，添加如下代码。

```
protected override void OnClientSizeChanged(EventArgs e)
{
    base.OnClientSizeChanged(e);
    Gl.glViewport(0, 0, this.ClientSize.Width, this.ClientSize.Height);
}
```

现在，在客户区大小发生变化时，例如进入了全屏模式，OpenGL 视口也会相应地改变。这样也可以在运行时调整窗体的大小，而不会产生问题。再次在全屏模式下运行程序，这一次三角形就大得多了。

窗体最初的大小可以通过 `ClientSize` 属性进行设置。

```
ClientSize = new Size(800, 600);
```

这会将窗体的客户区设置为 800 像素×600 像素。

6.4.2 宽高比

人都有两只眼睛。两只眼睛在脸上处在一个水平的位置，所以我们在水平方向上看到的范围比在垂直方向上看到的范围要广得多。宽屏电视和监视器就比方方正正或者垂直方向更长的电视或监视器用起来更舒服。视区的宽度和高度之间的关系叫做宽高比(**aspect ratio**)。特定的宽高比要比其他宽高比更让人感到舒服。在代码中提供一组不同的宽高比是十分容易的。

如果 `ClientSize` 被设为 800 像素×600 像素，则它的宽高比为 1.33，即宽度是高度的 1.33 倍。这个比例也被称为 4:3，这是最低值，随着宽屏变得越来越流行，16:9 的宽高比越

来越常见。PlayStation 3 的默认分辨率为 1 280 像素×720 像素，宽度是高度的 1.73 倍，这就是常说的 16:9 的宽高比。

```
if (_fullscreen)
{
    FormBorderStyle = FormBorderStyle.None;
    WindowState = FormWindowState.Maximized;
}
else
{
    ClientSize = new Size(1280, 720);
}
```

尝试不同的宽高比，看看自己喜欢哪一种。记住，如果想要把自己的游戏发布给很多人，他们支持的分辨率可能比你能支持的分辨率要低。尝试支持多种不同的宽高比和分辨率是有必要的。

当尝试各种分辨率时，可以观察到在特定的分辨率中，三角形看上去是被挤压和扭曲了。现在 OpenGL 的视口已经被正确地设置了，但是 OpenGL 默认的宽高比是 1:1，即一个正方形。它不能很好地映射为 4:3 或者 16:9。OpenGL 的宽高比之所以为 1:1，是因为这是默认设置的宽高比。为了修改这个设置，必须修改投影矩阵。

6.4.3 投影矩阵

计算机监视器和电视显示 2D 图片，但是 OpenGL 处理的是 3D 数据。投影矩阵可以将 3D 数据转换成 2D 屏幕上可以显示的数据。我们关注的投影矩阵分为两种：正射投影矩阵和透视投影矩阵。简单来说，正射投影矩阵用于 2D 图形，例如平视显示器元素、2D 游戏和文本。透视投影矩阵用于 3D 游戏，例如 FPS 游戏。

正射投影会忽略深度信息，某个项目离用户多远无关紧要，不管是远还是近，它的大小是不变的。而在透视投影中，越远的事物越小。这是两种投影的主要区别。

6.4.4 2D 图形

每个 3D 游戏都需要能够渲染 2D 图形。

在 Form.cs 中添加一个名为 Setup2DGraphics 的新函数。

```
private void Setup2DGraphics(double width, double height)
{
    double halfWidth = width / 2;
    double halfHeight = height / 2;
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    Gl.glOrtho(-halfWidth, halfWidth, -halfHeight, halfHeight, -100, 100);
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
}
```

这段代码包含多个新的 OpenGL 函数，但是它们都十分简单。OpenGL 有多种矩阵模式。值 GL_PROJECTION 会修改 OpenGL 的状态。状态改变后，所有的 OpenGL 命令都会影响投影矩阵。现在可以修改这个矩阵，以建立一个正射投影矩阵。

glLoadIdentity 会清除当前的投影信息。下一个命令 glOrtho 建立了正射投影矩阵。下面列出了这个函数的 6 个参数。

```
void glOrtho(GLdouble left,  
             GLdouble right,  
             GLdouble bottom,  
             GLdouble top,  
             GLdouble nearVal,  
             GLdouble farVal);
```

前 4 个参数指定了世界视图的大小。图 6-4 显示了正射投影，以及 6 个参数对这种投影的影响。现在，原点正处在屏幕的中央。我决定不加修改。要使原点位于左上角，可以使用下面的代码。

```
Gl.glOrtho(0, width, -height, 0, -100, 100);
```

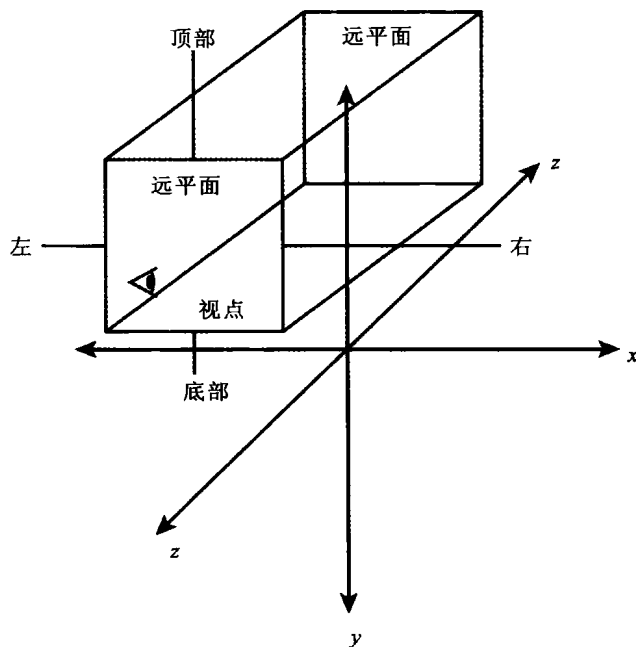


图 6-4 正射投影

最后的两个值是近平面和远平面。如果一个顶点的 z 位置比远平面更远，该顶点不会被渲染。如果比近平面更近，它也不会被渲染。一般来说，2D 图形的 z 位置都被设为 0，所以远近平面并不起作用。它们对于渲染 3D 图形就重要多了。

设置函数可以在 Form.cs 构造函数中调用。

```
public Form1()
{
    // Add all the states that will be used.
    _system.AddState("splash", new SplashScreenState(_system));
    _system.AddState("title_menu", new TitleMenuState());

    // Select the start state
    _system.ChangeState("splash");

    InitializeComponent();
    _openGLControl.InitializeContexts();

    if (_fullscreen)
    {
        FormBorderStyle = FormBorderStyle.None;
        WindowState = FormWindowState.Maximized;
    }
    else
    {
        ClientSize = new Size(1280, 720);
    }
    Setup2DGraphics(ClientSize.Width, ClientSize.Height);
    _fastLoop = new FastLoop(GameLoop);
}
```

每次窗体改变大小时，都需要重新创建投影矩阵。因此，在 `OnClientSizeChanged` 回调中还需要添加对 `Setup2DGraphics` 的调用。

```
protected override void OnClientSizeChanged(EventArgs e)
{
    base.OnClientSizeChanged(e);
    Gl.glViewport(0, 0, this.ClientSize.Width, this.ClientSize.Height);
    Setup2DGraphics(ClientSize.Width, ClientSize.Height);
}
```

标题状态会渲染一个三角形，但是三角形的最大宽度只有一个 OpenGL 单位。前面的投影矩阵的宽度和高度都是两个单位，所以三角形的大小很合适。这个新的投影矩阵的宽度和高度分别是 1 280 像素和 720 像素，三角形只占据一个像素，因此根本无法看到。

解决这个问题的一个简单的方法是使三角形变大。找到三角形的绘制代码，使其宽度和高度从 1 个像素变为 50 个像素。

```
Gl.glColor4d(1.0, 0.0, 0.0, 0.5);
Gl.glVertex3d(-50, 0, 0);
Gl.glColor3d(0.0, 1.0, 0.0);
```

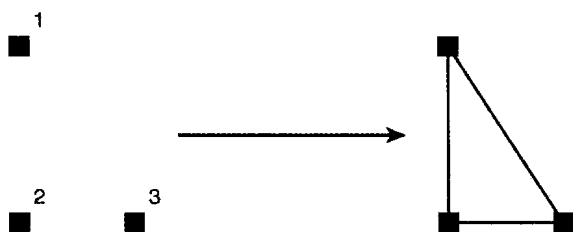
```
Gl.glVertex3d(50, 0, 0);  
Gl.glColor3d(0.0, 0.0, 1.0);  
Gl.glVertex3d(0, 50, 0);
```

这会使三角形再次可见。大多数 2D 图形使用两个三角形来构成一个四边形。然后对这个四边形应用纹理，使其形成 2D 游戏或者平视显示的基础。

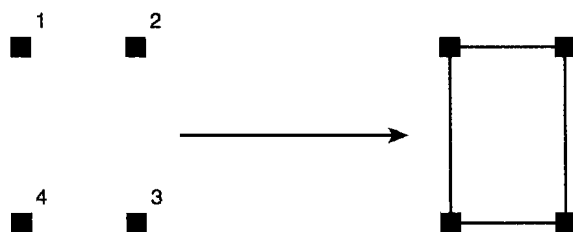
6.5 精灵

创建精灵的第一步是创建一个四边形。三角形由 3 个顶点构成，四边形则由 4 个顶点构成，如图 6-5 所示。精灵是一个非常基本的游戏元素，使其成为一个单独的类是一个好主意。还有一个专门负责绘制精灵的类 `Renderer`。在实现 `Render` 和 `Sprite` 类之前，首先考虑它们的使用方式会很有用。下面列出了一些伪代码，演示了如何使用精灵。

```
Renderer renderer = new Renderer();  
Sprite spaceship = new Sprite();  
spaceship.SetPosition(0, 0);  
spaceship.SetTexture(_textureManager.Get("spaceship"));  
renderer.DrawSprite(spaceship);
```



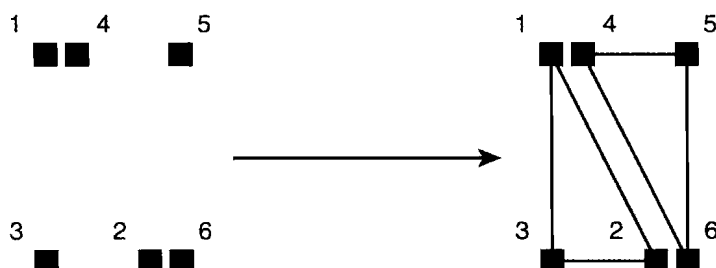
三角形由 3 个顶点构成，使用 `GL_TRIANGLES` 创建



嵌块由 4 个顶点构成，使用 `GL_QUADS` 创建

图 6-5 精灵的布局

正方形由 4 个顶点构成，但是通常使用 6 个顶点绘制两个三角形，通过这两个三角形构成一个正方形。图形卡非常擅长处理三角形，所以许多引擎都将它们的资源分解成三角形或者三角形带。分解后的正方形如图 6-6 所示。



通过 GL_TRIANGLES，可以使用 6 个顶点构成一个嵌块
如果顶点 1、4 和 2、6 的位置完全相同，就创建了一个嵌块

图 6-6 使用两个三角形构成的正方形

在阅读的同时进行编码将有助于理解本章的内容。没有必要启动一个新项目，只需重用 **GameStructure** 项目即可。创建一个从 **IGameObject** 继承的新类，将其命名为 **DrawSpriteState**。

```
class DrawSpriteState : IGameObject
{
    #region IGameObject Members

    public void Update(double elapsedTime)
    {
    }

    public void Render()
    {
    }

    #endregion
}
```

记得在文件的顶部包含了语句 `using Tao.OpenGl;`，然后才可以使用 **OpenGL** 命令。这个游戏状态将用于测试精灵绘制代码，它应该在程序启动后尽快加载。按照如下所示修改 **Form.cs** 中的代码。

```
public Form1()
{
    // Add all the states that will be used.
    _system.AddState("splash", new SplashScreenState(_system));
    _system.AddState("title_menu", new TitleMenuState());
}
```

```
_system.AddState("sprite_test", new DrawSpriteState());  
  
// Select the start state  
_system.ChangeState("sprite_test");
```

这会加载当前为空的精灵绘制状态。运行这段代码将显示一个空窗口。四方形通过将两个三角形排列为一个正方形来构成。OpenGL 按照顺时针方向绘制三角形。

```
public void Render()  
{  
    Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);  
    Gl.glBegin(Gl.GL_TRIANGLES);  
    {  
        Gl.glVertex3d(-100, 100, 0); // top left  
        Gl.glVertex3d(100, 100, 0); // top right  
        Gl.glVertex3d(-100, -100, 0); // bottom left  
    }  
    Gl.glEnd();  
}
```

这段代码将绘制四方形的上半部分。运行这段代码将产生一个如图 6-7 所示的图像。

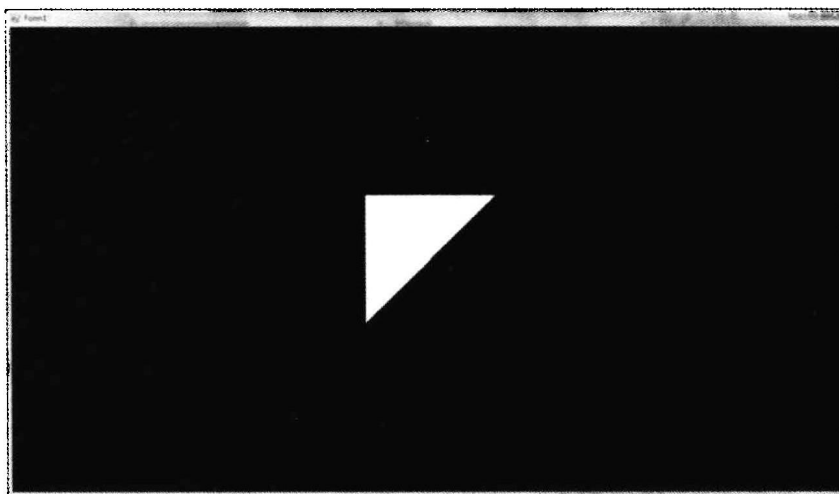


图 6-7 四方形的上半部分

绘制第二个部分很简单。按照顺时针方向，首先绘制右上角的顶点，然后绘制右下角的顶点，最后绘制左下角的顶点。创建这一部分的代码应该紧跟在创建上半部分的代码的后边。

```
Gl.glVertex3d( 100, 100, 0); // top right  
Gl.glVertex3d( 100, -100, 0); // bottom right
```

```
Gl.glVertex3d(-100, -100, 0); // bottom left
```

这两个三角形构成了一个完整的正方形。并不是所有的游戏精灵都会是正方形。因此，能够指定宽度和高度是非常重要的。当前的正方形是 200×200 。OpenGL 没有明确地规定单位，200 这个值的单位可以是英尺、米等。这里将屏幕设置为 200 个 OpenGL 单位对应的 200 个像素，但是并不总是保证如此。现在，精灵的宽度、高度和位置都被硬编码。应该使用变量替换这些数值。

```
double height = 200;
double width = 200;
double halfHeight = height / 2;
double halfWidth = width / 2;

Gl.glBegin(Gl.GL_TRIANGLES);
{
    Gl.glVertex3d(-halfWidth, halfHeight, 0); // top left
    Gl.glVertex3d( halfWidth, halfHeight, 0); // top right
    Gl.glVertex3d(-halfWidth, -halfHeight, 0); // bottom left

    Gl.glVertex3d( halfWidth, halfHeight, 0); // top right
    Gl.glVertex3d( halfWidth, -halfHeight, 0); // bottom right
    Gl.glVertex3d(-halfWidth, -halfHeight, 0); // bottom left
}
```

值与前面一样，但是现在将高度和宽度存储在变量中，这符合 DRY 原则。现在很容易修改高度和宽度，得到任意数量的形状有趣的矩形。

6.5.1 定位精灵

现在很容易修改精灵的大小，但是还没有办法修改位置。接下来就编写完成这种功能的代码。

```
double x = 0;
double y = 0;
double z = 0;
Gl.glBegin(Gl.GL_TRIANGLES);
{
    Gl.glVertex3d( x - halfWidth, y + halfHeight, z); // top left
    Gl.glVertex3d( x + halfWidth, y + halfHeight, z); // top right
    Gl.glVertex3d( x - halfWidth, y - halfHeight, z); // bottom left

    Gl.glVertex3d(x + halfWidth, y + halfHeight, z); // top right
    Gl.glVertex3d(x + halfWidth, y - halfHeight, z); // bottom right
    Gl.glVertex3d(x - halfWidth, y - halfHeight, z); // bottom left
}
```

```
}  
Gl.glEnd();
```

这与前面的代码类似，但是现在 x 、 y 和 z 位置可以修改。这个位置代表四方形的中心。尝试修改 x 、 y 和 z 值，然后四处移动四方形。

6.5.2 使用四方形管理纹理

将纹理应用到四方形上很容易。处理纹理时，棘手的部分是从硬盘上将纹理加载到内存中。需要创建一个纹理类来代表内存中的纹理，创建一个 `TextureManager` 类来存储纹理。在内部，`OpenGL` 通过一个整数 `id` 来引用纹理，纹理结构将存储该 `id`，以及纹理的宽度和高度。

```
public struct Texture  
{  
    public int Id { get; set; }  
    public int Width { get; set; }  
    public int Height { get; set; }  
  
    public Texture(int id, int width, int height) : this()  
    {  
        Id = id;  
        Width = width;  
        Height = height;  
    }  
}
```

纹理类十分简单。它的构造函数调用 `this` 构造函数，因为这是一个结构类型，需要为要使用的自动生成的访问器方法初始化成员。

接下来，`TextureManager` 从硬盘上加载纹理数据，并将纹理与 `OpenGL` 提供的 `id` 关联起来。纹理也会与一个人类可读的名称关联，以方便程序员使用。加载纹理的代码还要求将一个额外的引用 `Tao.DevIl` 添加到项目中。按照与添加 `Tao.OpenGL` 相同的方式添加它。`DevIl` 是 `Developer's Image Library` 的缩写。它可以加载大多数图片格式，供 `OpenGL` 使用。

`DevIl` 库还要求运行许多 `DLL` 文件，这些文件必须与二进制文件位于相同的目录中。这个目录可能是 `bin\debug`。找到 `Tao` 框架的目录，它很可能是 `C:\Program Files(x86)`(或者 `Windows XP` 中的 `C:\Program Files` 目录)。找到 `TaoFramework\lib` 并把 `DevIl.dll`、`ILU.dll` 和 `ILUT.dll` 复制到 `bin\debug` 目录中。在发布时，还需要将所有相关的 `dll` 复制到 `bin\release` 中。完成这些操作后，就可以开始使用 `DevIl` 了。

必须初始化 `DevIl` 库，并指定将在 `OpenGL` 中使用它。`Form.cs` 是初始化 `DevIl` 的一个不错的位置。确保在 `Form.cs` 文件的顶部添加了 `using` 语句，如下所示。

```
using Tao.DevIl;
```

然后在窗体的构造函数中，添加下面的代码。

```
// Init DevIl
Il.ilInit();
Ilu.iluInit();
Ilut.ilutInit();
Ilut.ilutRenderer(Ilut.ILUT_OPENGL);
```

创建一个新类 **TextureManager**。在文件顶部，添加 **DevIl** 和 **OpenGL** 的 **using** 语句。基本的 **TextureManager** 代码如下所示。

```
class TextureManager : IDisposable
{
    Dictionary<string, Texture> _textureDatabase = new Dictionary<string,
Texture>();

    public Texture Get(string textureId)
    {
        return _textureDatabase[textureId];
    }
    #region IDisposable Members

    public void Dispose()
    {
        foreach (Texture t in _textureDatabase.Values)
        {
            Gl.glDeleteTextures(1, new int[] { t.Id });
        }
    }

    #endregion
}
```

这个类实现了 **IDisposable**，它确保类在被销毁时会从内存中释放纹理。另外的唯一一个函数是 **Get**，它接受纹理的名称作为参数，并返回相关联的纹理数据。如果数据不存在，它将抛出一个异常。

TextureManager 中明显少了一个东西：它没有从硬盘加载纹理的函数。

```
public void LoadTexture(string textureId, string path)
{
    int devilId = 0;
    Il.ilGenImages(1, out devilId);
    Il.ilBindImage(devilId); // set as the active texture.
```

```
if (!Il.ilLoadImage(path))
{
    System.Diagnostics.Debug.Assert(false,
        "Could not open file, [" + path + "].");
}
// The files we'll be using need to be flipped before passing to OpenGL
Ilu.iluFlipImage();
int width = Il.ilGetInteger(IL_IL_IMAGE_WIDTH);
int height = Il.ilGetInteger(IL_IL_IMAGE_HEIGHT);
int openGLId = Ilut.ilutGLBindTexImage();

System.Diagnostics.Debug.Assert(openGLId != 0);
Il.ilDeleteImages(1, ref devilId);

_textureDatabase.Add(textureId, new Texture(openGLId, width,
height));
}
```

DevIL 库用于补充 OpenGL，所以它的接口与 OpenGL 很类似。生成一幅图片，然后绑定它。绑定图片意味着所有后续操作都会影响该图片。ilLoadImage 用于把纹理数据加载到内存中。然后对图片调用 iluFlipImage，这会沿 Y 轴翻转图片。大多数场景的图片格式都需要翻转，才能在 OpenGL 中正确地操作。然后查询图片的宽度和高度信息。最后，DevIL 实用程序库用于将纹理与 OpenGL id 绑定起来。id、宽度和高度都被包装在纹理类中，接着该类将被返回。此时 DevIL 在内存中仍然保存着纹理数据。需要使用 ilDeleteImages 释放这些数据，因为这些数据已经从 DevIL 移动到了 OpenGL 中。

测试 TextureManager 需要纹理。从本书配套光盘的 Asset 目录中可以找到一个 TIF 图像文件 face.tif。将该文件复制到项目目录中。再在 Visual Studio 中，右击项目，然后在弹出的快捷菜单中选择 Add Existing Item 命令。我们将把这个 TIF 文件添加到项目中。这将打开一个对话框。为了查看该图片，可能需要将过滤器从 C# Files 改为 All Files。选择 face.tif。最后一步是在 Solution Explorer 中选择 face.tif 文件，右击它，然后在弹出的快捷菜单中选择 Properties 命令，如图 6-8 所示。

显示的多个字段描述了图片文件的属性。将字段 Copy To Output Directory 改为 Copy If Newer。这会把图片文件复制到 bin 目录中，运行程序时，文件总是会在正确的位置。其实没有必要把图片文件添加到项目中，但这是跟踪资源并确保它们位于正确位置的一种简单的方法。

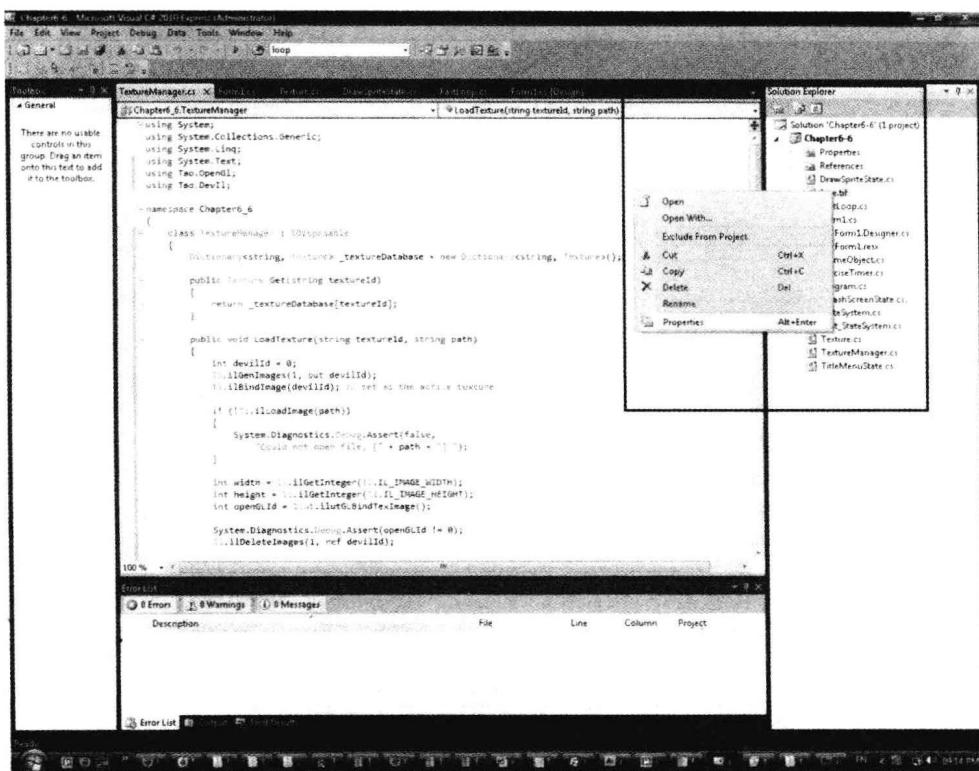


图 6-8 查看文件的属性

现在可以在 Form.cs 文件中创建 TextureManager 对象, 然后将其传入任何需要它的状态。

```
TextureManager _textureManager = new TextureManager();
```

```
public Form1()
{
    InitializeComponent();
    _openGLControl.InitializeContexts();

    // Init DevIL
    Il.ilInit();
    Ilu.iluInit();
    Ilut.ilutInit();
    Ilut.ilutRenderer(Ilut.ILUT_OPENGL);

    // Load textures
    _textureManager.LoadTexture("face", "face.tif");

    // Add all the states that will be used.
    _system.AddState("splash", new SplashScreenState(_system));
```

```

_system.AddState("title_menu", new TitleMenuState());
_system.AddState("sprite_test", new DrawSpriteState());

// Select the start state
_system.ChangeState("sprite_test");

```

Form.cs 中的这段设置代码会创建 TextureManager，初始化 DevII 库，然后尝试加载名为 face.tif 的纹理。运行该程序。如果能正确运行，那么最好，现在就可以从硬盘上加载纹理。如果不能运行，说明有一些地方出问题了。

如果得到异常 Unable to load DLL 'xx.dll': The specified module could not be found，则说明二进制文件不能找到 xx.dll。应该把这个 DLL 文件放到 bin/debug 目录中。检查该目录中是否有这个文件。如果没有，从 Tao 框架中将该文件复制过来。

如果得到异常 BadImageFormatException，这与尝试加载的纹理无关。相反，程序在加载 DevII 库时遇到了问题。最可能的原因是在 64 位平台上使用 Visual Studio 2008 进行开发，但库是针对 32 位进行编译的。解决这个问题的最简单方法是在 Solution Explorer 中右击项目，然后在弹出的快捷菜单中选择 Properties 命令。单击左侧的 Build 选项卡，如图 6-9 所示。

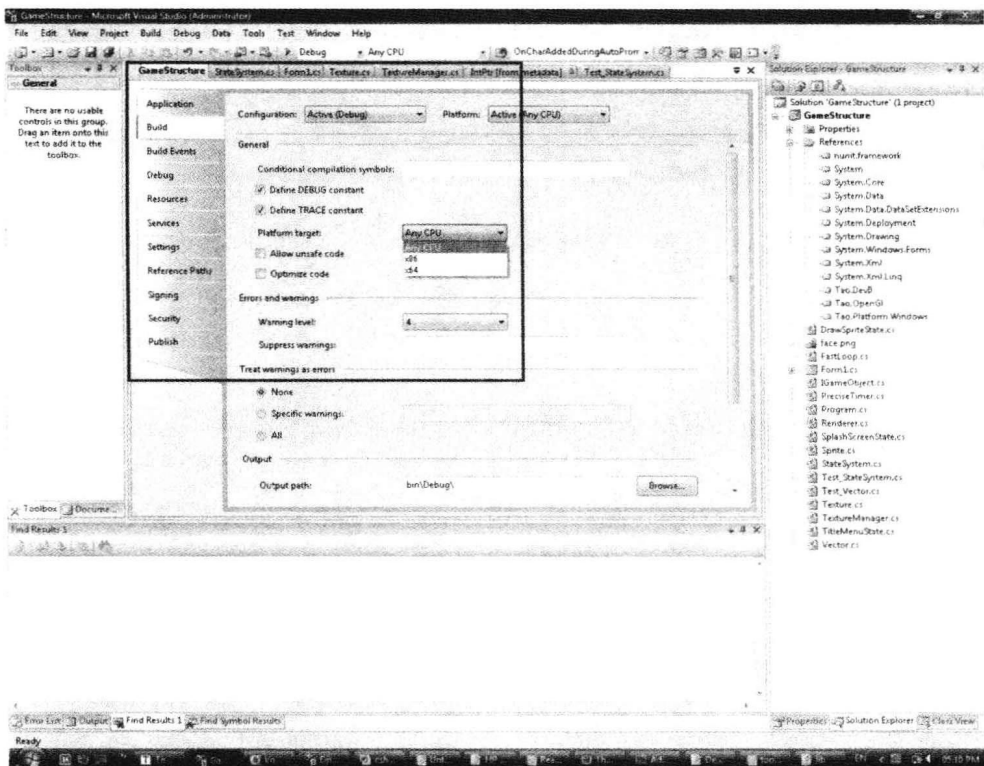


图 6-9 查看项目属性

Platform Target 旁边有一个下拉框。选择 x86 选项，这将针对 32 位系统生成程序。

最后，如果得到 Assertion Failed 信息，信息的文本是 Could not open file, [face.tif]，则说明二进制目录中没有 face.tif。可以将该文件复制到正确的位置，或者确保将其正确地添

加到了解决方案中。

6.5.3 纹理精灵

加载纹理后, 现在是该使用它们了。纹理映射在两个轴上都是从 0 到 1。(0, 0)是左上角, (1, 1)是右下角。对于纹理, 轴不叫 x 和 y , 而是叫 U 和 V 。

图 6-10 显示了每个顶点的 2D(U , V)坐标如何映射到 3D 顶点位置。当四方形示例的纹理可以工作后, 使用(U , V)映射更加容易。

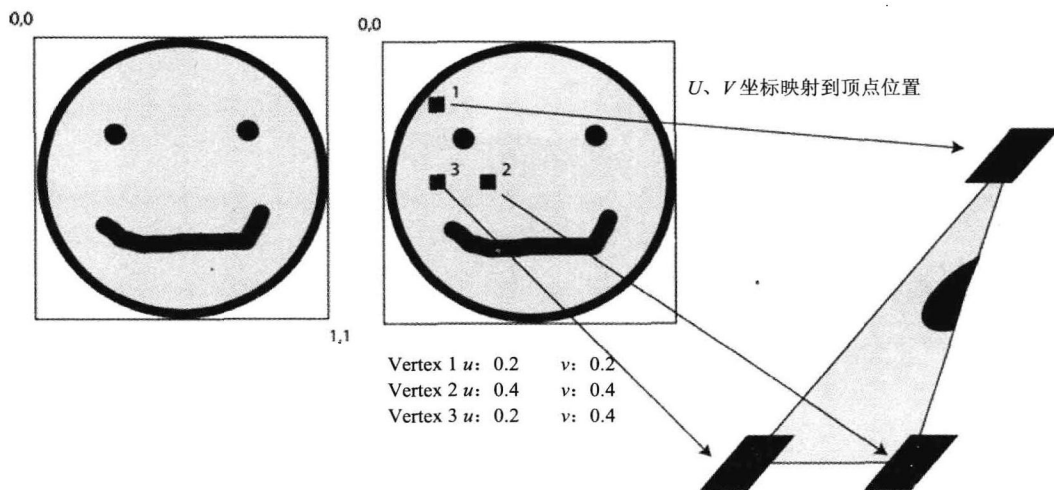


图 6-10 (U, V)映射

返回 `DrawSpriteState` 类。需要创建一个接受 `TextureManager` 作为参数的构造函数。

```
class DrawSpriteState : IGameObject
{
    TextureManager _textureManager;

    public DrawSpriteState(TextureManager textureManager)
    {
        _textureManager = textureManager;
    }
}
```

在 `Form.cs` 中, 需要把该 `TextureManager` 传入到 `DrawSpriteState` 构造函数中。在游戏中访问纹理很容易。

为了使用纹理, 需要告诉 `OpenGL` 启动纹理模式, 然后将纹理作为活动纹理绑定。所有顶点都使用当前绑定的纹理作为它们的纹理信息。最后, 每个顶点都需要关联一些 2D 纹理位置。这里是从(0,0)映射到(1,1), 所以将使用完整的纹理。

代码将为四方形的第一个三角形设置纹理信息, 结果如图 6-11 所示。



图 6-11 对半个正方形应用纹理映射

```
Texture texture = _textureManager.Get("face");
Gl.glEnable(Gl.GL_TEXTURE_2D);
Gl.glBindTexture(Gl.GL_TEXTURE_2D, texture.Id);

Gl.glBegin(Gl.GL_TRIANGLES);
{
    Gl.glTexCoord2d(0, 0);
    Gl.glVertex3d( x - halfWidth, y + halfHeight, z); // top left
    Gl.glTexCoord2d(1, 0);
    Gl.glVertex3d( x + halfWidth, y + halfHeight, z); // top right
    Gl.glTexCoord2d(0, 1);
    Gl.glVertex3d( x - halfWidth, y - halfHeight, z); // bottom left
```

第二个三角形的顶点以相同的方式设置。

```
Gl.glTexCoord2d(1, 0);
Gl.glVertex3d(x + halfWidth, y + halfHeight, z);    // top right
Gl.glTexCoord2d(1, 1);
```

```

Gl.glVertex3d(x + halfWidth, y - halfHeight, z); // bottom right
Gl.glTexCoord2d(0, 1);
Gl.glVertex3d(x - halfWidth, y - halfHeight, z); // bottom left

```

现在这会将纹理完整地映射到四方形。现在尝试修改四方形的尺寸，看看将会发生什么。

现在应用纹理时使用了大量的幻数，根据 DRY 原则，应该把它们移动到变量中。四方形的 (U, V) 映射可以通过两个坐标描述，即左上角的 (U, V) 位置和右下角的 (U, V) 位置。

```

float topUV = 0;
float bottomUV = 1;
float leftUV = 0;
float rightUV = 1;

Gl.glBegin(Gl.GL_TRIANGLES);
{
    Gl.glTexCoord2d(leftUV, topUV);
    Gl.glVertex3d(x - halfWidth, y + halfHeight, z); // top left
    Gl.glTexCoord2d(rightUV, topUV);
    Gl.glVertex3d(x + halfWidth, y + halfHeight, z); // top right
    Gl.glTexCoord2d(leftUV, bottomUV);
    Gl.glVertex3d(x - halfWidth, y - halfHeight, z); // bottom left

    Gl.glTexCoord2d(rightUV, topUV);
    Gl.glVertex3d(x + halfWidth, y + halfHeight, z); // top right
    Gl.glTexCoord2d(rightUV, bottomUV);
    Gl.glVertex3d(x + halfWidth, y - halfHeight, z); // bottom right
    Gl.glTexCoord2d(leftUV, bottomUV);
    Gl.glVertex3d(x - halfWidth, y - halfHeight, z); // bottom left
}
Gl.glEnd();

```

现在很容易修改 U 、 V 坐标。将左上角设为 $(0,0)$ ，将右下角设为 $(2,2)$ 。将右下角修改为 $(-1,-1)$ 。为左上角和右下角尝试不同的位置，看看会发生什么。

6.5.4 alpha 混合精灵

使精灵部分透明是一种很常见的需求。本书配套光盘的 `Assets` 文件夹中还有一个精灵 `face_alpha.tif`。这个图片文件有 4 个通道，分别是红色、绿色、蓝色和 `alpha`。`alpha` 通道移除了笑脸旁边的白色边框。按照添加前一个纹理的方式，将 `face_alpha` 文件添加到项目中。需要把它加载到 `TextureManager` 中。

```

// Load textures
_textureManager.LoadTexture("face", "face.tif");

```

```
_textureManager.LoadTexture("face_alpha", "face_alpha.tif");
```

在 DrawSpriteState 的 Render 调用中，将下面这一行

```
Texture texture = _textureManager.Get("face");
```

改为

```
Texture texture = _textureManager.Get("face_alpha");
```

现在运行代码产生的图像与之前产生的图像完全相同。这是因为还没有告诉 OpenGL 需要处理透明。OpenGL 中的透明是通过混合实现的。OpenGL 将已经绘制到帧缓冲区的像素与将要绘制到帧缓冲区的像素进行混合。

```
Gl.glEnable(Gl.GL_TEXTURE_2D);  
Gl.glBindTexture(Gl.GL_TEXTURE_2D, texture.Id);  
Gl.glEnable(Gl.GL_BLEND);  
Gl.glBlendFunc(Gl.GL_SRC_ALPHA, Gl.GL_ONE_MINUS_SRC_ALPHA);
```

首先需要启用混合。这可以在启用 2D 纹理模式后面的代码中进行设置。然后必须设置 blend 函数。blend 函数接受两个参数。第一个参数修改要绘制到帧缓冲区的像素的值，第二个参数修改将被覆盖的帧缓冲区中的像素的值。将绘制到帧缓冲区中的传入像素也叫做“源像素”。GL_SRC_ALPHA 是使用传入像素的 alpha 值的指令。GL_ONE_MINUS_SRC_ALPHA 是用 1 减去传入像素的 alpha 值的指令。这两个指令使用其 alpha 值将传入像素混合到帧缓冲区中。glBlendFunc 修改源像素和帧缓冲区像素的 R、G、B 的值，然后将这些值的和写入到帧缓冲区。

到目前为止给出的示例中，帧缓冲区的初始颜色均为黑色。每个像素都采用 RGBA 形式(0,0,0,1)。渲染的笑脸包含多种不同的像素类型，角落附近的像素是白色的，但是 alpha 值为 0。RGBA 值一般都是(1,1,1,0)。

当渲染了笑脸纹理的角落区域的像素后，它们的 alpha 值将为 0。1 减去源 alpha 的值为(1-0)，仍然为 1。计算新的帧缓冲区颜色的方式是，将传入像素与源像素的 alpha 相乘，然后将当前帧像素与 1 减去源像素的 alpha 值后得到的值相乘，然后将两个结果相加。

$$\begin{aligned} & \text{incomingRed} \times 0 + \text{frameBufferRed} \times 1 \\ & \text{incomingGreen} \times 0 + \text{frameBufferGreen} \times 1 \\ & \text{incomingBlue} \times 0 + \text{frameBufferBlue} \times 1 \end{aligned}$$

可以看到，使用这种混合时，传入的像素将被忽略，使笑脸的角落透明显示。一般的公式是：

$$(\text{incomingRGB} \times \text{incomingAlpha}) + (\text{framebufferRGB} \times (1 - \text{incomingAlpha}))$$

尝试计算这个公式的结果，就好像传入的 alpha 值是 1 或者 0.5。这就是 OpenGL 执行混合操作的方式。

运行程序后，得到的笑脸的白色背景已被删除。

6.5.5 颜色调制精灵

当前的精灵代码十分全面。设置精灵纹理、 (U, V) 映射、位置、宽度和高度都十分容易。对精灵进行处理的最常见的一种操作是修改颜色。在许多游戏中，文本可以修改颜色，图片可以闪烁为黄色，以吸引用户的注意。这通常是通过颜色调制完成的。

在创建旋转的三角形时，已经介绍了这种技术的基础。每个顶点都被赋予了一种颜色。如果精灵有一种颜色，然后所有的顶点都共享这种颜色，就更容易处理了。

只需要设置一次颜色，然后所有的顶点都将具有该颜色值。

```
float red = 1;
float green = 0;
float blue = 0;
float alpha = 1;

Gl.glBegin(Gl.GL_TRIANGLES);
{
    Gl.glColor4f(red, green, blue, alpha);
```

这段代码会使精灵显示为红色。修改 **alpha** 将影响整个精灵的透明度。使用这段代码时，很容易知道如何通过将 **alpha** 值从 0 逐渐修改为 1，实现淡入效果。

在某些情况下，给精灵提供渐变色可能比提供纯色更好。实现方法是，将底部的顶点设置为一种颜色，将顶部的像素设置为另外一种颜色。OpenGL 将会对颜色进行插值，并自动创建一种渐变。

6.5.6 Sprite 类和 Render 类

前面已经展示了精灵的基础框架，但现在的代码还不够整洁，而且很难重用。需要将所有的代码包装到类中，颜色、位置和 (U, V) 点数据都需要放到各自单独的类中。使用已经存在的 C# 类表达这些数据结构很理想，但更好的做法是创建自己的类，因为后边需要修改它们，以便它们能够正确地使用 OpenGL。

Sprite 类将包含精灵数据，但还需要一个单独的 **Renderer** 类来负责渲染精灵。以这种方式划分功能后，在以后就可以优化渲染代码。

下面给出了 3 个结构。

```
[StructLayout(LayoutKind.Sequential)]
public struct Vector
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }

    public Vector(double x, double y, double z) : this()
    {
```

```
        X = x;
        Y = y;
        Z = z;
    }

}

[StructLayout(LayoutKind.Sequential)]
public struct Point
{
    public float X { get; set; }
    public float Y { get; set; }

    public Point(float x, float y)
        : this()
    {
        X = x;
        Y = y;
    }
}

[StructLayout(LayoutKind.Sequential)]
public struct Color
{
    public float Red { get; set; }
    public float Green { get; set; }
    public float Blue { get; set; }
    public float Alpha { get; set; }

    public Color(float r, float g, float b, float a)
        : this()
    {
        Red = r;
        Green = g;
        Blue = b;
        Alpha = a;
    }
}
```

在游戏编程中，向量是一种很常用的工具。它们在概念上与 3D 空间中的位置不同，但是有相同的(x, y, z)值。向量一般用于表示位置，因为这样会使代码更加简单。这里的向量不包含任何方法，但是后面在研究向量的用途时，我们将会使其更加丰满。

向量结构和其他所有结构都将元数据附加到了[StructLayout(LayoutKind.Sequential)]

上,这就需要使用另外一个 `using` 语句。

```
using System.Runtime.InteropServices;
```

这条元数据告诉编译器,按照 C 程序语言的处理方式在内存中布局结构。这样就更容易与使用 C 语言编写的 OpenGL 库进行交互。

`Point` 结构将用于描述(U , V)坐标。纹理坐标并不需要双精度,所以使用了浮点数。`Color` 用于描述顶点的颜色。

`Renderer` 类如下所示。

```
using Tao.OpenGl;
using System.Runtime.InteropServices;

namespace GameStructure
{
    public class Renderer
    {
        public Renderer()
        {
            Gl.glEnable(Gl.GL_TEXTURE_2D);
            Gl.glEnable(Gl.GL_BLEND);
            Gl.glBlendFunc(Gl.GL_SRC_ALPHA, Gl.GL_ONE_MINUS_SRC_ALPHA);
        }

        public void DrawImmediateModeVertex(Vector position, Color color,
            Point uvs)
        {
            Gl.glColor4f(color.Red, color.Green, color.Blue, color.Alpha);
            Gl.glTexCoord2f(uvs.X, uvs.Y);
            Gl.glVertex3d(position.X, position.Y, position.Z);
        }

        public void DrawSprite(Sprite sprite)
        {
            Gl.glBegin(Gl.GL_TRIANGLES);
            {
                for (int i = 0; i < Sprite.VertexAmount; i++)
                {
                    Gl.glBindTexture(Gl.GL_TEXTURE_2D, sprite.Texture.Id);
                    DrawImmediateModeVertex(
                        sprite.VertexPositions[i],
                        sprite.VertexColors[i],
                        sprite.VertexUVs[i]);
                }
            }
        }
    }
}
```

```

    }
    Gl.glEnd();
}
}
}
}

```

在构造函数中，**Renderer** 设置相关的纹理和混合模式。现在这些操作在程序启动时执行一次，而不是像前面那样每帧执行一次。**DrawSprite** 函数负责渲染精灵。所有的 **OpenGL** 调用与之前的相同，一个调用设置纹理，一个调用设置颜色、位置和 *U*、*V* 坐标。

然后就只剩下 **Sprite** 类了。通过渲染器中使用精灵的方法，可以推断出它的不少函数。游戏通常有很多精灵，所以应该尽可能使类变得轻量级，只包含必不可少的成员。

```

public class Sprite
{
    internal const int VertexAmount = 6;
    Vector[] _vertexPositions = new Vector[VertexAmount];
    Color[] _vertexColors = new Color[VertexAmount];
    Point[] _vertexUVs = new Point[VertexAmount];
    Texture _texture = new Texture();

    public Sprite()
    {
        InitVertexPositions(new Vector(0,0,0), 1, 1);
        SetColor(new Color(1,1,1,1));
        SetUVs(new Point(0, 0), new Point(1, 1));
    }

    public Texture Texture
    {
        get { return _texture; }
        set
        {
            _texture = value;
            // By default the width and height is set
            // to that of the texture
            InitVertexPositions(GetCenter(), _texture.Width, _texture.
Height);
        }
    }

    public Vector[] VertexPositions
    {
        get { return _vertexPositions; }
    }
}

```



```
}

public Color[] VertexColors
{
    get { return _vertexColors; }
}

public Point[] VertexUVs
{
    get { return _vertexUVs; }
}

private Vector GetCenter()
{
    double halfWidth = GetWidth() / 2;
    double halfHeight = GetHeight() / 2;

    return new Vector(
        _vertexPositions[0].X + halfWidth,
        _vertexPositions[0].Y - halfHeight,
        _vertexPositions[0].Z);
}

private void InitVertexPositions(Vector position, double width, double height)
{
    double halfWidth = width / 2;
    double halfHeight = height / 2;
    // Clockwise creation of two triangles to make a quad.

    // TopLeft, TopRight, BottomLeft
    _vertexPositions[0] = new Vector(position.X - halfWidth, position.Y + halfHeight, position.Z);
    _vertexPositions[1] = new Vector(position.X + halfWidth, position.Y + halfHeight, position.Z);
    _vertexPositions[2] = new Vector(position.X - halfWidth, position.Y - halfHeight, position.Z);

    // TopRight, BottomRight, BottomLeft
    _vertexPositions[3] = new Vector(position.X + halfWidth, position.Y + halfHeight, position.Z);
    _vertexPositions[4] = new Vector(position.X + halfWidth, position.Y - halfHeight, position.Z);
    _vertexPositions[5] = new Vector(position.X - halfWidth, position.Y -
```

```
halfHeight, position.Z);
    }

    public double GetWidth()
    {
        // topright - topleft
        return _vertexPositions[1].X - _vertexPositions[0].X;
    }

    public double GetHeight()
    {
        // topleft - bottomleft
        return _vertexPositions[0].Y - _vertexPositions[2].Y;
    }

    public void SetWidth(float width)
    {
        InitVertexPositions(GetCenter(), width, GetHeight());
    }

    public void SetHeight(float height)
    {
        InitVertexPositions(GetCenter(), GetWidth(), height);
    }

    public void SetPosition(double x, double y)
    {
        SetPosition(new Vector(x, y, 0));
    }

    public void SetPosition(Vector position)
    {
        InitVertexPositions(position, GetWidth(), GetHeight());
    }

    public void SetColor(Color color)
    {
        for (int i = 0; i < Sprite.VertexAmount; i++)
        {
            _vertexColors[i] = color;
        }
    }
}
```

```

public void SetUVs(Point topLeft, Point bottomRight)
{
    // TopLeft, TopRight, BottomLeft
    _vertexUVs[0] = topLeft;
    _vertexUVs[1] = new Point(bottomRight.X, topLeft.Y);
    _vertexUVs[2] = new Point(topLeft.X, bottomRight.Y);

    // TopRight, BottomRight, BottomLeft
    _vertexUVs[3] = new Point(bottomRight.X, topLeft.Y);
    _vertexUVs[4] = bottomRight;
    _vertexUVs[5] = new Point(topLeft.X, bottomRight.Y);
}
}

```

由于包含了访问器函数和一些重载的函数，Sprite 类很庞大。它的默认构造函数使用空纹理创建了一个 1×1 的精灵。设置好纹理后，精灵的宽度和高度会自动设置为纹理值。创建精灵后，可以根据需要修改位置、尺寸、纹理和 U 、 V 坐标。

6.5.7 使用 Sprite 类

在把精灵代码打包到类中以后，接下来介绍如何使用该类。创建一个新的游戏状态 TestSpriteClassState，将其加载到状态系统中，使其成为当程序运行时运行的默认状态。TestSpriteClassState 需要接受 TextureManager 作为构造函数的参数，就如 DrawSpriteState 一样。

```

Renderer _renderer = new Renderer();
TextureManager _textureManager;
Sprite _testSprite = new Sprite();
Sprite _testSprite2 = new Sprite();

public TestSpriteClassState(TextureManager textureManager)
{
    _textureManager = textureManager;
    _testSprite.Texture = _textureManager.Get("face_alpha");
    _testSprite.SetHeight(256*0.5f);

    _testSprite2.Texture = _textureManager.Get("face_alpha");
    _testSprite2.SetPosition(-256, -256);
    _testSprite2.SetColor(new Color(1, 0, 0, 1));
}

public void Render()
{
    Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

```

```
Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);  
_renderer.DrawSprite(_testSprite);  
_renderer.DrawSprite(_testSprite2);  
Gl.glFinish();  
}
```

这个状态渲染两个不同的精灵：一个精灵被挤压，另一个精灵偏离了屏幕中心，并显示为红色。这段代码简单易用。主要的工作都在 **Sprite** 类和 **Renderer** 类中完成。这两个类只编写了一次，但是可以在任何位置使用，所以使它们尽可能地友好易用是非常有必要的。

通过随着时间流逝修改(U,V)坐标或者修改纹理，可以使这些精灵运动起来。位置也可以随时间改变。这应该在更新循环中完成。自由地进行研究，看看可以实现什么效果。如果不确定从哪里入手，可以继续阅读后面的内容，在学习如何创建游戏的过程中，这些概念将变得越来越明显。

第 7 章

渲染文本

大多数游戏都需要文本，因为至少它们都需要显示玩家得分或者菜单。

文本渲染也可以在游戏进行中显示变量的值。创建一个简单的字体系统并不困难，每个字母和数字都表示一个精灵。然后文本字符串将被转换成一个精灵列表。这种方法很容易扩展到其他语言中，不过亚洲语言(例如中文)需要更多的纹理来表示不同的笔画。

7.1 字体纹理

图 7-1 所示的是一个包含整个罗马字母表、数字和一些标点符号的纹理。可以把这个纹理作为绘制文本的理想基础。如果使用的工具正确，创建这种纹理并不困难。Andreas Jönsson 的 Bitmap Font Generator 是一个用于生成位图字体的出色工具，可以从 Internet 上或者本书配套光盘的 Apps 文件夹中找到。它可以把 True Type 字体转换成可以在游戏中高效渲染的位图。图 7-1 所示的纹理就是以这种方式创建的。从本书配套光盘的 Assets 文件夹中可以找到这个字体纹理，以及描述该纹理的笔画的(U, V)坐标的一个数据文件。

为了显示这种字体，我创建了一个新项目，其中包含一个新的游戏状态 TextTestState。我将字体纹理作为 font.tga 添加到了项目中(从配套光盘的 Assets 文件夹中可以找到该纹理)，还使用 font 这个 id 把它添加到了纹理管理器中。创建一个新纹理，将其 font 纹理设置为该纹理，然后在渲染循环中渲染字体。

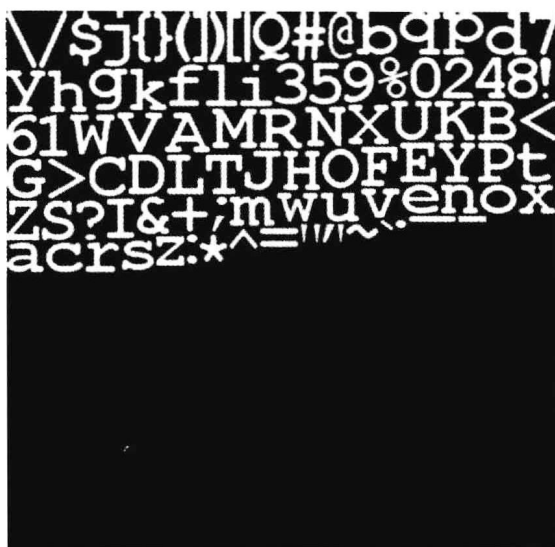


图 7-1 一个字体纹理

```
class TextTestState : IGameObject
{
    Sprite _text = new Sprite();
    Renderer _renderer = new Renderer();
    public TextTestState(TextureManager textureManager)
    {
        _text.Texture = textureManager.Get("font");
    }
    public void Render()
    {
        Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        _renderer.DrawSprite(_text);
    }
    public void Update(double elapsedTime)
    {
    }
}
```

这会在黑色背景上把整个字体显示为白色。修改清除色，以确认透明度可以正确工作。然后取消对包含 `SetColor` 的代码行的注释。这会使字体变成黑色。使用精灵代码很容易修改字体颜色。

要渲染单个字符，必须修改精灵的 UV 设置。尝试输入下面的 UV 信息。

```
_text.Texture = textureManager.Get("font");
_text.SetUVs(new Point(0.113f, 0), new Point(0.171f, 0.101f));
```

这会生成一个很大的美元符号。美元符号非常大，因为精灵的大小仍然被设为 256 像素×256 像素。使美元符号看起来很自然的分辨率是 15 像素×26 像素。修改宽度和高度，得到的结果应该如图 7-2 所示。

```
_text.SetUVs(new Point(0.113f, 0), new Point(0.171f, 0.101f));  
_text.SetWidth(15);  
_text.SetHeight(26);
```

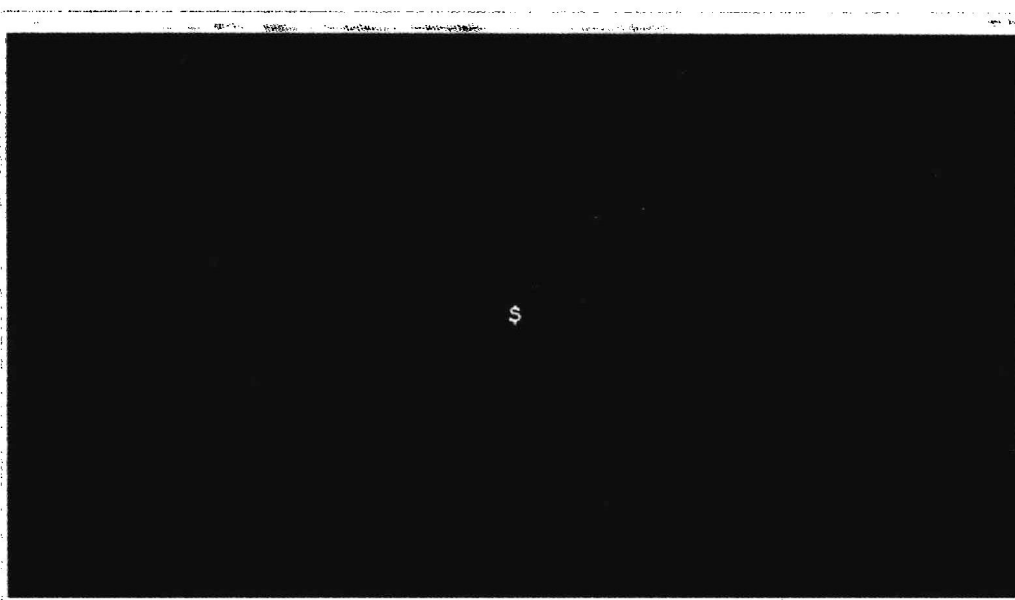


图 7-2 单个字符

这个 UV 数据在美元符号周围形成了一个小方框，它可以清晰地显示出来，如图 7-3 所示。关于每个字符在纹理映射中的位置的信息是由 **Bitmap Font Program** 在生成纹理的同时生成的。可以使用这些信息来获取正确的 UV 坐标。



图 7-3 单个字符的 UV 信息

7.2 字体数据

字体数据是一个简单的文本文件，用于标识纹理中的全部字符。本书的配套光盘中提供了字体数据，需要采用添加纹理的方式把它添加到字体项目中。记住要设置它的属性，以确保将它复制到生成目录中。下面显示了数据文件的前几行。

```
info face="Courier New" size=-32 bold=1 italic=0 charset="" unicode=1
stretchH=100 smooth=1 aa=1 padding=0,0,0,0 spacing=1,1 outline=0
common lineHeight=36 base=26 scaleW=256 scaleH=256 pages=1 packed=0
alphaChnl=0 redChnl=0 greenChnl=0 blueChnl=0
page id=0 file="font_0.tga"
chars count=95
char id=32 x=253 y=21 width=1 height=1 xoffset=0 yoffset=26
xadvance=19 page=0 chnl=15
char id=33 x=247 y=21 width=5 height=20 xoffset=7 yoffset=6
xadvance=19 page=0 chnl=15
char id=34 x=136 y=101 width=9 height=9 xoffset=4
```

前3行是头信息，可以忽略。它们包含了描述字体的信息：字体大小为32，类型为Courier New。我们的字体系统并不需要知道这些信息。

第4行描述了字体文件中包含多少个字符，在这里共有95个字符。在这一行之后，可以看到每个字符的信息，字符在纹理中的像素位置，以及像素的宽度和高度。

在接连渲染一个词中的几个字符时，`xoffset` 和 `yoffset` 用于对齐字符。`y` 字符的偏移量比1更大。`xadvance` 参数表示渲染了当前字符后，在 x 轴上的前进量。我们将只使用具有一个纹理的字体，这样就可以安全地忽略 `page` 值。通道信息也可以忽略，只是有些时候，在使用压缩技术时会将不同的字符写入不同的颜色通道中，此时就需要使用通道信息。

需要把字符读入合适的类中。

```
public class CharacterData
{
    public int Id { get; set; }
    public int X { get; set; }
    public int Y { get; set; }
    public int Width { get; set; }
    public int Height { get; set; }
    public int XOffset { get; set; }
    public int YOffset { get; set; }
    public int XAdvance { get; set; }
}
```

`CharacterData` 类将我们感兴趣的参数简单地集合到了一起。在数据文件中，每个字符都将有一个 `CharacterData` 对象。

7.2.1 解析字体数据

`CharacterData` 类将存储在字典中。字典的键是它们代表的字符，如下面的代码所示。

```
CharacterData aData = _characterDictionary['a'];
```

给定一个字符串后，迭代所有字符并获得每个字符的相关数据将是十分简单的。为了使用数据文件填充字典，需要使用解析器。下面显示了一个最简单的解析器的代码。当向它提供一个字体数据文件的路径后，它将返回一个填充了字符数据的字典。

```
public class FontParser
{
    static int HeaderSize = 4;
    // Gets the value after an equal sign and converts it
    // from a string to an integer
    private static int GetValue(string s)
    {
        string value = s.Substring(s.IndexOf('=') + 1);
        return int.Parse(value);
    }
    public static Dictionary<char, CharacterData> Parse(string filePath)
    {
        Dictionary<char, CharacterData> charDictionary = new
        Dictionary<char, CharacterData>();
        string[] lines = File.ReadAllLines(filePath);
        for(int i = HeaderSize; i < lines.Length; i+=1)
        {
            string firstLine = lines[i];
            string[] typesAndValues = firstLine.Split(" ".ToCharArray(),
                StringSplitOptions.RemoveEmptyEntries);
            // All the data comes in a certain order,
            // used to make the parser shorter
            CharacterData charData = new CharacterData
            {
                Id = GetValue(typesAndValues[1]),
                X = GetValue(typesAndValues[2]),
                Y = GetValue(typesAndValues[3]),
                Width = GetValue(typesAndValues[4]),
                Height = GetValue(typesAndValues[5]),
                XOffset = GetValue(typesAndValues[6]),
                YOffset = GetValue(typesAndValues[7]),
                XAdvance = GetValue(typesAndValues[8])
            };
            charDictionary.Add((char)charData.Id, charData);
        }
    }
}
```

```

        return charDictionary;
    }
}

```

这个解析器很简单，而且没有进行任何错误检查或验证。必须包含 `using System.IO` 语句，然后才能从硬盘上读取文本文件。每个 `CharacterData` 结构都被填充，然后它的 `Id` 将被强制转换为一个用作索引的字符。`Id` 是代表该字符的 ASCII 数值，将数值强制转换为 C# 的 `char` 类型将把它转换为正确的字符。

7.2.2 使用 CharacterData

下面从较高的层面上展示了文本系统的工作原理。代码中创建了一个新状态，以演示字体系统的应用。

```

class TextRenderState : IGameObject
{
    TextureManager _textureManager;
    Font _font;
    Text _helloWorld;
    Renderer _renderer = new Renderer();
    public TextRenderState(TextureManager textureManager)
    {
        _textureManager = textureManager;
        _font = new Font(textureManager.Get("font"),
            FontParser.Parse("font.fnt"));
        _helloWorld = new Text("Hello", _font);
    }
    public void Render()
    {
        Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        _renderer.DrawText(_helloWorld);
    }
    public void Update(double elapsedTime)
    {
    }
}

```

这里使用了两个新类。首先是一个 `Font` 类，它确定了要使用的字体。`Font` 类包含对字体纹理和字符数据的引用。第二个类是 `Text` 类，用于渲染文本。可以加载几种不同的字体，在它们之间进行切换十分简单。

`Renderer` 中新加了一个额外的方法 `DrawText`。`DrawText` 接受一个文本对象作为参数，并使用该对象渲染文本。文本类只是一个精灵的集合，所以渲染器可以重用它的 `DrawSprite` 代码。

还有一个类没有包含在示例中，即 `CharacterSprite` 类，它用于代表位图文本字符串中的各个字符。`CharacterSprite` 类只有两个成员：一个是代表字母的精灵，另外一个为 `CharacterData` 类，后者包含关于字符以及精灵的使用方式的信息。

```
public class CharacterSprite
{
    public Sprite Sprite {get; set;}
    public CharacterData Data { get; set; }
    public CharacterSprite(Sprite sprite, CharacterData data)
    {
        Data = data;
        Sprite = sprite;
    }
}
```

`Text` 类是 `CharacterSprite` 的一个列表，它负责排列字母的顺序。给定一个简单的文本字符串以后，它为每个字符创建一个 `CharacterSprite`，然后排列它们的顺序。`Text` 还处理正确的偏移量。代码如下。

```
public class Text
{
    Font _font;
    List<CharacterSprite> _bitmapText = new List<CharacterSprite>();
    string _text;
    public List<CharacterSprite> CharacterSprites
    {
        get { return _bitmapText; }
    }
    public Text(string text, Font font)
    {
        _text = text;
        _font = font;
        CreateText(0, 0);
    }
    private void CreateText(double x, double y)
    {
        _bitmapText.Clear();
        double currentX = x;
        double currentY = y;
        foreach (char c in _text)
        {
            CharacterSprite sprite = _font.CreateSprite(c);
            float xOffset = ((float)sprite.Data.XOffset) / 2;
            float yOffset = ((float)sprite.Data.YOffset) / 2;
```

```

        sprite.Sprite.SetPosition(currentX + xOffset, currentY - yOffset);
        currentX += sprite.Data.XAdvance;
        _bitmapText.Add(sprite);
    }
}
}

```

这个类非常简单。**CreateText** 函数是类的核心所在，它将字符放在正确的位置。对于每个精灵，检查其 **CharacterData**，然后按指定的数量移动 x 个位置。每个字符精灵也有一个偏移量。所有的精灵都根据它们的中心进行定位，但是偏移量从左上角开始计算。为了将这个偏移量转换为相对中央位置的偏移量，需要将偏移值减半。因为偏移量是整数，所以在执行减半操作时需要将其转换成浮点数。如果没有将数值转换成浮点数，那么浮点信息将被丢弃，文本字符将无法正确地对齐。

最后要考虑的一个类是 **Font** 类。**Font** 类保存一个将字符转换成 **CharacterData** 对象的字典。给定一个词或句子，所有的字母都可以用于索引字典，然后将返回一组 **CharacterData**，此时就可以使用它们创建文本精灵。

```

public class Font
{
    Texture _texture;
    Dictionary<char, CharacterData> _characterData;
    public Font(Texture texture, Dictionary<char, CharacterData>
characterData)
    {
        _texture = texture;
        _characterData = characterData;
    }
    public CharacterSprite CreateSprite(char c)
    {
        CharacterData charData = _characterData[c];
        Sprite sprite = new Sprite();
        sprite.Texture = _texture;
        // Setup UVs
        Point topLeft = new Point((float)charData.X / (float)_texture.Width,
                                   (float)charData.Y / (float)_texture.Height);
        Point bottomRight = new Point( topLeft.X + ((float)charData.Width /
(float)_texture.Width),
                                       topLeft.Y + ((float)charData.Height /
(float)_texture.Height));
        sprite.SetUVs(topLeft, bottomRight);
        sprite.SetWidth(charData.Width);
        sprite.SetHeight(charData.Height);
        sprite.SetColor(new Color(1, 1, 1, 1));
    }
}

```

```

        return new CharacterSprite(sprite, charData);
    }
}

```

提供的(U,V)坐标以像素计量,但是 OpenGL 纹理的索引范围为 0~1。通过把像素坐标的 x 和 y 值除以纹理的宽度和高度,可以把像素值转换为 OpenGL 坐标。CharacterData 数值都存储为整数,需要把它们强制转换为浮点数,这样做除法之后得到的结果中就会保留小数位。精灵的高度和宽度通过 CharacterData 信息设置,颜色默认被设为白色。创建了精灵后,将使它成为一个 CharacterSprite,然后返回这个类。

7.3 渲染文本

字体代码现在可用了。文本的一个直接应用是显示 fps(frames per second, 每秒帧数)。通过 fps 可以知道游戏代码的运行速度。每秒帧数用于衡量每秒执行游戏循环的次数。现代游戏实现的帧率一般在 30~60fps。每秒帧数并不是决定流畅的图形显示的唯一一个因素,一致的帧率也很重要。假如游戏的帧率一般在 60fps 左右,但有时候会掉到 30fps,图像的显示就不如总是运行在 30fps 的游戏图像流畅。

在项目中创建并添加一个新的游戏状态。我选择了 FPSTestState,但是名称并不重要。确保将它添加到 StateSystem,并使它成为默认加载的第一个状态。这个状态要求将 TextureManager 传入到构造函数中来创建字体对象。下面所示为渲染文本的代码。

```

class FPSTestState : IGameObject
{
    TextureManager _textureManager;
    Font _font;
    Text _fpsText;
    Renderer _renderer = new Renderer();
    public FPSTestState(TextureManager textureManager)
    {
        _textureManager = textureManager;
        _font = new Font(textureManager.Get("font"),
            FontParser.Parse("font.fnt"));
        _fpsText = new Text("FPS:", _font);
    }
    #region IGameObject Members
    public void Render()
    {
        Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        _renderer.DrawText(_fpsText);
    }
}

```

```
public void Update(double elapsedTime)
{
}
#endregion
}
```

在测试代码前，需要编写 **Renderer** 的 **DrawText** 调用。这会将文本绘制到屏幕的中央，就现在而言这没有问题。**DrawText** 方法可遍历文本并绘制每个精灵。

```
public void DrawText(Text text)
{
    foreach (CharacterSprite c in text.CharacterSprites)
    {
        DrawSprite(c.Sprite);
    }
}
```

在把这个方法添加到渲染器以后，运行代码将在屏幕上渲染出文本“FPS:”，如图 7-4 所示。

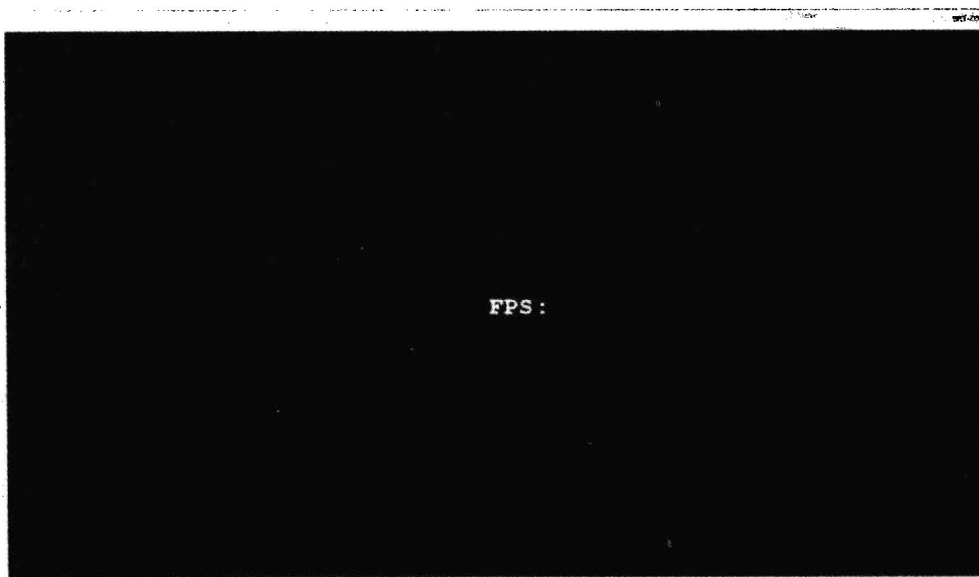


图 7-4 渲染出的文本“FPS:”

7.3.1 计算 FPS

计算每秒帧数很简单。统计出每秒中帧的数量，然后把这个数值显示到屏幕上。帧数越高，游戏运行得越快。**FPS** 是一个应该显示在屏幕上的有价值的统计数据，这样在开发游戏的过程中就很容易注意到在添加某种特性后，**fps** 是不是突然下降。这样在开发的早期就可以知道并避免低级的错误。

为了在每次游戏循环中知道帧的数量，可以将一个变量 `_numberOfFrames` 加 1。更新循环中的 `elapsedTime` 指出每一帧用了多长时间，将所有的 `elapsedTime` 值相加，就可以知道经过了多长时间。经过 1s 后，`_numberOfFrames` 就是在 1s 内渲染的帧数。可以将这些代码包装到一个类中，如下所示。

```
public class FramesPerSecond
{
    int _numberOfFrames = 0;
    double _timePassed = 0;
    public double CurrentFPS { get; set; }
    public void Process(double timeElapsed)
    {
        _numberOfFrames++;
        _timePassed = _timePassed + timeElapsed;
        if (_timePassed > 1)
        {
            CurrentFPS = (double)_numberOfFrames / _timePassed;
            _timePassed = 0;
            _numberOfFrames = 0;
        }
    }
}
```

这个类计算每秒的帧数。必须在每帧中调用它的处理方法。将 `FramesPerSecond` 添加到 `FPSTestState` 中，这样就可以使用 `Text` 类在屏幕上显示每秒帧数。

```
class FPSTestState : IGameObject
{
    TextureManager _textureManager;
    Font _font;
    Text _fpsText;
    Renderer _renderer = new Renderer();
    FramesPerSecond _fps = new FramesPerSecond();
    // Constructor and Render have been ommitted.
    public void Update(double elapsedTime)
    {
        _fps.Process(elapsedTime);
    }
}
```

当运行此状态时，就会记录 `fps`。一般来说，`FramesPerSecond` 类不会存在于 `FramePerSecond` 类中，而是很可能在 `Form` 类中。本例中，在 `FPSTestState` 中更容易测试 `FramesPerSecond`。

```
public void Render()
```

```
{  
    Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);  
    _fpsText = new Text("FPS: " + _fps.CurrentFPS.ToString("00.0"), _font);  
    _renderer.DrawText(_fpsText);  
}
```

渲染循环会在渲染“FPS:”后显示被转换为字符串的 fps 值。在 ToString 方法中还提供了一些格式信息,这会使代表 fps 的 double 类型在被转换为字符串后只有一个小数位,而且在小数点前有两个或更多个数位。

运行程序,看看自己的帧率是多少。不同计算机的帧率相差很大。这个程序中没有进行多少处理操作,所以帧率会很高。图 7-5 显示了运行该程序时的输出。

最初,我的计算机上显示的帧率为大约 60fps,这是因为我的显示器设置中打开了垂直同步(V-Sync)。关闭垂直同步后得到的数字可以更好地指示每秒的帧数。

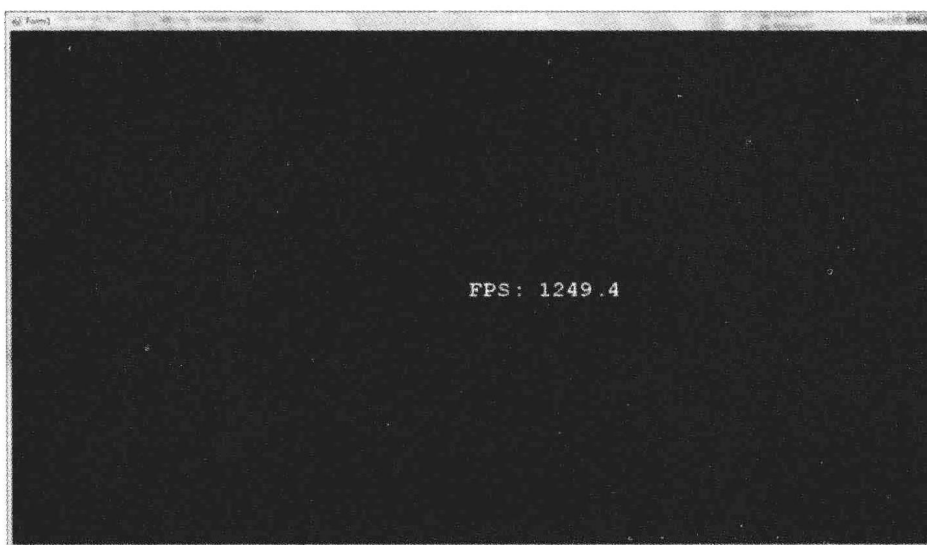


图 7-5 一个 fps 计数器

7.3.2 垂直同步和帧率

计算机屏幕每秒刷新一定的次数。垂直同步这个选项确保只以屏幕能够读取的速度填充帧缓冲区。这可以避免屏幕上出现图像失真,例如撕裂,当把数据写入到引起可见的撕裂效果的屏幕上时,帧缓冲区就会发生改变。

在一些图形卡中,默认是打开垂直同步的。垂直同步是指显示器的刷新率(显示器更新其显示的频率)。如果显示器以 60Hz 的频率刷新,且垂直同步已被打开,那么 fps 计数器不会超过 60fps。多数情况下这都没有问题,但是在开发游戏和分析帧率时,不锁定帧率是非常重要的。通常可以通过显卡设置关闭垂直同步,但是不同显卡之间禁用垂直同步的具体方法是不同的。

7.3.3 性能分析

fps 计数器可以用于一些基本的性能分析。现在游戏渲染大约 10 个带有颜色信息(fps 文本)的纹理四方形。2D 游戏可能在每个区块中都使用一个四方形,并为玩家和游戏敌人使用多个四方形。假设粒子系统中包含 10 000 个四方形。要同时在屏幕上显示的话,这个数目不算小了,而且对于大多数游戏来说这么多四方形也足够了。可以执行一个不算太完善的测试来查看当前的精灵是否可以处理这么多的四方形。

```
renderer.DrawText(_fpsText);  
for (int i = 0; i < 1000; i++)  
{  
    _renderer.DrawText(_fpsText);  
}
```

这将渲染 fps 文本 1000 次,也就是总共大约 10 000 个四方形。在我的计算机上, fps 从超过 1000 降到了刚刚超过 30。对于大多数 2D 游戏, 30fps 没有什么问题。这意味着当前精灵代码的效率对于大多数 2D 游戏完全可以接受。图形卡较老的计算机可能就没这么幸运了,所以在本章后面将介绍一些提高效率的措施。

7.4 优化 Text 类

现在 Text 类的功能已经很完善了,但是再添加一些使它更容易使用的方法有益无害。例如,还没有设置文本字符串位置的方法。另外,确实需要有一个测量文本的方法,以便可以对齐字符串。通常还需要把文本按列对齐,这可以通过为文本提供一个最大宽度实现。如果文本超过这个宽度,则将换行。这是一个非常不错的特性,在游戏中填写文本框时更是如此。

下面是设置 Text 位置的辅助方法。

```
public void SetPosition(double x, double y)  
{  
    CreateText(x, y);  
}
```

为了重新定位文本,需要重新计算四方形。这并不是最优方式,但是进行编码很容易,而且不太可能导致在游戏编程中出现瓶颈。

修改整条文本的颜色的函数也需要更加简化。

```
public void SetColor(Color color)  
{  
    _color = color;  
    foreach (CharacterSprite s in _bitmapText)  
    {  
        s.Sprite.SetColor(color);  
    }  
}
```

```
}  
}
```

在这段代码中, **Text** 有一个颜色成员, 用于存储文本的当前颜色。调用 **CreateText** 时, 所有的顶点都将被重新创建, 包括颜色分量。把当前颜色存储到 **Text** 类以后, 在重新创建顶点时可以保留 **Text** 的颜色。代码中添加了一个重载的 **SetColor** 函数(不需要颜色参数)供 **CreateText** 函数使用。

```
public void SetColor()  
{  
    foreach (CharacterSprite s in _bitmapText)  
    {  
        s.Sprite.SetColor(_color);  
    }  
}
```

在 **CreateText** 函数的结尾, 需要添加额外的一行。

```
SetColor();
```

尝试对齐屏幕上的文本时, 文本的宽度和高度非常重要, 因此, 应该有一种使用像素测量文本字符串的方法。**Font** 类中的 **MeasureText** 方法可以提供此功能。

```
public Vector MeasureFont(string text)  
{  
    return MeasureFont(text, -1);  
}  
public Vector MeasureFont(string text, double maxWidth)  
{  
    Vector dimensions = new Vector();  
    foreach (char c in text)  
    {  
        CharacterData data = _characterData[c];  
        dimensions.X += data.XAdvance;  
        dimensions.Y = Math.Max(dimensions.Y, data.Height + data.YOffset);  
    }  
    return dimensions;  
}
```

代码中有两个 **MeasureFont** 方法: 第一个方法是重载后的方法, 不需要最大宽度作为参数; 第二个方法是实际进行测量的地方。

第二个 **MeasureFont** 方法返回的向量包含了以 *X* 值和 *Y* 值表示的宽度和高度。*Z* 分量未被使用。这些信息作为向量返回, 而不是其他某种数据结构(例如 **PointF**), 这是因为向量存储 **double** 类型, 位置也正是存储为 **double** 类型。宽度和高度主要用于修改文本的其他部分或精灵的位置, **double** 类型意味着不需要进行强制转换。向量也更容易扩展和变换。

文本的测量方式是，迭代字符数据并累加 x 轴上的前进量来计算整个字符串的宽度。字符串的高度就是最高的字符的高度。

我们没有在每次需要时计算宽度和高度，因为在 `Text` 类中存储这些尺寸信息更加方便。

```
public class Text
{
    Font _font;
    List<CharacterSprite> _bitmapText = new List<CharacterSprite>();
    string _text;
    Vector _dimensions;
    public double Width
    {
        get { return _dimensions.X; }
    }
    public double Height
    {
        get { return _dimensions.Y; }
    }
}
```

每次改变文本时，都需要更新尺寸成员。代码中只有一个位置的文本改变了：`CreateText` 方法。

```
private void CreateText(double x, double y)
{
    _bitmapText.Clear();
    double currentX = x;
    double currentY = y;
    foreach (char c in _text)
    {
        CharacterSprite sprite = _font.CreateSprite(c);
        float xOffset = ((float)sprite.Data.XOffset) / 2;
        float yOffset = ((float)sprite.Data.YOffset) / 2;
        sprite.Sprite.SetPosition(currentX + xOffset, currentY - yOffset);
        currentX += sprite.Data.XAdvance;
        _bitmapText.Add(sprite);
    }
    _dimensions = _font.MeasureFont(_text);
    SetColor();
}
```

在最后新添加了一行，用以测量字符串的大小。为了确认代码可以工作，尝试居中 `fps` 文本，或者在一行或者一列中两次渲染 `fps` 文本。

最后要添加到 `Text` 类中的功能是设置最大宽度的能力。这将提供长文本换行的功能。

当尝试将文本放到文本框或者确保文本不会超出屏幕的边缘时，这种特性非常有用。尝试自己弄明白算法需要怎么做才能使用最大宽度格式化文本。

- 将文本划分为词。
- 获取文本中的下一个词。
- 测量词的长度。
- 如果当前长度大于最大宽度，则开始一个新行。

Text 类需要重新定义 CreateText 方法来处理最大宽度参数。

```
private void CreateText(double x, double y)
{
    CreateText(x, y, _maxWidth);
}
private void CreateText(double x, double y, double maxWidth)
{
    _bitmapText.Clear();
    double currentX = 0;
    double currentY = 0;
    string[] words = _text.Split(' ');
    foreach (string word in words)
    {
        Vector nextWordLength = _font.MeasureFont(word);
        if (maxWidth != -1 &&
            (currentX + nextWordLength.X) > maxWidth)
        {
            currentX = 0;
            currentY += nextWordLength.Y;
        }
        string wordWithSpace = word + " "; // add the space character that was
        removed.
        foreach (char c in wordWithSpace)
        {
            CharacterSprite sprite = _font.CreateSprite(c);
            float xOffset = ((float)sprite.Data.XOffset) / 2;
            float yOffset = (((float)sprite.Data.Height) * 0.5f) + ((float)
            sprite.Data.YOffset);
            sprite.Sprite.SetPosition(x + currentX + xOffset, y - currentY -
            yOffset);
            currentX += sprite.Data.XAdvance;
            _bitmapText.Add(sprite);
        }
    }
    _dimensions = _font.MeasureFont(_text, _maxWidth);
    _dimensions.Y = currentY;
}
```

```
SetColor(_color);  
}
```

这段代码依赖于 `_maxWidth` 成员。如果 `_maxWidth` 等于 -1，那么不会进行换行。否则，在超过 `maxWidth` 值中指定的像素数后，文本将进行换行。下面是另外一个构造函数，它接受最大宽度作为参数。

```
int _maxWidth = -1;  
public Text(string text, Font font) : this(text, font, -1) { }  
public Text(string text, Font font, int maxWidth)  
{  
    _text = text;  
    _font = font;  
    _maxWidth = maxWidth;  
    CreateText(0, 0, _maxWidth);  
}
```

图 7-6 显示了使用新的 `maxWidth` 参数后换行的文本，这是使用下面的代码生成的。

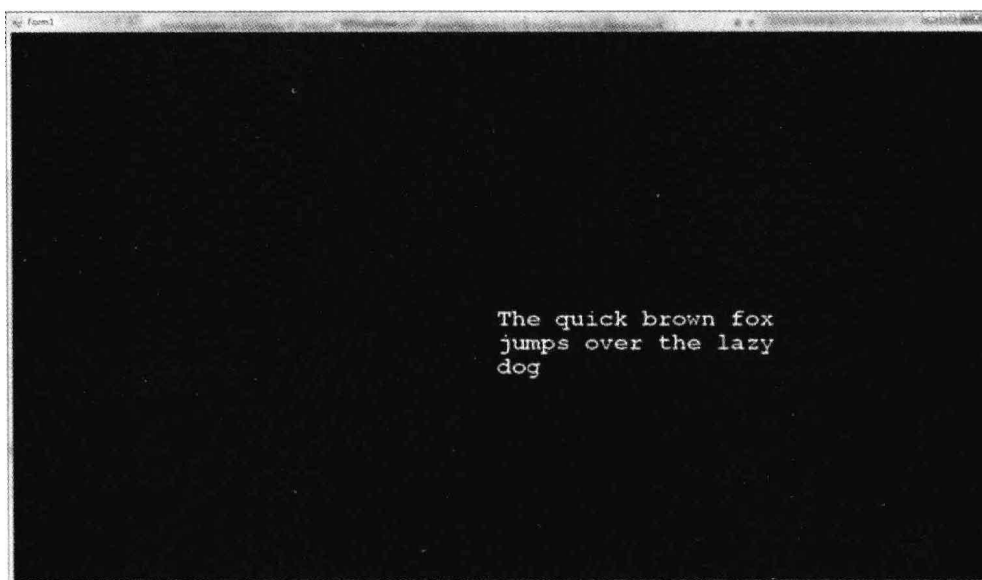


图 7-6 文本换行

```
Text longText = new Text("The quick brown fox jumps over the lazy dog",  
    _font, 400);  
_renderer.DrawText(longText);
```

这段文本换行代码没有考虑换行符或者制表符，但是扩展这段代码并不困难。

现在已经添加了使 `Text` 类非常易用的全部基本功能。另外一个有必要添加的特性是缩放功能，以便根据给定的量缩放文本。

```
longText.SetScale(0.5); // this would halve the text size
```

也可以扩展 `MeasureText` 方法，使其接受一个 `maxWidth` 参数。这样的话，即使文本会换行，也可以正确地计算宽度和高度。

7.5 使用 `glDrawArrays` 进行快速渲染

我们已经展示了使用帧率对 2D 游戏进行性能分析，当前的代码还不错。但是只需要对其进行一些小改动，性能就可以得到显著地提升。

```
public void DrawSprite(Sprite sprite)
{
    Gl.glBegin(Gl.GL_TRIANGLES);
    {
        for (int i = 0; i < Sprite.VertexAmount; i++)
        {
            Gl.glBindTexture(Gl.GL_TEXTURE_2D, sprite.Texture.Id);
            DrawImmediateModeVertex(
                sprite.VertexPositions[i],
                sprite.VertexColors[i],
                sprite.VertexUVs[i]);
        }
    }
    Gl.glEnd();
}
```

这是当前的渲染代码，它针对每一个精灵执行一次。上面的代码的问题是，每次调用 `glEnd` 时，所有数据都被发送到图形卡，然后 CPU 会暂停运行，直到图形卡发回一条消息，指出它接收了全部顶点以前，CPU 不会进行任何处理。处理 10 000 个精灵时，等待的时间会比较长。绘图调用越多，游戏运行越慢。

这个问题的解决办法是，尽可能多地同时绘制精灵。这种做法一般叫做批处理(batching)。不是让 CPU 告诉 GPU，“绘制这个精灵，现在绘制这个精灵，现在绘制这个精灵”，而是创建所有精灵的一个列表，然后告诉 GPU，“绘制所有这些精灵”。等待的时间减少了很多，所以代码运行速度快了很多。如果代码以这种方式组织，只需要很少的修改就可以实现这种性能提升。

需要使用一个新类收集所有的顶点信息，然后把这些信息发送到图形卡。`Batch` 很适合作为这个类的名称。该类将接受精灵，然后在一些较大的数组中打包所有的顶点信息。然后它提供这些数组的 `OpenGL` 指针，并发送一个绘制命令。这个新类将使用 `Tao` 框架，所以记得要包含合适的 `using` 语句。

```
public class Batch
{
```

```

const int MaxVertexNumber = 1000;
Vector[] _vertexPositions = new Vector[MaxVertexNumber];
Color[] _vertexColors = new Color[MaxVertexNumber];
Point[] _vertexUVs = new Point[MaxVertexNumber];
int _batchSize = 0;
public void AddSprite(Sprite sprite)
{
    // If the batch is full, draw it, empty and start again.
    if (sprite.VertexPositions.Length + _batchSize > MaxVertexNumber)
    {
        Draw();
    }
    // Add the current sprite vertices to the batch.
    for (int i = 0; i < sprite.VertexPositions.Length; i++)
    {
        _vertexPositions[_batchSize + i] = sprite.VertexPositions[i];
        _vertexColors[_batchSize + i] = sprite.VertexColors[i];
        _vertexUVs[_batchSize + i] = sprite.VertexUVs[i];
    }
    _batchSize += sprite.VertexPositions.Length;
}
}

```

最大顶点数是批(batch)在告诉 OpenGL 渲染它包含的所有顶点之前, 可以变得多大。接下来的成员是描述所有顶点信息的数组: 顶点的位置、颜色、和(U,V)坐标。_batchSize 成员跟踪批(batch)的大小, 需要把这个信息传递给 OpenGL 以绘制数组。还需要把它与最大顶点数进行比较, 以确定何时绘制批(batch)。

批(batch)使用 AddSprite 函数收集精灵。AddSprite 首先检查添加一个精灵会不会使批变得过大。如果答案是肯定的, 那么它强制绘制当前批(batch), 然后清空批(batch)。在这之后, 迭代精灵顶点信息, 然后把这些信息添加到批数组中。

还需要其他两个函数处理批(batch)的绘制。

```

const int VertexDimensions = 3;
const int ColorDimensions = 4;
const int UVDimensions = 2;
void SetupPointers()
{
    Gl.glEnableClientState(Gl.GL_COLOR_ARRAY);
    Gl.glEnableClientState(Gl.GL_VERTEX_ARRAY);
    Gl.glEnableClientState(Gl.GL_TEXTURE_COORD_ARRAY);
    Gl.glVertexPointer(VertexDimensions, Gl.GL_DOUBLE, 0,
        _vertexPositions);
    Gl.glColorPointer(ColorDimensions, Gl.GL_FLOAT, 0, _vertexColors);
}

```

```
        Gl.glTexCoordPointer(UVDimensions, Gl.GL_FLOAT, 0, _vertexUVs);
    }
    public void Draw()
    {
        if (_batchSize == 0)
        {
            return;
        }
        SetupPointers();
        Gl.glDrawArrays(Gl.GL_TRIANGLES, 0, _batchSize);
        _batchSize = 0;
    }
}
```

Draw 函数在两者之中更加重要。如果批为空，**Draw** 函数不会执行操作。否则，它会调用 **SetupPointers** 函数。**SetupPointers** 首先使用 **glEnableClientState** 描述顶点格式。在这里，描述的顶点具有颜色、位置和(U,V)信息。完成这些操作后，使用 **glPointers** 调用告诉 OpenGL 从哪里寻找这些信息。

所有的 **glPointer** 调用的格式都相同。第一个参数是元素的数量。**glVertexPointer** 控制着元素的位置，它有 3 个元素，分别对应于 X、Y 和 Z 轴。纹理使用两个元素定义，它们分别对应于 U 和 V。**Color** 使用 4 个元素定义：红色、绿色、蓝色和 alpha。OpenGL 将使用这些指针来获取它将渲染的图形信息。

指针需要指向每个数组的首元素的内存地址，从该位置开始的内容将被顺序读取。然后，它使用数组的大小信息来确定何时停止读取内存。正因如此，**Vector**、**Color** 和 **Point** 结构各自的定义中都有 **[StructLayout(LayoutKind.Sequential)]** 特性。这些结构的成员的顺序非常重要，交换它们的位置后，渲染的结果将与预期不同。

为了告诉 OpenGL 从这些指针开始读取，然后渲染数据，执行将返回到 **Draw** 方法。**Draw** 方法中调用了 **glDrawArrays** 方法，它接受一个类型、步幅和要绘制的顶点数作为参数。类型是 **GL_TRIANGLES**，因为每个精灵都是由两个三角形构成的，而且三角形也是解释顶点信息的方式。为这个方法传入的步幅值为 0。步幅指定了读取每个顶点后跳过多少内存空间。有时候，出于效率考虑，需要把不同的顶点信息打包在一块连续的内存空间中。步幅确保了可以跳过无关紧要的位。在批(batch)中，所有的数据都是相关的，所以不需要使用步幅。最后一个参数是要渲染的顶点数，这个信息记录在 **_batchSize** 成员中。

Draw 方法中的最后一个命令是重置 **_batchSize**。不需要清空数组中的数据，因为在使用新的精灵数据重写它们之前，不会绘制这些数据。

7.5.1 修改渲染器

对于批(batch)绘制，使用 **glBegin** 和 **glEnd** 是不够的，但是它也不是过于复杂。最后一个任务是在 **Renderer** 中使用新的批(batch)方法替换原来的 **glBegin** 和 **glEnd** 方法。

```
class Renderer
{
```



```
Batch _batch = new Batch();  
public void DrawSprite(Sprite sprite)  
{  
    _batch.AddSprite(sprite);  
}  
public void Render()  
{  
    _batch.Draw();  
}  
}
```

转换渲染器类很简单：添加一个 **Batch** 对象作为其成员，进行绘制时将把精灵添加到该对象中。代码中还添加了一个 **Render** 方法，每一帧都需要调用该方法。如果在帧结束时批中还有没有绘制的精灵，**Render** 类将确保绘制它们。

Renderer 当前并不能很好地处理不同的纹理。绘制一个大批(batch)中的所有精灵时，该批(batch)中的所有精灵都必须有相同的纹理。处理这种情况的一个简单的方法是在每次添加精灵时检查纹理，如果纹理不同，则绘制批(batch)。

7.5.2 对批(batch)绘制方法执行性能分析

文本会像以前一样渲染，但是现在渲染的速度变快了。重复前面渲染 10 000 个精灵的示例时，我的计算机上的帧率从大约 30fps 变成了超过 80fps，增长非常显著。这意味着程序在老式的计算机上的运行速度会比较流畅。

7.6 小结

现在已经向代码库中添加了文本绘制功能，并改进了精灵渲染代码。几乎是时候创建一个游戏了，但是在那之前，还需要认真地了解一下游戏中用到的数学。

第 8 章

游 戏 数 学

不了解高等数学知识也可以编写游戏，但是游戏对图形的要求越高，程序员就需要知道越多的数学知识。数学分为多个领域，在编写游戏时，某些领域比其他其他领域应用得更加频繁。几何对于描述 3D 和 2D 世界很重要；矩阵和向量对于描述世界和世界中的实体之间的关系非常有帮助；三角函数可以创建特殊效果，并使对象的运动更加自然；补间函数在表达不变时间内的移动时很方便。知道的数学知识越多，在解决游戏编程中遇到问题时可以使用工具就越多。

8.1 三角函数

游戏编程中经常使用三角函数 `sine` 和 `cosine`。对某个数值应用 `sine` 和 `cosine` 将返回-1~1之间的某个值。返回的这些值形成了一个波形，即一个均匀震荡的曲线。当平滑地上下移动某个对象时，这条曲线非常有用，例如它可以创建自然的脉冲颜色、平滑的震荡缩放和其他游戏中可以使用的效果。

8.1.1 绘制图形

了解 `cosine` 和 `sine` 的用途的最佳方式是创建一个沙盒程序(sandbox program)，在其中可自由尝试各个值。在一个图形上绘制 `sine` 和 `cosine` 波形的程序是一个不错的起点。下面的游戏状态将绘制两条轴，并绘制一个图形。并不需要创建一个新项目，可以把这个状态添加到已有的代码中，然后将其设置为默认状态。

```
class WaveformGraphState : IGameObject
{
```

```
double _xPosition = -100;
double _yPosition = -100;
double _xLength = 200;
double _yLength = 200;
double _sampleSize = 100;
double _frequency = 2;
public delegate double WaveFunction(double value);

public WaveformGraphState()
{
    Gl.glLineWidth(3);
    Gl.glDisable(Gl.GL_TEXTURE_2D);
}

public void DrawAxis()
{
    Gl.glColor3f(1, 1, 1);

    Gl.glBegin(Gl.GL_LINES);
    {
        // X axis
        Gl.glVertex2d(_xPosition, _yPosition);
        Gl.glVertex2d(_xPosition + _xLength, _yPosition);
        // Y axis
        Gl.glVertex2d(_xPosition, _yPosition);
        Gl.glVertex2d(_xPosition, _yPosition + _yLength);
    }
    Gl.glEnd();
}

public void DrawGraph(WaveFunction waveFunction, Color color)
{
    double xIncrement = _xLength / _sampleSize;
    double previousX = _xPosition;
    double previousY = _yPosition + (0.5 * _yLength);
    Gl.glColor3f(color.Red, color.Green, color.Blue);
    Gl.glBegin(Gl.GL_LINES);
    {
        for (int i = 0; i < _sampleSize; i++)
        {
            // Work out new X and Y positions
```

```

double newX = previousX + xIncrement; // Increment one unit on the x
// From 0-1 how far through plotting the graph are we?
double percentDone = (i / _sampleSize);
double percentRadians = percentDone * (Math.PI * _frequency);

// Scale the wave value by the half the length
double newY = _yPosition + waveFunction(percentRadians) *
(_yLength / 2);

// Ignore the first value because the previous X and Y
// haven't been worked out yet.
if (i > 1)
{
    Gl.glVertex2d(previousX, previousY);
    Gl.glVertex2d(newX, newY);
}

// Store the previous position
previousX = newX;
previousY = newY;
}
}
Gl.glEnd();
} // Empty Update and Render methods omitted
}

```

成员变量 `_xPosition`、`_yPosition`、`_xLength` 和 `_yLength` 用于定位和描述图的大小。`_sampleSize` 变量决定了图的平滑度。采样大小是指用于绘制图中的线的顶点数。每个顶点的 y 位置由特定的一个函数决定，例如 `sine` 或 `cosine`。`_frequency` 变量用于计算波形震荡的频率，频率越高，震荡次数越多。

默认值描述了一个位于 $(x:-100, y:-100)$ 、每个轴长 200 像素的图。得到的图足够大，方便查看，但是 `DrawGraph` 函数会认为图的 x 轴和 y 轴的范围为 0~1。

在成员变量后定义了一个委托。

```
public delegate double WaveFunction(double value);
```

图经常被定义为 $x=x', y=f(x')$ ，其中 x' 是 x 的下一个值，普通的 x 是前一个值。 x 值通常递增固定的数值，而 y 值从 x 值计算得出。`WaveFunction` 委托描述了公式中的 f ，该函数接受一个 `double` 值作为参数，并返回一个 `double` 值。这与 `cosine` 和 `sine` 函数的签名相同。通过使用 `WaveFunction` 类型，`DrawGraph` 函数可以接受 `cosine`、`sine` 或其他任意波形函数作为参数，并不需要编写其他代码。

状态的构造函数将线宽设置为 3 像素，这样就更容易看到图中的线。它还关闭了纹理的状态。如果打开了纹理状态，线看上去可能有点暗，因为它们被错误地赋予了无效的

纹理。

`DrawGraph` 函数是游戏状态中最重要的函数，它负责绘制图。`DrawGraph` 使用采样率决定如何分布顶点，使任意一条线都会占据整个图的长度。定义角度有两种常见的方式：度数和弧度。人们都很熟悉度数，知道整个圆周是 360° ，半个圆周是 180° 。弧度是使用 Pi 衡量度数的一种方法，整个圆周是 $2 \times \text{Pi}$ ，半个圆周是 Pi 。C# 中的 `Sin` 和 `Cos` 函数接受以弧度表示的角度作为参数，因此，计算 y 轴的值时， x 轴的 $0 \sim 1$ 之间的值会乘以 Pi 。

`DrawGraph` 函数的内循环从原来的位置计算出新位置，然后绘制一条从原来的位置连接到新位置的线。为了实际演示代码，需要编写一个 `Render` 函数来为正弦函数和余弦函数调用 `DrawGraph`。图的输出如图 8-1 所示。

```
public void Render()  
{  
    DrawAxis();  
    DrawGraph(Math.Sin, new Color(1,0,0,1));  
    DrawGraph(Math.Cos, new Color(0, 0.5f, 0.5f, 1));  
}
```

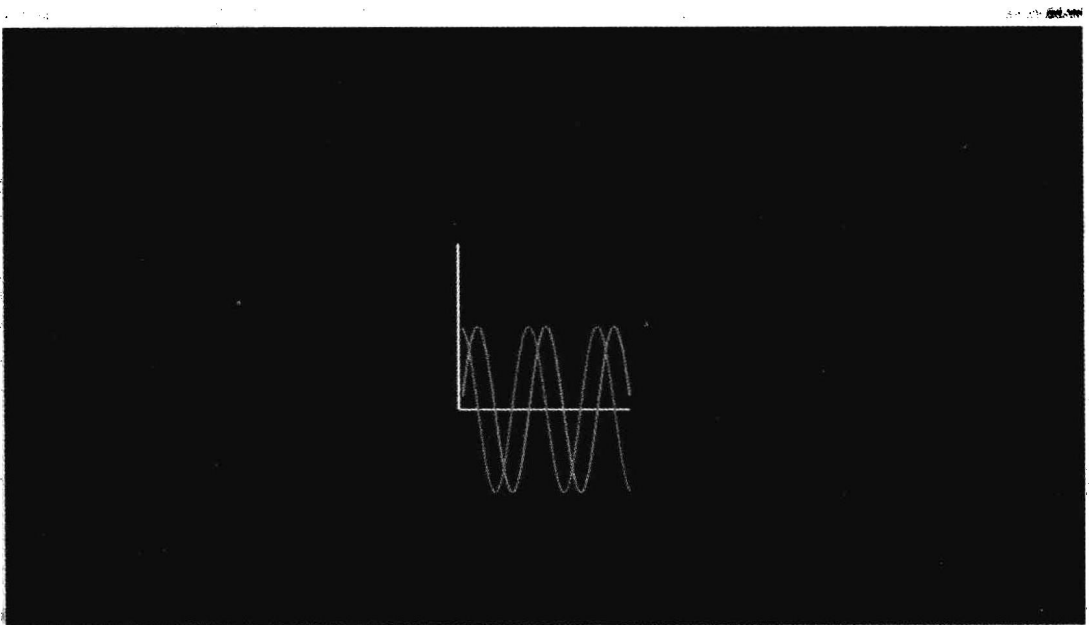


图 8-1 正弦函数和余弦函数的图

`DrawGraph` 函数有两个参数，一个是用于绘制图的函数，另一个是决定图的颜色值。正弦和余弦是波形，将这两个波形加起来，很容易产生有趣的新波形。通过使用匿名方法，可以创建一个新的波形函数。下面的代码段创建了一个图，将余弦和正弦加起来，并把结果缩小了一半。

```
DrawGraph(delegate(double value)
```

```
{  
    return (Math.Sin(value) + Math.Cos(value)) * 0.5;  
}, new Color(0.5f, 0.5f, 1, 1));
```

尝试运行下面的代码段，然后观察得到的结果。

```
DrawGraph(delegate(double value)  
{  
    return (Math.Sin(value) + Math.Sin(value + value)) * 0.5;  
}, new Color(0.5f, 0.5f, 1, 1));
```

这些图看起来很有趣，但如果在游戏中没有应用，它们看上去就有点学术化。接下来，我们就使用这些函数来使精灵动起来。

8.1.2 使用三角函数实现特殊效果

创建一个新的游戏状态 `SpecialEffectsState`。这个状态将演示如何使用 `Text` 类以及刚才讨论的正弦和余弦函数创建很酷的特效。

```
class SpecialEffectState : IGameObject  
{  
    Font _font;  
    Text _text;  
    Renderer _renderer = new Renderer();  
    double _totalTime = 0;  
  
    public SpecialEffectState(TextureManager manager)  
    {  
        _font = new Font(manager.Get("font"), FontParser.Parse("font.fnt"));  
        _text = new Text("Hello", _font);  
    }  
  
    public void Update(double elapsedTime)  
    {  
    }  
  
    public void Render()  
    {  
        Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);  
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);  
        _renderer.DrawText(_text);  
        _renderer.Render();  
    }  
}
```

基本的状态只是渲染出文本“Hello”。使用这个正弦值很容易使文本的透明度从 0 逐渐变化为 1，然后又从 1 变化为 0。这里使用的是文本，但是使用精灵或模型也一样很简单。

```
public void Update(double elapsedTime)
{
    double frequency = 7;
    float _wavyNumber = (float)Math.Sin(_totalTime*frequency);
    _wavyNumber = 0.5f + _wavyNumber * 0.5f; // scale to 0-1
    _text.SetColor(new Color(1, 0, 0, _wavyNumber));

    _totalTime += elapsedTime;
}
```

总时间记录状态运行了多久。该数值不断增加，变得越来越大，最终会返回到 0。它为正弦函数填充数值，得到与前面类似的波形。正弦波将被缩放为在 0~1 之间震荡，而不是在-1~1 之间震荡。频率也将被增长，使脉冲发生的次数更多。运行这段代码，查看得到的效果。

有了跳入又跳出视野的效果以后，下一步是使文本不断变幻色彩。每个颜色通道都被赋予一个不同的正弦或余弦波，每个通道的强度都会随时间改变。

```
public void Update(double elapsedTime)
{
    double frequency = 7;
    float _wavyNumberR = (float)Math.Sin(_totalTime*frequency);
    float _wavyNumberG = (float)Math.Cos(_totalTime*frequency);
    float _wavyNumberB = (float)Math.Sin(_totalTime+0.25*frequency);
    _wavyNumberR = 0.5f + _wavyNumberR * 0.5f; // scale to 0-1
    _wavyNumberG = 0.5f + _wavyNumberG * 0.5f; // scale to 0-1
    _wavyNumberB = 0.5f + _wavyNumberB * 0.5f; // scale to 0-1

    _text.SetColor(new Color(_wavyNumberR, _wavyNumberG,
    _wavyNumberB, 1));

    _totalTime += elapsedTime;
}
```

很容易随意使用这段代码来获得各种不同的效果。三角函数不只可以修改颜色通道，还可以修改文本的位置，如下例所示。要改变文本的位置，必须向 `Text` 类添加一个 `SetPosition` 方法。为移动文本，组成每个字符的所有顶点的位置都需要改变。最简单的方法是在新位置重新创建所有的字符。

```
public void SetPosition(double x, double y)
{
    CreateText(x, y);
}
```



```
}
```

定义了 `Text` 类的 `SetPosition` 方法后，可以在更新循环中使用它来创建新的基于文本的特殊效果。

```
public void Update(double elapsedTime)
{
    double frequency = 7;
    double _wavyNumberX = Math.Sin(_totalTime*frequency)*15;
    double _wavyNumberY = Math.Cos(_totalTime*frequency)*15;

    _text.SetPosition(_wavyNumberX, _wavyNumberY);

    _totalTime += elapsedTime;
}
```

这段代码会沿着一个不太规整的圆移动文本。这一次不需要把数值缩放到 0~1 之间，相反，它们被加大了，这样文本的移动会更加明显。根据需要，可以使用不同的函数来修改文本的位置。

最后这个示例使文本的每个字符运动起来，就好像有一个波形在传递词一样。为了设置文本的每个字符的运动，必须在 `Sprite` 类中添加一个新的 `GetPosition` 方法。精灵的位置从精灵的中心开始计算。

```
public Vector GetPosition()
{
    return GetCenter();
}
```

把上面的代码添加到 `Sprite` 类以后，可以在更新循环中使用新的 `GetPosition` 方法。

```
public void Update(double elapsedTime)
{
    double frequency = 7;

    int xAdvance = 0;
    foreach (CharacterSprite cs in _text.CharacterSprites)
    {
        Vector position = cs.Sprite.GetPosition();
        position.Y = 0 + Math.Sin((_totalTime + xAdvance) * frequency)*25;
        cs.Sprite.SetPosition(position);
        xAdvance++;
    }

    _totalTime += elapsedTime;
}
```

8.2 向量

向量是一个常用的游戏编程工具。了解向量数学最简单的方式是在游戏中使用向量，很快就可以熟悉向量的各种用法。前面的章节中已经创建了一个简单的向量类，但是到现在为止还没有定义它的任何属性。

8.2.1 向量的定义

在游戏编程中，向量用于操作和描述 3D 游戏世界。向量的数学定义是既有大小又有方向的量。大小就是向量的长度，如图 8-2 所示。

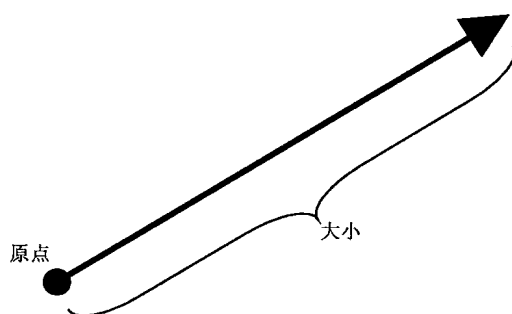


图 8-2 分解向量

游戏编程中最常用的向量是 2D、3D 和 4D 向量。在使用投影矩阵将顶点从 3D 转换到 2D 空间时，会用到 4D 向量。纸面上看起来，向量和位置很类似。位置[0,3,0]和向量[0,3,0]有相同的内部值，但是它们代表不同的概念。位置使用坐标系定义世界中的一个绝对位置。向量描述一个方向(在这里为向上)和大小或者说长度(在这里为 3)。图 8-3 比较了两者的不同。例如，“离地面 3 英里”不是一个位置，而是使用位置和长度描述了一个位置，换句话说，它是一个向量。

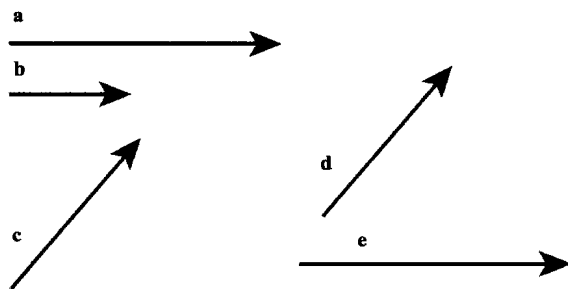


图 8-3 比较向量和位置

在图 8-3 中，向量 a 和 3 是相同的向量，因为它们具有相同的方向和大小。它们的原点不同，所以从图中看来似乎两者是不同的。向量 c 和 d 也是相同的。向量 b 与向量 a 和

e 的方向相同,但是大小要小得多,所以它是一个不同的向量。

在游戏编程中,向量用于回答如下问题:

- 敌人的飞船位于[0,0,90],玩家的飞船位于[0,80,-50]。敌人应该向哪个位置发射导弹?在每一帧中,如何更新导弹的位置,使其朝向玩家不断移动?
- 玩家与导弹的位置小于 1m 吗?
- 玩家在墙的哪一边?
- 玩家在看 NPC 吗?
- 给定一个表示枪的后部的四方形,子弹将向哪个方向飞出?
- 光应该如何从这个表面反射出去?
- 把汽车向前移动 3m。
- 玩家击中了一个敌人,敌人应该向哪个方向运动?
- 如何将子弹的动力增加 100?
- 哪个玩家最接近外星人的飞船?

为了回答这些问题,需要扩展向量类,在其中添加基本的向量方法。本章中的示例将以前面章节中的代码为基础。其中的大多数示例都要求创建一个新的游戏状态,并使其成为活动的状态以测试代码。本章所有的示例代码都可以从本书配套光盘中的 Code\Chapter 8 目录中找到。下面的小节将解释各种向量操作,并列出完成向量类所需添加的代码。游戏引擎中有时候会有 Vector2d、Vector3d 和 Vector4d,但是在这里只创建一个向量结构会更加简单一些,这个向量仍然可以用于任意的 2D 操作。本书中的内容不会涉及 4D 向量。

```
[StructLayout(LayoutKind.Sequential)]
public struct Vector
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
    public Vector(double x, double y, double z) : this()
    {
        X = x;
        Y = y;
        Z = z;
    }
}
```

8.2.2 长度操作

长度操作以一个向量作为参数,返回该向量的大小。对于简单的向量,如[0,1,0],很容易看出长度为 1。但是对于复杂一些的向量,如[1.6,-0.99,8],很难一眼看出其长度。如下所示的公式可以计算向量的长度。

$$\|v\| = \sqrt{x^2 + y^2 + z^2}$$

v 旁边的两条竖线是数学上表示向量长度的一种方法。该公式对于任意维度的向量都

是相同的：计算成员的平方值，将这些值相加，然后取其平方根。

用代码表示这个公式很简单。这里使用了两个函数：一个用于计算成员的平方值，然后对这些平方值求和；另一个函数执行求平方根的运算。

```
public double Length()
{
    return Math.Sqrt(LengthSquared());
}
public double LengthSquared()
{
    return (X * X + Y * Y + Z * Z);
}
```

如果想要比较两个向量的长度，可以采用 **LengthSquared** 操作，而不是采用 **Length** 操作，这样可以省去求平方根的操作，从而使代码更高效一些。

8.2.3 向量的相等性

如果所有的成员值(*X*、*Y*和*Z*)都相等，则认为向量是相等的。向量没有位置，它们只是从某个原点开始的一个方向。图 8-3 显示了一组向量，可以看到，即使一些向量放到了不同的位置，它们仍然是相等的，因为它们的成员是相等的。不管是从你的房子向北 3 英里，还是从吉萨金字塔向北 3 英里，它都一样是向北 3 英里。

为向量创建一个 **Equals** 函数是很简单的。

```
public bool Equals(Vector v)
{
    return (X == v.X) && (Y == v.Y) && (Z == v.Z);
}
```

在代码中，如果重载了 **=** 操作符的话，就更方便了。现在，还不能编写下面的代码。

```
// Cannot write this
if (vector1 == vector2)
{
    System.Console.WriteLine("They're the same")
}

// Instead must write
if (vector1.Equals(vector2))
{
    System.Console.WriteLine("They're the same")
}
```

要使用 **=** 操作符，需要重载该操作符，还需要重写其他一些函数或操作符，如 **GetHashCode**、**!=** 和 **Equals(Object obj)**。

```
public override int GetHashCode()
{
    return (int)X ^ (int)Y ^ (int)Z;
}

public static bool operator ==(Vector v1, Vector v2)
{
    // If they're the same object or both null, return true.
    if (System.Object.ReferenceEquals(v1, v2))
    {
        return true;
    }

    // If one is null, but not both, return false.
    if (v1 == null || v2 == null)
    {
        return false;
    }

    return v1.Equals(v2);
}

public override bool Equals(object obj)
{
    if (obj is Vector)
    {
        return Equals((Vector)obj);
    }
    return base.Equals(obj);
}

public static bool operator !=(Vector v1, Vector v2)
{
    return !v1.Equals(v2);
}
```

重载`==`操作符时使用了大量的代码。唯一有点新奇的函数是`GetHashCode`。哈希值是一个数值,尝试(但是并不保证)唯一地标识一个对象,C#的`Dictionary`结构中用到了哈希值。当重写相等性时,也需要重写哈希值,否则编译器很难知道哪一个才是正确的哈希值。

8.2.4 向量加法、减法和乘法

向量加法操作很简单,就是把第一个向量的各个成员与第二个向量的各个成员分别相加。下面的向量执行向量加法。

```
public Vector Add(Vector r)
{
    return new Vector(X + r.X, Y + r.Y, Z + r.Z);
}

public static Vector operator+(Vector v1, Vector v2)
{
    return v1.Add(v2);
}
```

重载二元的加法操作符+时，会自动重载+=。同理也适用于*=和/=。

图 8-4 显示了将两个向量加到一起后的结果。当试图在 3D 空间中得到特定的偏移值时，经常把两个向量加到一起。例如，假设想要在玩家头顶放置一个 3D 光环模型。玩家的原点在一只脚的中间。使用一个向量表示从玩家的脚到玩家头部的中央 $[0, 1.75, 0]$ 之间的偏移。如果添加一个向量 $[0, 0.2, 0]$ ，这可以得到一个非常适合放置光环的位置。图 8-5 显示了这个操作。

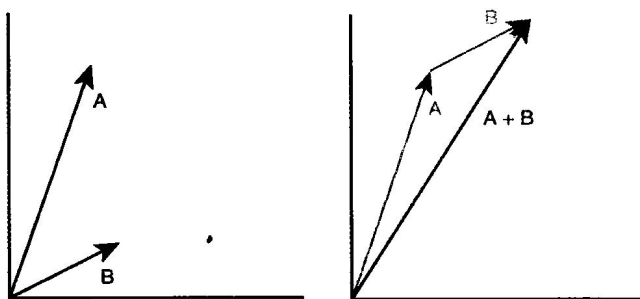


图 8-4 向量加法

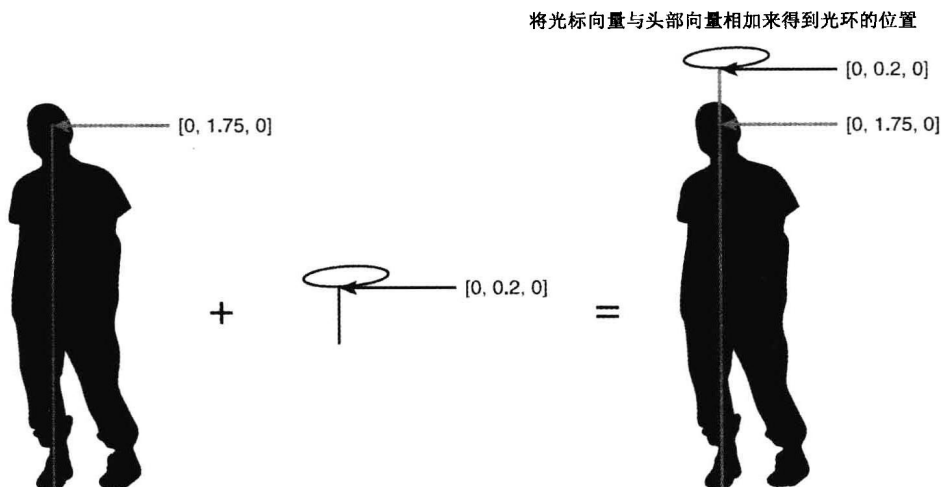


图 8-5 向玩家添加光环

向量减法总是用于获取空间中两点之间的向量。减法计算与加法计算很类似，但成员是相减而不是相加。

```
public Vector Subtract(Vector r)
{
    return new Vector(X - r.X, Y - r.Y, Z - r.Z);
}
public static Vector operator-(Vector v1, Vector v2)
{
    return v1.Subtract(v2);
}
```

将两个向量相减后的结果如图 8-6 所示。在太空战斗中，一艘飞船可能想要击落另外一艘飞船。飞船 A 可以从飞船 B 的位置减去自己的位置(两个位置都使用向量表示)，这将得到从 A 到 B 的向量(见图 8-7)。这个向量的方向可以用于瞄准导弹，或者将一艘飞船朝向另外一艘飞船的方向。

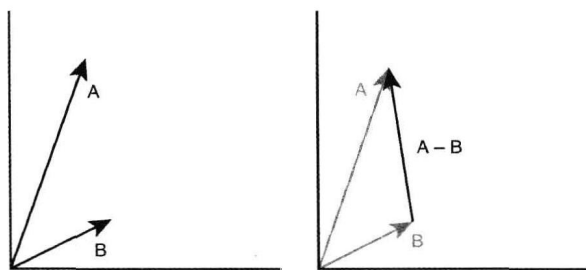


图 8-6 向量减法

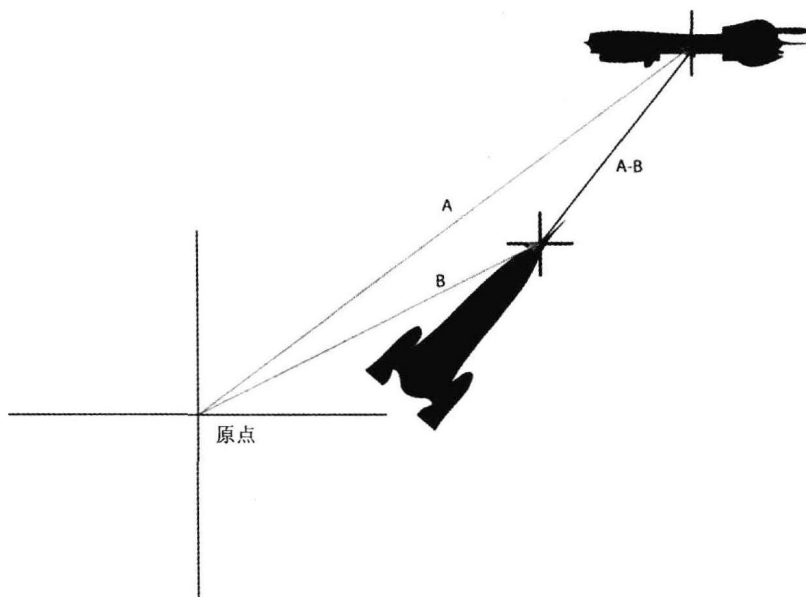


图 8-7 获得两艘飞船之间的向量

向量乘法是指将一个向量与一个标量相乘。标量就是 `int` 或者 `double` 这样的普通数值。如果一个向量的所有元素都乘以另外一个元素，这种计算就称为点积。下面列出了点积的计算方法。

```
public Vector Multiply(double v)
{
    return new Vector(X * v, Y * v, Z * v);
}

public static Vector operator * (Vector v, double s)
{
    return v.Multiply(s);
}
```

图 8-8 显示了将向量与标量相乘后得到的结果。将向量与标量相乘会缩放向量，所以乘以 2 会使向量的长度加倍。将向量乘以 -1 会使向量指向与当前方向相反的方向。如果在 3D 游戏中玩家角色被击中，可以将代表子弹轨迹的向量乘以 -1，得到相反的方向。现在这个向量从玩家的身体出发，逆着子弹的轨迹指向外部，所以非常适合用来实现鲜血喷出的效果(见图 8-9)。

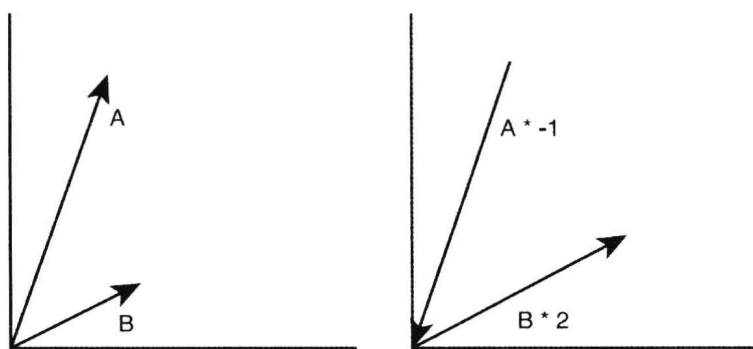


图 8-8 向量与标量的乘法



图 8-9 使用向量实现鲜血喷出的效果

8.2.5 法向量

法向量是长度为 1 的向量，也称为单位向量。单位向量是在不考虑大小时表示方向的绝佳方式。归一化(normalize)操作保持向量的方向不变，但是使其大小变为 1。如果把一个单位向量与一个标量相乘，得到的向量的长度会与标量值相同。如果向量的长度未知，而想使其长度为 6，就可以归一化该向量，然后乘以 6。

```
public Vector Normalize(Vector v)
{
    double r = v.Length();
    if (r != 0.0) // guard against divide by zero
    {
        return new Vector(v.X / r, v.Y / r, v.Z / r); // normalize and return
    }
    else
    {
        return new Vector(0, 0, 0);
    }
}
```

从技术上来说，这段代码并不正确，这是因为 **0** 向量是无法归一化的，而这段代码在归一化 **0** 向量时没有采取任何操作。通过计算向量的长度，然后把每个元素除以该长度，可以归一化向量。归一化向量的效果如图 8-10 所示。

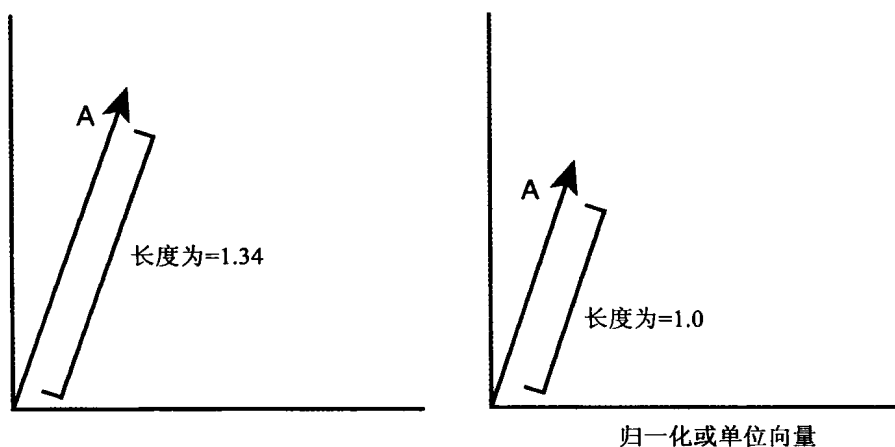


图 8-10 归一化向量

在游戏编程中，方向很重要，它们通过法向量来定义。你可能听说过法线映射(normal mapping)这个词。那是一种纹理，其中每个像素代表一个法向量，纹理在 3D 模型中被拉伸，光照计算会考虑到渲染后的模型的每个像素上的法向量。法线映射使最终模型的表面上可

以展现出比没有使用法线映射时多得多的细节。

假设有一个二维向量，它包含 X 元素和 Y 元素。可以分别使用 $[0,1]$ 、 $[0,-1]$ 、 $[-1,0]$ 和 $[1,0]$ 创建上、下、左、右 4 个向量。这创建了一个类似于十字的形状。如果在上、下、左、右 4 个向量之间再创建 4 个法向量，就可以开始模拟圆的过程了。这是一个单位圆，即其半径为 1。图 8-11 显示了一个单位圆。如果使用 3 元素向量 $[X,Y,Z]$ ，那么就会创建一个单位球。如果所有向量的长度都为 2，那么就会创建半径为 2 的球，以此类推。

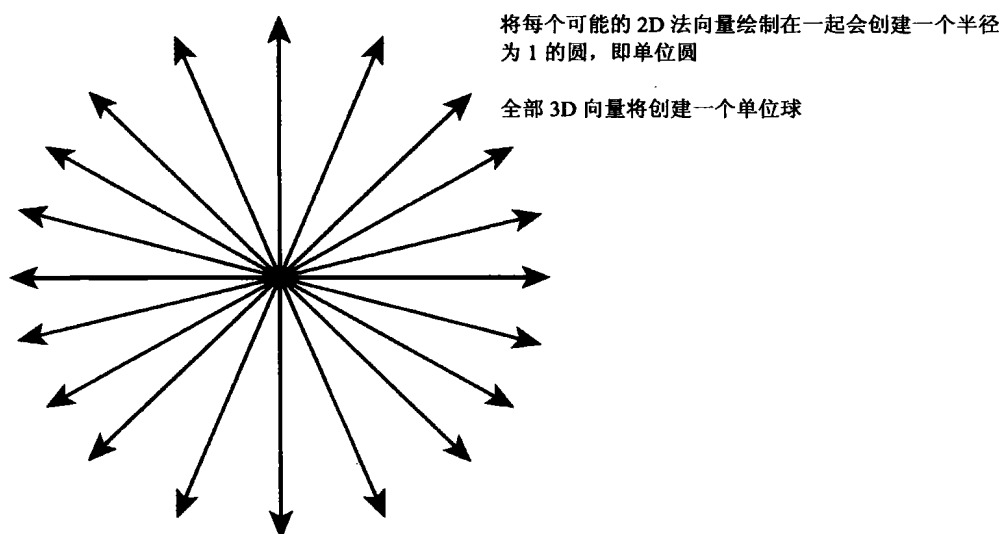


图 8-11 单元圆

测试圆或球与某个点是否相交十分简单，其实就是测试一个点是否在圆或者球内。以圆为例。圆通过圆心和半径定义。如果在 $[5,6]$ 处有一个单位圆，要判断点 $[5.5,6.5]$ 是否在该圆内，第一步是计算出该点与圆心的位置。这是通过对圆心的向量与该点的向量做减法得到的： $[5, 6] - [5.5, 6.5] = [0.5, 0.5]$ ，减法的结果是从该点到圆心的一个向量。然后求出该向量的长度，也就是该点与圆的距离。计算得到的长度为 0.707。如果这个长度小于圆的半径，那么点在圆内；如果长度大于圆的半径，则点在圆外。如果两者相等，则点在圆周上，如图 8-12 所示。

相同的方法也适用于球和点的相交性测试。这可以快速测试玩家是否在特定的位置，或者鼠标是否单击了特定的区域。如果不能想象出来，可以尝试在纸上绘制几个例子，直到自己熟悉了这种方法。此时，试着思考一下如何进行圆与圆或者球与球的相交性测试。

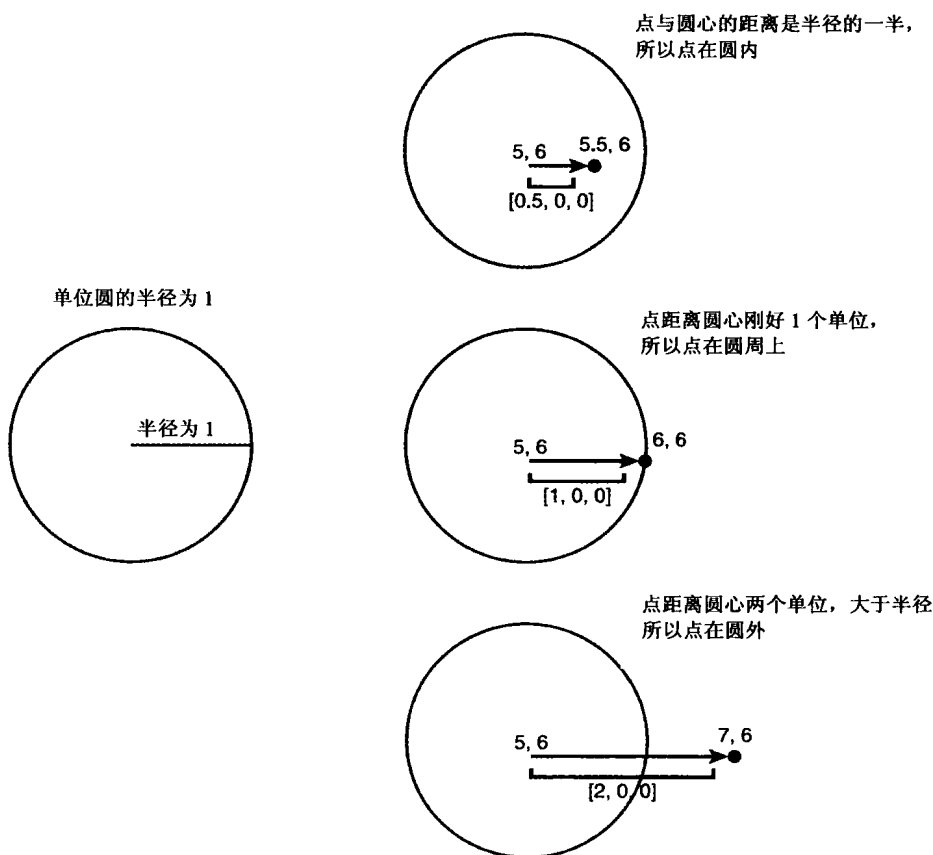


图 8-12 圆和点的相交性测试

8.2.6 点积运算

点积一种运算，以两个向量为操作数，返回一个数值。返回的数值与两个向量之间的角度有关。

$$\mathbf{A} \cdot \mathbf{B} = |\mathbf{A}| |\mathbf{B}| \cos(\theta)$$

向量 \mathbf{A} 和向量 \mathbf{B} 的点积运算返回 \mathbf{A} 和 \mathbf{B} 的长度与两者夹角的余弦的乘积。图 8-13 以图形化方式显示了点积运算。如果向量 \mathbf{B} 和 \mathbf{A} 经过了归一化，那么它们的长度值为 1，公式将简化为 $\mathbf{A} \cdot \mathbf{B} = \cos(\theta)$ 。要想得到角度值，可以使用反余弦运算。在 C# 中，反余弦函数是 `Math.acos`，它返回以弧度计算的角度值。

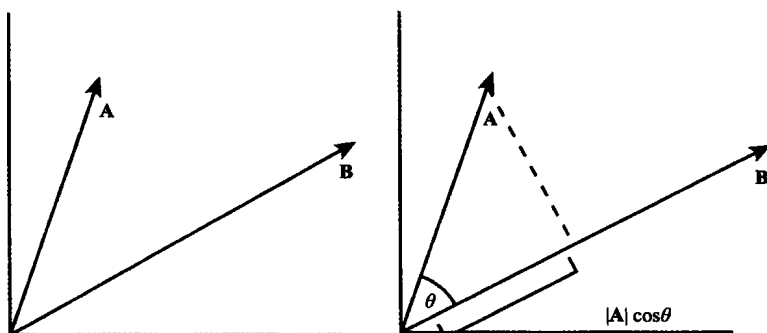


图 8-13 点积运算

点积运算非常适合确定游戏中的对象是否彼此相向。它也适合用来测试某个对象是否在一个对象的某一边，3D 游戏中经常使用这种方法来保证角色不能穿墙。

点积运算本身很简单：第一个向量的所有元素与第二个向量的对应元素分别相乘，然后这些值加到一起，得到一个标量。点积在 3D 图形学中是一种很常用的运算，所以运算符*常被重载以代表这种运算。

```
public double DotProduct(Vector v)
{
    return (v.X * X) + (Y * v.Y) + (Z * v.Z);
}

public static double operator *(Vector v1, Vector v2)
{
    return v1.DotProduct(v2);
}
```

点积对于判断点在平面之前还是之后非常有用。平面是一个二维表面，就像一张纸。纸可以摆放在任何位置，摆放成任意角度，就像是一个集合平面。区别在于，纸张是有边缘的，平面却没有，它们沿着两个维度无限延展，就像是没有边界的纸张。平面通过使用点和法向量定义(见图 8-14)。点确定平面在空间中的位置，法向量确定了平面的指向。

在游戏中，可能把 3D 平面放在某个位置来标志变化：如果玩家通过了这个平面，则他通关，或者出现一个 boss，或者玩家掉入海中。测试玩家位于平面哪一侧的方法如下。

- 创建一个从平面位置指向玩家位置的向量。
- 对平面的法向量和新创建的向量求点积。
- 如果结果为 0，玩家刚好位于平面上。
- 如果结果大于 0，玩家位于平面的法向量一侧。
- 如果结果小于 0，玩家位于平面的另一侧。

图 8-15 以图形化的方式显示了这个测试。

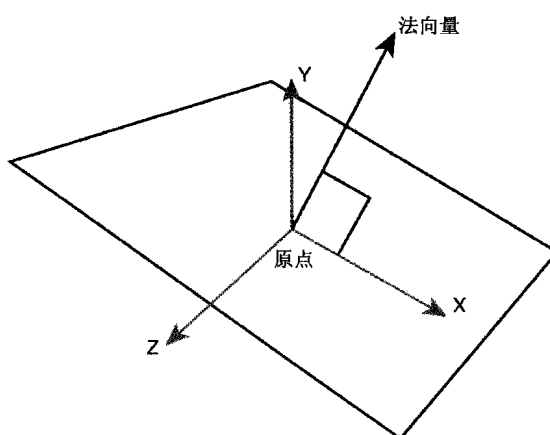


图 8-14 一个平面

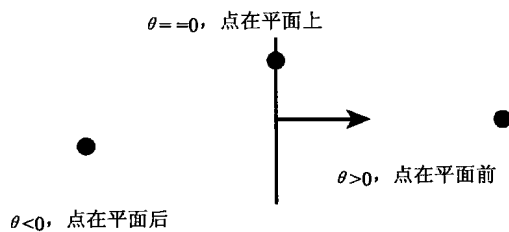
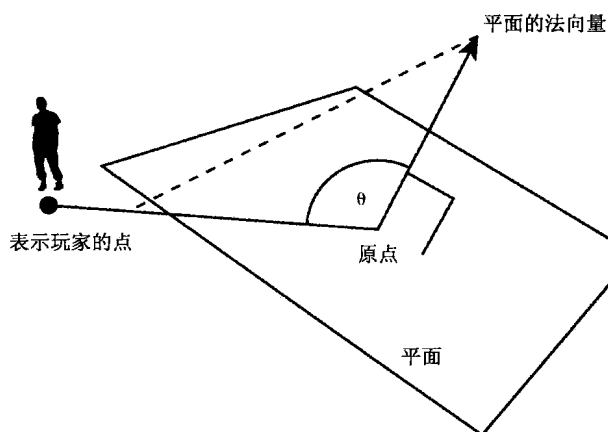


图 8-15 玩家和平面位置

点积测试也用于背面剔除(back face culling)。背面剔除用于查看某个多边形是否没有面对摄像机。默认情况下，多边形并不包含两个边，它们只包含由法向量指示的一边。如果多边形没有面对摄像机，则玩家看不到它们，这意味着不必告诉图形硬件它们的存在。点积可以用来筛选掉没有面对摄像机的所有多边形。

8.2.7 叉积运算

最后要讨论的一个向量运算是叉积。与前面的运算不同，这种运算只能处理包含 3 个或更多个元素的向量。也就是说，简单的[X,Y]向量是没有叉积运算的。这种运算比前面的运算更加复杂，但是结果却很直观。叉积对两个向量执行运算，返回与传入的向量垂直的一个向量。如果有一张桌子，它的一面从[0,0,0]指向[0,0,1]，另一面从[0,0,0]指向[1,0,0]，那么得到的叉积向量是从桌子的表面指向上方的向量[0,1,0]。图 8-16 显示了叉积运算。

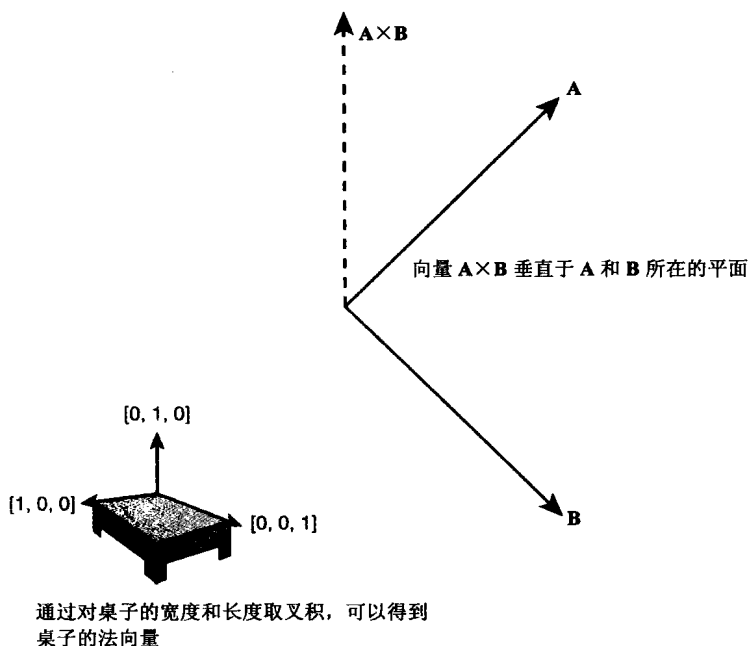


图 8-16 叉积运算

计算叉积的公式如下。

$$\mathbf{A} = \mathbf{B} \times \mathbf{C}$$

$$\mathbf{A} = \begin{bmatrix} A_x \\ A_y \\ A_z \end{bmatrix} \mathbf{B} = \begin{bmatrix} B_x \\ B_y \\ B_z \end{bmatrix} \mathbf{C} = \begin{bmatrix} C_x \\ C_y \\ C_z \end{bmatrix}$$

$$A_x = B_y C_z - B_z C_y$$

$$A_y = B_z C_x - B_x C_z$$

$$A_z = B_x C_y - B_y C_x$$

上面的公式看起来有点让人望而生畏，但是用代码表达后，就没那么复杂了。使用这个公式时，不必考虑代码的具体工作原理，而只需知道自己想要得到一个从计算叉积的两个向量指向外部的一个向量。

```

public Vector CrossProduct(Vector v)
{
    double nx = Y * v.Z - Z * v.Y;
    double ny = Z * v.X - X * v.Z;
    double nz = X * v.Y - Y * v.X;
    return new Vector(nx, ny, nz);
}

```

叉积对于计算出一个平面的法向量十分有用。例如，在 3D 世界中可能有一列向前行驶的火车。火车行驶角度可以是任意的，所以很难确定“向前”的含义。首先，需要一个与火车同向的法向量。如果在火车上你发现了一个面对着自己希望的方向的向量，那么计算这个多边形的两个面的叉积所得到的向量将沿着火车指向前方，如图 8-17 所示。这个向量可被归一化，使其只表示方向。将这个方向乘以某个标量，然后加到火车的位置上，火车将向前移动标量确定的距离。

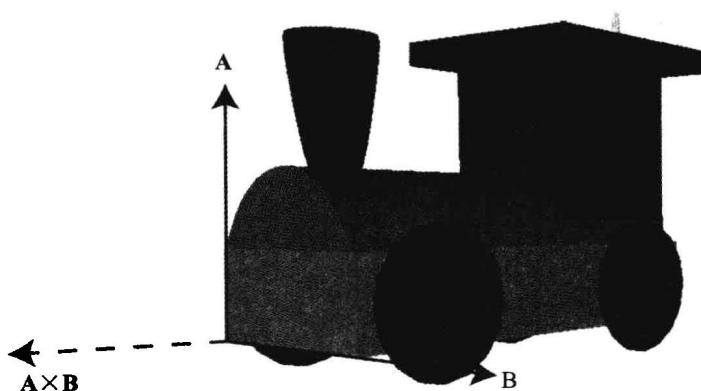


图 8-17 计算火车向前的法向量

8.2.8 关于向量结构的最后一点内容

现在已经完成了一个很不错的、功能完善的向量结构，但是稍加一些处理，还可以使这个结构更加容易使用。首先，可以重写 `ToString` 方法，使其输出一些有用的内容。在 `Visual Studio` 中使用调试器时，将自动调用 `ToString` 方法。重写这个方法后，可以立刻了解向量，而不需要仔细分析向量的定义并查看它可以取各个值。

```

public override string ToString()
{
    return string.Format("X:{0}, Y:{1}, Z:{2}", X, Y, Z);
}

```

0 向量是一个特殊的向量，它没有单位向量。**0 向量**不代表方向，也没有大小。它有些类似于向量中的 `null`。因此，把它定义为一个常量是很有帮助的。

```
[StructLayout(LayoutKind.Sequential)]
```

```
public struct Vector
{
    public static Vector Zero = new Vector(0, 0, 0);
```

这是所需的全部向量操作。通过这个简单的结构，可以构建并操作一个完整的 3D 世界。如果现在还不能完全理解向量的所有特性，也不用担心，这是一个熟能生巧的过程。

8.3 二维相交

相交用于判断两个形状何时重叠或相交。在所有的图形编程(包括游戏)中这都很重要。检测鼠标指针是否在一个按钮上就是一种相交测试。在游戏中，检测一个导弹是否击中飞船也是一种相交性测试。2D 相交十分简单，可以作为一个不错的起点。

8.3.1 圆

圆通过位置和弧度定义。相交以图形化方式显示最容易理解。接下来就创建一个游戏状态 `CircleIntersectionState`，使其成为默认加载的状态。

```
class CircleIntersectionState : IGameObject
{
    public CircleIntersectionState()
    {
        Gl.glLineWidth(3);
        Gl.glDisable(Gl.GL_TEXTURE_2D);
    }
    #region IGameObject Members

    public void Update(double elapsedTime)
    {
    }

    public void Render()
    {
    }
    #endregion
}
```

现在状态只是准备好 OpenGL，使其可以画线。接下来需要创建 `Circle` 类。

```
public class Circle
{
    Vector Position { get; set; }
    double Radius { get; set; }
```



```
public Circle()
{
    Position = Vector.Zero;
    Radius = 1;
}

public Circle(Vector position, double radius)
{
    Position = position;
    Radius = radius;
}

public void Draw()
{
    // Determines how round the circle will appear.
    int vertexAmount = 10;
    double twoPI = 2.0 * Math.PI;

    // A line loop connects all the vertices with lines
    // The last vertex is connected to the first vertex
    // to make a loop.
    Gl.glBegin(Gl.GL_LINE_LOOP);
    {
        for (int i = 0; i <= vertexAmount; i++)
        {
            double xPos = Position.X + Radius * Math.Cos(i * twoPI /
vertexAmount);
            double yPos = Position.Y + Radius * Math.Sin(i * twoPI /
vertexAmount);
            Gl.glVertex2d(xPos, yPos);
        }
    }
    Gl.glEnd();
}
```

默认情况下，在原点位置定义一个圆，其半径为 1 个单位。现在还没有编写绘制圆的代码。绘制圆时，将使用 OpenGL 的立即模式，即只绘制圆的轮廓。正弦和余弦函数用于确定在什么位置绘制组成圆周的各个顶点。

为了在 `CircleIntersectionState` 内测试绘图函数，应在类中添加一个成员 `_circle`。

```
Circle _circle = new Circle(Vector.Zero, 200);
```

现在以原点为圆心、创建一个半径为 200 的圆。为了看到圆，需要修改渲染方法。

```
public void Render()  
{  
    _circle.Draw();  
}
```

运行程序，得到的结果如图 8-18 所示。

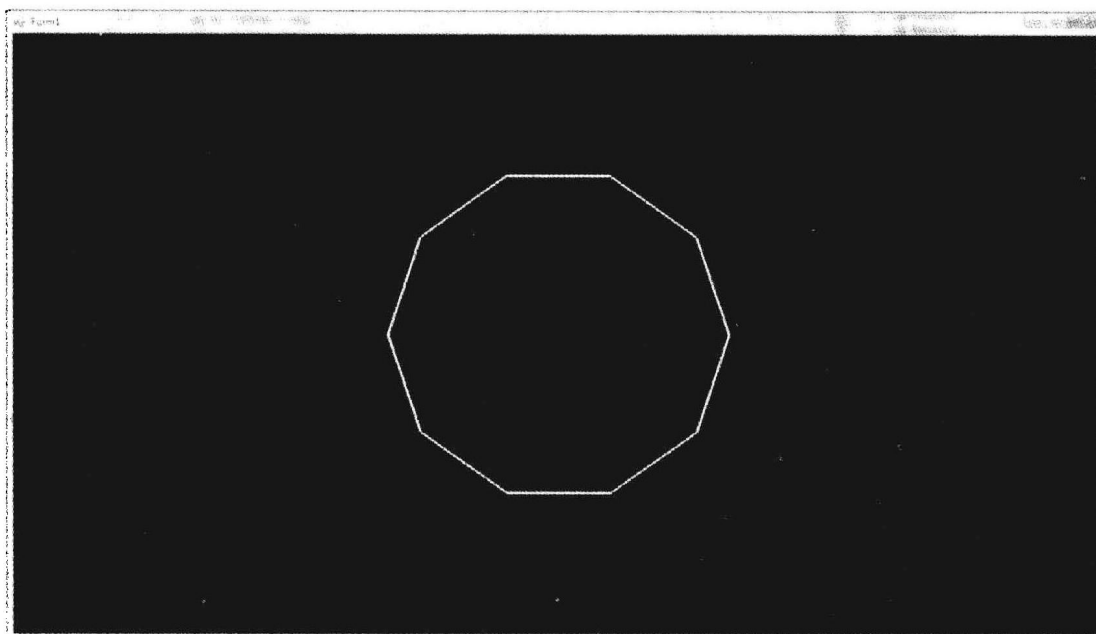


图 8-18 渲染一个圆

圆通过使用 10 个顶点绘制而成，所以看上去不是十分平滑。为了增加圆的平滑度，应增加 `Circle` 类的 `Draw` 方法中使用的顶点数。

为了演示相交性，最好能够为圆上色。使用白色的圆表示没有相交的圆，使用红色的圆表示圆与某个东西相交。实现这种功能很简单。在 `Circle` 类中，修改代码，添加一个在 `Draw` 函数中使用的颜色成员。

```
Color _color = new Color(1, 1, 1, 1);  
public Color Color  
{  
    get { return _color; }  
    set { _color = value; }  
}  
  
public void Draw()  
{  
    Gl.glColor3f(_color.Red, _color.Green, _color.Blue);
```

默认情况下，所有的圆都将被渲染成白色，但是颜色可以随时改变。

介绍向量的部分已经解释了如何进行圆与点的相交性测试，方法是：计算点与圆心的距离，如果这个距离大于圆的半径，那么点位于圆之外。为了以图形化的方式进行测试，可以把鼠标指针作为一个点。

获得鼠标指针位置的操作稍微有点棘手，因为这涉及了不同的坐标系。OpenGL 原点位于窗体的中心，但是光标的位置使用不同的坐标系确定，它的原点是窗体的左上角。也就是说，OpenGL 认为窗体的中心是位置(0,0)，但光标认为窗体的左上角才是[0,0]。图 8-19 显示了 3 个不同的坐标系。窗体的坐标原点标记为 a，控件的坐标原点标记为 b，OpenGL 的坐标原点标记为 c。为了将鼠标指针从窗体坐标系转换为 OpenGL 坐标系，需要向窗体中再添加一些代码。

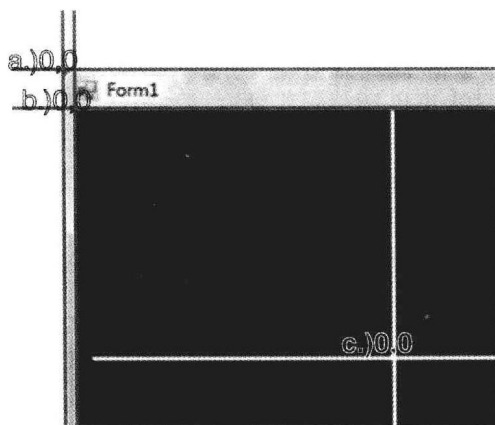


图 8-19 窗体的坐标系

首先需要简单的 `Input` 类来记录鼠标输入。

```
public class Input
{
    public Point MousePosition { get; set; }
}
```

`Input` 类将在窗体中初始化和更新。想知道鼠标位置的 `GameStates` 需要把 `Input` 对象放到自己的构造函数中。现在在窗体类中添加一个 `Input` 对象。

```
public partial class Form1 : Form
{
    Input _input = new Input();
```

在窗体中需要创建一个新函数来更新输入类，该函数将在每一帧中调用。

```
private void UpdateInput()
{
    System.Drawing.Point mousePos = Cursor.Position;
    mousePos = _openGLControl.PointToClient(mousePos);
```

```
// Now use our point definition,
Point adjustedMousePoint = new Point();
adjustedMousePoint.X = (float)mousePos.X - ((float)ClientSize.Width
/ 2);
adjustedMousePoint.Y = ((float)ClientSize.Height / 2)-(float)mouse-
Pos.Y;
_input.MousePosition = adjustedMousePoint;
}
```

```
private void GameLoop(double elapsedTime)
{
    UpdateInput();
}
```

UpdateInput 使用 PointToClient 函数将光标的位置从窗体的坐标系转换到控件的坐标系。然后将基于 OpenGL 控件的中心，把光标位置转换到 OpenGL 坐标系中。这是将控件的 X 坐标和 Y 坐标减半实现的。现在，最终的坐标正确地将光标的位置从窗体坐标映射到 OpenGL 坐标。如果把光标放到 OpenGL 控件的中心，Input 类将报告位置(0,0)。

在结束对窗体代码的讨论之前，还需要编写最后一项功能：必须把新的输入对象添加到圆的状态的构造函数中。

```
_system.AddState("circle_state", new CircleIntersectionState(_input));
```

还必须修改状态的构造函数。

```
Input _input;
public CircleIntersectionState(Input input)
{
    _input = input;
}
```

将输入传递给状态以后，就可以使用光标的位置了。有必要确认代码都在正确工作。最简单的方法是，使用 OpenGL 在光标所在的位置绘点。

```
public void Render()
{
    Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    _circle.Draw();

    // Draw the mouse cursor as a point
    Gl.glPointSize(5);
    Gl.glBegin(Gl.GL_POINTS);
    {
        Gl.glVertex2f(_input.MousePosition.X,
            _input.MousePosition.Y);
    }
}
```

```
    Gl.glEnd();  
}
```

运行程序，光标总是会带着一个小方块。注意代码中添加了一个 `glClear` 命令。试着删除 `glClear` 命令，看看会发生什么事情。

现在鼠标指针已经可以工作，接下来返回到相交性代码。状态的更新循环将执行相交性测试。

```
public void Update(double elapsedTime)  
{  
    if (_circle.Intersects(_input.MousePosition))  
    {  
        _circle.Color = new Color(1, 0, 0, 1);  
    }  
    else  
    {  
        // If the circle's not intersected turn it back to white.  
        _circle.Color = new Color(1, 1, 1, 1);  
    }  
}
```

这是 `intersect` 函数的用法，接下来就实际编写该函数。这个测试需要使用大量向量操作，所以把指针对象转换成了一个向量。

```
public bool Intersects(Point point)  
{  
    // Change point to a vector  
    Vector vPoint = new Vector(point.X, point.Y, 0);  
    Vector vFromCircleToPoint = Position - vPoint;  
    double distance = vFromCircleToPoint.Length();  
  
    if (distance > Radius)  
    {  
        return false;  
    }  
    return true;  
}
```

运行程序，观察光标移入和移出圆时会发生什么情况。

8.3.2 矩形

矩形的相交测试代码不需要非常完善。我们需要的矩形只有按钮，而按钮总是会与轴对齐。这样，代码就比处理任意对齐的矩形时简单了很多。

如果点在矩形左边的右侧，右边的左侧，上边的下侧，底边的上侧，那么该点在矩形

内。可以采取与圆的示例相似的方法从视觉上显示矩形与点的关系。

```
class RectangleIntersectionState : IGameObject
{
    Input _input;
    Rectangle _rectangle = new Rectangle(new Vector(0,0,0), new Vector
(200, 200,0));
    public RectangleIntersectionState(Input input)
    {
        _input = input;
    }

    #region IGameObject Members

    public void Update(double elapsedTime)
    {
        if (_rectangle.Intersects(_input.MousePosition))
        {
            _rectangle.Color = new Color(1, 0, 0, 1);
        }
        else
        {
            // If the circle's not intersected turn it back to white.
            _rectangle.Color = new Color(1, 1, 1, 1);
        }
    }

    public void Render()
    {
        _rectangle.Render();
    }
    #endregion
}
```

这个状态与之前关于圆的示例的状态很类似。记住，要使它作为默认状态加载。矩形本身和圆一样，是使用线循环构成的。

```
public class Rectangle
{
    Vector BottomLeft { get; set; }
    Vector TopRight { get; set; }
    Color _color = new Color(1, 1, 1, 1);
    public Color Color
    {
```

```
    get { return _color; }
    set { _color = value; }
}

public Rectangle(Vector bottomLeft, Vector topRight)
{
    BottomLeft = bottomLeft;
    TopRight = topRight;
}

public void Render()
{
    Gl.glColor3f(_color.Red, _color.Green, _color.Blue);
    Gl.glBegin(Gl.GL_LINE_LOOP);
    {
        Gl.glVertex2d(BottomLeft.X, BottomLeft.Y);
        Gl.glVertex2d(BottomLeft.X, TopRight.Y);
        Gl.glVertex2d(TopRight.X, TopRight.Y);
        Gl.glVertex2d(TopRight.X, BottomLeft.Y);
    }
    Gl.glEnd();
}
}
```

rectangle 类可以创建和绘制矩形。唯一还缺少的函数是最重要的 **intersect** 函数。

```
public bool Intersects(Point point)
{
    if (
        point.X >= BottomLeft.X &&
        point.X <= TopRight.X &&
        point.Y <= TopRight.Y &&
        point.Y >= BottomLeft.Y)
    {
        return true;
    }
    return false;
}
```

运行程序，将光标移入矩形。与圆的示例一样，矩形会变成红色，表明相交性代码起作用了。

8.4 补间

补间(tween)是指随时间将一个值改为另一个值。补间可以用来创建动画、改变位置、颜色、大小或其他你可能想到的值。补间在 Adobe Flash 中的应用最为广泛，Adobe Flash 中也提供了许多内置的补间函数。

8.4.1 补间概述

通过一个示例了解补间的工作原理是最简单的，然后我们将深入探究其细节。可以在已有的代码库中使用这个状态，但是如果想要创建一个新项目，就要添加对 `Tao.DevIL` 的引用，以及添加 `Sprite`、`Texture` 和 `TextureManager` 类。

```
class TweenTestState: IGameObject
{
    Tween _tween = new Tween(0, 256, 5);
    Sprite _sprite = new Sprite();

    public SpriteTweenState(TextureManager textureManager)
    {
        _sprite.Texture = textureManager.Get("face");
        _sprite.SetHeight(0);
        _sprite.SetWidth(0);
    }

    public void Render()
    {
        // Rendering code goes here.
    }

    public void Update(double elapsedTime)
    {
        if (_tween.IsFinished() != true)
        {
            _tween.Update(elapsedTime);
            _sprite.SetWidth((float)_tween.Value());
            _sprite.SetHeight((float)_tween.Value());
        }
    }
}
```

代码中使用 `Tween` 对象，在 5s 内使一个精灵从无变到大小为 256。这里的 `Tween` 构造函数接受 3 个参数。第一个参数是初始值，第二个参数是目标值，最后一个参数是从初始值变化到目标值所需的时间。

更新循环检查补间是否完成。如果没有，就更新补间。精灵的宽度和高度被设为补间的值，该值在 0~256 之间。

在上面的示例中，补间线性地从初始值变化到最终值。这意味着在 2.5s 后，补间的值变为 128。补间并非必须是线性的，在变化到目标值的过程中，它们可以逐渐加快变化速度，或者逐渐减慢变化速度。通过使用一个时间函数表示位置，可以获得这种改变补间类型的能力。

```
public void function(double time)
{
    // Create a position using the time value
    return position;
}
```

实际的补间函数要比上面的代码复杂一些。下面的函数用于执行线性插值。

```
public static double Linear(double timePassed, double start, double
distance,double duration)
{
    return distance * timePassed / duration + start;
}
```

补间代码默认使用线性补间，但是可以添加许多不同的补间。图 8-20 显示了许多这样的补间。

Internet 上有许多 Flash 补间函数，将它们转换成 C# 代码并不困难。

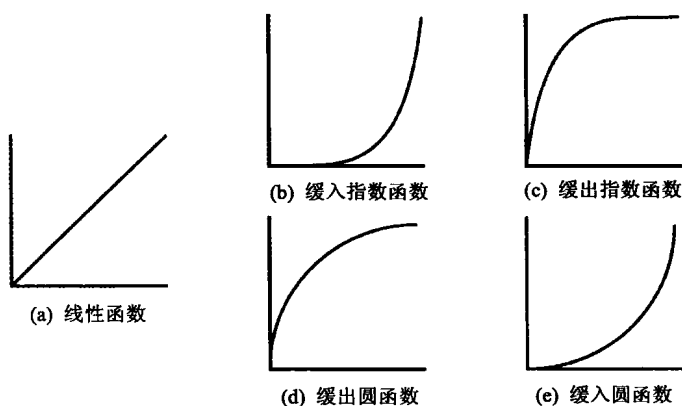


图 8-20 5 类补间函数

8.4.2 Tween 类

Tween 类封装了表示变量的值随时间变化的这种思想。该类的完整代码如下所示。

```
public class Tween
{
```

```
double _original;
double _distance;
double _current;
double _totalTimePassed = 0;
double _totalDuration = 5;
bool _finished = false;
TweenFunction _tweenF = null;
public delegate double TweenFunction(double timePassed, double start,
double distance, double duration);

public double Value()
{
    return _current;
}
public bool IsFinished()
{
    return _finished;
}

public static double Linear(double timePassed, double start, double
distance, double duration)
{
    return distance * timePassed / duration + start;
}

public Tween(double start, double end, double time)
{
    Construct(start, end, time, Tween.Linear);
}

public Tween(double start, double end, double time, TweenFunction
tweenF)
{
    Construct(start, end, time, tweenF);
}

public void Construct(double start, double end, double time, TweenFunction
tweenF)
{
    _distance = end - start;
    _original = start;
    _current = start;
    _totalDuration = time;
```

```

        _tweenF = tweenF;
    }

    public void Update(double elapsedTime)
    {
        _totalTimePassed += elapsedTime;
        _current = _tweenF(_totalTimePassed, _original, _distance,
            _totalDuration);

        if (_totalTimePassed > _totalDuration)
        {
            _current = _original + _distance;
            _finished = true;
        }
    }
}

```

Tween 类有两个构造函数，它们都调用了 **Construct** 方法。构造函数允许用户指定补间函数的类型，或者使用默认的随时间线性变化。补间函数通过 **TweenFunction** 委托定义。**TweenFunction** 委托在这个类中的唯一实现是 **Linear** 函数。在默认构造函数中可以看到它的用法。

```
Construct(start, end, time, Tween.Linear);
```

Construct 方法记录了补间的初始值、最终值和执行补间操作的时间。也可以传入一个补间函数来确定值随时间如何变化。**Construct** 方法可记录这些值，并计算出从初始值到最终值之间的距离。该距离将被传递给相关的补间函数。

Tween 对象将在每一帧中更新，经过的时间在每一次更新调用中将被累加。这样 **Tween** 对象就知道补间的进度。补间函数的委托会修改补间的当前值。最后，更新函数会检查补间是否结束，如果结束，则将结束标记设为 **true**。

如果只有一个线性函数，**Tween** 类就没有那么有价值了。下面列出了其他一些可以添加到 **Tween** 类的函数，图 8-20 中也显示了它们。

```

public static double EaseOutExpo(double timePassed, double start, double
distance, double duration)
{
    if (timePassed == duration)
    {
        return start + distance;
    }
    return distance * (-Math.Pow(2, -10 * timePassed / duration) + 1) + start;
}

```

```

public static double EaseInExpo(double timePassed, double start, double
distance, double duration)
{
    if (timePassed == 0)
    {
        return start;
    }
    else
    {
        return distance * Math.Pow(2, 10 * (timePassed / duration - 1)) + start;
    }
}

public static double EaseOutCirc(double timePassed, double start, double
distance, double duration)
{
    return distance * Math.Sqrt(1 - (timePassed = timePassed / duration - 1) *
timePassed) + start;
}

public static double EaseInCirc(double timePassed, double start, double
distance, double duration)
{
    return -distance * (Math.Sqrt(1 - (timePassed /= duration) * timePassed)
- 1) + start;
}

```

8.4.3 使用补间

现在已经完成了 Tween 类的创建, 接下来就展示该类的一些强大的能力。和前面一样, 首先需要创建一个新状态, 将其命名为 TweenTestState。

这个状态需要将本书前面使用的笑脸纹理添加到项目中, 并将其 Copy To Output Directory 属性设为 Copy if newer。在窗体构造函数中, 应该将笑脸纹理加载到 TextureManager 中。

```
_textureManager.LoadTexture("face", "face_alpha.tif");
```

在加载了纹理后, 可以使用它在 TweenTestState 类中创建一个精灵。

```

class TweenTestState : IGameObject
{
    Sprite _faceSprite = new Sprite();
    Renderer _renderer = new Renderer();
    Tween _tween = new Tween(0, 256, 5);
    public TweenTestState(TextureManager textureManager)
    {

```

```

        _faceSprite.Texture = textureManager.Get("face");
    }

    #region IGameObject Members

    public void Process(double elapsedTime)
    {
        if (_tween.IsFinished() != true)
        {
            _tween.Process(elapsedTime);
            _faceSprite.SetWidth((float)_tween.Value());
            _faceSprite.SetHeight((float)_tween.Value());
        }
    }

    public void Render()
    {
        Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        _renderer.DrawSprite(_faceSprite);
        _renderer.Render();
    }
    #endregion
}

```

补间会将精灵的宽度和高度由 0 一直变为 256。运行代码并查看动画。精灵将以一种平滑美观的方式逐渐变大。接下来只对代码做一些小小的修改，但结果却会产生巨大变化。

```
Tween _tween = new Tween(0, 256, 5, Tween.EaseInExpo);
```

再次运行程序，现在精灵会逐渐变大。然后变化会逐渐加快，直到变到最大。一点小小的修改，动画的播放方式却彻底改变了。这是调整已有动画的一种很好的方法。尝试其他的补间函数，查看它们的用途，并试着修改其余的参数，以便了解补间函数的工作原理。

```

Tween _alphaTween = new Tween(0, 1, 5, Tween.EaseInCirc);
Color _color = new Color(1, 1, 1, 0);
public void Process(double elapsedTime)
{
    if (_tween.IsFinished() != true)
    {
        _tween.Process(elapsedTime);
        _faceSprite.SetWidth((float)_tween.Value());
        _faceSprite.SetHeight((float)_tween.Value());
    }
}

```

```
if (_alphaTween.IsFinished() != true)
{
    _alphaTween.Process(elapsedTime);
    _color.Alpha = (float)_alphaTween.Value();
    _faceSprite.SetColor(_color);
}
}
```

这里添加了另外一个补间。该补间使精灵的透明度从 0 逐渐变为完全不透明。这是淡入文本的一种很好的方法。精灵的位置也可以通过补间改变。尝试使用补间，将精灵从屏幕外移动到屏幕的中央。另外一个很好的示例是将精灵的透明度从 0 变化为 1，然后引发另一个补间，将透明度从 1 变化为 0。这些函数可以逐个调用，使补间循环执行。

8.5 矩阵

图形编程中到处会用到矩阵。矩阵有很多种应用，但是本节将只关注与图形和游戏编程有关的那一部分。

矩阵是一种数学结构，为描述或者执行 3D 模型或者由四方形构成的精灵上的众多操作提供了一种便捷的方法。这些操作包括变换(在 3D 空间内移动模型)、旋转、缩放、偏航(使形状“靠”向特定的方向)和投影(例如，将 3D 空间中的点转换到 2D 空间中的点)。许多这样的操作都可以手动完成，例如，精灵类已经通过把向量添加到每个顶点位置而执行了平移。

与向量相比，矩阵在执行这类操作时有几个优势。实现不同操作的矩阵可以相互结合，得到单个矩阵，该矩阵定义了这些操作的组合。例如，一个矩阵将模型旋转 90° ，一个矩阵将模型放大两倍，一个矩阵将模型向左移动两英里，这几个矩阵可以结合成一个同时完成所有操作的矩阵。矩阵乘法可以实现这种操作，即将多个矩阵相乘，得到的结果矩阵是所有操作的组合。

组合操作并不是矩阵唯一的优势。矩阵可以求出逆矩阵，它执行的操作与原矩阵执行的操作相反。如果创建一个绕 Z 轴旋转 5° 的旋转矩阵，然后对一个模型应用该矩阵，那么通过对矩阵求逆矩阵，然后将这个逆矩阵应用到模型上，可以使模型返回到原来的位置。通过将模型的每个顶点与矩阵相乘，可以将矩阵应用到模型上。

8.5.1 矩阵的定义

矩阵是数值组成的网格。与向量类似，矩阵也可以有多个维。我们定义的向量的维数为 3，其中的 3 个维度分别是 X、Y 和 Z。在 3D 图形学中，最常用的矩阵是 4×4 矩阵，如图 8-21 所示。

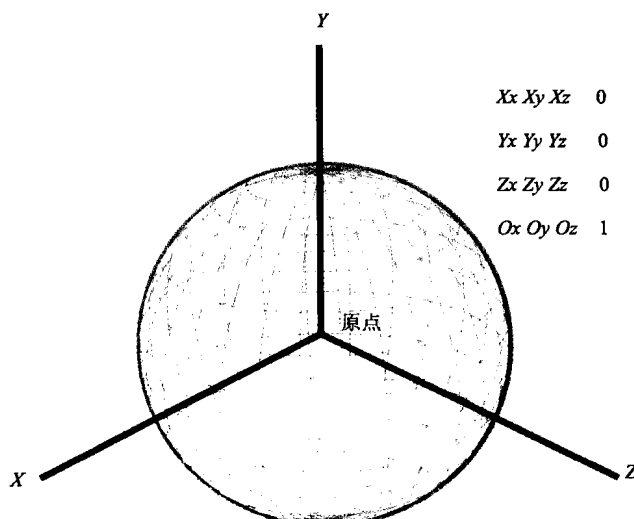


图 8-21 矩阵的视觉表示

图 8-21 使用 3 个轴向量和一个位置向量(原点)来表示矩阵。 $XxXyXz$ 描述了 x 向量, $YxYyYz$ 描述了 y 向量, $ZxZyZz$ 描述了 z 向量, $OxOyOz$ 描述了原点向量。这些向量是模型的轴及其原点。可以使用它们来快速确定对象的位置以及对象在 X 、 Y 和 Z 轴上面对的方向。如果所有的轴都被归一化, 那么对象不被缩放; 如果轴的长度都为 2, 那么对象被放大到原来的两倍。这种可视化的思考矩阵的方法使得它们更易于使用。

图 8-21 中的最后一列是 $[0,0,0,1]$ 。这个列中的值不会改变, 它们只在计算投影(从 3D 空间变换到 2D 空间)时使用。这种投影操作不太常见, 所以我们将使用的矩阵为 4×3 矩阵。最后一列将始终为 $[0,0,0,1]$ 。

在介绍矩阵操作之前, 先创建一个基本的类。

```
public class Matrix
{
    double _m11, _m12, _m13;
    double _m21, _m22, _m23;
    double _m31, _m32, _m33;
    double _m41, _m42, _m43;
}
```

这个 **Matrix** 类是一个 4×3 矩阵。它的成员变量的分布方式与图 8-18 类似, 但是没有第 4 列(我们知道第 4 列总是 $[0,0,0,1]$, 所以不需要存在这几个值)。Vector、Color 和 Point 都是结构, 但是 **Matrix** 是一个更大的结构, 所以把它声明成了一个类。

在下面的小节中, 我们将为 **Matrix** 类添加各种操作。

8.5.2 单位矩阵

单位矩阵是这样的一种矩阵: 当将它与另外一个矩阵相乘时, 它不会修改那个矩阵。数字 1 是实数中的一个单位, 将任何数值与 1 相乘都会得到原来的数值。

单位矩阵是方阵。下面是 3×3 矩阵和 4×4 矩阵的单位矩阵。

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

创建一个矩阵操作时，单位矩阵是一个绝佳的起点。对模型应用单位矩阵时，顶点不会发生改变，所以只有在单位矩阵之上执行的操作才会得到执行。如果对任何模型应用全零矩阵，该模型将会消失，所有的顶点都将缩减为一个奇点，就像黑洞那样。矩阵类应该默认被初始化为单位矩阵。

将如下定义添加到矩阵类中。

```
public static readonly Matrix Identity =
    new Matrix(new Vector(1, 0, 0),
               new Vector(0, 1, 0),
               new Vector(0, 0, 1),
               new Vector(0, 0, 1));

public Matrix() : this (Identity)
{
}

public Matrix(Matrix m)
{
    _m11 = m._m11;
    _m12 = m._m12;
    _m13 = m._m13;

    _m21 = m._m21;
    _m22 = m._m22;
    _m23 = m._m23;
    _m31 = m._m31;
    _m32 = m._m32;
    _m33 = m._m33;
    _m41 = m._m41;
    _m42 = m._m42;
    _m43 = m._m43;
}

public Matrix(Vector x, Vector y, Vector z, Vector o)
{
    _m11 = x.X; _m12 = x.Y; _m13 = x.Z;
```



```

    _m21 = y.X; _m22 = y.Y; _m23 = y.Z;
    _m31 = z.X; _m32 = z.Y; _m33 = z.Z;
    _m41 = o.X; _m42 = o.Y; _m43 = o.Z;
}

```

这段代码添加了一个常量单位矩阵和几个构造函数。默认的构造函数通过把单位矩阵传递给拷贝构造函数，将成员初始化为单位矩阵。第二个构造函数就是拷贝构造函数。拷贝构造函数是通过一个参数调用的构造函数，参数的类型与被构造的对象的类型相同。拷贝构造函数复制参数的所有成员数据，所以被创建的对象与参数完全相同。最后一个构造函数接受4个参数，前3个参数是代表每个轴的向量，最后一个参数是代表原点的向量。

8.5.3 矩阵乘法和矩阵与向量的乘法

Matrix 类最重要的方法是其乘法方法。乘法用于组合矩阵和顶点位置的变化。只有当第一个矩阵的宽度和第二个矩阵的高度相同时，才可以执行矩阵乘法。矩阵乘法和矩阵与向量的乘法是使用相同的算法执行的。

乘法的定义如下。

$$C_{i,k} = A_{i,j}B_{j,k}$$

C 是乘法的结果，*i* 是矩阵 **A** 中的行数，*k* 是矩阵 **B** 中的列数。*j* 是 *i* 和 *k* 的可能的和的数量。在矩阵乘法中，相乘矩阵原来的形状不同，结果矩阵的形状也会不同。同样，不要因为这些数学公式看上去让人生畏而感到担心，因为知道如何和何时使用矩阵才是这里最重要的。

在我们的矩阵中，如果包含了最后一列[0,0,0,1]，那么它的宽度与高度相等。因此，根据矩阵乘法的定义，编写矩阵乘法的代码如下所示。

```

public static Matrix operator *(Matrix mA, Matrix mB)
{
    Matrix result = new Matrix();

    result._m11 = mA._m11 * mB._m11 + mA._m12 * mB._m21 + mA._m13 * mB._m31;
    result._m12 = mA._m11 * mB._m12 + mA._m12 * mB._m22 + mA._m13 * mB._m32;
    result._m13 = mA._m11 * mB._m13 + mA._m12 * mB._m23 + mA._m13 * mB._m33;

    result._m21 = mA._m21 * mB._m11 + mA._m22 * mB._m21 + mA._m23 * mB._m31;
    result._m22 = mA._m21 * mB._m12 + mA._m22 * mB._m22 + mA._m23 * mB._m32;
    result._m23 = mA._m21 * mB._m13 + mA._m22 * mB._m23 + mA._m23 * mB._m33;

    result._m31 = mA._m31 * mB._m11 + mA._m32 * mB._m21 + mA._m33 * mB._m31;
    result._m32 = mA._m31 * mB._m12 + mA._m32 * mB._m22 + mA._m33 * mB._m32;
    result._m33 = mA._m31 * mB._m13 + mA._m32 * mB._m23 + mA._m33 * mB._m33;

    result._m41 = mA._m41 * mB._m11 + mA._m42 * mB._m21 + mA._m43 * mA._m31 +
    mB._m41;
}

```

```

        result._m42 = mA._m41 * mB._m12 + mA._m42 * mB._m22 + mA._m43 * mB._m32 +
mB._m42;
        result._m43 = mA._m41 * mB._m13 + mA._m42 * mB._m23 + mA._m43 * mB._m33 +
mB._m43;

        return result;
    }

```

向量与矩阵的乘法类似。

```

public static Vector operator *(Vector v, Matrix m)
{

    return new Vector(v.X * m._m11 + v.Y * m._m21 + v.Z * m._m31 + m._m41,
        v.X * m._m12 + v.Y * m._m22 + v.Z * m._m32 + m._m42,
        v.X * m._m13 + v.Y * m._m23 + v.Z * m._m33 + m._m43);
}

```

8.5.4 平移和缩放

平移是一个非常简单的操作。向量的最后一行是对象的原点，平移操作将使用这一行。创建一个矩阵来修改平移时，只需要修改单位矩阵的最后一行。平移矩阵如下所示。

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ t_x & t_y & t_z & 1 \end{bmatrix}$$

其代码也很简单。

```

public void SetTranslation(Vector translation)
{
    _m41 = translation.X;
    _m42 = translation.Y;
    _m43 = translation.Z;
}

public Vector GetTranslation()
{
    return new Vector(_m41, _m42, _m43);
}

```

缩放矩阵理解起来也很简单。记住，每一行代表一个轴。可以把矩阵的第一行看作一个向量。这个向量是被缩放对象的 X 轴， X_x 维度越大，在 X 轴方向上缩放得也越大。第二行是 Y 轴，第三行是 Z 轴，它们也可以被看作向量。如果每个向量的长度都为 2，那么将沿着每个轴均匀地缩放对象。

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
public void SetScale(Vector scale)
{
    _m11 = scale.X;
    _m22 = scale.Y;
    _m33 = scale.Z;
}

public Vector GetScale()
{
    Vector result = new Vector();
    result.X = (new Vector(_m11, _m12, _m13)).Length();
    result.Y = (new Vector(_m21, _m22, _m23)).Length();
    result.Z = (new Vector(_m31, _m32, _m33)).Length();
    return result;
}
```

缩放也可以是非均匀缩放，在某个轴上缩放的程度可能比其他轴上缩放的程度更大。为了找出在某个轴上的缩放程度，只需取出相应的行并把它转换成一个向量，然后应用向量的求长度操作，得到的结果就是缩放的量。

8.5.5 旋转

围绕任意一个轴进行旋转的数学算法比前面介绍的要复杂得多。知道旋转的具体工作原理并不重要，重要的是知道旋转的结果。旋转矩阵由一个轴 \mathbf{u} 和一个标量值(θ)构成，其中 \mathbf{u} 被定义为一个归一化向量，而 θ 用弧度作为单位描述了旋转量。如果想要旋转的模型是一个瓶塞，那么归一化向量 \mathbf{u} 可以用一个穿过瓶塞的针表示。角度(θ)代表针的旋转量，针的旋转会带动瓶塞的旋转。

$$\begin{bmatrix} 1 + (1 - \cos\theta)(u_x^2 - 1) & (1 - \cos\theta)u_xu_y - u_z\sin\theta & (1 - \cos\theta)u_xu_z + u_y\sin\theta & 0 \\ (1 - \cos\theta)u_xu_y + u_z\sin\theta & 1 + (1 - \cos\theta)(u_y^2 - 1) & (1 - \cos\theta)u_yu_z - u_x\sin\theta & 0 \\ (1 - \cos\theta)u_xu_z - u_y\sin\theta & (1 - \cos\theta)u_yu_z + u_x\sin\theta & 1 + (1 - \cos\theta)(u_z^2 - 1) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

```
public void SetRotate(Vector axis, double angle)
{
    double angleSin = Math.Sin(angle);
    double angleCos = Math.Cos(angle);
    double a = 1.0 - angleCos;
    double ax = a * axis.X;
```

```

double ay = a * axis.Y;
double az = a * axis.Z;
_m11 = ax * axis.X + angleCos;
_m12 = ax * axis.Y + axis.Z * angleSin;
_m13 = ax * axis.Z - axis.Y * angleSin;

_m21 = ay * axis.X - axis.Z * angleSin;
_m22 = ay * axis.Y + angleCos;
_m23 = ay * axis.Z + axis.X * angleSin;

_m31 = az * axis.X + axis.Y * angleSin;
_m32 = az * axis.Y - axis.X * angleSin;
_m33 = az * axis.Z + angleCos;
}

```

如果在一帧中多次使用正弦和余弦函数，开销是很大的，因此，代码中将它们的使用降到了最低。轴向量应该被归一化，但是 **SetRotate** 中没有对此进行检查。

8.5.6 求逆矩阵

求逆矩阵对于逆转给定矩阵的操作非常有用。为计算逆矩阵，首先需要确定矩阵的行列式。每个方阵都有自己的行列式。只有行列式不为 0 时，矩阵才是可逆的。

```

public double Determinate()
{
    return _m11 * (_m22 * _m33 - _m23 * _m32) +
           _m12 * (_m23 * _m31 - _m21 * _m33) +
           _m13 * (_m21 * _m32 - _m22 * _m31);
}

```

然后这个行列式可以用于计算矩阵上部的 3×3 部分，即缩放和旋转部分。矩阵的平移部分则是手动计算的。

```

public Matrix Inverse()
{
    double determinate = Determinate();
    System.Diagnostics.Debug.Assert(Math.Abs(determinate) >
    Double.Epsilon,
    "No determinate");

    double oneOverDet = 1.0 / determinate;

    Matrix result = new Matrix();
    result._m11 = (_m22 * _m33 - _m23 * _m32) * oneOverDet;
    result._m12 = (_m13 * _m32 - _m12 * _m33) * oneOverDet;

```

```

result._m13 = (_m12 * _m23 - _m13 * _m22) * oneOverDet;

result._m21 = (_m23 * _m31 - _m21 * _m33) * oneOverDet;
result._m22 = (_m11 * _m33 - _m13 * _m31) * oneOverDet;
result._m23 = (_m13 * _m21 - _m11 * _m23) * oneOverDet;

result._m31 = (_m21 * _m32 - _m22 * _m31) * oneOverDet;
result._m32 = (_m12 * _m31 - _m11 * _m32) * oneOverDet;
result._m33 = (_m11 * _m22 - _m12 * _m21) * oneOverDet;

result._m41 = -(_m41 * result._m11 + _m42 * result._m21 + _m43 *
result._m31);
result._m42 = -(_m41 * result._m12 + _m42 * result._m22 + _m43 *
result._m32);
result._m43 = -(_m41 * result._m13 + _m42 * result._m23 + _m43 *
result._m33);

return result;
}

```

使用这段代码，可以对我们将使用的任何矩阵求逆。

关于 **Matrix** 类的内容就是这些，现在这个类对于 2D 和 3D 应用程序都适用。接下来就实际使用这个类。

8.5.7 对精灵执行矩阵操作

创建一个新的游戏状态 **MatrixTestState**。这个状态将绘制一个精灵，并对该精灵应用各种矩阵。下面是绘制精灵的代码，你应该已经很熟悉这段代码了。

```

class MatrixTestState : IGameObject
{
    Sprite _faceSprite = new Sprite();
    Renderer _renderer = new Renderer();
    public MatrixTestState(TextureManager textureManager)
    {
        _faceSprite.Texture = textureManager.Get("face");
        Gl.glEnable(Gl.GL_TEXTURE_2D);
    }
    public void Render()
    {
        Gl.glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        _renderer.DrawSprite(_faceSprite);
        _renderer.Render();
    }
}

```

```
    }  
    public void Update(double elapsedTime)  
    {  
    }  
}
```

代码中使用了前面章节中用到的脸部精灵。矩阵将在 `MatrixTestState` 构造函数中应用到精灵上。

```
Matrix m = new Matrix();  
m.SetRotate(new Vector(0, 0, 1), Math.PI/5);  
  
for (int i = 0; i < _faceSprite.VertexPositions.Length; i++ )  
{  
    _faceSprite.VertexPositions[i] *= m;  
}
```

运行代码，可以注意到脸部已经被旋转过了。旋转是沿着 Z 轴(0,0,1)执行的，这是从屏幕内部指向外部的轴。把脸部精灵想象成屏幕上的一张纸。为了进行旋转，你在纸上按了一个图钉，使其附着到屏幕上了。图钉就代表 Z 轴。旋转纸精灵就是在沿着该轴旋转。对于正射投影中的 2D 对象， Y 轴和 Z 轴对于旋转来说没有什么用处，但是在 3D 游戏中就不一样了。任何一个归一化的轴都可以用于旋转对象，不只是 X 、 Y 和 Z 这 3 个主轴。

在代码示例中，旋转量以弧度 `Math.PI/5` 给出，这个值等于 36° 。旋转矩阵将被应用到组成精灵的每个顶点。我们现在已经使用了一个稍微旋转了精灵的矩阵。接下来就添加一个缩放精灵的矩阵。通过使用矩阵乘法，缩放矩阵将与旋转矩阵结合起来。修改构造函数中的已有代码，使其如下所示。

```
Matrix m = new Matrix();  
m.SetRotate(new Vector(0, 0, 1), Math.PI/5);  
  
Matrix mScale = new Matrix();  
mScale.SetScale(new Vector(2.0, 2.0, 0.0));  
  
m *= mScale;  
  
for (int i = 0; i < _faceSprite.VertexPositions.Length; i++ )  
{  
    _faceSprite.VertexPositions[i] *= m;  
}
```

这段代码创建的缩放矩阵将对象在 X 轴和 Y 轴上放大一倍。这个矩阵和旋转矩阵通过矩阵乘法结合到一起，然后把结果赋值给矩阵 `m`。然后这个合并后的矩阵 `m` 被应用到脸部精灵，以缩放和旋转该精灵。矩阵乘法的顺序很重要：将矩阵 `a` 和矩阵 `b` 相乘的结果不一定与将矩阵 `b` 和矩阵 `a` 相乘的结果相同。尝试采用不同的矩阵，以熟悉它们之间相互组合

后可以得到什么样的结果。

最后的代码段将演示逆矩阵。逆矩阵会逆转一个矩阵操作。如果将旋转-缩放矩阵乘以其逆矩阵，结果将是一个单位矩阵。对精灵应用单位矩阵时，不会产生任何效果。

```
Matrix m = new Matrix();
m.SetRotate(new Vector(0, 0, 1), Math.PI/5);

Matrix mScale = new Matrix();
mScale.SetScale(new Vector(2.0, 2.0, 2.0));

m *= mScale;
Vector scale = m.GetScale();
m *= m.Inverse();

for (int i = 0; i < _faceSprite.VertexPositions.Length; i++)
{
    _faceSprite.VertexPositions[i] *= m;
}
```

你还可以尝试平移矩阵，并试着以不同的顺序组合矩阵，看看会得到什么效果。

8.5.8 修改精灵来使用矩阵

精灵现在有一个平移方法 `SetPosition`，但是它没有类似的 `SetScale` 或 `SetRotation` 方法。这些方法很有用，结合补间函数使用它们可以得到奇妙的效果，因此有必要添加它们。修改精灵类很简单，但是还需要添加一些额外的成员和方法。

```
double _scaleX = 1;
double _scaleY = 1;
double _rotation = 0;
double _positionX = 0;
double _positionY = 0;

public void ApplyMatrix(Matrix m)
{
    for (int i = 0; i < VertexPositions.Length; i++)
    {
        VertexPositions[i] *= m;
    }
}

public void SetPosition(Vector position)
{
    Matrix m = new Matrix();
    m.SetTranslation(new Vector(_positionX, _positionY, 0));
```

```
        ApplyMatrix(m.Inverse());
        m.SetTranslation(position);
        ApplyMatrix(m);
        _positionX = position.X;
        _positionY = position.Y;
    }

    public void SetScale(double x, double y)
    {
        double oldX = _positionX;
        double oldY = _positionY;
        SetPosition(0, 0);
        Matrix mScale = new Matrix();
        mScale.SetScale(new Vector(_scaleX, _scaleY, 1));
        mScale = mScale.Inverse();
        ApplyMatrix(mScale);
        mScale = new Matrix();
        mScale.SetScale(new Vector(x, y, 1));
        ApplyMatrix(mScale);
        SetPosition(oldX, oldY);
        _scaleX = x;
        _scaleY = y;
    }

    public void SetRotation(double rotation)
    {
        double oldX = _positionX;
        double oldY = _positionY;
        SetPosition(0, 0);
        Matrix mRot = new Matrix();
        mRot.SetRotate(new Vector(0, 0, 1), _rotation);
        ApplyMatrix(mRot.Inverse());
        mRot = new Matrix();
        mRot.SetRotate(new Vector(0, 0, 1), rotation);
        ApplyMatrix(mRot);
        SetPosition(oldX, oldY);
        _rotation = rotation;
    }
}
```

新添加的这些函数使得程序员可以根据需要旋转和缩放精灵。它们的开销更高一些，但是把这些函数转移到3D模型中很简单。可以对这些函数进行整理和缩减，以获得性能提升。

8.5.9 优化

将注意力放到游戏上，只在必要时进行优化，这一点十分重要。对于大多数游戏，**Matrix**类已经足够快，但是程序员都很喜欢进行优化，下面就此给出了一些提示。矩阵涉及大量运算，所以减少矩阵的使用量一般来说是一个好主意。优化程度最高的代码是从不会运行的代码。

现在，每个精灵由两个三角形组成，总共有 6 个顶点，但是实际上，用 4 个顶点就可以创建一个精灵。在这一方面，一个不错的起点是查阅 OpenGL 文档中的索引缓冲区函数。

现代 CPU 使用专门的硬件 SIMD(Single Instruction, Multiple Data, 单指令多数据)来执行矩阵和向量运算。SIMD 指令会显著增加矩阵计算的速度。但是，在编写本书时，只有 C# 的 Mono 实现支持 SIMD 操作。

大多数测试程序都会在调试模式下生成，发布模式要快得多。如果在 Solution Explorer 中右击项目名称并在弹出的快捷菜单中选择 Properties 命令，还可以使用其中的一个选项来打开优化。

垃圾回收是使 C# 比 C++ 这样的语言更慢的一项特性。避免垃圾回收导致速度变慢的最佳方式是减少主循环中创建的对象数量。每当使用关键字 **new** 时，就创建了一个对象。最好是在构造函数中创建对象一次，或者将对象定义为某个对象的成员，而不是在处理循环或者渲染循环中创建它。

一些矩阵操作会创建矩阵。这使得它们更加易于使用，对象创建也被优化了，但是还可以使它们更加高效。在调试时，右击窗口后在弹出的快捷菜单中有一个命令 Go to Disassembly。这会显示每一行 C# 代码生成的 IL(intermediate Language, 中间语言)。IL 指令数越少，代码运行得越快，前提是发布生成优化没有移除这些 IL 指令。遗憾的是，反汇编中不会显示编译器执行的任何优化。

第 9 章

创建游戏引擎

前面的章节中开发出了非常好的可重用的代码。应该把这些代码放在自己的引擎库中，而不是在各个项目中不断复制它们。这样就可以在多个游戏项目间共享这些代码，使得所有的修改和改进只需在一个地方完成。

9.1 新的游戏引擎项目

游戏引擎项目与我们到现在为止创建的任何项目都不同。这个游戏不会自己运行，相反，它由一个独立的游戏项目使用。也就是说，游戏引擎更应该被称为一个库，而不是一个程序。学习更多技术后，可以在这个库中添加更多的代码，并根据自己的需要调整这个库。图 9-1 显示了几个项目如何同时使用这个库。

关闭 Visual Studio 中的全部项目，然后选择 File|New|Project 命令，创建一个新项目。这将打开如图 9-2 所示的对话框。选择 Class Library 选项，因为游戏引擎是其他项目使用的一个库。在名称旁边，我简单地输入了 Engine，但是你可以为自己的游戏引擎使用任意名称。

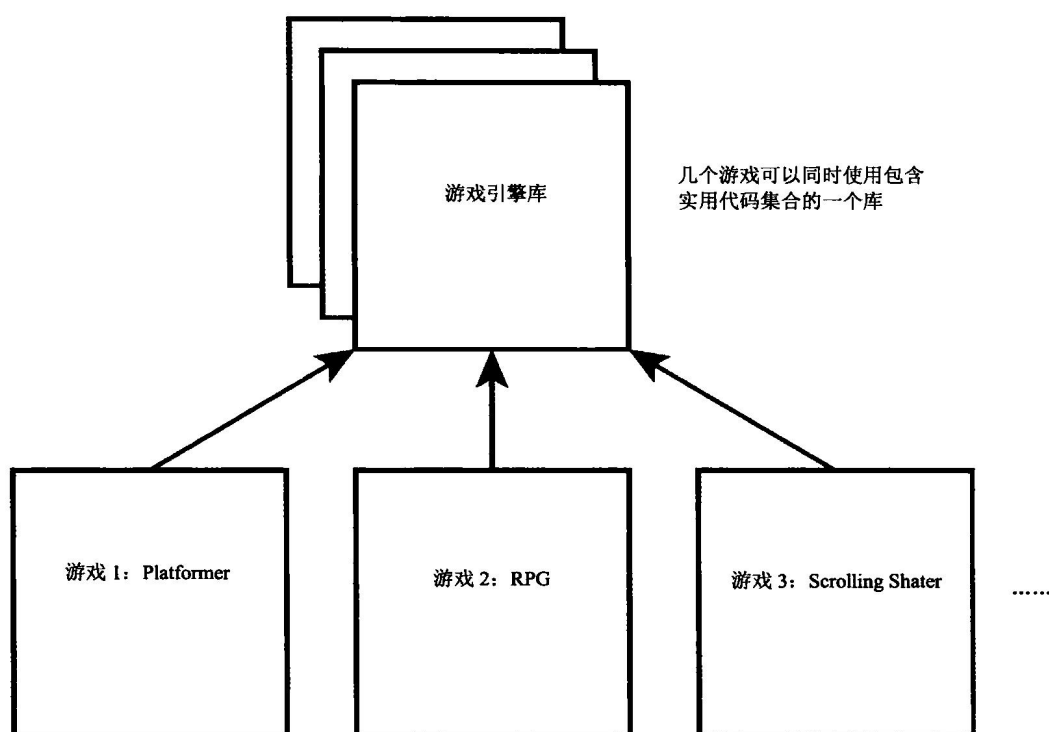


图 9-1 使用游戏引擎库

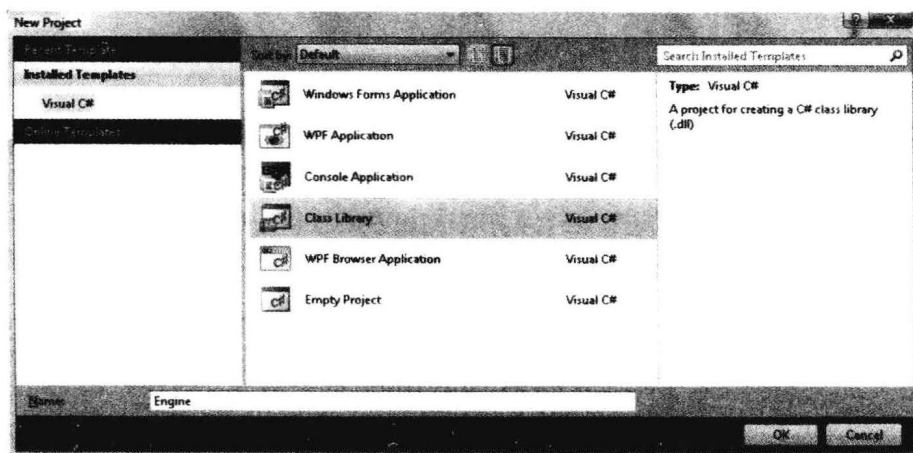


图 9-2 创建一个 Class Library 项目

如果使用的是 Visual Studio 2008,记得将生成类型修改为 x86,这一点很重要,因为 Devil 库现在只支持 32 位的项目。右击项目,在弹出的快捷菜单中选择 Properties 命令,然后单击 Build 选项卡。在 Configuration 下拉列表中,选择 All Configurations 选项,然后在 Platform Target 下拉列表中选择 x86。图 9-3 显示了这些设置。如果结果与图 9-3 不同,那么说明你已经针对 32 位进行了设置。在 Visual Studio 2010 中,这些设置应该被自动处理。但是如果确实需要在

Visual Studio 2010 中编辑生成的目标, 可以在 Solution Explorer 中右击解决方案, 然后在弹出的快捷菜单中选择 Properties Configuration 命令, 然后单击 Configuration Manager。在 Configuration Manager 中, 可以添加 x86 平台, 然后为引擎项目选中该设置。

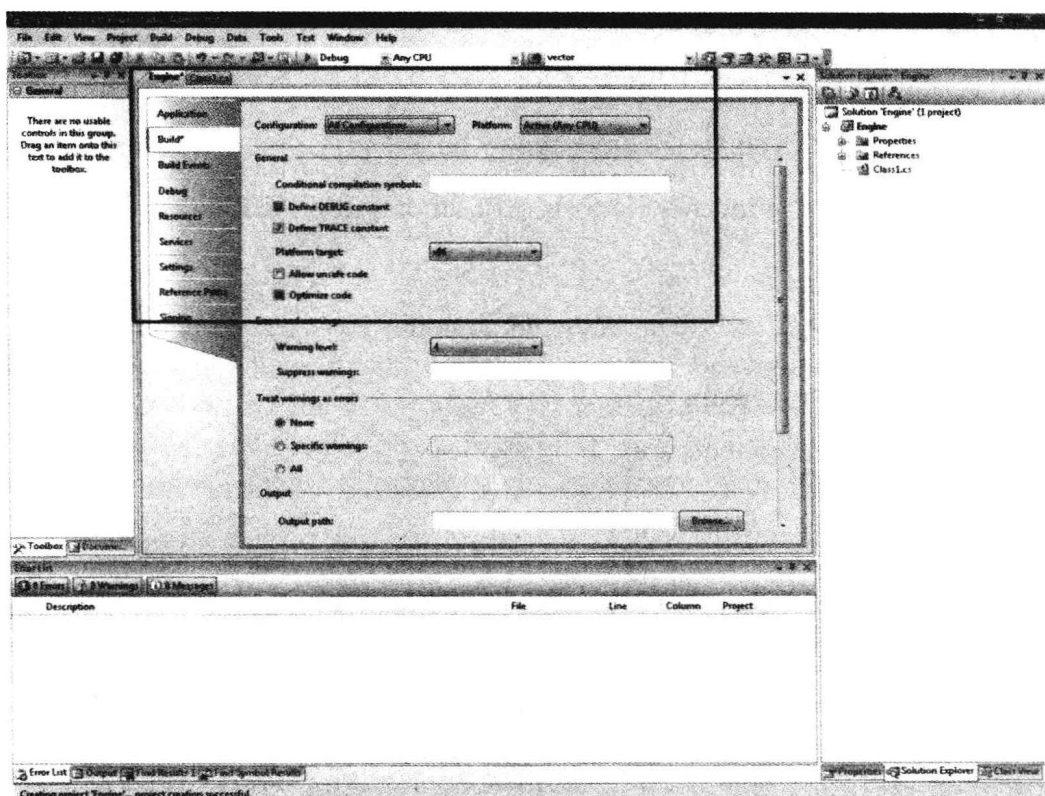


图 9-3 为 32 位项目进行设置

下一步是添加所有的类、测试和构成引擎的引用, 然后把新项目添加到源代码控制中。可以通过选择 Add Reference 对话框的 Browse 选项卡并手动找到 DLL 文件来添加 Tao 引用, 这些 DLL 文件应该位于 Tao 框架的安装目录中(在 Vista 和 Windows 7 上应该是在 C:\Program Files (x86)\TaoFramework\bin, 在 Windows XP 操作系统上应该是在 C:\Program Files\TaoFramework\bin)中。或者, 你也可能发现在 Add References 对话框的 Recent 选项卡中已经存在了这些库。

需要的引用包括:

- Tao framework OpenGL binding for .NET
- Tao framework Tao.DevIL binding for .NET
- Tao framework Windows Platform API binding for .NET
- nuint.framework
- System.Drawing
- System.Windows.Forms

核心的引擎组件包括:

- Batch

- CharacterData
- CharacterSprite
- Color
- FastLoop
- Font
- FontParser
- IGameObject
- Input
- Matrix
- Point
- PreciseTimer
- Renderer
- Sprite
- StateSystem
- Text
- Texture
- TextureManager
- Tween
- Vector

这些核心的引擎类都应该被设为 `public`。如果被设为 `private`、`protected` 或 `internal`，那么使用引擎库的任何游戏都无法访问它们。可能还需要把一些 `private`、`protected` 或 `internal` 函数设为 `public`，以便能够从库中访问它们。在遇到这样的函数时，将它们修改为 `public` 即可。

在开发游戏时，可以使用自己创建的实用代码来扩展引擎。引擎绝不应该包含特定游戏的操作代码，而应该足够通用，以便可以作为许多不同的游戏的基础。

9.2 扩展游戏引擎

我们目前创建的类已经涵盖了游戏引擎中需要具有的许多类。有一些遗漏之处：输入库可以改进得更好，还不支持处理多个纹理，也不支持声音。这些简单的修改可以使引擎更加完善。

你可能注意到了，在尝试运行游戏引擎项目时，会出现这个错误：A project with an Output Type of Class Library cannot be started directly。类库没有 `main` 函数，它们不会直接运行代码。相反，其他项目会使用类库中包含的代码。这意味着，如果想要测试引擎的某个部分，需要创建一个使用该引擎的新项目。

9.2.1 在项目中使用游戏引擎

需要创建一个使用引擎类的新项目。这个项目用于运行引擎代码和测试我们将做出的各种改进。

在 Visual Studio 中关闭引擎项目，然后选择 File|New|Project 命令来创建一个新项目。这一次选择 Windows Form Application 选项，而不是选择 Class Library 选项。这个项目将用于测试引擎库，所以将其命名为 EngineTest。单击 OK 按钮，生成一个新项目。

现在，这个项目需要加载引擎库。Visual Studio 使用两个术语：解决方案(Solution)和项目(Project)。每次启动一个新项目时，Visual Studio 都创建一个解决方案来包含该项目。解决方案的名称通常与主项目的名称相同，在这里，包含 EngineTest 项目的解决方案也叫做 EngineTest，如图 9-4 所示。



图 9-4 包含一个项目的解决方案

解决方案可以包含几个不同的项目。EngineTest 解决方案需要包含引擎项目，这样才能使用引擎代码。添加新项目是十分简单的。在 Solution Explorer 中右击 EngineTest 解决方案，弹出的快捷菜单如图 9-5 所示。选择 Add|Existing Project 命令，打开 Add Existing Project 对话框。

对话框将提供计算机上的项目目录的一个视图(见图 9-6)。其中一个子目录包含了引擎库。Visual Studio 使用目录结构显示解决方案及其项目。打开 Engine 文件夹(如果为游戏引擎使用了其他名称，则应找到相应的文件夹)。在该文件夹内，有另外一个叫做 Engine 的文件夹(第一个 Engine 文件夹是解决方案，第二个 Engine 文件夹是项目)。打开第二个 Engine 文件夹。在该文件夹内有一个文件 Engine.csproj，这就是代表 C#程序的文件。打开该文件。

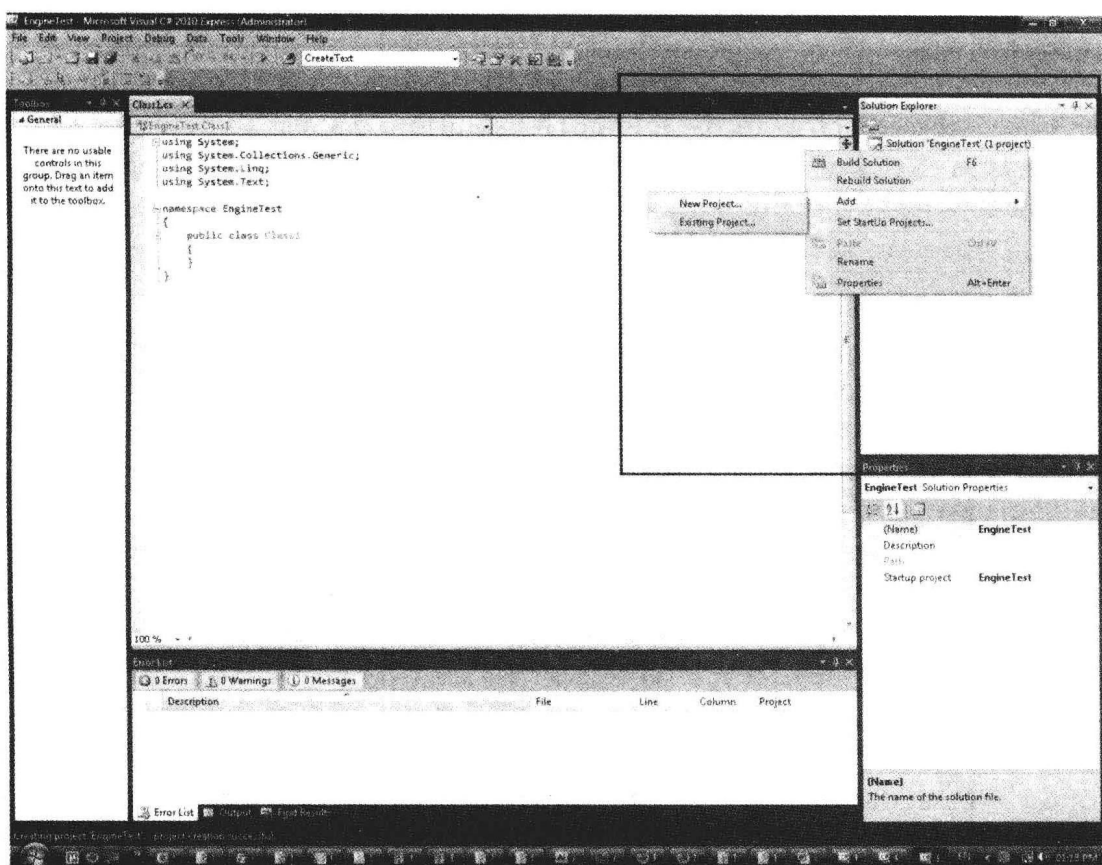


图 9-5 向解决方案中添加项目

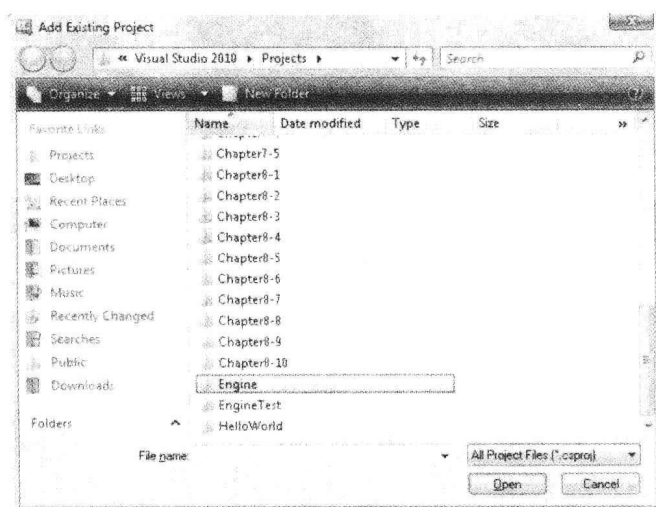


图 9-6 解决方案文件夹列表

Solution Explorer 现在有两个项目：类库 Engine 和使用引擎库的项目 EngineTest。引擎

库项目还没有被复制到 **EngineTest** 解决方案中，它只是一个引用，而它的全部代码仍然位于原来的位置。这样，使用引擎库处理多个项目变得更加简单，但是却不会使代码重复。引擎代码可以从任意解决方案中进行编辑，做出的所有更改将在使用该库的所有项目间共享。

注意，**EngineTest** 项目使用的字体为粗体。这是因为当运行解决方案时，**EngineTest** 项目将被执行。解决方案可能包含几个可以生成可执行文件的项目，所以 **Solution Explorer** 需要知道你想要运行和调试哪个项目。通过在 **Solution Explorer** 中右击 **Engine** 项目的名称，在弹出的快捷菜单中选择 **Set as Start Up Project** 命令，使其成为启动项目。**Engine** 项目的名称现在将变成粗体，**EngineTest** 项目的名称将由粗体变成普通字体。运行这个项目将得到一个错误，指出类库是不可以执行的。右击 **TestEngine** 项目，再次将其设为启动项目，这样就可以再次运行解决方案。如果还想为游戏开发关卡编辑器或其他工具，这种功能非常有用。可以把它作为使用引擎和游戏项目的额外的项目添加到解决方案中。

Engine 和 **TestEngine** 位于同一个解决方案中，但是现在它们还无法彼此访问。**TestEngine** 需要访问 **Engine** 中的代码，但是 **Engine** 却不需要知道 **TestEngine** 项目。在 **Solution Explorer** 中，展开 **EngineTest** 项目，项目下是一个 **References** 文件夹，如图 9-7 所示。

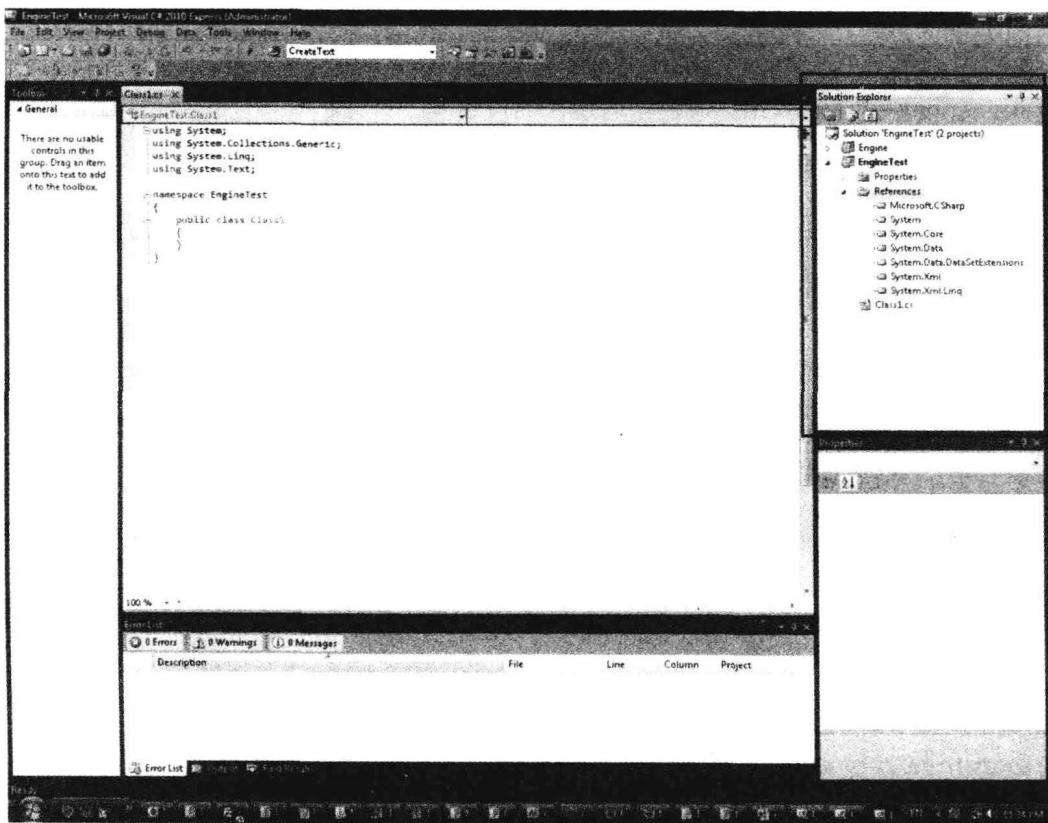


图 9-7 References 文件夹

右击 **Folder** 图标，在弹出的快捷菜单中选择 **Add References** 命令。打开的对话框正是我们之前用于添加 **JUnit** 和 **Tao** 库的引用的 **Add Reference** 对话框。该对话框的顶部包含多

个选项卡，默认情况下会选择.NET 选项卡。.NET 选项卡列出了对计算机上安装的.NET 库的引用。现在我们希望包含对 Engine 对象的引用。选择 Projects 选项卡，如图 9-8 所示。

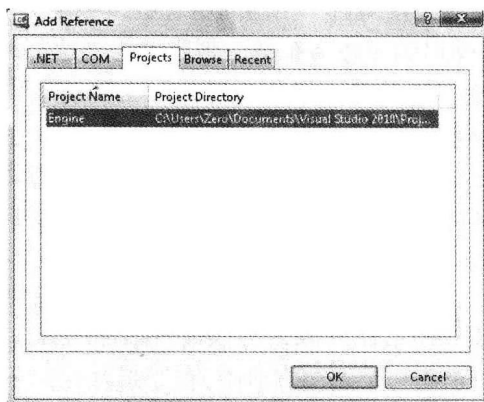


图 9-8 添加对项目的引用

Projects 选项卡中只有一个项目，选择它，然后单击 OK 按钮。现在，EngineTest 项目中有了对 Engine 项目的引用。它还需要添加下面的.NET 引用：System.Drawing、Tao.OpenGL、Tao.DevIL 和 Tao.Platform.Window。添加这些引用后，就可以把 SimpleOpenGL 控件添加到窗体上，并设置好 OpenGL。通过使用默认的设置类，可以把其中的一些引用移动到引擎库中。需要把 DevIL、ILU 和 ILUT 的 DLL 文件复制到 EngineTest 项目的 bin/debug/和 bin/release/目录中。

在 EngineTest 窗体上放置一个 SimpleOpenGL 控件，并将其 dock 属性设置为 fill，这与对之前的项目采取的操作相同。现在查看 form.cs 代码。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace EngineTest
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
}
```

这是 Visual Studio 为 Windows Form 项目默认创建的代码。顶部的 using 语句引用窗体使用的不同的库。现在有了一个 Engine 项目，可以添加一个新的 using 语句。

```
using Engine;
```

现在就可以访问引擎库中的全部类了。还需要编写普通的设置代码。下面是新的游戏项目的设置代码。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
using Tao.OpenGl;
using Tao.DevIl;
using Engine;

namespace EngineTest
{
    public partial class Form1 : Form
    {
        bool _fullscreen = false;
        FastLoop _fastLoop;
        StateSystem _system = new StateSystem();
        Input _input = new Input();
        TextureManager _textureManager = new TextureManager();

        public Form1()
        {
            InitializeComponent();
            simpleOpenGlControll1.InitializeContexts();

            InitializeDisplay();
            InitializeTextures();
            InitializeGameState();

            _fastLoop = new FastLoop(GameLoop);
        }
    }
}
```

```
private void InitializeGameState()
{
    // Load the game states here
}

private void InitializeTextures()
{
    // Init DevIL
    Il.ilInit();
    Ilu.iluInit();
    Ilut.ilutInit();
    Ilut.ilutRenderer(Ilut.ILUT_OPENGL);

    // Load textures here using the texture manager.
}

private void UpdateInput()
{
    System.Drawing.Point mousePos = Cursor.Position;
    mousePos = simpleOpenGLControl1.PointToClient(mousePos);

    // Now use our point definition,
    Engine.Point adjustedMousePoint = new Engine.Point();
    adjustedMousePoint.X = (float)mousePos.X - ((float)ClientSize.
        Width / 2);
    adjustedMousePoint.Y = ((float)ClientSize.Height / 2) - (float)
        mousePos.Y;
    _input.MousePosition = adjustedMousePoint;
}

private void GameLoop(double elapsedTime)
{
    UpdateInput();
    _system.Update(elapsedTime);
    _system.Render();
    simpleOpenGLControl1.Refresh();
}

private void InitializeDisplay()
{
    if (_fullscreen)
    {
        FormBorderStyle = FormBorderStyle.None;
    }
}
```

```

        WindowState = FormWindowState.Maximized;
    }
    else
    {
        ClientSize = new Size(1280, 720);
    }
    Setup2DGraphics(ClientSize.Width, ClientSize.Height);
}

protected override void OnClientSizeChanged(EventArgs e)
{
    base.OnClientSizeChanged(e);
    Gl.glViewport(0, 0, this.ClientSize.Width, this.ClientSize.
        Height);
    Setup2DGraphics(ClientSize.Width, ClientSize.Height);
}

private void Setup2DGraphics(double width, double height)
{
    double halfWidth = width / 2;
    double halfHeight = height / 2;
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    Gl.glOrtho(-halfWidth, halfWidth, -halfHeight, halfHeight,
        -100, 100);
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
}
}
}

```

这类设置代码可以用于启动任意数量的项目，所以有必要将它作为一个文件保存在某个位置，或者将它重构到引擎中。修改一下 `UpdateInput` 代码，使其与前面示例中的 `UpdateInput` 有所不同。代码中有两个 `Point` 类，一个来自于 `System.Drawing`，另一个来自于我们的 `Engine` 库。为了使编译器知道我们想要使用哪个 `Point` 类，需要使用完全限定的名称，例如：

```

// Using the fully qualified name removes potential confusion.
System.Drawing.Point point = new System.Drawing.Point();
Engine.Point point = new Engine.Point();

```

运行这个测试程序将产生一个空白的屏幕，这意味着代码运行正常。现在可以使用这个项目来运行需要的任何引擎测试。还可以根据需要添加额外的游戏状态。现在，引擎还无法很好地支持多个纹理，所以这就是我们接下来要面临的挑战。

9.2.2 多个纹理

引擎库现在无法很好地处理具有不同纹理的精灵。为了演示它如何处理不同的纹理，创建一个新的游戏状态 `MultipleTexturesState`。和前面的状态一样，它应该实现 `IGameObject`，并通过 `StateSystem` 加载。这个状态将测试纹理系统，因此，其构造函数中还需要接受一个 `TextureManager` 对象。加载代码可以放在 `form.cs` 的 `InitializeGameState` 中。

```
private void InitializeGameState()
{
    // Load the game states here
    _system.AddState("texture_test", new MultipleTexturesState
(_textureManager));

    _system.ChangeState("texture_test");
}
```

这个新状态需要创建具有不同纹理的精灵。本书配套光盘的 `Assets` 目录中提供了两个新的纹理：`spaceship.tga` 和 `spaceship2.tga`。这是两个不同的飞船的纹理，它们很大，但是细节度很低。将这两个 `.tga` 文件添加到解决方案中，并设置其属性，以便将它们复制到所构建的目录中。

需要把新纹理加载到 `TextureManager` 中，这是在 `InitializeTextures` 方法中完成的。

```
// Load textures here using the texture manager.
_textureManager.LoadTexture("spaceship", "spaceship.tga");
_textureManager.LoadTexture("spaceship2", "spaceship2.tga");
```

现在已经加载了纹理，可以在 `MultipleTexturesState` 中使用它们了。

```
class MultipleTexturesState : IGameObject
{
    Sprite _spaceship1 = new Sprite();
    Sprite _spaceship2 = new Sprite();
    Renderer _renderer = new Renderer();

    public MultipleTexturesState(TextureManager textureManager)
    {
        _spaceship1.Texture = textureManager.Get("spaceship");
        _spaceship2.Texture = textureManager.Get("spaceship2");

        // Move the first spaceship, so they're not overlapping.
        _spaceship1.SetPosition(-300, 0);
    }

    public void Update(double elapsedTime) {}
}
```

```
public void Render()
{
    _renderer.DrawSprite(_spaceship1);
    _renderer.DrawSprite(_spaceship2);
    _renderer.Render();
}
```

这个状态创建了两个精灵，分别对应于每个飞船纹理。第一个飞船在 X 轴上向后移动了 300 像素，这可以避免飞船相互重叠。运行这个状态后，看到的结果如图 9-9 所示。

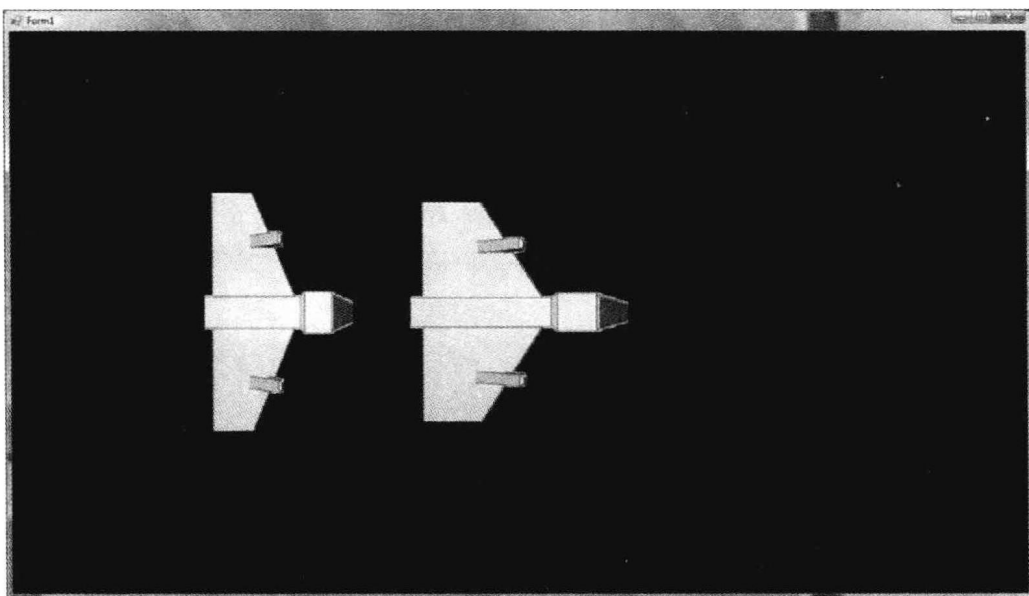


图 9-9 错误的纹理

在图 9-9 中，第二个飞船绘制正确，但是第一个飞船的纹理却是错误的。第一个飞船的精灵的大小是其纹理的大小，所以与第二个飞船相比，看上去被挤压变小了，而第二个飞船既具有正确的尺寸，又具有正确的纹理。

这里的问题是，没有告诉 OpenGL 什么时候纹理改变了。如果它不知道，就会使用上一次设置的纹理。在游戏循环中，如果每帧中大量改变纹理，游戏的速度会降低，但是适量的纹理更改不是需要担心的问题，实际上对于大多数游戏，肯定会发生这种更改。

Renderer 类需要修改。**Renderer** 类将所有的精灵组织成一个批次，以便可以将它们同时发送到图形卡。现在，**Render** 类不检查精灵使用了哪一个纹理。它需要进行检查以后才知道是否使用新纹理绘制精灵。如果是，就可以告诉 OpenGL 使用旧纹理绘制现在批次中具有的精灵，然后使用新纹理开始一个新批次。

当前的 **Renderer.DrawSprite** 代码如下。

```
public void DrawSprite(Sprite sprite)
```

```
{  
    _batch.AddSprite(sprite);  
}
```

新的逻辑代码如下形式。

```
int _currentTextureId = -1;  
public void DrawSprite(Sprite sprite)  
{  
    if (sprite.Texture.Id == _currentTextureId)  
    {  
        _batch.AddSprite(sprite);  
    }  
    else  
    {  
        _batch.Draw(); // Draw all with current texture  
  
        // Update texture info  
        _currentTextureId = sprite.Texture.Id;  
        Gl.glBindTexture(Gl.GL_TEXTURE_2D, _currentTextureId);  
        _batch.AddSprite(sprite);  
    }  
}
```

运行项目，得到的结果如图 9-10 所示。

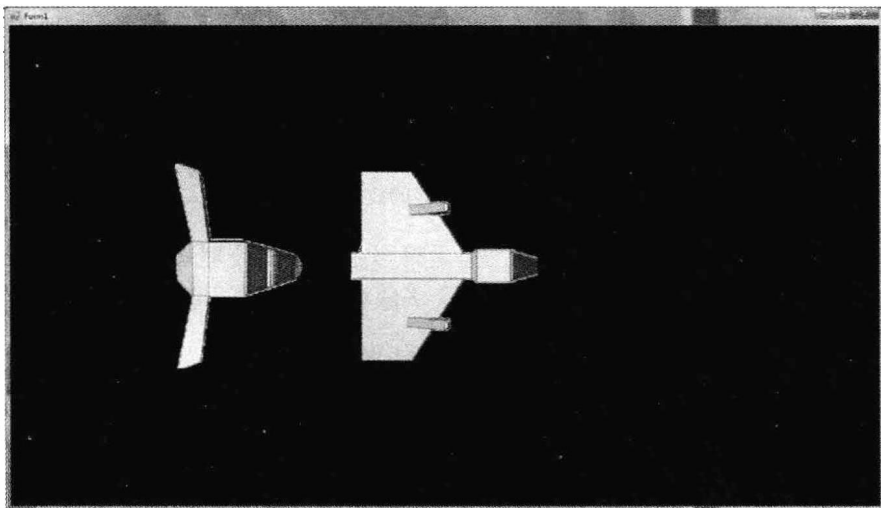


图 9-10 正确的纹理

与之前的代码相比，这段代码稍微有点复杂，但是它使引擎可以支持具有不同纹理的精灵。考虑运行这段代码的最坏情形：绘制的每个精灵都强制要求改变纹理。避免出现这种情形的方法是，选出被绘制的精灵，并确保一起发送最大数量的使用相同纹理的顶点。还有一种方法可以使纹理更改发生的次数变得最小，即不使用大量单独的小纹理，而是将

纹理组合到一起, 形成一个较大的纹理, 这常被称为纹理集合(atlas)。然后精灵可以引用这个较大的纹理, 但是修改其顶点的(U , V)坐标, 使它们只能使用纹理的一个小部分(这就是字体渲染的工作原理)。一般来说, 如果不是问题, 就没有必要修复。

9.3 添加声音支持

声音很容易被忽略, 但是它可以改变玩游戏时的感觉。使用 OpenAL 库, 添加声音也是非常简单的。在了解细节问题之前, 借助一段简单的代码来了解一下一个简单的声音库接口的工作原理。

```
SoundManager soundManager = new SoundManager();
soundManager.Add("zap", "zap.wav");
Sound zapSound = soundManager.Play("zap")
if (soundManager.IsSoundPlaying("zap"))
{
    soundManager.StopPlaying(zapSound);
}
```

这段代码表明, 声音库应该采用与纹理管理器类似的方式管理声音。当播放声音时, 将返回一个引用, 该引用可以用来控制声音。使用 `IsSoundPlaying` 方法可以检查是否在播放声音, 使用 `StopPlaying` 方法可以停止播放声音。如果有循环声音和音量控制就更好了。

9.3.1 创建声音文件

为了测试新的声音代码, 还需要有一些声音。如果粗略地进行分类, 游戏中有两类声音: 类似于枪声的声音效果和背景音乐或环境音。

生成声音效果的一个好方法是使用 Tomas Pettersson 开发的 `sfxr` 程序, 本书的配套光盘中提供了该程序, 其界面如图 9-11 所示。

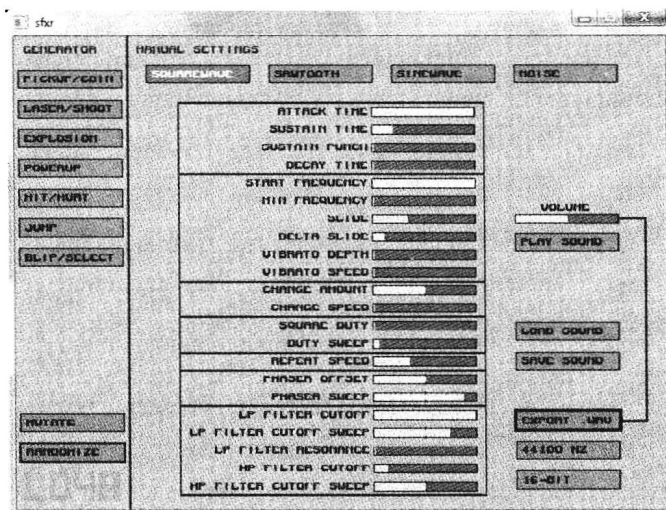


图 9-11 使用 `sfxr` 创建声音效果

sfxr 为游戏随机生成声音效果。声音效果听起来与 NES 控制台生成的声音有些类似。不断地单击 **Randomize** 按钮,生成新的声音。找到自己喜欢的声音后,可以把它导出为一个.wav 文件。本书配套光盘的 **Assets** 目录中包含了两个生成的声音效果,分别是 **soundeffect1.wav** 和 **soundeffect2.wav**。按照添加纹理的方式,把它们添加到 **TestEngine** 项目中。sfxr 声音非常适合作为怀旧游戏的声音效果或者暂用的声音效果。

9.3.2 开发 SoundManager

OpenAL 是许多现代游戏使用的一个专业级的库,它广泛支持更高级的声音效果,例如从一个 3D 位置播放声音。它还提供了生成多普勒效应(Doppler Effect)的函数。多普勒效应的一个常见的示例是警车或者救护车从身旁开过:在车辆开过时,警报器的频率也会改变。因为我们将开发的是一个 2D 游戏,所以不会使用这种功能,但是它在有些场合是非常有用的。仔细阅读 OpenAL 的文档,然后自己进行试验,以最大程度地利用好该库。

为了使用 OpenAL,需要在引擎库项目中添加对它的引用。引用的名称是 **Tao Framework Tao.OpenAl Binding for .NET**。如果 **Add Reference** 对话框的.NET 选项卡下没有列出这个引用,则选择 **Browse** 选项卡,找到 Tao 框架的安装目录\TaoFramework\bin,然后从那里选择该引用。为使用 OpenAL,还需要把 **alut.dll**、**ILU.dll** 和 **OpenAL32.dll** 复制到 **bin/debug** 和 **bin/release** 目录中。在 Tao Framework 的安装目录\TaoFramework\lib 中可以找到这些 DLL 文件。根据简单的理想化的 API 和 **TextureManager** 类的工作方式,可以创建一个框架类。首先需要创建声音类,它代表播放的声音。

```
public class Sound
{
}
```

然后可以使用这个声音类来构建声音管理器框架类。记住,应该把这些类添加到引擎项目中,因为许多游戏都可以使用这些代码。

```
public class SoundManager
{
    public SoundManager()
    {
    }

    public void LoadSound(string soundId, string path)
    {
    }

    public Sound PlaySound(string soundId)
    {
        return null;
    }
}
```

```
public bool IsSoundPlaying(Sound sound)
{
    return false;
}

public void StopSound(Sound sound)
{
}

}
```

声音硬件只能同时播放有限数量的声音。可以播放的声音的数量叫做声道数。现代的声音硬件一般可以同时播放 256 种声音。OpenAL 找出可用声道的方法，不断地请求声道，当硬件不能提供新声道时，就得到了最大声道数。

```
readonly int MaxSoundChannels = 256;
List<int> _soundChannels = new List<int>();

public SoundManager()
{
    Alut.alutInit();
    DiscoverSoundChannels();
}

private void DiscoverSoundChannels()
{
    while (_soundChannels.Count < MaxSoundChannels)
    {
        int src;
        Al.alGenSources(1, out src);
        if (Al.alGetError() == Al.AL_NO_ERROR)
        {
            _soundChannels.Add(src);
        }
        else
        {
            break; // there's been an error - we've filled all the channels.
        }
    }
}
```

每个声道都用一个数字标识，所以可用的声道可以被存储为一个整数列表。SoundManager 构造函数初始 OpenAL，然后找到所有可用的声道，这样的声道最多有 256 个。OpenAL 函数 alGenSources 生成一个声源，用于保存声道。然后代码检查是否存在错误，如果存在错误，则意味着 OpenAL 无法生成更多的声源，此时就应该跳出循环。

发现声源后，需要把它们添加到 **EngineTest** 项目中。应该在 **form.cs** 类中创建声音管理器，然后把它传递给任何想要使用它的状态。

```
public partial class Form1 : Form
{
    bool _fullscreen = false;
    FastLoop _fastLoop;
    StateSystem _system = new StateSystem();
    Input _input = new Input();
    TextureManager _textureManager = new TextureManager();
    SoundManager _soundManager = new SoundManager();
}
```

最后一行创建了声音管理器对象，它将找出可用的声道数。下一步是从硬盘加载文件到内存中。需要创建一个新结构来保存从硬盘加载的声音文件的数据。

```
public class SoundManager
{
    struct SoundSource
    {
        public SoundSource(int bufferId, string filePath)
        {
            _bufferId = bufferId;
            _filePath = filePath;
        }
        public int _bufferId;
        string _filePath;
    }
    Dictionary<string, SoundSource> _soundIdentifier = new
        Dictionary<string, SoundSource>();
}
```

SoundSource 结构存储关于加载的声音的信息。它只是在声音管理器的内部使用，所以在 **SoundManager** 类的内部定义该结构。当 **OpenAL** 加载声音数据后，将返回一个整数，需要播放声音时，就使用这个整数来引用声音。声音标识符将声音的名称映射到内存中的声音数据。字典是一个很大的表，其中列出了游戏可用的所有声音，以及用于标识声音的简单的英文字符串。

为了加载声音文件，需要把语句 **using System.IO**; 添加到文件的顶部。**LoadSound** 函数将使用游戏中用到的声音填充 **_soundIdentifier** 字典。

```
public void LoadSound(string soundId, string path)
{
    // Generate a buffer.
    int buffer = -1;
    Al.alGenBuffers(1, out buffer);
}
```

```

int errorCode = Al.alGetError();
System.Diagnostics.Debug.Assert(errorCode == Al.AL_NO_ERROR);

int format;
float frequency;
int size;
System.Diagnostics.Debug.Assert(File.Exists(path));
IntPtr data = Alut.alutLoadMemoryFromFile(path, out format, out size,
    out frequency);
System.Diagnostics.Debug.Assert(data != IntPtr.Zero);
// Load wav data into the generated buffer.
Al.alBufferData(buffer, format, data, size, (int)frequency);
// Everything seems ok, add it to the library.
_soundIdentifier.Add(soundId, new SoundSource(buffer, path));
}

```

LoadSound 方法首先生成一个缓冲区。这个缓冲区是内存中的一个区域，存储了从硬盘上加载的声音数据。**OpenAL** 实用函数 **alutLoadMemoryFromFile** 用于将.wav 文件的数据读入内存。然后，包含文件数据的内存将与同时读入的格式、大小和频率数据一起放到缓冲区中。最后，使用 **SoundId** 将这个缓冲区保存到字典中，以便在以后标识它。

现在就可以使用声音管理器来加载添加到项目中的声音了。

```

public Form1()
{
    InitializeComponent();
    simpleOpenGLControl1.InitializeContexts();

    InitializeDisplay();
    InitializeSounds(); // Added
    InitializeTextures();
    InitializeGameState();
}

```

InitializeSound 方法将负责加载所有的声音文件。

```

private void InitializeSounds()
{
    _soundManager.LoadSound("effect", "soundEffect1.wav");
    _soundManager.LoadSound("effect2", "soundEffect2.wav");
}

```

为了测试声音管理器，我们需要一些播放声音的代码。当 **OpenAL** 播放声音时，会返回一个用于引用所播放声音的整数。我们的声音管理器将把这个整数包装到一个 **Sound** 类中，这样在代码中就可以看出它的来源。现在它并不是一个随机的整数，而是一个有类型的声音对象。如果播放声音时发生错误，将返回-1。如下所示是包装了 **OpenAL** 的引用整

数的 Sound 类。

```
public class Sound
{
    public int Channel { get; set; }

    public bool FailedToPlay
    {
        get
        {
            // minus is an error state.
            return (Channel == -1);
        }
    }
    public Sound(int channel)
    {
        Channel = channel;
    }
}
```

现在已经定义了声音包装器类，可以添加 **PlaySound** 方法了。声音必须在一个声道上播放。声道的数量是有限的，所以所有的通道都被占用的情况是有可能发生的。如果是这种情况，声音将不会被播放。这是一个很简单的探索过程，但是对于同时使用几百种声音的游戏，最好给每种声音分配一个优先级，这样具有高优先级的声音就可以占用具有低优先级的声音所使用的声道。优先级可能与声音的音量和距离玩家的远近程度有关。

需要编写一个方法来确定指定的声道是否被占用。一种简单的方法是询问 **OpenAL** 当前是否在该声道上播放声音。如果没有，那么该声道未被占用。应该把这个新添加的 **IsChannelPlaying** 方法添加到 **SoundManager** 类中。

```
private bool IsChannelPlaying(int channel)
{
    int value = 0;
    Al.alGetSourcei(channel, Al.AL_SOURCE_STATE, out value);
    return (value == Al.AL_PLAYING);
}
```

OpenAL 函数 **alGetSourcei** 查询特定声道的属性，该属性由第二个参数确定。在这里我们询问声源当前的状态。声音在这个声道上的音量大小和播放速度也可以通过这种方式查询。**IsChannelPlaying** 函数检查声道的当前状态是否被设为正在播放，如果是，它返回 **true**，否则返回 **false**。

定义了 **IsChannelPlaying** 函数后，可以使用它来构建另外一个返回空闲声道的函数。将这个新函数命名为 **GetNextFreeChannel**，它将迭代函数 **DiscoverSoundChannels** 在构造函数中确定的各个声道。如果不能找到空闲的声道，它将返回错误标识-1。

```
private int FindNextFreeChannel()
{
    foreach (int slot in _soundChannels)
    {
        if (!IsChannelPlaying(slot))
        {
            return slot;
        }
    }

    return -1;
}
```

这个函数将寻找可以播放声音的空闲声道。这样就可以更加轻松地编写一个健壮的 **PlaySound** 方法，使其能够处理有限数量的声道。

```
public Sound PlaySound(string soundId)
{
    // Default play sound doesn't loop.
    return PlaySound(soundId, false);
}

public Sound PlaySound(string soundId, bool loop)
{
    int channel = FindNextFreeChannel();
    if (channel != -1)
    {
        Al.alSourceStop(channel);
        Al.alSourcei(channel, Al.AL_BUFFER, _soundIdentifier[soundId].
            _bufferId);
        Al.alSourcef(channel, Al.AL_PITCH, 1.0f);
        Al.alSourcef(channel, Al.AL_GAIN, 1.0f);

        if (loop)
        {
            Al.alSourcei(channel, Al.AL_LOOPING, 1);
        }
        else
        {
            Al.alSourcei(channel, Al.AL_LOOPING, 0);
        }
        Al.alSourcePlay(channel);
        return new Sound(channel);
    }
    else
```

```
{
    // Error sound
    return new Sound(-1);
}
}
```

这里实现了两个 **PlaySound** 方法，一个方法接受指定是否需要循环播放声音的标志作为参数，另一个方法接受的参数要少一个，它假定默认情况下只应该播放声音一次。**PlaySound** 方法找到空闲的声道，并将缓冲的数据从 **SoundSource** 加载到声道上。然后它重设声道的默认属性，包括音高和增益，并设置循环标志。循环标志决定声音在播放一次后是结束播放还是重复播放。最后，方法播放声音并返回一个 **Sound** 对象。

现在声音系统可以很好地工作，能够对其进行测试了。它可以从硬盘上加载文件，并根据需要播放它们。下面是一个新的测试状态，演示了声音播放功能。

```
class SoundTestState : IGameObject
{
    SoundManager _soundManager;
    double _count = 3;

    public SoundTestState(SoundManager soundManager)
    {
        _soundManager = soundManager;
    }

    public void Render()
    {
        // The sound test doesn't need to render anything.
    }

    public void Update(double elapsedTime)
    {
        _count -= elapsedTime;
        if (_count < 0)
        {
            _count = 3;
            _soundManager.PlaySound("effect");
        }
    }
}
```

按照普通的方式加载状态，并确保它是默认运行的状态。每 3 秒钟，它会播放第一种声音效果。可以尝试自由修改这个时间，或者同时播放两种声音效果。

下面的代码段同时播放两种声音，这表明 **SoundManager** 正确地使用了硬件声道。


```
public void Update(double elapsedTime)
{
    _count -= elapsedTime;

    if (_count < 0)
    {
        _count = 3;
        _soundManager.PlaySound("effect");
        _soundManager.PlaySound("effect2");
    }
}
```

SoundManager 类还需要最后的几个函数来变得更加完善。它需要能够测试是否在播放声音。还需要能够停止播放声音。音量控制也将十分有用。

```
public bool IsSoundPlaying(Sound sound)
{
    return IsChannelPlaying(sound.Channel);
}

public void StopSound(Sound sound)
{
    if (sound.Channel == -1)
    {
        return;
    }
    Al.alSourceStop(sound.Channel);
}
```

检查是否在播放声音的代码重用了检查声音是否在特定声道上播放的代码。停止播放声音的函数使用一个 **OpenAL** 方法来停止特定声道上的声音。同样，可以在声音状态中测试这些新函数。

```
public void Update(double elapsedTime)
{
    _count -= elapsedTime;

    if (_count < 0)
    {
        _count = 3;
        Sound soundOne = _soundManager.PlaySound("effect");
        Sound soundTwo = _soundManager.PlaySound("effect2");

        if (_soundManager.IsSoundPlaying(soundOne))
        {
```

```

        _soundManager.StopSound(soundOne);
    }
}

```

这里先播放第一种声音，然后立即检查声音是否正在播放。该方法返回 **true**，然后另一个调用会停止声音。这种声音永远都不会被听到，因为在游戏循环结束时，它总是被停止。

```

float _masterVolume = 1.0f;
public void MasterVolume(float value)
{
    _masterVolume = value;
    foreach (int channel in _soundChannels)
    {
        Al.alSourcef(channel, Al.AL_GAIN, value);
    }
}

```

在 **OpenAL** 中，可以通过指定声道上的增益来修改音量。增益的取值范围为 0~1。这里为每个声道设置了一个主音量。音量还存储在一个类变量中。当播放新声音时，它将重写声道当前的增益设置，所以需要重新应用增益设置。这可以在 **PlaySound** 方法的底部完成。

```

{
    Al.alSourcef(channel, Al.AL_GAIN, _masterVolume);
    Al.alSourcePlay(channel);
    return new Sound(channel);
}

```

这就为所有的声道设置了一个主音量控制。也可以逐个声道设置音量，当使音乐淡出或者变得稍微可以听见时，这种方法比较有用。

```

public void ChangeVolume(Sound sound, float value)
{
    Al.alSourcef(sound.Channel, Al.AL_GAIN, _masterVolume * value);
}

```

这里将特定声道的音量与主音量相乘，以确保当玩家将主音量设置较低时，新声音不会突然变得很大。值应该在 0~1 之间，下面又列出了一些测试代码，以显示音量的改变。

```

public SoundTestState(SoundManager soundManager)
{
    _soundManager = soundManager;
    _soundManager.MasterVolume(0.1f);
}

```

这会将音量设置为前一个值的 1/10。如果再次运行测试状态，可以立即注意到产生的

差别。声音管理器的最后一个任务是确保它正确关闭。声音管理器创建了大量对声音文件和数据的引用，当它被销毁时需要释放这些引用。完成这种操作的最佳方式是实现 `IDisposable` 接口。

```
public class SoundManager : IDisposable
{
    public void Dispose()
    {
        foreach (SoundSource soundSource in _soundIdentifier.Values)
        {
            SoundSource temp = soundSource;
            Al.alDeleteBuffers(1, ref temp._bufferId);
        }
        _soundIdentifier.Clear();
        foreach (int slot in _soundChannels)
        {
            int target = _soundChannels[slot];
            Al.alDeleteSources(1, ref target);
        }
        Alut.alutExit();
    }
}
```

这个函数遍历所有的声音缓冲区并释放它们，然后遍历声卡的所有声道也释放它们。

9.4 改进输入

目前的引擎只对输入提供了有限的支持。可以查询输入类来找出鼠标光标的位置，但是除此以外没有其他支持。街机游戏的理想输入是游戏手柄，这是一种专门用来玩游戏的硬件。第一人称射击游戏、策略游戏和模拟游戏一般更适合结合使用鼠标和键盘。因此，引擎应该更完善地支持鼠标，而且应该能够查询键盘的状态。

9.4.1 包装游戏控制器

最适合街机游戏的控制器是游戏手柄。PC 似乎总是跟不上控制台控制器的脚步，但是近来控制台控制器已经开始使用 USB 连接，所以在 PC 上添加对控制台控制器的支持也变得十分简单。在玩家群中，这些控制器也十分流行，因为它们的质量高且得到了广泛的支持，从市面上找到它们也很容易。

`TaoFramework` 中对控制器的支持还可以，但不够出色。它支持摇杆和模拟(analog)按键，但是不支持诸如振动等触感效果，更不用说更加奇特的 `Wiimote` 了。对于这类较新的控制器，也存在外部 C#库，但是要使用它们的话，需要对其进行一些研究。

我们这个引擎中对控制器的支持仅限于更加标准的控制器。对于 PC 玩家，最流行的控制器可能是 `Xbox 360` 控制器，因为这种控制器可以被 `Windows` 直接识别，并且有许多按键和摇杆。游戏手柄的代码将足够灵活，以允许使用任何标准的手柄，但是为了测试的

目的, 代码将针对 Xbox 360 手柄进行开发。图 9-12 显示了 Xbox 360 的游戏手柄。

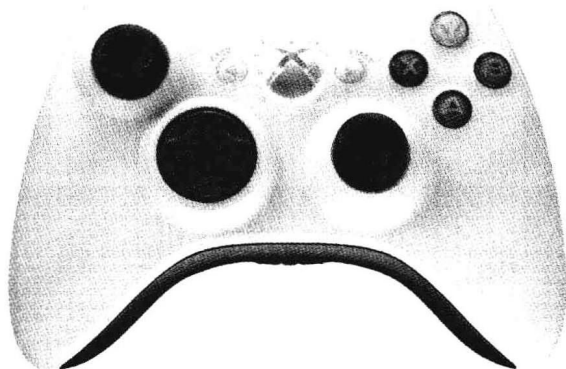


图 9-12 Xbox 360 游戏控制器

Xbox 360 有一些不同类型的控制器部件。它的上部包含两个摇杆。我们希望在 X 轴和 Y 轴上获得-1~1 之间的值, 以确定玩家向哪里移动摇杆。D-pad 支持基本的上、下、左、右命令。控制器上部的其余控制器部件是一些简单的按键: Back、Start、A、B、X 和 Y。它们只有两个状态, 即按下和未被按下, 所以非常简单, 使用一个布尔变量表示其状态即可。在控制器前方还有 4 个肩部(shoulder)按键, 靠近上部的两个按键是简单的按键, 但是后面的两个按键却是触发器。触发器很特殊, 因为它们不是数字开关按键, 而是具有一个取值范围 0~1, 值的大小与按下的程度相应。所有这些控制器部件一起组成了一个非常复杂的游戏手柄, 需要创建多个类来描述每类控制器部件的不同功能。

首先, 创建一个新的游戏测试状态。将这个状态命名为 **InputTestState**, 并像前面一样使其成为默认加载的状态。游戏手柄包装在 Tao 框架的 SDL 部分中, 这意味着需要添加另外一个引用: Tao framework SDL Binding for .NET(如果没有这个引用, 则需要选择 Browse 选项卡, 找到 TaoFramework\lib 目录, 选择 Tao.Sdl.dll)。还需要把另外一个 DLL 复制到 bin 目录中: sdl.dll。我们将逐个构建对各个控制器部件的支持, 最终完成对整个控制器的支持。如果使用的不是 Xbox 360 控制器, 也可以按照本节介绍来构建对其他控制器的支持。只需在必要的地方替换控制器部件即可。

控制器是玩家与游戏交互的主要方式。因此, 总是需要知道控制器的当前状态。每一帧中都将包含代码来询问控制器其各个控制器部件的状态, 然后我们更新内存中的表示。不断地查询设备有时候也叫做轮询(polling)。

SDL 库要求首先初始化其手柄子系统, 然后才能使用手柄。现在将把初始化代码放到 **InputTestState** 中, 但是最终将需要把这些代码移动到输入类中。

```
bool _useJoystick = false;
public InputTestState()
{
    Sdl.SDL_InitSubSystem(Sdl.SDL_INIT_JOYSTICK);
    if (Sdl.SDL_NumJoysticks() > 0)
    {
```

```

        // Start using the joystick code
        _useJoystick = true;
    }
}

```

设置代码是很容易被理解的。首先初始化 SDL 的摇杆子系统，如果存在摇杆，就可以开始使用它了。接下来，需要在引擎库项目中创建一个类来代表控制器。因为我们将表示一种非常具体的控制器，所以将这个类命名为 `XboxController`。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Tao.Sdl;

namespace Engine
{
    public class XboxController : IDisposable
    {
        IntPtr _joystick;
        public XboxController(int player)
        {
            _joystick = Sdl.SDL_JoystickOpen(player);
        }

        #region IDisposable Members

        public void Dispose()
        {
            Sdl.SDL_JoystickClose(_joystick);
        }

        #endregion
    }
}

```

这段代码使用一个玩家索引创建摇杆。当要创建一个支持两个或更多个玩家的游戏时，需要为每个玩家创建一个控制器。摇杆也是可以丢弃的，它在被销毁后将释放对控制器的引用。

现在可以在测试状态中创建控制器。

```

XboxController _controller;
public InputTestState()
{

```

```

    Sdl.SDL_InitSubSystem(Sdl.SDL_INIT_JOYSTICK);
    if (Sdl.SDL_NumJoysticks() > 0)
    {
        // Start using the joystick code
        _useJoystick = true;
        _controller = new XboxController(0);
    }
}

```

我们要包装的第一类控制器部件是摇杆。摇杆非常适合移动角色和定位摄像机。制作的摇杆仍有可以改进之处。有时候控制器平放在桌子上，摇杆的位置在正中央，但是摇杆仍会报告它们被摇动。这个问题的解决办法是，在摇杆被摇动到一定阈值之前，忽略其输出。忽略的部分叫做死区(dead zone)，不同控制器的死区是不同的(因此，在指定死区时，可以把值设置得稍大一些)。

摇杆被认为有两个轴：从左到右的 X 轴和从上到下的 Y 轴。SDL 将轴信息作为一个 short 值返回，但更方便的表示方法是 -1~1 的 float 值。-1 是摇杆被摇动到最左边(或最下边)时的取值，1 是摇杆被摇动到最右边(或最上边)时的取值。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Tao.Sdl;

namespace Engine
{
    public class ControlStick
    {
        IntPtr _joystick;
        int _axisIdX = 0;
        int _axisIdY = 0;
        float _deadZone = 0.2f;

        public float X { get; private set; }
        public float Y { get; private set; }

        public ControlStick(IntPtr joystick, int axisIdX, int axisIdY)
        {
            _joystick = joystick;
            _axisIdX = axisIdX;
            _axisIdY = axisIdY;
        }
    }
}

```

```
public void Update()
{
    X = MapMinusOneToOne(Sdl.SDL_JoystickGetAxis(_joystick,
        _axisIdX));
    Y = MapMinusOneToOne(Sdl.SDL_JoystickGetAxis(_joystick,
        _axisIdY));
}

private float MapMinusOneToOne(short value)
{
    float output = ((float)value / short.MaxValue);

    // Be careful of rounding error
    output = Math.Min(output, 1.0f);
    output = Math.Max(output, -1.0f);

    if (Math.Abs(output) < _deadZone)
    {
        output = 0;
    }

    return output;
}
}
```

SDL 表示的模拟控制器部件使用的轴可以通过 `SDL_JoystickGetAxis` 轮询。摇杆由两个轴组成。控制器可能会使用多个不同的轴，所以在构造函数中传入了两个索引来标识想这个摇杆代表的轴。这些标识数字将随控制器的类型不同而发生改变。表示手柄上不同控制器部件的数字并不保证对于每类手柄都是相同的。一种手柄的左摇杆的索引可能为 1，但是另外一种手柄可能使用 5 来索引左摇杆。因此，允许玩家重新映射他的控制器部件通常是一个好主意。

每一帧中都将调用一次 `Update` 方法，根据摇杆的位置，该方法使用 -1~1 之间的值更新 `X` 和 `Y` 的值。此外，还有一个用于死区的小缓冲区，以忽略控制台的轻微移动。

Xbox 控制器有两个摇杆，所以现在可以更新控制器类以反映这一点。

```
public ControlStick LeftControlStick { get; private set; }
public ControlStick RightControlStick { get; private set; }
public XboxController(int player)
{
    _joystick = Sdl.SDL_JoystickOpen(player);
    LeftControlStick = new ControlStick(_joystick, 0, 1);
    RightControlStick = new ControlStick(_joystick, 4, 3);
}
```

```
    }

    public void Update()
    {
        LeftControlStick.Update();
        RightControlStick.Update();
    }
```

控制器上有一些控制器部件，在测试状态时可以使用它们来四处移动游戏中的对象。**InputTestState** 类的 **Update** 函数会更新 SDL 摇杆系统，然后更新控制器，从而更新控制器上的全部控制器部件的值。

```
    public void Update(double elapsedTime)
    {
        if (_useJoystick == false)
        {
            return;
        }

        Sdl.SDL_JoystickUpdate();
        _controller.Update();
    }

    public void Render()
    {
        if (_useJoystick == false)
        {
            return;
        }

        Gl.glDisable(Gl.GL_TEXTURE_2D);
        Gl.glClearColor(1, 1, 1, 0);

        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        Gl.glPointSize(10.0f);
        Gl.glBegin(Gl.GL_POINTS);
        {
            Gl.glColor3f(1, 0, 0);
            Gl.glVertex2f(
                _controller.LeftControlStick.X * 300,
                _controller.LeftControlStick.Y * -300);
            Gl.glColor3f(0, 1, 0);
            Gl.glVertex2f(
                _controller.RightControlStick.X * 300,
                _controller.RightControlStick.Y * -300);
        }
```



```
    }  
    Gl.glEnd();  
}
```

Render 函数将绘制一个白色的背景，然后绘制一个绿点和一个红点来表示每个摇杆。点的大小被加大了，而且纹理模式也被禁用，以便我们更容易看到这些点。运行程序，然后在屏幕上移动点。摇杆的值只能在-1~1之间。这个值不够大，无法明显地在屏幕上移动这个点。因此，我们将这个值乘以 300。Y轴的值被乘以-300，以便逆转它的方向。尝试去掉这个负号，看看你自己更喜欢哪一种控制器部件方案。

下一个要包装的控制器部件是按键。Xbox 360 控制器上实际有 10 个按键，分别是 X、Y、A 和 B 按键，两个肩部按键，以及两个摇杆的前推操作。与前面一样，向引擎库项目添加下面的控制器部件类。

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Tao.Sdl;  
  
namespace Engine  
{  
    public class ControllerButton  
    {  
        IntPtr _joystick;  
        int _buttonId;  
  
        public bool Held { get; private set; }  
  
        public ControllerButton(IntPtr joystick, int buttonId)  
        {  
            _joystick = joystick;  
            _buttonId = buttonId;  
        }  
  
        public void Update()  
        {  
            byte buttonState = Sdl.SDL_JoystickGetButton(_joystick,  
_buttonId);  
            Held = (buttonState == 1);  
        }  
    }  
}
```

按键的 **Update** 函数会更新 **Held** 变量。现在就可以在控制器类中添加按键了。

```
public ControllerButton ButtonA { get; private set; }
public ControllerButton ButtonB { get; private set; }
public ControllerButton ButtonX { get; private set; }
public ControllerButton ButtonY { get; private set; }

// Front shoulder buttons
public ControllerButton ButtonLB { get; private set; }
public ControllerButton ButtonRB { get; private set; }

public ControllerButton ButtonBack { get; private set; }
public ControllerButton ButtonStart { get; private set; }

// If you press the control stick in
public ControllerButton ButtonL3 { get; private set; }
public ControllerButton ButtonR3 { get; private set; }
public XboxController(int player)
{
    _joystick = Sdl.SDL_JoystickOpen(player);
    LeftControlStick = new ControlStick(_joystick, 0, 1);
    RightControlStick = new ControlStick(_joystick, 4, 3);
    ButtonA = new ControllerButton(_joystick, 0);
    ButtonB = new ControllerButton(_joystick, 1);
    ButtonX = new ControllerButton(_joystick, 2);
    ButtonY = new ControllerButton(_joystick, 3);
    ButtonLB = new ControllerButton(_joystick, 4);
    ButtonRB = new ControllerButton(_joystick, 5);
    ButtonBack = new ControllerButton(_joystick, 6);
    ButtonStart = new ControllerButton(_joystick, 7);
    ButtonL3 = new ControllerButton(_joystick, 8);
    ButtonR3 = new ControllerButton(_joystick, 9);
}
```

按键都需要更新它们的状态。

```
public void Update()
{
    LeftControlStick.Update();
    RightControlStick.Update();
    ButtonA.Update();
    ButtonB.Update();
    ButtonX.Update();
    ButtonY.Update();
}
```

```
ButtonLB.Update();
ButtonRB.Update();
ButtonBack.Update();
ButtonStart.Update();
ButtonL3.Update();
ButtonR3.Update();
}
```

为了在屏幕上表示这些按键，还需要在测试状态中添加一个新函数。

```
private void DrawButtonPoint(bool held, int yPos)
{
    if (held)
    {
        Gl.glColor3f(0, 1, 0);
    }
    else
    {
        Gl.glColor3f(0, 0, 0);
    }
    Gl.glVertex2f(-400, yPos);
}
```

这个函数使得渲染所有的按键变得很容易，在按下这些按键后，它们的颜色也会发生改变。函数调用可以放在测试状态的 **Render** 函数中，渲染摇杆的轴的代码之后，**Gl.glEnd()** 语句之前。

```
DrawButtonPoint(_controller.ButtonA.Held, 300);
DrawButtonPoint(_controller.ButtonB.Held, 280);
DrawButtonPoint(_controller.ButtonX.Held, 260);
DrawButtonPoint(_controller.ButtonY.Held, 240);
DrawButtonPoint(_controller.ButtonLB.Held, 220);
DrawButtonPoint(_controller.ButtonRB.Held, 200);
DrawButtonPoint(_controller.ButtonBack.Held, 180);
DrawButtonPoint(_controller.ButtonStart.Held, 160);
DrawButtonPoint(_controller.ButtonL3.Held, 140);
DrawButtonPoint(_controller.ButtonR3.Held, 120);
```

运行测试状态，现在按键和轴都将显示在屏幕上。每个按下的按键都在屏幕上有视觉表示。

现在只剩下两类控制器部件需要处理：触发器按键和 **D-pad**。触发器实际上是通过一个轴表示的：左触发器由轴上的 0~1 表示，右触发器由轴上的 0~1 表示。这意味着包装触发器的代码与包装手柄的代码很类似。

```
using System;
```

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Tao.Sdl;

namespace Engine
{
    public class ControlTrigger
    {
        IntPtr _joystick;
        int _index;
        bool _top = false; // The triggers are treated as axes and need splitting
up
        float _deadZone = 0.24f;
        public float Value { get; private set; }

        public ControlTrigger(IntPtr joystick, int index, bool top)
        {
            _joystick = joystick;
            _index = index;
            _top = top;
        }

        public void Update()
        {
            Value = MapZeroToOne(Sdl.SDL_JoystickGetAxis(_joystick, _index));
        }

        private float MapZeroToOne(short value)
        {
            float output = ((float)value / short.MaxValue);

            if (_top == false)
            {
                if (output > 0)
                {
                    output = 0;
                }
                output = Math.Abs(output);
            }

            // Be careful of rounding error
        }
    }
}
```

```

        output = Math.Min(output, 1.0f);
        output = Math.Max(output, 0.0f);

        if (Math.Abs(output) < _deadZone)
        {
            output = 0;
        }

        return output;
    }
}
}

```

ControlTrigger 类在轴上进行处理，但是只接受轴的一半的值。手柄上只有两个触发器，所以添加到控制器类中的代码并不多。在构造函数中，使用相同的轴设置两个触发器。

最后，必须把两个触发器添加到控制器的 **Update** 函数中。

```

public ControlTrigger RightTrigger { get; private set; }
public ControlTrigger LeftTrigger { get; private set; }
// in the constructor
RightTrigger = new ControlTrigger(_joystick, 2, false);
LeftTrigger = new ControlTrigger(_joystick, 2, true);
// in the update function
RightTrigger.Update();
LeftTrigger.Update();

```

从视觉上表示这些触发器十分简单：再渲染两个点，每个点代表一个触发器。触发器被按下的程度越大，点移动的距离越远。使用的颜色稍有不同，以便可以把它与摇杆相区分。把这段代码添加到测试状态中按键和摇杆的可视化代码附近，要在 **glBegin** 和 **glEnd** 语句之间。

```

Gl.glColor3f(0.5f, 0, 0);
Gl.glVertex2f(50, _controller.LeftTrigger.Value * 300);
Gl.glColor3f(0, 0.5f, 0);
Gl.glVertex2f(-50, _controller.RightTrigger.Value * 300);

```

最后要添加的控制器部件是 **D-pad**。**D-pad** 将被视为上、下、左、右 4 个按键。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Tao.Sdl;

```

```

namespace Engine
{
    public class DPad
    {
        IntPtr _joystick;
        int _index;

        public bool LeftHeld { get; private set; }
        public bool RightHeld { get; private set; }
        public bool UpHeld { get; private set; }
        public bool DownHeld { get; private set; }
        public DPad(IntPtr joystick, int index)
        {
            _joystick = joystick;
            _index = index;
        }

        public void Update()
        {
            byte b = Sdl.SDL_JoystickGetHat(_joystick, _index);
            UpHeld = (b == Sdl.SDL_HAT_UP);
            DownHeld = (b == Sdl.SDL_HAT_DOWN);
            LeftHeld = (b == Sdl.SDL_HAT_LEFT);
            RightHeld = (b == Sdl.SDL_HAT_RIGHT);
        }
    }
}

```

控制器只有一个 D-pad，但仍然需要添加它。

```

public DPad Dpad { get; private set; }

public XboxController(int player)
{
    _joystick = Sdl.SDL_JoystickOpen(player);
    Dpad = new DPad(_joystick, 0);
    // ... later in the code
    public void Update()
    {
        Dpad.Update();
    }
}

```

还需要把 D-pad 添加到更新循环中，这样就完成了对控制器上所有控制器部件的包装。最后，可以通过重用按键显示代码来可视化 D-pad。

```

DrawButtonPoint(_controller.Dpad.UpHeld, 80);

```

```
DrawButtonPoint(_controller.Dpad.DownHeld, 60);  
DrawButtonPoint(_controller.Dpad.LeftHeld, 40);  
DrawButtonPoint(_controller.Dpad.RightHeld, 20);
```

现在就提供了对 Xbox 360 控制器上全部控制器部件的支持, 利用这些代码构建对其他类型的控制器的支持, 相对来说也很简单。控制器使用起来十分简单, 但是按键还需要再进行一些处理。现在, 按键会报告它是否被按下, 而在游戏中, 知道按键是否被按下通常更有帮助。例如, 按下按键一次可以用来选择菜单选项或者实现射击操作。这段代码很简单, 把它们添加到 **ControllerButton** 类中。

```
bool _wasHeld = false;  
public bool Pressed { get; private set; }  
public void Update()  
{  
    // reset the pressed value  
    Pressed = false;  
  
    byte buttonState = Sdl.SDL_JoystickGetButton(_joystick, _buttonId);  
    Held = (buttonState == 1);  
  
    if (Held)  
    {  
        if(_wasHeld == false)  
        {  
            Pressed = true;  
        }  
        _wasHeld = true;  
    }  
    else  
    {  
        _wasHeld = false;  
    }  
}
```

只有在按下按键的一帧中, **Pressed** 才是 **true**, 而只要按键被按下, **Held** 的值就为 **true**。控制器类和它使用的各种控制器部件会产生大量的代码, 这些代码彼此之间存在一定的关系, 如果把它们组织到一个单独的名称空间 **Engine.Input** 中, 会更容易使用。重新组织代码是构建可重用库的一个重要部分。所有控制器类都和输入有关, 所以可以创建一个单独的输入子库。

右击引擎项目, 然后从快捷菜单中选择 **New Folder** 命令, 如图 9-13 所示。

将新文件夹命名为 **Input**, 然后把所有的控制器部件类拖放到该文件夹中。还应该把 **Input** 类添加到这个文件夹中, 得到的结果如图 9-14 所示。只要在 **Input** 文件夹中创建一个新类, 它就将自动使用名称空间 **Engine.Input**。对于我们刚才添加的类, 需要手动修改它们的名称空间。方法是 **namespace Engine** 一行修改为 **namespace Engine.Input**。

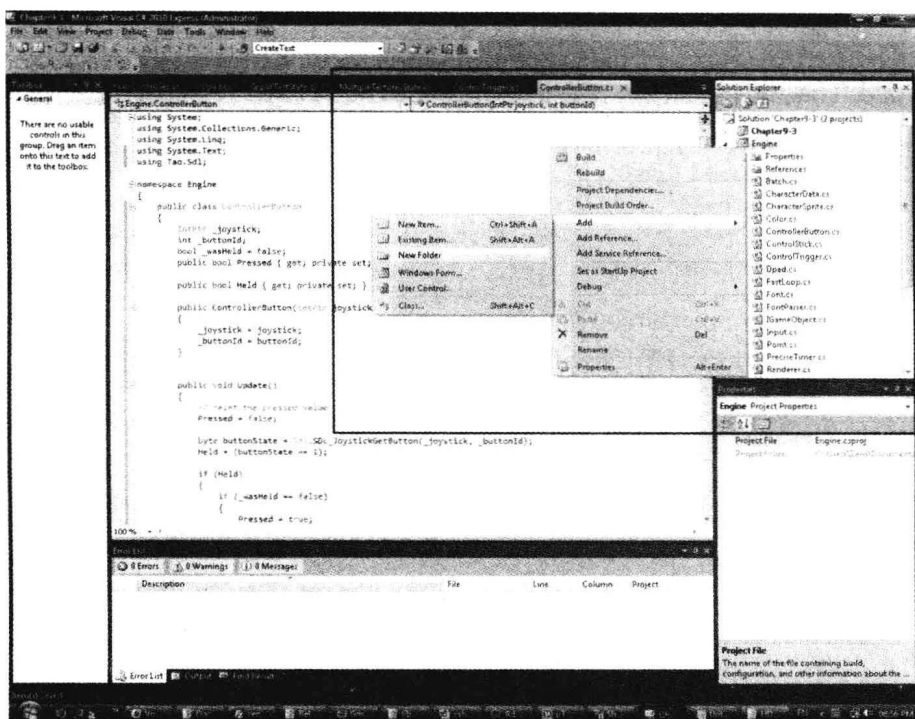


图 9-13 创建新的项目子文件夹

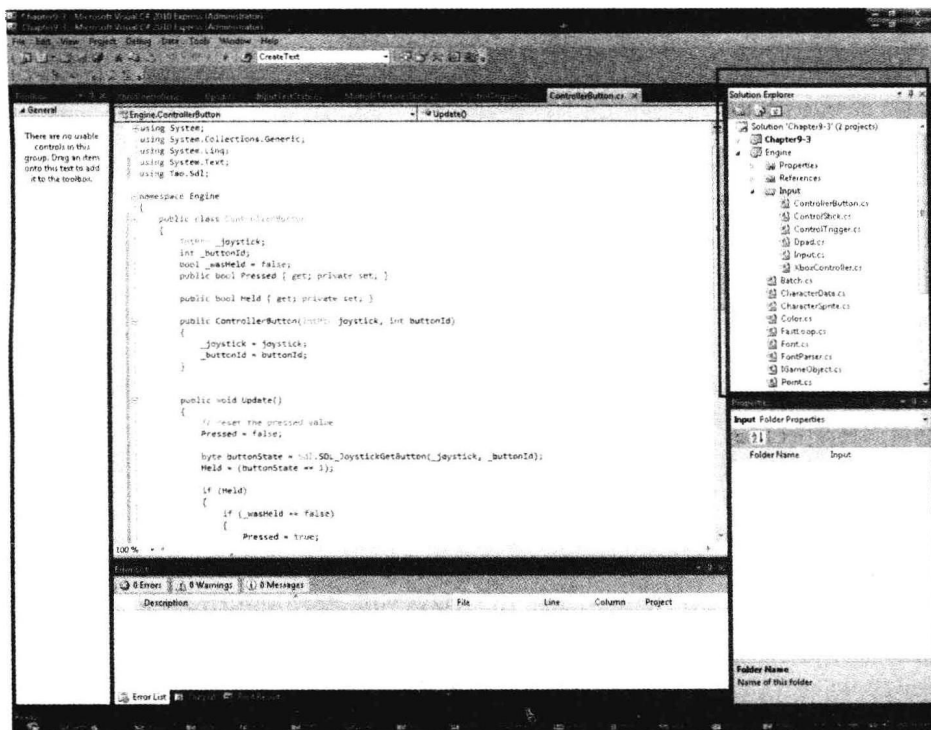


图 9-14 分离出 Input 类

尝试运行代码。可能会产生一些错误，指出不能找到输入类。为了解决这些错误，在文件顶部添加语句 `using Engine.Input;`。

最后，应该把控制器添加到 `Input` 类中。在这里使用了 Xbox 360 控制器，并假定如果任何用户想要使用一个不同的控制器，该控制器将具有与 Xbox 360 控制器相同的功能。

```
public class Input
{
    public Point MousePosition { get; set; }
    bool _usingController = false;
    XboxController Controller { get; set; }

    public Input()
    {
        Sdl.SDL_InitSubSystem(Sdl.SDL_INIT_JOYSTICK);
        if (Sdl.SDL_NumJoysticks() > 0)
        {
            Controller = new XboxController(0);
            _usingController = true;
        }
    }

    public void Update(double elapsedTime)
    {
        if (_usingController)
        {
            Sdl.SDL_JoystickUpdate();
            Controller.Update();
        }
    }
}
```

这里的 `Input` 类只支持一个控制器，但是将其扩展为支持多个控制器并不困难。

9.4.2 添加更好的鼠标支持

现在对鼠标的支持还很简单。在 `form.cs` 中计算出光标相对于窗体的位置，然后使用这个位置来更新输入类的鼠标位置。鼠标输入与窗体绑定在一起，所以在一定程度上，输入类必须知道窗体。在 `Engine.Input` 名称空间中创建一个新类 `Mouse`，用于存储关于鼠标的当前位置的信息。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
```

```
using System.Windows.Forms;

namespace Engine.Input
{
    public class Mouse
    {
        Form _parentForm;
        Control _openGLControl;

        public Point Position { get; set; }

        public Mouse(Form form, Control openGLControl)
        {
            _parentForm = form;
            _openGLControl = openGLControl;
        }

        public void Update(double elapsedTime)
        {
            UpdateMousePosition();
        }

        private void UpdateMousePosition()
        {
            System.Drawing.Point mousePos = Cursor.Position;
            mousePos = _openGLControl.PointToClient(mousePos);

            // Now use our point definition,
            Engine.Point adjustedMousePoint = new Engine.Point();
            adjustedMousePoint.X = (float)mousePos.X - ((float)_parentForm.
                ClientSize.Width / 2);
            adjustedMousePoint.Y = ((float)_parentForm.ClientSize.Height / 2)
                - (float)mousePos.Y;
            Position = adjustedMousePoint;
        }
    }
}
```

Mouse 类使用传入到构造函数中的窗体和 **OpenGL** 控件来更新自己的位置。需要把这个新的 **Mouse** 类添加到 **Input** 类中。

```
public class Input
{
    public Mouse Mouse { get; set; }
```

它替换之前的代码以获得鼠标位置。这个新的 `Mouse` 类还需要使它的 `Update` 函数可以从 `Input` 类中调用。

```
public void Update(double elapsedTime)
{
    if (_usingController)
    {
        Sdl.SDL_JoystickUpdate();
        Controller.Update();
    }
    Mouse.Update(elapsedTime);
}
```

`Mouse` 类没有在 `Input` 类中构造，因为 `Input` 类不用知道窗体或者简单的 `OpenGL` 控件的信息。相反，应在 `form.cs` 构造函数中构造鼠标对象。

```
public Form1()
{
    InitializeComponent();
    simpleOpenGLControll1.InitializeContexts();
    _input.Mouse = new Mouse(this, simpleOpenGLControll1);
}
```

现在已经创建了 `Mouse` 类，窗体类中的 `UpdateInput` 函数可以被简化了。

```
private void GameLoop(double elapsedTime)
{
    UpdateInput(elapsedTime);
    _system.Update(elapsedTime);
    _system.Render();
    simpleOpenGLControll1.Refresh();
}

private void UpdateInput(double elapsedTime)
{
    // Previous mouse code removed.
    _input.Update(elapsedTime);
}
```

`elapsedTime` 从主 `GameLoop` 方法传递给 `UpdateInput` 方法。

考虑到我们可能会想要让鼠标支持悬停事件，将 `elapsedTime` 传递给了 `UpdateInput` 和 `Update` 函数。例如，在实时策略游戏中，可能会将鼠标指针在某个对象上悬停一两秒，检测到这种悬停行为后，游戏可能弹出一个工具提示，显示该对象的名称和各种相关信息。

可以按照处理控制器按键的方式处理鼠标按键。按键将包含 `Held` 和 `Pressed` 成员。对于鼠标，`Pressed` 成员的状态对应于 `Windows` 窗体的单击事件。游戏手柄通过轮询进行工作，

这意味着在每一帧中，程序查询游戏手柄以确定哪些按键被按下，以及摇杆的位置在哪里。鼠标的工作方式与游戏手柄稍有不同，轮询起来不太容易。Windows 窗体控件与鼠标事件是关联在一起的。事件是当发生某个动作时调用代码的一种方式，例如移动鼠标，以及单击或双击鼠标按键。这些事件可以和函数关联起来。我们将使用 Click 事件来确定鼠标按键何时被按下，使用 Up 和 Down 事件来确定何时鼠标按键被按住不放。

```
bool _leftClickDetect = false;
bool _rightClickDetect = false;
bool _middleClickDetect = false;

public bool MiddlePressed { get; private set; }
public bool LeftPressed { get; private set; }
public bool RightPressed { get; private set; }

public bool MiddleHeld { get; private set; }
public bool LeftHeld { get; private set; }
public bool RightHeld { get; set; }

public Mouse(Form form, Control openGLControl)
{
    _parentForm = form;
    _openGLControl = openGLControl;
    _openGLControl.MouseClick += delegate(object obj, MouseEventArgs e)
    {
        if (e.Button == MouseButton.Left)
        {
            _leftClickDetect = true;
        }
        else if (e.Button == MouseButton.Right)
        {
            _rightClickDetect = true;
        }
        else if (e.Button == MouseButton.Middle)
        {
            _middleClickDetect = true;
        }
    };

    _openGLControl.MouseDown += delegate(object obj, MouseEventArgs e)
    {
        if (e.Button == MouseButton.Left)
        {
            LeftHeld = true;
        }
    }
}
```

```
    }
    else if (e.Button == MouseButton.Right)
    {
        RightHeld = true;
    }
    else if (e.Button == MouseButton.Middle)
    {
        MiddleHeld = true;
    }
};

_openGLControl.MouseUp += delegate(object obj, MouseEventArgs e)
{
    if (e.Button == MouseButton.Left)
    {
        LeftHeld = false;
    }
    else if (e.Button == MouseButton.Right)
    {
        RightHeld = false;
    }
    else if (e.Button == MouseButton.Middle)
    {
        MiddleHeld = false;
    }
};

_openGLControl.MouseLeave += delegate(object obj, EventArgs e)
{
    // If you move the mouse out the window then release all held buttons
    LeftHeld = false;
    RightHeld = false;
    MiddleHeld = false;
};
}

public void Update(double elapsedTime)
{
    UpdateMousePosition();
    UpdateMouseButtons();
}

private void UpdateMouseButtons()
```

```
{
    // Reset buttons
    MiddlePressed = false;
    LeftPressed = false;
    RightPressed = false;

    if (_leftClickDetect)
    {
        LeftPressed = true;
        _leftClickDetect = false;
    }

    if (_rightClickDetect)
    {
        RightPressed = true;
        _rightClickDetect = false;
    }

    if (_middleClickDetect)
    {
        MiddlePressed = true;
        _middleClickDetect = false;
    }
}
```

在鼠标的构造函数中，有 4 个 OpenGL 控件事件关联着匿名委托。鼠标单击事件会检测到鼠标的左、右或中间按键是否被按下，并设置一个布尔值来报告是否发生了该事件。在 `UpdateMouseButtons` 函数中，检测布尔值用于设置公有的 `MiddlePressed`、`LeftPressed` 和 `RightPressed` 变量。按下事件只应该针对一个帧发生，所以在函数的开始将所有的按键按下标志设为 `false`。在重设之后，检查检测变量，如果检测到按下事件，则把检测变量设置为 `true`。以同样的方式可以支持双击事件。

接下来的 3 个事件确定是否有鼠标按键被按住不放了。`Down` 事件检测鼠标按键何时被按下，`Up` 事件检测鼠标按键何时被释放。这些事件十分简单，它们会切换每个按键的 `Held` 状态。第三个事件检测鼠标何时离开控件。这一点很重要，因为一旦鼠标离开控件，就无法再报告事件。这意味着用户可以在控件内部单击，离开控件，并释放鼠标按键。释放事件将不会被传递，而 `Held` 标志将无法与鼠标的实际状态保持一致。因此，如果鼠标离开了控件区域，那么所有的 `Held` 标志都将被设为 `false`。

通过创建新的游戏状态 `MouseTestState`，并使其成为 `EngineTest` 项目中的默认游戏状态，可以测试鼠标输入。

```
class MouseTestState : IGameObject
{
```

```
Input _input;

bool _leftToggle = false;
bool _rightToggle = false;
bool _middleToggle = false;

public MouseTestState(Input input)
{
    _input = input;
}

private void DrawButtonPoint(bool held, int yPos)
{
    if (held)
    {
        Gl.glColor3f(0, 1, 0);
    }
    else
    {
        Gl.glColor3f(0, 0, 0);
    }
    Gl.glVertex2f(-400, yPos);
}

public void Render()
{
    Gl.glDisable(Gl.GL_TEXTURE_2D);
    Gl.glClearColor(1, 1, 1, 0);

    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    Gl.glPointSize(10.0f);
    Gl.glBegin(Gl.GL_POINTS);
    {
        Gl.glColor3f(1, 0, 0);
        Gl.glVertex2f(_input.Mouse.Position.X, _input.Mouse.Position.Y);

        if (_input.Mouse.LeftPressed)
        {
            _leftToggle = !_leftToggle;
        }

        if (_input.Mouse.RightPressed)
```

```
{
    _rightToggle = !_rightToggle;
}

if (_input.Mouse.MiddlePressed)
{
    _middleToggle = !_middleToggle;
}

DrawButtonPoint(_leftToggle, 0);
DrawButtonPoint(_rightToggle, -20);
DrawButtonPoint(_middleToggle, -40);

DrawButtonPoint(_input.Mouse.LeftHeld, 40);
DrawButtonPoint(_input.Mouse.RightHeld, 60);
DrawButtonPoint(_input.Mouse.MiddleHeld, 80);
}
Gl.glEnd();
}

public void Update(double elapsedTime)
{
}
}
```

这个测试状态重用了之前测试游戏手柄的 `InputTestState` 中的 `DrawButtonPoint` 函数。这个测试状态将渲染一个在鼠标光标的下面的点，然后渲染另外 6 个点来代表按键状态。顶部的 3 个点代表每个按键的 `Held` 状态。按下一个按键，与它相应的点将会变亮，释放该按键，它会变暗。底部的 3 个按键代表按键的按下状态。每一次单击按键时，代表该按键的点将会切换颜色。

对于基本的鼠标支持，只需要这些代码。另外，还可以添加其他一些功能，例如检测鼠标的双击动作，或者创建一种方法来轮询滚轮操作，但是对于大多数游戏来说，目前介绍的代码足够了。还剩下的唯一一个需要添加的控制方法是键盘。

9.4.3 添加键盘支持

键盘与游戏手柄和鼠标不同，因为与键盘进行交互的方法取决于具体的情况。如果使用一个界面来让用户输入他的姓名，那么需要一个回调函数来报告用户按下的每个字符。但是在格斗游戏中，只需要确定是否按下了特定的键，而不关心其余的键。上面是两种交互模式，每一种都很重要，需要提供对它们的支持。

键盘与鼠标一样，使用一个基于事件的系统。窗体具有 `OnKeyDown` 和 `OnKeyUp` 事件，可以附加委托。但是，这些事件会忽略方向键，但是在游戏中方向键很重要，经常用于控制移动。除了方向键，`Alt` 键等控制键也会被忽略。这些键一般在窗体中都有某种含义，因

此不用于一般用途。游戏需要使用这些键，所以需要实现另外一种轮询键的方法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.InteropServices;
using System.Windows.Forms;

namespace Engine.Input
{
    public class Keyboard
    {
        [DllImport("User32.dll")]
        public static extern short GetAsyncKeyState(int vKey);

        Control _openGLControl;
        public KeyPressEventHandler KeyPressEvent;

        class KeyState
        {
            bool _keyPressDetected = false;
            public bool Held { get; set; }
            public bool Pressed { get; set; }

            public KeyState()
            {
                Held = false;
                Pressed = false;
            }

            internal void OnDown()
            {
                if (Held == false)
                {
                    _keyPressDetected = true;
                }
                Held = true;
            }

            internal void OnUp()
            {
                Held = false;
            }
        }
    }
}
```

```
    }

    internal void Process()
    {
        Pressed = false;
        if (_keyPressDetected)
        {
            Pressed = true;
            _keyPressDetected = false;
        }
    }
}

Dictionary<Keys, KeyState> _keyStates = new Dictionary<Keys,
    KeyState>();

public Keyboard(Control openGLControl)
{
    _openGLControl = openGLControl;
    _openGLControl.KeyDown += new KeyEventHandler(OnKeyDown);
    _openGLControl.KeyUp += new KeyEventHandler(OnKeyUp);
    _openGLControl.KeyPress += new KeyPressEventHandler(OnKeyPress);
}

void OnKeyPress(object sender, KeyPressEventArgs e)
{
    if (KeyPressEvent != null)
    {
        KeyPressEvent(sender, e);
    }
}

void OnKeyUp(object sender, KeyEventArgs e)
{
    EnsureKeyStateExists(e.KeyCode);
    _keyStates[e.KeyCode].OnUp();
}

void OnKeyDown(object sender, KeyEventArgs e)
{
    EnsureKeyStateExists(e.KeyCode);
    _keyStates[e.KeyCode].OnDown();
}
```

```
private void EnsureKeyStateExists(Keys key)
{
    if (!_keyStates.Keys.Contains(key))
    {
        _keyStates.Add(key, new KeyState());
    }
}

public bool IsKeyPressed(Keys key)
{
    EnsureKeyStateExists(key);
    return _keyStates[key].Pressed;
}

public bool IsKeyHeld(Keys key)
{
    EnsureKeyStateExists(key);
    return _keyStates[key].Held;
}

public void Process()
{
    ProcessControlKeys();
    foreach (KeyState state in _keyStates.Values)
    {
        // Reset state.
        state.Pressed = false;
        state.Process();
    }
}

private bool PollKeyPress(Keys key)
{
    return (GetAsyncKeyState((int)key) != 0);
}

private void ProcessControlKeys()
{
    UpdateControlKey(Keys.Left);
    UpdateControlKey(Keys.Right);
    UpdateControlKey(Keys.Up);
    UpdateControlKey(Keys.Down);
    UpdateControlKey(Keys.LMenu); // this is the left alt key
}
```

```
    }

    private void UpdateControlKey(Keys keys)
    {
        if (PollKeyPress(keys))
        {
            OnKeyDown(this, new KeyEventArgs(keys));
        }
        else
        {
            OnKeyUp(this, new KeyEventArgs(keys));
        }
    }
}

}
```

键盘状态把每个键视作具有 **Pressed** 和 **Held** 成员的按键。子类 **KeyState** 包含键盘上每个键的状态。**Keyboard** 构造函数接受对 **OpenGL** 控件的引用作为参数，并添加对其 **KeyUp** 和 **KeyDown** 事件的委托。然后，这些事件用于更新整个键盘的状态。**KeyPress** 事件也有一个委托，这又会触发另外一个事件 **KeyPressEvent**，并通过该事件传递数据。当用户输入玩家名或数据时，将使用 **KeyPressEvent**。当使用键盘作为游戏设备时，可以把键视为按键，并使用 **IsKeyPressed** 和 **IsKeyHeld** 查询它们。

键盘类中稍微有些复杂的部分是键的轮询。为了实现这种轮询，需要导入 **User32.dll** 中的 **C** 函数，并在文件的顶部添加 **using System.Runtime.InteropServices;** 语句。**KeyUp** 和 **KeyDown** 事件不会针对方向键触发，所以这些键的状态可通过 **GetAsyncKeyState** 确定。当按下键时，**PollKeyPress** 函数使用 **GetAsyncKeyState** 返回 **true**，否则返回 **false**。在每一帧中都会轮询方向键和 **Alt** 键，并更新其状态。

创建一个新的测试状态来确认键盘可以工作。这里没有这个测试状态的代码，因为它与鼠标状态十分类似。测试普通的键和方向键，以确认代码可以正确工作。感到满意之后，对游戏引擎的修改就完成了。下一步，我们就要开始创建游戏。

第 10 章

创建一个简单的卷轴射击游戏

本章将介绍如何开发一个简单的游戏，首先设计一个基本的计划，然后展示其实现过程。实现将以实效的迭代方式完成。高层次的第一遍开发将使游戏结构可以工作，然后优化这个结构，直到它接近对游戏的最初描述。

10.1 一个简单的游戏

通过一个简单的游戏就可以演示到目前为止介绍的所有技术。我们将开发一个 2D 卷轴射击游戏。这类游戏开发起来很简单，而且可以通过添加更多特性进行扩展。开发这个游戏的一种实效方法是尽快创建一个可以工作的游戏，但是提前计划好最初的各个阶段仍然十分重要。图 10-1 显示了游戏流程的一个高层概览。

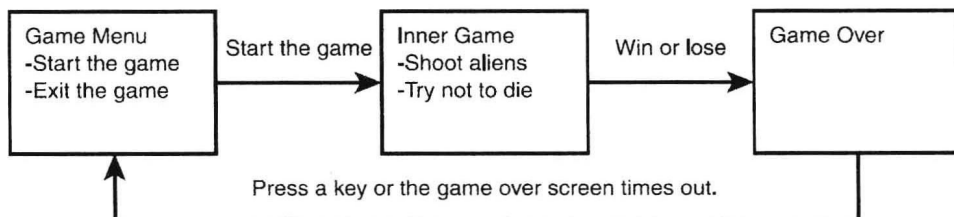


图 10-1 高层游戏流程

这个游戏的思想是创建一个简单但是完整的游戏。因此，游戏将包括启动屏幕、游戏主体和游戏结束屏幕。游戏主体中将包含玩家可以移动的飞船。玩家应该能够通过按下按键来从飞船前部发射子弹。有一个关卡就可以了，但是多添加几个关卡的效果会更好。当

玩家从开始状态进入游戏主体后，关卡便开始。在一定的时间后，关卡结束。如果在关卡结束后玩家仍然存活，则认为玩家胜利，否则玩家失败。游戏中需要一些敌人，它们可以朝向玩家移动，并且能够发射子弹。敌人具有生命值，需要被击中几次后才会死亡。敌人死亡时应该显示爆炸效果。

通过阅读这个简短的游戏描述，很容易开始构建一个类和交互的列表。一种不错的方法是，为主类绘制几个框，然后绘制一些箭头来表示主要的交互。我们需要 3 个主游戏状态，其中游戏主体状态最复杂。通过游戏描述，可以看到需要的一些重要的类包括 **Player**、**Level**、**Enemy** 和 **Bullet**。关卡需要包含和更新玩家、敌人和子弹。子弹可以与敌人和玩家发生碰撞。

在游戏的主体中，玩家可以操纵飞船并消灭侵犯自己的敌人。玩家的飞船并不会在太空中实际移动，相反，移动效果是模拟出来的。玩家可以把飞船移动到屏幕上的任意位置，但是“摄像机”总是停留在屏幕正中。为了产生在太空中加速穿行的效果，背景将朝向与玩家前进的方向相反的方向进行卷动。这可以显著地简化玩家跟踪代码或摄像机代码。

这个游戏很小，所以借助这个不太正式的描述就可以开始编码了。所有的游戏代码都放在游戏项目中，而我们生成的可用于多个项目的代码可放在引擎库中。一个更加认真的游戏计划可能需要几个小测试程序，游戏状态非常适合为这种编码思想构建一个基本结构。

10.2 第一遍实现

高层视图把游戏分为了 3 个状态。第一遍编码将创建这 3 个状态，并使它们可以工作。

新建一个 **Windows Forms Application** 项目。我把这个项目命名为 **Shooter**，但是你可以随意选择其他的名称。这里概述一下设置项目的过程。建立解决方案的方式与前面章节建立 **EngineTest** 项目的方式相似。**Shooter** 项目使用了下面的引用：**Tao.DevIL**、**Tao.OpenGL**、**Tao.Platform.Windows** 和 **System.Drawing**。它还需要引擎 **Engine** 项目。为此，应该把 **Engine** 项目添加到解决方案中(右击 **Solution** 文件夹，在弹出的快捷菜单中选择 **Add Existing Project** 命令，然后找到并添加 **Engine** 项目)。解决方案中有了 **Engine** 项目后，就可以添加对它的引用了。为了添加对 **Engine** 项目的引用，右击 **Shooter** 项目引用文件夹，在弹出的快捷菜单中选择 **Add Reference** 命令，然后打开 **Projects** 选项卡，选择 **Engine** 项目。

Shooter 项目将使用 **OpenGL**，所以在 **Form** 编辑器中，将 **SimpleOpenGLControl** 拖放到窗体上，并将其 **Dock** 属性设为 **Fill**。右击 **Form1.cs**，在弹出的快捷菜单中选择 **View Code** 命令。需要为这个文件添加游戏循环和初始化代码，如下所示。

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
```

```
using System.Windows.Forms;
using Engine;
using Engine.Input;
using Tao.OpenGl;
using Tao.DevIl;

namespace Shooter
{
    public partial class Form1 : Form
    {
        bool _fullscreen = false;
        FastLoop _fastLoop;
        StateSystem _system = new StateSystem();
        Input _input = new Input();
        TextureManager _textureManager = new TextureManager();
        SoundManager _soundManager = new SoundManager();

        public Form1()
        {
            InitializeComponent();
            simpleOpenGLControll1.InitializeContexts();

            _input.Mouse = new Mouse(this, simpleOpenGLControll1);
            _input.Keyboard = new Keyboard(simpleOpenGLControll1);

            InitializeDisplay();
            InitializeSounds();
            InitializeTextures();
            InitializeFonts();
            InitializeGameState();

            _fastLoop = new FastLoop(GameLoop);
        }

        private void InitializeFonts()
        {
            // Fonts are loaded here.
        }

        private void InitializeSounds()
        {
            // Sounds are loaded here.
        }
    }
}
```

```
private void InitializeGameState()
{
    // Game states are loaded here
}

private void InitializeTextures()
{
    // Init DevIL
    Il.ilInit();
    Ilu.iluInit();
    Ilut.ilutInit();
    Ilut.ilutRenderer(Ilut.ILUT_OPENGL);

    // Textures are loaded here.
}

private void UpdateInput(double elapsedTime)
{
    _input.Update(elapsedTime);
}

private void GameLoop(double elapsedTime)
{
    UpdateInput(elapsedTime);
    _system.Update(elapsedTime);
    _system.Render();
    simpleOpenGLControl1.Refresh();
}

private void InitializeDisplay()
{
    if (_fullscreen)
    {
        FormBorderStyle = FormBorderStyle.None;
        WindowState = FormWindowState.Maximized;
    }
    else
    {
        ClientSize = new Size(1280, 720);
    }
    Setup2DGraphics(ClientSize.Width, ClientSize.Height);
}
```



```

protected override void OnClientSizeChanged(EventArgs e)
{
    base.OnClientSizeChanged(e);
    Gl.glViewport(0, 0, this.ClientSize.Width, this.ClientSize.
Height);
    Setup2DGraphics(ClientSize.Width, ClientSize.Height);
}
private void Setup2DGraphics(double width, double height)
{
    double halfWidth = width / 2;
    double halfHeight = height / 2;
    Gl.glMatrixMode(Gl.GL_PROJECTION);
    Gl.glLoadIdentity();
    Gl.glOrtho(-halfWidth, halfWidth, -halfHeight, halfHeight,
-100, 100);
    Gl.glMatrixMode(Gl.GL_MODELVIEW);
    Gl.glLoadIdentity();
}
}
}

```

在这个 **Form.cs** 的代码中，创建了一个 **Keyboard** 对象，并把它指派给了 **Input** 对象。为使这段代码可以工作，必须在 **Input** 类中添加一个 **Keyboard** 成员，如下所示。

```

public class Input
{
    public Mouse Mouse { get; set; }
    public Keyboard Keyboard { get; set; }
    public XboxController Controller { get; set; }
}

```

需要把下面列出的 DLL 文件添加到 **bin\Debug** 和 **bin\Release** 文件夹中：**alut.dll**、**DevIL.dll**、**ILU.dll**、**OpenAL32.dll** 和 **SDL.dll**。现在，这个项目就可以用于开发游戏了。

这是我们创建的第一个游戏，当游戏运行时，如果窗体的标题栏显示的不是 **Form1**，而是其他内容，那就更好了。在 **Visual Studio** 中修改这些文本很容易。在 **Solution Explorer** 中双击文件 **Form1.cs**，打开窗体设计器。单击窗体，并进入 **Properties** 窗口(如果没有看到 **Properties** 窗口，则从菜单栏中选择 **View|Properties Window** 命令)。**Properties** 窗口中列出了窗体的全部属性。找到 **Text** 属性，将其值改为 **Shooter**，如图 10-2 所示。

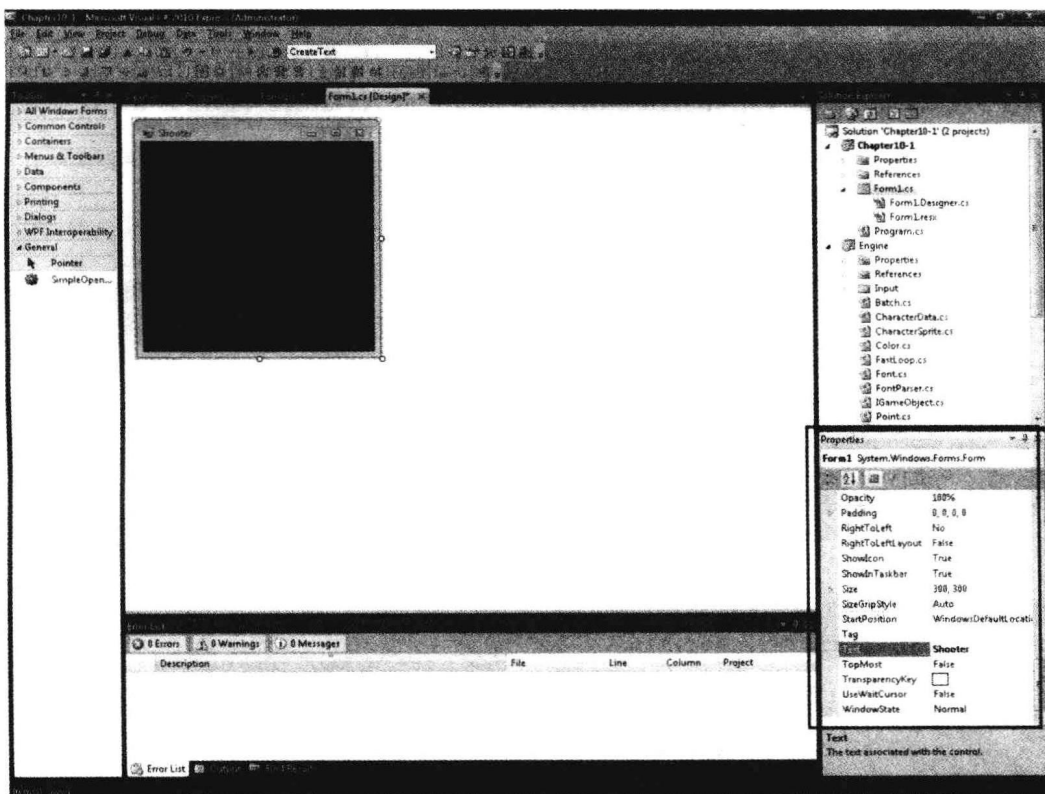


图 10-2 修改窗体标题

10.2.1 开始菜单的状态

第一个要创建的状态是开始菜单。在第一遍编码中,菜单只提供了两个选项: **Start Game** 和 **Exit**。这些选项是一种按钮,所以这个状态需要两个按钮和一些标题文本。图 10-3 显示了这个屏幕的模拟图。

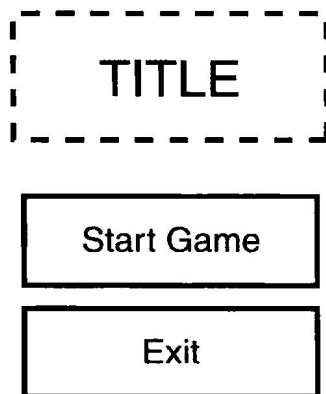


图 10-3 屏幕的模拟图

标题将使用本书前面定义的 **Font** 类和 **Text** 类创建。本书配套光盘中有有一个字体叫做

title font, 这是 48 像素的字体, 具有适合在视频游戏中使用的外形。将 **.fnt** 和 **.tga** 文件添加到项目中, 并设置它们的属性, 以便在生成项目时它们会被复制到 **bin** 目录中。

需要把字体文件加载到 **Form.cs** 文件中。如果要处理的字体很多, 可能有必要创建一个 **FontManager** 类, 但是因为我们只会使用一种或两种字体, 所以可以把它们存储为成员变量。加载字体文件的代码如下。

```
private void InitializeTextures()
{
    // Init DevIL
    IL.ilInit();
    ILU.iluInit();
    ILUT.ilutInit();
    ILUT.ilutRenderer(ILUT_OPENGL);

    // Textures are loaded here.
    _textureManager.LoadTexture("title_font", "title_font.tga");
}

Engine.Font _titleFont;
private void InitializeFonts()
{
    _titleFont = new Engine.Font(_textureManager.Get("title_font"),
        FontParser.Parse("title_font.fnt"));
}
```

字体纹理在 **InitializeTextures** 函数中加载, 当在 **InitializeFonts** 方法中创建字体对象时将使用这个纹理。

可以把标题字体传递到 **StartMenuState** 构造函数中。将下面的新的 **StartMenuState** 添加到 **Shooter** 项目中。

```
class StartMenuState : IGameObject
{
    Renderer _renderer = new Renderer();
    Text _title;

    public StartMenuState(Engine.Font titleFont)
    {
        _title = new Text("Shooter", titleFont);
        _title.SetColor(new Color(0, 0, 0, 1));
        // Center on the x and place somewhere near the top
        _title.SetPosition(_title.Width/2, 300);
    }

    public void Update(double elapsedTime) { }
```

```
public void Render()
{
    Gl.glClearColor(1, 1, 1, 0);

    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    _renderer.DrawText(_title);
    _renderer.Render();
}
```

StartMenuState 使用传入构造函数的字体创建标题文本。文本的颜色为黑色，水平居中显示。渲染循环将屏幕清除为白色，然后绘制文本。为了运行这个状态，需要把它添加到状态系统中，并设置为默认状态。

```
private void InitializeGameState()
{
    // Game states are loaded here
    _system.AddState("start_menu", new StartMenuState(_titleFont));
    _system.ChangeState("start_menu");
}
```

运行程序后，看到的结果如图 10-4 所示。

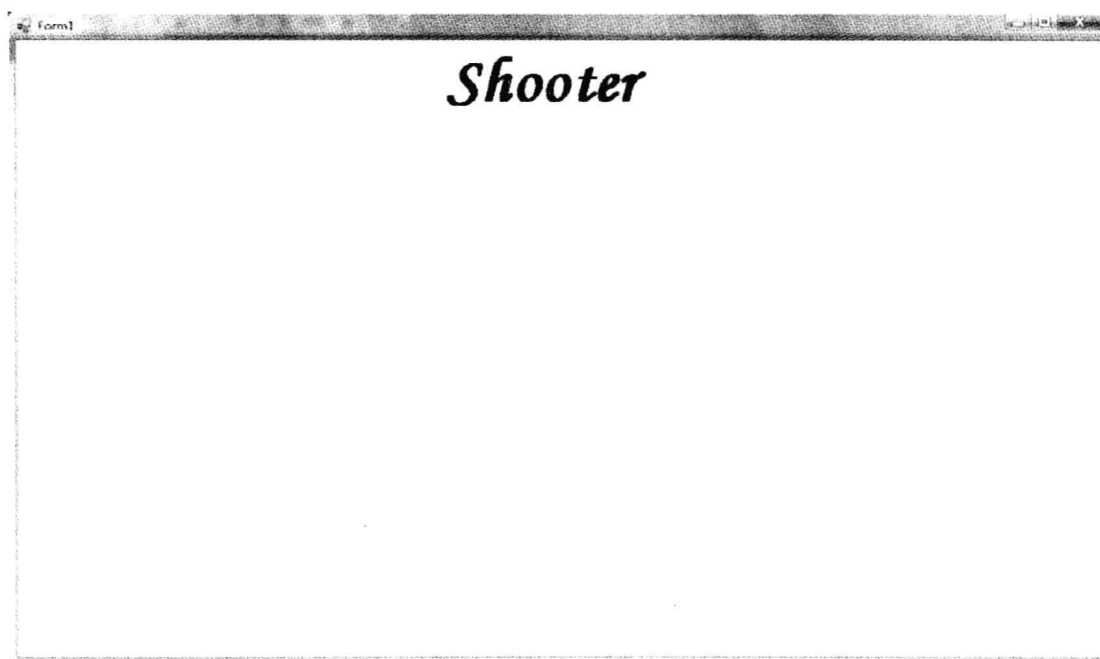


图 10-4 渲染标题

这个阶段只是第一遍处理。在以后可以细化标题，使其更加美观。就现在而言，功能是最重要的。为了完成标题屏幕，需要添加开始和退出选项。这两个选项都是按钮，意味着需要创建一个按钮类。

按钮将在垂直列表中表示。任何时候，都会有一个按钮是选中的按钮。如果用户按下了键盘上的 Enter 键，或者游戏手柄上的 A 按键，那么当前选择的按钮将被按下。

按钮需要知道自己何时被选中，也就是获得焦点。它们还需要知道自己被按下时应该采取什么操作，这是一个非常适合使用委托的场合。在构造函数中可以向按钮传递一个委托，当按下按钮时就调用这个委托。按钮还需要设置自己位置的方法。使用代码表示对按钮的这些需求后，得到的类如下所示。**Button** 类是可以重用的，所以可以把它添加到 **Engine** 项目中，以便将来的项目也可以使用该类。

```
public class Button
{
    EventHandler _onPressEvent;
    Text _label;
    Vector _position = new Vector();

    public Vector Position
    {
        get { return _position; }
        set
        {
            _position = value;
            UpdatePosition();
        }
    }

    public Button(EventHandler onPressEvent, Text label)
    {
        _onPressEvent = onPressEvent;
        _label = label;
        _label.SetColor(new Color(0, 0, 0, 1));
        UpdatePosition();
    }

    public void UpdatePosition()
    {
        // Center label text on position.
        _label.SetPosition(_position.X - (_label.Width / 2),
            _position.Y + (_label.Height / 2));
    }
}
```

```
public void OnGainFocus()
{
    _label.SetColor(new Color(1, 0, 0, 1));
}

public void OnLoseFocus()
{
    _label.SetColor(new Color(0, 0, 0, 1));
}

public void OnPress()
{
    _onPressEvent(this, EventArgs.Empty);
}

public void Render(Renderer renderer)
{
    renderer.DrawText(_label);
}
}
```

Button 类不直接处理用户输入，相反，它依赖于使用它的代码向它传递相关的输入事件。**OnGainFocus** 和 **OnLoseFocus** 方法根据焦点的状态修改按钮的外观，这样用户就可以知道当前选中了哪个按钮。当按钮的位置改变时，标签文本的位置也会更新并居中。**EventHandler** 中包含当按下按钮时调用的函数，它描述了一个接受对象和事件参数枚举作为参数的委托。

玩家输入由 **Menu** 类检测，它通知相关按钮已被选中或按下。**Menu** 类包含一个按钮列表，任何时候只有一个按钮可以拥有焦点。用户可以使用手柄或者键盘导航菜单。**OnGainFocus** 和 **OnLoseFocus** 会修改按钮的标签文本，这样用户就可以知道当前哪个按钮拥有焦点。

获得焦点时，字体的颜色变为红色，否则字体的颜色为黑色。还有其他进行区分的方法，例如可以放大文本、修改背景图片或者使用补间显示或者删除其他某个值，但是现在只是第一遍编码，所以不需要进行这些改进。

菜单将把按钮垂直排列在一列中，所以将它命名为 **VerticalMenu** 很合适。**VerticalMenu** 也是一个可重用的类，也可以添加到 **Engine** 项目中。菜单还需要添加按钮的方法和一个 **Render** 方法。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine.Input; // Input needs to be added for gamepad input.
using System.Windows.Forms; // Used for keyboard input
```

```
namespace Engine

{
    public class VerticalMenu
    {
        Vector _position = new Vector();
        Input.Input _input;
        List<Button> _buttons = new List<Button>();
        public double Spacing { get; set; }

        public VerticalMenu(double x, double y, Input.Input input)
        {
            _input = input;
            _position = new Vector(x, y, 0);
            Spacing = 50;
        }

        public void AddButton(Button button)
        {
            double _currentY = _position.Y;

            if (_buttons.Count != 0)
            {
                _currentY = _buttons.Last().Position.Y;
                _currentY -= Spacing;
            }
            else
            {
                // It's the first button added it should have
                // focus
                button.OnGainFocus();
            }

            button.Position = new Vector(_position.X, _currentY, 0);
            _buttons.Add(button);
        }

        public void Render(Renderer renderer)
        {
            _buttons.ForEach(x => x.Render(renderer));
        }
    }
}
```

按钮的位置被自动处理。每次添加按钮时,会把新按钮添加到 Y 轴上其他按钮的下方。**Spacing** 成员决定了按钮之间的距离,默认为 50 像素。菜单本身也有一个位置,允许把按钮作为一个整体四处移动。这个位置只在构造函数中进行设置。在构造 **VerticalMenu** 后,其位置不能改变,因为这需要一个额外的方法在新位置重新排列所有的按钮。这种功能很不错,但是却不是必须具有的。**Render** 方法使用 C#新增的 **lambda** 操作符来渲染所有按钮。

菜单类还不处理用户输入,但是在添加处理输入的代码之前,首先将菜单与 **StartMenuState** 关联起来,以便测试菜单和按钮是否可以正确工作。按钮上的标签将使用与标题文本不同的字体。从本书的配套光盘上找到 **general_font.fnt** 和 **general_font.tga**,把它们添加到项目中。然后需要在 **Form.cs** 文件中设置这个新字体。

```
// In form.cs
private void InitializeTextures()
{
    // Init DevIL
    IL.ilInit();
    ILU.iluInit();
    ILUT.ilutInit();
    ILUT.ilutRenderer(ILUT_OPENGL);

    // Textures are loaded here.
    _textureManager.LoadTexture("title_font", "title_font.tga");
    _textureManager.LoadTexture("general_font", "general_font.tga");
}

Engine.Font _generalFont;
Engine.Font _titleFont;
private void InitializeFonts()
{
    // Fonts are loaded here.
    _titleFont = new Engine.Font(_textureManager.Get("title_font"),
        FontParser.Parse("title_font.fnt"));

    _generalFont = new Engine.Font(_textureManager.Get("general_font"),
        FontParser.Parse("general_font.fnt"));
}
```

这个新的通用字体可以传递给构造函数中的 **StartMenuState**,以构造垂直菜单。此时也会传递 **Input** 类,所以需要把 **using Engine.Input** 语句添加到 **StartMenuState.cs** 文件顶部。

```
Engine.Font _generalFont;
Input _input;
VerticalMenu _menu;
```



```
public StartMenuState(Engine.Font titleFont, Engine.Font generalFont,
Input = input)
{
    _input = input;
    _generalFont = generalFont;
    InitializeMenu();
}
```

实际的菜单创建工作是在 `InitializeMenu` 函数中完成的，这样可以避免 `StartMenuState` 构造函数变得拥挤。`StartMenuState` 创建了一个在 *X* 轴居中、在 *Y* 轴上方 150 像素位置的垂直菜单。这会在标题文本下方以比较整洁的方式放置菜单。

```
private void InitializeMenu()
{
    _menu = new VerticalMenu(0, 150, _input);
    Button startGame = new Button(
        delegate(object o, EventArgs e)
        {
            // Do start game functionality.
        },
        new Text("Start", _generalFont));

    Button exitGame = new Button(
        delegate(object o, EventArgs e)
        {
            // Quit
            System.Windows.Forms.Application.Exit();
        },
        new Text("Exit", _generalFont));

    _menu.AddButton(startGame);
    _menu.AddButton(exitGame);
}
```

这里创建了两个按钮：一个用于退出游戏，一个用于开始游戏。显然，添加其他按钮也并不困难(例如，使用这个系统添加载入游戏、致谢、设置或访问网站等按钮也相当简单)。`Exit` 按钮的委托被完全实现，当调用它时，将退出程序。`Start` 菜单按钮的功能现在还是空的，当创建了游戏主体状态后，将实现相关功能。

现在已经成功地创建了垂直菜单，但是只有添加到渲染循环以后才可以看到它。

```
public void Render()
{
    Gl.glClearColor(1, 1, 1, 0);
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    _renderer.DrawText(_title);
}
```

```
        _menu.Render(_renderer);  
        _renderer.Render();  
    }
```

运行程序，菜单将在标题下显示出来。

现在只剩下输入处理代码还没有实现。菜单将通过游戏手柄的左控制杆或键盘进行导航。需要一些额外的逻辑来判断控制杆何时被摇上或者摇下。如下所示为 `VerticalMenu` 类的 `HandleInput` 类，其中显示了这种逻辑。可能还需要修改 `Input` 类，使 `Controller` 成员变为公有，这样就可以从 `Engine` 项目以外访问它了。

```
bool _inDown = false;  
bool _inUp = false;  
int _currentFocus = 0;  
public void HandleInput()  
{  
    bool controlPadDown = false;  
    bool controlPadUp = false;  
  
    float invertY = _input.Controller.LeftControlStick.Y * -1;  
  
    if (invertY < -0.2)  
    {  
        // The control stick is pulled down  
        if (_inDown == false)  
        {  
            controlPadDown = true;  
            _inDown = true;  
        }  
    }  
    else  
    {  
        _inDown = false;  
    }  
  
    if (invertY > 0.2)  
    {  
        if (_inUp == false)  
        {  
            controlPadUp = true;  
            _inUp = true;  
        }  
    }  
    else
```

```
{
    _inUp = false;
}

if (_input.Keyboard.IsKeyPressed(Keys.Down)
    || controlPadDown)
{
    OnDown();
}
else if(_input.Keyboard.IsKeyPressed(Keys.Up)
    || controlPadUp)
{
    OnUp();
}
}
```

需要在 `StartMenuState.Update` 方法中调用 `HandleInput` 函数。如果没有添加这个调用，将无法检测到任何输入。`HandleInput` 检测到与垂直菜单有关的特定输入，然后调用其他函数来处理输入。现在只有两个函数 `OnUp` 和 `OnDown`，它们将修改当前拥有焦点的菜单项。

```
private void OnUp()
{
    int oldFocus = _currentFocus;
    _currentFocus++;
    if (_currentFocus == _buttons.Count)
    {
        _currentFocus = 0;
    }
    ChangeFocus(oldFocus, _currentFocus);
}

private void OnDown()
{
    int oldFocus = _currentFocus;
    _currentFocus--;
    if (_currentFocus == -1)
    {
        _currentFocus = (_buttons.Count - 1);
    }
    ChangeFocus(oldFocus, _currentFocus);
}

private void ChangeFocus(int from, int to)
{
}
```

```

        if (from != to)
        {
            _buttons[from].OnLoseFocus();
            _buttons[to].OnGainFocus();
        }
    }
}

```

通过按键盘上的上下方向键，焦点会在垂直菜单的按钮之间上下移动。焦点还会发生环绕。如果对菜单最顶部的按钮按上方向键，焦点将转移到菜单底部的按钮。`ChangeFocus`方法减少了重复的代码，它告诉一个按钮已丢失焦点，并告诉另外一个按钮已获得焦点。

现在可以选择按钮，但是还没有代码来处理按钮被按下的情况。修改 `VerticalMenu` 类，以检测到手柄上的 A 按键或键盘上的 `Enter` 键何时被按下。检测到这些键被按下，将调用当前选中的按钮的委托。

```

// Inside the HandleInput function
else if (_input.Keyboard.IsKeyPressed(Keys.Up)
        || controlPadUp)
{
    OnUp();
}
else if (_input.Keyboard.IsKeyPressed(Keys.Enter)
        || _input.Controller.ButtonA.Pressed)
{
    OnButtonPress();
}
}

private void OnButtonPress()
{
    _buttons[_currentFocus].OnPress();
}

```

运行代码，并使用键盘或手柄来导航菜单。按下 `Exit` 按钮将退出游戏，但是按下 `Start` 按钮现在不会发生任何操作。`Start` 按钮需要将状态修改为游戏主体状态，这意味着 `StartMenuState` 需要能够访问状态系统。

```

private void InitializeGameState()
{
    _system.AddState("start_menu", new StartMenuState(_titleFont,
        _generalFont, _input, _system));
}

```

还需要修改 `StartMenuState` 构造函数，它将保存对状态系统的引用。

```

StateSystem _system;
public StartMenuState(Engine.Font titleFont, Engine.Font generalFont,

```

```
Input input, StateSystem system)
{
    _system = system;
```

Start 按钮可以使用这些代码在自己被按下时修改状态。**Start** 按钮在 **InitializeMenu** 方法中建立, 需要对其做如下修改。

```
Button startGame = new Button(
    delegate(object o, EventArgs e)
    {
        _system.ChangeState("inner_game");
    },
    new Text("Start", _generalFont));
```

inner_game 状态还不存在, 但是马上我们就会开发该状态。对于第一遍编码, 现在的 **Start** 菜单已经完整了。运行程序得到的结果如图 10-5 所示。

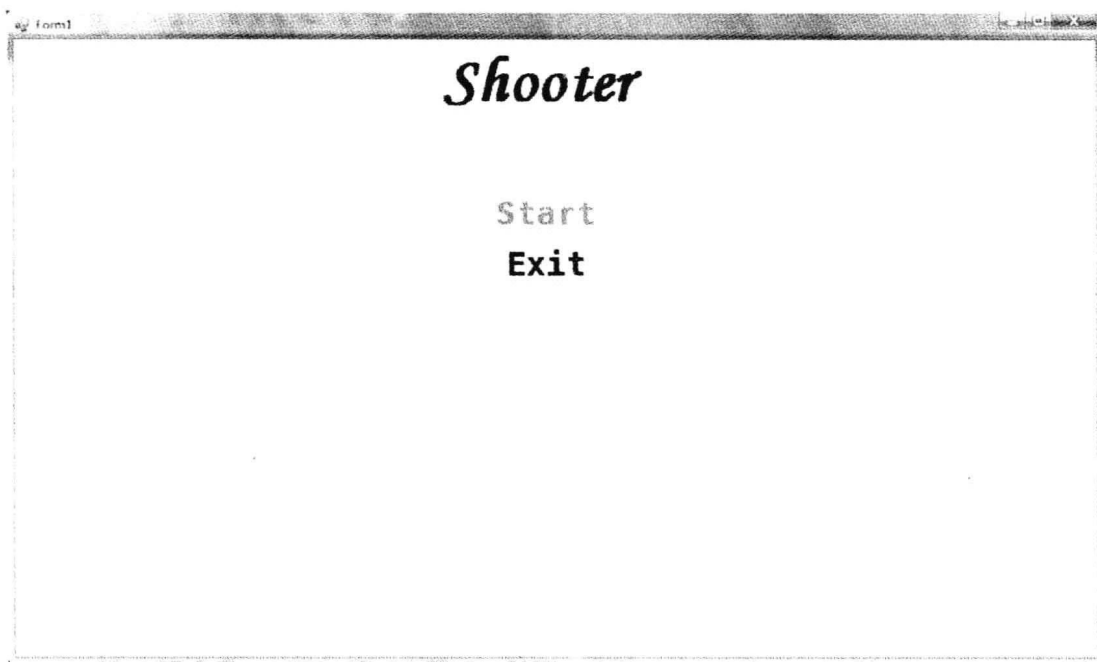


图 10-5 第一遍编码中的 **Start** 游戏菜单

在后面进行编码时可以根据需要修改这个菜单, 添加更多的动画、演示等。

10.2.2 游戏主体状态

对于第一遍编码, 游戏主体将尽可能简单。它会等待几秒, 然后变为游戏结束状态。游戏主体状态。还需要向游戏结束状态传递一些信息, 以报告玩家是获胜还是失败。

需要使用一个 **PersistentGameData** 类来存储与玩家有关的信息, 包括获胜还是失败的信

息。最终，游戏主体将允许玩家玩一个射击游戏，但是在第一遍编码中不提供这种功能。

游戏主体的关卡将持续固定的时间，如果到了指定的时间后玩家依然存活，那么玩家获胜。关卡需要的时间通过 `LevelDescription` 类描述。就现在而言，这个类只包含关卡的持续时间。

```
class LevelDescription
{
    // Time a level lasts in seconds.
    public double Time { get; set; }
}
```

`PersistentGameData` 类中将包含对当前关卡的描述，以及玩家在该关卡中是否获胜的信息。

```
class PersistentGameData
{
    public bool JustWon { get; set; }
    public LevelDescription CurrentLevel { get; set; }
    public PersistentGameData()
    {
        JustWon = false;
    }
}
```

`JustWon` 成员在构造函数中被设为 `false`，因为玩家不能在还未创建游戏数据的时候就获胜。在 `Form.cs` 文件中创建 `PersistentGameData` 类。添加一个将从构造函数中调用的新函数 `InitializeGameData`，它应该在调用 `InitializeTextures` 之后、创建游戏字体之前调用。

```
PersistentGameData _persistentGameData = new PersistentGameData();
private void InitializeGameData()
{
    LevelDescription level = new LevelDescription();
    level.Time = 1; // level only lasts for a second
    _persistentGameData.CurrentLevel = level;
}
```

建立这个类之后，设计 `InnerGameState` 就会变得很容易。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine;
using Engine.Input;
using Tao.OpenGl;
```

```
namespace Shooter
{
    class InnerGameState : IGameObject
    {
        Renderer _renderer = new Renderer();
        Input _input;
        StateSystem _system;
        PersistantGameData _gameData;
        Font _generalFont;

        double _gameTime;

        public InnerGameState(StateSystem system, Input input, PersistantGameData
gameData, Font generalFont)
        {
            _input = input;
            _system = system;
            _gameData = gameData;
            _generalFont = generalFont;
            OnGameStart();
        }

        public void OnGameStart()
        {
            _gameTime = _gameData.CurrentLevel.Time;
        }

        #region IGameObject Members

        public void Update(double elapsedTime)
        {
            _gameTime -= elapsedTime;

            if (_gameTime <= 0)
            {
                OnGameStart();
                _gameData.JustWon = true;
                _system.ChangeState("game_over");
            }
        }

        public void Render()
```

```

    {
        Gl.glClearColor(1, 0, 1, 0);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        _renderer.Render();
    }
    #endregion
}
}

```

构造函数接受 `StateSystem` 和 `PersistentGameData` 作为参数。通过使用这些类, `InnerGameState` 可以确定游戏何时结束, 并修改游戏状态。构造函数还接受输入和通用字体, 因为在第二遍编码中添加功能时它们会很有用。构造函数调用 `OnGameStart`, 该方法设置的变量 `gameTime` 确定了游戏关卡的持续时间。现在还没有关卡内容, 所以时间被设为 1s。

`Update` 函数对关卡时间进行倒计时。时间结束后, 状态将被修改为 `game_over`。 `gameTime` 通过调用 `OnGameState` 重设, 因为此时玩家仍然存活, `gameData` 对象的 `JustWon` 标志被设为 `true`。游戏主体状态的 `Render` 函数将屏幕清除为粉红色, 所以状态变化的时间变得很明显。

应该使用 `InnerGameState` 类向 `Form.cs` 文件的状态系统中添加另外一个状态。

```

_system.AddState("inner_game", new InnerGameState(_system, _input,
_persistentGameData, _generalFont));

```

游戏主体的第一遍编码到此结束。

10.2.3 游戏结束状态

游戏结束状态是一个简单的状态, 告诉玩家游戏已经结束, 他获胜或者失败了。这个状态通过使用 `PersistentGameData` 类确定玩家是否获胜。它会在短暂的时间内显示这些信息, 然后使玩家回到开始菜单。玩家可以通过提前按下按钮来强制 `GameOverState` 结束并返回到开始菜单。

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine;
using Engine.Input;
using Tao.OpenGl;

namespace Shooter
{
    class GameOverState : IGameObject
    {
        const double _timeOut = 4;
    }
}

```



```

double _countDown = _timeOut;

•   StateSystem _system;
    Input _input;
    Font _generalFont;
    Font _titleFont;
    PersistantGameData _gameData;
    Renderer _renderer = new Renderer();

    Text _titleWin;
    Text _blurbWin;

    Text _titleLose;
    Text _blurbLose;

    public GameOverState(PersistantGameData data, StateSystem system,
Input input, Font generalFont, Font titleFont)
    {
        _gameData = data;
        _system = system;
        _input = input;
        _generalFont = generalFont;
        _titleFont = titleFont;

        _titleWin = new Text("Complete!", _titleFont);
        _blurbWin = new Text("Congratulations, you won!", _generalFont);
        _titleLose = new Text("Game Over!", _titleFont);
        _blurbLose = new Text("Please try again...", _generalFont);

        FormatText(_titleWin, 300);
        FormatText(_blurbWin, 200);

        FormatText(_titleLose, 300);
        FormatText(_blurbLose, 200);
    }

    private void FormatText(Text _text, int yPosition)
    {
        _text.SetPosition(-_text.Width / 2, yPosition);
        _text.SetColor(new Color(0, 0, 0, 1));
    }

    #region IGameObject Members

```

```

public void Update(double elapsedTime)
{
    _countDown -= elapsedTime;
    if ( _countDown <= 0 ||
        _input.Controller.ButtonA.Pressed ||

        _input.Keyboard.IsKeyPressed(System.Windows.Forms.Keys.
Enter))
    {
        Finish();
    }
}

private void Finish()
{
    _gameData.JustWon = false;
    _system.ChangeState("start_menu");
    _countDown = _timeOut;
}

public void Render()
{
    Gl.glClearColor(1, 1, 1, 0);
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
    if (_gameData.JustWon)
    {
        _renderers.DrawText(_titleWin);
        _renderers.DrawText(_blurbWin);
    }
    else
    {
        _renderers.DrawText(_titleLose);
        _renderers.DrawText(_blurbLose);
    }
    _renderers.Render();
}
#endregion
}
}

```

需要像 form.cs 类中的其他类一样加载这个类。

```
private void InitializeGameState()
```

```
{  
    // Game states are loaded here  
    _system.AddState("start_menu", new StartMenuState(_titleFont,  
_generalFont, _input, _system));  
    _system.AddState("inner_game", new InnerGameState(_system, _input,  
_persistantGameData, _generalFont));  
    _system.AddState("game_over", new GameOverState(_persistantGameData,  
_system, _input, _generalFont, _titleFont));  
    _system.ChangeState("start_menu");  
}
```

GameOverState 为玩家获胜和失败的情况分别创建了一个标题和一条消息。然后它使用 **JustWon** 成员来决定显示哪条消息。**GameOverState** 还有一个计数器，最终这个状态将会结束，并让用户回到开始菜单。

创建了这 3 个状态后，游戏的第一遍编码也就完成了。虽然这个游戏没什么意思，但是它已经是一个完整的游戏。第 10.3 节将为游戏主体添加更多细节，并细化整体结构，使其看上去比这里更好一些。

10.3 开发游戏主体

现在的游戏主体不允许交互，并且在几秒之后就会结束。为了使游戏主体状态更像是一个游戏，需要引入一个 **PlayerCharacter**，并允许玩家四处移动这个角色。这是第一个目标。创建游戏时，逐个实现一系列较小的、定义良好的目标是很重要的，这样代码编写起来就更加简单。在这个游戏中，**PlayerCharacter** 是某种类型的飞船。

实现第一个目标后，需要让玩家感觉他正在关卡中前进。这是通过卷动背景纹理实现的。下一个小目标是允许玩家发射子弹。子弹需要向某个目标射击，所以还需要添加敌人。每个目标都是一个小步，可合理地引出下一步，以这种方式可以快速构建出一个游戏。

10.3.1 移动玩家角色

玩家由使用精灵和纹理创建的飞船表示。飞船由方向键或游戏手柄上的左控制杆控制。

控制 **PlayerCharacter** 的代码不会直接放在 **InnerGameState** 类中。**InnerGameState** 类应该是一个轻量级的、易于理解的类。关卡代码的主要部分将存储在 **Level** 类中。每次玩家玩一个关卡时，都会创建一个新的关卡对象来替换原来的关卡对象。每次创建新的关卡对象确保了不会发生以前玩关卡时遗留的数据引起奇怪的错误的情况。

```
using System;  
using System.Collections.Generic;  
using System.Linq;
```

```
using System.Text;
using Engine;
using Engine.Input;
using System.Windows.Forms;
using System.Drawing;

namespace Shooter
{
    class Level
    {
        Input _input;
        PersistentGameData _gameData;
        PlayerCharacter _playerCharacter;
        TextureManager _textureManager;

        public Level(Input input, TextureManager textureManager,
            PersistentGameData gameData)
        {
            _input = input;
            _gameData = gameData;
            _textureManager = textureManager;
            _playerCharacter = new PlayerCharacter(_textureManager);
        }

        public void Update(double elapsedTime)
        {
            // Get controls and apply to player character
        }

        public void Render(Renderer renderer)
        {
            _playerCharacter.Render(renderer);
        }
    }
}
```

这段代码描述了一个关卡,它在构造函数中接受 `Input` 和 `PersistentGameData` 作为参数。`Input` 对象用于移动 `PlayerCharacter`。`PersistentGameData` 可以用于记录得分或者其他应该在多个关卡中记录的数据。纹理管理器用于创建玩家、敌人和背景精灵。

`PlayerCharacter` 类将包含代表玩家飞船的精灵。本书配套光盘的 `Assets` 目录中包含一个 `spaceship.tga` 精灵。需要把这个精灵添加到项目中,并修改其属性,以便它会被复制到生成目录中。在 `form.cs` 的 `InitializeTextures` 方法中加载这个纹理。

```
private void InitializeTextures()
{
    // Init DevIL
    Il.ilInit();
    Ilu.iluInit();
    Ilut.ilutInit();
    Ilut.ilutRenderer(Ilut.ILUT_OPENGL);

    _textureManager.LoadTexture("player_ship", "spaceship.tga");
```

现在已经把玩家精灵加载到了 `TextureManager` 中，可以编写 `PlayerCharacter` 类了。

```
public class PlayerCharacter
{
    Sprite _spaceship    new Sprite();

    public PlayerCharacter(TextureManager textureManager)
    {
        _spaceship.Texture = textureManager.Get("player_ship");
        _spaceship.SetScale(0.5, 0.5); // spaceship is quite big, scale
it down.
    }
    public void Render(Renderer renderer)
    {
        renderer.DrawSprite(_spaceship);
    }
}
```

现在的 `PlayerCharacter` 类只渲染出飞船。为了实际看到飞船，需要在 `InnerGameState` 中创建一个 `Level` 对象，并将其与 `Update` 方法和 `Render` 方法关联起来。`Level` 类有自己的 `Render` 和 `Update` 方法，所以添加到游戏主体状态中十分方便。`Level` 类使用 `TextureManager` 类，这意味着必须修改 `InnerGameState` 的构造函数，使其接受一个 `TextureManager` 对象。在 `form.cs` 文件中，需要把 `textureManager` 对象传递给 `InnerGameState` 构造函数。

```
class InnerGameState : IGameObject
{
    Level _level;
    TextureManager _textureManager;
    // Code omitted

    public InnerGameState( StateSystem system, Input input, TextureManager
textureManager,
        PersistantGameData gameData, Font generalFont)
```

```
{
    _textureManager = textureManager;
// Code omitted

    public void OnGameStart()
    {
        _level = new Level(_input, _textureManager, _gameData);
        _gameTime = _gameData.CurrentLevel.Time;
    }

// Code omitted
    public void Update(double elapsedTime)
    {
        _level.Update(elapsedTime);

// Code omitted
    public void Render()
    {
        Gl.glClearColor(1, 0, 1, 0);
        Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);
        _level.Render(_renderer);
    }
}
```

运行代码并启动游戏。飞船将会闪现,然后游戏突然结束。为了充分测试 **InnerGameState**, 必须增加关卡的长度。关卡长度在 **form.cs** 文件的 **InitializeGameData** 函数中设置。找到相关代码,增加关卡长度,30s 即可。

飞船的移动是非常简单的,没有加速或物理建模。控制杆和方向键直接映射到飞船的移动上。**PlayerCharacter** 类需要一个新的 **Move** 方法。

```
double _speed = 512; // pixels per second
public void Move(Vector amount)
{
    amount *= _speed;
    _spaceship.SetPosition(_spaceship.GetPosition() + amount);
}
```

Move 方法接受一个指定飞船的移动方向和移动量的向量作为参数。该参数将与 **speed** 值相乘,以增加移动量。然后这个新向量与飞船的当前位置相加,得出飞船在太空中的新位置,再将飞船精灵移动到这个位置。街机游戏中的所有基本移动都是这样完成的。通过对更加符合物理原理的系统进行建模,例如加速和摩擦,移动可以产生真实的效果,但我们还是坚持使用基本的移动代码。

飞船根据 **Input** 类中的值四处移动,这是在 **Level** 的 **Update** 循环中处理的。

```
public void Update(double elapsedTime)
{
    // Get controls and apply to player character
    double _x = _input.Controller.LeftControlStick.X;
    double _y = _input.Controller.LeftControlStick.Y * - 1;
    Vector controlInput = new Vector(_x, _y, 0);

    if (Math.Abs(controlInput.Length()) < 0.0001)
    {
        // If the input is very small, then the player may not be using
        // a controller; he might be using the keyboard.
        if (_input.Keyboard.IsKeyHeld(Keys.Left))
        {
            controlInput.X = -1;
        }

        if (_input.Keyboard.IsKeyHeld(Keys.Right))
        {
            controlInput.X = 1;
        }

        if (_input.Keyboard.IsKeyHeld(Keys.Up))
        {
            controlInput.Y = 1;
        }

        if (_input.Keyboard.IsKeyHeld(Keys.Down))
        {
            controlInput.Y = -1;
        }
    }

    _playerCharacter.Move(controlInput * elapsedTime);
}
```

游戏手柄的控制代码十分简单。创建一个描述控制杆的摇动的向量(Y轴的值通过乘以-1取反,所以向上推控制杆时,飞船将向上飞,而不是向下飞)。这个向量与经过的时间相乘,这样不管帧率如何,移动都是固定的。被缩放后的这个向量将用于移动飞船。代码中还提供了对键盘的支持。先检查控制杆的值,如果控制杆没有移动,则检查键盘的按键。代码假定如果玩家没有移动控制杆,可能就在使用键盘玩游戏。键盘没有控制杆那么细致,它只能将上、下、左、右表示为0或1的绝对值。键盘输入用于创建一个向量,以便可以

像处理控制杆输入一样处理键盘输入。使用 `IsKeyHeld` 而不是 `IsKeyPressed`，因为假定当用户按住左方向键时，他希望持续向左移动，而不是移动一次就停止。

运行程序，现在可以在屏幕上四处移动飞船。移动的目标完成了。

10.3.2 使用卷动背景模拟移动

添加背景很简单，本书配套光盘的 `Assets` 目录中有两个基本的星空纹理，分别是 `background.tga` 和 `background_p.tga`。将这两个文件添加到解决方案中，并修改它们的属性，以便像前面的其他纹理那样把它们复制到生成目录中。然后在 `form.cs` 的 `InitializeTextures` 函数中把它们加载到纹理管理器中。

```
_textureManager.LoadTexture("background", "background.tga");  
_textureManager.LoadTexture("background_layer_1",  
    "background_p.tga");
```

选择的这两个背景可以层叠，产生比只使用一个纹理更加有趣的效果。

背景将通过使用 UV 卷动(UV scrolling)运动起来。这可以通过创建一个新类 `ScrollingBackground` 实现。可以重用这个卷动背景类，使开始和游戏结束菜单更加有趣，但是目前最要紧的是游戏主体。

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using Engine;  
  
namespace Shooter  
{  
    class ScrollingBackground  
    {  
        Sprite _background = new Sprite();  
  
        public float Speed { get; set; }  
        public Vector Direction { get; set; }  
        Point _topLeft = new Point(0, 0);  
        Point _bottomRight = new Point(1, 1);  
  
        public void SetScale(double x, double y)  
        {  
            _background.SetScale(x, y);  
        }  
    }  
}
```



```

public ScrollingBackground(Texture background)
{
    _background.Texture = background;
    Speed = 0.15f;
    Direction = new Vector(1, 0, 0);
}

public void Update(float elapsedTime)
{
    _background.SetUVs(_topLeft, _bottomRight);
    _topLeft.X += (float)(0.15f * Direction.X * elapsedTime);
    _bottomRight.X += (float)(0.15f * Direction.X * elapsedTime);
    _topLeft.Y += (float)(0.15f * Direction.Y * elapsedTime);
    _bottomRight.Y += (float)(0.15f * Direction.Y * elapsedTime);
}

public void Render(Renderer renderer)
{
    renderer.DrawSprite(_background);
}
}
}

```

关于滚动背景类，值得注意的是它有一个方向向量 **Direction**。该向量可以修改滚动的方向。在普通的射击游戏中，背景通常从右向左滚动。修改滚动方向可以使背景在任意期望的方向上滚动。在太空探险游戏中，背景朝着与玩家的移动相反的方向移动，此时这种技术很有用。

滚动类还有一个 **Speed** 成员，严格来说它并不是必需的，因为滚动的速度可以编码为向量的大小，但是把速度分离出来以后，修改起来更加方便。**Direction** 和 **Speed** 用来在 **Update** 方法中移动顶点的 (U, V) 数据。

现在可以把这个背景添加到 **Level** 类中。

```

ScrollingBackground _background;
ScrollingBackground _backgroundLayer;

public Level(Input input, TextureManager textureManager, PersistentGameData
gameData)
{
    _input = input;
    _gameData = gameData;
    _textureManager = textureManager;
}

```

```
_background = new ScrollingBackground(textureManager.Get  
("background"));  
_background.SetScale(2, 2);  
_background.Speed = 0.15f;  
  
_backgroundLayer = new ScrollingBackground(textureManager.Get  
("background_layer_1"));  
_backgroundLayer.Speed = 0.1f;  
_backgroundLayer.SetScale(2.0, 2.0);
```

这两个背景对象在构造函数中创建，它们分别被放大一倍。之所以放大背景，是因为纹理大约是屏幕区域大小的一半，放大纹理可以保证纹理足以覆盖整个游戏区域，不会在边缘位置留下空白。

两个背景以不同的速度卷动，产生了所谓的视差效果。人脑通过一些被称作深度线索(depth cue)的不同线索来理解 3D 世界。例如，每只眼睛看到的角度是不同的，视野的差异可以用来确定第三个维度，这被称为双眼线索(binocular cue)。

视差就是一种线索。简单来说，离观察者越远的对象看上去移动得越慢。假设你自己在驱车远行，在目光尽处有一片连绵的大山。大山看上去不怎么移动，但是公路两旁的树木却飞一样的被抛在身后。这是一种深度线索，大脑知道山在很远的地方。

模拟视差很容易。快速卷动的星空中的星星看上去离飞船很近，而移动较慢的背景中的星星看上去离飞船较远。背景类只需要以不同的速度移动，这样就使背景产生了深度感。

由于需要渲染和更新背景对象，还需要继续修改代码。

```
public void Update(double elapsedTime)  
{  
    _background.Update((float)elapsedTime);  
    _backgroundLayer.Update((float)elapsedTime);  
  
    // A little later in the code  
  
public void Render(Renderer renderer)  
{  
    _background.Render(renderer);  
    _backgroundLayer.Render(renderer);
```

运行代码，查看视差效果，如图 10-6 所示。对于本书提供的星空，不太容易注意到这种效果，所以你可以自由修改图片，或者再添加几个层。

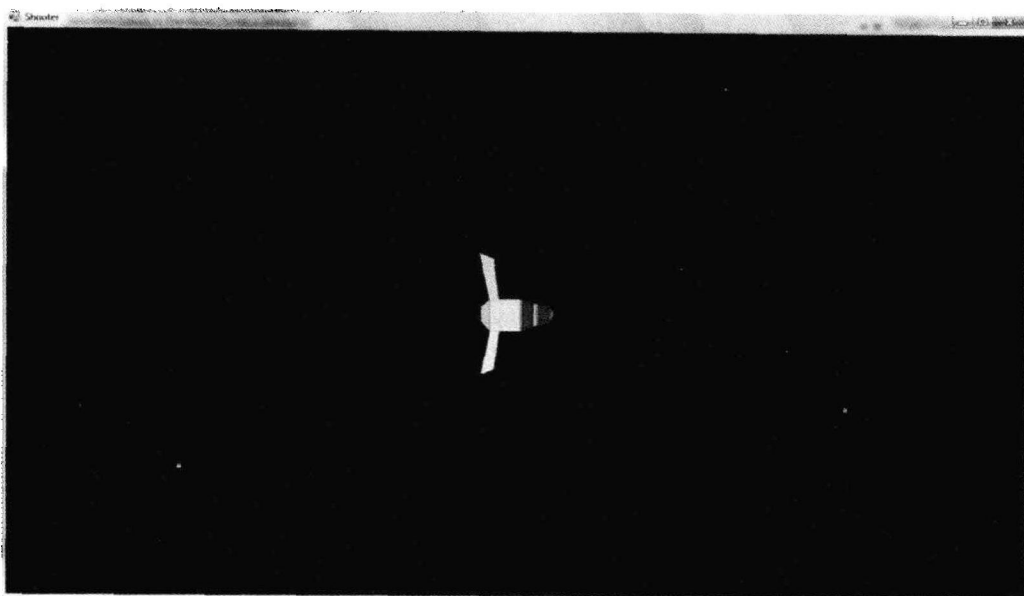


图 10-6 卷动背景

现在飞船看上去在太空中越变越大或越变越小，看上去有点像真正的游戏了。下一个任务是添加敌人。

10.3.3 添加一些简单的敌人

敌人将使用精灵来表示，所以它们将使用 `Sprite` 类。敌人的精灵应该与玩家精灵不同，所以添加本书配套光盘的 `Assets` 目录中的新精灵纹理 `spaceship2.tga`。修改其属性，以便当生成游戏时它被复制到 `bin` 目录中。

如下代码将纹理加载到纹理管理器中。

```
_textureManager.LoadTexture("enemy_ship", "spaceship2.tga");
```

添加了这个纹理后，就可以构造一个类来简单地表示敌人。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine;

namespace Shooter
{
    class Enemy
    {
        Sprite _spaceship = new Sprite();
        double _scale = 0.3;
    }
}
```

```

public Enemy(TextureManager textureManager)
{
    _spaceship.Texture = textureManager.Get("enemy_ship");
    _spaceship.SetScale(_scale, _scale);
    _spaceship.SetRotation(Math.PI); // make it face the player
    _spaceship.SetPosition(200, 0); // put it somewhere easy to see
}

public void Update(double elapsedTime)
{
}

public void Render(Renderer renderer)
{
    renderer.DrawSprite(_spaceship);
}

}
}

```

敌人将由 **Level** 类渲染和控制。很可能同时出现多个敌人，所以最好创建一个敌人列表。

```

class Level
{
    List<Enemy> _enemyList = new List<Enemy>();

    // A little later in the code

    public Level(Input input, TextureManager textureManager, PersistantGameData
gameData)
    {
        _input = input;
        _gameData = gameData;
        _textureManager = textureManager;
        _enemyList.Add(new Enemy(_textureManager));

        // A little later in the code

        public void Update(double elapsedTime)
        {
            _background.Update((float)elapsedTime);
            _backgroundLayer.Update((float)elapsedTime);
            _enemyList.ForEach(x => x.Update(elapsedTime));
        }
    }
}

```

```
// A little later in the code
public void Render(Renderer renderer)
{
    _background.Render(renderer);
    _backgroundLayer.Render(renderer);
    _enemyList.ForEach(x => x.Render(renderer));
}
```

代码很标准。创建敌人列表后，向其中添加了一个敌人。然后使用 `lambda` 语法更新并渲染列表。现在运行程序，应该看到两个飞船：玩家飞船和面对玩家飞船的敌人。在当前的代码中，玩家可以飞过敌人飞船，而没有任何反应。如果玩家撞到了敌人的飞船，应该遭受损伤，或者游戏状态应该变为游戏结束状态。但是在发生这种行为前，需要检测碰撞。

这里的碰撞检测就是本书前面探讨的矩形-矩形碰撞。在对碰撞进行编码之前，绘制出敌人飞船的包围框会很有用。通过使用 `OpenGL` 的立即模式和 `GL_LINE_LOOP`，这并不困难。将下面的代码添加到 `Enemy` 类中。

```
public RectangleF GetBoundingBox()
{
    float width = (float)(_spaceship.Texture.Width * _scale);
    float height = (float)(_spaceship.Texture.Height * _scale);
    return new RectangleF( (float)_spaceship.GetPosition().X - width / 2,
                          (float)_spaceship.GetPosition().Y - height / 2,
                          width, height);
}

// Render a bounding box
public void Render_Debug()
{
    Gl.glDisable(Gl.GL_TEXTURE_2D);

    RectangleF bounds = GetBoundingBox();
    Gl.glBegin(Gl.GL_LINE_LOOP);
    {
        Gl.glColor3f(1, 0, 0);
        Gl.glVertex2f(bounds.Left, bounds.Top);
        Gl.glVertex2f(bounds.Right, bounds.Top);
        Gl.glVertex2f(bounds.Right, bounds.Bottom);
        Gl.glVertex2f(bounds.Left, bounds.Bottom);
    }
    Gl.glEnd();
    Gl.glEnable(Gl.GL_TEXTURE_2D);
}
```

这里使用了 `C#` 的 `RectangleF` 类，因此，需要在 `Enemy.cs` 文件的顶部添加 `using System.Drawing`

库。函数 `GetBoundingBox` 使用精灵来计算出其周围的包围框。包围框的宽度和高度根据精灵进行缩放，所以即使精灵被缩放，包围框也是正确的。`RectangleF` 构造函数接受左上角的 x 和 y 位置，以及矩形的宽度和高度作为参数。精灵的位置就是其中心的位置，所以为了获得左上角的坐标，必须从其位置减去一半的宽度和高度。

`Render_Debug` 方法在精灵旁边绘制了一个红色的方框。应该从 `Enemy.Render` 方法中调用这个 `Render_Debug` 方法。在不需要它时，任何时候都可以删除这个调试函数。

```
public void Render(Renderer renderer)
{
    renderer.DrawSprite(_spaceship);
    Render_Debug();
}
```

运行代码后，敌人的周围将显示一个红色的方框，如图 10-7 所示。可视的调试例程是理解代码功能的一种极佳的方式。

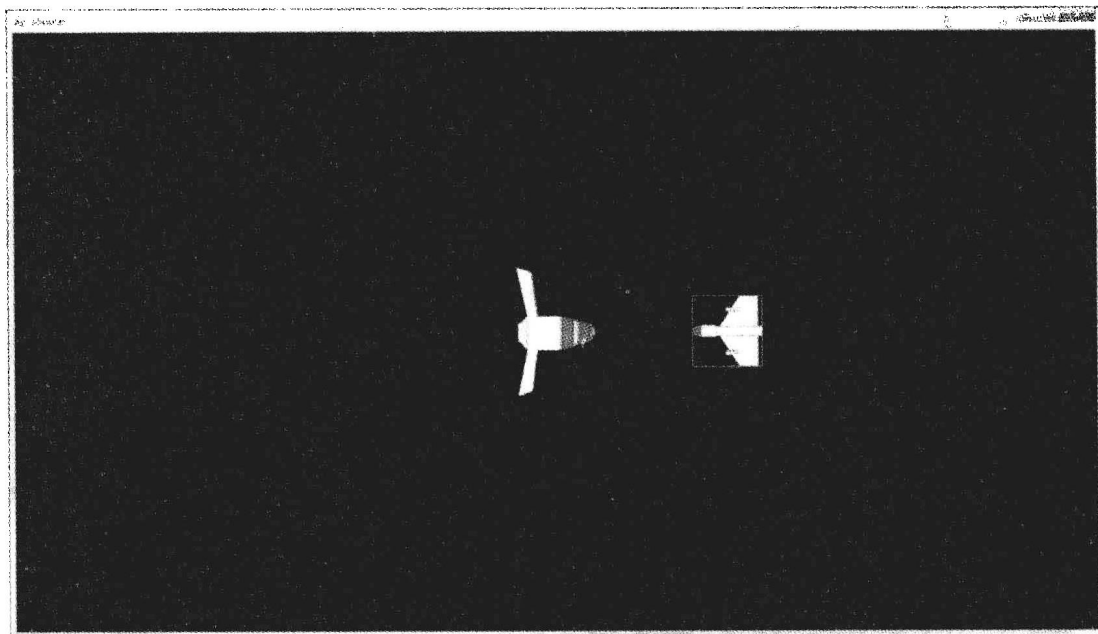


图 10-7 敌人包围框

`GetBoundingBox` 函数可以用来确定敌人是否与其他某个对象发生碰撞。现在，玩家飞船没有 `GetBoundingBox` 函数，而 DRY(Don't Repeat Yourself)原则意味着不应该简单复制这段代码。相反，需要创建一个新的父类来把这种功能集中到一起，然后 `Enemy` 和 `PlayerCharacter` 都可以从该父类继承。

在使 `Enemy` 和 `PlayerCharacter` 类一般化之前，需要修改这个 `Sprite` 类。为了简化包围框绘制函数，精灵中应该包含一些方法来报告当前的缩放比例。

```
public class Sprite
```

```
{
    double _scaleX = 1;
    double _scaleY = 1; public double ScaleX
    {
        get
        {
            return _scaleX;
        }
    }

    public double ScaleY
    {
        get
        {
            return _scaleY;
        }
    }
}
```

修改 **Sprite** 类是对 **Engine** 库的修改，对这种修改不应掉以轻心。这里的修改是一个很有益的修改，将来任何使用 **Engine** 库的项目都会受益。更新 **Sprite** 方法后，可以在 **Shooter** 项目中创建出 **Entity** 类。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine;
using Tao.OpenGl;
using System.Drawing;
namespace Shooter
{
    public class Entity
    {
        protected Sprite _sprite = new Sprite();

        public RectangleF GetBoundingBox()
        {
            float width = (float)(_sprite.Texture.Width * _sprite.ScaleX);
            float height = (float)(_sprite.Texture.Height * _sprite.ScaleY);
            return new RectangleF((float)_sprite.GetPosition().X - width / 2,
                (float)_sprite.GetPosition().Y - height / 2,
                width, height);
        }
    }
}
```

```

// Render a bounding box
protected void Render_Debug()
{
    Gl.glDisable(Gl.GL_TEXTURE_2D);

    RectangleF bounds = GetBoundingBox();
    Gl.glBegin(Gl.GL_LINE_LOOP);
    {
        Gl.glColor3f(1, 0, 0);
        Gl.glVertex2f(bounds.Left, bounds.Top);
        Gl.glVertex2f(bounds.Right, bounds.Top);
        Gl.glVertex2f(bounds.Right, bounds.Bottom);
        Gl.glVertex2f(bounds.Left, bounds.Bottom);
    }
    Gl.glEnd();
    Gl.glEnable(Gl.GL_TEXTURE_2D);
}

}
}

```

Entity 类包含一个精灵，以及渲染该精灵的包围框的一些代码。定义了 Entity 以后，Enemy 类可以得到显著简化。

```

public class Enemy : Entity
{
    double _scale = 0.3;
    public Enemy(TextureManager textureManager)
    {
        _sprite.Texture = textureManager.Get("enemy_ship");
        _sprite.SetScale(_scale, _scale);
        _sprite.SetRotation(Math.PI); // make it face the player
        _sprite.SetPosition(200, 0); // put it somewhere easy to see
    }

    public void Update(double elapsedTime)
    {
    }

    public void Render(Renderer renderer)
    {
        renderer.DrawSprite(_sprite);
        Render_Debug();
    }
}

```



```
public void SetPosition(Vector position)
{
    _sprite.SetPosition(position);
}
}
```

现在 **Enemy** 也是一个 **Entity**，不再需要自己对精灵的引用。对 **PlayerCharacter** 类可以应用相同的重构。

```
public class PlayerCharacter : Entity
{
    double _speed = 512; // pixels per second

    public void Move(Vector amount)
    {
        amount *= _speed;
        _sprite.SetPosition(_sprite.GetPosition() + amount);
    }

    public PlayerCharacter(TextureManager textureManager)
    {
        _sprite.Texture = textureManager.Get("player_ship");
        _sprite.SetScale(0.5, 0.5); // spaceship is quite big, scale it down.
    }

    public void Render(Renderer renderer)
    {
        Render_Debug();
        renderer.DrawSprite(_sprite);
    }
}
```

再次运行代码，现在敌人和玩家的周围都有了合适的包围框。

现在的规则是，如果 **PlayerCharacter** 击中一个敌人，游戏结束。以后可以通过给敌人增加生命值来细化游戏。为了尽快得到可以工作的游戏，现在选择一击致命。

首先需要做出的修改发生在 **InnerGameState** 中，它需要能够确定玩家角色何时死亡，此时当前关卡无法完成。

```
public void Update(double elapsedTime)
{
    _level.Update(elapsedTime);
    _gameTime -= elapsedTime;
}
```

```
        if (_gameTime <= 0)
        {
            OnGameStart();
            _gameData.JustWon = true;
            _system.ChangeState("game_over");
        }

        if (_level.HasPlayerDied())
        {
            OnGameStart();
            _gameData.JustWon = false;
            _system.ChangeState("game_over");
        }
    }
}
```

这里在 **Level** 类中新添加了一个额外的函数 **HasPlayerDied**，用于报告玩家角色是否死亡。这里在 **gameTime** 后检查玩家角色是否死亡，意味着如果时间到了，但是玩家在最后一刻死亡，他仍然无法赢得关卡。

在 **Level** 类中，需要实现 **HasPlayerDied** 方法。它只是 **PlayerCharacter** 的当前状态的一个简单包装器。

```
public bool HasPlayerDied()
{
    return _playerCharacter.IsDead;
}
```

死亡标志包含在 **PlayerCharacter** 类中。

```
bool _dead = false;
public bool IsDead
{
    get
    {
        return _dead;
    }
}
```

当玩家与敌人发生碰撞时，可以设置死亡标志，此时游戏将会结束，玩家输掉了关卡。关卡还需要一些代码来处理敌人飞船与玩家飞船发生的碰撞。这种碰撞处理是在 **Level** 类中完成的，该类可以访问 **PlayerCharacter** 和敌人列表。

```
private void UpdateCollisions
()
{
    foreach (Enemy enemy in _enemyList)
```

```
{
    if (enemy.GetBoundingBox().Intersects(_playerCharacter.
GetBoundingBox()))
    {
        enemy.OnCollision(_playerCharacter);
        _playerCharacter.OnCollision(enemy);
    }
}

public void Update(double elapsedTime)
{
    UpdateCollisions();
}
```

Level 的 **Update** 方法在每一帧中都会调用碰撞处理代码。碰撞通过迭代敌人列表，并检查它们的包围框是否与玩家的包围框发生交叉而确定。交叉则是通过 C# 的 **RectangleF IntersectWith** 方法判断的。如果玩家的包围框和敌人的包围框发生交叉，则对玩家和敌人调用 **OnCollision**。**Player.OnCollision** 方法将被传入敌人对象，它与玩家发生碰撞。**Enemy.OnCollision** 将被传入玩家对象。代码中没有测试敌人与其他敌人的碰撞，因为游戏中假定发生这种情况是没有问题的。

在 **Enemy** 和 **PlayerCharacter** 类中都需要实现 **OnCollision** 类。下面是需要添加到 **Enemy** 类的一个框架方法。

```
internal void OnCollision(PlayerCharacter player)
{
    // Handle collision with player.
}
```

不同于 **Enemy**，**PlayerCharacter** 类实际上有一些功能，它的实现如下所示。

```
internal void OnCollision(Enemy enemy)
{
    _dead = true;
}
```

当玩家与敌人发生碰撞时，他的死亡标志将被设为 **true**，导致游戏结束。现在游戏已经部分可玩了，可能发生的结果包括玩家赢或者玩家输。从现在开始，将进一步细化游戏，使其拥有更好的可玩性。

10.3.4 添加简单的武器

游戏中的武器主要是各种各样的子弹。一个不错的目标是当玩家按下手柄上的 A 键或者键盘上的空格键时发射子弹。敌人也可以发射子弹，创建子弹系统时记住这一点很重要。

为了试验子弹，还需要添加另外一个纹理。从本书配套光盘的 **Assets** 文件夹中找到 **bullet**。

tga，把它添加到项目中，并像前面那样设置它的属性。然后需要把这个纹理加载到纹理管理器中。

```
_textureManager.LoadTexture("bullet", "bullet.tga");
```

加载纹理后，下一步自然应该创建 **Bullet** 类。该类将有一个包围框和一个精灵，所以也可以从 **Entity** 继承。应该在 **Shooter** 项目中创建该类。

```
public class Bullet : Entity
{
    public bool Dead { get; set; }
    public Vector Direction { get; set; }
    public double Speed { get; set; }

    public double X
    {
        get { return _sprite.GetPosition().X; }
    }

    public double Y
    {
        get { return _sprite.GetPosition().Y; }
    }

    public void SetPosition(Vector position)
    {
        _sprite.SetPosition(position);
    }

    public void SetColor(Color color)
    {
        _sprite.SetColor(color);
    }

    public Bullet(Texture bulletTexture)
    {
        _sprite.Texture = bulletTexture;

        // Some default values
        Dead = false;
        Direction = new Vector(1, 0, 0);
        Speed = 512; // pixels per second
    }

    public void Render(Renderer renderer)
```

```
{
    if (Dead)
    {
        return;
    }
    renderer.DrawSprite(_sprite);
}

public void Update(double elapsedTime)
{
    if (Dead)
    {
        return;
    }
    Vector position = _sprite.GetPosition();
    position += Direction * Speed * elapsedTime;
    _sprite.SetPosition(position);
}
}
```

Bullet 类有 3 个成员：子弹的飞行方向、子弹的速度，以及报告子弹是否消亡的标志。类中提供了子弹精灵的位置 **setter** 和 **getter**。另外，还有颜色的 **setter**，因为让子弹具有颜色是很合理的。玩家的子弹只能伤害敌人，敌人的子弹只能伤害玩家。为了使玩家能够区分各种子弹，给它们添加了不同的颜色。

看到位置的 **setter** 和 **getter** 以及颜色的 **setter** 后，你可能会想，使 **Sprite** 类成为公有类是不是会更好一些。然后如果我们想要修改位置或颜色，就可以直接修改子弹精灵。每种情况都有不同，但一般原则是，使尽可能多的数据保持私有，并为需要修改的数据提供接口。而且，**bullet.SetColor()** 比 **bullet.Sprite.SetColor()** 也更容易理解。

构造函数接受一个子弹纹理作为参数，并为颜色、方向和速度设置默认值。速度以每秒像素数度量。最后两个方法是 **Render** 和 **Update**。**Update** 循环使用子弹的方向和速度更新子弹的位置。位置增量与自上一帧后经过的时间相乘，这样在任意一台计算机上，移动都是一致的。**Render** 很简单，它只是绘制子弹精灵。如果子弹的 **Dead** 标志被设为 **true**，**Render** 和 **Update** 循环都不会采取任何操作。

游戏中将有大量的子弹四处飞行，所以还需要一定的逻辑来处理这种情况。离开屏幕的子弹需要被关闭。将这些逻辑放到一个 **BulletManager** 类中是很不错的。编写一个 **BulletManager** 类的方法有两种：简单直观的方法和高效使用内存的方法。这里介绍的 **BulletManager** 采用了简单直观的方法：当从屏幕上销毁敌人时，从 **BulletManager** 中删除对它的引用，并在代码中销毁对象，释放它占用的任何内存。每次玩家射击时，创建一个新子弹。这是最基本

的，但是在游戏循环中创建和删除大量的对象不是一个好主意，它们会降低代码运行的速度。创建和删除对象一般都会降低操作速度。

在管理子弹时，更加高效地使用内存的方法是创建子弹的一个列表(例如包含 1000 个子弹)。大多数子弹都是无效的，每次用户射击时，都会搜索列表并激活一个子弹。不需要创建新对象。如果全部 1000 个子弹都是活动的，那么禁止用户发射子弹，或者使用启发式方法使当前的某个子弹失效(例如生存时间最长的子弹)，并让玩家使用该子弹。以这种方式回收子弹是编写 `BulletManager` 的更好的方式。理解简单的管理器的应用后，你就可以自行把它转换为高效使用内存的子弹管理器。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine;

namespace Shooter
{
    public class Bullet : Entity
    {
        public bool Dead { get; set; }
        public Vector Direction { get; set; }
        public double Speed { get; set; }

        public double X
        {
            get { return _sprite.GetPosition().X; }
        }

        public double Y
        {
            get { return _sprite.GetPosition().Y; }
        }

        public void SetPosition(Vector position)
        {
            _sprite.SetPosition(position);
        }

        public void SetColor(Color color)
        {
            _sprite.SetColor(color);
        }
    }
}
```

```
    }

    public Bullet(Texture bulletTexture)
    {
        _sprite.Texture = bulletTexture;

        // Some default values
        Dead = false;
        Direction = new Vector(1, 0, 0);
        Speed = 512; // pixels per second
    }

    public void Render(Renderer renderer)
    {
        if (Dead)
        {
            return;
        }
        renderer.DrawSprite(_sprite);
    }

    public void Update(double elapsedTime)
    {
        if (Dead)
        {
            return;
        }
        Vector position = _sprite.GetPosition();
        position += Direction * Speed * elapsedTime;
        _sprite.SetPosition(position);
    }
}
```

BulletManager 只有两个成员变量：它管理的子弹的一个列表，和代表屏幕边界的一个矩形。记得在文件的顶部包含 `using System.Drawing` 语句，这样才能使用 `RectangleF` 类。屏幕边界用于确定子弹是否离开了屏幕并可以被销毁。

构造函数接受一个描述游戏区域的矩形的 `playArea`，并把它赋给 `_bounds` 成员。`Shoot` 方法用于向 **BulletManager** 添加一个子弹。添加了子弹后，**BulletManager** 类会跟踪它，直到该子弹离开了游戏区域或者击中了飞船。`Update` 方法会更新被跟踪的所有子弹，然后检查是否有子弹飞出了边界。最后，**BulletManager** 会删除 `Dead` 标志被设为 `true` 的任何子弹。

`CheckOutOfBounds` 函数在子弹和游戏区域之间使用矩形相交测试，以确定子弹是否飞出了游戏区域。`RemoveDeadBullets` 使用了一个有意思的小技巧，它反向迭代子弹列表，并删除所有失效的子弹。这里不能使用 `Foreach`，也不能使用前向迭代。如果进行前向迭代并删除子弹，列表将减少一个元素，当循环到达列表结尾时，将产生一个越界错误。反转循环迭代的方向可以解决这个问题。列表的长度不重要，它总是从 0 开始。

`Render` 方法很标准，它渲染出全部子弹。

最好把 `BulletManager` 放到 `Level` 类中。如果在 `Level.cs` 文件的顶部没有包含 `using System.Drawing` 语句，那么在使用 `RectangleF` 类之前需要添加该语句。

```
class Level
{
    BulletManager _bulletManager    new BulletManager(new RectangleF(-1300
/ 2, -750 / 2, 1300, 750));
```

`BulletManager` 被赋予了一个游戏区域，该区域比实际的窗口大小略大一些。这就提供了一个缓冲区，使得销毁子弹时它们已经完全离开了屏幕。然后需要把 `BulletManager` 添加到 `Level` 类的 `Update` 和 `Render` 方法中。

```
public void Update(double elapsedTime)
{
    UpdateCollisions();
    _bulletManager.Update(elapsedTime);

// A little later in the code

public void Render(Renderer renderer)
{
    _background.Render(renderer);
    _backgroundLayer.Render(renderer);

    _enemyList.ForEach(x => x.Render(renderer));
    _playerCharacter.Render(renderer);
    _bulletManager.Render(renderer);
}
```

最后才渲染 `BulletManager` 类，这样将在其他对象之上渲染子弹。现在，`BulletManager` 被完全整合了，但是如果不能让玩家发射子弹，就没有办法测试它。为了给玩家提供这种能力，`PlayerCharacter` 需要能够访问子弹管理器。在 `Level` 构造函数中，将 `BulletManager` 传递给 `PlayerCharacter` 构造函数。


```
_playerCharacter = new PlayerCharacter(_textureManager,  
_bulletManager);
```

然后需要修改 **PlayerCharacter** 类的代码，以接受和存储对 **BulletManager** 的引用。

```
BulletManager _bulletManager;  
Texture _bulletTexture;  
public PlayerCharacter(TextureManager textureManager, BulletManager  
bulletManager)  
{  
    _bulletManager = bulletManager;  
    _bulletTexture = textureManager.Get("bullet");  
}
```

PlayerCharacter 构造函数还存储了在发射子弹时将使用的 **bulletTexture**。为了发射子弹，需要创建一个 **bullet** 对象，并使其在玩家的附近开始显示，然后把它传递给 **BulletManager**。**PlayerCharacter** 类中新增加的 **Fire** 方法将负责完成这些处理。

```
Vector _gunOffset = new Vector(55, 0, 0);  
public void Fire()  
{  
    Bullet bullet = new Bullet(_bulletTexture);  
    bullet.SetColor(new Color(0, 1, 0, 1));  
    bullet.SetPosition(_sprite.GetPosition() + _gunOffset);  
    _bulletManager.Shoot(bullet);  
}
```

通过在构造函数中建立的 **bulletTexture** 可创建子弹。它被设置为绿色，但是你可以选择其他任意一种颜色。子弹的位置被设置为从玩家的飞船偏移一定位置，这样子弹看起来是从飞船的前端发射出去的。如果没有偏移，子弹刚好出现在飞船精灵的正中，这看上去有点奇怪。这里没有修改子弹的方向，因为在 *X* 轴上前向是默认值。默认的速度设置也是可以接受的。最后，将子弹分配给 **BulletManager**，并使用 **Shoot** 方法正式把它发射出去。

玩家现在可以发射子弹，但是没有检测输入和调用 **Fire** 方法的代码。玩家的所有输入将在 **Level** 类的 **Update** 方法中被处理。将输入代码放到 **Update** 方法中看上去有点混乱，所以我把这些代码提取为一个新函数 **UpdateInput**，这样代码看上去就更加整洁了。

```
public void Update(double elapsedTime)  
{  
    UpdateCollisions();  
    _bulletManager.Update(elapsedTime);  
  
    _background.Update((float)elapsedTime);  
    _backgroundLayer.Update((float)elapsedTime);  
}
```

```

        _enemyList.ForEach(x => x.Update(elapsedTime));

        // Input code has been moved into this method
        UpdateInput(elapsedTime);

    }

    private void UpdateInput(double elapsedTime)
    {
        if (_input.Keyboard.IsKeyPressed(Keys.Space) || _input.Controller.
        ButtonA.Pressed)
        {
            _playerCharacter.Fire();
        }
        // Pre-existing input code omitted.
    }

```

取出 `Update` 循环的所有输入代码，并把它放到新的 `UpdateInput` 方法的结尾。然后从 `Update` 方法中调用这个 `UpdateInput` 方法。另外还添加了一些新代码来处理玩家发射子弹的动作。如果按下了键盘上的空格键或者手柄上的 A 按键，那么玩家就发射了一个子弹。运行程序，尝试飞船新增的开火功能。

从图 10-8 中可以看到新添加的子弹。每次玩家按下发射按键时，都会创建子弹。处于可玩性考虑，最好降低发射速度，使飞船在连续发射子弹之间有一个恢复时间。修改 `PlayerCharacter` 类如下。

```

Vector _gunOffset = new Vector(55, 0, 0);
static readonly double FireRecovery = 0.25;
double _fireRecoveryTime = FireRecovery;
public void Update(double elapsedTime)
{
    _fireRecoveryTime = Math.Max(0, (_fireRecoveryTime - elapsedTime));
}

public void Fire()
{
    if (_fireRecoveryTime > 0)
    {
        return;
    }
    else
    {
        _fireRecoveryTime = FireRecovery;
    }
}

```

```
Bullet bullet = new Bullet(_bulletTexture);  
bullet.SetColor(new Color(0, 1, 0, 1));  
bullet.SetPosition(_sprite.GetPosition() + _gunOffset);  
_bulletManager.Shoot(bullet);  
}
```

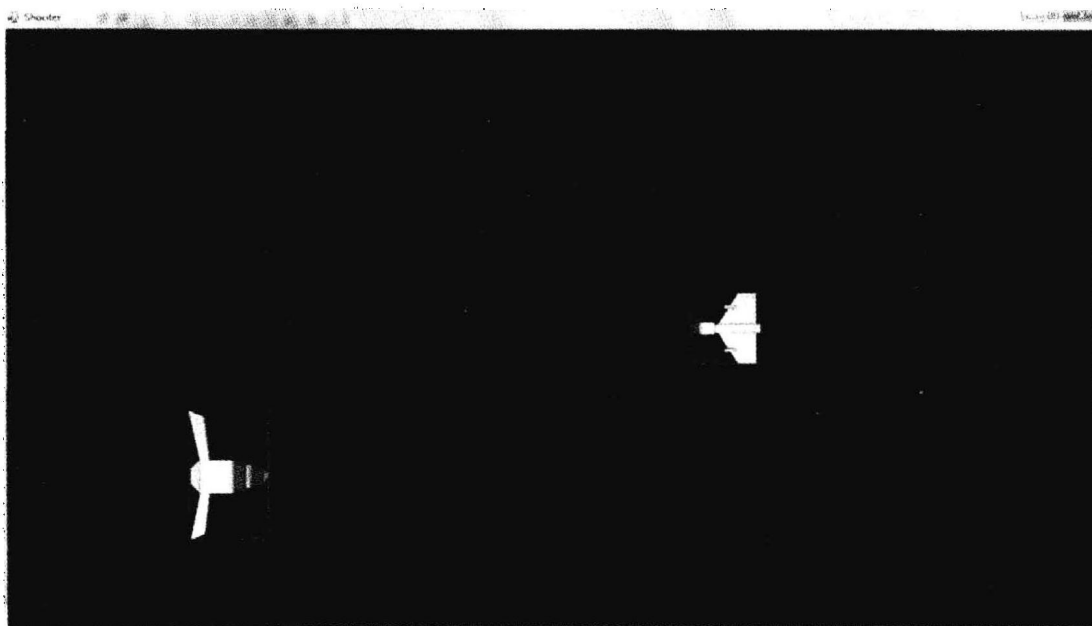


图 10-8 发射子弹

为了对恢复时间进行倒计时,需要在 `PlayerCharacter` 类中添加一个 `Update` 方法。`Update` 方法将对恢复时间进行倒计时,但是绝不会低于 0。这是使用 `Math.Max` 函数完成的。设置了恢复时间后,如果飞船仍然处在从上一次发射恢复的状态,`Fire` 命令将被立即返回。如果恢复时间为 0,飞船可以发射子弹,恢复时间将被重设,以便重新开始倒计时。

还需要对 `Level` 类做一点小修改:它需要调用 `PlayerCharacter` 的 `Update` 方法。

```
public void Update(double elapsedTime)  
{  
    _playerCharacter.Update(elapsedTime);  
  
    _background.Update((float)elapsedTime);  
    _backgroundLayer.Update((float)elapsedTime);  
  
    UpdateCollisions();  
    _enemyList.ForEach(x => x.Update(elapsedTime));  
    _bulletManager.Update(elapsedTime);  
}
```

```
UpdateInput(elapsedTime);  
}
```

再次运行程序，现在不能像以前那样快速发射子弹了。这是你自己的游戏，所以只要你觉得合适，可以随意调整恢复速度。

10.3.5 伤害和爆炸

现在添加了子弹，但是它们直接穿过敌人飞船，却不会造成任何伤害。接下来我们就来解决这个问题。敌人应该知道自己何时被击中，并相应地做出恰当的反应。我们将首先处理碰撞问题，然后创建一个爆炸动画。

碰撞代码在 **Level** 类的 **UpdateCollisions** 函数中处理。需要扩展这个函数，使其也可以处理子弹和敌人之间的碰撞。

```
private void UpdateCollisions()  
{  
    foreach (Enemy enemy in _enemyList)  
    {  
        if (enemy.GetBoundingBox().Intersects(_playerCharacter.  
GetBoundingBox()))  
        {  
            enemy.OnCollision(_playerCharacter);  
            _playerCharacter.OnCollision(enemy);  
        }  
  
        _bulletManager.UpdateEnemyCollisions(enemy);  
    }  
}
```

在 **for** 循环的末尾新添加了一行，让 **BulletManager** 检查子弹与当前敌人之间是否发生碰撞。**UpdateEnemyCollisions** 是 **BulletManager** 中的新函数，所以需要实现它。

```
internal void UpdateEnemyCollisions(Enemy enemy)  
{  
    foreach (Bullet bullet in _bullets)  
    {  
        if (bullet.GetBoundingBox().Intersects(enemy.GetBounding-  
Box()))  
        {  
            bullet.Dead = true;  
            enemy.OnCollision(bullet);  
        }  
    }  
}
```

子弹和敌人之间的碰撞通过检查包围框是否相交进行判断。如果子弹击中了敌人，则销毁子弹并通知敌人发生碰撞。

子弹击中敌人后，有多种不同的方式进行反应。可以立即销毁敌人并播放爆炸动画，或者可以使敌人遭受一定的伤害，还需要再击中敌人几次后才销毁它。为敌人分配生命值这种功能在将来可能会需要，不如现在就将这种功能添加进来。为敌人添加一个 **Health** 成员变量，然后可以实现子弹的 **OnCollision** 函数。

```
public int Health { get; set; }
public Enemy(TextureManager textureManager)
{
    Health = 50; // default health value.
    //Remaining constructor code omitted
}
```

Enemy 类已经有一个 **OnCollision** 方法，但是该方法用于敌人与 **PlayerCharacter** 的碰撞。我们将创建一个新的重载后的 **OnCollision** 方法，它只关心与子弹的碰撞。当子弹击中敌人后，敌人会受到一些伤害，其生命值会降低。如果敌人的生命值低于 0，则会被销毁。如果玩家击中了敌人，并使其受到了伤害，需要在视觉上有一些显示，以表明敌人被击中了。表示这种反馈的一种好方法是在零点几秒的时间内使敌人飞船闪烁黄色。

```
static readonly double HitFlashTime = 0.25;
double _hitFlashCountDown = 0;
internal void OnCollision(Bullet bullet)
{
    // If the ship is already dead then ignore any more bullets.
    if (Health == 0)
    {
        return;
    }

    Health = Math.Max(0, Health - 25);
    _hitFlashCountDown = HitFlashTime; // half
    _sprite.SetColor(new Engine.Color(1, 1, 0, 1));

    if (Health == 0)
    {
        OnDestroyed();
    }
}

private void OnDestroyed()
{
}
```

```
// Kill the enemy here.  
}
```

`OnDestroyed` 函数现在还只是一个占位函数,稍后我们才会关心如何销毁敌人。在 `OnCollision` 函数中,第一个 `if` 语句会检查飞船的生命值是否为 0。在这里将忽略任何额外的伤害,因为玩家已经击毁了敌人,游戏不需要再考虑其他命中的子弹。接下来将 `Health` 的值减少 25,这是一个随意选取的数字,代表子弹击中一次时造成的伤害。`Math.Max` 用于确保生命值不会低于 0。当被击中时,飞船将闪烁黄色。设置的倒计时用于代表闪烁的时间。飞船精灵被设为黄色,其 `RGBA` 值为(1,1,0,1)。最后,检查生命值,如果生命值等于 0,则调用占位函数 `OnDestroyed`,爆炸将在这个函数中触发。

为了使飞船开始闪烁,需要修改 `Update` 循环。在该方法中需要倒计时闪烁时间,并把颜色从黄色改为白色。

```
public void Update(double elapsedTime)  
{  
    if (_hitFlashCountDown != 0)  
    {  
        _hitFlashCountDown = Math.Max(0, _hitFlashCountDown - elapsedTime);  
        double scaledTime = 1 - (_hitFlashCountDown / HitFlashTime);  
        _sprite.SetColor(new Engine.Color(1, 1, (float)scaledTime, 1));  
    }  
}
```

`Update` 循环修改敌人飞船的闪烁颜色。如果闪烁时间降低为 0,那么闪烁已经结束,不需要再进行更新。如果 `_hitFlashCountDown` 不等于 0,则将它减去从上一帧后经过的时间。这里再次使用 `Math.Max` 来确保倒计时不会低于 0。然后倒计时被减小到 0~1 之间的一个值,以此表示闪烁的进度:0 代表闪烁刚开始,1 代表闪烁已结束。将 1 减去得到的这个值,从而使数字的意义刚好相反:1 代表闪烁刚开始,0 代表闪烁已经结束。然后使用这个缩放后的数值在 0~1 之间改变颜色的蓝色通道。这会使飞船的闪烁颜色从黄色变为白色。

运行程序,然后多次击中敌人飞船,它会多次闪烁黄色,然后被销毁,从而停止了响应。但是,敌人的飞船不能只是停止响应,它们应该爆炸。

产生一个出色的爆炸效果的最简单的方法是使用动画精灵。图 10-9 显示了一次爆炸的关键帧纹理贴图。纹理使用过程式爆炸生成器创建,该生成器可以从 Positech 游戏(<http://www.positech.co.uk/content/explosion/explosiongenerator.html>)上免费下载。

图 10-9 总共有 16 个帧,横向 4 个、纵向 4 个。通过读取这个纹理并修改(U,V)坐标,使其随时间从第一个帧移动到最后一个帧,可以创建动画精灵。动画精灵只不过是另外一类精灵,所以为了创建它们,需扩展已有的 `Sprite` 类。动画精灵可以被多种不同的游戏使用,所以应该在 `Engine` 项目中而不是在游戏项目中创建它们。

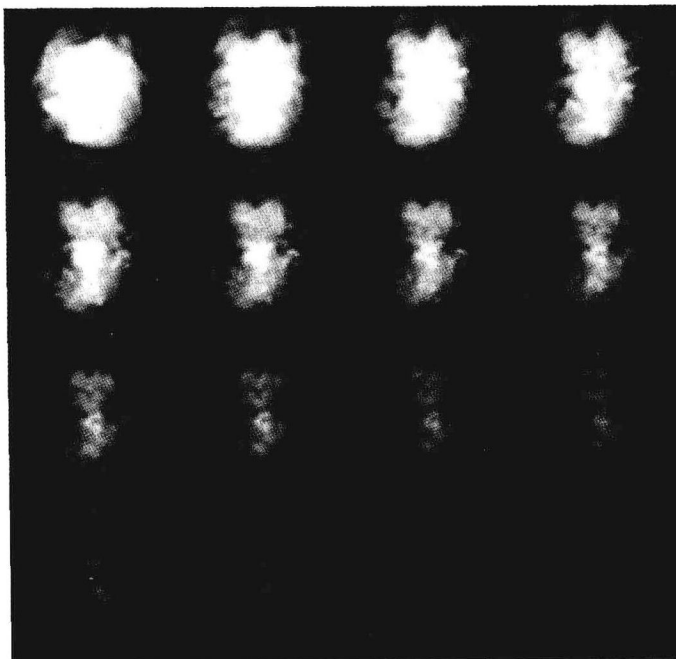


图 10-9 动画显示的爆炸纹理贴图

```
public class AnimatedSprite : Sprite
{
    int _framesX;
    int _framesY;
    int _currentFrame = 0;
    double _currentFrameTime = 0.03;
    public double Speed { get; set; } // seconds per frame
    public bool Looping { get; set; }
    public bool Finished { get; set; }

    public AnimatedSprite()
    {
        Looping = false;
        Finished = false;
        Speed = 0.03; // 30 fps-ish
        _currentFrameTime = Speed;
    }

    public System.Drawing.Point GetIndexFromFrame(int frame)
    {
        System.Drawing.Point point = new System.Drawing.Point();
        point.Y = frame / _framesX;
    }
}
```

```
        point.X = frame - (point.Y * _framesY);
        return point;
    }

    private void UpdateUVs()
    {
        System.Drawing.Point index = GetIndexFromFrame(_currentFrame);
        float frameWidth = 1.0f / (float)_framesX;
        float frameHeight = 1.0f / (float)_framesY;
        SetUVs(new Point(index.X * frameWidth, index.Y * frameHeight),
            new Point((index.X + 1) * frameWidth, (index.Y + 1) * frameHeight));
    }

    public void SetAnimation(int framesX, int framesY)
    {
        _framesX = framesX;
        _framesY = framesY;
        UpdateUVs();
    }

    private int GetFrameCount()
    {
        return _framesX * _framesY;
    }

    public void AdvanceFrame()
    {
        int numberOfFrames = GetFrameCount();
        _currentFrame = (_currentFrame + 1) % numberOfFrames;
    }

    public int GetCurrentFrame()
    {
        return _currentFrame;
    }

    public void Update(double elapsedTime)
    {
        if (_currentFrame == GetFrameCount() - 1 && Looping == false)
        {
            Finished = true;
            return;
        }

        _currentFrameTime -= elapsedTime;
    }
}
```



```
if (_currentFrameTime < 0)
{
    AdvanceFrame();
    _currentFrameTime = Speed;
    UpdateUVs();
}
}
```

这个 `AnimatedSprite` 类的工作方式与 `Sprite` 类完全相同，只不过可以告诉 `AnimatedSprite` 类纹理在 X 和 Y 方向上有多少帧。调用 `Update` 循环时，帧随时间改变。

这个类有好几个成员，但是它们主要用于描述动画和跟踪其进度。 X 和 Y 方向上的帧数由 `_framesX` 和 `_framesY` 成员变量描述。对于图 10-9 中的示例，这两个变量都将被设为 4。`_currentFrame` 变量是精灵的 (U,V) 当前被设置的帧。`_currentFrameTime` 是动画前进到下一帧之前，当前帧将占用的时间。`Speed` 度量每一帧占用的时间，单位为 s。`Looping` 决定动画是否应该循环。`Finished` 是一个标志，当动画结束后被设置为 `true`。

`AnimatedSprite` 的构造函数设置一些默认值。新创建的精灵不循环，`Finished` 标志被设为 `false`，帧速被设为每秒大约 30 帧，`_currentFrameTime` 被设为 0.03s，这会使动画以每秒 30 帧的速度运行。

`GetIndexFromFrame` 方法接受如图 10-10 所示的索引为参数，返回索引位置的 (X,Y) 坐标。例如，索引 0 将返回 $(0,0)$ ，索引 15 将返回 $(3,3)$ 。通过把索引除以行的长度，索引被分解为 (X,Y) 坐标，除法得到了行数，从而也得到了索引的 Y 坐标。 X 坐标就是将 Y 行移除后索引剩下的部分。进行平移时，这个函数非常有用，可以计算出特定帧的 (U,V) 坐标。

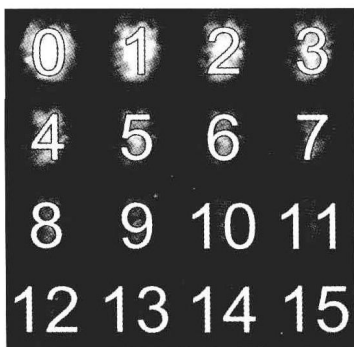


图 10-10 索引

`UpdateUVs` 使用当前帧索引来修改 (U,V) ，所以精灵可以正确地代表帧。它首先使用 `GetIndexFromFrame` 获得当前帧的 (X,Y) 坐标，然后计算各个帧的宽度和高度。纹理坐标在 0~1 之间，单个帧的宽度和高度通过用 1 除以 X 轴和 Y 轴的帧数计算出来。计算出单个帧的尺寸后，可以通过把帧的宽度和高度乘以当前帧的 (X,Y) 坐标计算出来 (U,V) 的位置，这将得到帧在纹理贴图左上角的点。`SetUVs` 方法需要左上角的点和右下角的点作为参数。右下角的位置通过把左上角的点加上一个帧的宽度和高度计算出来。

SetAnimation 用于设置纹理贴图在 X 方向和 Y 方向上的帧数。它调用 **UpdateUVs**，以更新精灵来显示正确的帧。**GetFrameCount** 获得动画中的总帧数。**AdvanceFrame** 方法将动画移动到下一帧，如果到达最后一帧，则索引将变为 0。这种环绕是使用取模运算符%完成的。取模运算符用于计算整数除法的余数。理解取模运算符的用途的最佳方式是向你提供一个你可能已经很熟悉的示例：时间。钟表上有 12 个数字，它以与 12 取模的方式工作：13 点与 12 取模是 1 点。在这里，模数等于动画中的总帧数。

Update 方法负责更新当前帧，并使爆炸看上去以动画方式显示。如果 **Looping** 被设为 **false**，并且当前帧是最后一帧，那么 **Update** 方法会立即返回，且 **Finished** 标志被设为 **true**。如果动画还未结束或正在循环，则更新帧倒计时 **_currentFrameTime**，如果该值小于 0，则需要修改帧。帧的更新是通过调用 **AdvanceFrame**、重置 **_currentFrameTime** 并最终更新(U,V)完成的。

把 **AnimatedSprite** 类添加到 **Engine** 项目中后，可以测试爆炸。从本书配套光盘的 **Assets** 文件夹中找到 **explode.tga** 文件，把它添加到项目中，并像以前一样设置属性。然后可以在 **form.cs** 中加载它。

```
_textureManager.LoadTexture("explosion", "explode.tga");
```

一种快速测试动画的方法是把它作为动画精灵直接加载到 **Level** 中。

```
AnimatedSprite _testSprite = new AnimatedSprite();public Level(Input
input, TextureManager textureManager, PersistantGameData gameData)
{
    _testSprite.Texture = textureManager.Get("explosion");
    _testSprite.SetAnimation(4, 4);

    // a little later in the code
    public void Update(double elapsedTime)
    {
        _testSprite.Update(elapsedTime);

        // a little later in the code

    public void Render(Renderer renderer)
    {
        // Background and other sprite code omitted.
        renderer.DrawSprite(_testSprite);
        renderer.Render();
    }
}
```

运行程序并进入关卡，现在将播放一次爆炸动画。这证明了代码工作正确(见图 10-11)。

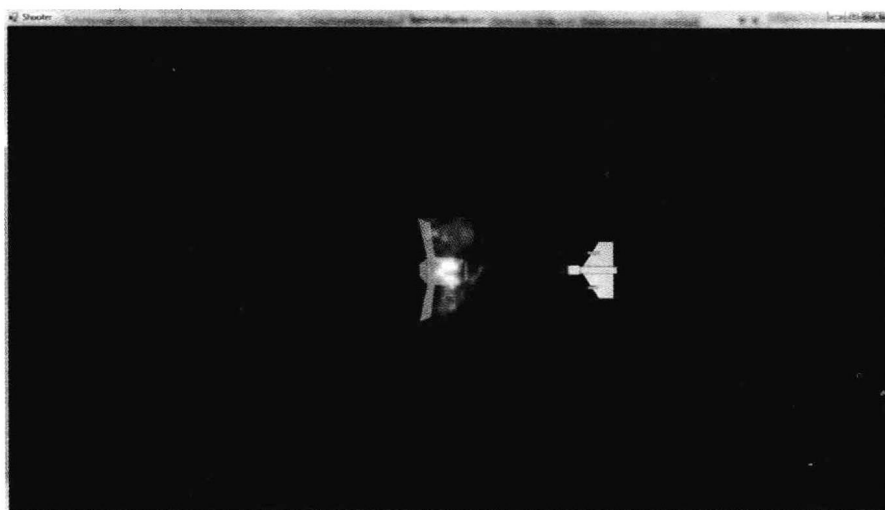


图 10-11 游戏中的爆炸效果

10.3.6 管理爆炸和敌人

在第 10.3.5 节中，我们创建了爆炸效果，但是只有毁灭敌人时才应该引发这种爆炸效果。为此，需要创建两个新系统：一个处理爆炸和一般的游戏效果，另一个处理逼近的敌人。

处理爆炸的方式应该与处理子弹的方式类似，即创建一个专门的管理器来处理爆炸的创建和销毁。将来你自己创建项目时，可能想添加更多的效果，例如烟雾、火星或者增加玩家能力的东西。EffectsManager 类应该在 Shooter 项目中创建。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine;

namespace Shooter
{
    public class EffectsManager
    {
        List<AnimatedSprite> _effects = new List<AnimatedSprite>();
        TextureManager _textureManager;

        public EffectsManager(TextureManager textureManager)
        {
            _textureManager = textureManager;
        }

        public void AddExplosion(Vector position)
```

```
{
    AnimatedSprite explosion = new AnimatedSprite();
    explosion.Texture = _textureManager.Get("explosion");
    explosion.SetAnimation(4, 4);
    explosion.SetPosition(position);
    _effects.Add(explosion);
}

public void Update(double elapsedTime)
{
    _effects.ForEach(x => x.Update(elapsedTime));
    RemoveDeadExplosions();
}

public void Render(Renderer renderer)
{
    _effects.ForEach(x => renderer.DrawSprite(x));
}

private void RemoveDeadExplosions()
{
    for (int i = _effects.Count - 1; i >= 0; i-)
    {
        if (_effects[i].Finished)
        {
            _effects.RemoveAt(i);
        }
    }
}
}
```

EffectsManager 引发爆炸，播放爆炸动画直到其结束，然后删除爆炸效果。注意它与 **BulletManager** 类很相似。这些独立的管理器可以合并为一个通用的管理器，但是通过使它们保持独立，游戏对象之间的交互会更加具体而高效。爆炸不关心与敌人或玩家的碰撞检测，但是子弹则相反。在独立的管理器中，分离出每个对象的特定需求很容易：爆炸只需要运行动画，而子弹需要检查与所有敌人的相交。当游戏中只有少量的对象时，独立的管理器的效果很好，但是如果游戏中将存在很多个不同的实体，那么更通用的实体管理器是更好的选择。

EffectsManager 需要在 **Level** 类中实例化，并与渲染和更新循环关联在一起。

```
EffectsManager _effectsManager;
```

```
public Level(Input input, TextureManager textureManager, PersistantGameData
gameData)
{
    _input = input;
    _gameData = gameData;
    _textureManager = textureManager;
    _effectsManager = new EffectsManager(_textureManager);

    // code omitted

    public void Update(double elapsedTime)
    {
        _effectsManager.Update(elapsedTime);

        // code omitted

    public void Render(Renderer renderer)
    {
        // Background, sprites and bullet code omitted
        _effectsManager.Render(renderer);
        renderer.Render();
    }
}
```

ExplosionManager 现在已被关联，可以用于同时启动多个爆炸。为了使敌人在死亡时可以启动爆炸，需要使它们能够访问管理器，所以将管理器作为参数传递给 **Enemy** 的构造函数。

```
EffectsManager _effectsManager;

public Enemy(TextureManager textureManager, EffectsManager
effectsManager)
{
    _effectsManager = effectsManager;
```

现在敌人就可以在死亡时引发爆炸了。

```
private void OnDestroyed()
{
    // Kill the enemy here.
    _effectsManager.AddExplosion(_sprite.GetPosition());
}
```

在 **Level.cs** 文件中，需要把 **EffectsManager** 传入 **Enemy** 的构造函数。完成这个操作以后，在游戏中多次击中敌人将会消灭敌人并引发爆炸。

接下来，为敌人创建自己的管理器，这是为了开发出完整可工作的游戏需要创建的最

后一个管理器。

```
public class EnemyManager
{
    List<Enemy> _enemies = new List<Enemy>();
    TextureManager _textureManager;
    EffectsManager _effectsManager;
    int _leftBound;

    public List<Enemy> EnemyList
    {
        get
        {
            return _enemies;
        }
    }

    public EnemyManager(TextureManager textureManager, EffectsManager
effectsManager, int leftBound)
    {
        _textureManager = textureManager;
        _effectsManager = effectsManager;
        _leftBound = leftBound;

        // Add a test enemy.
        Enemy enemy = new Enemy(_textureManager, _effectsManager);
        _enemies.Add(enemy);
    }

    public void Update(double elapsedTime)
    {
        _enemies.ForEach(x => x.Update(elapsedTime));
        CheckForOutOfBounds();
        RemoveDeadEnemies();
    }

    private void CheckForOutOfBounds()
    {
        foreach (Enemy enemy in _enemies)
        {
            if (enemy.GetBoundingBox().Right < _leftBound)
            {
                enemy.Health = 0; // kill the enemy off
            }
        }
    }
}
```

```

    }
}

public void Render(Renderer renderer)
{
    _enemies.ForEach(x => x.Render(renderer));
}

private void RemoveDeadEnemies()
{
    for (int i = _enemies.Count - 1; i >= 0; i-)
    {
        if (_enemies[i].IsDead)
        {
            _enemies.RemoveAt(i);
        }
    }
}
}

```

还需要在 **Enemy** 类中添加另外一个函数来检查敌人是否被消灭。

```

class Enemy : Entity
{
    public bool IsDead
    {
        get { return Health == 0; }
    }
}

```

如果敌人的生命值等于0,则**Enemy**类的**IsDead**方法返回**true**,否则返回**false**。**EnemyManager**与**BulletManager**一样有越界检查,但是它稍有不同。卷轴射击游戏中的敌人一般从屏幕的最右边开始出现,然后越过玩家从屏幕左边离开。越界检查比较敌人包围框最右边的点与屏幕最左边的点,这样就可以删除没有被玩家消灭、从屏幕左边逃脱的敌人。

现在需要修改 **Level** 类来引入这个新的管理器,并删除旧列表。

```

// List<Enemy> _enemyList = new List<Enemy>(); <- Removed
EnemyManager _enemyManager;

public Level(Input input, TextureManager textureManager, PersistantGameData
gameData)
{
    _input = input;
    _gameData = gameData;
}

```

```

_textureManager = textureManager;

_background = new ScrollingBackground(textureManager.Get
("background"));
_background.SetScale(2, 2);
_background.Speed = 0.15f;

_backgroundLayer = new ScrollingBackground(textureManager.Get
("background_layer_1"));
_backgroundLayer.Speed = 0.1f;
_backgroundLayer.SetScale(2.0, 2.0);

_playerCharacter = new PlayerCharacter(_textureManager,
_bulletManager);

_effectsManager = new EffectsManager(_textureManager);
// _enemyList.Add(new Enemy(_textureManager, _effectsManager));
<- Removed
_enemyManager = new EnemyManager(_textureManager, _effectsManager,
-1300);
}

```

对碰撞处理也需要做一些修改，现在在检查与敌人的碰撞时，它将使用 **EnemyManager** 中的敌人列表。

```

private void UpdateCollisions()
{
    foreach (Enemy enemy in _enemyManager.EnemyList)

```

为了能够看到敌人，需要修改 **Update** 和 **Render** 循环。

```

public void Update(double elapsedTime)
{
    // _enemyList.ForEach(x => x.Update(elapsedTime)); <- Remove this line
    _enemyManager.Update(elapsedTime);

    // Code omitted

public void Render(Renderer renderer)
{
    _background.Render(renderer);
    _backgroundLayer.Render(renderer);

    // _enemyList.ForEach(x => x.Render(renderer)); <- remove this line

```



```
_enemyManager.Render(renderer);
```

现在运行程序。击中敌人几次后，敌人会爆炸并消失。这看起来更像是一个真正的游戏了。现在最明显的缺陷是，只有一个敌人，而且这个敌人还不会移动。

10.3.7 定义关卡

当前的关卡只持续 30s，并且只有一个敌人，所以不是十分有趣。如果有一个定义关卡的系统，那么就可以为这个关卡增添一些激动人心的元素。关卡定义是在特定时间生成的敌人的一个列表。因此，关卡定义需要一种定义敌人的方法，下面的代码可以作为一个不错的起点。应该把 **EnemyDef** 类添加到 **Engine** 项目中。

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Engine;
namespace Shooter
{
    class EnemyDef
    {
        public string EnemyType { get; set; }
        public Vector StartPosition { get; set; }
        public double LaunchTime { get; set; }
        public EnemyDef()
        {
            EnemyType = "cannon_fodder";
            StartPosition = new Vector(300, 0, 0);
            LaunchTime = 0;
        }

        public EnemyDef(string enemyType, Vector startPosition, double
launchTime)
        {
            EnemyType = enemyType;
            StartPosition = startPosition;
            LaunchTime = launchTime;
        }
    }
}
```

这里使用了一个字符串来描述敌人类型。在代码中，可以提供几种不同类型的敌人：

体型小而移动迅速的敌人、体型大而移动缓慢的敌人等。默认的敌人类型是 **Cannon Fodder**(炮灰)。敌人的初始位置为屏幕右侧，启动时间是敌人将在关卡中出现的时间。关卡时间从某个较大的数值倒计时为 0。如果 **gameTime** 小于启动时间，将创建一个 **enemy** 对象并将其添加到关卡中。

EnemyManager 类将处理生成敌人的操作。这意味着需要修改构造函数，并添加一个将出现的敌人的列表。

```
List<EnemyDef> _upComingEnemies = new List<EnemyDef>();
public EnemyManager(TextureManager textureManager, EffectsManager
effectsManager, int leftBound)
{
    _textureManager = textureManager;
    _effectsManager = effectsManager;
    _leftBound = leftBound;
    _upComingEnemies.Add(new EnemyDef("cannon_fodder", new Vector(300,
300, 0), 25));
    _upComingEnemies.Add(new EnemyDef("cannon_fodder", new Vector(300,
-300, 0), 30));
    _upComingEnemies.Add(new EnemyDef("cannon_fodder", new Vector(300, 0,
0), 29));

    // Sort enemies so the greater launch time appears first.
    _upComingEnemies.Sort(delegate(EnemyDef firstEnemy, EnemyDef
secondEnemy)
    {
        return firstEnemy.LaunchTime.CompareTo(secondEnemy.LaunchTime);
    });
}
```

_upcomingEnemies 列表是按照启动时间排序的敌人定义的一个列表。启动时间越长，敌人定义在列表中出现的位置越高。每一帧都检查列表中最顶部的项，看其是否准备好启动。只需要检查最顶部的敌人定义，因为列表已经排好序了。如果列表没有排序，则需要检查列表中的每一项来确定哪个敌人定义的启动时间大于当前的 **gameTime**，因而需要稍后启动。

启动敌人的操作是在 **EnemyManager** 的 **Update** 循环中完成的，**Update** 将调用新方法 **UpdateEnemySpawns**。

```
private void UpdateEnemySpawns(double gameTime)
{
    // If no upcoming enemies then there's nothing to spawn.
    if (_upComingEnemies.Count == 0)
    {
        return;
    }
}
```

```

    }

    EnemyDef lastElement = _upComingEnemies[_upComingEnemies.Count - 1];
    if (gameTime < lastElement.LaunchTime)
    {
        _upComingEnemies.RemoveAt(_upComingEnemies.Count - 1);
        _enemies.Add(CreateEnemyFromDef(lastElement));
    }
}

private Enemy CreateEnemyFromDef(EnemyDef definition)
{
    Enemy enemy = new Enemy(_textureManager, _effectsManager);
    enemy.SetPosition(definition.StartPosition);
    if (definition.EnemyType == "cannon_fodder")
    {
        // The enemy type could be used to alter the health or texture
        // but we're using the default texture and health for the cannon
        fodder type
    }
    else
    {
        System.Diagnostics.Debug.Assert(false, "Unknown enemy type.");
    }

    return enemy;
}

public void Update(double elapsedTime, double gameTime)
{
    UpdateEnemySpawns(gameTime);
}

```

EnemyManager 中的 **Update** 方法和 **Level** 类已被修改,接受一个 **gameTime** 参数。**gameTime** 是一个用于倒计时的数字,当减小到 0 时,关卡将结束。这个值用于确定何时创建新敌人。**InnerGameState** 需要把这个 **gameTime** 值传递给 **Level** 对象的 **Update** 方法, **Level** 会把它传递给 **EnemyManager**。

```

// In Level.cs
public void Update(double elapsedTime, double gameTime)
{
    _enemyManager.Update(elapsedTime, gameTime);
}
// In InnerGameState.cs
public void Update(double elapsedTime)

```

```
{  
    _level.Update(elapsedTime, _gameTime);
```

`gameTime` 从游戏主体状态一直传递给 `EnemyManager` 中的 `UpdateEnemySpawns` 函数。`UpdateEnemySpawns` 首先检查 `_upcomingEnemies` 列表中是否有即将出现的敌人, 如果没有, 该方法不执行操作。如果有即将出现的敌人, 代码会检查列表的顶部, 看是否有准备启动的敌人。如果敌人定义已经准备启动, 则从 `_upcomingEnemies` 列表中移除该定义, 并使用该定义创建的新的敌人对象。然后新创建的敌人将被添加到 `_enemies` 列表中, 并在游戏世界中生成。

`CreateEnemyFromDef` 接受一个 `EnemyDef` 对象, 并返回一个 `Enemy` 对象。现在只有一种类型的敌人, 所以这个函数很简单, 不过它还有很多添加新敌人类型的空间。

现在运行程序, 随着关卡的时间流逝, 将会显示 3 个敌人。

10.3.8 敌人的移动

卷轴射击游戏中的敌人应该从屏幕右边蜂拥而上, 并试图从屏幕左边攻击玩家。图 10-12 显示了敌人的前进方向。玩家的子弹已经有移动代码, 敌人可以重用这些代码。这样得到的代码是可以工作的, 但是敌人的移动十分呆板: 它们将直直地从右向左移动。敌人的移动本应该更加有趣, 对此, 最简单的实现方法是为每个敌人预定义一个由多个路径点组成的路径。敌人将经过所有的路径点并从左侧离开。



图 10-12 敌人的前进方向

路径可以简单地被描述为从屏幕右侧通向屏幕左侧的一系列点。图 10-13 显示了一个由路径点组成的路径, 它可以描述敌人在游戏区域经过的路径。

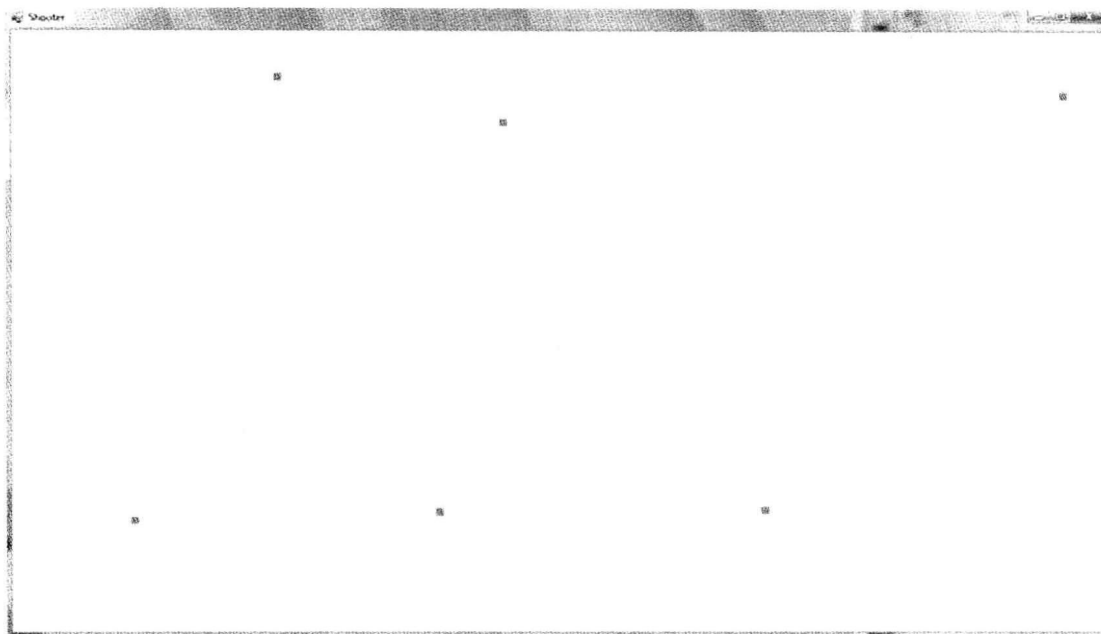


图 10-13 由路径点组成的路径

路径点可以连接到一起，得到如图 10-14 所示的路径。这是敌人可以采用的路径，但边角是锯齿形的。如果路径更加平滑就好了。样条函数是创建平滑路径的一种好方法。图 10-15 显示了一个 Catmull-Rom 样条，这种样条一定会穿过所有的控制点。Pixar 的 Edwin Catmull 曾参与创作了《玩具总动员(Toy Story)》，他与 Raphael Rom 共同创建了这种样条。

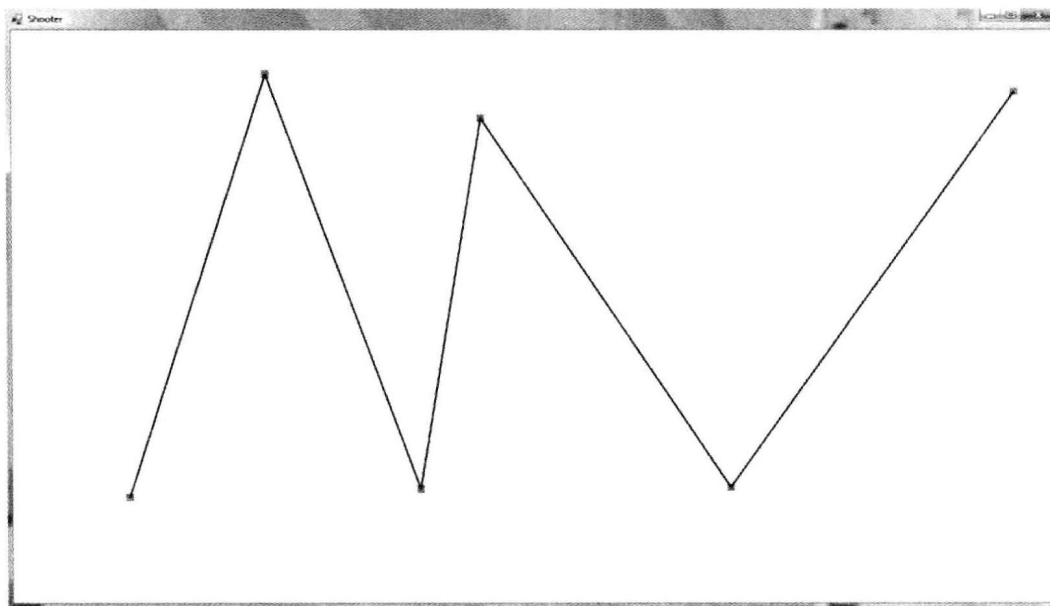


图 10-14 路径的线性插值

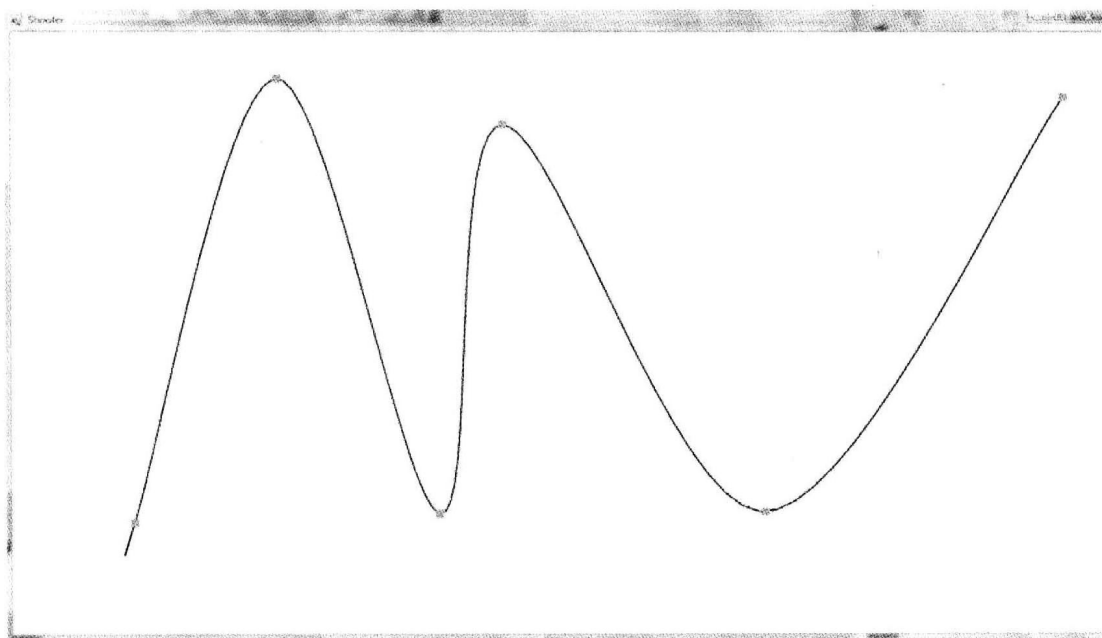


图 10-15 样条路径

样条显然更加平滑，但是需要创建另外一个类。样条是对曲线的数学描述。

Catmull-Rom 样条可以获得组成样条的任意两点之间的位置 t 。在 Catmull-Rom 样条中， t 两边的两个点以及这两个点的邻点将参与计算，如图 10-16 所示。

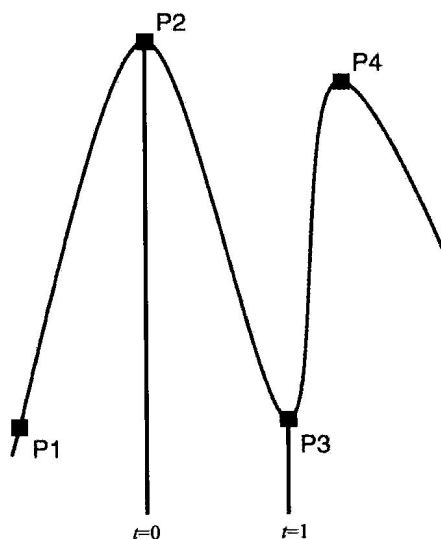


图 10-16 Catmull-Rom 样条

一旦可以获得任意两个邻点之间的 $t(0\sim1)$ 的值，就可以进行推广，使 $t(0\sim1)$ 可以映射到整条线上，而不只是一个线段。使用 t 和 4 个点计算出位置的方法如下：

$$f(t) = 0.5 * \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} * \begin{bmatrix} -1 & 3 & -3 & -1 \\ 2 & -5 & -4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{bmatrix} * \begin{bmatrix} P0 \\ P1 \\ P2 \\ P3 \end{bmatrix}$$

看上去有点让人生畏：将 3 个矩阵与一个标量相乘，该标量对所有 4 个点进行加权，并决定如何把 t 的值转换为一个位置。准确理解这个公式的工作原理并不重要(当然，鼓励你做一些研究)，只要知道应用这个公式时会得到什么结果就行了。

下面给出了 Catmull-Rom 样条的 C#实现。应该把这个类添加到 Engine 项目中，因为它可以用在多个项目中。这段样条代码可以用在 3D 中，用于操作摄像机或者沿着一条路径移动 3D 实体等任务。Spline 类的接口基于以 Radu Gruian 的 C++ Overhauser 为基础的代码(网址为 <http://www.codeproject.com/KB/recipes/Overhauser.aspx>，Code Project 网站可能要求注册后再查看该文章，注册是免费的)。

```
public class Spline
{
    List<Vector> _points = new List<Vector>();
    double _segmentSize = 0;

    public void AddPoint(Vector point)
    {
        _points.Add(point);
        _segmentSize = 1 / (double)_points.Count;
    }
    private int LimitPoints(int point)
    {
        if(point < 0)
        {
            return 0;
        }
        else if (point > _points.Count - 1)
        {
            return _points.Count - 1;
        }
        else
        {
            return point;
        }
    }

    // t ranges from 0 - 1
    public Vector GetPositionOnLine(double t)
    {

```

```

        if (_points.Count <= 1)
        {
            return new Vector(0,0,0);
        }

        // Get the segment of the line we're dealing with.
        int interval = (int)(t / _segmentSize);

        // Get the points around the segment
        int p0 = LimitPoints(interval - 1);
        int p1 = LimitPoints(interval);
        int p2 = LimitPoints(interval + 1);
        int p3 = LimitPoints(interval + 2);

        // Scale t to the current segment
        double scaledT = (t - _segmentSize * (double)interval) / _segmentSize;
        return CalculateCatmullRom(scaledT, _points[p0], _points[p1],
            _points[p2], _points[p3]);
    }

    private Vector CalculateCatmullRom(double t, Vector p1, Vector p2,
        Vector p3, Vector p4)
    {
        double t2 = t * t;
        double t3 = t2 * t;

        double b1 = 0.5 * (-t3 + 2 * t2 - t);
        double b2 = 0.5 * (3 * t3 - 5 * t2 + 2);
        double b3 = 0.5 * (-3 * t3 + 4 * t2 + t);
        double b4 = 0.5 * (t3 - t2);

        return (p1 * b1 + p2 * b2 + p3 * b3 + p4 * b4);
    }
}

```

Spline 类使用起来很简单。可以添加任意数量的点，**Spline** 会把它们连接起来。连线从 0 到 1 进行索引，线上的位置 0.5 将返回线的中点所在的位置。这样在前面的 **Tween** 类中使用样条就变得简单了。样条要求各个控制点均匀分布，以得到均匀的 t 值。

每个敌人都将获得一个新的 **Path** 类，以指导其穿过关卡。这个 **Path** 类是我们开发的射击游戏所特有的，所以应该在 **Shooter** 项目中创建它。

```

public class Path
{
    Spline _spline = new Spline();
}

```



```

Tween _tween;

public Path(List<Vector> points, double travelTime)
{
    foreach (Vector v in points)
    {
        _spline.AddPoint(v);
    }
    _tween = new Tween(0, 1, travelTime);
}

public void UpdatePosition(double elapsedTime, Enemy enemy)
{
    _tween.Update(elapsedTime);
    Vector position = _spline.GetPositionOnLine(_tween.Value());
    enemy.SetPosition(position);
}
}

```

Path 的构造函数接受时间和一个点的列表作为参数，它使用这些点创建样条和补间对象。**travelTime** 决定敌人使用多长的时间穿过样条定义的路径。**UpdatePosition** 方法更新补间，并从样条获得一个新位置，以便重新放置敌人。下面的代码修改 **Enemy** 类，使其使用 **Path** 类。

```

public Path Path { get; set; }
public void Update(double elapsedTime)
{
    if (Path != null)
    {
        Path.UpdatePosition(elapsedTime, this);
    }
    if (_hitFlashCountDown != 0)
    {
        _hitFlashCountDown = Math.Max(0, _hitFlashCountDown - elapsedTime);
        double scaledTime = 1 - (_hitFlashCountDown / HitFlashTime);
        _sprite.SetColor(new Engine.Color(1, 1, (float)scaledTime, 1));
    }
}
}

```

现在所有的敌人都具有路径。因为路径将会定义敌人从哪里开始出现，可以从 **EnemyDef** 中删除 **StartPosition** 变量。敌人可以穿过关卡，但是首先需要给它们提供一个路径。

在 `EnemyManager` 中, 创建一个敌人后, 需要给它提供路径。下面的代码为 `cannon_fodder` 敌人类型提供了一个从右向左的路径, 当到达中间位置时, 它向上运动。敌人走完整个路径的时间为 10s。

```
private Enemy CreateEnemyFromDef(EnemyDef definition)
{
    Enemy enemy = new Enemy(_textureManager, _effectsManager);
    //enemy.SetPosition(definition.StartPosition); <- this line can be
    removed
    if (definition.EnemyType == "cannon_fodder")
    {
        List<Vector> _pathPoints = new List<Vector>();
        _pathPoints.Add(new Vector(1400, 0, 0));
        _pathPoints.Add(new Vector(0, 250, 0));
        _pathPoints.Add(new Vector(-1400, 0, 0));

        enemy.Path = new Path(_pathPoints, 10);
    }
    else
    {
        System.Diagnostics.Debug.Assert(false, "Unknown enemy type.");
    }

    return enemy;
}
```

现在, 通过编辑 `EnemyManager` 的构造函数, 可以定义一个更加有趣的关卡。

```
public EnemyManager(TextureManager textureManager, EffectsManager
effectsManager, int leftBound)
{
    _textureManager = textureManager;
    _effectsManager = effectsManager;
    _leftBound = leftBound;

    _textureManager = textureManager;
    _effectsManager = effectsManager;
    _leftBound = leftBound;
    _upComingEnemies.Add(new EnemyDef("cannon_fodder", 30));
    _upComingEnemies.Add(new EnemyDef("cannon_fodder", 29.5));
    _upComingEnemies.Add(new EnemyDef("cannon_fodder", 29));
    _upComingEnemies.Add(new EnemyDef("cannon_fodder", 28.5));
}
```

```

_upComingEnemies.Add(new EnemyDef("cannon_fodder", 25));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 24.5));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 24));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 23.5));

_upComingEnemies.Add(new EnemyDef("cannon_fodder", 20));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 19.5));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 19));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 18.5));

// Sort enemies so the greater launch time appears first.
_upComingEnemies.Sort(delegate(EnemyDef firstEnemy, EnemyDef
secondEnemy)
{
    return firstEnemy.LaunchTime.CompareTo(secondEnemy.LaunchTime);
});
}

```

现在，敌人使用路径来描述如何在关卡中移动，所以不再需要每个敌人定义的初始位置。这意味着需要重新编写 `EnemyDef` 类。

```

public class EnemyDef
{
    public string EnemyType { get; set; }
    public double LaunchTime { get; set; }

    public EnemyDef()
    {
        EnemyType = "cannon_fodder";
        LaunchTime = 0;
    }

    public EnemyDef(string enemyType, double launchTime)
    {
        EnemyType = enemyType;
        LaunchTime = launchTime;
    }
}

```

再次运行代码，可以看到一队敌人在屏幕的上半部分曲折移动，如图 10-17 所示。

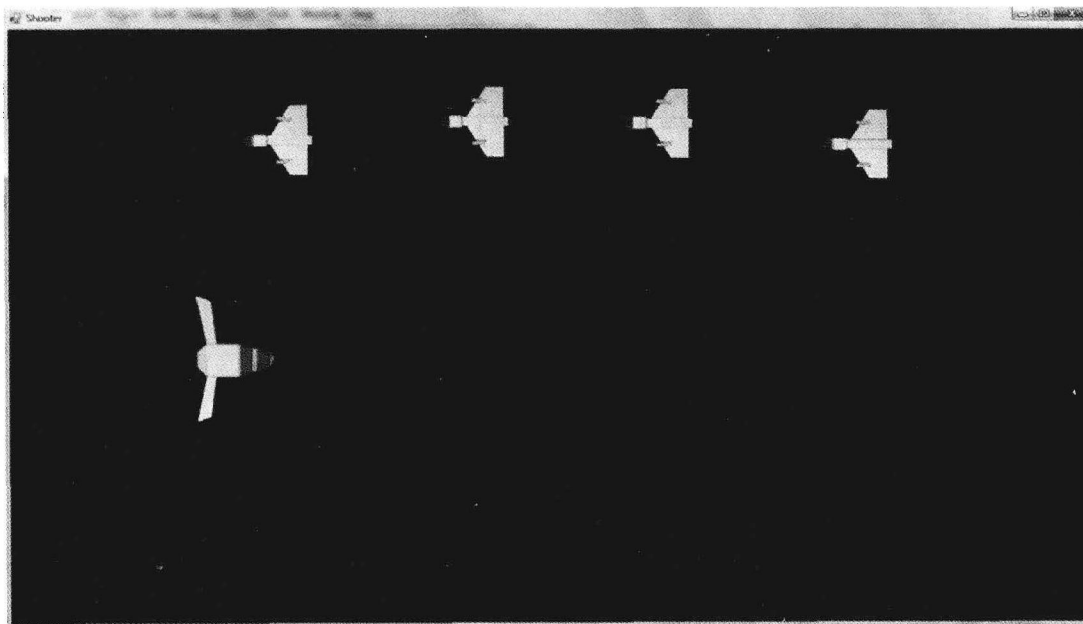


图 10-17 更有趣的关卡

现在可以再添加几种敌人类型，让关卡变得更好玩。

```
private Enemy CreateEnemyFromDef(EnemyDef definition)
{
    Enemy enemy = new Enemy(_textureManager, _effectsManager);

    if (definition.EnemyType == "cannon_fodder")
    {
        List<Vector> _pathPoints = new List<Vector>();
        _pathPoints.Add(new Vector(1400, 0, 0));
        _pathPoints.Add(new Vector(0, 250, 0));
        _pathPoints.Add(new Vector(-1400, 0, 0));
        enemy.Path = new Path(_pathPoints, 10);
    }
    else if (definition.EnemyType == "cannon_fodder_low")
    {
        List<Vector> _pathPoints = new List<Vector>();
        _pathPoints.Add(new Vector(1400, 0, 0));
        _pathPoints.Add(new Vector(0, -250, 0));
        _pathPoints.Add(new Vector(-1400, 0, 0));

        enemy.Path = new Path(_pathPoints, 10);
    }
    else if (definition.EnemyType == "cannon_fodder_straight")
```

```

{
    List<Vector> _pathPoints = new List<Vector>();
    _pathPoints.Add(new Vector(1400, 0, 0));
    _pathPoints.Add(new Vector(-1400, 0, 0));

    enemy.Path = new Path(_pathPoints, 14);
}
else if (definition.EnemyType == "up_1")
{
    List<Vector> _pathPoints = new List<Vector>();
    _pathPoints.Add(new Vector(500, -375, 0));
    _pathPoints.Add(new Vector(500, 0, 0));
    _pathPoints.Add(new Vector(500, 0, 0));
    _pathPoints.Add(new Vector(-1400, 200, 0));

    enemy.Path = new Path(_pathPoints, 10);
}
else if (definition.EnemyType == "down_1")
{
    List<Vector> _pathPoints = new List<Vector>();
    _pathPoints.Add(new Vector(500, 375, 0));
    _pathPoints.Add(new Vector(500, 0, 0));
    _pathPoints.Add(new Vector(500, 0, 0));
    _pathPoints.Add(new Vector(-1400, -200, 0));
    enemy.Path = new Path(_pathPoints, 10);
}
else
{
    System.Diagnostics.Debug.Assert(false, "Unknown enemy type.");
}

return enemy;
}

```

每种敌人都有一个有趣的路径，可以放到一起来形成一个更加有趣的关卡。下面所示是 **EnemyManager** 构造函数中用于设置新的 Level 的代码。

```

public EnemyManager(TextureManager textureManager, EffectsManager
effectsManager, int leftBound)
{
    _textureManager = textureManager;
    _effectsManager = effectsManager;
    _leftBound = leftBound;
}

```

```

_upComingEnemies.Add(new EnemyDef("cannon_fodder", 30));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 29.5));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 29));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 28.5));

_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 30));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 29.5));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 29));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 28.5));

_upComingEnemies.Add(new EnemyDef("cannon_fodder", 25));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 24.5));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 24));
_upComingEnemies.Add(new EnemyDef("cannon_fodder", 23.5));

_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 20));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 19.5));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 19));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_low", 18.5));

_upComingEnemies.Add(new EnemyDef("cannon_fodder_straight", 16));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_straight", 15.8));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_straight", 15.6));
_upComingEnemies.Add(new EnemyDef("cannon_fodder_straight", 15.4));
_upComingEnemies.Add(new EnemyDef("up_1", 10));
_upComingEnemies.Add(new EnemyDef("down_1", 9));
_upComingEnemies.Add(new EnemyDef("up_1", 8));
_upComingEnemies.Add(new EnemyDef("down_1", 7));
_upComingEnemies.Add(new EnemyDef("up_1", 6));
// Sort enemies so the greater launch time appears first.
_upComingEnemies.Sort(delegate(EnemyDef firstEnemy, EnemyDef
secondEnemy)
{
    return firstEnemy.LaunchTime.CompareTo(secondEnemy.LaunchTime);
});
}

```

试试这个关卡，可能会发现关卡的难度有点高。现在可以根据需要修改代码，使游戏简单一些。试试提高玩家发射子弹的速度或者飞船的速度。

10.3.9 敌人攻击

敌人沿着有趣的路径通过关卡，但是它们还是比较被动的。玩家可以向它们狂轰乱炸，而它们不会采取任何行动。敌人应该通过某种方法保护自己，本节就赋予它们这种能力。

已有的 **BulletManager** 类只处理玩家的子弹。敌人的子弹将只影响玩家，而玩家的子弹将只影响敌人。因此，创建单独的子弹列表最简单。这意味着需要使 **BulletManager** 的一些函数更加通用，以接受一个子弹列表。

```
public class BulletManager
{
    List<Bullet> _bullets = new List<Bullet>();
    List<Bullet> _enemyBullets = new List<Bullet>();

    // Code omitted

    public void Update(double elapsedTime)
    {
        UpdateBulletList(_bullets, elapsedTime);
        UpdateBulletList(_enemyBullets, elapsedTime);
    }

    public void UpdateBulletList(List<Bullet> bulletList, double
elapsedTime)
    {
        bulletList.ForEach(x => x.Update(elapsedTime));
        CheckOutOfBounds(_bullets);
        RemoveDeadBullets(bulletList);
    }

    private void CheckOutOfBounds(List<Bullet> bulletList)
    {
        foreach (Bullet bullet in bulletList)
        {
            if (!bullet.GetBoundingBox().Intersects(_bounds))
            {
                bullet.Dead = true;
            }
        }
    }

    private void RemoveDeadBullets(List<Bullet> bulletList)
    {
        for (int i = bulletList.Count - 1; i >= 0; i--)
```

```

        {
            if (bulletList[i].Dead)
            {
                bulletList.RemoveAt(i);
            }
        }
    }

    internal void Render(Renderer renderer)
    {
        _bullets.ForEach(x => x.Render(renderer));
        _enemyBullets.ForEach(x => x.Render(renderer));
    }

```

上面的代码为敌人的子弹引入了另一个列表，现在还需要有一个让敌人发射子弹的函数和一个检查子弹是否击中玩家的函数。

```

public void EnemyShoot(Bullet bullet)
{
    _enemyBullets.Add(bullet);
}

public void UpdatePlayerCollision(PlayerCharacter playerCharacter)
{
    foreach (Bullet bullet in _enemyBullets)
    {
        if (bullet.GetBoundingBox().Intersects(playerCharacter.
            GetBoundingBox()))
        {
            bullet.Dead = true;
            playerCharacter.OnCollision(bullet);
        }
    }
}

```

UpdatePlayerCollision 与已有的 UpdateEnemyCollision 方法非常类似，最终它们应该被合并到一起。但是在游戏开发的这一次迭代中，使两者独立会更加简单一些。PlayerCharacter 类需要一个新的接受子弹对象的 OnCollision 方法。

```

internal void OnCollision(Bullet bullet)
{
    _dead = true;
}

```

现在 PlayerCharacter 有两个碰撞方法：一个用于子弹，一个用于敌人。PlayerCharacter

在接触到敌人或者子弹后将会死亡，所以这些方法是冗余的。以这种方法编写它们是为了使扩展游戏变得更加简单。知道玩家角色与什么发生碰撞很重要。如果玩家具有生命值，那么与敌人发生碰撞导致的伤害可能比与子弹发生碰撞导致的伤害更大。如果添加了导弹、地雷或者各种增加玩家能力的物品，那么也可以有额外的碰撞检测方法来处理与它们的碰撞。

射击游戏是一种非常严格的游戏。如果玩家与飞船发生碰撞，将立即失败。同样，如果与子弹发生碰撞，玩家也会失败。现在 **BulletManager** 需要在 **Level** 类的 **Update** 循环中添加一个额外的调用，以测试敌人的子弹是否击中了玩家。

```
private void UpdateCollisions()
{
    _bulletManager.UpdatePlayerCollision(_playerCharacter);
}
```

为使敌人可以使用新增加的发射子弹的功能，需要使它们能够访问 **BulletManager** 类。可以把 **BulletManager** 传递给 **EnemyManager**，并借此传递给各个敌人。下面的代码中从 **Level** 类的构造函数把它传递给了 **EnemyManager**。

```
public Level(Input input, TextureManager textureManager, Persistent
GameData gameData)
{
    _input = input;
    _gameData = gameData;
    _textureManager = textureManager;
    _effectsManager = new EffectsManager(_textureManager);
    _enemyManager = new EnemyManager(_textureManager, _effectsManager,
    _bulletManager, -1300);
}
```

在下面的代码中，**EnemyManager** 存储了对 **BulletManager** 的引用，并在构造敌人的时候使用这个引用。

```
BulletManager _bulletManager;
public EnemyManager(TextureManager textureManager, EffectsManager
effectsManager, BulletManager bulletManger, int leftBound)
{
    _bulletManager = bulletManger;

    // Code omitted

private Enemy CreateEnemyFromDef(EnemyDef definition)
{
    Enemy enemy = new Enemy(_textureManager, _effectsManager,
    _bulletManager);
}
```

现在敌人有了 **BulletManager** 类，可以借助它开始发射子弹了。现在的问题是，敌人应该在什么时候发射子弹？它们不能在每一帧中都发射子弹，那样的话游戏就太难了。敌人

也不应该同时发射子弹，那样的话游戏也会太难。这里的技巧是为每个敌人随机设置发射时间。

```

public double MaxTimeToShoot { get; set; }
public double MinTimeToShoot { get; set; }
Random _random = new Random();
double _shootCountDown;

public void RestartShootCountDown()
{
    _shootCountDown = MinTimeToShoot + (_random.NextDouble() *
MaxTimeToShoot);
}

BulletManager _bulletManager;
Texture _bulletTexture;
public Enemy(TextureManager textureManager, EffectsManager
effectsManager, BulletManager bulletManager)
{
    _bulletManager = bulletManager;
    _bulletTexture = textureManager.Get("bullet");
    MaxTimeToShoot = 12;
    MinTimeToShoot = 1;
    RestartShootCountDown();

    // Code omitted
public void Update(double elapsedTime)
{
    _shootCountDown = _shootCountDown - elapsedTime;
    if (_shootCountDown <= 0)
    {
        Bullet bullet = new Bullet(_bulletTexture);
        bullet.Speed = 350;
        bullet.Direction = new Vector(-1, 0, 0);
        bullet.SetPosition(_sprite.GetPosition());
        bullet.SetColor(new Engine.Color(1, 0, 0, 1));
        _bulletManager.EnemyShoot(bullet);
        RestartShootCountDown();
    }
}

```

创建敌人后，它会通过设置一个计时器来确定下一次发射子弹的时间。计时器使用 C# 的 `Random` 类和最小及最大时间来设置，计时器应该在最小值和最大值之间。所有的飞船将在不同的时间发射子弹。`RestartShootCountDown` 方法设置敌人下一次发射子弹的时间。

`Math.NextDouble` 返回 0~1 之间的一个随机数,它将被放大到 `MinTimeToShoot` 和 `MaxTimeToShoot` 成员变量之间。

`Update` 循环减小 `_shootCountDown` 的值,当这个值等于或者小于 0 时,敌人就会发射子弹。敌人的子弹被设为比玩家的子弹慢,其方向与玩家的子弹相反,并且颜色被设为红色,以便与玩家的子弹进行区分。当敌人发射子弹后, `_shootCountDown` 计时器将被重置。

敌人朝着屏幕的左侧发射子弹。你可能想让游戏变得更加困难一些,使敌人瞄着玩家发射子弹。为此,敌人必须具有对 `PlayerCharacter` 的引用。这样只需要确定玩家相对于敌人飞船的方向。如果决定向敌人增加瞄准能力,下面这个简单的代码段可以提供帮助。

```
Vector currentPosition = _sprite.GetPosition();
Vector bulletDir = _playerCharacter.GetPosition() - currentPosition;
bulletDir = bulletDir.Normalize(bulletDir);
bullet.Direction = bulletDir;
```

对游戏的第二遍细化到此结束。敌人可以向玩家发射子弹,并按照有趣的方式四处移动。在消灭敌人时它们将会爆炸,显示出一个效果令人满意的火球。

10.4 继续迭代

在开发过程经历了两次基本的迭代后,我们有了一个很出色、但是很基础的卷轴射击游戏。还有很大的发挥空间,可以把这个项目开发成有自己的特色的游戏。现在这是你的项目,可以根据你自己的需要进行开发。如果感到有一些困惑,可以考虑下面的建议。

- 一个非常简单的第一步是引入一种新的敌人类型。只需要再添加一个 `else if` 语句,还可以修改路径或者生命值。完成这些修改之后,考虑创建一个新的敌人纹理,并把新敌人修改为使用该纹理。这样游戏一下子就有趣得多了。
- 在卷轴射击游戏中,得分很重要。得分可以使用 `Text` 类显示。每次玩家消灭一个敌人时,得分都应该增加。
- 添加声音也很简单。需要像本书前面那样创建一个声音管理器,并且可以为射击、爆炸和受到损伤等生成各种合适的声音。然后只需要确定爆炸、损伤和射击事件发生的位置,并调用声音管理器来播放正确的声音。主要的工作是在对象间传递声音管理器,以便可以在需要的地方使用它。
- 代码中存在大量的管理器和一些具有类似代码的函数。如果使这些管理器成为通用管理器,并移除重复的代码,整体的代码可以更加紧凑、更易于扩展。一个不错的起点是看看各个管理器使用了哪些类似的方法,然后考虑扩展 `Entity` 类,以便可以创建一个通用的 `EntityManager`。
- 游戏只有一个关卡,定义在 `EnemyManager` 构造函数中。这样做的扩展性不太好。一个好的入手项目是在一个文本文件中定义关卡。在启动时,程序可以读取文本并加载关卡定义。关卡定义可以十分简单,例如:

```
cannon_fodder, 30
```

```
cannon_fodder, 29.5  
cannon_fodder, 29  
cannon_fodder, 28.5
```

- 每一行都有一个敌人类型和一个启动时间，两者用逗号分隔。这样解析起来和读入到 `Level` 定义类中都十分简单。关卡数据可能应该存储在 `PersistentGameData` 类中。
- 加载了一个关卡后，创建一个新的关卡文件很简单，这样就一下子具有了创建多关卡游戏的可能。成功地完成了一个关卡后，不是返回到 `StartGameState`，而是返回到 `InnerGameState`，只不过开始使用下一个关卡。如果游戏有多个关卡，那么存储玩家在关卡中的进度是一个不错的功能。保存游戏数据的一个非常简单的方法是将玩家得分和当前关卡写入到一个文本文件中。
- 可以不让玩家以线性的方式(1、2、3、4)闯关，而是为玩家展示游戏世界大地图，让玩家选择接下来玩哪一个关卡。游戏世界大地图一般将所有的关卡表示为通过路径连接在一起的节点。《超级马里奥(Super Mario)》游戏系列中的一部分就使用了这种系统。通过使用大地图，很容易引入一些秘密的路径和关卡，只有玩家在前一个关卡中完成得特别出色时，才可以发现它们。
- 如果玩家被敌人飞船或者子弹击中，`PlayerCharacter` 死亡，游戏结束。更好的做法是让玩家飞船也具有生命值，使其可以像敌人一样每次受到一定的伤害。生命值可以用屏幕上的一个生命值条表示，每次玩家被击中一次，生命值条会缩短一些。还可以引入生命的概念：玩家一开始有几条生命，每条生命允许玩家再玩一次关卡，当生命数变为 0 时，玩家将输掉游戏。
- 游戏关卡越靠后，一般会越难。为了给玩家提供帮助，可以给他提供更好的武器，以及可以修复飞船遭受的损坏的物品。在卷轴射击游戏中，敌人会掉落一些增加玩家能力的物品。可以创建一个新的 `Item` 类并添加到场景中(可以通过 `EffectsManager` 实现)，以便玩家可以拾取。急救包可以恢复玩家的生命值。新武器可以造成更大的伤害，或者一次发射两发而不是一发子弹。
- 还可以添加通过其他按钮使用的辅助武器。例如，炸弹或者激光发射器可以作为发射次数有限的辅助武器。
- RPG 元素是增加深度感的一种极为流行的方式。敌人可以掉落金钱(或者可以在以后卖出的碎片)。在每一关过后，玩家可以购买新武器，升级现有武器，以至购买新的飞船。甚至还可以通过修改 `PlayerCharacter` 的 `_gunOffset` 成员，允许玩家在飞船上的不同位置安装武器。
- 可以进一步利用 RPG 元素，在游戏中添加解说。这可以在关卡中使用文本框和通过脚本确定的敌人和玩家的移动来实现。也可以把故事元素添加到大地图上，或者在完成每一关后添加。
- 关卡最后的大 boss 也是卷轴射击游戏中必不可少的一个元素。可以聚集几种不同类型的敌人来作为 boss。boss 的各个部分可以被击毁，但是只有当 boss 的全部部分都被击毁后，`PlayerCharacter` 才会获胜。

- 卷动太空背景很枯燥，加上一些动画显示的飞船残骸、远方的超新星和行星后，可以变得更加生动。卷动背景可以在任何时候修改，所以只需要做一点工作，很容易就能产生飞船朝着行星表面飞行的效果。
- 最后一个建议是，可以增加本地多玩家模式，这其实很简单，只需要把另一个游戏控制器或者键盘的输入重定向到第二个 `PlayerCharacter` 上。还需要修改一些逻辑，以便当一个玩家角色死亡后，另一个玩家还可以继续玩游戏。

第 11 章

创建自己的游戏

你可能有一些非常优秀的游戏想法想要用程序实现，所以选择了阅读本书。本章为创建游戏提供了一般性的建议，并介绍了各种游戏类型面临的挑战及应对方法。

本章将快速介绍游戏创建的过程，并简要介绍在开始编程时可能遇到的一些有趣的问题。本章描述了编写游戏时建议采取的步骤，并提供了一些算法的代码实现。在本书的配套光盘上，可以找到一个列表，其中针对这里介绍的游戏类型列出了一些阅读材料和相关资源。

11.1 项目管理

编写游戏很有趣，回报也很丰富，但是完成一个游戏项目要困难多了。本节将介绍开发项目时可以采用的方法。知道何时放弃一个游戏项目和何时坚持完成一个项目很重要。

开发游戏需要有一个想法，所以可以随身携带一个笔记本，任何时候脑海中涌现一个自己觉得非常好的想法时，就把它记下来。另外一个可以采取的做法是仔细分析自己喜欢玩的游戏，试着把游戏分成几个系统，写下自己喜欢的游戏特征。如果看到一个好的 GUI、特效、谜题或游戏交互，试着自己创建它们。重新创建现有游戏的各个部分是提高自己技能的一种好方法。

当有了一个优秀的游戏想法时，就可以进入下一个阶段：可行性测试。你有创建这个游戏的资源吗？如果游戏需要几小时的全动态视频(full-motion video)、几百小时的游戏时间或几千个具有完整动作的怪物，那么除非你周围有一些水平极高的朋友，或者你有充足的资金，否则可能需要把目标订得小一些。理想的游戏应该可以对你的能力提出挑战，但是同时仍然是可以完成的。如果刚刚接触游戏开发，那么就从一些能够在几天内完成的小

项目开始(例如 *Pong* 或 *Space Invaders*), 逐渐开始开发一些要求更高的项目。

在对游戏的设想通过了可行性测试后, 需要把它分解为几个可以管理的部分。这是设计中的较高层次, 但是很有趣。你对开发一个出色的游戏有了一个模糊的想法, 但是现在需要把这个想法落到实地, 确定各个部分如何组合在一起。一个不错的做法是在纸上用方框勾勒出游戏中的各个类和系统, 并使用箭头表示出它们之间的关系和交互。把这个粗略的高层面计划贴到墙上或者工作区的某个地方很有帮助, 当你开始仔细研究项目的细节时, 它可以提供指导。如果觉得自己还没有达到绘制这种方框和箭头的阶段, 那么再运用想象力, 在脑中构思一遍对游戏的想法。启动游戏后出现什么? 开始第一个关卡后呢? 可以采取什么操作? 游戏世界对这些操作做出什么反应?

假设希望创建这样一个游戏: 必须通过唱不同的歌让角色在屏幕上执行不同的操作和与环境交互。分解这种设计时, 最先映入眼帘的第一个大系统是玩家输入, 所以需要进行研究, 以便能够检测出玩家是否在唱歌, 以及确定玩家在唱什么歌。唱歌部分需要一个完全属于自己的小项目。至少, 需要有一个系统通过麦克风从玩家那里获得声音输入, 另外一个系统确定玩家唱的不同的歌。必须检查游戏中的每个角色, 看玩家当前唱的歌对它们是不是重要。还需要定义角色可以采取的动作、游戏的输赢状态, 以及表示游戏世界的方法。这个只有一行的游戏设计提出了这样一些问题。下一次在纸上完成对游戏的设想时, 应该会得出许多需要回答的问题和一系列需要研究的主题。

下一个阶段是确定设计阶段提出的问题和任务的优先级。并不是每个游戏都有需要进行研究的特征, 但是如果你的游戏要求你事先做一些调查研究, 那么就要首先完成研究工作。研究可能很检查, 只需要在 Google 上进行快速地搜索, 也可能需要咨询经验丰富的人或者阅读过相关书籍或文章的人。可以使用小的玩具程序来尝试不同的研究想法, 如果成功了, 就可以对这种想法进行整理, 然后整合到最终的项目中。

完成了所有的研究工作后, 询问自己这样一个问题: “最少做哪些工作就可以得到一个能基本工作的游戏?” 然后做这些工作, 如果可以的话, 最好一次性完成这些工作。有了游戏的第一次迭代结果, 返回去逐渐改进游戏就容易多了。如果有几天时间没有参与开发, 则很难拾起一个半成品游戏。

游戏编程是需要时间的, 通常每次开发最少需要 4 个小时左右的时间, 特别是当你特别专注于手头的任务而忘了时间时更是如此。计划任务列表(to-do list)是指导开发工作的一个好方法。建议在计划任务的下边写上开始完成这个任务所需要执行的最少步骤, 例如:

- (1) 设置开发环境, 并创建一个基本的窗口和游戏循环。

打开 Visual Studio, 创建一个新项目, 命名为 Project Minstrel。

- (2) 构建 Simple Song Classifier 框架类。

添加 SongClassifier.cs 文件, 并添加 System.Speech 库。

计划任务后面的短句子是可以立即执行的工作。开始第一步以后, 应该完成整个计划任务。不要在每次开始开发项目后都积留下未完成的计划任务, 然后在下一次丢弃上一次的计划任务列表并重新开始工作。特别重要的工作是可以复制的。

如果看不到项目的出路, 不要害怕停止项目的开发。每个项目(即使是未完成的项目)都可以教给你一些东西(即使学到的是“我再也不那么做了”, 也是有帮助的)。对于喜欢的项目, 你会实现更多的特性, 因为你是因为喜欢而开发, 而不是因为自己已经投入了大量

的时间，而不得不继续开发。

如果希望得到反馈或者帮助，Internet 上有许多游戏编程社区，他们很乐意尝试你的游戏并提出建议和意见(比如 Independent Game Source, <http://forums.tigsource.com/index.php>。本书配套光盘上还提供了其他社区的地址)。在线社区还会定期举办游戏编程竞赛，并且如果你想寻找其他程序员或美术人员进行合作开发，他们也提供了这种服务。

11.2 显示方法

游戏主要分为两类：诸如《*Super Mario*(超级马里奥)》、《*Tetris*(俄罗斯方块)》和《*Pokémon*(口袋妖怪)》的 2D 游戏，以及诸如《*Quake*(雷神之锤)》、《*Fallout 3*(辐射 3)》和《*Grand Theft Auto*(侠盗猎车手)》的 3D 游戏。一般来说，2D 游戏编程要相对简单一些。

11.2.1 2D 游戏

本书已经对 2D 游戏做出了全面的介绍，所以如果你想开发的游戏是一个 2D 游戏，那么已经有了相当大的优势。2D 游戏中经常使用像素风格的图像，但是却并不是一定如此，在呈现一个 2D 游戏方面仍然有许多创新的空间。图像可以采用剪纸画、粉笔画、矢量图形(使用 OpenGL 的 GL_LINE 绘制模式)、抽象画或者黑色轮廓图(黑色轮廓图的优势在于十分易于绘制)。2D 游戏中涉及的数学一般比 3D 数学要简单一些，但是这还是要看具体的游戏。大多数程序员在学习 3D 游戏编程之前会先尝试编写 2D 游戏，很多关于游戏开发的知识与游戏是 2D 还是 3D 是无关的。

11.2.2 3D 游戏

游戏程序员的重要技能之一是能够独立地研究和学习新的编程和开发技术。有时候一个游戏想法会用到新的、程序员还不熟悉的技术(例如，在游戏中想要使用一根绳子，此时就需要研究模拟绳子的方式)，另外一些时候则可能只是为了研究的乐趣或挑战，但是在这个过程中，发现了一个关于游戏的新想法。现在的大型游戏大部分都是 3D 的，仅这一点就让人有了学习 3D 游戏开发的动力。

3D 图形编程涉及庞杂而令人感到畏惧的数学、技术和术语，所以每次只学习一些可以掌握的知识，然后逐渐掌握 3D 图形编程是一种很重要的方法。打算把自己的第一个项目做得与最新的 FPS 射击游戏一样，那只会让自己失望。更好的做法是设计一个可以快速实现的项目，并且使这个项目可以成为更复杂的项目的基础。对于 3D 图形，在屏幕上显示一个盒子是一个可以实现的第一步，要是这个盒子可以旋转就更好了。这之后的一个中间步骤可以是重新创建第 10 章中的卷轴射击游戏，但是使用 3D 模型而不是精灵。

本书的代码示例在 Form.cs 文件中经常使用一个叫做 Setup2DGraphics 的函数。编写一个对应的 Setup3DGraphics 并不困难。

```
private void Setup3DGraphics(double width, double height)
{
    double halfWidth = width / 2;
```

```
double halfHeight = height / 2;
Gl.glMatrixMode(Gl.GL_PROJECTION);
Gl.glLoadIdentity();
Glu.gluPerspective(90, 4 / 3, 1, 1000);
Gl.glMatrixMode(Gl.GL_MODELVIEW);
Gl.glLoadIdentity();
}
```

这段代码与 `Setup2DGraphics` 调用几乎一样,但是使用了函数 `gluPerspective` 而不是 `glOrtho`。`gluPerspective` 函数的第一个参数是一个视野,第二个参数是宽高比,最后两个参数分别是近平面和远平面。这与查看 3D 场景的摄像机镜头的参数类似。

如果调用 `Setup3DGraphics` 而不是 `Setup2DGraphics`,那么就开始了 3D 游戏编程。下面的游戏状态使用 OpenGL 的立即模式渲染一个 3D 的金字塔,得到的结果如图 11-1 所示。确保在运行此状态之前,将所有的 `Setup2Dgraphics` 调用替换成了 `Setup3DGraphics` 调用。

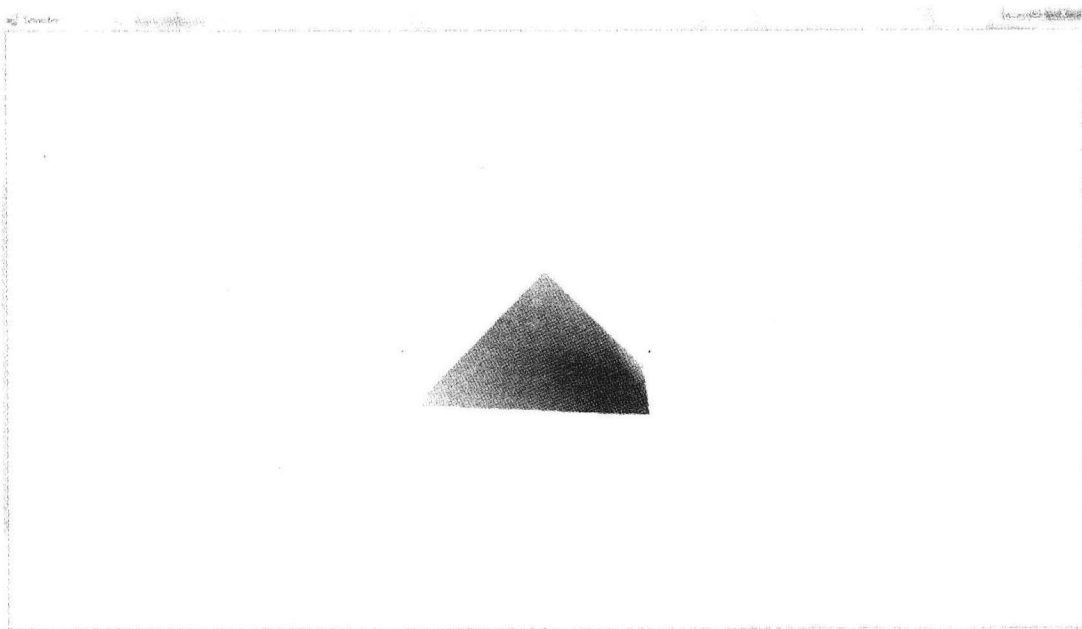


图 11-1 3D 金字塔

```
class Test3DState : IGameObject
{
    public Test3DState(){}
    public void Update(double elapsedTime){ }

    public void Render()
    {
        Gl.glDisable(Gl.GL_TEXTURE_2D);
        Gl.glClearColor(1, 1, 1, 0);
    }
}
```

```
Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);

// This is a simple way of using a camera
Gl.glMatrixMode(Gl.GL_MODELVIEW);
Gl.glLoadIdentity();
Vector cameraPosition = new Vector(-75, 125, -500); // half a meter back
    on the Z axis
Vector cameraLookAt = new Vector(0, 0, 0); // make the camera look at the
    world origin.
Vector cameraUpVector = new Vector(0, 1, 0);
Glu.gluLookAt( cameraPosition.X, cameraPosition.Y,
    cameraPosition.Z,
                cameraLookAt.X, cameraLookAt.Y,
    cameraLookAt.Z,
                cameraUpVector.X, cameraUpVector.Y,
    cameraUpVector.Z);

Gl.glBegin(Gl.GL_TRIANGLE_FAN);
{
    Gl.glColor3d(1, 0, 0);
    Gl.glVertex3d(0.0f, 100, 0.0f);

    Gl.glColor3d(0, 1, 0);
    Gl.glVertex3d(-100, -100, 100);

    Gl.glColor3d(0, 0, 1);
    Gl.glVertex3d(100, -100, 100);

    Gl.glColor3d(0, 1, 0);
    Gl.glVertex3d(100, -100, -100);

    Gl.glColor3d(0, 0, 1);
    Gl.glVertex3d(-100, -100, -100);

    Gl.glColor3d(0, 1, 0);
    Gl.glVertex3d(-100, -100, 100);
}
Gl.glEnd();
}
```

代码中新增加了一个 `Glu.gluLookAt` 函数调用, 该函数设置了 3D 摄像机的位置以观看场景。摄像机的位置由 3 个向量组成: 第一个向量是摄像机在世界中的位置; 第二个向量是

摄像机应该指向的位置；第三个向量指向上方。如果通过将 Y 分量设为-1 来逆转向上的向量，那么图 11-1 所示的场景将会倒转过来。

如果想学习更多关于如何在游戏使用 3D 图形的知识，建议执行下面的步骤。

- 使用 OpenGL 的立即模式和 GL_QUADS 建立一个 3D 场景，并绘制一个立方体。
- 将盒子的顶点写入到一个文件中，然后再载入它们。这是你的第一个简单文件格式。
- 更新代码，使其使用顶点缓冲区，而不是立即模式。
- 研究索引缓冲区，并更新代码和文件格式，以使用这些索引缓冲区。
- 研究.obj 文件格式。这是一种纯文本格式。编写一个简单的程序，把.obj 文件加载到一组索引和顶点缓冲区中。忽略材质和参数曲面信息(提示：记住 OpenGL 要求表面以顺时针顺序传入)。

上述步骤足以开始进行 3D 游戏编程。当可以加载简单的.obj 文件后，就可以使用已有的 Matrix 类移动和变换它们。

当准备好操作 3D 模式时，首先从.MD2 文件格式开始。*Quake II*使用这种文件格式来存储敌人和角色。从 Internet 上可以下载几百种免费的测试文件来测试自己的加载器。进入这个阶段后，你可能对自己接下来需要学习 3D 图形编程的哪一部分有了更清晰的认识。

11.3 游戏类型

不同的游戏具有不同的大小和结构，可以把它们大致划分为不同的类型。相同类型的游戏一般采用类似的方式进行编程。

11.3.1 文字类游戏

开始游戏编程时，文字类游戏是最简单的一类游戏，因为它们不需要图像。程序员可以将精力完全集中在故事情节上。文字类游戏也分为很多种，但是最常见的是交互式的小游戏。下面所示是在一个简单的交互式小说类游戏中进行交互的片段。

```
You are in a dark room, you can see nothing.
>Search
You search around in the dark, your fingers brush against something
  metallic.
>Examine metallic thing
It's hard to make out in the dark but it seems to be a flashlight.
>Turn on flashlight.
The flashlight flickers to life.
```

交互式的小游戏包含两个主要的部分：保存世界描述的一个数据结构和解释用户输入的一个解析器。描述世界的数据结构可能非常简单。

```
class Room
{
    public string Description {get; set;}
```

```
public List<Room> Exits{get; set;}
public Room(string description, List<Room> exits)
{
    Exits = exits;
    Description = description;
}
}
```

Room 类是游戏世界中的一个地方，它有一个描述自己的文本字符串，以及一个由通往不同房间的出口组成的列表。玩家可以得到每个房间的描述，并通过出口在地图上四处走动。在解决谜题后可以添加和删除出口。

解析器是匹配模式的一段代码。用户可以在控制台输入 **Look**，然后按下 **Enter** 键，以观察整个房间。代码中对这些动作的处理方式是：

```
if (_input == "look")
{
    // The look function writes the description to the console.
    _currentRoom.Look();
}
```

更复杂的语句需要更多的解析，例如 **use hammer on doorstep**。下面的代码有一点理想化，但是从中可以看出分解用户命令的方式。

```
if(_input.StartsWith("use"))
{
    _input = _input.RemoveSubstring("use")
    int result = _input.Find("on")

    if( result == -1) break; // pattern not recognized

    string firstParameter = input.SubString(0, result)
    string secondParameter = input.SubString(result + "on".Length(),
        input.Length());

    // the use function will use these strings to look up objects
    Use(firstParameter, secondParameter);
}
```

这段代码假定所有的输入文本都采用小写。如果玩家输入了大写字符，函数将无法工作。解决这种问题的方法是使用下面的代码处理输入字符串：

```
_input = _input.ToLower()
```

为了在 **Visual Studio** 中创建一个文字类冒险游戏，选择 **Console Application** 而不是 **Windows Forms Application**，如图 11-2 所示。

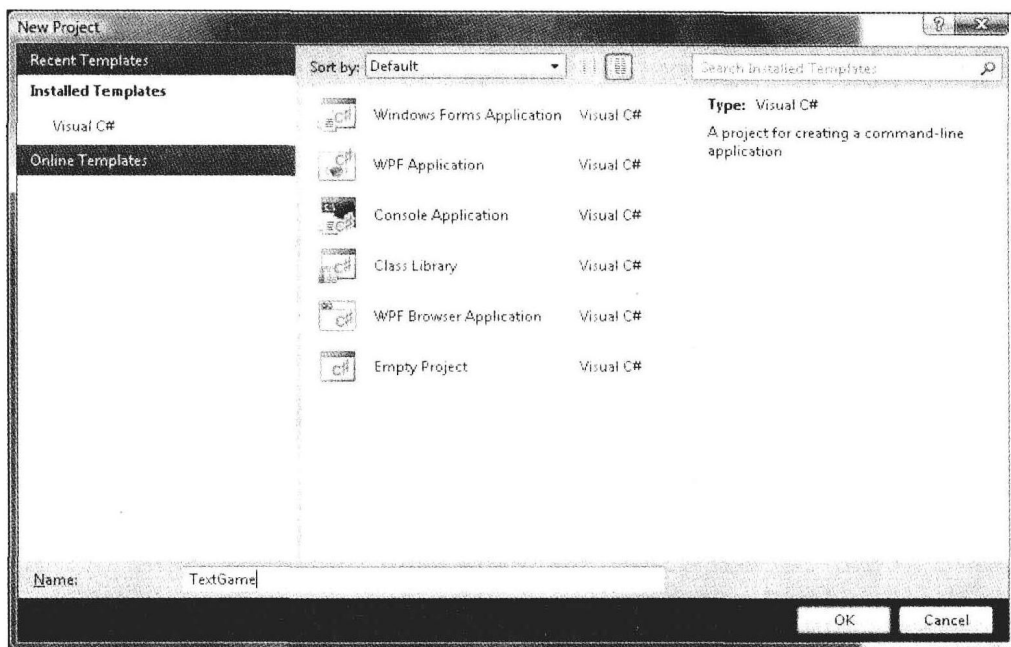


图 11-2 创建一个控制台应用程序

启动控制台项目后，可以使用 `Windows.Console.ReadLine()` 从控制台读取数据，使用 `Windows.Console.WriteLine()` 向控制台写入数据。

关于创建交互式小说的一个好资源是 Inform Programming 语言(<http://inform7.com/>)。它允许用户输入接近自然语言的英语文本来描述一个交互式小说世界。

11.3.2 益智游戏

在开发自己的第一个游戏时，益智游戏是一个不错的选择。益智游戏一般需要较少的资源，并且相对来说理解起来也很简单。流行的益智游戏包括拼图、数独、Boggle、俄罗斯方块和宝石迷阵(Bejeweled)。许多非常流行的游戏就是以一些益智游戏的集合作为基础，例如 PC 游戏 7th Guest，DS 上的《Professor Layton(雷顿教授)》游戏，甚至于 Brain Training。这些游戏理解起来和玩起来都很简单，但是在开发它们时程序员仍然会感受到挑战性和乐趣。

第一次进行游戏编程时，俄罗斯方块是一个很不错的选择。初看起来，它很简单，但是开始分析游戏组成时会看到一些有意思的挑战。俄罗斯方块的游戏目标是构成一个由方块组成的没有间隙的水平行。创建这样一行后，该行会消失，其上的方块会落下，取代该行。不同颜色和形状的方块从游戏区域的顶端落下，玩家必须旋转和移动方块来尽可能多地构成水平无间隙行。随着游戏逐渐进行，落下的方块的速度会加快。

创建俄罗斯方块的第一个目标是使一些 1×1 的小方块从游戏区域的顶端落到底端并停止移动。继续落下的方块应该堆积在已经落下的方块之上。图 11-3 显示了最初的这个目标。当这些代码可以工作后，就完成了编写俄罗斯方块游戏的主要工作。

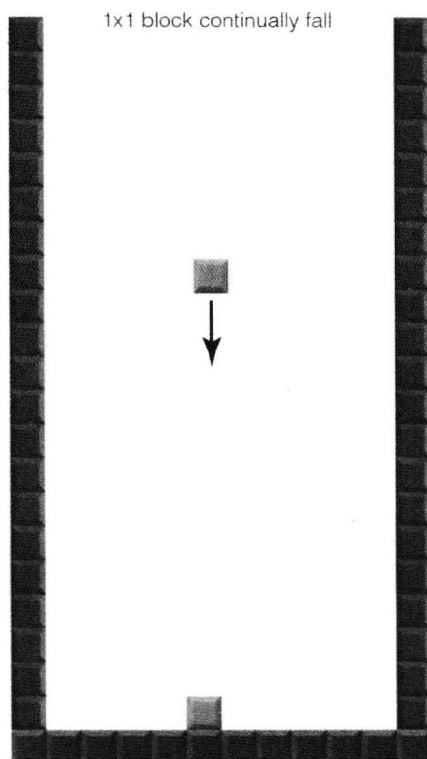


图 11-3 创建俄罗斯方块游戏的第一步

开发游戏的下一个阶段是检测玩家什么时候输掉游戏。当方块堆积得太高，以至于接触到游戏区域的顶端时，玩家输掉游戏。接着应该实现基本的玩家输入，以允许玩家将方块向左或向右移动一个单位。方块停止下落后，玩家不能再移动方块，他们也不能把方块移动到游戏区域以外。俄罗斯方块没有结束的关卡，还有其他一些与其类似的经典游戏，它们的特征都是：玩家不可能获胜，只能尽力延缓失败的时间。没有明确的胜利状态可以编程，但是游戏的主要目标之一是构成完整的一行方块。程序应该能够检测到完成的行并删除它。行之上的方块应该下落一层。这也非常适合引入一个得分系统，并在屏幕上显示玩家的得分。

下一步是创建俄罗斯方块的构成部分，也就是各个方块。建议把这些方块设计为前面阶段中使用过的 1×1 方块，用户必须可以移动这些方块，但是我并没有尝试将所有的精灵围绕方块的中心旋转 90° ，而是针对可以旋转方块的每个方向，存储了方块的 4 个版本。当玩家按下旋转按键时，可以使用一个新的版本替换当前版本，其中的新版本是方块旋转后应该变成的版本。不要使 1×1 的方块不断地从游戏区域的顶端落到底部，而是应该创建一些随机的方块。

俄罗斯方块的最后一个挑战是链式反应规则。图 11-4 显示了这种规则。有时候完成一行后，该行上方的方块会悬空停留在原来的位置。这些方块应该落下去。当落下去后，这些方块可能会使另外一行完整，从而引起链式反应。

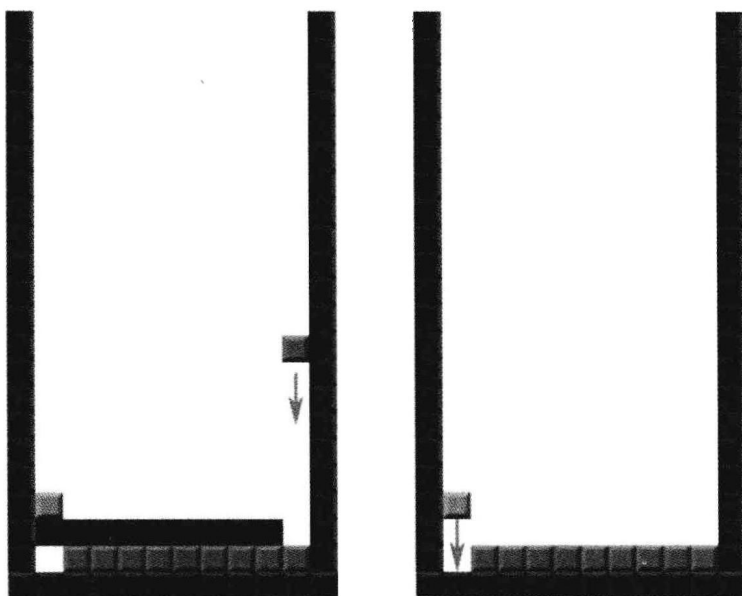


图 11-4 链式反应规则

链式规则可能不易处理。在一行完成后，需要搜索游戏区域，查看是否有可以落下的自由方块。然后使这些方块落下，在恢复游戏之前，需要检查所有的行，看是否有行完成。如果按照这里的粗略介绍进行编程，可以比较轻松地编写出一个不太差的俄罗斯方块游戏。

11.3.3 第一人称射击游戏

第一人称射击游戏可能让程序员感到有点畏惧，因为它们严要求程序员熟悉大量复杂的数学知识。3D 游戏的许多复杂性就在于计算出玩家当前可以看到的对象的算法。如果程序知道玩家可以看到什么，就不需要渲染他们看不到的对象。这可以显著提高游戏速度。这些算法通常是某种类型的空间划分，但是只有当你可以编写基本的 FPS 时，才需要学习这些算法。

下面列出了开始编写 FPS 游戏时可以采取的步骤。

- 创建一个简单的 3D 场景，使其具有可以改变位置的摄像机类(使用 `Glu.gluLookAt`)。
- 在摄像机下绘制一个大正方形，使其看上去像是一个地板。可以选择为这个正方形贴上纹理。
- 编写代码，使方向键可以移动摄像机的位置和观察角度。

如果成功地完成了这些步骤，得到的程序就有些 FPS 游戏的感觉了，但是还不能使用鼠标移动摄像机的位置，游戏中也没有真正的关卡。进一步开发游戏就需要大量的数学知识了，值得研究的一些主要的公式如下。

需要研究并实现射线与三角形、射线与平面、射线与球和射线与四方形的碰撞检测。每种检测应该返回一个或者多个相交点。射线由一个位置和一个方向组成。使用射线表示激光并检测角色的脚是否触碰了地板是很方便的。

通过射线检测，可以创建线段与三角形、线段与平面、线段与球和线段与四方形的碰

撞检测。线段用于检测玩家或者玩家的子弹是否穿过了墙或敌人。线段通过从玩家在上一帧中的位置到玩家在当前帧中的位置绘制一条线创建。如果线段穿过了墙，则意味着玩家试图穿墙。这是不合理的，所以需要使玩家的位置沿着线段回退，直到玩家不再能穿过墙。

对于摄像机的移动，需要研究极坐标，因为它们经常用于把鼠标的移动转换成游戏世界中摄像机的移动。还需要研究 BSP(Binary Space Partitioning, 二叉空间划分)树，因为它们可以高效地渲染 3D 关卡。研究第一人称射击游戏的最佳方法是单独处理游戏的每个元素，然后创建一个简单的测试程序，以确定代码的工作符合预期。在完成了所有的技术后，可以把所有的元素结合到一起，创建一个更加复杂的游戏。

11.3.4 策略游戏

策略游戏主要分为两种类型：回合制游戏，它们与传统的桌面战棋类游戏有密切的关系；实时游戏，它们与桌面游戏的关系比较疏远。回合制游戏比较容易编程。玩家在每个回合中可以采取规定数量的动作，当不能再执行动作后，回合结束。实时游戏允许玩家快速执行多种动作，但是每种动作都需要一定的时间才能完成。回合制游戏一般使用 2D 图像，而实时游戏一般使用 3D 图像。

策略游戏一般采用自上至下的视图，允许玩家尽可能多地查看游戏世界。如果图像是 2D 的，那么区块和精灵经常用于表示不同类型的陆地、单位和角色。3D 游戏通常使用 3D 高度图。在 Engine 类中，精灵是一个 2D 四方形，假设该四方形被多次分割，使其表面类似于一个网格，如图 11-5 所示。图 11-6 显示了一个示例高度图。每个顶点都可以在 Y 轴上上下下移动，从而形成地形的感觉。高度图通常由灰度图像指定。图像的每个像素对应于网格中的每个像素，高度取决于像素的强度。位于黑色和白色正中间的灰度像素值可能表示顶点没有移动。黑色像素将顶点向负向移动一个单位，白色像素值将顶点向正向移动一个单位。

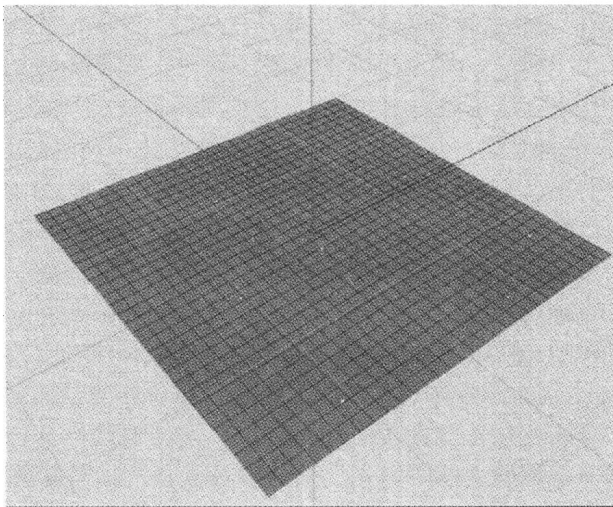


图 11-5 顶点网格

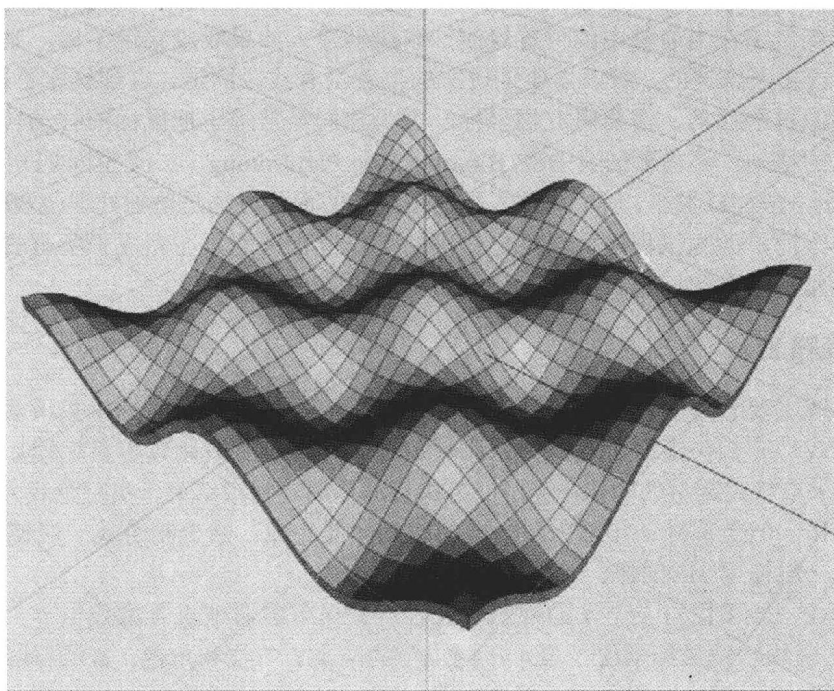


图 11-6 顶点位置移动后的网格形成了一个高度图

使用位图表示高度的优点是可以使用一个绘图包绘制一个高度图。高度图在计算机游戏中有大量应用，读完这里的介绍后，读者应该很容易识别出它们。实时策略游戏经常使用高度图表示地形，使用 3D 模型表示游戏中的每个单位。

策略游戏(特别是基于战斗的策略游戏)的一个困难之处在于对人工智能(Artificial Intelligence, AI)的设计。AI 系统分为两种。第一种是自己的单位的 AI。如果命令一辆坦克从一个位置前往另外一个位置，坦克必须采取最短路径，并避免碾压平民。这种问题一般叫做寻路，最简单的解决方法是研究 A*算法，并实现该算法的一种合适的变体。并不是所有的游戏都使用 A*算法来寻路，但这是一种很流行的算法，而且理解起来也很简单。

实时策略游戏中的第二类 AI 是控制计算机对手的 AI。这种 AI 必须对玩家提出挑战，但是不能看上去是在作弊，而且应该是可以被击败的。这个问题有很多种新奇的解决方案，有的听起来甚至让人感到很神秘，例如神经网络，但是最常见的解决方案是有点普通的状态机和一些规则。策略游戏的 AI 必须有许多状态，例如 **build-up-units**、**attack** 和 **defend**。每种状态都有关联的规则，例如，**build-up-units** 状态的规则可能是：如果单位数量大于 10，进入 **attack** 状态。**attack** 状态的规则可能是：如果攻击单位距离敌人基地较远，则朝向敌人基地前进。这个状态可能还包含首先攻击什么目标以及特定目标具有哪种优先级等规则。这些规则通常可以在一个数据文件中定义，然后通过大量的调整和测试进行细化。

11.3.5 角色扮演游戏

在游戏开发人员中，角色扮演游戏很受欢迎，但是它们也颇具挑战性。角色扮演游戏

包含很多物品、位置、敌人和角色，它们都需要文本描述和艺术表示。另外还需要开发一个系统来创建一个完整的游戏：对话系统、世界探索系统、物品管理系统、升级系统和战斗系统。

1. 类 Rogue 游戏

之所以得名类 Rogue 游戏，原因在于它们与早期的地下城游戏 *Rogue* 十分相似。类 Rogue 游戏几乎总是使用 ASCII 字符，而不是图像。与文字类游戏类似，这意味着程序员可以把主要时间集中在游戏而不是开发图形上。一些比较受欢迎的类 Rogue 游戏包括 *ADoM*(<http://www.adom.de/>)、*Nethack*(<http://www.nethack.org/>)和 *Crawl*(<http://crawl.develz.org/wordpress/>)。 *Dwarf Fortress*(<http://www.bay12games.com/dwarves/>)中也使用了 ASCII，这是一个大型的、通用的奇幻世界创建器和模拟器，允许用户控制多个矮人来建筑地下栖息地。ASCII 游戏不一定很简单，*Dwarf Fortress* 就包含一个复杂的天气模拟器、流体物理、心理学模型等。

使用 C# 创建 ASCII 游戏类似于创建一个文字类游戏，需要创建一个控制台应用程序。但是与文字类游戏不同，ASCII 游戏需要一个游戏循环。每次更新后，使用文本把世界输出到控制台上。输出世界后，控制台光标需要移回控制台窗口的开始位置。控制台光标是一个位置，规定程序写控制台时把文本输出到什么地方。下一次世界更新将在前一次更新的文本上输出，从而起到更新的作用。下面的代码使用 @ 符号代表玩家，渲染了一个小型 ASCII 地图。

```
static void Main(string[] args)
{
    int _mapWidth = 10;
    int _mapHeight = 10;
    int _playerX = 0;
    int _playerY = 0;
    bool _playerIsAlive = true;
    while (_playerIsAlive)
    {
        for (int i = 0; i < _mapHeight; i++)
        {
            for (int j = 0; j < _mapWidth; j++)
            {
                if (j == _playerX && i == _playerY)
                {
                    Console.Write('@');
                }
                else
                {
                    Console.Write('.');
                }
            }
        }
    }
}
```

```
        Console.WriteLine();  
    }  
    Console.WriteLine();  
}  
Console.SetCursorPosition(0, 0);  
}  
}
```

虽然这段代码很短，但是却演示了类 **Rogue** 游戏的基础。下一步是读取方向键输入，并在这个小型世界中移动角色。

2. 基于区块的角色扮演游戏

基于区块的游戏使用叫做区块(tile)的小精灵构建 2D 世界。不只是角色扮演游戏，其他各种游戏中都可能使用区块，但是在早期的日式 RPG 中这种技术非常流行，例如第 7 代之前的《*Final Fantasy*(最终幻想)》游戏、*Chrono Trigger*、早期的《*Zelda*(塞尔达传说)》游戏等。区块图像在手持设备(如 DS 和 PSP)中仍然十分常见。图 11-7 所示是基于区块的游戏的一个示例。描述世界的所有区块被打包在一个或多个叫做地图块(tilemap)的纹理中。图 11-8 显示了用于构建图 11-7 的地图块。

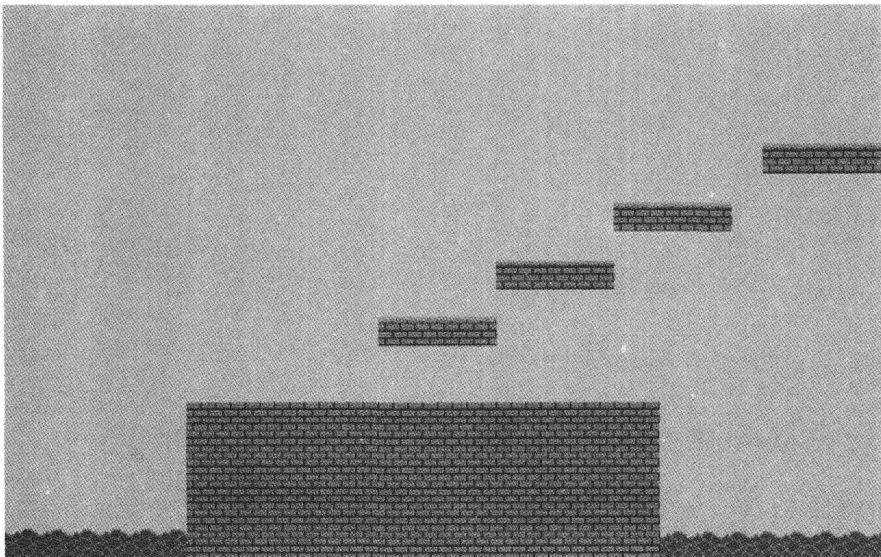


图 11-7 基于区块的游戏

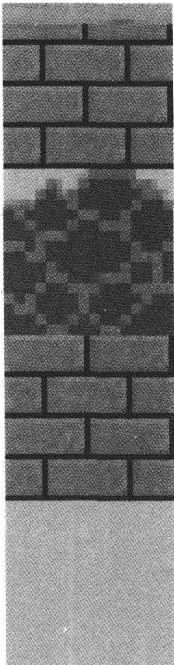


图 11-8 地图块

基于区块的游戏的一个不错的起点是创建一个代表世界的文本文件。例如：

```
#####  
#           #
```

```
#           #
#S           E#
#####
```

这里的#代表阻止角色移动的墙。S 代表关卡开始，E 代表结束。下面这些粗略的代码演示了如何构建一个关卡。

```
double startX = 0;
double startY = 0;
double tileWidthHeight = 32;

Dictionary<char, TileData> tileLookUp = LoadTileDefinitions();
string levelDef =
    "#####\n" +
    "#           #\n" +
    "#           #\n" +
    "#S           E#\n" +
    "#####\n";
TileMap tileMap = new TileMap();
double currentX = startX;
double currentY = startY;
int xPosition = 0;
int yPosition = 0;
foreach (char c in levelDef)
{
    if (c == '\n')
    {
        xPosition = 0;
        yPosition = yPosition + 1;
        currentX = startX;
        currentY -= tileWidthHeight;
        continue;
    }
    Tile t = tileLookUp[c].CreateTile();
    t.SetPosition(currentX, currentY);
    tileMap.AddTile(xPosition, yPosition, t);
    xPosition++;
    currentX += tileWidthHeight;
}
```

代码迭代了基于文本的区块定义，并把每个角色转换成了一个区块。每个区块都是一种精灵，其位置是精心设计的，与邻近的区块紧密相连。ASCII 字符用于索引一个由一组区块定义组成的字典，区块的定义中包含区块数据。区块数据中可能包含一个纹理，还会包含一些决定玩家是否可以在该区块上行走的标志，还可能包含一些特殊的属性，例如当玩

家踏到该区块上时游戏结束。区块定义用于创建区块，区块使用精灵绘制自身。创建了区块后，它将被添加到地图块上。在游戏循环中，地图块用于渲染所有的区块。玩家在区块之后被渲染，可以自由地四处走动。

基于区块的游戏的一个常见特征是区块分层。创建几个地图块之后把它们叠起来，这可以创建出视差效果，使得背景的移动速度比玩家的移动速度慢。另外，这种技术还允许玩家出现在地图上特定元素的后面。

3. 3D 角色扮演游戏

创建 3D 角色扮演游戏的方法有很多种。《辐射 3》、《Oblivion(湮没)》和《生化奇兵》使用了 FPS 的方法。其他 3D 游戏将摄像机放到角色的上方，使其向下观察玩家角色和游戏世界，以此来模拟传统的基于区块的方法。《暗黑破坏神 3》和《Twilight(暮光之城)》是这种风格的两个典型的例子。第三种方法是使用第三人称摄像机，如 N64 以后的《塞尔达传说》或者《Mass Effect(质量效应)》。

基于 FPS 的方法使玩家以角色的视角观察游戏世界，从而产生更强的现场感。为了创建这类游戏，需要首先能够开发一个简单的 FPS 游戏。FPS 游戏一般发生在紧闭的建筑中，分为几个任务或者关卡，而 RPG 一般有一块广阔的开放区域和一个巨大的游戏世界。允许角色从一个城市无缝地走到另一个城市的开放世界的代码很难编写，它要求在角色四处移动时动态地加载和卸载数据。在 FPS 游戏中实现这一点很难，因为如果爬到一个很高的位置，那么他应该可以鸟瞰世界，看到他出发的地方。这意味着世界的低细节版本需要在大多数时候都保存在内存中，而当玩家接近某个区域时，需要显示出更多的细节。如果对这种算法感兴趣，那么建议搜索 ROAM(Real-time Optimally Adapting Meshes, 实时优化自适应网格)。和其他游戏一样，从简单的小型项目入手，逐渐开发出更复杂的项目是一个好主意。

这种开放世界问题的一种更简单的方法是把玩家限制在一个可以一次加载到内存中的小区域内。《System Shock(网络奇兵)》和《生化奇兵》使用这种限制来推动情节。在《网络奇兵》中，玩家被限制在空间站中，只能通过中央电梯进入不同的关卡。在电梯移动的过程中，旧关卡被卸载，新关卡被加载。《生化奇兵》采用的方式类似，只不过是把玩家限制在了一个水下基地中。

第三人称游戏与 FPS 游戏很类似，但是摄像机被往回拉一些，允许任何时候都可以在屏幕上看到玩家角色。使用有限的内存和计算机资源表示巨大的动态世界的问题仍然存在。

《质量效应》通过把关卡分散到不同的星球上，实现对游戏世界的分割。玩家从飞船降落到星球表面时，关卡将被加载。《塞尔达传说》有时候在加载一个区域和卸载另一个区域时显示一个加载界面，有时候会使用 dog leg。dog leg 是关卡的延伸，在这里除了玩家通过的狭窄走廊外，什么都看不到。走廊的形状看上去就像是狗的一条腿，或者一个十分不严格的 L 形。走廊很长，所以在玩家从走廊中走出来之前，有足够的时间加载新关卡。dog leg 不一定必须是走廊，还可以是山路，或者高大建筑物之间的狭窄通道。

从上方观察下方的摄像机不像 FPS 或者第三人称摄像机视角那样受巨大世界问题的严重影响，因为摄像机只能看到世界的一小部分。使用 FPS 摄像机的玩家可以向地平线看去，看到远远延伸出去的陆地，但是从上方观察下方的摄像机却并非如此。玩家附近总是有一

个很小的固定区域，这使得加载和卸载关卡的某个部分变得更加容易。从上至下的视图在基于区块的方法中也可以很好地应用，但是每个区块不是精灵，而是一个 3D 网格，与周围的 3D 网格可以匹配起来。这样可以更快地开发关卡：建模几个部分后，可以把它们连接起来，从而相当快速地形成一个关卡。

11.3.6 平台游戏

平台游戏在业余和独立的游戏开发人员那里很受欢迎。创建这种游戏需要一点数学，但是这些数学一般很简单，而且在 2D 平台游戏的领域，有大量不同的可玩性想法可供探索。大多数平台游戏都使用基于区块的方法，这与第 11.3.5 节介绍的方法很类似。

平台游戏一般需要一点物理建模，重力几乎总是需要进行建模的。物理部分可以使用速度和加速度的基本公式编写，也可以使用第三方物理库，例如 Box2d(存在一个 C#端口 Box2dx)。如果决定手动编写所有系统，那么可以使用 **Sprite** 类和简单的矩形与矩形之间的碰撞检测快速得到一个可以运行的小游戏。

在下面这个非常简单的游戏状态中，使用左右方向键移动红色方块，使用上方向键可以进行跳跃。

```
class PlatfomerTestState : IGameObject
{
    class PlatformEntity
    {
        const float _width = 16;
        const float _height = 16;
        RectangleF bounds = new RectangleF(-_width, -_height, _width,
            _height);
        public void Render()
        {
            Gl.glBegin(Gl.GL_LINE_LOOP);
            {
                Gl.glColor3f(1, 0, 0);
                Gl.glVertex2f(bounds.Left, bounds.Top);
                Gl.glVertex2f(bounds.Right, bounds.Top);
                Gl.glVertex2f(bounds.Right, bounds.Bottom);
                Gl.glVertex2f(bounds.Left, bounds.Bottom);
            }
            Gl.glEnd();
            Gl.glEnable(Gl.GL_TEXTURE_2D);
        }

        public Vector GetPosition()
        {
            return new Vector(bounds.Location.X + _width, bounds.Location.
```

```
        Y + 16, 0);
    }

    public void SetPosition(Vector value)
    {
        bounds = new RectangleF((float)value.X - _width, (float)value.
            Y - _height, _width, _height);
    }
}

PlatformEntity _pc = new PlatformEntity();
Input _input;
double _speed = 1600;
Vector _velocity = new Vector(0, 0, 0);
bool _jumping = false;
double _gravity = 0.75;
double _friction = 0.1;

public PlatformerTestState(Input input)
{
    _input = input;
}

#region IGameObject Members
public void Update(double elapsedTime)
{
    if (_input.Keyboard.IsKeyHeld(Keys.Left))
    {
        _velocity.X -= _speed;
    }
    else if (_input.Keyboard.IsKeyHeld(Keys.Right))
    {
        _velocity.X += _speed;
    }

    if (_input.Keyboard.IsKeyPressed(Keys.Up) && !_jumping)
    {
        _velocity.Y += 500;
        _jumping = true;
    }

    _velocity.Y -= _gravity;
    _velocity.X = _velocity.X * _friction;
```



```
Vector newPosition = _pc.GetPosition();
newPosition += _velocity * elapsedTime;

if (newPosition.Y < 0)
{
    newPosition.Y = 0;
    _velocity.Y = 0;
    _jumping = false;
}
_pc.SetPosition(newPosition);
}

public void Render()
{
    Gl.glDisable(Gl.GL_TEXTURE_2D);
    Gl.glClearColor(1, 1, 1, 0);
    Gl.glClear(Gl.GL_COLOR_BUFFER_BIT);

    Gl.glEnable(Gl.GL_LINE_SMOOTH);
    Gl.glLineWidth(2.0f);
    Gl.glPointSize(10.0f);
    Gl.glColor3d(0, 0, 0);
    _pc.Render();
}
#endregion
}
```

这里有一个叫做 **PlatformEntity** 的较小的类，该类绘制一个简单的红色线框，并提供了四处移动该方框的方法。**Process** 循环中有一些非常简单的类似于卡通的物理建模。计算出实体的新位置后，对这个位置进行检测，看实体是否落到了 Y 轴上的零点以下。如果是，则向上推实体。这是尝试不同的控制方法的一个不错的起点。可以添加精灵来使场景中具有更多的角色。另外，还需要额外的碰撞代码来处理自由浮动的平台。

11.4 结束语

现在，你应该有了一些非常好的游戏思路，也知道了如何实现它们。记住要从简单的项目着手，首先尽早地创建一个可以基本工作的版本，然后再进行细化。如果遵循这些步骤，很快就可以创建一个让自己感到骄傲的游戏，你会迫不及待地要把它发布出去。游戏编程是一个很有趣的体验，祝你好运。

附录 A

推荐阅读材料

本附录推荐了一些优秀的书籍，对于创建游戏项目或者提高你的技能和增加你的知识可以提供极大的帮助。本附录只介绍了书籍，但是本书配套光盘中提供了一个 HTML 文件，其中包含了指向有价值的网站和在线论文的超链接。

A.1 编程实践

编程实践贯穿从想法的闪现到项目的完成的整个过程。这方面的书籍为如何完成开发以及如何比较轻松地完成产品程序提供了一些指导。它们并不是专门针对游戏，里面介绍的内容可以应用到所有的软件开发项目，但仍然是值得阅读的。

***The Pragmatic Programmer: From Journeyman to Master*(ISBN 0-201-61622-X)**，作者 Andrew Hunt 和 David Thomas

The Pragmatic Programme: From Journeyman to Master 的篇幅不长，介绍了如何完成软件项目。它并没有专门介绍特定的语言，而是介绍了整个软件开发过程。虽然篇幅短，但是 *The Pragmatic Programme: From Journeyman to Master* 中包含了有价值的信息，如果想要提高自己的技能，建议一定要阅读这本书。

***Code Complete Second Edition: A Practical Handbook of Software Construction* (ISBN-13:978-0735619678)**，作者 Steve McConnell

Code Complete Second Edition: A Practical Handbook of Software Construction 是鼓励实效编程风格的另外一本书，但是与 *Code Complete Second Edition: A Practical Handbook of Software Construction* 不同，这本书对细节的介绍更加深入。*Code Complete* 介绍了如何编

写紧凑、整洁并且易于扩展和调试的代码。它稍微侧重于 C++/C，但是大部分内容仍然适用于 C#或其他任何编程语言。

A.2 C#语言和软件架构

阅读这些书的读者应该熟悉 C#或一门类似的语言。因此，它们的目标是使读者更深入地理解语言。软件架构是一个术语，用来描述计算机程序的设计和结构，包括如何把代码分解成各个部分，以及这些部分如何进行通信。

***CLR via C#, 3rd Edition*(ISBN-13: 978-0735627048), 作者 Jeffrey Richter**

这本书的重点不是 C#，而是作为 C#基础的虚拟机。理解这个虚拟机有助于提高 C#程序的速度和效率。

***Head First Design Patterns*(ISBN-13:978-0596007126), 作者 Eric T Freeman, Elisabeth Robson, Bert Bates 和 Kathy Sierra**

在设计模式方面，一般都会推荐 *Design Patterns:Elements of Reusable Object-Oriented Software*，但是那本书有点枯燥。*Head First Design Patterns* 更容易被理解，也更有趣。设计模式是一些简短的描述，说明了如何编写代码来解决软件开发中的常见挑战。有必要理解这些模式，以了解其他人如何应对设计问题，以及理解当人们批评过度使用单例模式或者建议使用装饰模式时说的是什么意思。这本书使用 Java 来解释模式，但是把示例转换为使用 C#并不困难。

A.3 数学编程和图形编程

如果想要扩展自己对数学编程和图形编程的知识，下面介绍的两本书是一个不错的起点。

***3D Math Primer for Graphics and Game Programming*(ISBN-13: 978-1556229114), 作者 Fletcher Dunn 和 Ian Parberry**

我在几家游戏开发工作室中工作过，也参观过不少游戏开发工作室，经常可以看到这本书摆在某个人的桌上。这本书介绍了理解 3D 游戏的基本工作原理所需的全部数学。它比介绍相同内容的几乎任何书都要容易理解，但仍然是一本介绍高等数学的书，读者需要付出不少努力才能理解其中介绍的内容。它也是一本优秀的参考书，里面提供了大量 C++代码，把它们转换成 C#代码并不困难。书中还提供了大量习题。

***Computer Graphics:Principles and Practice in C(2nd Edition)*(ISBN-13:978-0201848403), 作者 James D. Foley, Andries van Dam, Steven K. Feiner 和 John F. Hughes**

如果想要自己编写与 OpenGL 类似的库，可以阅读这本书。这是一本优秀的参考书，涉及的领域很广，并且提供了使用 C 语言编写的代码示例。书中介绍了原理性的知识，例如表示颜色的不同方式、监视器的工作原理、绘线算法、光栅器的编写方法等。计算机图形学的原理没有改变，虽然如此，这本书已经开始呈现老态了。这本书没有介绍着色器和现代图形硬件，而是介绍了越来越少使用的系统，例如 PHIGS(被 OpenGL 取代的一种 API)。

A.4 OpenGL

关于 OpenGL，有两本很优秀的书籍，就是通常所称的红宝书(Red Book)和橙宝书(Orange Book)。红宝书介绍标准的 OpenGL 库，橙宝书介绍使用 OpenGL 着色语言 GLSL 的着色器。

OpenGL Programming Guide: The Official Guide to Learning OpenGL(ISBN-13: 978-0321552624)，作者 Dave Shreiner

这是红宝书，介绍了 OpenGL 的全部基础知识，注意在最新版本中 OpenGL 的哪些部分发生了变化。书中的示例使用 C++ 编写，但是在 C# 中几乎所有的 OpenGL 调用都是相同的，因此可以使用书中的代码。

OpenGL Shading Language 3rd Edition，作者 Randi J. Rost, Bill Licea-Kane, Dan Ginsburg, John M. Kessenich, Barthold Lichtenbelt, Hugh Malan 和 Mike Weiblen

这是橙宝书，介绍了使用 OpenGL 时更加现代的着色器驱动的方法。这本书使用 GLSL，但是在理解了一种着色语言后，很容易转向使用另外一种，因为它们都十分类似。