数字媒体艺术与技术系列丛书
Digital Media Technology and Interactive Arts

# 三维游戏引擎设计与实现

# Implementation of 3D game engine

耿卫东 陈凯 李鑫 徐明亮 著

ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

数字媒体艺术与技术系列丛书

# 三维游戏引擎设计与实现

耿卫东 陈 凯 李 鑫 徐明亮 著

ZHEJIANG UNIVERSITY PRESS
浙江大学出版社

目　录

序
前言
第1章　游戏引擎发展概况
1.1 何为游戏引擎
1.2 世界游戏引擎发展概况
1.2.1 引擎的诞生（1992-1993年）
1.2.2 引擎的转变（1994-1997年）
1.2.3 引擎的革命（1998-2000年）
1.2.4 跨世纪的引擎（2001- ）
1.3 国内游戏引擎发展概况
1.4 游戏引擎的发展趋势
第2章　游戏引擎的总体架构设计
2.1 游戏引擎的构架
2.2 客户端体系结构
2.2.1 登录服务器
2.2.2 大厅服务器
2.2.3 中心服务器
2.2.4 数据服务器
2.2.5 游戏服务器
第3章　三维场景管理模块的设计
3.1 场景图
3.2 有向包围盒
3.3 节点包围球
3.4 场景图的渲染

第4章 三维渲染管道的设计
4.1 渲染器
4.2 材质管理
4.3 顶点缓冲区和索引缓冲区
4.3.1 顶点缓冲区
4.3.2 索引缓冲区
4.4 静态模型
第5章 骨骼动画技术的实现
5.1 动作数据格式解析
5.1.1 骨架
5.1.2 蒙皮骨骼
5.1.3 关键帧骨骼动画
5.2 骨骼动画的更新
5.3 进阶骨骼动画
5.4 动画浏览器
第6章 粒子特效
6.1 粒子系统的设计与实现
6.2 关键帧技术及自定义发射器示例
6.3 粒子系统编辑器
第7章 图形用户界面模块
7.1 GUI模块构架
7.2 GUI控件
7.2.1 文字标签
7.2.2 图片框
7.2.3 按钮、选择框、组选框
7.2.4 滚动条
7.2.5 文本编辑框

7.2.6 列表框
7.2.7 下拉列表框
7.3 GUI编辑器
第8章 输入模块
8.1 关于DirectInput
8.2 输入模块的基本框架
8.3 鼠 标输入
8.4 键盘输入
8.5 游戏杆输入
8.6 输入模块的两个例子
第9章 网络模块的设计与实现
9.1 Winsock套接字概述
9.2 完成端口模型
9.3 I/O模型的选择与比较
9.4 游戏服务器概述
9.5 游戏平台架构分析
9.6 服务器模块剖析
9.6.1 登录服务模块
9.6.2 大厅服务模块
9.6.3 中心服务模块
9.6.4 数据服务模块
9.7 服务器端基础类库分析
9.7.1 服务器底层类库需求
9.7.2 游戏服务器功能描述
9.7.3 后台管理模块
9.8 预测与同步技术
9.9 Dead Reckoning（DR）算法介绍与应用

第10章 音效模块的设计与实现
10.1 DirectX对音频的支持
10.2 建立DirectSound
10.2.1 创建DirectSound对象
10.2.2 设置协作级别（Cooperative Level）
10.2.3 检索硬件信息
10.2.4 扬声器的设置
10.2.5 压缩
10.2.6 建立缓冲区
10.2.7 音乐文件加载
10.3 声音的播放与控制
10.4 3D音效
10.4.1 3D空间、声源、听者
10.4.2 最大最小距离
10.4.3 处理模式
10.4.4 声音的锥效应
10.4.5 声源的创建
10.4.6 听者的创建
10.5 音效模块封装
第11章 游戏中的人工智能技术
11.1 Game AI的特点
11.2 常用Game AI技术
11.3 路径规划与移动技术
11.4 有限状态机
11.5 脚本技术
11.6 群聚技术
11.7 遗传算法

11.8 神经网络
附录
Ⅰ. 数学库
Ⅰ.Ⅰ 常用计算函数
Ⅰ.Ⅱ 向量—Vector2
Ⅰ.Ⅲ 向量—Vector3
Ⅰ.Ⅳ 向量—Vector4
Ⅰ.Ⅴ 矩阵
Ⅰ.Ⅵ 四元数
Ⅰ.Ⅶ 射线
Ⅰ.Ⅷ 平面
Ⅰ.Ⅸ 球
Ⅰ.Ⅹ 平截锥体
Ⅰ.Ⅺ 相交测试
Ⅱ. 程序附录
参考文献
附录1

**图书在版编目（CIP）数据**

三维游戏引擎设计与实现 / 耿卫东著. —杭州：浙江大学出版社，2008.10
（数字媒体艺术与技术系列丛书）
ISBN 978-7-308-05835-3

Ⅰ. 三… Ⅱ. 耿… Ⅲ. 三维－动画－游戏－软件开发－高等学校－教材 Ⅳ. TP311.5

中国版本图书馆CIP数据核字（2008）第034921号

**三维游戏引擎设计与实现**

耿卫东 陈 凯 李鑫 徐明亮 著

**策 划** 希言 许佳颖

**责任编辑** 陈晓嘉

**文字编辑** 冯骏

**封面设计** 彭韧等

**出版发行** 浙江大学出版社
（杭州天目山路148号 邮政编码310028）
（E-mail: zupress@mail.hz.zj.cn）
（网址：http://www.zjupress.com
http://www.press.zju.edu.cn）
电话：0571-88925592 88273066（传真）

排　　版　杭州中大图文设计有限公司

印　　刷　杭州杭新印务有限公司

开　　本　787mm × 1092mm　1/16

印　　张　13.25

字　　数　287千

版印次　2008年10月第1版　2008年10月第1次印刷

书　　号　ISBN 978-7-308-05835-3

版权所有　翻印必究

浙江大学出版社发行部邮购电话　（0571）88925591

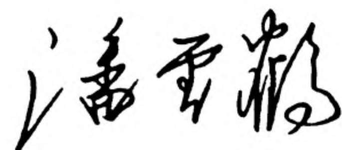
# 序

随着全球经济的高速发展，IT技术、传媒和通信产业的交融使数字媒体这一信息技术与媒体艺术的交叉领域，得到了前所未有的发展，日益成为现代服务业的重要支撑和优先发展方向，成为我国未来新产业的重要增长点。国家极其重视数字媒体产业的发展和交叉性人才的培养，自2005年以来，陆续成立了国家数字媒体产业化基地、国家动画产业基地和国家动画教学研究基地，这大大推进了我国数字媒体学科的建设和数字媒体产学研的结合，进而推动了数字经济的健康发展。

目前，数字媒体教育在国内高等教育体系中还处于新兴阶段，人才缺口大，缺少系列化的优秀教材。很多高校和研究机构对此进行了有益的探索，形成了一定的成果。浙江大学出版社推出的这套《数字媒体艺术与技术系列丛书》是一种很好的尝试。该丛书结合教育部数字媒体新专业的建设要求，从数字媒体产业人才的实际需求出发，整合了各类高校优质教学资源，注重技术与艺术的有机结合，既包括数字媒体的基础内容，又包括数字媒体艺术、技术的专业技能，体现了数字媒体与具体行业领域的结合以及国内外数字媒体产业发展方向的研究成果和动态。

我很高兴地看到，《数字媒体艺术与技术系列丛书》不仅得到了教育部计算机教学指导委员会、CAD&CG国家重点实验室、中国图形图像学会等单位的大力支持，而且得到了影视、动画、游戏等各类媒体行业和IT产业领域的知名专家和学者的指导。这种专业教育与产业的融合、技术与艺术结合的探索，必将对培养适合产业发展需求的高素

质人才、对我国高校数字媒体教育的发展和学科建设起到积极的推动作用！



2008.7于北京

---

＊ 本文作者为中国工程院常务副院长，院士

# 前言

游戏引擎是当今游戏产业发展的核心技术驱动力。一款优秀的游戏引擎不仅能够极大地提高游戏开发效率，而且能给开发者带来丰厚的版权收益，这使得国内外的游戏公司纷纷投入到游戏引擎的研究开发中。经过近几年的快速发展，游戏引擎已经步入一个百家争鸣的局面，诞生了很多高水准的商业化游戏引擎，其中包括Unreal、CryEngine等。但遗憾的是，国内的游戏引擎发展尚处于初级阶段，好的游戏引擎比较少，相关的学习资料也非常缺乏，这使得我国游戏产业的持续发展面临严峻的技术挑战。

回顾作者在三维游戏引擎方面的研究与教学实践，一方面，作者在三年前即开始了三维游戏引擎方面的研究开发：首先是获得杭州市产学研科技创新项目“益智类小型网络游戏”的资助，开发了三维游戏引擎的一个初步框架平台；后来，该三维游戏引擎框架平台在图形建模、绘制和动画等方面的部分关键算法和内部核心技术的研究开发陆续得到了国家自然科学基金项目（60373032）、国家自然科学基金重点项目（60633070）、国家“863”高科技计划项目（2006AA01Z313和2006AA01Z335）、长江学者和创新团队发展计划（IRT0652）、浙江省科技厅重点项目（2006C13096和2006C23055）以及宁波市科技局的市校合作项目的资助，逐步形成了一个相对完整的、具有自主知识产权的小型三维游戏引擎——CAP Engine，并在一款交互仿真学习游戏的实际开发中得到了成功应用。另一方面，作者一直在浙江大学计算机学院数字媒体技术专业从事游戏程序设计的教学工作，因此萌生了撰写三维游戏引擎开发技术相关书籍的想法，并得到了浙江大学出版社的大力支持。

本书的撰写结合了CAP Engine的具体设计与实现工作。作者希望和读者一起了解和分享三维网络游戏引擎的开发过程以及其中可能会碰到的困难点；更希望通过这本书，吸引更多的优秀人才参与到游戏引擎的探索中来，为我国的游戏产业向更高层次发展形成强劲的后续动力。全书共分11章，首先简述了国内外游戏引擎的发展现状，接着重点分析了如何完成游戏引擎中主要模块的设计与实现，主要技术内容包括游戏引擎的构架、三维场景管理模块的设计、渲染器的设计、骨骼动画的设计、粒子特效及编辑器的设计、GUI及其编辑器的设计、输入输出模块的设计、网络模块的设计、音效模块的设计、人工智能模块的设计以及引擎总体架构。每个模块的叙述都包括其原理、设计及实现。为了使读者对三维游戏引擎的设计与实现有更切身的体会，本书紧密结合实例，剖析了自主开发的游戏引擎——CAP Engine。在随书光盘中，读者可以找到CAP引擎的所有源代码和若干DEMO，以及引擎的参考文档。相信读者能够通过CAP引擎更透彻地理解游戏引擎中的技术，也可以在该引擎的基础上开发自己的游戏。

在读者对象上，本书主要针对有一定游戏开发经验的读者，或是有相关领域经验的读者，如计算机应用专业和数字媒体专业的高年级本科生或研究生。对于那些对游戏引擎感兴趣但又没有相关开发经验的读者，可以尝试通过参考随书代码，完成自己的引擎或游戏设计。

实际上，游戏引擎的设计和实现并没有想像中的那么难，尤其是入门级引擎的开发对于有游戏开发经验的人来说是很容易跨出的一步。当然，游戏引擎技术目前也在飞速发展之中，作者最大的愿望之一是本书所介绍的内容能为读者在掌握三维游戏引擎技术方面起到抛砖引玉的作用。

当然，游戏引擎的开发是一个复杂的系统工程，不仅需要深邃的内核技术，也需要遵循软件工程的规范，更需要融合不同类型游戏的创作风格、惯例和流程。由于作者的知识面有限，书中肯定存在很多片面和错误之处，恳请读者批评指正。

耿卫东　陈凯　李鑫　徐明亮

2008年6月

# 第1章　游戏引擎发展概况

本章作为全书的绪论，主要介绍何为游戏引擎、引擎在游戏中的作用以及它的进化对于游戏的发展产生了哪些影响、世界游戏引擎发展概况、国内游戏引擎发展概况以及游戏引擎技术的发展趋势等。

## 1.1　何为游戏引擎

简单地说，游戏引擎就是“用于控制所有游戏功能的主程序”，从计算碰撞、物理系统和物体的相对位置到接受玩家的输入，以及声音的输出等等功能，都是游戏引擎需要负责的事情。它扮演着中场发动机的角色，把游戏中的所有元素捆绑在一起，在后台指挥它们有序地工作。因此，无论是2D游戏还是3D游戏，无论是角色扮演游戏、即时策略游戏、冒险解谜游戏还是动作射击游戏，哪怕是一个小游戏，都有一段起控制作用的代码。经过不断地进化，如今的游戏引擎已经发展为由多个子系统共同构成的复杂系统，从建模、动画到光影、粒子特效，从物理系统、碰撞检测到文件管理、网络特性，还有专业的编辑工具和插件，几乎涵盖了开发过程中的所有重要环节。一个典型的游戏引擎一般包含以下几个组件。

- 光影计算。它决定了场景中的光源对处于其中的人和物的影响方式。游戏的光影效果完全是由引擎控制的，折射、反射等基本的光学原理以及动态光源、彩色光源等高级效果都是通过引擎技术实现的。
- 动画技术。目前游戏所采用的动画系统可以分为两种：一是骨骼动画系统，一是模型动画系统。前者用内置的骨骼带动物体

产生运动，比较常见；后者则是在模型的基础上直接进行变形。

- 物理系统。它可以使游戏世界中的物体运动遵循客观世界规律。例如，当游戏人物跳起的时候，系统内定的重力值将决定他能跳多高，以及他下落的速度有多快，子弹的飞行轨迹、车辆的颠簸方式等也都是由物理系统决定的。另外，碰撞检测也是物理系统的另一个核心部分，它可以检测游戏中各物体的动态交互情况。
- 实时渲染。它是引擎最重要的功能之一，也是引擎所有部件当中最复杂的，它的强大与否直接决定着最终的游戏画面质量。
- 人机交互。它负责玩家与电脑之间的沟通，处理来自键盘、鼠标、摇杆和其他外设的信号。
- 网络接口。如果是网络游戏引擎，则网络代码也会被集成在引擎中，用于管理客户端与服务器之间的通信。

通过上面这些介绍可以了解到：游戏引擎相当于游戏的框架。框架搭好后，关卡设计师、建模师、动画师只要往里填充内容就可以了。

## 1.2　世界游戏引擎发展概况

曾经有一段时期，游戏开发者关心的只是如何尽量多地开发出新的游戏并把它们推销给玩家。尽管那时的游戏大多简单粗糙，但每款游戏的平均开发周期也要8~10个月以上。这一方面是由于技术的原因，另一方面则是因为几乎每款游戏都要从头编写代码，造成了大量的重复劳动。渐渐地，一些有经验的开发者开始借用上一款类似题材

的游戏中的部分代码作为新游戏的基本框架，以节省开发时间和开发费用。

其实，每一款游戏都有自己的引擎，但真正能获得他人认可并成为标准的引擎并不多。综观其十多年的发展历程，我们可以发现引擎最大的驱动力来自于3D游戏，尤其是3D射击游戏。动作射击游戏同3D引擎之间的关系相当于一对孪生兄弟，它们一同诞生、成长，互相为对方提供发展的动力。

## 1.2.1 引擎的诞生（1992-1993年）

1992年，3D Realms公司/Apogee公司发布了《德军司令部》（Wolfen-stein 3D）。这部游戏在整个电脑游戏发展史上占据重要地位，它开创了第一人称射击（FPS, First Person Shoot）游戏的先河。更重要的是，它在X轴和Y轴的基础上增加了一根Z轴，在由宽度和高度构成的平面上增加了一个向前向后的纵深空间。

引擎诞生初期的另一部重要游戏是出自id Software公司的一款非常成功的第一人称射击游戏——《毁灭战士》（Doom）。Doom引擎在技术上大大超越了Wolfenstein 3D引擎，如后者中的所有物体大小都是固定的，所有路径之间的角度都是直角，也就是说玩家只能笔直地前进或后退。这些局限在《毁灭战士》中都被突破。尽管游戏的关卡还是维持在2D平面上进行制作，没有“楼上楼”的概念，但墙壁的厚度可以为任意的，并且路径之间的角度也可以为任意的，这使得楼梯、升降平台、塔楼和户外等各种场景成为可能。

更值得纪念的是，Doom引擎是第一个被用于授权的引擎。1993年底，Raven公司采用改进后的Doom引擎开发了一款名为《投影者》

（ShadowCas ter）的游戏，这是游戏史上第一例成功的嫁接手术。1994年，Raven公司采用Doom引擎开发出《 异教徒》（Heretic），为引擎增加了飞行的特性，成为跳跃动作的前身。1995年，Raven公司采用Doom引擎开发《 毁灭巫师》（Hexen），加入了新的音效技术、脚本技术以及一种类似集线器的关卡设计，使玩家可以在不同关卡之间自由移动。Raven公司与id Software公司之间的一系列合作充分说明了，引擎的授权无论对于使用者还是开发者来说都是大有裨益的，只有把自己的引擎交给更多的人去使用才能使引擎不断地成熟起来。

Doom引擎的授权费为id Software公司带来了一笔可观的收入。在此之前，引擎只是作为一种自产自销的开发工具，从来没有哪家游戏商考虑过依靠引擎赚钱，Doom引擎的成功打开了一片新的引擎授权市场。

## 1.2.2 引擎的转变（1994-1997年）

在引擎的进化过程中，由3D Realms公司在1994年开发的Build引擎是一个重要的里程碑，该引擎的“肉身”就是家喻户晓的《 毁灭公爵》（Duke Nukem 3D）。

《 毁灭公爵》 已经具备了今天第一人称射击游戏的所有标准内容，如跳跃、360度环视以及下蹲和游泳等特性。在Build引擎的基础上，先后诞生过14款游戏，例如《 农夫也疯狂》（Redneck Rampage）、《 阴影武士》（Shadow Warrior）和《 血兆》（Blood）等。虽然Build引擎很成功，不过从总体上看，它并没有为3D引擎的发展带来任何质的变化，突破的任务最终由id Software公司的《 雷神之锤》（Quake）完成了。

Quake引擎是当时第一款完全支持多边形模型、动画和粒子特效的真正意义上的3D引擎，它是网络游戏的始作俑者。《 雷神之锤》把网络游戏带入大众的视野之中，促成了电子竞技产业的一大发展。

之后，id Software公司很快推出《 雷神之锤2》，一举确定了自己在3D引擎市场上的霸主地位。《 雷神之锤2》更充分地利用3D加速和OpenGL技术，在图像和网络方面有了质的飞跃，Raven公司的《 异教徒2》（HereticⅡ ）和《 军事冒险家》（Soldier of Fortune）、Ritual公司的《 原罪》（Sin）、离子风暴工作室发布的《 安纳克朗诺克斯》（Anachronox）等都采用了《 雷神之锤2》引擎。

与此同时，另一个划时代的游戏引擎《 虚幻》（Unreal）在Epic Megagmes公司问世了。它除了精致的建筑物外，游戏中的许多特效即便在今天看来依然很出色，比如荡漾的水波，美丽的天空，庞大的关卡，逼真的火焰、烟雾和力场等效果。从单纯的画面效果来看，《 虚幻》是当之无愧的佼佼者，其震撼力完全可以与人们第一次见到《 德军司令部》时的感受媲美。

Unreal引擎可能是使用最广的一款引擎，在推出后的两年之内就有18款游戏与Epic公司签订了许可协议。同时，Unreal引擎的应用范围不限于游戏制作，还涵盖了教育、建筑等其他领域。

### 1.2.3　引擎的革命（1998-2000年）

两款划时代的作品同时出现在1998年——Valve公司的《 半条命》（Half-Life）和LookingGlass工作室的《 神偷：暗黑计划》（Thief: The Dark Project），它们对后来的作品以及引擎技术的进化具有非常深远的影响。

曾获得无数大奖的《半条命》采用的是Quake和QuakeII引擎的混合体，Valve公司在这两部引擎的基础上加入了两个很重要的特性：一是脚本序列技术，这一技术可以令游戏以合乎情理的节奏通过触动事件的方式让玩家真实地体验到情节的发展；二是对人工智能引擎的改进，敌人的行动与以往相比明显更加狡诈。这两个特点赋予了《半条命》引擎鲜明的个性，在此基础上诞生的《要塞小分队》、《反恐精英》和《毁灭之日》等优秀作品又通过网络代码的加入令《半条命》引擎焕发出了更为夺目的光芒。

《神偷》采用的是Looking Glass工作室自行开发的Dark引擎，Dark引擎在人工智能方面的水准相当高。游戏中的敌人懂得根据声音辨认玩家的方位，能够分辨出不同地面上的脚步声，在不同的光照环境下有不同的目力，发现同伴的尸体后会进入警戒状态，还会针对玩家的行动做出各种合理的反应。如今的绝大部分第一人称射击游戏都或多或少地采用了这种隐秘的风格，包括2002年发布的《荣誉勋章：盟军进攻》（Medal of Honor: Allied Assault）。遗憾的是，由于Looking Glass工作室的过早倒闭，Dark引擎未能发扬光大。

从2000年开始，3D引擎朝着两个不同的方向分化。一是如《半条命》、《神偷》那样通过融入更多的叙事成分和角色扮演成分以及加强游戏的人工智能来提高游戏的可玩性；二是朝着纯粹的网络模式发展，在这一方面，id Software公司再次走到了整个行业的最前沿，他们在QuakeII出色的图像引擎的基础上加入更多的网络成分，破天荒推出了一款完全没有单人过关模式的纯粹的网络游戏——《雷神之锤3竞技场》（QuakeIIIArena），它与Epic公司稍后推出的《虚幻竞技场》（Unreal Tournament）一同成为引擎发展史上的一个转折点。

## 1.2.4 跨世纪的引擎（2001- ）

进入21世纪后，有许多优秀的3D射击游戏陆续发布，其中一部分采用的是QuakeⅢ和Unreal Tournament等第三方引擎，而更多的游戏开发公司采用的则是自己开发的引擎，比较有代表性的包括网络射击游戏《 部落2》（Tribes 2）、第一人称射击游戏《 马科斯• 佩恩》、《 红色派系》（Red Faction）和《 英雄萨姆》 等。

《 部落2》 采用的是V12引擎，这款引擎虽然无法同QuakeⅢ和Unreal Tournament相提并论，但开发者为它制定的许可模式却相当新颖，用户只需花上100美元就可以获得引擎的使用权。虽然随之而来的一系列规定相当苛刻，但是对于那些规模较小的独立开发者来说，这个超低价引擎仍然具有非常大的吸引力。

《 马科斯• 佩恩》 采用的是MAX-FX引擎，这是第一款支持辐射光影渲染技术（Radiosity Lighting）的引擎，它能够结合物体表面的所有光源效果，根据材质的物理属性及其几何特性，准确地计算出每个点的折射率和反射率，让光线以更自然的方式传播过去，为物体营造出十分逼真的光影效果。MAX-FX引擎的另一个特点是“子弹时间”特效（Bullet Time），这是一种《 黑客帝国》 风格的慢动镜头，在这种状态下连子弹的飞行轨迹都可以看得一清二楚。MAX-FX引擎的问世把游戏的视觉效果推向了一个新的高峰。

《 红色派系》 采用的是Geo-Mod引擎，这是第一款可任意改变几何体形状的3D引擎，也就是说，玩家可以使用武器在墙壁、建筑物或任何坚固的物体上炸开一个缺口，穿墙而过，或者在平地上炸出一个弹坑躲进去。此外，Geo-Mod引擎还有高超的人工智能特性。

《英雄萨姆》采用的是Serious引擎，这款引擎最大的特点在于异常强大的渲染能力，面对大批涌来的敌人和一望无际的开阔场景，玩家丝毫不会感觉到画面的停滞，而且游戏的画面效果也相当出色。

可以看出，2001年问世的几部引擎依旧延续了此前两年多来的发展趋势，一方面不断地追求真实的效果，例如MAX-FX引擎追求画面的真实，Geo-Mod引擎追求内容的真实，《军事冒险家》（Soldier of Fortune）的GHOUL引擎追求死亡的真实；另一方面则继续朝着网络的方向探索，如《部落2》、《要塞小分队2》（Team Fortress 2）。

2005年后，id Software公司的《雷神之锤4》和《毁灭战士3》重新建构了一个以单人游戏为主的引擎。与此同时，老对手Epic游戏公司也开发了新一代引擎Unreal 3。从这几款引擎展示的几段采用新引擎实时渲染的动画片段来看，它们的确完全超越了市面上的其他引擎。除了上面的两家传统游戏公司外，Crytek开发的CryEngine 2、甚至免费引擎Nebula 2也都给我们展现了一个出色的效果。游戏引擎产业的蓬勃发展以及下一代引擎概念的提出，预示着一个新的引擎时代的到来。

## 1.3 国内游戏引擎发展概况

国产游戏产品研发一直比较落后。游戏引擎的研发由于前期投入高、风险大，所以相比国外十数年的游戏引擎发展历史，国内的引擎发展可以说尚处于初级阶段。

2000年左右发布的“风魂”引擎算是最早走红的游戏引擎之一了，国内有很多2D游戏使用这个引擎，这里就包括大名鼎鼎的《大话西游》系列及《梦幻西游》。但其实“风魂”也不算是一个游戏引

擎，准确地说只是一个2D图形引擎，它并没有物理、AI属性及方便的编辑工具。

同一时期的引擎还有可乐吧的FancyBox。FancyBox是一个基于浏览器技术的游戏开发平台，它解决了如何在浏览器中无缝运行程序，比如联网、脚本语言、下载的自动管理等技术，以及如何去解释游戏这种复杂应用的思想。第一款用FancyBox开发的游戏是可乐吧的“打雪仗”，获得了初步成功。此后，可乐吧开始开发大型网络游戏“奇域”。于是，FancyBox平台开始基于简化网络游戏的编写代码、维护、运营的全部过程为出发点，并逐步完善；同时对网络游戏中的图形图像技术、网络技术、客户端服务器技术等底层全部封装，形成标准化设计。

但风魂和FancyBox都不能算是3D游戏引擎，真正的国产3D游戏引擎的始祖要推涂鸦软件公司发布的起点引擎（Origin Engine）。“起点引擎”，意如其名，这款引擎希望成为中国本土化引擎的一个起点。“起点引擎”不论是在画面效果还是在渲染性能上，都有出色的表现，已初步具备了同国外游戏引擎相抗衡的实力。在内容创建工具上，起点引擎也提供了强大的集成式编辑工具——起点编辑器，能满足各种类型游戏的内容创建工作。

虽然国内的游戏引擎有了一定的发展，但与国外的差距还是非常大！尤其体现在对先进技术的运用上以及市场推广上。由于中国游戏引擎开发的落后现状，不少计划进入自主游戏研发领域的中国公司纷纷做出了从国外购买游戏引擎的决策，这其中不乏Unreal等价格昂贵的知名大作。然而，购买国外引擎的硬伤同时也令本土厂家痛苦不堪。第一，成本昂贵，价格是大多数中小型游戏开发商无法承受之重；第二，国外厂商对国内市场重视程度不够，很难保证良好的售后

服务；第三，人员的沟通不便，出现技术问题不能得到及时的解决。因此，国内游戏业的持续发展还是需要本土的游戏引擎技术，尽管其目前的发展之路还任重而道远。

## 1.4 游戏引擎的发展趋势

游戏引擎在经历了十余年的发展后正步入一个百家争鸣的格局。游戏引擎有走大而全发展线路的，也有走小而精发展线路的。虽然几个大公司的游戏引擎还是占据着大部分的市场，但不断有好的游戏引擎在市场上冒泡，且大有挑战老大哥的势头。在这个格局中，在中国这个游戏引擎刚刚起步的大市场中，机遇还是很多的。也许现在的技术积累还不够，但要看到我们的后发优势；也许因为人力财力很难走大而全的线路，但可以试试朝小而精的方向发展。

尽管游戏引擎的不断进化使游戏的技术含量越来越高，但最终决定一款游戏是否优秀的因素在于使用技术的人而不是技术本身，“内容为王”是每个游戏引擎研发人员所应该永远铭记心头的金科玉律。正如《无人永生》开发小组所说：（商业化游戏开发的）所有问题最终都会归结为一点——你的游戏是否好玩。

# 第2章　游戏引擎的总体架构设计

游戏引擎是一项巨大的软件工程，通常由许多模块共同组成，比如：渲染器、特效、GUI、动画系统、音效、物理、I/O系统（输入输出）、AI和网络支持。如果希望开发一款有着国外一流引擎品质的引擎，则还需要支持顶点着色器和像素着色器、细节层次模型（LOD）、大规模地形绘制等等特征。

此外，引擎还需要提供一些工具以方便程序员开发游戏。比如场景编辑器、材质编辑器、特效编辑器、模型动画导出插件、动画调试器等。如今，评判一款游戏引擎质量高低的标准中，工具占的比重已经越来越大了。因为引擎的初衷就是方便开发人员，所以如果引擎制作人员希望开发人员在更短的时间内完成品质更好的游戏，就需要花费比引擎内核更多的时间去编写工具。

本章会从客户端和服务器两个角度来分析游戏引擎由哪些部件组成，各个部件又是如何协同工作的。

## 2.1　游戏引擎的构架

从游戏开发人员的角度看，游戏引擎为开发者提供了开发需要的API，提供了一些核心的库以及一些开发辅助工具，它能够显著提高游戏开发效率，降低对开发人员的知识要求。

与别的应用程序一样，游戏引擎也有比较分明的层次关系。因此，在引擎开发初期，需要明确它的主要构成及其层次结构。

图2-1描述了CAP引擎的层次结构图。它一共分为五层。

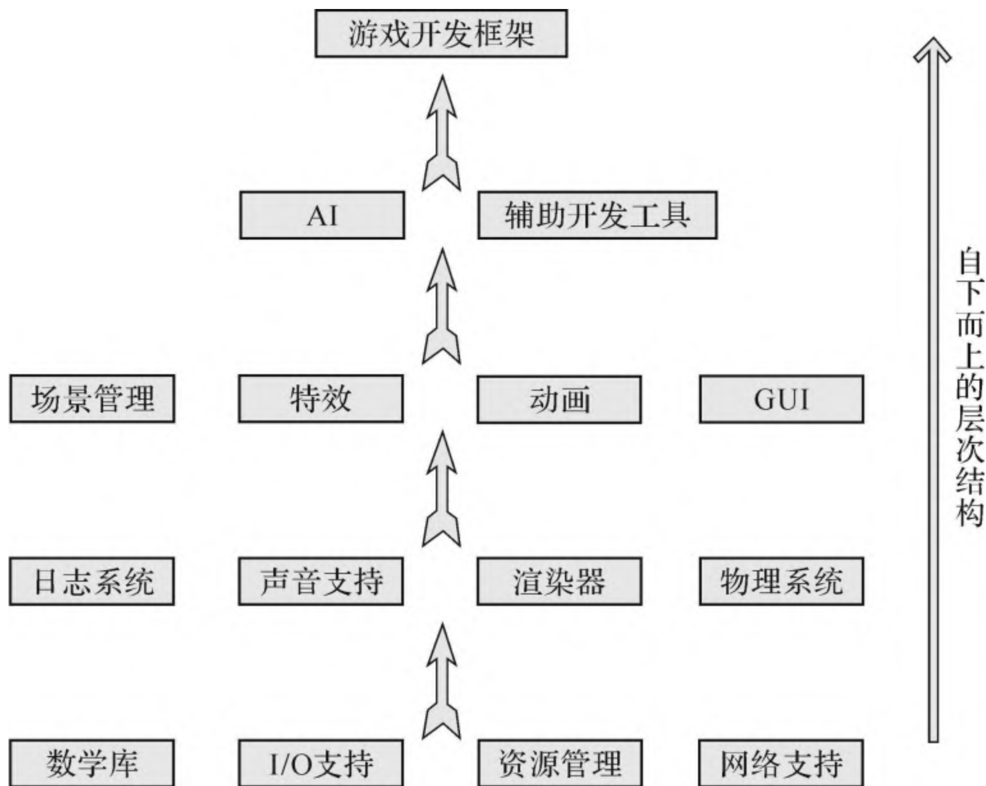



图2-1　CAP引擎自下而上的层次结构

CAP引擎的最底层由数学库、I/O支持、资源管理、网络连接支持四个模块组成。其中，数学库提供了通用数学运算库（比如三角函数、快速近似运算）和三维数学运算库（比如向量、矩阵）；I/O支持提供了鼠标、键盘、手柄等输入设备的支持；资源管理帮助应用程序最合理地使用内存，对系统内存进行调度并对资源实现引用计数管理；网络连接主要提供了SOCKET连接、数据包发送功能。这些底层模块是整个引擎的基石，它们的高效、稳定实现是整个引擎及其应用高效运转的保证。

日志系统、声音支持、渲染器及物理系统也是引擎比较底层的模块。其中，日志系统主要用于信息输出及调试使用；声音支持使游戏更加丰富多彩；渲染器作为引擎最重要的模块之一，提供了二维及三维图形的渲染、光影效果的处理、材质的渲染等等非常重要的功能；物理系统主要负责物体的物理模拟，包括最重要的碰撞检测和动力学模拟。

场景管理、特效、动画和GUI是引擎中较高层的模块。它们也是游戏开发人员接触最多的模块。场景管理模块管理了整个游戏世界，使游戏应用能更高效地处理场景的更新及事件；特效使得游戏更加绚丽，其能力的高低往往决定了游戏的画面是否优美逼真；动画模块主要管理关键帧动画和骨骼动画，它丰富了角色，使之动作更逼真；GUI则负责用户和系统的交互，它往往决定了一个游戏应用的风格。

AI和辅助开发工具是引擎中层次最高的模块。AI处理游戏中的逻辑，负责角色对事件的反应、复杂智能算法的实现等。此外，AI中的有限状态机模块往往成为游戏的运行框架基础。辅助开发工具是一款引擎是否人性化、是否真正提高开发效率的标志，是它们将引擎和开发人员的心联系了起来。辅助开发工具往往提供了所见即所得的开发环境，它是对引擎API的再次封装。一款完整的游戏引擎一般都会包括场景编辑器、材质编辑器、动画编辑器、GUI编辑器、逻辑编辑器等辅助工具。

最后的游戏开发框架严格意义上不属于游戏引擎的一部分。但是，如果光是有引擎的各个模块，没有一个架子将它们整合，开发游戏还是很困难的。于是，一些引擎开发公司往往会配套开发一个游戏应用框架，加快游戏在引擎上开发。游戏应用框架负责整合引擎众模块，使之协调统一，游戏开发人员最终也是在该框架下开发游戏。此

外，也可以针对同一个引擎内核，对应不同的游戏类型，开发不同的游戏框架。

当然，引擎的层次结构并不唯一，开发者可以根据具体应用的需要选择适合自己的引擎体系结构。

## 2.2 客户端体系结构

网络游戏自然要有网络支持，而网络支持中最重要的就是服务器技术。如果希望开发的网络游戏能有更多的同时在线玩家、能让玩家更流畅的进行游戏，就需要好好地设计服务器的架构。

网络游戏的服务器组一般包括登录服务器、大厅服务器、中心服务器、数据服务器和游戏服务器。

### 2.2.1 登录服务器

建立一个独立的登录服务器的目的是减轻其他服务器的负担，该服务器负责处理用户的登录请求及新用户注册的工作，并帮助合法用户建立与大厅服务器的连接。同时，在开发登录服务器时，还需要提供数据加密功能。因为用户需要在网络中传输密码，为了保护密码在传输过程中的安全，需要对其进行加密。此外，独立的登录服务器还可以避免恶性DoS攻击对游戏的影响。

### 2.2.2 大厅服务器

大厅服务器是系统对用户的一个主要接口，它维护着游戏列表和游戏房间列表，负责组织玩家进入不同游戏或是给一个游戏分配不同

房间。同时，大厅还可以成为玩家间交流的平台。由于大厅需要承载所有的玩家，因此开发时有必要进行一些优化，比如本书第9章中介绍的短连接。

### 2.2.3　中心服务器

中心服务器目前主要负责两方面的事务：玩家状态维护和房间映射管理。

虽然有些网络游戏将其独立开来，但也有网络游戏是将中心服务器的功能放到登录服务器上和大厅服务器上。两者均有利有弊，这主要取决于应用负载如何。

### 2.2.4　数据服务器

建立一个专门的数据服务器是非常有必要的，它可以保证数据的安全存储和传输。

### 2.2.5　游戏服务器

游戏服务器是处理具体游戏的逻辑和玩家消息的分发的。从开发的角度上说，游戏服务器和其他服务器的联系相对较少，可以独立开发。但另一方面，游戏逻辑服务器中逻辑处理的好坏和AI算法的效率往往决定了该网络游戏的同时在线人数。所以游戏服务器中的算法研究对于网络游戏开发来说是相当重要的。同时，一款网络游戏可以有多个游戏服务器，多个游戏服务器可以运行不同的游戏，也可以一起

运行一个游戏。所以多个游戏服务器之间的动态交互也是一个研究课题，即游戏世界的无缝连接技术。

在本书的第9章中还详细介绍了网络通信模型和多种服务器的开发，读者可以深入阅读。

# 第3章　三维场景管理模块的设计

三维场景需要有序的组织，这是任何一个3D引擎都必须解决的问题。因此，场景管理是三维游戏引擎的一个核心模块，它的设计好坏直接影响了游戏的运行速度。

在三维游戏中，最常用的场景管理方式是场景图和八叉树（或四叉树）。其中场景图对动态场景的管理非常有效，而八叉树或四叉树则主要管理预定义的静态场景，其中八叉树主要应用在三维室外场景，四叉树则在室内场景中应用较多。通常，游戏中会将场景图和八叉树结合使用来管理场景。图3-1所示为场景编辑器。图3-2为场景的八叉树分割。

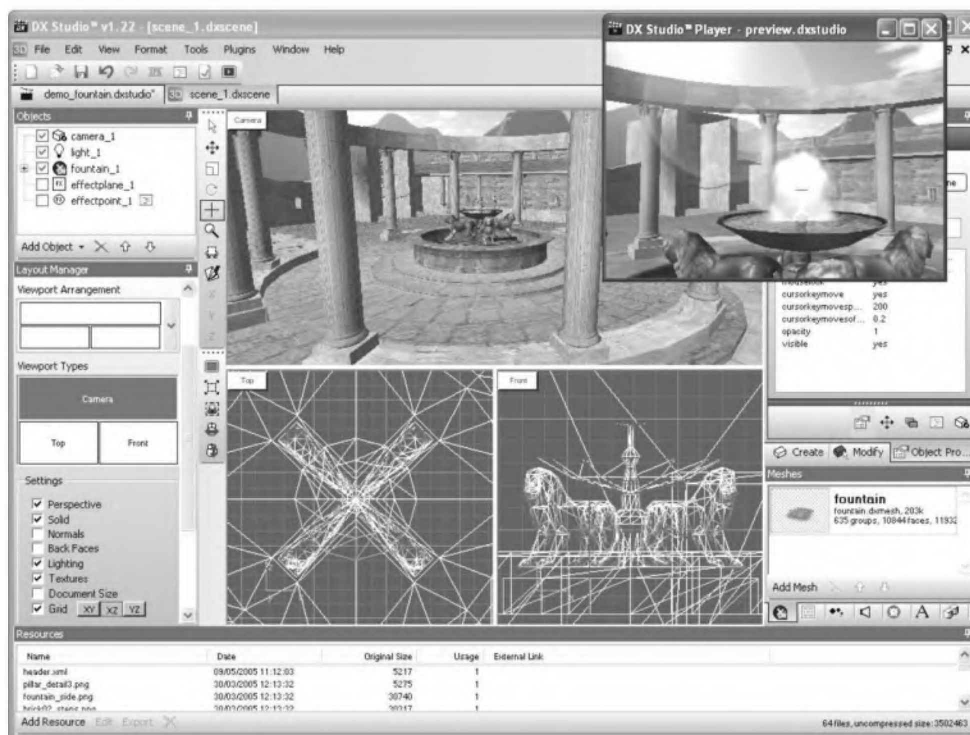


图3-1　场景编辑器

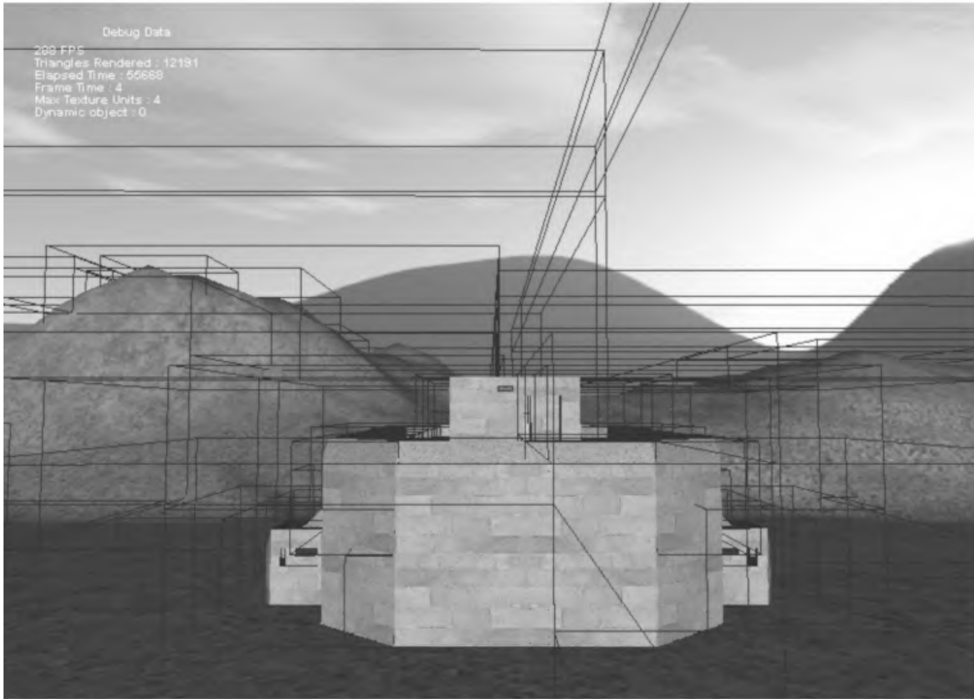



图3-2　场景的八叉树分割

本章将主要介绍如何设计一个场景图，如何集成有向包围盒和节点包围球到场景图中，以及如何合理地对渲染对象排序，减少渲染时间。

# 3.1　场景图

在三维游戏中，场景的组织与管理一般通过场景图（Scene Graph）来完成。场景图是一种将场景中的各种数据以图的形式组织在一起的场景数据管理方式。它一般用K叉树表示，树的每个节点都可以有0~n个子节点。场景图的根节点一般是一个逻辑节点，代表着整个场

景，根节点下的每个节点则存储着场景中物体的数据结构，包括几何体、光源、照相机、声音、物体包围盒、变换、LOD等其他属性。

图3-3中构造了一个简单的场景图。在这个场景中有一辆公交车在路上行驶，车上有司机和乘客1。乘客1戴了一顶帽子还别了一部手机，同时路边有些小车和树。可以发现，公交车、小车和树都是独立的个体，他们之间没有交互，只是他们的相对位置关系会随着运动而改变；而司机、乘客1和公交车的关系有点像“父子”关系，车的运动会带动人一起运动，车旋转也会带动人一起旋转。当然，这里说人的运动是指人的绝对运动，他们相对于车是静止的。同理，乘客1与他的手机、帽子也是“类似”的父子关系。

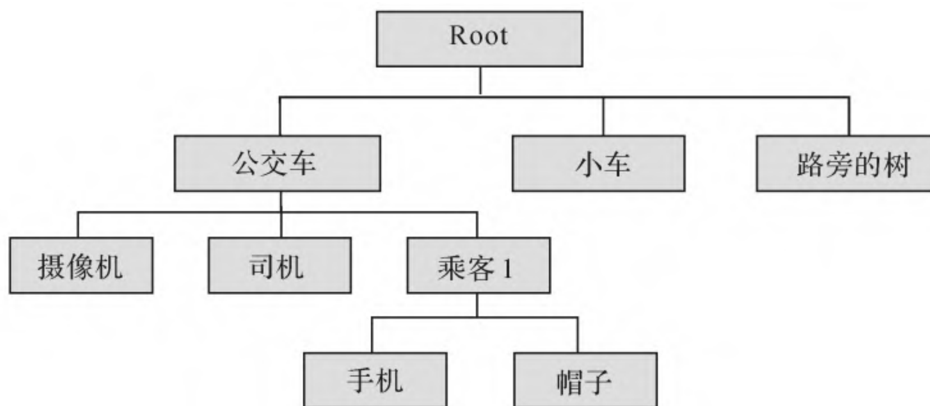



图3-3　场景图示例

分析了物体之间的关系后，可以按照如下步骤构造场景图：

首先，新建一棵K叉树，它有一个空的根节点，位置在世界原点。根节点其实是个逻辑节点，它并没有实际的模型，也不能显示。

其次，在根节点下面挂接三个子节点，分别表示公交车、小车和树。设定子节点与父节点，即世界原点的相对平移和相对旋转。接着，将司机、乘客挂接在公交车这个节点上，将手机和帽子节点挂在乘客节点上，并将它们与父节点之间的位置关系设置正确。到目前为止，这个场景图的搭建算是基本完成了，每个节点都在正确的位置上。

再次，将摄像机挂接在公交车的上方，使其能和公交车一起运动，于是显示的就是一个类似极品飞车中从车外部看场景的效果；如果将摄像机和司机绑定，并定位在司机的眼睛上，那就是一个第一人称运动游戏了。

经过这三步，一个基本的场景图就构造完毕了。

既然场景图是一个树结构，我们就会希望场景图中的父节点状态可以影响子节点的状态，比如如果父节点不可见，则子节点也会不可见（想像一下公车不可见，人当然就不可见了）；而且父节点的运动也会带动子节点运动，比如父节点的平移、旋转、缩放都会带动子节点一起运动。

由于场景图中的所有节点都必须满足上面的需求，所以可以先设计一个节点基类，场景图中挂载的所有节点都继承自该基类。

例程3-1是一个最简单的场景图节点基类的声明，这个基类包含了所有基本信息。

**例程3-1　最基本的场景图节点基类声名**

```
classGEObject
{
```

```
Protected:
    string m_strName;                    // 节点的名字，唯一
    bool   m_bIsVisible;                 // 是否可见
    GEObject*m_pFather;                  // 父节点指针
    list<GEObject*> m_pChildList; // 子节点链表
    Vector3m_vPositionRelative;          // 相对父节点位置
    Matrix3m_mRotateRelative;            // 相对父节点的旋转矩阵
    float   m_fScale;                     // 模型整体的缩放比例（不会影响子节点）
    Matrix4m_mTransformation;             // 世界变换矩阵（只包括平移旋转，无缩放）
    Matrix4m_mFinalTransformMatrix;      // 最终变换矩阵（包括平移，旋转和缩放）
}
```

在该基类中，m_strName属性是节点的名字。每一个节点都有一个名字，该名字在一个场景图中必须是唯一的，因为场景图中节点的搜索是通过名字这个关键词来搜索的。如果在建立场景图的时候不小心将两个节点赋了同样的名字，则下次要得到其中一个节点的时候可能得到的并不是所希望的那个，这时，整个系统就会不稳定。所以，在新建节点的时候必须注意命名的不重复性。当然，系统也会告诉用户是否定义了一个重名，这就是场景管理器的功能了。关于场景管理器的设计，在后面会详细讲述。

一个节点可以是可见的也可以是不可见的，m_bIsVisible属性就定义了一个节点是否可见。对于不可见的节点就不需要绘制，同时它的子节点也不需要绘制。可以通过图3-3的场景图来说明这个结论。想像一下，如果公交车有隐形功能（当然，只是假设），那么车就是不可见的，既然车不可见了，那么车中的人和人身上的物品也自然不可见了。所以说，父节点的可见性会影响子节点的可见性。除此之外，

不可见节点的有向包围盒（OBB）和节点包围球（Bounding Sphere）都不需要更新。

父节点指针和子节点链表这两个属性可以说是场景图的根基，是它们将场景图有序地联系起来。m_pFather存储了节点的父节点指针。一个节点可以有父节点也可以没有父节点，当它没有父节点时就是一个根节点；m_pChildList存储的是子节点链表，节点的子节点数量没有限制，而当它没有子节点时就是一个叶节点。图3-3中，乘客1是公交车的子节点同时也是手机和帽子的父节点。

那么，父节点和子节点是如何联系起来的呢？这就是属性m_vPosition-Relative和m_mRotateRelative的作用所在了。其中m_vPositionRelative表示子节点相对父节点的位移，m_mRotateRelative表示子节点相对父节点的旋转。这里，读者也许会问，为什么不存节点的绝对位置和旋转而要存相对于父节点的位移和旋转呢？在想得到它的位置时还要一层一层地去查它的父节点，不是很不直接吗？理论上，如果设计得不好，的确会存在这个问题，但是在明白场景图的实质后，就不会再这样认为了。在场景图中，父节点的旋转、移动和缩放都会影响子节点的状态，假设乘客1走动或旋转，则其帽子、手机都会跟着一起运动，但事实上它们相对乘客1来说是静止的，所以在乘客1自身更新后帽子和手机的坐标自然就更新了。这样一来，对一个节点操作就无需关心它的父节点或子节点，非常方便。

缩放属性m_fScale是一个浮点型。在这里，没有支持非整体缩放（如长宽放大5倍，而高放大3倍），因为在非整体缩放时还要考虑物体法向的变化，非常得麻烦，并且大多数的情况都是整体缩放，所以这里就只考虑了整体缩放。看到这里，读者也许会问：为什么存相对位移和旋转却不存相对缩放呢？原因在于父节点的缩放因子不会直接

影响子节点的缩放。可以想像公交车缩小了，那么乘客1也会同时缩小。接着乘客1下车，这时乘客1节点先脱离公交车这个父节点，然后挂到场景图的根节点上去。这时如果我们存储的是相对缩放因子，人就会放大，而这显然不是希望的结果。但如果我们存储的是世界缩放因子，则乘客1就不会因为下车而放大了。这就是我们选择绝对缩放因子的原因。

最后两个属性分别是节点的世界平移缩放矩阵（m_mTransformation）和最终的世界平移旋转缩放矩阵（m_mFinalTransformMatrix）。事实上，这两个属性不是必须的，通过一些计算也可以得到这些值，但为了更好的效果，只能多消耗一点内存了。

除了基本属性以外，这个节点类还要有一些基本操作。在介绍这些操作前，需要在节点类里再增加一个数据成员：

```
boolm_bDirty; // 是否需要重新计算最终变换矩阵（因为移动、旋转或缩放）
```

当m_bDirty为真时节点的世界变换矩阵才需要更新，否则便无需更新。该属性可以避免不必要的计算，提高效率。下面具体分析各个类的操作。

首先是最简单也是最基本的Get操作。如例程3-2所示。

### 例程3-2　场景图节点基类的Get操作

```
/ **得到当前位置（世界坐标系中）*/
inlineVector3     GetPosition()const
{ returnm_mTransformation.GetOffset();}
/ **得到前向向量z（规一化的向量）*/
```

```
inlineVector3     GetDir()const
{ returnm_mTransformation.GetDirUnnormalized();}
/ **得到右向向量x（规一化的向量）*/
inlineVector3     GetRight()const
{ returnm_mTransformation.GetRightUnnormalized();}
/ **得到上向向量y（规一化的向量）*/
inlineVector3     GetUp()const
{ returnm_mTransformation.GetUpUnnormalized();}
/ **得到物体的旋转矩阵*/
constMatrix3& GetRotationMatrix()const;
/ **得到世界变换矩阵（经过平移、旋转，不包括缩放）*/
inline constMatrix4& GetTransformMatrix()const
{ returnm_mTransformation;}
/ **得到最终世界变换矩阵（包括平移、旋转和缩放）*/
inline constMatrix4& GetFinalTransformMatrix()const
{ returnm_mFinalTransformMatrix;}
/ **得到相对位置*/
inline constVector3& GetPositionRelative()const
{ returnm_vPositionRelative;}
/ **得到相对前向向量z（规一化的向量）*/
inlineVector3     GetDirRelative()const
{ returnm_mRotateRelative.GetRow(2);}
/ **得到相对右向向量x（规一化的向量）*/
inlineVector3     GetRightRelative()const
{ returnm_mRotateRelative.GetRow(0);}
/ **得到相对上向向量y（规一化的向量）*/
inlineVector3     GetUpRelative()const
{ returnm_mRotateRelative.GetRow(1);}
/ **得到物体的相对旋转矩阵*/
inline constMatrix3& GetRotationMatrixRelative()const
{ returnm_mRotateRelative;}
/ **得到物体的相对变换矩阵（只有旋转和位移信息，无缩放信息）*/
constMatrix4& GetTransformMatrixRelative()const;
floatGetScale()const{ returnm_fScale;}
```

```
/ **检测是否有这个子节点*/
GEObject *CheckHasChild(const string& childName)const;
/ **得到子节点列表*/
inline list< GEObject*> & GetChildList()
{ returnm_pChildList;}
```

这些函数基本上都是内联函数，效率较高，并且一般都很简单，一看即明，不需要太多的讲解。函数GetRotationMatrix()和函数GetTransform-MatrixRelative()的定义如例程3-3所示，也比较简单，只是用了点小技巧返回一个常数引用，提高了函数效率。（注：返回常数引用可能带来潜在问题，但由于Matrix类数据量较大且该函数引用次数较多，才采用该解决方案）从这个小地方我们也可以看到，在游戏引擎开发过程中，效率始终是一个不能忘记的要点。设计人员要争取在同样时间内画更多帧，完成更多任务。

**例程3-3　GetRotationMatrix()和GetTransformMatrixRelative()的实现**

```
constMatrix3& GEObject::GetRotationMatrix()const
{
    staticMatrix3mRot;
    mRot.SetRow(0, GetRight());
    mRot.SetRow(1, GetUp());
    mRot.SetRow(2, GetDir());
    returnmRot;
}
constMatrix4& GEObject::GetTransformMatrixRelative()const
{
    staticMatrix4m;
    m = m_mRotateRelative;
    m.SetOffset(m_vPositionRelative);
    m.SetScale(1);
```

```
    m.SetProj(Vector3::ZERO);
    returnm;
}
```

现在看一下节点类中最重要的函数群——几何变换函数，它包括平移函数、旋转函数和缩放函数。例程3-4说明了平移函数的实现。在Move函数中，只要改变该节点相对父节点的位移即可（需要注意父节点有自旋转的情况），同时将该节点的更改标志符置位。这个标志符就是新加入的m_bDirty，当它置位时表示该节点的变换矩阵需要更新；之后更新节点包围球。关于节点包围球的话题会在第3.2节详细讨论。MoveTo函数的作用是调用了Move函数，丰富了用户接口却没有增加维护成本。

**例程3-4　平移函数的实现**

```
// 移动一段距离
voidGEObject::Move(constVector3& offset)
{
    if(offset.IsZero())
        return;
    // 不要忘了考虑父节点的旋转矩阵的影响
    staticMatrix3 _mrv;
    _mrv = Matrix3::IDENTITY;
    if(m_pFather)
m_pFather->m_mTransformation.Extract3x3Matrix(_mrv);
    _mrv = _mrv.Transpose();
m_vPositionRelative + = offset *_mrv;
    m_bDirty = true;              // 节点的最终变换矩阵需要更新
        m_bDirtyBB = true;        // 节点的包围球需要更新
}
// 从当前位置移动到目标位置
voidGEObject::MoveTo(constVector3& newPos)
```

```
{
    Move(newPos - GetPosition());
}
```

相比较平移函数，旋转函数数量上就要多一点，不过原理都是差不多的，在这个引擎中，共设计了5个旋转函数，如例程3-5所示。

**例程3-5　旋转函数的实现**

```
/ **绕自己的x轴转-点头*/
voidGEObject::RotationPitch(float radian)
{
    if(radian = = 0)
       return;
    m_mRotateRelative.RotationX(radian);
    m_bDirty = true;
    m_bDirtyBB = true;
return;
}
// 以下三个函数的实现与上RotationPitch很像，就省略其实现，请参阅 随书源
代码
/ **绕自己的y轴转- 摇头*/
voidGEObject::RotationYaw(float radian);
/ **绕自己的z轴转- 翻转*/
voidGEObject::RotationRoll(float radian);
/ **绕自己的三轴同时转*/
void  RotationPitchYawRoll(float  radianX,       float  radianY,
float
radianZ);
/ **将该节点方向定位到朝向vDir，上方向是vUp */
void GEObject::SetDirectionRelative(const Vector3& vDir, const
Vector3&
vUp)
{
```

```
Matrix4::LookTowardsUp(vDir, vUp)
.Extract3x3Matrix(m_mRotateRelative);
m_bDirty = true;
m_bDirtyBB = true;
}
```

最后是缩放函数，如例程3-6所示。这个函数要稍微复杂一点，用到了递归实现。这主要是因为存储的是整体缩放因子，所以父节点的缩放会影响子节点的缩放因子。另一方面，在节点缩放后，该节点和其所有子节点的距离都会发生改变，所以用递归实现会简单漂亮。

### 例程3-6　缩放函数的实现

```
voidGEObject::Scale(float scale)
{
    m_fScale *= scale;
    m_bDirty = true;
    m_bDirtyBB = true;
    for(list< GEObject*>::iterator ptr = m_pChildList.begin();
        ptr! = m_pChildList.end(); + + ptr)
{
        (*ptr)-> m_vPositionRelative *= scale;    // 父子的相对距
离发生改变
        (*ptr)-> Scale(scale);
    }
}
```

完成了几何变换函数后，下一个需要解决的就是最终世界变换矩阵的求法了。其实求这个矩阵是不难的，但还是要注意那个被重复提及的词——效率。要让算法的重复计算次数最少，不作不必要的更新或是计算。

## 例程3-7　更新节点及其子树的变换矩阵函数定义

```
/**更新自己的变换矩阵（从自身开始寻根）
*@return [constMatrix4&]：计算后的最终变换矩阵(不包括缩放)
*@note：不会更新父节点或子节点
*/
constMatrix4&GEObject::UpdateTransformation()
{
    m_mTransformation=GetTransformMatrixRelative();
    GEObject*pNode=m_pFather;
    while (pNode)
{
        m_mTransformation=m_mTransformation
                                                                  *pNode-
>GetTransformMatrixRelative();
        pNode=pNode->m_pFather;
    }
            m_mFinalTransformMatrix=Matrix4::GetScale(m_fScale,
m_fScale, m_fS-cale)
              *m_mTransformation;
    if (m_pMesh)
        m_OBB.UpdateOBB();
    m_bDirty=false;
    returnm_mTransformation;
}
/**更新以自己为根的一棵子树的变换矩阵
*@param [bool] compulsive：是否强制更新
*/
voidGEObject::UpdateSubSceneGraph(boolcompulsive)
{
    //先更新自己
    booldirty=m_bDirty||compulsive;
    if (dirty)
    {
```

```
        m_mTransformation=GetTransformMatrixRelative();
        if (m_pFather)
            m_mTransformation*=m_pFather->m_mTransformation;
           m_mFinalTransformMatrix=Matrix4::GetScale(m_fScale，
m_fScale，
                m_fScale)*m_mTransformation;
        if (m_pMesh)
            m_OBB.UpdateOBB();
        m_bDirty=false;
    }
    //更新儿子（自己不脏，但可能儿子脏）
    for (list<GEObject*>::iteratorptr=m_pChildList.begin();
        ptr!=m_pChildList.end();++ptr)
    {
        (*ptr)->UpdateSubSceneGraph(dirty);
    }
}
```

例程3-7中有两个函数，分别实现更新自身节点（UpdateTransformation()）和更新以该节点为根节点的子树（UpdateSubSceneGraph()）。

首先，函数UpdateTransformation()实现了自身世界变换矩阵的更新。节点的世界变换矩阵是由该节点的缩放矩阵和该节点的世界平移旋转矩阵相乘得到的；而在我们定义的场景图结构中，一个节点的世界平移旋转变换矩阵是通过该节点的相对平移旋转矩阵与其所有父节点的相对平移旋转矩阵相乘得到的。因此，可以从节点开始，层层搜索其父节点，得到节点的世界平移旋转矩阵，然后将该节点的缩放矩阵和刚得到的世界平移旋转矩阵相乘得到最终世界变换矩阵。至此，标志该节点为没有更改。在这个算法中，节点的父节点是否更改不影响我们的最后结果；同时，我们也不改变父节点的更改标志属

性。应该说，这个算法的效率不是那么高，因为它没有对父节点没有更改的情况做优化。

函数UpdateSubSceneGraph()更新了以该节点为根节点的子树。在写算法前，需要首先明白什么时候需要更新最终变换矩阵：在节点自身运动过后或是父节点运动过后需要更新节点的最终变换矩阵。因此，该函数有一个参数（bool compulsive），表示是否强制更新该节点，当有父节点运动后该参数置位。

在对更新条件了解后再来看一下函数实现。这是一个自上而下的更新，因为只有当父节点更新后子节点的更新才有意义，所以第一步操作是评断是否需要更新自身节点，若是，则更新该节点的变换矩阵。接下去是更新一层子节点，这里用了递归实现。如果自身节点更新了，则子节点的强制更新参数置位，函数结束。可以看出，在场景图没有运动时，这个函数是不会有什么消耗的；即使有节点运动，它也只是更新运动节点的子树。并且节点的世界变换矩阵的更新比UpdateTransformation()函数要简单很多，它总是假定父节点是最新的，所以子节点世界变换矩阵的计算只需要用父节点的世界变换矩阵和节点的相对变换矩阵相乘即可以得到。是不是比你想像中简单？函数虽然非常短小，但却同时实现了功能还保证了效率。写完这个函数，还需要问自己一个问题：这个函数在什么时候被调用？被谁调用？在游戏中，场景图每一帧都要被更新，但每一帧的更新次数不能超过一次！所以这个函数是在每一帧绘制前被场景的根节点调用的。在被根节点调用后，整个场景图就更新了。

最后要介绍的是函数群Attach和Detach系列，函数群负责子节点的挂载和卸载。共有四个函数，分别实现挂载一个子节点、卸载一个

子节点、将自身挂载到一个父节点上去和卸载所有子节点的功能。函数实现如例程3-8所示。

**例程3-8　子节点挂载卸载函数群**

```
boolGEObject::Attach(GEObject*father)
{
    if (father==m_pFather)
        return false;
    //先告诉原父节点，detach本节点
    if (m_pFather)
        m_pFather->Detach(m_strName);
    m_pFather=father;                    //然后设置新父节点
    m_pFather->UpdateTransformation();   //更新父的变换矩阵
    //更新儿子的相对变换矩阵（平移、旋转）
    Matrix4mChildRelative=m_mTransformation
                                                    *m_pFather-
>m_mTransformation.Inverse();
    mChildRelative.Extract3x3Matrix(m_mRotateRelative);
    m_vPositionRelative=mChildRelative.GetOffset();
    //让其新父亲认儿子
    m_pFather->m_pChildList.push_back(this);
    //包围球的更新
                if    (!m_BoundingSphere.IfInSphere(m_pFather-
>m_BoundingSphere))
        m_pFather->m_bDirtyBB=true;
    return true;
}
boolGEObject::Detach(conststring&name)
{
    list<GEObject*>::iteratorp;
    for (p=m_pChildList.begin();p!=m_pChildList.end();++p)
    {
        //告诉儿子，要卸载它，然后，儿子也就没有父亲了
```

```
        if ((*p)->GetName()==name)
        {
             (*p)->UpdateTransformation();     //更新儿子的世界变换矩阵
            (*p)->m_mRotateRelative=(*p)->GetRotationMatrix ();
            (*p)->m_vPositionRelative=(*p)->GetPosition();
            (*p)->m_pFather=NULL;
            m_pChildList.erase(p);
            m_bDirtyBB=true;
            return true;     //卸载成功
        }
    }
    return false;     //没有这个儿子
}
//下面两个函数比较简单，可以参考随书代码
voidGEObject::DetachAll();
boolGEObject::AddAttach(GEObject*child);
```

这个函数群中Attach函数和Detach函数是最重要的。Detach函数的功能是删除被调用节点的一个名字为函数参数的一层子节点。首先，被调用节点（暂定为节点A）遍历其子节点，找到名字为函数参数的节点，我们称它为节点D；接着，将节点D的世界变换矩阵、相对平移向量和相对旋转矩阵都更新，使之不再依赖父节点，这时节点D的相对变换矩阵和世界变换矩阵的值就是一样的了；然后，将节点D的父节点设为空，同时将节点D和A节点的链接删除；最后，告诉父节点更新包围球（这部分内容将在第3.2节介绍）。

Attach函数是传入一个可以被挂载的父节点指针，然后函数调用节点就被挂载到该父节点上去了。这个函数的大部分逻辑和Detach函数很像，但不要忘记先将子节点从原父节点上卸载。

至此讲解的是一个最基本的场景图节点基类，在这个基类的基础上可以增加很多属性，比如速度、加速度、运动、模型、动画等。另外还有很多框架函数需要声名，比如初始化函数、渲染函数、动作处理函数、结束函数等。在基类中定义这些函数有助于框架的统一。关于这些属性和框架函数的定义，本书由于篇幅有限就不细说了，读者可以通过本书附带的源代码光盘中学习。

在有了节点基类后，可以根据应用的需要派生出子类。在GEEngine中，场景图中所有的类都是继承自这个基类的，比如灯光类、照相机类、人物类、粒子系统类等。图3-4说明了引擎中场景图节点类和其派生类的关系。

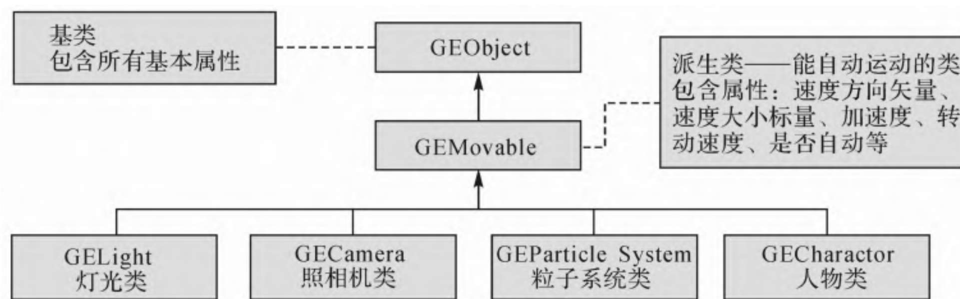



图3-4　场景图节点中各类的继承关系

在对场景中的物体抽象后，又是通过什么来组织和管理场景图的呢？这就需要场景图管理类（类GESceneManager）的支持了。类GESceneManager提供了一些接口操作场景图，比如创建一个节点、删除一个节点、搜索一个节点、场景图的更新、动作和显示，都是通过这个类来实现的。与节点类相比，GESceneManager要简单很多，只需要不断调用在节点实现的接口就好了。不过，这个类要担负起方便用

户管理的责任。比如创建一个照相机类的函数声明和使用如例程3-9所示。

### 例程3-9　创建照相机节点函数声明及使用

```
//classGESceneManager:
GECamera*CreateCamera(const string&name, GEObject*father,
        constVector3&offset=Vector3::ZERO, float radianX=0.f,
        float radianY=0.f, float radianZ=0.f, float fov=PI/2,
        floatnearclip=0.17f, float farclip=1000.f);
//创建一个照相机节点
GESceneManager::GetSingleton()->CreateCamera("camera",
GEObject::c_
                                                Root,
                                                 Vector3(100,2,
0));
```

给出一些参数（事实上很多参数都可以是默认的）就可以非常方便地插入一个照相机节点。然而，例程中的GetSingleton()是什么呢？其实，这里有个设计模式的思想——单例模式。一个游戏中只有一个场景管理器，所以我们在类中定义一个GESceneManager的对象，它是一个静态对象。对于这种单例模式的应用可以参照例程3-10。在引擎的设计与实现中使用了很多单例模式，所以必须要了解它是如何实现的。而关于GEObject::c_Root的解释是：限定一个引擎应用中只能有一个场景图，这样就只有一个根节点，将它设为类静态成员可以方便操作，减少不必要的潜在错误。

### 例程3-10　关于单例模式在引擎中的应用

```
classGESceneManager{
public:
```

```
        inlineGESceneManager&GetSingleton()         {return
c_SceneManager;}
    inline constGESceneManager&GetSingleton() const
    {return c_SceneManager;}
protected:
    staticGESceneManagerc_SceneManager;  //静态类对象
    GESceneManager();    //构造函数说明为保护，防止外部生成新的实例
}
```

讲到这里，读者应该对如何设计3D场景图有了一定了解。不过，如果看过OGRE的源代码，应该有所疑问：在OGRE所有的叶节点才是对象节点，而所有枝节点都是逻辑节点，那么为什么没有这么做呢？我想，这也是我们的引擎在设计上的一个优点！

先来看看OGRE是如何组织场景的。首先，和我们一样，OGRE为了解决场景管理的问题提出了几个重要的概念并将它们实现为引擎中的类。

- Entity：场景元素，Mesh（模型）在场景中的实例。
- Light：场景元素，现实世界的光在场景中的实例。
- Camera：场景元素，现实世界的观察者在场景中的实例。
- SceneNode：抽象的场景管理单元。

注意到SceneNode和其他几个概念不一样，其他几个概念都是现实事物的程序抽象，而SceneNode在现实世界中并不存在，它是OGRE为了组织场景而提出的抽象概念，这也就是刚才说的逻辑节点。

OGRE场景管理和我们引擎的场景管理相似，都是将现实世界中的场景划分成抽象的不同空间，区域中还可以划分成不同的小空间，每个空间由一个节点对象来管理。但是，这个节点对象（SceneNode）只

处理移动、旋转和缩放等与空间相关的行为，并不是一个实体。然后，在每个SceneNode上可以挂接各种场景元素（如Entity、Light、Camera等），而且场景元素本身并不负责与空间位置相关的行为，全部交给SceneNode来做。可以看出，OGRE将逻辑节点和对象节点分开了。图3-5是一张符合OGRE场景管理模式的场景结构图。

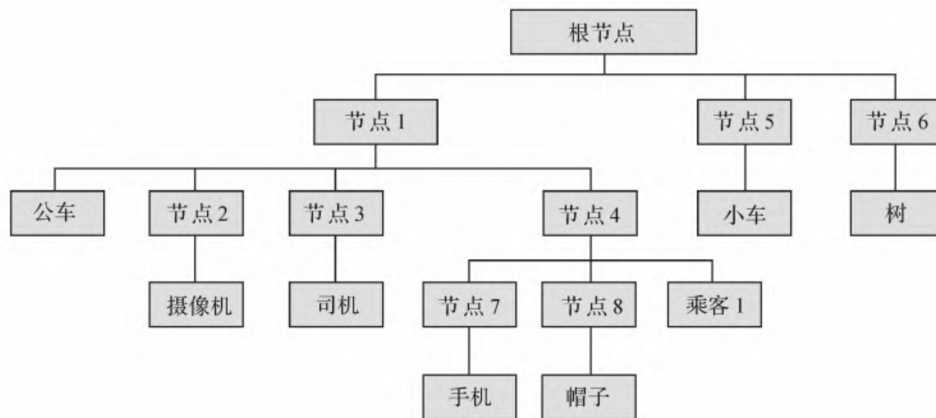



图3-5　OGRE的场景图实例

看晕了吗？这可是和图3-3一样的场景图！相同的对象数，但是节点数整整翻了一倍。对比OGRE，我们将对象节点和逻辑节点的概念进行了合并，一个枝节点可以是一个对象节点也可以是一个逻辑节点。这种设计大大简化了场景图的复杂程度而且也更加贴近真实世界的情况。

# 3.2　有向包围盒

有向包围盒（Oriented Bounding Box，OBB）是一个最贴近物体的长方体（图3-6）。它随着物体的移动而移动，随着物体的旋转而旋转，当物体缩放时，它也一起缩放。简而言之，它是一个可旋转的轴

平行包围盒（Axis Aligned Bounding Box，AABB）。在场景管理中，它被用作最基本的碰撞检测。

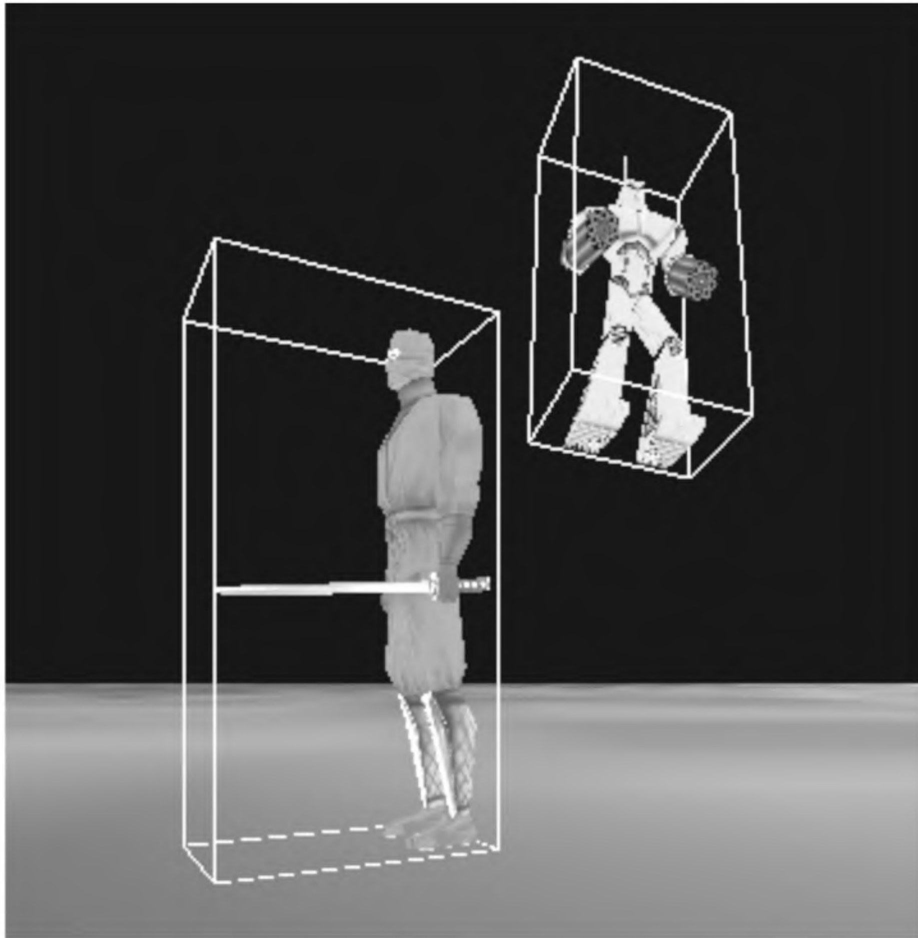


图3-6　有向包围盒

先来看看OBB类的声明，如例程3-10所示。每个有向包围盒都有一个中心点，这个中心点不一定是长方体的重心，它是所包围物体模型的中心，也就是说，中心在哪里取决于美工建模的时候定在哪里。为什么把OBB的中心放在模型的中心而不是长方体的重心呢？这是为了方便物体旋转的时候包围盒的更新。物体旋转是围绕美工给它定的中心旋转的，所以，它的包围盒也围着这个中心旋转。有了中心点的概念

后，我们来看m_fRight、m_fLeft、m_fFront、m_fBack、m_fUp、m_fDown这6个数据成员，它们分别代表包围盒六个表面离中心点的距离。这6个数据在读取模型的时候就赋值了，也就是说，它们存储的是未经变换的模型的包围盒的六个面与中心点的距离，并且这些距离是带正负号的，比如中心与左面的距离一般都是负数。

之后的两个数据成员是m_fRadiusOriginal和m_fRadius，他们分别代表变换前与变换后包围盒对角线距离的一半。m_fRadiusOriginal在读取模型的时候赋值，而m_fRadius在节点缩放后才重新计算。为什么要有这两个数据成员呢？它们都是为后面将要介绍的节点包围球服务的。

成员m_OBB是有向包围盒生成的碰撞检测用的六面体。而m_Object是包围盒包围的那个物体的指针。所有包围盒的更新都是通过这个指针得知的。

包围盒的更新（UpdateOBB）是OBB类中最重要的函数之一，函数负责更新长方体六个面的方程、并更新半对角线长和长方体的体心。这个函数只有在物体移动、旋转或是缩放时才会调用，避免不必要的更新。同时，虽然该函数较长，但是内部实现却非常简单。调用函数GetFinalTransformMatrix()、GetRight()、GetUp()、GetDir()和GetScale()等都是不怎么需要计算的内联函数，保证了函数的执行效率。

### 例程3-11　有向包围盒类

```
classOrientBoundingBox
{
public:
```

```
    void Init(GEObject*object, float fRight, float fLeft, float
fUp,
                float fDown, float fFront, float fBack);
    floatGetRadius() const{returnm_fRadius;}
    constVector3&GetCenter() const{returnm_vCenter;}
    constFrustum&GetOBB() const{returnm_OBB;}
    voidUpdateOBB();
protected:
    GEObject*m_Object;              //有向包围盒的主体
    Vector3m_vCenter;               //OBB的体心
    float m_fRight;                 //OBB的六个面离中心点的距离（有正
负）
    floatm_fLeft;
    floatm_fFront;
    floatm_fBack;
    floatm_fUp;
    floatm_fDown;
    floatm_fRadiusOriginal;         //原始包围盒对角线的一半（未缩放）
    floatm_fRadius;                 //包围盒对角线的一半
    Frustum m_OBB;                  //六面体
};
```

# 3.3　节点包围球

在绘制的时候，不可见范围内的物体是不需要绘制的，甚至是不需要更新的。所以，在三维场景的绘制中如果能剔除不在视锥内的物体，将会极大地改善游戏运行速度，在大场景中尤其明显。三维场景的物体剔除有很多种方法，但其中最简单，也是效率最高的就是利用节点包围球了。

在随书引擎中，每个节点有两个包围球。一个是只包围自身节点的包围球，称为自身包围球，如果这个包围球在视锥外则说明该节点

不可见，不需要绘制。另一个包围球是包围自身节点和该节点所有子节点的包围球，称为子树包围球。如果该包围球在视锥外则说明以该节点为根节点的子树都在视锥外，无需绘制。图3-7中是一个子树包围球的演示。

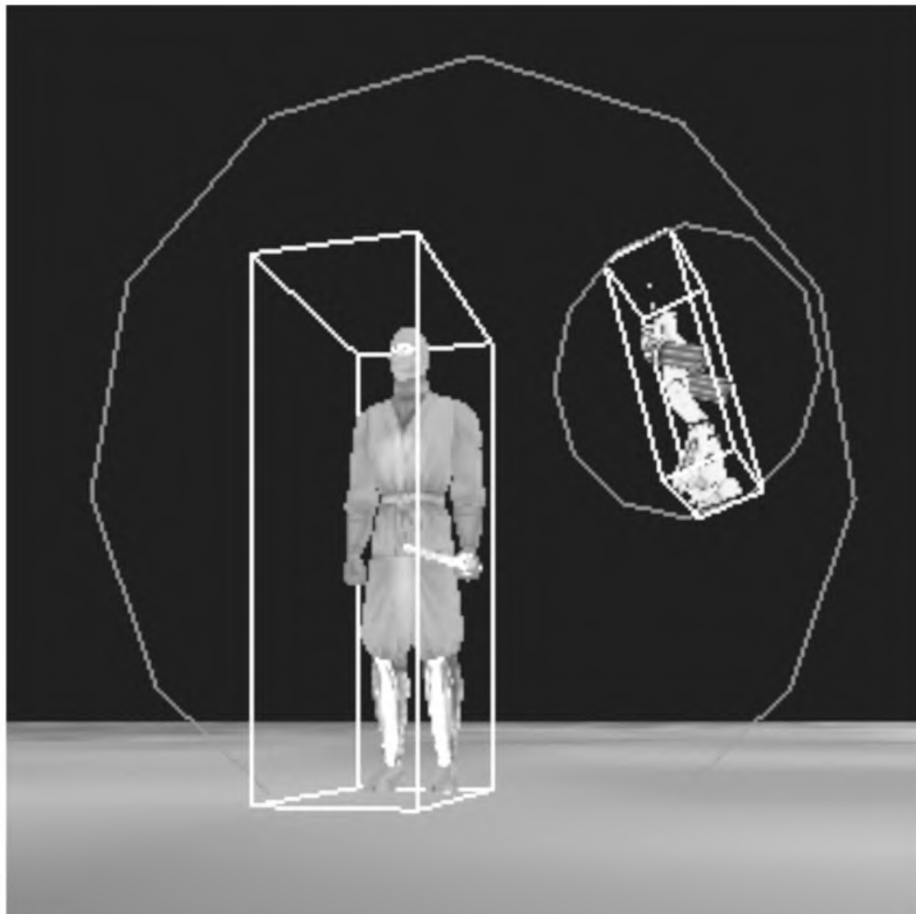


图3-7　节点包围球示例

应该说，在场景图中集成包围球不是一件难事，但在游戏引擎中，不但要保证正确性，还要保证实时性，避免所有不必要的计算。所以，编码时需要关注以下两件事：

- 什么时候需要更新节点包围球。
- 如何更新节点包围球。

分析第一个问题，可以把需要更新的情况总结为以下几点：

（1）第一次初始化的时候，初始化自身的包围球和子树包围球。

（2）自身节点有移动、旋转或是缩放，更新自身包围球和子树包围球。

（3）子节点有移动、旋转或是缩放，更新子树包围球。

（4）增加或是减少一个子节点时更新子树包围球。

再进一步分析可以得出结论：如果每个节点都管理好自己的子树包围球，那么每个父节点都只需要关心第一层子节点的子树包围球就可以更新自身的子树包围球了。这个结论非常重要，它明确了每个节点的职责——管好第一层子节点！基于这个思想，在类GEObject中增加一个数据成员bool m_bDirtyBB，它置位与否标志着该节点的包围球是否需要更新。在程序中，如下情况需要将该数据成员置位：

（1）初始化时。

（2）自身节点移动、旋转或缩放时。

（3）一层子节点的子树包围球变化时。

（4）增加或删除子节点时。

然后，再分析如何更新包围球。这是一个自下而上的更新，树的深度遍历，只有子节点更新完毕了，才更新父节点。例程3-12展示了

如何更新场景图中节点的包围球，其中函数GEObject::UpdateBoundingSphere()的功能是更新一个节点的包围球，而函数GESceneManager::UpdateBoundingSphere（Gebject* root）则是调用GEObject::UpdateBoundingSphere()去完成场景图中所有节点的包围球更新。第一个函数（GEObject::UpdateBounding-Sphere()），即单个节点的包围球的更新是分为以下几步进行的：

（1）若节点数据成员m_bDirtyBB为false，说明无需更新，返回。

（2）先更新自身包围球（从节点的有向包围盒得到）。

（3）生成一个临时包围球，赋初值为自身包围球。

（4）依次遍历一层子节点，计算临时包围球与子节点的子树包围球之间的最小包围球，并将之赋给临时包围球。

（5）比较临时包围球与该节点的子树包围球，如不同则将临时包围球赋给节点包围球，并将父节点的m_bDirtyBB置位。

第二个函数（GESceneManager::UpdateBoundingSphere（Gebject*root））则相对逻辑简单，如果你要更新整个场景图的节点包围球，可如下调用：

```
GESceneManager::GetSingleton()->UpdateBoundingSphere(GEObject::c_Root);
```

在引擎中，对于场景图的优化需在细节上下很多功夫，才能收到一定效果。但并不是说，这样的优化就够了，其实还可以做很多优化，比如背面剔除、八叉树、二叉树等。

## 例程3-12　节点包围球的更新

```
voidGEObject::UpdateBoundingSphere()
{
    //这里假设子节点的包围球都是最新的（确实也是，因为是个树的前续遍历）
    if (m_bDirtyBB==false)
        return;
    bool ifSelfBBChanged=false;     //节点包围球是否进行了修改
    //自身包围球的更新
    m_SelfBoundingSphere.SetCenter(m_OBB.GetCenter());
    m_SelfBoundingSphere.SetRadius(m_OBB.GetRadius());
    Sphere boundingSphere=m_SelfBoundingSphere;
    //如果子节点包围球改变，或该节点在外部被改动，则该节点包围球需要调整
    for (list<GEObject*>::iteratorp=m_pChildList.begin();
        p!=m_pChildList.end();++p)
    {
        //父在子外，则无需更新包围球
                        if   (boundingSphere.IfOutOfSphere((*p)-
>m_BoundingSphere))
           continue;
        //得到两圆心的向量和距离
        Vector3 v=(*p)->m_BoundingSphere.GetCenter()
                      -boundingSphere.GetCenter();
        floatd=v.Length();
        if (boundingSphere.IfInSphere((*p)->m_BoundingSphere))
        {    //说明父在子内
                                  boundingSphere.SetCenter((*p)-
>m_BoundingSphere.GetCenter ());
                                  boundingSphere.SetRadius((*p)-
>m_BoundingSphere.GetRadius ());
        }
        else    //父子相交或是相离
        {
```

```
boundingSphere.SetCenter(boundingSphere.GetCenter()+
                v.GetNormalized()*(d-boundingSphere.GetRadius()
                +(*p)->m_BoundingSphere.GetRadius())*0.5f);

boundingSphere.SetRadius((d+boundingSphere.GetRadius()
                +(*p)->m_BoundingSphere.GetRadius())*0.5f);
        }
    }
    //如果新得到的包围球和原包围球不同
    if (!(boundingSphere==m_BoundingSphere))
    {
        ifSelfBBChanged=true;
        m_BoundingSphere=boundingSphere;
        if(m_pFather)
            m_pFather->m_bDirtyBB=true;
    }
    m_bDirtyBB=false;        //自身包围球和子树包围球更新结束
}
voidGESceneManager::UpdateBoundingSphere(GEObject*root)
{
    for (list<GEObject*>::iterator p=root->GetChildList().begin
();
        p!=root->GetChildList().end();++p){
            UpdateBoundingSphere(*p);
    }
    root->UpdateBoundingSphere();
}
```

# 3.4　场景图的渲染

三维引擎中渲染总是最重要的步骤，而场景图的渲染又是重中之重。因此将场景图的渲染处理好对于引擎来说非常重要。

在场景图渲染中最重要的优化便是将场景图的节点按照绘制状态分类，对于相同状态的物体只设置一次状态。状态切换是指任意影响画面生成的函数调用，比如纹理、材质、光照、混合等函数。因为状态切换是一项非常耗时的工作，所以要尽量将相同状态的物体一起绘制，以避免不必要的状态切换。因此，在引擎中做如下处理场景图的渲染：

（1）遍历场景图，比较物体的自身包围球是否在视锥内，若不是，则无需绘制。

（2）提取节点的材质信息，根据一些关键属性分类。比如是否接受光照、是否透明、采用何种混合方式等，并按类别将节点指针存入数组。

（3）绘制不同组的节点。

在我们的引擎中，类GERenderQueue的作用便是存储分类后的节点。目前有以下五种类型的节点：透明的节点、粒子系统节点、平面阴影投射面节点、平面阴影投射物节点和除此之外的普通节点。另外也可以根据具体要求增加和减少类别，比如是否受光照、是否有Shader、是否有LOD等，接口非常简单。

因为一个引擎应用中只有一个渲染队列，所以也将类GERenderQueue设计成单例模式。它有3个最重要的函数，分别是：

- HRESULT Render()；渲染所有经过排序的节点。
- void ClearRenderQueue()；清除渲染队列，准备下一帧的调用。

- inline size_t PushToXXXRQ（GEObject*obj）；将一个节点加入某一个渲染队列，并返回渲染对列的大小。

应该说，类GERenderQueue非常的简单，但简单不代表效果差，相反，场景图节点的渲染重排序对于提高场景的渲染效率非常重要。

# 第4章　三维渲染管道的设计

游戏引擎的一个重要任务是管理游戏的数据和艺术内容，并决定把什么内容绘制到屏幕上以及如何绘制。当游戏程序员构造一个渲染器时，如何将三维空间的物体渲染到二维的计算机屏幕上去，永远是他设计的第一步。这个工作也许看似简单，但所谓的好与坏的差别就在于你是否能够在同样的时间内显示更多的内容，或是说显示同样的内容占 用尽可能少的时间。

在本章，读者将会了解到以下内容：

- 什么是渲染器、它是如何工作的。
- 如何设计符合需要的渲染器。
- 如何管理材质。
- 如何利用顶点缓冲区和索引缓存区提供渲染效率。
- 如何定义模型并渲染。

## 4.1　渲染器

渲染器是一个游戏引擎中最重要的模块之一，如果想做一款最新潮的、最吸引眼球的游戏，一个高效的渲染器必不可缺！不过也不是任何时候都需要把渲染器做的那么顶尖，比如当设计一款休闲类游戏的时候，就没有必要花太多的时间来完善渲染器了。此外，如果是一个游戏引擎入门者，那么更不要花太多的时间在那些最新的技术上，因为旧的技术就已经学不完了。

在DirectX的SDK中，提供了一个Windows下的Direct3D通用文件框架（在SDK目录下的Simples\C++\Common\Src），这个框架自从推出后就为许多游戏编程社团所接受了。这个框架能够帮助开发者提高开发效率，尤其是初学者，因为：

- 通常它可以帮助设计人员避免许多在Direct3D编程中面临的“怎么做”的问题，这样就可以将注意力更多地集中在真正的内容上。
- 它是一个被普遍应用的且经过严格测试的基础起点，可以大大减少调试时间。
- 所有DirectX SDK上的DirectX3D示例都使用了这个框架，因此通过学习这些示例可以了解如何使用它。
- 自己开发的产品代码是可以基于这些通用文件的，所以掌握如何使用这个框架将会加快开发进度。

本节将会介绍如何基于这个框架构造游戏框架和渲染器。

首先，需要在框架的基础上定义自己的应用类（引擎中是GEApp），这个类封装了引擎的底层应用框架，它包含了一些框架函数，这些函数都是纯虚函数，必须被子类继承。

（1）virtual HRESULT ConfirmDevice（D3DCAPS9*，DWORD，D3DFORMAT，D3DFORMAT）：这个函数会被最先执行，它是用来检查显卡能力是否支持需要的特性。比如显卡是否支持模板缓冲区、是否支持Shader之类的。如果显卡不支持，框架会切换到使用参考光栅器，或切换为顶点软件处理，或者提示出错。

（2）virtual HRESULT OneTimeSceneInit()：所有游戏中一次性初始化的东西都是在这里初始化的。在这个函数中初始化的事物是与设备无关的东西。函数需要和FinalCleanUp()搭配使用，因为初始化的数据可以在Fi-nalCleanUp()中销毁。

（3）virtual HRESULT InitDeviceObjects()：该函数负责初始化与设备相关的对象。在D3D设备最初初始化或应用中发生设备丢失、改变后重新初始化设备时需要调用此函数。函数与Delete DeviceObjects()搭配使用。

（4）virtual HRESULT RestoreDeviceObjects()：当应用程序的窗口大小改变时需要调用此函数，它负责设置窗口的投影矩阵和渲染状态等等。除了在程序开始时以外，该函数都会和Invalidate DeviceObjects()成对调用。

（5）virtual HRESULT FrameMove()：该函数每一帧都会执行。所有动画的代码都是在该帧执行。此外，如果有需要实时更新的代码，也会在该处执行，比如场景图中的包围球、包围盒更新就是在该处执行的。并且所有的交互操作也是在这里执行的。因此，该函数非常重要。

（6）virtual HRESULT Render()：作为渲染的入口点在每一帧都被调用，任何模型的绘制、GUI的绘制、纹理的绘制、灯光的应用、材质的应用都是在该函数内部实现的。和FrameMove的重要性一样，它也是引擎框架中的最重要函数之一。

（7）virtual HRESULT InvalidateDeviceObjects()：当显示窗口大小发生改变前，需要首先将原设置无效化，这就是此函数的功

能。

（8）virtual HRESULT DeleteDeviceObjects()：在D3D设备被改变时，需要先销毁原来的设备相关资源。

（9）virtual HRESULT FinalCleanup()：销毁几何数据和文件对象等非设备资源。它是游戏应用被摧毁前的最后一步，需要保证内存没有被泄漏。

这些函数都有非常严格的执行顺序，为了防止引擎用户更改其执行顺序，可以使用设计模式中的模板方法对它们进行封装。如此，当用户构建自己的实例时，就可以不用关心它们是如何被调用的了。

程序启动的执行顺序是：

ConfirmDevice()->OneTimeSceneInit()->InitDeviceObject()->RestoreDeviceObjects()

程序运行期间会执行一个循环：

FrameMove()->Render()

如果运行期间改变窗口大小，框架会调用：

InvalidateDeviceObjects()->RestoreDeviceObjects();

如果更改设备（HAL或REF），调用顺序是：

InvalidateDeviceObjects()->DeleteDeviceObjects()->InitDeviceOb-jects()->RestoreDeviceObjects();

序退出时执行：

InvalidateDeviceObjects()->DeleteDeviceObjects()->FinalCleanUp()

在有了底层框架类GEApp后，如果要实例化一个应用，就必须继承类GEApp，并实例化这些纯虚框架函数。随书引擎中有一个GEApp的子类MyApp，它是将引擎的其他模块很好地融入到了游戏框架中。比如说，My-App的FrameMove方法继承了GEApp的方法，它的内部实现如例程4-1所示，其中封装了输入的更新、游戏逻辑的处理、GUI的更新、场景的更新等。

### 例程4-1 MyApp::FrameMove()

```
HRESULTMyApp::FrameMove()
{
    GEMouse::GetSingleton()->Update(m_hWnd);    //鼠标更新
    GEKeyboard::GetSingleton()->Update(m_hWnd);    //键盘更新
    m_Game.FrameMove(m_fElapsedTime);    //游戏类更新
    GEGUIManager::GetSingleton()->FrameMove();    //GUI更新
    GESceneManager::GetSingleton()->FrameMove(m_fElapsedTime);//场景更新
    m_pCamera->MakeViewTransform();    //相机更新
    m_pCamera->MakeProjectionTransform();
    returnS_OK;
}
```

但有了框架后还不够，还需要完善和填充。接下来要提到的是类GE-GraphicLayer，这是一个图形层类，是渲染器的最重要的类。它封装了Di-rect3D的接口，目的是尽可能地不让客户程序员接触Direct3D的内容。在GEGraphic类中，封装了以下几个数据成员：

```
{
    HWND                        m_hWnd;                 //主窗口句柄
    D3DPRESENT_PARAMETERS       m_d3dpp;                //设备创建的参数
    LPDIRECT3D9                 m_pD3D;                 //D3D指针
    LPDIRECT3DDEVICE9           m_pd3dDevice;           //D3D设备指针
    D3DCAPS9                    m_d3dCaps;              //设备能力的描述
    D3DSURFACE_DESC             m_d3dsdBackBuffer;      //后缓冲区表面描述
    int                         m_iW indowRCW idth;     //窗口模式下窗口的宽
    int                         m_iW indowRCHeight;     //窗口模式下窗口的高
    int                         m_iFullScreenW idth;    //全屏模式分辨率的宽
    int                         m_iFullScreenHeight;    //全屏模式分辨率的高
    float                       m_fFov;                 //相机的FOV
    float                       m_fNear;                //相机的近面距离
    float                       m_fFar;                 //相机的远面距离
}
```

在窗口可以显示前，还需要初始化D3D设备，这个过程比较固定，一般都是遵循DirectX的SDK来做。

首先，必须创建D3D，这对应引擎中的HRESULT GEGraphicLayer::CreateD3D9()方法。这一步中，需要根据D3D版本号创建相应的D3D对象，并赋给类的D3D指针。然后得到并保存所有的显示卡、模式和设备信息，这些信息会在应用程序查询显示卡能力的时候使用。

接着，需要初始化三维显示环境。这个步骤要稍微复杂一些，简单地说，可以分为三步：

（1）选择需要创建的设备的参数，比如屏幕大小、软件模拟模式或是纯硬件模式、窗口模式或是全屏独占 模式等等。

（2）根据选择的模式创建D3D设备。

引擎中的函数HRESULT GEGraphicLayer::CreateD3DDevice()实现了这个步骤。

（3）重定位窗口，存储设备参数并保存后备缓冲区。

在程序运行中，可能还要处理设备丢失和设备销毁，这些功能也必须在类GEGraphic内实现。

在程序绘制的时候，每帧绘制都要遵循一定的顺序，以保证效率和正确性：第一步是清空缓冲区，调用函数标志开始绘制，接着绘制物体和纹理到后备缓冲区，当所有物体都绘制完毕后需要调用函数标志绘制结束，最后将后备缓冲区中的内容显示出来。

GEGraphic类除了需要提供最基本的框架外，还需要提供绘制相关接口，基本的接口包括以下几个：

- 灯光相关函数，包括设置灯光、得到灯光、关闭灯光等。
- 纹理设置、纹理状态设置。
- 渲染状态设置。
- 顶点格式设置。
- 点绘制、线绘制。
- 设置投影矩阵相关。
- 设置视域矩阵相关。

这些函数在类GEGraphic中都有定义，读者可以自行参照源代码学习。

## 4.2　材质管理

在DirectX中，材质对应类D3DMATERIAL9，它包括材料的漫反射系数、环境光反射系数、镜面反射系数、自发光系数以及镜面反射的权。可以看出，DirectX对材质的定义都是围绕光照的。

```
typedef struct_D3DMATERIAL9{
    D3DCOLORVALUE Diffuse;      //漫反射
    D3DCOLORVALUE Ambient;     //环境光
    D3DCOLORVALUE Specular;    //镜面反射
    D3DCOLORVALUE Emissive;    //自发光
    floatPower;                //镜面反射的权，可以表现反射高光的锐度
}D3DMATERIAL9;
```

但是，在现实世界中，物体的材质还包括透明度、纹理等。因此，为了更贴近现实，也为了统一一些管道有关操作，引擎需要对材质重新定义。

在CAP引擎中，材质类对应类GEMaterial，它除了包含DirectX中一般意义的材质，还包含了以下特性。

（1）纹理层（可以有多层）。

（2）各层纹理的混合方式。

（3）背景混合方式。

（4）深度缓存设置。

（5）裁减方式。

（6）纹理过滤形式。

（7）Shading方式。

（8）其他效果：雾化、Shader的支持、阴影选项等。

但为什么要对材质进行管理呢？因为设置渲染状态是十分耗时的，而这些关于渲染的设置在底层图形库（DirectX）中都是作为状态管理的，即一旦设置就保持这个设置，直至下次设置成新的状态。渲染状态繁多，而不同模型之间不同的渲染状态常常很少，如果能避免设置那些相同的渲染状态、只设置需要改变的状态的话将很大地提高程序运行时的FPS。

所以，可以在引擎的材质管理模块内保存当前所有的渲染状态，在设置新的材质对象时，每个选项都将与当前状态比较，如果相同就忽略，避免重复的设置。更具体地说，在类GEMaterial中有一个静态成员GEMaterial GE-Material::g_LastMaterial；它保存了上一次的

所有渲染状态，当你要应用一个新的材质的时候，它会将新的材质和上次的材质比较，然后仅修改不同的部分。

不过，如何保证每次的修改量也是最小呢？这就需要渲染队列来帮忙了。在每次渲染前，都需要遍历一遍场景图，对应需要绘制的节点按照其材质分类，最后分类绘制渲染队列。这样就保证了两个相邻绘制的物体的材质改变是最小的。

有了材质管理的概念后，还需要将材质脚本化，因为脚本化可以使材质改变时不需要重新编译程序，而且脚本化也可以为材质编辑的开发做好前期准备工作。

材质的脚本语言非常简单明了，类似自然语言，但又能表现逻辑性，如例程4-2所示。

**例程4-2　材质脚本**

```
Material
{
    //属性名              属性值
    AmbientColor          0.3    0.3    0.3      //环境光参数
    DiffuseColor          1  1  1                //漫反射参数
    SpecularColor         1  1  1                //镜面反射
    EmissiveColor         0  0  0                //自发光
    Power                 3                      //镜面反射权值
    DepthCheck            true                   //深度测试是否打开
    DepthW rite           true                   //深度缓冲是否可写
    Lighting              true                   //是否接受光照
    IsTransparent         false                  //是否透明
    UseAlphaBlendFactor false                     //是否使用Alpha测试
参数
    AlphaTestEnabled      false                  //是否打开Alpha测试
```

```
    CullingMode         CounterClockwise      //裁减模式
    FilteringMode       linear                //过滤模式
    //纹理层，可以有多层纹理
    Texture
    {
        File            Texture/skybox_X_.JPG //纹理层的纹理文件
        ColorOP         add                   //纹理层的混合方式
    }
}
```

脚本使用关键词解析。在引擎中，类GEMaterialLoader负责材质脚本的解析。需要首先调用类的静态函数bool GEMaterialLoader::Load（const string&FileName），接着Loader就开始读入*.mtl文件并解析脚本以载入。解析时，首先判断脚本起始的关键词声明是否为“Material”以判断载入文件类型是否有错。通过检测后，Loader将顺序读取每一行起始的关键字，将字符串转换成小写字符后根据匹配来判断该行为设置哪个参数，如无匹配则抛出错误信息。如若匹配，随即读到的关键字后的值则是要设置的参数的值。

此外，要注意脚本中对纹理层的解析略有不同。在读入纹理的文件名后，系统将先由纹理管理器创建相应的纹理，并将其挂入资源列表中，然后再将文件名存入GEMaterial类中，随后读入的ColorOp也将设置到纹理层的相应属性中去。

当然，材质也是可以动态创建的，没有脚本也可以。比如，当需要创建一个适合GUI层的材质，它可以如下实现：

```
{
    GEMaterial temp=GEMaterial();              //默认参数的材质
    temp.SetLighting(FALSE);                   //无需光照
    temp.SetAlphaTest(TRUE);                   //打开Alpha测试
```

```
    temp.SetAlphaTestFunc(D3DCMP_GREATER);    //Alpha大于等于的才画
    temp.SetAlphaReference(0);                  //Alpha测试的基数是0
    temp.AddTextureLayer();                      //增加一个纹理层，置空纹理
    temp.SetTextureColorOP(D3DTOP_MODULATE); //纹理层的颜色混合
    temp.SetTextureAlphaOP(D3DTOP_MODULATE); //纹理层的Alpha混合
    temp.SetTransparent(true);                  //支持透明
    temp.SetSceneBlend(SBT_ALPHA_BLEND);      //与底面的混合方式
    //你可以将temp返回，这就是你要的材质
}
```

也就是说，在设计引擎时，需要给出手动设置材质的方法。

# 4.3 顶点缓冲区和索引缓冲区

## 4.3.1 顶点缓冲区

在计算机中所描绘的3D图形世界中，任何物体都由点、线和多边形组成，而所有的多边形都可以由三角形逼近。我们知道：线由两个点构成，三角形由三个点构成，所以点——我们这里所说的计算机图形学中的顶点（ver-tex），是3D世界中的基本元素。

在所有的绘制工作开始前，必须对顶点的构成有一定了解。

顶点在Direct3D中有以下属性：

- 位置：顶点的位置，可以分别指定x，y，x三个值，也可以使用D3DXVECTOR3结构来定义。

- RHW：齐次坐标W的倒数（Reciprocal of the Homogenous W）。如果顶点为变换顶点的话，就要有这个值。设置这个值意味着你所定义的顶点将不需要Direct3D的辅助（不能作变换、旋转、放大缩小、光照等），要求你自己对顶点数据进行处理。
- 混合加权：顶点的混合加权。
- 顶点法线：经过顶点且和由顶点引出的边相垂直的线，即和顶点对应的三角形面垂直。法线用三个分量来描述它的方向，这个属性用于光照计算。
- 顶点大小：设定顶点的大小，这样顶点就可以不限于只占 一个像素了。
- 漫反射色：即光线照射到物体上产生漫反射的着色。
- 镜面反射色：也就是高光，它可以让物体看起来更逼真，且能够表现不同的质感。
- 纹理坐标：在多边形上覆盖纹理时，需要使点正确地对应纹理坐标。纹理坐标可以是一维的也可以是二维、三维或是四维的。纹理坐标是相对坐标，取值范围是0~1，左上角是（0,0），右下角是（1,1）。

当然，不是说顶点都要包含所有的属性，而是需要根据实际情况选择不同的属性；但是，顶点属性声明的顺序是不能改变的，必须按照上面的顺序声明。

### 例程4-3　顶点的定义

```
structTDUVVertex
{
    float     x, y, z, rhw;     //顶点位置
    D3DCOLOR diffuse;           //漫反射颜色
```

```
    float    tu, tv;          //纹理坐标
    staticDWORD FVF;         //顶点的格式
}
DWORD TDUVVertex::FVF=D3DFVF_XYZRHW| D3DFVF_DIFFUSE| D3DFVF_TEX1;
```

例程4-3是一个经过变换的、有漫反射和纹理贴图的顶点。顶点结构体的最后一个静态成员FVF是灵活顶点格式（Flexible Vertex Format），表示顶点的格式。在定义完一个顶点后，必须要同时定义它的顶点格式。通常情况下，习惯用#define来定义FVF，比如#define FVF=D3DFVF_XYZRHW。不过，用结构体内的静态数据成员对代码管理更有帮助，尤其是一个应用中有许多种不同的顶点格式的时候。如此，在使用的时候就非常简单明了：TDUVVertex::FVF。

在定义了顶点后，如果要显示图元还必须将顶点对应的图元绘制出来。这里就牵涉到顶点传输方式问题了。最简单的方式，就是将顶点在绘制的时候直接从内存里往图形显示卡里传，这就是直接绘制顶点的方式；第二种就是将顶点缓存在AGP中，然后绘制；第三种就是使用索引缓存器，这样可以用更少的容量存储顶点。

所谓顶点缓存区就是一块存储顶点的内存缓冲区。它可以存储各种类型的顶点，无论是变换的还是无需变换的，无光照的或是有光照的；同时，可以对顶点缓冲区中的顶点进行操作，比如几何变换、应用光照或是裁减。顶点缓存区的好处是可以重用几何变换过的顶点，比如当创建一块顶点缓冲区，对其中的顶点进行变换、光照、裁减后，可以渲染它很多次，但却不需要重新传输顶点数据。

顶点缓冲区在DirectX中被定义成接口IDirect3DVertexBuffer9，使用起来参数很多，也比较复杂。为了使游戏程序员能够更加专注与

游戏的编写，必须要对接口IDirect3DVertexBuffer9进行封装。于是，定义了一个新类GEVertexBuffer，它有一个数据成员如下：

```
IDirect3DVertexBuffer9*GEVertexBuffer::m_DX9VB [protected]
```

也就是说，可用接口类IDirect3DVertexBuffer9来重新实现我们的新类。在这个类中，要处理顶点缓冲区的创建、顶点数据的填充以及设备丢失后的操作。其中，属顶点缓冲区的创建形式的封装最为重要，因为不同的创建形式对应了不同的填充形式和设备丢失后的操作。

顶点缓冲区有以下几种创建方式。

（1）以Managed方式创建

这是最常用的方式。数据存放与显存中，同时自动在系统内存中备份顶点、处理设备丢失时的顶点数据填充。

（2）以Default方式创建

这种方式将不在内存中保存数据备份，需要在设备丢失时重新从硬盘中装载数据。当顶点数据很大时，在系统内存中备份数据将占 用大量空间，而设备丢失并不经常发生，所以这种情况下可以以Default方式创建缓冲区，节省内存空间。但是这种做法的缺点是设备丢失时需要重新读取硬盘、解析顶点数据，用户停顿感比较明显，不过设备丢失在游戏运行过程中并不频繁。

（3）附加Dynamic方式创建

当缓冲区需要频繁锁定时（比如骨骼动画），以Dynamic方式创建将极大地提高程序运行效率。如果缓冲区以Dynamic方式创建，有以下两种情况。

①重新填充缓冲区中的数据：以Discard方式锁定缓冲区，这时，如果显卡正在读取该缓存，DirectX将立刻从锁定函数返回，并指向一块新的缓冲区，原来的缓冲区将在显卡读取完毕后释放掉。这样，程序就不用在锁定时挂起直至显卡读取完数据。

②只添加新的数据：以NoOverWrite方式锁定缓冲区。这时，如果显卡正在读取该缓存，DirectX将立刻从锁定函数返回，并指向该缓冲区的尾部，所以程序可以添加新的数据在缓冲区末尾，而不影响显卡读取缓存中前面的数据。

缓冲区创建，按照需求定义GEVertexBuffer的构造函数，见例程4-4。

**例程4-4　顶点缓冲区类的构造函数**

```
/**
*构造函数
*@param size          要创建的缓存大小
*@param FVF          缓冲区的顶点格式
*@param stride          缓存内顶点的步长
*@param primitiveType   图元类型（三角形条带、线段列表等）
*@param managed      以managed方式创建，由DirectX底层自动管理设备丢失，
*                    无需引擎在系统内存中备份数据
*@param dynamic      如果缓存需要被频繁更新，且更新时原来的数据全部丢弃或
*                    只在原数据后面添加，则使用此标志将是操作效率更高
```

```
*@param autoBackupForRestore如果以非managed方式创建缓冲区，选择此标志可以
*                   使引擎在系统内存自动备份数据用于设备丢失的处理；如果是非managed方式，而且未选择此标志，则需要在处理设备丢失时提供数据源
*/
GEVertexBuffer::GEVertexBuffer(UINT size,
                              DWORD FVF,
                              UINT stride,
                              GEPrimitiveType primitiveType,
                              boolmanaged=true,
                              booldynamic=false,
                              boolautoBackupForRestore=true);
```

大部分情况下，客户程序员应该让GEVertexBuffer以managed方式创建，但两种情况下客户程序员可能不希望以managed方式创建，因为：

（1）该顶点缓冲区用于骨骼动画，要频繁更新缓冲区数据，这种情况下非managed效率较高（内部使用D3DPOOL_DEFAULT）。

（2）模型很大，不希望在内存中备份而占 用过多内存（代价是设备恢复时要从硬盘读入数据）。

在这两种情况下，客户程序员需要在构造函数里指明managed为false。

如果客户程序员选择让GEVertexBuffer在系统内存中备份顶点数据、自

动处理设备恢复，那么，在新建一个GEVertexBuffer对象之后，必须调用它的函数PrepareSysBuffer()；反之，如果客户程序员出于某些原因选择让GE-VertexBuffer不要在系统内存中备份数据（如：因为该顶点缓冲区很大，在系统内存中备份将浪费极大空间），就无需调用PrepareSysBuffer()，但在处理设备恢复时需要提供外来的数据源。如果顶点缓冲区需要大量更新并且更新方式是完全丢弃原数据或只添加新数据，就必须在构造函数中指定dyna mic为true。不过也要注意到，并非所有需要频繁更新的顶点缓冲区都可以以dynamic方式创建。

除了封装缓冲区的创建方式外，类GEVertexBuffer还需要处理在设备丢失时以及程序销毁时处理顶点的填充及释放。在处理设备丢失和设备初始化时还需要特别注意缓冲区不同构造方式对应的不同处理方式。

此外，还需要提供一些帮助函数，比如提供顶点个数和绘制方式自动得到图元数量，算法如下：

- 如果是点序列（POINT LIST）：图元数=顶点数
- 如果是线序列（LINE LIST）：图元数=顶点数÷2
- 如果是线条带（LINE STRIP）：图元数=顶点数－1
- 如果是三角形序列（TRIANGLE LIST）：图元数=顶点数÷3
- 如果是三角形条带（TRIANGLE STRIP）：图元数=顶点数－2
- 如果是三角扇形（TRIANGLE FAN）：图元数=顶点数－2

顶点缓冲区可以提高程序的效率，不过只有顶点缓冲区还是不够的，尤其是面对复杂模型的时候。比如，在图4-1中，有6个顶点构成4个三角形，只能用三角形序列的方式来构造这四个三角形（方法A）。

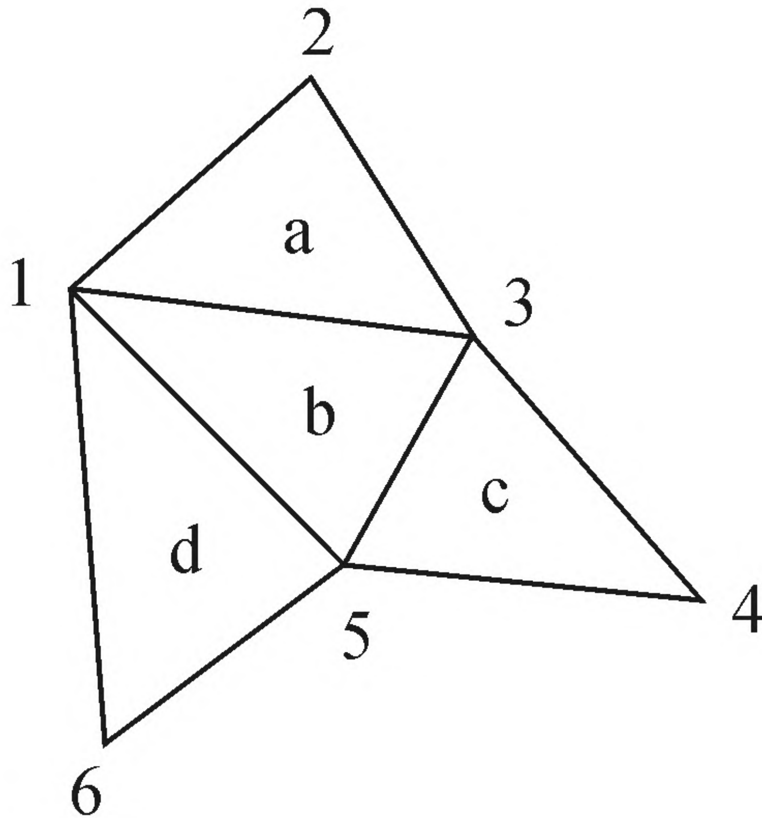



图4-1　复杂图形示例

方法A：

顶点缓冲区：顶点1，顶点2，顶点3

顶点1，顶点5，顶点3

顶点3，顶点4，顶点5

顶点1，顶点5，顶点6

顶点缓冲区大小：12

绘制方法：三角形序列（TRIANGLE LIST）

结果发现，顶点缓冲区内的顶点数量是实际顶点数量的2倍，如果图形再复杂一些的话倍数就更多了。由于单个顶点占 用内存比较多，所以，这是非常消耗内存空间的。于是，索引缓冲区的概念就出来了。

## 4.3.2　索引缓冲区

索引缓冲区中保存的是顶点缓冲区中顶点的索引号，它的功能和指针很像，可以直接调用顶点缓冲区的对应顶点。有了索引缓冲区，顶点缓冲区中的顶点就不必要重复存储了，这大大提高了存储效率。方法B中描述了如何使用索引缓冲区完成与方法A一样的工作。

方法B：

顶点缓冲区：顶点1，顶点2，顶点3，顶点4，顶点5，顶点6

索引缓冲区：1，2，3

1，5，3

3，4，5

1，5，6

顶点缓冲区大小：6

绘制方法：三角形序列（TRIANGLE LIST）

在DirectX中，索引缓冲区对应接口类IDirect3DIndexBuffer9。它的很多概念和顶点缓冲区非常相似，比如创建方式、处理设备丢失

等，对于这些概念，这里就不作介绍了，基本和顶点缓冲区一样。同时，因为DirectX中的接口类IDirect3DIndexBuffer9的操作相对复杂、参数繁多，所以也有必要对它进行二次封装。在随书引擎中，索引缓冲区对应类GEIndexBuffer。

因为一个索引缓冲区必定对应一个顶点缓冲区，而一个顶点缓冲区可以对应多个索引缓冲区，所以在类GEIndexBuffer中保存一个GEVertexBuffer的指针。此外，互相对应的顶点缓冲区和索引缓冲区的创建形式必定是一样的。所以，当顶点缓冲区只对应一个索引缓冲区时，也可以直接创建顶点缓冲区并一起生成一个对应的顶点缓冲区。

总之，在有可能的情况下，应当一直使用顶点缓冲区和索引缓冲区，而不是由应用程序分配的普通内存块。

## 4.4　静态模型

静态模型，也就是通常所说的mesh，是一个游戏中最重要的部件之一。一个游戏中必定有许许多多的模型。游戏中的人是模型、建筑是模型、车辆是模型……基本上所见到的都是模型。与此同时，模型一般都是些复杂图形，程序员是不可能自己编程构造顶点去设计一个模型的。基本上所有游戏中的模型都是从模型脚本中得到的，因此，模型必须实现脚本化。

一般来说，模型的设计和建模都是美工完成的，而美工使用的建模软件通常不是3D Studio Max，就是Maya。虽然这些建模软件已经将模型脚本化了，但是这些脚本，无论是.3DS还是.max都包含了太多信息，并且都没有为游戏引擎这类实时应用作优化。而游戏需要使用更

快、更简单的文件格式。所以，必须在这两种工具上（至少是一个工具上）提供导出插件，使美工能够使用自己熟悉的建模软件建模且直接导出我们需要的模型格式。

但是，导出插件的设计和实现是相当复杂的一件事，尤其是所设计的插件需要匹配建模软件的不同版本，这带来了极大的工作量。如果是一个初学者，或者想把精力放在那些真正值得去关心的事情上（比如模型的数据结构设计），那么最好使用一个第三方的导出插件和第三方的模型脚本格式。

模型的脚本格式很多，下面简单介绍几种常见的格式。

第一种就是微软的X文件格式。X文件格式是Direct3X定义的，它是由模板（template）驱动的，模板定义了如何存储一个数据对象，这样用户便可以自己定义具体的格式，DirectX的SDK中也预定义了一些模板。此外，对于模型的层次关系的处理上，模板使用“可选成员（optional member）”作为数据对象的子对象来保存。子对象可以是另一种数据对象或是对一个先前数据对象的引用或是一个二进制的对象。

由于X文件的细节是全公开的，所以支持的第三方插件很多，比如3DS Max上的Panda导出插件，而且DXSDK本身就带3DS Max和Maya的导出插件。此外，可以通过DXSDK自带的Mesh Viewer观察模型文件是否正确。

此外还可以用Quake3（雷神之锤3）引擎的md3文件。md3格式是一种常用的文件格式，并且经历过广泛的测试，在Quake3和许多其他游

戏中，md3被证明为一种极好的模型格式。md3格式支持LOD、SHADER、可切换纹理等等高级特征。

对于不同的模型，CAP引擎使用了不同的解决方案。如果模型是附带骨骼的，则使用的是开源引擎OGRE的模型脚本以及它的模型导出插件（如图4-2）。用这个插件可以从3DS Max中将模型导出成XML脚本。在安装了这个插件后，3DS Max中就会出现对应的工具栏，美工就可以非常方便地将模型转变成XML脚本。而针对没有骨骼信息的模型，我们开发了在MAX8上运行非常良好的导出插件，它导出的文件更小，且效率更高。

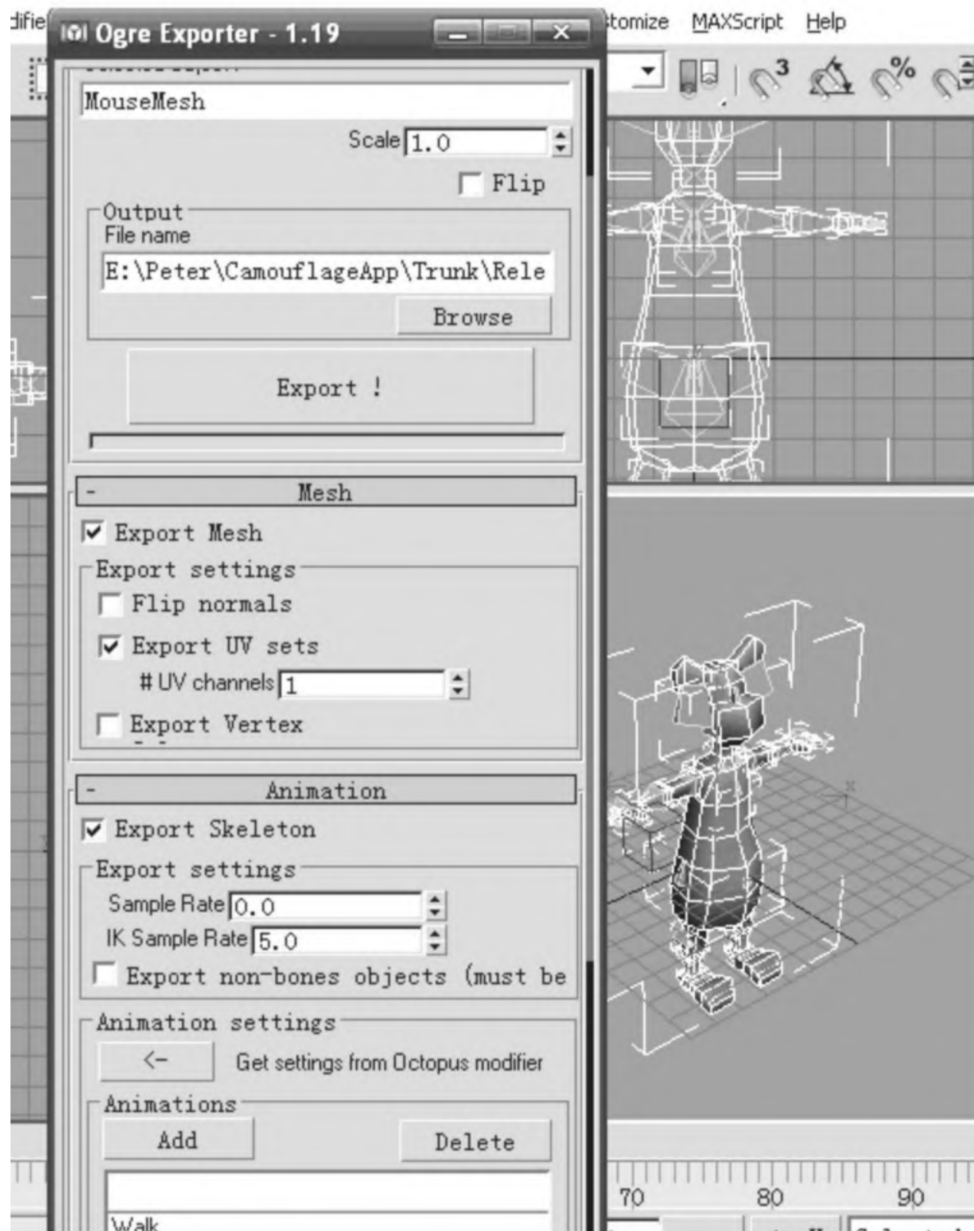



图4-2　OGRE的3DS Max5导出插件

有了脚本后，就可以解析脚本，并填充模型的数据结构了。

作为一个静态模型，就是一群以一定顺序排列的顶点的集合。但是，对于游戏来说，效率和可扩展性是要保证的，所以我们在模型（Mesh）和顶点之间又增加了一个子模型层（Submesh），即Mesh按材

质的不同分成SubMesh，每个SubMesh使用相同材质，一个Mesh的所有SubMesh共用顶点缓冲区，每个SubMesh有自己的索引缓冲区和材质。大致的关系如图4-3所示。

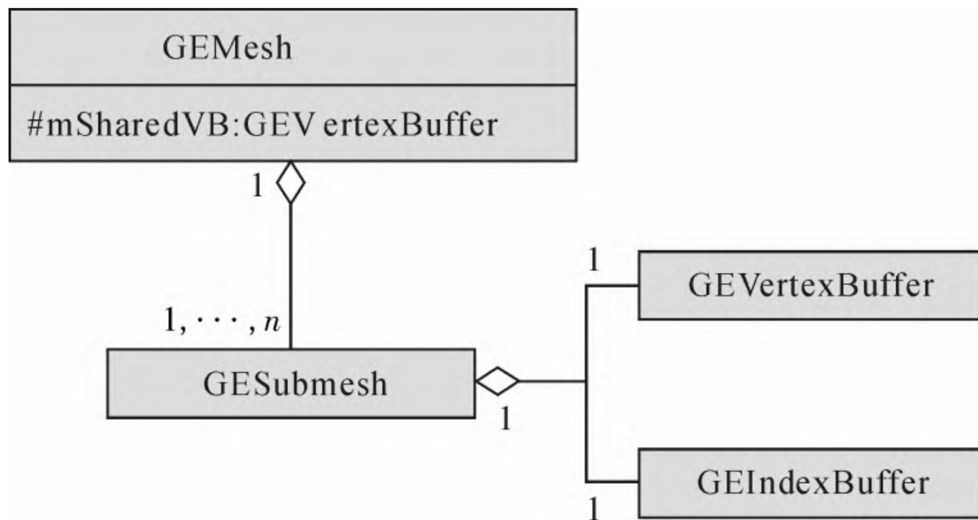



图4-3　静态模型类图

在图中可以看出，GEMesh类有一个公共顶点缓冲区，这个顶点缓冲区可以分享给模型的所有Submesh使用，每一个Submesh都有自己的材质，也就是说，每一个Submesh都可以有自己的纹理。比方说，有一个人的全身模型，他有穿衣服和裤子，在游戏中，他的衣服和裤子可能会更换，但衣服和裤子不总是同时更换。在这个应用中，可以将人的模型顶点数据放在GEMesh中的公共顶点缓冲区中，然后上半身和下半身是两个不同的Submesh，它们的索引缓冲区对应相应的顶点，并且这两个Submesh对应各自的纹理文件。于是，当需要更换人物的衣服时，只需要更改上半身的Submesh的纹理文件就可以了，而不需要更改下半身的任何数据。

那既然有了Mesh中的公共缓冲区，Submesh又为什么要有自己的顶点缓冲区呢？这是因为，一个复杂模型可能由许多的部分组成，但每个部分的顶点格式可能不同。假设有一个壶，它的壶身是紫砂的，它的盖子是透明玻璃的。为了考虑效率，紫砂的壶身我们不考虑镜面反射，而对透明玻璃盖我们考虑镜面反射，于是，壶身和壶盖的顶点数据结构就不同了，也就是说顶点的格式不同了，为此，必须将壶盖和壶身分属不同的Submesh。

在定义好模型的数据结构后，还需要解决模型加载的问题。由于本节讲述的是静态模型，所以将不会涉及骨骼动画使用的骨架结构。

首先，模型的存储使用XML文件或是2进制文件，XML文件格式是用于调试的，而二进制文件则是真正用于游戏的。以下所有的讲解都会围绕XML格式展开。

**例程4-5　XML模型文件**

```
<mesh>
    <geometry vertexcount="n">
        <vertexbufferpositions=".."normals="..">
            <vertex>
                <position...>
                <normal...>
            </vertex>
            …
        </vertexbuffer>
                                                            <vertexbuffer
texture_coord_dimensions_0=".."texture_coords=
"..">
            <vertex>
                <texcoord u=".."v=".."/>
```

```
            </vertex>
        </vertexbuffer>
    </geometry>
    <submeshes>

<submeshmaterial=".."usesharedvertices=".."use32bitindexes=".."
operation-type="..">
            <faces count="n">
                <face v1="0"v2="1"v3="2"/>
            </faces>
            <geometry vertexcount="n">
                <vertexbufferpositions=".."normals="..">
                    <vertex>
                        <position>
                        <normal>
                    </vertex>
                    </vertexbuffer>
                <vertexbuffer texture_coord_dimensions_0=".."

texture_coords="..">
                    <vertex>
                        <texcoord u=".."v=".."/>
                      </vertex>
                    …
                </vertexbuffer>
            </geometry>
        </submesh>
    </submeshes>
</mesh>
```

例程4-5中展示了Mesh的XML格式。简单概况就是先定义Mesh的公共顶点区顶点数据，接着定义Submesh的顶点，之后定义Submesh的材质，最后定义Submesh中组成面的顶点索引。

在解析模型脚本的时候按照关键字解析，但注意大的顺序是不能变的，比如mesh的定义必须在Submesh之前。

最后，要实现的就是模型显示了，这一步在有了前面的基础后就水到渠成了。在一个模型显示时，首先加载它的共享顶点缓冲区部分，然后逐次遍历所有的Submesh，如果有Submesh使用的共享顶点缓冲区，则将它绘制出来，在所有使用共享顶点缓冲区的Submesh绘制结束后再绘制使用独立顶点缓冲区的Submesh。

至此，静态模型的数据结构、解析和显示都完成了。

# 第5章　骨骼动画技术的实现

3D角色动画是计算机动画技术的一个重要组成部分。无论是在离线渲染环境下，还是在实时渲染环境下，3D角色动画都得到了广泛的应用。

在离线渲染环境下，主要应用于动画电影制作和各类广告制作。动画电影制作中所使用的3D角色动画技术的一个重要特点是动画数据量大，渲染需要耗费大量时间，因此动画作品必须预先制作，渲染，然后转化成视频文件播放。

在实时渲染环境下，主要应用于虚拟现实和视频游戏，甚至是建模软件、动画制作软件。现在，随着计算机硬件技术的发展，特别是带有硬件加速功能的显卡性能的提高，很多曾经只能在离线环境下应用的技术，都转移到实时渲染环境中来。其中，实时渲染的角色动画技术得到了发展且被广泛的应用。目前，实时角色动画技术大体可分为三种类型。

第一类是骨骼动画（Skeletal Animation）。骨骼动画中的角色由若干独立的部分组成。每一个部分对应着一个独立的网格模型，不同的部分按照角色的特点组织成一个层次结构。比如说，一个人体模型可以由头，上身，左上臂，左前臂，左手，右上臂，右前臂，右手，左大腿，左小腿，左脚，右大腿，右小腿，右脚等各部分组成。而某个部分，可能是另一个部分的子节点，同时又是另一个部分的父节点。比如人体模型中，右前臂就是右上臂的子节点，同时也是右手的父节点。而右上臂是上身的子节点，后者则是躯体的子节点。通过

改变不同部分之间的相对位置，比如夹角，位移等等，就可以实现所需要的各种动画效果。这类动画的优点很多。首先，在动画序列的关键帧中只需要存储节点间的相对变化，因此动画文件占用的空间很小。其次，可以实现很多复杂的动画效果，如果应用程序支持反向动力学还可以动态实现预先存储的动画序列之外的新的动画效果。当然这类动画也有不少缺点。其中之一是由于角色模型是一个层次模型，要获得某一个部分相对于世界坐标的位置，必须从根节点开始遍历该节点所有的祖先节点累计计算模型的世界变换。但最关键的问题是在不同肢体部位的结合处往往会有很明显的接缝，这会严重地影响模型的真实感。

第二类是变形体动画（Morphing Animation）。这种动画中的角色由一系列的渐变网格模型构成。在动画序列的关键帧中记录着组成网格的各个顶点的新位置或者是相对于原位置的改变量。通过在相邻关键帧之间插值来直接改变该网格模型中各个顶点的位置就可以实现动画效果。简单地说，关键帧动画其实就是模型特定姿态的一个“快照”。通过在帧之间插值的方法，可以得到平滑的动画效果。相对于关节动画，单一网格模型动画的角色看上去更真实，也不会有关节动画所面临的接缝问题。由于没有使用层次模型，获得模型网格顶点在世界坐标中位置的计算量也很小。但是，这类动画的适应性很弱，角色很难通过实时计算来与环境进行良好的互动，以获得预先存储的动画序列之外的动画效果。另一方面，由于关键帧要存储网格模型所有的顶点信息，动画文件占用的空间比较大。

第三类是蒙皮骨骼动画（Skinned Skeletal Animation）。蒙皮骨骼动画可以看做是关节动画和渐变动画的结合，如图5-1所示。模型网格的顶点可以被几块骨头一起影响，每块骨头对该顶点的权重不

同，所有骨头对该顶点的作用之和就是该顶点最终的效果。蒙皮骨骼动画同时兼有关节动画的灵活和渐变动画的逼真，也是本章介绍的主要内容。

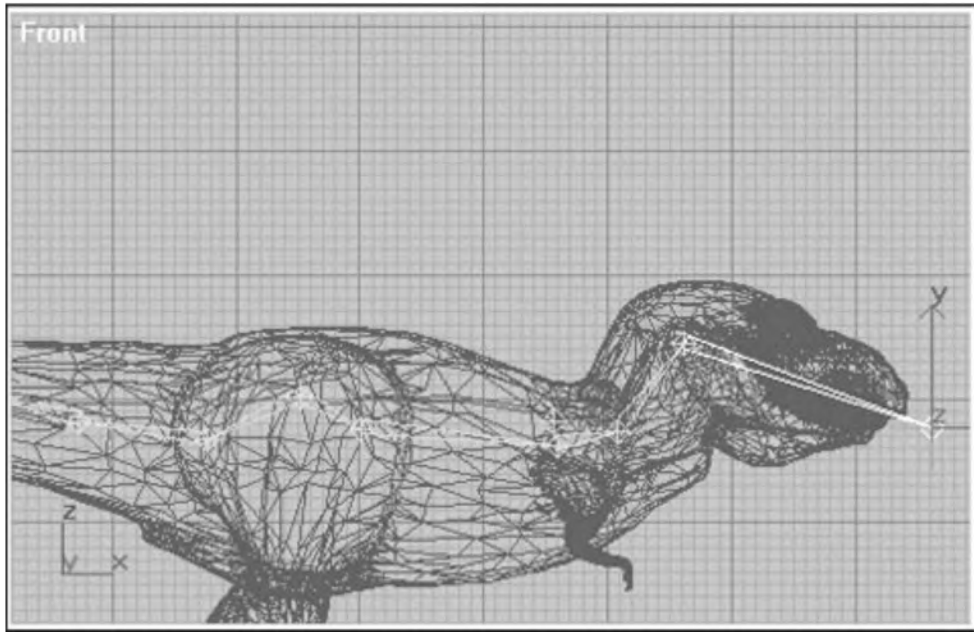



图5-1　蒙皮骨骼动画的制作过程

下面将首先介绍如何定义蒙皮骨骼动画的数据格式，接着会介绍如何更新骨骼动画。

# 5.1　动作数据格式解析

骨骼动画中有几个数据结构需要定义，分别是骨架、蒙皮骨架和关键帧骨骼动画。本节会依次带你实现自己的骨架动画结构，记住，不是.X文件哦！是你自己的格式！

## 5.1.1　骨架

骨骼动画中最关键的部分是骨架，骨架构成了物体的层次结构。所以在实现骨骼动画时，我们需要首先定义骨架的数据结构。

如图5-2所示，骨架就是一系列互相连接的骨骼的集合。在这骨骼集合中，有唯一的根骨骼点构成骨架的中心，别的骨骼都是作为子孙骨骼附着在根骨骼上。

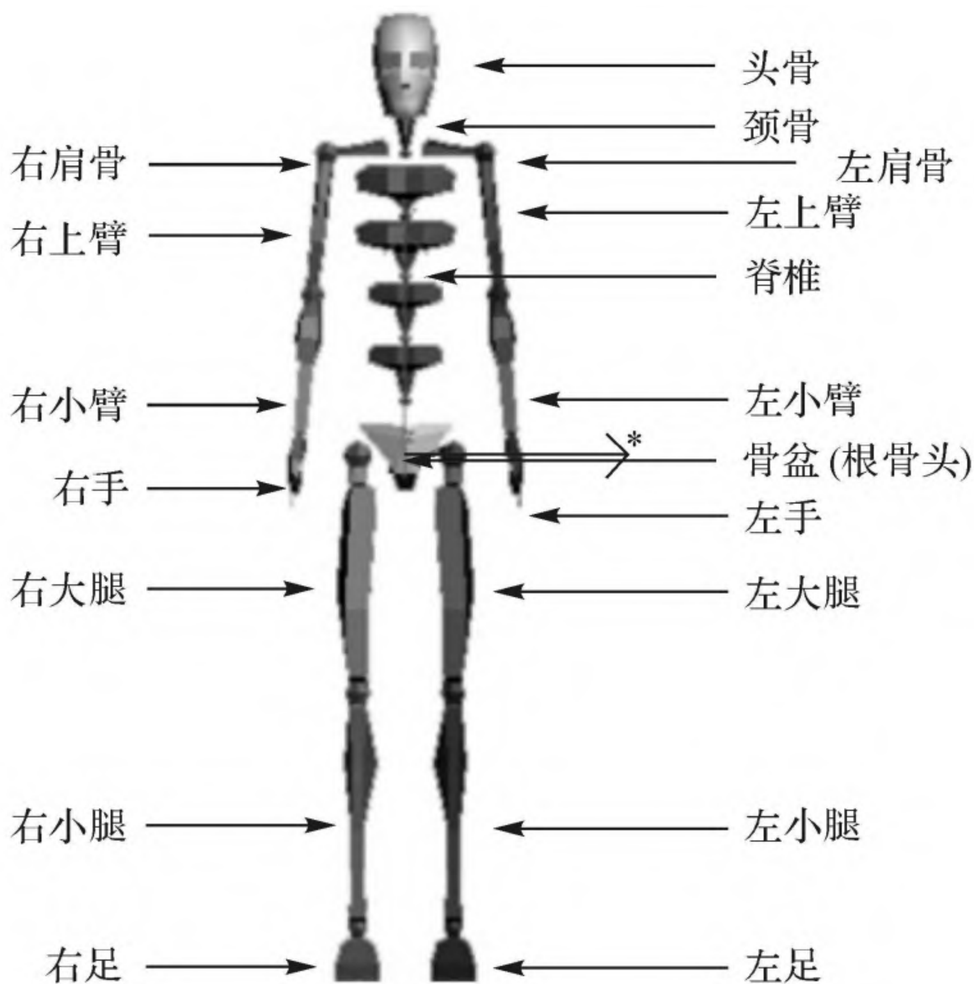



图5-2　骨架结构示意

在有了这个骨架后，就可以从中得到骨架的层次模型，如图5-2，5-3所示。

图5-3　骨架的层次结构

是不是有点像场景管理中的场景图？是的，它们都是以树为基础的层次关系，它们的更新方式也非常类似。甚至，我们可以把骨骼就理解成场景中的一个节点，父节点可以带动子节点运动。想想，当你抬起大腿的同时，小腿是不是也一起被抬起了？

在了解了骨架结构后，就可以开始定义骨架和骨骼的数据结构了。

首先定义骨架类：Skeleton

```
classSkeleton{
private:
    vector<Bone*>mRoots;                //持有所有根节点的容器
    BoneIndexMapmBones;                 //所有骨骼及其索引号
    BoneNameMapmBonesByName;            //所有骨骼及其名字
}
```

其中，BoneIndexMap和BoneNameMap都是typedef，如下所示：

```
typedefmap<unsigned short，Bone*>BoneIndexMap;
typedefmap<string, Bone*>BoneNameMap;
```

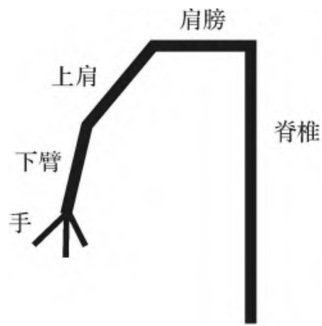
不难看出，骨架其实就是一系列骨骼的集合。其次，还会定义两个Map，分别存放骨骼的序号索引和名字索引，方便查找。简单地说，

骨架就像汽车，而骨骼就是汽车上的零件。当然，在全面了解汽车的功能前，还需要先搞清楚各个零件的作用。

首先，同一骨架中的每个骨骼都有一个唯一的编号和唯一的名字，通过这唯一的编号或名字得到相应的骨骼节点。同时，一块骨骼只属于一个骨架，通过保存骨骼所属骨架的指针来加速骨骼查找过程。

其次，因为骨骼是通过父子节点联系的，所以每个骨骼都需要存储对应的父节点和子节点。于是，在骨骼对象中，我们存储了父节点的索引以及子节点的索引数组。这样，就可以通过根骨骼节点得到骨架的层次关系。在这里，需要补充的是：为了提高骨架更新的效率，从骨架描述文件读入骨架信息并建立骨架后我们还需要为每块骨骼保存父节点指针和子节点指针数组。那么，为什么不在一开始就保存这些指针呢？这是因为在读入骨架文件的时候，骨骼的顺序是未知的。所以，在所有的骨骼读取完成后，需要通过索引号建立起该骨骼与父子骨节点的指针。

最后，每块骨骼都需要与其子节点和父节点建立联系。我们的解决方案是在骨骼对象中存放相对于父节点的平移、旋转和缩放，通过遍历所有的父节点得到该子节点的绝对平移、旋转和缩放。如图5-4所示。




| 骨头 | 绝对变换 |
| --- | --- |
| 脊椎 | M脊椎 |
| 肩膀 | M肩膀×M脊椎 |
| 上臂 | M上臂×M肩膀×M脊椎 |
| 下臂 | M下臂×M上臂×M肩膀×M脊椎 |
| 手 | M手×M下臂×M上臂×M肩膀×M脊椎 |

注：M是组合变换矩阵，由R（旋转）、T（平移）、S（缩放）这三个矩阵级乘得到

图5-4　骨骼绝对变换的获得

这个过程和第三章中介绍的场景图的更新完全一样，大家可以从第三章中得到更为详细的过程说明。

功能需求定义完后，就需要定义数据成员了。类设计中，数据成员的定义永远是最重要的。骨骼类Bone的数据成员可以参考例程5-1。

其中大部分成员都已经讲解了它们的作用，不过还需要注意类成员中的初始变换逆矩阵mInverseInitialTransform。因为在骨骼动画中，所有的动画都是使用相对变换的，所以需要先把骨骼放到世界中心（pivot），然后作用变换。

### 例程5-1　Bone类的数据成员

```
classBone
{
private:
    unsigned shortmIndex;                          //骨头的索引号
    stringmName;                                   //骨头名
    Skeleton*mSkeleton;                            //骨头所属的骨架
    intmParentIndex;                               //父节点的索引号
    vector<unsigned short>mChildrenIndex; //所有子节点的索引号
    Vector3    mInitialScale;                      //初始变换中的放缩
```

```
    Vector3    mInitialTranslation;                   //初始变换中的平移
    QuaternionmInitialRotation;                       //初始变换中的旋转
    Vector3    mRelativeScale;                        //某时刻相对变换中的放缩
    Vector3    mRelativeTranslation;                  //某时刻相对变换中的平移
    QuaternionmRelativeRotation;                      //某时刻相对变换中的旋转
    Vector3    mAbsoluteScale;                        //某时刻计算所的绝对变换中的放缩
    Vector3    mAbsoluteTranslation;                  //某时刻计算所的绝对变换中的平移
    QuaternionmAbsoluteRotation;                      //某时刻计算所的绝对变换中的旋转
    Matrix4    mInverseInitialTransform;              //初始变换的逆变换
    bool    mFullTransformNeedUpdate;                 //最终变换矩阵显要更新的标志
    Matrix4    mFullTransform;                        //最终变换矩阵
}
```

了解了骨骼结构，就需要定义如何在文件中存储骨骼了。为了和Mesh的文件存储有很好的连贯性，可以采用xml的方式来存储骨架。文件的格式如例程5-2所示，其中，只截取了描述骨骼动画文件中关于骨架定义的部分。文件中，先定义了每一块骨头的详细参数：索引、名称、初始平移和初始旋转；接着定义所有的骨头的父子关系。

**例程5-2　骨架的描述文件**

```
<skeleton>
    <bones>
        <bone id="0"name="Joint1">
            <position x="1.0"y="2.0"z="3.0"/>
            <rotation angle="30">
```

```
                <axis x="1"y="0"z="0">
            </rotation>
        </bone>
        …………
        …………
    </bones>
    <bonehierarchy>
        <boneparentbone="Joint2"parent="Joint1"/>
        <boneparentbone="Joint3"parent="Joint2"/>
        …………
        …………
    </bonehierarchy>
<skeleton>
```

### 5.1.2　蒙皮骨骼

只定义了骨骼的骨架还不能正常表达动作，还需要将顶点附着到骨骼上去。在最开始的关节动画中，所有的顶点都是只对应一个骨骼，这样，就造成了动画时不同部分的结合处往往会有明显的接缝。为了解决这个问题，可以让每个顶点受一个或多个关节（骨骼）影响，不同的关节对顶点影响的权不同，且所有关节对同一顶点影响的权之和为1。这就是蒙皮骨骼动画。

图5-5中比较了关节动画和蒙皮骨骼的效果差异。可以发现，中图的关节动画有明显的接缝，而右图的蒙皮骨骼动画则圆润得多。以顶点一为例，在蒙皮骨骼中，关节一和关节二对顶点一都有影响，假设两个关节对顶点一的影响是一样的，即关节一和关节二对顶点一的权是0.5，则在运动的时候顶点一的变形计算公式如下：

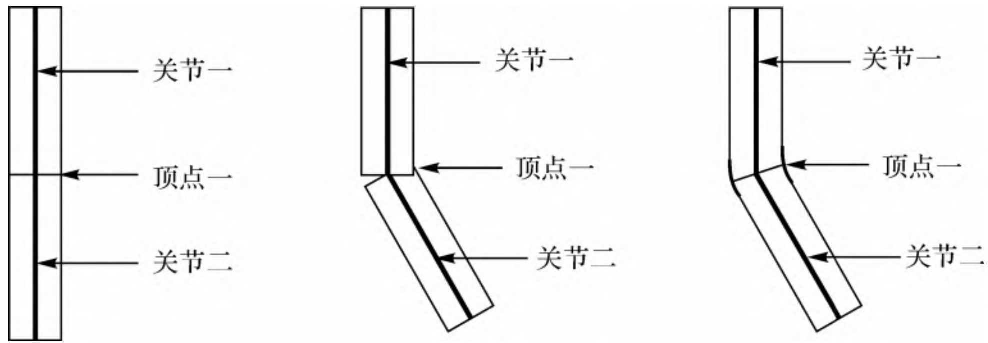



图5-5　关节动画（中）与蒙皮骨骼（右）比较

$$M_{顶点一}=\frac{1}{2}M_{关节一}+\frac{1}{2}M_{关节二}$$

在文件中，可以用顶点和关节索引作为关键词存储顶点和关节的权关系，格式如下所示：

[Vertex ID Bone ID Weight]

下面的脚本就是以如上的文本格式保存的[ 顶点－关节－权] 关系：

[0　17　0.3]

[0　12　0.7]

它表示0号顶点受到17号关节和12号关节的作用，其中17号关节的影响权值为0.3，12号关节的影响权值为0.7。

同理，对于上面的例子也可以用XML的格式来表示，这会更加通用：

```
<BoneAssignments>
                                          <VertexBoneAssignment
VertexIndex="0"BoneIndex="17"Weight="0.3"/>
                                          <VertexBoneAssignment
VertexIndex="0"BoneIndex="12"Weight="0.7"/>
    ……
    ……
</BoneAssignments>
```

## 5.1.3 关键帧骨骼动画

所谓动画就是通过连续播放一系列画面，给视觉造成连续变化的图画；关键帧动画就是由动画师设计动画中的关键画面，也即所谓的关键帧，然后由计算机根据一定的规则生成中间帧；而关键帧骨骼动画则是在关键帧中设计好骨骼的位置、朝向等信息，然后中间帧插值计算骨骼的位置、朝向。

关键帧技术是骨骼动画中的核心技术，设计的时候可以一层一层分开实现，如图5-6所示。

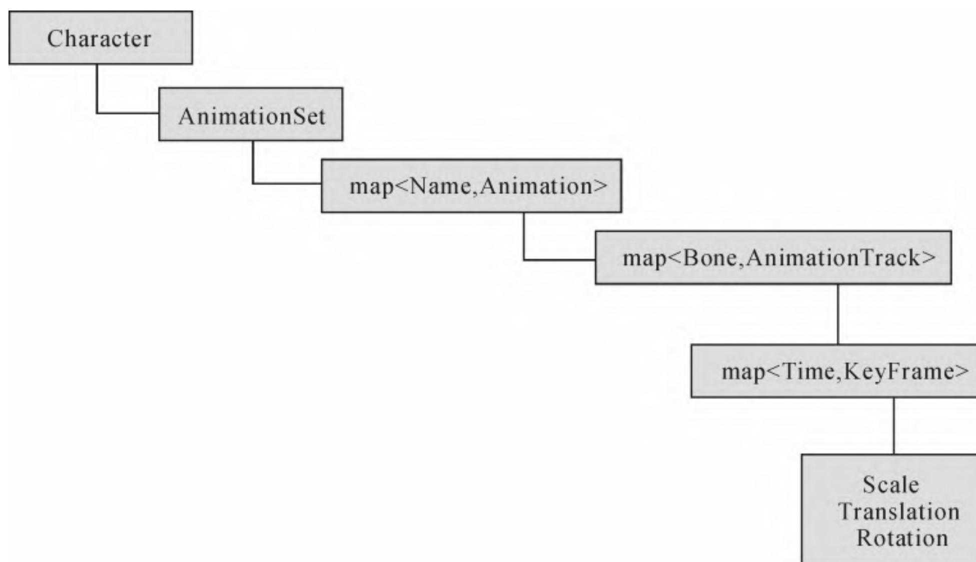



图5-6　骨骼动画实现的层次结构（自上而下）

一个角色可以有多个动作序列，比如一个足球运动员的角色，他可以有I-dle、Walk、Run、Shot、Jump、Celebrate等许多动作，每个动作都是一个独立的骨骼动画。于是，在角色类中可以包含一个数据成员——动画集（Animation-Set）。动画集是角色专属动画的集合，它可以被设计成一个以动画名为关键词，动画（Animation）作为数据的映射（map），然后通过动画名来调用相对的动画。

动画，这里应该叫做关键帧骨骼动画，是由一系列关键帧组成的。主要的数据包括动画的时长和动画序列构成。其中，时长就是动画的持续时间，动画序列则是关键帧的集合。以跳（Jump）的动画作为例：先是蹲下一点点，接着跳起，然后落下有个着地缓冲，最后恢复站立姿态。最简单的做法是抽象成四个关键帧（蹲下、起跳、着地、站立），每个关键帧都由若干关节的关键帧组成，比如蹲下的关键帧就是由大腿骨和小腿骨的动作组成。于是我们可以把Animation设计成时间和关键帧的映射map<Time，AnimationTrack>。这样比较符合人的思维模式，因为一个动作本身就是分成几个子动作组成的。但

是，这种设计也有不方便的地方，尤其是骨骼动画更新的时候。如果在更新骨骼动画的时候是以骨架、骨骼为单位的，那么我们需要每次查找是否该骨骼在该时刻需要更新，这会影响骨骼动画实现的效率。于是，可以采取略微变通的方式，因为在骨骼动画中，单个骨骼是最小的动作单位，当你需要让角色做一个动作时，就必须指定每个骨骼的动作序列，故以骨骼作为关键词，其和子关键帧组成映射关系map<Bone，AnimationTrack>。还是以跳这个动作作为例子，记录的就是大腿骨在初始时刻弯曲，接着伸直，最后再弯曲这么一个以单位骨骼为关键词的序列。

以骨骼为单位的子关键帧（AnimationTrack）是一个以时刻为关键词，由时刻和该时刻变换（KeyFrame）组成的映射map<Time，KeyFrame>。其中，变换包括了关节在该时刻的目标平移、目标旋转和目标缩放，即在该时刻该关节的目标变换。

至此，骨骼动画的数据格式定义完成，接下去就是脚本化的工作了。和前面的Mesh一样，可采用XML的格式来存储动画，如例程5-3所示。同时，脚本的层次结构和图5-6的层次结构一样。先申明角色的动画集（每一个动画分开描述），接着是以单个关节为单位的关键帧集，最后是以时刻为关键字描述关键帧。

**例程5-3　骨骼动画的脚本化**

```
<animations>
    <animation name="Attack1"length="1">
        <tracks>
          <track bone="Joint1">
            <keyframes>
              <keyframe time="0">
                <translate x="0"y="-14.8238"z="0"/>
```

```
            <rotate angle="0">
              <axis x="1"y="0"z="0"/>
            </rotate>
            <scale x="1"y="1"z="1"/>
          </keyframe>
            <keyframe time="0.275">
            ……                    //平移、旋转、缩放
            </keyframe>
            ……                    //别的关键帧
            </keyframes>
        </track>
      ……                          //别的骨头
      </tracks>
</animation>
……                                //别的动画
</animations>
```

# 5.2 骨骼动画的更新

在定义了骨骼动画的数据格式后，就需要在游戏中应用相应的骨骼动画，并显示令人兴奋的动画效果了！

最普通的骨骼动画的更新可以分为几个步骤：

（1）得到当前应用的动画。

（2）得到目前运行到该动画的什么时刻。

（3）确定该时刻之前和之后的两个关键帧，并根据该时刻与前后两个关键帧时刻的时间值插值计算出该时刻该骨骼相对于父骨骼的新变换矩阵。

（4）应用骨骼当前的变换矩阵，整体更新一遍骨架。

（5）根据骨架得到蒙皮网格的每一个顶点位置。

（6）根据网格模型顶点的新位置和朝向绘制角色网格。

下面一一分析这几个步骤的实现。

首先，得到当前该应用什么动画。要定义一个动画控制器类（Animation-Control），每个拥有骨骼动画的角色都会有个AnimationControl的对象，对象中记录了该角色的骨架及其拥有的动画集，同时，还记录当前应用的动画指针和需要混合的动画指针。关于动画混合后面会详细解释，简单地说就是将两个动画混合起来。有了动画控制器，就可以知道当前应用的动画了。

接着，在动画控制器类还需要记录当前动画的运行时间，比如一个动画的长度是5秒，现在运行到了3.7秒。当然，有些动画是需要循环播放的，比如角色的走路动画，这时候时间戳在动画循环时就要能够自动调整。

第三步是求出在该时刻骨架应用什么变换。这一步是最重要的一步。首先，对于骨架中的每一块骨骼，都要根据当前动画运行的时间戳确定该骨骼在该时间戳之前和之后的两个关键帧。在关键帧中记录了该骨骼在该时刻的变换，包括平移、旋转和缩放。当然，一般来说不会有平移信息，因为相对于父骨骼的平移很容易将骨架四分五裂。下面用一个例子来说明如何做骨骼动画的关键帧插值。

假设关键帧A和关键帧B都作用于骨骼K。A的作用时刻是Time1，B的作用时刻是Time2；同时，A的平移、旋转、缩放分量分别是T1、R1

和S1，B的平移、旋转、缩放分量分别是T2、R2和S2。那么在时刻Time（Time1<Time<Time2）的平移（T）、旋转（R）和缩放（S）分量的计算可以采用下面的公式：

$$scale=(Time-Time1)/(Time2-Time1)$$

$$T=T1\times scale+T2\times(1-scale)$$

$$R=R1\times scale+R2\times(1-scale)$$

$$S=S1\times scale+S2\times(1-scale)$$

该公式是采用线性插值的方法，这也是最简单的插值方法。在某些时候，线性插值会造成问题。比如会改变物体的体积，于是，在这种情况下，可以采用球面插值来实现。关于球面插值的公式和注意事项，读者可以参考相关书籍介绍。在得到骨骼在该时刻的平移、旋转和缩放分量后，骨骼的新变换矩阵也就自然得到了。

第四步是根据骨骼的当前变换矩阵，整体更新一遍骨架。骨架的更新是从根骨骼开始，层层向下更新。这和场景图的更新方式是一样的。在此步骤完成后，骨架就变成了该时刻应该呈献的POSE形状。

第五步是蒙皮骨骼动画的关键步骤，它负责使顶点能随着骨骼一齐运动。在这一步中，对于蒙皮网格中的每一个顶点，需要逐一计算它们在世界坐标中新的位置和朝向。所有这些新位置按照每一骨骼的影响权重加权求和。其权重和应该恰好为1。以计算顶点A的位置为例，首先找到影响顶点A的所有骨骼，然后计算每一骨骼对顶点A的影响之和。也就是说，计算在该骨骼独立作用下顶点的新位置。计算公式如下：

顶点的新位置=初始状态顶点的位置× 初始状态骨骼的世界变换矩阵的逆矩阵× 骨骼的新变换矩阵（I ）

在公式（I）中，为什么初始状态顶点的位置要先与初始状态骨骼世界变换矩阵的逆矩阵相乘呢？前面说过，骨骼的新变换矩阵是相对于父骨骼变换的；另一方面这个新变换矩阵是世界变换矩阵，它的任何变换是相对于世界坐标系原点的。因此需要把最初状态顶点变换到相当于父骨骼节点是原点的位置上，再进行矩阵变换。

对于第五步，还需要补充一个概念：二级网格容器（Secondary Mesh Con-tainer）。它是干什么的呢？在骨骼动画中，物体网格顶点的位置是需要改变的，但是却必须保留原顶点位置信息。因为每次顶点变形都是在原顶点位置基础上变换的，所以必须把骨骼动画中蒙皮网格顶点的位置信独立息保存，这个保存的地方就是二级网格容器，它是一个和原网格容器大小相同、格式相同的顶点缓冲区。

在完成了顶点的更新后，就只剩下模型的绘制了。骨骼动画中模型的绘制和传统模型的绘制在方法上并没有什么区别。唯一的区别就是绘制的是二级网格容器中的顶点，而不是一级网格容器中的顶点。

至此，骨骼动画的更新和绘制算是完成了。只要读者将本章第一节的内容理解后，本节应该是可以理解的。

## 5.3 进阶骨骼动画

上两节讲述的是最普通的骨骼动画，这样的动画只能算是够用，还不能满足口味越来越刁钻的玩家的需要。

比如说，骨骼动画动作设计过程中，对于每一个角色都需要单独设计一套骨骼动画，比如边走路边挥手和边走路边开枪就是两套不同的动画。不同的人走路时手的摆动幅度不同，于是小幅度的摆动手走路和大幅度的摆动手走路又是两个不同的动画。又比如一个角色从走路动画过渡到跳跃动画的过程。由于未知两个动画的切换时刻，所以两个动画的切换时刻极可能是非连续的，造成画面的跳跃，这是在游戏中一定要避免的问题。

本节将会简单介绍解决这两个问题的方法：ANIMATION BLEND和ANIMATION MORPHING。

## 1. ANIMATION BLEND

ANIM ATION BLEND解决了动画组合的问题，它可以在一个角色上同时作用多个动画。比如有两段动画，一段是走，另一段是开枪，将两个动画组合起来就是边走边开枪了。如果是将走路的动画和挥手的动画组合就是边走边挥手了。同理，多个动画也可以组合。有了这种技术后，美工的工作量就可以大大减少，而动画片断则如同零件一样，不同的零件组合在一起就构成了不同的大件（动画）。简单的代码设计如例程5-4所示。

**例程5-4  ANIMATION BLEND**

```
/**将动画A和动画B同时作用在骨架上
*  @param [Animation*] A：动画A
*  @param [Animation*] B：动画B
*  @param [float] PowerA：动画A的权重，默认为1
*  @param [float] PowerB：动画B的权重，默认为1
*/
```

```
voidAnimationBlend（Animation*A, Animation*B, floatPowerA=1.f,
                   floatPowerB=1.f）
{
    //应用动画A，对于骨架中的每一块骨头来说，A的变换矩阵都乘以PowerA
    //应用动画B，对于骨架中的每一块骨头来说，B的变换矩阵都乘以PowerB
}
```

## 2. ANIMATION MORPHING

ANIM ATION MORPHING是解决动画过渡问题的技术。当一个角色从动画A过渡到动画B时，在一段时间内，我们同时作用动画A和动画B，但是根据时间的变化赋予动画A和动画B不同的权重，如此动画的过渡就平滑。简单的代码实例如例程5-5所示。

**例程5-5　ANIMATION MORPHING**

```
/**将当前动画过渡到目标动画pTarget
*  @param [Animation*] pTarget：目标动画
*  @param [float] Time：动画过渡完成需要时间
*  @param [float&] temp:Morphing了多少时间，初始值为0
*  @return [bool]：是否Morphing结束
*/
boolAnimationMorphing(Animation*pTarget, floatTime, float&temp)
{
    //得到当前使用的动画
    Animation*pNow=GetCurrentAnimation();
    if(pTarget==pNow)             //前后的动画不能相同
        return true;
    floatscale=temp/Time;         //Morphing到的程度，0~1
    Matrix4 transformation=pNow->GetTransformation()*(1-scale)
             +pTarget->GetTransformation()*scale; //Morphing后的
变换
```

```
    //在骨架上应用变换，代码省略
    temp+=deltaTime;                //deltaTime是上一帧到本帧之间的
时间差
if(temp>=Time){
    pNow=pTarget;                   //Morphing结束
        return true;
    }
    else
        return false;
}
```

在ANIMATION BLEND和ANIMATION MORPHING上，还有很多可以增加的工作，比如上层使用如何封装、Morphing时间的动态选择等等。在这里只是做了个简单介绍，读者可以查阅介绍骨骼动画的书籍进行深入学习。

# 5.4　动画浏览器

随书光盘中附带了一个动画浏览器，它可以通过脚本配置的方法测试模型及动画文件的正确性，如图5-7所示。

图5-7　动画浏览器

# 第6章　粒子特效

粒子系统（Particle System）就是用大量的简单图元来表示某个物体，高性能的粒子系统可以真实地模拟出一些虚拟场景，比如火焰、风雪、血光等。一个粒子系统中有多种类型的子系统，每个子系统又由大量的粒子组成，高效地管理大量的粒子以及计算大量粒子的实时参数是一个难点。本章详细介绍了一个完整的粒子系统的设计与实现，其中包括自定义发射器和如何扩展，并介绍了如何设计并实现粒子系统编辑器，使特效的编辑所见即所得。

## 6.1　粒子系统的设计与实现

在自然界中，粒子以各种各样的方式聚集在一起形成，拥有了自身的属性和运动规律，构成了复杂的世界。一个常见的粒子系统由两部分组成：粒子和粒子群。粒子包括了一系列的属性：速度、颜色、纹理、位置等和一系列的行为。粒子群则管理一定数量的粒子，它负责创建粒子、更新粒子的属性、消亡粒子等，并利用这些粒子创造出一种特效。所以一个粒子群就是一种特效。如图6-1所示为两个粒子系统模拟的特效——火和烟。

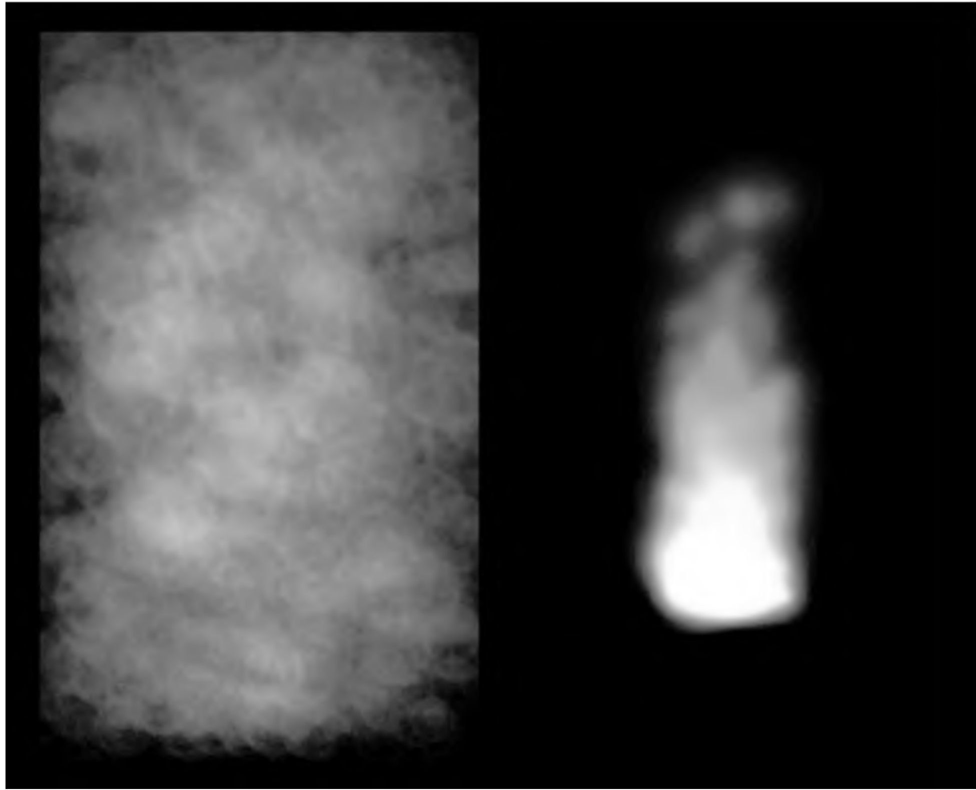


图6-1　用粒子系统模拟的烟和火

高级的粒子系统表达的物体越真实，粒子的数量就越多，同时会带来的是需要大量的计算和代码。因此设计一个好的粒子系统的数据结构很重要，否则将会大大地降低粒子系统的运行速度。

一个好的粒子系统应该达到以下目标。

（1）具有实时的效果：设计的粒子系统应该可以高效的计算每个粒子的各项参数，并且有剩余的时间让CPU进行其他的计算，以使系统流畅地运行。

（2）灵活性：可以动态地设置系统中的每个粒子的各项参数，进而控制粒子的行为，使得粒子系统可以模拟不同的特效。

除此之外，构造一个好的粒子系统的还应该使内存消耗尽可能低。所以，当一个粒子消失后，不能将它立刻释放，因为内存的释放和申 请很消耗时间的，而是设置粒子标志为消失。设置为消失的粒子可以被重生。最后，当一个粒子群自身的生命周期结束且该群下的所有粒子都消失后，才释放这个粒子群的内存。这个思想就像网络应用时的线程池，在这里我们将它称为粒子池。

在有了一些概念后，可以抽象定义粒子类。从物理上看，粒子是一个点，为了显示，这个点会有一个大小，同时，粒子有位置、颜色、加速度、速度、透明度等物理属性。此外，标志一个粒子生命的时间属性也非常重要。于是，粒子类（GEParticle）的数据成员如表6-1所示。

表6-1　粒子类GEParticle的数据成员

| 名称 | 数据类型 | 描述 |
|---|---|---|
| m_vPrevLocation | Vector3 | 上一帧更新时，粒子的位置 |
| m_vLocation | Vector3 | 粒子当前位置 |
| m_vVelocity | Vector3 | 粒子当前速度 |
| m_cColor | GEColor | 粒子当前颜色 |
| m_cColorDelta | GEColor | 粒子每帧更新时颜色变化的幅度 |
| m_fAge | float | 粒子当前的存在时间 |
| | | |

| m_fLifeTime | float | 粒子的消失时间 |
|---|---|---|
| m_fSize | float | 粒子的大小 |
| m_fSizeDelta | float | 粒子每次更新时大小变化的幅度 |
| m_fAlpha | float | 粒子的透明度 |
| m_fAlphaDelta | float | 粒子每次更新时透明度变化的幅度 |
| m_fAcceleration | float | 粒子受到的加速度大小 |
| m_fAccelerationDelta | float | 粒子每次更新时受到加速度变化的幅度 |
| m_pParent | GEParicleSystem* | 粒子所属的粒子群指针 |

在表中，可以发现几乎每个物理属性都有一个类似的XXDelta成员，这一个成员代表对应的那个物理属性的变换幅度。由于不希望产生的每个粒子都是一样大小、一样颜色的，所以有一个变化可以使特效不呆板、更自然。同时，给这个变化量定一个幅度，可以更特效更合理，看起来更逼真。

类GEParticle的函数成员非常简单，除了构造函数和析构函数外，只有一个更新函数bool GEParticle::Update（float fTimeDelta）。该函数每一帧都会调用，传入的参数是本帧调用和上帧调用的时间差。它实现了如下操作：

（1）更新粒子的生命值，检查其是否应该消亡，如是，则标志其为消亡且退出更新。

（2）记录上帧的位置，若是连线效果则记录给定时间前的位置。

（3）根据当前速度和两帧时间差得到现在的位置。

（4）根据加速度和时间差得到当前速度。

（5）如果该特效有向心引力，则根据引力计算速度。

（6）更新颜色、透明度、加速度和粒子大小。

在有了粒子后，需要类GEParticleSystem来管理粒子的产生、更新和消亡。在引擎中我们可以将粒子系统集成到场景管理中去，就可以更方便地管理特效了。比如有个人在抽烟，可将一个模拟烟的粒子系统挂载到香烟头上，这样烟就会随着香烟的运动而运动了。并且，粒子系统的发射器可以是多种形状的，比如普通的火特效的发射器是一个点，剑光特效的发射器是一条直线，而更多特效的发射器是一个长方体。因此，不同发射器要对应不同类，这些类都是通用发射器类的子类，重载发射函数。类层次关系如图6-2所示。

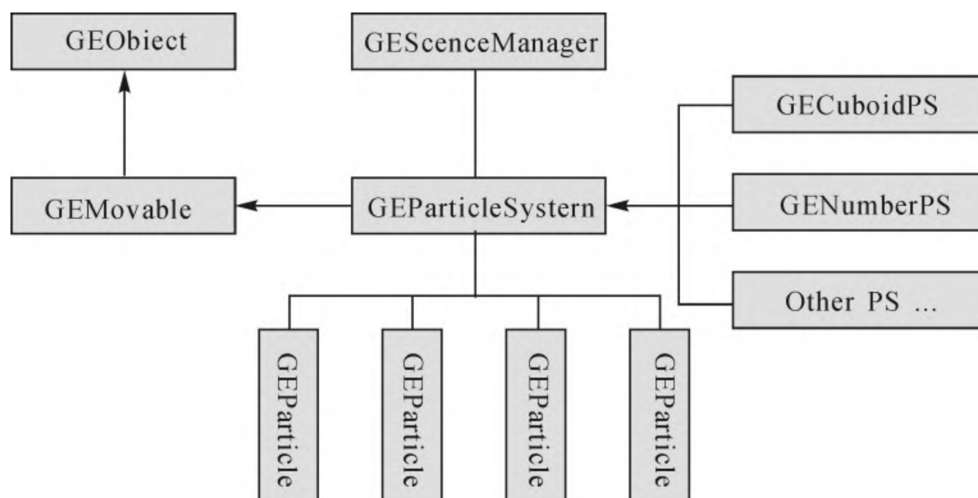



图6-2　粒子系统的类层次关系

粒子群类（GEParticleSystem）的设计关系到粒子系统的功能强大与否和效率高低。因此，在设计好类的层次关系后再定义粒子群类的数据结构，如表6-2所示。

表6-2　粒子群类GEParticleSystem的数据成员

| 名称 | 数据类型 | 描　述 |
| --- | --- | --- |
| m_strTexture | String | 粒子群使用的唯一纹理的文件名(一个粒子群只能使用唯一的相同纹理) |
| m_vPrevLocation | Vector3 | 上一次更新时,粒子群的位置 |
| m_uParticlesPerSec | unsigned int | 粒子群每秒钟发射的粒子数 |
| m_uParticlesAlive | unsigned int | 粒子群当前存在的粒子数 |
| m_fSpeed | float | 粒子群发射粒子的速度 |
| m_fSpeedVar | float | 粒子群发射粒子的速度变化幅度 |
| m_fLife | float | 粒子群发射的粒子的最大存在时间 |
| m_fLifeVar | float | 粒子群发射的粒子的最大存在时间变化幅度 |
| m_fAge | float | 粒子群剩余生命时间 |
| m_cColorStart | GEColor | 粒子群发射的粒子开始的颜色 |
| m_cColorVar | GEColor | 粒子群颜色控制的变化幅度 |
| m_cColorEnd | GEColor | 粒子群发射的粒子最后的颜色 |
| m_fSizeStart | float | 粒子群发射的粒子开始的大小 |
| m_fSizeVar | float | 粒子群大小控制的变化幅度 |
| m_fSizeEnd | float | 粒子群发射的粒子最后的大小 |
| m_fAlphaStart | float | 粒子群发射的粒子开始的透明度 |
| m_fAlphaVar | float | 粒子群透明度控制的变化幅度 |
| m_fAlphaEnd | float | 粒子群发射的粒子最后的透明度 |

续表

| 名称 | 数据类型 | 描　述 |
|---|---|---|
| m_vAccelerationStart | Vector3 | 粒子群发射的粒子最开始受到的加速度,可以模拟风、重力等作用 |
| m_fAccelerationVar | float | 粒子群加速度控制的变化幅度 |
| m_vAccelerationEnd | Vector3 | 粒子群发射的粒子最后受到的加速度 |
| m_fTheta | float | 粒子群的发射方向的变化幅度 |
| m_bIsMoving | bool | 可以控制粒子群是否随机移动 |
| m_bIsAttractive | bool | 可以控制粒子群是否对发射的粒子有引力 |
| m_bIsColliding | bool | 可以控制粒子群发射的粒子是否有碰撞 |
| m_bIsTracted | bool | 一种绘制方式,将该位置和上次的位置连接绘制,能产生如雨,光之类的效果 |
| m_fTimeLap4Track | float | 粒子的位置更新周期(用于模拟雨水,光等效果),只有当 m_bIsTracted 为真时有效 |
| m_fAttractPower | float | 引力加速度,只有在 m_bIsAttractive 为真的时候才作用 |
| m_rParticles | Particle * | 粒子池,动态决定大小,是个指针数组 |

这些参数基本一看即明，但有些数据成员非常奇妙，比如表示绘制方式的bool成员m_bIsTracted，以及表示引力效果的bool成员m_bIsAttractive。这两个效果开关虽小，但是可以产生不同凡响的效果。如图6-3所示，同样的设置，只是左图有划痕效果，而右图没有划痕效果，可以看到差别是非常大的。图6-4也展示了划痕效果的强大。

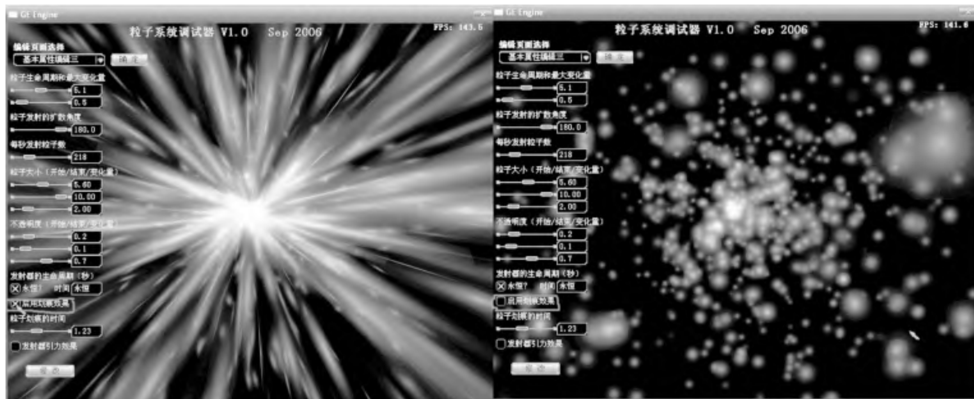



图6-3　划痕属性对效果影响比较

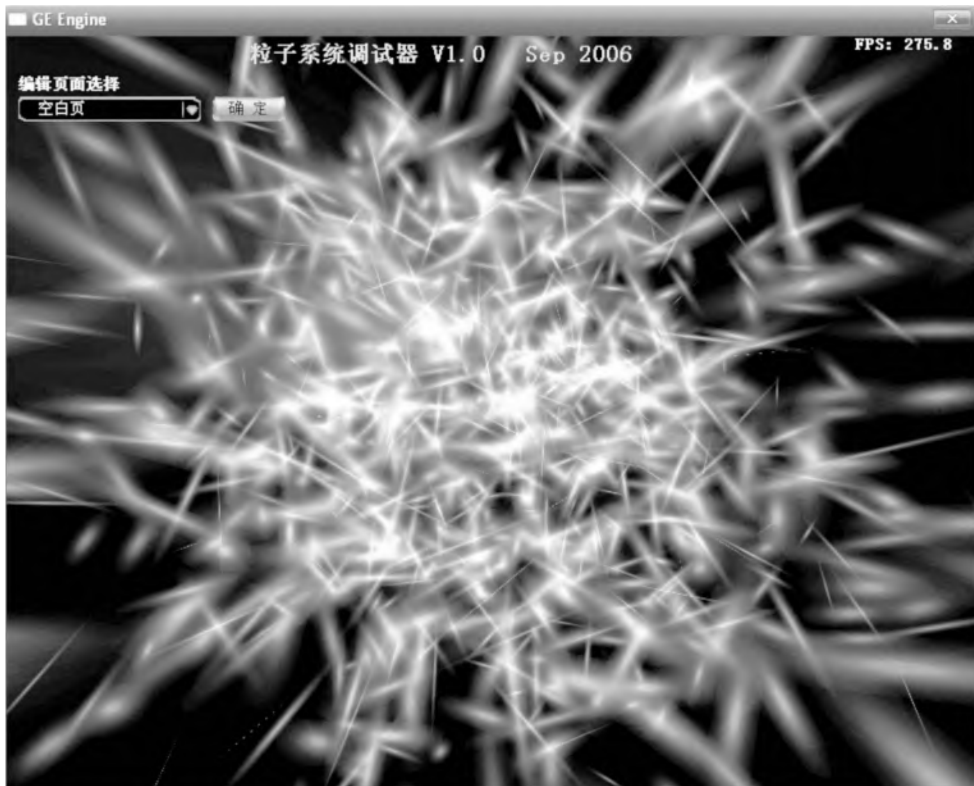



图6-4　划痕属性的杂光束效果

在领教了划痕的强大后，让我们再看看引力属性的威力。如图6-5所示，左右两图除了引力属性不同外，其余属性都相同。左图为无向

心引力的魔法效果，而右边为有向心引力的魔法效果。可以看出，两种效果的差别是非常大的。

m_fAge属性表明粒子群的剩余生命时间，用一个宏PS_LIFE_FOREVER表示永恒存在。当粒子群生命结束时它就不会产生新的粒子了，接着它将等待自己管理的所有粒子消失，至此，这个粒子群才算完成了自己的使命，最后从父节点上卸载并释放内存。

除此之外，粒子群类作为GEMovable类的子类，它集成了GEObject和GEMovable的所有属性。你可以将一个粒子系统挂接到场景图的任意一个节点上，它可以自由运动，也可以随父节点运动。令人开心的是：你不用为此多写一行代码，因为类GEObject和类GEMovable都帮你做了。

总的来说，这些属性都比较简单，如果能够巧用，会得到意外的惊喜！比如，在粒子群生命周期内更改粒子群的属性，使之从一种特效过渡到另一种特效，这种渐变效果相当不错。如图6-6所示，这两张图是从图6-4的效果渐变成图6-5右的效果。

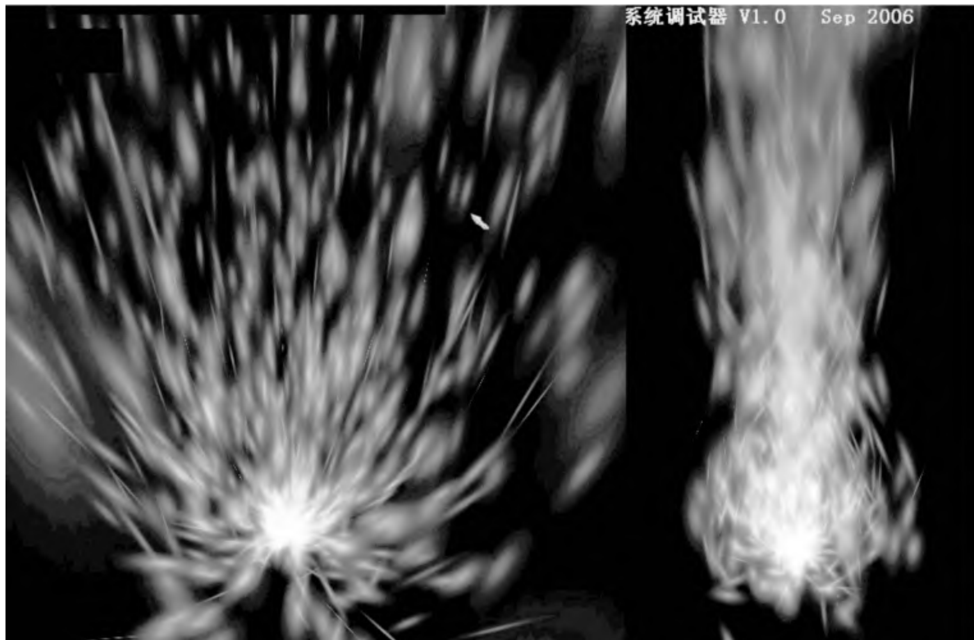



图6-5　引力属性对效果影响比较

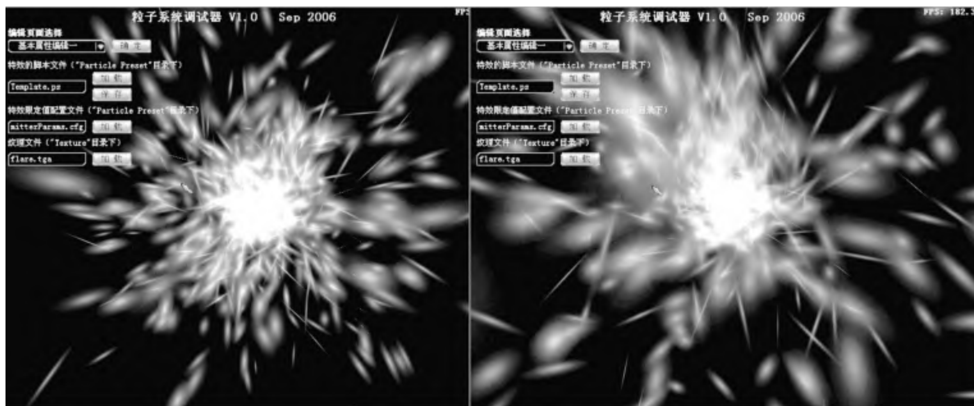



图6-6　效果的渐变

在设计了粒子群类的数据成员后，要开始设计它的成员函数了。首先当然是些Get、Set函数，这些函数比较简单，就不分析了。在类GEParticleSys-tem中，最重要的函数是更新函数和渲染函数。

例程6-1所示为更新函数。可以分解为以下步骤：

（1）调用父类的更新函数。

（2）调用发射器的一个用户特定更新（UserFunc4Emitter），这是一个预留接口，用户可以重载发射器类，并重载这个函数。比如，定义一个会不定时切换效果的发射器，在重载此函数时可以在函数内部写一段随机切换效果的代码。

（3）对于未被激活的系统返回。

（4）遍历管理的粒子池，对于存在的粒子调用其更新，并统计存在的粒子数量。在这一步中我们看到函数UserFunc4Particle被调用。这个函数与UserFunc4Emitter类似，都是一个默认实现为空的虚函数，如果一个自定义的粒子系统需要在运行时改变粒子的行为，则可以重载这个函数。顺便补充一下，这是设计模式里的一种思想－模板模式（Template），它定义一个操作中算法的骨架，将一些步骤的执行延迟到其子类中。利用模板模式可以使代码得到最大程度的重用，非常的灵活。

对于已经消亡的系统将其从父节点卸载，并释放内存。这里，对于已经消亡的系统的定义是剩余存活时间为负，但大于宏定义PS_LIFE_FOR EVER，并且粒子池中的所有粒子都已经结束自己的生命周期。

当粒子群的生命未完时调用函数_NewParticle()，产生粒子。

## 例程6-1　更新函数

```
HRESULTGEParticleSystem::FrameMove(floatdeltTime)
{
    GEMovable::FrameMove(deltTime);              //调用父类
```

```
    UserFunc4Emitter();                                     //可自定义发射器行
为，预留
    if(!m_bActived)                                        //是否激活
        return S_FALSE;
    m_uParticlesAlive=0;
    for(int i=0; i<PS_MAX_PARTICLES;++i)
    {
        if(m_rParticles[i].m_fAge>=0.0f)
            if(m_rParticles[i].Update(deltTime))
            {     //粒子不存在时Update返回FALSE
                UserFunc4Particle(m_rParticles[i]);
                ++m_uParticlesAlive;
            }
    }
    if(m_uParticlesAlive==0&&m_fAge>PS_LIFE_FOREVER&&m_fAge<=0)
    {     //说明此系统已经消亡
        this->m_pFather->Detach(m_strName);
        delete this;
        return S_FALSE;
    }
    //如果m_fAge<=PS_LIFE_FOREVER；就是说这个粒子系统是个永恒的系统
    if(m_fAge>0)
        m_fAge-=deltTime;
    //只有当系统还未消亡的时候才需要产生新的粒子
    if(m_fAge>0||m_fAge<=PS_LIFE_FOREVER)
        _NewParticle(deltTime);
    return S_OK;
}
```

在粒子系统的渲染步骤中，需要将所有存在的粒子以公告板（Billboard）的方式绘制出来。

公告板是一种始终朝着观察者的一个物体，通常它是一个多边形。因为3D场景中的粒子要始终正对着观察者，所以我们可以使用公告板技术来绘制粒子。这种应用的典型例子就是QuakeIII和Unreal中绚丽的子弹光焰。

渲染函数的执行步骤如下：

（1）设置材质、纹理和顶点数据格式。

（2）记录上次更新时的3D坐标。

（3）遍历粒子池中的每一个粒子，如果粒子是生存状态，则继续以下步骤，否则函数返回。

（4）粒子的绘制将由一个矩形应用相应纹理来实现。因此先填充矩形4个顶点的颜色信息和透明度信息。

（5）应用公告板技术，根据当前的照相机位置和朝向确定矩形4个顶点的坐标。

（6）将矩形绘制出来。

以上步骤中，第五步最为复杂，读者可以参考随书光盘中的源程序（GE-ParticleSystem.cpp）。由于渲染函数比较复杂，书中篇幅有限就不列出了。

在设计完粒子群后，需要在类GESceneManager中定义一个工厂函数负责在场景图中创建一个粒子系统，函数原型如下：

```
GEParticleSystem*CreateParticleSystem ( const string&name, string scriptsFile, GEObject*father,
```

```
      constVector3&offset=Vector3::ZERO,  float  radianPitch=0,
float
        radianYaw = 0,
            float radianRoll=0, bool isVisible=true）;
```

其中的scriptsFile是该粒子系统的脚本文件，会在6.3节介绍。

至此，一个功能比较齐全的粒子系统算是大功告成了。

# 6.2　关键帧技术及自定义发射器示例

本节要介绍粒子系统的两个高级话题：关键帧技术和自定义发射器。

所谓关键帧就是指粒子系统的行为可以以帧为单位来控制，或是说粒子的行为和时间满足一个函数关系。例如，有一个粒子系统来模拟烟火。烟火的行为是从一个点中喷射出来，到一定高度后像四周散开，最后消失；要模拟一个燃烧的中文字的特效，且字随着时间而变化。像这种粒子在中途行为发生很大变化的特效就要借助关键帧技术了。

在上一节的介绍中有两个函数：

（1）virtual void UserFunc4Particle（GEParticle&particle）;

（2）virtual void UserFunc4Emitter();

第一个函数是粒子的关键帧函数，第二个则是粒子群的关键帧函数。如果你想要实现一个关键帧，就必须重载类GEParticleSystem，

并重载这两个函数。

现在假设我们要实现烟火效果，我们可以如下步骤设计烟火类（Protech-nyPS）：

（1）ProtechnyPS继承自GEParticleSystem，多一个私有成员bool m_bExpo。

（2）ProtechnyPS的生命周期只有0.1秒（这个数值可以根据个人偏好，表示放烟火的瞬间）。如此，烟火只在瞬间有粒子喷出。

（3）重载函数UserFunc4Emitter，功能是在一定时候设置m_bExpo为真。

（4）重载函数UserFunc4Particle。函数中可以设置粒子在m_bExpo为真时改变行为，做散开状。并发出爆炸声。

经过如上四步，一个烟火特效就完成了。是不是很简单呢？这里还要给两个接口函数UserFunc4Particle和UserFunc4Emitter记上一功。而燃烧的中文字特效也差不多的设计思路，只需要重载UserFunc4Particle和UserFunc4Emitter便可。

接下来再看一下什么是自定义发射器。举个例子，一个火焰特效的所有粒子不能都从一个点发出，这样会失真。于是定义一个范围，所有的粒子都在这个范围内生成，这样就会真实很多。图6-7展示了效果的不同，其中左图的粒子是从一点发出，右图是从一个长方体的范围内发出。

类GEParticleSystem的函数_NewParticle是一个内部调用的保护函数。这个函数的功能是根据粒子群的属性生成一个新的粒子。在这

个函数内部会调用一个虚函数—Vector3 GEParticleSystem::EmitFromSomeShape()，这个函数负责调节生成粒子相对与该粒子群的平移，默认返回一个零向量。EmitFromSomeShape函数非常重要，重载这个函数就可以实现自定义的发射器形状。

而图6-7左所示的火焰就是利用了最简单的矩形发射器。关键步骤只有一步：重载影响粒子生成位置的虚函数EmitFromSomeShape()。只需要在函数中返回一个在长方形内的随机向量就可以了。又是那么的简单？是的！但简单不等于效果简单，图6-8实现了一个数学发射器，粒子发射出来后的构成了数字的形状。

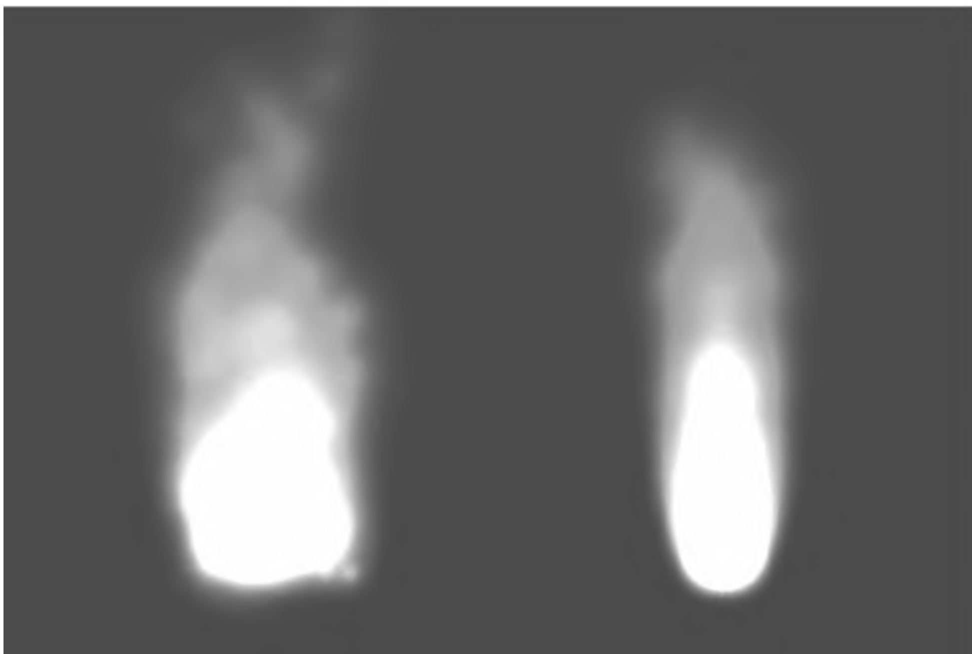


图6-7　不同发射器的火焰比较

图6-8　数字型自定义发射器

可以发现，利用关键帧技术和自定义发射器，粒子系统变得异常强大，它能模拟的特效更多更逼真了。

# 6.3　粒子系统编辑器

粒子系统的主要功能是模拟一系列特效。如果需要模拟出一个好的特效，就必须能适当地调节其参数。一般说来，这是美工的工作，但如果没有一个所见即所得的编辑器，粒子系统的调试就会非常困难！设计人员需要改完一个参数后，再运行程序，再改再运行……如此反复！为了使美工能更加方便地调试特效，引擎需要提供一个粒子系统的编辑器。在这个编辑器中，可以非常方便地调节各项参数、切换纹理、加载不同特效、保存特效等等。

如果特效不能脚本化，那么在编辑器中调试的特效就不能保存。所以，还必须先将特效脚本化。脚本化最大的好处就是粒子系统的效果不依赖于编译，比如我们对游戏中的一个特效不满意，只需要更改它的脚本文件然后保存就可以了，无需对程序重新编译。

粒子系统的脚本化的重点就是提取关键属性的值，并有条理地组织起来。例程6-2是图6-5右特效的脚本文件。这里没有采用XML脚本，而是用了自定义脚本，虽灵活简单，但是需要付出潜在风险的代价。这是一个薄弱环节，读者可以试着将它改为XML文件。它使用我们自己开发的脚本支持注释，支持整数、浮点数、bool值以及字符串的读写。所有的属性都用<属性名>来做前缀，后面跟对应的属性值。当然，因为此脚本没有形成一个标准，所以也可以自己定义你需要的格式，比如在GUI的脚本中，使用了<属性名_begin>属性<属性名_end>的格式。

### 例程6-2　魔法火焰特效的脚本

```
//发射器类型
<Emitter_Type>    PS_CUBOID
//发射器的配置文件路径
<Emitter_config>    EmitterParams.cfg
//TexturePath, All theTextures should can be found in the folder of 'Tex-ture'
<Texture>        flare.tga
//ColorStart        R.G.B(INTEGER 0~255)
<Color_start>    183    163    87
//ColorEnd      R.G.B(INTEGER 0~255)
<Color_end>        215    3    0
//ColorVariation    R.G.B(INTEGER 0~255)
<Color_var>        127    0    147
//AlphaStart, AlphaVar, AlphaEnd(FLOAT 0~1)
<Alpha>        0.47    0.14    0.129697
//SizeStart, SizeVar, SizeEnd(FLOAT>0)
<Size>            3.35    2    8.955
//Speed, SpeedVar(FLOAT>0)
<Speed>        34.0152    10
//Theta: for the varon the up direcionwhen emit(FLOAT 0~180)
```

```
<Theta>            179.99
//System Life
<Life>            -820912
//ParticleLife, LifeVar(FLOAT>0)
<PLife>            3.22    6.2
//AccelerationStart, AccelerationVar, AccelerationEnd
//(FLOAT VECTOR3  .X.Y.Z)
<Accel_start>    0    10    0
<Accel_var>        0.15625
<Accel_end>        0    64.8674    0
//Particles perSecond(INT    >0)
<Frequency>        370
//IsMoving, IsAttractive, IsColliding, IsConnectTract(BOOL:  0
or1)
<bMove>        0
<bAttractive>    1
<bCollid>        0
<bTract>        1
//Time lap forTrack  (FLOAT>0)
<Time4Track>    0.318182
<PS_Extra_Begin>
<Cuboid_Size>    8    0    8
<PS_Extra_End>
```

特效脚本本身并没有什么难点，只要了解属性对特效的影响便可以读懂了。

特效的脚本化后，需要设计编辑器了。编辑器的设计需要符合简单、快捷、人性化的几个基本要求。

整个编辑器目前有5个菜单界面。如图6-9所示是主菜单，在主菜单里可以选择要进入的子菜单。其中基本属性编辑界面一、二、三都

是粒子系统的通用属性，而类型附加属性编辑界面则是针对不同的特效类型而定。

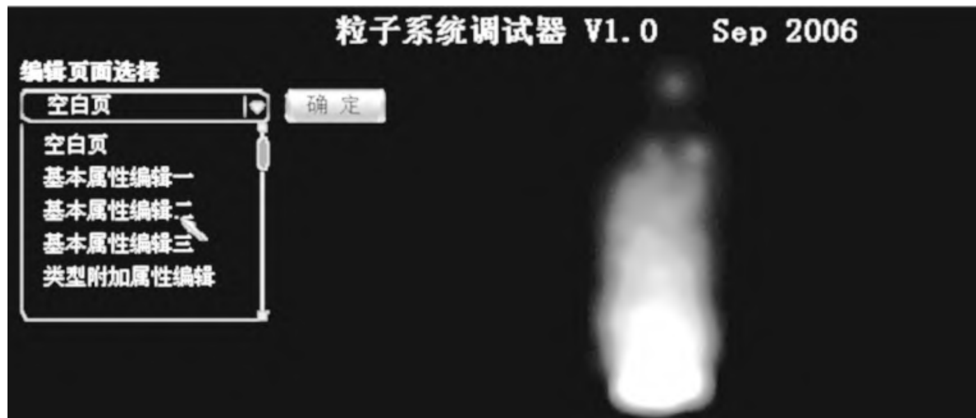



图6-9 特效编辑器主菜单界面

图6-10中是特效基本属性编辑界面一，在该界面中可以加载不同的特效脚本文件，也可以保存自己设定的粒子特效。此外，还可以加载不同的纹理作用在特效上。在此菜单中还可以看到“特效限定值配置文件”，这是一个对特效的取值范围的一个限定配置文件，关于此部分的内容，读者可以阅读源代码中类GEParticleSystem的头文件。

图6-10　特效基本属性编辑一界面

图6-11是特效基本属性编辑界面二、三。绝大多数的属性都可以在这两个界面中编辑。也许你注意到了每个滚动条后都跟了一个编辑框。这是什么呢？其实编辑框负责将滚动条对应属性的值显示出来，此外，编辑框还有一个非常重要的功能就是微调属性。因为滚动条对属性的调整都不是那么精确，比如想将X向的加速度调整为5.6，也许你怎么都无法完成，但通过在编辑框内输入需要的值却能够轻松完成。

图6-11　特效基本属性编辑二、三界面

图6-12则是类型附加属性编辑界面。此界面的形式由所编辑特效的发射器类型决定，比如一个长方体的发射器，则它的附加属性就是长宽高的调整，而一个数字型发射的附加属性则是数字的大小、间隔，以及显示什么数字。

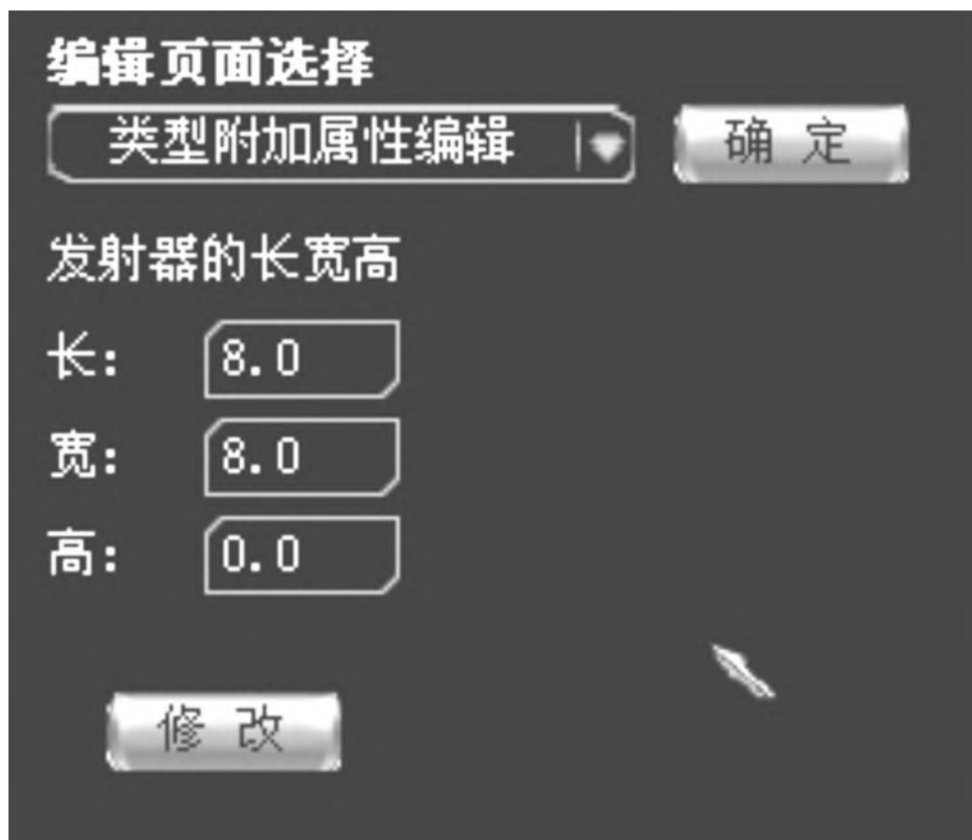



图6-12　特效附加属性编辑—长方体发射器

# 第7章　图形用户界面模块

图形用户界面（Graphic User Interface）模块，即GUI，是人和系统进行交互的接口，包括系统给用户提供各种信息，及用户通过键盘和鼠标等输入设备进行操作的媒介。

GUI模块主要包括各种图形界面的控件以及控件间的组织机制，用于处理游戏输入及响应消息机制和绘制GUI的渲染模块。GUI需要设计人性化，接近人的各种操作习惯。游戏引擎中的GUI是为游戏开发人员提供可编辑的图形用户界面，构成二维的控制面板及接收玩家的输入和显示系统的相应输出。

在游戏中，GUI的设计非常重要。图7-1所示为杚帝国时代III枠GUI。杚帝国时代枠的设计者们曾提出“前15分钟法则”。即对一个游戏来说，如果入门玩家不能在前15分钟顺利地弄明白基本操作和策略并进行游戏，或者铁杆玩家不能在前15分钟感到有趣和挑战的话，他们将永远离开这个游戏，不再进一步尝试。因此，前15分钟就决定了一个游戏的命运：是被惨遭删除，还是可以陪伴玩家度过几天甚至几个月。然而在短短的15分钟内，游戏可能还无法向用户展示全部的魅力，甚至游戏设计者在设计时想要炫耀的“与众不同”的技术和创意都没有时间来展示。因此，一款游戏要通过最初的挑剔，必须快速吸引住玩家，而游戏的界面设计是其中很重要的一环。一个方便美观的GUI设计也许就催生了一个经典游戏；反之，谁都不愿意玩一个粗糙界面的游戏。

图7-1　柸帝国时代III枠之GUI

游戏的界面设计与游戏的类型、整体风格联系十分紧密，没有给定的原则指出哪种界面设计是合理的或是吸引人的。尽管如此，游戏软件的界面也并非没有规律可循。

（1）游戏的界面清晰明了。没有人会愿意在玩游戏的时候还要按照很繁琐步骤设置游戏。

（2）对于操作复杂的游戏，提供类似教程活向导资料，帮助用户快速上手。

（3）游戏的界面与游戏的整体风格相一致。不能为了特别突出游戏的界面而破坏更重要的游戏整体风格。

（4）游戏的界面设计中还应考虑到国际化的需求。在设计游戏界面中的文字、图片要考虑到发行国家的风土人情和习惯。使用地国家禁忌的文字、图片更不能在游戏界面中出现。另外，由于各个国家文字长短不一，当一个游戏的界面用一种文字表示的时候可能搭配合适，用另外一种文字表示的时候可能就显得界面搭配失衡。在游戏界面的设计中也要考虑到这个问题。

正因为GUI是如此的重要，所以一个完整的游戏引擎必定要自己设计GUI，以满足需求。在本章中，将会着重讲述如何设计并组织GUI控件以及如何设计一个的GUI辅助编辑器。

# 7.1 GUI模块构架

GUI模块中最基本的组成是控件，控件可以有很多种：文本标签、按钮、滚动条、下拉列表框、列表框等等。通过观察可以发现它们都有一些相同的属性，比如名字、位置、是否可见、是否可以编辑、控件大小等等，因此我们设计所有的控件类都继承自一个基类（GEGUIElementBase），基类中定义控件共有的属性和方法。

控件放在哪里呢？当然是一个页面中，即GUI容器，在引擎中对应类GEGUIContainer。GUI容器类和控件类的关系是聚集（aggregation），容器类中有一个基类指针数组，存放着页面内所有控件的指针，如此，容器类就可以更新、显示控件了。在某些情况下，一个页面也可以被镶嵌到另一个页面中，所以容器类本身也必须是GEGUIElementBase的子类。

一个游戏中会有很多GUI页面，比如开始菜单页面、物品属性页面、结束页面等等。所以，需要有一个机制去管理那么多的GUI页面，这就是GUI管理器类（GEGUIManager）的作用了。GUI管理器类中维护着一个游戏中所有GUI页面的指针数组，它负责调度页面的更新、显示、操作。图7-2所示为引擎中GUI模块的类图示意。

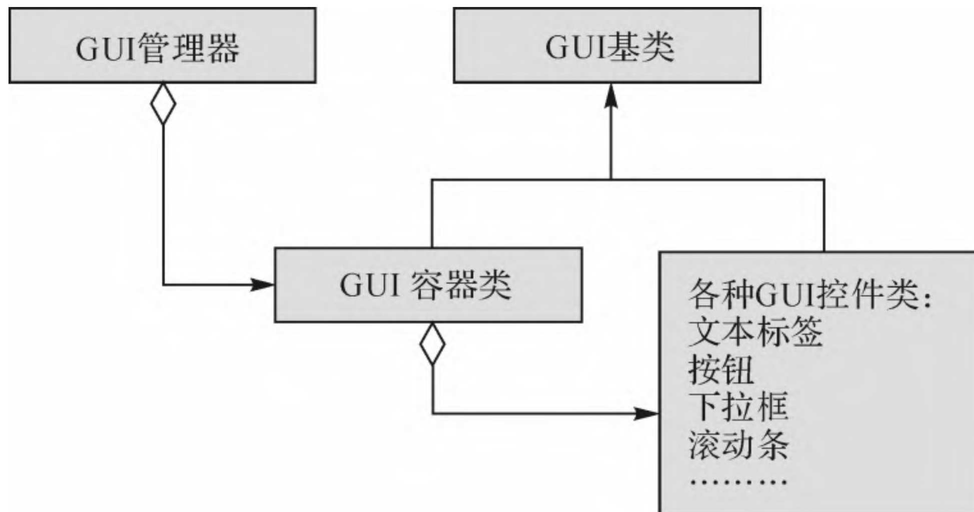



图7-2　GUI模块类图示意

至此，就有了GUI管理器→GUI容器→GUI控件这样一个控制流了。假设要调节页面P中的一个控件C，就可以先通过GUI管理器得到页面P的指针，然后通过P的指针得到控件C的指针，再操作这个指针。

解决了如何控制GUI控件后，需要设计GUI控件的创建。由于GUI控件是归GUI容器管理的，所以GUI控件的创建也应该由GUI容器负责。但是，GUI控件的种类繁多，需要创建函数的接口统一，以便将来的扩展，可以使用设计模式中的Builder方法，接口如下：

```
GEGUIElementBase*GEGUIContainer::CreateElement(GEGUIElementType
type);
```

其中，参数中的GEGUIElementType是一个枚举类型，所有控件的类型都有枚举。于是，如果需要在一个容器中增加一个控件，只需调用容器的Crea-teElement函数就可以了。函数会返回一个控件基类指针，程序负可以显式地将它转化为正确的类型。如果将来需要增加一个控件类型，此创建函数的接口无需改变，只需要在

GEGUIElementType中增加新类型就可以了。这是一种良好的设计，在可能的情况下，需要让引擎足够灵活且接口变动最小。

与GUI控件的创建类似，控件的删除也是通过GUI容器来操作的。因此，我们可以归纳为GUI容器负责其下属的GUI控件的所有操作，包括创建、删除、更新、渲染等等。

# 7.2 GUI控件

GUI模块中的核心就是GUI控件，控件设计的好坏直接关系到模块的可用性。常用的控件包括文字标签、图片框、按钮、文本编辑框、滚动条、列表框、下拉列表框、选择框、组选框等。这些控件一般都十分简单，但它们是GUI模块的基石，可以用它们来组成更复杂的GUI控件，比如文件搜索框控件。由于这些控件都有很多公共的特征，所以我们设计了一个GUI基类（GEGUIElementBase），例程7-1中说明了基类中有哪些基本数据成员。

**例程7-1 GUI基类的数据成员**

```
/**GUI组件基类*/
classGEGUIElementBase
{
protected:
    GEGUIElementTypem_eType;                //控件类型
    stringm_strName;                        //控件名字
    intm_nID;                               //控件ID，唯一
    Positionm_OffsetFromParent;             //控件和其父亲的位移
    Positionm_Position;                     //控件的位置
    Sizem_Size;                             //大小
    boolm_bEnable;                          //是否可编辑
```

```
    boolm_bVisible;                                    //是否可见
    intm_iShowLevel;                                    //在显示时的层次，0在最
下
    GEGUIElementBase*m_pParent;                        //父控件指针
    boolm_bFocus;                                      //是否被Tab键切换到
    boolm_bTabEnable;                                  //是否支持Tab键切换
    intm_iTabIndex;                                    //Tab键切换的顺序
};
```

大部分数据成员都比较简单，但以下几个成员需要稍稍注意一下：

- m_nID表示控件的ID，一个控件的ID在游戏应用中是唯一的、不重复的，由GUI管理器统一分配并管理。
- m_pParent是父控件的指针，父控件可以是GUI容器，也可以是一个控件，比如一个选择框控件就是由一个按钮控件和一个静态文本框控件组成的。因此父控件指针的类型是公共基类GEGUIElementBase。
- m_OffsetFromParent成员则保存控件与其父控件的相对平移，这里选择保存相对平移的原因和场景节点保存与父节点的相对平移的原因有些像，都是为了方便操作，同时也方便了GUI控件的脚本化（关于控件的脚本化将会在第三节详细介绍）。

控件基类GEGUIElementBase的成员函数大多是Get和Set函数，负责操作其数据成员。因为比较简单，这里因为篇幅有限就不一一列出，大家可以阅读随书光盘中EngineSource\Framework\GEGUIElementBase.cpp中的代码。除此之外，还有以下四个模板函数：

- virtual HRESULT Render()=0;
  负责渲染控件。它是一个纯虚函数，每一个派生出的控件都要实现。
- virtual HRESULT Update() {return S_OK;}
  控件的更新函数，内部实现输入输出。
- virtual HRESULT OneTimeInit() {return S_OK;}
  控件最开始的初始化，一般都为空。
- virtual HRESULT FinalCleanUp() {return S_OK;}
  控件在被删除前的行为定义。

在了解了GUI控件基类的设计后，下面介绍九大基础控件的设计与实现。

## 7.2.1　文字标签

文字标签就是静态文本框，它负责将一段文本以一定的形式显示到屏幕上去。如图7-3所示，文本标签类继承自GUI控件基类。

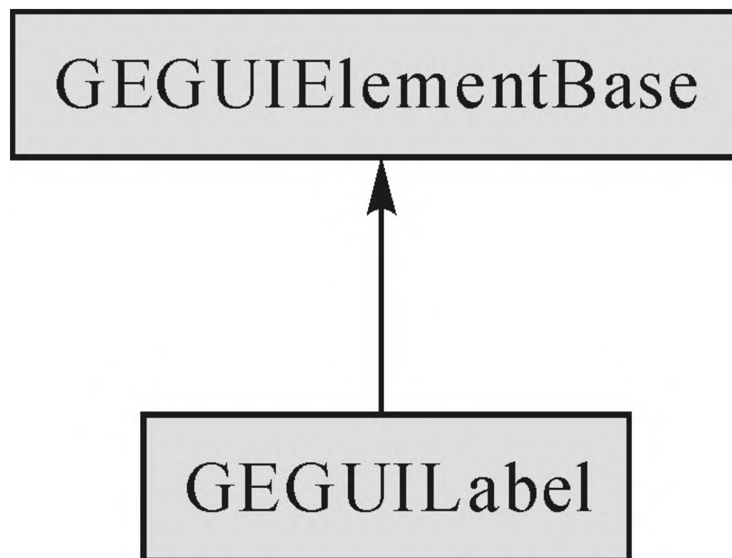


图7-3　文本标签类的继承关系

文字标签控件是所有控件中最基本最简单的控件（图7-4所示为一个文字标签实例），同时它也是GUI模块中用得最多的控件。用得多有两层含义，一是由于游戏中有很多信息需要在屏幕上显示，所以它被使用多；二是因为它经常被用于别的控件的组成，比如按钮上的文字、选择框上的文字其实都是文字标签。

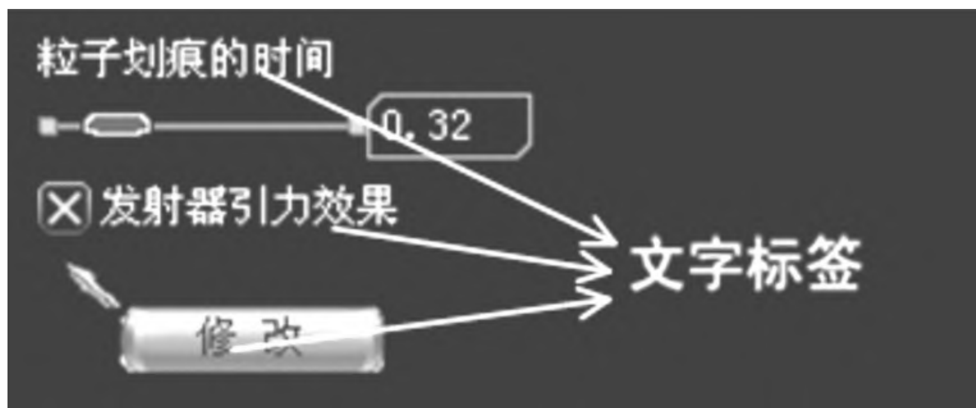



图7-4　文字标签实例

文字标签控件除了继承自控件基类的基本功能外，它还要满足以下几点需求：

（1）能够设置字体，包括字体、字号、样式（粗、斜、普通）。

（2）能够设置文本在文本框内的对齐形式。

（3）能够设置文本的颜色。

（4）能够设置文本的内容。

针对这四点需求，在文本控件类（GEGUILabel）中需要增加以下几个数据成员：

（1）string m_sText;

文字标签的显示文字。

（2）int m_iFontID;

文字标签的字体的ID。此ID由字体管理器（GEFontManager）管理，如果你需要使用一个未被创建的字体，则需要通过字体管理器来创建并返回给你此字体的ID。

（3）GEColor m_Color;

文字的颜色。此颜色还带有半透明信息，因此文字标签支持半透明效果。

（4）ALIGN_FORMAT m_Format;

标签文字的对齐方式。ALIGN_FORMAT是一个枚举类型，它有九种对齐方式，分别是左上角对齐（AF_TOP_LEFT）、左垂直居中对齐（AF_VMD_LEFT）、左下角对齐（AF_BTM_LEFT）、上水平居中对齐（AF_TOP_CTR）、水平垂直都居中对齐（AF_VMD_CTR）、下水平居中对齐（AF_BTM_CTR）、右上角对齐（AF_TOP_RT）、右垂直局中对齐（AF_VMD_RT）、右下角对齐（AF_BTM_RT）。齐全的对齐方式可以使GUI的界面更加美观。

文本标签类还需要实现virtual HRESULT GEGUILElementBase::Render()方法，将标签文字正确显示出来。

## 7.2.2　图片框

图片框控件负责在屏幕上以2维的形式显示图片。图片框控件类也是继承自GUI控件基类，如图7-5所示。

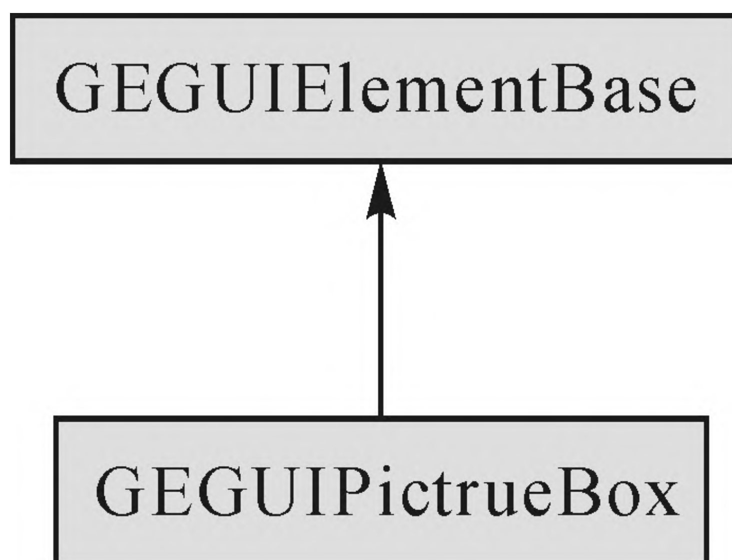



图7-5　图片框类的继承关系

图片框也是非常重要非常基础的控件，可以说是由图片框控件和文字标签控件一起组成了其他所有控件。比如按钮中的背景图、列表框的背景图等等其实都是图片框。图7-6展示了在游戏中图片框控件的应用。在图中，背景图是一个图片框控件，按钮的背景图是由几个图片框构成的，图片的下方是一个半透明的图片框。

图7-6　图片框使用示例—游戏的菜单界面

图片框的主要功能是显示图片，可以按如下方式设计它的数据成员：

- string m_sImage
  图片框控件需要存储显示纹理的名字。因为引擎实现了内存管理，所以可以通过图片的名字向纹理管理器发送请求从而得到纹理指针。
- GEColor m_cDiffuse
  有时候显示图片时希望能够改变色调，比如将一个亮的场景适当变黑或是将菜单背景随机变色，这时就可以用颜色混合来实现了。
- GE_TLVERTEX m_vVertex[4]
  纹理的显示需要给出4个顶点，这很简单。这里是4个经过变换

的2维顶点。

- 因为很多时候我们不需要显示整张纹理图片，而是其中的一部分，所以我们还得记录4个点的UV坐标值。

图片框控件的函数设计也不会很难，只需要提供图片的更改、色调更改等方法就好了，由于篇幅有限，大家可以参照附盘中的代码进行学习（Engine-Source\Framework\）。但有两点要注意，一是控件所有的位置相关函数都要重载，比如SetPosition、Move等函数，原因是控件位置更改时需要更新用于显示的4个顶点值；二是不要忘记重载基类中的FinalCleanUp函数，因为图片框控件会在内存中加载用于显示的纹理图片，所以在控件生命周期结束前需要释放这段内存。

### 7.2.3 按钮、选择框、组选框

按钮是GUI中第三大控件，它的设计较文字标签和图片框稍微复杂一些。按钮，意如其文，其明显的功能就是可以用鼠标点击，并且有反应，至于反应是外观上有变化还是有事件响应就是另外一回事了。那么，有哪些控件符合我们定义的按钮？首先，传统的按钮当然属于按钮，比如图7-6中的按钮；单选框也是，它也是按下去有反应，只是它的反应体现在外观上而已；单选框是了，组选框当然也是了。所以，按钮、选择框、组选框都属于按钮一类，它们都响应鼠标点击事件，它们之间最大的不同在于如何相应点击事件。

面向对象软件设计的一个要求就是最大程度地重用代码，既然按钮、选择框、组选框同属一类，那么我们就设计它们继承自按钮基类，如图7-7所示。

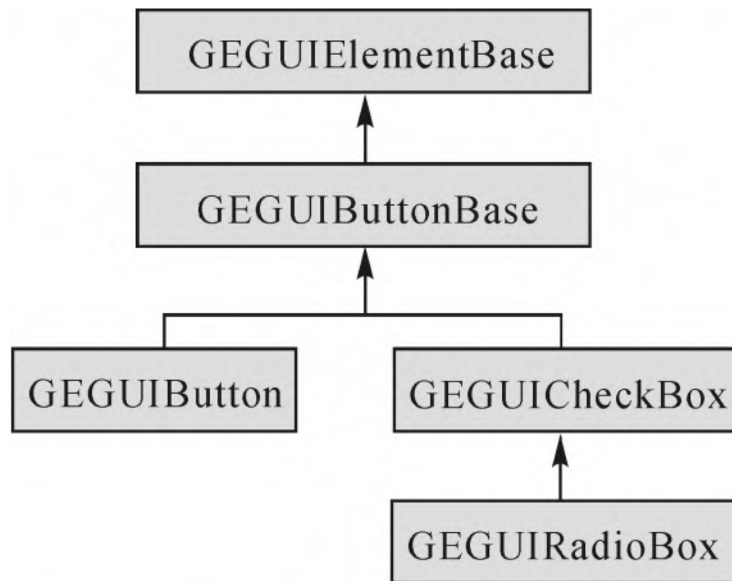



图7-7　按钮类继承关系

按钮基类GEGUIButtonBase是继承自控件基类GEGUIElementBase，它是所有有类似按钮行为的控件的基类。类GEGUIButton表示普通的按钮，直接继承自类GEGUIButtonBase。单选框对应类GEGUICheckBox，也是直接继承自GEGUIButtonBase，而组选框GEGUIRadioBox则是继承自GEGUICheckBox，这是因为组选框是一种特殊的选择框，它只是比普通的选择框多了组的信息，且定义同一组中的组选框最多只能选中一个。

在分析了类的层次结构后，便开始设计按钮基类的数据和行为。因为之前对按钮已经有了一个定性的认识，所以这一步就变得水到渠成了。

同时，需要定义按钮的几种状态。一般来说，按钮有四种状态：普通状态、鼠标移上去后的提示状态、被点下后的状态以及不可点状态。可定义一个枚举结构表示按钮的四种不同状态：

enum ButtonState{BTS_NORMAL，BTS_MOUSEOVER，BTS_BUTTONDOWN，BTS_DISABLE};

所以，一个按钮最多会对应4张不同的图片，每张图片都对应一种状态。当然，如果一个按钮的两种状态都是一样的表现形式，就可以少一张图。

此外，按钮上还要有文字。这个文字就是一个文字标签控件，它负责显示按钮上的提示文字。

至此，总结一下按钮基类有哪些数据成员：四个图片框控件指针、一个文字标签指针和一个按钮状态枚举变量。

有了数据成员，按钮基类还需要一些操作。首先，它需要一些操作图片框的函数，如例程7-2所示，这些函数包括得到符合当前按钮状态的图片框、设置对应按钮状态的图片框等等。其次按钮上的文字标签也需要提供完整的一套操作函数，比如更改内容、更改字体、更改显示颜色、更改对齐方式等等。最后还需要提供按钮状态的Get和Set函数。

**例程7-2　类GEGUIButtonBase部分函数**

```
classGEGUIButtonBase
{
Public:
    /**得到符合当前按钮状态的图片框控件指针*/
    GEGUIPictureBox*GetPicture()const;
    /**得到相应按钮状态对应的图片框控件指针*/
    GEGUIPictureBox*GetPiciure(constButtonState&state)const;
    /**设置相应按钮状态对应的图片框控件*/
            void   SetPicture(constButtonState&state,   const
```

```
string&image);
                    void     SetPicture(constButtonState&state,
GEGUIPictureBox*pic);
    /**文字标签的控制函数*/
    GEGUILabel*GetLabel()const;                              //Get
函数
    void SetLabel(GEGUILabel*label);                         //Set
函数
    void SetLabelText(conststring&text);                     //文字
标签内容
    void SetLabelPosition(int lefttopx, int lefttopy);       //文字
标签文本位置
    void SetLabelSize(intwidth, intheight);                  //文字
标签大小
    void SetLabelAlignMode(constALIGN_FORMAT&format);        //文字
标签对齐方式
    void SetLabelTextColor(constGEColor&colorkey);           //文字
标签文字颜色
    /**按钮状态的函数*/
    constButtonState&GetButtonState()const;                  //Get
函数
    void SetButtonStatus(constButtonState&state);            //Set
函数
}
```

在提供函数后还必须重载位置调整相关的函数，比如SetPosition、Move等函数，以及实现Update和Render等接口。至此就已经完成了为实现按钮、选择框和组选框而设计的基础类GEGUIButtonBase。图7-8所示为选择框、组选框图示。

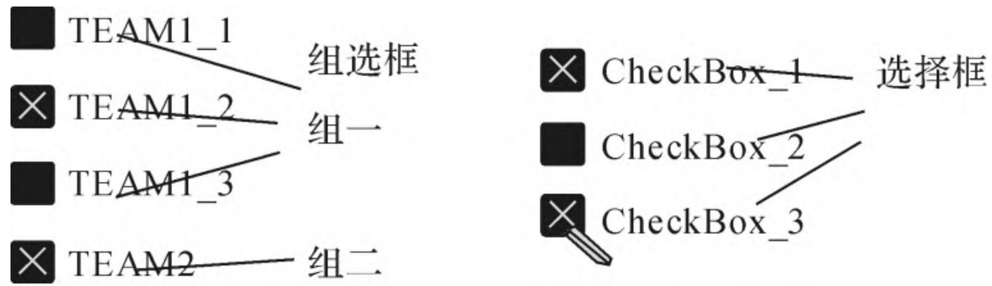



图7-8　选择框、组选框图示

类GEGUIButton是普通的按钮类，它的主要特征是能触发事件，因此我们要提供消息响应机制，并且要能够绑定全局函数和类函数，即提供如下接口：

- void BindFunction（void（*func）()）；全局函数绑定
- template<typename T>void BindFunction（T*object，void（T::*func）()）；类函数绑定

关于其内部实现，可以参考光盘中的源程序：GEEventHandler.h和GEGUIButton.h。

同时，每帧都需要检查按钮的状态更新，例程7-3用伪代码表示更新逻辑。

**例程7-3　按钮的状态更新逻辑**

```
if（按钮是disable状态）
    return;
else if（鼠标停留在按钮上）{
    switch（按钮的当前状态）
    {
    case一般状态：
        如果鼠标左键没有被按下，则切换状态到"按钮高亮提示"
```

```
        break;
    case按钮高亮提示:
        如果鼠标左键刚被按下，则切换状态到"按钮按下"
        break;
    case按钮按下:
        如果鼠标左键刚被松开，则切换状态到"按钮高亮提示"，并触发函数
        break;
    }
}
else
    设置按钮状态为"一般状态"
```

选择框的设计比按钮设计更简单，它们的基本逻辑都是一样的，只是选择框不需要触发事件，读者可以参照随书光盘中的源代码进行学习。

组选框要稍微复杂一点，它需要保证同一组中的组选框有且只能有一个被选中。如此一来，可以设计成每一个组选框都知道有哪些别的组选框和自己是一组的，那么当它被选中时就可以将另一个的选中取消。但这种设计却是不合理的，因为在这中设计中，一个组中所有控件都必须保存其组内所有控件的信息，造成信息冗余。于是，我们想应该设计一个第三方来管理每一个组选框组。这就是GEGUIRadioBoxGroup和GEGUIRadioBoxManager的作用了。

GEGUIRadioBoxGroup负责同一个组的控件互斥，当一个组选框被点选时，它会去请求对应的组（GEGUIRadioBoxGroup）设置其状态为选中，而同时原来被选中的那个组选框就设置成未被选中了。

而GEGUIRadioBoxManager负责创建和删除组。当创建一个组选框的组属性为新值时就创建一个新的组，而当删除一个组选框时检查其

对应的组内的控件数，如果剩余组选框数为零时就要将这个组删除。

这两个类的功能比较简单，大家可以阅读光盘中的源代码进行学习。

### 7.2.4　滚动条

在有限的范围内显示更多的内容，这就是滚动条控件最大的功能。在很多控件中都要使用滚动条作为组成，比如列表框、文本编辑框、下拉列表框等等，甚至图片框也可以加入滚动条，可以看的出来，滚动条也是一个基本控件。从形式上来说，滚动条分为横行滚动条和纵向滚动条，它们的功能基本一样，只是表现形式不同而已了。从组成上来说，滚动条是由四个部分组成的：左（上）端头、右（下）端头、中轴以及最重要的调节块。如图7-9所示。

图7-9　滚动条图示

既然横行滚动条和纵向滚动条在功能上没什么区别，只是表现形式不同，所以它们可以从一个滚动条基类（GEGUIScrollBar）继承而

来，基类将滚动条的功能抽象出来，只是将表现层的内容放到子类去实现。如此，类的层次就比较清晰了。而且，甚至可以轻松实现斜的滚动条或是圈型的滚动条。滚动条类层次关系如图7-10所示。

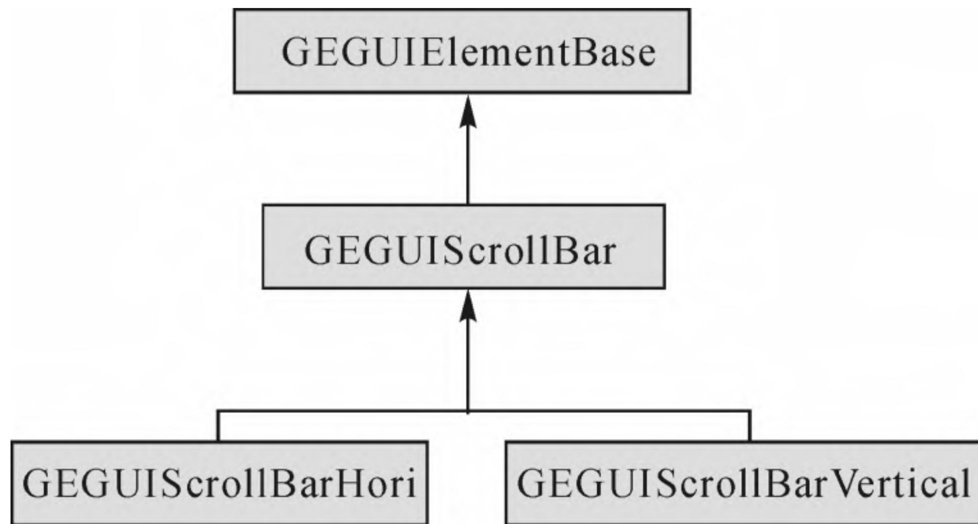



图7-10　滚动条类层次关系

任何时候，滚动条都有个值，这个值是和调节条与滚动条控件的相对位置相关的。因此需要先定义一个计算公式：

$M=(D_{LT}-D_{BL})/(L_B-L_T)$

式中，M为滚动条的值；

$D_{LT}$ 为左（上）端头与调节条在长度方向的相对位移；

$D_{BL}$ 为调节条与中轴在长度方向的相对位移；

$L_B$ 为中轴的长度；

$L_T$ 为调节条的长度。

于是，随着调节条的移动，可以得到0~1的值，0表示调节条和左（上）端头贴着，1则表示调节条和右（下）端头贴着。

其次，我们还需实现设置图片并重载位置相关函数。这一步的思路和按钮类相似，就不细说了，大家可以阅读光盘中源代码（EngineSource\Frame-work\GEGUIScrollBar.cpp）。

实现了滚动条基类后，横向滚动条和纵向滚动条的实现也是水到渠成的事了。这两者之间除了界面上的区别外，在调节条的移动上还有一点小区别，一个是横行移动，一个是纵向移动。就这么简单？是的，就这么简单，就可以将滚动条设计出来了。

### 7.2.5　文本编辑框

毫不夸张地说，文本编辑框是GUI基础控件中最复杂的控件了，同时也是使用频率相当高的控件之一。之所以复杂，是因为它的模式多、操作复杂。让我们先来定义一下它的需求。

（1）能够输入文字、删除文字、定位文字。

（2）支持不同的输入模式：纯数字型输入、浮点型输入、字母输入、密码输入。

（3）能够支持不同的输入法。

（4）支持不同的风格：单行的编辑框、多行的编辑框，并都要有滚动条集成。

对于第一条需求，必须能够得到键盘输入，并同时定义一个光标类，负责输入框中的光标定位。光标类需要记录的是对应文本编辑框指针以及在文本框中第几个字母后显示。同时，我们还需要注意到，在一个界面中，最多只能有一个光标存在，所以我们可以在类GEGUITexture中保留一个静态光标的对象。这样就能保证光标存在的唯一性。有了光标后，我们就可以在编辑框中编辑文字了，所有增加的文字必定在光标后一位，所有删除的文字必定是光标前一位。

第二条需求要求文本编辑框能够根据输入模式的不同而有不同的反应。比如纯数字输入时就不会对字母和标点有反应，而密码输入时所有输入的表现形式都是*。因此，在文本输入框内定义一个枚举类型，表示各种输入模式：

```
enum SingleLineType{TM_ASC, TM_INT, TM_FLOAT, TM_PSW};
```

然后，针对不同的输入模式，在控件的更新函数Update中作不同处理。

第三条需求是支持不同的输入法，这里主要指中文输入法。由于DirectX Input没有处理中文输入，所以需要自己写一个输入法类。类的功能是能够调用已经安装在系统中的输入法，并正确得到输入的文字。控件则负责每次更新时请求输入法类返回新的文字输入。关于输入法类的编写，不是本章的重点，资料也比较多，大家可以通过阅读源代码中的类GEIME学习如何编写输入法类。

最后一个需求是提供单行和多行两种显示模式，并提供滚动条的集成。单行和多行最本质的区别是在显示上，一个只能输入一行，也就是不支持换行符，当输入的文字宽度超出可显示的区域后，文字自

动向后滚动。而多行不但需要支持换行符，而且当输入的单行文字宽度超出可显示的区域后执行自动换行操作。同时，多行字符框需要集成纵向滚动条，使之可以更方便地浏览文字内容。图7-11是一个文本框图例。

图7-11　文本框图例

## 7.2.6　列表框

列表框的功能是在一个矩形区域内将一系列条目名列出来，并且能用鼠 标点选任意的条目。它的组成单元有以下几个：四周的边框、中间的背景图和可选的右滚动条和下滚动条，如图7-12所示。

图7-12　列表框图示（左）列表框类继承关系（右）

设计列表框时，首先要考虑的是用什么来存储列表框中的条目。这里可能选择是：数组（array）、向量（vector）、链表（list）……数组虽然最快，但它是定长的，所以不能使用；向量倒是一个不错的选择，既能调节容量，速度也快；链表虽然在容量的调节上更有效率，但遍历太慢，也不能使用。综合来看，向量是最好的方案。最后我们还希望头尾的插入删除方便，因此选择了双向队列std::deque<string>作为解决方案。

此外，还需要记录被选中的条目的索引号，类里用一个整型来记录，并将其初始化为－1，表示一开始没有任何被选中。至此，列表框的数据成员就设计好了。

列表框的基本操作和别的控件也类似，唯一不同的就是需要判断哪些条目需要显示在控件内。这个方法在void GEGUIListBox::_DrawList()中实现。它的主要思路是根据滚动条的值以及此控件可以显示几行条目得到在控件内显示的第一行条目的索引号，然后逐次显示，如果是显示被选中的条目时则换一种不同的颜色。

## 7.2.7　下拉列表框

下拉列表框和列表框非常相像。它的特殊点在于可以控制是否将管理的列表框显示出来，所以它可以在极小的范围内显示更多的内容。基于这个特点，下拉列表框经常被用于显示空间不够的情况，比如游戏中的状态菜单、属性选择等等。

如图7-13所示，下拉列表框由一个头和一个列表框组成。头部的左侧显示的是选中的条目，右侧是控制是否显示列表框的按钮。列表框则位于头部下方。考虑到控件的界面要统一，因此没有让列表框成为下拉列表框的一个数据成员，而是又重新实现部分代码。虽然效率高些，但代码量就大了。因此，读者可以使用私有继承列表框的方式来实现下拉列表框。这样，下拉列表框的继承关系就变成图7-13右所示了。这是一个多继承的关系，下拉列表框类私有继承列表框类，这样就可以重用列表框的代码。如果读者对C++很熟悉，可以使用这种设计，反之，多继承可能就会成为你的魔鬼。

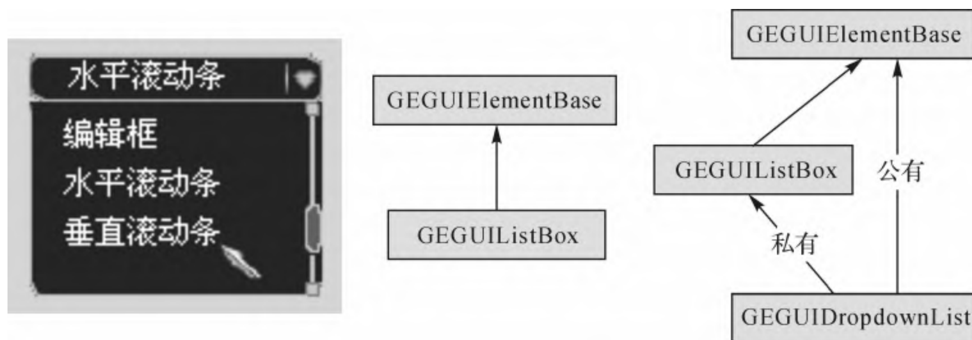



图7-13　下拉列表框图示（左）下拉列表框类继承关系（中）继承关系改进（右）

上面讲述了如何设计并实现9大基础控件，没有想像的那么难，对吗？由于篇幅有限，不可能将所有的实现代码一一分析，大家可以对照附盘中的源代码进行学习。除了这9大控件外，也可以设计满足自己要求的控件，比如说我们在引擎中用一个列表框和两个按钮实现了一个文件搜索控件（GEGUI-FileFinder）。

# 7.3 GUI编辑器

有了一套优秀的GUI控件就可以构建一个优秀的界面。首先，程序员让美工将界面的布局设计出来，并加好尺寸，然后可能会如下编写代码：

创建一个页面

增加一个按钮，位置、大小、文字、透明度……

增加一个列表框，位置、大小、增加内容……

增加一个……

增加一个……

……

终于写完了，100个控件！累……接着编译，错误？原来少写了一个参数……改完编译的问题，运行！什么？这个控件怎么到这里来了？这里本来有个控件的啊……一堆问题，你陷入了困境。要是美工在设计界面的时候能够直接生成代码就好了……

是啊，要是美工可以在设计的同时生成代码该是多好的一件事啊！没错，这就是GUI编辑器的作用！美工可以直接使用GUI编辑器设计界面，并同时完成界面脚本的输出。有了脚本程序员要做的唯一的一件事可能就像下面的伪代码一样轻松：

读取一个界面脚本，返回一个界面指针。

调用该指针的渲染函数。OK！

是的，程序员只需要读取美工利用GUI编辑器生成的脚本就OK了。

既然GUI编辑器是那么的方便，现在开始它的设计之旅吧！图7-14所示为一个GUI编辑器。

图7-14　GUI编辑器

首先要做的是将每个控件都脚本化。之前已经介绍过如何将粒子系统脚本化，其实它们是相通的。如果大家写过HTML的话，这里就稍微简单一点了。必须先定下使用什么脚本语言，其次定下每个控件的脚本格式。

之前在粒子系统的脚本化中，我们自己定义了一种脚本语言，这里为了有个好的延续性，继续使用自己的脚本语言。其实，这种脚本有些类似与HTML和XML，但它是静态脚本，不能加入事件的机制。下面将以文字标签和按钮为例讲解如何脚本化控件。

图7-15左是图7-15右文字标签的脚本。可以看见，每个关键词都是使用<XX>的形式，这样有利于阅读。控件脚本以<element_begin>表示开头，紧接着<Label>表示控件类型，这里<Label>表示文字标签，如果是<ListBox>则表示列表框。控件类型的不同，会导致脚本的解析策略不同。在标明控件类型后就是控件所有关键属性的解析了。对于文字框来说，有名字<name>、大小<size>、与父控件的相对平移<offset>、可见性<visible>、文字颜色<color>、字体<font>、标签对齐方式<align>、文字<text>、层次<level>共9个属性。在每个属性后面跟着属性值，以字体<font>为例，后面首先跟着“宋体”表示字体，再后面的“10”表示字号是10号；再比如对齐方式<align>后面的AF_VMD_LEFT表示左对齐并垂直局中。最后，以<element_end>结尾表示一个控件脚本的结束。

```
<element_begin>
    <Label>
        <size>    200    54
        <offset>  148    195
        <visible> 1
        <color>   1      1      0      1
        <font>    宋体   10
        <align>   AF_VMD_LEFT
        <text>    GE Game Engine ^o^
        <level>   1
<element_end>
```



图7-15　文字标签对应的脚本文件

完成了文字标签的脚本格式定义后，还需要定义一个函数使控件能够存入脚本文件。这个函数比较简单，只需要按照格式去写脚本文件就好了，读者可以自己去实现或是阅读引擎中的源代码。

最后，还需要编写文字标签脚本解析的函数。如下所示伪代码说明了整个流程。

```
9大属性，每个属性都给出一个默认值
    读取第一个属性"<XX>"
    While（true）{
        Switch（<XX>）
        {
            如果是<element_end>则break
            按照不同的属性，修改相应的值
            继续读属性
        }
    }
    创建一个新的文字标签控件，并返回指针
```

OK，文字标签的脚本化工作算是完成了。

文字标签是一个最基本的控件，而如同按钮、列表框之类的则是有几个基本控件组成的。下面我们将通过选择框脚本化的分析来说明如何实现复杂控件的脚本化。

选择框是一种按钮（如图7-16所示），因此它由5部分组成：普通状态下的图片框、鼠 标移上去后的提示图片框、被点下后的图片框、不可点状态图片框以及一个说明文本框。因此，它的脚本化分为两个部分，一是它本身的脚本化，二是它的5个子控件的脚本化。

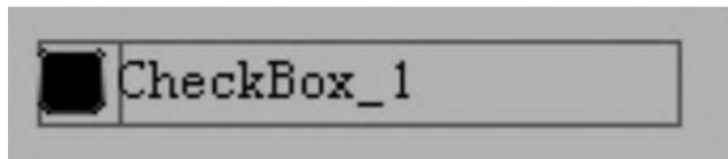



图7-16　GUI编程器的选择框

**例程7-4　GUI编辑器中的选择框（上）及选择框的脚本化（下）**

```
<element_begin>
<CheckBox>
    <name>    CK1
    <size>    150    20
    <offset>    64    201
    <visible>    1
    <enable>    1
    <level>    1
    <element_begin>
        <PicLabel>
            <size>    16    16
            <offset>    0    2
            <level>    0
```

```
            <visible>    1
            <diffuse>    1    1    1    1
            <texture>    Texture/GUI-OGRE.tga
             <uvMap>     0.16015625     0.42578125     0.22265625
0.48828125
    <element_end>
    <element_begin>
        <PicLabel>
            <NULL>
    <element_end>
    <element_begin>
        <PicLabel>
            <size>    16    16
            <offset>    0    2
            <level>    0
            <visible>    1
            <diffuse>    1    1    1    1
            <texture>    Texture/GUI-OGRE.tga
              <uvMap>     0.16015625     0.5703125     0.22265625
0.6328125
    <element_end>
    <element_begin>
        <PicLabel>
            <NULL>
    <element_end>
    <element_begin>
        <Label>
            <size>    131    20
            <offset>    19    0
            <visible>    1
            <color>    0    0    0    1
            <font>    宋体    10
            <align>    AF_VMD_LEFT
            <text>    CheckBox_1
```

```
            <level>    0
    <element_end>
<element_end>
```

例程7-4中的脚本是图中选择框的脚本。脚本也是以<element_begin>表示开始，以<element_end>表示结束。在<element_begin>后紧跟表示控件类型的<CheckBox>，然后是一些关键属性的值，比如名字、大小、位置、层次等等。这些属性其实每种控件都需要保存。在自身的属性记录完毕后，就是其子控件的脚本了。先是依次记录了4个图片框的脚本，再记录了说明文本框的脚本。可以发现，每个子控件脚本都是内嵌到父控件内的，与独立控件的脚本没有任何差异。另外仔细观察可以发现第二和第四个图片框子控件的脚本中只有<NULL>，这表示两个子控件为空，因为没有设计选择框的鼠标覆盖提示图片框和不可点状态图片框。

控件的脚本化基本就这两种形式：独立控件，比如文字标签、图片框；有子控件的控件，比如按钮、列表框等。

在解决了控件的脚本化后，还要在每个控件内实现在编辑条件下的Up-date函数，这些函数会响应鼠标编辑控件大小、位置的操作。

然后，还要设计各种编辑界面及功能。界面设计的首要原则是能够满足所有关键属性编辑。其次，界面的设计要简单明了。图7-17是GUI编辑器中的文字标签控件的编辑页面和文本编辑框的编辑界面。

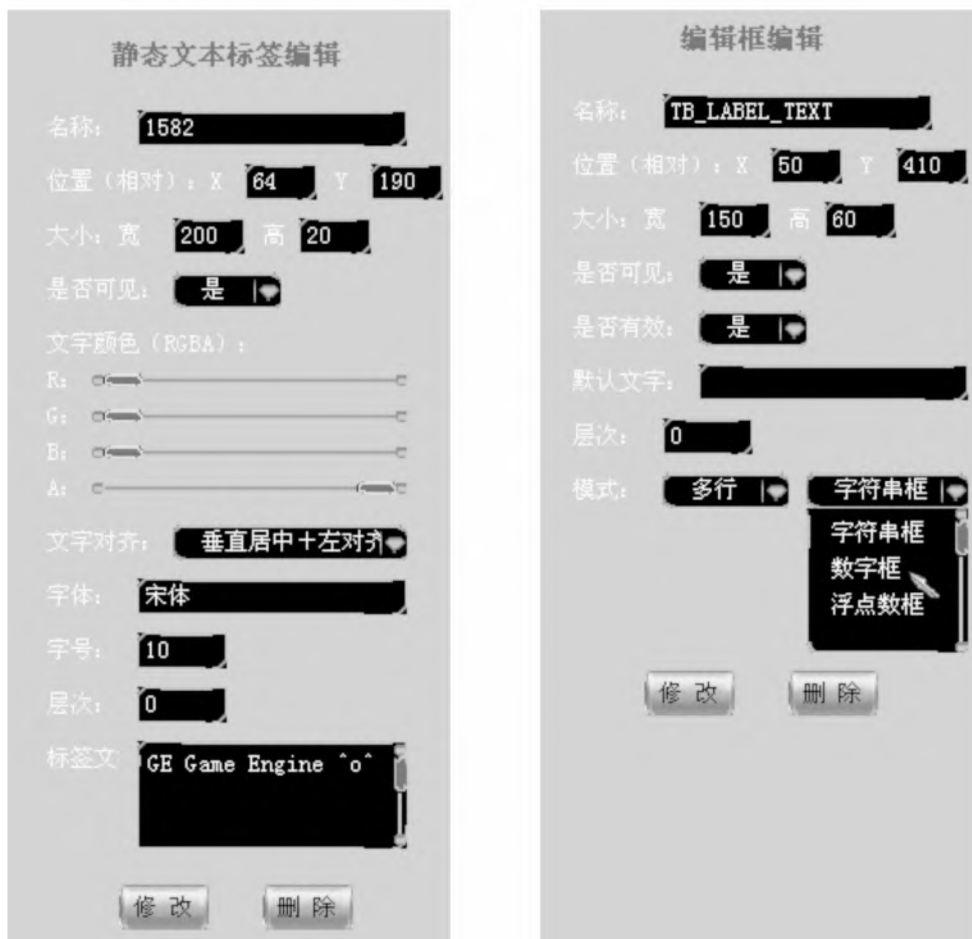



图7-17 文字标签编辑界面（左）、文本编辑框编辑界面（右）

在完成界面设计后，相应的功能编写比较简单，大家可以通过随书源代码进行学习。

在完成控件脚本化和每个控件的编辑界面实现后，GUI编辑器算是完成了80%，可以使用了，接着要做的是实现使用鼠标和键盘修改。比如点中一个控件，可以按Del键删除；可以用Ctrl+C复制一个控件，Ctrl+V粘贴；再比如可以使用鼠标结合Ctrl键框选。这些功能不是那么的必要，但有了这些功能会显得专业，用户使用也会得心应手。不过这些功能的实现要稍微复杂一些，读者可以自己去试试。

# 第8章　输入模块

玩家和游戏的交互是通过输入设备完成的。传统的输入设备包括鼠标、键盘、游戏杆和游戏手柄，而一些高级的互动游戏的输入设备还包括方向盘（舵轮）、踏板、6-DOF（自由度）空间定位球、游戏枪、“Cybersex”服装等。

作为一个游戏引擎，你必须对基本的输入设备进行封装，这里就包括鼠标、键盘和游戏手柄。本章将会介绍如何设计一个合理的输入模块：

- 关于DirectInput。
- 输入模块的基本框架。
- 鼠标输入。
- 键盘输入。
- 手柄输入。

## 8.1　关于DirectInput

DirectInput是一个输入设备的应用程序接口（API），其中就包括鼠标、键盘、游戏杆及其他游戏控制器如力回馈（输入/输出）设备，如图8-1所示。

图8-1　游戏专用输入设备

和Direct3D、DirectDraw一样，DirectInput也是一个奇迹。如果没有Di-rectInput，这会儿读者肯定正在电话前和世界各地的输入设备制造商企求驱动程序呢（DOS、Win16、Win32等，每样都要一个），那绝对是噩梦般的一天！幸好，DirectInput替用户将所有的问题一扫而光！

DirectInput是一个不依赖硬件的虚拟输入系统，它允许硬件制造商开发应用与统一接口下的传统的和非传统的输入设备。因此不需要拥有所有的输入设备的驱动程序。用户只要同DirectInput打交道，而DirectInput会把用户的代码翻译成输入设备能够理解的代码。

图8-2给出了DirectInput同硬件驱动程序以及实际输入设备之间的关系。

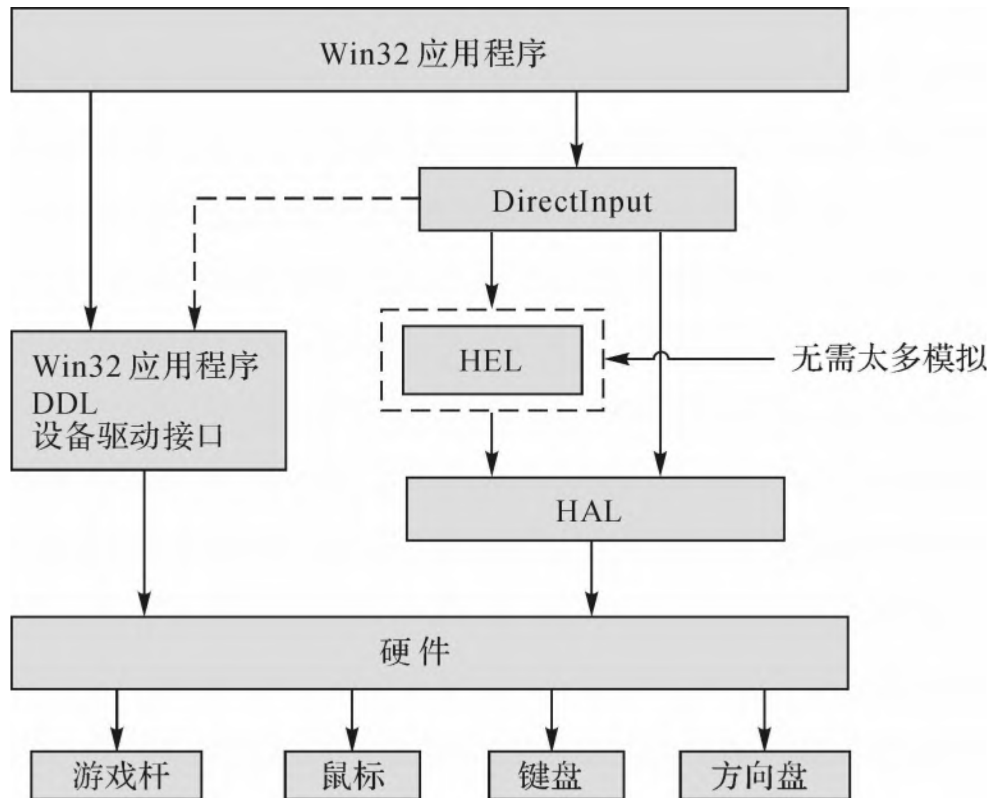



图8-2　DirectInput系统级示意图

正如你看到的那样，你总是被HAL（Hardware Abstraction Layer，硬件抽象层）所隔离，需要用HEL（Hardware Emulation Layer，硬件仿真层）进行仿真的东西不多。

在知道了DirectInput的架构后，我们再看看DirectInput的能力有多少：

- 除了支持Microsoft Win32 API不支持的设备服务外，DirectInput还能以直接访问硬件驱动的方式提供比Microsoft Windows消息更快的访问方式。
- 即便应用程序处于后台，DirectInput也可以应用程序获取输入设备数据。同样，它对任意类型的输入设备都提供完全支持，

其中包括力回馈控制器。

- 通过Action映射，应用程序不需要知道具体使用的是什么类型的设备就可以获取输入数据。

可以看出，DirectInput扩充了服务，并改良了性能，这对于游戏、模拟以及实时会话来说真是太棒了！

但同时也要注意一个事实，即对于使用键盘作文本输入或鼠标导航的应用程序来说，DirectInput不会提供更多优势。

总的来说，DirectInput的封装是游戏引擎必需要做的工作，同时，它也极大地丰富了游戏的操作手段和可玩性。

# 8.2　输入模块的基本框架

无论最后是使用鼠标、键盘、摇杆或是其他设备，它们都是DirectInput组件，也就是说，它们在DirectInput中都有统一的接口：

- IDirectInput8：这是启动DirectInput必须创建的主COM对象。幸运的是，DirectInput8Create()封装好了这些COM的内容。一旦创建了IDirect-Input8接口，就可以通过它来进行调用，以设置DirectInput的属性，并创建或获得想使用的任何输入设备。
- IDirectInputDevice8：这个接口是从主IDirectInput8接口创建而得，是和设备（如鼠标、键盘、游戏杆等IDirectInput8设备）通信的渠道。此外，还允许使用轮询设备（Polled Device）；有一些游戏杆设备需要被轮询。

正因为所有支持DirectInput的设备都有那么多的共性，所以在具体实现每一种设备前最好构造一个公共基类，该基类抽象了所有设备都要做的一些操作并封装了可以被隐藏的细节。这一节的主要内容就是带读者一起设计这个公共基类。

这里需要设计的是一个输入模块的基类，它抽象了DirectInput设备的公共数据成员和公共操作。关于公共数据成员，在上面已经介绍了；下面来分析一下使用DirectInput设备都有哪些步骤要做：

（1）创建DirectInput对象。使用该对象的方法来列举设备，并创建Di-

rectInput设备对象。

（2）列举设备。如果需要使用的仅有鼠标或键盘的话，该步可以跳过。如果需要确定系统中是否还有其他输入设备，就需要使用DirectInput进行列举。当DirectInput找到一个用户所设置的条件的设备时，可以让用户检查该设备的能力，同样也可以得到一个设备唯一标识，通过标识就可以创建该设备的设备对象。

（3）创建所要使用设备的DirectInputDevice对象。这里需要用到上一步中的标识符。对于系统鼠标或键盘来说，可以使用标准的全局唯一标识。

（4）设置设备有关信息。对每一个设备来说，首先需要设置它的协作级别，也就是说该设备以什么样的方式同系统或其他应用程序共享；然后是用来标识数据包中设备对象的数据格式，如按钮或轴（axes）；如果需要获取缓冲数据的话——也就是说，宁愿基于事件而不是状态——那么还需要设置缓冲区大小。另外，在该步骤中，可

以获取设备的信息并因此可以设计应用程序的行为。最后还可以设置某些属性值，如游戏杆轴的返回值的范围。

（5）获得设备。该步告诉DirectInput设备已经准备好接收数据了。

（6）获取数据。每隔一定时间后，如一个消息循环或绘制循环后，可以得到每个设备的当前状态或者自从上次检测以来的事件记录。如果需要的话，只要发生事件，DirectInput就可以通知用户。

（7）根据数据进行相应操作。应用程序可以响应按钮、轴的状态，或者响应按键按下、释放时产生的事件。

（8）关闭DirectInput。退出前，应用程序应该不再需要所有设备了，记得释放它们，然后再释放DirectInput对象。

以上不是实现DirectInput的唯一途径，为了利用今后各式各样的输入设备，可以向用户提供简单化的配置，而可以使用操作映射（Action Mapping）。关于操作映射的细节，可以翻阅相关书籍了解。

有了一定概念后，就可以设计输入模块的公共基类的接口了。

首先是数据成员，输入设备的抽象都有两个共同的数据成员：

- LPDIRECTINPUT8 m_pDI; //<声明DirectInput对象指针
- LPDIRECTINPUTDEVICE8 m_pDevice; //<声明输入装置对象指针

在函数接口方面，需要给出如何创建设备、取得设备、释放设备以及删除设备，因此以下四个函数是必不可少的：

- boolOnCreateDevice （ HWND hwnd ， GUID guid ，struct_DIDATAFORMAT const*format，intsize）； //<创建鼠标键盘等输入设备
- HRESULT FreeDevice(); //<删除输入设备
- HRESULTAcquireDevice(); //<取得输入设备
- HRESULTUnacquireDevice(); //<释放输入设备

下面，先解释如何创建DirectInput设备，也就是函数OnCreateDevice是怎么实现的。

（1）通过调用DirectInput8Create()创建IDirectInput8接口。返回值是IDirectInput8接口。

（2）（可选）查询设备的GUID。在这步中，读者将从DirectInput查询属于鼠标、键盘、游戏杆或其他通用设备。这通过回调函数枚举实现。一般而言，用户请求DirectInput对某一类型/子类型的所有设备进行枚举。Direct-Input通过回调过滤它们，然后就可以建立一个GUID数据。很麻烦是吗？幸运的是这只是需要对游戏杆之类设备做的事情，因为一般情况下用户可以使用通用鼠标或键盘，它们有常备的GUI。

（3）对你想在游戏中使用的每一个设备，创建时必须调用CreateDevice()传递一个GUID。CreateDevice()时IDirectInput8的接口函数，所以在进行此次调用之前，必须先获得IDirectInput8接口；如果用户不知道想创建的设备的GUID，这一步在第2步之后。键盘和鼠标各自对应有内建的GUID。GUID_SysKeyboard：这是全局的，总是代表主键盘设备的GUID。GUID_SysMouse：这是全局的，总是代表主鼠标设备的GUID。

（4）一旦枚举一个设备，就必须对它设置协调层级。这由IDirectInput-Device8::Set- CooperativeLevel()来完成。

（5）从IDirectInputDevice8接口调用SetDataFormat()函数，设置每一个设备的数据格式。这在实际应用中有点费劲，但在概念上还好理解。数据格式是程序员想为将要格式化的每个设备事件分配一个什么样的数据包。这就是DirectInput的好处之一！它给了程序员灵活性。谢天谢地，程序员可以使用一些全局预定义好了的相当智能化的数据结构，所以程序员不必要自己来设置一个。

（6）用IDirectInputDevice8::SetProperty()设置你想要的任何设备的性能。这是对设备描述表敏感的，也就是说，一些设备有的特征另外一些设备就可能没有。同时，因为在一个设备上认可的任何东西都要通过调用Set-Property()实现，所以程序员必须知道想要设置什么。

（7）通过调用IDirectInputDevice8::Acquire()获得每一个设备。这只是用户的设备同应用程序关联，并告诉DirectInput过后你要从这些设备中获取数据。

UnacquireDevice() 函数是通过调用IDirectInputDevice8::Unacquire()实现的。它将用户之前关联的设备脱离关联。

而FreeDevice()函数则是将创建的IDirectInput和IDirectInputDevice8指针删除。

除了设备相关函数之外，我们还需要定义2个重要的方法，判断对应键是否被按下或是松开。可能读者会说，这还不简单，DirectInput

几乎全部都处理了啊！首先定义一个按键状态数组，然后只要每一帧都调用DirectInput设备接口的GetDeviceState函数，并通过查询函数返回后的按键状态数组不就行了吗？是的，这种说法没错，但还少考虑了一种应用的情况。假如想做一款格斗游戏，当玩家按下键盘的“J”后会发生拳击动作。但是，我们不希望用户可以通过常按“J”键而发生连续拳击事件，也就是说，规定从玩家按下“J”键拳击到松开“J”键为一次拳击动作的全序列。如果用户按下“J”键后没有松开则不会发生下一次的拳击事件。这是一个非常普遍的逻辑，但如果使用未丰富的DirectInput原始方法实现则会比较复杂，它的实现可能如下：

```
voidAction{
        static boolbeginAction=false;
        if（!beginAction&&玩家按下“J”键）{
            beginAction=true;
            出拳;
        }
        else if（beginAction&&“J”键未被按下）
            beginAction=false;
    }
```

但是，如果能知道某时按下动作是不是一个全序列动作的开始该有多好呢。也就是说，当按下“J”键的时刻系统告诉我们“J”被按下了（发送消息M），而在按住“J”的时间段里系统告诉我们状态没有发生任何改变，但一旦玩家松开“J”键则立刻报告按键状态改变。于是，就只用在每次M消息发出的时候出拳就好了，非常简单。

有了这个概念后，开始设计按钮状态（按下）检测函数：

```
boolGEInput::IsKeyDown(constBYTE storage[], constBYTE oldStorage[], int
                                                    key,
                                                            bool
lastSensitive=
                                                    false);
```

函数并不复杂，其中的四个参数分别代表意义如下：

- const BYTE storage[]：存放最新时刻的按键状况的数组。
- const BYTE oldStorage[]：存放上一次更新时的按键状况的数组。
- int key：要检测的是什么键。
- bool lastSensitive：按键状态检测是否和上次按键状况敏感。

这里，需要着重解释的是bool变量lastSensitive。当它为false时，表示本次状态查询的结果只和当前的按键状态有关；当它为true时，本次的状态查询结果还和上一帧时的按键状态有关，比方说，如果本帧查询时按键是按下状态，但是上帧也是按下状态，也就说从上帧到本帧并没有输入上的改变，所以输出false。

于是，上述的拳击代码可以更改如下：

```
voidAction{
        if（敏感模式下“J”被按下）
          出拳；
    }
```

是不是简单了很多？在随书光盘的引擎中，输入模块的每个设备抽象都支持了这种操作，即将敏感状态检测和非敏感状态检测区分。

按钮的松开状态检测函数和IsKeyDown函数基本类似：

```
boolGEInput::IsKeyUp(const   BYTE   storage[   ],   const   BYTE
oldStorage[ ],
                                intkey,
                                bool lastSensitive=false);
```

此外，GEInput还需要支持的操作是每帧将当前的按键状态数组保存到旧按键状态数组中去，并更新当前的状态数组。

至此，输入模块的公共基类就完成了。

# 8.3　鼠标输入

鼠标是电脑游戏中最重要的输入设备之一，DirectInput也直接支持了鼠标设备。在本节中，主要介绍如何在类GEInput上构建鼠标输入类。

鼠标有两个输入功能信息：一是位置信息，二是按键消息。其中，鼠标的位置信息包括鼠标指针的位置信息以及滚轮的信息。此外，在设计的时候还要注意以下三点：

（1）DirectInput返回的是鼠标移动信息，而不是鼠标的位置信息。因此，需要保存初始鼠标位置，并在每次更新时根据上帧的鼠标移动信息更新鼠标位置。在这个过程中，还需要注意鼠标的位置不能超过屏幕范围，也就是说鼠标的移动范围和屏幕的分辨率或是窗口的大小信息有关。

（2）游戏中通常需要的是鼠标的相对位置信息，也就是说，需要的是相对与游戏窗口左上角的坐标信息。但是，通过DirectInput得到

是却是鼠标的绝对坐标，因此需要提供接口将绝对坐标转变成游戏窗口相对坐标。

（3）在游戏中，通常会屏蔽Windows对鼠标的显示，并在鼠标位置上应用自己的纹理文件。

根据上述要求，可以定义以下数据成员

```
classGEMouse{
    protected:
        Point2D<int>            m_Pos;            //鼠标位置
        Point2D<int>            m_ClinetPos;         //鼠标在客户
区的相对位置
        Bound                   m_Bound;          //鼠标移动范围
        DIMOUSESTATE2     m_State;              //鼠标状态结构
        DIMOUSESTATE2     m_OldState;         //鼠标上次状态结构
    }
```

在上述结构中，类型Bound是一个结构体，它包括表示屏幕范围的上、下、左、右四个值。

在定义了鼠标类的数据成员后，它需要哪些方法呢？

- bool GEMouse::OnCreateMouse（HWND hwnd，int left=LEFT，int top=TOP，int right=RIGHT，int bottom=BOT TOM）；鼠标设备创建：该函数会调用GEInput中的设备创建函数，并同时初始化鼠标的移动范围
- bool GEMouse::Update（HWND hwnd）；鼠标信息的更新：该方法每帧都要被调用，它首先将当前的状态数据（m_State）存入旧状态（m_OldState）中，然后更新鼠标信息到当前状态结构中，并更新鼠标位置信息，最后将鼠标的全局坐标转变成相对

坐标。这里，需要顺便一提的是如何将全局鼠标坐标转变为局部鼠标坐标。读者可以通过调用下面的Windows API来完成：Bool ScreenToClient（HWND hwnd，LPPOINT lpPoint）也许有些读者觉得太简单了，而作者只是希望每个人都不要在这种事情上浪费时间而已。

- bool GEMouse::IsMouseKeyDown（int type，bool lastSensitive=false）; bool GEMouse::IsMouseKeyUp（int type，bool lastSensitive=false）；重载IsKeyDown和IsKeyUp函数。在类GEInput中，这两个函数的参数比较多，每次查询的时候还需要指定状态数组的位置，这在客户区调用的时候是不合适的。因此，我们需要重载这两个函数，重载后的函数如下：bool GEMouse::IsKeyDown（MouseKeyType key，bool lastSensitive=false）; bool GEMouse::IsKeyUp（MouseKeyType key，bool lastSensitive=false）；也就是说，函数的参数中去掉了状态数组，但是使用的方法还是如上节说的，需要区分是否和上帧按键比较。此外，函数中的key参数表示按键类型。
- int GEMouse::GetScroll(); Point2D<int>GEMouse::GetPos(); int GEMouse::GetX(); int GEMouse::GetY()；得到表示鼠标位置信息的X、Y坐标以及上帧的滚轮改变量。这里返回的鼠标位置是相对于客户端窗口左上角的坐标。
- void GEMouse::SetPos（int x，int y）；设置鼠标位置。这个接口也许不太用得到，但当需要强制移动鼠标到某个位置时却很有用。比如说，某个游戏中需要捕捉特殊点，但是玩家又不可能真的每次都那么准确能捕捉到这个特定点，于是客户程

序员可以设置当玩家的鼠标移动到此特殊点周围时，自动定位鼠标到那个特殊点。这样，玩家会很感激你！

最后要指明的一点是：几乎在每个PC游戏中都会用到鼠标，而且是只用到一个鼠标。于是，我们可以将鼠标类设计成单例模式（Singleton），减少客户程序员的工作量和出错机会。

OK，鼠标类的函数接口也讲完了！希望你现在说，原来设计鼠标类是那么简单的一件事！

# 8.4 键盘输入

有了实现鼠标类的经验后，键盘类的实现就更加简单了。

如下所示是引擎中键盘类GEKeyboard的数据成员：

```
BYTE    m_KeyState[KEYBOARDBUFFERSIZE];         //<键盘状态结构
BYTE    m_OldKeyState[KEYBOARDBUFFERSIZE];     //<键盘上帧的状态结构
```

代码中，KEYBOARDBUFFERSIZE是一个宏类型，它表示键盘数组的大小，也就是键盘最多有几个键。在宏定义中，我们将此值设为256。针对市面上的键盘，应该是足够大了。

键盘的函数接口也比鼠标要简单。主要是以下几个函数：

- bool GEKeyboard::OnCreateKeyboard（HWND hwnd）；创建键盘输入设备，该函数会调用GEInput中的设备创建函数。
- bool GEKeyboard::IsKeyDown（int type，bool lastSensitive=false）；判断键盘对应键是否被按下，重载类

GEInput中的IsKeyDown()方法。

- bool GEKeyboard::IsKeyUp（int type，bool lastSensitive=false）；判断键盘对应键是否被松开，重载类GEInput中的IsKeyUp()方法。
- bool GEKeyboard::Update（HWND hwnd）；更新键盘状态信息，该方法每帧都要被调用，它首先将当前的按键状态数据（m_KeyState）存入旧状态（m_OldKeyState）中，然后更新键盘的按键信息到当前状态结构中。

同时，和鼠标一样，一个计算机一般都只用一个键盘，为了方便客户程序员，我们也提供了键盘类的Singleton模式实现。

在上面的键盘按键检测函数中，参数int type表示键盘的按键类型。对于DirectInput来说，键盘的每一个键都对应一个宏，比如A键对应宏DIK_A，而空格键对应DIK_SPACE。每个键的宏都是以DIK_（DirectInput Key的缩写）开头，并且定义在DINPU T.H中，大家可以翻阅DirectX SDK看到完整版本。

# 8.5 游戏杆输入

游戏杆是DirectInput设备中最复杂的。游戏杆（JoyStick）这个词实际上包括除鼠标和键盘外的所有可能的设备。当然，手柄也是一种游戏杆设备，而且是最常见的游戏杆设备。使用游戏杆设备和使用鼠标设备从本质上来看并没有多大区别，比如它的使用由以下几步完成：

- 用EnumDevices()枚举设备，该步骤是新增的。
- 用CreateDevice()创建设备。

- 用SetCooperativeLevel()设置协作等级。
- 用SetDateFormat()设置数据格式。
- 用SetProperties()设置游戏杆范围、死区和其他性能，这一步在鼠 标和键盘中是没有的。
- 用Acquire()获取游戏杆。
- 用Poll()函数查询游戏杆。只有游戏杆设备需要轮循，它为了保证在调用GetDevicesState()的时候，没有中断驱动程序的游戏杆具有有效数据。
- 用GetDeviceState()读取游戏杆的状态。

所以，游戏杆设备可以重用我们在前面设计的GEInput类，也就是说游戏杆类（在我们引擎是类GEJoystick）可以从类GEInput继承。和键盘、鼠 标一样，手柄类新增数据成员要求能够存储手柄的按键状态。在DirectInput中，已经定义了一个满足我们要求的结构——DIJOYSTATE2，因此，类GE-Joystick的新增数据成员如下：

```
DIJOYSTATE2         m_State;            //手柄本次获取的按键状态
DIJOYSTATE2         m_OldState;         //手柄上次获取的按键状态
```

在类GEInput中，已定义了函数OnCreateDevice()，并且在鼠 标类和键盘类中都使用了该方法，但是因为手柄设备的创建和鼠 标、键盘创建的方法不同（在上面已经提到），所以在类GEJoystick中需要对此函数进行重载。新函数中绝大部分代码和GEInput::OnCreateDevice()相似，要着重解释的是以下三个新步骤。

（1）枚举设备

因为在传统PC机上都有且只用一个键盘和鼠 标，所以不需要对鼠 标、键盘进行枚举，而是默认有且只有一个设备存在。但是对于游戏

杆就不能这样了，因为可能有一个游戏杆、也可能有多个游戏杆，甚至没有游戏杆。另一个原因是，用户对插入的游戏杆设备的类型一无所知，而即使已经知道它的类型，用户还是需要准确知道它们中一个或是多个设备的GUID。所以无论如何，都需要扫描它们，因为用户需要GUID来调用CreateDevice()函数创建相应设备。

在DirectInput中，使用函数IDIRECTINPU T8::EnumDevices()来实现设备的枚举。它的原型如下：

```
HRESULT EnumDevices(
    DWORD          dwDevType;          //设备类型
    LPDIENUMCALLBACK lpCallback;   //回调函数指针
    LPVOID          pvRef;           //32位值，用于数据回传
    DWORD          dwFlags           //搜索的类型
);
```

EnumDevices()函数的第一个参数表示要枚举的设备类型，因为是枚举手柄，所以使用DirectInput中对手柄的宏定义一DIDEVTYPE_JOYSTICK。函数的第二个参数是一个指向回调函数的指针，DirectInput将用它调用它发现的每一个设备。这个回调函数必须遵循一定的格式编写，下面是格式的一个样本：

```
BOOL CALLBACK EnumDevsCallback(
    LPDIDEVICEINSTANCE lpddi;
    LPVOID data);
```

因为回调函数必须是个全局函数，所以需要在类中设计一个静态函数（EnumDeviceCallback()）作为回调函数，它的功能是返回枚举到的设备的GUID。

第四个参数是搜寻的类型，因为我们只对已连接的设备感兴趣，所以使用宏DIEDFL_AT TACHEDONLY。

EnumDevices()的工作方式是：它有一个内部循环，为它找到的每一个设备调用回调函数。因此，如果一台计算机上正连接着许多设备，则用户的回调函数就会被调用多次。

（2）游戏杆范围

游戏杆是模拟设备，轴的运动范围是有限的。换句话说，当用户获取游戏杆的位置时，它返回了X=100，Y=300，这是什么意思呢？这代表轴在什么位置呢？用户是无从知道的。因为用户没有一个参考值。所以，需要设置想读取的任何模拟轴的范围。例如，用户可以决定把X、Y轴分别设成－1000~1000和－2000~2000。在我们的引擎中，将这两个范围都设成了－128~128。这是由SetProperty()函数来实现的。

但这一个函数还是不够的，因为不知道设备上有几个轴，也许有的手柄有1个摇杆，而有的手柄有两个摇杆。因此，还需要枚举摇杆。与上面说到的枚举设备类似，这里也需要用到回调函数。

DirectInput使用EnumObjects()实现摇杆的枚举。它的原型如下：

```
HRESULT EnumObjects(
    LPDIENUMDEVICEOBJECTSCALLBACK lpCallback,    //回调函数
    LPVOID pvRef,                                //32位值，用于数据回
传
    DWORD dwFlags                                //枚举类型
);
```

类GEJoystick中，定义了静态函数EnumObjectsCallback()作为回调函数。回调函数内部负责限定轴的范围。而EnumObjects()函数的第三个参数设成DIDFT_ALL，表示所有对象。

（3）游戏杆盲区

摇杆是个向上的、有体积的杆子。用户希望它在偏离中心后能够向他发送数据。但是，用户希不希望摇杆在轻微偏离中心时也发送数据呢？比如说，只是把手放在摇杆上，由于手和摇杆的接触导致摇杆轻微偏离中心；或是摇杆被垂直放置，因为重力作用，摇杆轻微偏离中心。显然，这两张情况下用户都不希望摇杆传送数据。于是，我们可以设置一个范围，摇杆在这个范围内都属于自然状态，不发送数据。我们将这个范围称为盲区。

盲区的设置很简单，不过要注意的一点是盲区的单位是10000，也就是说，如果将10%的摆动设为盲区的话，就需要设置盲区为10000 ×10%=1000。

除了游戏杆设备的创建以外，其他的操作都和鼠标、键盘差不多，无非是一些Get操作和判断按键状态操作，读者可以参考源代码进行学习。

# 8.6 输入模块的两个例子

为了更直观地说明判断按键状态的函数中敏感和非敏感的区别，我们创建了一个例程DITest.exe。例程中你可以使用鼠标、键盘和手柄测试每一个按键以及摇杆，并区分敏感按键模式和非敏感按键模式（图8-3）。此外，为了让大家能更好地感受摇杆的威力，引擎还创建

了例程JOYTEST.exe，在例程中，你可以使用摇杆控制忍者旋转、跑、跳和攻击（如图8-4所示），并感受摇杆和鼠标、键盘的区别。

图8-3　敏感按键模式和非敏感按键模式的体会例程

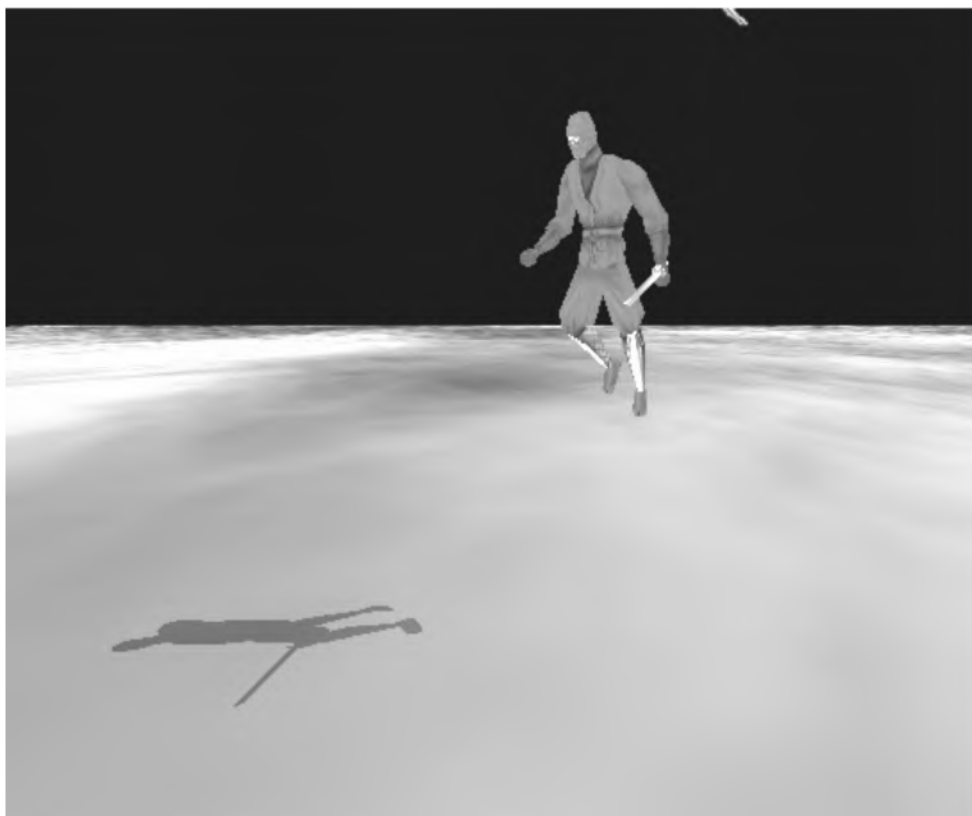


图8-4　摇杆（手柄）体验例程

# 第9章　网络模块的设计与实现

网络模块是网络游戏引擎与传统单机游戏引擎的主要区别之一。游戏中的网络模块给游戏玩家提供了一个前所未有的平台，使得在世界各地被地理区域隔离开的玩家，汇聚在一起共同游戏、交流。1996年的一款游戏枟雷神之锤枠，将网络游戏带入大众的视野，同时也成就了id公司在游戏业界的神话地位。而其中的网络对战模块，甚至开创了一个崭新的产业——电子竞技产业。而在今天，网络游戏产业在世界各地正以惊人的速度吸引着越来越多的投资者和玩家投身其中。最具代表性的是韩国，游戏产业已成为其支柱产业。

本章将会从基础网络连接模型的入手，分析各种网络连接模型之间的优劣，并将详细介绍目前在Windows平台下最具代表性的完成端口模型。在了解网络连接模型的基础之上，深入分析休闲网路游戏的架构设计。最后，将会介绍目前网络游戏中常用到的同步方法——导航算法。

## 9.1　Winsock套接字概述

Winsock提供了两种套接字模式和六种套接字I/O模型，实现对套接字上的I/O行为进行控制。套接字模式包括：阻塞和非阻塞两种模式；套接字I/O模型包括阻塞（blocking）、选择（select）、异步选择（WSAAsyncSelect）、事件选择（WSAEventSelect）、重叠（overlapped）和完成端口completionport（完成端口）。

套接字模式决定了在当前套接字上调用Winsock函数后所产生的效果。在阻塞模式下，调用任何一个Winsock API函数，都会有一种共同的效果——程序将阻塞等待该函数的返回。而在非阻塞模式下，处理收发数据或者处理连接的Winsock API调用会立即返回。某些情况下，这些调用会返回WSAEWOULDBLOCK错误，这说明在调用期间函数没能获得足够的资源来完成请求的操作。举例来说，如果在一个非阻塞套接字上调用recv，而当前系统接收缓冲区内没有数据，那么recv调用就会返回WSAEWOULDBLOCK错误。

阻塞模式和非阻塞套接字有各自的优缺点，阻塞套接字非常容易使用，但在需要建立大量连接进行管理的环境下难以胜任；非阻塞套接字在使用上存在一些难度，Winsock提供了几种非常实用的套接字I/O模型供我们使用。

（1）阻塞模型：阻塞模型就是建立在阻塞套接字之上的一种最简单的I/O模型，阻塞模型的优势在于模型简洁，实现简单，对于简单应用和快速原型化的情况下，阻塞模型非常实用。但缺点在于很难将它扩展到大量连接的情况下。

（2）选择模型（select）：选择模型的工作原理是利用select函数对I/O进行管理，select函数在调用后会阻塞一段时间，并监控一组套接字中是否有数据到达某个套接字，或者能否向某个套接字写入数据。使用select模型与非阻塞套接字结合，可以在单线程中实现多个套接字管理。

（3）异步选择模型：异步选择模型提供了以Windows消息为基础的套接字事件通知。只需要调用WSAASyncSelect函数，传入我们感兴趣的套接字和感兴趣的消息集合，接收消息的窗口句柄就可以使用异

步选择模型。异步选择模型的优点是可以在系统开销不大的情况下处理多连接，缺点是，即使应用程序本身不需要窗口，也不得不提供一个额外的窗口用来接收消息，而且在单窗口程序中处理成千上万个套接字事件消息，也可能会成为性能的瓶颈。

（4）事件选择模型：事件选择模型和异步选择模型都是允许应用程序接收一个或多个套接字上的事件通知。差别在于，事件选择模型使用Event对象而不是Windows消息来完成事件通知。该模型与异步选择模型相比更加高效而且不需要窗口环境，但是缺点是它每次只能等待64个事件。

（5）重叠模型：重叠模型可使用事件对象通知，也可以使用完成例程队已经完成的请求加以处理。

# 9.2　完成端口模型

可以把完成端口看成系统维护的一个队列，操作系统把重叠I/O操作完成的事件通知放到该队列里，因为该队列主要是进行“操作完成”事件的通知，所以命名为“完成端口”（Completion Ports）。一个socket被创建后，可以在任何时刻和一个完成端口联系起来。要注意的是，完成端口是Windows采用的一种I/O控制机制，除了套接字句柄之外，还可以接收其他东西，比如文件句柄。在使用完成端口模型之前，首先要创建一个完成端口对象，用它来管理I/O请求，需要调用CreateCompletionPort函数，该函数定义如下：

```
HANDLE CreateIoCompletionPort（ HANDLE FileHandle, HANDLE ExistingComple-tionPort, DWORD CompletionKey, DWORD NumberOfConcurrentThreads）;
```

在介绍该函数的各个参数之前，首先需要了解该函数实际上实现两个截然不同的功能，第一个功能，该函数用来创建一个完成端口对象；第二个功能，该函数将一个完成端口对象和一个句柄绑定起来。一开始使用该函数创建完成端口对象的时候只需要关心一个参数：NumberOfConcurrentThreads，其余的参数都是无关紧要的。NumberOfConcurrentThreads参数定义了一个完成端口上允许同时处于激活状态的线程数量。

一般来说，一个应用程序可以允许同时有多个工作线程来处理完成端口上的通知事件。工作线程的数量依赖于程序的具体需要。但是在理想的情况下，应该为一个CPU创建一个线程。因为在完成端口理想模型中，每个处理器上各运行一个工作线程，可以避免频繁的线程切换。在实际使用中，只需将该参数设置为0，系统就会按照安装的处理器数量设置NumberOfCon-currentThreads。在实际开发的时候，还要考虑这些线程是否经常出现堵塞操作。如果某线程进行堵塞操作，比如Sleep或WaitForSingleObject函数，系统则将其挂起，让别的线程获得运行时间。因此，如果有这样的情况，可以多创建几个线程来尽量利用时间。

当成功创建完成端口对象后，需要将套接字同刚创建的完成端口模型绑定起来，并再次调用CreateIoCompletionPort函数，同时提供前三个参数——FileHanlde、ExistingCompletionPort和CompletionKey，其中FileHandle参数指定要与完成端口进行绑定的I/O对象句柄，ExistingCompletionPort参数是需要进行绑定的完成端口句柄，CompletionKey参数是指要与该套接字句柄关联在一起的单句柄数据（per-handle data），可将它作为指向自定义数据结构的指针。

将套接字与一个完成端口对象绑定之后，就可以重叠递交发送和接收操作，并依赖完成端口对象，接收有关I/O操作的完成情况通知。然后调用GetQueuedCompletionStatus函数，让工作线程等待完成端口上的事件通知。该函数定义如下：

```
BOOL GetQueuedCompletionStatus(HANDLE CompletionPort, LPDWORD umberOf-BytesTransferred, PULONG_PTR lpCompletionKey, LPOVERLAPPED*lpOverlapped, DWORD dwM illiseconds);
```

其中CompletionPort参数对应完成端口句柄，lpNumberOfBytesTrans-ferred参数表示本次I/O操作中实际传输的字节数，lpCompletionKey参数是原先在调用CreateCompletionPort时传入的单句柄数据，lpOverlapped参数用于接收I/O操作的WSAOVERLAPPED结构，因为该参数可以获取该次I/O操作中绑定的数据，所以在实际应用中该参数相当重要，最后一个参数dwMilliseconds用来指明调用者希望在本次等待完成端口上事件通知的最长时间（毫秒），如果将该参数设置为INFINITE，除非等到完成端口上的事件通知，否则本次调用会无休止的等待下去。

现在已经可以建立起完成端口模型的基本框架了，下面使用完成端口模型开发一个回应服务器：

（1）使用CreateIoCompletionPort创建一个完成端口对象。一般情况下，可以将最后一个参数设置为0，指定在每个处理器上最多允许执行一个工作线程。

（2）判断系统内有多少处理器。

（3）根据第（2）步得到的结果，为每个处理器创建一个工作线程。

（4）创建一个监听套接字，并保存accept函数返回的新套接字。

（5）创建需要的单句柄数据数据结构，同时在结构中存入新套接字的句柄。

（6）再次调用CreateIoCompletionPort函数，将在第一步创建的完成端口对象与第四步得到的新套接字绑定起来，并将在第（5）步创建的单句柄数据通过CompletionKey传递给CreateIoCompletionPort。

（7）在新套接字上进行重叠I/O操作，工作线程会为这些I/O请求进行服务，并继续处理以后的I/O请求。

（8）重复步骤（4）~（7）。

**例程9-1　完成端口模型框架**

```
HANDLE completionPort;
WSADATA wsd;
SYSTEM_INFO systemInfo;
SOCKADDR_IN internetAddr;
int internetAddrLen=sizeof(SOCKETADDR_IN);
SOCKET listenSocket;
typedef struct_PER_HANDLE_DATA
{
    SOCKET    Socket;
    SOCKADDR_STORAGE ClientAddr;
    //在这里还可以加入其他和客户端关联的数据
}PER_HANDLE_DATA, *LPPER_HANDLE_DATA;
//初始化W indowsSocket2.2
```

```
StartW insock(MAKEWORD(2, 2), &wsd);
//第一步：创建完成端口
completionPort=CreateIoCompletionPort(
    INVALID_HANDLE_VALUE, NULL, 0, 0);
//第二步：获得系统信息
GetSystemInfo(&systemInfo);
//第三步：创建工作线程，每个CPU对应一个
for(int i=0; i<systemInfo.dwNumberOfProcessors; i++)
{
    HANDLE threadHandle;
            threadHandle=CreateThread(NULL,   0,   WorkerThread,
completionPort, 0, NULL);
    //关闭线程句柄
    CloseHandle(ThreadHandle);
}
//第四步：
//创建监听Socket
listenSocket=WSASocket(AF_INET, SOCK_STREAM, 0, NULL, 0,
                         WSA_FLAG_OVERLAPPED);
internetAddr.sin_family=AF_INET;
internetAddr.sin_addr.s_addr=htonl(INADDR_ANY);
internetAddr.sin_port=htons(8088);
bind(listenSocket, (PSOCKADDR)&internetAddr, internetAddrLen);
listen(listenSocket, 5);
while(true)
{
    PER_HANDLE_DATA*PerHandleData=NULL;
    SOCKADDR_IN remoteAddr;
    SOCKET acceptedSocket;
    acceptedSocket=WSAAccept(Listen, (SOCKADDR*)&remoteAddr,
                            &internetAddrLen);
    //Step 6：初始化单句柄数据
    PerHandleData=(LPPER_HANDLE_DATA)GlobalAlloc(GPTR,
```

```
sizeof(PER_HANDLE_DATA));
    printf("Socketnumber%d connected\n", Accept);
    PerHandleData->Socket=acceptSocket;
              memcpy(&PerHandleData->ClientAddr,    &remoteAddr,
internetAddrLen);
    //Step 7:
    //套接字与完成端口绑定
                   CreateIoCompletionPort((HANDLE)acceptSocket,
CompletionPort,
                                 (DWORD)PerHandleData, 0);
    //Step 8：发出对客户端的I/O请求，等待完成消息
    WSARecv(...);
}
```

工作线程函数：

```
DWORDW INAPIWorkerThreadProc(LPVOID CompletionPortID)
{
    HANDLE CompletionPort=(HANDLE)CompletionPortID;
    DWORD BytesTransferred;
    LPOVERLAPPED Overlapped;
    LPPER_HANDLE_DATA PerHandleData;
    LPPER_IO_DATA PerIoData;
    DWORD SendBytes, RecvBytes;
    DWORD Flags;
    while(TRUE)
    {
        //等待完成端口消息，未收到消息的时候则阻塞线程
        ret=GetQueuedCompletionStatus(CompletionPort,
            &BytesTransferred, (LPDWORD)&PerHandleData,
            (LPOVERLAPPED*)&PerIoData, INFINITE);
        if(BytesTransferred==0&&
            (PerIoData->OperationType==RECV_POSTED||
            PerIoData->OperationType==SEND_POSTED))
        {
            closesocket(PerHandleData->Socket);
```

```
            GlobalFree(PerHandleData);
            GlobalFree(PerIoData);
            continue;
        }
        if(PerIoData->OperationType==RECV_POSTED)
        {
            //Do somethingwith the received data
            //in PerIoData->Buffer
        }
        Flags=0;
                            ZeroMemory(&(PerIoData->Overlapped),
sizeof(OVERLAPPED));
        PerIoData->DataBuf.len=DATA_BUFSIZE;
        PerIoData->DataBuf.buf=PerIoData->Buffer;
        PerIoData->OperationType=RECV_POSTED;
          WSARecv(PerHandleData->Socket, &(PerIoData->DataBuf),
1,
                 &RecvBytes, &Flags, &(PerIoData->Overlapped),
NULL);
    }
}
```

## 9.3　I/O模型的选择与比较

完成端口模型目前是Windows平台上性能和伸缩性方面表现最好的I/O模型，但是完成端口模型在开发和使用方面都存在一定的难度，对于客户端和连接数量较少、吞吐量要求不高的服务器模块，可以选择其他I/O模型。比如对于游戏客户端来说，往往只需要游戏服务器建立单一连接，对于吞吐率方面基本没有要求，而且客户端肯定有Windows窗体，那么可以选择WSAAsyncSelect模型，因为WSAAsyncSelect模型本身的事件通知就是基于Windows消息机制，对于已经存在窗体的应用

程序来说，只需要很简单的操作就可以实现WSAAsyncSelect模型。如果客户端使用了MFC类库，可以直接使用CSocket类，CSocket类使用的就是WSAAsyncSelect模型。对于没有窗体的客户端程序来说（Console程序），可以选择使用WSAEventSe-lect或者重叠I/O模型，也可以获得较高的网络性能。

对于网络游戏服务器，由于需要同时对成千上万的玩家进行服务，则需要使用完成端口模型以获得最好的性能。对于在Native环境下的完成端口模型开发，可以查看Anthony Jones和Jim Ohlund所著的杬Windows网络编程枠一书，其中还有针对服务器性能和伸缩性方面的一些讨论。如果想了解完成端口的底层实现机理，可以查阅Mark E. Russinovich和David A. Solo-mon所著的*Windows Internals* 。

如果服务器端在.Net Framework基础上开发，那么可以直接选择使用.Net Framework提供的Socket类，该类底层使用完成端口模型实现，在接口封装和稳定性方面都有不错的表现。当然，由于Framework在封装完成端口模型时着重考虑的是易用性和稳定性，所以在性能方面和伸缩性方面都有可以挖掘的空间，比如Framework中工作线程使用的是System. Thread中的ThreadPool线程池来进行支持，一方面ThreadPool是static class，导致应用程序中只能有一个线程池；另外ThreadPool只能设置最大线程数量，由系统根据请求状况动态调节，而无法设置固定线程数量，这方面的优化可参考William Kennedy所写的*IOCP Thread Pooling in C* #。

# 9.4　游戏服务器概述

回忆一下平时玩网络休闲游戏的过程：打开客户端，弹出登录对话框，输入用户名和密码，登录成功后转到大厅，然后选择自己想玩的游戏，并进入到游戏房间内，选择一张有空位置的桌子，迫不及待地开始游戏。

现在要自己设计实现一套休闲游戏平台，考虑一下整个流程的背后到底发生了什么：首先是登录部分，输入用户名和密码，服务器肯定需要根据输入的信息来验证我们的身份，这是我们想到的第一部分，验证身份信息。

然后，转入到游戏大厅中，在大厅中可以看到游戏列表。游戏列表信息不应该固化在玩家客户端，因为会为玩家提供很多有趣的游戏（不错的目标），但最好不要在每次加入新游戏时都强迫玩家更新他们的客户端，所以选择从服务器动态获取游戏列表和房间列表信息。这是第二部分，游戏列表和房间信息所做的交互。

接下来要做的是系统最重要的一部分，选中一款游戏，进入到这款游戏的某个房间，房间中有目前想要玩这款游戏的玩家列表，选择好对手之后就开始游戏。这个过程也是系统中最复杂的部分，考虑一下游戏房间这个概念，每个游戏房间内有若干张桌子（一般来说有50~100张），随着运营规模的不断扩大，系统中可能存在数百个游戏房间，每个房间可能有50~100张桌子，每张桌子上有2~10人，每个房间内可能同时存在100~1000人，对于房间内玩家的行为，我们可能需要向房间内的所有其他玩家广播，这部分通讯应该由谁来管理？这就是第三部分，游戏房间内管理。

当选择了一个桌子后，客户端会启动一个游戏进程，接下来就要开始进行游戏。游戏逻辑部分对网络的使用往往是比较简单明了的，

对所有玩家广播，对某个玩家发送游戏指令，接受某个玩家发来的游戏指令等等。这是第四部分，游戏逻辑管理。

刚才提出的四个部分：登录验证、游戏和房间信息、房间内部信息、游戏逻辑管理，基本就构成了游戏平台的前台。这四个模块之间存在一定的逻辑交互，比如登录验证的结果会影响到后续服务，房间内部信息依赖于玩家选择的房间，房间内部信息与游戏逻辑之间存在很强的关联，考虑以下场景：如果在游戏刚刚开始的时候关掉游戏客户端，而房间管理模块还没有收到相关的通知，那如果此刻在房间中选择另外的座位，就发生下面的事情，游戏逻辑管理模块认为游戏已经开始，而我们在游戏开始之后掉线了，但此时房间管理模块在收到游戏开始消息之前先收到了我们更换桌子的请求，它会同意并热心地把我们分配到新的桌子上，之后他会收到游戏开始信息，并在不久之后收到我们掉线的通知消息。房间管理模块此时处于一种非常尴尬的处境，它收到一条消息说A桌的某个玩家掉线了，但实际上他发现那个玩家正在B桌中，如果你认为这种状况仍然是可以解决的，那么考虑一下，如果玩家在B桌中已经开始游戏了怎么办，如果他在B桌上又重演刚才在A桌上表演的技巧，并顺利到了C桌呢？这并不只是费尽心思构想出来的虚假场景，要清醒地认识到网络是无常的，所有能想像到的不利状况最终都会出现。

看来已经可以引入下一节了，下一节会介绍各个模块之间的逻辑交互和物理部署。

# 9.5　游戏平台架构分析

通过上一节的分析了解到，身份验证服务器的结果需要被之后的所有逻辑模块用到，房间管理模块与游戏逻辑管理模块之间有很紧密的联系。那么，图9-1是一种比较简洁的能够满足上述需求的物理架构。

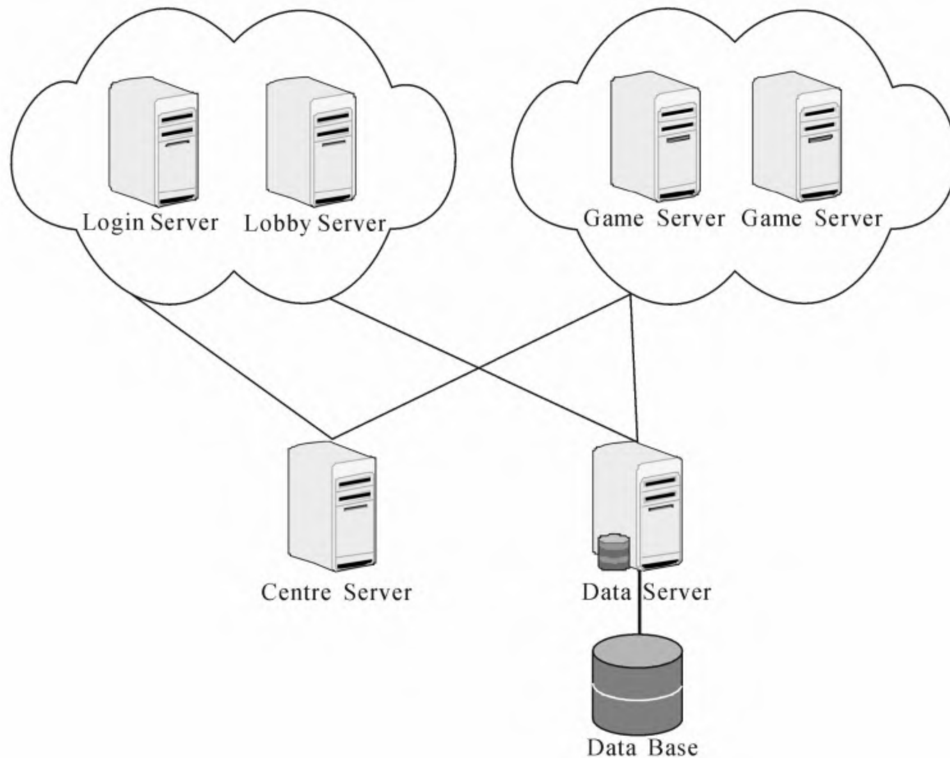



图9-1　网络游戏服务器构架图

服务器端模块由登录服务器（LoginServer）、大厅目录服务器（LobbySer-ver）、中心服务器（CenterSessionServer）、游戏服务器（ConcreteGameServer）和数据中心服务器（DataCenter）组成。

先来看看登录服务器（LoginServer），它相当于逻辑层面上的身份验证模块，客户端在完成登录验证步骤之后，迅速断开与登录服务

器的连接，转到大厅目录服务器。登录服务器将玩家的登录验证信息发送到中央服务器，供其他模块使用。

由于房间管理模块和游戏逻辑模块过于紧密，将他们并在一起称为“游戏服务器”，这样就解决了两个模块之间的同步问题。

登录服务器和大厅目录服务器与客户端之间采用“短连接”模式，即客户端在完成与该模块的交互之后，迅速断开与该模块的连接，可以有效节省服务器资源。当客户端从大厅目录服务器获取到足够的信息之后，也立即断开与大厅目录服务器之间的连接。在一定间隔时间后，客户端再次连接大厅目录服务器，并获取新的信息。这样做可以将服务器的负载压力分散，客户端与大厅目录服务器之间采用的“短连接”模式，可以有效地提高大厅负载人数。

玩家根据大厅目录服务器提供的游戏列表和房间信息，选择某个游戏房间后，客户端会重新开启一个新的进程与具体的游戏服务器进行连接。这一部分在目前流行的休闲游戏平台有两种不同的做法，一种是新开启的游戏进程与游戏服务器建立新的连接，该游戏进程要负责房间内消息和界面管理，还要处理游戏内部的游戏逻辑；另外一种做法是大厅客户端与游戏服务器建立新的连接，房间内的管理由大厅客户端负责，当玩家选定桌子后才开启新的游戏客户端，并且新开启的游戏客户端共享大厅客户端与游戏服务器的套接字，进行游戏逻辑通讯。这两种不同方式的主要区别在于由谁来负责房间内的消息管理，第一种是新开启的客户端负责管理，这样的优点是不同的游戏房间管理模块可以不同，以后扩展新的游戏类型会非常方便，缺点是涉及跨进程窗口从属问题。第二种方式是由大厅负责游戏房间管理，这样无论是界面风格和消息格式游戏都需要统一起来，优点是游戏客户端开发可以简单一些，只需要考虑具体的游戏逻辑实现，缺点是缺乏

灵活性，而且要涉及跨进程套接字共享。这两种设计方式各有利弊，但实质差别并不大，可以自己选择一种实现。

# 9.6　服务器模块剖析

## 9.6.1　登录服务模块

登录模块所负责的职能主要有两方面：

（1）负责验证玩家的登录请求是否合法。

（2）如果玩家成功登录，将玩家的状态信息提交之中心服务器。

登录模块部分还需要考虑一些安全性方面的问题，如果客户端将用户输入的用户名和密码直接提交给登录服务器进行验证，那一旦通讯信息被第三方截获，玩家的密码就完全暴露在第三方手中。当然，可以对通讯报文进行加密，但考虑到效率和成本，可选择的加密方法往往都是对称加密算法（公开密钥算法需要第三方证书系统，加密解密效率也远低于对称加密算法）。客户端也是提供公开下载的，一旦第三方完整窃听了整个通讯过程，那他便可以反编译客户端，追踪出用于加密解密的密钥。所以不能把希望放在报文加密上，而是要从协议设计本身考虑，尽可能的保护系统安全性，保护用户密码的隐秘性。

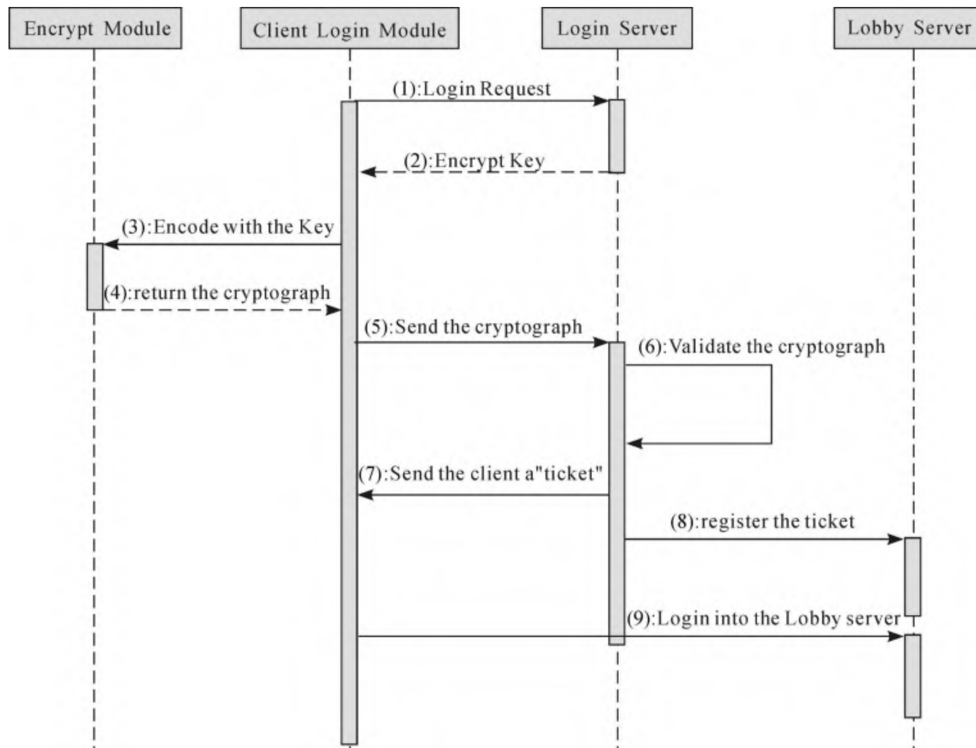



图9-2　登录流程时序图

我们需要验证用户输入的密码是否正确，而又不希望用户的密码直接在网络上传输，就可以使用MD5算法。客户端将用户输入的密码进行MD5加密后提交至服务器，服务器端保存有用户正确密码MD5后的值，经过对比就可得知用户输入的密码是否正确。使用MD5算法可以在一定程度上保证用户密码安全性，但仅仅这样还不够，因为如果用户正确密码的MD5值被第三方得到（这个值会在通讯中出现），他就可以利用该值合法登录到系统。需要加入一些干扰，服务器在玩家请求身份验证的时候生成一个随机数，并把该随机数返回给客户端，客户端将这个随机数与玩家输入的密码的MD5值进行混合，然后进行MD5加密，再将得到的结果提交至服务器，服务器端也与客户端一样，将该随机数与用户正确密码的MD5值混合后再进行MD5加密，对比两个结果

便可知道用户输入密码是否正确，而且即使该值被第三方截获，第三方也无法利用该值登录系统。

### 9.6.2　大厅服务模块

大厅模块是整个系统提供对外服务的一个主要接口，它提供了：

（1）游戏列表。

（2）游戏房间列表以及游戏房间的相关信息，比如房间内的人数、房间的游戏系数等。

（3）游戏房间对应的具体游戏服务器信息，比如IP、端口信息等。

在玩平台上任何一个游戏的时候，大厅都是处于开启状态的。实际上，大厅客户端与大厅服务器之间的连接时间是很短暂的。我们看到的大厅客户端中的信息，比如游戏列表，游戏房间列表或是游戏房间内的人数等，都是大厅客户端定期重新连接大厅服务器请求更新的，这样做对客户端来说可能并没有什么影响，但对于服务器端来说节省了相当多的资源。

### 9.6.3　中心服务模块

中心服务器目前主要负责两方面的权责：玩家状态维护和房间映射管理。

（1）玩家状态维护：主要负责用户全局状态维护。将用户状态单独抽象出来，由具体服务器维护，才能将各个业务流程拆散。中心服

务器的权职：维护用户状态和维护用户的passport（身份信息）。这样，可以实现同一账号只允许一个用户使用，其他的登录请求将被回绝。而当用户登录之后，其他的应用服务器只需要向中心服务器请求验证用户发送的passport是否正确。

（2）房间映射管理：主要负责将具体的游戏房间与游戏服务器绑定起来，当游戏服务器开启并决定对外服务后，将会向中心服务器发送注册信息，主要包含服务器提供的游戏类型，支持的人数上限等。而系统管理员可以通过映射配置工具，将配置好的房间映射至已注册的游戏服务器。该种注册机制，可以实现无限制的扩容能力，并可以实现底层本的不间断服务停机维护。在系统管理员决定要对某台游戏服务器进行停机维护后，可在游戏服务器端向中心服务器申 请停止服务器，中心服务器将绑定在该游戏服务器的房间自动解除。新的玩家便不会进入到即将维护的服务器。

### 9.6.4　数据服务模块

数据服务模块主要作用是为整个系统提供一个统一的数据访问层，系统中所有的数据库访问都必须通过数据访问模块，这样在物理部署方面，可以将数据库服务器完全隔离在防火墙之后，并将访问策略设置为只允许数据访问模块所在的服务器进行访问，这样提高了数据安全性。还可以数据访问模块中所有的重要数据操作进行日志记录，对于所有的数据改动状况都可以做到有据可查。

## 9.7　服务器端基础类库分析

### 9.7.1　服务器底层类库需求

服务器端开发，需要开发出一套比较完善的类库。我们不希望开发新游戏的时候还要从完成端口模型开始写起，对于通用模块，可以提供高层次的封装，使通用模块的开发变得简单易行。在封装的同时也需要提供各个层次的所有事件注册接口，保证开发的灵活性。

对于网络层模块，类库事件需要涵盖用户连接、用户断线、接收到合理消息、接收到非法消息（针对恶意用户构造的非法协议包，提交给上层用来指定处理策略）。

对于协议层，类库需要提供通用的消息处理模型，针对不同的应用需求可定义所需的消息。如果有较高的安全性考虑，类库还需要提供动态的加密解密模块，类库可以提供消息发送之前触发的事件注册接口，可以在消息发送之前可以进行高强度的加密。

对于应用层，类库需要提供了针对休闲游戏服务器端各种应用的通用事件，比如用户申 请与成功登录事件，或是登录失败事件（可以根据不用的需求，定制登录失败的处理，比如短时间内禁止重试，记录进入系统日志），用户选择房间事件，用户成功进入房间的事件，用户选择位置的事件，等等。

## 9.7.2　游戏服务器功能描述

### 1. 动态配置模块

游戏服务器端需要实现可动态调整大厅服务器IP地址，房间数量，房间底分，进入房间分值限制，房间人数限制，房间桌子数量等基础参数。

图9-3　游戏列表配置工具截图

## 2. 即时监控模块

需要即时监控模块来进行系统目前健康状况的追踪以及系统效率测试，需要监控的常规范围包括：

底层模块：

- CPU使用率
- 内存使用量
- 异常数量
- 网络带宽使用量

应用层模块：

- 房间列表
- 玩家列表，可查看玩家状态信息
- 房间内所有桌子的状态（该桌子上的游戏状态）
- 房间内所有玩家的状态（包括桌子上的玩家和在未坐下的玩家）
- 可选定桌子监控其所有消息

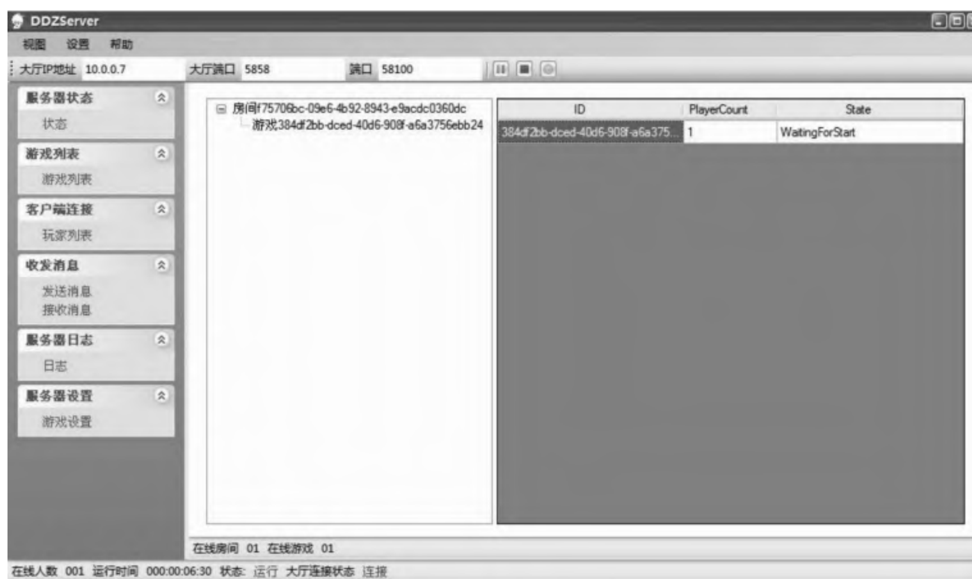



图9-4　服务器房间监控截图

## 3. 日志模块

日志模块最好可以采用动态分级机制，可以根据不同需求动态配置，以记录不同级别的日志信息。一般日志级别共分为五级，按其重要性依次排列。

一般模式下只需要显示相对重要的信息，比如财务操作，游戏结果，系统以及网络异常等等。但是在某些情况下可能需要动态更改记录等级，比如在性能分析，或是抵御黑客进攻时，可以通过查询详细的日志信息来辅助分析。

图9-5　服务器日志监控界面截图

所有日志被同一记录到文本中，一旦服务器出现异常，可以通过日志浏览器察看产生异常的服务器所记录的消息，确定异常的原因，从而解决问题，这样其他做有利于产品正式发布后对开发期未能发现的Bug进行及时修复。

## 9.7.3　后台管理模块

### 1. 会员管理模块

会员管理模块包括会员信息管理，会员状态管理（是否冻结账号）。后台可以针对不同界别的操作，设置不同权限的管理员组，比如信息浏览员，信息审核员，财务操作人员等等不同的管理员组，进行安全高效的管理。

## 2. 游戏管理模块

游戏管理模块可以方便的加入新的游戏，新的游戏房间，可以管理游戏房间的可配置参数。比如房间内桌子数量，可容纳的总人数，基础分，翻倍限制等等。

## 3. 财务管理模块

财务管理模块包括财务流程操作和财务报表两部分。财务流程操作主要针对资金的进出，比如针对某些玩家使用现金，支票或者汇款支付，财务后台提供资金流入的接口。针对玩家的兑换或者领奖请求，系统也提供了资金流出渠道。

财务报表模块主要包括系统的收支统计，具体玩家的收支统计，提供日表，周表，月表，季表，年表等标准报表，也支持用户自定义时间段进行统计。

一个完善的休闲游戏网络架构需要包括网络底层通讯，协议层编码解码，可动态调整的加密解密模块，应用层通用游戏逻辑，与中心服务器的通用交互模块，针对恶意连接的防范以及灵活的处理策略，多级动态配置日志系统，错误捕捉处理程序等等。

# 9.8 预测与同步技术

同步机制是网络游戏开发中最让开发人员头痛的部分之一，良好的同步机制要求能够保证每个客户端看到的东西大致相同，而网络游戏赖以通讯的媒介－以太网，在稳定性和速度上都很难担此重任，所以需要从软件层上尽可能减少由网络因素带来的误差。

先来考虑一下网络游戏中最常见的移动行为交互流程，有一个玩家A，处在PtA1点位置，在TA1时刻向服务器发了条指令，说自己要到PtA2点。假设服务器收到该条指令的时间是TS1，然后服务器会想其他玩家广播这条消息，消息的内容是“玩家A要移动至PtA2点位置”。假设玩家A收到这条广播消息的时间是TA2，另外一个玩家B收到这条消息的时间是TB2。如果不采用任何同步机制，玩家A在发出消息的TA1时刻开始在本地执行移动操作，玩家B在收到消息的TB2时刻开始执行玩家A的移动操作，那么玩家A与玩家B的客户端上玩家A开始移动行为的时间将会出现TB2-TA1的误差。TB2-TA1，我们需要一种同步方案来消除它。

最容易想到的一种方案，将所有客户端的时间与服务器进行同步（类似NTP的同步方案），之后发送的每个报文内加入时间戳，客户端在发送指令前，将当前时间记录下来存在指令报文内。这样其他玩家收到该条报文后与本地时间进行比较，便可以知道该条指令的准确发生时间。在上述的移动行为例子中，玩家B在TB2时间收到了玩家A发出的移动报文，他可以准确得知玩家A是在TA1时间开始移动的，那么玩家B的客户端中可以使用一些手段将中间的时间差弥补过来，可以算出在TB2-TA1的时间内玩家A应该移动到什么位置，然后直接将玩家A移动到该位置，或者设置一个t时间，使得在t时间之后，客户端B中看到的

玩家A位置与客户端A同步，但这两种方案会导致玩家出现速度不稳定和瞬间移动现象。

# 9.9 Dead Reckoning（DR）算法介绍与应用

美国国防部曾重金投资于一个用于军事训练的分布式模拟系统，其中包括国防部高级研究项目机构下的SIMNET项目中研发出来的分布式交互模拟（Distributed Interactive Simulation）协议-DIS。DIS包含了隐藏延时和减少带宽使用的技术一称为Dead Reckoning（DR）。DR最早使用在网络坦克群模拟，在之后的一些对同步要求较高的网络游戏（其中包括魔兽世界WOW，虚幻Unreal）和GPS定位系统中被广泛使用。

DR的基本概念就是预先在一些运动模拟算法之上建立一种协议，网络中的每个终端都可以用该协议来模拟游戏中的个体的行为，并且协议规定当推断算法和实际值的误差超过某个阀值的时候可以进行修正。在DIS协议中，当一个游戏个体被创建的时候，拥有该个体的计算机向网络上其他计算机发送该个体的状态协议数据单元（Protocol Data Unit，PDU），个体状态的PDU包含了能够唯一确定该个体状态和下一步运动状态的信息，包括位置，速度，加速度和方向。当参与分布模拟的其他计算机接收到该PDU的时候，它们创建该类型个体的本地备份。这样，每个网络上节点可以见到该个体（包括拥有该个体的终端也需要在本地创建该节点的备份）。

刚才提到了拥有该个体的终端也需要在本地维护一份该节点的备份，作用在于这个备份节点的状态信息与网络中其他终端维护的该个

体状态信息会完全同步。这样，当本地实际个体的运动状态发生变化后（比如玩家下达了不同的行动指令），就可以用节点的实际信息同该备份节点信息作比较，当大于协议定的阀值后，终端向网络中的其他终端发送新的PDU数据包进行同步。

来看一个例子，假设节点NodeA中有一个游戏人物PlayerA，PlayerA开始运动的时候，NodeA会向所有的节点广播一次PlayerA的PDU信息，包括：速度，方向，加速度。其他节点在收到PDU信息后就开始模拟PlayerA的运动轨迹和路线，包括NodeA本身，PlayerA在玩家的控制下会不断的改变方向，在运动过程中，NodeA会在不停地在后台比较其真实坐标和在模拟运动坐标的差值，当差值大于之前定义的极限误差时，则计算出当前PlayerA的实际速度S，方向O和加速度A，广播给网络中其他所有节点。其他节点在收到这条PDU之后，利用之前商定的算法，重新调整PlayerA的运动。

如果极限误差定义得大了，其他节点看到的偏差就会过大，如果极限偏差定义得小了，网络带宽就会增大。所以，这个极限误差应该根据具体游戏要求来制定。如果是回合制网络游戏，极限误差可以定义得稍大一些，以减少带宽。但是如果是操作要求较高的网络游戏，那么就需要把极限误差定得小一些，否则可能造成不同客户端之间看到状况差距很大，给玩家带来困扰。

Dead Reckoning的常用算法：

（1）目标点=原点+速度*时间

（2）目标点=原点+速度*时间+1/2*加速度*时间

（3）目标点=原点+速度*时间+1/2*加速度*时间

从原则上解决了网络延迟导致的不同步的问题，并且有效地减少了带宽，不过该算法主要用于移动中的同步，当然，移动的同步是网络游戏中同步的最大的问题。

具体的Dead Reckoning预测算法可参考*Defeating Lag With Cubic Splines* 书。

# 第10章　音效模块的设计与实现

音效模块的选择基本被DirectX一统天下了，那就是DirectX中的DirectSound、DirectMusic和DirectAudio。在最新的DirectAudio体系下，DirectSound和DirectMusic共同组成了DirectAudio。DirectSound能够处理波形声音，是处理数字声音的最主要的部件。DirectMusic则可以处理多种文件格式，MIDI、本身自带格式、WAV文件等，最后把音符信号传给Direct-Sound来处理。

本章主要介绍如何使用DirectSound来实现各种音乐相关控制，包括以下主题：

- 什么是声音、DirectX对声音有哪些支持。
- DirectSound的建立。
- 声音的播放和控制。
- 3D音效是如何实现的。
- 如何将DirectSound集成到引擎。

## 10.1　DirectX对音频的支持

原始森林中的一棵树倒下了可是没有听到它倒下的声音，那么它发出声音了吗？这个问题看起来比较奇怪，但是这有助于我们理解声音的实质：声音就是在一定介质中传播的一种波，比如说在空气中、在水中。对于人来说，只有当这种波引起了耳膜的震动才能叫做声音。

大自然中的很多种声音都有一种恒定的波形——正弦波。然而人能听到的声音却很复杂，不是能用简单的正弦波能描述的。图10-1给出了这两种声音波形的图示。

图10-1　声音的属性—振幅、频率

现实生活中的声音通过介质传播到人耳，那么声音是如何记录下来的呢？应该说，声音的采集与使用不是太复杂，比如说我们想录制一段2秒钟的对话，只要以恒定的频率来对声音进行采样然后存储可以了。如果我们以1000Hz的频率来录制的话，这就意味着2秒钟的对话被分成了2000等份。每一等份被称作是样品，对于每个样品我们记录其振幅以用来描述声音的大小。在规定的时间点上记录每个样品的振幅就得到了所谓的声音文件。

图10-2给大家展示了这样一个实例：声音被分成$N$个样品，正如我们看到的一样，当数字化后，声音也就失去了以前的波形，因为采样点不够多。也许您已经想到了，采样频率越大，我们得到的声音也就越逼真。但是具体多少才是最好的呢？这取决于我们想要得到的质量，以下是经常用到的频率：8000Hz，11025Hz，22050Hz和44100Hz。最后一个是CD标准，它的效果已经相当好了。

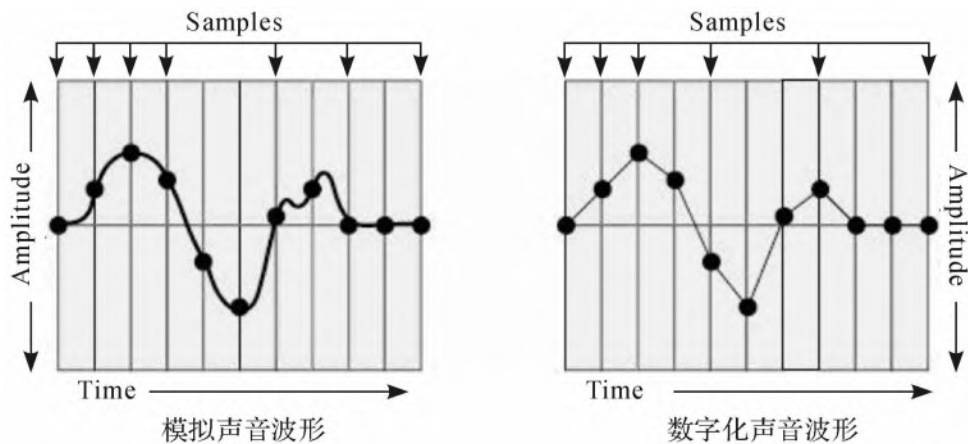



图10-2　模拟声音和数字化声音的比较

前面提到需要采集样品的振幅，那么如何来存储这些振幅呢？可以用8位，或者16位来存储这些振幅。如果用8位也就意味着我们有256个等级，而16位的话就有65536个等级。有的时候需要用更高的32位等。还有一个问题就是用几个通道？立体声要用两个通道，而一般的声音是一个通道。另外还可以利用一些特殊的加密或者压缩的算法对这些数据进行处理。不过更多的是使用WAV格式的声音文件，因为它是未压缩的，处理起来更加快捷。在DirectX中，最常使用DirectSound来处理WAV格式的声音文件。

除了WAV格式以外，在电脑游戏中还经常使用MIDI格式的音频。MIDI是存储音符的标准，MIDI文件里包含了如何演奏的说明（音符的高低、何时停顿等），然后专门的硬件设备就根据这些说明来演奏。MIDI一共使用128种标准设备接口，你可以只使用一部分设备，比如使用设备0（钢琴）等。一首歌曲可以分为多个音轨，每个音轨用特定的设备来播放。还可以把一些支持MIDI的乐器跟PC机相连，用乐器来给我们的PC机传送音符。因为PC机里都有音乐设备接口，所以可以播放相应的声音。

由于记录的是演奏的说明，所以一段背景音乐的MIDI格式文件往往非常小，因此在一些不追求音质的应用中可以选择使用MIDI。但是，MIDI也有一个致命伤，那就是它的播放效果取决于播放设备，也就说，如果两台电脑的播放设备（声卡）不同，那么两台电脑播放出来的音乐可能差别比较大。这是因为MIDI存储的是如何演奏，就像是给你一个五线谱音乐，不同的人唱出来的可能完全不同。虽然电脑硬件不会相差那么大，但是也会有比较明显的差异。

此外，DirectMusic也有自己的音乐格式，它与MIDI格式类似，只不过DirectMusic增添了不少的功能。比如说创建动态音乐序列以丰富我们的听觉经历。我们可以改变节拍、乐器等来改变音乐。DirectMusic使用.SGT扩展名，其他相关的扩展名包括：.BND、.CDM、.STY等。当我们用DirectMusic来创作歌曲时，相关的文件都会被DirectMusic自动处理。

在最新的DirectX Audio中，提供了新的体系结构来播放集成的音乐和声音效果。尽管仍然使用名称DirectSound和DirectMusic，但在它们之间已经没有明显的区别。DirectX Audio有两个组成部分：DirectSound和Direct-Music。在这两个部分中，DirectMusic的变化比较大（与前面的版本相比较），DirectSound几乎跟以前的一样（8.0）。

DirectSound是处理数字声音的最主要的部件，而DirectMusic可以处理多种文件格式：MIDI、本身自带格式、WAV文件等，最后把音符信号传给DirectSound来处理。大家可以从图10-3中理清它们三者的关系。

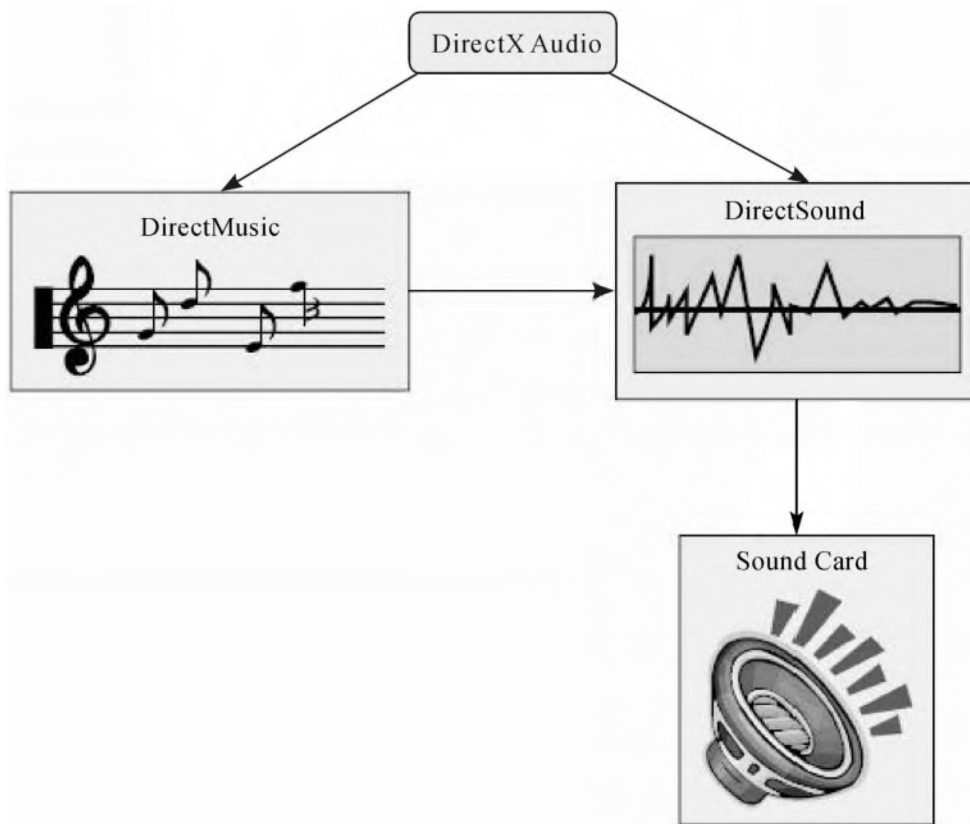



图10-3　DirectSound、DirectMusic、DirectAudio的联系与协作

新的音频体系结构将DirectMusic合成器作为主要的DirectX Audio声音生成器。这一高度优化的可下载声音级别2（DLS2）合成器可以创建所有的声音，对它们进行混音，并将结果发送到DirectSound缓存，以便进行进一步的处理。DirectMusic合成器也可以在输出之前将多个独立的声音进行混音。这样，多个独立的声音可以通过同一种音频效果进行处理，并分配到三维空间中的同一个位置。它们只使用一个DirectSound3D缓存，将CPU的使用和对三维硬件的要求降至最低。

下面重点讲述如何用DirectSound来播放、控制声音，以及如何在引擎中支持3D音效。

# 10.2 建立DirectSound

DirectSound是DirectX API的音频（waveaudio）组件之一，它可以提供快速的混音、硬件加速功能，并且可以直接访问相关设备，当然，最主要的是它提供的功能与现有的设备驱动程序保持兼容性。

DirectSound允许进行波型声音的捕获、重放，也可以通过控制硬件和相应的驱动来获得更多的服务。它的优势当然和DirectX的其他组件一样——速度，它允许你最大效率的使用硬件，并拥有良好的兼容性。有了这两个优点以后，你还需要考虑是否用它吗？

下面，一起来看看DirectSound可以为你做些什么？

（1）很方便地了解硬件能力，并且根据当前计算机硬件配置硬件来决定最好的解决问题的方法。

（2）弥补驱动程序的不足——通过属性设置以便硬件能力可以完全发挥，即便是驱动程序没有很好的支持该功能。

（3）短传输延迟时间的混音可以快速地响应流。

（4）3D音效。

（5）捕获声音。

应该说，DirectSound的功能是足够强大的，至少，对于非专业级应用是绝对够了，这里也包括游戏。下面，让我们来分步分析如何建立DirectSound。

## 10.2.1　创建DirectSound对象

创造DirectSound对象最简单的方法是调用DirectSoundCreate函数。

```
LPDIRECTSOUND8 lpds;HRESULT hr=DirectSoundCreate(NULL, &lpds, NULL));
```

该函数的第一个参数是硬件设备，NULL表示使用默认的设备；第二个参数是远程指针LPDIRECTSOUND8的地址，也就是创造的DirectSound对象放置的地址，第三个参数必须为NULL，保留参数，暂时没有用。

当没有相应的设备或设备在别的程序的控制下不能响应用户的呼叫时，函数返回出错（非DS_OK）。这时，如果程序继续工作，所有和DirectSound对象相关的操作都将不可进行！

## 10.2.2　设置协作级别（Cooperative Level）

因为Windows是一多任务环境，可以允许多个应用程序同时工作，当然也会产生多个程序在同一时刻使用同一设备工作的情况，通过协作级，DirectX可以保证所有的程序在使用同一设备时不会发生冲突。所以每个使用DirectSound的程序都应该有一协作级用来决定允许访问的设备。

DirectSound有四种合作级别：标准级（DSSCL_NORMAL）、优先级（DSSCL_PRIORITY）、独占级（DSSCL_EXCLUSIVE）和写主缓冲级

（DSSCL_WRITEPRIM ARY），其中游戏普遍使用优先级这种级别，因为可以使程序在同一采样条件下作出最柔韧的输出。过程如下所示：

HRESULT hr=lpds->SetCooperativeLevel（hwnd，DSSCL_PRI-ORITY）；那么，为什么要使用优先级这种写作级别呢？我们首先来分析4种级别的不同：

- 标准级（DSSCL_NORMAL）：该级别只能使用22kHz、立体声（STER-EO）、8位的音乐，并且不能直接地写主缓冲，也不能使用压缩过的声音。
- 优先级（DSSCL_PAIORITY）：可以实现硬件混合（hardware mixing），可以设置主缓冲的声音格式（可以根据需要来使用不同质量的音乐）和压缩过的音乐。
- 独占级（DSSCL_EXCLUSIVE）：当应用程序在前台工作时，其他程序是不可使用声音的。
- 写主缓冲级（DSSCL_WRITEPRIMARY）：最高的合作级，程序可以直接的操纵主缓冲，而且程序必须直接地写主缓冲区（最基层的操作）。在这种级别，次缓冲区将不可用。而且，除了该级别外，所有试图LOCK主缓冲的操作都将失败，也就是说只有该级别可以对主缓冲进行写操作！当使用写主缓冲级的程序处于前台时，后台所有程序的第二缓冲都将停止且丢失，而如果这时使用写主缓冲级的程序转到后台工作，它的主缓冲也将丢失并且在又一次转到前台工作时应该还原（Restore）。

如果要设置协作级为写主缓冲级，则应先确定现在是否可以使用该级别——调用IDirectSound8::GetCaps()函数，检查DSCAPS结构里是否有DSCAPS_EM ULDRIVER标志。

### 10.2.3　检索硬件信息

DirectSound允许应用程序检索硬件信息。当然，在一般情况下，这样做是不必要的，因为DirectSound可以自动有效的使用硬件加速，我们完全可以不用去管硬件是否具有某些能力，不过为了提高程序的效率，这样做还是有用处的。

其中，检索硬件信息使用IDirectSound8::GetCaps方法，例如：

```
DSCAPS dscaps;

dscaps.dwSize=sizeof(DSCAPS);

HRESULT hr=lpds->GetCaps();
```

### 10.2.4　扬声器的设置

DirectSound的扬声器设置可以用来调节输出音量的大小和优化3D效果。

在WIN98及其他更新的操作系统中，可以通过IDirectSound8::GetSpeakerConfig()来获得当前扬声器的设置，并通过IDirectSound8::SetSpeakerConfig()来改变扬声器设置；而在WIN95里IDirectSound8::GetSpeakerConfig()只是简单的返回一个默认值或是返回使用IDirect-Sound8::SetSpeakerConfig()设置后的值。

### 10.2.5　压缩

应用程序可以通过IDirectSound::Compact()来获得最多的可用内存，当然从前面对合作级别的讨论我们可以发现使用压缩的前提是程序的合作级别至少应该是优先级。

## 10.2.6　建立缓冲区

在初始化DirectSound时，它会自动地为程序创建一主缓冲，这个主缓冲的作用就是混音并送到输出设备。

除了主缓冲外，程序至少还应该创建一个辅助缓冲，辅助缓冲的作用是储存将要使用的声音，它可以在不使用的时候释放掉，但是，主缓冲是不可释放的！

用户可以在同一段物理内存上创建两个或更多的辅助缓冲（使用IDi-rectSound8::DuplicateSound-Buffer()方法），但是如果最初的缓冲（原本）所在的硬件内存资源不支持多缓冲，那么这个调用将以失败告终。

在硬件支持的情况下，DirectSound还可以同时播放多个声音（硬件混音），当然，目前的计算机基本都支持硬件混音。同时，DirectSound播放声音的时间延迟很短，如果在播放声音的同时播放动画，用户也感觉不到延迟。不过，如果DirectSound需要通过软件仿真来完成这一动作，那么延迟时间将延长5~8倍。

通常情况下，用户并不需要和主缓冲打交道，DirectSound会自己来管理它的，除非用户要自己写混音操作，那时，DirectSound才会让用户自行管理主缓冲。

除了主缓冲以外，DirectSound还有辅助缓冲。它可以分为两种——静态缓冲和流缓冲。静态缓冲是一段内存空间对应一段完整的声音，它可以一次将全部的声音存入缓冲，所以播放的速度比较快；而流缓冲并不将全部的数据一次读入缓冲，而是在播放声音时动态的读入，它的好处在于占用空间较小。这两种辅助缓冲形式可以适应不同的程序需求。但一般地说，如果声音是需要再三播放的，而且容量有限（好比游戏音效），那么使用静态缓冲就更有助于提高程序的效率；相反，如果是很冗长的音乐，还是使用流缓冲的好。

辅助缓冲的建立如例程10-1所示。

**例程10-1　DirectSound辅助缓冲区的建立**

```
LPDIRECTSOUNDBUFFER CreateSoundBuffer(constDWORD&flags,   //缓冲区属性
UINT size,                     //缓冲区大小
constWAVEFORMATEX&format)    //对应的音乐格式
{
    LPDIRECTSOUNDBUFFER pSBuf;                      //返回的缓冲区指针
    DSBUFFERDESC desc;                              //缓冲区描述结构
    memset(&desc, 0, sizeof(DSBUFFERDESC));         //清空结构内容
    desc.dwSize          =sizeof(DSBUFFERDESC);     //配置结构大小
    desc.dwFlags        =flags;                 //缓冲区属性
    desc.dwBufferBytes=size;                     //设定缓冲区大小
    desc.lpwfxFormat=(LPWAVEFORMATEX)&format;     //设定缓冲区格式
    //建立次缓冲区
        HRESULT  result=lpds->CreateSoundBuffer(&desc,  &pSBuf, NULL);
    if(result!=DS_OK){                              //不成功
        pSBuf=NULL;
```

```
        returnNULL;
    }
    else                                            //成功
        return pSBuf;
}
```

这里，有必要说明函数的参数flags和format如何设置。

缓冲区属性flags首先应该指明该缓冲区是静态缓冲（设置DSBCAPS_STATIC标志）还是流缓冲，默认值是流缓冲还是流缓冲。然后，应该声明该缓冲需要用到的控制选项。这项工作需要你为DSBUFFERDESC结构设置以DSBCAPS_CTRL为首的标志（这些标志可以是单独的来使用，也可以同时设置几个）。可用的控制有3D属性、频率、Pan（左右声道的差值）、音量、播放进度。为了能在所有的声卡上都可以获得做好的效果，最好只设置需要的控制选项。如果一块声卡支持硬件缓冲但不支持底盘控制（pan control），那么DiractSound只会在DSBCAPS_CTRLPAN标志没有被声明时使用硬件加速。这也就是说，DirectSound通过控制选项来决定如何为缓冲来分配硬件资源。如果使用一个缓冲不支持的控制，譬如为一个并没有声明DSBCAPS_CTRLVOLUME标志的缓冲去调用IDirectSoundBuffer8::SetVolume()方法，也是不可能成功的。

缓冲区音乐格式属性format可以自己设置，但游戏中一般直接从音乐文件中得到。我们不妨用WAV格式音乐文件为例，分析如何加载音乐到缓冲区。

## 10.2.7 音乐文件加载

WAV格式的加载的过程非常固定，不过里面涉及很多音乐文件的编码问题，没有必要在这里一一解释。一些东西是固定的，用户只需要拿来用就好了，要将精力放在一些需要创造性的方面，比如面向对象的封装，不是吗？

例程10-2说明了如何加载WAV音乐文件到次缓冲区中，参数为文件路径和返回的次缓冲区指针。

### 例程10-2 DirectSound辅助缓冲区的建立

```
voidLoadMusic(char*musicFile，LPDIRECTSOUNDBUFFER&lpDSBuffer)
{
    /**载入声音*/
    //开启多媒体文件
        MIO  hmmio=mmioOpen(musicFile,  NULL,  MM  IO_ALLOCBUF| MM
IO_READ);
    //RIFF区块的信息
    MMCKINFO ckRiff;
    ckRiff.fccType=mmioFOURCC('W', 'A', 'V', 'E');
    //设定档案类型
    mmioDescend(hmmio, &ckRiff, NULL, MM IO_FINDRIFF);
    //设定区块类型
    MMCKINFO  ckInfo;                          //子区块的信息
    ckInfo.ckid=mmioFOURCC('f', 'm', 't', '');
    mmioDescend(hmmio, &ckInfo, &ckRiff, MM IO_FINDCHUNK);
    //读取档案格式
    WAVEFORMATEX  swfmt;                       //声音结构
    mmioRead(hmmio, (HPSTR)&swfmt, sizeof(swfmt));
    mmioAscend(hmmio, &ckInfo, 0);      //跳出子区块
    ckInfo.ckid=mmioFOURCC('d', 'a', 't', 'a'); //设定区块类型
    mmioDescend(hmmio, &ckInfo, &ckRiff, MM IO_FINDCHUNK);
    //建立次缓冲区失败
```

```
lpDSBuffer=CreateSoundBuffer(DSBCAPS_STATIC| DSBCAPS_CTRLVOLUME,
ckInfo.cksize, swfmt);
    //锁定缓冲区
    DWORD    bytesAudio;
    LPVOID    pAudio;
    lpDSBuffer->Lock(0, size, &pAudio, &bytesAudio, NULL, NULL,
NULL);
    //读取音文件数据
    mmioRead(hmmio, (HPSTR)pAudio, bytesAudio);
    //解除锁定缓冲区
    lpDSBuffer->Unlock(pAudio, bytesAudio, NULL, NULL);
    //关闭音乐文件
    mmioClose(hmmio, 0);
}
```

至此为止，所有的前期准备工作都做好了！下面就将在游戏中播放美妙的音乐了！

# 10.3　声音的播放与控制

一般来说，在DirectSound中播放声音要通过以下步骤：

（1）锁定辅助缓冲的一部分以获得你所需要的那部分缓冲的基址。

（2）向缓冲写数据。

（3）解锁缓冲区。

（4）使用IDirectSoundBuffer::Play方法来播放声音。如果是使用的流缓冲，还需要反复地执行（1）~（3）步骤。

不过，因为在游戏中的音乐文件一般都很短小，所以经常使用静态缓冲区，并且第（1）、（2）、（3）步都在建立缓冲区的时候完成了。于是，我们只需要关注第四步，以及如何控制声音播放。

IDirectSoundBuffer::Play （ DWORD dwReserved1, DWORD dwPriority, DWORD dwFlags）

函数有三个参数，第一个参数是保留参数，设为NULL，第二个参数在非DSBCAPS_LOCDEFER属性的缓冲区时也必须设为NULL，第三个参数可以设置音乐是否重复播放。如果需要重复播放该音乐，则应该设成DSB-PLAY_LOOPING，否则还是NULL。

除了播放音乐外，DirectSound还可以对音乐播放进行控制。可以通过 IDirectSoundBuffer::GetVolume() 和 IDirectSoundBuffer::SetVolume()来获得或设置该缓冲的音量，设置主缓冲的音量将改变声卡的设置。同样地，也可以通过 IDirectSoundBuffer::GetFrequency() 和 IDirectSoundBuffer::SetFrequency()来获得或设置声音的频率，通过IDirectSoundBuffer::GetPan()和IDirectSoundBuffer::SetPan()来检索或改变左右声道的相对差。但是不可以改变主缓冲的相应设置，也就是说，这些缓冲控制都必须在设置了相应的标志位后才可以使用。

IDirectSoundBuffer::Play()方法通常都从当前的播放进度开始播放音乐。在缓冲刚建立时，播放进度是指向0，而当一段音乐播放完毕以后，播放进度指向那段音乐数据最末端的下一字节，同样的，当音乐被停止时，播放进度也指向停止位置的下一字节。可以通过 IDirectSoundBuffer::GetCurrent-Position() 和

IDirectSoundBuffer::SetCurrentPosition()来检索或设置播放进度。

比如，要让一段缓冲对应的音乐从头开始重复播放，则可以如下编写代码：

```
LPDIRECTSOUNDBUFFER lpSBuf;                         //建立的次缓冲区，设已经填充
lpSBuf->SetCurrentPosition(0);                      //设定播放位置为头
lpSBuf->Play(0, 0, DSBPLAY_LOOPING);                //设定重复播放
```

当希望停止播放音乐时，可以调用IDirectSoundBuffer::Stop()方法。

同时，对DirectSound来说混音是很容易的，它允许你同时播放多个辅助缓冲，它可以自己来完成这些任务。只要程序正确的指定DSBCAPS_STATIC标志，DirectSound就可以最大限度的使用硬件加速。而且，如果所有的缓冲都使用同一种声音格式而且硬件输出也是使用这种格式，那么DirectSound的混音将不需要在格式转换上花任何的工夫，从而大到最优的效果。

# 10.4　3D音效

DirectSound 3D可以说是DirectSound的精华所在，在3D游戏的开发中，3D音效一般都是采用DirectSound 3D来完成的。

## 10.4.1　3D空间、声源、听者

Directsound 3D是通过软件模拟来实现3D音效的，所以要先讲一下Dsound的3D模拟空间。这个空间类似现实空间，可以用笛卡儿坐标系来描述Dsound的3D空间，有$x$，$y$，$z$三个坐标轴。

在这个模拟空间中Dsound提供了模拟的声源对象和倾听者对象（liste-ner），声源和听者的关系可以通过三个变量来描述：在三维空间的位置，以及运动的速度，以及运动方向。

位置即声源和听者在三维空间的所在位置，随着两者的相对位置不同，则听者便会听到不同的声音效果。

速度为声源和听者在三维空间中的移动速度，此项特性同样会改变两者在空间的坐标，以产生不同的声音效果。

声源和听者相对运动的方向也会影响听者听到的声音效果，因为声音是具有方向性的。

知道了3D声源以及3D环境中的听者，那么怎么产生3D音效呢？一般来说，在产生3D音效的时候，主要有下面的几种情况：

- 是声源不动，而听者在模拟的3D空间进行运动。
- 是听者不动，让声源在模拟的3D空间进行运动。
- 听者和声音同时在运动。

DirectSound提供了听者和声源对象的接口，可以通过上面提到的三种方式设置改变声源或者听者的位置，运动速度和方向就可以形成3D音效了。

在3D环境中，通过IDirectSound3DBuffer8接口来表述声源，这个接口只有创建时设置DSBCAPS_CTRL3D标志的缓冲区才支持这个接口，

这个接口提供的一些函数可以用来设置和获取声源的一些属性。在一个虚拟的3D环境中，可以通过主缓冲区来获取IDirectSound3DListener8接口，通过这个接口可以控制着声学环境中的多数参数，比如多普勒变换的数量，音量衰减的比率等等。不过，DirectSound的接口很简单，因为大量的计算工作都在DirectSound内部完成了。

## 10.4.2 最大最小距离

当听者越接近声源，那么听到的声音就越大，距离减少一半，音量会增加一倍。但是，当你继续接近到声源，当距离缩短到一定距离后，音量就不会持续增加。这就是声源的最小距离。

声源的最小距离，就是声音的音量开始随着距离大幅度衰减的起始点。例如，对于飞机，这个最小距离也许是100m，但是对于蜜蜂，这个最小距离是2cm，根据这个最小距离，当听者距离飞机400m时，声音的音量就要衰减一半，对于蜜蜂来说，当超过4cm的时候，音量衰减一半。

DirectSound的缺省最小距离DS3D_DEFAULTMINDISTANCE定义为1个单位，或者是1m。我们规定，声音在1米处的音量是full volume，在2m处衰减一半，4m处衰减为1/4，以此类推。对于大多数声音来说，我们要设置一个比较大的最小距离，这样，当声音运动的时候，不至于衰减的这么快。

声源的最大距离，是指声源的音量不再衰减的距离，称为声源的最大距离。对于Directsound 3D buffer缺省的最大的距离DS3D_DEFAULTMAX-DISTANCE是1 Billion。也就是说，当声音超出我们

的听觉范围以外的时候，衰减还是在继续。在VXD驱动下，为了避免不必要的计算处理，我们在创建buffer的时候就要设置一个合理的最大距离。

声源的最大距离也可以用来确保玩家始终都能够听到声音。例如，如果将某种声音的最小距离设置为100m，那么声音可能在1000m的处衰减的可能就听不到了；如果将最大的距离设置为800m，这样就可以保证声音在超过800m时，也能够听到。

## 10.4.3　处理模式

3D Sound Buffers有三种处理模式，Normal、Head-Relative和Disabled。

在Normal模式下，声源的位置和方向是真实世界中的绝对值，这种模式适合声源相对于听者不动的情形。

在Head-Reative模式下，声源的所有3D特性都跟听者的当前的位置、速度以及方向有关，当听者移动，或者转动方向，3D buffer就会自动的重新调整世界坐标。这种模式可以应用实现一种在听者头部不停的嗡嗡叫的声音，那些一直跟随着听者的声音，根本没有必要用3D声音来实现。

在Disabled模式下，3D声音失效，所有的声音听起来好像发自听者的头部。

## 10.4.4　声音的锥效应

没有方向的声音在各个方向上的振幅都相同，有方向的声音在该方向上的振幅最大，声音的锥效应分为内部的锥效应，和外部的锥效应，锥的外部的角度应该大于等于锥的内部角度。

在锥的内部，只需要考虑Buffer中的基本音量以及距离听者的距离远近，而何听者的方向，声音的音量就跟没有锥效应一样。

在锥的外部，正常的音量被削弱了，从0到负的百分之几分贝。

在锥体的内部和外部之间，是一个过渡带，从内部的音量到外部的音量，逐渐降低。图10-4显示了声音的锥效应。

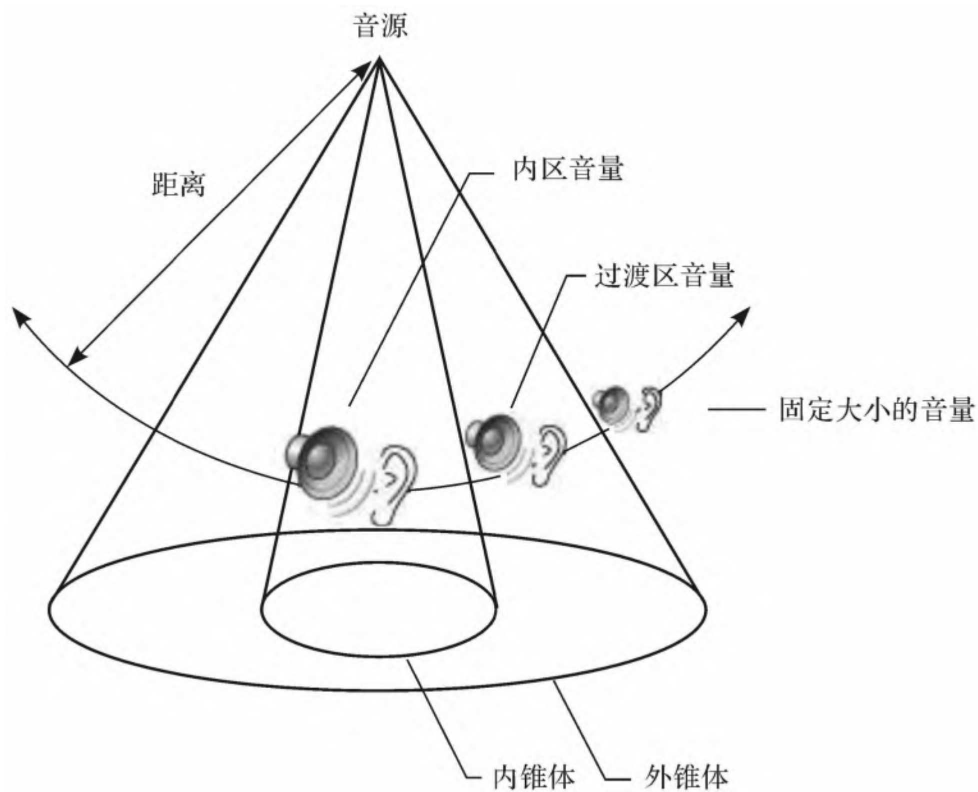



图10-4　音量和声音锥的关系

每一个的3D Buffer都有一个声音锥，但是缺省的情况下，每一个3D声音buffer都像一个没有衰减的声源，因为锥体内外都没有声音的音量的降低，锥的角度内外的角度是360度，除非我们的应用程序改变这个设置，否则的话，3D声音没有方向感。

利用3D声音，可以给程序添加比较奇异的特技。例如，可以将声音源放到房间的中间，将声源的方向朝向门的方向，然后将声音的锥体角度包含门的宽度，将锥体的外部设置的更宽一些，然后将锥体的外部音量设置为0。这样，当听者只有在门的附近才能听到声音，在正对门的位置，声音才更响。

## 10.4.5　声源的创建

声源的特性在上面已经解释得差不多了，那怎么使用DirectSound在程序中创建一个3D声源呢？

其实，3D声源和2D声源的创建非常相似。它有以下三个不同：

- 需要定义一个IDIRECTSOUND3DBUFFER接口的指针，用来存储和控制3D音效。
- 在缓冲区的属性设置时，必须指明是DSBCAPS_CTRL3D。它表示该缓冲区支持3D音效。
- 在声音填充到次缓冲区后，需要调用IDirectSoundBuffer::QueryInter-face函数，得到接口IDIRECTSOUND3DBUFFER。

此外，声源IDIRECTSOUND3DBUFFER还支持以下操作：

- 得到声源位置接口：GetPosition()
- 设置声源位置接口：SetPosition()
- 得到声源音量不变的最小距离：GetMinDistance()
- 设置声源音量不变的最小距离：SetMinDistance()
- 得到声源速度：GetVelocity()
- 设置声源速度：SetVelocity()
- 得到声源朝向：GetConeOrientation()
- 设置声源朝向：SetConeOrientation()
- 得到声音锥的张角：GetConeAngles()
- 设置声音锥的张角：SetConeAngles()
- 得到声音锥外部的音量：GetConeOutsideVolume()
- 设置声音锥外部的音量：SetConeOutsideVolume()

## 10.4.6 听者的创建

听者在DirectSound中是用接口IDIRECTSOUND3DLISTENER来表示，它有位置、朝向和速度属性。

可以按如下步骤建立一个听者：

```
LPDIRECTSOUND3DLISTENER SetupListener(LPDIRECTSOUND8 lpDS){
                        LPDIRECTSOUND3DLISTENER      pListener;
//听者
                              LPDIRECTSOUNDBUFFER       pPBuf;
//临时的主缓冲区
                                    DSBUFFERDESC         desc;
//缓冲区描述结构
                      memset(&desc,      0,      sizeof(desc));
//清空结构内容
                                      desc.dwSize=sizeof(desc);
```

```
//配置结构大小
                desc.dwFlags=DSBCAPS_PRIMARYBUFFER| DSBCAPS_CTRL3D;
//缓冲区属性
                                           desc.dwBufferBytes=0;
//设定缓冲区大小
                                          desc.lpwfxFormat=NULL;
//设定缓冲区格式
                lpDS->CreateSoundBuffer(&desc,    &pPBuf,    NULL);
//创建缓冲区
                 pPBuf->QueryInterface(IID_IDirectSound3DListener,
(VOID**)&pListener);  //创建听者
                                              pPBuf->Release();
//释放主缓冲区
    return pListener;
}
```

上面的代码省略了错误检测，不过它描述了如何建立一个听者。最重要的步骤是需要将缓冲区的属性设为DSBCAPS_CTRL3D，并要求一个IDI-RECTSOUND3DLISTENER接口。最后返回的听者指针就可以用了。

IDIRECTSOUND3DLISTENER接口支持一些操作，主要是得到或设定听者的位置、朝向、速度，可以在自己的引擎中对它进行封装。

# 10.5　音效模块封装

学会使用DirectSound只是第一步，对于编写游戏引擎来说，还需要封装，并提供统一、高效、简单的接口。

在设计前，先提出功能需求：

（1）声音资源应该能够管理，即查找、删除、插入、防止重复资源加载等。

（2）3D声音和2D声音从本质上说都是声音，只是3D声音多了几个3D的属性。

（3）在一个游戏客户端中，要么有一个听者，要么没有听者。因为玩家只有一个，要是想以不同物体的角度听声音的话，可以切换游戏中玩家位置。

（4）3D声音要能参与到场景图的互动中去。

（5）听者要能参与到场景图的互动中去。

（6）听者、3D音源和场景图中的物体要相对独立。

针对这些要求，可以按如下方式设计引擎中的音效模块：

（1）设计一个类GESound，它抽象了DirectSound中声音的共同特征，比如都有缓冲区、播放的形式都一样、都能加载、释放等等。

（2）设计类GE2DSound和GE3DSound，它们都继承自GESound，并分别抽象2D声音和3D音源。

（3）听者必须是单例模式，保证一个应用中不会有超过1个听者的存在。

（4）3D声音类GE3DSound可以保存一个GEObject的指针，于是GE3DSound的很多属性和行为可以从绑定的5这个Object上得到，比如位置、朝向。

（5）听者类GEListener保存一个GEObject的指针，于是GE3DSound的很多属性和行为可以从绑定的这个Object上得到，比如位置、朝向。如果需要切换聆听角度，可以切换指针指向希望的物体上去。

（6）3D声音类GE3DSound和听者类GEListener和场景图的GEObject类是聚合关系，相对独立。

OK，大功告成！图10-5是引擎中对音效模块的封装示意图。

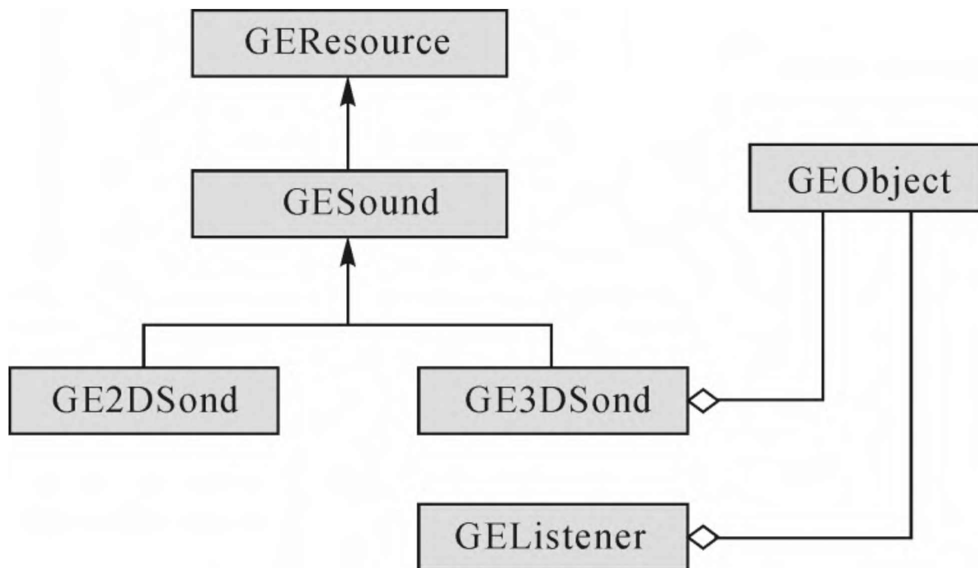



图10-5　引擎中的音效模块封装

# 第11章　游戏中的人工智能技术

随着多媒体技术的迅猛发展和更加成熟，游戏世界中的多媒体表现效果将被发挥到极致；同时，用户对游戏质量的要求也正随着游戏产业的发展而大幅提升，必将把注意力从游戏的声光色影等视觉、听觉中回归到游戏的本质——游戏的可玩性上。用户迫切期盼的将是充满着具有更具智能、有思想、有灵气的游戏角色的游戏世界。只有娱乐性强的高质量有特色的优秀作品才能在市场生存，未来游戏发展和创新的后劲将依赖于人工智能技术在游戏平台上实现的水平，而游戏也将成为学者们研究AI的高效实践平台，游戏中的人工智能（Game AI）技术将是下一代游戏引擎将要突破的技术重点之一。

## 11.1　Game AI的特点

Game AI技术与传统的“学院派”AI技术有一定的区别，它主要体现在目的和方法上的不同。学院派AI似乎不食人间烟火，永远在痛苦地求索一个完美的答案；而Game AI需要满足高效性和实时性的约束，因此它务实灵活，只要能用、够用即使退而求其次也可。虽然目前Game AI技术在游戏软件中的应用效果还不尽如人意，但只要我们再多努力一些，就能获得“不完美但很震撼”的效果！

下面我们从总体上了解一下常用的Game AI技术。首先需要说明的是，Game AI技术在计算机实现上主要有两种不同的方式：

（1）工程学方法（Engineering Approach）：采用传统的编程技术，使系统呈现智能的效果，而不考虑所用的方法是否与人或动物机

体所用的方法相同。如规则推理、有限状态机（FSM）、游戏脚本驱动等。

（2）模拟法（Modeling Approach）：它不仅要看效果，还要求实现方法也和人类或生物体所用的方法相同或相类似，如遗传算法（GA），神经网络（ANN）等。

采用工程学方法，需要人工详细规定程序逻辑，并通常都通过硬编码实现。如果游戏逻辑非常简单尚可；但如果游戏逻辑非常复杂，NPC（Non-Play-er Character）数量和活动空间增加，人工编程的复杂度就会急剧增加，容易出错，很难调试，最终繁杂的代码会超出人工处理的极限。

而采用采用模拟法，程序员要为每一个NPC角色提供一个智能系统（模块）来进行控制，这个智能系统开始什么也不懂，就像初生婴儿那样，但它能够自学，能渐渐适应环境，应付各种复杂情况。利用这种方法来实现人工智能，程序员需要具有生物学的思考方法：遗传算法模拟人类或生物的遗传、进化机制；人工神经网络则模拟生物大脑中的神经细胞的活动方式。为了得到相同智能效果，两种方式通常都可以用。由于模拟法编程时无须对角色的活动规律做详细规定，应用于复杂问题，通常会比工程法更省力。但模拟法都是使用局部最优去逼近全局最优秀，这就很难避免局部收敛太快或状态抖动的情况出现。

不管是使用工程法还是模拟法，Game AI有一个原则，需要具体问题具体分析，根据不同的问题上下文背景选用最合适的解决方法，不要希望一种技术能包揽一切。

最后，需要主意的是在游戏软件开发中技术是为内容服务的，Game AI技术是为了增加游戏的可玩性，所以游戏角色既不能太愚蠢，毫无挑战的电脑对手会使玩家失去兴趣；也不能太“聪明”，太智能的电脑对手会挫伤玩家的积极性和自信，甚至会让玩家产生对电脑作弊的怀疑。

# 11.2 常用Game AI技术

Game AI是一个很宽泛的主题，但游戏开发中常用实用Game AI技术主要有：

- 智能体个体行为的模拟仿真，如障碍规避、路径规划、跟踪、追杀、躲避、逃跑、截击、埋伏、偷袭等。
- 智能体群体行为的模拟仿真，如群体间的对抗与合作、群体的列队规划，群体的聚集与离散等群体行为。
- 使用有限状态机技术对游戏角色进行建模。
- 使用认知建模、虚拟人技术刻画智能体的个性和情感。
- 使用神经网络、遗传算法等技术实现智能体的自学习能力。
- 使用数据驱动的脚本系统处理游戏中简单的智能行为。
- 使用模糊逻辑，贝页斯决策技术增加游戏的不确定性。

这些技术在具体产品中都得到了应用：

表11-1 Game AI常见技术的比较

| 技术 | 优点 | 缺点 | 实际应用产品 |
| --- | --- | --- | --- |
| 有限状态机（Finite State Machine） | 简单，通用，可以和其他技术混合使用，计算开销小，求解能力相对较强，可以管理整个游戏场景，也可以操纵单个的游戏对象和人物。现在，FSM 技术的变体"Fuzzy FSM"（模糊有限状态机）技术正在实际开发中流行 | 智能行为过于确定性，规模大时可能会产生组合爆炸 | 《帝国时代》、《半条命》、《Doom》、《Quake》等比较成功的游戏中均使用了该技术 |
| A * 寻路算法 | 能逼真地模拟智能体的路径规划，使之能选择一条代价较小的路径绕过障碍物或危险性区域 | 计算代价大 | 几乎所有的游戏中都使用了该技术 |
| 脚本语言 | 简单，可以被不懂编程技术的策划人员使用，能编写"剧本"清晰有效的控制游戏的发展。除此之外，还能驱动事件、为 NPC 的智能行为建模、某些任务的自动化、人机对话等 | 过于确定性 | 几乎所有的游戏中都使用了该技术 |
| 模糊逻辑 | 更多的灵活性、可变性，适合求解比较复杂的非线性问题，能增添游戏中的不确定性，比如游戏中的战略战术决策、行为选择、NPC 的健康状况计算、情绪的状态变化控制等 | 使用模糊逻辑每次都要从头计算，系统实现上比较复杂，对开发人员的数理素养要求比较高。实际中的应用：在 Swat2、Close Combat、Petz 等游戏中使用了该技术 | 在《半条命》、《Black & White》等游戏中均得到了使用 |
| 群体行为模拟 | 能较逼真的模拟出智能体群的列队、聚集、移动等群体行为 | 计算代价大，且只适用于有限的场合 | 在《半条命》、《Nation》、《Enemy》等游戏中均得到了使用 |
| 决策树 | 可读性强，训练和评估的效率高，即使在数据有部分缺失的情况下也能较好工作 | 需要训练和调整 | 在《Black & White》游戏中使用该技术进行分类、预测和学习 |
| 神经元网络 | 使用灵活，能增加较多的不确定性 | 需要训练和调整，挑选神经元的输入/输出变量比较困难，算法的实现复杂，计算代价大 | 在游戏中可用于分类、预测、学习、模式识别、行为控制等，在《Black & White》、《Creatures》和《Heavy Gear》等游戏中得到了应用 |
| 遗传算法 | 在很大的、很复杂的、没有明确模型的问题求解中非常有效 | 需要充分的训练和调整，速度比较慢，算法的实现比较复杂，计算代价大 | 在游戏中可用于优化、学习、策略形成、行为进化等方面，在《Creature》和《Return Fire Ⅱ》等游戏中得到了应用 |

# 11.3 路径规划与移动技术

路径规划与移动问题是Game AI的一个重要部分，它通常是指：智能体个体或智能体群在障碍物规避、体力消耗、动力学规律、生命威胁等因素的约束下规划出一条路径，使智能体能以较小的计算代价从当前位置到达目标位置。因为游戏运行过程中会对众多智能体角色频繁进行路径规划问题的计算，它的表现效果和性能直接影响到一款游戏的成败，所以游戏软件对路径规划准确性、高效性和实实性的有着非常高的要求。因此，3D游戏中往往先对游戏场景中的地图进行预处理，使用Waypoints和Navigation Mesh来简化路径规划，减少寻路中的节点数目，图11-1就是游戏中预先设定的Way-points。

图11-1　路径规划实例

游戏中的智能体角色在进行寻路时通常分两个阶段：

（1）从起始位置到目标位置规划出一条合适的路径，使规划代价和通过该路径的代价在当前游戏上下文背景下较小。

（2）在每帧游戏画面的时间片处理中沿着规划好了的路径逐步接近目标位置。在这个过程中可能会遇到规划阶段所没有考虑到的移动对象，例如移动的障碍物，同伴或敌人等，这时就需要进行碰撞检测处理。

通常移动阶段相对简单些，而路径规划阶段要复杂得多，它需要考虑较多的上下文环境因素：

（1）游戏地图的结构。

（2）有无障碍物、障碍物是静止还是运动。

（3）目标是静止还是运动。

（4）周围的地形状况（上坡还是下坡，陆地还是沼泽）。

（5）智能体当前的生理状况（如疲劳、饥饿、受伤等）。

（6）天气状况（如晴天、白天、夜晚、下雨、刮风等）。

（7）智能体周围有无敌人和其他危险。

（8）智能体的一些参数（可视范围，白天高于夜晚，晴天高于阴天，听力范围，杀伤范围等）。

（9）智能体的动力学属性（位置，速度，受力分析，加速度，动力学规律约束）等。

（10）若是在群体环境下进行路径规划，还要考虑同一群体能智能体之间的规划共享与协作等。

处理游戏中的路径规划和移动问题需要注意以下几点：

（1）没有一种通用的方法适合解决所有的路径规划问题，这是因为路径规划问题的解决方法依赖于上下文环境的设定以及游戏软件对准确性、高效性和实时性的要求。

（2）在这些因素的约束下，使得采用图论中求取起点（A）到目标点（B）的最短通路等传统方法不适用游戏中的路径规划问题的求解答。比如，使用图论方法求得的最短路径也许要经过有敌人的危险区域，那将威胁智能体的安全；或者最短路径可能是一条通过山地的上坡或迎风方向的路径，那样的话通过该路径会消耗更多的体力和时间，综合考虑得不偿失。

（3）除了以上众多的约束因素，游戏中的路径规划问题还往往在较大的状态空间内求解，而游戏对实时交互性和效率的要求极高，在计算代价较大、耗费时间较长的情况下游戏必须要牺牲最优解而选择代价较小的次优解。

为了简化问题和降低计算代价，通常需要简化地图、预处理状态空间、预设路径和在群寻路时规划共享。

以前在算法和数据结构等课程中学习过的深度优先搜索或广度优先搜索算法都属于盲目搜索算法，它需要遍历巨大的状态空间，计算

代价非常大，在游戏软件中不实用。游戏中需要考虑很多上下文环境约束因素，因此，往往选用A*启发式搜索算法。A*算法的描述如下：

（1）添加开始节点到Open List。

（2）重复下面的过程：

①查找Open List中具有最小F值的节点。我们把它作为当前节点。

②把它放入Close List。

③对当前节点的所有相邻节点的每一个，重复如下操作：

i. 如果它不可行走，或者存在于Close List，忽略它，否则执行下面操作；

ii. 如果它不在OpenList，将它添加进去。以当前节点作为其父亲。记录这个节点的F，G和H值。

如果它已经在Open List了，检查到达那个节点的路径是否更优，以G值为测量值。更低的G值意味着更好的路径。如果找到，这个节点的父亲改为当前节点，并重新计算这个节点的G和F值。如果你保持Open List按F值排序的话，可能需要重新排序来解决这个变化。

④结束循环，当：

i. 将目标节点加入到Open List，此时路径已经找到。

ii. 没有找到目标节点，并且Open List是空的。此时，没有路径。

（3）保存路径。从目标节点往回走，从每个节点走到它的父亲节点，直到抵达开始节点，即是路径。

在算法步骤（2）的③中，从Open表中选出一个相邻的节点，然后重复早先的过程。但是我们选择哪一个呢？这就需要一个代价估算的机制来作为选择标准，通常，用一个公式表示：

F=G+H

G=从起始节点到当前节点的代价；H=从给定的节点到最终目标点的评估移动代价。这种方式通常称作试探法，这是一个猜测值。在找到路径之前，我们不知道实际的路径，因为途中有各种东西（墙，水，沼泽，敌人的攻击范围等等）。

我们需要的路径是这样生成的：反复的遍历Open表选择具有最小F值的节点。

当将H值永远设为0时，A*算法就蜕化为Dijkstra's Algorithm了，当将G值永远设置为0时，A*算法就蜕变为Best-First-Search algorithm了。

下面，我们通过一个实例说明A*算法的实现：

**例程11-1　A*算法的C++实现**

```
const intPATHFINDER_MAX_NODES=1000;
const intPATHFINDER_DEF_COST=10;
classCPathNode
{
private:
public:
```

```
    CPathNode*m_Parent;
    intm_iDistFromStartCost;
    intm_iDistFromGoalCost;
    intm_iTotalCost;
    intm_iEnumerated;
    intm_iX;
    intm_iY;
    void vReset(void);
    CPathNode();
};
classCPath
{
public:
    CPathNode    *m_Path[ PATHFINDER_MAX_NODES ];
    int           m_iNumNodes;
};
classCPathFinder
{
private:
    int    m_iStartX;
    int    m_iStartY;
    int    m_iEndX;
    int    m_iEndY;
    intm_iGoalFound;
    intm_iGoalNode;
    //Function pointer
    int(*iGetMapCost)(int iX, int iY);
public:
    CPathNode    *m_OpenList;
    CPathNode    *m_ClosedList;
    CPath        *m_CompletePath;
    int           m_iActiveOpenNodes;
    int           m_iActiveClosedNodes;
    CPathFinder();
```

```
    ~CPathFinder();
    void vSetCostFunction(int(*ptr)(int, int));
     void vSetStartState(int iStartX, int iStartY, int iEndX,
int iEndY);
            int   iGetGoalDistCost(int   iX,   int   iY,   int
iCost=PATHFINDER_DEF_COST);
           int   iGetStartDistCost(int   iParentStartCost,   int
iCost=PATHFINDER_DEF_COST);
    boolbFindPath(int iMaxNodes=PATHFINDER_MAX_NODES);
    boolbOpenNodes(CPathNode*node);
    boolbDoesNotExistOnOpenList(int iX, int iY);
    boolbDoesNotExistOnClosedList(int iX, int iY);
          int   iAddNodeToList(int   iList,   int   iX,   int   iY,
CPathNode*ParentNode, int iCost=PATHFINDER_DEF_COST);
    int iFindCheapestNode();
    void vResetPath(void);
};
```

具体实现请参照本书所附代码。

其运行结果如图11-2所示。

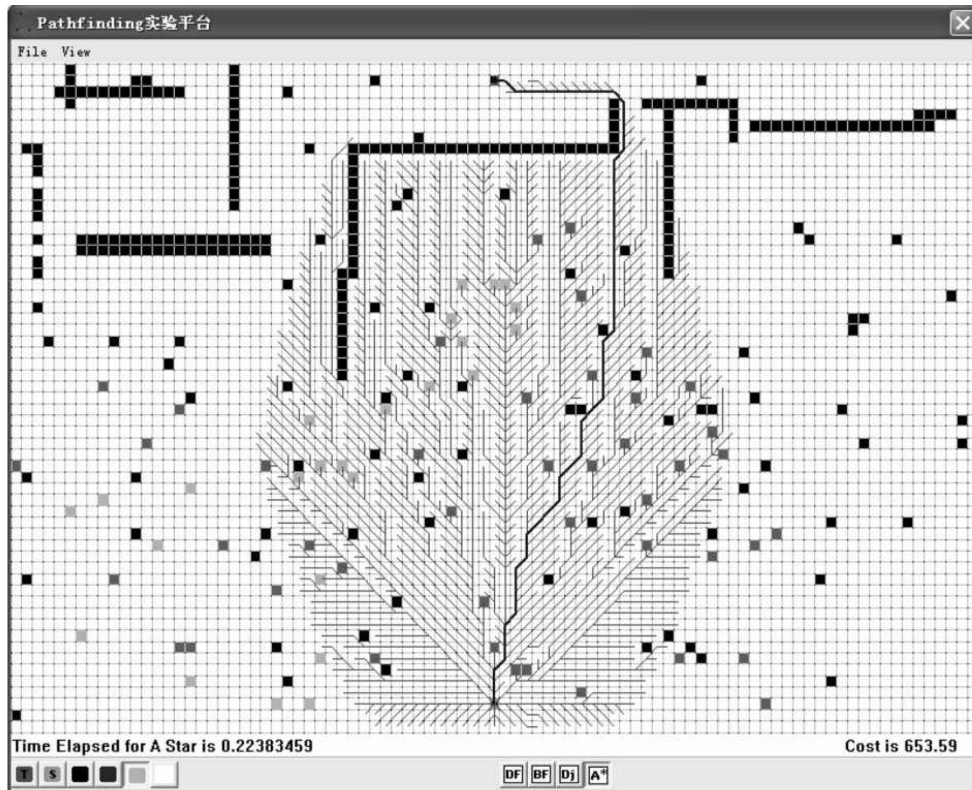



图11-2　A*算法运行过程

# 11.4　有限状态机

有限状态机技术在游戏开发中被得到了广泛的应用，因为它简单、高效和易于扩展。图11-3给出了一个游戏中常见的有限状态机示例。如果使用基于规则的AI系统，则它是上下文无关的，当同一条件集满足多个决策时出现二意性；而有限状态机（FSM）是上下文相关的，可以消除二意性。但FSM过于确定，不过近来出现了基于模糊逻辑的FSM技术。

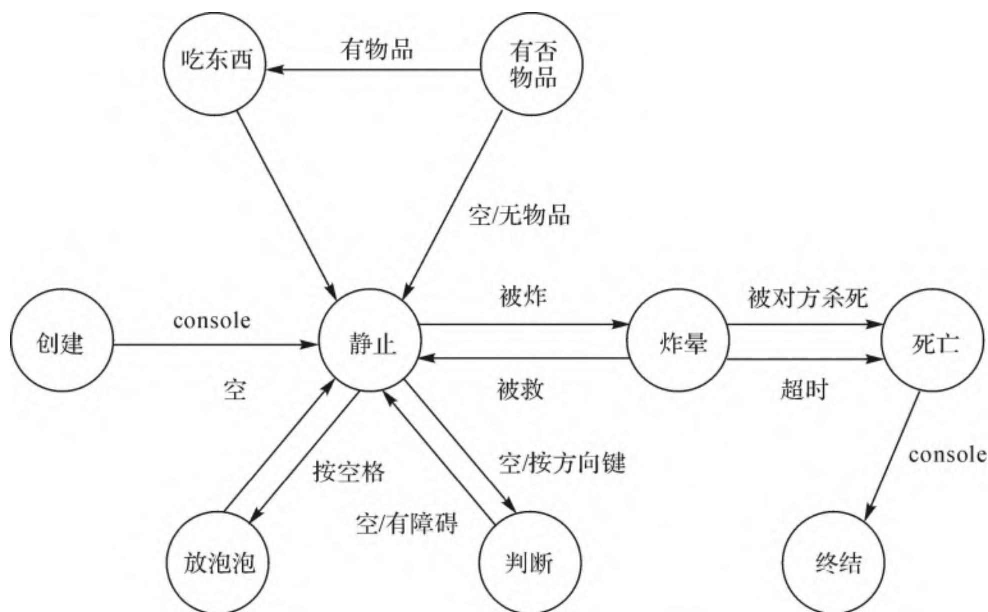



图11-3　游戏中的有限状态图

FSM在游戏中主要有以下两种模型：

Mealy Machine模型：

“A Mealy machine is an FSM whose actions are performed on transi-tions”

Moore Machine模型：

“A Moore machine's actions reside in states”

Moore Machine模型比较适用于在游戏中建模和实现。游戏中的每个智能体都由它自己的FSM控制，由于智能体众多且逻辑复杂，所以在实际的开发中有必要编写可视化工具辅助设计每个智能体的状态机。

FSM的技术实现主要有两种：

（1）基于硬编码的实现（Code-Based FSM）

- 硬编码实现，即游戏的逻辑模块中充斥着大量的if-else和switch-case语句。
- 基于宏定义的编码实现，即采用C/C++中的大量宏（Macro）技术设计一种类似于MFC框架实现的代码自动生成程序。
- 采用面向对象技术把状态与状态跃迁函数抽象和封装。

（2）基于数据驱动的脚本语言实现（Data-Driven Script Language FSM）

- 即与脚本系统模块整合将FSM的特征体现在脚本编译器和脚本解释器中。

若按照面向对象的方法进行抽象和分析，效率代价比较大（每个状态通常都被抽象为一个状态类的对象），而宏实习则代码较为紧凑，效率较高。

例程11-2的代码是一个有限状态机以面向对象方法的C++实现，为了让读者更专注于系统有限状态机本身，只提供一个最简单的实现。

**例程11-2　有限状态机的实现**

```
/**状态类*/
template<class entity_type>
classState
{
public:
    virtual~State(){}
    //thiswillexecutewhen the state is entered
    virtualvoidEnter(entity_type*)=0;
```

```
    //this is the statesnormalupdate function
    virtualvoidExecute(entity_type*)=0;
    //thiswillexecutewhen the state is exited
    virtualvoidExit(entity_type*)=0;
              //executes  if  the  agentreceivesamessage  from
themessagedispatch-er
      virtualboolOnMessage(entity_type*, constTelegram&)=0;
};
/**状态机类*/
template<class entity_type>
classStateMachine
{
private:
      entity_type*m_pOwner; //a pointer to the agentowns this
instance
    //the record ofcurrentstate the agent is in
    State<entity_type>*m_pCurrentState;
    //a record of the laststate the agentwas in
    State<entity_type>*m_pPreviousState;
    //this is called every time the FSM is updated
    State<entity_type>*m_pGlobalState;
public:
StateMachine(entity_type*owner):m_pOwner(owner),
m_pCurrentState
    (NULL), m_pPreviousState(NULL), m_pGlobalState(NULL);
    virtual~StateMachine(){}
    //use thesemethods to initialize the FSM
                  void    SetCurrentState(State<entity_type>*s)
{m_pCurrentState=s;}
                   void    SetGlobalState(State<entity_type>*s)
{m_pGlobalState=s;}
                 void    SetPreviousState(State<entity_type>*s)
{m_pPreviousState=s;}
    //call this to update the FSM
```

```
voidUpdate()
{
    //if a globalstate exists, call it's executemethod, else do noth-ing
        if(m_pGlobalState)          m_pGlobalState->Execute(m_pOwner);
  //same for the current state
          if(m_pCurrentState)      m_pCurrentState->Execute(m_pOwner);
}
boolHandleMessage(constTelegram&msg)const
{
              if(m_pCurrentState&&m_pCurrentState->OnMessage(m_pOwner, msg)){
      //firstsee if the current state is valid
      //and that itcan handle themessage
      return true;
  }
            else  if(m_pGlobalState&&m_pGlobalState->OnMessage(m_pOwner, msg))
  {
          //if not, and if a global state has been implemented
      //send themessage to the global state
      return true;
  }
  else
      return false;
}
//change to a new state
voidChangeState(State<entity_type>*pNewState)
{
    assert(pNewState&&"<StateMachine::ChangeState>:trying to as-sign null state to current");
```

```
        //keep a record of the previous state
        m_pPreviousState=m_pCurrentState;
        //call the exitmethod of the existing state
        m_pCurrentState->Exit(m_pOwner);
        //change state to the new state
        m_pCurrentState=pNewState;
        //call the entrymethod of the new state
        m_pCurrentState->Enter(m_pOwner);
    }
    //change state back to the previous state
    voidRevertToPreviousState()
    {
        ChangeState(m_pPreviousState);
    }
    //returns true if the current state's type is equal to the
class's
    //passed as a parameter.
    bool isInState(constState<entity_type>&st)const
    {
        if(typeid(*m_pCurrentState)==typeid(st))
            return true;
        else
            return false;
    }
        State<entity_type>*    CurrentState()    const{return
m_pCurrent-State;}
                    State<entity_type>*          GlobalState()
const{returnm_pGlobal-State;}
     State<entity_type>*  PreviousState()const{return m_pPrevi-
ousState;}
     //only everused during debugging to grab the name of the
current state    std::stringGetNameOfCurrentState()const
    {
        std::string s(typeid(*m_pCurrentState).name());
```

```
        //remove the'class'part from the frontof the string
        if(s.size()>5)
            s.erase(0, 6);
        return s;
        }
};
```

图11-4中我们使用有限状态机技术实现一个简单的足球人游戏，完整代码见本书所附带的光盘，该游戏运行如下。

图11-4　基于有限状态机的足球人游戏

# 11.5　脚本技术

随着引擎复杂度的增加，硬编码最终会遇到问题。硬编码是高效的，但通常灵活性不足，任何增加和修改都要重新编译整个源代码。这种情况下，最好将经常改动的部分外化，可以从内核引擎周围的另一模块中运行，这些模块可以用特定编程语言编写。这就是游戏脚本

（Script）。脚本是为了简化某种特定程序的复杂任务而设计的一种编程语言，它是当今大型团队编制大型游戏时常用的方法。脚本的使用肯定会有一定的性能影响，但其长处是扩展性和灵活性，而不是效率。

可以说每一款游戏软件中都不同程度的使用了脚本技术，游戏脚本主要在以下几个方面使用：

（1）用于配置文件。

（2）用于逻辑控制。

（3）作为AI脚本。

（4）处理界面触发事件。

在更为复杂的环境中，比如大量玩家参与的网络游戏中，脚本语言是一种强有力的工具。用来处理多处理器的操作细节都被完全隐藏在脚本语言中。发送网络事件给客户是一件轻而易举的事情。脚本语言甚至能自动完成对象状态的存储操作。图11-5表示了游戏脚本系统在游戏引擎中与其他各个模块的关系。

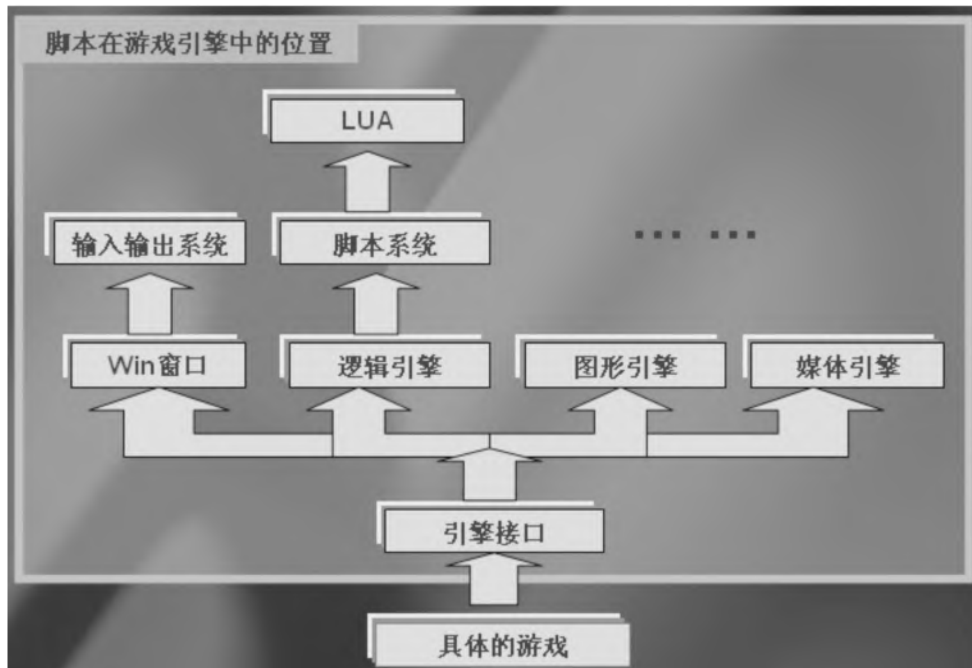



图11-5　脚本技术在游戏引擎中的作用

针对不同类型的游戏开发以及游戏开发的不同方面，有以下几种常用的基本脚本系统可供我们选择：

（1）基于命令的脚本系统。

（2）动态链接的脚本模块。

（3）内嵌现有的脚本语言。

（4）开发自己的脚本语言。

下面我们将分别以实例来介绍如何使用这些脚本技术。

基于命令的脚本系统非常像LOGO语言一样，它们全部由接受一个到两个参数的针对具体程序的命令组成。例如，RGP游戏中使用脚本调用一些针对该游戏的功能函数，从而完成某些常见任务，比如在游戏

世界中移动游戏玩家的位置、获得物品、与其他游戏角色进行对话等。我们来看一段这样的脚本代码：

```
MovePlayerTo 10, 20
PlayerTalk"Something ishidden in these bushed…"
PlayAnimation SEARCH_BUSHES
PlayerTalk"It's the red sword!"
GetItem RED_SWORD
```

尽管基于命令的脚本系统缺少变量定义、表达式、分支结构、循环体、函数等常见的语言结构，从而在一定程度上限制了他们的灵活性，但是他们仍然可以将线性任务转化为常见的宏（macro）。不但游戏软件中经常这么做，甚至向MS Word和Photoshop这样的商业应用软件也允许用户用类似的宏来记录它们的行为，从而使这些行为可以在以后重现。

下面来看一个使用基于命令的脚本系统所实现的游戏程序片段实例：

**例程11-3　脚本技术的简单实现**

```
#defineMAX_SOURCE_LINE_SIZE          4096       //脚本文件的最大行数
#defineMAX_COMMAND_SIZE              64         //一个指令的最大长度
#defineMAX_PARAM_SIZE                1024       //参数的最大长度

//————命令名——————————————————————————
#defineCOMMAND_DRAWBITMAP            "DrawBitmap"
#defineCOMMAND_PLAYSOUND             "PlaySound"
#defineCOMMAND_PAUSE                 "Pause"
#defineCOMMAND_WAITFORKEYPRESS       "WaitForKeyPress"
#defineCOMMAND_FOLDCLOSEEFFECTX      "FoldCloseEffectX"
#defineCOMMAND_FOLDCLOSEEFFECTY      "FoldCloseEffectY"
```

```
#defineCOMMAND_EXIT "Exit"

//———全局变量——————————————————————————————
char**g_ppstrScript=NULL;                //指向当前脚本的指针
intg_iScriptSize;                        //当前脚本语句的长度
intg_iCurrScriptLine;                    //当前脚本行的行号
intg_iCurrScriptLineChar;                //当前脚本行的当前字符

//———函数原型——————————————————————————————
intLoadScript(char*pstrFilename);              //载入脚本文件
voidUnloadScript();                            //释放脚本文件
voidResetScript();                             //设置脚本文件的位置
voidRunScript();                               //运行脚本命令
voidGetCommand(char*pstrDestString);           //获取脚本命令
intGetIntParam();                              //获取整数参数
voidGetStringParam(char*pstrDestString);       //获取字符串参数
```

其运行效果如图11-6。

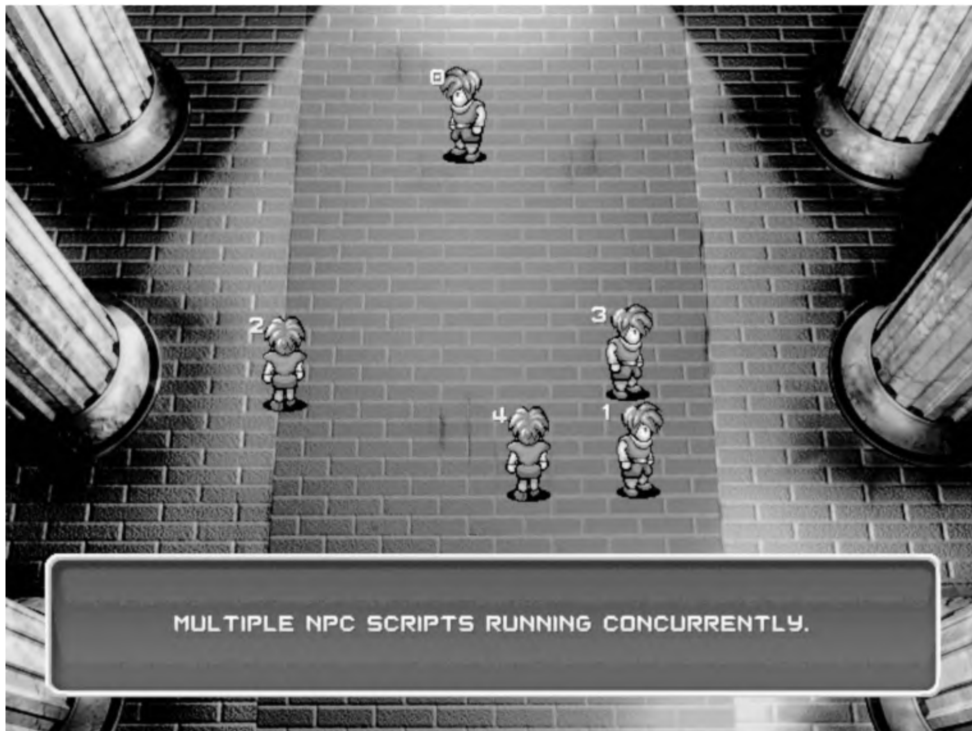


图11-6　简单脚本技术的应用

当一个复杂的脚本在一个虚拟机上运行时，它的运行速度明显要比直接使用纯粹的机器码在CPU上运行慢得多。为了解决这个问题，许多游戏采用动态链接脚本模块。这些模块其实就是一些C/C++程序段，它们像游戏本身一样被编译成纯粹的机器码并且在运行时被链接和载入。这种脚本技术在Quake和Half-Life游戏中非常流行。但在多人在线的网络游戏中使用这种脚本技术，主机几乎不能控制这些脚本的行为，这使得它们自身以及整个系统都变得非常脆弱，很容易遭受黑客攻击。

图11-7为Half-Life中使用这种脚本技术控制游戏角色行为的游戏抓图。

图11-7　脚本技术在枟半条命枠游戏中的应用

考虑到开发时间和成本的限制，更多时候都是将现有的成熟脚本语言内嵌到我们的游戏程序中实现相应的脚本功能。目前在游戏中用的较多的脚本语言主要有Lua、Ruby和Python。

在使用这些脚本语言时，首先需要对它们进行包装，从而可以方便的集成到我们的游戏程序使用。有关这些语言使用的细节本书不再赘述，在它们的官方网站上都有详细的文档和使用教程。在此我主要通过一个实例来说明如何在游戏中使用Lua语言。使用Ruby或Python时，方法也类似如此。

Lua（葡萄牙语，意为月亮）是一种轻量级的强大的扩展语言，用纯ANSIC编写，实现了垃圾收集、反射、面向对象等机制。Lua是由巴西里约热内卢天主教大学计算机系的Roberto Ierusalimschy等人于1993年开发的。Lua是所有动态语言中间平均效率最高的一个。现在不仅发展出了解释器，还发展出了编译器。游戏开发里Lua是使用最广泛的脚本语言，它包括很多良好的语言特性集，同时又很容易学习。除此之外，Lua与C/C++结合很紧密。Lua可以作为嵌入式语言使用，通过自身的API和宿主程序进行通信和交互。通过Lua的C库执行才是游戏中常用的方式。作为一种自由软件的脚本语言，Lua由于在游戏界日益成为主流脚本语言，引起了业界的广泛关注。

在设计游戏引擎时，可以这样包装Lua脚本语言使之无缝的与引擎的其他模块集成（如图11-8所示）。

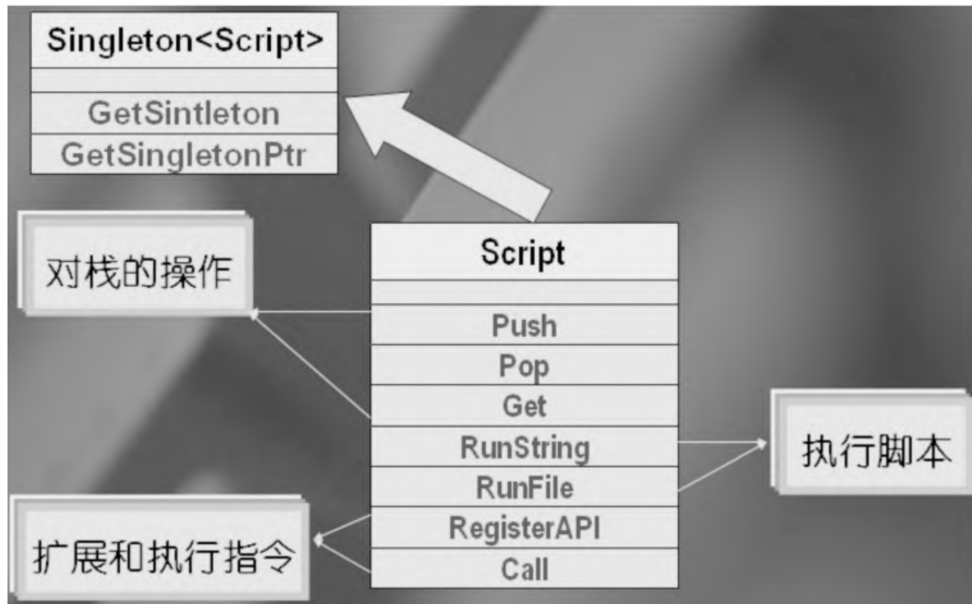



图11-8　脚本文件的执行原理

例如，通过Lua脚本语言处理游戏引擎GUI系统中的界面触发事件，如图11-9所示。

❖处理界面触发事件

一个界面对应两个文件：
属性文件
脚本文件

```
// Scn.stb            --Scn.sct
[Root]                Scn={}
MyWin=Window          function Scn:Init()
                      end
[MyWin]
MyBtn=Button          function Scn:Destroy()
x=0                   end
...
                      function Scn:MyBtn_Click()
[MyBtn]                   … …
x=10                  end
```

图11-9　脚本技术在游戏GUI系统上的应用

例程11-4使用一段Lua脚本语言来控制游戏中的精灵行为，该Lua脚本如下所示：

**例程11-4　Lua脚本的使用实例**

```
————Constants——————————————————————————————
ALIEN_COUNT            =12;           //屏幕中出现的精灵个数
MIN_VEL                = 2;           //最低速度
MAX_VEL                = 8;           //最高速度
ALIEN_W IDTH           = 128;         //精灵宽度
ALIEN_HEIGHT           = 128;         //精灵高度
HALF_ALIEN_W IDTH      = ALIEN_W IDTH/2;
HALF_ALIEN_HEIGHT      = ALIEN_HEIGHT/2;
ALIEN_FRAME_COUNT      = 32;          //动画帧数
ALIEN_MAX_FRAME        = ALIEN_FRAME_COUNT-1;
ANIM_TIMER_INDEX       = 0;          //动画的时间编号
MOVE_TIMER_INDEX       = 1;          //移动的时间编号
————GlobalVariables————————————————————————
Aliens = {};                         //精灵
CurrAnimFrame = 0;                   //动画中的当前帧
————Functions——————————————————————————————
//初始化程序
function Init()
        forCurrAlienIndex=1, ALIEN_COUNT do
        localCurrAlien={};
        ——设置X, Y坐标
        CurrAlien.X=GetRandomNumber(0, 639-ALIEN_W IDTH);
        CurrAlien.Y=GetRandomNumber(0, 479-ALIEN_HEIGHT);
        ——设置速度的X, Y分量
        CurrAlien.XVel=GetRandomNumber(M IN_VEL, MAX_VEL);
        CurrAlien.YVel=GetRandomNumber(M IN_VEL, MAX_VEL);
        ——设置旋转方向
        CurrAlien.SpinDir=GetRandomNumber(0, 2);
        Aliens [CurrAlienIndex]=CurrAlien;
```

```
    end
end
//下一帧的贴图操作
functionHandleFrame()
    BlitBG();
    forCurrAlienIndex=1, ALIEN_COUNT do
        localX=Aliens [CurrAlienIndex].X;
        localY=Aliens [CurrAlienIndex].Y;
    localSpinDir=Aliens [CurrAlienIndex].SpinDir;
        ifSpinDir==1 then
            FinalAnimFrame=ALIEN_MAX_FRAME-CurrAnimFrame;
        else
            FinalAnimFrame=CurrAnimFrame;
        end
        BlitSprite(FinalAnimFrame, X, Y);
    end
    BlitFrame();
    ifGetTimerState(ANIM_TIMER_INDEX)==1 then
        CurrAnimFrame=CurrAnimFrame+1;
        ifCurrAnimFrame>=ALIEN_FRAME_COUNT then
            CurrAnimFrame=0;
        end
    end
    ifGetTimerState(MOVE_TIMER_INDEX)==1 then
        forCurrAlienIndex=1, ALIEN_COUNT do
            localX=Aliens [CurrAlienIndex].X;
            localY=Aliens [CurrAlienIndex].Y;
            localXVel=Aliens [CurrAlienIndex].XVel;
            localYVel=Aliens [CurrAlienIndex].YVel;
            X=X+XVel;
            Y=Y+YVel;
            Aliens [CurrAlienIndex].X=X;
            Aliens [CurrAlienIndex].Y=Y;
                if X>640-HALF_ALIEN_W IDTH or X<-HALF_ALIEN_W
```

```
IDTH then
                XVel=-XVel;
            end
            ifY>480-HALF_ALIEN_HEIGHT orY<-HALF_ALIEN_HEIGHT
                thenYVel=-YVel;
            end
            Aliens [CurrAlienIndex].XVel=XVel;
            Aliens [CurrAlienIndex].YVel=YVel;
        end
    end
end
```

该lua脚本的运行效果如图11-10所示。

图11-10　Lua脚本语言的使用

自己从头开发一个新的高性能的脚本语言是一项复杂而耗时的工作。但是，创造出了脚本语言带来的收益往往会大大超出我们的付出。

# 11.6　群聚技术

群聚技术（Flocking，Swarming，Herding）是由Craig Reynolds在1987年的SIGGRAPH会议上发表的论文Flocks，Herds and Schools: A Distrib-uted Behavioral Model中首次提出的。在该论文中，Reynolds定义了物群模拟的3种基本准则（或称导航行为，Steering Behavior）:

（1）分离（Separation）：物群中的每个智能体之间保持一定的距离，这样可以避免互相穿透或重叠，其效果如图11-11所示。

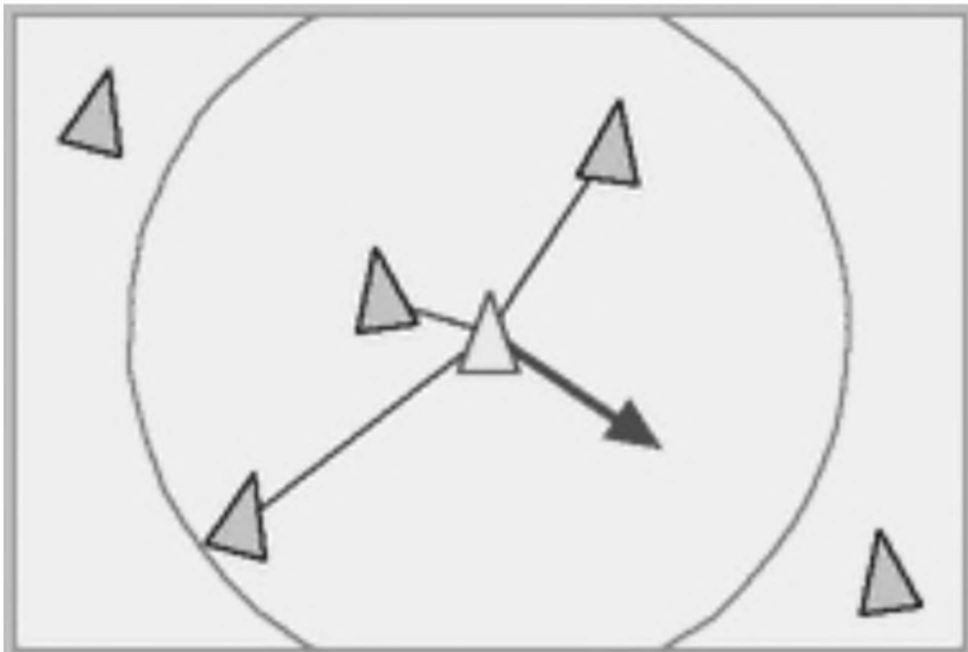


图11-11　分离（Separation）

（2）列队（Alignment）：物群中的每个智能体与其他智能体保持相同的航向，在实现中通常都让每个智能体的航向所有智能体的平均航向逼近，其效果如图11-12所示。

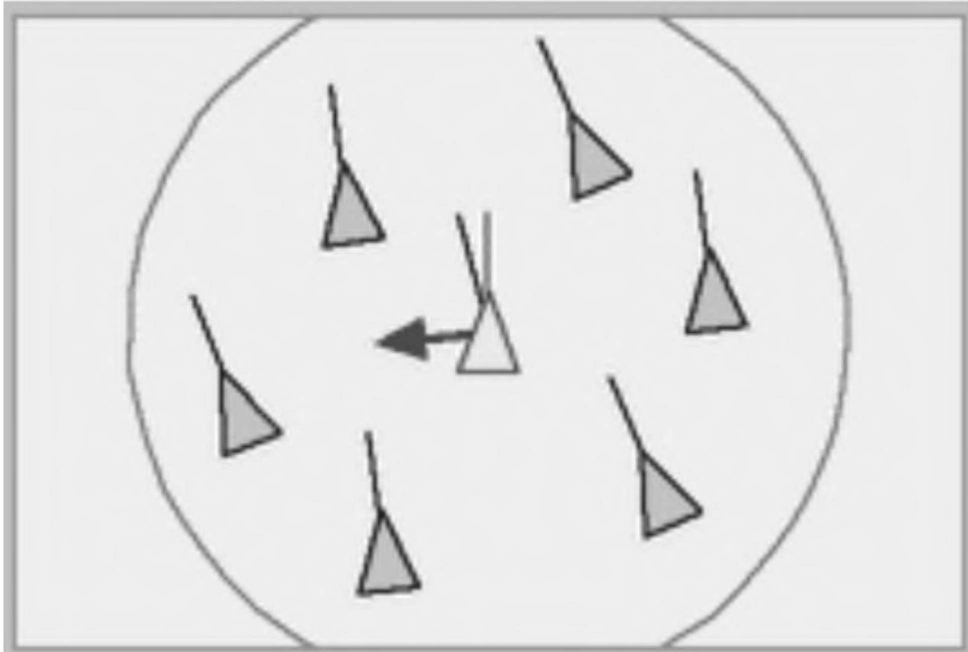


图11-12　列队（Alignment）

（3）内聚（Cohesion）：物群中的每个智能体之间的距离不能拉的太远，这样可以避免其离群掉队，在实现中通常都让每个智能体向着所有智能体中心位置的平均位置逼近，其效果如图11-13所示。

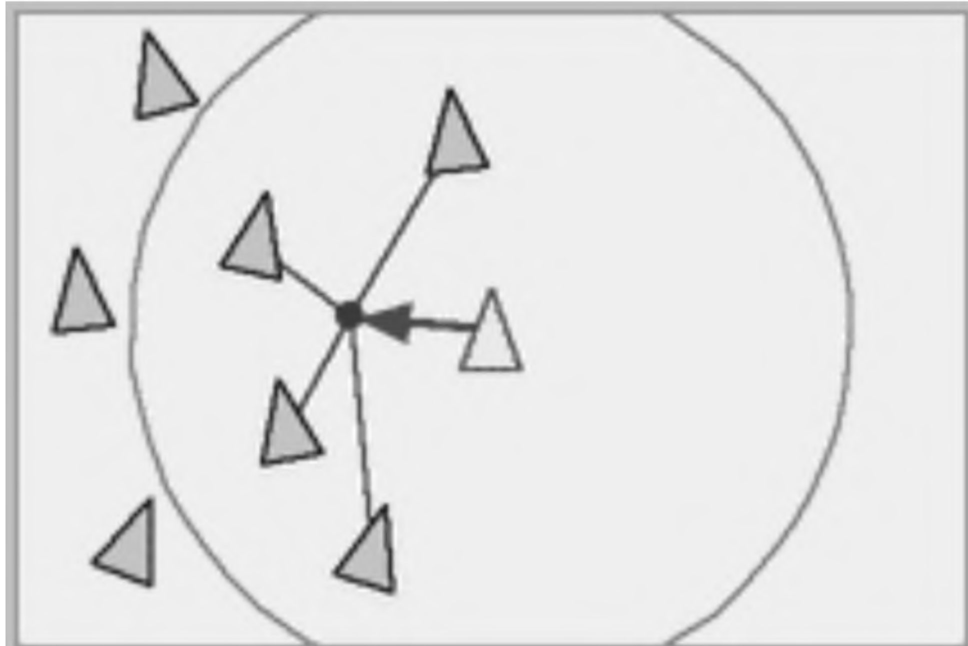


图11-13　内聚（Cohesion）

尽管这3条约束条件都很简单，但其表现效果已经很逼真了。游戏开发者们都在这3条基本准则的基础上，结合具体游戏上下文场景的特点和要求，又制定了一些扩展准则，这些扩展准则中使用最频繁的理所当然的应该是如何规避开障碍物和敌人，即规避（Avoidance）：避开障碍物和敌人，其效果如图11-14所示。

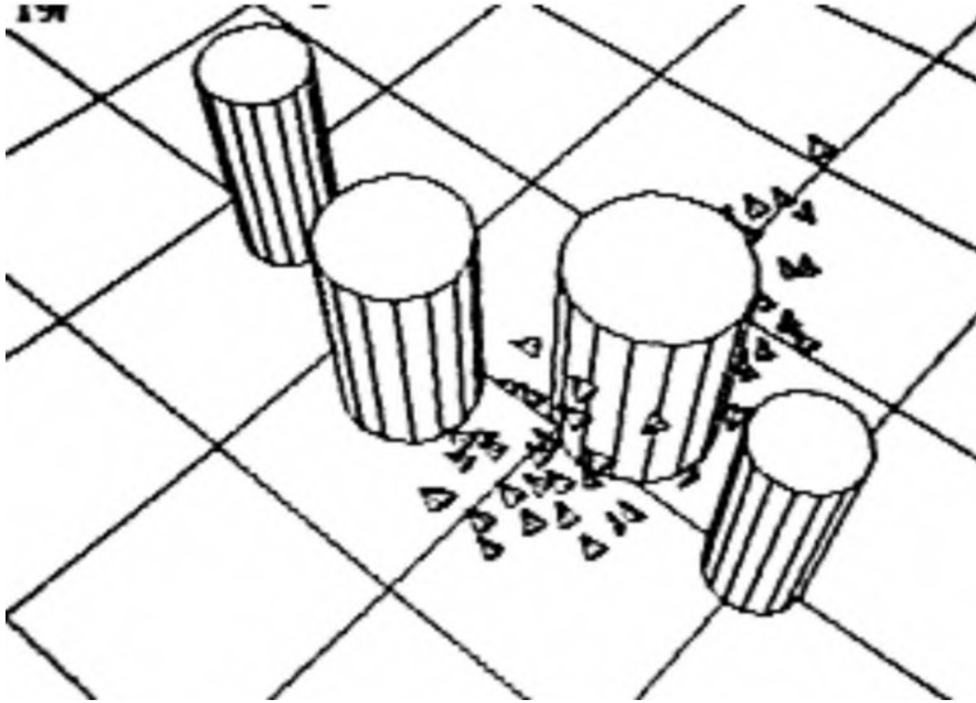


图11-14　规避（Avoidance）

增加这一准则之后，物群就能具备初步应未知环境的能力，使得物群中的所有成员能在未知甚至动态的环境中避开障碍物和敌人，并动态的若即若离的作为一个整体行动。实践表明，这种技术对视频游戏非常实用，最典型的应用就是RTS游戏中的部队编队和RPG游戏中的智能体群的行为模拟。

群聚技术在实现时往往需要与物理引擎中的动力学模块结合起来。本书光盘中包含了一个使用群聚技术的代码实例，其运行结果如图11-15，其中的圆圈表示随机生成的障碍物，大三角形为敌人，小三角形为物群中的智能体。

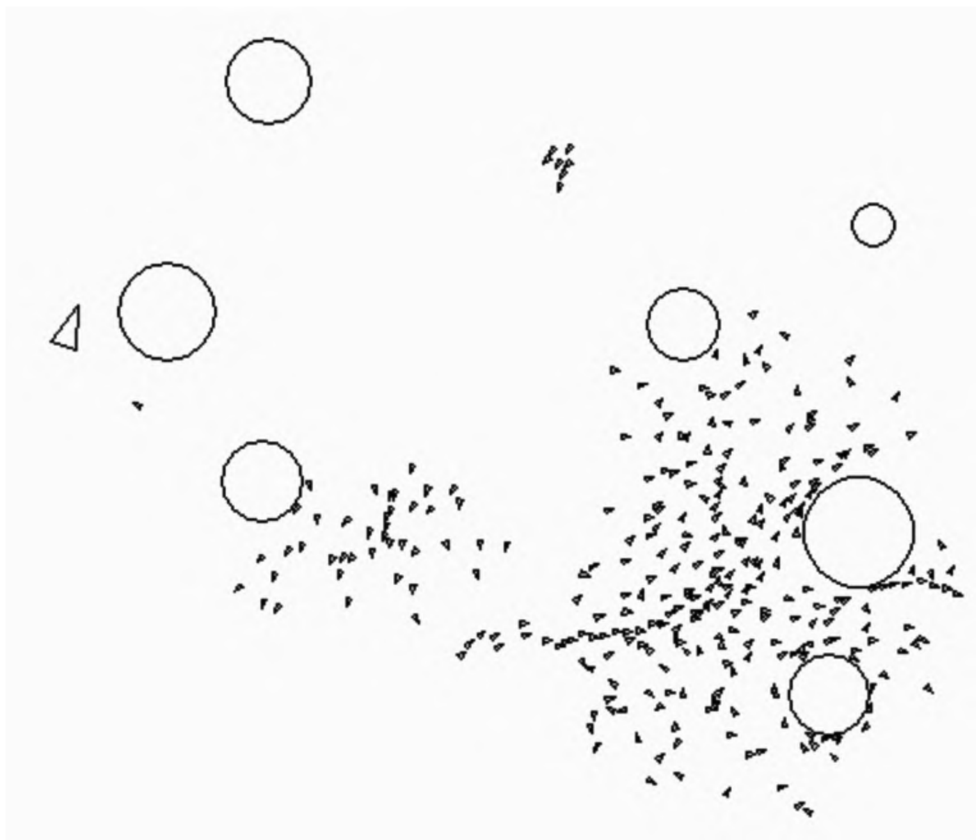


图11-15　群聚技术实例

## 11.7　遗传算法

遗传算法的工作过程其实就是模拟生物的进化过程。首先，应该确定一种编码方法，使得你的问题的任何一个潜在可行解都能表示成为一个“数字染色体”。然后，创建一个由随机染色体组成的初始群体，其中的每个染色体代表一个不同的后选解，并在一段时期中，以培育适应性最强的个体的办法让它进化。在此期间，染色体的某些位置上要加上少量变异。经过许多代的进化后，可能会收敛到一个解。遗传算法不确保一定能收敛到一个解，并且如果得到解，也不确保一定是最优解。但使用遗传算法有一个很大的优点：我们不必知道怎样

去具体解决一个问题，仅需设计出一种对可行解进行编码的方式，从而就可以利用遗传算法机制编写程序来尝试解决问题。

当设计出代表可行解的染色体的编码方式后，就可以按照如下算法步骤实现遗传算法：

（1）检查每个染色体，并赋给其一个适应性数值量化它解决问题的性能。

（2）从当前的染色体集中选出2个成员，选出的概率正比于该染色体的适应性值。

（3）按照预先设定的杂交率对选出的染色体对的编码进行杂交（crossover）。

（4）按照预先设定的变异率对选出的染色体编码的相应位上的值进行翻转（flip）。

（5）重复（2）~（4）步，直到又一个新的染色体群被生成。

算法由步骤（1）~（5）的一次循环被称为一个世代（generation），而把整个的迭代循环称为一个时代（epoch）。为了使读者能够理解遗传算法，下面我们应用遗传算法来求解迷宫中的寻路问题。该迷宫的场景非常简单：它左边有一个入口，右边有一个出口，而出口和入口之间随机散落着一些障碍物。在迷宫的入口处放置一个名为Bob的虚拟人物，然后他将绕过障碍物寻找迷宫的出口。我们用一个二维整数数组表示迷宫，其中0代表可通过的位置，1代表墙壁或障碍物，5为入口，8为出口。

下面来设计一种染色体编码方案，使得该问题能被遗传算法解决。我们发现，每次Bob只能朝着东、南、西、北这四个方向向前移动一个位置（该位置不是墙壁或障碍物）。所以，可以为这四个方向进行编码，这些编码代表着Bob所能采取的各个行动，即代表着遗传算法中的染色体。编码后的染色体应该就是分别代表这4个方向信息的字符串。如果用二进制数为这4个方向编码，每个方向的编码只需要两个二进制位就可以了，如表11-2所示。

表11-2　染色体编码方案与Bob选择移动方向的关系

| 二进制编码 | Bob能选择的移动方向 |
| --- | --- |
| 00 | 向北 |
| 01 | 向南 |
| 10 | 向东 |
| 11 | 向西 |

这样，如果得到了一个随机的二进制字符串，就能根据它译出Bob行动时所遵循的一系列方向。例如染色体编码序列：

111110011011101110010101所代表的基因就是：

11，11，10，01，10，11，10，11，10，01，01，01

从而，该染色体序列所代表的Bob的动作序列就是：

向西移动→向西移动→向东移动→向南移动→向东移动→向西移动→向东移动→向西移动→向东移动→向南移动→向南移动→向南移

动

让Bob按照这个动作序列在迷宫地图上移动，每当Bob选择一个移动方向时，都计算出Bob沿着该方向所能到达的最远点位置（即停止于该方向的墙壁或障碍物前）。根据该最远点位置，返回一个数值来量化代表该方向的染色体的适应性，该数值反比于Bob最终位置离出口的距离，即他所到达的位置距离出口愈近，奖励给他的适应性数值就愈高。如果Bob实际已到达了出口位置，他将得到满分1，这时循环就会自动结束，这时就已经得到了一个解，但这个解不一定是最优解。

对上述遗传算法的代码实现，请参看本书所附带的代码，其运行效果如图11-16所示。

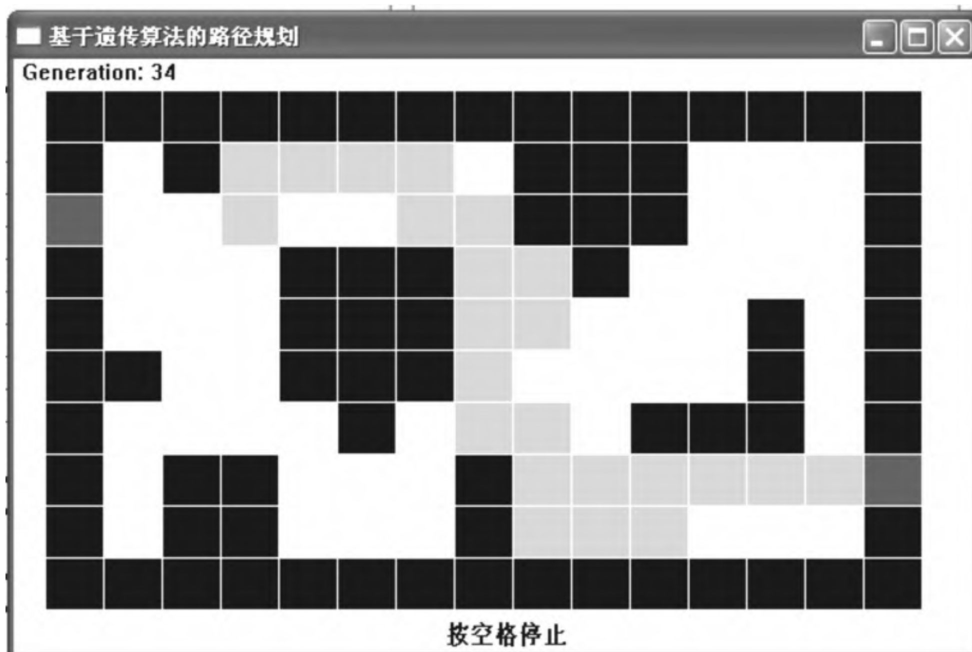



图11-16　基于遗传算法思想的寻路算法实例

需要说明的是，运行该程序，有时可能得不到一条通往出口的路径，我们会发现Bob一直在一个局部区域不停地走来走去。这是因为染色体群过快的收敛于一个特殊类型的染色体，该染色体导致染色体内的成员间的差异性消失，从而使得仅仅依靠变异本身已不能去发现一个解。

# 11.8　神经网络

人工神经网络通过模拟生物大脑的工作原理来求解复杂问题。因为生物的大脑是由许多神经细胞组成的，所以模拟大脑的人工神经网络ANN也是由许多称为人工神经细胞（Artificial neuron，也称人工神经元）的细小结构模块组成。我们可以认为人工神经细胞是生物体真实神经细胞的一个简化版，但采用了计算机来模拟实现。一个人工神经网络需要使用多少个人工神经细胞，其差别可以非常大，这完全取决于这些人工神经网络准备实际用来做什么。

下图为一个人工神经网络的示意图，其左边的几个灰底圆中所标字母w代表浮点数，称为权重（weight，或权数、权值）。进入人工神经网络的每一个输入（input）都与一个全重w相联系，正是这些权重将决定神经网络的整体活跃性。如果权重被设置为正数，就会有激发（excitory）作用；如果权重被设置为负数，就会有抑制（inhibitory）作用。当输入信号进入神经细胞时，它们的值将与他们对应的权重相乘，作为该神经细胞的输入。图中的大圆圈代表神经细胞的细胞核，细胞核把所有这些经过权重调整后的输入全部加起来，形成单个的激励值（activation value），用于激发神经细胞进入兴奋态（输出1）或不进入兴奋态（输入0）。激励值也可正可负，

如果它超过某个阈值，就会产生一个值为1的信号输出；反之，就会产生一个值为0的信号输出。

大脑里的生物神经细胞和其他的神经细胞是相互连接在一起的。为了创建一个人工神经网络。人工神经细胞也要以同样的方式相互连接在一起。为此可以有许多不同的连接方式，其中最容易理解也是广泛应用的，就是如下图所示的前馈网络（feedforward network）。

前馈网络的每一层神经细胞的输出都向反馈（feed）给它们的下一层，直至获得整个网络的输出。

下面将通过一个实例来帮助读者理解神经网络技术在Game AI中的应用。本例将使用神经网络来控制智能扫雷机的行为。该扫雷机工作在一个很简单的环境中，里面只有扫雷机以及随机散布的许多地雷。我们将要创建的神经网络使用无监督学习（unsupervised learning）模式自行进行演化去寻找地雷。为了实现这一功能，网络的权重将被编码到基因组中，并用一个遗传算法来演化它们。

该实例的具体实现见本书附带的代码，其运行结果如图11-17所示。

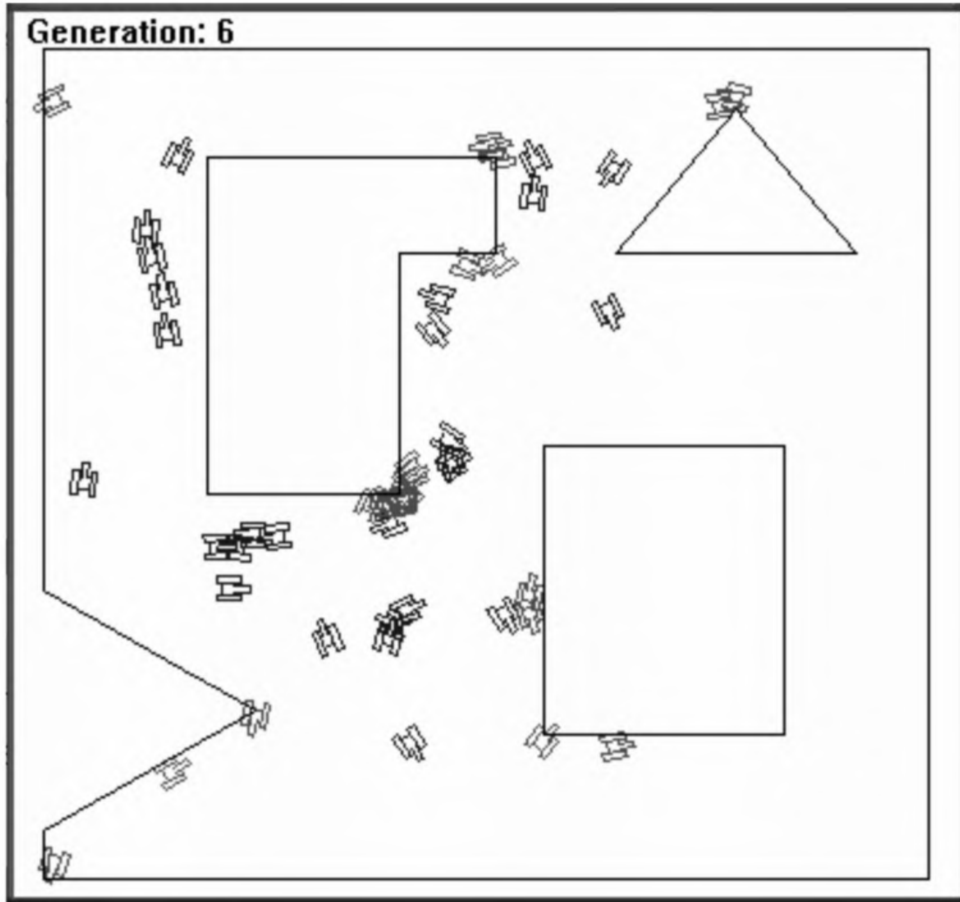
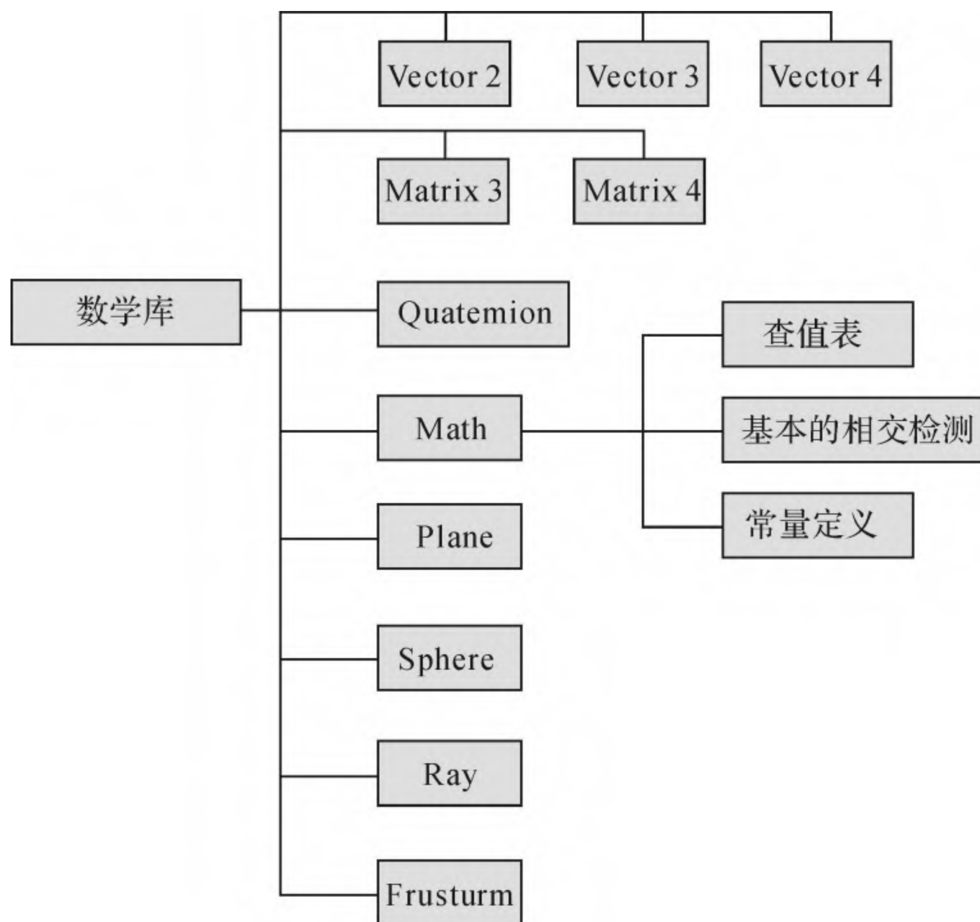



图11-17　神经网络在游戏中的应用实例

# 附录

## Ⅰ. 数学库

数学库作为游戏引擎的底层模块，需要支持向量、矩阵、四元数、平面、球、射线、平截锥体以及常值表、相交检测等。该库必须注重运算效率，能满足游戏设计的需要。




图附-1　游戏引擎数学库的组成

## Ⅰ.Ⅰ 常用计算函数

对于最常用的数学函数，我们完全可以使用<math.h>中定义的函数，如取整、对数、三角函数等等。最常用的函数如下所示：

```
/*Trigonometric functions*/
  float acosf(float);
  float asinf(float);
  float atanf(float);
  float atan2f(float, float);
  float cosf(float);
  float sinf(float);
  float tanf(float);
/*Hyperbolic functions*/
  float coshf(float);
  float sinhf(float);
  float tanhf(float);
/*Exponentialand logarithmic functions*/
  float expf(float);
  float logf(float);
  float log10f(float);
  float modff(float, float*);
/*Power functions*/
  float powf(float, float);
  float sqrtf(float);
/*Nearest integer, absolute value, and remainder functions*/
  float ceilf(float);
  int abs(int);
  float fabsf(float);
  float floorf(float);
  float fmodf(float, float);
```

基于游戏实时性的特殊要求，我们有必要在游戏初始化时建立一个三角函数查值表，以便于游戏中的快速计算。以下函数为三角快速查值函数：

```
float acosFast(float);
float asinFast(float);
float atanFast(float);
float cosFast(float);
float sinFast(float);
float tanFast(float);
```

**例程附-1　三角函数查值表的创建和使用**

```
voidMath::BuildSinTable(unsigned int iTableSize){
    c_TrigTableSize=iTableSize;        //查值表的大小，即精度
    floatangle;
    for(unsigned int i=0;i<c_TrigTableSize;++i){
        angle=TWO_PI*i/c_TrigTableSize;
        c_SinTable[i]=sinf(angle);     //逐项赋值
}
floatMath::SinTable(float fRadian){
    int idx;
    if(fValue>=0)
        idx=int(fRadian*c_TrigTableFactor)%c_TrigTableSize;
    else
        idx=c_TrigTableSize-(int(-fRadian*c_TrigTableFactor)
            %c_TrigTableSize)-1;
    return c_SinTable[idx];
}
```

## Ⅰ.Ⅱ　向量—Vector2

向量分Vector2、Vector3、Vector4三种。其中，Vector2处理2维平面事务，Vector3和Vector4则处理3维世界的事务。

Vector2的数据成员为float Vector2::x和float Vector2::y，分别表示一个二维向量X方向和Y方向的分量。表附-1中列出了二维向量类需要支持的操作，这些操作的具体实现可以参考随书光盘中的代码。

表附-1　Vector2需要实现的功能

| 函数声明 | 函数功能描述 |
|---|---|
| Vector2() | 缺省的构造函数 |
| Vector2( float InX, float InY ) | 构造函数重载 |
| Vector2(const Vector2& v) | 拷贝构造 |
| float& operator [] | 按下标取值 |
| float operator [] (int i) const | 按下标取值,不允许修改 |
| void operator() (float nx, float ny) | 重载()直接赋值 |
| bool operator == (const Vector2& w) const | 重载==比较两个向量是否相等,是返回1,否则返回0 |
| Vector2 operator - () const | 重载取负运算符 |
| Vector2& operator *= (float s) | 重载*=运算符,将向量放大或缩小 |
| Vector2 operator + ( const Vector2& V ) const | 重载+运算符,计算两个向量和 |
| Vector2 operator - ( const Vector2& V ) const | 重载-运算符,计算两个向量差 |
| Vector2& operator /= (float s) | 重载/=运算符,计算向量与常数的商 |
| Vector2 operator/( float s ) const | 重载/运算符,计算向量与常数的商 |
| Vector2 operator * ( float Scale, const Vector2& V ) | 重载*运算符,计算向量与常数的积 |
| float Average() const | 计算 x 轴分量与 y 轴分量的平均值 |
| float Dot(const Vector2& v) const | 计算两个向量的点积 |
| float Length() const | 求向量的模 |
| float LengthSquared() const | 计算模的平方,用于进行向量长度比较 |
| bool Normalize() | 判断是否可以将向量单位化如果向量为0则返回0,否则返回1 |
| Vector2 GetNormalized() const | 将向量单位化如果向量不为0,单位化,否则将其置为0向量 |
| Vector2 Perpendicular(void) const | 求原向量的正交向量 |
| bool IsZeroLength(void) const | 判断向量是否为0向量 |
| std::ostream& operator << ( std::ostream& o, const Vector2& v ) | 重载<<运算符,可以将二维向量按照Vector2(x,y)的格式输出 |
| Vector2 GetRotatedInRadian(float radian) | 将向量安装弧度 radian 旋转得到新的向量 |
| Vector2 GetRotatedInDegree(float degree) | 将向量按照角度 degree 旋转得到新的向量 |

## Ⅰ.Ⅲ　向量—Vector3

三维向量是数学库中最重要的部分，它的数据成员由三个浮点数型组成，分布表示3维向量在X、Y、Z轴上的三个分量。表附-2中列出了三维向量类需要支持的操作。

表附-2　Vector3需要实现的功能

| 函数声明 | 函数功能描述 |
| --- | --- |
| Vector3() | 缺省的构造函数 |
| Vector3( float InX , float InY , float InZ ) | 初始化构造函数 |
| float& operator [] (int i) | 按下标取向量的各分量 |
| Vector3 operator * ( float Scale, const Vector3& V ) | 重载运算符 * ,计算常数与向量的乘积 |
| Vector3 operator+ ( const Vector3& V ) const | 重载运算符+ ,计算两个向量的和 |
| float Yaw () const | 得到向量相对于 $x$ 轴的偏离角度 |
| float Pitch () const | 得到向量相对于 $y$ 轴的偏离角度 |
| float Roll () const | 得到向量相对于 $z$ 轴的偏离角度 |
| Vector3 GetNormal() const | 判断向量距离哪个轴最近,若 $x$ 分量最大则离 $x$ 轴最近 ,返回(1,0,0) |
| float ScalarProjectionOntoVector (Vector3& v1) | 得到向量 1 在向量 2 上投影的长度 |
| Vector3 ProjectionOntoVector (const Vector3& v1) const | 得到向量 1 在向量 2 上投影的向量 |
| Vector3 Lerp (const Vector3& v1, float t) const | 线性插值 ,即求在两个向量之间连线上比例为 t 处点所表示的向量值 |
| float RadAngle(const Vector3& v1) const | 计算两个向量之间的偏离的弧度 |
| Vector3 Reflected(const Vector3& n) | 求向量 1 的关于以向量 2 为法向量的平面的反射向量 |
| float CosAngle(const Vector3& v1) const | 计算两个向量直接偏离角度的余弦值 |
| float DistanceToLine (const Vector3& p0, const Vector3& p1) const | 计算一个向量到另外两个向量之间连线的距离 |
| float Average() const | 计算三维向量三个分量的平均值 |
| inline Vector3 HalfWay (const Vector3& dest) const | 两个向量的中点向量 |
| std::ostream& operator << ( std::ostream& o, const Vector3& v ) | 重载运算符<< ,使之安装格式 Vector3 (v $x$, $y$, $z$)输出 |
| static Vector3 MakeDirection(float yaw, float pitch, float roll) | 将一个向量按照参数中的角度旋转 ,并且单位化 |
| Vector3 NormalFromTriangle (const Vector3& v0, const Vector3& v1, const Vector3& v2) | 求 v0 ,v1 ,v2 三点所组成的平面的法向量 |
| static Vector3 Cross (const Vector3& v1, const Vector3& v2) | 将两个向量叉乘 ,并且单位化 |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| static Vector3 CrossNoNorm(const Vector3& v1, const Vector3& v2) | 两个向量的叉积 |
| void Zero() | 将向量置为 0 |
| float DotSelf() const | 计算自身的点积,即求模的平方 |
| Vector3 operator-( const Vector3& V ) const | 重载运算符 -,计算两个向量的差 |
| Vector3 operator*( float Scale ) const | 重载运算符 *,计算向量与一个常数的积 |
| Vector3 operator/( float Scale ) const | 重载运算符/,求向量与常数的商 |
| Vector3 operator / ( const Vector3& rhs) const | 重载运算符/实现两个向量的除法,对应分量相除 |
| Vector3 operator * ( const Vector3& rhs) const | 重载运算符 *,计算两个向量的乘积,每个分量对应相乘 |
| bool operator==( const Vector3& V ) const | 重载运算符==,判断两个向量是否相同,每个分量都相同才认为两者相等 |
| bool operator !=( const Vector3& V ) const | 重载运算符 !=,判断两个向量是否不相同,任一个分量不相同均认为两者不相等 |
| Vector3 operator-() const | 重载运算符 -,取向量的负向量,即各分量取负 |
| void operator+=( const Vector3& V ) | 重载运算符+=,计算当前向量与向量 V 的和,对应分量做和。 |
| void operator=( const Vector2& V ) | 重载 运算符=,将 V 的各分量赋值给当前向量各分量 |
| void operator -=( const Vector3& V ) | 重载运算符 -=,计算当前向量与向量 V 的差,对应分量做差。 |
| void operator *=( float Scale ) | 重载运算符 *=,计算当前向量与常数 scale 的乘积,每个分量与之做积 |
| void operator/=( float V ) | 重载运算符/=,计算当前向量与常数 V 的商,每个分量分别做商 |
| float Length() const | 求向量的模 |
| float LengthSquared() const | 计算模的平方,用于进行向量长度比较 |
| void Set(float x, float y, float z) | 将向量各分量重新赋值 |
| bool Cmp(const Vector3& v) | 比较两个向量是否相等,利用重载的==实现 |
| void Add(const Vector3& v) | 当前向量与向量 V 做加法 |
| void Sub(const Vector3& v) | 当前向量与向量 V 做减法 |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| float DotProduction(const Vector3& v) const | 求当前向量与向量V的点积 |
| Vector3 CrossProduction(const Vector3& v) | 求当前向量与向量V的叉积 |

最后需要补充的是向量点乘和叉乘一定不能搞混。向量的点乘是两个向量对应分量乘积的和，结果是一个数值，代表的物理意义是力在一个方向上的做功；而叉乘的结果是一个和原来两个向量垂直的向量，是一个向量。两个函数的实现如下：

```
Vector3DotProduct(constVector3&v1, constVector3&v2){     //点乘
    return v1.x*v2.x+v1.y*v2.y+v1.z*v2.z;
}
Vector3 CrossProduct(constVector3&v1, constVector3&v2){//叉乘
    returnVector3((v.y*v2.z)-(v.z*v2.y),
                  (v.z*v2.x)-(v.x*v2.z),
                  (v.x*v2.y)-(v.y*v2.x));
}
```

## Ⅰ.Ⅳ　向量－Vector4

四维向量的作用是处理物体的平移、选择、缩放。我们选择从Vector3继承来实现Vector4，另外增加一个第四维数据成员：float Vector4::w。所有实现的操作如表附-3所示，实现代码可以参考随书光盘中的代码。

表附-3　Vector4需要实现的功能

| 函数声明 | 函数功能描述 |
| --- | --- |
| Vector4() | 缺省的构造函数,初始化 |
| Vector4(const float v[4]) | 重载构造函数,用一个大小为 4 的数组初始化 |
| Vector4(float x, float y, float z, float w) | 重载构造函数,用四个浮点数初始化 |
| Vector4 Lerp(const Vector4& v1, float t) | 线性插值,即求在两个向量之间连线上比例为 t 处点所表示的向量值 |
| Vector4(const Vector4& w) | 拷贝重载构造函数,用已有对象初始化向量 |
| Vector4 operator * (float s) const | 重载运算符 * ,计算四维向量与常量的乘积 |
| Vector4& operator *= (float s) | 重载运算符 *= ,计算四维向量与常量的乘积并且赋值给原向量 |
| Vector4 operator * (const Vector4& a) const | 重载 * 计算两个四维向量的乘积,对应分量做乘积 |
| Vector4& operator *= (const Vector4& a) | 重载 *= 计算两个四维向量的乘积,对应分量做乘积并且赋给当前向量 |
| Vector4 operator + (const Vector4& a) const | 重载+计算两个四维向量的和,对应分量做加法 |
| Vector4& operator += (const Vector4& a) | 重载+=计算两个四维向量的和,对应分量做加法并且结果赋给当前向量 |
| Vector4 operator − (const Vector4& a) const | 重载一计算两个四维向量的差 |
| Vector4& operator −= (const Vector4& a) | 重载一=计算两个四维向量的差,对应分量做减法并且结果赋给当前向量 |
| Vector4 operator / (float s) const | 重载/计算四维向量与一个常数 s 的商 |
| Vector4& operator /= (float s) | 重载/=计算四维向量与一个常数 s 的商,并将将结果赋给当前向量 |
| float Length() const | 计算四维向量的模 |
| float Average() const | 计算四维向量四个分量的平均值 |
| Vector4 Normalized() const | 将一个四维向量单位化 |
| Vector4 operator * (const Matrix4& m) | 计算四维向量与四维矩阵的乘积,具体做法是将向量看成 1×4 的矩阵,然后乘以 m 矩阵,按照矩阵乘法法则结果仍为 1×4 的矩阵,即新的四维向量 |

## Ⅰ.Ⅴ 矩阵

游戏中矩阵是用来做坐标变换的，也可以用来表示三维坐标系中的方向。在引擎中应该提供3 × 3矩阵和4 × 4矩阵。所有需要实现的操作如表附-4和表附-5所示，实现代码可以参考随书光盘中的代码。

表附-4 Matrix3需要实现的功能

| 函数声明 | 函数功能描述 |
|---|---|
| inline Matrix3 () | 构造函数,初始化 3 * 3 矩阵为一个单位矩阵 |
| inline explicit Matrix3 (const float arr[3][3]) | 构造函数,用一个 3_3 数组初始化 3 * 3 矩阵 |
| inline Matrix3 (const Matrix3& rkMatrix) | 构造函数,用参数 3 * 3 矩阵初始化矩阵 |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| Matrix3 (float fEntry00, float fEntry01, float fEntry02, float fEntry10, float fEntry11, float fEntry12, float fEntry20, float fEntry21, float fEntry22) | 构造函数,用 9 个 float 参数初始化 3 * 3 矩阵 |
| inline float * operator[] (int iRow) const | []操作符重载,得到参数指定行的首地址 |
| Vector3 GetColumn (int iCol) const | 得到 Matrix 对应列 |
| void SetColumn(int iCol, const Vector3& vec) | 对 Matrix 相应列进行设置 |
| void FromAxes(const Vector3& xAxis, const Vector3& yAxis, const Vector3& zAxis) | 根据三个坐标轴上的投影得到一个矩阵 |
| inline Matrix3& operator= (const Matrix3& rkMatrix) | 把已知 3 * 3 矩阵的值赋给 3 * 3 矩阵对象 |
| bool operator== (const Matrix3& rkMatrix) const | 判断两个 3 * 3 矩阵是否相等 |
| Matrix3 operator+ (const Matrix3& rkMatrix) const | 两个 3 * 3 矩阵相加 |
| Matrix3 operator- (const Matrix3& rkMatrix) const | 两个 3 * 3 矩阵相减 |
| Matrix3 operator * (const Matrix3& rkMatrix) const | 两个 3 * 3 矩阵相乘 |
| Matrix3 operator- () const | 3 * 3 矩阵乘以 -1 |
| Vector3 operator * (const Vector3& rkVector) const | 3 * 3 矩阵和列向量相乘 |
| friend Vector3 operator * (const Vector3& rkVector,const Matrix3& rkMatrix) | 行向量与 3 * 3 矩阵相乘 |
| Matrix3 operator * (float fScalar) const | 3 * 3 矩阵与标量相乘 |
| Matrix3 Transpose () const | 对 3 * 3 矩阵求转置矩阵 |
| bool Inverse (Matrix3& rkInverse, float fTolerance = 1e-06) const | 对 3 * 3 矩阵求逆矩阵,如果逆矩阵存在,把结果赋给第一个参数,返回 bool 1。如果逆矩阵不存在,则返回 bool 0,并把与原矩阵相乘为 O 的矩阵赋给第一个参数 |
| Matrix3 Inverse (float fTolerance = 1e-06) const | 对 3 * 3 矩阵求逆矩阵,如果逆矩阵存在,则返回逆矩阵;如果逆矩阵不存在,则返回零矩阵与原矩阵相乘为 O 的矩阵 |
| float Determinant () const | 求 3 * 3 矩阵的行列式值 |
| void Orthonormalize () | 对 3 * 3 矩阵正交化 |

续表

| 函数声明 | 函数功能描述 |
|---|---|
| void QDUDecomposition (Matrix3& rkQ, Vector3& rkD, Vector3& rkU) const | QDU 分解;把 3 * 3 矩阵分解成一个正交矩阵、一个对角矩阵、一个上三角矩阵(01 02 12 位置)结果分别传给三个参数(由于后两个矩阵只有三个数字,故用 vector3 来表示) |
| float SpectralNorm () const | 求矩阵的范数 |
| void FromAxisRadian (const Vector3& rkAxis, const float fRadians) | 一个 3 维参数向量做旋转轴,float 参数作为旋转角度,旋转后到一个 3 * 3 矩阵的值赋给调用对象矩阵(与调用矩阵原来的值无关) |
| bool ToEulerAnglesXYZ (float& rfYAngle, float& rfPAngle, float& rfRAngle) const | 计算在欧拉变换中分别绕 XYZ 多少角度(弧度)能得到调用对象矩阵,把角度返回给三个参数。如果有唯一解函数返回值为真,如果解不唯一则返回值为假。 |
| bool ToEulerAnglesXZY (float& rfYAngle, float& rfPAngle, float& rfRAngle) const; | 计算在欧拉变换中分别绕 XZY 多少角度(弧度)能得到调用对象矩阵,把角度返回给三个参数。如果有唯一解函数返回值为真,如果解不唯一则返回值为假。 |
| bool ToEulerAnglesYXZ (float& rfYAngle, float& rfPAngle, float& rfRAngle) const; | 计算在欧拉变换中分别绕 YXZ 多少角度(弧度)能得到调用对象矩阵,把角度返回给三个参数。如果有唯一解函数返回值为真,如果解不唯一则返回值为假。 |
| bool ToEulerAnglesYZX (float& rfYAngle, float& rfPAngle, float& rfRAngle) const; | 计算在欧拉变换中分别绕 YZX 多少角度(弧度)能得到调用对象矩阵,把角度返回给三个参数。如果有唯一解函数返回值为真,如果解不唯一则返回值为假。 |
| bool ToEulerAnglesZXY (float& rfYAngle, float& rfPAngle, float& rfRAngle) const; | 计算在欧拉变换中分别绕 ZXY 多少角度(弧度)能得到调用对象矩阵,把角度返回给三个参数。如果有唯一解函数返回值为真,如果解不唯一则返回值为假。 |
| bool ToEulerAnglesZYX (float& rfYAngle, float& rfPAngle, float& rfRAngle) const; | 计算在欧拉变换中分别绕 ZYX 多少角度(弧度)能得到调用对象矩阵,把角度返回给三个参数。如果有唯一解函数返回值为真,如果解不唯一则返回值为假。 |
| void FromEulerAnglesXYZ (const float fYAngle, const float fPAngle, const float fRAngle); | 在欧拉变换坐标系中,分别绕 XYZ 如参数值的角度(弧度),把得到的矩阵赋给调用对象矩阵 |

续表

| 函数声明 | 函数功能描述 |
|---|---|
| void FromEulerAnglesXZY (const float fYAngle, const float fPAngle, const float fRAngle); | 在欧拉变换坐标系中,分别绕 XZY 如参数值的角度(弧度),把得到的矩阵赋给调用对象矩阵 |
| void FromEulerAnglesYXZ (const float fYAngle, const float fPAngle, const float fRAngle); | 在欧拉变换坐标系中,分别绕 YXZ 如参数值的角度(弧度),把得到的矩阵赋给调用对象矩阵 |
| void FromEulerAnglesYZX (const float fYAngle, const float fPAngle, const float fRAngle); | 在欧拉变换坐标系中,分别绕 YZX 如参数值的角度(弧度),把得到的矩阵赋给调用对象矩阵 |
| void FromEulerAnglesZXY (const float fYAngle, const float fPAngle, const float fRAngle); | 在欧拉变换坐标系中,分别绕 ZXY 如参数值的角度(弧度),把得到的矩阵赋给调用对象矩阵 |
| void FromEulerAnglesZYX (const float fYAngle, const float fPAngle, const float fRAngle); | 在欧拉变换坐标系中,分别绕 ZYX 如参数值的角度(弧度),把得到的矩阵赋给调用对象矩阵 |
| void EigenSolveSymmetric (float afEigenvalue[3],Vector3 akEigenvector[3]) const; | 计算 3 * 3 矩阵的特征值与特征向量分别赋给第一个和第二个参数 |
| static void TensorProduct (const Vector3& rkU, const Vector3& rkV ,Matrix3& rkProduct); | 前两个向量参数的张量积(3 * 3 矩阵)赋给第三个参数 |
| static Matrix3 LookTowards(const Vector3& dir); | 根据参数向量得到一个指向 dir 方向同时 up 方向时 y 轴的 3 * 3 矩阵 |
| static Matrix3 LookToUp (const Vector3& up); | 根据参数向量得到一个指向 up 方向同时右边是 x 轴的 3 * 3 矩阵 |
| static Matrix3 LookTowardsUp (const Vector3& to ,const Vector3& up); | 根据参数向量 to ,up,得到一个对着 to 方向同时向上的方向是 up 的 3 * 3 矩阵 |

表附-5　Matrix4需要实现的功能

| 函数声明 | 函数功能描述 |
| --- | --- |
| inline Matrix4() | 初始化矩阵为 4 * 4 单位矩阵 |
| inline Matrix4 (float m00, float m01, float m02, float m03, float m10, float m11, float m12, float m13, float m20, float m21, float m22, float m23, float m30, float m31, float m32, float m33 ) | 用 16 个变量初始化矩阵 |
| inline Matrix4(const Matrix3& m3x3) | 通过 3 * 3 矩阵初始化一个 4 * 4 矩阵 |
| inline Matrix4(const Quaternion& rot) | 通过一个四元数初始化一个 4 * 4 矩阵 |
| inline float * operator [] ( int iRow ) | []重载,得到 4 * 4 矩阵的一行的首地址,适用于变量 |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| inline const float * const operator [ ] ( int iRow ) const | 同上,适用于常量 |
| Matrix4 operator * ( const Matrix4 &m2 ) | *操作符重载,两个 4*4 矩阵相乘 |
| inline void operator *= ( const Matrix4 &m2 ) | *=操作符重载,两个 4*4 矩阵相乘,把结果赋给调用对象 |
| inline Matrix4 operator + ( const Matrix4 &m2 ) const | +操作符重载,两个 4*4 矩阵相加 |
| inline Matrix4 operator - ( const Matrix4 &m2 ) const | -操作符重载,两个 4*4 矩阵相减(调用对象减去参数) |
| inline bool operator == ( const Matrix4& m2 ) const | ==操作符重载,判断两个 4*4 矩阵是否相等 |
| inline bool operator != ( const Matrix4& m2 ) const | !=操作符重载,判断两个 4*4 矩阵是否不相等 |
| inline void operator = ( const Matrix3& mat3 ) | =操作符重载,把参数 3*3 矩阵赋值给 4*4 矩阵,结果为 4*4 单位矩阵的前三行三列加上参数 3*3 矩阵 |
| static Matrix4 LookTowards(const Vector3& dir) | 根据参数向量得到一个指向 dir 方向同时 up 方向时 y 轴的 4*4 矩阵 |
| static Matrix4 LookToUp (const Vector3& up) | 根据参数向量得到一个指向 up 方向同时右边是 x 轴的 4*4 矩阵 |
| static Matrix4 LookTowardsUp (const Vector3& to ,const Vector3& up) | 根据参数向量 to ,up ,得到一个对着 to 方向同时向上的方向是 up 的 4*4 矩阵 |
| Vector3 GetRight() const | 得到 4*4 矩阵右方向的标准基 |
| Vector3 GetUp() const | 得到 4*4 矩阵上方向的标准基 |
| Vector3 GetDir() const | 得到 4*4 矩阵前方向的标准基 |
| Vector3 GetRightUnnormalized() const | 得到 4*4 矩阵右方向的非标准基 |
| Vector3 GetUpUnnormalized() const | 得到 4*4 矩阵上方向的非标准基 |
| Vector3 GetDirUnnormalized() const | 得到 4*4 矩阵前方向的非标准基 |
| Vector3 GetColumn (int iCol) const | 得到对应列,返回一个 vector3 |
| void SetColumn(int iCol, const Vector3& vec) | 对对应列进行设置 |
| const Matrix4* RotationX(const float radian) | 绕 X 轴自身旋转 弧度 radian,返回旋转后 4*4 矩阵的指针 |
| const Matrix4 * RotationY (const float radian) | 绕 Y 轴自身旋转 弧度 radian,返回旋转后 4*4 矩阵的指针 |
| const Matrix4 * RotationZ (const float radian); | 绕 Z 轴自身旋转 弧度 radian,返回旋转后 4*4 矩阵的指针 |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| const Matrix4 * RotationXYZ(const float radianX, const float radianY, const float radianZ); | 分别绕 XYZ 三轴自身旋转 参数弧度后，返回旋转后 4 * 4 矩阵的指针 |
| static Matrix4 GetRotaionMatrixX(const float radian); | 得到这样一个 4 * 4 矩阵:它乘以对象矩阵后,让对象矩阵得到绕 X 轴旋转 radian 弧度的效果 |
| static Matrix4 GetRotaionMatrixY(const float radian); | 得到这样一个 4 * 4 矩阵:它乘以对象矩阵后,让对象矩阵得到绕 Y 轴旋转 radian 弧度的效果 |
| static Matrix4 GetRotaionMatrixZ(const float radian); | 得到这样一个 4 * 4 矩阵:它乘以对象矩阵后,让对象矩阵得到绕 Z 轴旋转 radian 弧度的效果 |
| static Matrix4 GetRotaionMatrixXYZ(const float radianX, const float radianY, const float radianZ); | 得到这样一个 4 * 4 矩阵:它乘以对象矩阵后,让对象矩阵得到分别绕 XYZ 三轴旋转参数弧度后的效果 |
| inline Matrix4 Transpose(void) const | 得到 4 * 4 矩阵的转置矩阵 |
| inline Matrix4 SetTrans( const Vector3& v ) | 用一个 3 维向量来设定 4 * 4 矩阵的平移转换的值(即设定[0][3]、[1][3]、[2][3]位置的值) |
| inline Matrix4 SetOffset( const Vector3& v ) | 用一个 3 维向量来设定 4 * 4 矩阵的偏移量的值(即设定[3][0]、[3][1]、[3][2]位置的值) |
| inline Vector3 GetTrans() const | 获得 4 * 4 矩阵的平移转换的值(即[0][3]、[1][3]、[2][3]位置的值) |
| inline Vector3 GetOffset() const | 获得 4 * 4 矩阵的偏移量的值(即设定[3][0]、[3][1]、[3][2]位置的值) |
| inline void MakeTrans( const Vector3& v ) | 用一个 3 维向量把调用对象矩阵(无论矩阵原来是怎么样的)变成一个平移的转换矩阵(如预期结果) |
| inline void MakeOffset( const Vector3& v ) | 用一个 3 维向量把调用对象矩阵(无论矩阵原来是怎么样的)变成一个偏移的转换矩阵(如预期结果) |
| inline void MakeTrans( float tx, float ty, float tz ) | 用 3 个 float 变量把调用对象矩阵(无论矩阵原来是怎么样的)变成一个平移的转换矩阵(如预期结果) |
| inline void MakeOffset( float tx, float ty, float tz ) | 用 3 个 float 变量把调用对象矩阵(无论矩阵原来是怎么样的)变成一个偏移的转换矩阵(如预期结果) |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| inline static Matrix4 GetTrans ( const Vector3& v ) | 用一个 3 维向量把调用对象矩阵（无论矩阵原来是怎么样的）变成一个平移的转换矩阵，并把这个矩阵作为函数返回值（如预期结果） |
| inline static Matrix4 GetOffset ( const Vector3& v ) | 用一个 3 维向量把调用对象矩阵（无论矩阵原来是怎么样的）变成一个偏移的转换矩阵，并把这个矩阵作为函数返回值（如预期结果） |
| inline static Matrix4 GetTrans ( float t_x, float t_y, float t_z ) | 用 3 个 float 变量把调用对象矩阵（无论矩阵原来是怎么样的）变成一个平移的转换矩阵，并把这个矩阵作为函数返回值（如预期结果） |
| inline static Matrix4 GetOffset ( float t_x, float t_y, float t_z ) | 用 3 个 float 变量把调用对象矩阵（无论矩阵原来是怎么样的）变成一个偏移的转换矩阵，并把这个矩阵作为函数返回值（如预期结果） |
| inline const float * Get4_4Array() const | 把 4 * 4 矩阵的首地址赋给一个 4_4 数组；通过数组可以方便的输出矩阵内部的值 |
| inline void SetMatrix(const float * p4_4Array) | 用一个 4_4 常量数组的值赋给 4 * 4 矩阵 |
| inline void Identify() | 把 4 * 4 矩阵初始化或设为 4 * 4 单位矩阵 |
| inline void Zero() | 把 4 * 4 矩阵初始化或设为 4 * 4 零矩阵 |
| inline void SetScale( const Vector3& v ) | 用参数 3 维向量设置对象 4 * 4 矩阵的 scale 值（[0][0]，[1][1]，[2][2]位置的值） |
| inline static Matrix4 GetScale ( const Vector3& v ) | 用参数 3 维向量设置对象 4 * 4 矩阵（可以为任何值的矩阵）的 scale 值（[0][0]，[1][1]，[2][2]位置的值），并把其他除[3][3]外的值设为 0，[3][3]设为 1，并把结果矩阵作为函数的返回值 |
| inline static Matrix4 GetScale( float s_x, float s_y, float s_z ) | 用 3 个 float 值设置对象 4 * 4 矩阵（可以为任何值的矩阵）的 scale 值（[0][0]，[1][1]，[2][2]位置的值），并把其他除[3][3]外的值设为 0，[3][3]设为 1，并把结果矩阵作为函数的返回值 |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| inline void Extract3x3Matrix ( Matrix3& m3x3 ) const | 把对象 4 * 4 矩阵的前 3 行前 3 列交叉部分赋给参数 3 * 3 矩阵 |
| inline Quaternion ExtractQuaternion() const | 通过一个 4 * 4 矩阵得到一个四元数 |
| static Matrix4 * MatrixLookAtLH( Matrix4 &pOut,const Vector3 &pEye,const Vector3 &pAt,const Vector3 &pUp ) | 通过三个参数 3 维向量构造一个在左手系坐标系里的 4 * 4 矩阵,通过参数 1 输出 |
| static Matrix4 * MatrixLookAtRH( Matrix4 &pOut,const Vector3 &pEye,const Vector3 &pAt,const Vector3 &pUp ) | 通过三个参数 3 维向量构造一个在右手系坐标系里的 4 * 4 矩阵,通过参数 1 输出 |
| inline Matrix4 operator * (float scalar) | * 操作符重载,4 * 4 矩阵与参数标量相乘,返回相乘后的结果矩阵 |
| inline friend std::ostream& operator <<( std::ostream& o , const Matrix4& m ) | <<操作符重载,按行输出 4 * 4 矩阵 |
| Matrix4 Adjoint() const | 求 4 * 4 矩阵的伴随矩阵,并作为函数返回值返回 |
| float Determinant() const | 对 4 * 4 矩阵求其行列式值 |
| Matrix4 Inverse() const | 对 4 * 4 矩阵求逆矩阵,返回值为逆矩阵 |
| inline friend Vector4 operator * (const Vector4& v , const Matrix4& mat) | * 操作符重载,4 维向量与 4 * 4 矩阵相乘返回一个 4 维向量 |

# Ⅰ.Ⅵ　四元数

四元数在实时绘制中非常重要，它可以快速地实现绕任意轴旋转的问题，同时，它也能避免欧拉旋转的万向锁问题。引擎中实现的四元数函数如表附-6所示。

表附-6　四元数类

| 函数声明 | 函数功能描述 |
| --- | --- |
| Quaternion (float fW ,float fX , float fY , float fZ) | 构造函数,为四元数赋值 |
| Quaternion (const Quaternion& rkQ) | 拷贝构造函数 |
| Quaternion nlerp(float fT , const Quaternion& rkP, const Quaternion& rkQ , bool shortestPath) | 线性插值后单位化,计算快但不够精确 |
| void FromRotationMatrix (const Matrix3& kRot) | 从旋转矩阵中提炼四元数 |
| void ToRotationMatrix (Matrix3& kRot) const | 由四元数提炼旋转矩阵 |
| void FromAngleRadianAxis (const float rfAngle, const Vector3& rkAxis) | 绕任意轴旋转一角度,返回旋转四元数 |

续表

| 函数声明 | 函数功能描述 |
| --- | --- |
| void ToAngleRadianAxis (float& rfAngle, Vector3& rkAxis) const | 由四元素返回绕的轴及角度 |
| Vector3 xAxis() const | X 轴分量 |
| Vector3 yAxis() const | Y 轴分量 |
| Vector3 zAxis() const | Z 轴分量 |
| void FromAxes (const Vector3& xaxis, const Vector3& yaxis, const Vector3& zaxis) | 对 Quaternion 中的 W X Y Z 进行赋值 |
| void ToAxes (Vector3& xaxis, Vector3& yaxis, Vector3& zaxis) const | 四元数的 X、Y、Z 轴的分量 |
| Quaternion operator+ (const Quaternion& rkQ) const | 操作符重载,四元数相加 |
| Quaternion operator− (const Quaternion& rkQ) const | 操作符重载,四元数相减 |
| Quaternion operator * (const Quaternion& rkQ) const | 操作符重载,四元数相乘 |
| Quaternion operator * (float fScalar) const | 操作符重载,四元数相乘 |
| Quaternion operator− () const | 操作符重载,四元数取负 |
| float Norm () const | 长度的平方 |
| Quaternion Inverse () const | 四元数的逆变换 |
| Quaternion UnitInverse () const | 单位逆变换 |
| Quaternion Exp () const | 只适用于单位四元数 |
| Quaternion Log () const | 只适用于单位四元数 |
| Vector3 operator * (const Vector3& v) const | 用四元数旋转三维向量 |
| bool Equals (const Quaternion& rhs, const float tolerance) const | 判断两个四元数是否相等,容忍误差为 tolerance |
| Quaternion Slerp (float fT, const Quaternion& rkP, const Quaternion& rkQ, bool shortestPath) | 球面线性插值 |
| Quaternion SlerpExtraSpins (float fT, const Quaternion& rkP, const Quaternion& rkQ, int iExtraSpins) | 球形线性插值 |
| Quaternion Squad (float fT, const Quaternion& rkP, const Quaternion& rkA, const Quaternion& rkB, const Quaternion& rkQ, bool shortestPath) | 球面二次插值 |
| float Normalise(void) | 单位化,长度平方 |
| float GetRoll(void) const | 四元数的 Z 轴旋转分量 |
| float GetPitch(void) const | 四元数的 X 轴旋转分量 |
| float GetYaw(void) const | 四元数的 Y 轴旋转分量 |

## Ⅰ.Ⅶ　射线

射线的数据表示就是一个原点和一个方向：Vector3 m_Origin，Vector3 m_Direction。函数成员如附表-7所示。

附表-7　射线类的功能

| 函数声明 | 函数功能描述 |
| --- | --- |
| Ray() | 缺省的构造函数初始化 |
| Ray(const Vector3& origin, const Vector3& direction) | 用两个向量初始化光线的起点和方向 |
| void SetOrigin(const Vector3& origin) | 改变光线的起点，将其设置为参数中的origin的向量 |
| const Vector3& GetOrigin(void) const | 得到光线的起点向量 |
| void SetDirection(const Vector3& dir) | 改变光线的方向，将其设置为新的方向 |
| const Vector3& GetDirection(void) const | 得到光线的方向向量 |
| Vector3 GetPoint(float t) const | 得到当前光线上距离起点比例为t的向量点 |
| Vector3 operator*(float t) const | 重载运算符*，得到方向相同，长度为原光线t的新光线的终点向量，与GetPoint相同功能 |
| std::pair<bool, float> Intersects(const Plane& p) const | 判断光线与给定平面p是否相交，如果相交则返回一个bool值true和起点到交点的距离 |
| std::pair<bool, float> Intersects(const Sphere& s) const | 判断光线与给定球面p是否相交，如果相交则返回一个bool值true和起点到交点的距离 |

## Ⅰ.Ⅷ　平面

3D空间中的平面可以用如下方程表示：

Ax+By+Cz+D=0

其中：Vector3（A，B，C）是平面的法向，D是原点到平面的距离。所以平面的数据成员就是法向量Vector3 m_vNormal和原点到平面的距离float m_fD。这里需要注意的是点到平面的距离是可以为负的，当该点在平面的背向，即与平面法向量方向相反的方向时，该点到平面的距离为负。

平面类Plane函数成员如附表-8所示：

附表-8　平面类的函数成员

| 函数声明 | 函数功能描述 |
|---|---|
| plane() | 缺省的构造函数初始化 |
| void Init (const Plane& rhs) | 用平面 rhs 初始化当前平面 |
| void Init (const Vector3& rkNormal, float fConstant) | 用一个向量 rkNormal 作为平面的法向量,fConstant 为原点到平面的距离初始化平面 |
| void Init (const Vector3& rkNormal, const Vector3& rkPoint) | 用一个向量 rkNormal 作为平面的法向量,rkPoint 为另外一点初始化平面 |
| void Init (const Vector3& rkPoint0, const Vector3& rkPoint1, const Vector3& rkPoint2) | 利用平面上的三点初始化平面,用其中两个向量确定平面法向量,另外一点和法向量的点积为原点到平面的距离 |
| Side GetSide (const Vector3& rkPoint) const | 判定 rkPoint 在平面的那一侧,对于平面的侧定义了正,负,无,分别表示与法向量同向,反向,在平面上 |
| float GetDistance (const Vector3& rkPoint) const | 点 rkPoint 到平面的距离 |
| bool operator==(const Plane& rhs) const | 重载运算符==,判断两个平面是否相等 |

# Ⅰ.Ⅸ 球

球的数据表示就是圆心和半径：Vector3 m_Center， float m_Radius。功能函数如附表-9所示。

附表-9 球类的函数成员

| 函数声明 | 函数功能描述 |
| --- | --- |
| Sphere() | 构造函数，初始化圆心和半径 |
| Sphere(const Vector3& center, float radius) | 重载构造函数，用制定向量点为圆心，指定常数为半径 |
| float GetRadius(void) const | 得到球的半径 |
| void SetRadius(float radius) | 改变球的半径 |
| const Vector3& GetCenter(void) const | 得到球的圆心坐标 |
| void SetCenter(const Vector3& center) | 改变球的圆心坐标 |
| bool Intersects(const Sphere& s) const | 判断当前球是否与传入的球有相交，利用球心距离和两球半径和相比 |
| bool IfInSphere(const Sphere& s) const | 判断该球是否在传入的球的内部，利用球心距离和半径差相比 |
| bool IfOutOfSphere(const Sphere& s) const | 判断该球是否在传入的球的外部，利用球心距离和半径差相比 |
| bool Intersects(const Plane& plane) const | 判断该球是否与传入的平面相交，用球心到平面的距离和球的半径相比较 |
| bool Intersects(const Vector3& v) const | 判断点向量是否在该球的内部，利用点和球心的距离与半径比较 |
| bool operator == (Sphere s) const | 判断两个球是否相同，半径和圆心均相同 |

## Ⅰ.Ⅹ　平截锥体

平截锥体是一个六面体，在游戏中最大的用处是描述照相机的视域。平截锥体的最大特征是上下两面都是四边形且平行。

我们在实现平截锥体类的时候可以用六个平面组成：Plane m_Plane[6]，平面的顺序依次为：左，右，上，下，近，远，同时使用一个枚举来定义不同的面：

```
enum FRUSTUM_PLANE{
    PLANE_LEFT,
    PLANE_RIGHT,
    PLANE_TOP,
    PLANE_BOTTOM,
    PLANE_FRONT,
    PLANE_BACK
};
```

平截锥体的功能需求和应用场景相对简单，所以没有必要提供太多的功能。在引擎中，我们只提供了取得相对平面和相交测试的功能。

## Ⅰ.Ⅺ　相交测试

游戏中的相交测试一定要注重效率，同时也要考虑精确度。

我们举一个很简单的例子，即二维长方形的相交测试，传入参数是两个长方形的左上角坐标和长、宽，传出bool型表示是否相交。

```
boolRectIntersects(int rectAX, int rectAY, int rectAW idth, int rectAHeight, int rectBX, int rectBY, int rectBW idth, int rectBHeight)
{
    //排除不碰撞的情况，就是碰撞了
       if((rectAX-rectBX>=rectBW  idth)|| (rectBX-rectAX>=rectAW-idth)
         || (rectAY-rectBY>=rectBHeight)|| (rectBY-rectAY>=rectA-Height))
      return false;
    else
```

```
        return true;
}
```

程序写的相对短小简单，但同时也实现功能。

引擎中，实现了许多图形的相交测试，如附表-10所示。

附表-10　相交测试的函数

| 函数声明 | 函数功能描述 |
| --- | --- |
| PointInTri2D ( float px, float pz, float ax, float az, float bx, float bz, float cx, float cz ) | 判断点是否在一个 2D 三角形的内部 |
| std::pair<bool, float> Intersects (const Ray& ray, const Plane& plane) | 判断光线与给定平面 p 是否相交@ returns [pair<bool, float>]：第一个返回值表示是否相交，第二个返回值是起点到交点的距离 |
| std::pair<bool, float> Intersects (const Ray& ray, const std::vector<Plane>& planes, bool normalIsOutside) | 射线和一个体的相交检测@ returns [pair<bool, float>]：第一个返回值表示是否相交，第二个返回值是起点到交点的距离 |
| bool Intersects (const Ray& ray, const Sphere& sphere, bool discardInside) | 判断射线与给定球面 p 是否相交@ returns [pair<bool, float>]：第一个返回值表示是否相交，第二个返回值是起点到交点的距离 |
| bool Intersects(const Sphere& sphere, const Plane& plane) | 球和平面的相交检测 |
| bool Intersects(const Sphere& sphere, const Frustum& frustum) | 球和六面体的相交检测 |

# Ⅱ. 程序附录

| 文件目录 | 补充说明 |
| --- | --- |

GE Engine

- Engine Develop 引擎开发文件夹
  - Framework 引擎的主体
  - GEMath 数学库
  - GEInput 输入库
  - GUIEditor GUI编辑器
  - EffectEditor 特效编辑器
  - AnimationViewer 动画浏览器
- Net 网络
  - IOCP 完成端口模型
- AI 人工智能
  - 路径规划与移动技术
- 有限状态机
- 脚本技术
  - Lua脚本
  - 简单脚本
- 群聚技术
- 遗传算法
- 神经网络

模型及动画导出插件

| 项目 | 说明 |
| --- | --- |
| OGRE模型及动画导出器 | 1.2版，配合3Ds Max 5.0 |
| 模型导出脚本（.GEM .TXT） | 执行脚本 |
| 关键帧动画导出脚本（.GEA .TXT） | 执行脚本 |
| 关键帧动画文件辅助生成工具 | 帮助生成动画执行序列 |

Release

GUI Editor.exe

Effect Editor.exe

Animation Viewer.exe

| 项目 | 说明 |
| --- | --- |
| 小Demo | 每个文件夹中有若干Demo |

Demo1

Demo2

Demo3

Demo4

Demo DirectInput

部分Demo代码

Docs

| | |
|---|---|
| GE Engine Framework.chm | 引擎主体文档 |
| GE Engine Math.chm | 数学库文档 |
| GE Engine Input.chm | 输入库文档 |

# 参考文献


1. 游戏引擎演化史。http://www.gameres.com, 2005

2. 闫辉。网络游戏发展迅速，能否成为软件开发新天堂。程序员，2004

3. 3D Engine的设计架构。http://www.gameres.com

4. David H Eberly. 3D Game Engine Design: A Practical Approach to Real-Time Comput-er Graphics. Morgan Kaufmann Publishers Inc., 2000

5. David H Eberly. 3D Game Engine Architecture. Morgan Kaufmann Publishers Inc., 2004

6. Mage小组。OGRE使用指南。http://www.gameres.com

7. Tomes Akenine-Moller, Eric Haines. 普建涛译。实时计算机图形学。北京大学出版社，2004

8. Mark DeLoura. Game Programming Gems 1. Charles River Media, Inc

9. Upstill. The RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics. Addison-Wesley, 1990

10. Wolfgang F. Engel. Direct3D游戏编程入门教程。人民邮电出版社

11. Mark DeLoura. Game Programming Gems 2. Charles River Media, Inc

12. Mark DeLoura. Game Programming Gems 3. Charles River Media, Inc

13. Mark DeLoura. Game Programming Gems 4. Charles River Media, Inc

14. Mark DeLoura. Game Programming Gems 5. Charles River Media, Inc

15. Jim Adams. ADVANCED ANIMATION WIT H DIRECTX. Course Technology PTR

16. John van der Burg. Building an Advanced Particle System. Game Developer, 2000

17. Jeff Lander. The Ocean Spray in Your Face. Game Developer, 1998

18. Geczy, George. 2D Programming in a 3D World: Developing a 2D Game Engine Using DirectX 8 Direct3D. Gamasutra, 2001

19. McReynolds et al. Advanced Graphics Programming Techniques Using OpenGL course notes. SIGGRAPH 99

20. Forsyth, Tom. Impostors: Adding Clutter. Game Programming Gems 2.Charles Riv-er Media, 2001

21. Mason McCuskey. Developing a GUI using C++ and DirectX. http://www.gamed-ev.net

22. Erich Gamma et al. Design Patterns. Addison Wesley

23. [ 美] Andre LaMothe. Windows游戏编程大师技巧。北京：中国电力出版社，2004

24. Jason Clark. 掌握DirectX和DirectInput力反馈游戏杆。www.gameres.com

25. Anthony Jones, Jim Ohlund. Windows网络编程。北京：清华大学出版社，2002

26. W. Richard Stevens. TCP/IP详解。北京：机械工业出版社，2005

27. IOCP Thread Pooling in C#. http://www.codeproject.com

28. Shared Source CLI. http://research.microsoft.com/programs/europe/rotor/

29. Dead Reckoning: Latency Hiding for Networked Games. http://www.gamasutra.com

30. Defeating Lag with Cubic Splines. http://www.gamedev.net

31. 普悠玛数位科技。Visual C++游戏设计入门。北京：机械工业出版社，2002

32. M. Buckland. Programming Game AI by Example. Wordware Publishing, Inc., 2004

33. A. Carbone, J. Sheard. Developing a Model of Student Learning in a Studio-Based Teaching Environment, Informing Science&IT Education Conference, Cork, Ireland, 2002

34. P. Lester. A* Pathfinding for Beginners. http://www.policyalmanac.org/games/aStarTutorial.htm, 2005

35. D. Livingston, A. McMonies. Is DarkBasic to be considered harmful. 5th Game-On In-ternational Conference, Reading, UK, 2004

36. C. W. Reynolds. Steering Behaviors For Autonomous Characters. Game Developers Conference, 1999

37. http://www.red3d.com/cwr/boids/

38. Flocks, Herds, Schools. A Distributed Behavioral Model. SIGGRAPH '87

39. Game Programming Gems 1. Charles River Media, Inc

40. Game Programming Gems 2. Charles River Media, Inc

41. Game Programming Gems 3. Charles River Media, Inc

42. Game Programming Gems 4. Charles River Media, Inc

43. Game Programming Gems 5. Charles River Media, Inc

# 附录1

CD链接网址：http://downloads.zjupress.cn/3dgame05835.zip