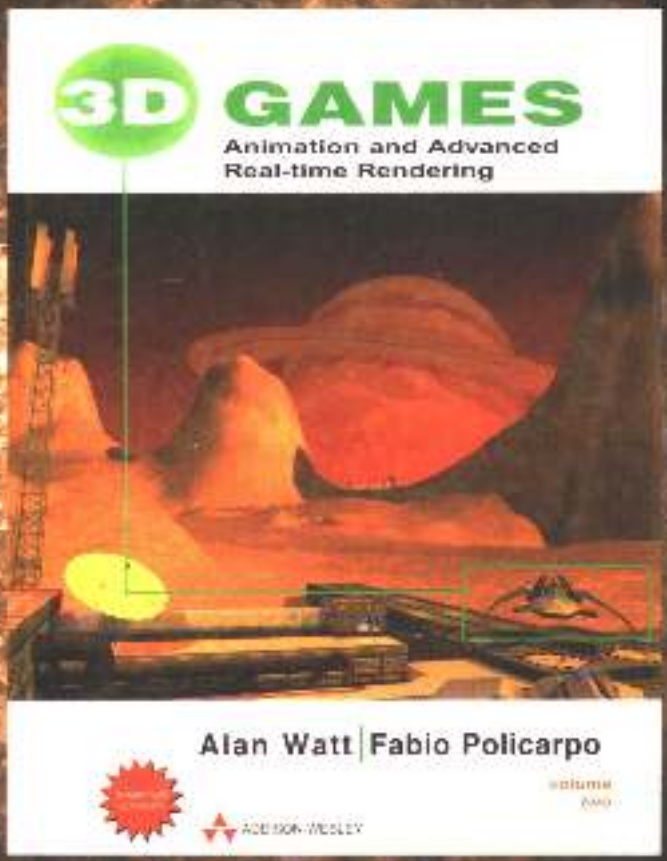
计 算 机 科 学 丛 书

3D游戏

卷2 动画与高级实时渲染技术

（英）Alan Watt Fabio Policarpo 著 沈 明 陈文斌 朱怡波 等译



3D Games

Animation and Advanced Real-time Rendering, Vol 2

机械工业出版社
China Machine Press

本书从实践的角度出发，详细介绍3D游戏开发的高级技术。全书若重讨论三个主题：游戏开发的一般过程（构造过程、实时处理过程和软件设计），实时渲染过程，角色动画。所有主题均围绕一个具体的游戏开发系统Fly3D SDK 2.0（包含在光盘中）加以介绍。

本书旨在为当今的三维游戏引擎技术提供一个综合的解决方案，将游戏理论技术与具体引擎代码分析相结合，使读者尽快地进入开发者角色，了解整个游戏开发过程和客户（游戏设计者）的需求，并初步具备游戏引擎的开发能力。

本书适合作为高等院校相关专业的教学参考书，亦可供3D游戏开发人员参考与学习。

随书光盘包括：

- 完整的Fly3D SDK，包括引擎、前端、插件和实用程序的源代码
- 演示片段
- 引擎指南、参考手册和教程

光盘中包含的软件可在任何Microsoft Windows系统上运行，但需要完全支持OpenGL的三维视频卡。如果要对源代码作改动，则需要Microsoft Visual C++ 6.0。为了顺利创建场景，光盘中还包含3D Studio Max 3.x和4.x插件。另外，在http://www.fly3d.com.br上有Fly3D SDK的文档、更新资料、新的演示、FAQ和讨论区。

作者简介

Alan Watt 英国谢菲尔德大学计算机科学系讲师，是该校计算机图形学研究室主任，曾经编写过多本优秀著作，包括《3D计算机图形学》（即将由机械工业出版社出版）和《The Computer Image》。

Fabio Policarpo 工作在里约热内卢的软件开发者，他是Paralelo计算机公司的创始人，目前正致力于三维动作多玩家游戏的研究。

ISBN 7-111-15776-1

9 787111 157762

HZ BOOKS 华章图书

华章网站 http://www.hzbook.com

网上书店：www.china-pub.com

北京市西城区百万庄南街1号 100037

读者服务热线：(010)68995259，68995264

读者服务信箱：hzedu@hzbook.com

ISBN 7-111-15776-1/TP・4113

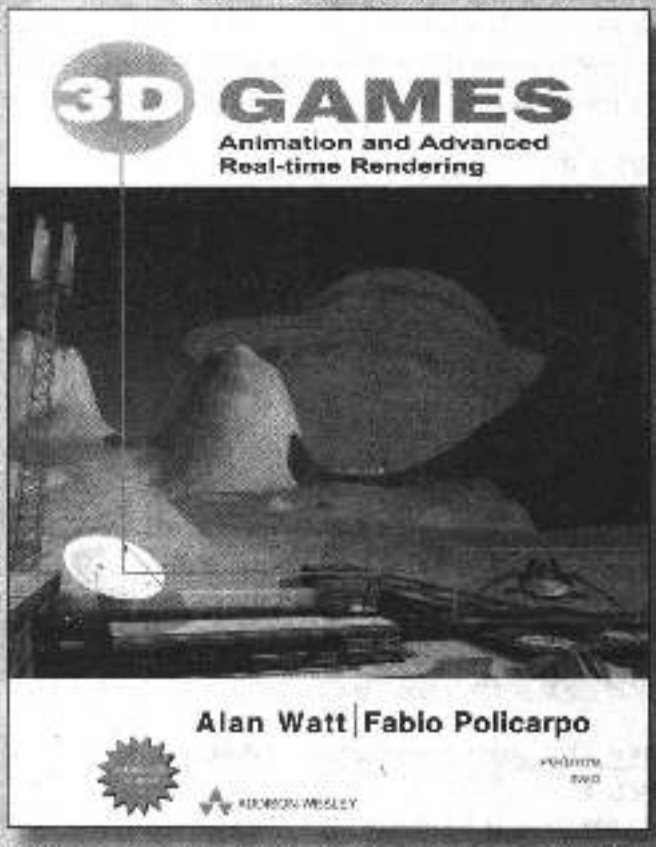
定价：58.00元（附光盘）

计 算 机 科 学 丛 书

3D游戏

卷2 动画与高级实时渲染技术

（英）Alan Watt Fabio Policarpo 著 沈一帆 陈文斌 朱怡波 等译



3D Games

Animation and Advanced Real-time Rendering, Vol 2

机械工业出版社
China Machine Press

本书从实践的角度出发，详细介绍3D游戏开发的高级技术，并具体描述了一个游戏引擎的构建过程。全书着重讨论三个主题：游戏开发的一般过程（构造过程、实时处理过程和软件设计）；实时渲染过程；角色动画。所有主题均围绕一个具体的游戏开发系统Fly3D SDK 2.0（包含在光盘中）加以介绍。

本书旨在为当今的三维游戏引擎技术提供一个综合的解决方案，使读者尽快地进入开发者角色，了解整个游戏的开发过程并初步具备游戏引擎开发能力。

本书适合作为高等院校相关专业的教学参考书，同时可供相关技术人员和游戏开发人员阅读。

Alan Watt，Fabio Policarpo：3D Games：Animation and Advanced Real-time Rendering，Volume two（ISBN：0-201-78706-7）.

Copyright ©2003 by Pearson Education Limited.

This translation of *3D Games：Animation and Advanced Real-time Rendering，Volume two*（ISBN：0-201-78706-7）is published by arrangement with Pearson Education Limited.

本书中文简体字版由英国Pearson Education培生教育出版集团授权出版。

版权所有，侵权必究。

本书法律顾问　北京市展达律师事务所

本书版权登记号：图字：01－2003－3910

图书在版编目（CIP）数据

3D游戏　卷2　动画与高级实时渲染技术/（英）沃特（Watt，A.），（英）波力卡波（Policarpo，F.）著；沈一帆等译．－北京：机械工业出版社，2005．4

（计算机科学丛书）

书名原文：3D Games：Animation and Advanced Real-time Rendering，Volume Two

ISBN 7-111-15776-1

Ⅰ．3…　Ⅱ．①沃…②波…③沈…　Ⅲ．三维－动画－游戏－软件开发　Ⅳ．TP311．5

中国版本图书馆CIP数据核字（2004）第130125号

机械工业出版社（北京市西城区百万庄大街22号　邮政编码　100037）

责任编辑：朱起飞

北京瑞德印刷有限公司印刷·新华书店北京发行所发行

2005年4月第1版第1次印刷

787mm×1092mm　1/16·29（彩插2印张）印张

印数：0 001-4000册

定价：58.00元（附光盘）

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294



图2-1　在Ⅱ游戏应用中使用的引擎



图1-1　整体游戏层次视图



a)



b)



c)

图1-2　a)在CAAD应用中起分割平面作用的结构化元素，b)在CAAD应用中的细节元素，c)整个视角应向应展示的结构化元素和细节元素

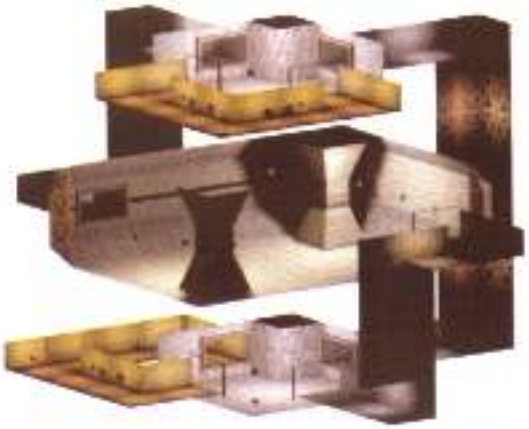
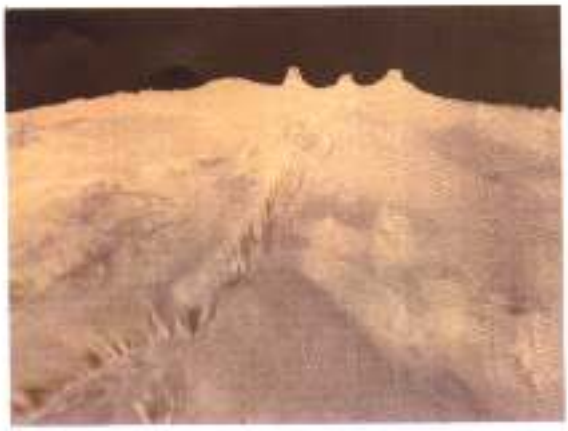


图 1-6 为了产生无迹向节点的一个层次架构



a)

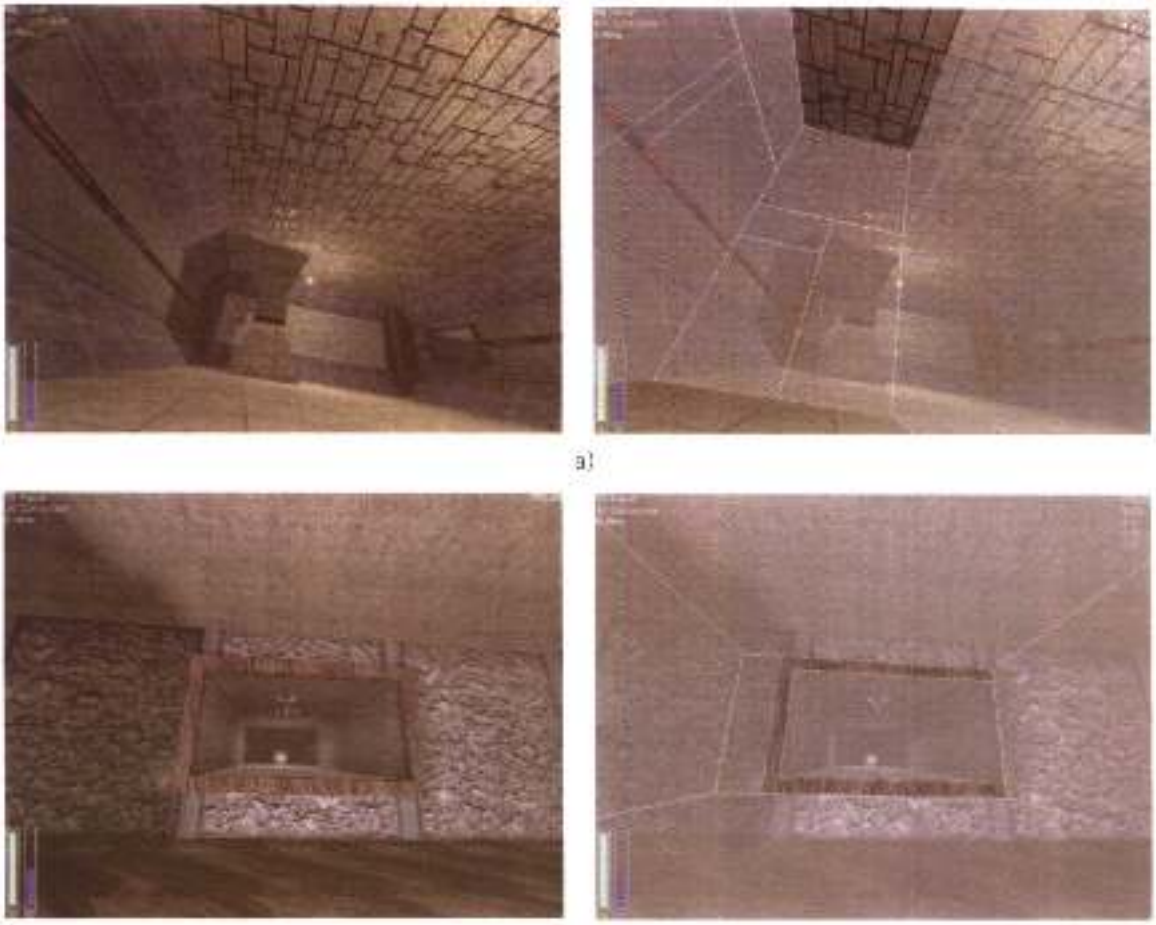


b)



c)

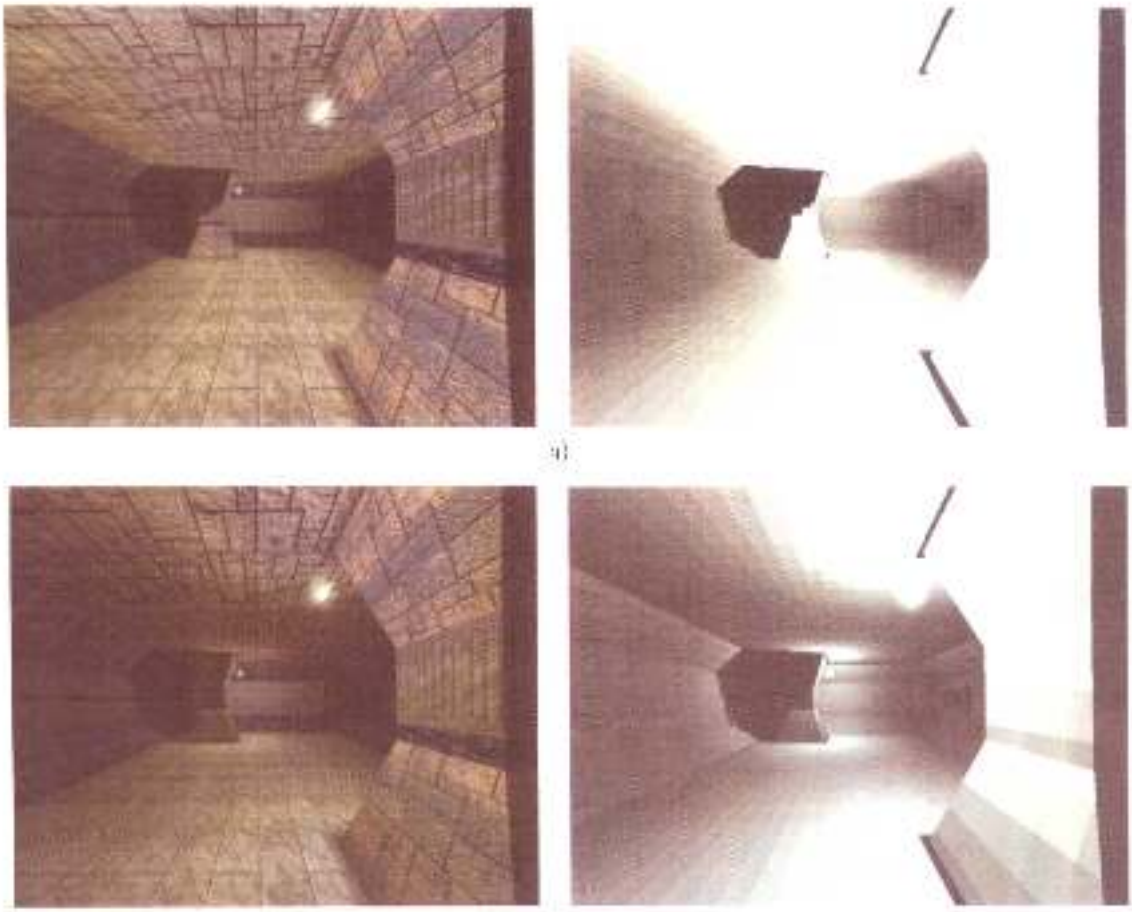
图 1-7 a) 火星表面的复杂地形；b) 展示高级节点次的场景；c) 单一BSP区域的内容



a)

b)

图1-11 a)游戏某一层设计一个空旷场景的伪入口，明确的连接面被显示出来；b)在这个例子中，伪入口平面和真正的入口相一致



a)

b)

图 [illegible] a)照亮光照贴图——平方衰减定律(纹理滤波关闭后使得光照贴图像素可见)
b)照亮光照贴图　平方衰减定律加L,N(纹理滤波关闭后使得光照贴图像素可见)

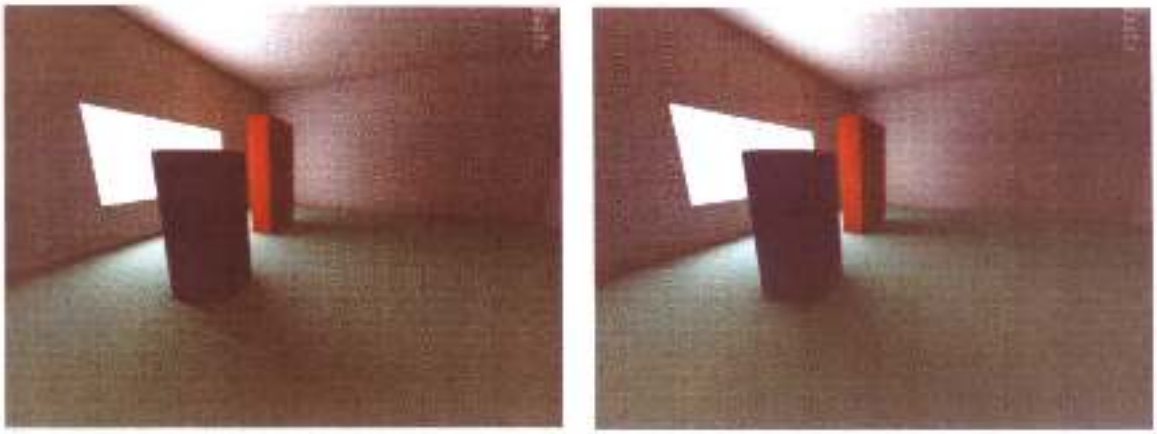


图1-22　用蝎尔漫方法照亮的简单环境

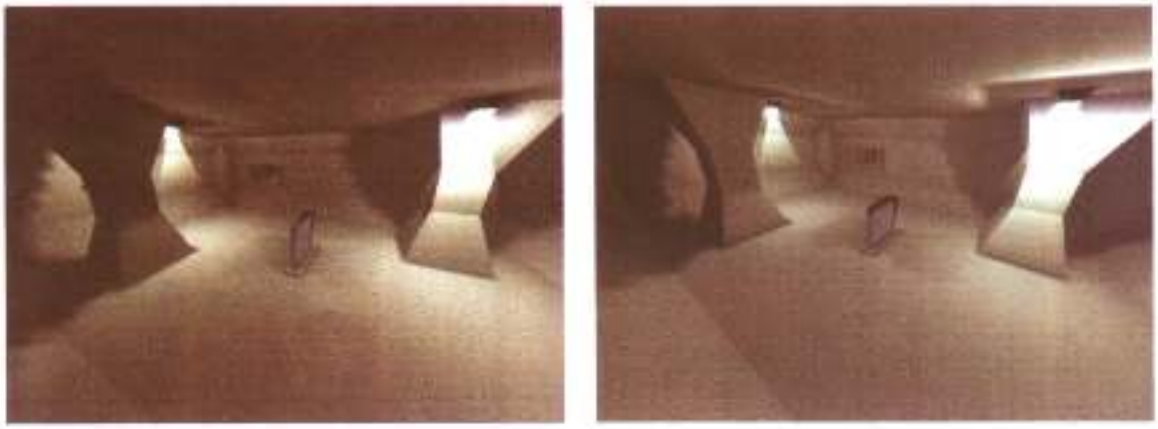


图1-23　使用和不使用辐射度渲染之间的一个比较

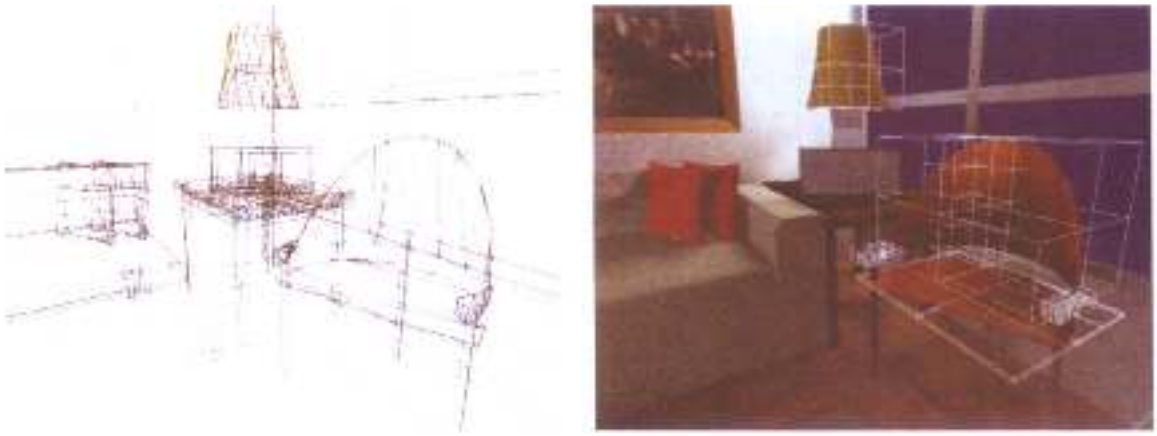


图1-24　一个以贝济埃曲面和三角网格生成八叉树的场景

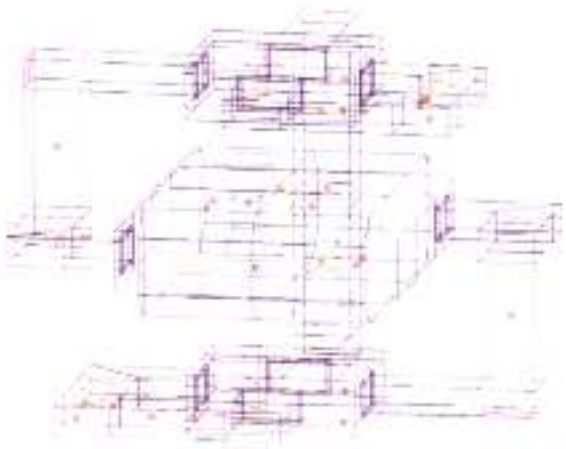


图 A1-3　用不同几何图形显示的线框层次
蓝色：结构化多边形
绿色：细节多边形
红色：游戏实体
紫色：曲面
黄色：光线



a)



b)

图 A1-5　添加光照后的环境



图 A1-6　添加实体



图 A1-11　添加细节物体后的场景

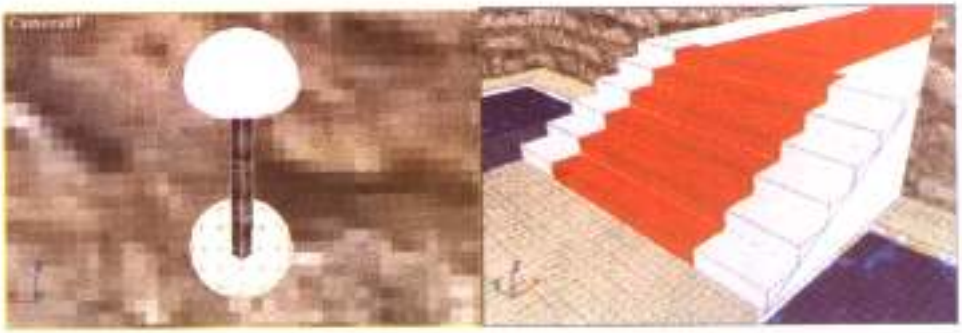


图 A1-12　三角形和多边形创建物体

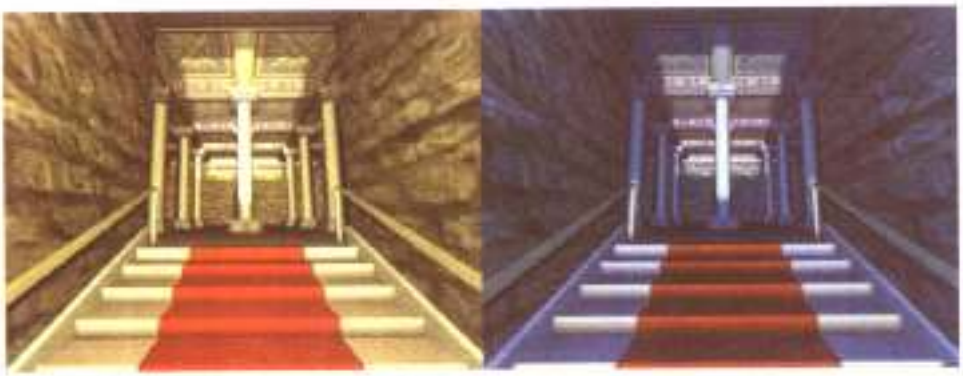
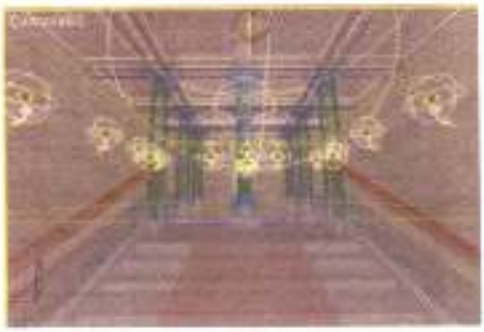


图 A1-13　添加光照



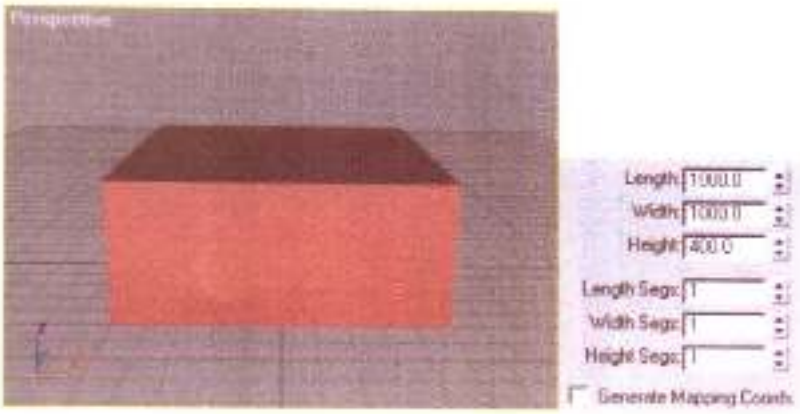


图 A1-14 创建长方体结构化面

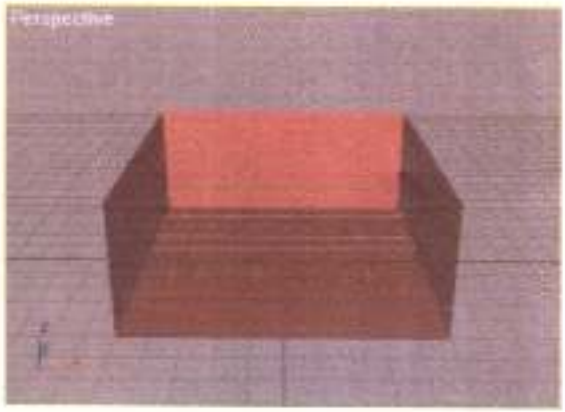


图 A1-15 反转法向前

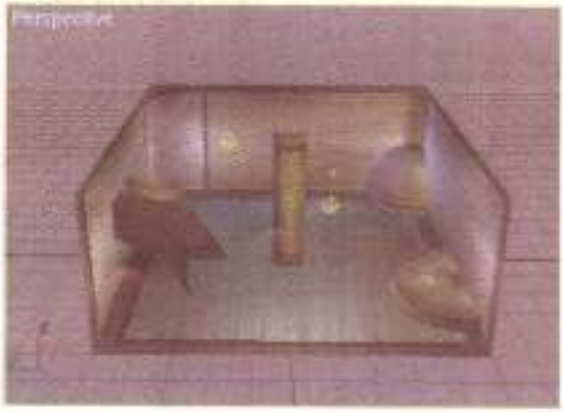


图 A1-16 添加光照

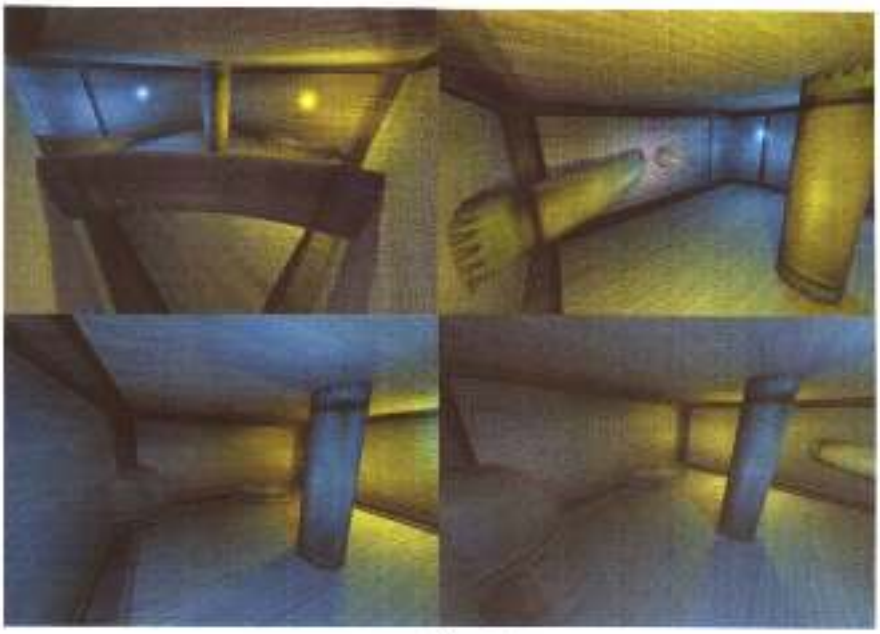
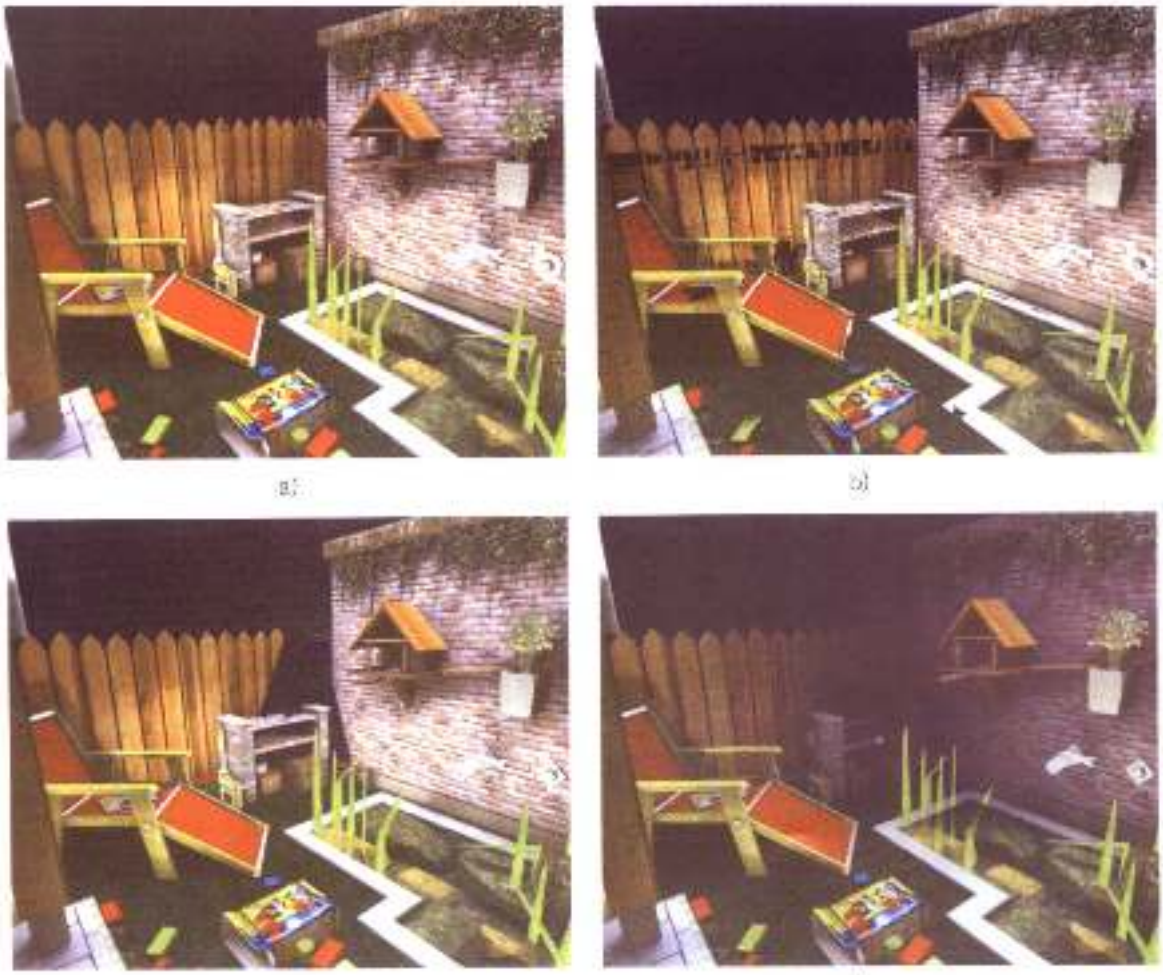


图 A1-25　层次的屏幕截图



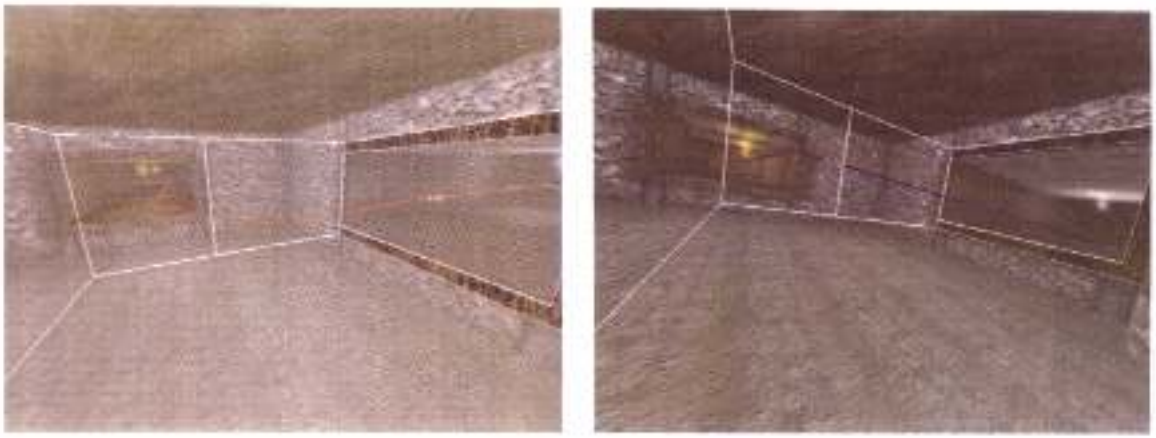
a)　　b)

图 2-2　Padgarden 层和视见约束体。a）用视见约束体裁剪 BSP 叶节点后的场景（共 1453 个面），b）用视见约束体裁剪面后的场景（共 587 个面）



a) b) c) d)

图2-3 a) Padgarden 层次正确的远近裁剪面设置；b) 远裁剪面太远，导致Z缓冲的精度不够；c) 远裁剪面太近，可以明显看出远裁剪面；d) 使用雾化效果降低过近的远裁剪面的影响

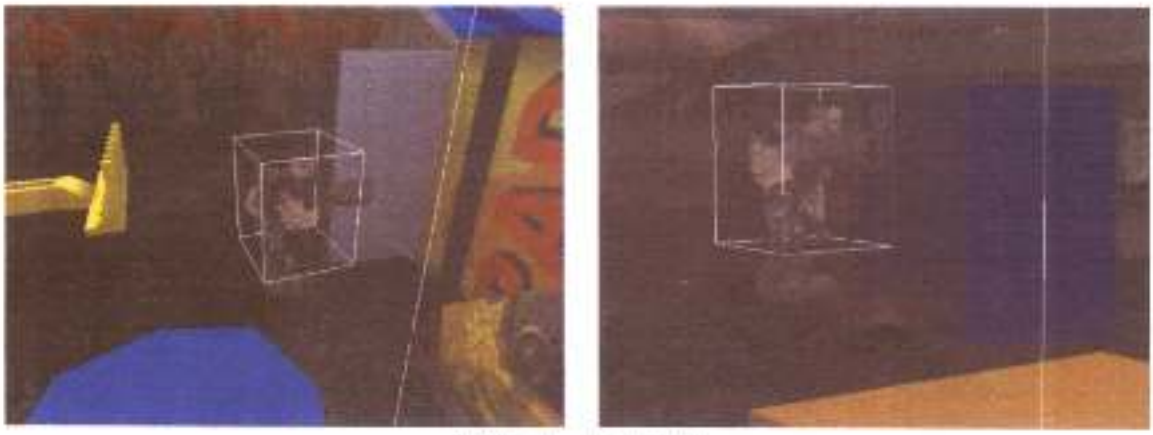


a) b)

图2-18 a) 使用A*算法得到的从凉凸体到日坏凸体路径的一部分；b) 为了使路径更"平滑"，可以去掉路径上的某个顶点，然后检查去掉顶点之后的路径是否会发生碰撞



图A2-1 第三人称照相机模式



图A2-2 边/边碰撞

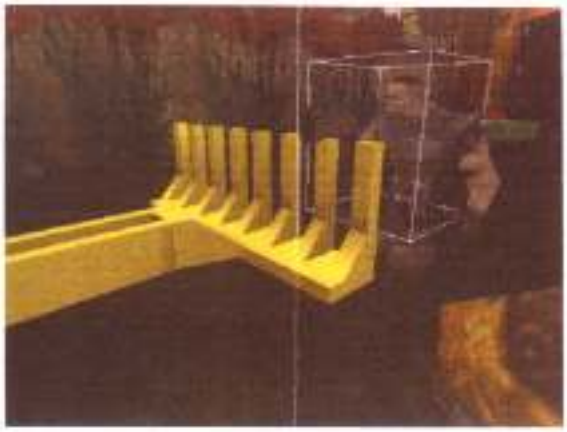


图 A2-3 场景顶点/AABB碰撞



图 A2-4 AABB顶点/场景碰撞

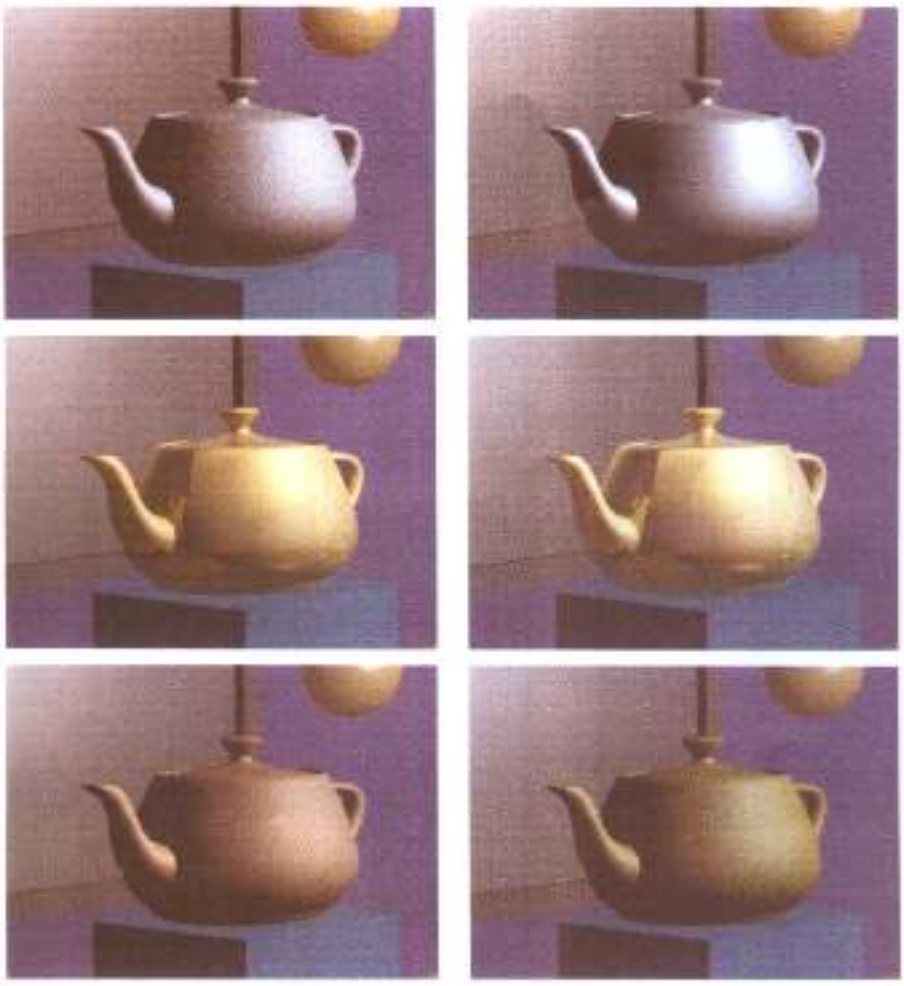


图 4-8 在相同光照条件下渲染的不同材质，在抱光材质的情况中，该模型被用作光线追踪器的一个局部分量

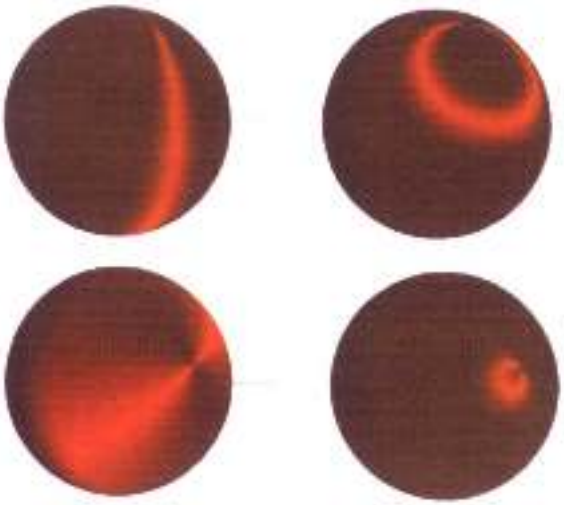


图4-10 球面上的Banks各向异性着色模型，绿线表示光向方向，颗粒从沿经线排列到沿纬线排列水平地变化（观察点保持不变），观察点沿垂直方向变化

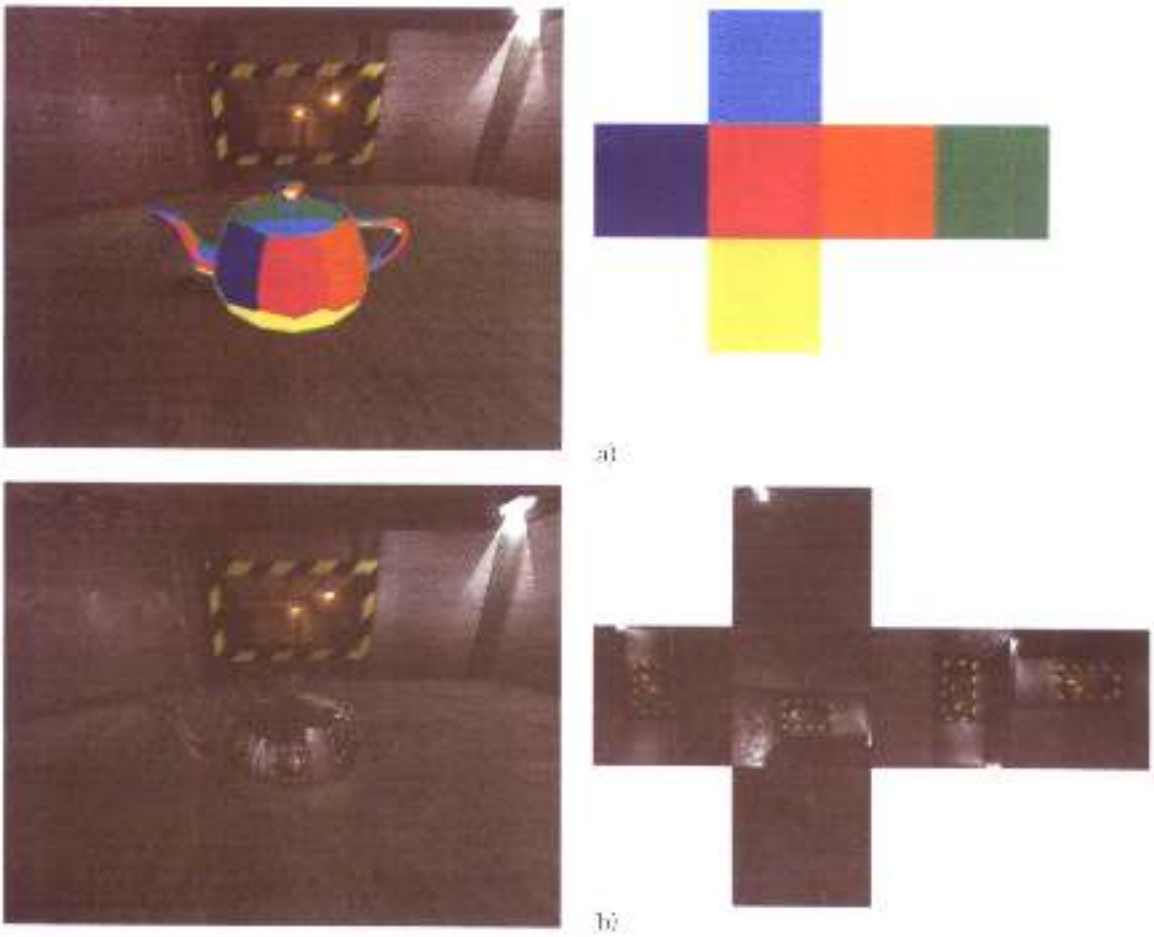


图4-12 a) 显示了一个使用立方体映射的物体，这里的图由6种颜色组成
b) 同一个用立方映射的物体，它现在反映它所在的环境



图4-20　在同样的光照下使用可分离的近似方法渲染的不同材质



图4-23　最简单的曲面着色法——Phong 着色



图4-24　图4-23 的一个变体

第4-23节图4-30使用从斯坦福大学实时可编程着色项目处得到的一个出色系统生成
(www.graphics.stanford.edu)



图4-25 混合有光泽的材质和反射材质



图4-26 应用图4-25中的材质



图4-27 卡通渲染



图4-28 卡通渲染的一个变体

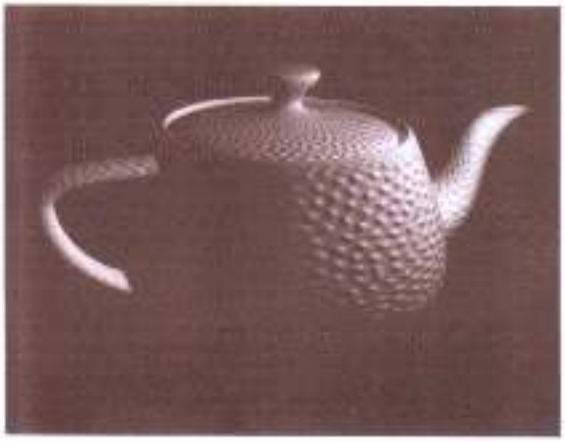


图4-29 凹凸贴图



图4-30 应用图4-29中的凹凸纹理



图5-2 使用铭纹理的玻璃会层桌

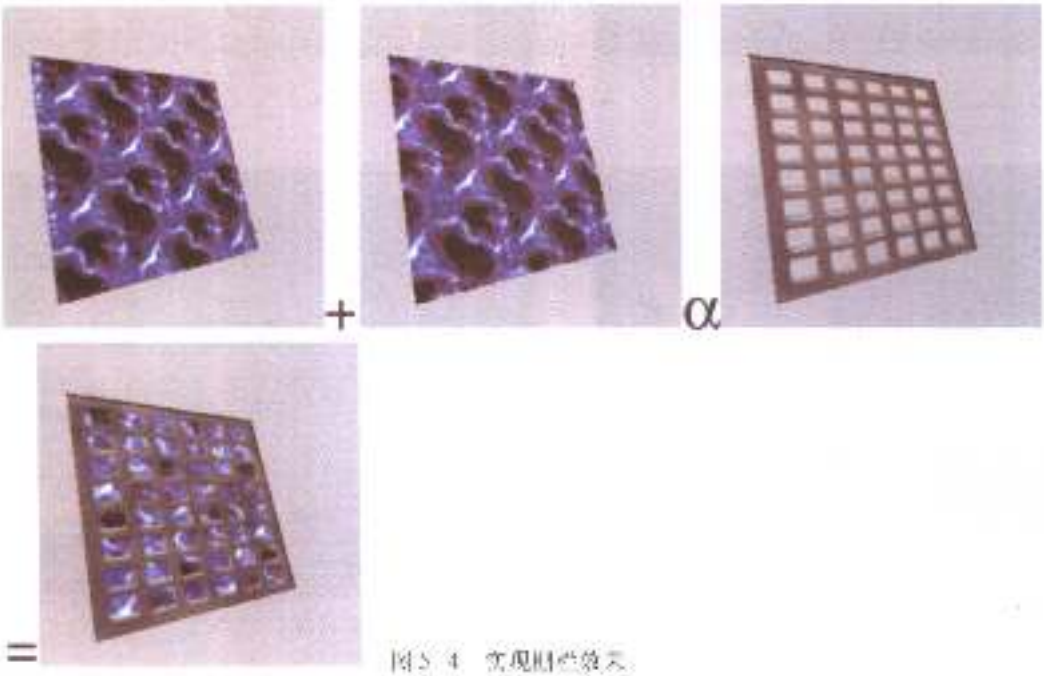


图5-4 实现阻碍效果

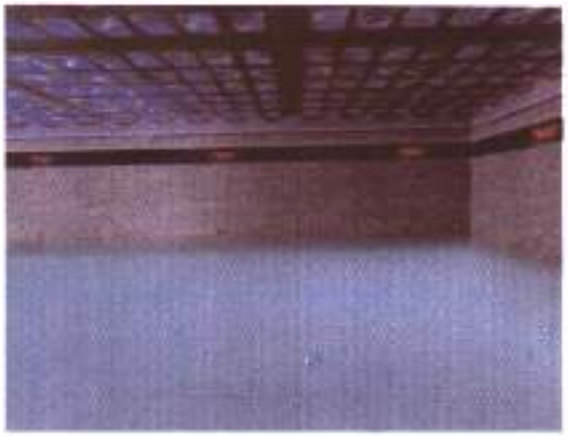


图5 5 加入容积雾后的最终效果





图5 7 栅栏效果的实例



图5 8 随机成白色噪声纹理，其上有一条滚动的红线

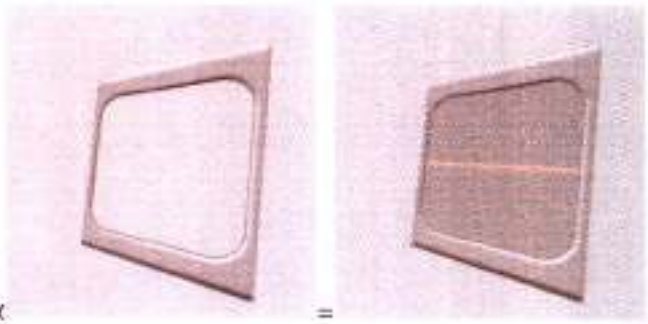


图 5-9 加上框架纹理



图 5-10 加上法线贴图（凹凸混合）

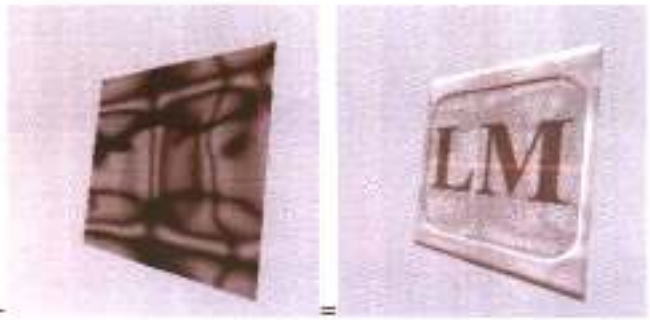


图 5-11 加上镜纹理后的最终效果



图 5-12 把镜纹理映射限制在玻璃屏幕上



图5-13 在一个游戏中的指视器效果

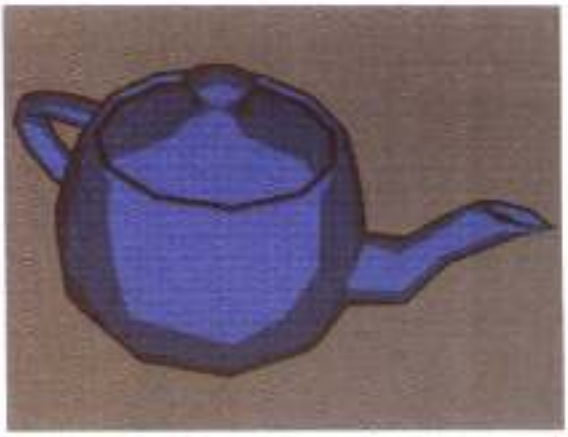


图5-15 三色卡通演染的顶点程序。黑色由影北后的物体所绘出，两种蓝色的明暗代表了粗糙的N.L.用暗



图5-17 水面依据波形顶点程序轻轻起伏的扁水池（从图像中稍小和暗下的水平面可以看出）。静止时图像类似表示，在动画时此效果营造出一个很好的气氛



图5-20 用来演染球体的方形图——颜色代表法向量的方向





图5-21 游戏引擎中的方形图被弯折的光源顶点。第二幅图关闭了alpha测试以显示方形图的背景





图 5-22　两帧模式 1 入口图像的视图。玩家移动时，入口图像保持不变。它是从放置在入口/镜子物体后上方的光源上的观点所生成的。从图像中可看到入口/镜子物体的后部。一般来说，入口将出在另一房间内



图 5-24　入口模式 2（镜子模式）反射出玩家



图 5-25　一个细分后的入口/镜子物体的投影





图5-26　在这幅图中，入口目标可相是放置在入口／镜子物体后上方的光源，相对于入口目标的入口照相机朝向与相对于入口的普通照相机相同。因而当玩家在入口／镜子物体附近移动时，可以看见目标的不同区域。相比之下，模式1的概念总是相同的





图5-27　燃烧尾迹特效



图5-31　所使用的纹理

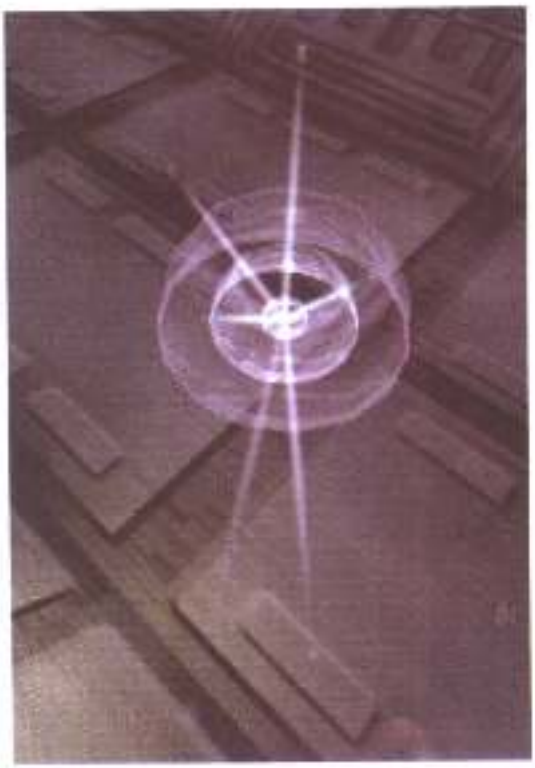


图5-32　最终效果

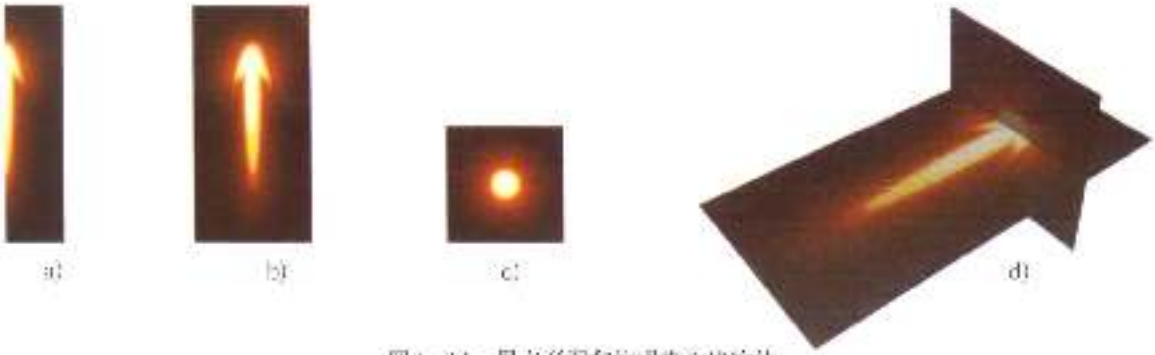


图 3-34 用方形面和纹理来构建物体

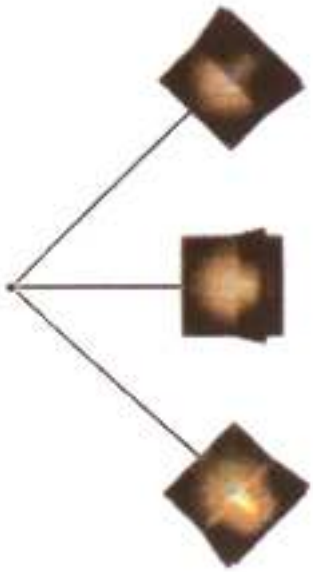


图 3-35 带有可观察者的脉冲云物体





图 3-36 在一个游戏场景中的脉冲云

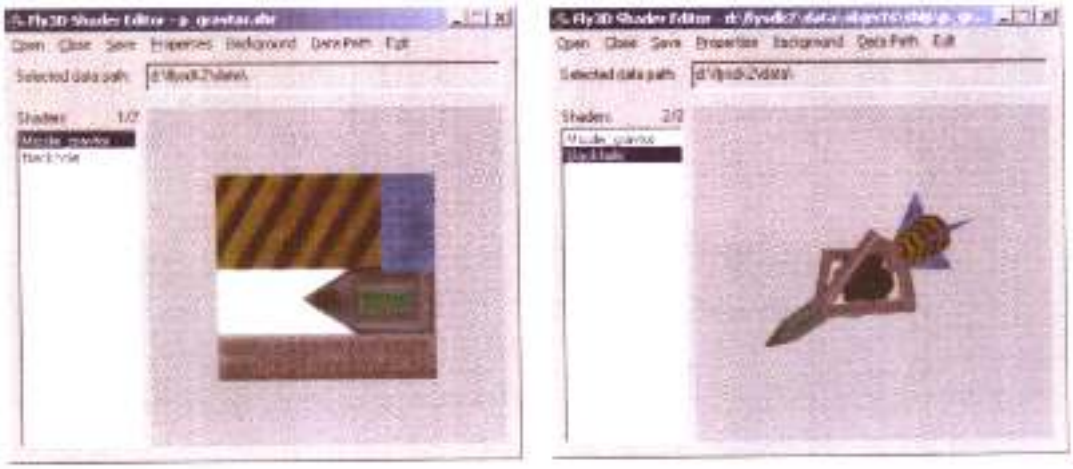


图 A5.1 渲染导弹的两个着色器

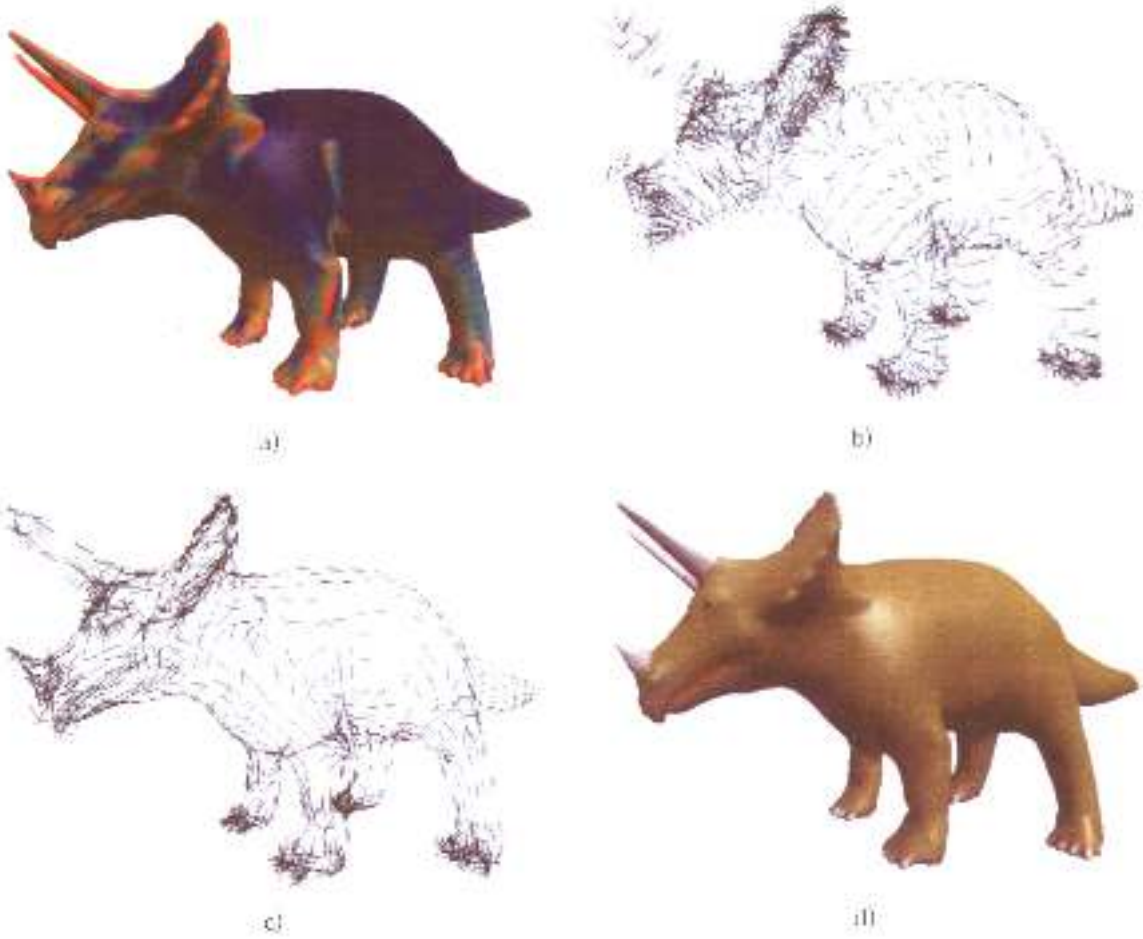


图 6.19 在三角恐龙网格上进行离散微分几何算子的例子。a)平均曲率($1/2(k_1+k_2)$)的渲染；b)标准最大主曲率(k_1)的向量场；c)标准最小主曲率(k_2)的向量场；d)正常渲染的网格



图6-23 a）原始网格（Stanford Bunny的一个低分解率版本）
b）基础网格和顶点的参数化形式

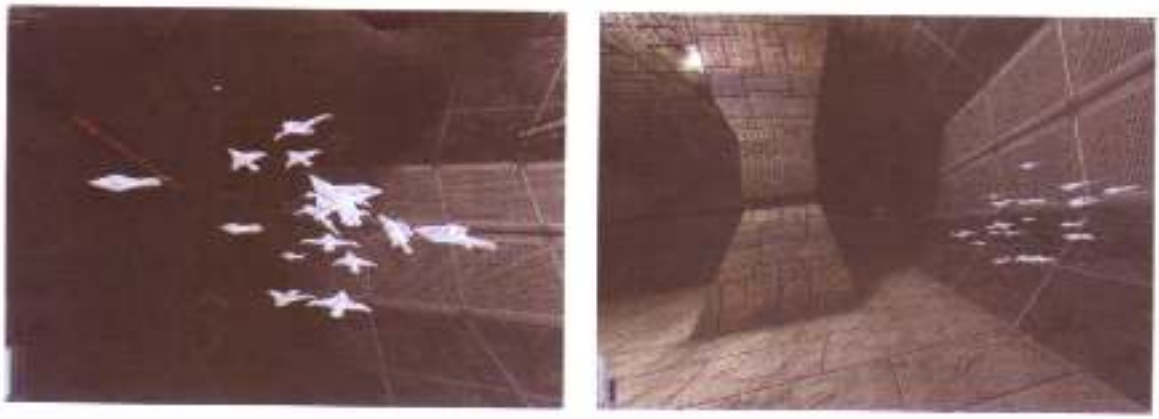
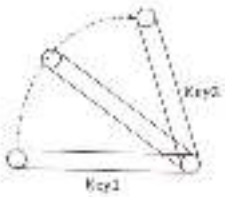


图7-2 在游戏中应用群落模型的两个视图





图7-6 由于顶点插值产生的变形（与图7-7比较）



a)



b)



c)

图 7-7　在消除皮肤层的几何扭曲方面，骨架动画体下顶点动画：a）对旋转量进行插值消除了顶点插值的扭曲；b）整个图像中关键帧的插值（和图 7-4 比较）；c）使用的骨架



图 7-8　骨架间的四元数插值控制根节点生成沿贝济埃曲线运动的位移和方向变化

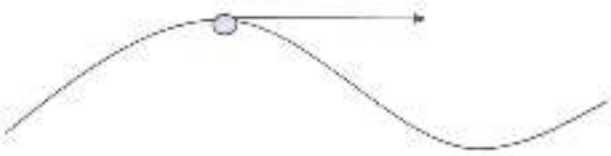




图 7-12　根据贝济埃曲线控制角色颈节点的位置和取向





图7-14　两个人从不同的位置走行到“蹲下”状态的进入状态点。两条路径在同一点结束，所以角色在这一点开始处理时，从而可以将蹲下的动画与步行的动画成功合成



图7-16　经过障碍物，从红色的位置移动到蓝色的位置。在这个例子中，对象都被套进了AABB

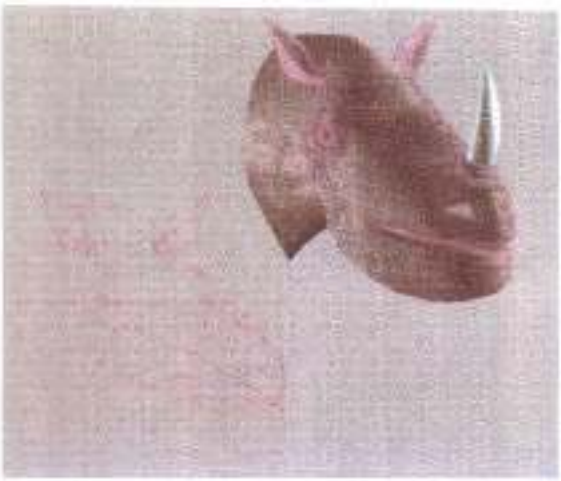
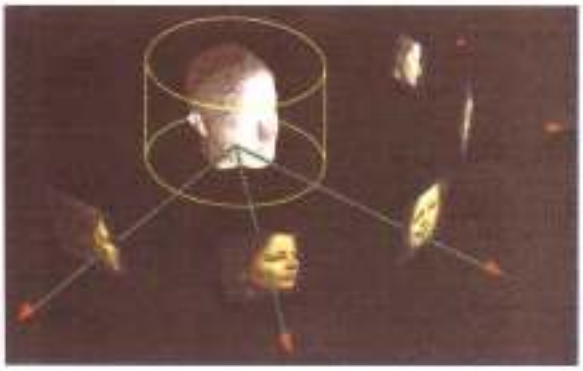


图8-1　在3D Studio MAX中用样条框架建模



点p可能出现在不上一幅图上



合成的纹理贴图 T

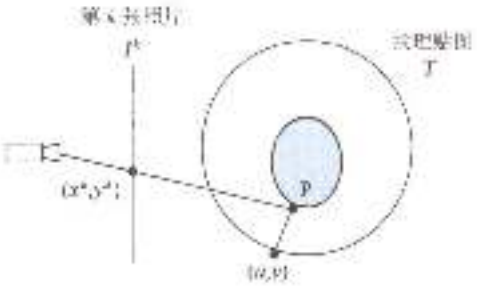
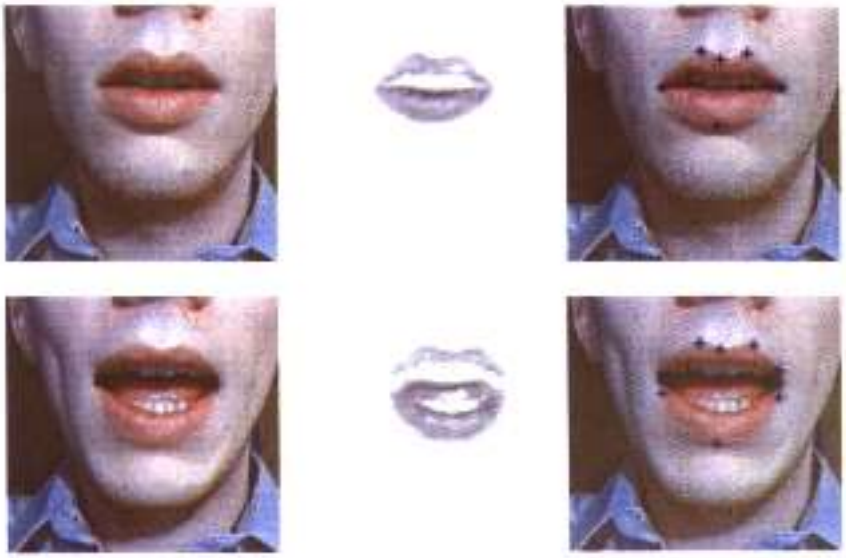


图9-4　均建一个“圆柱形”的纹理贴图



原始图像　　嘴唇分割　　复制

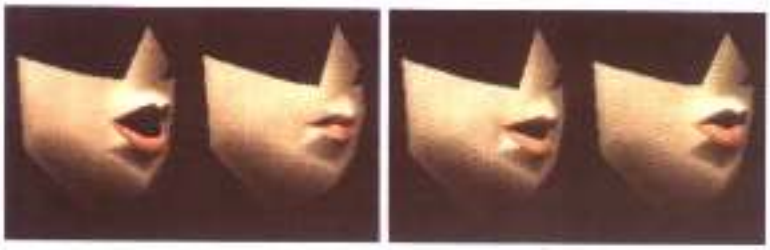
图9-6　嘴唇上的图像处理操作



图9-7　使用snake进行嘴唇跟踪。第一行是嘴唇外部轮廓，第二行是嘴唇内部轮廓



下颚张开　　嘴呼变成圆形



嘴唇闭上　　嘴舌上抬

图9-9　MOTHER中关于表效的极端变化



图9-14　Pixar公司1988年制作的《Tin Toy》中的婴儿





图A9 1 Fly3D执行中的两个骨肌肉模型（Emmanuel Tanguy授权，圣尔德大学）

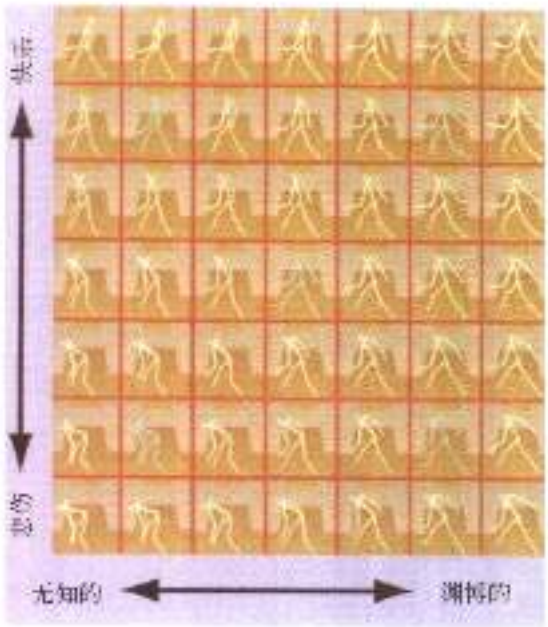


图 10-8 从动词实例产生一个连续的运动空间（绿色－实例）





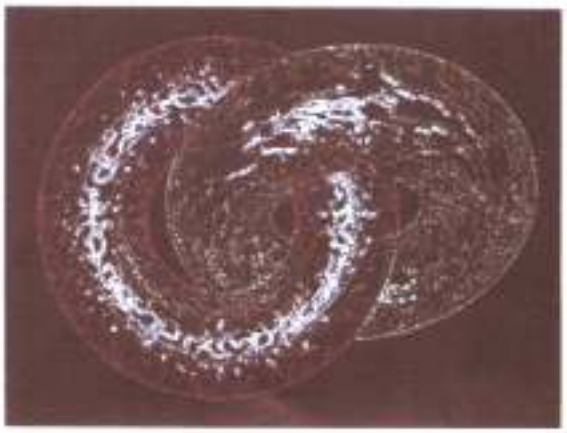


图 10-13 两个经过过滤操作处理的经典图像：低通过滤的结果模糊不清，因为它除去了高频部分；高通过滤通过移除变化度的变量（低频率成分）来强调细节



图 10-22　看进角色／物体交互的问题。如果用同样的动画脚本处理“拾”相似的大物体，就会出现角色／物体的穿透现象（在多像中可见）。另一个可选方案是，为每个物体写不同的脚本

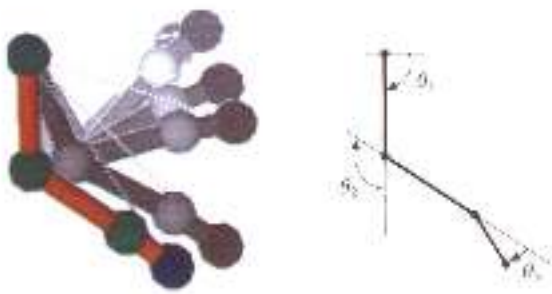


图 11-8　带关节角度约束的三链臂结构的微分 IK 解法

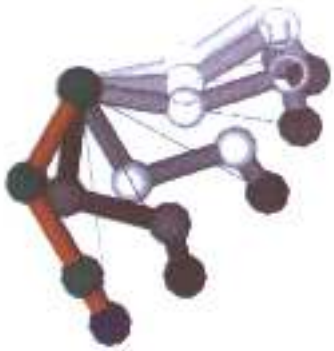


图 11-9　与上面的解法相比，不带关节约束的三链臂结构的情形

出版者的话

文艺复兴以降，源远流长的科学精神和逐步形成的学术规范，使西方国家在自然科学的各个领域取得了垄断性的优势；也正是这样的传统，使美国在信息技术发展的六十多年间名家辈出、独领风骚。在商业化的进程中，美国的产业界与教育界越来越紧密地结合，计算机学科中的许多泰山北斗同时身处科研和教学的最前线，由此而产生的经典科学著作，不仅擘划了研究的范畴，还揭橥了学术的源变，既遵循学术规范，又自有学者个性，其价值并不会因年月的流逝而减退。

近年，在全球信息化大潮的推动下，我国的计算机产业发展迅猛，对专业人才的需求日益迫切。这对计算机教育界和出版界都既是机遇，也是挑战；而专业教材的建设在教育战略上显得举足轻重。在我国信息技术发展时间较短、从业人员较少的现状下，美国等发达国家在其计算机科学发展的几十年间积淀的经典教材仍有许多值得借鉴之处。因此，引进一批国外优秀计算机教材将对我国计算机教育事业的发展起积极的推动作用，也是与世界接轨、建设真正的世界一流大学的必由之路。

机械工业出版社华章图文信息有限公司较早意识到“出版要为教育服务”。自1998年开始，华章公司就将工作重点放在了遴选、移译国外优秀教材上。经过几年的不懈努力，我们与Prentice Hall，Addison-Wesley，McGraw-Hill，Morgan Kaufmann等世界著名出版公司建立了良好的合作关系，从它们现有的数百种教材中甄选出Tanenbaum，Stroustrup，Kernighan，Jim Gray等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及庋藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作；而原书的作者也相当关注其作品在中国的传播，有的还专诚为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，为进一步推广与发展打下了坚实的基础。

随着学科建设的初步完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都步入一个新的阶段。为此，华章公司将加大引进教材的力度，在“华章教育”的总规划之下出版三个系列的计算机教材：除“计算机科学丛书”之外，对影印版的教材，则单独开辟出“经典原版书库”；同时，引进全美通行的教学辅导书“Schaum's Outlines”系列组成“全美经典学习指导系列”。为了保证这三套丛书的权威性，同时也为了更好地为学校和老师们服务，华章公司聘请了中国科学院、北京大学、清华大学、国防科技大学、复旦大学、上海交通大学、南京大学、浙江大学、中国科技大学、哈尔滨工业大学、西安交通大学、中国人民大学、北京航空航天大学、北京邮电大学、中山大学、解放军理工大学、郑州大学、湖北工学院、中国国家信息安全测评认证中心等国内重点大学和科研机构在计算机的各个领域的著名学者组成“专家指导委员会”，为我们提供选题意见和出版监督。

这三套丛书是响应教育部提出的使用外版教材的号召，为国内高校的计算机及相关专业

的教学度身订造的。其中许多教材均已为M. I. T.，Stanford，U.C. Berkeley，C. M. U. 等世界名牌大学所采用。不仅涵盖了程序设计、数据结构、操作系统、计算机体系结构、数据库、编译原理、软件工程、图形学、通信与网络、离散数学等国内大学计算机专业普遍开设的核心课程，而且各具特色——有的出自语言设计者之手、有的历经三十年而不衰、有的已被全世界的几百所高校采用。在这些圆熟通博的名师大作的指引之下，读者必将在计算机科学的宫殿中由登堂而入室。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证，但我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。教材的出版只是我们的后续服务的起点。华章公司欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

电子邮件：hzedu@hzbook.com
联系电话：（010）68995264
联系地址：北京市西城区百万庄南街1号
邮政编码：100037

专家指导委员会

（按姓氏笔画顺序）

尤晋元	王　珊	冯博琴	史忠植	史美林
石教英	吕　建	孙玉芳	吴世忠	吴时霖
张立昂	李伟琴	李师贤	李建中	杨冬青
邵维忠	陆丽娜	陆鑫达	陈向群	周伯生
周立柱	周克定	周傲英	孟小峰	岳丽华
范　明	郑国梁	施伯乐	钟玉琢	唐世渭
袁崇义	高传善	梅　宏	程　旭	程时端
谢希仁	裘宗燕	戴　葵		

秘　书　组

武卫东　　温莉芳　　刘　江　　杨海玲

译 者 序

近年来，随着数字技术突飞猛进的发展，计算机图形学（Computer Graphics，CG）作为一种图形设计的方法及工具，已在世界范围内得到了普遍的重视，成为计算机科学中最为活跃的领域之一。计算机图形学是一门建立在计算机科学、数学、物理、心理学以及艺术基础上的综合学科，它主要研究如何在计算机中表示图形，以及利用计算机进行图形计算、处理和显示的相关原理与算法，目前已被广泛地应用于娱乐、计算机辅助设计（Computer Aided Design，CAD）、科学可视化及系统可视化领域，形成了一系列新的研究方向。

在当今世界上，在CG技术方面处于领先地位的国家是美国和日本。自1968年美国科学家第一次在实验室中将自己亲属的照片扫描进计算机（这也是对计算机图形学的首次尝试）开始，计算机图形学已经在美国发展了整整35年，其中1975年开始举办的SIGGRAPH（计算机图形艺术联合）展不仅极大推动了美国CG技术的发展，而且已经发展成为世界CG技术的年度展览会。同时，日本也依赖其特有的动漫文化产业的支持，为全球CG业，尤其是电子游戏及动画领域输送了大量人才，推动了各种游戏软件及硬件的发展。如今在这些国家，CG技术已经广泛深入到影视制作、游戏制作、个人艺术创作、多媒体教育等社会各个层面，每年给国家带来近千亿美元的经济利润。可以说，CG已经成为一种产业，深刻影响着一个国家的经济和文化发展。

三维游戏开发是计算机图形学中一个重要的研究方向，对开发人员而言，不仅需要熟练掌握图形学中的原理和技术，同时还需具备人工智能、计算机网络等多方面的知识。随着游戏产业朝着专业化的方向发展，游戏程序的开发逐渐分离为两大块内容：开发游戏引擎和使用游戏引擎开发游戏。其中游戏引擎是针对某一类特定形式的游戏所定制的二次开发平台，它使游戏设计者可以较少地关心程序技术本身，而专心致力于游戏可玩性的设计。本书旨在为当今的三维游戏引擎技术提供一个综合的解决方案，将游戏理论技术与具体引擎代码分析相结合，使读者阅读后能够初步具备游戏引擎的开发能力。

本书分为两卷。卷1主要从理论上探讨最新的游戏引擎技术，包括建模技术、真实感图形生成技术、实时渲染技术、动画技术，以及声音、输入输出控制和物理引擎等高级技术，力求使读者能有一个较完备的理论知识背景，在日后开发中对各方面的技术具备系统的把握能力。

卷2从实践的角度出发，具体描述了一个游戏引擎的构建过程，着重于基本开发工具的使用、软件构架、开发技巧和优化等方面，并结合具体的游戏代码详细说明，帮助读者将卷1的理论付诸于实践，使读者尽快进入开发者角色。此外，卷2还初步介绍了游戏设计原理和游戏引擎的使用，主要目的在于使游戏引擎开发者能够了解整个游戏开发过程和客户（游戏设计者）的需求。

本书最后给出了一个完整的三维游戏引擎实例，该引擎采用C++语言开发，基于Windows平台运行，并且支持DirectX9。我们鼓励读者在看懂实例的基础上进一步加入自己的想法，不断完善，从而增强实际动手能力，成为一名专业的游戏引擎开发人员。

由于时间仓促，翻译中出现的任何不妥之处还请各位读者见谅。在一年多的翻译过程中，复旦大学的领导和老师给予了大力的支持和帮助，同时我们还得到了复旦大学计算机科学与工程系高性能可视化仿真实验室的协助。本书主要由沈一帆、陈文斌、朱怡波翻译，参加翻译的还有金万军、顾源泓、张睿、牟涛、滕莉、陈峥、李宏宇、施荣杰、岳军、李晨佳、周斌、杨树林、陆瀛海、吴鸿智、杜浩、葛云鹏、曹杰、曹贡献、张蕊、吴若寒、雷迦吟、罗毅、陈珊珊、黄俊青、梁玉婷、朱祯菁。

译　者

2004 年 11 月

前　言

这本书主要介绍游戏制作中一些高级的技术，分为以下三个部分：

（1）一般过程，包括：

- 构造过程——游戏系统所需要的离线处理过程。
- 实时过程——游戏引擎执行的与应用无关的过程。
- 软件设计——如何用科学的设计把所有相关元素整合起来，如何将应用和游戏引擎结合。

（2）实时渲染过程。实时渲染的基础理论和目前已有的实践将在各自的章节中加以介绍。另一章介绍几何处理（也就是多边形网格理论）中一些重要的和容易被忽略的问题。

（3）角色动画的理论和实践。

本书中这些主题将结合一个具体的游戏系统 Fly3D SDK 2 加以讲述，这样做的目的是在一个实际的游戏系统中尽可能多地引出文中介绍的那些技术。由于这里既考虑了理论又考虑了实践，所以我们有必要选择一个特殊的设计方式。希望这种特殊化不会对读者理解一般原理造成影响。

并不是所有文中介绍的技术都在 Fly3D 中得到实现，有一些章节是纯理论的。然而，就是在这些理论性的章节中，大多数的重要方法也已被独立地实现了。对此，我们希望感兴趣的读者能得到足够的知识，从而可以在我们提供的引擎或者他们自己的引擎上实现这些技术。

在编写本书的时候（2002 年 8 月），游戏与其他应用相结合的潜力很大。用较小的处理成本实现较复杂的场景目前已经可行了。文中所提到的今后的一个发展目标就是构造用于除游戏以外的其他应用的工具。

引擎是执行游戏的系统，实用程序是像编辑器、BSP 构造器那样的工具。通过这些工具，可以设计游戏，并可进行一些必要的预处理操作，通过引擎为实时处理准备好游戏数据。游戏引擎利用离线阶段构造好的有效数据结构来执行游戏。除了有效地渲染静态内容，引擎还必须能处理动态物体间的交互以及游戏与玩家间的交互。

[WATT01] 中介绍了一个基本的游戏引擎和相关的 BSP 理论。文中还涉及了用在 Fly3D 第 2 版中的更先进的技术。这些发展体现了经验的演化和不断优化得到新方法的过程。我们希望这些结果代表了游戏系统发展的现状。

系统的特性有：

-/+	完全插件导向
-/+	支持复杂地形和封闭环境的 BSP/PVS 绘制管理。BSP 的一般递归
+	通过使用凸体的伪入口和 A*算法自动地寻找路径
+	至多有 8 通道的固定功能的着色器
-	多纹理支持
+	着色器集成了硬件顶点编程和像素编程

-/+	为静态几何结构创建带有柔和阴影的光照贴图
-	带有雾贴图的容积雾
-	用来更新光照贴图和动态物体的带有距离衰减的动态有色光
+	动画网格（带有动画混合（animation blending）的顶点变形）
+	带有骨架的动画网格（带有动画混合的骨骼/皮肤）
+	用逆向运动学进行 MoCap 的重定位
0	支持着色器的网格文件格式（.F3D）
0	骨架动画网格文件格式（.K3G）
+	几种表面的类型：大的表面（超过三个顶点）、三角带、三角扇、三角汤（tri-soup）和曲面片
-	对于使用双二次贝济埃面片的曲面进行动态 LOD
+	硬件渲染选项，包括卡通/轮廓渲染
-	动态阴影——使用模板（stencil）阴影体
-	采用客户机/服务器架构（使用 DirectPlay）的多用户支持
0	立体声和三维音效支持（使用 DirectSound）
-	鼠标和键盘输入（使用 DirectInput）
-	Intel Pentium3 的矢量和矩阵数学优化
0	直接从压缩文件中读取数据
+	带有实时预览的 .fly 文件脚本编辑器
+	带有实时预览的 .shr 文件着色器编辑器
0	3D Studio Max（版本 2～4）的插件，用于输出包含任意数量动画（每个都可以有任意数量关键帧）的 .F3D 和 .K3G 文件
0	3D Studio Max（版本 3、4）的插件，用于输入和输出 .FMP（Fly3D Map）层次文件
0	flyBuild 贴图处理应用程序和 flyBuild 构造前端
0	Quake 3 层次转换器允许使用与 Quake3 兼容的层次编辑器

- 代表在［WATT01］中所描述的部分
+ 代表本书涉及的部分
0 代表没有在文章中涉及但出现在光盘中的实用程序

引擎和 SDK 的功能使开发者能够用尽可能低的处理成本实现高级的视觉效果。这种发展以及美术制作工具的演化已经成为现代游戏的一个重要方面，因为它使得设计师和美术师能够集中精力创作，同时也将程序员和美术师在游戏制作过程中扮演的角色分割开来。我们将在第 3 章中给出典型例子。

随着硬件性能的不断提高以及更高级的管理和渲染应用的发展，使更高复杂度的实现成为可能，并使游戏引擎除了游戏之外，可以开发出更多潜在的应用。引擎现在已经被应用到如下领域：

- 三维游戏：两种游戏类型在文中被广泛地作为例子。
  常见的第一人称射击类游戏，玩家在一个复杂的场景中漫游。
  非线性的角色动画类型，主角被玩家操纵在环境中自主地漫游，并与周围的物体

交互。

- 三维互联网应用/游戏：提供 ActiveX 控件，包括 Fly3D 所有的功能。互联网上实时的三维模拟是非常吸引人的。可能的应用包括在三维环境中购物、三维浏览和三维行销应用。
- 三维建筑的实时漫游：游戏引擎为一个潜在的应用是实时的建筑内部的展现。图 P-1（彩页中也有）显示了游戏引擎在购物超市和公寓的设计中被用作一个 CAAD 工具。
- 通用的三维可视化：复杂性主要来源于几何形状的复杂和高级渲染技术的需要。我们设计了一个包含此类场景的例子——火星部分表面的多边形表示（见图 1-7）。

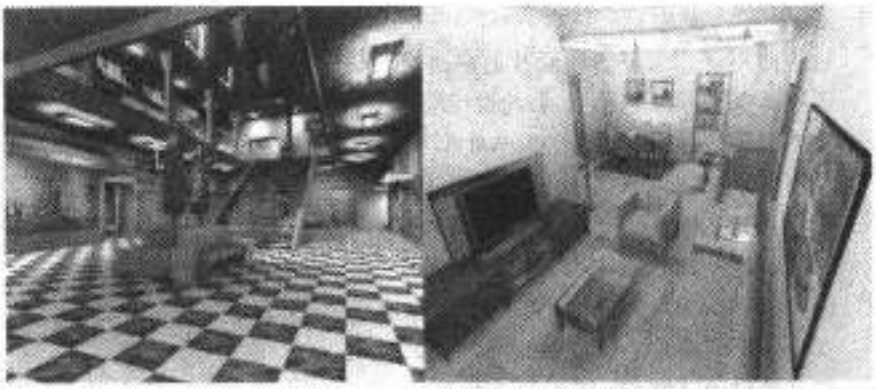


图 P-1　在非游戏应用中使用的引擎

目　录

出版者的话
专家指导委员会
译者序
前言

第一部分　高级游戏系统剖析

第 1 章　高级游戏系统剖析Ⅰ:构造过程和静态光照 …… 1
1.1　数据结构 …… 1
1.1.1　顶点 …… 1
1.1.2　面 …… 2
1.1.3　包围盒 …… 4
1.2　构造过程 …… 4
1.2.1　从场景几何中创建 BSP 树 …… 4
1.2.2　路径规划的凸体和 PVS 计算 …… 11
1.2.3　处理复杂的地形 …… 13
1.2.4　BSP 叶节点中的面 …… 16
1.2.5　寻找叶凸体 …… 16
1.2.6　凸体和伪入口 …… 19
1.2.7　潜在可视集 …… 24
1.3　光照贴图的构造 …… 28
1.3.1　生成光照贴图的坐标 …… 28
1.3.2　光照贴图的打包 …… 29
1.3.3　对光照贴图的解释 …… 30
1.4　BSP 管理 …… 31
1.5　高级静态光照——辐射度 …… 37
附录 1.1　构造实践 …… 49
附录 1.2　辐射度理论基础 …… 64
第 2 章　高级游戏系统剖析Ⅱ:实时处理 …… 69
2.1　视见和 BSP …… 69
2.1.1　生成视见约束体的面 …… 69
2.1.2　远近裁剪面和视见约束体 …… 75
2.2　照相机控制 …… 77
2.3　使用 BSP 的基本碰撞检测和反弹 …… 80
2.3.1　碰撞和 BSP 遍历 …… 80
2.3.2　粒子/场景检测和反弹 …… 81
2.4　特殊的碰撞检测和反弹 …… 83
2.4.1　AABB 的定义 …… 84
2.4.2　AABB 类的定义和静态成员的定义 …… 84
2.4.3　碰撞检测和碰撞反弹 …… 86
2.4.4　使用 AABB 的伪碰撞反弹 …… 87
2.4.5　使用 AABB 的碰撞检测 …… 88
2.4.6　AABB 顶点与场景面相交 …… 89
2.4.7　场景顶点与 AABB 面相交 …… 90
2.4.8　AABB 边与场景边相交 …… 92
2.4.9　更精确的碰撞检测 …… 95
2.4.10　使用碰撞阈值 …… 95
2.5　基本的路径规划 …… 96
附录 2.1　实时处理的演示 …… 102
第 3 章　高级游戏系统剖析Ⅲ:软件设计与应用编程 …… 105
3.1　应用的种类 …… 105
3.1.1　插件 …… 105
3.1.2　前端 …… 111
3.1.3　工具 …… 115
3.2　Fly3D 引擎体系结构 …… 115
3.2.1　FlyMath …… 115
3.2.2　FlyDirectX …… 116
3.2.3　FlyRender …… 118
3.2.4　FlyEngine …… 120
附录 3.1　编写一个插件 …… 130

第二部分　实时渲染

第 4 章　实时渲染 …… 145
4.1　简介 …… 145
4.2　顶点、像素和贴图 …… 146
4.2.1　基本的逐像素着色 …… 146
4.2.2　着色和坐标空间 …… 148
4.2.3　25 年来主流的插值着色方法和颜色贴图 …… 149
4.2.4　标量表示 …… 151

4.3 因式分解法 …… 156
4.3.1 使用因式分解着色模型的逐像素着色——各向同性模型 …… 157
4.3.2 使用因式分解着色模型的逐像素着色——各向异性模型 …… 162
4.4 BRDF 和真实材质 …… 164
4.5 使用 BRDF 进行逐像素着色 …… 167
4.6 环境贴图参数化 …… 170
4.6.1 环境贴图参数化:立方映射 …… 170
4.6.2 环境贴图参数化:球面映射 …… 171
4.6.3 环境贴图参数化:对偶抛物面贴图 …… 174
4.6.4 环境贴图——可比点 …… 176
4.6.5 立方贴图和向量规范化 …… 177
4.7 实现 BRDF:可分离的近似 …… 177
4.8 着色语言和着色器 …… 181
4.8.1 着色语言:简单的历史回顾 …… 181
4.8.2 RenderMan 着色语言 …… 182
4.8.3 实时渲染的着色语言 …… 184
第 5 章 实时渲染:实践 …… 192
5.1 基本着色器 …… 192
5.1.1 渲染状态 …… 192
5.1.2 着色器排序 …… 193
5.1.3 着色器类的实现 …… 195
5.2 渲染状态 …… 196
5.2.1 全局设定 …… 196
5.2.2 局部设定 …… 196
5.3 着色器实例 …… 200
5.3.1 环境映射和铬映射效果——玻璃、金属和铬 …… 201
5.3.2 移动发光告示牌 …… 202
5.3.3 简单栅栏效果 …… 203
5.3.4 高级栅栏效果 …… 203
5.3.5 监视器效果 …… 204
5.4 实时硬件渲染 …… 207
5.4.1 顶点编程 …… 207
5.4.2 像素编程 …… 219
5.4.3 使用寄存结合器的像素编程 …… 219
5.4.4 纹理地址编程 …… 223
5.4.5 纹理地址编程——Phong 映射 …… 224
5.4.6 顶点和像素编程以及多步着色器 …… 224
5.5 动态纹理 …… 224
5.6 特效 …… 228
5.6.1 燃烧尾迹 …… 228
5.6.2 加速器 …… 230
5.6.3 脉冲星 …… 230
附录 5.1 使用和探索着色器 …… 233
附录 5.2 NVIDIA GeForce 3 上的顶点编程 …… 235
附录 5.3 NVIDIA 寄存结合器操作 …… 237
第 6 章 几何处理 …… 240
6.1 简介 …… 240
6.2 推动因素和定义 …… 241
6.2.1 离线和实时阶段 …… 241
6.2.2 拓扑因素 …… 242
6.2.3 离散简化与连续简化 …… 242
6.2.4 物体内部分辨率变化 …… 243
6.2.5 对称性/可逆性 …… 243
6.2.6 局部简化操作 …… 243
6.3 排序(误差)标准 …… 244
6.3.1 排序标准——外观相似 …… 244
6.3.2 排序标准——局部体积不变 …… 245
6.3.3 排序标准——二次误差度量 …… 246
6.3.4 排序标准——简化外壳 …… 247
6.4 简化与属性 …… 248
6.4.1 简化与游戏纹理 …… 250
6.4.2 简化和蒙皮模型 …… 250
6.4.3 算法框架 …… 250
6.4.4 顶点去除算法的重新三角形划分 …… 251
6.5 实例分析 …… 252
6.5.1 实例分析 1——渐近式网格技术 …… 252
6.5.2 实例分析 2——使用微分几何 …… 256
6.5.3 实例分析 3——网格重新划分算法 MAPS …… 260
附录 6.1 数学背景 …… 264
附录 6.2 演示 …… 269

第三部分 动画制作

第 7 章 角色动画 …… 271
7.1 简介 …… 271
7.2 顶点动画与合成 …… 274
7.3 骨架动画 …… 279

7.4 低层次动画管理 …… 284
7.4.1 行进的路径规划 …… 286
7.4.2 骨架动画和面向对象的动画控制 …… 290
7.4.3 对障碍物的躲避 …… 290
7.4.4 路径规划总结 …… 292
附录7.1 用四元数描绘旋转 …… 292
附录7.2 四元数的实现 …… 299
附录7.3 角色动画中效率的考虑 …… 303
第8章 动画成形方法 …… 310
8.1 简介 …… 310
8.2 样条框架 …… 311
8.3 自由形状变形 …… 312
8.4 扩展自由形状变形(EFFD) …… 315
8.5 曲线变形——铰线 …… 316
8.6 皮肤控制 …… 318
8.6.1 面向表面的自由形状变形(SOFFD) …… 318
8.6.2 骨架皮肤精致化 …… 319
8.6.3 组合皮肤和形状混合 …… 322
附录8.1 使用径向基函数进行离散数据插值 …… 324
第9章 高级角色动画之要素 …… 325
9.1 引言——一种拟人的游戏界面 …… 325
9.2 将语言表述转变为动画——示例 …… 326
9.2.1 IMPROV(纽约大学媒体研究实验室) …… 327
9.2.2 PAR体系结构(宾夕法尼亚大学人体建模和仿真中心) …… 329
9.2.3 具体化的对话界面代理(MIT媒体实验室) …… 330
9.2.4 游戏结论 …… 331
9.3 面部动画、视觉语音和跟踪 …… 332
9.4 用于控制、渲染和跟踪面部网格的模型 …… 333
9.4.1 基于图像的建模、渲染和跟踪 …… 334
9.4.2 跟踪方法 …… 336
9.4.3 参数化 …… 341
9.4.4 伪肌肉模型 …… 343
9.4.5 面片技术 …… 345
9.5 视觉语音 …… 348
9.6 面部动画和MPEG-4 …… 351
9.7 渲染问题 …… 351
9.8 总结和问题 …… 353
9.8.1 参数化与照片真实性 …… 353
9.8.2 网格表示 …… 353
9.8.3 皮肤的渲染 …… 353
9.8.4 没有声音很多面部动画更好看 …… 353
9.8.5 情感和语音 …… 353
附录9.1 一个伪肌肉模型的实现 …… 355
第10章 基于运动捕捉的角色动画 …… 356
10.1 简介 …… 356
10.2 运动数据 …… 358
10.3 骨架和MoCap——BVH格式 …… 359
10.4 运动数据的基本处理 …… 361
10.4.1 加速和减速运动 …… 361
10.4.2 混合和时间扭曲 …… 362
10.4.3 对齐运动序列 …… 363
10.4.4 运动扭曲 …… 364
10.5 MoCap中的插值 …… 365
10.5.1 B样条表示法 …… 365
10.5.2 运动混合——动词和副词 …… 367
10.6 经典信号处理和MoCap …… 368
10.6.1 傅里叶理论 …… 369
10.6.2 傅里叶理论和非周期数据 …… 373
10.6.3 傅里叶理论和采样数据 …… 374
10.6.4 采样和走样现象 …… 377
10.6.5 反走样滤波器 …… 379
10.6.6 时间域中的过滤——卷积 …… 379
10.7 信号处理和MoCap数据 …… 380
10.7.1 傅里叶域中的插值/外推法 …… 381
10.7.2 使用拉氏算子的多分辨率滤波 …… 382
10.8 运动编辑:基于约束的方案 …… 383
10.8.1 运动中的动力学约束 …… 384
10.8.2 运动中的运动学约束 …… 386
10.8.3 每帧重定位法 …… 387
10.8.4 时空法 …… 390
附录10.1 示范 …… 390
第11章 反向运动学原理 …… 391
11.1 例子——二链臂 …… 392
11.2 雅可比矩阵 …… 394
11.3 IK方法 …… 396
11.3.1 使用雅可比阵的微分方法 …… 396

11.3.2 最优法 …… 400
11.3.3 循环坐标下降法(CCD) …… 401
11.4 反向运动学的实践方案 …… 403
11.4.1 混合方法——分析法 + 约束最优化法 …… 404
11.4.2 混合方法——三阶段:分析法 + 约束最优化 + 分析法 …… 404
11.4.3 防止自碰撞 …… 407
11.4.4 IK 与运动目标 …… 409
参考文献 …… 410

第一部分　高级游戏系统剖析

第1章　高级游戏系统剖析Ⅰ：构造过程和静态光照

离线（off-line）过程通常被分解为构造的过程。工具集（utilities）用来构造一个文件，引擎通过此文件来执行它的实时过程。整个构造过程产生如下：

1）将内容构造器（content builder）中的输出转化为一个具有最优数据结构的图文件。

2）构造 BSP 结构。

3）建立光照贴图，这一步本身又分为三步：

a）生成光照贴图中的坐标

b）光照贴图包装（packing）

c）阐述光照贴图

4）构造潜在可视集（Potential Visible Set，PVS）。

BSP 的构造最先发生，这是因为光照贴图和 PVS 的构造会削减 BSP 的效率。因此，在构造过程和游戏运行过程中，BSP 管理都在运行。

1.1　数据结构

在构造过程之初，内容构造器软件的输出被转化为我们所需的某种优化形式。数据结构对于复杂游戏的运行效率来说是举关重要的。运行效率必须同时考虑以下两个因素：所用的几何图形存储方法和图形硬件的运用（尤其是新的扩展运用）。我们首先考虑存储顶点——最底层的数据结构。

1.1.1　顶点

下面给出一种常规的存储顶点的方法（其他几种类似方法在这里不再列出）。

```
class FLY_ENGINE_API flyVertex
{
public:
   vector pos;
   vector texcoord;
   vector normal;
   unsigned color;
};
```

这种方法用一个单一的结构表示所有顶点的属性。texcoord 变量存储了纹理坐标和光照贴图坐标（纹理坐标是两个单精度浮点数 x，y；而光照贴图坐标是另外两个单精度浮点数 z，w）。这表示，如果两个顶点在空间中拥有相同的位置，但是拥有不同的法线、颜色或者纹理坐标，那么这两点不能被共享（即必须用两个不同的顶点去表示它们）。

这样我们存储了一个在层次中包含所有所用顶点的顶点矩阵。通常，一个具有相同位置

坐标，不同纹理坐标、颜色等属性的顶点实体会出现很多次。

```
flyVertex *vert=new flyVertex[nvert];
```

每一个面由数量固定且在顶点矩阵中是一个相继的整体的顶点集确定。这表示，矩阵中最初 n 个顶点表示面 0，而接下来的 m 个顶点表示面 1，依次类推；而且没有一个面与另一个面共享同一个顶点。这就使得顶点矩阵的使用总是可行的。假如上述任意的信息被存储在面结构中，那么我们就不能使用顶点矩阵，这是因为相继的整体间的位移不是一个常数。例如对三角形来说，每一个面我们可以给出三个纹理坐标。这就会使得不同的面要共享某些相同的顶点，但是不允许使用顶点矩阵，这是因为纹理坐标之间不会被固定大小的字节分隔，所以至少应有三个或者三个以上连续的纹理坐标。

1.1.2 面

面分为五类：

1）一个大的多边形（n 个顶点的多边形）

2）一个贝济埃面片（$npu \times npv$ 个顶点定义其控制网格）

3）三角“汤”（tri-soup）（n 个顶点定义三角形网格）

4）三角带（tri-strip）

5）三角扇（tri-fan）

使用可变长度还是固定长度的面结构是一个重要的考虑因素。固定长度的结构相对更好一些。当我们遇到无组织化的顶点矩阵时，需要可变长度结构，而且顶点矩阵所表示的面需要一个可变长度的指针列表去指向这个顶点矩阵。

当采用有组织的顶点矩阵时，面结构仅仅需要两个整数（顶点序列由顶点的数目决定）。不管顶点的数目被参考与否，面结构总有固定的大小。

下面是面结构的类定义：

```
class FLY_ENGINE_API flyPlane
{
  public:
    vector normal; // plane normal
    float d0;      // the perpendicular distance
                   // from the plane to the origin

  // computes the perpendicular distance from a point
  // to the plane

  inline float distance(vector &v)
    { return vec_dot(normal,v)-d0; }
};

class FLY_ENGINE_API flyFace : public flyPlane
{
  public:
    int facetype; // face type (1,2,3,4,5)

    int sh;       // shader
    int lm;       // light map

    int nvert;         // number of vertices
    int vertindx; // first vertex

    int patch_npu;     // only facetype==FACETYPE_PATCH
```

```
    int patch_npv;
    flyBezierPatch *patch;
    int ntriface;       // only facetype==FACETYPE_TRUSURF
    int ntrivert;
    int *trivert;
}
```

最初的五个变量在所有面类型中出现。某些面类型需要更多的数据。对于第一类的面（多边形），所有顶点必须共面并且是凸多边形。尽可能多地使用大的多边形是很重要的。考虑矩形——用四个顶点表示，而用三角形表示时，我们必须向图形硬件传送六个顶点。

我们用以下的 OpenGL 命令画多边形（从顶点 *vertindx* 开始画出由 *nvert* 顶点组成的多边形）。

```
glDrawArrays(GL_POLYGON,vertindx,nvert);
```

对于第二类的面（双二次贝济埃面片），我们需要指定两个以上的整数（在每个方向 u，v（*npu*，*npv*）上控制顶点的数目）。和其他类型的面一样，每个控制点的顶点都存储于顶点矩阵中。*npu* 与 *npv* 的乘积等于矩阵中代表顶点的控制点个数。

我们使用控制点从顶点 vertindx 开始，并用 nvert 顶点建立一个面片对象。

```
nvert=npu*npv;
patch=new flyBezierPatch;
patch->set_control_points(npu,npv,&vert[vertindx]);
```

我们使用三角带画面片。贝济埃面片具有可被用作一个简单细节层次（LOD）工具设备的优势［WATT01］。基于几何学错误的因素考虑，它们被极有效地均匀细分为最高层次的细节。在渲染过程中，通过跳过三角片中的行和列，取得一个与观察距离相称的 LOD。我们使用 flyBezierPatch 类中的 draw（）函数来画有动态 LOD 的面片模型。

第三类的面（三角汤）用来模拟细节层次（如灯、雕像等）。它们需要一个新的整数来定义三角形面的个数（*ntriface*），以及一个用来检索构成三角形面的面顶点 3**ntriface* 的矩阵。要注意的是，这样的面在三角形之间可以共享顶点。大的多边形由一个层次的体系结构组成，而小的对象以背景墙的一个部分出现，二者之间存在差异。这些通过随后的 BSP 方案做出不同处理。只有第一类面可以被用作 BSP 的划分平面。第三类面所表示的对象是细节上的小对象，它们出现在所有那些被其夹住的 BSP 节点中。第三类的面需要一个额外的整数矩阵来定义顶点之间的三角剖分。而每个三角形在矩阵中有三个整数检索顶点。

```
ntrivert=ntriface*3;
trivert=new int[ntrivert];
```

对于这一类型的面，我们拥有可以生成 m 个三角形面的 n 个顶点（面可以共享相同的顶点）。我们用以下的 OpenGL 命令画出它。

```
glDrawElements(GL_TRIANGLES, ntrivert,GL_UNSIGNED_INT,trivert);
```

每个第四类的面（三角带）代表一个独立的三角片序列。如果面中包含 n 个顶点，则 $n-2$ 个三角形可以被表示出来。通过如下命令可以画出三角带。

```
glDrawElements(GL_TRIANGLE_STRIP,ntrivert,GL_UNSIGNED_INT,trivert);
```

每个第五类的面（三角扇）代表一个独立的三角扇序列。如果面中包含 n 个顶点，则 $n-2$ 个三角形可以被表示出来。通过如下命令可以画出三角扇。

```
glDrawElements(GL_TRIANGLE_FAN,ntrivert,GL_UNSIGNED_INT,trivert);
```

1.1.3　包围盒

所有静态面和动态对象被封装在轴对齐包围盒（AABB）中。如我们所见，这些被用在许多不同的最优化上下文中，包括碰撞检测和视见约束体选择（view frustum culling）。

1.2　构造过程

现在我们检验游戏中需要准备使用场景几何的操作序列并照亮它。场景几何在一个编辑器中建模，并且被保存在我们所知的某个图文件中。它由原始的场景图形（先前叙述中的一系列面及其顶点）组成。这是构造程序中的主要输入。这个程序生成了 BSP 文件、PVS 文件、光照贴图以及游戏代码。游戏代码是以层次中所定义的游戏实体以及游戏的类型为基础而生成的。

1.2.1　从场景几何中创建 BSP 树

为了区分结构化面和细节面而首先对层次几何进行详细阐述，是建立一个 BSP 树的好方法。结构化面是大的几何图形，例如那些代表背景墙的面，它们由于在 BSP 树中被用为分割的平面而有所不同。这样的面将自身形成为单一的凸体，即整个游戏场景（见图 1-1，彩页中也有）。

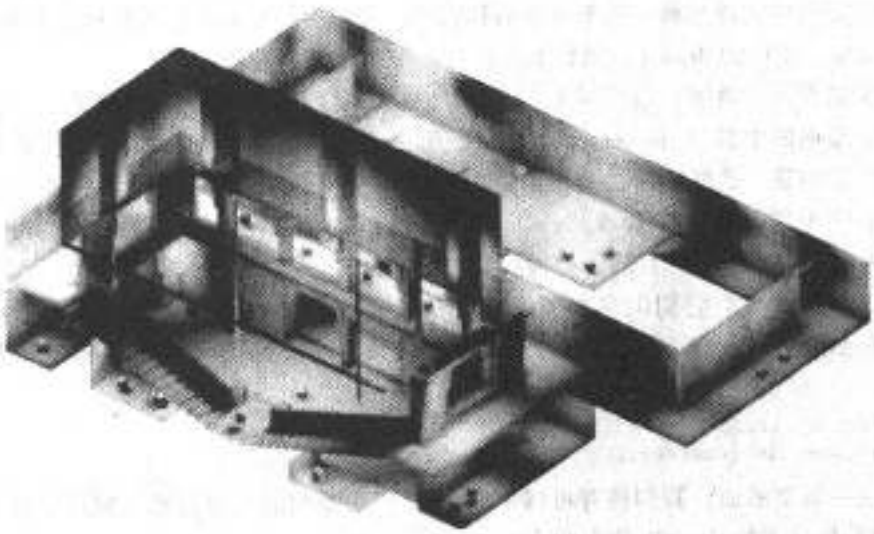
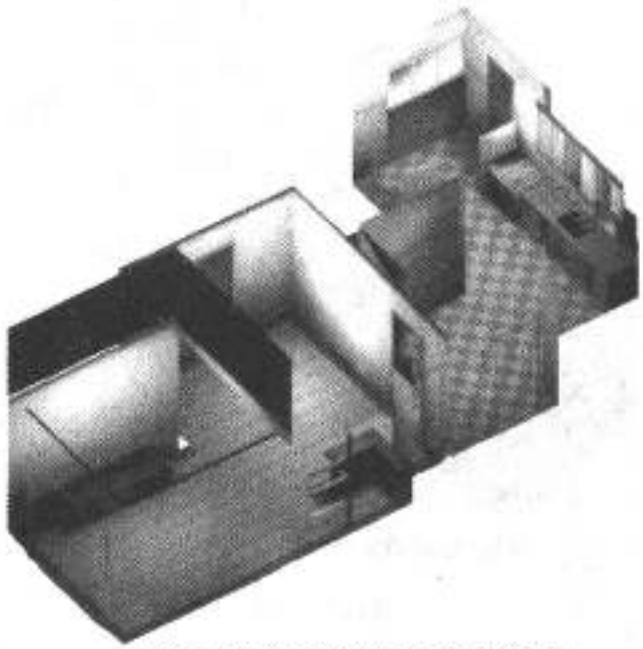


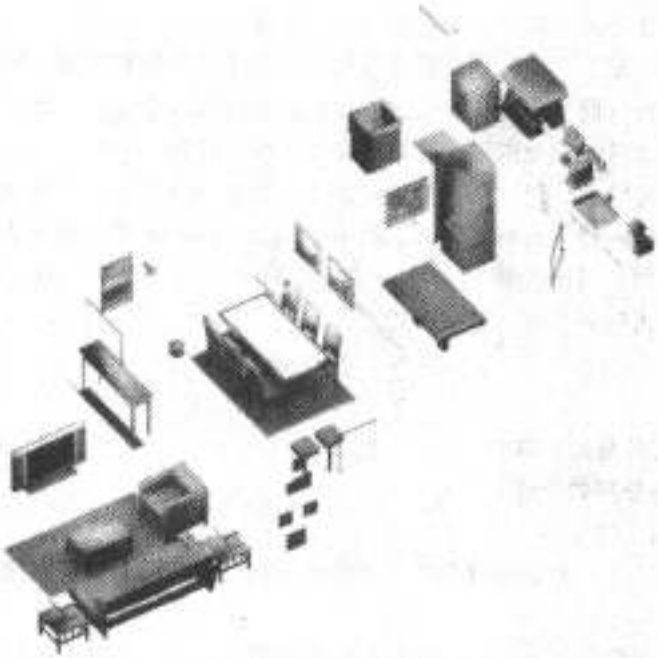
图 1-1　整体游戏层次视图

细节面属于被上述凸体所包含的小的对象。这个策略的目标是将一个场景分成最小数目的凸体。这种方法比仅考虑任意面与其他面所拥有的相同状态的方法要好得多。该方法导致了由小元素构成的面变成为分割平面的情况的出现。图 1-2（彩页中也有）展示了在一个计算机辅助体系结构设计（Computer Aided Architectural Design，CAAD）应用中一个层次在游戏引擎中的实现。在这里离所房间的墙壁形成了结构化面，而所有的家具以及墙上的着色等在

BSP构造过程中不起分割平面作用的对象被分类为细节面。因此包含在被其关住的BSP叶节点中。

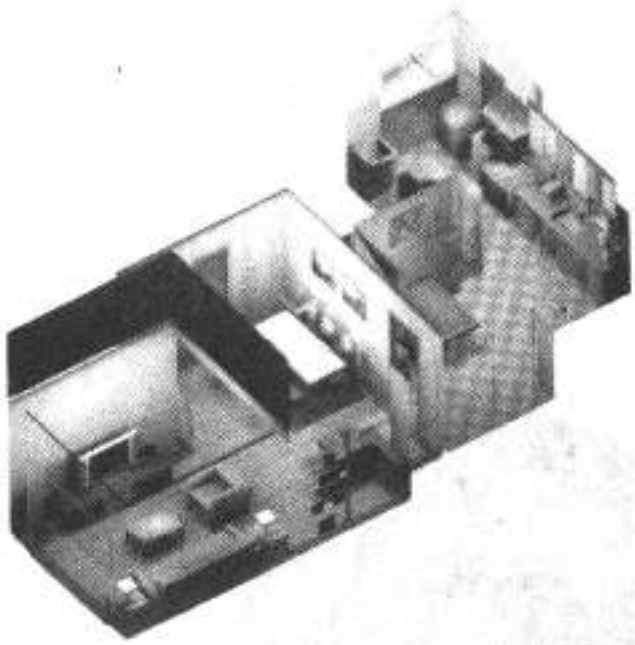


a) 在CAAD应用中起分割平面作用的结构化元素



b) 在CAAD应用中的细节元素

图 1-2



计算个高所定间房展示的结构化元素和细节元素

图 1-2　（续）

在 BSP 树中，每个节点与一个面序列和一个平面联系在一起。平面从结构化面中导出，通常平面的个数要比面的个数少。节点通过对一个递归函数的调用产生。这个函数会决定在可用平面中哪一个会是下一次分割的最佳候选者。若序列中没有更多的平面，那么当前的节点就被设计成叶节点。同样，若节点没有与之相联系的面，就会被删除。

为了选择节点中最好的分割平面，我们应考虑所有可用的平面。每个平面都会被检测以决定它们引用的最终结果。用一个计数器记录正向和逆向上的面的个数以及面的分割数目。最优的平面会显示出一种平衡的分割状态和不相交性。这个过程与每个节点的平面数目是非线性的关系——正如下面所论述的，这也是我们削减使用 BSP 区域复杂度的另一个调整。以下是一个有效评估每个平面度量的公式：

$$N1 - N2 + 4N3$$

其中：

N1 是正向分割的个数

N2 是逆向分割的个数

N3 是面相交的个数

这反映了一个事实，那就是四个面不平衡的作用与一个被相交的面被复制到两个子平面的作用相当。

被选的平面生成两个子平面，并且每个面会被测试以决定它在平面的一个侧面还是在另一个侧面上，又或者它自身被平面相交。相交的面被分割并且添加到它的两个子平面中。而被选择的平面会从当前节点的平面序列中删除。如果一个平面从未使用过，那么它仅仅被添

加到子平面的序列中。如此随着树的成长，节点会包含越来越少的面和平面（见图 1-3）。

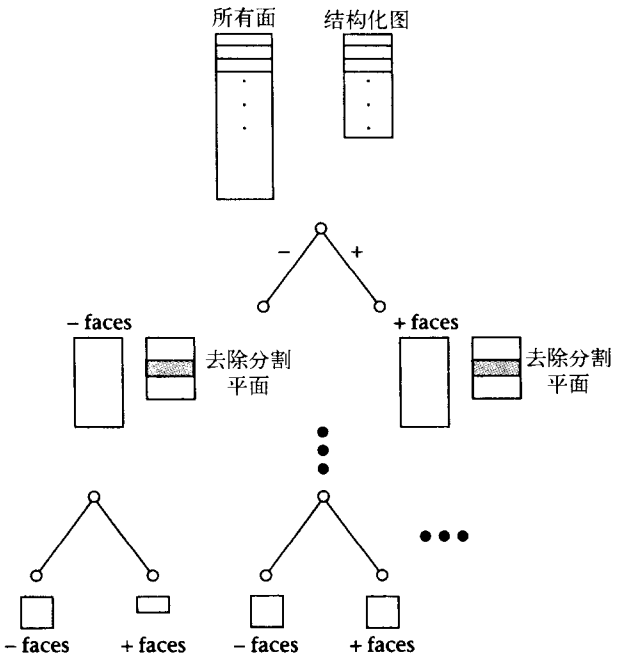


图 1-3　BSP 策略：每个（非叶）节点有一个面列表和一个候选分割平面列表

以下是寓所房间层次的统计表：

数据

顶点数目：　11975
面数目：　2652
需要包含面的独特平面数目：　339
包围盒大小：　706.00 1197.00 460.00

被选择单元的大小：　1024.00

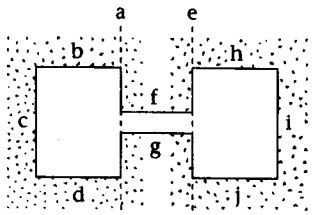
BSP 统计

BSP 区域：　2
叶面数目：　4913
树叶数目：　628（正向侧面 409 个，逆向侧面 219 个）
节点数目：　936

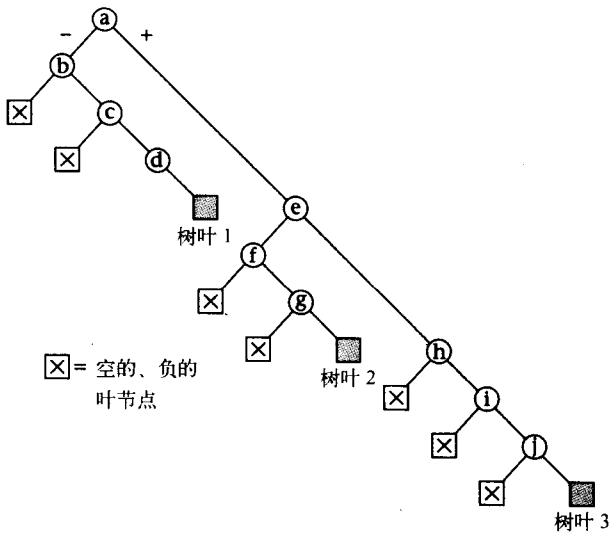
如上所述，包围盒的大小与整个层次相关联。这个概念用于更深度的详细阐述，它将把场景分成 BSP 区域，每个 BSP 区域拥有自己的 BSP 树。上述的例子只用了两个区域。BSP 构造过程与求分割平面数目的函数所需的复杂度之间不是线性的关系，并且对于例如地形的复杂情况，更深度的详细解释是必需的。这可以通过定义最大 BSP 单元大小，即地形包围盒的一个子划分来完成。整个层次被分成轴对齐的平面，直到单元的尺寸比预先定义的尺寸小为止。这些单元从而从属于先前描述的算法。这些单元划分有效地构成了结构化面。作为第

一步的近似，我们假设多边形的空间密度是一个常数，这样统一的划分是一个切合实际的策略。

图 1-4a 展示了一个简单层次的设计。它由 12 个表示墙壁的面和 3 个分别表示地板和天花板的面构成。如这个方法所示，它被分为形成叶节点的 3 个凸体。其结构由 18 个凸多边形（四边形）和 10 个平面组成。如果仅考虑墙壁，图 1-4b 展示了这个层次的递归构造过程。根节点选择了平面 *a*，而子节点序列选择了平面 *b*、*c*、*d*。它们中每一个生成了空的、负的节点，并且递归继续产生树叶 1。树叶 1 是平面 *a* 左边的凸体。这个体包含了所有属于这个凸体的面。该过程会继续以相似的方式生成另两个叶节点。



a)层次的计划图



b)简单层次的BSP树

图 1-4　对层次构造 BSP 树，导致在每个叶节点形成凸体

下面的代码实现了一个基本 BSP 的构造。它先读取一个图文件，然后创建一个根节点，并把所有的面指针加到根节点的面列表中，把所有结构化面的平面加到根节点的平面列表中。然后调用最初把场景划分进 BSP 区域的 *split_axis* 函数。

```
int flyEngineBuild::build_bsp(const char *fmpfile)
{
  if (load_fmp(str)==0)
    return 0;

  flyBspNodeBuild *bspb=new flyBspNodeBuild;
```

```
    int i;
    for( i=0;i<nfaces;i++ )
    {
      if (faces[i].flag==FLY_FACEFLAG_STRUCTURAL)
        bspb->add_plane(faces[i].normal,faces[i].d0);
      bspb->faces.add(&faces[i]);
    }

    bspb->split_axis();

    save_bsp(str))

    return 1;
  }
```

split_axis 从计算节点中的面的包围盒开始。在轴中的包围盒尺寸比网格尺寸要大的情况下，它将在一个与轴共线的平面中分割。该平面处于将盒子分成两半的中间位置。所有的面被分配到子节点中，并且在其每个子节点中这个函数还会被递归调用。当在每个轴中的节点包围盒比网格尺寸小时，节点会被处理。对于一个地形层次，在这一步时所有的面会被转换成一个拥有共有点的三角形网格。对于一个封闭的环境而言，在 BSP 区域中构造凸体的规范 BSP *split* 函数会被调用。

```
void flyBspNodeBuild::split_axis()
{
  int i,j;

  bbox.reset();
  for( i=0;i<faces.num;i++ )
    if (faces[i]->facetype==FLY_FACETYPE_LARGE_POLYGON &&
      faces[i]->flag && faces[i]->sortkey)
      for( j=0;j<faces[i]->nvert;j++ )
        bbox.add_point(faces[i]->vert[j]);

  for( i=0;i<3;i++ )
    if (bbox.max[i]-bbox.min[i]>bspgridsize)
      break;

  if (i==3)
  {
    bsp_sectors++;
    if (landscape==0)
      split();
    else
      trimesh_landscape();
  }
  else
  {
    normal.vec(0,0,0);
    normal[i]=1;
    d0=((bbox.max[i]-bbox.min[i])*0.5f+bbox.min[i]);

    child[0]=new flyBspNodeBuild;
    child[1]=new flyBspNodeBuild;

    for( i=0;i<faces.num;i++ )
    {
      j=classify_face(faces[i]);
      if (j==-1)
      {
        child[0]->faces.add(faces[i]);
```

```
            child[1]->faces.add(faces[i]);
            child[0]->add_plane(faces[i]);
         }
            else
            {
               child[j]->faces.add(faces[i]);

               child[0]->add_plane(faces[i]);
            }
         }

         faces.clear();
         planes.clear();

         child[0]->split_axis();
         child[1]->split_axis();
      }
   }
```

基于可用的平面，规范的 BSP *split* 函数方法找出最佳的候选者，然后递归调用，直到每个平面都被用过为止。

```
void flyBspNodeBuild::split()
{
   if (planes.num==0)
      return;

   int p=find_split_plane();
   if (p==-1)
      return;

   normal=planes[p].normal;
   d0=planes[p].d0;
   planes.remove(p);

   child[0]=new flyBspNodeBuild;
   child[1]=new flyBspNodeBuild;

   side=-1;
   child[0]->side=0;
   child[1]->side=1;

   float f1,f2;
   int i,j,n,flag;
   flyFace *face;

   for( i=0;i<faces.num;i++ )
   {
      face=faces[i];
      flag=0;
      for( j=0;j<face->nvert;j++ )
      {
         f1=distance(face->vert[j]);
         if (f1>-0.01f && f1<0.01f)
            continue;

         if (flag==0)

      {
         f2=f1;
         flag=1;
      }
      else
```

```
        if (f2*f1<0)
          break;
    }
    if (flag==0)
      n=0;
    else
    if (j==face->nvert)
      if (f2>=0)
        n=0;
      else
        n=1;
    else
      n=-1;

    if (face->flag==FLY_FACEFLAG_STRUCTURAL)
      j=find_plane(face->normal,face->d0);
    else j=-1;

    if (n!=-1)
    {
      child[n]->faces.add(face);
      if (j!=-1)
        child[n]->add_plane(planes[j].normal,planes[j].d0);
    }
    else
    {
      child[0]->faces.add(face);
      child[1]->faces.add(face);
      if (j!=-1)
      {
        child[0]->add_plane(planes[j].normal,planes[j].d0);
        child[1]->add_plane(planes[j].normal,planes[j].d0);
      }
    }
  }
  faces.clear();
  planes.clear();

  if (child[0]->faces.num)
    child[0]->split();
  else
  {
    delete child[0];
    child[0]=0;
  }

  if (child[1]->faces.num)
    child[1]->split();
    else
    {
      delete child[1];
      child[1]=0;
    }
  }
```

1.2.2　路径规划的凸体和 PVS 计算

构造过程的最终目标是为了划分为凸体。如果这是可行的，那么就如图 1-4 所示的一样，仅仅会有正向叶节点。这样的划分方案可用于路径规划（见 2.5 节）。然而，成功地完

成这一步意味着在层次建模过程中的约束。考虑图 1-1 中所展示的游戏层次的统计。

数据

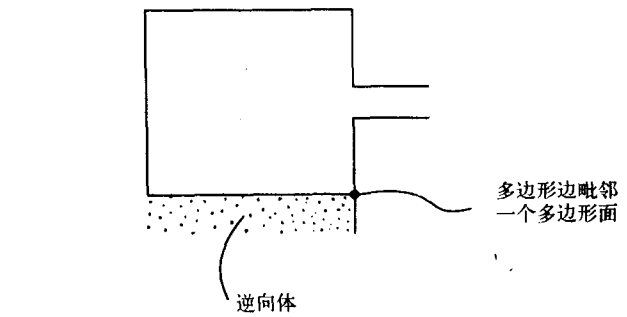
顶点数目：　3635
面数目：　733
需要包含面的独特平面数目：　166
包围盒大小：　1120.00 1630.00 677.00

被选择单元的大小：　1532.00

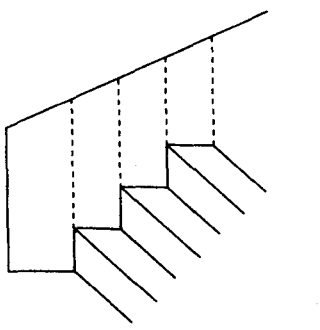
BSP 统计

BSP 区域：　2
叶面数目：　1615
树叶数目：　295（正向侧面 207 个，逆向侧面 88 个）
节点数目：　505

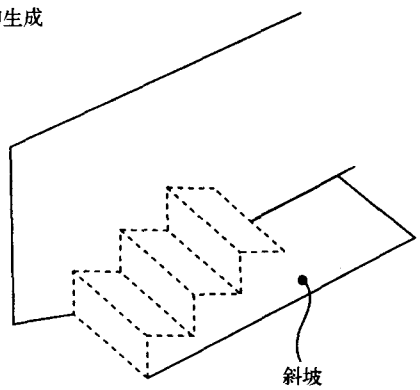
在这种情况下，88 个逆向节点来源于阶梯结构。凸体当且仅当多边形之间边与边相连时才可以被创立。如果一个多边形的边与另一个多边形的面邻接，那么以上的条件将不被满足(见图 1-5)。在这个层次阶梯结构中，我们必须把与阶梯邻接的墙壁多边形进行分割。换句话说，阶梯可以被给予细节状态，不过它们必须能通过一个不被渲染的溢出斜坡而被调回。



a) 逆向体的生成



b) 建模约束在阶梯结构中避免逆向多边形——划分墙面多边形



c) 建模约束在阶梯结构中避免逆向多边形——用斜坡和阶梯细节状态详细阐述的结构

图 1-5　在建模层次中的凸体约束

然而，如此的建模约束是前后依赖的，稍有些不方便。

图1-6（彩页中也有）展示了一个没有逆向体的层次，它是通过加入如BSP树不会产生逆向节点的建模约束而设计出来的。

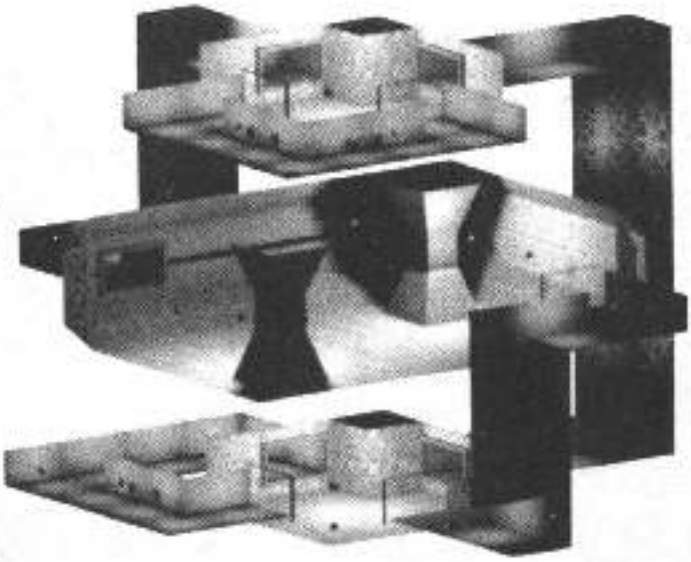


图1-6 为了产生无逆向节点的一个层次建模

数据

顶点数目：	1827
面数目：	440
需要包含面的独特平面数目：	76
包围盒大小：	2176.00 2304.00 1664.00
被选择单元的大小：	1532.00

BSP统计

BSP区域：	7
叶面数目：	637
树叶数目：	131（正向侧面131个，逆向侧面0个）
节点数目：	375

逆向节点既可以默许存在，也可以删除。如果我们打算建立一个封闭的层次，逆向节点就代表玩家永远不能到达空间体。

1.2.3 处理复杂的地形

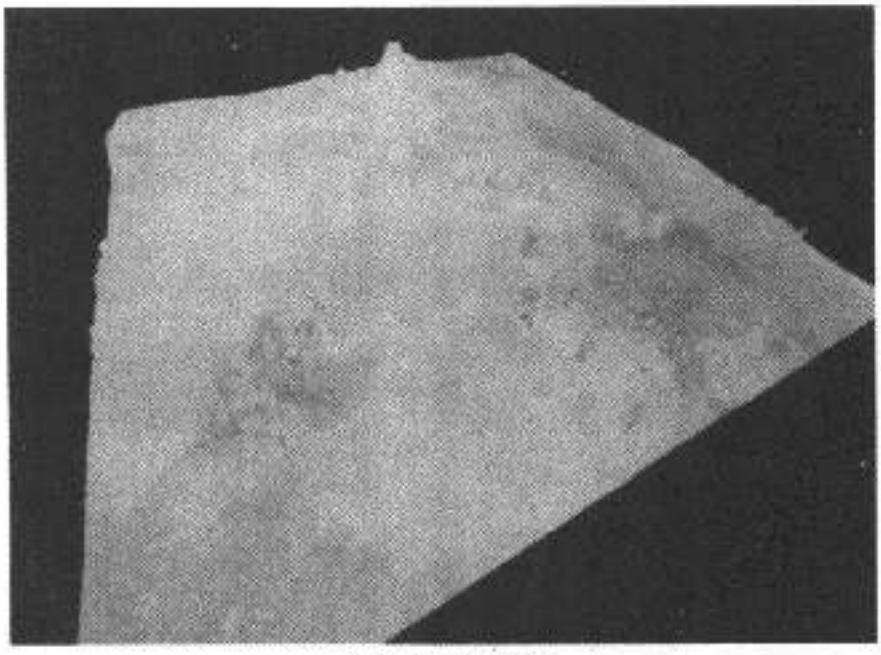
我们刚刚论述过的策略已经足够有效地处理复杂的地形数据。图1-7展示了从拥有大量（对于当前游戏实践）多边形数（133000个多边形）的复杂地形上空飞过的情况。这是通过NASA数据重现出来的火星的一部分表面，同时也展示了一个单一的划分单元的内容。下面是这个例子的统计：

数据

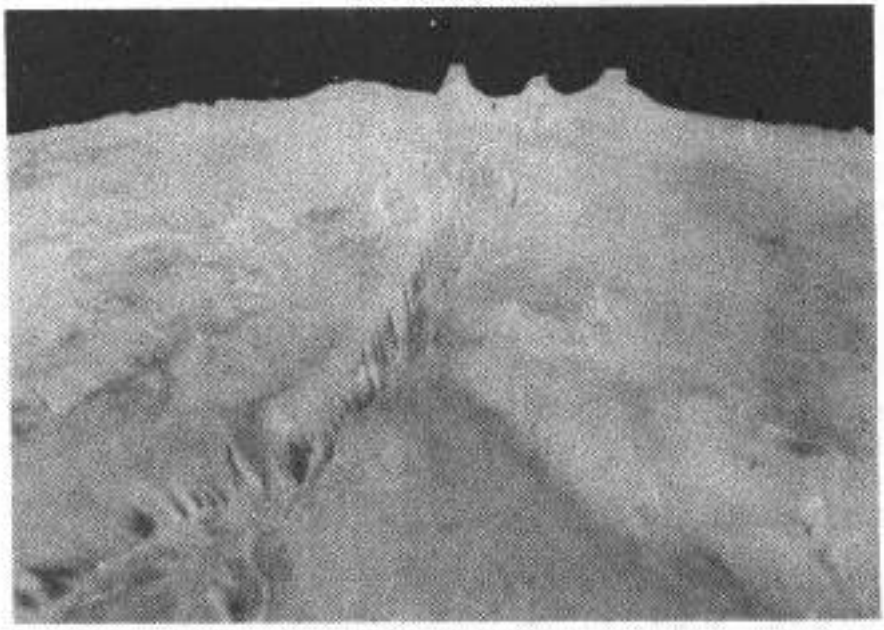
顶点数目：　400866

面数目：　133622

需要包含曲的独特平面数目：　108325

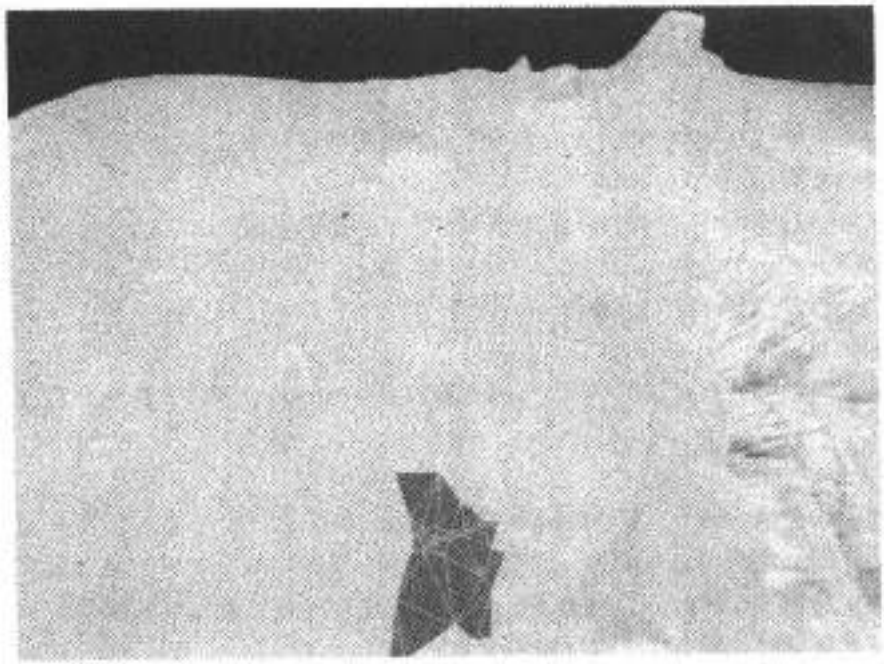


a) 火星表面的复杂地形



b) 显示点细节层次的近景

图　1-7



c) 单一 BSP 区域的内容

图 1-7　（续）

包围盒大小：	400000.00 400000.00 40235.29
被选择单元的大小：	20000.00
BSP 统计	
BSP 区域：	1014（等于在地形情况中的树叶数目）
叶面数目：	149188
树叶数目：	1014（正向侧面 509 个，逆向侧面 505 个）
节点数目：	1013

我们注意到，在这种例子中 BSP 区域和数目与树叶的数目是相等的。地形中不包含可以用作结构化元素的自然结构。因此我们选择使用一个大小远小于场景尺寸的单元。结果使用了 10^3 个区域。

另一个对地形必要的详细阐述是要考虑一个单一的 BSP 区域或者单元所代表的对象（见图 1-7c）。我们如前一样继续进行下去并使用规范 BSP 区域策略，但这是极其浪费的。我们必须有效地把独立的地形三角形当作要分割的结构化面。根据定义，结构化面不能共有顶点并且会以存储 3 倍于面数目的顶点而告终，这在火星的例子中是一个很大的数字。由以上的统计可以看出，当有 133×10^3 个面时就会有 400×10^3 个顶点。为了解决这个问题，我们把每个区域中的三角形当作一个三角形网格（第三类面）。这意味着将所有的三角形置于一个区域中去共享它们的顶点。这样彻底地削减了在区域中的顶点数目。例如在一个固定的网格中（忽略其边界），每个顶点被六个面所共有。然而，在相同区域中有不同纹理贴图的地形的情况下，顶点取得不同的纹理贴图不会导致崩溃。在这种情况下我们必须在区域中创

建与纹理贴图数目一样多的面。每个面包含所有属于这个区域的三角形并且拥有相同的材质。

以下是对这种详细阐述的地形范例的统计：

BSP 统计

BSP 区域：	1014（等于在地形情况中的树叶数目）
叶面数目：	1014
树叶数目：	1014（正向侧面 509 个，逆向侧面 505 个）
节点数目：	1013
顶点数目：	92546
面数目：	1014

这使得面与树叶的数目比为 1:1，同时也使顶点的数目减少了很大的一个系数。

1.2.4 BSP 叶节点中的面

最后我们来考虑如何处理包含在 BSP 树中的面。每个 BSP 叶节点包括了一列与之相联系的面。由于划分平面与之相交，一个面可以给出一个以上的 BSP 叶节点。当在绘制、照亮或者碰撞中使用 BSP 树时，我们必须标记已经处理过（被绘制过、相交等）的面，这样当检测另一片树叶时，就不再要重复相同面的计算。

1.2.5 寻找叶凸体

现在讨论凸体的诱因。先前论述的 BSP 策略的一个重要因素是封闭凸体的创建。以降序访问树中从根到任何叶节点的每一个节点，我们得到了一系列的平面，而这些平面自身形成了一个凸体。这个体既由如层次背景墙这样的结构化元素构成，又由那些与空的空间相交的分割平面形成。不管怎样，通过建模约束，我们保证这样的体是存在的，并且是凸的、封闭的。我们将这些体用在 PVS 计算的重要过程中，同样也用作路径规划的基础。

对于 PVS 计算和路径规划，我们必须明确地求与每个叶节点相联系的凸体。也就是说，必须求形成凸体的多边形。这个过程是明确的，并分四步进行：

1）求形成凸体的平面。

2）求与这些平面相交的顶点。

3）求对每个面有贡献的顶点。

4）对这些顶点排序来定义多边形。

第一步是为了求那些含有凸体的面。通过对树从根到树叶的横跨访问，同时收集每个节点平面，这些很容易办到。下一步是为了在凸体中求与这些平面相交的顶点。为了完成这一步，我们尝试集合中三个平面每一个可能的组合。依次检查这些组合是为了求那些相交于一点的平面。这些相交的点是潜在的顶点。然后测试这些潜在的顶点是否属于凸体。图 1-8a 展示了这一步是如何完成的：一个潜在的顶点分别由集合中其他所有平面进行测试，如果它满足到集合中其他所有平面的距离都为正这一条件，那么，其被证实是顶点。

接下来的代码可以求形成平面的凸体顶点。对于每个树叶有一个平面的矩阵，三个被选择的平面中每一个平面的法线被保存在一个 3×3 矩阵中。每个平面由其法线 $\mathbf{N}_x$、$\mathbf{N}_y$、$\mathbf{N}_z$ 及其位移 **D** 定义。为求交点我们需要解以下用矩阵方式表示的方程：

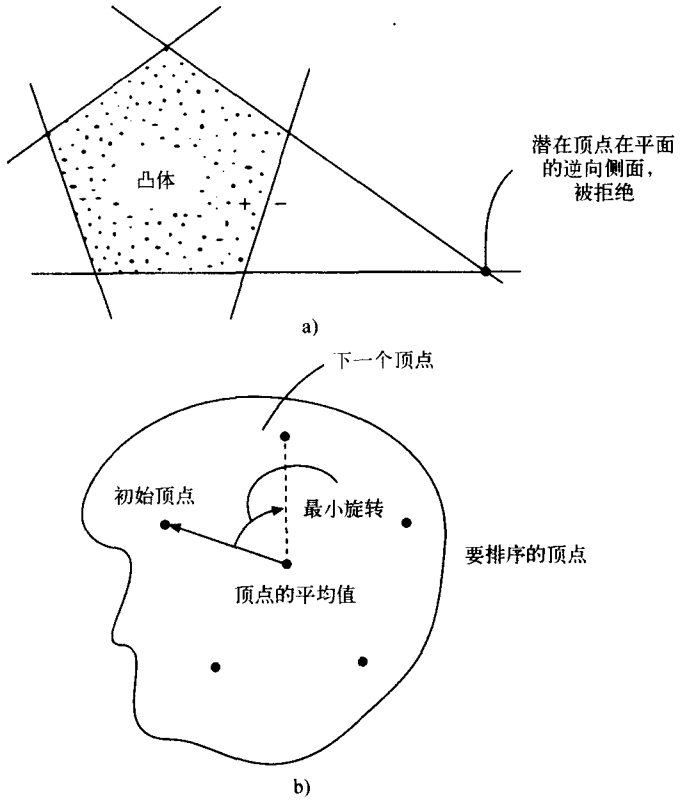


图 1-8　求形成叶凸体的多边形

$$\begin{bmatrix} N1_x & N1_y & N1_z \\ N2_x & N2_y & N2_z \\ N3_x & N3_y & N3_z \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} D1 \\ D2 \\ D3 \end{bmatrix}$$

```
void flyEngineBuild::build_leaf_vertices()
{
   int p[3],n,i;
   flyVector mat[3],m[3],v;
   float f1,f2;

   #define MAT_DET(mat)
(mat[0][0]*mat[1][1]*mat[2][2]+mat[0][1]*mat[1][2]*mat[2][0]+mat[0]
[2]*mat[1][0]*mat[2] [1]-mat[0][0]*mat[1][2]*mat[2][1]-
mat[0][1]*mat[1][0]*mat[2][2]-mat[0][2]*mat[1][1]*mat[2][0])

   for( n=0;n<nleaf;n++ )
   {
      for( p[0]=0;p[0]<leafplanes[n].num-2;p[0]++ )
      for( p[1]=p[0]+1;p[1]<leafplanes[n].num-1;p[1]++ )
      for( p[2]=p[1]+1;p[2]<leafplanes[n].num;p[2]++ )
      {
         mat[0]=leafplanes[n][p[0]].normal;
         mat[1]=leafplanes[n][p[1]].normal;
         mat[2]=leafplanes[n][p[2]].normal;
         f1=MAT_DET(mat);
```

```
        if(f1>=-0.01f && f1<=0.01f)
          continue;
        for( i=0;i<3;i++ )
        {
          m[0]=mat[0]; m[1]=mat[1]; m[2]=mat[2];
          m[0][i]=leafplanes[n][p[0]].d0;
          m[1][i]=leafplanes[n][p[1]].d0;
          m[2][i]=leafplanes[n][p[2]].d0;
          f2=MAT_DET(m);
          v[i]=f2/f1;
        }
        for( i=0;i<leafverts[n].num;i++ )
          if ((v-leafverts[n][i]).length2()<0.01f)
            break;
        if (i<leafverts[n].num)
          continue;
        for( i=0;i<leafplanes[n].num;i++ )
          if (leafplanes[n][i].distance(v)<-0.01f)
            break;
        if (i<leafplanes[n].num)
          continue;
        leafverts[n].add(v);
      }
    }
  }
```

在建立顶点列表之后，现在我们要求由这些顶点形成的多边形。换句话说，就是要求每个平面所拥有的顶点。最后找到每个顶点集的排序以此来定义每个凸体的多边形。这个最终的过程如图 1-8b 所示。为了求每个排序，我们要：

1）计算顶点的平均值。由于每一个多边形都是凸的，所以顶点的平均值一定是一个包含在多边形中的点。

2）定义一个从平均点到任意顶点之间的矢量。序列中的下一个顶点就是旋转（rotation）最小的那个矢量的顶点。

下列代码实现了一个凸多边形的无序顶点集合的排序过程。它用了顺时针点乘的概念产生从 -3 到 1 的返回值（见图 1-9）。

```
float clockwise_dot(const flyVector &v1,const flyVector &v2,const
flyVector &normal)
{
  float dot=FLY_VECDOT(v1,v2),f;
  flyVector v;
  v.cross(v1,v2);
  f=FLY_VECDOT(v,normal);

  if(FLY_FPSIGNBIT(f))
    dot=-dot-2;
  return dot;
}

void flyPolygon::order_verts()
{
  int i,j,next;
  float maxdot,f;
  flyVector centre,v0,v1;

  // find polygon center
  centre.null();
  for(i=0;i<verts.num;i++)
```

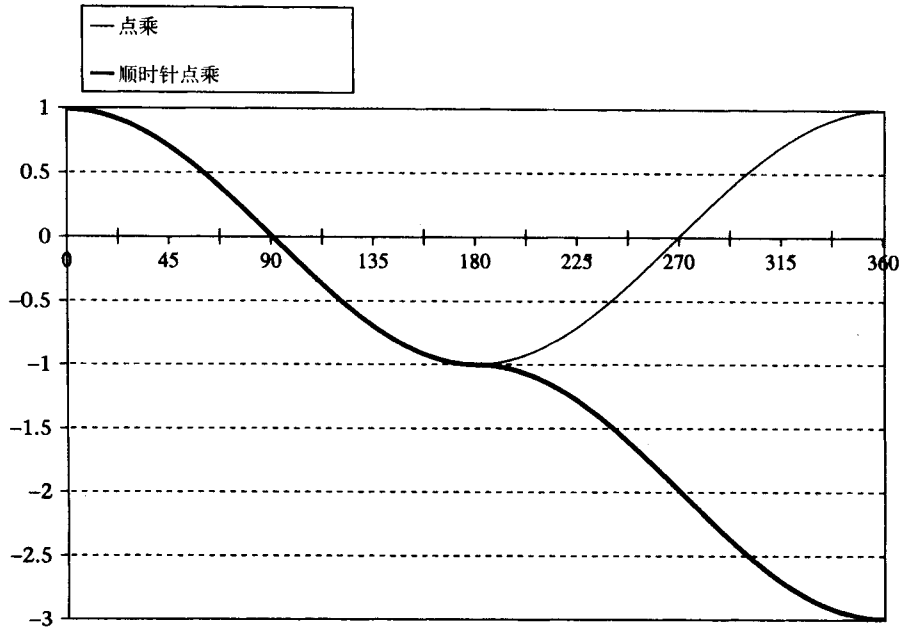


图 1-9　顺时针点乘

```
    centre+=verts[i];
  centre*=1.0f/verts.num;
  // find vector for first vertex
  v0=verts[0]-centre;
  v0.normalize();

  // for every other vertex
  for(i=0;i<verts.num-2;i++)
  {
    // find vertex with biggest clockwise dotproduct
    maxdot=-4.0f;
    next=-1;
    for(j=i+1;j<verts.num;j++)
    {
      v1=verts[j]-centre;
      v1.normalize();
      if((f=clockwise_dot(v0,v1,normal))>maxdot)
      {
        maxdot=f;
        next=j;
      }
    }
    // swap vertex to correct order
    flyVertex aux=verts[i+1];
    verts[i+1]=verts[next];
    verts[next]=aux;
  }
}
```

1.2.6　凸体和伪入口

为了提供一个像有向图那样可以被路径计划器（path planner）使用的结构，在构造过程

中 BSP 树的凸体可以更进一步被拓展。这样的一个 AI 方法可以通过一个定位层次的独立代理程序取得。其基本的思想是寻找一个与邻近体相连接的“伪入口”（pseudo-portal）。伪入口代表了两个凸体之间的一条清楚的路径。图 1-10 展示了被分为三个凸体的一个非常简单的层次。把区域 A 从区域 B 和 C 划分出来的平面既包含了背景墙，也包含了伪入口。图1-11（也见彩页）展示了游戏层次中伪入口的例子。

因此，整个问题被定义为：寻找可以形成伪入口平面的子区域。

我们可以按如下的方法进行：

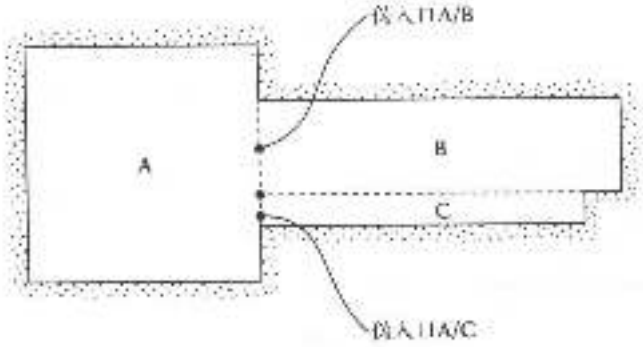
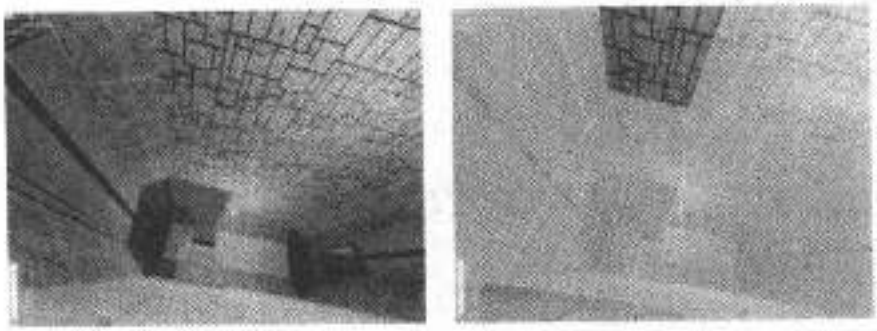
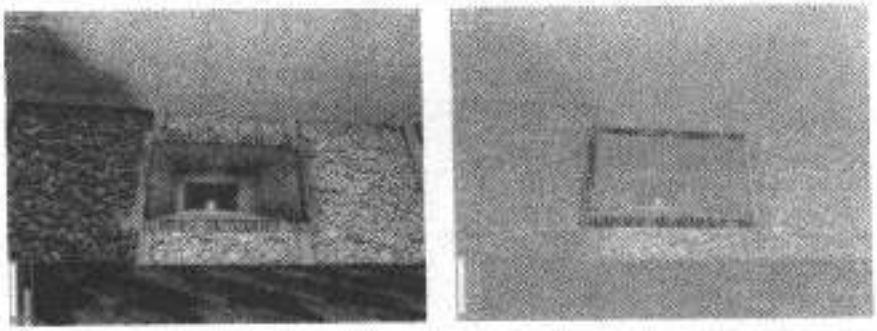


图 1-10　凸体和伪入口



a) 游戏某一层次中一个空旷场景的伪入口，明确的连接面被显示出来



b) 在这个例子中，伪入口下面和真正的入口相一致

图　1-11

```
for 每个凸体
    for 每个凸体中的面
        if 这个面与另一个凸体中的某个面都在同一个平面上
          在这两个面之间检测 2D 交点
            if 某个交点存在 then 定义一个伪入口
                加入有向图
                继续下一个面
        这个面一定是一个背景墙且可以被消除
```

检测一个包含在另一个平面中的面是简单直接的。我们需要考虑平面的偏移和法线，并且必须记住两个拥有相反方向法线的平面可以重合。

```
int test_coplanar(flyPolygon &p1,flyPolygon &p2)
{
   float dot=FLY_VECDOT(p1.normal,p2.normal);
   if(dot>0.999f && (float)fabs(p1.d0+p2.d0)<0.01f)
      return TRUE;
   return FALSE;
}
```

求两个凸多边形相交产生的多边形更加困难。若这两个多边形在同一个平面上，我们可以在这个平面上进行设计，这样就可以把问题削减成一个二维的问题。我们需要最大区域投影，这可以通过求三个有较大绝对值的法向分量（x，y，z）并撤销这个分量的方法来获得。

为了在 2D 中求出交集，我们必须求：

1）第一个多边形在第二个多边形中的所有顶点。

2）第二个多边形在第一个多边形中的所有顶点。

3）所有的边交集。

这就给出相交多边形顶点的一个无序的点集合。前一节中介绍的点排序方法在此可以用来求该多边形。为了使这个算法得到充分加强，必须考虑某些难点：

1）完全重合的多边形必须产生一个有效的多边形。

2）由相交边连接的多边形不应生成一个退化的多边形，即该算法必须返回零交集。

为解决这些问题，必须在开始的容量测试中结合阈值。第一个问题如下所示，当测试某点是否在多边形中时，该点与多边形所有边的距离都要考虑。我们只接受其距离通过阈值测试的点。同样地，当向输出多边形列表添加顶点时，必须检查顶点是否是列表中的一部分。第二个问题事实上与第一个相反，我们使用阈值测试。

边相交的问题与 2.4.3 节的碰撞检测边相交相似，只不过这里的两条边是静态的。在这种情况下，我们必须在 2D 下交叉两条线段（两条多边形的边）。以下的两个测试可以完成：

1）如果一条边中每个顶点到另一条边的距离为正，那么这两条边是不可能相交的。这意味着，这两个顶点在含有另一条边的（不确定）直线的同一个侧面上。

2）在另一条边上的相同测试。

3）如果以上的两个测试都成功了，那么存在一个交集。

图 1-12 展示了三种情况。（p_1，p_2）和（p_3，p_4）是两条边，每条都是相交多边形的边。我们可以按如下步骤进行：

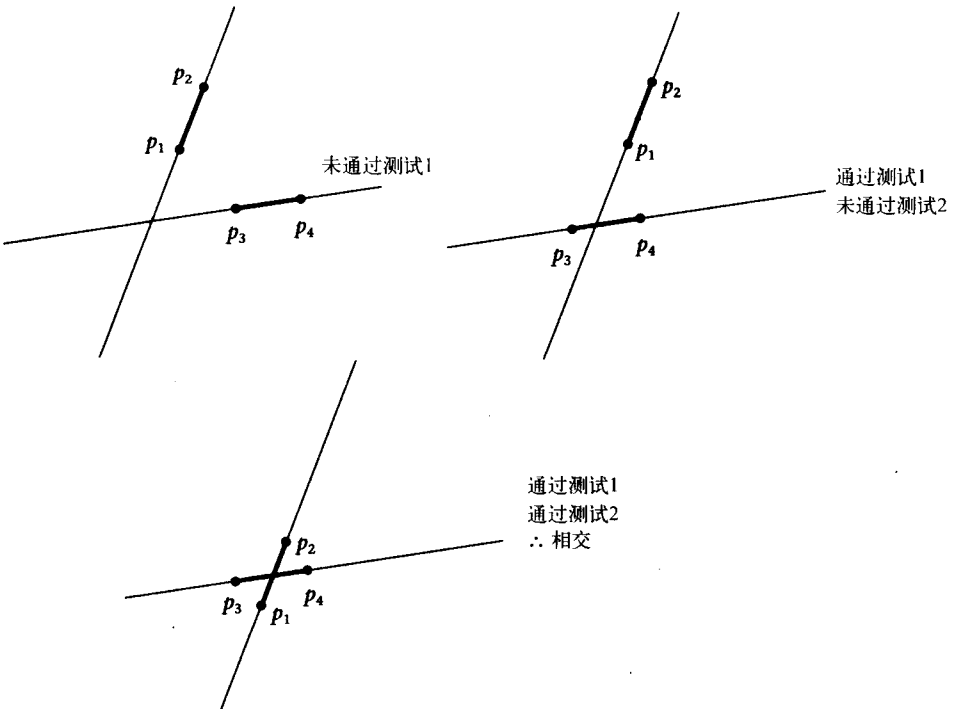


图 1-12　边相交

从顶点到边的距离

$$d_1 = p_2 - p_1;$$
$$d_2 = p_3 - p_4;$$

确定包含边的直线

直线 1：$p_1 + dist_1 * d_1$;

直线 2：$p_3 + dist_2 * d_2$;

***dist*$_1$ 的解**

$$dist_1 = (d_2.x*(p_1.y - p_3.y) - d_2.y*(p_1.x - p_3.x))/(d_2.y*d_1.x - d_2.x*d_1.y);$$

求交点

交点：$p_1 + dist_1 * d_1$

下面的代码整合以上这些方法，由以下几步组成：

1）两个多边形的相交测试。

2）对顶点的容量测试。

3）边相交测试。

```
int flyPolygon::intersect(flyPolygon &in, flyPolygon &out)
{
  // clear output polygon
  out.verts.clear();

  // find vertices from polygon in that are inside current polygon
  build_edgeplanes();
  intersect_verts(in,out);
```

```
  // find vertices from current polygon that are inside polygon in
  in.build_edgeplanes();
  in.intersect_verts(*this,out);

  // find all intersections from edges of the current polygon
  // with edges from the in polygon
  intersect_edges(in,out);

  // return number of vertices in output polygon
  // >2 means a valid polygon
  return out.verts.num;
}
int flyPolygon::intersect_verts(const flyPolygon &in, flyPolygon &out)
{
  int i,j,k;

  // for all vertices from in polygon
  for(i=0;i<in.verts.num;i++)
  {
    // test if distance to current polygon edges is smaller then
threshold
    // (edge plane normal points inside polygon)
    for(j=0;j<edgeplanes.num;j++)
      if(edgeplanes[j].distance(in.verts[i])<-0.1f)
        break;
    // if passed all above tests, vertex is inside current polygon
    if(j==edgeplanes.num)
    {
      // test if vertex is not already in output list
      for(k=0;k<out.verts.num;k++)
        if((out.verts[k]-in.verts[i]).length2()<0.1f)
          break;
      // if not in output list, add it
      if(k==out.verts.num)
        out.verts.add(in.verts[i]);
    }
  }

  return out.verts.num;
}

int flyPolygon::intersect_edges(const flyPolygon &in, flyPolygon &out)
{
  int i,j,k,a,b;
  float f1,f2;
  flyVector p1,p2,d1,d2;

  // for every edge from current polygon
  for(i=0;i<verts.num;i++)
    // for every edge from in polygon
    for(j=0;j<in.verts.num;j++)
    {
      // test1: check if in edge vertices are on same plane
      // of current polygon edge
      f1=edgeplanes[i].distance(in.verts[j]);
      f2=edgeplanes[i].distance(in.verts[(j+1)%in.verts.num]);
      if((f1*f2)>-0.1f)
        continue;

      // test2: check if current edge vertices are on
      // same plane of in polygon edge
      f1=in.edgeplanes[j].distance(verts[i]);
```

```
        f2=in.edgeplanes[j].distance(verts[(i+1)%verts.num]);
        if((f1*f2)>-0.1f)
          continue;

        // passed test 1 and test2: intersection exists
        // compute edge vector d1
        p1=verts[i];
        d1=verts[(i+1)%verts.num]-p1;
        d1.normalize();

        // compute edge vector d2
        p2=in.verts[j];
        d2=in.verts[(j+1)%in.verts.num]-p2;
        d2.normalize();

        // project onto maximum area plane
        if (fabs(normal.x)>fabs(normal.y)) a=0; else a=1;
        if (fabs(normal[a])<fabs(normal.z)) a=2;
        if (a==0) { a=1; b=2; } else
        if (a==1) { a=0; b=2; } else { a=0; b=1; }

        // compute distance of intersection
        float dist= (d2[a]*(p1[b]-p2[b])-d2[b]*(p1[a]-p2[a]))/
              (d2[b]*d1[a]-d2[a]*d1[b]);

        // compute intersection point
        p1+=dist*d1;

        // test if vertex in not already in output list
        for(k=0;k<out.verts.num;k++)
          if((out.verts[k]-p1).length2()<0.1f)
            break;
        // if not in output list, add it
        if(k==out.verts.num)
          out.verts.add(p1);
      }

    return out.verts.num;
  }
```

下一章将给出使用这个结构进行路径计划的一个简单的实时方法。

1.2.7　潜在可视集

在建立了一个凸体后，我们很容易用这些去定义 PVS。这是一个以树叶数目为大小的正方形连通矩阵，其中二元元素（i，j）= 1 表示叶 i 与叶 j 相连接。这意味着如果我们处于叶 i 时，叶 j 是潜在可视的。BSP 树自身仅仅使树叶的可见度排序变得更容易。这个集合包含了场景中所有的多边形。当一棵 BSP 树用于绘制时，我们遍历整棵树去求被视见约束体裁剪的树叶面，然后用 PVS 连通矩阵去消除所有其他不是潜在可视的树叶。

PVS 评估最直接的方法就是使用一些基于样本点的方法。不管使用怎样的算法，PVS 需要考虑体中的样本点，并查看任何样本点在其他的体中是否是可视的。尽管效率在构造过程中并不必要，但是如果不优化，那么 PVS 运算会花费极长的时间。我们考虑 m 个体、每个体含有 n 个样本点的情况，最坏的运算情况是 $O((mn)^2)$。因此，每个凸体的样本集合的结构是一个重要的设计因素。

我们可以使用凸体的伪入口去建立 PVS。为了做到这一步，我们设置样本去跨越测试伪

入口的面，并使用光线相交去确定在任意伪入口上任意体的任意一点是否都是可见的。图 1-13展示了四个凸体 A、B、C、D。图中 B 与 A 直接相邻，因此 B 对 A 而言是可视的。为了测试从 A 到 C 的可视性，我们创建了从体 A 伪入口的样本点开始，到体 B、C 之间伪入口的样本点为止的光线。在这个例子中，光线从伪入口 A/B 到达伪入口 B/C。因此，C 对 A 是潜在可视的。现在测试 A/D 的可视性。任何从 A 到 D 的光线会被拦截，所以 D 对 A 是不可视的。将这个方法嵌入递归的结构中，这样对于体 A 而言，将找到所有的潜在体。

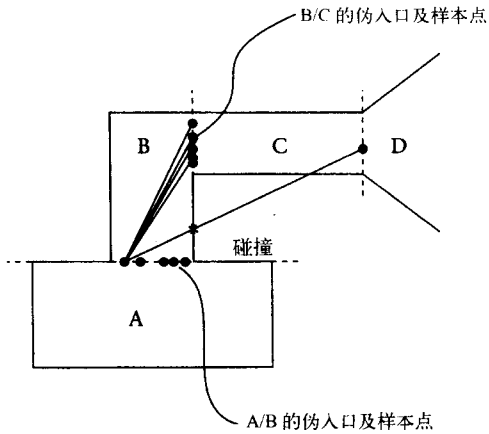


图 1-13　三个凸体 A、B、C 及其入口

我们用一组随机的样本点作为伪入口的样本。对于相同数目的样本，一组良好分布的随机点相比一组均匀间隔的点而言，会得到一个比完整确定的解法更接近的结果。由多边形而生成的点的数量需成为一个多边形区域的函数。通过计算在多边形包围盒里的一个随机点并将其规划入多边形平面中，在多边形中生成随机的点集。然后我们需要测试这些点是否包含在多边形中。如果不满足，则产生一个新的样本。这个检测需要进一步详细阐述，以避免生成的点与多边形的边重合——此时碰撞测试可能会发生精度错误。

为了避免出错，递归结构需要仔细地构建。每个叶节点的可见度都必须计算出来，并且将相同的算法单独应用到每一个叶节点中。

主要的伪代码：

```
for 每个叶节点
    创建当前节点的直接邻节点列表
    调用递归遍历当前节点和相邻列表中的节点
```

递归的伪代码：

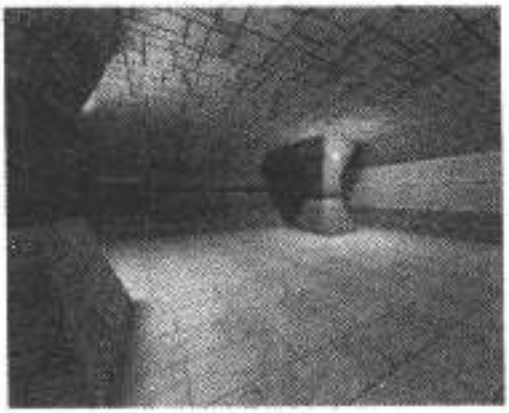
```
    将遍历过列表中所有叶节点的全部邻节点加入创建的新的列表中
for 每个新列表中的节点
    测试它对最初叶节点的可视性
    如果是不可视的，将它从新的列表中删除
如果新的列表是非空的，递归回原始的树叶和新的邻节点列表
```

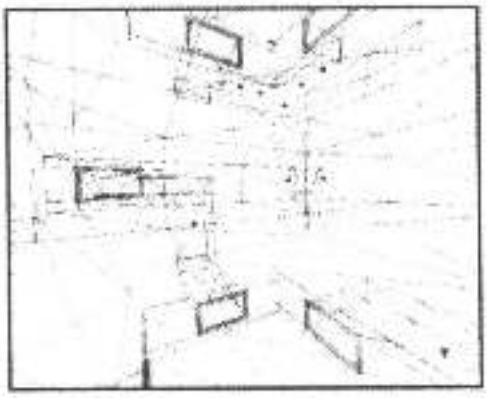
```
void flyEngineBuild::compute_visibility()
{
   int i,j,k;
   // set all pvs bits to not visible (0)
   memset(pvs,0,pvssize);

   // create flag array and neighbours list
   flags=new char[nleaf];
   flyArray<flyBspNodeBuild *> list;

   // for every leaf node
   for( i=0;i<nleaf;i++ )
```

```
    {
      // set node self visibility to 1 (pvs matrix diagonal)
      pvs[i*pvsrowsize+(i>>3)] =1<<(i&7);

      // clear flag array and set current leaf node
      memset(flags,0,nleaf);
      flags[i]=1;

      // for every neighbour, set it as visible and add it to the list
      for( j=0;j<leaf[i]->neighbors.num;j++ )
      {
        k=leaf[i]->neighbors[j]->leaf;
        pvs[i*pvsrowsize+(k>>3)] =1<<(k&7);
        flags[k]=1;
        list.add((flyBspNodeBuild *)leaf[i]->neighbors[j]);
      }
      // recurse computing visibility
      compute_visibility(i,list);
    }

    // free flags
    delete[] flags;
}

void flyEngineBuild::compute_visibility(int
leafnum,flyArray<flyBspNodeBuild *> &list)
{
    int i,j,k,n,l;
    flyArray<flyBspNodeBuild *> list2;

    // create new list with all neighbours from all nodes
    // in received list that have not been already used
    for( n=0;n<list.num;n++ )
    for( i=0;i<list[n]->neighbors.num;i++ )
    {
      j=list[n]->neighbors[i]->leaf;
      if (flags[j]==0)
      {
        list2.add((flyBspNodeBuild *)leaf[j]);
        flags[j]=1;
      }
    }

    // for every node in the new list
    for( i=0;i<list2.num;i++ )
    {
      // for every node in original list
      for( j=0;j<list.num;j++ )
      {
        // test for a connecting portal between the two nodes
        for( k=0;k<list[j]->neighbors.num;k++ )
          if (list[j]->neighbors[k]==list2[i])
            break;
        if (k<list[j]->neighbors.num)
        {
          // if a portal is found, test its visibility against
          // all portals from original node
          l=list[j]->leaf;
          for( n=0;n<leaf[leafnum]->neighbors.num;n++ )
            if (test_visibility(leafnum,n,l,k))
```

```
            break;
         if (n<leaf[leafnum]->neighbors.num)
            break;
      }

   }
   if (j==list.num)
   {
      // if no portals leading to new list node are visible
      // remove it from the new list as it is not used
      flags[list2[i]->leaf]=0;
      list2.remove(i--);
   }
   else
   {
      // if any portal leading to the new list node
      // set it as visible in the pvs matrix
      j=list2[i]->leaf;
      pvs[leafnum*pvsrowsize+(j>>3)]|=1<<(j&7);
   }
 }

 // if new list is not empty, recurse using the new list
 if (list2.num)
   compute_visibility(leafnum,list2);
}

int flyEngineBuild::test_visibility(int leafnum,int portalnum,int
testleaf,int testportal)
{
 // return true if portal (leafnum,portalnum) can see
 // portal (testleaf,testportal)

 // compute portal areas
 float a1=leaf[leafnum]->portals[portalnum].area();
 float a2=leaf[testleaf]->portals[testportal].area();

 // compute number of random sample points in each portal as a
function of area
 int i1,i2,j1,j2;
 j1=(int)(a1/pvsgridsize)+1;
 j2=(int)(a2/pvsgridsize)+1;

 // ray intersect all portal sample points
 flyVector v1,v2;
 for( i1=0;i1<j1;i1++ )
   for( i2=0;i2<j2;i2++ )
   {
      leaf[leafnum]->portals[portalnum].random_point(v1);
      leaf[testleaf]->portals[testportal].random_point(v2);

      // if any ray does not have a collision
      if (0==collision_test(v1,v2,FLY_TYPE_STATICMESH))
         // portals are visible to each other, return true
         return 1;
   }

 // portals are not visible, all rays collided, return false
 return 0;
}
```

图 1-14 所示为一个层次的已渲染视图以及两个线框图（包含和不包含 PVS）。





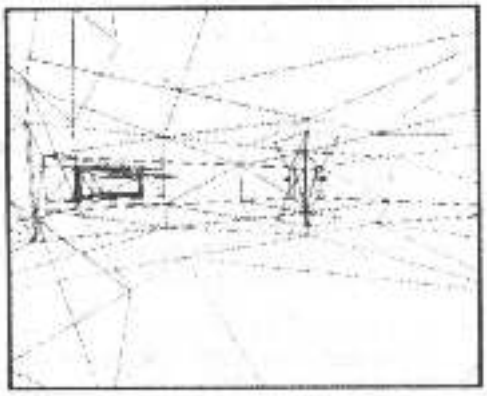


图 1-14　同一房间的二幅视图——已渲染，没有 PVS 的线框，有 PVS 的线框

1.3　光照贴图的构造

光照贴图是一个较为常用的 3D 游戏技术，它的作用是存储预计算过的光照。将光照情况存储在光照贴图中，意味着我们可以使用处理纹理映射的硬件来实现静态照明。光照贴图的构造过程分成三个截然不同的阶段：

1）生成光照贴图的坐标。

2）将许多光照贴图打包成较大的光照贴图。

3）应用反射模型，照亮光照贴图。

1.3.1　生成光照贴图的坐标

我们按照下列方法生成光照贴图坐标。对于每个面片我们考虑包含它的那个平面，在其中找到这个面片的包围盒。在这里与面片重合的那个平面是一个好的选择。考虑此面片从二维空间映射到二维空间，重合的面即意味着面片在光照贴图中的投影区域等于面片本身的区域。由此计算好的光照以最高的精确度被缓存。如图 1-15 所示，光照贴图的纹理坐标定义为：

$$t_{cx} = \frac{v_{1x} - p_{1x}}{p_{2x} - p_{1x}}$$

$$t_{cy} = \frac{v_{1y} - p_{1y}}{p_{2y} - p_{1y}}$$

这些仍需要进一步的改善，因为许多单独的光照贴图会被打包（在下一节中论述）成大图。我们需要将硬件纹理滤波技术应用于光照贴图，否则在最终绘制出的图像中光照贴图的像素边界将会可见。这意味着我们必须保证在一个光照贴图的像素正被过滤时，硬件不参与相邻光照贴图中像素的过滤过程。这一步可以通过对坐标稍微“缩水”一点点来实现，这样每个光照贴图的边界处都会有一个 1/2 像素大小的区域。因此：

$$t_c = \frac{t_c\ (s-1)}{s} + \frac{1}{2s}$$

这里 s 是光照贴图的维度（x 或 y）。

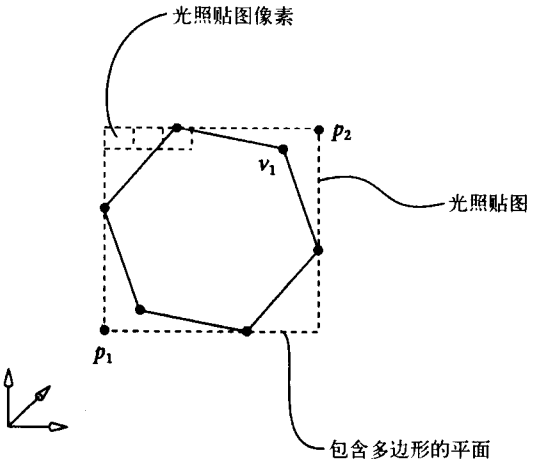


图 1-15　多边形及其包含的平面和光照贴图像素

1.3.2　光照贴图的打包

有效地把光照贴图打包成大的纹理贴图是一个很重要的优化过程。前面所介绍的方法会生成与面的数目一样多的光照贴图，而这种小纹理会导致绘制过程中大量地使用纹理交换过程（texture swapping）。把许多光照贴图打包转化为一个大纹理则可以把进行纹理交换的次数减到最低。

这就是经典的箱柜打包问题——将剩余的最大原料塞到最大的可用空间中去。图 1-16 展示了一个较好的算法。这个算法的过程如下所示：

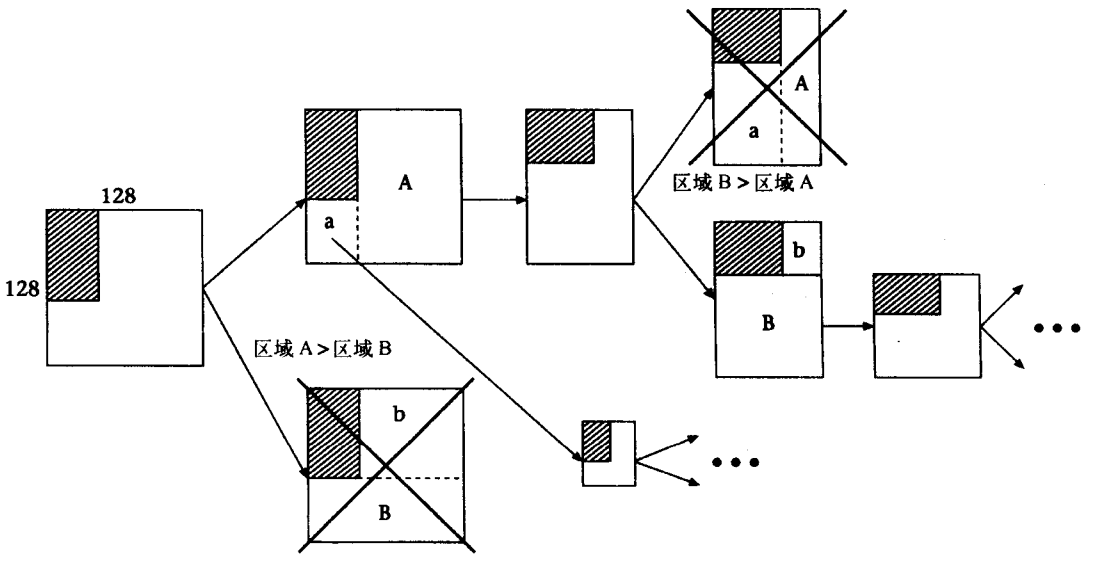


图 1-16　光照贴图打包算法的图示

1）按面积将光照贴图排序。

2）初始化一个工作区域（通常 128×128 像素）。

3）在排序过的表中进行循环，将能放入工作区域的所有可用光照贴图中最大的一个置入工作区域。如果对当前的工作区域来说没有光照贴图可以放入，则返回第 2 步。

4）被插入的图定义了当前工作区域中的两个剩余区域：

A 被定义为从图的右手边开始到区域结束的区域；

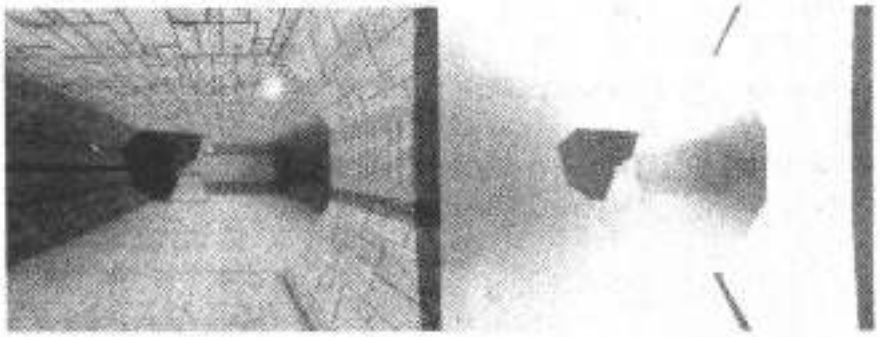
B 被定义为从图的底边开始到区域结束的区域。

5）调用第 3 步，在 A、B 的较大区域中递归。

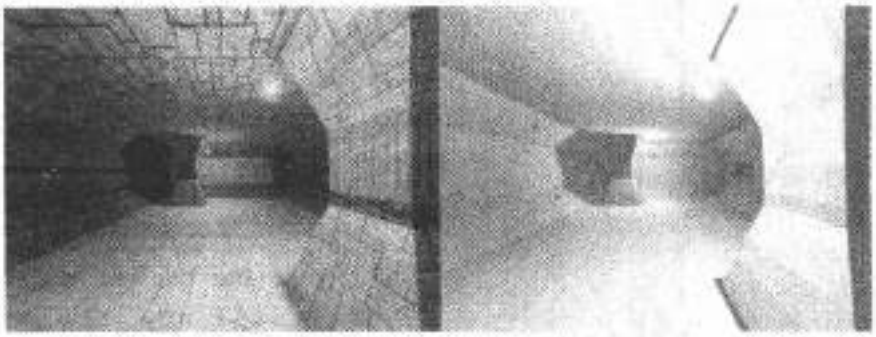
6）调用第 3 步，在先前递归中的较小区域中递归（如果前一次的递归在 A 中就选区域 a；如果前一次的递归在 B 中就选区域 b）。

1.3.3　对光照贴图的解释

光照贴图是指应用光照模型以缓存照明情况。光照贴图可以接收包括辐射度在内的任何视图独立的光照模型。这是一种缓存光照的方法，它独立于计算光照的方法。我们必须同时考虑面片是否在阴影中，并由此相应地调制光照。整个过程起始于递归调用 BSP 树含有每个光线所涉及的坐标，以此找到对于光线在其影响范围下的面。然后对每一个（面）光照贴图进行如下的处理：



a）简单光照贴图——平方衰减定律（纹理滤波关闭后使得光照贴图像素可见）



b）简单光照贴图——平方衰减定律与 L.N（纹理滤波关闭后使得光照贴图像素可见）

图 1-17

```
for 每个在光照贴图中的像素
    if 像素在多边形中
    找出像素的世界坐标
        从像素点开始沿光线方向应用光线相交碰撞检测
            if 没有相交则应用光照反射模型
```

阴影的出现是因为所有的光照贴图被初始化为环境光。应用光照模型提高在环境（阴影）层次的光照。对于柔和阴影来说，光心在一固定半径中被加以随机扰动，从而形成大量光照采样点。这其中的每一个采样点都被用于光线碰撞检测，并且通过碰撞的次数计算强度，以给出光照贴图像素的亮度值。

简单的光照模型如下定义：

1）基于距离的线性或平方衰减。

2）L.N 明暗效果与距离衰减一起出现。

在引擎中，每个亮度可以被设置成以上模型的任意组合。用不同的光照模型可以产生出不同的环境氛围。图 1-17 所示为同一层次在不同的光照贴图照明下的效果。

1.4　BSP 管理

在游戏应用程序中使用 BSP 管理是一个建构良好且很通用的方法。如我们所见，它在游戏的建立和执行过程中都可以使用。在执行过程中，为了观察和绘制，BSP 树至少被调用一次。其他的递归必须从光照贴图中给出照射所产生动态的光线引起。碰撞测试尽可能多地利用了 BSP，这是一个被每条交线所调用的递归（1.4.1 节）。正是其一般性的作用使得这个方法变得流行并且耐用。在这一节我们来探讨使得这个方法一般化的高级扩展。

一般的递归方法

我们现在回过头来考虑在 BSP 管理中用不同函数递归访问 BSP 树中最好的一个。我们把它叫做一般的递归。BSP 管理中另一个重要的问题是动态对象。大的动态对象可以拥有数个叶节点，从而导致了这个问题。例如，考虑图 1-18 中的情况。一个大的对象被一个划分的平面紧紧夹住，并且夹在其间其本源仅仅占据了这一单一的划分。一个碰撞光线完全照射到上方的区域中，相交的对象不会像使用单一相交方法那样返回碰撞。这个问题同样是由动态光线和绘图所产生的。处理所有脱离实体超过一个叶节点的问题的最好方法是用一般递归方法。

当递归一棵 BSP 树时，我们总会问这样的问题：哪些叶节点子集包含在对象空间中？例如，在碰撞检测的情况下，我们用光线递归求包含潜在碰撞的对象。图 1-19 展现了三种从 BSP 树中选择节点的方法。第一种是由点和半径生成的球。在递归时，所有与球相交的则为 BSP 叶节点；第二种——射线，由两点生成线段，与线段相交的则

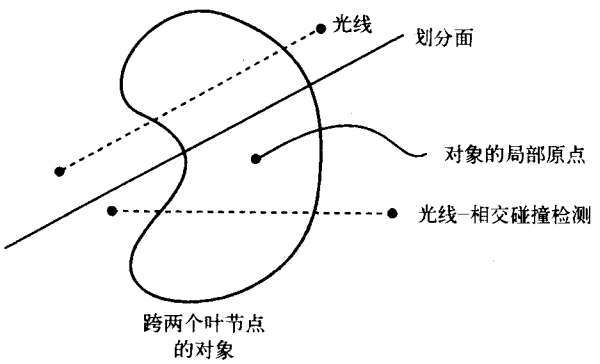


图 1-18　光线相交碰撞检测和大的对象

为叶节点；第三种是第二种的推广，由多个线段构成的体，与体相交的则为叶节点，视见约束体是最常见的表示。

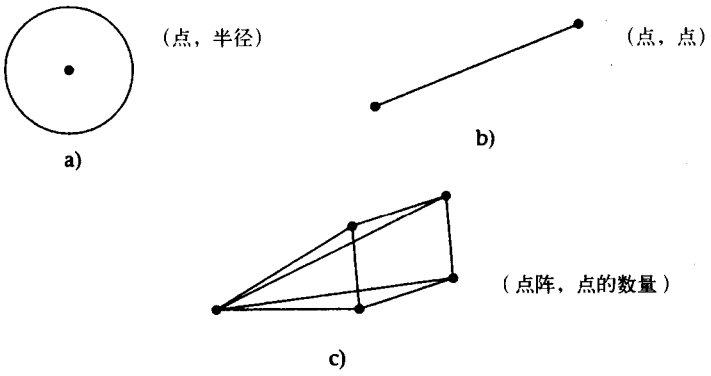


图 1-19　一般递归模式

一般递归是如下的一种方法，它用图 1-19 定义的其中一个选项作为选择的结构，从根节点开始遍历 BSP 树。对于每个由结构选出的节点，如果需要的话我们可以进行 PVS 连通性测试。这对于含有视见约束体或者一个作用球的渲染递归是十分有用的。图 1-20 展示了一个单一的层次计划。位于某个作用半径下（如图所示）的动态光线会找到代表两个房间的节点。通过 PVS，这个看不见光的房间可以从它的作用下被消除。

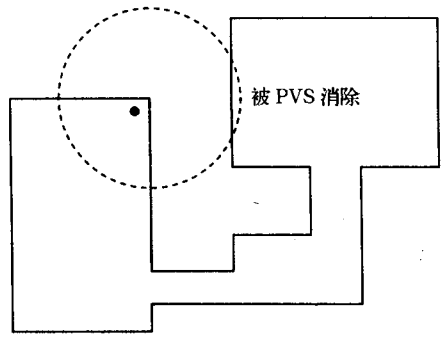


图 1-20　动态光线和一般递归。这个看不见光的房间被 PVS 消除

一般递归的有效实现

在考虑一般递归方法时，产生了两个观点。首先，这个方法可以在不同的模式中操作。图 1-21演示了六个操作模式中的四个，这六个操作模式产生于三个可以使用 PVS 的模式。这三个基本模式如下：

- 递归 BSP，仅选择裁剪特定对象（可以是球体、线段、一系列点）的叶节点。返回叶节点的列表。
- 递归 BSP，选择裁剪特定对象的叶节点中的所有对象。返回对象的列表。
- 递归 BSP，选择裁剪特定对象的叶节点中属于某一个类型的所有对象。返回同一类型对象的列表。

当一个 PVS 叶节点被这些模式中的任何一个遍历时，选定的叶节点在做更深一步的考虑前，会作 PVS 封闭测试。最有效执行这个方法的途径并不是通过使用正常的递归，而是使用单一指针的栈。它消除使用递归函数调用的额外成本。

下面的定义用于快速浮点数点检测（使用整数）。这些采用了一些继承于天然测试的策略。它们将浮点数的每一位当作整数来处理。例如当检测到零时，就像检测整数零一样，浮

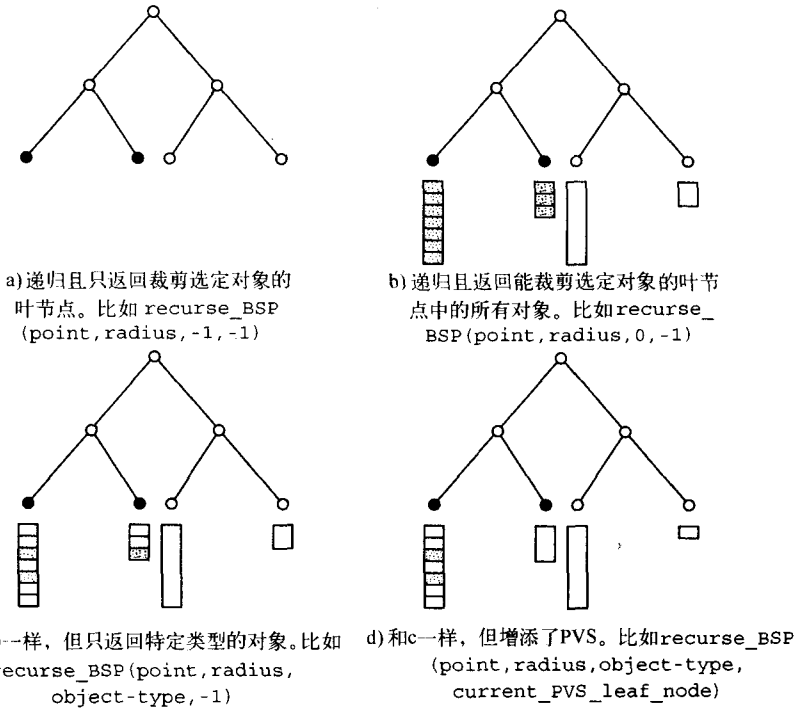


图 1-21　一般的 BSP 递归过程：四个例子

点数的每一位都是零。而检测浮点数的符号只需要一位就可以满足。

```
#define FLY_FPBITS(fp)      (*(int *)&(fp))
#define FLY_FPABSBITS(fp)   (FP_BITS(fp)&0x7FFFFFFF)
#define FLY_FPSIGNBIT(fp)   (FP_BITS(fp)&0x80000000)
#define FLY_FPONEBITS       0x3F800000
```

下面的定义用来检测 PVS 矩阵的每一位。它决定了从一个节点到另一个节点是否是可视的。参数 *from* 是叶节点序列中最原始的节点，参数 *to* 是节点序列中经过检测从 *from* 叶节点可视的叶节点。

```
#define FLY_PVS_TEST(from,to) \
   (*(pvs + (from)*pvsrowsize + ((to)>>3)) & (1 << ((to) & 7)))
```

我们现在考虑一般递归的方法。对作用球递归方法的伪代码执行如下：

```
recurse_bsp( point, radius, objtype, pvsnode )
{
   clear selection list

   push(bsp root node)

   while (stack is not empty)
   {
      node = pop_node()
      if (node is leaf)
      {
         if (pvsnode no equal to -1 or PVS_TEST(pvsnode,node))
         {
            add node to slection list

            if (objtype not equal to -1)
```

```
            if (objtype equal to 0)
              add all objects in node to selection list
            else
              add all objects in node with type objtype
              to selection list
        }
      }
      else
      {
        dist = distance from point to node plane

        if (abs(dist)<radius)
        {
          push(node child 0)
          push(node child 1)
        }
        else
        {
          if (dist>0)
            push(node child 0)
          else
            push(node child 1)
        }
      }
    }
  }
```

以下是使用球体的 BSP 树递归的全部代码。球体由点 *p* 和半径 *rad* 定义。如果 *elemtype* 设置为 -1，递归仅选择那些被球体裁剪的叶节点。如果 *elemtype* 设置为 0，递归也选择包含在指定叶节点中的所有对象（注意不在球体内的对象有可能被返回，因为它们中有被球体裁剪的叶节点）。如果 *elemtype* 设置为 >0，只有此类型 *id* 的对象会被返回。

```
void flyEngine::recurse_bsp(flyVector& p,float rad,int elemtype,int
pvsleaf)
{
  static flyBspNode *stack[64];
  flyBspNode *n;
  float d;
  int nstack=1;
  stack[0]=bsp;

  cur_bsprecurse++;
  nselnodes=0;
  nselobjs=0;
  while(nstack)
  {
    n=stack[--nstack];

    if (n->leaf!=-1)
    {
      if (pvsleaf==-1 || PVS_TEST(pvsleaf,n->leaf))
      {
        selnodes[nselnodes++]=n;
        if (elemtype!=-1)
        {
        flyBspObject **elem=&n->elem[0];
        for( int e=0;e<n->nelem;e++,elem++ )
          if ((elemtype==0 || (*elem)->type==elemtype) &&
            (*elem)->lastbsprecurse!=cur_bsprecurse)
            {
```

```
                (*elem)->lastbsprecurse=cur_bsprecurse;
                selobjs[nselobjs++]=(*elem);
              }
          }
        }
      }
      else
      {
        d=n->distance(p);

        if (fabs(d)<rad)
          {
          if (FP_SIGN_BIT(d)==0)
            {
            if (n->child[1])
              stack[nstack++]=n->child[1];
            if (n->child[0])
              stack[nstack++]=n->child[0];
            }
          else
            {
            if (n->child[0])
              stack[nstack++]=n->child[0];
            if (n->child[1])
              stack[nstack++]=n->child[1];
            }

            }
        else
          if (FP_SIGN_BIT(d)==0)
          {
            if (n->child[0])
              stack[nstack++]=n->child[0];
          }
          else
          {
            if (n->child[1])
              stack[nstack++]=n->child[1];
          }
      }
    }
}
```

将伪代码扩充为全部代码也包括从前至后的排序。

下面是使用线段和点列表作为选择对象一般排序的伪代码：

```
recurse_bsp( point1, point2, objtype, pvsnode )
{
  clear selection list

  push(bsp root node)

  while (stack is not empty)
  {
    node = pop_node()

    if (node is leaf)
    {
      if (pvsnode not equal to -1 or PVS_TEST(pvsnode,node))
      {
```

```
            add node to selection list

            if (objtype not equal to -1)
              if (objtype equal to 0)
                add all objects in node to selection list
              else
                add all objects in node with type
                objtype to selection list
          }
        }
        else
        {
          dist1 = distance from point1 to node plane
          dist2 = distance from point2 to node plane

          if (dist1*dist2<0)
          {
            push(node child 0)
            push(node child 1)
          }
          else
          {
            if (dist1>0)
              push(node child 0)
            else
              push(node child 1)
          }
        }
      }
    }

    recurse_bsp( points[], numpoints, objtype, pvsnode )
    {
      clear selection list

      push(bsp root node)

      while (stack is not empty)
      {
        node = pop_node()

        if (node is leaf)
        {
          if (pvsnode not equal to -1 or PVS_TEST(pvsnode,node))
          {
            add node to slection list

            if (objtype not equal to -1)
              if (objtype equal to 0)
                add all objects in node to selection list
              else
                add all objects in node with type objtype
                to selection list
          }
        }
        else
        {
          dist0 = distance from points[0] to node plane

          for i = 1 to numpoints
          {
            dist = distance from points[i] to node plane
            if (dist1*dist2<0)
              break;
```

```
        }
        if (i<>numpoints)
        {
          push(node child 0)
          push(node child 1)
        }
        else
        {
          if (dist0>0)
            push(node child 0)
          else
            push(node child 1)
        }
      }
    }
  }
```

1.5 高级静态光照——辐射度

这一节需要参阅附录 1.2，它给出了经典辐射度理论的处理方法。

在 1.3.3 节我们说过，既然构造过程是离线的，那么任何的光照贴图可以用于指定照射强度值去照亮图。当前，可用来向环境发散的最高质量照射方法是辐射度。这常用于 CAAD (Computer Aided Architecture Design)，为由实际照明装置照射的室内装饰，生成精确实时的光照效果（这与通常计算机图形学中的点光源近似是相反的）。然而，辐射度方法极为耗时，这给构建过程加上了一个不可接受的等待时间。在这一节，我们介绍一种快速辐射度方法，来扩充可用的 BSP/PVS 装置。

第一个要关注的问题是：为什么要用有附带时间损失的辐射度？答案是质量；尽管用辐射度与用单一亮度模型（非全局）绘制相同的图二者之间的差异很小，但精确地说人眼对这些细微的差别还是很敏感的。由于辐射度方法是全局的照明方法，环境中那些不能直接“看见”光线资源的区域有正确计算的反射照明，而不是在其周围设置一个（独立的）组件。如我们所知，阴影算法只能计算几何图形的阴影，而不是阴影面中反射的强度。辐射度算法不能把阴影的计算分开。阴影从算法中“显现”，即阴影中的区域与环境中其他的区域没有区别，通过这种方法的全局本性也拥有它们自己的强度集。同样，从阴影中的区域到不在阴影中的区域的过渡在辐射度方法中正确地渲染。仅能处理阴影几何图形的阴影算法通常只能计算实边的阴影。在辐射度方法中，除非光线有非零的发射率，光线自身会得到与其他区域一样的对待。这意味着，一个区域中的光线资源的设备成为了这个方法的一部分。

辐射度方法是一个世界性的空间算法，它计算了环境中每个表面上的光强度。这个解法在光照贴图中取得。通过将环境分成所谓的面片，这个算法计算出了解决方法，而在游戏设备中使用光照贴图像素作为面片是很方便的（比在 3D 环境中的图形还要精确）。

对这个方法的一个简单理解是，想像一个起光源作用的单一面片，当我们把其打开时就会照射起先黑暗的环境。我们计算在环境中初始面片可以看见的所有面片，并根据发射面与其他可见面之间的几何关系（从形成要素中得知），可以将光能量传送到这些面片中去。因此在第一次遍历后，我们得到了接收到光的面片的数目，同时它们也成为了发射面，并将接收到的光根据其自身面的反射系数反射出去。我们继续这个过程直到趋同为止，此时已经没有更多的光能可以考虑了。趋同的现象会产生是因为定义中的反射系数总是小于 1 的（当一

个面片接收到光能后，它总是反射出比接收时的量小的光能）。事实上这只是一个简化的解释，渐进的精确方法在附录 1.2 中描述。

图 1-22（彩页中也有）展示了对于简单场景的一种辐射度解决方法。第一张图是通过最

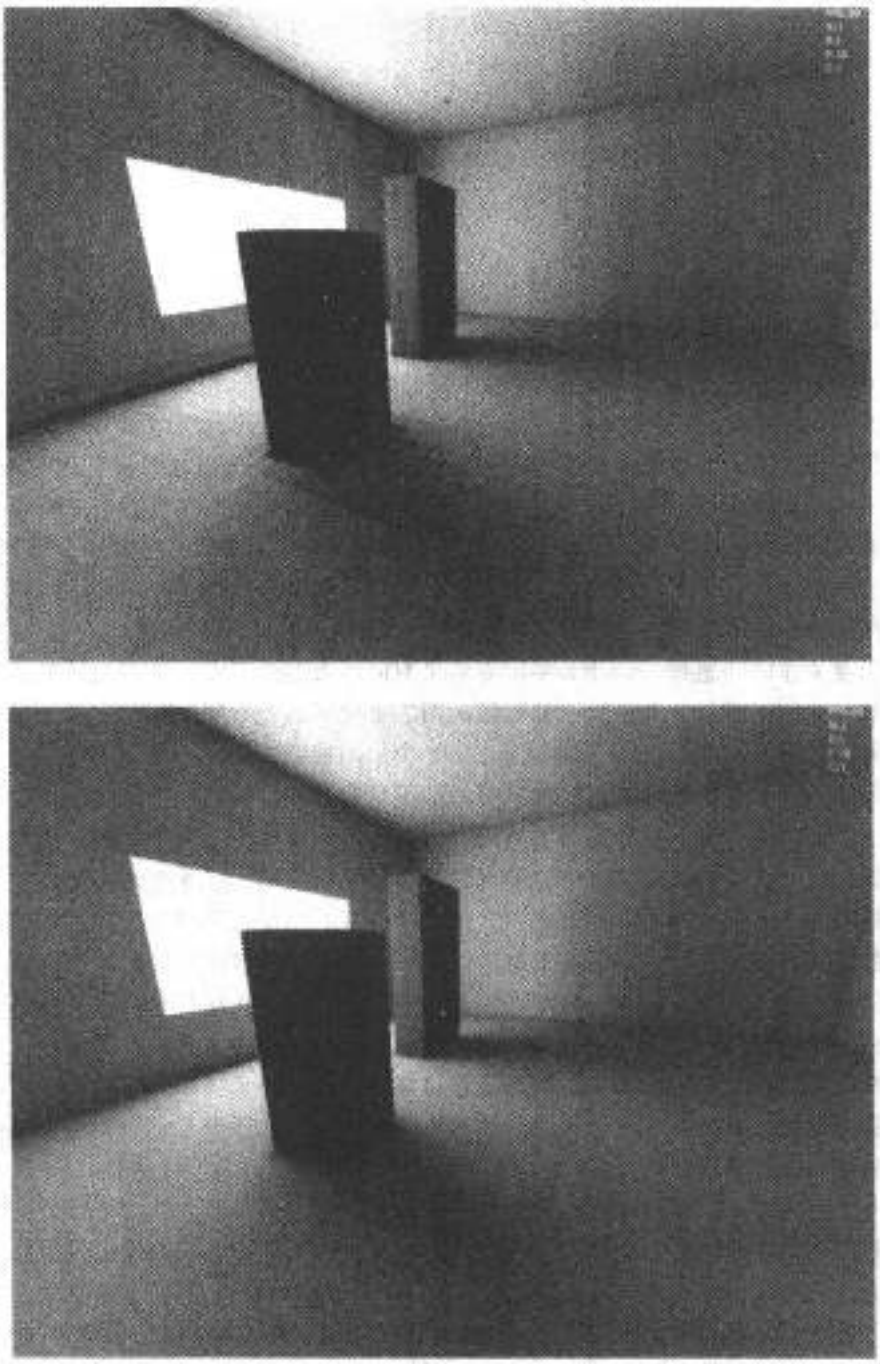


图 1-22　用辐射度方法照亮的简单环境

低限度的辐射度解决方案进行着色的，在所遇到的光照贴图像素中计算出的辐射度值是一个常数。在第二张图中，辐射度解决方案受到给出优选结果的双线性插入法的支配。这些图片展示了辐射度方法的两个性质。首先，全局、自然的解决方法是直观的。仅有的光线是从窗户而来；有窗的墙并不直接接收照射。它通过不直接的照射而被照亮。第二，一种叫做色彩溢出（color bleeding）的现象很明显出现在了有窗的墙上。这意味着色块溢出到了那些没有直接照射的墙。

因此在这样单一重复的框架中，我们必须涉及三种运算：能量传送计算以及作为其媒介的外形因素计算和评估一对面片之间的可视性的运算方法。对于外形因素计算，我们做了一个（有些不合理的）假设，那就是外形因素不会随面片的延伸而改变以及不经整合就使用公式（式 1-1）。近似的正确性取决于面片组的大小以及这些面片间的距离。

能量传送计算和外形因素计算可以按如下执行：

```
emmitted_energy=E*pixel_area
form_factor=(dot1*dot2)/(2*PI*R^2)
transferred_energy=emmitted_energy*face_reflectance*factor
receiving_pixel_R += transferred_energy
receiving_pixel_E += transferred_energy
```

以上的定义要嵌入下面的结构中：

```
初始化所有光照贴图像素
    R = 0（R 存储每个像素积累的能量）
    E = 发射面的值（如光线、窗等）
计算每个光照贴图的总能量
计算所有场景的能量，以此作为所有光照贴图的总能量
while 所有还没有发出的场景能量 > 阈值
求还没有发出能量最多的光照贴图
for 所有在选定的光照贴图中的像素
    在场景空间中求出像素位置和法线
    从像素中发出能量
    对该像素点对应的 E 清零
光照贴图总能量清零并把这部分能量转移到总的场景能量中
```

在计算从当前发射的像素可见的那些像素时，我们调用一个基于 PVS 节点选择和一个快速光线相交检测（考虑到效率，限制每个发射面只有一条交线；换句话说，每个像素的部分可视性是不被考虑的）。一个极端的执行情况是当场景复杂度增加时所带来的巨额成本。执行时使用下列三个方法：

```
Intensity(C) = grayscale value for color C

// main radiosity lighting
void flyEngineBuild::radiosity_lighting()
{
   int l,count=radmaxpasses;
   float f;
   printf("\n");
   while(count--)
   {
      l=find_lm(radminenergy);
      if (l==-1)
         break;
```

```
        f=Intensity(unshotenergy);
        printf("Radiosity energy left : %-12.2f (%5.1f%%)\r",
          f,100.0f-100.0f*f/starttotalenergy);

        if (((flyLightMapBuild *)lm[l])->emmit_light()==0)
          break;
      }
      printf("\n");
    }

    // finds light map with largest energy to emit
    int flyEngineBuild::find_lm(float energy)
    {
    int i,j=-1;
    flyVector v;
    float f;
    for( i=0;i<lm.num;i++ )
    {
      v=((flyLightMapBuild *)lm[i])->totalenergy;
      f=Intensity(v);
      if (f>energy)
      {
        energy=f;
        j=i;
      }
    }
    return j;
}

// emit light from all pixels in light map
int flyLightMapBuild::emmit_light()
{
    int x,y,i,j,k;
    flyVector p,n,energy;

    float fi=1.0f/sizex,fj=1.0f/sizey,fu,fv;

    flyBspNode *node;
    flyMesh *mesh;
    flyFace **face;
    float *f=emm;
    flyLightMapBuild *lmb;
    flyVector transferedenergy(0);

    fv=fj*0.5f;
    for( y=0;y<sizey;y++,fv+=fj )
    {
      fu=fi*0.5f;
      for( x=0;x<sizex;x++,fu+=fi,f[0]=f[1]=f[2]=0,f+=3,cur_emm++ )
      {
      map_point_local(fu,fv,p,n);
      energy.vec(pixelarea*f[0],pixelarea*f[1],pixelarea*f[2]);

      node=g_flyengine->find_node(p);
      if (node==0)
        continue;
      j=node->leaf;

      for( i=0;i<g_flyengine->nleaf;i++ )
      if (FLY_GLOBAL_PVS_TEST(j,i) &&
        g_flyengine->leaf[i]->elem.num &&
```

```
            g_flyengine->leaf[i]->elem[0]
               ->type==FLY_TYPE_STATICMESH)
         {
            mesh=((flyStaticMesh *)g_flyengine->leaf[i]
               ->elem[0])->objmesh;
            if (mesh)
         {
         face=mesh->faces;
         for( k=0;k<mesh->nf;k++ )
            if (face[k]->lastupdate!=cur_emm &&
               face[k]->lm!=-1) // lightmap radioity
            {
               face[k]->lastupdate=cur_emm;
               lmb=(flyLightMapBuild *)g_flyengine
                  ->lm[face[k]->lm];
               if (this!=lmb)
                  transferedenergy+=lmb
                  ->receive_light(p,n,energy);
            }
            else // vertex colour radiosity
            if (face[k]
               ->facetype==FLY_FACETYPE_TRIANGLE_MESH)
            {
               face[k]->lastupdate=cur_emm;

   transferedenergy+=tri_receive_light(face[k],p,n,energy);
            }
         }
      }
      }
   }
   ((flyEngineBuild *)g_flyengine)->unshotenergy-=totalenergy;
   totalenergy.vec(0,0,0);

   return 1;
}

// loops all pixels in light map and
flyVector flyLightMapBuild::receive_light(const flyVector& P,const
flyVector& N,const flyVector& E,int flag)
{
   int x,y;
   float fi=1.0f/sizex,fj=1.0f/sizey,fu,fv;
   float *f=emm,*r=rad;
   flyVector p,n,dir,e;
   float l2,dot1,dot2;

   flyVector receivedenergy(0);

   ((flyEngineBuild *)g_flyengine)->unshotenergy-=totalenergy;
   fv=fj*0.5f;
   for( y=0;y<sizey;y++,fv+=fj )
   {
      fu=fi*0.5f;
      for(
x=0;x<sizex;x++,fu+=fi,receivedenergy+=flyVector(f[0]*pixelarea,f[1]*
pixelarea,f[2]*pixelarea),f+=3,r+=3 )
      {
         // map pixel to 3D point p and normal n
         map_point_local(fu,fv,p,n);
      dir=P-p;
```

```
        l2=dir.length2();
        if (l2<1.0f)
          continue;

        dir*=1.0f/(float)sqrt(l2);

        dot1=FLY_VECDOT(dir,n);
        if (dot1<=0.01f)
          continue;

        if (flag==0) // point light emit all 360 degrees
        {
          dot2=-FLY_VECDOT(dir,N);
          if (dot2<=0.01f)
            continue;

          e=(E*reflectance)*((dot1*dot2)/(l2*FLY_2PI));
        }
        else
          // surface light emit 180 degrees
          e=(E*reflectance)*(dot1/(l2*2.0f*FLY_2PI));
        if ((lmshadows&8) &&
          g_flyengine->collision_test(P-
dir,p+dir,FLY_TYPE_STATICMESH))
          continue;

        f[0]+=e.x;
        f[1]+=e.y;
        f[2]+=e.z;

        r[0]+=e.x;
        r[1]+=e.y;
        r[2]+=e.z;
      }
    }

    e.vec(receivedenergy.x-totalenergy.x,receivedenergy.y-
totalenergy.y,receivedenergy.z-totalenergy.z);
    totalenergy=receivedenergy;
    ((flyEngineBuild *)g_flyengine)->unshotenergy+=totalenergy;

    return e;
}
```

图 1-23（彩页中也有）展示了一个游戏层次使用正常着色和使用辐射度之间的一个比较。质量上的提升是很明显的。当然，实际上两种渲染之间的许多差异被结构化绘制所掩盖，但是回到更早提到过的一点，那就是人眼对环境中照明不充分是很敏感的。所有这些图片可见的一个问题是边缘的阴影。在经典辐射度理论中，这个问题有自己的研究领域，在[WATT00]中有各种网格化策略的概述。基本上，我们需将在计算的辐射度上有快速改变区域中的面片进行细分。但是这不是一个简单的问题，因为我们要到计算出解决方案才会知道这些区域在哪儿。

将质量问题放在一边，我们现在回过来考虑效率。每个类型面的三角网格细节或者弯曲的贝济埃曲面是一个三角形列表。如果弯曲或者细节网格太复杂的话，由于它们对每个三角形进行检测，线相交计算会变得很慢。解决这个问题的一个简单方法是将每个面封入一个八叉树（octree）中。

八叉树是一个包围盒的树，其每个树叶上我们有一个表示代表裁剪节点包围盒的三角形

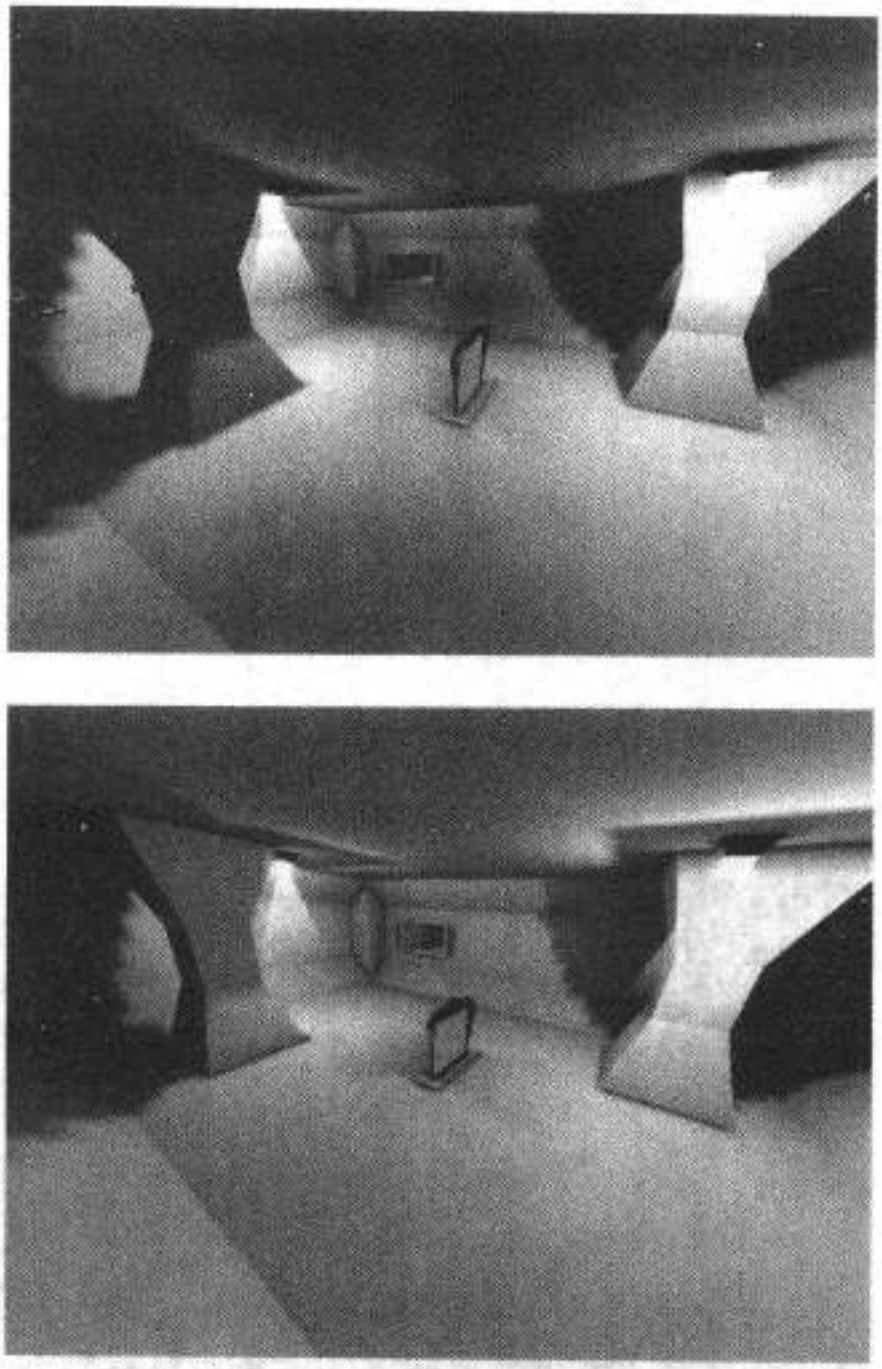


图 1-23 使用和不使用辐射度渲染之间的一个比较

索引的整数列表。该节点在包围盒的中心点上被细分（创建 8 个新的子节点）。我们开始把所有的面放在根节点上面，并调用一个递归的方法，该方法递归的停止以其细分集是否有效为根据。光线相交方法递归了与包围盒相交的树并且仅在相碰撞的盒子上继续递归。碰撞的

盒子使用 *clip_bbox* 方法去找在八叉树节点中被一个提供的包围盒所裁剪的面。当对于父节点中的 N 个面，没有一个子节点会有超过 $N*0.6$ 个面时，那么我们认为这个细分是有效的。这些面被分为 8 个区域。如果超过半数的面进入了不同的节点中，那么我们认为它是好的。图 1-24（见彩页）展示了一个以贝济埃曲面和三角网格生成八叉树的场景。

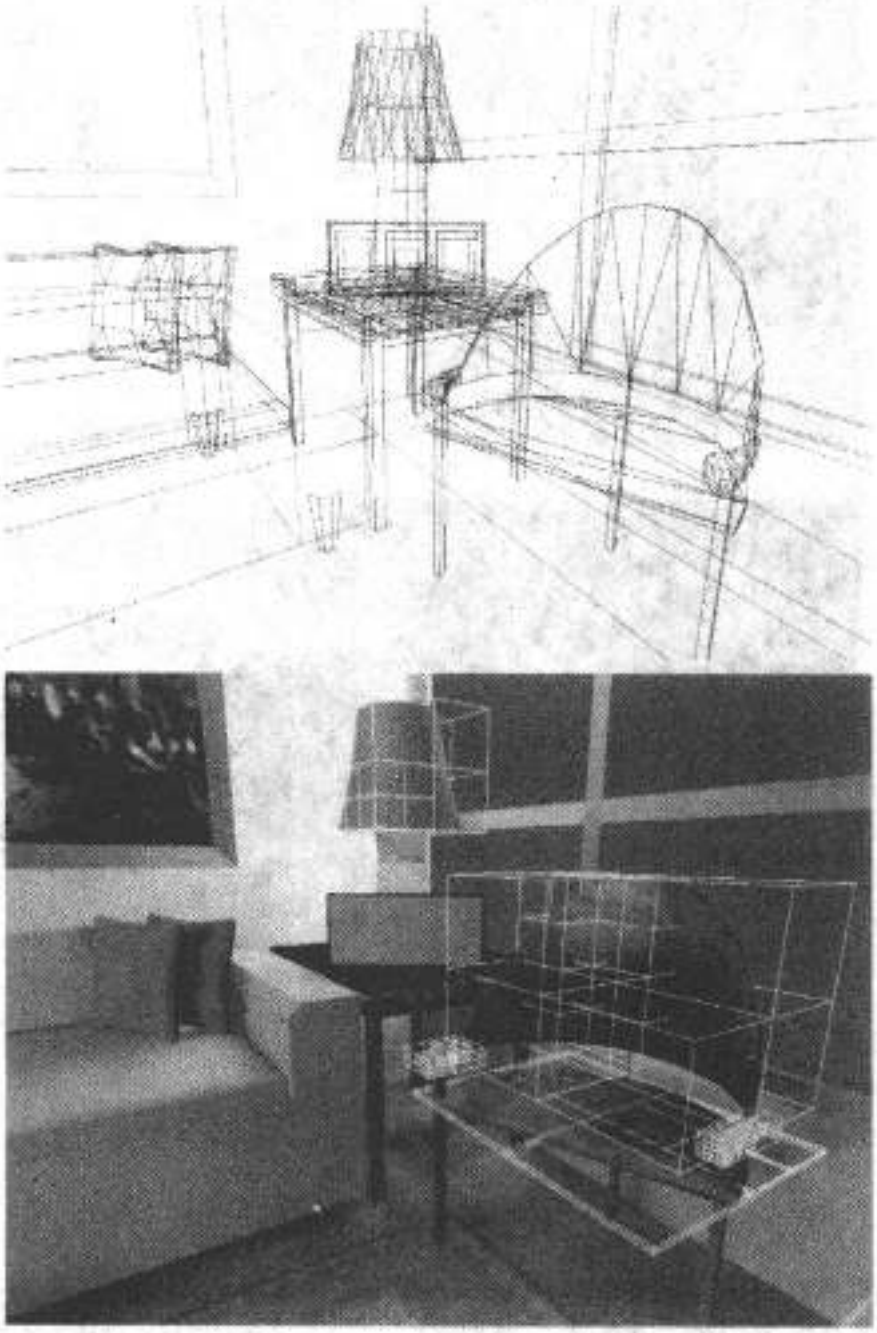


图 1-24　一个以贝济埃曲面和三角网格生成八叉树的场景

下面的代码执行了八叉树算法。

```
//! OcTree node class
class FLY_ENGINE_API flyOcTreeNode
{
public:
   flyBoundBox bbox;          //!< Node bound box
   flyArray<int> faces;       //!< Node triangle faces
   flyOcTreeNode *nodes[8]; //!< Node childs

   //! Default constructor
   flyOcTreeNode();

   //! Default destructor
   virtual ~flyOcTreeNode();

   //! Copy constructor
   flyOcTreeNode(flyOcTreeNode& in);

   //! Split faces into child nodes if a subdivison is needed (used
   //on the octree build process)
   void build_node(int *triverts,flyVertex *verts);
};

//! OcTree class
class FLY_ENGINE_API flyOcTree
{
public:
   flyOcTreeNode *root;    //!< Root node for octree
   flyFace *face;          //!< Face from where octree was
                           //created
   //! Default construçtor
   flyOcTree();

   //! Default destructor
   virtual ~flyOcTree();

   //! Copy constructor
   flyOcTree(flyOcTree& in);

   //! Operator equal
   void operator=(flyOcTree& in);

   //! Free the tree data
   void reset();

   //! Builds the octree for the given triangle face or Bezier face
   void build_tree(flyFace *f);

   //! recurse octree and ray intersect test the triangles (just
   // bool result with no intersection info)
   int ray_intersect_test(const flyVector& ro,const flyVector&
            rd,float dist) const;

   //! recurse octree and ray intersect the triangles for the
   // closest collision
   int ray_intersect(const flyVector& ro,const flyVector&
            rd,flyVector& ip,float& dist) const;
```

```
  // Recurse octree and find all faces from nodes clipped by the
  //given bbox
  void clip_bbox(const flyBoundBox& bbox,flyArray<int>& faces) const;

  //! Draw the bbox as wireframe
  void draw();
};

// Build octree from triangles found in face f
void flyOcTree::build_tree(flyFace *f)
{
  face=f;
  root=new flyOcTreeNode;

  for( int i=0;i<f->ntriface;i++ )
       root->faces.add(i);
  root->bbox=f->bbox;

  root->build_node(f->trivert,f->vert);
}

// Split node into eight sub-nodes, separate faces belonging to each
//node and test
// if split was within threshold parameters (if not, collapse split
//and stop recursion)
void flyOcTreeNode::build_node(int *triverts,flyVertex *verts)
{
  if (faces.num<=FLY_OCTREE_MINFACES)
    return;

  int i,j;
  flyVector center=(bbox.min+bbox.max)*0.5f;

  for( i=0;i<8;i++ )
    nodes[i]=new flyOcTreeNode;

  nodes[0]->bbox.min.vec(bbox.min.x,bbox.min.y,bbox.min.z);
  nodes[0]->bbox.max.vec(center.x,center.y,center.z);

  nodes[1]->bbox.min.vec(center.x,center.y,bbox.min.z);
  nodes[1]->bbox.max.vec(bbox.max.x,bbox.max.y,center.z);

  nodes[2]->bbox.min.vec(center.x,bbox.min.y,bbox.min.z);
  nodes[2]->bbox.max.vec(bbox.max.x,center.y,center.z);

  nodes[3]->bbox.min.vec(bbox.min.x,center.y,bbox.min.z);
  nodes[3]->bbox.max.vec(center.x,bbox.max.y,center.z);

  nodes[4]->bbox.min.vec(bbox.min.x,bbox.min.y,center.z);
  nodes[4]->bbox.max.vec(center.x,center.y,bbox.max.z);

  nodes[5]->bbox.min.vec(center.x,center.y,center.z);
  nodes[5]->bbox.max.vec(bbox.max.x,bbox.max.y,bbox.max.z);

  nodes[6]->bbox.min.vec(center.x,bbox.min.y,center.z);
  nodes[6]->bbox.max.vec(bbox.max.x,center.y,bbox.max.z);

  nodes[7]->bbox.min.vec(bbox.min.x,center.y,center.z);
  nodes[7]->bbox.max.vec(center.x,bbox.max.y,bbox.max.z);

  int v;
  flyBoundBox bb;
```

```
    for( i=0;i<faces.num;i++ )
    {
      v=faces[i]*3;
      bb.reset();
      bb.add_point(verts[triverts[v]]);
      bb.add_point(verts[triverts[v+1]]);
      bb.add_point(verts[triverts[v+2]]);
      for( j=0;j<8;j++ )
        if (nodes[j]->bbox.clip_bbox(bb.min,bb.max))
          nodes[j]->faces.add(faces[i]);
    }
    for( i=0;i<8;i++ )
      if (nodes[i]->faces.num>faces.num*3/5)
        break;

    if (i<8)
    {
      for( i=0;i<8;i++ )
      {
        delete nodes[i];
        nodes[i]=0;
      }
    }
    else
    {
      faces.free();

      for( i=0;i<8;i++ )
        if (nodes[i]->faces.num==0)
        {
          delete nodes[i];
          nodes[i]=0;
        }
        else
          if (nodes[i]->faces.num>FLY_OCTREE_MINFACES)
            nodes[i]->build_node(triverts,verts);
    }
}

// Ray intersect octree faces returning true or false
// only on first intersection and with no intersection info
int  flyOcTree::ray_intersect_test(const  flyVector&  ro,const
flyVector& rd,float dist) const
{
    static flyOcTreeNode *stack[64];
    static float f1,f2;
    if (root->bbox.ray_intersect(ro,rd,f1,f2)==-1)
      return 0;

    flyOcTreeNode *n;
    int nstack=1,i;
    stack[0]=root;

    while(nstack)
    {
      n=stack[--nstack];
      if (n->faces.num==0)
      {
        for( i=0;i<8;i++ )
          if (n->nodes[i] &&
            n->nodes[i]
            ->bbox.ray_intersect(ro,rd,f1,f2)!=-1)
```

```
            stack[nstack++]=n->nodes[i];
      }
      else
         if (face->ray_intersect_tri_test(n->faces.buf,
         n->faces.num,ro,rd,dist))
            return 1;
   }
   return 0;
}
// Ray intersect octree faces returning face number (-1 on no
// intersection),
// closest intersection point (ip) and intersection point distance
(dist)
int flyOcTree::ray_intersect(const flyVector& ro,const flyVector&
rd,flyVector& ip,float& dist) const
{
   static flyOcTreeNode *stack[64];
   static float f1,f2;
   if (root->bbox.ray_intersect(ro,rd,f1,f2)==-1)
      return -1;
   flyOcTreeNode *n;
   int nstack=1;
   stack[0]=root;
   int i,min_face=-1;
   float min_dist=FLY_BIG;
   flyVector min_ip;
   while(nstack)
   {
      n=stack[--nstack];
      if (n->faces.num==0)
      {
         for( i=0;i<8;i++ )
            if (n->nodes[i] &&
               n->nodes[i]
               ->bbox.ray_intersect(ro,rd,f1,f2)!=-1)
               stack[nstack++]=n->nodes[i];
      }
      else
      {
         i=face->ray_intersect_tri(n->faces.buf,
            n->faces.num,ro,rd,ip,dist);
         if (i!=-1 && dist<min_dist)
         {
            min_dist=dist;
            min_ip=ip;
            min_face=n->faces[i];
         }
      }
   }
   ip=min_ip;
   dist=min_dist;
   return min_face;
}
// Recurse octree filling in faces array with all faces inside
// octree leaf nodes clipped by specified bound box
void flyOcTree::clip_bbox(const flyBoundBox& bbox,flyArray<int>&
faces) const
{
```

```
    static flyOcTreeNode *stack[64];
    faces.clear();
  if (bbox.clip_bbox(root->bbox.min,root->bbox.max)==0)
    return;

  flyOcTreeNode *n;
  int nstack=1,i;
  stack[0]=root;

  while(nstack)
  {
    n=stack[--nstack];
    if (n->faces.num==0)
    {
      for( i=0;i<8;i++ )
        if (n->nodes[i] &&
          bbox.clip_bbox(n->nodes[i]->bbox.min,
            n->nodes[i]->bbox.max))
          stack[nstack++]=n->nodes[i];
    }
    else
      faces+=n->faces;
  }
}
// Draw recursing octree and draw leaf nodes' bound boxes
void flyOcTree::draw()
{
  static flyOcTreeNode *stack[64];

  flyOcTreeNode *n;
  int nstack=1,i;
  stack[0]=root;

  root->bbox.draw();

  while(nstack)
  {
    n=stack[--nstack];
    if (n->faces.num==0)
    {
      for( i=0;i<8;i++ )
        if (n->nodes[i])
          stack[nstack++]=n->nodes[i];
    }
    else
      n->bbox.draw();
  }
}
```

附录 1.1　构造实践

在每个以实践为目标的章节末，我们给出课本中所描述的内容的实际演示。第一个演示有关于构造过程。

(1) 从现有的一个层次构造

一个 .fmp 文件中包含现有层次中的原始几何图形。我们需要通过创建 BSP、PVS 以及光照贴图来进行构造过程。对此，我们使用 *flyBuild* 控制台前端（第 3 章）。为了使事情变得尽可能地简单，这都由一个叫做 *flyBuild* 的简单界面来控制。基于选项的选择，用合适的命令

行语句调用 *FlyBuild*。

打开 *FlyBuild* 工具，选择贴图文件 *ship_mp1.fmp* 并确保所有的选项设置如图 A1-1 所示。这会产生一个包括 PVS 和阴影的完整的层次构造。在构造完成之后，整个层次已准备好运行。在高级选项窗口中有意义的参数包括：

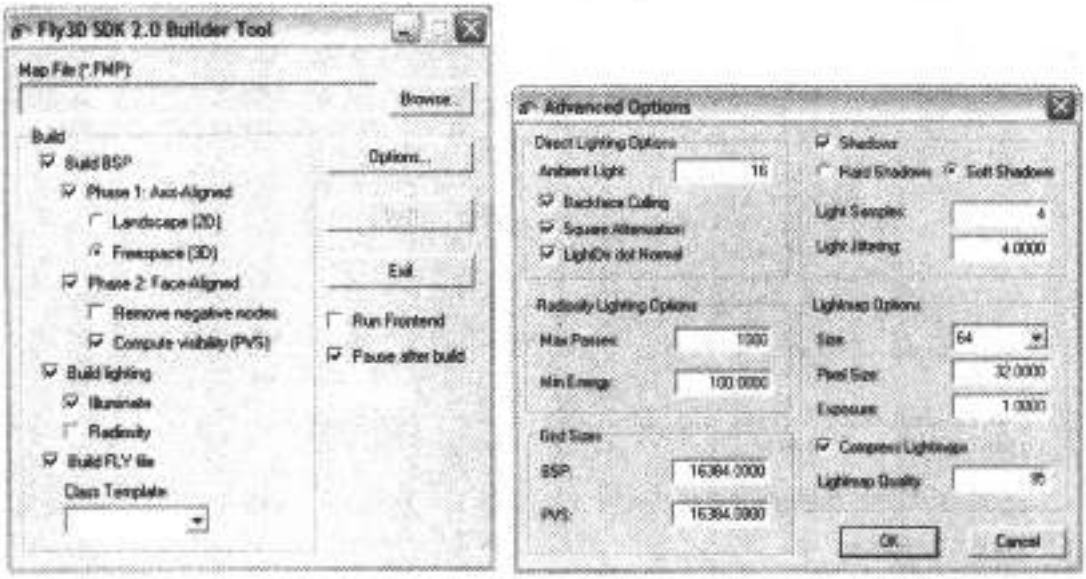


图 A1-1　构造器工具与选项

- 将光照贴图像素的大小改为更大或更小的值。这会产生更粗劣或更细致的阴影（用 F8 键关掉纹理滤波以更好地进行观察）。
- 选择硬的阴影，光照贴图计算会变得更快，但这是以低质量的阴影为代价的。
- BSP 网格的大小改变会使得层次中 BSP 段数目改变。对小的层次来说，这并不会明显地改变其构造过程的速度。然而，对于大的层次这是一个很重要的参数（如火星地形图 *mars.fmp*）。载入这个层时要注意，你必须检查 *landscape* BSP 盒。

(2) 将现有的层次导入层次编辑器

载入 3D Studio MAX 并确保已经安装了 Fly3D 插件。用 3D Studio MAX 导入命令选择 Fly3D MAP 格式（.fmp）。然后从 SDK 数据文件夹中选择一个已有的演示。

在 3D Studio MAX 中载入图中所有的几何图形，并用合适的纹理贴图和映射坐标去创建材质。游戏中所有的光线会被转换成 3D Studio MAX 的光线，而所有的游戏实体都置于 3D Studio MAX 对象服务的用户参数中，含有简单的参数范围。

光线在 3D Studio MAX 中的绘制与实际游戏中出现的相似。只有光线的颜色和长距离衰减参数被使用，而 3D Studio MAX 阴影贴图会充分预览构造过程中的发射光线。当创建新的光线时，你只要设置一些涉及的参数就可以了。

图文件中的几何图形会被载入，并基于不同几何图形类型被分为不同的对象（见图

A1-2，彩页中也有）。以下是所支持的几何类型的列表和它们的特征标识符：

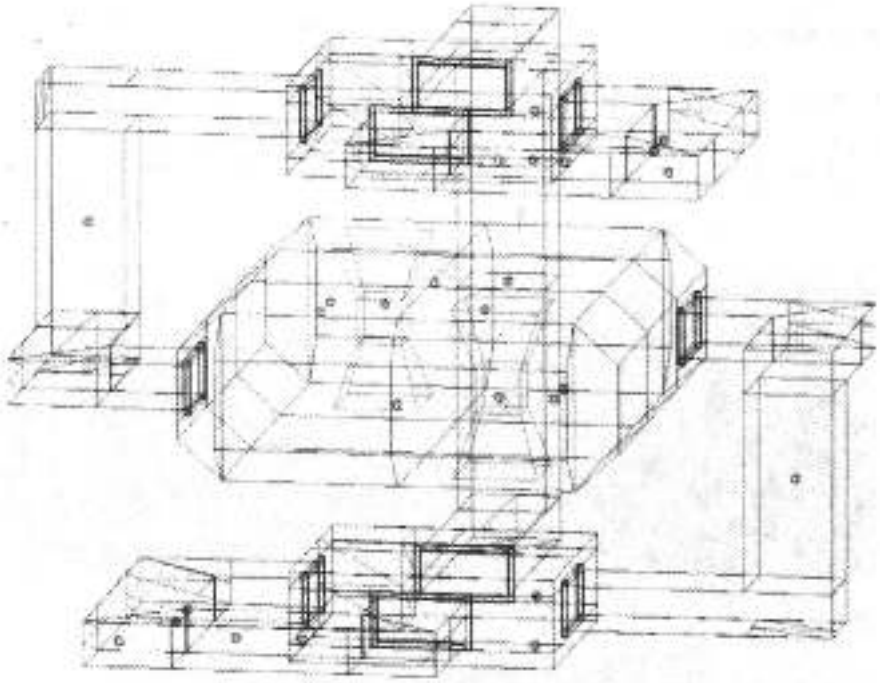


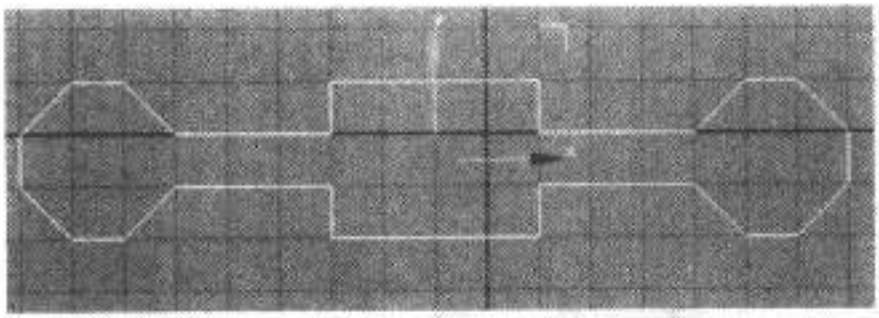
图 A1-2 用不同几何图形展示的线框层次
黑色 结构化多边形 绿色 细节多边形 红色 游戏实体 紫色 曲面 黄色 光线

- 结构化多边形（*） 这样标记的物体将被转换成一个由大的凸多边形（一个多边形有任意数目的顶点）组成的集合。把所有这些结构化多边形连起来，必然形成一个封闭的凹形体。一个 BSP 分割平面会来分割自这些面所定义的所有平面。结构化多边形只能通过边与其他结构化多边形相连接（一条边不能通过表面与另外一个多边形相连）
- 细节多边形（&） 这样标记的物体将被转换成一个由大多边形（一个多边形有任意数目的顶点）组成的集合。这些多边形在这一层次中可以自由地放在任何位置，而且不需要被不接触到的几何物体
- 细节三角形网格（^） 每个这样标记的物体被认为是一个基于三角形的细节表面。用于表示灯、雕像和所有小细节几何物体的模型
- 地形表面（#） 这样标记的物体将被用于表示地形。它们将被存储为一个个独立的三角形，并被构造工具以一种不同的方式处理。在 BSP 构造程序结束的时候，处于同一 BSP 段的三角形将被转化为一个细节三角形网格（类型^），允许来自同一个 BSP 段的面共享它的顶点。
- 贝济埃曲面（~） 每个这样标记的物体将被认为是一个双二次贝济埃曲面，在 u 和 v 方向上拥有任意数目的面片。网格上的每个顶点被认为是一个曲面的控制点。
- 游戏实体（$） 这样标记的物体表示在 Fly3D 插件中定义的游戏实体，例如 power-

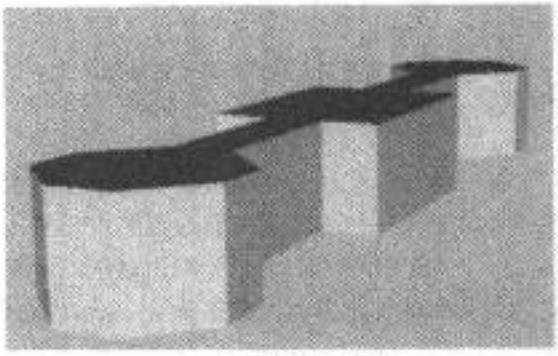
up、birthpad 和所有动态的物体。你可以在编辑器里面确定它们的位置、最终它们被自动转换成拥有位置、旋转和其他用户定义的属性的层次脚本。

(3) 创建简单的层次

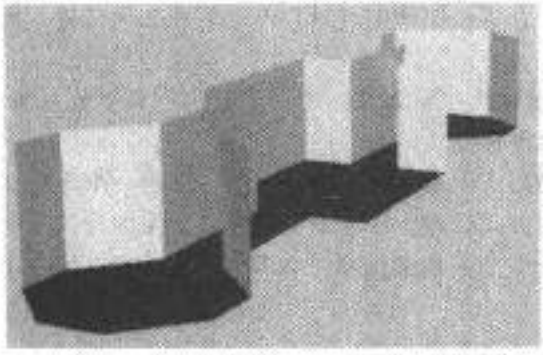
创建一个新的层次首先需要为所有的几何物体建模，然后用标准的 3D Studio MAX 材质（多对质也支持）对这些物体进行纹理贴图，最后用合适的标志标记这些物体。实体可以是任何形状的几何物体（在上面的例子中用球体表示）。实体最初的位置和方向将在层次被导



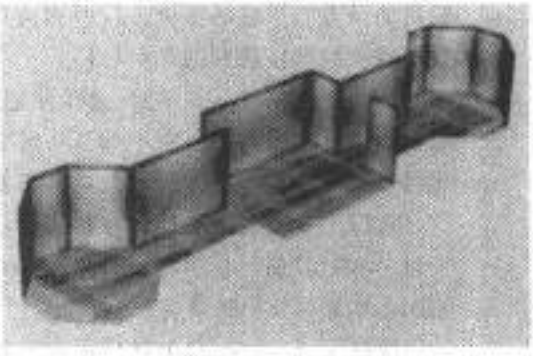
a) 层次的平面版



b) 层次的三维模型



c) 转换后的层次的三维模型



d) 贴图后的层次的三维模型

图 A1.3

出的时候获得。在简单演示中，我们要求在层次中至少有一个 birthpad 实体；否则，当层次被激活的时候，船将被放置在原点处（这样船可能在层次的外面而看不见）。

要构造一个层次，我们可以按照下面的步骤。

构造层次的结构化面

用画线工具绘制层次的平面图。模型的平面图必须由一个封闭的多边形组成，而且必须与平面图的栅格相吻合（见图 A1-3a）。

用 Extrude 修改器把平面图转换成三维模型（见图 A1-3b）。

用 Edit Mesh 修改器反转各个平面的法向量。首先在修改器中选择 face 节点，然后通过选择窗口选择模型中所有的面，最后点击 Invert Normals（反转法向量）按钮。这是非常重要的一步，它把一个拥有指向外面的法向量的三维物体转换为指向内部（见图 A1-3c）。

在 UVW Map 修改器中选择盒子映射（box mapping），为层次设置合适的长度、宽度和高度值，生成纹理坐标。

用标准材质和位图漫反射映射定义所需要的材质并把它应用到模型中（见图 A1-3d）。你也可以对模型的不同部分应用不同的材质。

下一步是把创建好的网格设置成“结构化”（structural）。Fly3D 使用前缀标记系统来识别不同类型的网格和面。要把网格和面设置成结构化面，只需要修改它的名字，在它的名字前面添加“*”。例如，一个命名为“structure”的网格要改名为“*structure”。

让层次和模型保持一定的特性是非常重要的，特别是当建造更加复杂的环境的时候：所有的结构化面必须形成一个封闭的凹形体。这个可以通过使所有的边满足有且只有两个面共享一条边来达到。如果这些特性能够满足，那么 BSP 构造例程就可以生成有效的正向叶节点，从而使它可以利用 PVS 简述和入口技术来寻找路径。

添加细节物体

在这一步中，我们将把细节物体添加到场景中，使场景更加真实。典型的细节物体包括像框、灯子、桌子、箱子等。图 A1-4 展示了添加灯子和箱子后的场景。

细节物体同样遵循前缀标记模式。因此，如果物体是由三角形面组成的，“^”必须添加到物体名字的开头；如果物体的面由 3 个以上顶点组成，那么应该添加“&”到物体名字的开头。

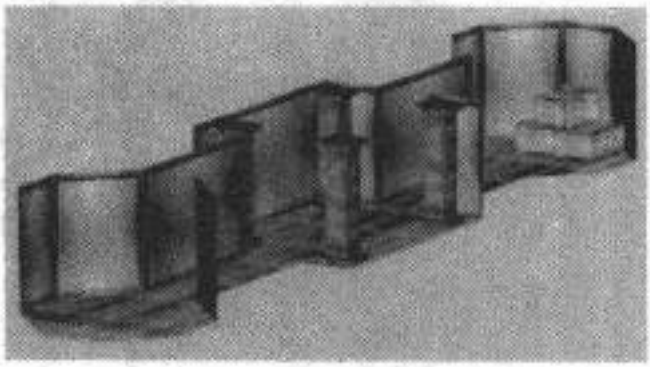


图 A1-4 添加细节物体后的层次三维模型

把细节物体添加到层次中的时候要特别注意，它们的体积必须能够完全放在由结构化模型定义的凹形体内。但是，细节面并没有像结构化面那样的限制：边和顶点能够被任意数目的面以任何方式共享。

另外要注意，由拥有超过 3 个顶点的面组成的物体将使用光照贴图进行静态的光照；而由三角形面组成的物体将使用顶点光照技术进行光照。

添加光照

在这一步中，我们将添加光照到场景中（见图 A1-5，彩页中也有）。光照必须是 Omni 类型，它们的颜色和照明半径在 Lar attenuation 子菜单中定义。光照不需要在它们的名字前面添加前缀标记。

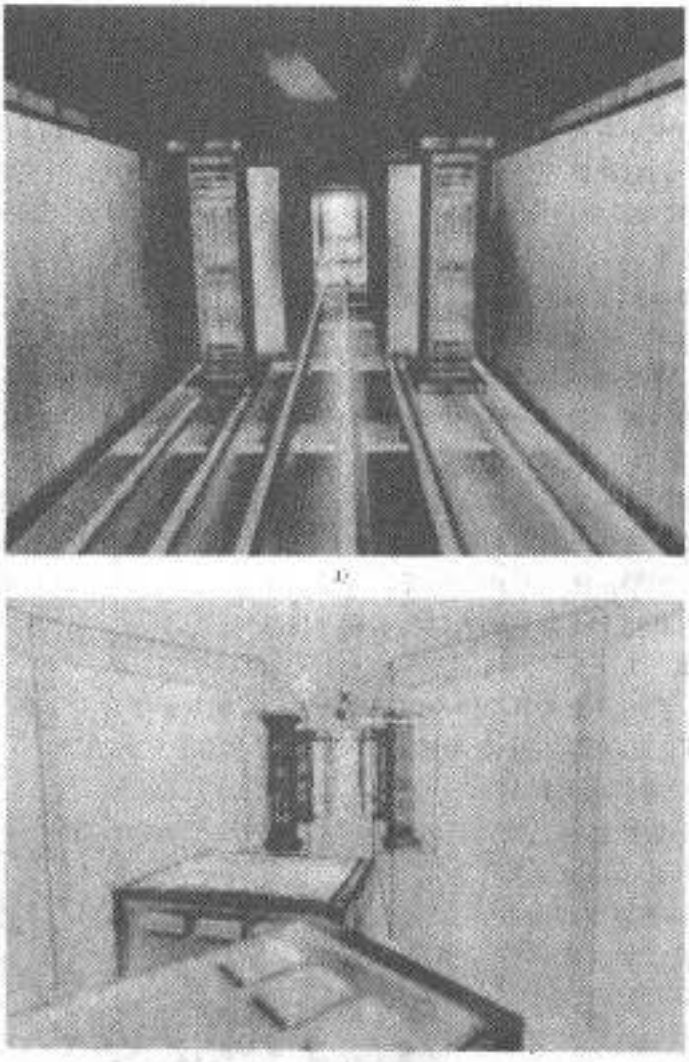


图 A1-5　添加光照后的环境

添加实体

迷宫层次的最后一步是添加实体。实体可以是 power-up、birthpad、入口、镜子等。在这个例子中，一个 birthpad 和一个 power-up 实体被添加到场景中。这些实体的名字必须符合 Fly3D 插件中相应类型的名字。这些实体用简单的网格表示，比如一个立方体。图 A1-6（彩页中也有）显示了 birthpad 和 power-up 实体（红色高亮部分）。

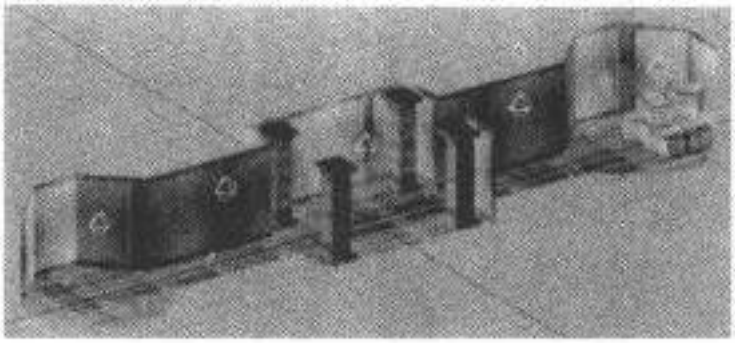


图 A1-6　添加实体

实体必须在其名字前面添加“S”前缀。

(4) 导出创建的层次

使用 3D Studio MAX 的导出命令导出创建的层次，不要忘了选择 Fly3D Map（.fmp）格式（见图 A1-7）。在所有标记的几何物体被处理之后，一个对话框将显示导出的层次的统计信息，显示每种类型有多少元素。

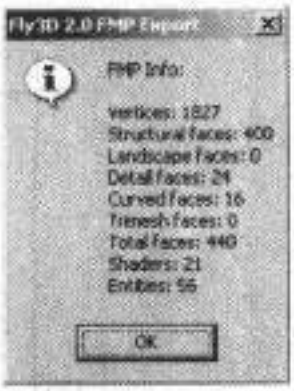


图 A1-7　显示的层次统计信息

(5) 建模、导出和构造更复杂的层次

这一节将介绍怎样为一个复杂的层次建模和处理光照、构造结构、细节物体、曲面、贴图纹理，并且把它作为 Fly3D 场景导出。可以用 3D Studio MAX 3.x 或者 4.x 作建模工具。用户应该具备关于这个软件的基础知识。另外，应该通过 flyInstPlugins.exe 应用程序安装与 MAX 版本相对应的 Fly3D 导出/导入插件和曲面插件。

一个新层次的构想

构造一个新层次的第一步跟建筑工程中的第一步一样，就是要构想结构和细节，决定尺寸和用什么类型的材质、纹理等。在图 A1-8 中的素描是在着手构造之前一个关于层次和细节物体的构想草图。

构造层次的结构化面

结构化面是那些定义了场景的几何外形的面，它们把空间分成“环境内”和“环境外”两部分，比如墙、天花板、地板等。让结构模型保持一定的特性是很重要的，特别当为更复杂的环境建模的时候：所有的结构化面必须形成一个封闭的凹形体。这个可以通过使所有的边满足有且只有两个面共享一条边来达到。如果这些特性能够满足，那么 BSP 构造例程就可以生成有效的正向叶节点，从而使它可以利用 PVS 筛选和入口技术来寻找路径。

在这个例子中（见图 A1-9），场景的结构是在 3D Studio MAX 中构造的，要注意，Snap to Grid 选项必须是打开的。层次结构也可以在 AutoCAD 中构造，然后导出为 .3ds 文件，以便在 MAX 中导入。

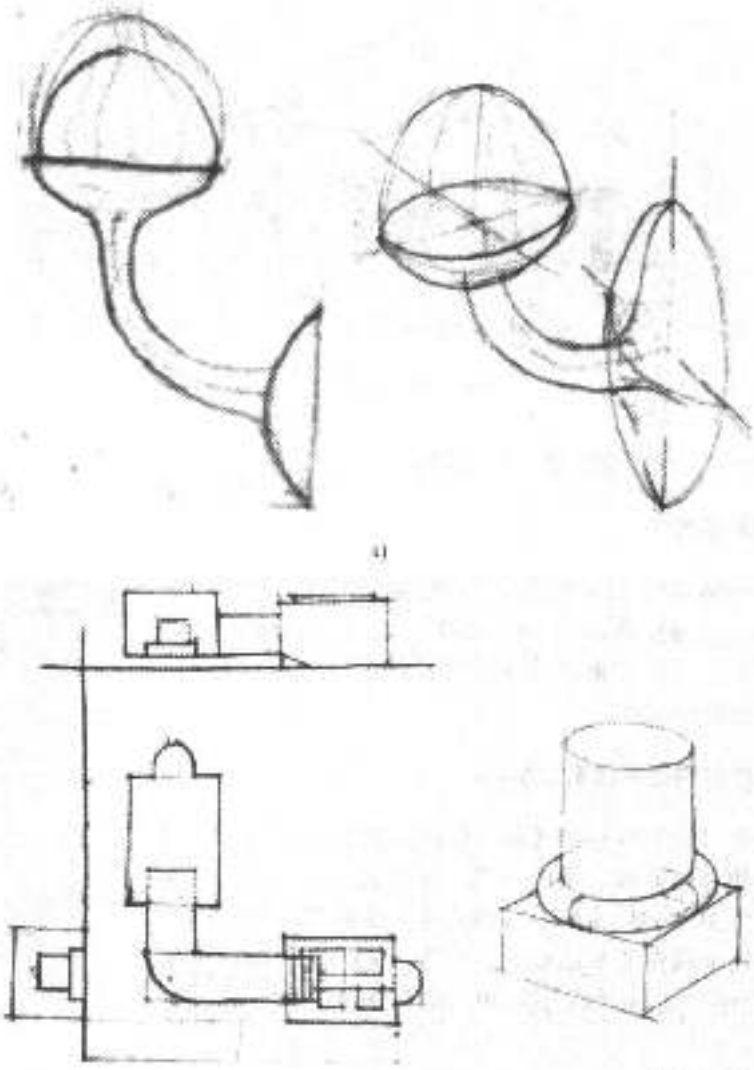


图 A1-8　细节物体和层次的构想草图

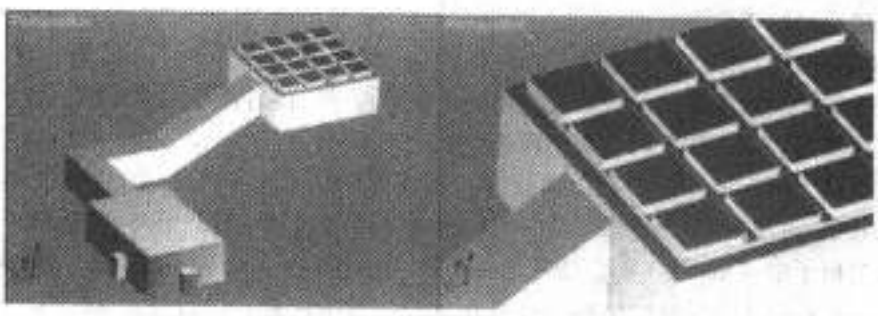


图 A1-9　层次示例

反转面的法向量是必需的，这样面就转向结构的内部，如图 A1-10 所示。

下一步就是用 UVW mapping 修改器为结构化面贴上纹理，可以使用 Planar、Box 或 Cylindrical 映射模板，也可以使用 Face Map。要使结构化面的贴图更加容易，可以利用 Detach 例程。Detach 例程可以把面分离出来，这样就可以为每个面采用单独的贴图方式。

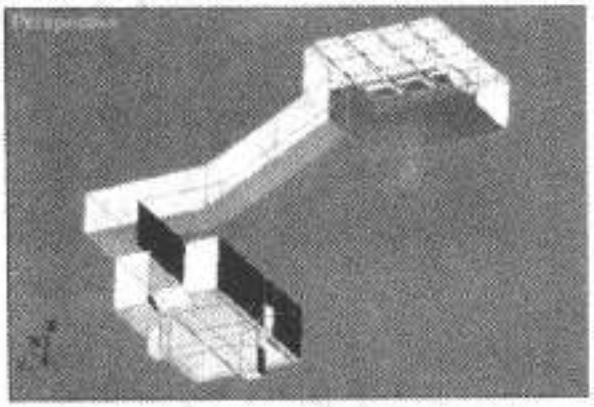


图 A1-10 反转面的法向量后的场景

下一步是把创建好的网格设置成“结构化”。Fly3D 使用前缀标记系统来识别各种不同类型的网格和面。要把网格的面设置成“结构化”，只需要修改它的名字，在它的名字前添加“*”。例如，一个名叫“structure”的网格需要改名为“*structure”。

添加细节物体

在这一步，细节物体将被添加到场景中，使场景看起来更真实。典型的细节物体包括像框、柱子、桌子、箱子、台阶等。图 A1-11（彩页中也有）演示了柱子和台阶加到场景中的情况。



图 A1-11 添加细节物体后的场景

细节物体同样遵循前缀标记模式。因此，如果细节物体是由三角形面组成的，“-”应该加到名字前面；如果细节物体的面由多于 3 个顶点组成，那么“&”应该加到名字前面。

图 A1-12（彩页中也有）演示了一个三角形细节物体（左边）和一个多边形细节物体（右边）。

当把细节物体添加到层次中时要特别小心。它们的体积必须全部位于由结构化模型定义的凹形体内。但是细节面并没有像结构化面那样的限制：边和顶点能够被任意数目的面以任何方式共享。另外要注意，由拥有超过 3 个顶点的面组成的细节物体，要使用光照贴图进行静态的光照，而由三角形面组成的物体要使用顶点光照技术进行光照。

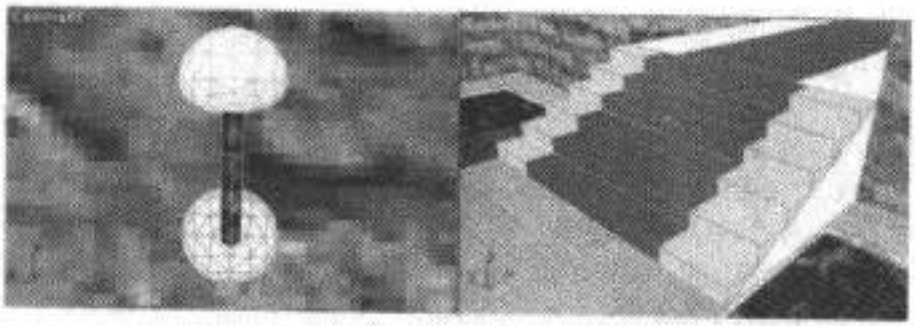


图 A1-12　三角形和多边形细节物体

添加光照

在这一步中，我们将把光源添加到场景中（见图 A1-13（彩页中也有））。光照必须是 Omni 类型，它们的颜色和照明半径在 farattenuation 了菜单中定义。光照不需要在它们的名字前面添加前缀标记。

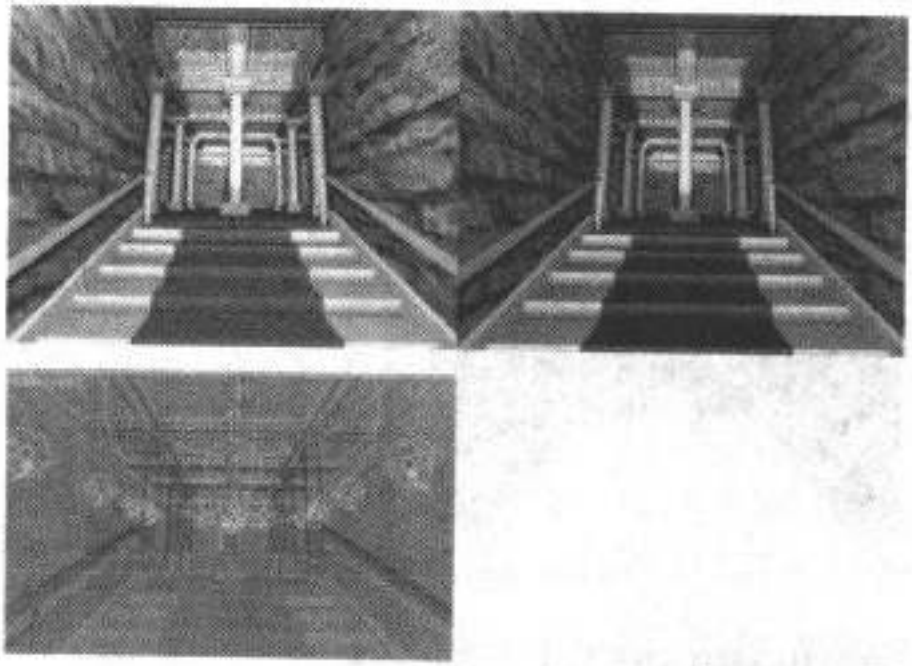


图 A1-13　添加光照

在光源 Far Attenuation 了菜单中，默认的光照半径是：Start = 80，End = 500。

光照是一个层次中增加气氛和景深的主要手段。层次的设计者最好多测试几种灯光的配置方案，以达到所需视觉效果。

(6) 用曲面建模

在这一节，我们介绍如何在 3D Studio MAX 中使用曲面物体插件为 Fly3D 构造特殊类型

曲面。这个曲面物体插件利用引擎的动态 LOD 算法和其他一些优化，特别用来构造与 Fly3D 融合在一起的曲面。

第一步：创建结构化面

在这一步中，我们将构造一个长方体，用作层次自身的结构化面，我们将在其中构造需用的曲面。在例子中，我们创建了一个大小为 1000 × 1000 × 400 的长方体，如图 A1-14 所示（彩页中也有）。

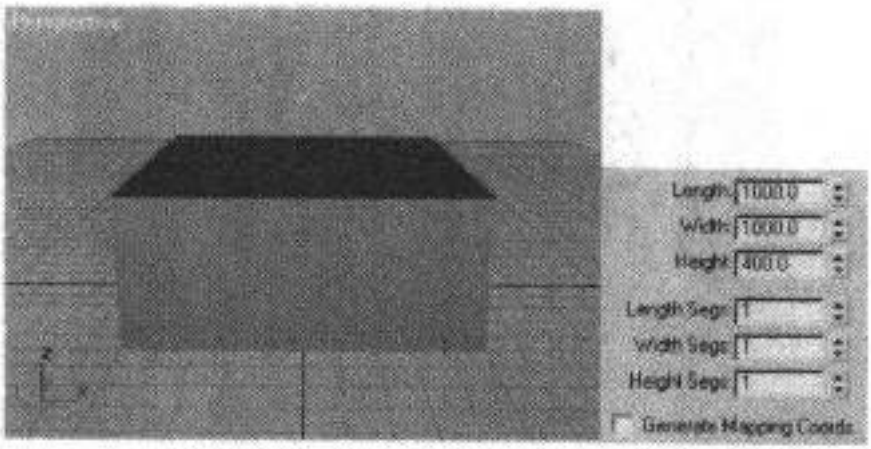


图 A1-14 创建长方体结构化面

第二步：反转法向量

现在应该反转长方体各个面的法向量，使它的内部面是可见的，而外部面不可见（见图 A1-15（彩页中也有））。当然，“-”应该被加到长方体的名字（在本例中是 structure）前面，这样后面的导出插件就能够识别出它是层次的结构化面。

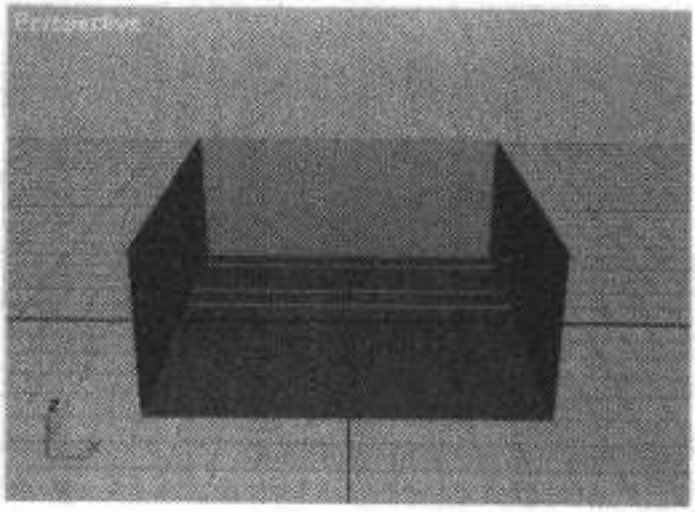


图 A1-15 反转法向量

第三步：贴图

在这一步中，我们将给长方体贴上纹理，使它更好看一些（见图 A1-16）。

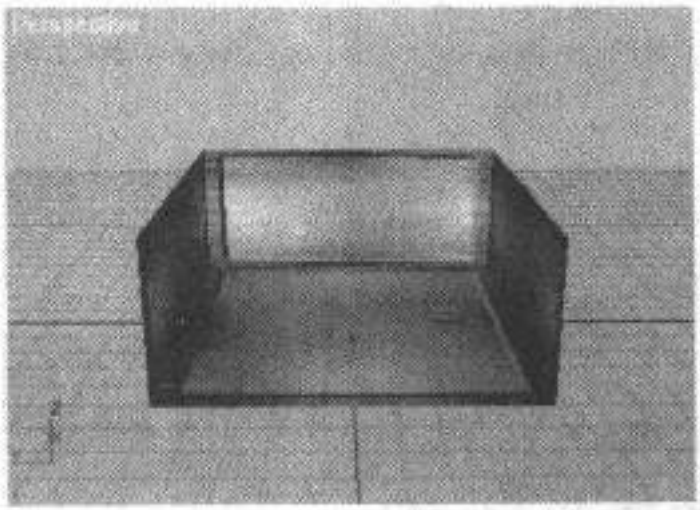


图 A1-16　添加纹理

第四步：创建曲面

Fly3D 曲面构造插件在 Create/Geometry 菜单 Parallelo 类型中。

点击 flycurve 按钮，你将看到有多种曲面类型可用，比如圆柱面、圆锥面、球面、球冠等。还有一些可供使用的选项，比如尺寸、曲面的段数。

首先，创建一个圆柱（4×1 段）放在房间的中间，尺寸设置为 50×50×200（如图A1-17所示）。

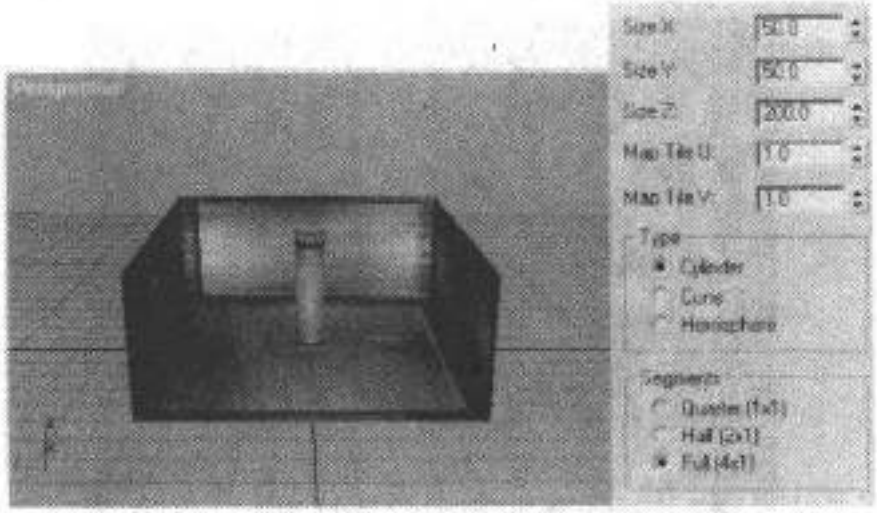
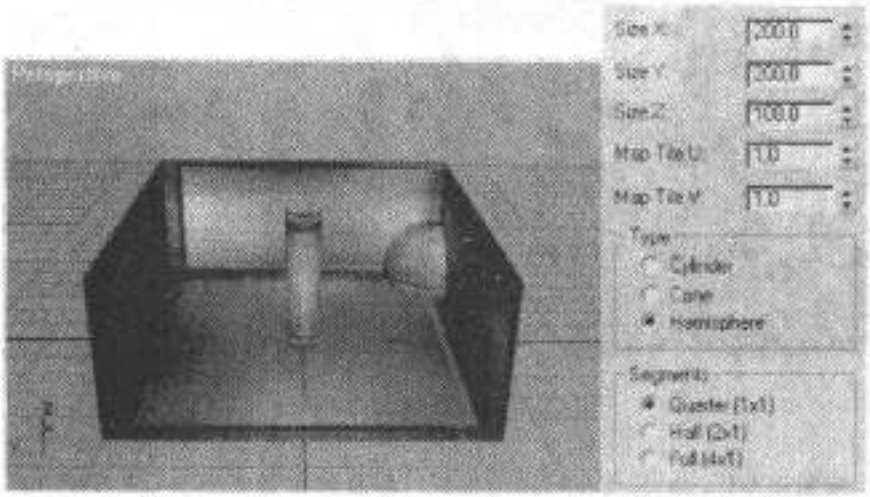


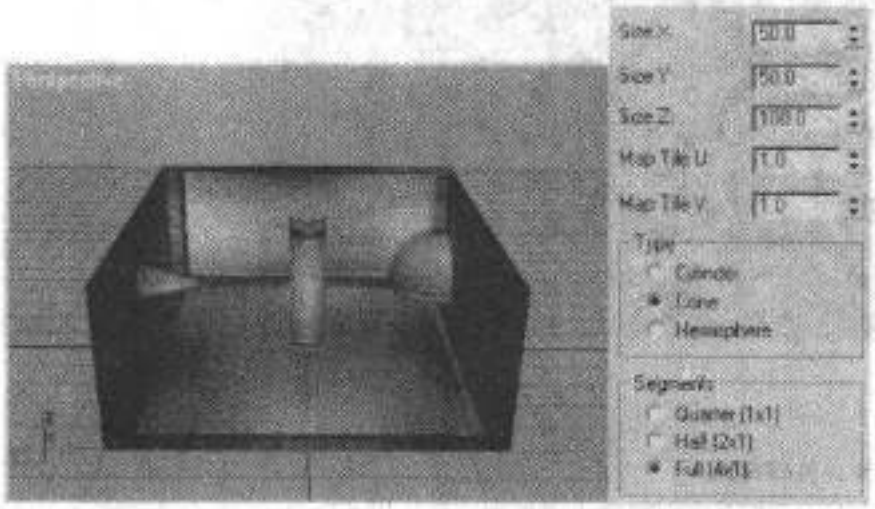
图 A1-17　创建圆柱

现在，创建1/4球体，把它放在房间的任意位置。要创建1/4球体，我们在Type栏中选择Hemisphere，然后在Segments栏中选择Quarter（1×1）。它的大小为200×200×100，如图A1-18所示。



图A1-18 创建1/4球体

这次，我们创建另一种曲面——圆锥。在Type栏中选择Cone选项，在Segments栏中选择Full（4×1），大小设置为50×50×100，把它放置在靠墙的位置，如图A1-19所示。



图A1-19 创建圆锥

下一步，创建一个带底面的半圆柱。首先，创建一个半圆柱的主体，大小为200×200×20（如图A1-20所示）。

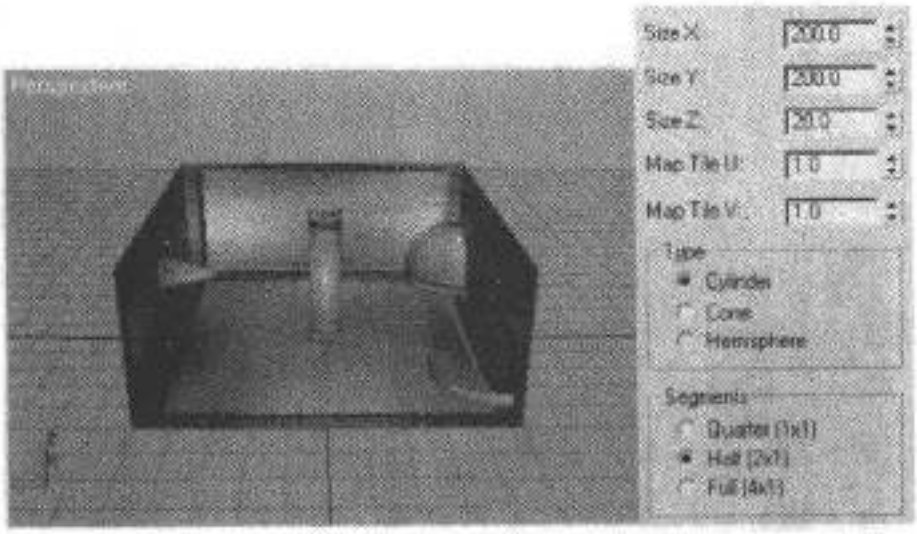


图 A1-20　创建带底面的半圆柱

现在，复制该圆柱，把它放在同样的位置，但是这次要选中 Part 栏中的 Cap Up 选项，为圆柱生成上底面，如图 A1-21 所示。

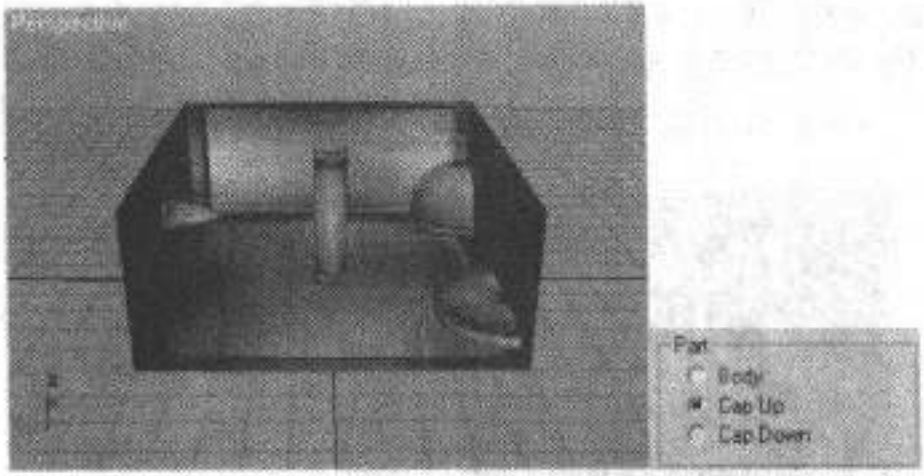


图 A1-21　创建带底面的半圆柱

第五步：编辑曲面

在这一步，我们再次创建一个圆柱面，然后用 Edit Mesh 修改器来移动曲面上的顶点。首先，创建一个完整（Full（4×1））的圆柱体，在 Part 栏中选中 Body。

然后，应用 Edit Mesh 修改器来编辑圆柱曲面的顶点。注意，一旦使用了这个修改器，那么圆柱面就变成不可见的了，取而代之的是曲面的控制点。我们移动的实际是曲面的控制点，但在 Fly3D 中看见的仍然是曲面（如图 A1-22 所示）。

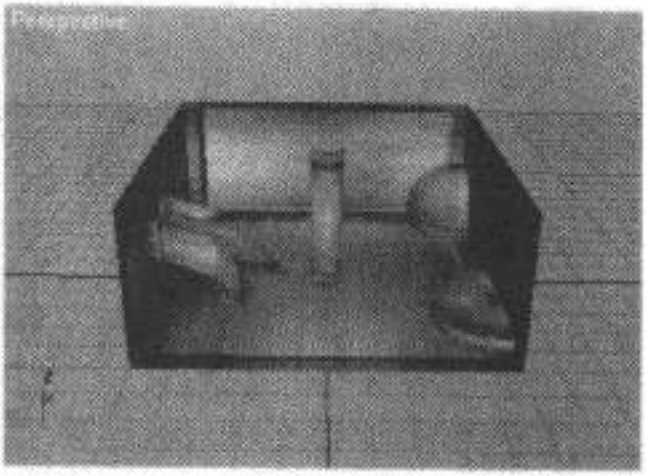


图 A1-22 创建凹柱曲面（连接六面的和靠近视点的墙面）

我们创建的最后一个曲面也是圆柱面（实际上是 1/4 圆柱面），但是这次将使用 Invert Normals选项来产生一个圆弧形的墙角。这个 1/4 圆柱的大小是 200 × 200 × 200（见图 A1-23）。

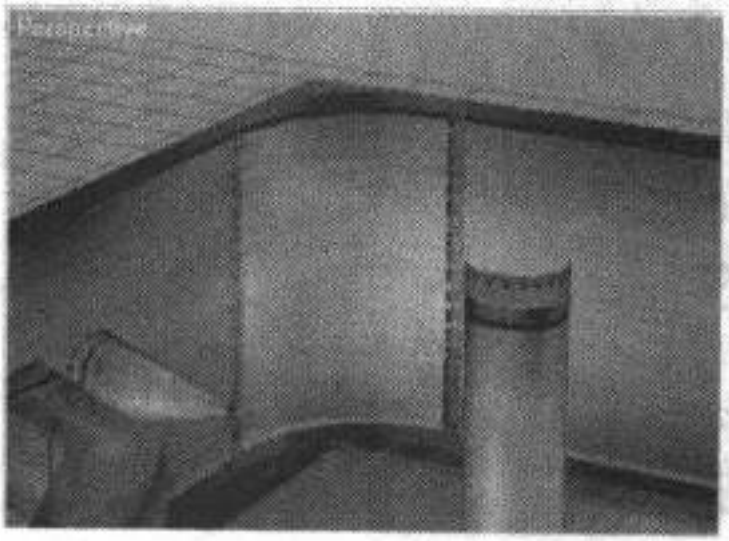


图 A1-23 创建 1/4 圆柱

第六步：光照、添加实体、导出

在这一步，我们把光照和 birthpad 添加到层次中（见图 A1-24（彩页中也有））。

然后把层次导出为 Fly3D 文件格式（.fmp），使用 flyBuilder 来生成整个层次，运行引擎的一个前端来观察整个层次的效果。在图 A1-25（彩页中也有）中，是这个层次在 Fly3D 中的一些屏幕截图。

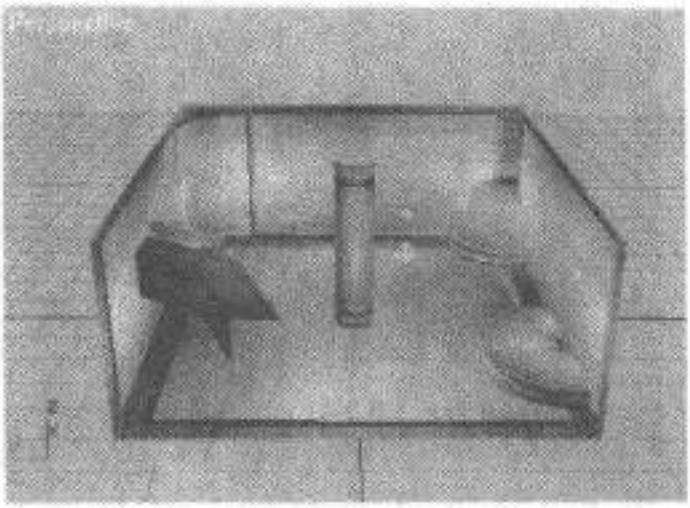


图 A1-24　添加光照

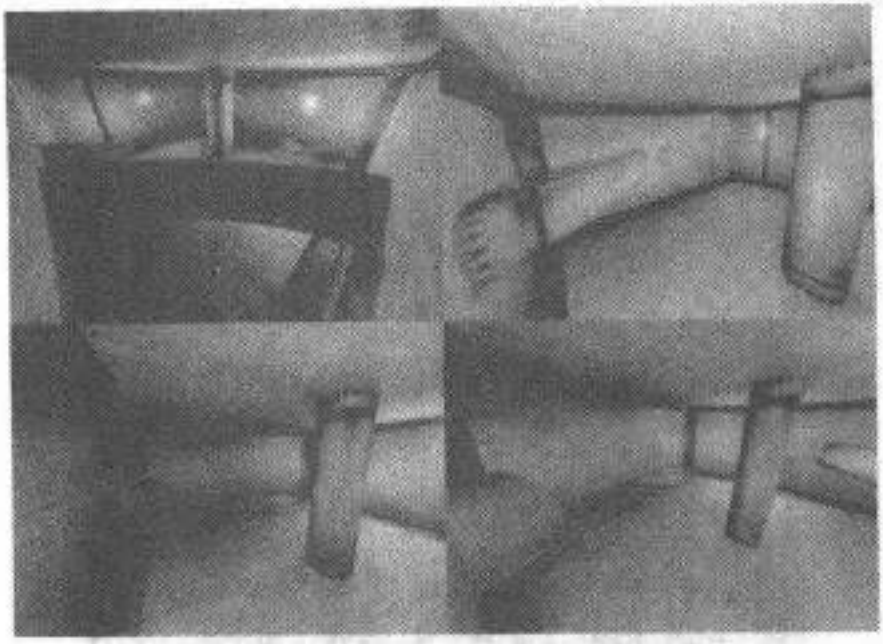


图 A1-25　层次的屏幕截图

附录 1.2　辐射度理论基础

1984 年，康奈尔大学的研究者们在热辐射传导原理的基础上，发明了辐射度方法[GORA84]，也就是现在我们知道的经典辐射度算法。它模拟了在“漫反射表面”之间光线的相互作用。这样也就意味着，它只能用于渲染完全由“漫反射表面”组成的场景。

为了完成整个场景的渲染，场景中的每个表面被分割成一个个的小单元，称为“面片”(patch)，并在每个面片所吸收的光线能量的基础上建立了一组方程。在场景中，每个面片都反射来自其他面片的光线。如果它是光源的话，它也会自己发射光线——光源跟其他的面片是一样的，除了它们有非零的自发射量。面片与面片之间的相互作用跟它们之间的几何位置有很大的相关性，比如距离、相互之间的朝向等。两个距离很近的平行的面片会有比较大的相互作用。如果我们为环境中的每个面片计算它与其他所有面片之间的相互作用，最终可能会找到一个平衡的状态，也就是我们需要的光照结果。

康奈尔大学研究小组的一个主要贡献是发明了一种计算两个面片之间几何关系的有效方法——半立方体（hemi-cube）算法。实际上，在 20 世纪 80 年代大多数对辐射度方法的改进和创新都是出自这个小组。

这个算法的代价是 O（N^2），这里 N 是整个环境被分割成的面片的数目。要降低处理的成本，可以把面片分割得大一些，并且假设在一个面片上的光线强度是不变的。但是，这样又可能产生质量上的问题——如果光照的不连续与面片的边不一致，那么就会产生小的瑕疵。这个面片的大小的限制就是在实践中这个算法只能用于计算漫反射相互作用的原因。在漫反射中，光线总是在一个表面上自然、缓慢地变化。把镜面反射添加到辐射度方法中成本是很高昂的，现在仍然有许多研究关注于这个问题。因此，我们会遇到奇怪的情形：两个主要的相互作用方法——光线跟踪和辐射度——是相互排斥的，因为它们计算的自然现象是不同的。光线跟踪不能计算漫反射相互作用，而辐射度方法又不能与镜面反射结合在一起。虽然如此，辐射度方法依然生成了一些在计算机图形学上最真实的图像。

辐射度方法不需要任何改进就可以处理阴影。几何物体的阴影的计算，可以是光线跟踪算法的一部分，或者是某个反射模型绘制器的一个算法。但是，更准确地说，阴影的强度是漫反射相互作用的一部分，因而只能被其他算法近似地计算。而辐射度方法却能轻而易举地解决阴影的计算。它像计算其他光线强度一样，不需任何特殊处理即可得到阴影。惟一的问题是，面片的大小可能降低描绘阴影的边界的精度。在阴影的边界区域，漫反射光线的强度的变化率比较高，普通面片的大小就可能在边界上产生明显的锯齿现象。

辐射度方法是一种物体空间算法，用于解决在一个环境中离散的点和面片的光线强度计算，而不是为了计算投影到一个平面后图像像素的亮度。

辐射度理论

辐射度方法是一个能量吸收和能量平衡的方法，为计算在一个封闭环境中所有表面的辐射度提供了一个解决方法。输入到系统的能量是来自那些有非零自发射量的表面。实际上，在这个算法中一个光源跟其他表面没有区别，除了它拥有一个非零的初始辐射度。这个方法是建立在所有的表面都是完全的漫反射和理想的 Lambertian 表面的基础上。

辐射度，B 为在单位时间单位面积内离开一个面片的能量，它是自身发射的和反射的能量的总和：

$$B_i \mathrm{d}A_i = E_i \mathrm{d}A_i + \rho_i \int_j B_j F_{ji} \mathrm{d}A_j$$

为了说明这个等式，我们假设有一个面片 i：

辐射度 × 面积 = 自身发射的能量 + 反射的能量

E_i 是一个面片自身发射的能量。反射的能量是把入射能量乘以反射率 ρ_i 得到的。入射能量是环境中的其他面片辐射到面片 i 的能量，也就是我们对 $B_jF_{ji}\mathrm{d}A_j$ 对于所有的 j ($j \neq i$) 求积分。$B_jF_{ji}\mathrm{d}A_j$ 表示从面片 j 传递到面片 i 的能量。F_{ji} 是一个常量，称为形状因子，它表示面片 j 和面片 i 之间的关系。

我们利用面片 i 和面片 j 之间关于形状因子的等式：

$$F_{ij}A_i = F_{ji}A_j$$

然后除以 $\mathrm{d}A_i$ 得到：

$$B_i = E_i + \rho_i \int_j B_jF_{ij}$$

对于一个离散的环境，我们用求和来代替求积分，辐射度的值表示离散的小面片单位时间辐射出的能量总和，可得：

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_jF_{ij}$$

在一个封闭的环境中，每个面片都有这样一个方程，那么对于整个环境来说，就产生了 n 个这样的方程：

$$\begin{bmatrix} 1-\rho_1F_{11} & -\rho_1F_{12} & \cdots & -\rho_1F_{1n} \\ -\rho_2F_{21} & 1-\rho_2F_{22} & \cdots & -\rho_2F_{2n} \\ \vdots & \vdots & \cdots & \vdots \\ \rho_nF_{n1} & -\rho_nF_{n2} & \cdots & 1-\rho_nF_{nn} \end{bmatrix} \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} E_1 \\ E_2 \\ \vdots \\ E_n \end{bmatrix}$$

每个面片的辐射度 B_i 就是这个方程的解。但是，这里仍然有两个问题。其一，我们需要一种方法来计算形状因子。另外，需要计算视图，把这些面片显示出来。这样我们需要一个线性插值方法，比如 Gouraud 着色或细分图案。当然，这里要求线性插值的面片本身将是可见的。

在上面的方程中，ρ_i 是已知的，F_{ij} 是整个环境中几何结构的一个函数。E_i 一般为 0，只有那些提供了光照的表面，E_i 才是非零的，它们代表了整个系统输入的光照。实际上，反射率跟光的波长是相关的，因此上面的方程应该认为是单一频率的光照的解；一个完整的解应该考虑有多少种颜色，并为所有的颜色解方程得到。应该注意到，在这一阶段，对于一个平面或者一个突起的面 $F_{ii}=0$，即它所辐射的光线没有到达它自身。另外，根据形状因子的定义，形状因子矩阵的每一行的和应该是 1。

既然形状因子是关于系统的几何结构的一个函数，那么它们只需要计算一次。但是，辐射度方法仍然受到了计算形状因子所花费时间的限制。形状因子表示了两个面片 A_i 和 A_j 之间的辐射交换，跟它们之间的相对朝向和距离有关。计算式为：

$$F_{ij} = \frac{\text{离开面片 } A_i \text{ 直接射向 } A_j \text{ 的辐射能量}}{\text{离开面片 } A_i \text{ 的所有辐射能量}}$$

可以用下面的公式表示：

$$F_{ij} = \frac{1}{A_i} \int_{A_i} \int_{A_j} \frac{\cos\phi_i \cos\phi_j}{\pi r^2} \mathrm{d}A_j \mathrm{d}A_i$$

几何结构如图 A1-26 所示。在实际应用环境中，相对于 A_i，A_j 可能完全或部分不可见。积

分式需要乘以一个遮挡因子，该因子是二元的（0 或 1），它表示微分区域 dA_i 是否能看到 dA_j。除了一些特别的形状外，计算这个二重积分是比较困难的。

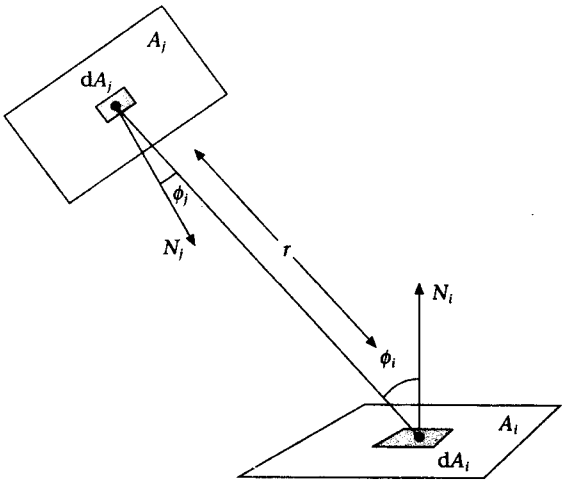


图 A1-26　两个面片 i 和 j 的形状因子几何关系

渐进细化

1988 年，康奈尔大学的研究小组提出了一种叫做“渐进细化”的方法［COHE88］。开发这个方法的最初动因是为了让设计者能够较快地得到解（近似解）。在常规的辐射度矩阵的计算（比如使用 Gauss-Seidel 方法）中，某一行的方程反映了单独一个面片 i 的辐射度：

$$B_i = E_i + \rho_i \sum_{j=1}^{n} B_j F_{ij}$$

这是根据其他所有面片的辐射度值得到的面片 i 辐射度估计值，这个称为“吸收”（gathering）。这个方程的意思是：对于面片 i，逐一访问场景中的所有其他面片，根据形状因子，从面片 j 传递适当量的光线给面片 i。该算法以行为单位，逐行处理。当把整个矩阵都处理了一遍，这时才更新所有的解（在 Gauss-Seidel 方法中，一旦计算出了一个新的值，就立即把新的值应用到后面的计算中）。如果我们一边求解一边把场景绘制出来，那么将看到场景中的面片是按照该面片对应的方程在方程组中的行数的次序逐渐变亮的。

渐进细化方法的想法是每次迭代更新所有的面片。这个被称为“发射（shooting）”，即每个面片 i 的能量又传递到其他所有面片。这两种处理方法的差别在图 A1-27a 和 b 中用图表的方式展示出来。下面介绍如何对原方法进行调整以实现渐进细化方法。

单独一项表示了面片 i 对于面片 j 的辐射度的贡献：

$$B_j \text{ due to } B_i = \rho_j B_i F_{ji}$$

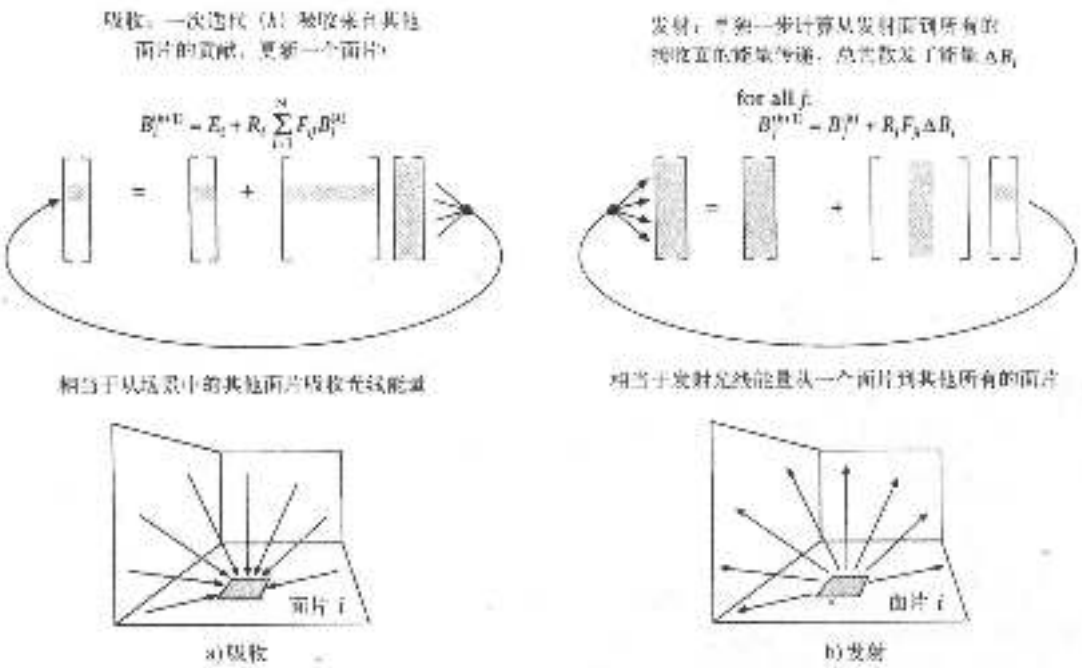
利用它们之间关于形状因子的等式，上式可以变为：

$$B_j \text{ due to } B_i = \rho_j B_i F_{ij} A_i / A_j$$

这个关系式对所有的面片 j 都是成立的。它可以用于决定面片 i 对环境中每个面片 j 的贡献。单个辐射源（面片 i）向环境中发出光线，每一个面片 j 的辐射度被同时更新。因此，只要计算出第一行的形状因子，就可以得到整个场景中的第一个近似值。这样就消除了很高的启动和预算开销。

这个过程一直重复，直到收敛。开始时，所有的辐射度设置成 0 或者它们的非零自发射值。在这个计算迭代过程中，每一步所有面片的辐射度都被更新。随着求解的进行，面片 i 辐射度的估计值会越来越精确。对于一次迭代，环境中已经包含了前一次估计值 B_j 的贡献。而现在，这个被称作“未发射”的辐射度，也就是前一次估计值与当前估计值的差额，将在这次迭代中被全部加入到环境中。

如果我们一边计算一边绘制出场景的话，那么整个场景开始时是黑的，然后随着每个面片辐射度的增加会逐渐地变亮。根据每个面片可能的辐射量，我们可以对面片从大到小进行



a) 吸收　　b) 发射

图 A1.27　辐射度求解策略

排序，这样能够加快该过程的“视觉收敛”。也就是说，那些发光的面片或者光源应该被最先处理。这样可以比较快地得到较好的光照。接下来处理的是那些从光源等处接收的光最多的块。使用这种排序的方法，整个求解的过程就与环境中光线的传播很相似。尽管这样产生了一个更好的、更形象的顺序，但解仍然是逐步从暗的场景变到亮的场景。要解决这个影响，可以向场景中添加环境光项。这个项仅仅用来增强显示，并不是解的一部分。这个环境光项的值是建立在当前环境中所有面片的辐射度估计值的基础上。随着求解的进行，场景的光照逐渐收敛，而环境光的值逐渐降低。

在这个算法中，对于每次迭代要完成四个主要的步骤。它们是：

1）找出拥有最大辐射度或者发射能量的面片。

2）计算形状因子矩阵的一列，即环境中从这一面片到其他每片的形状因子。

3）更新每一个接收面片的辐射度。

4）根据第 3 步计算出的辐射度值与前面的辐射度值的差，减少临时的环境光项。

第2章　高级游戏系统剖析Ⅱ：实时处理

在前一章中，我们了解了游戏系统必需的离线建模处理过程。在这一章中，我们进一步考虑游戏系统所必需的实时处理。下面列出的主题，我们认为是一个能处理高度复杂游戏的现代游戏引擎采用的普遍实时方法。关于渲染的高级方面、动画和其他一些方法将在第4～11章涉及。根据应用程序的要求，在游戏系统中可能会也可能不会用到这些方法。本章更偏重于讲解在这些方法中采用的优化处理；关于这些主题的更多材料可以在［WATT01］中找到。这一章的所有处理都使用了BSP树场景管理。

这一章涉及的主题是：

- **视见约束体的选取**　是一个实时的优化，能够进一步提高场景模型的优化，使我们可以只选择那些在视见约束体中的BSP叶节点。
- **照相机控制**　是一个被人们忽视的问题，但它对于一个动作游戏的感觉和质量有很大的贡献。
- **碰撞检测和反弹**　快速而准确的碰撞检测和反弹是非常重要的，它是游戏的感觉和质量的主要贡献者。
- **路径规划**　我们介绍一种简单的方法，它利用了前一章介绍的伪入口。通过使用A*算法来提高它的性能之后，这个实时方法能够作为基础甚至高级的人工智能程序。

2.1　视见和BSP

在任何游戏中，主要的循环是选择要渲染的面。因此，一个好的视见约束体的选取对于高度复杂的场景（其可见面可能只是总数的一小部分）是非常重要的。下面是一个基本方法的处理步骤：

1）根据照相机的位置、方向、视线的角度、视域的长宽比例和远裁剪面，可以生成视见约束体的面。（这里只使用了5个面，因为我们忽略了近裁剪面的裁剪作用。）

2）遍历BSP，选择所有在视见约束体中的叶节点。

3）在BSP遍历过程中，使用PVS来降低被选择的叶节点数目。

4）为了进一步地裁剪，对于被选择叶节点的每一个面，我们裁剪那些接触到视见约束体包围盒的部分。在整个过程中，使用AABB作为包围盒。

5）所有剩下的面都会被渲染出来。

2.1.1　生成视见约束体的面

我们可以很容易地计算出视见约束体的顶点。再利用这些顶点，就可以生成视见约束体的面。下面定义的类将会被用到：

```
class FLY_ENGINE_API flyFrustum
{
  public:
    flyVector verts[5];
```

```
    flyVector planes[5];
    int bboxindx[8][3];

  inline int clip_bbox(const flyBoundBox& bbox) const;
  void build(const flyVector& pos,const flyVector& X,
      const flyVector& Y,const flyVector& Z);
};
```

通过计算视见约束体的顶点我们可以生成面，然后利用三对顶点作为轴就可以计算出这些面的法向量。

```
void flyFrustum::build(const flyVector& pos,const flyVector& X,const
flyVector& Y,const flyVector& Z)
{
  float farplane=flyRender::s_farplane;
  float
disty=farplane*(float)tan(flyRender::s_camangle*0.5f*FLY_PIOVER180);
  float distx=disty*flyRender::s_aspect;

  verts[0]=pos;

  verts[1].x = pos.x - farplane*Z.x + distx*X.x + disty*Y.x;
  verts[1].y = pos.y - farplane*Z.y + distx*X.y + disty*Y.y;
  verts[1].z = pos.z - farplane*Z.z + distx*X.z + disty*Y.z;

  verts[2].x = pos.x - farplane*Z.x + distx*X.x - disty*Y.x;
  verts[2].y = pos.y - farplane*Z.y + distx*X.y - disty*Y.y;
  verts[2].z = pos.z - farplane*Z.z + distx*X.z - disty*Y.z;

  verts[3].x = pos.x - farplane*Z.x - distx*X.x - disty*Y.x;
  verts[3].y = pos.y - farplane*Z.y - distx*X.y - disty*Y.y;
  verts[3].z = pos.z - farplane*Z.z - distx*X.z - disty*Y.z;

  verts[4].x = pos.x - farplane*Z.x - distx*X.x + disty*Y.x;
  verts[4].y = pos.y - farplane*Z.y - distx*X.y + disty*Y.y;
  verts[4].z = pos.z - farplane*Z.z - distx*X.z + disty*Y.z;

  planes[0].cross(verts[2]-verts[1],verts[0]-verts[1]);
  planes[0].normalize();
  planes[0].w=FLY_VECDOT(verts[0],planes[0]);

  planes[1].cross(verts[3]-verts[2],verts[0]-verts[2]);
  planes[1].normalize();
  planes[1].w=FLY_VECDOT(verts[0],planes[1]);

  planes[2].cross(verts[4]-verts[3],verts[0]-verts[3]);
  planes[2].normalize();
  planes[2].w=FLY_VECDOT(verts[0],planes[2]);

  planes[3].cross(verts[1]-verts[4],verts[0]-verts[4]);
  planes[3].normalize();
  planes[3].w=FLY_VECDOT(verts[0],planes[3]);

  planes[4].cross(verts[3]-verts[1],verts[2]-verts[1]);
  planes[4].normalize();
  planes[4].w=FLY_VECDOT(verts[1],planes[4]);

  static int table[8][3]=
    { {4,5,6},{4,5,2},{4,1,6},{4,1,2},
    {0,5,6},{0,5,2},{0,1,6},{0,1,2} };
  int i,j;
```

```
    for( i=0;i<5;i++ )
    {
      j =  (FLY_FPSIGNBIT(planes[i].x)>>29)|
           (FLY_FPSIGNBIT(planes[i].y)>>30)|
           (FLY_FPSIGNBIT(planes[i].z)>>31);
      bboxindx[i][0]=table[j][0];
      bboxindx[i][1]=table[j][1];
      bboxindx[i][2]=table[j][2];
    }
  }
```

当物体或场景的 AABB 与视见约束体相交时可以做一个重要的优化。每个 AABB 需要测试一下，检查它是否与视见约束体的面相交。传统的处理这个问题的方法是，对于视见约束体的每个面，要检查它到包围盒每个顶点的距离。如果没有顶点有一个正的距离值，那么这个盒子在面的后面而且没有相交。如果任何一个顶点有正的距离值，那么我们继续处理下一个面。如果所有的面都处理后，只要有一个点在正的一边，那么这个盒子就要被视见约束体裁剪。代码如下：

```
int flyFrustum::clip_bbox(const flyVector& min,const flyVector& max)
const
{
  const flyVector *v=frustum_clip;
  for( int i=0;i<5;i++,v++ )
  {
    if (v->x*min.x+v->y*min.y+v->z*min.z-v->w>0)
      continue;
    if (v->x*max.x+v->y*max.y+v->z*max.z-v->w>0)
      continue;
    if (v->x*max.x+v->y*min.y+v->z*min.z-v->w>0)
      continue;
    if (v->x*min.x+v->y*max.y+v->z*min.z-v->w>0)
      continue;
    if (v->x*min.x+v->y*min.y+v->z*max.z-v->w>0)
      continue;
    if (v->x*max.x+v->y*max.y+v->z*min.z-v->w>0)
      continue;
    if (v->x*min.x+v->y*max.y+v->z*max.z-v->w>0)
      continue;
    if (v->x*max.x+v->y*min.y+v->z*max.z-v->w>0)
      continue;

    return 0;
  }

  return 1;
}
```

我们可以把测试一个面的过程简化为“一票否决”，因为所有的包围盒是 AABB。当每个视见约束体面创建时，我们可以为每个需要跟这个面进行测试的 AABB 定义一个顶点。如果该点到这个面的距离是负的，那么所有的顶点都一定是负的。为了确定某个 AABB 相对于一个视见约束体面的测试顶点，我们利用该面的法向量找到包含该法向量的八叉体（octant）。在这样做之前，首先为每个视见约束体面预先计算一个顶点索引（vertex index）。顶点索引是一个比特域，它表示面法向量每个分量 x，y，z 的符号（见图 2-1）。例如：

法向量（+，+，+）可以编码成比特 000 = 0 十进制数

法向量（-，-，-）可以编码成比特 111 = 7 十进制数

法向量（+，-，+）可以编码成比特 010 = 2 十进制数

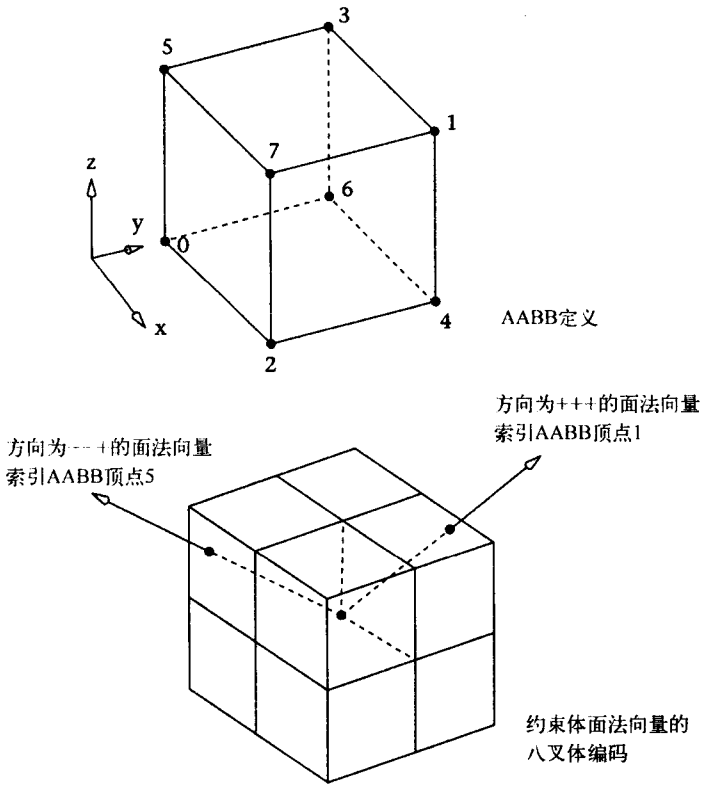


图 2-1　把面（法向量）和要测试的 AABB 顶点结合起来

下面是为每个视见约束体面构造比特域的代码

```
for( int i=0;i<5;i++ )
  bbox_verts[i]=
     (FLY_FPSIGNBIT(planes[i].x)>>29)|
     (FLY_FPSIGNBIT(planes[i].y)>>30)|
     (FLY_FPSIGNBIT(planes[i].z)>>31);
```

FLY_FPSIGNBIT 定义语句像访问整数一样来访问浮点数。它并不把浮点数转换成整数，而是像整数一样使用浮点数的比特位。

```
#define FLY_FPBITS(fp)   (*(int *)&(fp))
```

因此对应关系是：

面法向量编码	AABB 要测试的顶点
0	1
1	4
2	7
3	2
4	3
5	6
6	5
7	0

下面的代码使用上面的图表，对每个面只测试一个顶点。但是 switch 语句不是很有效，我们不得不根据 AABB 的最大最小定义构造一个临时的顶点。

```
int flyFrustum::clip_bbox(const flyVector& min,const flyVector& max)
const
{
  flyVector v;
  for( int i=0;i<5;i++ )
  {
    switch(bbox_verts[i])
    {
    case 0: v=max; break;
    case 1: v.vec(max.x,max.y,min.z); break;
    case 2: v.vec(max.x,min.y,max.z); break;
    case 3: v.vec(max.x,min.y,min.z); break;
    case 4: v.vec(min.x,max.y,max.z); break;
    case 5: v.vec(min.x,max.y,min.z); break;
    case 6: v.vec(min.x,min.y,max.z); break;
    case 7: v=min; break;
    }

    if (FLY_VECDOT(v,planes[i])-planes[i].w<0)
      return 0;
  }
  return 1;
}
```

更好的方法是使用下面的表格：

面法向量编码	AABB 要测试的顶点
0	4, 5, 6
1	4, 5, 2
2	4, 1, 6
3	4, 1, 2
4	0, 5, 6
5	0, 5, 2
6	0, 1, 6
7	0, 1, 2

Min	0: min.x	1: min.y	2: min.z	3: not used
Max	4: max.x	5: max.y	6: max.z	7: not used

第一个表格中，每组选择最大最小顶点的适当组合来定义需要的顶点。最后的代码如下：

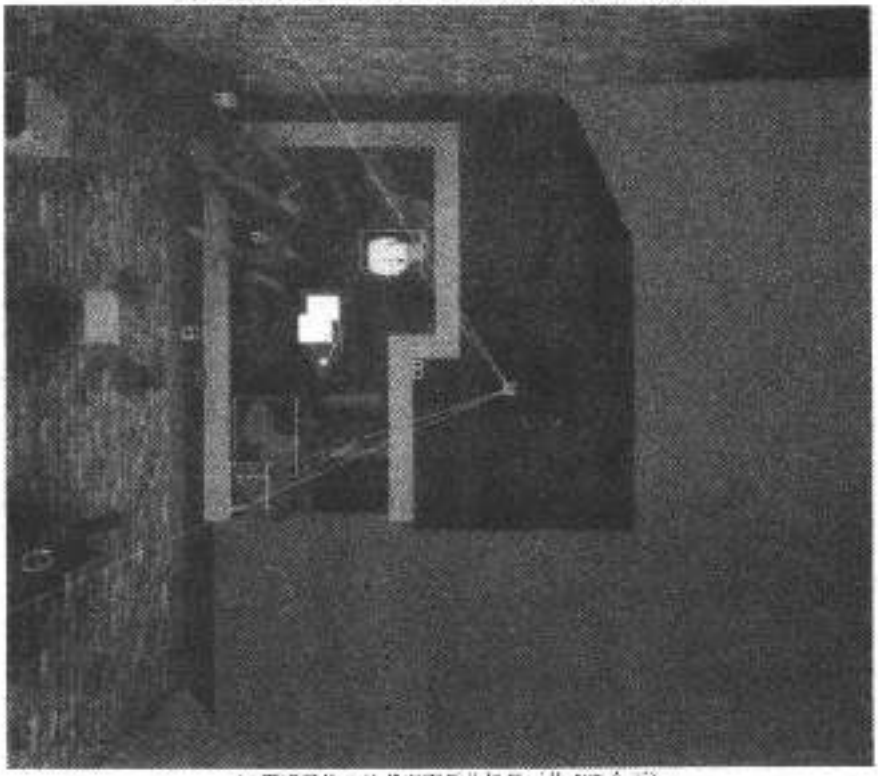
```
int flyFrustum::clip_bbox(const flyBoundBox& bbox) const
{
  float *f=(float *)&bbox.min.x;

  for( int i=0;i<5;i++ )
    if (planes[i].x*f[bboxindx[i][0]]+
      planes[i].y*f[bboxindx[i][1]]+
      planes[i].z*f[bboxindx[i][2]]-planes[i].w<0)
      return 0;
  return 1;
}
```

图 2-2（彩页中也有）演示了这个策略对于一个中等复杂的层次的有效性。这个例子是

直接从操作视点上的一个视点渲染的。这个层次的一些统计如下：



a) 用视见约束体截面 BSP 叶节点后的场景（共 1453 个面）



b) 用视见约束体截面后的场景（共 587 个面）

图 2-2　Padgarden 层和视见约束体

层次中 AABB 面的数目	5204
被视见约束体裁剪后 BSP 叶节点数	1453
被视见约束体裁剪的 AABB 的数目	587

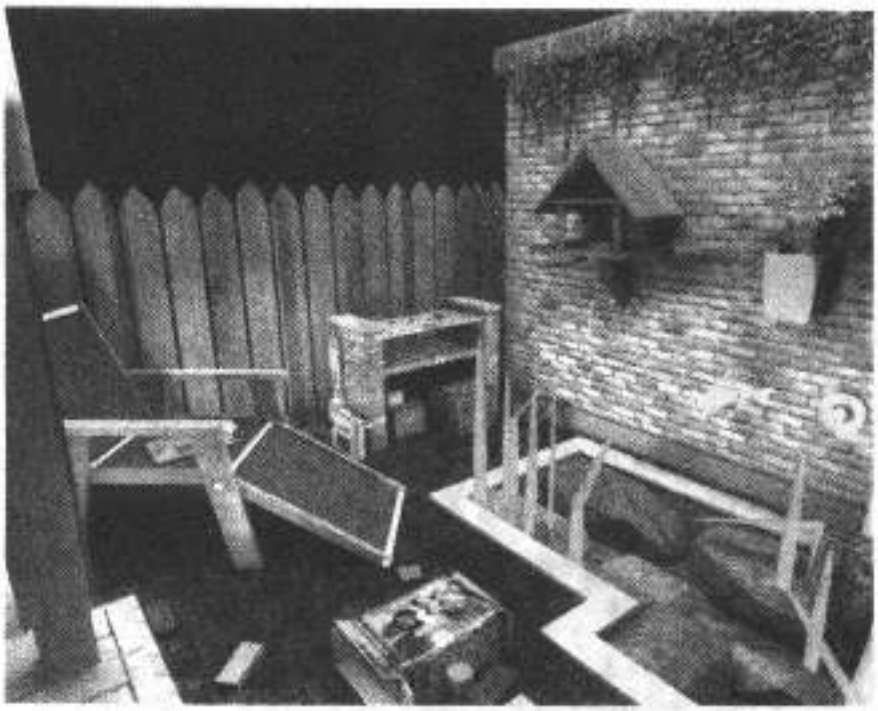
2.1.2 远近裁剪面和视见约束体

这里还有一些关于视见约束体参数的问题，特别要注意远近裁剪面距离的设置。

许多程序出现的一个常见问题是当视点离墙很近时，我们可以看到墙的另一边。这是由于视见约束体所使用的粒子碰撞检测的缘故（视点本身是一个粒子）。当视点到墙的距离小于近裁剪面的距离时，墙另一边的物体就会被渲染出来。另一个因素是 Z 缓冲的精度：近裁剪面的距离足够大更有利于去掉深度缓冲中的错误。

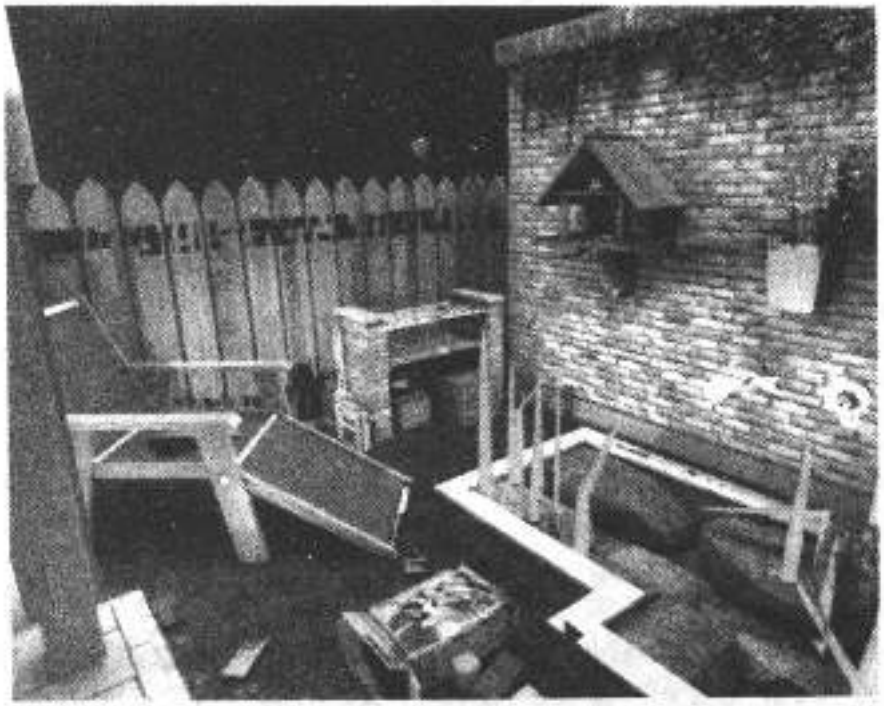
我们总是趋向于把视点包含在一个包围盒中，要么是一个玩家的包围盒（第一人称游戏），要么是一个照相机的包围盒（第三人称游戏）。通过这个方法，我们保证视点离所有墙的距离有一个最小值。这就意味着我们有一个固定的（大的）近裁剪面距离，当移动到离墙很近时，不会看到墙对面的物体。一个比较好的近裁剪面距离设置是包含视点的包围盒的“半径”的 0.5 倍。“半径”指的是盒的中心到最近的面的距离。

现在考虑远裁剪面。这应该动态设置成当前帧中我们要渲染的最远距离（保持最高的 Z 缓冲精度）。图 2-3（彩页中也有）演示了远裁剪面设置选项的作用。

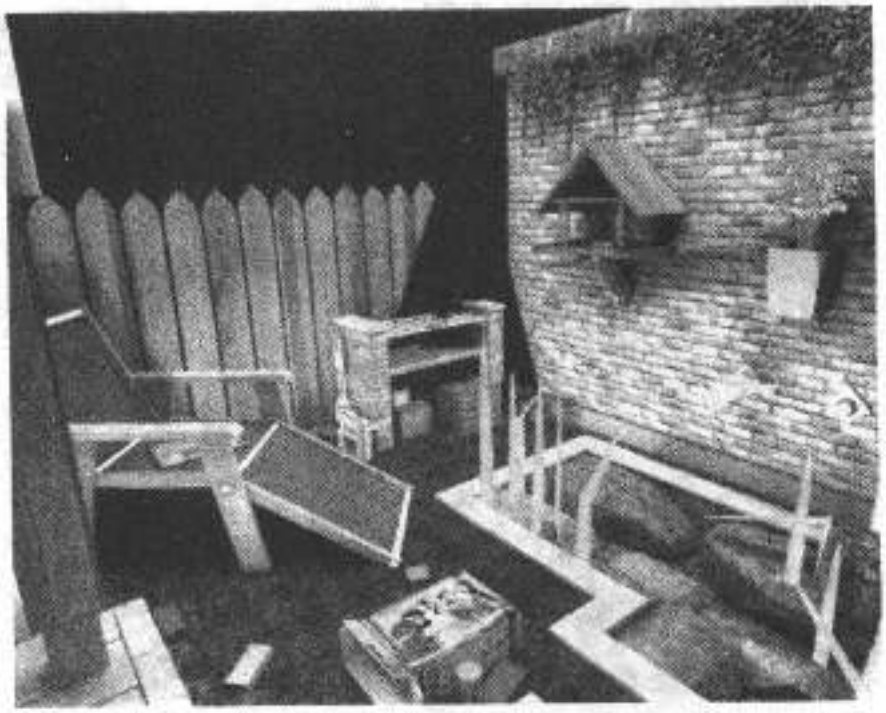


a）Polygarden 层次三角形远近裁剪面设置

图 2-3

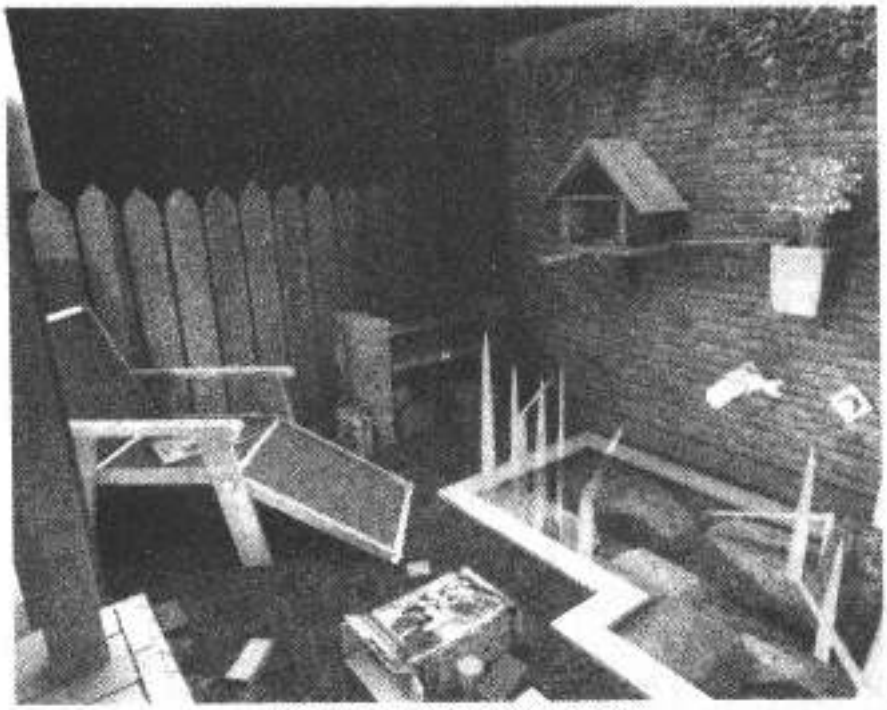


b) 远裁剪面太近，导致Z缓冲的精度不够



c) 远裁剪面太近，可以明显看出远裁剪面

图 2-3　（续）



d) 使用雾化效果降低过近的近裁剪面的影响

图 2-3　（续）

2.2　照相机控制

在游戏中最普通的照相机是第一人称照相机。它只是简单继承了游戏中玩家与角色的交互驱动的运动方式，因此很容易实现。另外一种照相机是脚本照相机（scripted camera），此时照相机成了一个动画物体。这方面的技术将在第 8 章介绍。这里我们讨论的是第三人称照相机。第三人称照相机一般放在游戏角色的上面或者后面。当考虑使用一个第三人称照相机时，就会出现更多的可能性（和困难）。我们先列出这样一个照相机的限制和要求：

- 这个照相机到一面墙的距离永远不要小于近裁剪面。
- 永远不要跑到层次的外面。
- 它应该平滑地移动和旋转。
- 照相机应该根据角色的运动采用不同的运动方式，比如根据平滑角色运动的不连续性。
- 照相机也要使用包围盒进行碰撞检测处理（就像其他动态的物体一样）。
- 当角色在墙的附近并且其背靠近墙时，照相机应该能够进入角色体内。

通过把它分成下面几个部分，我们可以设计一个具有如下行为的照相机：

1）根据角色的位置找出照相机的目标位置。

2）检查目标位置的有效性。

3）使用照相机的包围盒进行碰撞检测，计算照相机需要的平移量，对其做适当的平移。

4）同时，计算照相机需要做的旋转，使照相机面向新的方向。

对于当前帧，我们根据角色移动到的位置计算出所需的照相机目标位置。可以按照下面

的步骤来做：

1）计算目标位置。

2）检查目标位置的有效性（它可能在墙的错误一边）。这个可以用一条从角色位置点到照相机目标位置点的光线的相交性来判断（见图 2-4）。

3）如果目标位置是无效的，那么我们需要把照相机位置沿着这条光线往回移动，直到它的位置在墙的正确的一边。也就是说，移动照相机位置直到它到墙的距离为：

$$\sqrt{3}d/2$$

这里，d 是照相机包围盒的大小；近裁剪面的距离 < $d/2$。

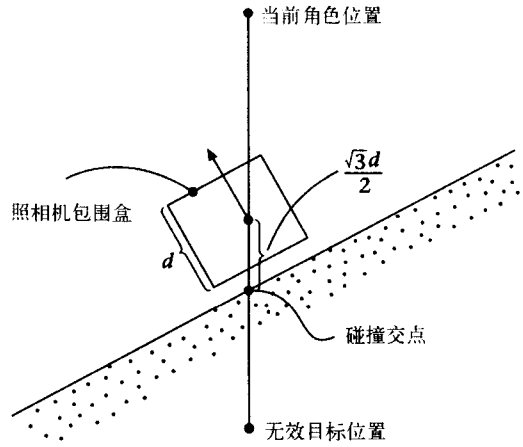


图 2-4　把第三人称照相机放置在一个无效的目标位置

现在我们有了一个有效的照相机目标位置，然后来考虑照相机怎么运动到它的新位置上。对于追踪照相机来说，这一步必须要认真考虑。基础的方法会产生不理想的效果，照相机加速移向角色然后又移动回来，就像照相机在振荡一样。

最直接的方法是分别计算照相机的平移运动和旋转运动。在计算照相机的运动时，我们还要调用碰撞检测例程，检查照相机的包围盒在它的目标位置上是否与场景发生碰撞。（准确地说，这与下一节描述的角色的包围盒的碰撞是一样的。）如果照相机不能移动到它的目标位置上，那么碰撞检测和反弹例程就有责任提供一个新的目标位置。

对于平移运动，我们可以定义一个最大速度，把照相机放在要求的目标位置上，或者放在到目标位置的某个中间位置上。这取决于最大速度与帧之间的时间间隔的乘积是大于还是小于要移动的距离。这个方法的伪代码如下：

```
vec = destination - character_position
normalise (vec)
if len > max_velocity* dt then len = max_velocity* dt
box_collision (character_position, character_position + vec.len)
```

这个方法有一个缺点，当最大速度超过角色的速度时，照相机就会“粘”在角色上。相反，如果角色的速度大于最大速度，那么它又会远离照相机。更好的方法是定义一个最大加速度而不限制速度。当照相机到达目标位置时，它就立即停下来。代码如下：

```
// find target position of the camera based on
// dist and height from character position
targetpos=target->pos+dist*target->Z+height*target->Y;

// collision detection to find a valid target position
if (collision_bsp(target->pos,targetpos)!=0)
  targetpos=hitip-(target->Z*(radius*(float)sqrt(3)));

// compute displacement vector for camera
displace=targetpos-pos;

// compute required velocity to cover displacment in this frame
targetvel=displace*(1.0f/dt);
```

```
// for each axis
for(i=0;i<3;i++)
{
   // compute required velocity change
   f=targetvel[i]-vel[i];
   // if current velocity and desired velocity
   // are in same direction (same sign)
   if(targetvel[i]*vel[i]>0)
   {
      // crop with maximum acceleration
      if (f>maxaccel*dt)
         f=maxaccel*dt;
      if (f<-maxaccel*dt)
         f=-maxaccel*dt;
      // change velocity
      vel[i]+=f;
   }
   else
      // braking, infinite acceleration
      vel[i]+=f;
}

// compute new destination position
newpos=pos+vel*dt;
```

对于旋转运动，我们可以定义一个最大角速度。不这样做的话，照相机就会立即与要求的方向对齐，从而造成不连续的运动。这个方法的伪代码跟平移的伪代码很相似，如下：

vec = destination − character_position

normalise(vec)

angle = acos(vec_dot(required_lookdn, vec)

if(angle > max_rot_vel* dt)**then** angle = max_rot_vel* dt

rotate(angle)

旋转指两个单独的方面。一方面，我们需要使旋转照相机面向角色的视线方向，旋转在包含定义旋转角度的两个向量的平面内进行。另一方面，照相机的上下方向还需要与角色的上下方向吻合。跟平移一样，更好的方法是定义一个最大的旋转加速度：

```
// find desired look direction vector
v=pos-target->pos;
v.normalize();

// compute the angular velocities in each axis
float targetrotvel[2];
targetrotvel[0]=acos(vecdot(Y,target->Y))/dt;
targetrotvel[1]=acos(vecdot(Z,v))/dt;

// for each axis
for(i=0;i<2;i++)
{
   // compute required angular velocity change
   f=targetrotvel[i]-rotvel[i];

   // if current angular velocity and desired
   // angular velocity are in same direction
   if(targetrotvel[i]*rotvel[i]>0)
   {
   // crop with maximum angular acceleration
```

```
        if(f>maxrotaccel*dt)
          f=maxrotaccel*dt;
        if(f<-maxrotaccel*dt)
          f=-maxrotaccel*dt;
        // change angular velocity
        rotvel[i]+=f;
      }
      else
        // braking, infinite angular acceleration
        rotvel[i]+=f;
}

// rotate
rotate(Y,target->Y,rotvel[0]*dt);
rotate(Z,v,rotvel[1]*dt);
```

2.3　使用 BSP 的基本碰撞检测和反弹

有效的、准确的碰撞检测是三维游戏中至关重要的一部分。通用的碰撞检测是一个复杂的仍处于研究中的问题（[WATT01]）。在这一节中，我们将探讨使用 BSP 管理的粒子/场景、包围盒/场景的碰撞检测方法。使用 BSP 管理方案的碰撞检测对于粒子、包围盒等简单情形来说是最简单的。但是，时间必须处理得很准确，我们不能容忍任何“丢失”的时间。标准游戏类型能够很好地运行简单碰撞反弹模型，使碰撞被正确地处理。对粒子/场景碰撞的反弹，可以把粒子想像成一个无限小的球。决定它反弹作用的有两个因子，反弹因子决定法线方向上速度的反作用，摩擦因子决定切线方向上速度的反作用。这个简单的模型如图 2-5 所示。

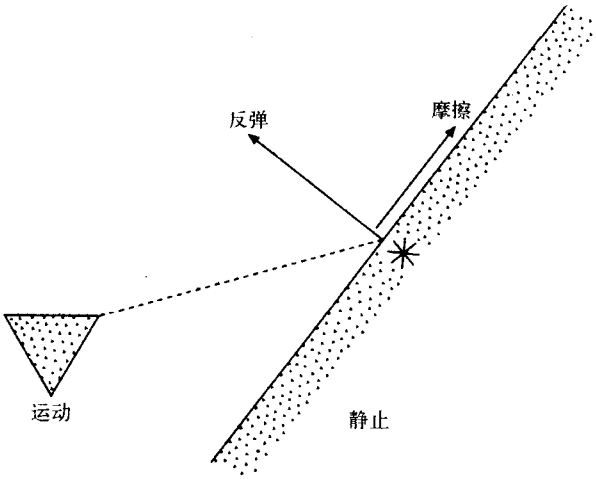


图 2-5　运动物体与面的碰撞

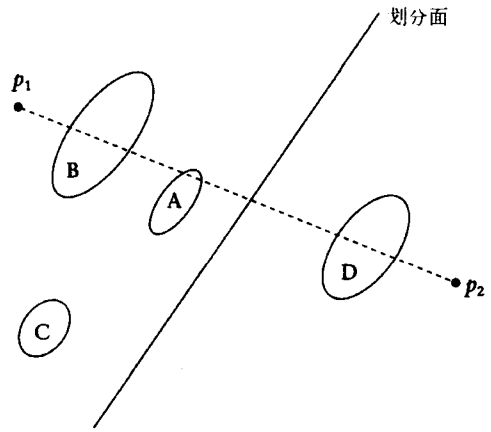


图 2-6　光线相交碰撞检测和 BSP 节点

2.3.1　碰撞和 BSP 遍历

BSP 中，一般的遍历和选择对于碰撞检测可以进行优化。如图2-6所示，这里演示了一个从 p_1 到 p_2 的碰撞检测，p_1 和 p_2 被几个面所隔开。我们从前往后遍历 BSP，检查节点中的物体是否有相交。如果没有相交，继续检查下一个节点。这里所做的优化是当在某个节点

检测到碰撞之后，就可以终止遍历。在这个例子中，碰撞以从 A 到 B 的顺序被找到，而 D 不会被测试。

```
collision_bsp(point1, point2)
{
  push(bsp root node)
while (stack is not empty)
{
  node = pop_node()
  if (node is leaf)
  {
    ray intersect all objects in node
    return closest intersection
  }
  else
  {
    dist1 = distance from point1 to node plane
    dist2 = distance from point2 to node plane
    if (dist1*dist2<0)
    {
      if (dist1>0)
      {
        push(node child 0)
        push(node child 1)
      }
      else
      {
        push(node child 1)
        push(node child 0)
      }
    }
    else
    {
      if (dist1>0)
        push(node child 0)
      else
        push(node child 1)
      }
    }
  }
}
```

有时候我们并不需要求得最近的碰撞，而只需简单地指出一个碰撞发生了，所以当我们找到第一个相交的面后就终止处理。这个对于处理光照是很有用的，例如产生阴影。另外，如果有一个跟踪角色的自引导导弹，我们只需要知道在视线上是否有物体，而不需要确定这个插入的物体是什么。

2.3.2 粒子/场景检测和反弹

现在我们来考虑粒子与静止的物体（如墙）之间的碰撞。最简单的方法是：让光线与包含这个多边形的面相交，然后检查交点是否包含在多边形中。考虑如图 2-7a 的情形，粒子从 p_0 以速度 $\mathbf{v}_0$ 向 p_1 运动：

$$p_1 = p_0 + \mathbf{v}_0^* \mathrm{d}t$$

求得交点 p_x，p_x 离真正的交点有一小段距离。然后我们可以把最终的粒子的位置设为 p_x，计

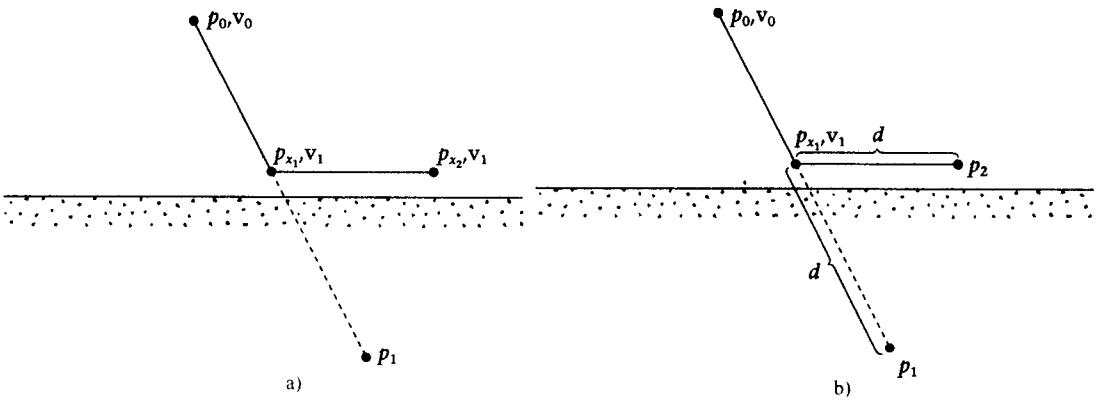


图 2-7　粒子碰撞与“丢失”时间

算出粒子在 p_x 的新速度，如下所示：

$$\mathbf{v}_r = f(\mathbf{v}_0, \mathbf{N}, \text{bump\_factor}, \text{friction\_factor}) \tag{2-1}$$

这里，**N** 是交点的表面法向量，friction_factor 和 bump_factor 在 2.4.4 节中说明。

这个公式的计算可以按以下程序实现：

```
compute_reflection(dir,normal,bump_factor,friction_factor,refdir)
{
   reflectdir = dir + normal*(-2.0f*vec_dot(normal,dir));
   reflectdir.normalize();

   normalfactor=vec_dot(normal,reflectdir);

   len=dir.length();'

   normalvel=normal*(normalfactor*len);
   tanvel=reflectdir*len-normalvel;
   refdir=normalvel*bump_factor+tanvel*friction_factor;
}
```

这个方法的问题是如果没有碰撞的话，我们把粒子放在 p_x 的时刻，它其实应该位于 p_1。如果我们这么做，那么显然就会“丢失”时间。实际上，在这一时刻，粒子已经碰撞，反弹并移动了一段距离。

要解决这个问题，在每个碰撞点我们都需要进入一个循环。考虑图 2-7b 中的顺序。当找到第一个碰撞点 p_x，我们根据公式 2-1 计算在 p_x 的新速度。为了简化，假设反弹因子是 0，而摩擦因子是 1，那么新速度将与面平行。我们利用这个速度计算出距离 d，并把粒子移动到 p_2。现在我们做了精确的碰撞检测和从 p_x 到 p_2 的反弹。因此我们必须一直循环直到从一点到另一点没有碰撞发生，然后返回最终的位置和速度。使用这个方法没有时间丢失，粒子也将呈现出正确的滑行运动。整个处理过程的伪代码如下：

```
compute_collision(p0,v0,p1,v1)
{
   p = p0
   repeat
```

```
    {
        dir = p1 - p
        if(dir length is 0)
          break

        normalize dir
        if(no collision from p to p1)
          break
        p0 = intersectpoint
        compute_reflection(dir, normal, bump_factor, friction_factor, refdir);
        v1 = refdir* v1
        normalize refdir
        p1 = refdir* length(p1 - p0)
    }
}
```

2.4　特殊的碰撞检测和反弹

讨论了一般的碰撞检测后，我们现在来看看对于很多应用程序来说，这个重要的功能是如何充分地优化。在许多应用程序（比如第一人称射击游戏）中有许多快速移动的物体，而且它们与其他移动的物体及场景几何物体发生相互作用。

我们不必再重复一个好的碰撞检测系统对于一个游戏的重要性。这里“好”是什么意思呢？穷举式的多边形物体/多边形物体碰撞检测是很花费时间的，因此我们不得不为了效率而做出一些妥协。请考虑下面一些情况：在动作游戏中，玩家要穿过一个复杂的环境，这个环境已经被预先处理过了，在我们的例子中被转换为 BSP 树。一些弹射物在环境周围飞行直到它们撞到静止物体或者其他运动的物体。第一人称射击游戏平台能够支持其他一些应用，比如全部遍历。开发一个好的、我们能够负担得起的碰撞检测系统是非常有价值的。

在这些情况下，我们采用的解决方法是精确地检测在动态物体的包围盒与单一层次或静态物体之间的碰撞。在一个场景中，这是一个不对称的解决方法，为什么动态物体使用包围盒，而其他物体使用实际的多边形呢？答案是：因为 BSP 的预处理，且它的结果是最精确的模型，所以我们能承受这样的代价。

角色包围盒的大小影响了游戏中角色的运动，并决定了角色能够去哪里：他能挤过去的最小的通道，他能掉下去的洞的最小直径。包围盒并不需要完全包住一个角色。也许允许一部分穿过包围盒而使包围盒内更紧密会更好，正如附录 2.1 中演示的那样。

注意，一个运动角色的包围盒必须包含它所有动作的幅度。如果根据运动状态，我们不断改变包围盒的大小，那么这将抵消对于一个动态物体使用包围盒的效率优势。在多人游戏中也有相似的情况，虽然不同的角色有不同的尺寸，为了游戏的公平性，所有的角色都应该有相同的包围盒大小。因此我们看到，包围盒体积大小的选取并不是只取决于它所包含的几何物体。

准确的碰撞反弹也是花费很大的，要精确的计算需要把它作为运动模拟的一部分。反弹

是关于质量、角速度、线速度和碰撞物体之间的摩擦的一个复杂的函数。对于许多游戏类型来说，一个简化的反作用模型就已经足够了，它由前面讨论的摩擦因子和反弹因子控制。

现在我们来考虑包围盒的性质。曾经有许多研究关注于包围盒的外形、包围盒的有效性和处理代价之间的平衡。另外，在许多要求严格的应用程序中，包围盒被安排成层次结构。在我们的例子中，选择使用 AABB 来包含动态的物体，并且把它用于快速碰撞检测方法中。

现在描述碰撞检查算法：

- 一个 AABB 包含了一个静态层次中拥有复杂几何外形的动态物体。
- 一个 AABB 包含一个动态物体和另外一个由 AABB 表示或者由它实际的网格面表示的动态物体。

这里介绍的算法是为了计算 AABB 与任意边数的凸多边形的碰撞。但是，它也可以很容易地扩展到其他几何物体，比如动态 LOD 贝济埃曲面和三角汤。这个算法已经在程序中实现和测试过；对于曲面和一些其他几何类型也都支持。至于引擎中的实现则已经超出了这里介绍的内容。该算法扩展了以前存在的算法（比如［COHE95］和［GOTT96］），添加了有效的窄相位（narrow phase）碰撞检测处理，并使用了 BSP 树进行快速的全局裁剪。

碰撞检测通过使用下面基本的相交检查来实现：

- 光线/多边形相交检查。
- 光线/AABB 相交检查。
- 边/边相交检查。

2.4.1　AABB 的定义

在计算相交时，AABB 是普通且比较流行的选择。这是因为在相交计算中，对于一个动态物体，根据定义，它的几何属性一般是不会改变的，因此我们可以预先计算出。AABB 是通过它的最大最小点（6 个浮点数）定义的。静态的常数可以用来代表 AABB 的顶点的法线（8）、面的法线（6）、边（12）和边的法线（12）。对于任意大小的 AABB，这些都是常数，并可以实现为常数静态成员。

2.4.2　AABB 类的定义和静态成员的定义

```
Class FLY_ENGINE_API flyBoundBox
{
  public:
    flyVector min,max;

  static int facevert[6][4];
  static int edgevert[12][2];
  static int edgefaces[12][2];
  static float vertnorm[8][3];
  static float edgenorm[12][3];
  static float facenorm[6][3];
  // gets AABB vertex position given an vertex index
inline flyVector get_vert(int ind)
  {
    switch(ind)
    {
    case 0: return min;
    case 1: return max;
```

```
    case 2: return flyVector(max.x,min.y,min.z);
    case 3: return flyVector(min.x,max.y,max.z);
    case 4: return flyVector(max.x,max.y,min.z);
    case 5: return flyVector(min.x,min.y,max.z);
    case 6: return flyVector(min.x,max.y,min.z);
    case 7: return flyVector(max.x,min.y,max.z);
    default: return flyVector(0,0,0);
    }
  };

  // get distance from the origin to one of
  // the 6 AABB face planes given a face index
  inline float get_plane_dist(int ind)
  {
    return ind>2?max[ind-3]:min[ind];
  };

  // check if two AABB intersect
  inline int clip_bbox(flyVector& bbmin, flyVector& bbmax)
  {
    if (max.x>bbmin.x && min.x<bbmax.x &&
      max.y>bbmin.y && min.y<bbmax.y &&
      max.z>bbmin.z && min.z<bbmax.z)
      return 1;
    return 0;
  }

  // checks if a point is inside a bounding box
  inline int is_inside(flyVector& p)
  {
    return p.x>min.x && p.x<max.x &&
           p.y>min.y && p.y<max.y &&
           p.z>min.z && p.z<max.z;
  }
};

int flyBoundBox::facevert[6][4]=
    { {1,7,2,4},{0,5,3,6},{1,4,6,3},
      {0,2,7,5},{1,3,5,7},{0,6,4,2} };

int flyBoundBox::edgevert[12][2]=
    { {0,6},{6,4},{4,2},{2,0},
      {1,3},{3,5},{5,7},{7,1},
      {0,5},{3,6},{4,1},{7,2}                    };

int flyBoundBox::edgefaces[12][2]=
    { {0,2},{4,2},{3,2},{1,2},
      {4,5},{0,5},{1,5},{3,5},
      {0,1},{0,4},{3,4},{1,3}                    };

float flyBoundBox::vertnorm[8][3]=
    { {-FLY_COS45,-FLY_COS45,-FLY_COS45},
      { FLY_COS45, FLY_COS45, FLY_COS45},
      { FLY_COS45,-FLY_COS45,-FLY_COS45},
      {-FLY_COS45, FLY_COS45, FLY_COS45},
      { FLY_COS45, FLY_COS45,-FLY_COS45},
      {-FLY_COS45,-FLY_COS45, FLY_COS45},
      {-FLY_COS45, FLY_COS45,-FLY_COS45},
      { FLY_COS45,-FLY_COS45, FLY_COS45}     };
```

```
float flyBoundBox::edgenorm[12][3]=
                        { {-FLY_COS45,            0,     -FLY_COS45},
                        {           0,     FLY_COS45,     -FLY_COS45},
                        {    FLY_COS45,            0,     -FLY_COS45},
                        {           0,    -FLY_COS45,     -FLY_COS45},
                        {           0,     FLY_COS45,      FLY_COS45},
                        {   -FLY_COS45,            0,      FLY_COS45},
                        {           0,    -FLY_COS45,      FLY_COS45},
                        {    FLY_COS45,            0,      FLY_COS45},
                        {   -FLY_COS45,    -FLY_COS45,              0},
                        {   -FLY_COS45,     FLY_COS45,              0},
                        {    FLY_COS45,     FLY_COS45,              0},
                        {    FLY_COS45,    -FLY_COS45,              0}};

float flyBoundBox::facenorm[6][3]=
     {  {-1,0,0},
        {0,-1,0},
        {0,0,-1},
        {1,0,0},
        {0,1,0},
        {0,0,1}  };
```

2.4.3　碰撞检测和碰撞反弹

我们现在扩展 2.3.2 节中介绍的粒子碰撞检测方法，使用 AABB 来代替粒子。这样首先便于把碰撞检测和碰撞反弹区分开来。传递给碰撞检测主函数的参数包括：一个 AABB（相对于它的原点的最大最小点），当前位置（p_1）（在上一帧物体到达的位置），及要求的目标位置（p_2）（在这一帧物体要移动到的位置）（见图 2-8）。

这个函数将检查提供的 AABB 是否能从 p_1 移动到 p_2。如果发现了一个碰撞，那么它将通过调用碰撞反弹代码来处理，并且一直这样循环以找到所要求的路径。

对于如图 2-9 所示的简单的包围盒/面相交，只需要两次循环就可以了。第一次碰撞把包

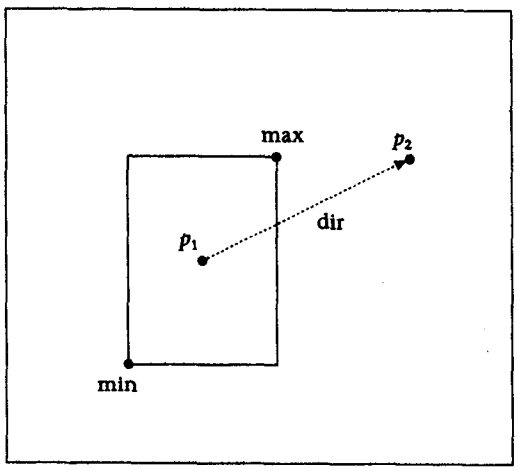


图 2-8　由最大最小点定义的 AABB 从 p_1 移动到 p_2

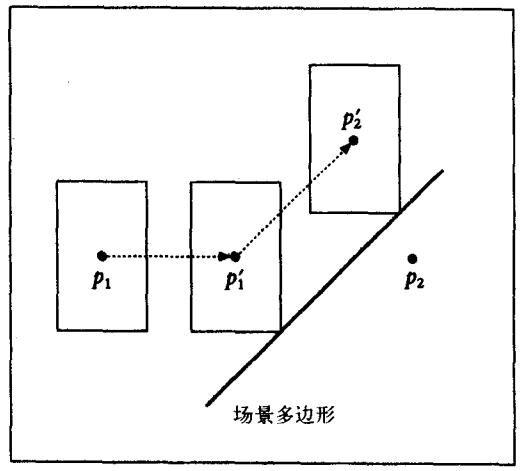


图 2-9　碰撞检测和碰撞反弹递归。AABB 从 p_1 移动到 p_2。碰撞检测例程发现在 p_1' 发生碰撞。碰撞反弹例程确定新的目标位置 p_2'。例程继续检测在 p_1' 和 p_2' 之间是否有碰撞。直到没有碰撞发生，递归结束

围盒移动到 p_1'，并使用反弹代码计算出新的目标位置 p_2'。然后它再次循环，对从 p_1' 移动到 p_2' 的过程进行碰撞检测。因为在 p_1' 和 p_2' 之间没有碰撞，所以循环终止，并且返回 p_2' 作为 AABB 的当前位置。在某些情况下，需要更多次的循环，因为在 p_1' 和 p_2' 之间会再次发生碰撞。

这个碰撞检测主函数的伪代码如下：

Calculate the reflection vector (for the AABB as a particle) and normalise it

· Calculate the collision normal (normal to the colliding plane)

normal_factor = vec_dot (collision_normal, reflect_dir)

length_left = length ($p_2 - p'_1$)

v_1 = collision_normal* (normal_factor* length_left)

v_2 = reflect_dir* length_left - v_1

$p'_2 = p'_1 + v_1$* bump + v_2* friction

v_1 是与碰撞法向平行的向量，v_2 平行于碰撞面。

2.4.4　使用 AABB 的伪碰撞反弹

两个多边形物体的精确碰撞反弹计算花费是很大的。这些花费不仅来自反弹计算本身的开销，而且来自这样一个事实：为了初始化这样一个计算，我们必须在多边形/多边形级别上进行碰撞检测。此外，我们并不是碰撞两个多边形，而是用 AABB 来代替一个单一实体与一个多边形碰撞。

2.3.2 节中介绍的粒子模型是最简单的碰撞反弹模型，也差不多是游戏制作中事实上的标准，特别是在第一人称射击类游戏中。它把碰撞物体当作粒子，把要碰撞的面当作一个平面。现在我们将更深入地讨论这个模型。

粒子一般从表面反射回来，与表面法线形成的反射角等于入射角。这种行为可以做适当的修改，通过反弹和摩擦因子使它更加合理。这两个因子的范围是从 0 到 1。摩擦因子为 0 表示无限大的摩擦力（包围盒将粘在碰撞面上），为 1 则表示没有摩擦力。反弹因子为 0 表示没有反弹（包围盒将沿着碰撞面滑动），为 1 则表示物体在法线方向上的速度分量将不变（包围盒将没有任何衰减地反弹）。在第一人称射击类游戏中，最流行的反弹/摩擦设置是 0/1。这样，当玩家碰到墙时他将沿着墙移动。这里粒子是玩家概念上的中心；我们使用这个去计算反弹，但是不用它去移动物体。在碰撞的过程中，玩家的几何外形不应该被穿透，我们必须把 AABB 的几何结构整合到反弹的计算当中。

反弹的行为模式总结如下：

反弹因子	摩擦因子	作　用
0	1	物体将沿着碰撞面移动，物体关于碰撞面法线的分量被破坏，而在平行方向上的分量没有任何损失
1	0	物体将沿着碰撞法线移动。物体平行于碰撞面的速度分量因为无限大的摩擦力而被破坏
非零	非零	物体将沿着碰撞法线和碰撞面之间的一个方向运动。这个角度由这两个因子的比例决定
0	0	物体将粘在表面上

2.4.5　使用 AABB 的碰撞检测

这个碰撞检测的方法就是要找出，当一个由最大最小点定义的包围盒从 p_1 移动到 p_2 时是否会碰到任何东西。如果碰到了，我们需要给出碰撞点、碰撞法线等信息。要达到这个要求，需要进行若干步计算。然而，幸运的是，我们可以使用点积测试来精简其中的几步，从而提高实时的处理性能。

考虑一个由 8 个顶点、12 条边和 6 个面定义的 AABB。我们需要计算下面集合中最近的碰撞：

a）AABB 的每个顶点与场景几何结构的碰撞。我们需要用光线/多边形相交方法。

b）场景几何结构的每个顶点与 AABB 的 6 个面的碰撞（这可以通过优化的光线/AABB 相交例程简化）。

c）AABB 的 12 条边与场景中每条边的碰撞。

这一过程看起来有很大的计算量。但是通过选取可以把计算量降低到最小，我们可以只计算那些真正产生了碰撞点的边。首先，构造一个临时的包围盒，这个包围盒包含了源包围盒和目标包围盒，如图 2-10 所示。

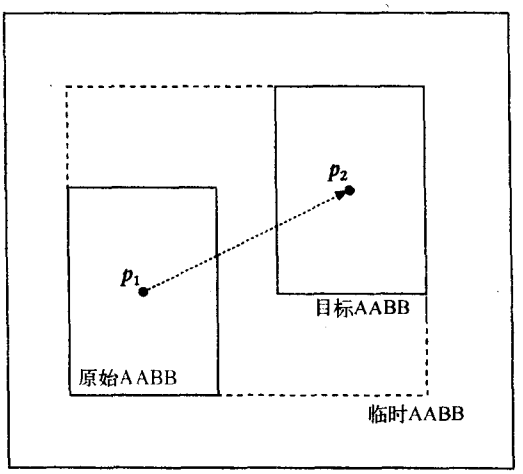


图 2-10　包含所有运动的临时的 AABB，通过向原始的 AABB 添加点（max + dir）和点（min + dir）得到

在物体移动的过程中，这个临时 AABB 必然一直包含该移动物体的 AABB。临时 AABB 实现了第一次的选取操作。我们使用这个包围盒遍历 BSP 树，找出所有与该包围盒相交的 BSP 叶节点。对于这些叶节点上的所有面，我们使用非常简单快速的 clip_bbox 代码去检查面的 AABB 与临时 AABB 是否相交，选取相交的面（见 2.4.2 节）。

至此，我们已经选择了那些只与临时包围盒相交的面。现在，我们要对它们做精确的碰撞测试，包括 a）、b）、c）三方面的测试。

在每次的相交检测中，进行进一步的选取，如下所示：

a）AABB 的 8 个顶点与场景几何结构之间的碰撞：可以丢弃那些法向量与运动方向的点积为负的 AABB 的点。

b）场景中几何结构的顶点与 AABB 的 6 个面之间的碰撞：可以丢弃那些法向量与运动方向的点积为正的面上的点。

c）AABB 的 12 条边与场景中其他边的碰撞：可以丢弃那些与运动方向点积为负的 AABB 的边；同样可以丢弃那些法向量与运动方向点积为正的面的边。

这个碰撞检测例程的伪代码如下：

```
create temporary bound box (original AABB plus move direction* len)
recurse bsp tree with temporary bound box
for all selected leaf nodes
```

```
for all faces in each leaf node
    if face bound box clips temporary bound box
     {
        for each of the 8 AABB vertices
            if vec_ dot (vertex normal, move direction) is positive
                ray intersect face from vertex and move direction
          if intersected, keep if distance smaller than current
       for each of the face vertices
          if vec_ dot (vertex normal, move direction) is negative
            ray intersect AABB from vertex and inverted move direction
            if intersected, keep if distance smaller than current
       for each of the 12 AABB edges
         if vec_ dot (AABB edge normal, move direction) is positive
           for each of the face edges
          if vec_ dot (face edge normal, move direction) is negative
            face edge intersect from from AABB edge and move direction
            if intersected, keep if distance smaller than current
     }
return closest intersection if any
```

2.4.6　AABB 顶点与场景面相交

这是最简单的情况，也是最常见的计算（如图 2-11 所示）。

我们要测试包围盒的顶点（8 个），检查它们在运动方向（dir）上的运动是否会撞到任何场景中的面，这些面是由临时的包围盒裁剪下来的。为了减少计算量，我们去掉那些法向量与运动方向的点积为负的顶点。碰撞点的法向量被碰撞面的法向量定义。我们使用一个简单的多边形光线相交方法，如下：

```
check if face is backfacing the move direction (dir) by the sign of the dot
product from face normal and move dir
compute the intersection distance from ray defined by vertex position and
move direction against the face plane
if distance is negative, return no intersection
compute intersection point using the distance
check if intersection point is inside the polygon
```

光线/多边形碰撞检测代码如下：

```
// computes intersection from a ray defined by its origin (ro) and
// direction vector (rd). Returns true on a collision and the
// intersection point (ip) and collision distance (dist)

int flyFace::ray_intersect(flyVector& ro,flyVector& rd,flyVector&
ip,float& dist)
{
```

```
    // back face culling

    float x=FLY_VECDOT(normal,rd);
    if (FLY_FPSIGNBIT(x)==0)
      return 0; // no intersection

    // compute intersection distance from ray and face plane

    dist=(d0-FLY_VECDOT(normal,ro))/x;
    if (FLY_FPSIGNBIT(dist))
      return 0; // no intersection

    // compute intersection point in face plane

    ip.x=ro.x+rd.x*dist;
    ip.y=ro.y+rd.y*dist;
    ip.z=ro.z+rd.z*dist;

    // check if intersection point is inside the polygon
    // testing the dot products with the polygon edge normals

    for( int i=0;i<nvert;i++ )
      if ((ip.x-vert[i].x)*en[i].x+
        (ip.y-vert[i].y)*en[i].y+
        (ip.z-vert[i].z)*en[i].z > 0)
        return 0; // no intersection

    return 1; // intersection found
}
```

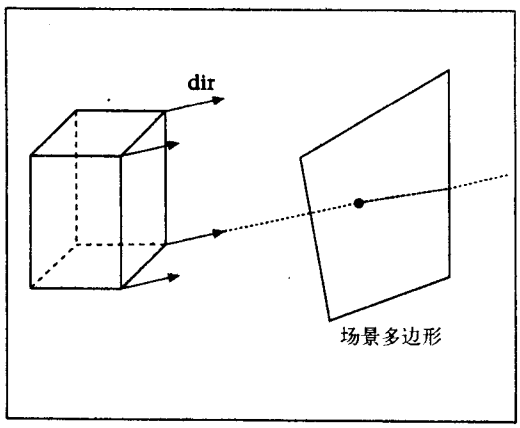


图 2-11　AABB 的顶点与场景多边形碰撞。在此使用标准的光线/多边形碰撞检测方法。光线由 AABB 的顶点和运动方向（dir）确定

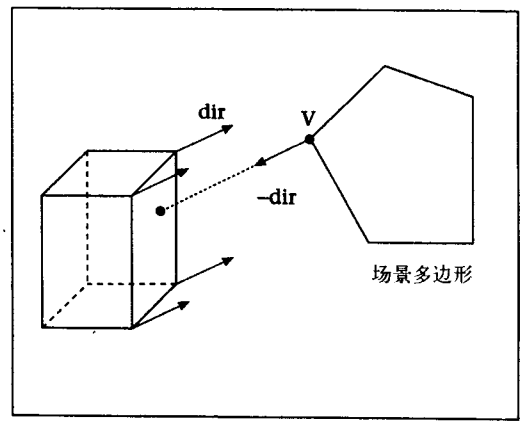


图 2-12　场景多边形顶点与 AABB 的面碰撞。这时，我们使用光线/长方体碰撞检测方法。光线由场景顶点（v）和运动方向的反方向（-dir）确定

2.4.7　场景顶点与 AABB 面相交

前面情况的一种推广就是场景的顶点与 AABB 的相交。这里，我们把由临时包围盒裁剪得到的面上的顶点与包围盒在运动方向的反方向上相交（如图 2-12 所示）。

对于这一部分的计算，我们使用一种简单的、优化的光线/AABB 相交例程。为了减少计

算量，我们去掉那些不在临时包围盒里面的顶点。如果一个面与临时包围盒相交，它可能只有一部分顶点位于临时包围盒之中。

碰撞点的法线是交点所在 AABB 面的法线。

光线/AABB 相交是在［WATT01］中描述的快速的算法。在每一轮，我们处理一对 AABB 的平行面，计算沿着光线到第一个面的距离（t_{near}）和到第二个面的距离（t_{far}）。较大的 t_{near} 值和较小的 t_{far} 值将会被保留。如果较大的 t_{near} 大于较小的 t_{far}，那么光线将不会与盒子相交。

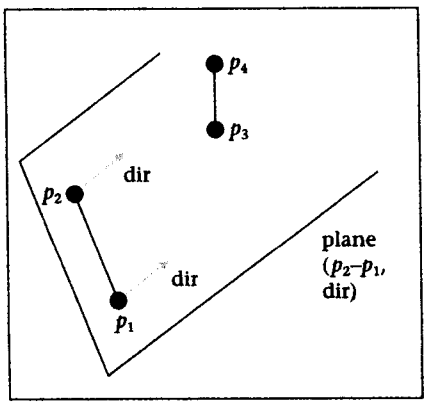
光线/AABB 碰撞检测代码如下：

```
// collide ray defined by ray origin (ro) and ray direction (rd)
// with the bounding box. Returns -1 on no collision and the face
index
// for first intersection if a collision is found together with
// the distances to the collision points (tnear and tfar)

int flyBoundBox::ray_intersect(flyVector& ro,flyVector& rd,float&
tnear,float& tfar)
{
  float t1,t2,t;
  int ret=-1;
  tnear=-FLY_BIG;
  tfar=FLY_BIG;

  int a,b;
  for( a=0;a<3;a++ )
  {
    if (rd[a]>-FLY_SMALL && rd[a]<FLY_SMALL)
      if (ro[a]<min[a] || ro[a]>max[a])
        return -1;
      else ;
    else
    {
      t1=(min[a]-ro[a])/rd[a];
      t2=(max[a]-ro[a])/rd[a];
      if (t1>t2)
      {
        t=t1; t1=t2; t2=t;
        b=3+a;
      }
      else
        b=a;
      if (t1>tnear)
      {
        tnear=t1;
        ret=b;
      }
      if (t2<tfar)
        tfar=t2;
      if (tnear>tfar || tfar<FLY_SMALL)
        return -1;
    }
  }

  if (tnear>tfar || tfar<FLY_SMALL)
    return -1;

  return ret;
}
```

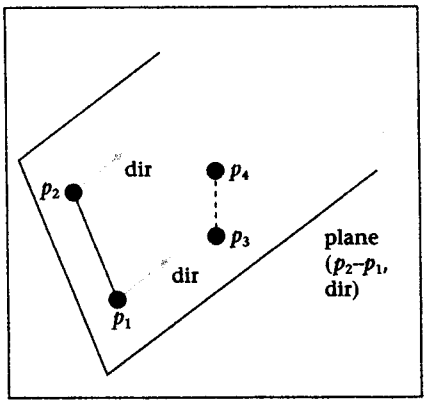
2.4.8　AABB 边与场景边相交

最难计算的情况是包围盒的边与被临时包围盒裁剪得到的场景面的边相交。像前面一样，可以去掉那些法向量与运动方向点积为负的边。我们通过两个点（p_1，p_2）来定义包围盒的边，通过另外两个点（p_3，p_4）来定义场景中的边。下面我们要找出当边（p_1，p_2）沿着运动方向运动时，是否会撞到边（p_3，p_4），并沿着运动方向矢量和相交点返回一定距离。

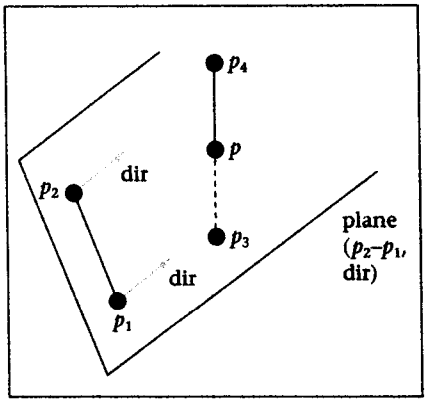
我们首先由边（p_1，p_2）和运动方向矢量（dir）生成一个平面。第一次检测如图 2-13 所示。



a) 没有碰撞，因为 p_3 和 p_4 都位于由边（p_1,p_2）和运动方向（dir）定义的运动平面上



b) 没有碰撞，因为 p_3 和 p_4 都位于运动平面下



c) 可能发生碰撞，因为 p_3 和 p_4 分别位于运动平面的两边。这时，边的两个端点到运动平面的距离的点积一定为负

图　2-13

这里有三种情况，而只有一种可以产生碰撞。在前两种情况（如图 2-13a 和 b 所示）中，顶点 p_3，p_4 都在面的同一边，没有相交的可能。在第三种情况中，可能有相交点，因为点 p_3，p_4 位于面的不同边。如果这样的情况（如图 2-13c）发生了，我们计算出在边（p_3，p_4）上的交点（p）。然后，像图 2-14 那样结束。最后，我们要做的就是让边（p_1，

p_2）和边（p，－dir）相交。

因为这两条线在同一平面内，我们把这些 3D 线投影到坐标平面上，使用快速 2D 线/线相交算法。如果在平面内这两条线相交，那么在 3D 空间中它们也相交。然后，测试交点是否位于点 p_1，p_2 之间。这个可以使用如图 2-15 所示的简单的点积来进行有效的计算。图 2-15a 演示了交点在边的两个端点外的情形，图 2-15b 演示了交点在边（p_1，p_2）的端点之间的情况，即一个碰撞发生了。

碰撞点的法向量由两条碰撞边的叉积定义。因为场景中的边的方向不是预先定义的，所以我们必须测试该法向量与运动方向的点积：如果是正的，那么我们必须反转法向量（－normal.x，－normal.y，－normal.z）。

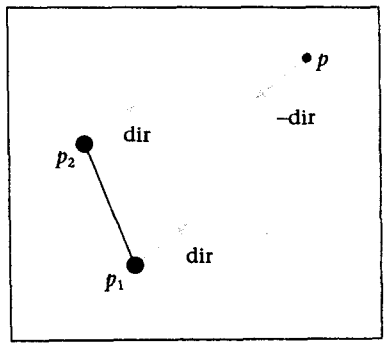
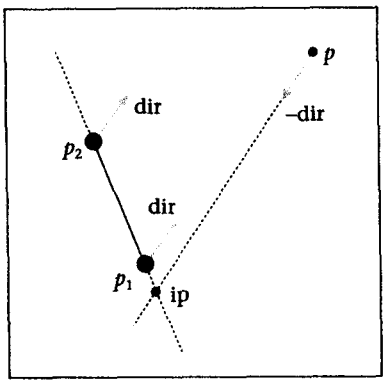
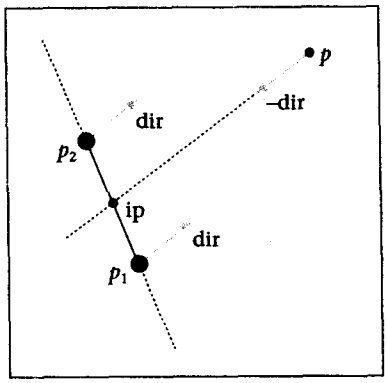


图 2-14　如果发现图2-13c的碰撞情形，那么计算出边（p_3，p_4）和运动平面的交点 p，然后计算线（p_1，p_2）和（p，－dir）的交点



a) 没有碰撞，因为由线（p_1，p_2）和线（p,-dir）得到的交点不在（p_1，p_2）之间，（p_1,-ip）·（p_2,-ip）为正



b) 有碰撞，因为由线（p_1，p_2）和线（p,-dir）得到的交点在（p_1，p_2）之间，（p_1,-ip）·（p_2,-ip）为负

图　2-15

这个算法的伪代码如下：

```
edge_edge_collision(p1,p2,dir,p3,p4,dist,ip)
{
    build plane defined by edge(p1,p2)and move direction(dir)
    if edge verts(p3,p4)is in same side of plane return false
    compute intersection point of line(p3,p4)and plane
    compute largest axis projection of plane
    compute line/line intersection in 2d from line(p1,p2)and
    line(p,-dir)
    if new intersection point is not between edge verts(p1,p2)return false
    return true
}
```

边/边碰撞检测代码为：

```
// collide edge (p1,p2) moving in direction (dir) colliding
// with edge (p3,p4). Return true on a collision with
// collision distance (dist) and intersection point (ip)
int flyBoundBox::edge_collision(flyVector& p1,flyVector&
p2,flyVector& dir,flyVector& p3,flyVector& p4,float& dist,flyVector&
ip)
{
   flyVector v1=p2-p1;
   flyVector v2=p4-p3;

   // build plane based on edge (p1,p2) and move direction (dir)
   flyVector plane;
   plane.cross(v1,dir);
   plane.normalize();
   plane.w=FLY_VECDOT(plane,p1);

   // if colliding edge (p3,p4) does not cross plane return no
collision
   // same as if p3 and p4 on same side of plane return 0

   float temp=(FLY_VECDOT(plane,p3)-plane.w)*(FLY_VECDOT(plane,p4)-
plane.w);
   if (FLY_FPSIGNBIT(temp)==0)
     return 0;

   // if colliding edge (p3,p4) and plane are parallel return no
collision

   v2.normalize();
   temp=FLY_VECDOT(plane,v2);
   if(FLY_FPBITS(temp)==0)
     return 0;

   // compute intersection point of plane and colliding edge (p3,p4)

   ip=p3+v2*((plane.w-FLY_VECDOT(plane,p3))/temp);
   // find largest 2D plane projection

   FLY_FPABS(plane.x);
   FLY_FPABS(plane.y);
   FLY_FPABS(plane.z);
   int i,j;
   if (plane.x>plane.y) i=0; else i=1;
   if (plane[i]<plane.z) i=2;
   if (i==0) { i=1; j=2; } else if (i==1) { i=0; j=2; } else
{ i=0; j=1; }

   // compute distance of intersection from line (ip,-dir) to line
(p1,p2)

   dist=(v1[i]*(ip[j]-p1[j])-v1[j]*(ip[i]-p1[i]))/
     (v1[i]*dir[j]-v1[j]*dir[i]);
   if (FLY_FPSIGNBIT(dist))
     return 0;
   // compute intersection point on edge (p1,p2) line

   ip-=dist*dir;

   // check if intersection point (ip) is between edge (p1,p2)
vertices
```

```
  temp=(p1.x-ip.x1*ip2.x-ip.x)-(p1.y-ip.y1*ip2.y-ip.y)+
(p1.x-ip.x1*ip2.x-ip.x);
  if (PLY_SPSIGNBIT(temp))
    return 1;  // collision found!

  return 0; // no collision
}
```

2.4.9　更精确的碰撞检测

在许多应用程序中，把角色包含在一个 AABB 中是不够精确的。例如，在足球游戏中，我们需要知道足球碰到了运动员的哪个部位；在格斗游戏中，在战斗队列中的一个战士可能需要根据对手的攻击落在哪里，去选择相应的动作。在这种情况下，可以使用一组层次关系的包围盒，如图 2-16 所示。

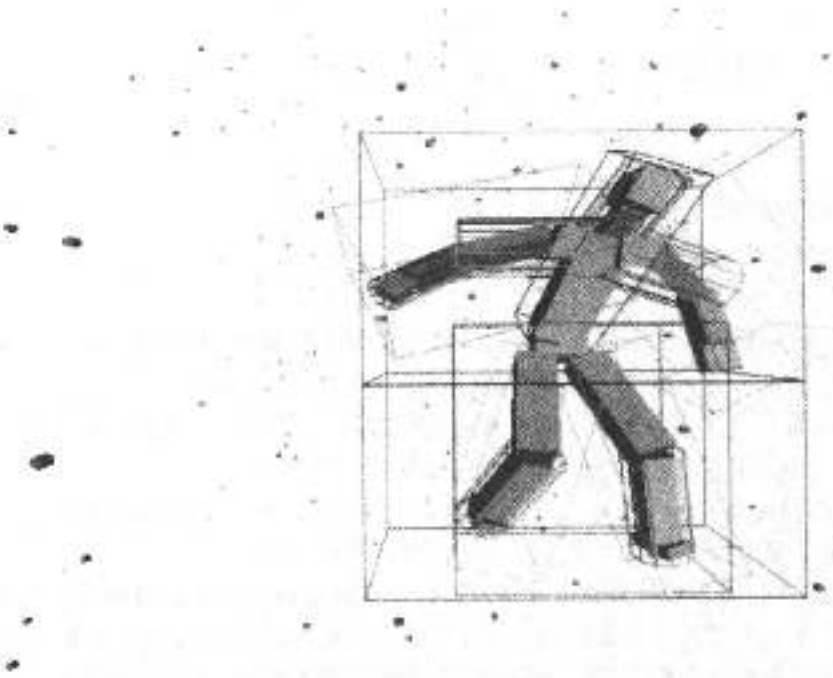


图 2-16　包含角色的包围盒层次结构（由 Dan Hawson 提供）

这里使用了多种包围盒，包括 AABB、OBB（方位包围盒）和圆柱体。

2.4.10　使用碰撞阈值

在这一节，我们将关注碰撞检测中重要的浮点数精度问题。引入这个问题的原因是我们不允许因为浮点数的不精确而穿过一面墙或者一个玩家。通常在这样的情况下，我们需要使用一个距离阈值。我们可以通过检查许多情况来说明这个问题。最简单的情况例如垂直地向一面墙运动（如图 2-17a）。有 3 种可能：

1）目标点 p_2 在碰撞面的前面，离墙面有一定距离。这种情况下，我们可以直接移动到

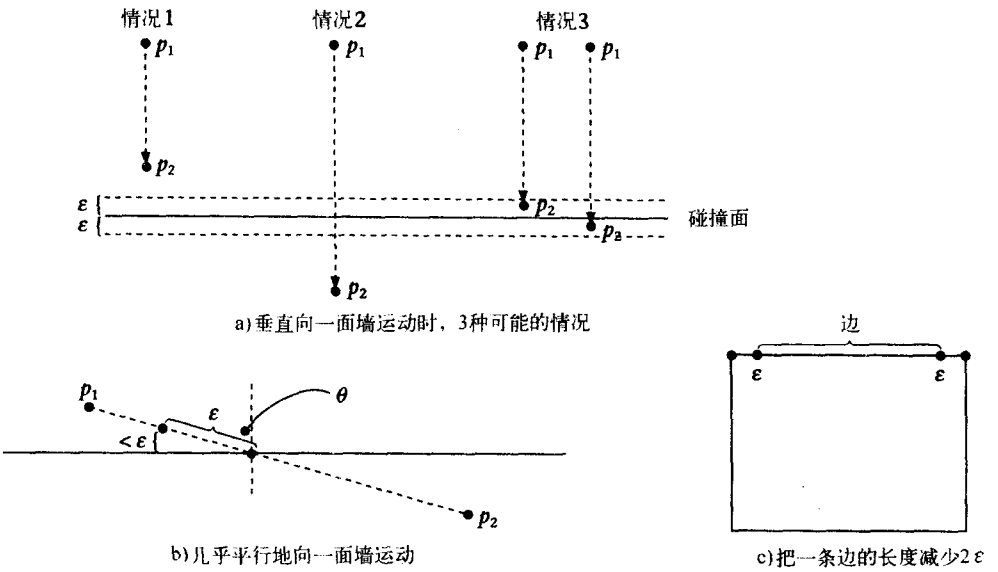


图 2-17　使用碰撞阈值

p_2，而不会有任何问题。

2）p_2 穿过墙面一定距离。在这种情况下，一个碰撞发生了，我们可以移动到碰撞点上面 ε 处。

3）目标点 p_2 到墙面的距离在 ε 之内。这样可能产生问题，因为我们可以认为发生了碰撞，也可以认为没有发生碰撞。如果发生了碰撞，可以像前面的情况那样处理，这样不会有问题。但是，如果没有发生碰撞，并且移动到位置 p_2，那么我们到碰撞面的距离就会小于距离阈值。要解决这个问题，我们按照下面来处理：

i）总是把目标位置加上阈值 ε。如果没有碰撞发生，移动到原来的目标位置。但是如果发生了碰撞，移动到从碰撞点在运动方向上减去阈值的位置。

ii）上面的方法对于垂直或者接近垂直向碰撞面移动的物体来说足够了。但是，如果运动方向接近于平行碰撞面，那么在运动方向上回退阈值距离并不能满足要求（如图 2-17b）。在这种情况下，我们应该根据运动方向和碰撞面法线的夹角 θ，返回一个校正了的阈值。校正公式如下：

$$p_2 = \text{intersection\_point} - \text{dir}^* (\varepsilon/\text{vecdot}(\text{normal}, \text{dir}))$$

这里dir = normalise($p_2 - p_1$)。

当考虑边时，一个相关的问题产生了。如果考虑边的全部范围（从一个顶点到另一个顶点），当两个多边形发生碰撞并且在阈值内，我们不能区分是顶点碰撞还是边碰撞。这种情况可以通过把所考虑的边的范围减去 2ε 来解决（如图 2-17c）。

2.5　基本的路径规划

很显然，路径规划取决于应用程序。许多应用程序把路径规划限制在二维的范围内。在这一节中，我们会介绍一个基本的三维方法，它可以应用于许多应用程序，并可运行于实时

处理。

如 1.2.6 节中描述的，我们可以从凸体生成一组伪入口。使用这些伪入口，我们来寻找一条从源位置到目标位置的路径。最简单可行的方法是从一个凸体所有邻居中随机选择一个，然后移动到这个邻居。

在下面的例子中，我们从当前凸体的中心移动到与被选择邻居相关的伪入口的中心，最后移动到目标凸体的中心。或者，可以选择直接从入口中心移动到入口中心。因为我们可以保证这些体是突起的。我们知道这样的一系列路径永远不会穿过一面墙或其他静态结构。这是能够应用 AI 程序的基础，以提供跟应用程序相关的路径规划。

```
int target::step(int dt)
{
   flyVector dir=targetpos-pos;
   float dist=dir.length();

   if(dist<0.1f)
   {
      find_target();
      dir=targetpos-pos;
      dist=dir.length();
   }

   if(FLY_FPBITS(dist)!=0)
      dir*=1.0f/dist;
   flyVector p=pos+dir*(maxvel*dt);
   if((p-pos).length()>dist)
      pos=targetpos;
   else
      pos=p;

   align_z(dir);

   return 1;
}

void target::find_target()
{
   if(nextnode)
   {
      targetpos=nextnode->centre;
      nextnode=0;
   }
   else
   {
      if(clipnodes.num)
      {
         int r=rand()%clipnodes[0]->neighbors.num;
         nextnode=clipnodes[0]->neighbors[r];
         targetpos=clipnodes[0]->portals[r].get_centre();
      }
   }
}
```

一定要记住：如果用这个方法从许多凸体中寻找一条有效的路径，那么产生的路径形状会依赖于凸体的分解。这个方法可能产生一条不是很直的路径。然而不管怎样，一条有效的路径还是根据构造处理过程中的信息产生出来了。

现在考虑如何自动寻找一条路径。使用 A*算法，我们可以非常有效地找到一条从给定

源点到给定目标点的路径。首先，找到包含开始点和目标点的叶节点，然后应用 A * 算法找到我们将穿过的所有叶节点。根据这个必须穿过的叶节点的列表，我们可以计算出路径，线性地穿过伪入口中心（或者是曲线地穿过，使用贝济埃曲线，控制点在伪入口中心）。

图 2-18a（彩页中也有）演示了导弹随机选择层中任何地方的一个 power-up，计算出一条从它当前的位置到 power-up 的路径（红色）。图 2-18b 演示了一个简单“平滑”处理应用于这条路径的效果。我们尝试去掉路径上的一些点，然后检查这条路径是否仍然正确，即它是否与场景相交。

A* 路径选择

A*算法最初在 1968 年提出，这是一个经典的 AI 算法，它能够直接用于寻找一条从源点到目标点的最佳路径。虽然 AI 不在本书的讨论范围内，但是既然我们已经把一个层次分解成一组凸体以作为构造处理的一部分，那么我们将介绍利用凸体寻找最佳路径的 A*算法。记住，我们已经找到了所有可行的从源凸体到目标凸体的路径，现在，A*算法将从中找出代价最小的路径。

我们应该注意到游戏中的 AI 系统很可能是多层的。而这里，我们讨论的是控制物体在层或者游戏环境中选择哪条路径、怎么走的部分，这一部分可能组成 AI 系统的最低两层。在最底层，几何分解（凸体）使我们可以找到可能的路径；在此之上，我们分析这些路径，根据一些标准找出最佳的一条。在这一层上面将是更高级的行为控制：例如，目标节点是如何选择的，到目标点的过程中要遵循什么策略？依据应用程序，还有其他许多细节的问题需要考虑。在我们的例子中，避免与细节物体和其他动态物体的碰撞必须与物体的运动同步处理。如果物体使用一条路径，而一部分路径的大小不允许物体穿过，那该怎么办呢？是应该把这个标准作为路径寻找算法的一部分；还是应该让物体继续探索下去，当“碰”到后再返回呢？

正如我们已经讨论过的，这些主题作为 AI 教材的一部分更合适；但我们也认为实现一个自动的可能路径选择作为实时处理中更高级别的 AI 的基础，是现代游戏引擎进行实时处理的必需的一部分。

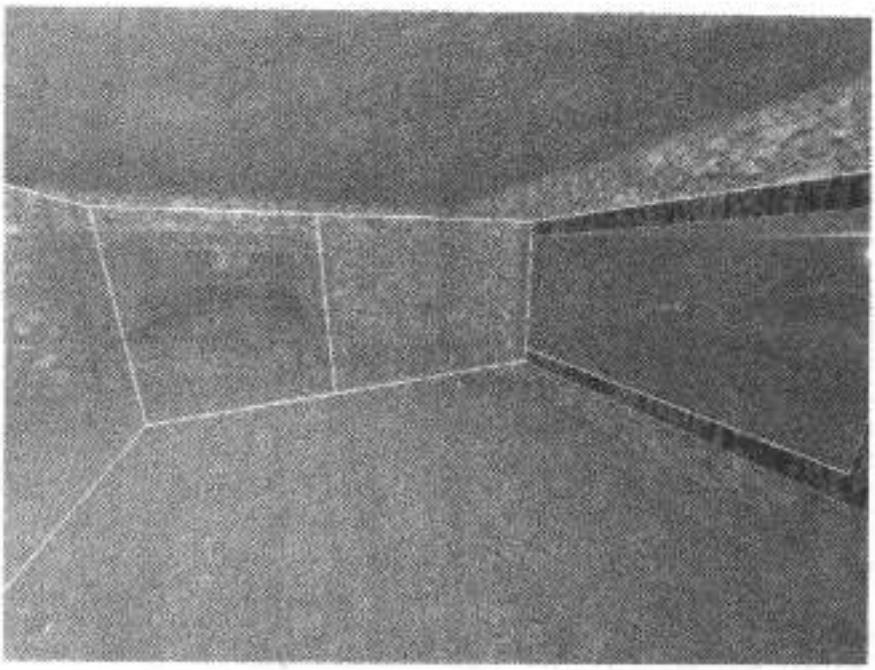
A*给当前路径或迭代中的每个节点分配一个代价函数 $f(n)$。这个函数如下：

$$f(n) = g(n) + h(n)$$

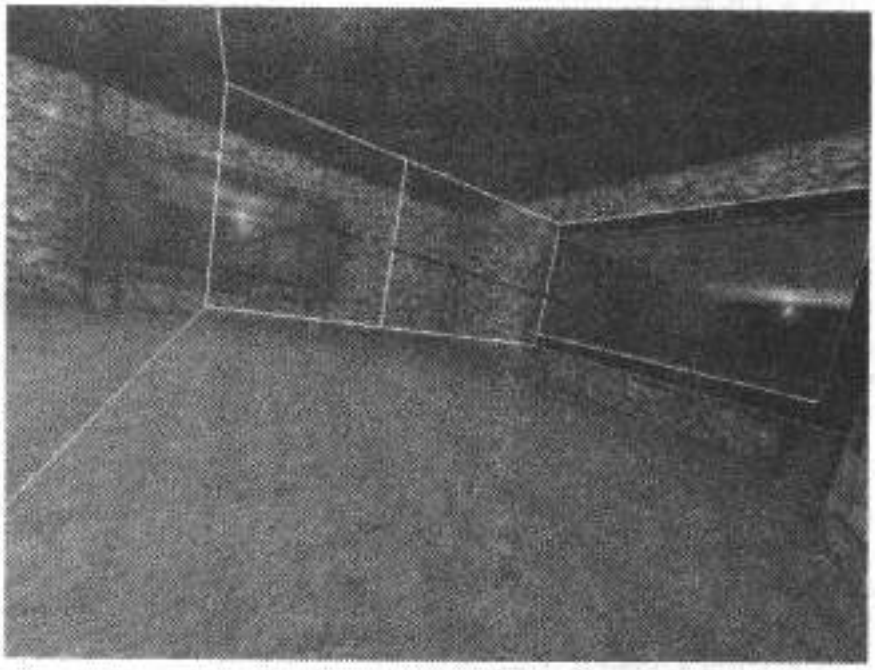
这里，$g(n)$ 是从开始节点到当前节点 n 的路径的代价，$h(n)$ 是启发性地从 n 到目标节点代价最小的路径。

在我们的例子中，“代价”是与距离有关的。例如，考虑要通过大量的中间城市找出两个城市之间的最佳路径。对于当前的城市及 n，$g(n)$ 是从开始城市到 n 的距离，$h(n)$ 是从 n 到目标城市的直线距离。h 被认为是“可容许的启发”，因为它计算出了一个小于等于从 n 到目的地最佳路径的代价。（一般情况下，对于所有的 n，$h(n)$ 可以设为 0，算法降低为盲搜索。）在这种情况下，A*算法保证返回一条最小代价路径。

这个算法寻找最优路径的能力依赖于启发函数 $h(n)$。在决定下一步怎么走时，这个函数的权越重，被处理的节点的数目就越少，而找到次优路径的可能性越高。在使用 A*时的平衡就是：它越接近于盲（精确）搜索，它将访问的错误节点越多。Steve Rabin 的关于 A*速度优化的文章［RABI01］讨论了启发代价对于结果的质量和搜索速度的影响。



a) 使用 A* 算法得到的从敌占体到目标占体路径的一部分



b) 为了使路径更“平滑”，可以去掉路径上的某个顶点，
然后检查去掉顶点之后的路径是否会发生碰撞

图 2-18

这个算法将探索由连接节点的路径组成的图中所有可能的路线，并且它将优先访问拥有最小 f 值的节点。对于每次迭代，总是有这个算法已经访问过的或者还没有访问的节点。对于每个被访问的节点，它可能是已经找到的从源路径到目标路径的一部分。节点上的信息需要更新。节点只记住最小代价路径。

这个方法的实现，是通过在程序中维护两个列表 Open 和 Closed，分别对应未访问的和访问过的节点。两个列表都被初始化成空列表。在第一次迭代时，开始节点被放在 Open 列表中，算法访问它的邻居节点（直接与该节点相连的节点）。像这样对第 n 个节点的邻居节点的测试叫做 n 的展开。每次迭代，算法从 Open 列表中删除最佳节点（f 最小）并展开这个节点。当展开中包含的节点 n' 已经被访问过时，则可能这个展开得到的一条路径比以前得到的路径拥有更小的代价。这时算法更新节点。

下面的结构是 A*搜索需要知道的基本情况。

```
struct a_star_node
{
   int leaf;
   float cost;
   float estimate;
   a_star_node* parent;
};
```

每个 A*节点对应于场景中的一个凸体。leaf 变量是外部凸体数组的一个索引，它定义这个关系。cost 和 estimate 变量分别代表 $f(n)$ 的两个组成部分 $g(n)$ 和 $h(n)$。parent 指针使 A*能够从目标节点反向构造路径。

下面的方法在一个节点列表中寻找“最佳”节点。一般来说，“最佳”节点是拥有到目标节点最低估计代价的节点。

```
int get_best_node(flyArray<a_star_node*> list,flyBspNode*
dest,a_star_node* &node)
{
   float f,mindist=FLY_BIG;
   int best=-1;

   for(int i=0;i<list.num;i++)
      if(list[i]->leaf!=-1)
      {
         f=list[i]->cost+list[i]->estimate;
         if(f<mindist)
         {
            mindist=f;
            best=i;
            node=list[i];
         }
      }

   return best;
}

float get_cost(flyBspNode* a,flyBspNode* b)
{
   return (a->centre-b->centre).length();
}
```

A*还需要一个方法找出给定的节点是否在给定的列表中。如果在，那么这个节点在列表中的索引是多少：

```
int is_in(flyArray<a_star_node*> list, a_star_node* node)
{
  if(node->leaf==-1)
    return -1;
  for(int i=0;i<list.num;i++)
    if(list[i]->leaf==node->leaf)
      return i;

  return -1;
}
```

最后，如前面描述的，下面的方法是 A*算法本身。

```
int a_star( flyBspNode* source,flyBspNode* dest,
            flyArray<a_star_node*> nodes,flyArray<flyBspNode*>& path)
{
  flyArray<a_star_node*> open,closed;
  a_star_node* node=0;
  int ind;

  nodes[source->leaf]->leaf=source->leaf;
  nodes[source->leaf]->cost=0;
  nodes[source->leaf]->estimate=get_cost(source,dest);
  nodes[source->leaf]->parent=0;
  open.add(nodes[source->leaf]);

  while(open.num)
  {
    ind=get_best_node(open,dest,node);
    open.remove(ind);

    if(g_flyengine->leaf[node->leaf]==dest)
    {
      path.add(g_flyengine->leaf[node->leaf]);
      while(node->parent)
      {
        node=node->parent;
        path.add(g_flyengine->leaf[node->leaf]);
      }

      return 1;
    }
    else
    {
     for(int i=0;i<g_flyengine->leaf[node->leaf]->neighbors.num;i++)
     {
        int neighbor_leaf=g_flyengine->leaf[node->leaf]->
        neighbors[i]->leaf;
        a_star_node* newnode=nodes[neighbor_leaf];

        if(newnode->leaf==-1)
        {
          newnode->leaf=neighbor_leaf;
          newnode->cost=get_cost(source,g_flyengine->leaf
          [neighbor_leaf]);
        }

        float newcost=node->cost+
          get_cost(g_flyengine->leaf[node->leaf],
          g_flyengine->leaf[newnode->leaf]);
        if(((is_in(open,newnode)!=-1)||
            (is_in(closed,newnode)!=-1))&&
            (newcost>=newnode->cost))
            continue;
```

```
            else
            {
               newnode->parent=nodes[node->leaf];
               newnode->cost=newcost;
               newnode->estimate=
                  get_cost(g_flyengine->leaf[newnode->leaf],dest);

               int j=is_in(closed,newnode);
               if(j!=-1)
                  closed.remove(j);
               j=is_in(open,newnode);
               if(j!=-1)
                  open[j]=newnode;
               else
                  open.add(newnode);
            }

            closed.add(node);
         }
      }
   }

   return 0;
}
```

附录 2.1　实时处理的演示

这一节的演示是关于前文中描述的使用控制台命令或者使用编辑器的实时处理，由一些参数实验组成。

(1) 视见约束体选择

这个演示展示了视见约束体选择的作用。它设置一个照相机，从场景上面垂直地向下看，并且用线画出视见约束体。场景由视见约束体裁剪得到的所有 BSP 叶节点绘制得到(使用或者不使用 VFC)。

1) 运行任何层。

2) 按 ESC 唤出控制台。使用控制台命令 set debug 4 并且使用标准模式里面的控件来平移和旋转视见约束体。这个模式使用 VFC，只渲染由视见约束体裁剪得到的场景的一个子集。

3) 使用控制台命令 set debug 12 关掉 VFC。

4) 使用 F6 来开关 PVS。

(2) 第三人称照相机

1) 运行 ship_ Mp1 演示。这个演示实现了第三人称照相机（按 z 变换到第三人称模式）(见图 A2-1（彩页中也有))。

2) 在编辑器（或者控制台）里，你可以尝试设置照相机的参数。例如，在控制台里面：

```
set cam.height - 100          //从下向上观察
set cam.dist 250              //从远处观察
set cam.maxaccel 0.0002       //脱离照相机
```



图 A2-1　第三人称照相机模式

(3) 碰撞检测

这个演示将使你熟悉这一章描述的碰撞检测方法论的行为。

1) 运行 **Padgarden** 层。

2) 按 **ESC** 唤出控制台。使用命令 set debug 1 显示包含第一人称的所有动态物体的 AABB。(注意，对于动画的静态物体，例如 power-up，它的包围盒包含在一个大的包围盒里面，这个包围盒包围了它们整个的运动。这样效率会更高，因为只需要计算一次这样的包围盒，而不是每帧都要重新计算。)

3) 边/边：像下面第三人称视角的屏幕截图一样，试着移动到相似的位置。这两张截图(图 A2-2 (彩页中也有)) 只演示了边/边碰撞。当你全心关注这样的情形，就会得到关于碰撞反弹代码的一个好办法。

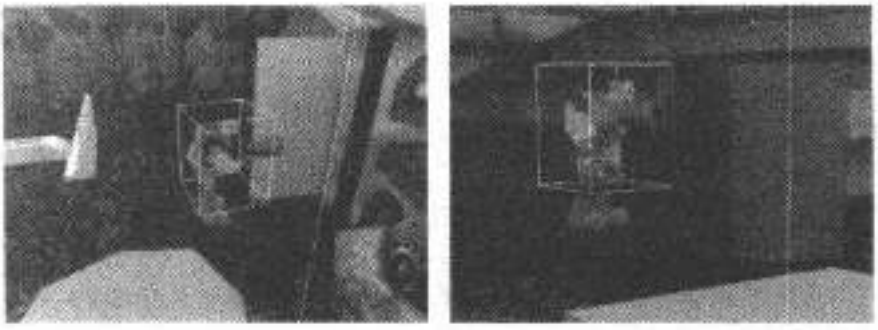


图 A2-2　边/边碰撞

4) 场景顶点/**AABB**：这里，黄色花匠工具上的顶点碰到了玩家的 AABB (图 A2-3 (彩页中也有))。注意，玩家的包围盒实际上并没有包围整个玩家。这是因为我们必须根据与动画范围无关的静态物体做决定，选择如何在包围盒的紧密程度和碰撞检测的精确性之间寻求平衡。因此，我们容忍一些穿透。这里还有一些关于孔径大小等的考虑，决定玩家是否能通过。在多人游戏中，所有的玩家都应该有相同大小的包围盒，因此他们能够到达完全一样的场景区域。

5) **AABB** 顶点/场景：这里，玩家在另外一个动态物体 (船) 中 (图 A2-4 (彩页中也有))。在这种情况下，玩家会与船的面碰撞，而船的面本身是运动的。如果船正在运动，它将是一个要测试的 AABB。

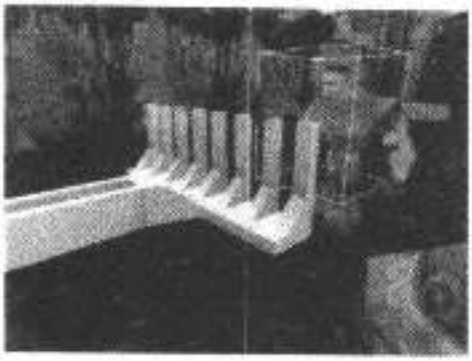


图 A2-3　场景顶点/AABB 碰撞

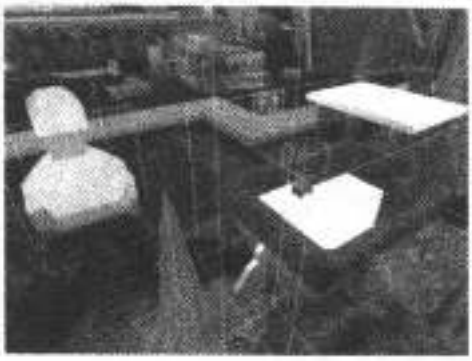


图 A2-4　AABB 顶点/场景碰撞

6）**玩家运动——摩擦和反弹**：玩家是动态的物体，如我们前面讲的，碰撞反弹由当前的速度、反弹和摩擦因子决定。在控制台里面输入 set player.friction 0，把摩擦因子改成 0，这样将产生无限的摩擦力，使玩家粘在碰撞点。反弹因子决定沿碰撞法线的反作用。把它设置成 1 意味着你在碰撞法线方向上的速度将与碰撞前一样。通常在第一人称射击类游戏中，摩擦因子 = 1，反弹因子 = 0。在这种情况下，玩家将沿接触面滑动直到改变方向。

7）**玩家质量和质重力**：当一个玩家跳跃或者从高处掉下来时，它的质量和重力控制下面的运动。一个跳跃垫子根据定义有一个目标位置。当跳跃垫子激活时，它将根据当前的质量和重力，推动玩家以恰当的运动方式到达目标位置。把玩家的质量设置为 5（更重），注意观察效果。因为玩家变重了，但是他必须被退到相同的目标位置，所以需要更强的力，而玩家着陆也更重了。

伪入口演示

这个演示运行了一个拥有双重伪入口面的层次。总是选择一个穿过伪入口的方向，从而更容易地穿过层次。

要观察包含照相机中心的叶节点的所有入口，则使用下面的控制台命令：

set debug 4

这个可以在任何层次中使用。当你移动时，你将能够看到来自所有节点的所有入口。

第 3 章　高级游戏系统剖析Ⅲ：软件设计与应用编程

（**注意**：这一章应与光盘上的引擎参考手册一起阅读。）

这一章的目的是介绍怎样用前两章介绍的方法制作一个游戏引擎，怎样用这个游戏系统和其他部件（比如前端（front-end），这些部件对于操作系统是必不可少的）创建一个新的功能。同时，这一章也对怎样使用这个系统进行描述。

一个游戏引擎应该满足两个要求：第一，它必须能有效地渲染所有静态和动态物体以及它们之间的交互动作。第二，它应该能很容易地辅助新的功能和游戏实体。对于系统能够实现的应用中需要的所有进程，这个游戏引擎都应该能实现。这样，所有我们决定的行为都应该具有一般性——比如，碰撞检测应该是这个引擎的一部分。现行的游戏类型包括很多普通的进程。问题是：什么方法是既可以实现一般行为又可以使游戏构造器参与进来的最佳方法。一个很好的方法就是把所有的游戏专用行为制作成一些插件，这样所有主要的操作系统就可以在一个前端的应用中调用它。用这种方法很容易给软件加进一个新的行为或特征，而不需要重新编译。这样，一个新的游戏或应用就可以被制作成一个插件的引擎链接文件，并且可以运用引擎界面上的所有类、方法和变量。

一个好的引擎最重要的特征之一就是效率。在这个实例中，体现其性能的最关键一点在于在代码中保持高相关度的方法，以及类的紧密性和可实现性，从而避免当需要一块数据时用间接的不必要层次。另一方面，不需相互了解的类和模块可以分开保存，从而保证可移植性、稳定性和代码的安全性。

下面是我们针对 Fly3D 而采用的一个特殊软件设计的描述。像许多游戏引擎，事实上和任何软件项目一样，开发第 1 版时得到的经验都可以促进第 2 版的开发。这一章详细描述了第 2 版 Fly3D 的软件体系结构，给第 1 版的提高提供了动力［WATT00］。虽然这几乎是针对 Fly3D 的，但是，在一个复杂的可导航的环境中，游戏应用的引擎允许用户控制的行为，这证明了这一类游戏引擎的普遍规律。我们给出方法不是作为“游戏软件设计中的结论”，而是作为一个艺术状态下游戏软件开发包（SDK）的一个例子。

3.1　应用的种类

为实现游戏系统的新功能，我们用下面几种应用：

- 插件（plug-in）
- 前端（front-end）
- 工具（utility）

3.1.1　插件

插件是实现新的或特定游戏行为的动态链接库。从命名可以看出，一个 dll 文件在运行时可以被链接到应用上。插件的行为可以通过在它里面定义的 C++ 类的虚拟功能来实现。插件中定义的每一个新类必须从引擎 flyBspObject 中衍生得到，它从中继承了一些函数和属

性，例如位置、速度等。

一般插件的特征

每一个插件都必须满足从它的 dll 库输出三个的全局方法才能作为 Fly3D 的插件，它们是：

```
int num_classes()
```

这个方法必须返回插件实现的类的个数。

```
flyClassDesc *get_class_desc(int i)
```

这个方法必须返回第 i 个实现的类的类描述指针。

```
int fly_message(int msg,int param,void *data)
```

这是这个插件处理从引擎或者其他插件得到信息的地方。

插件的种类

每一个插件都可以是如下类型之一：

- 处理性插件；
- 定义一个对象集合的插件，即一个插件仓库；
- 上面两种插件的合成。

一个完整的游戏可以被制作成一个单独的插件，但任何应用都应该分成不同的模块，以方便代码的重用、维护和查错。

处理性插件

处理性插件不用枚举类。它们通过从 fly_message () 的 dll 文件输出的全局方法来实现它们的功能。它在提示插件初始化场景、结束场景的同时被引擎调用，同时使插件更新状态，通过三维而不是二维的方式显示。

这种插件可以处理下面的标准消息以及用户定义的消息。

```
FLY_MESSAGE_INITSCENE        //初始化场景
FLY_MESSAGE_UPDATESCENE      //更新场景
FLY_MESSAGE_DRAWSCENE        //绘制场景（三维）
FLY_MESSAGE_DRAWTEXT         //二维绘制
FLY_MESSAGE_CLOSESCENE       //结束场景
FLY_MESSAGE_MPMESSAGE        //多人消息
FLY_MESSAGE_MPUPDATE         //多人更新
```

用来枚举对象的插件

这种插件枚举从 flyBspObject 引擎类衍生的新游戏对象。每一个插件可以枚举无限数目的通过基类虚拟方法来实现功能的对象。每一个对象都可以输出预先定义好的类型参数（比如整型、浮点型、向量、颜色、字符串等）。

合成型插件

合成型插件枚举对象并处理消息。一个完整的游戏可以通过这样的插件来实现，但为了构件和代码结构的重用，最好把一个应用分成几个插件。

合成型插件的例子

这部分（看下面的代码）的例子是一个具有盒子几何信息的简单动态对象。

首先，每一个对象都有下面类的实例：

- 对象参数（浮点型、整型、向量、颜色、对象指针）
- 构造器
- 复制构造器
- 析构器
- 用户方法
- 虚拟方法

头文件定义了一个通过插件 observer 和它的类描述 observer_desc 实现的新类。这个 observer 是从基类 flyBspObject 衍生出来的，并且通过虚拟函数 init 和 step 实现功能。observer 由 5 个浮点参数构成，这 5 个参数分别定义了键盘的旋转速度、鼠标的旋转速度、玩家加速、最大速度和减震因子。注意，在构造函数中我们必须用默认值初始化所有的类参数，这样当创建这种类型的新对象时，这个对象就有效了。每一个对象都必须通过所有对象参数从补充对象上拷贝下来的参考 Const 类来实现一个复制构造器（基类的复制构造器因此而得名）。对象的析构器必须是虚拟的。

对象的行为通过虚拟函数来实现。方法 clone 必须返回一个和原对象一样的实例。它的实现调用了复制构造器和返回值。方法 init 在添加到模拟前被每个对象调用一次；这里对象服从于任何所需用户的初始化。在这个例子中，对象的包围盒（用来做碰撞检测）在半径参数的基础上被建立起来。

同样，对于插件中实现的每个类，必须生成其描述类。这个描述类从引擎的 flyClassDesc 类中衍生出来，实现一些极其简单的虚拟函数：

```
create ()          //返回这个类的一个新实例
get_name ()        //返回一个类名字符串
get_type ()        //返回标志这个类的一个整数
```

主函数的行为模拟也被加入到方法 step 中。这个方法收到了已用时间（elapsed time）（原先的帧数用毫秒表示），并且相应地更新了对象的状态。在这个例子中，更新对象的状态意味着根据它的当前速度和受力移动了它，并同时应用了碰撞检测。这一精确的操作序列通过对象 observer 按下面步骤实现：

1）检查输入（鼠标和键盘）。

2）对现行的速度向量应用减震因子，并加进最大速度。

3）计算所需的目标位置和速度。

4）应用碰撞检测，返回一个位置和一个速度（如果碰撞发生的话）。

最后的方法 get_custom_param_desc 是用来返回引擎对象的用户参数的（在 5 个浮点参数的例子里）。对每个参数下面的信息都必须添加：

- 参数名字（字符串）
- 参数类型（整数）
- 参数数据指针（空指针）

插件的处理部分是通过 fly_message 全局输出插件方法来实现的。在这个例子中我们处理两个消息——三维场景绘制和二维场景绘制。三维场景绘制从观察者的角度来渲染世界。二维场景绘制当前的帧速度等。

observer.h

```
enum
{
   TYPE_OBSERVER=0x199,
};

class observer : public flyBspObject
{
public:
   // object parameters
   float radius;
   float rotvel;
   float mousevel;
   float moveforce;
   float maxvel;
   float veldamp;

   // constructor
   observer() :
      radius(20),
      rotvel(0.1f),
      mousevel(0.1f),
      moveforce(0.01f),
      maxvel(0.1f),
      veldamp(0.001f)
   { type=TYPE_OBSERVER; }

   // copy constructor
   observer(const observer& in) :
      flyBspObject(in),
      radius(in.radius),
      rotvel(in.rotvel),
      mousevel(in.mousevel),
      moveforce(in.moveforce),
      maxvel(in.maxvel),
      veldamp(in.veldamp)
   { }

   // destructor
   virtual ~observer()
   { }

   // custom methods
   void check_keys(int dt);

   // virtual methods
   void init();
   int step(int dt);
   int get_custom_param_desc(int i,flyParamDesc *pd);
   flyBspObject *clone()
   { return new observer(*this); }
};

class observer_desc : public flyClassDesc
{
public:
   flyBspObject *create() { return new observer; };
   const char *get_name() { return "observer"; };
   int get_type() { return TYPE_OBSERVER; };
};
```

observer.cpp

```
#include "..\..\lib\Fly3D.h"
#include "observer.h"

observer_desc cd_observer;

__declspec( dllexport )
int num_classes()
{
   return 1;
}
__declspec( dllexport )
flyClassDesc *get_class_desc(int i)
{
   switch(i)
   {
   case 0:
      return &cd_observer;
   default: return 0;
   }
}

__declspec( dllexport )
int fly_message(int msg,int param,void *data)
{
   switch(msg)
   {
   case FLY_MESSAGE_DRAWSCENE:
      g_flyengine->set_camera(g_flyengine->cam);
      g_flyengine->draw_bsp();
      break;
   case FLY_MESSAGE_DRAWTEXT:
      {
      char str[64];
      sprintf(str,"FPS:%i ",g_flyengine->frame_rate);
      g_flyrender->draw_text( flyRender::s_screensizex-56, 0, str );
      }
      break;
   }
   return 1;
}

void observer::init()
{
   bbox.min.vec(-radius,-radius,-radius);
   bbox.max.vec(radius,radius,radius);
}

int observer::step(int dt)
{
   check_keys(dt);

   float len=vel.length();
   if (len<0.01f)
      vel.null();
   else
   {
      vel/=len;
      len-=dt*veldamp;
```

```
    if (len>maxvel)
      len=maxvel;
    if (len<0.0f)
      len=0.0f;
    vel*=len;
  }
  static flyVector p,v;
  p=pos+vel*(float)dt;
  v=vel+force*((float)dt/mass);
  box_collision(p,v);
  pos = p;
  vel = v;

  return 1;
}

void observer::check_keys(int dt)
{
  unsigned char *keys=g_flydirectx->keys;

  if (keys[0x38])  // ALT key
    {
    if (keys[0xcb])  // left arrow
      vel-=X*(moveforce*dt);

    if (keys[0xcd])  // right arrow
      vel+=X*(moveforce*dt);

    if (keys[0xc8])  // up arrow
      vel+=Y*(moveforce*dt);

    if (keys[0xd0])  // down arrow
      vel-=Y*(moveforce*dt);

    if (keys[0x1f])  // S key
      vel-=Z*(moveforce*dt);

    if (keys[0x2d])  // X key
      vel+=Z*(moveforce*dt);
    }
   else
    {
    if (keys[0xc8])  // up arrow
      rotate(-dt*rotvel,X);

    if (keys[0xd0])  // down arrow
      rotate(dt*rotvel,X);

    if (keys[0xcb])  // left arrow
      rotate(dt*rotvel,Y);

    if (keys[0xcd])  // right arrow
      rotate(-dt*rotvel,Y);

    if (keys[0x10])  // Q key
      vel-=X*(moveforce*dt);

    if (keys[0x12])  // E key
      vel+=X*(moveforce*dt);

    if (keys[0x1f])  // S key
      vel-=Z*(moveforce*dt);
```

```
    if (keys[0x2d])  // X key
      vel+=Z*(moveforce*dt);
    }

  if (keys[0x1e])  // A key
    rotate(dt*rotvel,Z);
  if (keys[0x20])  // D key
    rotate(-dt*rotvel,Z);

  if (g_flydirectx->mouse_smooth[0]) // mouse X
    rotate(-g_flydirectx->mouse_smooth[0]*mousevel,Y);
  if (g_flydirectx->mouse_smooth[1]) // mouse Y
    rotate(g_flydirectx->mouse_smooth[1]*mousevel,X);
}

int observer::get_custom_param_desc(int i,flyParamDesc *pd)
{
  if (pd!=0)
  switch(i)
  {
  case 0:
    pd->type='f';
    pd->data=&radius;
    pd->name="radius";
    break;
  case 1:
    pd->type='f';
    pd->data=&rotvel;
    pd->name="rotvel";
    break;
  case 2:
    pd->type='f';
    pd->data=&mousevel;
    pd->name="mousevel";
    break;
  case 3:
    pd->type='f';
    pd->data=&moveforce;
    pd->name="moveforce";
    break;
  case 4:
    pd->type='f';
    pd->data=&maxvel;
    pd->name="maxvel";
    break;
  case 5:
    pd->type='f';
    pd->data=&veldamp;
    pd->name="veldamp";
    break;
  }
  return 6;
}
```

3.1.2 前端

前端是可执行的，它在现行的操作系统里提供了一个用户界面和执行系统。在这一部分，我们介绍一下这种可执行程序的典型集合。

前端的例子如下。

简单的前端（flyFrontend.exe，flyFe.exe）

这里用了基本的操作序列：

1）初始化主窗口、三维引擎，渲染和其他必要部件。

2）装载一个层次（静态几何信息、插件、模型等）。

3）循环——为每一帧调用仿真器（flyEngine -> step ()）。

4）结束引擎和自由配置的资源。

然后形成了游戏主界面并且运行游戏。比如，flyFrontend.exe 是引擎主要的前端应用。它包含一个模拟发生的渲染窗口和一个打开 .fly 文件的菜单。在前端中按 F1 键，就可以运行一个可以选择单用户或者多用户模式的菜单，装载一个 .fly 文件并且启动一个仿真器，或者执行一个预先记录好的演示文件。包含在这个界面里的还有下面一些渲染选项：

F1：菜单

F2：弱音器

F3：记录演示程序

F4：截屏

F5：节点模式

F6：PVS 触发器

F7：映射触发器

F8：纹理过滤触发器

F9：多纹理触发器

F10：模板触发器

F11：线框触发器

F12：清屏

编辑器前端（flyEditor.exe，flyShader.exe）

这是更复杂、更庞大的应用，它使得用户在运行游戏时进入游戏配置和实体内。它是非常有用的开发工具，可以使插件的参数在开发者的实验中公开出来。

flyEditor.exe 是 Fly3D 的主要编辑工具。它包括将一个场景的所有实体都分类显示的树状图，一个参数都编辑显示的列表和一个有游戏或应用正在运行的渲染窗口。flyEditor.exe 允许即时参数改变可视化效果，即你可以改变列表中的一个值并立即看到渲染的结果。你也可以在游戏或应用中加载附加的插件、增加或删除实体、修改已存在的实体和保存一个做过上述改动的 .fly 文件。

flyShader.exe 是一个阴影效果的编辑工具。你可以用它加载一个已知场景的整个阴影列表（一个 .shr 文件），编辑效果并在渲染窗口看到正在运行的阴影。你也可以加载一个 .f3d 网格对象，编辑它的阴影并看它运行。顶点项也可以用一个脚本文件应用于网格中。

flyShader.exe 以其所有可能的阴影效果为特征，它最多有 8 个阴影通道，还有 alpha 通道合成和 RGB 生成函数，还包含不同关键帧的动画地图、卷轴、旋转和纹理比例、环境映射、纹理箝位等。

MFC 中的前端（Fly3D.ocx，flyEditor.exe）

这些程序把微软基础类库和激活新的 Windows 构件的引擎类结合起来运用。Fly3D.ocx

是 Fly3D ActiveX Control。利用它，整个 Fly3D 引擎在一些浏览应用中都是可用的，例如网络三维游戏、三维浏览、三维虚拟购物、三维呈现等。先构造一个含有 Fly3D.ocx 的 HTML 文件，再用控制命令加载一个 .fly 文件开始 3D 仿真。

控制台前端（flyServer.exe，flyBuild.exe）

这些前端不需要渲染，而是通过一个命令行界面处理它所有的输入输出。它们被用于数据处理或者把整个游戏作为一个服务器仿真。

最简单的前端的代码如下。下面是操作序列：

1）生成一个应用窗口。

2）初始化 DirectX、渲染和引擎。

3）设置全屏模式。

4）加载一个实例层（在这个例子里是菜单）。

5）开始仿真直到应用结束。

6）释放所有资源并关上应用窗口。

```
#include <windows.h>
#include "..\..\lib\Fly3D.h"

char szTitle[100]="MyGame Title";
char szWindowClass[100]="MyGame";
// loads the menu level
void LoadLevel(HWND hWnd, HINSTANCE hInst)
{
   fly_init_directx(hWnd,hInst);
   fly_init_render(hWnd,hInst);
   fly_init_engine(hWnd,hInst,FLY_APPID_FLYFRONTEND);

   flyRender::s_fullscreen=1;
   g_flyrender->set_full_screen();

   g_flyengine->open_fly_file("menu.fly");
}

int WINAPI WinMain (HINSTANCE hInst, HINSTANCE hPrev, LPSTR lp, int nCmd)
{
WNDCLASS wcl;
MSG msg;

// register window class
wcl.style           = CS_HREDRAW | CS_VREDRAW;
wcl.lpfnWndProc     = (WNDPROC)WinFunc;
wcl.cbClsExtra      = 0;
wcl.cbWndExtra      = 0;
wcl.hInstance       = hInst;
wcl.hIcon           = LoadIcon(NULL, IDI_WINLOGO);
wcl.hCursor         = 0;
wcl.hbrBackground   = 0;
wcl.lpszMenuName    = NULL;
wcl.lpszClassName   = szWindowClass;
if (!RegisterClass (&wcl))
{
   MessageBox (0, "Can't register Window", "ERROR", MB_OK);
   return 0;
}

// create main window
```

```
HWND hWndMain = CreateWindowEx(0,szWindowClass,szTitle,WS_POPUP,
                0, 0,GetSystemMetrics( SM_CXSCREEN ),
                GetSystemMetrics(SM_CYSCREEN ),
                NULL,NULL,hInst,NULL );

// load the level
ShowWindow (hWndMain, SW_MAXIMIZE);
LoadLevel (hWndMain, hInst);

// main loop
while (1)
   {
   while (PeekMessage(&msg, NULL, 0, 0, PM_NOREMOVE) == TRUE)
      {
      if (GetMessage(&msg, NULL, 0, 0))
         {
         if (g_flyengine)
            if (msg.message==WM_KEYDOWN)
               g_flyengine->con.key_down(msg.wParam);
            else if (msg.message==WM_CHAR)
               g_flyengine->con.key_char(msg.wParam);

         TranslateMessage(&msg);
         DispatchMessage(&msg);
         }
      else
         return TRUE;
      }

   if (g_flyrender && g_flyengine)
      if (g_flyengine->step())
         g_flyengine->draw_frame();
   }
}

// main window message processing
LRESULT CALLBACK WinFunc (HWND hWnd, UINT mens, WPARAM wParam,
LPARAM lParam)
{
   switch (mens)
   {
   // window resize
   case WM_SIZE:
      if (g_flyrender)
         g_flyrender->resize(LOWORD(lParam),HIWORD(lParam));
      break;

   // window activation
   case WM_ACTIVATE:
      if (g_flyengine)
         if (LOWORD(wParam)==WA_INACTIVE || g_flyengine->con.mode)
            g_flyengine->noinput=1;
         else g_flyengine->noinput=0;
      break;

   // quit app
   case WM_DESTROY:
      fly_free_engine();
      fly_free_render();
      fly_free_directx();
      PostQuitMessage(0);
      break;
   }
```

```
    return DefWindowProc (hWnd, mess, wParam, lParam);
}
```

3.1.3 工具

工具在游戏执行过程中应用不到。一般它们把数据从其他格式转化过来，或者处理内部数据。它们可以利用 API 或者 Fly3D。例子如下：

- 用于导入导出的 3DSMax 插件（.FMP，.F3D，.K3C）。这些插件加载并保存从 3D Studio Max 的游戏层和动画实体转化来的数据。
- 其他游戏格式的层次转化器。它把几何信息和纹理贴图从其他格式转化成引擎的格式，这样其他层次的编辑器都可以用来生成 Fly3D 的内容。
- 用于 Visual C++ 的插件向导。这是微软的 Visual C++ 向导，它通过一些对话框生成一些代码的结构，从而使得 Fly3D 插件向导的生成相对简单了一些。

3.2 Fly3D 引擎体系结构

首先，Fly3D 引擎体系结构被分为四个部分：

FlyMath	向量、矩阵、四元数、顶点定义
FlyDirectX	DirectInput、DirectPlay、DirectSound 的封装界面
FlyRender	OpenGL 和纹理的主界面
FlyEngine	引擎的主界面

这些都和插件 OpenGL 和输入设备相关，关系如图 3-1 所示。

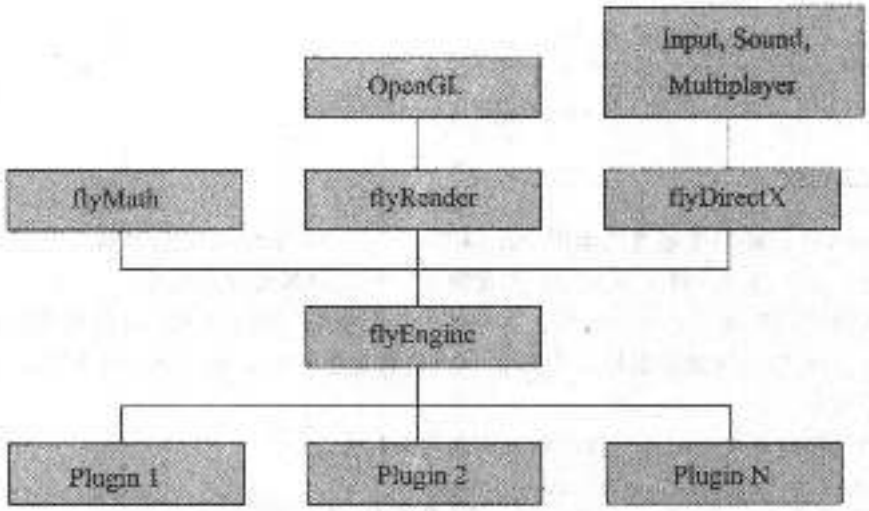


图 3-1 Fly3D 的体系结构

3.2.1 FlyMath

FlyMath 动态链接库实现了和游戏中一般的三维数学知识相关的类和定义。

```
class FLY_MATH_API flyVector; // 4 floats
class FLY_MATH_API flyQuaternion;    // 4 floats
class FLY_MATH_API flyMatrix; // 16 floats
class FLY_MATH_API flyPlane; // 5 floats
class FLY_MATH_API flyVertex; // 8 floats, 1 int
```

- 类 flyVector 实现了一个四部件的向量（x，y，z，w），并且包括了一系列像加、点乘、叉乘、归一化之类的操作。
- 类 flyQuaternion 实现了普通的四元组函数，如 slerp，及四元组和矩阵之间的转换。
- 类 flyMatrix 实现了一些如矩阵乘法、向量矩阵乘法、平移、旋转、缩放之类的函数。
- 类 flyPlane 实现了基于法线和原始距离的一个平面。它有一些如点到平面间距离之类的方法。
- 类 flyVertex 是一个有和顶点相关的所有信息的类，这些信息有：位置、法线、纹理坐标、光照贴图坐标和颜色。

一个普通的数学操作的例子如下：

```
flyVector v1(1); // initialize to (1,1,1,1)
flyVector v2(2,2,2); // initialize (2,2,2,0)
flyVector v3(0,1,0,1); // initialize (0,1,0,1)

flyVector v=v1+v2*v3; // v=(1,3,1,1);

float distance=(v1-v2).length();
float dotprod=FLY_VECDOT(v1,v2);

v.cross(v1,v2); // sets v to cross product of v1 and v2

flyMatrix m;
m.set_rotation(10,flyVector(0,0,1)); // set to a rotation matrix of
10 degrees around z axis

v.vec(1,2,3); // sets v to (1,2,3,0)
v.normalize(); // set v to unit length

v=v*m; // rotate vector v with matrix m
```

3.2.2 FlyDirectX

FlyDirectX 动态链接库是我们运用 DirectX 的一个界面。只有需要这种特征的插件和前端才需要初始化这个部分。这个库实现了处理输入、声音和多人游戏的类。

这个库的主要类就是 flyDirectx 类，这个类的一个全局实例 g_flydirectx 也是可用的。通过 g_flydirectx，就可以用键盘和鼠标检查输入、给仿真加上声音、输入多用户信息、加入和生成多用户游戏等。

为初始化和释放这个构件，我们用下面两种全局方法：

```
// global DirectX initialisation method
FLY_DIRECTX_API void fly_init_directx(HWND hwnd,HINSTANCE hinst);

// global DirectX release method
FLY_DIRECTX_API void fly_free_directx();
```

初始化这个构件之后，它就可以通过下面的全局变量获得：

```
// global flyDirectx instance
extern FLY_DIRECTX_API flyDirectx *g_flydirectx;
```

通过这个初始化的全局指针，我们就可以存取有关输入、声音和多用户的方法。

输入处理

输入处理需要下面两种方法。方法 get_input 从输入设备采样，并将采得的数据存储在公共输入变量中。这个方法在每帧被引擎调用一次，而插件需要的就是为设备信息检查公共输入变量。

输入变量包含一些信息，比如哪一个键被按了（key[]）、鼠标的位置和偏移量（mouse_pos[]，mouse_delta[]）、鼠标的按键（mouse_down，mouse_click）等。

```
// input methods
// get user input from input devices
void get_input();
// reset all input
void zero_input();

// input variables
unsigned char keys[256]; // keyboard keys
int mouse_pos[2];        // current mouse position in screen pixels
int mouse_delta[2];      // mouse displacement from last frame
float mouse_smooth[2];   // smoothed movement displacement
char mouse_down;         // bitfield: which mouse buttons are down
char mouse_click;        // bitfield: which mouse buttons have been
                         // clicked
```

声音工具

声音处理可以用下面的方法。

```
// load .wav sound file
int load_wav_file(LONG cchBuffer,HPSTR pchBuffer,LPDIRECTSOUNDBUFFER
*buf,LPDIRECTSOUND3DBUFFER *buf3d);
```

加载一个 .wav 声音文件。

```
// clone an existing sound
LPDIRECTSOUNDBUFFER clone_sound(LPDIRECTSOUNDBUFFER buf);
```

复制一个现存的声音拷贝。

```
// sets position, velocity and alignment of a listener
void set_listener(const float *pos,const float *vel,const float
*Y,const float *Z);
```

从一个游戏使用者的角度，对声音的正确感知依靠于他/她作为一个虚拟游戏人物的位置和速度。

```
// set master sound volume
void set_master_volume(int volume);
```

设置音量：音量 0 是最大音量，较小音量值都是负数。

多用户处理

在［WATI01］中描述了一个完整的多用户应用。这里我们简单描述一下使插件进入 DirectPlay 的方法。对于多用户应用，使用 API 实现最好。它给因特网一个简单的界面，并使你避免建立和处理消息的底层编程。

一个用户应用会生成一个服务器或一个客户端。服务器通过一个服务器前端初始化。通常一个应用必须生成一个客户端。生成客户端的操作序列如下：

1）调用 init_multiplayer，传递要连接的服务器地址。

2）如果没有专门的服务器地址，就连接到局域网内第一个可获得的服务器。

3）调用 enum_games，给出专门服务器可获得的游戏。

4）调用一个游戏的 join_game，连接到这个游戏。

一旦一个游戏被连接，就可以发送和接收消息。消息通过方法 send_message 发送。这个方法发送一个特定长度的消息给指定的玩家（如果没有指定的玩家，就送到所有的玩家）。

多用户的方法如下：

```
// initialise multi-player session
int init_multiplayer(const char *netaddress=0);
// destroy multiplayer session
void free_multiplayer();

// get a pointer to the list of available games
flyMPGames *enum_games(LPGUID app_guid);
// join an existing game
int join_game(LPGUID game_guid,const char *player_name,unsigned
color=0xFF808080);
// create (host) a new game
int create_game(LPGUID app_guid,const char *game_name);
// get player IP address from its DirectPlay id
char *get_player_address(DWORD dpid);

// send a message over the net
void send_message(const flyMPMsg *msg,int len,DWORD dpid=0);
// get total number of messages
int get_num_messages();
// get a message
flyMPMsg *get_message(DWORD *size);
// add a new player to the environment
int add_player(const char *name,DWORD dpid,void *data,unsigned
color=0xFF808080);
// remove a player from the environment
void *remove_player(int i);
```

3.2.3 FlyRender

这个动态链接库的目的就是把 OpenGL 的界面装入，激活渲染环境、纹理管理和硬件编程。它是引擎的渲染模块，处理所有与渲染任务相关的操作和数据结构，并将 OpenGL 作为图形的 API。这个动态链接库输出它的主要类 flyRender 和这个类的一个全局实例 g_flyrender，并允许完全进入它的函数。

我们用下面两个全局方法来初始化和释放这个构件：

```
// global render manager initialisation method
FLY_RENDER_API void fly_init_render(HWND hWnd,HINSTANCE hInst);
// global render manager release method
FLY_RENDER_API void fly_free_render();
```

初始化这个构件之后，它通过下面的全局变量可得：

```
// global flyTexCache instance
extern FLY_RENDER_API flyTexCache *g_flytexcache;
// global flyRender instance
extern FLY_RENDER_API flyRender *g_flyrender;
```

用纹理管理器可以处理一些事情，比如动态加载新的纹理，改变一个完全纹理贴图或它的一部分，就像光照贴图可能来自于一个动态光照。同样，像过滤选项之类的纹理设置可以特定于一个纹理（不是每个纹理）。多纹理的硬件最优化允许从任何纹理单元中简单选取。

下面的纹理管理方法有这些方便之处：

```
// Dynamically add a new texture to the cache, passing all info
int add_tex(const char *name,int sx,int sy,int bytespixel,const
unsigned char *buf,int flags);

// Dynamically add a new texture to the cache, passing the picture
index int add_tex(int np,flyPicture **pic,int flags);

// Update the texture in the texture manager with a new pixel array
void update_tex(int pic,int sx,int sy,int bytespixel,const unsigned
char *buf);

// Update part of a texture in the texture manager.
// Only the sub-texture pixels are passed.

// If 'x' and 'y' are 0 and 'sx' and 'sy' the size of the image,
// it will work just like the update_picture function.
void update_subtex(int pic,int x,int y,int sx,int sy,int
bytespixel,const unsigned char *buf);

// Select a texture unit
inline void sel_unit(int u);

// Select a texture
inline void sel_tex(int t);

// Select a texture and unit
inline void sel_tex(int t,int u);
```

类 flyRender 处理 OpenGL 的初始化和释放、OpenGL 的扩充、窗口大小变换（包括全屏转换，三维、二维绘制和像线框渲染之类的渲染标志和背景颜色等）。

渲染方法

```
// Initialise the state for every frame
void init();

// Resize the rendering window
void resize(int sx,int sy);
// Change to fullscreen mode
void set_full_screen();

// Enter drawing mode
void begin_draw();
// Leave drawing mode
void end_draw();

// Enter 2D drawing mode
void begin_draw2d();
// Leave 2D drawing mode
void end_draw2d();

// Draw text aligned to the left of the given position
void draw_text(int x,int y,const char *text,int
size=FLY_FONTS_SIZE,int n=-1);
// Draw text centralised on the given position
void draw_text_center(int x,int y,const char *text,int
size=FLY_FONTS_SIZE);
```

渲染标志

```
static int s_screensizex;        // screen width in pixels
static int s_screensizey;        // screen height in pixels
```

```
static int s_fullscreen;        // fullscreen flag
static int s_clearbg;           // background clear flag
static int s_antialias;         // anti-aliasing flag
static int s_wireframe;         // wireframe flag
static int s_fog;               // fog flag
static int s_stencil;           // stencil flag
static float s_brightness;      // brightness level
static float s_ambient;         // ambient lighting level
static float s_aspect;          // screen aspect ratio
static float s_camangle;        // camera angle
static float s_nearplane;       // near rendering plane
static float s_farplane;        // far rendering plane
static float s_background[4];   // background colour
```

3.2.4 FlyEngine

这个引擎自身处理静态场景、插件、动态物体的加载和保存并执行仿真。FlyEngine 是引擎的主要模块，而且是插件和后端模块间的有效连接。这个动态链接库实现了一些类和有用的方法，这些类和方法处理应用状态的每帧更新，协调不同的插件，增强仿真功能。它也包括一些装载场景所需的离线阶段、初始化插件和物体、把仿真从一个有效的初始状态激活。

这个引擎模块主要以类 flyEngine 为特点，这个类包含了引擎最重要的数据和方法，比如数据加载、光照计算和 BSP 递归函数，还有 BSP 树、顶点和面片的几何信息及所有的仿真全局参数。你也可以通过它的全局实例（调用整个应用中都可以获得的 g_flyengine）获得数据，使用这个类中的方法。

这个引擎部件独立于渲染部件，这种独立使用的一个例子就是服务器的前端，它用这个引擎执行所有仿真而不绘制任何东西。

这个引擎的所有类的列表如下：

```
template <class T> class FLY_ENGINE_API flyArray;
class FLY_ENGINE_API flyString;
class FLY_ENGINE_API flyBoundBox;
class FLY_ENGINE_API flyFrustum;
class FLY_ENGINE_API flyLocalSystem;
class FLY_ENGINE_API flyBaseObject;
class FLY_ENGINE_API flySound;
class FLY_ENGINE_API flyPolygon;
class FLY_ENGINE_API flyFace;
class FLY_ENGINE_API flyMesh;
class FLY_ENGINE_API flyAnimatedMesh;
class FLY_ENGINE_API flySkeletonMesh;
class FLY_ENGINE_API flyBezierCurve;
class FLY_ENGINE_API flyBezierPatch;
class FLY_ENGINE_API flyParticle;
class FLY_ENGINE_API flyBspNode;
class FLY_ENGINE_API flyBspObject;
class FLY_ENGINE_API flyStaticMesh;
class FLY_ENGINE_API flyLightMap;
class FLY_ENGINE_API flyLightMapPic;
class FLY_ENGINE_API flyLightVertex;
class FLY_ENGINE_API flyClassDesc;
class FLY_ENGINE_API flyParamDesc;
class FLY_ENGINE_API flyConsole;
class FLY_ENGINE_API flyShader;
class FLY_ENGINE_API flyShaderFunc;
class FLY_ENGINE_API flyShaderPass;
```

```
class FLY_ENGINE_API flyInputMap;
class FLY_ENGINE_API flyFile;
class FLY_ENGINE_API flyDll;
class FLY_ENGINE_API flyDllGroup;
class FLY_ENGINE_API flyEngine;
```

我们现在对所有的类都有了一个简单的概念。详细的内容请参考引擎参考手册。

```
template <class T> class FLY_ENGINE_API flyArray;
```

这个模板类实现了一个任意类型的动态可调整大小的数组。这个数组可以用标准的操作符“[]”来索引，元素可被动态地加入数组。这个设备的一个特征就是当新加入一个元素但现有数组已经没有空间时，它会扩大一倍尺寸。这就实现了有效的重置。另一个特征就是这个数组只会增长（不会缩小）而最终到达一个所需的有限大小。当这个限度达到时它就和预置定长数组一样有效。

```
class FLY_ENGINE_API flyString;
```

这个设备和前一个相似，像一个字符数组一样操作。好几个操作符都被包含进来，以方便连接一个字符串及格式化等。

```
class FLY_ENGINE_API flyBoundBox;
```

这个类实现了一个场景中每个物体都必须有的 AABB。它在碰撞计算中用于第 2 章中讨论的引擎的视图选择。这个类包括碰撞检测需要的所有详细功能，包括被用作优化了的 AABB/多边形碰撞检测的部分的 ray_intersect。

```
class FLY_ENGINE_API flyFrustum;
```

这个类实现了一个视见约束体，保存了它的顶点和平面，也提供了一个截面的建造方法、一个快速包围盒裁剪测试。

```
class FLY_ENGINE_API flyLocalSystem;
```

这个类实现了一个由三个垂直正交的向量定义的局域系统（见图 3-2）。方法包括：

- 根据一个旋转矩阵旋转系统
- 绕向量 v 旋转系统 ang 角度
- 在 v 和 u 定义的平面内，把系统从 v 向 u 旋转最大为 maxang 的角度
- 根据一个已知向量排列系统的 x 轴

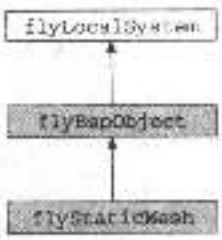


图 3-2　flyLocalSystem 类依赖关系

```
class FLY_ENGINE_API flyBaseObject;
```

这个类是所有物体的基类（见图 3-3）。好几个其他类都是这个类的衍生类。它有物体的名字和一个指针，该指针指向特定物体链表的下一个物体。

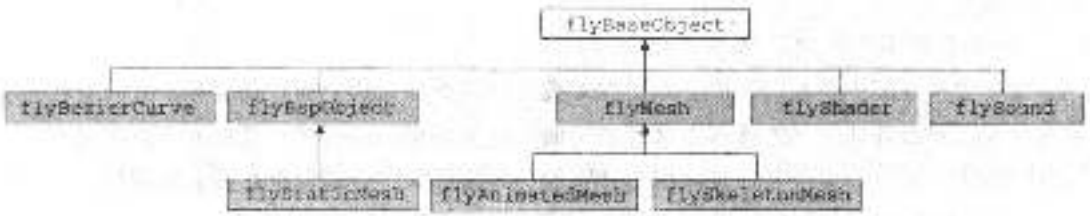


图 3-3　flyBaseObject 类依赖关系

```
class FLY_ENGINE_API flySound;
```

这个类实现了可以从一个.wav文件中装载的粗糙声音文件（见图3-4）。

```
class FLY_ENGINE_API flyPolygon;
```

这个类实现了具有任意数目的顶点的凸多边形。还有的方法可以裁剪用户生成的平面多边形并且可以重组顶点。

```
class FLY_ENGINE_API flyFace;
```

这个类（见图3-5）实现了引擎支持的任意类型的面，包括：

- 正常面（大面）
- 贝济埃面片
- 三角表面
- 三角带
- 三角扇

```
class FLY_ENGINE_API flyMesh;
```

这个类（见图3-6）实现了一个三维物体多边形网格。它的面涉及到局域面或全局引擎的BSP面。



图3-4　flySound类依赖关系



图3-5　flyFace类依赖关系

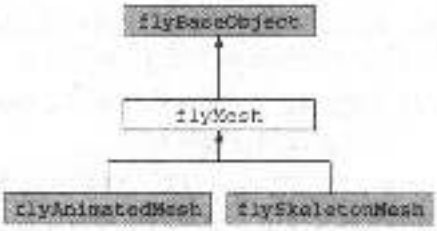


图3-6　flyMesh类依赖关系

```
class FLY_ENGINE_API flyAnimatedMesh;
```

这个类（见图3-7）实现了基于键点的顶点动画网格，建立了多动画混合的方法。

```
class FLY_ENGINE_API flySkeletonMesh;
```

这个类（见图3-8）实现了动画骨架网格，包含插入、动画混合和加皮肤方法。

```
class FLY_ENGINE_API flyBezierCurve;
```

贝济埃曲线是用这个类实现的（见图3-9）。世界坐标和切线都可以在曲线的任意点得到，也可以切分曲线直到曲线误差小于最大误差因子。（曲线可以有任意维数，如二维、三维、四维等。）

```
class FLY_ENGINE_API flyBezierPatch;
```

这个类实现了一个双二次或者双三次参数贝济埃面片。这个面片在u和v方向有任意数量的片段。它由好几个立方体或二次贝济埃面片连接而成。运行时，其表面可以在任意细节层次被渲染。照亮的面片是在最高层上计算的，其结果适应于任何正在被渲染的层。

nsu是u方向上面片的数目，nsv是v方向上面片的数目。

upu和upv是每个方向上控制点的数目。

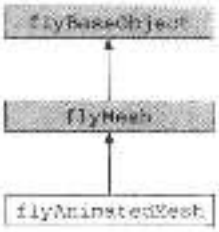


图 3-7　flyAnimatedMesh 类依赖关系

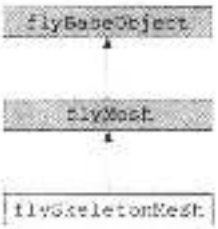


图 3-8　flySkeletonMesh 类依赖关系

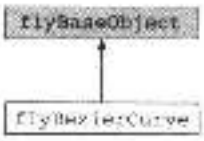


图 3-9　flyBezierCurve 类依赖关系

对二次面片：nsu = (npu − 2) /2　　nsv = (npv − 2) /2

对二次面片：nsu = (spu − 1) /2　　nsv = (spv − 1) /2

表面根据细分层次是离散的。u 和 v 方向上顶点的数目是：

nvertu = (1 << levelu)*nsu + 1　　nvertv = (1 << levelv)*nsv + 1

运行时，可以选定任意细节层次进行表面绘制。方法 set_detail 设置了所需的细节层次。每个方向上顶点的数目都发生了变化：

nvertskip = (1 << nleveldrop)

```
class FLY_ENGINE_API flyParticle;
```

单一的一个粒子（不可见）可以用这个类实现（见图 3-10）。

```
class FLY_ENGINE_API flyBspNode;
```

这个类（见图 3-11）实现了一棵 bsp 树上的一个节点。如果它没有子节点（child[0] = child_1] = 0），节点内部的元素被存在动态数组 elem 中。

```
class FLY_ENGINE_API flyBspObject;
```

这个类（见图 3-12）实现了运动的和被 BSP 树看作 AABB 的任何物体。

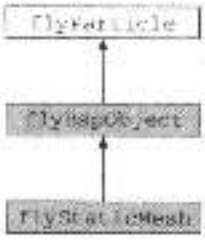


图 3-10　flyParticle 类依赖关系

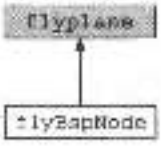


图 3-11　flyBspNode 类依赖关系

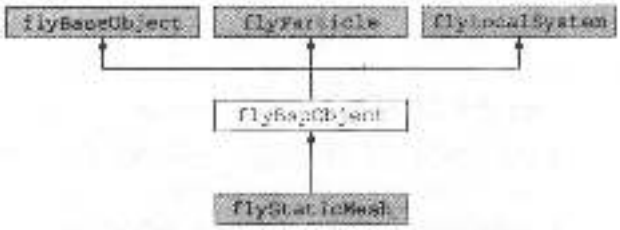


图 3-12　flyBspObject 类依赖关系

```
class FLY_ENGINE_API flyStaticMesh;
```

这个类（见图 3-13）实现了 BSP 叶节点中的面组。

```
class FLY_ENGINE_API flyLightMap;
```

这个类实现了一个光照贴图。有同样光照贴图的一些面共享同样的光照图像。

```
class FLY_ENGINE_API flyLightMapPic;
```

这个类实现了绘制时所需要的光照贴图纹理。这个纹理可以有许多光照图像，就像类 flyLightMap 的许多实例都会用到类 flyLightMapPic 一样。

```
class FLY_ENGINE_API flyLightVertex;
```

这个类持有动态硬件光照的信息。当一个动态物体接收到一个 FLY_OBJMESSAGE_ILLUM 消息时，它用 add_light 方法将光照参数加到这个类中。渲染时它在将网格绘制到获取硬件光照的位置前调用方法 init_draw。渲染结束后，它调用 end_draw 来重置硬件光照。

```
class FLY_ENGINE_API flyClassDesc;
```

这个元类实现了一个类描述，用于插件管理及对象初始化和引用。任何想被引擎识别和引用的插件类均应有这个类的一个子类，以及相应的类描述。

```
class FLY_ENGINE_API flyParamDesc;
```

这个类包含 Fly3D 插件对象参数的信息。每一个在 Fly3D 插件中定义的类（派生自 flyBspObject）都有任意数目的参数。每一个参数都在它的数据中包含一个 flyParamDesc 类。

```
class FLY_ENGINE_API flyConsole;
```

这个类实现了控制台，包含执行命令和命令行。控制台还可以在运行时输出信息，使用浏览键 "home"、"end"、"page up" 和 "page down" 可以浏览控制台消息。

```
class FLY_ENGINE_API flyShader;
```

这个类（见图 3-14）实现了一个多通路阴影。

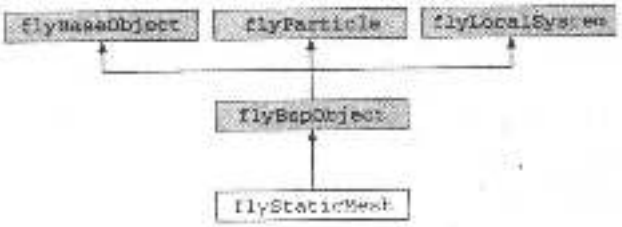


图 3-13　flyStaticMesh 类依赖关系



图 3-14　flyShader 类依赖关系

```
class FLY_ENGINE_API flyShaderFunc;
```

这个类实现了一个典型的阴影函数，它的类型有：正弦函数、三角函数、四边形、锯齿和反锯齿。

```
class FLY_ENGINE_API flyShaderPass;
```

这个类实现了一个阴影通道。一个阴影包含各种阴影通道。

```
class FLY_ENGINE_API flyInputMap;
```

这个类把键盘和鼠标输入与游戏动作对应起来。

```
class FLY_ENGINE_API flyFile;
```

这个类实现了一个 Fly3D 场景数据文件（.fly 文件）的控制器和界面。

```
class FLY_ENGINE_API flyDll;
```

这个类包含每一个 Fly3D 插件动态链接库的信息，包括动态链接库中实现的类的个数和指向动态链接库输出函数的指针。

```
class FLY_ENGINE_API flyDllGroup;
```

这个类实现了一组 Fly3D 插件的动态链接库。每一个插件的动态链接库都可以枚举任意数目的 flyBspObject 类。

```
class FLY_ENGINE_API flyEngine;
```

这个类中可获得的方法组成引擎功能的主界面。

为初始化和释放引擎，我们用下面两种全局方法：

```
// global engine initialisation method
FLY_ENGINE_API void fly_init_engine(HWND hWnd,HINSTANCE hInst,int
appid=FLY_APPID_NONE);
// global engine release method
FLY_ENGINE_API void fly_free_engine();
```

初始化这个构件之后，通过下面的全局变量就可以使用它了：

```
// global flyEngine instance
extern FLY_ENGINE_API flyEngine *g_flyengine;
```

加载和保存

加载以用户建立的配置文件开始，并用到工具 flyConfig，激活了特定机器背景的配置。

```
// Load Fly3D configuration file (Fly3D.ini)
int load_ini();
// Save Fly3D configuration file (Fly3D.ini)
int save_ini();
```

下面一系列的操作包括一个完全场景的加载：静态几何信息、BSP 树、PVS、光照贴图、纹理贴图、插件、动态物体（以及它们所有的资源，如声音、三维面片等）。这些都包括在一个 .fly 文件中，这个文件是一个包含所有这些资源的参照脚本（一个文本文件）。保存只包括对这个文本文件的存档。

```
// Open a .fly file
int open_fly_file(const char *file);
// Close a .fly file
void close_fly_file();
// Save to a .fly file
int save_fly_file(const char *file);
```

任何在游戏进行过程中的动态加载都用到下面这些方法。文件名要求得到一份资源，而一个指向这份资源的指针被送回。如果已经加载了这份资源，就返回指向这个现存物体的指针。

```
// Load a picture to the texture cache or return
// its index if already loaded
int get_picture(const char *file,int droplevel=0);
// Load a shader file
int load_shaders(const char *file);
// Get a mesh object
flyMesh *get_model_object(const char *name);
// Get a sound object
flySound *get_sound_object(const char *name);
// Get a bezier curve
flyBezierCurve *get_bezier_curve(const char *name);
```

场景更新

以如下方式激活场景更新：

```
// Update scene for elapsed time from last frame
int step();
// Update scene for elapsed time dt in milliseconds
void step(int dt);
```

仿真的主循环就是下面这一系列操作的执行：

1）查询 DirectX 界面对用户输入设备进行采样。

```
// get input
if (noinput)
  g_flydirectx->zero_input();
else g_flydirectx->get_input();
```

2) 所有在前一帧里根据动态光照改变的光照贴图都保存了在构建过程中计算得到的原值。

```
for( i=0;i<lmchanged.num;i++ )
{
  map=lm[lmchanged[i]];
  map->load(lmpic[map->pic]);
g_flytexcache->update_subtex(
  map->pic+lmbase,map->offsetx,map->offsety,
  map->sizex,map->sizey,3,map->bmp);
}
```

3) 所有在前一帧里改变的雾贴图都被清除了。雾贴图受雾的量的影响而改变。视见约束体在运动的时候会分割雾。这样，任何一个玩家都在雾中或透过雾看，雾贴图的像素就需要更新。

```
for( i=0;i<fmchanged.num;i++ )
{
  map=fm[fmchanged[i]];
  memset(map->bmp,0,map->bytesxy);
  g_flytexcache->update_subtex(
    map->pic+fmbase,map->offsetx,map->offsety,
    map->sizex,map->sizey,4,map->bmp);
}
```

4) 如果运行一台客户机，就会检测从上次玩家状态更新起经过的时间。如果这段时间比预定义的网络更新间隔（由网络连接速度决定）要长，就会将一个消息传给所有的插件，通知它们根据当前更新的状态发送消息。

```
// if in client multiplayer mode
if (g_flydirectx->mpmode==FLY_MP_CLIENT)
  {
  // update client objects to server at slower intervals
  static int last_mp_update=0;
  if (cur_time-last_mp_update>mpdelay)
    {
    last_mp_update=cur_time;
    dll.send_message(FLY_MESSAGE_MPUPDATE,0,0);
    }
  }
```

5) 现在所有动态物体都进行如下步骤：

for 每一个激活物体
  调用物体的虚拟阶跃函数
  if 物体的虚拟阶跃函数返回值为真 **then**
    从 BSP 树去掉，并将重新计算物体在新的位置裁剪的所有节点
  else 物体不移动
  if 物体的生命 < 0 **then**
    把物体从 BSP 树上去掉并销毁

6) 所有被物体阶跃函数（动态光照）改变的光照贴图都根据纹理管理器而被更新。

```
for( i=0;i<lmchanged.num;i++ )
  {
```

```
map=lm[lmchanged[i]];
g_flytexcache->update_subtex(
   map->pic+lmbase,map->offsetx,map->offsety,
   map->sizex,map->sizey,3,map->bmp);
}
```

7）相似地，所有被改变的雾贴图都要根据纹理管理器而被更新。

```
for( i=0;i<fmchanged.num;i++ )
   {
   map=fm[fmchanged[i]];
   g_flytexcache->update_subtex(
      map->pic+fmbase,map->offsetx,map->offsety,
      map->sizex,map->sizey,4,map->bmp);
   }
```

8）消息 FLYM_UPDATESCENE 被发送到所有加载了的插件。这使得这些插件更新它们的状态，并根据要求渲染它们的层面。

```
// step all running plug-ins
dll.send_message(FLY_MESSAGE_UPDATESCENE,dt,0);
```

9）如果在服务器或客户机上运行，check_multiplayer 被调用处理多用户的消息。这样询问消息队列中可得的消息。如果是一个系统消息（玩家加入、玩家退出等），它就会被处理。一个游戏应用消息通过插件的 fly_message 导出函数被传递给其他插件。这些插件将根据应用对消息作出相应的应答。

```
// if in multiplayer, check multiplayer messages
if (g_flydirectx->mpmode!=FLY_MP_NOMP)
   check_multiplayer();
```

场景绘制

draw_frame 的激活将导致所有绘制操作的执行。因此，“绘制”指对场景数据执行几何操作和渲染。至少调用一个 draw_bsp，引起所有选定的面都被传到 draw_faces（渲染发生的一个方法）。draw_faces 用阴影技术将面最好地排序来减小状态改变（见第 5 章）。

```
// Send drawing messages to all objects and draw
// everything in the current frame
void draw_frame();
// Sets the current camera to the given object
void set_camera(flyBspObject *d);
// Draw all scene elements viewed from the current camera
void draw_bsp();
// Draw the given faces
void draw_faces(int nfd,flyFace **fd,flyVertex *v,int sort=1);
```

draw_bsp 在一帧里可以被调用多次。比如，多视图是可行的，每一个需要它自己的 BSP 递归。同样，单独的面片物体也会调用 draw_faces 来用阴影渲染自己。

碰撞检测

碰撞检测（见第 2 章）可以被一个粒子激活，也可以被一个 AABB 激活。对一个粒子，我们用：

```
// Collision detection from p1 to p2, computes the closest collision
int collision_bsp(const flyVector& p1,const flyVector& p2,int
elemtype=0);
```

这返回有关 $p1$ 和 $p2$ 最近交叉点的信息。如果这个方法返回值为真，就可在下面的引擎公共变量中获得像碰撞法线之类的碰撞信息。

```
// ray intersection data
```

```
flyBspObject *hitobj;          // current hit object
flyMesh *hitmesh;              // current hit mesh
int   hitface,                 // current hit face
      hitsubface,              // current hit subface
      hitshader;               // current hit face shader
flyVector    hitip,            // last collision intersection point
      hitnormal;               // last collision hit normal
float hitdist;                 // last collision hit distance
```

另外，下面的方法返回 $p1$ 和 $p2$ 之间的第一次碰撞，但不能得到关于碰撞的信息：

```
// Collision detection from p1 to p2, returns when finds the first
// collision
int collision_test(const flyVector& p1,const flyVector& p2,int
elemtype=0);
```

这比第一种方法快，在我们仅需要知道碰撞是否发生的时候采用。我们也可以用参数 elemtype（如果为 0，所有元素类型都被检查）把碰撞限制在某一些游戏元素类型。

AABB 碰撞检测被放置在类 flyBoundBox 和类 flyBspObject 中。

BSP 树的一般递归

一般递归实现了基于栈的有效递归（第 1 章里详细介绍的）。用法如下：

```
// Recurse the BSP selecting nodes and objects clipped by the
sphere centred at p with radius rad
void recurse_bsp(const flyVector& p,float rad,int elemtype,int
pvsleaf=-1);

// Recurse the BSP selecting nodes and objects between p1 and p2
void recurse_bsp(const flyVector& p1,const flyVector& p2,int
elemtype,int pvsleaf=-1);

// Recurse the BSP selecting nodes and objects clipped by the
volume defined by the array of points p
void recurse_bsp(const flyVector *p,int np,int elemtype,int
pvsleaf=-1);

// Recurse the BSP selecting nodes and objects clipped by the box
defined by min and max
void recurse_bsp_box(const flyVector& min,const flyVector& max,int
elemtype,int pvsleaf=-1);
```

所有的选择都被限制在一个特定的物体类型，对节点裁剪使用 PVS。

插件消息

插件是一组物体，它们之间有两种类型的通信——物体消息和插件消息。物体消息在虚拟函数 flyBspObject 的消息方法里处理。插件消息在 fly _message 函数输出的动态链接库中处理。消息被送到物体，激活一个一般性的 BSP 递归。

```
// Recurse the BSP sending messages to all selected objects
void send_bsp_message(const flyVector& p,float rad,int msg,int
param,void *data,int elemtype=0,int pvsleaf=-1);

// Send a message to all plugin DLLs
int send_message(int msg,int param,void *data) const;
```

函数 send_message 将消息发给所有当前场景中的插件动态链接库，同时调用所有动态链接库的 fly_message 导出函数。send_bsp_message 给特别的影响域中所有 flyBspObject 衍生的类发出一个消息。

添加复制和激活物体

在仿真过程中，许多物体都可以被插入到场景中。比如，当一个射击玩家开火时，一个射弹必须被加进游戏，包括它的起始位置（根据枪的位置）和速度（根据射弹的特性）。一旦它撞到了一个表面（敌人或墙），射弹必须从游戏中去掉，一个爆炸物体可能被插到射弹碰撞的地方。当然，同一物体的许多复制品（比如枪的射弹）在同一时间可能共存于仿真中。因此，以这种方法控制着物体的加入和删除：每一个复件都有自己的参数值，并且被独立控制。同样，当一个复件被插入到仿真中时，它必须保留这种类型物体的原始值，即先前插入的物体不能改变这个原始值。

这种一致性是通过总是保存一种有原始属性和参数值的物体达成的。这样，要把一个物体的新的复件加入仿真，引擎就使用克隆函数制造保存了的物体的一个复件，然后把这个复件插入到环境中。被添加的这个复件叫做激活物体。插入物体的这个行为叫激活（activation）。这样，同一个物体的好几个激活复件就会有它们自己的不断变化的参数值，而原始保存的物体不会变化，这样物体新复件的激活可以靠克隆一个原始物体来实现。

在执行一个 open_fly_file 命令时，由一个 .fly 文件定义的所有物体都被加载到保存物体的链表中。当保存一个 .fly 文件时，保存物体链表中所有物体的性质都被保存到文件中。

任何保存的物体都可以用类 flyEngine 中的如下命令激活（被克隆并添加到 BSP 中）。

```
// Activate an object from the stock
void activate(flyBspObject *d);

// Add an object to the BSP tree
void add_to_bsp(flyBspObject *obj);
```

这会把物体加到激活物体的链表的尾部，同时被加进 BSP 树。激活物体的链表中所有的物体都有它们自己的阶跃函数，这些函数每帧都被调用，如果在当前视见约束体内且不被 PVS 挡住，就被选择绘制。

场景查询

要查询物体所在的场景，我们需要一个物体指针，或者在物体中寻找一种特定的类型。我们也可以通过保存物体的链表或激活物体的链表查询物体。

```
// Get the stock object with the given name
flyBspObject *get_stock_object(const char *name) const;
// Get an active object with the given name
flyBspObject *get_active_object(const char *name) const;
// Get the stock object immediately after 'o' in the respective
array
flyBspObject *get_next_stock_object(flyBspObject *o,int type=0) const;
// Get the active object immediately after 'o' in the respective
array
flyBspObject *get_next_active_object(flyBspObject *o,int type=0) const;
```

物体参数设置和查询

引擎中每一个参数都用一个字符串表示。有了物体的名字和参数名，就可以查询到包含了参数值的字符串，我们也可以从现有的字符串传递参数值。同样，也可以查询到全局引擎参数并以相同的方式设定它们。

```
// Set an object's parameter value
int set_obj_param(const char *objname,const char *param,const char
*value);
```

```
// Get an object's parameter value
int get_obj_param(const char *objname,const char *param,flyString&
value) const;
// Set a global parameter value
int set_global_param(const char *name,const char *value);
// Get the value of a non-scene dependant global parameter
int get_global_param_desc1(int i,flyParamDesc *pd);
// Get the value of a scene dependant global parameter
int get_global_param_desc2(int i,flyParamDesc *pd);
```

混合方法

我们以列举一些混合方法来结束这个部分。第一种在控制台上用标准的 C++ 格式输出一个文本字符串。

```
// Print a string out on the console
void console_printf(const char *fmt, ...);
```

接下来两种方法用来得到对应的输入控制——键盘输入和鼠标输入。一个输入可以通过名字来查询——比如 walk _forward，并返回一个句柄。这个句柄再被用于 check _input _map 中，传递检验输入的当前状态的句柄。flyConfig 被用来指定键盘输入和鼠标按键对应的字符串游戏行为。

```
// Get a input map
int get_input_map(const char *name);
// Checks the given input map for input
int check_input_map(int i);
```

最后一种方法重现了 BSP 树，以找出包含指定点的节点。

```
// Recurse the BSP and return the node where the point p is located
flyBspNode *find_node(const flyVector& p) const;
```

附录 3.1　编写一个插件

1. 介绍

这个编程指导介绍了 Fly3D 插件的开发。这是一个简单插件的经典例子，它用了引擎的一些功能，并显示了用 Fly3D 的典型技术。

下面的步骤包括必要工具的安装、插件的简单开始和把简单模块复杂化的改善。

读这个指导需要对 C++ 有一定的熟练程度，并对 Visual C++ 有一定的了解。

2. 安装 Fly3D、3DSMax 插件和 Visual C++ 向导

Fly3D 的安装过程如下：运行安装可执行程序，按照安装指导选择安装文件夹等。

拷贝完所有文件后，flyConfig.exe 就会自动运行（见图 A3-1），用户可以选择一个图形渲染器，设置一些渲染选项、配置文件和键盘设置等。对于渲染器，应该选择加速模式。

保存配置，flyConfig.exe 关闭后，插件的安装程序就会自动运行（见图 A3-2）。这个软件允许用户为 3D Studio MAX 3.x、4.x 和 Visual C++ 插件向导安装 Fly3D 的输入输出插件。安装 MAX 插件时，必须选择正确的 3DSMax 版本和安装路径。而对于插件向导，只需选择 Visual C++ 路径。

安装好后，在安装过程全部结束以后，用户必须立即重启系统。

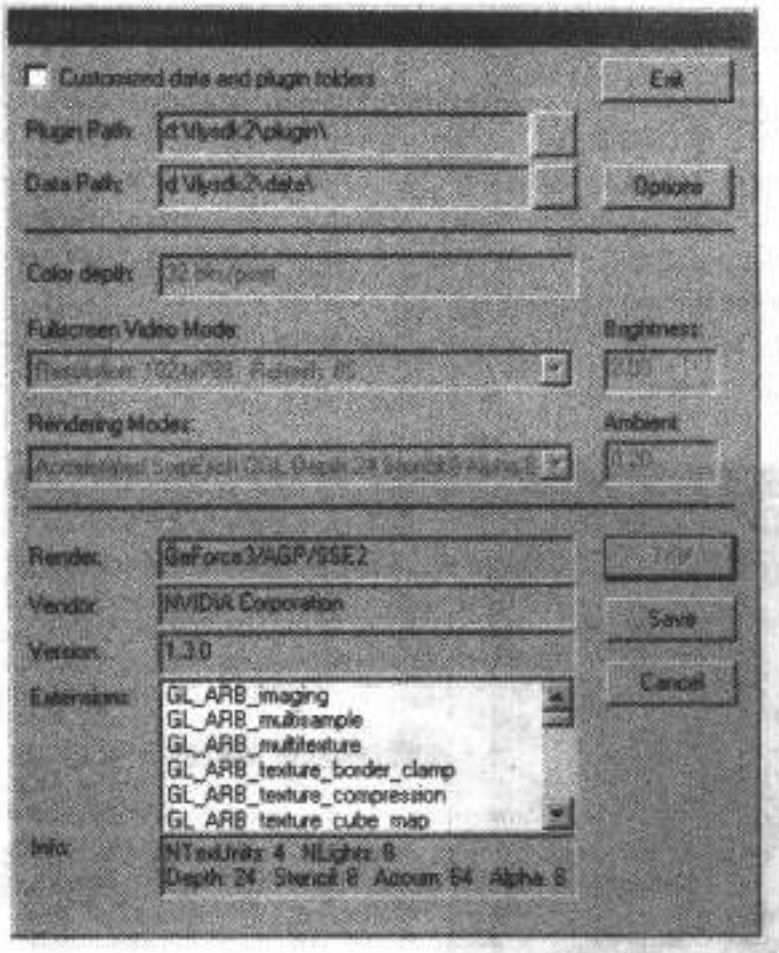


图 A3-1 Fly3D 配置

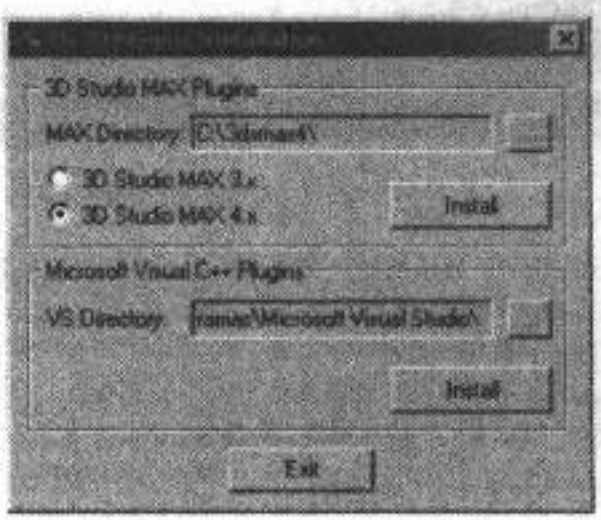


图 A3-2 Fly3D 插件安装

3. 用 Visual C++向导制作一个 Fly3D 插件，实现两个类：照相机和对象

在这一步中，将用 Visual C++插件向导从头创建一个新的 Fly3D 插件。

首先，必须运行 Visual C++。在“file”菜单中，下菜单“new”将打开一个选择创建某类型文档的窗口。Fly3D 插件向导会在“project”标签中被枚举。选择它，为新工程取一个名字，选择一个保存这个工程的文件夹。如果 Fly3D 插件不在列表中，重新运行插件的安装程序，正确地安装向导。在 Fly3D 安装目录下的“util”文件夹中就可以找到 flyInstPlugins.exe。参考第 2 步中更多的安装信息。

在插件向导的第 1 步中（见图 A3-3），必须给类命名，并添加到插件中。在指导实例中，对象和照相机两个类被创建。

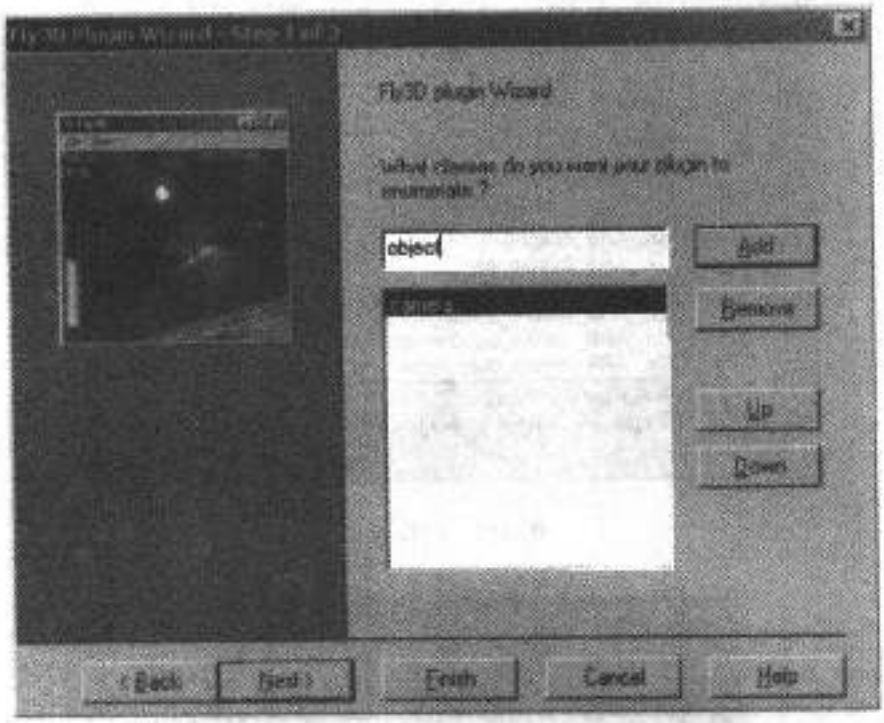


图 A3-3　Fly3D 插件向导——第 1 步

在第 2 步中（见图 A3-4），枚举插件中类输出的所有参数，包括名字、类型、默认值。每一个参数都必须具有 Fly3D 预先定义的类型中的一种（在相应的组合框中枚举的）。

在我们的指导实例中，照相机类有两个参数：“mousevel”（定义了鼠标指针速度的浮点指针）和“movevel”（定义了照相机移动速度的另一个浮点指针）。对象类只有一个参数：“3d static mesh”类型的“objmesh”，即对象的三角网格。

4. 用 flyEditor 制作包含新的插件的 .fly 文件

一个插件被成功创建以后，下面的步骤包括创建一个新的 .fly 文件，并用 flyEditor 的前

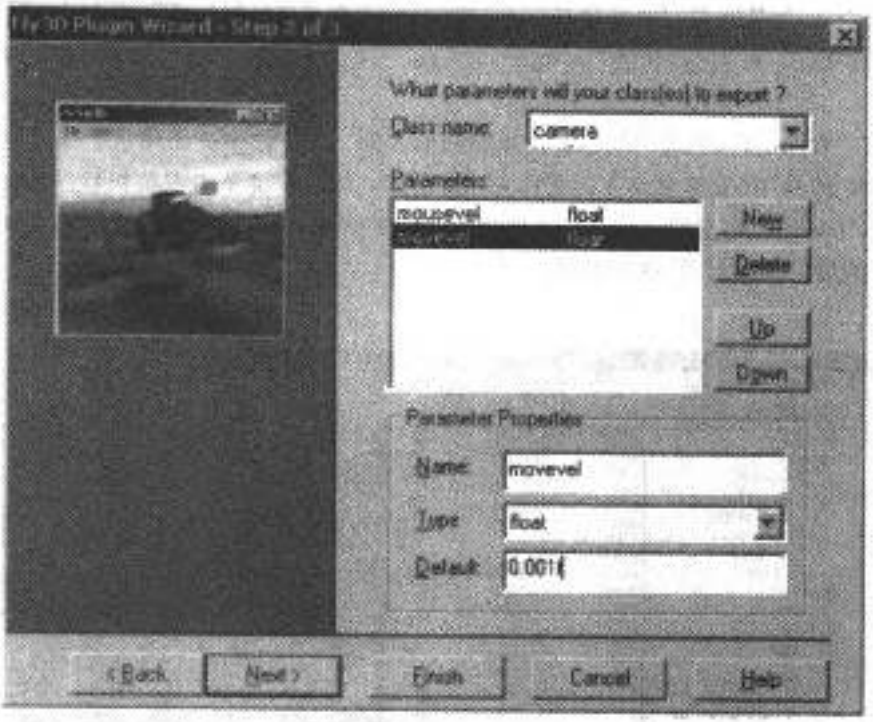


图 A3-4　Fly3D 插件向导——第 2 步

将它包含进插件中。

首先运行 flyEditor.exe（见图 A3-5），把它放置在安装 Fly3D 的文件夹里。在“file”菜单的“save”子菜单保存一个 .fly 文件，要把一个 BSP 文件加载到场景中，保存这一步是必需的。

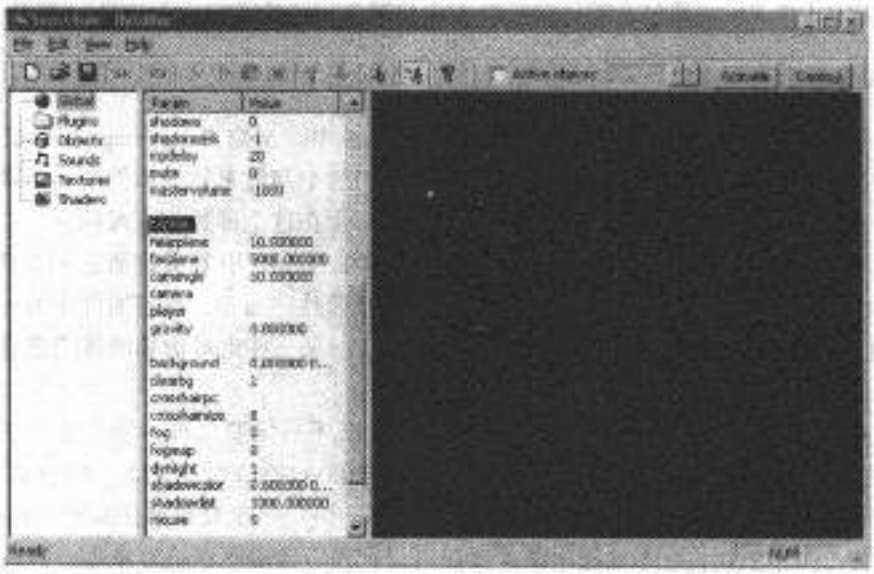
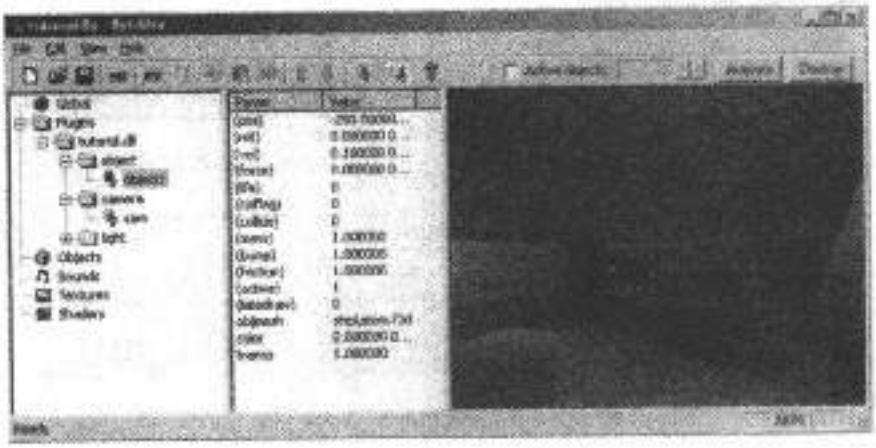


图 A3-5　flyEditor

保存了.fly文件之后，必须加载一个BSP文件。这一步通过编辑“Global”组中的“bspfile”参数实现。（在左边的树状图中选择“Global”项，然后在中间的列表视图中查找“bspfile”。）

现在，“Tutorial1”生成的动态链接库必须插入到.fly文件中。编译这个插件，将所得的dll文件拷贝到Fly3D安装目录下的文件夹“plugin”。然后右键单击左边树状图中的“plugin”项，选择“Insert”，将它插入到.fly文件中。一个对话框出现，这个dll文件被关闭。FlyEditor中文件的最后画面如图A3-6所示。



图A3-6　FlyEditor和游戏

向导的第3步包含Fly3D安装路径的简单选取。

最后，Fly3D插件向导必须在你的工程中创建3个文件：“Tutorial1.h”、“Tutorial1.cpp”和“Tutorial1.def”。头文件包含插件向导第1步和第2步中枚举的所有信息：类和参数。

5. 处理引擎的全局消息

在指导插件向导的.cpp文件生成的所有全局方法中，函数fly_message需要引起特别的注意。当帧被更新、绘制、场景结束的时候，引擎用这个函数来通知插件诸如场景初始化之类的事件。任何希望对这些事件作出响应的插件都必须在这个函数中实现响应。

在仿真的时候，无论什么时候更新状态或绘制帧，Fly3D引擎发送消息FLY_MESSAGE_UPDATESCENE和FLY_MESSAGE_DRAWSCENE给这些插件。通常，这些插件中的一个必须接收消息FLY_MESSAGE_DRAWSCENE并对其作出响应，从一些点对仿真的视图进行绘制，否则什么都画不出来。

在指导的例子中，消息FLY_MESSAGE_INITSCENE不一定要实现，因为这个插件不需要对场景进行初始化。同样，消息FLY_MESSAGE_UPDATESCENE、FLY_MESSAGE_DRAWTEXT、FLY_MESSAGE_CLOSESCENE也不一定要实现。唯一一定要被插件实现的消息是FLY_MESSAGE_DRAWSCENE，它必须把引擎的照相机设置到需要的照相机对象上，并告诉引擎从照相机的地方开始绘制场景。实现这些的函数fly_message的代码如下：

```
__declspec( dllexport )
int fly_message(int msg,int param,void *data)
{
   switch(msg)
   {
      case FLY_MESSAGE_INITSCENE:
      break;
      case FLY_MESSAGE_UPDATESCENE:
      break;
      case FLY_MESSAGE_DRAWSCENE:
      {
         g_flyengine->set_camera(g_flyengine->cam);
         g_flyengine->draw_bsp();
      }
      break;
      case FLY_MESSAGE_DRAWTEXT:
      break;
      case FLY_MESSAGE_CLOSESCENE:
      break;
   }

   return 1;
}
```

6. 实现对象的方法

在这一步中，对象类方法将被实现。这些方法基本上都是继承自类 flyBspObject 的虚拟函数。

初始化

```
void object::init()
```

这是对象的初始化函数。在我们的小例子中，它只需要初始化对象的轴向对齐的包围盒(每一个 Fly3D 实体都有一个)。因为每个对象都有一个网格，对象的 AABB 必须是属于网格的。我们的初始化函数有这样的代码：

```
void object::init()
{
   if(objmesh)
      bbox=objmesh->bbox;
}
```

对对象初始化方法的小修改参考第 8 步。

步骤

```
int object::step(int dt)
```

这是每帧更新的函数。在仿真过程中，每帧被调用一次，而每一个实体都必须在这个函数中更新它的状态。在这个例子中，对象不发生变化。所以没有必要在这个函数中实现所有的东西：只要返回一个有效值（对有效物体是 1)。对象阶跃函数的修改参考第 9 步和第 11 步。

```
int object::step(int dt)
{
   return 1;
}
```

绘制

```
void object::draw()
```

这是绘制函数。它被用来用 OpenGL 调用绘制物体。在指导的例子中，实现了一个简单

的绘制：对象被变换到三维空间的位置上（根据变量 pos），旋转到现在的方位（根据变量 mat），网格就是这样绘制的。代码如下：

```
void object::draw()
{
  if(objmesh)
  {
    glPushMatrix();
    glTranslatef(pos.x,pos.y,pos.z);
    glMultMatrixf((float *)&mat);
    objmesh->draw();
    glPopMatrix();
  }
}
```

消息接收

```
int object::message(const flyVector& p,float rad,int msg,int
param,void *data)
```

这是消息接收函数。它必须根据已知消息类型检验参数 msg 的值。现在，这个例子的简单对象不会收到任何消息，返回 1 作为普通函数结束的标志。给对象的消息函数添加消息处理请参考步骤 11。

```
int object::message(const flyVector& p,float rad,int msg,int
param,void *data)
{
  return 1;
}
```

获得自定义参数描述

```
int object::get_custom_param_desc(int i,flyParamDesc *pd)
```

这是前端根据插件的类输出变量获得信息的函数。它总是返回类中输出变量的总数（在本例中是 1），改变第 i 个参数，用所需的 type、data 指针和 name 填写 flyParamDesc 实例 pd。type 必须是一个代表 Fly3D 默认参数类型的字符（在 Fly3D 目录下文件夹 util 文档 class_typeid.txt 中有描述）；data 必须是一个指向变量自身的指针；name 必须是前端使用的参数名字符串。代码如下：

```
int object::get_custom_param_desc(int i,flyParamDesc *pd)
{
  if (pd!=0)
    switch(i)
    {
      case 0:
        pd->type='m';
        pd->data=&objmesh;
        pd->name="objmesh";
      break;
    }

  return 1;
}
```

对象方法 get_custom_param_desc 的补充请参考步骤 8。

7. 实现照相机的方法

在这一步中，照相机类的方法被实现。这些方法基本上都是继承自类 flyBspObject 的虚

拟函数。

初始化

```
void camera::init()
```

这是照相机的初始化函数。和对象类一样，我们只需初始化照相机轴向对齐包围盒(AABB)。但因为照相机没有网格，必须创建一个 AABB。在下面的代码中，照相机的 AABB 用 50 来初始化。

```
void camera::init()
{
  bbox.min.vec(-25,-25,-25);
  bbox.max.vec(25,25,25);
}
```

步骤

```
int camera::step(int dt)
```

这是每帧更新的函数。在这种方法中，照相机必须检测输入并更新它的位置（移动 dt 毫秒）。在这个例子中，照相机是这样移动的：鼠标左键扮演一个加速器（按下时，照相机向前移动），鼠标轴用作旋转（俯仰和偏航）。下面的代码可以被分成三部分：

1）获得类 flyDirectX 的变量 mouse_down，常量值 FLY_MOUSE_L 被用来检测鼠标左键自上一帧以来是否被按；如果测试成功，照相机速度将根据其面对的方向（Z 轴负方向）、移动速度和逝去的时间增加一个值；如果测试失败，照相机的速度不变。

2）鼠标轴由类 flyDirectX 的变量 mouse_smooth 来测试（平滑鼠标运动）。注意，这个变量是分别对应 X、Y 鼠标轴的两个位置数组。如果测试成功，照相机根据自上一帧以来鼠标移动的速度和鼠标指向的位置绕其他轴旋转。Y 轴旋转的负值是鼠标旋转的反方向。

3）照相机必须移动。这一部分代码将在步骤 9 中解释，照相机的移动和碰撞避免与对象的是一样的。

```
int camera::step(int dt)
{
  if (g_flydirectx->mouse_down&FLY_MOUSE_L)
    vel=Z*movevel*(float)(-dt);
  else
    vel.null();

  if (g_flydirectx->mouse_smooth[0]) // mouse X
    rotate(-g_flydirectx->mouse_smooth[0]*mousevel,Y);

  if (g_flydirectx->mouse_smooth[1]) // mouse Y
    rotate(g_flydirectx->mouse_smooth[1]*mousevel,X);

  flyVector p,v;
  p=pos+vel*(float)dt;
  v=vel+force*((float)dt/mass);
  box_collision(p,v);
  pos = p;
  vel = v;

  return 1;
}
```

绘制

```
void camera::draw()
```

这是绘制函数。因为照相机不会出现在仿真中，所以这里没有绘制的必要，方法为空。

消息接收

```
int camera::message(const flyVector& p,float rad,int msg,int
param,void *data)
```

这是消息接收函数。像对象一样，照相机不接收任何消息，对于正常结束，方法返回值为 1。

```
int camera::message(const flyVector& p,float rad,int msg,int
param,void *data)
{
  return 1;
}
```

获得自定义参数描述

```
int camera::get_custom_param_desc(int i,flyParamDesc *pd)
```

这是前端根据插件的类输出变量获得信息的函数。它和同名对象的方法相似，最后一步将给出完整的解释。代码如下：

```
int camera::get_custom_param_desc(int i,flyParamDesc *pd)
{
  if (pd!=0)
    switch(i)
    {
      case 0:
        pd->type='f';
        pd->data=&movevel;
        pd->name="movevel";
      break;
      case 1:
        pd->type='f';
        pd->data=&mousevel;
        pd->name="mousevel";
      break;
    }
  return 2;
}
```

8. 给插件类添加新的成员变量

在这一步中，将以插件中已有类的附加变量为例。一个新的变量将被添加到对象类：网格颜色。

首先，变量必须在 .h 文件中被添加到对象类的定义里。下面这行代码被添加：

```
flyVector color;
```

类 flyVector 定义了一个四维的浮点指针向量。一种颜色由向量的 **x**，**y**，**z** 分量表示，**w** 用来表示透明度（0 到 1，0 表示完全透明，1 表示完全不透明）。

对于这些值要在前端里输出和编辑的变量，它们必须在方法 get_custom_param_desc 中枚举。经此方法改变后的代码如下：

```
int object::get_custom_param_desc(int i,flyParamDesc *pd)
{
  if (pd!=0)
    switch(i)
    {
      case 0:
        pd->type='m';
```

```
            pd->data=&objmesh;
            pd->name="objmesh";
          break;
          case 1:
            pd->type='c';
            pd->data=&color;
            pd->name="color";
          break;
          case 2:
            pd->type='f';
            pd->data=&color.w;
            pd->name="transp";
          break;
        }

      return 3;
    }
```

注意，数据类型 c 表示颜色，flyEditor 将根据要编辑的参数类型打开一个颜色浏览框。对于透明变量，将使用一个标准的浮点指针编辑框。

既然网格颜色在插件外已经被编辑了，它必须被指定给对象的网格颜色。类 flyMesh 有一个需要设置对象颜色参数的颜色成员变量。这将在对象初始化方法中完成，具体如下：

```
void object::init()
{
  if(objmesh)
  {
    bbox=objmesh->bbox;
    objmesh->color=color;
  }
}
```

9．给对象类添加运动和碰撞检测

这一步以对象类的移动仿真和碰撞检测为例。对象以恒定加速度移动，并从墙上反弹回来，它的速度向量用从 flyParticle（通过 flyBspObject）继承而来的变量 vel 表示。碰撞由从 flyBspObject 继承而来的 box_collision 处理，flyBspObject 实现了碰撞检测和由对象轴向对齐包围盒的反弹。

代码结构如下：首先，更新对象的位置和速度——根据速度和时间更新位置，根据受力、质量、时间更新速度。然后，调用方法 box_collision 来处理碰撞检测和自动反弹（根据对象的 bump 和 friction 参数反弹，由 flyBspObject 继承而来），更新传递给它的位置和速度参数。最后，这些值被指定给对象的初始位置和速度。所有这些被插入到对象的 step 方法中：

```
int object::step(int dt)
{
  flyVector p,v;
  p=pos+vel*(float)dt;
  v=vel+force*((float)dt/mass);
  box_collision(p,v);
  pos = p;
  vel = v;

  return 1;
}
```

10. 给插件添加一个新类：光照

在这一步中，一个新类被添加到指导中的插件里。因为下一步是把光照加入仿真中，新类必须有一个光源。

首先，一个新的类型必须加到 .h 文件顶部枚举的类型中。这个类型称为 TYPE_LIGHT。插件中每一个类都有一个相应的类型。枚举如下面所示：

```
enum
{
  TYPE_OBJECT=100000,
  TYPE_CAMERA,
  TYPE_LIGHT,
};
```

现在定义光照类。像场景中所有其他类一样，它必须是类 flyBspObject 的衍生类。光照类有两个成员变量：color 和 illumradius，分别表示光照的颜色和半径。同样，一个构造器和一个复制构造器被生成。构造器给成员变量指定初始值，并设置变量 type 为以前定义的类型（从 flyBspObject 继承而来）；复制构造器调用 flyBspObject 的复制构造器，根据被复制的原对象给成员变量赋值。所有的 flyBspObject 纯虚函数都必须实现，整个光照类的定义如下：

```
class light : public flyBspObject
{
public:
  flyVector color;
  float illumradius;
  light() :
    color(1),
    illumradius(100)
  { type=TYPE_LIGHT; }

  light(const light& in) :
    flyBspObject(in),
    color(in.color),
    illumradius(in.illumradius)
  { }

  virtual ~light()
  { }

  void init();
  int step(int dt);
  void draw();
  int get_custom_param_desc(int i,flyParamDesc *pd);

  flyBspObject *clone()
  { return new light(*this); }
};
```

另一个重要步骤就是为新建的光照类创建描述类。描述类允许外部模块查看插件里的类和它们的输出成员。所有的描述类都很相似，光照的描述类和照相机的描述类或对象的描述类也很相似。

```
class light_desc : public flyClassDesc
{
public:
  flyBspObject *create() { return new light; };
```

```
  const char *get_name() { return "light"; };
  int get_type() { return TYPE_LIGHT; };
};
```

在上面的类中，方法 create 必须返回描述类中的一个新实例；方法 get_name 必须返回一个包含友类名字的字符串；方法 get_type 必须返回定义类的类型。

描述类生成以后，插件的输出方法必须修改为包含这个新光照类。两个方法都要修改：方法 num_classes（返回插件中类的总数）和方法 get_class_desc（返回一个指向插件中描述类的指针）。具体如下：

```
__declspec( dllexport )
int num_classes()
{
  return 3;
}
__declspec( dllexport )
flyClassDesc *get_class_desc(int i)
{
  switch(i)
  {
    case 0:
      return &cd_object;
    case 1:
      return &cd_camera;
    case 2:
      return &cd_light;
  }

  return 0;
}
```

最后，光照类的方法必须实现。这非常简单：init 方法为光照创建一个轴向对齐包围盒；方法 step 和方法 draw 什么都不做；方法 get _custom _param _desc 枚举类中的两个成员变量。它们具体实现如下：

```
void light::init()
{
  bbox.max.vec(-10,-10,-10);
  bbox.min.vec(10,10,10);
}

int light::step(int dt)
{
  return 0;
}

void light::draw()
{

}

int light::get_custom_param_desc(int i,flyParamDesc *pd)
{
  if (pd!=0)
    switch(i)
    {
      case 0:
        pd->type='c';
```

```
            pd->data=&color;
            pd->name="color";
          break;
          case 1:
            pd->type='f';
            pd->data=&illumradius;
            pd->name="illumradius";
          break;
        }
    return 2;
}
```

下一步是有关给仿真添加光照和阴影，并用新建的光照类表示光照。

11. 给对象添加动态光照和动态阴影

指导的最后一步是实现对象类的光照和阴影，这样物体的网格就被照亮并能投射阴影。

首先，成员变量 flyLightVertex 被添加到对象类中。flyLightVertex 是 Fly3D 引擎内部的一个类，它实现了动态光照数组。在对象类定义中作如下声明：

```
flyLightVertex dynlights;
```

现在对象在它周围的环境中搜索照亮它的光源。这是通过引擎的方法 recurse_bsp 在对象的阶跃函数中实现的：

```
int object::step(int dt)
{
  flyVector p,v;
  p=pos+vel*(float)dt;
  v=vel+force*((float)dt/mass);
  box_collision(p,v);
  pos = p;
  vel = v;

  g_flyengine->recurse_bsp(pos,2048,TYPE_LIGHT);
  for(int i=0;i<g_flyengine->selobjs.num;i++ )
  {
    light *l=(light *)g_flyengine->selobjs[i];

    if (g_flyengine->collision_test(pos,l-
>pos,FLY_TYPE_STATICMESH)==0)
      dynlights.add_light(l->pos,l->color,l->illumradius);
  }

  return 1;
}
```

注意，在光源和对象之间存在墙（FLY_TYPE_STATICMESH）的时候，调用 collision_test 检测。

如果对象已经找到了它周围的光源，它就一定会被光源照亮。这是用 flyLightVertex 的方法 init_draw 在对象绘制函数中实现的：

```
void object::draw()
{
  if(objmesh)
  {
    dynlights.init_draw(this);

    glPushMatrix();
```

```
        glTranslatef(pos.x,pos.y,pos.z);
        glMultMatrixf((float *)&mat);
        objmesh->draw();
        glPopMatrix();

        glDisable(GL_LIGHTING);
    }
}
```

上面添加的两行代码对被变量 dynlights 存储的动态光源照亮的对象来说是足够的。最后一项任务就是阴影投射，由对象类来实现。它通过重载 flyBspObject 的虚拟函数 draw_shadow 实现：

```
void object::draw_shadow()
{
    int i=dynlights.get_closest(pos);
    if (i!=-1)
    {
        glPushMatrix();
        glTranslatef(pos.x,pos.y,pos.z);
        glMultMatrixf((float *)&mat);
        flyVector v=dynlights.pos[i]-pos;
        v.normalize();
        objmesh->draw_shadow_volume(v*mat_t);
        glPopMatrix();
    }
}
```

方法 get_closest 在数组中找到最近的光源；方法 draw_shadow_volume 根据已知位置和方向绘制网格的阴影。

最后，关于对象搜索周围光源要注意一点。还有一种决定对哪种对象应用哪种光源的方法：光照向位于半径定义球体内（包括其自身位置）的所有对象发送消息。当光源数目比被照亮的对象数目少时，这种方法比较适用。如果不是处于这种情况，则应该用指导中实现它的方法，即让对象寻找自己周围的光源。

第二部分 实时渲染

第4章 实时渲染

在这里我们分两章来对渲染进行说明，一章侧重于理论（本章），另一章则侧重于实际应用（第5章）。使GPU可编程的最新硬件的发展便是一个特殊的论题。正因如此，虽然对本章所讨论的理论技术的选择很大程度上受到了硬件模型的影响，我们还是认为把理论同实际操作分开讨论比较好。本章着重点在于用实时的逐像素着色来实现高质量的渲染，而这依赖于纹理映射硬件的扩展。本章中我们较多地从映射可实现技术的角度来讨论渲染理论。

4.1 简介

在本书撰写的时候（2002年），我们似乎已经处在了一个时代的边缘，许多先进的渲染技术以前只能在离线工具中使用，现在已经可以在交互式工具中实现。这都要归功于现在在用户的硬件上可以轻松实现大规模图形处理的能力。而且这其中很大一部分是伴随着这样一个转变而来的——即把渲染技术从CPU转移到可编程的GPU上。我们将在第5章中讨论这一很重要的新进展。这些新进展的出现都要归功于游戏市场的巨大需求，本章将仔细考察这些渲染技术背后的理论是如何同游戏应用结合在一起的。

在20世纪90年代有一个可用的3D电脑游戏技术，就是光照贴图形式下预计算技术的应用。这种技术使人们可以使用复杂阴面也是很有趣的层次，通过这些层次一名枪手可以四处游荡并且还可以遇见他的对手。不过光照贴图还要受到一些限制。作为预计算技术，它们只能照亮静止的物体，而对运动物体的着色技术还局限在Gouraud着色和一些简单的实现（譬如物体一边移动一边“拾取”周围环境的光线）这样的阶段。本章将考察许多不同的着色方式，通过这些方式，最新的硬件发展被用来高效地对动态物体进行着色。我们特别感兴趣的是被称为“逐像素着色”的方法，这种方法可以对每个像素使用互不相关的着色计算——就像在Phong镜面项中那样。这个项用来区分逐像素着色和逐顶点着色。这种处理方法只在一个顶点进行着色计算，然后使用硬件插值来产生像素的强度（Gouraud着色）。

我们同时应该注意到，在最近对于离线渲染研究的一段时间中，许多研究都着重于全局的照明方法，比如光线跟踪、辐射度以及对渲染方程式求一般解。这些研究在过去甚至现在仍然为这样一个主张所驱动，即为了追求照片级的真实感，我们必须找到一个全局照明的解决方案。（众所周知，当前对全局照明的解决方案都超出了用户硬件的处理能力，至少对实时渲染而言就是如此。）然而事实上90年代在Pixar电影产品中取得的高质量着色效果仅仅依赖于RenderMan着色技术——一个不用执行全局照明的API——本章中我们要从交互速度的角度考察这种基于局部反射模型技术所产生的高质量效果。还应该注意的是，许多景象严格地说是全局照明的结果，可以通过成本较低的机制来逼近，而这些机制可以和实时工具结

合在一起。环境映射和阴影算法是这种机制很好的例子。而且可以肯定的是，这些工具在一个全局照明可行的解决方案出现之前还会继续使用一段时间。

促使我们使用高级渲染技术的动力当然就是影像质量。人们经常肤浅地从几何复杂度的角度来讨论影像质量，而几何复杂度本身又被理解为多边形/物体的数目。这个争论引出了一个同样肤浅的假设，认为影像的质量问题可以通过运算能力日益提高的硬件得到解决。然而多边形的数目只是影响影像质量的一个因素。这可以通过考察 Pixar 的两部卡通作品《玩具总动员》（Toy Story）（1995）和《玩具总动员 2》（Toy Story 2）（1999）看出来。比较这两部产品，Pixar 在同时期的硬件上重新渲染了这两部电影的各帧。对所有帧的渲染时间取平均后发现，后一部电影的处理时间是前者的 5 倍。然而，《玩具总动员 2》只是在前作的基础上将“多边形/帧”的处理复杂度提高了一倍而已……

4.2　顶点、像素和贴图

新的硬件使程序员能够在 GPU 上对顶点处理和像素处理进行控制，第 5 章中将详细考察这些硬件。这些设备也就是我们所称的顶点着色器和像素着色器，它们采用 DX8 API 架构并由硬件生产商 NVIDIA 公司生产，如此得名也许主要就是因为它们是用来对单个顶点或单个像素进行光照。我们所感知到的着色表面的质量很大程度上都要依赖于我们是否建立了顶点着色的模型（譬如 Gouraud 着色方法），然后再用 GPU 从顶点间插值；抑或是否建立了一个需要对每个像素进行求值计算的模型（譬如 Phong 着色方法或者是凹凸映射）。（我们可以注意到，对于 Phong 着色方法而言，只要物体网格有足够高的多边形分辨率，顶点模型和像素模型在着色质量上几乎没有什么差别。）因此，在实时渲染中大部分得到提高的质量都来自于程序员对逐像素功能性的访问。需要注意的是，这并不意味着我们可以访问独立的像素功能性。这种功能可以对新的纹理映射的周边进行考察，并且对各设备进行寻址。从这一角度而言，这个成果可以看作是使用贴图进行渲染的一个新进展。这种策略在图形学和游戏中使用已经有很长时间了。

4.2.1　基本的逐像素着色

我们首先简要回顾一下 Phong 经验模型——现代渲染技术的奠基石。这是一个局部反射模型，因为这个模型只考虑被着色的物体和光源的交互。（局部反射模型不同于全局反射模型，后者模拟所有光的交互，包括物/物交互。虽然局部反射模型对于单个物体可以提供高质量的着色，但是它们不能用来计算重要的全局景象，对于全局景象最重要的是阴影。）

这个模型可以写成如下形式：

$$\text{反射光} = \text{环境光} + \text{漫反射光} + \text{镜面反射光}$$

或者

$$I = I_a + I_l(k_d(\mathbf{N}.\mathbf{L}) + k_s(\mathbf{R}.\mathbf{V})^n)$$

这里：

I_a 是表示环境光的任意常量

I_l 表示光源的强度

$\mathbf{N}$ 表示相关点的曲面法向

$\mathbf{L}$ 表示光照方向的向量或者从相关点到点光源的向量

R 表示关于 **L** 的镜面反射的方向

V 表示视见向量

k_d 表示“携带”物体颜色的漫反射系数

k_s 表示设定光源颜色的镜面反射系数

n 表示发光参数的幂

这里需要注意的是，除非只用物体和远处的所谓位置光之间的距离进行着色，否则我们不考虑由于距离导致的光衰减。而且，我们还能将 I_l 因子分解为反射系数。如果只考虑漫反射光照，同时将光源移到无限远处（这样光源转化为线源而不是点源），光照就退化为只含有曲面法线作为自变量的函数：

$$I_{\text{diffuse}} = f(\mathbf{N})$$

这是游戏中光照贴图的一个基本方程。它说明了我们可以在只有视点变化的情况下对游戏中的静止物体预计算它的光照贴图。如果光源和静止物体之间没有相对移动，只要用这个方程预计算着色就可以了。在某些情况下，预计算光照贴图可以实时更新，比如说把爆炸的效果“画到”邻近静止的物体上去。[WATT01] 完整地讨论了这样的具体实现。

Blinn [BLIN77] 将上述模型作了简化，它把 (**R**.**V**) 项替换为 (**N**.**H**) 项，这里 **H** 表示半程向量，由下式给出：

$$\mathbf{H} = \mathbf{L} + \mathbf{V}$$

和上面一样，如果我们把光源移到无限远，就得到下面的方程：

$$I_{\text{specular}} = f(\mathbf{N}, \mathbf{V})$$

这样，漫反射部分就是视图独立的，可以运用预计算，同时还可以把它缓存在光照贴图中。镜面反射部分则是曲面法线和视见向量的函数。对于动态物体和镜面光照，我们可以使用带点积纹理混合（4.2.4 节）的法向图或是前滤波的环境贴图（4.5.1 节）来计算：

$$I = f(\mathbf{N}, \mathbf{V})$$

在这些方程的实现中所蕴涵的就是所谓的自阴影估计：

$$I_{\text{diffuse}} = (k_d \max\_of(0, \mathbf{N}.\mathbf{L}))$$

$$I_{\text{specular}} = (k_s \max\_of(0, \mathbf{N}.\mathbf{H})^n)$$

这里如果 (**L**.**N**) 和 (**N**.**H**) 为负就归零。

对凹凸贴图进行着色时需要用到两条法线——未受扰动曲面的每个像素的法线 **N** 和受扰动的法线 **N′**，这两个自阴影估计这样计算：

$$I_{\text{diffuse}} = I_l(k_d{}^* \text{s}^* \max\_of(0, \mathbf{N}'.\mathbf{L}))$$

$$I_{\text{specular}} = I_l(k_s{}^* \text{s}^* \max\_of(0, \mathbf{N}'.\mathbf{H})^n)$$

$$\text{这里 s} = \begin{cases} 1, & \mathbf{L}.\mathbf{N} > 0 \\ 0, & \mathbf{L}.\mathbf{N} \leqslant 0 \end{cases}$$

否则就可能发生这样的情况——就算受扰动的像素面是曲面的一部分，也可以看到本来看不到的那个光源。Kilgard [KILG99] 就指出，用一个阶跃函数来表示自阴影项会产生间歇形式的、短暂的、虚假的人造痕迹，并且这个项转化成一个斜坡函数，如下：

$$\begin{cases} 1 & (\mathbf{L}.\mathbf{N}) > c \\ \dfrac{1}{c}(\mathbf{L}.\mathbf{N}) & 0 < (\mathbf{L}.\mathbf{N}) \leqslant c \\ 0 & (\mathbf{L}.\mathbf{N}) \leqslant 0 \end{cases}$$

这里 c 的推荐值是 0.125。

把基本的 Phong 着色模型简化为一个只含有 **N** 和 **V** 的方程后，我们就有多种不同的方法在硬件上运用纹理映射来实现它。最直接的方法就是用一个法向图来表示 **N**，我们将在 4.2.4 节中描述。

4.2.2　着色和坐标空间

实时渲染中一个重要的论题是坐标空间，光照计算就是在坐标空间中进行的。这在理论上并不重要——我们只要简单地把所有影响光照方程的东西都定义在同一个坐标空间中就可以了——实际上，往往重要的是效率。

对于逐像素光照我们经常（比如在 OpenGL 中）在视线空间中进行计算，在这种情况下，物体的法线必须要转换到视线空间中。要转换法线，我们得使用转换几何结构的逆矩阵的转置。在 OpenGL 中，这是用模型视图矩阵的逆转置矩阵来完成的。

$$\mathbf{N}_{eye} = (\mathbf{M}^{-1})^{T}\mathbf{N}_{obj}$$

或者我们也可以选择在对象空间中计算。使用哪个空间并没有什么影响——计算结果是一样的。

在逐像素着色的情况中，我们对每个像素使用着色方程。在许多应用中，我们拥有或是通过继承从贴图中得到每个像素的法线。实际上这些法线都处在一个邻近于该像素的坐标系统中，或者更准确地说，邻近于投影到该像素的物体上的一个点的坐标系统。如果可以在这个系统中计算光照，就会方便一点。我们可以在物体的顶点处转换视线和光线向量，然后通过插值来得到每个像素的值（如图 4-1）。这样，像素的法线就从纹理贴图中提取了出来。

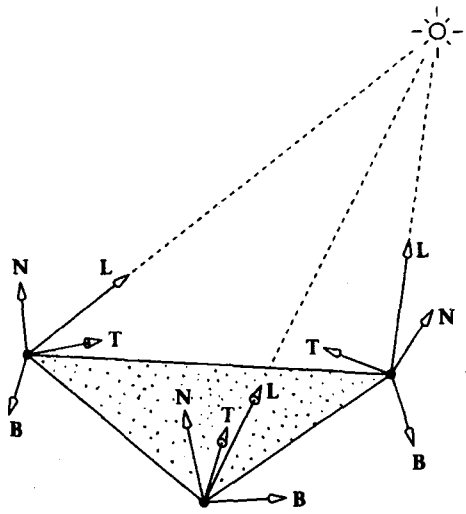


图 4-1　切线空间由顶点法线 **N** 和包含在切平面中的两个向量 **B** 和 **T** 构成。这对每个顶点和每个像素都是局部的

这样，我们需要计算一下在每个顶点的正交基 **TBN**，这里

N 是顶点的法线

T 是切向量

$\mathbf{B} = \mathbf{T} \times \mathbf{N}$ 是双法线

这样运用矩阵

$$\begin{bmatrix} \mathbf{T}_x & \mathbf{T}_y & \mathbf{T}_z \\ \mathbf{B}_x & \mathbf{B}_y & \mathbf{B}_z \\ \mathbf{N}_x & \mathbf{N}_y & \mathbf{N}_z \end{bmatrix}$$

光线向量就转换到了这个空间，也就是我们所知道的曲面局部空间[⊖]。

⊖ 这个空间也被称为切线空间或是纹理切线空间。

对于一个多边形物体，给定经过一个顶点的法线 **N**，它的切线处在垂直于这个向量的平面内。要选择物体曲面切线的方向，我们可以运用纹理贴图的参数，然后选择那个总是指向纹理贴图的坐标轴的方向（如图 4-2）。这可以计算如下：

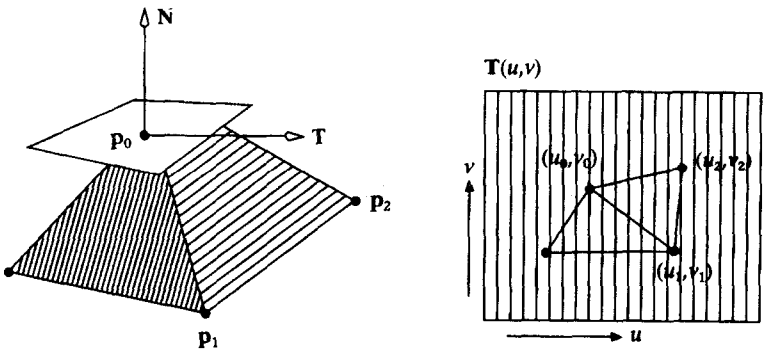


图 4-2 求某个顶点处的切向量

对每个顶点而言，对于每一个使用这个顶点的三角形：

$$\mathbf{v}_0 = \mathbf{p}_1 - \mathbf{p}_0$$

$$\mathbf{v}_1 = \mathbf{p}_2 - \mathbf{p}_0$$

$$\Delta u_0 = u_1 - u_0$$

$$\Delta u_1 = u_2 - u_0$$

$$\mathbf{T} = \Delta u_0 \mathbf{v}_1 - \Delta u_1 \mathbf{v}_0$$

这里，(u_0, v_0)、(u_1, v_1) 和 (u_2, v_2) 是与 $\mathbf{p}_0$、$\mathbf{p}_1$、$\mathbf{p}_2$ 相关的纹理坐标。对向量 **T** 取平均就得到了一个处于切平面中的向量。

如果物体有一般的参数化表示 $O(u, v)$，我们就有一个用向量定义的切平面的概念：

$$\frac{\partial O}{\partial u} 和 \frac{\partial O}{\partial u}$$

它的法线为：

$$\frac{\partial O}{\partial u} \times \frac{\partial O}{\partial u}$$

4.2.3 25 年来主流的插值着色方法和颜色贴图

自从 20 世纪 70 年代发明插值着色方法以来，纹理贴图一直都被用来添加细节信息及“视觉相关点”到物体表面。这种方法包含在许多图形处理软件当中，同时也在游戏产业中得到了很大程度的拓展（光照贴图）。的确，如果没有纹理映射硬件，那么游戏产业很可能就发展不成现在这样轻松的文化产业。

当然，“纹理贴图”有些用词不当，比较准确的应该是“颜色贴图”。颜色/纹理贴图的普遍存在，以及它们在光/物体的交互过程中总是被作为分开的实体对待，这两个因素很可能就导致了真实纹理所具有的明显的物理性质，也就是依赖于观察的角度和照射的角度。你无法用一张砖墙的照片来模拟砖块的纹理。（对阴影也是一样。当作为全局照明交互的结果

时，在插值着色器中，它们从光/物体的交互中独立出来。）事实上，现在有一个公用的图像数据库（www.cs.columbia.edu/CAVE/curet），它由 BTF（双向纹理函数），即作为观察和光线照射方向的函数的纹理贴图组成。数据库中的这些图像都是通过对真实表面的物理测量生成的，这个数据库包括了 14000 幅图像。使用 BTF，传统的映射可以用来准确地模拟出表示为观察方向和光照方向的函数的纹理，这样它就可以实现自阴影。这里的问题就是要扩充这个数据库中的数据。Dana et al.［DANA99］对每个纹理使用 205 幅贴图，并通过在这个数据库中的贴图间进行插值来为所需的视角和光线照射角计算 BTF。很明显，这会在实时应用中产生纹理管理的问题。

光照、贴图和游戏中的技术——一些历史

3D 游戏中一个授权使用的软件技术就是光照贴图。光照贴图把场景中从所有光源发出并经所有（静止的）物体表面反射回来的光缓存起来。当观察者绕着一个场景移动时，从光照贴图中我们就可以得到视图独立物体的着色。光照贴图的两个很大的优点是它使得光照能够预计算出来，而且对于视图独立的光照——漫反射——光照贴图可能会很大。也就是说，光照贴图对低频的反射光作了缓存，而且是视图独立的。

在支持多纹理的硬件和提供额外灵活性的纹理混合模式的帮助下，这种通用的方法现在已经有了进一步的发展。举例来说，Quake3 的开发者们所开拓的一项技术拓展了光照贴图，使它能够包含高频细节。这项技术就是我们在上一节中所讨论的 BTP 技术的一个变形——它只用了 4 幅贴图/曲面而不是 205 幅。对于凹凸映射这是一个比较粗糙但高效的近似。它对一系列的高频光照贴图做计算，这些光照贴图把从物体表面反射回来的光缓存成一些不同的观察方向。理想情况下，我们希望能对从所有方向反射回来的光都作缓存，但这会产生一些限制。不失一般性，比如取 4 个观察方向，每个观察向量都成 45°仰角。现在，随着观察者相对物体表面移动，物体表面细节的三维特性必须表现得很明显。这种效果通过计算当前观察方向和每个 45°角预计算所得的观察向量的点积，然后把计算的结果数值作为这 4 幅贴图的混合因子来实现。于是在多纹理管道中，随着照相机的移动，我们所需要做的全部的动态计算就是观察方向/凹凸贴图角度的点积，然后用这些结果去混合凹凸贴图。

当前传统的贴图方法归纳如下：

光照现象	贴图名称	视图依赖/视图独立（已调整的/已缓存的参数）	预计算的/更新的
物体颜色	纹理	视图独立 k_d	预建模
物体纹理	凹凸	视图依赖（N）	预建模
物体漫反射	光照	视图独立（L.N）	预计算
静止物体阴影	光照	视图独立	预计算
雾	雾	视图依赖	预计算
移动的光照	光照	视图独立	通过限制光的影响更新光照贴图
物体镜面反射	镜面贴图，光泽表面贴图	视图依赖	预计算
物体/环境镜面反射	环境	视图依赖	预计算

一个一般的使用贴图混合形式的光照方程也许可以写成如下形式：

C = 雾*（纹理贴图*（光照贴图） + 镜面贴图 + 环境贴图）

这里为了简化，我们把混合限制为加运算和乘运算。

在使用前一次凹凸映射逼近的情况下，光照贴图项可以扩展为：

((((光照贴图*凹凸 1）*凹凸 2）*凹凸 3）*凹凸 4）

这里，光照贴图/纹理贴图的运算符号为*，这一运算依照缓存在光照贴图中对光照的计算结果使纹理贴图变暗或是变亮。参考一下 Phong 局部反射模型我们应该很容易就看到这个效果。这个模型把漫反射部分和镜面反射部分加在一起。漫反射的部分是：

$$k_d(\mathbf{L}.\mathbf{N})$$

此处 k_d 即漫反射系数，等于纹理贴图，同时（L.N）等于缓存在光照贴图中的反射强度。镜面反射部分则全部用使用反射视图向量做索引的镜面贴图替代。

在上述的方程中，我们有一个用括号表示的层次关系。很明显雾应该被提到所有之前计算的结果之外，因为它会对整个图像产生影响。其他的变量就不是那么明显了，也许我们还需要实验来确定。比如，把光照镜面贴图同光照和纹理贴图的乘积相加：

（纹理贴图*光照贴图） + 镜面贴图

可能会导致镜面反射的细节在阴影中的区域显得过于明亮（表现为发射性的）。在这样的情形下，相比而言使用

（纹理贴图 + 镜面贴图）*光照贴图

会更好。这个问题的出现源于我们可能使用光照贴图来缓存直接反射和阴影。通常 Phong 方法在整个方程解出来以后才把阴影的区域加到图像上去。

4.2.4 标量表示

3D 模型的标量表示就是把几何变化编码为 2D 图。这种形式的表示中最常见的例子就是凹凸贴图，在这里，一个低分辨率的网格被一个包含高分辨率细节信息的贴图所增强。我们下面要讲的几种方法，在表示形式和低分辨率的网格的来源方面都不尽相同。

对多边形网格的标量表示起源于人们认识到三角形网格中存在着大量的冗余。首先，这里存在着网格的连接信息，这些信息必须存放在物体的数据结构中。第二，用顶点的坐标三元组来表示几何位置是冗余的。传统的几何压缩方法（参见［DEER95］）运用诸如近似的数值和对顶点的偏移来计算顶点位置。另外，使用三角带来确定相邻三角形之间的连接冗余也是一个常见的方法。有了向量表示的方案，原来的网格就被低分辨率或是基本网格结合一个低振幅标量置换式贴图所替代。这种方法的潜在优点在压缩可能性中得到了突出的反映。

微分几何学告诉我们，应该可以用一个单独的向量值来表示一个几何位置。这个向量以位移的形式沿着物体表面的法线从切平面延伸到物体表面。乍一看这好像自相矛盾。当然，我们有一个置换式贴图来描述穿过那个表面的几何变量，但仍然需要对位移的表面进行描述。这个明显的矛盾可以用这样的工具解决，用通常的术语来讲，这一工具意味着我们对物体表面和置换式贴图保存一个低分辨率的表示信息，它包括一些高分辨率细节。

凹凸贴图

凹凸贴图是 2D 模式，由程序生成或绘出，它使用高频细节对物体的表面进行调整。本节将对凹凸贴图作深入的阐述。首先是一些背景资料。

凹凸贴图的关键思想是，物体表面上小范围几何曲面的变化可以用一个含两个变量的标量函数——高度场来编码。如果想把一个基于顶点的表示方法具体化到同样好的细节层次，每个顶点都需要三个标量来表示。然而，这意味着我们需要物体表面的一个参数化法，以使我们能够从某个 3D 对象空间映射到 2D 参数化空间。

考虑第一个置换映射。在一个地形模型中，参数化法可能是一个规则的栅格，同时物体——那个地形——只是一个高度场。一般我们通过置换物体表面一点的方法对物体进行高频调制。在地形的例子中，物体是一个平面，高度场和物体之间的映射是一对一的映射。如果那个物体是一个球体——比如一个星球——那么我们仍然能把地形编码为高度映射 $H(u, v)$。球体表面上的一个点 (x, y, z) 可以运用如下的相对纬度/经度映射到 (u, v) 空间：

$$v = \text{纬度}/\pi = \arccos(-z)/\pi$$

$$u = \text{经度}/2\pi = \begin{cases} \arccos\left(\dfrac{x}{\sin(\text{纬度})}\right)/2\pi & y \geqslant 0 \\ 2\pi - \arccos\left(\dfrac{x}{\sin(\text{纬度})}\right)/2\pi & y < 0 \end{cases}$$

这样，对于球体表面的点 **P**，我们沿着物体表面的法线把它移动到 **P′**：

$$\mathbf{P}' = \mathbf{P} + H(u, v)\mathbf{N}_p$$

置换映射除了在计算代价方面和传统的、依据所要求的细节层次对几何体进行建模的方法类似以外，它还能解决我们遇到的所有问题。使用标准的渲染器，置换必须在管道的开始处应用到物体上。

凹凸贴图带来的巨大突破是它具有置换映射的效果，但是又可以以同样的方式应用于作为颜色映射的渲染的同一阶段。着色方程使物体表面看上去好像被一个标量高度场 $B(u, v)$ 置换过了一样，凹凸贴图就是这样旋转和某个像素相关的表面法线的。

凹凸贴图算法中的变化与扰动编码到凹凸贴图中的方式有关，而这个方式依赖于坐标空间的选择。经典的方法［BLIN78］使用一个偏移向量对法向做扰动。

$$\mathbf{N}' = \mathbf{N} + \mathbf{D}$$

此处 **D** 是位于物体外表面的切平面中的向量（见图 4-3a）。

$$\mathbf{D} = \frac{\dfrac{\partial B}{\partial u}\left(\mathbf{N} \times \dfrac{\partial \mathbf{O}}{\partial \mathbf{v}}\right) - \dfrac{\partial B}{\partial v}\left(\mathbf{N} \times \dfrac{\partial \mathbf{O}}{\partial \mathbf{u}}\right)}{|\mathbf{N}|}$$

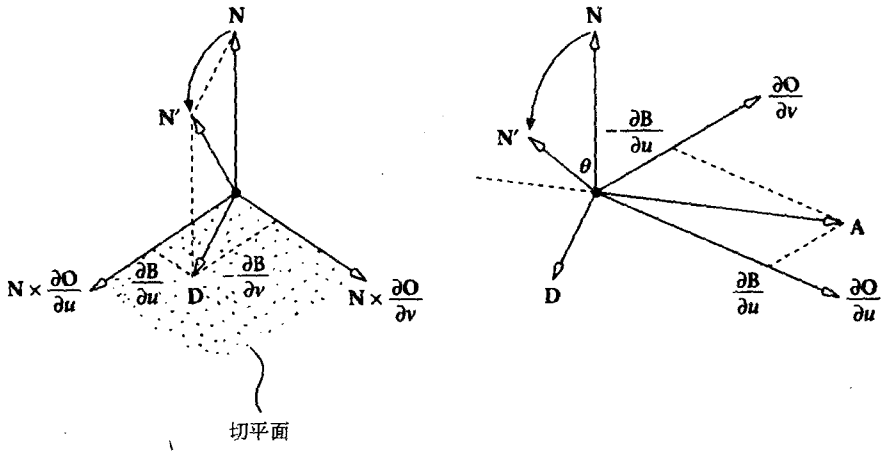
那个凹凸贴图或者是 $B(u, v)$，或者是我们预先计算并缓存起来的 $\mathbf{D}(u, v)$。如果预先计算 **D**，那么那个图就和物体的几何结构联系在一起了。另一方面，$B(u, v)$ 可以被几个物体的表面所共享，但是需要进行以上的计算。

法向贴图

另外一个方法是把扰动作为向量旋转存储起来（见图 4-3b）：

$$\frac{\mathbf{N}'}{|\mathbf{N}'|} = \mathbf{N}\mathbf{R}(u, v)$$

这里 **R** 是一个 3×3 的旋转矩阵 $\begin{bmatrix} r_{00} & r_{01} & r_{02} \\ r_{10} & r_{11} & r_{12} \\ r_{20} & r_{21} & r_{22} \end{bmatrix}$。



a) D 是切平面内的偏移向量　　b) N 绕 A 轴旋转 θ 角度

图 4-3 凹凸贴图可以用偏移向量或者符号来表示

在这个方法中 N 绕着位于切平面中的轴 A 旋转。由于 N 和 N′位于同一个平面中，旋转的轴可以如下给出：

$$\mathbf{N}\times\mathbf{N}'=\mathbf{N}\times\mathbf{D}=|\mathbf{N}|\left(\frac{\partial \mathrm{B}}{\partial v}\frac{\partial \mathrm{O}}{\partial u}-\frac{\partial \mathrm{B}}{\partial u}\frac{\partial \mathrm{O}}{\partial v}\right)=|\mathbf{N}|\mathbf{A}$$

这里，A 就是旋转的轴。

可以看到：

$$\tan\theta=\frac{|\mathbf{N}|}{|\mathbf{D}|}=\frac{|\mathbf{N}|}{|\mathbf{A}|}$$

这种方法的明显缺陷——每个（u，v）需要 9 个标量——可以通过正切或是物体表面的局部空间进行改进。法向贴图已经位于这个空间中了，于是我们可以对表示光照方向的向量进行变换。

在这个空间中 N 变成 [0，0，1]，这样我们就得到：

$$\mathbf{N}'=[\mathrm{r}_{02}\quad \mathrm{r}_{12}\quad \mathrm{r}_{22}]$$

一个很明显的问题是，法向贴图和凹凸贴图之间有什么区别呢？回答是毫无差别，这两种图都是将曲面法向的变化编码为一幅纹理贴图。凹凸贴图经常服务于这样的应用，所要求的纹理由画家使用一个绘图程序包建立——我们并不经常精确地模拟物体表面几何结构中的变化，但是要模拟材质的外观。法向贴图则是保存精确的几何变化信息，这样我们可以使用法向贴图来实现从网格几何结构对物体表面信息的退耦。我们可以用一个低解析度的网格来对复杂的物体进行渲染，并且使用纹理贴图的操作来进行细节上的渲染，如图 4-4 所示。图中，以同一幅图来看，表面保持着“相同”的分辨率，但是多边形的分辨率降低了。

使用法向贴图来进行渲染包含着这样的思想，认为物体的外表面：

- 作为低分辨率或可变分辨率多边形网格的位置信息。
- 作为高分辨率法向贴图的表面细节信息。

法向贴图在实际应用中的问题是它们的生成，这也是人们对它们不如对随处可见的凹凸贴图那样熟悉的原因。凹凸贴图可以交互式生成或者程序化生成，而且只需模拟物体外表面

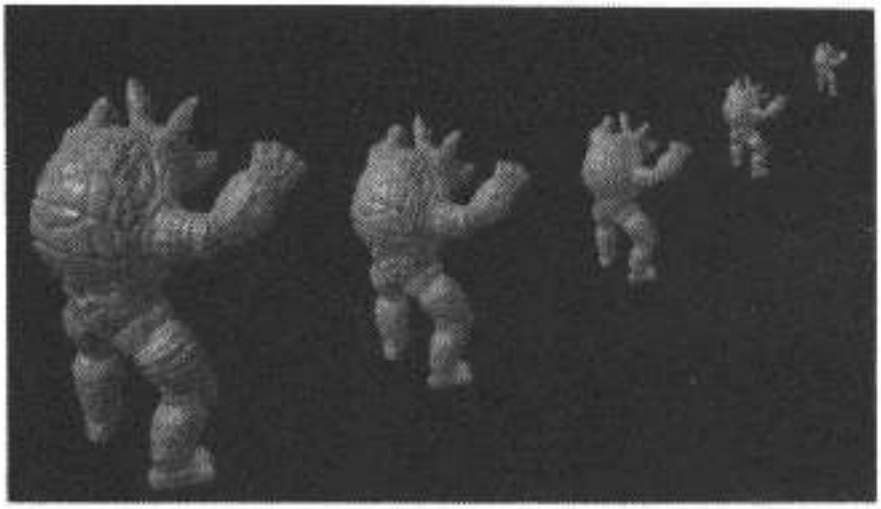


图 4-4　一幅以多种细节层次对模型进行渲染的图。细节层次分别包含有 7809，3905，1951，975 和 488 个三角形（由近至远）

的置换，而不是精确地模拟出它的全部细节。要为一个多边形网格物体生成一幅法向贴图，需要以下一些组成信息：

1）一个高分辨率的物体模型。

2）一个简化版的 1）以及一个可以用来使法向贴图能在渲染中用作传统纹理的参数。

3）一个以某种方法调用 1）和 2）的几何结构比较，以导出穿过大型多面体表面的法线的方法。在下一节，6.4 节及 6.5.3 节将给出解决这个问题的方法。

这类方法中一个特殊、直接的方法在［WATT01］中有详细的讨论。（本书中高分辨率的光照贴图被用作详细信息，但是原理是一样的。它用贝济埃二次曲面作为基本的几何体来离线计算光照贴图，然后在渲染的过程中根据希望的 LOD 来重采样二次曲面的网格几何。不管选择哪个 LOD，我们都使用原来的高解析度曲面的光照贴图来对图像进行纹理处理。

贴图和轮廓边处理

前面小节中讨论的方法——根据物体外表面把纹理映射到一个低分辨率的基本网格——忽视了轮廓边的处理。平均来说，一个轮廓边中有$\sqrt{n}$条边（这里 n 是物体表面的数目）。此处低分辨率的几何体就可见了。轮廓边的视觉质量很重要，因为它给出物体形状的视觉感受。同样，粗糙的轮廓边也抵消了使用高质量着色模型带来的效果。这些因素催生了 Sander et al.［SAND00］，称之为轮廓裁剪。这个方法包括把轮廓边从一个高分辨率的模型中解压缩出来，把它渲染进模板或者是 α 缓冲区，以创建裁剪低分辨率渲染的掩模。他们的工作指出了这种方法的两个明显的要求。

首先，因为经过渲染的粗糙网格的图像将用原来网格的轮廓边来裁剪，所以粗糙网格 M^1 必须包括原来的网格 M^n。Sander 等人提出了一个叫做渐进外壳（progressive hull）的结构，同时，就如它的名字所示的那样，这是渐进网格的修改版（参见第 6 章）。一个外部的渐进外壳是这样定义的：

$$\bar{M}^n \subseteq \ldots \subseteq \bar{M}^0$$

修改简化渐进网格的过程，使得所有在 $\bar{M}^{k+1}$ 中的几何结构都包含在 $\bar{M}^k$ 中。这就意味着物体的外表面必须保持不变或是往外移动[⊖]。

这种方法的第二个要求是实时地对 M^n 的轮廓边进行解压缩。这可以通过把边按序排列到一个层次结构中来完成。轮廓边定义为某一边，它的相邻面分别是前向和后向的。一棵树被组织成面的簇，它的每个节点都包含一个代表完全前向或是后向空间区域的面的簇。遍历这棵树就可以在运行时对外部的轮廓边进行解压缩。

置换细分表面

[LEE00] 统一了细分表面和置换式贴图。置换式贴图一直以来都被看作纹理贴图。但是由于要加入作为几何扰动的高频信息代价高昂，它们一直主要用于高端渲染系统中。它们的实时应用被用在动画工具中，比如生成水面，在这里由于 2D 参数化对图中的分析过程是直接的。

基本网格或是细分表面控制网格可以用依据 QEM（参见第 6 章）排序的传统边折叠变化而得。在简化过程中候选的边折叠进一步作如下限制。由于外表面被简化了，细分乘积的法线空间局部相似于对应原来网格的法线空间。这保证了使用沿域表面法向的标量置换，平滑的域表面——化简过程的最后结果——可以精确地表示出原来的网格。

置换细分表面中的基本表示是一个控制网络，它用循环细分方法[⊜]和一张置换式贴图来生成一个边界面，这里置换式贴图沿着边界面的法线扰动边界面上的点。最后的结果称为 DSS，它使用置换式贴图将点从平滑的域表面中移出去。

一个明显的好处是细节信息用单个的标量或标量图而不是用向量图表示。这是一个优化的表示方法——外表是某个域表面上的高度/标量场（就像地形可以表示成一个 2D 规则排列上的高度场一样）。这对于要求高度压缩的应用工具来说是十分理想的。置换式贴图可以事先用 mip-mapping 过滤，这和用法向贴图这样一个间接的方法来过滤形成鲜明对比。而且，在控制网格和置换式贴图之间有一个“简单”的参数化。如果这种方法用于逐像素着色，我们需要计算置换面的法线。虽然这可以从细分表面很快地计算出来，它还是需要邻近的信息。如果我们事先计算法向贴图，那么就失去了那个标量和过滤的优势，同时这种方法就退化为对于非常高分辨率表面的法向贴图的生成方法。在这个层面上，这种方法对于生成法向贴图的“渲染”方法是否还有优势还不太清楚。

预处理阶段如下：

1）使用边折叠算法的一个变形从原来高分辨率网格提取出控制网格。

2）生成一个有限表面。这将是原来表面的一个平滑版本。

3）沿表面法线方向通过对有限表面采样和光线投射从这些位于原来表面的采样点生成置换式贴图。

⊖ 可以用一个类似的过程来定义一个内部的渐进外壳：

$$\bar{M}^0 \subseteq \ldots \subseteq \bar{M}^n$$

这两者从两边一起规定了原始网格的边界（回忆一下化简包迹线方法 [COHE98]）。

⊜ 相对于插值方法，循环细分是对三角形网格的一个近似方法（Catmull-Clark 是对方形网格的一个近似方法）。在插值方法中，控制网络上的点同时也在有限表面上，但是有限表面的质量通常比近似方法要低。

我们可以直接获得所需的法线。我们有：

$$\mathbf{s} = \mathbf{p} - D\mathbf{N}$$

这里：

s 是被置换表面上的点

p 是有限表面上的对应点

N 是过 **p** 点的法线

D 是置换标量

N 从下式获得：

$$\mathbf{N} = \frac{\partial \mathbf{p}}{\partial u} \times \frac{\partial \mathbf{p}}{\partial v}$$

这里，分量利用细分表面上的第一个派生物模型很容易就可以计算出来。$\mathbf{N}_s$（即过细分表面上点 **s** 的法线）的计算由置换表面上切向量的叉乘给出：

$$\mathbf{N}_s = \frac{\partial \mathbf{s}}{\partial u} \times \frac{\partial \mathbf{s}}{\partial v}$$

$$\frac{\partial \mathbf{s}}{\partial u} = \frac{\partial \mathbf{p}}{\partial u} + \frac{\partial D}{\partial u}\mathbf{N} + D\frac{\partial \mathbf{N}}{\partial u}$$

对其他的切线有相同的表示形式。在凹凸贴图中，对于小的置换，可以忽略掉最后一项（它含有二阶导数）。图 4.5 给出了这种方案的一个概观，这种方法和 6.5.3 节中讲述的贴图方案有一些相同之处。



图 4.5　置换细分表面（DSS）：第一幅图是控制网格；第二幅——极表面——通过循环细分由第一幅图生成；第三幅——DSS——通过置换式贴图由第二幅图生成

4.3　因式分解法

我们现在来考察怎样利用纹理贴图硬件的处理能力来产生更详尽的功能。这通常采用因式分解模型，并把预计算出来的着色部分的值存储到二维的纹理贴图中来实现。随后进行的对纹理贴图中的硬件插值和没有重新标准化的向量插值（有时也称为快速 Phong 着色）是等价的。因此，这些方法需要依据这样的限制条件，即从每个像素获取的值并不完全等于在每个像素应用该模型而获得的值。这个方法可以同任何一种以一维或二维查询表来实现的着色模型一起使用，包括基本 Phong 模型。在这种技术中使用的模型有 Cook-Terrance 模型和

Banks 各向异性模型。

4.3.1 使用因式分解着色模型的逐像素着色——各向同性模型

在 1975 年 Phong 的研究成果发表两年后，J.Blinn［BLIN77］发表了一篇描述如何在计算机图形学中运用基于物理的镜面反射部分的论文。1982 年 Cook 和 Terrance［COOK82］扩展了这一模型，以解释高亮部分的光谱组成——它们对材质和光线的入射角的依赖。相比由 Phong 模型获得的进展，这些进展在高亮部分的尺寸大小和颜色上有一点细微的影响。这个模型仍然将入射光分解为漫反射和镜面反射部分，而且这项新工作主要集中在镜面反射部分，漫反射部分则像前面一样计算。这个模型在渲染光滑发亮的类金属表面时表现得最为成功。通过镜面反射的高亮部分中颜色的变化，可以将具有相似颜色的金属渲染成不同颜色。

高亮部分形状的确定问题是十分微妙的。高亮部分只是光源物体上的反射源的一个图像。除非物体的表面是平的，否则这个影像会被物体扭曲。而且随着入射光线角度的变化，它会落到物体的不同部分，同时它的形状也将改变。因此我们就有了一幅高亮度的图像，它的总体形状依赖于入射光线照射的物体表面区域和视线观察方向，后者决定了从视线观察方向可以看到多少高亮部分。这些是决定我们看到的物体表面亮光光斑形状的几何因素，使用 Phong 模型很容易就可以计算出来。

决定高亮图像的物理因素是对光照强度的依赖以及相对于所考察过的该表面上那一点的切平面的入射光角度的颜色。这向我们揭示了材质的特质，同时也使我们，比如说，可以将金属和非金属的物体区分开来。

和光反射相关的物理模拟指的是我们试图模拟造成光反射的表面的微几何结构，而不是像我们在 Phong 模型中做的那样只是用一个经验项简单地模仿光的行为。

这个对镜面反射光高亮部分的早期模拟有 4 个相互关联的部分，它是基于物理微面模型的，在这个模型中，围绕着一般表面有着对称的 V 形凹槽（图 4-6a）。我们现在依次简要地描述一下这些部分。（更全面的描述和渲染的例子可以在［WATT00］中找到。）

1）建立一个关于微面的方向的统计分布，它对于从一个特殊观察方向中照过来的光给出一个项 D。可以使用一个简单的高斯曲线：

$$D = k\ \exp[-(\delta/m)^2]$$

这里 δ 是微面相对于一般表面法线的倾斜角，即倾斜角在 **N** 和 **H** 之间，m 是该分布的标准差。计算这一朝向角度的分布就可以很容易得到这一朝向的微面数量，也就是能够反射从视见方向照射来光线的微面数量。图 4-6b 显示了 $m = 0.2$ 和 0.6 的两个反射凸角。

使用微面来模拟光反射对物体表面粗糙度的依赖引出了两个假设：

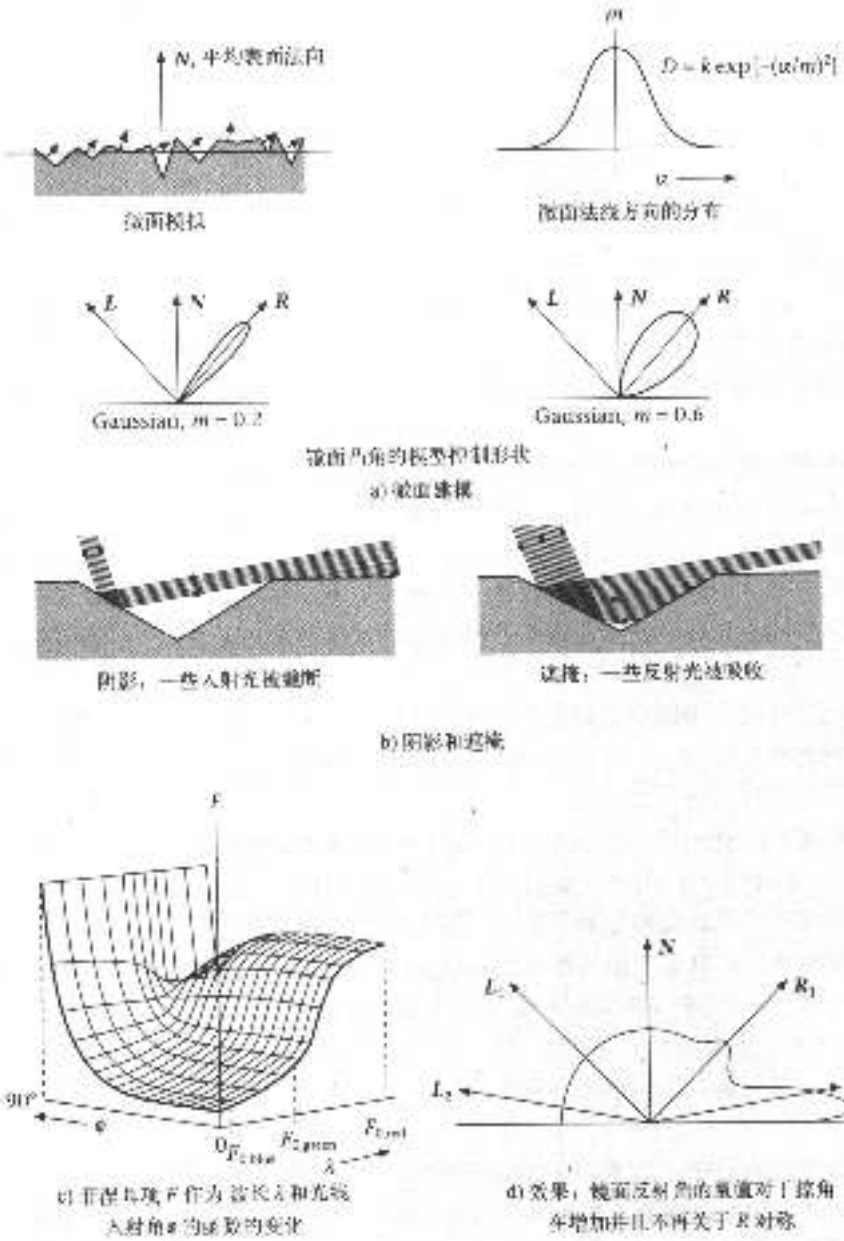
a）虽然微面在物理上来说很小，但是假设它大于光的波长。

b）入射光束的直径足够大，能够和足够产生典型反射光行为的微面相交。

这样这个因素控制了镜面反射凸角突出的程度。

2）视见向量或是光照方向向量开始接近一般表面，光的干涉效果就发生了。这些效果称为阴影和遮掩。当一些反射光被微面吸收时发生遮掩，当入射光被中途截断时就产生阴影，如图 4-6b 所示。

Blinn 在［BLIN77］中给出了这个因式对于 **L**、**V** 和 **H** 的一个详细的推导。对于遮掩：



微面凸角的模型控制形状

a) 微面建模

b) 阴影和遮掩

c) 菲涅耳项 F 作为波长 λ 和光线入射角 φ 的函数的变化

d) 效果：镜面反射角的重值对于掠角在增加并且不再关于 R 对称

图 4-6　Cook 和 Torrance 模型——图示式的概观

$$G_m = 2(\mathbf{N}.\mathbf{H})(\mathbf{N}.\mathbf{V})/\mathbf{V}.\mathbf{H}$$

对于阴影，只要互换向量 **L** 和 **V**，它与前者的几何结构就完全一样了。对于阴影我们有：

$$G_s = 2(\mathbf{N}.\mathbf{H})(\mathbf{N}.\mathbf{L})/\mathbf{V}.\mathbf{H}$$

G 的值必须取 G_m 和 G_s 的较小值，这样：

$$G = \min\_of\{1, G_s, G_m\}$$

3) 引入另外一个纯几何项以解决低入射角照射的问题。随着观察的视角和入射光的角

度逐渐接近，观察到的镜面反射大大增加了。随着视见向量和一般表面法线的夹角朝着90°增加，观察者看到越来越多的微面，这可以由这一项来解释：1/N.V。

也就是说，观察者看到的增加的微面区域反比于视见向量和一般表面法线的夹角。如果有一束入射光以小角度射入，那么相比观察者从一个靠近法线的入射角度观察将有更多的光反射向他。这个效果可以用阴影效果来解释，它在观察方向接近一般表面方向时也会发生。

4）接下来要考虑的是菲涅耳项 F。这项主要考虑被反射回去的光相对于被吸收的光的量——一个依赖于被看作完美镜面的材质的因素，我们的微面就是这样的。换句话说，前面对整个表面作为一种和理想镜面有着相同表现特性的微面集合建立了模型，我们现在考虑一个完全平滑的表面。这个因素决定了作为入射角和波长函数的反射凸角的强度（见图4-6c）。这里对于波长的依赖可以用镜面反射高亮部分中细微的颜色效果来解释。

$$F = \frac{1}{2}\left\{\frac{\sin^2(\phi-\theta)}{\sin^2(\phi+\theta)} + \frac{\tan^2(\phi-\theta)}{\tan^2(\phi+\theta)}\right\}$$

这里：ϕ 是光的入射角

θ 是光的折射角

$\sin\theta = \sin\phi/\mu$

其中 μ 是材质的折射率

这些角度在图4-7中画出来。F 是最小值，也就是说，大多数光在 $\phi = 0$ 或是垂直射入时被吸收了。$\phi = \pi/2$ 时，表面不吸收光，同时 F 等于全部入射光。F 依赖于波长，因为 μ 是波长的函数。

这一项的实际作用是解释作为入射角度函数的镜面反射高亮部分的颜色上的细微变化。对任何材质而言，当光线几乎平行于表面入射时，高亮部分的颜色接近于光源的颜色。从其他的角度而言，颜色依赖于光的入射角度和材质本身。

这一项的作用就是使得反射光的强度随着入射角度的增加而增加（就像前面1/N.V项那样）——较少的光被吸收，更多的被反射回去。（一个更细微的效果是随着光入射角的增加，镜面反射的凸角从理想镜面的方向移走了。）

这个模型的总体效果表现在图4-6d中，它表现了两个形状/大小明显不同的镜面反射凸角。在Phong模型中，镜面反射凸角总是具有相同的大小和形状。

这样把上面的四项放到一起，镜面反射项就成为：

镜面反射部分 = $DGF/$（N.V）

这里，D 是微几何结构项。

G 是阴影/遮掩项

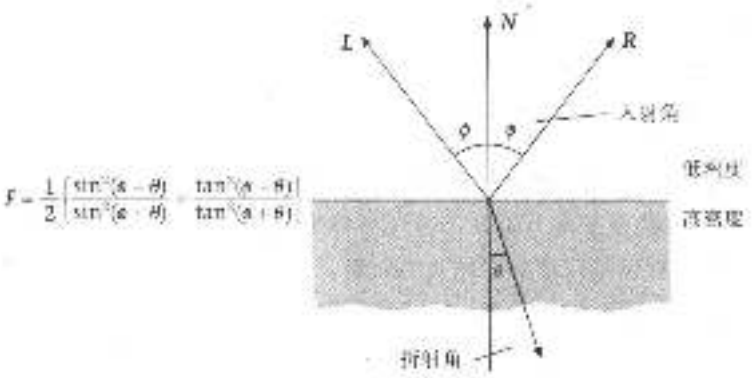
F 是菲涅耳项

（N.V）是光线照射项

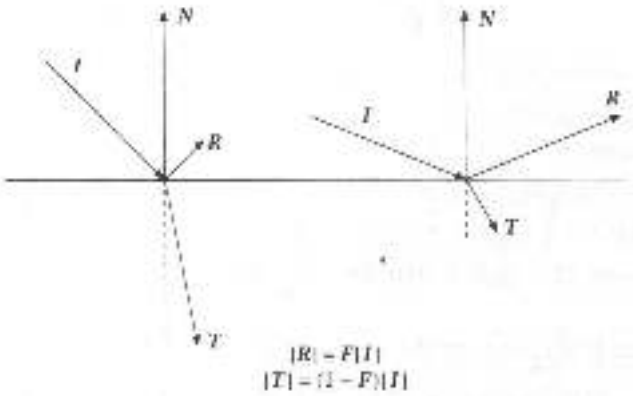
入射光的方向控制着细微的关于高亮部分形状和颜色的二级效果。当我们试图模拟出比如闪亮的塑料和金属的区别时，这种效果是很重要的。比如说，金在白光的照射下散发出黄橙橙的亮光，而当白光掠过它表面时那个亮光会变白色。

镜面反射项是单独计算的，然后和一致的漫反射部分组合起来以产生一个双边反射率的分布函数（或称为BRDF）：

$$\text{BRDF} = sR_s + dR_d \quad (\text{这里 } s + d = 1)$$



a) 菲涅耳方程中的角度



b) 两个例子，演示了该方程的使用状况

图 4-7　菲涅耳方程

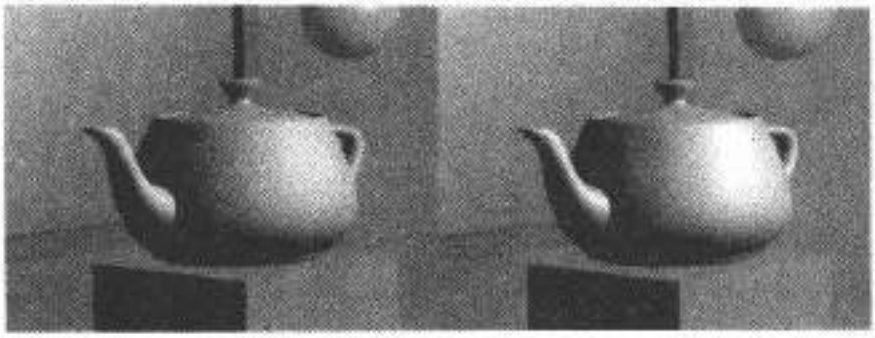
这个公式强调指出了朝任何特定方向反射回去的光（在计算机图形学中我们主要关注沿视线观察方向 **V** 反射回来的光）都是一个函数，不仅关于这个反射的方向，而且还关于入射光入射的方向。一个 BRDF 可以写做：

$$\text{BRDF} = f(\theta_{in}, \phi_{in}, \theta_{out}, \phi_{out}) = f(\mathbf{L}, \mathbf{V})$$

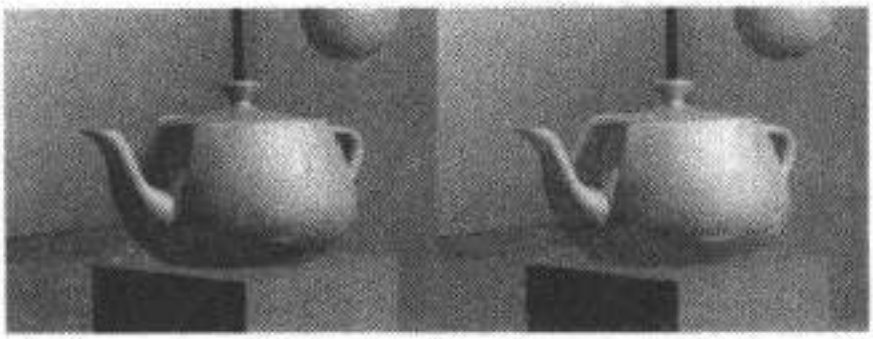
（参见图 4-11）而且在计算机图形学中使用的许多模型从模拟出这些依赖性的角度讲，它们自身之间都是不同的。4.3.2 节中给出了一个更广泛的处理方法。

通常，金属用 $d = 0$ 和 $s = 1$ 来模拟，闪亮的塑料则用 $d = 0.9$ 和 $s = 0.1$ 来模拟。请注意，如果对于金属 d 没为 0，那么镜面反射项将在物体整个表面的范围内对物体的颜色起控制作用。请把这一点和 Phong 反射模型中物体的颜色总是由漫反射的部分来控制做一比较。正因为如此，Phong 模型就无法产生出金属外观的表面，而且所有使用 Phong 模型渲染的表面物体都有明显的塑料外观。使用这个模型在同样的光照条件下选择不同的材质产生的效果如图4-8（也见彩页）所示。从这个示例我们可以看到镜面反射提光特性的广泛的差异变化。

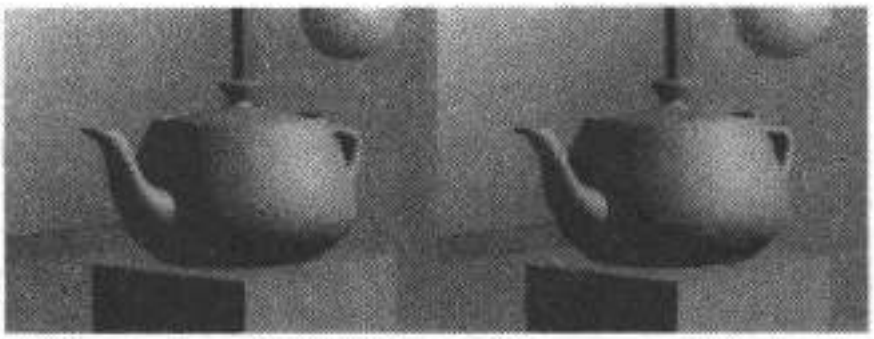
这样对给定的 **L** 和 **V**，反射光线由法线方向向量朝向 $\mathbf{H} = \mathbf{L} + \mathbf{V}$ 的微面的比例乘以光线入射角度的菲涅耳因子给出。如果往外射出的光线不被遮蔽，入射的光线没有被掩盖，那么



铸铝与磨光铝



磨光金与抛光金



黎铜与青铜

图4-8 在相同光照条件下渲染的不同材质。在抛光材质的情况中，该模型被用作光线追踪器的一个局部分量

这里的每一个微面都对反射光线起到了一定的作用。因此这个积的大小由阴影和遮掩项决定，而与 **N**.**V** 成反比。以微面为基础的模型中的阴影和遮掩项会带来许多问题，这是因为很多可能与 $D(\alpha)$ 一致的表面几何会引起不同的阴影和遮掩。

要运用纹理硬件来实现 Cook 和 Torrance 模型，我们首先考察一下需要用到的项之间的相关性。F（菲涅耳项）对于固定的 n，只依赖于光线入射角 **L**.**H** = **H**.**V**。D 依赖于角度 $\hat{\alpha}$，或者等价的 $\mathrm{Cos}\hat{\alpha}$。阴影和遮掩项是4个点积的函数，但是简化的函数可以把点积数减少为两个——**L**.**N** 和 **N**.**V** [HEID99]，给定一个点或是有方向的光源：

$$R_s = \frac{F(\cos\phi)D(\cos\delta)G(\cos\alpha,\cos\beta)}{\pi\cos\beta}$$

这里：

$$\cos\varphi = \mathbf{L}.\mathbf{H} = \mathbf{H}.\mathbf{V}$$
$$\cos\delta = \mathbf{N}.\mathbf{H}$$
$$\cos\alpha = \mathbf{L}.\mathbf{N}$$
$$\cos\beta = \mathbf{N}.\mathbf{V}$$

然后将它因式分解为两个表达式，分别存入一个二维纹理中：

$$F(\cos\phi)D(\cos\delta)\text{and}\frac{G(\cos\alpha,\cos\beta)}{\pi\cos\beta}$$

这里，对于常量 **L** 和 **V** 这步操作可以设定为多纹理的单遍渲染或是两遍渲染，就像我们将要在第 5 章中讨论的那样。纹理矩阵[⊖]设为：

$$\begin{bmatrix} 0 & 0 & 0 & \cos\theta \\ H_x & H_y & H_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} N_X \\ N_Y \\ N_Z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta \\ \cos\delta \\ 0 \\ 1 \end{bmatrix}$$

和

$$\begin{bmatrix} L_x & L_y & L_z & 0 \\ H_x & H_y & H_z & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} N_X \\ N_Y \\ N_Z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\alpha \\ \cos\beta \\ 0 \\ 1 \end{bmatrix} \tag{4-1}$$

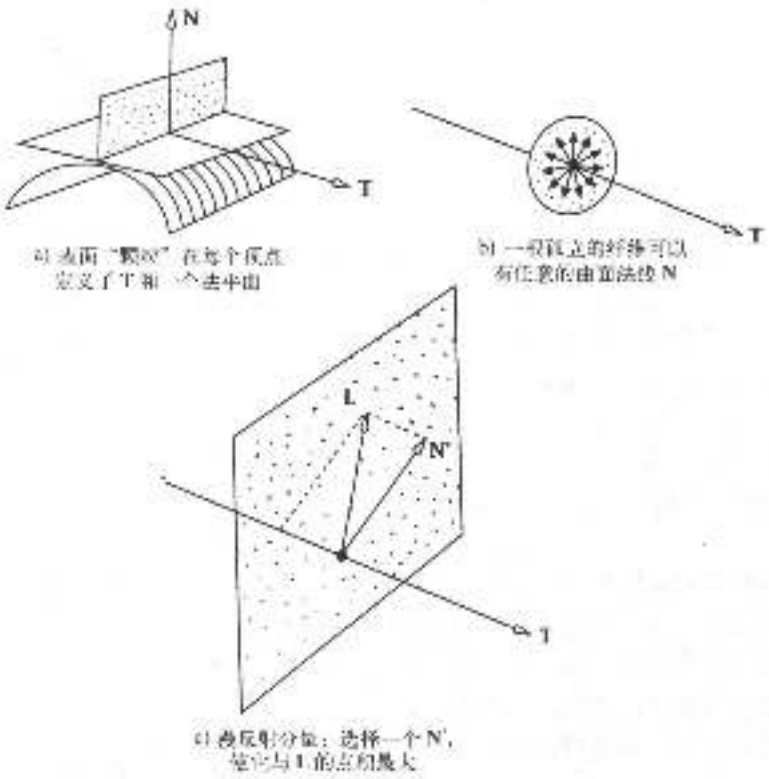
4.3.2　使用因式分解着色模型的逐像素着色——各向异性模型

前面的章节讲述了各向同性着色模型，在这个模型中我们考虑用单个角度来描述光线的入射角。这是入射光相对于表面法线做出的仰角。当反射回去的光依赖于这个角度和入射光的方位角时，各向异性的行为就发生了。各向异性 BRDF 的例子非常多。头发和抛光过的金属对于各向异性 BRDF 是不同的材质，这个例子在计算机图形学中一直都很受关注。这样来说各向异性 BRDF 就是当我们绕着法线旋转时不断变化的 BRDF。当 **L** 和 **V** 同材质的“颗粒”排列成一行时，反射达到最大，同时材质高亮部分的特征也被表现出来。

一个非常简单但是很有效的各向异性模型由 Banks 在 1994 年提出来［BANK94］。实质上它使用了 Phong 模型，但替换了各向异性几何模型中特有的向量。这个模型可以简单地想像成表面由纤维构成——就像盖在圣诞球上的缎子。这里我们认为纤维是很纤细、很长的圆柱体。这是一个定义在每个点上的模型（见图 4-9）在每个这样的点上都有一个和材质的颗粒对齐的切向量；整个模型用一个向量场来定义。这就定义了一个法线平面或者说是法线空间，使我们能够对法线 **N** 有选择的余地。我们假设最主要的光反射都来自于法线 **N′**，它和光线向量的点积最大。我们在 Phong 反射模型中使用这一法线就得到：

$$I = I_a + I_l(\mathbf{N}.\mathbf{L})(k_d(\mathbf{N}'.\mathbf{L} + k_s(\mathbf{N}'.\mathbf{H})^n)$$

⊖ OpenGL 中的纹理矩阵在应用之前可以用来操作纹理坐标（标量转换等）。



a) 表面"颗粒"在每个顶点定义了T和一个法平面

b) 一般孤立的纤维可以有任意的曲面法线N

c) 漫反射分量：选择一个N'，使它与L的点积最大

图 4-9 各向异性的反射

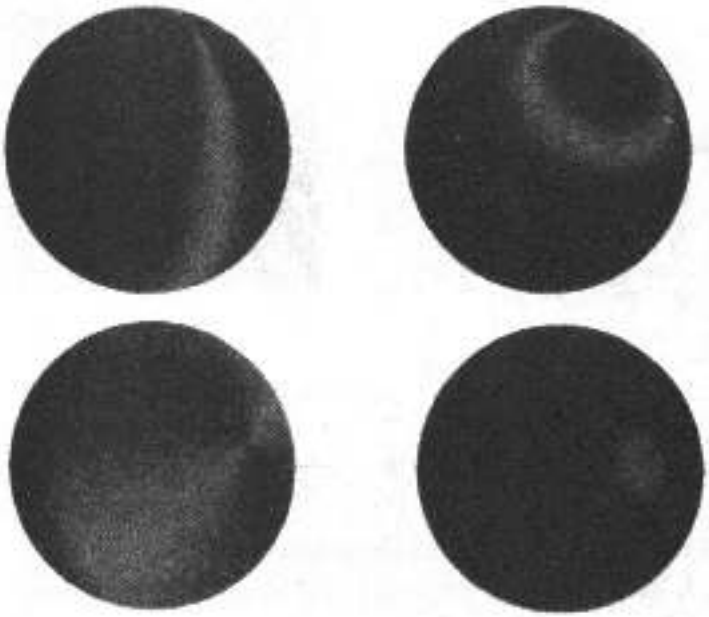


图 4-10 球面上的 Banks 各向异性着色模型。绿线表示光的方向。颗粒从沿经线排列到沿纬线排列水平地变化（观察点保持不变），观察点沿垂直方向变化

这样反射强度就成为了 L 和 V 分别与切平面形成的角度的一个函数。这意味着使用一个用切线 T 索引的预计算的二维纹理和在式（4-1）中的同一个纹理矩阵，我们就可以实现它了。

图 4-10（彩页中也有）显示了使用这个模型渲染的球面，材质的颗粒沿经线对齐或沿纬线对齐。

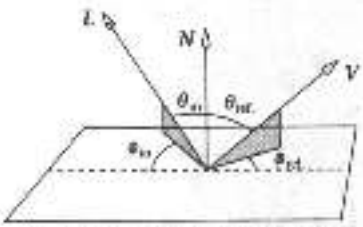
4.4　BRDF 和真实材质

本节实际上是前两节的一个概括，这两节中我们提出了处理光/物体相互作用的某些方面的特殊模型。对一个特定表面的光/物体的相互作用和材质完全由它的 BRDF 来刻画。我们首先从解释什么是 BRDF 开始，然后考察一下实时模拟它的方法。

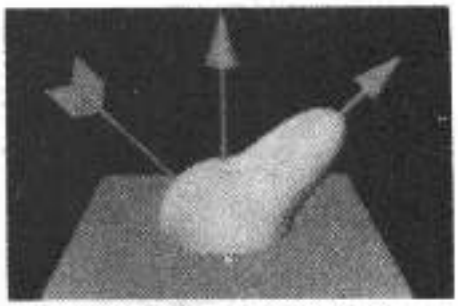
BRDF（双向反射率分布函数）对物体表面上某特定点 x 的光照的行为进行量化。这个公式强调指出了朝任何特定方向反射回去的光（在计算机图形学中我们主要关注沿视线观察方向 V 反射回来的光）都是一个函数。不仅关于这个反射的方向，而且还关于入射光入射的方向。BRDF 可以写做：

$$\mathrm{BRDF} = f(\theta_{in}, \phi_{in}, \theta_{ref}, \phi_{ref})$$

同时计算机图形学中的许多反射模型都根据模拟它们时的依赖条件来相互区分。图 4-11 显示了这些角度以及从一个特定角度集合计算出来的 BRDF。渲染过的 BRDF 显示了沿这一方向上入射的一根无限细光束的反射光（沿任何方向射出）的量值。在实际操作中光可能会从多个方向射到物体表面上的一个点，这样需要通过把每个入射光束各自的 BRDF 相加才能获得全部的反射光线。这一特殊模拟出来而不是测量出来的 BRDF 是使用一个标准的 Phong 模型计算出来的。



a) BRDF 把它向 L 上的入射光线同沿方向 V 的反射光线联系起来，把它们作为角度 θ_{in}、ϕ_{in}、θ_{ref}、ϕ_{ref} 的函数



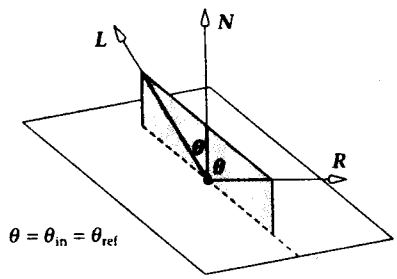
b) BRDF 的一个实例

图 4-11　双向反射性函数

在真实的表面上 BRDF 是六个变量的函数，另外两个变量是物体表面上我们的关注点 x 和光的波长。在计算机图形学中一般考虑均匀的材质——与点 x 无关，而且我们将忽略对波长的依赖（虽然图 4-12 很清楚地显示了对波长的依赖很重要）。

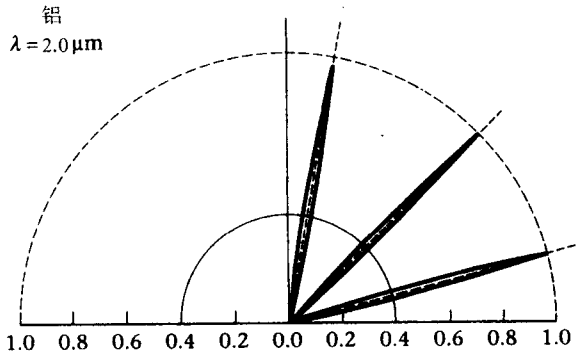
下面我们将给出 BRDF 的一个精确的定义。它是在 ω_{ref} 方向上反射光的微分与入射方向上光照的微分的比值：

$$f_r(\omega_{in} \to \omega_{ref}) = \frac{L_{ref}(\omega_{ref})}{L_{in}(\omega)\cos\theta_{in}\,\mathrm{d}\omega_{in}}$$

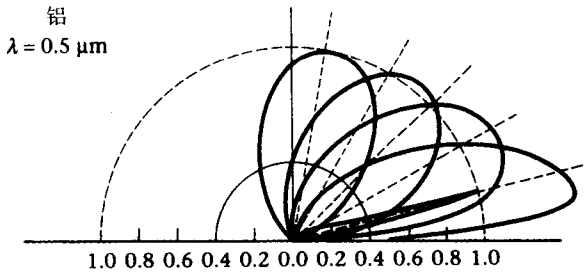


包含镜面方向 *L* 和 *R* 的平面

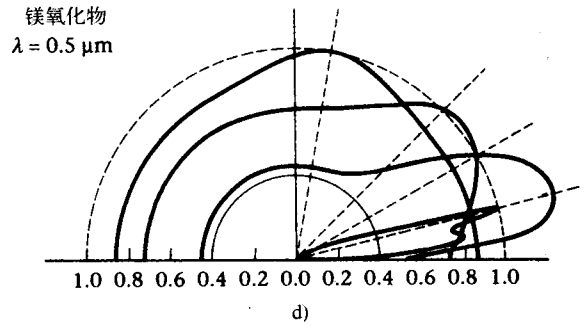
a)



b)



c)



d)

图 4-12　不同的材质和波长的光线的 BRDF 横截面

这里：

ω_{in}是绕着光线入射方向的微分角

ω_{ref}是绕着光线出射方向的微分角

θ_{in}是在光线入射方向和表面法线方向之间的夹角

这样，这个定义就可以用到下面的反射方程中：

$$L_{ref}(\omega_{ref}) = \int_{\Omega} f_r(\omega_{in} \rightarrow \omega_{ref}) L_{in}(\omega_{in}) \cos\theta_{in} d\omega_{in}$$

这里，$f_r(\omega_{in} \rightarrow \omega_{ref})$ 就是 BRDF。

它对 BRDF 和以指定点为球心的半球面上所有方向上的入射光线的乘积求积分，来给出该点的反射的发光度。

需要注意的是角度 ω_{in}和 ω_{out}都是相对于局部 TBN 框架（4.2.2 节）而言的，而且对于 BRDF 渲染我们需要为每个顶点都定义一个局部框架。这和使用简单的模型，比如使用向量 **L**、**V** 和 **N** 的 Phong 模型来渲染不同，因为在 BRDF 项中 Phong 模型只依赖于 ω 的 θ 部分。

许多年来计算机图形学一直都使用简单的、带有很多约束条件的 BRDF——就像图 4-11 所示的那样——Phong 模型。图 4-12 给出了这种模型和实际情况之间的差别。这个实例处在 BRDF 横截面中含有 L 和 R 的平面——不同光线入射（也是反射）θ 角的镜面反射方向。特别要注意的是反射凸角差距很大，因为它是入射光波长、入射角和材质的函数。在使用铝质材质时我们可以看到，它要么表现为一个镜面要么表现为依赖于入射光波长的有方向性的漫反射面。同时我们还要考虑到在具体实践中入射光从来都不是单色的（因此对每个所考虑的光的波长都需要一个单独的 BRDF），我们看到反射光的表现远比用三波长条件下的 Phong 模型要复杂。

我们需要重点区分的一个地方是区分各向同性表面和各向异性表面。各向同性表面显示的 BRDF 的形状只依赖于（θ_{in}，θ_{ref}，$\phi_{ref} - \phi_{in}$）。也就是说，如果那个面绕着法线旋转的话，BRDF 并不会改变。各向异性表面是，比如说，刷过的铝或是保留着铣床造成的结构的表面。在表面刷过的情况下，镜面反射凸角取决于入射光线和表面颗粒形成的夹角。这正好是我们在 4.3.2 节中用 Banks 各向异性模型模拟出来的行为。

现实中另外一个复杂的方面是大气的特性。大多数用在局部反射模型中的 BRDF 都被限制用在光在真空中从不透明材质上反射的情况。大多数情况下我们不考虑反射光在大气中的散射（同样也不考虑光在到达物体前在大气中的散射）。这样做的理由当然是简单化，并将光强度计算简化为对表面形状分类的向量和光照方向及观察方向的简单比较。

我们可以想像一下，如果对每种材质都有一个 BRDF，那么光-物体的相互作用问题就解决了。但是，虽然经过了 25 年的研究，直到今天还有很多问题悬而未决。比如：

- 如果我们想使用实时的 BRDF，那么从哪里得到那些数据呢？有些金属的数据可以从冶金学的文献中查到，但绝对是不够完整的。
- 我们以什么样的精度表示 BRDF 呢？应该考虑的区域中哪些是收到入射光照的呢？这块区域是否足够大，能够使统计模型在表面上一致呢？它是否要大到能包括如划痕那样的表面的不完全之处呢？这是一个还没有解决的问题。
- 在计算机图形学中我们怎么模拟 BRDF 呢？这最后一点是大多数模型之间存在差异的原因。特别地，我们经常要区分经验模型和物理模型。经验模型指的是模型光-物

体相互作用的模型。比如，在 Phong 模型中用一个数学方程来表示镜面反射的凸角。在 Cook 和 Torrance 模型中用一个统计分布模型来表示表面的几何结构，这就是所谓的物理模型。有意思的是，对于经验模型与物理模型相比较在视觉上的作用并没有普遍一致的看法。很多时候通过仔细调节经验模型的参数比使用物理模型能得到更好的效果。

4.5 使用 BRDF 进行逐像素着色

在下面章节中我们将讲述的方法对 BRDF 进行了分解，这样使得这个函数中的信息能用 2D 纹理贴图来表示，同时可以使用一个多遍/多纹理的算法来对表面进行渲染。

首先我们来看一个“类 Phong 模型的”BRDF 的特殊情形，并且看一下怎样才能使用传统的环境贴图来实现它们。然后更简略地看一下 BRDF 以及任意一个函数怎样才能用这样一种方法分解或压缩，使它能够映射到 2D 纹理中。

实现 BRDF：预滤波环境贴图

本节要考察的是使用方便的硬件支持的环境贴图方法使我们能够模拟材质，而不是使用理想镜面的环境贴图方法[⊖]。这也使我们，比如说可以通过把信息索引到一幅图中来表现漫反射和 Phong 着色。我们也曾提醒自己，同样作为一个经验模型，Phong 着色是无法实现的，因为它是一个局部模型，而且只反射光源的图像。通过计算一个镜面反射的环境贴图来去掉这个限制，这里的镜面反射包括了对其他物体和光源的反射。我们通过理想状态下的预滤波环境贴图来做到这一点[⊖]。实际上，光源无穷放大就形成其周围环境，同时，对那个点我们提前计算出所有这些光源和这个点的相互作用，并把结果缓存起来。可是在这么做之前我们要提醒自己，当考虑真实性时所有的环境贴图技术都受到两个限制。

首先，反射光作为物体大小的函数只是在几何上正确。环境贴图把从某个单独点照射过来的光线缓存起来，而且只有当物体无限小时——它退化为一个点——反射的图像才是正确的。第二，虽然它可以实现从环境——其他物体上反射回去的光，但是自反射不可能出现，而且如果物体不是凸的，那么显然就错了。

我们从回顾环境贴图和提出包括一种材质的 BRDF 的模型开始讨论。要注意，环境贴图是对所有照射到 3D 空间中的一个点的光线的缓存。对于放置在某个基准点上面的相对于它所处环境很小的物体，我们可以写成：

$$\text{沿视线观察方向 } \mathbf{V} \text{ 反射回去的光} = \text{沿方向 } \mathbf{R}_v \text{ 的入射光}$$

这里，$\mathbf{R}_v$ 是相对于位于物体表面上某点 $\mathbf{p}$ 的反射回去的视见向量。

$$\mathbf{R}_v = 2(\mathbf{N}.\mathbf{V})\mathbf{N} - \mathbf{V}$$

这里，$\mathbf{N}$ 是经过点 $\mathbf{p}$ 的表面法线。

这就是用于传统环境映射的模型，它模拟了一个理想镜面。要模拟出和它的 BRDF 相似的实际表面效果，可以使用反射方程：

⊖ 传统上，环境贴图一直用于渲染闪亮物体表面上的环境详细信息。

⊖ 本处的术语——预滤波有一点儿使人糊涂。它被同时用来描述将理想环境贴图转换成模拟一个实际表面的过程以及用来产生反走样的 mip 贴图。这是因为这两个操作都是用滤波或者是卷积操作生成的。

$$L_{ref}(\omega_{ref}) = \int_{\Omega} f_r(\omega_{in} \rightarrow \omega_{ref}) L_{in}(\omega_{in}) \cos\theta_{in} d\omega_{in}$$

这里，$f_r(\omega_{in} \rightarrow \omega_{ref})$ 是双向反射率函数（BRDF）。

积分的范围是以我们考察的点为球心的半球面，而且可以看到积分实际上在用 BRDF 计算从所有入射方向照射过来的光——一个过滤操作。这个方程提示了使用强制的预滤波包含：对每个目标环境贴图中的像素，我们进行一次包含所有源图像素的积分——一个复杂度为 N^2 的操作，这里 N 是图中像素的个数。

我们可以把 $L_{in}(\omega_{in})$ 看作未过滤的环境贴图，把 $L_{ref}(\omega_{ref})$ 仅仅看作过滤的或是目标环境贴图。这样对一个物体表面上的点，它存储了从物体向外射出的光线而不是存储了入射光信息的环境贴图。它实际上是 BRDF 围绕着的环境。

实现这个预滤波操作的一个很有用的方法由 Miller 和 Hoffman 在 1984 年提出［MILL84］，它计算单独的漫反射和 Phong 镜面反射贴图。这样就把 BRDF 拆分成一个半球面（理想漫反射体的 BRDF）和一个单独的镜面反射凸角。这样的两幅图使人们可以运用调和或是混合的因子实现一系列的效果。Phong 模型适合于简单的预滤波，因为它的镜面反射角对所有 $\mathbf{R}_v$ 都保持不变，而且它关于 $\mathbf{R}_v$ 径向对称。图 4-13 显示了预滤波过程的一个直观表示。这里我们考虑一个源环境贴图 $E(\mathbf{L})$ 和两个目标贴图 $D(\mathbf{L})$、$S(\mathbf{L})$。漫反射贴图 $D(\mathbf{L})$ 把所有对由 $\mathbf{N}$ 给出的半球面有作用的入射光线的总和缓存起来。镜面反射贴图 $S(\mathbf{L})$ 把所有 $\mathbf{R}_v$ 方向周围的入射光线的有用部分缓存起来，这些光线促成了沿视线方向 $\mathbf{V}$ 的镜面反射。

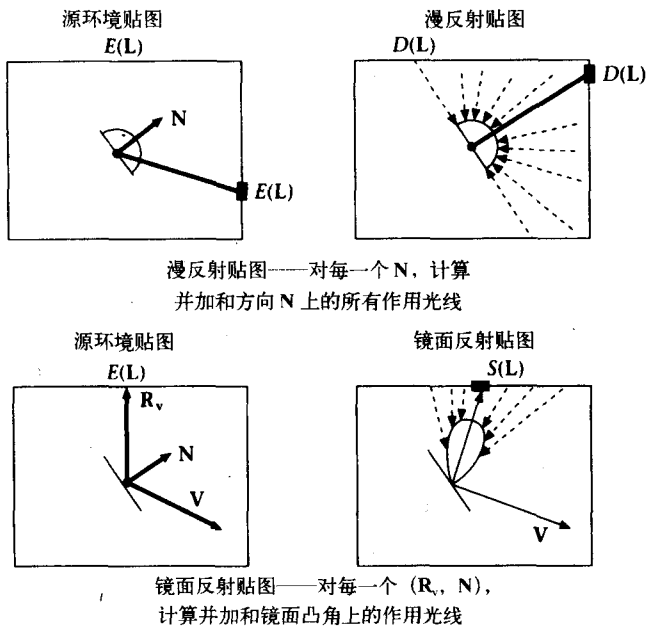


图 4-13　预滤波一个环境贴图

对一个理想的漫反射表面，它的 BRDF 简化为 k_d——Phong 反射模型中的漫反射系数，同时反射方程可以用“标准的”计算机图形学向量写成：

$$L_{\text{diffuse}}(\mathbf{N}) = k_{\text{d}}\int_{\Omega} L_{\text{in}}(\mathbf{L})(\mathbf{N}.\mathbf{L})\mathrm{d}\omega(\mathbf{L})$$

它只依赖于表面法向量。

Phong 模型把镜面反射项表示为：

$k_{\text{s}}(\mathbf{R}.\mathbf{V})^{n}$ 或是等价的 $k_{\text{s}}(\mathbf{R}_{\text{v}}.\mathbf{L})^{n}$

这里：

R 是反射光向量

V 是视见向量

$\mathbf{R}_{\text{v}}$ 是经过反射的视见向量

L 是表示光线照射方向的向量

k_{s} 是镜面反射系数

n 是发光指数

同时镜面反射贴图的方程变为：

$$L_{\text{specular}}(\mathbf{R}_{\text{v}}) = k_{\text{s}}\int_{\Omega}(\mathbf{R}_{\text{v}}.\mathbf{L})^{n}L_{\text{in}}(\mathbf{L})(\mathbf{N}.\mathbf{L})\mathrm{d}\omega(\mathbf{L})$$

对于漫反射贴图，对 **N** 的每个值我们把 **L**（以区域为权的点积或称为 Lambertian 项）的所有值相加起来。这形成了原来模拟漫反射的图像一个很模糊的版本——物体和场景的漫反射相互作用。在镜面反射贴图的情形下，这个操作也是产生原来图像的一个很模糊的版本，在这里图像模糊的程度取决于它的下标 n。当然要意识到这不是实现 BRDF 的常规方法。它只能用于类似 Phong 带有镜面反射凸角的 BRDF 的情况，这里的镜面反射凸角关于 $\mathbf{R}_{\text{v}}$ 中心对称，而且它的形状不是角度（$\mathbf{R}_{\text{v}}.\mathbf{N}$）的函数。

表面上某点的反射强度是：

$$k_{\text{d}}D(\mathbf{N}) + k_{\text{s}}S(\mathbf{R}_{\text{v}})$$

镜面反射的贡献可以进一步用一个近似的菲涅耳项来给出：

$$(1-F)k_{\text{d}}D(\mathbf{N}) + Fk_{\text{s}}S(\mathbf{R}_{\text{v}})$$

使用这个表达式表示，比如对于高折射指数（玻璃），镜面反射项对掠角占支配地位，而对于金属来说镜面反射项对所有角度都很高。

在包含动态物体的游戏应用中，随着物体在场景中移动我们需要重新计算环境贴图，并且还要对它进行预滤波。我们现在考虑这样做的原因。

首先，要注意到因为漫反射贴图的分辨率很低，所以需要对每一帧都创建一个新的漫反射贴图。第二，要注意到预滤波操作如果像在前面章节中所介绍的那样来实现，代价非常昂贵而且也不适合实时的应用。

对于快速的实时预滤波，Heidrich［HEID00］提出了一个使用纹理硬件的解决方法。这个方法的一个前提条件是需要将半球面上的不变平移滤波器内核映射到纹理空间中的不变平移循环内核。（Phong 模型有一个不变平移滤波器内核——一个固定的轴对称的镜面反射凸角。然而，如果使用立方图的话，那么把这个内核映射到纹理空间的映射就不是不变的。）如果可以得到这样一个映射，那么就可以使用 OpenGL 的成像子集——它支持 2D 不变平移滤波器。

Heidrich 指出抛物面贴图（4.6.3 节）近似地显示了这个性质，并且论证了扭曲依赖于

内核的半径以及它到贴图中心的距离。不过，内核的大小也随着它到贴图中心的距离而变化——那个映射不是平移的，建议的解决方法是生成两个预滤波的环境贴图，然后在作为到贴图中心的距离的函数渲染过程中把它们混合起来。

4.6 环境贴图参数化

要使用环境贴图我们需要对它进行参数化。常用的方法有立方体映射方法或是球面映射方法，这两个方法硬件都支持。立方体方法的缺点是它需要 6 幅图，而且如果 mip-mapping 单独用到每幅图中会看得出接缝。球面的方法则有对环境采样高度不统一的缺点，而且只适合视见方向和图建立的方向相同的情形。换句话说，随着视见方向的改变必须建立一幅新的图。最新的对偶抛物面贴图似乎克服了这两个“传统”方法的缺点。

4.6.1 环境贴图参数化：立方映射

立方映射的方法很流行，因为使用传统的渲染系统可以很容易地构造出（环境）贴图。实际上环境贴图是形成立方体的 6 个面（见图 4-14）。要生成一幅立方图，视点需要固定在物体的中心以接收环境贴图，同时从视点出发渲染场景的 6 个透视图。

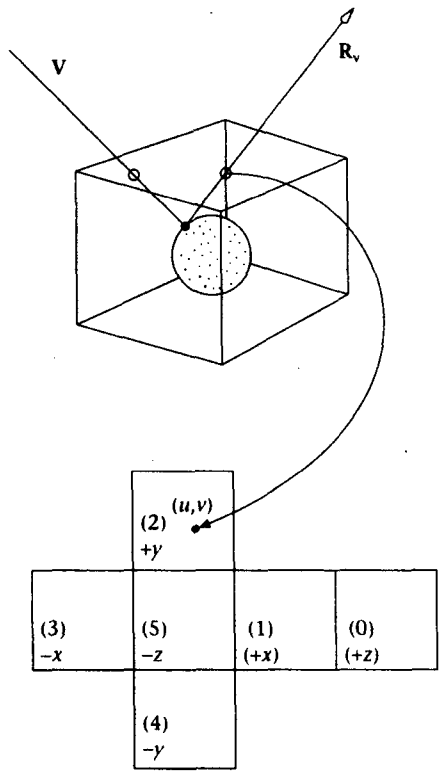


图 4-14　立方的环境映射或是立方体映射

有了环境贴图我们需要计算出从三维视见向量到其中一个二维贴图的映射。如果考虑反射的视见向量在环境贴图立方体的同一帧中，对一个标准化的反射向量 $\mathbf{R}_v$：

1）找到和它相交的面——贴图的编号。这涉及用数量值和符号来考虑 $\mathbf{R}_v$ 的组成部分。

2）把那个组成部分映射到（u，v）坐标。比如，如果 z 是最大的组成部分而且是负数的话，那么面 5（图 4-15（也见彩页））是和它相交的。这样纹理坐标可以这样给出：

$$u = x/z$$

$$v = y/z$$

图 4-15 显示了一个游戏环境中立方体映射的例子。那个茶壶是用简单的六色图和它自身的层次环境来进行立方体映射的。请注意就像你预见的那样，单个图并不会“折叠”成立方体。就是说，如果组成图像以它们的规划折叠到立方体的内部，那么它们就不能正确地连接到一起。这是上面用到的简单公式的结果，这要求组成图像的相对方向，如图所示。

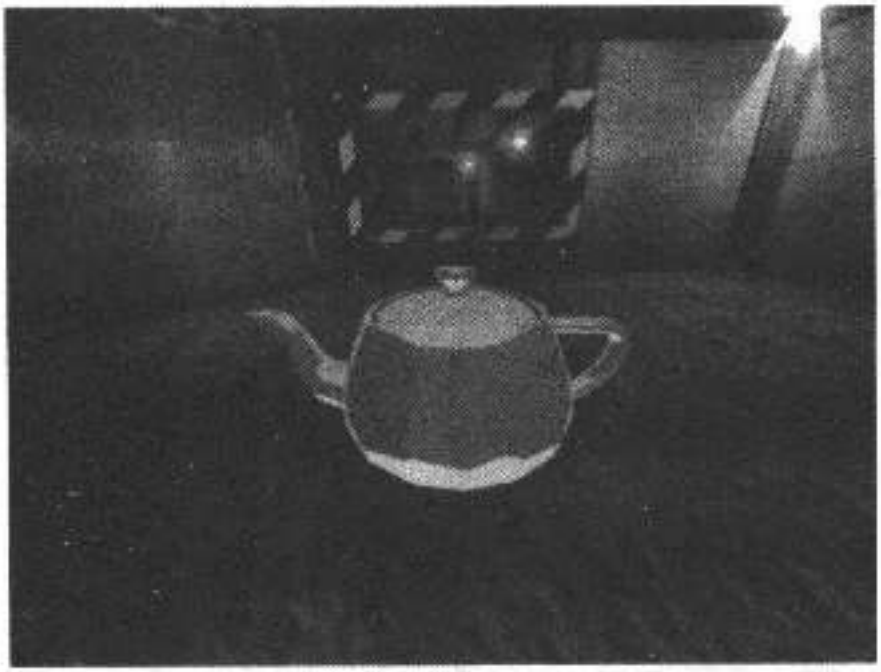
现在对于动态物体，在游戏的环境中运用立方体映射仍然代价太高，速度太慢——6 个视图必须对物体当前位置的每帧进行渲染。然而我们可以通过构造一个新的立方图，就是说，每秒 5 次或是在物体移动了一定距离以后减小这个代价。同时还可以使用动态纹理技术（pbuffer），就像我们将在第 6 章中讲述的那样。

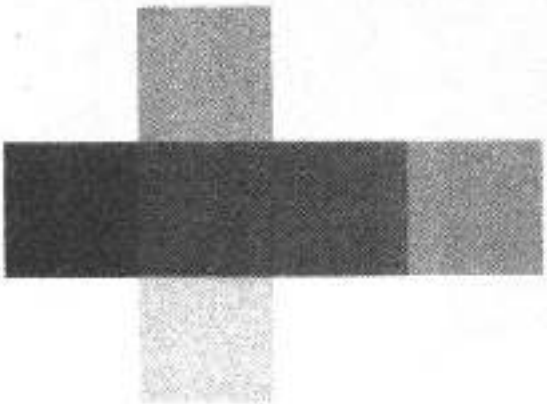
20世纪80年代流行起来的立方体环境贴图的一个应用是把一个生动的计算机图形物体"打磨"（matte）成真实的环境。在那样的情况下，环境贴图就根据真实环境的照片构造出来，同时（镜面反射的）计算机图形物体可以被打磨到场景中。这样好像它是场景的一部分，并且反映着它的周围环境。

4.6.2 环境贴图参数化：球面映射

Blinn 和 Newell［BLIN76］最先使用环境映射是在球面中，而不是立方体。环境贴图由经纬着色组成（见4.2.4节）。这个简单技术的主要问题是球面两极的独特性。

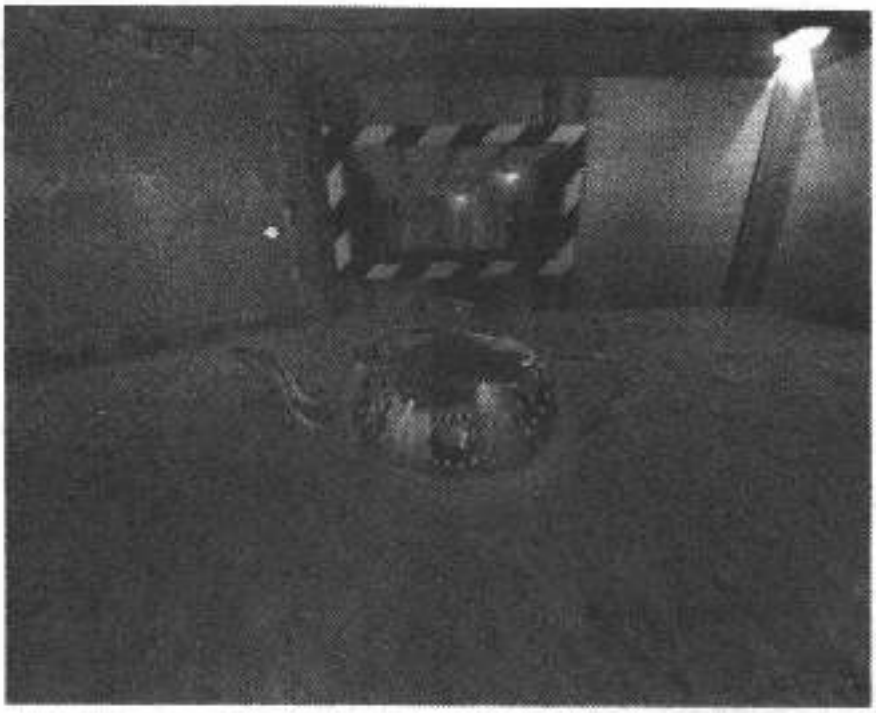
一个可选择的球面映射形式（在 OpenGL 中有支持）由圆形图组成，这个圆形图是对环

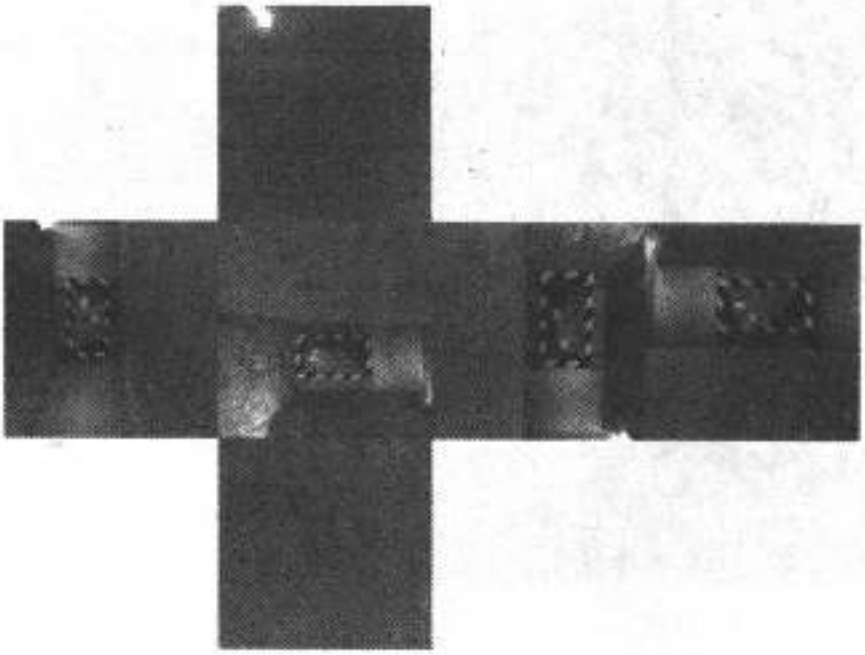




a)显示了一个使用立方体映射的物体，这里有图由6种颜色组成

图 4.15

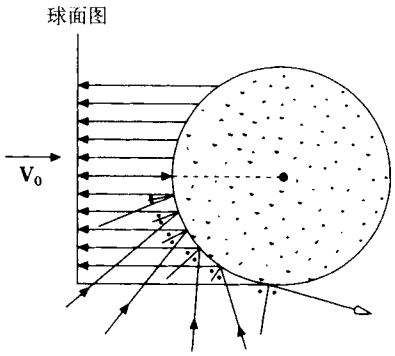




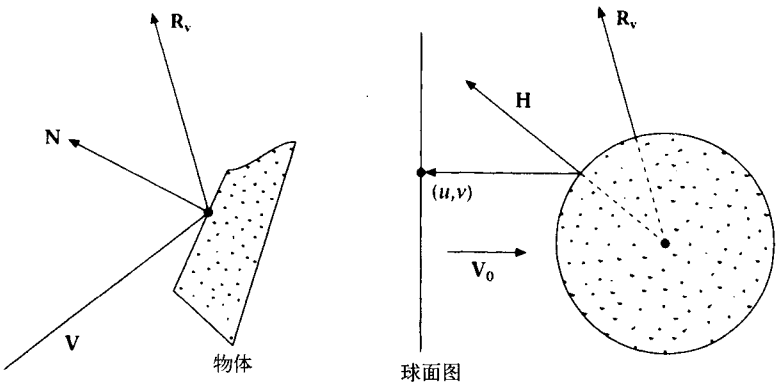
b) 同一个用立方映射的物体，它现在反映它所在的环境

图 4-15　（续）

境的反射的正交投影，就像我们在理想镜面上看到的那样，小到难以看出来的球面。这样一幅贴图是从它在图 4-16a 中结构的角度显示出来的。虽然这个图对照射到参考点上的入射光使用正交投影进行了缓存，在整个过程精确度的范围内，它还是可以用来生成一幅正常的透视投影。



a) 通过把一束平行光反射进环境中构建一个球面图



b) 访问那个图：计算 $\mathbf{R}_v$，计算 $\mathbf{H}$ 和 u，v

图 4-16 球面的环境贴图

为了索引到图中，我们还是像前面一样反射视见向量 $\mathbf{R}_v$。接着计算 $\mathbf{H}$，在参考视见方向 $\mathbf{V}_0$ 和 $\mathbf{R}_v$ 之间的半程向量。$\mathbf{H}$ 定义了一根法线，因而在虚拟球面上的点对应于图上的一个点，这个点存储了对于方向 $\mathbf{V}$ 光亮度的信息。假设参考的视见方向 $\mathbf{V}_0$ 在眼睛或视见空间中和 z 的负半轴相对应，那么纹理坐标就是在规范化的 $\mathbf{H}$ 向量的 x 轴和 y 轴分量，而且这些都是在硬件中计算的。

这个贴图可以通过投射出一束平行光线或是发自贴图纹素的光线来生成，它从虚拟的球面反射这些光线，并且把环境中被反射光线照射到的点分配到纹素中去。

详细细节如下：

$$\mathbf{H} = \frac{-\mathbf{V}_0 + \mathbf{R}_v}{\| -\mathbf{V}_0 + \mathbf{R}_v \|} = \frac{(0,0,1) + \mathbf{R}_v}{\| (0,0,1) + \mathbf{R}_v \|}$$

所需的球面图的（u，v）坐标就可以由规范化向量 $\mathbf{H}$ 的 x 和 y 方向分量给出，这个规

范化向量经过缩放、平移 1/2 以把原来的范围 [-1, 1] 转换为 [0, 1]。这样：

$$u = \frac{R_x}{2p} + \frac{1}{2}$$

$$u = \frac{R_y}{2p} + \frac{1}{2}$$

其中 $p = (R_x^2 + R_y^2 + (R_z + 1)^2)^{\frac{1}{2}}$。

有两个相互关联的问题和这种参数化有关，如图 4-16a 所示。这种方法对环境的采样率变化范围很大。随着我们由图，即一个圆的中心向外移动——采样率实际上在增加。同时在观察者前面的环境缩聚成了和虚拟切面相对应的一个小圆环：对应于我们的光线追踪模型中和球面相切的光线的图边界全部终止在观察者面前的同一点，同时那条边界由相同的像素点组成。这样，这种方法就只适合视见方向等于或靠近参考方向的情况。我们就是从这个参考方向建立起这幅图。

要生成一幅球面图，一个实际的方法是把立方图折叠起来 [BLYT00]。立方图的面和球面的关系如图 4-17 所示。这可以运用纹理映射来进行硬件加速。我们可以把它看作是使用光线追踪来生成（这幅图）的逆过程。立方图上的每个纹素（s, t）看作是一个反射向量的终点，这个反射向量由从虚拟球面反射回去的（参考）视见向量产生，同时上述的纹素（s, t）对应于球面图上的纹素（u, v）。Blythe 提出了一种技术，在这种技术中一个折叠网格被提前计算出来，全部细节参见 [BLYT00]。

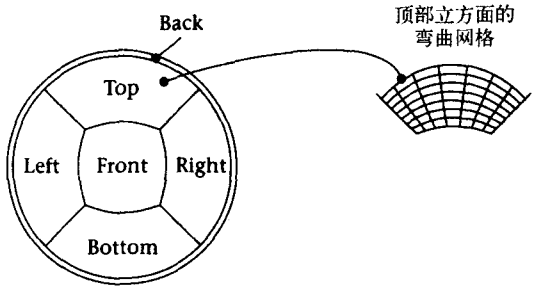


图 4-17　把一个立方图折叠成球面图

4.6.3　环境贴图参数化：对偶抛物面贴图

对偶抛物面贴图由 Heidrich 和 Siedel [HEID98] 首先提出来。它似乎解决了球面贴图的两个大问题——视见依赖和人造痕迹（参见下一节）。对偶抛物面贴图比球面贴图有更好的采样特征，而且不存在切线异常的问题。

它的缺点是需要两个图而不是一个，这就意味着需要四幅图来支持漫反射的合成和镜面反射的着色。

抛物面贴图背后的原理今天人们已经很熟悉了，它就像卫星接收天线圆盘。平行的入射光线被汇聚到一个点上。在对偶抛物面贴图中我们使

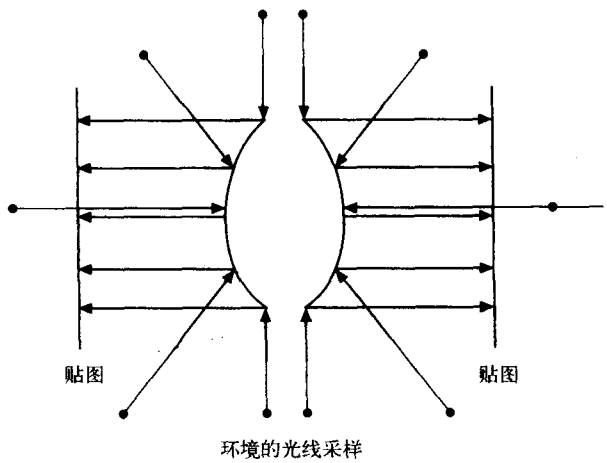


图 4-18　对偶抛物面贴图

用抛物面凸起的那面（见图 4-18），从纹理贴图照过来的平行光从焦点向外面的环境扩散，比在球面上更均匀。用一个正交投影照相机沿 z 轴的负半轴方向看到的影像由下式给出：

$$I(x,y)=\frac{1}{2}-\frac{1}{2}(x^2+y^2),x^2+y^2\leqslant 1 \tag{4-2}$$

它的优点的关键之处在于采样的光线是从一个点中射出来的，而在球面贴图的情形中，我们假设球面必须小到察觉不到。由于那幅图是视见独立的，我们认为物体的原点就是世界坐标系的原点，同时生成该图的视见方向沿 z 轴的负半轴方向。

依照［BLYT00］中的处理方法，在世界空间中，我们有反射向量：

$$\mathbf{R}_{v0}=\mathbf{M}^{-1}\mathbf{R}_v$$

这里，$\mathbf{R}_v$ 和以往一样是经过反射的视见向量，$\mathbf{M}$ 是模型/视图矩阵。

需要访问的贴图——前面或是后面——都由这个向量的 z 方向的分量给出，即无论它是朝向还是背向视点。$\mathbf{R}_{v0}$是位于抛物面上同一点 $\mathbf{x}$ 的向量 $\mathbf{D}_0=(0,0,1)$ 的反射向量，也就是说：

$$\mathbf{R}_{v0}=2(\mathbf{N}_0\cdot\mathbf{D}_0)\mathbf{N}_0-\mathbf{D}_0$$

这里 $\mathbf{N}_0$ 是抛物面在点 $\mathbf{x}$ 的法线，由下式给出：

$$\mathbf{N}_0=\frac{1}{(x^2+y^2+1)^{\frac{1}{2}}}\begin{bmatrix}x\\y\\1\end{bmatrix}$$

综合这两个方程，对某些值 k 就有：

$$\mathbf{D}_0+\mathbf{R}_0=\begin{bmatrix}k\cdot x\\k\cdot y\\k\end{bmatrix}$$

这样，最后这个向量用它的 z 方向分量除一下就可以得到我们想要的纹理坐标。

Heidrich 和 Siedel 指出，除了对 $\mathbf{R}_v$ 的计算以及最后的除法运算，所有的运算都是线性的；如果最后的除操作用一个透视除法来实现，那么这个转换就可以连接成一个纹理矩阵：

$$\begin{bmatrix}u\\v\\1\\1\end{bmatrix}=\mathbf{TPS}(\mathbf{M}_1)^{-1}\begin{bmatrix}R_{Vx}\\R_{Vy}\\R_{Vz}\\1\end{bmatrix}$$

这里：

$$\mathbf{T}=\begin{bmatrix}\frac{1}{2}&0&0&\frac{1}{2}\\0&\frac{1}{2}&0&\frac{1}{2}\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

是把［-1，1］区间上的 2D 坐标缩放，平移到区间［0，1］。

$$\mathbf{P}=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&1&0\end{bmatrix}$$

是影响除 z 操作的着色贴图。

$$\mathbf{S}=\begin{bmatrix}-1&0&0&D_{0x}\\0&-1&0&D_{0y}\\0&0&1&D_{0z}\\0&0&0&1\end{bmatrix}$$

从 $\mathbf{D}_0$ 中减去 $\mathbf{R}_0$。$\mathbf{M}_1^{-1}$ 是（仿射）的模型/视图矩阵线性部分的逆矩阵。这样，我们把 $\mathbf{R}_0$ 加到贴图中，它就输出前贴图或是后贴图的 2D 纹理矩阵，这个图依赖于 $\mathbf{D}_0$ 的方向。

对偶抛物面贴图可以采用类似于球面贴图的方法——通过把立方贴图的面折叠起来，如图 4-19 所示的那样建立起来，关于怎样完成这个过程的细节请参阅［BLYT00］。

图 4-19 强调了每一幅图都包含了环境的一个不完全的拷贝，而且这其中有重叠的信息。不过在这些重叠的部分中，对同一个环境的信息，其中的一幅图会比另一幅有更好的采样率。

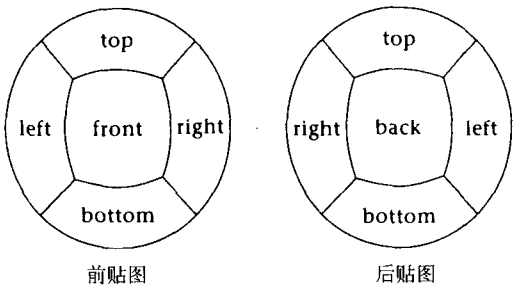


图 4-19　立方/对偶抛物面贴图的对应关系

鉴于我们有了一幅基本的环境贴图，我们需要一种方法来把它们综合起来。每幅图主要包含不在另一幅图中出现的细节信息。但是也会有重叠部分，这发生在中心圆以外的区域。考虑这样的圆：

$$x^2+y^2\leqslant 1$$

其中（x，y）是式（4-2）中的坐标点。

在每幅贴图中，完整的环境信息都包含在这个圆中。这就引出了以下的合并步骤。上面的贴图保证了如果一个反射向量映射到其中一个贴图的圆中，它将落在另一幅贴图中圆的外面。因此我们可以将在每幅贴图中位于该圆中的纹素的 α 通道编码为 1，而位于其外面的则编码为 0。对一个双通路的方法，我们首先考虑前面的纹理，使它可以进行 α 测试，然后用 $\mathbf{D}_0=(0,0,-1)$ 建立一个纹理矩阵，在第二通路中把 $\mathbf{D}_0$ 设置为（0，0，1）。

4.6.4　环境贴图——可比点

正如我们已经讨论的那样，几何上的不精确和所有环境贴图的参数化都有关系，这归因于这样一个假设，那就是正在被映射的物体小到察觉不到。这无可争议地引起了环境贴图不精确的问题最无可争议的理由，因为作为观察者，我们通常都不太清楚经过映射后，物体位于的场景究竟是什么形状。另外，使用不同环境方法也会引起非均匀采样的问题。随着该方向趋近于视见方向，球面贴图的采样趋近于 0，但是当该方向变为切线方向时，采样将变得非常大。

更成问题的是那些源自参数化自身特性的扰动以及使用硬件来对纹理进行插值。所有参数化中的非均匀插值导致环境贴图成为折叠起来的网格，而不是正常的网格。如图 4-17 所示的那样。不过，纹理插值假设纹理的坐标是适当的。在球面映射中，这会导致另外一个明显的人造痕迹——绘制出的物体轮廓边会闪烁。这是因为直角坐标可能会在球面贴图的圆中插值，而不是沿着球面的边界折叠起来。

随着用于立方贴图硬件的进步，六幅图附加的复杂度对用户来说就是很清晰的了，而且只需要一个纹理操作就可以了。这个结果使我们可以容易得到构造和使用视见独立的设备。就像我们前面看到的那样，球面贴图是视见依赖的，而且虽然对偶抛物面贴图是视见独立的，但是要构造一个这样的贴图远比构造一个立方贴图要难，而且它还需要两次纹理操作。

4.6.5 立方贴图和向量规范化

立方贴图可以用来实现使用在插值性着色的向量规范化中，在这个着色中对每个像素点都要求进行规范化操作。这个概念是很简单的。在传统的环境贴图操作中我们用反射的视见向量索引到贴图中。操作的结果是一个三元组——一组 RGB 颜色。当然我们也可以返回一个索引向量规范化的结果。这样环境贴图就变成了一张用向量索引起来的查找表，它返回它规范化的值。

我们使用在 4.6.3 节中讲述过的同一个逆映射来构造向量规范化纹理贴图的值。每一个 (u, v) 纹素位置都加上一个适当的面索引来定义出将会被索引到立方图上这个位置的向量。接着这个向量经过规范化存储在纹理贴图中。在这个操作阶段所有拥有相同方向但是不同大小的向量将会被索引到这个位置。一个细节是规范化的向量位于 $[-1, 1]$ 范围内，但是纹理贴图在 $[0, 1]$ 范围中取值。这样，当预计算规范化值的时候就需要使用一个范围压缩变换，而当要访问该图时则要使用逆向变换。

这是很有用的，比如，半程向量如下的实时着色可以使用立方贴图来进行规范化，同时随着 **L** 和/或 **V** 变化使用逐像素镜面反射着色操作。

$$\mathbf{H} = \frac{\mathbf{L} + \mathbf{V}}{\| \mathbf{L} + \mathbf{V} \|}$$

4.7 实现 BRDF：可分离的近似

上一节中我们考察了通过把 Phong 局部反射模型分离成两个部分环境贴图或立方贴图的方法来实现它的简单情况。本节将基于 Kautz 和 McCool [KAUT99] 及 Wynn [WYNN00] 的报告以更一般的视角来考察 BRDF。

因式分解方法（4.3 节）分解参数的 BRDF 或是着色模型——这个模型用 **L** 和 **V** 项解析地定义，这样它们就可以被预计算出来并存储在纹理贴图中。本节中我们要考察一下任意一个 BRDF——物理测量的数据，比如说，能够从一个 4 维的函数降到 2 维的函数，而且这样的话也能在纹理硬件中实现。

我们要问的第一个问题是：BRDF 是怎样建模的？人们在这个领域已经做了很多的研究，我们将只是通过给出一些有代表性的例子来回答这个问题。首先一个显然的数据源是测量的

数据和公共的数据库⊖。数据是采用一个测角反射计来采集的，测角反射计机械地改变微小光源的位置，同时一个光谱探测器在半球面上进行采样。其次，我们可以通过假定表面的一个特殊的物理模型并且运用，比如说光线追踪那样的光-面相互作用的算法，来预计算出所有可能的入射和出射方向来生成一个 BRDF。Cabral 在［CABR87］中给出了这种方法的一个早期的例子。在这篇论文中，面是用一列三角形的微面来表示的，这些微面的方向通过扰动一个凹凸贴图的顶点来设定。这种方法有时也称为虚拟测角反射计。

假设因此我们有了一组表示一个 BRDF 的采样值，接下去这样做，把一个 BRDF 看成是一组以表或是矩阵形式排列的采样值，矩阵的每一行表示一个固定的出射方向 ω_{ref}，同时每一列表示一个固定的入射方向 ω_{in}。该矩阵中的每一个元素是一个单样本的 BRDF（θ_{in}，ϕ_{in}，θ_{ref}，ϕ_{ref}）。对于一个 BRDF 的 n^4 个样本，矩阵中第一行的 n^2 个元素包含：

$$\mathrm{BRDF}(\theta_{in,0},\phi_{in,0},\theta_{ref,0},\phi_{ref,0})\mathrm{BRDF}(\theta_{in,0},\phi_{in,1},\theta_{ref,0},\phi_{ref,0})$$

$$\mathrm{BRDF}(\theta_{in,1},\phi_{in,0},\theta_{ref,0},\phi_{ref,0})\cdots\mathrm{BRDF}(\theta_{in,n},\phi_{in,n},\theta_{ref,0},\phi_{ref,0})$$

需要注意的是对任何可能的采样方法，这个数对现有的硬件都大到无法承受。（例如，一个相对低分辨率的 64^4（$\times 3 bytes$）样本最终会产生一个 50MB 的 4 维查找表。

这个过程的目标是把 BRDF 表示成可分离的分解，在这里它表示成低维函数 G_k 和 H_k 乘积的和：

$$\mathrm{BRDF}(\theta_{in},\phi_{in},\theta_{ref},\phi_{ref}) \approx \sum_{k=1}^{N} \mathrm{G}_k(\theta_{in},\phi_{in})\mathrm{H}_k(\theta_{ref},\phi_{ref})$$

我们特别关注 N 有较小数值的分解，如果可能，我们希望获得 $N=1$ 的表示：

$$\mathrm{BRDF}(\theta_{in},\phi_{in},\theta_{ref},\phi_{ref}) \approx \mathrm{G}_k(\theta_{in},\phi_{in})\mathrm{H}_k(\theta_{ref},\phi_{ref})$$

虽然如果使用足够项的话，任何一个 BRDF 都可以正确地表示出来，但是我们对单个项分解比较感兴趣。已经证实了只要使用一个“好的”参数化，这完全是有可能的。而且这样我们就可以对使用任意 BRDF 的实时渲染使用纹理硬件。

这个方法的两个主要限制是：

1）必须选择一个能够产生好分解的参数化；这样近似：

$$\mathrm{BRDF}(\theta_{in},\phi_{in},\theta_{ref},\phi_{ref}) \approx \mathrm{G}_k(\theta_{in},\phi_{in})\mathrm{H}_k(\theta_{ref},\phi_{ref})$$

才是合理的。Kautz 等人指出球面参数化对大多数，但不是全部的 BRDF 都很好用，一个比较好的参数化是 Gram-Schmidt 半角差值向量参数化。

2）这个参数化必须和 2D 纹理贴图的线性插值一起使用。

⊖ Columbia-Utrecht 反射和纹理数据库。
这和 4.2.3 节中讲述的是同一个数据库。实际上，它同时包括了 BRDF 和 BTF。请参阅［DANA99］以获得这两个函数不同之处的准确描述。这个数据库包括：
1）对超过 60 种不同样本反射测量的 BRDF（双向反射分布函数）数据库，每个样本都在超过 200 种视见和照明方向的组合下进行观察。
2）来自两个最新 BRDF 模型的，带有适当参数的 BRDF 参数数据库：Oren-Nayar 模型和 Koenderink 等表示。这些 BRDF 参数可以直接用于图像分析和图像合成。
3）取自 60 个不同样本的带有图像纹理的 BTF（双向纹理函数）数据库，这里的每一个样本都在超过 200 种不同的视见和照明方向的组合下进行观察。

因此，这个方法就是一个预处理的过程，在这个过程中 BRDF 被采样或生成出来，然后被分解成为一个 2×2D 的纹理贴图。

传统的分解方法是单值分解法或者称为 SVD。Kautz 等人[KAUT99] 指出虽然这个方法能够产生出最优化的近似，但是它在时间和空间上的代价非常高昂。同时他提出了一个比较简单的技术，称作规范化分解或称为 ND。他们指出，当考虑视觉效果时，在大多数情况下，单个项的 ND 效果和使用单个项的 SVD 的效果一样好。我们把 ND 定义成：

对上面讲述的 BRDF 矩阵的每一行 R，计算 p 范数：

$$G_1(\omega_{\text{in}}) = \sum_{j=1}^{n^2} |\text{BRDF}|^p(\omega_{\text{in},j}, \omega_{\text{ref},R})^{\frac{1}{p}}$$

对 BRDF 矩阵的每一列 C 计算平均值：

$$H_1(\omega_{\text{ref}}) = \frac{1}{n}\sum_{j=1}^{n^2} \frac{\text{BRDF}(\omega_{\text{in},C}\omega_{\text{ref},j})}{G_1(\omega_{\text{in}})}$$

就是说每一列都用它对应行的范数来规范化。

我们现在来考察一下参数化的问题。到现在为止我们使用的一直是标准的参数化——球面坐标——这里 $(\theta_{\text{in}}, \phi_{\text{in}}) \times (\theta_{\text{ref}}, \phi_{\text{ref}})$ 规定了和局部表面框架有关的入射和反射方向。这不适合 2D 贴图的标准硬件线性插值，但是如果使用立方贴图，就能减少插值形成的人工痕迹。由于 BRDF 只定义在入射方向的半球面上，只需要使用立方贴图的上半部分 $(+z)$。只要和一个纹素相对应的 (θ_t, ϕ_t) 有一个 BRDF 样本，把 G 和 H 函数映射到立方贴图是很直接的（参见 4.6.1 节）。否则必须使用类似双线性插值的步骤。

一个很重要的问题是动态的范围。BRDF 可以取很大的值，同时保留这些未改变的数据将降低分离的质量。这样，所有的值都必须限制在一个预先确定的最大范围之内。我们有：

$$\text{BRDF} = \delta\hat{G}\hat{H}$$

这里 δ 是一个浮点的缩放比例值，$\hat{G}$ 和 $\hat{H}$ $(\in [0, 1])$ 分别是 G_1 和 H_1 除以最大值后得到的值。

这对于实时执行又提出了一个问题，那就是我们在纹理合成的过程中不能用一个任意的浮点数来缩放，而且 Wynn [WYNN00] 提出了以下的函数：

$$\text{BRDF} = D\frac{S}{D}\hat{G}\hat{H} = D\left(\sqrt{\frac{\delta}{D}}\hat{G}\right)\left(\sqrt{\frac{\delta}{D}}\hat{H}\right)$$

$$= D\hat{g}\hat{h}$$

这里 D 是 (1, 2, 4) 的最小值，因此 $\delta < D$ 或 $D = 4$（如果 $\delta > D$）。

假定 BRDF 数据和计算 G 和 H 的预处理过程，我们的反射方程就成为：

$$L_{\text{ref}}(\omega_{\text{ref}}) = \int_{\Omega} DG_1(\omega_{\text{in}})H_1(\omega_{\text{ref}})L_{\text{in}}(\omega_{\text{in}})\cos\theta_{\text{in}}\,\text{d}\omega_{\text{in}}$$

图 4-20（彩页中也有）说明了这样的一个例子。





抛光盒和各向异性盒





橙色外壳

图 4-30　在同样的光照下使用可分离的近似方法渲染的不同材质

4.8 着色语言和着色器

前面的章节中我们考察了不同的渲染方法——对物体表面进行着色的不同方法。本章最后的一个论题是着色器（shader）。此处将提供一些工具，它们使得不同的渲染部分可以结合在一起以产生期望的效果。“着色器”是一个广泛使用的术语，它有很多不同的表现。同时，因为硬件的发展，在最近的计算机游戏中我们对它也很有兴趣。就当前而言这可能意味着3个不同点，但是在渲染方面却是相互关联的。

1）它在 RenderMan API 中最初是表示（而且现在仍然表示）一个模块（使用类 C 语言写的），这个模块使我们可以实现高层次上对渲染组件的控制。这里这个词指的是一个特殊的代码模块——一个面或是光线着色器。这些都规定了怎样按照一个面的材质和反射模型参数（表面着色器）来对它进行着色，以及怎样计算出光源的强度和色彩以备表面着色器的使用。同任何专门的高级语言一样，它的目标是使程序员能够编写程序，并且以一种更简单的方式对直接渲染的可能性进行实验。这方面的内容将在接下来的章节中介绍。

2）最近在游戏技术中，它被用来指在固定功能的图形硬件上使用多通路/多纹理功能（facility）。渲染状态的详细说明和对一个物体多通路/多纹理的渲染就是我们所知道的着色器。这里，我们可以使用一个着色编辑器来对这个效果进行实验，而不需要编写代码模块，这里着色编辑器的最后输出就成为一个物体的着色器或是物体的一部分。这个用处将在第5章的第一部分讨论。

3）更近一些时间，硬件生产商采用了术语“顶点着色器”和“像素着色器”来描述可编程 GPU 的硬件功能，使用这些功能我们可以编写操作这些图形硬件的低层次的代码。这方面的内容将在第5章中讲述。

最近的研究［PROU01］主要关注于第1方面和第3方面的内容，这里，人们可以使用高层次的 RenderMan 型的语言和相关的编译器，而不需要通过编写汇编代码来控制可编程 GPU 的功能。这就导致了现在这样一个不幸的局面，所获得的可编程 GPU 都需要用每个厂商特定的汇编代码来编写。

虽然这样的策略目前还只限于研究机构，但是很可能在 OpenGL 2.0（和 DirectX）中会出现一种可编程图形硬件的着色语言。总之，建立这一标准的目的是提供对图形处理器硬件无关的访问。不过随着可编程 GPU 的出现，带有一大堆专用的（厂商）扩展，OpenGL 只是变得更麻烦了。

4.8.1 着色语言：简单的历史回顾

使用一个带有基本功能的渲染器是件很简单的事情，它只涉及对选项的选择——Gouraud 或是 Phong 着色，有阴影或是没有阴影，等等。对一个提供了许多反射模型、纹理贴图、纹理混合、透明度等的渲染器，“着色语言”是控制这些选项的一种好方法。

Cook 在［COOK84］中首次提出了这样一个思想，然后逐渐发展成为 RenderMan［PIXA88］，［UPST89］的设计。我们在这里直接从［UPST89］中引用原文。关于着色语言的重要性，Upstill 如是说：

> 存在另外一个选择，它基于让用户通过更多地访问着色系统本身，而不是它的外部接口。关键的事情是让用户能访问系统中有用的部分，而不必用一些无关的信

> 息来加重它们的负担……通过使用着色语言编写出一个适当的着色器，程序员可以扩展旧有的着色模型或者构建一个全新的着色模型。光源可以用任何放射的分布来定义，而且可以很容易引入新奇的曲面特性。这些过程的任何参数都可以设定为一个常数值，这个值可以在曲面上平滑地变化，或者用一个曲面图任意调整。

Cook 的着色语言方法包含着消除“固定”的反射方程，就像带有自身项线性组合的 Phong 模型包含点积一样，同时这个方法使得我们可以指定组件以及这些组件相互结合的方式。这种方法考虑到了现有方法学的规格说明及一个可以用来实验新组合的测试平台。Cook 把着色过程分离为概念上独立的光源规格说明的任务、表面反射和大气影响。

表面反射的规格说明被证明是非常灵活强大的。这可以使用一个把着色操作组织成树状结构的着色树来实现。提出使用一棵树的想法是为了方便把一个给定点的若干效果和整个过程综合起来。我们假设树的根在顶部，而叶子在底部。和着色过程有关的数值（Cook 把它们称为“外观参数”）在树的叶子处生成。接着这些数值往上传递，然后在节点处进行处理。节点从这些值中取出一个值或更多值，然后把它们综合起来生成一个值，这个值依次往上传递。最后到达树顶，输出是由根节点往上传递的数值，这也是对那个面最后着色的颜色值。

比如，把 Phong 着色的操作分解成一棵着色树给我们一个镜面反射的节点和一个漫反射的节点。漫反射节点的输入是表面的法线以及生成一个强度值的光线向量。每个镜面反射节点有三个值：表面法线、眼睛的位置和表面的粗糙度，并输出一个镜面反射分量。这样，标准的 Phong 反射模型就可以表示成一棵树（见图 4-21）。

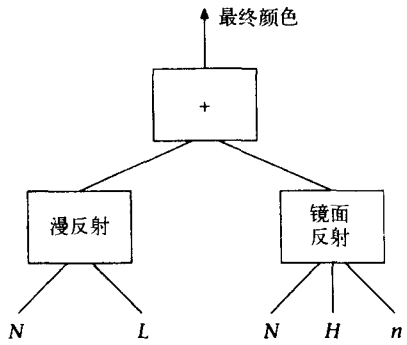


图 4-21　表示成一棵着色树的 Phong 反射模型

Perlin 在［PERL85］中拓展了 Cook 的思想，它构造了一种嵌入在称为像素流编辑器（或称为 PSE）的环境中的着色语言。PSE 比最初的着色树提供的那些编辑器允许更多的控制结构流。该语言支持条件结构和循环控制结构、函数定义和逻辑运算。使用这种系统的结果是使渲染过程能分解成中间图像或阶段，这里每个阶段都是经过 PSE 的一个通路。

4.8.2　RenderMan 着色语言

前面讨论的思想逐渐归纳成了 RenderMan 中一种完整的着色语言［PIXA88］、［UPST89］、［HANR90］。众所周知，Pixar 成功地使用 RenderMan 制作了动画长片，例如《玩具总动员》和《虫虫特工队》。这些产品的高质量证明了这种方法的成功；同时记住一点：RenderMan 并没有包含任何全局的照明模型——这些模型的使用一般和高质量的渲染相联系。

在构造 RenderMan 的诸多动机中，Hanrahan 讨论了以下几点。这种语言应该含有某些窍门，提供测试不同效果的平台。举例来说，许多着色中的效果都是通过纹理和局部反射模型的不同组合得到的。我们必须认清这个事实，并且在开发这种语言中的小窍门时方便地运用这种操作。它就像支持程序和模块那样支持着色器的概念。

[HANR90] 中陈述这种语言的目标是：

1）开发出一种基于光线光学抽象着色模型，这里的光线光学同时适合于全局和局部照明模型。就独立于特定的算法或者是硬件或软件的实现而言，它也是抽象的。

2）定义渲染程序和着色模块间的接口。所有逻辑上内建的着色模块可以访问到的信息，对于着色语言的用户来讲也应该可以访问到。

3）提供一种易用的高级语言。它应该拥有一些功能——点和颜色类型及算子、积分语句、内建函数——这些特征使得我们可以自然地表示着色计算。

这种语言使我们可以编写出被称为着色器的程序。这些着色器模拟了局部的过程，而且可以被单独使用或是一起使用——若干种着色器一起使用以生成最后的图像。着色器通过相应的输入和输出来区分类型。三种主要的着色器是：

1）光源着色器：这种着色器计算从光源射出的光线的颜色。它们把光源的位置和光线射向曲面点的方向作为输入，输出打在表面点上的光线的颜色。典型情况下，光源会有一个颜色、一个强度、一个方向依赖以及一个和距离有关的衰减。

2）表面反射着色器：这些着色器实现了一个局部反射模型。同时通过加总所有入射的光线以及考虑表面的反射特性来计算出某个方向的反射光。它们对入射光源分布没有任何假设，这些光源的分布可以直接来自于一个光源或是由另外一个物体反射过去的二级光。

3）体着色器：这些着色器实现光穿过一个体的效果。这可以是物体外部的一个体（在这种情况下着色器就是一个大气着色器），也可是物体内部的一个体。

这些实体结合在一起的方式的表现如图 4-22 所示。

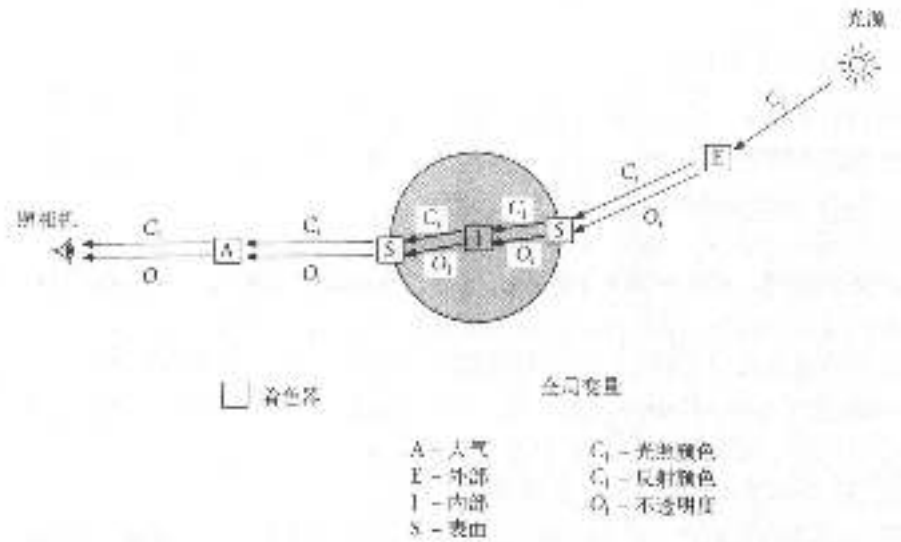


图 4-22 着色器的 RenderMan 数据流

下面是从 [UPST89] 中摘抄的 RenderMan 表面着色器的一个实例：

```
surface
plastic()
  float Ks                = .5,
```

```
        Kd              = .5,
        Ka              = 1,
        Roughness = .1,
  color specularcolor = 1)
{
point Nf = faceforward(N,I);

Oi = Os;
Ci = Os *(Cs * (Ka*ambient() +Kd*diffuse(Nf)
   + specularcolor*Ks*specular(Nf, -I, roughness));
}
```

这只是其中一个 RenderMan 标准（或预定义）着色器的代码，它实现了标准的 Phong 着色。如果你熟悉 Phong 反射模型（4.2.1 节），应该能读懂这里大部分的代码。预定义的全局变量用来在两个着色器之间传递信息或计算结果。Ci 表示一个着色器的输出结果，Cs 是当前的表面颜色，或反射系数，它受物体的约束。从点反射回来的光是Ci，一般这是一个含有入射光线和Cs 的表达式。它最简单的形式是：

$$Ci = Os * Cs * \quad \text{（一个反射模型的具体实现）}$$

Os 是表面的不透明度，一个完全不透明的物体的不透明度设为 1。不透明度Oi 和光线有关，而且通常的表面着色器会包含这样的赋值：

$$Oi = Os$$

把不透明度和光线联系起来是一个令人感到困惑的概念，但是透明度由此而产生。

[UPST89] 中给出了详细描述的着色器的更多实例。从这里可以看出这种语言表面上和 C 语言很相像。着色器是通过在它的定义之前先定义一个着色语言关键词：光、位移、表面或体来定义的。

4.8.3　实时渲染的着色语言

正如我们将要在下一章中看到的那样，复杂的效果现在也能在可编程的 GPU 上实现了。同时，GPU 的低级编程却一点也不直接。另外，新的硬件生产商很可能开发出新的功能。显然这要求着色语言被编译后能在许多 GPU 上运行[⊖]。

在最近通报的成果中，Proudfoot 等人 [PROU01] 正好提出了这个问题。他们使着色器能够用高级语言编写，这有点像 RenderMan。（第 5 章详细解释了其中一种 GPU 的编程模型，该语言就是为它而设计的，在阅读本小节时你可能要参看这一部分的内容。）

虽然传统高级语言存在的目的就是要把用户从低级语言的复杂性中解脱出来，但是在可编程 GPU 的情形下还有一些额外的复杂性——可编程的硬件处在图形管道的不同阶段。这反映了硬件的结构，有些计算在 GPU 中进行，有些则在顶点编程硬件或是像素编程硬件中进行。特别地，Proudfoot 支持四种计算频率：

- 常数——编译过程中只计算一次的表达式，并且不用可编程的管道再计算。
- per-primitive——每组原语计算一次。
- per-vertex 和 per fragment——映射到硬件的各个顶点和各个像素部分。

从用户的程序中推断计算频率是编译器的职责。这种语言使程序员能够把这些计算混和

⊖ 在游戏业中为了对付不同的硬件平台、控制台或是 PC 的 GPU，我们消耗了大量的人力。

到一个着色器中。这样就让用户从必须为每个硬件阶段编写不同的代码的困境（甚至还不得不弄清楚在不同硬件上以不同计算频率进行的计算）中解脱出来。这通过编译器的前端把程序分割成三种类型的计算并对每个计算组和每组中不同的结构调用单独的编译器后端来实现[1]。

和在 RenderMan 中一样，该系统支持两种常见的着色器类型：表面着色器和光着色器。表面着色器返回一个 RGBA 颜色，这个颜色将被合成到帧缓冲区中。光着色器计算光照的强度和表面着色器使用的光的颜色。在这种着色语言中，线性积分操作对每个光源计算出"per light"表达式，并把它们加总得到最后的结果。比如：

```
surface float4
lightmodel_diffuse(float4 ka, float4 kd, )
{
  perlight float Ndotl = max(0, dot(N,L));
  return ka * Ca   integrate(kd * Ndotl * Cl);
}
```

对所有有效的光源进行漫反射着色操作。

接下来的一个表面着色器的例子将使我们能够看到这种语言的体系和语法，「PRO001」给出了这个例子的细节。所有的例子都是使用从斯坦福大学实时可编程着色项目处得到的一个出色系统生成的（www.graphics.stanford.edu）。

图 4-23（彩页中也有）展示了最简单的可能的着色器。它是把标准 Phong 着色参数设为常数，同时使用 lightmodel() 方法来计算每个像素的颜色，对所有照得到的或是有效的光进行积分。

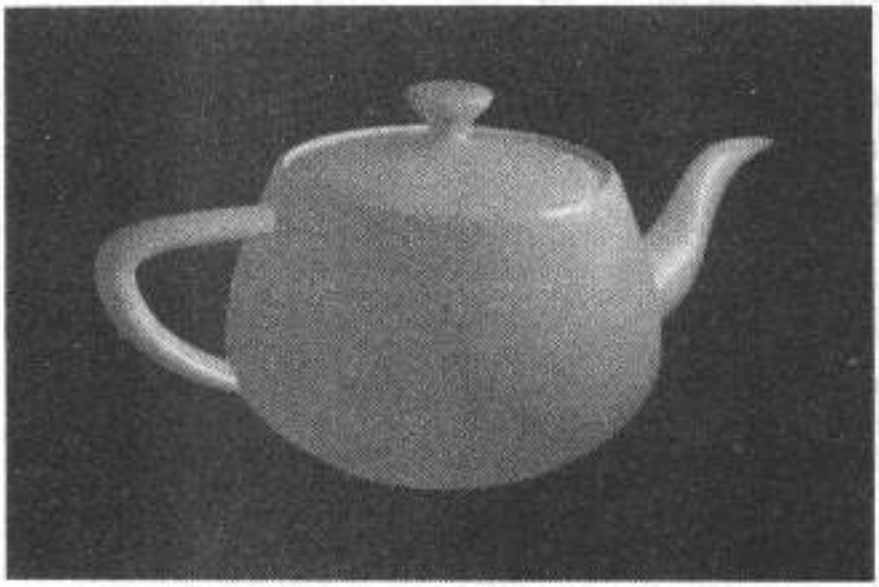


图 4-23 最简单的曲面着色器——Phong 着色，使用从斯坦福大学实时可编程着色项目处得到的一个出色系统生成（www.graphics.stanford.edu）

[1] 它们当前的实现包括两个不同的 CPU 后端，二个顶点后端和两个像素或是片断后端。

```
surface float4
lightmodel (float4 a, float4 d, float4 s, float4 e, float shi
{
  perlight float diffuse = dot(N,L);
  perlight float specular = pow(max(dot(N,H),0),shi);
  perlight float4 tr = d * max(0, diffuse, 0) +
                       s * selectdiffuse > 0, specular, 0);
  return a * Ua + integrate(tr * C_) + e;
}

constant float4 Ma = { 0.35, 0.35, 0.35, 1.00 };
constant float4 Md = { 0.50, 0.50, 0.50, 1.00 };
constant float4 Ms = { 1.00, 1.00, 1.00, 1.00 };
constant float4 Me = { 0.00, 0.00, 0.00, 0.00 };
constant float  Msh = 333;

surface shader float4
default ()
{
  return lightmodel(Ma, Md, Ms, Me, Msh);
}
```

第二个例子（见图4-24，彩页中也有）显示了一个简单的变体，这个变体中环境和镜面反射的颜色在同镜面反射两者都已经改变了。

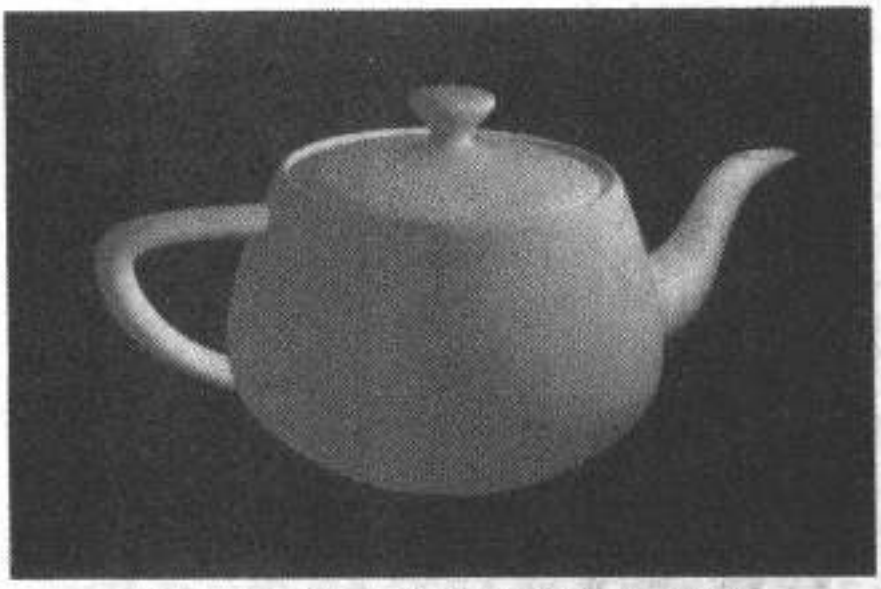


图4-24　图4-23的一个变体。使用从斯坦福大学实时可编程着色项目处得到的一个白色系统生成（www.graphics.stanford.edu）

```
constant float4 Ma = { 0.35, 0.25, 0.25, 1.00 };
constant float4 Md = { 0.70, 0.70, 0.70, 1.00 };
constant float4 Ms = { 0.00, 0.00, 1.00, 1.00 };
constant float4 Me = { 0.00, 0.00, 0.00, 0.00 };
constant float  Msh = 50;

surface shader float4
```

```
Default ()
{
  return lightmodel(Ka, Kd, Ks, Ke, Ksh);
}
```

图4-25（彩页中也有）显示了两种不同材质的混和，一种是有光泽的，另一种是漫反射的。漫反射材质作为基底或是背景材质。有光泽的材质则在所给的纹理图是白色（纹理图作为黑色或是白色的掩模（mask））的地方加入到背景材质中去。最后的结果用一个综合的表达式来计算，该表达式把基底材质（用一个漫反射模型来计算）和有光泽材质同纹理图颜色的乘积相加。这样黑色的纹理像素点就不会被渲染成有光泽的，因为它们乘以0（有光泽的部分只出现在纹理是白色的地方）。



图4-25 混合有光泽的材质和漫反射材质，使用从斯坦福大学实时可编程着色项目处得到的一个赤色系统生成（www.graphics.stanford.edu）

```
surface shade( float )
glossy_mocap (texref gloss, float4 uv)
{
  float4 base_a = ( 0.1, 0.1, 0.7, 1.00 );
  float4 base_d = ( 0.70, 0.40, 0.10, 1.00 );
  float4 base_s = ( 0.07, 0.04, 0.01, 1.00 );
  float4 base_e = ( 0.00, 0.00, 0.00, 1.00 );
  float base_sh = 15;

  float4 gloss_a = ( 0.07, 0.04, 0.01, 1.00 );
  float4 gloss_d = ( 0.07, 0.04, 0.01, 1.00 );
  float4 gloss_s = ( 1.00, 0.90, 0.60, 1.00 );
  float4 gloss_e = ( 0.00, 0.00, 0.00, 1.00 );
  float gloss_sh = 25;

  float4 Cbase = lightmodel(base_a, base_d, base_s, base_e, base_sh);
  float4 Cgloss = lightmodel(gloss_a, gloss_d, gloss_s, gloss_e,
```

```
gloss_sh);

    float4 uv_gloss = invert(scale(.335,.335,1)) * uv;
    return Cbase + Cgloss * texcube(gloss, uv_gloss);
}
```

在图 4-26（彩页中也有）中，效果被颠倒了，有光泽的效果被用在了纹理图是黑色（0）的像素点上。纹理图白色的区域用简单的漫反射材质来映射。结果是月牙形区域被渲染成漫反射的，而背景则被渲染成了有光泽的。这可以通过颠倒纹理的颜色并把它作为有光泽材质的相乘因子来实现。

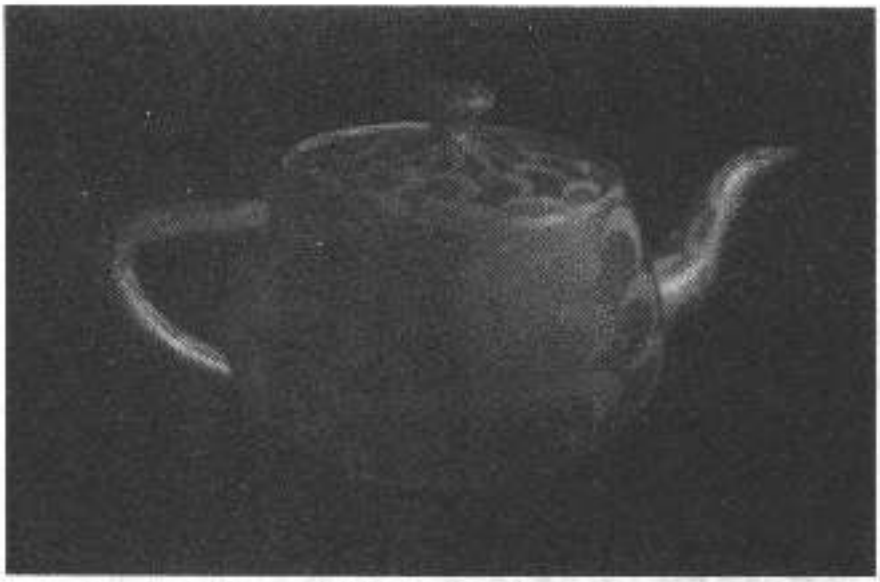


图 4-26　颠倒图 4-25 中的材质，使用从斯坦福大学实时可编程着色项目处得到的一个白色系统生成（www.graphics.stanford.edu）

```
surface shader float4
glossy_moons (texref gloss, float4 uv)
{
    float4 base_a = ( 0.1, 0.1, 0.1, 1.00 );
    float4 base_d = ( 0.90, 0.40, 0.10, 1.00 );
    float4 base_s = ( 0.07, 0.04, 0.01, 1.00 );
    float4 base_e = ( 0.00, 0.00, 0.00, 1.00 );
    float base_sh = 15;

    float4 gloss_a = ( 0.07, 0.04, 0.01, 1.00 );
    float4 gloss_d = ( 0.07, 0.04, 0.01, 1.00 );
    float4 gloss_s = ( 1.00, 0.90, 0.50, 1.00 );
    float4 gloss_e = ( 0.00, 0.00, 0.00, 1.00 );
    float gloss_sh = 25;

    float4 Cbase = lightmodel(base_s, base_d, base_a, base_e, base_sh);
    float4 Cgloss = lightmodel(gloss_s, gloss_d, gloss_a, gloss_e,
gloss_sh);

    float4 uv_gloss = (invertscale(.335,.335,1)) * uv;
```

```
    float4 col = texture(gloss, uv gloss);
    float4 invcol = { 1, 1, 1, 0 } - col;

    return Cgloss*invcol + Cbase*col;
}
```

接下来的例子（图4-27，彩页中也有）显示了卡通渲染的一个简单形式（卡通渲染的更高级形式参见第5章）。它使用了光线方向和曲面法线的点积（**N·L**）来对一个只含两种不同颜色的维纹理图进行索引。

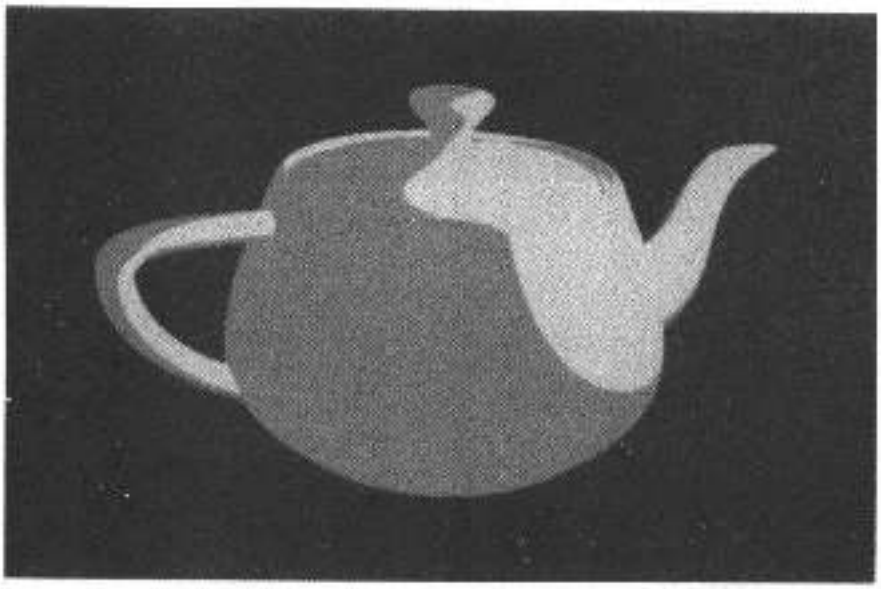


图4-27 卡通渲染，使用从斯坦福大学实时可编程着色项目处得到的一个图色系统生成（www.graphics.stanford.edu）

```
surface float4
lightmodel_cartoon (texref cartoon, float4 a, float4 d)
{
  perlight float fr = max(dot(N,L),0);
  // clamp upper and to avoid texture border color
  float4 uv = { min(integrate(fr) - 0.2, 0.75), 0, 0, 1 };
  return a * Ca + d * texture(cartoon, uv);
}

surface shader float4
cartoontest (texref cartoon)
{
  return lightmodel_cartoon(cartoon, {.4, .4, .8, 1}, {.4, .4, .8,
1});
}
```

更改图4-28（彩页中也有）中的阈值可以改变明/暗面积的比例。

图4-29（彩页中也有）是一个标准的凹凸贴图的示例。它综合所有照射到它上面的光线，并且基于（**T**，**L**）、（**B**，**L**）和（**N**，**L**）的点积计算出凸起的颜色。

在图4-30（彩页中也有）中，通过对**T·L**和**B·L**取相反数颠倒了凹凸的效果。这就产生了茶壶上面小洞的效果。

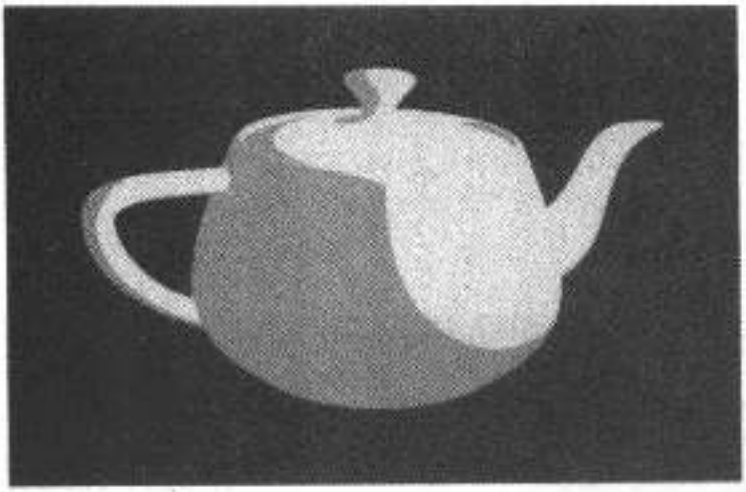


图 4-28　卡通渲染的一个变体，使用从斯坦福大学实时可编程着色项目处得到的一个日色系统生成（www.graphics.stanford.edu）

```
surface float4
lightmodel_cartoon (texref cartoon, float4 a, float4 d)
{
  perlight float ft = max(dot(N,L),0);
  // clamp upper end to avoid texture border color
  float4 uv = { min(integrate(ft+0.4, 0.75), 1, 0, 1 );
  return a * Ca + d * (texture(cartoon, uv);
}

surface shader float4
cartoontest (texref cartoon)
{
  return lightmodel_cartoon(cartoon, (.4, .4, .9, 1), (.4, .4, .8,
1));
}
```



图 4-29　凹凸贴图，使用从斯坦福大学实时可编程着色项目处得到的一个日色系统生成（www.graphics.stanford.edu）

```
surface shader float4
bumpdiffreal (texref bumps, float4 uv)
{
  perlight float3 L = { dot(T,L), dot(B,L), dot(N,L) };
  return integrate(C) * bumpdiff(bumps, uv, 0.0);
}
```

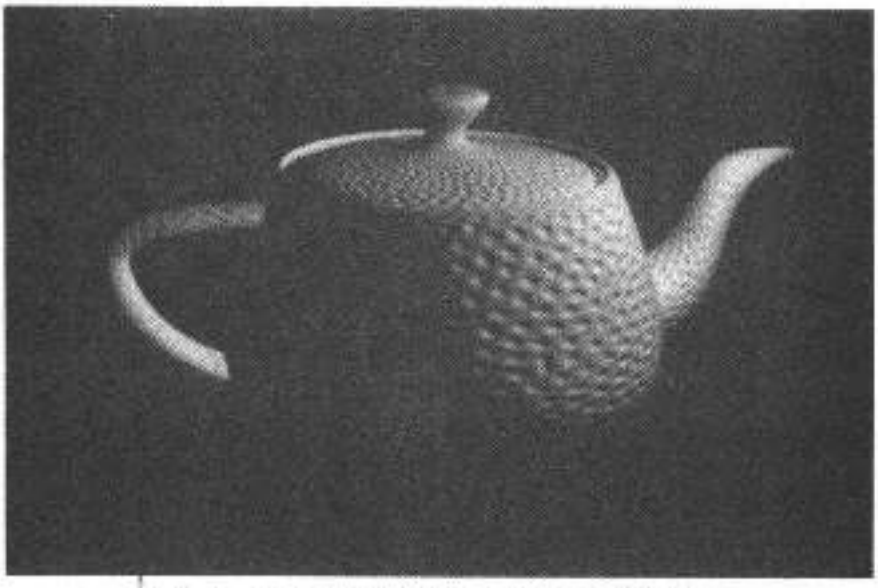


图 4-30 颠倒图 4-29 中的凹凸纹理。使用从斯坦福大学实时可编程着色项目处得到的一个出色系统生成（www.graphics.stanford.edu）

```
surface shader float4
bumpdiffreal (texref bumps, float4 uv)
{
  perlight float3 L = { dot(T,L), dot(B,L), dot(N,L) } * (-1,-1,1);
  return integrate(C) * bumpdiff(bumps, uv, 0.0);
}
```

第5章　实时渲染：实践

这一章将介绍如何为GPU编程，内容分如下两个部分：

1）为固定功能GPU编程。初看上去这好像是个矛盾，即在固定功能GPU上，能通过在每一步中改变渲染状态和纹理的多步渲染以及多纹理的方法来实现可编程的像素处理。尽管随着硬件的发展，这种方法将最终消亡，但它目前仍然非常流行，而且是令人惊奇地有效。

2）为可编程GPU编写代码。遵循一定的限制规则，程序员可以在渲染流水线中使用顶点和像素处理中所提供的功能。

以上两种编程方法很可能将在未来消亡。前者会随着固定功能GPU被可编程GPU替代而消失，并且如我们在前面章节中所讨论的，随着与硬件无关的着色语言的使用，后者也将被淘汰。即便如此，它们依然代表了当今的潮流。

我们假定读者在阅读本章时已经对OpenGL的渲染架构有所了解。[BLYT00]是一个非常好的相关背景知识的参考。

5.1　基本着色器

在固定功能GPU中，着色器（shader）指的是和某个物体或其部分所结合在一起的实体。它通过编辑器起作用，真正功能是让美工设计并调试各种不同的纹理贴图组合及动画。正如先前章节中所说的那样，术语“着色器”有更加一般的含义，但当前游戏业却只利用了这个狭义功能。其实着色器完全符合Cook的着色树定义（见4.8.1节），其相关操作仅限于纹理映射和混合。用本章介绍的方法实现后，着色器有效地提供了固定功能GPU的高级接口。

结合使用多步渲染和多纹理能够很容易地构建基本着色器。初级多步渲染方法会使用与材质对应的渲染状态来画出多边形。这个过程可能会重复多次，在每次调用时按照需要改变状态的设定。每一次这样的操作叫做一步着色，并且最终渲染是建立在帧缓冲区中的。而通过支持多纹理的硬件，我们能够把多步渲染合并为一步多纹理渲染。（尽管如此，使用支持多纹理的硬件并不能按其拥有的纹理单元的多少给速度带来相应乘数级的提高。这是因为我们仅仅把n个状态设定和n步渲染操作替换成了n个状态设定加上一步渲染操作，只节约了光栅化的时间。）

5.1.1　渲染状态

渲染状态是一个记录设定信息的结构，随后的渲染操作中将会用到这些设定。以下列出了一些常用的例子：

- 几何标记：线框/填充、平面明暗/平滑明暗、单/双面渲染、背面/正面筛选；
- 几何变换：缩放、平移、旋转和透视等；
- 材质标记：漫反射颜色、镜面反射颜色、透明度；
- 光照配置：光源数、位置、颜色、衰减；

- 纹理映射：纹理贴图的选取、纹理空间的变换；
- Z 缓冲选项：z 测试（<，≤，=，≥，>，总是，从不）、允许/禁止 Z 缓冲写入；
- 混合选项：以何种方式混合当前绘制操作和帧缓冲区中的已有信息（替换、叠加和减去等）。

在状态设定中有一个重要的实际/效率因素，那就是应该最小化状态变化的次数。我们还记得每个多边形可能有一个不同的着色器，每一个着色器可能有很多步的渲染。为了最小化变化的次数，要利用着色器对被渲染的面进行排序。接着设置第一个着色器的第一步渲染状态，并且绘制所有使用这个着色器的面。然后设置第一个着色器的第二步渲染状态，再绘制一遍所有使用这个着色器的面。仅在当前着色器的所有渲染步骤都完成后，我们才转到下一个着色器。这样的话，对每个渲染状态只设置了一次，没有任何重复。因此在每帧中我们都只做了最小限度的状态改变。

5.1.2　着色器排序

排序过程如下所述。当递归 BSP 树进行绘制时：对于每一个在视见约束体中（并且没有被 PVS 剔除）的叶节点，我们把它的面指针加入一个数组，这个数组保存着需要在当前帧中绘制的面。然后排序这个由面指针构成的数组，把所有共享同一个着色器的面放在一起。

绘制过程如下：

```
对于每组面
    设置着色器状态
    对于着色器的每一步
        设置着色器当前步状态
        绘制这个组中所有的面
        恢复着色步状态
    选着色器的下一步
    恢复着色器状态
下一组面
```

注意，设置着色器状态和设置着色器当前步状态是有区别的。一个着色器拥有一些适用于所有步渲染的全局设定。仅属于某一步的状态是由设置着色器当前步状态来设置的。

另一方面要注意，我们一遍又一遍地绘制相同的几何结构。这表明可以通过预编译的顶点数组来加快速度。所有的变换和裁剪只对被渲染面执行一次，结果存在 3D 图形卡的板载内存中，从而加速了随后对相同几何结构的绘制。

还有最后一个因素是光照贴图。所有共享同一个着色器的面并不一定共享相同的光照贴图。如果不把这个因素考虑在内，就会产生太多的光照贴图切换。因此我们把着色器和光照贴图纹理一起组合成排序关键字，来最小化光照贴图的切换。

下面显示了一个简单的着色器排序。对于每个面，循环检查所有其他面，把那些含相同着色器的面移到当前面的旁边：

```
for( i=0;i<nfaces;i++ )
  for( j=i+1;j<nfaces;j++ )
    if (sortkey[i]==sortkey[j])
    {
      swap(faces[j],faces[i+1])
                i++;
    }
```

我们可以用荷兰旗（Dutch flag）排序算法来加速这个过程：如果当前关键字和第一个关键字相同，把当前面移到最前；如果当前关键字和最后的关键字相同，把当前面移到最后。这个算法比原来的快两倍。在拥有相同关键字的面数量很大的情况下，这是个很好的排序算法。（实际情况中有很多面，但只有很少的不同关键字。）

为了扩展这个过程来同时处理着色器和光照贴图，我们要把面按照相同的着色器来分组，再在每个组内按照光照贴图来排序。这可以用两次排序来完成，如上所述，一次紧接着另一次。第一次排序只用着色器关键字，第二次使用着色器和光照贴图合成的关键字：

```
sortkey = (shader<<16) | lightmappic
```

sortkey 是一个 32 位整数（前 16 位记录着色器，后 16 位记录光照贴图）。

这个方法是优化的，因为第一次排序把有相同着色器的面分成组，然后第二次排序仅仅在此基础上把有相同光照贴图的面分组，不会影响按照着色器所分的组。

最终代码如下：

```
fd is the array of faces to be drawn
nfd is the number of face pointers in fd array

// sort faces by shader (sortkey&0xffff0000)
p1=0;
p2=nfd-1;
while(p1<p2)
{
   s1=fd[p1]->sortkey&0xffff0000;
   s2=fd[p2]->sortkey&0xffff0000;
   if (s1==s2)
      s2=-2;
   for( i=p1+1;i<p2;i++ )
      if (s1==(fd[i]->sortkey&0xffff0000))
      { // swap with begining
         f=fd[i];
         fd[i]=fd[++p1];
         fd[p1]=f;
      }
      else
      if (s2==(fd[i]->sortkey&0xffff0000))
      { // swap with end
         f=fd[i];
         fd[i--]=fd[--p2];
         fd[p2]=f;
      }
   p1++;
   if (s2!=-2)
      p2--;
}

// sort faces by composite key
// (sortkey=(shader<<16)|lightmappic)
p1=0;
p2=nfd-1;
while(p1<p2)
{
   s1=fd[p1]->sortkey;
   s2=fd[p2]->sortkey;
   if (s1==s2)
      s2=-2;
   for( i=p1+1;i<p2;i++ )
```

```
        if (s1==fd[i]->sortkey)
        { // swap with begining
          f=fd[i];
          fd[i]=fd[++p1];
          fd[p1]=f;
        }
        else
        if (s2==fd[i]->sortkey)
        { // swap with end
          f=fd[i];
          fd[i--]=fd[--p2];
          fd[p2]=f;
        }
      p1++;
      if (s2!=-2)
        p2--;
    }
```

5.1.3 着色器类的实现

一个简单的着色器类定义如下:

```
class FLY3D_API flyShaderPass
{
public:
   int flags;
   int tex;
   int blendsrc;
   int blenddst;

   int depthfunc;
   int alphafunc;
   float alphafuncref;

   int rgbgen;
   flyShaderFunc rgbgen_func;

   int tcmod;
   float tcmod_rotate;
   float tcmod_scale[2];
   float tcmod_scroll[2];
   flyShaderFunc tcmod_stretch_func;

   float anim_fps;
   int anim_numframes;
   int anim_frames[MAX_SHADER_ANIMFRAMES];

   void load(flyFile *fp,char *section,int i);
   void save (FILE *fp,int i);
   void set_state_1();
   void set_state_2();
};

class FLY3D_API flyShader
{
   int curpass;

public:
   int flags;
   int npass;
   flyShaderPass pass[MAX_SHADER_PASSES];

   int set_state(int cp);
   int restore_state();
   void load(flyFile *fp,char *section);
   void save(char *filename,int i);
};
```

5.2 渲染状态

我们现在详细深入地介绍渲染状态和其他能帮助实现着色器的选项。它们分为两个大类：在所有步渲染中都有作用的全局设定以及只对一步渲染有作用的局部设定。

5.2.1 全局设定

全局参数	描　述
无筛选	开启或关闭背面筛选（主要用于透明物体）
深度写入	开启或关闭 Z 缓冲写入（主要用于透明和特效物体）
碰撞	这是一个和游戏相关的参数（见下文）
透明	如果一个着色器设置了透明标记，它必须在不透明物体之后渲染
天空	用于天空盒和全景投影。没有东西会和标上“天空”的面碰撞。任何和天空相交的投掷物都会消失

包含在着色器内的游戏相关参数是非常有用的。例如，可以从熔岩或火焰着色器中得到损伤值。使用这种着色器渲染的物体会拥有造成损伤的能力。而当玩家走动时，我们可以随着步伐让地板发出相应的声音。

5.2.2 局部设定

局部参数	描　述
深度函数	根据 z 值来绘制像素的比较函数（<、≤、=、≥、>、总是、从不）
颜色	总是白色、网格中的顶点颜色或者一个生成颜色的函数（见另表）
混合	指定当前步渲染怎样和前步进行混合：替代（没有混合）、增加（1，1）、相乘（DestColor，0）、Alpha 透明度（SrcAlpha，1-SrcAlpha）
alpha 测试	根据 alpha 测试的结果来渲染像素，把当前像素的 alpha 值和一个参考值［0，1］来比较。使用以下的关系运算：<、≤、=、≥、>
光照贴图/纹理贴图	使用从 BSP 中得到的预定义光照贴图或者由用户选择一个纹理贴图
动画贴图	选择一些纹理和一个帧速率。纹理按照指定的帧速率切换（用于火焰、爆炸等）
纹理坐标修改器	见下面单独的表
环境映射	根据视点来修改纹理坐标（铬映射（chrome mapping））
纹理夹钳	是否平铺纹理

纹理坐标修改器以时间函数的形式来改变物体的纹理坐标。这样能有效地让纹理贴图产生动画效果。人们常用它来模拟移动的水面，比如小溪或从扰动中激起向外散开的涟漪。

纹理坐标修改器	描　述
缩放	纹理映射中 u，v 坐标的缩放因子（并不随时间变化）
旋转	旋转纹理贴图（度/秒）。旋转绕（u，v，w）空间的中心进行
卷动	根据时间变换纹理贴图的函数（单位/秒）。值 1 表示纹理每秒钟变换它的整个空间
拉伸	拉伸是一个根据时间来定义纹理缩放的函数（常用于脉冲辐射的效果）

以上定义的函数都是一元周期函数，常见的有正弦波、三角波和锯齿波。一个简单周期由如下公式中的 5 个参数所确定：

$$y = \text{位移} + \text{振幅} * \text{函数}((\text{时间} + \text{相位}) * \text{频率})$$

附录 5.1 中有相关的例子。

函数参数	描　述
类型	正弦波、方波、三角波、锯齿波或反锯齿波
位移	波形的平衡位。如果设为 0，波形将在正负值之间等幅摆动
振幅	函数振幅值的一半
相位	波形的超前或延后。值 0 到 1 对应了一个周期的波长。值 -0.25 会使一个正弦波变为余弦波。值 0.5 会使任何类型的波形翻转
频率	一个全周期的重复频率（赫兹）

用来生成周期函数的代码是很直观的，如下所示：

```
#define SHADER_FUNC_SIN              1
#define SHADER_FUNC_TRIANGLE         2
#define SHADER_FUNC_SQUARE           3
#define SHADER_FUNC_SAWTOOTH         4
#define SHADER_FUNC_INVERSESAWTOOTH  5
class FLY3D_API shader_func
{
  public:
    int type;
    float args[4]; /* offset, amplitude, phase, frequency */

  float eval();
  {
    float x, y;

    /* Evaluate a number of time based periodic functions */
    /* y = args[0] + args[1] * func( (time + arg[2]) * arg[3] ) */

    x = (flyengine->cur_time_float + args[2]) * args[3];
    x -= (int)x;

    switch (type)
    {
    case SHADER_FUNC_SIN:
      y = (float)sin(x * 6.28318530717958647692528676655 9f);
      break;

    case SHADER_FUNC_TRIANGLE:
      if (x < 0.5f)
        y = 4.0f * x - 1.0f;
      else
        y = -4.0f * x + 3.0f;
      break;

    case SHADER_FUNC_SQUARE:
      if (x < 0.5f)
        y = 1.0f;
      else
        y = -1.0f;
      break;

    case SHADER_FUNC_SAWTOOTH:
      y = x;
      break;

    case SHADER_FUNC_INVERSESAWTOOTH:
      y = 1.0f - x;
      break;
```

```
        }
        return y * args[1] + args[0];
    }
};
```

用于状态设定的代码不可避免地同引擎和 OpenGL 紧密相关。但仍有几点需要指出，尤其是对多纹理的处理支持。在第一步渲染中，我们设置全局参数并算出可用于多纹理的纹理单元个数，渲染完后再恢复先前的设定（正如我们在每步渲染之间所做的那样）。

先考虑不使用多纹理的渲染伪代码：

```
for each pass i
set the state for pass i
draw the object
restore the state for pass i
```

如果多纹理被支持的话，以上代码可以改为：

```
i = 0
while i < no of passes
      set the state for pass i
      draw the object
      restore the state for pass i
      increment i based on the number of texture units used
```

以下给出的代码是这两个过程的简化版本（光盘中有多纹理的完整代码）。set_ state 函数首先检查当前是否是第一步，如果是的话就设置全局状态。接着它初始化当前步中那些与纹理映射无关的设定（深度函数、混合以及颜色）。然后，如果当前步是光照贴图就直接返回，否则需设置纹理映射的参数。

```
int shader::set_state(int cp)
{
  curpass=cp;

  shader_pass *p=&pass[curpass];

  if (curpass==0)
  {
  if (flags & SHADER_NOCULL)
    glDisable(GL_CULL_FACE);
  else
    glEnable(GL_CULL_FACE);

  if (flags & SHADER_NODEPTHWRITE)
    glDepthMask(GL_FALSE);
  else
    glDepthMask(GL_TRUE);
  }

  p->set_state_1();

  if (p->flags & SHADER_LIGHTMAP)
    return 1;

  p->set_state_2();

  return 0;
}
int shader::restore_state()
{
```

```
  shader_pass *p=&pass[curpass];

  if (p->rgbgen == SHADER_GEN_VERTEX)
    glDisableClientState(GL_COLOR_ARRAY);

  if (p->flags & SHADER_LIGHTMAP)
    {
    if (curpass==0 && npass>1 &&
      flyengine->multitexture && ntextureunits>1)
      {
      tc->sel_unit(1);
      tc->sel_tex(-1);
      if (p[1].flags & SHADER_TCMOD)
        glPopMatrix();
      if (p[1].flags & SHADER_TCGEN_ENV)
        {
        glDisable(GL_TEXTURE_GEN_S);
        glDisable(GL_TEXTURE_GEN_T);
        }
      tc->sel_unit(0);
      return 2;
      }
    return 1;
    }
  if (p->flags & SHADER_TCMOD)
    glPopMatrix();
  if (p->flags & SHADER_TCGEN_ENV)
    {
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);
    }
  return 1;
}

void shader_pass::set_state_1()
{
  glDepthFunc(depthfunc);

  if (flags & SHADER_BLEND)
    {
    glEnable(GL_BLEND);
    glBlendFunc(blendsrc, blenddst);
    }
  else
    glDisable(GL_BLEND);

  if (rgbgen == SHADER_GEN_IDENTITY)
    glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
  else
  if (rgbgen == SHADER_GEN_WAVE)
    {
    float rgb = rgbgen_func.eval();
    glColor4f(rgb, rgb, rgb, 1.0f);
    }

  else
  if (rgbgen == SHADER_GEN_VERTEX)
    glEnableClientState(GL_COLOR_ARRAY);
  else
  if (rgbgen == SHADER_GEN_DEFAULT)
    glColor4fv(&flyengine->shadercolor.x);
}
```

```
void shader_pass::set_state_2()
{
   if (flags & SHADER_ALPHAFUNC)
      {
      glEnable(GL_ALPHA_TEST);
      glAlphaFunc(alphafunc, alphafuncref);
      }
   else
      glDisable(GL_ALPHA_TEST);

   if ((flags & SHADER_ANIMMAP) && anim_numframes>0)
      tc->sel_tex(anim_frames[(int)(anim_fps*flyengine-
>cur_time_float)%anim_numframes]);
   else tc->sel_tex(tex);

   if (flags & SHADER_TEXCLAMP)
      {
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
      }
   else
      {
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
      glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
      }

   if (flags & SHADER_TCGEN_ENV)
      {
      glEnable(GL_TEXTURE_GEN_S);
      glEnable(GL_TEXTURE_GEN_T);
      }

   if (flags & SHADER_TCMOD)
      {
      glPushMatrix();

      glTranslatef(0.5f, 0.5f, 0.0f);

      if (tcmod & SHADER_TCMOD_ROTATE)
        glRotatef(tcmod_rotate * flyengine->cur_time_float, 0.0f,
0.0f, 1.0f);

      if (tcmod & SHADER_TCMOD_SCALE)
        glScalef(tcmod_scale[0], tcmod_scale[1], 1.0f);

      if (tcmod & SHADER_TCMOD_STRETCH)
        {
        float y = tcmod_stretch_func.eval();
        glScalef(1.0f/y, 1.0f/y, 1.0f);
        }

      if (tcmod & SHADER_TCMOD_SCROLL)
        glTranslatef(
          tcmod_scroll[0] * flyengine->cur_time_float,
          tcmod_scroll[1] * flyengine->cur_time_float, 0.0);

      glTranslatef(-0.5f, -0.5f, 0.0f);
      }
}
```

5.3　着色器实例

在游戏设计中，人们广泛使用基本着色器来轻松实现很多特效。着色器可用于设计合成

纹理，这在结构上和4.8.1节中介绍的纹理树完全相同。它还能实现用来模拟水面流动和熔岩等的纹理坐标动画。下面将介绍一些例子来演示基本着色器的用途。

在游戏中，很大一部分基本着色器被用来实现常见的光照贴图/纹理贴图组合。以下的实例意在展示采用精心设计的合成算子和动画所构成的更复杂的效果。显然，构建具有原创性的全新着色器主要仰赖于美工的创新能力。

5.3.1 环境映射和铬映射效果——玻璃、金属和铬

所有效果使用球形环境映射的标准纹理坐标生成程序。所有例子都使用一块简单的铬纹理（见图5-1），它非常适用于金属和玻璃的材质。这块纹理本身含有随机、模糊和弯曲的线条。画面很暗是因为使用了特殊的混合模式（通常是叠加），以防止产生不必要的饱和。

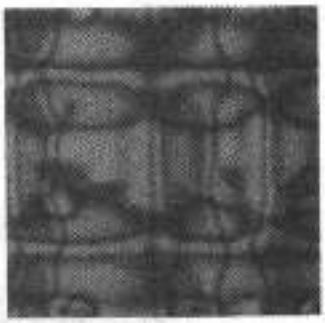


图5-1 铬纹理

可用一步渲染把以上纹理集进行环境映射来生成"干净"的金属表面（不需要特别的混合）。

玻璃的效果可以用相同的纹理来生成，只是得采用叠加混合模式（1，1）。

铬纹理需要两步渲染。第一步用未经混合的表面纹理，第二步用乘法/叠加混合模式(DestColor,1)混合铬纹理。

图5.2（彩页中出右）显示了使用这个方法渲染出的一个玻璃金属桌。通过变换视角观察，它给人以可信的真实玻璃感。在实时的走动观察中，以上静止图像固有的明显限制将不再存在。（游戏引擎的潜在非娱乐应用之一就是计算机辅助架构设计（Computer Aided Archit-

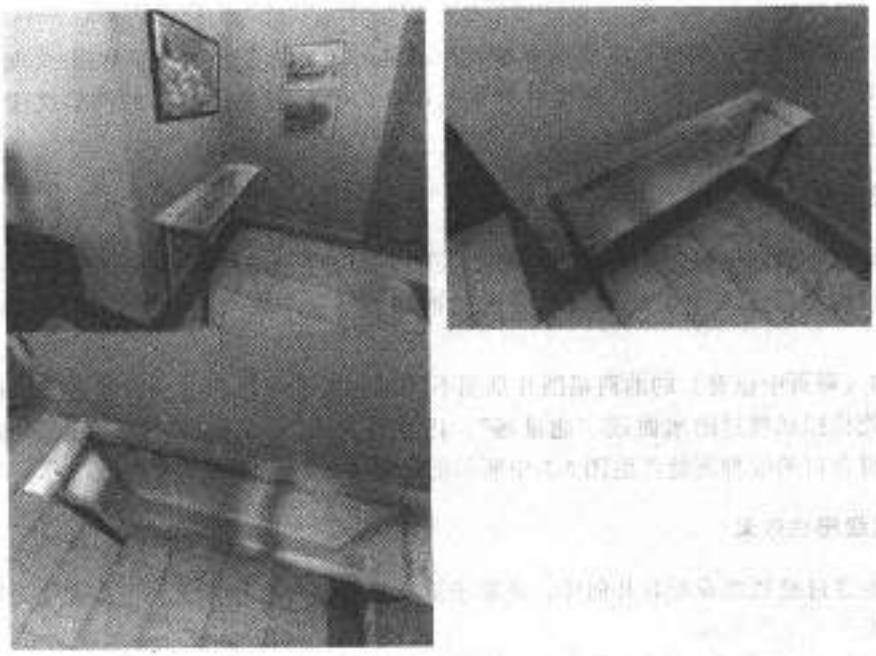


图5-2 使用铬纹理的玻璃金属桌

ectural Design，CAAD）。以前，游戏引擎的渲染质量还无法满足这样的应用。现在用着色器实现的高渲染质量意味着它可以胜任 CAAD 应用的要求。）

5.3.2　移动发光告示牌

墙上的移动告示牌效果能轻松地实现，如图 5-3 所示。

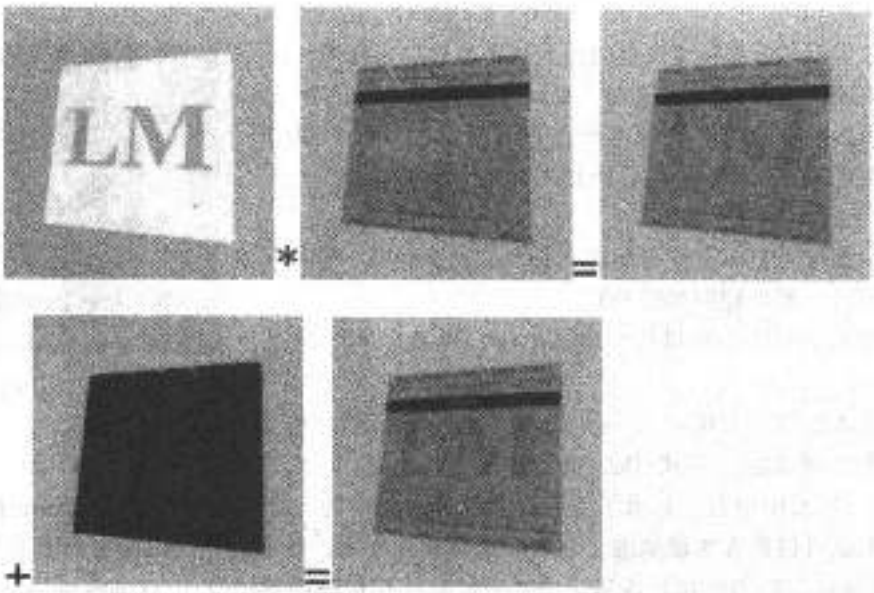


图 5-3　实现一个移动发光告示牌

第一个操作包括了两步：先使用乘法（DestColor，0）结合普通光照贴图/纹理贴图。再加上一个发光的告示牌，即一块在黑色背景上以 0.2 单元/秒速度滚动的简单纹理。这使用了叠加混合模式（1，1）。

5.3.3　简单栅栏效果

栅栏效果在游戏中很常见。它的原理就是用一块光照贴图加上一块纹理贴图来代表栅栏物体，通过沟槽显示出一块动画纹理或者其他部分场景。第一个例子使用较简单的动画纹理。

图 5-4（彩页中也有）的前两幅图片是被不同方向以不同速度卷动的纹理。它们叠加结合起来就能模拟风拂过的水面或“能量场”。再使用 alpha（SrcAlpha，1-SrcAlpha）结合已同光照贴图混合过的纹理就能产生图 5-5 中所示的最终效果。

5.3.4　高级栅栏效果

如果要透过栅栏观看游戏几何体，就要求更精细的组合操作。我们将从未经光照的栅栏纹理开始操作（见图 5-6a）。

设置 alpha 透明度为（SrcAlpha，1-SrcAlpha）的混合能使我们透过纹理观察。但是，这种

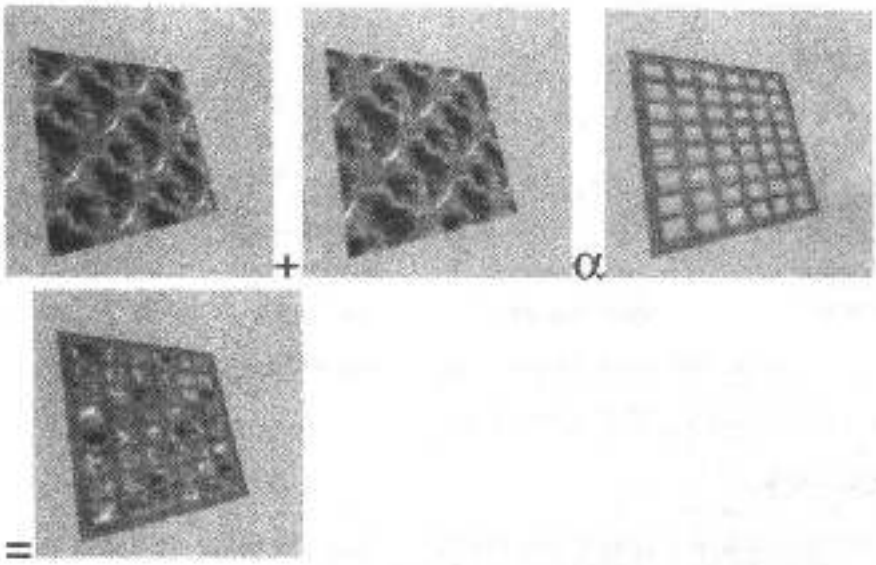


图 5-4　实现栅栏效果

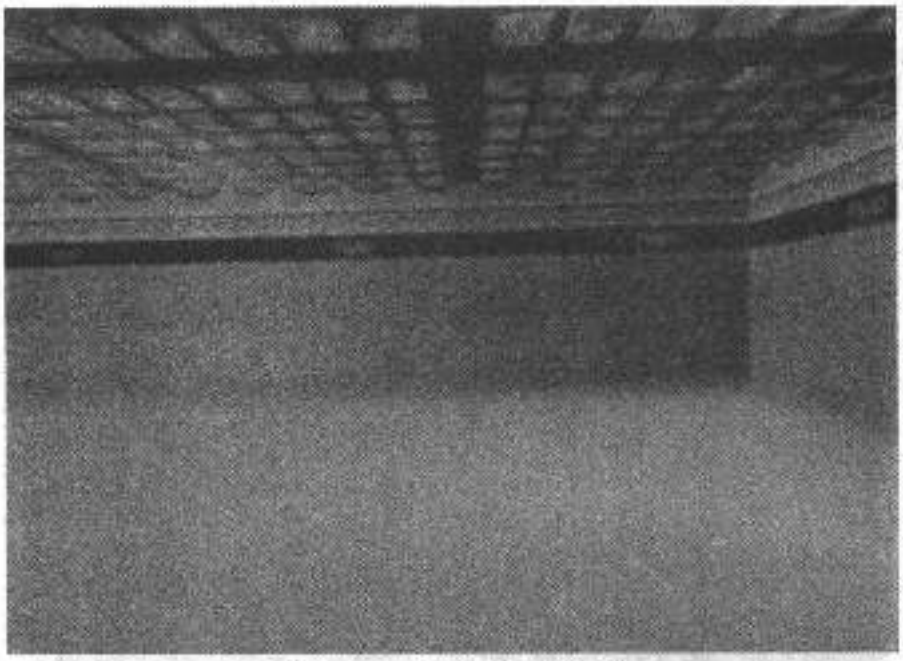
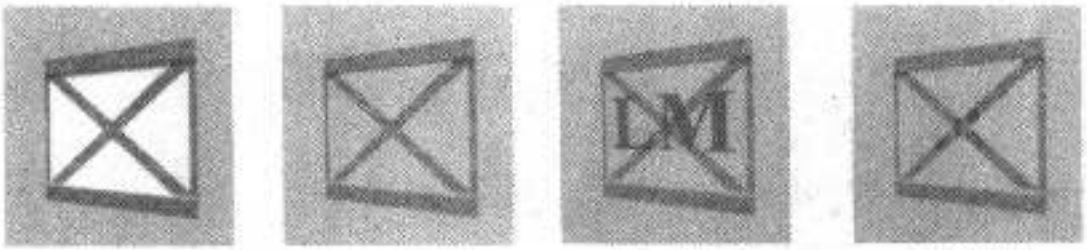


图 5-5　加入容积雾后的最终效果

方法并不能跳过全透明的点，相应的 Z 缓冲也会受到它们的“污染”。我们用 alpha 测试来解决这个问题，它在像素级上跳过全透明点，并保持这些点的 Z 缓冲值不发生改变。此外，还有一个问题就是纹理和光照贴图用乘法混合后，在光照贴图照亮栅栏的同时也会照亮透明像素。我们通过把 Z 缓冲测试设为相等来解决它。和上一步 alpha 测试中跳过的透明点一样，这里被跳过的点不会向 Z 缓冲写入内容，从而使相应的 Z 测试失败。



a) 一块未经充满的栅栏纹理　b) 在透明像素处的Z缓冲必须保持不变　c) 加入光照贴图并照亮透明像素　d) 设置Z缓冲侧就为相等

图 5-6　栅栏效果所需的操作。从这里能穿过栅栏看见游戏中的物体

图 5-7（彩页中也有）显示了栅栏的实例。

5.3.5　监视器效果

这个例子在渲染步数上是最复杂的（共5步）。所得的效果是一台显示白色噪声信号背景的监视器，在它的上面有一条滚动的红色水平线，最后还有文字加在上面。

我们从随机噪声纹理（用来模拟白色噪声信号）开始做起，让它快速地滚动。然后如图 5-8（彩页中也有）所示，用叠加混合（1，1）加上一条滚动的红色水平线。

接着往当前所得结果上加一个框架的 alpha 纹理（见图 5-9，彩页中也有），仅替换那些 alpha 值不为零的像素（SrcAlpha，1-SrcAlpha）。

随后用乘法混合（DestColor，0）加上光照贴图（见图 5-10，彩页中也有）。最后使用乘法/加法混合（DestColor，1），以球形环境映射的方式加上铬纹理（见图 5-11，彩页中也有）。

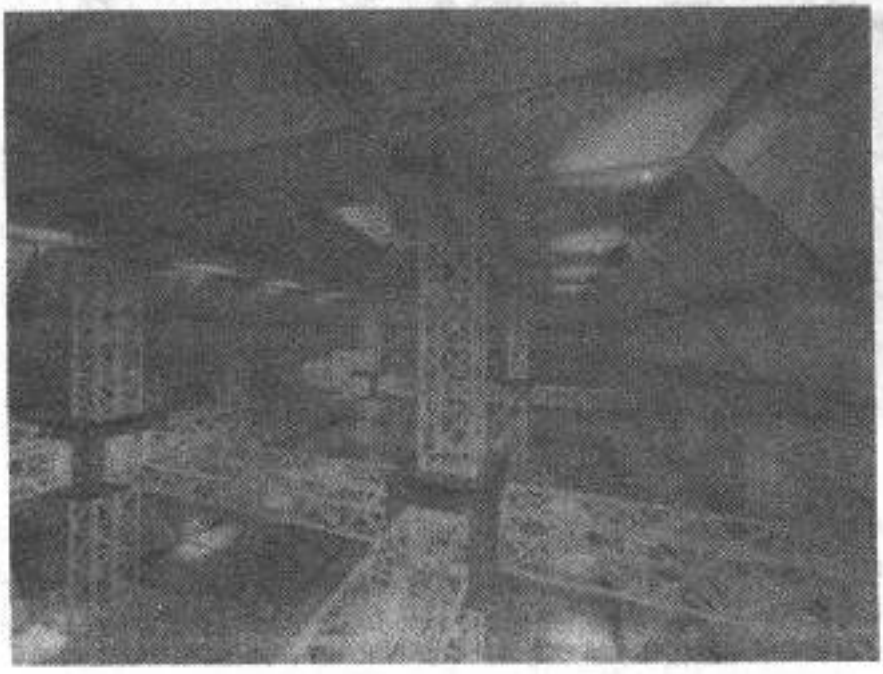


图 5-7　栅栏效果的实例

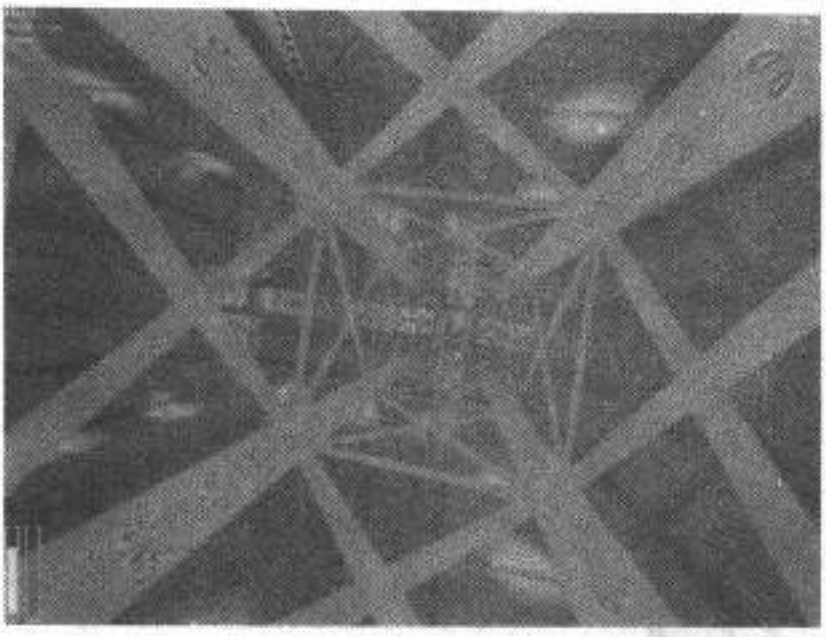


图5-7 （续）

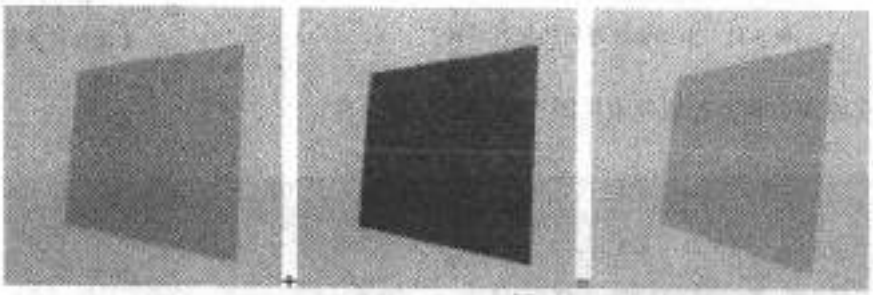


图5-8 随机或白色噪声纹理，其上有一条滚动的红线

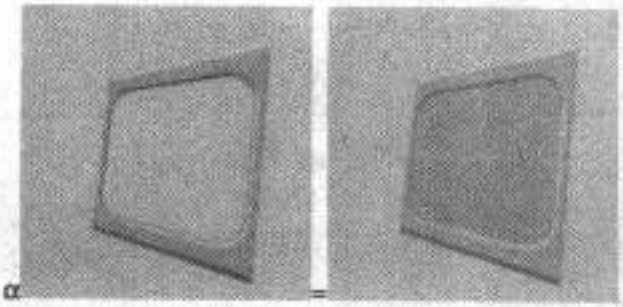


图5-9 加上框架纹理

这最后一步可以在添加alpha纹理之前进行，由此可以把铬纹理映射限制在监视器的玻璃屏幕上（见图5-12，彩页中也有）。

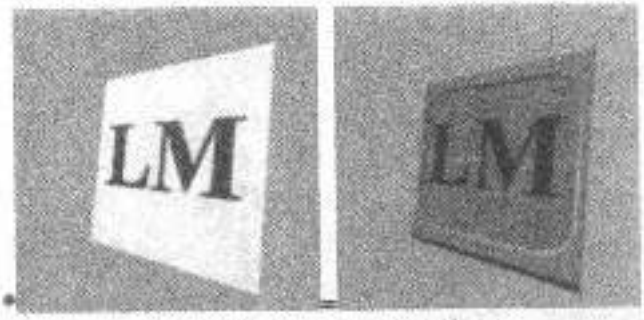


图 5-10　加上光照贴图（乘法混合）

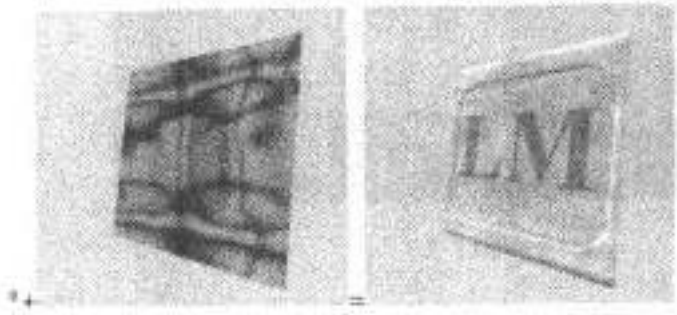


图 5-11　加上带纹理后的最终效果

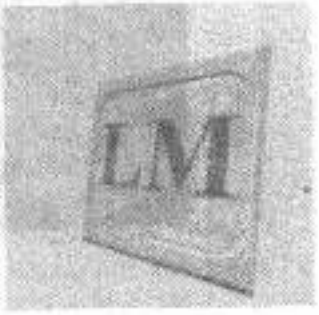


图 5-12　把铬纹理映射限制在玻璃屏幕上

最终在游戏中的监视器效果见图 5-13（彩页中也有）。

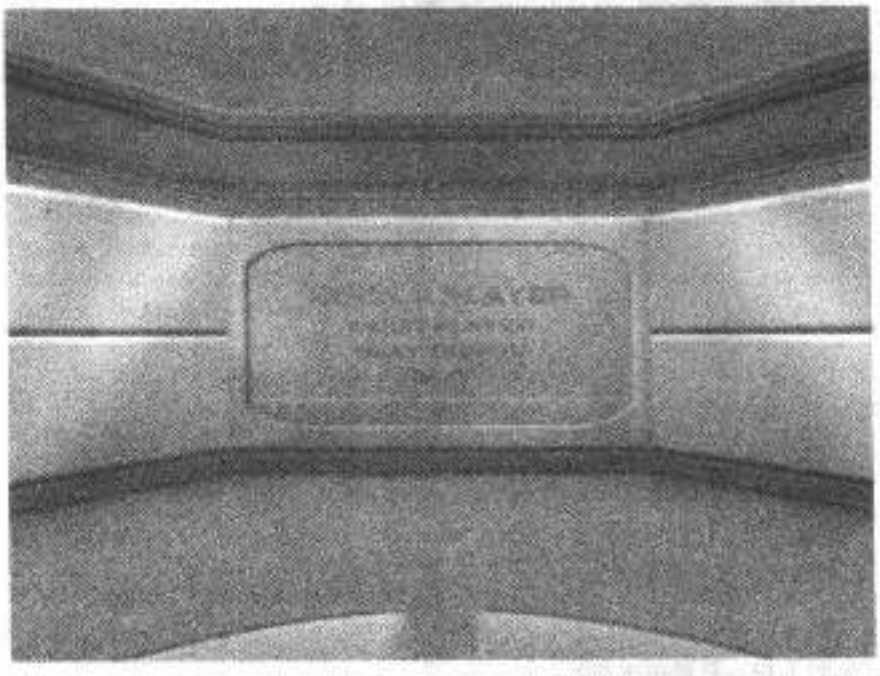


图 5-13　在一个游戏中的监视器效果

5.4　实时硬件渲染

在本章的第二部分，我们为基本的实时硬件渲染设计了编程方法。它们充分利用了 GPU 的可编程性。最近的硬件发展使得高级渲染技术从主机向图形硬件大规模转移。所以在撰写本书的时候（2002 年），实时渲染必须借助某种特定的图形硬件实现。这是个或多或少有些无奈的现实，特别在考虑到硬件演化如此迅速的时候。尽管最新的 GPU 功能十分强大和灵活，它们必须通过一个汇编语言的接口来编程。这个接口随生产商的不同而不同。但随着适用于各种不同 GPU 的高级着色语言和相关编译技术的发展（详见［PROU01］和第 4 章），我们还是看见了一丝曙光。基于前文所述的原因，我们选择了 NVIDIA 的 GeForce 图形硬件，并以他们定制的汇编语言来编程。撇开这个特殊性，通过阅读本章和包含在引擎中的例子，仍可以从中得出更具一般性的原理。

我们将讨论两个主要方面：

- 用于操作顶点流的顶点程序。
- 执行相同像素级操作（比如点积）的“固定”像素程序。

对 GeForce 程序模型的描述符合［PROU01］以及可从 www.nvidia.com 上获得的复杂文档。

当前的硬件已拥有（或即将拥有）实时渲染所有游戏几何结构——无论静态或动态物体——的能力，从而使以光照贴图形式实现的预计算光照失去意义。游戏引擎的理想目标就是统一静态和动态物体的光照。这能让两种物体的光照具有相同的质量和特性，并且所有光源能动态地改变位置、颜色和光照半径。而我们不用存储对应静态物体的光照贴图，只须使用相同的结构来存储法向量图。这个功能可以完美地实现例如凹凸映射和其他静态物体上的特效，从而带来渲染质量的新一轮提升。

5.4.1　顶点编程

遵循一定的限制（主要是常量存储上的），顶点编程把顶点看成独立实体，能在顶点上执行任何所需的操作。因此可以用它来实现高级光照和任何改变几何结构的操作，例如水面的扰动和蒙皮（skinning）。有一点仍需注意，用顶点程序进行光照，比如用来有效地实现动态物体上的 Phong 明暗时，网格必须要有相当高的分辨率（因为光照仅在顶点上计算）。但由于许多顶点程序是和固定逐像素程序（见下文）结合使用，所以这个要求就相应降低了。依靠着色器模型，顶点程序在游戏中既能用于静态物体也能用于动态物体。

顶点编程允许程序员使用 GPU 的功能。正如我们在本章的前面所学到的，这些功能包括了用设置操作模式来工作的固定功能单元。顶点处理硬件取代了当前硬件中的变换和光照单元，执行完全基于顶点的操作。没有任何拓扑信息被输入到这个单元中。由此，顶点程序必须负责本来由应用程序实现的在这个阶段的变换和其他特效。除此之外，其他图形流水线中的复杂操作仍然是固定的功能，它们包括显存存取、纹理管理和多边形管理等。当没有使用顶点编程时，可编程 GPU 回复到使用变换和光照单元的完全固定功能 GPU。我们可以在顶点程序和固定功能之间以及不同的顶点程序之间切换。如果只需要普通变换和光照，那么使用固定功能硬件会更加有效。这项功能还支持向后兼容。

顶点程序（vertex program）[⊖]（也称为顶点着色器）是一次只对一个顶点进行操作的汇编代码模块。程序操作遵循一般的汇编代码编程规范，使用一个寄存器——寄存器处理器，还包含作用于寄存器值的操作。使用两个寄存器的指令格式如下：

操作符　目标寄存器，源寄存器 1，源寄存器 2；

图 5-14 显示了顶点程序的程序模型。程序可以得到当前顶点的属性，并向输出属性寄存器写入。常量内存保存那些从 CPU 得到的数据，它们在一帧中保持不变。还有一个用于临时结果的寄存器文件。

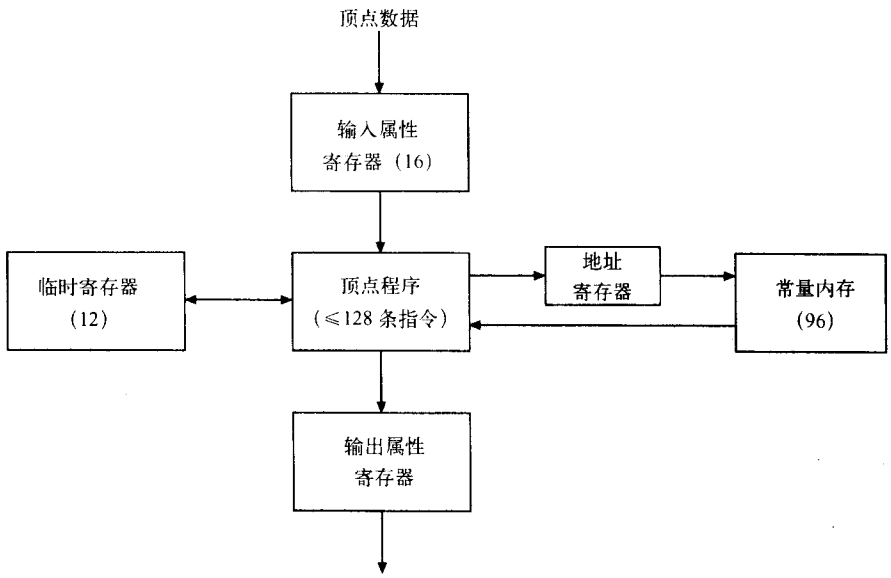


图 5-14　顶点程序模型（箭头表明程序的只读、只写或读写权限）

下面将概述怎样编写顶点程序以及怎样把它集成到游戏引擎中去。由于处理模式是寄存器-寄存器的，我们就先简要介绍一下寄存器组。

输入属性寄存器

一共有 16 个只读四重浮点寄存器用于保存一个顶点的数据或属性。例如：

位置（x，y，z，w）

纹理坐标（u，v，w，q）

颜色（r，g，b，a）

和顶点相关的分量会被保存在这些寄存器中。顶点程序对这些数据进行操作，把结果存到只写属性寄存器中。顶点程序对每个顶点执行一次，而无法生成新的顶点。它的结果只取决于被执行的指令、输入属性寄存器中的数据和常量内存中的数据。顶点程序不能产生能被顶点

⊖ 我们在全书中统一使用术语顶点程序，并为本章开头引入的模型和第 4 章中的用法保留着色器（shader）这个术语。在我们的系统中，着色器可以拥有一个顶点程序作为它多步渲染中的一步。而当术语着色器用在顶点程序上时，表明这个程序的功能仅仅是对顶点进行光照。但实际上顶点程序还可以对顶点进行几何操作。

流中下一个顶点持续使用的数据。除此以外，也不支持任何分支或循环指令。对于程序处理的每个顶点，顶点程序中的每条指令执行且仅执行一次。

常量内存

包含96个四重浮点数的常量内存同样是只读的。它的作用是保存会被程序使用的持续不变的数据，或只在每帧改变的数据。它们可以是用于Phong明暗处理的材质属性、光照方向向量等。尽管这块内存对顶点程序是只读的，但应用程序可以写入游戏中出现的那些只在每帧改变的数据。

临时寄存器

临时寄存器为顶点程序提供工作内存，它们是可读写的。一共有12个，同样是四重浮点数。

输出属性寄存器

输出属性寄存器用于存储顶点程序的结果，在大小上和输入属性寄存器吻合。在这以后，顶点会被GPU的剩余部分进一步处理。第一个输出属性寄存器保存顶点在裁剪空间的位置。每一个顶点程序都必须向这个寄存器写入数值。

地址寄存器

除此之外还有一个额外的只写临时寄存器——地址寄存器——用于存储一个比例因子。它被用于常量内存的间接寻址。

实践顶点编程

当前顶点编程的特点是不允许分支和循环的常规汇编代码编程。因为每条指令在一个时钟周期内执行，所以程序的性能直接和它所包含的指令数成正比。因此，通过确保没有冗余指令能对程序进行优化。而可以充分利用函数的向量性和源数据的“搅拌”（swizzle）函数（见下文）来达到这个目的。指令集的理论基础是基于在固定功能硬件中所使用的功能相同的指令和一份关于在固定硬件中使用功能的统计分析［LIND01］。

本节中给出的信息意在作为对顶点编程的非正式介绍，它们应该足以帮助读者理解相关实例。附录5.2给出了一份指令的详尽列表，附带有一张易于记忆的属性寄存器名列表等其他信息。

所有顶点程序必须遵守以下约束：

- 程序必须把顶点在裁剪空间中的位置写入o[HPOS]寄存器。
- 程序不能超过128条指令。
- 任何指令不能用一个以上的常量寄存器作为输入。
- 任何指令不能用一个以上的顶点属性寄存器作为输入。

最简单的顶点程序可能就是下列单条指令：

```
MOV o[HPOS], v[OPOS];
```

它把第一个输入寄存器的内容（对象的空间位置）拷贝到第一个输出寄存器（裁剪空间中的位置）。

指令能使用修改器修改它的输入和输出参数。输出寄存器的修改器是允许分量选择的写掩码，而输入寄存器的修改器是可改变正负的分量搅拌。搅拌指的是（x，y，z，w）的任何分量能在另一个分量的位置上出现或/和被重复。比如：

.xxxx 在所有位置重复x分量

.wzyx 颠倒分量的顺序

一共有 $4^4 = 256$ 种可能的排列，所有排列都是合法的。搅拌在叉积等计算中是很有效的。我们不久将会演示它。

指令集既包括简单算术操作，比如 MOV、MUL、ADD 和 MAD，也包括更复杂的指令，比如用于 Phong 明暗处理的 LIT。

LIT 是一个复杂的函数，以下伪代码给出了它的定义：

```
output.x = 1.0                    //将用来保存环境光照分量
output.y = max (N.L, 0.0)         //漫反射分量
output.z = 0.0                    //将用来保存镜面反射分量
if (N.L > 0.0and n = 0.0) then output.z = 1.0
else if (N.L > 0.0 and N.H > 0.0) then output.z = (N.H)^n
output.w = 1.0
```

LIT 假定源向量的 x 分量为 **N.L**，y 分量为 **N.H** 且 z 分量是镜面反射指数 n。执行 LIT 之后，分散的分量需要用 MAD 指令结合起来。

另一个有关光照的指令是用于构建衰减向量的 DST。用这条指令可以让一个向量通过点积来产生衰减系数，如下所示：

$$k_0 + k_1 * d + k_2 * d^* d = (k_0, k_1, k_2) \cdot (1, d, d^* d)$$

DST 以

$$(\text{n/a}, d^* d, d^* d, \text{n/a}) \text{和} (\text{n/a}, 1/d, 1/d, \text{n/a})$$

作为输入，并返回

$$(1, d, d^* d, 1/d)$$

可以用点积指令进行顶点变换。例如，可以用四维的点积指令来把一个顶点变换到裁剪空间：

```
DP4  o[HPOS].x,  c[0],  v[OPOS];
DP4  o[HPOS].y,  c[1],  v[OPOS];
DP4  o[HPOS].z,  c[2],  v[OPOS];
DP4  o[HPOS].w,  c[3],  v[OPOS];
```

这里假定模型视图矩阵和投影矩阵的乘积已经存储在c[0]到 c[3]中。

点积 R0 = R1 × R2 能用 MUL 和 MAD 来实现：

```
MUL R0,  R1.zxyw,  R2.yzxw;
MAD R0,  R1.yzxw,  R2.zxyw,  -R0;
```

这里 R0、R1 和 R2 都是临时寄存器。

以下代码执行常用的向量归一化。它用于 R0 中的向量：

```
DP3 R0.w,    R0,    R0;
RSQ R0.w,    R0.w;
MUL R0:xyz,  R0,    R0.w;
```

第一条指令 DP3 执行一次三维的点积。第二条 RSQ 是平方根。第三条执行归一化。

为了保持总体硬件设计的简洁，分支和循环是不被允许的，正如我们先前提到的那样。然而，if-then-else 语句的效果是可以实现的。考虑下面的语句：

if < 条件 > **then** x **else** y

它能够用这样的方法来实现：

计算 x

计算 y

使用 SLT 或 SGE 来生成一个条件结果（0 或 1）

x = x*条件结果

y = y*条件结果

顶点程序中的另一个常用需求就是 sine/cosine 函数。在游戏中常用它们来扰动水面。这里我们认为按一组谐振的合成波来确定 z 轴上位移是对风吹拂水面的一个很好近似。

sine 和 cosine 能用四项的泰勒展开式有效地计算出来。

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!}$$

$$\cos(x) = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!}$$

其中$(-\pi \leqslant x \leqslant \pi)$。

相应的算法是：

1) 设置$\left(1, -\frac{1}{3!}, \frac{1}{5!} - \frac{1}{7!}\right)$

2) 设置$\left(1, -\frac{1}{2!}, \frac{1}{4!} - \frac{1}{6!}\right)$

3) 把 x 从$(0 \leqslant x \leqslant 2\pi)$转换到$(-\pi \leqslant x \leqslant \pi)$

4) 生成向量(x, x^3, x^5, x^7)

5) 生成向量(x, x^2, x^4, x^6)

6) 计算点积$\sin(x) = \left(1, -\frac{1}{3!}, \frac{1}{5!} - \frac{1}{7!}\right) \cdot (x, x^3, x^5, x^7)$

7) 计算点积$\cos(x) = \left(1, -\frac{1}{2!}, \frac{1}{4!} - \frac{1}{6!}\right) \cdot (x, x^2, x^4, x^6)$

使用相应常量寄存器设置（见下文）的代码如下：

```
# scalar r0.x = cos(r1.x), r0.y = sin(r1.x)
MAD R0.x, R1.x, c[21].w, c[21].y;    # bring argument into
                                     # -pi, .., +pi range
EXP R0.y,  R0.x;
MAD R0.x,  R0.y, c[21].z, -c[21].x;
DST R2.xy, R0.x, R0.x;               # generate
                                     # 1, (r0.x)^2, .. (r0.x)^6
MUL R2.z,  R2.y, R2.y;
MUL R2.w,  R2.y, R2.z;
MUL R0,    R2,   R0.x;               # generate
                                     # r0.x, (r0.x)^3, .., (r0.x)^7
DP4 R0.y,  R0,   c[23];              # compute sin(r0.x)
```

光照的基本顶点程序

以下是对一个点光源执行标准变换和光照的完整顶点程序。记得顶点程序替换变换和光照硬件，因而它自己必须负责这个功能。一个顶点程序是夹在

!! VP1.0 和 END

之间的。我们将在下一节中进行详细介绍。

第一组指令把顶点转换到眼睛空间。光照在此进行。这需要保证模型视图矩阵已经被调入常量寄存器c[4]到 c[7]中。

为了把参数载入常量内存，用 OpenGL 的追踪功能是很方便的。应该充分利用 OpenGL 处理矩阵和其逆矩阵的功能。这里可以用

```
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_MODELVIEW,
  GL_INDENTITY_NV);
```

来把模型视图矩阵调入c[4]至c[7]中去。

接下去的一组指令为点光源计算光照单位向量。随后计算在眼睛空间内的顶点法向量。我们假定模型视图矩阵的逆矩阵的转置矩阵通过

```
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_MODELVIEW,
  GL_INVERSE_TRANSPOSE_NV);
```

被载入c[8]到c[11]中。然后才可以计算点积 **N**.**L**，以及该结果随后与漫反射或材质系数的乘法。

```
!!VP1.0
# standard transform diffuse lighting

# compute eye space vertex position
DP4 R0.x, c[4], v[OPOS];
DP4 R0.y, c[5], v[OPOS];
DP4 R0.z, c[6], v[OPOS];

# computes normalised light direction
ADD R0.xyz, c[30], -R0;
DP3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0, R0.w;

# computes normal in eye space
DP4 R1.x, c[8], v[NRML];
DP4 R1.y, c[9], v[NRML];
DP4 R1.z, c[10], v[NRML];

# compute N dot L and crop [0-1]
DP3 R2, R0, R1;
MAX R2, R2, c[20].x;

# set output colour to product light_colour*(N dot L)
MUL o[COL0], R2, c[31];

# set output texture co-ordinate
MOV o[TEX0], v[TEX0];

# set output vertex position in clip space
DP4 o[HPOS].x, c[0], v[OPOS];
DP4 o[HPOS].y, c[1], v[OPOS];
DP4 o[HPOS].z, c[2], v[OPOS];
DP4 o[HPOS].w, c[3], v[OPOS];

END
```

下一个顶点程序用 LIT 操作符完全实现了 Phong 明暗处理。它遵循标准近似，即假定在光照计算中，光源和视点都被放置在无穷远处，从而使得 Blinn 中途向量（halfway vector）**H** 在每帧中是一个常数。

```
!!VP1.0
# transform and Phong lighting

# compute eye space vertex position
DP4 R0.x, c[4], v[OPOS];
DP4 R0.y, c[5], v[OPOS];
```

```
DP4 R0.z, c[6], v[OPOS];

# computes normalised light direction
ADD R0.xyz, c[30], -R0;
DP3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0, R0.w;

# computes normal in eye space
DP4 R1.x, c[8], v[NRML];
DP4 R1.y, c[9], v[NRML];
DP4 R1.z, c[10], v[NRML];

# compute N dot L
DP3 R2.x, R0, R1;
DP3 R2.y, c[34], R1;
MOV R2.w, c[34].w;
LIT R3, R2;

# set output colour to ambient + diffuse*(L.N) + (H.N)^n
MAD R4, R3.y, v[COL0], R3.z;
ADD o[COL0], R4, c[35];

# set output texture co-ordinate
MOV o[TEX0], v[TEX0];

# set output vertex position in clip space
DP4 o[HPOS].x, c[0], v[OPOS];
DP4 o[HPOS].y, c[1], v[OPOS];
DP4 o[HPOS].z, c[2], v[OPOS];
DP4 o[HPOS].w, c[3], v[OPOS];

END
```

在多步着色器中集成顶点程序

顶点程序是通过着色器结构集成入引擎的。这是从控制固定功能硬件的程序扩展而来的。目前着色器能存取或拥有一个顶点程序。由此我们仍然能使用多步着色器的结构，只不过现在一步中可能含有一个顶点程序。

Fly3D着色器编辑器允许为着色器的每一步指定一个顶点程序文本文件（.vp)。当载入着色器时，相应的顶点程序文件被载入并编译。每步的顶点程序还有一个常规定义的着色器函数。在运行时，当选择着色器进行绘制时，这个函数被计算并且通过常量寄存器把结果送入顶点程序。

下列代码加载并编译一个顶点程序：

```
str=g_flyengine->flysdkdatapath+"programs\\"+programfile;
flyFile vpfile;
if (vpfile.open(str))
{
   char *buf=new char[vpfile.len+1];
   memcpy(buf,vpfile.buf,vpfile.len);
   buf[vpfile.len]=0;

   glGenProgramsNV(1,&vpid);
   glBindProgramNV(GL_VERTEX_PROGRAM_NV,vpid);

   nvparse(buf);
```

```
    delete buf;
    vpfile.close();
}
```

首先，程序文件被调入一个字符串中。然后生成并绑定程序的标识。事实上编译是通过调用 nvparse 执行的。它使用 NVIDIA parser[⊖] 来编译顶点程序的汇编代码。

以下 set_ pass 中的代码选择并激活顶点程序：

```
if (vpid!=-1)
{
    glBindProgramNV(GL_VERTEX_PROGRAM_NV, vpid);
    glEnable(GL_VERTEX_PROGRAM_NV);
    float f=vpfunc.eval();
    glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 16,f,f,f,0);
}
```

在这个过程中还计算程序函数，并把结果存储在一个程序常量中。

常量程序参数设置

常量寄存器的设置从某种程度上讲是和具体应用程序相关的。比如在实现一个蒙皮操作的顶点程序中，会用寄存器来存储骨骼矩阵。但对于一般的应用程序来说，一种好的做法是把寄存器设置为大多数顶点程序都很可能会用到的信息。

下表显示了对于单光源应用的一个简单合理的通用设置。

常量寄存器	内　容
0 ~ 3	模型视图矩阵和投影矩阵相乘后的矩阵
4 ~ 7	模型视图矩阵
8 ~ 11	模型视图矩阵转置的逆矩阵
12 ~ 15	模型视图矩阵的逆矩阵
20	(0, 0.5, 1, 2) ——能被重复以及搅拌的通用常量
21	Sin/Cos 辅助常量
22	Cos 泰勒展开式的系数
23	Sin 泰勒展开式的系数
24	程序函数的参数
25	程序函数 (f, f, f, 0)
26	时间 (t, t, t, 0)
30	眼睛空间中的光源坐标
31	光照颜色
32	光照衰减
33	**L** 光照向量
34	**H** 中途向量
35	环境颜色

以下程序进行了相关设置：

```
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 0, GL_MODELVIEW_PROJECTION_NV,
GL_IDENTITY_NV);
```

⊖ NVParse 是一个用来简化在 NVIDIA GPU 上进行顶点和像素编程的 OpenGL 工具。它是一个可以和原有 OpenGL 函数联合使用的库，用来：
- 简化配置纹理着色器的过程；
- 简化配置寄存结合器的过程；
- （使用改进的错误报告）调入顶点程序。

除此之外，NVParse 还提供对 DirectX8.0 顶点着色器和像素着色器的有限支持。

```
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 4, GL_MODELVIEW,
GL_IDENTITY_NV);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 8, GL_MODELVIEW,
GL_INVERSE_TRANSPOSE_NV);
glTrackMatrixNV(GL_VERTEX_PROGRAM_NV, 12, GL_MODELVIEW,
GL_INVERSE_NV);

glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 20, 0, 0.5f, 1,2);
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 21, PI, 0.5f,
2.0f*PI,1.0f/(2.0f*PI));
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 22, 1.0f, -
0.5f,1.0f/24.0f,-1.0f/720.0f);
glProgramParameter4fNV( GL_VERTEX_PROGRAM_NV, 23, 1.0f, -
1.0f/6.0f,1.0f/120.0f, -1.0f/5040.0f);

float f=vpfunc.eval();
glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV,24,
  vpfunc.args[0],vpfunc.args[1],vpfunc.args[2],vpfunc.args[3]);
float f=shader->pass->vpfunc.eval();
glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 25,f,f,f,0);

glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV,26,
  cur_time_float,cur_time_float,cur_time_float,0);

glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV,
  30,lightpos[0].x,lightpos[1].y,lightpos[2],0);
glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV,31,
  lightcolor[0],lightcolor[1],lightcolor[2],0);
glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV,32,
  lightatten[0],lightatten[1],lightatten[2],0);

glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 33,L.x,L.y,L.z,0 );
H=(L+Z)*0.5f;
H.normalize();
glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV, 34,H.x,H.y,H.z,32 );

glProgramParameter4fNV(GL_VERTEX_PROGRAM_NV,
35,ambient.x,ambient.y,ambient.z,0 );
```

下面将演示超越了标准变换和光照功能的例子。首先从一个卡通渲染的顶点程序开始。可以很方便地把这个任务分成两个程序。然后，把它们实现为着色器中不同的两步渲染。第一步存取一个用于替代光照的一维查找表。变换等其他过程和先前一样。**N.L** 设置成用来查找一维纹理表的 u 纹理坐标。v 纹理坐标设为 0。这样就能用卡通风格的上色来替代漫反射光照计算。注意，该程序几乎和前一个程序完全相同，除了加入了把 **L.N** 设置为 u 纹理坐标的指令。

```
!!VP1.0
# cartoon lighting

# compute eye space vertex position
DP4 R0.x, c[4], v[OPOS];
DP4 R0.y, c[5], v[OPOS];
DP4 R0.z, c[6], v[OPOS];

# computes normalised light direction
ADD R0.xyz, c[30], -R0;
DP3 R0.w, R0, R0;
RSQ R0.w, R0.w;
MUL R0.xyz, R0, R0.w;
```

```
# computes normal in eye space
DP4 R1.x, c[8], v[NRML];
DP4 R1.y, c[9], v[NRML];
DP4 R1.z, c[10], v[NRML];
# compute N dot L and crop [0-1]
DP3 R2, R0, R1;
MAX R2, R2, c[20].x;

# set output u texture co-ordinate to (L dot N)
MOV o[TEX0].x, R2.x;

# set output v texture co-ordinate to 0
MOV o[TEX0].y, c[20].x;

# set output colour to input vertex colour
MOV o[COL0], v[COL0];

# set output vertx position in clip space
DP4 o[HPOS].x, c[0], v[OPOS];
DP4 o[HPOS].y, c[1], v[OPOS];
DP4 o[HPOS].z, c[2], v[OPOS];
DP4 o[HPOS].w, c[3], v[OPOS];

END
```

下一个顶点程序执行第二步卡通光照功能——对边界阴影的加深。以下的简单操作能够有效地做到它。首先，我们启用正面筛选，用黑色渲染物体。这个物体有一些“膨胀”——其实就是通过按照法向量的方向移动顶点而扩大了一点点。然后再按正常尺寸绘制物体，用前面设计的卡通光照，并允许背面筛选。本演示的一个特点就是边界宽度是动起来的。这是通过把着色器函数传到常量寄存器c[16]实现的，如“在多步着色器中集成顶点程序”小节中所述。并且这个演示还是顶点程序使用可变参数值的例子。

```
!!VP1.0
# cartoon silhouette edge enhancement

MOV R0,v[OPOS];
MOV R1,v[NRML];
MAD R0,R1,c[25],R0;

DP4 o[HPOS].x, c[0], R0;
DP4 o[HPOS].y, c[1], R0;
DP4 o[HPOS].z, c[2], R0;
DP4 o[HPOS].w, c[3], R0;

MOV o[COL0], c[20].x;
MOV o[TEX0], c[20].x;

END
```

图 5-15（彩页中也有）显示了用以上代码渲染出的物体。对于一个动态物体来说，这种风格化在物体和光源的交互中能有效地给予用户相应感觉。在这幅图像中，加深的多边形分辨率是明显可见的。这再一次表明在使用顶点程序渲染时，网格必须有相对合理的高分辨率。

在本例中还有一个额外的实际问题需要考虑，那就是硬件的纹理过滤。我们把纹理过滤用到所有不通过卡通渲染的游戏物体上。然而在不希望纹理过滤影响卡通渲染物体的硬边风格的同时，我们仍然想对有颜色变化的边进行反锯齿处理。保证一维纹理贴图的尺寸比较大（比如 256 纹素）并且混合位于区域间的几个纹素就能够做到（见图 5-16）。我们还需设置纹理夹钳为一维纹理来防止在第一个和最后一个纹素之间的纹理过滤。

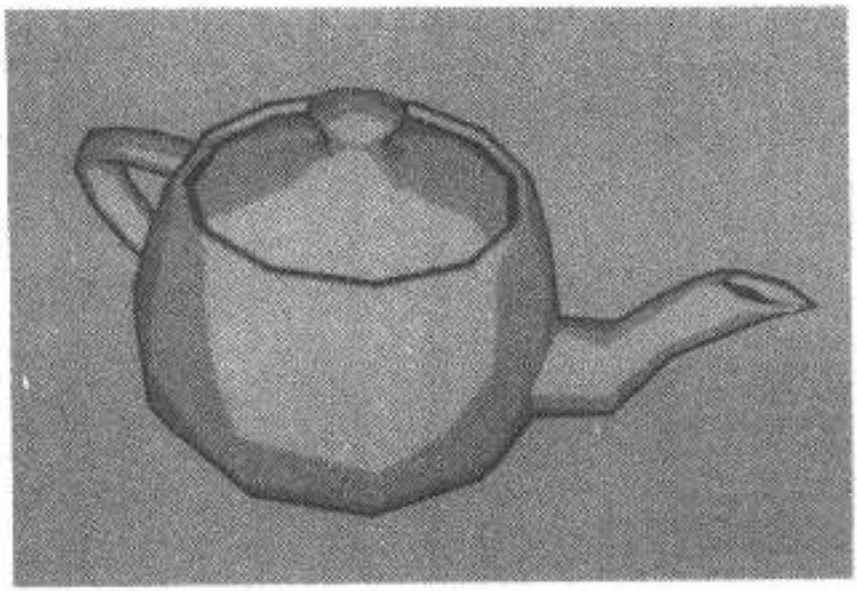


图5-15 三色卡通渲染的顶点程序。黑色由膨胀后的物体所绘出。两种蓝色的明暗代表了粗糙的N.L明暗

图5-16 在卡通渲染中使用的一块（256×1）纹理

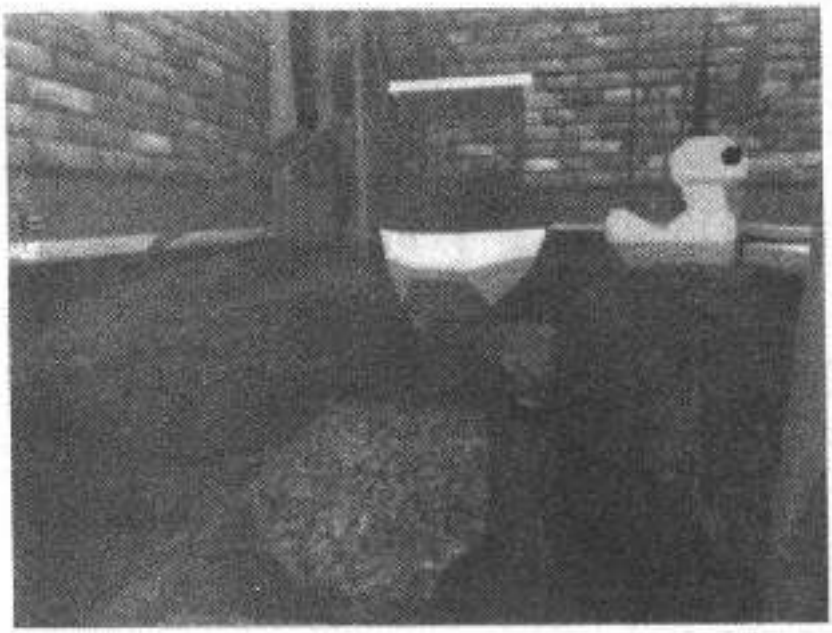


图5-17 水面依据波形顶点程序经轻通发的焙水池（从图像中爬了和脂下的水平面可以看出）。静止时图像难以表示，在动画时此效果有造出一个很好的气氛

顶点程序另一个常见的应用就是对表面的扰动，比如水面（见图 5-17，彩页中也有）。这里我们按照一个谐振函数对表面的 z 轴实行扰动。用以下简单公式来产生运动的波：

$$f(\text{pos}) = x + y\ \sin(z + w(\text{time} + \text{pos}))$$

这里：

x 是位移

y 是 sin 波的振幅

z 是 sin 波的相位

w 是频率

按以下计算：

```
Vpos.z += f(Vpos.x).f(Vpos.y)
```

用两列波来调节表面的高度。

```
!!VP1.0

MOV R1, v[OPOS];
ADD R1, R1, c[26].x;                    # add time
MAD R1, R1, c[24].w, c[24].z;           # multiply rate and add phase

# scalar r0.x = cos(r1.x), r0.y = sin(r1.x)
MAD R0.x, R1.x, c[21].w, c[21].y;       # bring argument into
                                        # -pi, .., +pi range
EXP R0.y, R0.x;
MAD R0.x, R0.y, c[21].z, -c[21].x;
DST R2.xy, R0.x, R0.x;                  # generate
                                        # 1, (r0.x)^2, .. (r0.x)^6
MUL R2.z, R2.y, R2.y;
MUL R2.w, R2.y, R2.z;
MUL R0, R2, R0.x;                       # generate
                                        # r0.x, (r0.x)^3, .., (r0.x)^7
DP4 R0.y, R0, c[23];                    # compute sin(r0.x)
DP4 R0.x, R2, c[22];                    # compute cos(r0.x)
# apply amplitude and offset
MAD R4.x, R0.x, c[24].y, c[24].x;

# scalar r0.x = cos(r1.y), r0.y = sin(r1.y)
MAD R0.x, R1.y, c[21].w, c[21].y;       # bring argument
                                        #into -pi, .., +pi range
EXP R0.y, R0.x;
MAD R0.x, R0.y, c[21].z, -c[21].x;
DST R2.xy, R0.x, R0.x;                  # generate
                                        #1, (r0.x)^2, .. (r0.x)^6
MUL R2.z, R2.y, R2.y;
MUL R2.w, R2.y, R2.z;
MUL R0, R2, R0.x;                       # generate
                                        # r0.x, (r0.x)^3, .., (r0.x)^7
DP4 R0.y, R0, c[23];                    # compute sin(r0.x)
DP4 R0.x, R2, c[22];                    # compute cos(r0.x)

# apply amplitude and offset
MAD R4.y, R0.x, c[24].y, c[24].x;

# multiply cos(x)*cos(y)
MUL R0.x, R4.x, R4.y;

# move R1 r0.x along normal direction
MOV R1, v[OPOS];
MAD R1.xyz, v[NRML], R0.x, R1;
```

```
# set output registers
DP4 o[HPOS].x, R1, c[0];
DP4 o[HPOS].y, R1, c[1];
DP4 o[HPOS].z, R1, c[2];
DP4 o[HPOS].w, R1, c[3];
MOV o[COL0], v[COL0];
MOV o[TEX0], v[TEX0];

END
```

5.4.2 像素编程

像素编程（也称为像素着色器）是一个容易引起误解的名字。目前没有消费级硬件完全支持像素编程，即能把每个像素看作一个独立的实体（和顶点编程类似）。虽然如此，还是存在能进行有限像素级操作的设备。例如，我们可用点积作为结合器来“结合”两块纹理，对每一块执行单独的计算，而对所有像素执行同样的生成函数。这个方法也称为纹理混合。

当前的像素编程指的是类似图 5-18 所示的架构。图中显示了 4 个纹理着色器，它们为 8 个寄存结合器提供输入。纹理着色器架构增强了普通的纹理查找，而且如我们将要看到的，它允许诸如独立寻址之类的操作。在这种寻址中，一个纹理查找的结果能用来寻址下一块纹理。

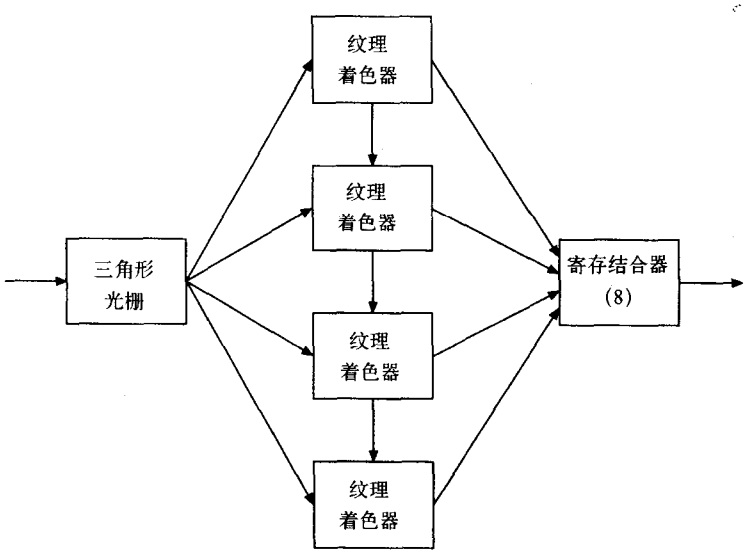


图 5-18　NVIDIA 的像素编程架构

5.4.3 使用寄存结合器的像素编程

寄存结合器以某种方式结合位于输入寄存器中的 RGBα 向量以及纹理。NVIDIA 架构的功能总结在以下列表中（附录 5.3 给出了完整的描述）。

- 4 个输入寄存器，称为 A、B、C 和 D，都能按以下方式结合：

  A < op > B

  C < op > D

AB + CD

这里 < op > 可以是：

点积　A·B

乘法　AB

遵循在附录 5.3 中介绍的有关结合的限制条件

除了上述操作，还可用以下 mux 操作：

$$\text{mux}(AB, CD) = (\text{Spare0}[\text{alpha}] \geqslant 1/2)\ ?\ AB : CD$$

输入寄存器里包含可用的纹理、主要（漫反射）和次要（镜面反射）颜色、两个常量颜色和两个空闲寄存器。

- 计算在 [- 1, 1] 的范围内进行。
- RGB 和 alpha 分量是分别处理的。
- 最终结合器阶段对每个像素把结果寄存器中的值合并为一个 RGB 颜色和 alpha 值。
- 当前硬件（GeForce3）支持 8 个通用结合器和一个最终结合器。

寄存结合器实验

本小节，我们使用 nvparse 来进行寄存结合器的编程实验。nvparse 提供一个使用高级语言赋值语句的用户编程模型。它只是完全掌控寄存结合器所需全部信息的一个子集。我们的想法是给出涉及该编程模型的一般性规律。

图 5-19 显示了一个硬件数据流程图。它意味着通过寄存结合器程序的信息按次序使用通用寄存器，把结果输出到寄存器组，并最终存到结果寄存器中。nvparse 支持的助记符在表 5-1 中给出。

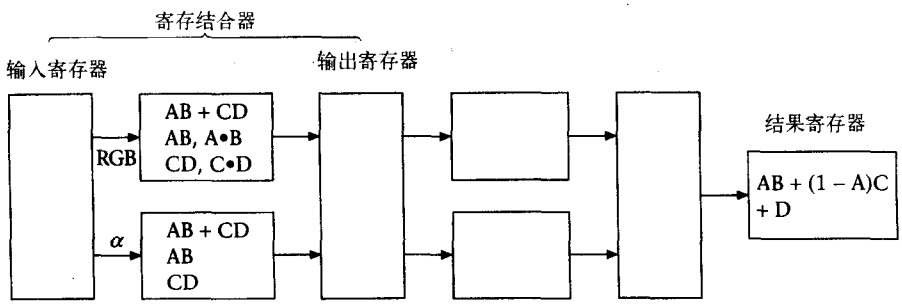


图 5-19　数据流和寄存结合器

表 5-1　nvparse 支持的寄存器助记符

寄　存　器	名　字	可　读	可　写
漫反射颜色	col0	是	是
镜面反射颜色	col1	是	是
纹理 0 颜色	tex0	是	是
纹理 1 颜色	tex1	是	是
纹理 2 颜色	tex2	是	是
纹理 3 颜色	tex3	是	是
空闲 0	spare0	是	是

(续)

寄存器	名字	可读	可写
空闲1	spare1	是	是
常量颜色0	const0	是	否
常量颜色1	const1	是	否
雾颜色和因子	fog	仅RGB	否
零	zero	是	否
忽略	discard	否	是

每个即将使用的结合器会被预设为一个特定状态，它是三个函数输出的某个模式：

A < op > B

C < op > D

AB + CD

使用点积的RGB函数必须忽略第三个结果，因而寄存器RGB函数只能有如下模式：

点积-点积-忽略

这个模式计算A·B和C·D，并使用nvparse助记符进行如下设置：

spare0 = expand(col0)·expand(tex0)

spare1 = expand(col1)·expand(tex1)

点积-乘法-忽略

这个模式计算A·B和CD，并使用nvparse助记符进行如下设置：

spare0 = expand(col0)·expand(tex0)

spare1 = col1*tex1

乘法-点积-忽略

这个模式计算AB和C·D，并使用nvparse助记符进行如下设置：

spare0 = col0*tex0

spare1 = expand(col1)·expand(tex1)

乘法-乘法-mux

这个模式计算AB、CD和mux(AB,CD)，并使用nvparse助记符进行如下设置：

discard = col0*tex0

discard = col1*tex1

spare1 = mux()

乘法-乘法-求和

这个模式计算AB、CD和(AB + CD)，并使用nvparse助记符进行如下设置：

discard = col0*tex0

discard = col1*tex1

spare1 = sum()

用代码实现最终结合器RGB函数计算是比较难的：

$$A*B + (1 - A)*C + D$$

对任意给定的寄存器A、B、C和D。在nvparse中，能用很多方法写RGB结果等式。

简单赋值：

```
out.rgb=tex0;
```

相乘：

```
out.rgb=tex0*final_product;
```

求和：

```
out.rgb=tex0+final_product;
```

线性插值(A*B+(1-A)*C)：

```
out.rgb=lerp(fog.a,const0,color_sum);
```

线性插值并求和：

```
out.rgb=lerp(fog.a,const0,color_sum)+const1;
```

寄存结合器实例：使用法向量图的逐像素漫反射明暗

本例使用了寄存结合器中的点积操作来实现动态物体的环境和漫反射明暗。这些动态物体是球体，由一个法向量图代表，（像粒子一样）在一个场景中弹跳。如图 5-20 所示（彩页中也有），颜色代表法向量的方向。球体被最近的光源照亮，当附近范围内没有光源时用环境光源代替。当球体足够靠近一个光源时，就引发了逐像素的漫反射明暗。

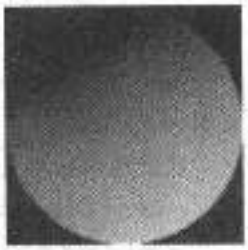


图 5-20　用来渲染球体的方形图——颜色代表法向量的方向

球体用一个方形告示牌来渲染，且没有旋转。按照定义，方形图必须一直和照相机/视见向量对齐，因此视见向量和方形平面对球体而言构成了一个局部坐标系。随着球体的移动，L 是相对于此坐标系计算（并归一化的）。渲染过程使用一个结合器在一步内完成。以下相当简洁的代码实现了所需的寄存结合器设置。

```
!!RC1.0
{
  rgb
  {
  spare0 = expand(col0) . expand(tex0);
  }
}
out.rgb = spare0 * const1 + const0;
out.a = tex0;
```

其具体细节是：

1）第一条语句设置第一阶段的 rgb 部分，扩展向量 L（以颜色传入）和 N（以纹理传入）并计算点积 L.N。

2）设置最终结合器的 rgb 部分是加上环境颜色（以常量 0 传入）并用漫反射颜色（以常量 1 传入）乘 L.N。

3）设置最终结合器的 alpha 部分为法向量图纹理的 alpha 值。用它结合 alpha 测试来剔除方形图中不包含球体法向量的部分。

图 5-21（彩页中也有）显示了这个效果在游戏场景中的应用。第二幅图关闭了 alpha 测试以显示实际的方形图。

为了编译寄存结合器程序，我们从一个 .rc 文件中调入程序，然后生成一个新的 OpenGL 编译列表。程序被送入编译列表中分析，以便于随后的快速选择。

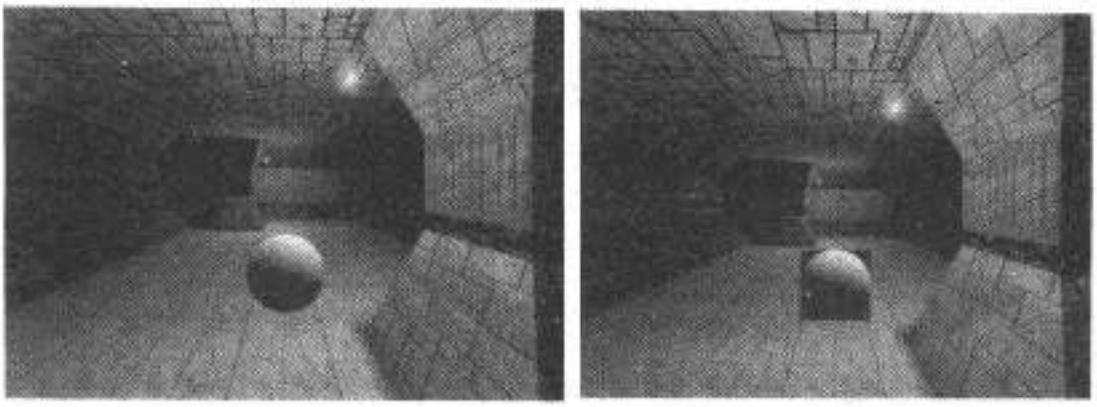


图 5-21 游戏引擎中的方形图块渲染流水线程序。第二幅图关闭了 alpha 测试以显示方形图块背景

```
flyString str=g_flyengine->flyackdatapath+"programs\\"+file;
flyFile rcfile;
if (rcfile.open(str))
  {
  char *buf=new char[rcfile.len+1];
  memcpy(buf,rcfile.buf,rcfile.len);
  buf[rcfile.len]=0;

  rclist=glGenLists(1);
  glNewList(rclist,GL_COMPILE);
  nvparse(buf);
  g_EndList();

  delete buf;
  rcfile.close();
  }
```

以下来自着色器 set_pass 中的代码通过调用编译列表，来设置所有的寄存结合器参数并开启寄存结合器的扩展集。

```
if (rclist)
  {
  glCallList(rclist);
  glEnable(GL_REGISTER_COMBINERS_NV);
  }
```

5.4.4 纹理地址编程

NVIDIA 所称的术语——纹理着色器是第二个提供给程序员的逐像素功能程序（见图 5-18）。增强后的纹理查找包含以下功能：

- **相关**：对于 2D 读取，它使用前一次读纹理的结果来影响当前阶段的（u，v），或直接使用前一次读纹理的结果作为（u，v）。
- **点积**：这个功能计算纹理坐标（u，v，w）向量与从前一个着色器阶段继承下来的一个向量相乘所得的高精度浮点点积。正如我们用寄存结合器所做的那样，它能用来计算明暗值。（实际上，从程序员的角度来看，在架构的不同阶段实现相同的功能会使人对架构本身感到困惑。）

5.4.5　纹理地址编程——Phong 映射

这个方法又一次使用了法向量图代表物体，并假定 **L** 和 **H** 是常量。其步骤如下：

1）使用点积功能来计算 **N.L**。

2）使用相同步骤来计算 **N.H**。

3）用以上两步的结果作为纹理坐标：

$$u = \mathbf{N.L} \quad v = \mathbf{N.H}$$

来查找被称为 Phong 贴图的纹理，它包含预先计算好的 **N.L** 和 **N.H** 的值。这看上去有些多此一举——我们用 **N.L** 和 **N.H** 作为纹理坐标来查找它们自身。其实，应该牢记我们只拥有设置并使用纹理来生成逐像素效果的函数。这一点凸现了本技术和完全像素程序技术的区别。

4）用一个寄存结合器实现指数函数。

5.4.6　顶点和像素编程以及多步着色器

如 5.4.1 节中所提到的，把顶点编程和像素编程包含在着色器结构中是很方便的。例如卡通渲染使用两步来绘制图像，每一步采用一个不同的顶点程序。然而，使用多步渲染仍然有一定的限制和排他性规则。它们是：

- 我们通常不能使用多纹理来合并全部类型着色器的所有步，如 5.1 节所提到的那样。如果两步渲染使用不同的顶点或像素程序，那么它们无法合并。
- 若没有使用多纹理，则寄存结合器程序只能使用一个纹理输入源。这给寄存结合器功能带来了限制。

对此，一种策略是只对静态结构使用多纹理合并，使用光照贴图和相乘混合，并让顶点和像素程序不在多纹理状态下操作。由此，每一步渲染可以和一块纹理、一个顶点程序和一个寄存结合器程序相联系。可以通过允许每一步渲染选择一个以上的纹理来突破单纹理限制，但这是以损失向后兼容性为代价的。

5.5　动态纹理

动态纹理是每帧中生成的纹理贴图。这表明纹理的生成和对纹理内存的传输应该在 GPU 中进行。有三种方法可以做到这一点。第一种方法，在用纹理生成新的一帧之前把纹理写入帧缓冲区，然后再传送到纹理内存。这种方法适用于大多数流行的 GPU。但使用帧缓冲区的方法有一定的缺陷：纹理分辨率由窗口的分辨率决定，而且像素格式也必须与窗口相同（实际上纹理可能不需要深度信息）。

第二种方法是使用像素缓存（NVIDIA 的像素缓存）。像素缓存的尺寸和像素属性是与当前显示模式无关的。使用像素缓存是很直观的，其过程包括直接渲染它，再把内容拷贝到纹理或把像素缓存当作纹理，然后像静态纹理一样使用它。

最后一种方法，我们可以先渲染到后备缓存，从这里拷贝到系统内存，再拷贝到纹理内存。这是最慢的方法，但不需要任何硬件支持，还允许在系统内存中进行修改。

动态纹理有很多应用。我们将学习到的最常见应用就是动态反射。这里动态反射要么指静态镜子中的反射，要么指一个物体在场景中运动时反射其周围的环境。其他应用包括动态

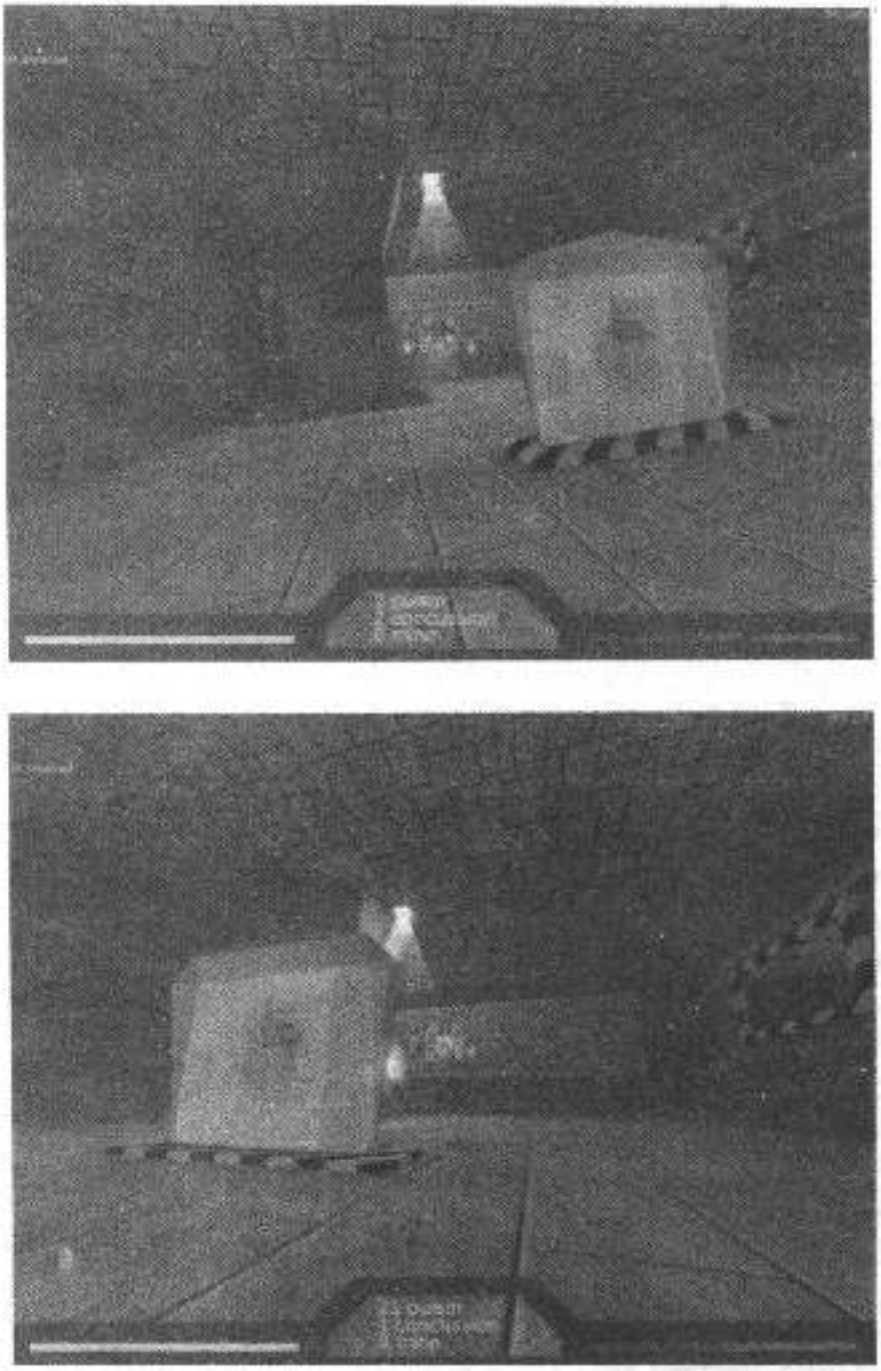


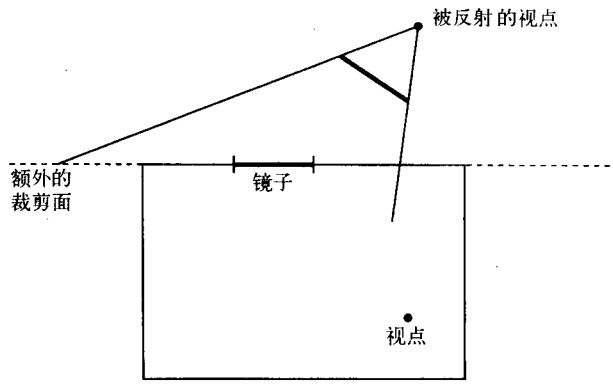
图 5-22 两项模式 1 入口图像的视图。玩家移动时，入口图像保持不变，它是从放置在入口后面了物体后上方的光源上的视点所生成的。从图像中可看到入口后镜子物体的后部。一般来说，入口的目标在另一场景内

过程式纹理生成，比如火焰和使用实时图像处理效果的应用。

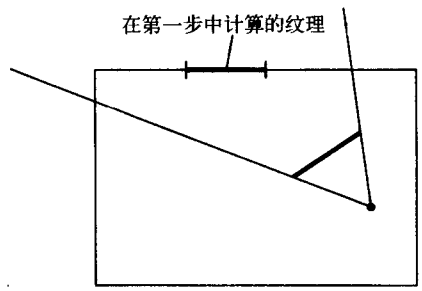
现在考虑动态纹理的一个简单的应用：镜子和入口。这里的入口指的是一个层次中的窗

户，从这里能看见相邻层次内发生的活动。你可以回顾第 1 章，由于 BSP 处理是局限在一个层次内的，因此无法实现对相邻层次的观察。入口能用某种形式的镜子算法实现；而镜子可以被认为是能从中看到反射内容的入口。

我们现在来看三种可能的入口“模式”。第一种模式相当于一架保安照相机，它被赋予一个视点和在层次内的视见方向。渲染后的入口图像作为一整块纹理加到入口多边形上。图 5-22（彩页中也有）显示了两个模式 1 的入口视图。



a) 第一步：从被反射的视点把场景渲染到一块纹理上去



b) 第二步：渲染正常场景

图 5-23　渲染镜面反射

玩家移动时，入口图像保持不变，它是从放置在入口/镜子物体后上方的光源上的视点生成的。因此这个图像包含地板和入口/镜子物体的后部。

第二种模式是镜子模式。图 5-23 显示理论上用以生成镜面反射的两个步骤。先从被反射的视点渲染，再正常渲染场景，把被反射的视图纹理映射到镜子物体上。生成反射图像时，我们必须在镜面的位置放置一个裁剪面以确保在镜子后没有物体（夹在被反射的视点和镜子之间）会出现在渲染后的反射图像中。图 5-24（彩页中也有）是入口/镜子物体对玩家的反射。

镜子算法可分解为以下 5 个步骤：[⊖]

1）找出镜子里的照相机位。这通过把当前视点沿着镜面法向量方向移动两倍视点到镜子的距离就能做到：

```
plane.normal=Z;
  plane.d0=FLY_VECDOT(pos,Z);
  camobj.pos-=2.0f*plane.distance(camobj.pos)*Z;
```

2）再找出反射原始照相机朝向的镜面照相机朝向（Z）。把原始照相机的 Z 分量（即观察朝向）投影到镜面坐标（X，Y，Z），反转 Z 的符号并重构向量。

```
v.x=FLY_VECDOT(X,camobj.Z);
v.y=FLY_VECDOT(Y,camobj.Z);
v.z=FLY_VECDOT(Z,camobj.Z);
v.z=-v.z;
camobj.Z=X*v.x+Y*v.y+Z*v.z;
```

⊖ 在理论上用两步算法生成一个正确的镜面反射是很容易的（见图 5-23）。第一步把反射后的场景渲染到帧缓冲区中。第二步正常渲染场景但不覆盖镜子区域。每一步我们都渲染 3D 物体，而不是 3D 物体混合 2D 反射图像。然而这种技术的第二步需要某些掩模功能，比如模板缓存，而目前模板缓存并不是所有 GPU 都支持的。

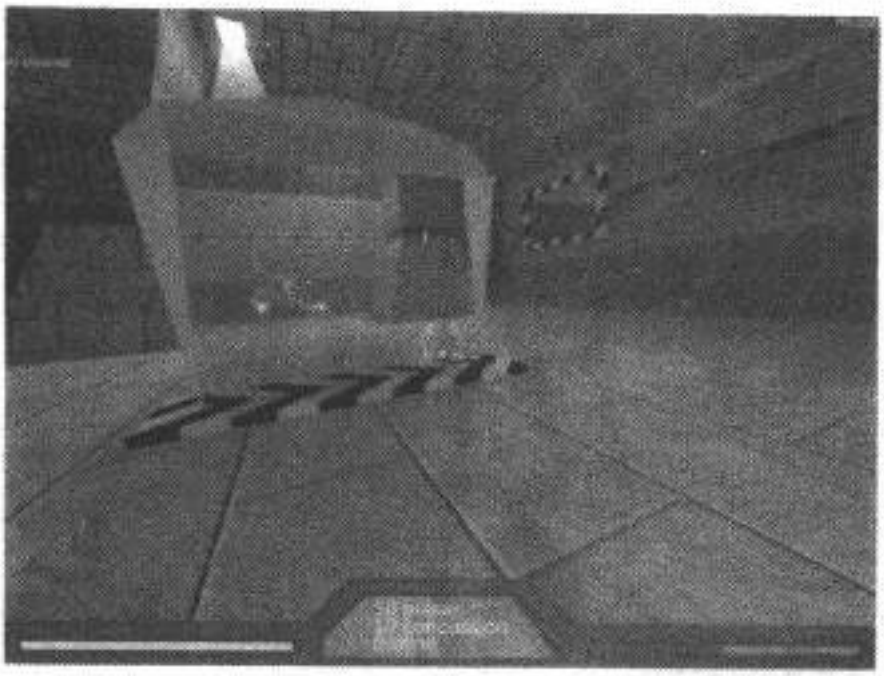


图5-24 入口模式2（镜子模式）反射出玩家

3）找出镜面照相机的X和Y轴。对Y向量执行相同的过程，计算Y和Z的叉积作为X。

```
v.x=VLV_VECDOT(X,camobj.Y);
v.y=VLV_VECDOT(Y,camobj.Y);
v.z=VLV_VECDOT(Z,camobj.Y);
v.z= v.z;
camobj.Y=X*v.x-Y*v.y+Z*v.z;
camobj.X.crossa(camobj.Y,camobj.Z);
```

4）把镜子顶点投影到屏幕空间以计算镜子的纹理坐标。渲染后的图像不能一整块加在镜子上；它通常比用来填充镜子图像的尺寸要大。当照相机靠近镜子时才使用一整块图像。

```
for( i=0;i<4;i++ )
{
  gluProject(
     mirror_verts[i].x,mirror_verts[i].y,mirror_verts[i].z,
     g_flyengine->cam_model_mat,
     g_flyengine->cam_proj_mat,
     g_flyengine->cam_viewport,
     &dx,&dy,&dz);
  mirror_texcoord[i][0]=(float)dx/g_flyengine->cam_viewport[2];
  mirror_texcoord[i][1]=(float)dy/g_flyengine->cam_viewport[3];
}
```

5）用渲染后的纹理绘制镜子多边形。

由于为镜子渲染的图像是透视投影，所以我们不能把它映射到一个方形图来表示镜子。纹理插值把方形图当作一个用来接收纹理的普通计算机图形对象。然而，我们必须“强迫”设置纹理插值为与投影后非线性入口顶点相对应的采样模式。图5-25（彩页中也有）直观地显示了这个问题。里面的图像是放置在场景中的一块玻璃。细分后的入口顶点就在那片绿点中。所以我们需要把入口或镜子细分为几个顶点。这个方法有个额外的优点，那就是可以用

细分来为镜子和入口实现一个动态 LOD 系统。距离镜子越远，顶点数就越少。

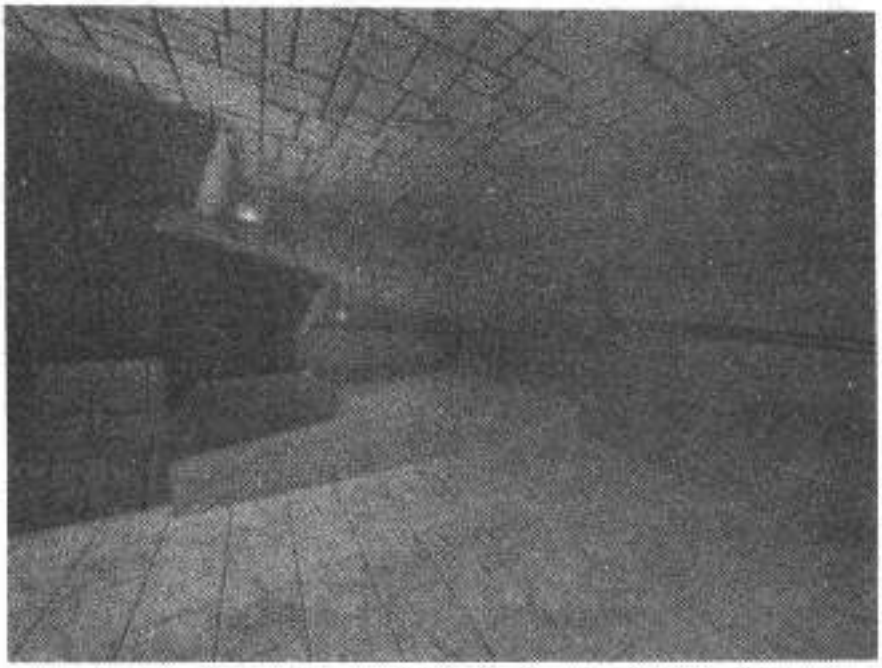


图 5-25　一个细分后的入口映射了场景的投影

第三种入口模式像镜子一样处理，但并不计算目标照相机位置，而是可以预定义在场景中的任何地方。这种模式下放置在任何位置的虚拟照相机取玩家的朝向为其本身的朝向。图 5-26（彩页中也有）表明了这种思路。而且此模式下，玩家通常可以飞入入口物体并被传送到入口的目的地。

5.6　特效

任何有关游戏实践渲染技术的章节都不会不提及“特效”。它们一般通过组合使用了“高技巧”的标准渲染方法来构建。特别地，很多特效使用告示牌——二维纹理映射实体来伪装不同的物体。游戏中大多数明显的视觉复杂度是由这些技术所带来的。我们通过给出典型例子来结束本章。

5.6.1　燃烧尾迹

图 5-27（彩页中也有）显示了两幅燃烧尾迹特效图。

组成特效的面是从 $\mathbf{P}_1$（飞船的涡轮发动机）到 $\mathbf{P}_n$ 绘制的，这里 n 是燃烧尾迹键保持按下的时间内生成的点数（见图 5-28）。纹理被加到面上，并不断重复直到尾迹停止，而且还总是紧接着一个渐变特效。

这个效果要用二块纹理来实现。第一块光环纹理（见图 5-29a）保留在 $\mathbf{P}_1$，并总是垂直于观察者。纹理被映射到由飞船的spritelight color变量设定颜色的面上。这个映射的结果是

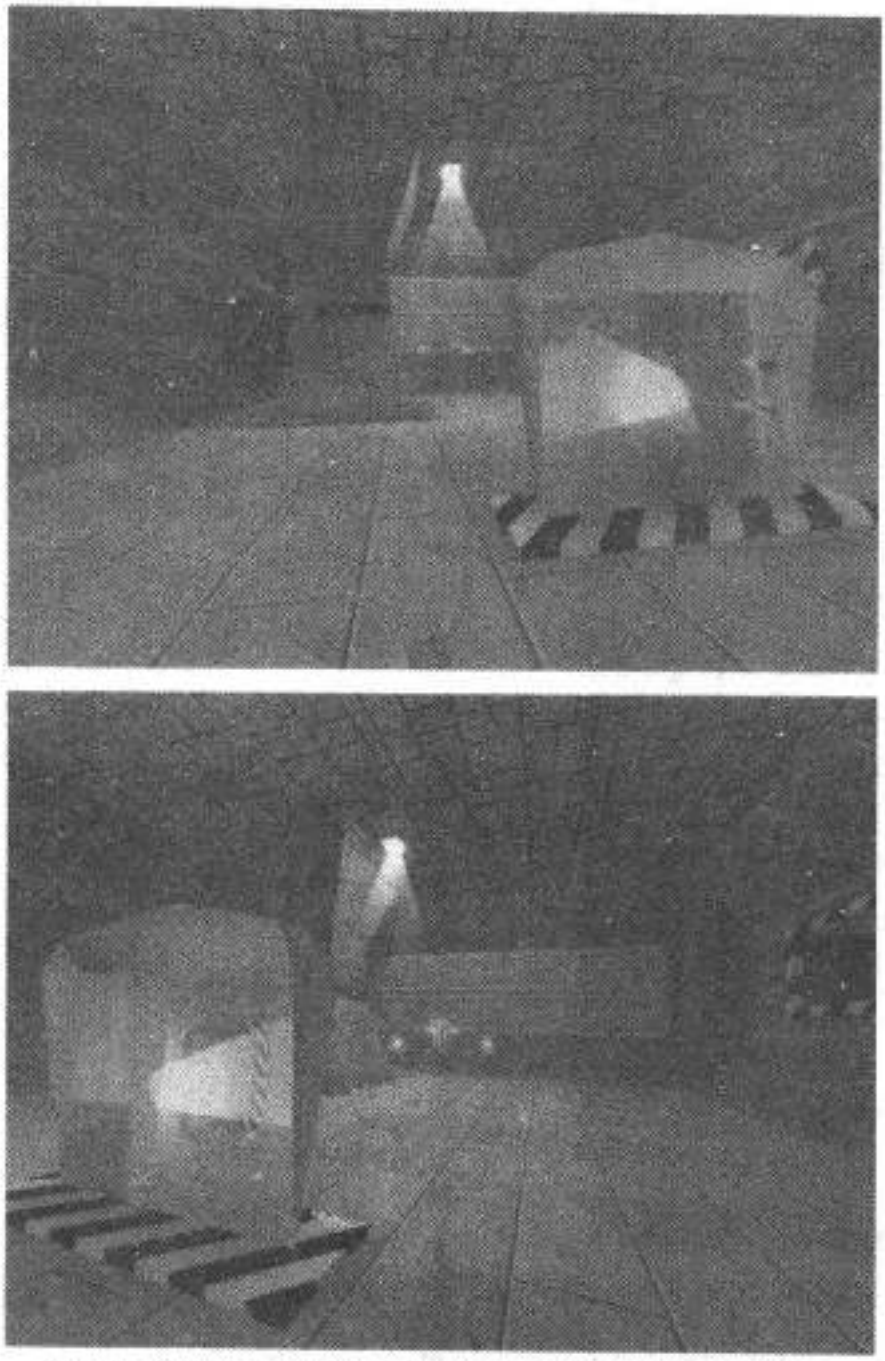


图 5-26 在这个例子中，入口目标同样是放置在入口/镜子物体后上方的光源。相对于入口目标区入口照相机朝向与相对于入口的普通照相机相同。因为当玩家在入口/镜子物体附近移动时，可以看见目标的不同区域，相比之下，模式 1 的视图总是相同的

将两种颜色（表面颜色和纹理颜色）相乘。纹理的颜色也根据飞船速度来进行修改，以增强移动的动感。当飞船停止时，纹理变得几乎看不见。第二块纹理（见图 5-29b）是纹理 Alpha

通道，最后一块（见图 5-29c）是 RGB 纹理贴图。

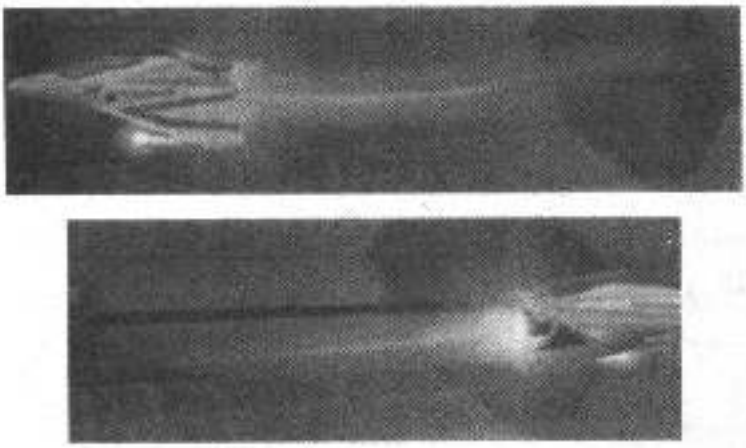


图 5-27　燃烧尾迹特效



图 5-28　当燃烧尾迹变换时，按照飞船的路径来生成方形图



a) 光环　　b) Alpha 通道　　c) RGB 通道

图 5-29　关于燃烧尾迹的纹理

5.6.2　加速器

这个特效包含不断改变球径的同心球体（图 5-30）。当它们达到原来大小的 1/3 时，就恢复到原始大小。球体用一块滚动的纹理来映射（见图 5-31，彩页中也有）。一个粒子系统发射出粒子，从球心发出的光线和粒子相接。光线物体是纹理映射的平面多边形（告示牌）。图 5-32（彩页中也有）显示了最终的效果。

5.6.3　脉冲星

这个最终特效结合使用了告示牌技术，它可以用来表示导弹尾迹或一个独立实体。图 5-33（彩页中也有）是由两个互相垂直的截面构成的。这在某种程度上解决了使用告示牌

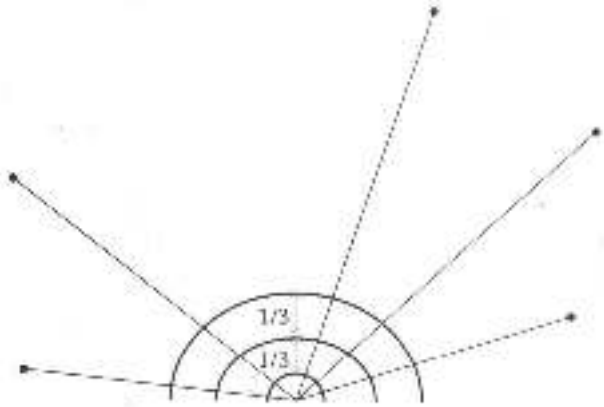


图 5-30 球体网格过

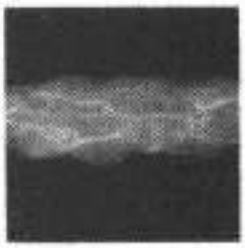


图 5-31 所使用的纹理

所产生的平面感问题。首先，绘制纵向截面的一半，再将它翻转以产生整个截面（图 5-33a 和 b，彩页中也有）。然后旋转这幅图像的一行像素来产生另一个截面（图 5-33c，彩页中也有）。最后把这个物体混合到帧缓冲区中，把火焰加入当前帧缓冲区并保持黑色背景不受此效果的影响。

使物体看上去有立体感的关键是把它绕着自己当前位置旋转，使得法线照相机位和当前

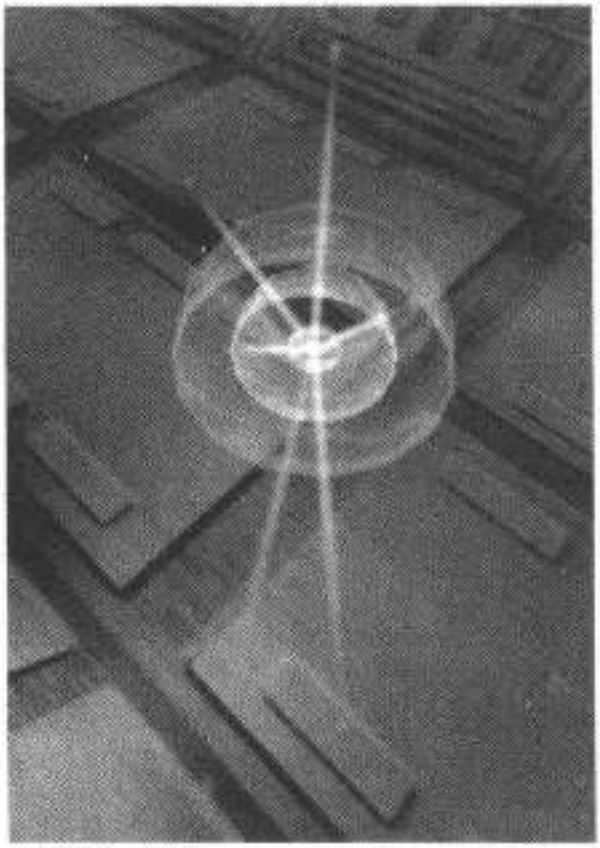


图 5-32 最终效果

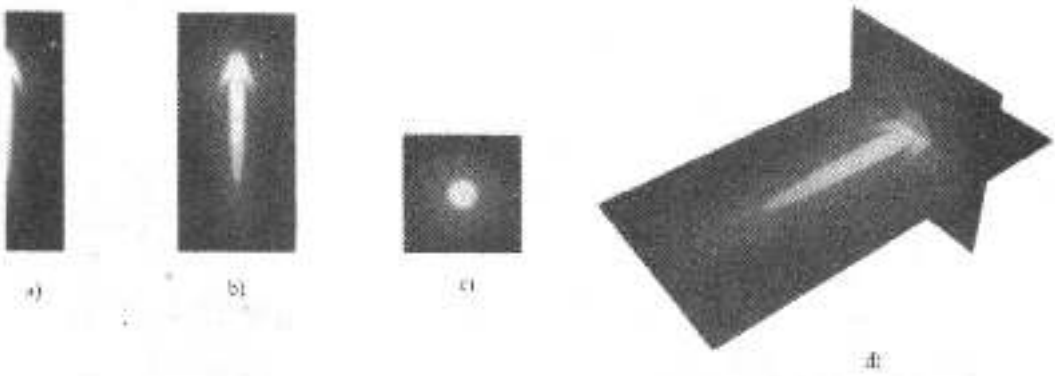


图 5-33　法方形图和纹理来构建物体

火焰物体位置的向量是纵向截面的法向量。照相机的位置和方向以及火焰物体的位置和方向的空间关系是不断改变的，因为我们假定通常发射武器的飞船能在火焰击中目标之前立即改变方向。

告示牌的朝向计算如下。首先，我们算出 $\mathbf{V}$（见图 5-34）：

$$\mathbf{V} = \mathbf{C}_p - \mathbf{M}_z$$

它包含了照相机和火焰物体的当前位置。$\mathbf{X}$ 是垂直于 $\mathbf{V}$ 和 $\mathbf{M}_z$ 的向量：

$$\mathbf{X} = \mathbf{V} \times \mathbf{M}_z$$

这里 $\mathbf{M}_z$ 是火焰的方向（即导弹的 Z 轴）。

我们把纵向截面固定在包含 $\mathbf{X}$ 和 $\mathbf{M}_z$ 的平面里。这样保证了无论火焰和观察者之间的空间关系如何变化，观察者总能看见告示牌恰巧为一个表示火焰的三维物体的截面。

图 5-35（彩页中也有）显示了总是垂直于观察者的脉冲星，图 5-36（彩页中也有）显示

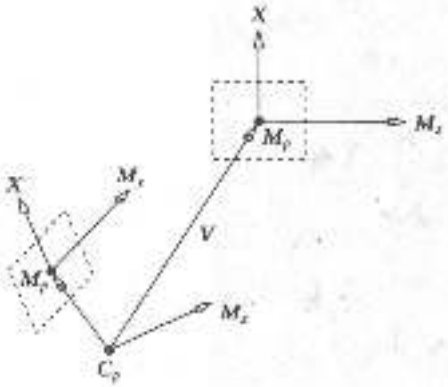


图 5-34　对两个 $\mathbf{M}_p$ 位置，法面有关 $\mathbf{C}_p$ 的导算告示牌

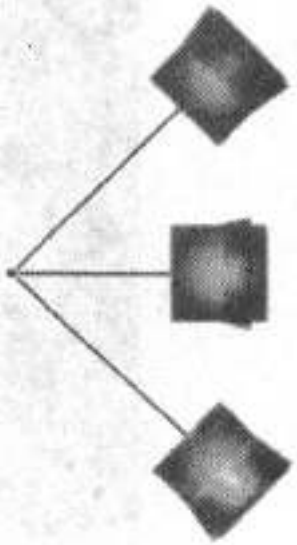


图 5-35　垂直于观察者的脉冲星物体

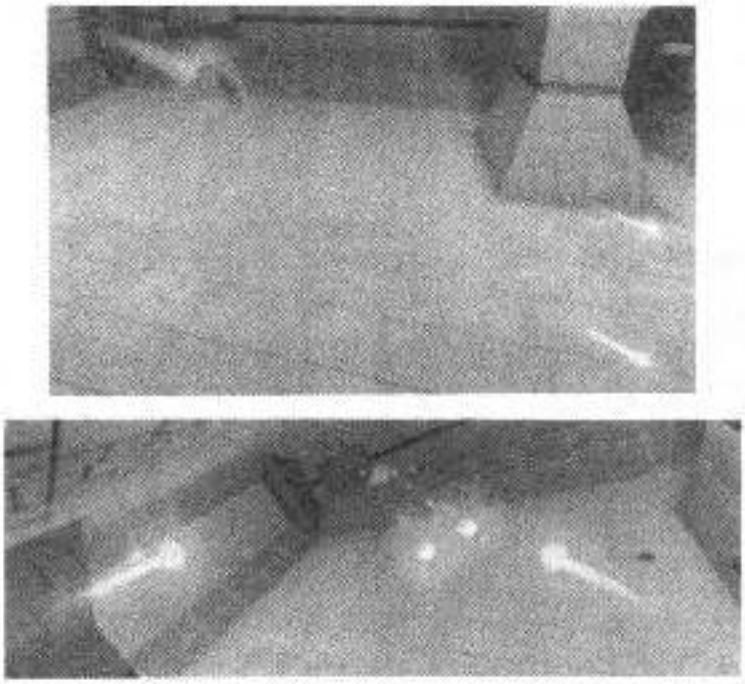


图 5-36 在一个游戏场景中的脉冲枪

了最终的效果。

附录 5.1 使用和探索着色器

着色器编辑器

高效使用着色器涉及到着色器编辑器，本节我们就展示这样一个工具的功能和结构。主窗口不是把载入的着色器显示为一个二维贴图，就是显示为一个纹理映射的物体。每个着色器文件能存储许多着色器，每个着色器都可以包含至多 8 步渲染。

图 A5-1（彩页中也有）显示了有两个着色器的着色器文件，它用于渲染 gravator 导弹。

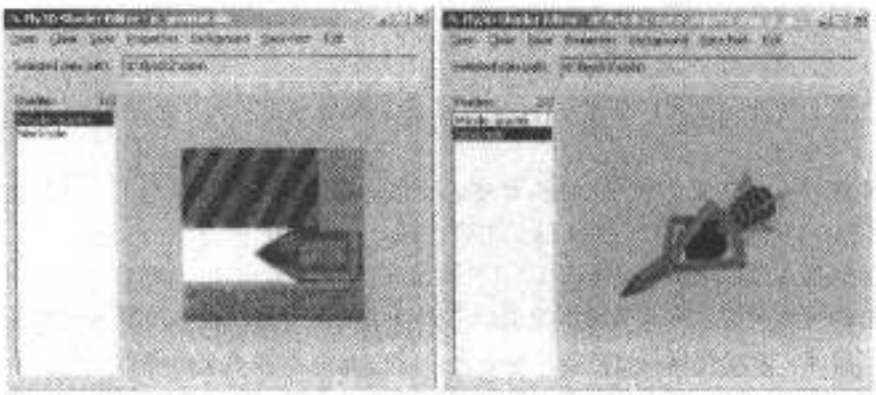
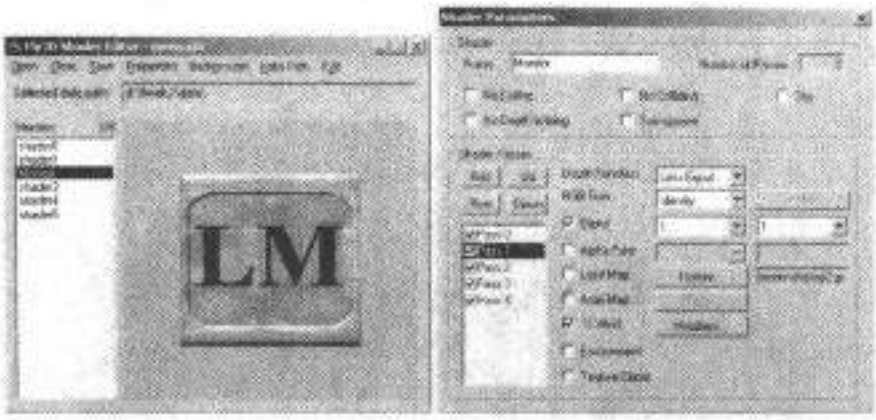
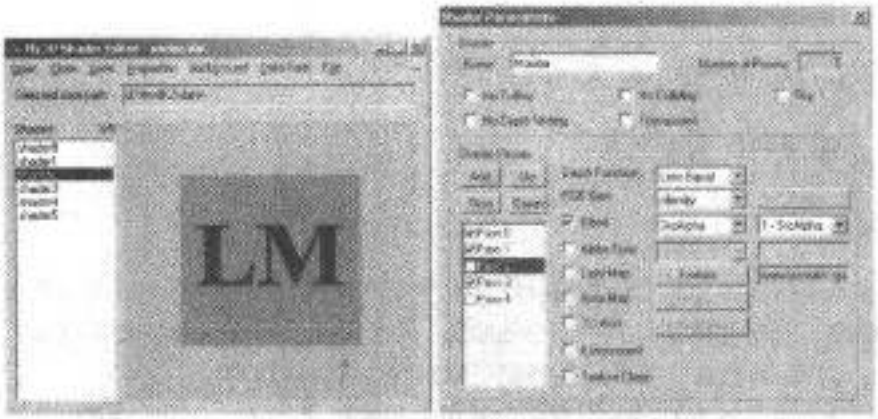


图 A5-1 渲染导弹的两个着色器

选择“属性”菜单会启动一个显示当前所选着色器步骤的新窗口，在窗口里可以进行修改。它能单独显示一个渲染步骤，也可显示任何几个步骤，只要在每步前的复选框上打勾就行了。在着色器属性窗口内可以修改任何着色器选项。图 A5-2 显示了监视器着色器及其参数。



a) 显示了所有5步渲染和第1步渲染状态的监视器着色器



b) 显示了第0、1和3步的监视器着色器

图 A5-2　监视器着色器

在着色器参数窗口里可以添加/删减渲染步骤，并通过在列表中上下移动来改变它们的顺序。

对监视器的红线渲染步骤，我们需要使用纹理坐标修改器。它是在一个单独的窗口内实现的（见图 A5-3）。在这个例子中我们要在垂直方向变换纹理。

周期函数是在另一个窗口中修改，用来调整颜色和定义缩放。图 A5-4 显示了 blue ball 导弹，它每秒闪烁两次，将其不透明度值设置在一半到百分百之间，不断来回改变。可以用正弦波、三角波、方波和锯齿波作为函数。共有四个浮点数定义了函数的位移、振幅、相位和频率。

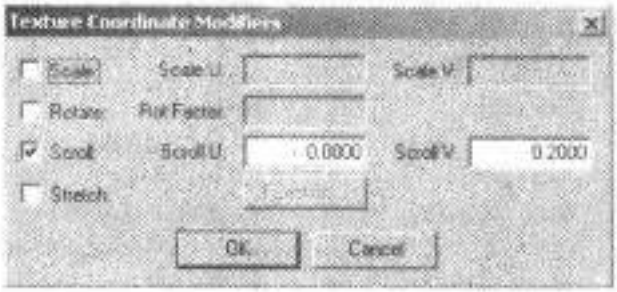


图 A5-3 垂直滚动纹理



图 A5-4 设置周期函数的参数

附录 5.2 NVIDIA GeForce3 上的顶点编程

本附录给出了 5.4 节所需的具体细节。有关顶点编程的详细文档参见 http://developer.nvidia.com。

指令集

指令集包含了 17 条寄存器-寄存器指令。每条指令操作四分量源寄存器并把结果输出到一个目标寄存器中。需要输入因子的指令（RCP，RSQ，EXP，LOG）必须指定一个修改器（'.x'，'.y'，'.z'或'.w'），来表明四分量寄存器中的哪个分量将被作为输入因子。生成输出因子的指令计算出结果因子后把它拷贝到寄存器的所有四个分量中（除非应用了输出寄存器掩码）。

指　令	参　数	操　作
NOP		不进行任何操作
MOV	dest, src	移动
MUL	dest, src1, src2	分量相乘
ADD	dest, src1, src2	分量相加
MAD	dest, src1, src2, src3	src1 和 src2 相乘后再加上 src3，把结果存入 dest
RSQ	dest, src	src 的平方根 dest.x = dest.y = dest.z = dest.w = 1/sqrt (src)
DP3	dest, src1, src2	三分量点积

（续）

指　令	参　数	操　作
DP4	dest, src1, src2	四分量点积
DST	dest, src1, src2	计算衰减向量（见正文）
LIT	dest, src	计算 Phong 光照（见正文）
MIN	dest, src1, src2	基于分量的求最小值操作
		dest.x = (src1.x < src2.x)? src0.x:src1.x 等
MAX	dest, src1, src2	基于分量的求最大值操作
		dest.x = (src1.x > = src2.x)? src0.x:src1.x 等
SLT	dest, src1, src2	dest.x = (src1.x < src2.x)? 1:0 等
SGE	dest, src1, src2	dest.x = (src1.x > = src2.x)? 1:0 等
EXP	dest,src.w	dest.x = 2**(int)src.w
		dest.y = src.w 的小数部分
		dest.z = 2**src.w
		dest.w = 1.0
LOGT	dest,src.w dest.	x = ((int)src.w)的指数
		dest.y = (src.w)的尾数
		dest.z = log2(src.w)
		dest.w = 1.0
RCP	dest, src.w dest.	x = dest.y = dest.z = dest.w = 1/src

顶点属性寄存器

一共有 16 个顶点属性寄存器。每个属性寄存器存储“逐顶点数据”，并可用数字或助记符来引用。

顶点属性寄存器名	助 记 符 名	助记符意义
v[0]	v[OPOS]	对象位置
v[1]	v[WGHT]	顶点权重
v[2]	v[NRML]	顶点法向量
v[3]	v[COL0]	主要颜色
v[4]	v[COL1]	次要颜色
v[5]	v[FOGC]	雾坐标
v[6]	—	—
v[7]	—	—
v[8]	v[TEX0]	纹理坐标 0
v[9]	v[TEX1]	纹理坐标 1
v[10]	v[TEX2]	纹理坐标 2
v[11]	v[TEX3]	纹理坐标 3
v[12]	v[TEX4]	纹理坐标 4
v[13]	v[TEX5]	纹理坐标 5
v[14]	v[TEX6]	纹理坐标 6
v[15]	v[TEX7]	纹理坐标 7

顶点结果寄存器

一共有15个顶点结果寄存器，它们的运算结果输出到负责设置与光栅化的硬件。

顶点结果寄存器名	分 量	解释说明
o[HPOS]	齐次裁剪空间位置	(x,y,z,w)
o[COL0]	主要颜色（正面）	(r,g,b,a)
o[COL1]	次要颜色（正面）	(r,g,b,a)
o[BFC0]	背面主要颜色	(r,g,b,a)
o[BFC1]	背面次要颜色	(r,g,b,a)
o[FOGC]	雾坐标	(f,*,*,*)
o[PSIZ]	点尺寸	(p,*,*,*)
o[TEX0]	纹理坐标集0	(s,t,r,q)
o[TEX1]	纹理坐标集1	(s,t,r,q)
o[TEX2]	纹理坐标集2	(s,t,r,q)
o[TEX3]	纹理坐标集3	(s,t,r,q)
o[TEX4]	纹理坐标集4	(s,t,r,q)
o[TEX5]	纹理坐标集5	(s,t,r,q)
o[TEX6]	纹理坐标集6	(s,t,r,q)
o[TEX7]	纹理坐标集7	(s,t,r,q)

临时寄存器

一共有12个临时寄存器。每个寄存器用名字‘Rn’引用，这里n是在[0, 11]范围内的整数。这些寄存器既可读又可写。在每一次调用顶点程序时，它们的初始值为(0, 0, 0, 0)。

地址寄存器

寄存器A0.x就是“地址寄存器”。这个寄存器是只写的，且只有它的x分量是可以写入的（例如“MOV A0.x, R0;”）。这个寄存器（使用‘.x’修改器）可以用作常量寄存器的索引。在每一次调用顶点程序时，其初始值为(0, 0, 0, 0)。

常量寄存器

常量寄存器用于存取放在常量内存空间中的常量数据。最典型的就是那些不随每个顶点改变的数据。一共有96个常量寄存器，用‘c[n]’来引用，这里n是在[0, 95]范围内的整数。存储在常量内存空间中的数据也可以使用地址寄存器来存取（例如“MOV R0, c[A0.x];”）。这些寄存器在普通顶点程序中是只读的，在顶点状态程序中既可读又可写。任何给定指令只能存取一个常量寄存器。然而，当读一个常量寄存器时，可以在指令中使用相同的常量寄存器作为多重源寄存器（例如‘ADD R0, c[5], c[5];’）。

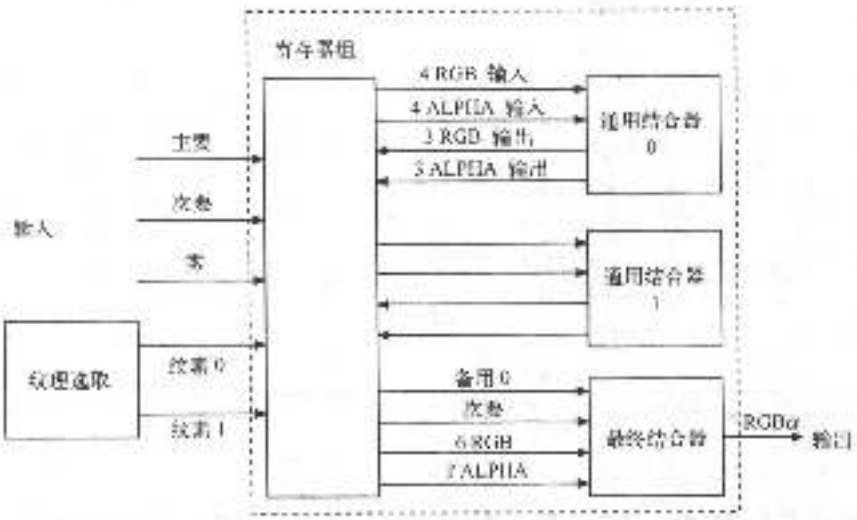
附录5.3 NVIDIA寄存结合器操作

这个附录是对5.4.3节的扩展，并给出了更多关于寄存结合器操作的细节。一篇相当易

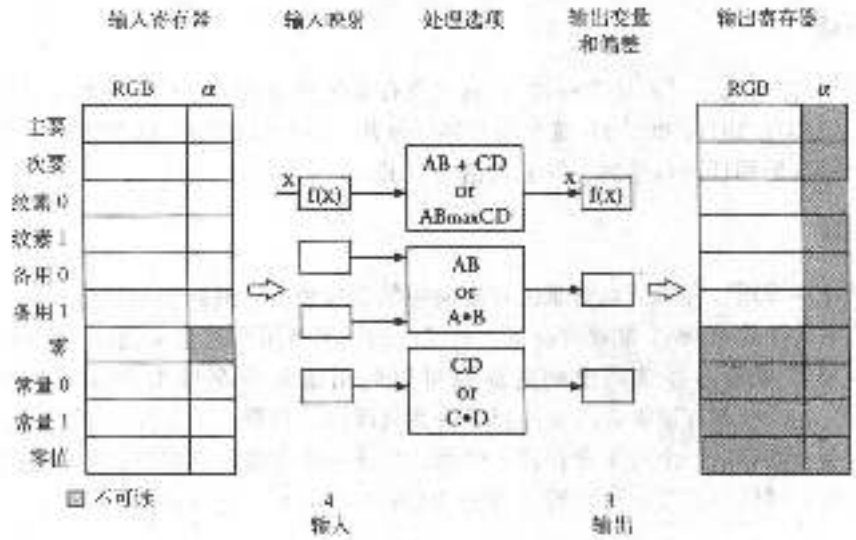
读的有关寄存结合器的文章是[KILG99]。

结合器数据流（对两个通用结合器和一个最终结合器）显示在图5-19中。从输入顺序流经通用结合器0的数据流要么成为通用结合器1（如果使用它）的输入数据，要么成为最终结合器的输入数据。

输入寄存器所包含的不同参数是：插值后的漫反射和镜面反射颜色（主要和次要颜色），开启的纹理单元中过滤后的纹素和雾因子（alpha）。两个空闲的寄存器作为扩放寄存器，四个常量寄存器包含雾的RGB颜色、两个RGBα颜色常量和零值。常量寄存器对于结合器操作是可读但不可写的。雾颜色和两个常量可由应用程序指定初始值。



a) 带两个通用和一个最终结合器的完整的寄存结合器架构



b) 针对RGB部分的通用结合器操作

图A5-5　NVIDIA寄存结合器架构

图 A5-5a 显示了数据流通过两个通用结合器的架构，并强调了对 RGB 和 alpha 的分别处理。每个通用结合器能输出三个 RGB 值和三个 alpha 值。因此用两个通用结合器能产生多达 12 个结果/像素，其中包括 4 个点积。

从图 A5-5b 中可推知，4 个变量 A、B、C 和 D 被赋给寄存器，并且执行了 8 个输入映射中的一个。三个输出值经过计算、缩放和混合，被变换到 [- 1，1]，再写入输出寄存器。

除了这些操作之外：

AB + CD

AB

CD

A·B

C·D

还可以用以下的 mux 操作：

$$\text{mux}(AB, CD) = (\text{Spare0}[\text{alpha}] \geq 1/2)\ ?\ AB : CD$$

在渲染之前，输入分配、输入映射、处理选项、输出度量和偏差由应用程序来配置。最终结合器计算并输出一个 RGBα 值，如图 A5-5a 所示，最终结合器总是计算：

$$AB + (1 - A)C + D$$

第6章 几何处理

6.1 简介

我们把在过去10年中提出的用于处理大规模多边形模型的算法和分析技术统称为几何处理。由于着色模型的研究始于30年前，几何处理长期以来一直在演变，并主要受到实时渲染复杂物体需求的推动。

虽然有人认为硬件的发展将能满足实时应用的需求，然而事实却并非如此。有很多证据能证明这一点。例如，在非游戏的应用中，不断增加的模型复杂度似乎总超前于当前的硬件水平。包含数百万三角形的CAD模型和VR应用也很常见。现今游戏虽然还未达到这样的复杂程度，但在将来很可能会使用远比目前复杂的场景。尽管当前的三角形处理速率很快，可带宽和内存问题仍会延续下去。尤其是3D内容在Internet上的传输，需用精心设计的渐进方式。可扩展性（scalability）是游戏业中众所周知的未解问题，即游戏在新旧不同的平台下会有不同的表现性能。在一个新的和一个稍旧的系统之间，三角形处理速率可能会存在着非常大的差异。

为什么我们要使用多边形表示形式呢？在过去，它们的“地位”并不确定。许多渲染程序被设计成用于处理双三次参数化网格，这其中最著名的可能就是REYES [COOK87]，他的模型面片被切割成大约1/2像素宽的微型多边形。尽管有以上这个形式和20世纪80年代研究出的众多其他表示形式，多边形网格仍然成功胜出，并且在事实上成为游戏业和绝大多数通用3D图形应用的标准。

游戏产业中早期的处理限制（如今正被迅速消除）推动了对美工精心设计出的模型的使用，不论是单个模型，还是小型LOD系列模型。在这里，用小规模多边形模型取得较好视觉效果的惟一途径就是让美工手工创造模型。图6-1所示的对象就是一个典型例子，对纹理和少量多边形的创造性应用构成了一个游戏人物原型。随着业界的不断进步，游戏人物的复杂度会不断增加，需要在一帧中显示多个精细模型的新游戏类型将会出现。

本章的目的是探究各种不同的描述多边形网格的方法。这些方法广泛应用于LOD渲染、渐进式传输和多分辨率网格的编辑中。使用一个低分辨率基础网格的多分辨率表示形式时，一个显著优点就是它能够在很大程度上简化动力学计算[⊖]。

事实上三角形网格并没有“天然”的多分辨率表示形式，因此本章介绍了多种不同的处理方法。我们可以把三角形网格看作由两个部分构成：顶点位置和可看作平面图的连通信息。我们并不能直接比较任意两个三角形网格，因为它们的连通图是不同的。这和信号处理相反，因为任一对单变量规则采样的函数能被分解成，举例来说，小波。但对于表面而言，除非存在潜在的参数化表示形式，比如贝济埃面片形式，否则不存在解析或“天然”的多分

⊖ 这正是骨骼动画的原理（第7章）。这里我们把它看作一个二层表现形式。基本层上是骨架，皮肤构成用于渲染的第二层。

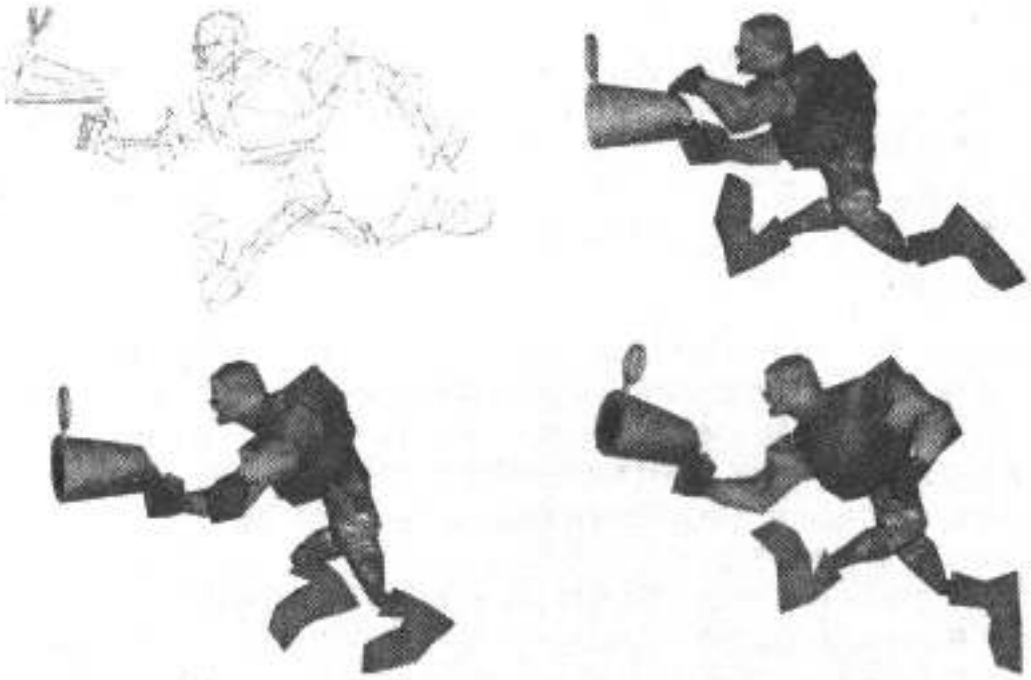


图 6-1 手工制作的人物动画范例子

辨率处理方法。

这一领域的研究在20世纪90年代的迅速发展意味着目前已有一大批成果可用。所以，仅仅一个章节无法详细回顾这些内容。我们所努力做的只是取出其中的主要原理，加以分类，并给出具有代表性的例子。我们将介绍多边形网格分段线性和微分几何技术之间的一个重要联系。在本章的最后还将介绍一些微分几何的背景知识。

应当指出的是，在当前术语“多边形网格”（polygon mesh）指的是三角形网格。不仅日渐强劲的硬件渲染系统被设计成用来处理三角形原型，而且三角形还拥有很好的拓扑学优点：它们能描述任意形状和拓扑的表面。而其他一些表示形式，比如双三次参数化面片，必须满足“四边性”的限制条件。

最后，我们注意到简化算法有两个相关的应用。第一，在显而易见的LOD应用中，我们先生成一系列的网格，例如渐近式网格，然后按照当前视见参数渲染其中适当的一个。第二个应用由第一个变化而得，它结合一张图（比如法向量图）来渲染一个低分辨率的基础网格。这样能够恢复网格在简化中“丢失”的细节（不包括网格的轮廓边）。这实际上是一种渲染方法，具体内容主要在第4章中简述。本章将介绍它的一部分。

首先，让我们来考查一下网格及其简化相关的一些基本概念，及进行简化的原因。

6.2 推动因素和定义

6.2.1 离线和实时阶段

正如游戏中构建场景管理和预计算光照一样，网格简化算法执行一个（通常是）耗时的

离线处理阶段。在这个阶段里，算法简化网格，并根据某些最优性原则构造多分辨率结构。这一阶段中所损失的，就是从网格中的某一级 M^k 简化到下一个较粗糙的级别 M^{k-1} 的质量。考虑一个简单的局部算法，它每次去除一个顶点，并通过在每一级检查去除每个顶点后的影响，再选取要去除的顶点。则对于原始网格 M^n，它的复杂度为 O（$n!$）。这样的代价在很多应用中都显得太高了。在实时阶段（顾名思义我们希望它实时运行），算法决定渲染某个分辨率的模型 M^k，随后把它从多分辨率表示形式中构造或提取出来。

6.2.2　拓扑因素

算法或者保留、或者修改网格的拓扑结构。应用程序也可以允许或禁止对拓扑结构的修改。一个物体的拓扑结构按它的属（genus），即它所含洞的数量进行分类。比如，类似球体或立方体的物体没有洞，则它的属为 0，环状或类环状物体为 1，茶杯茶壶等物体也是 1。

拓扑不变（topology-preserving）的算法保持物体的属不变，因此并不像修改拓扑的算法那样能高度简化网格。根据定义，一个修改拓扑的算法所生成的多分辨率表示形式，在渲染时随着洞的消失或重现，将会显示突变的效果。图 6-13 是一个简单的拓扑修改序列的例子。

若一个三角形网格的局部拓扑处处等价于圆盘，则它是 2D 流形网格。也就是说，考查表面上的任一顶点，都可以发现一个相连的三角形环——单环邻域（one-ring neighbourhood）。一个流形拓扑的网格中，任意两个三角形之间有且仅有一条公共边。除此之外，边界边有且只有一个面和它相连，并且任一顶点的邻域必包含一个面环。无边界的流形拓扑的网格（闭合面）满足欧拉公式：$F - E + V = 2 - G$。其中 F 是面数，E 是边数，G 是属数，V 是顶点数。

这个性质的重要之处在于，满足此性质的网格能够直接进行简化。我们将在本章中研究的算法应用也保持这一性质。而对于非流形（non-manifold）拓扑网格，简化算法还需特殊的启发式方法。

6.2.3　离散简化与连续简化

时至今日，大多数游戏应用程序使用模型的多级离散子模型。这些子模型之间的多边形数量有很大差异。它们是离线生成的，并由美工辛苦地优化而成。应用程序保存下模型所有的级别，并在运行时选择一个合适的级别。这种方法有两个显而易见的缺点，一是其数据结构比单一的高分辨率模型消耗更多的内存；二是创建的代价很高。由于所有级别的模型通常按多边形的数量进行分类，突变常会发生。除非用某些设备处理才行，比如混合，但它也有明显的缺陷，那就是必须同时渲染两个级别的模型。

在连续结构[⊖]中，一系列的模型在离线阶段生成。它通常被编码为基础网格 M^0 和一些在运行时执行简化逆操作所需的信息。在运行时，根据某些标准来选择一个合适级别的模型，构造它并加以渲染。

此类方法的一个流行的例子就是渐进式网格（progressive mesh），也称为 PM（见 6.5.1 节）。它有一个重要特性，那就是数据结构所需的内存通常情况下并不比原始网格 M^n 多。而且在 PM 中，一个被称为几何渐变（geomorphing）的方法，在简化逆操作过程中实现了伪连续渐变。

⊖ 术语“连续”需加以一些限制条件。由于简化过程每次去除一个顶点或一条边，使每一个级别的网格改变形状，所以其结构在某种意义上仍是离散的。

关于几何渐变的一个最简单的例子如下。可以试想通过把一条边简化为一个顶点来实现从精细模型向粗糙模型的变换：

$$M^i \to M^{i-1}$$

一个迭代的压缩算法在每一次边去除后，见图 6-2a，生成构成边的顶点（$\mathbf{V}_1$，$\mathbf{V}_2$），和新顶点 $\mathbf{V}$ 的位置。通过几帧的时间，我们可以把 $\mathbf{V}_1$ 和 $\mathbf{V}_2$ 移动到 $\mathbf{V}$。这样的话，边就渐渐收缩为一点，而不会突然消失。这个例子中一个明显的缺点在于当我们还需要显示 M^{i-1} 时，还必须渲染 M^i。

渐进式网格结构可以被表示为连续的系列模型，在其中从 M^i 到 M^{i-1} 的变换是一次单边去除；也可以视为一个离散 LOD 模型集合，在这里级别之间的变换构成了 n 次边去除操作。

6.2.4 物体内部分辨率变化[○]

在一系列模型中考查某个级别，则它的多边形分辨率在整个表面上趋于一致。这样的模型并不能很好地适用于绵延很长距离的大物体。一个经典的例子是在地面上以小角度观看地平线。整个地形被投影到大部分可视区域，而我们要求细节级别能按照距离视点的远近在模型内部变化。当前有一些方法可以使用，且它们中的大多数都利用基于高度图的地形的拓扑限制，也就是意味着 2D 空间上存在一个规则参数化形式。很多方法使用四叉树或三角形二叉树（例如 [DUCH97]）来满足自适应细分（adaptive subdivision）的要求。随后，实时复原过程从细分结构中构造物体，并根据屏幕空间误差标准来裁剪树枝。此类方法已有 20 – 30 年的历史，且是由模拟飞行产业开创的。但这种方法却很少被应用于一般的多面体（见 6.5.3 节）。

6.2.5 对称性/可逆性

虽然在处理时不对称，但几乎所有的方法在一定意义上是对称的。即简化的逆过程能得到和离线简化阶段时完全一样的模型。我们将在 6.5.3 节讨论它的一个例外类。

6.2.6 局部简化操作

普通的局部简化操作集合如图 6-2 所示。这些都是简化算法中最基本的操作。它们从

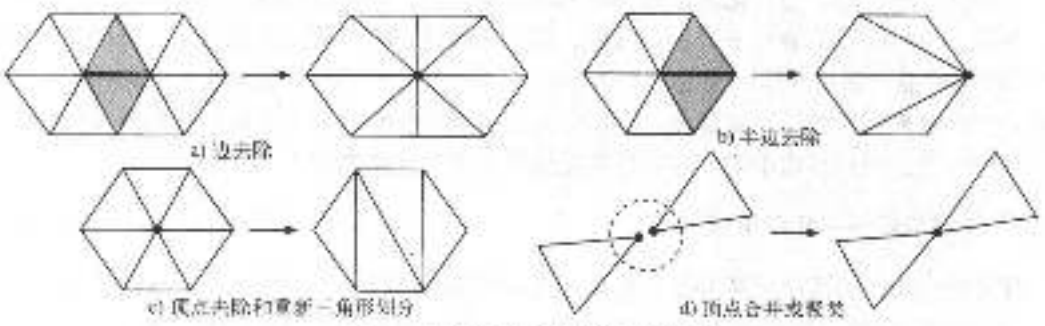


图 6-2 网格简化中使用的局部操作

○ 这有时也可被称为视点相关简化（这是一个多少令人费解的术语，因为所有的多分辨率渲染都是由某个视点相关的标准所控制的）。

M^k 去掉单个实体来得到 M^{k-1}。大多数算法都是重复使用以上操作中的一个，其中最常用的是边去除。局部简化操作的分类如下：

(a) 边去除：这个简单且流行的操作是很多方法的基础。它把一条边去除成一点。简单是其流行的主要原因。另一个原因是，在简化过程（用于寻找新顶点的最佳位置）和简化的逆过程中（重构边能根据时间连续生长，即几何渐变），都能运用连续的算法。允许为新（合并后的）顶点选择位置意味着由去除的三角形和相邻的区域形成的空间，可以有无数种重新三角形划分的方法。这里需注意新顶点的位置不一定要位于被剔除的边上。

(b) 半边去除：严格来讲，它是前面介绍的操作的子集。它把一条边去除到它的一个顶点上去（事实上这种方法有时被称为子集位移）。它能以平均 6 种方法中的一种来剔除顶点（顶点的度数是 6）。因此，本操作的优点是新顶点最优位置的搜索空间和顶点度数成线性关系。

(c) 顶点去除和重新三角形划分：这种操作去除一个顶点并把剩余的三角形网格重新划分。它的缺点是得到的 M^{k-1} 都趋于等边形。这并没有很好地利用顶点，特别是对于过于简单的网格。对于 n 个的三角形邻域，其搜索空间的维数是第（$n-2$）个 Catalan 数。即 14、42 和 132 分别对应 6、7 和 8 个的三角形邻域。

重新三角形划分是计算几何中一个重要的经典问题。正如我们将看到的，它的一般性使我们能结合使用微分几何度量。这能在顶点数较少的情况下得到更高质量的简化结果。

(d) 顶点聚类：这是一种考虑顶点位置而并非其连通情况的操作，它合并临近的顶点。此方法的重要性在于它可以应用于任意网格，而不像前面介绍的方法只能应用于流形网格。找出合并哪些顶点的最简单方法是均匀分割物体的包围盒，并把那些分布在一个单元格内的顶点作为合并的候选对象。

由上可知，使用重新三角形划分或半边去除得到的 M^{k-1}，它的顶点是 M^k 及其更高级别网格顶点的子集。而边去除方法允许顶点移动——M^{k-1} 中的新顶点可以在 M^k 中去除的边上移动。纯粹的边去除是保持拓扑结构的。但如果允许压缩任意顶点，哪怕它们之间没有边相连，则拓扑结构会发生变化。

6.3 排序（误差）标准

简化算法中另一个与简化质量密切相关的重要部分，是确定网格实体被去除的顺序的过程。无论在运行时要渲染哪一级别的网格，都需要确保它有尽可能高的质量。我们通过对每一个 M^k 上可能的顶点操作，加上一个操作于 M^k 与候选 M^{k-1} 的度量标准来做到这点。我们使用这个度量来选择去除特定顶点。不同的方法采用的标准有很大不同。我们将介绍其中有代表性的一些。简化算法中的大多数研究成果都集中在这个方面。

6.3.1 排序标准——外观相似

度量简化操作的质量的最直接方法是，用一个标准图像评估标准来估计 M^k 与 M^{k-1} 的不同。由于最终呈现在用户面前的是渲染后的图像，所以我们应当注重它的质量。

可简单定义评估两幅图像差异的标准：

$$E = \frac{1}{n^2}\sum_u\sum_v \| M_R^k(u,v) - M_R^{k-1}(u,v) \|^2$$

其中 $M_R^k(u,v)$ 和 $M_R^{k-1}(u,v)$ 是 M^k 和 M^{k-1} 的渲染图像，n 是像素个数。

这种简单的方法并没有揭示其中的一些实际难点。为了使此方法有效工作，我们必须用一个有代表性的视点样本来估计 E，但这会导致在离线阶段花非常高的代价。

在近期的一份报告中，Lindstrom 和 Turk［LIND00］采用了这种方法，并发现它能生成高质量的几何简化。报告中还陈述到，这种基于图像的方法与基于几何的方法相比，有注重轮廓边的质量、对模型隐藏部分高度的简化、注重光照过渡影响以及纹理敏感的简化等优点。他们使用边去除算法，根据最小视觉差异来确定边的去除顺序。算法对一系列样本视点计算上述表达式并求和。这些视点均匀分布在一个包围测试物体的球内，并与正十二面体的顶点一致（得到 20 个图像）。而此球球径是测试物体最小包围球球径的 2 倍。

作者用了一个独特的方法来解决图像生成代价高的问题。它通过对已存在的渲染图像进行增量变化来生成下一次迭代。对于一个候选边去除，算法渲染位于新顶点的单环邻域内的所有新三角形 T^{k-1}，来替换同一邻域内已存在的渲染过的三角形 T^k。这个过程再对所有样本视点重复执行。既然平均而言 $|T^k| = 10$，则一个仅渲染新三角形的算法将会是完美的。Lindstrom 和 Turk 称此为反渲染（unrendering）问题，让 T^k 在渲染 T^{k-1} 之前进行反渲染（用来显示之前被遮挡的部分表面）。为此，他们利用边去除的屏幕空间局部性，为每一个像素行与列保持一个三角形桶结构。由此产生以下用 T^{k-1} 替换 T^k 的算法：

1）清除屏幕空间中包含这两个集合的矩形区域 R。

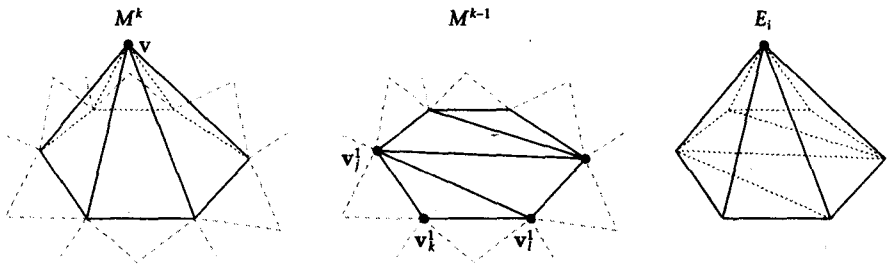
2）渲染所有 R 中的三角形（可见与隐藏的），但不包括那些将被反渲染的。

3）渲染集合 T^{k-1}。

6.3.2 排序标准——局部体积不变

一种直接度量顶点去除与重新三角形划分的几何学方法就是比较 M^{k-1} 与 M^k 体积的不同，即候选的顶点去除引起该顶点单环邻域中局部体积的变化（见图 6-3）。从图中可以看出，体积变化 E_i 是由以点 i 为顶点的那些面为侧面，和以重新三角形划分后的底面组成的多面体。在这个特定的例子中，我们可以认为 E_i 是由 4 个四面体组成，从而构成了多面体的体积：

$$\sum_{i=1}^{4}\frac{1}{6}\begin{vmatrix} \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z & 1 \\ \mathbf{v}_{jx}^i & \mathbf{v}_{jy}^i & \mathbf{v}_{jz}^i & 1 \\ \mathbf{v}_{kx}^i & \mathbf{v}_{ky}^i & \mathbf{v}_{kz}^i & 1 \\ \mathbf{v}_{lx}^i & \mathbf{v}_{ly}^i & \mathbf{v}_{lz}^i & 1 \end{vmatrix}$$



a) 顶点的邻域　b) 重新划分顶点区域　c) 结果误差多面体

图 6-3 误差多面体 E_i 的例子

同样的方法也能用于边去除。这里，新顶点通过压缩边来产生。Lindstrom 和 Turk 介绍了一种寻找顶点最佳位置的分析方法。他们注意到在边去除中，通过变形“扫过”的四面体体积可以分为正的和负的。比如在图 6-4 中，新顶点 v 移动到了新三角形所在的平面之下，因而四面体被归为负的一类。为了使体积变化为 0，可令所有有向四面体体积总和等于 0。由此产生了一个线性等式，从而限制 v 位置的解在一个平面内。在此平面内的进一步优化，可通过最小化每一个多面体的无向体积来得到。

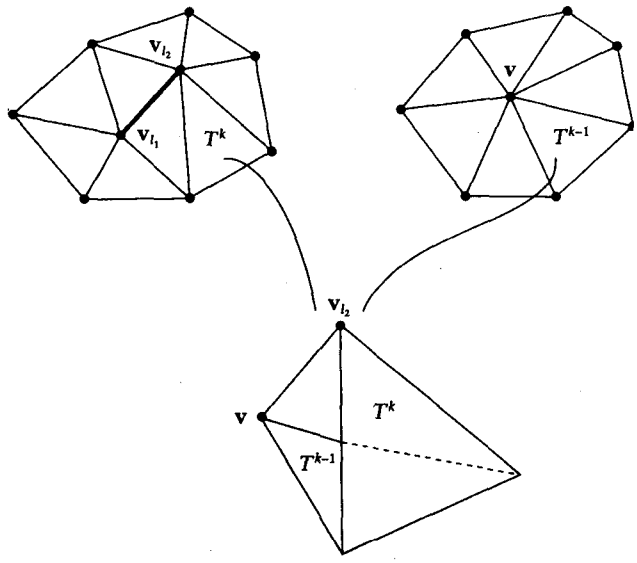


图 6-4　在边去除中使用误差多面体

6.3.3　排序标准——二次误差度量

最初为顶点聚合算法研究的二次误差度量（QEM）[GARL97] 也可用于边去除等其他简化操作。由候选顶点合并操作引入的误差是被合并顶点的 QEM 之和，而这个和正是新顶点的 QEM。

Garland 用了一个深刻的 2D 类比来说明他的 QEM。如下所示，图 6-5a 显示了一条候选去除边（$\mathbf{v}_i$，$\mathbf{v}_j$）。我们要求合并后顶点的位置是最优的。为了做到这些，线段与每一个顶点都相关：

$$L_i = \{A, C\}$$
$$L_j = \{B, C\}$$

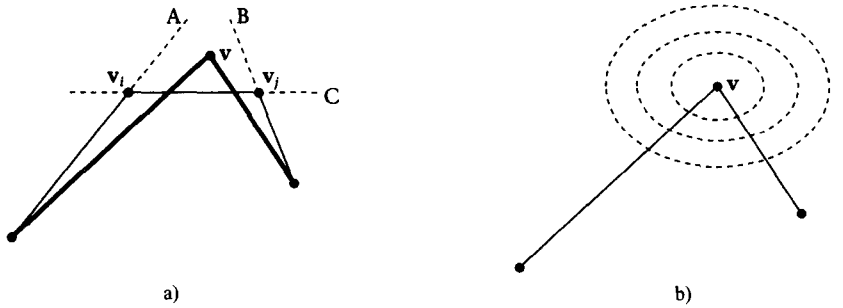


图 6-5　QEM 的 2D 类比，这些椭圆是位置 v 的等代价线

这些线段是连接顶点 i 和 j 的边。随着边的去除，我们定义一个新顶点 v。于是和新顶点相关联的边是集合

$$L = L_i \cup L_j = \{A, B, C\}$$

然后按照与 L 中所有线的距离平方和最小来选择 v 的位置。这正如图所示，在由 A、B、C

构成的三角形的中心上。图 6-5b 显示了一系列包围 v 的同心椭圆。它们代表等代价线：

$$E(\mathbf{v}) = \varepsilon$$

在 3D 中，我们把平面与顶点相关联，此时误差等代价线是二次曲面。这正是 QEM 得名的原因。每个顶点和一系列与它相接的表面关联起来。

我们现在使用［HOPP99］中给出的方法来应用 QEM。点 **p** 与一个表面之间的误差度量定义为：点 **p** 与包含这个表面的平面之间距离的平方，称为 $Q^f(\mathbf{v})$。由此定义一个点与顶点 **v** 之间的误差度量为：

$$\mathbf{Q}^v(\mathbf{v}) = \sum_{\substack{\text{faces}\\ \text{that meet at v}}} \text{area}(f)Q^f(\mathbf{v})$$

即每个交于顶点 **v** 的平面的误差度量的加权和。在一次边去除$(\mathbf{v}_1, \mathbf{v}_2)\rightarrow\mathbf{v}$之后，顶点 **v** 被分配到使得$\mathbf{Q}^v(\mathbf{v}) = Q^{v_1}(\mathbf{v}) + Q^{v_2}(\mathbf{v})$最小的位置。

也就是说，在若干候选边中选择具有最小$Q^v(\mathbf{p})$值的那一条。

现在我们对包含顶点 $\mathbf{v}_1$、$\mathbf{v}_2$、$\mathbf{v}_3$ 的面定义 $Q^f(\mathbf{v})$：

$$Q^f(\mathbf{p}) = (\mathbf{n}^T\mathbf{p} + d)^2$$

其中 **n** 是面的法向量$\dfrac{(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)}{\|(\mathbf{v}_2 - \mathbf{v}_1) \times (\mathbf{v}_3 - \mathbf{v}_1)\|}$

$$d = -\mathbf{n}^T\mathbf{v}_1$$

因此

$$\begin{aligned} Q^f(\mathbf{v}) &= \mathbf{v}^T(\mathbf{n}\mathbf{n}^T)\mathbf{v} + 2d\mathbf{n}^T\mathbf{v} + d^2 \\ &= \mathbf{v}^T(\mathbf{A})\mathbf{v} + 2\mathbf{b}^T\mathbf{v} + c \end{aligned}$$

是一个二次方程。其中：**A** 是 3×3 的对称矩阵，**b** 是列向量，c 是标量。

所以最佳的顶点 **v** 和它的位置从下式中求得：

$$2\mathbf{A}\mathbf{v} + 2\mathbf{b} = 0$$

即得到

$$\mathbf{A}\mathbf{v}_{\min} = -\mathbf{b}$$

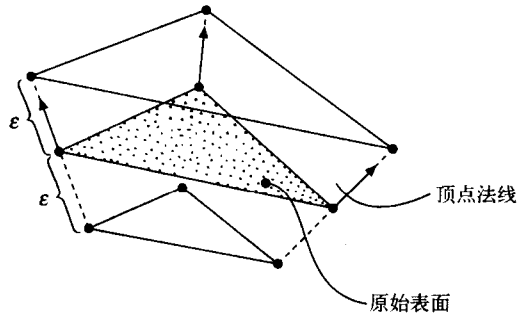
QEM 的一个主要优点就是它的效率，而且恰只有一个二次曲面与每个顶点相关联，共需要 10 个组成分量（**A** 有 6 个，**b** 有 3 个，c 有 1 个）。

6.3.4 排序标准——简化外壳

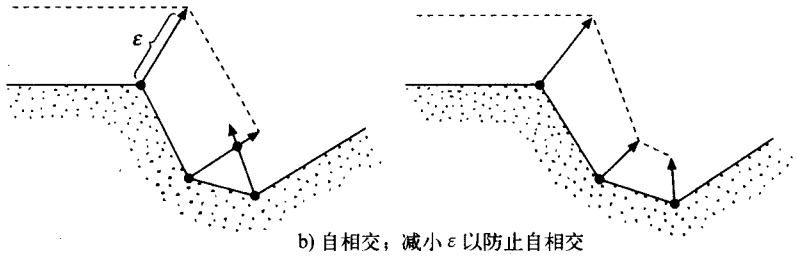
Cohen 等人［COHE96］在 1996 年发表的工作报告中，构造了包含原始表面的简化外壳。与偏移曲面类似，简化外壳与原始表面上下的偏离不会超过预先规定的容差 ε。在简化过程中，误差标准变为检查被修改的三角形是否相交于两个外壳中的任何一个。由此可确保简化网格和原始网格表面的误差在 ε 以内。此方法保持拓扑不变，并保证平面在任何地方都不超过用户指定的容差。

离线阶段包括简化外壳的构造，以及随后的简化过程。构造简化外壳在原理上是很直观的。我们只需要分别沿着顶点法向量的正向和负向，移动每一个顶点（见图 6-6a）。对单个三角形构造出的结果被称为基本三棱柱。但惟一的问题是可能发生自相交，而这必须避免。自相交可以在凹区域内由正偏移面引起，也可以在凸区域内由负偏移面引起。Cohen 等人通过允许在需要的地方适当减小 ε 来防止这种情况发生（见图 6-6b）。一个防止自相交的条件

是，顶点不能被移动到它所属边的 Voronoi 区域之外。考虑到在 3D 中确定 Voronoi 区域的代价非常之高，Cohen 等人采用了一种迭代办法，从使用一个与原始表面相同的外壳开始，在随后的每一步以一部分 ε 的量来移动外壳顶点，并检查其相邻三角形是否相交。



a) 简化外壳



b) 自相交；减小 ε 以防止自相交

图 6-6　简化外壳

在简化阶段，将一个新的候选三角形相对于外壳表面进行测试。由于三角形的顶点是原始表面顶点的子集，所以它们一定位于两个外壳之间，由此把对候选的有效测试转化为对两个外壳的相交测试。可以通过把三角形结合在空间划分方法中，以使这些测试更有效。

6.4　简化与属性

绝大多数的物体模型都具有顶点属性和参数形式。最常见的属性是由顶点坐标（u，v）从纹理中所确定的颜色。在游戏设计中，这种纹理参数通常是由美工交互地建立起来的。本节中，我们将考察模型被简化时参数形式是如何变化的。一般来说，如果我们构造一系列的模型，则对每一个模型需要使用相应的纹理坐标。如果在简化过程中移动了顶点，则需要计算新的纹理坐标。

如 6.5.1 节所述，另一种重要的简化方法是结合基础网格与细节贴图来代替 M^n 中的信息。在这种情况下，我们需要在边界阴影问题不太突出的时候终止简化过程。如果使用这种方法，那么我们只需得到基础网格的一个参数化形式。Cignoni 等人［CIGN98］通过在基础网格中采样来直接得到它。在简化之后，基础网格的每个面都以预先确定的分辨率被规则地采样。对每一个样本点，找出原始表面上和它相距最近的点，并由此确定其相关属性（纹理、法向量或位移贴图的值）。

［COHE98］中提出了一个更为复杂的方法，用于处理多分辨率表示形式中的属性。其应

用是为一个法向量图寻找新的纹理坐标（渲染时使用这个法向量图来表示表面的细节）。在多分辨率结构中，不同细节层次上的映射失真可能比颜色映射失真更容易被注意到（因为法向量图是用于表示物体几何形状的）。

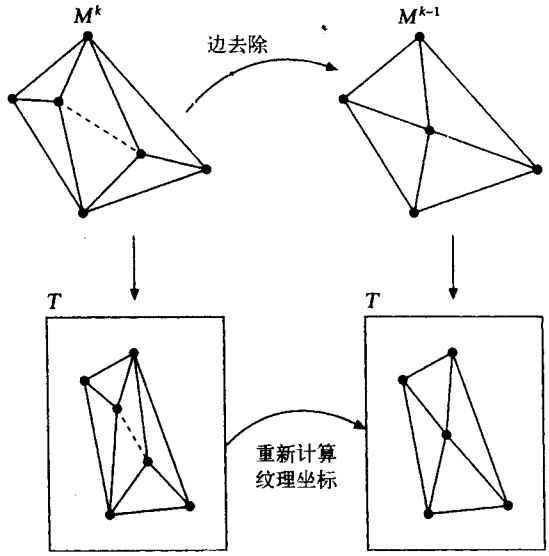
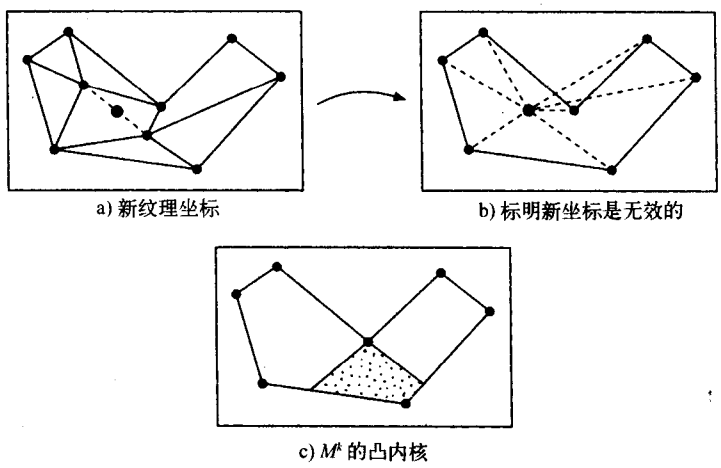


图 6-7 在边去除后计算新的纹理坐标

正如对网格亚采样须按测量 M^k 与 M^{k-1} 表面偏差的误差度量一样，纹理坐标中的偏差也必须最小化。我们考虑一个简单的边去除，这里新顶点位置是塌陷边的两个顶点坐标的中点，也即意味着新纹理坐标是先前两个顶点纹理坐标的中点（见图 6-7）。换句话说，网格简化的几何形式预先确定了新纹理坐标的计算。但是我们必须考虑其他因素的影响。考察图 6-8，它表明了以一个纹理空间中点作为新顶点纹理坐标是无效的情况。这是因为 M^{k-1} 中的新三角形（虚线），会接收它在 M^k 对应的母版三角形的边界以外的纹理。且当前两个三角形在纹理空间中重叠。可以看出：新纹理坐标必须位于由 M^k 的凸面内部形成的阴影区域（见图6-8c），以确保 M^{k-1} 中的三角形获得和 M^k 中相应简化区域相同的纹理区域。



a) 新纹理坐标　　b) 标明新坐标是无效的

c) M^k 的凸内核

图 6-8 非法纹理坐标和合法区域

由于新纹理坐标必须满足限定条件，我们需要一个纹理偏差标准，用它在限定区域内做最佳选择。我们通过把 M^k 中的点 x_k 与拥有同样纹理坐标（u，v）的点 x_{k-1} 进行比较来实现（见图 6-9），即：

$$C(\mathbf{x}_k) = F_k^{-1}(T(u,v)) \text{和 } C(\mathbf{x}_{k-1}) = F_{k-1}^{-1}(T(u,v))$$

其中 C 是 x 获得的 T 在映射 F_k 下的值。

随后这些顶点间的距离通过如下公式给出：

$$D(u,v) = |F_k^{-1}(T(u,v)) - F_{k-1}^{-1}(T(u,v))|$$

这是一个有效的使用纹理参数化表示的表面偏差度量。它又是对已广泛使用的手工操作过程的量化。在这个过程中，美工通过在纹理空间中拖动一个 2D 网格，把世界空间变换到纹理空间映射中。

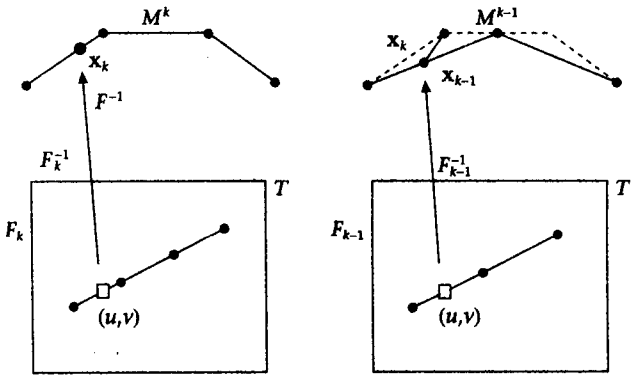


图 6-9　度量纹理偏差

Garland 和 Heckbert［GARL98］扩展了 QEM 以解决这个问题。他们视每个顶点位置与相应的 RGB 纹理值为 6D 空间中的一个点，并且阐述了 QEM 的一个扩展形式。他们这样做的原因是：首先，各属性间存在相互联系；其次，在 QEM 方法中被合并顶点的位置不一定要位于被去除的边上（因此属性无法进行线性插值）。由于在一个三角形上的纹理是进行线性插值的，所以这个 6D三角形顶点存在于 6D 空间中的 2D 平面上。扩展的 QEM 由此来度量点到平面的距离平方。

6.4.1　简化与游戏纹理

纹理偏差在某些游戏系统中可能是灾难性的，因为这些游戏系统中的物体常常使用了合成纹理。为了减轻硬件纹理管理难度，不同的贴图被组合成一张贴图。简化算法中的任何偏差都可能导致顶点得到完全不同的纹理，而不仅是在相同纹理上的一点扭曲。

另一个合成纹理问题与纹理的接缝有关。对于接缝处的普通渲染顶点，通常的处理方法是把它分裂开，并认为它们是分离的顶点，各自与相应的表面和纹理联系在一起。如果这样的模型输入到简化算法中，这些顶点可能会按照不同的方式塌陷，从而在网格中张开一个洞。而最好的解决方法是标记那些位于接缝处的边，并且不允许它们塌陷。

6.4.2　简化和蒙皮模型

上文中，我们的讨论集中在标准的多边形物体。然而在现今的游戏中，一个常见的复杂物体是游戏人物，它的顶点与一个或多个骨骼加权后联系在一起。我们可以方便地使用此类人物的标准姿势来进行建模和纹理映射，但却未必是输入简化算法的最佳模型表示。这是因为它不是游戏中人物在运动时所采用的典型姿态。一个更好的方法是使用从动作序列中采样的一些姿势，并计算边去除度量标准的平均值。当顶点和不止一根骨骼相连时，另一个问题会出现。所以当那些顶点和不同骨骼相连的边去除时，需为这个新顶点计算出新的相连关系。

6.4.3　算法框架

简化算法可使用的最简单可行的框架是贪婪算法。对任意细节级别的 M^k，我们先应用

误差标准 E_i（6.3节）来评估局部简化操作，而后构造一个误差降序排列的优先级队列。对于重新三角形划分或洞填充，这个过程将会产生并且估计所有可能三角形组合的结果。我们首先去掉优先级队列的顶部顶点，得到 M^{k-1}。在总体上确保按照误差度量去除实体后，能使模型间的差异最小。下一步，我们需要为那些所在三角形被上一次简化操作改变的顶点重新计算 E_i，并更新它们在队列中的位置。

去掉顶点后的影响可以通过与邻域或原始网格的几何形状相比来进行衡量（分别归类为局部或整体算法）。整体算法能保证误差可控。

6.4.4 顶点去除算法的重新三角形划分

如果新顶点恰位于被去除的边上，基于简单边去除操作的算法并不能给我们重新划分三角形的自由。顶点去除算法从另一个角度提出了这样一个问题，即如何把由去除顶点产生的洞重新划分为三角形。我们将介绍两种普通的技术，第一种作用于三维空间，第二种作用于二维空间。

3D 中的最小权重三角形划分

重新三角形划分既可以通过把顶点投射到平面从而作用在二维空间，也可作用于三维空间。虽然二维方法很流行，但如果我们可以确定相关三角形不会改变拓扑结构，就能很轻松地摆脱特定的从 3D 投影到 2D 的限制条件，直接在三维空间中处理。这种有效性检验可以通过使用三角形一致性朝向的概念在线性时间内完成。具体地说，我们试图寻找这样一个顶点，在重新三角形划分前后，它位于由三角形及相关法向量定义的每一个半空间中。如果存在这样的顶点，则拓扑结构没有改变。

一种常见的重新三角形划分算法是 Euclidean 最小权重三角形划分，简称 MWT [BARE94]。此算法的一个快速近似算法描述如下。首先定义近旁边（ear edge）为 $e = \{i, j\}$，它将 n 边形沿着边 e 分成三角形 $\{i, j, k\}$ 和 $n-1$ 边形（见图 6-10）。换句话说，它是横跨三个相连顶点的边。每条近旁边 e 被指派一个权重 $w(e)$，这是一个目标函数，需对所有给定的多边形取最小值。例如，w 可以被指定为

$$w(e) = |\mathbf{v}_i - \mathbf{v}_j|$$

由此算法就变成了 MWT 的一个近似。能按照简单贪婪方法构造它，逐步添加当前具有最小的权重 $w(e)$ 的近旁边 e，使相应多边形减少一个顶点。这个过程不断重复，直到进行重新三角形划分的区域中只剩一个三角形为止。在6.5.2节的实例分析中，我们修改了该算法以适合于使用微分几何。

2D 中的重新三角形划分

这种方法用于在 6.5.3 节中介绍的 MAPS 算法的简化阶段。其基本思想是，若问题可被映射到二维空间，那就能直观地进行重新三角形划分并可使用不少已有算法（例如 Delaunay 三角形划分）。此过程的简单例子如图 6-11 所示。

考察被去除顶点的单环邻域或星形物，我们希望把这个结构展开到平面上。我们用附录6.1中介绍的保角

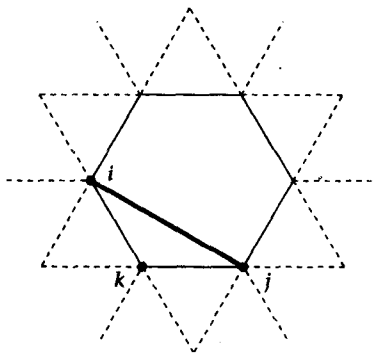


图 6-10 边 $\{i,j\}$ 被定义为近旁边

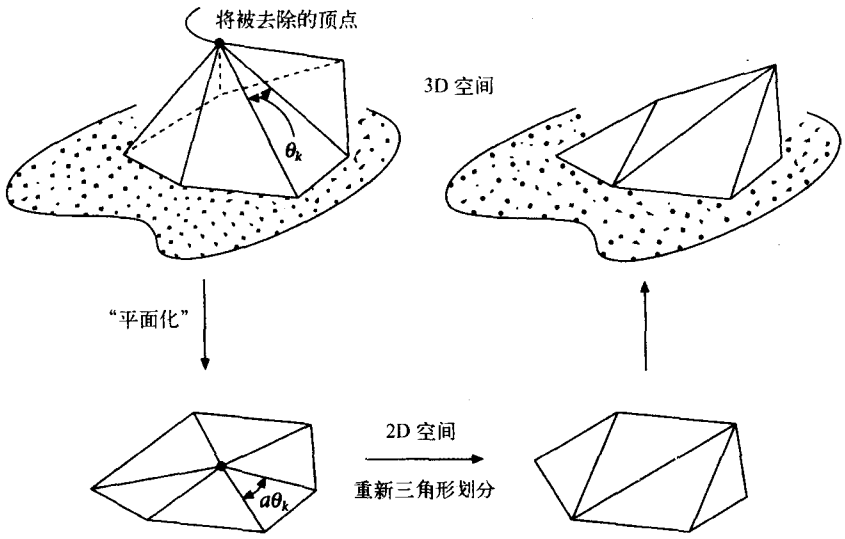


图 6-11　应用保角映射 z^a，把将要被去除顶点所在的星形物映射到一个平面上。在平面上去除顶点，并重新三角形划分

映射 $w = z^a$ 来做到这一点。考虑和被去除顶点相连的每个顶点，用它们来定义一个边集。并以此定义 z 平面中的边为：

$$r_k \exp(i\theta_k)$$

其中 r_k 是每条边的长度，θ_k 是在环状结构中两相邻边的夹角。

所需的映射函数通过下式给出：

$$r_k^a \exp(i\theta_k a)$$

其中

$$a = \frac{2\pi}{\Sigma\theta_k}$$

需要指明的是，正如图 6-11 所示，θ_k 是 3D 空间中相邻边的夹角。

6.5　实例分析

在这一节中，我们将学习三个完整的简化算法。我们选择它们来阐述简化算法中不同的模块怎样结合在一起，并将介绍一些有用的微分几何定义：

1）渐进式网格——这种被广泛接受的方法基于边去除。它被改进以适应依靠视点的选择性调整。

2）使用微分几何——在此方法中，我们引入微分几何参数，用于协助基于顶点去除的简化方法中的重新三角形划分。

3）重新网格划分——这是新近提出的方法，属于一般简化算法。它通过对基础网格的细分来产生用于渲染的网格。

6.5.1　实例分析 1——渐进式网格技术

我们现在来研究一个使用上文技术的典型应用。可能最流行的边去除算法就是 Hoppe 的

渐进式网格了（见图 6-12）[⊖]。Hoppe［HOPP96］意识到在保存所有涉及边去除的分解信息时，保留基础网格 M^0 加上重构信息，比保留原始网格 M^n 加上分解信息更好。存储 M^n 和边去除信息得到的结构远比原始网格大。而存储基础网格 M^0 和重构信息得到的结构一般要比 M^n 小一些。这种技术依据边去除的可逆性，把边去除的逆操作称为顶点分裂。由此可得不同细节级别之间所需的重构信息：

- 被分裂的顶点；
- 新顶点的位置；
- 要加入网格的三角形。

渐进式网格的一个重要特点是整个方法不必限于边去除/顶点分裂操作，它可以作为一种结构用于任何局部操作可逆的分解/合成方法。

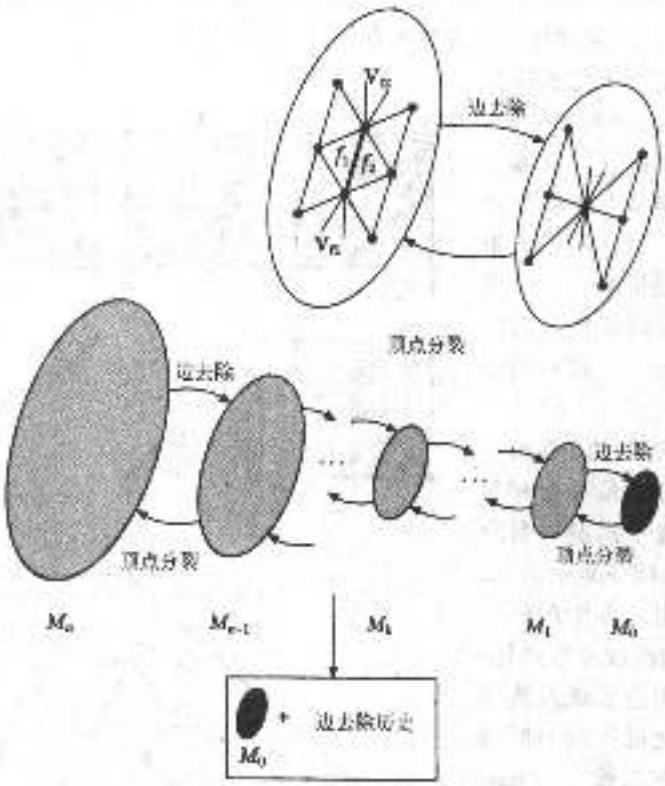


图 6-12 Hoppe 的渐进式网格方法的示例

Hoppe 的简化算法把网格逼真程度描述为一个需最小化的能量函数。并按照网格 M^n 上的部分点集 X 连同随机从表面上采样的一些点，对网格 M 进行优化（虽然这是一个耗时的过程，但只是作为离线预处理执行一次）。需要最小化的能量函数为

$$E(M) = E_{dist}(M) + E_{spring}(M)$$

其中，$E_{dist} = \sum_i d^2(x_i, M)$ 是点 x 到网格 M 距离的平方和。当一个顶点被去掉，它将趋向增大。

⊖ 渐进式网格技术可以在 DirectX 8.0 的 D3DX 库中调用。

$$E_{spring}(M) = \sum \kappa \| \mathbf{v}_i - \mathbf{v}_j \|^2$$

是一个弹性势能公式，用来协助优化。它相当于在每条边上放置了一条静止长度为零，弹性系数为 κ 的弹簧。

Hoppe 通过把所有（合法）的边去除变换放到一个优先队列中来排序优化过程。这里每个变换的次序由它的能量估计 ΔE 决定。在每一步迭代中，排在队列最前的（即拥有最小 ΔE 值的）变换被执行，然后重新计算在此变换邻域内的边的次序。当且仅当不改变网格的拓扑结构时，边去除变换才是合法的。例如若 $\mathbf{v}_i$ 和 $\mathbf{v}_j$ 是边界顶点，则边 $|\mathbf{v}_i, \mathbf{v}_j|$ 必是边界边——它不可能是一条连接两个边界顶点的内部边。

在之后的工作中［HOP97］，Hoppe 描述了如何扩展渐进式网格，使得多边形分辨率可以按照视点相关的标准而变化。他称其为选择性调整（selective refinement）。和许多地形算法类似，此常用方法是从一棵树下降并构造一棵子树，这棵子树是由某些屏幕空间评估标准决定的。和多边形模型一起使用的树被称为合并树或顶点层次。

在渐进式网格的迭代压缩算法进行（例如）边去除操作的同时，自底向上地构造此树（它实际上是一个森林，除非 M^0 是单个顶点）。这棵树的叶子代表 M^n，根代表 M^0。由于边去除操作合并两个顶点为一个，所以该树是二叉树。关于顶点树的一个简单例子如图 6-13 所示。

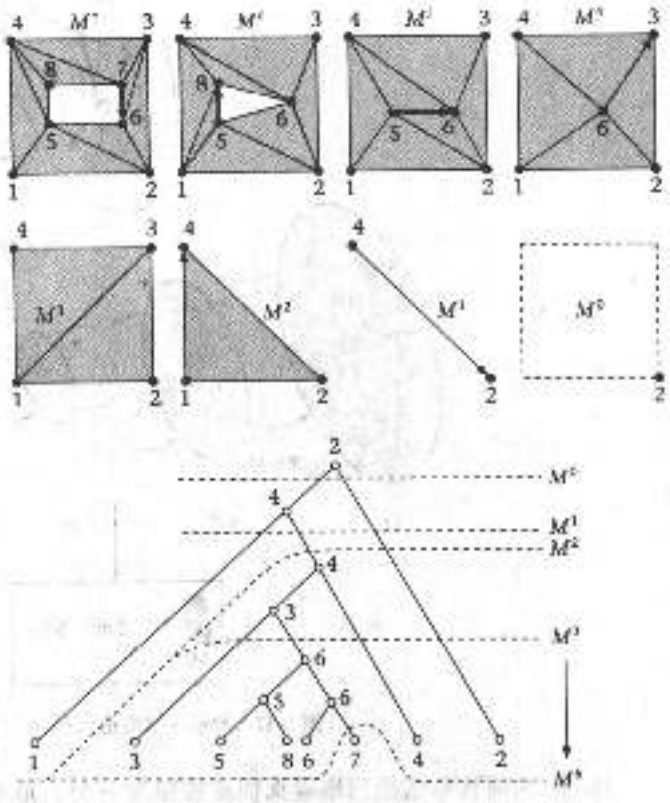


图 6-13　一个简单（改变拓扑）的例子和它的顶点树

通常在渐进式网格的重构阶段，重构出和分解阶段完全相同的模型。而且，这个模型以与分解阶段完全相反的顺序显示出来。我们可以考查简化过程中的一个时刻，来使树的增长过程形象化。在此阶段，顶点树会长成最终树的子集。随着简化过程的不断继续，边去除的“波动线”（wave front）从根向下移动以形成最终的树。如果考虑其相反的过程，则随着波动线在相反方向移动，顶点树会收缩；此相反过程是通过简化过程的历史来构建的（这是渐进式网格的通常用法）。

选择性调整的思想是，我们不需要反转生成过程，而可以按照某些屏幕空间评估标准，生成顶点树对应的任意有效子集。由此，我们可以生成一个有不同多边形分辨率的，符合评估标准的模型。通常情况下，此时生成的模型和任一在简化过程中生成的网格都不同。并且在运行阶段必须同时考虑执行边去除和顶点分裂来满足要求。

在渲染一帧画面之前，算法检查所有活跃顶点，对每个顶点或分裂，或去崇，或不进行任何操作。活跃顶点指的是网格中刚被演变的顶点。由于顶点分裂与边去除都是按一定次序执行的，而这个次序通常和简化过程的顺序并不同，所以在顶点层次中，必须定义相关性来判定被作操作是否合法。Hoppe把它称为前提条件（见图6-14）：

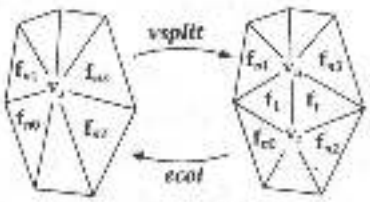


图6-14 Hoppe的渐进式网格的选择性调整中，对顶点分裂与边去除的前提条件的示例

一个顶点分裂是合法的当且仅当

1）v_s 是活跃顶点。

2）面集 $\{f_{l0}, f_{l1}, f_{r1}, f_{r2}\}$ 是活跃面。

一个边去除是合法的当且仅当

1）v_s 和 v_t 是活跃顶点。

2）与 f_l 和 f_r 相邻的面集 $\{f_{a1}, f_{a2}, f_{b1}, f_{b2}\}$ 是活跃面。

Hoppe应用三个因素作为标准，来决定是否应该调整一个节点。这些因素的作用如图6-15所示。

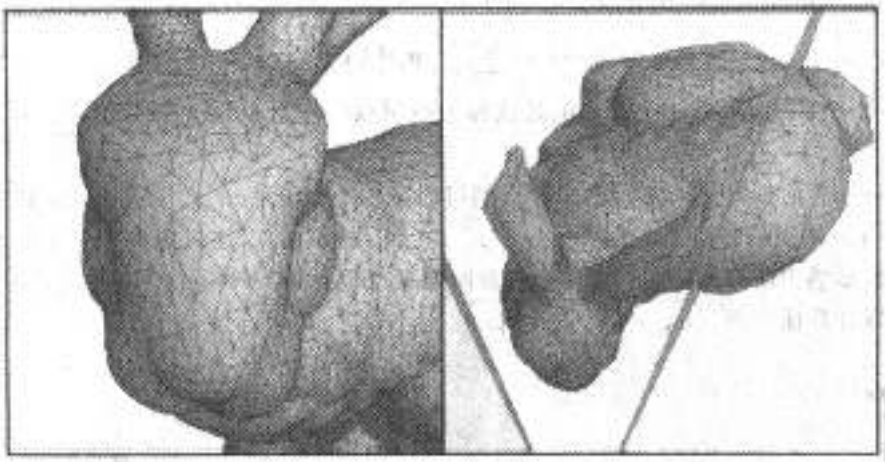


图6-15 Hoppe的视点相关调整。左图是演变视图，右图显示了视见约束体的位置

- **视见约束体**：使用它是为了使视见约束体外的网格变粗糙，从而减少图形负载。（在第2章中，我们介绍了对动态对象进行视见约束体裁剪的一个有效方法。但它保持那些和视见约束体相交的物体在视见约束体外的多边形分辨率不变。）在这个方法中使用一个以当前顶点为球心的，以能够包含它的所有后代区域的长度为球径的包围球。如果这个包围球完全位于视见约束体之外，则顶点不进行细化。
- **表面朝向**：一个类似的方法被应用于背面，以减少图形负载。在这里，为每个顶点及其后代构造了一个表面法向量的包围锥体（在高斯图上）。把它和视点结合起来，用来判定顶点是否位于某个背面区域中。
- **屏幕空间几何误差**：这是一个屏幕空间误差度量。作为LOD评估标准，它有很多不同的变种。在这里，网格被不断细化直到它与原始网格间的误差小于一个屏幕空间

容差。

6.5.2　实例分析 2——使用微分几何

前面的章节已经介绍了操作在三角形网格几何结构上的技术，这里三角形网格是由顶点位置和邻接信息所表示的。然而我们应该记得，一个三角形网格通常是一个光滑（或者至少是局部光滑）的表面的近似。这个概念使我们去考察基本微分几何方法的应用。为了使用微分几何，我们需要一个参数化表示形式。

给出不具有参数化表示形式的三角形网格，问题是我们能否为它构造出一个参数化表示形式？有许多不同方法可以做到这一点，每个方法都会得到不同的结果。显然，可以通过应用一个表面内插过程，把一个网格插值并转变为一个双二次参数面片网。即我们令该表面穿过顶点。原则上，直接进行表面内插是个困难的问题。（一种方法——分散数据插值——将在第 8 章中给出。）在考察参数化方法之前，我们首先要问此类方法的用处是什么？通常的回答是：它能够允许我们使用数学工具和其他很难应用于三角形网格的操作。

除非已有纹理贴图的参数化形式（如 6.5.3 节中所用的），否则我们的表示形式中并没有任何参数化形式。在计算机图形学中，一个物体常见的自然的参数化形式是双三次参数表示形式，它被定义为面片网络，其中的单个贝济埃面片被定义为

$$\mathbf{q}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3}\mathbf{p}_{ij}B_i(u)B_j(v)$$

其中 q 是在物体表面上的一个点；$\mathbf{p}_{ij}$ 是被称为控制点的一组 16 个的固定点集；B 是贝济埃基函数。

拥有一个表面参数化形式，意味着我们可以把一个（u，v）值代入等式，得到一个由参数（u，v）指定的表面上点的（x，y，z）值（见图 6-16）。这样的参数化表示形式解决了计算机图形学中的许多问题。例如，纹理映射变得很容易。我们还能在表面上应用微分几何——可以计算任一点（u，v）的偏导数。

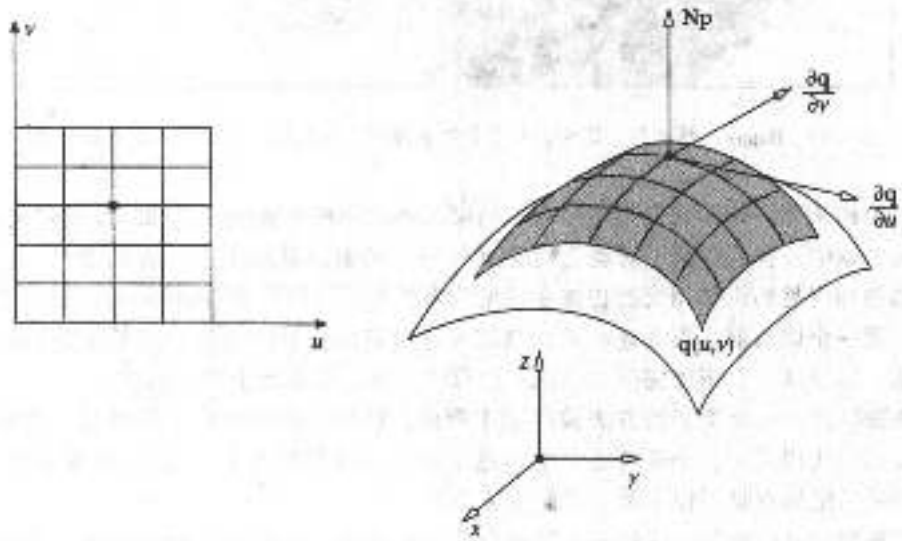


图 6-16　一个在（u，v）上参数化的表面

微分几何算符的二次曲面邻域参数化表示

（本节需和本章最后的附录6.1结合在一起阅读，即数学背景——基础微分几何。）

在6.3.2节，我们看到了如何在网格简化过程中使用一个简单的局部体积不变的误差评估标准。在这一节中，我们将了解如何在邻域上构造二次曲面。近似的三角形网格在这种方式下被广泛使用。这种方法仍然是插值法的一种变化形式，并且满足如下的常用假设：分段线性表面上的顶点都是“实际”表面的样本。由此，我们可以穿过这些点拟合二次曲面，所得结果比三角形网格更接近真实表面。随后使用微分几何为每一个邻域顶点计算表面曲率。通过使用这些信息，可以引入更加“聪明”的重新三角形划分策略。

我们定义双二次多项式如下：

$$f(u,v)=\lambda_1 u^2+\lambda_2 v^2+\lambda_3 u+\lambda_4 uv+\lambda_5 v \tag{6-1}$$

把这个等式与二阶泰勒展开式比较：

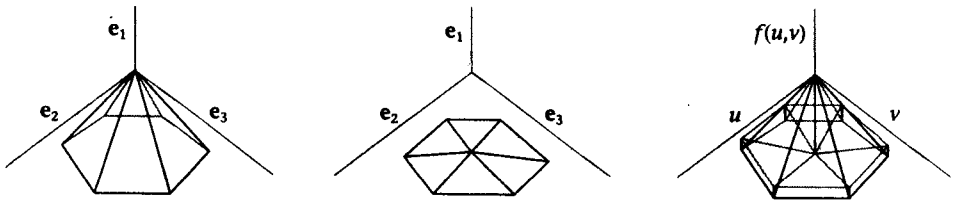
$$f(u,v)=f_u(0,0,)u+f_v(0,0)v+1/2(f_{uu}(0,0)u^2+2f_{uv}(0,0)uv+f_{vv}(0,0)v^2)$$

得到：

$$\lambda_1=2f_{uu}(0,0)$$
$$\lambda_2=2f_{vv}(0,0)$$
$$\lambda_3=f_u(0,0)$$
$$\lambda_4=f_{uv}(0,0)$$
$$\lambda_5=f_v(0,0)$$

为解出λ，我们为邻域构造一个局部参数化表示形式。局部坐标系统由基向量$\mathbf{e}_1$，$\mathbf{e}_2$，$\mathbf{e}_3$给出，其中$\mathbf{e}_1$是顶点法向量。我们把顶点投影到$\mathbf{e}_2$，$\mathbf{e}_3$张成的平面，以得到顶点对应的（u，v）；由距离平面的高度得到对应的f（见图6-17）。由此，顶点位置可在局部被解释为高度场：

$$\mathrm{x}(u,v)=f(u,v)\mathbf{e}_1+u\mathbf{e}_2+v\mathbf{e}_3 \tag{6-2}$$



a) 局部帧中的顶点邻域　b) 邻域在平面$\mathbf{e}_2\wedge\mathbf{e}_3$上的正交投影　c) 投影操作定义了被投影点和相应的$f(u,v)$值

图6-17 顶点邻域参数化的例子

等式6-1可以应用求解非线性系统的牛顿方法的一个变化形式来解决［PRES01］。这个过程的一个例子如图6-18所示。邻域内的边映射为二次曲线，三角形映射为两片二次曲面面片。这个表面是二阶可微的（即C^2，与原始的分段连续的网格C^0相对）。由此就可以确定三角形网格上的偏导数。

应用式6-2，我们得到x_i邻域表面的一阶和二阶偏导：

$$\mathbf{x}_u(0,0)=f_u(0,0)\mathbf{e}_1+\mathbf{e}_2=\lambda_3\mathbf{e}_1+\mathbf{e}_2$$

$$\mathbf{x}_u(0,0) = f_u(0,0)\mathbf{e}_1 + \mathbf{e}_3 = \lambda_1\mathbf{e}_1 + \mathbf{e}_3$$
$$\mathbf{x}_{uu}(0,0) = f_{uu}(0,0)\mathbf{e}_1 \quad = 1/2\lambda_1\mathbf{e}_1$$
$$\mathbf{x}_v(0,0) = f_v(0,0)\mathbf{e}_1 \quad = \lambda_2\mathbf{e}_1$$
$$\mathbf{x}_{vv}(0,0) = f_{vv}(0,0)\mathbf{e}_1 \quad = 1/2\lambda_2\mathbf{e}_1$$

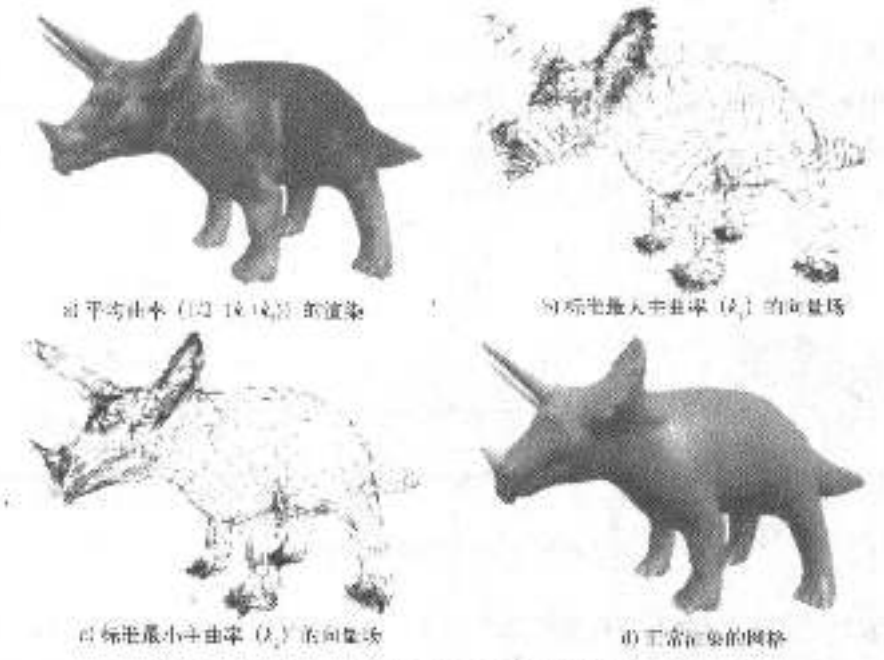
这正是我们所要的结果——对顶点 v 邻域表面的一阶和二阶偏导数。



a) 顶点邻域　b) 由下面顶点邻域正交投影形成的参数域　c) 参数化表示形式的 C^2 最小二乘近似

图 6-18　顶点邻域参数化和 C^2 最小二乘表面的例子

应用这些偏导数，我们很容易就可以在每个网格顶点上定义各种不同的微分几何变量（附录 6.1）。图 6-19（彩页中也有）显示了一个以颜色表示平均曲率的物体。



a) 平均曲率 $(1/2\ (\kappa_1+\kappa_2))$ 的渲染　b) 标准最大主曲率 (κ_1) 的向量场

c) 标准最小主曲率 (κ_2) 的向量场　d) 正常渲染的网格

图 6-19　在三角形龙网格上进行离散微分几何算子的例子

用微分几何进行重新三角形划分

本节我们将考查一个去除顶点后再重新三角形划分的方法。它的基本思路是把重新三角形划分基于在前面章节中定义的二阶偏导的大小和方向。由于这些偏导都是精确的，此方法

能在顶点数较少时生成较高质量的简化。当顶点数变得很少时，简化算法中质量不足的问题就很明显了。可以很容易地修改6.4.4节中的重新三角形划分结构，使它适用于微分几何参数。只要定义 $w(e)$（见6.4.4节）如下：

$$w(e) = \kappa_e^1 \left| (\mathbf{v}_1 - \mathbf{v}_2) \wedge \mathbf{d}_e \right| + (1 - \kappa_e^2) \left| \mathbf{v}_1 - \mathbf{v}_2 \right|$$

$$\kappa_i^1 = \begin{cases} \dfrac{|\kappa_i^1| - |\kappa_i^2|}{\kappa_{\min}} & \text{if } |\kappa_i^1| - |\kappa_i^2| < \kappa_{\min} \\ 1 & \text{else} \end{cases}$$

$$\mathbf{d}_e = \frac{\mathbf{d}_e}{|\mathbf{d}_e|}(e - \kappa_i^1)$$

其中：

κ_i^1 和 κ_i^2 是顶点 i 的主曲率

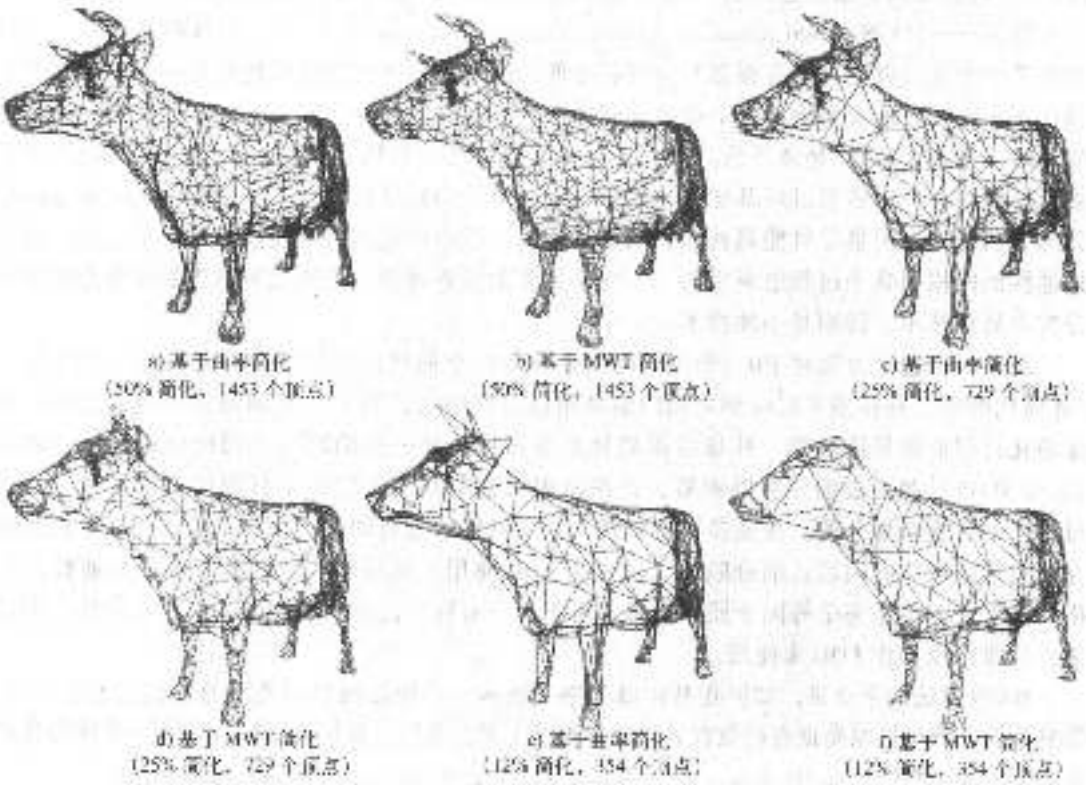
κ_i^2 是在曲率的两个主方向之间的相对曲率

$\kappa_{\min}$（$0 \le \kappa_{\min} \le 1$）是控制曲率敏感度的因子

$\mathbf{d} = \kappa_i^1$ 的方向

对于那些在一个方向上弯曲得很厉害的网格区域，这种增强的方法迫使被划分三角形的边与最小曲率方向对齐。注意，在简化进行过程中，主曲率并没有重新计算，它们只在网格发生改变前计算一次。

图6-20将这种方法与MWT算法进行比较。这两种算法之间的不同在图示的四肢部分是



a) 基于曲率简化
(50%简化，1453个顶点)

b) 基于MWT简化
(50%简化，1453个顶点)

c) 基于曲率简化
(25%简化，729个顶点)

d) 基于MWT简化
(25%简化，729个顶点)

e) 基于曲率简化
(12%简化，354个顶点)

f) 基于MWT简化
(12%简化，354个顶点)

图6-20 基于曲率和基于MWT简化奶牛网格的例子

显而易见的。使用基于微分几何的重新三角形划分得到的三角形，它们和腿的长轴对齐——这正是我们想要的效果。

微分算子的直接微商

在最近的一份报告中，Desbrun 等人［DESB00］介绍了直接操作三角形网格的微商公式。这种方法最显著的优点是公式可以直接应用于网格几何体，而不是像前几节介绍的那样应用于导出或内插的表面。例如，已有公认的计算顶点法向量的方法，它直接计算相邻面法向量的加权平均和。作者称，如果初始三角形网格是原始表面的一个很好的近似，则他们的结果接近于操作在光滑表面上的结果。他们在工作中使用了顶点 v 邻域的空间平均的概念。

首先考虑 Laplace-Beltrami 算子，它把点 $\mathbf{p}$ 变换为向量 $\mathbf{K}(\mathbf{p})$，即平均法曲率：

$$\mathbf{K}(\mathbf{p}) = 2\kappa_p \mathbf{N}_p = \lim_{A \to 0} \frac{\nabla A}{A}$$

其中，κ_p 是平均曲率，A 是包含 $\mathbf{p}$ 的无穷小区域，∇A 是 $\mathbf{p}$ 点的梯度。

（注意把平均法曲率表达为极限和高斯曲率极限表达式之间的相似之处。）

6.5.3　实例分析 3——网格重新划分算法 MAPS

在这一节中，我们将看到一类新的处理多边形网格的技术，称为网格重新划分算法［ECK95］，［KOBB98］和［LEE98］。这些算法之所以被为网格重新划分算法，是因为它们把任意连通性的三角形网格（由某些硬件（比如 3D 数字化设备）产生的，或者由建模软件输出的），转换成具有细分连通性（subdivision connectivity）的网格。我们将介绍这一领域中的一个算法——MAPS（Multi-resolution Adaptive Parameterization of Surfaces）。之所以选择它，是因为它是一个比较典型并且在很多其他领域中使用的算法。它的主要目的是为一个并不存在参数化表示的 2 流形模型构造一个参数化表示形式。这样的参数化形式，在比如使用法向量图渲染时（见第 4 章），是必须的。一个具有细分连通性的网格指的是一个与原先的网格非常接近的网格，但它是通过对基础网格进行细分并且移动最终的分段规则的网格上的顶点而得到的，所以这个网格尽可能逼近原始网格。由此，任意连通网格被转换成具有细分或半规则连通性的网格。这个过程结束后的一个立竿见影的优点就是，它现在可以更加充分地使用多分辨率处理技术，特别是小波技术。

现在，把这个方法和 PM（渐进式网格）技术作个简单比较可能会非常有用（图 6-21）。PM 简化网格，并存储不同级别之间的信息用以重构网格。这个从基础网格重构的过程，只是简化过程的简单逆过程，并最终得到和原始网格一模一样的网格。而网格重新划分算法（比如 MAPS）是通过细分基础网格，并在原先网格的几何结构中重新取样以扰动顶点，来对网格进行重新划分的。按照参数化形式，原始网格的重新划分就变成了确定“一个点是否在三角形区域”的问题。细分阶段（可作为 LOD 使用）的局限，就是具有细分连通性的网格只是逼近而不是完全等同于原始网格。由于细分可以在渲染物体上的任何一点停止，所以细分阶段可以当作 LOD 来使用。

MAPS 算法的重点是，如何在基础域网格上重构一个原始网格的光滑的参数化表示形式。简单地说，就是把原先顶点的位置表达成基本域上的三角形。我们将看到，这样一个参数化形式有着多种应用，包括 3D 变形、多分辨率可编辑的自适应简化、传输以及 LOD 渲染。

虽然 Lee等人不是最先使用这种方法的研究者（参见［ECK95］），但他们的算法具有简

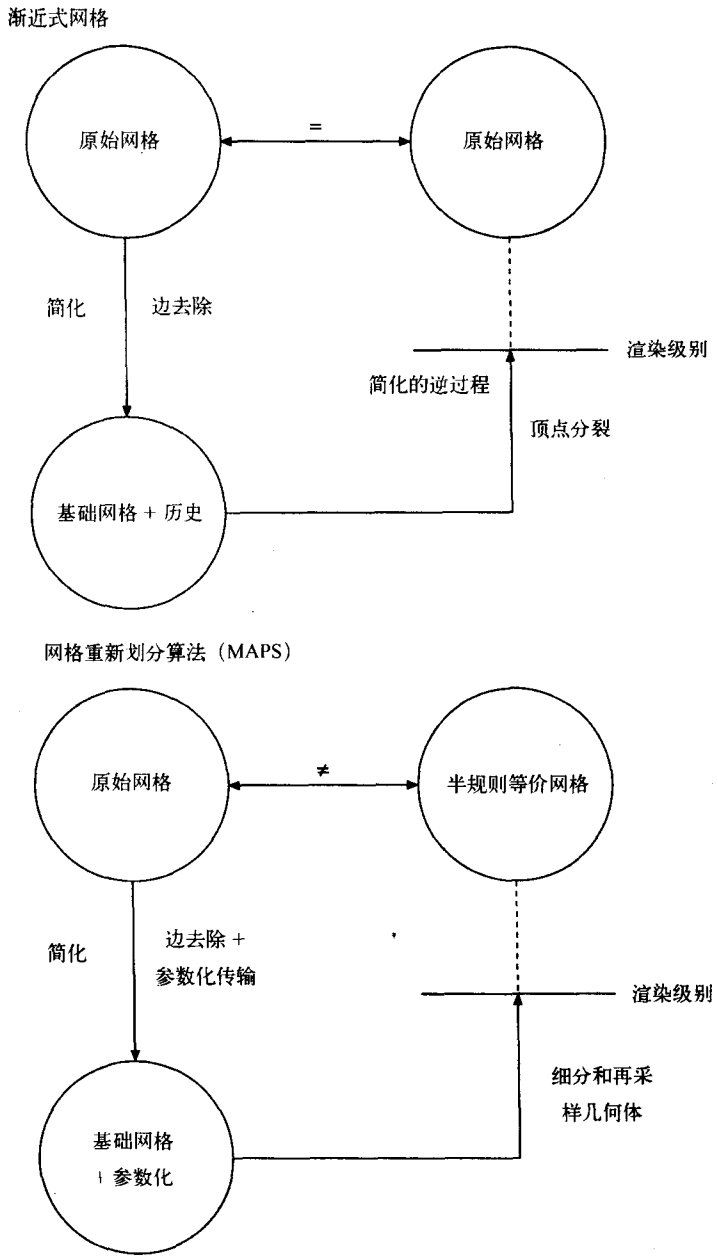


图 6-21 比较渐进式网格和网格重新划分算法的示意图

单和比大多数其他方法都易于实现的优点[⊖]。它还有一个优点，即模型上需要保留的特征线，或者“实”边，能在基础域中被保留下来。

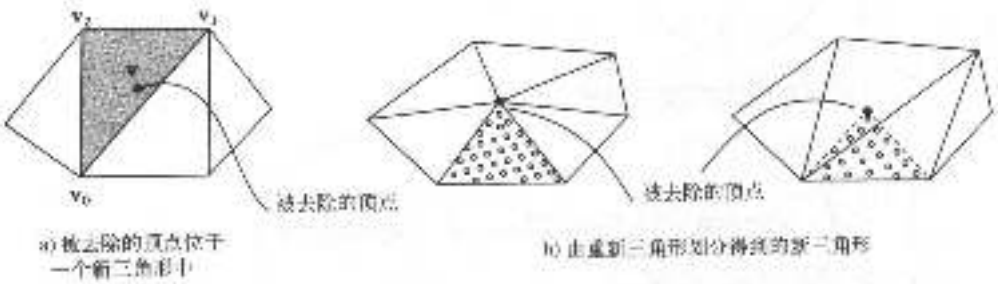
总之，该算法分成两种不同的阶段——简化阶段和网格重新划分阶段。Lee 等人在[LEE98]中使用了顶点移除，而在[LEE00]中用了边去除，加以Garland的二次误差度量

⊖ 一个 MAPS 的简化版本和代码架构在［LEE00］中给出。

（见6.3.3节）。无论使用哪种操作，都需要重新划分三角形。这通过使用在2D空间中受限的Delaunay三角形划分（或者CDT）来实现。这里CDT是DT的一个特例，它包含（新的）边界边作为最终三角形划分的一部分。一般DT是一个惟一或可判定的三角形划分，它同时对几个有关划分的质量指标（比如最小角）进行优化。

正如6.4.4节中所介绍的，在三维空间中的重新三角形划分是通过用保角映射 X^c（附录6.1）把被移除顶点的邻域映射到平面上来实现的。如前所述，此过程的目的是获得一个原始顶点在基础域上的参数化表示形式。随着简化过程的进行，我们对位于原始网格和基础域网格之间的中间网格向下计算参数化形式。

现在来具体考虑一下参数化形式是如何向下计算的。考察顶点v的移除（见图6-22a）。被去除顶点现在包含在由重新三角形划分所得的一个新（阴影）三角形中。把v用关于阴影三角形的重心坐标来表示（见附录6.1中重心坐标的定义）。即我们计算一个4元组 $(\alpha, \beta, \gamma, T)$，其中 (α, β, γ) 是v关于三角形T的重心坐标。这里T是v的相关三角形。



a）被去除的顶点位于一个新三角形中　　b）由重新三角形划分得到的新三角形

图6-22　在前面简化中被除去的重新参数化顶点

除了考虑刚被移除的顶点之外，还必须考虑在之前简化过程中被参数化的，即有关于那些已被删除三角形的重心坐标的顶点。需重新计算它们关于当前三角形的重心坐标。由此，所有原始顶点的参数化形式在简化过程中向下计算。最终我们得到了基础域，在基础域上所有的原始顶点是基础域三角形的参数化表示形式。顶点被有效地展平到基础域表面上。此过程结束时我们得到了：

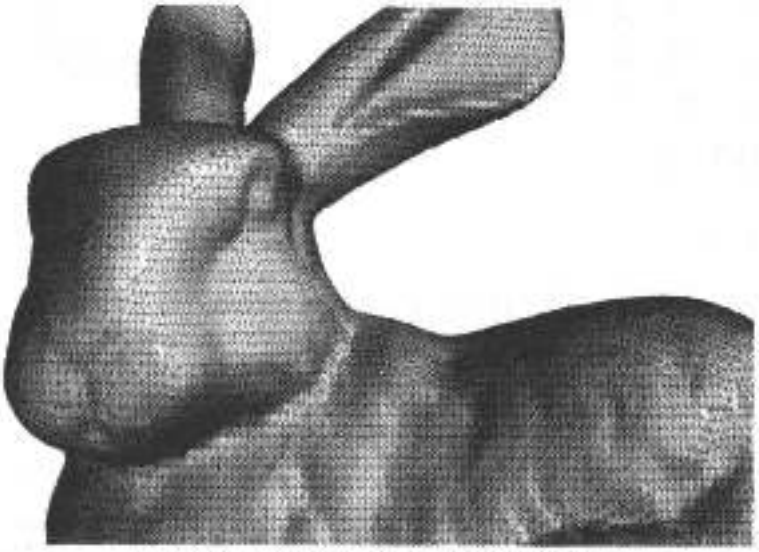
1）一个低分辨率或基础域三角形网格。

2）一个为每个原始顶点标明在基础域中所包含的三角形，以及关于此三角形的重心坐标的4元组 $(\alpha, \beta, \gamma, T)$。

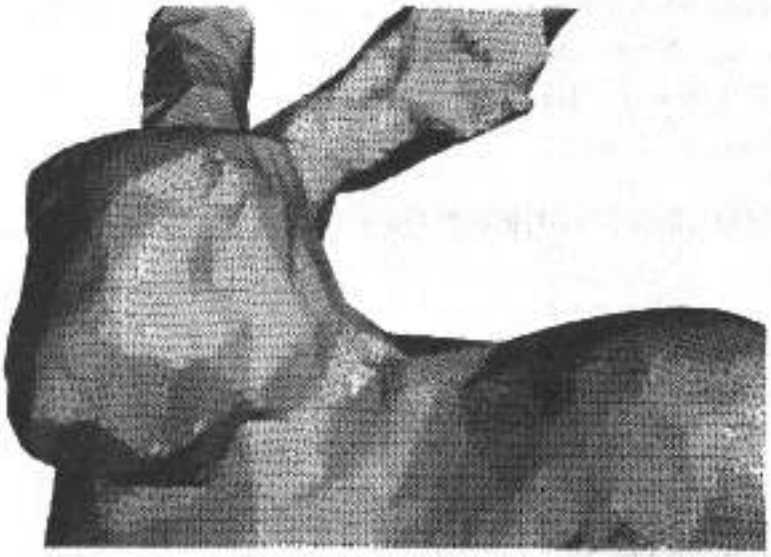
3）每个顶点的原始 (x, y, z) 坐标。

图6-23（彩页中也有）中有形象的示例。

为了使网格能重新划分到任何（细分）级别，我们按如下步骤进行。首先，在基础域网格三角形上，使用一个简单的四分法进行细分。这将产生位于基础域三角形平面上的细分顶点。现在，这些顶点需被放到一个新的位置，使它们分布在原始模型的表面上。这个过程的第一步，是在被展平的基础域网格中寻找包含这些顶点的原始网格三角形 $\mathbf{v}_i$。注意 $\mathbf{v}_i$ 已经被包含在一个基础域三角形平面中，而寻找此三角形是一个二维问题。在[GUIB85]中介绍了一个合适的算法。这是一个“行走”方法，从一条随机的边开始，然后一次朝 $\mathbf{v}_i$ 的



a) 原始网格（Stanford Bunny 的一个低分辨率版本）



b) 基于映射到顶点的参数化形式

图 6-23

一般方向移动一条边[1]。一旦发现了包含三角形，我们就找到了顶点 $\mathbf{v}_i$ 关于此三角形的重心坐标 (α, β, γ)。为了在原始表面上重新采样及随后把 $\mathbf{v}_i$ 放置在原始表面上，我们用：

$$\mathbf{v} = \alpha \mathbf{v}_a + \beta \mathbf{v}_b + \gamma \mathbf{v}_c$$

其中，$\mathbf{v}_a$、$\mathbf{v}_b$ 和 $\mathbf{v}_c$ 是原始顶点，它们对应于位于基础域三角形中的被展平三角形的顶点。

[1] 注意，此算法假设单有效的实现是优先使用一个数据结构。在这个结构中，网格中的边是有向的，并保存指向标始顶点、目标顶点的指针，指向目标顶点的上一条边以及指向起始顶点的下一条边的指针。这一条边构成一个三角形。

Lee 等人在网格重新划分阶段中调用了另一种强制过程。之所以这么做的原因是：参数化形式虽然在每个基础域三角形中是光滑的，但是在不同的基础域三角形之间不是全局光滑的。为了解决这个问题，他们采用了一种基于 Loop 细分的光滑方法。

附录 6.1　数学背景

(1) 基础微分几何

本节的目的是介绍微分几何的概念，使我们能够定量地刻画表面在一个点 **p** 的邻域内的行为。我们从最一般的情形来讨论，计算表面以多快的速度离开位于 **p** 点的切平面——这是微分几何得名的原因。这是通过单位法向量场在点 **p** 邻域内的变化率来衡量的，我们称之为线性映射 dN。

我们从介绍一些表面曲率的定义开始，后面会介绍如何计算它们。

法曲率

在方向 $\mathbf{e}_\theta$ 上的法曲率 $\kappa^{\mathrm{N}}(\theta)$ 是在表面上，并包含在 $\mathbf{e}_\theta$ 和 **N** 构成的平面里的曲线 C 的曲率（见图 6-24）。κ_1 和 κ_2 是 **p** 点上所有法曲率的极值，它们分别位于正交方向 $\mathbf{e}_1$ 和 $\mathbf{e}_2$ 上。我们称之为最大法曲率和最小法曲率。

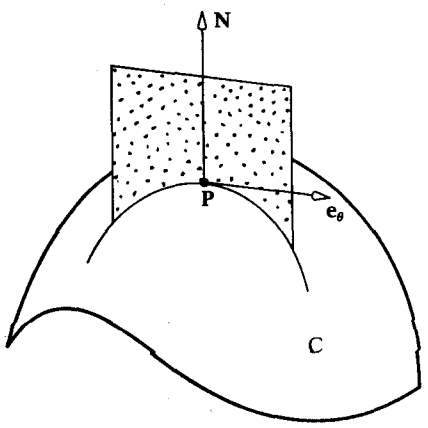


图 6-24　法曲率 $\kappa^{\mathrm{N}}(\theta)$ 是 **N** 与 $\mathbf{e}_\theta$ 所构成平面上曲线的曲率

平均曲率

平均曲率是所有法曲率的平均值或者主曲率之和的 1/2：

$$1/2(\kappa_1 + \kappa_2)$$

高斯曲率

高斯曲率定义如下：

$$\kappa_{\mathrm{G}} = \lim_{s \to 0} \frac{A}{s} = \frac{\mathrm{d}A}{\mathrm{d}s} = \kappa_1 \kappa_2 = |\mathrm{d}\mathbf{N}_p|$$

其中：

s 是表面上的无限小区域；

A 是高斯图；

$d\mathbf{N}_p$ 被称作线性映射，是我们随后将导出的一个微分。

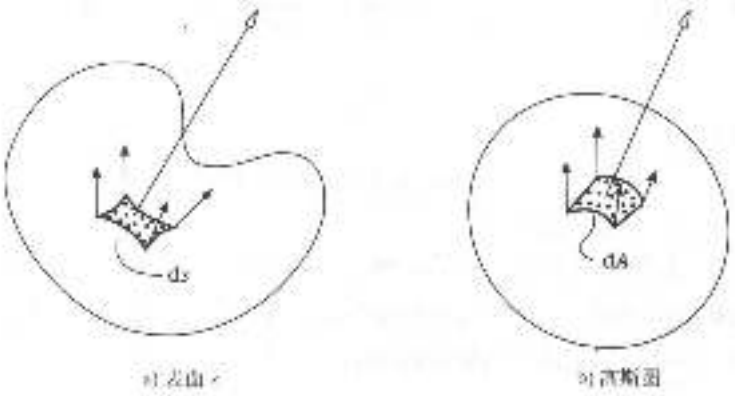
高斯图（见图 6-25b）是 s 的法向量在一个单位球上所构成的图——表面上的法向量变成了球上的点。由此可得平面面片的高斯曲率是 0，因为它的图是一个单点，而球面的高斯曲率是 1。

我们定义以 $\mathbf{x}(u,v)$ 为参数的表面为（见图 6-16）：

$$\mathbf{x}(u,v) = (x(u,v), y(u,v), z(u,v))$$

如果表面是可微的，可以定义：

$$\frac{\partial \mathbf{x}}{\partial u} = \mathbf{x}_u \text{ 和} \frac{\partial \mathbf{x}}{\partial v} = \mathbf{x}_v$$



a) 表面 S　　b) 高斯图

图 6-25　高斯曲率计算的定义

以及包含这些向量的切平面。切平面是由位于 x 的单位法向量所指定的：

$$\mathbf{N}(u,v)=\frac{\mathbf{x}_u\times\mathbf{x}_v}{|\mathbf{x}_u\times\mathbf{x}_v|}$$

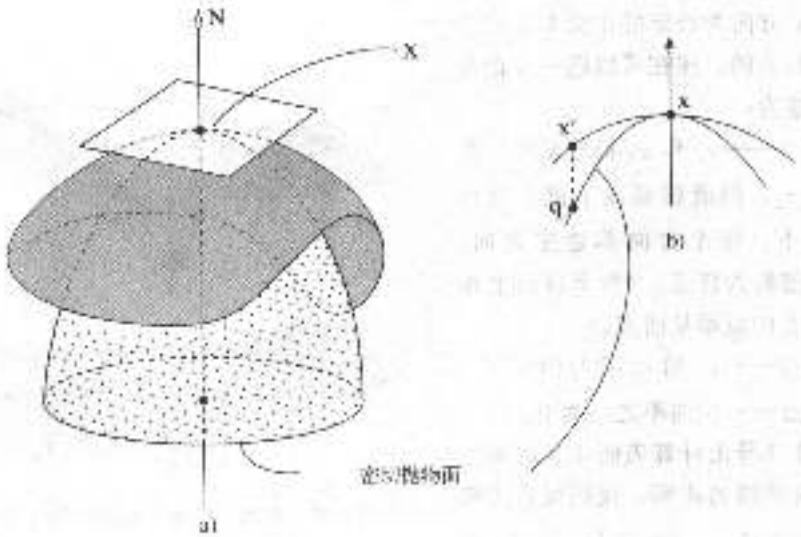
和把一阶导数表示为一个平面类似，二阶导数是一个二次曲面——即所谓的密切抛物面（见图 6-26）。如果表面是二次可微的（C^2），那么这样的抛物面一定存在。参见图 6-26b。如果我们定义 x′为一个靠近 x 的点，点 q 是一条穿过 x′与抛物线轴平行的直线与该物面的交点，则进一步定义：

$$d=|\mathbf{x}-\mathbf{x}'|$$
$$h=|\mathbf{q}-\mathbf{x}'|$$

若：

$$h/d^2\to 0\quad \text{as } \mathbf{x}'\to\mathbf{x}$$

就称此抛物面为密切抛物面。



a)

图 6-26　曲面上点 x 的密切抛物面

如果我们引入一个 (u, v) 平面与切平面重合的坐标系统，那么在 x 的邻域内可以把表面表示成：

$$z = f(u, v)$$

切平面是由下式给出的：

$$z = uf_u(0,0) + vf_v(0,0)$$

函数在这个邻域内的泰勒展开式是：

$$f(u,v) = f_u(0,0)u + f_v(0,0)v + 1/2(f_{uu}(0,0)u^2 + f_{uv}(0,0)uv + f_{vv}(0,0)v^2)$$

这个讨论使我们认识到了一个显而易见的结论：对表面上 x 的足够小的邻域，一次近似是由平面给出的，二次近似是由密切抛物面给出的。根据表面的形状，密切抛物面将取下面 4 种形式之一：

- 椭圆点——密切抛物面是椭圆形的；
- 双曲点——密切抛物面是双曲线；
- 抛物点——密切抛物面是一个抛物柱面；
- 平面点——密切抛物面退化成平面。

第二种情况如图 6-27 所示。当我们从 x 沿负法线方向向下移动切平面时，该平面将在密切抛物面上与椭圆、双曲线、或直线相交。即在表面上 x 的很小邻域内，我们将得到相同的交叉区域。换句话说，密切抛物面在极限上收敛于表面。

所有上述讨论都引向表面曲率的定量度量。在继续之前，我们先观察一下刚刚给出的几个分类。我们假定表面上一点的曲率是由最大法曲率 κ_1 和最小法曲率 κ_2 给出的——即 p 点的主曲率。这些曲率和那些在 p 点上方向为给定的正交主方向的表面曲线是相关的。现在可以进一步把我们的分类表述为：

- 椭圆点——κ_1 和 κ_2 符号相同。若 $\kappa_1 = \kappa_2$，则椭圆变成了圆。这种情况下，每个方向都是主方向，该点被称为脐点。（所有球面上和平面上的点都是脐点。）
- 双曲点——κ_1 和 κ_2 符号相反。
- 抛物点——主曲率之一为 0。

我们现在来导出计算表面曲率的表达式。为了计算曲线的曲率，我们观察法向量关于曲线的变化率。相似地，对一个表面来说，我们可以观察在 p 点邻域内的法

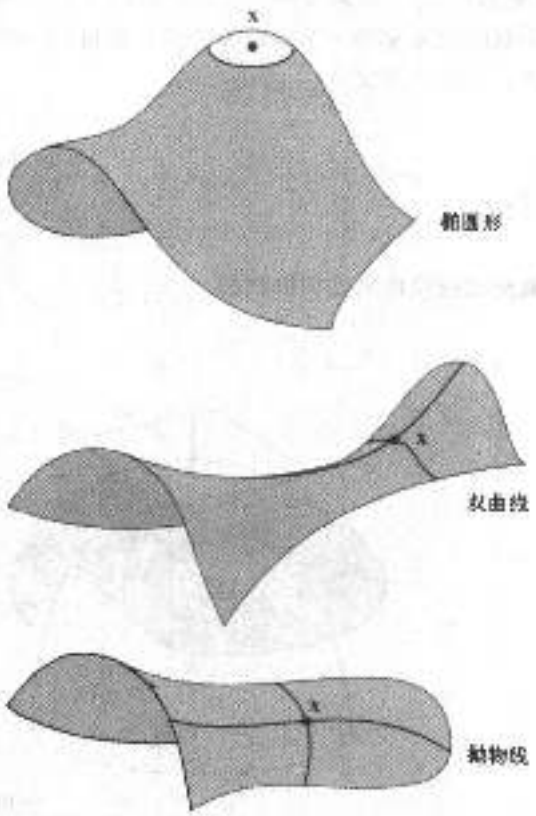


图 6-27　椭圆、双曲线和抛物线的部分。曲面在某一点的形状可以是椭圆、双曲线或抛物线的密切抛物面

向量的变化。首先考虑我们感兴趣的表面上穿过 **p** 点的所有曲线 C（见图 6-28）。每条曲线都能表示成位于表面上的一条参数曲线：

$$\mathbf{c}(t) = \mathrm{x}(u(t), v(t))$$

其中$\mathbf{c}(0) = \mathbf{p}$。

这条曲线的切向量为：

$$\dot{\mathbf{c}}(t) = \mathbf{x}_u \dot{u} + \mathbf{x}_v \dot{v}$$

它衡量了在这条曲线上法向量对于曲面的变化率。所以我们可得：

$$\mathrm{d}\mathbf{N}(\dot{\mathbf{c}}) = \mathbf{N}_u \dot{u} + \mathbf{N}_v \dot{v}$$

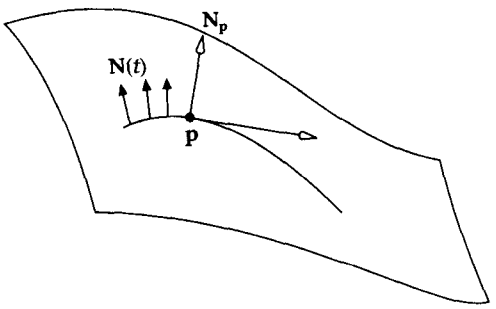


图 6-28 法向量对于位于曲面上的一条曲线的变化率

又由于 $\mathbf{N}_u$ 和 $\mathbf{N}_v$ 在曲面的切平面上，因此得到：

$$\mathbf{N}_u = a_{11}\mathbf{x}_u + a_{21}\mathbf{x}_v$$

$$\mathbf{N}_v = a_{12}\mathbf{x}_u + a_{22}\mathbf{x}_v$$

和

$$\mathrm{d}\mathbf{N}(\dot{\mathbf{c}}) = (a_{11}\dot{u} + a_{12}\dot{v})\mathbf{x}_u + (a_{21}\dot{u} + a_{22}\dot{v})\mathbf{x}_v$$

由此，基（$\mathbf{x}_u$，$\mathbf{x}_v$）d**N** 就由矩阵 a_{ij} 确定下来。

$\mathrm{d}\mathbf{N}_p$ 是高斯图的微分。它的行列式的值是高斯曲率 k。$\mathrm{d}\mathbf{N}_p$ 的负半部分轨迹是 **p** 点的平均曲率 H。回到先前的表面分类，可得：

- 表面在 **p** 点是椭圆，当$|\mathrm{d}\mathbf{N}_p| > 0$
- 表面在 **p** 点是双曲线，当$|\mathrm{d}\mathbf{N}_p| < 0$
- 表面在 **p** 点是抛物线，当$|\mathrm{d}\mathbf{N}_p| = 0$
- 表面在 **p** 点是平面，当 $\mathrm{d}\mathbf{N}_p = 0$

由此可以看出（例如［DOCA76］）：

$$a_{11} = \frac{fF - eG}{EG - f^2} \qquad a_{12} = \frac{gF - fG}{EG - f^2}$$

$$a_{21} = \frac{eF - fE}{EG - f^2} \qquad a_{22} = \frac{fF - gE}{EG - f^2}$$

其中

$$e = \mathbf{x}_{uu} \cdot \mathbf{N}_p \quad f = \mathbf{x}_{uv} \cdot \mathbf{N}_p \quad g = \mathbf{x}_{vv} \cdot \mathbf{N}_p$$

$$E = \mathbf{x}_u \cdot \mathbf{x}_u \quad F = \mathbf{x}_u \cdot \mathbf{x}_v \quad G = \mathbf{x}_v \cdot \mathbf{x}_v$$

$\mathrm{d}\mathbf{N}_p$ 的特征值给了我们所需的曲率定义：

平均曲率 $$H = \frac{1}{2}\frac{eG - 2fF + gE}{EG - F^2}$$

高斯曲率 $$\kappa_G = \frac{eg - f^2}{EG - F^2}$$

主曲率 $$\kappa_i = H \pm \sqrt{H^2 - K}$$

$\mathrm{d}\mathbf{N}_p$ 的特征向量 $\mathbf{e}_1$ 和 $\mathbf{e}_2$ 给出了主曲率的方向。

实例

考虑一个环面，它是通过把一个半径为 r 的圆以 a 为到轴心的距离扫过而得的。它可以被参数化表示为（见图 6-29）：

$$\mathbf{x}(u,v)=((r\cos u+a)\cos v,(r\cos u+a)\sin v,r\sin u)$$

$$0\leq u\leq 2\pi \quad 0\leq v\leq 2\pi$$

从中可得

$$\mathbf{x}_u=(-r\sin u\cos v,-r\sin u\sin v,r\cos u)$$

$$\mathbf{x}_v=(-(a+r\cos u)\sin v,(a+r\cos u)\cos v,0)$$

$$\mathbf{x}_{uu}=(-r\cos u\cos v,-r\cos u\sin v,-r\sin u)$$

$$\mathbf{x}_{vv}=(r\sin u\sin v,-r\sin u\cos v,0)$$

$$\mathbf{x}_{uv}=(-(a+r\cos u)\cos v,-(a+r\cos u)\sin v,0)$$

因此

$$E=r^2 \quad F=0 \quad G=(a+r\cos u)^2$$

等等。

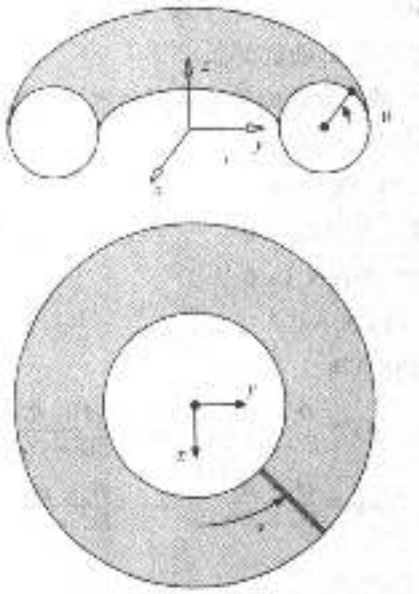


图 6-29　一个环面的参数化形式

(2) 保角映射 z^n

定义保角映射 z^n 为：

$$f(z)=w=z^n$$

这里 z 和 w 都是复数：

$$z=x+iy \qquad w=u+iv$$

举个例子，考虑映射：

$$w = z^2$$

可得到：

$$w = x^2 - y^2 + \mathrm{i}2xy$$

或者

$$u = x^2 - y^2 \qquad v = 2xy$$

在极坐标中我们可得：

$$z = r\ \exp(\mathrm{i}\theta)$$

$$w = z^2 = r^2\exp(2\mathrm{i}\theta)$$

它把 z 平面的第一象限（$0 \leqslant \theta \leqslant \pi/2$，$r \geqslant 0$）映射到整个 w 半平面。每个映射区域中的点惟一对应一个原区域中的点。在 w 平面内半径为 r 的圆，被变换为 w 平面内半径为 r^2 的圆。

现在考虑

$$w = z^a$$

它把一个角区域

$$0 \leqslant \theta \leqslant 2\pi/a,\ r \geqslant 0$$

映射到了整个 w 平面。

正是这个映射，或者说它的分段线性版本，是我们在 6.4.4 节中所用到的。

(3) 重心坐标

重心坐标（α，β，γ）表达了在三角形所在平面上的 **p** 点，相对于三个顶点 p_1、p_2、p_3 的位置：

$$\mathbf{p} = \alpha p_1 + \beta p_2 + \gamma p_3$$

$$\alpha + \beta + \gamma = 1$$

这可以解释为三个区域的相对权重：

$$\alpha = \frac{\Delta p p_0 p_1}{\Delta p_0 p_1 p_2} \qquad \beta = \frac{\Delta p p_1 p_2}{\Delta p_0 p_1 p_2} \qquad \gamma = \frac{\Delta p p_2 p_0}{\Delta p_0 p_1 p_2}$$

在我们的应用中，因为 p，p_0，p_1，p_2 都是已知的，所以可以确定 α、β 和 γ。

附录 6.2 演示

ProgMesh.exe 是一个包含简单边去除算法的渲染程序，细节详见［WATT01］。我们先导入一个合适的模型（比如 kagaroo.3ds）。再按下 detail level 的图标，并拖动窗口中的滑动条来执行边去除。注意这个模型最终是怎么分解的——在本算法里没有使用拓扑检查。

边去除可以通过用简单的启发式方法，或者用一个更严密的，衡量近似网格与原始网格的一个样本之间差异的方法来完成（我们在本章中已经详细说明了这一点）。可使用一个简单的度量来排序去除的边：

$$\frac{|\mathbf{V}_{f1} - \mathbf{V}_{f2}|}{|\mathbf{N}_{f1} \cdot \mathbf{N}_{f2}|}$$

即边的长度除以顶点法向量的点积。这个度量很有效，但一旦连续地应用它，网格将突然开始塌陷。因此使用一个更加成熟的方法来进行边选择将是非常必要的。

第三部分 动画制作

第7章 角色动画

7.1 简介

如前言所述，利用3D计算机图形学技术完成角色的动画制作是可行的。一些标准长度的影视作品就成功地运用了此项技术进行角色动画制作，Pixar公司的《玩具总动员》就是一个很好的例证。在这些影视作品中，角色动画问题得到了解决。然而，在游戏产品中，角色动画却是一个棘手的问题：我们必须在客户端实时地生成角色的动画，而且游戏的逻辑设定也要求为角色生成连续性动作序列，而这些又可用来控制一个自主性角色与游戏环境中其他角色或者玩家之间的相互作用。计算机动画绘图技术可以对单个动画序列进行细致的制作和调整。尽管高质量的实时渲染所面临的一些困难已经逐渐得到解决，但另一个问题，即非线性动画制作，则更加难以解决。

游戏中的角色动画常常被当作一个多层次的问题来处理，有鉴于此，我们将此项任务分解为几个子问题来处理。按本书写作时（2002年）的观点，任务分解后的层次如下：

低层次的几何控制

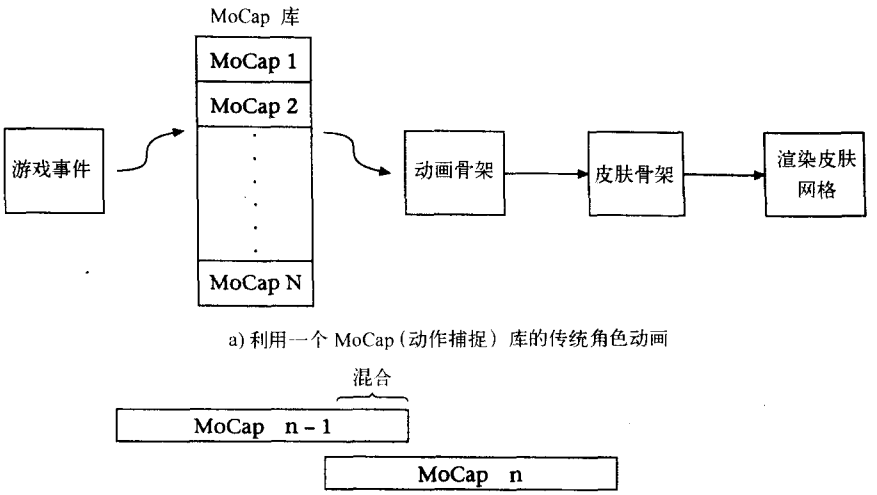
在最底层，动画制作要求必须生成送往GPU渲染的三角形。就在不久之前，角色是由一个低层次的多边形计数器来模拟的（见图7-1），而动画则是从顶点变化形成的关键帧生成的。本章将阐明，现今的模型拥有适合动画制作的骨架，骨架组成又与表面皮肤相联系；从而角色的动作（旋转和取向性）和姿势（适合角度）就随着骨架的运动提供给骨架和皮肤。这种骨架和皮肤相联系的方式决定了皮肤的变形动画制作技术。本章中，我们将主要研究基本的蒙皮操作，而在第8章中将详细研究变形动画技术中的各项要素。

专业变形动画技术

基本蒙皮技术中存在很多缺憾和问题，这就引发了更多控制皮肤变形的专业技术的出现。这些技术独立于一般的骨架运动，拥有自身的动画脚本，最好的例子是我们在第9章中提到的模拟说话时面部运动的动画。

骨架的动作控制

当今最流行的骨架动画制作脚本形式是MoCap技术（见第10章）。它拥有无与伦比的优势，因为它可以提供（或多或少的）易于获得而又令人信服的动画。但它也有局限性——任何没有被预先制作的动画都不能在游戏中调用。连同基本的动作控制一起，我们要求获得能够将一组动作序列与另一组混合的方法，这就是共通的游戏模型。首先存储一个动作脚本的汇总，当游戏逻辑选择了当前要求的一个脚本时，我们对当前动画定格并将下一个脚本与它混合（见图7-1）。本章中，我们研究动作脚本是如何应用于骨架的，更多关于MoCap技术的深入探讨将在第10章中进行。



a) 利用一个 MoCap（动作捕捉）库的传统角色动画

b) 混合相连的动作捕捉序列，生成连续性的动作序列

图 7-1　角色动画和动作捕捉

除了有关序列混合方面的明显要求外，还有其他对于 MoCap 技术的要求。这些要求大多与 MoCap 技术的适应性相关：如果要求角色捡起游戏环境中一组对象中的一个，我们应当为每个对象制作 MoCap 技术，还是设法使生成的动作序列能够适应于一般对象的处理，例如“抓住一个对象”？

MoCap 技术并不是控制关节结构的惟一方法，利用动力学原理模拟游戏中的刚体也是可行的，并且目前已在此方面获得了一些进展；但是用动力学模拟有关节的图像是很困难的。无论用何种方法模拟一个虚拟的人都是十分困难的，动力学模拟也不例外。即使是模拟有规律的走圈这样基本的动作也很难：这样的动作是由关节的旋转组成的，臀部上下移动并且左右摆动，更何况，动作必须要保证躯干的平衡性。在动力学模拟中，这个动作的实现必须借助于为关节提供实时变化的扭矩函数，通过函数计算出增加或者减少多少力量以及该时刻肢体的惯性矩。

在短期内，虚拟人体的动力学模拟还不能在动画制作的逼真性上取代经过良好调谐的运动学技术（包括动作捕捉技术）。从其自身而言，这项技术的制作人员未必能比熟练的漫画家或者负责动作捕捉的技术人员提供更加高质量的动画。然而，预测在未来动画制作中使用的高层次行为功能模拟中，这项技术可能成为最好的工具。运动学技术由于普遍使用了预置脚本或者经过预先制作，所以所有的动画制作都局限在预先计划的范围内。如果环境受到更高层次的进程控制，可能要求角色做出“未出现过”的动作，那么，行为功能模拟将是能完成这一任务的方法。行为功能模拟可能凭借某种语言来指定底层进程在最低层次上为人物进行完全的特写，就好像解释程序一样生成最终要求的动作。“X 应当伸展着手沿圆弧跑向 Y”可以作为一个控制两个代理相互作用的进程发出要求的例子。虚拟人体的动力学模拟不仅是高质量动作的发展而且也是“一般对象”模型的实现。虚拟人体应当能够完成任何真人人体能够做到的动作。角色的动力学模拟已经超越了本书讨论的范围——它现在尚未像笔者意识的那样被应用到游戏软件中。

低层次动画的构成

我们可以把前面的讨论看作对单个角色在不受约束的环境下从一个位置移动到另一个位置的处理。而在一个游戏软件中，动画的精确特性是由各个方面的要素共同构成的：

- 路径规划可能是其中最先要解决的问题：假定一个角色从起点运动到终点，精确的路径是怎样的？在一个包含复杂结构的游戏环境中，这并非一个微不足道的问题。怎样形成路径特性？一般而言，我们倾向于一条"自然"的路径——即曲线路径，而

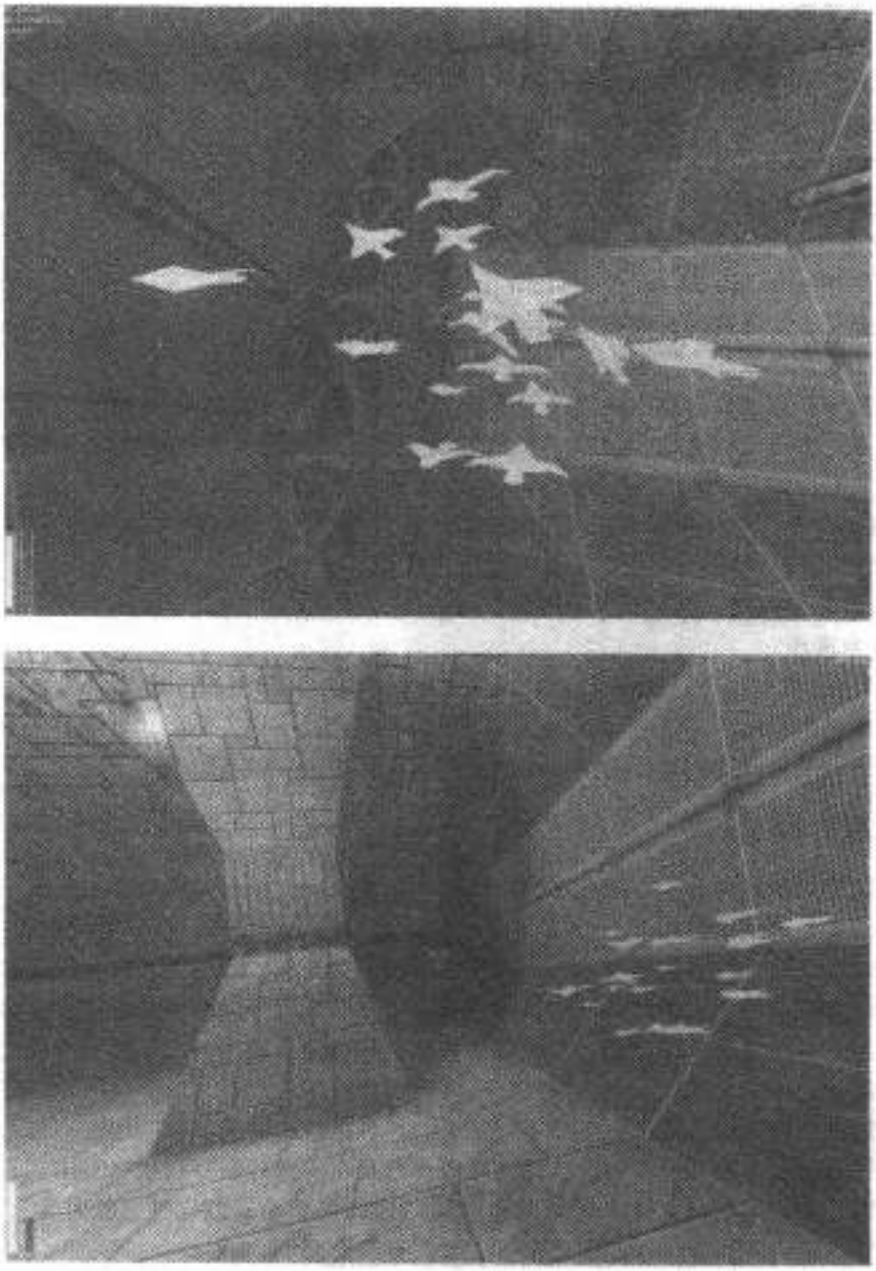


图 7-2 在游戏中应用群体模拟的两个视图

不是由精确的拐角构成的路径。我们只要将路径提供给骨架的根节点就可以实现骨架的全局转换。

- 角色需要被赋予避开障碍物的能力，通常包括碰撞检测能力和使用一些绕开障碍物的策略的能力。
- 角色需要和其他对象发生相互作用：它可以扭动把手打开门，可以捡起一个物体。角色与对象的相互作用使动画制作中的组织和动作的适应能力变得必要。因为对象是无生命的，由此我们将角色/对象间的交互置于这一层次，而不是更高层。

高层次控制

这一方面与高层次脚本的制作有关，这些脚本用于详细说明角色的动作如何完成以及角色之间如何相互作用，这可能涉及到决策层或者人工智能层。当然，这依赖于应用程序，在一项团队游戏中可能出现如下的指令：

- 继续持球；
- 把球传给最近的队友；
- 把球传给中等距离的队友；
- 把球传给最远的队友。

因为这方面是人工智能的研究领域，所以我们不在本书范围内讨论。Funge［FUNG99］写的书是对这个新兴领域的很好的参考，同时也是很好的学习资料。

一个简单却典型的高层次控制的例子是由 Reynolds［REYN87］首先提出的群居模型。这个已经有了很多应用（比如，畜群的惊跑场景）的模型运用了一些简单的“社会”规则/策略，按优先顺序如下：

- 碰撞避免：避免与邻近的伙伴碰撞；
- 速度匹配：尽量与邻近的伙伴在速度上匹配；
- 对心调整规则：尽量靠近周围的伙伴；
- 任意漫步。

图 7-2（彩页中也有）展示了一个在 Fly3D 中实现的群居模拟。尽管在这个例子中，群居的成员不是虚拟的人，这个运算法经过调整后可以模拟一群人的活动。最后一条规则——任意漫步——可以改变为模拟向某一指定方向进行运动。

7.2　顶点动画与合成

我们从考虑如何制作一个角色的网格模型的活动画面开始研究低层次的控制。首先，我们讨论顶点动画或变形的缺点。

由关键帧产生顶点动画是制作一个网格模型的活动画面的最简单方法。关键帧由建模软件完成，而中间的帧则由线性插值法实现。这意味着对于每个关键帧都必须要存储所有顶点的位置，对于一个包含复杂角色、涉及很多关键点的应用软件而言，这将要求很大的存储空间。在游戏中，即使是对单个角色，我们通常也会有许多预先录入的不同的动作序列。一个更为严峻的问题是网格的变形。必须牢记的是：角色是一个有关节的结构（不是一个刚体），顶点的线性插值法不能保证多边形的相关维数，而且在中间的帧将会扭曲躯体。若关键点是封闭的，例如走一个圆圈，这个问题的影响可以控制到最小；但这种方法并非一个好的选择，因为它涉及更多的关键点，因此要求更大的存储空间。

变形的问题在两个动作合成时显得尤为重要，图 7-3 描绘了在合成两个动作序列（例如

一个向前走圈的动作和一个向后走圈的动作）时采用的插值策略。我们考虑如图 7-3 所示的三种模式：在极端情况下，只有向前走或者向后走，此时没有任何问题。在时间 t_1，角色开始向前走，形成一个走动的动画画面，并且此时在关键帧间进行插值；到了时间 t_2，一个指示信号出现，角色将向后运动，这导致了三个不同的插值同时出现并且持续到时间 t_3，（$t_3 - t_2$）的时间间隔可能只有短短的 250 毫秒，在这段时间，向前运动与向后运动都在各自的关键帧间进行插值，同时，我们必须通过插值两种运动的结果来合成实际的运动。时间 t_2 是由用户或者玩家决定的，可以是向前运动开始后的任意时刻。一般而言，向前运动与向后运动的状态有很大不同，合成它们的插值将会生成很多的变形。[⊖]

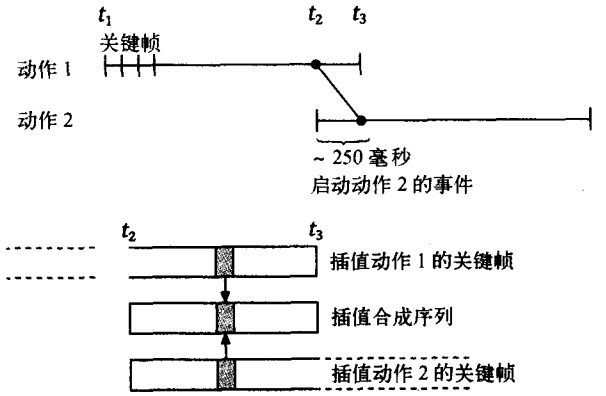


图 7-3 当合成两个关键帧序列时，出现三种形式的插值

显而易见，当合成不同的动画序列时会出现几何变形，然而，单个的序列内部同样会出现大量的几何变形。图 7-4（彩页中也有）说明了这一现象：示例角色同时摇摆着伸出的双手和整个躯体；角色体积的收缩是一个我们很不希望看到的情况。

顶点动画的另一个主要问题来源于角色可能并没有基本结构这样一个事实。如果角色是一个有关节的结构这一事实没有得到确认，将会导致很难建立起角色与场景的相互作用。角色的手在哪里？它的取向性如何？角色并不能以任意的方式出现。

尽管有以上的种种问题，顶点动画依然是一种简单快捷地进行动画制作的方法，这也是该项技术持久流行的原因。虽然有足够的论据证实上述观点，但是对于有关节的结构，这项技术只能被用于简单的应用软件制作。

控制关键帧动画

下面的方式是进行关键帧动画制作的最方便的方法：最简单的关键帧动画是线性的，无论是为了满足这一条件或者出于我们的实际应用需要，插值样条函数［WATT92］依赖于实际的要求以及关键点之间的距离，关键点之间的距离越大，就越需要立体的插值样条函数。

为了控制关键帧动画，我们可以建立一个能够通过整数下标检索所有动画的结构，每个动画有一个整数键值作为标志。然后就有方法可以通过动画和已有的键值设置当前的网格：

⊖ 应该注意的是，只有该种变形形式才是电影 2D 变形所允许的。该图像空间变形过程生成了一组中间帧，它们都是由代表了发生明显扭曲的 3 维物体的 2 维图像构成的。允许该扭曲产生后，2 个不同形状的物体之间的变形得以实现。要在同一物体 3 维方向上插入关键帧，这是完全不同的情况了，我们得到的是累赘变形。

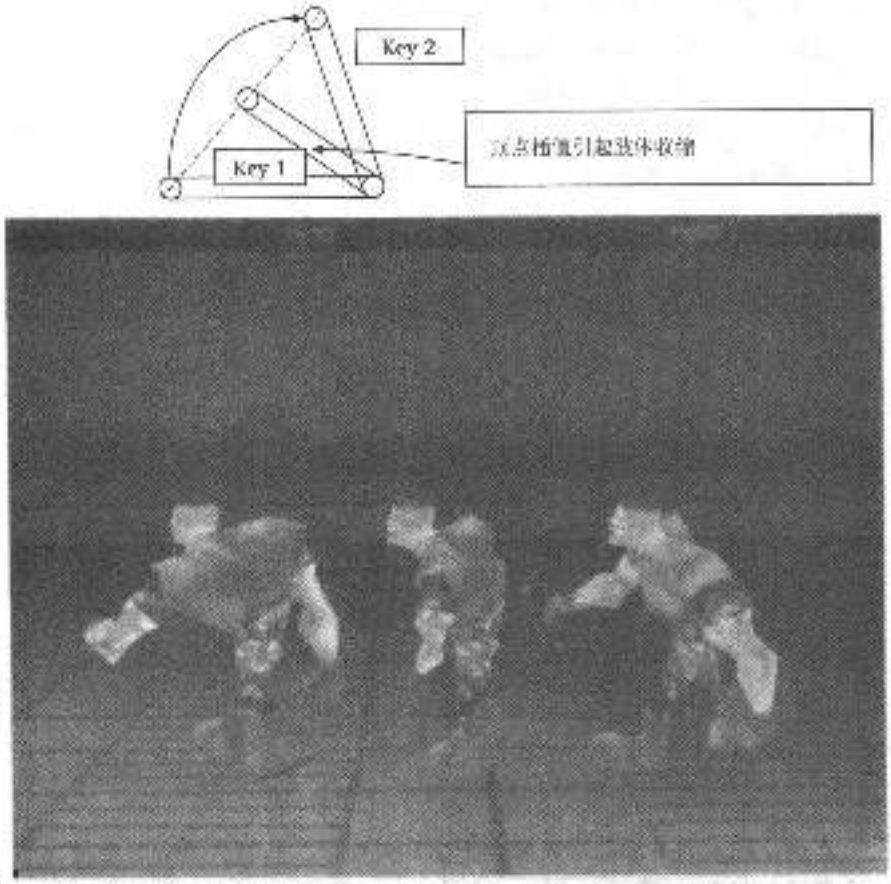


图 7-4　由于顶点插值产生的变形（与图 7-7 比较）

我们可能需要依据单个动画的单个关键点设置网格，也可能需要依据单个动画中的多个关键点的插值甚至依据多个动画序列来设置网格。

引擎类 *flyAnimatedMesh* 包含了上述所有的特性，包含有所有动画的编码。下面是依据已有动画来设置当前网格的方法：

```
set_state(int anim, int key)
```

依据动画和指定的关键点设置当前的网格，不包含插值，形成一个关键顶点的简单拷贝。

```
set_state(int anim, float key)
```

依据指定的动画与一个浮点数键值（0 到 1 之间）设置当前网格，寻找两个最靠近该浮点数键值的整数键值，对它们进行插值生成最终结果。

```
set_state(int anim1, float key1, int anim2, float key2, float factor)
```

依据对两个具有浮点数键值的动画的插值设置当前网格（方法同上），继而用指定的因数（0 到 1 之间）对结果进行插值。这样就能得到两个动画序列平滑的合成。

设置不同状态的源代码如下：

```
void flyAnimatedMesh::set_state(int anim, int key)
{
  int i,j=anim*nv + key;
  for( i=0;i<nv;i++ )
  {
```

```
      localvert[i].x=key_verts[j+i].x;
      localvert[i].y=key_verts[j+i].y;
      localvert[i].z=key_verts[j+i].z;
    }
}

void flyAnimatedMesh::set_state(int anim, float key_factor)
{
   int i,j,k;
   float s;
   vertex *v0;
   flyVector *v1,*v2;

   j=(int)(key_factor*animkeys[anim]);
   if (j==animkeys[anim])
      { j=0; key_factor=0.0f; }
   s=1.0f/animkeys[anim];
   key_factor=(key_factor-j*s)/s;

   v1=&key_verts[(animkeyspos[anim]+j)*nv];
   if (j==animkeys[anim]-1)
      k=0;
   else k=j+1;
   v2=&key_verts[(animkeyspos[anim]+k)*nv];

   v0=localvert;
   s=1.0f-key_factor;
    for( i=0;i<nv;i++)
    {
      v0->x = v1->x*s + v2->x*key_factor;
      v0->y = v1->y*s + v2->y*key_factor;
      v0->z = v1->z*s + v2->z*key_factor;
     v0++; v1++; v2++;
    }
}
void flyAnimatedMesh::set_state(int anim1, float key_factor1, int
anim2, float key_factor2, float blend)
{
   int i,j,k;
   float s,t,w;
   vertex *v0;
   flyVector *v1,*v2,*v3,*v4;

   j=(int)(key_factor1*animkeys[anim1]);
   if (j==animkeys[anim1])
      { j=0; key_factor1=0.0f; }
   s=1.0f/animkeys[anim1];
   key_factor1=(key_factor1-j*s)/s;

   v1=&key_verts[(animkeyspos[anim1]+j)*nv];
   if (j==animkeys[anim1]-1)
      k=0;
   else k=j+1;
   v2=&key_verts[(animkeyspos[anim1]+k)*nv];

   j=(int)(key_factor2*animkeys[anim2]);
   if (j==animkeys[anim2])
      { j=0; key_factor2=0.0f; }
   s=1.0f/animkeys[anim2];
   key_factor2=(key_factor2-j*s)/s;

   v3=&key_verts[(animkeyspos[anim2]+j)*nv];
   if (j==animkeys[anim2]-1)
```

```
    k=0;
  else k=j+1;
  v4=&key_verts[(animkeyspos[anim2]+k)*nv];

  v0 = localvert;
  s=1.0f-key_factor1;
  t=1.0f-key_factor2;
  w=1.0f-blend;
  for( i=0;i<nv;i++ )
  {
    v0->x=(v1->x*s+v2->x*key_factor1)*w+
          (v3->x*t+v4->x*key_factor2)*blend;
    v0->y=(v1->y*s+v2->y*key_factor1)*w+
          (v3->y*t+v4->y*key_factor2)*blend;
    v0->z=(v1->z*s+v2->z*key_factor1)*w+
          (v3->z*t+v4->z*key_factor2)*blend;
    v0++; v1++; v2++; v3++; v4++;
  }
}
```

我们现在考虑如何控制或者组织要求的动画序列。一个动画控制器应当存储当前动画的索引和开始时间，以及下一个动画的索引和开始时间。通常下一个动画的值被设置为“-1”(表示没有动画)，动画控制器所做的就是重复基于开始时间的当前动画。当下一动画连同它的开始时间得到设置，例如对应接口事件的一个变化，控制器就会运用最近的 *set_state* 方法来合成两个动画。当合成完成时，“下一动画”就变为“当前动画”，而此时的下一动画的值设为“-1”。

这种简易的控制器完成了对两个动画的合成，但是我们可以轻易地建立一种更加精细的结构来实现对当前动画的修正。例如，假设一个正在行走的角色遭到枪击，我们希望在行走的基础上添加“我受伤”的姿态。完成这一动作的插值策略包含两个插值（见图 7-5）。在时间 t_2，角色受到枪击，我们立刻将受伤的姿态添加到当前动画中。通过在 t_2 到 t_3 的时间间隔内运用一个从 0 变到 1 继而又变回到 0 的因数来完成这一操作，在时间 t_3 以后，正常的动画又将持续。这要求一种能够将当前网格与来自其他动画的单个关键点合成的方法。

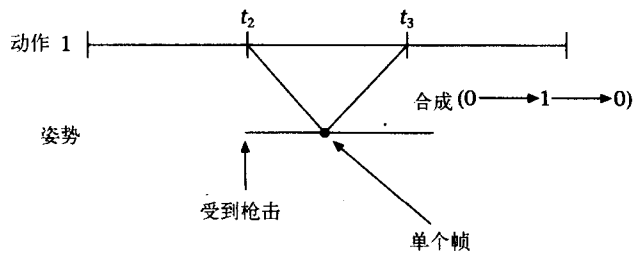


图 7-5　单个帧中反应姿势的插值

```
void flyAnimatedMesh::set_state_blendcur(int anim, int key, float
blend)
{
  flyVertex *v0=localvert;
  flyVector *v1=&key_verts[(animkeyspos[anim]+key)*nv];
  float f=1-blend;
  for( int i=0;i<nv;i++ )
  {
    v0->x = v0->x*f + v1->x*blend;
    v0->y = v0->y*f + v1->y*blend;
    v0->z = v0->z*f + v1->z*blend;
    v0++; v1++;
  }
}
```

这同样可以用于对两个序列合成动画的修正；我们还可以把一个动画序列看作修改器而不是用单个帧，这样可以避免调用复杂的 *set_state_blendcur* 方法。

控制器的代码如下：

```
#define ANIM_FRAME_TIME 33

int i1,i2,j1,j2;
float f1,f2,f3;

i1=objmesh->animkeys[cur_anim]*ANIM_FRAME_TIME;
j1=g_flyengine->cur_time-cur_anim_time;
f1=(j1%i1)/(float)i1;

if (next_anim==-1)
  objmesh->set_state(cur_anim,f1);
else
{
  i2=objmesh->animkeys[next_anim]*ANIM_FRAME_TIME;
  j2=g_flyengine->cur_time-next_anim_time;
  f2=(j2%i2)/(float)i2;

  if (j2>next_anim_dur)
  {
    cur_anim=next_anim;
    cur_anim_time=next_anim_time;
    next_anim=-1;
    objmesh->set_state(cur_anim,f2);
  }
  else
  {
    f3=(float)j2/next_anim_dur;
    objmesh->set_state(cur_anim,f1,next_anim,f2,f3);
  }
}

if (mod_anim!=-1)
{
  i1=objmesh->animkeys[mod_anim]*ANIM_FRAME_TIME;
  j1=g_flyengine->cur_time-mod_anim_time;
  if (j1>mod_anim_dur)
    mod_anim=-1;
  else
  {
    f1=1.0f-(float)abs(j1*2-mod_anim_dur)/mod_anim_dur;
    objmesh->set_state_blendcur(
      mod_anim,objmesh->animkeys[mod_anim]-1,f1);
  }
}
```

7.3 骨架动画

通过使用骨架动画可以弥补上一节中提到的顶点动画的种种几何缺陷，但这是以增加复杂性为代价的。（这里我们指的是算法的复杂性，而用于指定动画的数据的复杂性是降低的。）顾名思义，骨架动画是用允许动画控制的有关节的结构作为骨架，然后将易于渲染的网格贴到骨架上。在游戏中，制作人员精心地制作角色，使它们的每个顶点与一块或多块的骨骼发生联系，从而能够将网格或者皮肤贴上去。在另外一些软件中，我们可以看到通过程序生成的皮肤，此类设计正如平面设计一样，设计人员构造角色的“外表”的工作十分重要。

在骨架动画中，只有骨架的动画控制器需要被存储，同时，骨架节点的数目远远少于网格的顶点数目，所以对存储空间的要求大大降低了（虽然单个骨架节点的信息量是一个网格顶点

的4倍)。而且，许多不同的网格可以通过使用相同的骨架共享相同的动画。骨架动画还允许IK（见第11章）的使用。在几乎所有运用了动画的绘图中，网格与动画的分离问题都至关重要，而骨架动画能使这一问题变得简单。

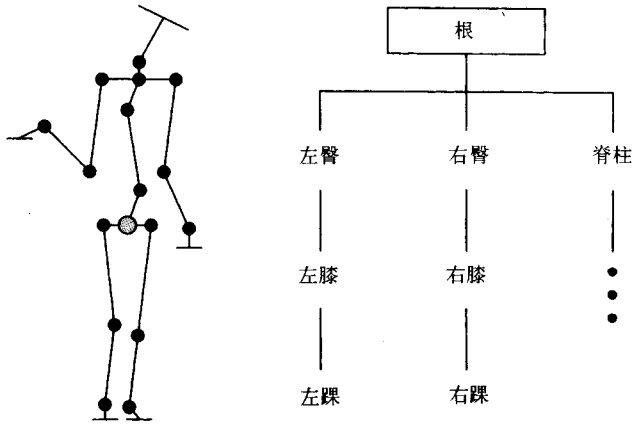


图7-6　在骨架动画中使用骨架层次结构

首先考虑骨架的结构，它存在数种可能的选择。用于存储骨架结构的最简单方法是：为每一个节点建立一个矩阵来描绘它相对于初始位置的旋转量和位移量。注意，这种方法既不能将节点明确地连接起来，也不能反映出骨架是一个层次；如果每一块骨头都有一个自身的定位，那么就需要用强制性的约束条件把所有的骨头联系起来，而所有这些约束条件需要在动作发生时得到应用。

下面考虑建立了明确的层次关系的树形结构，每个子节点有一个描绘它相对于其父节点的旋转量和位移量的矩阵。全局定位和方向都应用于根节点，所有其他的动作都与根节点相联系。

因而，层次结构是一个较好的选择。图7-6描绘了一个简单的骨架层次结构。

选择层次结构的另一个动机是我们可以方便地对关节难以实现的动作进行限制——例如，肘弯向错误的方向。

所有的皮肤顶点参量应当全部用统一单位来表示，上述的矩阵依据连接从根节点到发生动作顶点的所有骨头的位移量和旋转量形成。例如，为了使根节点与手上的节点 $\mathbf{v}$ 相匹配，我们用下面的公式：

$$\mathbf{v}_w = \mathbf{M}_{hand}\mathbf{v} = [\mathbf{T}_{waist}\mathbf{R}_{waist}\mathbf{T}_{stomach}\mathbf{R}_{stomach}\mathbf{T}_{chest}\mathbf{R}_{chest}\mathbf{T}_{upper\text{-}arm}\mathbf{R}_{upper\text{-}arm}\mathbf{T}_{forearm}\mathbf{R}_{forearm}\mathbf{T}_{hand}\mathbf{R}_{hand}]\mathbf{v} \qquad (7\text{-}1)$$

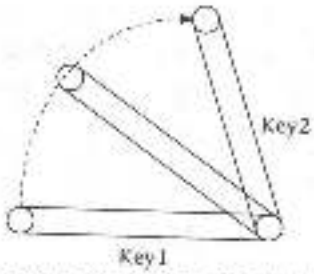
在整个动画中，骨架层次关系保持恒定，并且指定了连接各关节的骨头的长度，动画数据中包括了每块骨头的旋转量矩阵 $\mathbf{R}$。

图7-7（彩页中也有）使用了和图7-4（顶点动画）一样的角色，然而，这次的角色是由骨架动画贴上皮肤形成的。从中可以看出，骨架动画中，旋转量插值法消除了显著的失真；同时可以看出，动画角色的制作运用了简单的骨架。当前游戏中运用的骨架趋向于使用这种复杂程度。图中角色的手和脚中没有骨头，这当然就意味着脚不能因为踝关节的运动而转动，手不能因为腕关节的运动而转动。

可以通过单个数据结构建立一个骨架层次：

```
flyMatrix skeleton_node[num_bones];
int parent_node[num_bones];
```

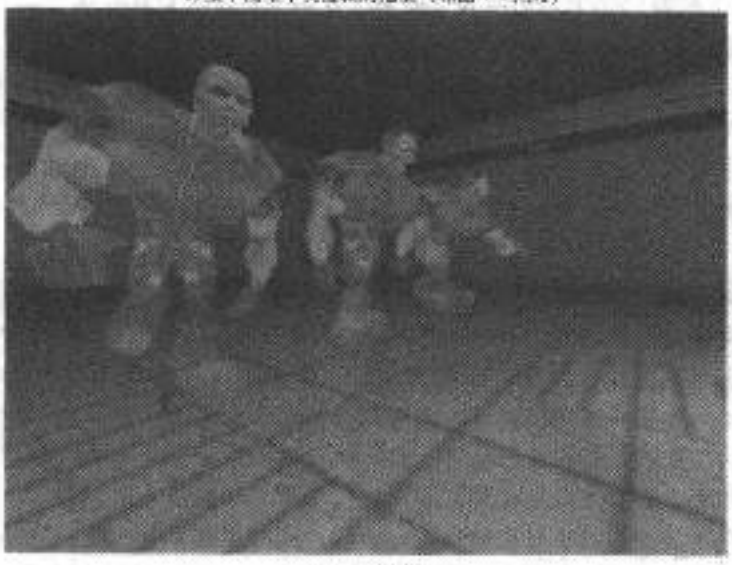
其中，矩阵数组 *skeleton_node*（每块骨头一个）记载了节点关于其父节点的变换；数组 *parent_node* 包含了每个节点的父节点索引，其中 -1 表示根节点。这种结构的优点在于：父节点被固定，同时用一个简单的矩阵数组成完成了对骨架的关键帧的存储。



a) 对数字量进行插值消除了顶点插值的样由



b) 整个图像中关键帧的插值（和图 7-4 比较）



c) 使用的骨架

图 7-7 在消除皮肤层的几何扭曲方面，骨架动画优于顶点动画

下面考虑由骨架决定顶点位置的方法，每个顶点至少和一块骨头直接关联。若一个顶点由单个骨架节点确定，我们称之为一个刚性节点。在实践中，一个顶点可能与多块骨头发生关联，与每块骨头的关联都包括该骨头的索引、一个加权因数（0 到 1 之间）以及一个位移矢量；该位移矢量是从骨头到顶点的位移的矢量，每一个顶点的所有加权因数之和为 1。我们用下面的矢量积公式计算顶点与它所关联的骨头的相对位置：

$$\sum_{i=0}^{n}\mathbf{M}_i\mathbf{d}_i w_i$$

此处：

$\mathbf{M}_i$ 表示骨头 i 的（全局）矩阵；

$\mathbf{d}_i$ 表示从顶点到第 i 块相连骨头的位移矢量；

w_i 表示相连的权重；

n 表示相连的骨头的数目。

这是为骨架蒙皮的最基本模型，虽然简单但也提出了很多问题，这些将在第 8 章中与很多其他归纳出的模型一起得到充分的说明。

下面的代码实现了由骨架生成所有顶点的策略：

```
// build the vertices based on current skeleton
void flySkeletonMesh::build_verts()
{
   int p=0;
   int i,j;
   for( i=0;i<nv;i++ )
   {
   verts[i].null();
   for( j=0;j<nvweights[i];j++,p++ )
     verts[i] += (m[vindex[p]]*vweight[p])*vweight[p].w;
   }
}
```

（注意，在上面的代码中，矩阵矢量的乘法采用了操作符重载算法。）上面简单的代码以骨架的当前位置为基础生成了网格所需的所有顶点，它们可以在帧中进行渲染。

下面考虑如何合成多个骨架，一旦合成问题的一些细节得到解决，就可以用与顶点动画相同的合成控制器，而且可以得到比使用线性插值法调用顶点生成程序更好的效果。

顶点动画中的插值法是在两个位置间进行简单的线性插值，现在我们必须解决在两个表示旋转量和位移量的矩阵之间进行插值的问题。当矩阵中都没有缩放元素时，这个问题很容易解决。

在这个阶段，我们应当考虑矩阵中旋转量和位移量的差异。对一个静态的对象而言，矩阵中位移量保持不变，可以很快确定它相对于父节点的距离，换言之，就是肢干的长度。而所有与手臂或腿相关的动作都来自关节的旋转；在关节保持不变的情况下，位移量保持不变。注意，对于一个实际的角色而言，上述观点并非完全正确，例如脊柱的弯曲超过所有的有效长度时；为了准确地模拟这一事件，我们将脊柱分为一块一块的脊骨。对于一个正在行走或者奔跑的角色，根节点发生位移，我们对位移量进行插值处理。

现在可以利用线性插值法对位移量进行插值，同时用四元数插值法（*slerp*）对旋转量进行插值（见附录 7.1）。下面的代码实现了对两个没有发生缩放的矩阵的插值。（注意，如果矩阵发生了缩放，我们不得不使用一种仿射几何的分离方法将旋转量和缩放分量分开。）

```
void FlyMatrix::slerp(FlyMatrix& m1,FlyMatrix& m2,float t)
{
  FlyQuaternion q1(m1),q2(m2),qt;
  qt.lerp(q1,q2,t);
  qt.get_mat(*this);
  m[3][0]=m1.m[3][0]*(1-t)+m2.m[3][0]*t;
  m[3][1]=m1.m[3][1]*(1-t)+m2.m[3][1]*t;
  m[3][2]=m1.m[3][2]*(1-t)+m2.m[3][2]*t;
}
```

需要特别注意的是，上述的插值和蒙皮程序相当简单。在上面的代码中，四元数构造器完成了将旋转矩阵同四元数的转换，四元数类中的 *slerp* 方法实现了两个四元数间的插值，而 *get_mat* 方法又将四元数转换回旋转量；位移量的线性插值也是这样完成的。预计算出每个骨架节点的关键值可以加快将旋转矩阵转化为四元数的运算速度。插值应当在局部坐标空间进行（矩阵与父节点相联系），而不是在全局坐标空间进行（矩阵直接与根节点联系）。

图 7-8（彩页中也有）描绘了运用此种策略生成的一个角色动画。*slerp* 控制了骨架中各块骨头间的插值，角色通过源于根节点的位移和方向变化，沿着贝济埃曲线行进。

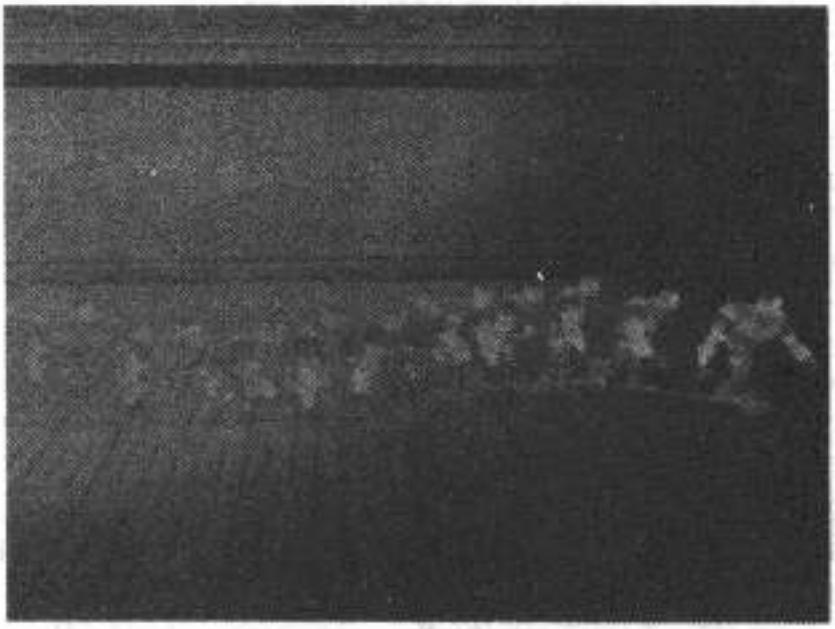


图 7-8 骨架间的四元数插值控制根节点生成
遵从贝济埃曲线运动的位移和方向变化

插值中 IK 的作用

IK（反向运动学）在当前游戏中主要用于一些简单的显示：控制游戏中人物注视的方向、瞄准射击等。简单而言，IK 仅仅被提供给一些实体，例如一条手臂。在解决更复杂结构的实时问题时，IK 显得过于复杂而且代价太大。相反在很多场合，综合使用多种方法能够获得很好的效果。举“注视”这个动作为例，当一个角色注视一件运动着的物体时，角色只能在脖子的转动范围内转动他的头。而一旦达到其转动范围的限度，必须调用身体的转动

来使“注视”动作得以延续。而另一个角色举枪瞄准指定的目标，我们会使用相似的方法来处理他，他的身体旋转时，IK 就能调准整条手臂的取向使手臂在可能的区间内运动。

“静态场景”动画是一种获得用户灵活控制的行走路径的简单方便的方法，使用这种方法制作角色走圈的动画时，根节点一直不改变位置，就好像角色在健身器上走路一样。当动画循环时，控制器对根节点进行简单的平移或者旋转。然而，这又提出了一个动画速度与用户控制速度同步的问题。为了达到同步通常会要求角色完成变化的动作。（事实上，实际的走圈过程中，臀部动作并不是保持不变的，而是在整个走动过程中不断改变的。）在第一人称射击游戏中，这一影响可以通过调节角色运动的速度来改善；但是当角色的前向运动变得更慢的时候，这一我们不希望出现的影响变得十分引人注意。游戏中需要让人物在用户的控制下沿某一路径以某一速度运动，理想的解决方法是实时地计算出正确的动画。这涉及到 IK 的正确使用，具体的问题将在第 11 章中讨论。

由此可以看出，目前 IK 是一个标准动画制作控制器转换模块，并在有关节的结构中得到应用。但是它尚不能够完成对整个动画序列的所有结构的控制，而且在近期内也不能实现这一目标。

7.4　低层次动画管理

正如已经看到的那样，游戏中简单的角色动画不考虑分类地将混合的动作序列（通常预先录制或者预先进行了脚本制作）合成角色连续的动作。本节就考察如何找到一种简单的策略来实现对动画序列的低层次组织。这包括对用户控制角色的简单研究，以及对角色在游戏环境中与周围对象发生相互作用从而调用各种动画序列的研究。面向对象的方法处理这一任务十分有效。我们同样会尽量将序列 $n-1$ 中的最后一帧图像的取向性与序列 n 的第一帧图像的取向性匹配，从而可以避免粗糙的两个序列合成技术。

生成一个完整的序列也就是要求生成角色与对象相互作用的动画序列，我们将这些序列分为 3 种类型：进入（状态）、退出（状态）和保持（状态）。进入和退出两个序列的时间长度都是一定的，保持序列的时间长度是任意的。这些序列是相互关联的，比如说一个行走动画需要角色从一个对象移动到另一个对象。保持的动画状态用来表示持久的动作或者角色保持不变的姿势，例如坐着或者站着不动，这种状态下的动画也包括角色站着不动观察四周。进入（状态）动画就是指角色进入保持状态的动画，保持状态持续一段时间直到角色状态发生变化，也就是退出（状态）发生。

现在我们用一个简单的剧情来说明这一概念（见图 7-9）。这是一个由用户或者人工智能控制的角色，他正在游戏环境中运动并与场景中的对象发生交互。

每一个对象都有进入、退出和保持的动画。在这个剧情中有三个对象和一个角色，有下面所示的动画：

椅子

进入　　坐下

保持　　放松并且四周张望

退出　　从椅子起身

电脑终端

进入　　坐下

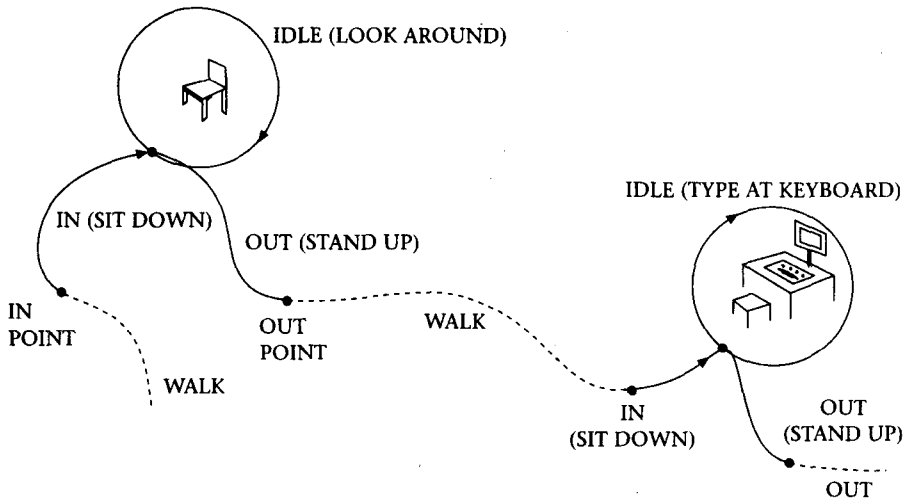


图 7-9 一种实现动画序列的低层次管理的简单方法

保持 通过键盘打字
退出 起身离开终端
地面
进入 没有动作——转到下一状态
保持 站着四周张望
退出 没有动作——转到下一状态
角色
仅有一个指示走动的动画序列用于过渡状态

所有的动画序列都是预计算出的或者预捕捉的，角色走动的动画也是预计算出的，但是根节点的取向性和位移量是随着游戏进程被计算出来的。角色希望走向椅子，坐下来休息一段时间，而后起身走到电脑终端开始打字。整个动画序列通过连接“打好包”的进入、保持、退出动画序列生成。在各个序列之间，角色从当前状态包中退出状态动画的最后一帧走到下一状态包进入状态动画的第一帧。与角色交互的每个对象的“进入”状态序列决定了角色的位置和取向性，这一位置和取向性将和进入状态序列中的第一帧动画相匹配。而在适当的位置以适当的取向性结束角色的行进是路径生成器的职责。与之相似，退出状态序列的结尾给出了下一次行进的位置和取向性。

再次观察图 7-9，考虑角色正处于椅子的保持状态——他正坐着四周张望。此时一个事件发生，要求他到电脑终端进行操作。next pack 缓冲区被设置为包含电脑终端的状态包，next pack 缓冲区的初始化导致了角色状态的变化以及椅子状态包中的退出状态的实现。角色需要寻找一条路径，实现从椅子状态包退出状态的最后一帧到电脑终端状态包进入状态的第一帧的转变，并且走过去以改变状态。这种策略可以通过一系列状态变迁图来描绘，存在四种可能的状态：进入、保持、退出和行进（见图 7-10）。

显然，这种结构只能用于处理一些受限的问题；但它的确在不依靠合成定位的条件下正确地处理了连接两个动画序列的问题。

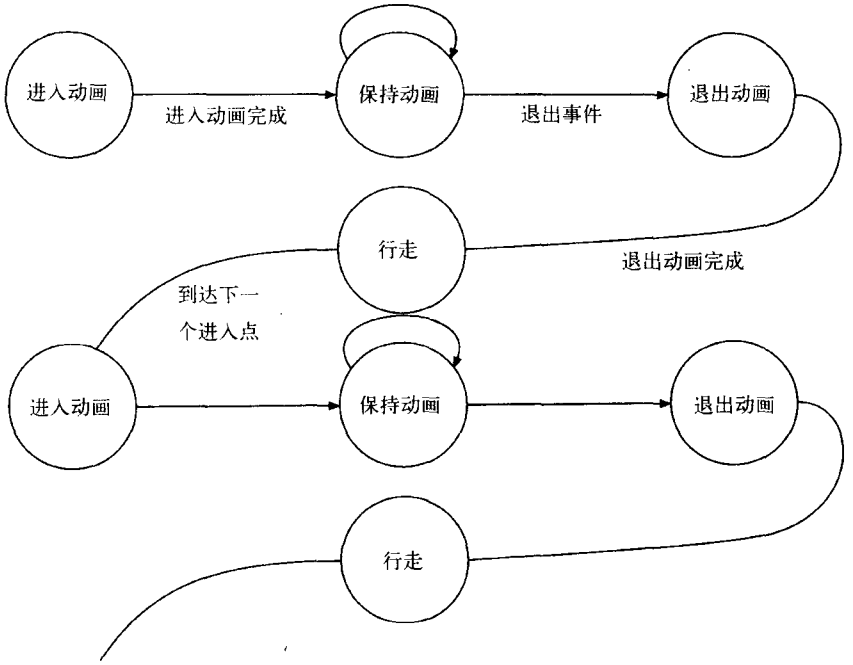


图 7-10　一组动作序列的状态变迁图

7.4.1　行进的路径规划

从最低的层次开始，我们可以考虑在两个由动画“连接”决定取向性的点之间生成一条路径。行进的路径可以用贝济埃曲线来生成，我们只需要保证退出动画的最后一帧的切线方向与行进动画的第一帧的切线方向相匹配，这可以通过下述的一个简单的例子来描述（见图 7-11）。

行进的开始和结束定义了两个控制点 p_0 和 p_3，它们分别是对象动画的退出点和进入点。点 p_1 的位置在退出动画的最后一帧的切线方向上，点 p_2 的位置在进入动画的第一帧的切线方向的相反方向上。

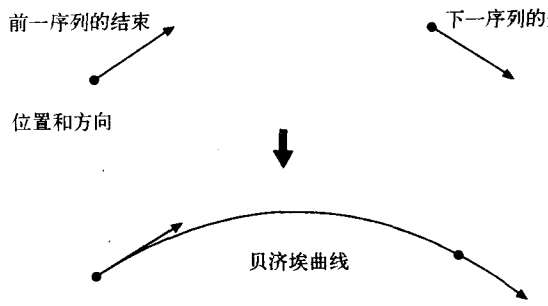


图 7-11　利用序列 n-1 最后的方向和序列 n 最初的方向生成贝济埃曲线

问题在于：我们如何为切线方向上的点定位？如果离控制点太近会导致一个过快的转折，而且角色也会很迅速地旋转；如果离控制点太远，将会生成一个冗长的行进过程。假定一个简单的定位是在 p_0 点和 p_3 点中间，这可以通过下面的代码实现：

```
// build walk path with a single bezier segment
bezier_curve build_curve_1seg(
  vector& out_pos,vector& out_dir,vector& in_pos,vector& in_dir)
{
  bezier_curve walk_path;
  walk_path.set_dim(2);
```

```
    vector v=out_pos-in_pos;
    v.z=0;
    path_dist=v.length()/3.0f;

    walk_path.add_point(&out_pos.x);

    v=out_pos+out_dir*path_dist;
    walk_path.add_point(&v.x);

    v=in_pos-in_dir*path_dist;
    walk_path.add_point(&v.x);

    walk_path.add_point(&in_pos.x);

    return walk_path;
}
```

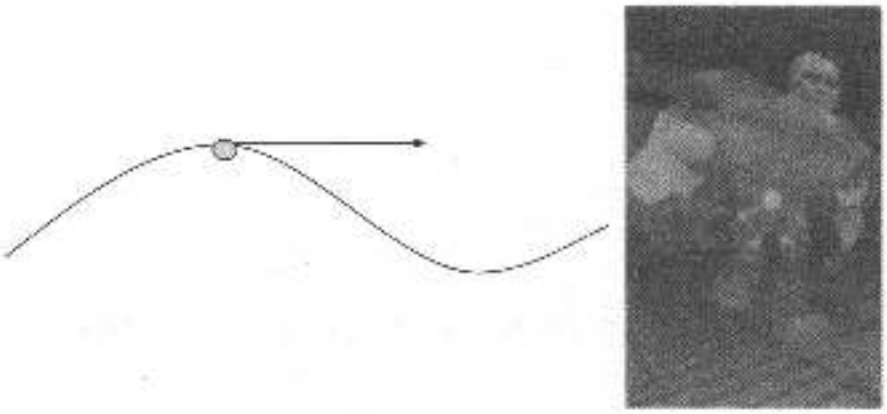
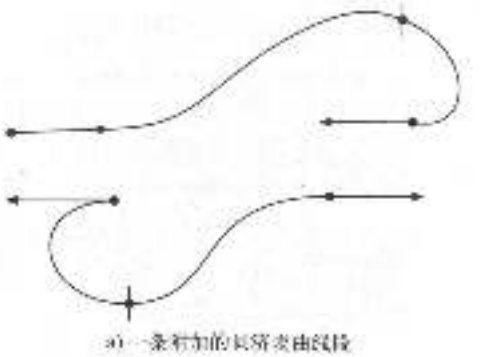


图 7-12 根据贝济埃曲线控制角色根节点的位置和朝向

一旦计算出贝济埃曲线路径，我们可以直接依据曲线的参数得出角色根节点的位移量和旋转量[○]，如图 7-12 所示（彩页中也有）。

上述的分析提供了单段的贝济埃曲线路径，但还有很多情况它并不能满足。当切向量共线时，会出现复杂的情况，在这些情况下，我们需要增加多段贝济埃曲线以得到一条合理的路径。在图 7-13 中，运用两段或三段曲线取得了令人满意的解决方案。

为了提供一个处理此类特殊问题的策略，我们需要确定出现了下列四种情况中的哪一种：向前-向前，向前-向后，向后-向前，向后-向后。可以利用两个点积来高效地完成这项工作。



a) 一条初始的贝济埃曲线段

b) 两条初始的贝济埃曲线段

图 7-13 当切向量共线时，用附加的贝济埃曲线段获得合理的路径

○ 注意，为实现位移，曲线参数中的相等间隔并不能得出位移——即等长下相等的间隔。这取决于应用和该角色沿曲线位移的速度。[WATT00] 给出了复杂参数化的简便解决方案。

```
// analyse the path
int analyse_path(
  vector& out_pos,vector& out_dir,vector& in_pos,vector& in_dir)
{
  verctor v=in_pos-out_pos;
  float dot1=vec_dot(out_dir,in_dir);
  float dot2=vec_dot(out_dir,v);

  if(dot1>0)
    if(dot2>0)
      return PATHCONFIG_FORTH_FORTH;
    else
      return PATHCONFIG_BACK_BACK;
    else
      if(dot2>0)
        return PATHCONFIG_FORTH_BACK;
      else
        return PATHCONFIG_BACK_FORTH;

    // unreachable
    return -1;
  }
```

下面的代码可以为任意情况创建路径：

```
// build walk path with up to three bezier segments
bezier_curve build_curve_3seg(
  vector& out_pos,vector& out_dir,vector& in_pos,vector& in_dir)
{
  bezier_curve walk_path;
  walk_path.set_dim(2);

  // calculate 1/3 of the distance
  vector v=pos-in_pos;
  v.z=0;
  path_dist=v.length()/4.0f;

  // add the control points
  walk_path.add_point(&out_pos.x);

  v=out_pos+out_dir*path_dist;
  walk_path.add_point(&v.x);

  vector v1=out_dir*path_dist,v2;
  v2.x=v1.y;
  v2.y=v1.x;
  v1.z=v2.z=0;

  vector v3=in_dir*path_dist,v4;
  v4.x=v3.y;
  v4.y=v3.x;
  v3.z=v4.z=0;

  int i=analyse_path();

  switch(i)
  {
    case PATHCONFIG_FORTH_BACK      :
    {
      v.cross(v3,v1);
      if(v.z<0)
```

```
            v4.negate();
          v=in_pos+v3+v4;
          walk_path.add_point(&v.x);

          v=in_pos+v4;
          walk_path.add_point(&v.x);
          v=in_pos-v3+v4;
          walk_path.add_point(&v.x);
        }
        break;
        case PATHCONFIG_BACK_FORTH      :
        {
          v.cross(v3,v1);
          if(v.z<0)
            v2.negate();

          v=out_pos+v1+v2;
          walk_path.add_point(&v.x);

          v=out_pos+v2;
          walk_path.add_point(&v.x);

          v=out_pos-v1+v2;
          walk_path.add_point(&v.x);
        }
        break;
        case PATHCONFIG_BACK_BACK       :
        {
          v.cross(v3,v1);
          if(v.z<0)
          {
            v2.negate();
            v4.negate();
          }

          v=out_pos+v1+v2;
          walk_path.add_point(&v.x);

          v=out_pos+v2;
          walk_path.add_point(&v.x);

          v=out_pos-v1+v2;
          walk_path.add_point(&v.x);

          v=in_pos+v3+v4;
          walk_path.add_point(&v.x);

          v=in_pos+v4;
          walk_path.add_point(&v.x);

          v=in_pos-v3+v4;
          walk_path.add_point(&v.x);
        }
      }

      v=in_pos-in_dir*path_dist;
      walk_path.add_point(&v.x);

      walk_path.add_point(&in_pos.x);

      return walk_path
    }
```

7.4.2　骨架动画和面向对象的动画控制

我们现在考虑这种策略是如何与骨架动画结合起来的。由于是进行骨架动画制作，我们可以用一个通用的骨架动画序列作为各个对象的进入、保持、退出状态包，然后将任意角色的网格与之相匹配。这意味着由不同网格描绘的角色都可以与对象发生相互作用。

下面继续考虑上一节的路径构造问题。我们需要从退出和保持状态位中分别找出最后一帧和第一帧图像的位置和方向。对于骨架动画而言，这是十分简单的，因为它们已经被包含在动画数据中。（注意，对于顶点动画而言，必须进行个别说明，因为它们不是动画数据的一部分。）

我们需要的信息是最后一帧（退出状态）和第一帧（进入状态）中根节点的矩阵。矩阵中的位移量提供了角色的位置，旋转量提供了角色的方向。矩阵中旋转量部分的 3×3 大小的块的每一列代表了当前旋转的一个分量，我们需要根据指定的要求选择 x 分量或者 y 分量，图 7-14（彩页中也有）描绘了这一操作策略。

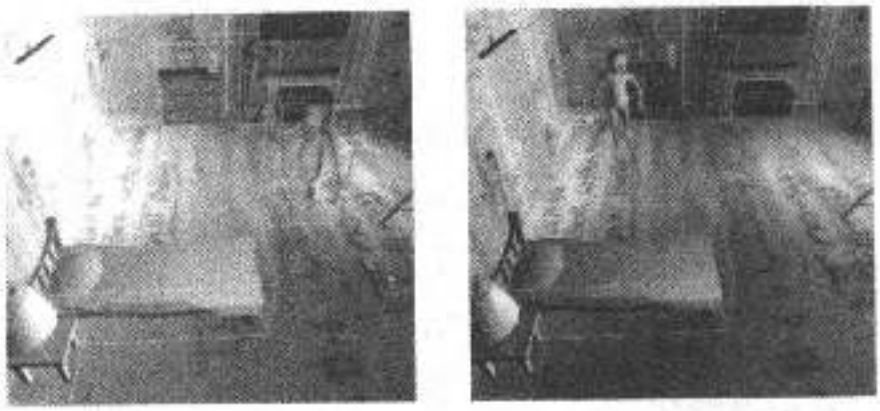


图 7-14　两个人从不同的位置步行到“锅子”状态的进入状态点。两条路径在同一点结束，所以角色在这一点的姿势相同。从而可以将端下的动画与步行的动画成功合成

最后注意，在这一简单的策略中，我们仅仅匹配了两个动画序列中角色的取向性。骨架的实际姿势——肢体的实际位置——并没有考虑在内。（这将是第 12 章中讨论的问题。）从而我们假设每一帧动画都在相似的位置结束或开始。

7.4.3　对障碍物的躲避

这一策略包含了一个对路径规划的预计算策略。如果一开始我们认为路线上没有障碍物，那么对于 n 个对象可以预先建立起 n 条贝济埃曲线，在行进时，角色沿着这些路径运动。

当存在障碍物时会有两种可能性：第一，当角色由用户控制时，用户可以很容易地将路径分解为多个部分然后组合出一条没有障碍的通路，从而避免与障碍物的碰撞。第二，当角色由游戏的人工智能控制时，我们需要一些实时的路径规划用于对障碍物的躲避。

在这种情况下，我们可以按如下方法继续下去：对等间距的时间间隔（由参量决定）抽样，利用光线交叉碰撞检测判断给出的曲线上会否发生碰撞，光线的方向由已经给出的曲线的切向量决定。如果在所有的抽样中都不会发生碰撞，那么就认为路径上没有障碍物。当检测到一次碰撞，角色可以在碰撞面上转过 90°（假设），然后利用一个新的退出状态矢量和原来的进入状态矢量构造一条新的贝塞尔曲线（见图 7-15）。这一过程可以递归进行，直到寻找到一条没有碰撞的路径，或者得出预先设置的参数有限制性——即找不到一条满足条件的可能路径。这一方法存在的问题在于它导致了很多不连续段的出现。

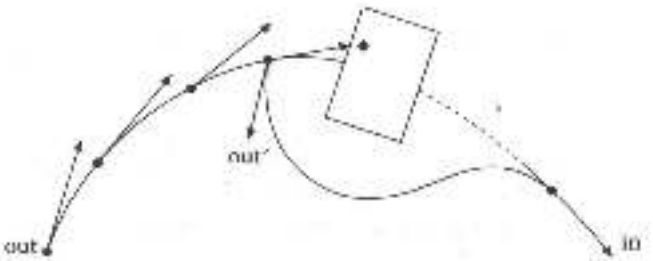


图 7-15 利用光线交叉碰撞检测检查障碍物，光线的方向由已经给出的曲线的切向量决定

我们可以在利用相同方法的同时进行预测，并且在得到符合条件的路径后再创建出连续的中间部分，而不是不断地沿递归的方法行进，由此可以避免不连续段的出现。这就会导致生成一条由三部分组成的曲线。其中，依据递归方法探测出碰撞点，可以避免中间部分曲线的延伸部分穿透到其他对象中。图 7-16（彩页中已有）描绘了这种简单的操作策略的一个例子，图中各对象都被装入 AABB，它们是用于测试碰撞的包围体（详见第 2 章）。

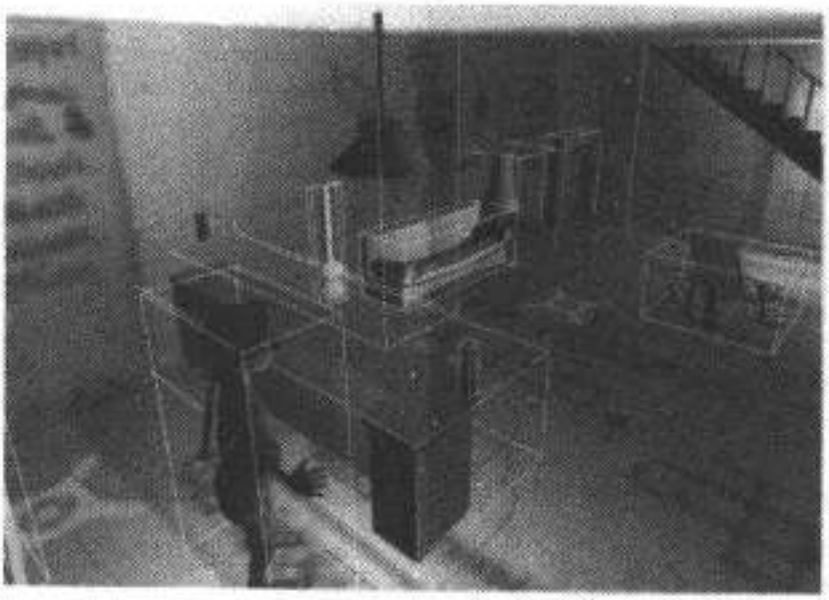


图 7-16 躲开障碍物，从红色的位置移动到蓝色的位置，在这个例子中，对象都被装进了 AABB

为对象使用包围体来避免碰撞这一方法可以适用于绝大多数的情况，然而要注意，用 AABB 封装的对象并不适用于复杂应用。（这点在第 2 章中讨论过。）

7.4.4　路径规划总结

我们现在将这些素材与第2章中提到过的几何路径规划策略结合起来，路径规划可以归纳为如下的操作序列：

1）利用在第2章中介绍的伪入口策略建立穿越层次的潜在路径图，这是创建过程。

2）对于一个在两个对象之间的行进序列，用A*寻找一条穿过入口的路径（同样在第2章中介绍）。

3）用上文提到的贝济埃策略生成一条经过2）中选择的节点的曲线路径。

4）避免与2）中选择的节点包含的对象发生碰撞。

上述方法可以和贝济埃路径规划策略结合起来。图7-17描绘了一个房间的退出点、另一个房间的进入点以及连接起点和终点的第三个房间。如图所示，连接入口确定了中间法向，整个路径由三段组成，每扇门的中点就是各段之间的连接点。

附录7.1　用四元数描绘旋转

一种很有用的引导性观点是把四元数看作一个与矩阵类似的算子，它把一个矢量变为另一个矢量，但是不像矩阵元素那样可以在无限的值中进行选择。与在矩阵中指定9个元素不同，我们采取定义4个实数的方法。首先看下面这个问题：一个矢量绕轴**n**发生θ度的角位移。

我们用（θ，**n**）来定义一个关于**n**轴的θ度的角位移，也就是说，用（θ，**n**）表示旋转量而不是用R（θ_1，θ_2，θ_3）来表示。考虑如图7-18所示的情况，矢量**r**发生角位移运动到**Rr**的位置。

这个问题可以通过分解**r**矢量得到简化，将**r**分解为与**n**平行的部分和与**n**垂直的部分，前一部分在旋转前后保持不变，后一部分在从**r**移动到**Rr**的平面内。

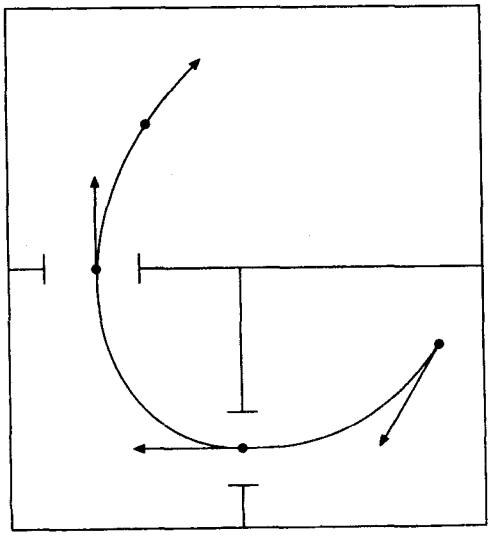


图7-17　通过入口构造一条多段贝济埃路径

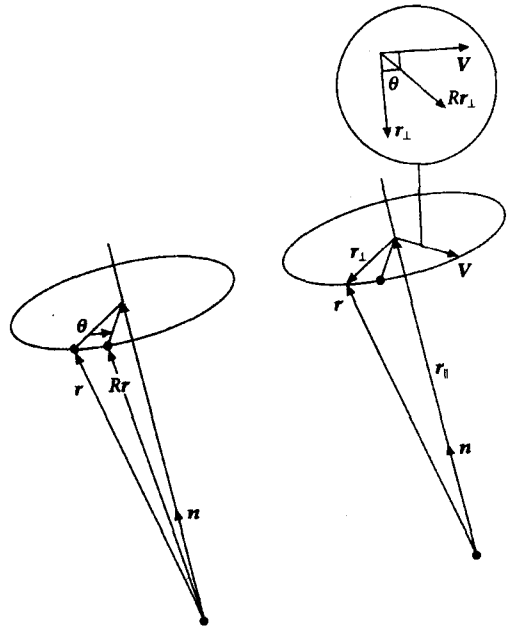


图7-18　**r**的角位移（θ，**n**）

$$\mathbf{r}_{\parallel} = (\mathbf{n}.\mathbf{r})\mathbf{n}$$

$$\mathbf{r}_{\perp} = \mathbf{r} - (\mathbf{n}.\mathbf{r})\mathbf{n}$$

$\mathbf{r}_{\perp}$ 旋转到了 $\mathbf{Rr}_{\perp}$ 的位置。我们在该平面内建立一个垂直于 $\mathbf{r}_{\perp}$ 的矢量 V，为了算出旋转角度，我们记：

$$\mathbf{V} = \mathbf{n} \times \mathbf{r}_{\perp} = \mathbf{n} \times \mathbf{r}$$

其中 × 表示叉乘，从而：

$$\mathbf{Rr}_{\perp} = (\mathrm{Cos}\theta)\ \mathbf{r}_{\perp} + (\mathrm{Sin}\theta)\mathbf{V}$$

因此

$$\begin{aligned}\mathbf{Rr} &= \mathbf{R}_{\parallel} + \mathbf{Rr}_{\perp} \\ &= \mathbf{R}_{\parallel} + (\mathrm{Cos}\theta)\mathbf{r}_{\perp} + (\mathrm{Sin}\theta)\mathbf{V} \\ &= (\mathbf{n}.\mathbf{r})\mathbf{n} + \mathrm{Cos}\theta(\mathbf{r} - (\mathbf{n}.\mathbf{r})\mathbf{n}) + (\mathrm{Sin}\theta)\mathbf{n} \times \mathbf{r} \\ &= (\mathrm{Cos}\theta)\mathbf{r} + (1 - \mathrm{Cos}\theta)\mathbf{n}(\mathbf{n}.\mathbf{r}) + (\mathrm{Sin}\theta)\mathbf{n} \times \mathbf{r} \end{aligned} \tag{7-2}$$

下面我们将说明，可以通过四元数的转化来表示矢量 $\mathbf{r}$ 的角位移。换言之，我们提供了一个像矩阵一样的四元数来完成对矢量的转化。

首先必须注意，这样的算子只需要 4 个实数（矩阵需要 9 个元素），我们需要知道：

- 矢量长度的变化；
- 旋转发生的平面（可以通过与两个坐标轴的夹角确定）；
- 旋转的角度。

也就是说，我们需要一种只包含欧拉定理要求的四个自由度的表示方法。由此我们选择使用单位四元数。顾名思义，四元数包括“4 个矢量”而且可以一般化为一个复杂的数来看待，在这个复杂的数中 s 作为实数部分或者标量部分，而 x，y，z 都是作为虚数部分：

$$\begin{aligned} q &= s + x\mathbf{i} + y\mathbf{j} + z\mathbf{k} \\ &= (s, \mathbf{v}) \end{aligned}$$

这里我们可以注意到它跟二维空间中用来表示二维的点或矢量的复杂数的相似之处，一个四元数可以标志一个四维空间中的点，而且如果 $s = 0$，则表示一个三维空间中的点或者矢量。在本节中，它用于表示旋转矢量的一个增量；$\mathbf{i}$，$\mathbf{j}$，$\mathbf{k}$ 都是单位四元数，等价于矢量系统中的单位矢量；但是，它们必须满足一定的约束：

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{ijk} = -1, \mathbf{ij} = \mathbf{k}, \mathbf{ji} = -\mathbf{k}$$

利用这些我们可以推导出加法和乘法法则，其中得出的结果都是四元数：

- 加法法则：

$$q + q' = (s + s', \mathbf{v} + \mathbf{v}')$$

- 乘法法则：

$$qq' = (ss' - \mathbf{v}.\mathbf{v}', \mathbf{v} \times \mathbf{v}' + s\mathbf{v}' + s'\mathbf{v})$$

四元数

$$q = (s, \mathbf{v})$$

的共轭四元数是：

$$\bar{q} = (s, -\mathbf{v})$$

而且四元数与其共轭四元数的乘积决定了四元数的大小：

$$q\bar{q} = s^2 + |\mathbf{v}^2| = q^2$$

如果 q 的绝对值为 1，那么 q 称为单位四元数。所有的单位四元数共同形成了四维空间上的

单位球，而且单位四元数在说明一般的旋转量时能起到很大的作用。

如果

$$q = (s, \mathbf{v})$$

那么存在一个 $\mathbf{v}$ 和一个 $\theta \in [-\pi, \pi]$，使得：

$$q = (\cos\theta, \mathbf{v}'\sin\theta)$$

而且如果 q 是一个单位四元数，那么

$$q = (\cos\theta, \sin\theta\mathbf{n}) \tag{7-3}$$

其中 $\mathbf{n}$ 的绝对值为 1。

下面我们考虑利用四元数对图 7-18 中的矢量 $\mathbf{r}$ 进行操作，$\mathbf{r}$ 被定义为四元数 $p = (0, \mathbf{r})$，并且将操作定义为：

$$R_P(p) = qpq^{-1}$$

也就是说，对以四元数形式表示的矢量 $\mathbf{r}$ 进行旋转，在它的左边乘上 q，右边乘上 q^{-1}，这保证了结果是 $(0, \mathbf{v})$ 形式的四元数（换句话说，一个向量）。q 被定义为一个单位四元数 $(s, \mathbf{v})$。容易得到：

$$R_P(p) = (0, (s^2 - \mathbf{v}.\mathbf{v})\mathbf{r} + 2\mathbf{v}(\mathbf{v}.\mathbf{r}) + 2s(\mathbf{v} \times \mathbf{r}))$$

利用命题 7-3 将上式代入式 7-2，得到：

$$\begin{aligned} Rq(p) &= (0, (\mathrm{Cos}^2\theta - \mathrm{Sin}^2\theta)\mathbf{r} + 2\mathrm{Sin}^2\theta\mathbf{n}(\mathbf{n}.\mathbf{r}) + 2\mathrm{Cos}\theta\mathrm{Sin}\theta(\mathbf{n} \times \mathbf{r})) \\ &= (0, \mathbf{r}\mathrm{Cos}2\theta + (1 - \mathrm{Cos}2\theta)\mathbf{n}(\mathbf{n}.\mathbf{r}) + \mathrm{Sin}2\theta(\mathbf{n} \times \mathbf{r})) \end{aligned}$$

现在将上式与式 7-2 比较，可以发现，除了角度上面的一个因数 2 以外，两个式子在形式上是相同的。从中可以得出结论：在四元数空间中，将矢量 $\mathbf{r}$ 旋转角位移 $(\theta, \mathbf{n})$ 等同于用单位四元数 $(\mathrm{Cos}(\theta/2), \mathrm{Sin}(\theta/2)\mathbf{n})$ 代替角位移，并且同时对四元数 $(0, \mathbf{r})$ 进行 $q(\,)q^{-1}$ 的操作。从而我们可以运用四元数代数法依据四个参数 $\mathrm{Cos}(\theta/2)$，$\mathrm{Sin}(\theta/2)\mathbf{n}_x$，$\mathrm{Sin}(\theta/2)\mathbf{n}_y$，$\mathrm{Sin}(\theta/2)\mathbf{n}_z$ 计算出角色的取向性，继而实现对该部分操作。

下面我们来看一个实际的例子，图 7-19 描绘了一个对象绕初始位置旋转的两种不同路径。第一幅图像运用欧拉角在 x 方向上对旋转量进行插值生成中间部分；第二幅图像对 y 及 z 方向上的旋转量都进行插值以生成中间部分。结果，中间部分的动画序列完全不同，第一幅产生的画面是直接的旋转而第二幅是一个扭曲的旋转。使用四元数的一个原因就是为了避免出现这种不同。

第一幅图像中单个的 x 轴上 π 角度的旋转用四元数：

$$(\mathrm{Cos}(\pi/2), \mathrm{Sin}(\pi/2)(1,0,0)) = (0, (1,0,0))$$

表示；相似地，y 轴上 π 角度的旋转用 $(0, (0,1,0))$ 表示，z 轴上 π 角度的旋转用 $(0, (0,0,1))$ 表示。现在，先发生 y 轴上 π 角度的旋转，再发生 z 轴上 π 角度的旋转的效果图就可以通过单个的四元数给出，这个四元数由上述两个四元数相乘形成：

$$\begin{aligned} (0, (0,1,0))(0, (0,0,1)) &= (0, (0,1,0) \times (0,0,1)) \\ &= (0, (1,0,0)) \end{aligned}$$

这一变换等价于单个的 x 轴上 π 角度的旋转。

最后需要注意，四元数一般专用于方向性的表示——它可以用于位移量的表示，但是将旋转量和位移量联系起来用一种相同的策略来处理并非易事。

四元数插值法

相比较于用欧氏几何的方法确定参数，我们优先考虑用四元数的方法确定参数。这一节

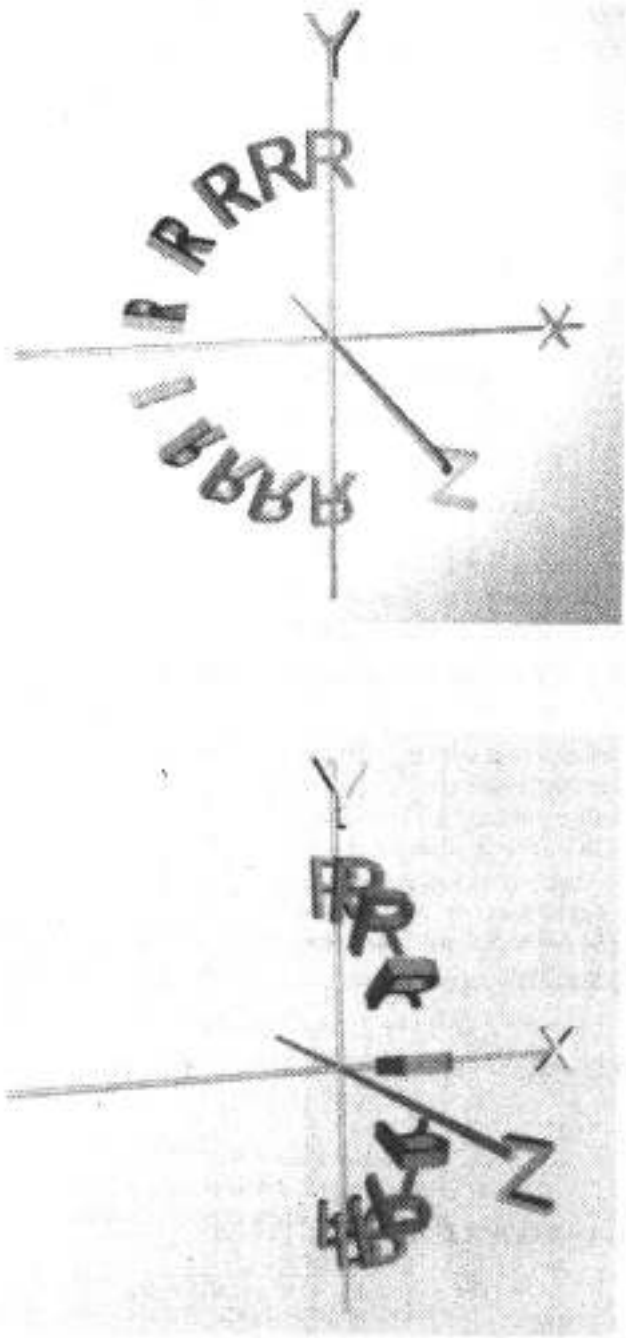


图 7-19 欧氏几何中，由插值法生成的相同关键点之间的不同路径

将讨论如何在四元数空间中进行旋转变量的插值。设想一个动画制作人员坐在工作台前，以所有合适的方法互动地创建一系列的关键方向；这些通常在主要旋转操作中完成，而现在，欧氏几何给动画制作人员带来的束缚可以被解除了，即不再需要以固定顺序为每一个关键点使用固定数目的主要旋转。通常，每个关键点被表示为单一的旋转矩阵，这一系列的矩阵维

而被转换为一系列的四元数。在关键四元数间进行插值的操作生成一系列的中间四元数，然后再将其转换回旋转矩阵，再把矩阵应用到对象上。四元数插值操作动画制作者极容易辩认出。

四元数空间的换进与换出

实现对四元数空间的换进与换出就是要求能够将一个一般的旋转矩阵变换为四元数，并能够将四元数转换回旋转矩阵。现在为了将矢量 p 旋转四元数 q 表示的角度，我们进行如下操作：

$$q(0,p)q^{-1}$$

此处 q 是四元数：

$$(\cos(\theta/2),\sin(\theta/2)\mathbf{n}) = (s,(x,y,z))$$

可以证明，这样的操作等价于为矢量提供一个如下的旋转矩阵：

$$\boldsymbol{M} = \begin{bmatrix} 1-2(y^2+z^2) & 2xy-2sz & 2sy+2xz & 0 \\ 2xy+2sz & 1-2(x^2+z^2) & -2sx+2yz & 0 \\ -2sy+2xz & 2sx+2yz & 1-2(x^2+y^2) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

通过这些方法，我们可以把四元数转化为旋转矩阵。

相反方向的映射（也就是将旋转矩阵转化为四元数）如下所示。要求是将一个普通旋转矩阵：

$$\begin{bmatrix} M_{00} & M_{01} & M_{02} & M_{03} \\ M_{10} & M_{11} & M_{12} & M_{13} \\ M_{20} & M_{21} & M_{22} & M_{23} \\ M_{30} & M_{31} & M_{32} & M_{33} \end{bmatrix}$$

（此处，$M_{03} = M_{13} = M_{23} = M_{30} = M_{31} = M_{32} = 0$ 而 $M_{33} = 1$）转换为上述的矩阵格式。当给定一个旋转矩阵时，首先要做的就是检验对角线元素 M_{ii} 的算术和，也就是：

$$4-4(x^2+y^2+z^2)$$

因为与旋转矩阵相一致的四元数都是单位大小的，我们有：

$$s^2+x^2+y^2+z^2 = 1$$

以及

$$4-4(x^2+y^2+z^2) = 4-4(1-s^2) = 4s^2$$

所以，对于一个 4×4 的齐次矩阵而言，可以得到：

$$s = \pm\frac{1}{2}\sqrt{M_{00}+M_{11}+M_{22}+M_{33}}$$

以及

$$x = \frac{M_{21}-M_{12}}{4s}$$

$$y = \frac{M_{02}-M_{20}}{4s}$$

$$z = \frac{M_{10}-M_{01}}{4s}$$

球形线性插值（slerp）

我们现在来讨论如何进行前文已经简略提到过的四元数空间插值。因为旋转量是与一个单位大小的四元数相匹配的，所以所有的旋转量与四元数空间中的四维单位超球体的表面相匹配。当关键方向上的曲线插值位于球体的表面时，考虑两个关键四元数之间的插值是一种最简单的情况；两个关键四元数之间最简单的线性插值导致它们中间的部分加速运动。图7-20描绘了这一过程在二维空间中的类比情况，图中显示，圆弧表面服从于线性插值法生成的路径，引起了不平衡的角度，从而导致了角速度的加快。

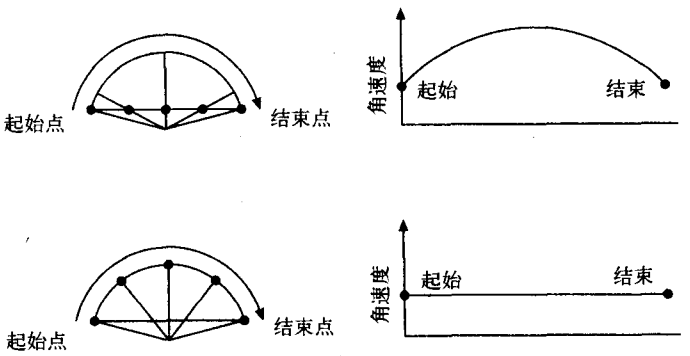


图 7-20 在二维空间中的类比：简单线性插值与简单球形线性插值的差异

这是因为我们并没有沿着超球体的表面向前运动而是抄了近路，为了保证稳定的旋转量，必须使用球形线性插值法（或称为 *slerp*）。此种方法中，我们沿着测地学规定的弓弧在两个关键点间行进。

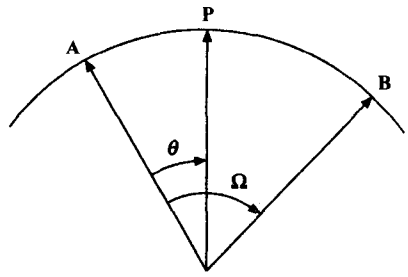


图 7-21 球形线性插值

球形线性插值法的公式在几何上很容易得出。考虑图 7-21 所示的情况：二维空间中，矢量 **A** 和矢量 **B** 之间有大小为 Ω 的夹角，矢量 **P** 与矢量 **A** 之间有大小为 θ 的夹角，**P** 是通过在 **A** 与 **B** 之间球形插值导出的，算式为：

$$\mathbf{P} = \alpha\mathbf{A} + \beta\mathbf{B}$$

一般地，我们可以由下面的关系解出 α 和 β：

$$|\mathbf{P}| = 1$$

$$\mathbf{A}.\mathbf{B} = \mathrm{Cos}\Omega$$

$$\mathbf{A}.\mathbf{P} = \mathrm{Cos}\theta$$

从而得到：

$$\mathbf{P} = \mathbf{A}\frac{\sin(\Omega - \theta)}{\sin\Omega} + \mathbf{B}\frac{\sin\theta}{\sin\Omega}.$$

在两个单位四元数 q_1 和 q_2 之间进行球形线性插值，其中

$$q_1 . q_2 = \cos\Omega$$

为了得出结果，我们需要将上面的式子推广到四维空间中，并且用 Ω 来替换 θ，得到下面的

式子：

$$\mathrm{slerp}(q_1, q_2, u) = q_1 \frac{\sin(1-u)\Omega}{\sin\Omega} + q_2 \frac{\sin\Omega u}{\sin\Omega}$$

其中 $u \in [0, 1]$。

现在考虑任意的两个关键四元数 p 和 q，从测地学的角度讲，存在两条连接它们的弧可以作为行走路径。其中一条比较长，应当避免选择这条弧作为行进路径。可能会有人简单地认为，我可以归纳为在矢量 p 和 q 之间以角度 Ω（其中 $p \cdot q = \cos\Omega$）进行插值或者在相反方向上以角度 $2\pi - \Omega$ 进行插值。然而，这并不能得出希望的效果；这是因为四元超球体取向性的结构并不是欧氏几何中三维球体的简单扩充。为了说明这个问题，我们需要注意到每个旋转量在四元数空间中都有两种表示方法，即 q 和 $-q$。也就是说 q 和 $-q$ 产生的效果是等同的；从而又可以导出代数操作 $q()q^{-1}$ 和 $(-q)()(-q)^{-1}$ 起到的作用是相同的，这点充分说明相反数代表了相同的旋转。因为这种拓扑上的特异性，我们在判定最短弧的时候需要特别地仔细。一种可行的策略是选择或者在四元数对 p 和 q 之间插值，或者在四元数对 p 和 $-q$ 之间插值。对于两个四元数 p 和 q，我们计算它们差值的量，即 $(p-q)(p-q)$；再计算两个四元数 p 和 $-q$ 差值的量，即 $(p+q)(p+q)$。比较这两个量，如果前者小于后者，那么我们已经在沿最短的弧行进，不需要再做什么。若后者比较小，就需要用 $-q$ 替换 q 并且继续运动下去。图 7-22 概略地描绘了这一想法。

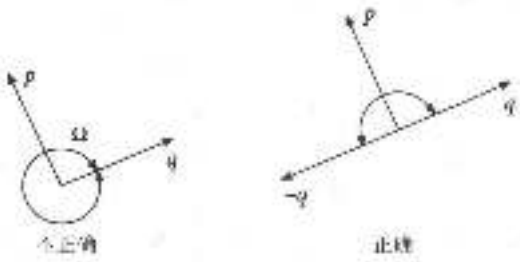


图 7-22　四元数超球体中最短弧的判定

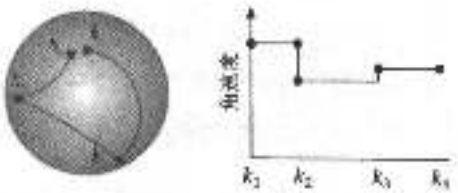


图 7-23　运用 slerp 在四个关键值间插值的二维近似

到目前为止，我们已经描述了球面上两个关键方向之间的球面插值法；而且，和在线性插值中一样，两个以上关键方向之间的球形线性插值会导致关键点动作的剧烈变化。图 7-23 以三维空间中的情况作了类似说明：球体表面的曲线在关键点不连续，图中同样说明关键点的角速度不是连续的值而是离散的值。我们可以在关键点的间隔中加入一定数目的帧，帧的数目与间隔的大小成比例，通过这一方法可以使所有帧中的角速度变为连续的。就是说，我们计算出一对关键点 q_i 和 q_{i-1} 之间的夹角 θ，如下：

$$\cos\theta = q_i \cdot q_{i-1}$$

其中两个四元数 $q = (s, \mathbf{v})$ 和 $q' = (s', \mathbf{v}')$ 的内积定义为：

$$q \cdot q' = ss' + \mathbf{v} \cdot \mathbf{v}'$$

整合出连续的路径是一项更加艰巨的任务，这一更高要求的连续性需要获得立方体样条的球体近似物。遗憾的是，因为我们现在是在一个四维超球体的表面进行操作，所以这个问题远远比在欧氏三维空间中创建样条要复杂。[DUFF86]、[CABR85] 以及 [SHOE87] 都是

用来解决这一问题的。

最后，我们来谈一个应用四元数时可能会出现的难点。四元数插值法的取向是不加区别的，它并不倾向于任何一个方向，两个关键点之间经过插值产生的运动仅仅取决于关键点的取向性；这给设计虚拟的摄影镜头带来一些不便。一般而言，当移动一个镜头时，要求电影的平面是竖直的——通常由一个“向上”矢量来指定。因为这一常识限制，在应用四元数的时候，使用方向倾向性的想法变得很难实现；如果想拥有方向倾向性，那么镜头的“向上”矢量需要重置或者进行其他的一些修改。

附录 7.2 四元数的实现

下面是定义四元数的类的源代码：

```
class FLY3D_MATH_API flyQuaternion
{
public:
   float x,y,z,w;

   // empty constructor
   flyQuaternion() {};
   // constructor from angle and axis
   flyQuaternion(float angle, flyVector &axis);
   // constructor from a rotation matrix
   flyQuaternion(flyMatrix &mat);

   // normalize quaternion
   void normalize();
   // get current quaternion axis and rotation
   void get_rotate(float &angle, flyVector &axis);
   // spherical linear interpolation
   void slerp(flyQuaternion& q1,flyQuaternion& q2,float t);
   // transform the quaternion into matrix
   void get_mat(flyMatrix &mat);

   // quaternion add operator
   flyQuaternion operator +(flyQuaternion &q)
   // quaternion multiply operator
   flyQuaternion operator *(flyQuaternion &q);
};
```

下面的代码从指定的轴和角度构建了一个绕轴旋转的四元数：

```
flyQuaternion(float angle, flyVector &axis)
{
   float f=(float)sin(angle*PiOver180*0.5f);
   x=axis.x*f;
   y=axis.y*f;
   z=axis.z*f;
   w=(float)cos(angle*PiOver180*0.5f);
};
```

下面的代码将 3×3 的旋转矩阵转化为一个四元数，通过这种到四元数空间的转换，使 slerp 得到应用：

```
flyQuaternion::flyQuaternion(flyMatrix &mat)
{
   float tr,s,q[4];
   int i,j,k;
```

```
    int nxt[3] = {1, 2, 0};

    tr = mat.m[0][0] + mat.m[1][1] + mat.m[2][2];

    // check the diagonal
    if (tr > 0.0)
    {
      s = (float)sqrt(tr + 1.0f);
      w = s/2.0f;
      s = 0.5f/s;
      x = (mat.m[1][2] - mat.m[2][1]) * s;
      y = (mat.m[2][0] - mat.m[0][2]) * s;
      z = (mat.m[0][1] - mat.m[1][0]) * s;
    }
    else
    {
      // diagonal is negative
      i = 0;
      if (mat.m[1][1] > mat.m[0][0]) i = 1;
      if (mat.m[2][2] > mat.m[i][i]) i = 2;
      j = nxt[i];
      k = nxt[j];

      s=(float)sqrt((mat.m[i][i]-(mat.m[j][j] + mat.m[k][k])) + 1.0);

      q[i]=s*0.5f;

      if(s!=0.0f) s = 0.5f/s;

      q[3] = (mat.m[j][k] - mat.m[k][j]) * s;
      q[j] = (mat.m[i][j] + mat.m[j][i]) * s;
      q[k] = (mat.m[i][k] + mat.m[k][i]) * s;
      x = q[0];
      y = q[1];
      z = q[2];
      w = q[3];
    }
}
```

下面的代码对四元数进行规范化，使它成为一个单位长度的量：

```
void flyQuaternion::normalize()
{
   float factor = 1.0f/(float)sqrt(x*x+y*y+z*z+w*w);

   x*=factor;
   y*=factor;
   z*=factor;
   w*=factor;
}
```

下面的代码返回当前四元数的坐标轴和旋转量：

```
void flyQuaternion::get_rotate(float &angle, flyVector &axis)
{
   angle=(float)acos(w)*2*PiUnder180;

   float f=(float)sin(angle*PiOver180*0.5f);

   axis.x=x/f;
   axis.y=x/f;
   axis.z=x/f;
}
```

下面的代码在两个四元数间进行插值，并且返回一个代表两个给定四元数之间旋转量的四元数；其中参数 t 必须在 0 到 1 的范围内：

```
void flyQuaternion::slerp(flyQuaternion& q1,flyQuaternion& q2,float t)
{
   float v;      // complement to t
   float o;      // complement to v (t)
   float theta; // angle between q1 & q2
   float sin_t; // sin(theta)
   float cos_t; // cos(theta)
   float phi;    // spins added to theta
   int flip;     // flag for negating q2

   cos_t = q1[0] * q2[0] + q1[1] * q2[1] + q1[2] * q2[2] + q1[3]
* q2[3];

   if (cos_t < 0.0)
   {
        cos_t = -cos_t;
        flip = 1;
   }
   else
        flip = 0;

   if (1.0 - cos_t < 1e-6)
   {
     v = 1.0f - t;
        o = t;
   }
   else
   {
     theta = (float)acos(cos_t);
     phi = theta;
     sin_t = (float)sin(theta);
     v = (float)sin(theta - t * phi) / sin_t;
     o = (float)sin(t * phi) / sin_t;
   }
   if (flip) o = -o;

   x = v * q1[0] + o * q2[0];
   y = v * q1[1] + o * q2[1];
   z = v * q1[2] + o * q2[2];
   w = v * q1[3] + o * q2[3];
}
```

下面的代码将四元数转换回 3×3 的旋转矩阵：

```
void flyQuaternion::get_mat(flyMatrix &mat)
{
   float wx, wy, wz, xx, yy, yz, xy, xz, zz, x2, y2, z2;

   // calculate coefficients
   x2 = x + x;
   y2 = y + y;
   z2 = z + z;
   xx = x * x2;
   xy = x * y2;
   xz = x * z2;
   yy = y * y2;
   yz = y * z2;
   zz = z * z2;
```

```
    wx = w * x2;
    wy = w * y2;
    wz = w * z2;

    mat.m[0][0] = 1.0f-(yy+zz);
    mat.m[1][0] = xy - wz;
    mat.m[2][0] = xz + wy;
    mat.m[3][0] = 0.0;

    mat.m[0][1] = xy + wz;
    mat.m[1][1] = 1.0f-(xx + zz);
    mat.m[2][1] = yz - wx;
    mat.m[3][1] = 0.0;

    mat.m[0][2] = xz - wy;
    mat.m[1][2] = yz + wx;
    mat.m[2][2] = 1.0f - (xx + yy);
    mat.m[3][2] = 0.0;

    mat.m[0][3] = 0;
    mat.m[1][3] = 0;
    mat.m[2][3] = 0;
    mat.m[3][3] = 1;
}
```

下面的代码实现了两个四元数的相乘：

```
flyQuaternion flyQuaternion::operator *(flyQuaternion &q)
{
    float A, B, C, D, E, F, G, H;
    flyQuaternion res;

    A = (q.w + q.x)*(w + x);
    B = (q.z - q.y)*(y - z);
    C = (q.w - q.x)*(y + z);
    D = (q.y + q.z)*(w - x);
    E = (q.x + q.z)*(x + y);
    F = (q.x - q.z)*(x - y);
    G = (q.w + q.y)*(w - z);
    H = (q.w - q.y)*(w + z);

    res.w = B + (-E - F + G + H) /2;
    res.x = A - (E + F + G + H)/2;
    res.y = C + (E - F + G - H)/2;
    res.z = D + (E - F - G + H)/2;

    return res;
}
```

下面的代码实现了两个四元数的相加：

```
flyQuaternion operator +(flyQuaternion &q)
{
    flyQuaternion res;
    res.x=x+q.x;
    res.y=y+q.y;
    res.z=z+q.z;
    res.w=w+q.w;
    return res;
};
```

附录 7.3 角色动画中效率的考虑

角色的动画制作是游戏中最依赖于 CPU 的操作之一。为了得到流畅的画面而对动画中的关键点进行插值的工作，对不同动画画面进行合成的工作，都需要占用 CPU 大量的工作时间。因为是对角色模型中每个顶点进行确定的计算，我们可以使用 SIMD（单指令多数据流）Pentium3 系统指令集来并行地实现对 x、y 和 z 顶点分量的操作，从而加快运算速度。在这个附录中，我们主要考虑使用 SIMD Pentium3 系统指令集运行本章中主要部分的代码的效率。

对齐内存的分配

SIMD Pentium3 系统指令集需要对数据进行对齐（矢量的 x 分量必须对齐，使得它的地址值可以被 16 整除）。我们只能通过使用能够分配和释放 128 位对齐内存的专用程序来确定所有的数据都已经对齐，同时也必须确定所使用结构的大小能够被 16 整除，从而使每个数组项都可以对齐。代码如下：

```
void *aligned_alloc(int n)
{
   char *data=new char[n+20];
   unsigned t=(unsigned)data+4;
   t=((t&0xf)==0?0:16-(t&0xf));
   *((unsigned *)&data[t])=t;
   return &data[t+4];
}

void aligned_free(void *data)
{
   if (data==0)
      return;
   int t=*(((unsigned *)data)-1);
   char *d=(char *)data;
   d-=t+4;
   delete d;
}
```

对于未对齐的数组：

```
flyVector *vecs=new flyVector[nvec];
flyMatrix *mats=new flyMarix[nmat];
flyVertex *verts=new flyVertex[nvert];

delete[] vecs;
delete[] mats;
delete[] verts;
```

对于已经对齐的数组：

```
flyVector *vecs=(flyVector *)aligned_alloc(sizeof(flyVector)*nvec);
flyMatrix *mats=(flyMarix *)aligned_alloc(sizeof(flyMarix)*nmat);
flyVertex *verts=(flyVertex *)aligned_alloc(sizeof(flyVertex*nvert);

aligned_free(vecs);
aligned_free(mats);
aligned_free(verts);
```

顶点网格动画中关键点的插值

在动画中为一个浮点键值的关键点的网格顶点插值（在两个关键点间进行线性插值）。

```
void flyAnimatedMesh::set_state(int anim, float key_factor)
{
   int i,j,k;
   float s;
   flyVertex *v0;
   flyVector *v1,*v2;

   // compute local key factor between the two
   // animation keys to be interpolated (key_factor)
   j=(int)(key_factor*animkeys[anim]);
   if (j==animkeys[anim])
      { j=0; key_factor=0.0f; }
   s=1.0f/animkeys[anim];
   key_factor=(key_factor-j*s)/s;

   // find vertex pointers for the two
   // animation keys to be interpolated (v1 and v2)
   v1=&key_verts[(animkeyspos[anim]+j)*nv];
   if (j==animkeys[anim]-1)
      k=0;
   else k=j+1;
   v2=&key_verts[(animkeyspos[anim]+k)*nv];

   v0=localvert;// output vertex array

   s=1.0f-key_factor; // compute 1-factors

#ifdef INTEL_SIMD
   if (g_processor_features&_CPU_FEATURE_SSE && g_flyengine->sse)
   {
      flyVector f1(s);
      flyVector f2(key_factor);
      i=nv*0x40;
      __asm
      {
         // xmm2=(s,s,s,s)
         // xmm3=(key_factor,key_factor,key_factor,key_factor)
         movups xmm2,f1
         movups xmm3,f2

         // init loop registers (ebx->flyVector, ecx->flyVertex)
         mov ebx,0
         mov ecx,0
      L1:
         // xmm0=v1[ebx]
         mov eax,v1
         movaps xmm0,[eax+ebx]

         // xmm0=v2[i]
         mov eax,v2
         movaps xmm1,[eax+ebx]

         // interpolate position to xmm0
         mulps xmm0,xmm2
         mulps xmm1,xmm3
         addps xmm0,xmm1

         // v0[i]=xmm0
         mov eax,v0
         movaps [eax+ecx],xmm0
         // increment loop registers
```

```
        add ebx,0x10
        add ecx,0x40

        // loop until all vertices are processed
        cmp  ecx, i
        jl L1
      }
    }
    else
#endif
    for( i=0;i<nv;i++)
    {
      v0->x = v1->x*s + v2->x*key_factor;
      v0->y = v1->y*s + v2->y*key_factor;
      v0->z = v1->z*s + v2->z*key_factor;
      v0++; v1++; v2++;
    }
```

顶点网格动画中插值的合成

对两个位于不同动画中的关键点间的网格顶点进行插值，先对其中一副动画中的网格顶点线性插值，再对另一副动画中的网格顶点线性插值，最后对两个结果线性插值，得出最终结果（一共进行了3次线性插值）。

```
void flyAnimatedMesh::set_state(int anim1, float key_factor1, int
anim2, float key_factor2, float blend)
{
   int i,j,k;
   float s,t,w;
   flyVertex *v0;
   flyVector *v1,*v2,*v3,*v4;

   // compute local key factor between the first
   // animation keys to be interpolated (key_factor1)
   j=(int)(key_factor1*animkeys[anim1]);
   if (j==animkeys[anim1])
     { j=0; key_factor1=0.0f; }
   s=1.0f/animkeys[anim1];
   key_factor1=(key_factor1-j*s)/s;

   // find vertex pointers on the first animation for
   // the two keys to be interpolated (v1 and v2)
   v1=&key_verts[(animkeyspos[anim1]+j)*nv];
   if (j==animkeys[anim1]-1)
     k=0;
   else k=j+1;
   v2=&key_verts[(animkeyspos[anim1]+k)*nv];

   // compute local key factor between the second
   // animation keys to be interpolated (key_factor2)
   j=(int)(key_factor2*animkeys[anim2]);
   if (j==animkeys[anim2])
     { j=0; key_factor2=0.0f; }
   s=1.0f/animkeys[anim2];
   key_factor2=(key_factor2-j*s)/s;

   // find vertex pointers on the second animation for
   // the two keys to be interpolated (v3 and v4)
   v3=&key_verts[(animkeyspos[anim2]+j)*nv];
   if (j==animkeys[anim2]-1)
     k=0;
```

```
    else k=j+1;
    v4=&key_verts[(animkeyspos[anim2]+k)*nv];

    v0 = localvert; // output vertex array

    s=1.0f-key_factor1; // compute 1-factors
    t=1.0f-key_factor2;
    w=1.0f-blend;

#ifdef INTEL_SIMD
    if (g_processor_features&_CPU_FEATURE_SSE && g_flyengine->sse)
    {
      flyVector f1(s);
      flyVector f2(key_factor1);
      flyVector f3(t);
      flyVector f4(key_factor2);
      flyVector f5(w);
      flyVector f6(blend);
      i=nv*0x40;

      __asm
      {
        // xmm2=(s,s,s,s)
        // xmm3=(key_factor1,key_factor1,key_factor1,key_factor1)
        // xmm2=(t,t,t,t)
        // xmm3=(key_factor2,key_factor2,key_factor2,key_factor2)
        movups xmm4,f1
        movups xmm5,f2
        movups xmm6,f3
        movups xmm7,f4

        // init loop registers (ebx->flyVector, ecx->flyVertex)
        mov ebx,0
        mov ecx,0
      L1:
        // xmm0=v1[i]
        mov eax,v1
        movaps xmm0,[eax+ebx]

        // xmm1=v2[i]
        mov eax,v2
        movaps xmm1,[eax+ebx]

        // interpolate first anim to xmm0
        mulps xmm0,xmm4
        mulps xmm1,xmm5
        addps xmm0,xmm1

        // xmm2=v3[i]
        mov eax,v3
        movaps xmm2,[eax+ebx]

        // xmm3=v4[i]
        mov eax,v4
        movaps xmm3,[eax+ebx]

        // interpolate second anim to xmm2
        mulps xmm2,xmm6
        mulps xmm3,xmm7
        addps xmm2,xmm3

        // blend the two anims to xmm0
```

```
            movups xmm1,f5
            movups xmm3,f6
            mulps xmm0,xmm1
            mulps xmm2,xmm3
            addps xmm0,xmm2

            // v0[i]=xmm0
            mov eax,v0
            movaps [eax+ecx],xmm0
            // increment loop registers
            add ebx,0x10
            add ecx,0x40

            // loop until all vertices are processed
            cmp ecx, i
            jl L1
        }
    }
    else
#endif
    for( i=0;i<nv;i++ )
    {
            v0->x=(v1->x*s+v2->x*key_factor1)*w+
                  (v3->x*t+v4->x*key_factor2)*blend;
            v0->y=(v1->y*s+v2->y*key_factor1)*w+
                  (v3->y*t+v4->y*key_factor2)*blend;
            v0->z=(v1->z*s+v2->z*key_factor1)*w+
                  (v3->z*t+v4->z*key_factor2)*blend;
            v0++; v1++; v2++; v3++; v4++;
    }
}
```

骨架网格顶点的计算

在骨架动画插值的最后（运用四元数方法），我们需要根据骨架中骨头的表示矩阵以及顶点的权重和偏移矢量来生成各顶点的位置（一个顶点可能受到多块骨头的表示矩阵的影响）。

```
void flySkeletonMesh::build_mesh()
{
    int i,j;
    int p=0;
#ifdef INTEL_SIMD
    if (g_processor_features&_CPU_FEATURE_SSE && g_flyengine->sse)
    {
        flyVertex *v1=localvert;
        flyVector *v2=vweight;
        flyMatrix *m1=m;
        int *i1=vindex;
        flyVector f1(0);

        // for all vertices
        for( i=0;i<nv;i++ )
        {
            j=nvweights[i];
            __asm
            {
                // xmm0=(0,0,0,0)
                movups xmm0,f1
```

```
    // ebx=&vindex[p]
    mov ebx,p
    add ebx,i1

    // initialize loop register
    mov ecx,0
L1:
    // compute matrix memory pos
    // for vindex[p]
    mov eax,[ebx]
    sal eax,0x6
    add eax,m1

    // load matrix m[vindex[p]] into xmm4-7
    movaps xmm4,[eax]
    movaps xmm5,[eax+16]
    movaps xmm6,[eax+32]
    movaps xmm7,[eax+48]

    // xmm3=v2
    mov eax,v2
    movaps xmm3,[eax]

    // multiply first matrix line
    // and accumulate into xmm0
    movaps xmm1,xmm3
    shufps xmm1,xmm1,0x00
    mulps xmm1,xmm4
    addps xmm0,xmm1

    // multiply second matrix line
    // and accumulate into xmm0
    movaps xmm1,xmm3
    shufps xmm1,xmm1,0x55
    mulps xmm1,xmm5
    addps xmm0,xmm1

    // multiply third matrix line
    // and accumulate into xmm0
    movaps xmm1,xmm3
    shufps xmm1,xmm1,0xAA
    mulps xmm1,xmm6
    addps xmm0,xmm1

    // multiply fouth matrix line
    // and accumulate into xmm0
    movaps xmm1,xmm3
    shufps xmm1,xmm1,0xFF
    mulps xmm1,xmm7
    addps xmm0,xmm1

    // increment loop registers
    add v2,0x10
    add ebx,0x4
    // loop until all matrices influencing
    // this vertex are processed
    inc ecx
    cmp ecx,j
    jl L1

    // v1=xmm0, v1++
```

```
            mov eax,v1
            movaps [eax],xmm0
            add v1,0x40

            // store current p
            sub ebx,i1
            mov p,ebx
          }
        }
      }
      else
#endif
      for( i=0;i<nv;i++ )
      {
        localvert[i].null();
        for( j=0;j<nvweights[i];j++,p++ )
          //   vertex = (matrix * vector) * float
          localvert[i] += (m[vindex[p]]*vweight[p])*vweight[p].w;
      }
    }
```

第 8 章　动画成形方法

8.1　简介

动画形状变化，在计算机图形学中最初被称为软对象动画（soft object animation），是最令人头痛的计算机动画问题之一。它固有的难度来自于它的表面复杂性与动画的简易性之间的矛盾。为了真实性和进行有效的渲染，我们使用有大量三角形的三角形网格。在包含低层次但复杂的表面模式下，进行形状动画是相当困难的。一个低层次的模型要求我们在时间的函数里面调整所有顶点的位置。因此在动画制作中，我们要寻找高层次的控制，而其中涉及的表面函数却显然会影响动画的难度和其他方面。在这一章中我们将会看到形状动画的各种通用的方法。在下一章，我们将会考察一个这种类型的动画的主要应用——面部表情和视觉语音（visual speech）。

像关节结构动画一样，我们对面部动画可以采用运动捕捉（MoCap）的方法，这也是应用在游戏中的主流方法。但是 MoCap 有其固有的不足：即只有预先记录的动画才可以实时激活。这就会剔除由文本生成器驱动的面部动画的主要应用——视觉语音。因为我们不能提前知道文本生成器会构造出什么句子。

在我们开始之前，先解决一个常识性的问题：能否用常规的顶点动画法将动作关键帧化生成软对象动画？回答是肯定的，顶点动画是一种可行的、较为常用的方法。但是，它不可能继续作为一个受欢迎的通用模型。再次考虑面部动画的例子。如果有两个姿态代表两个面部表情，我们可以通过插值来生成中间序列，然而，这样的一个序列不可能精确地模仿真实的生成序列。这意味着若考虑面部表情的动力学性质和它们的复杂性及细节，那么要使两个姿态对应脸上每一点在相同间距下的等速率移动几乎是不可能的。

在这一章我们将看到用于变形和改变对象形状的主要技术。它们大多可被认为是两层模型。底层是一个详细模型，最终呈现为三角形和多边形网格；在它上面是一个围绕这个详细模型且允许变形控制的框架。这个框架层的关键在于，它是由编辑器或者脚本机制操作和控制的。通常这个详细模型可包含任意数目的顶点，编辑框架层则显示详细层性质或使其按编辑器理解的方式变形。经典的 FFD 方法是体现这个概念的很好的例子。Sederburg 在［SEDE86］中做了如下的类比：

> 可以将 FFD 类比为一个清晰、灵活、嵌有一个或者多个需要变形的对象的平行六面体可塑型，这样的类比是贴切的。对象被设想成灵活的，它可以随着围绕它的可塑型发生变形。

这里灵活的可塑型是框架层。一个艺术家或者动画制作者用这个可塑型来工作，使被嵌入的具有任意复杂性的网格对象按照预定的方式随着可塑型发生变形。

另外一个模型是常规的皮肤和（skin summation）（参见第 7 章和这一章的 8.6.2 节），这个模型的很多变形方法是同等有效的。等式：

$$\mathrm{x}'_i = \sum_{i=0}^{n} \mathbf{M}_i \mathbf{d}_i w_i$$

是移动一个顶点 x_i 到一个新的顶点 x'_i——通过与它连接的一定数目骨骼来求和。在常规蒙

皮操作里，w_i 是由动画制作者设定的；在 FFD 方法里面则由定义解析。这个等式强调：在最底层，我们最终要根据一个顶点和一个更高层框架的关系移动该顶点，在皮肤和模式中，这个框架恰好是骨架。

绝大多数当前的形状动画方法符合两级层次。上层是变形模型，它接受一个动画或者自动控制，低层是对象的顶点运动变形脚本。以下是两个例外：

1）在变形器层中的一个实体与在对象上的许多实体（顶点）相一致。

2）变形器实体具有一个结构或者连贯性，这样对象形状变换可以很容易地脚本化。

为满足这两个需要，应用较普遍的模型是：

1）样条框架。目前是一个离线造型技术，变形器层次包含与艺术家交互的 B 样条线框结构。由此可以通过转换框架为面片和子划分产生一个底层模型。

2）骨骼或者皮肤。在第 7 章和这一章都做了讨论。变形器是一个包含关节动画的简化的人体骨架。皮肤顶点和骨骼（通常多于一个）相关，且每个相关联的、由权重控制的骨骼在一个顶点上相互影响。

3）FFD。这里变形器结构是关于一个控制点的 3D 数组，这些点控制了三立方贝济埃面片或体。在对象上的顶点是和控制点结合的，且在控制点控制面片变形时与面片一起运动。在最底层，该方法等价变为骨骼模型，且顶点通过权重来移动控制点。

4）假肌肉模型（第 9 章）。这种方法里受高层参数控制的肌肉模型和对象顶点结合。动画脚本控制了收缩肌肉的参数，肌肉在运动时牵动其关联的顶点。

5）线框。变形结构是通过移动其控制点来实现动画脚本对 3D 参数空间曲线的控制。这样的曲线和对象上顶点相结合，它们的当前位置和初始位置之间的差异常被用来牵动它们预定义作用域周围的顶点。

6）包裹材料。这里变形结构是自身的低分辨率的版本。这种方法与我们的二层模型完全一致。

变形技术可扩展至离线和实时应用。在完成一个设计的刻画，且不改变对象形状时，艺术家会使用建模工具，选择刻画一系列关键的姿态，这些姿态在游戏运行的时候是实时替换的。我们最终感兴趣的是这些实时技术的运用。特别地，我们想实现游戏事件驱动动画，以此替换艺术家预先刻画好的变形动画，使基于游戏事件的变形技术在低层模型中起作用。比如肘关节问题。这里，我们可以将这个关节封装在一个 FFD 中，这个 FFD 将对皮肤的所有顶点都给予合适的配置。一个更加复杂的例子是视觉语音，涉及控制下巴和嘴的运动使之与话语合成器发出的话语一致。底层话语单元（visemes）将控制面部以正确的方式变形。

在下文中我们首先介绍确定的变形技术，我们不区分那些目前用在离线刻画的技术和那些延伸到实时应用中的技术。在这两种情况下，各种方法的要求都是一致的，即我们需要由艺术家手工控制的或由游戏逻辑自动控制的框架层。也就是说，我们会在游戏实时自动操作控制下的交互和建模模式里面描述这些方法。接下来我们将会看到在游戏应用里面，最重要的两个问题是骨架控制和面部动画中的皮肤变形。由于较之几何变形模型更多地涉及到面部动画，我们将专门用一章来介绍它。

8.2　样条框架

之所以从这个方法开始，是因为它普遍应用在离线的造型或建模软件中，虽然所有的方

法都可以容许动画脚本。高级框架是一个 B 样条直角网络，这些 B 样条被理解为形成双三次参数化面片网络的边界线的一种低级表达方式（见图 8-1，彩页中也有）。

运用样条结构的艺术作品，经单独地调整，直到呈现为面片网格表示的所需形状。不像这一章描述的其他两个方法，这种变形模型实际上嵌入低级模型——即通过面片网格来控制。然而，面片网格可以被细分到任何级别和任何程度，这改进了别的模型的不足。这种技术给似乎是成功也较受欢迎，因为它给艺术家提供类学接口，而他们则把面片表面视作可移动的框架。例如，一个艺术家构建一张脸，通过一个样条的描述形成头部剖面的闭合回线。互动地构建样条曲线是一个 3D 艺术家必用的技术，同时，把形成面片边界的网格曲线转换为用于渲染和进一步操作的面片网格编程技术对 3D 艺术家来说则更为简单易懂（例如，参见［WATT 92］）。

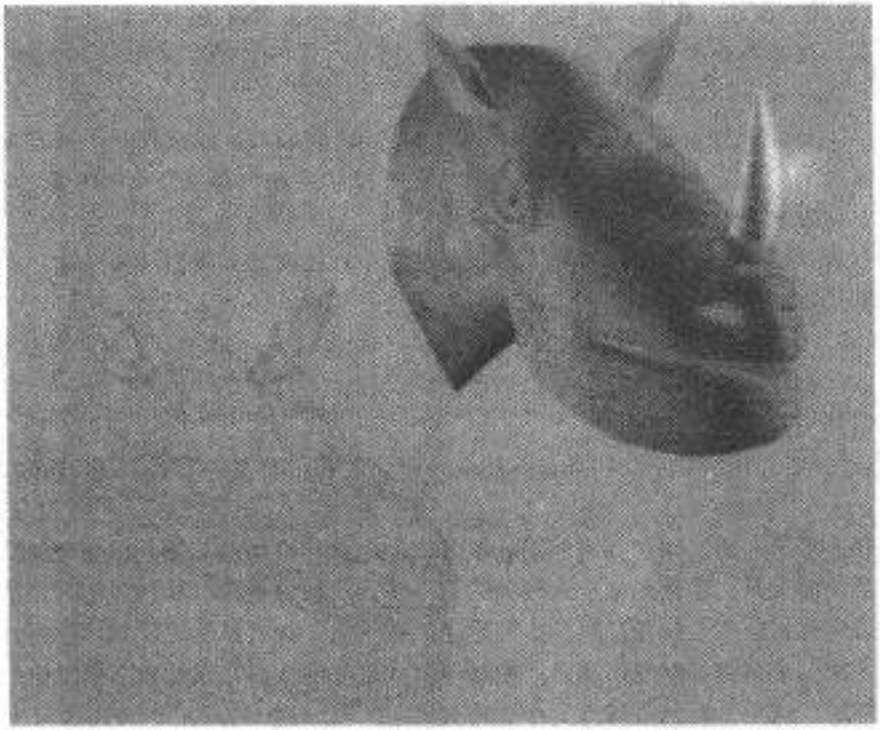


图 8-1　在 3D Studio MAX 中用样条框架建模

样条框架模型实质上是创建和控制面片网格的形状的方法。几年来许多技术已经考察这层用于面片网格低级控制的框架，它意味着用时间函数来管理控制点的位置。虽然其中很多方法都很有前景，但样条框架模型是惟一已经成为主流的技术。在第 9 章我们将看另外一种技术——用于面部动画的层次 B 样条。如前所述，这是一种有效的多层技术。

8.3　自由形状变形

虽然贝济埃建议将双三次参数化面片扩展为有二个或者更多参数的函数，但是这种计算机图形学技术最初是 Sederberg［SEDE86］提出的，他称此为自由形状变形（FFD）。Sederberg 的突破是他实现了任何表面对象模型的变形，例如，三角网格可以通过嵌入参数化面片空间

或运用某个所谓“被期待的”方法实现变形，因为面片可以通过控制点运动而变形。

我们可以定义一个三立方贝济埃面片为：

$$Q(u,v,w)=\sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3}\mathbf{P}_{ijk}B_i(u)B_j(v)B_k(w) \tag{8-1}$$

这里：

$\mathbf{P}_{ijk}$是一个 4×4×4 的控制点构成的点阵

$B_i(u)=B_j(v)=B_k(w)$是贝济埃基函数：

$$B_0(u)=(1-u)^3$$
$$B_1(u)=3u(1-u)^2$$
$$B_2(u)=3u^2(1-u)$$
$$B_3(u)=u^3$$

这个实体是一个平坦面片的同积等价物——这里有 16 个控制点位于这个平坦面片上。现在如果我们在点阵中移动这些点使体积变形。表面变形的 FFD 模型是一个三步方法：

1）在三立方体里面嵌入一个多边形网格对象。我们代入顶点在点阵空间的坐标来将变形表示出来。如果 **x** 是顶点位置，我们就有

$$\mathbf{x}=\mathbf{x}_0+u\mathbf{u}+v\mathbf{v}+w\mathbf{w}$$
$$u=\mathbf{u}.(\mathbf{x}-\mathbf{x}_0)$$
$$v=\mathbf{v}.(\mathbf{x}-\mathbf{x}_0)$$
$$w=\mathbf{w}.(\mathbf{x}-\mathbf{x}_0)$$

这里 $\mathbf{x}_0$ 定义 **uvw** 空间的原点。

2）通过移动控制点使点阵变形，每一点 **x** 迁移到了新的位置 **x′**。

3）**x′**的值通过式 8.1 用 **x** 的（u，v，w）坐标及新的控制点的位置来获得。

FFD 的特殊优点是点阵点的数目可以比对象点的数目要少得多，通常实际情况也是这样的。实际上 FFD 可以看作接受对它的控制点的线性变换且将变换传递到要变形的对象顶点上去的设备。虽然 FFD 提供了一个良好的变形模型，但是它们也有不足。实现所需全部形变的步骤是复杂的，通过变换一个 FFD 点阵来实现一个形变就很困难，不同形变之间又有相互影响，因此全部形变的实现就更困难了。虽然这种方法可以很好地处理一个铰链或者皮肤形变，但是针对高复杂性的形变，我们需要密集控制点阵，而且在骨骼模型控制下移动顶点可能较为容易。通常在控制点数组的矩形性质和要变形表面之间没有关系。这点在 8.6.1 节的 SOFFD 方法中做了说明，且在某种程度上下一节的 EFFD 也做了说明。

在图 8-2 里，FFD 包围了脸的一部分。64 个点的点阵发生了形变，正如图上所示的网格变形结果。

虽然 FFD 是非常流行的建模方法，但在涉及形状动画游戏逻辑的实时控制时，仍需要仔细考虑 FFD 的建模方式。虽然借助艺术家提前建立的模型，我们能很容易地保存变形运动，但是这多少减少了 FFD 建模方法的一般适用性。我们通过把曲线勾勒到点阵上来调用潜在有用的形变运动，进而达到连续的实时控制。这种方法在[WATT92]中做了详细的描述。例如，为了建立围绕一个轴的连续弯曲点阵形变我们可以使用两个控制函数——一个控制弯曲形状的点阵空间控制函数 f_{θ_0}和一个控制变形运动动画的时间控制函数f_{θ_t}。在图 8-3

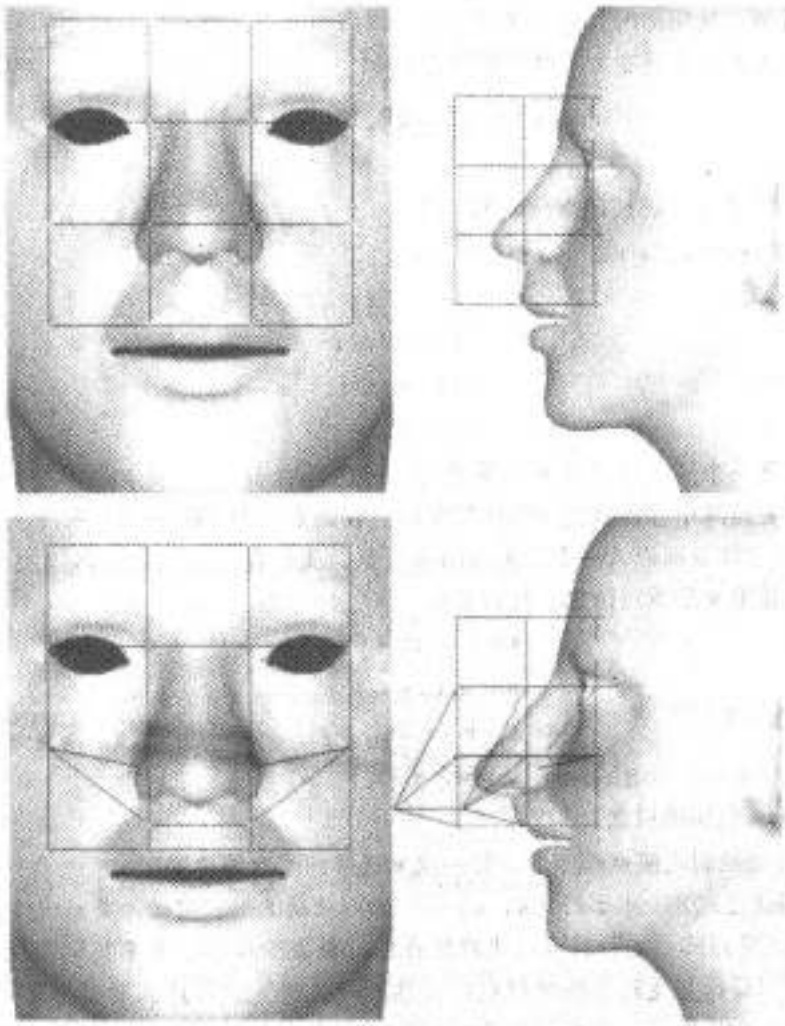


图 8-2　用在面部网格的鼻子区域上的 FFD

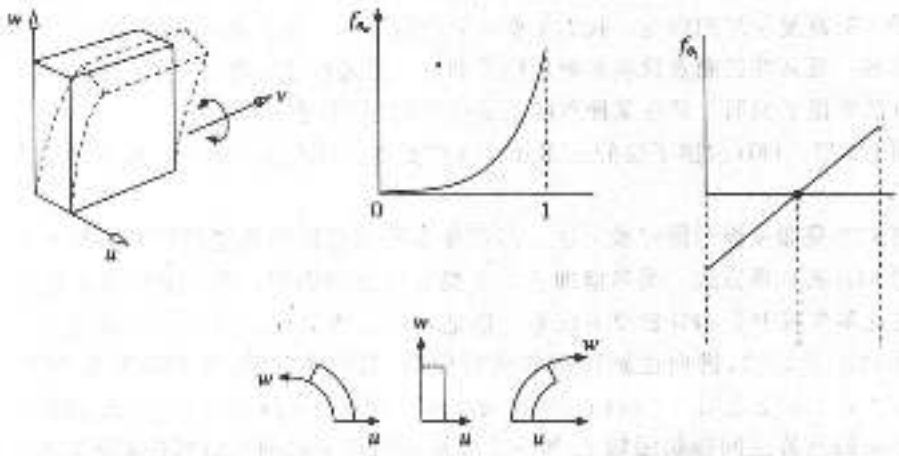


图 8-3　利用 FFD 上点阵点的动画

中表达式：

$$\delta = \theta_1 f_{s_u}(w) f_{s_v}(v)$$

引起点阵围绕 w 轴弯曲，正如图中所示。

FFD 变形是当点阵包围整个对象时最容易考虑的——换句话说就是全局变形。如图 8-2 所示，它们能够应用在局部，但是运用 8.2 节描述的方法可能更合适。

8.4　扩展自由形状变形（EFFD）

作为 FFD 的发展，Coquillart [COQU90] 引入这种方法来解决平行六面体形点阵引入的变形约束。这种方法叫做扩展 FFD 或者 EFFD，包括了一组有特殊结构的体的 FFD 集合。图 8-4 给出了一个由 6 个基本 FFD 体构成的柱状结构例子。如图所示，这一排的 FFD 两端面连接起来，这样所有的 u 轴都重合了。这个柱状单元现在完全由控制点的位置来定义。例如，所有位于 u 轴上满足 $u=0$ 和 $w=0$ 控制点重合了，所有满足 $u=1$，$w=0$ 的控制点也是这样。

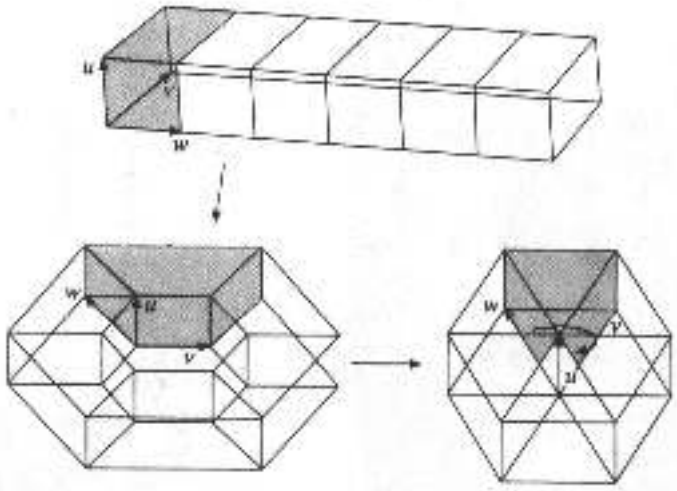


图 8-4　圆柱状 EFFD 的构建

一旦所需的体构造好了，就可以和 FFD 一样以相同的方式使用。虽然现在对象上 $\mathbf{x}-\mathbf{u}$ 点和点阵点不再有简单关系，但是 EFFD 组成元素只不过是 FFD 的变形。为了找到 $\mathbf{x}$ 的坐标 (u, v, w)，我们必须进行如下处理：

1）找含有 $\mathbf{x}$ 的单元。

2）用牛顿迭代的方法解

$$\mathbf{x} = Q(u, v, w) = \sum_{i=0}^{3}\sum_{j=0}^{2}\sum_{k=0}^{3} \mathbf{P}_{ijk} B_i(u) B_j(v) B_k(w)$$

这里 $\mathbf{P}_{ijk}$ 是在 EFFD 结构中的控制点位置。

在图 8-5 中，脸受到初始 EFFD 的支配。在这张插图中，方程由对一个 FFD 的交互式变形，并且用牛顿迭代方法对网格顶点和点阵点之间的一致性求解来建立。现在当 EFFD 被变形，网格顶点的运动结果是这个 EFFD 的非平行六面体形状的函数。

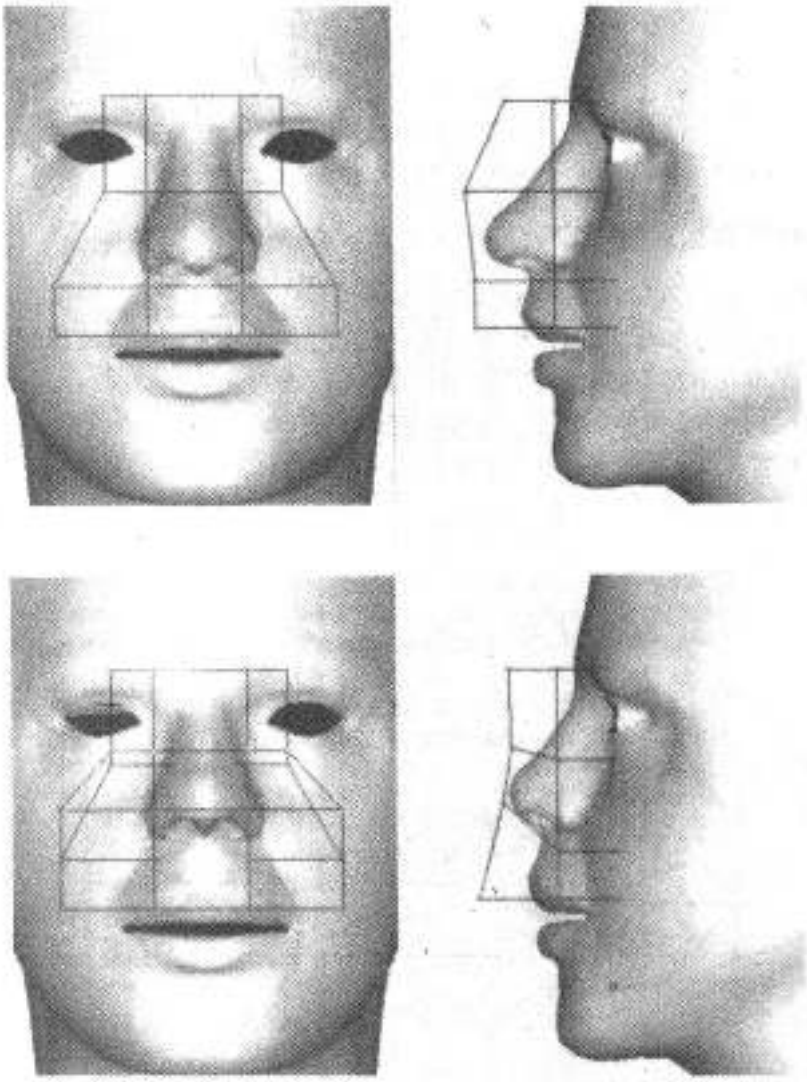


图 8-5　EFFD 变形鼻子区域（与图 8-2 比较）

8.5　曲线变形——铰线

下两个技术是最近才出现的，都门用一类隐式函数和一个作用域与引用对象可变物（曲线或者表面）的相对运动。铰线［SING98］是在 Alias Wavefront 公司的建模和动画软件 Maya 中所用的一种建模工具。这个方法具有和 B 样条构架思想——一个铰线构架可以建立用雕刻家的骨架（armature）的方法来围绕一个表面——共同的想法。它也在概念上和 CAGD 的一些概念相似。在 CAGD 里控制点集合也是被一条靠近表面的曲线（一个最小的正方形使得曲线和控制点相联系）所控制的。当曲线被移动时，控制点也随之移动并且表面发生变形。铰线非同寻常的活动会在一个组里面影响对象的一些区域。和 FFD 相似，这种方法也不假设任何基础表示——它可以为参数曲面或者一个多边形网格移动那些被控制的点。这个方法

和 FFD 的另一个相似性是，它们都运用一条标准曲线在对象和铰线之间产生一个初始绑定，并通过初始曲线和铰线之间的相对运动来控制变形（正如在初始 FFD 或者 EFFD 和变形后的 FFD 之间的运动控制嵌入格子的对象变形）。

铰线定义为一个元组（W，R，s，r，f）：

- W 和 R 是自由形态参数空间曲线——W 是铰曲线而 R 是一条参考曲线。W 和 R 最初是一致的。
- s 是一个控制曲线的径向缩放比例的标量。
- r 是一个曲线的影响半径。
- f 是一个密度或者域函数。这是单调递减函数且有

$$f(0)=1, f(x)=0 \text{ 当 } x \geqslant 1$$

当一条铰线与一个对象绑定时，就决定了一个作用体。根据密度函数 f：

$$F(\mathbf{x}, R)=f\left(\frac{\| \mathbf{x}-R(u_{\min}) \|}{r}\right)$$

这些区域里的点受影响。

这里：

$u_{\min}$是参数 u 的值，它最小化 R 和 $\mathbf{x}$ 的距离。

当 W 移动的时候，W 和 R 之间的差和 s 一起被用来影响对象点 $\mathbf{x}$。那些最初在指定的曲线 R 上的对象点和 W 恰好一起运动，而偏移体外的对象根本就不会移动。$F(\mathbf{x}, R)$平滑地控制在这两个极端之间的点的运动的衰减。R 的作用域内的形变被分割为 3 类——缩放、旋转还有平移。我们考虑一个点 $\mathbf{x}$ 怎样移动到图 8-6 所示的 $\mathbf{x}^{\text{def}}$。在图中点 $R(\mathbf{x}_R)$是 R 上离 $\mathbf{x}$ 最近的点，$W(\mathbf{x}_R)$是 W 上的相应点。

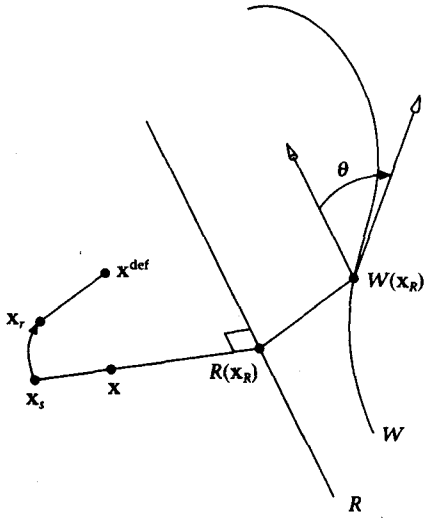


图 8-6　用在铰线上的符号

均匀缩放——$\mathbf{x}$ 移动到 $\mathbf{x}_s$：

$$\mathbf{x}_s=\mathbf{x}+(\mathbf{x}-R(\mathbf{x}_R))(1+(s-1)F(\mathbf{x}, R))$$

旋转——角度 θ 定义为切向量$W'(\mathbf{x}_R)$和$R'(\mathbf{x}_R)$之间的角度。这角度应用于 $\mathbf{x}_s$，围绕点 $R(\mathbf{x}_R)$轴 $W'(\mathbf{x}_R)\times R'(\mathbf{x}_R)$旋转$\theta F(\mathbf{x}, R)$为点 $\mathbf{x}_r$。

平移——将平移加上以给出 $\mathbf{x}^{\text{def}}$

$$\mathbf{x}^{\text{def}}=\mathbf{x}_r+(W(\mathbf{x}_R)-R(\mathbf{x}_R))F(\mathbf{x}, R)$$

我们注意到在三种情况下，动作都是由密度函数调整的。在旋转和平移中，动作是 W 和 R 之间的一定差异的函数。W 对 R 的运动影响了铰线作用域的上所有顶点的运动。

图 8-7 给出了一个用来控制扬眉动作的铰线的例子。这根铰线是一条单段贝济埃曲线且仅仅由面部网格上的 4 个控制点组成。在实时情况下，我们可以很容易地选择眉毛的运动和表情的控制，这些控制是预设来实现适当的网格变形。或者，我们可以有由一个或多个参数调用的连续的提升控制。我们从这个例子看到，铰线能完美地适用在局部或者细节形变中，虽然如前面提到的，铰线也能够以相同的方式用在 B 样条架构中。

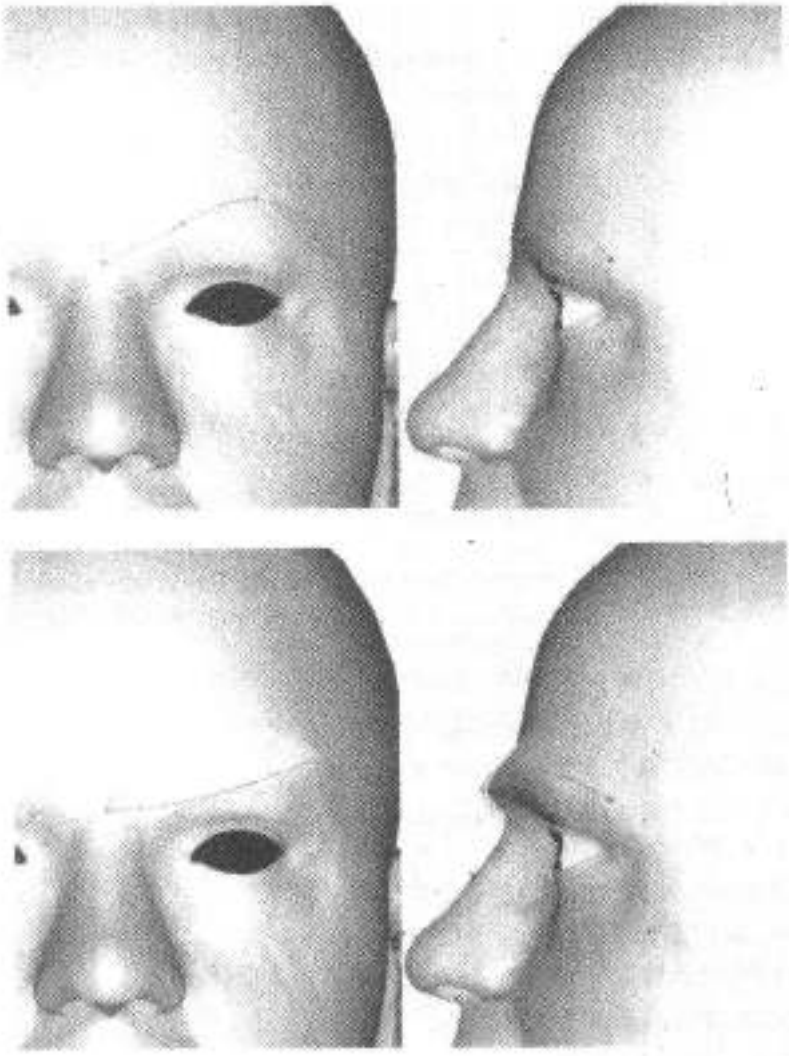


图 8-7 铰线（由一条单段贝塞尔曲线生成）被用于控制面部的扬眉动作

8.6 皮肤控制

8.6.1 面向表面的自由形状变形（SOFFD）

这个技术［SING00］也是在建模和动画系统 Maya 里面实现的，且可作为一个通用的变形技术，虽然作者强调它只在自动操作角色的皮肤上的使用。这个技术和前面提到的变形运动中的铰线方法是有联系的，变形运动受到来自最初绑定在需要变形的对象上的参照（控制）表面和可运动表面的影响。在这个意义上，SOFFD 技术将铰线方法从 3D 曲线到表面进行了扩展。若将它和 FFD 方法进行比较，我们说 SOFFD 技术强调基于表面的变形，而 FFD 基于体的变形。SOFFD 技术基于表面变形的动机在于，通过对一个对象的变形控制来结束一个表面和要变形的下一个对象视觉上相互关联。这和 FFD 中变形实体是控制点点阵是不同

的。该技术解决了变形结构的移动问题，并使在变形好的结构中产生所需要的变化更加清楚。另一个较之 FFD 的优点是其等价控制点可以任意分布，使得操作者很容易在任何需要的地方引入局部控制。

在 SOFFD 中，变形器是一个三元组 $\{D, R, local\}$

- D 是驱动表面
- R 是最初和物体表面绑定且和物体表面一起注册的参考表面
- $local$ 是一个标量

参考表面 R 通常是一个对象网格的低分辨率版本。R 和 D 初始是一致的。然后 D 被移动且 R 和 D 之间的偏离控制了对象的变形。$local$ 参数控制变形的位置。随着 $R-D$ 的背离，这个过程的登记阶段计算基于米为单位起作用的距离的作用权重，这些通过控制元素来控制对象点变形发生。一个 R 上的控制元素是一个三角形表面且应在对象表面上每一点计算下式：

$$f(d, local) = \frac{1}{1 + d^{local}}$$

这里 d 是从对象点到 R 的面最近的距离。对每一点 $\mathbf{x}$ 和每个控制元素 k 权重：

$$w_k^{\mathbf{x}} = f(d_k^{\mathbf{x}}, local)$$

是确定的，因此每一个 $\mathbf{x}$ 有一个权重向量与之相关。实际上与每个对象点相关的控制元素的数目是很小的。

R 是参考表面——控制单元对每一个面也定义一个从两条边导出的面坐标系统，为了计算它们的叉积和作用权重，注册阶段计算未变形对象点 $\mathbf{x}$ 在局部坐标系统的坐标。

变形阶段处理 D 中的面，改变形状、位置、方位并将每个对象点 $\mathbf{x}$ 映射为 $\mathbf{x}^{def}$。这些点移动，而在 D 的面坐标空间中的局部位置仍然与 D 移动一致。通过以下方法可以实现。通过面元素 k 变形的 $\mathbf{x}^{def}$的世界空间位置是：

$$\mathbf{x}_k^{def} = \mathbf{x}_k^R \mathbf{M}_k^D$$

这里：

$\mathbf{x}_k^R$ 是 $\mathbf{x}$ 在 R 的局部坐标

M_k^D 是 $\mathbf{x}$ 在驱动表面上面 k 的转换矩阵

$\mathbf{x}^{def}$，考虑和 n 个面相关的 $\mathbf{x}$ 变形后的位置：

$$\mathbf{x}^{def} = \sum_{k=1}^{n} \mathrm{w}_k^x \mathbf{x}_k^{def}$$

一个有关球体的简单例子如图 8-8 所示。

在蒙皮算法中这个方法的使用很简单。驱动表面 D 是绑定在骨架上的，而不是绑定在对象的皮肤上。类似骨架的动画，D 控制了皮肤的动画。在通常情况下，D 是一个低分辨率的皮肤动画，而艺术家面对处理 D 的任务，并不能完全用骨架来代替皮肤，就要用心处理好一个非常类似骨架但比较低分辨率的皮肤表面。

8.6.2　骨架皮肤精致化

现在我们回到第 7 章的骨架皮肤的主题，同时讨论在那一章中描述的基本模型在这里不能适用的原因以及该问题的解决。(本章描述的大多材料基于在［WEBE00］中的处理。)

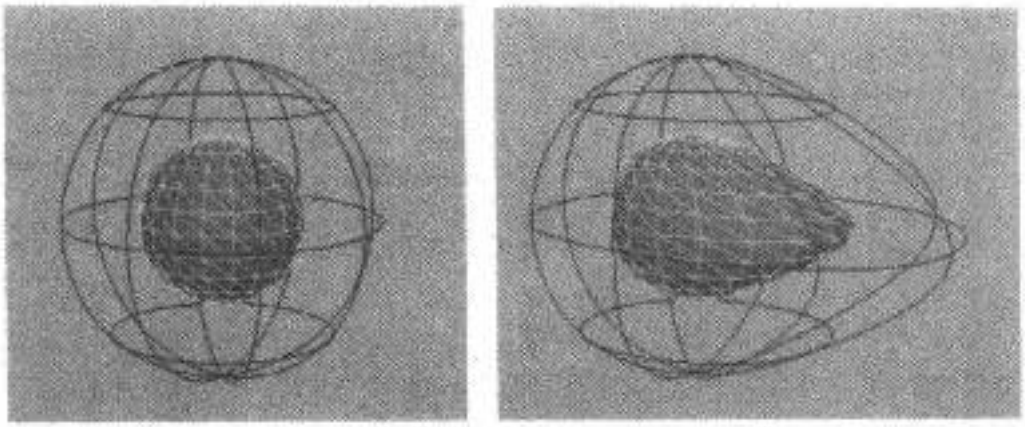


图 8-8　使用两个初始同心球的 SOFTD 的简单例子

我们从基本模型开始，为了便于理解，在此复述一下这个基本模型。为了计算一个顶点相对于其关联骨骼的位置，我们使用下面这个积：

$$\mathbf{x}_i' = \sum_{j=1}^{n} \mathbf{M}_j \mathbf{d}_i w_j \tag{8-2}$$

这里：

$\mathbf{M}_j$ 是骨骼 j 的（全局）矩阵

$\mathbf{d}_i$ 是顶点和第 i 个顶点之间的位移向量

w_j 是关联权重

n 是关联数目

我们现在看看如何一般化这个模型且修复它产生的图像缺陷。我们还将把该模型作为一种概念性测试标准，以便在此基础上考察更精细的模型。

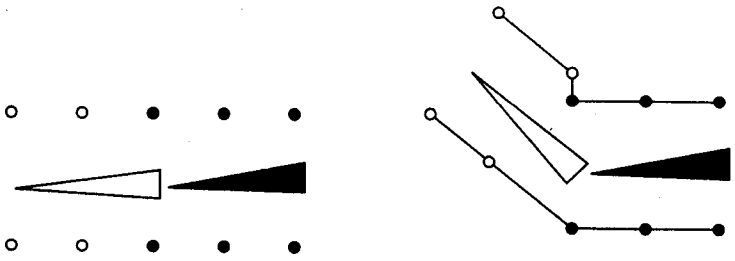
首先考虑如何用这个模型来实现围绕着一个旋转的关节的皮肤变形。例如，我们可以考虑把肘作为一个铰链关节的近似（1DOF）。当一个肘弯曲时，皮肤在一面折叠而在另一面舒展。如果皮肤在关节区域的顶点仅与一根骨骼相关，那么很明显一个不满足要求的变形发生了（见图 8-9a）。在图 8-9b 中，中心顶点与每根骨骼相连。求和时每个顶点都被一分为二，分别用于每根骨骼的变换、求和然后求平均。从图中可见中心顶点的结果位置位于连接各个分割开的组件的一条直线上。通常，皮肤在这个关节的钝角面舒展而在锐角面收缩（见图 8-9c）。

围绕连接两个延长骨骼的共同轴旋转的骨骼对其他序列产生重要影响。例如，旋转前臂不影响上臂，如图 8-10 所示，旋转 180°会导致折叠。这个明显的不一致性是一个蒙皮方案的特征，且是一个重要观察结果，该问题涉及到关节角度函数。人们不希望看到动画者不能直接操作变形，但目前的确不能确定继续操作权重 w_j 是否会导致任何的改进。

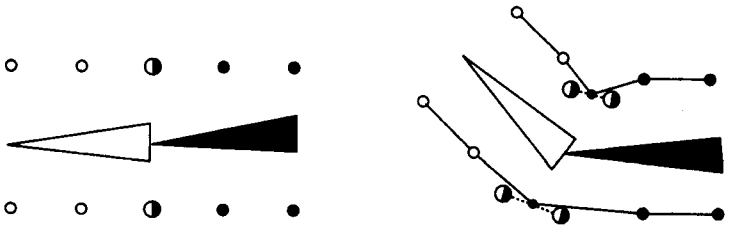
这个方案的优点在于骨骼可以被抽象，即皮肤不必和身体中的骨骼相关。肘的弯曲使得二头肌鼓起来，并从最重要骨骼的动画中获得次要骨骼的动画成形。这个想法也被用于细小的关节运动，如图 8-11 所示。这里补充链接结构控制在关节附近的节点，并且成为主要关节动画的结果动画。抽象骨骼还可以备用于面部动画。在真实情况下，面部运动不是由骨骼

来推动的（除了下巴），而是由肌肉推动。为了画出面部，我们将使骨骼围绕一个合适的轴转动来实现预期控制。骨骼的外部末梢必须移动，由此也可以避免它们凸出到面部的皮肤层（如图 8-12 所示）。

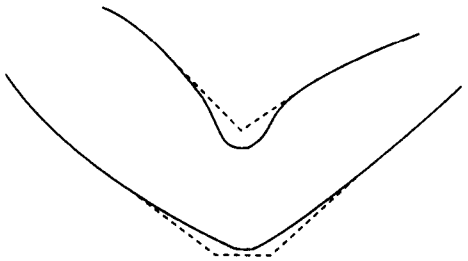
最后，提出这个方案的问题，所有的问题来源于这个事实：我们试图通过单独移动顶点精心设计一个多边形网格模型的变形，并且不考虑它们嵌入的环境。我们试图将顶点固定和



a）每个顶点附在一块骨骼上



b）两个中心顶点都附在两块骨骼上



c）在关节形成钝角的一面，皮肤行为满足性，收缩在形成锐角的一面

图 8-9　肘关节的皮肤行为

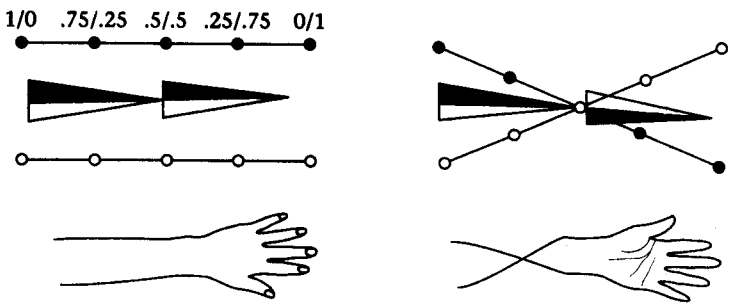


图 8-10　大关节角度问题

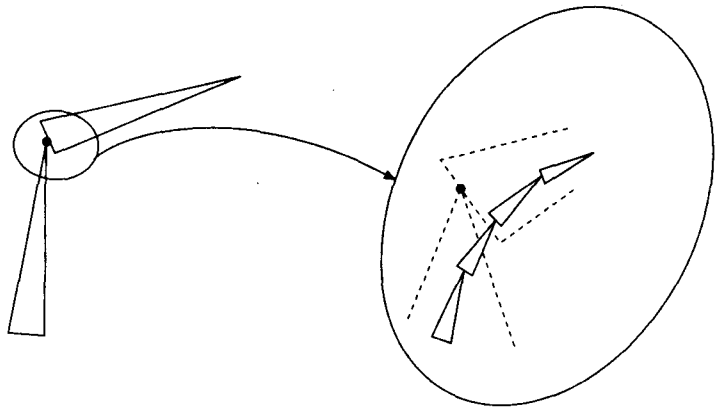
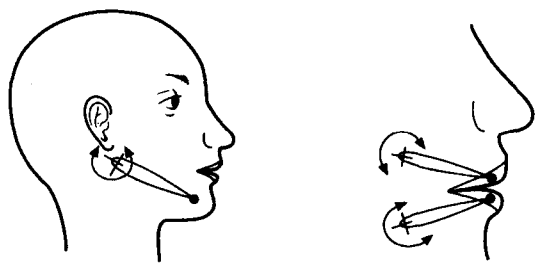


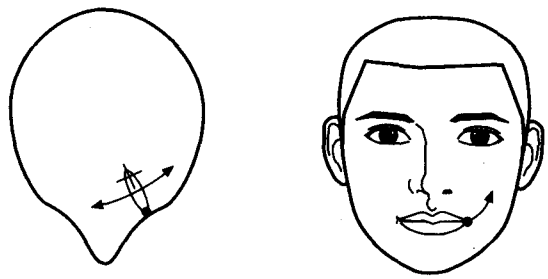
图 8-11　在关节区域使用补充骨骼来控制皮肤变形

关联到不只一块骨骼上。这虽然有用，但是所有的固定都是与当时环境相关的且多少有点不充分。我们需要拖动顶点的更好的方式。

更一般的方法是用形状混合或插值来组合皮肤。这是下一节的主题。这与骨架动画的动机（避免形状插值的底层方法）刚好是矛盾的。然而，若在两者之间达到一个平衡可能是很好的策略。蒙皮算法中的主要问题之一是在骨架动画的过程中接口之间形成的间隙，只能由艺术家来调整某些皮肤权重。



a) 面部动画的骨骼是抽象的（即没有解剖学基础的）



b) 面部动画的骨骼应当沿着头骨的曲率

图 8-12　用在面部动画中的骨骼是纯抽象的

8.6.3　组合皮肤和形状混合

正如我们看到的，前面技术的一个重要缺点就是会产生一些不理想的变形，而这必须通过复杂的或者间接的方法，即调整顶点-骨骼附着权重来修正。Sloan 等人［SLOA00］把这种方法定义为转换混合，表示为式 8-2，对于一个二根骨骼附着，可以表示为：

$$\mathbf{x} = \alpha \mathbf{M}_1^{\theta} \mathbf{x}_0 + (1 - \alpha)\mathbf{M}_0 \mathbf{x}_0$$

这里我们考虑带着一个自由度（肘关节）的两根骨骼且：

- $\alpha \in [0, 1]$ 是骨骼 1 的权重而（$1 - \alpha$）是骨骼 0 的权重（在式 8-2 中 $\alpha = w$ ）
- $\mathbf{M}_0$ 是骨骼 0 的全局矩阵
- $\theta \in [0, 1]$ 是关节角度，相对于骨骼 0，映射到范围 $[0, 1]$
- $\mathbf{M}_1^{\theta}$ 是骨骼 1 在角度 θ 的全局转换

他们建议将这个过程进行一般化，这样它就成为了一个转换和形状混合的组合。为了做

到这点他们处理如下。两个姿态（弯曲和伸直）被刻画为弯曲手臂上的一个顶点 $\mathbf{x}^1$ 和伸直手臂（伸展开的）上的顶点 $\mathbf{x}_0$ 的关键点集合。这些关键点也包括其他预期的影响，例如肌肉凸出，比如说一个弯曲的带有肌肉凸出的手臂，它也可以被舒展为伸直状。我们可以得出弯曲或者凸出手臂的静止位置：

$$\mathbf{x}_0^1 = (\alpha \mathbf{M}_1^{\theta=1}\mathbf{x}_0^1 + (1-\alpha)\mathbf{M}_0)^{-1}\mathbf{x}^1$$

且满足

$$\mathbf{x}^1 = \alpha \mathbf{M}_1^{\theta=1}\mathbf{x}_0^1 + (1-\alpha)\mathbf{M}_0\mathbf{x}_0^1$$

这是弯曲的手臂（见图 8-13）的静止或者伸直的位置，这样如果它被用于弯曲变换，它将产生必须的（想要或者不想要的）在弯曲部位的变形。现在我们要做的事情就是完成新的（弯曲的）手臂其他位置上的几何弯曲，对最初的手臂其他位置（形状弯曲）给定：

$$\mathbf{x'}_\theta = \theta \mathbf{x}_0^1 + (1-\theta)\mathbf{x}_0 \tag{8-3}$$

紧接着是一个给定最终结果的变换弯曲：

$$\mathbf{x}_\theta = (\alpha \mathbf{M}_1^\theta + (1-\alpha)\mathbf{M}_0)(\theta \mathbf{x}_0^1 + (1-\theta)\mathbf{x}_0), \alpha, \theta \in [0,1]$$

实际上，大量的形状或者关键点必须根据环境刻画。在这种情况下，所有的形状未被转换到静止位置且式 8-3 被一个多路形状弯曲替代。

Lewis 等人［LEWI00］把在 8.6.1 节描述的方法称为骨架子空间变形，即 SSD，因为需要的皮肤变形建立在骨架子空间上。非常好的例子是图 8-10 所示的交错影响。图中皮肤由于前臂的扭曲的影响在肘关节处塌陷。他们建议使用名为姿态空间变形（Pose Space Deformation）或者 PSD 的新方法，这个方法有助于动画制作者直接在骨架驱动变形之间进行关键帧插值。在 SSD 中，动画制作者不得不循规蹈矩地通过试探调整权重以达到预期的变形。姿态空间或者是骨架的关节状态的空间，或者是更为抽象的空间。例如 UI 的状态（面部动画中的微笑、扬眉毛等）产生骨架或者骨骼的姿态。姿态空间也可作为这些控制的组合。姿态空间作为一个插值域，且离散数据插值模式（见附录 8.1）用在中间插值。离散数据插值方法承认变形可以在变形空间的非均衡区间被刻画。从姿态空间到网格或者对象本地坐标帧的映射产生所需的（插值的）皮肤变形。Lewis 等人认为这种方法对实时合成是足够有效的，只是比形状或顶点插值开销大。

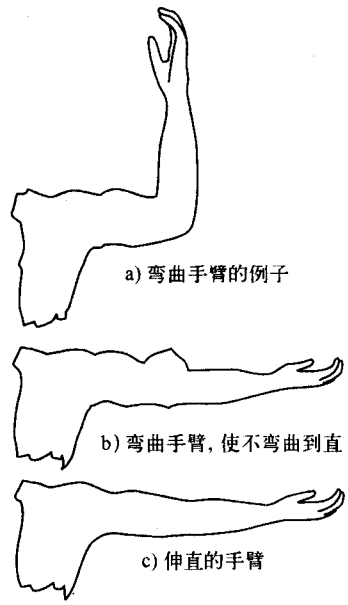


图 8-13　用在形状和转换混合中的手臂

使用 PSD 的过程分为如下几个步骤。首先，艺术家定义好一组姿态且刻画好或者给每一姿态的皮肤建好模型。姿态被定义为和其他姿态相关的姿态控制——关节角度或者 UI 操作——的配置或者状态。总的来说，姿态空间就是一个多维空间且每一维都是关节数目的函数，且它们的角度是自由的。在 n 维空间里面，姿态是一个点。姿态空间中的每一个点和一组变形顶点相关联，且由和静止部位配置相关的 δ 值决定和表达。因此点 $\mathbf{p}$ 在骨架关键帧下移动为：

$$\mathbf{p}' = \mathbf{p} + \boldsymbol{\delta}$$

离散数据插值是两阶段过程中必须的——刻画的关键帧决定权重组 w_i，我们将在附录8.1中详细地介绍。为了合成任意姿态的皮肤变形，所需的姿态放在姿态空间里面，式8-4用于查找新的顶点值。单独的插值在每一个顶点进行。

附录8.1　使用径向基函数进行离散数据插值

我们希望用$s(\mathbf{x})$逼近函数$f(\mathbf{x})$，其中在不同的点$\{\mathbf{x}_1, \ldots, \mathbf{x}_N\} \in \Re^d$给定如下关键值集合：

$$(f_1, \ldots, f_N)$$

这里 d 是姿态空间的维数。

我们选择的$s(\mathbf{x})$形式如下：

$$s(\mathbf{x}) = p(\mathbf{x}) + \sum_{i=1}^{N} w_i \phi(|\mathbf{x} - \mathbf{x}_i|),\ \mathbf{x} \in \Re^d$$

这里：

p 是一个低度多项式（或者是不存在——如同在我们的应用中）

$|\mathbf{x} - \mathbf{x}_i|$是 $\mathbf{x}$ 到 $\mathbf{x}_i$ 的欧几里德距离

w_i 是一个实数值权重

ϕ 是一个径向对称基函数

这个等式是到关键姿态的距离的非线性函数的一个线性组合。它给出了我们找到任意姿态$s(\mathbf{x})$用来充当关键姿态权线性组合的权重 w_i 的集合。

Φ 的一般选择包括：

- 薄板样条$\phi(r) = r^2 \log(r)$
- 高斯函数$\phi(r) = \exp(-cr^2)$

不言而喻，径向基函数对插值离散数据很有效，特别是它对给定的点很少有甚至没有限制（取决于 ϕ 的选择）。

引入如下的向量和矩阵：

$$\mathbf{w} = (w_1, \ldots, w_i)$$
$$\mathbf{f} = (f_1, \ldots, f_n)^T$$
$$\mathbf{G} = \mathbf{G}_{ij} = \phi(|\mathbf{x}_i - \mathbf{x}_j|), i, j = 1, \ldots \mathrm{N}$$

则未知权重参数集由下式给出：

$$\mathbf{w} = (\mathbf{G}^{\mathrm{T}}\mathbf{G})^{-1}\mathbf{G}^{\mathrm{T}}\mathbf{f}$$

第9章　高级角色动画之要素

虚拟人（virtual humans）将在5年之内拥有个性、情感和交谈的能力。他们将会有个人角色、性别、文化，并能了解环境。他们将会对行为作出反应并拥有预见和判断的能力。但是他们必须对情况有独立的感知才能做这些事情。他们得懂得语言，这样人类才能像面对真人一样和他们交流……

Norman Badler 1999 [BADL99]

9.1　引言——一种拟人的游戏界面

在这一章里我们考察现今的技术，并展望未来的发展。很显然，角色动画将不断发展，模拟人也将变得越来越真实。这种发展会包括许多技术，比如动画形体、视觉语音、计算机视觉、自然语言处理、人工智能等。我们将以这些发展中最关键的面部动画和跟踪为重点来学习目前关于它的技术和潜能。首先，我们来看一下它的基本概况。

在特殊的视觉语音中，面部动画是HCI中期待已久的里程碑，许多应用软件将使用它。在电脑游戏中它将作为一种新的交互形式的输出部分，使玩家可以通过使用高级语音命令与游戏进行交流。显然，今后的发展趋势是用语音而不是通过键盘与电脑进行交互。语音输入可以加强玩家和游戏人物之间的联系，从而使模拟角色人物更加逼真。语音输入方式还能传达抽象概念。读者可以考虑一下以下两句话的区别："向左边的门跑去"和"试着到门那里去并保存弹药"。

在游戏中使用这种系统的一个很明显的前提条件是，必须在人工智能功能性和可玩性之间作出折衷。直接控制人物去捡起武器打死怪物比发出'捡起武器然后打怪物'的命令更令人兴奋。在后面一种情况下，玩家变成了一个被动的观看者。

了解拥有这种功能的游戏的总体需求是很有用的。这样我们能够知道面部动画在总体环境中的作用，了解目前技术的发展水平。语音识别、自然语言的处理和面部动画（包括视觉语音和表情控制）构成了游戏和玩家之间的输入/输出层。这一层的功能需求是很明显的（见图9-1）。但这一层的体系结构的组织和功能性目前还不是很清楚，我们先宽泛地定义它为游戏的人工智能。要想将这一拟人的界面形态彻底地弄清楚，必须发展例如自然语言处理（NLP）等技术。可以预言，今后键盘和鼠标的交互方式将替换为语音命令形式以及真人和模拟人物的完全交互方式。

利用这种界面的一部分的技术已经得到了一些应用。例如，MoCap软件能够在数据收集的过程中实时地让一个模拟的人物动起来。这时候MoCap数据收集具有计算机视觉设备的作用，能够产生图像输出。人们普遍认为这是让行动者实时控制模拟人物的动画的另一个版本。软件中还有一个能够运行语音识别和输出到人造的可以说话的大脑的简单数据库查询系统。

界面层的一般任务是将NLP处理器输出的语义表达转化为图形动画。不用说，这是很难的——即使我们已经解决了NLP问题。例如"向右边的门跑去"等语句与环境交互的动作需要一条设计好的路径以避免障碍物。门可能有个把手，你得抓住并转动它。这很难用语

言表达清楚。另外，模拟人可能拥有不同的个性，这会决定它是怎么跑的。很快我们发现将路径规划概括为一个相关的动作设计，并通过对用户命令的语义解释来控制游戏可能和自然语言处理（NLP）一样难。

虽然目前有好的（昂贵的）语音识别技术，但是 NLP 仍然是一个普遍的研究问题。简而言之，NLP 问题就是从被识别的句子中提取其中的语义。这个问题的难点在于各种命令的功能。直接的祈使命令“向门跑去……”比“试着跑到门那里去，但要保存弹药”更容易解释。后面一种类型表达了一般的规则，这种规则可能不和当前情况直接相关。取而代之地，它们表达了一种抽象的知识。语义解释应该取决于游戏环境。而从直接的动作中提取目的意图比根据游戏环境进行抽象解释要简单一些。目前很多游戏可以用相当简单的 NLP 技术来控制。例如，通过多关键字的测定点位，基于模板的有限状态系统可以将输入映射到游戏引擎控制中。但是，随着类型的发展，越来越多的命令将被放置于 NLP 中。

我们能够这样断言：在很多游戏中命令将被限制于游戏环境中，大部分情况下，此模型会将自然语言的句法和语义上的模糊性限制到足以可靠地消除二义性的程度。[⊖] 目前关于游戏的 NLP 方面的工作，使用 DOOM 作为一个可操作的实体。

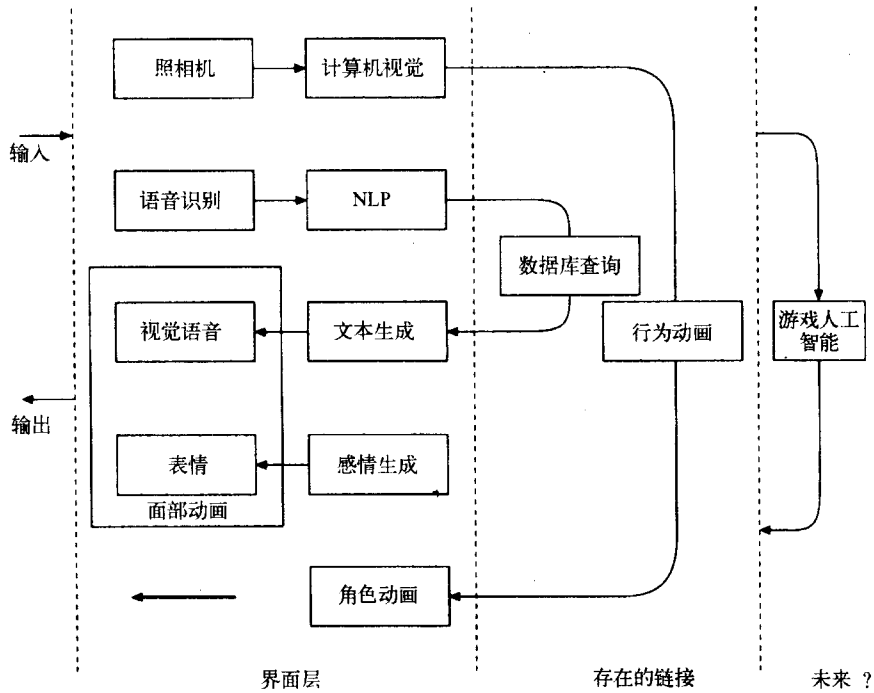


图 9-1　拟人界面层

9.2　将语言表述转变为动画——示例

在图 9-1 中，我们将处于拟人界面层之下的模块宽泛地定义为 AI，因为要想使用这种系

⊖ 语境/语义二义性是设计者很容易犯的错误，应提高警惕。一个经典的例子如下：协和式飞机速度之快有如脱弓之箭；果蝇喜欢吃香蕉。

统，应用软件必须拥有一种 AI 设备。不考虑应用程序的话，所有这些模块的一般性任务是将语言表达转变为非线性的动画序列。

解决这个问题通常是采用一种多级体系结构。此体系结构是由 Brooks 于 1989 年［BROO89］为了控制机器人最先提出的。最近用于动画的例子是 IMPROV［PERL96］和 PAR 体系结构［BIND00］，后者是由宾夕法尼亚大学为虚拟人 Jack 研制的。这方面的其他工作还有麻省理工大学对交谈的研究。实际上，人与人之间的交谈包括语音和面部以及其他一些姿态。

所有这些系统都是分层多级的组织结构，它们或多或少地展示了以下的功能：

- 低级控制。在最低的一级上虚拟人的行为被分解成为一个个小的动作程序（头的转动、右臂的挥舞等）。它们直接用来对骨骼模型进行控制。这些直接作用于特定的自由度上，即传统的动画控制。
- 动作合成（行为者内或行为者之间）。我们知道，人体的动作可以看作不同的低级动作的按序或者同时的执行，所以要建造一个结构来控制它们。我们需要：
  - 动作的安排；
  - 解决具有相同自由度的动作之间的冲突来促进动作的连贯性；
  - 混合动作，避免自身冲突；
  - 避免同等行为者之间的冲突。
- 高级行为控制。此层次结构的最顶端是一种从 NLP 模块接受“理解”高级指令的行为控制器。

最后的结果常被称为非线性动画，该动画是结合①、②、③的函数（①预先定制的规则（模拟人的个性），②玩家的命令，③与其他模拟人以及环境的交互）。

9.2.1 IMPROV（纽约大学媒体研究实验室）

这个系统适用于一些与早期高级动画脚本语言相似的语言。（这些语言从未获得大范围的采用，因为很明显，离线（off line）动画产品更需要艺术家直接进行低级控制。）它的动机是让低级动作控制与行为相结合，并让设计者们将一个故事经过直接翻译成行为脚本。在这篇关于拟人界面层的文章中有趣的一点是，尽管最初的重点是对包含多人物调和以及用户直接控制的游戏的设计，但是它使用了一种像英语的脚本语言。

在 IMPROV 中 Perlin 引入了“行为引擎”。他将其比作“大脑”并从中输出信息到动画引擎——“身体”中。动画引擎接受并执行这些对动作的描述。

低级动画通常通过对自由度的聚合控制应用于动作模块中。例如：

```
define ACTION "talk gesture"
{
R_UP_ARM   25:55    0          -35:65      {N0 0 N0}
R_LO_ARM   55:95    0          0           {N1 0 0}
R_HAND     -40:25   75:-25      120         {N1 N2 0}
}
```

每一行涉及一个单一的自由度 < 倾斜　滚动　偏离 > < 关于频率 f、2f、4f 的噪音函数 > 。在每一个时间拍这个噪音函数控制关节的运动。

设计者明确指出了高级行为的构成和并发活动的冲突解决。他们将动作分为具有各个优

先级的类，在每一个类中各动作又具有一定的优先级。例如：

global/persistent-higher priority

```
GROUP       stances
    ACTION      stand
    ACTION      walk
GROUP       gestures
    ACTION      no_waving
    ACTION      wave_left
    ACTION      wave_right
GROUP       momentary
    ACTION      no_scratching
    ACTION      scratch_head_left
```

local/transitory-lower priority

具有相同自由度的动作被归为一类。一个类中的某一个动作被选中将使这个动作的权从0变成1，同时此类中其他动作的权将渐渐变成0。在每一个时间拍权的和是由对每一个自由度的每一个动作的权相加而得的。

动作不断地控制代理人身体的活动。这些都源自高级脚本，例如：

```
define SCRIPT    "greeting"
{
    {"enter"}
    {wait 4 seconds}
    {"turn to camera"}
    {wait 1 second}
    {"wave" for 2 seconds
    "talk" for 6 seconds}
    {wait 3 seconds}
    {"sit"}
    {wait 5 seconds}
    {"bow" towards "camera"}
    {wait 2 seconds}
    {leave}
    }
```

因此脚本是用来触发动作以控制动画的。它们也能用来改变代理人的特性从而改变其行为：

```
define SCRIPT "eat dinner"
    {"eat"}
    {set my "appetite" to 0}
```

判定规则可以被定义并在脚本中调用：

{choose from { "tom" "dick" "harry"} based on "who is interesting"}

define DECISION-RULE "who is interesting"

factor {his/her "charisma"}　　influence 0.8

factor {his/her "intelligence"}　　influence 0.2

最后，实时的用户交互可以通过给界面层创建脚本元素来实现。

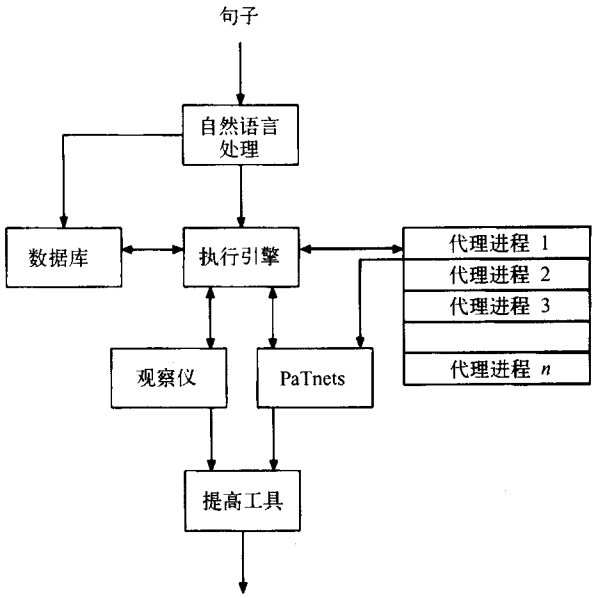


图 9-2　从最高层次的语言处理器开始绘制语言指令转变为 PAR 参数或者变量的各个过程

9.2.2　PAR 体系结构（宾夕法尼亚大学人体建模和仿真中心）

我们所考虑的第二个例子更接近于游戏模型，它是宾夕法尼亚大学目前正在研究的一项工作［BIND00］。它与 IMPROV 相似，也是三个层次的摘要。动作的发生器驱使图形模型，并行传输网络组织行为合成。高级表现是通过参数化动作表示法（PAR）实现的。

我们从最高层次开始讲起，语言处理器将一条语音命令转变成 PAR 形式的参数或者变量（见图 9-2）。这种处理器的一个主要功能是提供了在英语规范中遗漏的细节。例如，像"走到门口然后慢慢地转动把手"这样一条命令并没有指出该怎样抓住把手和朝哪个方向转动这些信息。这种 PAR 表示法也包含了适用性和开始以及终止条件。在上面这个例子中我们还需要知道代理人必须走几步才能到达门口，这些取决于他的高矮和开始的位置等。这种 PAR 表示法直接与并行传输网络（PaTnets）相连接。

有一个被报道的研究场景是模拟军事训练的。玩家/训练者（通过指挥或者标准的命令）控制一队士兵保卫一个入口。当这些士兵检查进入这个入口的车辆驾驶员的身份时，这些指令用来改变他们的行为。如果驾驶员的身份是间谍列表中的某一个，士兵们将用武力监禁

他。训练者不断细化士兵的行为从而使被发现的错误不再重演。例如，当士兵用枪威胁入侵者时应该找掩护，否则可能被枪击。训练者发出一条这样的指令："当你用武器对准驾驶员时必须找掩护……"

9.2.3 具体化的对话界面代理（MIT 媒体实验室）

前面的例子都是先创建一个条件，在这种条件下代理人改变自治的程度并对语音命令作出不同的反应。这一节所讲述的由 MIT 媒体实验室所做的这项工作，不再只注意一个设定的场景中代理人如何对玩家的指令作出反应，而是关注如何能够构造出以人类的方式交谈的代理人。换句话说，代理人能够融入场景中——至少是在谈话交互所涉及的方面。这意味着需要形成一种包括非语言交互的对话结构的谈话方式：

Communicative functions	**Communicative behaviour**
Initiation and termination	
React to new person	Short glance at other
Break away from conversation	Glance around
Farewell	Look at other, head nod, wave
Turn-taking	
Give turn	Look, raise eyebrows (followed by silence)
Want turn	Raise hands into gesture space
Take turn	Glance away, start talking
Feedback	
Request feedback	Look at other, raise eyebrows
Give feedback	Look at other, nod head

交互的例子

Rea 是一个具体的对话代理人，她主要的工作是房地产生意。她掌握着波士顿用于出售的楼盘和房屋的数据库信息。她可以拿出这些房产的照片和里面房间的照片并指出和讨论它们的显著特征。以下是一次真实交互的摘录：

Mike 走近投影屏幕。Rea 正转向侧面，空闲地左顾右盼。当 Mike 进入安置在屏幕上的两台照相机的拍摄范围时，Rea 转过来面向他并开始对话：

REA > 你好，我能帮助你嘛？

MIKE > 我想在 MIT 附近买所房子。

Rea 点头，表示她在听着。

REA > 我给你看一套房子。

一张房子的照片出现在屏幕上，Rea 眨眼看了看房子，然后看着 Mike。

REA > 它在萨默维尔。

MIKE > 给我介绍一下它。

Rea 抬起头看旁边准备说话。

REA > 它很大。

Rea 做了个很豪华的手势。

Mike 举起手表示有话要说，Rea 不再继续而是等待他说。

MIKE > 告诉我更多的信息。

REA > 当然。它有一个很好的花园。

Rea 用手势勾画了一个曲线表示花园是沿着房子两边伸展的。

MIKE > 它有多远？

REA > 到广场 T 站 5 分钟的路程。

Rea 通过用手指做了个走路的姿势表示是从 T 站走来 5 分钟。

MIKE > 房子有多大？

REA > 它有四个卧室，三个盥洗室……

Mike 打断了 Rea，Rea 立刻停止了说话。

MIKE > 等等，告诉我哪里是主卧室。

REA > 对不起，我没听清。你问什么？

MIKE > 哪里是主卧室？

REA > 它在楼上。

Rea 向上指了指。

MIKE > 哪里是主盥洗室？

REA > 它在主卧室的旁边。

Rea 把手放在一起表示卧室和盥洗室的连接关系。

关于房子的介绍继续着。

这个系统的体系结构如图 9-3 所示，它与图 9-1 中的一般结构一致。

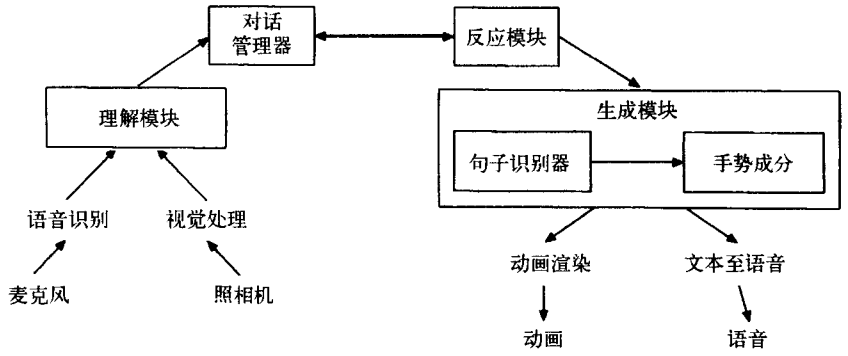


图 9-3　REA 系统体系结构

9.2.4　游戏结论

- 谈到电脑游戏，可以得出如下结论：我们将很快拥有这种使拟人界面和游戏应用相结合的工具。
- 这个例子阐述了这种层次结构在降低复杂度方面的作用，并被公认为是由如下三种形式组成的三层结构解决复杂度的好途径：
  - 高级行为规范；
  - 合成——一个当动作发展时控制代理人内部或者他们之间的程序化动画的结合的

过程；

- 低级动作控制。

此处的重点是输入、NLP 和身体/物体动画控制。而将面部动画工具链接到这个系统的输出部分的工作看起来很少。很明显，这种像人类代理者的行为依赖于他们的身体活动和面部动画/语音的质量。

9.3　面部动画、视觉语音和跟踪

本章的剩余部分将集中讨论一个可能是拟人界面中最重要的要素：与视觉语音和面部跟踪有关的面部动画，用于语音识别和感情识别。

面部动画是计算机图形学中一个尚未解决的问题。理由很明显，我们先从为什么很难得到逼真的面部动画谈起。我们可以将人物面部动画和相关的身体姿势分为 5 类：

1）简单的头部运动；

2）简单的眼睛运动；

3）面部不同部分复杂的变形所形成的表情；

4）讲话——嘴唇、下颚和舌头的复杂变形/运动；

5）在讲话时手臂和手掌的适当的姿势（并不仅限于此）。

从这些分类中我们可以明白其中的困难：

- **综合**　带表情的视觉语音包括所有这些运动。一个逼真的模型必须将它们恰当地结合起来。
- **同一/惟一性**　第二个困难是面的惟一性。理想情况下我们希望得到一个潜在的能够容许不同“面具”的变形模型。目前这种途径最普遍的显示方式是纹理映射到一般的多边形网格。网格的顶点被激活。这种映射的问题不同于人体动画的其他方面（例如身体和结构）。
- **渲染品质**　表情的细节很难通过标准的几何学和明暗模型充分地表现出来。即使是在较好的细节层次上，目前皮肤的纹理也不能达到很高的品质。在缺乏图片真实性品质的情况下，我们也会使用卡通人物的一些特性。

在游戏中，预先由艺术家设计好的帧序列占统治地位。有经验的艺术家作成的序列仍然比参数化/程序化的动画更加逼真。比方说，在人体动画与反向运动学之间作个对比，前者是由艺术家设计的，也可以基于动作捕捉技术而设计，因此前一种方式设计出的动画动作更为自然。

在另一方面，图形学上大部分进步性的工作似乎都发生在 1985 年到 1995 年间。从那时起这一领域趋于平静。这些研究的最终目的是，生成一个能够接受所有一般的身份并综合以上 5 个要素的高级参数化控制系统。换句话说，一个模型能够在一个很高的层次上进行脚本控制——“愤怒地读下面这个句子……”。

面部跟踪是一种在游戏中有重要应用潜力的计算机视觉方法。它与面部动画相对。和驱使面部模型进行 3D 变形比较，我们更希望通过照相机去实时地“读出”或者跟踪真实面部的 3D 动作。这一节很值得我们学习，不仅仅因为它给出了面部的实时电脑视觉显示方法，更因为最受欢迎的方法是通过综合来分析的。总的来讲，3D 模型存在于应用中，并与真实的影像比较。模型的参数形成了一个研究空间，我们试着在其中找到与影像匹配的方法。这

样，面部动画和计算机视觉就在这些应用中联系起来了。

未来的游戏中这些技术将得到应用。例如，游戏 AI 可以读出玩家面部的表情/感情。人类之间不再只通过语言还伴随着表情进行交流。另外一个很重要的应用是它能够协助进行语音识别。

面部跟踪可以用来从一个电脑头部图形中导出动画脚本。这样游戏应用可以在一个多玩家游戏环境中得到高品质的玩家动画。

9.4 用于控制、渲染和跟踪面部网格的模型

我们现在纵览一下研究过的几种主要的控制方法，接着看每一类的例子的描述。根据所用的方法，模型表示可以决定变形控制方法，遵从这些方法我们可以得到很多不同的途径。

从动画制作者的角度来看，主要的问题是什么是控制参数，它们如何与面部模型的动作相关联，如何操作它们以逼真地对网格进行变形以及它们是否足够合适？

关键帧的三维变形

这是标准方法——这个方法可以支持高品质动画，特别是在与基于图像的渲染技术结合使用时更能发挥作用。自动变形暗示着这样一个建模需求：存在一个一般的能被调整以适应特别头部的网格。这个方法的复杂之处在于建模和捕捉阶段。关键点之间的变形是简单的。

然而，欲用这样一个简单的方法生成正确的面部动画是不太可能的。就是说在面部表情变换时通过插值生成的中间帧不能符合现实的情况。如果它是正确的，就意味着面部所有的点能通过相同的动作从一点移到另一点。然而，真正的问题不是在于它是否正确，而是在于它能否生成足够正确的动画而变得逼真。变形方法的另外一个问题是动画是由已完成的帧作为基本成分生成的——这也许不能作为生成新奇动画的强大基础。

基于图像的方法和跟踪

基于图像或者模型的方法能够用于动画和跟踪。最近很多对面部动画的研究都与视频序列基于模型的跟踪有关。这意味着通过使用计算机图形模型的几何学信息和视频序列中的 2D 像素运动，并从视频序列的 2D 像素运动中推断出真实头部的 3D 运动。人们对该方法感兴趣的原因有很多：

1）它的作用相当于传统的动作捕捉，但后者很难应用到细微的面部变形。

2）捕捉面部表情/视觉语音的运动大概比对静态的面部姿态进行采样然后通过插值和混合来产生动画要更正确。

3）它在计算机视觉方面很有应用潜力：例如，感情的识别和最后的设计界面——表演驱动的动画。

4）它能用于在很低带宽的系统中传输动画数据并被 MPEG4 标准所识别。

参数化

这个方法的目的是抽取一个用来描述网格的变形活动特性的小参数集合。说话时嘴唇的动作就适用于该方法。使用较少的参数集合可以简化动画的生成。参数表现方式的产生是源于关键帧方法的限制性。它初次在面部动画中的应用要归功于 Parke［PARK82］。Parke 将面部形状参数和表情参数相结合，它们都直接与网格顶点相关联。表情控制的例子包括：眼皮的张开，眉毛的弯曲；眉毛的分开，下颚的转动，嘴巴的张开，嘴巴的表情，上嘴唇的位置，嘴角位置和眼神的凝视。这项工作的最终目的是产生一个模型，并可以通过设置合适的

参数集合来表现出任何一种特殊的面部和表情。一个关于指定面部表情参数的方案是 FACS（面部动作编码系统［EKMA73］）。它虽然不是为电脑动画而产生，但是已经被用于各项研究中。这个方案是基于动作单元的（例如，上嘴唇抬高器、下压嘴角器等），有 46 个类似单元被定义，它们是基于面部肌肉的动作而产生的。其中的一个集合可以结合起来形成一个面部动作。最近有关参数化的工作是通过分析观察所得的面部动作来找到参数集合的。

伪肌肉（向量肌肉）模型

这是最近几年最流行的模型之一，是由 Waters 于 1987 年引入的［WATE87］。他的想法是对那些有现实基础的多边形网格进行控制——肌肉控制。Waters 使用两类抽象肌肉——用于拉伸的线性肌肉和用于挤压的括约肌。它们只在一点上与网格空间相联系（就是说，它们不必与网格点相连）。它们有方向性指定，所以肌肉的控制独立于详细的面部拓扑。这个模型的控制参数基于 FACS。

面片技术

这是用来使网格模型变形的传统的计算机图形方法。在 FFD 范型中模型被贝济埃超面片包围着，超面片的控制点控制网格的变形。因为需要的变形过程的复杂性，面部的表面由许多贝济埃曲面片构成。使用面片技术时，所有的控制点允许层次控制。当然，我们也可以使用层次 B 样条。

基于身体的模型

为了解决基于向量的肌肉模型的限制，Terzopolous 和 Waters［TERZ93］引入了一种肌肉模型，它的结构具有生物的真实性且基于身体。这是一个复杂的三层合成组织模型，每一层都由一个弹性模型来模拟。这些层代表了皮肤组织和肌肉。每一层的弹力和硬度反映了不同的组织特性。面部拓扑通过构造元素来形成。一项研究表明 960 种元素产生了总共 6500 种弹力。这种复杂且昂贵的模型看起来好像并不比那些较简单的模型能生成更好的图像。

下面我们来进一步讨论这些控制方法，忽略关键帧的三维变形（第 7 章中已详细介绍过）和基于身体的模型（因为不能确定它们是否相关）。

9.4.1　基于图像的建模、渲染和跟踪

接下来两个方法使用基于图像的建模技术来捕捉动态面部表情中的细节。第一个从照片中摘取信息，第二个从视频序列中获得信息。它们将动作捕捉的哲学和基于图像的渲染相结合。它们的缺点是（如同 MoCap）动画必须由预先录制好的图像信息产生。（在视觉语音中使用基于图像的技术绝不是一个好的想法，但是由于大部分工作都是确立在图像空间上的，所以只有综合视觉语音这么一个观点是可能的。大概所有的游戏应用都得需要一个允许任何头部和照相机位置的 3D 模型。）

变形和基于图像的建模：从照片中进行 3D 变形

使用照片进行 3D 变形对于面部动画的一个难题——面部表情的真实性是一个具有吸引力且便宜的方法。感知面部表情是人类的本能之一，所以使用传统的 3D 图像技术会使面部表情逼真化成为一个很困难的工作。一个很明显的解决方法是使用基于图像的工具。它通过人为干涉和限制来绕过大部分的困难。一般的想法是对已有 2D 图像纹理的头部模型进行 3D 变形。

通过后期处理视频或者电影胶片产生的最成功的幻象之一是 2D 变形。一幅图像中将会产生两个目标之间变形造成的夸张扭曲。在计算机图形学中我们可以使用 3D 变形技术，因

为我们已有了 3D 几何概念并运用它们作为 3D 空间的纹理贴图来保留照片图像的高品质。

这个想法是由 Kurihara 等人于 1991 年提出的［KURI91］。一个涉及纹理混合问题的更精细的版本是由 Pighin 提出的［PIGH98］。后者是我们将讲述的。

变形和基于图像的建模：模型拟合

3D 变形技术依赖于一般的多边形网格头部模型，涉及特殊的头部照片和头部的特殊表情。虽然为了适应不同的表情进行了变形，但是因为捕捉的表情被纹理映射到同一类网格上，所以面部表情的变形变简单了，不再需要指定对应目标。这种方法有效地在 3D 空间中使用了 2D 变形技术，2D 平面变换到 3D 网格上。但是要得到好的品质就必须在前期处理中进行人为干涉。

每一个捕捉的表情都有很多图像与之对应（4 或者 5）。我们将选出网格顶点的一个子集使每一个捕捉的图像在某一些点上相关联。对于每一个图像，这将通过获得旋转矩阵和平移向量让照相机进行姿态恢复。这是通过对已初始化的近似照相机位置（前景、边景等）和已知的头部 3D 形状迭代来得到的。这种迭代改进了位置的估算，从而使得观察到的和预测的特征点的位置区别减小到最小。这样就知道真实头部特征点的 3D 位置了，于是网格也会进行变换来适应它。剩余的网格顶点将通过一个叫作离散插值的过程运用径向基函数进行变换来拟合真实面部（见附录 8.1）。通过已经变换的顶点位置和位移来计算剩余顶点的位移。这个过程有效地在一个已知的用于控制剩余顶点运动的位移中插入一个光滑向量值函数。这个过程相当于在已变换的点上进行表面插值。

变形和基于图像的建模：纹理合成

接下来的这个过程涉及从照片中摘取单个纹理贴图。方法是将每一幅照片图像投影到围绕着头部的圆柱体上（见图 9-4，彩页中也有），得到纹理贴图。

因为每个点 $\mathbf{p}$ 都被映射到很多照片上，所有需要将每一个单元图像按照加权函数 w^k 来组合形成纹理图案：

$$T(\mathbf{p}\rightarrow u,v)=\frac{\sum_k w^k(\mathbf{p})I^k(x^k,y^k)}{\sum_k w^k(\mathbf{p})}$$

其中 I^k 是第 k 张照片，(x^k, y^k) 是照片中对应于 $\mathbf{p}$ 的点。

w 用于评估：

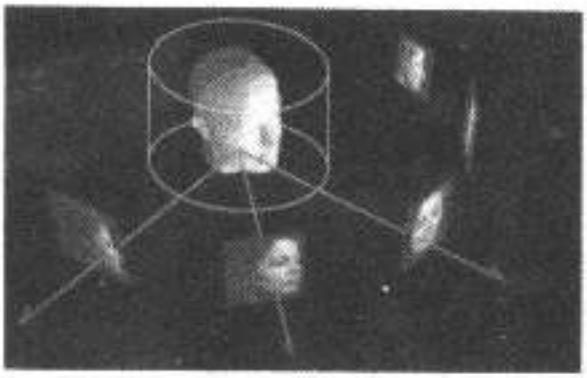
1）自封闭：除非点 $\mathbf{p}$ 在第 k 幅图像的前面并在其中可见，否则 $w^k(\mathbf{p})$ 为 0。对于那些具有凹度的物体这是需要考虑的。

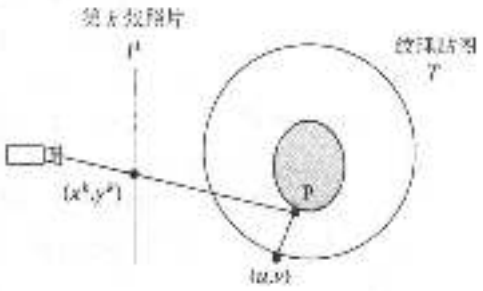
2）平滑性：$w^k(\mathbf{p})$ 必须平滑地变化以确保不在混合时出现裂痕。

3）位置的确定性：它是 $\mathbf{p}$ 点的表面法线和从点 $\mathbf{p}$ 到 (x^k, y^k) 的向量之间的点积。一幅纹理贴图的正确与否由关于这个角的一个函数来确定。

4）视角相似性：在独立视角的合成中，$w^k(\mathbf{p})$ 只依赖于 $\mathbf{p}$ 点在第 k 幅图像上的投影方向和在渲染图像上的投影方向之间的夹角。然而，这个因素不完全令人满意，因为这意味着我们将保留捕捉场景的光照。

纹理合成可以作为预处理（独立视角纹理映射），也可以在独立视角纹理映射的渲染过程中作为多通道来处理。每一个路径根据相同视角的因素来对结果进行记权。独立视角纹理映射可以生成高品质的渲染结果。





点 p 可能出现在不止一幅图上



合成的纹理贴图 T

图 9.4　构建一个“固柱形”的纹理贴图

如前所述，因为同类网格中的顶点具有对应关系，在表情之间通过变形产生动画序列是简单直接的。事实上，控制多边形网格的变形这个有关面部动画的基本问题通过对真实面部的所选点的 3D 位置自动恢复已经解决了，包括表面插值和手动干涉，前者确定剩余点的位移，后者初始化设置照片中某一个点与网格中点的对应关系。这项技术使得照片变得真实，但是它的明显缺点是动画变形基于预先记录好的表情。

可以扩展这个模型来处理视觉语音的问题，其中还需要很多捕捉的表情。

9.4.2　跟踪方法

基于 3D 模型的跟踪

基于模型的跟踪方法通常被称为视频克隆，它通过使用预先定义好的出现在视频序列中的头部 3D 模型来跟踪视频序列。它的目的是从一个 2D 图像序列中抽取网格变形信息，可以被看成是动作捕捉的精化。标准的动作抽捉技术是跟踪孤立点的动作，基于模型的跟踪是要找到网格中所有点的动作。它们的一个显著优点是，如果成功的话，它们可以抽捉到面部表情的动态微妙变化。

运用基于 3D 模型跟踪的一个最常见的方式是通过某种‘综合分析’的方法。尽管这个算法的细节可能很复杂，但这个想法比较简单。我们建立视频序列中每一幅图像对网格顶点

位置的初始化估计，显示出相对的模型并将其与视频图像进行比较。然后寻求使得合成的显示版本与视频图像的区别减小到某个极限的一种解决方法。

Pighin 等人［PIGH99］使用了在以前的方法里用到的基于图像的模型，将其用作综合比较分析中的模型。该方法假设任何视频图像可以通过一个预先记录的表情的线性结合匹配，当前的视频图像可以通过找到预先记录的模型的 3D 变形来有效地匹配。

这是通过给每一个表情模型赋予一个权 W_e 并通过限制 W_e 的值在所有加权表情空间中探求，直到找到一个解决方法。这些比较图像通过多步渲染出来，并通过将各部分结合来建立最后的图像：

$$I_{\text{final}} = \sum_e W_e I_e$$

其中 I_e 是第 e 个表情模型的图像。

权参数 W_e 的搜索空间存在一个限制，就是它们的总和必须是 1。**P** 点的一个完整的参数集合还包括头部的平移和旋转。一个迭代优化过程用下面这个误差函数计算视频画面和渲染的图像之间的差异：

$$E(\mathbf{P}) = \frac{1}{2} ||I_{\text{video}} - I_{\text{final}}(\mathbf{P})||^2 + D(\mathbf{P})$$

其中 $D(\mathbf{P})$ 是一个处罚函数。

为了减少由于使用照相机、摄影机和渲染图像产生的颜色区别而造成的匹配问题，渲染图像和摄影图像是已被带通滤波的。假设可以通过保留纹理贴图的初始光照得到的渲染图像或渲染时的光照来模拟摄影图像，这项技术的直接应用是产生高品质的行为动画和通过选定的参数使不同的面部模型变成动画。

在这一节中我们已经研究了一种依赖被跟踪的特殊面部的已知详细知识的跟踪方法。用于匹配的搜索空间由一个预先记录好的被跟踪面部表情样本组成。很明显，跟踪看不到的面部是很难的问题。在游戏应用中，一般情况下单一玩家将只与游戏交互，所以会碰到相同的情况。游戏可能可以容忍玩家的面部通过跟踪程序被‘注册’这样一种状态。但是这样的学习状态必须比现在讨论的情况简单很多。

在下一节中我们将讨论一个简单的跟踪问题，它将允许一个更容易的学习状态——嘴唇跟踪。

嘴唇跟踪

比上一节的方法简单的一种途径是仅仅考虑嘴唇跟踪。嘴唇跟踪既是语音识别的一个重要提示，又是视觉语音的数据收集方法。嘴唇跟踪算法分为两类——二维和三维。二维的算法依赖少量或者不依赖照相机和头部之间的相关动作。如果发生了头部和照相机相关动作，通过这种二维算法，嘴唇将会变形，于是这个算法将会在分辨由于头部运动造成的嘴唇变形和嘴唇的实际变形上花大量的工作。如果零相关动作约束是不现实的，那么需要使用三维算法。像上一节所说的，当三维的姿态成为算法的一部分时，由于姿态变化造成的嘴唇变形可以被消去。

二维嘴唇跟踪

二维嘴唇跟踪通常是一个两阶段过程。第一个阶段是传统的图像处理，它将从图像中分割出嘴唇信息；接下来是一个跟踪嘴唇动作的算法。

在合适的颜色空间里从一幅彩色图像中分割出肤色信息相对比较容易。我们假设可以用嘴唇的红色来识别在输入的图像中嘴唇的像素。先将输入图像中的 RGB 像素转化为一个分类的颜色空间。RGB 颜色空间形成了一个立方体，一个基于色调的方便的空间是 HSV 空间——一个六角圆锥体。在 HSV 模型中变化 H 对应于颜色的选择。降低 S（减少颜色的饱和度）对应于白色的增加。降低 V（使颜色下降）对应于黑色的增加。通过考虑这个六角圆锥体的几何值值，我们可以很容易地理解 RGB 和 HSV 之间的变换。如果将这个 RGB 立方体沿垂直于它的主对角线的平面进行投影，将形成一个六角形圆锥。

以下内容是 6 个 RGB 顶点和 HSV 模型中六角圆锥体的 6 个点的对应关系（见图 9-5a 和 b）。

RGB		HSV
(100)	red	(0,1,1)
(110)	yellow	(60,1,1)
(010)	green	(120,1,1)
(011)	cyan	(180,1,1)
(001)	blue	(240,1,1)
(101)	magenta	(300,1,1)

其中 H 以‘度’来计量。接下来的代码将一个 RGB 三元组转变为 HSV 形式。

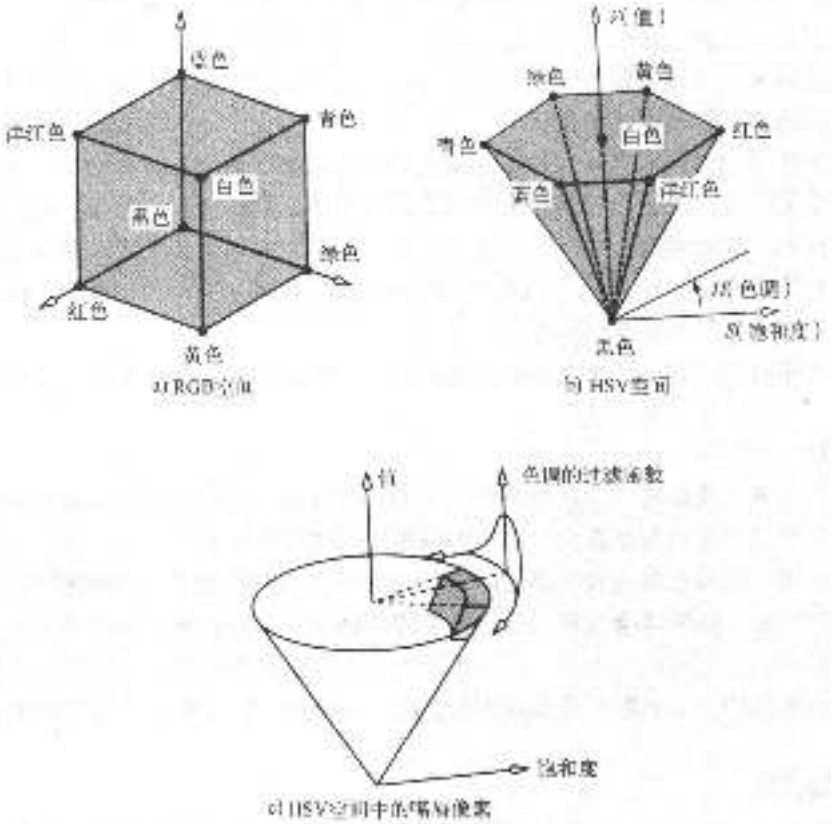


图 9.5　颜色空间和分割

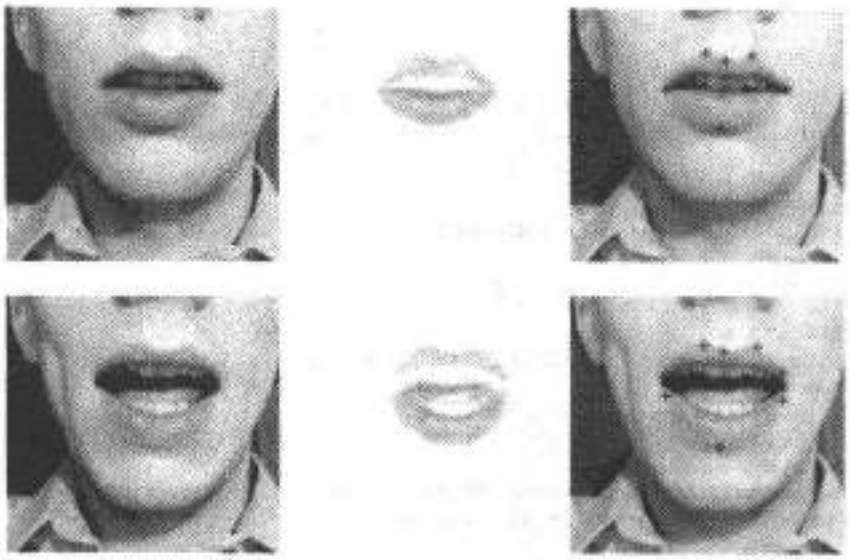
```
struct HSVTRIPLE
{
   FLOAT hsvHue;
   BYTE hsvSaturation;
   BYTE hsvValue;
};
HSVTRIPLE RGBToHSV(RGBTRIPLE rgb)
{
   HSVTRIPLE rtn;

   BYTE max = max(rgb.rgbtRed, max(rgb.rgbtGreen, rgb.rgbtBlue));
   BYTE min = min(rgb.rgbtRed, min(rgb.rgbtGreen, rgb.rgbtBlue));
   BYTE delta = max-min;

   rtn.hsvValue = max;
      //value is the maximum rgb value

   if(rtn.hsvValue)
      //if value is 0 colour is black and saturation
      //is 0 also
   { rtn.hsvSaturation = BYTE((float(delta)/max)*255); }
   else
   { rtn.hsvSaturation = 0; }

   if(rtn.hsvSaturation)
      //if saturation is 0 colour is grayscale
      //(black->white) and hue is undefined
   {
      if(rgb.rgbtRed==max)
      { rtn.hsvHue = float(rgb.rgbtGreen-rgb,rgbtBlue)/delta; }
      else if(rgb.rgbtGreen==max)
      { rtn.hsvHue = 2+float(rgb.rgbtBlue-rgb.rgbtRed)/delta; }
      else if(rgb.rgbtBlue==max)
      { rtn.hsvHue = 4+float(rgb.rgbtRed-rgb.rgbtGreen)/delta; }

      rtn.hsvHue *= 60;
      if(rtn.hsvHue<0)
      { rtn.hsvHue +=360; }
   }
   else
   { rtn.hsvHue = FLT_MAX; }

   return rtn;
}
```

一旦一个像素的颜色值被转换到 HSV 空间后，我们在图像中决定嘴唇的中值颜色然后在空间中建立一个对应于嘴唇像素的区域。换句话说，不在这个区域中的像素的 RGB 值将被设为 0。如图 9-5c 所示，为了清楚起见，这个六角圆锥用圆锥来表示。这个图表也展现了另外一个对色调值进行过滤或者使其急速下降的方案。朝向高饱和度和高值的斜线区分了非嘴唇皮肤和影子。急速下降装置也可以用于这些坐标中，这样一来就进行了一个三维过滤操作。这只是一个有很好工作效果的简单例子——众多的图像处理著作将揭示很多的分割算法。

图 9-6（彩页中也有）中显示了颜色空间的分割操作。如图所示，一旦嘴唇被分割出来，我们可以很容易地找到特征点。这些点将被用于下一个处理过程——嘴唇动作跟踪。

当嘴唇被分割出来以后，最普遍的嘴唇跟踪方法是使用一种动态轮廓模型（snake）。它是由 Kass 于 1987 年提出的［KASS87］。动态轮廓是在两个空间方向上延伸的最低限度能量

曲线。这些曲线本身是 B 样条或者一些其他的基础函数。这些曲线动态地匹配或者跟踪具有预设属性的图像轮廓，并在我们已有一定知识的应用中随着形状不停地变化，与之理想地相匹配。



原始图像　　精确分割　　精度

图 9-6　嘴唇上的图像处理操作

snake 需要用某种方法进行初始化，就是说，需要向算法提供接近所求轮廓的近似初始形状。这将由用户通过一些高级处理或者一个先前的 snake 过程来产生。在这种情况下将使用以前发现的精确特性，如图 9-7 所示（彩页中也有）。

当对一张图像使用动态轮廓模型时，这个需要最小化的能量是产生于 snake 当前变形状态的加权和$\mathbf{Q}(u)$：

$$E_{snake} = \int_0^1 E_{internal}\mathbf{Q}(u)\,du + E_{external}\mathbf{Q}(u)\,du$$

总共的能量值 E_{snake} 既依赖于 snake 的形状又依赖于沿着这条路径的图像函数值。内部的样条能量由弹性和硬度组成：

$$E_{internal} = w_1 \left|\frac{d\mathbf{Q}}{du}\right|^2 + w_2 \left|\frac{d^2\mathbf{Q}}{du}\right|^2$$

其中 w_1 和 w_2 分别是弹性和硬度限制。这些意味着当 snake 与轮廓相匹配时，像定义的一样，这将变成某种一致的边缘结构。一个用于匹配这样的图像结构的 snake 模型将比沿其路径具有较大随机偏移的显示具有较小的内部能量值。

外部项由图像信息和$\mathbf{Q}(u)$的交互得到。例如，如果我们定义：

$$E_{external} = -|\nabla I(\mathbf{x})|$$

那么这条曲线将达到斜度的最大值，就是边缘。注意，这个条件是非负的，这条曲线向高倾斜的方向移动以使两个能量条件的和最小。其他的图像约束也可以应用。例如，线可以被用

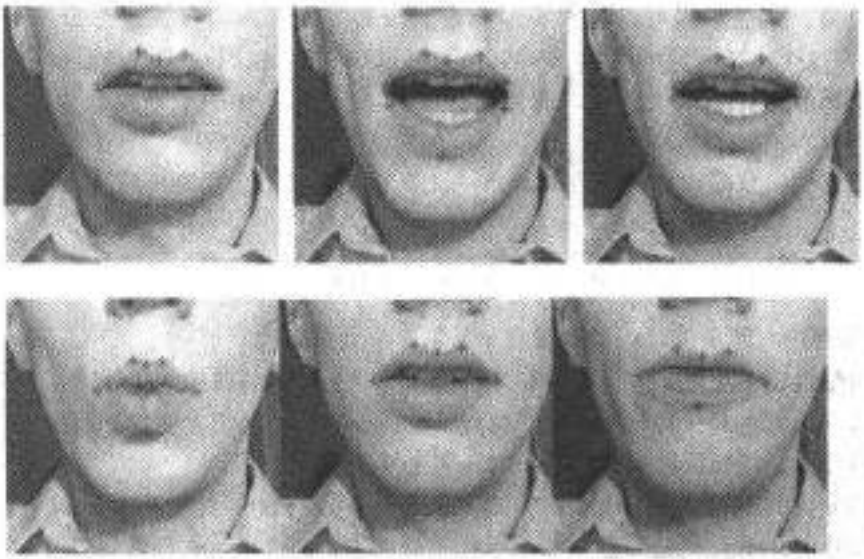


图 9-7 使用 snake 进行嘴唇跟踪。第一行是嘴唇外部轮廓。第二行是嘴唇内部轮廓

作吸引者；可以限制曲线所位于的区域使曲线只围绕同类的区域。限制可以存在于多个 snake 中——Kass 等人描述了一种立体的 snake，可以在立体的图像对中使用一对 snake。

图 9-7 显示了方法的实际应用。在这种情况下嘴唇的内外轮廓都被跟踪了。

三维嘴唇跟踪

二维的嘴唇跟踪可以通过“基于 3D 模型的跟踪”这一小节中描述的合成方法的分析来实现。下一节中所描述的嘴唇模型就运用了这种方法。

9.4.3 参数化

[REVE98] 中给出了面部动画和跟踪中参数化运用的一个好例子。其中的想法是得到一组能最好地实现视觉语音中嘴唇动作表达的高级参数。图 9-8 中显示了用到的 3D 模型，它是一个嵌入了 30 个控制点的表面。这些控制点分为 3 组，每一组 10 个点，它们组成了轮廓曲线。当控制点运动到其他的位置形成新的表面时，就发生了嘴唇运动。

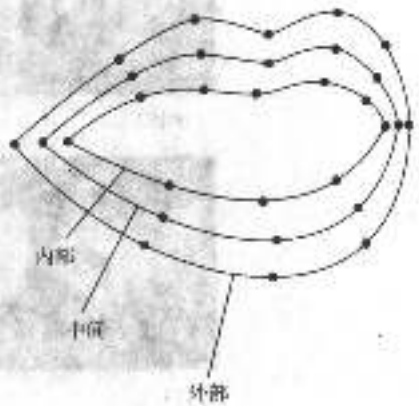


图 9-8 通过 30 个点定义的 3 条轮廓曲线

于是每一个嘴唇形状可以由一个 90 个分量的向量描述，就是说，它是 90 维空间里的一个点。我们将得到一个很大的训练集合，它们根据说话的方式可以分为 10 种（发音时嘴唇呈圆形的元音（rounded vowels）、不呈圆形的元音、摩擦音等）。然后我们运用主要组件分析（PCA）对每个嘴唇形状集合的数据进行分析来决定三个关节参数。

PCA（或 Karhune-Loeve 变换）是一种 N 维空间的线性变换方法，它可以将数据的底层属性沿坐标轴清

楚地显示出来。这些数据可以看作是多元概率分布。如果我们仅仅考虑三维（而不是90维），那么用这个方法可以生成一个椭圆体，它可以最好地包围这些点并返回它的轴。沿着椭圆体的主轴数据变换最大，所以我们使用PCA将互相依赖的坐标变换为独立无关联的坐标。

我们通过找到数据起源（重心）算出一个离散矩阵来找到数据的主轴。这个矩阵的元素是：

$$M_{ij} = \frac{1}{N}\sum_{r=1}^{n}(x_{ri} - x_i)(x_{rj} - x_j)$$

其中N是数据中点的个数，x_{ri}是数据点x的第i个分量。

特征向量给出了主轴，它们的特征值给出了与其分量相关联的变化。第一个分量占所有变化中最大的比例。其后的分量包含了垂直于前面分量的轴的最大变化。要表示每个分量中总变化的百分比，我们可以从它们的和中分离出特殊值。

在他们分析的10个主要的形状中，Revert等人［REVE00］发现PCA分析中前3个分量占了所有变化的94%。这些分量是嘴唇形状中沿着主轴的变化，它们被解释为：

- 第一分量（75%）是突出的圆形姿态。
- 第二分量（12%）是下嘴唇的动作。
- 第三分量（7%）是上嘴唇的动作。

所以PCA分析提出，在用于表示语音的嘴唇运动模型中，任何的嘴唇形状应通过这三个分量作为动画参数进行参数化的工作。这三个参数的任意线性组合将形成所选数据空间中的一个嘴唇形状。

在［REVE00］中，这项工作被扩展到6个参数来包含嘴唇和下颚的运动。这个改进的系统被称为Mother。Revert等人发现分析表明97.7%的变化可以被这6个参数所占据。这些参数被称为关节参数，其中一部分在图9-9中显示出来（彩页中也有）。

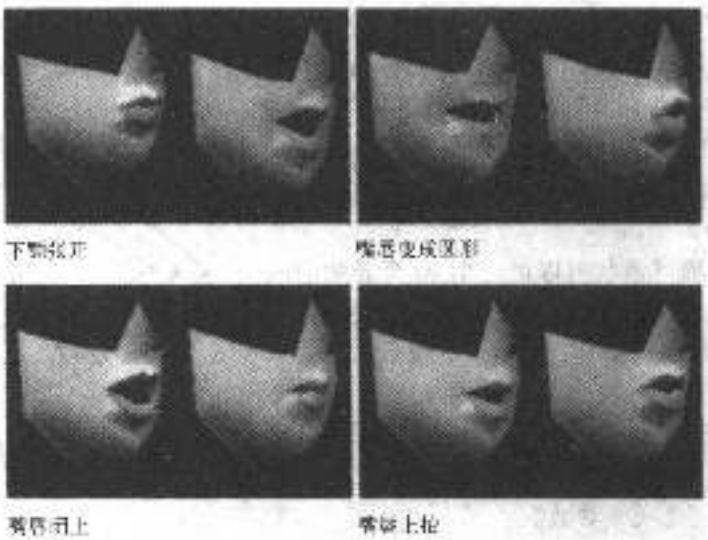


图9-9　MOTHER中关节参数的极端变化

一旦最好的参数被找到，它们将通过视频胶片综合跟踪方法被分析检验（与 Pighin 等人的规则相同）[PIGH99]。这 6 个参数被用作预报器，渲染出多边形网格模型。渲染模型与视频图像之间的差别可以用来完善参数值。渲染结果和视频图像之间的不同功能显示出作为预报器的参数的功效。跟踪错误概率只有 5%。这项工作证明了视觉语音可以用 6 个关节参数和一个低分辨率模型成功地参数化（对嘴唇和下颚）。

9.4.4　伪肌肉模型

如前所述，这是一个于 1987 年首次提出的流行的模型 [WATE87]。这个模型的引人之处在于它是基于对控制变形的面部肌肉的一种近似抽象。这种肌肉模型可以被看成是 FFD 的特殊形式。它们与多边形网格空间相联系（不必与顶点）；在一个典型的应用中 [EDGE01]，网格由 878 个多边形组成，它们的顶点被 25 个肌肉函数和下颚转动所控制。两种伪肌肉被模拟出来——线性肌（24）和括约肌（1）。括约肌是基于一种围绕在嘴唇周围的口轮匝肌（Obicularis Oris）。这些肌肉的收缩对应于嘴唇在说 'book' 或者 'bought' 时的圆形状态。

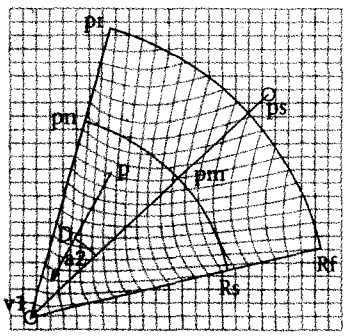


图 9-10　线性肌肉模型中的参数。格子代表收缩

线性肌通过两个点和一个方向与网格相联系。这些点被归类为附属点（v_1 等价于连接在骨头上的真实肌肉）和插入点（p_s 等价于连接在皮肤上的真实肌肉）。收缩表现为皮肤向附属点的运动（见图 9-10）。最大的位移发生在插入点，在附属点没有位移。下式给出在以附属点为中心的三角扇形区域变形（或者说 p 点的位移）：

$$\mathbf{p}' = \mathbf{p} + akr\,\frac{\mathbf{p}-\mathbf{v}_1}{|\mathbf{p}-\mathbf{v}_1|}$$

我们定义 $a = \cos(a2)$，$D = |\mathbf{p}-\mathbf{v}_1|$，则 r 定义为：

$$r = \begin{cases} \cos\left(\dfrac{1-D}{R_s}\right) & \text{当 } \mathbf{p} \text{ 在扇形}(\mathbf{v}_1\mathbf{p}_n\mathbf{p}_m)\text{内部} \\ \cos\left(\dfrac{D-R_s}{R_f-R_s}\right) & \text{当 } \mathbf{p} \text{ 在扇形}(\mathbf{p}_n\mathbf{p}_r\mathbf{p}_s\mathbf{p}_m)\text{内部} \end{cases}$$

其中 k 控制肌肉的力量或者变形运动的范围。其他的变量在图 9-10 中定义。

图 9-11 显示了关于面部网格的 18 个线性肌的分布。通过顶点集合的预处理循环可以标志出受一次特殊肌肉影响的区域中的点。

括约肌可以用参数化的椭圆体来模拟。在这个模型中受影响的顶点被拉向椭圆体的中心。位移的大小是顶点到椭圆中心的距离的函数。但是，我们必须注意使椭圆体的中心不受肌肉收缩影响，以防止顶点堆积在椭圆中心（见图 9-12）。同样，在实际中，这些肌肉向椭圆中心的突出会增加，我们可以用反转位移系数 d 来向前推动网格顶点。

对于括约肌模拟我们有以下式子：

$$\mathbf{p}' = \mathbf{p} + dk\,\frac{\mathbf{p}-\mathbf{c}}{|\mathbf{p}-\mathbf{c}|} \quad d = \begin{cases} fg \text{ for } |\mathbf{p}-\mathbf{c}| > l_y \\ 0 \text{ for } |\mathbf{p}-\mathbf{c}| \leqslant l_y \end{cases}$$

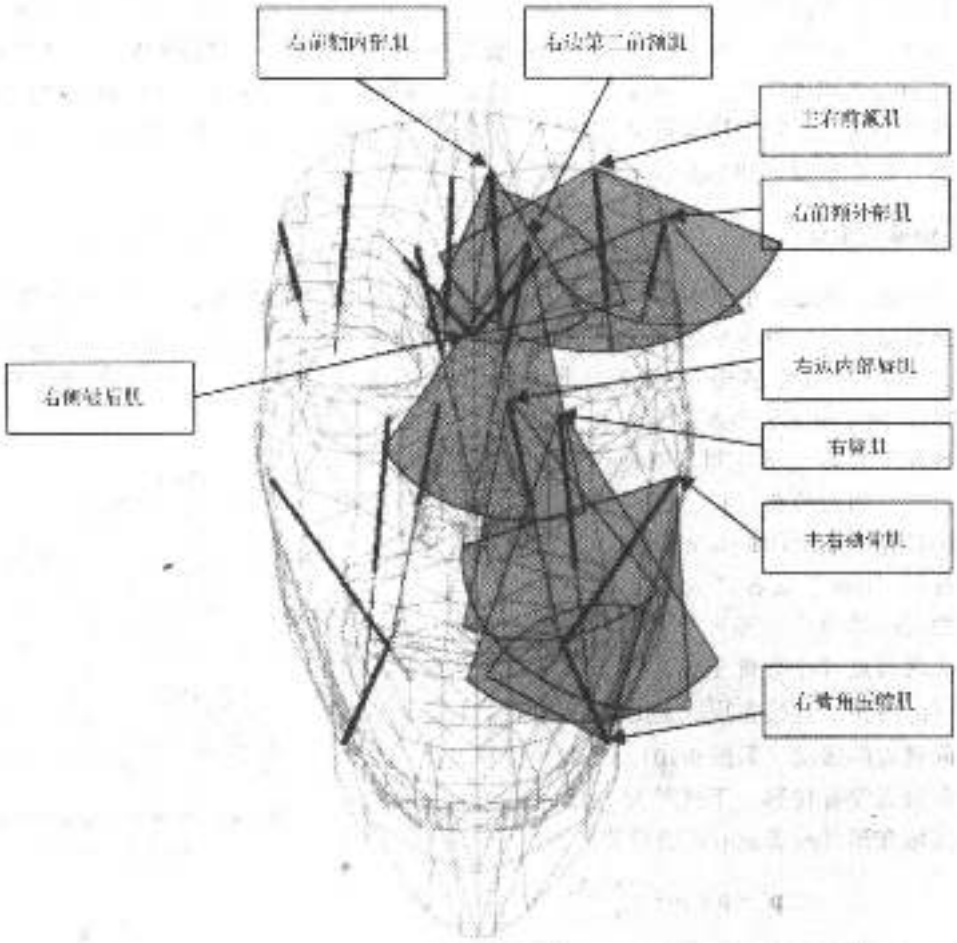


图 9-11　网格空间中 18 个线性肌肉模型的位置

$$f = 1 - \frac{\sqrt{l_x^2 p_x^2 + l_y^2 p_y^2}}{l_x l_y} \qquad g = \frac{|\mathbf{p} - \mathbf{c}|}{l_x}$$

另外一个可以实现的肌肉模型是一个薄片。在这种情况中这个区域的影响范围是一个矩形，空间均匀地变形，像纤维一样平行地拉向矩形的边缘而不是一个点。一个薄片肌肉的真实例子是额面额，它是前额肌肉的一块，用于拉升眉毛的外面部分。

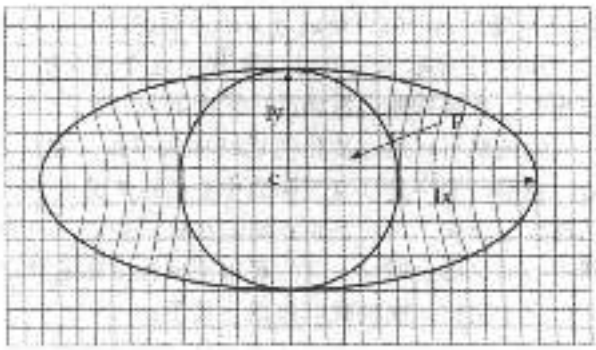


图 9-12　活动肌模型中的参数。格子代表收缩

通过这两个模型（线性肌和括约肌）的使用加上骨的转动，面部网格的变形就简化为对一些肌肉参数的操作。需要注意，这个方法独立于用于网格的

代表性方法。图 9-13 显示了 6 个使用这种模型得到的主要或普通的表情。

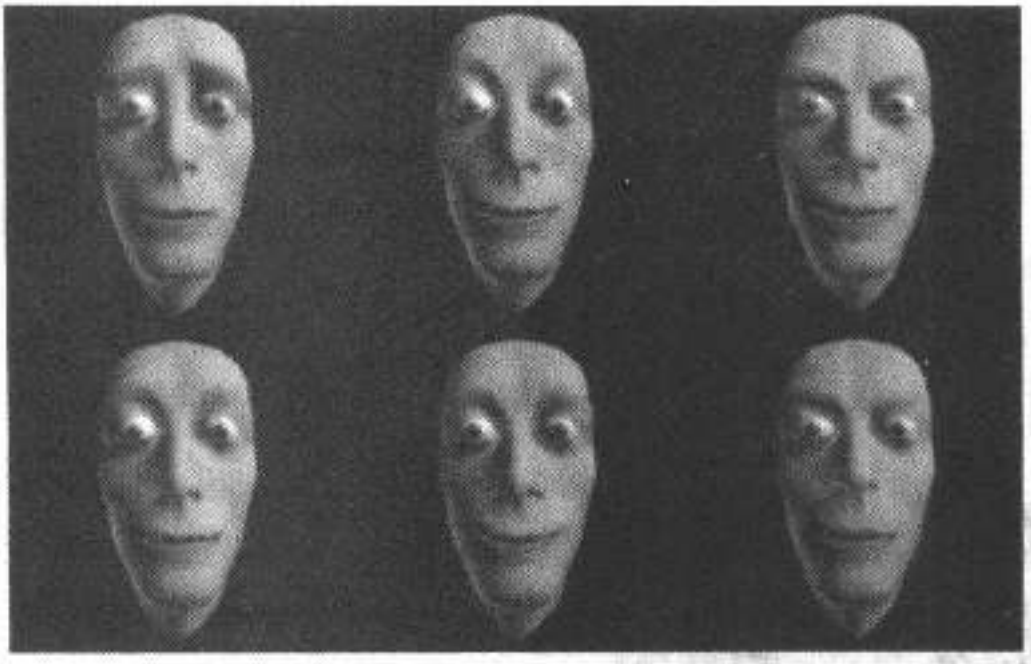


图 9-13　通过肉体肌肉模型产生的常见表情：伤心、惊讶、轻蔑、害怕、高兴、愤怒

使用这种伪肌肉模型的一个主要问题是它仅仅近似地模拟了真实面部肌肉。在这种线性肌肉的情况下，皮肤不是在以连接点为中心的受影响的圆锥区域被拉向这个连接点。很明显，伪肌肉模型可以通过很准确的模拟来更接近现实。例如，我们可以用预先写好的 FFD 脚本来更小心地模拟肌肉纤维的运动。虽然现在不能肯定这是否会更加真实化，但它也许是我们对于控制多边形网格的抽象所能得到的最好的情况。即使肌肉能被非常真实地模拟出来，我们仍然需要面对变形的模型是一个多边形网格这个事实。

9.4.5　面片技术

即使是一个很好的控制方法，也会因为多边形网格变形的固有困难而造成视觉问题。无论用什么高级的控制方法，进行多边形网格变形时都会造成在渲染几何结构时产生视觉瑕疵。惟一可以解决这个问题的办法是增加多边形网格细化程度。但是这同样会增加很多控制问题。使用双三次参数化面片是解决它的经典方法。虽然一个面片网格产生很多控制点，但是我们可以使用比多边形少很多的面片网格来达到同样的渲染效果。像在多边形网格中一样，高级控制抽象在面片网格中同样至关重要。以下显示了三个例子：(1) 存在一个伪肌肉控制层的面片网格；(2) 层次化 B 样条；(3) 视觉语音中的 FFD。(第 8 章详细介绍 FFD——包括一个简单的面部变形例子。)

面片网格 + 伪肌肉控制

这是 Pixar 公司发明的方法 [RGEV90]，他们曾说过一句格言："永远不要将多边形用于任何不平坦的东西上"。该方法最初被用于《Tin Toy》中——《玩具总动员》之前的短片。

这部作品在当时取得了巨大的成功（荣获1988年奥斯卡奖）。在图9-14（彩页中也有）中婴儿的脸被表现为邪恶的样子。这看起来好像是由皮肤动画中不寻常的皱纹体现出来的。在《Tin Toy》中的这个婴儿的面部，Pixar使用了超过2568个被向量/伪肌肉模型（43个线性肌和4个括约肌）控制的Catmull-Rom面片网格。

也许这激发了《玩具总动员》中的简单面部模型。这部动画片只对虚拟人物角色的面部动画进行了简单的处理。这远不能达到观众对虚拟人物逼真程度的要求。

层次化B样条

给定一个双三次参数化面片，一种使用原来控制点无法实现的用于局部细节形状改变的方法是细分面片（从而使得控制点的数量翻了4倍）。这个过程继续到有足够的控制点为止，此时可以通过移动一点来改变形状。层次化B样条［FORS88］是一种得到相同结果的经济的方法。面片被分层为子面片，从而实现表面上的细节形状变化。子面片的控制点数量正好可以实现需要的变化。子面片的变化不影响它们所在面片的完整性。这个概念在图9-15中有清楚的描述。



图9-14　Pixar公司1988年制作的《Tin Toy》中的婴儿

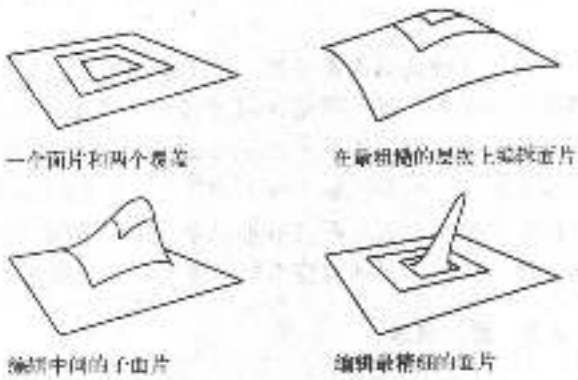


图9-15　层次化B样条——一个面片和两个覆盖

这个完整的结构形成了一个层次，不同细化级别上的覆盖的集合形成了表面。除了在细节改变的时候避免控制点数量的过分增长以外，由于此结构是多相分解的，它推动了在各个层次上的编辑和建模（见图9-16）。

肌肉模型用于移动控制点，由于肌肉收缩引起的皮肤膨胀很容易结合到这项方案中。这个方案的另外一个特殊的方面就是，肌肉模型被插入到各个层次之中或者之间。

FFD和面部动画

这项工作（参见［TAO99］）是基于自由形状变形（FFD）的，它是计算机图形建模技术中的一种方法，我们在第8章中已经详细描述过了。Tao等人用16个贝济埃容积块（volumes）

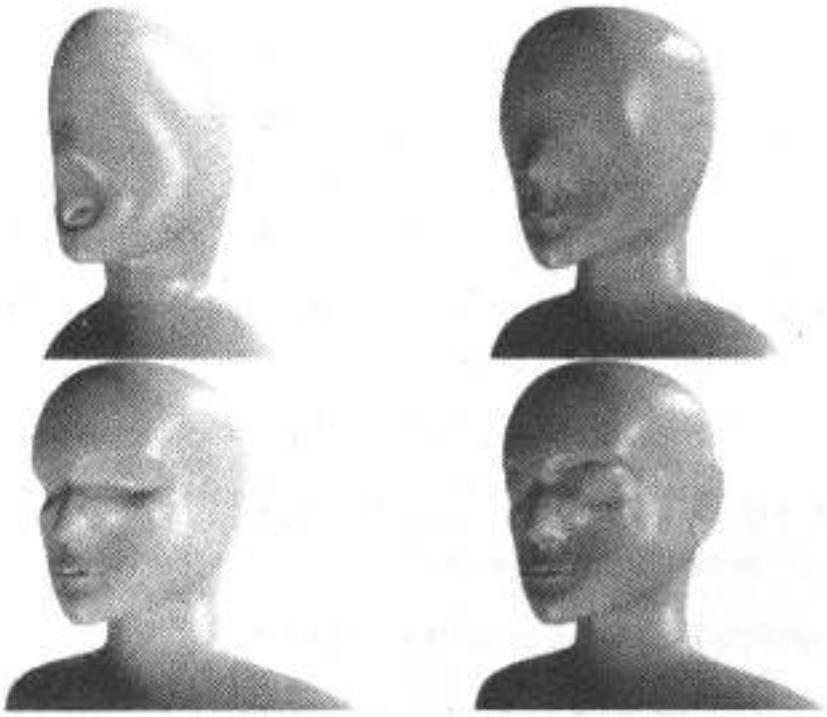


图 9-16 以 B 样条层次构建的面部

包围了一个面部模型（见图 9-17）。每一个面部网格被关联到一个特殊的贝济埃容积块中并被赋予（u，v，w）值。这便是第 8 章中所说的 EFFD。它们的集合连接表明它们必须拥有相同数量的边界控制点。

接着通过模型控制点的变形和观察真实图像交互地建立起 6 个通常的表情和 23 个发音嘴形。这是个艰巨的建模任务，但是它有个优点，就是多边形网格可以被任意分解（FFD 变形模型的主要优点）；相比之下贝济埃容积块中格子（grid）的分解可以比较粗糙。

第 i 个特殊的面部表情网格的总变形可以用矩阵形式写成：

$$\mathbf{V}_i = \mathbf{B}\mathbf{D}_i$$

这里 $\mathbf{V}_i$ 包括了每一个网格点的位移；

$\mathbf{D}_i$ 是一个向量集合，其中每一个都控制了与贝济埃容积块相关联的控制点的位移；

$\mathbf{B}$ 是包括贝济埃基础函数的矩阵。

由此在任何情况下（非刚性的）面部动作可以表示为：

$$\mathbf{V} = \mathbf{B}[\mathbf{D}_0, \mathbf{D}_1, \ldots, \mathbf{D}_n][p_0, p_1, \ldots, p_n]^T = \mathbf{BDP} = \mathbf{LP}$$

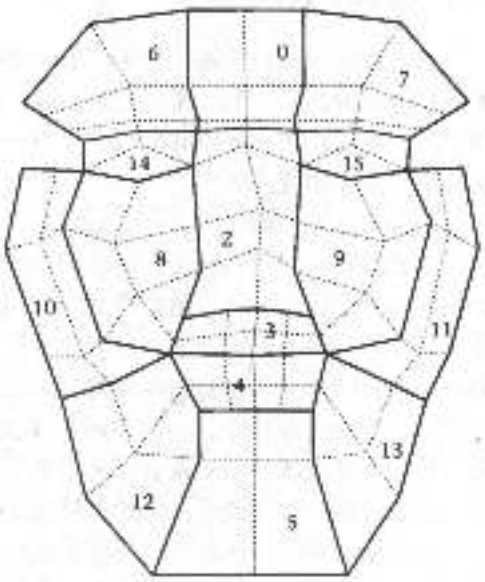


图 9-17 包围头部不同部分的 16 个 FFD 容积块

其中 p_i 是表情 $\mathbf{D}_i$ 的强度。

若包括刚性的动作 **R** 和 **T**，我们可以写出总的运动如下：

$$\mathbf{R}(\mathbf{V}_0 + \mathbf{LP}) + \mathbf{T}$$

其中 $\mathbf{V}_0$ 是中性网格。

这样一个形式给出了视频方法中基于模型的运动跟踪的基础。不像 Pighin 等人所用的方法[PIGH99]，该方法不依赖于视频与从 3D 模型渲染而得的图像之间的匹配。视频序列中选定特征点的 2D 帧间的动作是确定的。网格进行 3D 变形从而给对应的网格点提供相同的屏幕坐标。帧间视频运动向量通过模板匹配找到，在第 n 帧中的围绕特征点的像素窗口从第 $n+1$ 帧中搜寻出来。计算出的网格点运动向量定义为：

$$d\mathbf{V}_{2d} = \frac{\partial(\mathbf{M}(\mathbf{R}(\mathbf{V}_0 + \mathbf{LP}) + \mathbf{T}))}{\partial(\mathbf{T},\mathbf{W},\mathbf{P})}\Big|_{\mathbf{T}_n\mathbf{W}_n\mathbf{P}_n} \begin{bmatrix} d\mathbf{T} \\ d\mathbf{W} \\ d\mathbf{P} \end{bmatrix}$$

其中 **T** 和 **W**（旋转矩阵 **R** 的三个角）代表了 3D 模型的刚性运动；

P 代表了 3D 模型的非刚性或者变形运动；

M 是摄影机的投影矩阵。

这个等式的 LHS 可以从视频中跟踪，需要估计的未知变量是：

$$\begin{bmatrix} d\mathbf{T} \\ d\mathbf{W} \\ d\mathbf{P} \end{bmatrix}$$

这些被代入到先前帧的 $\mathbf{T}_n$、$\mathbf{W}_n$ 和 $\mathbf{P}_n$ 的解决方法中得到 $\mathbf{T}_{n+1}$、$\mathbf{W}_{n+1}$ 和 $\mathbf{P}_{n+1}$。我们所收集的 2D 运动向量比不确定的参数值多很多——一个约束过多的系统——从而可以使用一个有效的倒置（inversion）（最小平方估计）。

9.5　视觉语音

就像我们以前讨论的，视觉语音是 HCI 中期待已久的一个里程碑。它的到来毫无疑问会影响到电脑游戏。因为视觉语音的重要性，无论是在语音识别社区还是 HCI 界它都是一个热门的研究领域。在这一节我们仅会介绍一个简单的（也是粗糙的）方法，它可以实现捕捉的发音嘴形之间的变换。

在使用发音嘴形作为基础单元时，我们的假设是连接这些发音嘴形能产生一个伴随语音的可信的动画序列。动画接着被脚本化或者文本将转变为发音嘴形。这显然不能生成高品质的视觉语音。在连接语音的综合发展中我们需要有关视觉语音的课程（lessons）。如果一小组语音单元（音素）被用于语音综合系统中，产生的语音听起来虚假而且过于清晰以致有些做作。语音综合技术正趋向于存储更多的长的词汇——甚至单词——来解决这个问题。这当然又归结为 MoCap 哲学，长的序列被预先记录好。也许在不久的将来，最好的方法可能会在简单的记录或者手动的调音这两个极端之间。它将是游戏和发音嘴形变换之中所用的动画和声音的一对一对应关系。它的关键是较长的语音单元之间的变换能产生更好的品质。

研究这个基本问题的想法是给出视觉语音中困难的正确评价。视觉语音和面部动画一样是一个较难的领域。我们习惯于在观察别人嘴唇活动的同时听他们说话，这大概就是我们能够很敏感地听出视觉语音缺陷的原因。McGurk 作用是证实视觉语音重要性最好的方法，它

显示视觉可以改变对声音的感知。在这个著名的实验中听众听到一个综合的声音‘巴’。同样的声音伴随着一个人造面部出现，面部嘴唇形状是‘哇’。听众感觉听到了‘哇’，尽管听到的声音是‘巴’。这个实验阐明了在声音感知时视觉信息优先于听觉信息。

发音嘴形可以被当作视觉语音中的基本单元——它可以被描述为对应于基本听觉语音单位的嘴唇形状。发音嘴形形成了代表语音中声音的一个最小的特殊集合（见图 9-18）。为了从文本中获得视觉语音，一个句子被分裂成一些音素的序列，每个音素有一个发音嘴形与之对应。接着我们需要一些可以插入音素的方法。

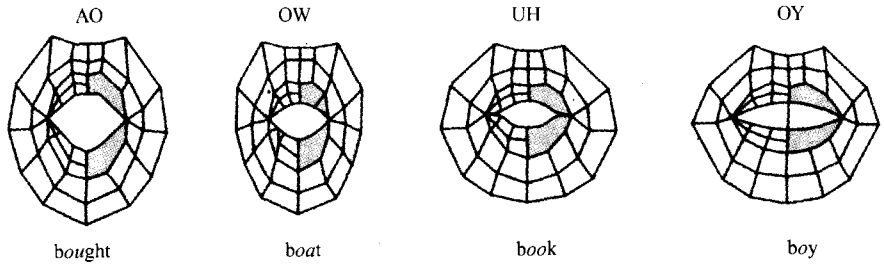


图 9-18　发音嘴形的例子

视觉语音中一个著名的问题是协同表达（coarticulation）作用。协同表达涉及一个特殊声音的音频信号的改变。它是一个关于哪些声音先到，哪些后到的函数。这种现象意味着以离散独立的单元使用音素作为视觉语音的基础是不正确的。事实上，最近在语音综合品质方面的改进主要源于这样一个事实，就是协同表达作用已经被综合到其中了。

协同表达作用的时间范围可以达到向前 5 个音素和向后 1 个音素。向前协同表达常常在当一段连续的谐音后跟着一个元音的情况下发生。协同表达不仅仅在视觉上，在听觉上也起作用。同一个音素的不同声音意味着不同的嘴唇形状。一个很好的例子是在读单词‘stew’时嘴唇呈圆形。

变换发音嘴形

主函数

首先考虑一个协同表达中音素目标值的变换的简单模型，它是由 Cohen 等人提出的[COHE93]。混合函数也叫主函数，是与每一个音素相关联的。主函数是非负的指数函数：

$$D_{sp} = \alpha_{sp} \exp(-\theta_{sp} |\tau|^c)$$

这意味着音素的持续时间随离分段中心的时间距离 τ 指数下降。这个因素随着 c 的增长而增长，并由比例参数 θ_{sp} 来调节。α_{sp} 控制音素持续时间的强度系数。τ 的计算公式为：

$$\tau = t_{Csp} + t_{0sp} - t$$

其中 t_{Csp} 是段的中心，t_{0sp} 是离段中心的偏移量。

每一个音素的每一个参数都需要一个主函数。我们的想法是通过将目标（target）值与一个因子相联系来对协同表达作用进行建模。这个因子是当前活跃的主函数的和。图 9-19 中的简单例子表示了单个主函数的协同表达值被设置为 0，而第二个片断向前运动，协同表达值受影响而产生变化。

事实上主函数同时解决了目标值的协同表达和插值。语音综合包括了将文本映射到 6 个

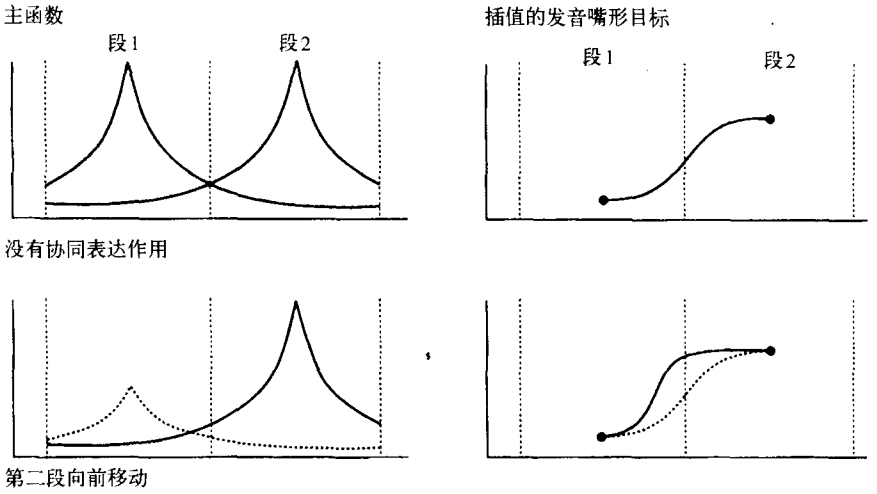


图 9-19　主函数。第一行没有表现出协同表达作用，而第二行表现出向前移动的第二个分段的影响

参数轨道上。协同表达作为这个过程的一部分进行建模。

肌肉控制和音素变换

对发音嘴形使用 Waters 肌肉模型意味着控制 12 个肌肉执行功能以及下颚的转动。这些可以通过已知的肌肉模型和嘴唇形状的相关性来交互地建立。表 9-1 总结了主要的嘴唇形状肌肉的交互作用。

表　9-1

模拟肌肉	外形动作
口轮匝肌（括约肌）	嘴唇变圆，嘴唇宽度变窄，嘴唇突出
笑肌（左/右线性肌）	嘴唇宽度增加
唇部压缩肌（左/右线性肌）	下嘴唇下降
唇部提高肌（左/右线性肌）	上嘴唇抬高
主颊骨肌（左/右线性肌）	嘴角抬高
三角肌（左/右线性肌）	嘴角降低
下颚转动肌	嘴巴张开

我们可以在发音嘴形中使用正弦函数来模拟肌肉收缩的流度，然后在其中做插值得到视觉语音。

肌肉控制和表情——发音嘴形交互作用

大部分与视觉语音相关的工作集中在嘴唇和下颚的动画上。相反地，在 9.4.1 节中描述的照片真实感工作（关于面部表情的变换）并没有语音的执行。很明显，带表情的视觉语音需要整个面部的动画。和情感内容一样，一般的语音伴随着视觉提示，例如眉毛的抬高和眨眼。肌肉模型有一个可以混合表情和嘴唇运动的机制（见图 9-20）。然而，这预示着用发音嘴形混合独立捕捉的面部表情是正确的。事实未必如此。我们可以考虑一段愤怒的语音。极端的愤怒可能产生缠结的嘴唇运动，当我们混合静态捕捉的愤怒和非情感状态下捕捉的语音

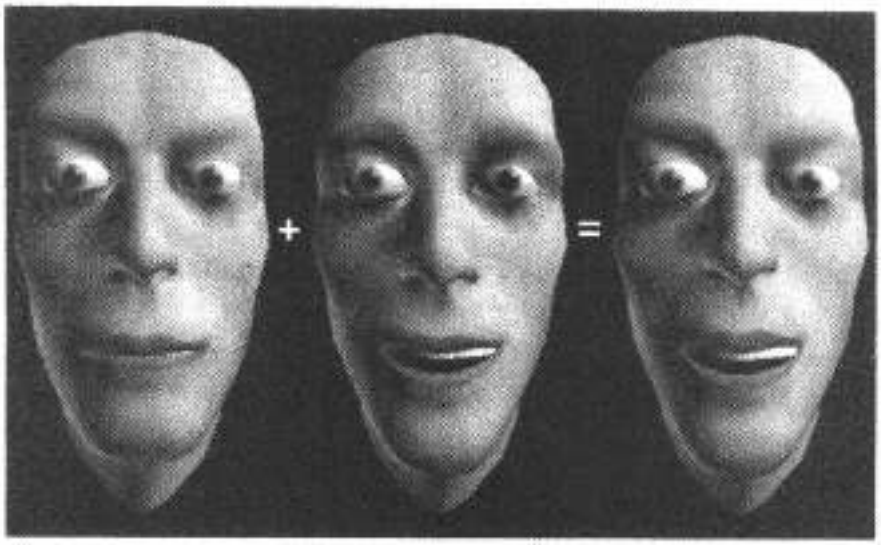


图 9-20 将愤怒的表情和发音嘴形/aa/混合

时，不会出现这种动作。

9.6 面部动画和 MPEG-4

MPEG-4 的主要目的是支持新的功能，尤其是使视听场景建模标准化。这里很重要的一部分是人造内容和自然内容的合成。电脑游戏将形成如下新型应用的一部分：虚拟对话人物，高级的人际通信系统，电子购物，多媒体广播等。在 MPEG-4 的人造内容中，一个很重要的部分是面部动画的标准化，它将支持真实或者假想的人。

基于肌肉的抽象是独立于几何表示的，它被应用到计算机图形学的很多研究中。在 MPEG-4 标准中就用到了它。就面部动画而言，MPEG-4 支持一种“分析-描述-综合”的方法。一个模型在源（source0）被分析，这个过程产生 FAP（面部动画参数）序列形式的动画信息。它们通过低带宽的通道传输并被用来在接收端渲染出网格。如果源和接收端都具有相同的网格几何形式，在这种情况下动画信息必须在通信建立阶段传输；否则可以用动画参数在接收端渲染一个不同的网格。

MPEG-4 中的面部动画所用到的其他数据是 FDP（面部定义参数）。这使得源可以在接收端设置一个面部模型或者改变一个已经存在的模型。FAP 插值表（FIT）允许源定义 FAP 的插值规则。这个工具使得源可以发送活跃 FAP 的一个子集，剩余的可以从子集中插值进去。例如，假设面部是对称的；上嘴唇内部的 FAP 可以被发送用来决定上嘴唇外部的动作等等。在这样的情况下，我们可以为了达到减少数据的目的而接受品质的降低。

基于最少面部动作的研究，68 个 FAP 被定义。两个是高级参数——发音嘴形和表情，它们有多个子参数；其他的是低级 1D 度量，它们用来描述颚、嘴唇、嘴巴、鼻子、脸颊、耳朵等器官上面的面部特征的运动。每一个 FAP 的运动都与一个中性面有关。

9.7 渲染问题

对于视频克隆以及相关的应用，流行的照片纹理贴图是一个很好的选择。然而，它有两

个很大的缺点。第一，光线/表面的交互作用是存在于数据捕捉环境中的。改变被渲染的头部的姿态将反映渲染场景的光线/交互作用。虽然这可以通过视图相关的纹理贴图来改善，但这最多是一个近似。第二，如果想对一个新的头部进行建模，我们不能使用照片纹理。

漫反射光通常基于 Lambert 余弦定律建模。它假设反射光都是各向同性的而且在强度上与入射角的余弦成比例。表面上的漫反射光模拟是不可能的，因为漫反射都是从真正进入介质的光线产生的。这些成分被吸收并在反射介质中分散开。波长相关的吸收依赖于介质颜色——入射的白色光线实际上被介质过滤了。是介质中的散射使得形成的光线近似于各向同性。因此漫反射的物理模拟必须基于表面下的散射。

Hanrahan 和 Krueger［HANR93］模型是基于物理的，用于散射和漫反射的研究。提出者特别指明它适合于自然界里的分层介质，例如生物组织（皮肤、树叶等）和无机物（雪、沙子等）。这个模型产生的结果当然是各向异性的——反映了很少的介质呈现各向同性漫反射行为。

Hanrahan 等人通过对皮肤进行双层建模来解释这个问题：外层有组织和色素颗粒，它们包含能够选择性吸收光线并产生褐色现象的黑色素；内层的血液和组织吸收绿色和蓝色，并假设它们引起了各向同性散射。

反射光线在表面上形成一个点，它被指定如下：

$$L_r = L_{rs} + L_{rv}$$

其中 L_{rs} 是由于表面散射形成的反射光线——有缺陷的镜面反射——而 L_{rv} 是由于表面下的散射形成的反射光线。

决定表面下的散射的算法基于一个 1D 传输模型并用 Monte Carlo 方法解决。

图 9-21 显示了简单情况下的这些反射现象。第一行显示了以入射角为函数的高/低镜面反射。当入射角高（high）时，反射光线受到表面散射或者镜面反射的影响；当入射角低（low）时，反射光线受到表面下散射的影响。第二行显示了由于表面下散射造成的反射圆裂片（lobe），并且可以看到介质能够显示出向后的、各向同性的或者向前的散射行为。（底部圆裂片并不决定 L_r。但是当考虑多层介质和薄的透明且背光的介质时它也是重要的因素。）第三行表示将 L_{rs} 和 L_{rv} 结合一般会造成各向异性的行为并显示出以下一些一般的特征：

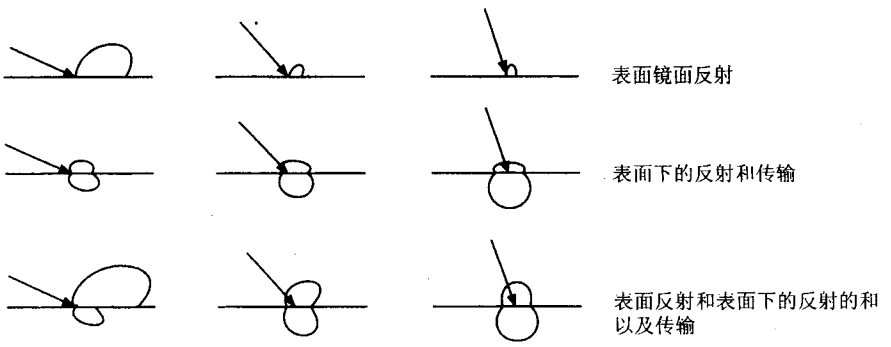


图 9-21　显示光线和物体为了成功渲染出皮肤而进行的交互

- 当介质层由于表面下散射增加而变厚时，反射增加。
- 表面下的散射可以是向后的、各向同性的或者向前的。
- 在与 Lambert 规则的半球比较下，表面下散射造成的反射将生成函数，这些函数中圆裂片顶部是平坦的。

这些因素导致了模型和 Lambert 规则之间细微的区别。这使得渲染的皮肤看起来很逼真。

9.8 总结和问题

9.8.1 参数化与照片真实性

使用高级参数化作为控制面部网格的方法的驱动力是希望用这种抽象得到特别的动画序列。对于高级抽象的研究到目前为止仅是部分成功。这也许是因为现实是对抗参数化的。面部表情的细微变化很难用很少的参数来表示。基于模型的跟踪一般能生成较好质量的动画，但是在动作捕捉时会受到其固有缺点的影响。然而，通过视频克隆的形式，参数化在游戏应用中大有潜力。

9.8.2 网格表示

毫无疑问，大多数工作（除了 Pixar 和面片表示）都是通过多边形网格来实现的。这反映了在主流的计算机图形应用方面多边形网格长达二十多年的统治地位，而面片表示只用于一些专门的领域。当然，多边形网格的视觉缺陷可以通过进一步分解来解决，但是同时也增加了控制问题的难度。

对比以平滑（smooth）/可预知（predictable）的方式变形的面片网格，多边形网格的变形使得网格离开它所在的建模的多边形分解中，这可能造成视觉缺陷。一个面片网格可以很容易地支持由于肌肉收缩造成的皮肤褶皱；这也许是多边形网格所不能达到的。

9.8.3 皮肤的渲染

对于实时应用而言，高质量的皮肤渲染还是一个未能解决的问题。

9.8.4 没有声音很多面部动画更好看

视觉语音中的一个主要困难是如何才能从一个有限的样本集中（例如发音嘴形）生成逼真的新句子，使得对观察者来说声音与画面自然地匹配。很多动画在关掉声音的情况下看起来更逼真。我们看到的和听到的存在着冲突——大概我们对模拟的协同表达中的差异非常敏感。用这种方法从文本中得到同时感知的动画也许比动画中的照片真实化更难。文本综合和制图都或多或少得到了解决，但是将它们结合起来是一个困难的事情。

9.8.5 情感和语音

在很多应用中情感语音是至关重要的——例如电脑游戏。嘴唇的形状必须在没有感情的说、一般性的说以及愤怒的说之间变化（否则声音听起来不可能有区别）。在生成表情动画时，就嘴唇形状而言，通过将静态表情和静态中性发音嘴形混合来模拟视觉语音是不正确的。

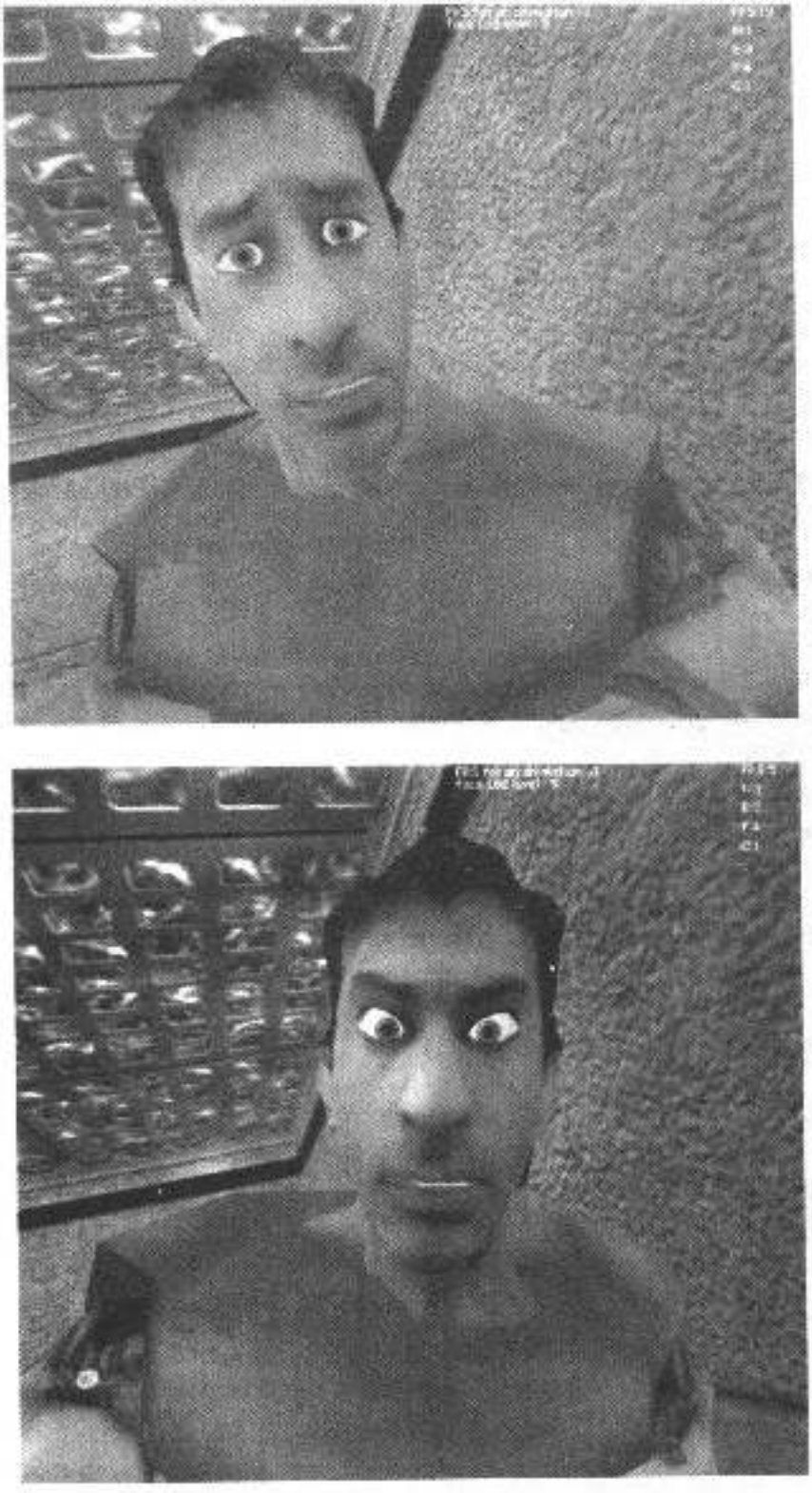


图 A9-1 Hy3D 执行中的两个肌肉模型（Emmanuel Turquin 授权，雷丰尔德大学）

附录 9.1　一个伪肌肉模型的实现

这里演示了一个使用肌肉模型方法的面部动画。这个动画在三种表情之间连续地插值。

为了看得更加清楚，用下述方式关掉动画演示中的身体动画、在编辑器中选择 anim_face/animface 为 1 而设置 body_anim（在参数窗口中）为 0。这个动画演示还包含了一个 LOD 设备，运用该设备，图像距离函数控制下的活动肌肉数量更少。图 A9-1（彩页中也有）显示了动画中两帧图像。

第 10 章　基于运动捕捉的角色动画

10.1　简介

尽管第一次使用运动捕捉（MoCap）是在电视和电影产业，而游戏产业却是第一个将这项技术作为常规方法来制作人形动画的。现在游戏产业对该项技术的应用占所有应用的 85%～90%，并且几乎所有使用此类角色的游戏都使用 MoCap 来驱动动画。理由很简单：动画的质量比动画师做得好并且动画脚本的生产代价有更低的倾向。

MoCap 的方便性和高质量是很有说服力的。考虑图 10-1，尽管该图是一个简单的线状骨架，但是根据 MoCap 数据产生的动画却能展现出复杂而极富表现力的运动。

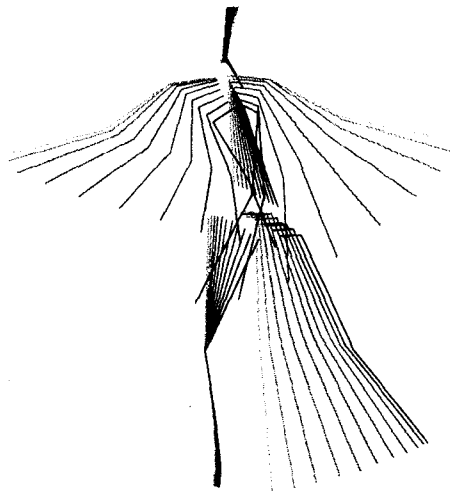


图 10-1　应用到线状骨架的 MoCap 数据

图中“幽灵”一样的序列显示了等时间间隔采样得到的分层关键帧。如果运动被充分地采样，这种技术能捕捉人物运动中所有的微妙细节。考察这幅插图，可以容易地看出运动的减速（右腿）。使用关键帧动画技术，从幽灵帧中的两个表示运动起点和终点的关键帧不能生成正确的运动。要用关键帧系统模拟这样的运动序列，动画师必须把关键帧放置在合适的位置，以便在最终的生成序列中能够展现出加速和减速。正是达到此目标的困难性和高代价促使了 MoCap 数据的使用。

现在，MoCap 技术已经比较成熟了，本章我们不关注数据收集技术以及低层次的处理，而是关心那些旨在弥补 MoCap 缺点的技术。当然，对未经加工的 MoCap 数据的低层次的后处理也是极其重要的，也需要相当多的精力。这些操作主要是清理工作，包括去噪，填充由于一个操作对象离开视域一定时间而引起的沟，克服由于两个操作对象重合而产生的走样。另一个重要的低层次处理是把数据——该数据以时间函数的形式标识出操作对象的位置——转换成关节旋转量的形式以适于驱动分层的骨架。

当前 MoCap 在游戏产业中的应用构成是，存储游戏中的序列词汇表，随着游戏的进行混合这些序列并使之适应实时要求（见图 7-1）。耗时且复杂的操作，比如目标重定（在与 MoCap 记录角色不同比例的角色上使用 MoCap 数据），是在离线时进行的。随着技术的发展，更多的复杂问题进入实时领域，这种情况可能会发生改变。在最新的报告中，Shin 等人在[SHIN01] 中处理了低层次的操作和实时目标重定。他们的应用程序是电脑木偶，定义了一个实时应用程序。这里，表演者的运动映射到一个动画角色，原型系统成功地为一个儿童电视节目创建了一个虚拟角色以及一个新闻播音员。将来电脑木偶可能会在多媒体领域找到用武之地，但是现在我们知道，传统的 MoCap 处理能够实时进行，这对于游戏产业来说意义

重大。

MoCap 技术的两个主要的缺陷是：

1）我们只能在游戏中使用事先录制好的脚本。这一点非常明显，但重点在于：尽管大量的序列（相应于游戏逻辑）被实时地存储和选择，这仍然是有着天生限制的进程。我们希望有一些工具，MoCap 序列能持续地适应不断发展的游戏，从已有的材料改变它们自身并产生新的序列。

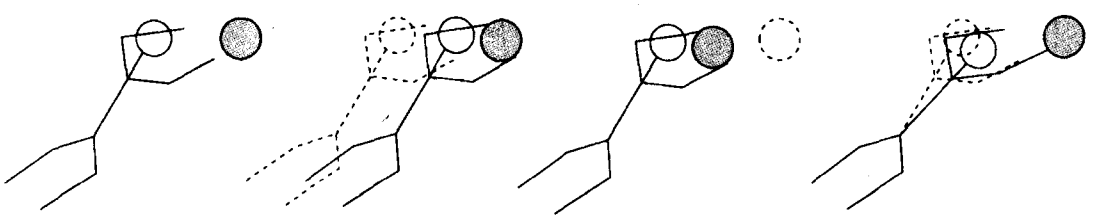
2）MoCap 数据仅对与这些数据所记录的表演者比例相同的虚拟角色才是有效的。当我们试图对不同比例的角色使用这些数据时，就遇到了困难。这就是所谓的目标重定问题（re-targeting problem）。

这种技术里还有很多其他难缠的问题。例如，对品质的考虑（这曾经意味着更少地借鉴电影产业）源于这样的事实，即：对角色皮肤表面上的点运动的采样得不到能使刚性连接的骨架运动起来的完全精确的“剧本”。好在当前对游戏角色动画的审美学要求比电影低，这促使 MoCap 技术在游戏中大规模应用。

在声称要处理游戏实时动画问题的应用中，处理 MoCap 的动机，是希望为 MoCap 数据开发的操作技术能够以交互的速率应用；并且很多是在游戏运行的时候应用的。理想情况下，我们希望一个人形角色的运动能随着游戏剧情的发展而产生个性的变化。最简单的操作可能就是简单的运动加速或减速。或者，我们可能要角色反抗，可以是通过变得生气或厌烦来达到此效果，人物个性就通过它走或跑的方式表现出来。我们希望通过使用某种方式修改现有的 MoCap 数据（例如，通过对现有序列的插值或混合）来实时产生这类运动。

该方法是由 Rose 等人在 [ROSE98] 的工作中作为例子提出的，其中的一幅图见图 10-8。图中显示了沿着两条情感线——快乐线（垂直线）和知识线（水平线）——的步行采样。每一个方框代表了一个运动序列。绿框是运动的例子，而黄色框则是插值和外推值。Rose 等人指出当只有有限个（黄色）运动序列指定时，存在合成运动的一个连续范围，此方法是可行的。

另一个普遍的要求是改变运动以满足一个约束。对于让角色捡起物体这个运动，运动数据可能是必要的。如果在游戏中角色要捡的物体更大或者在一个相对不同的位置，怎么办？例如，考虑一个足球游戏。守门员角色可能由 MoCap 库驱动，库中保存了：“扑向地面”、“扑向死角”等运动。所以守门员够到了球，游戏可能成了简单的“瞬间移动”或转移运动（见图 10-2）。更准确的方法应当是使用反向运动学（IK），改变运动序列中守门员的位置，以使他能抓到球。当然，这种情形还依赖于游戏逻辑的要求。情况也许会是这样，即我们要



a) 无修改的　b) 角色平移　c) 球平移　d) IK修改。在第一代游戏中常实现b)和c)

图 10-2　守门员可能性

求守门员抓住球但是却没有把手放置到球位置的 MoCap 序列——即使使用 IK 适应。这样，我们就只能瞬间移动了。或者，我们必须用自适应的 MoCap 序列覆盖球所有可能的位置和轨迹。

第一代使用运动捕捉的 3D 电脑游戏建立了运动脚本库，包含大约 200 个序列。一个游戏事件为角色选定一个特殊的序列，并从当前正在显示的序列过渡到新的序列。实时运动处理工具能大大地扩展这个库；或者，削减游戏中所需角落提序列数。

10.2　运动数据

我们把运动数据看成是简单时间变量的函数集 $f_i(t)$。每一个对应层次结构中的一个自由度。将其应用到如第 7 章所介绍的简单骨架的关节中。每个关节有 1～3 个自由度，我们对每一个自由度都使用单独的运动函数。一个计算机图形角色的自由度数量可达 40～50 个，而真实人体骨架有多于 250 个自由度。最初，这些函数都是离散的但是可以转化成连续函数，例如，通过插值变成（连续）B 样条曲线：

$$f_i(t) = \sum_{j=0}^{m_i \le 25} \mathbf{p}_j \mathrm{B}_j(t)$$

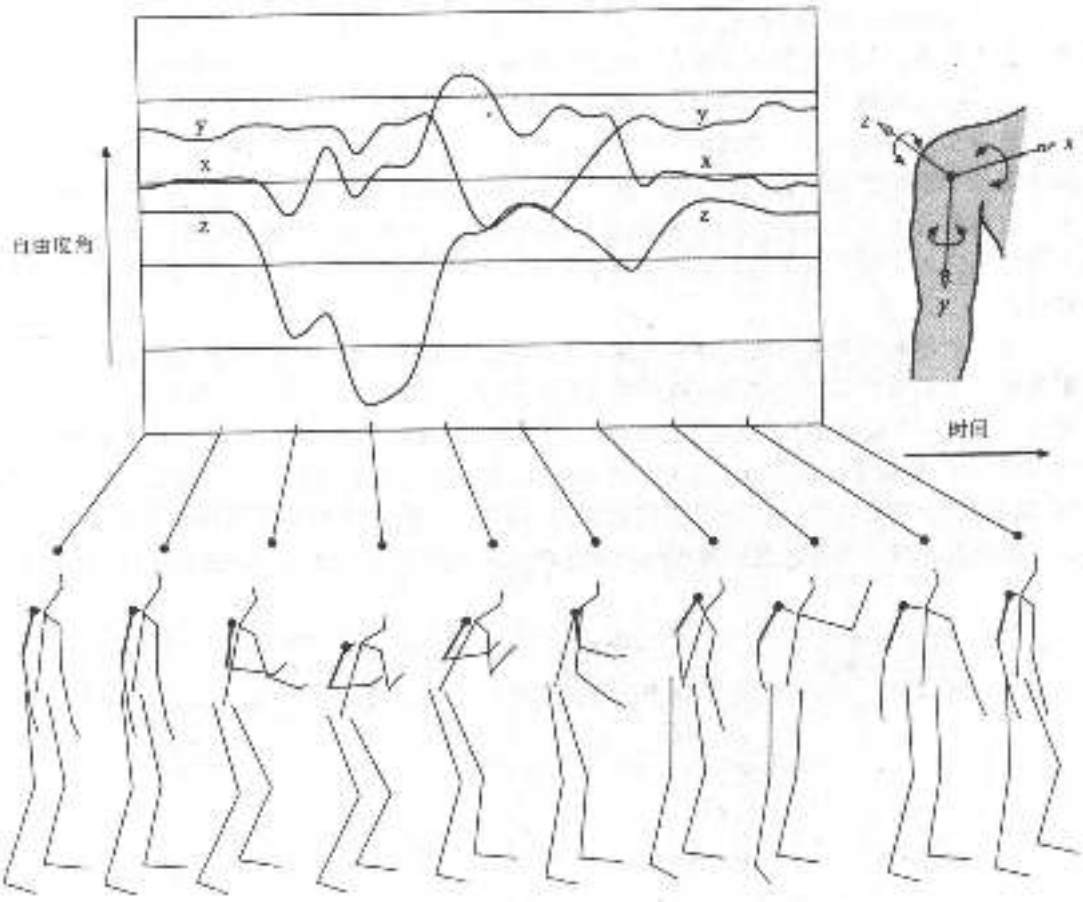


图 10.2　左膝（底）球窝间，上臂前 3 条自由度曲线

这样每条曲线均可以由控制点表示。

图 10-3 是一个角色接球时控制上臂关节角度的 3 个函数$f(t)$。图中显示了 3 条曲线，这些曲线控制了右上臂相对于肩关节的 3 个自由度。显示坐标系以关节点为中心。曲线下面的线条图加粗部分显示的是受控链。图中表示的运动是守门员抓住球然后抛出。这是一个非周期的瞬时运动（运动从开始到结束经历的时间很短）的例子。

运动曲线指定了绕关节三个局部坐标轴的相对于“放松”姿势的旋转量。我们注意图中曲线的以下特征。首先，这是一个瞬时运动；链随着运动的进行而移动，然后回到与运动开始时方向大致相同的状态。这与那些循环运动（如行走（见图 10-9）等不断重复的运动）形成了对比。其次，我们看到 Y 的旋转变化与 X 和 Z 的变化出现不协调现象（出现过负向最高点），而后两者的变化是大致协调的。第三，在角度函数和蹲伏伸展运动之间看起来有某种关系。在主瞬变基础上是其他一些较小的瞬变。这样微妙的身体运动细节用手工做起来是很困难的，因而有人猜测，正是很好的细节（比如瞬变时相位的区别）才达到了运动的真实感。但动画师只能通过反复尝试和调整来实现这样的细节。

10.3　骨架和 MoCap——BVH 格式

当首次收集而未经加工时，MoCap 数据是“平移的”；对于其每一个操作对象，它包含那个点以时间函数表示的位置。它必须被转换成能够在分层骨架中驱动关节旋转的格式，就如我们在上一节中描述的一样。这种格式依赖于层次的组织。在本节我们将看到一种典型的数据格式/骨架表示方法，就是 BVH 格式（Biovision Hierarchical Data）。使用这种骨架的动机见第 7 章。不管我们是否使用关键帧动画或 MoCap 数据，始终是有这些优点的。

下面是分层结构的说明或者说是 BVH 文件的文件头。缩进表示层次。每一个子节点指定了与其父节点间的常量偏移。当角色运动时，平移和旋转就应用到根节点（臀部），而其他节点则只应用旋转变化。关键帧应当使用完全相同的结构。惟一的区别是我们的数据更少——仅仅是每个关键姿势的旋转——而不是由 MoCap 设备以（比如）每秒 30 次采样所得的旋转量。

ROOT Hips

```
{
  OFFSET 0.00  0.00  0.00
  CHANNELS 6 Xposition Yposition Zposition Zrotation Xrotation Yrotation
  JOINT LeftHip
  {
    OFFSET 3.430000  0.000000  0.000000
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT LeftKnee
    {
      OFFSET 0.000000  -18.469999  0.000000
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT LeftAnkle
      {
        OFFSET 0.000000  -17.950001  0.000000
        CHANNELS 3 Zrotation Xrotation Yrotation
        End Site
        {
          OFFSET 0.000000  -3.119996  0.000000
        }
```

```
      }
    }
  }
JOINT RightHip
  {
    OFFSET -3.430000  0.000000  0.000000
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT RightKnee
    {
      OFFSET 0.000000  -18.809999  0.000000
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT RightAnkle
      {
        OFFSET 0.000000  -17.570000  0.000000
        CHANNELS 3 Zrotation Xrotation Yrotation
        End Site
        {
          OFFSET 0.000000  -3.250000  0.000000
        }
      }
    }
  }
JOINT Chest
  {
    OFFSET 0.000000  4.570000  0.000000
    CHANNELS 3 Zrotation Xrotation Yrotation
    JOINT LeftCollar
    {
      OFFSET 1.060000  15.330000  1.760000
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT LeftShoulder
      {
        OFFSET 5.810000  0.000000  0.000000
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT LeftElbow
        {
          OFFSET 0.000000  -12.080000  0.000000
          CHANNELS 3 Zrotation Xrotation Yrotation
          JOINT LeftWrist
          {
            OFFSET 0.000000  -9.820000  0.000000
            CHANNELS 3 Zrotation Xrotation Yrotation
            End Site
            {
              OFFSET 0.000000  -7.369996  0.000000
            }
          }
        }
      }
    }
    JOINT RightCollar
    {
      OFFSET -1.060000  15.330000  1.760000
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT RightShoulder
      {
        OFFSET -6.060000  0.000000  0.000000
        CHANNELS 3 Zrotation Xrotation Yrotation
        JOINT RightElbow
```

```
        {
          OFFSET 0.000000  -11.900000  0.000000
          CHANNELS 3 Zrotation Xrotation Yrotation
          JOINT RightWrist
          {
            OFFSET 0.000000  -9.520000  0.000000
            CHANNELS 3 Zrotation Xrotation Yrotation
            End Site
            {
              OFFSET 0.000000  -7.140012  0.000000
            }
          }
        }
      }
    }
    JOINT Neck
    {
      OFFSET 0.000000  17.620001  0.000000
      CHANNELS 3 Zrotation Xrotation Yrotation
      JOINT Head
      {
        OFFSET 0.000000  5.190000  0.000000
        CHANNELS 3 Zrotation Xrotation Yrotation
        End Site
        {
          OFFSET 0.000000  4.140008  0.000000
        }
      }
    }
  }
}
```

10.4　运动数据的基本处理

现在我们介绍修改 MoCap 数据的一个简单而基本的方法。“基本”是指算法直接操作数据，而不需要借助数学模型（比如插值和信号处理技术）。在游戏中我们主要希望这些操作能够实时地进行——按照心情和环境改变角色的运动。不管使用何种方法修改 MoCap，都有一个限制，即如果不以破坏初始数据的完整性及逼真程度为妥协的话，我们进行修改时的自由度又是多少？如果编辑操作产生了不自然的运动，这就意味着我们应该使用新的序列。修改 MoCap 序列的动机是丰富可用于游戏的数据库，从而丰富玩家的视觉体验。

10.4.1　加速和减速运动

如果要让一个角色跑得更快或更慢，我们可以通过修改 MoCap 数据来使运动加速或减速。虽然说原理简单，但这个操作还是需要认真对待的。

考虑使运动加速，这里我们把数据变换为：

$$f' = f(kt)(\text{其中 } k \text{ 是常数})$$

这意味着把每单位时间处理 n 个采样的数据转换成每单位时间 m 个采样（$m < n$）。我们把这种变换同等地应用到所有的运动曲线，以产生对整个运动的统一加速。然而，如果用越来越低的采样率对数据重新采样，会遇到走样问题。频率部件本应使频率增加一到两倍，却可能导致其频率减小或部件清零。最终的效果是实际的运动减慢了，而不是加速。要防止这种情况的发生，我们必须首先确定加速的极限是什么，然后在重采样之前应用一个反走样的滤

波器，这样可以滤掉数据中的高频内容（这是不能被新的采样频率正确采样的）。更多细节会在 10.6 节中讨论。

所以，如果我们确定要使用的最小采样率是 m，则意味着加速了 m/n，运动数据应经一个定点频率为 $2m$ 的低通滤波器滤波。当然，如果高于这个频率，内容就被损坏了。尽管这个简单操作的全部效果就是加速运动，它也必然地损坏高频内容。从降低采样率一定会损坏信息这个事实可以直观地看到这一点。我们要用更少的样本来表示这些数据。在 10.6 节还会就这一点进行讨论。

将运动减速意味着用更高的采样频率对数据重新采样。但这些数据已经在运动捕捉时采样过了。要增加初始采样频率，需要进行插值操作并将离散的表示转化为曲线（见 10.5 节），以便从样本间获得 $f(t)$ 的估计值。

注意，在一般情况下我们也需要插值来使运动加速直至加速因子 n/m 是一个整数。

10.4.2　混合和时间扭曲

当我们要生成在两个不同运动之间的过渡效果时，要对运动数据进行最一般的混合操作——如从走到跑。通过在“淡出”第一个序列的同时“淡入”另一个序列，可以简单地实现这种效果。这个简单的想法在第 7 章的关键帧动画部分已经介绍过了。

对所有的自由度 i

$$f_{Ti}(t) = \alpha(t_a) f_{Mi}(t) + (1 - \alpha(t_a)) f_{Ni}(t)$$

其中：

T 是过渡序列，M 和 N 是待混合的两个序列；

$\alpha(t_a)$ 在过渡期间从 0 到 1 取值。

如果曲线是由 B 样条控制点表示的，我们有：

对所有的自由度 i

$$d_{ij} = \alpha b_{ij} + (1 - \alpha) c_{ij} \quad 0 \leqslant \alpha \leqslant 1$$

其中 b_{ij} 和 c_{ij} 是待混合序列的控制点。

所谓“混合”是一种没有技术合法性的技术，意思是如果我们把走和跑混合起来以产生一个过渡，这样的过渡运动不太可能会与真实的过渡相匹配。然而，它还是很有效的，产生的运动还是相当对齐而且相似的。“相似”是指对于最好的结果，它们对同一个运动（比如行走）有不同的表现。“对齐”是说关键姿势——比如地上的一只脚——必须在两个序列中同时发生。这涉及到时间扭曲，它是非均匀重采样。我们伸展和收缩某个运动中的样本，以便使运动序列中的关键姿势与另一个序列中的同步调发生，然后进行混合。事实上，这是使运动加速和减速的常规方法。现在，我们按要求的同步方式不断修改时间。图 10-4 显示了一个例子。在该例中，我们要求姿势 k_1、k_2 和 k_3 在 t_1、t_2 和 t_3 时刻发生来匹配另一个序列中的关键姿势。

（回顾第 7 章介绍的方案，在该方案中，我们明确指定了角色的姿势，这样使得一个序列的结尾能与下一个序列的开头相匹配。然而在这种情况下，我们只关心角色的方向而不是肢体的整个姿势。）

我们在上一节中提到，若以比原始采样频率更低的采样率采样，无论何时重新采样都会

遇到问题。在这里也存在这样的考虑。

两序列间的时间扭曲可以用来生成不同的运动，而不需混合。考虑两种行走运动——一个是轻快步态，一个是酒醉步态。对酒醉步态用时间扭曲使之与轻快步态对齐，就产生了醉态的轻快步调。而用另一种方式扭曲则产生轻快的醉态步调。

在两个序列之间做混合或插值会产生新的序列，有时这称作多目标插值，这也是可以使用的。这里，我们可以在轻快步态和疲惫步态序列中插值以产生一个两种特征兼备的序列。

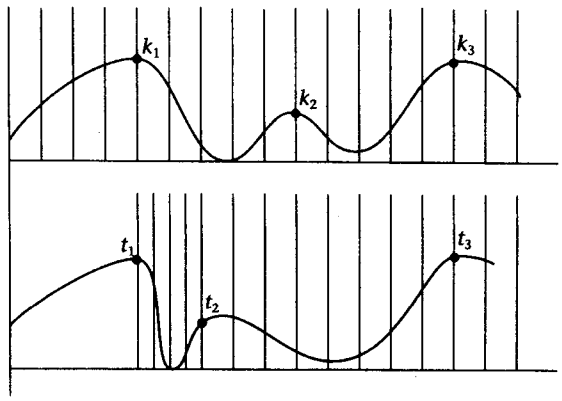


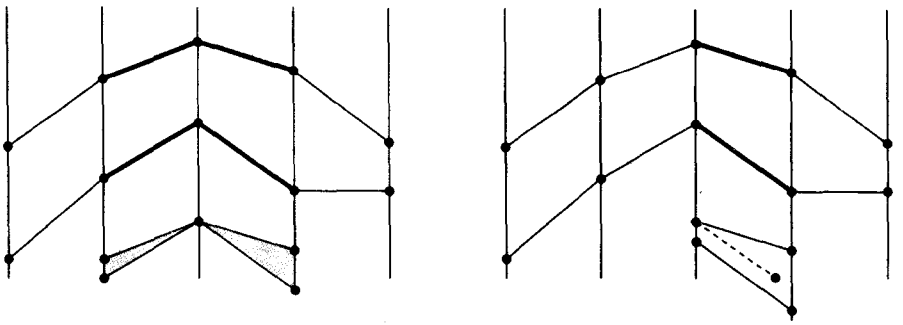
图 10-4　时间扭曲收缩和拉伸采样。在该例中，我们要求姿势 k_1、k_2 和 k_3 在 t_1、t_2 和 t_3 时刻发生

10.4.3　对齐运动序列

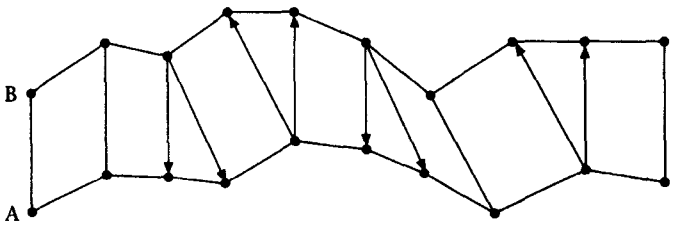
现在再来看看时间扭曲的细节问题。尽管上面我们考虑了采样间隔的变化——收缩和拉伸，但是并没有说明如何控制这个过程——如何知道在哪里进行时间收缩和拉伸。要进行时间扭曲，我们要找到对齐点。自动的时间扭曲算法必须找到收缩和拉伸时间的最佳组合，以使一个序列对齐到另一个序列。正如我们将要看到的，这是一个高成本的过程，因此只是离线方法。在开始之前我们要指出，找到合适的时间扭曲也许是不可能的。对于许多复杂的运动，如跳舞或体操，要找到对齐点是不可行且不合适的，而企图混合不相称的序列一般会产生不切实际的“高难度”运动。

Bruderlin 和 Williams［BRUD95］采用了一种几何方法，注意该问题与轮廓数据的三角形划分有关。[㊀]对于每个有 n 个样本的序列对，算法试探了序列 A 的顶点中 n 个点与序列 B 的 n 个顶点之间多种不同的对应性，并从中选择最好的一对。处理每个“对应”的代价是通过“工作量”来度量的，即：算法在将一个信号通过收缩或拉伸的方式变形为另一个信号的过程中所用的工作量。（注意，在图 10-4 中为把问题可视化，我们显示了收缩的时间间隔，但在任何实际方法中时间间隔都是常数，并且有这样的情况：或者两个信号间存在某种对应性，或者一个序列中的多个样本映射到另一个序列的一个样本上。）其代价函数是局部拉伸和混合之和。混合是每个信号中对应顶点三元组之间的角度差的函数（见图 10-5a）。而拉伸定义为对应顶点对间的距离的函数。当代价取到整个序列中的最小值时，一个顶点对应关系就可用了。在图示的例子中，或者 A 中的两个样本对应 B 中的一个样本，或者 B 中的两个样本对应 A 中的一个样本。之后，这种关系应用到算法的第二步：将 A“扭曲”到 B 或者相反。如果 B 扭曲到 A（记为 B_w）并且有 B 的多个样本与一个样本 A_i 对应，那么通过这些样本的平均值给出扭曲的 B。另一方面，如果 B 中的一个样本对应了 A 中的多个样本，我们就要通过对 B 进行曲线插值来找到新点。Bruderlin 和 Williams 将情况分为如下几种（从 B 的角度考虑）：

㊀ 这是从硬件（如激光测距仪或医学造影仪等）把数据转换成由三角形构成的计算机图形对象的问题。未经处理的数据由处在平行平面的封闭轮廓构成，这些轮廓代表扫描真实物体时的交叉部分。该问题是，用直线连接轮廓的采样点把轮廓之间的空间三角形化，从而把整个空间三角形化。封闭轮廓被转换成三角形化的对象。要实现这一点，必须采用一种能产生与这些轮廓曲面尽可能吻合的计算机图形曲面的方法。



a) 代价函数项



b) 应用代价函数后选择的对应关系

图 10-5　基于拉伸和混合代价的时间扭曲算法

- **置换**　连续样本的 1:1 对应。
- **删除**　B 中的多个样本映射到 A 中的多个样本。
- **插入**　B 中的一个样本映射到 A 中的多个样本。

为了计算 B 的扭曲版本（B 映射到 A），我们有：

- **置换**　如果对应是 $A_i = B_i$ 则 $B_i^{warp} = B_j$。
- **删除**　这是指 A_i 对应于 $(B_j, B_{j+1}, \ldots, B_{j+k})$，此时，$B_i^{warp} = average\_of(B_j, B_{j+1}, \ldots, B_{j+k})$。
- **插入**　意思是 $(A_i, A_{i+1}, \ldots, A_{i+k})$ 对应于 B_j，此时，我们要找到 $B_i^{warp}, B_{i+1}^{warp}, \ldots, B_{i+k}^{warp}$。这是通过对初始值 B_j 外推 B 样条做到的。

10.4.4　运动扭曲

运动扭曲或者运动偏移映射是指对运动数据信号的振幅做局部的调整。这样做的目的是在调整的同时保持数据的全局特征。比方说，有一个行走运动，我们希望角色走过一扇矮门。若已知一个角色走过门槛瞬间的关键姿势，就可以用它修改（代替）行走的运动数据。我们把这个操作简单定义为：

$$f'(t) = f(t) + d(t)$$

其中 $d(t)$ 是所需的偏移。

在这个表达式中，所需的振幅偏移被明确地定义为一个单独的函数——即所谓的偏移映

射。这是很方便的事，它使我们能越过很多帧把所需的偏移$d(t)$加到原始运动$f(t)$上。但简单地插入一个新的生存期仅一帧的运动会引起运动的不连续。$d(t)$可以通过从原有姿势中减去新的所需运动得到。在最简单的情况下，一个简单的新关键姿势$d(t)$会是一个单值（见图 10-6）。

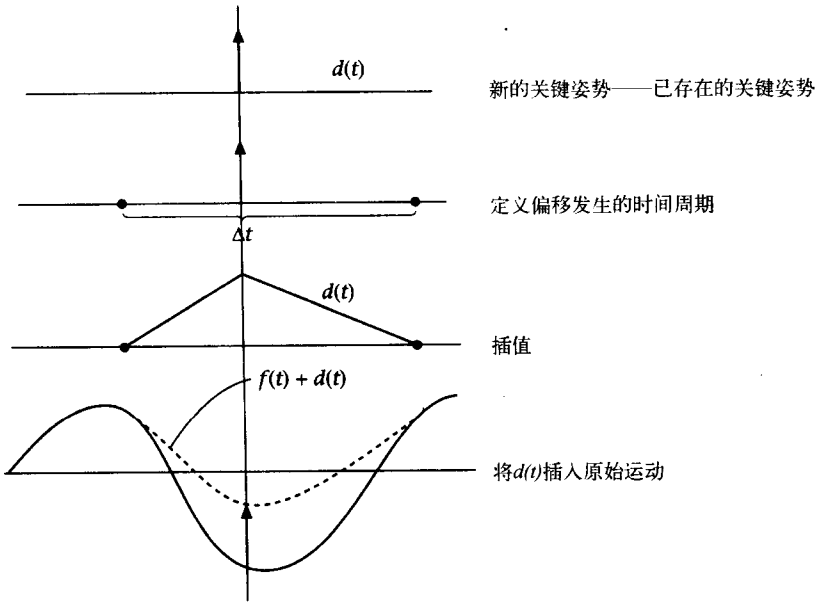


图 10-6　偏移映射

可以通过（比如说）线性插值在一段时间 Δt 内将它加到$f(t)$上，它可以“延展”$d(t)$以使之在 Δt 期间为非 0 值。所以分离偏移映射使我们能够定义它与$f(t)$合并的速率。（注意，这种方法只是 10.2 节使用的单帧“反应手势”的推广。）

这种方法对于实时的低层次运动控制的意义是，在设计 MoCap 驱动的角色运动时，我们要向前看，预测与物体或障碍的交互，从物体中提取适当的偏移映射，并为角色抓物体的过程（正在抓，但还未碰到）生成新的运动曲线。把偏移映射与物体组织在一起给了我们一个很自然的结构，比如，可用来计算在角色遇到不同的障碍物时其标准的行走运动如何改变。

10.5　MoCap 中的插值

MoCap 中有很多地方要使用插值。它可以用作数据简化技术——通过构造曲线把数据连接起来，从而把运动序列转化成一个 B 样条控制点集合。在这种情况下，我们在一个单运动序列中进行插值。或者，通过在序列之间进行插值可以拓展运动库。我们已经在 10.4.1 节提到这个想法，但那里用的是简单的单参数线性混合方法。本节我们要深入研究这个想法。

10.5.1　B 样条表示法

运动序列的典型的 B 样条表示法参见 Sudarsky 及 House 的［SUDA98］。他们使用非均匀

B 样条和插值构成了标准的 B 样条表示法：

$$f_i(t) = \sum_{j=0}^{\text{no\_of\_CPs}} \mathbf{p}_j \mathrm{B}_j(t)$$

根据以下启示选择合适的节点向量：

- 在曲率大的地方增加节点；
- 在噪声区域避免节点；
- 使用多个节点表示数据的不连续处。

可用最小二乘法通过最小化下式来找出控制点集合 **p**：

$$\sum_{j=0}^{n} (M(t_j) - f(t_j))^2$$

其中 $n+1$ 是序列中的样本点数。

通过噪声消除策略将其增强：

$$\sum_{j=0}^{n} \frac{1}{w_j + 1} (M(t_j) - f(t_j))^2$$

其中 w_j 是权重，代表$f(t_j)$与局部平均值间的偏离。

解下面的线性方程组可以求出最小二乘法的解：

$$\mathbf{Ap} = \mathbf{f}$$

还有一个可以达到很好插值效果的可选方案见［LEE99］，它在一个多分辨率框架中使用均匀 B 样条曲线（见 10.7.2 节的在信号处理中的多分辨率表示法）。这样做的主要动机是用多分辨率表示法来找到运动目标重定或运动适应问题的解。（10.8 节会处理这方面的内容；本节我们只关注表示法。）

多分辨率表示法是一个由粗糙到细致（或细致到粗糙）的层次结构，在任何层次都可访问它来给出原始数据的近似值或满足某种精确度的曲线。也就是说，我们把运动曲线看成一系列不断提炼的运动，这些运动以粗糙的近似值 f_0 开始，以精确的序列 f_h 结束。f_0 是最粗糙的层次而 f_1 是下一个：

$$x(t) = f_0(t) + f_1(t) + , \ldots , + f_h(t)$$

其中：

$$f_1(t) = f_0(t) + d_1(t)$$
$$f_2(t) = f_1(t) + d_2(t)$$

或者，可以写成：

$$x(t) = (\ldots((f_0(t) + d_1(t)) + d_2(t)) + , \ldots , + d_h(t))$$

通过下式计算或选定任一层 k：

$$f_k(t) = (\ldots((f_0(t) + d_1(t)) + d_2(t)) + , \ldots , + d_k(t))$$

因此一个运动序列用 $h+1$ 层表示，在此情况下，在均匀节点序列上定义三次 B 样条函数。节点序列本身定义了一个层次结构，这使它们每层的密度加倍。如果节点向量 τ_k 上有 $n+3$ 个控制点，则 τ_{k+1} 上有 $2n+3$ 个。这就有效地控制了 k 层近似值 f_k 的精确程度。插值过程从用最小二乘法找 f_0 开始（与前面相同）。这将是一个粗糙（光滑）的近似。一般，在每个

数据点上都有一个偏离值：

$$D_j(t) = x(t) - f_j(t)$$

这个偏离值就作为 f_k 的插值函数，其他以此类推。

关于我们所关心的插值质量，它消除了单层 B 样条插值的主要缺陷。这使因为节点向量的选择是很严格的。如果节点间距过于粗糙，那么在 B 样条曲线和数据之间就会出现背离；而距离过大数据点之间又会出现振荡。

10.5.2　运动混合——动词和副词

如前所述，很多运动混合涉及到两个序列的混合操作。在这个方法的一般扩展中，Rose 等人［ROSE98］描述了一个方法，在该方法中，他们举了很多“动词”的例子，并通过插值从动词中生成一个连续的运动空间。考虑图 10-7（并参看图 10-8（或见彩图）），图中展示了由一个动词描述的运动（比如走）的一个自由度的全部数据组织。

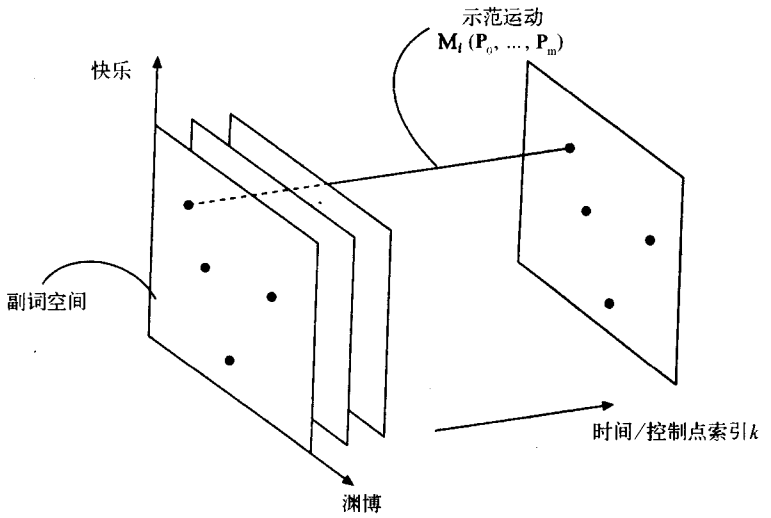


图 10-7　一个动词的一个自由度的数据组织

由此，运动（$\mathbf{p}_0$，…，$\mathbf{p}_m$）可表示为一个 B 样条控制点序列：

$$M(t) = \sum_{j=0}^{no\_of\_CPs} \mathbf{p}_j B_j(t)$$

每个平面中的控制点都指定了那个关键时刻的自由度值。这个平面是副词结构空间——例子中的两个轴分别是情绪轴和知识轴。换句话说，动词结构被副词结构参数化了。

系统的离线或组织阶段包含许多运动序列的手工检测，然后按照它们希望的角色特征把这些序列放到副词平面上的适当位置。这个方法的要求是，所有动词结构例子在结构上必须相似（例如，所有“走”运动的例子必须在同一个脚起步，包含的步数也相同）。另一个要求是，所有的例子一定要进行时间扭曲，使得相似的运动发生在同一时间。这使得例子的控制点 $\mathbf{p}_j$ 落在同一个副词平面。所以一个平面包含运动 M 中结构性相似部分的所有例子。

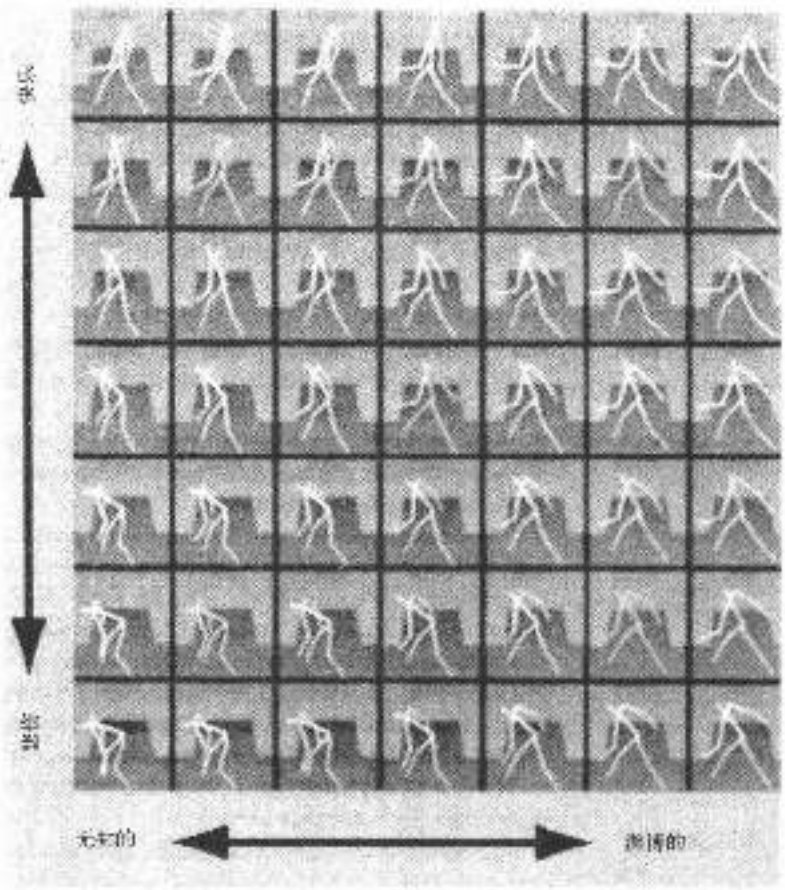


图 10-8　从动词实例产生一个连续的运动空间（绿色 = 实例）

一旦这些实例已经按照它们的运动质量（它们在副词空间上的 2D 位置）进行了分类，就可以用离散数据插值方法（见附录 8.1）在每个平面中插入实例 $\mathbf{p}_i$。选择这类插值方法是因为它们在处理多维稀疏数据时效果很好。这样，现在副词空间就定义了一个连续的控制点集合。

系统的目的是利用副词结构分类的实例进行实时的插值，及在游戏事件中调用特定的副词结构。其目标是使副词结构能根据游戏逻辑的要求连续地变化。

运动学的约束是在关键时间点用 IK（见第 11 章和 10.8.2 节）的方法施加的。从捕获的动词结构插值（其中假定末端效应器（end effector）的放置位置距离关键时间约束点很近）可以决定关键时间点姿势，然后再用快速的 IK 解法修改。

在随后的报告［SLOA01］中，这种方法被推广到形状插值中，详细情况参见第 8 章。现在，副词变成形容词，并用诸如男/女、老/幼等词来刻画形状运动的区别。

10.6　经典信号处理和 MoCap

了解一下傅里叶理论及由其产生的一些信号处理技术，对于全面理解 MoCap 处理是非常有用的。之所以这样说有两个基本原因。第一是这样使我们能够正确处理采样数据。其基

础是 Claude Shannon 在 1947 年提出的采样定理。这个雅致甚至有点预言性的定理是所有信号处理技术的基础——不管是何种应用。在所有诸如此类的信息信号（语音、音乐、视觉、工业控制信号等）都采用数字化采样和处理的年代，Shannon 定理精确地告诉我们，对于一个信息信号，要想不招致信息损失，最小的采样率应当是多少。这就表明，未经处理的数据（原始运动）在采集时就已经是“正确的”采样，但是，正如我们已经看到的，很多简单的 MoCap 处理还要对数据重新采样。例如，如果接受正确的采样约束，我们就能够实施时间扭曲操作而不至于引发什么问题。

运用傅里叶理论的第二个动机是，它提供了把 MoCap 函数分解为频率成分的方法。分别操作这些成分不仅使我们能够保持运动（比方说行走）的全局基本特征，还让我们可以通过剥落高频部分往运动中加入“突然”或“敏捷”等成分。（这就是在音频系统中我们用平衡滤波控制或音调控制时所做的工作——通过夸大或缩小不同波段的影响改变音乐的特性。）

显然，第一个动机很重要，但是对 MoCap 数据应用信号处理技术的用处和正确性还是有争议的，这个问题在 10.7 节再讨论。然而，已经有相当多的工作在使用信号处理技术了，并且确实取得了成效。

10.6.1　傅里叶理论

在科学和工程领域，所有线性变换的目的都是使原本复杂难解的问题变得简单易解。随着傅里叶变换的引入，很多功能强大的操作成为了可能，于是我们说数据转换到了傅里叶域上。以傅里叶变换作为看家工具的领域很多，而且自从 MoCap 技术一出现，傅里叶变换就被用来操作 MoCap 数据了。

傅里叶理论通常用来分析连续周期函数。周期函数是一个简单函数，它每隔一定时间不断地重复自身。这样的函数可以用傅里叶级数展开为谐函数。许多 MoCap 数据（比如，跑和走的循环）都是近似的周期函数。图 10-9 显示了股关节在跑循环中 3 个自由度的数据曲线。

近周期是很明显的。同样明显的还有偏离周期的小幅变化，这是极其重要的，因为它们使人的运动各具特色。完美的周期 MoCap 数据给人一种像机器人的感觉。关于近周期的概念见 Perlin 的著作（[PERL95] 及 9.2.1 节），其中把一个富于表情的运动模拟成一个基本运动加上噪声。由于噪声函数引起的扰动导致了人样的肢体运动。

当然，近周期数据在循环的 MoCap 序列中是实用的，但是我们先要看看更简单的情况——完美周期数据。这样我们就能想象变换的特点了——变换对于原始数据到底做了什么。周期数据是用傅里叶级数展开的，也就是：

$$f(t) = a_0 + \sum_{i=1}^{n} a_i \sin(i^* 2\pi st + \phi_i) \tag{10-1}$$

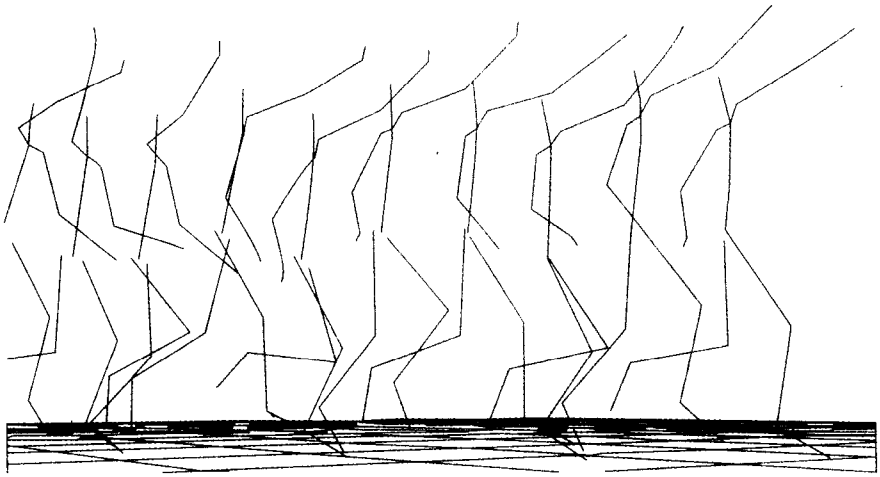
用语言来表达，就是任何周期信号都能分解成一系列带有振幅 a_i 和相位 ϕ_i 的正弦曲线 i。

正弦波（见图 10-10）：

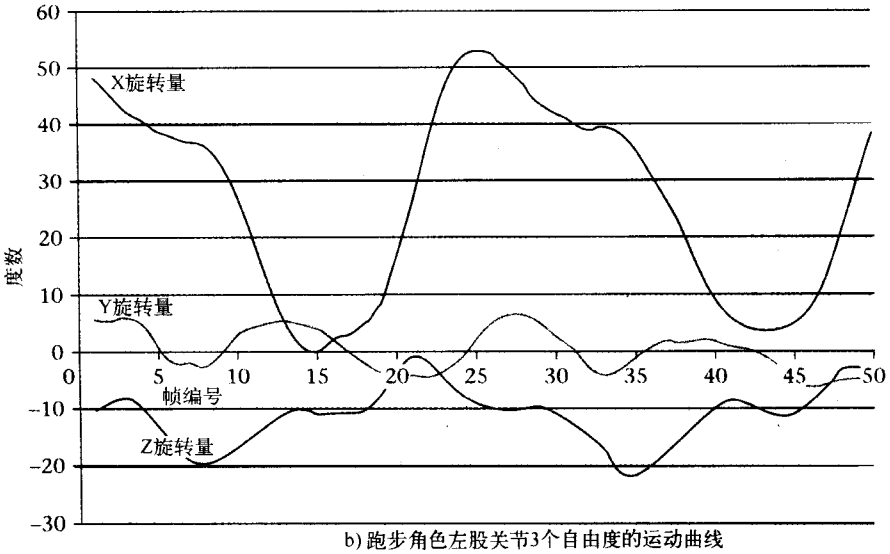
$$f(t) = a \sin(2\pi st + \phi)$$

由 3 个参数确定：振幅 a，频率 s，相位 ϕ。

在式 10-1 中，正弦曲线 $i = 1$ 与函数 $f(t)$ 有相同的周期或者说频率，它是基波。二次谐波，曲线 $i = 2$，其频率是基波的两倍。ϕ_i 指定了谐波的相位——它在时间轴的水平移动。或

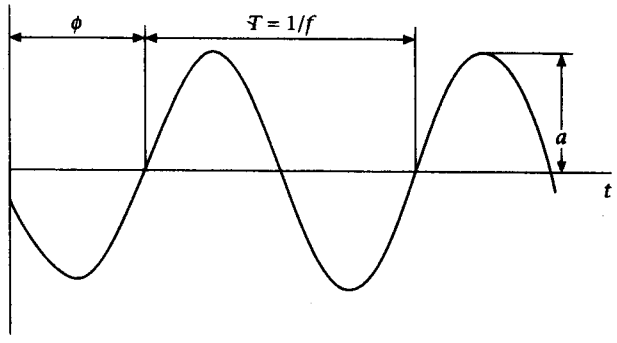


a)一个角色跑步及摆动运动的50帧——每5帧一步



b)跑步角色左股关节3个自由度的运动曲线

图　10-9



$f(t) = a\sin(2\pi st + \phi)$

图 10-10　一个简单的波形由 3 个参数确定：振幅 a、频率 f 和相位 ϕ

者说，通过把一簇正弦曲线（其傅里叶形式）加起来我们能合成任意的周期波形。这些正弦曲线都具有不同的振幅和相位。

通过一个标准的例子，我们可以对这种级数或分解的意义有直观的认识。考查图 10-11，它描述的是一个完美的周期信号——方波信号——以及展开到 $i = 1$、3 和 5 的傅里叶级数（在该例子中只有奇次谐波）。从图中可以看出：

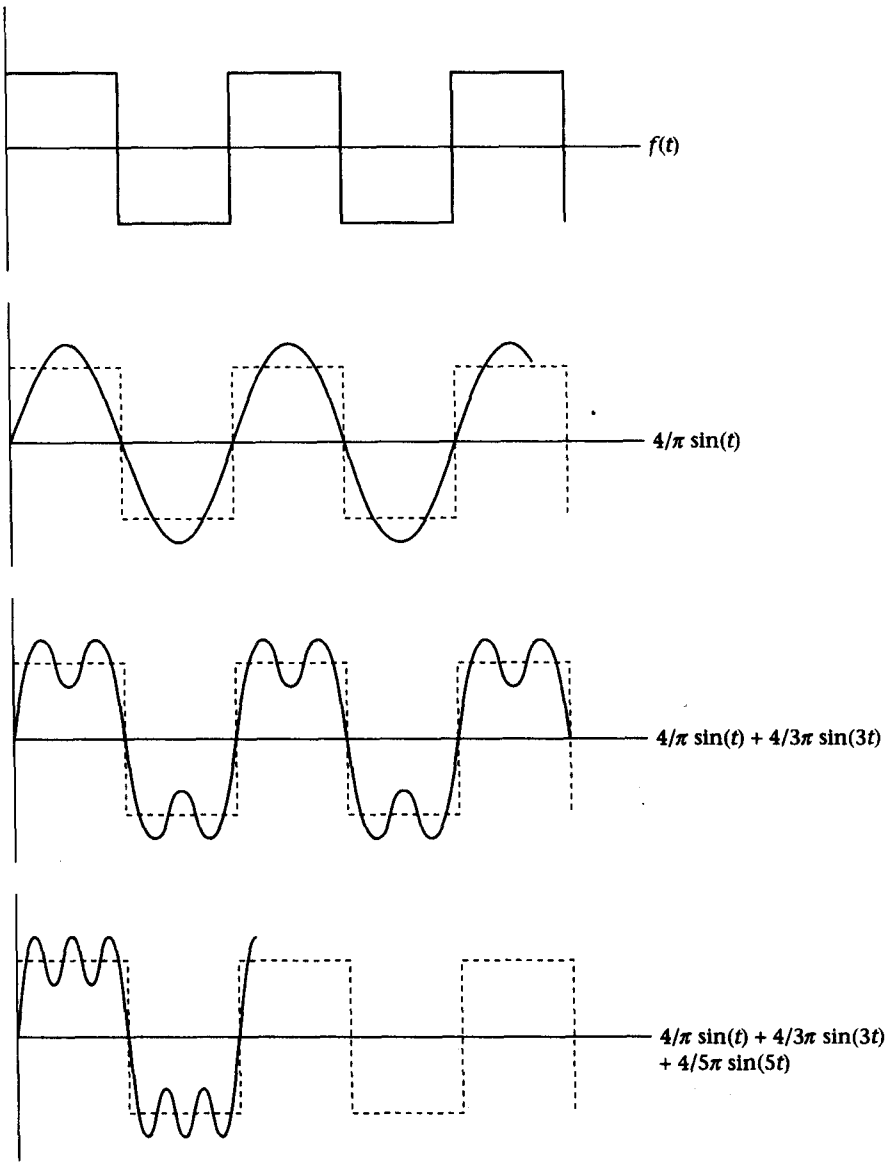


图 10-11　周期方波傅里叶展开的前 3 项

- 基波的频率与$f(t)$的相同。
- 当我们增加 i 时，和式包含越来越多的项，级数就更加逼近$f(t)$。波的边沿变得越

来越垂直，而且，每个波的顶部摆动也越来越小。

- a_0——作为直流成分或平均偏移——是0，这是因为该例中$f(t)$的平均值是0。

在图中还有一个不太明显的事实，就是该级数包含的项数是无限的，这是方波的边沿垂直导致的结果。实际上，没有一个物理系统能不需任何时间就改变其状态，所以不存在能包含无限频率成分的数据。[⊖]对于方波，谐波的振幅是：

$a_i = 4/\pi i$　　i 是奇数

$= 0$　　i 是偶数

即，随着频率的增加，奇次谐波的振幅迅速减小。在物理系统中，这意味着信号中的大部分能量集中在低频部分。高频部分的能量较少且含有细节成分（快速的改变）。

要得到频率成分的相对值 a_i，可以如下操作。首先把级数写成正弦曲线和余弦曲线之和的形式：

$$a_0 + \sum_{i=1}^{n} a_i \sin(i^* 2\pi st + \phi_i) = a_0 + \sum_{i=1}^{n} a_i \sin\, i^* 2\pi st + b_i \cos\, i^* 2\pi st$$

则：

$$a_0 = \frac{1}{T} \int_{-\frac{1}{2}T}^{\frac{1}{2}T} f(t)\, dt$$

$$a_n = \frac{2}{T} \int_{-\frac{1}{2}T}^{\frac{1}{2}T} f(t) \sin 2\pi st dt$$

$$b_n = \frac{2}{T} \int_{-\frac{1}{2}T}^{\frac{1}{2}T} f(t) \cos 2\pi st dt$$

这样，我们能把结果级数表达为所谓的线状频谱（见图10-12），其中线段的高度表示不同频率成分的相对振幅，而它们与原点间的距离表示频率。这是频率域表示形式。

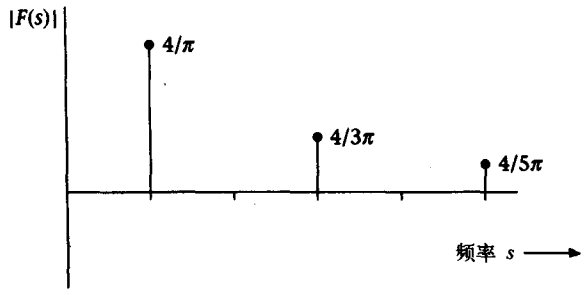


图10-12　周期方波的线状频谱

⊖ 在计算机图形学中，出现渲染的走样问题是因为对数据的频率成分没有限制（是通过逐像素计算数值来采样的）。在这种情况下，数据是一个数学模型或是数学化的定义。这就是为什么纹理映射中我们在超采样及均匀采样时并不是去除走样痕迹，而是简单地把它们的影响移到更高的频率上。然而，MoCap数据是从物理系统上采集的实际数据，它不包含无限高的频率成分。因此我们说实际数据波段总是有限的；在真实系统中，频率界限总有上界和下界。

现在再看 MoCap，如前所述，它在很多情况下都是近似周期的，这一点很重要。在 MoCap 中有高频和低频成分。一个周期运动（如行走）包括一个频率成分的范围，这在一定程度上反映了行走的本质或特色。一套轻快的行军运动包含比醉态的蹒跚有更高的频率。（部队行进的周期比醉态蹒跚更短，也反映了这种情况。）

通常，高频运动都是瞬时性的。我们来考查一套行走运动，行走者突然冲向天空。瞬时频率成分（比稳定步态频率高）会突然发生并消失。冲的运动只发生在冲的瞬间而已，但是在时间窗口中我们可以区分数据中的高频和低频成分。

再看图 10-12，它显示了一个简单的频率频谱。如果我们把数据转换成频率频谱数据，再单独对每个频率成分进行操作，然后把数据再转换回时间域，就得到了傅里叶域上的经典滤波操作。这使得我们能单独处理不同的频率成分，所以能够用某种方式修改信号的“特性”。我们有：

- $f(t) \rightarrow F(s)$求正弦级数
- $F(f) \rightarrow F'(s)$处理正弦级数（剥离它们）
- $f'(t) \leftarrow F'(s)$已滤波正弦求和得到$f'(t)$

这就是我们要研究的模型。

10.6.2　傅里叶理论和非周期数据

如前所述，实际数据至少在两个重要方面与方波的例子是很不同的。现在必须考虑离散（采样）数据——或者是近周期的或者是非周期的。这两个问题我们会单独考虑，现在先看非周期数据。所有运载信息的信号都是非周期的，MoCap 也不例外。这类函数使用傅里叶积分变换对进行变换：

$$F(s) = \int_{\infty}^{\infty} f(t)e^{-i2\pi st}dt \quad f(t) = \int_{\infty}^{\infty} F(s)e^{i2\pi st}ds$$

其中 i 是一个复数算子（而不是傅里叶级数定义中的正弦下标）。

这意味着傅里叶变换是一个复数函数。因为最简单的处理操作也要在$F(s)$量级上操作，我们暂时忽略这一方面。感兴趣的读者可以参考［BRACE65］。

第一个变换（向前变换）带我们进入一个傅里叶域，而第二个带我们从傅里叶域回到时间域。图 10-14b 显示了一个连续函数的窗口的傅里叶变换和一个周期函数的比较。我们要注意以下几点：

1）频谱是连续的。

2）频谱展示了最大和最小的截止频率。我们说信号的频带有一定的限制。所有实际的系统都是限制频带的，截止频率是系统本身的一个函数。

3）频谱有一个负的镜像。这是数学定义的结果，我们后面将忽略它。

傅里叶域中最普通的操作是（乘法的）过滤。这意味着度量每一个频率成分。我们定义一个过滤函数$H(s)$并且有：

$$f(t) \rightarrow F(s)$$
$$F'(s) = F(s)H(s)$$
$$f'(t) \leftarrow F'(s)$$

最普通的滤波器是低通、高通以及带通。这些是理想的滤波器，它们可以让某些频率的

光通过而阻止其他频率的光——我们选择数据中一定频率的成分，去除不要的部分。与原始数据比较，从过滤的数据中得到的运动的其他特征可能已经改变了。

借用一个图例，我们可以很容易地理解这一过程。图像操作很直观，因为图像不像 MoCap 数据，而是立即可以识别的结构。图 10-13（彩页中也有）展示了两个基本的过滤操作——低通的和高通的。

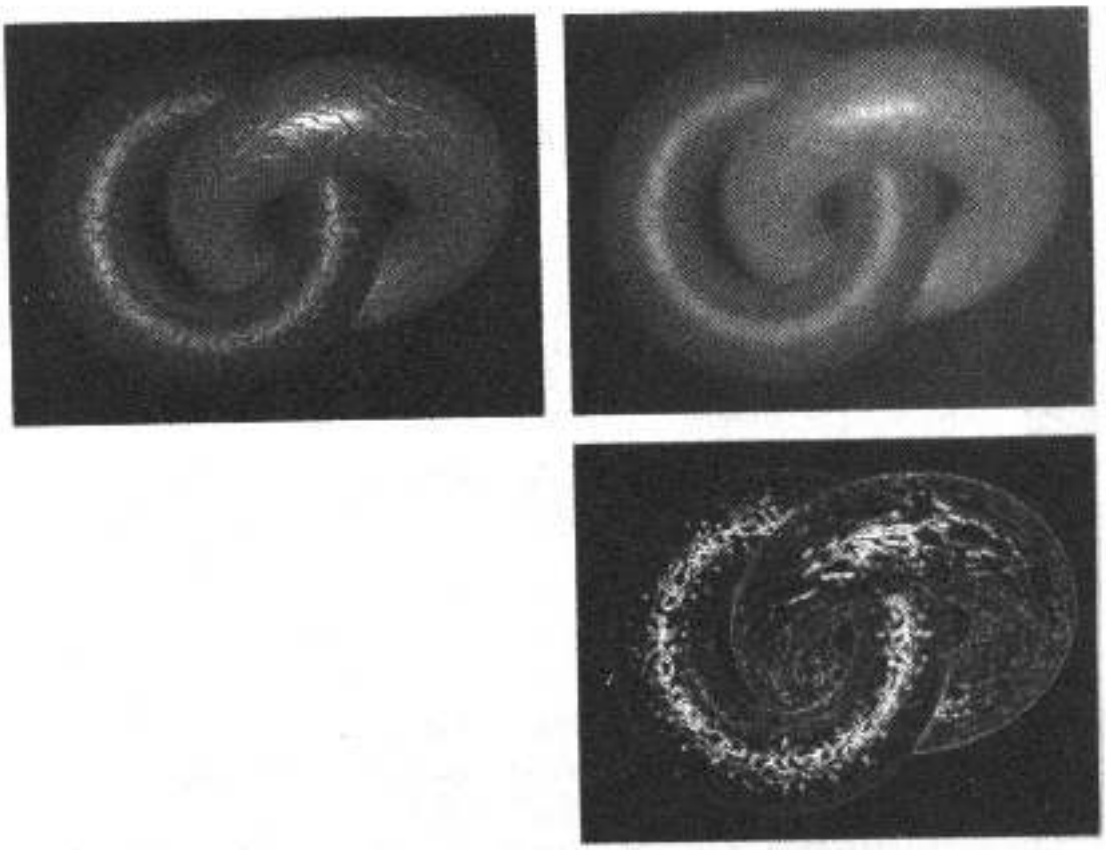


图 10-13　两个经过过滤操作处理的经典图像：低通过滤的结果模糊不清，因为它除去了高频部分；高通过滤通过移除变化慢的变量（低频率成分）来强调细节

低通滤波器会将图像弄得模糊不清，因为它去除了强烈的改变，就是那些图像变换最厉害的边。相反，一个高通滤波器能去除低频部分——图像中变换平缓的部分。这将强调边缘部分并使得图像变换缓慢的部分更加统一。

我们最后来看看这些操作如何影响 MoCap 数据；现在我们必须研究另外一个复杂的因素——MoCap 数据是离散的。

10.6.3　傅里叶理论和采样数据

任何计算傅里叶变换的计算机算法都要处理离散或者采样的数据，对于这些数据，一个离散傅里叶变换（DFT）对如下所示：

$$F(s) = \frac{1}{N}\sum_{t=0}^{N-1} f(t)\exp\left(-\mathrm{i}2\pi\frac{st}{N}\right)$$

$$f(t) = \sum_{f=0}^{N-1} F(s)\exp\left(\mathrm{i}2\pi\frac{st}{N}\right)$$

其中 N 是采样的个数。(注意，这些表达式很少用于程序中，取而代之的是使用快速傅里叶变换（FFT）来计算 DFT。)

我们先将这种变换看成是傅里叶积分变换的离散形式，然后再去讨论它所产生的困难。在图 10-14 中，我们演示了周期函数的线性频谱、非周期函数的连续频谱以及 DFT 之间的区别。图 10-14c 与图 10-14b 基本相同，除了它的$f(t)$和 $F(s)$是由线组成。在$f(t)$中这些代表数据采样。DFT 是一个有 N 个惟一频率成分的频谱，它们在理想状态下应该与$f(t)$的连续频谱采样版本等价。$F(0)$就是像以前所说的 DC（直流）或者平均项。

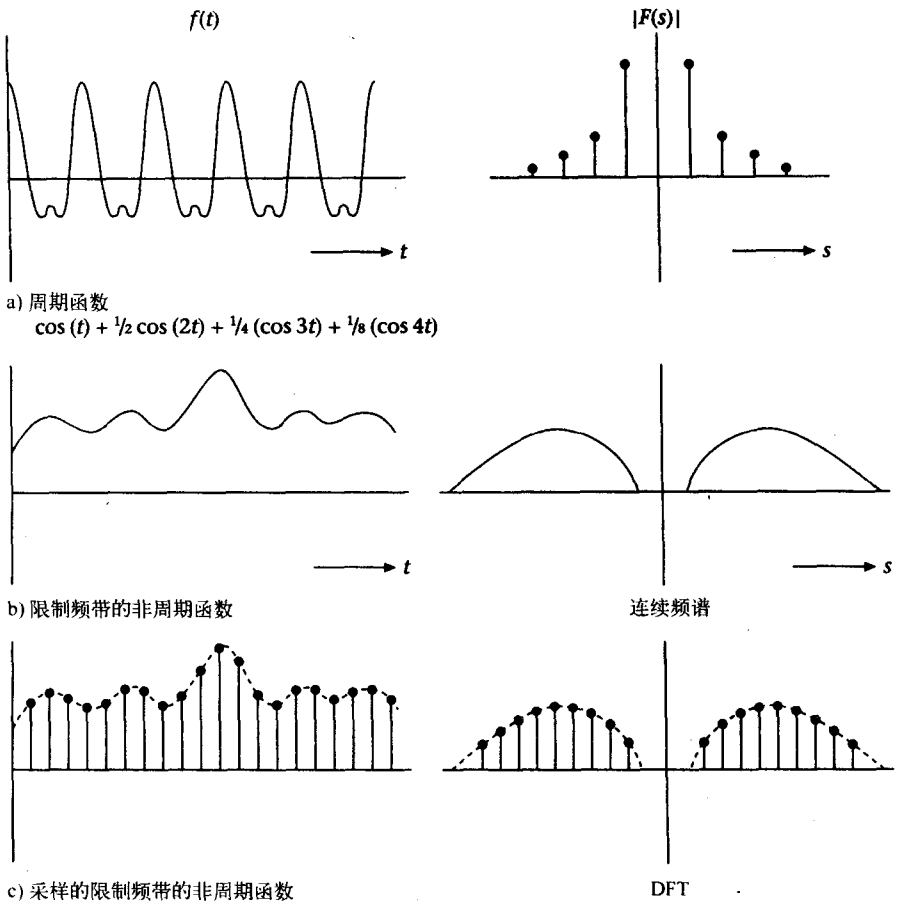


图 10-14　周期函数的频谱、非周期函数的频谱以及 DFT 之间的区别

关于 DFT 有两个重要的实际问题。第一，根据定义，输入数据具有有限的长度。在应用中这可能是一个完整的 MoCap 序列或者是一个很长的序列中的一个采样窗口。在任何一种情况下，我们必须在序列的任何一端使数据达到要求。如果让数据突然地开始和停止就会

人为地引入高频的成分。使用正弦波可以很容易地阐明这种称为“渗漏”（leakage）的现象。图 10-15 使用一个矩形窗口演示了这一效果——对信号的一个瞬时切割。当窗口的间隔是正弦波周期的倍数时，信号在窗口边界是 0，这样没有问题。如果窗口宽度不等于周期的倍数就会出现渗漏。

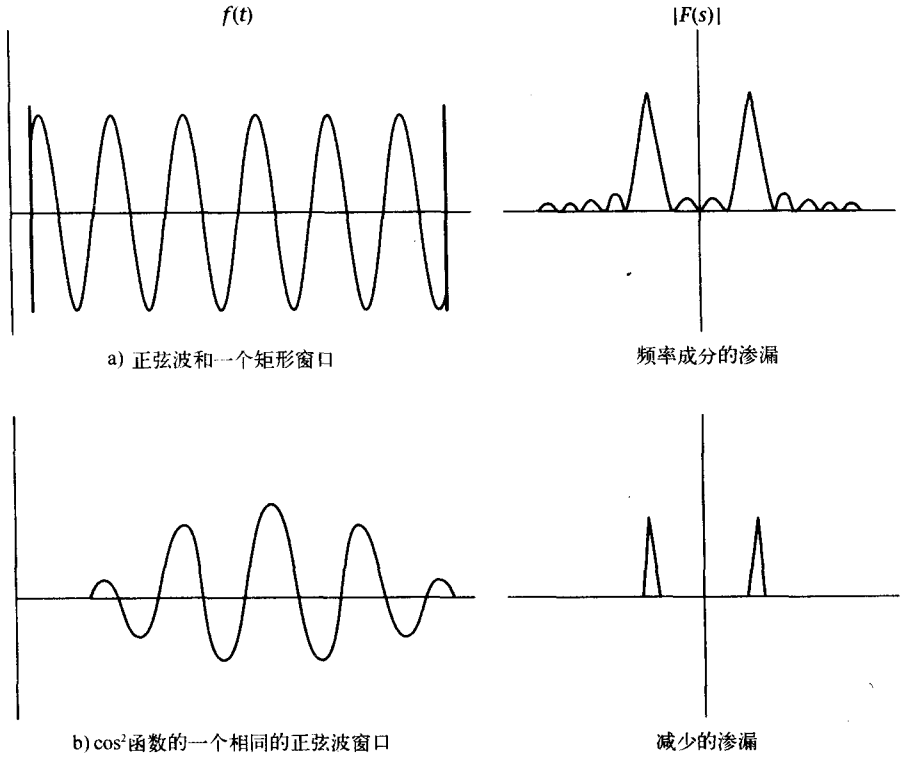


图 10-15　DFT 和渗漏

在 MoCap 数据中，在两种情况下要考虑渗漏问题。第一种是，如果我们想改变一个有限长度 MoCap 数据序列的特征，那么需要保证不发生渗漏。另外一种是需要循环或者串连操作的情况。这个表面上简单的过程包括了一个具有周期性活动（例如走路或者跑步）的窗口，用它产生一个具有窗口长度倍数的游戏运动。这里我们只是简单地按照需要的频率连接这些序列。然而，如果这个序列的开头和结尾不协调，将在连接点引入不存在于原始数据中的频率成分。

为了改善渗漏带来的影响，我们使用一种非矩形的窗口或者一个权重函数。一个流行的选择是 Hanning 或者余弦权重函数：

$$f'(t) = f(t)w(t)$$
$$w(t) = \cos^2(\pi t / T_0) \quad |t| \leqslant T_0/2$$
$$w(t) = 0 \quad |t| > T_0/2$$

其中 $T_0 = NT$（采样数 × 时间窗口）。

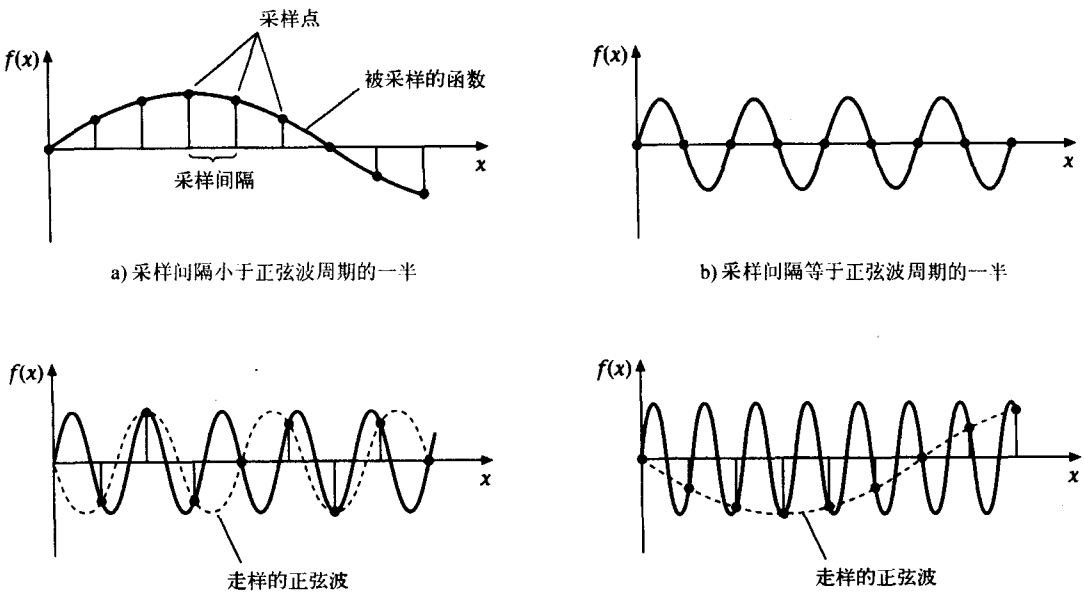
另外一个在使用 MoCap 数据时需要重点考虑的问题是采样，因为它关系到时间扭曲。

10.6.4　采样和走样现象

采样数据的问题是，如果不能在一个足够高的频率上进行采样就会产生走样现象。如 10.4.1 节所述，如果我们想去对原始数据再采样，MoCap 数据就会产生走样问题。同样的问题也出现在电脑游戏中，因为我们常常需要不平坦的空间图像间隔的采样。走样会造成很大的数据损失。这个问题的量值在 Shannon 的采样定理中给出，我们可以非正式地表述如下：

采样频率必须至少是信号中最高频率的两倍。

走样很容易阐述，如图 10-16 所示。这里我们考虑一个正弦波信号（当然，这个正弦波不包含任何信号，我们只是用它的规则性来阐述一个论点）。在图 10-16a 中，足够的信号被采样——采样频率大于正弦频率的两倍。（就是说，采样间隔比正弦波周期的一半小）。在图 10-16b 中，采样频率正好是波的频率的一半。信号消失了。另外两个例子显示了当采样频率低于波频率的一半时，采样的振幅看起来就像是从低频正弦波上采样一样——发生了走样现象。



a) 采样间隔小于正弦波周期的一半　　b) 采样间隔等于正弦波周期的一半

c) 采样间隔大于正弦波周期的一半　　d) 采样间隔远大于正弦波周期的一半

图 10-16　空间域中正弦波的采样表示

正弦波的欠采样和从一个采样中重建连续信号（图中的虚线）产生了原始信号的“别名”——另外一个比原来采样信号低频的正弦波。我们可以说这种情况的发生是因为采样的一致性和规律性干涉了信号的规律性。为了避免走样，必须在一个适当高于信号的频率上采样。

现在我们来推广图 10-16 中的例子，考虑一个频率域，这个域中$f(t)$包含的信号不是一个纯的正弦波。$f(t)$中的 t 是任何一个常规变量，而$f(t)$可以代表一个 MoCap 信号。$f(t)$的

频谱将显示一些“包迹”（envelope）（图 10-17a），它们的最高频率限制是 s_{max}。采样函数的频谱是一系列线（图 10-17b），它们理论上无限延伸并按照间隔 $s_{sampling}$ 分割开（采样频率）。空间域上的采样包括将 $f(t)$ 乘以一个采样函数。在频率域上的相同过程是卷积（我们将很快介绍卷积，现在暂且接受这个概念）。采样函数的频谱与 $f(t)$ 进行卷积产生图 10-17c 显示的频谱——$f(t)$ 采样版本的频谱。采样函数然后乘以一个重构滤波器来产生原始函数。这个过程在时间域上的一个典型例子是现代电话网络。其中最简单的形式是对语音波形采样、编码并通过通信信道传输数字化的信号。然后用重构滤波器解码采样信号为原始的信号。

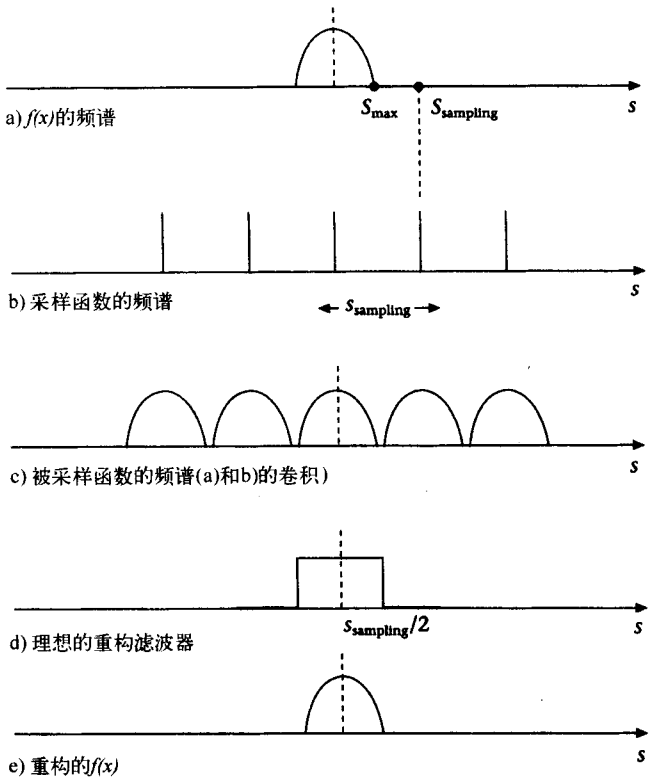


图 10-17　采样过程的频率域表示，其中 $s_{sampling} > 2s_{max}$

注意，频率域上的重构过程是空间域上的卷积。总而言之，空间域上的这个过程是将原始函数乘以一个采样函数，然后通过一个滤波器进行卷积。

在上述例子中有一个前提条件：

$$s_{sampling} > 2s_{max}$$

在第二个例子中（见图 10-18）我们同样展示乘法和卷积这两个过程，但是条件改为：

$$s_{sampling} < 2s_{max}$$

顺便提一下，$s_{sampling}/2$ 被称为 Nyquist 极限（Nyquist limit）。这里包迹代表了 $f(t)$ 中的信息。频谱好像是沿着 Nyquist 极限折叠起来（图 10-18e）。这个折叠是个损坏信息的过程：高频率的（图像细节）部分将遗失，低频率部分将产生走样。

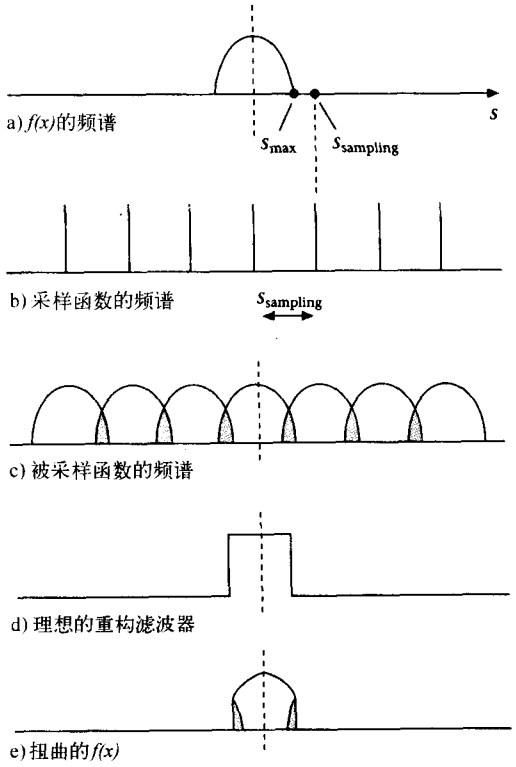


图 10-18　采样过程的频率域表示，其中 $s_{sampling} < 2s_{max}$

10.6.5　反走样滤波器

如果想采样一个信号，我们必须遵守采样定理。在很多实际应用中有一个固定的采样频率，我们简单地通过保证信号中没有包含高于 $2s_{sampling}$ 的频率成分来防止走样的产生。这是通过信号反走样滤波器来实现的。这个滤波器是低通过滤装置，截止频率为 $2s_{sampling}$。

像在 10.4.1 节中讨论的，在加速或者减速运动中也必须考虑这个问题。如果我们通过跳过采样来加速运动，那么可以有效地降低信号采样频率。当这个操作到达 Nyquist 极限时，运动会因为走样的发生而减速而不是加速。

10.6.6　时间域中的过滤——卷积

现在我们来考虑如何在时间域中进行一个单一的过滤操作。这等于将信号变换到频率域上，过滤并将得到的结果变换回时间域。一般地，我们希望从图像的一个效率点（efficiency point）开始做起。10.6.1 节给出了一个应用例子。

回忆前面所提到的知识，我们知道频率域中的过滤是一个缩放或乘法操作。我们曾间接提到了卷积，现在就来从细节上讨论这个操作。卷积定理是：

$$f(t) \otimes h(t) \Leftrightarrow F(s)H(s)$$

这就是说为了获得在频率域上 $F(s)$ 乘以 $H(s)$ 的结果，我们必须将 $f(t)$ 和 $h(t)$ 做卷积，

其中$h(t)$是 $H(s)$的傅里叶变换。

离散卷积可以定义为：

$$f'(t) = \sum_{i=-w}^{w} h(i)f(t+i)$$

图 10-19 中显示了一个形象化描述此操作的方法。一个所谓的“核心”沿着信号移动，我们认为它是固定的，将（$2w+1$）个数据样本乘以过滤器的权重$h(i)$。在每一个位置将它们加起来，结果用来替换滤波器当前位于其中的 MoCap 数据样本。

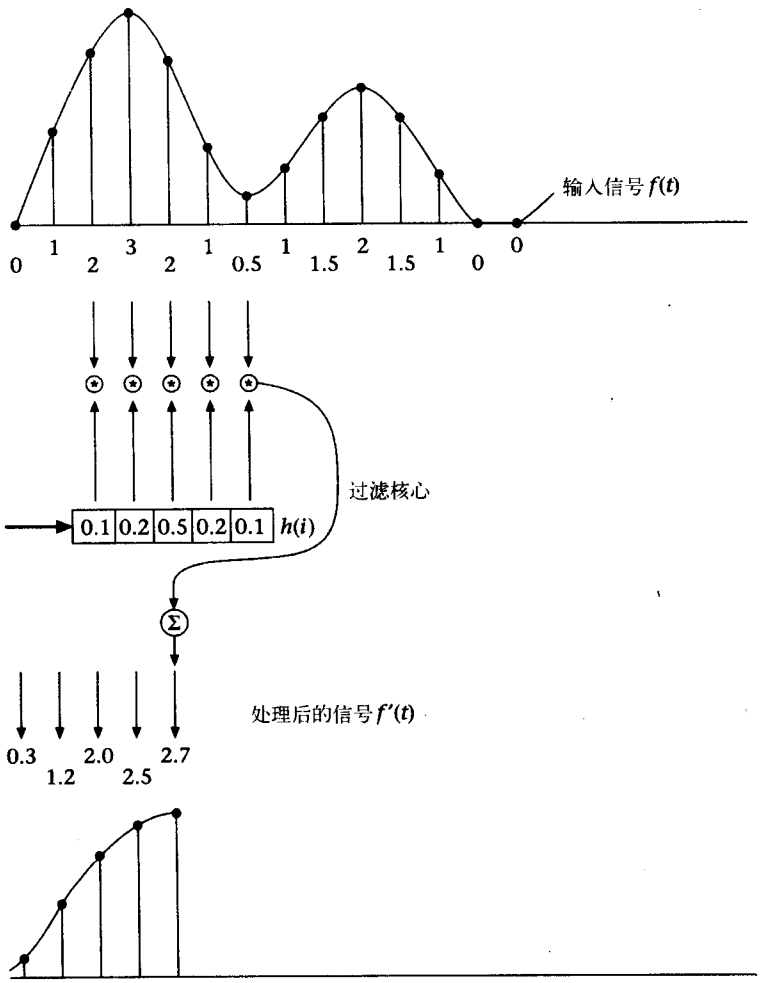


图 10-19　离散数字的卷积

10.7　信号处理和 MoCap 数据

就像在 10.6 节开头所提到的，虽然对于 MoCap 数据的信号处理是一个流行的技术，但是它的基础不是完全正确的。问题就是我们所知道的欧拉三角表示。传统的过滤技术假设运动数据$f(t)$是线性的。欧拉三角是非欧几里得的——向任何方向运动只会使你回到出发点。

然而，对于很细微的运动这种表示可能是近似线性的。在一篇最近的报道中 Lee 等人谈到了这个问题，并通过使用指数对数图作为角度位移表示将传统的过滤技术应用于变形过的方向数据上。

其他的使得傅里叶变换在 MoCap 处理中适用的潜在假设是，傅里叶参数有物理的意义或者与数据相关。这是下一节中介绍的内容的基础。在那里作者将傅里叶参数与运动中的“感情”连续起来。

在傅里叶域中我们使用了振幅、频率和相位这些成分。先考虑振幅值上的操作。让所有的成分使用一个统一的比例因子将会改变在循环运动（例如行走）中的步长。MoCap 数据是一个连接角的摆动，如果我们等量地增加所有的频率成分，将会增加摆动的振幅。一般的过滤模型通过增加一些振幅减少其他部分的振幅来改变频率成分的振幅。这样，运动的特征就改变了。一个快速的运动应该在高频成分显示更多的能量。为了使得运动更快我们应该增加这些成分。Bruderlin 和 Williams 描述频率过滤的作用如下（10.7.2 节中描述了这个方法）：

> 增加行走序列的中间频率将产生流畅但是看起来夸张的步态。相比之下增加高频部分将使运动出现急颤，而增加运动的低频部分将会产生一种关节运动减少的减弱的行走运动。

我们所必须考虑的频段的数量很低并依赖于 MoCap 设备的采样率。Unuma 等人[UNUM95]认为一个好的结果可以通过只操纵三个频率成分（最多 7 个）来得到。

基本原理的频率可以被改变从而更改了循环运动的速度。另外一个对于频率变量的操纵是“规范化”。我们会在下一节使用它。

现在来看两个使用经典信号处理技术操作 MoCap 数据的实例。

10.7.1　傅里叶域中的插值/外推法

在研究所谓“基于情感的人样动画的傅里叶原理”中，Unuma 等人[UNUM95]使用傅里叶域插值/外推从一个已有的运动序列得到新的或修改过的序列。所谓“情感”是指一个基本周期运动（如行走循环）上的次运动层次。这种设想使作者能萃取悲伤的和自然的步态之间的区别，并把这种区别应用到跑动中。

Unuma 等人定义了重新调整过的傅里叶功能模型：

$$f(t) = a_0 + \sum_{i=1}^{n} a_i \sin(it + \phi_i)$$

这是式 10-1 经频率或周期规范化得到的。它对齐了序列（周期不同）中的频谱振幅成分并且使得当插值或混合两个频谱时能得到一个新的运动。这个操作的效果应该与 10.4.3 节介绍的对齐算法相当。因为这种规范化操作，该研究局限于循环运动（或近周期运动）。

因此给定两个频谱 a_i 和 b_i 就可以形成一个插值函数：

$$f_{ab} = ((1-s)a_0 + \sum_{i=1}^{n}((1-s)a_n + sb_n)\sin(it + (1-s)\phi_{ai} + \phi_{bi}))$$

其中：

$0 \leqslant s \leqslant 1$ 使用插值；

$s > 1$ 或 $s < 1$ 使用夸大外推。

10.7.2　使用拉氏算子的多分辨率滤波

Bruderlin 和 Williams "BRUD95" 引入了一个对 MoCap 数据实施多分辨率滤波的有效方法。该方法通过时间域卷积构造信号的一个拉氏算子或通频带层次。通过看一个图例很容易理解这个概念。考虑两类图片金字塔结构，一个低通金字塔和一个通频带或拉氏金字塔。这两个例子见图 10-20。低通图由多幅图片组成，开始是最清晰的图片，后面是半清晰版本，再后面是四分之一清晰版本等。每一个版本都是由前一个版本用低通滤波产生。金字塔结构顶端是一个单像素图，是整个图片的平均值——DC 层次。[1]

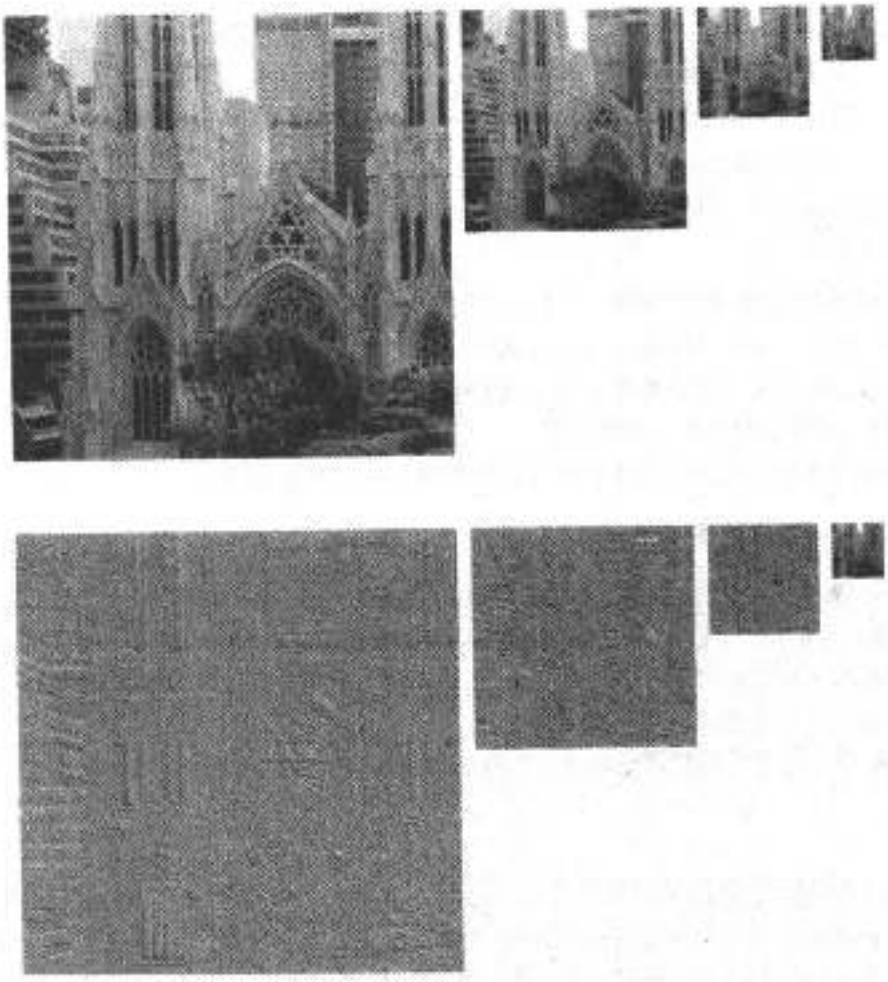


图 10-20　低通和通频带图片金字塔结构

通频带金字塔结构保存了低通金字塔中层次间的区别。例如，如果在最高分辨率的图片

[1] mip 贴图是这类图像分辨率最为普遍的例子，以理论图按这种方式存储，且当把纹理映射到物体上时，使用与屏幕大小或物体极影相关的距离选定一个处在合适分辨率等级的图片作贴图，距离用户很远的物体应当选择小或者粗糙的低分辨率图片作贴图。

中有 $m \times m$ 个像素，就能在低通金字塔中产生一个“滤波”版的$(m/2) \times (m/2)$的图片。然后把该滤波图片“张成” $m \times m$，并把它从原始图片中减掉，从而形成低通图片。

如果把低通金字塔标记成 G_0，G_1，…，G_n，其中 G_0 是原始图片，G_n 是单像素图片。把通频带金字塔标为 L_0，L_1，…，L_{n-1}，这样就有：

$$G_0 = L_0 \oplus (L_1 \oplus (L_2 \oplus \ldots \oplus (L_n \oplus (L_{n-1} \oplus G_n))$$

其中，$\oplus$指张量和。即从 G_n（单像素图）开始，把它张成 2×2 的图片并把它加进 L_{n-1}，于是得到 G_{n-1}。

Bruderlin 和 Williams 为处理 MoCap 数据构造的这种分层结构使其中的每个序列保持相同的长度，而不是把每层缩小一倍。这种分层结构可以展开成标准和式：

$$G_0 = G_n + \sum_{k=0}^{n-1} L_k$$

这是通过把滤波核心设置为 0 得到的。他们用到：

$w_1 = [c\ b\ a\ b\ c] = [0.625\quad 0.25\quad 0.375\quad 0.25\quad 0.625]$

$w_2 = [c\ 0\ b\ 0\ a\ 0\ b\ 0\ c]$

$w_3 = [c\ 0\ 0\ 0\ b\ 0\ 0\ 0\ a\ 0\ 0\ 0\ b\ 0\ 0\ 0\ c]$等等。

低通序列用下式计算：

$$G_{k+1} = w_{k+1} \otimes G_k \quad k = 1, 2, \ldots, n$$

其中$\otimes$表示卷积算子。

拉氏序列为：

$$L_k = G_k - G_{k+1}$$

根据多分辨率滤波要求，该序列有：

$$L_k = s_k L_k$$

待处理信号为：

$$G_0 = G_n + \sum_{k=0}^{n-1} L_k$$

因此，用 MoCap 数据扩展 n 倍的代价，L_k 可以预先计算出来。实时构造就仅有加权和求和了。

如果该方案用来混合两个序列，而不是改变一个简单序列的角色，那么在多分辨率混合中必须预先应用时间扭曲。

10.8　运动编辑：基于约束的方案

现在看一看基于约束的运动编辑方案。这与之前介绍的技术不同，它们源自一个新的满足（大量帧或序列的某一帧内）约束的运动。诸如混合或滤波等运动变换并不明确地包含约束——它们简单地操作运动数据，并不考虑运动的几何后果。

大多数基于约束的方法使用运动学约束，最普遍的是末端效应器（end effector）的期望位置。除了高昂的代价，没有其他原因能说明为什么不使用动力学约束，而且甚至连所谓的时空约束方法的发明人 Witkin 和 Kass［WITK88］都使用动力学约束。

基于约束的方法最普遍的应用是运动适应（motion fitting）。运动适应或目标重定是重要

而困难的问题，它有两个目的。第一个是将 MoCap 数据重用到与采集数据的真人身体比例不同的虚拟角色上。在行走运动中的一般例子是脚“浮”在地面上并且普遍存在“滑行”问题。看图 10-21，它描述的是不同大小的角色用相同的名为“开灯”的 MoCap 数据驱动的情形。由图中可以发现，当角色大小变化时，关节角度必须用非线性的方式改变。有些需要改变，而有些则不需要。这种情况下，很明显时关节角度必须进行修改，但是其他部分可以保持不变。然而，如果运动正在进行从小角色到大角色的目标重定，可能会出现其他约束。大角色的身体运动可能必须进行改变，以防止他的鼻子与台灯撞在一块。

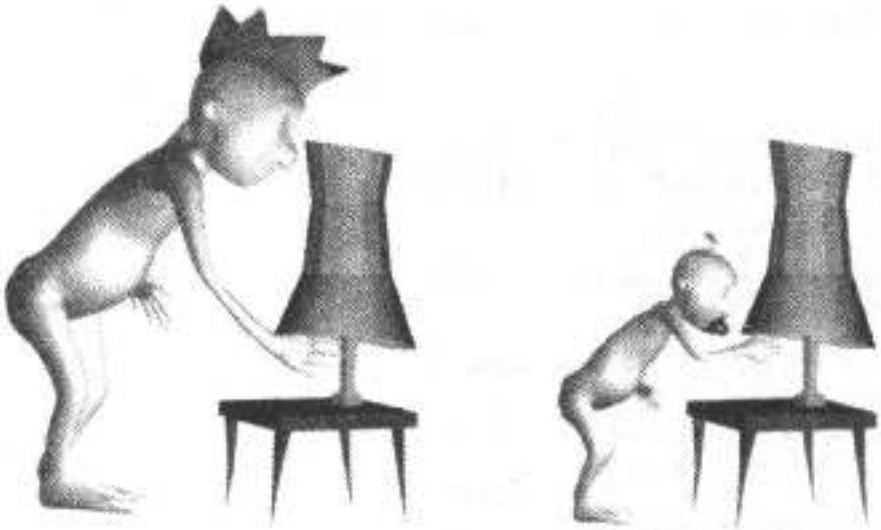


图 10-21　目标重定意味着在执行动作的大小不同的角色上使用相同的数据。注意图中肘关节角度的改变

另一个普遍的约束应用是使 MoCap 数据适应要求以产生游戏要求的姿势（该姿势未在原始序列中出现）。例如，如果要抓的物体的位置或大小发生了变化，我们可能要改变抓的运动。在图 10-22（彩页中也有）中描述了将一个抓物的 MoCap 序列应用到一个新的物体（与记录运动时的物体不同）。这种情况下手/物体的穿过状态是可见的。如果同样的 MoCap 序列用在相似的物体上，则用碰撞检测或者其他的设备来修改 MoCap 序列即可。否则，每个物体都需要使用一个新的 MoCap 序列。这个特殊问题是使用 IK 方法处理的，详见 10.8.3 节。

运动适应或目标重定经常被看作是约束满足问题。修改运动以使新约束得到满足的最好方法是什么？在大多数应用中通常使用反向运动（IK）来满足约束（见第 11 章）。

10.8.1　运动中的动力学约束

现在，我们简要地讨论动力学约束的用法。使用动力学约束的目的不仅仅在于它们在 MoCap 数据基于约束的运动变换中的潜在用途，更在于其在运动合成中的潜力。如果能用一种可以产生逼真运动的方式来有效地模拟人形角色的动力学特性，那就不需要 MoCap 技术了。本节的目的就是演示动力学约束的两种用法，并且用一个例子来显示为什么（任意复杂性的）角色的运动合成如此困难。

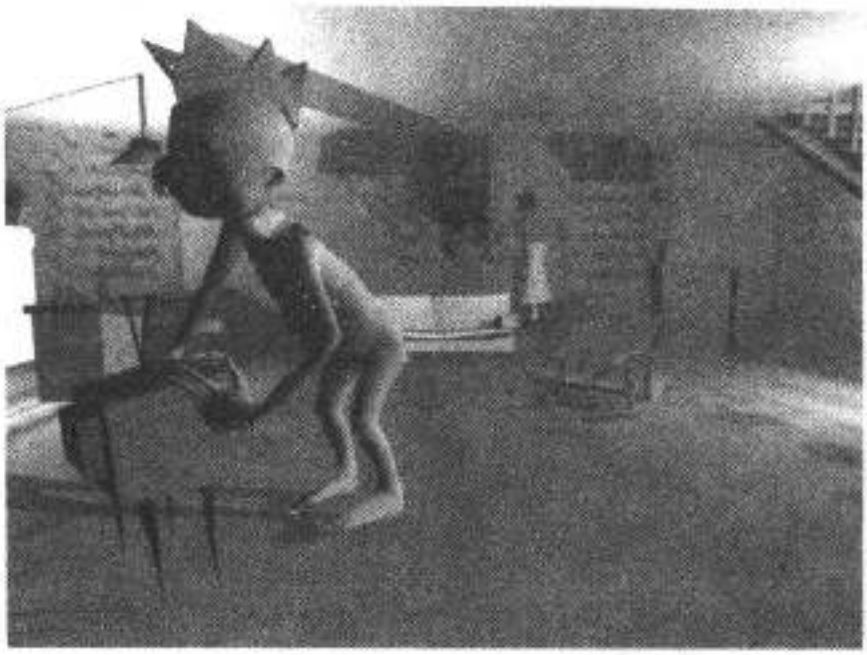
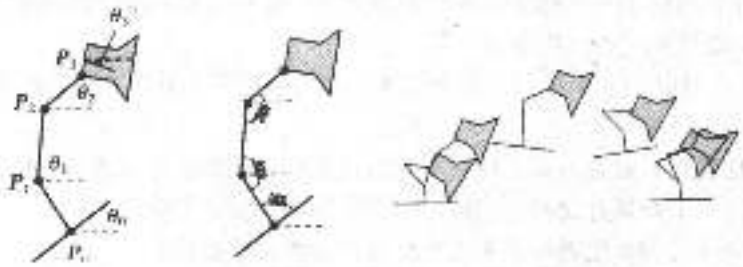


图 10-22　普通角色/物体交互的问题。如果用同样的动画脚本处理“抬”相似的大物体，就会出现角色/物体的穿透现象（在图像中可见）。另一个可选方案是，为每个物体写不同的脚本

Witkin 和 Kass［WITK88］研究了一个简单角色（Luxo Jr.[⊖]的平面模型，有四个关节角和平移参数，见图 10-23）的运动合成问题。这个项目的目标是为基于以类似“从 A 跳到 B”作为输入的角色合成运动。



a) 刚性铰链和无摩擦关节构成的平面收型　　b) “跳跃”运动的模拟

图 10-23　Witkin 和 Kass 的 Luxo Jr. 仿真（见［WITK88］中的叙述）

谈到该方案的成功，Witkin 和 Kass 叙述道：

做一个 Luxo 台灯，仅告诉它起点和终点，它就能进行逼真的跳跃运动。结果显示这些性质如预想的一样：屈伸、挤压、伸展，而且时间特性确实从运动意图的

⊖ 《Luxo Jr.》是一个著名的动画短片，是 1987 年 John Lasseter 为 Pixar 出品的。此项工作的动机（见［LASS87］）是把传统动画原理（挤压和伸展、预备、跟随等）结合到三维电脑动画中。

简单描述和它所在的物理环境中显现出来。

通过引入一个最小目标函数，再加上包含初始和终结姿势及位置的约束，就得到一个解法。其算法的框架是：找出使目标函数满足约束并且使之最小的关节角度曲线构成的集合。该系统中的力是“肌肉”力，地面和基座间的接触力是重力。肌肉是用三个角弹簧在关节处连接铰链来模拟的。弹簧弹力为：

$$F_i = k_i(\varphi_i - \rho_i)$$

其中 k_i 是弹性强度，φ_i 是关节角度，ρ_i 是静止角。

弹性强度和静止角都是允许变化的。弹力被用于构造目标函数，这个函数用于通过保证每一个时间拍肌肉消耗最小的能量来使运动的机械效率理想化。这里肌肉的能量由肌肉的力量和关节的角速度产生。这里的要点是：该系统解决角色运动和在所有不确定间隔中随时间变化的肌肉力量，而不是随着时间有序地前进。这个系统在工作时将时空约束作为输入接收，然后找一个使得肌肉消耗最少力量的方法。

这个例子既阐述了时空动态方法的潜力又说明了它的局限性。从本质上说它是一个简单的模型。这个问题有很大的复杂性，且代价的增长速度很快。

10.8.2　运动中的运动学约束

在前一节中我们假设想要的属性是由系统物理定义的动力学约束。就像我们讨论的，在MoCap 中进行约束的一般方法是运动学。当我们玩游戏时，几何上或者空间上的约束在其中被指定或者形成。运动必须与这些约束相适应。最一般的约束是关节限制和末端效应器位置（或者一般来说是角色上的任何点）。当约束被用于运动匹配时，我们假设原来定义的运动都满足所有的约束。如果改变运动的目标（re-targeting），那么可能就不再满足原来的约束。图 10-21 给出了一个简单的例子。在这种情况下，很明显必须保持末端效应器位置，但是为了满足约束我们得改变关节角。那幅图像显示了一般不可能在保持关节角的情况下满足末端效应器约束。这样就出现了一个问题：哪个更加重要？是末端效应器位置还是关节角？大部分方案都以末端效应器位置为先决条件。

Shin 等人在［SHIN01］中提到了这个问题，并基于以下启发对不同的约束分配了一个重要的度量（metric）：

1）根部的位置很可能是不重要的——它可以被移到新的位置来适应约束的要求。

2）当与另外一个物体有交互作用时，末端效应器的位置是重要的。

3）当与其交互的物体很近时，末端效应器的位置是重要的。

4）亲近必须受到限制，这取决于末端效应器是否移向物体（重要）或者离开那个物体（不重要）。

基于重要性的方法的想法是，重定目标的系统应该找到哪些约束是重要的而哪些不是。然后这个系统应该改变那些不重要的方面。例如，如果末端效应器移开了一个物体（与之交互的物体）那么它的重要值会降低，这样一来对应肢体的捕捉位置可以保持。

上面的启发在一个与接近物体的末端效应器相关的重要值中体现出来。距离函数定义如下：

$$D_{ij}(t) = \frac{d_{ij}(t) + d_{ij}(t + c\Delta t)}{2}$$

其中：

D_{ij}是当前距离和预测距离的平均值；

d_{ij}是末端效应器 i 和物体 j 之间的距离；

c 是一个正的常数。

这可以被近似为：

$$D_{ij}(t) = \frac{d_{ij}(t) + d_{ij}(t) + c\Delta t\dot{d}_{ij}(t)}{2}$$

$$= d_{ij}(t) + \lambda\dot{d}_{ij}(t)$$

其中$\dot{d}_{ij}(t)$是 $d_{ij}(t)$第一项的系数。它被规范化为：

$$\bar{D}_{ij} = \frac{D_{ij}}{D_{ij}^{max}}$$

其中 D_{ij}^{max}是末端效应器受到物体影响的最大距离。这个值大就意味着交互很强烈。这个值小就意味着末端效应器可以独立地离开这个物体，除非它们离得很近。

规范化距离的一个重要函数 p 定义如下：

$$p(\bar{D}_{ij}) = \begin{cases} 2\bar{D}_{ij}^3 - 3\bar{D}_{ij}^2 + 1 \text{ 如果 } \bar{D}_{ij} < 1 \\ \text{否则为 } 0 \end{cases}$$

这里的立方体（cubic）有以下属性：

$$p(1) = 0 \quad p(0) = 1 \quad p'(0) = 0 \quad p'(1) = 1$$

所以当末端效应器和物体之间的距离降到 0 时重要性值变成了 1，p 的变化率在每一个端点变小。

10.8.3　每帧重定位法

基于约束的方法明确强调，运动适应的目的就是要寻找满足约束的，并且保留原来运动特征的新运动。最简单的方法就是对每一帧进行操作，调整角色的姿势以满足该帧的约束。这种普通的方法最适用于实时任务。另一种可选的方法是通过考虑整个运动序列来找出解决方案，我们将在下一节中介绍。

现在我们考虑，如何通过简单的 IK 方法把一套走路运动重定位到一个新角色中。我们必须关注的问题是，如何防止足部滑动或者穿透到地面以下。我们可以选择仅当足部在地面以上时，打开 IK 方法；或者可以使用 IK 方法，为末端效应器的每一帧，通过整个序列中原始的足部运动，计算出肢体的角度。两种方法的惟一不同点在于代价：前一种方法代价更低，也是目前为止最常用的办法。

首先考虑对每一帧应用 IK 方法，图 10-24 演示了一个例子。

图 10-24 的第一幅图像表示原始的运动，可以跟图 10-9 比较。我们曾指出图 10-9 展示了一个近似完全的周期性。如果用来产生这幅图像的 MoCap 数据是完全周期性的话，那么每一个循环的幽灵序列将是完全相同的重复。在第二幅图像中，原始运动被应用到上下肢不同的新角色上。很明显在这幅图像中产生了足部滑动和穿透地面。在最后一幅图像中，IK 被应用到每一帧，使用了 11.3.1 节所介绍的微分方法。足部或者末端效应器在周期中的跨步阶段从地面提起，并紧跟在“足部离开地面”阶段的原始位置。使用了 MoCap 数据中平移

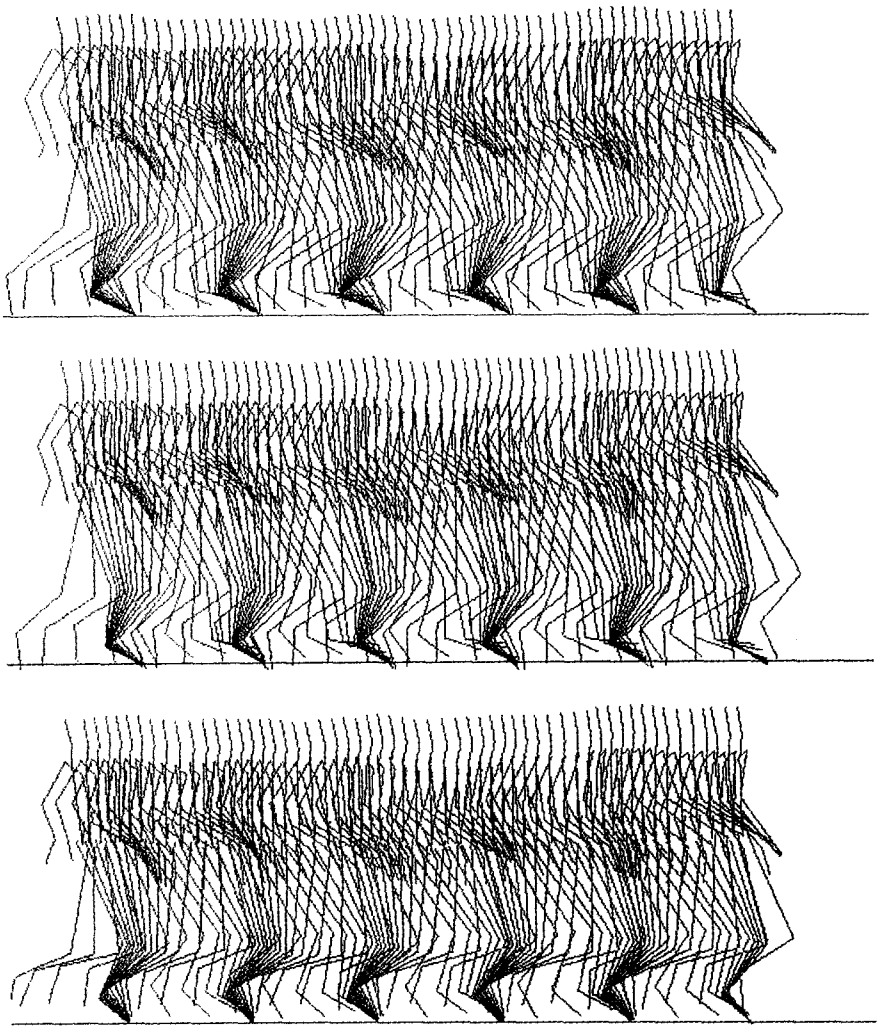


图 10-24　IK 应用于每一帧以重定位一个走路周期。第一幅图像为原始 MoCap 序列；第二幅为序列应用于一个比例改变的人物轮廓；第三幅为序列重定位到新轮廓

部分的足部高度被调整了。注意，虽然这个方法可以“工作”，但是对最后一个运动不太可行，因为它下肢的角度几乎接近竖直。这个缺陷可以通过施加关节角度限制来解决，在 10.8.2 节中会有介绍。

在前一种办法中，在身体前移的过程中，IK 被用来放置或者锁定足部到步伐位置。当足部第一次迈离地面的时候，IK 被初始化；当身体前移时，足部保持在全局空间的地面固定位置。当足部离开地面的时候，IK 被释放。

考虑一个针对常规的走路周期的方案。在相同的时间间隔中，指定足部必须保持在地面。当到达这样的一帧时，在时间 t_{left_on}，足部的 IK 被打开，用来计算上下肢的相对方向，以使在臀部往前移动的时候足部保持在地面上。在接下来足部保持在地面的帧里，IK 控制器使足部固定在第一次足部留在地面的关键帧（t_{left_on}）中的位置。这意味着 IK 打开时，没有任何不连续性需要处理。尽管如此，在 IK 关掉以后，腿部必须占据在帧 t_{left_off} 时所在的空

间位置。这是必须融合的一个空间不连续性。

MoCap 数据中的高频成分问题非常的重要——它包含了运动的细节——例如，踢腿的一个片断。重定位时保持细节和不引进原来不存在的高频成分同样的重要。

在对基于约束的方法进行的一个很有见解的分类中，Gleicher［GLEI01］把本节所描述的方法命名为 PFIK + F（Per Frame IK plus Filtering）。Gleicher 的方法是 IK 求解器所确定的约束应当满足，并带来尽可能少的运动改变。对每一个约束帧，一个 IK 求解器被激活，然后通过应用一个位移图（见 10.4.4 节）满足这些约束。这个位移图从一个粗略的层次出发并逐渐细化。在每一次迭代，计算原来和现在位移层次的和作为新运动。

Lee 等人［LEE99］通过使用一种多分辨率的方法来解决运动适应问题，这种方法结合了层次化 B 样条方法和 IK 方法。MoCap 数据通过指数图来表示。层次化 B 样条插值在 10.5.1 节中介绍。Lee 等人使用这个方法产生一个（由粗糙到细致的）层次化的位移图（见 10.5.1 节），逐渐的从原始运动 M_0 过渡到最终运动 M_h。

$$M_h = (\ldots((M_0 + d_1) + d_2) + \ldots + d_h)$$

定义位移层次为

$$\mathbf{d} = \sum_{k=1}^{h} d_k$$

对要求的运动的每一个特定帧，位移被 B 样条曲线所插值，并以图 10-6 所示的方式传播到相邻的帧。对每一约束帧，IK 求解器（第 11 章）给出了一个被要求满足约束的角色的格局 $M(t_j)$。在众多可能的解决方案中，使 $M(t_j)$ 和之前运动差异最小的那种方案被选择。算法变换运动 M_0 到 M_h，并满足一系列的约束帧（记为 (t_j, C_j)），如下：

for $k = 1$ **to** h

　for each 约束帧(t_j, C_j)

　　$M(t_j) = \text{IK\_ solver}(C_j, M_{k-1}(t_j))$

　　$d(t_j) = M(t_j) - M_{k-1}(t_j)$

　　$D = D \cup (t_j, d(t_j))$

通过曲线拟合 D 计算 d_k

$$M_k = M_{k-1} + d_k$$

在应用于 MoCap 上下文时，层次化 B 样条插值的优点依然保持。从粗糙层次到细化层次，并以此找到位移图，就意味着 M_0 中的运动细节要保留下来。使用单层 B 样条插值可能会因引进波动使这些细节受到干扰。这个最粗略层的位移图在节点之间有最大的空隙，以及最大范围的影响。在后面层次的位移图中，范围逐渐缩小到跟运动非常和谐。当 h 被设为 4 或 5 时，作者取得最好的结果。

Lee 等人指出他们的方法存在的一个局限是，无法处理跨越多帧的帧间约束。下一节介绍的方法将解决这个缺陷。

Gleicher［GLEI01］采用了同样的方法，但是它使用标准信号处理术语来描述，并且直接以欧拉角表示来计算。位移图再次被迭代地计算，作为新旧运动的差异，然后用一个低通滤波器正常过滤。在每一次迭代，滤波器的截止频率在增长（意味着越来越小的内核）。因此约束再一次和最低可能频率位移图重合。

10.8.4　时空法

时空法应用于 MoCap 数据取得的最早成果见［GLEI98］。像 Witkin 和 Kass 的动力学约束法一样，这个方法同时考虑整个运动序列。与上一节所讲的 PFIK + F 方法不一样，前者分开考虑每一帧。通过最小化为获得新运动所需要的变化，并限制它们的频率内容，它和PFIK + K 方法一样，可以完成这样的效果。此方法把变形作为由条件限制的优化问题。它使用最简单的方法比较运动和空间（或几何）约束。因为它固有的代价，以及它从一个序列转换到另一个序列时要计算整个序列的事实，这种方法是离线的。

和前一种方法一样，仍然使用位移图的概念。B 样条表示和可变的控制点间隔协同工作，以限制高频率的引进变化的内容。Gleicher 在每 2、4 或 8 帧使用控制点间隔，计算了一个原始运动的带通分解，选择跟层次架构中的最低的，能量贡献超过阈值的层相符合的关键间隔。

有条件限制的优化方法要求一个目标函数。最简单的例子为：

$$g(M) = \sum_{i=1}^{n} (M(t_i) - M_0(t_i))^2 = \sum_{i=1}^{n} d(t_i)^2$$

其中 $M_0(t)$ 是原始运动序列。

这个方案的总体目标是最小化满足新约束的变形所需要的努力。算法的初始化步骤是必须的，因为层次的根部的位置偏移量不是独立于缩放比例的（不像角度 MoCap 数据）。在 Gleicher 给出的一个例子中，一个人靠近一个物体然后把它捡起来。对于人物位置的约束是他必须（在中间这一帧中）足够接近物体从而可以碰到它。如果人物变小了那么整个运动序列将会远离物体。我们需要通过插入常数位移并对平移数据应用一个偏移量来纠正这一问题。

算法的每一步如下所示：

1）定义约束。

2）通过对运动的平移参数操作来改变 $M_0(t)$，从而给出 $M_1(t)$ 的初始化方法。

3）基于频率分解为曲线选择一个 B 样条表示。

4）找到一个解决位移的约束优化方法，从而在加到 $M_1(t)$ 后可以使得运动满足约束。

5）如果第四步不足以满足约束，用 $(M_1(t) + dt)$ 替换 $M_1(t)$ 并使用下一个较密的控制点集合。

在 PFIK + F 上的时空方法的一个明显好处是可以使得很多帧上的约束得到解决。在 PFIK + F 方法中出现的问题可以独立地处理。在［GLEI01］中 Gleicher 指出，这个方法主要的灵活性是它可以处理“不必在意”的约束。他给出了一个关于足部放置的约束。其实真正的约束（在大多数情况下）是敲击地面的脚——敲击的确切位置并不重要。

附录 10.1　示范

Wfft1D.exe 是一个一维 FFT 程序。它通过方程式解析器接受功能定义。例如，图 10-15 是通过以下式子产生的：

abs(x) < 4.5? sin(4*x):0

abs(x) < 4.5? cos(x/4)*cos(x/4)*sin(4x):0

第 11 章　反向运动学原理

角色的运动学动画包含了所有不强调动态分析的方法。当前的类人角色动画方法，无论离线动画还是实时动画，几乎都属于它研究的范畴。其中，规范的做法是正向运动学方法。然而因为某些原因，尤其是实时算法的发展，反向运动学正在发挥着越来越重要的作用。

本章通篇讨论的，是正向运动学驱动的简化角色系统，这些角色用骨架表示，而运动数据则从运动捕捉或者是由动画师参与的关键帧插值动画系统得到。在正向运动学中，通过这种方法指定角色的“姿势”：将角色表示成关节结构，其姿势信息表示为一系列的交角——状态向量 $\boldsymbol{\theta}$ 以及根节点的位置和方向向量（见 10.8.2 节）。

在正向运动学中有很多问题即使是花大工夫也是很难协调的。当然这也成为使用 MoCap 方法的动机——因为 MoCap 方法避免了这些问题。而在电脑游戏中反向运动学（IK）又能够很好地与 MoCap 数据兼容（见第 10 章）。

先由正向运动学和反向运动学作用空间的区别来考虑它们之间的不同：

1）关节空间是关节角度构成的多维空间。它的维数与模型骨架的自由度数相等。而模型的一个姿势则是该空间的一个点。当模型移动时，所有与该模型姿势对应的点则构成了该空间的线（或称路径、轨迹）。

2）末端效应器空间是一个由受动末梢构成的 m 维空间。m = 受动末梢数 × 对应自由度。

3）世界空间是显示角色的空间。

对于前两种空间，通过这种概念可以看出正向运动学与反向运动学的关系模式（见图 11-1）。

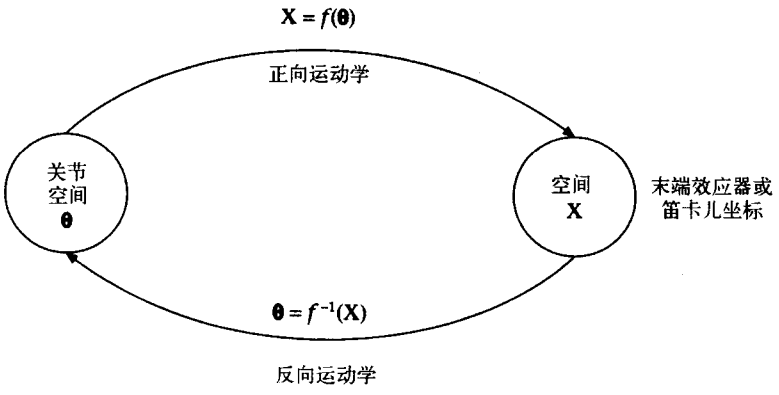


图 11-1　正向运动学和反向运动学

因此把正向运动学写成：

$$\mathbf{X} = f(\boldsymbol{\theta})$$

意思是骨架上任意一点的位置（通常是一个末端效应器）是关节角度的一个函数。这个表达式控制着骨架的姿势，以及在世界空间中显示角色时对其皮肤顶点位置的计算，顶点与骨架组织在一起（见第 8 章）。我们通过在正向运动学方程中加入随时间变化的关节角度变量，使角色动起来，而后用细化算法将骨架转化为（世界空间中的）覆盖有皮肤的角色。

对于 IK，相应的式子为：

$$\boldsymbol{\theta} = f^{-1}(\mathbf{X})$$

这里 **X** 通常是末端效应器的一个指定位置，在角色动画中就是角色的手和脚的目标位置。该表达式模拟了这样的情况：比方说，如果想移动手去抓一个物体，那就要把 **X** 设置为目标位置，用 IK 来计算满足条件的 **θ**。（尽管此处为了简化把 **θ** 作为角色的姿势的决定因素，但仍需要考虑根节点的位置。在物体抓取场景中，如果该物体在角色的抓取范围内，则只需更改 **θ**；否则，若物体超出了这个范围，IK 方法会修改根节点。）

在 MoCap 适应（对于关注电脑游戏的程序）中，IK 的动机作了这样的修改：给定一个满足一定约束的序列，我们希望使角色的动作能满足新的约束。但在最一般的情况下，游戏角色的比例与真实表演者是不同的，或者一个剧情要求对这种序列作一些变更——比如，地形变了，那么跑步的姿态就要进行这样的适应性调整。此处，用 IK 的方法来计算满足某种约束的角色姿势。之后的一些工序包括修改原始动作序列以使其看起来能够更自然地满足要求。这些话的确切意思在第 10 章提到过。很重要的一点是 IK 本身并非 MoCap 适应的解，它只是两阶段方法的其中一个阶段。

我们用一个简单的例子引入主题，这个例子称为二链臂（two-link arm）（见图 11-2）。尽管与人的骨架，甚至与电脑绘制的骨架比起来，它是一个简单得有些滑稽的例子，但在稍后的 11.1 节中我们仍将学习怎样用二链结构（虽然是非平面的）来简化骨架。二链装置的一边是固定的，另一边是末端效应器，两个臂用一个自由度（铰链）连接。反向动力学的研究是个难题，但是，从特殊问题起步推广到一般问题是个不错的办法。这个例子用来演示两方面问题：

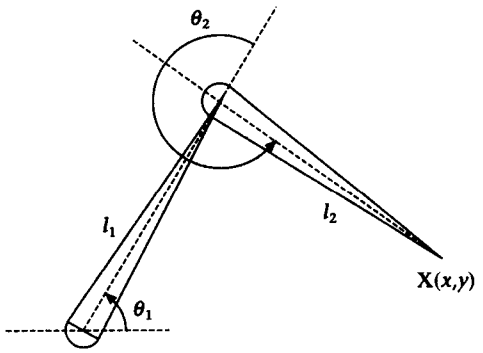


图 11-2　简单二链臂结构

- IK 解法中解析和封闭（closed-form）的求解概念。这些对于实时应用程序是极其重要的。虽然分析方法对于解决 50 个自由度的计算机图形骨架无能为力，但对于此结构的一部分使用闭式方案却是可行的，后面将会介绍。
- 雅可比概念的重要性。雅可比方法产生了最一般的 IK 解决方案。

闭式的发展多半得益于机器人领域，在那个领域人们的兴趣是有效地控制诸如 PUMA 臂的工业机器人（见［CRAI88］）。与电脑绘制的结构（50 自由度）以及真人的骨架结构（>250自由度）相比，这些机器人的自由度（通常 6 个）真是太少了。而闭式解法仅能解决自由度数不大于 6 的结构。

11.1　例子——二链臂

怎样研究简单链接机制呢？在每个链接点设置一个坐标框架，并且表达一种结构——如

一串“转换”的全部运动，这种“转换”表达了相对框架 n-1 的框架 n 的运动。这使我们能表达末端效应器的动作，即相对于框架 0 的框架 n：

$$ {}^{0}_{n}T = {}^{0}_{1}T\,{}^{1}_{2}T\,{}^{2}_{3}T\cdots{}^{n-1}_{n}T $$

（这个表达式是等式 7-1 的另一种形式。）末端效应器的位置和方向由各个链变换串联给出。

考虑一个简单的例子——平面双连杆结构。这里设置 4 个框架，框架 0 是不能移动的基本框架，框架 1 与基本框架共原点且能在纸平面内旋转，框架 2 也能在纸平面内旋转，而最终的末端效应器就是框架 3。一直到最后一个链终点都是固定不动的。这样，这个臂就有两个自由度，与它的运动所在空间的维数是一样的。

再把变换式写成如下形式。先考虑框架 1 相对于框架 0 是如何运动的——它只能旋转，因此有：

$$ {}^{0}_{1}\mathbf{T} = \begin{bmatrix} c_1 & -s_1 & 0 & 0 \\ s_1 & c_1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

类似地：

$$ {}^{1}_{2}\mathbf{T} = \begin{bmatrix} c_2 & -s_2 & 0 & l_1 \\ s_2 & c_2 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

其中 c_1，c_2，s_1 和 s_2 分别是 $\cos\theta_1$，$\cos\theta_2$，$\sin\theta_1$ 和 $\sin\theta_2$ 的缩写形式。并且：

$$ {}^{2}_{3}\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & l_2 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} $$

它反映了这样一个事实：末端效应器与此链是刚性连接——末端效应器仅有两个自由度。

反向运动学需要计算当末端效应器从一个位置移动到另一个位置时，一个铰链结构的构造和位置。这是一个理想的范例，因为在许多应用中，我们感兴趣的只是移向目标的动作，比如说我们要求末端效应器移到三维空间去执行某项任务等。如果能从末端效应器的动作中得出关节的动画，那么就有了对自主动作问题的潜在解法。在自主动作中，我们对于从结构中末端效应器得到的动作有很高加工要求。比如“绕过桌子拿起茶杯”。（注意，与当前应用 IK 适应 MoCap 序列的游戏应用程序相比这是个更高的目标。）然而，完全利用 IK 开发出来的动画，没有正向运动学或 MoCap 的良好的数据基础——因此这种动画效果看起来像极了木偶，所以我们把它看成是一个工具，而不是一个完整的解决方案。

反向运动学中的许多研究已经应用到机器人控制领域。在该领域，问题的复杂性与真人问题的复杂性相比确实降低了许多，而且在这些应用中该技术也更为有效。IK 方法通过控制关节马达使机器人移动到新的目标。在这种情况下，二链臂三角操控给出了如下闭式解法：

$$ \theta_2 = \cos^{-1}\left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right) $$

$$\theta_1 = \tan^{-1}\left(\frac{-l_2\sin\theta_2 x + (l_1 + l_2\cos\theta_2)\ y}{l_2\sin\theta_2 y + (l_1 + l_2\cos\theta_2)\ x}\right)$$

当链越来越多，越来越复杂时（随着自由度数量的增加），闭式解法就越发的难找了。闭式解法的另一个问题是很难将其推广到多重约束的情形，尤其是在多重约束相互关联时。不过它有个显著的优点，即快速！仅仅凭借一个公式就能得到解法，这与采用迭代的其他所有方法形成了鲜明的对照！

11.2　雅可比矩阵

在没使用闭式解法的应用程序中，可以构造反向运动学解法——通过研究末端效应器的小幅移动来解决问题，我们假设关节速度与末端效应器的速度满足线性关系：

$$\dot{\mathbf{X}} = \mathbf{J}\dot{\boldsymbol{\theta}}$$

J 是雅可比矩阵，它是末端效应器速度与关节速度相对应的多维导数。它是 $m \times n$ 的矩阵，n 是关节变量数（总的自由度数），而 m 是末端效应器的自由度数。J 的第 i 列代表由 θ_i 的增量引起的末端效应器位置和方向的增量。如果 J 的逆阵存在，则就有：

$$\dot{\boldsymbol{\theta}} = \mathbf{J}^{-1}\dot{\mathbf{X}}$$

我们将重新考虑二链臂并推导它的雅可比阵。为此，要推导结构中每一个框架的线速度和角速度的公式，而后表达出末端效应器相对于基本框架的速度。基本框架的线速度和角速度均为 0，我们从框架 1 开始。对框架 1，线速度是 0，因其只能旋转，就有：

$${}^1\omega_1 = \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_1 \end{bmatrix}$$

$${}^1\mathrm{v}_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

要计算框架 2 的角速度，只需再加上框架 1 的角速度：

$${}^2\omega_2 = \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_1 + \dot{\theta}_2 \end{bmatrix}$$

框架 2 原点的线速度值是框架 1 原点的相应值加上由框架 1 角速度引起的变化。

$${}^2\mathrm{v}_2 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} + \begin{bmatrix} \mathrm{c}_2 & \mathrm{s}_2 & 0 \\ -\mathrm{s}_2 & \mathrm{c}_2 & 0 \\ 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 0 \\ l_1\dot{\theta} \\ 0 \end{bmatrix} = \begin{bmatrix} l_1\mathrm{s}_2\dot{\theta}_1 \\ l_1\mathrm{c}_2\dot{\theta}_1 \\ 0 \end{bmatrix}$$

最终有：

$${}^3\omega_3 = {}^2\omega_2 = \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_1 + \dot{\theta}_2 \end{bmatrix}$$

$$ {}^{3}v_3 = \begin{bmatrix} l_1 s_2 \dot{\theta}_1 \\ l_1 c_2 \dot{\theta}_1 \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ l_2 \ (\dot{\theta}_1 + \dot{\theta}_2) \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ l_1 c_2 \dot{\theta}_1 + l_2 \ (\dot{\theta}_1 + \dot{\theta}_2) \\ 0 \end{bmatrix} $$

为了表示出末端效应器在基本框架坐标系统中的线速度，需要旋转矩阵：

$$ {}^{0}_{3}R = {}^{0}_{1}R^{1}_{2}R^{2}_{3}R = \begin{bmatrix} c_{12} & -s_{12} & 0 \\ s_{12} & c_{12} & 0 \\ 0 & 0 & 1 \end{bmatrix} $$

得出：

$$ {}^{0}v_3 = \begin{bmatrix} -l_1 s_1 \dot{\theta}_1 - l_2 s_{12} \ (\dot{\theta}_1 + \dot{\theta}_2) \\ l_1 c_2 \dot{\theta}_1 + l_2 c_{12} \ (\dot{\theta}_1 + \dot{\theta}_2) \\ 0 \end{bmatrix} $$

这样就得到由微分链式规则表示的雅可比阵：

$$ J = \begin{bmatrix} -l_1 s_1 - l_2 s_{12} - l_2 s_{12} \\ l_1 c_1 + l_2 c_{12} \quad l_2 c_{12} \end{bmatrix} $$

随着结构越加复杂化，使用微分法求雅可比阵越来越困难。事实上，必须要使用几何法构造复杂结构，即先确定大体的原则，然后按几何法计算平面二链臂的雅可比阵。该方法的原则可直接推广到三维空间，具体的做法见文献［WATT92］。

考虑用链结构中由始至终传播的速度表示的末端效应器的动作。一个 n 环的链中每个链接皆受到一个线速度（因为关节传动）和一个角速度（因链是绕关节点旋转的）的制约。我们理所当然地承认以下原则：

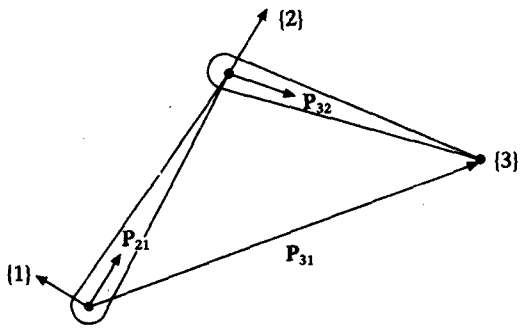


图 11-3　二链臂的向量与框架

- 链尾角速度等于全部角速度之和（取基本框架系统的值）；
- 链尾线速度等于全部中间框架的角速度与某个“相应向量”的叉积之和。其中“相应向量”是指关节终点到相应框架的原点的向量。

一个更简单的表示方法见图 11-3。有了以上原则，可有：

$$ \mathbf{v}_{30} = \boldsymbol{\omega}_{10} \times \mathbf{P}_{31} + \boldsymbol{\omega}_{21} \times \mathbf{P}_{32} $$

位置向量由下式给出：

$$ \mathbf{P}_{31} = (l_1 c_1 + l_2 c_{12}, \ l_1 s_1 + l_2 s_{12}) $$
$$ \mathbf{P}_{21} = (l_2 c_{12}, \ l_2 s_{12}) $$

角速度为：

$$ \boldsymbol{\omega}_{i0} = \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_1 \end{bmatrix} \boldsymbol{\omega}_{21} = \begin{bmatrix} 0 \\ 0 \\ \dot{\theta}_2 \end{bmatrix} $$

将以上值代入 $\mathbf{v}_{30}$ 的等式中，整理可得出与以前相同的结果。

在继续之前我们要指出，雅可比阵在该例中对所有的 $\boldsymbol{\theta}$ 都是有意义的。在大多数情况下，雅可比阵是 $\boldsymbol{\theta}$ 的一个函数，并且在迭代方法中要被一再地重复计算。

11.3　IK 方法

考虑到大多数较为让人感兴趣的系统都过于复杂，难以适用闭式解法，我们把目光投向那些可以处理的方式上去，它们是：

- 几何/分析法（geometric/analytical）。仅用一步就可以产生一个目标状态，因而很快。虽然像我们已经讨论过的一样，它不是一个对任意复杂的问题都通用的解法，但是它可以作为混合方法的一部分来使用。
- 微分算法（differential algorithms）。对小幅变化使用雅可比阵使问题线性化，并由迭代方式产生出解法。也就是说，对于研究小幅变化的线性，将末端效应器移向目标解的迭代方案，雅可比阵可以嵌入其中。任何迭代解法都利用了函数的收敛性。
- 非约束最优化方法（unconstrained optimization methods）。该方法讨论不满足约束的解法，而不是找到一个满足约束的解法。
- 循环坐标下降法（cyclic coordinate descent）。这也是一个向解逐步靠拢的算法。但此方法的每一步骤都涉及启发式构造。
- 混合方法（hybrid methods）。这是一个组合方案，其动机常常是实现实时应用。

11.3.1　使用雅可比阵的微分方法

首先重新设置授权等式：

$$\dot{\boldsymbol{\theta}} = \mathbf{J}^{-1}\ (\dot{\mathbf{X}})$$

写成迭代形式为：

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{-1}\ (\Delta\mathbf{x})$$

每次迭代中，将末端效应器移近目标位置 $\Delta\mathbf{x}$ 单位，并计算状态向量的相应的变化量 $\Delta\boldsymbol{\theta}$。当末端效应器移动到目标位置时迭代结束。写成伪代码为：

$\Delta\mathbf{x}$ = small movement in the direction of the goal

repeat

$\Delta\boldsymbol{\theta} = \mathbf{J}^{-1}(\Delta\mathbf{x})$

$\mathbf{x} = \mathrm{f}(\boldsymbol{\theta} + \Delta\boldsymbol{\theta})$

calculate new value of J and invert

$\mathbf{x} = \mathbf{x} + \Delta\mathbf{x}$

until goal is reached

三链臂（three-link arm）的一次迭代见图 11-4。迭代中的问题来自错误跟踪（tracking error），这错误在希望 $\mathbf{X}$ 发生的变化与实际变化不同时发生。当 $\Delta\mathbf{x}$ 太大时会发生错误跟踪，由下式给出：

$$\| \mathbf{J}\ (\Delta\boldsymbol{\theta})\ - \Delta\mathbf{x} \|$$

由此必须引入由一个 $\Delta\mathbf{x}$ 开始的更有效的迭代，用来计算错误跟踪以及再进一步细分 $\Delta\mathbf{x}$ 直到错误小于某个极限。

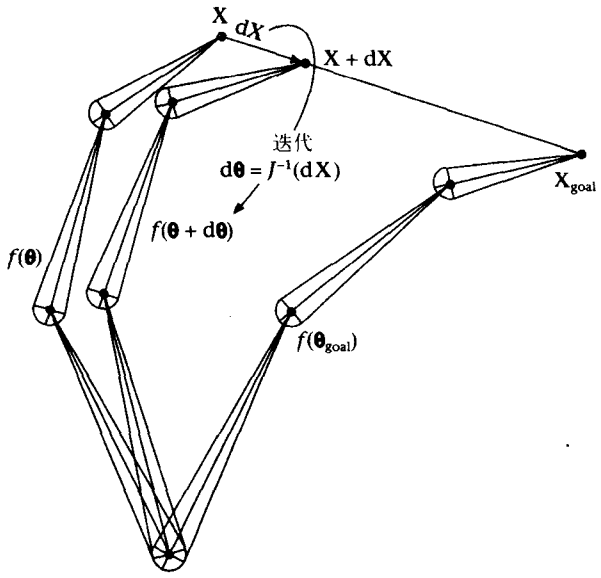


图 11-4　一次迭代中向目标位置移动的一步

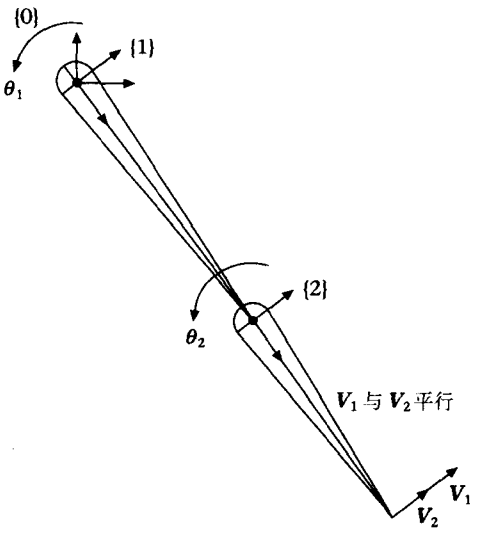


图 11-5　完全展开二链臂的奇异点

在这个看起来很简单的方法中还有其他一些固有的难点。首先，系统的行为在奇异点及其附近会出现问题。再来考虑完全展开的二链臂（见图 11-5）。θ_1 或 θ_2 的改变都会导致动作向相同的方向变化。我们称这种现象为一个自由度丢失。换句话描述该问题：即我们规定，不存在会导致末端效应器有朝向或背离基本框架的速度的状态空间速度。尽管奇异点的处理是该方法面临的一个难题，我们仍需要处理这些点。毕竟，一个完全展开的手臂或腿是很不自然的姿势。我们很快会讨论其他方法是怎样处理这一问题的。

从数学方面考虑，随着姿势的改变我们重新计算雅可比阵，而所谓的奇异点会影响矩阵的秩。矩阵的秩定义为矩阵中线性独立性最大的行数或列数。如果矩阵的行列式值是 0，则该矩阵就损失了它的秩。在二链臂的情况下有：

$$|\mathbf{J}| = \begin{vmatrix} -l_1 s_1 - l_2 s_{12} & l s \\ l_1 c_1 + l_2 c_{12} & l c \end{vmatrix} = l_1 l_2 s_2$$

当完全展开（θ_2 是 0 或者 π）时，上式值为 0。通常，在处理奇异点时，关节速度需要将末端效应器沿着链向趋于无穷大的方向移动。所以，采取更复杂的启发性的措施来避免奇异性是必要的。我们可以通过检测 J 的稀疏程度来确定状态是否处于奇异点附近。最普遍的处理方法［MACI90］是找到一个解，它能使下述和式最小：

$$\| \mathbf{J}(\Delta\boldsymbol{\theta}) - \Delta\mathbf{X} \|^2 + \lambda^2 \| \Delta\boldsymbol{\theta} \|^2$$

这里第二项表示的是状态空间速度。此策略的目的是使错误跟踪和状态空间速度最小。λ 是衰减因子，它定义了跟踪错误与状态空间速度的模的相对重要性。

最后还有一个困难是，对任何有意义的系统，雅可比阵都不是正方形的——在实际的结构中，$\mathbf{X}$ 的维数比 $\boldsymbol{\theta}$ 的维数小——因此它不可逆。造成此问题的实际原因是：骨架高度冗余——它们处理的自由度比要达到目标所必须的自由度要多。有个例子能说明这个问题，就是人的胳膊。如果不考虑手的复杂性，则末端效应器有 6 个自由度（含位置和方向），这就

形成了一个少于7个自由度的胳膊的骨架。它由两个球形关节（肩膀和手腕）加上一个铰链关节（肘部）组成。这可以在物理层面上反映出来：当我们把手和肩膀固定在某个位置时，同时就形成了一个与腕肩连线相垂直的平面，而肘部可以沿着该平面内的一条弧旋转（见图11-6）。动作是由肩关节的冗余自由度引起的。这叫做肘环（elbow circle）。

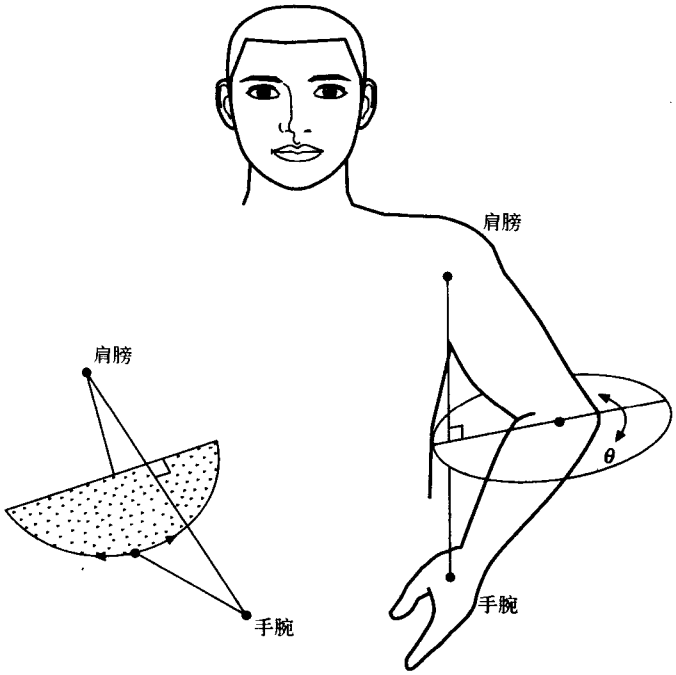


图11-6　肘部旋转角度 θ。肘关节可以在与腕肩连线垂直的平面中自由移动，而不会影响手的位置

额外的自由度意味着存在无穷多个解法，我们就说这种结构能够承受自发运动。也就是说，该结构可以移动以至形成一个新姿势而不会移动末端效应器。也可以说雅可比变换有一个包含了无数个关节空间速率的无效空间，这些关节空间速率都不会引起末端效应器的任何动作。对于一个冗余结构，不论使用什么方法，都只是从无限的解中找出一个解。我们通过选择一个满足要求的解来利用冗余性。也就是说，我们将一些约束合并到解法中。

运动学冗余操作提供了比非冗余操作更可靠的临界优势。特别是，我们可以将障碍避免、自碰撞避免、奇异避免加入其中。概括地说，冗余结构是一种更加灵活的结构。

这种情形可以用数学方式模拟为：

$$\Delta\boldsymbol{\theta} = \mathbf{J}^{+}\Delta\mathbf{X} + (\mathbf{I} - \mathbf{J}^{+}\mathbf{J})\Delta\mathbf{Z}$$

这里：

J 是方阵，称为伪逆阵

$\Delta\mathbf{X}$ 是主任务

I 是秩与关节空间的维数相等的单位阵

$\Delta\mathbf{Z}$ 是次任务

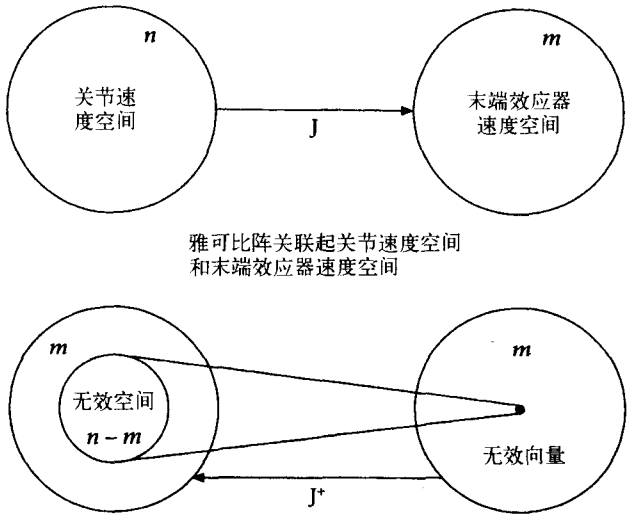


图 11-7　伪逆阵使 m 维的末端效应器速度空间与 m 维的关节速度空间的子空间相关联

雅可比阵 J(秩为 r 的 $m \times n$ 矩阵)的伪逆阵 J^+ 由下式给出：

$$J^+ = \begin{cases} (J^TJ)^{-1}J^T & \text{if } m > n = r \\ J^T(JJ^T)^{-1} & \text{if } r = m < n \end{cases}$$

解的第一部分称为伪逆阵，第二部分称为通解。通解导致末端效应器的速度为 0，这一点可以在图 11-7 中看到。在第一个图中，雅可比阵 J 使 n 维的关节速度空间和 m 维的末端效应器速度空间（$n > m$）相关联。在第二个图中，伪逆阵 J^+ 使 m 维的末端效应器空间与一个 m 维的关节速度空间的子空间相关联。$(I - J^+J)$ 是一个算子，它选择均一解集合中 $\Delta\mathbf{Z}$ 的成分。即，因为 $\Delta\mathbf{Z}$ 投影到无效空间并且末端效应器动作不变，我们设置它为次任务。

因此，次任务是用来满足约束的而不只是满足目标。我们知道，这是利用了冗余性。例如，我们能够在不改变末端效应器位置的情况下改变角色的姿势——这正好是图 10-21 的要求。实现伪代码如下：

计算　　$\Delta\boldsymbol{\theta} = J^+\Delta\mathbf{X}$

计算　　$\boldsymbol{\theta}_t = \boldsymbol{\theta}_{t-1} + \Delta\boldsymbol{\theta}$

计算　　$\Delta\mathbf{Z}$　（见下一节的例子）

通过计算下式投影到无效空间：

$$(I - J^+J)\ \Delta\mathbf{Z}$$

将结果加到 $\boldsymbol{\theta}_t$

次任务——关节限制

最一般的次任务是用来保证一个较好的关节角度姿势的。例如，从期望的配置中产生一个能最小化关节角动作的姿势——称为中值角（mid-range angle)。把 $\Delta\mathbf{Z}$ 设置为梯度：

$$\Delta\mathbf{Z} = -2\ (\boldsymbol{\theta} - \boldsymbol{\theta}_M)$$

这里：

$\boldsymbol{\theta}$ 是当前值

$\boldsymbol{\theta}_M$ 是中值

更一般地我们写成：

$$\Delta \mathbf{Z} = \Delta H$$

此处

$$H = \sum_{i=1}^{n \text{ of DOFs}} \alpha_i (\theta_i - \theta^M)$$

其中 α_i（$0 \leqslant \alpha_i \leqslant 1$）是定义关节僵硬程度的附加值。

图 11-8（彩页中也有）显示了在微分法中加入的关节角度约束产生的影响。图中，三链平面结构中的末端效应器（蓝色）移向一个假想的动画目标。其中的约束是：

$$-90° \leqslant \theta_1 \leqslant 90°$$

$$0 \leqslant \theta_2 \leqslant 140°$$

$$-10° \leqslant \theta_3 \leqslant 10°$$

图 11-9（彩页中也有）显示了相同的结构，其末端效应器也在几乎相同的位置，但此次关节约束取消了。

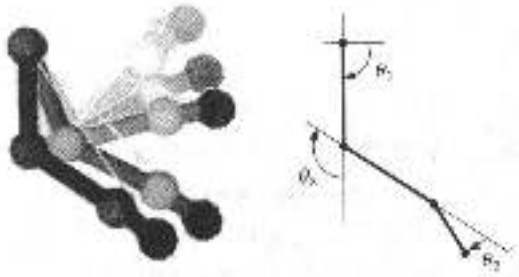


图 11-8　带关节角度约束的三链臂结构的微分 IK 解法

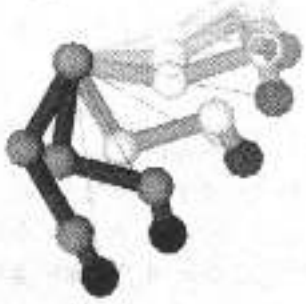


图 11-9　与上面的解法相比，不带关节约束的三链臂结构的情形

11.3.2　最优法

IK 可以作为非线性最优化问题来研究。这就避免了 IK 微分方法中的很多问题。值得一提的是该方法更为稳定。最小化方法使用标准迭代法（例子见［PRES01］），以将服从约束的错误函数最小化为：

$$\mathbf{E}(\boldsymbol{\theta}) = (\mathbf{X}_t - \mathbf{f}(\boldsymbol{\theta}))^2$$

其中 $\mathbf{X}_t$ 是目标位置或约束位置。

该方法能合并新添加的约束或第二目标（如关节限制），这种情况写成：

最小化满足约束 $\mathbf{C}(\boldsymbol{\theta})$ 的 $\mathbf{E}(\boldsymbol{\theta})$

迭代法扰动 $\boldsymbol{\theta}$ 使得 $\mathbf{E}(\boldsymbol{\theta})$ 减小。所有这种方法都会遇到经典的局部极小点问题，即可能会找局部最小值而不是全局最小值。

取决于末端效应器的目标是位置还是位置加上方向，$\mathbf{E}(\boldsymbol{\theta})$ 会更加复杂。基于此，有：

$$\mathbf{E}(\boldsymbol{\theta}) = \mathbf{E}_p(\boldsymbol{\theta}) + \mathbf{E}_o(\boldsymbol{\theta})$$

如果定义末端效应器的目标为 $\mathbf{X}_t$，其当前位置为 $\mathbf{X}_c$，而其方向目标和当前方向分别是（正

交的）矩阵 R_g，R_c，那么 E（θ）可以写成：

$$E(\theta) = \| (X_g - X_c) \|^2 + \sum_{j=1}^{3} ((r_{jg} \cdot r_{jc}) - 1)^2$$

其中 r_j 是矩阵的行。

11.3.3　循环坐标下降法（CCD）

CCD 算法是 Wang 和 Chen［WANG91］在 1991 年引入的，目的是用于工业机器人控制。这是一个快速的算法，它通过对关节角度的一次性操作使错误达到最小。尽管它与最优化方案有些关系，但它更是一种使用了非线性编程工具（CCD）来寻找解法的启发式直接搜索方法。该算法对链的初始构造和奇异构造都不敏感。我们对它的描述的依据是 Welman［WELM93］的阐述。

算法在每次迭代中忽略从根到末端效应器的整个链，针对每个关节角度将末端效应器的位置和方向最小化。如果只对末端效应器的位置感兴趣，这个算法是相当简单的。图 11-10 显示了平面情形的两次迭代图解。在每次迭代，把链 i 旋转 ϕi 角，即链的当前方向与链原点到目标位置连线的夹角。

因此：

$$\phi_i = \cos^{-1}(P_{ig} \cdot P_{ic})$$

其中：

P_{ic} 是从链 i 到当前末端效应器位置的向量

P_{ig} 是从链 i 到末端效应器目标位置的向量

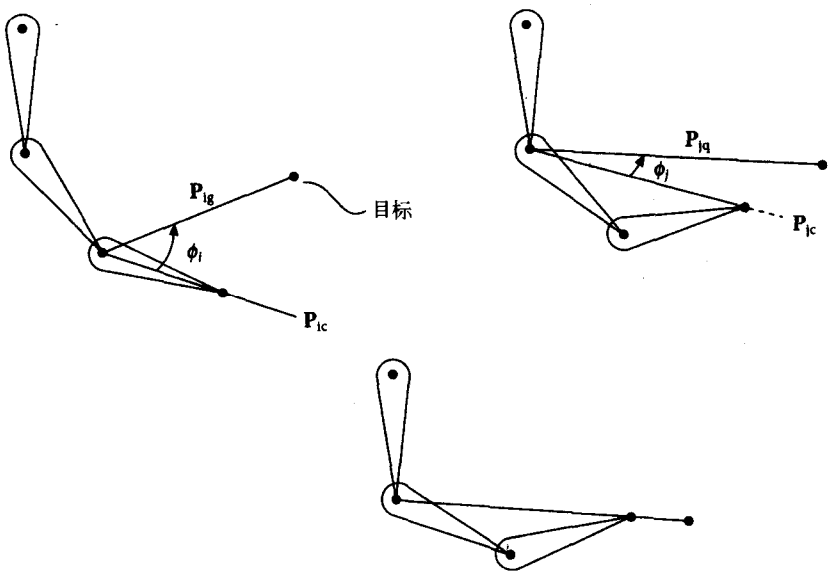


图 11-10　平面链 CCD 算法的两次迭代

如图所示，我们可以自由地把 P_{ic} 旋转到一个新的位置 P'_{ic}（ϕ），因为必须使 ϕ_i 最小，

应最小化下式：

$$g_1(\phi_i) = \mathbf{P}_{ig} \cdot \mathbf{P}'_{ic}(\phi_i)$$

在实际中，必须定义 $\mathbf{P}_{ic}$ 相对于关节 i 的特定自由度轴的旋转量，并设 $\mathbf{P}_{ic}$ 绕一个关节轴旋转角度 ϕ，有：

$$\mathbf{P}'_{ic}(\phi) = R_{(XYZ)i}(\phi)\mathbf{P}_{ic}$$

其中 $R_{(XYZ)i}(\phi)$ 是绕关节的 x,y,z 轴旋转(ϕ)角(相对于基本框架坐标系)的矩阵。

对一个 6 个自由度的末端效应器，定义一个类似的式子：

$$g_2(\phi_i) = \sum_{j=1}^{3} \mathbf{r}_{jg}\mathbf{r}_{jc}(\phi_i)$$

联立各式得到一个总的表达式（要对关节 i 进行最大化）为：

$$g(\phi_i) = w_p g_1(\phi_i) + w_o g_2(\phi_i) \tag{11-1}$$

其中 w_p 和 w_o 是位置和方向的权重因子。Wang 和 Chen 提出其值为：

$$w_p = \alpha(1 + \rho)$$

$$w_o = 1$$

其中 α 是一个基于全域尺寸的比例因子，它保证了算法不是比例依赖的。

$$\rho = \frac{\min(\|\mathbf{P}_{ig}\|, \|\mathbf{P}_{ic}\|)}{\max(\|\mathbf{P}_{ig}\|, \|\mathbf{P}_{ic}\|)}$$

现在，式 11-1 可以写成：

$$g(\phi_i) = k_1(1 - \cos(\phi_i)) + k_2\cos(\phi_i) + k_3\sin(\phi_i) \tag{11-2}$$

其中

$$k_1 = w_p(\mathbf{P}_{ig} \cdot R_{(XYZ)i})(\mathbf{P}_{ic} \cdot R_{(XYZ)i}) + w_o \sum_{j=1}^{3} (\mathbf{r}_{jg} \cdot R_{(XYZ)i})(\mathbf{r}_{jc} \cdot R_{(XYZ)i})$$

$$k_2 = w_p(\mathbf{P}_{ig} \cdot \mathbf{P}_{ic}) + w_o \sum_{j=1}^{3} (\mathbf{r}_{jg} \cdot \mathbf{r}_{jc})$$

$$k_3 = R_{(XYZ)i} \cdot [w_p(\mathbf{P}_{ig} \times \mathbf{P}_{ic}) + w_o \sum_{j=1}^{3} (\mathbf{r}_{jg} \times \mathbf{r}_{jc})$$

取目标函数（式 11-2）的一阶导数并设为 0，即：

$$(k_1 - k_2)\sin\phi + k_3\cos\phi = 0$$

有：

$$\phi = \tan^{-1}\frac{k_3}{(k_2 - k_1)}$$

该式在 $(-\pi/2 < \phi < \pi/2)$ 有一个可能的解，同时与之相对应的 $\phi + \pi$、$\phi - \pi$ 也是解。目标函数在一阶导数为 0 且二阶导数为负数时取最大值，因此最终应从上述解中选取一个满足第二个条件的值。

因为这种方法单独对每个关节进行操作，关节限制约束容易合并。但是，利用关节限制会影响算法的收敛性。该算法很明显的行为缺陷是，末端效应器附近的链会受到特别的偏爱，结果是仅有那个链会发生移动。这种效果可能会产生看起来不自然的结果，可以通过限制关节摇摆范围小于所计算出的动作来改善这种情况，且即使其值在关节限制允许的范围内，我们也不要放过。

11.4　反向运动学的实践方案

正如下面的例子将要显示的那样，IK 的实践方案是五花八门的。使用最多、最为普遍的简化方法是对结构的某些部分应用 IK 方法。我们没必要解决整个骨架的问题，可以分而治之，把大问题简化成能够处理的子问题。这样，我们就能在骨架中的任意两个关节之间应用 IK 方法，常见的简化是将胳膊和腿简化为二链结构。IK 求解过程能够只在节点之间操作，而末端节点可能就是另一个求解过程的输入或输出。

考虑一个任意的骨架结构（这在实际中是可能遇到的），它或许会包含一些分支——而这是我们所未考虑过的。我们可以在骨架中的任意两个节点之间应用 IK，惟一的规则是与末端效应器相对应的节点比与基本节点相对应的节点更能降低层次。可以把 IK 算法理解为一个附属引擎，对于骨架，有相应的函数来设定末端节点与基本节点（空节点）之间的所有节点的位置和方向，我们可以利用 IK 处理骨架的任意部分。很明显，我们可以用不止一个 IK 引擎来处理结构的不同部分。图 11－11 显示了为一个简单的人形骨架安排的 IK 引擎。根节点（黑色三角形）在骨盆处，基本节点（黑色正方形）在臀部和肩部，末端节点（黑色圆形）在手和脚上，而空节点（空心圆圈）在膝部和肘部。

当然，其他的安排也是可行的；例如，文献［PHIL91］中，将根节点放在一只脚上，把另一只脚当成末端节点，以便给站立的角色设置动画动作（包含弯曲、从一只脚到另一只脚的重量转移、转身等）。其他的规则也可以应用到那些骨架处理上存在引擎重合的情况。基本的步骤是，先为每个引擎编号，再赋予其优先级别——优先级高的比优先级低的先得到计

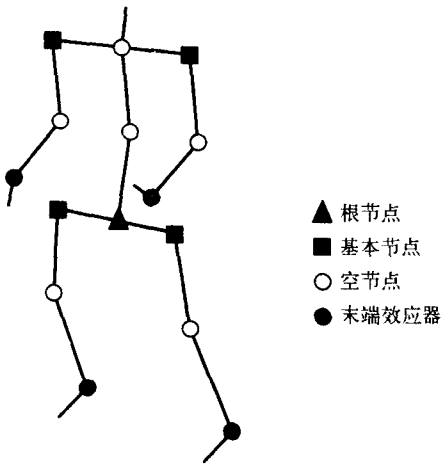


图 11-11　将简单骨架分成 4 × 2 的链结构，每一个都有单独的 IK 控制

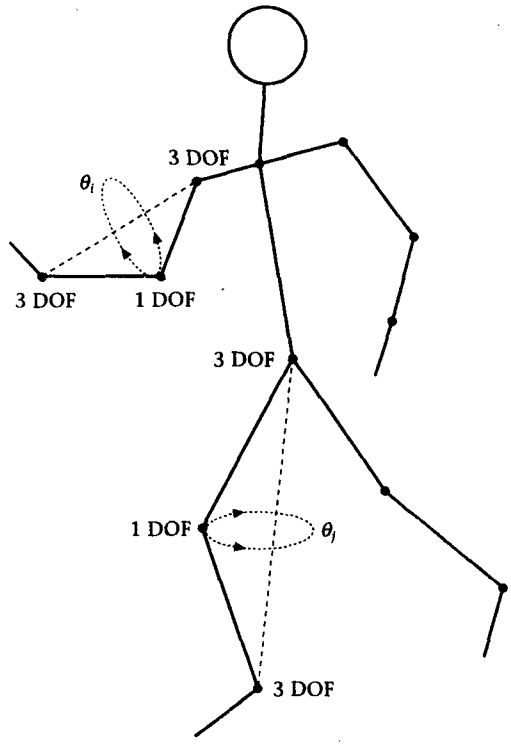


图 11-12　在［LEE99］中使用的 7 自由度肢体

算，而优先级低的则要处理剩余的所有空节点。

这些研究是当前及近期实际应用中的典型例子，而不能算是该领域的较为全面的综述。

11.4.1　混合方法——分析法＋约束最优化法

使用闭式解法的算法的优点是快速，同时还避免了微分方法的稳定性问题和最优化方法中的局部最小点问题。然而，闭式方法仅局限于简单的非冗余结构。因此，混合方法的动机是构造一个包含闭式解法的 IK 方案。

这就是 Lee 等人在［LEE99］中提出的方法，在该方法中，他们把分析法和约束最优化方法结合了起来。在他们的工作中，冗余臂自由度——图 11-6 中的肘环——被用在膝关节及肘关节，以简化角色的全局自由度公式（见图 11-12）。该模型有共 37 个自由度——其中 6 个用于骨盆的位置和方向，3 个用于脊椎，4×7 个用于四肢。采用这种方法的动机是运动捕捉适应，这方面的工作在第 10 章已经提到。

鉴于手和脚的位置都被约束固定了，角色所有可能的姿势均可用 4 个角度（θ_1，…，θ_4）加上肢体链接之外的自由度来指定。

IK 求解过程分为两个阶段：分析法求出 θ_i，然后是对惩罚方法（penalty method）的评定，该惩罚方法的输入为没有被肢体链接和那些通过改变 θ_i 可以移动的链接包含的自由度。

算法应用分析解法来固定肢体链的关节角度并且提供旋转角度 θ_i。然后把这种简化的约束集输入到最优化算法，由它找出剩余的（非四肢的）自由度，这里 θ_i 是可以自由改变的。

分析阶段见图 11-13，它是对一个手臂进行的。从一个初始位置开始，Lee 等人顺序地调整肘部、肩部、腕部关节角。首先，将肘部旋转 ϕ 以使腕部移动到距离肩 L 处，其中 L 是肩部到腕部的目标位置的距离。ϕ 由下式给出：

$$\phi = \cos^{-1}\left(\frac{l_1^2 + l_2^2 + 2(l_1^2 - r_1^2)^{\frac{1}{2}}(l_2^2 - r_2^2)^{\frac{1}{2}} - L^2}{2r_1 r_2}\right)$$

其中：

l_1 和 l_2 是上臂和前臂的长度

r_1 是肘部旋转轴到肩部的距离

r_2 是肘部旋转轴到腕部的距离

下一步是绕肩部旋转肢体以使配置达到要求。最终，调整腕部角度使之与期望的方向相一致。然后，冗余 θ_i 就可以利用了。

11.4.2　混合方法——三阶段：分析法＋约束最优化＋分析法

Shin 等人［SHIN01］把 IK 问题分成 3 个子问题：根位置估计，身体姿势计算，以及肢体姿势计算。同样，他们的动机也是动作适应——特别是对于实时动作适应的电脑木偶。他们指出，在许多问题中层次结构的根与其他关节如末端效应器相比通常没那么重要（见 10.8.2 节对于位置属性重要性的讨论）。所以，他们首先做的是把末端效应器移到距离根尽可能近的地方。移动根的位置是简单而有效的（与改变关节角度相比），Gleicher 在［GLEI98］中也有这方面的论述。

给定末端效应器的当前位置以及一个目标位置就定义一个位移向量，可以简单地将末端

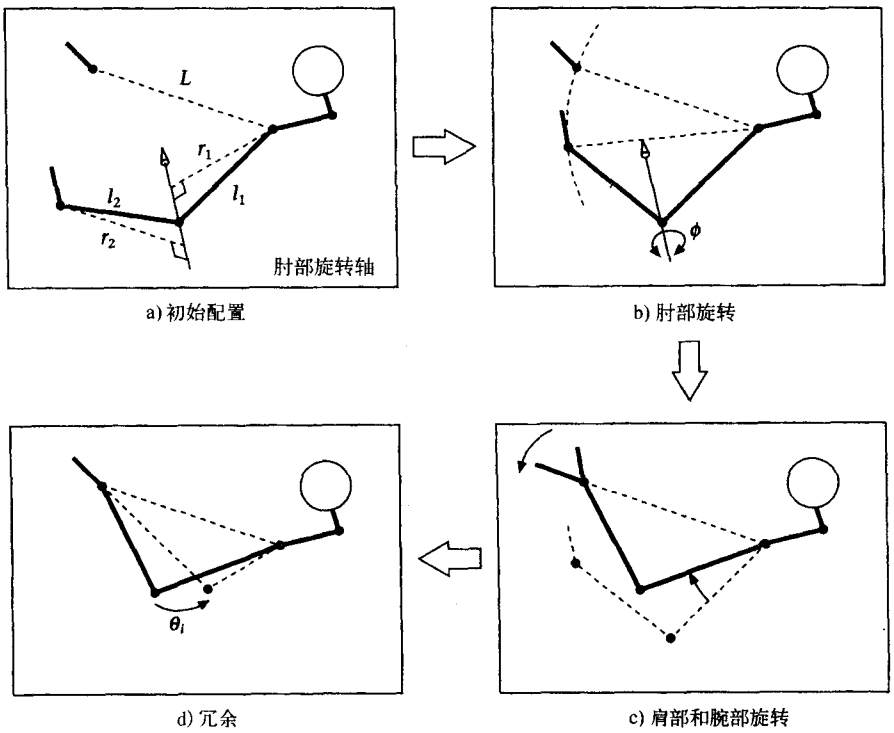


图 11-13　调整手臂姿势

效应器移动这样的位移。[⊖]如果有不止一个目标位置，那么可以使用一个加权平均值，但注意，这并非最好的解决办法。作者指出根的位置应当是可选的，这样移动根就可以使所有末端效应器都处于其可触及的范围内，然后计算新的姿势。

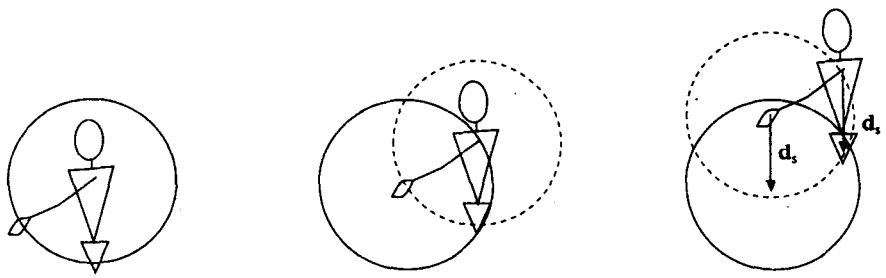


图 11-14　可达范围

为实现这一想法，他们使用了可达范围（reachable range）这个概念，相应于每一个末端效应器目标对应的根位置。图 11-14 描述了这个概念，并用 3 步得到了所谓的根球体。第一

⊖ 记住，该过程是动作适应，并且与角色的比例有关。我们不是在讨论所谓的远距离传输玩家，以至于他能成功地抓到离他伸直了的手还有两米的球。

个球——手的可达范围——是一个半径等于手臂展开长度而中心在肩关节的球体。第二个球描述的是肩部的范围，半径与前一个球相同，中心是一个给定的目标位置，肩部必须在这个球内部。如果肩关节在该球内的任何一个位置，伸直胳膊就可以够到目标。同样对于根也存在一个约束球体，通过从目标位置开始平移肩关节球，位移为向量 $\mathbf{d}_s$（$\mathbf{d}_s$ 为从肩部到根的向量）来得到。

现在，因为角色有 4 个末端效应器，就相应地有 4 个根部约束球，这些球的交集就是能够同时够到 4 个目标的位置集合。计算根位置的最佳位移就成了一个解析几何问题，所有的细节见［SHIN01］。但当所有这些球体不相交时，就要保留与最重要的末端效应器相对应的球，而舍弃其余三个（见 10.8.2 节）。

当根位置估计不能使所有的末端效应器都能够到它们的目标时，就需要第二个阶段来调整身体姿势。身体姿势由根位置、骨盆方向以及上身姿势构成。为此，定义目标函数：

$$E = E_g + \alpha E_p$$

第一项与等式 11-2（在 11.3.3 节定义的目标函数）的第一项类似，不同之处是，现在有 4 个关节点，对应于末端效应器的方程：

$$E_g = \sum_{i=1}^{4} E_i$$

其中

$$E_i = \begin{cases} (\| \mathbf{x}_i^c - \mathbf{x}_i^g \| - l_i)^2 \\ 0, \text{if} \| \mathbf{x}_i^c - \mathbf{x}_i^g \| < l_i \end{cases}$$

其中 $\mathbf{x}_i^c$ 和 $\mathbf{x}_i^g$ 是末端效应器（肩/髋）的当前位置及其对应的目标位置。l_i 为对应的肢体展开长度。

因为此项工作是由 MoCap 适应驱动的，E_p 使得相对于捕捉的姿势发生的移动最小，由下式给出：

$$E_p = \sum_{j=1}^{n} \beta_j \| \ln(\mathbf{q}_j^{-1} \mathbf{q}_j^c) \|^2 + \gamma (\| \mathbf{p}_e - \mathbf{p}_e^c \|)^2$$

其中：

$\mathbf{q}_j^c$ 是捕捉的第 j 段的方向

$\mathbf{p}_e^c$ 是根的估计位置

β 和 γ 是权重因子

上述表达式是两个距离平方的和。第一部分：

$$\ln(\mathbf{q}_j^{-1} \mathbf{q}_j^c)$$

是两个四元数间的测地距离或 slerp 距离（见附录 7.1），而第二部分是欧几里得距离。E_p 的设立使新的身体姿势尽可能地与捕捉的姿势相接近。

假设现在肩膀的位置已经设定，最后一步就是解决手臂的问题。我们要调整肩关节以便把腕部固定在目标位置。这会修改肘部旋转角度 θ。然后，重新调整该角度以使手臂姿势尽可能少地偏离捕捉的姿势。Shin 等人找到了该步骤的分析解法（与上一节中使用最优化解法的方法形成对比）。

回忆一下，两个单位四元数之间的夹角由下式决定（见附录 7.1）：

$$\mathbf{q}\cdot\mathbf{q}' = \cos\theta$$

肘部旋转的最小值是：

$$\text{minimum\_of}(\mathbf{q}\cdot\mathbf{q}^c)$$

其中：

$\mathbf{q}^c$ 是单位四元数，它表示肘部在捕捉的姿势上的旋转。

$\mathbf{q}$ 是待定四元数。

具体的公式推导如下。鉴于 $\mathbf{q}$ 与 $-\mathbf{q}$ 的等价性（两者代表相同的旋转），写成：

$$\theta = \text{minimum\_of}(\cos^{-1}(\mathbf{q}^c\cdot\mathbf{q})),\cos^{-1}(-\mathbf{q}^c\cdot\mathbf{q}))$$

当$|\mathbf{q}^c\cdot\mathbf{q}|$最小时，该式达到最大值。设 $\mathbf{q}$ 是在肘环上从参考四元数 $\mathbf{q}_0$ 旋转的角度 θ，有：

$$\mathbf{q} = \left(\cos\frac{\theta}{2},\ \mathbf{n}\sin\frac{\theta}{2}\mathbf{q}_0\right)$$

其中 $\mathbf{n}$ 是在肩部到腕部连线上的单位向量（见图 11-14）。

并且有：

$$\mathbf{q}^c = (w^c,\mathbf{v}^c)\quad \mathbf{q}_0 = (w_0,\mathbf{v}_0)$$

这样：

$$\begin{aligned}|\mathbf{q}^c\cdot\mathbf{q}| &= \left|(w^c,\mathbf{v}^c)\cdot\left(\cos\frac{\theta}{2},\mathbf{n}\sin\frac{\theta}{2}\right)(w_0,\mathbf{v}_c)\right| \\ &= \left|a\cos\frac{\theta}{2} + b\sin\frac{\theta}{2}\right| \\ &= (a^2+b^2)^{\frac{1}{2}}\left|\sin\frac{\theta}{2}+\alpha\right|\end{aligned}$$

其中：

$$\alpha = \tan^{-1}\frac{a}{b}$$

$$a = w^c w_0 + \mathbf{v}^c\cdot\mathbf{v}_0$$

$$b = w_0\mathbf{n}\cdot\mathbf{v}^c - w^c\mathbf{n}\cdot\mathbf{v}_0 + \mathbf{v}^c\cdot(\mathbf{n}\times\mathbf{v}_0)$$

所以$|\mathbf{q}^c\cdot\mathbf{q}|$在 $\mathbf{q}(-2\alpha+\pi)$ 或 $\mathbf{q}(-2\alpha-\pi)$ 时达到最大，并且当 $\mathbf{q}^c\cdot\mathbf{q}(-2\alpha+\pi) > \mathbf{q}^c\cdot\mathbf{q}(-2\alpha-\pi)$ 时 $\mathbf{q}(-2\alpha+\pi)$ 离 $\mathbf{q}^c$ 最近，否则 $\mathbf{q}(-2\alpha-\pi)$ 离 $\mathbf{q}^c$ 最近。

Tolani 等人［TOLI00］采用一个类似的方法，并且得出了一个可用于包含关节限制的肢体链的闭式解法。这种方法推论出一个关节角度及其导数的公式，作为肘部旋转角度 θ_i 的函数。在这种情况下，他们的算法使用额外的肘环自由度来避免关节限制，或者将肘部放置到尽可能靠近期望位置的地方。如果目标位置是可达的，该方法会找到一个解，否则还要再使用非约束最优化法和分析法。

11.4.3　防止自碰撞

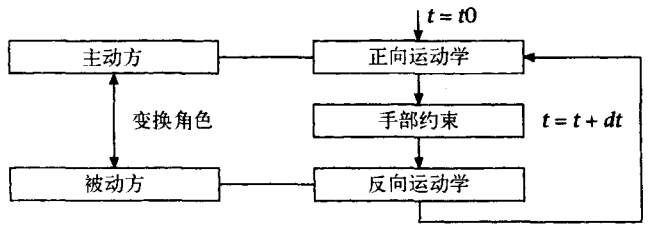
在此还要补充控制臂链中的一个问题，即自碰撞的防止。例如，把手“放置”到某个位置可能会导致肘部“进入”躯干。手臂的 IK 求解过程不涉及身体的躯干部分，因此自碰撞情况很容易发生。Huang 在［HUAN96］中建议使用启发式方法作为次要任务。这就要用到虚拟传感器，该设备常被称为实现抓取物体动作的探试程序通过使用标准的抓取模式包围物体，直至手指接触物体，手抓取动作可以与不同形状和比例的物体的交互相适应。附着球的

功能类似于虚拟传感器，使得这些成为可能。随着抓取动作的进行，对这些球和物体进行碰撞检测，从而控制手指的合拢动作。

在考虑自碰撞的情况下，Huang 在肘部、腕部、膝部、踝部都使用了球。检测方式是：

```
在 IK 解中：
for 每一个姿势 θ
    for 每一个球体碰撞
        在球和身体之间进行检测
            if 检测到碰撞 then 设置次要任务
                otherwise 接受当前姿势
```

例如，如果肘球与躯干发生碰撞，就要把 IK 应用到肩-肘-腕构成的链中，使肘部在反映冗余自由度的肘环中移动。依据排斥向量（与碰撞方向相反的向量）计算 $\Delta \mathbf{Z}$ 来设置次要任务。$\Delta \mathbf{Z}$ 是关节空间向量，它是从末端效应器空间的位移率 $\Delta \mathbf{d}$ 计算出来的。这只需要把肩部和肘部考虑成 IK 链（以肘部作为末端效应器），并计算：



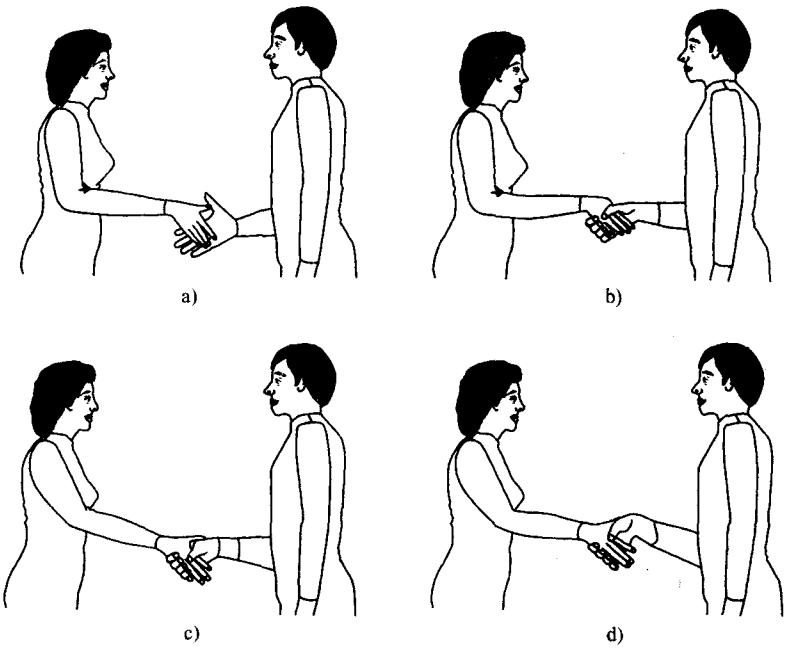


图 11-15　两个代理以握手的方式交互。在图 a 和 b 中，应用了抓的动作。在图 c 和 d 中，女表演者主动地移动（属正向运动学过程），而男表演者则配合她的动作，这是反向运动学（见［HUAN96］的论述）

$$\Delta \mathbf{Z} = \mathbf{J}^{+} \Delta \mathbf{d}$$

注意，这种简单的方案并不是在所有的情况下都有效。前臂仍然有可能“进入”躯干，即使肘球与腕球都没有与身体发生碰撞。

11.4.4　IK 与运动目标

在很多应用中，末端效应器的目标是运动的。例如，角色的目光被一个 IK 解所控制，而该 IK 解的输入为角色所感兴趣的移动物体的位置。物体本身会运动，并且反向运动学为支持两个动画对象之间的交互会设计一个方案。为构成场景，IK 技术所要做的是在观察者头部到所观察的物体间附上一个虚链。物体上的观察点就成了 IK 链中的末端效应器，而此 IK 链则包含了虚链和观察者的头部。

正向运动学和反向运动学的关系（见图 11-1）在交互代理的例子中得以巧妙地体现，这个例子是 Huang 在［HUAN96］中描述的，而且在涉及自主代理的场景中可能会发生。在例子中，两个代理用握手的办法实现“交互”。球传感器获得握手动作的启发式控制，握手之后，我们认为一个代理是主动的，由正向运动学驱动，而另一个则是被动的，由反向运动学驱动。被动代理的手按 IK 理论适应由于主动代理的握手动作而形成的运动目标（见图 11-15）。

参考文献

[ASHI00] Ashikhmin, M., Premoze, S. and Shirley, P. (2000) A microfacet-based BRDF generator. *Proc. SIGGRAPH 2000*, pp. 65–74

[BADL99] Badler, N., Palmer, M. and Bindiganavale, R. (1999) Animation control for real-time virtual humans. *Communications of the ACM*, **42** (8), August, 64–73

[BANK94] Banks, D.C. (1994) Illumination for diverse codimensions. *Proc. SIGGRAPH 94*, pp. 327–34

[BARE94] Barequet, G. and Sharir, M. (1994) Piecewise linear interpolation between polygonal slices. *Proc. 10th Annual ACM Symposium on Computational Geometry*, pp. 93–102

[BIND00] Bindiganavale, R., Schuler, W., Allbeck, J., Badler, N., Joshi, A. and Palme, M. (2000) Dynamically altering agent behaviour using natural language instructions. *Autonomous Agents*, June, 293–300. http://www.cis.upenn.edu/~hms/publications.html

[BLIN76] Blinn, J.F. and Newell, M.E. (1976) Texture and reflection in computer generated images. *Comm. ACM*, **19** (10), 362–7

[BLIN77] Blinn, J.F. (1977) Models of light reflection for computer synthesised pictures. *Computer Graphics*, **11** (2), 192–8

[BLIN78] Blinn, J.F. (1978) Simulation of wrinkled surfaces. *Computer Graphics*, **12** (3), 286–92

[BLYT00] Blythe, D. (2000) Programming with OpenGL advanced rendering. *SIGGRAPH 2000 Course Notes*

[BRACE65] Bracewell, R.N. (1965) The Fourier Transform and its Applications. New York: McGraw-Hill Book Company

[BROO89] Brooks, R. (1989) A robust layered control for a mobile robot. *IEEE Journal of Robotics and Automation*, **2** (1), 14–23

[BRUD95] Bruderlin, A. and Williams, L. (1995) Motion signal processing. *Proc. SIGGRAPH 95*, pp. 97–104

[CABR87] Cabral, B., Max, N. and Springmeyer, R. (1987) Bidirectional reflection functions from surface bump maps. *Proc. SIGGRAPH 87*

[CASS00] Cassell, J. (2000) More than just another pretty face: Embodied conversational interface agents. *Communications of the ACM*, **43** (4), 70–8

[CASS01] Cassell, J., Nakano, Y., Bickmore, T., Sidner, C. and Rich, C. (2001) Non-verbal cues for discourse structure. Association for Computational Linguistics Joint EACL – 2001 ACL Conference

[CAVA99] Cavazza, M. and Palmer, I.J. (1999) Natrual language control of interactive 3D animation and computer games. *Virtual Reality*, **4**, 85–102

[CIGN98] Cignoni, P., Montani, C., Rocchinin, C. and Scopigno, R. (1998) A general method for preserving attribute values on simplified meshes. *IEEE Visualisation 98 conf. proc*, pp. 59–66

[COHE88] Cohen, M.F., Chen, S.E., Wallace, J.R. and Greenberg, D.P. (1988) A Progressive Refinement Approach to Fast Radiosity Image Generation, *Computer Graphics*, 22 (4) *Proc. SIGGRAPH 88*, pp. 75–84

[COHE93] Cohen, M.M. and Massaro, D.W. (1993) Modeling coarticulation in synthetic visual speech. In N.M. Thalmann and D. Thalmann (eds), *Models and Techniques in Computer Animation*. Tokyo: Springer-Verlag, pp. 139–56. http://mambo.ucsc.edu/psl/pslfan.html

[COHE96] Cohen, J., Varshney, A., Manocha, D., Turk, G., Weber, H., Agarwal, P., Brooks, F. and Wright, W. (1996) Simplification envelopes. *Proc. SIGGRAPH 96*, pp. 119–28

[COHE98] Cohen, J., Olano, M. and Manocha, D. (1998) Appearance-preserving simplification. *Proc. SIGGRAPH 98*, pp. 115–22, Orlando, Florida, July, 19–24

[COOK82] Cook, R.L. and Torrance, K.E. (1982) A reflectance model for computer graphics. *Computer Graphics*, 15 (3), 307–16

[COOK84] Cook, R.L. (1984) Shade trees. *Proc. SIGGRAPH 84*, pp. 137–44

[COOK87] Cook, R.L., Carpenter, L. and Catmull, E. (1987) The REYES image rendering architecture. *Proc. SIGGRAPH 1987*, pp. 95–102

[COQU90] Coquillart, S. (1990) Extended free form deformation. *Proc. SIGGRAPH 90*, pp. 187–96

[CRAI88] Craig, J. (1988) *Introduction to Robotics Mechanics and Control*. Reading, MA: Addison-Wesley

[DANA99] Dana, K.J., Ginneken, V., Nayar, S.K. and Koenderink, J.J. (1999) Reflectance and texture of real-world surfaces. *ACM Transactions on Graphics*, 18 (1), January, 1–34

[DEER95] Deering, M. (1995) Geometry compression. *Proc. SIGGRAPH 95*, pp. 13–20

[DESB00] Desbrun, M., Meyer, M. and Shroder, P. (2000) *Differential geometry operaors in nD*. Multiresolution Modelling Group, California Inst. of Technology, pre-print

[DOCA76] Do Carmo, P. (1976) *Differential geometry of curves and surfaces*. Upper Saddle River, NJ: Prentice-Hall Inc.

[DUCH97] Duchaineau, M.A., Wolinsky, M., Sigeti, D.E., Miller, M.C., Aldrich, C. and Mineev-Weinstein, M.B. (1997) ROAMing terrain: Real-time optimally adapting meshes. *Proc. IEEE Visualization 97*, October, 81–8

[DUFF86] Duff, T. (1986) Splines in animation and modelling. *SIGGRAPH Course Notes*, 15

[EAST01] Eastlick, M. and S. Maddock, S. (2001) Triangle-mesh simplification using error polyhedra. *Proc. 19th Eurographics UK Chapter Annual Conference*, UCL, 3–5 April, pp. 1–10

[ECK95] Eck, M., DeRose, T., Duchamp, T., Hoppe, H., Lounsbery, M. and Stuetzle, W. (1995) Multiresolution analysis of arbitrary meshes. *Proc. SIGGRAPH 95*, pp. 173–82

[EDGE01] Edge, J. and Maddock, S. (2001) Expressive visual speech using geometric muscle functions. *Proc. Eurographics UK 2001*. http://www.dcs.shef.ac.uk/~jedge/

[EKMA73] Ekman, P. (1973) *Darwin and Facial Expressions*. New York: Academic Press

[FORS88] Forsey, D. and Bartels, R. (1988) Hierarchical B-spline refinement. *Proc. SIGGRAPH '88*, pp. 205–12.
http://www.cs.ubc.ca/nest/imager/contributions/forsey/dragon/facial.html

[FUNG99] Funge, J. (1999) *AI for Games and Animation: A modelling Approach.* AK Peters

[GARL97] Garland, M. and Heckbert, P. (1997) Surface simplification using quadric error metrics. *Proc. SIGGRAPH 97*, pp. 209–16

[GARL98] Garland, M. and Heckbert, P. (1998) Simplifying surfaces with colour and texture using quadric error metrics. *IEEE Visualisation 98 conf. proc*, pp. 263–9

[GARL99] Garland, M. (1999) Quadric based polygonal surface simplification. Ph.D. thesis, Tech. Rept. CMU-CS-99-105

[GLEI98] Gleicher, M. (1998) Retargetting motion to new characters. *Proc. SIGGRAPH 98*, pp. 33–42

[GLEI01] Gleicher, M. (2001) Comparative analysis of constraint based motion editing methods. *2000 Workshop on Human Modelling and Analysis*, Seoul, Korea

[GORA84] Goral, C., Torrance, K.E. and Greenberg, D. (1984) Modeling the interaction of light between diffuse surfaces, *Computer Graphics (ACM SIGGRAPH) Annual Conference Series*, **18** (3), pp. 213–22, Addison-Wesley

[GUIB85] Guibas, L. and Stolfi, J. (1985) Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. on Graphics*, **4** (2), 74–123

[HANR90] Hanrahan, P. and Lawson, J. (1990) A language for shading and lighting calulations. *Proc. SIGGRAPH 90*, pp. 289–98

[HANR93] Hanrahan, P. and Kreuger, W. (1993) Reflections from layered surfaces due to sub-surface scattering. *Proc. SIGGRAPH 91*, pp. 197–206

[HEID98] Heidrich, W. and Seidel, H.-P. (1998) View-independent environment maps. *Proc. 1998 Eurographics/SIGGRAPH Workshop on Graphics Hardware*, Lisbon, August

[HEID99] Heidrich, W. and Siedel, H. (1999) Realistic hardware-accelerated shading and lighting. *Proc. SIGGRAPH 99*

[HEID00] Heidrich, W. (2000) Environment maps and their applications. *SIGGRAPH 2000 Course Notes: Approaches for Procedural Shading on Graphics Hardware*

[HOPP96] Hoppe, H. (1996) Progressive meshes. *Proc. SIGGRAPH 1996*, pp. 99–108

[HOPP97] Hoppe, H. (1997) View-dependent refinement of progressive meshes. *Proc. SIGGRAPH 1997*, pp. 189–98

[HOPP99] Hoppe, H. (1999) New quadric metric for simplifying meshes with appearance attributes. *IEEE Visualization 1999*, October 1999, pp. 59–66

[HUAN96] Huang, Z. (1996) *Motion Control for Human Animation.* Thesis No. 1601, Swiss Federal Institute of Technology, Lausanne

[KASS87] Kass, M., Witkin, A. and Terzopoulos, D. (1987) Snakes: Active contour models. *International Journal of Computer Vision*, **1** (4), 321–31

[KAUT99] Kautz, J. and McCool, M.D. (1999) Interactive rendering with arbitrary BRDFs using separable approximations. *Proceedings of the 10th Eurographics Workshop on Rendering*, June, pp. 281–92

[KILG99] Kilgard, M.J. (1999) A practical and robust bump-mapping technique for today's GPUs. GDC 2000 and www.nvidia.com

[KOBB98] Kobbelt, L., Campagna, S., Vorsatz, J. and Seidel, H. (1998) Interactive multiresolution modelling on arbitrary meshes. *Proc. SIGGRAPH 98*, pp. 105–14

[KURI91] Kurihara, T. and Arai, K. (1991) A transformation method for modelling and animation of the human face from photographs. In N.M. Thalmann (ed.), *Computer Animation '91*, Tokyo: Springer-Verlag

[LASS87] Lasseter, J. (1987) Principles of traditional animation applied to 3D computer animation. *Proc. SIGGRAPH 87*, pp. 35–44

[LEE98] Lee, A., Sweldens, W., Schroder, P., Cowsar, L. and Dobkin, D. (1998) MAPS: Multiresolution Adaptive Parameterisation of Surfaces. *Proc. SIGGRAPH 98*, pp. 95–104

[LEE99] Lee, J. and Shin, S.Y. (1999) A hierarchical approach to interactive motion editing for human-like figures. *Proc. SIGGRAPH '99*, pp. 39–48

[LEE00] Lee, A. (2000) Building your own subdivision surfaces. www.gamasutra.com/features

[LEE00] Lee, A., Moreton, H. and Hoppe, H. (2000) Displaced subdivision surfaces. *Proc. SIGGRAPH 2000*, pp. 85–94

[LEE02] Lee, J. and Shin, S.Y. (2002) General construction of time domain filters for orientation data. *IEEE Trans. On Visualization and Computer Graphics*, **8** (2), 119–28

[LEWI00] Lewis, J.P., Cordner, M. and Fong, N. (2000) Pose space deformation: A unified approach to shape interpolation and skeleton-driven animation. *Proc. SIGGRAPH 2000*, pp. 165–71

[LIND00] Lindstrom, P. and Turk, G. (2000) Image-driven simplification. *ACM Trans. on Graphics*, **19** (3), July, 204–41

[LIND01] Lindholm, E., Kilgard, M. and Moreton, H. (2001) A user-programmable vertex engine. *Proc. SIGGRAPH 2001*

[MACI90] Maciejewski, A. (1990) Dealing with the ill-conditioned equations of motion for articulated figures. *IEEE Computer Graphics and Applications*, May, 63–71

[MILL84] Miller, G.S. and Hoffman, C.R. (1984) Illumination and reflection maps: Simulated objects in simulated and real environments. *SIGGRAPH 84 Course Notes; Advanced Computer Graphics Animation* (long unavailable but see also http://www.debevec.org/ReflectionMapping/miller.html)

[PARK82] Parke, F.I. (1982) Parameterized model for facial animation. *IEEE Comp. Graphics and Applications*, **2** (9), November, 61–68

[PERL85] Perlin, K. (1985) An image synthesiser. *Proc. SIGGRAPH 85*, pp. 287–96

[PERL95] Perlin, K. (1995) Real-time responsive animation with personality. *IEEE Trans. on Visualization and Computer Graphics*, **1** (1), March, 5–15

[PERL96] Perlin, K. (1996) IMPROV: A system for scripting interactive actors in virtual worlds. *Proc. SIGGRAPH 96*

[PHIL91] Philips, C.B. and Badler, N.I. (1991) Interactive behaviour for bi-pedal articulated figures. *Proc. SIGGRAPH 91*, pp. 359–62

[PIGH98] Pighin, F., Hecker, J., Lischinski, D., Szeliski, R. and Salesin, D. (1998) Synthesizing realistic facial expressions from photographs. *Proc. SIGGRAPH 98*

[PIGH99] Pighin, F., Szeliski, R. and Salesin, D. (1999) Resynthesizing facial animation through 3D model-based tracking. *Proceedings of ICCV 99*

[PIXA88] The Pixar Corp. (1988) *The Renderman Interface v 3.0.* Pixar Corp., San Rafael, CA, May

[PRES01] Press, W.H., Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. (2001) *Numerical Recipes in C++.* Cambridge: Cambridge University Press

[PROU01] Proudfoot, K., Mark., W.R., Tzvetkov, S. and Hanrahan, P. (2001) A real-time procedural shading system for programmable graphics hardware. *Proc. SIGGRAPH 2001*

[RABI01] Rabin, S. (2001) A* Aesthetic optimisation. In Mark DeLoura (ed.), *Game Programming Gems.* Hingham, MA: Charles River Media, pp. 264–71

[REEV90] Reeves, W.T. (1990) Simple and complex facial animation: case studies. *AUSGRAPH 1990,* Melbourne, Australia

[REVE98] Reveret, L. and Benoit, C. (1998) A new 3D lip model for analysis and synthesis of lip motion in speech production. *Proc. Second ESCA Workshop on Audio-Visual Speech Processing, AVSP'98,* Terrigal, Australia, 4–6, December

[REVE00] Reveret, L., Bailly, G. and Badin, P. (2000) MOTHER: A new generation of talking heads providing a flexible articulatory control for video-realistic speech animation. *Proc. 6th Int. Conference of Spoken Language Processing, ICSLP'2000,* Beijing, China, 16–20 October

[REYN87] Reynolds, C. (1987) Flocks, herds and schools. *Proc. SIGGRAPH 87,* pp. 25–34

[ROSE98] Rose, C., Cohen, M.F. and Bodenheimer, B. (1998) Verbs and adverbs: Multidimensional motion interpolation. *IEEE Computer Graphics and Applications,* September, 32–40

[ROSS93] Rossignac, J. and Borrel, P. (1993) Multiresolution 3D approximation for rendering complex scenes. *Modelling in Computer Graphics: Methods and Applications,* pp. 455–65

[SAND00] Sander, P., Gu, X., Gortler, S., Hoppe, H. and Snyder, J. (2000) Silhouette clipping. *Proc. SIGGRAPH 2000,* pp. 327–34

[SEDE86] Sedeberg, T.W. and Parry, S.R. (1986) Free form deformation of solid geometric models. *Proc. SIGGRAPH 1986,* pp. 151–60

[SHIN01] Shin, H.J., Lee, J., Gleicher, M. and Shin, S.Y. (2001) Computer puppetry: An importance based approach. *ACM Trans. on Graphics,* **20** (2), 67–94

[SHOE87] Shoemake, K. (1987) Quaternion calculus and fast animation. *SIGGRAPH Course Notes,* 10

[SING98] Singh, K. and Fiume, E. (1998) Wires: A geometric deformation technique. *Proc. SIGGRAPH 1998,* pp. 405–14

[SING00] Singh, K. and Kokkevis, E. (2000) Skinning characters using surface oriented free-form deformations. *Proc. Graphics Interface 2000,* May, Montreal

[SLOA00] Sloan, P., Rose, C. and Cohen, M. (2000) *Shape and animation by example.* Microsoft Research Tech. Report, MSR-TR-2000-79. www.research.microsoft.com

[SLOA01] Sloan, P., Rose, C. and Cohen, M.F. (2001) Shape by example. *2001 Symposium on Interactive 3D Graphics,* March

[SUDA98] Sudarsky, S. and House, D. (1998) Motion capture data manipulation and re-use via B-splines. *CAPTECH '98,* Geneva, pp. 55–69, Published as: Lecture

Notes in AI 1537. Berlin: Springer

[TAO99] Tao, H. and Huang, T. (1999) Explanation-based facial motion tracking using a piecewise Bézier volume deformation model. *Proc. IEEE Comput. Vision and Patt. Recogn., CVPR '99*. http://www.ifp.uiuc.edu/~tao/HTML/content.html

[TERZ93] Terzopoulos, D. and Waters, K. (1993) Analysis and synthesis of facial image sequences using physical and anatomical models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 15 (6), June, 569–79. Special Issue on 3-D Modeling in Image Analysis and Synthesis. http://www.cs.toronto.edu/~dt

[TOLI00] Tolani, D., Goswami, A. and Badler, N. (2000) Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical Models*, 62 (5), 353–88

[UNUM95] Unuma, M., Anjyo, K. and Takeuchi, R. (1995) Fourier principles for emotion-based human figure animation. *Proc. SIGGRAPH 95*, pp. 91–6

[UPST89] Upstill, S. (1989) *The Renderman Companion*. Reading MA: Addison-Wesley

[WANG91] Wang, L. and Chen, C. (1991) A combined optimisation method for solving the inverse kinematics problem of mechanical manipulators. *IEEE Transactions on Robotics and Automation*, 7 (4), 489–99

[WATE87] Waters, K. (1987) A Muscle Model for Animating Three-Dimensional Faces. *Proc. SIGGRAPH '87*, July, pp. 17–24 http://www.crl.research.digital.com/projects/facial/facial.html

[WATT92] Watt, A. *Advanced Animation and Rendering Techniques*. Harlow: Addison-Wesley

[WATT00] Watt, A. *3D Computer Graphics*, 3rd edn. Harlow: Addison-Wesley

[WATT01] Watt, A. and Policarpo, F. (2001) *3D Games: Real-time Rendering and Software Technology*, Volume 1. Harlow: Addison-Wesley

[WEBE00] Weber, J. (2000) Run-time skin deformation. www.intel.com/ial/3dsoftware/inmdex.html

[WELM93] Welman, C. (1993) *Inverse Kinematics and Geometric Constraints for Articulated Figure Manipulation*. MSc. Thesis, Simon Fraser University, September

[WITK88] Wilkin, A. and Kass, H., Spacetime Constraints. *Proc. SIGGRAPH 88*, pp. 159–68

[WYNN00] Wynn, C. (2000) *Real-Time BRDF-based Lighting using Cube-Maps*. Technical Report, NVIDIA Corporation. www.nvidia.com