世界著名计算机教材精选

PEARSON Prentice Hall

计算机图形学

（OpenGL版）（第3版）

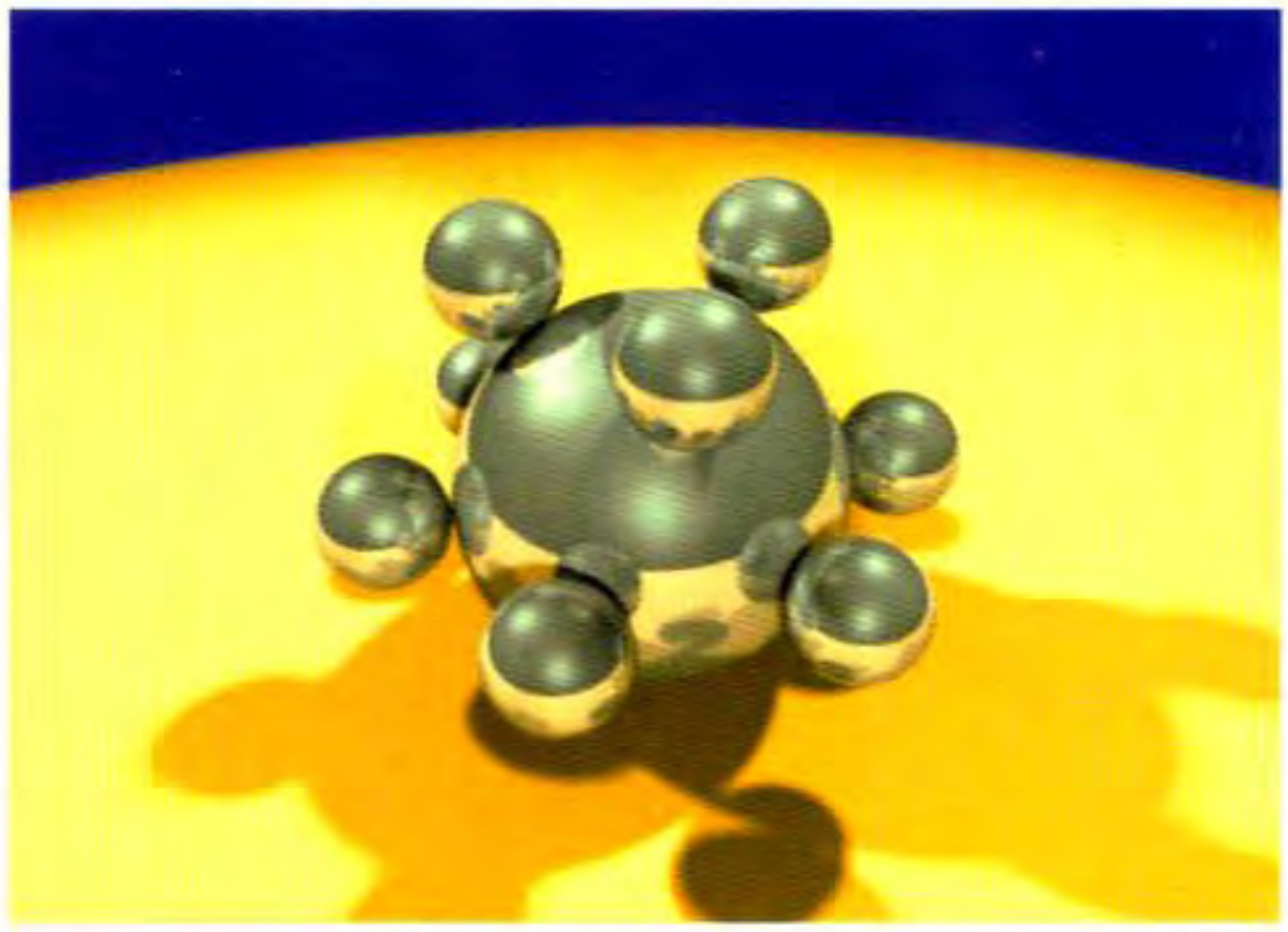
Francis S Hill, Jr.
Stephen M Kelley 著

胡事民 刘利刚
刘永进 张松海 等译

COMPUTER GRAPHICS USING OpenGL

Third Edition

清华大学出版社



彩图 1：包括反射和阴影的光线跟踪图像
（版权归 *Sven Maerivoet* 所有）

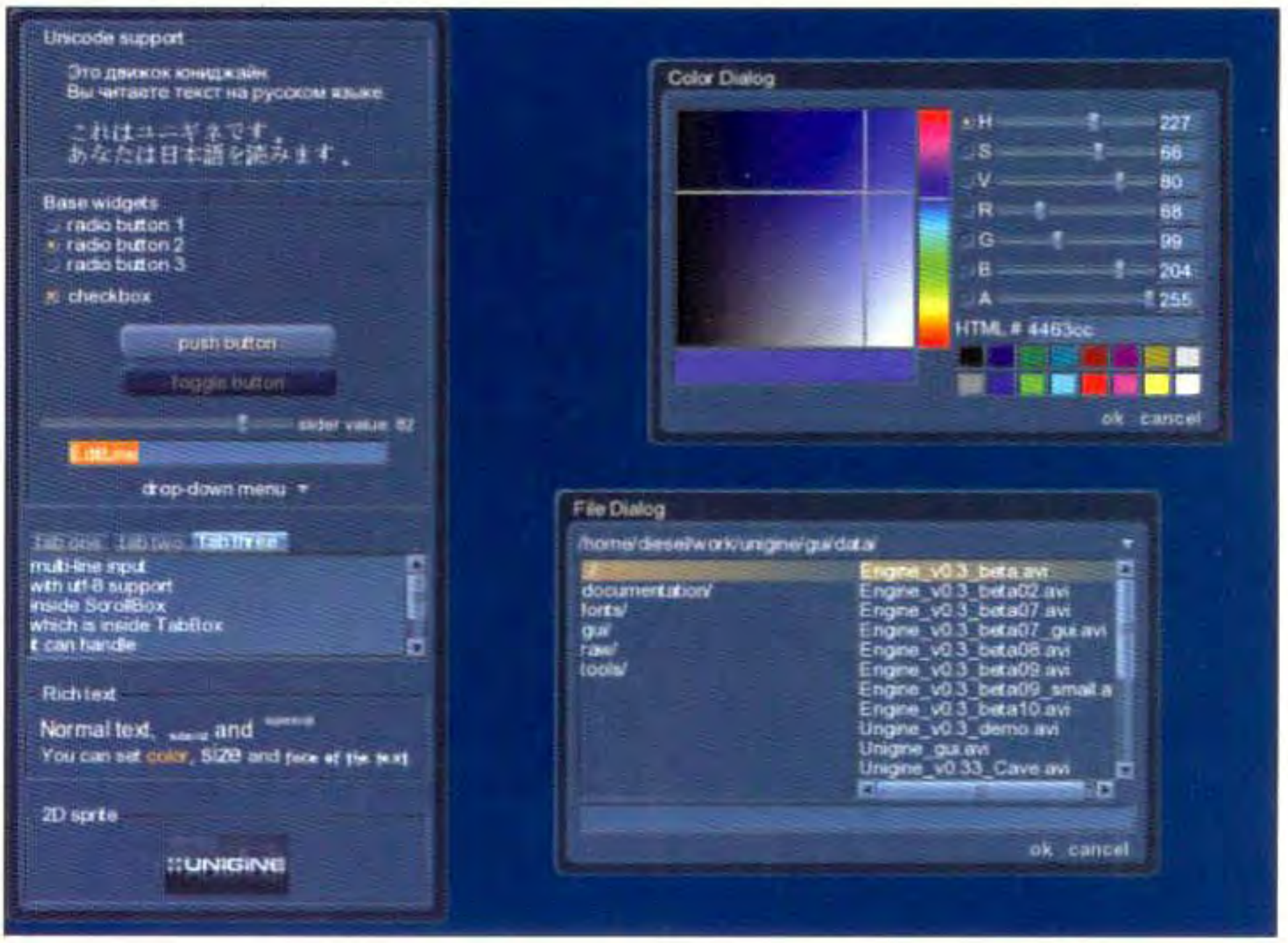
彩图 2：一个 3D 计算机游戏的截屏
（版权归 *High Moon Studios*，*Inc.* 所有）

彩图 3：一个带有绘图工具和菜单的绘图系统的截屏
（版权归 *Pixologic*，*Inc*. 所有）

彩图 4：火星地形的增强视图
（版权归 *NASA/NSSDC/Mary A. Dale-Bannister*，*Washington University in St. Louis* 所有）

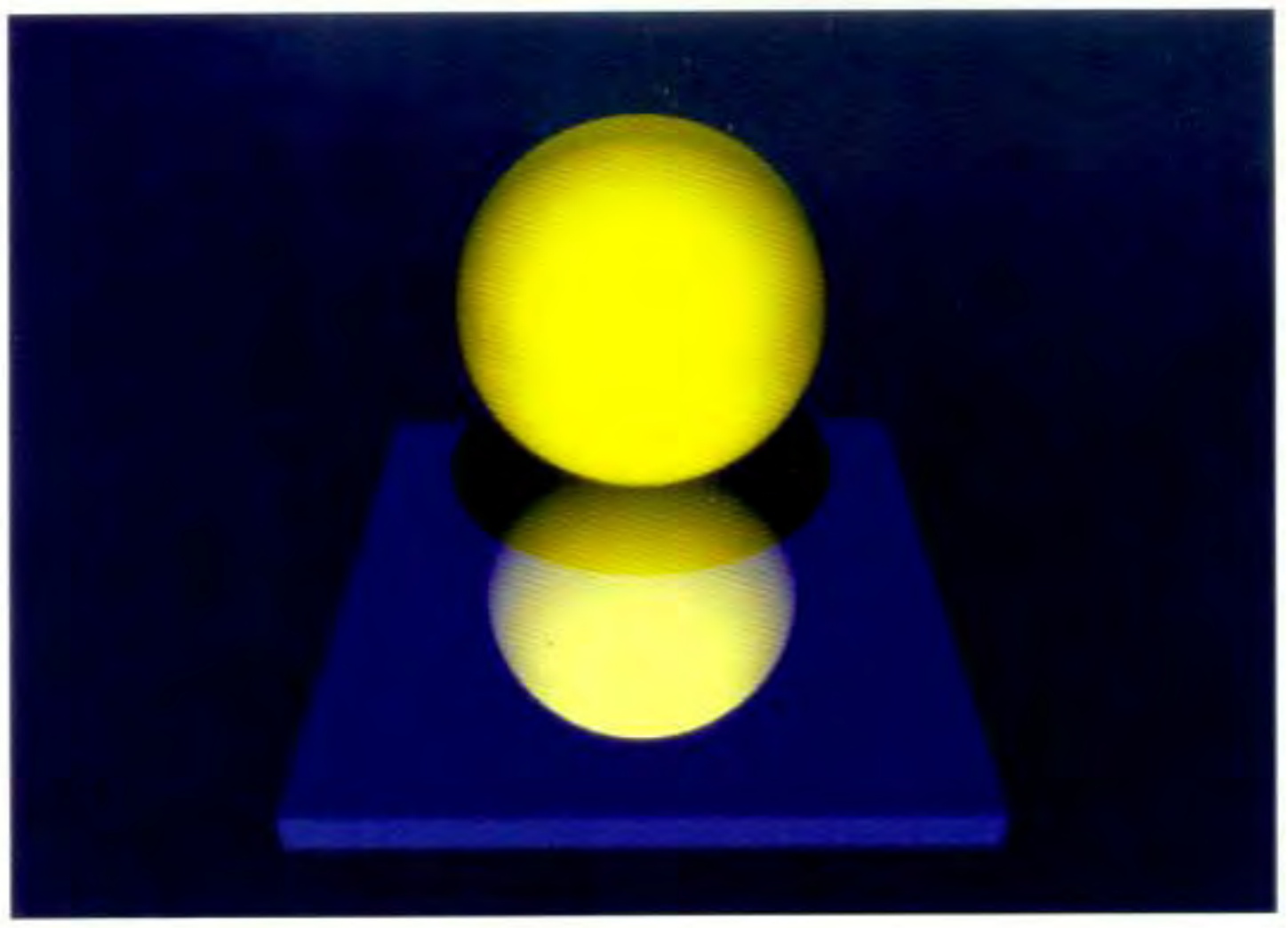
彩图 5：ATI 的视频卡

（版权归 *ATI* 所有）

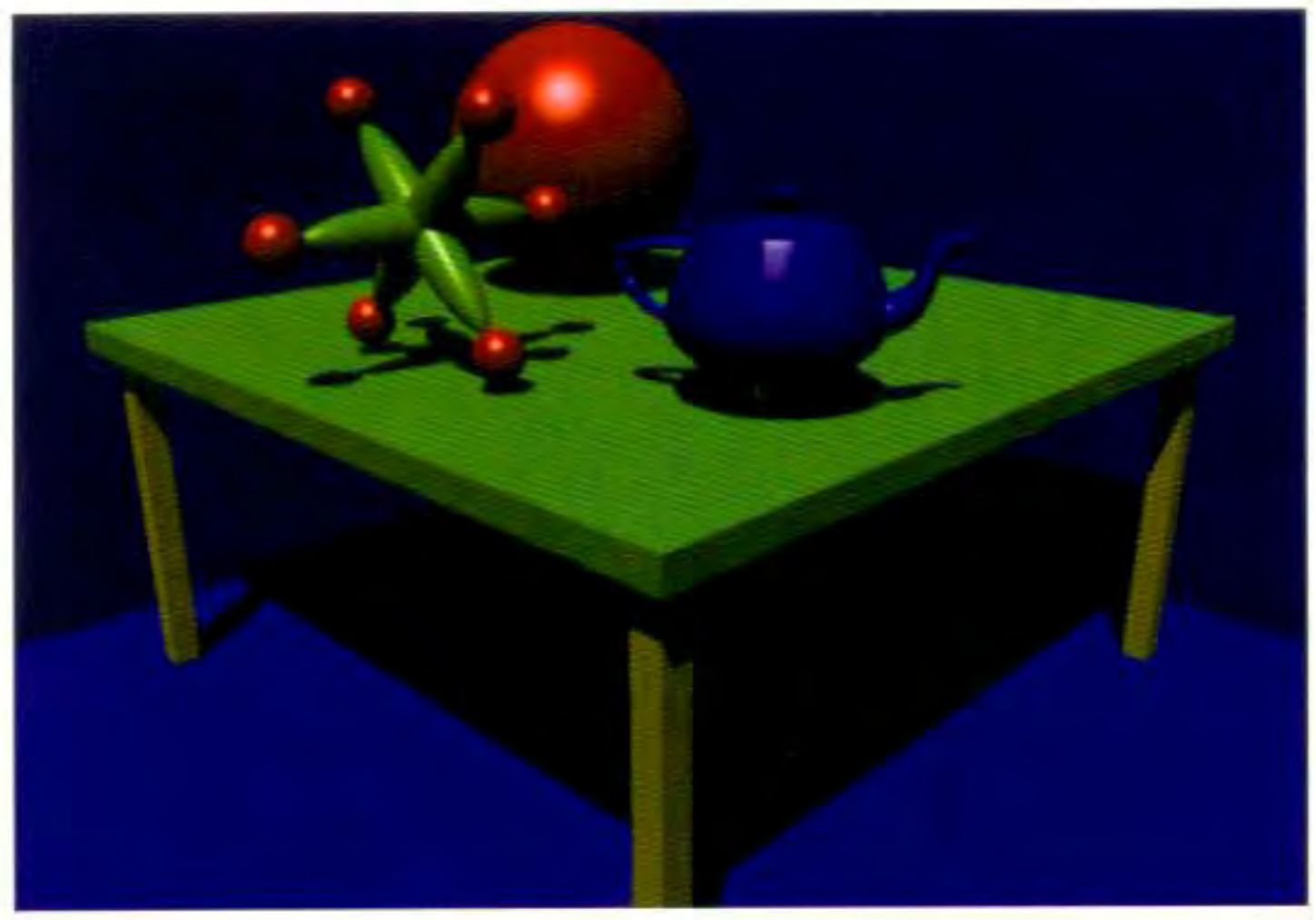


彩图 6：一个复杂的 OpenGL 菜单系统

（版权归 *Unigine* 所有）



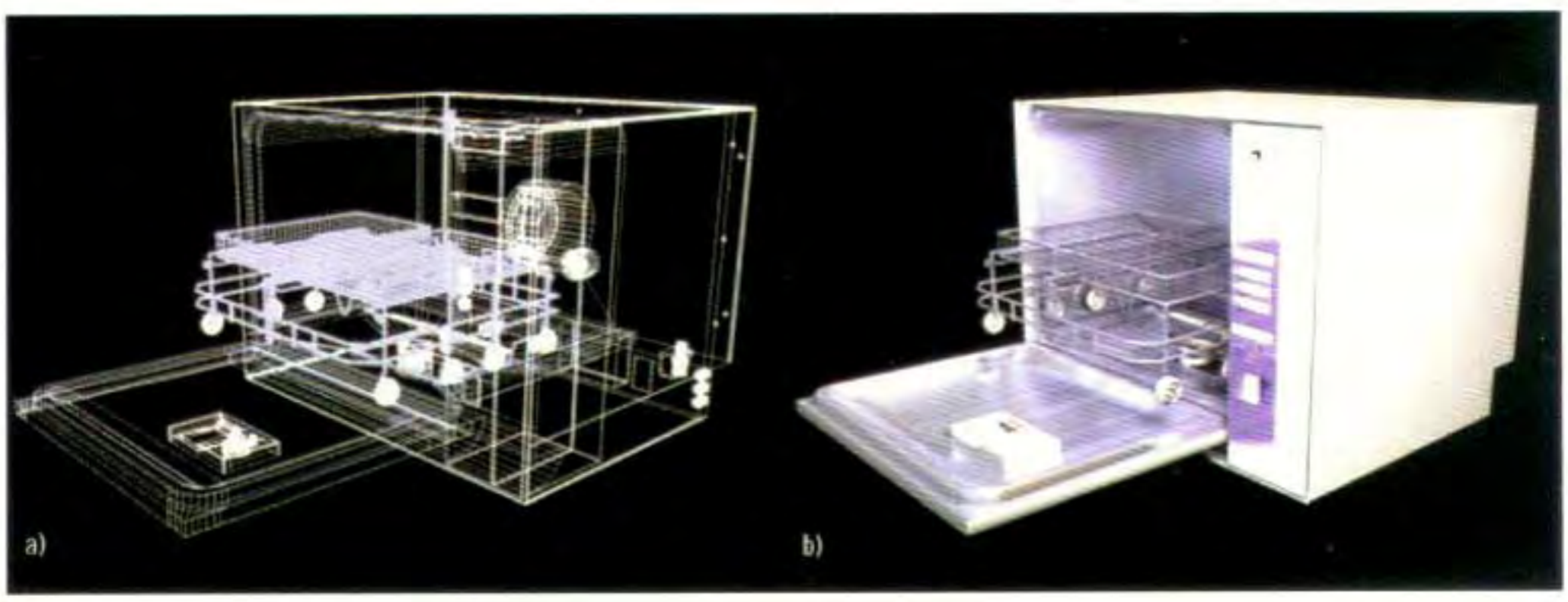
彩图 7：一个从有光泽表面反射的球
（版权归 *Sven Maerivoet* 所有）



彩图 8：一张桌子上放有经典的茶壶、球体和架子
（版权归 *Sven Maerivoet* 所有）

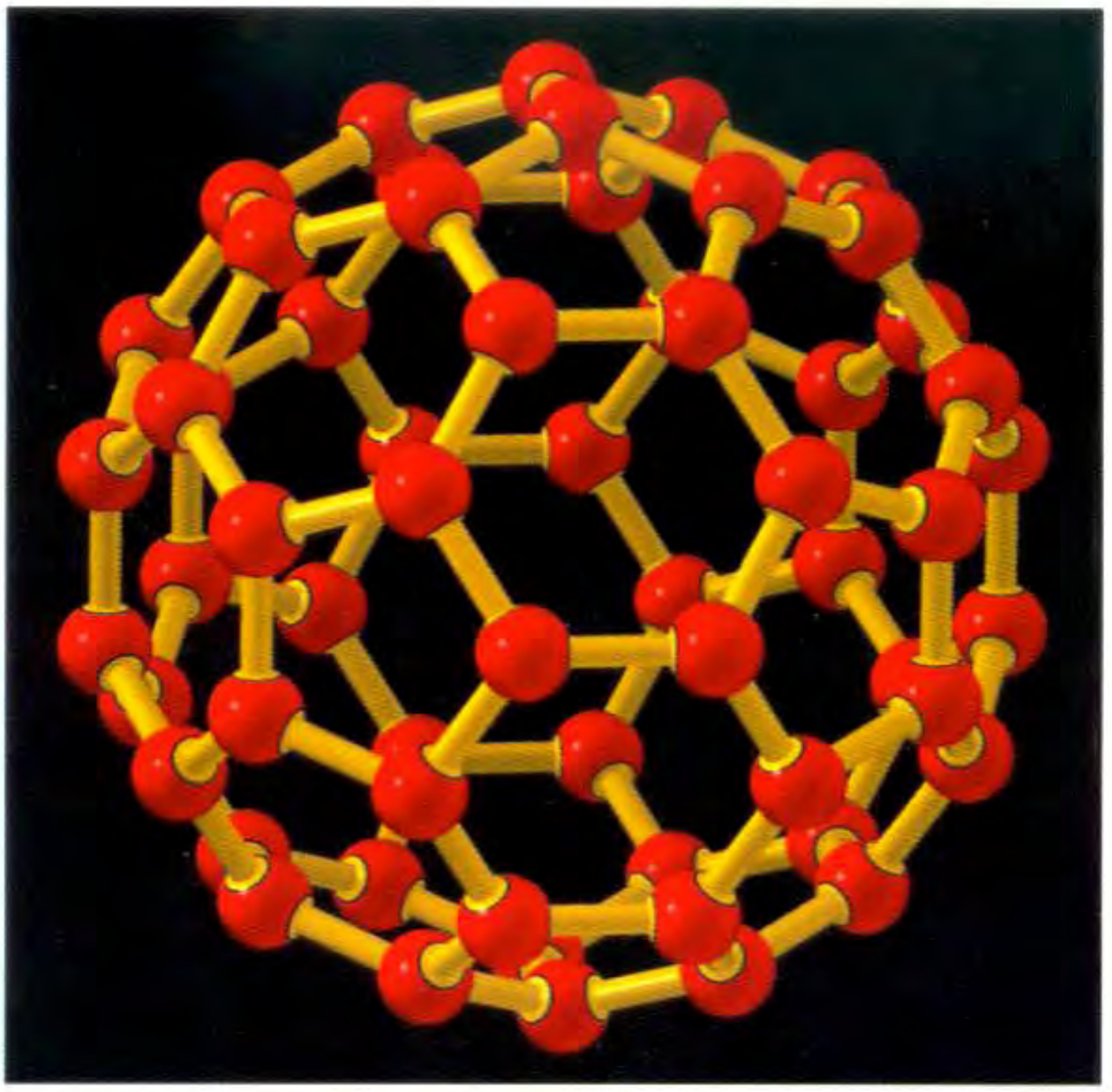
彩图 9：一个喷泉动画

（版权归 *Phillipp Crocoll* 所有），在线程序可用。



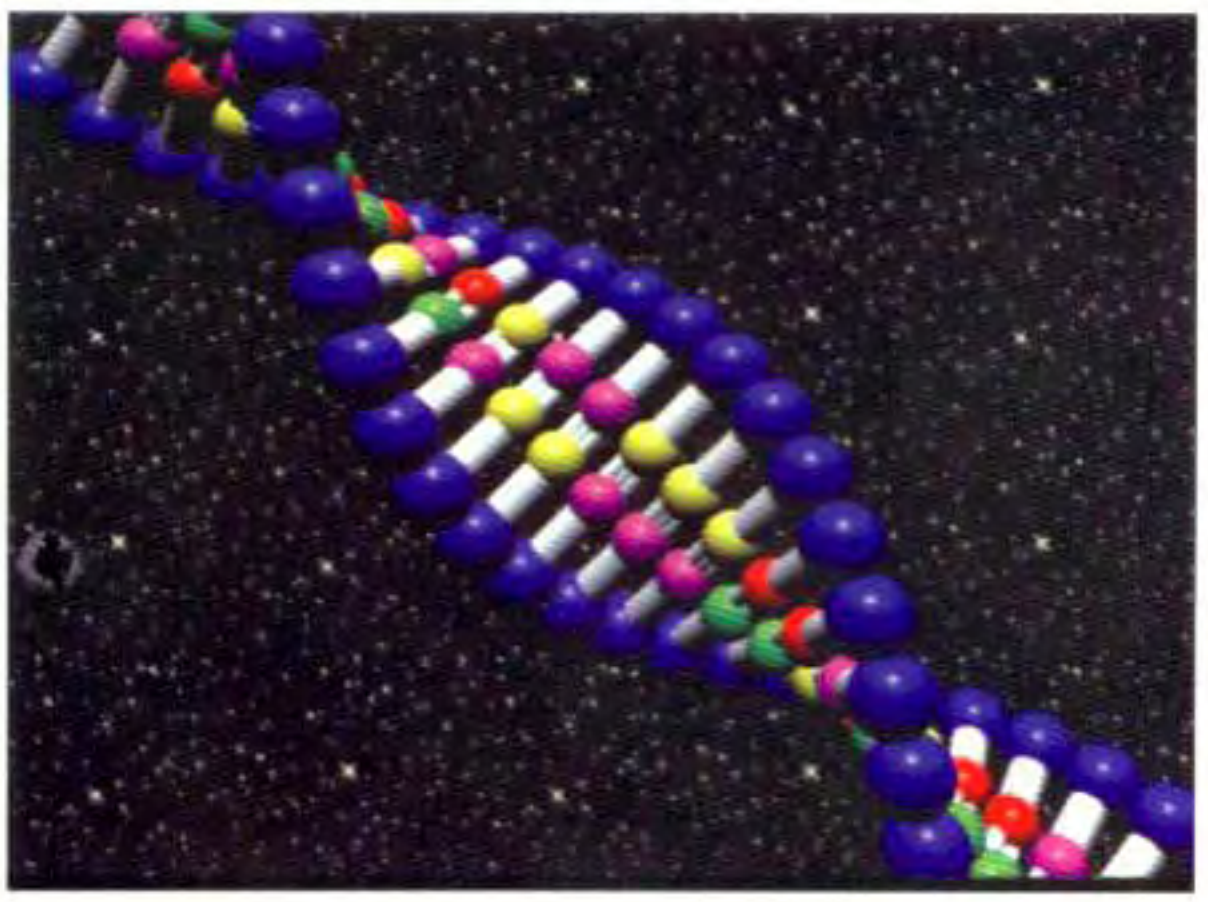
彩图 10：a）家具的线框模型；b）网格模型的渲染结果

（版权归 *Lamont Gilkey* 所有）



彩图 11：一个 Fullerine 分子结构视图

（版权归 *Dr. Koch/Lauher*，*SUNY* 所有）



彩图 12：双螺旋模型

（版权归 *Sven Maerivoet* 所有）

彩图 13：a）泰姬陵的照片；b）采用网格模型建模的泰姬陵的截屏

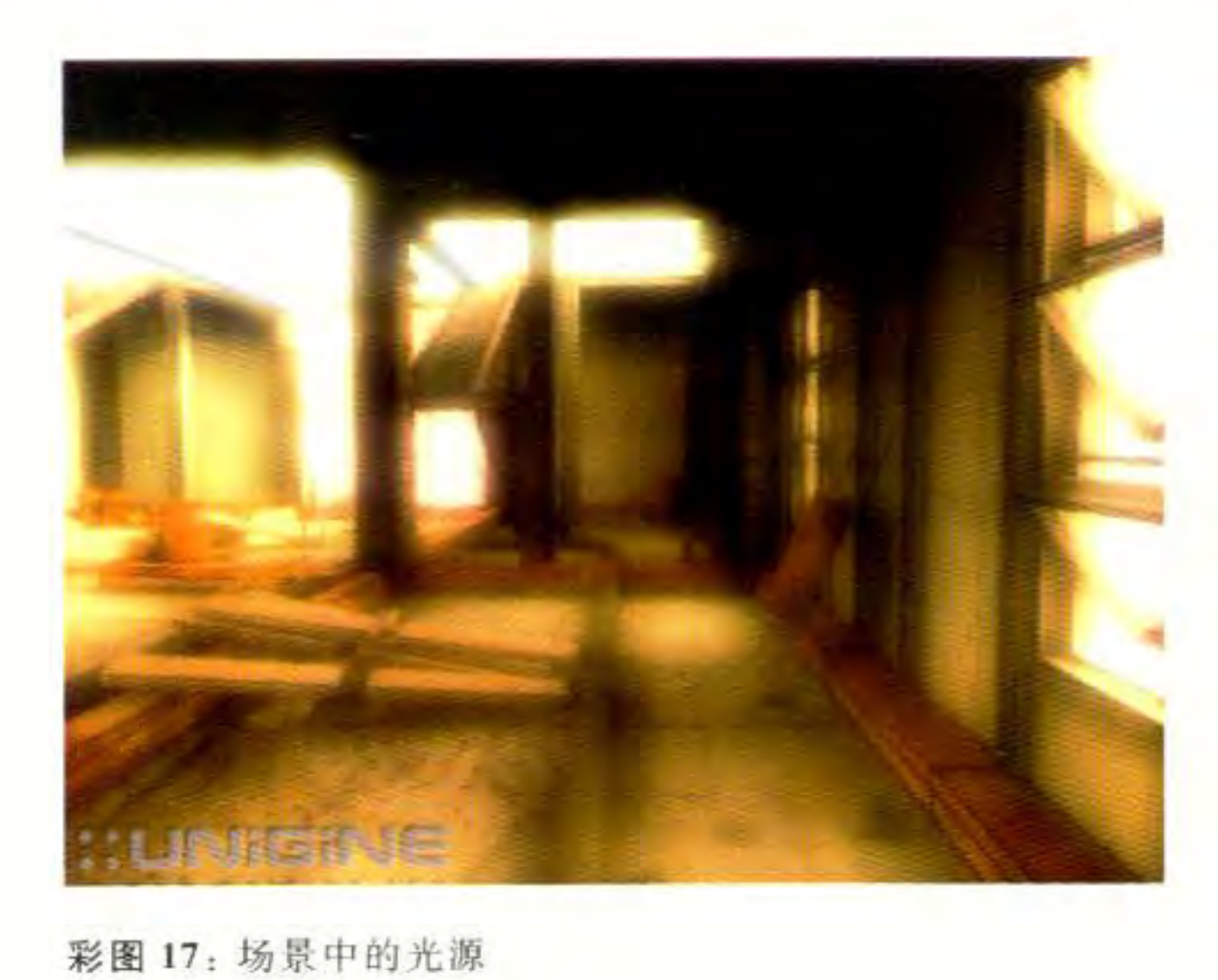
彩图 14：地形模拟图像（采用网格模型的山脉和粒子系统的云彩）
（版权归 *Rob Hall* 所有）

彩图 15：三点透视投影的建筑图

（版权归 *Andrew King*，*Architectural Illustrations*，*Inc.* 所有）

彩图 16：分别采用金、银、铜绘制模型的青蛙模型的三个渲染结果

（版权归 *Lutz Latta* 和 *Anto Matkovic* 所有）

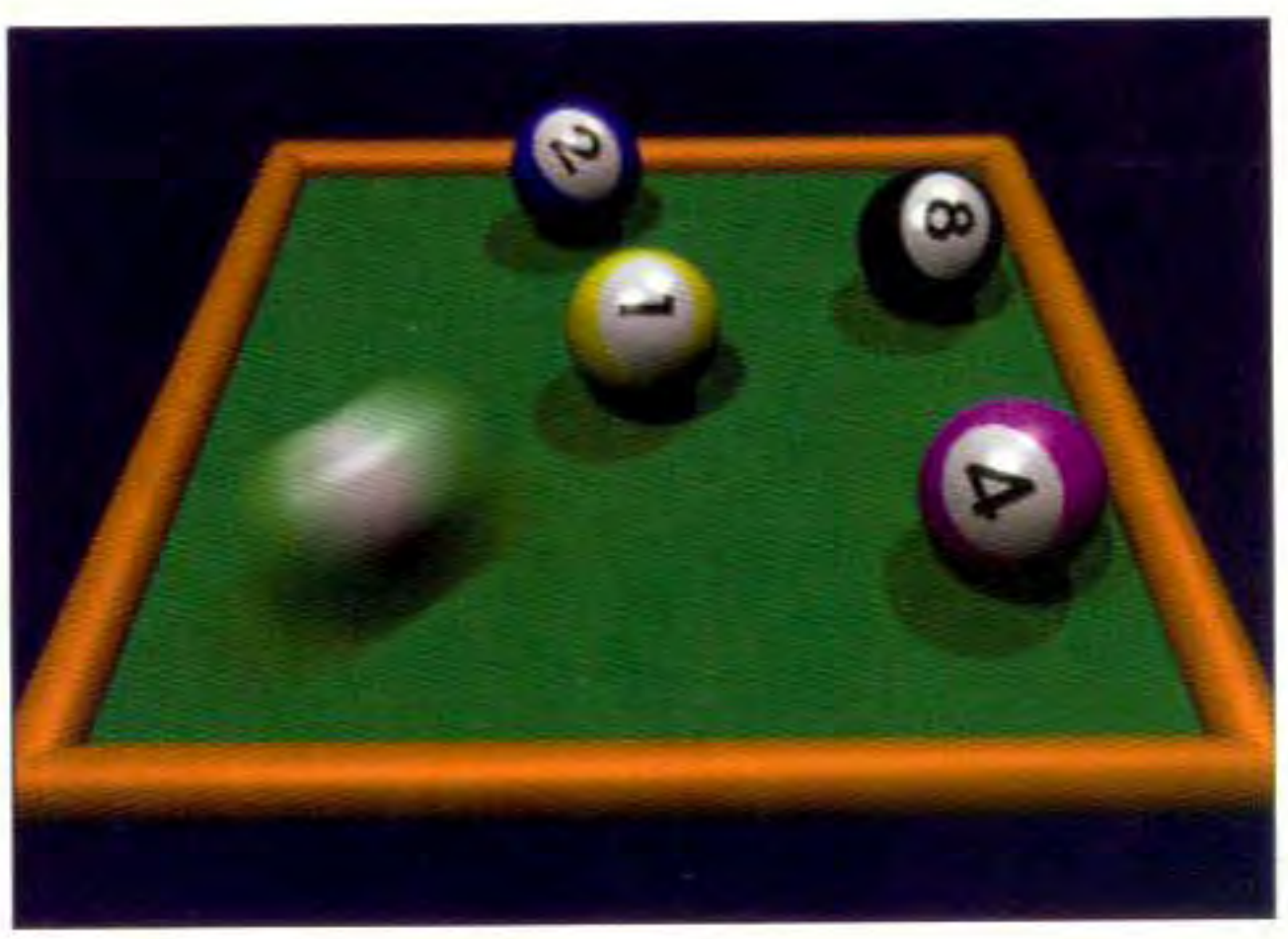


彩图 17：场景中的光源

（版权归 *Unigine* 所有）

彩图 18：展示大理石纹理的棋盘以及木头纹理和金属纹理的棋子

（版权归 *Sven Maerivoet* 所有）



彩图 19：包裹纹理图到台球上

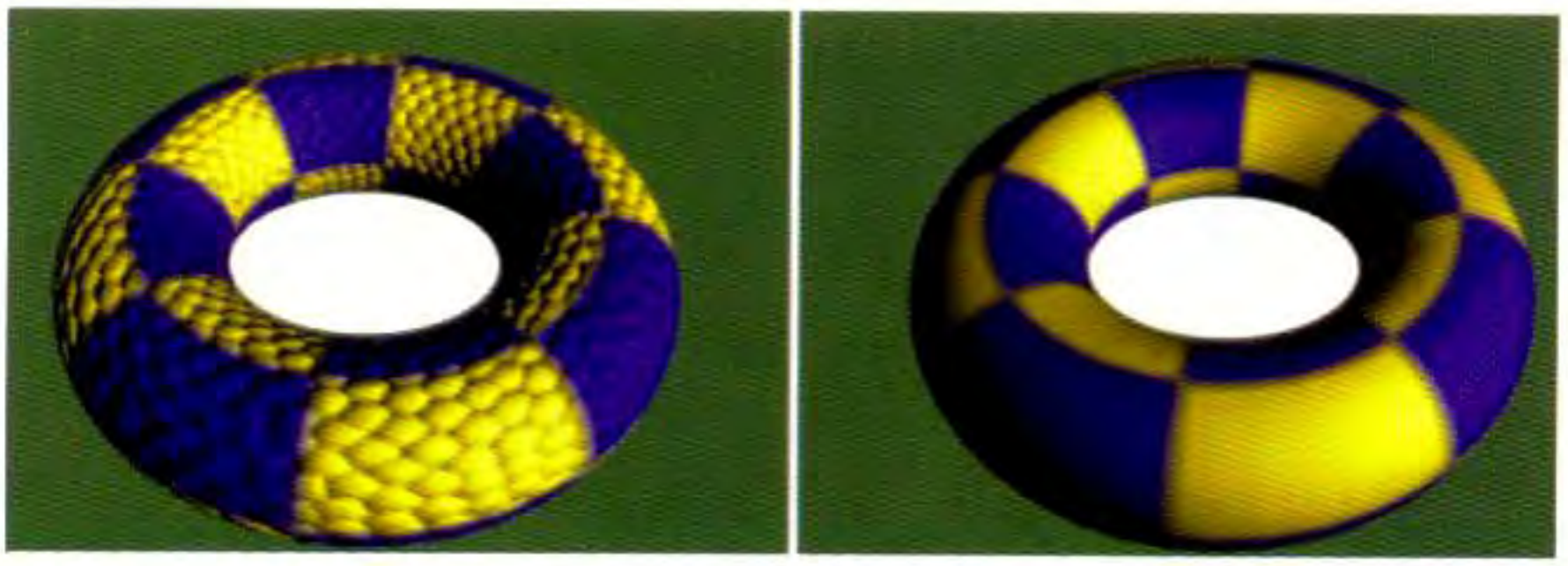
（版权归 *Sven Maerivoet* 所有）

彩图 20：铬环境图

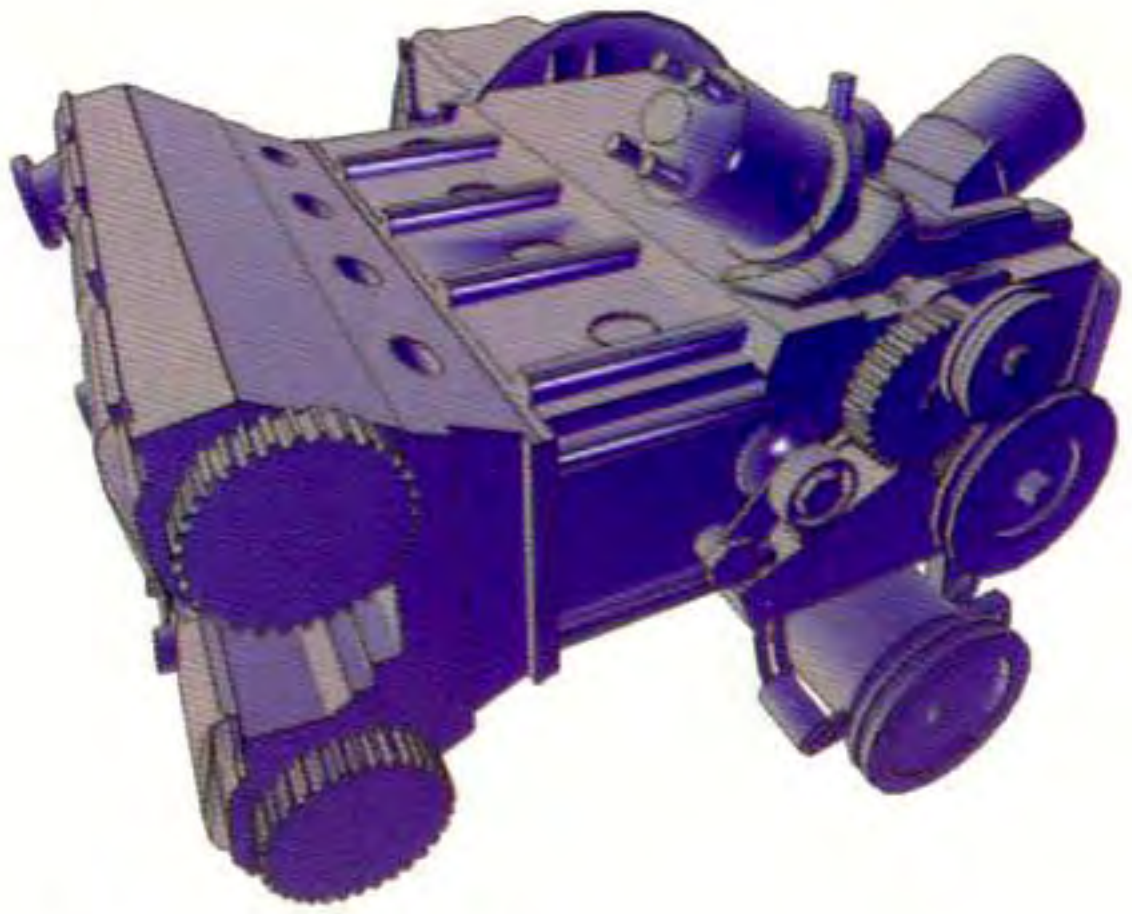
（版权归 *Eric Bpissard* 所有）

彩图 21：铬映射的例子

（版权归 *L. Latta* 和 *P. Debevec* 所有）

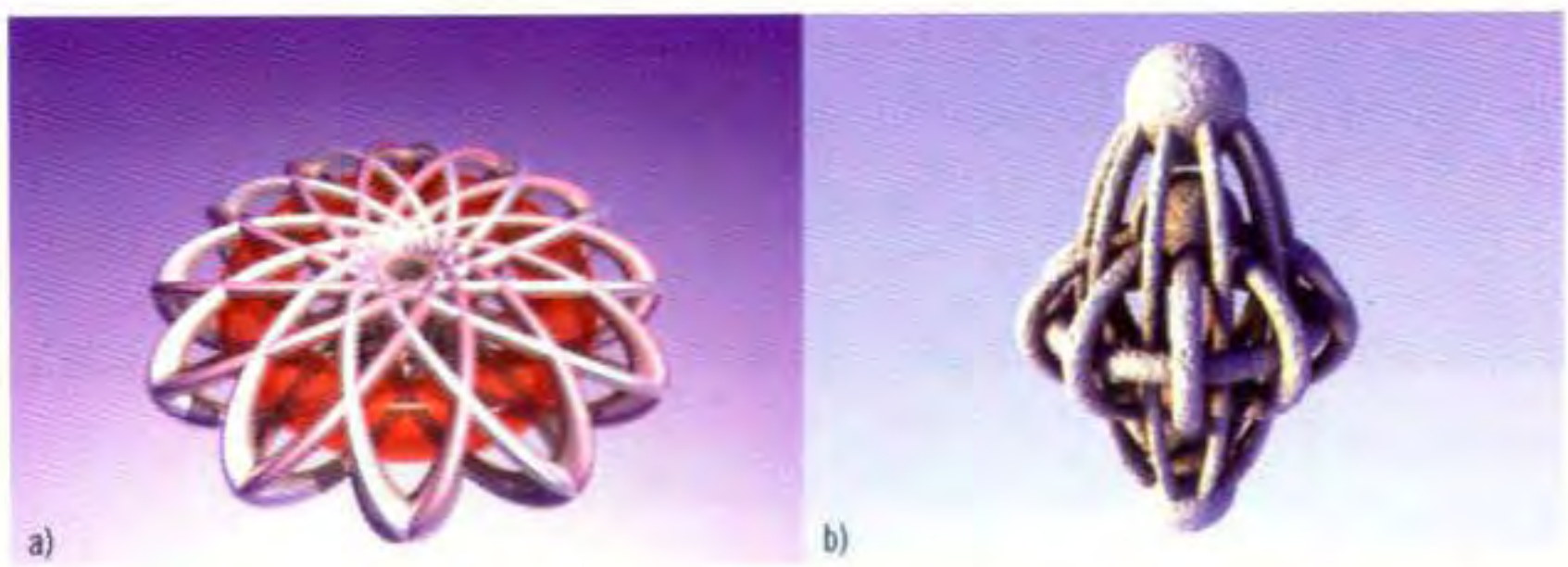


彩图 22：凹凸贴图的例子（左边有凹凸贴图，右边没有）



彩图 23：引擎的卡通绘制结果
（版权归俄亥俄州立大学的 *L. Molnar* 和 *J. Sun* 所有）

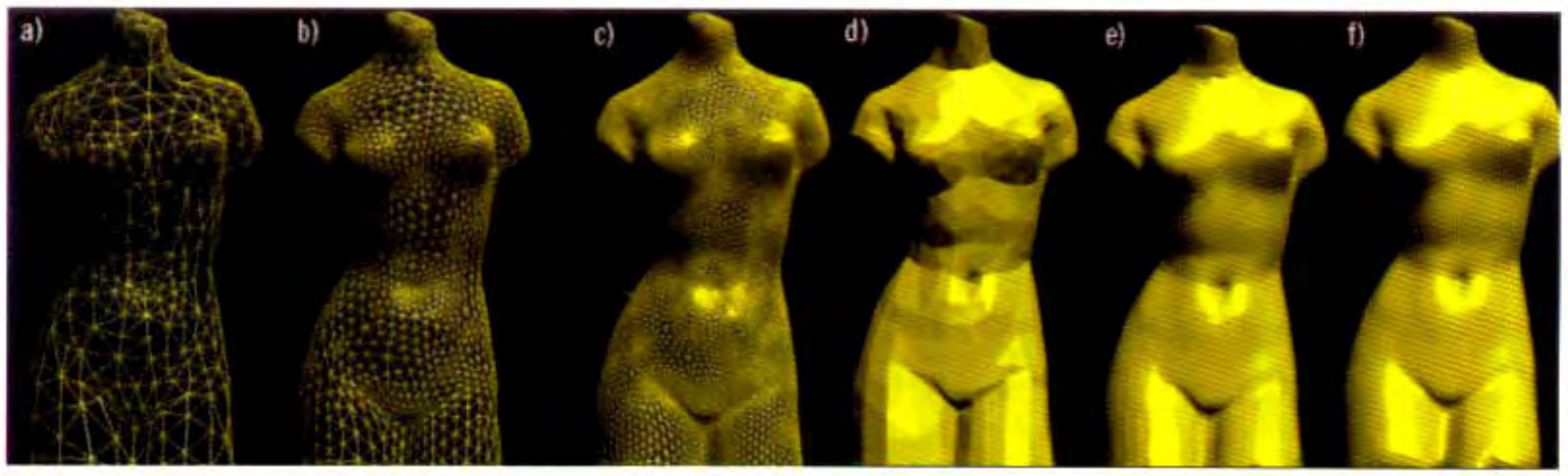
彩图 24：非真实感绘制
（版权归 *Kontantinos Sakelllis* 所有）



彩图 25：a）由 Bezier 面片构建的物体；b）一个错综复杂的捆绑物的复杂形状
（版权归 *Tor Olav Kristensen* 所有，图像是由 *POV-Ray* 生成）

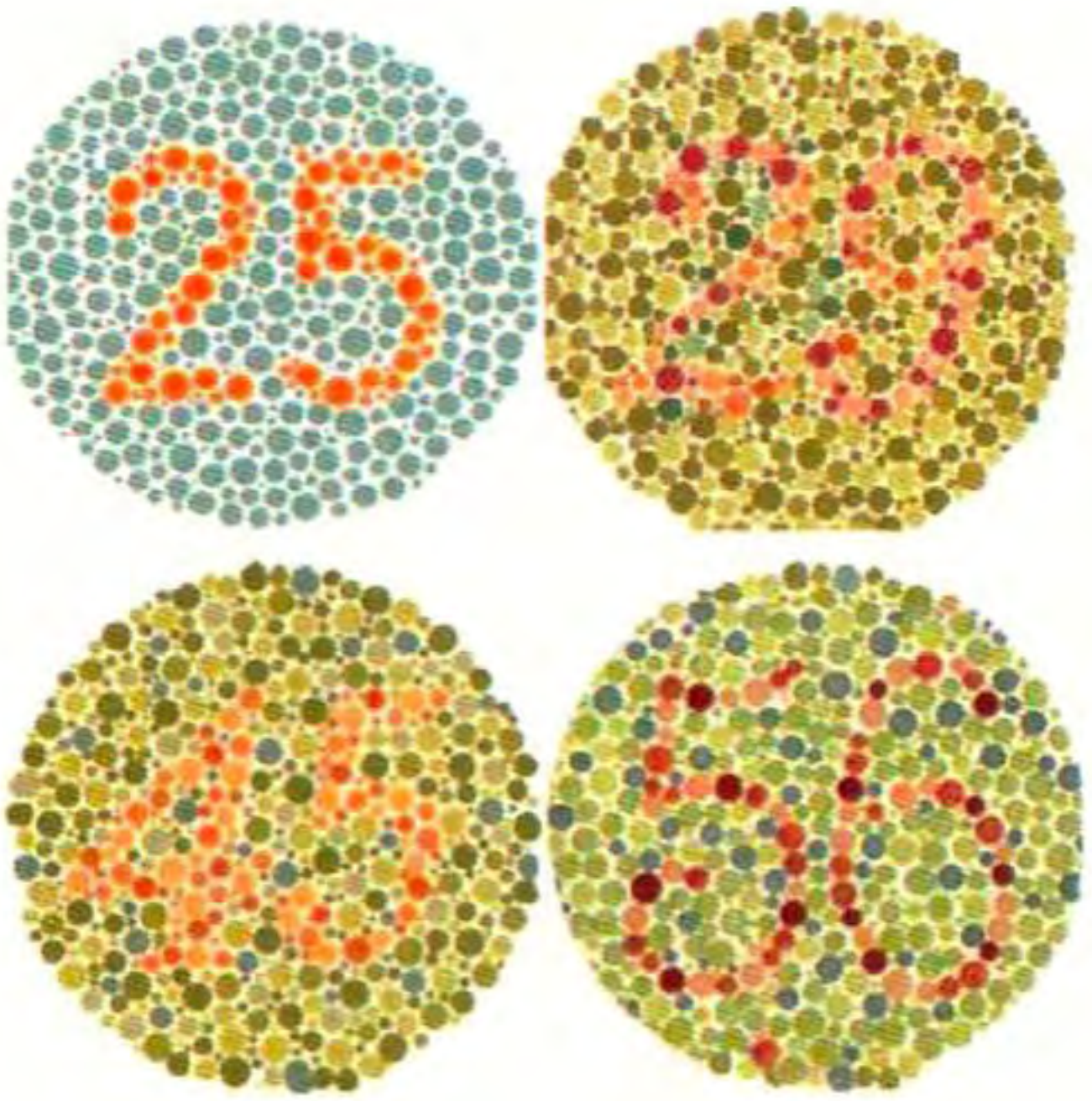
彩图 26：被随机弄乱了的球体模型

（版权归 *Tor Olav Kristensen* 所有，图像由 *POV-Ray* 生成）



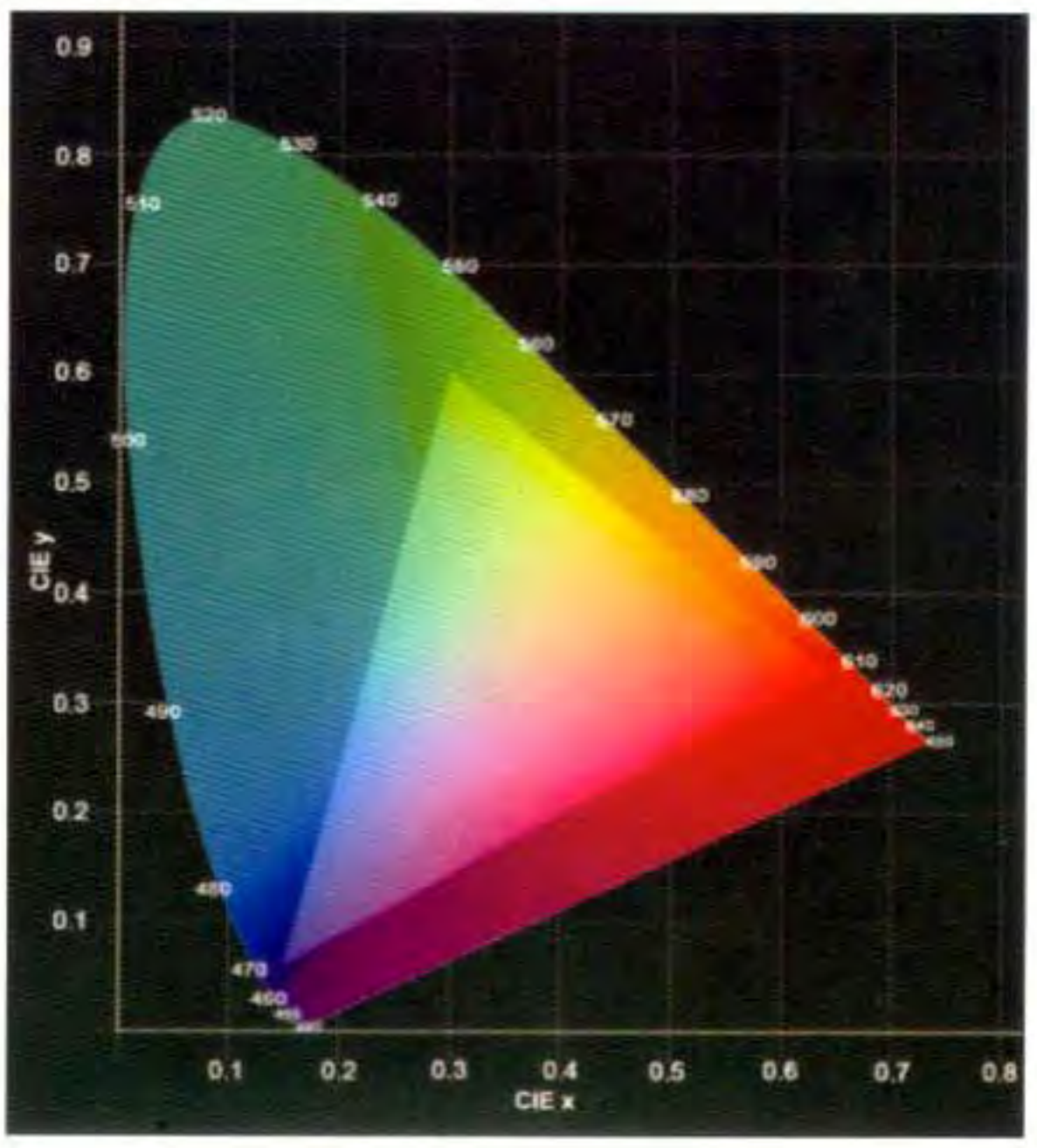
彩图 27：网格模型的连续细分。a)～c)为线框视图；d)～f)为明暗渲染视图

（版权归 *Jian Zhang* 所有）



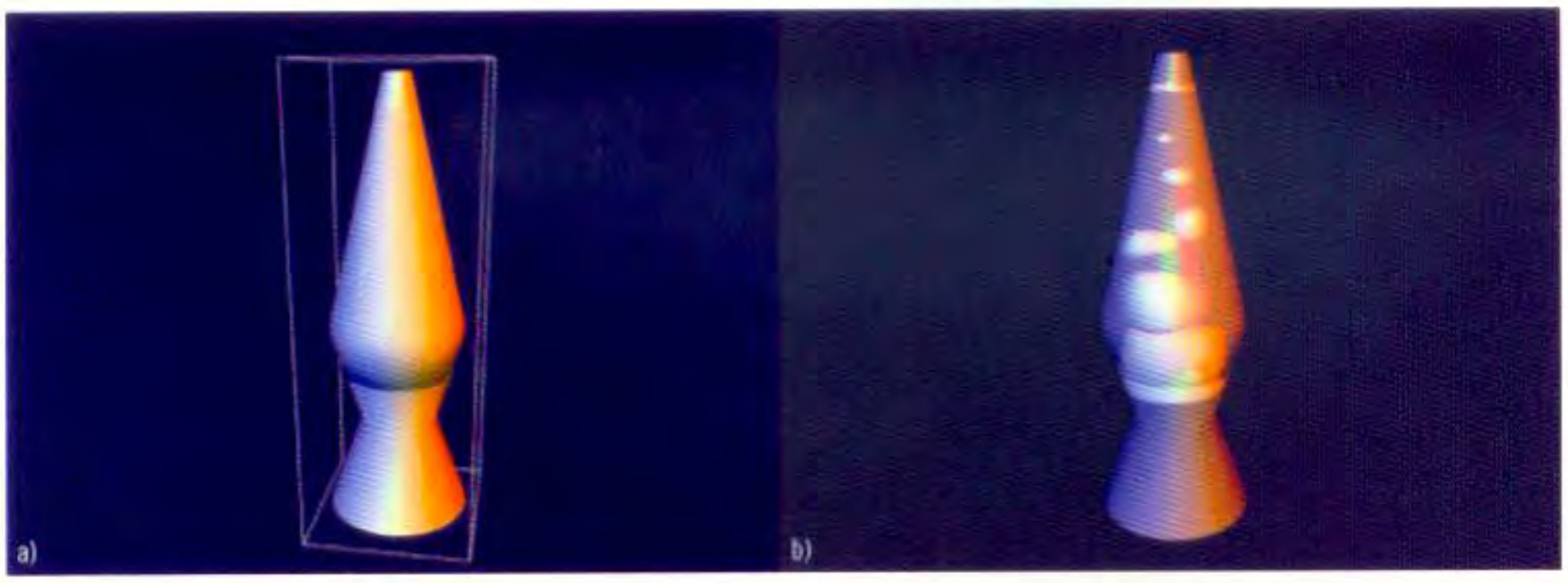
彩图 28：色音测试图

（版权归 *Terrance L. Waggoner* 所有）

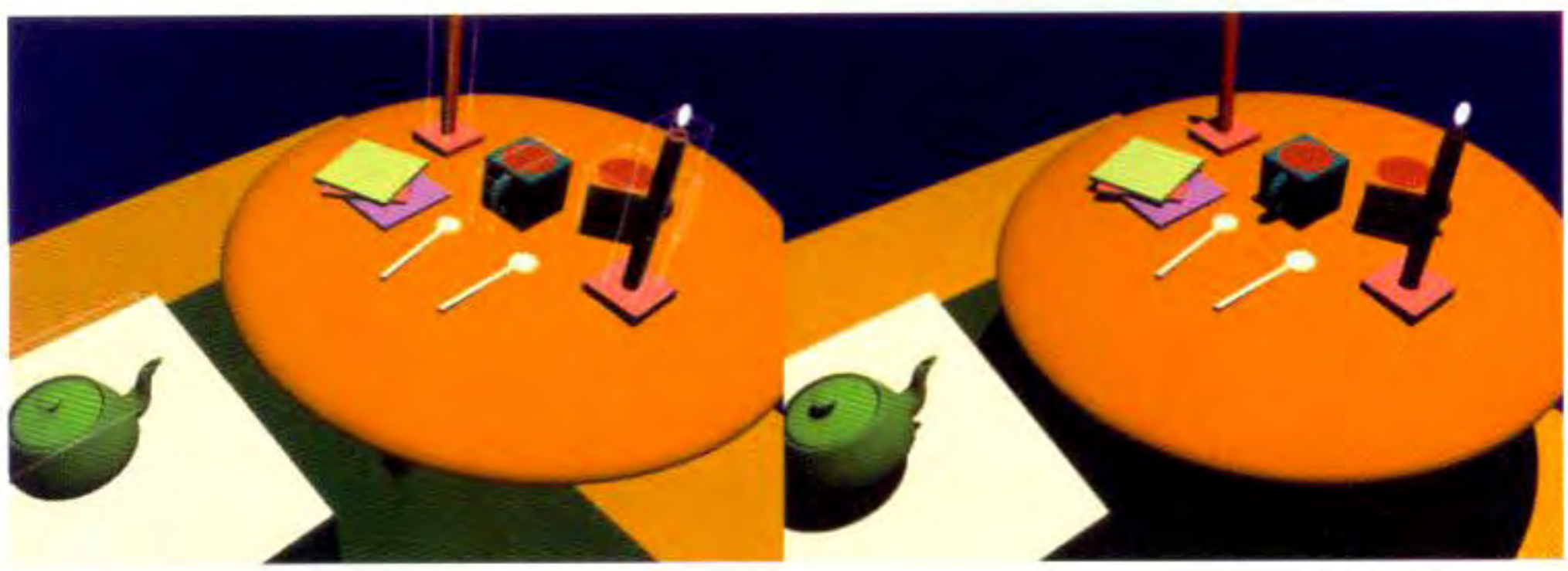


彩图 29：CIE 色度图

（版权归 *TechMind* 所有）

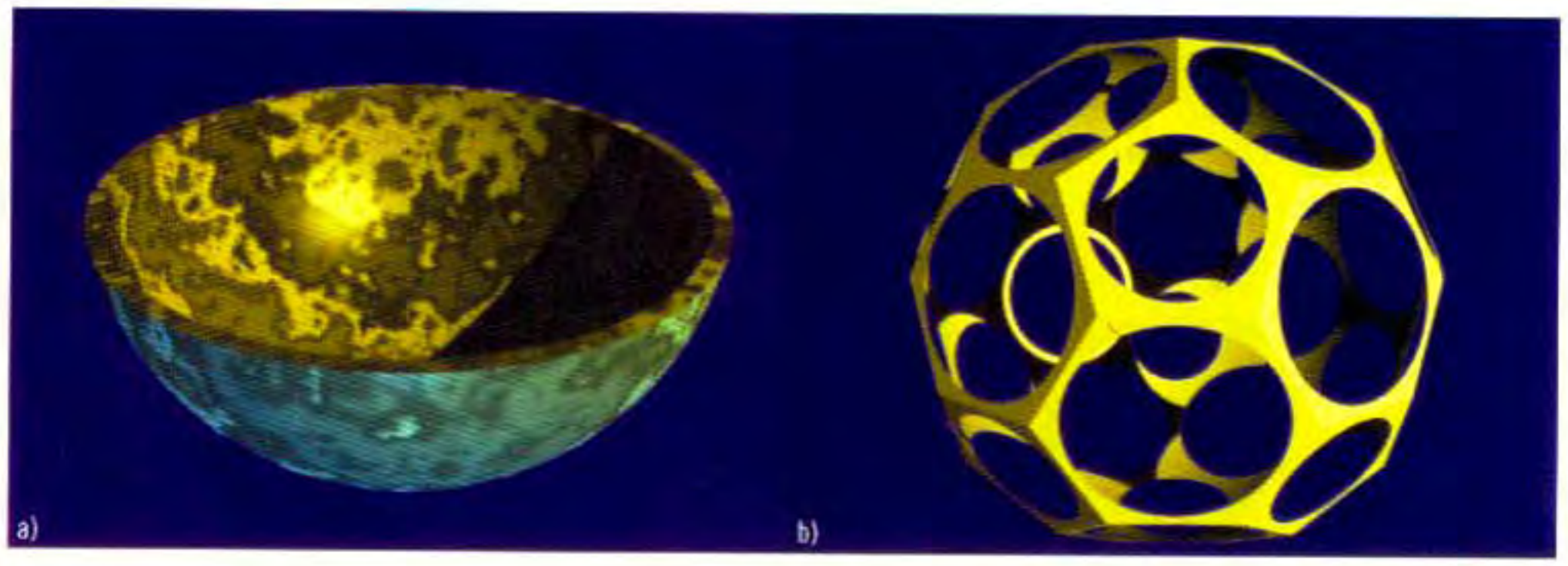


彩图 30：a）带包围盒的蜡灯网格模型（OpenGL 绘制）；b）左图模型的光线跟踪绘制结果（版权归 *Sven Maerivoet* 所有）



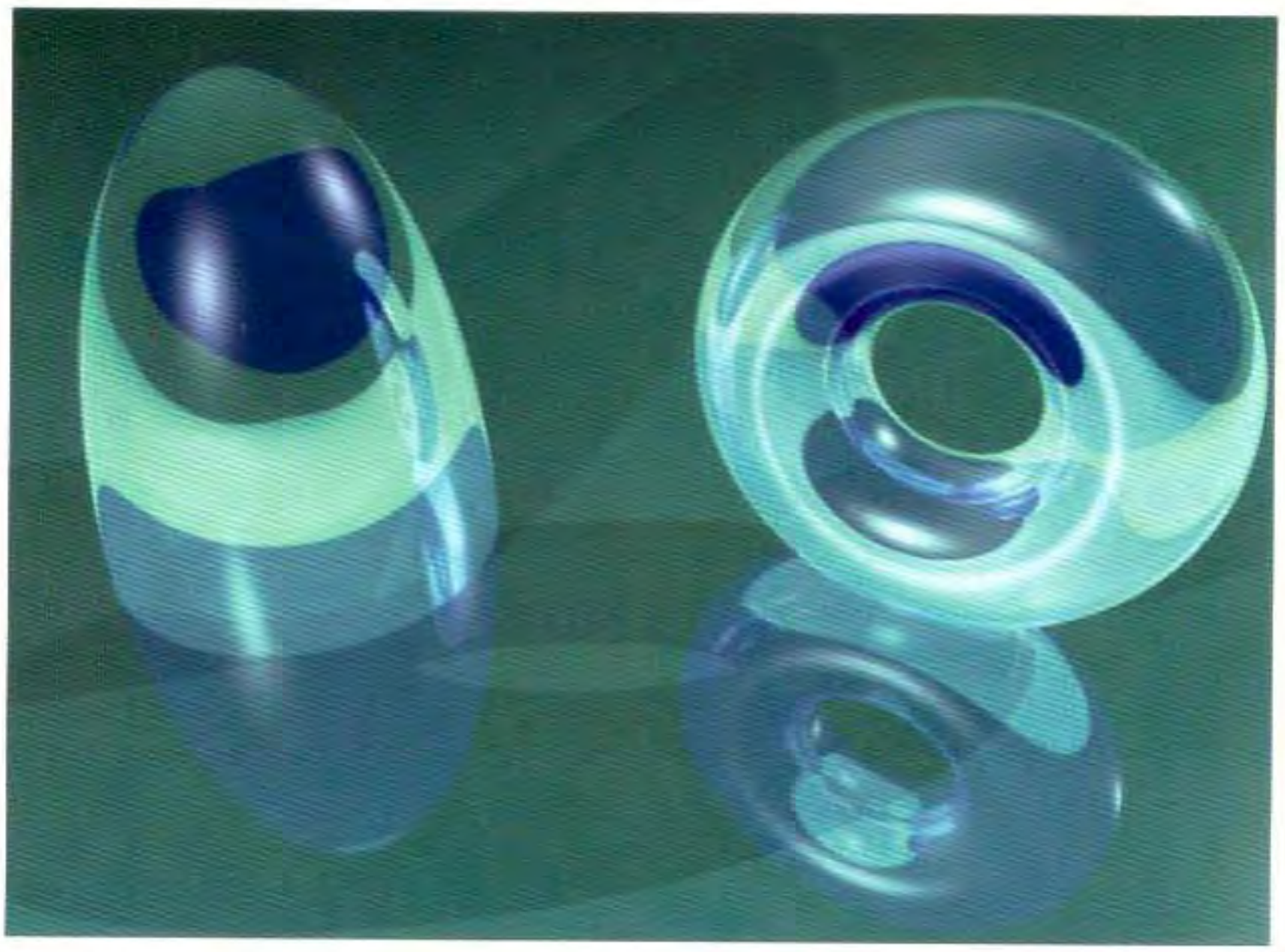
彩图 31：光线跟踪绘制以展示阴影的场景（版权归 *Sven Maerivoet* 所有）

彩图 32：递进光线跟踪：部分为预览，部分为最终绘制结果（版权归 *Sven Maerivoet* 所有）



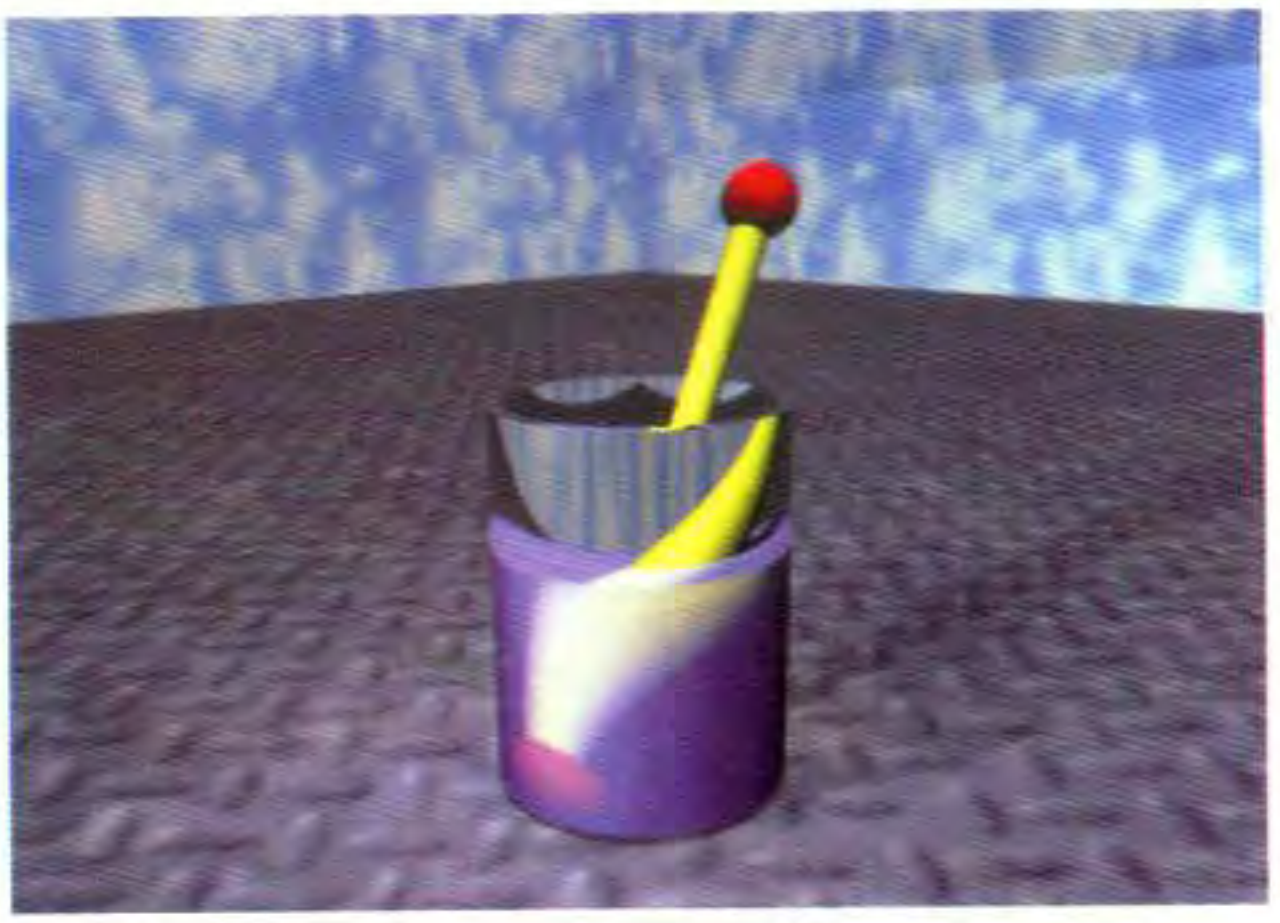
彩图 33：a）光线跟踪绘制的 CSG 物体；b）光线跟踪绘制的 CSG 物体，该物体为由巴克球截去的球体

（版权归 *Sven Maerivoet* 所有）



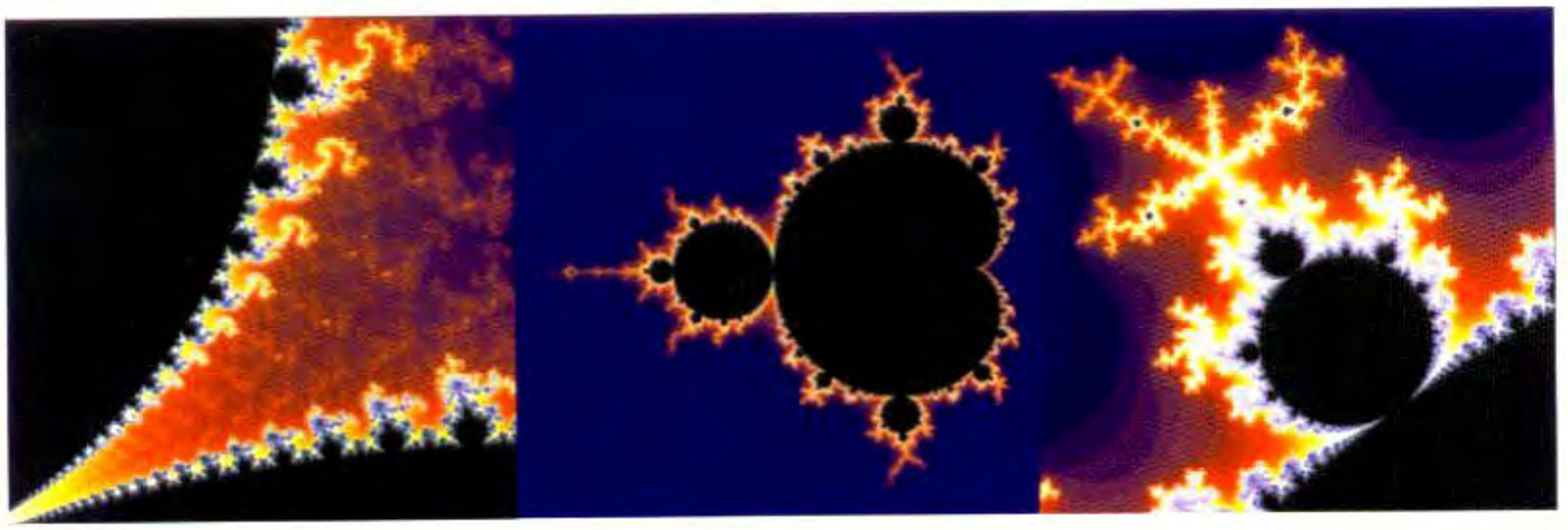
彩图 34：带反射、折射和阴影效果的两个玻璃物体

（版权归 *Sven Maerivoet* 所有）



彩图 35：带阴影、纹理和折射效果光线跟踪图像
（版权归 *Sven Maerivoet* 所有）

彩图 36：从车窗看到的彩虹效果
（版权归 *Emmanuel Agu* 所有）



彩图 37：多个奇异的 Mandelbrot 集合视图
（版权归 *Anton Feenstra* 所有）

Simplified Chinese edition copyright © 2009 by PEARSON EDUCATION ASIA LIMITED and TSINGHUA UNIVERSITY PRESS.
Original English language title from Proprietor's edition of the Work.

Original English language title: Computer Graphics: Using OpenGL, Third Edition by F. S. Hill, Jr. and Stephen M. Kelley. Copyright © 2009
EISBN: 0-13-149670-0
All Rights Reserved.

Published by arrangement with the original publisher, Pearson Education, Inc., publishing as Prentice Hall.

This edition is authorized for sale only in the People's Republic of China (excluding the Special Administrative Region of Hong Kong and Macao).
本书中文简体翻译版由 Pearson Education(培生教育出版集团)授权给清华大学出版社在中国境内(不包括中国香港、澳门特别行政区)出版发行。
北京市版权局著作权合同登记号　图字：01-2007-2625

本书封面贴有 Pearson Education(培生教育出版集团)激光防伪标签，无标签者不得销售。

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

图书在版编目（CIP）数据

计算机图形学(OpenGL 版)(第 3 版) / (美)希尔(Hill, F. S.),(美)克雷(Kelley, S. M.)著;胡事民等译. —北京：清华大学出版社，2009.2
(世界著名计算机教材精选)
书名原文：Computer Graphics: Using OpenGL, 3e
ISBN 978-7-302-18697-7

Ⅰ. 计…　Ⅱ. ①希… ②克… ③胡…　Ⅲ. ①计算机图形学—高等学校—教材 ②图形软件,OpenGL—高等学校—教材　Ⅳ. TP391.41

中国版本图书馆 CIP 数据核字(2008)第 154729 号

责任编辑：龙啟铭
责任校对：徐俊伟
责任印制：王秀菊

出版发行：清华大学出版社　　地　址：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　邮　编：100084
社 总 机：010-62770175　　邮　购：010-62786544
投稿与读者服务：010-62776969,c-service@tup.tsinghua.edu.cn
质 量 反 馈：010-62772015,zhiliang@tup.tsinghua.edu.cn
印 刷 者：北京市世界知识印刷厂
装 订 者：三河市兴旺装订有限公司
经　销：全国新华书店
开　本：185×260　印　张：45.75　插　页：8　字　数：1124 千字
版　次：2009 年 2 月第 1 版　　印　次：2009 年 2 月第 1 次印刷
印　数：1～3000
定　价：99.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。
联系电话：010-62770177 转 3103　　产品编号：025878-01

译者序

计算机图形学是利用计算机研究图形的表示、生成、处理和显示的一门重要的计算机学科分支，它是计算机科学中最活跃的分支之一。近年来，随着计算机及因特网技术的迅速发展，计算机图形学正越来越深入我们的生活，它在工业建模、视频处理、游戏制作、影视特技、生物信息和医药医疗等各行各业都有着及其重要的作用。可以说，计算机图形学的应用无所不在，且日益广泛，已经成为计算机科学技术与其他应用学科之间沟通的桥梁，成为许多计算机从业人员的必备素质之一。

本书原著在国外是很有影响力的教材，被许多国外著名的大学所采用。译者在尽量保持原书精髓的基础上，对原书做了一些表述上的调整，以符合中文习惯，原书中的一些笔误，也给予了纠正。本书译者中，胡事民、刘永进、张松海来自清华大学计算机科学与技术系，刘利刚来自浙江大学数学系，均在第一线从事计算机图形学的教学和科研工作。在本书的翻译中，胡事民教授负责第 1、2、8 章，刘利刚副教授负责 10、11、12 章，刘永进副教授负责 5、6、7 章，张松海博士负责 3、4、9 章。经过一年多的努力，本书终于完成翻译，并付印。在翻译过程中，得到了许多同事和学生的帮助，研究生徐昆、高岳、李勇、张砚、雷励星、陈韬、张一飞、藏域、程明明、来煜坤、勒力、杨永亮、张国鑫、王瑀屏、边哲、佟强、白石、沈超慧、卢少平、陈中贵、陈仁杰、陈军、胡建伟、孟敏、张冬梅等参与翻译了部分初稿，在此一并表示感谢。

由于译者水平有限，本书翻译过程中难有疏漏和不妥之处，敬请读者批评和指正。

前 言

对于有志于学习计算机图形学基本原理和技术，并且渴望自己能够编写优质图形学程序的读者，本书为他们提供了入门的详细介绍。计算机图形学具有强大的生命力并正在持续发展。动画电影已经展示给人们图形学知识所能做到的令人激动的视觉效果。电脑游戏和因特网中的图形效果同样激励人们去学习如何自己动手做出这样的效果。

图形软硬件系统正在以惊人的速度朝着更好、更快和更廉价的方向发展。每年从研究和业界中提出的大量新技术不断地涌现。然而，图形学最基本的原理和方法，却保持着稳定和连贯，并且这些知识中的大部分可以通过一个图形学课程学习得到。本书将图形学的一些基本思想和方法组织起来，通过阅读使读者从只会编写简单程序的水平，提高到能够自行设计并编写出优良图形学程序的水平。

适用的读者范围

该书被设定为本科高年级或研究生一年级的一个或两个学期的课程教材。该书也可用于自学。该书的主要对象是专业为计算机科学或计算机工程的学生，当然也适用于其他专业如物理和数学的学生。

对数学基础的要求

读者应该具备相当于大学一年级的数学水平；并具备初等代数、几何、三角函数和初等微积分的基本知识。向量和矩阵的有关知识将有助于本书的阅读，但不是必须的。本书在相关章节中介绍了向量和矩阵的基本知识，同时，附录中总结了向量和矩阵的关键概念。

计算机图形学中使用了大量的数学符号来表达线、面和视点之间的基本几何关系。虽然每个数学符号都很容易理解，但所需的符号数量却令人生畏。书中特别强调了使用每种符号或技术的原因，以及如何利用这些数学符号合理地描述图形程序中感兴趣的对象。

对编程水平的要求

一般情况下，读者应具有至少一个学期使用 C、C++ 或 Java 编写程序的经验。相当多有关图形的编程涉及到将几何关系直接转换为代码，包括使用变量、函数、数组、循环和条件测试等，而这些东西在不同编程语言中都是类似的。本书使用的编程语言是 C++，但是，有 C 语言背景的读者可以轻易地读懂大部分材料。

读者如果熟练掌握 C 中的结构或 C++ 中的类，将非常有助于本书的阅读。如用于描述复杂场景的图形对象结构，这些场景（如一个城堡或一架飞机）可能由许多部分组成，而这

些部分本身又由复杂的更小部分组成。如果能熟练使用链表或树结构也会有一些帮助,但不是必须的。

对于只懂得 C 但不懂得 C++ 的读者来说,需要掌握面向对象的编程理念。本书中定义了许多有用的类(如网格、场景、相机和纹理等类),并且说明了这些类为什么方便易用。在某些场合下,为了使程序更加清晰易读,我们使用了面向对象编程中的一些技术,如继承和多态,不过,我们并不过分强调面向对象的方法。

教学理念

第 3 版与前两版相比,几乎被完全重新组织和改写,但是基本的教学理念没有改变:我们坚信计算机图形学是在实际使用中学习到的,一定要动手编写程序并进行测试,才能真正地掌握计算机图形学。本书的一个主要目标就是让读者懂得如何将一个特定的设计任务转换成相应的几何结构,并找到一个合适的数学表达,最后将这个表达转换为算法和程序代码。本书将以循序渐进的方式让读者首先学习到如何编写简单的程序来绘制简单的图像,然后一步一步深入地掌握如何生成更加复杂的特殊效果的技巧。

练习和问题

本书包含超过 350 道来源于实际问题的练习题。其中大部分都是让读者停下来进行思考但并不需要编程的那种类型,这可以让读者自己测试一下知识掌握的程度。其余小部分则需要编程实现。

此外,在所有章节的末尾,列出了总共超过 50 个的实例分析。它们都是一些编程的项目,难易程度由简到繁,适合于作为课后作业。这些案例分析对相应章节内的内容进行了扩展。无论是否全部完成这些实例分析任务,它们都应该作为相应章节的一个有机组成部分。

每一个实例分析任务都注明了相应的难易程度,用以判断完成该任务所需的大致时间。学生编程能力各异,编程过程不可预测,但可大致总结如下。

难易程度

Ⅰ. 简单的练习,应该可以一晚完成。

Ⅱ. 较难任务,应该可以一周内完成,读者需要一定的时间来设计程序,并且有足够的时间反复测试和调试程序。

Ⅲ. 复杂任务,一般需要 3 周时间来完成设计和实现。完成它可以被视为一项重大进步。

OpenGL 的使用

刚开始学习计算机图形学时,一个很大的障碍就是如何绘制一幅图像。写一个程序通常很简单,但最终要通过某些工具在屏幕上绘制出来。现在,这些工具已经存在并且很容易获取。1992 年,Silicon Graphics 公司推出了 OpenGL。现在它已经成为最广泛使用的图形应用编程接口(API)之一。OpenGL 通过提供可以在程序中调用的方法接口来绘制图像,并且它可以在因特网上免费下载。正如附录 1 中所述,在大专院校和工业界中使用的各种计算机系统上,都可以使用 OpenGL。它易于安装和使用,目前作为标准的图形 API 已

经成熟。它的制定和修改由 OpenGL 评审委员会(一个负责引导 OpenGL 发展的工业协会)负责。

OpenGL 非常适用于计算机图形的一个特殊原因是因为它的设备独立性或可移植性。不同院校的机房使用不同种类的计算机,学生可以利用 OpenGL 在任何一台计算机上开发并运行程序。此外,该程序也可以在不同操作系统下、不同类型的另一台计算机上运行,两台计算机上生成的图像也一定相同。

OpenGL 提供了丰富且易用的二维图形和图像操作的 API 函数,然而 OpenGL 最强大之处在于处理三维图形。使用 OpenGL,读者可以在一个学期之内学习并掌握制作精美动画效果的技巧。

OpenGL 的发展历史

OpenGL 的第一个版本 1.0 在 1992 年发布。从那时起,OpenGL 一直都是一个成功的跨平台的图形 API。第一个版本可以很好地兼容一个叫做 IrisGL 的专有 API,这种 API 是由 Silicon Graphics 公司设计开发的。兼容的目的是为了建立一个工业标准。为此,Silicon Graphics 公司联合其他几家图形硬件公司,共同制定了一个开放标准,命名为 OpenGL。不足为奇,在随后的 10 年,人们努力使得 OpenGL 变得更好和更加强大。在这段时间,图形硬件也发展迅猛,产生了越来越强大的图形加速卡,这使得程序开发者迫切需要一个能够达到 OpenGL 最佳效果的 API。因此,OpenGL 提供了一系列的扩展函数,允许开发者在图形硬件新特征出现后就可以马上充分利用它们。

每当有一种强大的软件系统面世,标准的问题也随即产生。对于 OpenGL,为使它朝着可控、有序的方向发展,Silicon Graphics 成立了一个监督组织,该组织被称作 OpenGL 架构评审委员会(Architecture Review Board,ARB)。ARB 的主要任务是指导 OpenGL 规范的制定和一致性测试。OpenGL 规范详尽地描述了每个 OpenGL 方法和标识符的使用方法。目前,ARB 的成员有 3Dlabs、苹果公司、IBM、英特尔、nVidia、ATI、SGI 和 Sun Microsystems 等。

OpenGL 规范中定义的方法由各个图形硬件厂商在图形驱动程序中实现。这个过程需要大量的资金和时间,所以,通常在规范发布后过一段时间,支持该规范的驱动程序才会发布。最新的版本 OpenGL 2.0 在 2004 年 9 月发布,它包含了强大的 OpenGL 着色语言(Shading Language),将在第 8 章对此进行详细介绍。它的前一版本是 OpenGL 1.5,该版本发布于 2003 年。

计算机图形学是一个发展迅速的领域,这使得开发人员往往很急切地等待 OpenGL 官方版本的发布。所以,必须存在一种机制,使得小规模增加的最新 OpenGL 功能可以马上在最新的图形硬件上开发和测试。为此,ARB 提供了一个明确定义的扩展机制。在 OpenGL 扩展注册表(http://oss.sgi.com/projects/projects/ogl-sample/registry)中可以看到扩展的列表,其中定义了命名习惯,新扩展的指导方针和其他相关的扩展讨论。访问扩展是一个平台相关的任务,而可用的扩展取决于计算机上安装的图形硬件。一些库如 GLEW 和 GLEE 可以用来简化在应用程序中使用扩展。

扩展机制使得 OpenGL 以一个合理的成本运行。小规模的改动就可以拥有最新硬件的强大功能。这些扩展一旦被批准并在 OpenGL 扩展注册表中注册登记之后,即可以迅速

发挥作用。当某个扩展已经成熟到可以加入 OpenGL 的核心功能中时,ARB 就会讨论决定是否将其加入,然后发布新版本的 OpenGL。这些内容将在第 8 章中进行讨论。

C++ 编程语言的使用

C++ 是大多数工程和计算机科学专业的首选编程语言。相比于 C,C++ 有几个明显的优点,如引用传递函数参数,这使得显式指针不再需要,代码阅读更加易懂。通过使用流,文件的输入输出(I/O)也被极大地简化。一般来说,C++ 中的所有 I/O 的语法都比 C 中更为清晰。为了使得本书中的描述更为简单,我们没有特别强调 C++ 中运算符的使用。

此外,C++ 中的类很容易开发应用,如二维或三维中的点、线、窗口或颜色,这使代码更加简单而且更加鲁棒。使用类之后,具体的操作如几何对象细节隐藏、编写绘制函数包装绘制、测试某个对象是否相交等都更加清晰易读。

注重三维计算机图形学

由于 PC 上的游戏越来越流行,电影中令人眼花缭乱的动画越来越多,学生都对开发三维交互式的图形应用程序特别有兴趣。为了能让读者更快地进入三维图形学的主题,本书重新组织了第一版和第二版中的若干章节。在很多地方,二维和三维的概念结合在一起论述,这可以帮助理解两者之间的相似和区别。

使用场景设计语言来描述三维场景

如果用原始的 OpenGL 命令来描述一个包含很多三维物体的场景,将会非常不雅观而且费时。例如,如果使用 OpenGL 的命令逐一定义 6 个面来描述一个立方体将显得非常乏味。所以,在第 5 章中会介绍一个简单的场景描述语言 SDL(在附录中有更加详尽定义)。使用这个描述语言,学生可以使用熟悉的词汇来描述场景,如"立方体"、"球"和"旋转"等,并建立包含这些词汇的文件。这些文件可以在运行时读入到程序中。本书的附录(或本书的网站)中给出了一个能够阅读 SDL 文件并建立文件中所描述物体的解释程序。这样,用 OpenGL 来绘制场景文件中的物体列表就变得非常简单。

本书的结构和课程安排

本书包含多于一个学期的教学内容,甚至多于两个学期的内容(对前两版也是这种情况)。本书经过细致的编排,使得授课老师可以根据课程的长度和课程的背景选择不同的章节组合来授课。下面在介绍完各章的主要内容之后,将介绍几个建议的章节组合教学方案。

每章概要

第 1 章

本章结合若干展示图形学广泛应用的例子,简要介绍了图形学领域的基本范畴。本章具体描述了不同类型的图形显示系统和用于显示的基本元素(多边形、文本和图片等)。此外,对常用的输入设备如鼠标、键盘和数据手套等也有详细描述。

第 2 章

本章将让学生开始着手编写图形程序，包括用 OpenGL 编程和开发几个完整的线绘制程序(包括著名的 Sierpinski 垫圈)。本章讨论了用 OpenGL 绘制基本元素(如折线和多边形)的方法和如何在程序中交互地使用鼠标和键盘。本章最后的实例分析会给出一些有趣的编程题目，它们能让学生对如何编写图形程序有更加清晰的感觉。其中的一个典型实例分析是建立一个用于初始化 OpenGL 和打开一个 OpenGL 绘制窗口的类。

第 3 章

本章主要讲解窗口到视口变换的主要思想，以及在显示器上放置图片和确定图片大小的方法。具体讨论了以下内容：如何自我管理程序窗口和视口，而把细节留给 OpenGL 处理的方法；缩放、摇动镜头等有趣的效果和简单的图像动画；复杂的基于多边形的图像、圆和弧的绘制；二维和三维曲线的参数化表示方法。

第 4 章

本章将回顾向量和向量运算的基本知识，并指出在图形学中使用向量的巨大好处。熟悉向量知识的读者可以快速地浏览本章，把注意力放在如何用向量去描述程序中几何对象操作之间的关系。向量的一个很有用的性质是，许多向量运算符并不需要知道向量的维数即可操作。叉积是一个例外，它只适用于三维向量。

本章还介绍了坐标系的概念，并讲解了如何使用齐次坐标。本章讨论以下内容：点的仿射变换，用以区分点和向量(以避免掉进图形学编程的常见陷阱)；几个插值、初等 Bezier 曲线和直线相交的程序；细致地论述了直线被凸多边形剪裁的基本算法。

第 5 章

变换是图形学的重要内容之一。学生们有时很难直觉地理解变换，尤其是三维变换。本章将讨论图像变换的基本原理和二维、三维情况下使用仿射变换的坐标系。为加深理解，我们在一开始讨论变换时就使用了齐次坐标。需要特别注意的是三维中的旋转，它们很难被可视化。

本章还概述了 OpenGL 的绘制流水线和其中视图变换、投影变换、视点变换所扮演的角色。讨论了如何使用 OpenGL 工具绘制三维物体，如何使用场景描述语言，并给出了如何使用 SDL 读取器从文件中读取一个三维场景并绘制出来。

第 6 章

本章详细描述了用于复杂网格模型的建模和绘制工具，并给出了几个典型网格模型的构造方法，包括多面体(如十二面体和布基球)和其他复杂的形状(如穹顶、波浪形的水管和复杂表面)。同时，也讨论了如何绘制这些物体，包括平坦绘制和光滑绘制。

第 7 章

本章讨论了可用于灵活观察三维场景的各种工具，定义了产生透射效果的合成相机，并

讨论了它与 OpenGL 提供的低端视点工具之间的关系。本章中提供了一个方便易用的相机类,其中包含操作相机的细节,这使得实现场景漫游变得简单。

本章还介绍了透射投影的数学细节,同时讨论了 OpenGL 是如何通过操作矩阵来产生透射效果的。还讨论了使用齐次坐标(正如 OpenGL 中所使用的)的剪裁算法,以及产生立体视图的方法。

第 8 章

本章介绍了使三维场景变得更加真实的若干绘制方法,包括若干绘制模型(也叫光照模型)和不同光照分量(环境光、漫反射和镜面反射)反射特性的描述。还介绍了如何用 OpenGL 设立光源和改变物体表面材质的属性;细致讨论了 OpenGL 中使用的用于隐藏面消除的深度缓存算法。同时还介绍了如何将纹理(包括过程纹理和图片纹理)贴到三维物体的表面,以使得物体看上去更加逼真。最后介绍了简单阴影的生成方法。

第 9 章

本章讨论了在光栅显示器上操作图像的图形方法,并重新学习作为存储和操作图像的基本工具——像素图,和一些基于像素图的运算操作。详细介绍了传统的 Bresenham 线绘制算法和多边形区域填充方法。还讨论了经常困扰程序员的走样现象(或称为"锯齿"现象)和一些反走样技术。此外,还介绍了颜色抖动技术,该技术可让设备显示比允许数目更多的颜色数量。

第 10 章

本章主要讲解光滑曲线曲面的设计和绘制方法。介绍了贝塞尔曲线和 B-样条曲线的原理,以及有理 B-样条和 NURBS 曲线的理论。讨论了交互式曲线设计,这允许设计人员通过鼠标指定一系列控制顶点,然后通过一个曲线生成算法预览曲线。生成的曲线可能是顶点插值得到或者仅仅是靠近这些顶点。

本章还介绍了如何使用贝塞尔、B-样条和 NURBS 曲面片组合成复杂的曲面,并介绍了无缝拼接两片曲面的方法。

第 11 章

本章讨论了错综复杂的人类颜色感知系统并介绍了如何在数值上定量描述颜色的方法。介绍了 CIE 标准色度图和用它来计算颜色的许多不同方法。也讨论了不同设备的色域,几类不同的颜色空间和它们之间的转换方法。

第 12 章

本章中介绍的光线跟踪方法可以绘制真实感很强的三维场景。通过本章的学习,读者可以设计出一个基本的简单光线跟踪器。然后通过不断地加入一些高级功能,最终可以做出一个能够生成精美图片的完整光线跟踪器。本章首先介绍光线与不同形状的求交方法,然后介绍使用不同光照模型绘制物体的方法。还详细讨论了如何将纹理贴到光线跟踪的表面,包括三维纹理(如大理石)和基于图像的纹理。最后介绍了光线跟踪的加速方法,包括使

用包围盒和 BSP 树。

光线跟踪方法的优点包括：它能自动地处理隐藏面消隐和很轻易地得到物体的精确阴影。另外，它还能轻松加入从光滑表面反射的效果和穿越透明物体的折射效果。本章中具体讨论了如何实现这些效果。在本章结尾，讨论了如何对采用 CSG 方法建模的物体进行光线跟踪。

建议阅读方法

所有的阅读方法都将第 1～第 5 章作为基本内容，对于熟悉向量知识的读者，第 4 章可以不用详细阅读。

可选的课程计划

- 一学期的本科生课程，侧重点为三维图形学：阅读第 1～5 章，第 6、7 章的部分内容。
- 如果时间增加到两学期：再加上第 7 章的余下部分，和第 8、9、10 章的部分内容。
- 一学期的本科生课程，侧重点为二维光栅图形学：阅读第 1～3 章，第 4、5 章的部分内容。
- 如果时间增加到两学期：再加入第 7、8 章的部分内容，第 9、10 章的全部内容，第 11 章的部分内容。
- 一学期的研究生课程，侧重点为三维图形学：第 1～7 章，第 8 章的部分内容。
- 如果时间增加到两学期：加入第 8 章的余下部分，第 9 章的部分内容，和第 10～12 章的全部内容。
- 一学期的研究生课程，侧重点为二维光栅图形学：第 1～3 章，第 4～9 章的部分内容。
- 如果时间增加到两学期：加入第 9 章的余下部分，和第 10～12 章的部分内容。

本书附录

- 附录 1：OpenGL 和 GLUT 的获取和安装
- 附录 2：计算机图形学的数学基础
- 附录 3：一些有用的类和场景描述语言
- 附录 4：分形和 Mandelbrot 集合
- 附录 5：相对绘制和海龟作图

补充

书中的材料可以在本书的网站 www. fshiljr. com 下载。本书中的源代码有多种形式：有的是简单的函数，有的是完整可运行的程序。那些标有“完整运行程序”的源代码可以被成功的编译和测试运行。其中有许多代码样例和有用的类库，以及许多图片和纹理。所有这些都是免费下载。

致 谢

本书的第 1 版和第 2 版出自我在马萨诸塞大学授课最近 25 年的笔记。这段时间里,许多学生提供了帮助,包括开发示例程序,并给本课程提供改进的建议。他们也制作了许多优美的图形程序样例,其中一些即出现在本书中。对本书第 2 版和第 3 版给予了重大帮助的学生和同事有 Tarik Abou-Raya、Earl Billingsley、Dennis Chen、Daniel Dee、Brett Diamond、John Greco、Tom Kopec、Adam Lavine、Russell Turner、Bill Verts、Shel Walker、Noel Llopis、Russell Swan、Chandrashekhara A.、Emmanuel Agu、Tom Laramee、Chang Su、Xiongzi Li、Jung-Yao Huang、Andrew Slater、Anjul Srivastava 和 Steve Morin。如有遗漏,请见谅。

在本书第 3 版开始之际,若干同事给了我灵感和指导。我非常感激 Charles Hutchinson 教授(Dartmouth 大学),感谢他所提供的支持。我也非常感激 Michael Wozny 教授(RPI 大学),感谢在本书编写过程中他的热情和鼓励。感谢 Charlie Rupp,感谢他给我的许多图形学的创造性想法。我还要特别感谢 Daniel Bergeron (UNH 大学),他在本书第 1 版的连贯性和可读性方面做了主要的贡献。

我还要感谢以下这些人,以及其他许多没有提及姓名的人,感谢他们的建议和帮助:Edward Hammerand, Arkansas State University; Deborah Walters, SUNY at Buffalo; Suzanne M. Lea, University of North Carolina at Greensboro; John Neitzke, Northeast Missouri State University; Norman Hosay, University of New Haven; David F. McAllister, North Carolina State University; John DeCatrel, Florida State University; Steve Cunningham, California State University; Stanislaus, Paul Heckbert, Carnegie Mellon University; Algelo Yfantis, University of Nevada; Lee H. Tichenor, Western Illinois University; Norman Wittels, Worcester Polytechnic Institute; Matthew Ward, Worcester Polytechnic Institute; Richard E. Neapolitan, Northeastern Illinois University; Jack E. Bresenham, Winthrop University; Michael Goss, Colorado State University; Bikash Sabata, Wayne State University; Mike Purapura、Alcoa、Norton Starr, Amherst College;以及 Paul T. Barham, North Carolina State University。特别感谢新墨西哥大学的 Edward Angel,是他让我在本书第 2 版中使用图形标准 OpenGL,如果没有他的指引,第 2 版就是一个不够成功的项目。一部分编程样例程序由马萨诸塞大学的 Rob Hall 提供。Sven Maerivoet 基于我们的第 2 版,给了许多建议。

本书的一部分撰写工作是我在奥地利的格拉茨技术大学格拉茨分校信息处理和计算机支持媒体研究所学术度假,与 Hermann Maurer 博士(http://www.iicm.edu/maurer)一起工作时完成的。一部分是我在班加罗尔,印度科学院的富布莱特完成的。感谢在这些访问中他们给我的鼓励和帮助。

如果没有我亲爱的伙伴 Stephen Kelley(Intangible 公司 www.intangibleinc.com),本书不可能完成。2000 年 8 月,在美国国家自然科学基金会的资助下,我和他合作教授了计算机图形学的基础课程。Stephen 制作了该课程中的所有图片和示例程序,并负责制作视频和后期编辑。

编写本书之时，Stephen 总是非常热情和乐于助人，和他共事是很件快乐的事情。基于他高深的操作系统知识和良好的艺术感觉，他提供了强有力的技术支持，这些对教授计算机图形学是十分有用的。对于一个大项目，如果有一个好的伙伴是非常幸运的。就像这次，我非常幸运地能和 Stephen 共事。Stephen 也感谢他的家人和朋友对他的努力所给予的支持。

最后，感谢我的父母，我的妻子 Merilee，我的孩子 Greta、Jessie 和 Rosy，感谢他们的耐心和支持。我的女儿 Jessie，在 2005 年 8 月 22 日生下了我的外甥女 Lily，由于她来到这个世界，这个世界变得更加美丽。

致读者：如何看懂立体图片

本书中出现了若干立体图片，用于帮助讨论三维物体。它们通常作为一对并排的图片出现。要看到这些立体图片的效果，强迫左眼看左边的图，右眼看右边的图。最好是将图片放在离眼睛一手臂那么远。这需要一些练习：有些人能很快学会；有些人通过若干次尝试也能学会；有些人却永远不会。当然，就算没有立体的效果，这些图片仍然可以帮助理解。

一种练习的方法是：将两食指摆在眼睛前，相距约 2 英寸，然后眼睛盯着一定距离之外的空白墙壁。当然，每个眼睛都能看到两个手指，但两手指看上去在中间重合在一起。这种重合就是看立体图片时想要的：每只眼睛都能看到两个图片，但却在中间完全重合在一起。当中间的图片像这样重合在一起时，大脑就会根据它们生成一个三维的影像。有人发现，如果用一张白纸放在两个食指中间并靠在鼻子上很有用。这张白纸可以防止每个眼睛看到另一只眼睛该看到的区域。

目 录

第 1 章 计算机图形学概述 …… **1**

1.1 什么是计算机图形学 …… 1
1.2 计算机生成的图片用在哪里 …… 3
1.2.1 艺术、娱乐和出版行业 …… 3
1.2.2 计算机图形学、感知和图像处理 …… 7
1.2.3 过程监视 …… 7
1.2.4 仿真显示 …… 8
1.2.5 计算机辅助设计 …… 8
1.2.6 科学分析与体可视化 …… 9
1.3 计算机图形学中制作图像的基本元素 …… 11
1.3.1 折线 …… 11
1.3.2 文本 …… 13
1.3.3 填充区域 …… 14
1.3.4 光栅图像 …… 15
1.3.5 光栅图像的灰度和色彩表达 …… 18
1.4 图形显示设备 …… 21
1.4.1 线画显示 …… 21
1.4.2 光栅显示器 …… 21
1.4.3 视频卡/3D 加速器 …… 24
1.4.4 其他的光栅显示设备 …… 26
1.4.5 硬拷贝光栅设备 …… 27
1.5 图形输入的基本单元和设备 …… 28
1.5.1 逻辑上的输入图形基元类型 …… 28
1.5.2 物理输入设备的类型 …… 29
本章小结 …… 31
本章习题 …… 32
进一步阅读 …… 32

第 2 章 OpenGL 绘图入门 …… **33**

2.1 生成图像初步 …… 33

2.1.1 设备无关的编程和 OpenGL …… 35
2.1.2 窗口的编程 …… 36
2.1.3 如何打开一个窗口画图 …… 38
2.2 OpenGL 的基本图形元素 …… 40
2.2.1 几个点丛绘制的例子 …… 43
2.3 OpenGL 中的直线绘制 …… 50
2.3.1 绘制折线和多边形 …… 51
2.3.2 使用 moveTo( )和 lineTo( )绘制线段 …… 57
2.3.3 绘制边校正的矩形 …… 58
2.3.4 边校正矩形的长宽比 …… 58
2.3.5 填充多边形 …… 60
2.3.6 OpenGL 中的其他图形元素 …… 61
2.4 与鼠标和键盘的交互 …… 62
2.4.1 用鼠标交互 …… 62
2.4.2 键盘交互 …… 66
2.5 程序中的菜单设计与使用 …… 67
本章小结 …… 71
案例分析 …… 71
进一步阅读 …… 80

第 3 章 更多的绘图工具 …… 81

3.1 概述 …… 81
3.2 世界窗口和视口 …… 83
3.2.1 窗口到视口的映射 …… 86
3.3 裁减线 …… 95
3.3.1 如何裁减一条线 …… 95
3.3.2 Cohen-Sutherland 裁减算法 …… 96
3.4 正多边形、圆和圆弧 …… 99
3.4.1 正多边形 …… 99
3.4.2 正 n 边形的变种 …… 100
3.4.3 绘制圆弧和圆 …… 104
3.4.4 曲线的逐次细化 …… 105
3.5 曲线的参数形式 …… 107
3.5.1 曲线的参数形式 …… 107
3.5.2 绘制参数曲线 …… 110
3.5.3 极坐标形状 …… 111
本章小结 …… 113
案例分析 …… 114
进一步阅读 …… 120

第 4 章 图形学中的向量工具 …… **121**

4.1 概述 …… 121
4.2 向量回顾 …… 124
4.2.1 向量基本运算法则 …… 125
4.2.2 向量线性组合 …… 126
4.2.3 向量的度量和单位向量 …… 127
4.3 点积 …… 128
4.3.1 点积的性质 …… 129
4.3.2 两个向量的夹角 …… 129
4.3.3 b·c 的符号和正交性 …… 130
4.3.4 二维正交向量 …… 132
4.3.5 正交投影和点到直线的距离 …… 133
4.3.6 投影的应用:反射 …… 135
4.4 两个向量的叉积 …… 136
4.4.1 叉积的几何解释 …… 137
4.4.2 求平面的法向量 …… 138
4.4.3 判断平面多边形的凸性 …… 139
4.5 重要几何对象的表示 …… 140
4.5.1 坐标系统和坐标框架 …… 141
4.5.2 点的仿射组合 …… 143
4.5.3 两个点的线性插值 …… 145
4.5.4 使用内插的艺术和动画 …… 145
4.5.5 预览:用二次、三次内插生成贝塞尔曲线 …… 148
4.5.6 表示直线和平面 …… 149
4.6 求两个线段的交点 …… 154
4.6.1 直线求交的应用:过三点的圆 …… 155
4.7 直线和平面求交及裁剪 …… 157
4.8 多边形求交问题 …… 160
4.8.1 处理凸多边形和凸多面体 …… 161
4.8.2 射线与凸多边形的交点以及裁剪问题 …… 161
4.8.3 Cyrus-Beck 裁剪算法 …… 164
4.8.4 更高级的裁剪问题 …… 166
本章小结 …… 167
案例分析 …… 168
进一步阅读 …… 172

第 5 章 物体变换 …… **173**

5.1 概述 …… 174

5.2 几何变换初步 …… 174
5.2.1 点和物体变换 …… 177
5.2.2 仿射变换 …… 179
5.2.3 二维基本仿射变换的几何效果 …… 180
5.2.4 仿射变换的逆变换 …… 184
5.2.5 组合一个仿射变换 …… 185
5.2.6 二维组合变换的实例 …… 186
5.2.7 仿射变换的一些有用的性质 …… 189
5.3 三维仿射变换 …… 193
5.3.1 基本三维变换 …… 193
5.3.2 组合一个三维仿射变换 …… 197
5.3.3 旋转的组合 …… 198
5.3.4 总结三维仿射变换的性质 …… 202
5.4 如何实现坐标系变换 …… 202
5.5 在程序中使用仿射变换 …… 206
5.5.1 为后面的使用保存 CT …… 212
5.6 使用 OpenGL 绘制三维场景 …… 215
5.6.1 观察过程和图形绘制管道概述 …… 215
5.6.2 OpenGL 中的建模和视点工具 …… 218
5.6.3 用 OpenGL 绘制基本形状 …… 221
5.6.4 使用 SDL 从文件中读取一个场景的描述 …… 230
本章小结 …… 234
案例分析 …… 235
进一步阅读 …… 238

第 6 章 使用多边形网格建模 …… 239

6.1 概述 …… 240
6.2 多边形网格实体建模初步 …… 241
6.2.1 定义多边形网格 …… 242
6.2.2 3D 文件格式举例 …… 244
6.2.3 计算法向量 …… 245
6.2.4 网格的性质 …… 246
6.2.5 非实体对象的网格模型 …… 248
6.2.6 在程序中使用网格 …… 248
6.3 多面体 …… 251
6.3.1 棱柱 …… 252
6.3.2 柏拉图实体 …… 253
6.3.3 其他有趣的多面体 …… 258
6.4 拉伸形体 …… 259

6.4.1 生成棱柱 …… 259
6.4.2 带“扭结”的拉伸 …… 261
6.4.3 离散扫掠而成的旋转曲面 …… 268
6.5 用网格逼近光滑物体 …… 269
6.5.1 曲面的表示 …… 270
6.5.2 曲面的法向量 …… 271
6.5.3 仿射变换的影响 …… 273
6.5.4 三个“通用”形体:球面、柱面和锥面 …… 274
6.5.5 构造弯曲曲面的多边形网格 …… 277
6.5.6 直纹面 …… 279
6.5.7 旋转曲面 …… 281
6.5.8 基于 3D 曲线的管状体 …… 284
6.5.9 基于双变量显式函数的曲面 …… 284
6.6 粒子系统与基于物理的系统 …… 285
6.6.1 粒子系统 …… 285
6.6.2 基于物理的系统 …… 286
本章小结 …… 287
案例分析 …… 288
进一步阅读 …… 294

第 7 章 三维观察 …… 295

7.1 概述 …… 295
7.2 再次研究摄像机 …… 296
7.2.1 设置视景体 …… 297
7.2.2 摄像机的定位和定向 …… 297
7.3 在程序中指定一个摄像机 …… 301
7.3.1 交互地移动摄像机 …… 303
7.4 三维物体的透视投影 …… 307
7.4.1 点的透视投影 …… 308
7.4.2 直线的透视投影 …… 310
7.4.3 在图形绘制管道中加入透视投影 …… 313
7.5 生成立体视图 …… 326
7.6 投影的分类 …… 327
7.6.1 一点、两点以及三点透视 …… 328
7.6.2 平行投影的类型 …… 330
本章小结 …… 334
案例分析 …… 334
进一步阅读 …… 336

第 8 章 几何体的真实感渲染 …… **337**

8.1 概述 …… 337
8.2 渲染模型 …… 340
8.2.1 影响反射光的几何因素 …… 341
8.2.2 如何计算漫反射分量 …… 342
8.2.3 镜面反射 …… 343
8.2.4 环境光的作用和人类的视觉感知 …… 346
8.2.5 如何综合光的贡献 …… 347
8.2.6 增加颜色 …… 347
8.2.7 渲染和图形管线 …… 350
8.2.8 在 OpenGL 中使用光源 …… 351
8.2.9 在 OpenGL 中使用材质属性 …… 355
8.2.10 渲染 SDL 设定的场景 …… 356
8.3 平面着色与平滑着色 …… 357
8.3.1 平面着色和马赫带 …… 358
8.3.2 平滑着色 …… 359
8.4 增加隐藏面消隐 …… 362
8.4.1 深度缓存方法——OpenGL 使用的方法 …… 362
8.5 为面片添加纹理 …… 365
8.5.1 在平面上贴纹理 …… 367
8.5.2 渲染纹理 …… 369
8.5.3 什么是纹理调和 …… 376
8.5.4 一个 OpenGL 纹理的例子 …… 377
8.5.5 在曲面上卷绕纹理 …… 382
8.5.6 反射映射 …… 386
8.6 给物体添加阴影 …… 389
8.6.1 阴影简介 …… 389
8.6.2 利用阴影缓冲区绘制阴影 …… 391
8.6.3 辐射度的简要介绍 …… 393
8.7 OPENGL2.0 和着色语言(GLSL) …… 393
8.7.1 凹凸贴图 …… 394
8.7.2 非真实感绘制 …… 395
本章小结 …… 396
案例分析 …… 397
进一步阅读 …… 398

第 9 章 光栅显示工具 …… 399

9.1 概述 …… 399

9.2 处理像素映射 …… 401
9.2.1 对像素映射的操作 …… 401
9.2.2 对像素映射有用的数据类型 …… 402
9.2.3 图像缩放和旋转 …… 407
9.3 合并像素映射 …… 409
9.3.1 读-改-写循环 …… 410
9.3.2 alpha 通道和图像融合 …… 410
9.3.3 像素的逻辑合并 …… 414
9.3.4 BitBLT 操作 …… 418
9.4 自己实现直线绘制：Bresenham 算法 …… 419
9.4.1 Bresenham 直线绘制算法 …… 420
9.5 定义和填充像素区域 …… 429
9.5.1 区域定义 …… 429
9.6 处理符号定义的区域 …… 430
9.6.1 矩形定义的区域 …… 430
9.6.2 路径定义的区域 …… 431
9.7 填充多边形定义的区域 …… 432
9.7.1 边界上的哪些像素属于多边形 …… 433
9.7.2 改进算法性能 …… 436
9.8 走样和反走样技术 …… 439
9.8.1 反走样技术 …… 440
9.8.2 纹理的反走样 …… 443
9.8.3 使用 OpenGL 反走样 …… 446
9.9 生成更多的明暗效果和色彩 …… 447
9.9.1 误差扩散 …… 448
本章小结 …… 450
案例分析 …… 451
进一步阅读 …… 452

第 10 章 曲线和曲面设计 …… 453

10.1 概述 …… 453
10.1.1 作为运动轨迹的参数曲线 …… 454
10.1.2 运动的光滑性 …… 454
10.2 用多项式描述曲线 …… 458
10.3 关于交互式曲线设计 …… 461
10.4 用贝塞尔曲线进行曲线设计 …… 463
10.4.1 德卡斯特里奥算法 …… 464
10.5 贝塞尔曲线的性质 …… 468
10.6 寻找更好的混合函数 …… 472

10.6.1 局部控制问题 …… 472
10.6.2 混合函数集的性质 …… 473
10.6.3 分段多项式曲线和样条 …… 475
10.6.4 根据 $g(t)$ 建立一组混合函数 …… 476
10.6.5 样条曲线和基函数 …… 479
10.7 B 样条基函数 …… 480
10.7.1 B 样条函数的定义 …… 480
10.7.2 如何利用节点向量中的多重节点 …… 484
10.7.3 非封闭 B 样条曲线:标准节点向量 …… 485
10.8 B 样条曲线在设计中的重要性质 …… 488
10.8.1 多重控制顶点 …… 489
10.9 有理曲线和非均匀有理 B 样条曲线 …… 490
10.10 插值一瞥 …… 494
10.10.1 利用分段三次多项式插值 …… 494
10.10.2 Hermite 插值 …… 496
10.10.3 自然三次样条 …… 498
10.10.4 三次样条插值中斜率的计算 …… 500
10.10.5 交互指定切向量 …… 503
10.11 自由曲面造型 …… 504
10.11.1 基于 B 样条的直纹面 …… 504
10.11.2 基于 B 样条的旋转曲面 …… 505
10.11.3 贝塞尔曲面片 …… 506
10.11.4 贝塞尔曲面片的拼接 …… 507
10.11.5 B 样条曲面片 …… 509
10.11.6 NURBS 曲面 …… 510
10.11.7 细分曲面 …… 511
本章小结 …… 513
案例分析 …… 514
进一步阅读 …… 522

第 11 章 颜色理论 …… 523

11.1 概述 …… 523
11.1.1 眼睛:人类颜色感知的生理学基础 …… 524
11.1.2 红绿蓝色盲 …… 526
11.2 颜色描述 …… 526
11.2.1 主导波长 …… 526
11.2.2 颜色感知和匹配 …… 527
11.3 CIE 标准 …… 529
11.3.1 构造 CIE 色度图 …… 531

11.3.2 CIE 色度图的应用 …… 532
11.3.3 色域 …… 533
11.4 颜色空间 …… 534
11.4.1 RGB 颜色空间 …… 535
11.4.2 加/减色系统 …… 535
11.4.3 HLS 颜色模型 …… 537
11.5 索引颜色和颜色查找表 …… 538
11.6 颜色量化 …… 540
11.6.1 均匀量化 …… 541
11.6.2 流行度算法 …… 543
11.6.3 中值切分算法 …… 544
11.6.4 八叉树量化 …… 545
本章小结 …… 547
案例分析 …… 547
进一步阅读 …… 549

第 12 章 光线跟踪基础 …… 550

12.1 前言 …… 551
12.2 建立光线跟踪的几何场景 …… 552
12.3 光线跟踪过程纵览 …… 554
12.4 光线与物体求交 …… 555
12.4.1 光线与通用平面求交 …… 556
12.4.2 光线与通用球面求交 …… 557
12.4.3 光线与仿射变换后的物体求交 …… 558
12.5 一个光线跟踪器的应用 …… 560
12.5.1 光线和球体的求交程序 …… 565
12.5.2 一个完整的光线跟踪器 …… 567
12.6 光线和其他基本几何体求交 …… 568
12.6.1 与正方形求交 …… 568
12.6.2 与锥形柱体求交 …… 569
12.6.3 与立方体(或任何凸多面体)求交 …… 571
12.6.4 添加更多的基本形状 …… 576
12.7 绘制场景的明暗图 …… 577
12.7.1 计算交点处的法向量 …… 578
12.7.2 根据物体的表面材质为其着色 …… 578
12.7.3 基于物理的明暗着色模型—Cook Torrance 明暗处理 …… 580
12.8 添加表面纹理 …… 585
12.8.1 体纹理 …… 586
12.8.2 在表面上粘贴图像 …… 594

12.9 光线跟踪算法中的反走样技术 …… 596
12.10 包围盒技术 …… 597
12.10.1 长方体和球体包围盒 …… 599
12.10.2 投影包围盒 …… 603
12.10.3 其他光线跟踪加速方法：二叉空间剖分树技术 …… 605
12.11 添加阴影 …… 607
12.12 反射与透明 …… 610
12.12.1 光线的折射 …… 612
12.12.2 在函数 shade()中处理折射 …… 616
12.13 复合体：物体上的布尔操作 …… 618
12.13.1 CSG 体的光线跟踪 …… 620
12.13.2 布尔物体的数据结构 …… 622
12.13.3 布尔体与光线的交 …… 624
12.13.4 为 CSG 物体构建和使用包围盒 …… 627
12.14 光线跟踪与光线投射 …… 628
本章小结 …… 629
案例分析 …… 630
进一步阅读 …… 632

附录 1 图形工具：怎样获取和安装 OpenGL …… 633

A1.1 获取和安装 OpenGL …… 633

附录 2 计算机图形学的数学基础 …… 636

A2.1 一些关于矩阵及其运算的主要定义 …… 636
A2.1.1 矩阵运算 …… 636
A2.1.2 两个矩阵相乘 …… 637
A2.1.3 矩阵分块 …… 639
A2.1.4 矩阵的行列式 …… 639
A2.1.5 矩阵的逆 …… 640
A2.2 向量及其运算的一些性质 …… 641
A2.2.1 向量的正交、正交点积 …… 641
A2.3 球坐标和方向余弦 …… 642

附录 3 有用的类、例程以及 SDL …… 644

A3.1 2D 图形类 …… 644
A3.1.1 基本 2D 图形类 …… 644
A3.1.2 场景和支持类 …… 652
A3.1.3 一些用于光线跟踪的类 …… 680
A3.1.4 SDL：场景描述语言 …… 682

A3.1.5 Scene 类 …… 683
A3.2 SDL 语法 …… 683
A3.3 SDL 中的宏 …… 687
A3.4 扩展 SDL …… 688

附录 4 分形和曼德布洛特集合 …… 690

A4.1 引言 …… 690
A4.2 分形和自相似 …… 690
A4.3 曼德布洛特集合 …… 691
A4.3.1 曼德布洛特集合和迭代函数系统 …… 691
A4.3.2 如何判断点 c 是否属于曼德布洛特集合 …… 693
A4.3.3 描绘曼德布洛特集合 …… 694
A4.3.4 曼德布洛特集合的一些注释 …… 696

附录 5 相对性和海龟绘图 …… 698

A5.1 编写 moveRel()和 lineRel()函数 …… 698
A5.2 海龟绘图法 …… 699
A5.2.3 绘制曲径 …… 702
A5.2.4 其他类型的曲径 …… 702
A5.3 基于正多边形的图 …… 702
A5.3.1 正多边形 …… 703
A5.3.2 n-gon 的变种 …… 704

第1章 计算机图形学概述

本章学习目标

- 纵览计算机图形学领域
- 描述重要的图形输入和输出设备

预览

1.1 节介绍计算机图形学的相关领域。1.2 节给出了一些当今计算机图形学应用的例子。1.3 节介绍用计算机生成图像的基本方法。在 1.3.4 节中特别介绍了光栅图像的概念，它的应用贯穿本书始终。1.4 节描述了一些现在常用的图形显示设备，1.5 节简要介绍用于交互式图形应用的各种输入设备。

1.1 什么是计算机图形学

什么是计算机图形学？这是个好问题。“计算机图形学”这个术语，在不同的背景下具有不同的含义。简单来说，计算机图形是计算机产生的**图像**。如今无论在哪里你都能找到例子，特别是在杂志和电视上。比如本书利用计算机排版：每个字符都是利用存储在内存中的字符形状库来绘制的。在书和杂志中，计算机制作的图像随处可见。一些图像看上去如此自然，以致于你无法将它们同实景照片区分开来。另外一些图片有人造的或超越现实的外观，这是为了营造某种视觉效果。例如，当今的电影能够展示一些从来不曾存在的景象，这些景象是用计算机精心制作的，融合了真实的和想象的景象。

计算机图形学可以用来指制作这种图像（如插图 1 所示的彩色图像）的**工具**。本书将描述这些工具是什么和怎样使用它们去写程序。图 1.1 展示了用本书中描述的技术所生成的引人入胜的结果，我们将教会读者怎样制作这样的图像。

这类工具既包含硬件工具，又包含软件工具。硬件工具包括显示器、显卡和显示图形用的打印机等，也包括输入设备，例如鼠标、数据手套和跟踪球等，它们可以让用户指向特定方位和绘制图形。理所当然地，计算机本身就是一个硬件工具，它通过特殊的硬件设备使图形显示和图像捕获更加有效。

图 1.1 包括反射和阴影的光线跟踪图像

至于软件工具，对于一些常见的或许你已经很熟悉了：操作系统、编辑器（或文本处理器）、编译器

和调试器。这些工具几乎在任何编程环境中都能找到。对于图形学,必须有一个图形程序集合,它们本身能够绘制图片。例如,所有的图形库都有画简单的线或圆(或字符,如 G)的功能。有些图形库如 OpenGL 包含的功能远超出这些,既有拖动和管理带下拉菜单的窗口,又有输入框和对话框。一些图形库(也包括 OpenGL)还提供了非常复杂的三维功能,它们允许程序员在三维坐标系统中设置一个照相机给物体拍照。

在这本书里我们将展示如何利用图形库写程序,以及如何增强它们的功能。不久以前,程序员还不得不使用高度设备相关的函数库,为特定显示设备类型的特定计算机系统设计程序。当程序需要从一种系统转向另一种系统或从一种设备转向另一种设备时,就非常困难。在大多数情况下,程序员不得不对程序进行实质的改动以便它能被继续使用,这种处理费时且容易出错。幸运的是,现在这种情况好多了。已经存在一些设备无关的图形库,这种图形库允许程序员在程序中使用公共函数集,并使该程序能够在不同系统和显示设备下运行。OpenGL 就是这样一种函数库,它也是本书采用的最主要工具。OpenGL 创作图形的方法在学术界和工业界得到广泛的应用。我们将在第 2 章开始对 OpenGL 进行详细的讨论。

总之,计算机图形学一般是指包括这些工具及其所制作图片的整个研究领域。一般认为这个领域起始于 20 世纪 60 年代早期,由 Ivan Sutherland 在 MIT 所发表的博士论文"画板:一种人机交互图形通信系统"开始。随着学术界和工业界对图形学越来越感兴趣,显示技术和管理图像信息的算法得到了快速发展。图形学专业兴趣小组,简称 SIGGRAPH,成立于 1969 年,今天活跃于世界各地(不可错过的 SIGGRAPH 年会现在每年吸引 30000 参与者)。更多信息可访问 http://www.siggraph.org。今天全球许多公司把计算机图形技术作为其收入的主要来源,大多数高等院校的计算机科学与工程系也都开设了计算机图形学课程。

计算机图形学是一个非常吸引人的研究领域。你将学习编写的程序是创作图片,而不是生成字符串流或数字。相对于抽象数字,人们更愿意接受图形信息,并能够从图片中获取更多的信息。我们的眼一脑系统对于识别可视化的模式是高度协调的。阅读文本,当然也是模式识别的一种方式:我们快速识别字符形状,把它们转化为文字,然后解释它们的含义。只不过我们浏览图片会更敏锐一些。某些信息作为文本可能是难以理解的废话,转化为图形就变成可被快速识别的形状或模式。在一幅图片中蕴含的信息量可能是巨大的。我们不仅能识别图片中有什么,还能通过它的细节和纹理收集大量的信息。

人们研究计算机图形学有很多种原因。有的人只是用它作为有效的工具来描绘曲线和表达研究工作中处理的数据,有的人想制作计算机动画游戏,还有的人寻求艺术表达的新方法,多数人希望工作变得更有效率并能更好地交流思想。计算机图形学对这些都能有所帮助。

潜在的用户:一般来说,用户是对图形学感兴趣的人。计算机程序产生的图形有以下几种方式。

- **一帧一帧的绘制**:一帧一帧的绘制,期间要用户等待(非常枯燥)。
- **用户控制下的一帧一帧绘制**:一系列的帧可以被连续绘制,就像在 PowerPoint 中;用户按下一个键移动到下一张幻灯片,但除此外没有其他的交互方式(枯燥程度小很多)。

- **动画**：一系列的帧按一定速率处理，用户很乐意观看（通常是令人兴奋的，例如动画电影“超人特工队”和“怪物史莱克”）。
- **交互式程序**：观看交互式的图形表达时，用户利用输入设备如鼠标或键盘控制帧的流向，且流向是任意的，在写程序时是无法预知的。这种方式将令人赏心悦目。计算机游戏是交互式图形表达的常见例子。

本书对于上述所描述的人群，有时会使用“用户”这个词，有时会使用“观众”这个词。在某些条件下，像玩计算机游戏时，我们可能会使用“玩家”这个词。

图形学也包含输入方面的内容。程序制作、输出图片，或者结合程序中运行的算法与用户输入的数据生成图片。一些程序能够轻松接受通过键盘输入的字符和数字。另一方面，图形程序强调更自然一些的输入方式：如桌子上鼠标的移动，写字板上输入笔的线划，或者用户头和手的运动。本书将考察交互式计算机图形学的许多技巧；这就是说，我们把自然的用户输入方式与那些图形输出的制作结合起来。

1.2　计算机生成的图片用在哪里

计算机图形学能够绘制高度真实感的物体，还能绘制真实世界里看不到的事物（例如火山内部的景象，想象中的木星表面，或者住院病人的头颅内部）。程序员通过程序中的某些算法描绘感兴趣的物体，然后程序通过该模型生成图片。

本节我们将简单介绍一些计算机图形学的应用领域。这个列表是远远不全的：它仅仅起到提示作用。我们还将在实例分析中详细介绍其中的某些应用。

1.2.1　艺术、娱乐和出版行业

计算机图形学被广泛应用于电影、电视节目、书刊、游戏和杂志的制作。近几年来图形系统的成本显著下降，人们还开发出强大的软件和硬件工具来充分利用图形系统不断增长的性能。现在天才的设计者们惯于使用计算机来制造特殊的效果、动画和高质量的出版物。图 1.2 显示了一个计算机动作游戏的例子。

图 1.2　一个 3D 计算机游戏的截屏

计算机游戏的发展

许多读者学习计算机图形学是因为有潜在的兴趣想学习怎样写游戏软件。我们将逐步介绍写游戏的技巧,并概略介绍近几十年来游戏的发展。这是非技术上的总结,它揭示了计算机图形学领域正在(并也会继续)快速发展,并且使计算机游戏更加成功和更具有吸引力。

1952 年 A. S. Douglas 在剑桥大学发表了他的关于人机交互的博士论文。Douglas 发展了最早的图形化的计算机游戏之一:井字旗游戏的一个版本。1958 年 William Higinbotham 创造了另一先驱性的视频游戏,被称为"双人网球"的交互式游戏(很像 Pong),制作并展示在 Brookhaven 国家实验室的示波器上(见图 1.3)。

图 1.3 用于制作"双人网球"游戏的设备

1962 年 Steve Russell 发明了"空间战争",Nolan Bushnell 和 Ted Dabney 在 1971 年把它发展成为第一个钱币操作的电玩游戏。1967 年 Ralph Baer 写了第一个使用电视机作为显示器的视频游戏 Chase。Nolan Bushnell 和 Ted Dabney 在 1972 年成立了 Atari 计算机公司。1980 年 Atari 的 Asteroids 和 Lunar Lander 成为了最先注册版权的两款计算机游戏。Atari 公司还开发了为大家所熟悉的 Nintendo、SEGA 和 PlayStation 等游戏系统。

这只是一个简单的纵览,稍后我们将讨论如何写游戏。特别是我们会着重讲游戏中的场景建模,和程序怎样实现玩家和游戏中角色之间的交互,以及纹理的制作。纹理对于场景的真实感起着非常重要的作用。这些都要求数据更快地从 CPU 传送到显示器,以及需要显卡功能的不断增强。有一类需要很好显卡的游戏是第一人称射击游戏。1992 年推出的德军总部 Wolfenstein 3D 游戏是第一款这类游戏,它很快引发了其他的热门游戏例如毁灭战士 Doom 和雷神之锤 Quake 系列。

这些第一人称射击游戏使游戏者从主要参与者的视角沉浸于场景之中。看上去那么逼真的场景(敌人看上去相当凶恶)、被击中的效果、突然出现的阻止游戏者前进的障碍物,这些都使得游戏非常真实。与德军总部 Wolfenstein 3D 游戏相关的光线投射算法,将在第十二章探讨。

计算机游戏

图 1.2 和图 1.4 显示了两种不同类型的计算机游戏。玩家移动操纵杆、按按钮、触动扳机,计算机都必须快速的反应生成图像。一些专用设备常常被用来加速连续图像的生成。

街机游戏 Arcade games 给图形编程人员带来了最艰巨的挑战，因为动作必须高度逼真而且响应速度非常快。

图 1.4 一个德军总部 Wolfenstein 3D 的截屏

电影制作、动画和特技

电视节目经常可以见到计算机动画，一些效果惊人的动画可以被整合成特效电影。通过在胶卷或录像带上写入一系列的图像来制作动画，每个图像每一秒只有轻微的差别，人眼将图像混合就能看到流畅的动作。

通过因特网浏览

我们生活在一个网络世界里，似乎每个人都在网上冲浪。用户移动鼠标到屏幕上一点，点击选择下一个访问的网站，网页的信息就会通过互联网传送过来。浏览器必须快速解释网页上的数据，然后把高质量的文字和图形绘制到屏幕上。图 1.5 显示了万维网上的一幅截屏。

图 1.5 万维网上一个网页的截屏

幻灯片、书籍和杂志设计

计算机图形学通常利用电子排版程序来设计书刊的页面,用户可以通过交互地移动文本和图形来寻求最满意的版面。

另一种发布形式有时被称作陈述图形。高质量的商用或教育用幻灯片被设计用来给各种各样的观众演示,如图 1.6 的例示。幻灯片常常含有条形统计图表或圆形分格统计图表,以一种便于理解的方式来总结复杂的信息。它们必须是高质量且可视化的,以便于鲜明地表达观点。

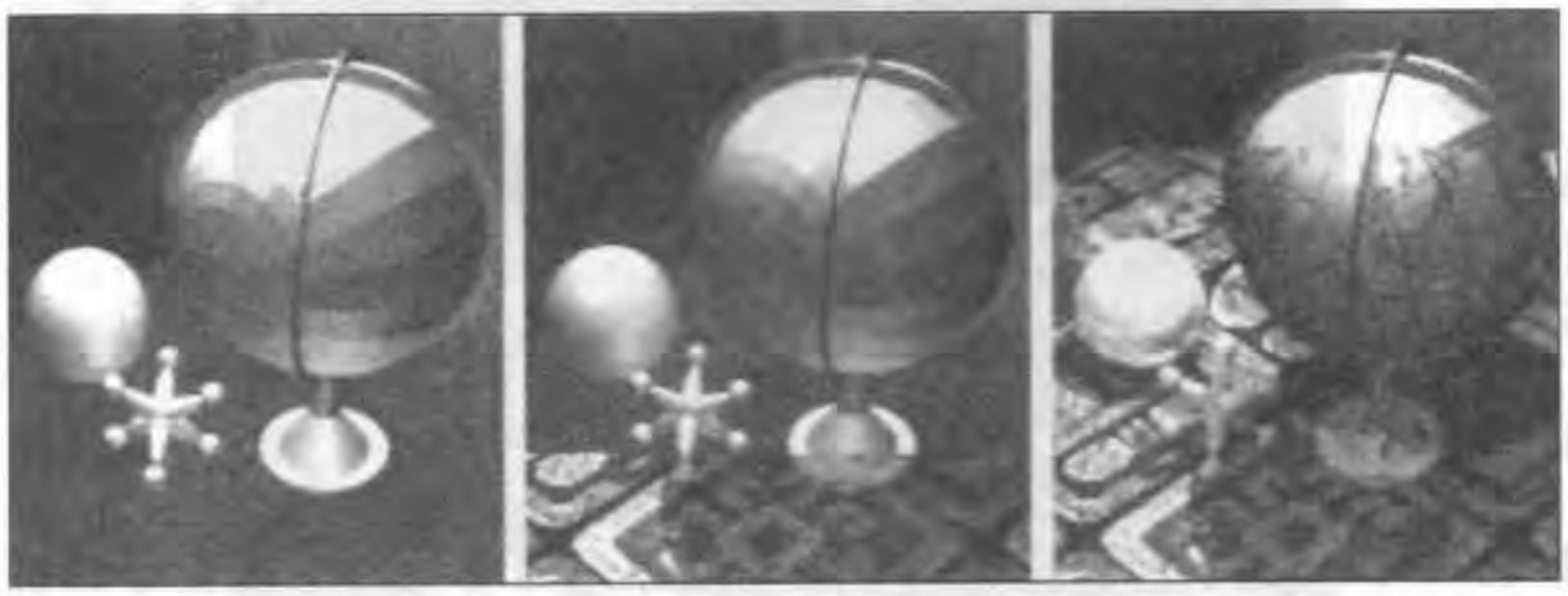


图 1.6 用于教学演示的幻灯片

绘图系统是制作图像的另一种工具。一个常见的例子就是 Adobe Photoshop,既是图像处理软件又是绘图系统。用户利用鼠标或写字板绘画草图,然后选择颜色或图案来创造想要的效果。图 1.7 显示了这样一个系统的截图。系统提供了各类工具:以前绘制的图像可以从大容量存储器中取出来和新图像混合;可以通过调色板获取并显示不同颜色;许多不同的纹理可以通过简单的命令创建。这些系统经常用来制作网上常见的那些网页。

图 1.7 一个绘图系统的截屏

1.2.2　计算机图形学、感知和图像处理

计算机图形学和图像处理的领域多年来相互融合：划分它们之间的界限变得越来越难了(也越来越不重要了)。本书主要阐述计算机图形学，但是我们也会讲一些以前属于图像处理领域的技术。

计算机图形学的主要任务就是制作图片和图像，并基于计算机的某种描述或模型合成它们。图像处理的主要任务，从另一方面说，是改善或改变已做好的图像，被修改的图像可能来自数字照片或者从视频捕获。图像处理可以去除图像中的噪声，提高对比度，锐化边缘，修正色彩。软件程序可用来搜寻图像的某些特征，增强它们的高光使图像更加引人注目或更加易于理解。

图 1.8(a)显示的是数字扫描一个照片得到的图像。图 1.8(b)显示的是提高对比度、去除噪声以及锐化边缘后得到的图像。

图 1.8　一个扫描图像的改善

(a) 原始图像；(b) 改善的图像

电影工业最近以戏剧化的方式采用了计算机图形学和图像处理的结合。想象一个经典电影中的场景——比如“海底总动员”或者“谁陷害了兔子罗杰”——就综合了计算机图像生成与数字处理技术。

1.2.3　过程监视

高度复杂的系统如工厂、发电站和空中交通控制系统必须严格监控，必须要有一个人在那里监控随时可能发生的问题并发出警报。监视器必须及时传送实时信息。每秒钟或每当需要的时候，数据就被传送到监测电台，转换成图形信息，呈现给操纵者。

空中交通控制系统由显示不同区域内飞机位置状况的一系列监视器组成。用户看到一个整体的图示。不同的图标通过闪现或改变颜色对用户发出提示或警报。

1.2.4 仿真显示

一些系统,像那些空中交通控制,确实存在并且能够实时监控。另外一些则从来不存在,如计算机中的一个算法和公式,但是它们依然可以被测试和运行,就像它们实际存在一样。通过它们可以获取有价值的模拟数据,这类信息通常使用图形表示来使它们变得易于使用。

多种系统能够被很有效地模拟:机器人往返于活火山的斜坡;人体对危险的外来物质入侵的反应;或者碳氢化合物的增长对全球变暖的影响。经典的例子就是**飞行模拟器**。这种系统是一架具有某一型号及飞行特点的模拟飞机,场景包括飞机场、山脉、植被,当然还有天空,所有的模型都能被很好地模拟。

飞机的动态运动通过计算机建模。在飞行模拟期间,当飞行员操作控制系统,程序计算出模拟飞机新的位置和速度,飞行员看到座舱外一个模拟的环境。飞行模拟器是要求最多和最难的图形应用程序,因为必须要快速反应。图 1.9 展示了一架飞机座舱的真实控制面板。

图 1.9 飞行控制面板的模拟

这些例子说明了计算机图形学的重要性:

计算机图形学能够显示物体仿佛它们真的存在一样,而实际上它们只是计算机内的模型而已。

1.2.5 计算机辅助设计

许多学科大量利用交互式计算机图形学来协助系统设计或制造——例如电钻外壳设计。计算机建立探讨中的设备模型,基于模型的图片可以展现给用户检验。设计者可以操作跟踪球或数据手套,对物体进行旋转或放大,从而对其进行查看。为了加速通常采用线框绘制,这就是为什么物体的形状往往是网格结构。

设计者可以仔细观察电钻目前的模型,然后找出需要改进的地方,修改并重新绘制。当形状看上去合适时,用户需要一个更真实的外观,这也许会花费很长时间去绘制。随着今后更深入的讨论,将介绍有提供阴影、高光和复杂细节的全彩色绘制算法。

这里也可能会用到分析和模拟。电钻的外观或许有不错的视觉效果,但外壳可能就太脆弱或太重,或者抓着很不舒服。用于电钻模型的算法通过分析它的重量来测试动配合是否适合它的外壳。算法能更进一步地测试外壳用钢铁或铝是否太昂贵,或者内部的部件在

最终的加工工序中是否便于组装。计算机可与设计制造业有力地结盟。本书关注的是根据设计者提供的准确信息制作出物体的图像。

计算机辅助建筑设计

计算机图形学也能够帮助建筑师设计建筑。这个模型可能是一座房子、学校或者医院的建筑平面图，如图 1.10 所示。建筑师能够调整这个数字平面图，用鼠标把一堵墙移到这里或者把一个窗户调整到那里。CAD 软件也可以制作一个完全的绘制版本，结构以三维形式呈现。计算机图形学允许建筑师移动窗户和门的位置，展现不同的纹理如砖或灰泥。通过交互式的控制，建筑师甚至能够做一个预演，给顾客展示房子建好后的样子。

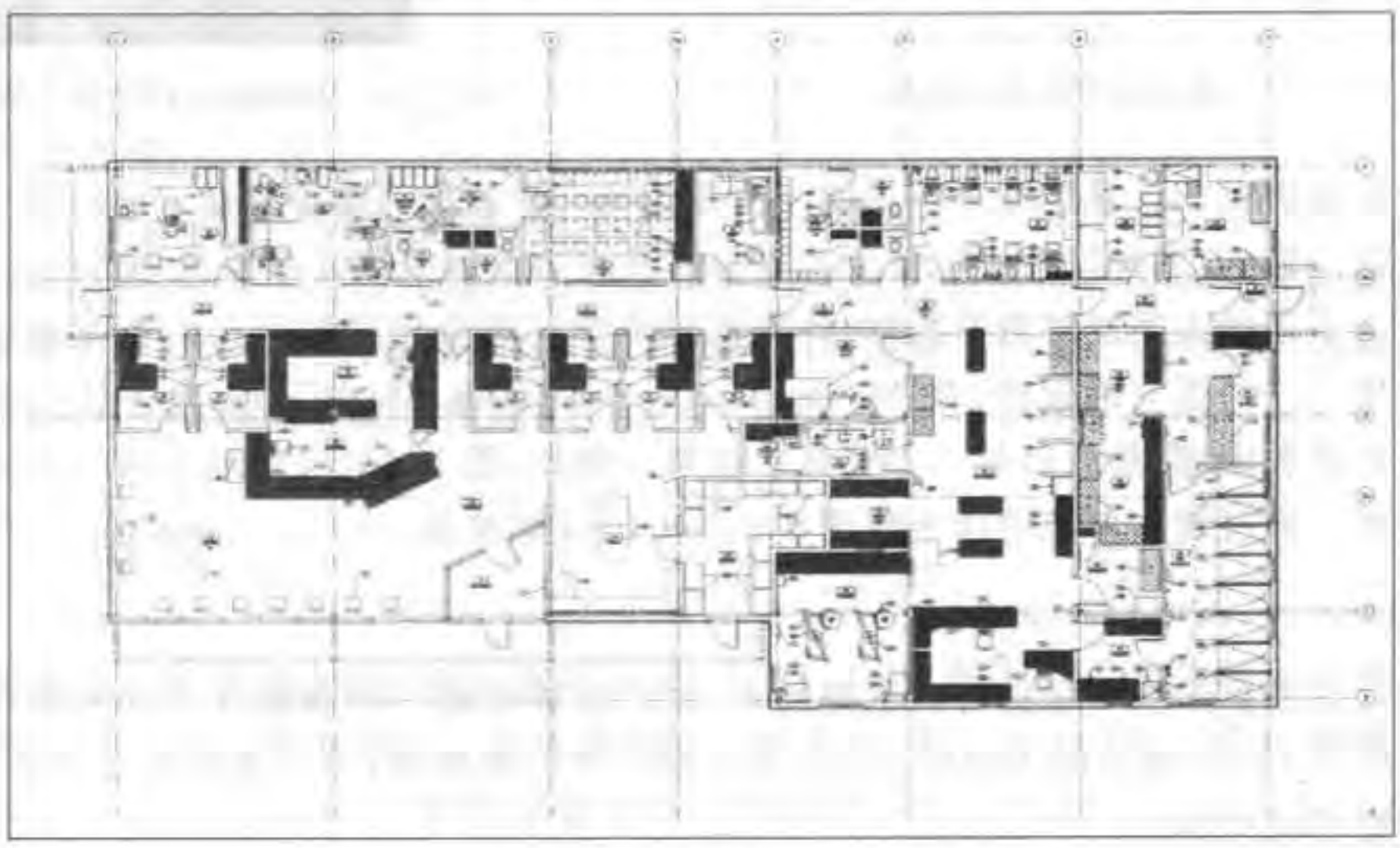


图 1.10　一个医院的平面图

1.2.6　科学分析与体可视化

科学数据往往是复杂的，一个实验中变量之间的关系难以可视化。图形学提供了一个极好的工具把科学信息以一种便于理解的方式表达出来。如果能以正确的方式观察数据，往往就会对正在进行的研究有新的发现。通过适当的方式展示数据，还有利于与同行更好的交流。

图 1.11 举了一个例子：一张曲面展示了人眼敏感的波动方式。表面高度代表一个数量（例如温度或黏度），三维点的另外两项代表其他的数值。与之相对的是，如果这些数据用一个表格的形式展现，那人们将不得不努力研究这个表格以期获得同样的信息。

体可视化

人体头部扫描是一个复杂的难题，但如果利用图像处理就变得简单了。不同颜色的区域及时提示医生关于大脑每个部位的健康情况。一个像大脑这样的三维物体图像可通过创建一系列切片合成为最终图像如图 1.12 所示。

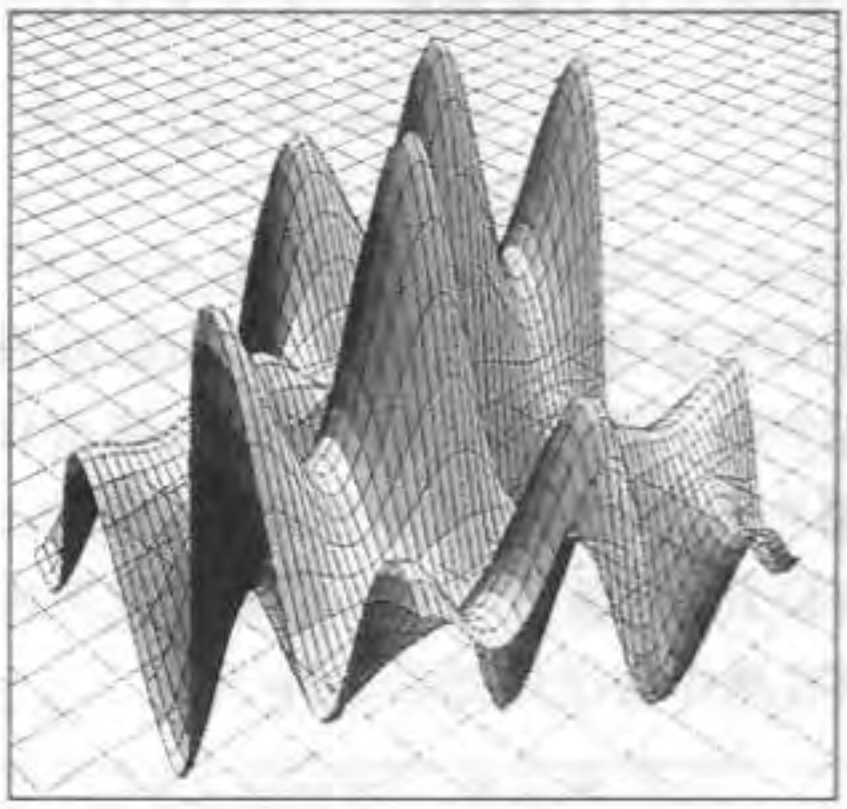


图 1.11 复杂科学数据的显示

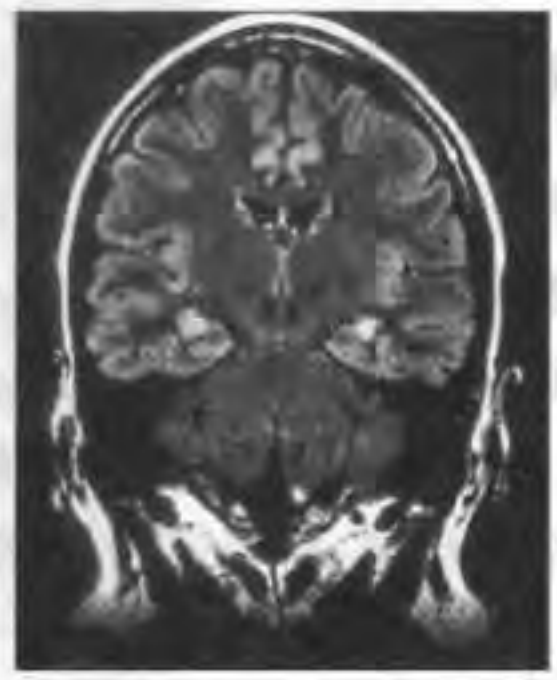


图 1.12 利用体可视化的人脑视图

这种方法引发了计算机图形学里的一个专门研究领域称为**体可视化**：它用图形来展示海量的数据。例如，多种软件可以辅助识别分析数据中的天气模式来预测恶劣的天气。

另外为了帮助人类理解测量数据，计算机图形学还适合于理解复杂的数学思想。人们已经设计出了一些强大的软件(如 Mathematica、MathLab、MathCad)，允许使用者把各种数值输入公式和定理，然后以某一方式显示结果。例如，图 1.13(a)显示了由一个数学公式生成的表面。通过键入单独的指令使用 Mathematica 来显示：

```
ParametricPlot3D[{t,u,Sin[t u]},{t,0,3},{u,0,3}]
```

这个公式的复杂性一眼就可以看得出来，用户能够调整一些参数从不同的视角来观察其表面。图 1.13(b)也采用 Mathematica 来绘制一个三维物体(一个星形的二十面体)来帮助用户研究它的结构。

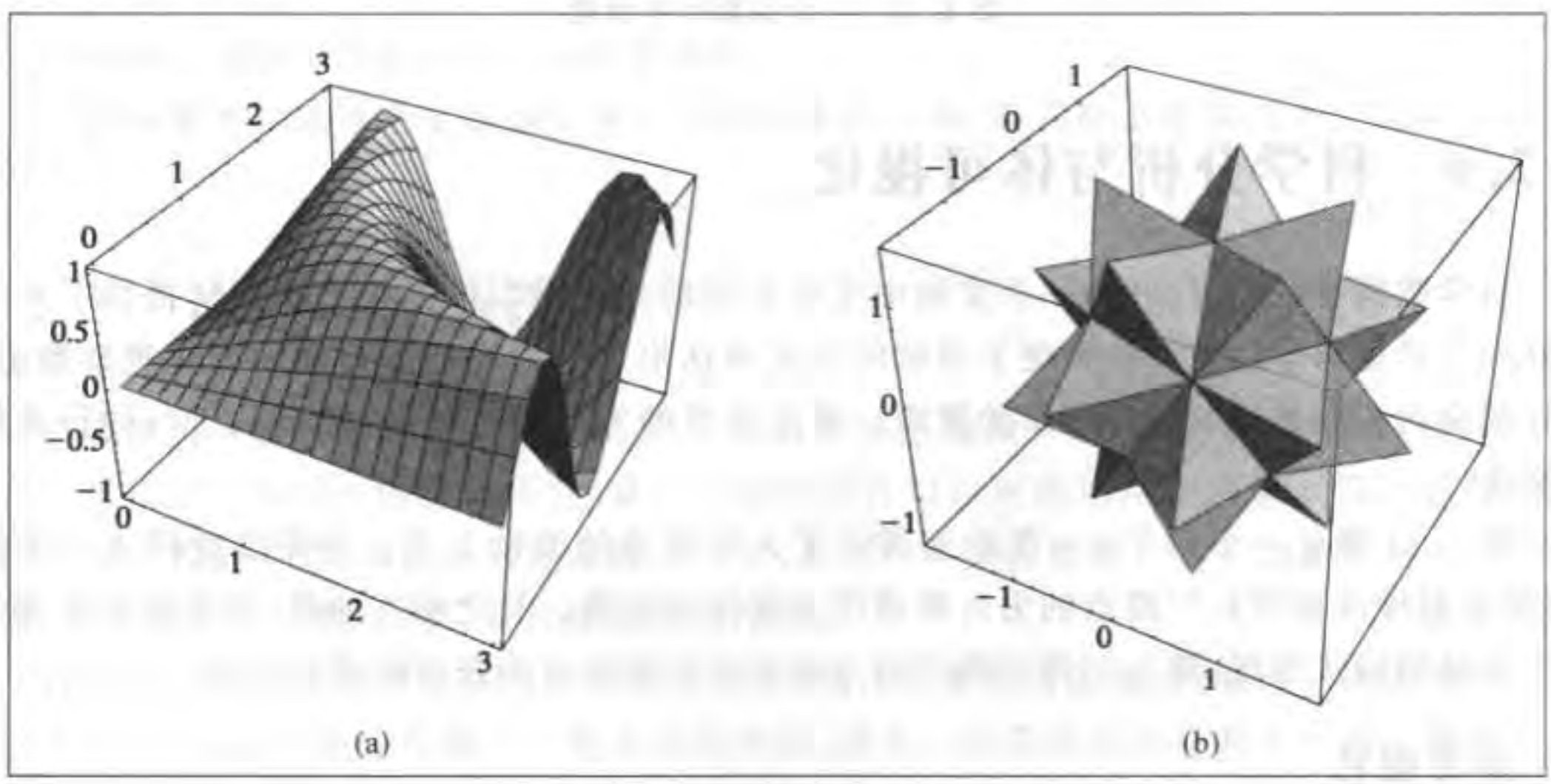


图 1.13 Mathematica 的显示

(a) 一个复杂的数学曲面；(b) 一个数学定义的实体

1.3　计算机图形学中制作图像的基本元素

计算机绘制的图像是由什么组成的？组成这些图像的基本元素被称为输出图元。一种有效的分类是：

- 点
- 线
- 折线
- 文本
- 填充区域
- 光栅图像

我们会发现这些类型有些重叠，但是这种分类提供了一个很好的起点。我们依次描述每种类型的图元，以及一些常用程序如 OpenGL 绘制它们的语句。这些工具更多的细节将在后面章节给出；当然，我们也探讨每个输出图元不同的属性。一个图形图元的属性就是那些影响外观的特点，例如色彩和粗细。

1.3.1　折线

折线是由一系列的直线段相互连接而成。图 1.14 中每个例子都包括了一些折线：图 1.14(a)所示为一条从恐龙鼻子一直延伸到尾巴的折线；图 1.14(b)用一条折线表示的数学方程图；图 1.14(c)所示为一个国际象棋棋子的线框图，它包含了许多折线来勾画它的轮廓。

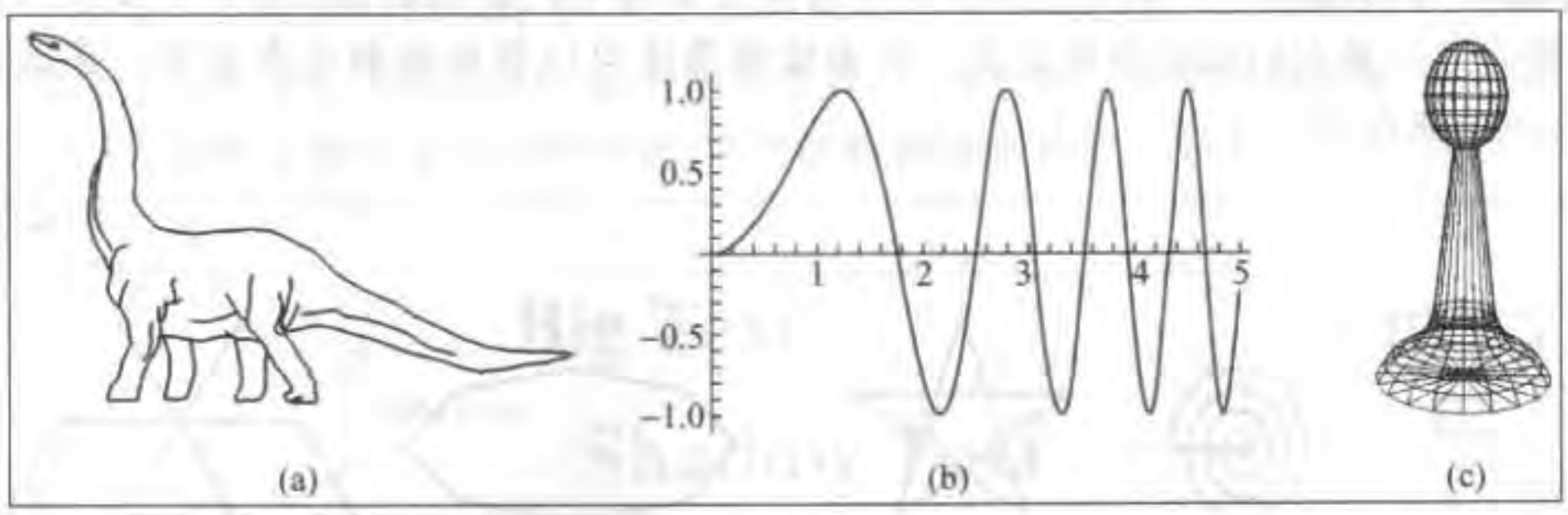


图　1.14

(a) 一幅恐龙的线条画；(b) 一个数学公式的图；(c)一个三维物体的线框图

注意一条折线可以看起来像平滑的曲线。图 1.15 显示了一个放大的曲线，能够看出它是由许多线段组成的，人眼把它们混合成一条光滑的曲线。

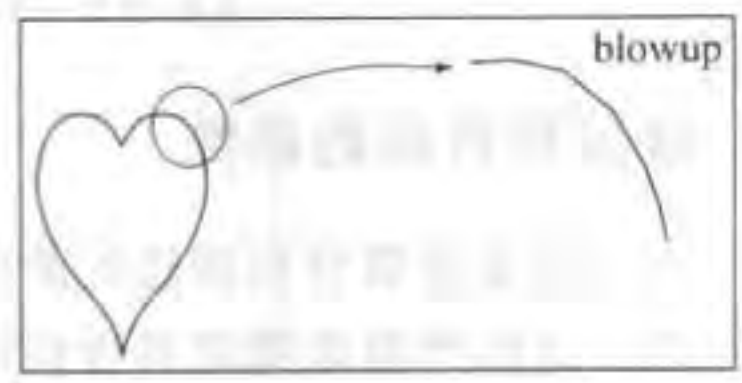


图 1.15　一个由直线段构成的曲线

由折线组成的图有时称为线条绘画(素描)。某些设备，像笔式绘图仪，就是专门为制作线条绘画而设计的。

最简单的折线是一条单独的直线段。一条直线段用它的两个端点定义,记为(x_1, y_1)和(x_2, y_2)。画一条直线的程序类似于

```
drawLine (x1, y1, x2, y2);
```

它画了两个端点间的一条直线。以后我们将详细描述这类的语句,并演示使用它们的大量例子。在某一点,我们可以定义像x_1(整数或实数)一样的坐标以及它的颜色。

一个特例是当一条线段收缩为一个点时就被画作一个点。甚至低级的点在计算机图形学中也有重要的用途,正如我们后来将看到的那样。画一个点可以使用如下的程序语句

```
drawDot (x1,y1);
```

但是,正如将要看到的,OpenGL 画点会有些不同。

当若干线段组成折线时,每条线段叫做边,两条相邻的边相交于一个顶点。折线的边可以相交,如本节中的那些图所示。折线可由顶点列表定义,每个点由坐标确定:

$$(x_0, y_0), (x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$$

例如,图 1.16 所示的折线由以下坐标序列给定:(2,4),(2,11),(6,14),(12,11),(12,4)…

画折线我们需要一个工具,如下面的一行代码:

```
drawPolyline (poly);
```

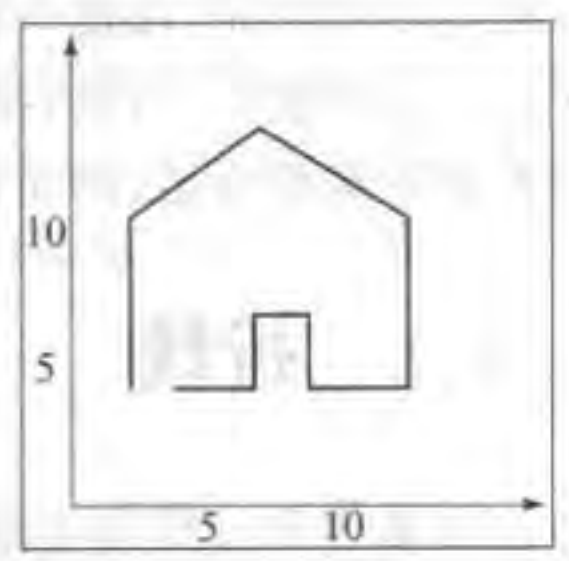


图 1.16 一个折线的例子

变量 poly 是以某种方式存贮所有顶点的列表。在一个程序中有多种方法来描述一个列表,每种方法都有各自的优缺点。

折线不需要形成一个闭合的图形,但是如果首尾顶点被同一条边相连,它就形成了一个多边形。另外,如果任意两条边都不相交,这种多边形就被称为简单多边形。图 1.17 展示了一些有趣的多边形;只有图 1.17(a)和图 1.17(b)是简单多边形。多边形是计算机图形学的基本元素,一部分原因是它们很容易被定义。许多绘制算法可以很好地操作多边形。多边形将在第 3 章中详细介绍。

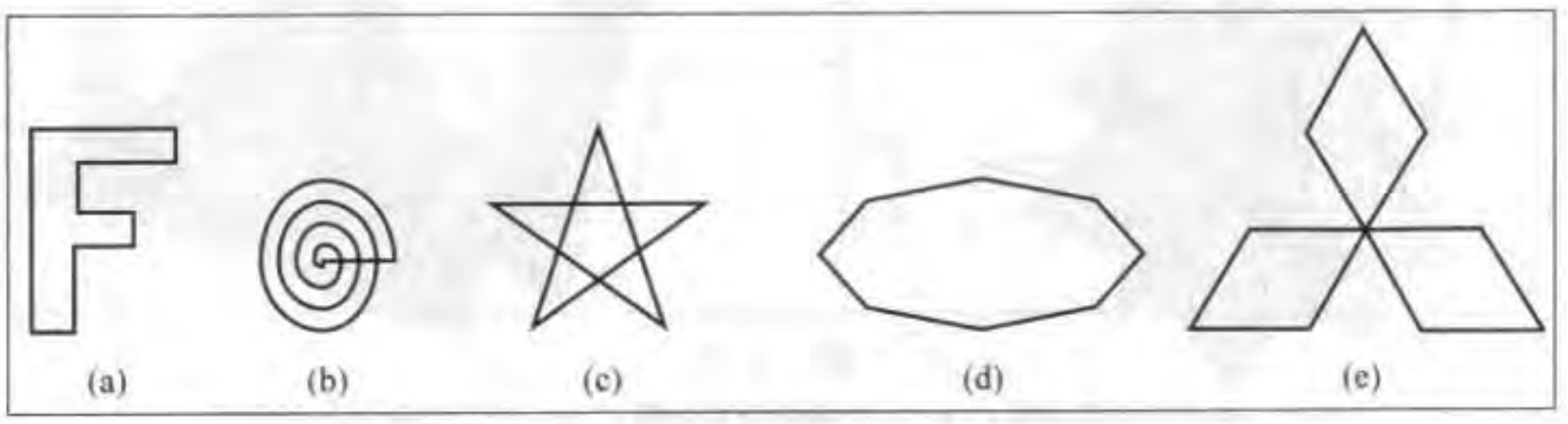


图 1.17 多边形的例子

线段和折线的属性

折线的重要属性包括边的颜色和粗细,虚线的方式,粗边在端点混合的方式等。一般来说一条折线的所有边都应具有相同的属性。

图 1.18 中的前两条折线有着粗细的不同。第三条折线是采用虚线方式绘制的。

图 1.18　不同属性的折线

当一条线的属性为粗，它的端点是有形状的，用户必须描述两条相邻的边如何结合。图 1.19 展示了各种可能性。图 1.19(a)显示了“平头端”的线会在接头处留下不好看的“裂缝”。图 1.19(b)显示了圆头端的线，连接处平滑。图 1.19(c)显示了斜角连接，图 1.19(d)显示了平整的斜角连接。一些软件包允许用户选择不同连接类型。

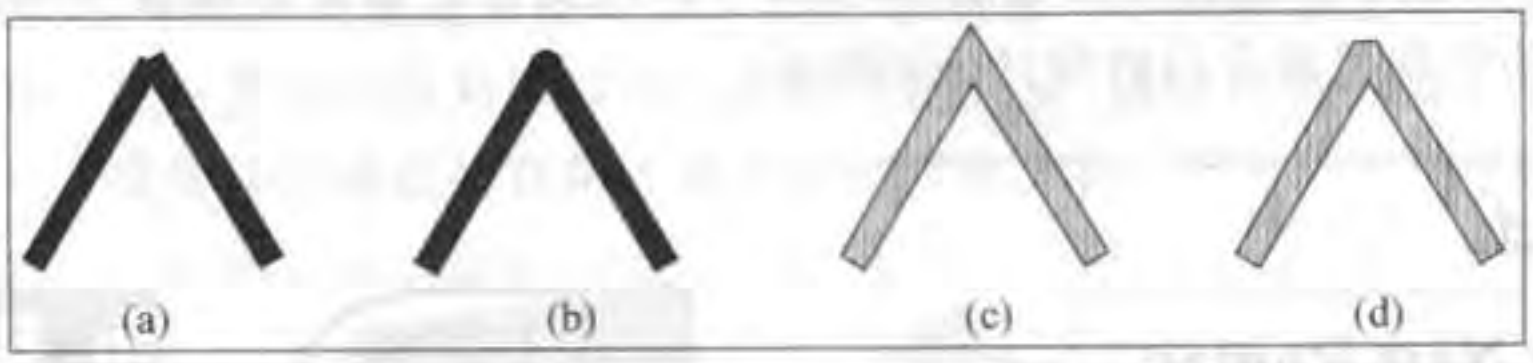


图 1.19　在一条折线中两条粗线段的连接方式

线的属性有时可通过下列命令设置：

```
setDash (dash 7)  或  setLineThickness (thickness)
```

1.3.2　文本

一些图形设备有两种不同的显示模式，**文本模式**和**图形模式**。文本模式用于字符的简单输入/输出，用来控制操作系统或者编写代码。这种方式显示的文本采用嵌入式的字符发生器。字符发生器能够绘制字母型、数字型和标点型字符，以及一些特殊符号集如♥、δ 和⊕。一般来说这些字符不能在显示器上任意放置，只能放置在规则网格的某一行和列上。

图形模式提供了更丰富的字符形状，字符能够被随意放置。图 1.20 是以图形方式绘制文本的例子。

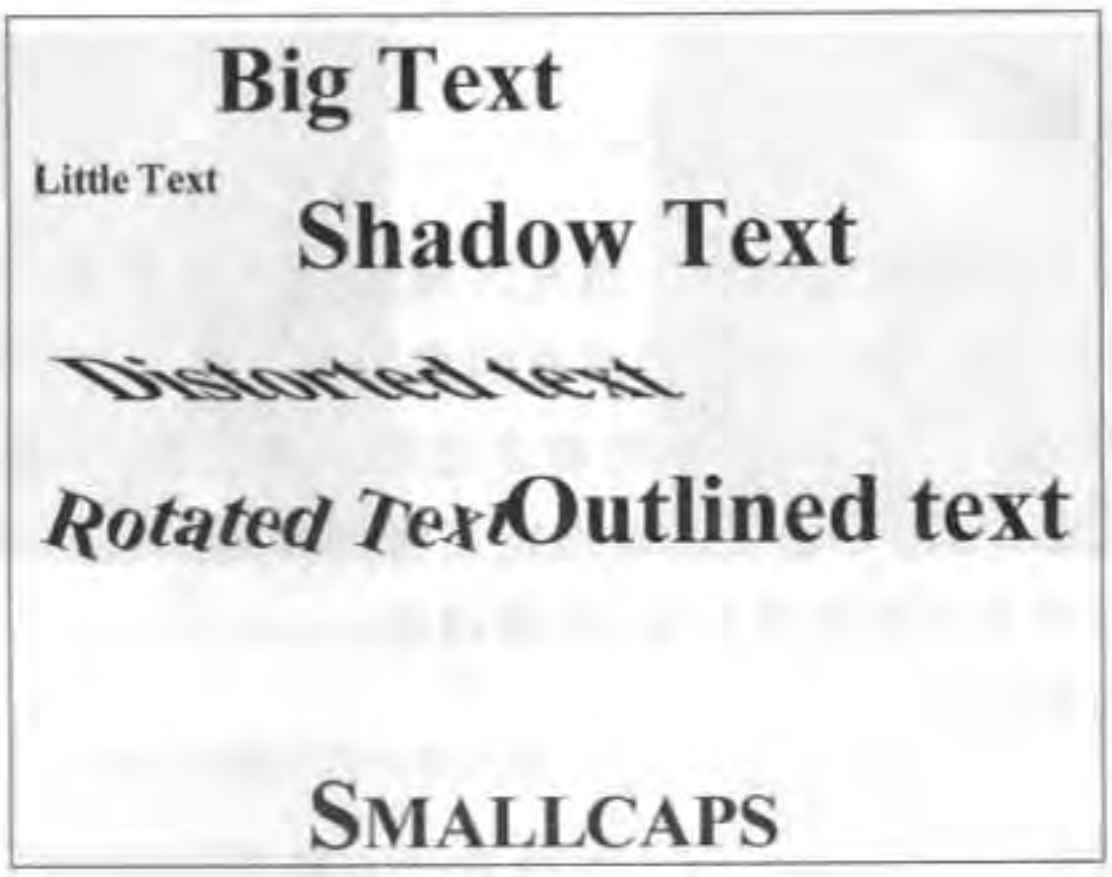


图 1.20　图形模式绘制的文本

绘制字符串的语句类似于：

```
drawString (x, y, string);
```

字符串的起点在(x, y)，绘制的系列字符存储在变量 string 中。

文本属性

文本有许多属性，其中最重要的是字体、色彩、大小、间距和倾斜度。

字体。字体是一类具有特殊风格和大小的字符形状集。图 1.21 中显示了不同的字体。如图 1.22(a)所示，每个字符的形状能够用折线表示(或者用更复杂的曲线表示，如第 11 章中贝塞尔曲线)，或通过点阵表示，如图 1.22(b)所示。图形包配置有预先定义好的字库，其他类型的字库可从专门的设计公司购买。

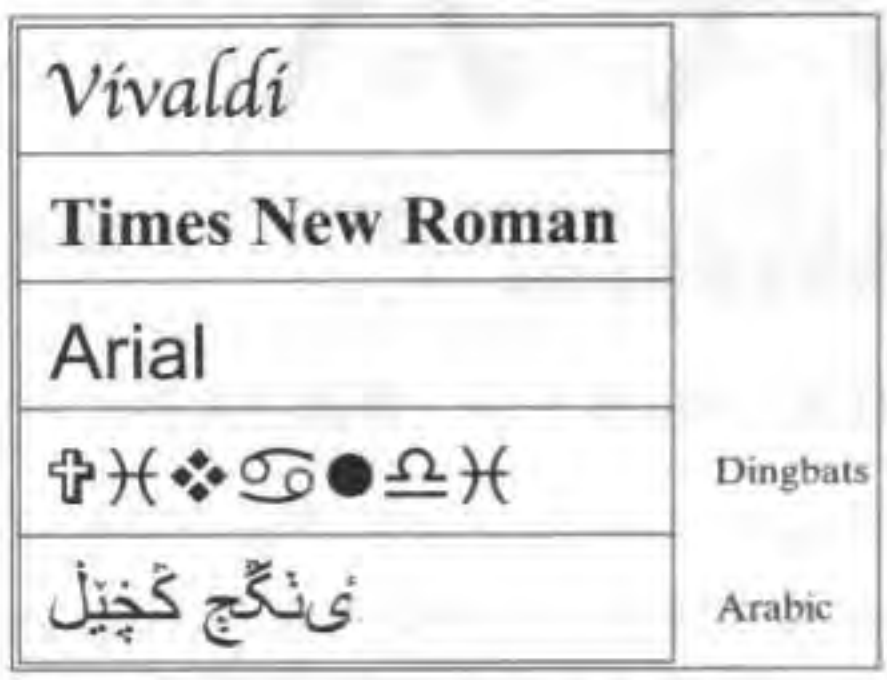


图 1.21 一些字体的例子

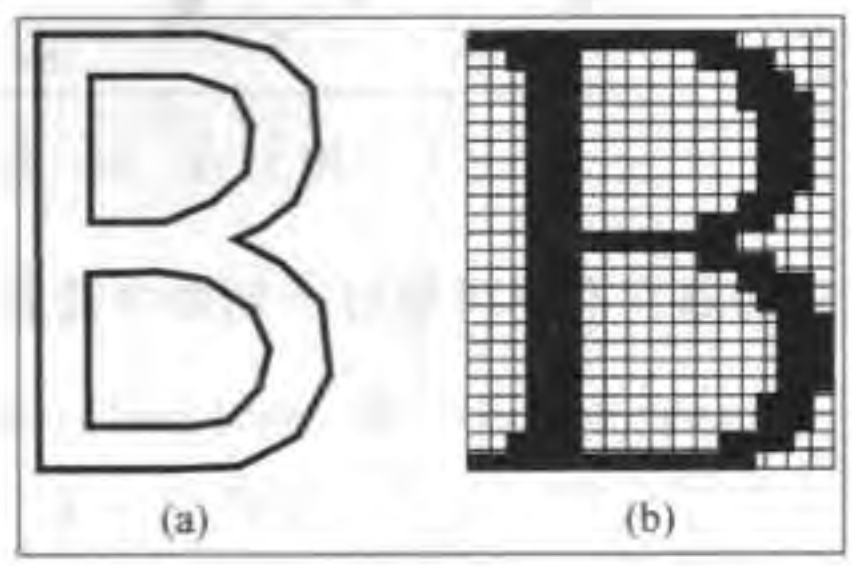


图 1.22 字符形状通过(a)折线和通过(b)点阵模式来表示

字符和字符串的倾斜度。字符可以沿着某一方向倾斜。倾斜的字符串常用来注释图形中的某一部分。

高质量文本的图形表达是一个复杂的研究对象。细微的差别就能将美观的文本变得丑陋。实际上，我们在日常生活中看到了太多的印刷品，以致于我们潜意识里希望字符显示保持特定的形状、间距和细微的平衡。

1.3.3 填充区域

填充图元是指填充时的颜色或图案。填充区域的边界经常是一个多边形(更复杂的区域将在第 4 章讨论)。图 1.23 演示了一些填充的多边形。多边形 A 的边界可见，与之相反 B 的边界没有绘制。多边形 C 和 D 不是简单多边形。多边形 D 甚至包含了多边形的洞。这样的形状依然能够被填充，但一定要准确地定义哪部分是多边形的内部，因此填充算法的区别依赖于定义。执行填充的算法将在第 10 章讨论。

填充多边形的语句类似于：

```
fillPolygon (poly, pattern);
```

其中变量 poly 保存多边形的数据(与表示折线一样的列表)，变量 pattern 是填充图案

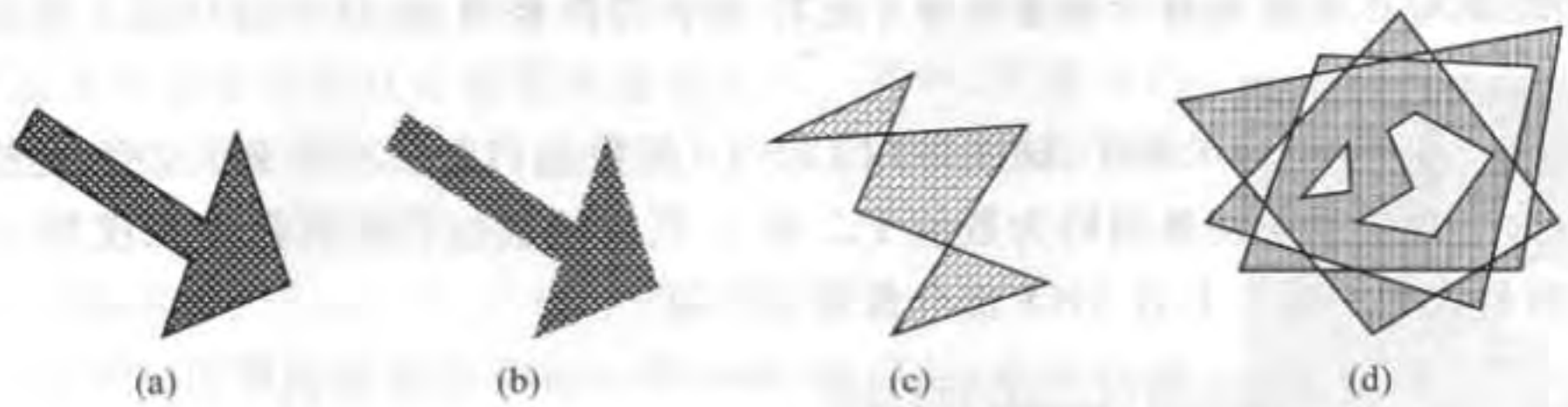


图 1.23　填充的多边形

的描述。我们将在第 4 章详细讨论。

图 1.24 展示了如何利用填充来给一个三维物体的每个不同的面着色。这个物体的每个多边形面填充了某一灰度色以对应该面反射的光线数量，这使得物体看上去沉浸于某一方向的光线中。三维物体的着色将在第 8 章中详细讨论。

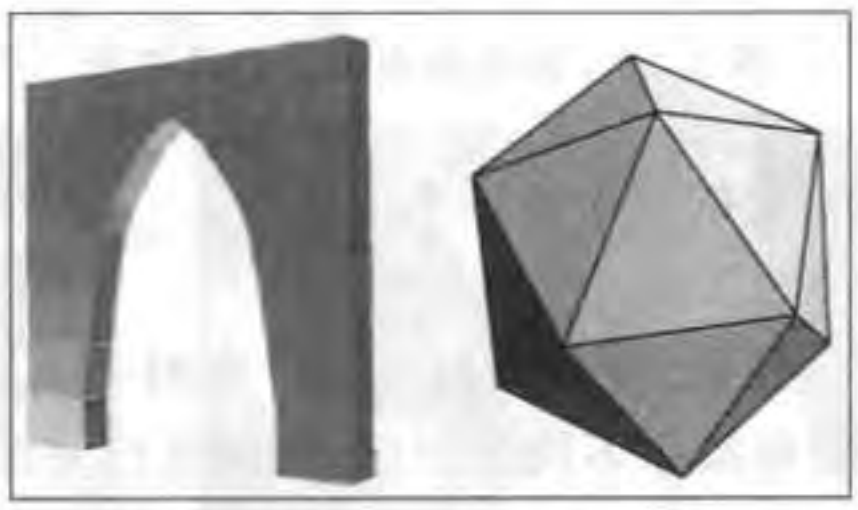


图 1.24　三维物体的多边形(面)填充以显示正确的着色

填充的属性包括边界的属性，以及填充的图案和颜色。

1.3.4　光栅图像

图 1.25(a)展示了一个国际象棋棋子的**光栅图像**。它由许多非常小的不同灰度的单元组成，其放大的版本如图 1.25(b)所示。每个单元被称为**像素**("图像元素"的简称)。一般来说人眼不能分辨这些独立的单元，它们混合在一起组成一幅图像。

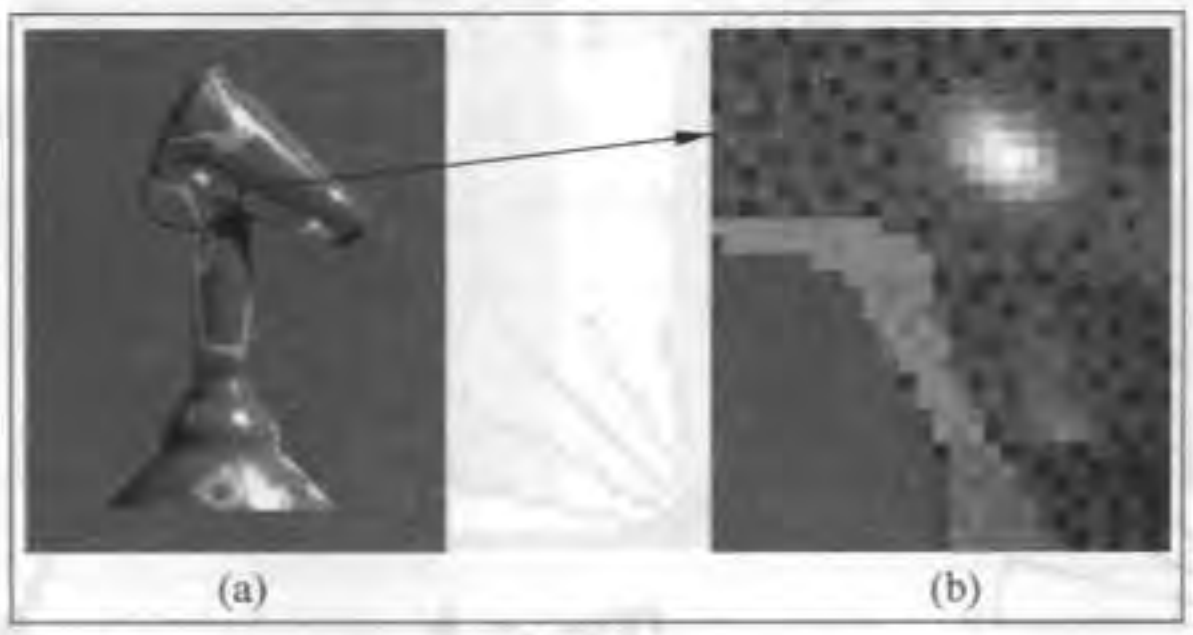


图　1.25

(a) 一个棋子的光栅图像；(b) 图像的局部放大图

光栅图像以数值数组的形式存储在计算机中。这个数组可以看成是矩形的，由某些行和列组成，每个数值代表存储的像素值。这个数组整体被称为一个像素图，或称为**位图**(也

有人认为位图应该指那些每个像素由单个比特表示的像素图,也就是由 0 或 1 组成的像素图)。

图 1.26 举了一个简单例子,该图像用 18×19 的数组(18 行×19 列)表示,包括 3 种灰度。假设这 3 种灰度级别被编码为数值 1、2 和 7,更高的数值代表更深的灰度级。图 1.26(b)表示的是这个图像左上方 6×8 部分像素图的值。

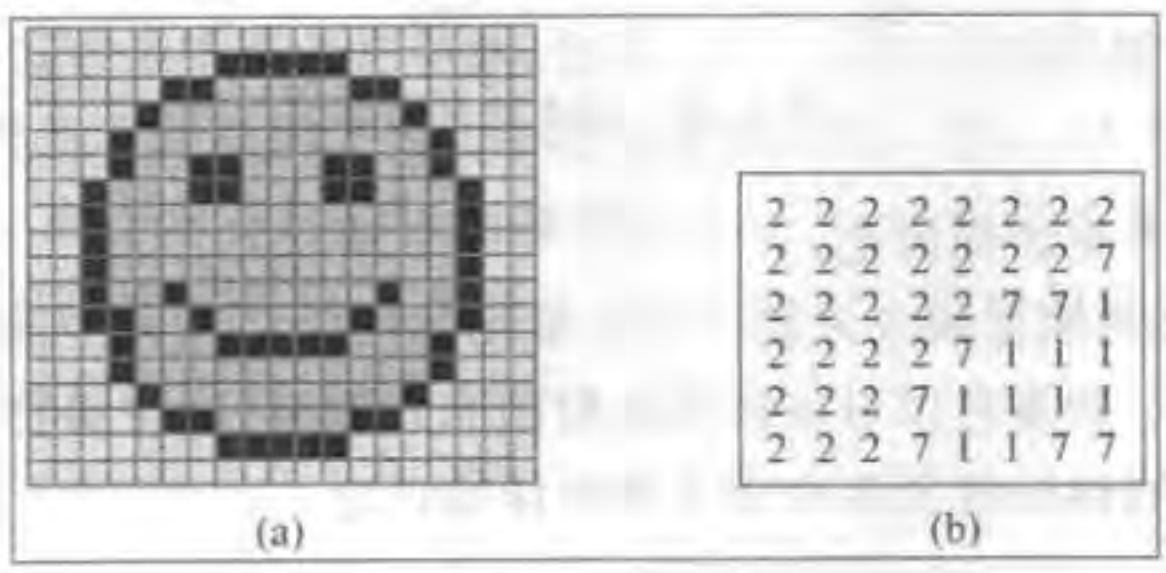


图 1.26 用位图表示的简单图形

光栅图像是怎样制作的?它的三个基本来源如下。

(1) 手工设计图像。

设计者决定每个单元所需的值,把它们记下来。有时一个绘图软件可以帮助如下工作更加自动化:设计者能够绘制和操作不同的图形,并浏览已完成的结果。当满意时,设计者把结果储存在一个文件里。图 1.26 就是用这种方式创造的。

(2) 计算机生成图像。

用一个算法来绘制一个在计算机内存中抽象建模的场景。举一个简单的例子,一个由桔黄色灯光照明的场景会包含一个独立的黄色球体。模型描述球体的大小和位置、光源的位置和一个假设的相机。光栅图像相当于相机的胶卷。为了制作这个光栅图像,算法必须计算落在相机胶卷上每个像素的色彩。这就是光线跟踪的图像生成方式,图 1.25 中的棋子就是用这种方式生成的。

光栅图像也经常会包含含有直线的图像,在这种图像中可以通过设置像素的正确颜色来表示直线。但是它需要大量的计算来确定那些最能拟合给定两个端点的理想直线的像素序列。Bresenham 算法提供了一个确定这些像素的有效方法。

图 1.27(a)展示了一个由若干直线,一段圆弧和一些文本字符组成的光栅图像。图 1.27(b)展示的是这个光栅图像的局部放大图,以便看清直线上单独的像素。对于水平

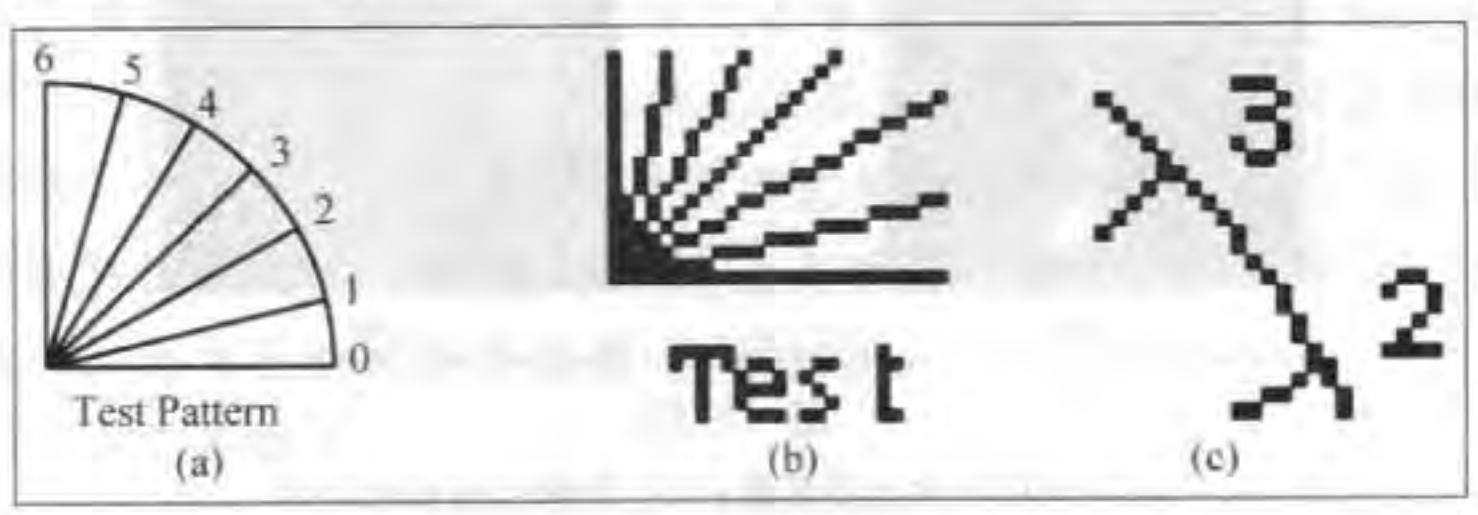


图 1.27

(a) 线和文本的集合;(b) 图(a)的放大图,产生了锯齿;(c) 放大到每个像素都清晰可见

或垂直的直线，黑色的方块像素排列得很好，形成了一条整齐的线。但对于其他的直线和圆弧，最佳像素集合表示的只是所需曲线的近似。另外，结果中的**锯齿**是光栅图像无法避免的。当视图放大到像素可见时，图像就会表现出图 1.27(a)的**失真**状况。

(3) 扫描图片。

照片或视频图像可以被数字化。将网格(虚拟的)放置在原始图像上，在每个网格点上设备读取匹配的最相近的颜色，然后将得到的位图储存在文件里以备后用。图 1.28 中小猫的图像就是用这种方式生成的。

图 1.28　一幅扫描的图像

因为光栅图像只是简单的数值数组，计算机能够很好地处理它们。例如，图 1.29 中显示了小猫图像的三幅连续放大图。这些是通过像素复制得到的(将在第 9 章讨论)。在图 1.29(a)的每个方向，每个像素被复制 3 次，图 1.29(b)为 6 次，图 1.29(c)为 12 次。

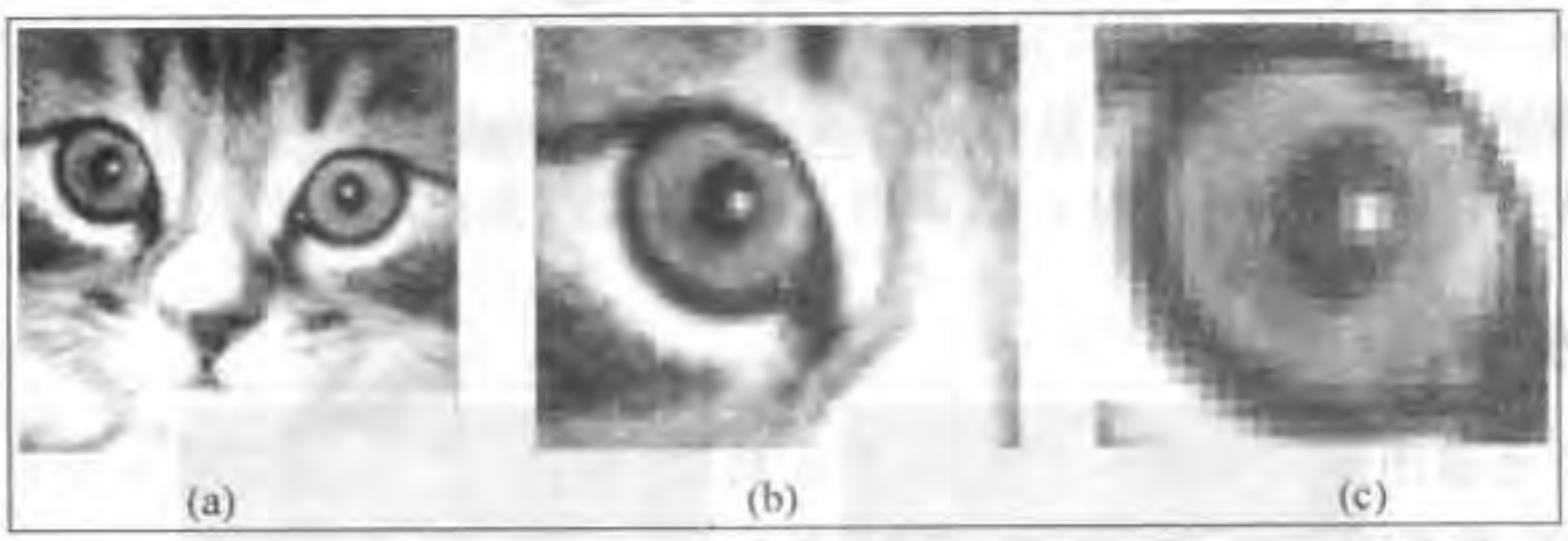


图 1.29　小猫图像的 3 幅连续放大图

(a) 3 倍放大；(b) 6 倍放大；(c) 12 倍放大

另外一个例子是，人们经常需要整理或者修改扫描图像，例如，去除噪声或突出重要细节。图 1.30(a)演示了改变小猫图像的灰度级来提高对比度从而使细节更加清楚的结果；图 1.30(b)演示了边界锐化的效果，这是通过一定形式的图像过滤获得的。

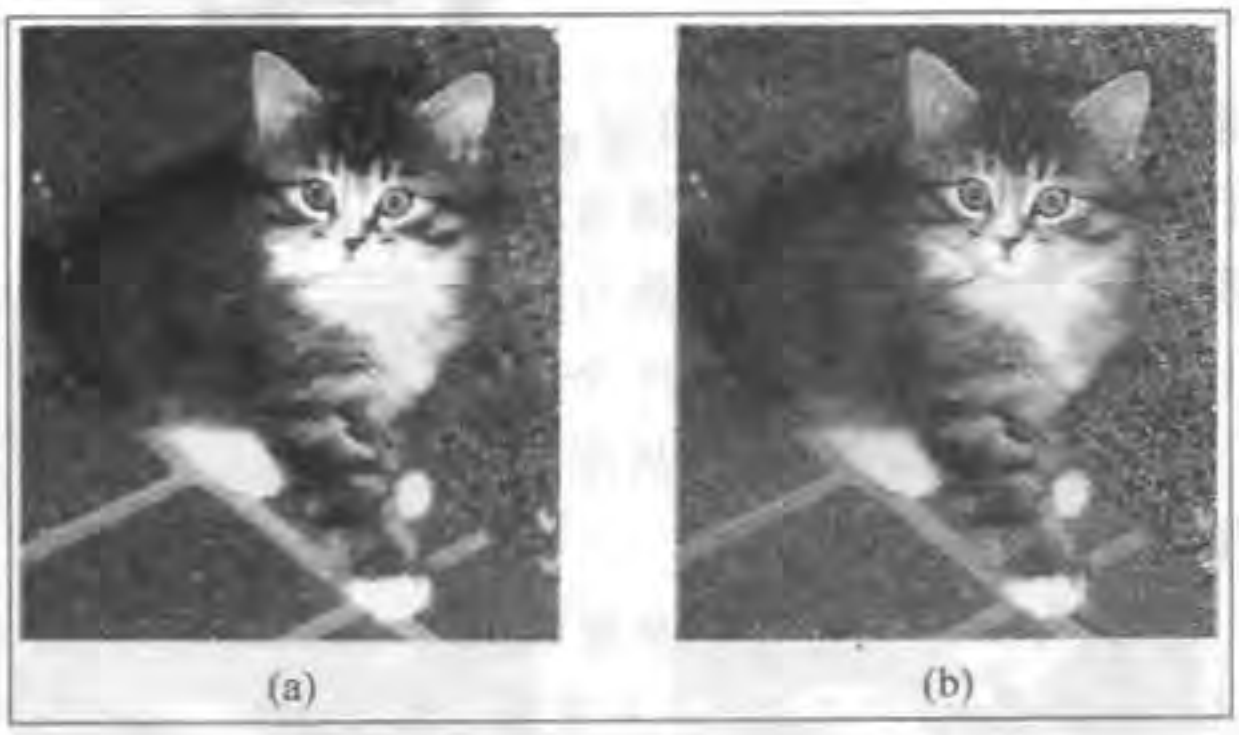


图 1.30　图像增强的例子

图 1.31 演示了编辑图像达到某一视觉效果的两个例子。图 1.31(a)展示了小猫图像的浮雕效果，图 1.31(b)展示该图的几何变形。

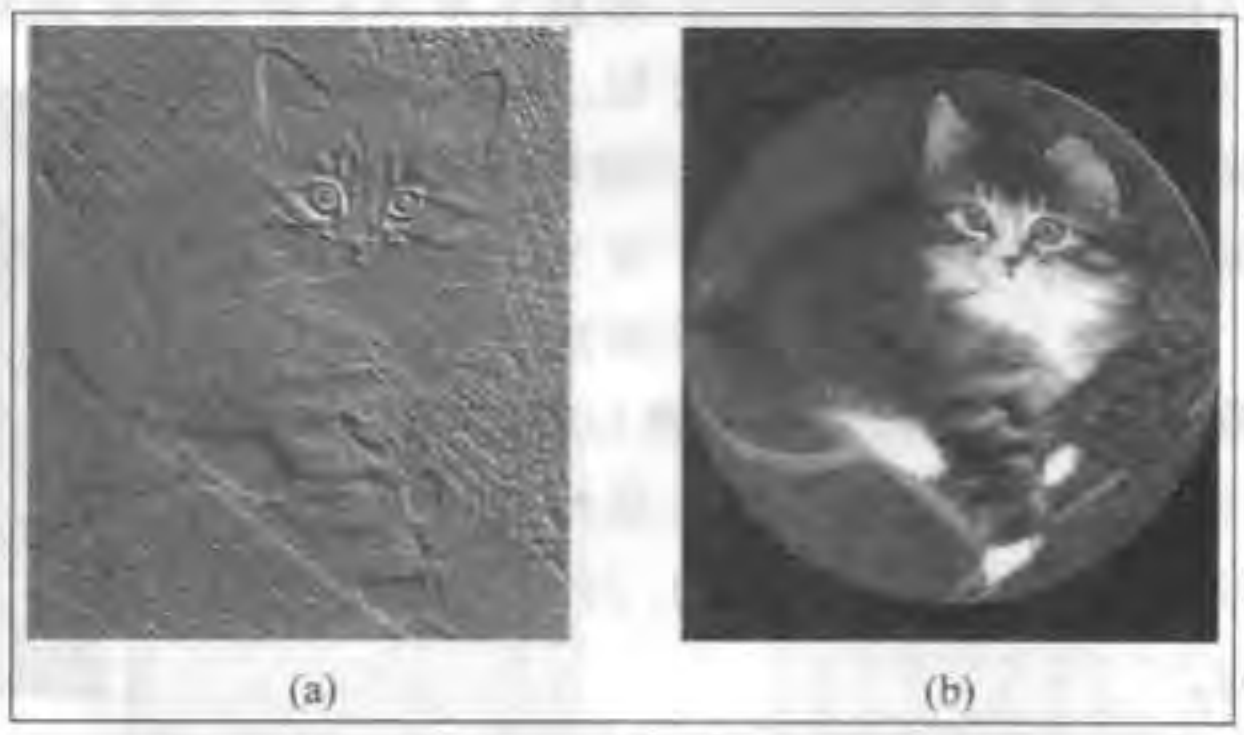


图 1.31
(a) 浮雕图像;(b) 几何变形的图像

1.3.5 光栅图像的灰度和色彩表达

光栅图像的一个重要方面就是位图的色彩或灰度的表现方式。这里我们简要概述最常用的几种方式。

灰度光栅图像

如果一个光栅图像只有两种像素值(如 0 和 1),就称为**二值图**。图 1.32(a)是一个简单的二值图,描绘的是计算机屏幕上常见的箭头形状的光标。它的光栅包括 16 乘以 8 个像素。图 1.32(b)是这个图像 1 和 0 排列的位图。这里 1 代表黑色,0 代表白色,但这种代表值可以被轻易反过来。既然 1 比特的信息对于区分两个值是足够用的,一个二值图常常被称为**每像素 1 位**的图像。

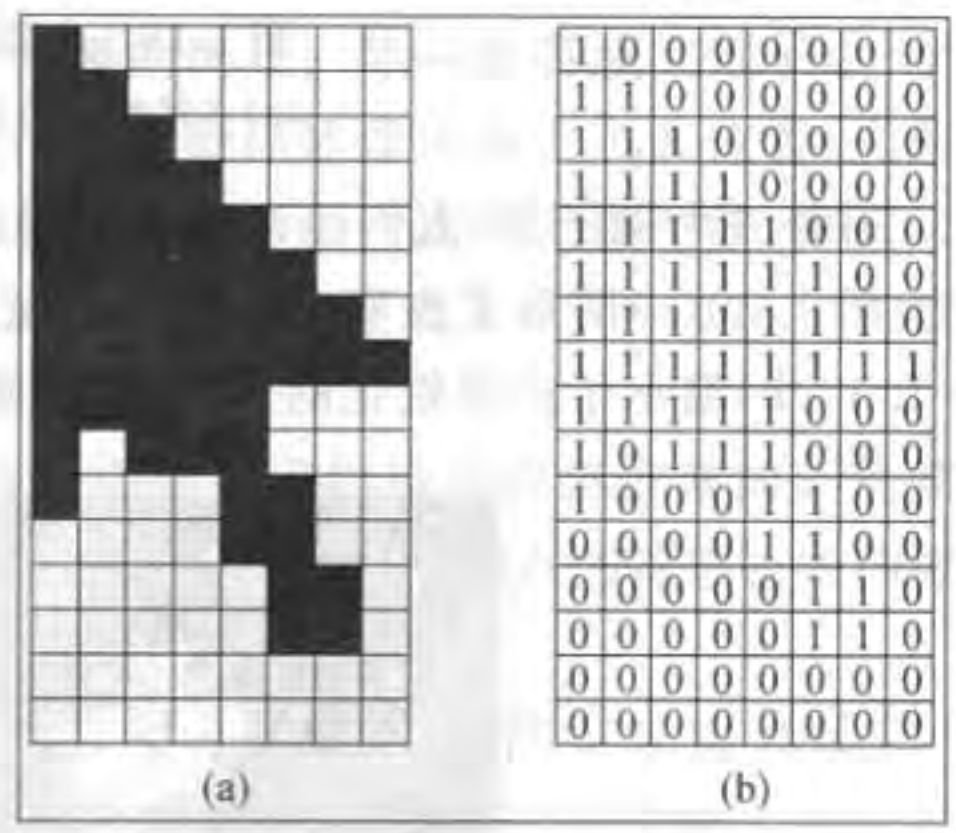


图 1.32
(a) 一个光标的二值图;(b) 它的位图

当灰度图的像素具有 2 个以上的数值时,每个像素就需要大于 1 位的存储量。灰度图常常按照它们的像素深度来划分,它需要的数目代表它们的灰度级。n 位的容量可以表示 2^n 个值,所以像素深度为 n 的图像可以有 2^n 个灰度级。最常见的 n 值有:

- 每像素 2 位 (2bits/pixel)产生 4 个灰度级。
- 每像素 4 位 (4bits/pixel)产生 16 个灰度级。
- 每像素 6 位 (8bits/pixel)产生 256 个灰度级。

图 1.33 显示了从黑到白的 16 个灰度级。16 级中每个像素值与一个二进制的 4 位数相对应,如 0110 或 1110。这里 0000 代表黑色,1111 代表白色,其他 14 个值代表中间的灰度级。

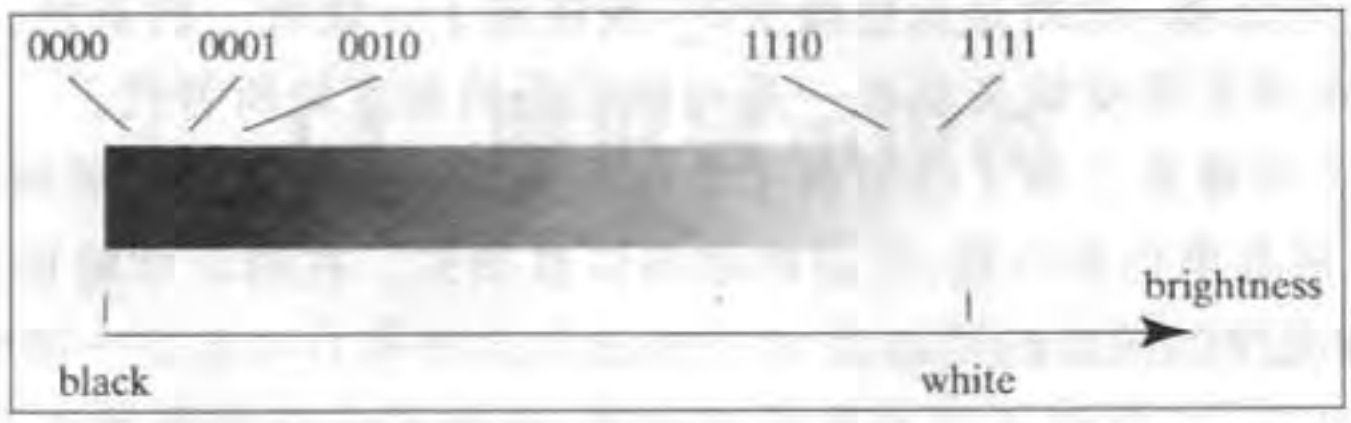


图 1.33　16 级灰度

许多灰度图像采用 256 个灰度级，因为 256 级已经可以较好表示扫描图像。256 级意味着每个像素具有某个 8 比特的值，如 01101110。像素值通常代表亮度，这里 00000000 代表黑色，11111111 代表白色，10000000 代表中间的灰色。

像素深度的影响：灰度量化

有时一个原本使用每像素 8 比特的图像会改变为每像素使用更少的比特数。如果一种显示设备不能显示一个很高的灰度级别，或者整幅图像占据了太多存储空间，这种情况就可能会发生。图 1.34 和图 1.35 显示了小猫图像的像素值用更少位数表示时的影响。图 1.34 中逼真度的损失几乎看不出来，该图分别采用 6 或 3 比特/像素（各自产生 64 和 8 个不同的灰度级）。

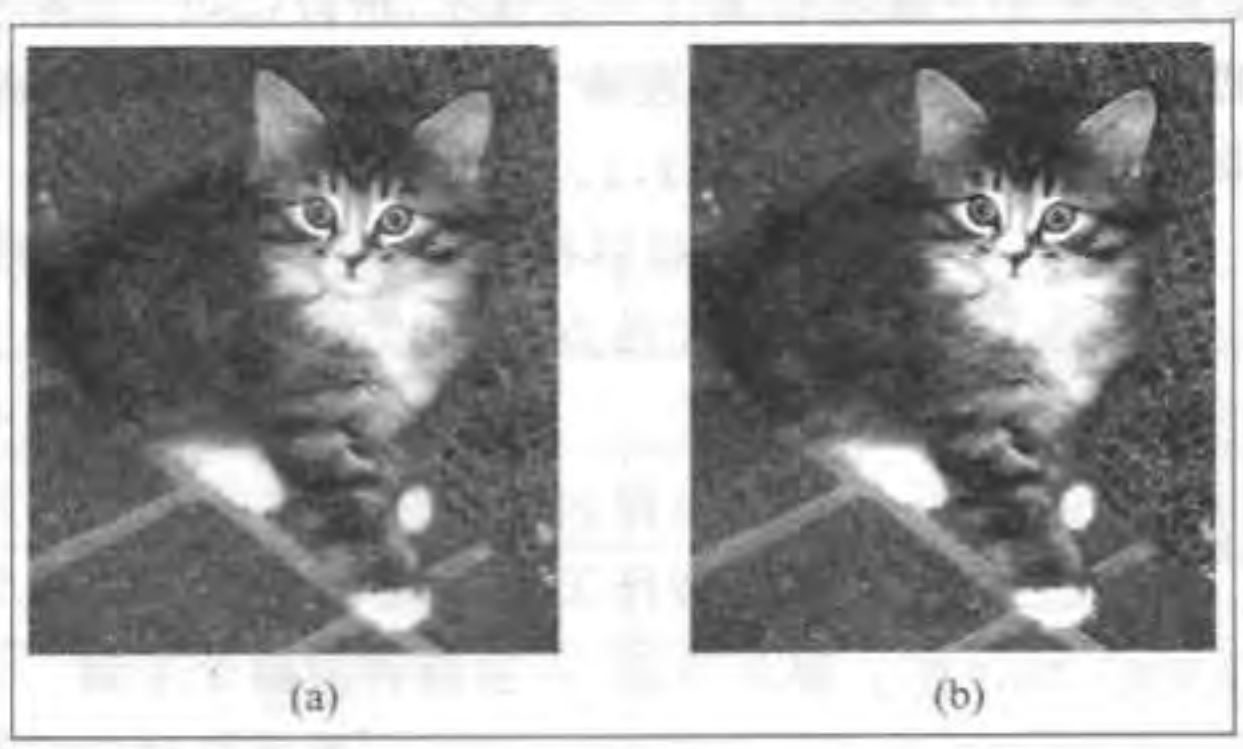


图 1.34　图像减为每像素 6 比特和每像素 3 比特

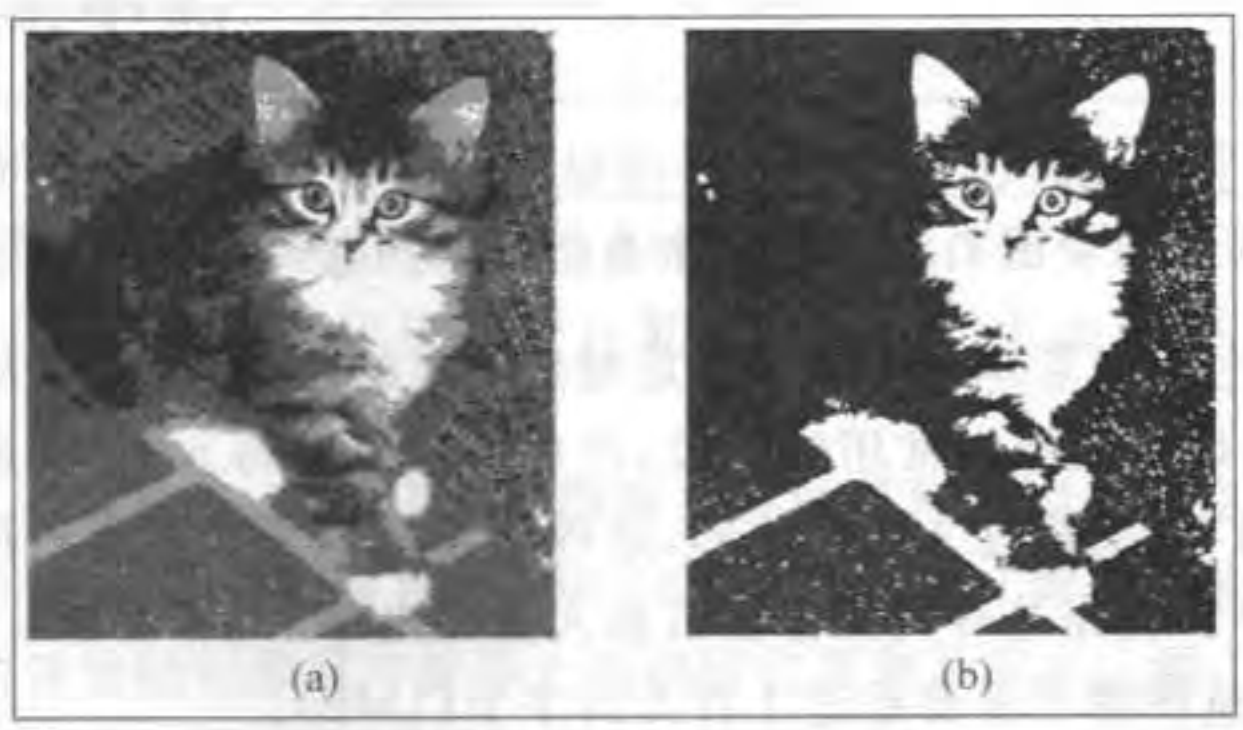


图 1.35　图像减为(a)每像素 2 比特和(b)每像素 1 比特

注意该图的一些区域,原图是灰色渐变的,现在成了一块统一的灰色。这通常被称为**条带化**,因为原图中本来是渐变的灰度被一系列固定值的灰度块所替代。

图 1.35 显示了每像素 2 和 1 位的例子。在图 1.35(a),4 级灰度清晰可见,存在大量的条带。图 1.35(b)只有黑色和白色,大量的原图信息损失。在第 9 章将讲述抖动等技术,旨在提高每像素位数太少的图像的质量。

彩色光栅图像

我们喜欢彩色图像是因为它们比灰度图更接近我们的日常经验。近几年来彩色光栅图像变得普及了,因为高质量彩色显示的成本变低了。数字化彩色照片的扫描仪成本现在也非常地适中。

彩色图像的每个像素都有一个彩色值,即一组可以代表颜色的数值。有一些方式可以将数值与彩色联系起来,而其中最常用的一种是将色彩描述为红、绿、蓝三色光的混合。在这种表示方式下,每个像素值是一个 3 **元组**,如(23、14、51),依次描述了红、绿、蓝三色光的强度。

用来表示每个像素色彩的比特数常称为它的色彩深度。红、绿、蓝被称为三原色,将在第 10 章讨论。(红,绿,蓝)三元组的每个值都有一个特定的比特数,色彩深度是这些值的和。3 元组的色彩深度允许每个元素 1 位。例如,像素值(0,1,1)表示红色是关着的,绿色和红色是开着的。大多数情况三原色叠加在一起显示,所以(0,1,1)表示绿光和蓝光叠加在一起,它代表青色。既然每种原色可以有或没有,这里就有 8 种可能的颜色,如图 1.36 中表格所示。可以想象,同等数量的红、绿和蓝(1,1,1)产生白色。

颜色值	显示颜色
0, 0, 0	黑色
0, 0, 1	蓝色
0, 1, 0	绿色
0, 1, 1	青色
1, 0, 0	红色
1, 0, 1	品红
1, 1, 0	黄色
1, 1, 1	白色

图 1.36 颜色值和色彩之间的一般对应

图 1.36 中三原色的色彩深度是不能满足精度需求的,通常需要更大的色彩深度。因为一个字节(8 比特)是计算机操作常用的单位,许多图像都有每元素 24 或 8 位的色彩深度。这样一来每个像素有 2^{24} 或超过 2 亿种色彩,这种图像被认为是真彩色图像,是人眼所能分辨的好的彩色复制品。但是这样的图像会占据大量的内存:每个像素 1 字节,一个 1280 乘以 1024 像素的高质量图像就需要超过 1 百万的字节(1MB)。

1.4 图形显示设备

我们将简要介绍一些显示计算机图形的硬件设备，这些设备包括显示器和打印机。近30年来已设计出了多种的图形显示设备，并且新设备还在不断涌现，其目的是为了更真实地再现艺术家或工程师所创造的高质量图像。本节我们将介绍当今制作图片的种类，以及如何使用它们和显示它们的各种设备。我们将了解衡量图像质量的方法，以及怎样比较各种不同的显示设备。

1.4.1 线画显示

某些设备只是绘制线条。因为早期时代的技术限制，最早的计算机图形学只能通过画线设备来显示。经典的例子是笔式绘图仪，笔式绘图仪根据计算机指定的位置到纸上移动笔，留下某种颜色的墨迹。某些绘图仪有多种笔，程序可以自动调用以绘制出不同的颜色。通常可选择的色彩非常少：每种颜色都需要使用一支单独的笔。线画的质量取决于定位笔的精度和线画的清晰度。

有各种不同的笔式绘图仪。平板绘图仪在二维的范围内在一张固定的纸上移动笔。滚筒式绘图仪通过滚筒的移动来提供纸的一个移动方向，通过滚筒上笔的左右移动来提供另一个移动方向。

有一类显示器，称为向量、随机扫描或书写显示器，能够产生线条图。它们的内部线路以点对点的方式在显示器表面扫描电子光束，从而留下轨迹。

然而向量显示器不能显示平滑的着色区域或扫描图像。区域填充通常用不同线画图案的交叉阴影线来模拟，如图 1.37 所示。除了个别的特殊应用，现在光栅显示器已取代了大部分向量显示器。

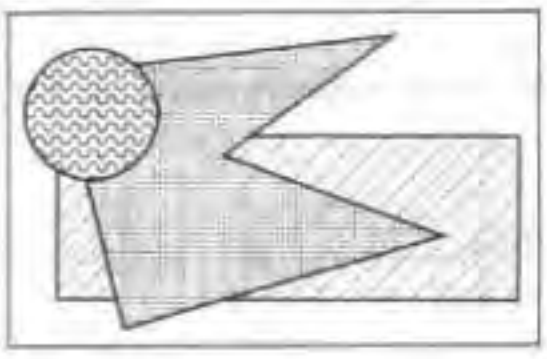


图 1.37　交叉阴影线模拟填充区域

1.4.2 光栅显示器

现在，显示计算机图像主要是采用光栅显示器。最常见的就是与个人电脑或工作站相连的视频显示器(见图 1.38)。其他的常见设备主要用来制作图像的硬拷贝(通常在纸上)：激光打印机、点阵打印机、喷墨打印机和胶片复印机。我们下面介绍其中最重要的几种设备。

光栅设备有一个显示屏幕，以用于显示图像。该屏幕可以表达为由若干行(如 480 行)和若干列(如 640 列)，则屏幕能同时显示 $480 \times 640 \approx 307200$ 像素。这样的显示器有一个内置的坐标系统将图像的给定像素与屏幕上的物理位置联系起来。图 1.39 举了一个例子。这里水平坐标 x 从左到右增长，垂直坐标 y 从上到下增长。这种颠倒的坐标系统在光栅设备中是最常见的。

图 1.38
(a) PC 的显示器；(b) X 光的平面显示器

光栅显示器总是与帧缓冲区联系在一起。帧缓冲区是存储器中一块足够容纳要显示图片的区域。帧缓冲区可以是显示器自带的物理存储，或者在主机中。个人计算机安装的显卡中有帧缓冲区所需的存储器。

图 1.40 展示了一个图像是怎样被制作和显示的。图形软件存储在系统内存，通过 CPU 的指令执行。程序计算出每个像素的数值并把它们装载到帧缓冲区中(这是后面讲述编程时我们要着重讲的部分：计算正确的像素值并把它们写入帧缓冲区)。

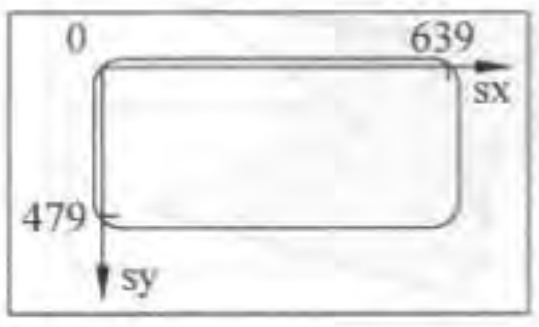


图 1.39 光栅显示器的内置坐标系

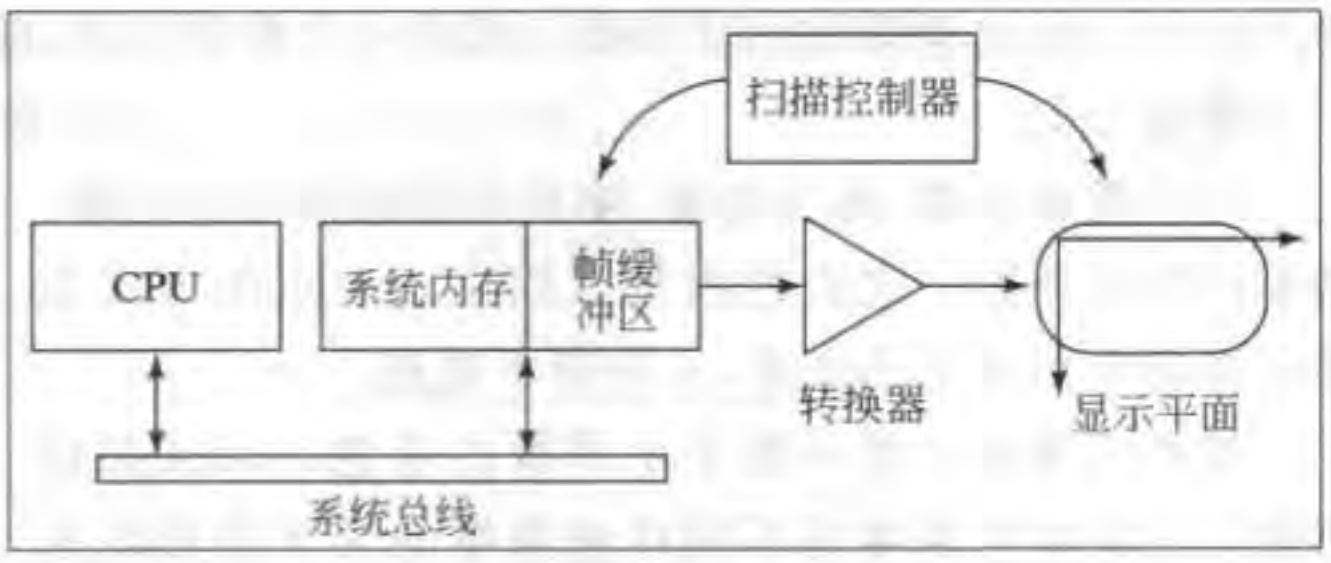


图 1.40 使用光栅显示器的计算机流程图

扫描控制器负责真正的显示处理过程。它自动运行(胜于用程序控制)，一个像素一个像素地重复同样的工作。它引发帧缓冲区通过转换器将每个像素输送到显示平面合适的物理位置。转换器接收像素值(比方说 01001011)，然后把它转换为相应的色彩值，并在显示器上生成一个彩色点。

扫描处理

图 1.41 描述了扫描处理的更多细节。其中的主要问题是帧缓冲区中的每个像素值是怎样输送到显示屏幕上正确的物理位置。帧缓冲区中的每个像素存储在某一存储单元，并对应着显示器上的某一位置。比方说，对应于屏幕上(136，252)位置的像素值，就存储在帧缓冲区存储单元 mem[136][252]中。当扫描控制器传送地址(136，252)到帧缓冲区，它发送值 mem[136][252]。转换器也同时在显示界面上找到位置(136，252)。值 mem[136][252]在转换线路中转换为相应的色彩，并被送到显示界面上的正确位置(136，252)。

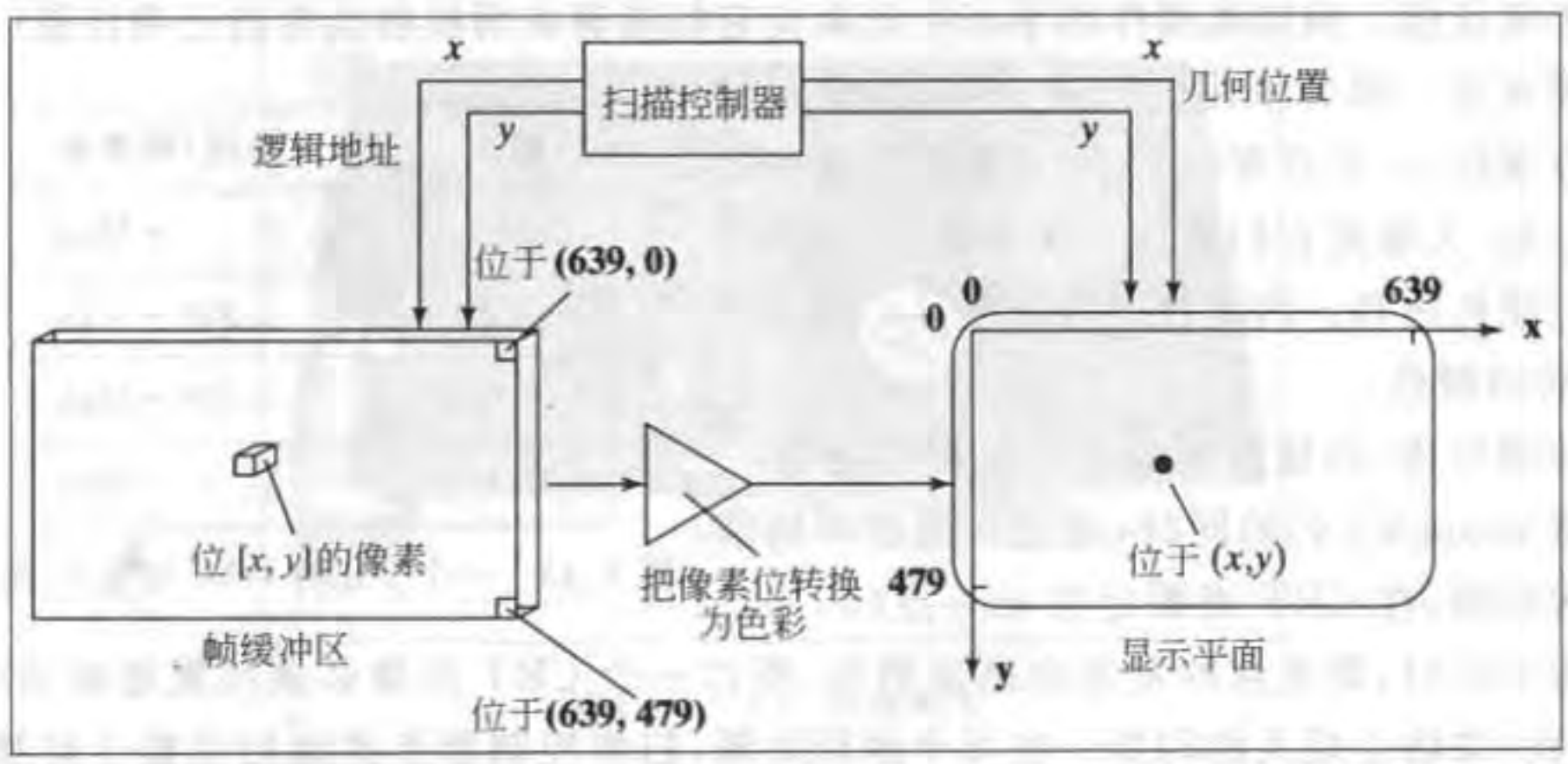


图 1.41　从帧缓冲区中扫描一个图像到显示器屏幕

在整个帧缓冲区中扫描图像，每个像素值只被扫描一次，它在屏幕上的对应点显示正确的强度或颜色。

在某些设备中，这种扫描必须每秒钟重复许多次，用来保持和刷新显示器上的图像。下面将讲到的视频显示器就要做大量的这种工作。

在介绍完这些一般性质之后，我们将简单介绍一些特殊的、不同类型的光栅设备。

视频显示器

许多视频显示器的功能基于阴极射线管(简称 CRT)，近似于电视机的显示器。图 1.42 中对采用视频显示器的系统，在前面所讲一般性描述的基础上又增加了一些细节，特别是从像素值到亮点的转换处理采用图示说明。所示的系统有 6 比特的色彩深度，所示帧缓冲区有 6 个位面，每个像素使用每个位面的 1 比特。

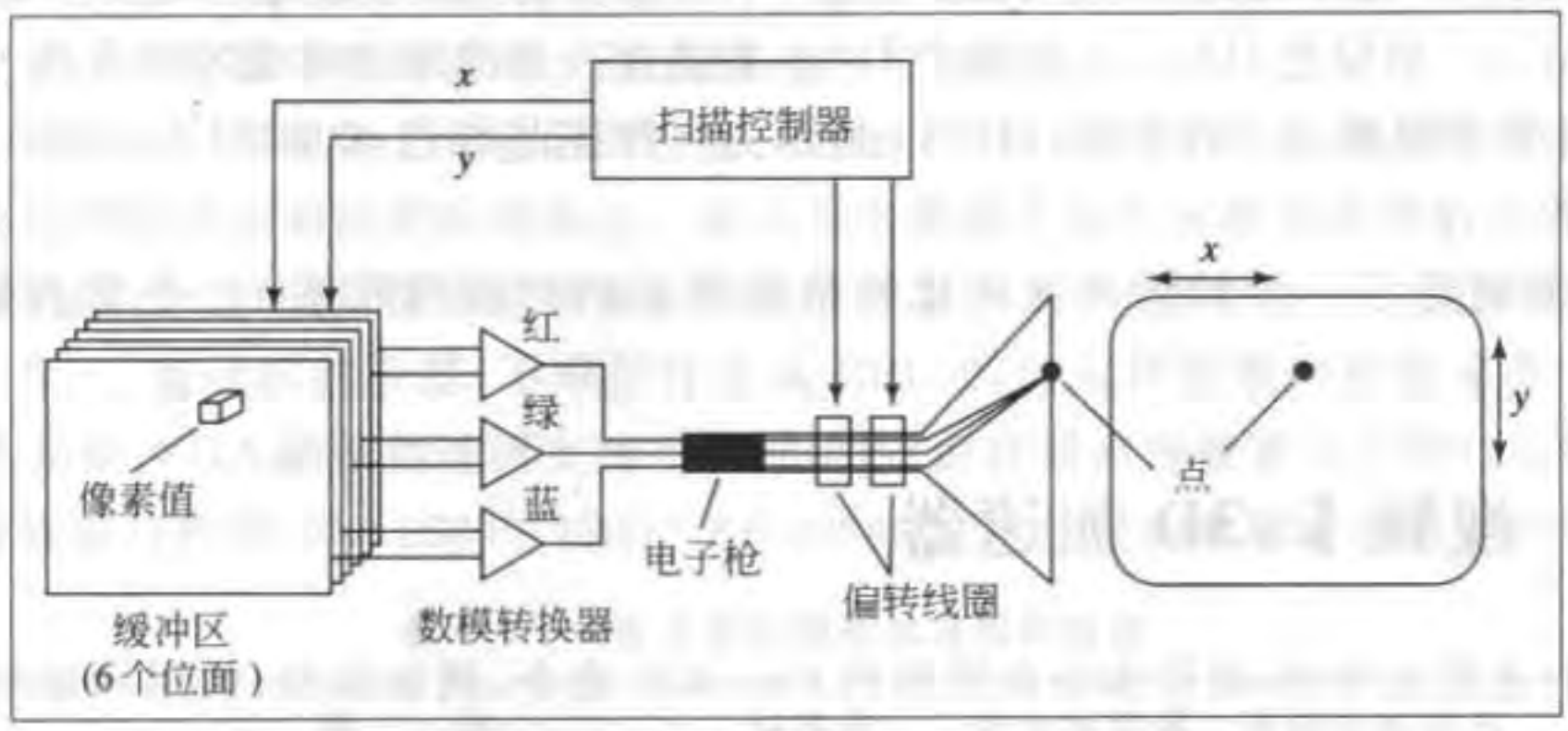


图 1.42　彩色视频显示系统的操作

每个像素的红、绿和蓝成分分别使用一对比特值。这些比特值输入到三个数模转换器(DAC)中，将逻辑值(如 01)转换成真实的电压值。数字输入值和输出电压之间的对应如图 1.43 所示，标出的最大值就是 DAC 能产生的最大电压。

三个电压值分别驱动阴极射线管(CRT)的三个电子枪，轮流发射三条电子光束，其亮

度正比于电压值。偏转线圈作用于三个光束使它们能够在阴极射线管的正确位置(x,y)刺激三个荧光点。因为采用的是荧光物质,受刺激时一个点变红,一个点变绿,一个点变蓝。这些点紧挨在一起,人眼把它们看成一个合成点,看到的颜色为三原色之和。因此合成点产生 4×4×4=64 种不同的颜色。

输入	电压/明亮度
00	0 * Max
01	0.333 * Max
10	0.666 * Max
11	1 * Max

图 1.43 一个 2 比特 DAC 的输入-输出特性

如前面所述,扫描控制器在访问帧缓存区一个像素值 mem[x][y]的同时,通过发送正确的信号给偏转线圈,在 CRT 表面定位于一点(x, y)。因为刺激不在时,荧光点的发光会迅速消失,所以一个 CRT 图像必须频繁地刷新(一般一秒钟 60 次)来防止烦人的闪烁。在每个刷新间隔,扫描控制器迅速地扫描整个帧缓冲存储器,把每个像素值传送到显示屏表面的正确位置。

帧缓冲区被一行一行的扫描,每行提供 CRT 屏幕一根**扫描线**的像素值。扫描的顺序一般是从左到右,从上到下。历史学家说这种习惯起源于扫描线,就像扫描线编号从上到下从 0 开始,并由此导致了上下颠倒的坐标系统。

大多数现代系统拥有每像素 24 或 32 比特的显示能力。这些设备曾经要比那些每种颜色只有几个字节的设备贵很多,但是现在它们的价格非常适中。24 比特的显示器每像素能提供 2^{24}=16 777 216 种不同的颜色,32 比特的显示器每像素能提供大约 2^{32}=2 亿种颜色(但是 8 比特经常用于透明效果的显示,这些我们随后将讲到)。

另一方面,**单色显示器**只有一种颜色但是有很多种亮度。单个的 DAC 把帧缓冲区中的像素值转换成电压级别,只驱动一个电子光束枪。这种阴极射线管只有一种荧光,所以它只能产生不同亮度的一种颜色。注意帧缓冲区中的 6 个位平面提供大约 2^6=64 级别的灰度。

彩色显示器与显示色彩的联系是固定的(不可编程的)。例如,像素值 001101 输送 00 给红色 DAC,11 给绿色 DAC,01 给蓝色 DAC,制造出一种亮绿和暗蓝的混合色 — 蓝绿色。(参见第 11 章颜色理论。)近似地,110011 显示为一种亮品红色,000010 显示为一种中等亮度的蓝色。

第 11 章讨论了一种帧缓冲区中比特值和色彩的可编程联系——色彩查询表(简称 LUT)。

1.4.3 视频卡/3D 加速器

在我们通览此书时,有许多绘制图形的 OpenGL 命令,例如画线、圆形和矩形。当执行这样的命令时,OpenGL 必须把大量的像素数据从 CPU 送到显示器。数据的传输,当然需要一些时间,虽然短暂,还是会起到不可忽视的影响,例如需要快速动画的时候,或者正在玩实时人机交互游戏的时候。经过近年来的迅速发展,图形影像已经变得越来越复杂。包括将每幅图像的海量数据迅速传递给显示器,或在足够短的响应时间内运行流畅的动画,都已经变得越来越有挑战性。

因为 CPU 和显示器这两种设备不在同一位置,数据必须通过连线传输。数据的传输

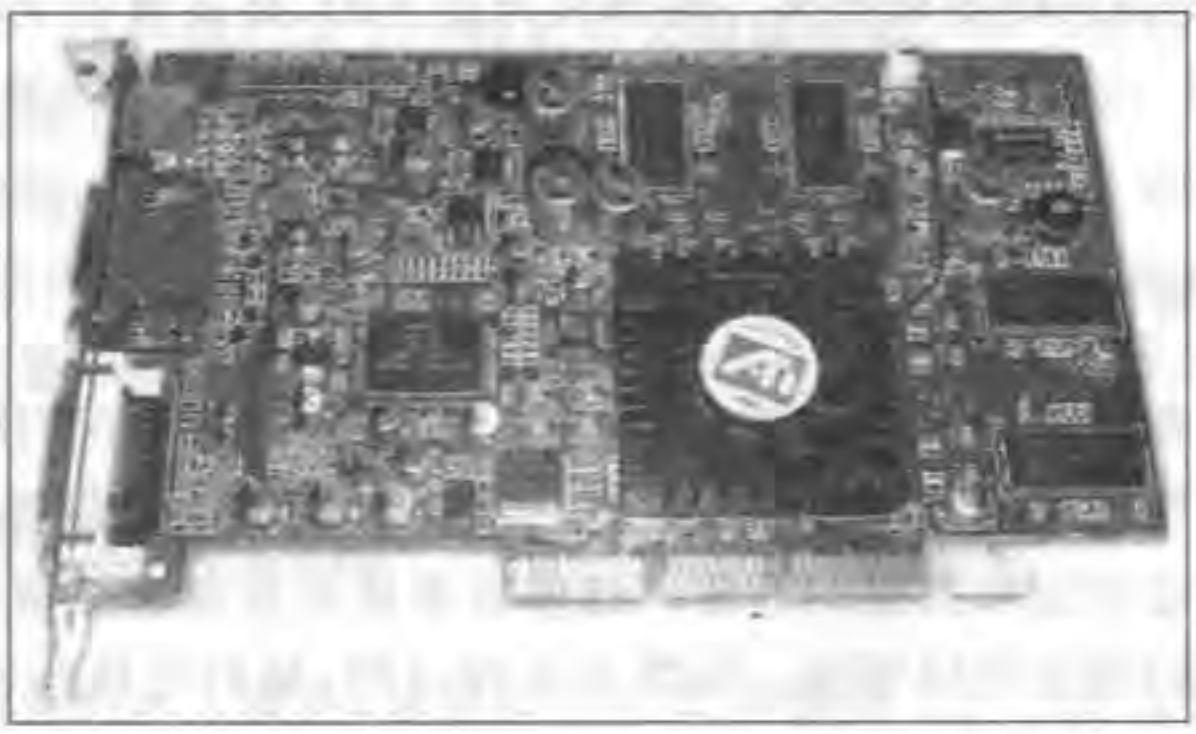


图 1.44　ATI 的视频卡

路径和控制传输的大规模数字逻辑线路，被统称为**总线**。一些类型的总线比另外一些有更快的响应，人们一直在努力确保总线是足够快的。（总线越快，价钱越高，性能越好。）

另一个重要的发展是**图形卡**或**3D 图形加速器**。这种硬件有着特别设计的线路，能够快速响应，从而满足数据从 CPU 到显示器的实时传输需要。图形卡比过去的'显示适配器'更昂贵；现在的加速器包含大量的板上集成存储器和特殊工艺的总线逻辑加速数据传输。（至少数据传输不是简单地通过一些逻辑门的开关控制；在数据通过总线传输前必须完成一系列复杂的、基于当前总线活动的时间检查和数据检测。）

无论何时图像成形，定义场景的几何数据必须经过大量的处理步骤。OpenGL 简单指定了这些步骤的属性和它们出现的次序，这些步骤通常称为**图形管线**(graphics pipeline)，我们在此书中许多地方都进行了讨论。

图形显示产业的风险是很高的，生产商之间激烈的竞争迫使他们迅速开发更快的图形加速器。不可避免的是，各种类型图形加速器的出现导致了对**标准**的需求，便于每个潜在的顾客知道他/她要买的加速器真实性能怎么样。接下来我们会介绍其中的一些标准，但是要明确的是这是一个快速变化的领域，所以标准也一直在变化。

表 1.1 列举了三代视频卡和它们的性能。创始于 1989 年，VESA（视频电子标准协会）是一个企业范围的显示和视频标准协会。表 1.1 中提供了每种标准加速器的型号和制造商名称、支持的显示分辨率、每像素能够显示的颜色数和刷新率（整个显示的图像多长时间被重新绘制一次）。通过这些数据，不难估计出从 CPU 到显示器的数据传输速率。例如，作为 VESA 会员的 VGA 图形加速器支持 256 种颜色（由此看出每像素 8 比特），每秒钟 60 次的刷新率意味着每秒钟 60×1600×1200×8＝921600000 比特的数据必须被送到显示器！

表 1.1　一些主要的图形加速器和特点

年度	型　号		制造商	最大像素值	支持的颜色数	刷新率
1981	MDA	Mono Display Adapter	IBM	720×350	2	50Hz
1990	SVGA	Super VGA	VESA	1600×1200	256	60Hz
1997	AGP	Accelerated Graphics Port	Inter	2048×1536	16.7 百万	100Hz

要了解最新的关于视频卡和总线加速的趋势，可在网上使用搜索引擎进行搜索。本书的网址也列举了一些资源。

作为对比,数字化音乐则简单得多。刻录在光盘上的、高质量的音乐只需要每秒钟 705 600 比特的传播速率!(这个每秒比特值基于每秒 44 100 个采样的采样率,每个采样代表 16 比特。)

不可编程的与可编程的硬件

早期的图形芯片和图形卡有固定功能的图形管道。1999 年 NVIDIA 发布了 GeForce 256,它是第一个可编程的视频卡。可编程性允许开发者指定他们想要这些强大的 GPU 做什么——通过编辑管道中加载数据的算法。这些进步使开发者通过精加工自己的顶点和像素绘制程序(或着色器)增强实用等级。如第 8 章探讨的,他们可执行在每个像素和在每个顶点处的操作。例如,可实现照明、毛发和透明玻璃等效果。

随着图形卡支持更多的指令和开发者要编写更加复杂的着色器,在低级汇编语言里编写着色器已是不可能的了。这导致了一系列高级着色语言的发展,如 Cg 和 OpenGL 2.0 的着色语言。Cg 是一种类似 C/C++ 的语言,虽然不像汇编语言那么有效,但允许快速开发,可以写出可读性好和可维护的代码,比汇编语言更具平台无关性。

想更全面的了解 GPU,可查阅 GPU Gems 系列书和浏览其出版公司的网址。

1.4.4 其他的光栅显示设备

视频显示器不是唯一的光栅显示设备:近来还出现了多种其他类型的设备。便携式计算机一般带有平面显示器,如图 1.45 所示。每个像素被一个水平格线和一个垂直格线定位:要打开一个像素,则激活对应的水平和垂直格线,在这个像素位置制造一个电场。这可以有效激活该点并改变它的亮度。

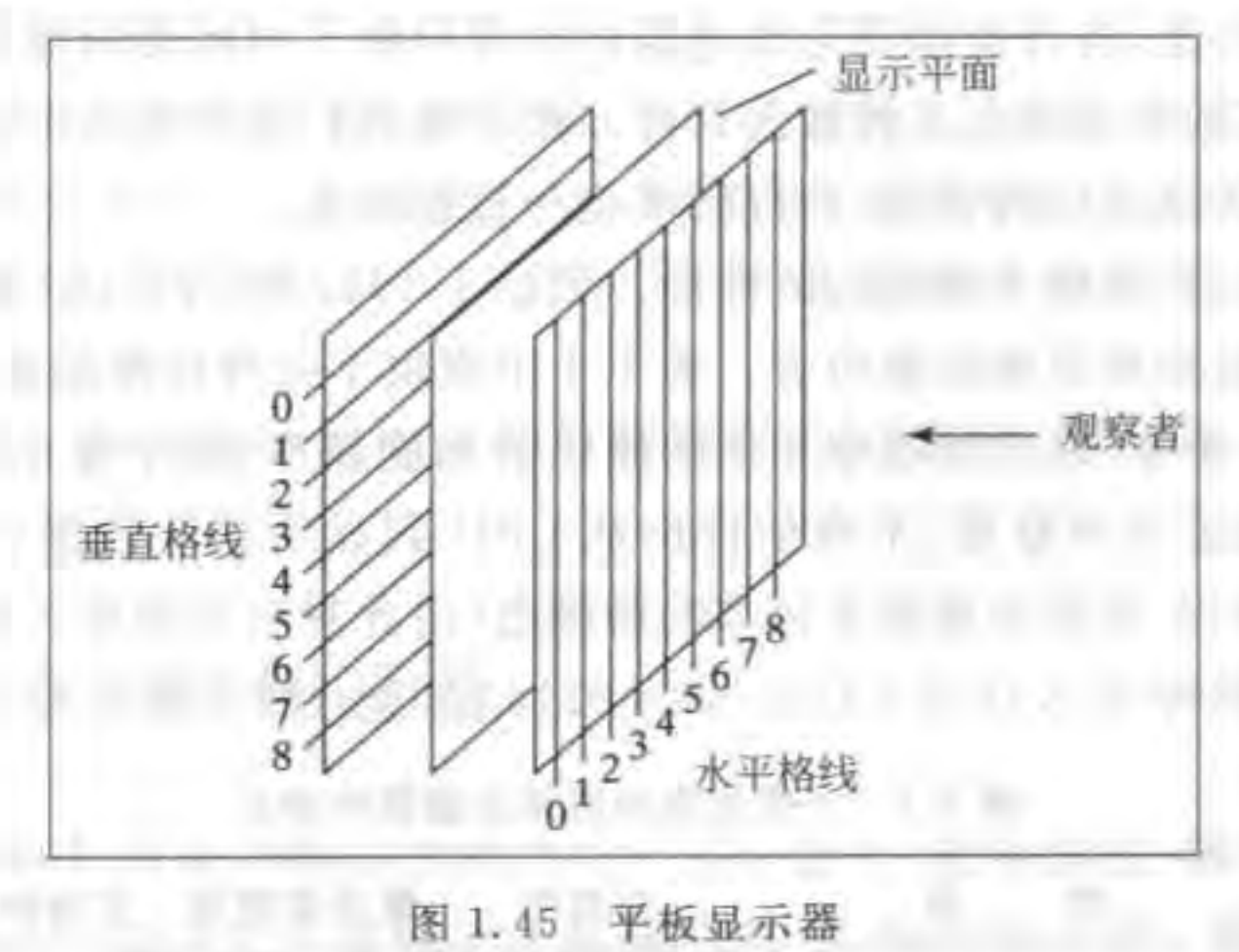


图 1.45 平板显示器

把电场转换为一个可视点的精确机制依赖于所采用的技术。**液晶显示器**(LCD)中,电场改变了 LCD 材料中长水晶分子的极性。这使它允许光穿过平面,或阻止光穿过。**有源矩阵板**是每个像素位置都对应一个微型晶体管的液晶显示器。晶体管响应电场,根据电场的比例调整液晶,决定显示亮度的不同级别。另外,晶体管能够存储液晶的调整状态,于是显

示器就不再需要刷新。这样就产生了一种更明亮的显示器。彩色液晶显示器的分辨率可以达到 800×1000 像素或更高。

等离子显示器结构上近似于图 1.45。不同的是，板内在每个像素位置放置了一个非常小的氖管，这个管子通过电场控制开关。像有源矩阵显示器一样它也不需要刷新。

其他类型的光栅显示器可参阅参考文献，如【Foley90，APeer85】。

1.4.5　硬拷贝光栅设备

常常有人需要图像的永久版本，通常保存在纸或胶卷上。一些光栅设备提供了光栅图像的硬拷贝，通过把帧缓存的信息以像素的形式传输到显示媒质上来保存图像。

- 激光打印机。激光打印机从其内部的帧缓存中扫描出光栅图案，然后激光迅速地扫过内部绘图表面。在激光扫过的某一点时，表面变得带电，吸引墨粉附着在点上，转到纸上的墨粉绘制成了图片。利用激光定位的高精确度，激光打印机能提供比喷墨打印机和点阵打印机高得多的分辨率。

图 1.46(a)展示了点阵打印机(最早大量生产的高速打印机类型)绘制的文本和图形，图 1.46(b)展示了激光打印机的输出。点阵打印机的点密度大约只有每英寸 70 点数(dpi)。现在点阵打印机通常用在收款台打印收据，例如餐馆和商店。另一方面，激光打印机能提供大约每英寸 600 多点数，所以能打印出高质量的图形。

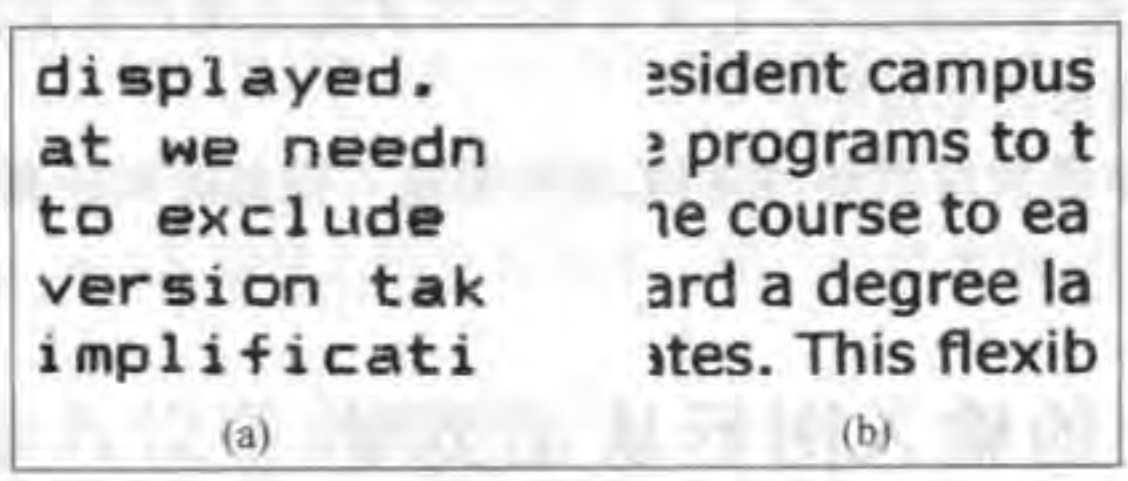


图 1.46　放大的点阵打印机和激光打印机图像

更高密度的图像可通过出版行业的印刷机获得，例如莱诺特朗照相排字机，每英寸 2540 点数。本书就是在这样的设备上印刷的。

当今，许多印刷机配置有内部的微处理器，通过编程可以解释 Postscript。Postscript 是一种页面描述语言，它能在打印页上生成高质量的文本和图像。当包含这种描述语言的文件印刷时，Postscript 编译器就生成了如图 1.47(b)所示的图像。

- 喷墨绘图机。喷墨绘图机用来制作彩色的硬拷贝光栅图像。很小的喷嘴扫过纸张，然后在每个像素的位置喷上正确的墨色。
- 胶片记录仪。在胶片记录仪中，屏幕是一卷照相胶片，当电子光束以光栅模式扫描时电子束使胶卷曝光。有时，胶片记录仪是独立的帧缓冲设备，或只是直接安装在液晶显示器上的简单相机。胶片记录仪常用来制作高质量的 35 毫米幻灯片或电影。磁带录像机用来制作存储在帧缓冲中的图像硬拷贝，然后在电视机上播放。

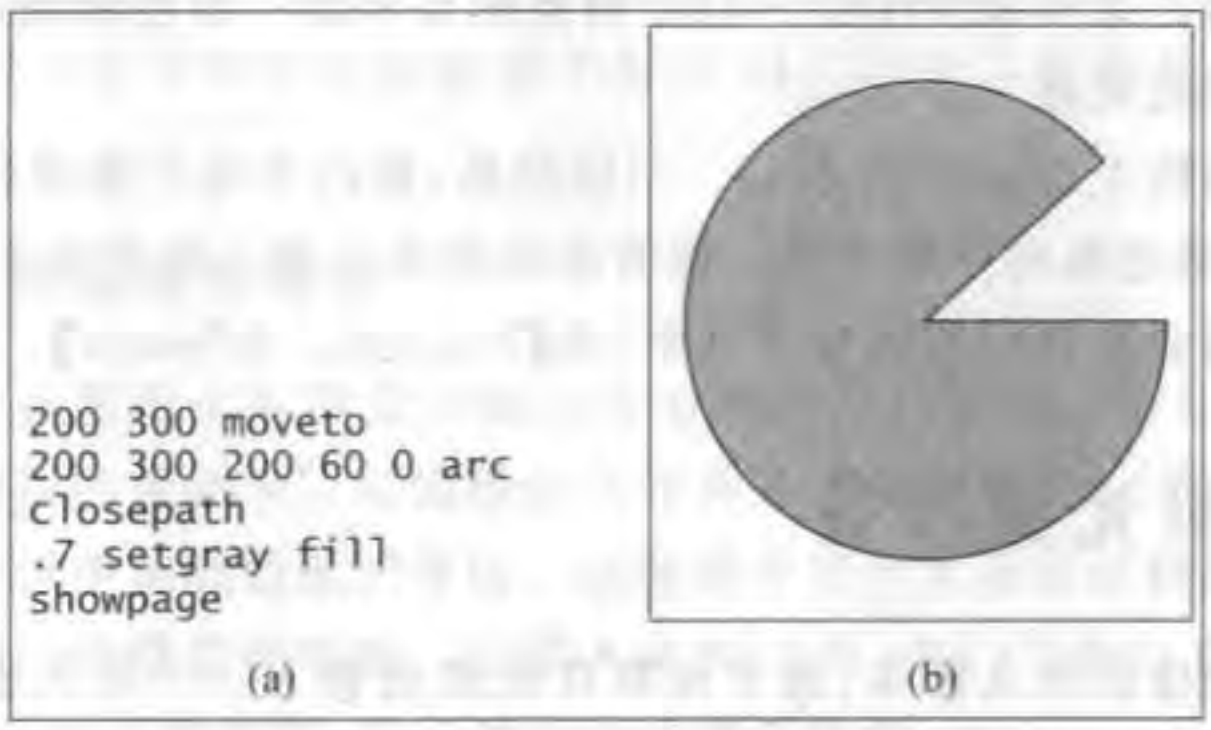


图 1.47 页面描述语言和用它生成的图像

1.5 图形输入的基本单元和设备

许多输入设备可以让用户控制计算机,但是键入命令的方式可能是笨拙的。指向屏幕上显示的特定目标以选择下一步做什么,可能更加轻松自然。

你能以两种方式考虑输入设备:物理上它是什么,逻辑上它做什么。每种设备是物理上的某件机器像鼠标、键盘或跟踪球,它采用某种方式配合用户手工操作,这些操作被评估并把相应的数字信息送回图形程序。

通过检查送到程序的数据类型,我们首先看看输入设备做些什么,然后我们介绍现在常用的数据类型。

1.5.1 逻辑上的输入图形基元类型

每个设备传送特定类型的数据(比如,数字、字符串或坐标位置)给程序。不同种类的数据被称为输入的基本单元。

重要的逻辑**输入基本单元**包括:

- **字符串**。字符串产生设备是大家最熟悉的,它生成**字符串**并为键盘的输入建模。当一个程序需要一个字符串时,程序暂停以等待用户的键入,输入的字符串后跟一终止字符来激活程序继续运行。
- **赋值**。**赋值设备**产生一个介于 0.0~1.0 之间的实数值,可用来确定划线的长度、运动的速度或图像的尺寸。它能够产生 0 到 1 之间的任意实数。
- **定位**。交互式图形的基本需求是能够让用户指向显示器的某个位置。**定位输入设备**实现了这个功能。它可以生成**坐标**(x, y)。用户操作输入设备(通常是鼠标)是为了把可见的光标指向某点,然后触发选择。它返回给应用程序 x、y 坐标和触发值。

当程序开始运行时,图形工作站被初始化。每个逻辑输入程序都和一个已安装的物理设备相联接。

- **选取**。选取输入设备是用来识别图像的某一部分以便于进一步处理。一些图形包允许图像定义成**段**的形式。段是相关图元的集合。例如，在饭店里顾客经常看到服务员按屏幕的不同部分来确认送给厨房的菜单。图 1.48(a)展示了饭店销售系统(POS)的例子。销售系统的功能是搜索屏幕来决定与服务员的指压区域相关联的食品项(如主菜、开胃菜、沙拉、餐后甜品、饮料等)。注意，图 1.48(a)所示的销售系统除了屏幕以外没有其他的输入设备。销售系统的触摸屏(图 1.48(b)所示)定义了段，用来识别用户确认的名字，以加速点菜过程。

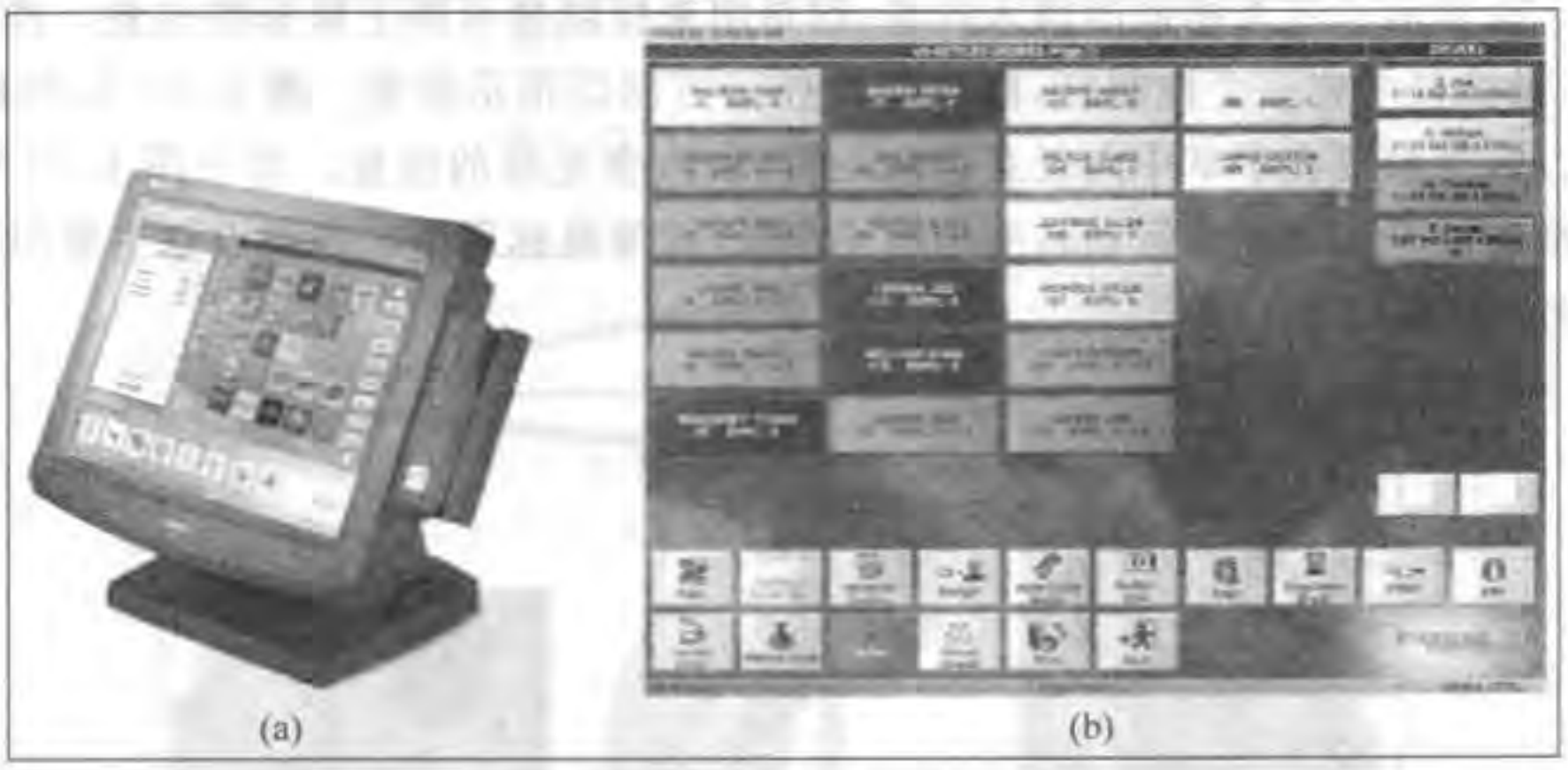


图　1.48

(a) 一个 POS 系统；(b) POS 系统触摸屏的放大图

当使用 pick()函数，用户通过某一物理输入设备指向图片的某一部分，然后图形包确定指向的是哪一个段。pick()返回段名给应用程序。用户能够擦除、移动或操作这些段。

1.5.2　物理输入设备的类型

我们从另一角度——与个人计算机或工作站相连的机器——来介绍一下输入设备。

- **键盘**。所有的工作站都配备有键盘。它根据要求输入字符串。因此键盘是最常见的生成串的逻辑设备。一些键盘有鼠标键或功能键，常用来产生选取操作。
- **按钮**。有时工作站安装有独立的一排按钮。用户按其中的一个按钮来实现一个选取输入功能。
- **鼠标**。鼠标也许是所有输入设备中大家最熟悉的，因为它便于操作。当用户在桌面上滑动鼠标时，鼠标将它的位置变化传送给工作站。工作站内的软件跟踪鼠标的位置，并在屏幕上移动一个图像光标——小箭头或十字。鼠标常用来执行定位功能。鼠标上通常有一些按钮，通过用户点击来触发行动。
- **写字板**。像鼠标一样，写字板用来生成定位。一个写字板，如图 1.49 所示，提供了一个用户可以用笔在上面滑动的区域。笔尖有一个微型开关。

图 1.49　图形写字板

通过按压笔,用户能够触发逻辑功能。

写字板特别适合数字化制图:用户可以在写字板表面铺一幅图,然后在图上移动笔,按压并输送每个新点给工作站。有时菜单区域也会印在写字板表面,用户可以选择菜单项。每个菜单项对应着一个合适的软件,点击即可运行。一个与写字板相似而又小巧的设备是 **PDA**(个人数字化助手),它也是通过使用笔选择显示屏上的特定位置来完成操作。

操纵杆和跟踪球

图 1.50 显示了三个相似的输入设备,均是用来控制显示屏上鼠标的位置。图 1.50(a)中的拱形操纵杆中有一个控制杆,能够转向任一方向以指示位置。图 1.50(b)的跟踪球中有一个大球,能够握在手掌中向任意方向旋转,以改变光标的位置。对于图 1.50 中的每个设备,内部的线路都把物理运动转变为电子信号,就像鼠标那样。这些设备主要用来作为赋值设备和定位输入设备。

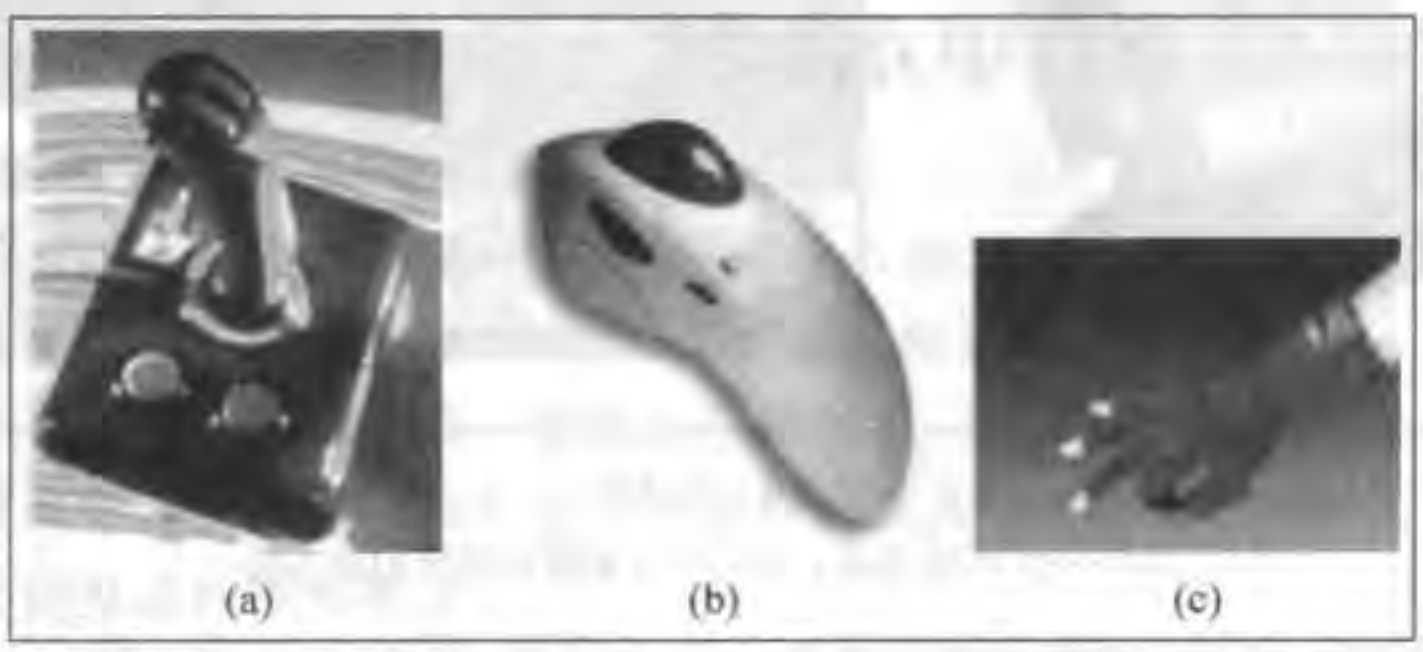


图 1.50

(a) 操纵杆;(b) 跟踪球;(c) 数据手套

数据手套

图 1.50(c)中所示的数据手套是一种相对比较新的输入设备。它被设计为通过手和手指的运动,提供给用户清楚的即时控制功能。数据手套内部的传感器获取手的细微运动并把它们转化成数值返回给程序。这种设备特别适合于与程序相关联的运动环境(比如用户控制一个虚拟的机器人手)。

三维物体的数字化和运动捕获

图 1.51(a)展示了一个能够测量空间中点的位置,并进而能够获取三维形状的设备。当激光束以 x,y 光栅模式扫描过实体目标表面时,图像捕获设备与物体之间的距离被记录下来。图 1.51(b)展示了数字化人形的合成图像。

图 1.52 展示了一个相似的设备,它能够实时跟踪一个运动物体上许多点的位置。它能够获取一个舞蹈家的所有运动细节,以用于动画或数据分析。过去也曾使用过一些其他设备,如光笔、拇指轮、桨等(相关的完整描述可在【Foley93】和【Rogers98】等书中找到)。

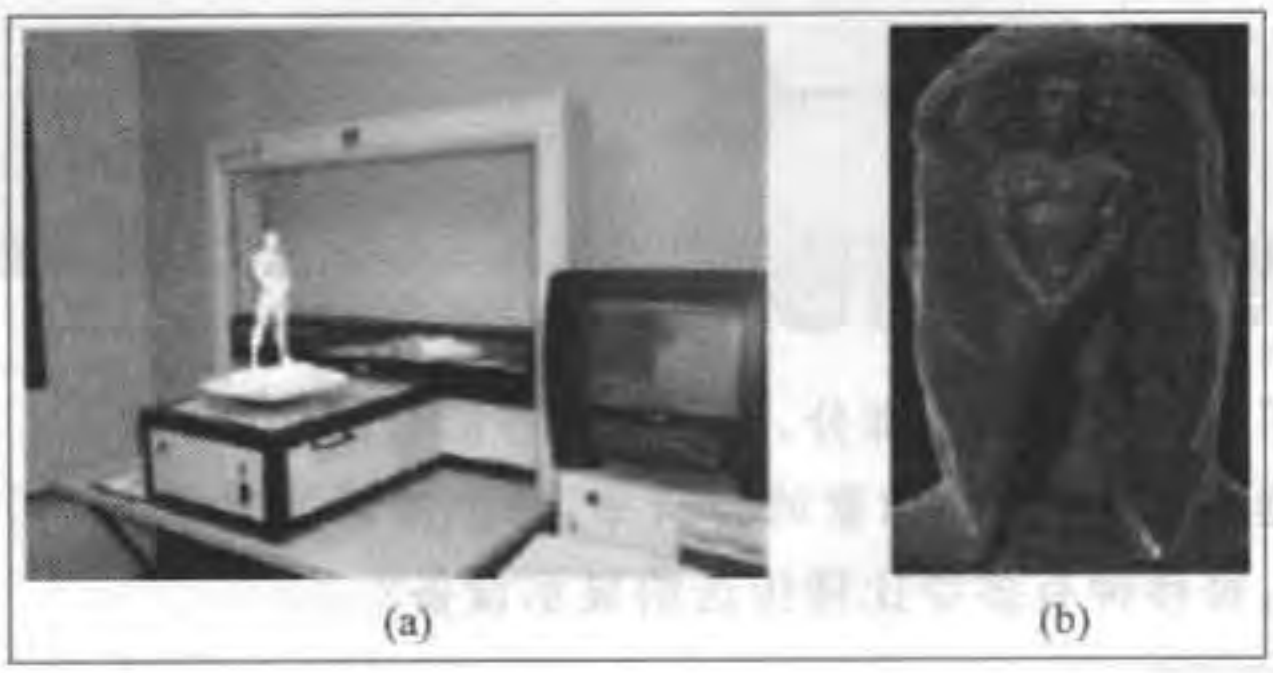


图 1.51　数字化三维模型

图 1.52　获取舞蹈家的动作

本章小结

本章我们介绍了计算机图形学的研究领域，展示了它的各个应用领域。我们还描述了多种绘图设备，其中最广泛使用的是光栅视频显示器、喷墨打印机以及激光打印机。我们也定义了主要的输出图元——折线、文本、填充区域和光栅图像——并描述了各自相关的属性。因为在图形学领域中着色图像在显示设备图片媒介中的重要作用，我们着重强调了光栅图像，对这种图像的探讨将贯穿本书。光栅图像的关键特性是它由数字集合组成，每个数字只能从有限个元素的集合中选取。计算机中二维像素值的存储位置是离散的，这种离散的表示在本书中将多次提到。

我们还描述了各种用于交互式计算机图形学的图形输入设备，并探讨了它们产生的各种输入图元。

本章习题

1.1 说出 5 种逻辑输入设备的名称(如字符发生器等),并说出能用来提供相应逻辑输入的一种物理输入设备(如键盘)。

1.2 说出计算机图形系统的主要部分,简单描述各部分的功能。

1.3 当一种显示器有 600×800 像素的分辨率,每个像素能显示 65 000 种颜色,刷新率是每秒 60 次时,每秒钟有多少比特传送给显示设备?

进一步阅读

一些书提供了很好的关于计算机图形学的介绍。Hearn 和 Baker【Hearn04】以一种轻松而有趣的方式总结了这个领域,并附带了大量的例子。Foley 和 Van Dam【Foley93】和 David Rogers【Rogers98】提供了许多种图形输入和输出设备的附加技术细节。一套被称为"图形学珍宝"的非常好的系列书【Gems】,1990 年第一次出版,汇集了全世界图形学研究者和实践者的众多新思想和"珍宝"。

还有一些关于图形学新技术的杂志。其中最有影响的是"IEEE Computer Graphics and Applications",它常常介绍一些受图形学影响的新领域进展情况的文章。反映图形学新技术的顶级会议和期刊是计算机图形学专业兴趣小组年会(Proceedings of SIGGRAPH【SIGGRAPH】)和美国计算机学会图形学会刊【TOGS】。

第2章 OpenGL绘图入门

本章学习目标

- 开始编写图形程序
- 学习 OpenGL 程序的基本组成
- 开发绘制直线、折线和多边形的基本图形工具
- 学会用鼠标和键盘控制程序

预览

2.1 节讨论了生成简单图形程序的基础知识，包括设备无关编程的重要性以及基于窗口和事件驱动编程的一些基本知识。2.2 节介绍了 OpenGL 作为设备无关的应用程序接口(API)的用法(全书都将基于这个用法)，说明了如何画不同的基本图形。通过北斗星图、Sierpinski 垫片图以及数学函数曲线图等绘图实例说明 OpenGL 的用途。2.3 节讨论了如何基于折线和多边形绘图，并开始建立个人的图形工具库。2.4 节讨论了交互式图形编程，通过交互式编程，用户可以使用鼠标和键盘控制程序的运行。本章结尾给出了许多案例，通过这些案例可以加深对本章主要概念的理解。

2.1 生成图像初步

像许多学科一样，通过练习(编写和测试生成各种不同图形的程序)可以很快地掌握计算机图形学。先从简单的任务开始，一旦掌握，就试着变变花样，看看有哪些变化，试着进一步绘制更复杂的场景。

注意：正如前言里的承诺，对每个书中讨论过的程序，本书提供了一个完整的程序清单。方便之处在于，这些源代码只需复制和粘贴就能是有效的，程序将编译和连接合适的 OpenGL 库。不便之处在于，读者需要自己一行行去扫描和理解代码(这等同于数学书里的数学公式)。最好的学习办法是认真、仔细地学习每行代码或等式。当你这么做时，就能理解每个新概念。

为了便于开始学习图形学，需要一个可以编写和执行程序的软硬件环境。对于绘图来说，这个环境还必须包括显示图形的硬件(通常是 CRT 或 LCD 显示器，我们把它称为“屏幕”)和一个软件工具库。

每个图形程序都以一些初始化工作为开始，由此建立程序所需的显示模型和坐标系。图 2.1 给出了一些可能遇到的类型。在图 2.1(a)中，整个屏幕都用于绘图：初始化时，将显

示器设置为"图形模式",并且建立了坐标系。坐标 x 和 y 以像素为单位,x 向右递增,y 向下递增。

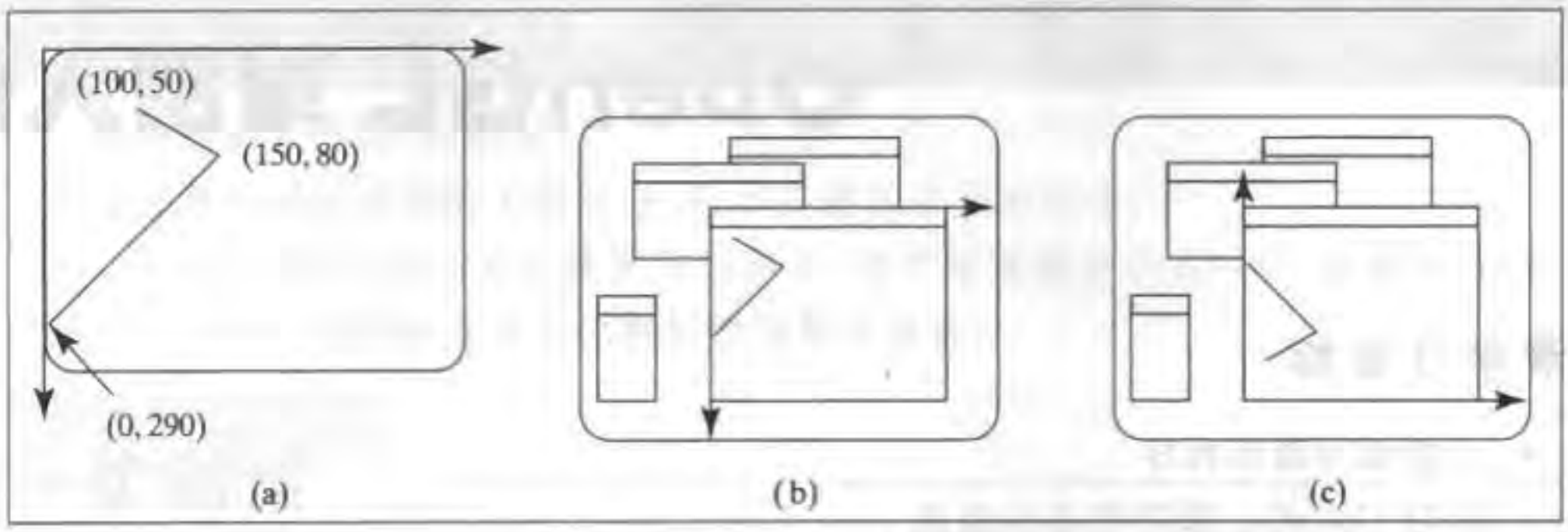


图 2.1 一些常见的显示布局

图 2.1(b)显示了现在常用的、基于窗口的系统。此类系统中,显示屏幕可以同时支持多个不同的矩形窗口。对于绘图程序来说,初始化工作包括创建和打开一个新窗口(我们把它称为"屏幕窗口"[①])。根据需要,以后还可以用命令创建多个这样的窗口。绘图命令使用一个附属于窗口的坐标系:通常,x 向右递增,y 向下递增。图 2.1(c)显示了另一种类型:初始坐标系是 x 向右递增,而 y 向上递增。

一般来说,每个系统都有一些初级绘图工具,以帮助用户快速入门。最基本的工具有诸如 setPixel (x, y, color)之类的名字,它把位于(x, y)上的单个像素设置为用 color 定义的颜色。这种工具有时有不同的名字,如 putPixel()、SetPixel()或 drawPoint()。像 setPixel()一样,也有一个绘制直线的工具,例如 line (x1, y1, x2, y2),它在(x_1, y_1)和(x_2, y_2)之间画一条直线。在其他系统中,这个工具可能称为 drawLine()或 Line()。下面的命令

```
line(100, 50,150,80);
line(150,80, 0,290);
```

可以在图 2.1 的任一系统中画出图形。实际上,每个系统都有 line()或类似的命令。

一些程序员还可以很方便地找到有稍许变化的画线函数,这里介绍两种相关的函数。但是要记住,这些变化并不重要,了解这些变化只是为了加深对基本绘图操作的理解。它们基于当前位置(cp)这个概念,即由当前位置给出画笔的位置,这个概念源自笔式绘图仪。第一个函数 moveTo (x,y)移动画笔到位置(x, y)(译者注:移动,不画线),然后将当前位置设为(x, y);第二个函数 lineTo (x, y) 从当前位置到(x, y)之间画出一条直线,并将当前位置(cp)更新为(x, y)。每个命令都将画笔从其当前位置移到一个新位置,一个可见,一个不可见。然后该新位置就变成当前位置(cp)。图 2.1 所示的图形可以用下面的命令绘制出来。

```
moveTo(100,50);
lineTo (150,80);
lineTo(0,290);
```

① 在图形学中,"窗口"一词用得有点过多,请注意区分。

再举一个例子，如图 2.2 所示，画一个中心在(1，1)，边长为 6 个单位的正方形。该正方形是对齐的，即它的每条边都平行于某一坐标轴(本书经常使用“对齐的”这个术语)。画这个正方形是容易的：用 moveTo 操作简单地将画笔移到一个角上，然后用 lineTo 使画笔画出正方形，最后，让画笔回到起始点。

画这个正方形的伪代码可以是这样的：

```
moveTo(4,4);          //画笔移到起始角
lineTo (-2,4);
lineTo(-2,-2);
lineTo(4,-2);
lineTo(4,4);          //闭合正方形
```

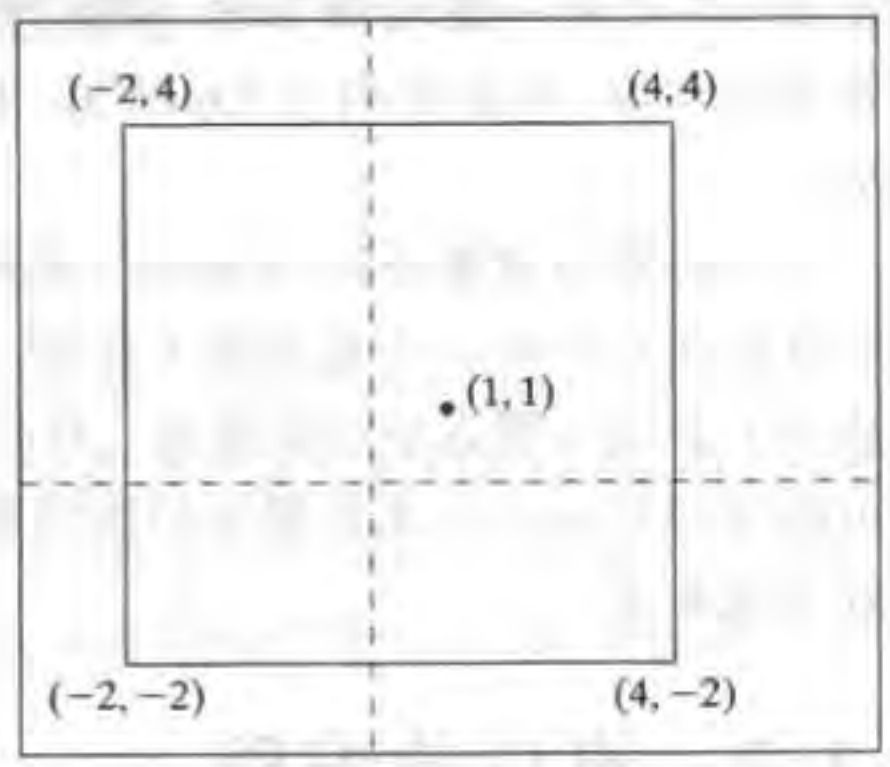


图 2.2　一个用 moveTo 和 lineTo 命令画的正方形

对于一个具体的系统，精力充沛的程序员就可以用这些初级工具，开发出一整套具有复杂功能的工具包，从而构建一个强大的图形实例代码库。利用这个个人构建的代码库就可以写出任何的图形应用程序。

但是，一个明显的问题是，每种图形显示都使用不同的基本命令来驱动，并且每种环境都有不同的工具集来产生图形的基本单位(点、线等)。这使得把程序从一种环境中移植到另一种环境中变得很困难，并且这是每个程序员迟早都要面临的问题。所以，程序员必须基于新环境库建立必要的工具，这可能要对库或应用程序的整体结构做很大的改动，这对程序员来说工作量是很大的。

2.1.1　设备无关的编程和 OpenGL

如果有一个统一的方法可以编写图形应用程序的话，这就太方便了。也就是说，同样的程序可以在不同的图形环境里编译和运行，并且能保证在每台机器上显示相同的图形。这就是众所周知的设备无关图形编程。OpenGL 提供了这样的工具：移植图形程序只需在新机器上安装合适的 OpenGL 库就可以了，不需要更改应用程序本身，可以用相同的参数在库中调用相同的函数，最终会产生相同的结果。OpenGL 绘制图形的方法已经被许多公司所采用，而且在所有重要的图形环境中 OpenGL 库已经存在[①]。你可能已经认识到 OpenGL 是一个开源图形库，也就是说，组成 OpenGL 库的所有程序都可以通过因特网(http://www.opengl.org)免费下载。这与那些需付费才能得到或使用起来有某些限制的图形库，形成了鲜明的对比。

OpenGL 常常被称为“应用程序接口（API)”。这个接口是程序员可以调用的例程，加上如何将这些例程组合在一起工作的模式。程序员仅仅“看见”接口，而不必处理位于常驻图形系统中特定的硬件或者软件。

我们将看到，绘制复杂三维场景时，OpenGL 将表现出其最强大的功能，也可以认为，用

① 附录 1 讨论了如何在不同的环境中得到和开始 OpenGL。

OpenGL 绘制简单二维物体有点大材小用。但是,用 OpenGL 绘制二维图形也有好处,因为它提供了一个统一的绘图方法。我们利用 OpenGL 提供的许多默认状态,先从比较简单的构造开始学习,然后再引入 OpenGL 中更强大的功能去编写程序,产生精心制作的三维图形。

虽然将要用封装好的 OpenGL 去开发我们的大部分图形工具,我们还是需要考查经典图形算法在 OpenGL 中是如何工作的。即使你用现成的 OpenGL 版本编写大多数程序,了解这些工具的实现方法也很重要。在特殊环境中,你可能想换一种算法来完成某项任务,或者可能遇到 OpenGL 无法解决的新问题(通观全书,应该指出,很少有用 OpenGL 不能达到的效果或要求)。

2.1.2 窗口的编程

如前所述,很多现代图形系统是基于窗口的,并且管理多个重叠窗口的显示。用户可以用鼠标在屏幕上移动窗口,也可以调整窗口的大小。使用 OpenGL,我们将在类似于图 2.1(c)的窗口上作图。

事件驱动编程

大多数基于窗口的程序,一个显著的特征是事件驱动。这意味着程序要响应不同的事件,如单击鼠标、按下键盘上的按键,或者重新调整窗口的大小。系统自动管理事件队列,该队列接收已经发生的特定事件信息,并按照先来先服务的顺序处理这些信息。程序员将程序组织成**回调函数**的集合,这些回调函数一有事件发生就执行。当回调函数执行完后,应用程序从队列移走相应的消息,再从调用处恢复成等待状态。在接下来的章节里,我们将看到这方面的例子。程序员必须在每个回调函数内部编写事件产生时将发生什么事情的代码。当然,不需要写什么时候调用回调函数的指令,调用回调函数是系统自动处理的,这一点对程序的编写者和代码的任何读者来说很重要。

因为存在的事件类型不多,所以系统对于每一种可能发生的事件类型都创建了一个回调函数。全书大部分回调函数已经命名和定义好。但是,在你自己的程序中,你可以用你喜欢的方式命名和定义它们。当系统将某个事件从队列中移走时,系统只执行与该事件类型相关的回调函数。对于那些习惯于用"先做这个,再做这个"结构编程的程序员来说,需要改变思路。这种新的结构更像是:"什么也不做,等待事件发生,事件发生后再做指定的事"。这种结构使人想到事件循环。在那种情况下,系统在重复的循环中耐心地等待,直到收到一个事件触发信号。

回调函数和特定事件类型的关联方式常常与系统有很大关系。但是 OpenGL 提供了下面描述的许多支持库,这些库为事件管理和多种其他功能提供了帮助。其中一个是 GLUT 库,即"GL 实用工具包(the GL Utility Toolkit)",它用来打开窗口,管理菜单和处理事件等。

注册回调函数

对程序员来说,一定有一个方法将每种类型的事件与要求的回调函数关联起来,这个方

法称之为注册回调函数。程序中用到的每一个事件类型都必须用回调函数注册，该回调函数的名字和定义由程序员选择。下面是使用 GLUT 库，名为 myMouse 的回调函数例子，它很方便地注册了与鼠标关联的事件：

```
glutMouseFunc(myMouse);
/* myMouse 是程序员按自己的喜好命名的,glutMouseFunc 是 GLUT 固有的名字 */
```

这行代码提示如何处理按下或点击鼠标时产生的事件。glutMouseFunc（注意大小写）是 GLUT 库的固有函数，但是，回调函数名字 myMouse 由程序员按自己的兴趣（事件类型）来命名，函数由程序员编写，被系统调用。程序员为 myMouse()编写代码，处理每个可能感兴趣的鼠标动作。如果应用程序中没有使用鼠标交互，glutMouseFunc()就不需要写或被注册。

为了帮助程序员简化编程工作，实际上可以利用 3 个库来开发和运行程序。这里只作简要的介绍，详细的介绍留给后面的章节。下面是 OpenGL 应用程序的常用库列表。

4 种主要的 OpenGL 库

(1) 基本 GL 库：OpenGL 库的基础。它提供 OpenGL 的基本函数。我们将看到，每个 OpenGL 函数都以字符 GL 开头。

(2) GLUT 库：GL 实用工具包（the GL Utility Toolkit），上面已经讨论过，它主要用来打开窗口、开发和管理菜单，以及管理事件等。GLUT 的菜单功能将在 2.5 节介绍。

(3) GLU 库：GL 实用库（the GL Utility Library），它提供高级例程，处理矩阵操作和绘制二次曲面如球和圆柱体（参见第 6 章）。GLU 库也提供将非凸和非简单多边形分解成简单形状（如三角形）的实用函数（基本的 OpenGL 处理不好这些操作）。它还在别的方面为简化程序员的工作提供帮助。所有这些在后面将作详细的介绍，或在因特网和附录的阅读材料【Woo04】中找到。

(4) GLUI 库：用户接口库（the User Interface Library），只要使用 GLUT，GLUI 就将适当地运行。GLUI 为 OpenGL 程序提供了良好的控制工具和菜单。2.5 节描述了 GLUI 的菜单和可利用的控制工具。

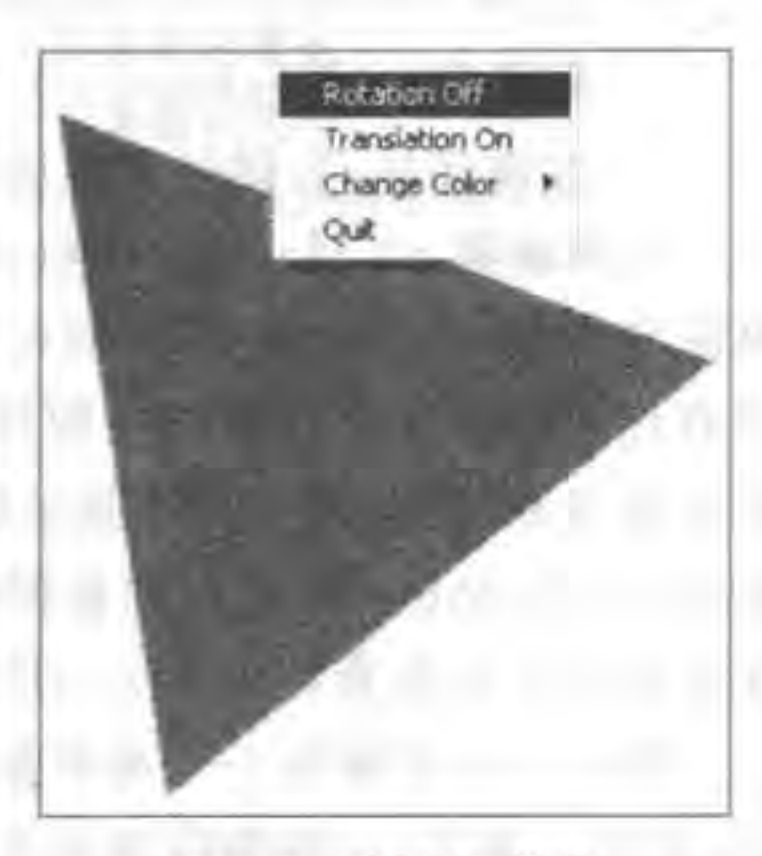


图 2.3　GLUT 菜单

图 2.4 展示的伪代码是事件驱动程序中 main 函数例子的一个基本框架，它为几类基本的事件类型，注册了必要的回调函数。本书中的大多数程序都以此框架（或稍作变化）为基础。主要有 5 种基本事件类型，每种都可以使用 GLUT 函数注册。

- glutDisplayFunc (myDisplay)：无论系统何时决定重画一个屏幕窗口，它都会发出一个重新绘制事件。出现下列情况之一就会发生该事件：第一次打开窗口，或覆盖它的窗口被移走而露出该窗口。函数 myDisplay()在这里被注册为重新绘制事件的回调函数。
- glutReshapeFunc (myReshape)：用户可以对屏幕窗口的形状进行调整，通常是用

```
// 在此包含 OpenGL 库(像图 2.11 一样)
void main()
{
    glutDisplayFunc(myDisplay);              //注册重绘函数
    glutReshapeFunc(myReshape);              //注册改变窗口形状函数
    glutMouseFunc(myMouse);                  //注册鼠标动作函数
    glutMotionFunc(myMotionFunc);            //注册鼠标运动函数
    glutKeyboardFunc(myKeyboard);            //注册键盘动作函数
    ...可能初始化其他工作...
    glutMainLoop();                          //进入主循环等待事件发生
}
...所有回调函数功能在此定义
```

图 2.4 使用 OpenGL 和 GLUT 的一个事件驱动程序样本

鼠标将窗口的一个角拉伸到一个新位置(只移动窗口并不产生该事件)。函数 myReshape()在这里被注册为改变窗口形状的事件。正如我们将要看到的,myReshape ()自动传送参数,报告被调整的窗口的新宽度和高度。

- glutMouseFunc (myMouse):当按下或释放鼠标的某个按钮时,就发生了鼠标事件。函数 myMouse()在这里被注册为鼠标事件发生时所调用的函数,它自动传送描述鼠标的位置和按钮状态参数。
- glutMotionFunc (myMotionFunc):当按下一个或几个鼠标按钮并且鼠标移动时,一个鼠标运动事件产生了。注意,正如我们在下面的例子所看到的一样,对于该事件的产生,鼠标按钮无需释放。另一方面,鼠标在没有按钮按下的情况下进入一个窗口,glutPassiveMotionFunc()将产生一个事件。
- glutKeyboardFunc(myKeyboard):这个函数用按下或释放键盘上按键的事件来注册函数 myKeyboard ()。myKeyboard ()自动传送参数,报告哪个键被按下。为了方便起见,它同时还传送按下键时鼠标的 x 和 y 坐标(相对屏幕窗口)。

回调函数 myMouse、myKeyboard 等由程序员命名,只要它们的名字不相同就可以。如果没有定义 myMouse,单击鼠标没有任何效果。同样,如果没有定义 myKeyboard,程序就没有任何的键盘交互功能。程序员应该注意,使用鼠标或键盘函数时,glutDisplayFunc()不会被自动调用,即重新绘制函数需要在鼠标或键盘回调函数中被显式的调用。相反,函数 glutPostRedisplay()将会刷新窗口显示,并且可以在程序中的任何位置被调用。2.4 节橡皮矩形例子中能看到函数 glutPostRedisplay()的作用。

图 2.4 中的最后一个函数是 glutMainLoop()。当执行这个函数时,程序绘制完初始图形,并进入循环,一直等待事件发生。用鼠标单击关闭窗口按钮(大多数情况下,关闭窗口按钮在每个窗口的右上角),程序正常终止。

2.1.3 如何打开一个窗口画图

画图的第一个任务就是打开一个用于画图的屏幕窗口。这个过程相当复杂,并且与系

统相关。因为 OpenGL 函数是设备无关的，所以它们对于任何指定系统中的窗口控制不提供任何支持。但是前面介绍过的 GLUT，却含有可以在所使用系统中打开窗口的函数。

图 2.5 扩充了上面的框架，显示了将在屏幕窗口中绘制图形的完整 main()函数。前 5 个函数调用工具包打开用于绘图的窗口。在你的第一个图形程序中，可以原样复制这些函数。图 2.5 中也给出了相应的注释，标出了多种参数的含义，以及怎样替换这些参数实现某种效果的方法。前 5 个函数初始化并显示屏幕窗口。下面简述每个函数。

- glutInit (&argc,argv)：该函数初始化工具包，其参数是传送有关命令行信息的标准参数，这里不会用到它们。
- glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)：该函数指明显示屏幕窗口如何被初始化。内置常数 GLUT_SINGLE 和 GLUT_RGB 是“或”的关系，表明应分配单个显示缓存，而且颜色由所需的红、绿、蓝三色的数量来指定(第 3 章将讨论产生平滑动画的双缓存模式)。
- glutInitWindowSize(640, 480)：该函数指定屏幕窗口的初始尺寸，宽 640 像素，高 480 像素。程序运行时，用户可以根据需要调整窗口的大小。
- glutInitWindowPosition(100, 150)：该函数指定窗口在屏幕上相对于左上角的位置，距左边 100 个像素，距顶端 150 个像素。程序运行时，用户可以根据需要移动窗口。
- glutCreateWindow("my first attempt")：该函数打开并显示屏幕窗口，并将标题 "my first attempt"(意为：我的第一次尝试)置于标题栏中。

```
//适当的# include 语句——参见图 2.11
void main(int argc, char ** argv)
{
    glutInit(&argc, argv);                          //初始化工具包
    glutInitDisplayMode(GLUT_SINGLE|GLUT_RGB);  //设置显示模式
    glutInitWindowSize(640,480);                    //设置窗口大小
    glutInitWindowPosition(100, 150);               //设置窗口在屏幕上的位置
    glutCreateWindow("my first attempt");           //打开屏幕窗口

    // 注册回调函数
      glutDisplayFunc(myDisplay);
      glutReshapeFunc(myReshape);
      glutMouseFunc(myMouse);
      glutKeyboardFunc(myKeyboard);

      myInit();                                     //必要的其他初始化工作
      glutMainLoop();                               //进入循环
}
```

图 2.5 使用 OpenGL 实用工具包 GLUT 打开初始窗口绘图源代码

main()中的其他函数注册前面描述过的回调函数，执行其他初始化工作，并且启动事

件循环处理。

2.2 OpenGL 的基本图形元素

我们要开发的编程技术可以用于绘制许多的几何形状,这些形状将组成有趣的图形。绘制命令将被放在与某个重新绘制事件相关联的回调函数中,例如上文提到的函数 myDisplay()。

我们首先必须建立一个描述图形对象的坐标系,并规定它们出现在屏幕窗口上的位置。计算机图形编程工作似乎就是不断地定义并管理各种坐标系。所以,这里以简单坐标系为开端,然后再进入更复杂的工作。

我们从一个直观的坐标系开始。该坐标系直接捆绑在屏幕窗口的坐标系上(如图 2.1(c)所示),其测量单位是像素。我们的第一个屏幕窗口例子如图 2.6 所示,宽为 640 像素,高为 480 像素。x 坐标从左边界的 0 增加到右边界的 639,y 坐标从底边的 0 增加到上边的 479。在讨论一些基本的元素后,我们将展示如何建立这种和其他种类的坐标系。

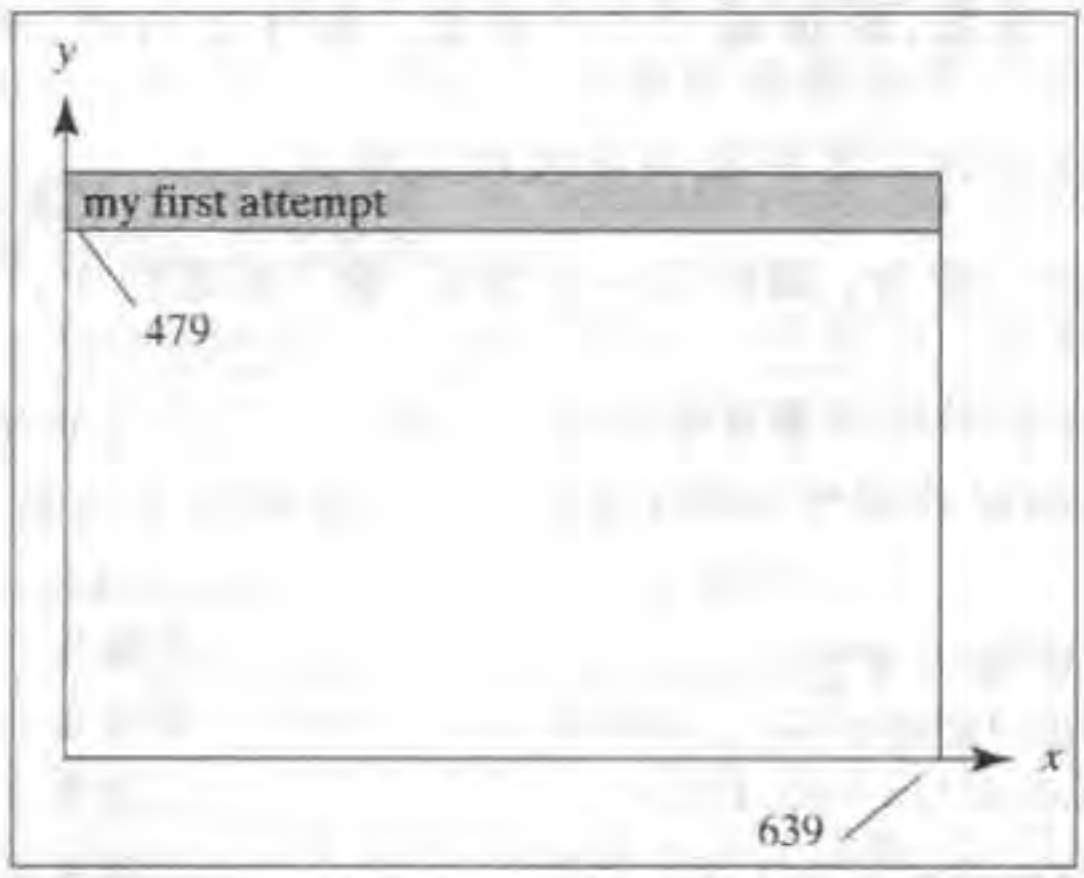


图 2.6 初始绘图坐标系

OpenGL 提供了绘制第一章所述的所有输出元素的工具。其中大多数,如点、线、折线和多边形,都由一个或多个顶点所定义。为了在 OpenGL 中绘制这些对象,必须传送一系列的顶点。这些顶点应该处于 glBegin()和 glEnd()这两个 OpenGL 函数之间。函数 glBegin()的参数确定要画哪个对象,指示 OpenGL 开始收集绘制对象的元素数据,glEnd()命令结束并完成对象的元素列表,并将所有数据送到 OpenGL 图形绘制管道中进行绘制。

例如,图 2.7 显示了在宽 640 像素,高 480 像素的窗口中画的 3 个点。这些点是用下面的命令序列绘制的:

```
glBegin (GL_POINTS);      //画独立的点
  glVertex2i (100, 50);   //点
  glVertex2i (100, 130);
  glVertex2i (150, 130);
glEnd();
```

常数 GL_POINTS 内置于 OpenGL 中。为了绘制其他类型的元素，可以将 GL_POINTS 置换成 GL_ LINES、GL_ POLYGON 等。以后会依次介绍它们。

随后将会看到，这些命令将顶点的信息送到图形绘制管道中。在那里，它们要经历几个处理步骤（详见第 5 章）。虽然实际的 OpenGL 绘制管道包含许多步骤，对于当前这个目标（画 3 点），只考虑管道按几个指定顺序操作，有哪些元素被送去显示就行了。

OpenGL 中的许多函数，如 glVertex2i()或 glColor3f()，有几个可变的部分，据此可以区别传送给函数的参数数目及类型。图 2.8 展示了不同的函数格式。

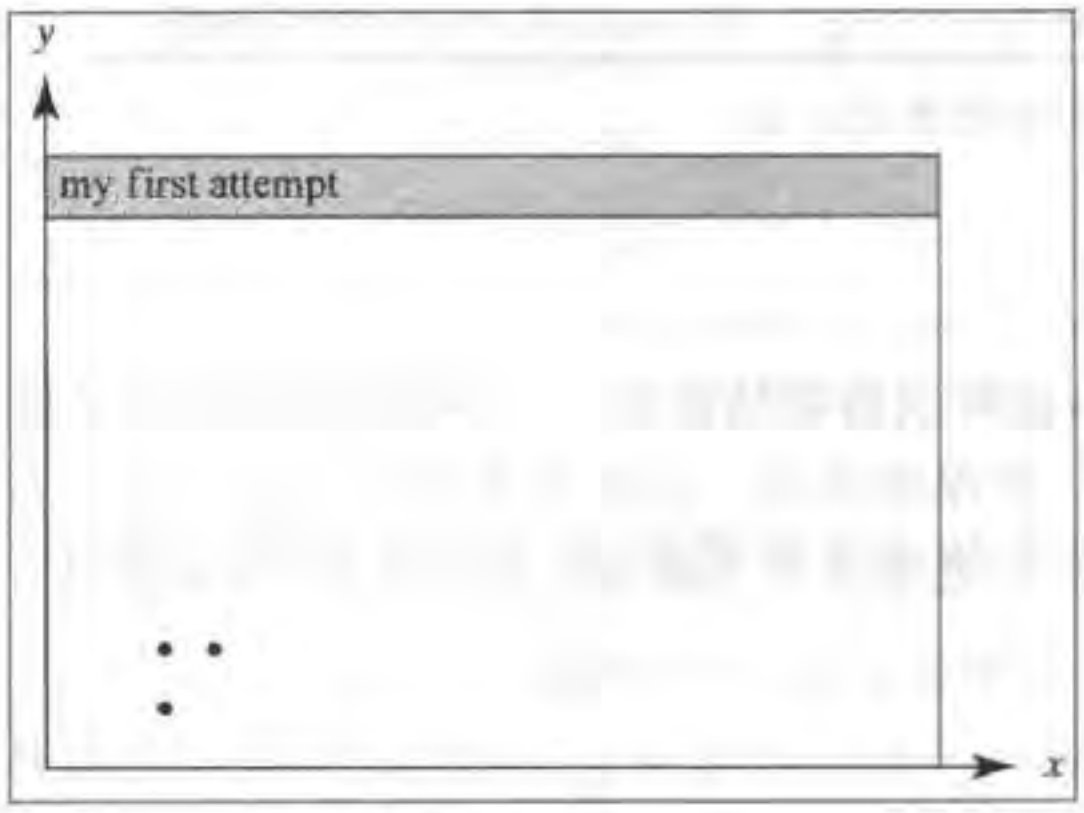


图 2.7　画 3 个点

图 2.8　OpenGL 命令格式

前缀 gl 表明函数来自基本的 OpenGL 库（相对的，glut 来自 GL 实用工具包 GLUT），接着就是基本命令的词根（例如 vertex 和 color），再接下来是发送给函数的参数数目（常常是 3 或 4 个），最后是参数的类型（i 表示 32 位整型数，f 表示浮点数等，后面有进一步描述）。为描述方便，以后如果仅仅使用基本命令而不去考虑函数的参数特性，我们用一个星号代替，如 glVertex * ()。

例如，为了生成上述同样的 3 个点，可以使用下面的命令，用浮点值代替整数值。

```
glBegin(GL_ POINTS);
    glVertex2f(100.0,50.0); //用浮点类型指定点
    glVertex2f(100.0,130.0);
    glVertex2f(150.0,130.0);
glEnd();
```

OpenGL 数据类型

OpenGL 内部使用特定的数据类型。例如，函数 glVertex2i()就使用 32 位的整数值。众所周知，有些系统将 C 或C++ 数据类型 int 理解为 16 位的整型，而有些系统则把它当作 32 位的整型。为在不同的系统中得到相同的显示，我们建议在程序中使用数据类型 GLint。同样，对于 float 或 double 类型的大小，也没有一致的标准。图 2.9 列出了 OpenGL 数据类型，有些类型在本书的后面章节才会碰到。

后 缀	数据类型	典型的 C 或C++ 类型	OpenGL 类型名
b	8 位整型	signed char	GLbyte
s	16 位整型	short	GLshort
i	32 位整型	int 或 long	GLint,GLsizei
f	32 位浮点型	float	GLfloat,GLclampf
d	64 位浮点型	double	GLdouble,GLclampd
ub	8 位无符号整型	unsigned char	GLubyte,GLboolean
us	16 位无符号整型	unsigned short	GLushort
ui	32 位无符号整型	unsigned int 或 unsigned long	GLuint,GLenum,GLbitfield

图 2.9 命令后缀和参数数据类型

OpenGL 状态

OpenGL 是由许多状态变量组成的状态机,这些状态包括点的大小、绘图的颜色和屏幕窗口大小等。在给定新值之前,状态变量的值一直保持不变。点的大小可以用 glPointSize()来设置,它是一个浮点变量。如果该变量是 3.0,通常这个点就会绘制成方形,每边都是 3 个像素。

画图的颜色可以用下面这条语句来设置:

```
glColor3f(red, green, blue);
```

其中红色、绿色和蓝色的值可以在 0.0~1.0 之间变化。例如,常见的颜色由下列一定数量的红色、绿色和蓝色构成(灰色由相同数量的红色,绿色和蓝色组成):

```
glColor3f(1.0,0.0,0.0);     //设置红色
glColor3f(0.0,1.0,0.0):     //设置绿色
glColor3f(0.0,0.0,1.0):     //设置蓝色
glColor3f(0.0,0.0,0.0):     //设置黑色
glColor3f(0.7,0.7,0.7):     //设置浅灰
glColor3f(0.2,0.2,0.2):     //设置中灰
glColor3f(0.1,0.1,0.1):     //设置深灰
glColor3f(1.0,1.0,1.0):     //设置亮白
glColor3f(1.0,1.0,0.0):     //设置亮黄
glColor3f(1.0,0.0,1.0):     //设置洋红
glColor3f(0.0,1.0,1.0):     //设置青色
```

第 11 章将详细讨论颜色。背景颜色用 glClearColor (red, green, blue, alpha)来设定,其中 alpha 用来设置透明度,后面将会详细讨论它(这里使用 0.0)。注意这条指令只是为后面的使用设置状态变量,它不做任何看得见的工作。为了将整个窗口都清除成背景色,需使用 glClear(GL_COLOR_BUFFER_BIT)函数。参数 GL_COLOR_BUFFER_BIT 是另一个 OpenGL 内置常数。

坐标系的建立

现在我们建立初始坐标系的方法看起来比较模糊,但是到下一章,讨论窗口、视口和剪

裁后就会比较清晰了。在此只考虑几条必需的命令。图 2.10 中的函数 myInit()用来设定坐标系很方便。在后面将会看到，OpenGL 会例行公事地执行很多变换，这些变换用矩阵来完成。在 myInit()中的命令通过控制管理矩阵来达到此目的。图 2.10 中的例程 gluOrtho2D()用来设定屏幕窗口所需要的变换，图 2.10 将屏幕窗口设置为 640 像素宽与 480 像素高。gluOrtho2D(0, 640.0, 0.0, 480.0)例程来源于 GLU 库。

```
void myInit(void)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
```

图 2.10　建立一个简单的坐标系

组合在一起：一个完整的 OpenGL 程序

图 2.11 是一个完整的程序，用来绘制图 2.7 中简单的 3 个圆点。随后就会看到，该程序很容易扩展成绘制更有趣物体的程序。初始化函数 myInit()设置坐标系、点的大小、背景颜色以及绘图颜色。绘图函数(由程序员提供)封装在回调函数 myDisplay()中。当然，回调函数是必须注册的。因为这个程序没有交互功能，所以就没有使用其他的回调函数。绘制完点以后就调用 glFlush()，以保证所有的数据被完全处理并显示。这对某些运行在网络上的系统来说是非常重要的：数据被缓冲到服务器上，只有当缓冲区的数据填充满或执行 glFlush()时，数据才会被发送到客户端显示。

2.2.1　几个点丛绘制的例子

点丛是由大量点组成的某种图案。下面介绍几个有趣的点丛例子，用基于图 2.11 的程序很容易产生并绘制这些点丛：在为重新绘制事件注册的回调函数 glutDisplayFunc()中，做一些适当的改变就可产生每个点丛。你应该去实现并测试每个例子，熟悉它们以积累你的经验。

例 2.2.1　北斗星群

图 2.12 显示了用 8 个圆点表示的北斗星图，在夜晚的天空中经常会看到它们。

北斗星群中这 8 颗星星(从夜空中的某一方位看)的命名和位置由下列数据给出：

{Dubhe, 289, 190}、{Merak, 320, 128}、{Phecda, 239, 67}、{Megrez, 194, 101}、{Alioth, 129, 83}、{Mizar, 75, 73}、{Alcor, 74, 74}、{Alkaid, 20, 10}(一些人裸眼只能看到北斗星群中的 7 颗。从前，在美国本土的一些土著部落中，它被用来测试眼睛的敏锐度，赢者将获得一晚的晚餐)。由于所包含的数据点非常少，所以很容易直接将它们列出来，或者将这些数据直接写到代码中。另一方面，绘制许多点时，数据直接列表的方式就不明智

```
#include <windows.h>                                    //根据系统需要使用
#include <gl/Gl.h>
#include <gl/glu.h>
#include <gl/glut.h>
//<<<<<<<<<<<<<<<<<<<<<<<<<<< myInit >>>>>>>>>>>>>>>>>>>>>>>>>>>
void myInit(void)
{
    glClearColor(1.0,1.0,1.0,0.0);                      //设置背景颜色为亮白
    glColor3f(0.0f, 0.0f, 0.0f);                        //设置绘图颜色为黑色
    glPointSize(4.0);                                   //设置点的大小为 4×4 像素
    glMatrixMode(GL_PROJECTION);                        //设置合适的矩阵—后面将解释
    glLoadIdentity();                                   //后面将解释
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);                 //接下来将完整地说明
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<< myDisplay >>>>>>>>>>>>>>>>>>>>>>>>>>>
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);                       //清屏
    glBegin(GL_POINTS);
        glVertex2i(100, 50);                            //画一些点(不知道多少)
        glVertex2i(100, 130);
        glVertex2i(150, 130);
    glEnd();
    glFlush();                                          //送所有输出到显示设备
}

//<<<<<<<<<<<<<<<<<<
```

例 2.2.2　Sierpinski 垫片

图 2.13 显示了迷人的 Sierpinski 垫片。它的点丛是由程序生成的，这就意味着每个点都由一个过程法则来确定。虽然这个法则非常简单，但是最后的图案显示的却是无穷复杂的分形(参见附录 5)。我们首先以直观的方式处理这些生成 Sierpinski 垫片的规则，并提供实现 Sierpinski 垫片的一段代码。

图 2.12　北斗星群

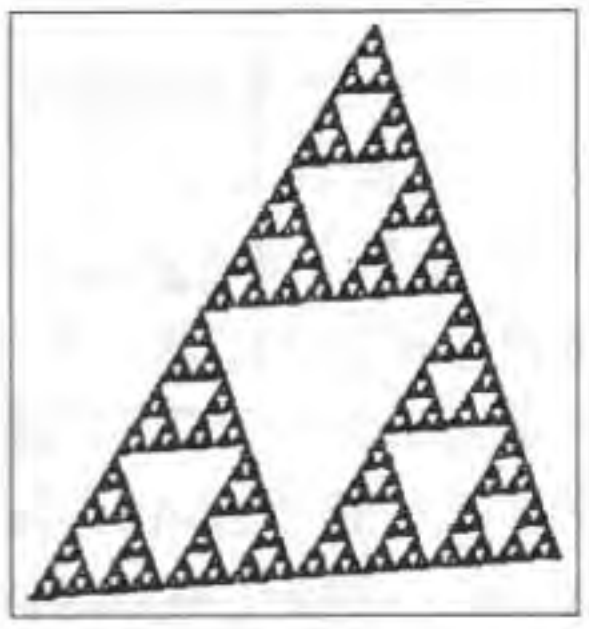


图 2.13　赛平斯基垫片

怎样产生 Sierpinski 垫片

Sierpinski 垫片通过多次调用 drawDot()生成，其参数是(x0,y0)，(x1,y1)，(x2,y2)等一系列点的位置。其中，这些位置参数由一个简单算法确定。drawDot 函数为：

```
void drawDot(Glint x,Glint y)
{
  glBegin(GL_POINTS);
      glVertex2i(x,y);
  glEnd();
}
```

Sierpinski 算法如下：第 k 个点表示为 $P_k=(x_k, y_k)$。每个点都基于前一个点 P_{k-1}。过程如下：

(1) 选择 3 个固定的点 T_0、T_1 和 T_2，构成一个三角形，称为每个 Sierpinski 垫片的父三角形，如图 2.14(a)所示。

(2) 随机挑选父三角形顶点 T_0、T_1 和 T_2 中的一点作为要绘制图形的初始点 P_0。

迭代下面的(3)～(5)步，直至图案填充完毕。

(3) 随机选择 T_0、T_1 和 T_2 中的一点，称为 T。

(4) 构造下一个点 P_k，作为 T 和前一个已建好的点 P_{k-1}之间的中间点[①]。也就是说，$P_k=P_{k-1}$和 T 之间的中间点。

(5) 用 drawDot()绘制 P_k。

图 2.14(b)展示了该过程的几次迭代。假定初始点 P_0 正好是 T_0，并让 T_1 是下一个被

① 为了找到(3, 12)和(5, 37)两点的中间点，分别将它们的 x 和 y 值平均后取整：(3, 12)和(5, 37)的中间点为((3+5)/2, (12+37)/2)=(4, 24.5)。注意，坐标无需是整数。

选中的点,那么所绘制的 P_1 就是 T_0 和 T_1 之间的中点。假定下一个选中 T_2,那么 P_2 就是 P_1 和 T_2 的中点。接着,假定再次选中 T_1,那么所形成的 P_3 就是如图所示的样子,依此类推。该过程不断地生成并绘制出新点(概念上来说是永不停止的),Sierpinski 垫片图形很快就出现了。

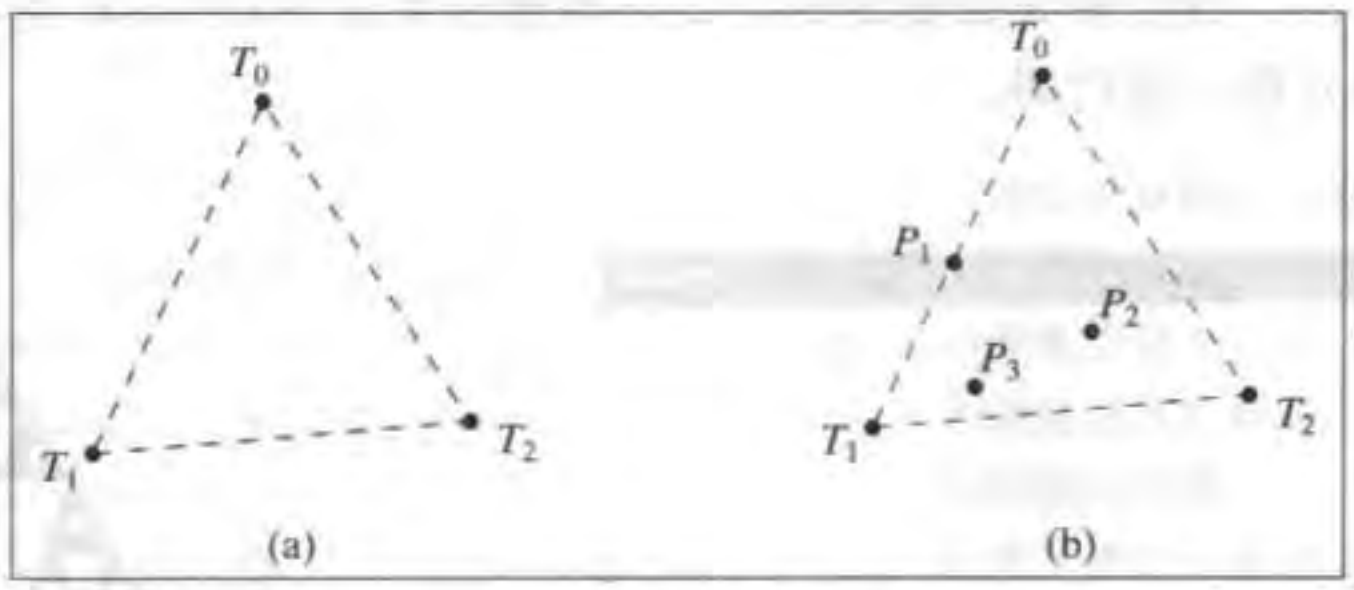


图 2.14 构建赛平斯基垫片

```
//一定要包括 math 或 time 头文件
//sierpinski_render 函数
void sierpinski_render(void)
{
    glClear(GL_COLOR_BUFFER_BIT);                    // 清屏
    GLintPoint T[3]={{10,10},{600,10},{300, 600}};      //定义三角形顶点
    int index=random()%3;                            //随机选择初始顶点
    GLintPoint point=T[index];                       //生成一个含 3 个顶点的数组
    drawDot(point.x, point.y);
    for(int i=0; i<55000; i++)                       //画 55000 个点
    {
        index=random()%3;
        point.x=(point.x+T[index].x) / 2;
        point.y=(point.y+T[index].y) / 2;
        drawDot(point.x,point.y);
    }
    glFlush();
}
```

图 2.15 产生赛平斯基垫片函数

可以很方便地定义一个简单的结构 GLPoint,用于描述坐标为浮点数的点:

```
struct GLPoint {
    GLfloat x,y;
};
```

然后,我们创建并初始化包含 3 个坐标 T[0]、T[1]和 T[2]的数组来定义三角形的 3 个顶点,例如 GLPoint T[3] = {{10.0, 10.0}, {300.0, 30.0}, {200.0, 300.0}}。在这个例子中,点产生后不需要将每个点 P_k 都存储起来,因为我们只是想要绘制它,然后继续下面

的步骤。因此，我们设一个变量 point 来保存这个一直在变化的点，每次迭代中 point 都会被更新成新值。

用 index=rand()%3 来随机选择点 T[i]，rand()以相同的概率返回 0、1 或 2 中的任一值。图 2.15 列出了 sierpinski_render 函数，该函数运行完将生成 55 000 个 Sierpinski 垫片点。在你的程序中，用函数 Sierpinski()替换回调函数 myDisplay()即可。

rand()是什么

rand()是一个返回 0 和某个上限值之间的伪随机数(看起来是随机的数)的函数。既然我们想选择父三角形的 3 个顶点中的一个，我们调用 index=rand()%3 得到了 0～2 之间(包含 0 和 2)的一个值。rand()是标准C++ 库的一部分，它返回 0 与某个上限值(我们的机器中这个值是 32 767，它存储在 RAND_MAX 变量中)之间的值。对于大多数的随机数产生器，其生成的数似乎是随机的。但实际上，一旦给定第一个数后，其后生成的数也就确定了。因此，我们需给它一个称为“种子”的、合适的初始值。在运行时，这个种子值与系统运行时钟相关联，因为系统时间是一个一直在变化的值。每一个时刻，种子是唯一的。在程序运行的每个时刻，使用系统时间可以真正地返回一个不同的随机序列。

其他有意思的 Sierpinski 垫片

可以制作多种 Sierpinski 垫片，例如改变点的颜色，或用鼠标将父三角形的顶点拖到新的位置。后者将在下面的章节中描述，本书的网页上提供了 Sierpinski 垫片的应用。

例 2.2.3　用点集绘制函数

假如想学习某个数学函数 $f(x)$随 x 变化的情况，例如，下面这个函数：

$$f(x) = e^{|-x|}\cos(2\pi x)$$

当 x 在 0～4 之间变化时，该函数如何变化呢？图 2.16 中所示的 $f(x)$对 x 的曲线，就可以很好地将它们之间的变化关系反映出来。

为了绘制该函数曲线，可以简单地在 x 值闭区间集合[0，4]中进行采样，并在坐标$(x, f(x))$处画一个点。在连续的 x 值之间，选择合适的增量，例如 0.005，进行采样，该过程可以用以下算法描述：

图 2.16　$e^{|-x|}\cos(2\pi x)$对 x 的函数曲线

```
glBegin(GL_ POINTS) ;
  for(GLdouble  x=0; x<4.0;x+=0.005)
    glVertex2d(x,f(x));
glEnd();
glFlush();
```

但是这里存在一个问题：它所产生的图形特别小，因为 0～4 之间的 x 值位于屏幕窗口左下角的前 4 个像素上。而且，任何 $f(.)$的负值位于窗口的下面，我们根本看不到它们。所以我们需要进行缩放并重置这些数值，以使它们所产生的曲线正好出现在合适的屏幕窗口区域内。这里采用强制手段来实现该目标，即挑选一些数值使绘制的图形恰当地显示在屏幕上。随后我们将开发一个一般性过程来处理这些调整，即所谓的“将世界坐标系变换为

窗口坐标系"的过程。

x 方向缩放 假定需要在 0～4 范围内进行比例变换,使它能覆盖屏幕窗口的整个宽度(屏幕窗口的整个宽度用 screenWidth 表示,单位为像素)。我们只须按 screenWidth/4.0 来缩放所有的 x 值即可,即:

```
sx=x * screenWidth/4.0;
```

当 $x=0$ 时,$sx=0$,当 $x=4.0$ 时,$sx=$screenWidth,这正是我们所需要的。

y 方向缩放与平移 函数 $f(x)$ 的取值在 -1.0～1.0 之间,所以必须同时进行缩放并移动它们。假设屏幕窗口的高为 screenHeight 个像素,为了将曲线置于窗口的中心,就应缩放 screenHeight/2,并向上平移 screenHeight/2:

```
sy=(y+1.0) * screenHeight/2.0;
```

这里,当 $y=-1.0$ 时,$sy=0$;当 $y=1.0$ 时,$sy=$screenHeight。

请注意这种将 x 转换成 sx,以及将 y 转换成 sy 的转换方式是:

$$sx = A * x + B$$
$$sy = C * y + D \tag{2.1}$$

只要适当地选择常数 A、B、C 和 D 的值就可以了。A 和 C 执行缩放;B 和 D 执行平移。缩放和平移操作是仿射变换的一种基本形式。第五章将深入探讨仿射变换,它提供了一种简便统一的方法,可以将任意指定范围内的 x 和 y 映射到屏幕窗口上。

当 A、B、C 和 D 的值被适当地设置后,就可以使用下面的代码来绘制点曲线。

```
GLdouble A, B, C, D, x;
A=screenWidth/4.0;
B=0.0;
C=screenHeight/2.0;
D=C;
glBegin(GL_POINTS);
    for(x=0; x<4.0; x+=0.005)
        glVertex2d(A * x+B, C * f(x)+D);
glEnd();
glFlush();
```

图 2.17 给出了函数绘制的完整程序,阐明了多种成分是怎样相互匹配在一起的。该程序的初始化与图 2.7 中绘制三个点的程序初始化有些相似。注意屏幕窗口的宽度和高度被定义为常数,并且使用在程序中需要的地方。

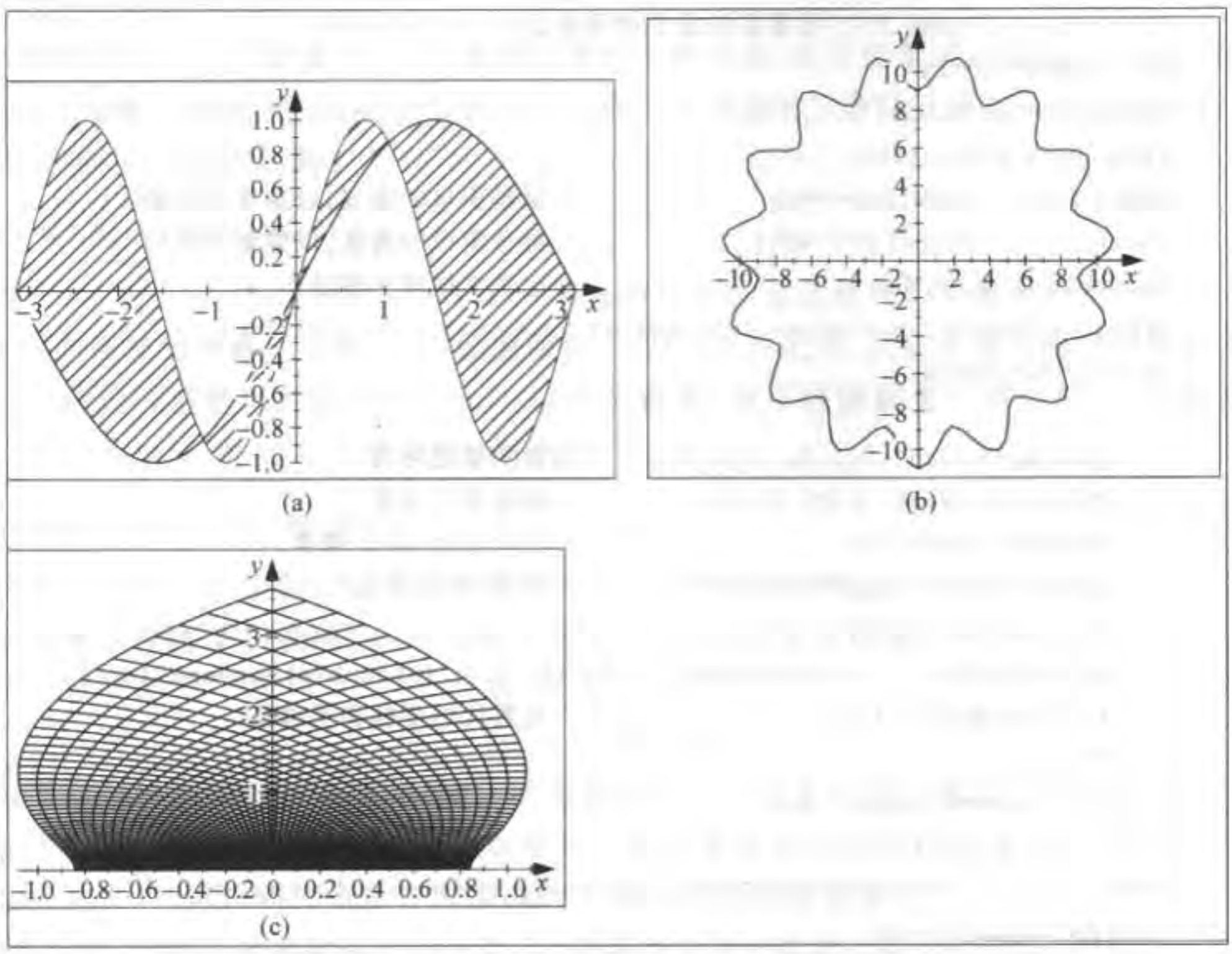
图 2.18(a)显示了两个重叠的函数,一个是 $f(x)=\sin(x)$,x 从 $-\pi$ 到 π。第二个函数是 $f(x)=\sin(2x)$,x 的变化范围与前者相同。

图 2.18(b)显示了仅用一个参数(参数 a)表示的曲线(第 3 章将详细介绍曲线的参数表示方法),x 坐标定义为 $x(a)=10+\sin(15 * a)$,变量 a 在 0～2π 间变化;y 坐标定义为 $y(a)=\sin(a)$,变量 a 在 0～2π 间变化。

图 2.18(c)是复变函数 $\sin(z^2)$ 的曲线,复变量 z 在一个正方形网格上变化。每个 z 的实部和虚部值映射成 $\sin(z^2)$ 的实数和虚数,并逐点绘制出图 2.18(c)。

```
#include <windows.h>//根据系统需要使用合适的 include 语句
#include <math.h>
#include <gl/Gl.h>
#include <gl/glut.h>
const int screenWidth=640;                  //屏幕窗口的宽度,以像素为单位
const int screenHeight=480;                 //屏幕窗口的高度,以像素为单位
GLdouble A, B, C, D;                        //比例变换和平移值
//<<<<<<<<<<<<<<<<<<<<<<<<<myInit >>>>>>>>>>>>>>>>>>>>>>>
void myInit(void)
{
    glClearColor(1.0,1.0,1.0,0.0);          //背景颜色为白
    glColor3f(0.0f, 0.0f, 0.0f);            //画图颜色为黑
    glPointSize(2.0);                       //点大小为 2×2 像素
    glMatrixMode(GL_PROJECTION);            //设置"相机形状"
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble)screenWidth, 0.0, (GLdouble)screenHeight);
    A=screenWidth/4.0;                      //设置比例变换和平移值
    B=0.0;
    C=D=screenHeight / 2.0;
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<myDisplay >>>>>>>>>>>>>>>>>>>
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);           //清屏
    glBegin(GL_POINTS);
    for(GLdouble x=0; x <4.0 ; x+=0.005)
    {
        GLdouble func=exp(-x) * cos(2 * 3.14159265 * x);
        glVertex2d(A * x+B, C * func+D);
    }
    glEnd();
    glFlush();                              //送所有数据到显示
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<main >>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char ** argv)
{
    glutInit(&argc, argv);                  //初始工具包
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);     //设置显示模式
    glutInitWindowSize(screenWidth, screenHeight);  //设置窗口大小
    glutInitWindowPosition(100, 150);  //设置窗口在屏幕上的位置
    glutCreateWindow("Dot Plot of a Function");      //打开屏幕窗口
    glutDisplayFunc(myDisplay);                      //注册重画函数
    myInit();
    glutMainLoop();                                  //进入永久循环
```

图 2.17　一个完整的绘制点曲线函数程序图



2.18 三个不同的二维数学函数曲线

练习

练习 2.2.1 任意函数 $f()$ 的函数曲线。

考虑绘制 $f(x)=e^{|-x|}\cos(2\pi x)$ 的函数曲线，其中 x 从 x_{low} 变到 x_{high}，$f(x)$ 则从 y_{low} 到 y_{high}。求适当的缩放和平移因子，使曲线正好位于宽为 W 像素、高为 H 像素的屏幕窗口中。

2.3 OpenGL 中的直线绘制

如第 1 章所述，直线绘制是计算机图形学的基础算法。几乎每个图形系统都配有绘制直线的例程。使用 OpenGL 可以很方便的绘制直线：用 GL_LINES 作为 glBegin()的参数。为了绘制(40，100)和(202，96)两点间的直线，可以使用如下代码：

```
glBegin (GL_LINES);      //这里使用常数 GL_ LINES
    glVertex2i(40,100);
    glvertex2i(202,96);
glEnd();
```

为了方便，可以将这个代码置于例程 drawLine()中：

```
void drawLine (GLint x1, GLint y1, GLint x2, GLint y2)
```

```
{
    glBegin(GL_LINES);
    glVertex2i(x1,y1);
    glVertex2i(x2,y2);
    glEnd();
}
```

另一个例程 drawLineFloat()也可以类似地实现(想想如何实现)。

如果在 glBegin(GL_LINES)和 glEnd()之间指定了多个顶点,它们将会被成对的取出,并且在每对之间画出一条单独的线。图 2.19(a)所示的 tic-tac-toe 底板可以用下面的命令画出:

```
glBegin(GL_LINES);
  glVertex2i(10,20);      //第一条水平线
  glVertex2i(40,20);
  glVertex2i(20,10);      //第一条垂直线
  glVertex2i(20,40);
(在此的另外两条线还需调用 4 次 glVertex2i())
glEnd();
glFlush();
```

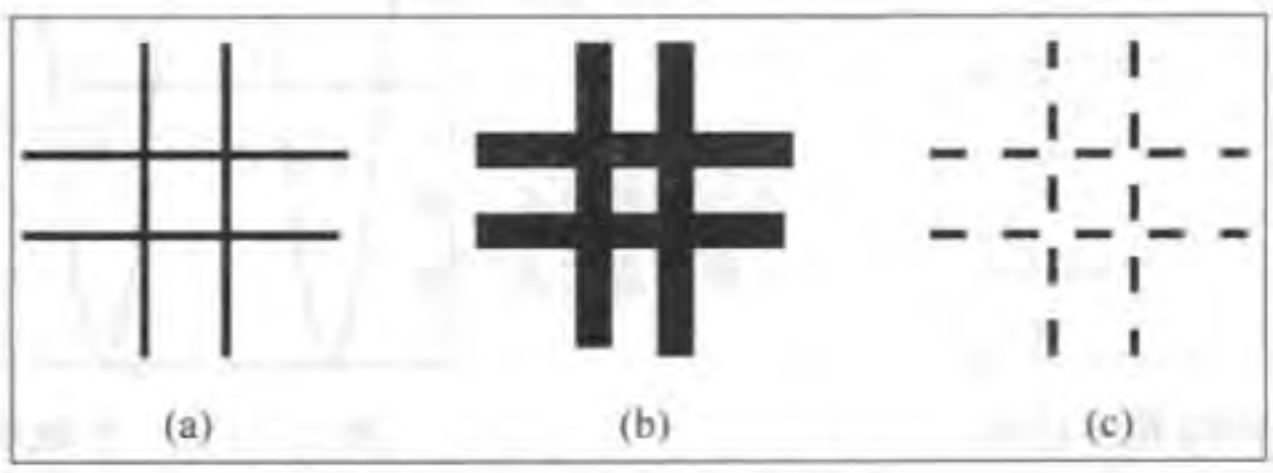


图 2.19　4 条线构成的简单图形

OpenGL 提供了设定线条属性的工具。线条颜色的设定方式和点颜色的设定方式相同,都是使用 glColor3f()。图 2.19(b)所示的粗线条使用 glLineWidth (4.0)设置,默认宽度是 1.0。图 2.19(c)展示了点划线(点和破折号)。

2.3.1　绘制折线和多边形

第 1 章介绍过,折线就是一系列头尾相连的线段的集合。它由一个有序点列表来描述,如:

$$p_0 = (x_0, y_0), \quad p_1 = (x_1, y_1), \cdots, p_n = (x_n, y_n) \tag{2.2}$$

在 OpenGL 中,折线称为线带,通过在 glBegin (GL_LINE_STRIP)和 glEnd()之间依次指定顶点对:p_0 到 p_1,p_2 到 p_3 等来绘制。例如下面的代码:

```
glBegin(GL_LINE_STRIP);          //画一条开放式折线
  glVertex2i(20,10);
  glVertex2i(50,10);
```

```
    glVertex2i(20,80);
    glVertex2i(50,80);
  glEnd();
  glFlush();
```

产生如图 2.20(a)所示的折线。折线的颜色、宽度及点划线等属性的设置和单条线段一样。如果需将最后一点和第一点相连,把折线变为多边形,只需用 GL_LINE_LOOP 替换 GL_LINE_ STRIP 即可,其多边形如图 2.20(b)所示。

用 GL_LINE_LOOP 绘制的多边形不能填充颜色或图案。要想绘制填充的多边形,必须使用 glBegin(GL_POLYGON),这将在后面介绍。

例 2.3.1 画线图

在例 2.2.3 中,我们看到的是在一系列(x_i, $f(x_i)$)位置处画点来绘制函数 $f(x)$相对 x 的曲线。线图是这种概念的延伸,只不过将这些点简单地用线段连在一起以形成折线。图 2.21 是一个基于下列函数的例子:

$$f(x) = 300 - 100\cos(2\pi x/100) + 30\cos(4\pi x/100) + 6\cos(6\pi x/100)$$

其中,x 步长为 3,共 100 步。

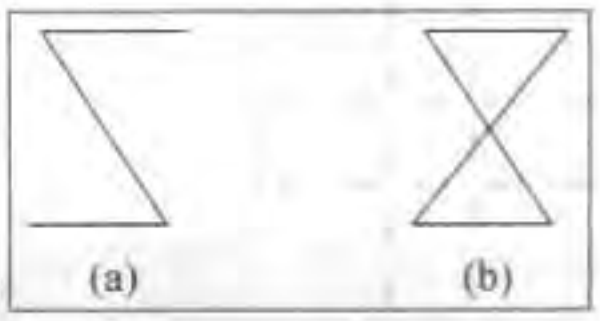


图 2.20 折线和多边形

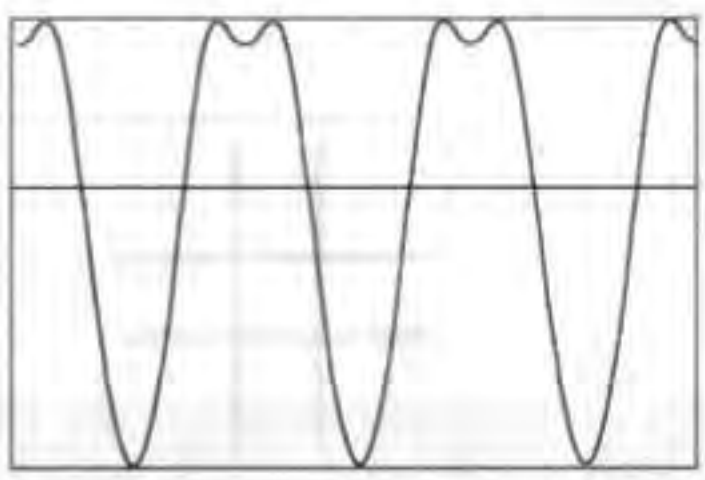


图 2.21 一个数学公式图

线图的处理过程几乎与点图一样,只需将图 2.17 稍微改动。象点图需做缩放和平移一样,为了将图放置在窗口中的合适位置,线图也需要做变换。常数 A、B、C 和 D 的计算方法与以前一样(参见等式(2.1))。图 2.22 给出了 myDisplay 函数中内部绘制循环必须做的修改。

```
<计算常数 A,B,C,D 用于缩放和平移>
glBegin(GL_LINE_STRIP);
    for(x=0; x<=300; x+=3)
        glVertex2d(A*x+B, C*f(x)+D);
glEnd();
glFlush();
```

图 2.22 线图函数代码片断

例 2.3.2 绘制存储在文件中的折线

大多数有趣的图形都是由包含很多线段的折线构成。将折线存储在文件中,使用起来比较方便。这样的图形可以随时根据需要进行绘制。

编写一个绘制存储在文件中的折线的程序并不难。图 2.23 就是一个这样的例子。

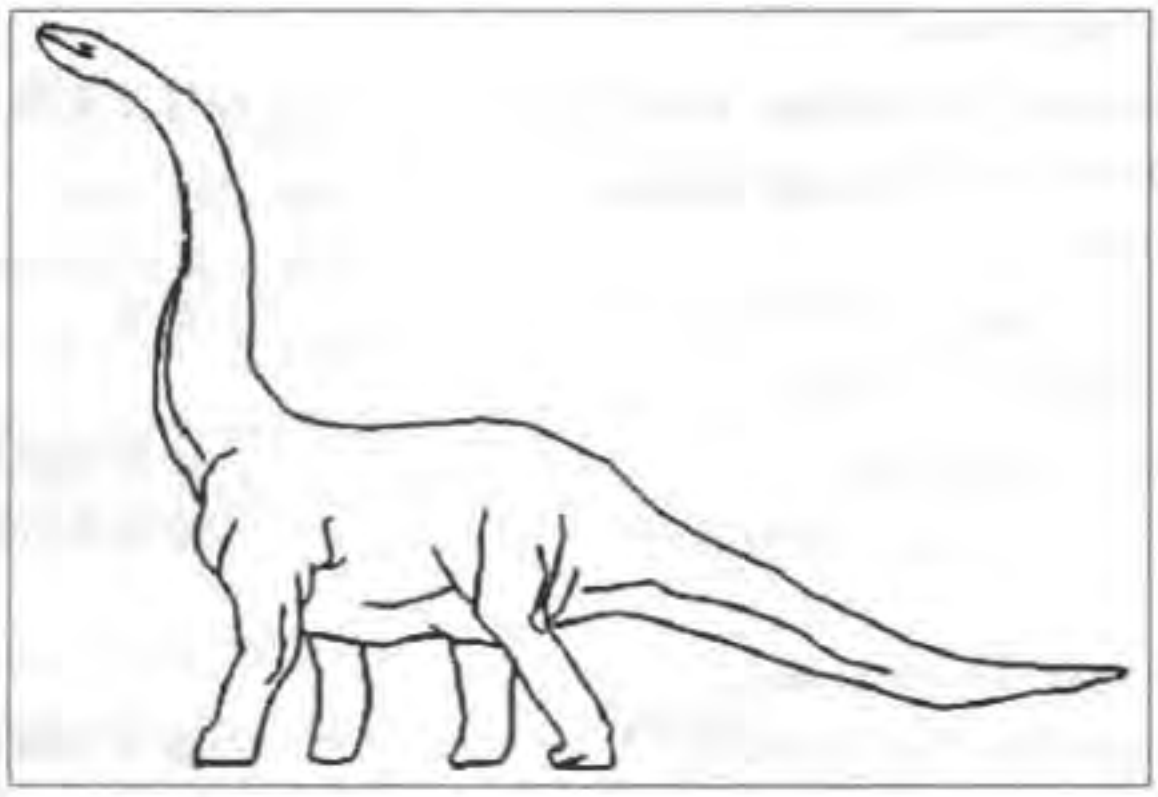


图 2.23　绘制存储在文件中的折线

假定文件 dino.dat 按下列格式包含一系列折线（其中注释部分不包含在文件内）。

```
21   文件中折线的数量
4                    第一条折线的点数
169 118              第一条折线的第一个点
174 120              第一条折线的第二个点
179 124
178 126
5                    第二条折线的点数
298 86               第二条折线的第一个点
304 92
310 104
314 114
314 119
29
32 435
10 439
...                  等等
```

图 2.24 给出了一个用C++ 编写的完整的函数 drawPolyLineFile()，该函数用于绘制存储在文件中的图形。读入文件的文件名（由字符串 fileName 指定），再依次读入每条折线并进行绘制。像 myDisplay()一样，程序使用函数 drawPolyLineFile()作为重新绘制事件的回调函数。必须确定 A、B、C 和 D 的值，以便正确地缩放这些折线。第 3 章将开发一种一般性的方法来完成这个任务。

这个版本的 drawPolyLineFile()只做了很少的错误检查。如果文件不能打开——可能是把错误的名字传送给该函数了——这个例程就简单地返回一个错误。如果文件中有坏数据，比如在需要整数的地方却出现了实数，其结果是不可预料的。现在只能把该例程当做开发更强版本例程的一个起点。这个函数的一些改进可以是：当每次调用这个特殊的函数时，不需要再重读文件，而是将折线数据存在内存里（在这里，这些数据将立刻用于绘制，用

```
{
    fstream inStream;
    inStream.open(fileName, ios ::in);          //打开文件
    if(inStream.fail())
        return;
    glClear(GL_COLOR_BUFFER_BIT);               //清屏
    GLint numpolys, numLines, x ,y;
    inStream>>numpolys;                         //读折线数量
    for(int j=0; j<numpolys; j++)               //读每条折线
    {
        inStream>>numLines;
        glBegin(GL_LINE_STRIP);                 //画下一条折线
        for (int i=0; i<numLines; i++)
        {
            inStream>>x>>y;                     //读下一对 x,y
            glVertex2i(x, y);
        }
        glEnd();
    }
    // 结束 j 循环
    glFlush();
    inStream.close();
}
//结束 drawPolyLineFile()
```

图 2.24 画存储在文件中的折线函数

后就丢弃了)。

例 2.3.3 参数化图形

图 2.25 展示了一个由几条折线组成的简易房子。它可以用图 2.26 中的部分代码绘制出来。但对于绘制门和窗,什么代码才合适呢?

图 2.26 不是一个灵活的方法。每个端点的位置都被强制写入代码中,所以 hardwirededHouse()只能绘制出相同大小和位置的房子。但如果我们把这个图参数化,并将参数值传送到例程中,就有更好的灵活性。用这种方法,可以绘制出有着不同参数值的物体簇。图 2.27 给出了这种方法,其中用参数指定房屋顶点的位置以及房子的宽度和高度。绘制烟囱、门和窗等细节留给读者作为练习。

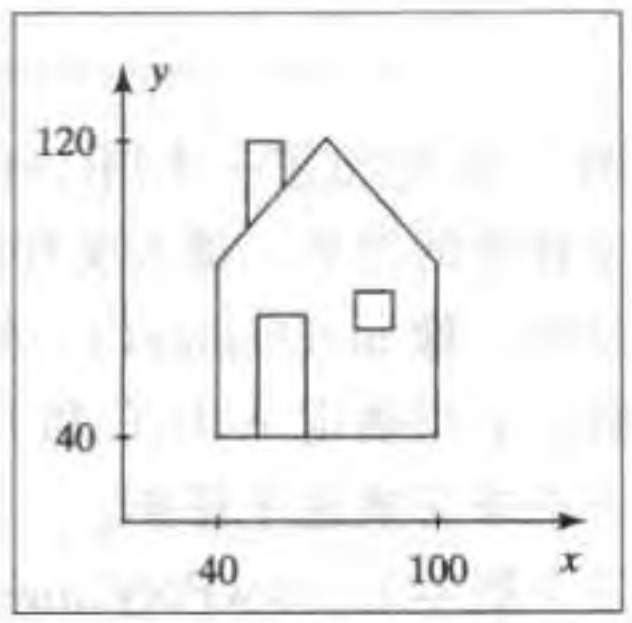


图 2.25 一栋房子

```
void hardwiredHouse(void)
{
    glBegin(GL_LINE_LOOP);
        glVertex2i(40, 40);          //画房子的外形
        glVertex2i(40, 90);
        glVertex2i(70, 120);
        glVertex2i(100, 90);
        glVertex2i(100, 40);
    glEnd();
    glBegin(GL_LINE_STRIP);
        glVertex2i(50, 100);         //画烟囱
        glVertex2i(50, 120);
        glVertex2i(60, 120);
        glVertex2i(60, 110);
    glEnd();
        //画门
        //画窗户
}
```

图 2.26 用固定尺寸画房子

```
void parameterizedHouse(GLintPoint peak, GLint width, GLint height)
 //房子的顶点: peak, 给定房子大小: width,height
 {
    glBegin(GL_LINE_LOOP);
      glVertex2i(peak.x, peak.y);        //画房子的外形
      glVertex2i(peak.x+width/2, peak.y-3*height/8);
      glVertex2i(peak.x+width/2  peak.y-height);
      glVertex2i(peak.x-width/2, peak.y-height);
      glVertex2i(peak.x-width/2, peak.y-3 * height/8);
    glEnd();
    //用同样的方式画烟囱
    //画门
    //画窗户
}
```

图 2.27 画一个参数化的房子

这个例程可以用于绘制图 2.28 所示的村庄，用不同的参数值多次调用 parameterizedHouse()就可以了(房子是如何翻转的？图中所有的房子都能用所给出的例程绘制吗)。

例 2.3.4 构造一个折线绘图器

我们将会看到，一些应用程序计算并用一个列表存储折线的顶点。所以自然会想到，应

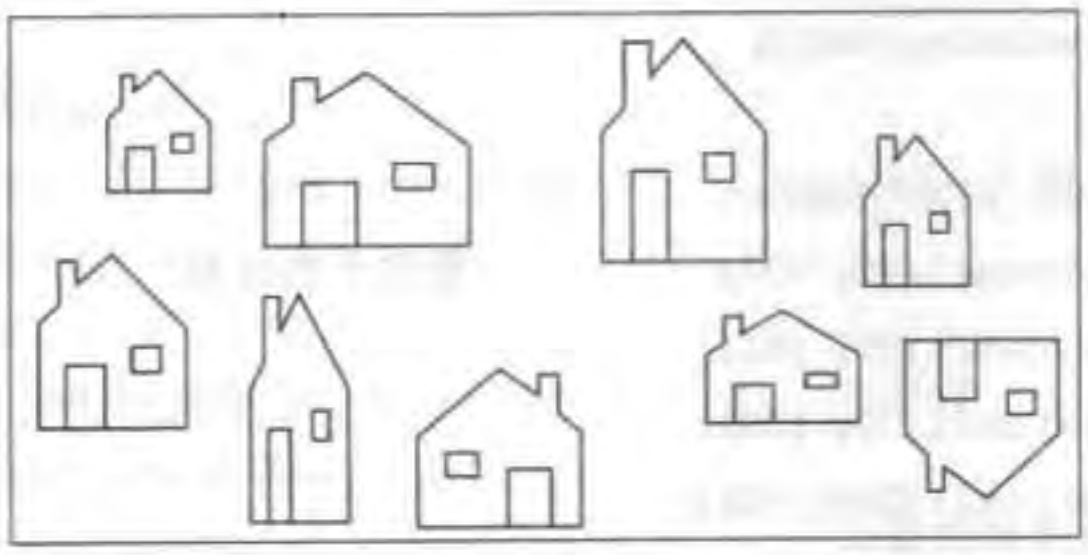


图 2.28 用 parameterizedHouse()画的村庄

该往不断扩大的例程工具箱中加入一个接受列表作为输入参数并绘制相应折线的函数。这个列表可能是数组或者链表的形式,在此使用数组形式,并定义图 2.29 中所示的类来保存数组。图中的 struct GLintPoint 将在下面的章节中讨论。对于那些想处理大量数据(可能超过 10 000 个数据)的程序员来说,C++ STL(标准模板库)容器可能是一个安全的数据结构。

```
struct GLintPoint {
Glint x, y;
};
class GlintPointArray {
  const int MAX_NUM=10000;
// 使用符号常数 MAX_NUM 方便以后改动
public:
  int num;
GLintPoint pt[MAX_NUM];
};
```

图 2.29 顶点数组的数据类型

图 2.30 给出了一个绘制折线的例程。这个例程采用了参数 closed:如果 closed 非 0,则折线中最后一个顶点就会与第一个顶点相连,使图形闭合成为一个多边形。closed 的值决定了 glBegin()的参数。这个例程只是简单地将折线的每个顶点送给 OpenGL。

```
void drawPolyLine(GlintPointArray poly, int closed)
{
    if(closed)
      glBegin(GL_LINE_LOOP);     //此情况下画闭合的折线
    else
      glBegin(GL_LINE_STRIP);    //此情况下画开放的折线
        for(int i=0; i<poly.num; i++)
            glVertex2i(poly.pt[i].x, poly.pt[i].y);
    glEnd();
    glFlush();
}
```

图 2.30 用链表绘制折线或多边形

2.3.2 使用 moveTo()和 lineTo()绘制线段

我们在 2.3 节一开始就看到，用 OpenGL 绘制线段是很容易的。既可以直接用一组命令 glBegin(GL_LINES)，…，glEnd()，也可以用象 drawLine()这样的函数来绘制线段。

前面说过，moveTo 和 lineTo 函数可以管理一个虚拟画笔，它的位置叫做当前位置或 CP。CP 很像一个 OpenGL 的状态变量，像画画的颜色和背景颜色一样，它也在程序的控制之下。总结这两个函数的效果如下。

```
moveTo (x, y);      //设定 CP 到(x,y)
lineTo (x,y);       //在 CP 和(x,y)之间画一条线，然后将 CP 更新为(x,y)
```

于是可以调用 moveTo(x_1, y_1)和 lineTo (x_2,y_2)在(x_1, y_1)和(x_2, y_2)之间画一条直线。使用下面的代码，可以很容易地绘制出基于点列表(x_0, y_0)，(x_1, y_1)，…，(x_n, y_n)的折线。

```
moveTo(x[0],y[0]);
for(int i=1; i<n; i++)
   lineTo(x[i],y[i]);
```

可以直接在 OpenGL 顶端建立 moveTo 和 lineTo 函数，为此必须定义并保持我们自己的 CP。对于整数坐标的情况，用图 2.31 所示的实现方法就可以办到。

```
GLintPoint CP;              //全局变量

//<<<<<<<<<<<<<< moveto >>>>>>>>>>>>>>
void moveTo(GLint x, GLint y)
{
    CP.x=x; CP.y=y;         //更新 CP
}
//<<<<<<<<<<<<<  lineTo >>>>>>>>>>>>>>>>>
void lineTo(GLint x, GLint y)
{
    glBegin(GL_LINES);      //画线
        glVertex2i(CP.x, CP.y);
        glVertex2i(x, y);
    glEnd();
    glFlush();
    CP.x=x; CP.y=y;         //更新 CP
}
```

图 2.31 用 OpenGL 定义 moveTo()和 lineTo()

注意：变量 CP 是全局变量，要小心处理。

2.3.3 绘制边校正的矩形

多边形的一个特例是边校正的矩形。之所以叫这个名字,是因为它的边与坐标轴平行。我们可以创建函数来绘制一个边校正的矩形,不过 OpenGL 已提供了现成的函数 glRecti()。

```
glRecti(GLint x1,GLint y1,GLint x2,GLint y2);
//以(x1,y2)和(x2,y2)为对角点绘制矩形
```

该命令基于两个给定点绘制一个边校正矩形,并以当前颜色填充矩形。图 2.32 展示了用下面所示代码绘制的图形。

```
glClearColor(1.0,1.0,1.0,0.0);          //设置白色背景
glClear(GL_COLOR_BUFFER_BIT);           //清除窗口
glColor3f(0.6,0.6,0.6);                 //浅灰
glRecti (20,20,100,70) ;
glColor3f(0.2,0.2,0.2);                 //深灰
glRecti(70,70,150,130);
glFlush();
```

注意第二个矩形画在了第一个的上面。

图 2.33 展示了两个更加复杂的例子。图 2.33(a)是随机生成的矩形雪花,图 2.33(b)是人们很熟悉的具有不同灰度级的棋盘。如何生成这样的图形,留给读者作为练习。

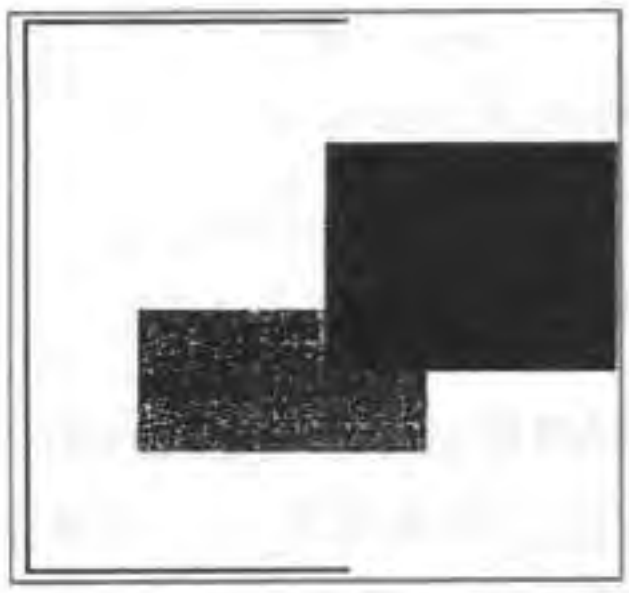


图 2.32 两个颜色填充的边校正矩形

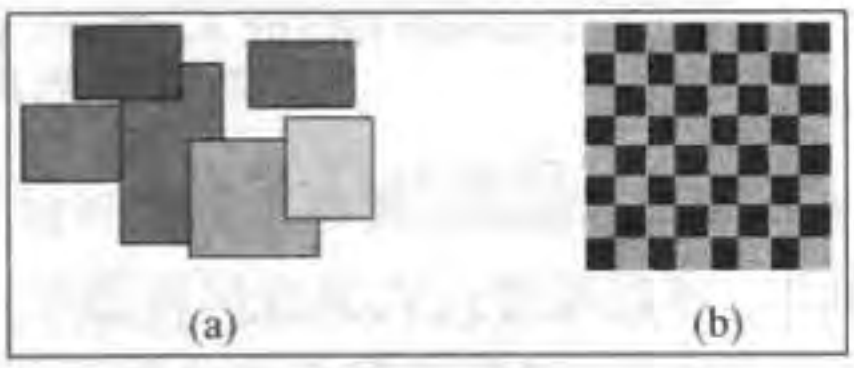


图 2.33

(a) 随机的矩形雪花;(b) 棋盘

2.3.4 边校正矩形的长宽比

边校正矩形的主要特性是它的尺寸、位置、颜色以及形状。它的形状可以用长宽比来描述,矩形的长宽比定义为宽度与高度的比值[①]:

$$长宽比 = \frac{宽}{高} \tag{2.3}$$

不同长宽比的矩形如图 2.34 所示。

① 注:一些作者将它定义为高/宽。

矩形 A 是一个 8.5 英寸×11 英寸的纸张形状，即所谓的山水画定位（也就是说，矩形的长大于高）。矩形 A 的长宽比是 1.294。矩形 B 具有电视屏幕的长宽比，即比例为 4/3。而 C 是著名的黄金分割矩形（将在案例 2.3 中具体描述），它的长宽比接近 $\phi = 1.618034$。长方形 D 是一个长宽比为 1 的正方形。E 是一个标准肖像纸张的形状，高宽比是 0.7727。最后，F 是一个又高又细的长方形，高宽比为 $1/\phi$。注意，矩形只有在边校正的情形下，谈论长宽比才有意义。当长宽比为 r 的边校正矩形旋转，使其一边与一相邻边平行时，长宽比变为 1/r。

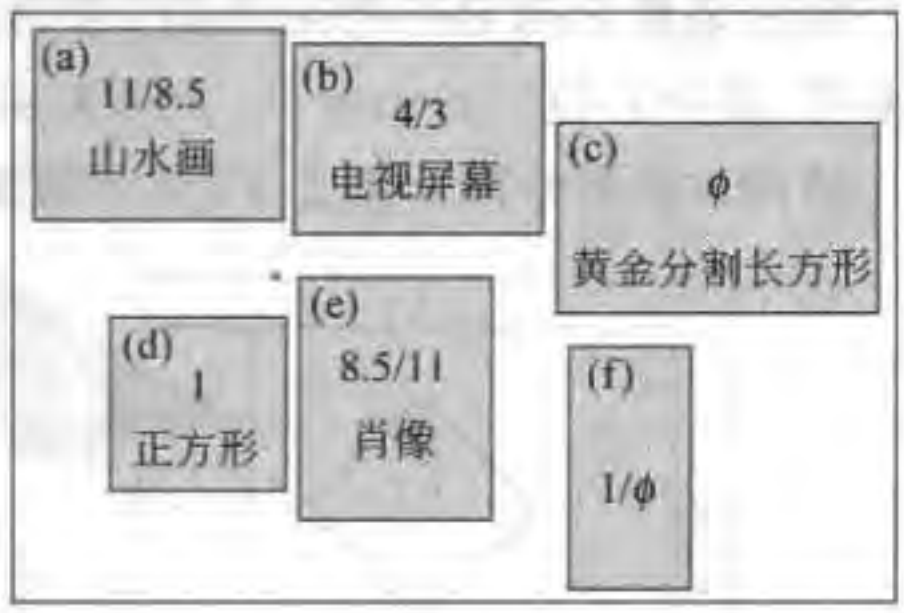


图 2.34 边校正矩形长宽比的例子

练习

练习 2.3.1 画棋盘

（在看答案之前，请先动手试试）写出例程 checkerboard (int size)，用来绘制图 2.33(b) 中所示的棋盘。将该棋盘的左下角置于(0, 0)处。这 64 个正方形的边长都设为 size 个像素。为这些正方形选择两种好看的颜色。

解：第 ij 个正方形的左下角位于(i * size, j * size) 处($i=0,\cdots,7$; $j=0,\cdots 7$)。使用下面的代码，可以将颜色在(r_1, g_1, b_1)和(r_2, g_2, b_2)间变换。

```
if((i+j)%2==0)      //如果 i+j 是偶数
    glColor3f(r1, g1, b1);
else
    glColor3f(r2, g2, b2);
```

练习 2.3.2 指定矩形的其他方法

除了指定两个相对的角点外，还有其他描述边校正矩形的方法。其中两种可能的方法是：

(1) 它的中心、高和宽；

(2) 它的左上角、宽以及长宽比。

写出函数 drawRectangleCenter()和 drawRectangleCornersize()传送这些指定的参数。

练习 2.3.3 不同的长宽比

写一小程序，用它绘制长宽比为 R 的矩形，R 由用户指定。假定显示窗口的绘图空间为 400×400。想办法使这个矩形尽可能的大。也就是说，如果 $R>1$，该矩形就会在横向上撑满绘图空间；如果 $R<1$，它就会在纵向上撑满绘图空间。

练习 2.3.4 绘制参数化的房子

细化图 2.27 所示的函数 parameterizedHouse()，以便在给定的高度和宽度下，按照适当的比例绘制门、窗户和烟囱。

练习 2.3.5 使用参数缩放和平移图形

写出函数 void drawDiamond (GLintPoint center, int size),以绘制图 2.35 中所示的简单菱形,其中心位于 center 处,大小为 size。

使用该函数绘制出图 2.36 所示的菱型雪花。

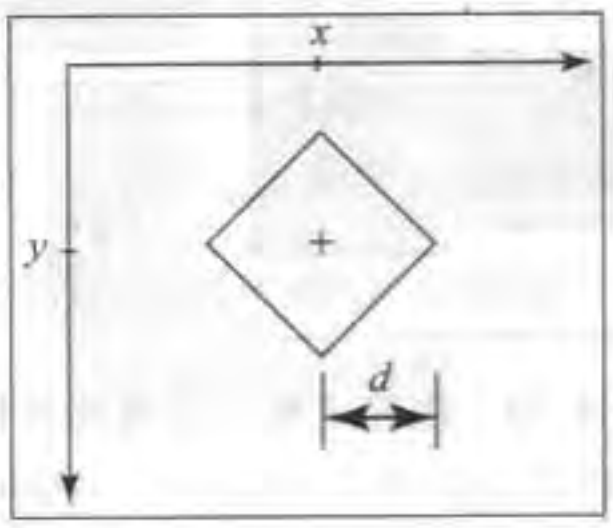


图 2.35 一个简单菱形

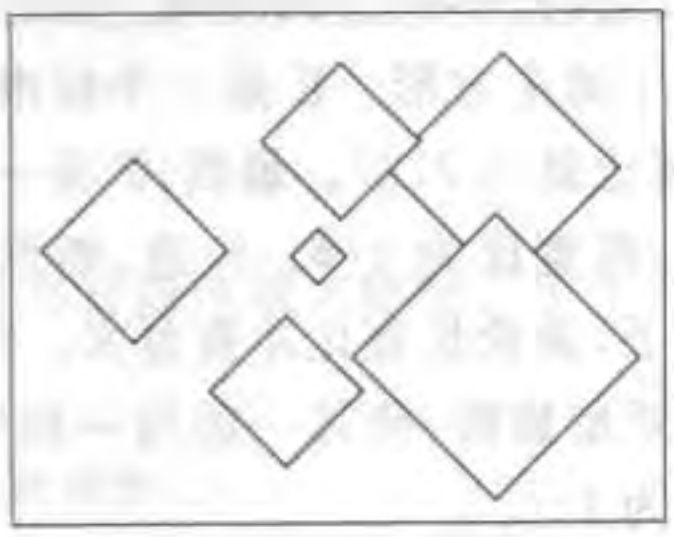


图 2.36 一些菱形

2.3.5 填充多边形

到目前为止,我们已经能够用 OpenGL 绘制非填充多边形了,也可以绘制单色填充的边校正矩形。OpenGL 同样支持用图案或彩色填充更一般的多边形,前提是这些多边形必须是凸的。

凸多边形:如果多边形中任意两点的连线完全位于多边形内部,那么这个多边形是凸的。

图 2.37 显示了几个多边形,其中只有 D、E 和 F 是凸的(按凸多边形的定义检查一下)。D 肯定是凸的,所有的三角形都是凸的。A 甚至算不上是简单图,所以它不可能是凸的。B 和 C 都在某一点"向内弯曲"(可以在 B 上找出两个点,它们之间的连线不完全位于 B 内部)。

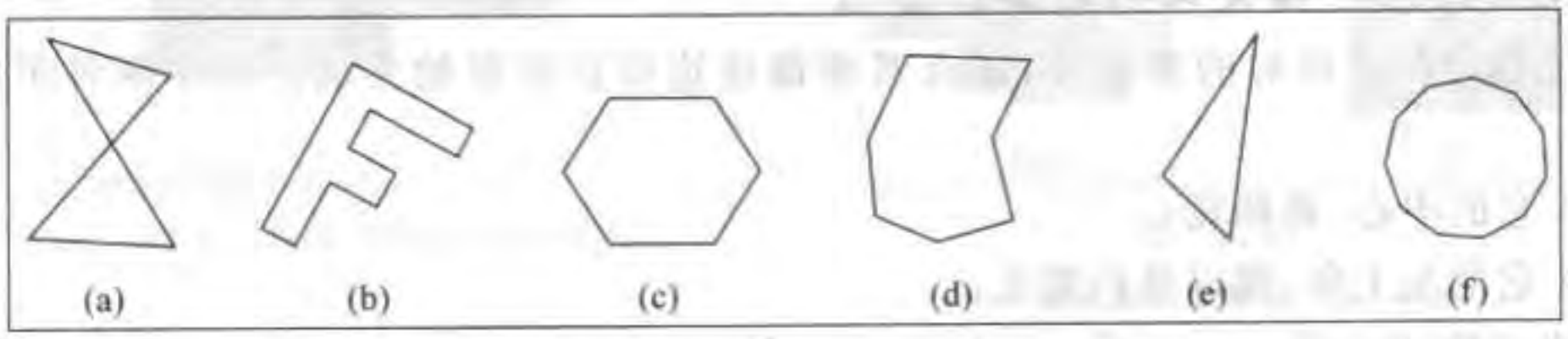


图 2.37 凸多边形和非凸多边形

要基于顶点绘制一个凸多边形,可以使用顶点列表(x_0, y_0),(x_1, y_1),…,(x_n, y_n),但是要把它们放在 glBegin(GL_POLYGON)和 glEnd()之间:

```
glBegin(GL_POLYGON);
glVertex2f(x0, y0);
glVertex2f(x1, y1);
      ...
glVertex2f(xn, yn);
glEnd();
```

该多边形以当前的颜色进行填充,也可以用点划线图案来填充(参见案例 2.5)。以后

我们将会把一些图像贴在多边形上，作为应用纹理的一部分。

图 2.38 显示了一些填充的凸多边形。第 9 章将考查一个用于填充任意多边形（无论凹凸）的算法。

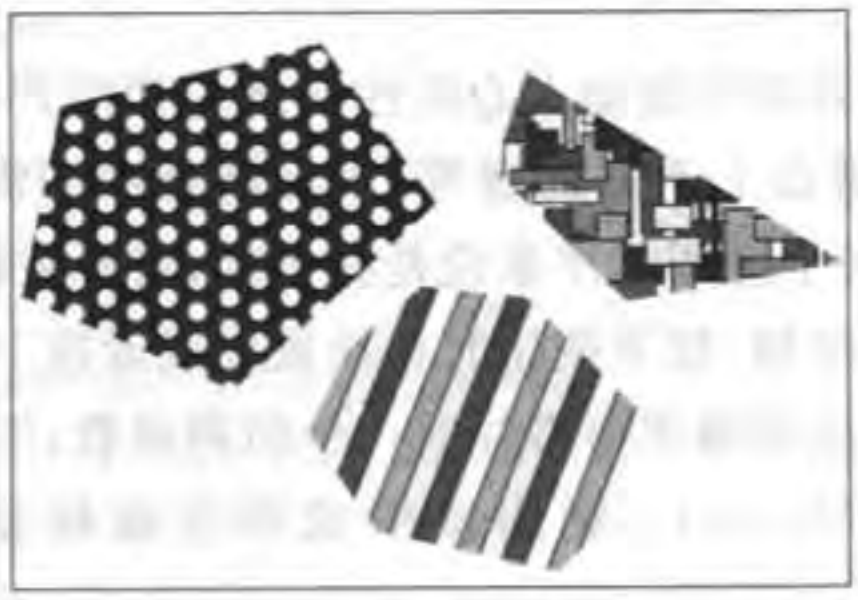


图 2.38 几个填充的凸多边形

2.3.6 OpenGL 中的其他图形元素

除了点、线和多边形，OpenGL 还支持绘制五种其他元素，图 2.39 显示了这些元素的例子。为了绘制其中某个元素，需要在使用 glBegin()时指定下列常数。

- GL TRIANGLES：一次使用 3 个顶点，每次绘制一个独立的三角形。
- GL_QUADS：一次使用 4 个顶点，每次绘制一个独立的四边形。
- GL_TRIANGLE_STRIP：基于 3 个一组的顶点 v_0，v_1，v_2；然后是 v_1，v_2，v_3；接着是 v_2，v_3，v_4。依此递推，绘制一系列三角形（按同样的顺序，即逆时针方向，遍历所有的三角形）。
- GL_TRIANGLE_FAN：基于 3 个一组的顶点 v_0，v_1，v_2；然后是 v_0，v_2，v_3；接着是 v_0，v_3，v_4。依此递推，绘制一系列与 v_0 相连的三角形。
- GL_QUAD_STRIP：基于 4 个一组的顶点 v_0，v_1，v_3，v_2；然后是 v_2，v_3，v_5，v_4；接着是 v_4，v_5，v_7，v_6。依此递推，绘制一系列四边形（按同样的顺序，遍历所有的四边形，如逆时针方向）。

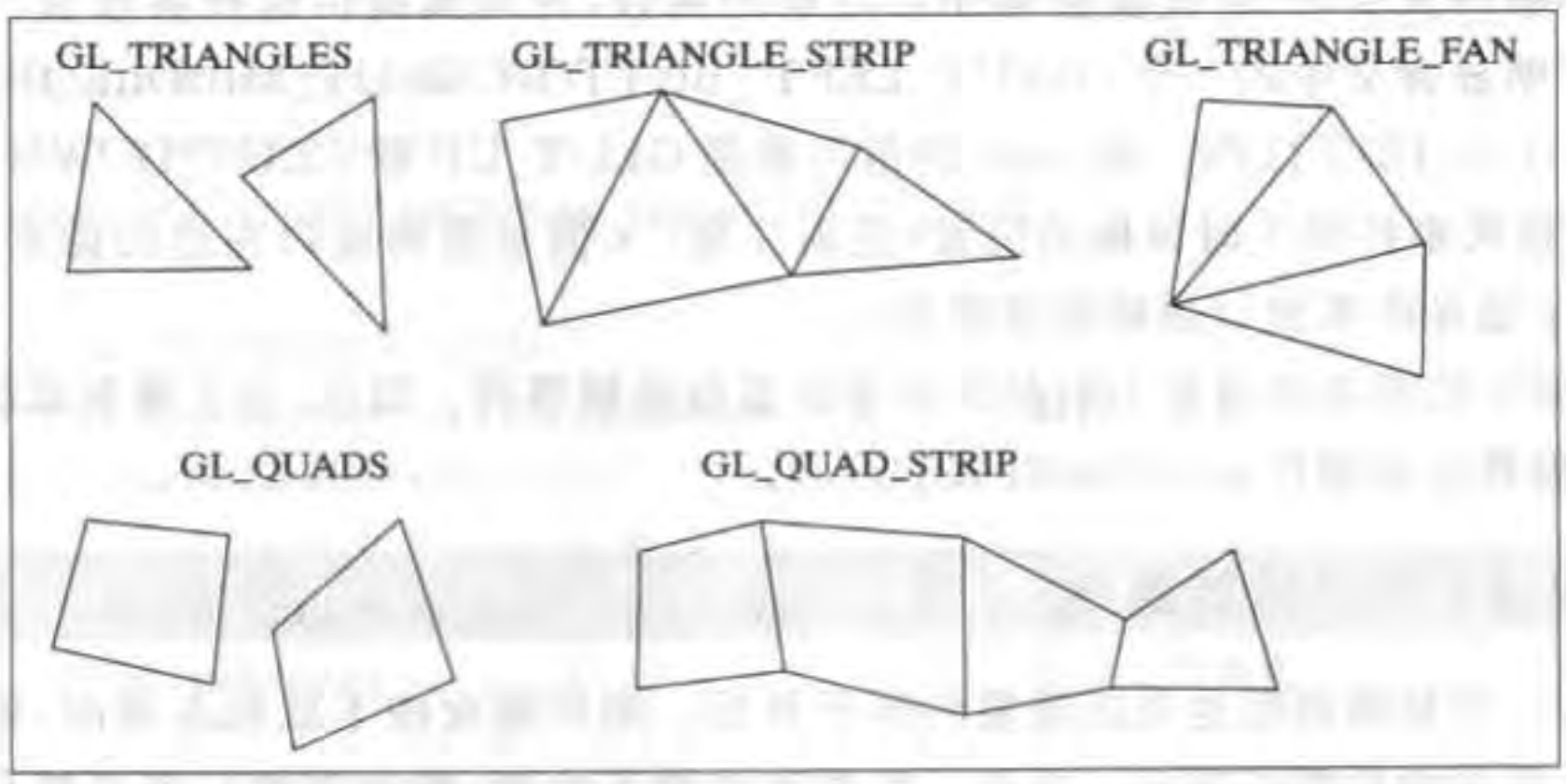


图 2.39 其他几何图形元素

2.4 与鼠标和键盘的交互

交互式应用程序富有成效的和激动人心的性质之一是用户能用人类自然的动作(如用鼠标指示和单击鼠标、按下键盘上不同的键等)来控制程序的流程。单击鼠标时鼠标的位置、按下的键盘上键的特性对于应用程序来说是有用的,需要被恰当的处理。

当用户按下或释放鼠标按钮、按下按钮时移动鼠标或者按下和松开键盘按键时,就会产生一个相关事件。程序员可以用每类事件注册一个回调函数,例如使用如下函数:

- glutMouseFunc(myMouse):利用按下或释放鼠标按钮时发生的事件来注册 myMouse()。
- glutMotionFunc(myMovedMouse):利用按下按钮同时移动鼠标的事件来注册 myMovedMouse()。
- glutKeyboardFunc (myKeyboard):利用按下和松开键盘按键的事件来注册 myKeyBoard()。

人们希望计算机对于人类自然的输入,例如说话和手势,能有更多的反应。对于这点,今天的计算机还有更长的路要走。通过移动鼠标来控制计算机,虽然只是一个笨拙的辅助方式,但与那些没有多少人懂的、晦涩神秘的输入符号序列相比,已经自然许多了。在本节中我们将学习在程序中如何使用鼠标和键盘。

2.4.1 用鼠标交互

与鼠标有关的数据怎样发送给应用程序呢?这是在设计将鼠标注册到 glutMouseFunc 的回调函数时必须回答的问题。回调函数的名字可以任意取(例如用 myMouse),但是,它一定要带 4 个 int 参数,其原型如下:

```
void myMouse(int button, int state, int x, int y);
```

当鼠标事件发生时,系统就会调用已注册的函数,并向其提供这些参数值。Button 值可能是下面明显含义中的一个:GLUT_LEFT _BUTTON、GLUT_MIDDLE_BUTTON 和 GLUT_RIGHT_BUTTON。而 state 的值可能是 GLUT_UP 和 GLUT_DOWN 中的一个。x 和 y 的值指明事件发生时鼠标的位置(但需注意:x 值是距离窗口左边的像素数,这在意料之中,但 y 值是距离窗口顶端的像素数)。

记住,事件处理器本身并不引起到屏幕的重新绘制事件。因此,为了看到鼠标事件的效果,鼠标处理器应该调用 glutPostRedisplay()。

例 2.4.1 用鼠标放置点

我们从一个初级的但也是很重要的例子开始。用户每次按下鼠标左键时,就会在屏幕窗口上鼠标所在的位置绘制出一个点。如果用户按下右键,就改变窗口背景颜色。下面的 myMouse()版本就可以完成这项工作。由于鼠标位置的 y 值是距离屏幕窗口顶端的像素

数，所以不应该在(x, y)上绘制该点，而在(x, screenHeight-y)上绘制，其中 screenHeight 是假设的窗口高度，以像素来计量。

```
void myMouse(int button,int state,  int x,  int y) {
    if(state==GLUT_DOWN) {
        if(button==GLUT_LEFT_BUTTON) {
           drawDot(x. screenHeight -y);
           glFlush();
        }
        else if (button==GLUT_RIGHT_BUTTON) {
           glClearColor(1.0f,0.0f,0.0f,0.0f);     //红色
           glClear(GL_COLOR_BUFFER_BIT);
           glFlush();
        }
    }
    return;
}
```

例 2.4.2 使用鼠标指定一个矩形

这里，我们想让用户画尺寸大小由鼠标输入的长方形。用户在两个点上单击鼠标，指定边校正矩形的两个角点，就可画出该矩形。每个边校正矩形的数据不需要保留：新的矩形会取代前一个矩形，用户右击鼠标即可清屏。

图 2.40 所示的例程在静态(static)数组 corner[]中存储一对角点，该数组设置成静态，所以在两次调用例程之间数组仍会被保存。变量 numCorners 跟踪已添加的角点数，当这个数达到 2 时，就会绘制出矩形，然后将 numCorners 重置为 0。

例 2.4.3 用鼠标控制 Sierpinski 垫片

可以很容易地扩充前述的 Sierpinski 垫片例程，使用户可以用鼠标指定初始三角形的 3 个顶点。可以使用与前例相同的办法：将 3 个顶点一起置于数组 comers[]中，当这 3 个点可用时，就绘出了 Sierpinski 垫片。具体的 myMouse()例程内容如下：

```
static GLintPoint corners[3];
static int numCorners=0;
if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
{
    corner [numCorners].x=x;
    corner [numCorners].y=screenHeight -y;     //翻转 y 坐标
    if(++nurnCorners==3)
    {
        Sierpinski(corners);                   //绘制垫片
        numCorners=0;                          //返回 0个角点
    }
}
glFlush();
```

```
void myMouse(int button, int state, int x, int y)
{
    static GLintPoint corner[2];                    //创建一个数组
    static int numCorners=0;                        //初始值为 0
    if(state==GLUT_DOWN)  {
       if(button==GLUT_LEFT_BUTTON) {
          corner[numCorners].x=x;
          corner[numCorners].y=screenHeight-y; //翻转 y 坐标
          if(++numCorners==2)  {
            glRecti(corner[0].x, corner[0].y, corner[1].x, corner[1].y);
            numCorners=0;                           //回到 0 个角点
          }
       }
       else if(button==GLUT_RIGHT_BUTTON)  {
          glClear(GL_COLOR_BUFFER_BIT);
          glFlush();
       }
    }
```

图 2.40 画鼠标控制大小的矩形回调函数

这里,Sierpinski ()与图 2.13 一样,不同之处是该三角形的 3 个顶点是被当作参数传送的。

鼠标运动

鼠标移动时产生另一个鼠标事件。回调函数,记为 myMovedMouse(),通过使用下列函数之一注册该事件:

```
glutMotionFunc(myMovedMouse);
glutPassiveMotionFunc(myMovedMouse);
```

第一个函数在按下一个或几个鼠标按钮(但不松开)鼠标在窗口内移动时调用。第二个函数是在没有按下鼠标按钮的情况下,鼠标在窗口内移动时调用。该回调函数必须有两个参数,且须为原型 void myMovedMouse (int x, int y)。x 和 y 值都是事件发生时鼠标所在的位置。

使用 glutPassiveMotionFunc()的一个例子是绘制并显示橡皮矩形:随着用户移动鼠标,矩形将相应地变大或变小。在下面的程序中,用户单击鼠标建立矩形的一个角点,然后在不按下鼠标按钮的情况下任意移动鼠标。用这种方式移动鼠标将产生一个事件,它调用 myPassiveMotion(int x, int y),并且,当前鼠标的位置决定了橡皮矩形的第二个角点。

这里,为了使动画显示更加平滑,我们设置 GLUT 使用双缓冲渲染模式(这种机制将在第 3 章中详细介绍)。尽管在前面的例子中,有一些绘制发生在 myDisplay 函数之外,这些绘制还是应该尽量避免的。因此,我们设置了几个全局变量,每次调用 glutPostRedisplay()渲染矩形时,就相当于强制 GLUT 调用我们自己的绘制函数 myDisplay()。

```
/* 允许用户画橡皮矩形程序：长方形大小随鼠标变化 */
//包括所有合适的 include 文件
struct GLintPoint {
    GLint x, y;
};
//全局变量
GLintPoint    corner[2];
bool    selected=false;
int    screenWidth=640, screenHeight=480;
void myDisplay() {
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glColor3f(1.0f, 1.0f, 1.0f);
    if(selected) {
        glBegin(GL_QUADS);
        glVertex2i(corner[0].x, corner[0].y);    //画一个长方形
        glVertex2i(corner[0].x, corner[1].y);
        glVertex2i(corner[1].x, corner[1].y);
        glVertex2i(corner[1].x, corner[0].y);
        glEnd();
    }
    glutSwapBuffers();
}
void myMouse(int button] int state, int x, int y) {
    if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN) {
        corner[0].x=x;
        corner[0].y=screenHeight-y;
        selected=true;
    }
    glutPostRedisplay();
}
void myPassiveMotion(int x, int y) {
    corner[1].x=x;
    corner[1].y=screenHeight -y;
    glutPostRedisplay();
}
int main(int argc, char * argv) {
    glutInit(&argc, argv);
    //初始化窗口
    glutInitWindowSize(screenWidth, screenHeight);
    glutInitWindowPosition(0, 0);
```

图 2.41 画橡皮矩形的完整程序

```
    glutInitDisplayMode(GLUT_RGB I GLUT_DOUBLE);
    //创建窗口
    glutCreateWindow("Rubber Rect Demo");
    //设置投影矩阵
    glMatrix 问 ode( GL_PROJECTION );
    glLoadIdentity();
    gluOrtho2D(0, screenWidth, 0, screenHeight);
    glMatrixMode(GL_MODELVIEW);
    //清渲染面
    glClearColor(0.0f ,0.0f, 0.0f, 0.0f);     //背景为黑色
    glViewport(0 , 0, screenWidth, screenHeight);
    glutMouseFunc(myMouse);
    glutDisplayFunc(myDisplay);
    glutPassiveMotionFunc(myPassiveMotion);
    glutMainLoop();
    retun(0);
}
```

图 2.41(续)

2.4.2 键盘交互

按下键盘上的某个键就会产生一个键盘事件,并放入消息队列中。回调函数 myKeyboard()就是通过 glutKeyboardFunc(myKeyboard)来注册这种事件。该函数必须有如下的原型:

```
void myKeyboard(unsigned int key, int x, int y);
```

按键的值就是所按键的 ASCII 值[①]。数值 x 和 y 指示事件发生时鼠标所在的位置(如前所述,y 值是距离窗口顶端的像素数)。

程序员可以利用键盘上的按键为用户提供许多快捷键,它们可在程序中的任意位置被激活。大多数 myKeyboard ()的实现方法都由很长的 switch 语句组成,一个 case 对应一个感兴趣的键。注意:需要记住,switch 语句会给程序带来很大的危险,一定不要忘记用 break 结束每个语句。当然,你有可能忘记。在编程历史上,这曾导致过巨大的灾难,包括 AT&T 在 1990 年 1 月长距离电话网络事故中发生的大错。如果想更多地了解这个故事,可以查找以下这个资料:《Expert C programming: Deep C Secret》(Peter van der Linden 著)【van der Linden 94】。图 2.42 显示了这样一种可能的方案:按下 p 键就会在鼠标位置绘制一个点,按下 E 键就会退出程序。注意,如果用户一直按着 p 键并移动鼠标,就会产生一个快速点序列,这就模拟了"徒手"绘图。

---

① ASCII(American Standard Code for Information Interchange)代表美国标准信息交换码,ASCII 表可从因特网得到。

```
void myKeyboard(unsigned char theKey, int mouseX, int mouseY)
{
  GLint x=mouseX;
  GLint y=screenHeight-mouseY;
  switch(theKey)
  {
    case 'P'
            drawDot(x, y);        //在鼠标位置画点
            break;
        case 'E':
            exit(-1);             //终止程序
          default:
            break;                //什么也不做
    }
}
```

图 2.42 一个键盘回调函数的例子

2.5 程序中的菜单设计与使用

如果一个图形程序能够提供菜单功能，对用户来说，使用就方便多了。正如在第 1 章简要说明的一样，程序运行期间，会在适当的时刻出现一个菜单。菜单出现在程序已经生成的图形前面，给用户提供许多选项。像我们所看到的一样，选项有不同的类型。然后程序会等待用户做出一个或多个选择。当用户做出选择后（通常是用鼠标或键盘），菜单消失，重新显示刚才的图形（菜单出现前的图形），接着，程序按用户选择的路线运行（设计合适的菜单对用户来说方便易用，无需太多的解释）。

GLUT 和 GLUI 菜单

这里通过介绍两种不同的 OpenGL 库，GLUT 和 GLUI 库，来学习两个级别的菜单管理。虽然 GLUT 菜单（回顾一下图 2.3）比 GLUI 菜单简单，但仍然很有用。下面首先介绍一个较好使用 GLUT 菜单的应用程序。

实例

我们要构建一个的菜单，允许用户在程序运行中，改变一个正在旋转的三角形的颜色。用户在 OpenGL 窗口里右击鼠标激活菜单，如图 2.43 所示，菜单提供了 4 种颜色供选择。

用户在选项位置左击鼠标来选择需要的颜色。当然除此之外，还有许多其他的控制方式，例如，键盘的按键控制。

设计代码和使用 GLUT 菜单

即使设计十分简单的菜单，对程序员来说，在程序中编写代码来指定菜单的形状、位置、

条目和动作似乎仍是一个艰难的任务。从编程角度来看,需要考虑将菜单画成一个正方形,用合适的方式定位和标记菜单的选项,允许点击鼠标右键激活菜单(使其可视)和用鼠标选择其中一个选项。选择完后,菜单消失。对程序员来说,这似乎是一个使人畏缩的任务,但是,事实上,GLUT 为我们全都处理了:只需调用几个 GLUT 函数,就很容易完成所有这一切。

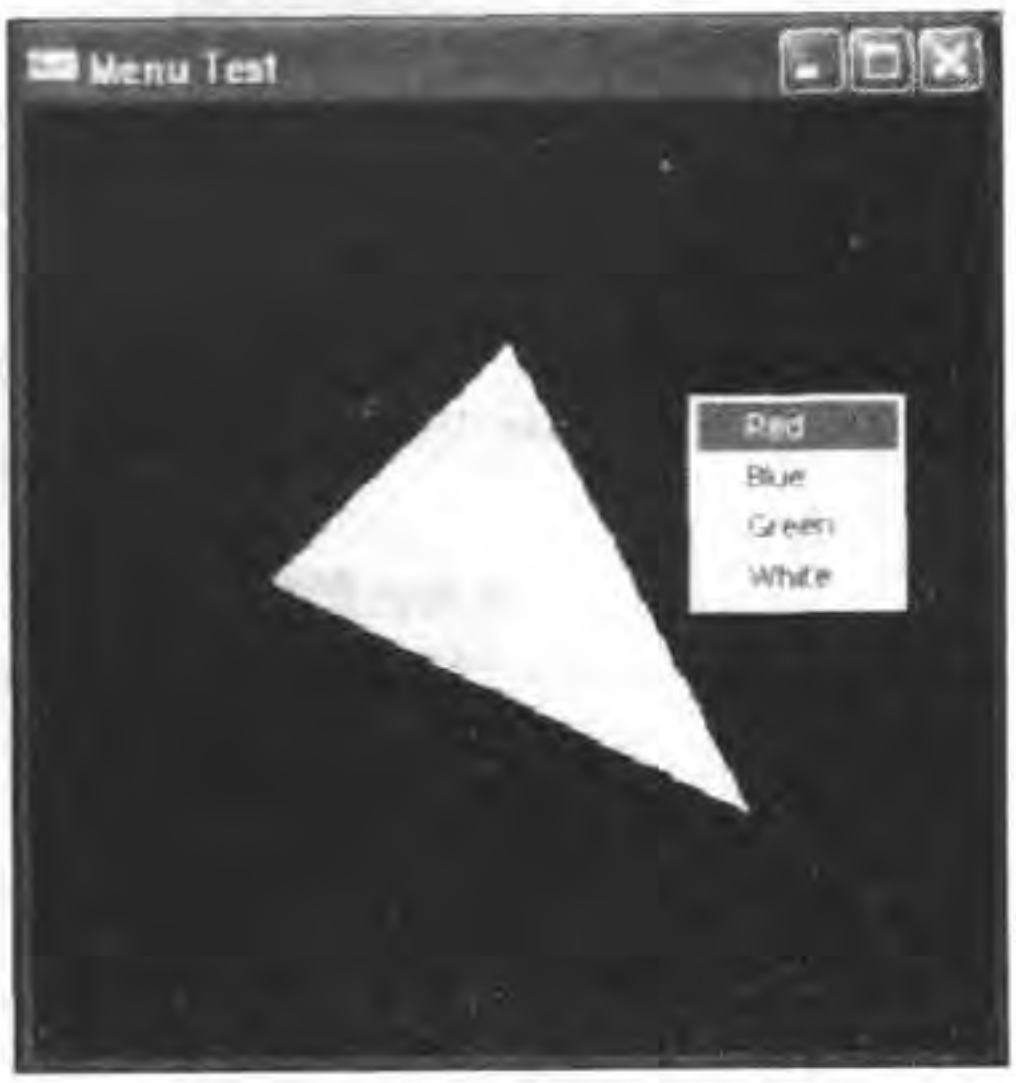


图 2.43 一个激活的 GLUT 菜单

首先,必须创建菜单,使用 void glutCreateMenu()初始化菜单句柄。除了 glutCreateMenu(),还需要定义实际的菜单条目。我们使用带有一个参数 ProcessMenuEvents 的函数 glutCreateMenu(ProcessMenuEvents)来定义。我们必须编写 ProcessMenuEvents 函数,处理根据菜单选项来执行的动作。ProcessMenuEvents 是一个典型的 switch 语句,它根据选择的菜单条目,在应用程序中设置一些值。

使用函数 glutAddMenuEntry(),可在菜单中添加选项,这是设计菜单本身必要的一步。对于上述的这个例子,我们添加如下的条目(为方便起见,可使用枚举数据类型或 defines 功能)

```
glutAddMenuEntry("Red", RED);
glutAddMenuEntry("Blue", BLUE);
glutAddMenuEntry("Green", GREEN);
glutAddMenuEntry("White", WHITE);
```

程序员可自由地添加菜单条目,不需要预先告诉 GLUT 有多少条目,当用户单击鼠标做出选择时,每个菜单条目都能显示一串字符,并产生一个与此相关的值。一旦菜单设计好,可以用 glutAttachMenu(GLUT_RIGHT_BOTTOM)建立与鼠标按钮的联系(可用的鼠标按钮有左、中或右)。这就是 GLUT 菜单处理的所有要求。注意,不需要单独的鼠标句柄。图 2.44 提供了一个完整的含有 GLUT 菜单的工作程序。

```
#include <gl/glut.h>
#include <gl/glu.h>
#include <gl/gl.h>
#define RED 1
#define GREEN 2
#define BLUE 3
#define WHITE 4
```

图 2.44 一个演示使用 GLUT 菜单的完整程序

```
float angle=0.0;                          //三角形旋转角
float red=1.0,blue=1.0,green=1.0;         //可能的三角形颜色
void renderScene(void) {
    //画三角形的回调函数
    glClear(GL_COLOR_BUFFER-BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glRotatef(angle,0.0,1.0,0.0);         //一点点旋转三角形
    glColor3f(red,green,blue);            //改变三角形的颜色
    glBegin(GL_TRANGLES);                 //画三角形
    glVertex3f(-0.5,-0.5,0.0);
    glVertex3f(0.5, 0.0,0.0);
    glVertex3f(0.0,0.5,0.0);
    glEnd();
    angle++;
    glutSwapBuffers();
}

void processMenuEvents(int option) {
    //鼠标的选择选择颜色
    switch(option) {
        case RED: red=1.0; green=0.0; blue=0.0; break;
        case GREEN: red=0.0; green=1.0; blue=0.0; break;
        case BLUE: red=0.0; green=0.0; blue=1.0; break;
        case WHITE: red=1.0; green=1.0; blue=1.0; break;
    }
}
//-----------------MAIN-----------------------------
void main(int argc, char * argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(320,320);
    glutCreateWindow("Menu Test");   //打开一个 OpenGL 窗口
    glutDisplayFunc(renderScene);    //注册显示函数
    glutIdleFunc(renderScene);
    //调用函数创建菜单
    //右击鼠标激活函数
    //创建菜单并关联菜单事件
    glutCreateMenu(processMenuEvents);
    glutAddMenuEntry("Red",RED);
    glutAddMenuEntry("Blue",BLUE);
    glutAddMenuEntry("Green",GREEN);
    glutAddMenuEntry("White",WHITE);
    glutAttachMenu(GLUT_RIGHT_BUTTON);//系在右鼠标按钮上
    glutMainLoop();
}
```

图 2.44（续）

GLUI 菜单

GLUT 给用户提供了很好的、对程序流程的高级控制能力。但是,有时用户想对逻辑输入设备进行更好的控制和更广泛的设置(回顾第 1 章)。这里,我们学习 GLUI 库(GL User Interface,GL 用户接口),GLUI 提供不同种类的控制。图 2.45 显示了一个使用 GLUI 的相当复杂的菜单,以下描述每个条目时都参考这个图(下面的条目有些经常出现,有些则不太明显)。

- Buttons(按钮)——例如用于 OK 和 Cancel 的按钮,在图中 Button 的例子为"Button、Another Button 等。
- CheckBoxes(检查盒) ——on/off(开/关)选择,例如线框图与实体图之间切换观看,图中的例子为 CheckBox1。
- Radio Buttons(单选按钮) ——选择设备;在某个时刻,只能从单选按钮组中选择一个按钮。图中标记了 Radio Button 1、Radio Button 2、Radio Button 3 等。
- Static Text(静态文本)——显示一串短消息,帮助用户管理程序。用户不能改变静态文本,所以它不是一个输入设备。图中标记为 Static Text。
- Editable Text areas(可编辑文本区)——字符串设备;用户能输入字符串,空格,并且改变输入的字符。
- Spinners(微调控制项)——数值输入设备;当用户用鼠标单击上或下箭头时,数值快速地增加或减少。
- Panels(面板) ——一组接口元件,为了方便观看,能最大化、最小化。图中显示了一个标记为 Rollout(open)的例子,当按下"—"时,面板收缩到标记为 Rollout(closed)的单个对象。不要奇怪,单击"+"会展开被收缩的接口对象(例如 Hi There 和 123),并且允许它们中的每个都成为一个输入设备。
- List Boxes(列表盒) ——一组下拉式子菜单,提供一个可选的输入设备(图中标记为 ListBox 1 和 ListBox 2)。
- Rotation and Translation devices(旋转和平移设备) ——像弓形球的设备,提供数值的输入,用户用鼠标连续地移动弓形球的上部或箭头的头部,可以看到目标在平移或旋转。

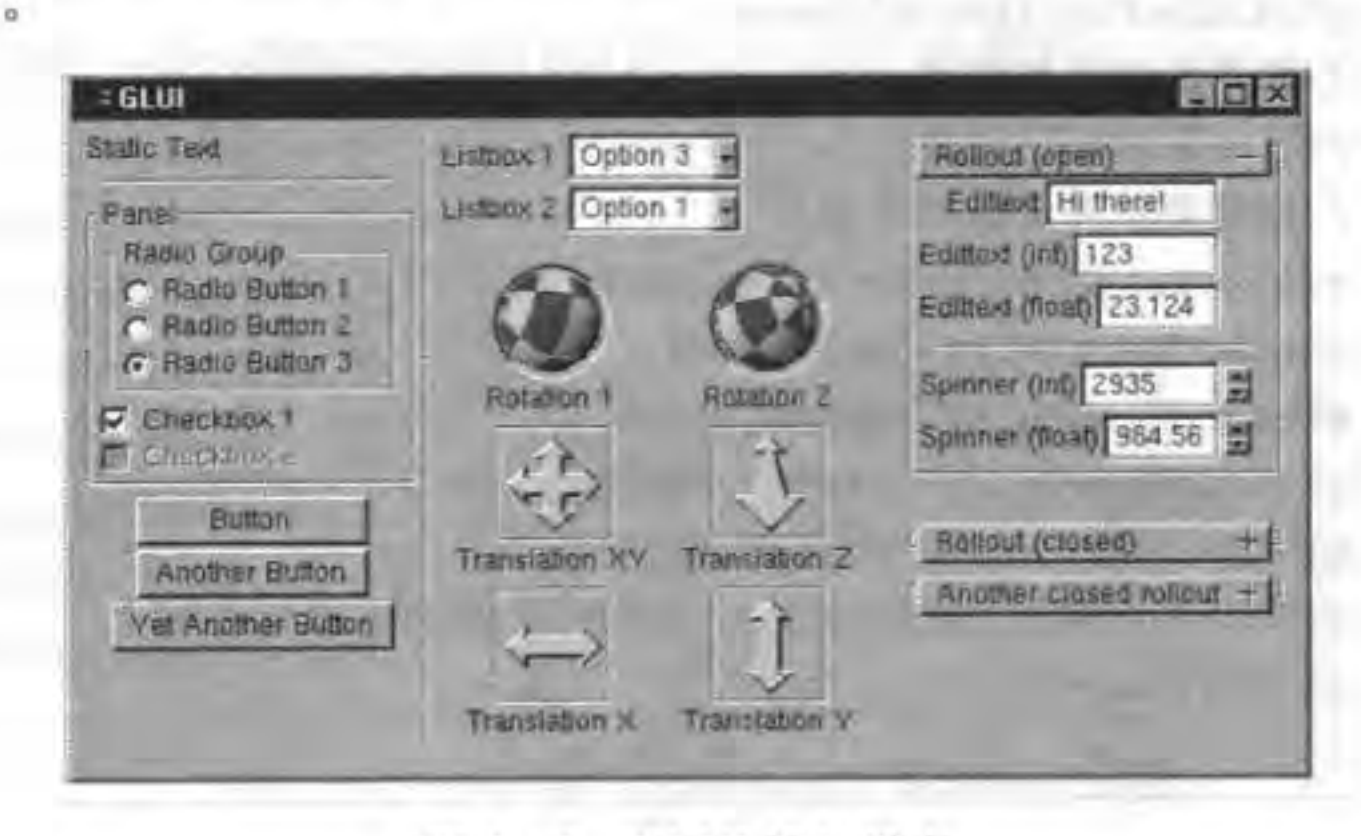
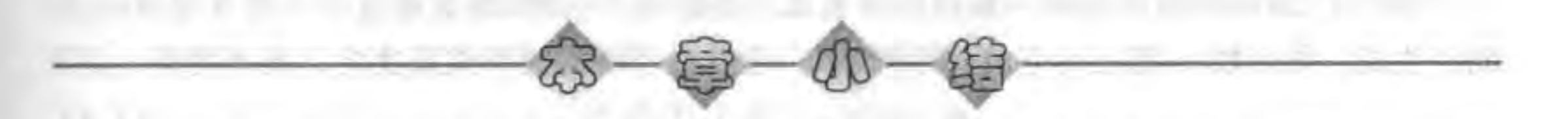


图 2.45 GLUI 接口样本

- Separators(隔离器)——程序的可视化助手(不是输入设备),能帮助管理屏幕上的许多选项和元件。例如,Static Text 下面的线就是一个隔离器。

注意,GLUI 菜单比 GLUT 更加复杂,我们能想象得到,定义和管理 GLUI 菜单的代码也必然更加复杂。像 GLUT 一样,GLUI 本身处理了许多困难的任务。一个完整的、经 P. Rademacher[1] 授权的、演示 GLUI 控制能力的工作程序,可以从本书的网站上得到。



编写图形应用程序中最难的部分是开头:如何在程序中将硬件和软件结合在一起,生成最初的几个基本图形。OpenGL 应用程序接口对此提供了很大帮助,它提供了功能强大却简单易用的例程集来进行绘图。OpenGL 的最大优点之一是设备无关,这样在一种图形环境中写出来的程序,就能在不做修改的情况下应用到其他环境中。

现在所编写的大多数图形应用程序都是基于窗口环境。这种程序在屏幕上打开一个用户可以移动并调整大小的窗口,还可以对单击鼠标和敲击键盘做出反应。我们已经学习了怎样使用 OpenGL 函数便捷地创建这样的程序。

基本元素绘制例程用于绘制由点、线、折线和多边形组成的图形,并且和更加强大的例程结合在一起,构成个人图形工具箱的基础。本章列举了一些例子阐述这些工具的使用,并且介绍了在程序中使用键盘和鼠标进行交互的方法。我们还介绍了 moveTo()和 lineTo()方法以便于绘制直线,并特别强调了使用 OpenGL 函数 glRecti()画边校正矩形。

下面介绍的案例进一步提供了编程实例,帮助读者更深入地理解前面讨论过的主题,或者引申出有趣的相关主题。

案例分析

为了巩固所学知识,最好的方法是尝试使用这些介绍过的知识。在学习最初几章时更要如此,因为一开始学习计算机图形学,不了解程序的编写是必须克服的障碍。为了强调这一点,每章都以案例结尾,其中介绍的编程项目不仅本身有趣,而且浓缩了该章中所提到的知识。

有些案例只是简单的练习,只需要搞懂本章中给出的伪代码并按步骤运行程序就可以了。有些案例则很有挑战性,并且可以作为课程中大型编程项目的基础。很难判断某人完成某个项目需要多少时间,每一个案例的难度只是一个大致的粗略猜测。

难易程度

Ⅰ:简单的练习,可以在课上完成。

Ⅱ:较难任务。可能需要几天才能完成。

Ⅲ:复杂任务。可能需大约两周时间。

---

[1] Paul Rademacher 与 Nigel Stewart 做 GLUI 方面的工作,GLUI 项目的主页是 http://glui.sourceforge/net。

案例 2.1 伪随机点云

(难度：Ⅱ)每调用一次随机数生成器(RNG)rand()(该函数是标准C++ 库函数)，就产生一个 0～$N-1$ 之间的数。每个数值似乎都是随机的，与前一个数没有任何关系。

在一些随机概率起很大作用的游戏中，如纸牌和掷骰游戏，随机数有着广泛的应用。

事实上，rand()所产生的一系列数字根本不是随机产生的，而是通过一个很有规律的机制产生的：每个数 n_i 都是由它前一个数 n_{i-1}，通过一个特殊的等式来决定。典型的等式是：

$$n_i = (n_{i-1} * A + B)\%N \tag{2.4}$$

其中，A、B 和 N 都是适当选择的常数。对 n_{i-1} 的操作看上去像把 n_{i-1} 二进制表示中不同的位杂乱无章的拼凑在一起形成了新的 n_i 值。但实际上每个数都是由前一个数独一无二的确定。基于式(2.4)的一组有效的数字可以是 $A=1103515245$、$B=12345$、$N=32767$。将 n_{i-1} 乘 A 再加上 B 构成了一个很大的数值，模运算将这个数限制在 0～$N-1$ 范围内。这个过程从设置 n_0 的种子(seed)数开始。

因为这些数字只是表面上看是随机的，所以将它们称为伪随机数。选择 A、B 和 N 的数值很重要，这些值稍有差异，生成的数列就会出现极不相同的特征。更详细的介绍可以在【Knuth，Weiss98】中找到。

散乱点图

有些实验产生了一些由许多数字对(a_i，b_i)组成的数据，目的是可视地推理出 a 值和 b 值是怎样相关的。例如，对许多人进行测量，想了解他们的身高和体重之间是否有紧密联系。

散乱点图可以给出这些数据的直观视觉效果。每个人的(身高、体重)数据都可当作点绘制出来，所以只需使用工具 drawDot()。图 2.46 显示了一个例子。它表明：人的身高和体重之间大概是线性关系，尽管有些人特殊，比如 A 有点瘦。

这里用散乱点图来可视化地检测随机数生成器的性质。每调用一次函数 rand(N)，它就会返回一个 0，…，$N-1$ 之间的数，好像是随机挑选的，与先前 rand()产生的数毫无关系。但是这些数字真的不相干吗？

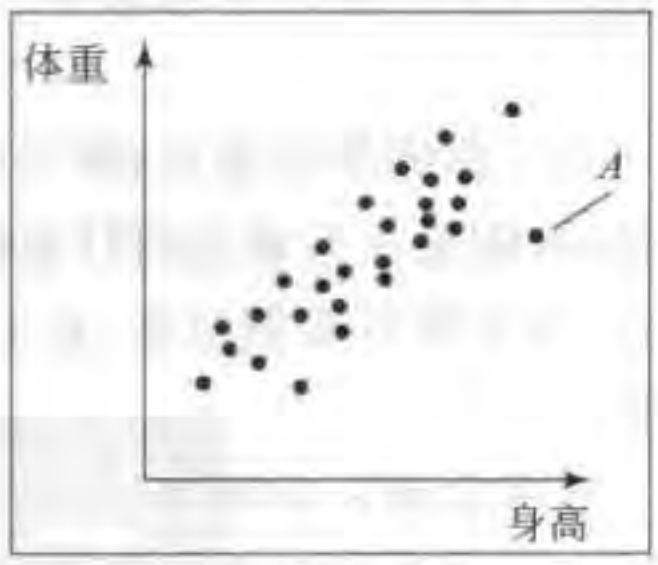


图 2.46 人的身高体重散点图

做个简单的试验，根据 rand()所返回的数构建散乱点图，下面这个例程依次调用 rand()两次，并且绘制第一个坐标值对第二个坐标值的图(每点由一对坐标值确定)。可以用 drawDot()来实现：

```
for(int i=0; i<num; i++)
    drawDot(rand(),rand());          //num 是用户选的很大的数
```

或者用基本的 OpenGL 命令实现，例如将 for 循环置于 glBegin()和 glEnd()之间：

```
glBegin(GL_POINTS);
    for(int i=0; i<num; i++)         //num 数操作
        glVertex2i(rand(),rand());
glEnd();
```

第二种方法可能更有效些，它省去了频繁调用 glBegin()和 glEnd()而带来的开销。

图 2.47 展示了一个可能得到的典型点图。在整个正方形中，点是均匀分布的，以使你确信 0～$N-1$ 之间的数的出现几率相同，而且某个数和它前一个数没有任何关系。

图 2.48 显示了低级随机数生成器产生的点图。在图 2.48(a)中，某些数太密，点不是均匀分布的。在图 2.48(b)中，x 值与 y 值之间有一些关系，在一个小范围内，对于大量的 x 值，y 值似乎集中在一条与 x 轴成 45°角的直线上。在图 2.48(c)中，显示的是最不可接受的情况：在产生了几十个数后，样式重复，不再产生新的数据点。

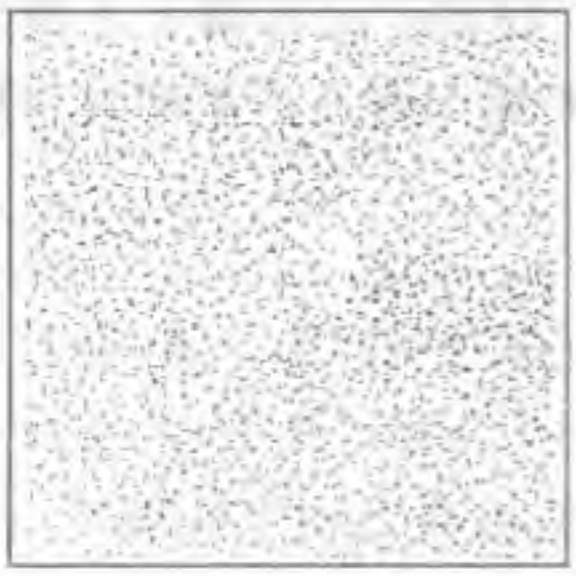


图 2.47　500 个随机点的点云

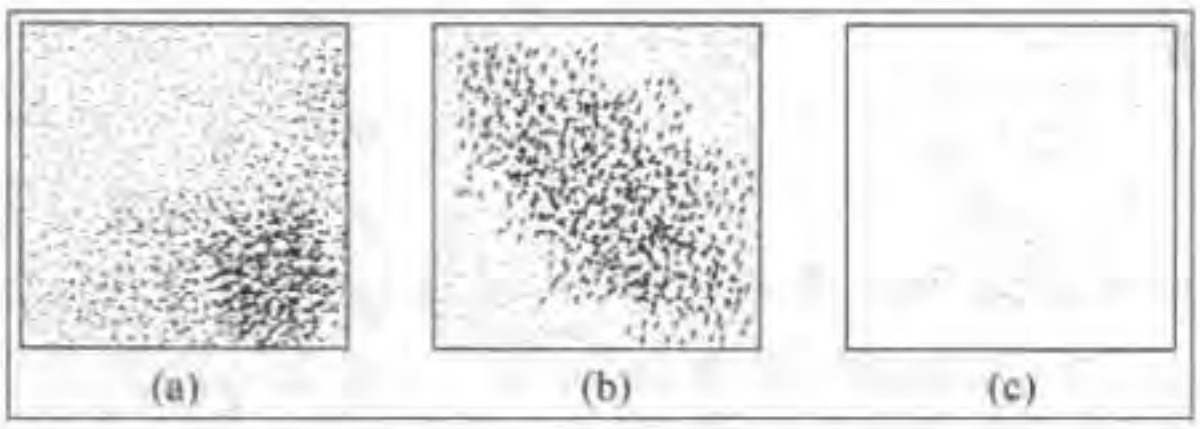


图 2.48　低级随机数生成器产生的点图

绘制这样的点图可以对所产生数字的均匀性做一个粗略的观察，但是这种检测并不是一种全面检测。

写出一个能够产生随机点阵图的程序，使用一些不同的随机数产生器生成(x, y)对。在基本随机数生成器中尝试取不同的常数 A、B 和 C，观察它对点丛产生的效果。

研究 RNG 内部工作机理的工作已有很长的历史，这里推荐你使用你喜欢的引擎搜索关键字随机数生成器或 random number generators，然后阅读检索到的材料。

案例 2.2　迭代函数系统介绍

(难度：Ⅱ)绘制 Sierpinski 垫片时的重复操作就是迭代函数系统(IFS)的一个例子，本书中会遇到很多迭代函数系统。许多计算机生成的有趣图形(如分形、Mandelbrot 集合等)都是基于这种 IFS 的变化产生的。

手工计算器提供了对简单 IFS 做实验的工具。添加某个数字 num(正数)然后按下平方根键，就生成了一个新数 $\sqrt{num}$。再一次按下平方根键，对新数再取平方根，就会生成新的数。一直按下平方根键，直到你满意为止。执行这个操作，实际上就是用平方根函数进行迭代，并且每个结果都用作下一个平方根的输入值。初始值 $num=64$ 会产生一系列数字：64、8、2.8284、1.68179，…(想想该序列数有收敛值吗?)。

图 2.49 显示了该系统的框架图。可以看出，每个输出值反馈给输入值以形成平方根，从而一次又一次地循环。

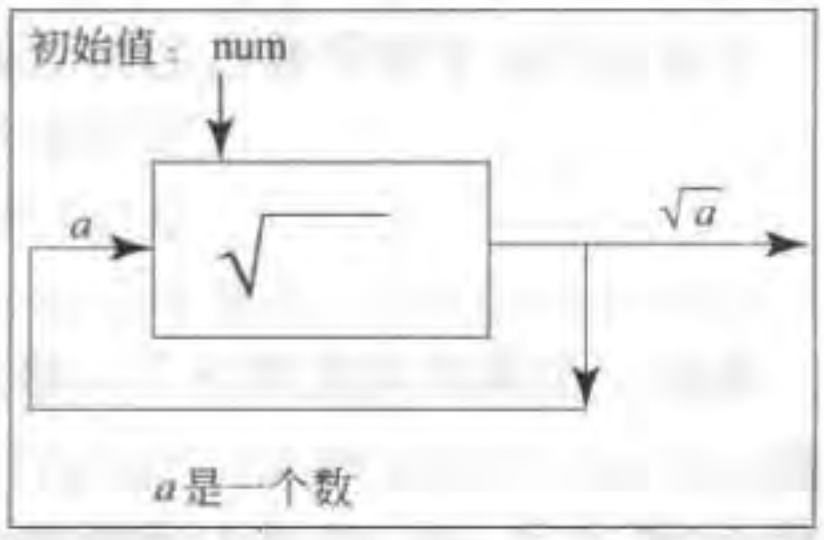


图 2.49　重复取平方根

从这个例子中，我们至少能提出一个不正式的定义：迭代系统是一个函数，或者是函数的集合，函数的输出值通过反馈，成为下一步迭代相应函数的输

入值。

在这个例子中,迭代的函数是 $f(x)=\sqrt{x}$,或者用符号表示为 $f(.)=\sqrt{.}$,即"平方根器"。还可以使用其他函数,例如下面的函数。

- $f(.)=2(.)$;"倍增器",倍增它的参数。
- $f(.)=\cos(.)$;"cos 函数"。
- $f(.)=4(.)(1-(.))$;"逻辑"函数,用于混沌(chaos)理论(参见第 3 章)。
- $f(.)=(.)^2+c$;常数 c,用来定义 Mandelbrat 集合(参见附录 4)。

有时给从 IFS 中出现的每个数字都命一个名字很有好处。我们可以称第 k 个数字为 d_k,并认为该过程开始于 $k=0$(通过将初始值 d_0 输入到系统中),则 IFS 所产生的数字序列就是:

$$
\begin{aligned}
&d_0\\
&d_1=f(d_0)\\
&d_2=f(f(d_0))\\
&d_3=f(f(f(d_0)))\\
&\cdots
\end{aligned}
$$

因此,d_3 是通过应用函数 $f(.)$ 3 次得到的。这可以叫做函数 $f(.)$对初始值 d_0 的第 3 迭代。若要更简明些,可以将 $j()$的第 k 次迭代表示为:

$$d_k=j^{[k]}(d_0) \tag{2.5}$$

意思为在 $j(.)$对 d_0 应用了 k 次之后所产生的数值(注意,这不是指数值 $f(d_0)$的 k 次幂)。还可以使用递归的形式,例如:

$d_k=f(d_{k-1})$,对于给定的 d_0 值,$k=1,2,3,\cdots$

这一系列的值可能是无穷的,数值 $d_0,d_1,d_2,d_3,d_4,\cdots$序列称为该系统的"$d_0$ 轨道"。

例子:对于函数 $f(.)=\sqrt{.}$来说,64 的轨道是 64、8、2.8284、1.68179、…。10000 的轨道是 100、10、3.162278、1.77828、…(那么 0 的轨道是什么呢? 0.1 的轨道又是什么呢?)。

例子:对于"倍增器"函数 $f(.)=2(.)$来说,7 的轨道是 7、14、28、56、112、…第 k 次迭代是 7×2^k。

例子:对于函数 $f(.)=\sin(.)$来说,1 的轨道可以用计算器给出,结果是 1,0.8414、0.7456、0.6784、…,非常缓慢地逼近数值 0(那么对于 cos(.)来说,1 轨道是什么? 值得注意的是,该轨道收敛到何值?)。

项目 1:预报冰雹序列

考虑迭代以下这个迷人的简单函数 f(.):

$$f(x)=\begin{cases}\dfrac{x}{2} & \text{如果 } x \text{ 是偶数}\\ 3x+1 & \text{如果 } x \text{ 是偶数}\end{cases} \tag{2.6}$$

变量 x 为偶数时就缩小了一倍,而为奇数时则被扩大了。例如,17 的轨道是下面这个序列:17、52、26、13、40、20、10、5、16、8、4、2、1、…如果中间某个数达到 2 的某个幂时,该序列就会"像冰雹一样"落到 1,并且陷入了一个短循环中。一个没有答案的数学问题如下:

未解答的问题:每一个轨道都落在 1 上吗? 也就是说,是否存在这样一个正数,当把它

作为起始点并用这个"冰雹函数"迭代，最终不是落在 1 上？没有人知道答案，但是该序列的错综复杂性已经被广泛地研究过【Hayes84】。

解决这个问题的任何人都可获得大笔现金奖励，外加名誉。

本书的网站上可以找到关于这个问题的一些资源。

写一个程序，绘制 $y_k = f^{[k]}(y_0)$ 对 k 所生成的数值序列。用户给出 0～4000000000 之间初始值 y_0（unsigned long 型变量可以有这样的大小）。每个 y_k 值都被画成点图（k，y_k）。继续绘制这些点，直至 y_k 到达数值 1 为止（如果它能够达到的话）。

因为冰雹序列可能很长，并且 y_k 值会变得很大，所以就需要在显示它们之前对它们进行缩放。回顾 2.2 节，可以确定合适的 A、B、C 和 D 值，使（k，y_k）绘制在以下窗口坐标：

$$sx = (A * k + B)$$
$$sy = (C * y_k + D) \tag{2.7}$$

这样，整个序列将被适当地放置在窗口中。

注意，你并不知道这个序列将有多长，或者 y_k 值有多大，只能在该序列产生后才能知道。一个简单的解决办法是首先只运行序列而不画图——跟踪最大值 yBiggest（与 y_k 关联），还要跟踪序列到达所需要的 1 时迭代的次数 kBiggest。随后利用这些数值确定 A、B、C 和 D 值。然后再次运行该序列，并将其绘制出来。

一个好奇的问题：对于初始值为 1～1 000 000 之间的任何冰雹序列，最大的 yBiggest 和最大的 kBiggest 是什么？

用产生点的函数进行迭代　通过一些函数 $f(.)$ 将数值进行迭代已经很有趣了，但通过一些函数来迭代坐标点则更加有趣，因为我们可以用 drawDot() 将产生的点绘制出图案来。所以，我们考虑函数 $f(p)$，初始点为 $p=(x, y)$，通过将函数产生的点又作为输入，每个新形成的点又反馈到同一个函数产生另一个新点，如图 2.50 所示。这里，p_{k-1} 被用来创建第 k 次迭代 $p_k = f^{[k]}(p_0)$，然后，又反馈产生 p_{k+1}，如此循环。

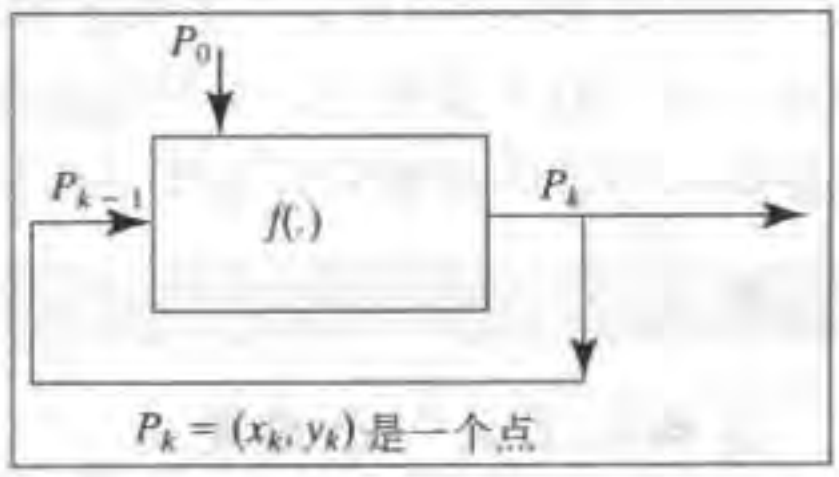


图 2.50　迭代函数点序列生成器

再次强调，我们将点序列 $p_0, p_1, p_2, \cdots$ 称为 p_0 的轨道。

注意：可以把 Sierpinski 垫片看作 IFS。

按照 IFS 的说法，Sierpinski 垫片的第 k 个点 p_k 是从 p_{k-1} 利用等式：

$$p_k = (p_{k-1} + T[\text{random}(3)])/2 \tag{2.8}$$

得到的，即它是前点 p_{k-1} 和随机选的父三角形 3 个点中的一个，所形成的中间点。

必须注意，x 和 y 分量要分别构造。因此，所迭代的函数是：

$$f(.) = ((.) + T[\text{random}(3)])/2$$

写一个应用程序生成 Sierpinski 垫片，使用式（2.8）的 IFS 概念，用前面的点来表示新点。在白背景上画生成的序列点，直到绘制的点数达到 2000。将屏幕窗口大小设为 600×800 像素，显示标题为 OpenGL makes this all so easy!（OpenGL 使所有这一切变得如此容易）。屏幕窗口是黄金分割的矩形吗？在你的程序中，可以试验不同的绘制颜色、点大小和窗口标题。至少在一个试验中，调整窗口大小使其成为一个黄金分割的矩形。

项目 2：姜饼人

图 2.51 所示的姜饼人是基于另外一个 IFS 产生的。可以将其绘制成点丛。这张图是混沌理论【Peitgen88，Gleick87，Schroeder91】中的常见图形，这是因为它是“奇特吸引体”的一种形式：这些序列的点被吸引进了一个区域中，构成了具有奇特六边形空洞的姜饼人。

在产生姜饼人的过程中没有随机数；每个新点 q 都是遵循下列规则由前一个点 p 所创建的：

$$q.x = M(1+2L) - p.y + |p.x - LM|$$
$$q.y = p.x \tag{2.9}$$

显然，常数 M 和 L 都是精心选择的，用以在显示器上缩放和放置该姜饼人(值 $M=40$，$L=3$，对于 640×480 像素的显示器来说是个不错的选择)。

注意新点 q 的 y 坐标就是前点 p 的 x 坐标。

图 2.51 一个典型的姜饼人

为了更加清晰地了解它的产生过程，最好用特定的数手工迭代几步。

编写一个程序，允许用户用鼠标为迭代选择起始点，然后绘制构成姜饼人的点。开始时用 M、L 的建议数，然后用别的数做试验，使用一些不同的显示，如 800×600 或 1024×768，你推荐的 M、L 的优选数是多少？

你将会观察到，对于一个给定的起始点，在图案重复之前(也就是说，它不再改变了)只出现一些为数不多的点。不同的起始点会引发不同的图案。修改你的程序，使得你可以用鼠标输入另外的起始点并添加到图形中。

练习

姜饼人上的一个固定点

试着指出这个处理过程中会出现一个固定点：$((1+L)M, (1+L)M)$。也就是说，等式(2.9)确定的过程中具有固定点(这可能是产生姜饼人的、最不好的一个起始点，为什么?)。

案例 2.3 黄金分割

(难度：Ⅰ)长宽比是矩形的一个重要属性。几个世纪以来，有一个特殊的长宽比，由于它在艺术品中带来的惊人效果，已经得到了广泛的关注：这就是黄金矩形的长宽比。黄金矩形被当作最令人赏心悦目的矩形，它既不太窄也不太宽。它出现在古希腊帕台农神庙、列奥纳多·达·芬奇的《蒙娜丽莎》、萨尔瓦多·达利的《最后的晚餐》以及许多 M. C. Esher 的作品中。黄金矩形的实质是基于一个迷人的量值：黄金比例 $\phi=1.618033989\cdots$。这个数值 ϕ 在计算机图形中被广泛地使用。

黄金矩形的形状有独特的性质(回顾图 2.34(c))：如果从该矩形中移走一个正方形，剩下的部分仍然是一个黄金矩形！必须具有怎样的值才能具有这个性质呢？注意在图 2.43(c)中，移走一个正方形后，剩下的长方形高为 1。所以，黄金矩形必有 $1/\phi$ 宽，因此

$$\phi = 1 + \frac{1}{\phi} \tag{2.10}$$

这导出一个二次方程，容易解出

$$\phi = \frac{1+\sqrt{5}}{2} = 1.618033989\cdots \tag{2.11}$$

这大概是标准的 3×5 索引卡片的长宽比，从等式(2.10)可以看出，如果 Φ 减 1，ϕ 的倒数是 $1/\phi=0.618033989\cdots$，这是黄金矩形立在其短边时的长宽比。

数 ϕ 在许多方面是不平常的，两种有趣的形式是：

$$\phi = \sqrt{1+\sqrt{1+\sqrt{1+\sqrt{1+\cdots}}}} \tag{2.12}$$

和连续的分数表示

$$\phi = 1 + \cfrac{1}{1+\cfrac{1}{1+\cfrac{1}{1+\cdots}}} \tag{2.13}$$

这两个等式是精确的，都容易证明(怎样证明?)。

这种黄金矩形包含一个小黄金矩形的概念，说明了图形的一种“无限回归”形式：图形包含图形，再包含图形，无限地包含下去。图 2.52 演示了这个现象：从图形中不停地移走正方形，产生另一个黄金矩形。

编写一个应用程序以绘制黄金矩形的回归图，并使这个黄金矩形位于屏幕窗口(宽 600 像素，高 400 像素)的中心(首先确定最大的黄金长方形的大小和位置，以匹配屏幕窗口大小。图形应当持续回归直至最小的长方形大约为 1 像素大小)。

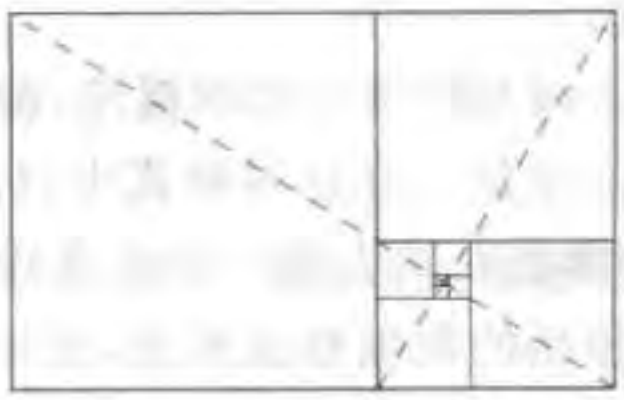


图 2.52　黄金矩形的无限回归

关于黄金比例还有很多要说的内容，在【Gardner61、Hill78、Huntley70 和 Ogilvy69】中可以找到许多乐趣。比如，在下一章我们会看到黄金五角形，在第 9 章会看到两个柏拉图实体、十二面体和二十面体，它们都含有 3 个相互垂直的黄金矩形。

练习

其他具有黄金比例的物体

等式(2.10)显示了 ϕ 是一个涉及数字 1 的重复平方根。那么下面这个式子的值是多少?

$$w = \sqrt{k+\sqrt{k+\sqrt{k+\sqrt{k+\cdots}}}}$$

关于 ϕ 和黄金矩形

(1) 证明等式(2.12)和等式(2.13)的正确性。

(2) 找出图 2.52 中两条对角线的交点，并且证明这个点就是黄金矩形的收敛点。

(3) 使用等式(2.10)来得到下面这个关系

$$\phi^2 + \frac{1}{\phi^2} = 3 \tag{2.14}$$

黄金轨道

从等式(2.12)和等式(2.13)可以看出,黄金比例 ϕ 是重复应用一定函数的极限值。第一个函数是 $f(.)=\sqrt{1+(.)}$。第二个函数是什么呢?将这些表达式看成迭代函数,ϕ 被看作是轨道对某个起始值的收敛值(起始值隐含在表达式中的"…"中)。可以用手摇计算器来探讨该使用哪些起始值而且仍然可以使过程收敛到 ϕ。

案例 2.4 如何创建和使用折线段文件

(难度:Ⅱ)复杂图形如 dino.dat 由一个很大的折线集合构成。折线数据一般都是存在一个文件里,以便在将来需要重新构建这个图形时,只要将折线数据读到程序中就可以了。2.3.1 节介绍了这种文件的一个实用格式,并且描述了画折线例程 drawPolyLineFile()。

图 2.23 中所示的恐龙文件 dino.dat 在本书的网站上可以找到,此外网站上还有其他的折线文件工使用。

(1) 编写一个程序,可以从文件中读取折线数据并依次绘制每条折线。用文本编辑器至少生成一个你自己感兴趣的折线文件,将文件中的数据用你编写的程序绘制出来。设计好你的程序,确保画折线时不需要每次都读折线数据文件,可以用数组或链表方式存储折线数据。

(2) 扩展上面的程序,使其能够接受其他文件格式。例如,程序可以接受差分编码的 x 和 y 坐标。在这种格式中,折线的第一个点(x_1, y_1)按照以前的方法编码,但是其余的(x_i, y_i)都是在减去前一个点之后才编码的。因此,该文件包含($x_i - x_{i-1}$, $y_i - y_{i-1}$)。这种编码与原始的数据格式相比,使用的有效数字少,所以产生的数据文件也更加紧凑。用这种格式做个实验。

(3) 对前面的文件格式做些修改,使文件中每条折线都和一个颜色值相关。颜色值与表明折线包含点数的数值在一行上。用这种格式做个实验。

(4) 修改折线绘制例程,使它在折线的点数为负数时绘制一个封闭多边形。像在这里:

```
-3              负号:所以这是个多边形
0      0        该多边形的第一个点
35     3        该多边形的第二个点
57     8        同样将这个点连接到多边形的第一个点上
5               正号,离开该格式,并和往常一样打开
0      1
12     21
23     34
...
```

第一个折线绘制成为三角形:它的最后一个点和第一个点相连。

案例 2.5 迷宫

(难度:Ⅲ)在迷宫中寻找路径似乎永远具有魅力(参见【Coxeter74】)。你可以在计算

机上生成一个复杂的迷宫，并画成图形来观察是如何穿越它的。图 2.53 显示了一个具有 100 行和 150 列的长方形迷宫，我们的目标是寻找一条路线从左边界的开口穿过迷宫到达右边界的开口处。虽然可以用试错法手工遍历该迷宫，但是开发一个算法自动进行遍历会更有趣。

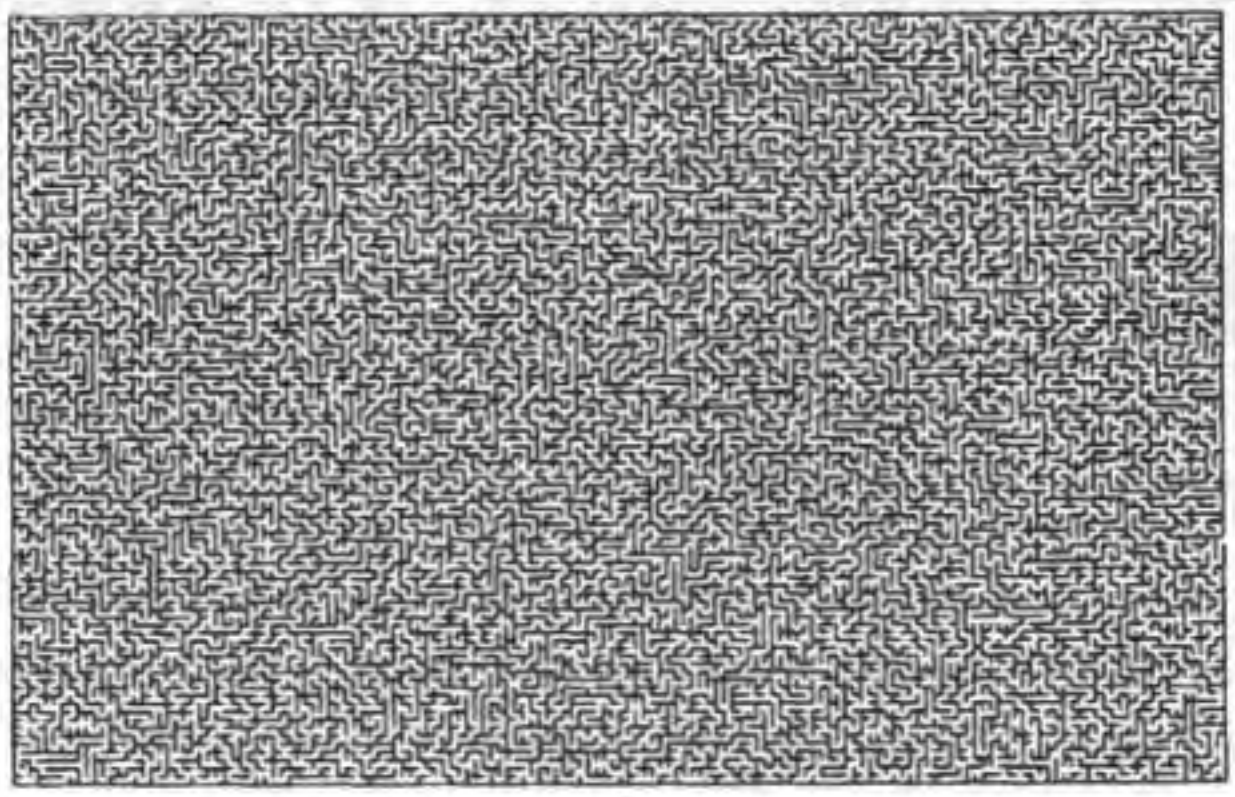


图 2.53　迷宫

编写并执行一个程序：

(1) 生成并显示一个具有 R 行和 C 列的矩形迷宫；

(2) 找到并显示一条从迷宫起点通向终点的路径。

该程序可以随机生成迷宫，但是它们必须是恰当的。也就是说，$R \times C$ 个单元格的每一个都有一个唯一的、弯弯曲曲的路线连接到每个其他单元格。

怎样表示一个迷宫呢？一种方法是，对每个单元格都做出声明，无论其北墙完整还是东墙完整，建议使用下面的数据结构：

```
char northWall[R][C], eastWall [R][C]
```

如果 northWall[i][j]是 1，那么第 ij 个单元格就有固定的墙；否则该墙就不存在。第 0 行是起始行，这一行的单元格都位于迷宫的底部，并且北墙构成了迷宫的底边界。同样，指定 eastWall[i][0]出现在迷宫左边界的豁口。

创建一个迷宫　开始时所有的墙都是完整的，这个迷宫只是由水平线和垂直线组成的网格。用程序绘制出该网格，将一个不可见的“老鼠”放在任意选择的单元格上，该老鼠的任务是“吃掉”墙壁，以连接相连的单元格。该老鼠检查 4 个相邻的单元格(上、下、左、右)，并询问每个邻居是否有 4 堵完整的墙。如果不是，该单元格就已经被访问过了，所以它已经在某条路径上了。老鼠可能会检测几个未被访问的候选单元格：随机地选择一堵墙并将其咬穿，并将其他候选单元格的位置存储到栈上，擦除被吃掉的墙壁，然后老鼠重复该过程。当它陷进一个死胡同时(被已访问过的单元格包围)，就会弹出一个未访问过的单元格，继续上述过程。当栈为空时，迷宫中所有的单元格都被访问过了，起点和终点单元格就会被随机选择出来，一般都沿着迷宫的边界选择。随着老鼠咬穿墙壁，观看迷宫的动态形成过程十分有趣。

运行迷宫　为了运行这个迷宫，我们使用一个回溯算法。在每一步，老鼠都会努力移向随机的方向。如果没有墙壁，它就将该位置存入栈上，并移向下一个单元格。老鼠所在的单

元格用一个红点绘制出来。当老鼠钻进死胡同时,可以将该单元格的颜色变为蓝色,并通过从栈中弹出而回溯。该老鼠甚至可以将墙壁举起以避免再次陷进死胡同。

补遗　合适的迷宫不太有挑战性,因为你可以用肩靠墙的规则来遍历它们。按照这个规则,你可以通过左肩蹭左墙的方式跟踪这个迷宫。遇到死胡同时,可以扫描一周并重新跟踪路径,永远是和墙壁打交道,你最终会到达目的地。为了使其更加有趣,将起点和终点置于迷宫的内部,仍然让老鼠吃掉多余的墙壁(可能会随机吃掉 1/20)。按照这种方式,就可能形成循环,它们围绕终点单元格,从而击败了肩靠墙的方法。

进一步阅读

还有几本书介绍了 OpenGL 的使用。其中 Woo、Neider 和 Davis 所写的《OpenGL 编程指南》是一份很好的资料。【Ball 和 Coxeter74】提供了一本涉及许多数学方面主题的书,这本书很有吸引力,可读性很强,以一种新鲜的方式解答了一些似乎怪异的事,为那些以解决难题和娱乐数学为乐趣的人提供了大量的难题。

因特网上有丰富的资源。例如,OpenGL 主要资源仓库 http://www.opengl.org/,Peter Van de Linden 编写的"Expert C Programming: Deep C Secrets"【VanderLinden94】,提供了 C 语言方面有深度的资源。

第3章 更多的绘图工具

本章学习目标

- 介绍视口和裁剪
- 开发窗口到视口的变换
- 开发一个经典的裁剪算法
- 创建世界坐标中绘图的工具
- 开发选择窗口和视口的方法，以达到最佳视觉效果
- 基于正多边形，弧线，圆建立图形
- 描述参数定义的曲线，并且理解它们的绘制技术

预览

3.1 节介绍世界坐标系和世界窗口。3.2 节描述窗口到视口的变换。这使得程序员能在一个合理的坐标系统下工作，所有的图片都按期望的方式被映射到显示器表面上，从而简化了图形应用程序。这一节还讨论了程序员（以及用户）如何选择窗口和视口以实现他们所期望的绘图。一个关键的性质是窗口和视口的纵横比必须要一样，否则就会有变形。图 3.1(a)是在正确的纵横比下的一张猫的照片；图 3.1(b)是在纵横比特别小时，同一张猫的照片。

某些步骤是可以自动完成的。在 3.3 节中我们设计了一个经典的裁剪算法，它可以删除图像中所有在世界窗口外的部分。

正如序言中所说，面向对象的程序设计（OOP）会在本书的各个部分中使用，用于简化程序员的应用程序开发，或者预防程序员犯一些常见的但又很难发现的错误。一个完整的 OOP 的介绍已经超出了本书的范围，但它的一些关键的特性会在使用时介绍。

第 3.4 节讲述了如何通过正多边形、弧线、圆来绘制有趣的图形。第 3.5 节描述了曲线的不同表示方法，并介绍了非常有用的参数形式，这种形式允许直接绘制复杂的曲线。我们考虑了在二维和三维情况下的曲线。这一章的最后是一些实例学习。

3.1 概　　述

在第 2 章，我们绘图使用的是屏幕窗口的基础坐标系，它本质上是以像素为单位，**屏幕坐标**在 x 方向从 0（屏幕左边）延伸到 screenWidth－1，y 方向上从 0（通常是屏幕上方）延伸到 screenHeight－1。这意味着我们只能使用非负的 x 和 y，并且如果要绘制一个合适大小

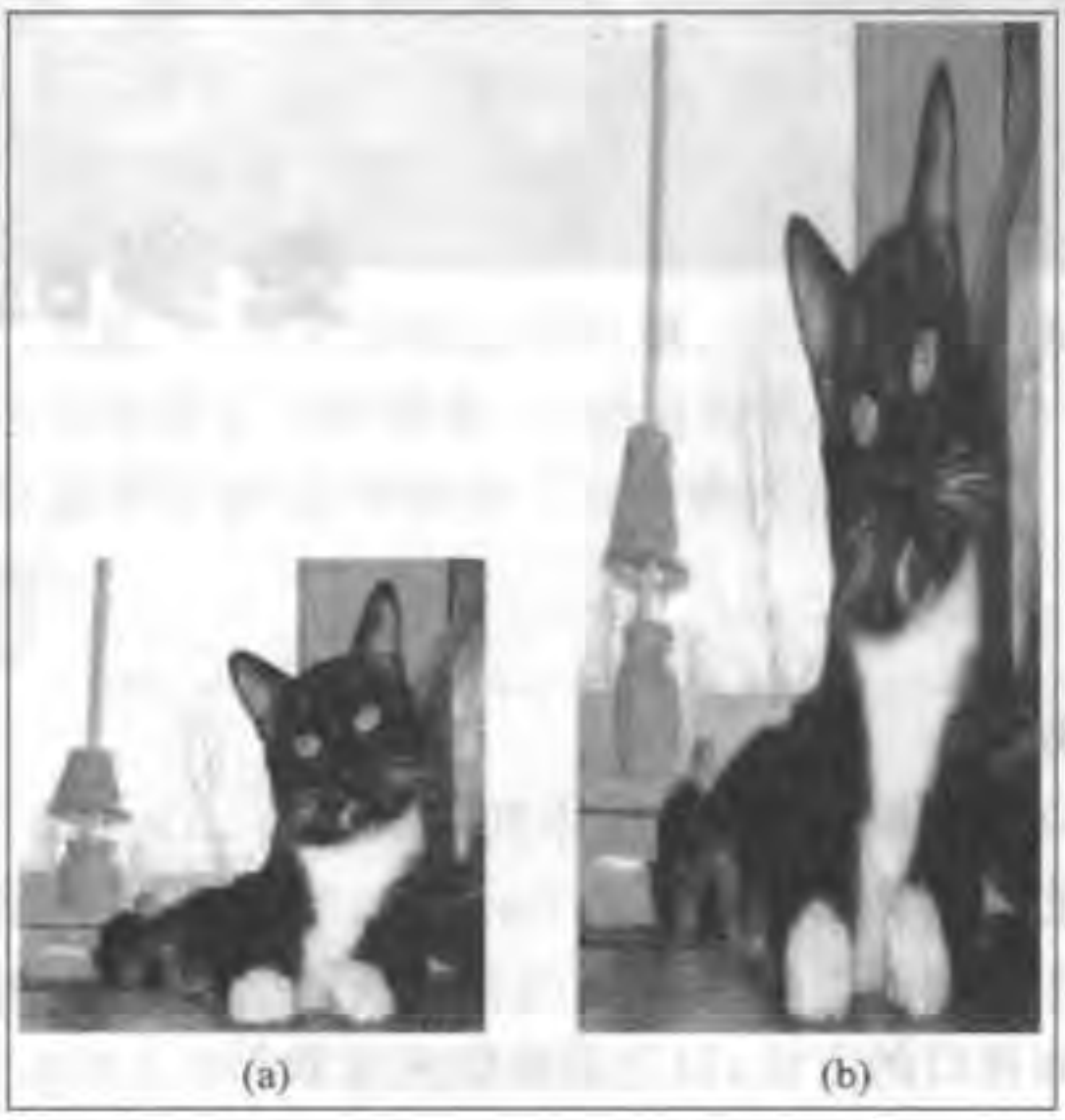


图 3.1

(a) 合适的纵横比下的照片;(b) 同一张照片,但窗口特别窄

的图片,这些数值需要被扩大到一个比较大的范围(几百个像素)。

但是,对一个给定的问题,我们可能并不想按像素来考虑。或许 x 在 $-1\sim1$ 之间变化,y 在 $-100\sim20$ 之间变化,思考会更加自然一些(回想一下,在图 2.15 的画点图的程序中,缩放和移动数值是非常不方便的)。显然,我们希望将程序中用于描述对象几何信息的数值,和那些用于表示对象在屏幕中大小和位置的数值区分开来。前者通常被看作一个**建模**(modeling)的任务,后者是一个**观看**(viewing)的任务。

在这一章我们要介绍一些方法,使得程序员可以用最适合他们手中问题的坐标系来描述对象,并且可以自动的缩放和平移图片,使得其能正确地在屏幕窗口中显示。这个描述对象的空间被称为**世界坐标系**(场景中的物体在实际世界中的坐标)。在一个合适的单位下,它就是数学中常用的笛卡儿 xy 坐标系。

我们在世界坐标系中定义一个对齐的矩形(aligned rectangle,即矩形的边与坐标轴平行——译者注)的**世界窗口**[①]。这个世界窗口指明要绘制世界的哪一部分。对此可以理解为,任何在窗口中的部分需要被绘制,而窗口外的部分被裁减并不被绘制。OpenGL 会自动地做裁减。

此外,我们在显示器的屏幕窗口上定义一个对齐的矩形的**视口**。OpenGL 会自动建立世界窗口和视口的**变换**(包括缩放和平移)。当世界窗口中所有对象都被绘制时,对象在世界窗口中的部分会被自动的映射到视口中——换句话说,被映射到**屏幕坐标**中,即像素在显示器上的坐标。选择对齐的矩形作为窗口和视口(屏幕窗口)的形状是有一些重要的原因。对齐的矩形在几何上很简单:只需要四个参数就可以完全描述它。此外,检测一个点是在窗口内还是窗口外非常容易,这可以大大简化裁减算法。第三个重要的性质是对齐的矩形

① 如前所述,"窗口"在图形学中有很多含义,常常导致混淆。必要的时候,我们试着用"世界窗口","屏幕窗口"来区分不同的含义。

是凸的，这也简化了一些在绘制过程中的算法。因此，程序员就象通过一个窗口去看要被绘制的物体，并将窗口中能被看到的部分的快照放在显示器的视口中。这个窗口/视口的方法使得一些自然的操作变得容易，比如放大场景中的细节，以及鸟瞰一个场景。

窗口/视口的这种联合最主要的优点在于，一旦窗口和视口被指定了，程序员可以为期望的图片编写一个算法，而不用考虑图片的大小和位置。而所有潜在的操作（缩放、位置和裁减）会被自动完成，并用于将图片放到视口中去。

我们首先讨论映射，它为我们提供坐标系之间的自动转换。然后我们再讨论如何进行裁减。

3.2　世界窗口和视口

我们用一个例子来说明世界窗口和视口的使用。假设你想检查一个数学函数的性质，比如信号处理中非常重要的函数 sinc，它的定义是：

$$\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x} \quad 当\ x \neq 0\ 且\ \mathrm{sinc}(0) = 1 \tag{3.1}$$

sinc()函数在 0 点的值被定义为 1。你想知道 x 变化时，sinc(x)如何弯曲和扭动。假设你知道 x 在 $-\infty$ 到 $+\infty$ 间变化，而 sinc(x)的值在 $-1\sim1$ 之间变化，并且对 x 在 0 附近的值特别感兴趣。（对于更复杂的函数，你可能根本不知道**这些**）所以你想画一条中心在(0,1)附近的曲线，并且展示当 x 在很小范围内（比如 $-4.0\sim4.0$ 之间）变化时的 sinc(x)。注意：sinc 不能直接在 $x=0$ 处计算，因为那会导致除数为 0 的错误；因此，我们需要先判断 x 是否是 0。

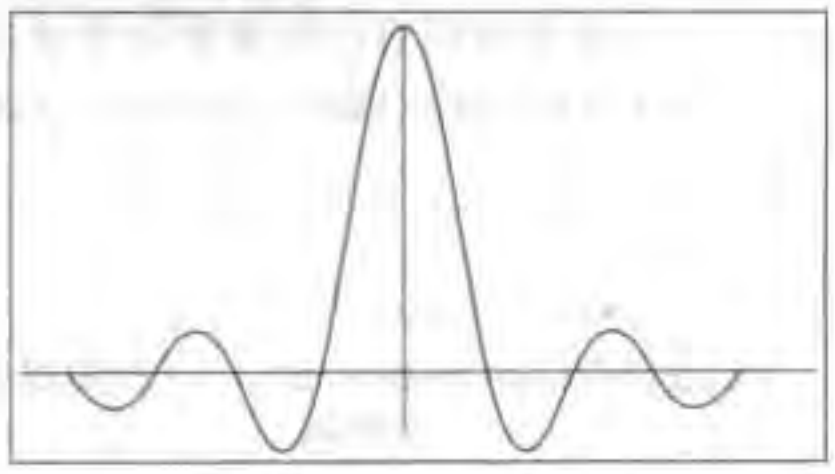


图 3.2　sinc 函数，图 3.3 程序执行结果的截图

图 3.2 给出了绘制这个函数的一个例子。在学习完本节后面的窗口/视口工具后，你就能编写出图 3.3 中绘制这个函数的完整的程序。它使用 setWindow()来建立世界窗口，并使用 setViewport()来建立视口。在图 3.3 后，我们会直接讨论如何定义这些函数。

非常重要的问题：当编写这个程序并将其编译成机器码时（也就是**编译时间**），程序员可能完全不知道 sinc 函数的这部分图像会在哪或有多大。这时使用世界窗口就变得特别的重要。

注意到这些例子的代码是在针对该问题的自然坐标系中操作的。x 是在 -4.0 到 4.0 中以很小的增量变化。这里关键的问题是如何缩放和平移不同的(x,y)，使得图像能在屏幕窗口中恰当的显示。

通过设置一个世界窗口和一个视口，并建立它们之间的一个合适的映射，这样我们就可以完成适当的缩放和平移。这个窗口和视口都是由程序员指定的对齐的矩形。如图 3.4，窗口是在世界坐标系中，而视口是屏幕窗口的一部分。由图可知，在世界窗口中的东西会被缩放和平移，以显示在视口中，其余的都被裁减而不显示出来。

```
//绘制 sinc 的函数

#include <windows.h>
#include <iostream.h>
#include <math.h>
#include <gl/GL.h>
#include <gl/GLU.h>
#include <gl/GLUT.h>
const float pi=3.14159265358979; //用这个常量近似 pi

// --- setWindow ---
void setWindow(GLdouble left, GLdouble right, GLdouble buttom, GLdouble top)
{
    //定义我们自己的函数设置窗口(后面会详细解释)
    glMatrixModel(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom, top);
}

// --- setViewport ---
void setViewport(GLint left, GLint right, GLint bottom, GLint top)
{
    //定义我们自己的函数设置视口(后面会详细解释)
    glViewport(left, bottom, right-left, top-bottom);
}

// --- myDisplay ---
void myDisplay(void)      //用世界坐标绘制 sinc 函数
{
    glClear(GL_COLOR_BUFFER_BIT);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glBegin(GL_LINE_STRIP);
    /* 自动使用定义好的窗口和视口,正确的裁剪和映射 */
    for (float x=-4.0; x<4.0; x+=0.1)      //绘制图像
    {
        if (x==0.0)
            glVertex2f(0.0, 1.0);
        else
            glVertex2f(x, sin(3.14159*x)/(3.14159*x));
    }
    glEnd();
    glFlush();
}
```

图 3.3 显示 sinc 函数的完整程序

```
// --- myInit ---
void myInit(void)
{
    glClearColor(1.0, 1.0, 1.0, 0.0);          //白色背景
    glColor3f(0.0f, 0.0f, 1.0f);               //蓝色线条
    glLineWidth(2.0);                          //线宽是 2
}

// --- main ---
void main(int argc, char ** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 150);
    glutCreateWindow("The Famous Sinc Function");
    glutDisplayFunc(myDisplay);
    myInit();
    setWindow(-5.0, 5.0, -0.3, 1.0);     /* 调用函数设置窗口 */
    setViewport(0, 640, 0, 480);         /* 调用函数设置视口 */
    glutMainLoop();
}
```

图 3.3（续）

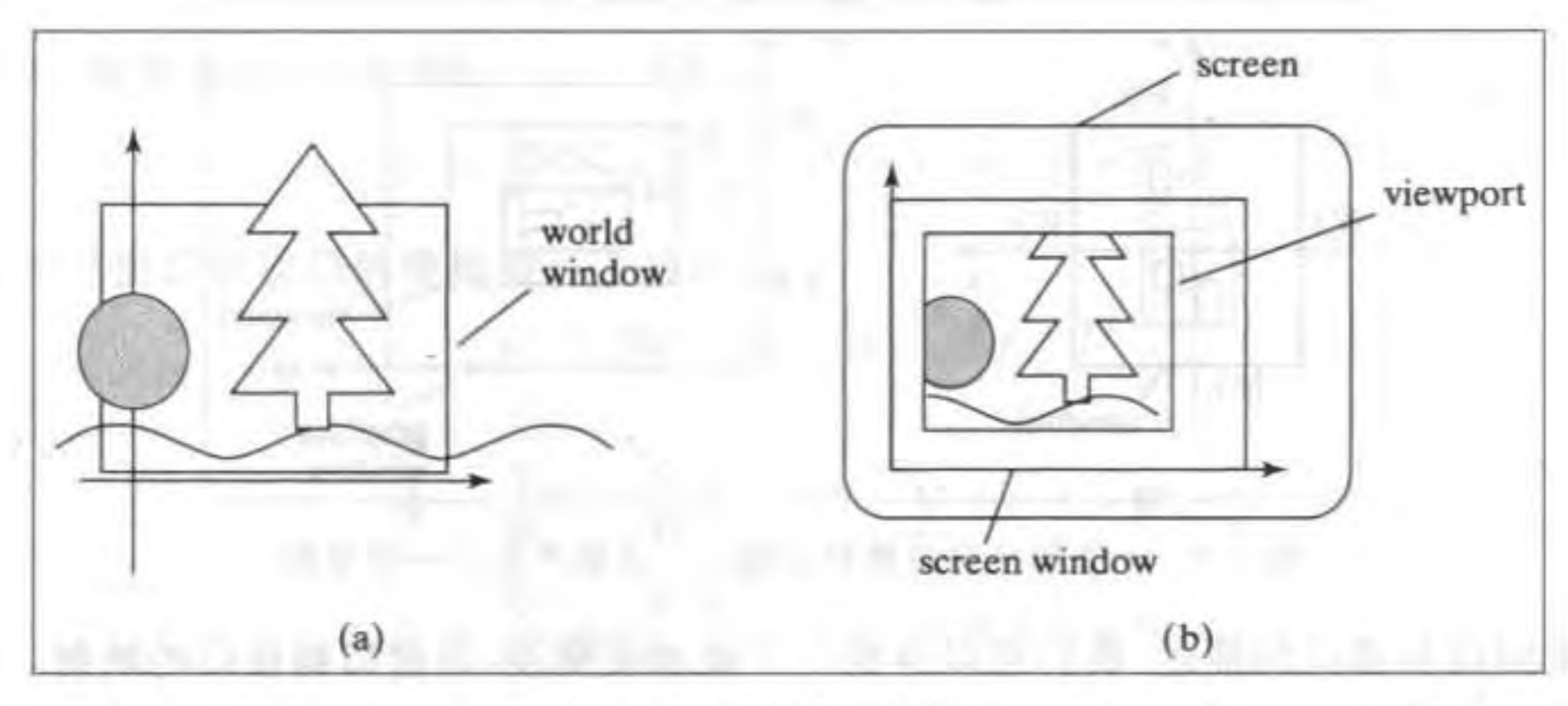


图 3.4　世界窗口和视口

我们不仅想描述如何让 OpenGL 来做这件事——这很容易，还想描述它在内部是如何操作的，以便解释所用的底层算法。这一章我们只考虑二维的版本，但在之后（第 5 章）我们会看到这些想法如何被自然地应用于相机看到的三维世界。

3.2.1 窗口到视口的映射

图3.5显示了世界窗口和视口的更多细节：世界窗口由它的左、上、右、下的边界描述，分别是$W.l$、$W.t$、$W.r$和$W.b$。视口在屏幕窗口坐标系中(从屏幕的某个位置展开)，被同样的描述，使用$V.l$、$V.t$、$V.r$和$V.b$，单位是像素[①]。

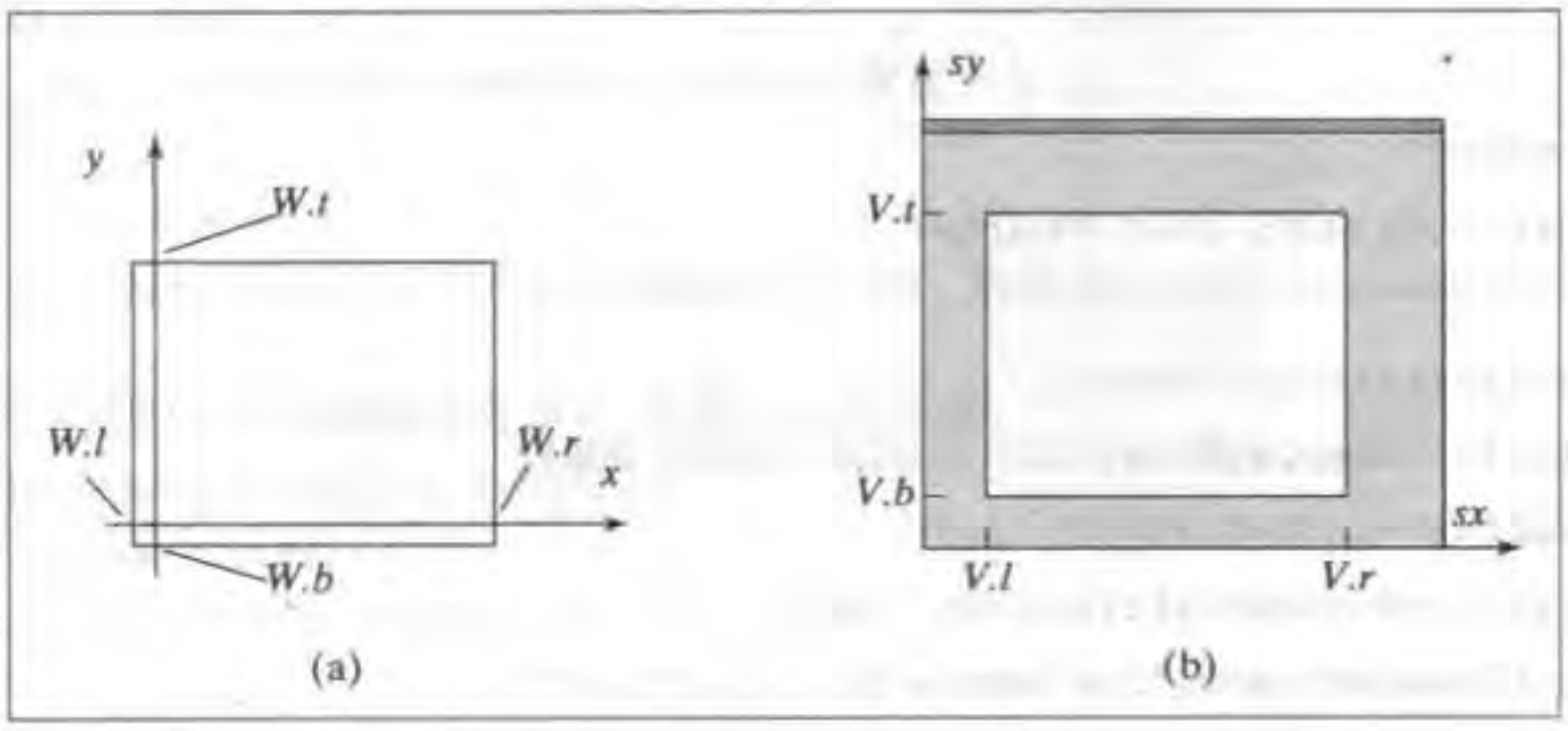


图3.5 指定(a)窗口和(b)视口

世界窗口必须是对齐的矩形，但它可以有任意大小以及出现在任何位置。类似的，视口也可以是任意对齐的矩形，虽然它常常被选成完全位于屏幕窗口内。此外，世界窗口和视口并不需要有同样的纵横比，当然不同的纵横比会带来图像的变形。如图3.6所示，出现变形是因为窗口中的图像必须被拉伸以适合视口。随后我们将看到如何设置视口的纵横比，使之总是和窗口匹配，即使用户改变了屏幕窗口的大小。

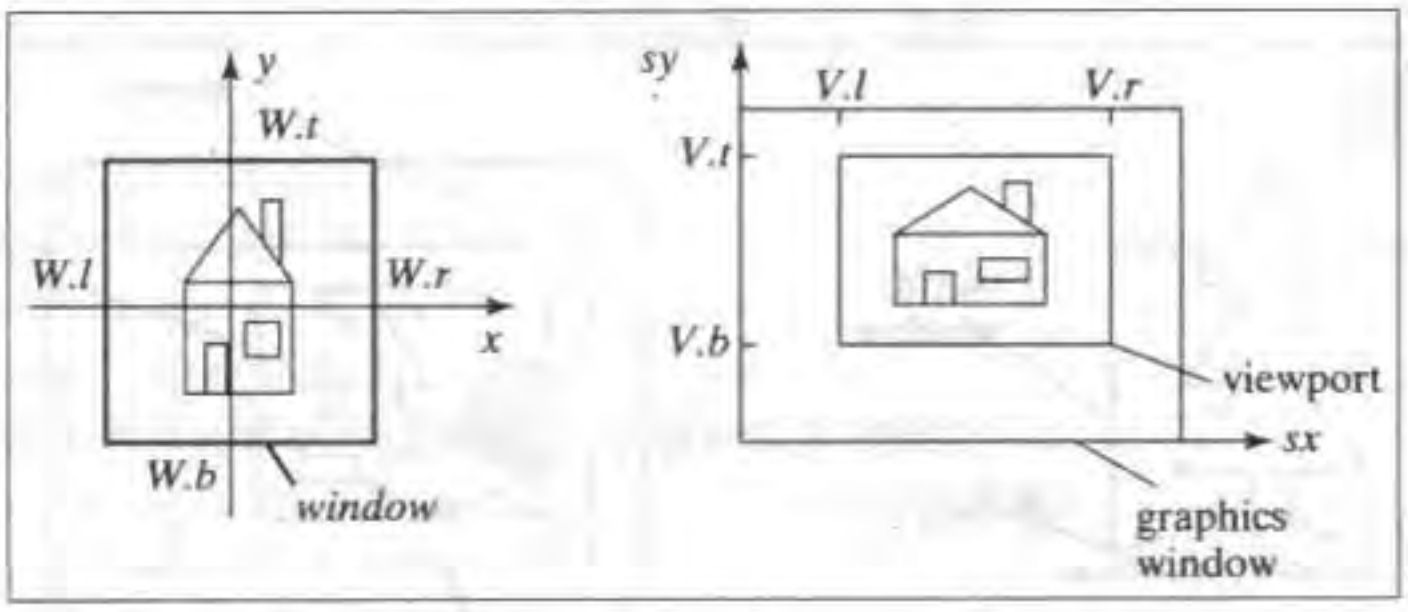


图3.6 一张图从窗口映射到视口，这里产生了一些变形

给定窗口和视口的描述，我们可以得到一个**映射或变换**，即**窗口到视口的映射**。这个映射是基于一个等式，它对每一个在世界坐标下的点(x, y)，产生屏幕坐标系中的一个点(sx, sy)。我们希望这个映射是"保持比例"的映射，也就是说，例如x是位于窗口中距离左边界40%处，则sx应该位于视口中距离左边界40%处。同样，对于某个比例f，如果y在窗口中距离下边界$1/f$高的位置，则sy应该是在视口中距离下边界同样的$1/f$高的位置。

保持比例的性质使得这个映射有线性形式：

① 为了简明，我们在数学等式中用"l"表示"左边"，"t"表示"上边"，等等。

$$sx = A * x + C$$
$$sy = B * y + D \tag{3.2}$$

其中 A、B、C、D 是常数。常数 A、B 缩放 x 坐标和 y 坐标，而 C、D 平移它们。

如何确定 A、B、C、D 呢？首先考虑 x 的映射，如图 3.7，保持比例的性质说明 $sx-V.l$ 与视口宽度 $V.r-V.l$ 的比例，必须等于 $x-W.l$ 与窗口宽度 $W.r-W.l$ 的比例。

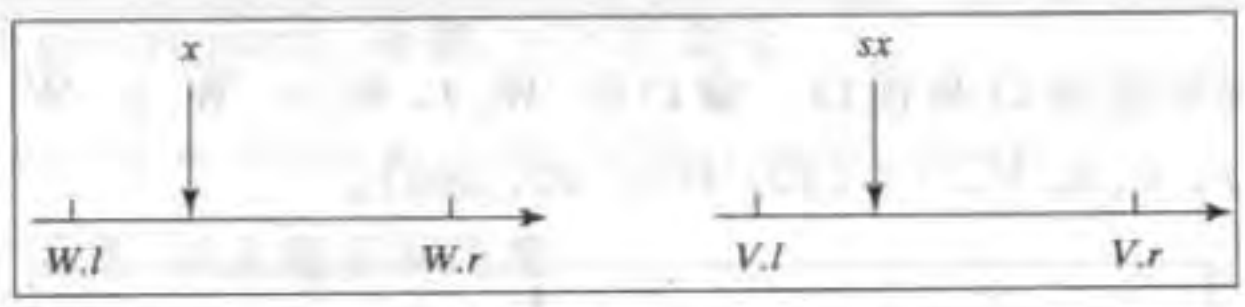


图 3.7　x 到 sx 映射的保持比例的性质

基本上，如果你位于视口的中间，那么你必须也位于窗口的中间，并且对于其他比例也是类似的。所以：

$$\frac{sx - V.l}{V.r - V.l} = \frac{x - W.l}{W.r - W.l}$$

或者，经过一些代数变换后：

$$sx = \frac{V.r - V.l}{W.r - W.l}x + \left(V.l - \frac{V.r - V.l}{W.r - W.l}W.l\right).$$

现在，把 A 看作放大 x 的部分，而 C 看作常数，我们有：

$$A = \frac{V.r - V.l}{W.r - W.l}, \quad C = V.l - A \cdot W.l$$

类似的，y 方向上的保持比例性质规定：

$$\frac{sy - V.b}{V.t - V.b} = \frac{y - W.b}{W.t - W.b}$$

把 sy 写成 $By+D$，得到：

$$B = \frac{V.t - V.b}{W.t - W.b}, \quad C = V.b - B \cdot W.b$$

现在对窗口到视口的变换做一下总结：

$$sx = Ax + C, \quad sy = By + D \tag{3.3}$$

和

$$A = \frac{V.r - V.l}{W.r - W.l}, \quad C = V.l - A \cdot W.l$$
$$B = \frac{V.t - V.b}{W.t - W.b}, \quad C = V.b - B \cdot W.b$$

警告：请仔细检查这一部分！

这个映射可用于任意点(x, y)，不管它是否在窗口之中。在窗口中的点映射到视口中的点，而窗口外的点映射到视口外的点。

(重要！)用等式(3.3)仔细检查映射的下列性质。

a. 如果 x 是在窗口的左边界：$x=W.l$，则 sx 是在视口的左边界：$sx=V.l$。

b. 如果 x 是在窗口的右边界，则 sx 是在视口的右边界。

c. 对于某个比例 f，如果 x 位于窗口宽度 $1/f$ 处，则 sx 位于视口宽度 $1/f$ 处。

d. 如果 x 在窗口左边界外($x<W.l$),则 sx 在视口的左边界外($sx<V.l$)。如果 x 在窗口右边界外,情况也是类似的。

问题:这个变换的推导是否使得窗口和视口的纵横比自动地保持一致?

此外,你还应当检查 y 到 sy 的类似的特征。

例 3.2.1

考虑如图 3.8 所示的窗口和视口。窗口的($W.l$, $W.r$, $W.b$, $W.t$)=(0, 2.0, 0.1, 0),视口的($V.l$, $V.r$, $V.b$, $V.t$)=(40, 400, 60, 300)。

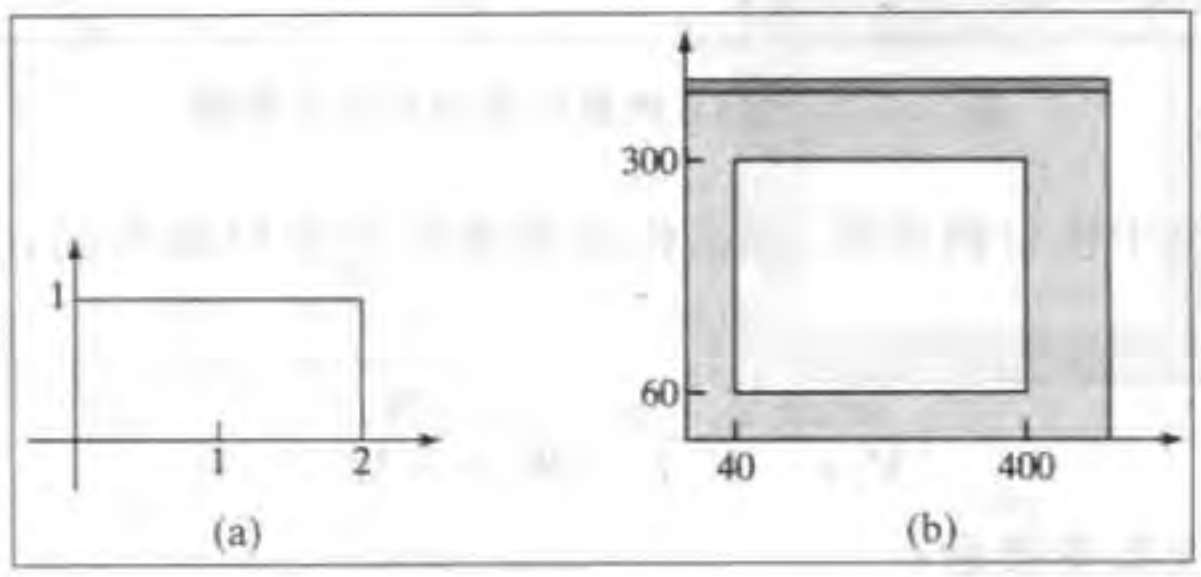


图 3.8 一个窗口和视口的例子

由等式(3.3),我们得到:

$$A = 180, \quad C = 40$$
$$B = 240, \quad D = 60$$

所以对于这个例子,窗口到视口的映射是:

$$sx = 180x + 40$$
$$sy = 240y + 60$$

请检查这个映射是否对我们感兴趣的一些点进行了恰当的映射,比如:

- 每一个窗口的角点都确实被映射到了视口对应的角点。例如,(2.0, 1.0)映射到了(400, 300)。
- 窗口的中心(1.0, 0.5)被映射到了视口的中心(220, 180)。

练习

练习 3.2.1 建立映射

对于世界窗口(−10.0, 10.0, −6.0, 6.0)和视口(0, 600, 0, 400)这个例子,计算 A、B、C 和 D 的值。

"这个窗口和视口有同样的纵横比吗?

答:没有!"

设立窗口到视口的映射

OpenGL 使得窗口到视口的映射非常容易。OpenGL 通过一系列变换完成所需要的映射,自动的传送给定的每一个顶点(通过 glVertex2 * ()命令)。它还自动的剪裁掉对象在世界窗口之外的部分。我们所需要做的工作只是正确的设置这些变换,其余的工作都由

OpenGL 完成。

对于二维绘图来说，世界窗口由函数 gluOrtho2D()设定，它的原型是：

```
void gluOrtho2D(GLdouble left, GLdouble right, GLdouble buttom,
                GLdouble top);
```

对于三维的情况，有另外两个参数需要设定，但这里我们先不考虑。

视口的设定通过 glViewport()函数，它的原型是：

```
void glViewport(GLint x, GLint y, GLint width, GLint ehight);
```

它设置视口的左下角，以及宽度和高度。

因为 OpenGL 通过矩阵来完成所有的变换，因此 gluOrtho2D()[1]的调用必须在 glMatrixModel(GL_PROJECTION)和 glLoadIdentity()这两个函数之后。(在第 5 章将详细讨论在这后面的事情)

因此，要建立例 3.2.1 中的窗口和视口，我们需要使用如下代码：

```
glMatrixModel(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 2.0, 0.0, 1.0);     //设置窗口
glViewport(40, 60, 360, 240);       //设置视口
```

在这之后，每一个使用 glVertex2 * (x, y)发送给 OpenGL 的点，会进行等式(3.3)的映射，并且在窗口的边界处会被自动裁剪掉边缘。(在第 7 章我们会看到在三维情况下是怎么做的，在那里会清楚地看到二维的情况不过是三维情况的一个简单特例)。

如果我们将设定窗口的部分放到 setWindow()函数中(如图 3.3 和图 3.9 所示)，程序就更易读了。我们还使用 setViewport()函数来掩饰 OpenGL 的 glViewport(..)细节。为

```
// — setWindow —
void setWindow(GLdouble left, GLdouble right, GLdouble buttom, GLdouble top)
{
    glMatrixModel(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(left, right, bottom, top);
}

// — setViewport —
void setViewport(GLint left, GLint right, GLint bottom, GLint top)
{
    glViewport(left, bottom, right-left, top-bottom);
}
```

图 3.9　方便的设置窗口和视口的函数

---

① 出现词根“ortho”是因为这种设置窗口的方法事实上就是在 3D 设置一个所谓的正交投影(“orthographic projection”)，这将在第 7 章中看到。

了让 setViewport()更容易使用,我们将其参数作了轻微的调整使得它和 setWindow()相匹配:它们都是按左、右、下、上的顺序。我们的 setViewport 函数使用左、右、下、上做参数,但它调用 OpenGL 最底层的 glViewport()函数,它的参数为 left、bottom、right-left 和 top-bottom。(显然,right-left 等于宽度,top-bottom 等于高度。)

回顾我们曾在第 2 章的 OpenGL 程序中设置窗口和视口的方法。在图 2.11(画点)的程序中使用:

(1) 在 main()中:

```
glutInitWindowSize(640, 480);      //设置窗口大小
```

它设置屏幕窗口的大小是 640×480。因为没有使用 glViewport 的命令,我们使用的是默认的视口。默认的视口就是整个屏幕窗口。

(2) 在 myInit()中:

```
glMatrixModel(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(0.0, 640.0, 0.0, 480.0);
```

这些函数将世界窗设置为对齐的矩形,其两个角的坐标是(0,0)和(640.0,480.0),正好与视口的大小相等。所以这里的窗口到视口是没有任何改变。这比较适合作为入门的首选。

例 3.2.2 sinc 绘制函数的回顾

这个早期的例子显示了一个 OpenGL 程序的所有细节,包括所有的初始化,这可以打消读者对编译和运行程序的顾虑,而且没有省略任何重要部分。比如图 3.3 的那个画 sinc 函数的程序就是这样的一个例子。

例 3.2.3 绘制文件中的折线

在第 2 章,我们使用图 2.23 中的 drawPolylineFile("dino.dat") 函数画了一个图 3.10 中的恐龙。图中的折线数据都存放在"dino.dat"这个文件中。当时还未介绍世界窗口和视口,所以我们只是使用了一些确定的或者默认的东西,很幸运我们也得到了这个恐龙。

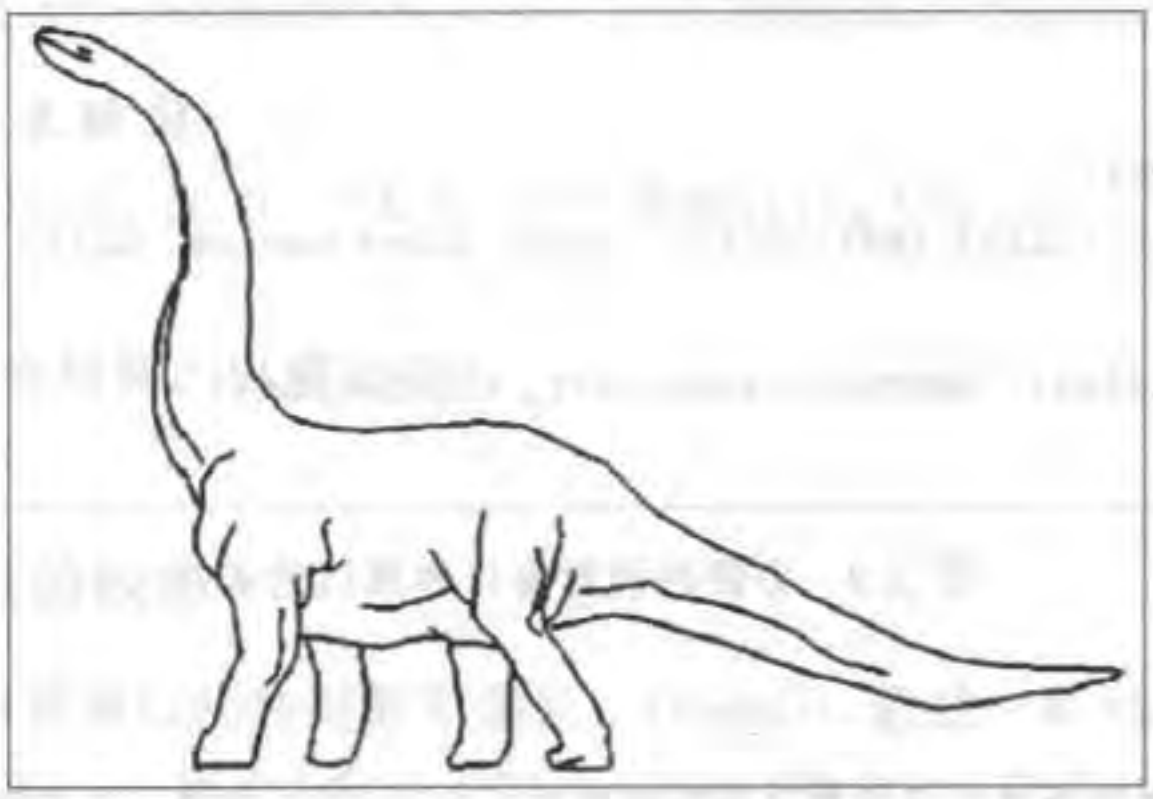


图 3.10 世界窗口中的恐龙

现在我们明白为什么可以绘制出这个恐龙。我们所使用的世界窗口恰好包含了恐龙的数据。所有 dino.dat 中的折线在一个矩形内，其两个角分别是(0，0)和(640，480)，所以这个窗口没有裁剪掉任何部分。

有了设定窗口和视口的工具，我么可以进一步的控制这些状态。下两个例子将演示这一点。

例 3.2.4　用恐龙图形平铺屏幕窗口

为了增加些趣味性，我们可以按某种样式绘制多个恐龙。如果将它们一个接一个的放在整个屏幕窗口，这称作**平铺**屏幕窗口。这个放在不同部分被复制的图片通常叫做**图案**。通过对每个图案实例使用不同的视口，可以很容易做到在屏幕窗口上的平铺。图 3.11(a)显示了一个包括 25 个复制图案的平铺。这个平铺使用下面的代码生成：

```
setWindow(0, 640.0, 0, 440.0);                    //设置一个固定的窗口
for (int i=0; i<5; i++)                           //对于每一列
{
    for (int j=0; j<5; j++)                       //对于每一行
    {
        glViewport(i * 64, j * 44, 64, 44);       //设置下一个视口
        drawPolylineFile("dino.dat");             //再绘制一遍
    }
}
```

每一个图案在一个 64×44 像素大小的视口中被绘制，它的纵横比 64/44 和世界窗口的一致。这样绘制每个恐龙时，都不会发生扭曲。

图 3.11(b)展示了另外一个平铺的例子，此处图案改为交替倒置以产生奇妙的效果。这通过在每 2 次循环时(当 i+j 是偶数时)将窗口翻转实现的：即交换等式(3.3)中的 top 和 bottom 的值。因此，之前的双重循环改为：

```
for (int i=0; i<5; i++)
{
    for (int j=0; j<5; j++)
    {
        if ((i+j)%2==0)                           //如果 i+j 是偶数
```

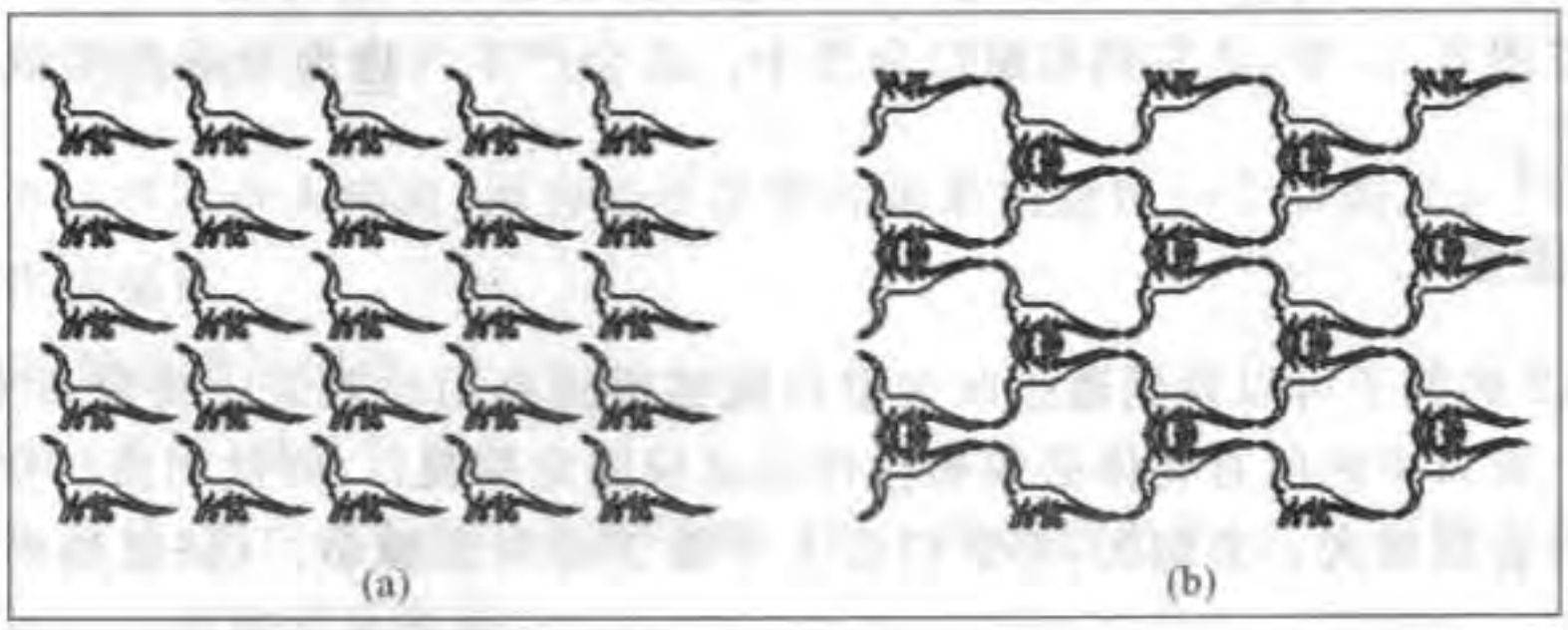


图 3.11　用恐龙的副本平铺屏幕

```
            setWindow(0.0, 640.0, 0.0, 440.0);      //正常的窗口
        else
            setWindow(0.0, 640.0, 440.0, 0.0);      //颠倒窗口
        glViewport(i * 64, j * 44, 64, 44);         //设置下一个视口
        drawPolylineFile("dino.dat");               //在窗口再绘制一遍①
    }
}
```

类似的实验可以创造出视觉上非常有趣的图片。

例 3.2.5 裁减图片的某些部分

设置合适的窗口,也可以裁减一个图片。OpenGL 自动裁减对象在世界窗口外面的部分。假设我们有 3 个视口。图 3.12 左边部分包括一些不同大小的六边形,每一个相对其相邻的图案有轻微的旋转。如果用 W1 作窗口,会显示图 3.12 中间的图案。如果 W2 作为窗口,会显示图 3.12 右边的图案。裁减用于防止折线的不恰当的重叠。假设这个图片是通过执行函数 hexSwirl()(后面会介绍)绘制出来的。注意如果没有裁剪,那么图 3.12 的中间和右边都会显示整个六边形构成的漩涡,并填满这个视口。图 3.12 的三个部分可以用下面的代码框架绘制:

```
setWindow(...);          //每张图片都改变窗口
setViewport(...);        //每张图片使用同样的视口
hexSwirl();              //调用同样的函数
```

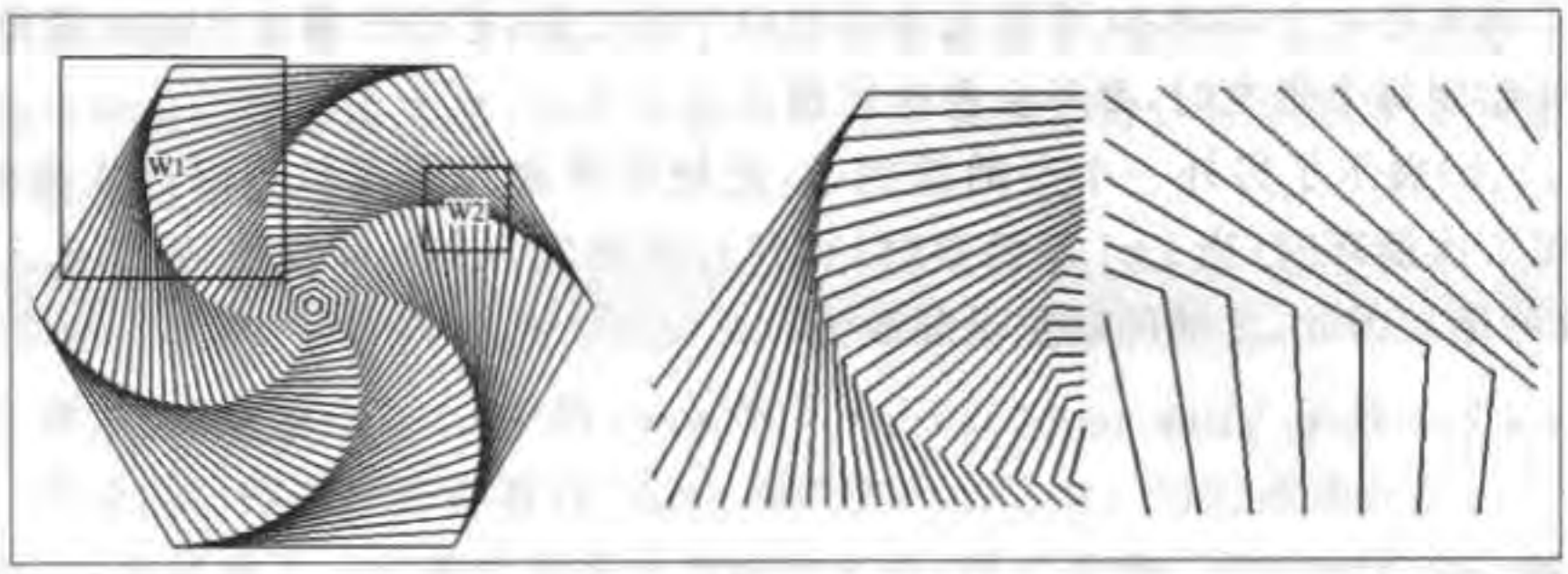


图 3.12 用窗口裁剪图像的一部分

另一方面,显示的内容还依赖于窗口的设置。我们会在下面讨论 hexSwirl()如何工作。注意到在图 3.12 中,从左到右窗口会变小。这会产生六边形漩涡的缩放效果,我们会在后面讨论它。

缩放和漫游

从图 3.12 的例子可以看到通过改变窗口能够得到有用的效果。将窗口变小很像用相机**放大**对象。窗口中的所有物体必须被拉伸以适应固定的视口,因此当窗口变小的时候,其里面的内容就会被放大。类似的,将窗口变大等效于将对象**缩小**。(试想如果图 3.10 的窗

① 看上去改变视口会更容易些,但 OpenGL 不允许视口的高度是负数

口被增大成原来的两倍，看看恐龙会变成什么样子）。相机也可以**漫游**（有时也叫作“全视”），顺着场景滑行，在不同的时间拍摄不同的部分。通过将窗口平移到一个新的位置，漫游就可以很容易实现。

例 3.2.6　在动画中放大图片

考虑建立一个动画，这个动画中相机逐步靠近图 3.12 中六边形的某一部分。我们生成一系列图片，这些图片通常被称作**帧**，每一帧都使用较小的窗口。当这些帧被快速播放时，这个视觉效果就像是相机在靠近对象。

图 3.13 展示了几个所用到的窗口：它们是同心的，并具有固定的纵横比，但相继的帧的窗口尺寸在逐步缩小。对于其中的每一个窗口将视口内所绘的图案可视化。

图 3.14 展示了一个实现这种放大效果的伪代码框架。绘制每一帧都先清一次屏幕，同时窗口越来越小（但固定中心点和纵横比），并将窗口中的图片画在一个固定的视口中。

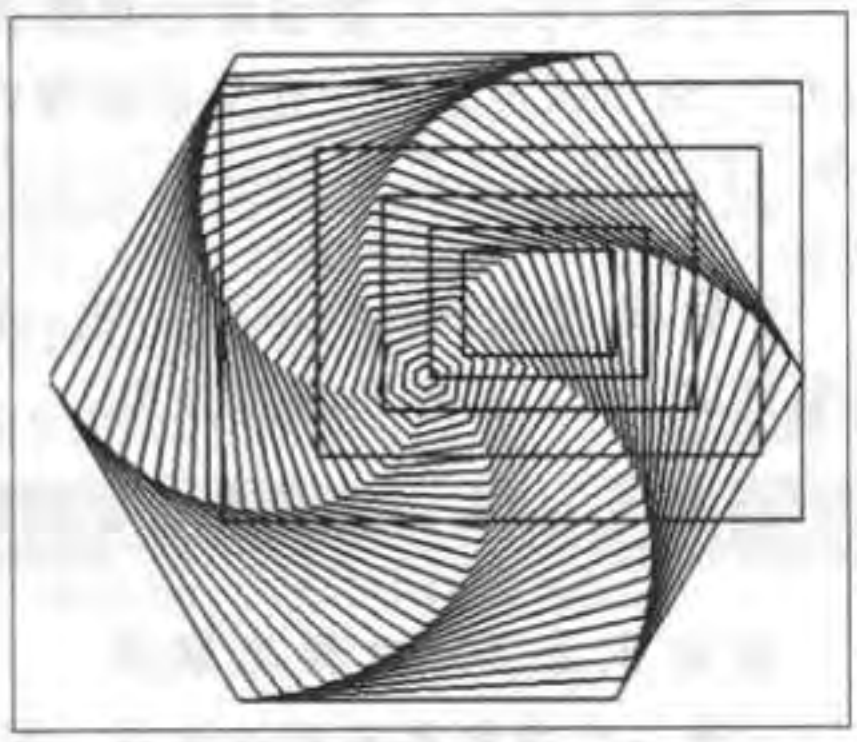


图 3.13　六边形漩涡的放大

```
float cx=0.3, cy=0.2;                          //窗口中心
float H, W=1.2, aspect=0.7;                    //窗口性质
set the viewport
for(int frame=0; frame<NumFrames; frame++)     //对每一帧
{
    clear the screen                           //清除之前图案
    W*=0.7;                                    //减小窗口宽度
    H=W * aspect;                              //保持纵横比不变
    setWindow(cx-W, cx+W, cy-H, cy+H); {       //设置下一个窗口
    hexSwirl();                                //绘制对象
}
```

图 3.14　做一个动画

实现一个连续的动画

前面的方法并不令人满意，因为需要花费时间来拍摄每一个新图片。用户看到的只是下面两个动作的循环：

a. 迅速擦除当前的图像。

b. 新图像（可能）被缓慢重绘。

这里的问题是，用户会看到的是新的帧一行一行创建出来，这是很让人混乱和苦恼的。用户想看到的是下面的重复循环：

a. 当前图像的稳定显示。

b. 将当前的图像迅速的用一个已绘制完成的新图像替换。

这里的技巧是：在用户注视当前图像时，在一个“屏幕外内存”的地方绘制新图像，然后瞬间将绘制好的新图像显示在显示器上。OpenGL 提供了**双缓存**来完成这个任务。它使用一些内存来模拟一个额外的屏幕，这个屏幕不显示在真实显示器上，并且所有的绘制都在这个缓存中进行。(第 9 章会详细讨论这种“屏幕外内存”。)然后 glutSwapBuffers()会将缓存中的图片转移到屏幕窗口上，用户就可以看到了。

为了让 OpenGL 能够使用双缓存，我们在 main 函数中使用 GLUT_DOUBLE 而不是 GLUT_SINGLE 来初始化显示器模式：

```
glutInitDisplayModel(GLUT-DOUBLE | GLUT_RGB);      //使用双缓冲
```

然后，即使绘制折线需要花费一些时间，动画中的图像也会被立即替换到下一张，从而产生更加平滑且舒服的视觉效果。

练习

练习 3.2.2 回旋的漩涡

再来一个裁剪和平铺的例子。图 3.15(a)显示了一个由一个特殊的窗口定义的六边形漩涡。在这个例子中，窗口固定不变，但视口每张图都会变。图 3.15(b)显示了这个图片被复制多次，并且并排平铺在显示器中的结果。试着挑出每一个漩涡。(有些漩涡被翻转了，是哪些呢?)这个结果会让眼睛眩晕，部分是由于眼睛会倾向于将一些小元素构成整个图案。除了翻转之外，下面这段代码创建了这个图案。myDisplay()函数设置了一次窗口，然后在不同视口里面一遍又一遍的绘制剪裁过的漩涡。

```
void myDisplay(void)
{
    clear the screen
    setWindow(-0.6, 0.6, -0.6, 0.6);          //漩涡要被绘制的部分
    for(int i=0; i<5; i++)                    //做一个 5×4 的图案
        for(int j=0; j<4; j++)
        {
            int L=80;                         //每个视口的平移量
```

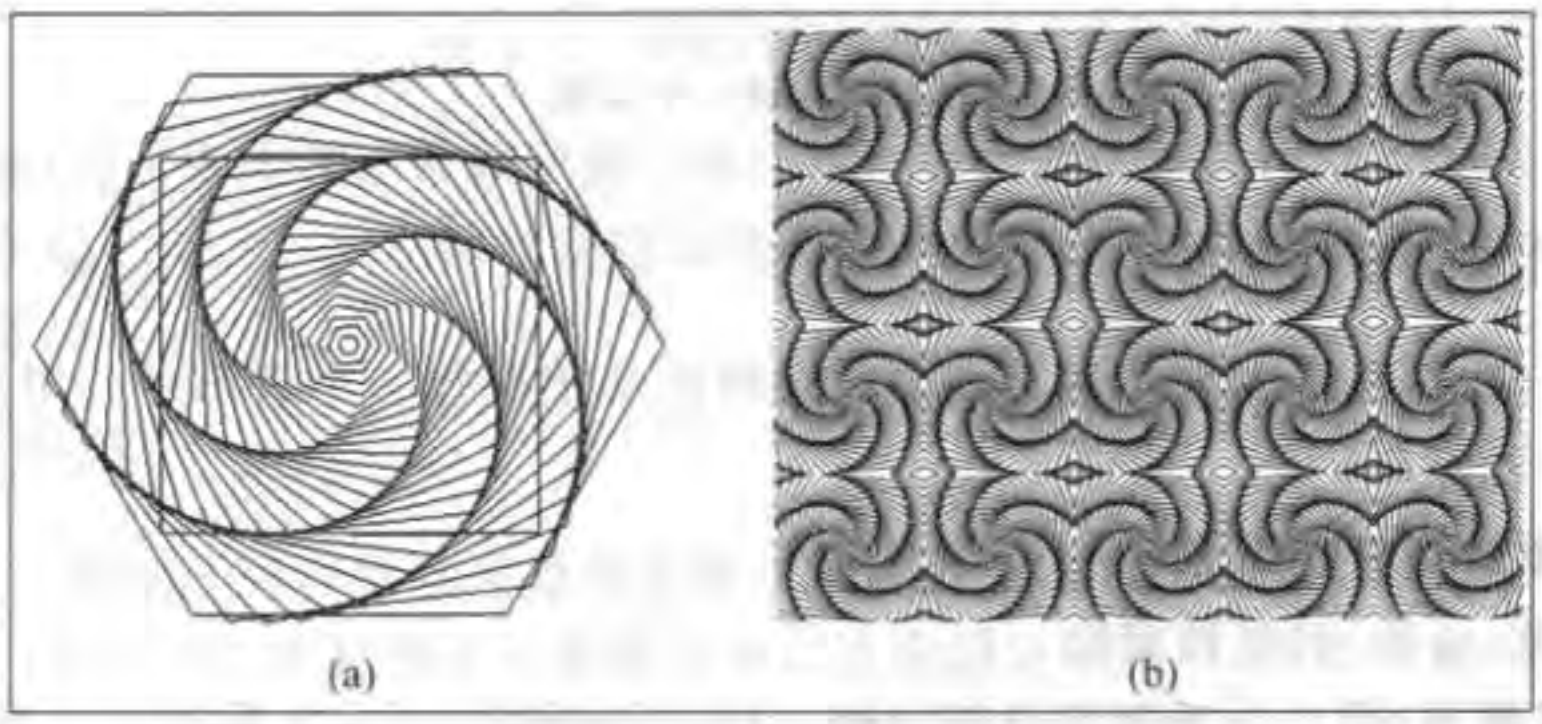


图 3.15

(a) 固定窗口中的六边形漩涡；(b) 用多个视口形成的平铺

```
            setViewport(i * L, L+i * L, j * L, L+j * L);    // 下一个视口
            hexSwirl();
        }
    }
```

3.3 裁 减 线

裁减是图形学中的一个基本任务，用于保持对象在给定区域外的部分不被绘制出来。目前已经开发出了很多裁减算法。在 OpenGL 的环境下，每个对象都会被一个特殊的算法，自动的裁减到世界窗口(第 7 章会对二维和三维的情况详细讨论)。

因为 OpenGL 为用户进行了裁减，这将可能引诱我们跳过对裁减的学习。但裁减器所使用的想法，是非常基本的并出现在很多地方。在后面的章节里我们会看到各种用于裁减的方法。下面我们将讨论一个经典的线裁剪算法，即 Cohen-Sutherland 裁减器。

3.3.1 如何裁减一条线

Cohen－Sutherland 裁减器计算端点为 p1 和 p2 的线段的哪一部分(如果有的话)位于世界窗口之内，并返回那个部分的端点。

我们将开发一个函数：int clipSegment(Point2& p1, Point2& p2, RealRect window)，它接受两个二维的点，以及一个对齐的矩形。参数 p1、p2 的类型是 Point2(见附录 2)。这个函数将以 p1、p2 为端点的线段裁减到窗口边界处。如果线段的某一部分在窗口内，则将新的端点存放在 p1、p2 中，并且函数返回 1，说明线段的某些部分是可见的。注意到 p1、p2 是传的引用，所以新的端点在函数返回后仍然有效。如果这条线完全被裁减掉了，函数返回 0，说明没有任何部分可见。

图 3.16 展示了一个典型的裁减器情况，涵盖了一些可能的裁剪行为。函数 clipSegment()对每条线段执行下列事件中的一件。

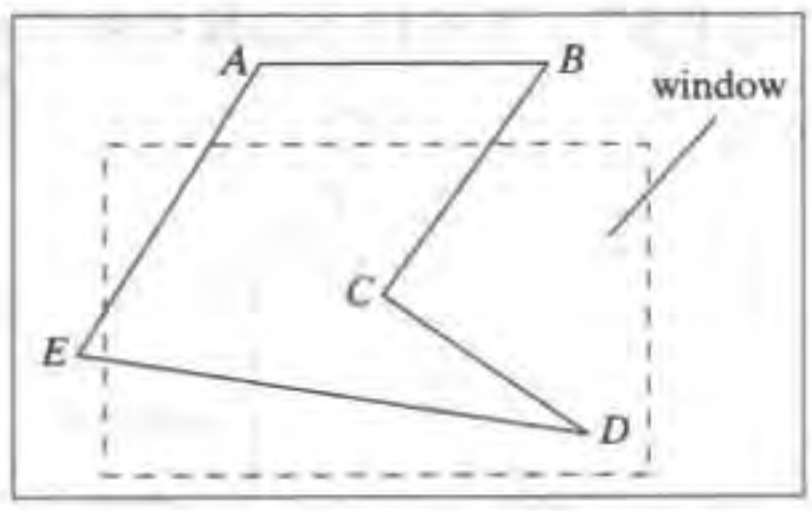


图 3.16　用窗口边界裁剪线

1. 如果整条线段都在窗口内(例如线段 CD)，函数返回 1。

2. 如果整条线段在窗口外(例如线段 AB)，函数返回 0。

3. 如果一个端点在窗口内，另外一个在窗口外(例如线段 ED)，函数将一个端点裁减，并返回 1。

4. 如果两个端点都在窗口外，但有部分线段穿过窗口(例如线段 AE)，函数将裁减两个端点，并返回 1。

对于一个窗口，有很多可能的线段排放方式。线段可以位于窗口的左边，右边，上面或者下面。还可以穿过窗口的任一个(或者两个)边界，如此等等。所以我们需要一个组织良好且有效的方法来识别主要的情况，并且计算出裁减后的端点。效率是很重要的，因为一个

通常的图画可能包含上千条线段,每条线段都必须依照窗口裁减。Cohen-Sutherland 算法针对这个问题提供了一个快速的分治算法。其他的裁减方法会在第 4 章开始的时候讨论。

3.3.2 Cohen-Sutherland 裁减算法

Cohen-Sutherland 算法在两种常见的情况下,可以快速的检测和分发。这两种情况被称为**平凡接受**和**平凡拒绝**。如图 3.17 所示,线段 AB 的两个端点都在窗口 W 内,因此 AB 整个线段都在窗口内。因此,AB 会被平凡的接受,而不需要裁减。当使用一个大的窗口时,窗口会包含大多数线段,则经常会发生这种情况。另外一方面,如果 C 和 D 这两个端点都位于 W 的某一条边外,则线段 CD 整个都在窗口外。这时 CD 是被平凡的拒绝,并且不会被绘制出来。当使用一个小的窗口,而线条很多且许多线段在窗口外时,这种情况会经常发生。

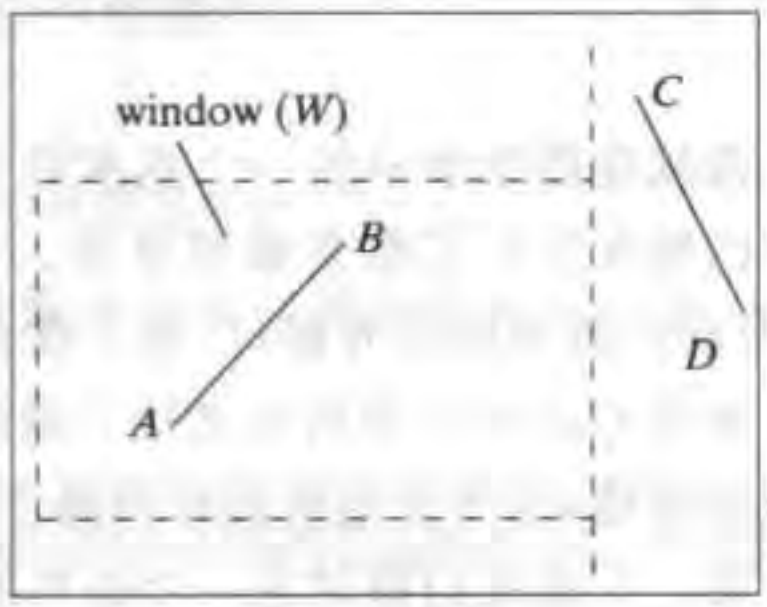


图 3.17 线段的平凡接受和平凡拒绝

检测平凡接受或者平凡拒绝

检测线段是平凡接受或者平凡拒绝的过程应该非常快速,因为它只需要检测端点的情况,而不是整条线段。我们需要一个快速的算法来检测一条线段是平凡的接受或者拒绝。为了容易地完成这个任务,我们给线段的每一个端点计算一个"窗口内部/外部编码",用于后续的测试。这种针对单个点的"内部/外部"测试是非常快的,因为窗口使用的是一个对齐的矩形。图 3.18 展示了这个计算过程。点 P 在窗口 W 的左上部分。一个点与窗口边界的相对位置有四种情况(在上面且在左边,在上面且在右边,等等),这些相对关系被记录在 P 的一个有四个元素的码字里:第一个元素是 T,说明是"在左边",而第二个元素是 T,说明 P"在上面"。另外两个字段中的 F 表示"在右边"和"在下面"。

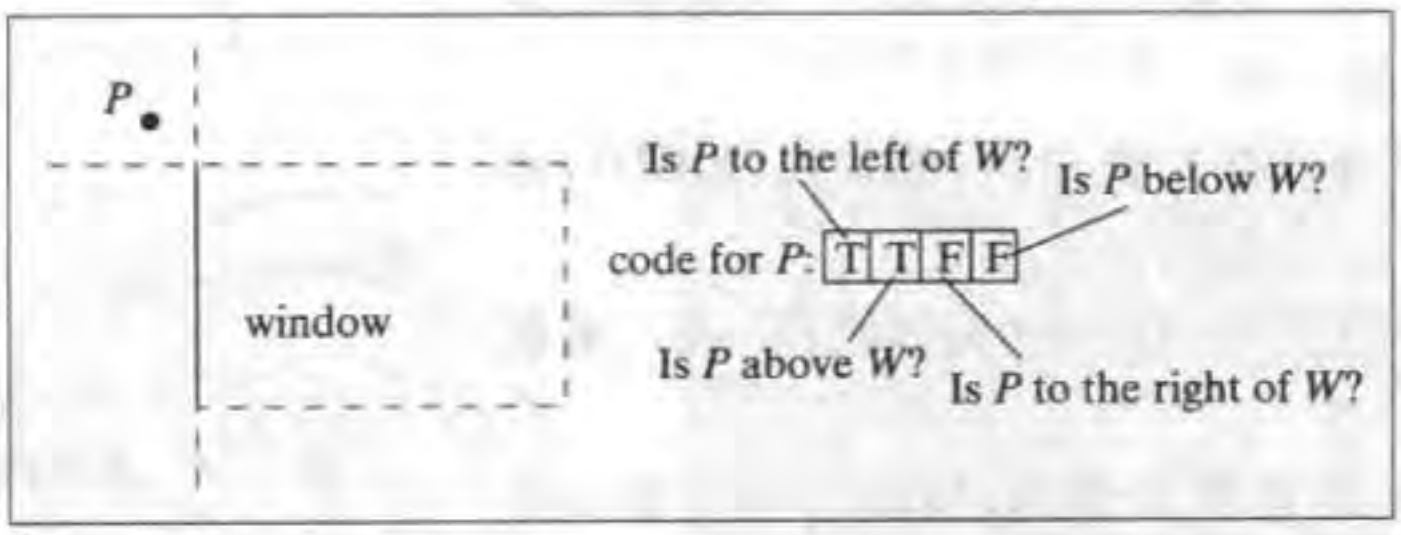


图 3.18 根据 P 在窗口的位置,对 P 做编码

例如,如果 P 在窗口里面,则其码字是 FFFF;如果 P 在下面但不在左边也不在右边,则其码字是 FFFT。图 3.19 显示了这 9 种可能的区域,及其对应的码子。

我们为所检测线段的每一个端点都计算出一个码字,那么平凡的接受和平凡的拒绝可以很容易的由这些码字得出:

TTFF	FTFF	FTTF
TFFF	FFFF window	FFTF
TFFT	FFFT	FFTT

图 3.19　点的内部/外部编码

- 平凡接受：两个码字都是 FFFF。
- 平凡拒绝：两个码字在某一位元素上都是 T：两个点都是在窗口左边，或者都在它上面，等等。

使用 C/C++ 的位操作能力，就可以非常有效地实现码字的计算和测试。

没有平凡接受或平凡拒绝时的截断

Cohen-Sutherland 算法使用一个分治的策略。如果线段不是被平凡接受和平凡拒绝，则它会被窗口的某一个边界分成两个部分。其中一个部分在窗口之外，这部分会被丢掉。另一部分有被看到的潜在可能性，因此整个过程将对四个窗口边界的下一个重复进行。这引发了如下策略：

```
do {
    形成 p1, p2 的码字;
    if (平凡接受) return 1;
    if (平凡拒绝) return 0;
    将线段在下一个窗口边界处截断;舍去在外面的部分;
} while (1);
```

这个算法最多四次循环就会终止，因为每次循环中，我们只保留线段在先前窗口边界测试中幸存的部分，而这里只有四个这样的边界。因此，最多四次循环，就可以确保出现平凡的接受或拒绝的情况。

现在我们看看在每条边界处，是如何进行截断的。图 3.20 显示了一个涉及窗口右边界的例子。

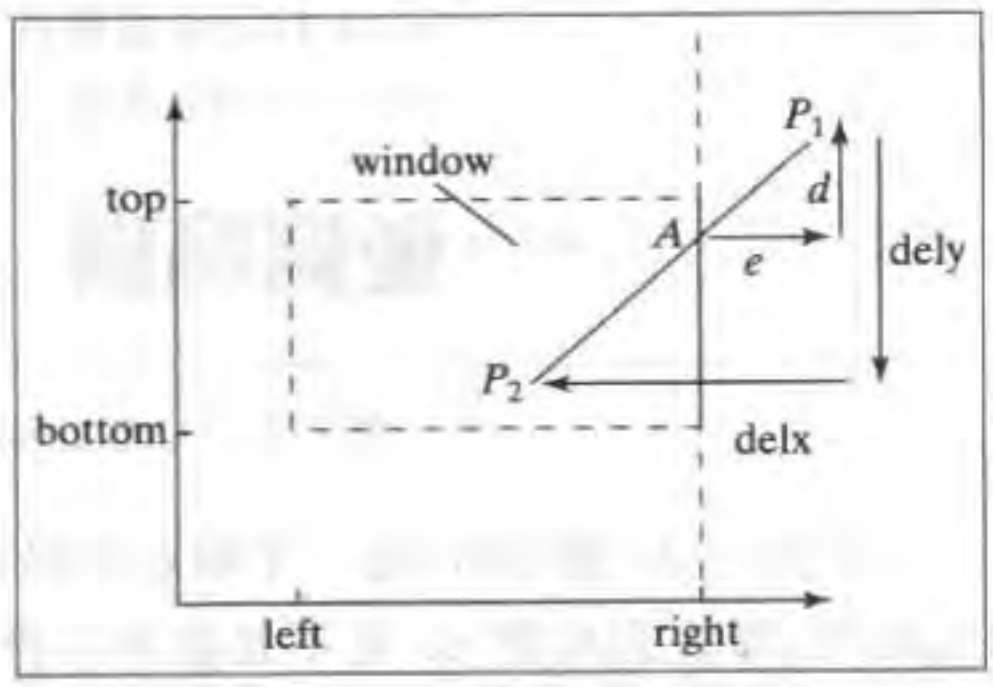


图 3.20　在边界处裁剪线段

点 p1 的坐标需要被重新计算。它的 x 坐标显然是 W. right，即窗口的右边界。它的 y 坐标需要用图中 d 的值调整 p1. y 得到。由三角形的相似性

$$\frac{d}{dely}=\frac{e}{delx}$$

其中 e 是 p1. x－W. right，且

$$\text{delx} = \text{p2. x} - \text{p1. x};$$
$$\text{dely} = \text{p2. y} - \text{p1. y}; \tag{3.4}$$

是这两个端点坐标的差。由此，很容易确定 d，而在原来 p1. y 上加上一个增量就可以得到

新的 p1.y:

$$p1.y += (W.right - p1.x) * dely/delx \tag{3.5}$$

可以用类似的方法对窗口其他三个边界进行裁剪。在有些计算中会出现 dely/delx,另一些时候是 delx/dely。我们必须考虑到除以0的情况,事实上,对于垂直的线,delx 是0,而对于水平线,dely 是0。但正如练习中讨论的,当分母是0的时候,这行危险的代码不会被执行到,因此除以0的情况是不会发生的。

图3.21的 clipSegment()函数集合了前面所有的想法。线段的端点是通过引用传进去的,因此在调用函数时,函数对端点的修改在函数外也是可见的。其中 Point2 存放一个二维的点,RealRect 存放一个对齐的矩形。在3.4节会详细讨论这两个类型。

```
int clipSegment(Poitn2& p1, Point2& p2, RealRect W)
{
    do {
        if (平凡接受) return 1;   //部分可见
        if (平凡拒绝) return 0;   //完全不可见
        if (p1 在窗口外面)
        {
            if (p1 在窗口左边) 用左边界截断,更新 p1;
            else if (p1 在窗口右边) 用右边界截断,更新 p1;
            else if (p1 在窗口下面) 用下边界截断,更新 p1;
            else if (p1 在窗口上面) 用上边界截断,更新 p1;
        }
        else   //p2 在窗口外面
        {
            if (p2 在窗口左边) 用左边界截断,更新 p2;
            else if (p2 在窗口右边) 用右边界截断,更新 p2;
            else if (p2 在窗口下面) 用下边界截断,更新 p2;
            else if (p2 在窗口上面) 用上边界截断,更新 p2;
        }
    } while (1);
}
```

图3.21 Cohen-Sutherland 裁减器的伪代码

每次执行 do 循环时,每一个端点的码字会被重新计算和测试。当平凡接受和拒绝的检测失败时,算法会检测 p1 是不是在窗口外,如果是,则将这个端点裁减到窗口边界。如果 p1 在窗口内,则 p2 必然在窗口外,因此 p2 会被裁减到窗口边界。

这个版本的算法按左,右,下,上的顺序裁减。如果线段等概率的位于世界窗口的任意地方,则选择裁减顺序就无关紧要了。图3.22展示了一个需要全部4次裁减的情况。第一次裁减将 p1 改到 A;第二次将 p2 改为 B;第三次发现 p1 仍在窗口的下面,因此将 A 改为 C。最后一次将 p2 改为 D。不管如何选择裁减顺序,总有一种情况需要全部四次裁减。

裁减是一个基本操作,近年来受到了广泛关注,也设计出来了很多其他的方法。

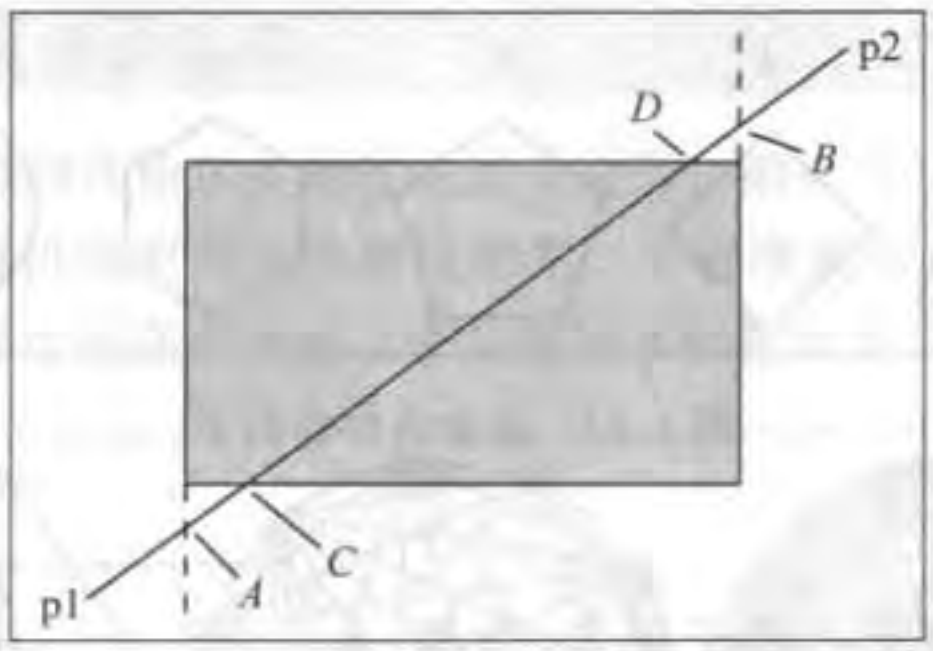


图 3.22　需要四次裁减的线段

练习

练习 3.3.1　手工模拟 clipSegment()

对于一个给定的窗口(left,right,bottom,top)=(30,220,50,240),和下面的线段,手工执行裁剪函数:

1. $P_1=(40,140)$, $P_2=(100,200)$;
2. $P_1=(10,270)$, $P_2=(300,0)$;
3. $P_1=(20,10)$, $P_2=(20,200)$;
4. $P_1=(0,0)$, $P_2=(250,250)$;

在每一个例子中,确定裁减后线段的端点,并写通过在图纸上绘制草图,来做视觉上的验证。

练习 3.3.2　潜在的除以零的情况

说明当等式(3.4)中的分母为 0 时,分子也是 0。进一步的说明,除以 0 的这种可能的危险操作永远不会发生。

3.4　正多边形、圆和圆弧

3.4.1　正多边形

图形学中经常绘制的一种形状是有 n 条边的正多边形。一个多边形是"正"的,如果它是简单的,而且所有的边相等,所有的内角都相等。正如第一章所述,如果一个多边形没有两条边互相交叉,则它是简单的。更精确的,只有相邻的边能相互接触,而且只在它们的端点处接触。我们称有 n 条边的正多边形为正 n 边形。常见的例子是正四边形(正方形),正八边形,等等。正三边形是一个等边三角形。图 3.23 显示了各种例子。

如果正 n 边形有很多条边,则这个正 n 边形看上去会像一个圆。事实上,这将在后面用于实现一种绘制圆的方法。正 n 边形的 n 个顶点位于一个圆上,即所谓的正 n 边形的外接圆,而且他们的位置很容易被计算。图 3.24 是一个六边形的例子:圆周上每隔 60°有一个

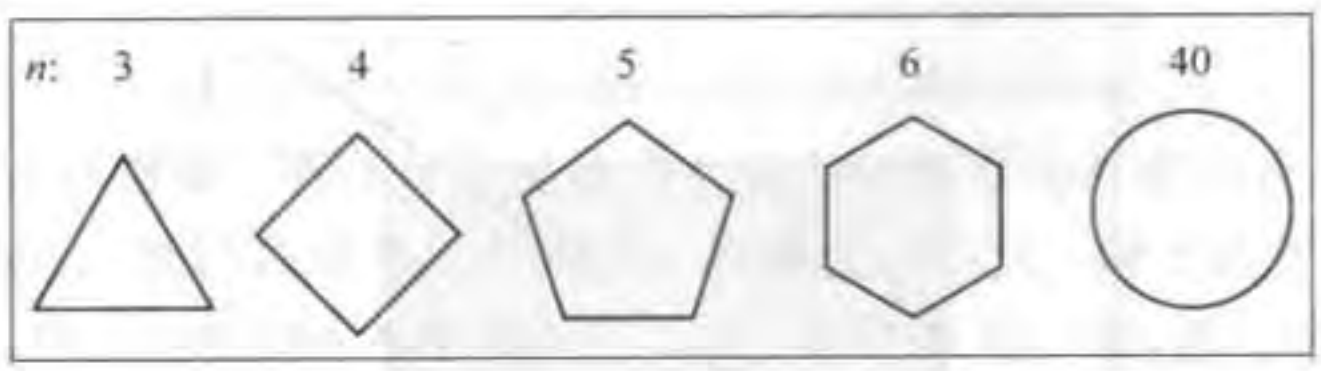


图 3.23 正多边形的例子

顶点,它们等距离的排放。

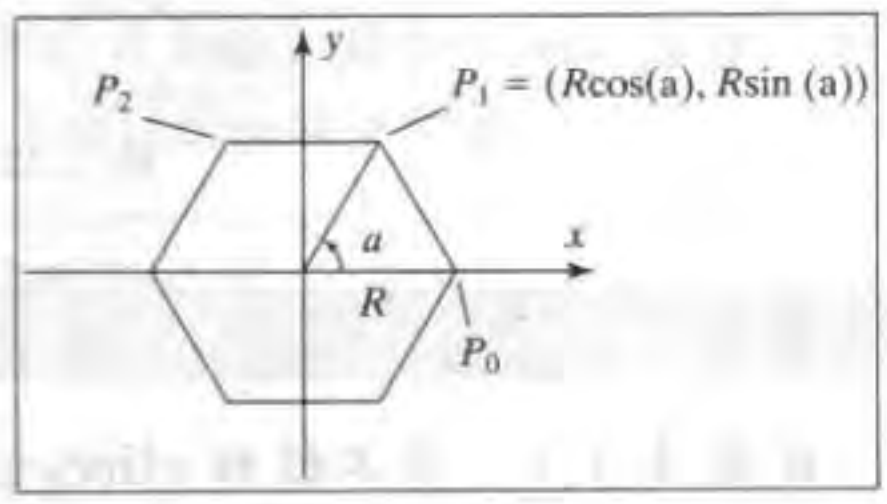


图 3.24 正 6 边形的顶点

半径为 R 的外接圆的圆心位于原点,而第一个点 P_0 位于 x 轴的正方向。其他的顶点位于下面等式指定的位置:$P_i=(\cos(ia),\sin(ia))$,$i=1,\cdots,5$,其中 $a=\pi/3$,sin 和 cos 就是我们熟悉的正弦函数与余弦函数。对这个等式稍作修改,就可以处理中心点在(cx, cy)的情况。我们只需要在 x 和 y 的坐标上分别增加 cx 和 cy 就行了。要将正六边形缩放 S 倍,只需将 R 乘上 S。要将其旋转角度 A,只需在 cos 和 sin 函数的参数中增加 A 就行了。更一般的几何变换方法将在第 5 章详细讨论。

总结一下,一个绘制圆 S 的算法是绘制一个有很多边的正 n 边形。这可以通过计算外接圆内的 n 个点的位置来完成,执行 moveTo() 到其中一个点,然后再执行 lineTo() 到其他每一个点。这可以绘制出一个可以接受的圆(如果 n 足够大),但这是一个非常低效的方法,因为它需要计算很多遍 sin() 和 cos()。在一篇 Jim Blinn 的文章"How Many Ways Can You Draw a Circle?"【Blinn96】中,描述了一类更有效的绘制圆的算法。

3.4.2 正 n 边形的变种

基于正 n 边形顶点,可以绘制有趣的变种。如图 3.25 所示,n 边形的顶点可以用多种方式连接,从而得到各种图形。图 3.25(a)中通过连接相邻定点,形成标准正 n 边形,图 3.25(b)通过连接相隔的顶点得到了一个**星形**,图 3.25(c)通过将每一个顶点和其他所有顶点相连接,形成了一个有趣的**花环**。下面我们就讨论这个花环。其他的图形将在练习中描述。

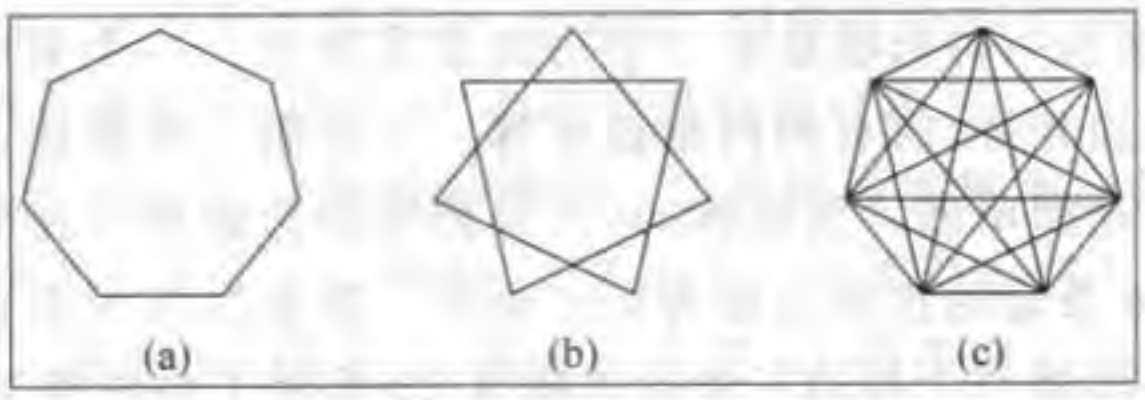


图 3.25

(a) 正 7 边形;(b) 一个星形;(c) 一个 7 花环

例 3.4.1　花环和黄金 5 花环

花环是一个每个顶点都和其他顶点相连的正 n 边形。图 3.26 展示了 5、11 和 17 的花环。花环有时会被用作一个计算机图形设备的测试图案。它有序的形状足以揭示任何扭曲，而通过观察相会于每个顶点的那组线条所呈现出来的拥挤和模糊程度，可以确定设备的分辨率。

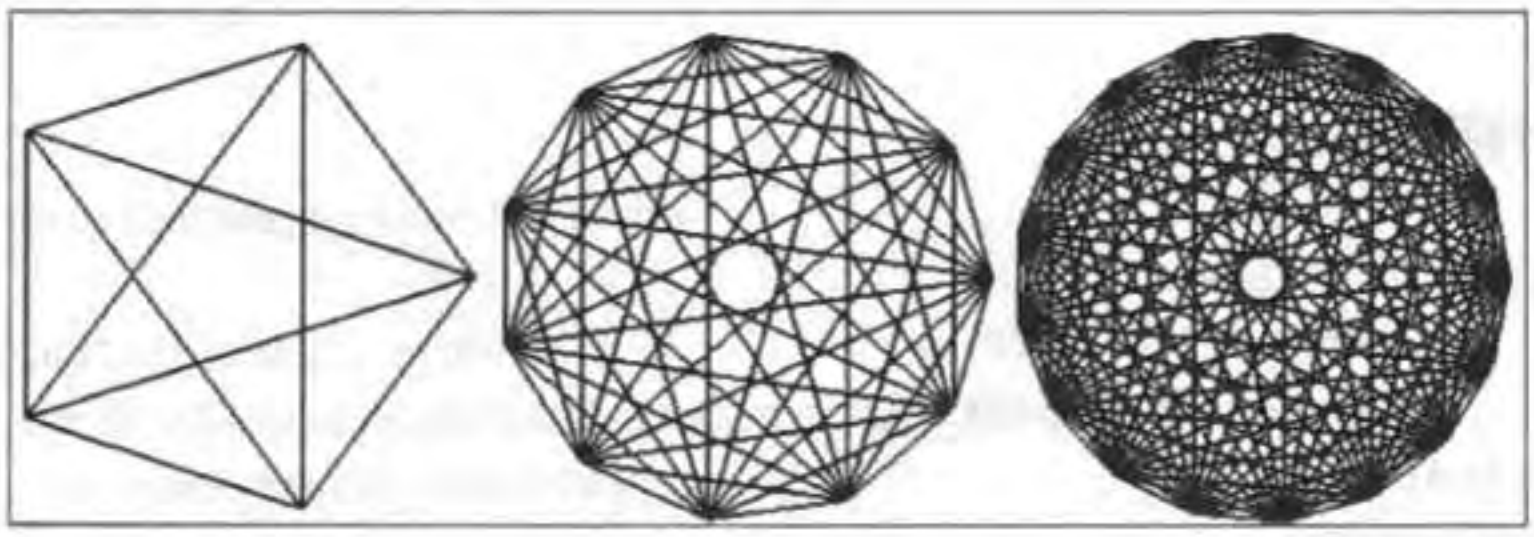


图 3.26　5、11 和 17 花环

绘制花环很容易：只要将每一个点和其他顶点相连。绘制花环的完整代码如图 3.27 所示：

```
#include <windows.h>
#include <gl/GL.h>
#include <gl/GLU.h>
#include <gl/GLUT.h>
#include <iostream.h>
#include <math.h>

//一个基于正边行绘制花环的程序

class GLintPoint
{
public:
    GLint x, y;
};

//point 2 类
class Point2
{
public:
    float x, y;
    void set(float dx, float dy) {x=dx; y=dy;}
    void set(Point2& p) {x=p.x; y=p.y;}
    Point2(float xx, float yy) {x=xx, y=yy;}
    Point2() {x=y=0;}
};
```

图 3.27　一个绘制花环的完整程序

```
Point2 currPos;
Point2 CP;

void moveTo(Point2 p)
{
    CP.set(p);
}

void lineTo(Point2 p)
{
    glBegin(GL_LINES);
        glVertex2f(CP.x, CP.y);
        glVertex2f(p.x, p.y);
    glEnd();
    glFlush(0);
    CP.set(p);
}

void myInit(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glClearColor(1.0, 0.0, 0.0, 0.0);
    //背景是红的
    glColor3f(0.0, 0.0, 1.0);
    //用蓝色绘制
}

void rosette(int N, float radius)
{
    Point2 *pointlist=new Point2[N];
    GLfloat theta=(2.0f * 3.1415926536) / N;
    for (int c=0; c<N; c++)
    {
        pointlist[c].set(radius*sin(theta*c), radius*cos(theta*c));
    }
    for (int i=0; i<N; i++)
    {
        for (int j=0; j<N; j++)
        {
            moveTo(pointlist[i]);
            lineTo(pointlist[j]);
        }
    }
    }
}
```

图 3.27（续）

```
void render()
{
    //显示的回调函数
    glClear(GL_COLOR_BUFFER_BIT);
    glViewport(10, 10, 640, 480);
    rosette(5, .66f);
    glFlush();
}

void main(int argc, char ** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glMatrixModel(GL_PROJECTION);
    glLoadIdentity();
    glutInitWindowSize(640, 480);
    glutCreateWindow("Rosette");
    glutDisplayFunc(render);     //注册显示的回调函数
    myInit();
    glutMainLoop();
}
```

图 3.27（续）

5 花环特别的有趣，因为它体现了很多黄金分割率 ϕ（参见第 2 章）的实例。图 3.28(a) 显示了一个 5 花环，它由一个外五边形和一个内五角星构成。希腊人从这张图中看到了神秘的意义。它的线段有一种有趣的关系：每一个线段要比短于它的线段长 ϕ 倍。而且由于五角星的边又构成了一个内五边形，因此可以实现一个无穷的嵌套，就如 3.28(b) 所示的那样。

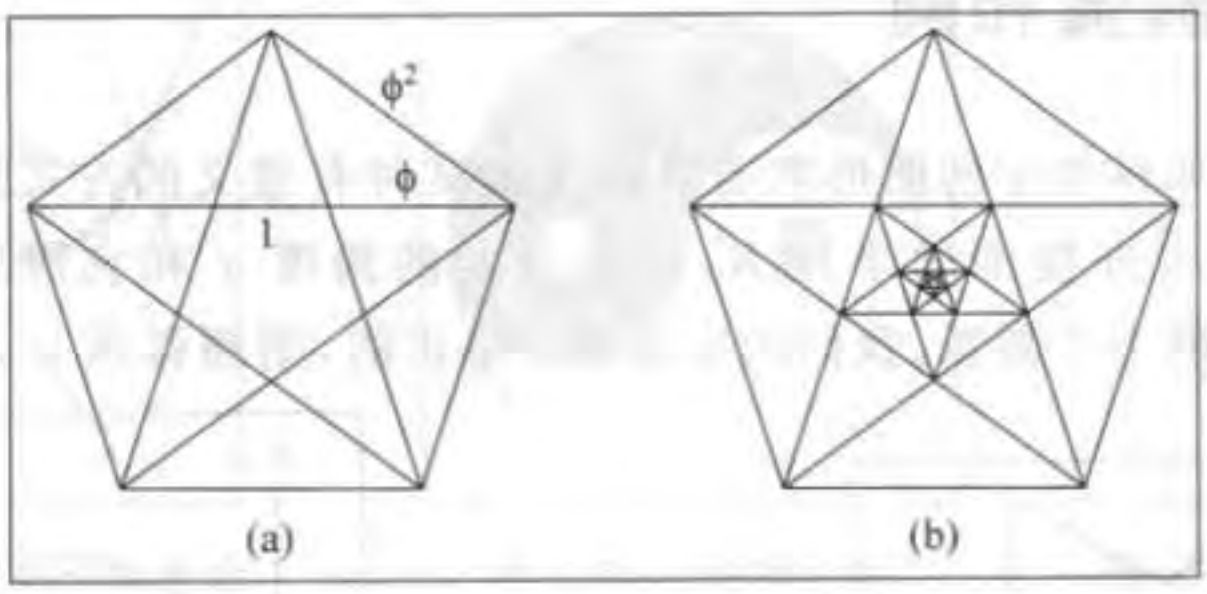


图 3.28　5 花环和它的无穷嵌套——五边形和五角星

例 3.4.2　基于两个同心多边形的图形

图 3.29 显示了一些由两个同心圆建造的图像（它们的主要目的是带来视觉效果），外半径是 R，内半径是 fR，f 是某个因子。每一个图都使用了一个正 n 边形的变种，其半径在内径和外径之间交替。图 3.29(a)、图 3.29(b) 展示了大家所熟悉的，基于 6 边形和 10 边形的

公司标志。图 3.29(c)是基于 14 边形,而图 3.29(d)明确的显示了内圆。

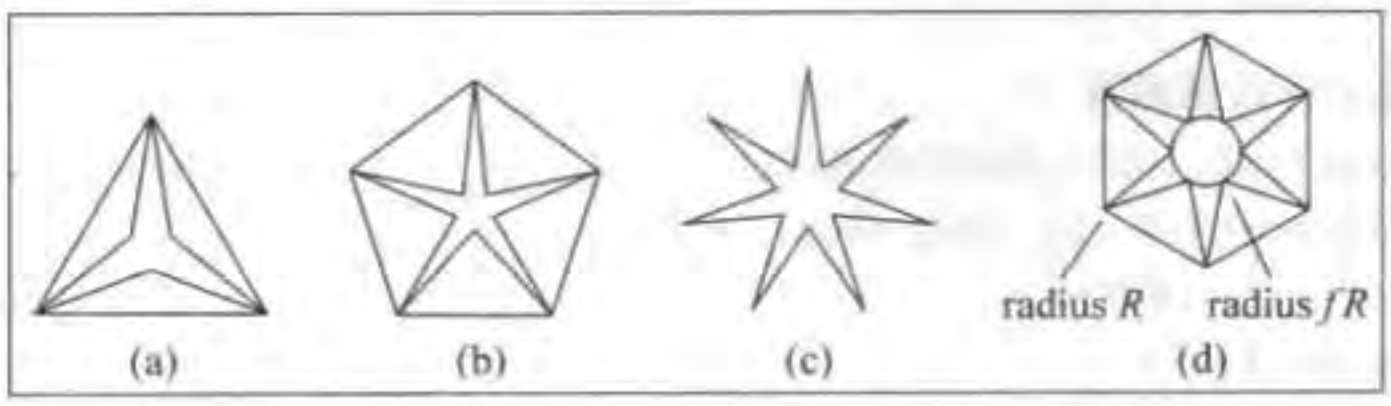


图 3.29 基于正 n 边形的有趣的图形角星

练习

练习 3.4.1 星形和花环

将正五边形每隔一个点相连,就可以得到一个五角星。将其推广到任意一个 n 是奇数的正 n 边形,并开发一个函数可以绘制这种所谓的星形。可以通过一个初始的 moveTo(),后面跟一系列 lineTo()完成这个函数码?如果 n 是偶数的时候,是怎样的情况?

练习 3.4.2 五角星中的几何学

证明 5 花环中每一个线段的长度,和下一条较短线段的长度的比值是黄金分割比率。解决问题的一个方法是证明五角星中的三角形都是黄金三角形(参见第 2 章),它的某一个内角是 $\pi/5$。证明 $2*\cos(\pi/5)=\phi$ 且 $2*\cos(2\pi/5)=1/\phi$。另一种方法是使用五角星中的两组相似三角形,并且利用 ϕ 满足 $\phi^3=2\phi+1$ 的关系。

练习 3.4.3 绘制一个著名的标志图

图 3.30 中的标志包括一个图片的三个实例,相互旋转了一定角度。编写一个函数来绘制这个形状。

3.4.3 绘制圆弧和圆

许多艺术、建筑和科学中的图形需要将圆弧按某种有意义的方式放置。一个圆弧可以方便的通过其中心 c,外接圆的半径 R,以及开始的角度 a 和它展开的角度 b 来表示。图 3.31 显示了这样的一个圆弧。我们约定如果 b 是正的,则圆弧从 a 逆时针方向展开。如

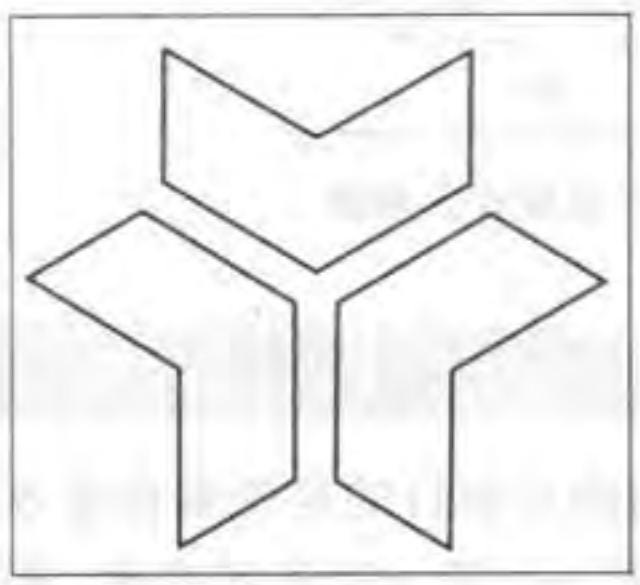


图 3.30 Massachusetts 大学的标志

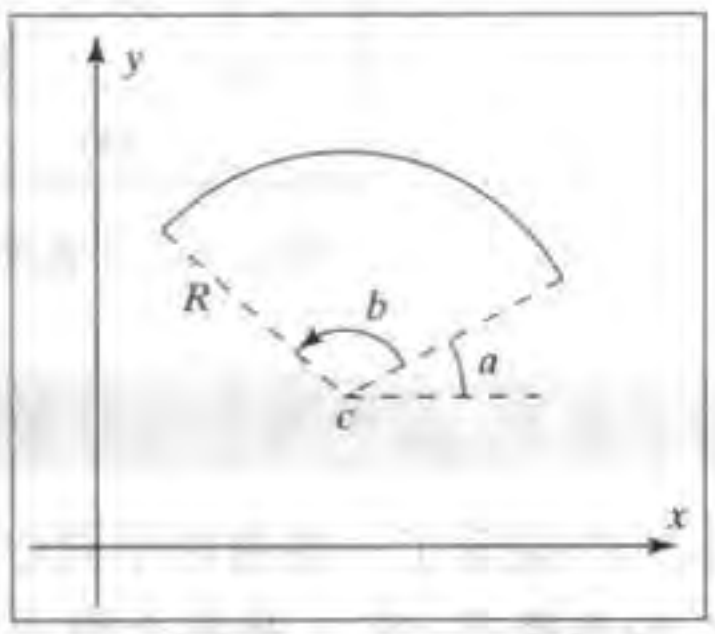


图 3.31 圆弧的定义

果 b 是负的，则按顺时针方向。圆是圆弧的一个特例，它展开的角度是 360°。

我们可以写一个程序 drawCirlce()，它通过指定圆心和半径来绘制。还有一些其他方法来描述圆，这些方式在交互式图形学和计算机辅助设计中有重要的应用。两个常见的方式是：

1. **已知圆心以及一个圆上的点**。这里，只需要知道半径，就可以调用画圆的函数了。如果 c 是圆心，p 是圆上的点，那么半径就是 c 到 p 的距离，利用勾股定理就可以计算这个距离。

2. **给定圆必须经过的三个点**。我们都知道，三个不共线的点可以唯一确定一个圆。第 4 章会讨论如何寻找这种圆的圆心和半径。

例 3.4.3　将圆弧融合在一起

使用两个互相相切的圆的一部分，可以得到更复杂的形状。图 3.32 说明了这个基本原理。两个圆在 A 点处相切，切线是 L。正因为如此，由粗线所示的两个圆弧在 A 点紧密地融合在一起，没有任何视觉上的裂痕或者棱角。类似的，圆弧可以和任意一个切线融合，例如在 B 点。

练习

练习 3.4.4　哲学中的圆圈

在中国哲学和宗教中，存在阴阳两种法则，它们相互作用以影响所有人的命运。图 3.33 显示了精致的阴阳标志。黑色部分是阴，代表阴性的，而亮的部分是阳，代表阳性。假定它在某一个坐标系中居中，请详细描述这个几何符号的细节。

练习 3.4.5　一个著名的标志图

图 3.34 显示了一个著名的汽车商标，它由一个等边三角形中的竖直的三角形所形成的。它的外三角形被两个同心圆代替。当这三个内点的合适的位置确定后，写一个函数绘制这个标志图。

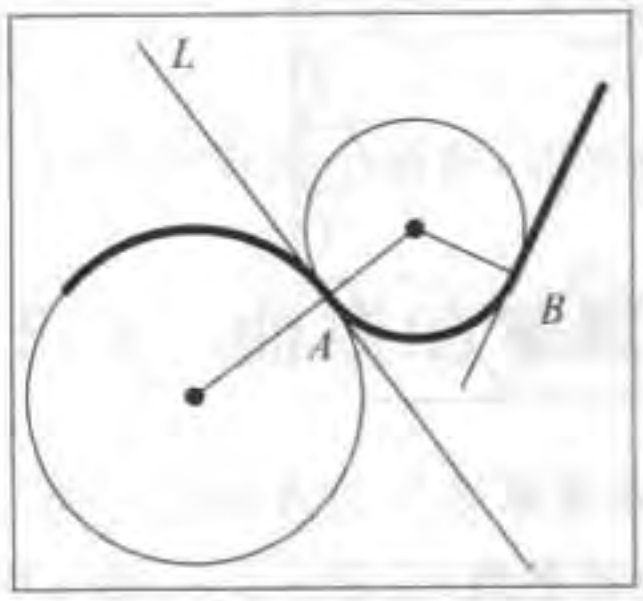


图 3.32　用切线将两个圆融合

图 3.33　阴阳符号

图 3.34　一个著名的标志图

3.4.4　曲线的逐次细化

逐次细化一个简单的曲线，可以递归形成非常复杂的曲线。最简单的例子可能就是 1904 年瑞典数学家 Helge von Koch 发现的 **Koch** 曲线。它在数学界受到了广泛的关注，因为它有无限的长度，却包含在一个有限的区域中【Gardner78】。对于那些学习计算机图形学

的人,它是一个可以生成的令人着迷的曲线,而且它非常容易用程序实现。

在这里,连续产生的 Koch 曲线记作 K_0, K_1, K_2, …。第 0 代图形是 K_0,它是一个长度为 1 的水平线段。图 3.35 是 K_1。为了得到 K_1,将 K_0 平均分为 3 部分,并且将中间部分用一个边长为 1/3 的三角形"提高"代替。显然,总长度变成了 4/3。图 3.35 还有第 2 阶的曲线 K_2,它通过将 K_1 的四条线段都"提高"一下形成。

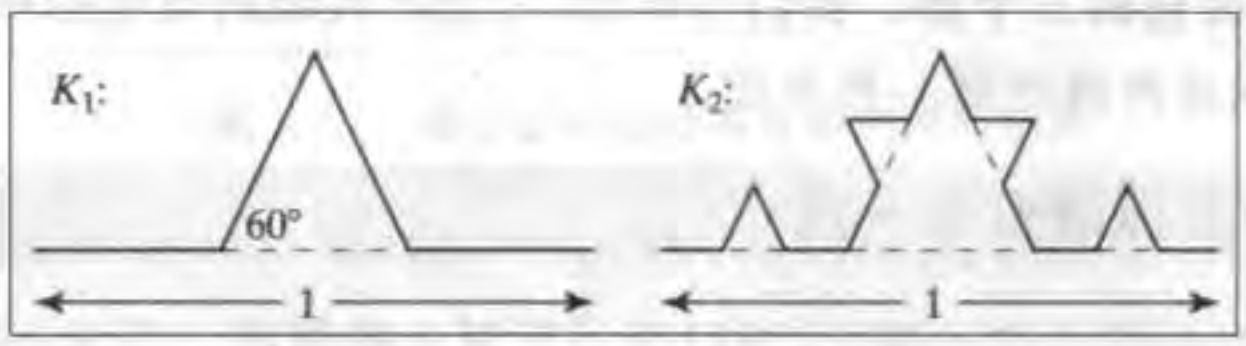


图 3.35 Koch 曲线的两代

我们可以由 K_n 得到 K_{n+1},用一个自然归纳的步骤:

将 K_n 的每一条线段平均分为 3 的段,然后将中间的那段用一个等边三角的"提高"代替。

在这个过程中,每一个线段的长度变为了原来的 4/3,所以总的曲线长度也是它上一代的 4/3。因此,K_i 的总长度是$(4/3)^i$,它随着 i 的增加而增加。随着 i 增大到无穷,曲线的长度也增大到无穷。

图 3.36 的 Koch 雪花是另外一个有趣的形状,它通过将 3 个 Koch 曲线连在一起而成。第 i 代图形 S_i 的周长是 3 倍于简单 Koch 曲线的周长,因此它是 $3(4/3)^i$,显然它也是随着 i 增大而无限增大。但 Koch 雪花中的面积的增大却非常缓慢,而事实上它是有极限的(见练习),S_∞ 的面积只是 S_0 面积的 8/5! 所以 Koch 雪花的边越来越粗糙,越来越长,但它的面积仍然是有限的。图 3.37 显示了第 5 代的 Koch 雪花。

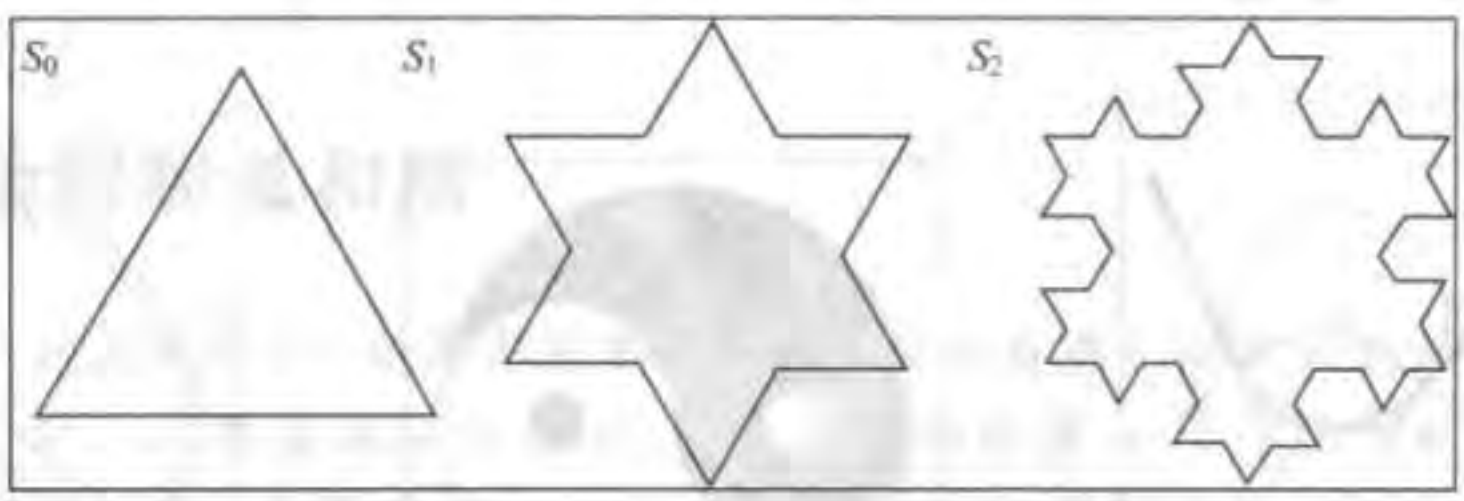


图 3.36 Koch 雪花的前几代

自相似

Koch 曲线在下面的意义下是**自相似**的。在 K_n 的某一部分放置一个小的窗口,观察它粗糙的形状。然后再将窗口缩小 10 亿倍,观察放大后的曲线。如果 n 足够大,那么这部分的曲线仍然会和原来的曲线有着类似的形状和粗糙程度。这一小部分可以再被放大 10 亿倍,但它的形状仍然是一样的。只有极限情况下的 K_∞ 是真正的自相似,但这种自相似的效果可以在一个图形学的设置中被近似。只要将 n 变得

图 3.37 Koch 雪花的第五代 S_5

足够大，使得即使在最大程度上放大时，最小的线段的长度也比两个像素之间的距离要小就行。

3.5　曲线的参数形式

描述曲线的形状主要有两种方法：隐式形式和参数形式。**隐式形式**用函数 $F(x,y)$ 来描述曲线，并且提供 x 和 y 坐标的关系：当且仅当 (x, y) 满足下式时，(x, y) 在曲线上：

$$F(x,y)=0 \quad (x, y) \text{ 在曲线上的条件} \tag{3.6}$$

例如，一条穿过点 A 和点 B 的直线，有隐式形式：

$$F(x,y)=(y-A_y)(B_x-A_x)-(x-A_x)(B_y-A_y) \tag{3.7}$$

一个圆心在原点，半径是 R 的圆，有隐式形式：

$$F(x,y)=x^2+y^2-R^2 \tag{3.8}$$

使用隐式形式的一个好处是，你可以非常容易的判断一个点是否在曲线上：只要在有疑问的点上计算 $F(x, y)$ 就行了。有一类曲线，曲线的内部和外部是有意义的，在这种情况下，$F(x,y)$ 也被叫**做内部-外部函数**。其意义是：

$$\begin{aligned} &F(x,y)=0 \text{ 对所有在曲线上的 } (x, y) \\ &F(x,y)>0 \text{ 对所有在曲线外的 } (x, y) \\ &F(x,y)<0 \text{ 对所有在曲线内的 } (x, y) \end{aligned} \tag{3.9}$$

有一些曲线对 x 来说是**单值的**。如果 $g(x)$ 是单值的，那对于每一个 x，都只有一个函数值。事实上，只有单值的函数是“合法”的函数。对于这些曲线，函数的隐式形式可以写成 $F(x,y)=y-g(x)$。另外一些函数对 y 来说是单值的，因此存在函数 $h()$，使得曲线上所有点满足 $x=h(y)$。还有一些函数对 x 和 y 来说都不是单值的：$F(x, y)=0$ 不能写成 $y=g(x)$ 或者 $x=h(y)$ 的形式。例如圆，可以表示成：

$$y=\pm\sqrt{R^2-x^2} \tag{3.10}$$

但这里有两个函数，而不是一个。其中一个函数用正号，另一个函数用负号。

3.5.1　曲线的参数形式

曲线的参数形式在参数取不同值的时候，会产生出曲线上不同的点。参数形式可以用于更广的一类曲线，因此推荐使用这种形式，特别是当人们想绘制或者分析曲线的时候。参数形式使人想起一个点随着时间运动，我们可以把它转换成一支笔绘制曲线的运动。粒子沿曲线运动的路径由两个函数 $x()$ 和 $y()$ 确定，如果是三维的情况，有三个函数：$x()$、$y()$ 和 $z()$，它们确定粒子在时刻 t 的位置。参数 t 通常被看作是时间。而曲线本身就是粒子随着时间在某一个区间内的变化所经过的点。对于任一曲线，如果我们可以设计出合适的函数 $x()$ 和 $y()$ 或者 $x()$、$y()$ 和 $z()$，则他们可以简明而精确的表示出这条曲线。

图 3.38 所示的是经典的 Etch-A-Sketch[①] 设备，它提供了一个形象的二维模拟。当旋

① Etch-A-Sketch 是 Ohio Art 的商标。

转按钮时,藏在盒子内部的笔会在屏幕上划出一条可见的细线。一个按钮控制笔水平方向位置,另一个按钮控制垂直方向位置。如果根据 $x(t)$ 和 $y(t)$ 来旋转按钮,就能绘制出这个参数的曲线。复杂的曲线需要足够灵巧的手工操作。关于设计曲线和曲面的全面分析见第 10 章。

图 3.38 Etch-A-Sketch 绘制参数曲线

直线和椭圆

等式(3.8)所示的直线通过 A、B 这两个点。我们选择一个参数形式,在 $t=0$ 时访问 A 点,在 $t=1$ 时访问 B 点,从而得到:

$$x(t) = A_x + (B_x - A_x)t$$
$$y(t) = A_y + (B_y - A_y)t \tag{3.11}$$

因此,当 t 在 0~1 之间变化时,$P(t)=(x(t),\ y(t))$ 会扫过直线上在 A 与 B 之间的所有点(请仔细检查此处)。

另外一个典型的例子是**椭圆**,它是圆的一个轻微的泛化。它可以通过参数形式描述:

$$x(t) = W\cos(t)$$
$$y(t) = H\sin(t),\quad 0 \leqslant t \leqslant 2\pi \tag{3.12}$$

这里,W 是椭圆的"半宽",H 是椭圆的"半高"。椭圆的一些几何性质会在练习中探讨。当 W 和 H 相等时,椭圆就是一个半径为 W 的圆。图 3.39 显示了一个椭圆,以及它的 $x(.)$ 和 $y(.)$。

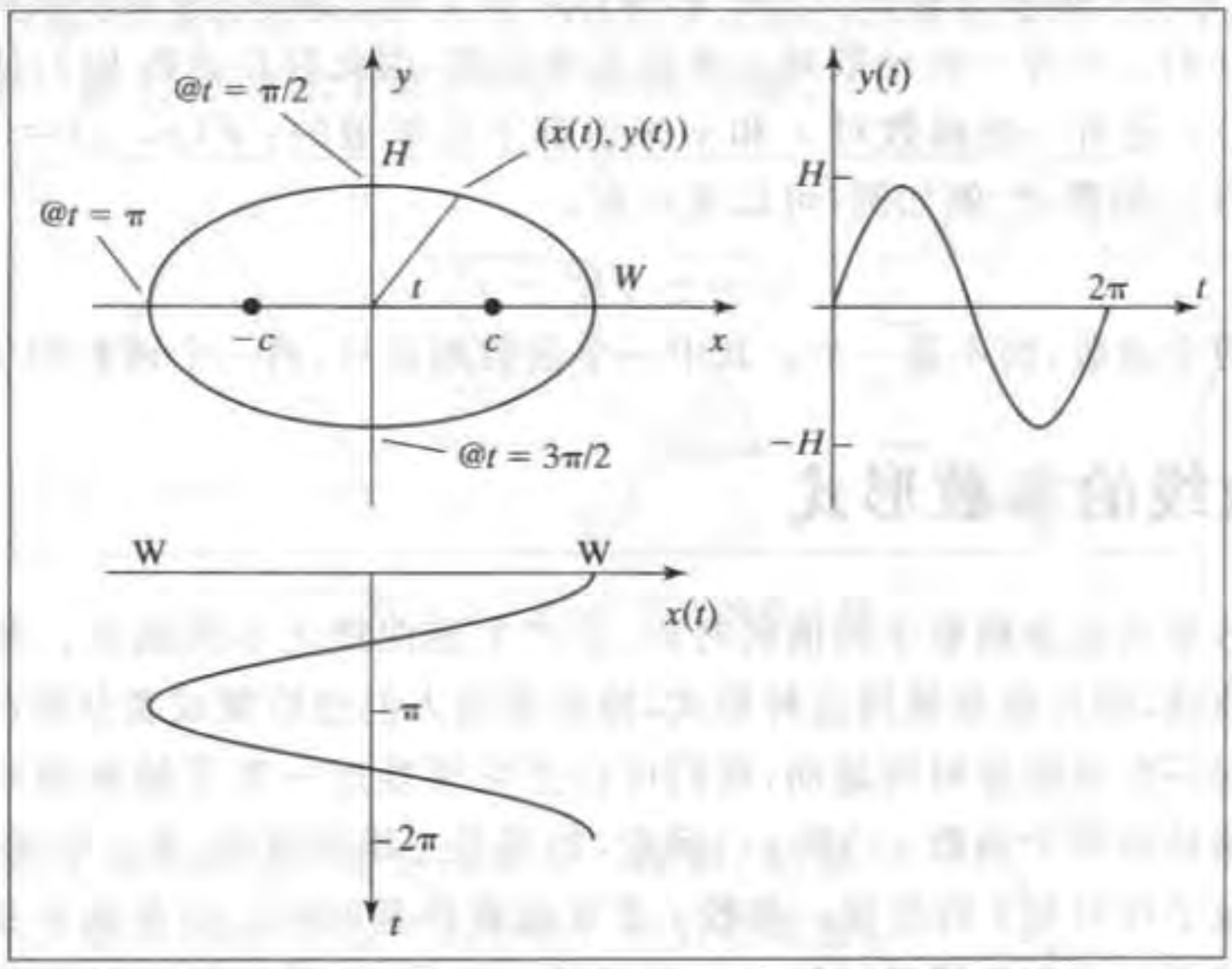


图 3.39 用参数方式表示椭圆

当 t 从 0 到 2π 变化时,点 $P(t)=(x(t),\ y(t))$ 沿着椭圆移动一周,起点(也是终点)是 $(W,\ 0)$。这个图片显示了在不同时刻点的位置。在 Etch-A-Sketch 上观察绘制一个椭圆是有用的。按照波动的方式来回转动按钮,一个模仿 $W\cos(t)$,另一个模仿 $H\sin(t)$(这个动作

很难手工完成)。

从参数形式求隐式形式——"隐式化"

假定我们想检查等式(3.12)中的参数形式是否真的表示椭圆。怎么才能通过参数形式得到隐式形式?基本的步骤是联合 $x(t)$ 和 $y(t)$ 两个方程,再设法消去变量 t。这提供了一个对任意 t 都成立的关系。进行这个步骤并不总是那么容易——没有一个适用于所有参数形式的简单的步骤。但对于这个椭圆,我们可以对 x/W 和 y/H 取平方,并使用众所周知的关系 $\cos(t)^2+\sin(t)^2=1$ 得到下面这个大家所熟悉的椭圆方程:

$$\left(\frac{x}{W}\right)^2+\left(\frac{y}{H}\right)^2=1 \tag{3.13}$$

下面的练习将探讨椭圆和其他一些经典曲线的性质。它们还指出了**圆锥曲线**的一些有用的性质,以后我们会用到这些性质。阅读下面的练习,即使你不停下来解每一道题目。

练习

练习 3.5.1　椭圆的几何特征

椭圆是到两个焦点的距离和为常数的点的集合。图 3.39 中的 $(c,0)$ 是其中一个焦点,$(-c,0)$ 是另一个焦点。证明 H、W、c 满足:$W^2=H^2+c^2$。

练习 3.5.2　离心率

椭圆的离心率,$e=c/W$,用来衡量椭圆和圆的差别。圆的离心率是 0。举个有趣的例子,太阳系中的行星都有近似圆的轨道,e 从 1/143(金星)变化 1/4(冥王星)。地球的轨道离心率是 1/60。当离心率接近于 1,那么这个椭圆就会变成一条直线。在变成直线前,e 会非常接近 1。当椭圆的 $e=0.99$ 时,它的高宽比 H/W 是多少?

练习 3.5.3　其他的圆锥曲线

椭圆是三种圆锥曲线中的一种。如图 3.40,圆锥曲线是用一个平面来切开一个圆锥时所形成的。这些圆锥曲线如下。

- 椭圆:平面沿着圆锥的等分半圆锥切开。
- 抛物线:平面平行于圆锥的边。
- 双曲线:平面切开圆锥的两个等分半平面。

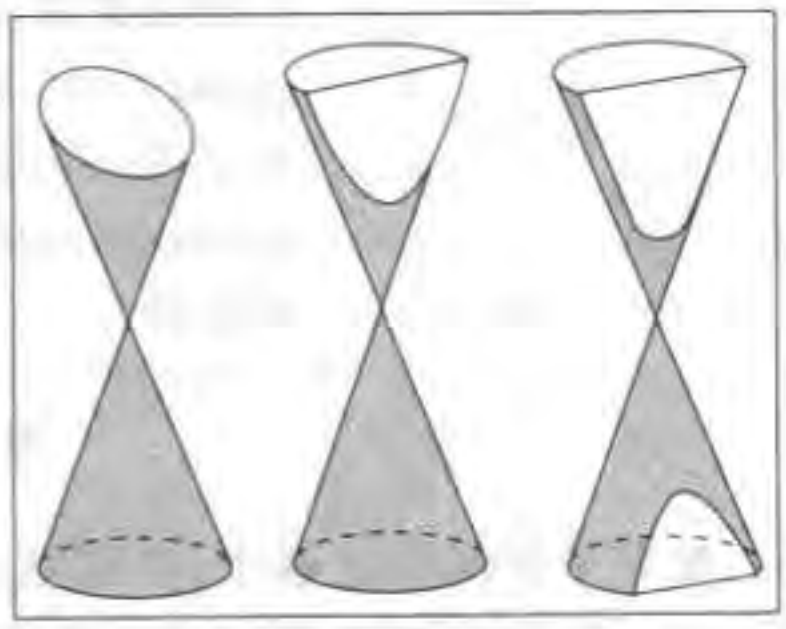


图 3.40　经典的圆锥曲线

抛物线和双曲线都有有趣且有用的几何特性。他们都有简单的隐式形式和参数形式。

证明下面的参数形式和隐式形式的表达是一致的。

- 抛物线:隐式形式为 $y^2-4ax=0$

  参数形式

$$\begin{aligned} x(t) &= at^2 \\ y(t) &= 2at \end{aligned} \tag{3.14}$$

- 双曲线:隐式形式为 $(x/a)^2-(y/b)^2=1$

  参数形式

$$x(t)=a\sec(t)$$

$$y(t) = b\tan(t) \tag{3.15}$$

用于扫描双曲线的 t 的范围是什么？注意：双曲线定义为到两个固定焦点的距离的差的值是常数的点的集合。如果两个焦点的坐标分别是$(-c, 0)$和$(+c, 0)$，证明 a、b 满足 $c^2=a^2+b^2$。

3.5.2 绘制参数曲线

当某个曲线有参数表达形式可以使用时，可以直接绘制出这条曲线。这是参数形式相对于隐式形式的主要优势。如图 3.41(a)所示，假定一个曲线 C 有参数形式 $P(t)=(x(t), y(t))$，其中 t 从 0 变化到 T。我们只需用非常紧凑的间隔采集 $P(t)$的样本，从而仅使用直线段来很好的近似曲线进行绘制。选择一个时间的序列$\{t_i\}$，对每一个 t_i，可以计算出曲线上的位置 $P_i=P(t_i)=(x(t_i), y(t_i))$。如图 3.41(b)所示，曲线 $P(t)$可以由基于点列 P_i 的折线来近似。

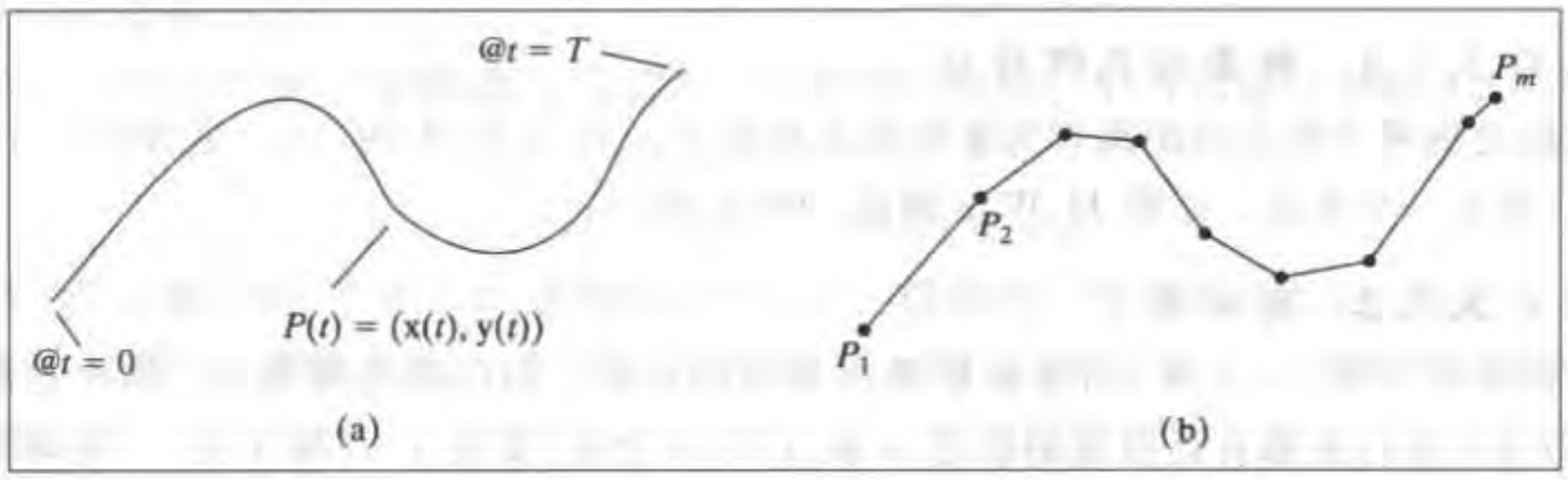


图 3.41 用折线近似曲线

图 3.42 是在已知采样时间数组 $t[i]$时，绘制曲线$(x(t), y(t))$的一段代码。

```
// 使用 t[0], ..., t[n-1] 中的采样时间,
// 绘制曲线(x(t), y(t))
glBegin(GL_LINES);
for(int i=0; i<n; i++)
glVertex2f((x(t[i]), y(t[i]));
glEnd();
```

图 3.42 用 t 中的点绘制曲线

如果采样间隔足够小，肉眼就会自然地将这些线段融合在一起，从而看到的是一条连续的曲线。在曲线变化剧烈的间隔里，应将采集点设得更密一些，而在曲线波动平滑的地方，采样点分布的更稀疏一些。采样间隔究竟要多接近，或者近似曲线的质量需求，要取决于当时的情况。

如果只是对特殊的曲线，代码通常可以简化。等式(3.14)中的椭圆可以 n 个等间距的 t 的值绘制出来，代码如下：

```
#define TWOPI 2 * 3.14159265
```

```
glBegin(GL_LINES);
    for (double t=0; t<TWOPI; t+=TWOPI/n)
        glVertex2f(W * cos(t), H * sin(t));
glEnd();
```

出于绘图的目的，参数形式绕过了隐式形式和显式形式的所有困难。曲线是可以多值的(multivalued，例如二维曲线中，一个 x 有多个与之对应的 y 在曲线上——译者注)，并且可以任意多次自交。竖直的线也不需要特殊处理：此时，$x(t)$ 在某个时间间隔中是常数。后面我们会看到在三维空间中也可以直接绘制曲线：使用了 t 的三个函数，在 t 时刻曲线上的点的坐标是 $(x(t), y(t), z(t))$。

练习

练习 3.5.4　一个样例曲线

计算并且绘制前面代码片段中所要绘制的点，其中 $W=2, H=1, t=2\pi i/9, i=0, 1, \cdots, 4$。

练习 3.5.5　绘制一个标志图

图 3.43 是一个著名的标志，它由一个同心的圆和椭圆构成。假定你有一个绘图的工具，drawEllipse(W, H, clr)，它可以绘制等式(3.14)的椭圆，并用 clr 颜色填充。假定绘制颜色时，之前的颜色会被覆盖。为这个标志图中的椭圆选择合适的尺寸，并给出绘制该图所需的命令序列。

计算机图形学中的一些特殊的曲线例子，有助于巩固本节提出的想法。

图 3.43　一个由圆和椭圆组成的标志图

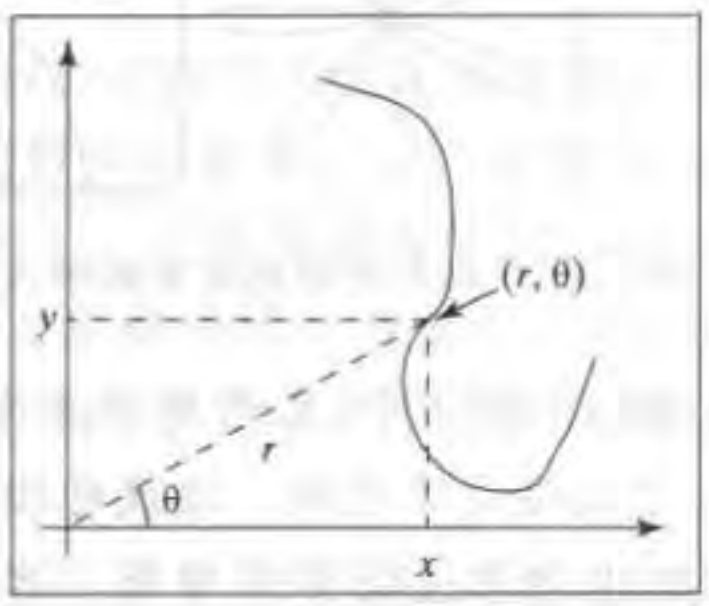


图 3.44　极坐标轴

3.5.3　极坐标形状

极坐标可以用来表示和绘制一些有趣的曲线。如图 3.44 所示，曲线上的每一点被一个角度 theta(从 x 轴的正方向开始测量)和一个径向距离 r 来表示。如果 r 和 θ 都是 t 的函数，那么当 t 变化时，$(r(t), \theta(t))$ 就会绘制出一条曲线。这个曲线也可以用笛卡儿坐标 $(x(t), y(t))$ 来表示，其中：

$$
\begin{aligned}
x(t) &= r(t)\cos(\theta(t)) \\
y(t) &= r(t)\sin(\theta(t))
\end{aligned}
\tag{3.17}
$$

但对于一大类优美的曲线,可以采用简化形式。在这些例子中,半径 r 被直接表示为 θ 的函数,而绘制曲线的参数就是 θ 本身。对于每一个点(r, θ),对应的笛卡儿坐标(x, y)满足:

$$\begin{aligned} x &= f(\theta) \cdot \cos(\theta) \\ y &= f(\theta) \cdot \sin(\theta) \end{aligned} \tag{3.18}$$

当用这种方式表达时,曲线可以被单个函数 $f()$ 完整表示出来。用极坐标形式表示的曲线,可以容易的生成和绘制出来。参数是 θ,它可以在该形状相符合的间隔变化。最简单的例子是半径为 K 的圆:$f(\theta)=K$。图 3.45 显示了一些用极坐标表示的很简单的图形。

- 心脏线:$f(\theta)=K(1+\cos(\theta))$。
- 玫瑰曲线:$f(\theta)=K\cos(n\theta)$,其中 n 指定玫瑰中的花瓣个数。后面给出了两个例子。
- 阿基米德螺线:$f(\theta)=A\theta$(见图 3.46)。参数 A 指出螺旋增长的速度(注意当 $\theta=2\pi$ 时,螺旋的半径是 $A2\pi$)。

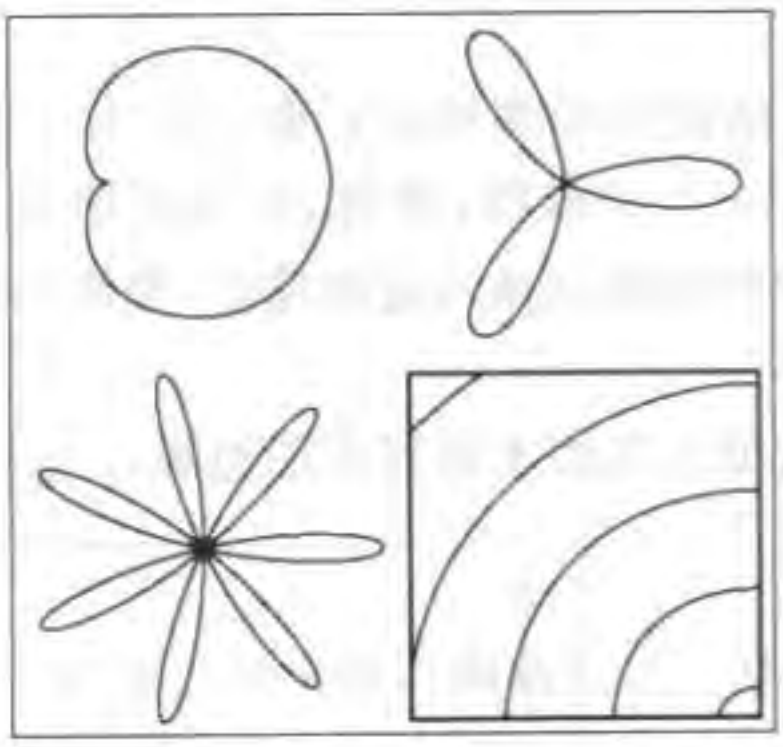


图 3.45 具有简单的极坐标形式的曲线的例子

图 3.46 一个阿基米德螺线

在每一个例子中,K 给出了曲线整体的尺寸。因为心脏线是周期的,它可以通过让 θ 从 0 到 2π 之间变化来绘制。玫瑰曲线在 n 是整数时是周期的。而当 θ 从 0 开始增长时,阿基米德螺线也会永远保持的增长。这种螺旋线被广泛的用于制造凸轮(就像汽车中的凸轮轴),以将旋转运动变为线性运动。

圆锥曲线(椭圆,抛物线和双曲线),都具有下面这种极坐标形式:

$$f(\theta) = \frac{1}{1 \pm e \cdot \cos(\theta)} \tag{3.19}$$

其中,e 是圆锥曲线的离心率。$e=1$ 时,曲线是抛物线;当 $0 \leqslant e < 1$ 时,是一个椭圆;$e>1$ 时,是一个双曲线。

对数螺旋线

对数螺旋线(或"等角螺旋线")$f(\theta)=Ke^{a\theta}$,如图 3.47(a)所示,也是一种很重要的形状【Coxeter61】。这个曲线在一个常数角度 α 处切割所有径向线,其中 $a=\cot(\alpha)$。这是唯一一种在任意的尺度变换下具有相同形状的螺旋线。将这种螺旋线进行任意放大后,再经过

适当的旋转，可以和原螺旋线重合。因此，它也被称为是自相似的。我们在以前看到过自相似这个术语，之后我们还会看到它与 Mandelbrot 集合（见附录 4）的联系。类似的，旋转等角螺旋线，会让人觉得它变大了或者变小了【Steinhaus69】[①]。

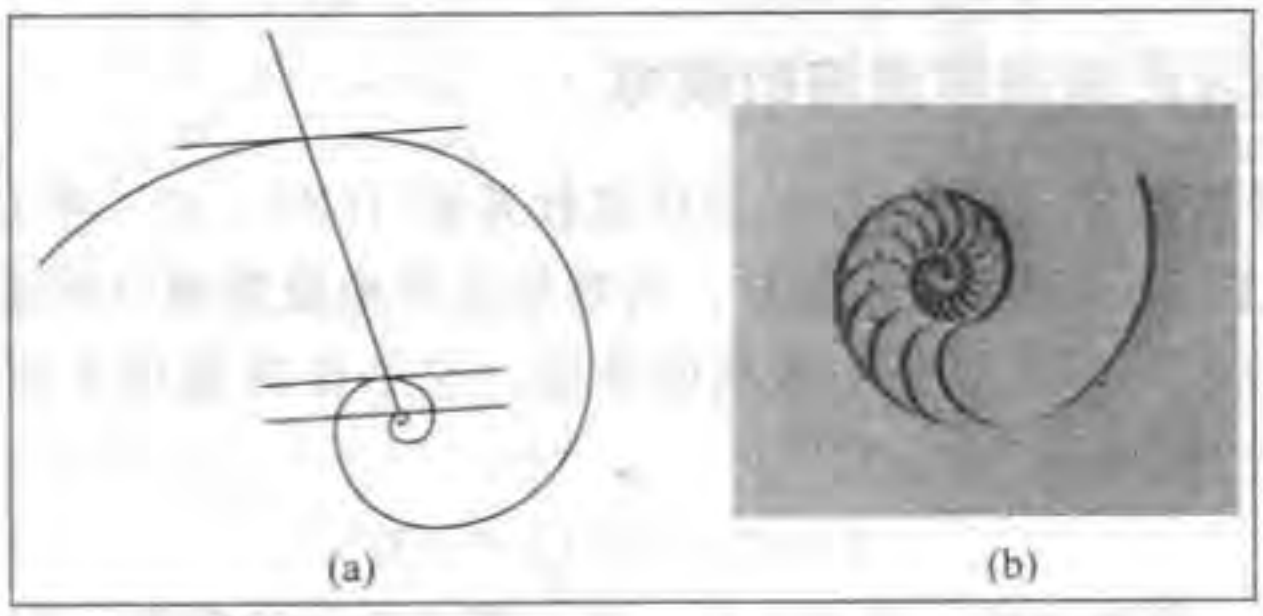


图　3.47

（a）对数螺旋线；（b）鹦鹉螺

这种保持形状的特性似乎被一些动物所采用，例如鹦鹉螺这种软体动物（见图 3.47(b)）。这种动物生长时，它的壳会沿着一条对数螺旋线增长，从而保持同样的形状【Gardner61】。做一个迷人的边注：鹦鹉螺相邻的两个腔的体积比是黄金分割率！

其他类型的曲线会在练习和案例中讨论。在【Yates46，Seggern90，Shikin95】中，可以看到一个值得关注的曲线特征的详细列表。

本章小结

在这一章里，我们开发了几个工具，它们可以使应用程序的程序员使用最方便的世界坐标系，思考并直接解决手头的问题。对象通过高精度的实数坐标系来定义和建模，而不需要考虑对象将显示在屏幕的哪个位置，以及生成的图像有多大。这些考虑都推迟到随后对窗口和视口的选择上——手工的或自动的——它们决定了对象哪些部分被显示，以及如何显示在屏幕上。这种方法将建模的阶段和视图阶段分开，允许程序员或者用户在各阶段集中解决相关问题，而不需为显示设备的细节而分心。

窗口的使用，使放大或者缩小场景，以及漫游场景的不同部分变得非常容易。从日常使用照相机的经验可以熟悉这些操作。视口的使用，使程序员能够将图像或者图像集放在显示器上的期望的位置，从而合成最终的图像。我们讨论了保证窗口和视口的纵横比不变的技术，从而避免图像变形。

我们还开发了一些附加的工具，用于创建正多边形，圆弧和圆。我们还介绍了曲线的参数形式，这是一种非常自然的描述曲线的方法。它使得绘制曲线变得很容易，即使是那些多值的，相互交叉的，或者在某些地方是垂直的曲线。

---

① Descartes 在 1938 第一次描述了这个曲线。Jacob Bernoulli（1654—1705）对之非常着迷，甚至在他位于瑞士巴塞尔的墓碑上都刻着这条曲线，并题为 Eadem mutate resuiggo："虽然某些状况改变了，我却保持不变。"

案例分析

案例 3.1 学习逻辑图和混沌的模拟

(难度：Ⅱ)第 2 章结尾，我们讨论了迭代函数系统(IFS)。另一种 IFS 提供了对**混沌**世界的有趣的观察(见【Gleick87，Hofs85】)。它需要适当地设置窗口和视口。通过重复应用称为**逻辑图**的函数 $f(.)$，可以生成一系列的数值。这个函数通过下面的方程描述一条抛物线：

$$f(x) = 4\lambda x(1-x) \tag{3.20}$$

(检查它确实描述了一条抛物线)，其中 λ 是一个 0 到 1 之间的常数。从一个在 0～1 之间的起始点 x_0 开始，迭代应用函数 $f(.)$，可以得到**轨迹**(回忆在第 2 章的定义)：

$$x_{k-1} = f^{[k]}(x_0)$$

也就是说，$x_1 = f(x_0)$，$x_2 = f(f(x_0))$，$x_3 = f(f(f(x_0)))\cdots$，以此类推。

下面我们研究一下这个序列是如何得到的。通过函数在 x 上的值得到一个新的值 y(一个垂直的移动从 x 轴到$(x, f(x))$，然后再把 y 的值当作下一个 x 的值(一个水平的移动从$(x, f(x))$到$(f(x), f(x))$)，重复这个过程，就可以得到轨迹。一个复杂的世界潜伏在这里。

图 3.48 显示了当 x 在 0～1 之间变化时，抛物线 $y=4\lambda x(1-x)$的形状。其中 $\lambda=0.7$。(在 x 取多少的时候，这个曲线达到它的最大值?)

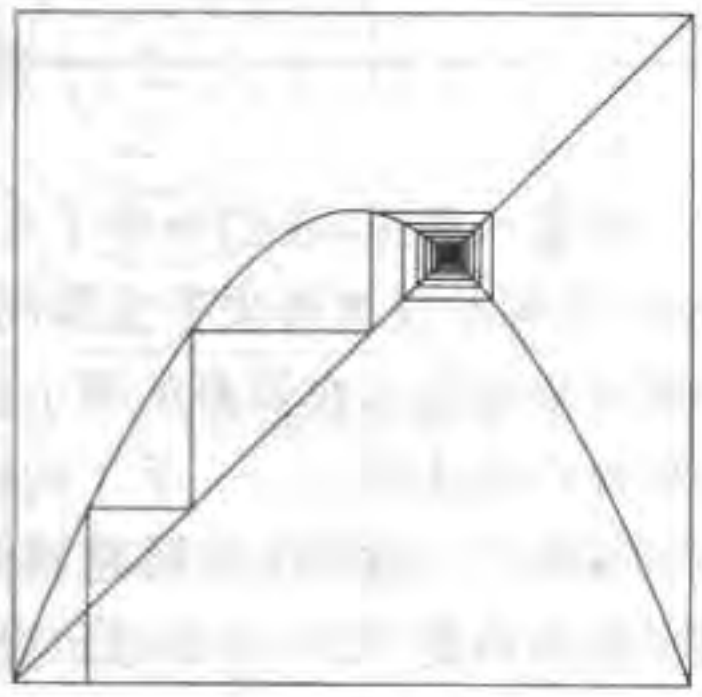


图 3.48 $\lambda=0.7$ 时的逻辑图

这里我们选择起点 $x_0=0.1$(一个任意值)，在 x 轴的这一点上，向抛物线方向画一条垂线，它将终止在 0.252(检查这个终止点)。在图像上，这就是绘制一条从(0.1，0)到(0.1，0.252)的线。为了找到曲线上的下一个点，我们从(0.1，0.252)绘制一条水平线到 $y=x$ 这条线，它会在(0.252，0.252)处终止，因此 $x_1=0.252$。下面我们计算函数在 $x_1=0.252$ 上的值。如图 3.48 所示，这在视觉上这就是移动到 $y=x$ 这条线上。接下来在新的值上计算 $f(.)$，就是向抛物线绘制一条垂直的线。像其他 IFS 一样，一直重复这个过程。从上一个点(x_{k-1}, x_k)，绘制一条水平线到(x_k, x_k)，再绘制一条垂直线到(x_k, x_{k+1})。这个图显示了当 $\lambda=0.7$ 时，这些数值很快就收敛到个稳定的“吸引点”上，这是一个满足 $f(x)=x$ 的固定点(当 $\lambda=0.7$ 时，这个数值是多少?)这个“吸引点”不依赖于起始点；这个序列总是很快的收敛到一个最终的数值。

如果 λ 是一个很小的值，这个动作就会更加简单：只有一个“吸引点”在 $x=0$ 处。但是当 λ 增大时，就会发生一些奇怪的事情。图 3.49(a)显示了 $\lambda=0.85$ 时的情况。这个序列的轨迹进入一个无限的循环，而不会到一个最终值。这里有几个吸引点，图中展示了在这个有限循环中，位于每条垂线上的吸引点。当 λ 增加到临界值 $\lambda=0.892486418\cdots$ 时，这个过程变成了一个真正的混沌。

图 3.49(b)是 $\lambda=0.9$ 的例子。对大多数起始点，轨迹都是周期的，周期间的轨迹数目

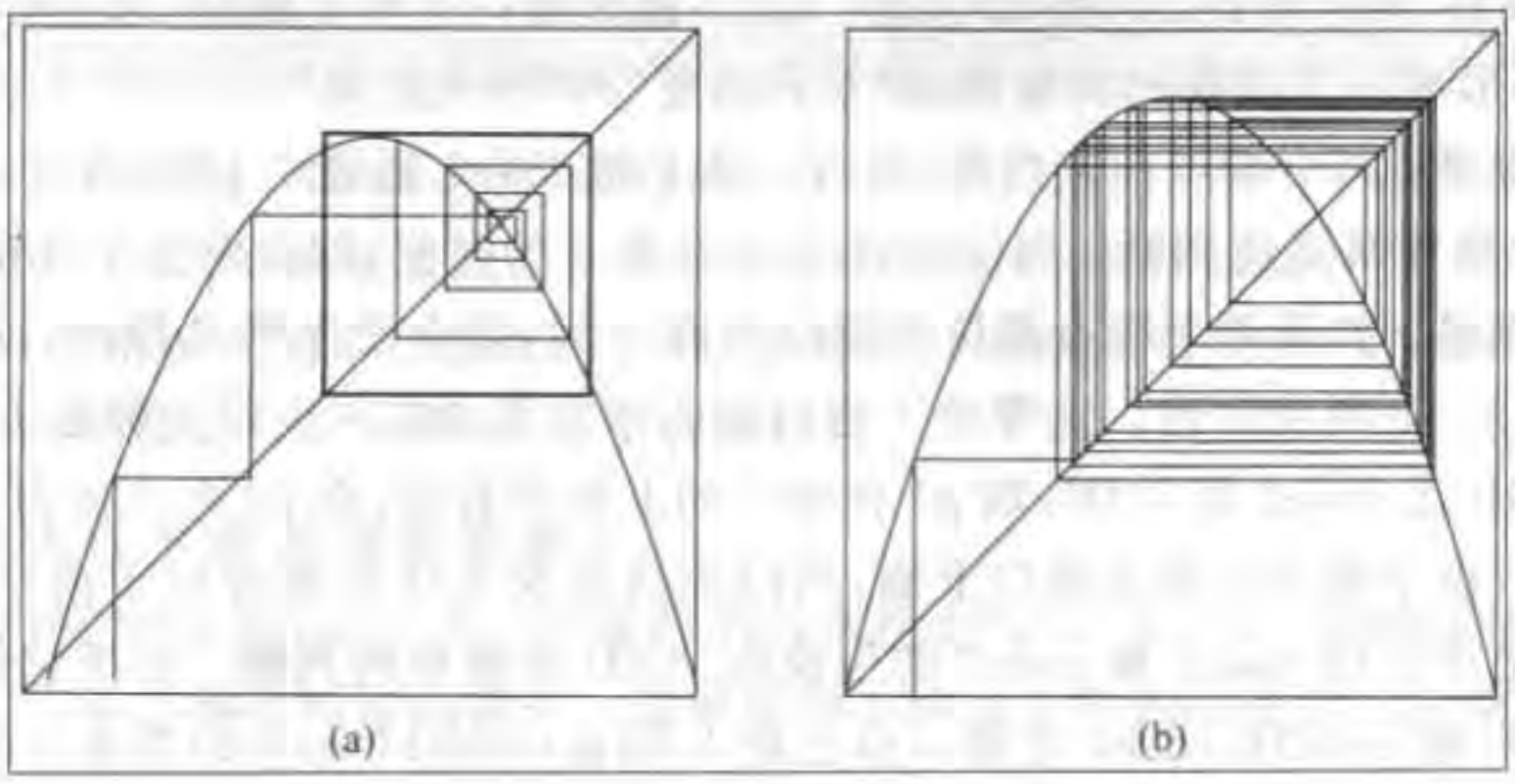


图 3.49

(a)$\lambda=0.85$ 和(b)$\lambda=0.9$ 时的逻辑图

变得非常的大。其他一些点产生非周期的运动，并且起始点很小的改动会导致非常不同的行为。在 1975 年，Mitchell Feigenbaum 首先发现这种现象的显著特征之前，大多数研究人员都认为，系统的微小改变(在这个例子中，将起始点做微小改变，或者将 λ 在 0.85～0.9 之间变化)应当导致行为上的微小改变，并且像这种简单的系统不会表现出极其复杂的行为。

Feigenbaum 的工作开拓了一个新的领域来研究复杂的非线性系统的本性，也被称为混沌理论【Gleick87】。用这种逻辑图做实验非常有趣。

写一个程序，允许用户研究逻辑图的重复迭代行为，就像图 3.49 所示的那样。用户设置一个合适的窗口和视口，使得可以清晰的看到整个逻辑图。进一步的，当用户给定 x_0 和 λ 的值，程序可以绘制出这个系统产生的有限循环。

案例 3.2　在 C/C++ 中实现 Cohen-Sutherland 算法

(难度Ⅱ)在 3.3.2 节中描述了 Cohen-Sutherland 算法的基本流程。这里，我们将讨论并充实一些在 C/C++ 中的实现细节，研究这些语言提供的高效的，底层的位操作。

如图 3.19 所示，我们首先要形成一个“内部/外部”码字，它说明 P 点与窗口的相对位置。一个 8 位二进制的 code 就够了：其中 4 位二进制用于捕获 4 个信息。依次测试 P 点与各个窗口边界，如果它在边界外，对应的位被设置为 1，用来表示真。图 3.50 显示了这是怎么做到的。code 被初始化为 0，然后每一独立位用 OR 位操作来设置为合适的值。数字 8、4、2 和 1 是简单的掩码。例如，因为 8 的二进制是 00001000，对一个数字和 8 进行 OR 位操作，会讲这个数字从右数的第 4 位设置为 1。

```
unsigned char code=0;            //初始所有位都是 0
...
if (P.x<window.l) code|=8;       //设置第 3 位
if (P.y>window.t) code|=4;       //设置第 2 位
if (P.x>window.r) code|=2;       //设置第 1 位
if (P.y<window.b) code|=1;       //设置第 0 位
```

图 3.50　设置 P 点的“内部/外部”码子中的点绘制曲线

在裁剪器中,两个端点 p1 和 p2(如图 3.20)都和窗口进行了测试,且它们的码字 code1 和 code2 已经形成。下面我们需要测试"平凡接受"和"平凡拒绝"。

- **平凡接受**:两个端点都在内部,所以 code1 和 code2 都是 0。在 C/C++ 中,这可以用 OR 位操作来迅速判断:当(code1|code2)是 0 的时候,就会发生平凡接受。
- **平凡拒绝**:如果两个端点都位于窗口的同一侧,就会发生平凡拒绝:都位于窗口左边,上面,下面或右边。这等价于它们的码字中至少在一个位上都是 1。例如,code1 是 0110 且 code2 是 0100,则 p1 在窗口的上面和右边,而 p2 在上面但并不在左边或右边。由于两个点都在窗口上面,所以该线段没有任何部分位于窗口内。因此,平凡拒绝可以用 code1 和 code2 的位操作 AND 来简单的判断:如果它们在某些位上都是 1,则 code1&code2 在那一位也是 1,因此(code1&code2)是非 0 的。

既没有平凡接受,也没有平凡拒绝,就截断。

另一个实现问题是如何对窗口外的部分做有效的截断。假设我们知道码字为 code 的点 P 在窗口外。我们可以通过测试 code 的每一位来知道 P 位于窗口的哪一边,再通过等式(3.5)就可以完成截断。图 3.51 是一个截断的过程,它寻找新的点(例如图 3.20 中的 A)来替换 P。它通过用一个掩码和 code 进行 AND 位操作,来确定 P 和窗口的相对位置。它还需要传入事先计算好的 dely 和 delx。

```
ChopLine(Point2 &P, unsigned char code, float dely, float delx)
{
    if (code & 8) { //左边
        P.y+=(window.l-P.x) * dely / delx;
        P.x=window.l;
    }
    else if (code & 2) { // 右边
        P.y+=(window.r-P.x) * dely / delx;
        P.x=window.r;
    } else if (code & 1) { // 下面
        P.x+=(window.b-P.y) * delx / dely;
        P.y=window.b;
    } else if (code & 4) { // 上面
        P.x+=(window.t-P.y) * delx / dely;
        P.y+=window.t;
    }
}
```

图 3.51 截断线段在窗口外的部分线

写一个 Cohen-Sutherland 算法的完整实现,将这里描述的代码和 3.3.2 节中的代码整合在一起。通过绘制一个窗口,以及一组随机选择的线段来测试算法,将在窗口内的部分显示为红色,窗口外的部分显示为黑色。

练习

为什么不会发生“除数为 0”的情况

考虑一个 delx 为 0 的垂直线段。为什么代码 P.y+=(window.l−P.x) * dely/delx 导致的除数是 0 的情况永远不会发生？同样，解释为什么用于计算 delx/dely 或者 dely/delx 的四条语句，在除数是 0 的时候，永远不会被执行。

在同一迭代中进行两次截断？

如果我们将“else if (code & 2)”这样的行替换为“if (c & 2)”，并尝试依次进行两次线段的“截断”，可能会得到性能的提升。说明这种方法也会导致端点的计算错误，从而导致灾难性的结果。

案例 3.3　用键盘让恐龙具有生命

(难度Ⅱ)回忆图 3.10 中的恐龙折线画，将 dino.dat 显示在一个特殊的位置上。写一个应用程序，它允许用户用键盘上的'a'和'd'来控制恐龙向左和右移动，以及用'w'和's'控制恐龙上下移动。为使恐龙平滑运动，可能需要用到双缓冲。

案例 3.4　绘制圆弧

(难度Ⅱ)在建筑作品的历史中，一直使用圆弧。它们的结构的强度和装饰上的美观使它们成为建筑设计中非常重要的元素，很多形状被融入教堂，桥梁，门道等建筑中。

图 3.52 展示了两个基本的圆弧形状。图 3.52(a)的圆弧中心位于原点，宽度是 $2W$。该圆弧从基线上面 H 高的位置开始。它的主要元素是一个半径 $R=W$ 的半圆。根据风格可以调整 H/W 的比例。例如，H/W 可以和黄金分割率相关。

图 3.52(b)显示了第二种著名的形状的理想版本，这个被称为**尖拱**或“等边”圆弧，通常在教堂中出现[①]。这里，两个圆弧的半径 $R=2W$，在中心点上方相遇。(每个圆弧各扫描了多少度？)

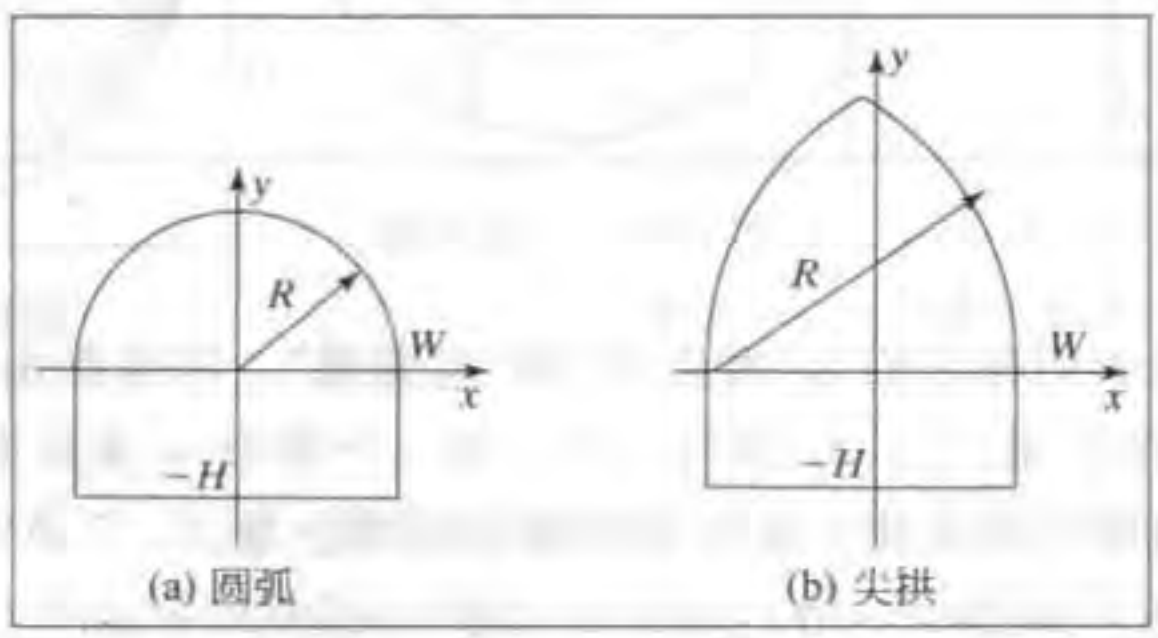


图 3.52　两种基本的弧形

图 3.53 展示了**葱形穹顶**[②](或“龙骨”)。这个圆弧出现在公元 1300 年，并在整个中世

① 源自 J. Fleming, H. Honour 和 N. Pevsner, Dictionary of Architecture (London: Penguin Books, 1980)。

② 源自古时法国的葱形穹顶，指一个 S 形状的曲线。

纪末期的建筑界中都非常流行。半径是 fR 的圆坐落在一个半径为 R 的圆弧上,其中 f 是一个因子。这种结构可以先确定两个圆的位置(C 点的坐标是多少?),每侧的两个圆弧融合在一起形成一个平滑的尖顶。按照 W 和 f 计算不同圆弧的参数,是很有趣的一件事情。

开发一个例程,它可以绘制上面所描述的各个圆弧。再写一个应用程序,用于绘制一系列有趣的圆弧,它们可能会出现在城堡、清真寺或者你设计的桥梁中。

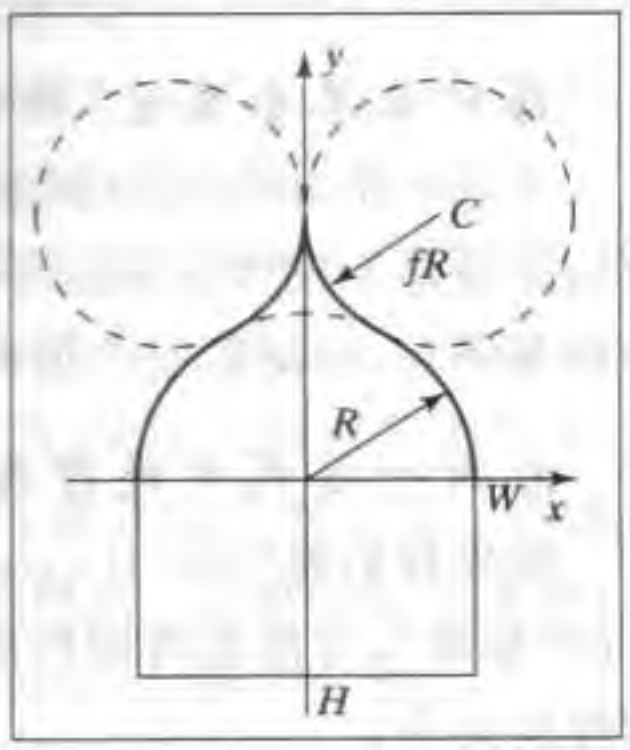


图 3.53 葱形穹顶

案例 3.5 一些在物理和工程中使用的图形

(难度Ⅱ)本案例涉及一些从物理和工程中的某些问题引发出来的一系列有趣的图形。第一幅图阐述了圆垂直相交的物理原理,第二幅图创建了一个用于学习电磁现象的图表。

静电场

图 3.54 中的圆形图案在物理和电子工程学中用于表示围绕在通电线圈周围的静电场场线。它也出现在数学关于复数变量的解析函数中。这里我们只是简单的将它看作是一组圆圈的优美组合,并考虑如何绘制它们。

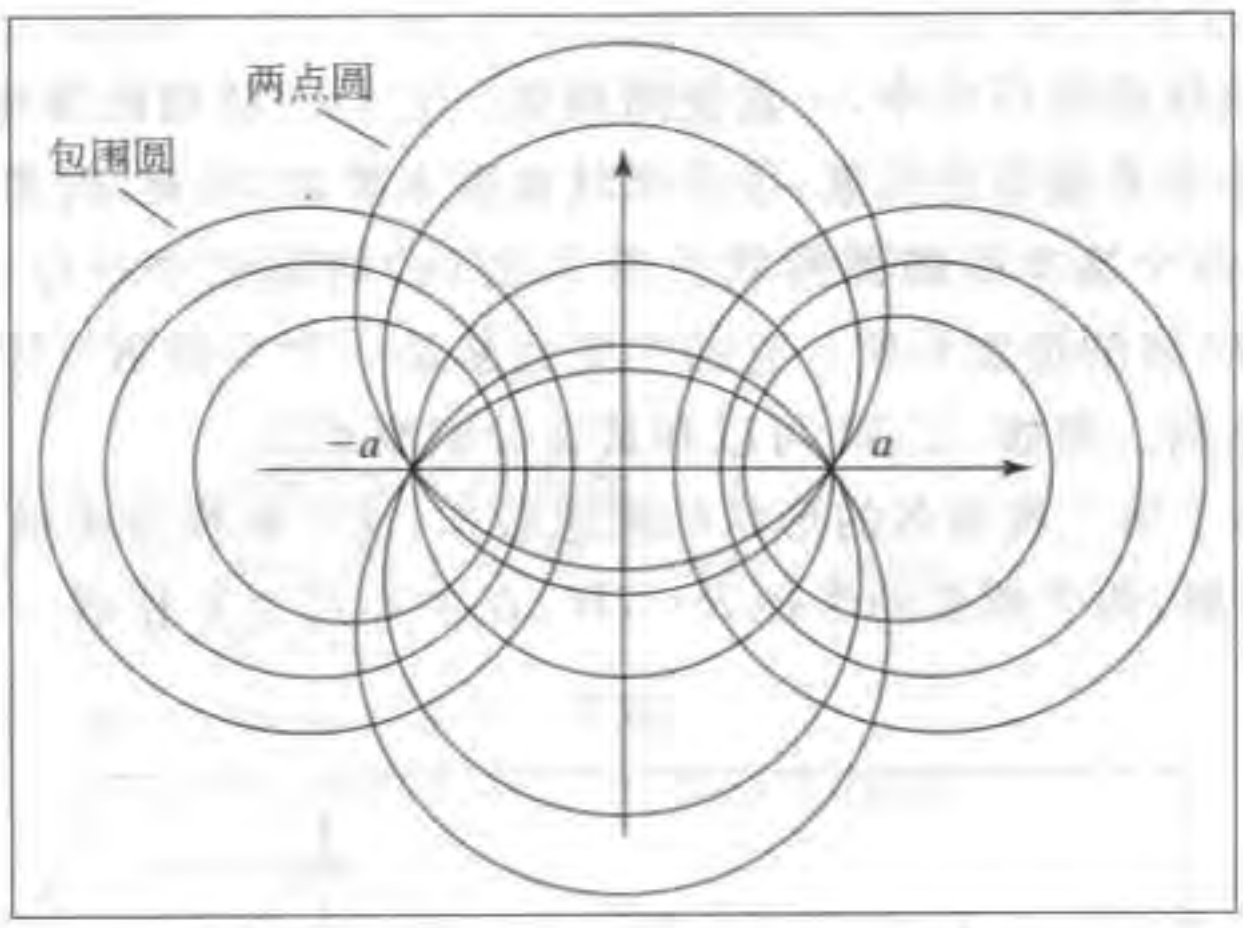


图 3.54 正交的圆

这里有两组圆圈,我们称它们为"两点圆"和"包围圆"。两点圆由经过两个给定点的一组圆构成。假定这两个点是$(-a, 0)$和$(a, 0)$。用一个参数 m 来区分不同的两点圆,而对于每一个 m,可以生成两个两点圆。这些圆的圆心和半径是:

$$center = (0, \pm a) \quad 以及 \quad radius = am$$

其中,m 在 1 到无穷之间变化。

包围圆中的圆包围着$(-a, 0)$或者$(a, 0)$中的某一个点。用参数 n 来区分不同圆的圆心和半径,它们的值是:

$$center = (\pm an, 0) \quad 以及 \quad radius = a\sqrt{n^2 - 1}$$

其中,n 在 1 到无穷之间变化。包围圆也被称为"Appolonius 圆",它们是由追逐问题【Ball

and Coxetr 74】引发的。Apollonius 圆上的点到$(-a, 0)$和$(a, 0)$的距离的比值是常数(这个常数和 a 与 n 的关系是什么?)

包围圆和两点圆有紧密的联系。每一个包围圆都是以直角"切割"两点圆。因此这两类圆被称为相互**正交**的。

写一个程序可以绘制这两类正交的圆。选择 m 和 n 值的集合,使得图形看上去很平衡,也很好看。

Smith 图表

在 Smith 图表中可以找到另一种圆圈图案,它经常出现在电子工程中,与电磁传输线有关。图 3.55 显示了 Smith 图表中的两个正交的家族。家族中所有圆都穿过一个共同点(1, 0)。A 家族中的圆心在$(1-m, 0)$,半径是 m,B 家族的圆心在$(1, \pm n)$,半径是 n,其中 n 和 m 都在 0 到 π 之间变化。写一个程序绘制这两类圆。

案例 3.6　平铺图案

(难度Ⅱ)计算机图形学提供了强有力的工具创建基于几何对象的令人满意的画面。其中最迷人的画面类型是那种从各个方向都无限重复的。它们被称为各种**平铺**和**重复的图案**。

a. **基础平铺**。图 3.56 展示了一个基本平铺的例子。一个图元,在这里它是四个 1/4 圆以一种简单的方式排列在一个方形区域中。为了在全平面绘制基于这个图元的平铺,我们创造一系列并排放置的视口以覆盖显示器表面,然后在每一个视口中绘制这个图元。

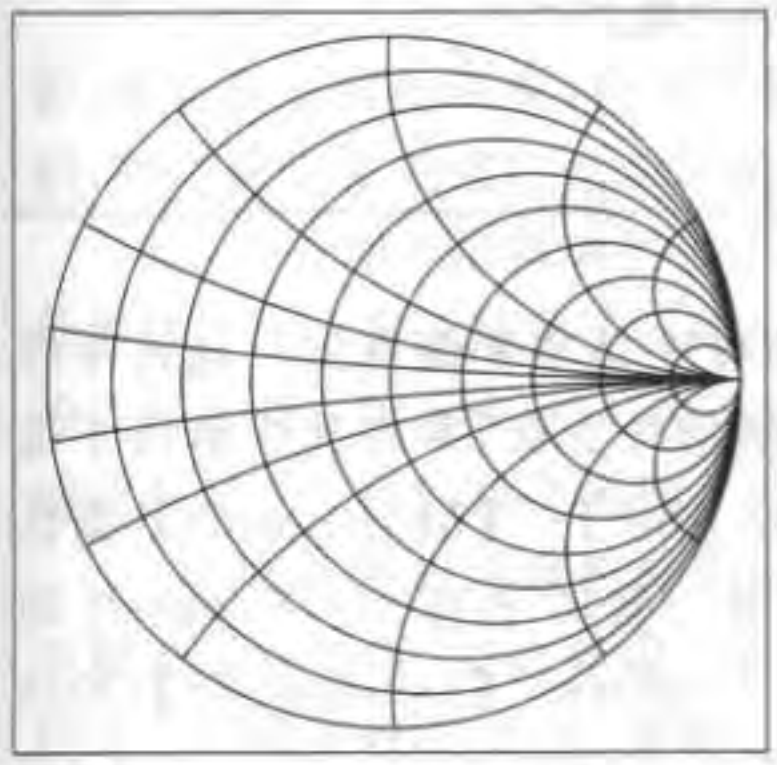


图 3.55　Smith 图表

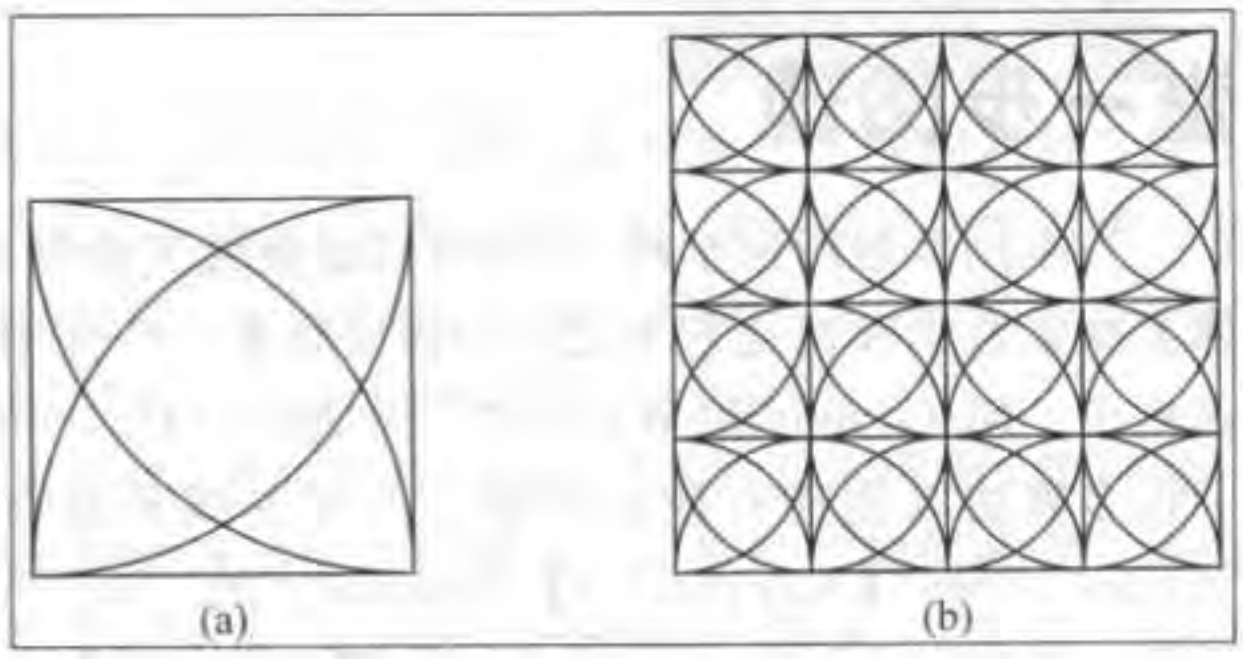


图 3.56　一个图元和用它平铺后的结果

写一个程序:

(a) 建立一个方形窗口,绘制一些有趣的图元(可能是裁剪后的部分);

(b) 在一系列视口中连续绘制这些图元,这些视口相邻排列并覆盖整个显示器表面。

至少用两种图元执行你的程序。

b. **Truchet 平铺**。把上面的方法稍作改动,从一个候选的图元"池"里相继的随机选择

图元。图 3.57(a)显示了著名的 Truchlet 平铺[①],它基于圆心分别在方形对角的两个 1/4 圆。瓦片(tile,平铺中使用的图元,英文中平铺(tiling)也有瓦片的意思——译者注)0 和瓦片 1 的不同只是做了 90°的旋转。

写一个程序,在整个视口上绘制 Truchet 平铺。相邻的两个瓦片使用随机选择的瓦片 0 或者瓦片 1。

也可以使用不是圆弧的曲线,如图 3.58 所示。为了避免结果曲线中的尖角,应当如何设置每个曲线与瓦片边界的角度呢?也可以扩展这个概念,使用多于两个瓦片。

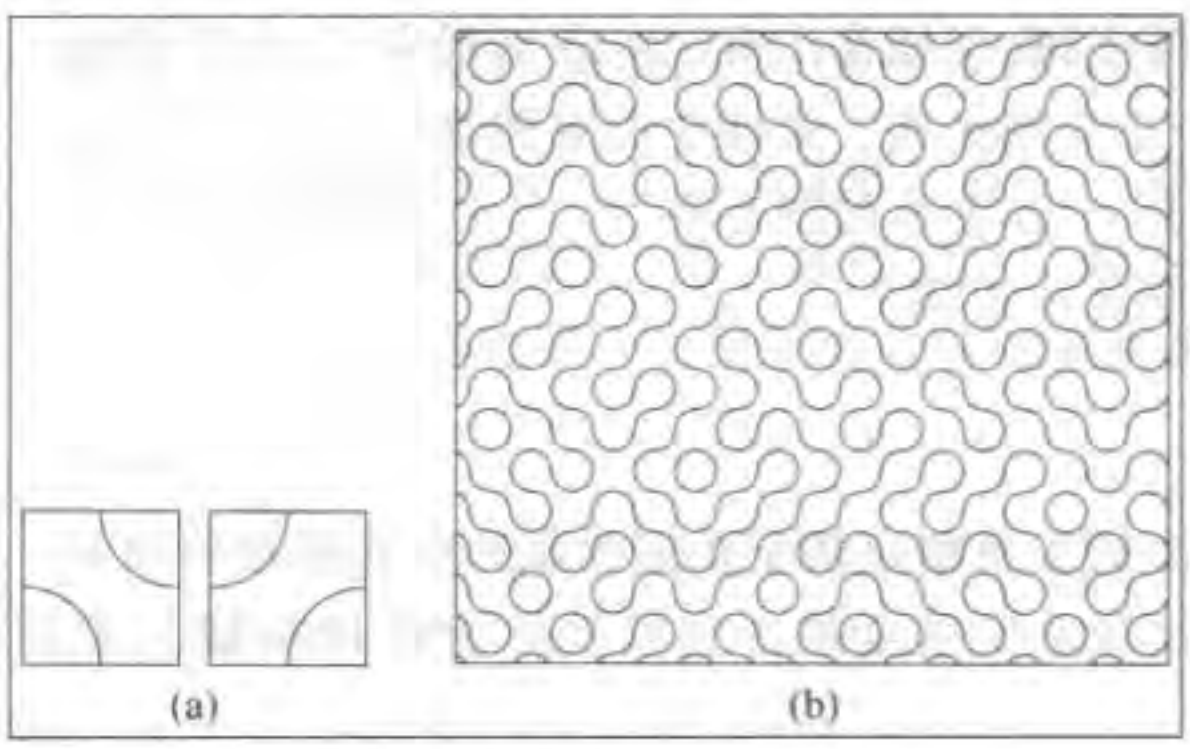


图 3.57 Truchet 平铺

(a) 两个瓦片;(b) 一个 Truchet 图案结果

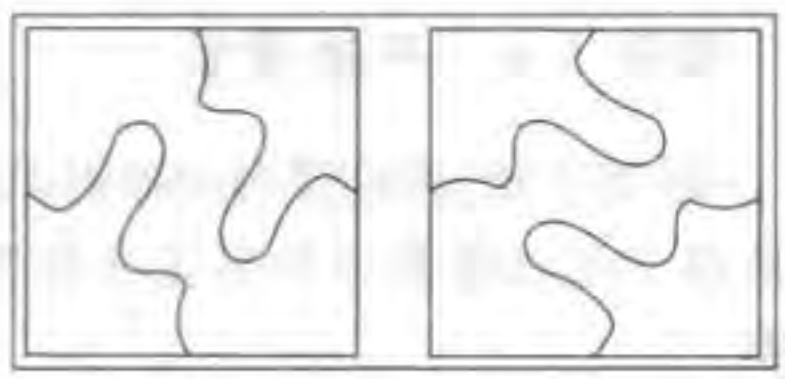


图 3.58 Truchet 平铺的扩展

扩展前面的程序,并设计两个附加的图案,使它可以绘制基于随机选择(随机旋转)这些瓦片的 Truchet 平铺。确保你所设计的图案能正确地融合在一起。

进一步阅读

当刚刚接触图形学时,写出可以绘制优美曲线和图案的程序是非常惬意的。这将带领你去探索数学和视觉艺术之间的深层联系。许多书可以提供指导方针,并在其中给出了很多例子。McGregor 和 Watt 的"The Art of Graphics for the IBM PC"【Mcgregor86】,提供了很多算法以创建有趣的图案。关于曲线和集合特别值得一看的书有 Jay Kappraff 的"Connections"【Kappraff91】,Dewdney 的"The Armchair Universe"【Dewdney88】,Stan Ogilvy 的"Excurisions in Geometry"【Ogilvy69】,Pedoe 的"Geometry and the Visual Arts"【Pedoe76】,Roger Sheperd 的"Mind Sights"【Shep90】,和 Martin Gardner 的一系列关于数学的书籍(例如"Time Travel"【Gardner88】和"Penrose Tiles to Trapdoor Ciphers"【Gardner89】)。Coxeter 写了一些关于几何很好的书,例如"Introduction to Geometry"【Coxeter69】和"Mathematical Recreations and Essays"【Ball74】,另外 Hoggar 的"Mathematics for Computer Graphcis"【Hoggar92】则讨论了一些迭代函数系统的特征。

① Smtit,C. "The Tiling Patterns of Sebastian Truchet and the topology of structural hierarchy." Leonardo. 20:4, pp: 373-385, 1987. (refd in Pickover, p. 386)。

第4章 图形学中的向量工具

本章学习目标

- 回顾向量算术，将向量与图形学中感兴趣的对象联系起来
- 在几何概念和它们的代数表示之间建立联系
- 线和面的参数化描述
- 正确区分点和向量
- 在图形学中使用点积
- 开发一些工具，能对三维空间中的对象进行操作，如两个向量的叉积

预览

为了处理计算机图形学中的几何对象，本章开发了一些有用的工具。4.1 节介绍在图形学中使用向量的原因以及主要使用的坐标系。4.2 节首先回顾向量的基本思想，介绍向量允许的主要操作。尽管大多数操作结果可以应用到任意维，在这里我们主要关注二维和三维的向量。4.3 节回顾非常有用的点积操作，并运用它解决数个几何问题，如进行正交投影，求点到直线的距离，以及计算一条光线被镜面反射后的方向。4.4 节回顾两个向量的叉积，并讨论它在三维图形学中的重要应用。

4.5 节介绍坐标框架的表示方法和齐次坐标，并强调点和向量是截然不同的两种几何对象类型。这一节还开发了两种主要的数学表示方法，用于表示线和面，并分别给出了两种方法的适用范围。接下来，这一节介绍了点的仿射组合，并描述了一种有趣的名为 tweening(内插)的动画形式。最后首次提到了贝塞尔曲线，演示了这种动画形式的一个应用。

4.6 节首先告诉我们使用向量会大大简化“求两直线段交点”这种基本问题，然后进一步展示了在处理“三点确定一圆”问题时使用向量的巨大优势。4.7 节讨论如何找到一条光线与直线或平面的交点，并将这一问题涉及的表示方法应用到裁剪问题中。4.8 节主要研究了如何使用凸多边形和凸多面体裁剪直线，并给出了功能强大的 Cyrus-Beck 裁剪算法。

本章最后的案例将对以上工具做进一步扩展，并为提高读者图形学编程技能创造机会。这些实例包括处理多边形、实现二维光线跟踪实验、制作内插动画的趣味实验以及开发高级裁剪工具。

4.1 概　　述

在计算机图形学中，我们要处理三维世界中的物体对象(二维世界和其中的对象可以看作它们的特例)。所有需要绘制的对象，以及将它们拍下来的“摄像机”，都拥有形状、位置和

方向等属性。我们必须编写合适的计算机程序来大致描述这些对象,并描述出光线如何围绕在它们周围,照亮它们,这样我们才能计算出最终在显示器上的每一个像素的值。让我们来举一个例子,设想这样一个动画,摄像机镜头飞过一个丘陵场景,里面包含了各种各样的建筑、树木、公路和汽车,那么摄像机都拍下了什么?这些最终都需要转换成数字,这是一个苛刻的要求。

但是,在图形学中,有两大基本工具能帮助我们实现上述要求——向量分析和变换。通过学习这些工具,我们可以设计出一些方法来描述我们所遇到的各种几何对象,并学会如何把这些几何方法转换成数字。把一切转换成数字听起来似乎有些枯燥呆板并且没有价值,但是有两个重要因素会改变人们的看法:首先,寻找正确数字的"过程"是迷人的;其次,最终的绘制景象往往令人惊叹!这些将指引我们去学习图形学程序设计中那些至关重要的算法。

在这一章中,我们将考查向量代数的基本操作,并理解它们是怎样应用到图形学中的。变换将在第 5 章介绍。我们将从最基本的内容开始,逐步开发出一些重要的工具和解决问题的方法,它们将多次出现在本书中。通过之前对图形学的了解,你可能会在本章中发现许多熟悉的概念,但是将向量分析应用到几何场景处理还是非常新的内容,并且依然值得深入研究。把一个场景中众多向量的各种属性搜集起来,并将它们与我们在图形学中遇到的实际几何问题相结合,寻找出一个解决方案,将是非常有用的。

向量为什么如此重要?

我们将举一些例子,在这些例子中,向量起到了至关重要的作用,希望它们能激发读者学习向量的兴趣。图 4.1 是三个图形学中出现的几何问题,类似的例子还有很多(有些将在后续章节出现)。

图 4.1(a)是一个计算机辅助设计问题:用户用鼠标在显示设备上设置了三个点,他需要画出经过这三个点的唯一圆。(你能看出这个圆吗?)在给定的坐标系下,所求圆的圆心在哪里呢?在 4.6 节中我们将看到,如果不使用向量,这个问题会很棘手,但是如果使用了适当的向量工具,就变得相当简单。

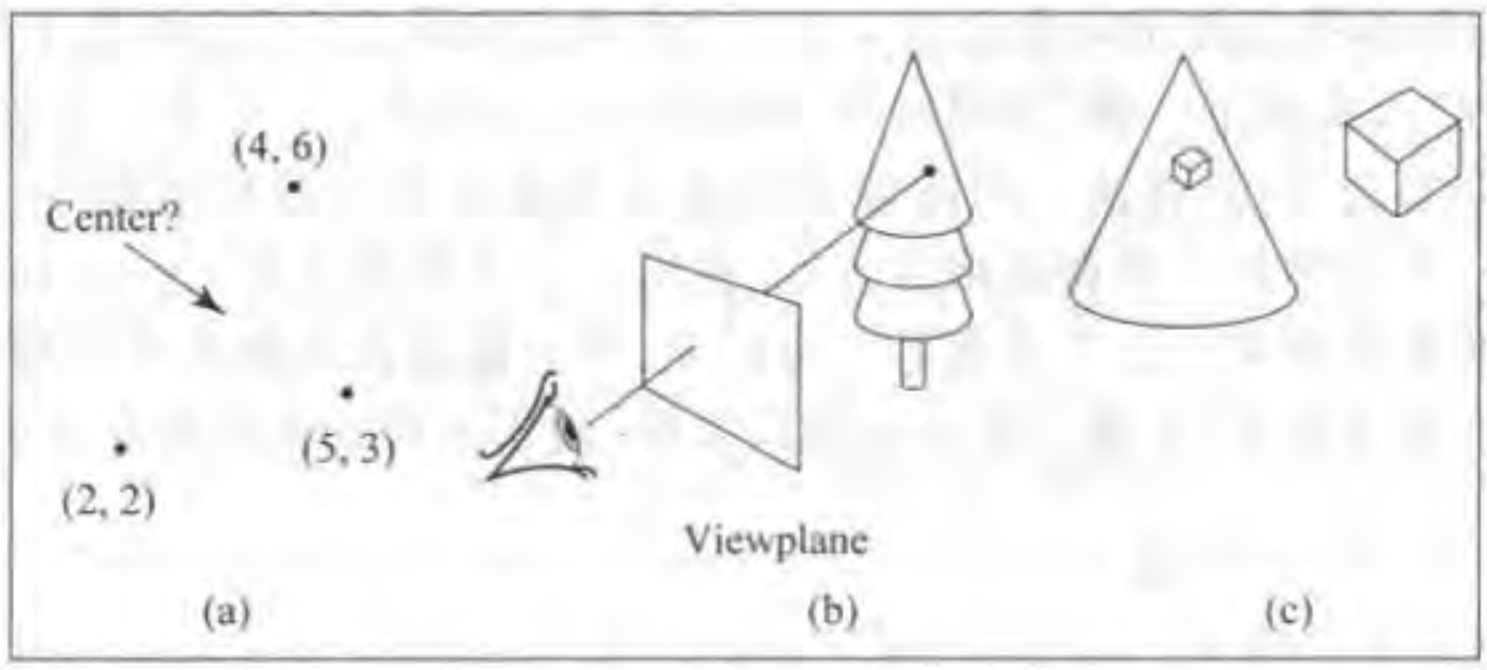


图 4.1 三个非常适合用向量解决的几何问题

图 4.1(b)是一个摄像机在拍摄一个有圣诞树的场景,摄像机必须在它的视平面(viewplane,类似于物理摄像机的胶卷平面)上形成这棵树的影像,这一影像将最终被转换为用户显示设备的一个屏幕窗口。这棵树的影像会出现在视平面的什么位置,它的实际形

状又该如何？要回答这些问题，我们需要仔细研究透视投影，而向量工具会极大的简化这一问题。如果觉得这个问题过于简单，那么想象一下，你正在制作一个动画，摄像机正沿着某个轨迹对场景聚焦，同时又在旋转，好了，去写个例程来产生整个图像序列吧！

图 4.1(c)是一个表面反光的圆锥体，在它上面可以看到一个立方体的镜像。在给定圆锥体、立方体和摄像机位置的前提下，怎样才能获得反射影像的准确位置，它的颜色和形状又如何？在第 12 章学习光线跟踪时，我们将进一步推广向量工具的使用，使它很好地解决我们面临的问题。

一些基础知识

我们使用的所有点和向量都是基于某一坐标系定义的。图 4.2 是一些常用的坐标系。每个坐标系都有一个原点 O 以及从 O 出发的一些轴，这些轴一般都互相垂直。每个轴上都有距离刻度，一个点的坐标就是由它沿每个轴向离开原点的距离而定。图 4.2(a)是一个常用的二维坐标系。当它扩展到三维，我们会发现两个版本：左手坐标系和右手坐标系。图 4.2(b)是一个右手三维坐标系，图 4.2(c)是一个左手三维坐标系。

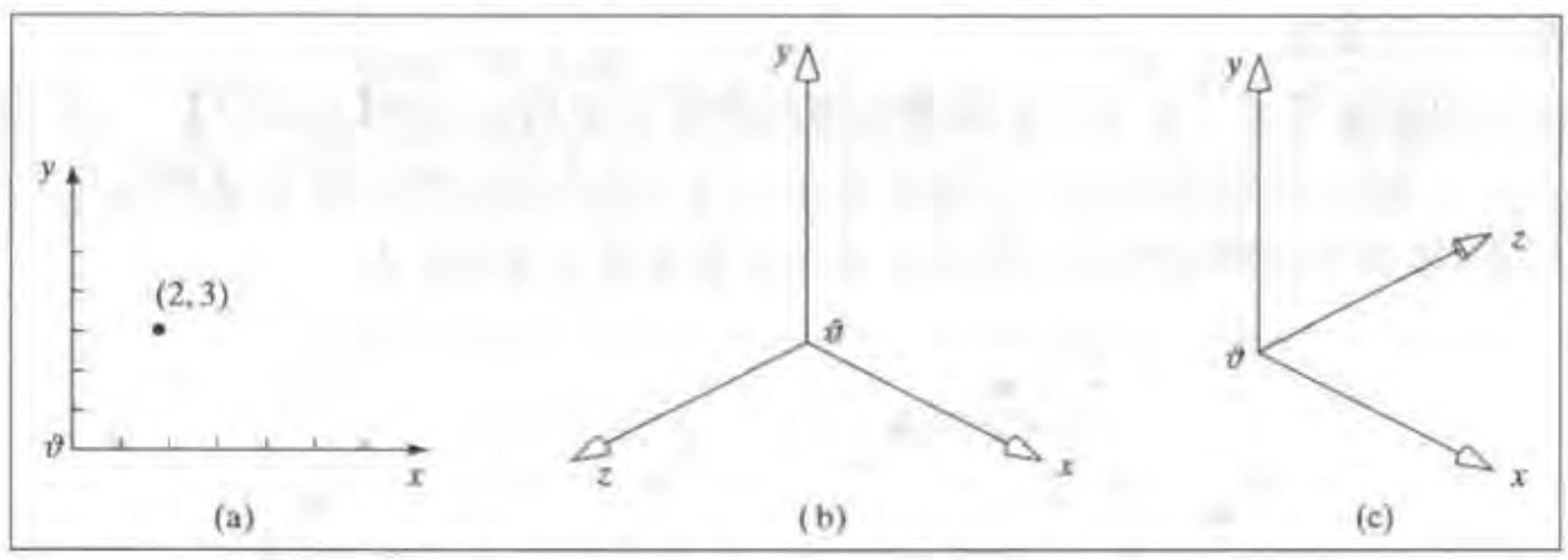


图 4.2　常用的二维和三维坐标系

右手和左手三维坐标系

在右手坐标系中，如果你右手绕 z 轴，沿 x 轴正方向握向 y 轴正方向，如图所示，你的拇指将指向 z 轴正方向。在左手系中，你必须用左手，才能让你的拇指指向 z 轴正方向。右手坐标系在数学、物理和工程中更方便常用，本书将使用右手坐标系建立对象模型。但是左手坐标系在处理图形学中的观察系统(viewing system)和摄像机系统时有很好的适用性。

我们将首先回顾向量的基础知识，学习如何使用它们，尤其是如何在图形学中应用它们。在 4.5 节中，我们将返归本源，介绍点和向量的一个重要区别，如果忽略这一区别，将会为图形学编程带来巨大困难。

在前面我们有个疑问，向量为什么如此重要。现在我们回到这个主题。不难发现，向量在数学开发中易于表示，它们的长度和方向属性比较直观；此外，与基于像素的“屏幕空间”(光线跟踪是它的一个应用)相比，向量能精确的描述更底层、更抽象的“对象空间”；更进一步地，向量无需描述位置信息，所以储存向量会稍微节省空间。

但是最重要的原因是，向量可以组合出新的物体对象！它们可以使用加、减、数乘运算，或者通过其他操作得到一个标量(如点积)，甚至得到三维空间里新的向量(如叉积)，而这些

结果都有很直观、便于理解的属性。唯一没有定义的,是两个向量相除的商。向量的乘积不但易于计算,而且可以帮助我们解决一些原本棘手的几何问题。我们将在后面深入探讨向量的这些特性。

4.2 向量回顾

向量算法提供了一种统一的方法来对几何思想进行代数的表示。在图形学中尽管我们需要处理二维、三维和四维的向量,但是很多结论只需要给出一次,因为它们能很好的应用到任意维度。这种应用的广泛性可以将图形学中的很多问题归纳为一种表达方式,从而很好的解决更广泛的任务。

从几何的角度看,向量是具有长度和方向的实体,但是没有位置(作为对比,点只有位置,没有长度和方向)。向量可以对应很多物理量,如作用力、位移和速度。一个向量一般用一个指向某个方向、有一定长度的箭头来表示。在几何中把向量看成从一个点到另一个点的位移是很有价值的。

图 4.3 用向量展示了北斗七星随着时间的推移是如何运动的【Kerr97】。每颗星的当前位置用一个点表示,运动速度用一个向量表示。每个箭头的尖端代表了这颗星在 50 000 年之后的位置,你会发现那时候它们看起来完全不像是北斗七星了!

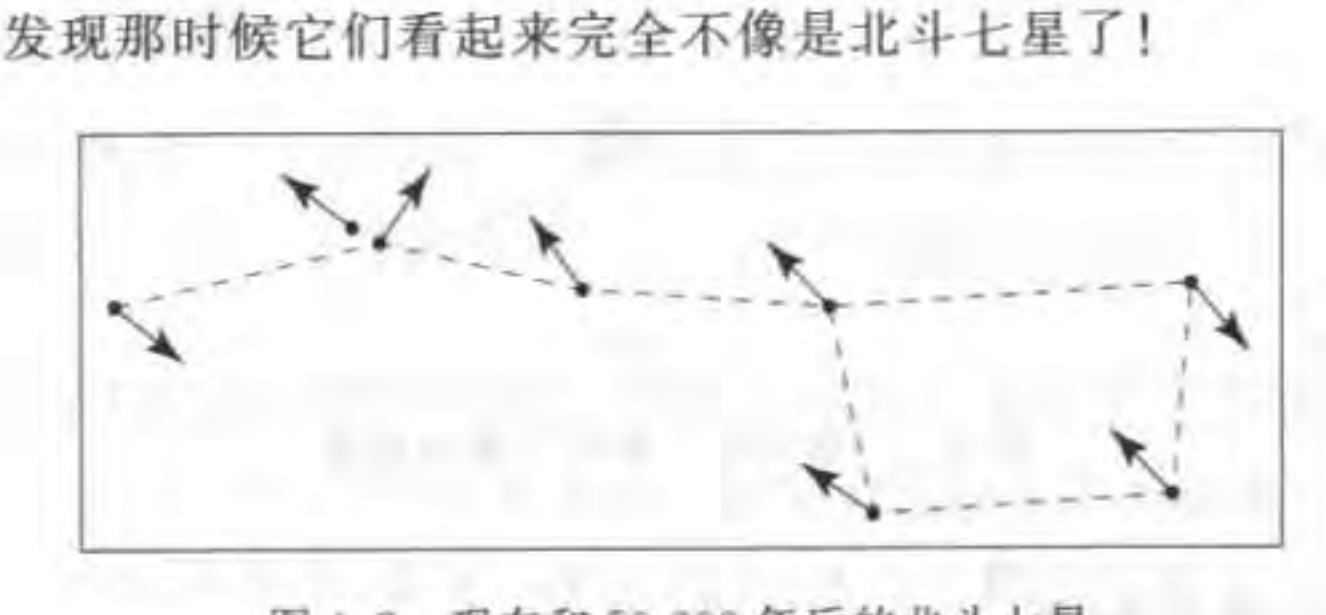


图 4.3 现在和 50 000 年后的北斗七星

如图 4.4(a)所示,在一个二维坐标系中有点 $P=(1,3)$ 和 $Q=(4,1)$。从 P 点到 Q 点的位移用向量 $\mathbf{v}=(3,-2)$ 表示[①],该位移可由 Q 点坐标和 P 点坐标的对应分量相减得到。这表明为从 P 点到达 Q 点,需要向下平移 2 再向右平移 3。由于向量是一个位移,所以它有长度和方向但没有固定位置:图中两个标有 $\mathbf{v}$ 的箭头实际上表示了同一个向量。图 4.4(b)给出了三维中对应的情况:$\mathbf{v}$ 是从点 P 到点 Q 的向量。通常的说法是:

两个点的差是一个向量:$\mathbf{v}=Q-P$

换个角度,我们也可以说点 Q 是由点 P 平移向量 $\mathbf{v}$ 得到的;或者说 $\mathbf{v}$ 偏移(offset)P 得到 Q。从代数上说,Q 可以表示成:

$$Q = P + \mathbf{v}$$

一个点与一个向量的和还是一个点:$P+\mathbf{v}=Q$

① 一般用大写字母表示点,用粗体小写字母表示向量。

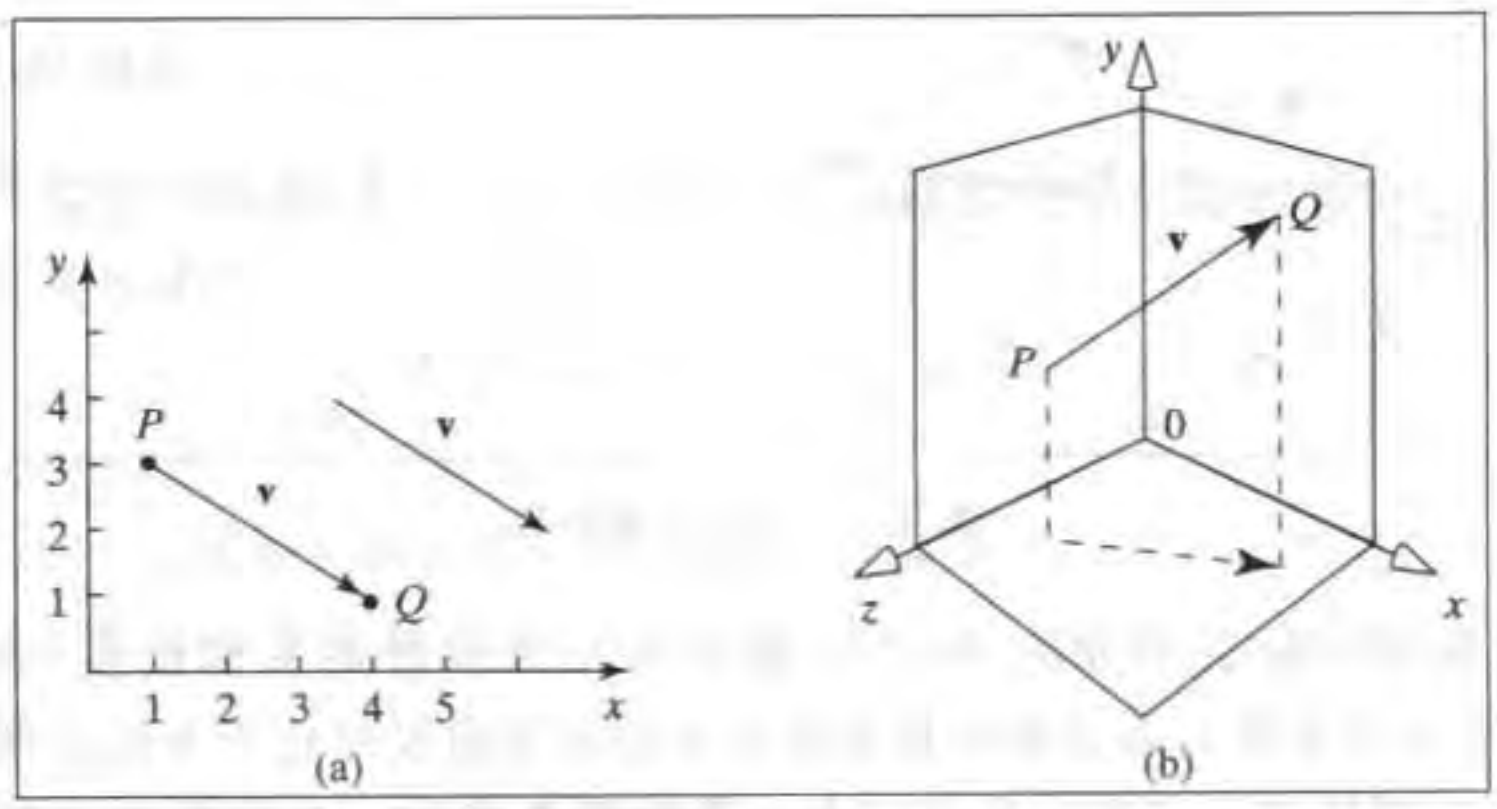


图 4.4　表示位移的向量

从这一点来说，我们可以把向量表示成它所有分量的列表，一个 n 维向量就是一个 n 元组：

$$\mathbf{w} = (w_1, w_2, \cdots, w_n) \tag{4.1}$$

通常我们只关心二维和三维向量，比如 $\mathbf{r}=(3.4,-7.78)$ 或 $\mathbf{t}=(33,142.7,89.1)$。在后面的章节中，我们会探索向量和它的表示法之间的区别。事实上我们将使用一种扩展的表示方法表示向量(和点)。用行矩阵的形式表示向量如 $\mathbf{t}=(33,142.7,89.1)$ 确实方便于纸上书写，但是必要时还需要用列矩阵来表示：

$$\mathbf{r} = \begin{pmatrix} 3.4 \\ -7.78 \end{pmatrix} \quad 或 \quad \mathbf{t} = \begin{bmatrix} 33 \\ 142.7 \\ 89.1 \end{bmatrix}$$

在第 5 章将会看到，将一个点或向量与一个矩阵相乘时，需要列矩阵的表示方法。

4.2.1　向量基本运算法则

向量允许两个基本操作：向量相加和与一个标量(实数)的数乘[①]。因为如果 $\mathbf{a}$ 和 $\mathbf{b}$ 是两个向量，s 是一个标量，$\mathbf{a}+\mathbf{b}$ 和 $s\mathbf{a}$ 都是有意义的。例如，令 $\mathbf{a}=(2,5,6)$ $\mathbf{b}=(-2,7,1)$，我们可以获得两个新的向量：

$$\mathbf{a}+\mathbf{b} = (0,12,7)$$
$$6\mathbf{a} = (12,30,36)$$

这些操作是通过应用于各分量完成的。图 4.5 是一个二维的例子，其中 $\mathbf{a}=(1,-1)$，$\mathbf{b}=(2,1)$，我们可以用两种不同的图示表示这两个向量的和。在图 4.5(a)中，两个向量从同一点出发[②]，从而形成平行四边形的两个边。两个向量的和向量就是这个平行四边形的对角线——从两个向量的公共起点出发的对角线。这种方法——向量加法的“平行四边形法则”——是两个作用力作用在物体上的直观表示：对角线表示合力。

---

① 有的系统中标量可以是复数，我们不讨论这种情况。

② 由于向量没有固定的位置，把它们画在某个位置只是为了直观，图中的结果向量也是一个道理。

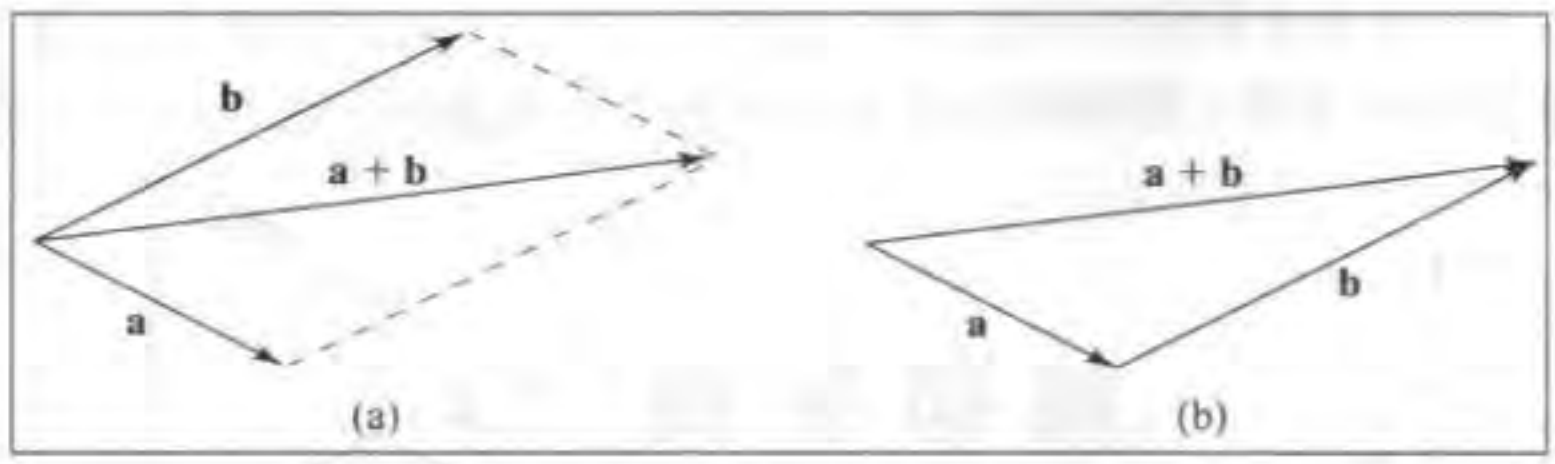


图 4.5 两个向量的和

图 4.5(b)是另一种表示方法,一个向量从另一个向量的头部出发(也就是说,我们把 **b** 的尾部置于 **a** 的头部),那么和向量就是从 **a** 的尾部出发终止于 **b** 的头部。求和过程形成一个三角形,直观地表示了和向量是两个位移的简单相加。和向量的分量是各向量对应分量之和,与代数求和的结果一致。

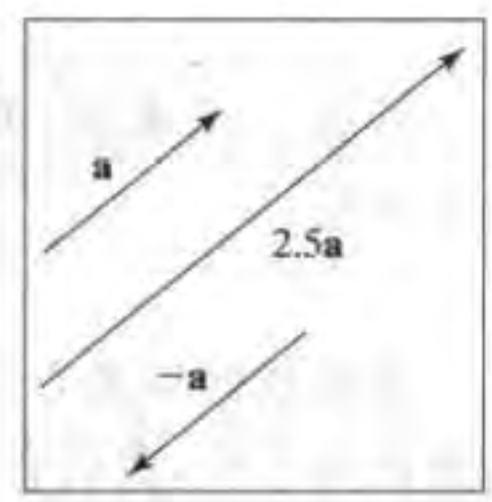


图 4.6 向量的数乘

图 4.6 表示了向量的数乘。$s=2.5$ 时,向量 $s\mathbf{a}$ 具有与向量 **a** 相同的方向,但长度是 **a** 的 2.5 倍。当 s 为负数时,$s\mathbf{a}$ 的方向与 **a** 相反,图中给出了 $s=-1$ 的例子。

减法可以通过一次加法和一次数乘得到:**a**−**c** 实际上就是 **a**+(−**c**)。图 4.7 给出了这一操作的几何解释,求 **a** 与 **c** 的差就是求 **a** 与−**c** 的和(图 4.7(b))。根据平行四边形法则,该向量相当于从 **c** 的头部出发,到 **a** 的头部结束的向量(图 4.7(c))。这可以看作是 **a** 与 **c** 形成的平行四边形的一条对角线,注意到另一条对角线表示了和向量 **a**+**c**。

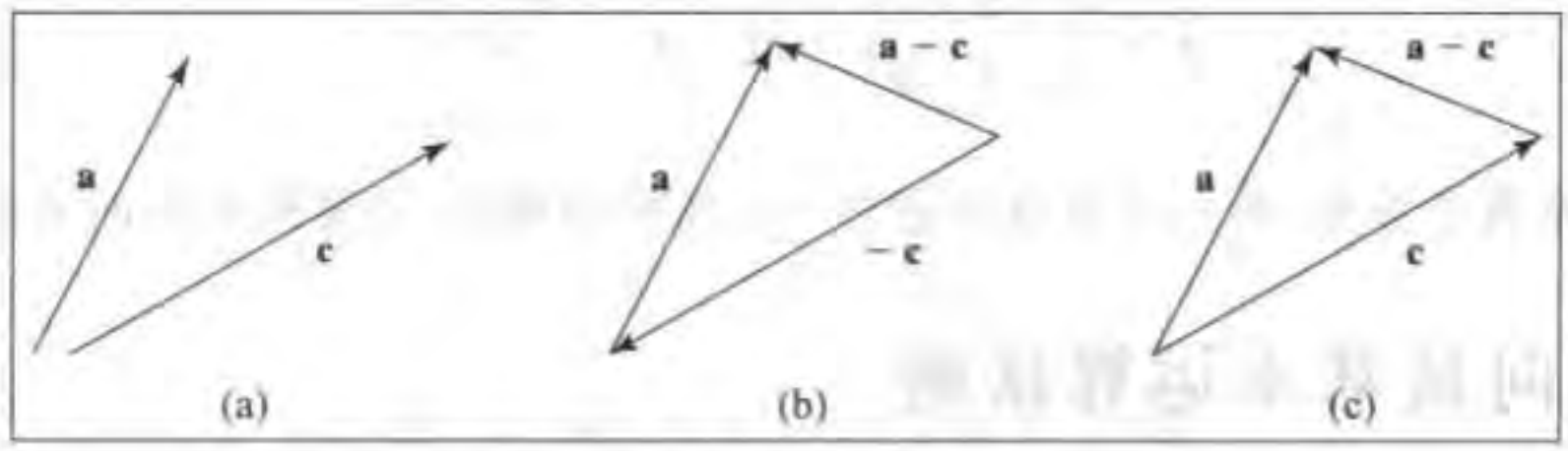


图 4.7 向量相减

4.2.2 向量线性组合

掌握了向量的加法和数乘,我们可以定义任意多个向量的线性组合。为得到两个向量 **v** 和 **w**(维数相同)的线性组合,我们首先分别对它们数乘上标量 a 和 b,然后再将两个加权后的向量相加,即 $a\mathbf{v}+b\mathbf{w}$。对于 m 个向量,有如下更一般的定义:

定义:m 个向量 $\mathbf{v}_1,\mathbf{v}_2,\cdots,\mathbf{v}_m$ 的**线性组合**是具有如下形式的向量

$$\mathbf{w}=a_1\mathbf{v}_1+a_2\mathbf{v}_2+\cdots+a_m\mathbf{v}_m \tag{4.2}$$

其中 $a_1,a_2,\cdots,a_m$ 是标量。

例如,基于标量 2 和 6 的线性组合 2(3,4,−1)+6(−1,0,2),得到向量(0,8,10)。在后续章节中我们会遇到相当复杂的向量线性组合,尤其是在用样条函数表示曲线曲面时。

两种特殊的线性组合,仿射组合和凸组合,在图形学中非常重要。

向量的仿射组合

如果线性组合的系数 $a_1, a_2, \cdots, a_m$ 的和等于 1，那么它就是**仿射组合**。因此等式(4.2)中的线性组合若满足：

$$a_1 + a_2 + \cdots + a_m = 1 \tag{4.3}$$

就是一个仿射组合。

例如，$3\mathbf{a}+2\mathbf{b}-4\mathbf{c}$ 就是 $\mathbf{a}$、$\mathbf{b}$、$\mathbf{c}$ 的一个仿射组合，但是 $3\mathbf{a}+\mathbf{b}-4\mathbf{c}$ 不是。为了使两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的线性组合成为一个仿射组合，可以令一个向量的系数为任意标量 t，另一个则为 $(1-t)$，于是有：

$$(1-t)\mathbf{a} + (t)\mathbf{b} \tag{4.4}$$

注意到仿射组合的条件与向量本身无关，只与作为系数的标量有关。

点和向量的仿射组合会出现在不同场合，后面将会看到。

向量的凸组合

凸组合在数学中具有重要的位置，在图形学中也有很多应用。凸组合是对仿射组合加以更多的限制得来的。不但线性组合的所有系数之和需要是 1，每一个系数还必须是非负数。等式(4.2)中的线性组合必须满足以下条件才是一个凸组合：

$$a_1 + a_2 + \cdots + a_m = 1 \tag{4.5}$$

并且对所有 $i = 1, \cdots, m$，有 $a_i \geqslant 0$，这将导致所有的 a_i 必须位于 0 和 1 之间(为什么?)

因此，$0.3\mathbf{a}+0.7\mathbf{b}$ 是 $\mathbf{a}$ 和 $\mathbf{b}$ 的凸组合，但 $1.8\mathbf{a}-0.8\mathbf{b}$ 不是。凸组合系数的集合 $a_1, a_2, \cdots, a_m$ 有时也叫做一个单位划分，表示将一个单位数量的“物质”分成一个个小块。凸组合经常应用于类似这样的场合：一个人要制作单位数量的某种配方，需要把正数数量的不同成分合并在一起。凸组合经常出现在意想不到的地方，例如，我们会发现第 8 章中的样条曲线事实上就是某些向量的凸组合，又如在第 11 章讨论颜色时，我们会发现颜色也可以看成向量，这样单位亮度的任何颜色都可以看成是三个基本颜色的凸组合！

4.2.3　向量的度量和单位向量

假如 n 维向量 $\mathbf{w}$ 已给定了 n 个分量 $(w_1, w_2, \cdots, w_n)$，那么如何定义和计算它的大小(或长度)？我们用 $|\mathbf{w}|$ 表示向量的大小并把它定义成从向量尾部到头部的距离，根据勾股定理

$$|\mathbf{w}| = \sqrt{w_1^2 + w_2^2 + \cdots + w_n^2} \tag{4.6}$$

例如，$\mathbf{w}=(4,-2)$ 的大小就是 $\sqrt{20}$，$\mathbf{w}=(1,-3,2)$ 的大小就是 $\sqrt{14}$。长度为 0 的向量表示为 0。注意如果 $\mathbf{w}$ 是从点 A 到点 B 的向量，那么 $|\mathbf{w}|$ 就是点 A 到点 B 的距离(为什么?)。

有时经常需要缩放向量，使它的长度为 1。这一过程叫做向量的**归一化**，归一化的结果叫做**单位向量**。例如，为得到 $\mathbf{a}$ 的归一化向量，我们用 $1/|\mathbf{a}|$ 数乘 $\mathbf{a}$：

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{|\mathbf{a}|}, \quad 其中 |\mathbf{a}| \neq 0 \tag{4.7}$$

为了辨认方便,我们在归一化后的向量上加上一个^符号,很显然它是一个单位向量:$|\hat{\mathbf{a}}| = 1$,并与 $\mathbf{a}$ 有相同的方向。例如,如果 $\mathbf{a}=(3,-4)$,那么 $|\mathbf{a}|=5$,归一化的结果为 $\hat{a}=\left(\frac{3}{5},\frac{-4}{5}\right)$。有时我们也把单位向量看成**方向**。注意任何一个向量都可以写成它的大小乘上它的方向:如果 $\hat{\mathbf{a}}$ 是 $\mathbf{a}$ 归一化后的向量,向量 $\mathbf{a}$ 总可以写成 $\mathbf{a}=|\mathbf{a}|\hat{\mathbf{a}}$。鉴于计算机程序不可避免的错误和误差,有时我们关心的向量的长度会变成 0。因此,在归一化一个向量前,一定要确认它的长度是非零的!

练习

练习 4.2.1 向量归一化

归一化下面的向量:

(1) (1,−2,0.5)

(2) (8,6)

(3) (4,3)

4.3 点 积

有另外两个功能强大的工具一直推动着向量的应用:点(内)积和叉积。点积得到一个标量,而叉积只能作用于三维向量,它的结果是产生另一个向量。叉积的一个重要性质是提供了一种检测平面多边形凸性的简单方法。在本节,我们将回顾点积的基本特性,主要是为了开发一种正交的表示方法。然后,我们将利用点积解决图形学中几个重要的几何问题。接下来我们会介绍叉积,并用它解决一些三维的几何问题。

两个向量的**点积**很容易定义和计算。点积可以应用到任意维向量。对于二维向量 (a_1,a_2) 和 (b_1,b_2),它们的点积是一个值为 $a_1b_1+a_2b_2$ 的标量。也就是说,计算点积时,只需将两个向量相应的分量相乘,然后将结果相加。例如,(3,4)和(1,6)的点积为 27,(2,3)和(9,−6)的点积是 0。

点积的定义很容易推广到 n 维向量:

定义:两个 n 维向量的 $\mathbf{v}=(v_1,v_2,\cdots,v_n)$ 和 $\mathbf{w}=(w_1,w_2,\cdots,w_n)$ 的点积 d 表示为 $\mathbf{v}\cdot\mathbf{w}$,

$$d = \mathbf{v}\cdot\mathbf{w} = \sum_{i=1}^{n} v_i w_i \tag{4.8}$$

例 4.3.1

- (2,3,1)Z 和(0,4,−1)的点积是 11
- (2,2,2,2)·(4,1,2,1.1)=16.2
- (22,2,7)·(12,−9,11)=323

• $(169,0,43)\cdot(0,375.3,0)=0$

4.3.1　点积的性质

点积具有四个很明显的基本性质，从它的定义中很容易推出（见练习）：

1. 对称性：　$\mathbf{a}\cdot\mathbf{b}=\mathbf{b}\cdot\mathbf{a}$
2. 线性：　$(\mathbf{a}+\mathbf{c})\cdot\mathbf{b}=\mathbf{a}\cdot\mathbf{b}+\mathbf{c}\cdot\mathbf{b}$
3. 同质性（Homogeneity）：　$(s\mathbf{a})\cdot\mathbf{b}=s(\mathbf{a}\cdot\mathbf{b})$
4. $|\mathbf{b}|^2=\mathbf{b}\cdot\mathbf{b}$

性质 1 说明两个向量的点积结果与它们的次序无关：点积是**可交换的**。性质 2 和性质 3 说明点积是**线性的**，也就是说，向量之和的点积等于分别点积后再求和，对点积中的一个向量做数乘相当于对点积结果做数乘。最后一个性质也很重要，它表明一个向量与自身的点积得到的是这个向量**长度的平方**。性质 4 也常写作 $|\mathbf{b}|=\sqrt{\mathbf{b}\cdot\mathbf{b}}$。

下面的运算将告诉我们如何用以上的性质简化带有点积的表达式，这些结果将用于后续章节。

例 4.3.2　化简 $|\mathbf{a}-\mathbf{b}|^2$

化简两个向量 $\mathbf{a}$ 和 $\mathbf{b}$ 的差长度（平方）的表达式，得到如下结果：

$$|\mathbf{a}-\mathbf{b}|^2=|\mathbf{a}|^2-2\mathbf{a}\cdot\mathbf{b}+|\mathbf{b}|^2 \tag{4.9}$$

推导过程如下：令 C 等于 $|\mathbf{a}-\mathbf{b}|^2$。

由性质 4，C 为如下点积：

$$C=|\mathbf{a}-\mathbf{b}|^2=(\mathbf{a}-\mathbf{b})\cdot(\mathbf{a}-\mathbf{b})$$

由线性：$C=\mathbf{a}\cdot(\mathbf{a}-\mathbf{b})-\mathbf{b}\cdot(\mathbf{a}-\mathbf{b})$。

通过对称性和线性进一步化简得：$C=\mathbf{a}\cdot\mathbf{a}-2\mathbf{a}\cdot\mathbf{b}+\mathbf{b}\cdot\mathbf{b}$。

通过性质 4 即得到所求结果：$C=|\mathbf{a}|^2-2\mathbf{a}\cdot\mathbf{b}+|\mathbf{b}|^2$。

用加号代替上例中的减号，便可以得到以下的相似结果，这个关系式也很有用。

$$|\mathbf{a}+\mathbf{b}|^2=|\mathbf{a}|^2+2\mathbf{a}\cdot\mathbf{b}+|\mathbf{b}|^2 \tag{4.10}$$

4.3.2　两个向量的夹角

点积最重要的应用就是计算两个向量的夹角，或者两条相交直线的夹角。图 4.8 给出了二维的情况，其中向量 $\mathbf{b}$ 和向量 $\mathbf{c}$ 与 x 轴的夹角分别为 ϕ_b 和 ϕ_c，从基本三角函数可得：

$$\mathbf{b}=(|\mathbf{b}|\cos\phi_b,\ |\mathbf{b}|\sin\phi_b)$$

$$\mathbf{c}=(|\mathbf{c}|\cos\phi_c,\ |\mathbf{c}|\sin\phi_c)$$

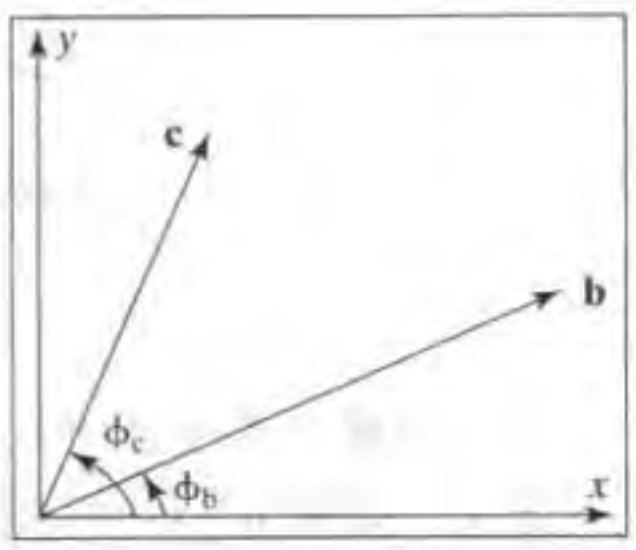


图 4.8　求两个向量的夹角

我们希望得到 $\mathbf{b}$ 和 $\mathbf{c}$ 点积的一个有用的形式。我们把上式代入点积等式，将它们对应的分量相乘，再把结果相加，于是得到：

$$\mathbf{b}\cdot\mathbf{c}=|\mathbf{b}||\mathbf{c}|\cos\phi_c\cos\phi_b+|\mathbf{b}||\mathbf{c}|\sin\phi_b\sin\phi_c$$

再次应用三角函数等式,得到

$$\mathbf{b}\cdot\mathbf{c}=|\mathbf{b}||\mathbf{c}|\cos(\phi_c-\phi_b)$$

最终,对任意两个向量 **b** 和 **c**,我们有:

$$\mathbf{b}\cdot\mathbf{c}=|\mathbf{b}||\mathbf{c}|\cos(\theta) \tag{4.11}$$

其中 θ 是 **b** 和 **c** 之间的夹角。在这里,θ 是 **b** 和 **c** 的两个夹角中较小的一个。这样,$\mathbf{b}\cdot\mathbf{c}$的值随着 **b** 和 **c** 之间夹角的余弦变化而变化。同样的结果也适用于三维、四维乃至任意多维,前提是两个向量具有相同的维数。

为得到更简洁的形式,在等式两边同除以$|\mathbf{b}||\mathbf{c}|$并使用单位向量的表示形式 $\hat{\mathbf{b}}=\mathbf{b}/|\mathbf{b}|$,我们得到:

$$\cos(\theta)=\hat{\mathbf{b}}\cdot\hat{\mathbf{c}}$$

这正是我们想要的结果:两个向量 **b** 和 **c** 之间夹角的余弦等于归一化后向量的点积。

例 4.3.3

计算 $\mathbf{b}=(3,4)$和 $\mathbf{c}=(5,2)$之间的夹角。

解:因为$|\mathbf{b}|=5$,$|\mathbf{c}|=5.385$,所以 $\hat{\mathbf{b}}=(0.6,0.8)$,$\hat{\mathbf{c}}=(0.9285,0.3714)$。点积 $\hat{\mathbf{b}}\cdot\hat{\mathbf{c}}=0.85422=\cos(\theta)$,因此 $\theta=31.326°$。可以在图形计算器上画出这两个向量,测量出它们之间的夹角,从而验证我们的结果。

4.3.3 b·c 的符号和正交性

回顾一下三角函数,当$|\theta|$小于 90°时(锐角)$\cos(\theta)$是正的,$|\theta|$等于 90°时(直角)$\cos(\theta)$等于 0,当$|\theta|$大于 90°时(钝角)$\cos(\theta)$是负的。由于两个向量的点积和它们之间夹角的余弦成正比,我们可以得到以下关于两个非零向量夹角与点积的关系:

$$\begin{aligned}&\text{如果 } \mathbf{b}\cdot\mathbf{c}>0 \quad \text{夹角小于 } 90°\\&\text{如果 } \mathbf{b}\cdot\mathbf{c}=0 \quad \text{夹角等于 } 90°\\&\text{如果 } \mathbf{b}\cdot\mathbf{c}<0 \quad \text{夹角大于 } 90°\end{aligned} \tag{4.12}$$

以上结果如图 4.9 所示。许多算法测试都需要去判断点积的符号。

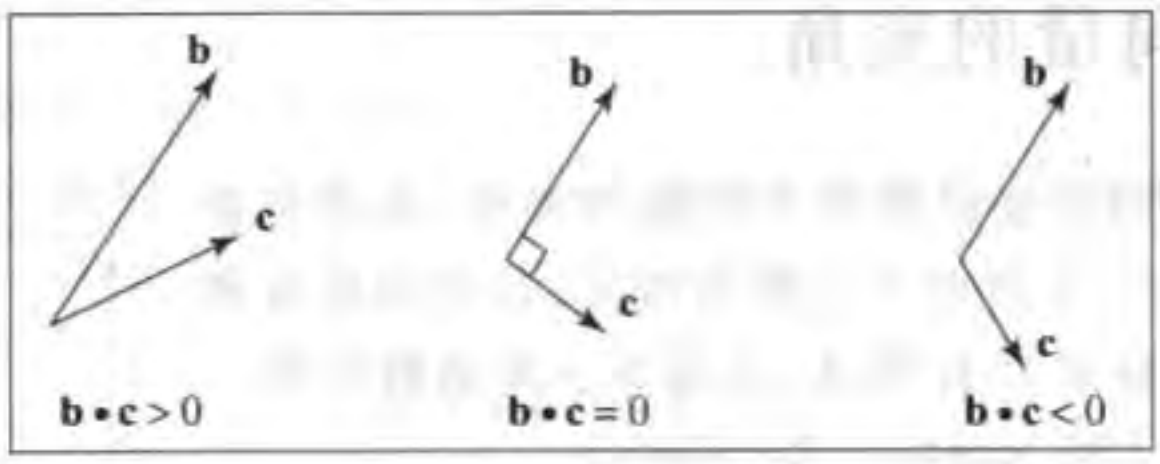


图 4.9 点积的符号

两个向量夹角等于 90°,即正交的情况,具有特殊的重要性。

定义:如果 $\mathbf{b}\cdot\mathbf{c}=0$,则称向量 **b** 和向量 **c** 是**正交**的。 (4.13)

"正交"(perpendicular)也称作"直交"(orthogonal)或"垂直"(normal),三个名称可以

互换。

最常见的正交的例子是那些沿着二维或三维坐标轴方向的向量，如图 4.10 所示。在图 4.10(a)中，二维向量(1,0)和(0,1)是相互正交的单位向量。它们的三维形式由于很常用，所以又叫做**标准单位向量**。在此我们把这三个向量称为 **i**、**j** 和 **k**。

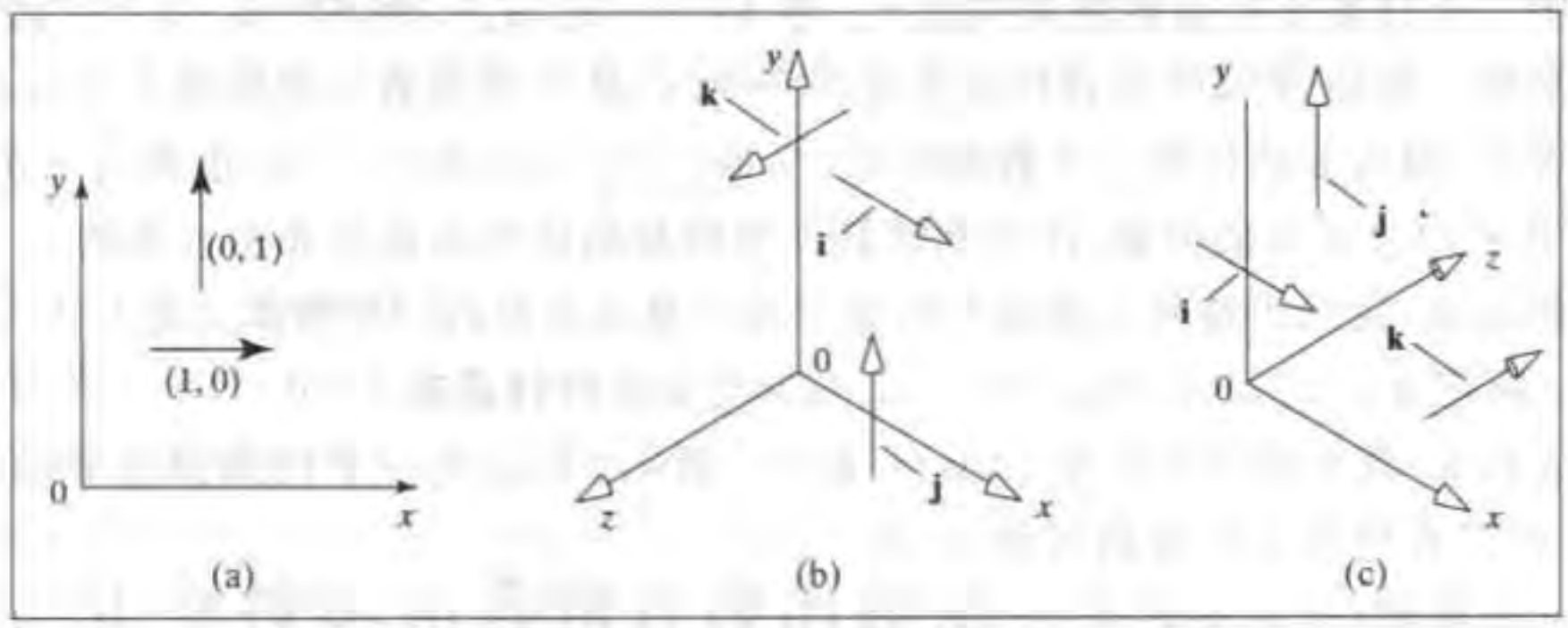


图 4.10　单位标准向量

定义：三维空间的**标准单位向量**是有如下分量的向量：

$$\mathbf{i}=(1,0,0),\quad \mathbf{j}=(0,1,0),\quad \mathbf{k}=(0,0,1) \tag{4.14}$$

图 4.10(b)是右手系的标准单位向量，图 4.10(c)是左手系的标准单位向量，注意 **k** 总是指向 z 轴正方向。

有了标准单位向量的定义，我们可以把任意一个三维向量如(a,b,c)写成另一种形式：

$$(a,b,c)=a\mathbf{i}+b\mathbf{j}+c\mathbf{k} \tag{4.15}$$

例 4.3.4

向量 $\mathbf{v}=(2,5,-1)$等价于 $2(1,0,0)+5(0,1,0)-1(0,0,1)$，后者就是 $2\mathbf{i}+5\mathbf{j}-\mathbf{k}$。

这种表示形式把一个向量表示成一组基向量的线性组合，这样会简化很多纸笔类的计算。在 4.4 节中将看到，这一形式在处理叉积时尤其方便。

练习

练习 4.3.1　求向量夹角

求向量(2,3)和(−3,1)之间的夹角，并确定它是锐角、钝角还是直角。用图形计算器验证你的结果，然后再计算三维向量(1,3,−2)和(3,3,1)的夹角。

练习 4.3.2　测试正交性

以下的向量中有哪几对是相互正交的？(3,4,1)，(2,1,1)，(−3,−4,1)，(0,0,0)，(1,−2,0)，(4,4,4)，(0,−1,4)和(2,2,1)？

练习 4.3.3　勾股定理

参见等式(4.9)和等式(4.10)，若其中的向量 **a** 和 **b** 是正交的，这两个表达式的值相等，这在几何上看起来似乎讲不通。证明这个结果事实上是对的，并且把它们与勾股定理联系起来。

4.3.4 二维正交向量

假设有二维向量 $\mathbf{a}=(a_x,a_y)$,哪些向量与它正交呢?一个方法是通过交换它的两个分量并把其中一个分量反号得到所求向量[①]。令 $\mathbf{b}=(-a_y,a_x)$,则点积 $\mathbf{a}\cdot\mathbf{b}=0$,因此 $\mathbf{a}$ 和 $\mathbf{b}$ 确实是正交的。例如,若 $\mathbf{a}=(4,7)$,那么 $\mathbf{b}=(-7,4)$是一个垂直 $\mathbf{a}$ 的向量。与 $\mathbf{a}$ 正交的向量有无穷多个,因为 $\mathbf{b}$ 的任何一个数乘结果,比如$(-21,12)$和$(7,-4)$也都与 $\mathbf{a}$ 正交。(给定 $\mathbf{a}$,写出几个与它正交的向量)注意到计算二维向量的正交向量是非常容易的。

为方便起见,我们用符号$\perp$表示与给定二维向量 $\mathbf{a}$ 正交的一个特殊向量:

定义:给定 $\mathbf{a}=(a_x,a_y)$,则 $\mathbf{a}^\perp=(-a_y,a_x)$与 $\mathbf{a}$ **逆时针正交**。 (4.16)

注意 $\mathbf{a}$ 和 $\mathbf{a}^\perp$ 具有相同的长度:$|\mathbf{a}|=|\mathbf{a}^\perp|$。图 4.11(a)是一个任意向量 $\mathbf{a}$ 以及由此得到的 $\mathbf{a}^\perp$。可以看到从 $\mathbf{a}$ 的方向转到 $\mathbf{a}^\perp$ 的方向需要向左旋转(向右旋转得到$-\mathbf{a}^\perp$ 的方向)。

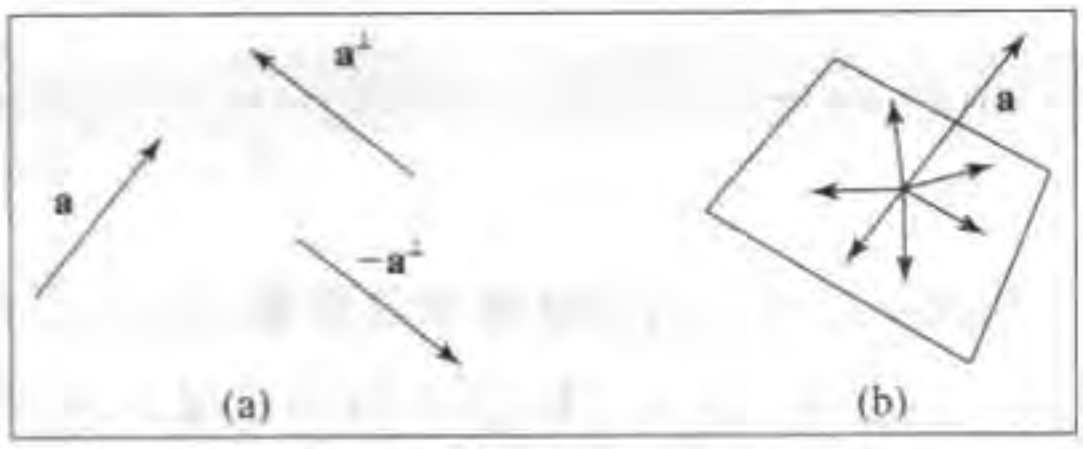


图 4.11 正交与向量 $\mathbf{a}$ 的向量 $\mathbf{a}^\perp$

在下节里我们会了解到这一表示方法的实用性。图 4.11(b)说明在三维空间中,与给定向量 $\mathbf{a}$ 正交的向量方向不再唯一,任何一个处于与 $\mathbf{a}$ 垂直的平面上的向量都符合条件。但是,后面提到的叉积会很容易处理这样的向量。

练习

练习 4.3.4 $\mathbf{a}^\perp$ 的一些有趣的属性

在某些问题中,把正交符号$\perp$看成一个对它的参数进行 90°左旋的操作符会很有用。这样 $\mathbf{a}^\perp$ 就是对 $\mathbf{a}$ 做$\perp$操作得到的向量,正像$\sqrt{x}$就是对 x 做开平方操作得到的值一样。从这个角度看,$\perp$有如下的性质:

1. 线性:$(\mathbf{a}+\mathbf{b})^\perp=\mathbf{a}^\perp+\mathbf{b}^\perp$,对任意标量 A,有$(A\mathbf{a})^\perp=A\mathbf{a}^\perp$。
2. $\mathbf{a}^{\perp\perp}=(\mathbf{a}^\perp)^\perp=-\mathbf{a}$(两次正交等于求反)。

练习 4.3.5 正交点积

如果用一个向量的正交向量与另一个向量做点积,如 $\mathbf{a}^\perp\cdot\mathbf{b}$,会发生有趣的现象。我们把这样的点积叫做正交点积【Hill95】。由 $\mathbf{a}^\perp$ 的基本定义可得:

- $\mathbf{a}^\perp\cdot\mathbf{b}=a_xb_y-a_yb_xZ$ (正交点积的值)
- $\mathbf{a}^\perp\cdot\mathbf{a}=0$ ($\mathbf{a}^\perp$ 正交于 $\mathbf{a}$)
- $|\mathbf{a}^\perp|^2=|\mathbf{a}|^2$ ($\mathbf{a}^\perp$ 和 $\mathbf{a}$ 具有相同的长度)
- $\mathbf{a}^\perp\cdot\mathbf{b}=-\mathbf{b}^\perp\cdot\mathbf{a}$ (反对称性) (4.17)

第四个等式说明正交点积是反对称的:将正交算符$\perp$从一个向量移到另一个向量将改

① 这等价于一个熟知的事实:相互垂直直线的斜率互为负倒数。在第 5 章中我们会看到"交换并取反"的操作和旋转 90 度有很自然的联系。

变点积的正负号。后续章节中我们会在必要时讨论正交点积的其他性质。

练习 4.3.6　计算正交点积

计算 $\mathbf{a}\cdot\mathbf{b}$ 和 $\mathbf{a}^{\perp}\cdot\mathbf{b}$，其中 $\mathbf{a}=(3,4)$，$\mathbf{b}=(2,1)$。

练习 4.3.7　行列式

证明 $\mathbf{a}^{\perp}\cdot\mathbf{b}$ 可以被写成行列式形式（矩阵和行列式的定义见附录 2）：

$$\mathbf{a}^{\perp}\cdot\mathbf{b}=\begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}$$

练习 4.3.8　其他性质

(1) 证明：$(\mathbf{a}^{\perp}\cdot\mathbf{b})^2+(\mathbf{a}\cdot\mathbf{b})^2=|\mathbf{a}|^2|\mathbf{b}|^2$。

(2) 证明：如果 $\mathbf{a}+\mathbf{b}+\mathbf{c}=0$，那么 $\mathbf{a}^{\perp}\cdot\mathbf{b}=\mathbf{b}^{\perp}\cdot\mathbf{c}=\mathbf{c}^{\perp}\cdot\mathbf{a}$。

4.3.5　正交投影和点到直线的距离

在图形学中经常出现这样三个特别的几何问题：

- 把一个向量**投影**到另一个向量上
- 把一个向量**分解**成不同方向上的分量
- 找到一个点到一条直线的距离

如果我们使用正交向量和正交点积，这三个问题会大大简化。前面提到使用正交点积的前提条件是有关的向量都是二维的。

如图 4.12(a)，已知两点 A，C 和一个向量 $\mathbf{v}$，问题如下：

(1) 点 C 到经过点 A 且方向与 $\mathbf{v}$ 平行的直线 L 的距离是多少？

(2) 从 C 向 L 做垂线，垂足在哪里？

(3) 怎样将向量 $\mathbf{c}$ 分解为两个部分，一个沿着 L 的方向，一个沿着与 L 垂直的方向？

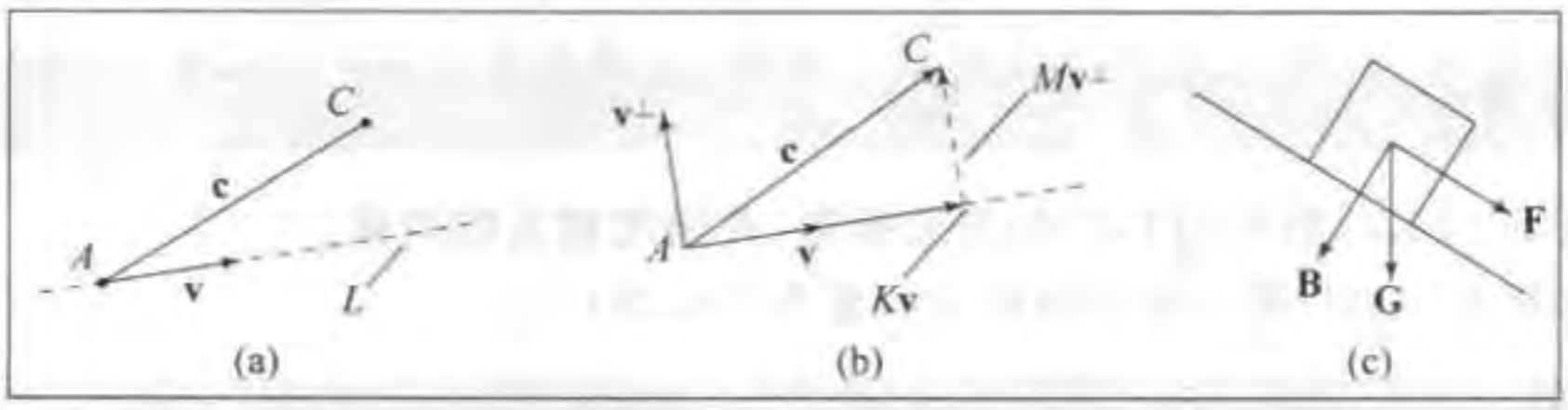


图 4.12　把一个向量分解为两个正交的向量

图 4.12(b)定义了更多的量：$\mathbf{v}^{\perp}$ 由向量 $\mathbf{v}$ 逆时针旋转 90 度得到。从 $\mathbf{c}$ 向 L 做垂线，可以将 $\mathbf{c}$ 分解成沿 $\mathbf{v}$ 方向的部分 $K\mathbf{v}$，和垂直 $\mathbf{v}$ 方向的部分 $M\mathbf{v}^{\perp}$，其中 K 和 M 是待定的常数。我们有：

$$\mathbf{c}=K\mathbf{v}+M\mathbf{v}^{\perp} \tag{4.18}$$

给定 $\mathbf{c}$ 和 $\mathbf{v}$，我们想得到 K 和 M。求出这两个值后，我们可以说 $\mathbf{c}$ 到 $\mathbf{v}$ 的正交投影是 $K\mathbf{v}$ 并且点 C 到直线的距离是 $|M\mathbf{v}^{\perp}|$。

图 4.12(c)给出了产生上述问题的一个实例。我们想分析重力向量 $\mathbf{G}$ 如何作用在物体上使它沿斜面下滑的。为了解决这个问题，我们需要将 $\mathbf{G}$ 分解成沿斜面的力 $\mathbf{F}$ 和垂直斜面

的力 **B**,也就是说,我们必须找到 **F** 和 **B**,使得 **G**=**F**+**B**。

等式(4.18)实际上是两个等式:等式左右两边的 x 分量和 y 分量必须分别相等。等式中含有两个未知数 K 和 M,因此我们有两个未知数和两个方程,可以应用克莱默法则。可是谁还记得什么是克莱默法则?在这里我们将使用一个便于记忆的巧妙方法快速地求解我们的未知数。它和克莱默法则是等价的,但是用起来更简单。

求解含两个未知数的两个方程的方法是消去一个未知数,我们的方法是让等式两边同时与 **v** 做点积:

$$\mathbf{c}\cdot\mathbf{v}=K\mathbf{v}\cdot\mathbf{v}+M\mathbf{v}^{\perp}\cdot\mathbf{v} \tag{4.19}$$

含 $\mathbf{v}^{\perp}\cdot\mathbf{v}$ 的一项令人满意地消失了(为什么?),我们立即得到了 K:

$$K=\frac{\mathbf{c}\cdot\mathbf{v}}{\mathbf{v}\cdot\mathbf{v}}$$

同理,让等式(4.18)两边同时与 $\mathbf{v}^{\perp}$ 做点积可以得到 M:

$$M=\frac{\mathbf{c}\cdot\mathbf{v}^{\perp}}{\mathbf{v}^{\perp}\cdot\mathbf{v}^{\perp}}$$

合并以上结果,我们得到:

$$\mathbf{c}=\left(\frac{\mathbf{v}\cdot\mathbf{c}}{|\mathbf{v}|^2}\right)\mathbf{v}+\left(\frac{\mathbf{v}^{\perp}\cdot\mathbf{c}}{|\mathbf{v}^{\perp}|^2}\right)\mathbf{v}^{\perp}\quad(\text{把 }\mathbf{c}\text{ 分解到 }\mathbf{v}\text{ 和 }\mathbf{v}^{\perp}\text{ 方向上}) \tag{4.20}$$

上述等式对任意向量 **c** 和 **v** 都成立。沿 **v** 方向的部分又叫做向量 **c** 投影到向量 **v** 的**正交投影**。注意如果 **c** 与 **v** 正交,投影会消失。等式的第二项,即垂直 **v** 的部分,准确简明地给出了 **c** 与 **v**"不同的部分",它的大小就是 **c** 到直线的距离:

$$\text{距离}=\left|\frac{\mathbf{v}^{\perp}\cdot\mathbf{c}}{|\mathbf{v}^{\perp}|^2}\mathbf{v}^{\perp}\right|=\frac{|\mathbf{v}^{\perp}\cdot\mathbf{c}|}{|\mathbf{v}^{\perp}|}$$

(检查一下上式右边第二种形式是否和第一种形式相等。)根据图 4.12(b),我们可以说:点 C 到经过点 A 且方向为 **v** 的直线的距离是:

$$\text{距离}=\frac{|\mathbf{v}^{\perp}\cdot(C-A)|}{|\mathbf{v}^{\perp}|} \tag{4.21}$$

例 4.3.5

求向量 **c**=(6,4)到 **v**=(1,2)的正交投影,并画出相关的向量。

解:由等式(4.20)第一项可得所求向量为(14,28)/5。

例 4.3.6

求点 C=(6,4)到过点(1,1)和(4,9)的直线的距离。

解:令 A=(1,1),则 **c**=(5,3),**v**=(4,9)−(1,1)=(3,8)。由等式(4.21)可得距离 $d=31/\sqrt{73}$。

练习

练习 4.3.9 分解

把向量 **g**=(4,7)表示为 **b**=(3,5)和 $\mathbf{b}^{\perp}$ 的线性组合。点(4,2)+**g** 到经过点(4,2)且沿 **b** 方向的直线的距离是多少?

练习 4.3.10　把方块拉下斜面

一个方块停放在与水平面成 30°角的斜面上，方块受到得重力为 1 牛顿。那么它受到沿斜面方向的力是多大？

练习 4.3.11　求距离

点(6,11)到经过点(2,5)和(4,−1)的直线的距离是多少？在图形计算器上验证结果。

4.3.6　投影的应用：反射

为了显示光线经过镜面反射的过程，或者台球相互碰撞的过程，我们需要计算出对象被给定表面反射后的运动方向。在本章末尾的案例中，我们将介绍一种应用，它可以跟踪一条在四面反光的密室里来回反射的光线，或者跟踪一个在台球桌上到处反弹的台球的轨迹。每一次反弹，都需要计算一个新的反射方向，本节将告诉我们如何计算这个方向。

当光线被镜面反射时，通过物理学我们知道反射角必须等于入射角。下面我们将介绍如何用向量和投影计算这个新的方向。

有许多不同的方法可以得到反射方向，在这里我们选择【Hearn94】中提出的一种较直观的方法。为了简便，我们可以只考虑二维向量，因为这一方法对向量的维数没有要求，二维的结果可以直接应用到三维。

图 4.13(a)给出了一条从光源出发，沿方向 **a** 射出的光线，遇到直线 L 后沿未知方向 **r** 射出(注意有些作者倾向于认为 **a** 是朝向光源的方向，这相当于本例的 −**a**)。**r** 和 **a** 具有相同的长度，这是由于理想镜面反射的入射光和反射光具有相同的光强。接下来我们寻找 **r** 的方向。向量 **n** 的方向与直线垂直，图中角 θ_1 的大小必须等于角 θ_2，在极限情况下，当 θ_1 为 90°时(光线贴着表面擦过)，θ_2 也必须为 90°；当 θ_1 为 0°时(光线垂直表面入射)，θ_2 也必须为 0°(我们最终的反射等式必须能应用于各种角度的入射角，所以也必须满足这些极限情况)。一个类似的情况如图 4.13(b)所示，除了把向量 −**a** 加到了向量 **r** 上。(先求出 **r**−**a** 可以简化问题，因为这样可以立即得到 **r**。)根据平行四边形法则，向量 **r**−**a** 与 **n** 平行(验证这一点)。

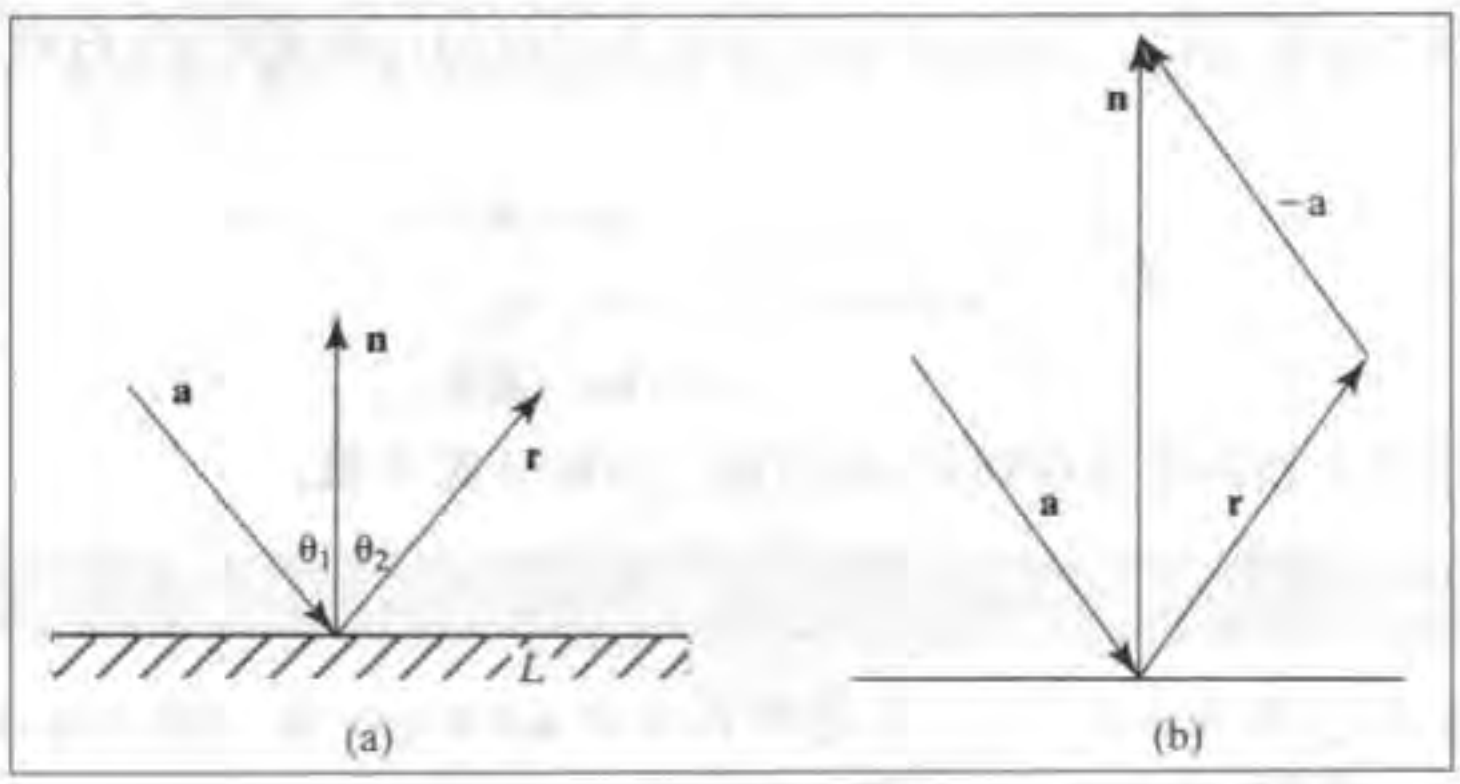


图 4.13　光线被平面反射的过程

为了简便，假设向量 **a** 和 **n** 已经被归一化，那么 **a** 在 **n** 上的射影为 −**a**·**n**，从图中可以

看出向量 $\mathbf{r}-\mathbf{a}$ 是在这一射影基础上又加上一个 $-\mathbf{a}\cdot\mathbf{n}$,也就是说 $\mathbf{r}-\mathbf{a}$ 是沿方向 $\mathbf{n}$,长度为 $2(-\mathbf{a}\cdot\mathbf{n})$ 的向量,于是:

$$\mathbf{r}-\mathbf{a}=2(-\mathbf{a}\cdot\mathbf{n})\mathbf{n},\text{最终:}$$

$$\mathbf{r}=\mathbf{a}-2(\mathbf{a}\cdot\mathbf{n})\mathbf{n}\quad(\text{反射方向})\tag{4.22}$$

在三维中,物理学要求反射方向 $\mathbf{r}$ 必须位于由 $\mathbf{n}$ 和 $\mathbf{a}$ 决定的平面上。在第 5 章可以看到,以上 $\mathbf{r}$ 的表达式实际上满足这个条件。

等式(4.22)的结果看上去有些复杂,不便于记忆,但是花些时间记住它对后面的讨论很有帮助。粗略地看上去,它说明反射向量 $\mathbf{r}$ 和 $\mathbf{a}$ 基本一样,只不过减掉了"一部分"$\mathbf{n}$。这一部分的大小在 0($\mathbf{a}$ 和 $\mathbf{n}$ 正交)和 2($\mathbf{a}$ 和 $\mathbf{n}$ 平行)之间(你能验证这一结论吗?)。

例 4.3.7

令 $\mathbf{a}=(4,-2)$,$\mathbf{n}=(0,3)$。由等式(4.22)可得 $\mathbf{r}=(4,2)$。入射角和反射角都等于 $\tan^{-1}(2)$。

练习

练习 4.3.12 求反射方向

对于 $\mathbf{a}=(2,3)$,$\mathbf{n}=(-2,1)$,求反射方向。

4.4 两个向量的叉积

两个向量的叉积(又叫向量积)是另一个三维向量。它有许多有用的属性,但最常用的一个是它与原来的两个向量都正交。叉积只对三维向量有意义。

给定三维向量 $\mathbf{a}=(a_x,a_y,a_z)$ 和 $\mathbf{b}=(b_x,b_y,b_z)$,它们的叉积表示为 $\mathbf{a}\times\mathbf{b}$。它的定义用到了标准单位向量 $\mathbf{i},\mathbf{j},\mathbf{k}$:

$\mathbf{a}\times\mathbf{b}$ 的定义:

$$\mathbf{a}\times\mathbf{b}=(a_yb_z-a_zb_y)\mathbf{i}+(a_zb_x-a_xb_z)\mathbf{j}+(a_xb_y-a_yb_x)\mathbf{k}\tag{4.23}$$

这个形式很难记忆,所以它也经常被写作容易记忆的行列式形式(行列式的相关内容见附录 2):

$$\mathbf{a}\times\mathbf{b}=\begin{vmatrix}\mathbf{i} & \mathbf{j} & \mathbf{k}\\ a_x & a_y & a_z\\ b_x & b_y & b_z\end{vmatrix}\tag{4.24}$$

于是只要记住了如何求解行列式,就记住了如何计算叉积。

例 4.4.1

对于 $\mathbf{a}=(3,0,2)$ 和 $\mathbf{b}=(4,1,8)$,直接计算可得 $\mathbf{a}\times\mathbf{b}=-2\mathbf{i}-16\mathbf{j}+3\mathbf{k}$,$\mathbf{b}\times\mathbf{a}$ 等于什么?

从叉积定义中我们可以总结出以下代数性质:

1a. $\mathbf{i}\times\mathbf{j}=\mathbf{k}$

1b. $\mathbf{j}\times\mathbf{k}=\mathbf{i}$

1c. $\mathbf{k}\times\mathbf{i}=\mathbf{j}$

2. $\mathbf{a}\times\mathbf{b}=-\mathbf{b}\times\mathbf{a}$　　（反对称性）

$=\mathrm{a}\cdot(\mathrm{a}\times\mathrm{b})=-()\mathrm{a}\cdot(\mathrm{b}\times\mathrm{a})$

3. $\mathbf{a}\times(\mathbf{b}+\mathbf{c})=\mathbf{a}\times\mathbf{b}+\mathbf{a}\times\mathbf{c}$　　（线性）　　(4.25)

4. $(s\mathbf{a})\times\mathbf{b}=s(\mathbf{a}\times\mathbf{b})$　　（同质性）

5. $\mathbf{a}\cdot(\mathbf{a}\times\mathbf{b})=\mathbf{b}\cdot(\mathbf{a}\times\mathbf{b})=0$　　（正交性）

以上性质在左手坐标系和右手坐标系下都适用。注意等式 $\mathbf{i}\times\mathbf{j}=\mathbf{k}$ 中向量的逻辑（字母）顺序，这可以帮助我们记忆叉积的方向。性质 5 指出 **a** 和 **b** 的叉积与 **a**、**b** 两个向量都正交。这个性质会非常有用，比如我们想在三维场景中摆放摄像机以拍到我们最想要的画面，就要用到这一性质。在第 5 章和第 7 章这一点会变得很重要。

练习

练习 4.4.1　证明 4 个性质

证明上面提到的点积的性质 2、3、4。

练习 4.4.2　$\mathbf{a}\times\mathbf{b}$ 是否与 **a、**b** 正交？**

证明 **a** 和 **b** 的叉积与 **a**、**b** 两个向量都正交（提示，不要使用分量来证，本题可以更简单地证明）。

练习 4.4.3　叉积不满足结合律

证明叉积不满足结合律，也就是说，$\mathbf{a}\times(\mathbf{b}\times\mathbf{c})$ 不一定等于 $(\mathbf{a}\times\mathbf{b})\times\mathbf{c}$。

练习 4.4.4　另一个有用的性质

直接对分量进行计算，证明叉积的长度满足：

$$|\mathbf{a}\times\mathbf{b}|=\sqrt{|\mathbf{a}|^2\,|\mathbf{b}|^2-(\mathbf{a}\cdot\mathbf{b})^2}$$

4.4.1　叉积的几何解释

由叉积的定义可知两个向量的叉积 $\mathbf{a}\times\mathbf{b}$ 是另一个向量，但是这个向量与原来的两个向量在几何上有什么关系，我们为什么对它感兴趣呢？图 4.14 给出了答案。叉积 $\mathbf{a}\times\mathbf{b}$ 有下列有用的性质：

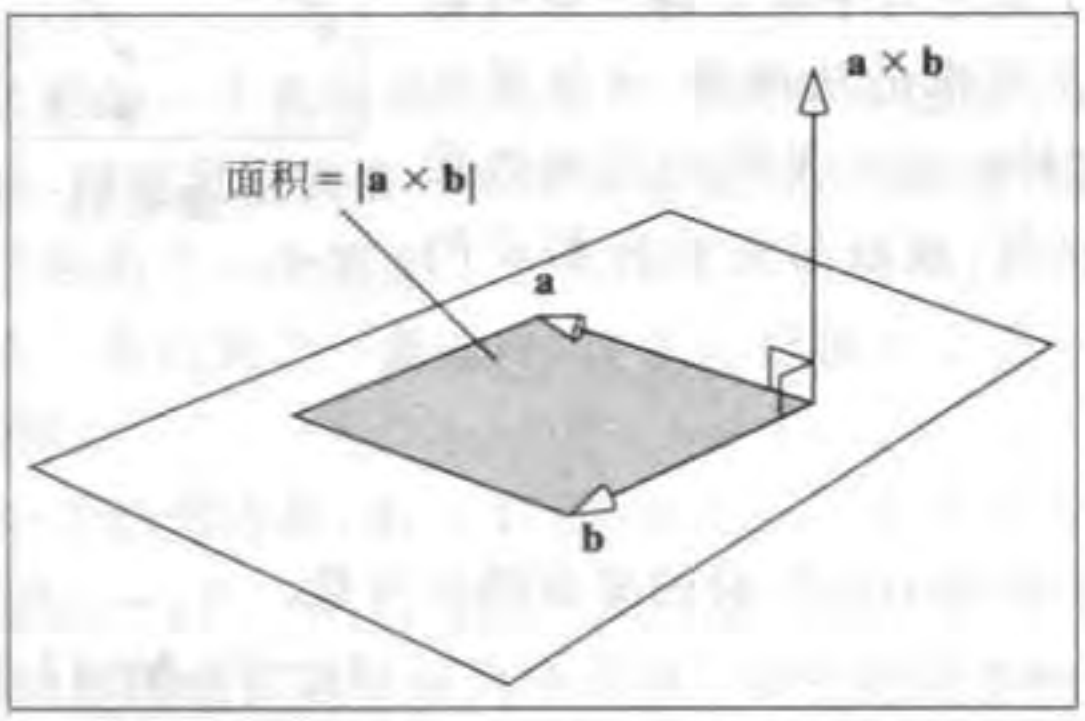


图 4.14　叉积的几何解释

1. $\mathbf{a}\times\mathbf{b}$ 与 $\mathbf{a}$、$\mathbf{b}$ 两个向量都正交,如前所述。

2. $\mathbf{a}\times\mathbf{b}$ 的长度等于由 $\mathbf{a}$ 和 $\mathbf{b}$ 决定的平行四边形的面积,这个面积大小为:

$$|\mathbf{a}\times\mathbf{b}|=|\mathbf{a}||\mathbf{b}|\sin(\theta) \tag{4.26}$$

其中 θ 是 $\mathbf{a}$ 与 $\mathbf{b}$ 的夹角,这个夹角是从 $\mathbf{a}$ 到 $\mathbf{b}$ 还是从 $\mathbf{b}$ 到 $\mathbf{a}$ 取决于哪一个小于 180 度。一个特殊情况是 $\mathbf{a}\times\mathbf{b}=0$,当且仅当 $\mathbf{a}$ 和 $\mathbf{b}$ 方向相同或相反,或者任意一个是零向量。如果 $\mathbf{a}$ 与 $\mathbf{b}$ 正交,它们叉积的长度是多少?

3. 在右手系中,$\mathbf{a}\times\mathbf{b}$ 可由右手法则得到。例如,将右手四指从 $\mathbf{a}$ 握向 $\mathbf{b}$,$\mathbf{a}\times\mathbf{b}$ 的方向就是大拇指所指的方向。(在左手系下,就要使用左手法则。)注意 $\mathbf{i}\times\mathbf{j}=\mathbf{k}$ 满足这一点。

例 4.4.2

令 $\mathbf{a}=(1,0,1)$ 和 $\mathbf{b}=(1,0,0)$。这两个向量很容易画出来,因为它们都在 xz 平面上(把它们画出来)。容易看出 $\mathbf{a}$ 和 $\mathbf{b}$ 定义的平行四边形面积为 1。因为 $\mathbf{a}\times\mathbf{b}$ 与 $\mathbf{a}$、$\mathbf{b}$ 都正交,它应该与 y 轴平行,与 $\pm\mathbf{j}$ 成正比。无论在左手还是右手系下,用相应的手从 $\mathbf{a}$ 握向 $\mathbf{b}$,拇指都会指向 y 轴正方向。通过(4.23)式直接计算可以验证:$\mathbf{a}\times\mathbf{b}=\mathbf{j}$。

练习

练习 4.4.5 证明性质

证明上面提到的叉积的三个性质。

4.4.2 求平面的法向量

在下一节中我们会看到,有时我们必须求一个平面的法向量 $\mathbf{n}$ 的各个分量。假设该平面经过三个已知点,我们可以通过叉积求这个法向量。

任意三个不共线的点 P_1、P_2、P_3 可以确定一个平面,如图 4.15 所示。

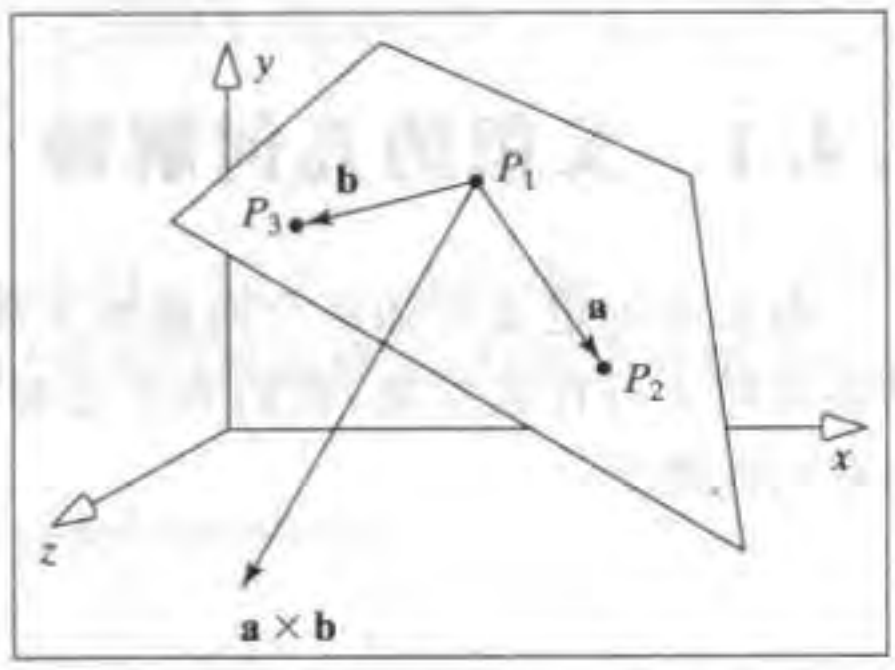


图 4.15 三点确定一个平面

为找到它的法向量,我们建立两个向量,$\mathbf{a}=P_2-P_1$ 和 $\mathbf{b}=P_3-P_1$。它们的叉积 $\mathbf{n}$,表示为 $\mathbf{a}\times\mathbf{b}$,必然与 $\mathbf{a}$、$\mathbf{b}$ 都正交,于是它与平面上每一条线都垂直(为什么?)。它就是所求的法向量。(如果三个点在一条直线上会怎样?)这个叉积的任何数乘结果还是该平面的法向量,包括与它方向相反的 $\mathbf{b}\times\mathbf{a}$。

例 4.4.3

求过点(1,0,2)、(2,3,0)、(1,2,4)的平面的法向量。

解:直接计算可得,$\mathbf{a}=(2,3,0)-(1,0,2)=(1,3,-2)$,$\mathbf{b}=(1,2,4)-(1,0,2)=(0,2,2)$,因此它们的点积 $\mathbf{n}=(10,-2,2)$。(注意正如我们所期望的,$\mathbf{n}$ 与 $\mathbf{a}$、$\mathbf{b}$ 都正交。)

注意：由于叉积运算用到了减法，所以计算 **n** 的过程会受到计算机表示数字精度的影响，当 **a**、**b** 夹角很小时影响尤为明显。我们在练习中给出了一种更加鲁棒的法向量计算方法。

练习

练习 4.4.6　点的顺序的选择会对结果有影响吗？

在例 4.4.3 中，如果我们用不同的顺序选择点，比如说令 $\mathbf{a}=(1,0,2)-(2,3,0)$，$\mathbf{b}=(1,2,4)-(2,3,0)$，还会产生相同的平面吗？证明可以得到相同的平面。

练习 4.4.7　求平面

对下面每一个点的三元组，求出经过三点的平面（若存在）的法向量。

(1) $P_1=(1,1,1)$，$P_2=(1,2,1)$，$P_3=(3,0,4)$

(2) $P_1=(8,9,7)$，$P_2=(-8,-9,-7)$，$P_3=(1,2,1)$

(3) $P_1=(6,3,-4)$，$P_2=(0,0,0)$，$P_3=(2,1,-1)$

(4) $P_1=(0,0,0)$，$P_2=(1,1,1)$，$P_3=(2,2,2)$

练习 4.4.8　求法向量

计算图 4.16 中几何物体每个面的法向量。其中立方体的顶点为$(\pm1,\pm1,\pm1)$，四面体的顶点为$(0,0,0)$，$(0,0,1)$，$(1,0,0)$和$(0,1,0)$。

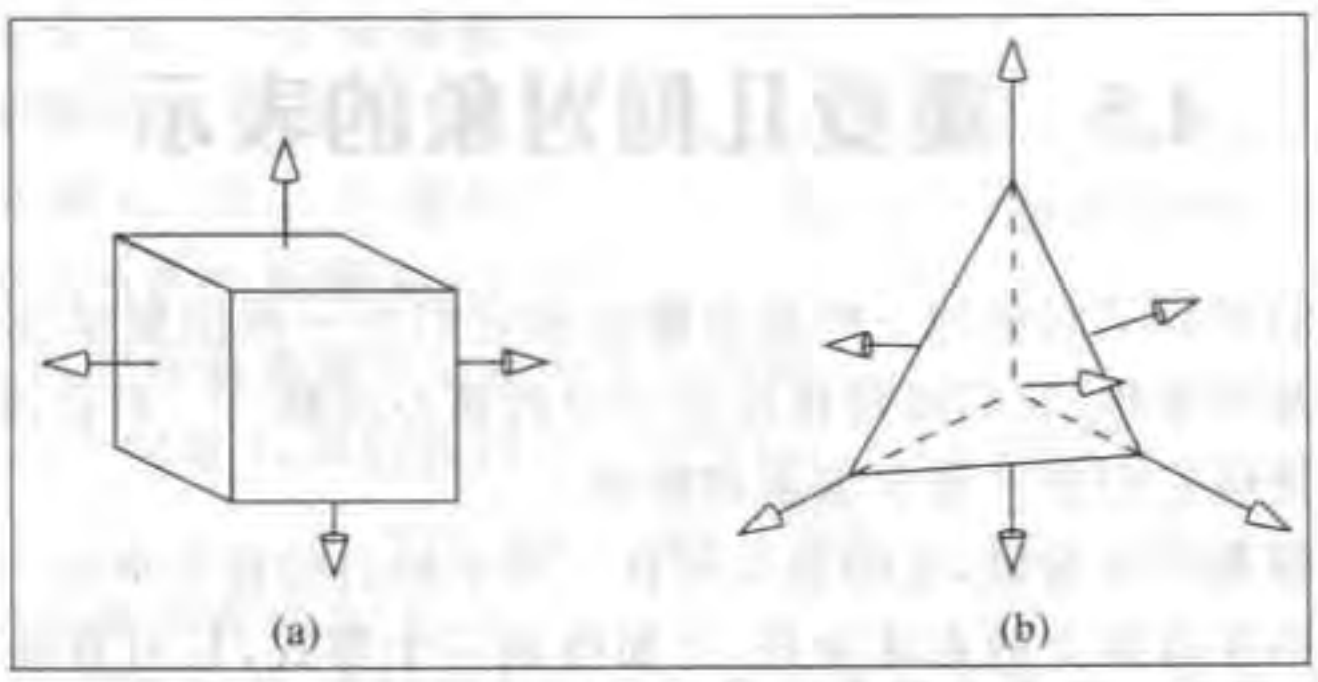


图 4.16　计算每个面的法向量

4.4.3　判断平面多边形的凸性

如第 2 章所述，多边形的一个重要属性是它是否是凸多边形。这个问题在后面讲到裁剪、明暗处理时还会碰到，因此我们需要一种简单有效的方法判断多边形的凸性。

幸运的是，叉积已经给出了一个答案。它利用了凸多边形这样的一个属性：遍历一个凸多边形的所有边，当从一条边到下一条边时，在多边形顶点会有一次左转或右转，它们必然都是左转或者都是右转。

图 4.17(a)给出了一个凸多边形，图 4.17(b)给出了一个非凸多边形。图 4.17(a)中的六边形有六个顶点 $P_0,P_1,\cdots,P_5$，从 P_0 到 P_1 的边被标记为向量 **a**。类似地，向量 **b** 是从 P_1 到 P_2 的边。这样多边形的边总共形成六个向量。我们把这些向量叫做**边向量**。注意从多边形的每条边到下一条边都是一个右转。对于每一个右转，叉积 $\mathbf{a}\times\mathbf{b}$ 指向多边形所在平

面的内侧;对于每一个左转,叉积指向平面外侧。你能看出为什么吗?注意在图 4.17(b)中,这些转向不是相同的:其中有 5 个左转,还有 5 个右转。由于它们不相同,多边形也不是凸的。以上的分析给出了如下结果:

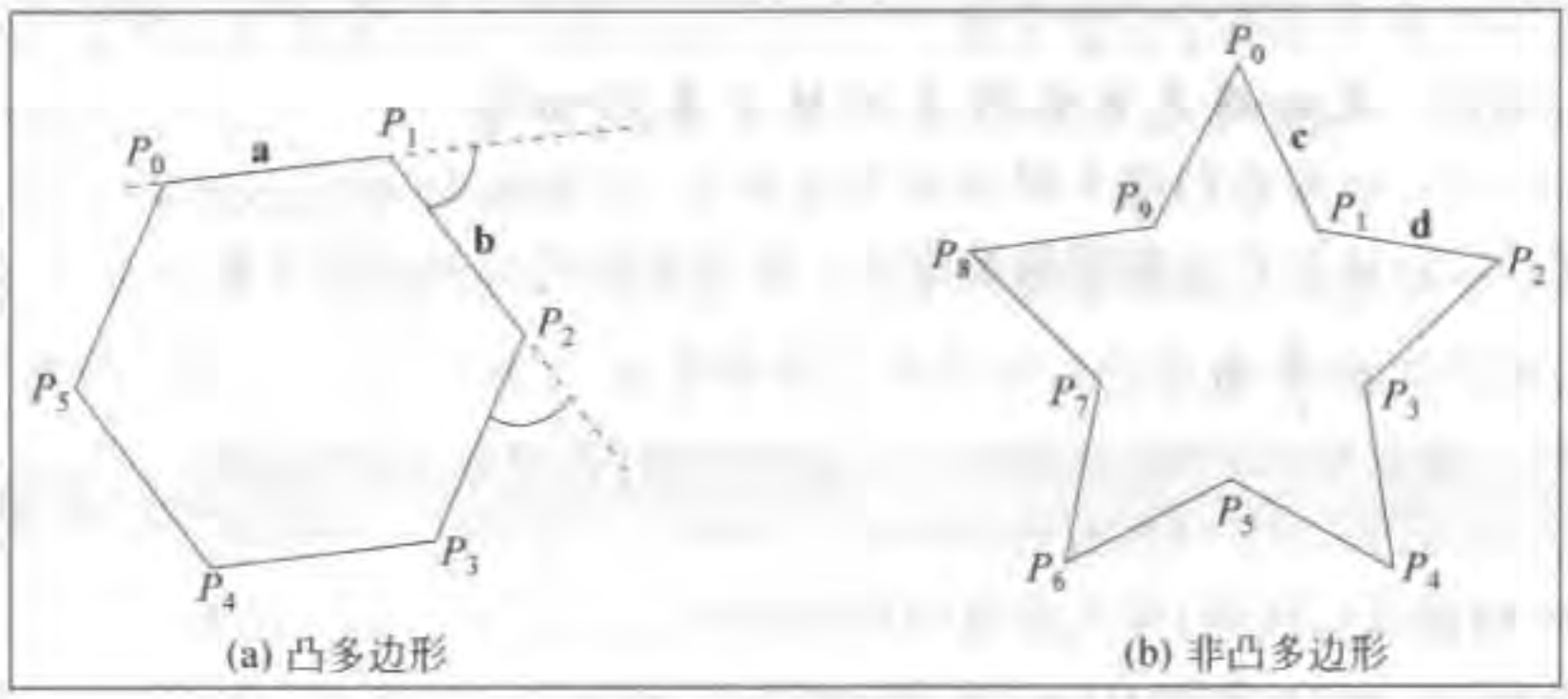


图 4.17 判断多边形的凸性

如果一个多边形所有相邻边向量的叉积都指向平面内侧或者都指向平面外侧,那么这个多边形是凸多边形。

实现这个判断需要计算 n 次叉积并判断这些叉积结果的 y 分量是否有相同的正负号。

4.5 重要几何对象的表示

在前几节中我们回顾了向量的一些基本概念和它们在一些图形学几何问题中的应用。现在开始介绍一些基本思想,它们将简化图形学中的核心问题——对直线和平面的操作,而"直"和"平"的特性使得它们非常易于表示和操作。

表示直线和平面是什么意思,它的意义何在?表示的目的在于得到一个等式或方程,能将在直线上的点和不在直线上的点区分开,比如得到一个等式,只对直线上的点成立;或者是一个方程,随着某个参数的变化返回直线上所有的点。用这个表示方法可以做出如下判断:点 P 是否在直线上?直线与另一条直线或者其他物体的交点在哪里?还有一个非常重要的问题,一条直线会把它所在的平面分成两个部分,我们经常需要知道,某个点 P 在这两个部分中的哪一个里面。

为了能正确地处理直线和平面,我们必须(有些出乎意料地)回顾一些基本概念如点和向量的区别,以及它们各自的表示方法。需要这么做的原因是,为了表示直线和平面,我们需要对点进行相加和数乘,而这些操作对点来说没有直观的意义。为了解释这一点我们将引入坐标框架的表示方法,它可以让我们认清点和向量的明显区别,并告诉我们什么时候对点进行相加操作是合理的。坐标框架的使用将最终引入**齐次坐标**,后者作为计算机图形学中非常重要的工具,将大大简化许多算法。在本书中我们将只在几个地方详细描述坐标框架的使用,主要是关于坐标系变换和摄像机围绕场景运动[①](见第 5、6、7 章)。但是即使没

① 这是一个图形学编程人员很容易误入歧途的地方:他们渲染的结果在简单场景下看起来正常,场景一旦复杂起来,就会出现莫名其妙和令人眼花缭乱的错误。

有明确指出，在每种情况下也都会有一个基本的坐标框架。

4.5.1　坐标系统和坐标框架

在前几节讨论向量的时候我们认为一个向量，如 $\mathbf{v}=(3,2,7)$ 就是一个确定的三元组。对于一个点 $P=(5,3,1)$ 也是如此。这样看上去点和向量是相同的。但实际上点和向量是完全不同的事物：点有确切位置但没有大小和方向；向量有大小和方向但没有确切的位置。

当然，$\mathbf{v}=(3,2,7)$ 就是一个在某坐标系下分量为 $(3,2,7)$ 的向量 $\mathbf{v}$。类似地，$P=(5,3,1)$ 就是一个在某坐标系下坐标为 $(5,3,1)$ 的点。一般情况下，这种对象和其表示形式之间的不清晰不会带来太大的问题。但是在不止一个坐标系的情况下（这在图形学中很常见），问题就显现出来了，尤其是当你想把点或向量从一个坐标系转换到另一个坐标系的时候。

我们通常认为一个坐标系就是从原点出发的三个轴，如图 4.2(b) 所示。但实际上，坐标系是位于“世界”中某一位置的，而它的三个轴最好表示成三个两两正交的向量。在某些特殊情况中，确定坐标系的确切位置是很重要的。所以我们把三维坐标系统[①]推广为三维坐标框架。一个**坐标框架**包含一个叫做原点的特定点 ϕ，和三个两两正交的单位向量 $\mathbf{a}$、$\mathbf{b}$ 和 $\mathbf{c}$。图 4.18 给出了一个位于世界中某个点 ϕ 的坐标框架，并把它的三个向量 $\mathbf{a}$、$\mathbf{b}$、$\mathbf{c}$ 画成看起来像是从点 ϕ 出发的轴。

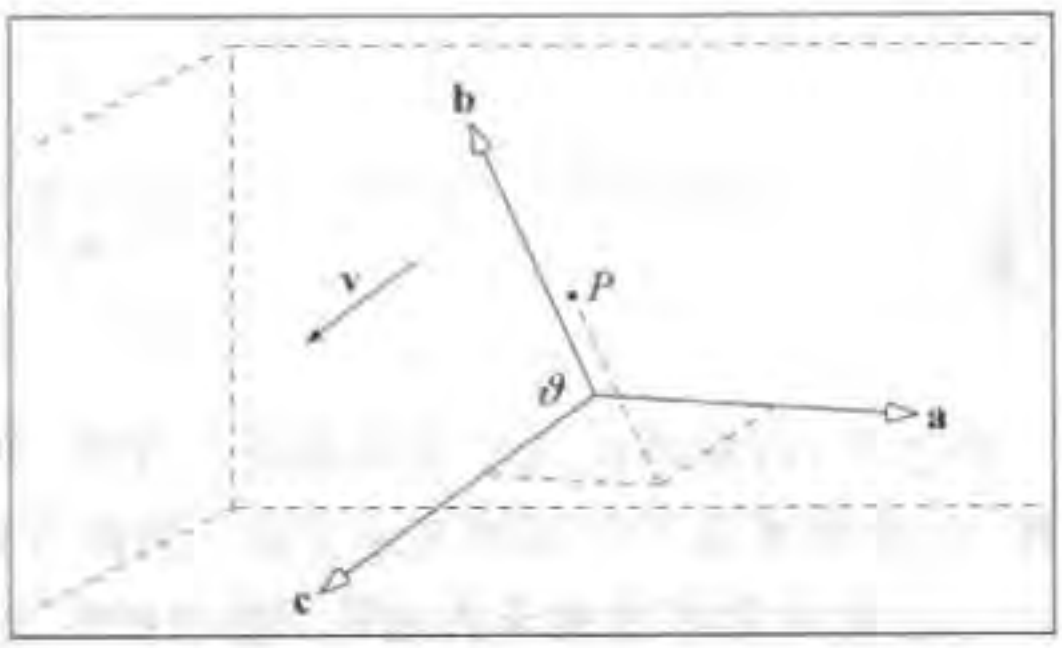


图 4.18　“世界”中的一个坐标框架

现在，为了表示一个向量 $\mathbf{v}$，我们找到三个数字 (v_1, v_2, v_3) 使得

$$\mathbf{v} = v_1\mathbf{a} + v_2\mathbf{b} + v_3\mathbf{c} \tag{4.27}$$

并认为 $\mathbf{v}$ 在这个系统中表示为 (v_1, v_2, v_3)。

另一方面，为表示一个点 P，我们把它的位置看成是原点通过一定量的偏移产生的：我们用三个数字 (p_1, p_2, p_3) 表示向量 $P-\phi$：

$$P - \phi = p_1\mathbf{a} + p_2\mathbf{b} + p_3\mathbf{c}$$

于是可以把 P 写做：

$$P = \phi + p_1\mathbf{a} + p_2\mathbf{b} + p_3\mathbf{c} \tag{4.28}$$

这样点 P 的表示不仅是一个三元组，而是一个三元组加上一个原点，说明 P 处于从原点偏移 $p_1\mathbf{a}+p_2\mathbf{b}+p_3\mathbf{c}$ 的位置上。这样做是为了让坐标系的原点“明确化”。这在有多个坐标框架和从一个坐标框架向另一个做转换时非常重要。

注意我们之前定义标准单位向量 $\mathbf{i}=(1,0,0)$、$\mathbf{j}=(0,1,0)$、$\mathbf{k}=(0,0,1)$ 时，实际上把它们的表示形式定义在一个潜在的坐标框架下。通过等式(4.27)得 $\mathbf{i}=1\mathbf{a}+0\mathbf{b}+0\mathbf{c}$，向量 $\mathbf{i}$ 就是 $\mathbf{a}$ 本身！这只是一个命名的问题——你是在讨论这个向量还是讨论它在一个坐标框架下的表示。我们一般不必区分它们。

---

① 同样适用于二维系统。

注意你不能明确的指出 ϕ 的位置或者 **a**、**b**、**c** 的方向,除非有另外的坐标框架来表示现有的这个。从现有的坐标框架看,ϕ 可表示为(0,0,0),**a** 可表示为(1,0,0)等等。

点和向量的齐次表示

使用相同的基本对象($\mathbf{a},\mathbf{b},\mathbf{c},\phi$)表示点和向量是非常有用的。由等式(4.27)和等式(4.28),向量 $\mathbf{v}=v_1\mathbf{a}+v_2\mathbf{b}+v_3\mathbf{c}$ 相当于有四个分量($v_1,v_2,v_3,0$),点 $P=p_1\mathbf{a}+p_2\mathbf{b}+p_3\mathbf{c}+\phi$ 相当于有四个分量($p_1,p_2,p_3,1$)。第四个分量表示该对象是否包含 ϕ。我们可以通过一个矩阵乘法等式化地表示任意 **v** 和 P(一个行向量乘以一个列向量——见附录 2):

$$\mathbf{v}=(\mathbf{a},\mathbf{b},\mathbf{c},\phi)\begin{bmatrix}v_1\\v_2\\v_3\\0\end{bmatrix} \tag{4.29}$$

$$P=(\mathbf{a},\mathbf{b},\mathbf{c},\phi)\begin{bmatrix}p_1\\p_2\\p_3\\1\end{bmatrix} \tag{4.30}$$

在这里,行矩阵代表了坐标框架的性质,列矩阵代表了感兴趣的特定对象的表示方法。这样,向量和点有了不同的表示方法:第四分量为 0 就是向量,第四分量为 1 就是点。这种表示方法也叫做**齐次表示方法**[①]。使用齐次坐标算得上是计算机图形学的一个标志,它不但可以帮助区分点和向量,还提供了一套仿射变换下简洁的表示方法。它的代价是当用计算机程序表示齐次坐标下感兴趣的点和向量时,需要通过扩展一个 1 或 0[②],得到一个四元组。这一点在我们需要把点和向量进行坐标框架转换时尤为重要。

点和向量的一般表示方法(三维对象是一个三元组,二维对象是一个二元组)和齐次表示方法之间的相互转换非常简单:

从一般坐标到齐次坐标:

- 如果是点,附加一个 1。
- 如果是向量,附加一个 0。

从齐次坐标到一般坐标:

- 如果是向量,那么最后一位分量是 0,把 0 删除。
- 如果是点,那么最后一位分量是 1,把 1 删除。

OpenGL 使用四维齐次坐标表示所有的顶点,如果你输入一个 3 元组坐标(x,y,z),它会立刻被转换成($x,y,z,1$)。如果你输入一个二维坐标(x,y),它首先会被附加一个值为 0 的 z 分量,然后附加一个 1,以得到($x,y,0,1$)。OpenGL 中所有的运算都是在四维齐

---

① 事实上这里的讨论并不彻底。在第 7 章学习投影时,我们会看到使用齐次坐标可以允许一个额外的操作,使得它们体现"齐次"的意义。在讲投影之前,我们没有必要介绍这个操作。

② 在二维中,点是三元组($p_1,p_2,1$),向量是三元组($v_1,v_2,0$)。

次坐标下进行的。

向量的线性组合

注意在齐次坐标下当我们把点和向量按坐标进行组合时一切都是那么和谐：所有的定义和操作都是一致的：

- 两个点$(x,y,z,1)$和$(u,v,w,1)$的差是$(x-u,y-v,z-w,0)$，正如我们所期待的，它是一个向量。
- 一个点$(x,y,z,1)$和一个向量$(d,e,f,0)$的和$(x+d,y+e,z+f,1)$是另一个点。
- 两个向量的和：$(d,e,f,0)+(m,n,r,0)=(d+m,e+n,f+r,0)$，还是一个向量。
- 向量的数乘依然有意义：$3(d,e,f,0)=(3d,3e,3f,0)$。
- 向量的任意线性组合也有意义。两个向量 $\mathbf{v}=(v_1,v_2,v_3,0)$和 $\mathbf{w}=(w_1,w_2,w_3,0)$，由任意的标量 a 和 b，我们有 $a\mathbf{v}+b\mathbf{w}=(av_1+bw_1,av_2+bw_2,av_3+bw_3,0)$，它还是一个合理的向量。

向量的线性组合有很好的解释，那么点的线性组合又如何呢？答案是否定的，除了在下面讲到的一种特殊情况外。

4.5.2　点的仿射组合

用标量 f 和 g 对两个点 $P=(P_1,P_2,P_3,1)$和 $R=(R_1,R_2,R_3,1)$进行线性组合：

$$fP+gR=(fP_1+gR_1,fP_2+gR_2,fP_3+gR_3,f+g)$$

我们知道如果 $f+g=0$ 这是一个合理的向量（为什么？）。但更重要的是只有在 $f+g=1$时它才是一个合理的点！回忆一下等式(4.2)可知，当线性组合的系数和为1时，也叫做仿射组合。我们会发现唯一合理的点的线性组合是仿射组合。例如，$0.3P+0.7R$ 是一个合理的点，$2.7P-1.7R$ 也一样，亦如中点 $0.5P+0.5R$，但是 $P+R$ 不是一个点。对三个点 P，R 和 Q 来说，我们可以得到合理的点 $0.3P+0.9R-0.2Q$，而不是 $P+Q-0.9R$。

事实：点的任意仿射组合是一个合理的点。

但是，如果 $f+g$ 不等于 1，那么任意点的任意线性组合

$$E=fP+gR \tag{4.31}$$

在几何上有什么错误呢？问题会出现在当我们平移坐标系的原点时【Goldman85】。假设原点平移了向量 $\mathbf{u}$，那么 P 就转换为 $P+\mathbf{u}$，R 就转换为 $R+\mathbf{u}$。如果 E 是一个合理的点，它也应该平移到点 $E'=E+\mathbf{u}$，但实际上我们得到：

$$E'=fP+gR+(f+g)\mathbf{u}$$

只有在 $f+g=1$ 的情况下它才是 $E+\mathbf{u}$。

两个点的简单相加 P_1+P_2 并不是一个正确的点，如图 4.19 所示。点 P_1 和 P_2 出现在两个坐标系中，其中一个坐标系是由另一个平移得到。如果把每个点看成是从原点到它的一个向量，那么 P_1+P_2 在两个坐标系下指向不同的两个点。也就是说 P_1+P_2 取决于坐标系的选择。注意，作为对比，仿射组合 $0.5(P_1+P_2)$与坐标系的选择无关。

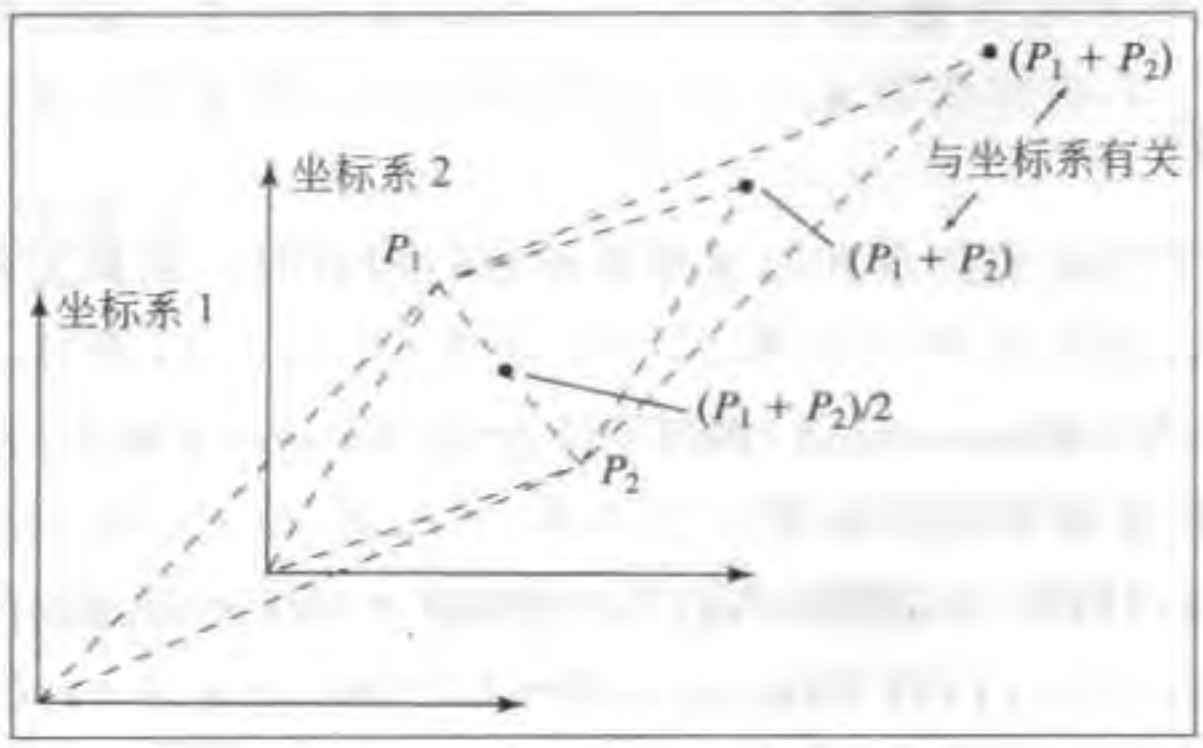


图 4.19 点的相加是不合理的

点与向量的和是点的仿射组合

还有一个有趣的方法可以验证点的仿射和,它还引出了一个图形学中的常用工具。这个方法不需要使用齐次坐标。

假设一个点是由点 A 偏移一个向量得到的,这个向量是 $\mathbf{v}$ 数乘标量 t 后的缩放结果。这个点表示为 $A+t\mathbf{v}$。这是一个点与一个向量的和,所以它是一个合理的点。如果我们认为向量 $\mathbf{v}$ 是另一个点 B 与点 A 的差——$\mathbf{v}=B-A$——那么点 P 可以表示为:

$$P = A + t(B - A) \tag{4.32}$$

它也是一个合理的点。但是经过简单的代数变换就可以得到:

$$P = tB + (1 - t)A \tag{4.33}$$

也就是点的仿射组合(为什么?)。这更加证明了点的仿射和的合理性。事实上,任何点的仿射和都可以写成一个点加上一个向量。如果你对于把点的仿射和写成等式(4.33)的形式很不适应(我们经常要用到这种形式),可以把它简单的记成是等式(4.32)的变形。

例 4.5.1 三角形的质心

如图 4.20 所示,三角形 T 的顶点为 D、E、F。我么用以上的方法证明 T 的三条中线相交于一点,并且这个点在每条中线的 2/3 处,该点叫做 T 的质心(重力的中心[①])。

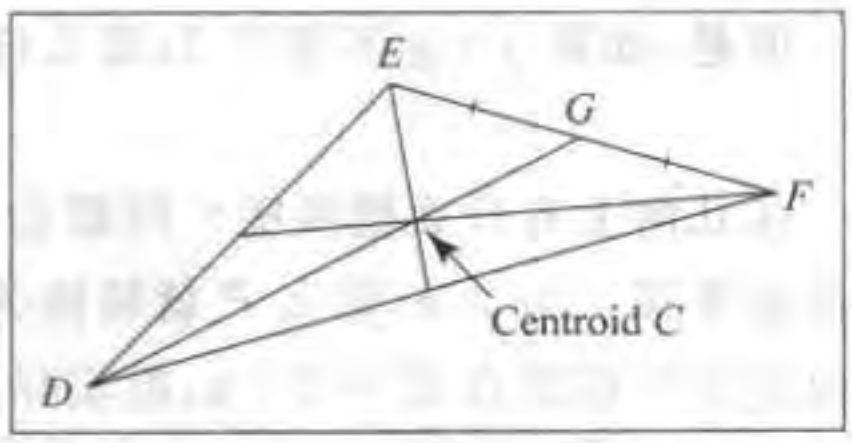


图 4.20 通过仿射组合求三角形的质心

由中线的定义知从 D 点出发的中线是从 D 到它对应边的中点的连线。所以 $G=tE+(1-t)F$,其中 $t=1/2$。假设有先知告诉我们质心在 D 到 G 的 2/3 处,那么我们把质心 C 的参数表达式写成:

$$C = \frac{1}{3}D + \frac{2}{3}G \tag{4.34}$$

把前面 G 的表达式带入,就得到 C 的最终结果:

① 之所以叫重心是因为如果把一个薄板剪成三角形 T 的样子,用一根线连接在它的质心把它吊起来,它会静止在水平位置。重力在质心各方向施加的力士相等的,所以薄板能保持水平。

$$C = \frac{1}{3}D + \frac{2}{3}\left(\frac{E+F}{2}\right) \tag{4.35}$$

试试看！先知是怎么知道质心恰好处于 2/3 的位置呢？巧妙之处在于【Pedoe70】，由于等式(4.35)的结果对于 D、E、F 是对称的，那么该点也必然处于从 E、F 出发的中线的 2/3 处，于是三条线必相交于该点，即 C 点是质心。可以发现这些等式实际上是点的仿射组合。

这个结果可以很好的推广到正 N 边形：质心位置就是 N 个顶点位置的平均值，也是一个仿射组合。对于任意多边形，等式则复杂得多。

4.5.3　两个点的线性插值

点的仿射组合如等式(4.33)所示：

$$P = A(1-t) + Bt$$

实现了点 A 和点 B 的**线性插值**。也就是说，x 分量 $P_x(t)$ 是 A_x 和 B_x 之间比例 t 处的值，y 分量以及三维中的 z 分量也是如此。这是一个足够重要到可以获得专用名的操作，于是 lerp()(线性插值的意思)变得常用起来。在一维，lerp(a, b, t)得到 a 和 b 之间比例 t 处的值(这里 a 和 b 只是标量，不是点)。图 4.21 给出了 lerp()的简单实现。

```
float lerp(float a, float b, float t)
{
    return a+(b-a) * t;                    // 返回一个浮点数
}
```

图 4.21　lerp()实现线性插值

于是，t 为 0 时，lerp 的结果为 a；t 为 1 时，lerp 的结果为 b。$t=1/2$ 时，lerp 得到 a 和 b 的中点。

同样，有时需要得到点 A 和点 B 之间直线段上比例 t 处的点 $P(t)$，这个点一般也叫做点 A 和点 B 在比例 t 处的内插(tween)。结果点的每个分量是通过对 A、B 点对应的分量做 lerp()得到的，所以 Tween()函数很容易实现。

例 4.5.2

令 $A=(4,9)$，$B=(3,7)$，那么 Tween(A, B, t)返回$(4-t, 9-2t)$，例如 Tween(A, B, 0.4)返回(3.6,8.1)(用图形计算机验证结果)。

4.5.4　使用内插的艺术和动画

第 3 章曾讲过我们可以生成从一张图画转换为另一张图画的有趣动画。最简单的情况是两张图画的内容都只是顶点数目相同的多边形(或者一系列多边形)。假设第一张图画 A 是顶点为 A_i 的多边形，第二张图画 B 是顶点为 B_i 的多边形，其中 $i=0,\cdots,n-1$。我们可以生成多边形 $P(t)$，叫做 t 处的内插，它由如下顶点组成：

$$P_i(t) = (1-t)A_i + tB_i$$

如果我们从 0 到 1 连续地选取 t,也就是说,令 $t=0,0.1,0.2,\cdots,0.9,1.0$,可以看到生成的多边形序列以 A 的形状开始,以 B 的形状结束,而在中间是两个形状的融合。t 值较小时,这个形状更像 A,但当 t 变大时,它平滑地过渡到了 B。例如,对 $t=0.25$,点 $P_i(0.25)$ 是从 A 到 B 变化 25%的情况。

图 4.22 是一个简单的例子,多边形 A 是一个房子的形状,多边形 B 是一个 T 字型。房子的 R 点对应"T"的 S 点。R 和 S 的内插点都在 R 到 S 的直线段上,$t=1/2$ 处的内插在线段 RS 的中点。图中显示了 $t=0,0.25,0.5,0.75,1.0$ 处的插值多边形。

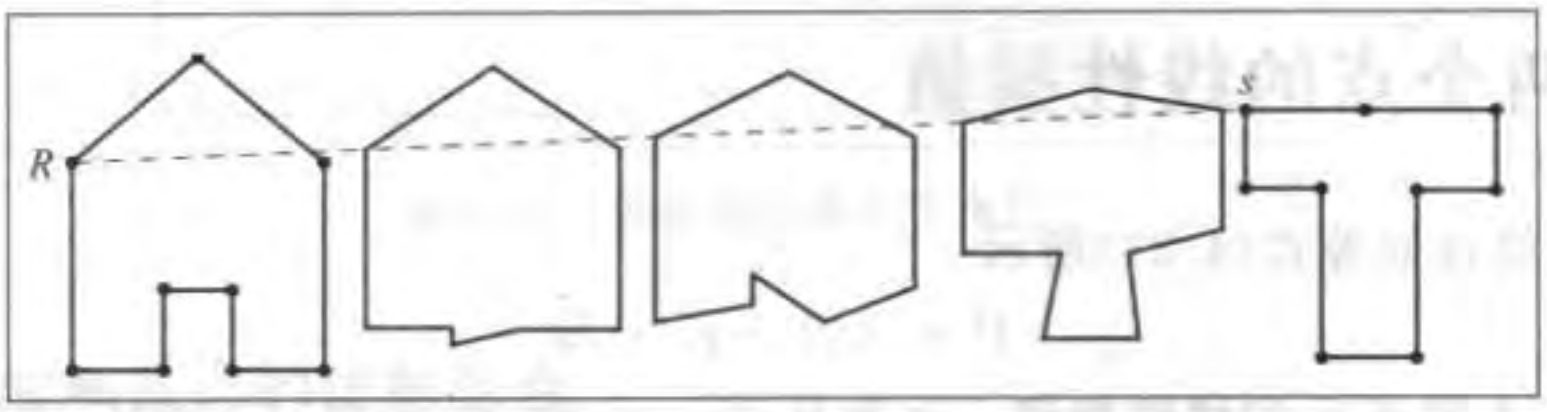


图 4.22 把房子渐变成 T 字型

图 4.23 给出了 drawTween()函数,它可以画出在给定的 t 处,n 边形 A 和 B 的内插形状。

```
void canvas:: drawTween(Point2 A[], Point2 B[], int n, float t)
{   // 画出多边形 A 和 B 在 t 时刻的内插结果
    for(int i=0; i<n; i++)
    {
        Point2 P;
        P=Tween(A[i], B[i],t);
        if(i==0)  moveTo(P.x, P.y);
        else      lineTo(P.x, P.y);
    }
}
```

图 4.23 两个多边形的插值(回忆第 2 章 lineto()和 moveTo()的定义)

drawTween()可以用在 A 和 B 来回插值变化的动画循环中,只要让 t 从 0 变为 1,再从 1 变回 0,如此循环即可。在第 3 章中讲到的双缓冲,可以将一个显示的插值结果瞬时地转换成下一个。代码如下:

```
for(t=0, delT=0.1; ;t+=delT)
//来回循环的插值动画
{
        <clear the buffer>
        drawTween(A, B, n, t);
        glutSwapBuffers();
        if(t>=1.0 || t<=0.0) delT=-delT;
        //让 t 向相反方向变化
}
```

图 4.24 是这种多边形内插在艺术中的一个应用，图中给出了三个插值的中间结果。由于两组初始多边形距离足够远，使得它们之间有足够的空间画出三个中间结果，而不会发生相互覆盖，让五个图形很好的绘制在一帧上。

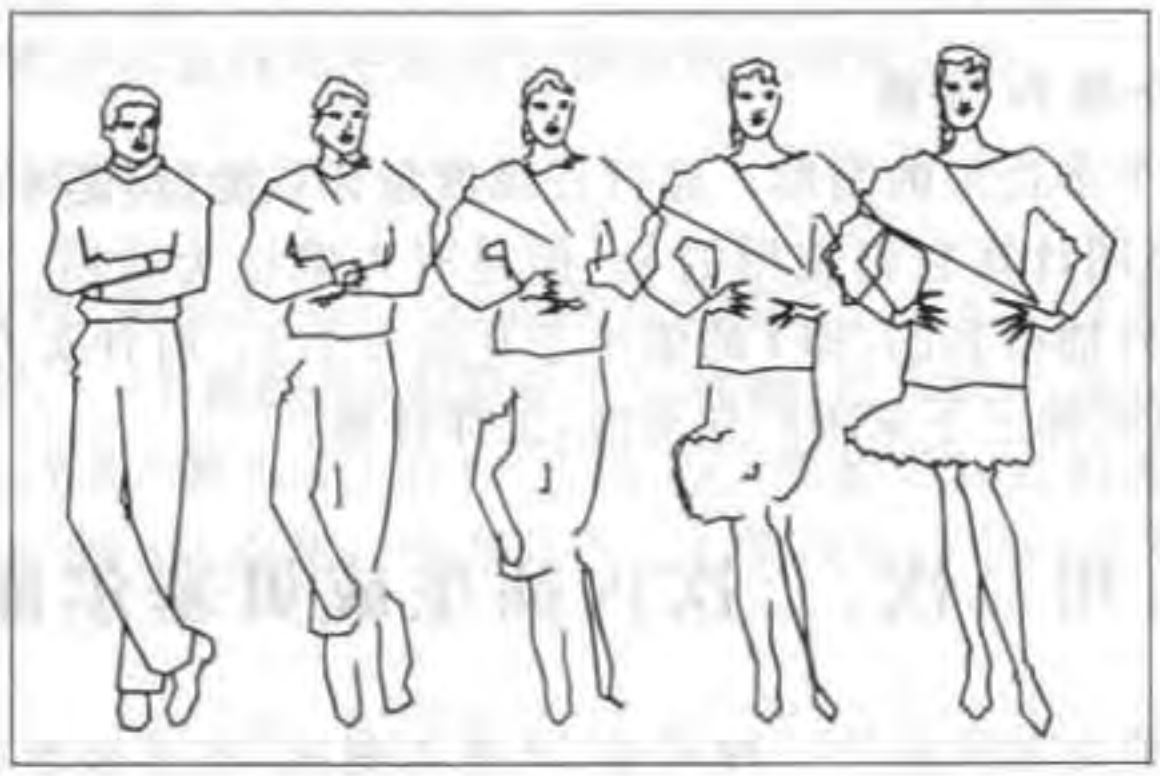


图 4.24　从男人变为女人

Susan E. Brennan 使用这个方法制作了一些名人画像的讽刺漫画。图 4.25 给出了一个例子。第二和第五张脸（分别是 $t=0$ 和 $t=1$ 处的图形）是基于数字化的 Elizabeth Taylor 和 John F. Kennedy 的肖像。第三、四张脸是它们的内插，而第一张和最后一张则是基于"外插"——当 t 值大于 1 时，$(1-t)$ 将变为负值。外插可以产生讽刺漫画的效果，从某种意义上说，是从多边形 A 变到多边形 B 后再沿着变化趋势变化下去的结果。小于 0 的 t 值也会用到，这将产生类似的效果。

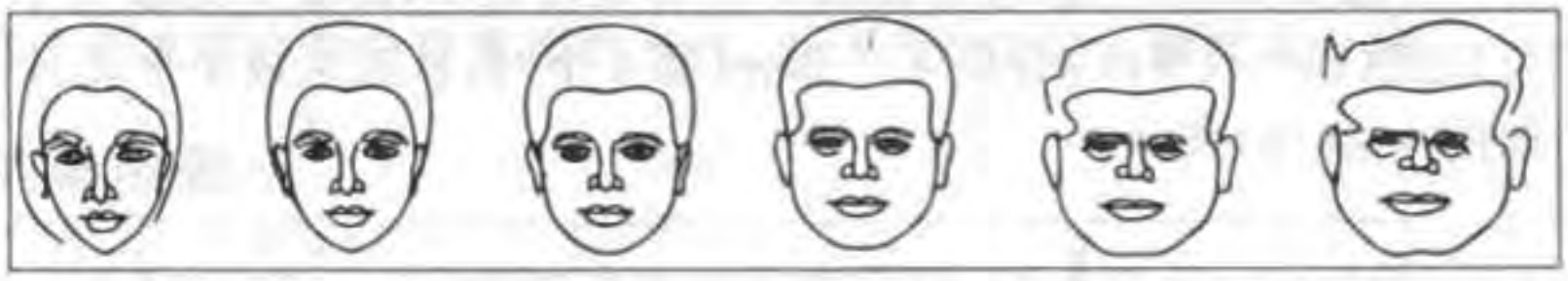


图 4.25　脸部讽刺漫画：内插和外插

在电影行业中使用内插动画可以减少动画制作成本，例如制作卡通片。在早期，艺术家制作一秒钟的电影必须画 24 幅画，因为电影每秒播放 24 帧。有了计算机的帮助后，艺术家只需要画出某个片断的第一帧和最后一帧，也叫**关键帧**，剩下的帧由计算机自动生成。例如，当卡通片中的角色在某半秒的动画片断中移动不是很快，艺术家就可以只画出这个片断的第一和最后一帧，并将它们数字化，计算机可以用线性插值生成其余 10 帧。这样大大节省了艺术家的时间。见本章末的案例，有一个程序设计的项目，可以产生这样的效果。

练习

练习 4.5.1　内插的极限情况

如果多边形 A 的所有点 A_i 都是一样的，那么内插的结果怎样？多边形 B 是怎样一帧帧出现的？

练习 4.5.2 外插

多边形 A 是一个顶点为(1,1),(−1,1),(−1,−1),(1,−1)的正方形,多边形 B 是一个顶点为(4,3),(5,−2),(4,0),(3,−2)的楔形。在图形计算器或纸上画出 $t=-1$,−0.5,0.5 和 1.5 时的形状 $P(t)$。

练习 4.5.3 外插和内插

假设并排演示五个多边形的图形。经过仔细度量你认为(但是不一定正确)中间的三个图形内插得到的,并试图计算它们使用的 t。但是有人确认为最后一个图形实际上是第一个图形和第四个图形外插得到的,即 t 的值小于 0 或大于 1。有什么方法验证这个吗?如果是外插,它使用的 t 值能确定下来吗?如果能,怎样计算?

4.5.5 预览:用二次、三次内插生成贝塞尔曲线

在第 8 章我们会介绍如何设计一种叫做"贝塞尔曲线"的复杂形状。有趣的是,它的基本思想只是一组点的简单内插。在前面的线性插值中我们给出了单位划分(就是把 1 分成和为 1 的几部分)。如果把 1 按如下方法划分为 3 部分,就可以推广到二次线性插值。首先把 1 写成

$$1=((1-t)+t)^2$$

从而把 1 分成 3 部分 $(1-t)^2$、$2(1-t)t$ 和 t^2。显然它们的和为 1,所以它们可以用来生成三个点 A、B 和 C 的仿射组合:

$$P(t)=(1-t)^2A+2t(1-t)B+t^2C \tag{4.36}$$

这就是点 A、B 和 C 的贝塞尔曲线。图 4.26(a)给出了 t 从 0 变到 1 时轨迹 $P(t)$的形状。它从 A 点平滑地运动到 C 点(注意曲线并没有经过点 B,而仅仅是接近了它一下)。进一步地,我们可以把 $((1-t)+t)^3$ 分成 4 个部分(哪 4 个?),从而生成 4 个点 A、B、C、D 的三次插值曲线,如图 4.26(b)所示。

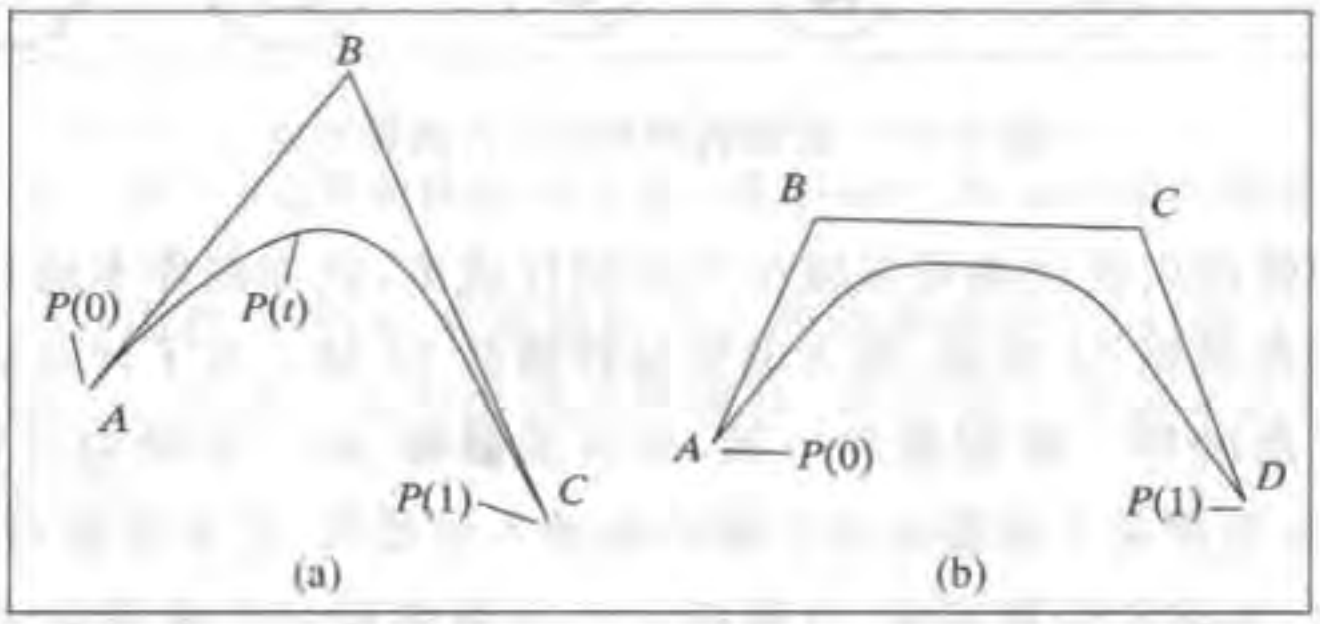


图 4.26 贝塞尔曲线是内插的结果

(a) 二次;(b) 三次

练习

练习 4.5.4 试一试

用图形计算机画出三个点 A、B 和 C。对 $t=0,0.1,\cdots,0.9,1$,用等式(4.36)计算出 $P(t)$的位置,并用直线段把它们顺序连起来。得到的形状一定是一条抛物线吗?

4.5.6　表示直线和平面

我们现在开始介绍一些基本形式，用来从数学上表示直线和平面。在图形学的程序设计里，用这些基本形式表示直线和平面的数据结构是很常见的。

二维和三维空间的直线

两点确定一条直线，例如图 4.27(a)的点 B 和点 C。它的长度是无限的，经过两个点并向两个方向无限延伸。一个**直线段**(为简单也叫**线段**)也由两个点定义，又叫**端点**。但是只从一个点延伸到另一个点(图 4.27(b))。它的"父"线是经过它的两个端点的无限长直线。射线是"半无限"的，它由一个点和一个方向确定。射线从一个点出发，沿给定方向延伸到无穷远(图 4.27(c))。

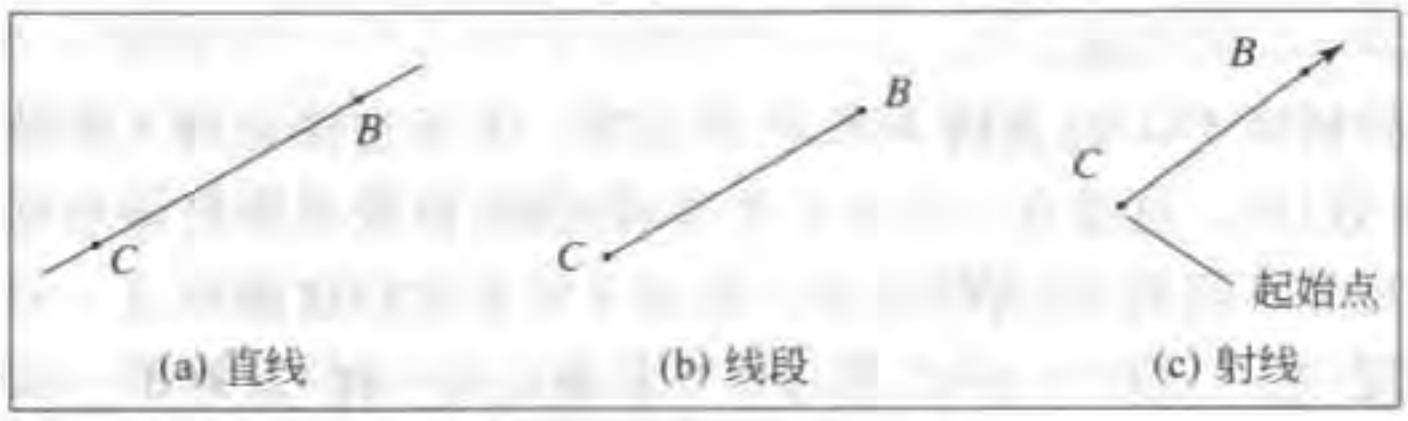


图 4.27　直线、线段和射线

线段和射线这些对象我们是非常熟悉的，给出它们的重要属性和表示方法也是非常有用的。射线将在第 12 章学习二维和三维**光线跟踪**时非常重要。读者会非常愿意看到，在第 12 章，使用光线跟踪将渲染出令人惊叹的真实画面，这都归功于计算机图形学。在本节，我们将介绍计算机图形学中最重要的直线、射线表示方法：**参数表示法**。

直线的参数表示

等式(4.32)和等式(4.33)的构造方法非常有用，因为随着 t 的变化，点 P 的轨迹就是由 C 点和 B 点确定的直线。于是这个构造方法可以让我们命名和计算该直线上的任意一点，即用**参数** t 来区分直线上的每一个点。设直线为 L，令 $P(t)$ 表示参数 t 处的点的位置，令 $\mathbf{b}=B-C$，我们有：

$$L(t) = C + \mathbf{b}t \tag{4.37}$$

随着 t 的变化，$L(t)$ 的位置也在直线上变化。(我们前面曾见过，如果把 t 看成时间也会很有用。)图 4.28是向量 $\mathbf{b}$ 和直线 L 经过 C 和 B 点。(这里给出的是二维示例，三维的情况是类似的。)注意看当 t 取不同值时，$L(t)$ 的位置如何变化。当 $t=0$ 时，$L(0)$ 计算的结果是 C，所以在 $t=0$ 时，$L(t)$ 的位置在 C 点。同理当 $t=1$ 时，$L(1)=C+(B-C)=B$。当 t 变化时，我们在 C 点加上一个伸长或缩短的向量 $\mathbf{b}$，就得到直线上一个新点。如果 t 比 1 大，这个点落在 CB 的延长线上，如果 t 小于 0，这个点落在 BC 的延长线上。

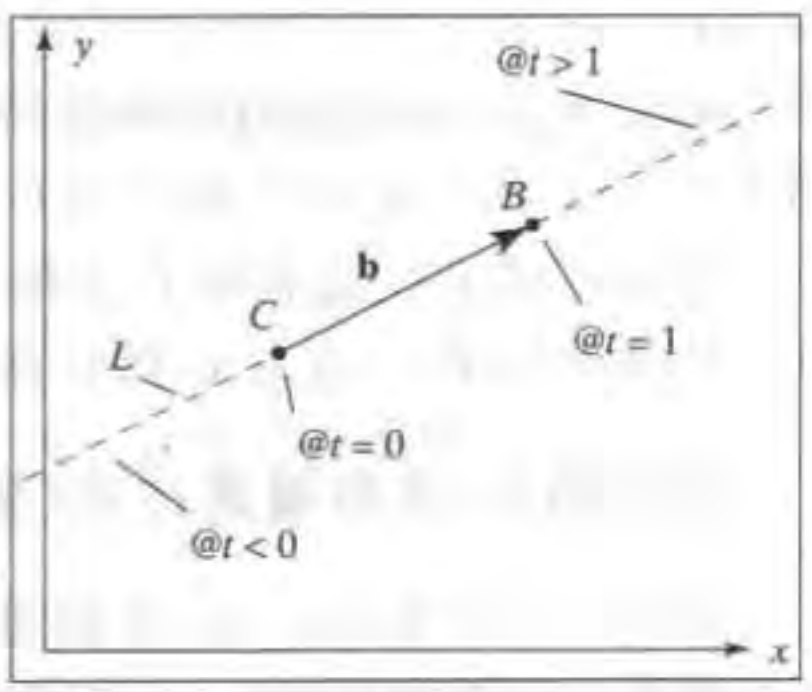


图 4.28　直线的参数表示 $L(t)$

对于一个固定的 t 值，如 $t=0.6$，等式(4.37)能给出经过 C 和 B 的直线上的唯一点——$L(0.6)$，所以这个等式描述了一个点。但是它也可以看成是随 t 变化时产生直线上每个点的坐标的函数，所以它也叫做**直线 L 的参数表示**。

图 4.27 的直线、射线和线段都可以用等式(4.37)中相同的 $L(t)$ 表示，只是参数 t 的取值范围有所不同：

$$\begin{aligned} &\text{线段：} \quad 0 \leqslant t \leqslant 1 \\ &\text{射线：} \quad 0 \leqslant t < \infty \\ &\text{直线：} \quad -\infty < t < \infty \end{aligned} \tag{4.38}$$

C 点也叫做射线的**起始点**。

一个非常有用的事实是当 t 在 0 和 1 之间时，$L(t)$ 处于 C 和 B 之间比例 t 处。例如，当 $t=1/2$ 时，点 $L(0.5)$ 处于 C 和 B 的中点。当 $t=0.3$ 时，点 $L(0.3)$ 处于 C 和 B 之间 30%的位置上。这可以由等式(4.37)推出：因为 $|L(t)-C|=|\mathbf{b}||t|$ 且 $|B-C|=|\mathbf{b}|$，所以 $|t|$ 的值是 $|L(t)-C|$ 与 $|B-C|$ 的比值。

我们也可以分析点 $L(t)$ 沿直线 L 移动的速度。由于它在时间 t 内经过了 $|\mathbf{b}||t|$ 的距离，它的速度是常数 $|\mathbf{b}|$。这里有一点非常重要的区别，曲线的参数表示形式($p(t)$)和这个曲线上的运动路径是不同的，因为仅由 $p(t)$ 随着 t 的变化扫过曲线这一点并不能给出该点沿路径运动的速度(这一点对于 $p(t^2)$ 和 $p(t^3)$ 来说也是一样)。对于一条给定的运动路径曲线，有无穷多种参数表示方法。回忆一下直线的斜截式：$y=m*(x-a)$，它使用直线的斜率和 $y=0$ 处 x 的值建立直线方程。对一个垂直的直线，它的斜率是无穷大，此时斜截式失效。这也是斜截式在计算机图形学中不常用的一个原因；可以正确表示垂直直线的参数表达式无疑是更优的。

例 4.5.3 二维空间中的直线

求过 $C=(3,5)$ 和 $B=(2,7)$ 两点的直线的参数表示。

解：建立向量 $\mathbf{b}=B-C=(-1,2)$ 可得到直线的参数表示 $L(t)=(3-t,5+2t)$。

例 4.5.4 三维空间中的直线

求过 $C=(3,5,6)$ 和 $B=(2,7,3)$ 两点的直线的参数表示。

解：建立向量 $\mathbf{b}=B-C=(-1,2,-3)$ 可得到直线的参数表示 $L(t)=(3-t,5+2t,6-3t)$。

还有其他参数化直线的方法，不过它们都不常用。例如：

$$W(t)=C+\mathbf{b}t^3$$

点 $W(t)$ 同样扫过直线 L 上每一个点。当 $t=0$ 时它在 C 点，当 $t=1$ 时它在 B 点。与 $L(t)$ 不同的是，$W(t)$ 在从 C 到 B 的过程中速度是变化的。

直线的点-法向量表达式(隐式形式)

这种形式与等式(4.38)是相同，也是直线的参数化表达式，但是它能更好的揭示潜在的几何关系。二维直线的一个常见形式如下：

$$fx+gy=1 \tag{4.39}$$

其中 f 和 g 是常数。这表示所有符合该等式的点(x,y)都在这条直线上，所以它实际上相当于给出了属于一条直线的点的条件。**注意**，这只在二维有效，三维空间中的直线需要两个等式。

这个等式可以被写成一个点积：$(f,g)\cdot(x,y)=1$，所以，对直线上每个点，用某个固定的向量与之做点积必然得到相同的结果。我们考察向量(f,g)的几何解释，便可以得到直线的点-法向量表达式。一个点-法向量表达式由直线上的一个任意一个点 P 和一个垂直于直线的向量 V 组成。因此，一条直线可以由(P,V)确定，不同的直线有不同的(P,V)。点-法向量表达式在裁剪和光线跟踪中很有用。严格来说，点-法向量表达式没有维度的概念，但是，二维的直线有点-法向量表达式而三维的直线没有。

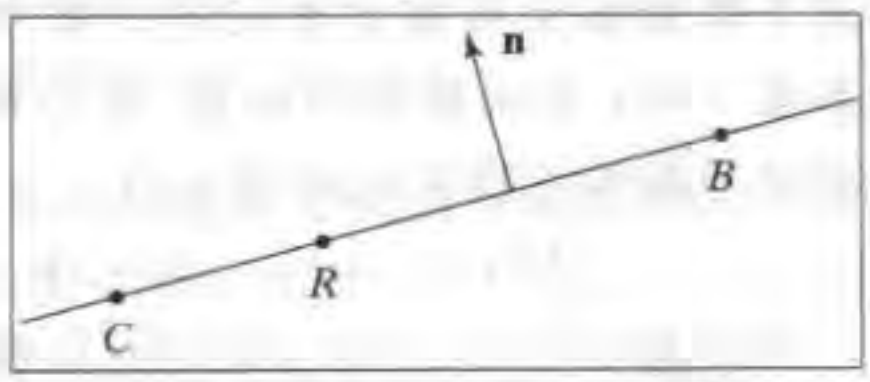


图 4.29　求二维直线的点-法向量表达式

在二维空间中，假设我们知道直线 L 经过点 C 和点 B，如图 4.29 所示。它的点-法向量表达式是什么呢？如果我们能找到一个正交于直线的向量 $\mathbf{n}$，那么对每个直线上的点$R=(x,y)$，向量 $R-C$ 一定与 $\mathbf{n}$ 正交，所以我们得到关于 R 的等式：

$$\mathbf{n}\cdot(R-C)=0 \quad \text{（点-法向量表达式）} \tag{4.40}$$

这就是直线的点-法向量表达式，它说明对于直线上每一个点 R，上式的点积一定得到 0。两条直线可以由直线上的点和直线的法向量区分。

我们依然需要找到 $\mathbf{n}$。用 $\mathbf{b}$ 表示 C 到 B 的向量，则 $\mathbf{b}^{\perp}$ 就是我们要找的 $\mathbf{n}$。为得到直线的点-法向量表达式，$\mathbf{b}^{\perp}$ 的任何数乘结果都可作为法向量 $\mathbf{n}$。

例 4.5.5　点-法向量表达式

设直线 L 经过点 $C=(3,4)$和点 $B=(5,-2)$，那么 $\mathbf{b}=B-C=(2,-6)$，$\mathbf{b}^{\perp}=(6,2)$（把它们画出来）。选择 C 点作为点-法向量表达式中的点，我们得到点-法向量表达式：$(6,2)\cdot((x,y)-(3,4))=0$，或 $6x+2y=26$。如果需要，等式两边可以同除以 26（或者任何非 0 的数）。

若给定了直线的表达式 $fx+gy=1$，可以很简单的得到直线的法向量。只需再把它写成$(f,g)\cdot(x,y)=1$，很显然法向量 $\mathbf{n}$ 就是(f,g)（或是它的任意数乘结果）。例如，直线 $5x-2y=7$的法向量是$(5,-2)$，或者更一般地，是$(5K,-2K)$，其中 $K\neq 0$。

如果给定了直线的点-法向量表达式，我们也可以直接得到它的参数表达式。假设直线 L 的点-法向量表达式为 $\mathbf{n}\cdot(P-C)=0$，其中 $\mathbf{n}$ 和 C 是已知的，那么它的参数表达式就是 $L(t)=C+\mathbf{n}^{\perp}t$（为什么？）。如果二维直线方程已知，可以用以下步骤得到直线的参数表示：

第 1 步：用前面提到的方法找到法向量 $\mathbf{n}$。

第 2 步：计算直线上一点(C_x,C_y)，可以将任意的 C_x 代入直线方程求得 C_y。

三维空间中的平面

三维空间中使用大量多边形，所以多边形所在的平面必须有一种简洁的表示形式。一个多边形（通常是一个对象的面）处于它的**父平面**上，我们经常需要裁剪出平面中的对象，或

找到某个面所在的平面。

平面的参数表示

平面的参数表达式由三个元素组成：它上面的一个点 C 和两个(不平行)的向量 $\mathbf{a}$ 和 $\mathbf{b}$，如图 4.30 所示。如果给定平面上(不共线)的三个点 A、B、C，我们有 $\mathbf{a}=A-C$ 和 $\mathbf{b}=B-C$。

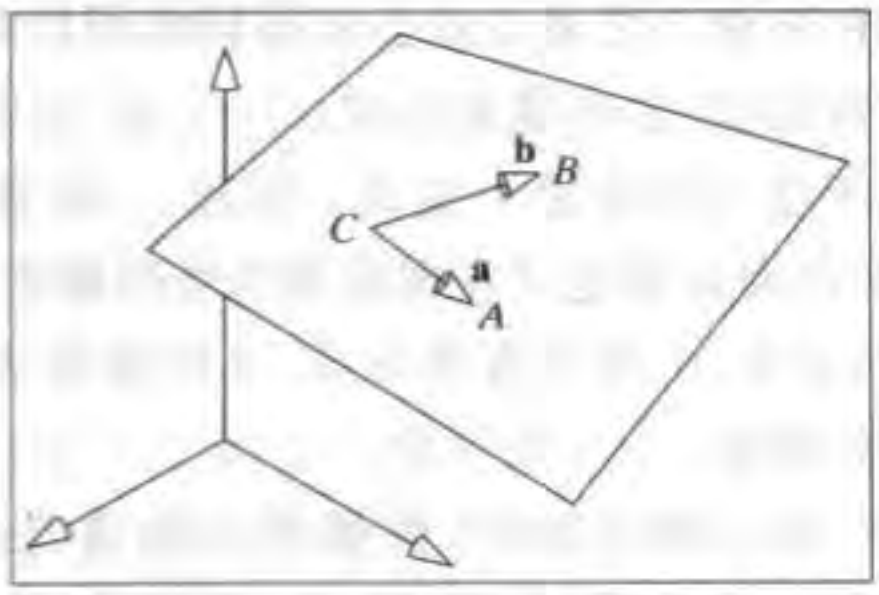


图 4.30 使用参数表示定义平面

为建立平面的参数表达式，我们注意到平面上任意一个点都可以表示成向量和的形式：C 加上 $\mathbf{a}$ 的某个数乘结果再加上 $\mathbf{b}$ 的某个数乘结果。如果用参数 s 和 t 表示数乘的标量，我们有 $C+s\mathbf{a}+t\mathbf{b}$。这就是平面的参数表达式 $P(s,t)$：

$$P(s,t)=C+s\mathbf{a}+t\mathbf{b} \tag{4.41}$$

对任意给定的 s 和 t，我们可以得到平面上对应的点。例如，$s=t=0$ 对应了 C 点本身，$s=1$、$t=-2$ 对应点 $P(1,-2)=C+\mathbf{a}-2\mathbf{b}$。

注意面的参数表达式中有两个参数，而曲线则只需要一个。事实上，如果固定其中一个参数，比如令 $s=3$，那么 $P(3,t)$ 就是只有一个变量的函数，它表示一条直线：$P(3,t)=C+3\mathbf{a}+\mathbf{b}t$。

等式(4.41)同时说明我们可以把 $P(s,t)$ 写成一个点的仿射组合形式，这更进一步说明了单位划分的价值(仔细思考这一点)：

$$P(s,t)=s\mathbf{a}+t\mathbf{b}+(1-s-t)C \tag{4.42}$$

例 4.5.6 求给定三个点的平面的参数表达式

设平面经过点 $A=(3,3,3)$，$B=(5,5,7)$，$C=(1,2,4)$。由等式(4.41)，向量 $\mathbf{a}$ 和 $\mathbf{b}$ 为 $\mathbf{a}=(2,1,-1)$ 和 $\mathbf{b}=(4,3,3)$。再根据等式(4.42)可得：$P(s,t)=(1,2,4)+(2,1,-1)s+(4,3,3)t$。也可由等式(4.42)写成仿射组合形式：$P(s,t)=s(3,3,3)+t(5,5,7)+(1-s-t)(1,2,4)$。

平面面片

正如我们可以通过限制直线参数表达式的参数 t 的取值范围来得到射线和线段一样，我们也可以限制平面的参数 s 和 t。

在等式(4.41)的参数表达式中，s 和 t 的取值范围是 $-\infty$ 到 ∞，这样平面将延伸到无穷远。有时我们想处理平面的一个“部分”，例如它上面的平行四边形。这样的部分叫做**平面面片**，这个名称可以让我们把平面想象成是由无数小面片拼成的。在后续章我们会讨论如何将纹理映射到平面面片上。后面我们还会讲到用非平面的面片拼接曲面。许多实体建模的过程需要将不同形状的面片拼成物体的表面。

一个平面面片是通过限制平面的参数 s 和 t 的取值得到的。例如，人们通常把 s 和 t 限制在 0 和 1 之间。这样的片面随着 $\mathbf{a}$、$\mathbf{b}$ 和 C 的选取有不同的位置和朝向。图 4.31(a)给出了**参数空间**中一个正方形的 s、t 取值范围，图 4.31(b)给出了对象空间中在这一限制下得到的面片。

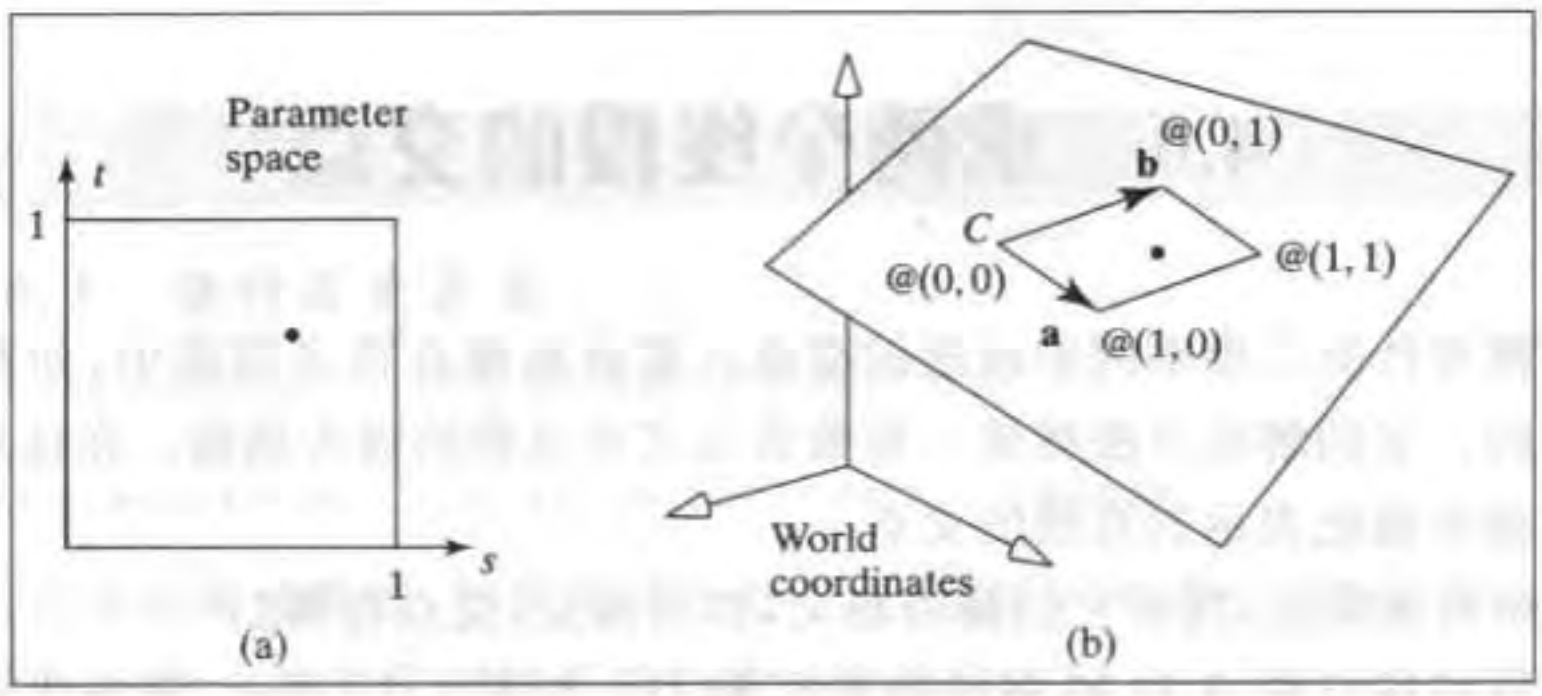


图 4.31 定义平面面片的两个空间

后面我们会看到，把纹理映射到面片的过程需要找到从参数空间到对象空间的一个合适的映射。参数空间中的每个点(s,t)都对应了三维空间中面片上的一点 $P(s,t)=C+\mathbf{a}s+\mathbf{b}t$。这个面片是一个平行四边形，它的顶点对应了参数空间正方形的 4 个顶点，满足：

$$
\begin{aligned}
P(0,0) &= C \\
P(1,0) &= C+\mathbf{a} \\
P(0,1) &= C+\mathbf{b} \\
P(1,1) &= C+\mathbf{a}+\mathbf{b}
\end{aligned}
\tag{4.43}
$$

向量 **a** 和 **b** 决定了面片的大小和朝向。如果 **a** 与 **b** 正交，面片会成为矩形，进一步，如果 **a** 和 **b** 长度相等，面片会成为正方形。改变 C 点只能改变面片的位置而不能改变它的形状和朝向。

例 4.5.7 求面片

令 $C=(1,3,2)$，$\mathbf{a}=(1,1,0)$，$\mathbf{b}=(1,4,2)$，计算平面面片的顶点。

解：由(4.43)式可得：$P(0,0)=(1,3,2)$，$P(0,1)=(2,7,4)$，$P(1,0)=(2,4,2)$，$P(1,1)=(3,8,4)$。

例 4.5.8 建立一个给定属性的面片

找到合适的 **a**、**b** 和 C，使得产生的面片是一个边长为 4 的正方形，且以原点为中心，与 xz 平面平行。

解：面片的顶点在$(2,0,2)$、$(2,0,-2)$、$(-2,0,2)$和$(-2,0,-2)$。任选一点作为 C，如$(2,0,-2)$。则 **a** 和 **b** 的长度都为 4，并平行 x 轴或者 z 轴。可以令 $\mathbf{a}=(-4,0,0)$，$\mathbf{b}=(0,0,4)$。

练习

练习 4.5.5 找一个面片

找到点 C 和向量 **a**、**b**，使由它们生成的面片的四个顶点为$(-4,2,1)$、$(1,7,4)$、$(-2,-2,2)$和$(3,3,5)$。

4.6 求两个线段的交点

我们经常需要计算二维空间中线段的交点。它会出现在很多问题中,如判断一个多边形是否是简单的。它的解决方法将展示参数表达式和点积的强大功能。在这里我们的讨论仅限于找到两条参数化表示的直线的交点。

问题:已知两条线段,判断它们是否相交,如果相交,交点在哪?

设一条线段的端点是 A 和 B,另一条是 C 和 D。如图 4.32 所示,两条线段的位置关系可以有多种情况。它们可能不相交(图 4.32(a)和(b)),可能相交于一点(图 4.32(c)和(d)),或者有一段重合图 4.32(e)。它们可能平行也可能不平行。我们需要一种规范的方法,可以处理所有可能的情况。

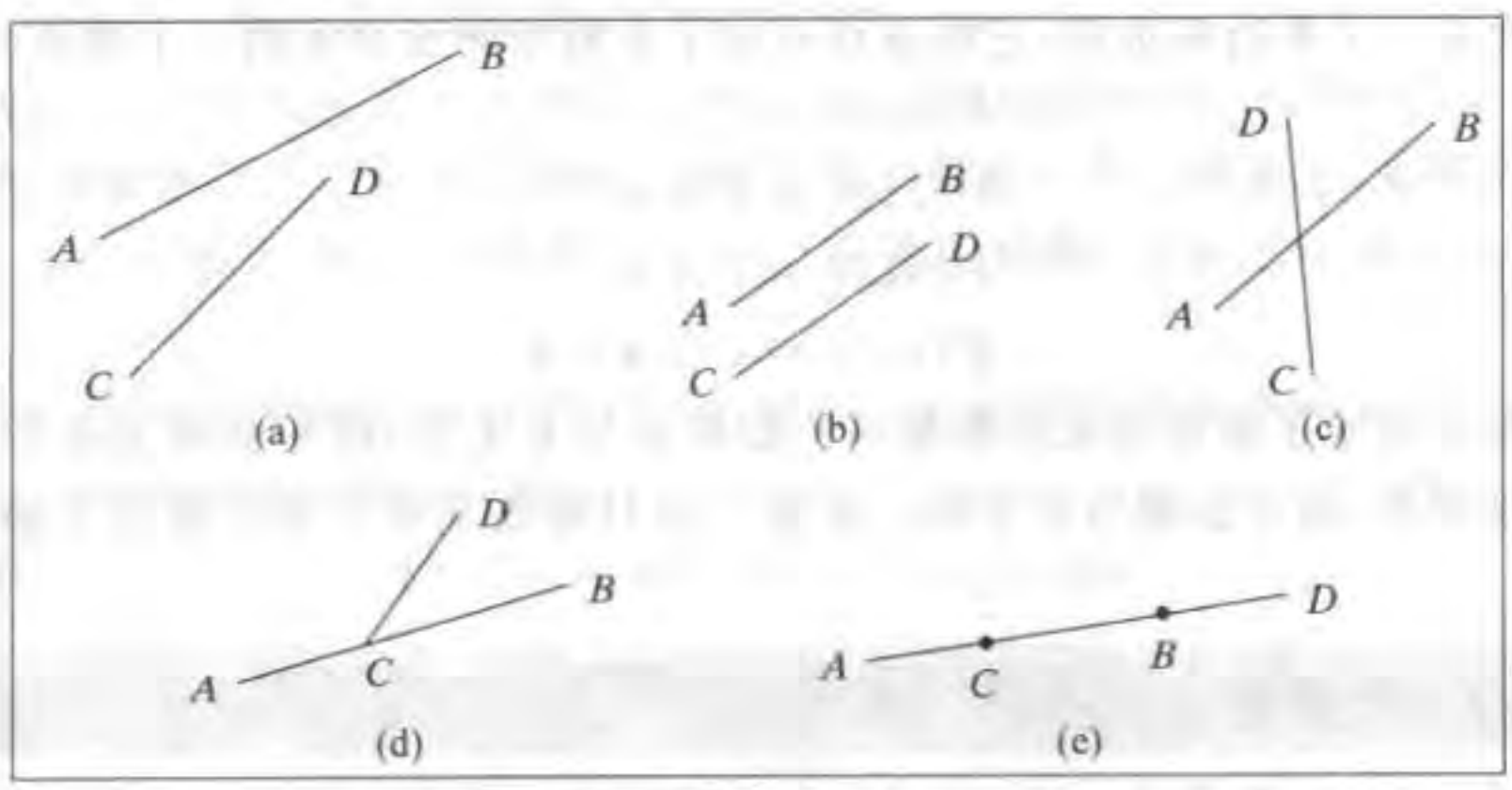


图 4.32 两条直线的多种位置关系

每条线段都有一个**父直线**,即包含它的无限长直线。两条父直线将相交于二维空间中的某个点,除非它们是平行的。我们需要先找到这个点。

我们先求出问题中两条线段的参数表达式。令 A 到 B 的线段为 AB,则

$$AB(t) = A + \mathbf{b}t \tag{4.44}$$

为了简便,我们令 $\mathbf{b}=B-A$。随着 t 从 0 变为 1,有限长线段上的点都会被经过一次。

类似地我们令 C 到 D 的线段为 CD,它的参数表达式为(使用一个不同的参数 u):

$$CD(u) = C + \mathbf{d}u,$$

其中 $\mathbf{d}=D-C$。我们对不同的线段使用了不同的参数,一个为 t,一个为 u,这是为了独立地描述两条直线上不同的点。(如果使用相同的参数,两条直线上的点就关联在一起了。)

如果两条父直线有交点,必然存在一定的 t 和 u 的值,使两个等式相等:

$$A + \mathbf{b}t = C + \mathbf{d}u$$

为了简便,令 $\mathbf{c}=C-A$,我们可以把上式写成关于三个已知向量和两个未知参数的方程:

$$\mathbf{b}t = \mathbf{c} + \mathbf{d}u \tag{4.45}$$

这实际上是含两个未知数的两个线性方程,类似于等式(4.22)。它们可以用通常的方法

求解。

练习

练习 4.6.1　设计求交算法

编写函数 segIntersect()，使它能用到如下的程序段中：

```
if(segIntersect(A, B, C, D, &InterPt))<do something>
```

它用四个点表示两条线段，如果线段没有交点，返回 0，如果有，则返回 1。如果相交，交点的值会存入 InterPt。如果父直线不相交该函数返回－1。

练习 4.6.2　测试多边形的简单性

回顾一下，如果一个多边形除了相邻边在端点处相交外没有边的相交，它就是一个简单多边形。编写一个函数 int isSimple(Polygon P)，它使用穷举算法测试多边形的任意一对边是否相交，如果相交返回 0，否则返回 1(Polygon 是一个描述多边形的类)。这是一个简单但不是很有效的算法。【Moret91】和【Preparata85】提出了更优化算法，它根据 x 和 y 的值对边进行排序，从而提高效率。

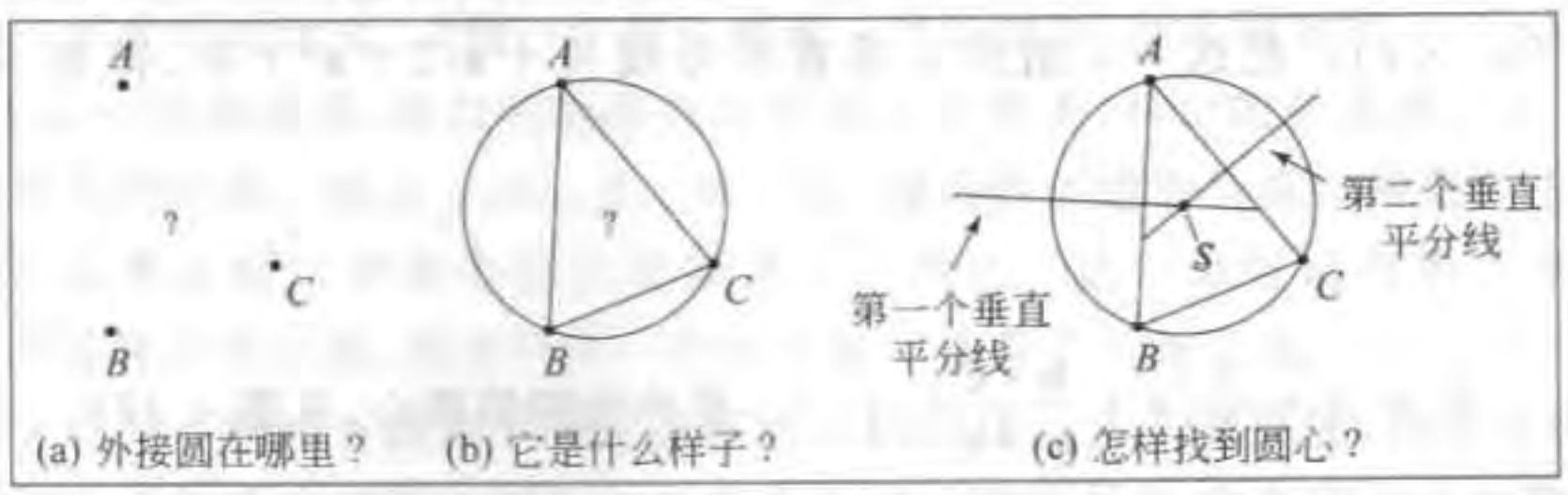
练习 4.6.3　线段求交

对下面的每一对线段，判断它们是否相交，如果相交，求出交点。

1. $A=(1,4)$，$B=(7,1/2)$，$C=(7/2,5/2)$，$D=(7,5)$
2. $A=(1,4)$，$B=(7,1/2)$，$C=(5,0)$，$D=(0,7)$
3. $A=(0,7)$，$B=(7,0)$，$C=(8,-1)$，$D=(10,-3)$

4.6.1　直线求交的应用：过三点的圆

假设一个设计者需要这样一个工具，它可以画出经过给定三点的唯一圆。用户用鼠标在显示设备上给出三个点 A、B 和 C，如图 4.33(a)所示，然后一个过三点的圆会自动生成，如图 4.33(b)所示。过三点的唯一圆叫做以这三点为顶点的三角形的**外切圆**或**外接圆**。它是什么样的呢？为了把它显示出来，需要知道它的圆心和半径，我们需要一段计算这些数据的程序。在解决这个问题的过程中，请注意如果不使用向量和点积，这个问题会变得多么复杂！



(a) 外接圆在哪里？　(b) 它是什么样子？　(c) 怎样找到圆心？

图 4.33　求外接圆

问题

求过给定三点的唯一圆。

方法

图 4.33(c)给出了解法。我们在二维中讨论这个问题,这样一个向量的正交向量就有意义了。所求圆的圆心 S 到三个顶点的距离必须都相等,于是它必须处于三角形 ABC 的每条边的垂直平分线上(一个线段 AB 的垂直平分线 M 是一条与 AB 正交且平分 AB 的直线)。这样如果我们能计算出两条垂直平分线的交点,就能确定圆心 S。在图 4.33(c)中线段 AC 的垂直平分线正交于 AC 且平分它。

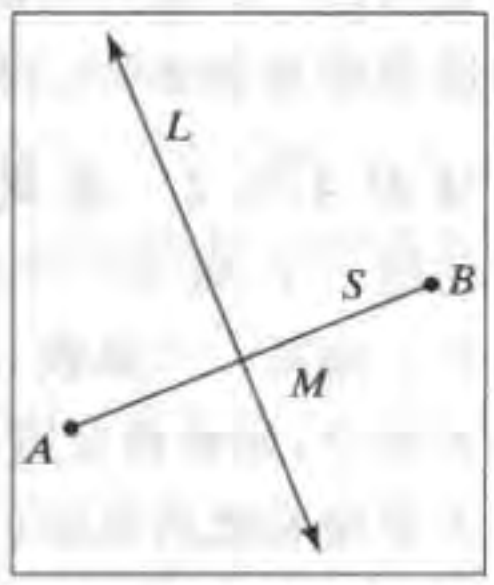


图 4.34 线段的垂直平分线

我们先给出求线段垂直平分线的参数表达式的方法。图 4.34是一个端点为 A 和 B 的线段 S。它的垂直平分线 L 是一条经过 S 的中点 M 且与 S 垂直的无限长直线。不过我们知道中点 $M=(A+B)/2$,并且 S 的法向量方向由 $(B-A)^{\perp}$ 给出,所以垂直平分线有如下参数表达式:

$$L(t)=\frac{1}{2}(A+B)+(B-A)^{\perp}t \quad (AB\text{ 的垂直平分线}) \tag{4.46}$$

现在我们可以计算三个点的外切圆了。我们需要求出 AB 和 AC 的垂直平分线的交点 S。为了简便定义如下向量:

$$\begin{aligned}\mathbf{a}&=B-A\\ \mathbf{b}&=C-B\\ \mathbf{c}&=A-C\end{aligned} \tag{4.47}$$

为找到 AB 的垂直平分线我们需要 AB 的中点和与 AB 垂直的方向。前者就是 $A+\mathbf{a}/2$(为什么),后者为 $\mathbf{a}^{\perp}$。于是垂直平分线的参数表达式为 $A+\mathbf{a}/2+\mathbf{a}^{\perp}t$。在这一过程中不妨注意一下这一点——这个非常值得一做——那就是点的组合总是以**仿射和**的形式出现!类似地,AC 的垂直平分线是 $A-\mathbf{c}/2+\mathbf{c}^{\perp}u$,其中参数为 u。点 S 就在两条垂直平分线的交点上,即求解:

$$\mathbf{a}^{\perp}t=\mathbf{b}/2+\mathbf{c}^{\perp}u$$

(这里我们用到了 $\mathbf{a}+\mathbf{b}+\mathbf{c}=0$)。为消掉 u 我们在等式两边同时点积 $\mathbf{c}$,于是得到 $t=1/2(\mathbf{b}\cdot\mathbf{c})/(\mathbf{a}^{\perp}\cdot\mathbf{c})$。把这一 t 值代入垂直平分线 $A+\mathbf{a}/2+\mathbf{a}^{\perp}t$ 中,得到 S 点的显式形式[①]:

结果

$$S=A+\frac{1}{2}\left(\mathbf{a}+\frac{\mathbf{b}\cdot\mathbf{c}}{\mathbf{a}^{\perp}\cdot\mathbf{c}}\mathbf{a}^{\perp}\right) \quad (S\text{ 是外切圆的圆心,见图 4.33}) \tag{4.48}$$

(注意如果不使用向量,这个计算过程将是多么的复杂。)外切圆的半径是从 S 点到任

① S 的其他闭式表达式在以前的书中出现过,见[Goldman90]和[Lopez92]。

意一个顶点的距离，也就是$|S-A|$。它就是等式(4.48)最后一项的大小。经过一些计算(推导一下)后我们得到：

$$\text{半径} = \frac{|\mathbf{a}|}{2}\sqrt{\left(\frac{\mathbf{b}\cdot\mathbf{c}}{\mathbf{a}^{\perp}\cdot\mathbf{c}}\right)^2+1} \quad \text{(外切圆的半径)} \tag{4.49}$$

结论：顶点为 A、B 和 C 的三角形的外切圆是以等式(4.48)为圆心，等式(4.49)为半径的圆。

当 S 和半径已知后，可以用第 3 章的 drawCircle()函数画出所求圆。

例 4.6.1

求以点 $A=(3,5)$和点 $B=(9,3)$为端点的线段 S 的垂直平分线 L。

解：直接计算可得，中点 $M=(6,4)$，$(B-A)^{\perp}=(2,6)$，所以 L 的表达式为 $L(t)=(6+2t,4+6t)$。画出 S 和 L，看结果是否正确。

每一个三角形还都有一个**内切圆**，在计算机辅助设计中经常需要计算它。本章末的一个案例会介绍如何求内切圆，并介绍有趣的“9 点圆”。

练习

练习 4.6.4　垂直平分线

求以点 $A=(0,6)$和点 $B=(4,0)$为端点的线段的垂直平分线的参数表达式，画出线段和垂直平分线。

练习 4.6.5　求外切圆

令三角形的三个点为 $A=(3,6)$、$B=(4,2)$和 $C=(8,-12)$。计算它的外切圆的圆心和半径。

4.7　直线和平面求交及裁剪

求一条直线与另一条直线或一个平面交点的问题在图形学中出现的非常频繁。在图形学中有很多裁剪算法，为了使问题相对容易一些，我们重点介绍一种裁剪算法：Cyrus-Beck 裁剪算法。其实在 4.6 节中我们已经遇到过一个裁剪算法，就是求两条直线的交点。这里我们先解释一下裁剪的定义。“用一个窗口裁剪一个多边形”就是用如图 4.35(a)中的窗口裁剪图 4.35(a)中的多边形，窗口内的多边形的部分会留下，其余的会去掉。在 4.35(b)中可以看到裁剪后的结果。在点 a、b、c、d、e 和 f 处，被裁剪多边形与窗口相交。注意，这里被裁剪的多边形边界从窗口里出去后又重新进入了两次。这一点可以与第 3 章的 Cohen-Sutherland 裁剪算法作比较，后者只用一个矩形窗口裁剪了一条直线。

我们会介绍另一种用直线或平面裁剪多边形的方法。它非常直观，容易理解。我们将在 4.8 节中把这个算法的结果应用到用凸多边形或凸多面体裁剪直线上。在第 7 章，我们会看到这是观察三维对象的一个基本步骤。在第 12 章我们将使用相同的方法得到光线跟踪的基本概念。

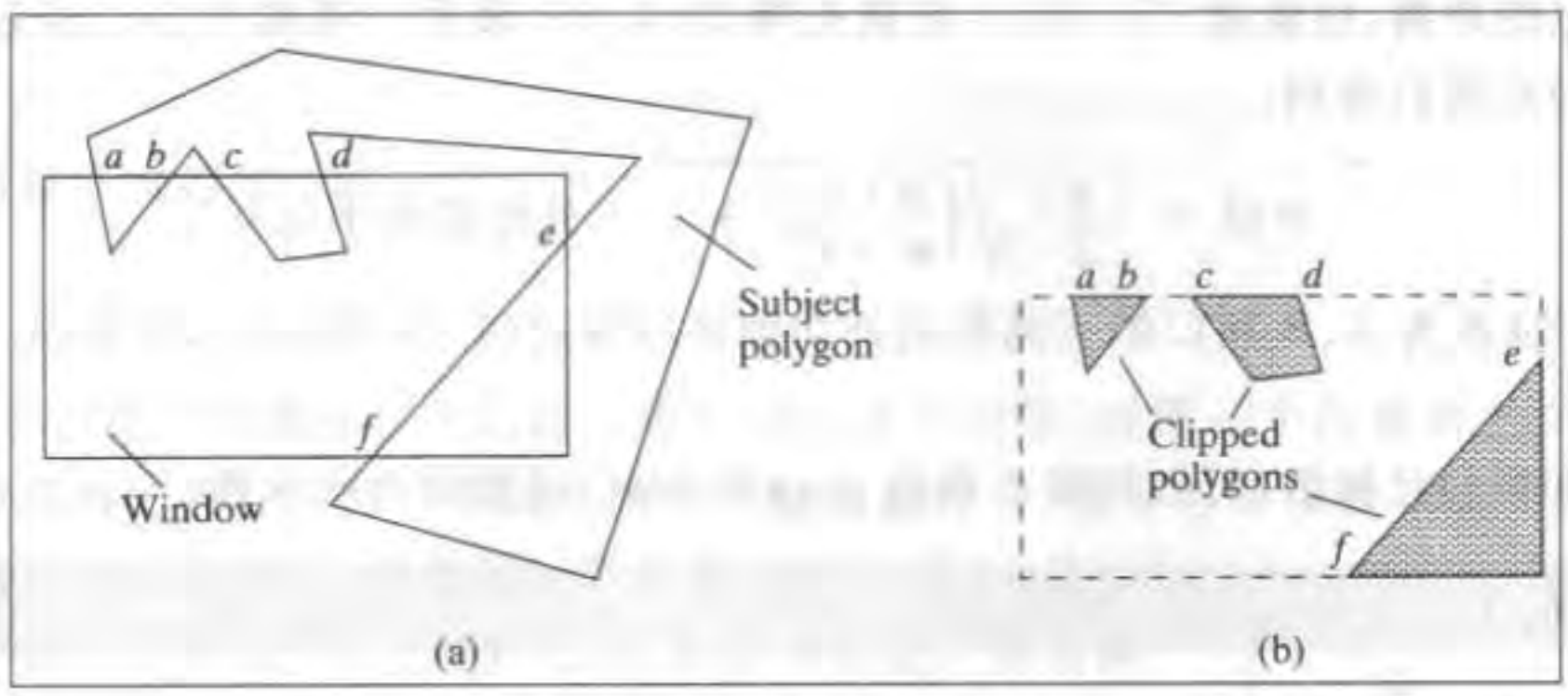


图 4.35 用窗口裁剪多边形

在二维空间中我们想得到两条直线的交点,在三维空间中我们想得到一条直线与一个平面的交点。这两个问题可以同时解决,因为等式的表达是基于点积的,所以不论相关的向量是二维的还是三维的,它们都有相同的表达式。

设一条直线的参数表达式为:$R(t)=A+ct$,而且我们把它看成一条射线。我们想计算射线与图中的物体交于何处。我们把问题集中在求射线与平面(多面体的面)的交点上。进一步讲,在二维,就是与一条直线——凸多边形边的交点;在三维,就是与一个平面——凸多面体的凸多边形面的交点。设点 P 为平面上一点且平面的法向量为 $\mathbf{n}$,如图 4.36 所示。二维情况下就是点 P 在直线上且向量 $\mathbf{n}$ 与直线正交。图 4.36(a)是射线与直线相交,图 4.36(b)是它与平面相交。我们要计算交点。

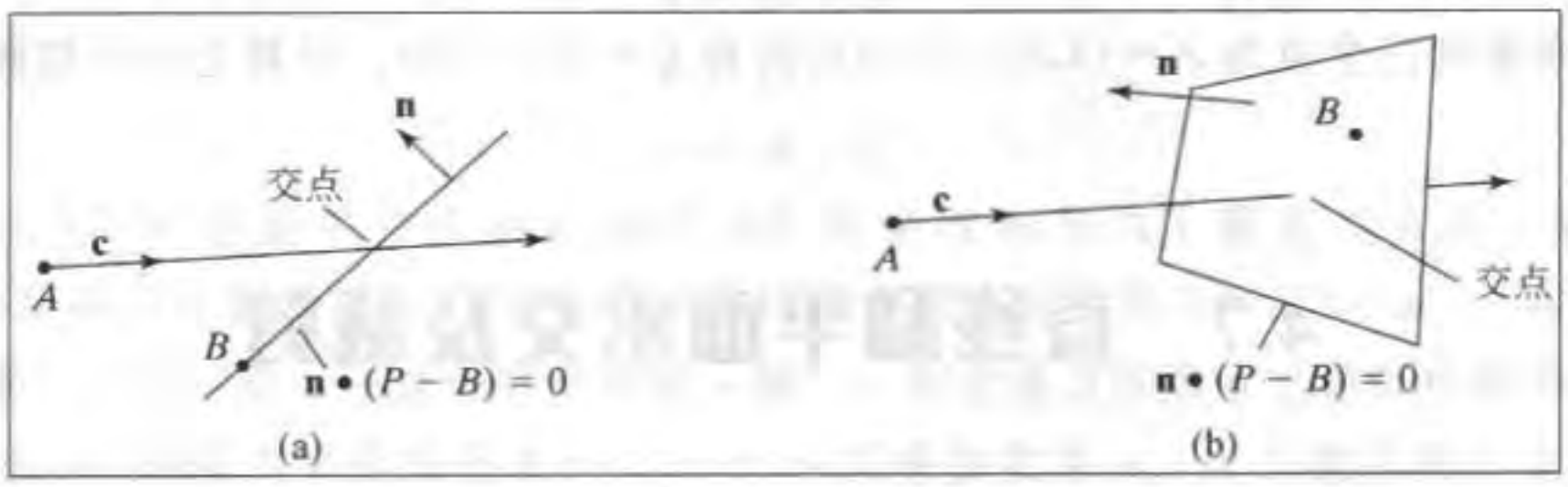


图 4.36 射线与直线或平面的交点在哪里?

假设在 $t=t_{hit}$ 处,射线与物体相交(因为可以把参数 t 看成时间,所以 t_{hit} 可以叫做**击中时间**,交点可以叫做**击中点**)。首先考虑二维情况,问题中的直线有点-法向量表达式。在这一 t 值处,直线和射线点的坐标必然相同,所以 $A+\mathbf{c}t_{hit}$ 必然满足直线的点一法向量表达式。于是我们把这个交点带入直线的点一法向量方程,得到 t_{hit} 的一个条件:

$$\mathbf{n}\cdot(A+\mathbf{c}t_{hit}-B)=0$$

可以把它写成

$$\mathbf{n}\cdot(A-B)+\mathbf{n}\cdot\mathbf{c}t_{hit}=0$$

这是 t_{hit} 的一个线性方程,解得:

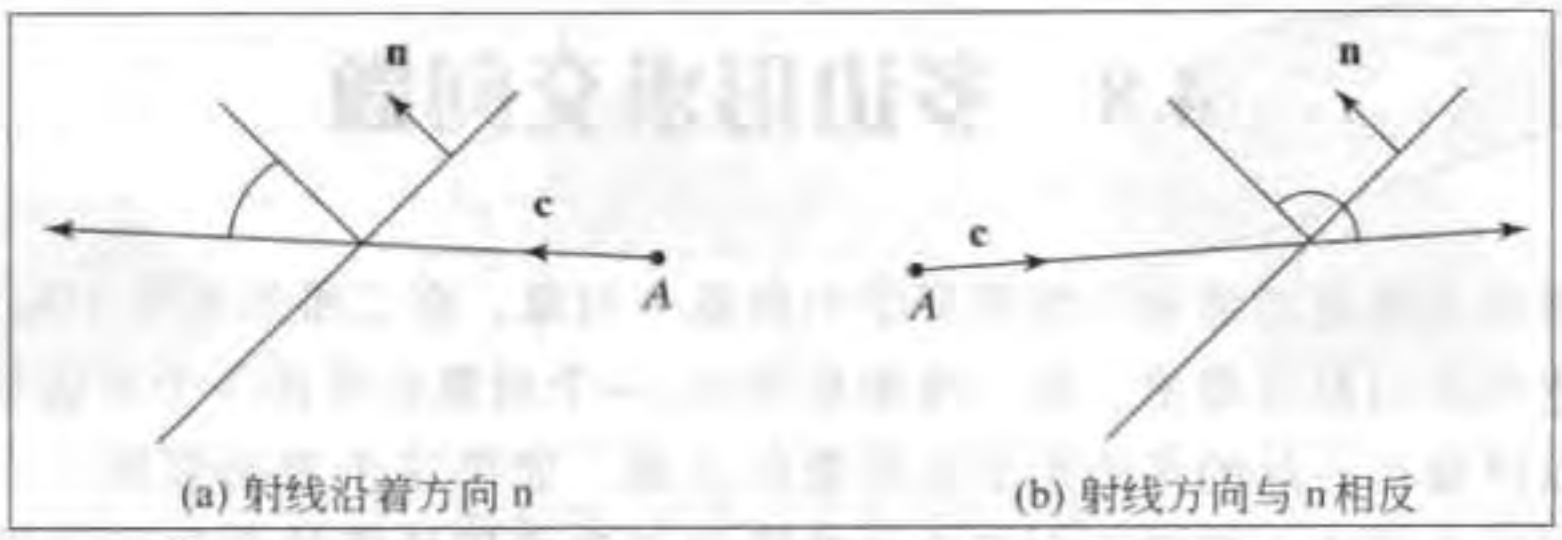
$$t_{hit}=\frac{\mathbf{n}\cdot(B-A)}{\mathbf{n}\cdot\mathbf{c}}\quad(\text{击中时间})\tag{4.50}$$

作为一个比值我们总需要检查 t_{hit} 的分母是否为 0，这种情况发生在 $\mathbf{n}\cdot\mathbf{c}=0$ 时。此时射线的方向与问题中的物体平行，这种情况下射线与物体没有交点或处于物体内[①]。

计算出击中时间后，击中点及交点就很容易得到（注意不要混淆击中时间和击中点）。把 t_{hit} 代入射线的表达式中：

$$P_{hit}=A+\mathbf{c}t_{hit}\quad（击中点）\tag{4.51}$$

在以后的问题中我们还需要知道射线击中直线或平面的大致方向——是沿着法向量 $\mathbf{n}$ 还是与 $\mathbf{n}$ 相反的方向。这将会很重要，因为我们需要知道射线是进入物体还是从物体中射出。图 4.37 给出了射线射向直线的两种情况。在图 4.37(a)中射线方向 $\mathbf{c}$ 和法向量 $\mathbf{n}$ 的夹角小于 90°，我们说射线是沿着 $\mathbf{n}$ 的方向的；在图 4.37(b)中射线方向 $\mathbf{c}$ 和法向量 $\mathbf{n}$ 的夹角大于 90°，我们说射线与 $\mathbf{n}$ 的方向相反。



(a) 射线沿着方向 n　　(b) 射线方向与 n 相反

图 4.37　射线的方向

要判断是以上哪种情况很容易，因为 $\mathbf{n}\cdot\mathbf{c}$ 的符号直接说明了 $\mathbf{n}$ 与 $\mathbf{c}$ 的夹角是大于 90° 还是小于 90°。综上所述，我们一共有三种情况：

$$\begin{aligned}&\text{如果 } \mathbf{n}\cdot\mathbf{c}>0\text{，射线沿法向方向}\\&\text{如果 } \mathbf{n}\cdot\mathbf{c}=0\text{，射线与直线平行}\\&\text{如果 } \mathbf{n}\cdot\mathbf{c}<0\text{，射线与法向相反}\end{aligned}\tag{4.52}$$

练习

练习 4.7.1　射线与直线和平面的交点

求射线 $A+\mathbf{c}t$ 射向物体 $\mathbf{n}\cdot(P-B)=0$（二维中直线或三维中平面）的击中时间和击中点。

(1) $A=(2,3)$，$\mathbf{c}=(4,-4)$，$\mathbf{n}=(6,8)$，$B=(7,7)$。

(2) $A=(2,-4,3)$，$\mathbf{c}=(4,0,-4)$，$\mathbf{n}=(6,9,9)$，$B=(-7,2,7)$。

练习 4.7.2　与平面相交的射线

找到射线 $(1,5,2)+(5,-2,6)t$ 与平面 $2x-4y+z=8$ 的交点。这在明暗处理算法中非常重要，它可以告诉我们通过一个实体对象表面上的一个多边形面片（平面）反射到观察点的光强是多少。

---

① 如果分子也等于 0，射线将完全处于直线（二维）或平面（三维）上。（为什么？）

练习 4.7.3 两个平面求交的结果是什么?

在这里我们先问一个几何问题,这个问题在我们学习透视投影和消隐时会有很实际的意义。我们知道两个平面的交点是一条直线,如何求到它?设两个平面为 $\mathbf{n}\cdot(P-A)=0$ 和 $\mathbf{m}\cdot(P-B)=0$,求它们交线的参数表达式。你可以发现最容易的做法是:

(1) 首先,得到其中一个平面的参数表达式,如令第二个平面为 $C+\mathbf{a}s+\mathbf{b}t$。

(2) 其次,将这一表达式带入第一个平面的点-法向量表达式,这样得到一个与参数 s、t 有关的线性方程。

(3) 然后用 t 把 s 表示出来,也就是得到 $s=E+Ft$。(求出 E 和 F 的表达式)

(4) 最后,所求直线可以写成 $C+\mathbf{a}(E+Ft)+\mathbf{b}t$。

注意:三维向量的正交向量不能唯一地定义,所以不能使用。

4.8 多边形求交问题

我们知道多边形是二维和三维图形学中的基本对象。在二维图形学中它们的直线边界使它们的描述和绘制都很简单。在三维图形学中,一个对象经常用一个多边形网格建模,就是用一组可以拼接在一起的多边形形成对象的表面。如果这个表面包围了一个封闭空间,这样的网格叫做多面体。事实上多面体的精确定义很难用这样的几句话说清,在这里只是根据读者经验中多面体的形状给出一个容易理解的定义。我们会在第 6 章深入探讨网格和多面体。

图 4.38 给出了一个二维多边形和一个三维多面体,它们需要在某个图形学应用中分析或绘制。这里有三个重要的问题:

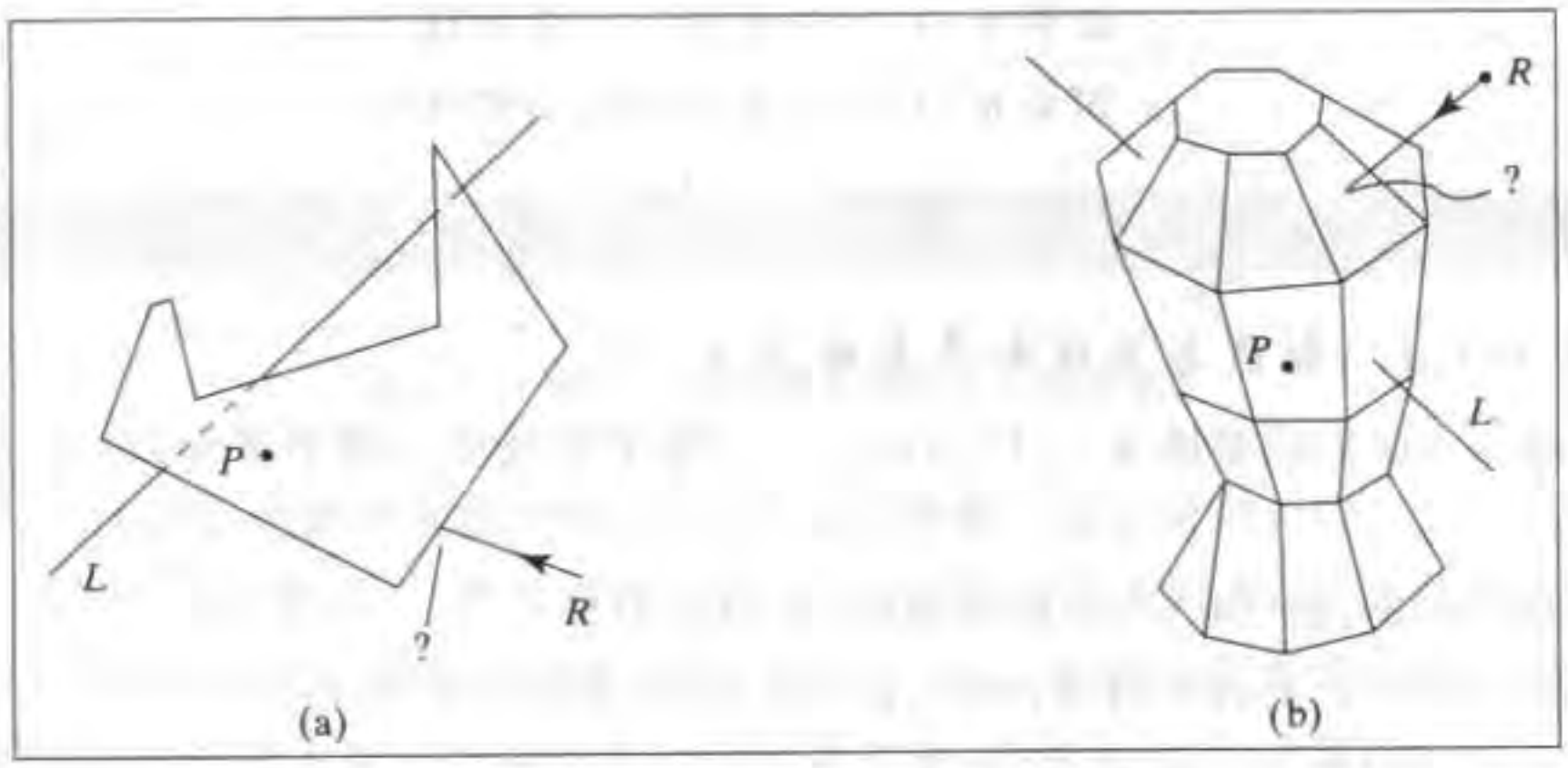


图 4.38 直线和多边形类对象的相交问题

(1) 给定一个点 P,它在对象的里面还是外面?

(2) 给定一条射线,它与对象的第一个交点在哪里?

(3) 给定一条直线,它在对象里面和外面的部分分别是什么?

一个简单的例子如下:直线 $3y-2x=6$ 的哪些部分在以 $(0,3)$,$(-2,-2)$,$(-5,0)$,$(0,7)$,$(1,1)$ 为顶点的多边形中?

4.8.1　处理凸多边形和凸多面体

一条直线和一个多边形或多面体求交的一般情况非常复杂，我们会在 4.8.4 小节中提到。但是如果多边形和多面体是凸的，问题就简单得多。这是因为凸多边形可以完全由一组边界直线描述，而在三维，一个凸多面体完全可以用一组边界平面描述。所以我们只需要检测直线与这些无限长(大)的直线或平面的相交情况即可。

图 4.39 是一个二维上的例子。图 4.39(a)是一个凸五边形，图 4.39(b)是它的边界直线 L_0，L_1 等。每条边界直线定义了两个半空间：包含五边形的内半空间和与五边形没有公共点的外半空间。图 4.39(c)是边界直线 L_2 定义的外半空间的一个示意。

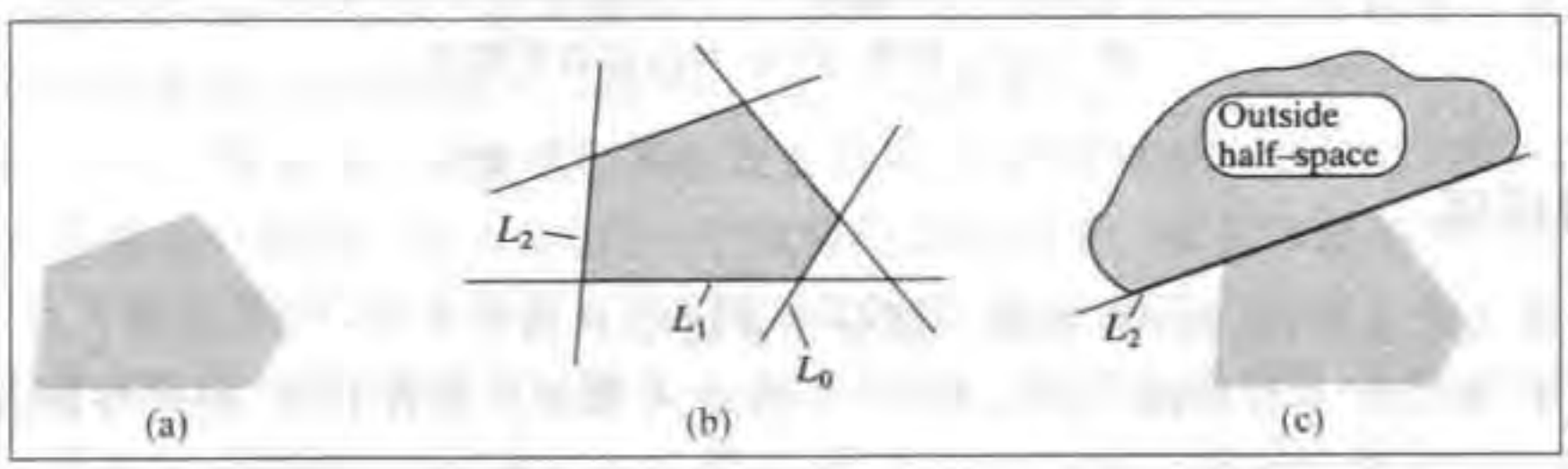


图 4.39　凸多边形和凸多面体

处理凸多边形的一大优势是我们只需要判断无限长直线的交点，而不需要检查交点是否超过某个端点——回想一下我们之前提到的求交测试的复杂性。

对于一个三维中的凸多面体，每个边界平面都定义了一个内半空间和一个外半空间，以及一个指向外部的法向量。多面体是所有内半空间的交(所有边界平面定义的内半空间的公共点集)。

4.8.2　射线与凸多边形的交点以及裁剪问题

我们在 4.7 节中介绍了一个求射线与直线或平面交点的方法。我们可以用这个方法找到射线与凸多边形或凸多面体的交点。

求交问题

射线 $A+\mathbf{c}t$ 在哪里与凸多边形 P 相交？

图 4.40 是一条射线 $A+\mathbf{c}t$ 与凸多边形 P 相交的情况。我们想知道射线与 P 的所有交点。因为 P 是凸的，所以射线只与 P 相交了两次：一次进入一次出去。设发生这两次相交的 t 分别为 t_{in} 和 t_{out}。如果我们能求出这两个击中时间，交点就很容易得到。

进入时的击中点：　$A+\mathbf{c}t_{\text{in}}$

出去时的击中点：　$A+\mathbf{c}t_{\text{out}}$　(4.53)

射线在 P 中的部分对应的 t 在区间$[t_{\text{in}}, t_{\text{out}}]$中。

注意求出 t_{in} 和 t_{out} 不仅能解决求交问题，还能解决裁剪问题。如果我们知道 t_{in} 和 t_{out}，我们就知道了直线 $A+\mathbf{c}t$ 的哪些部分是在多边形 P 中的。通常的裁剪问题描述如下：

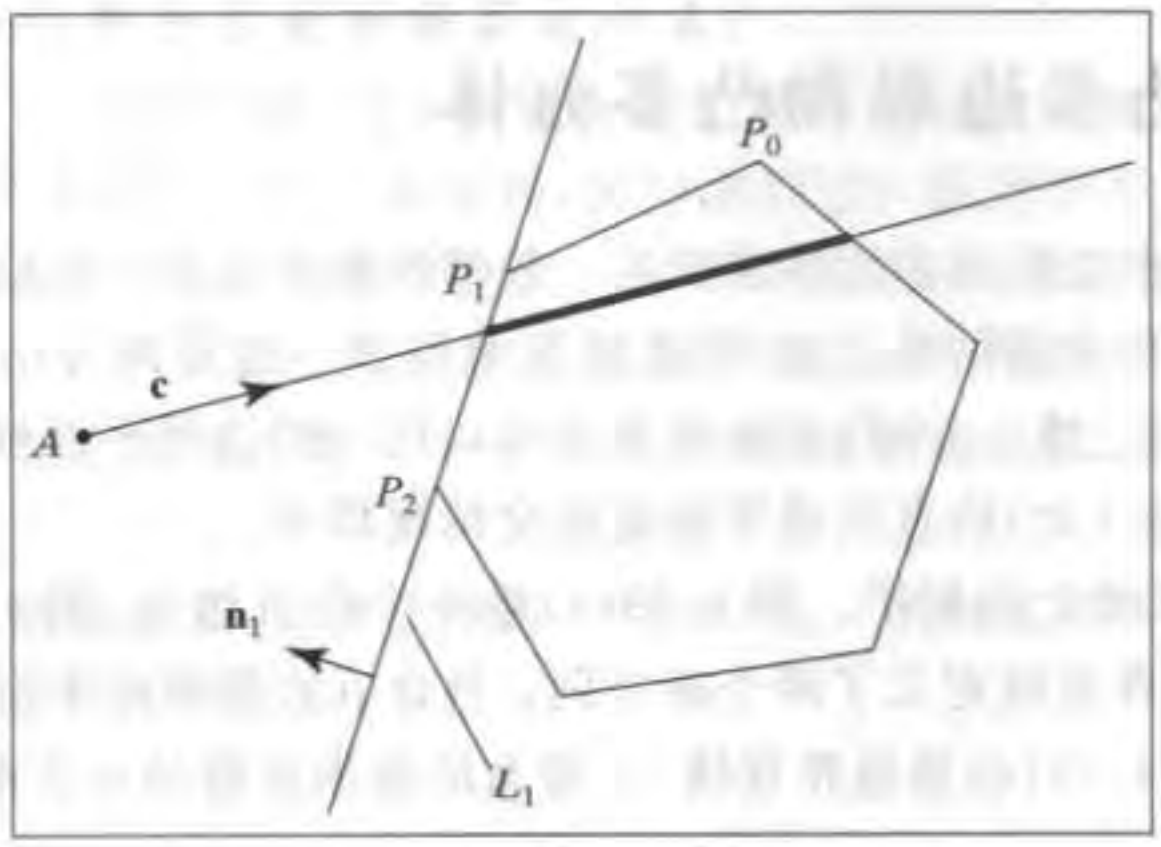


图 4.40 射线 $A+\mathbf{c}t$ 与凸多边形相交

裁剪问题

对于点 A 和点 C,线段 AC 的哪一部分在多边形 P 内?

图 4.41 给出了几种情况。图 4.41(a)中的点 A 和点 C 都在 P 外,但是线段 AC 的一部分在 P 内。

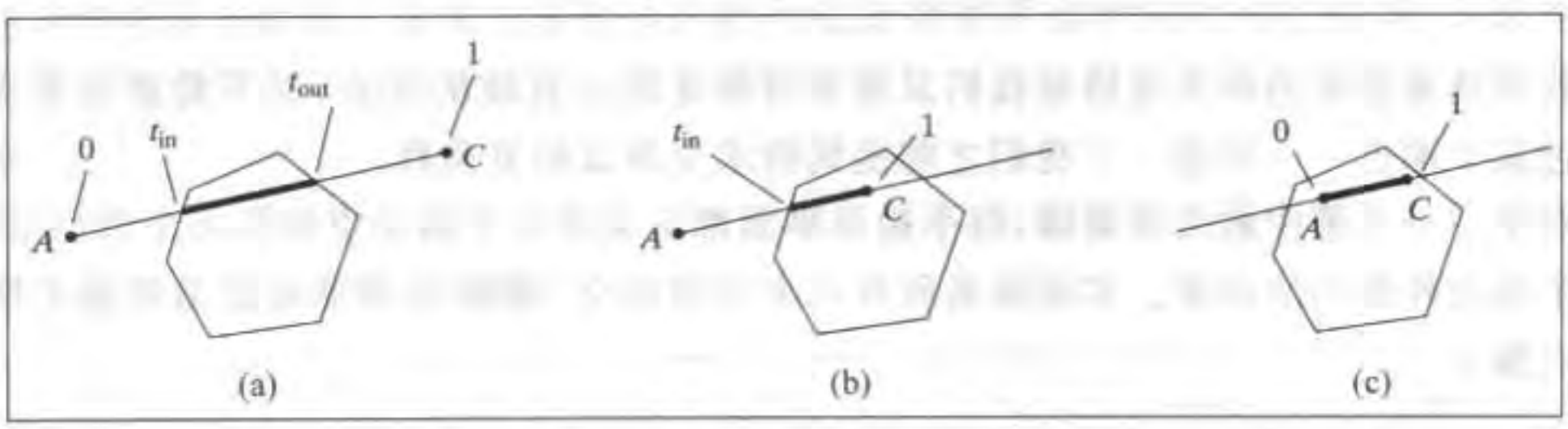


图 4.41 一条线段被一个多边形 P 裁剪

如果我们认为 AC 是射线 $A+\mathbf{c}t$ 的一部分,其中 $\mathbf{c}=(C-A)$,那么点 A 对应了射线上 $t=0$处的点,点 C 对应了 $t=1$ 处的点。注意点的仿射组合再次出现。这些特殊的 t 值都在图中标示了出来。为得到裁剪后的线段,我们用前面的方法计算 t_{in}和 t_{out}。裁剪后线段的端点为 $A+\mathbf{c}t_{in}$和 $A+\mathbf{c}t_{out}$。在图 4.41(b)中,点 C 在多边形 P 内,导致 t_{out}比 1 大。这样裁剪后的线段端点为 $A+\mathbf{c}t_{in}$和 C。在图 4.41(c)中,点 A 和点 C 都在 P 内,所以裁剪后的线段还是 AC。

一般情况下我们计算 t_{in}并比较它与 0 的大小。其中较大者作为裁剪后线段的第一个端点的参数值。同理,1 和 t_{out}中较小的值作为第二个端点的参数值。所以裁剪后线段的端点为:

$$A' = A + \mathbf{c}\max(0, t_{in})$$
$$C' = A + \mathbf{c}\min(t_{out}, 1) \tag{4.54}$$

现在,如何计算 t_{in}和 t_{out}? 我们必须依次考虑 P 的每一条边界直线与射线 $A+\mathbf{c}t$ 的交点。假设二维多边形的每条边界直线都存储为点-法向量形式$\{B,\mathbf{n}\}$。其中 B 是直线上一

点，**n** 是直线的**外法向量**（如图 4.36(a)中的 **n**）：它指向多边形的外侧。因为它指向外侧，所以(4.52)式的检测变成：

$$\begin{aligned}&\text{如果 } \mathbf{n}\cdot\mathbf{c}>0\text{，射线的方向是射出 } P\\&\text{如果 } \mathbf{n}\cdot\mathbf{c}=0\text{，射线与直线平行}\\&\text{如果 } \mathbf{n}\cdot\mathbf{c}<0\text{，射线的方向是射入 } P\end{aligned}\tag{4.55}$$

对每条边界直线我们可以得到：

(1) 射线击中边界直线的时间（通过等式(4.50)）。

(2) 击中时射线的方向是射入还是射出多边形（通过等式(4.50)）。

如果射线是射入的，我们知道射线真正进入 P 的时间（如果它确实进入了）不可能早于这个刚找到的射入时间。我们把进入 P 的最早可能时间保存为 t_{in}，对求交得到的每一个射入时间 t_{hit}，我们用 $\max(t_{in}, t_{hit})$ 作为新的 t_{in}。同理，我们把从 P 出去的最晚可能时间保存为 t_{out}，并对每次得到的射出时间 t_{hit}，用 $\min(t_{out}, t_{hit})$ 替换 t_{out}。

把区间 $[t_{in}, t_{out}]$ 看成是 t 的**候选区间**会很有帮助，它的意思是在此区间内的 t 所对应的射线上的点**可能**在对象内。图 4.42 是一个裁剪问题的例子。我们知道如果 t 在候选区间中，点 $A+\mathbf{c}t$ **可能**在 P 内，但是如果 t 不在候选区间中，该点绝对不会在 P 内。随着每一条边界直线被求交算法检测，候选区间随着 t_{in} 的增大或 t_{out} 的减小而变短，就像两端被一次次砍掉一样。开始时，我们把 t_{in} 初始化为 0，把 t_{out} 初始化为 1，这时候选区间为 $[0,1]$。

图 4.42　射线在多边形内的参数候选区间

裁剪算法如下：

1. 初始化候选区间为 $[0,1]$。[①]

2. 对每一条边界直线，用等式(4.50)计算击中时间 t_{hit}，并判断这一击中是射入还是射出：

- 如果是射入，令 $t_{in}=\max(t_{in}, t_{hit})$
- 如果是射出，令 $t_{out}=\min(t_{out}, t_{hit})$

如果 t_{in} 大于 t_{out}，说明射线与 P 完全不相交，结束检测。

3. 检测过多边形所有的边之后，如果候选区间非空，那么从 $A+\mathbf{c}t_{in}$ 到 $A+\mathbf{c}t_{out}$ 的线段是处于 P 内的。对于直线裁剪问题，这两点是裁剪后直线的端点；对于射线求交问题，它们分别是射线进入多边形和从多边形出去的点。

注意，只要候选区间为空，检测即告结束。这也称为**提前退出**：如果在计算过程中能提前知道射线在多边形外，我们立即结束检测以节省计算时间。

---

① 对于射线求交问题，如果射线的两端都延伸到无穷远，我们令 $t_{in}=-\infty$，$t_{out}=\infty$。在实际应用中给 t_{in} 赋一个很小的负值，给 赋一个很大的正值。

图 4.43 是一个裁剪的例子：我们要找到线段 AC 在多边形 P 内的部分。先将 t_{in} 初始化为 0，t_{out} 初始化为 1。射线从 $t=0$ 处的 A 点出发向 C 点前进，在 $t=1$ 处到达。我们用它依次和每条边界直线 L_0，L_1 等求交，并在必要时更新 t_{in} 和 t_{out}。

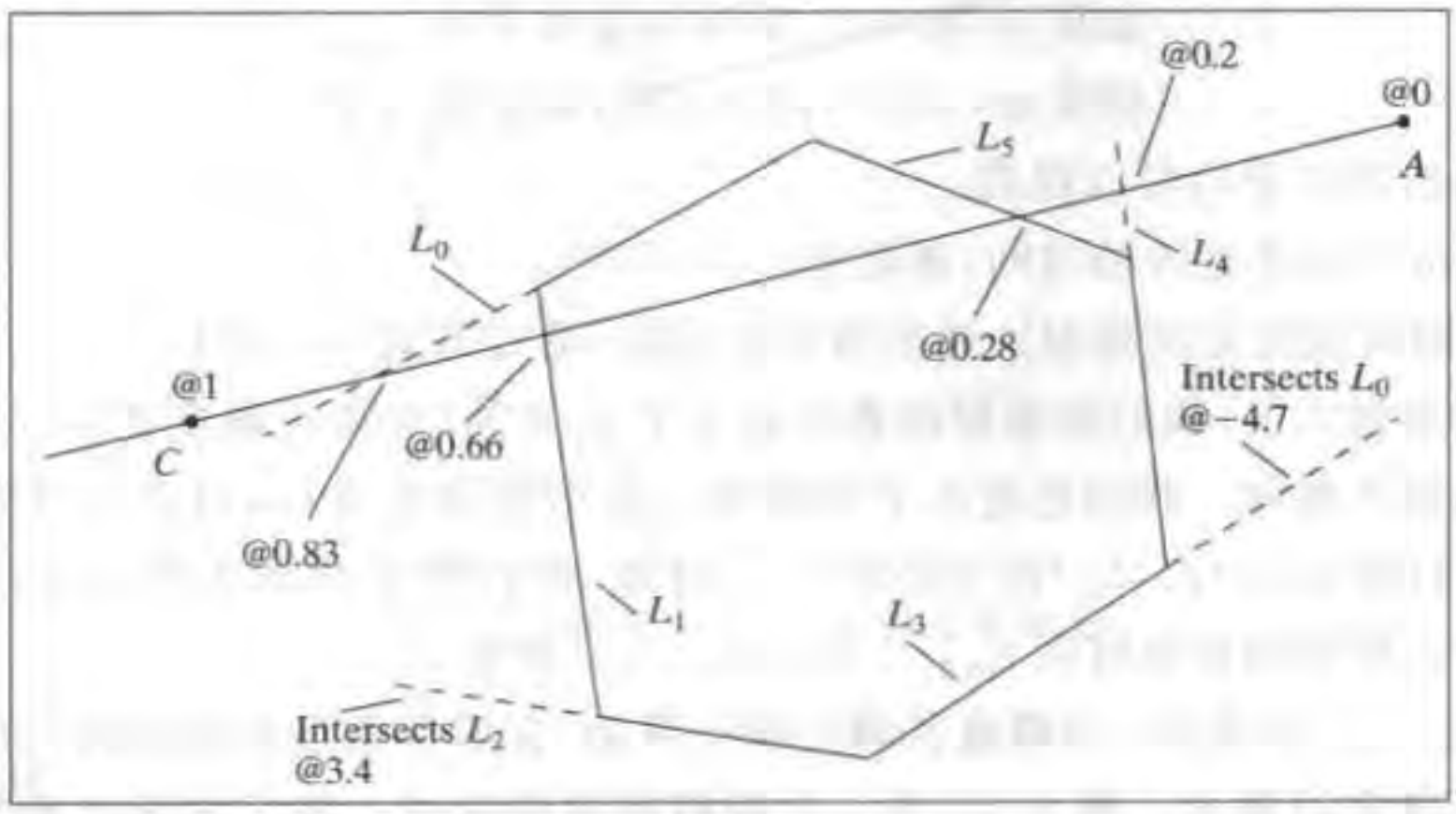


图 4.43 检测一条射线在凸多边形内的部分

例如，当我们检测到 L_0 时，我们得到一个 $t=0.83$ 时刻的击中点，且是射出多边形的。于是 t_{out} 更新为 0.83，候选区间变为[0,0.83]。然后测试与 L_1 的交点得到一个射出的击中点 $t=0.66$，它把候选区间减小到[0,0.66]。接下来测试 L_2，结果是一个 $t=3.4$ 的射出点，这对我们没有任何帮助，因为我们已经知道在 $t>0.66$ 处射线肯定在多边形外了。与 L_3 的测试得到 $t=-4.7$ 的射入点，比当前的 t_{in} 要小。与 L_4 的测试得到 $t=0.2$ 的射入点，于是 t_{in} 更新为 0.2。最后，与 L_4 的测试给出一个 $t=0.28$ 的射入点，到此我们结束遍历。最终的候选区间是[0.28,0.66]。事实上射线在 t 等于 0.28 到 0.66 处都在多边形内。

测试的直线	t_{in}	t_{out}
0	0	0.83
1	0	0.66
2	0	0.66
3	0	0.66
4	0.2	0.66
5	0.28	0.66

图 4.44 t_{in} 和 t_{out} 的更新过程

4.8.3 Cyrus-Beck 裁剪算法

现在我们用前面的方法来编写一个例程，使它能用任意凸多边形裁剪一条直线段。这个方法最先由 Cyrus 和 Beck 提出【Cyrus78】。在后面我们会介绍一个基于类似想法，但更高效的矩形窗口裁剪算法，它由 Liang 和 Barsky【Liang84】提出。

这个实现 Cyrus-Beck 裁剪算法的例程接口如下：

```
int CyrusBeckClip(Line& seg, LineList& L);
```

它的参数由一个被裁剪的线段 seg(它包含两个端点 seg.first 和 seg.second)和多边形的边界直线链表组成。它用 L 中的每条直线以前面提到的方法对 seg 进行裁剪，并把裁剪后的线段返回到 seg 中。(这就是 seg 必须引用传参的原因。)例程将返回：

- 0，如果线段完全不在 P 内(候选区间为空)。
- 1，如果线段的某个部分在 P 内部。

图 4.45 给出了 Cyrus-Beck 算法的伪代码。类型 LineSegment、LineList 和 Vector2 都是可以恰当存放问题中所需数据的数据结构类型(见练习)。变量 numer 和 denom 分别存放等式(4.50)的分子和分母：

$$numer = \mathbf{n} \cdot (\mathbf{B} - A)$$
$$denom = \mathbf{n} \cdot \mathbf{c} \tag{4.56}$$

```
int CyrusBeckClip(LineSegment& seg, LineList& L)
{
    double numer, denom;                    //用来表示与每条直线的击中时间
    double tIn=0.0, tOut=1.0;
    Vector2 c, tmp;
    <form vector: c=seg.second-seg.first>
    for(int i=0; i<L.num; i++)              //用每一条边界直线裁剪
    {
      < form vector tmp=L.line[i].pt-first>
      numer=dot(L.line[i].norm, tmp);
      denom=dot(L.line[i].norm, c);
      if(!chopCI(numer, denom, tIn, tOut)) return 0; //提前退出
    }
    // 调整线段的端点,先调整第二个端点
    if (tOut<1.0 )                          // 改变第二个端点
    {
      seg.second.x=seg.first.x+c.x * tOut;
      seg.second.y=seg.first.y+c.y * tOut;
    }
    if (tIn>0.0)                            //改变第一个端点
    {
      seg.first.x=seg.first.x+c.x * tIn;
      seg.first.y=seg.first.y+c.y * tIn;
    }
    return 1;                               // 保留了一部分线段
}
```

图 4.45　二维凸多边形对应的 Cyrus-Beck 裁剪算法的伪代码

注意必须先更新 seg. second 的值，因为在更新 seg. first 和 seg. second 的时候我们必须使用 seg. first 的旧值。

例程 chopCI()如图 4.46 所示。它用前面提到的变量 numer 和 denom 来计算射线与每条边界直线相交的时间，并判断射线在击中点的方向是射入多边形还是射出，然后把候选区间中不属于多边形内部的裁掉。

如果射线与直线平行，它可能完全在直线的内半空间或完全在外半空间。可以证明 $numer = \mathbf{n} \cdot (\mathbf{B} - A)$是判断这些情况的依据，见练习。

```
int chopCI(double& tIn, double& tOut, double numer, double denom)
{
    double tHit;
    if (denom<0)                          // 射线是射入的
    {
        tHit=numer / denom;
        if (tHit>tOut) return 0;          // 提前退出
        else if (tHit>tIn) tIn=tHit;      // 选择较大的
    }
    else if(denom>0)                      //射线是射出的
    {
        tHit=numer / denom;
        if(tHit<tIn) return 0;            //提前退出
        if(tHit<tout) tOut=tHit;          //选择较小的
    }
    else                                  // denom 为 0:射线与直线平行
    if(numer<=0) return 0;                //在外半空间
    return 1;                             // 候选区间非空
}
```

图 4.46 用一条边界直线裁剪

三维情况：用凸多面体裁剪直线

Cyrus-Beck 裁剪算法在三维中同样适用，且与二维类似。在三维空间中，窗口的边变成了在三维空间中一个平面上的凸多边形区域，线段处于三维空间中的直线上。chopCI()完全不需要改变(因为它只在 numer 和 denom 中使用点积)。当然，CyrusBeckClip()中的数据类型必须扩展为三维的，并且当调整线段端点时，必须增加对 z 分量的调整。

练习

练习 4.8.1 Cyrus-Beck 裁剪算法中的数据类型

为 Cyrus-Beck 裁剪算法中的数据类型 LineSegment、LineList 和 Vector2 提供合适的定义，使用结构体或类定义均可。

练习 4.8.2 numer<=0 时的情况

画出 chopCI()中的变量 numer 计算时用到的向量，并证明，若射线 $A+\mathbf{c}t$ 和边界直线 $\mathbf{n}\cdot(P-B)=0$ 平行，当且仅当 numer>0 时，射线才会完全落在边界直线的内半空间中。

练习 4.8.3 求裁剪后的直线

用 Cyrus-Beck 裁剪算法求以(2,4)和(20,8)为端点的线段，被顶点为(0,7)，(9,9)，(14,4)和(2,2)的四边形窗口裁剪后得到的部分。

4.8.4 更高级的裁剪问题

裁剪算法是计算机图形学的基础算法，人们已经开发出了许多高效的算法。图 4.47 给

出了几个裁剪算法的功能和特点的比较。图 4.47(a)列出了算法的名称，图 4.47(b)给出了在 OpenGL 中这些算法的实现程度，图 4.47(c)给出了每个算法适用的裁剪多边形和窗口。OpenGL 尝试实现了裁剪算法，但实际上实现的没有想象中的好(一般程序不会崩溃而是给出一个错误的绘制结果，尤其是应用程序企图用某个图案填充被裁剪的多边形的时候)。到此为止，我们已经介绍过用矩形窗口裁剪线段的 Cohen-Sutherland 算法和用任意凸多边形或凸多面体裁剪线段的 Cyrus-Beck 算法。Cyrus-Beck 算法可以推广到用任意凸多边形或凸多面体裁剪一条直线。但有时我们需要更复杂的裁剪。我们将简单介绍两个这样的算法，但是它们的详细内容超出本书的范围，在章末会提供可进一步阅读的文献。

(a) 裁剪算法	(b) 是否可用 OpenGL	(c) 描述
Cohen-Sutherland	是	用矩形或立方体(二维为正方形)裁剪直线段
Cyrus-Beck	尝试实现，结果不稳定	用任意凸多边形裁剪直线
Sutherland-Hodgman	否	用任意凸多边形裁剪任意多边形(凸或非凸)
Weiler-Atherton	否	用任意多边形裁剪任意多边形

图 4.47　不同裁剪算法的比较

Sutherland-Hodgman 裁剪算法与 Cyrus-Beck 方法类似，都是用凸多边形进行裁剪。但不同于 Cyrus-Beck 的裁剪对象——一条直线，它的裁剪对象可以是一个多边形(不必须是凸的)。最重要的是，它的输出也是一个多边形(或一个多边形的集合)。在裁剪过程中保持多边形结构的重要性，会充分体现在当裁剪后的多边形需要用某个图案或颜色填充的时候。如果多边形的每条边单独被裁剪，就不可能做到这一点。

Weiler-Atherton 裁剪算法可以用任意多边形 W 裁剪另一个任意多边形 P，无论它们是凸的还是非凸的。它可以输出 P 在 W 内部的部分(内部裁剪)，也可以输出 P 在 W 外部的部分(外部裁剪)。另外，P 和 W 内部都可以有洞。可以预见到，这个算法比我们介绍过的其余算法要复杂的多，对它的详细讨论超出了本书的范围。

尽管如此，在需要用到这些高级裁剪算法的应用程序中，Weiler-Atherton 算法是一个非常受欢迎的工具。

本章小结

向量提供了一种方便的方法来表示许多几何关系，它们支持的操作可以用代数的方法有效地处理几何对象。通过使用向量，计算机图形学中的许多算法得到了简化并变得更高效。由于大部分向量操作的表达式与向量所在空间的维度无关，使得我们可以得到在二维和三维空间中同样正确的结果。

两个向量的点积得到一个基本量，它可以简化向量长度和向量夹角的计算。它可以用来求一个向量到另一个向量的正交投影、三个点的外切圆圆心以及被反射后的射线方向。它经常被用来检测两个向量是否正交，或者更一般地，检测两个向量的夹角是大于还是小于 90°。二维向量 $\mathbf{a}$ 左转 90°得到的向量 $\mathbf{a}^{\perp}$ 也非常有用，特别是点积 $\mathbf{a}^{\perp}\cdot\mathbf{b}$ 提供了能说明 $\mathbf{a}$ 和 $\mathbf{b}$ 相关性的有用信息。

叉积也提供了与三维空间中两个向量夹角有关的信息,另外,它还得到一个与两个向量都正交的向量。它在计算平面法向量时会经常用到。

在开发算法的过程中,为相关的图形对象提供简洁的表达式是非常重要的。我们主要使用了两个基本形式——参数表达式和隐式表达式。参数表达式随着参数的变化可以遍历对象上每一个点,因此参数相当于对象上不同点的索引。隐式表达式给出了对象上点所满足的一个等式,而且仅有这些点满足。它通常的形式为:二维中 $f(x,y)=0$,三维中 $f(x,y,z)=0$,其中 $f()$是某个函数。给定一个点,把它带入函数 $f()$得到的值可以告诉我们该点在不在对象上。$f()$的符号还可以告诉我们点在对象的哪一边。本章介绍了图形学中两种基本平直对象的表示方法:直线和平面的表示方法。这两种对象的参数表达式和隐式都是线性的。它们的隐式形式可以写成对象的法向量和对象上的一个向量的点积,这一点有很大的启发作用。

向量可以进行任意的线性组合,但是点不行。点只允许进行仿射组合,否则当改变其所在的坐标系时——这在图形学中很常见——会产生混乱的结果。点的仿射组合在图形学中很有用,可以用来制作动画,并且生成贝塞尔曲线的内插(tweening)就是以它为基础的。

直线或射线的参数表达式在两条直线求交或射线与多边形、多面体求交中尤其有用。这些问题不但本身很重要,而且还是在图形学中地位显著的裁剪算法的基础。通读它们,并将解决它们的过程形象化是非常有价值的。

案例分析

案例 4.1 内插动画

(难度:Ⅱ)设计两个有趣的多边形,如果 4.48 中的 A 和 B。确保 A 和 B 的顶点数相同,可能需要在 B 点顶部的边上人为添加几个顶点。

(1) 编写一个与图 4.23 中的 drawTween(A, B, n, t)类似的例程,画出多边形 A 和 B 在 t 时刻的插值。

(2) 编写一个例程,可以绘制出这样的一个动画序列。令 t 从 0 逐渐变为 1,绘制出这一过程中 A 和 B 的内插动画序列,然后再让 t 逐渐减小到 0,如此反复,直到用户按下任意键中止绘制。这个动画序列就是从 A 变形成 B 再变回 A 的循环过程。利用 OpenGL 中的双缓冲技术使绘制的动画平滑。

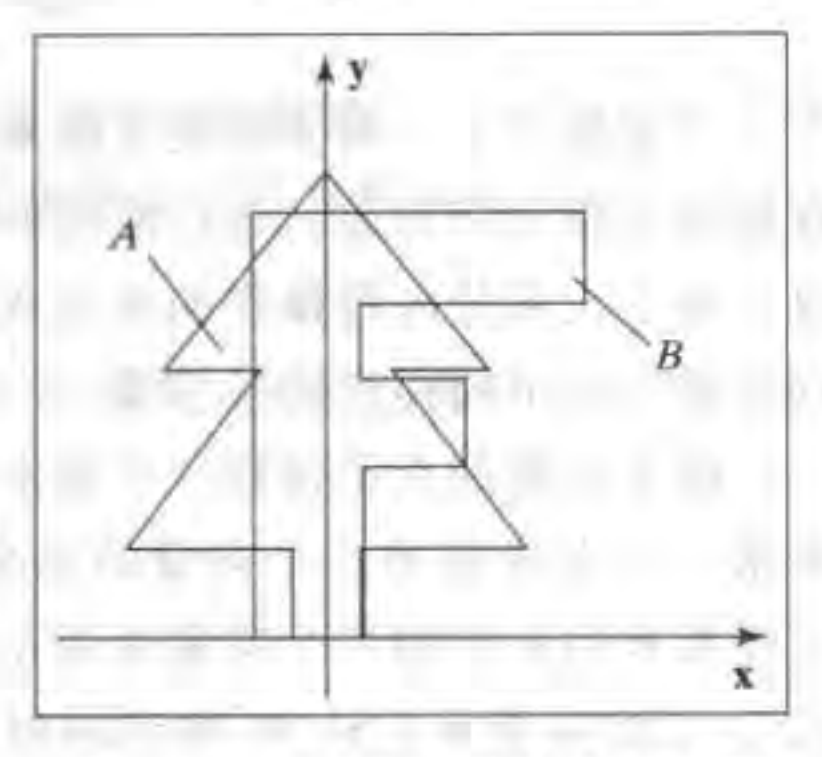


图 4.48 两个多边形的内插动画

案例 4.2 各种圆的问题

(难度:Ⅱ)编写一个应用程序使得用户可以用鼠标确定三角形的三个顶点,然后程序会用不同的颜色自动绘制出三角形、三角形的**内切圆**、**外切圆**和**九点圆**。程序还要支持用户用鼠标把三角形的任意顶点更改到

任意位置，并能立即绘制出新的三角形及前面提到的三种圆。

我们在 4.6.1 节中介绍了如何绘制外切圆，在这里我们会介绍如何求得内切圆和九点圆。

内切圆

这是一个在三角形内部且与三条边都相切的圆[1]。图 4.49(a)给出了三角形 ABC 和它的内切圆。

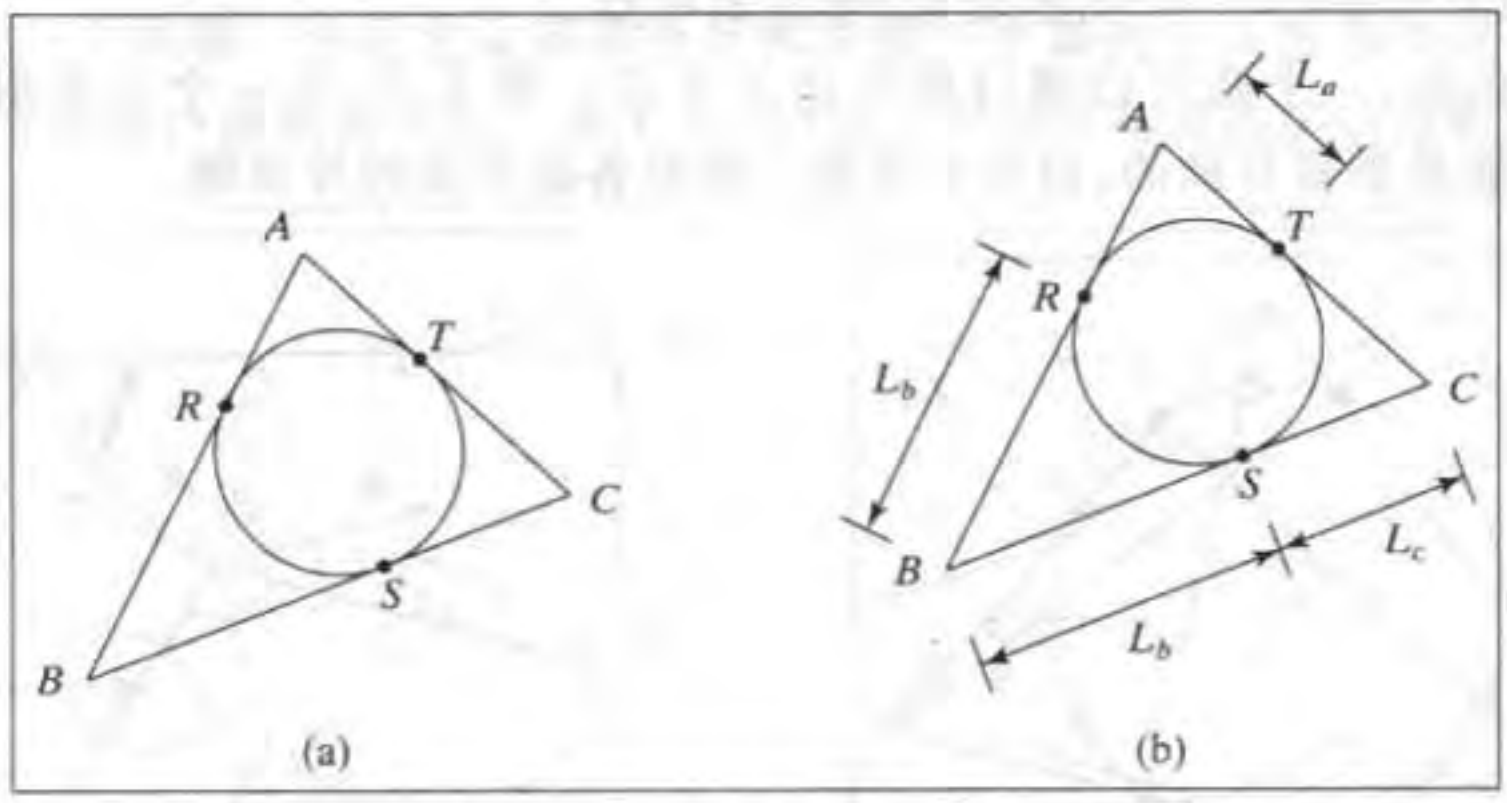


图 4.49　三角形 ABC 的内切圆是三角形 RST 的外切圆

与外切圆类似，问题的难点在于确定内切圆的圆心。一个直观的方法[2]是把 ABC 的内切圆简单地看成另外三个点 RST 的外切圆，如图 4.49(b)所示。

如此我们只需求得点 R、S 和 T 的坐标，然后用 4.6.1 中求外切圆的方法即可解决问题。图 4.49(b)中分别标出了点 R、S、T 到点 A、B、C 的距离。根据圆的对称性，距离 $|B-R|$ 和 $|B-S|$ 应该相等，同理还有另外两段相等的距离。于是我们有(使用等式(4.47)中 **a**、**b**、**c** 的定义)：

$$|\mathbf{a}| = L_b + L_a, \quad |\mathbf{b}| = L_b + L_c, \quad |\mathbf{c}| = L_a + L_c$$

可以用它们解出 L_a 和 L_b：

$$2L_a = |\mathbf{a}| + |\mathbf{c}| - |\mathbf{b}|, \quad 2L_b = |\mathbf{a}| + |\mathbf{b}| - |\mathbf{c}|$$

这样 L_a 和 L_b 皆为已知，于是 R、S 和 T 可由下式给出：

$$R = A + L_a \frac{\mathbf{a}}{|\mathbf{a}|}$$

$$S = B + L_b \frac{\mathbf{b}}{|\mathbf{b}|}$$

$$T = A - L_a \frac{\mathbf{c}}{|\mathbf{c}|} \tag{4.57}$$

(请验证这些等式!)

我们可以把由点 A、B、C 计算出点 R、S、T 的过程用一个简单例程 getTangentPoints(A, B, C, R, S, T)封装起来。这样做的好处是，如果我们已经有了一个例程 excircle()，

---

① 注意，求解内切圆也解决了求与平面上三条不共线直线都相切的唯一圆的问题。

② 由 Russell Swan 提出。

它可以计算三个点的外切圆圆心和半径,那么我们可以在求内切圆的例程中使用它。请实现这些工具。

九点圆

三角形有九个特殊点:

- 三个边的中点。
- 三个高的垂足。
- 三个顶点与垂心(三个高的交点)连线的中点。

值得注意的是,一个圆可以通过所有这 9 个点! 图 4.50 是一个三角形**九点圆**[①]的例子。九点圆可能是最容易画的,因为它就是三角形各边中点的外切圆。

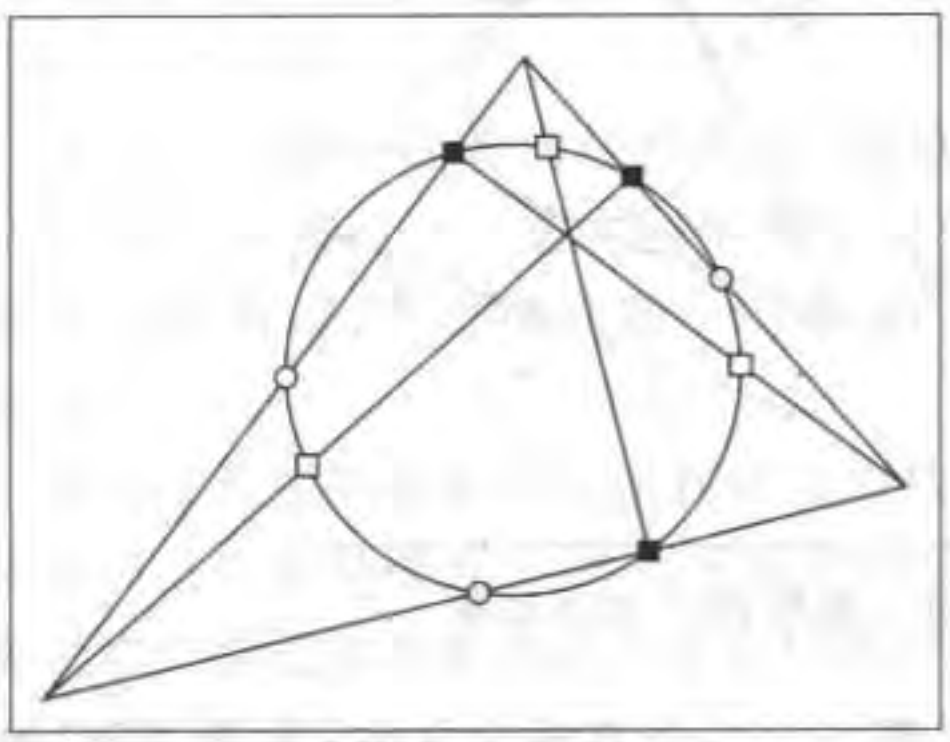


图 4.50 九点圆

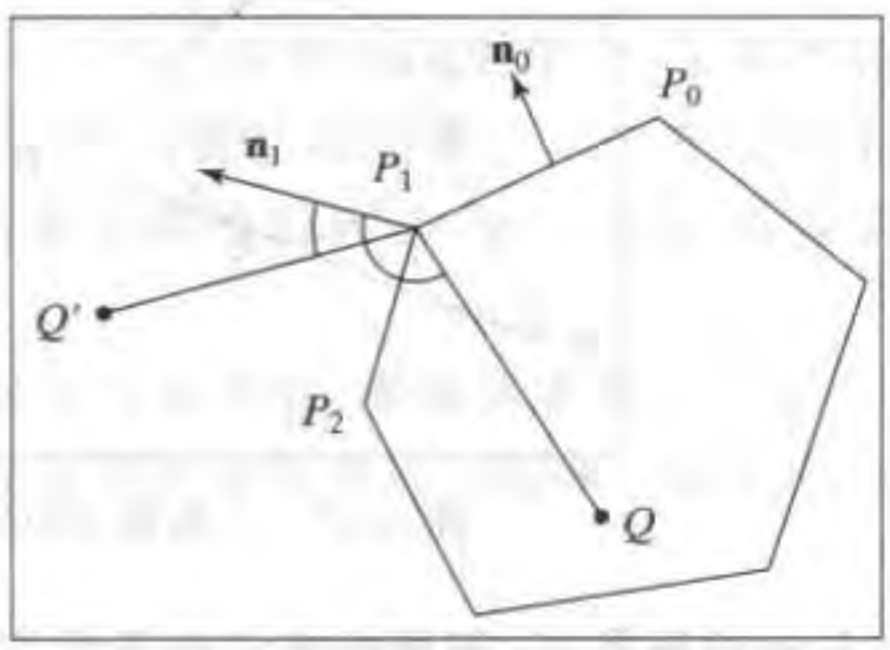


图 4.51 点 P 是否在多边形 Q 内

案例 4.3 点 Q 是否在凸多边形 P 内?

(难度: Ⅱ)假设给定一个凸多边形 P,它的顶点为 $P_0, P_1, \cdots, P_{N-1}$。然后,对于一个给定点 Q,需要判断 Q 是否在 P 内。不过通过 4.8.1 节对凸多边形的讨论,我们知道这相当于判断 Q 是否在 P 的每条边界直线所确定的内半空间里。对每条边界直线 L_i 我们只需要测试向量 $Q-P_i$ 与它的外法向夹角是否大于 90°即可,其中 P_i 是对应的多边形顶点。

事实:如果对所有 $i=0,1,\cdots,N-1$,都有 $(Q-P_i)\cdot\mathbf{n}_i<0$,点 Q 位于 P 的内部。

(4.58)

对经过 P_1 和 P_2 的边界直线的测试如图 4.51 所示。对于 P 内部的点 Q,与 $\mathbf{n}_1$ 的夹角大于 90°;对于 P 外部的点 Q',与 $\mathbf{n}_1$ 的夹角小于 90°。

编写并测试一个程序,可以生成随机一组点或由用户使用鼠标指定一组点来定义多边形 P(要确保 P 是凸的):

(1) 画出多边形 P。

(2) 然后生成并画出一些点 Q,可以是随机的也可以由用户用鼠标确定。测试这些 Q 点是否在 P 的内部。

(3) 根据测试结果输出"在内部"或"不在内部"。

---

① "这个圆是基础几何课程里第一个真正令人兴奋的地方。"

案例 4.4　密室中的光线反射(二维光线跟踪)

(难度：Ⅲ)本案例将应用本章介绍的一些工具和思想创建一个绚丽迷人却又简单的模拟程序。这个模拟程序实现了一种射线跟踪的过程，为便于形象化，该实现仅基于二维世界。我们会在第 12 章详细介绍三维光线跟踪。

这一模拟过程跟踪一个在密室墙壁之间弹来弹去的小弹球。图 4.52(a)是一个有 6 面墙的凸的密室 W 的横截面，它其中还有 3 个凸的柱子。弹球从 S 点出发，沿方向 c 按直线运动，直到遇到一个障碍。然后它从障碍处反弹并沿一个新的方向按直线运动。它将一直这样运动下去直到用户按某键中止。图 4.52(b)是一个小球运动的多边形轨迹的例子。

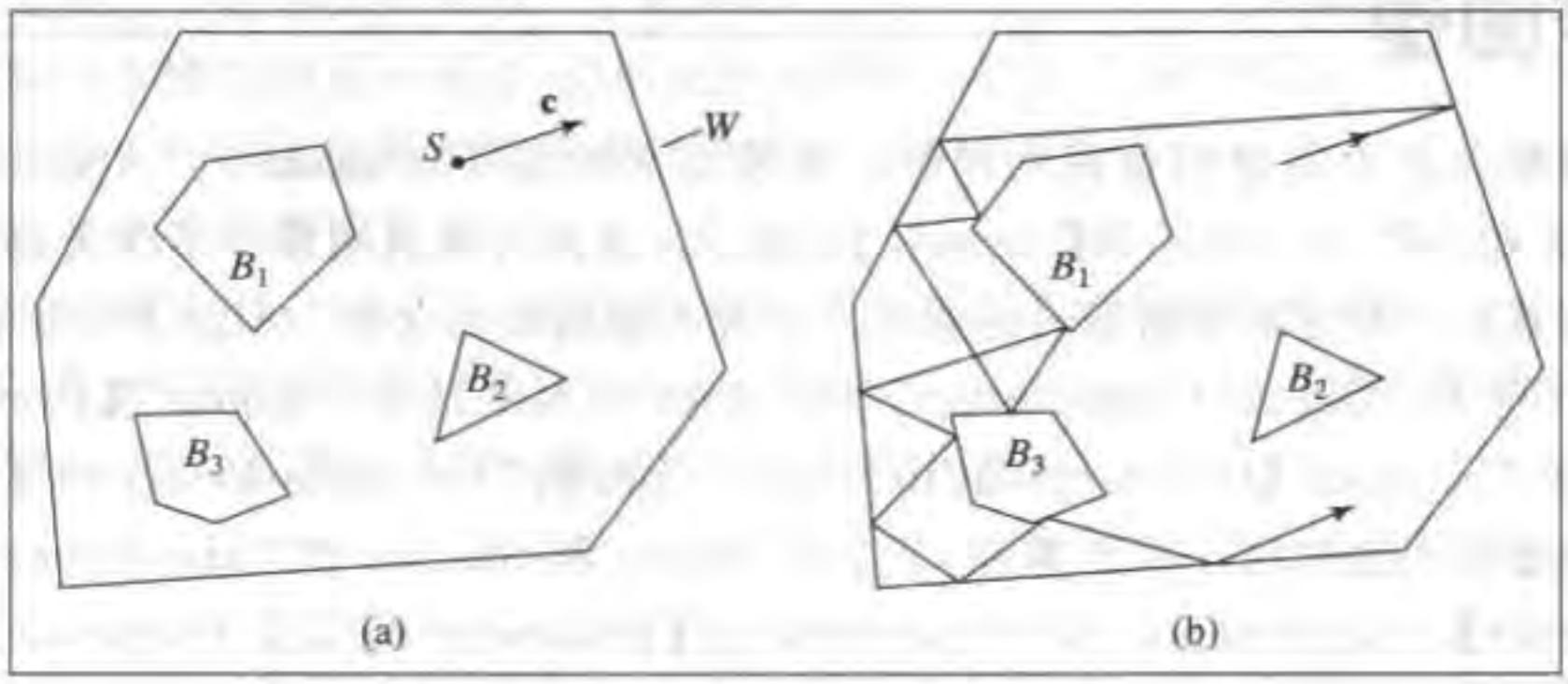


图 4.52　二维光线跟踪实验

对于射线的任意给定点 S 和方向 $\mathbf{c}$，跟踪它的轨迹需要两步操作：

- 找到射线击中的第一面墙。
- 计算射线经过这面墙(二维中的一条直线)反射后的新方向。

这两个操作都在本章中讨论过。注意每当一条新的射线产生，它的起点总是在某面墙或某个柱子上，也就是上一次的碰撞点。

我们把密室表示为一个凸多边形的列表：$pillar_0$，$pillar_1$，…，并令 $pillar_0$ 为密室的墙，一切都发生在它内部。这些凸多边形以点的数组形式存放。对每一条由点 S 出发沿方向 $\mathbf{c}$ 运动的射线，我们扫描所有的凸多边形，得到射线与每个凸多边形的交点情况。这一测试可以用 4.8.3 节的 Cyrus-Beck 算法完成。如果射线击中了多边形，击中时间应该是射线进入多边形的时间。

我们想知道射线最先击中了哪个多边形，可以通过在扫描所有多边形的过程中不断更新最早的击中时间来确定这一多边形。在这里我们只需考虑正的击中时间，因为负的击中时间对应的击中点在射线前进的相反方向上。当最早的击中点被找到后，可以绘制出从 S 到它的射线。

我们还需要知道在离开上一个击中点后，这条被反射过的射线的方向是什么。根据等式(4.22)，反射射线的方向 $\mathbf{c}'$ 跟入射方向 $\mathbf{c}$ 有如下关系：

$$c' = c - 2(c \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} \tag{4.59}$$

其中 $\hat{\mathbf{n}}$ 被击中的墙或柱子面的单位法向量。这样我们可以编写一段程序，当检测到射线击中一个多边形时，可以画出弹球的轨迹，它要么是从密室的内壁上反弹，要么是从密室

里柱子的某个面上反弹。我们可以读入一个保存多边形列表的文件得到密室和柱子的输入,然后让用户指定射线的起点和方向。

案例 4.5 Cyrus-Beck 裁剪算法

(难度:Ⅱ)编写并运行一段程序,实现用一个凸多面体裁剪一组直线的功能。用户可以用鼠标确定多边形的顶点(当按下"C"键终止增加顶点时,画出多边形并开始裁剪)。然后程序会生成一组直线段,它们的端点都是随机的。对每一条这样的直线段,首先把它整个绘制为红色,然后将裁剪得到的多边形内部的部分绘制为蓝色。

进一步阅读

许多书都有关于向量的很好的内容。比较有名的是 Hoffmann 的"About Vectors"。GRAPHICS 和 GPU Gems 系列【Gems05】收集了许多由计算机图形学工作者提供的,关于向量和几何算法中新方法和新结论的源码。另有 3 篇较好的文章:Alan Paeth 的"A Half-Angle Identity for Digital Computation: The Joys of the Half Tangent"【Paeth91】,Ron Goldman 的"Triangles"【Goldman90】,和 Lopez-Lopez 的"Triangles Revisited"【Lopez92】。还有三本书更深入地探讨了几何算法,它们是 Moret 和 Shapiro 的"Algorithms from P to NP"【Moret91】,"Algorithms and Programming"【Holloway02】以及 Preparata 和 Shamos 的"Computational Geometry, An Introduction"【Preparata85】。

第5章 物体变换

本章学习目标

- 构造从一个物体变换到另一个物体的工具
- 介绍仿射变换的基本概念，仿射变换是旋转、缩放、平移的组合
- 在计算机程序中编写函数用来对物体实施仿射变换
- 开发坐标系变换的工具
- 学习怎样使用 OpenGL 去设置摄像机用来对三维场景进行绘制
- 学习怎样使用场景设计语言(Scene Design Language, SDL)设计场景，如何编写程序读取 SDL 文件并且将它描述的场景绘制出来

预览

本章的主要目的是逐步介绍在 OpenGL 程序中广泛使用的一类特殊的变换族，称为仿射变换(Affine Transformation)。

5.1 节讲述在计算机图形学中使用二维和三维变换的动机，同时提出一些基本的定义。

5.2 节定义了二维仿射变换以及当前变换(Current Transformation, CT)的概念，并且用矩阵的形式表示出来。使用坐标系可以明确地表示出哪些物体发生了变化，以及它们是怎么被改变的。本节还展示了基本的仿射变换是如何进行缩放、旋转、平移和剪切操作的。5.2.5 节表明将很多个仿射变换组合在一起也可以成为一个变换，并且这个变换也是仿射变换，同样可以用一个矩阵来表示。5.2.7 节讨论了仿射变换的一些关键性质——最主要的是它可以保持物体的共线、共面和平行的关系——同时说明为什么它在计算机图形学中这么流行。

5.3 节将这些概念扩展到三维仿射变换，并且表明所有二维的性质在三维情况下仍然成立。三维变换——尤其是三维旋转——比二维旋转更加复杂和更加难以可视化，所以我们特别着重于描述不同的旋转操作以及几种旋转形成的组合操作。

5.4 节讨论了点变换和坐标系变换之间的关系。

5.5 节介绍了在一个图形程序中如何对变换操作实施管理，以及这些变换如何简化一个图形程序所需要的操作。我们列举了很多的例子来描述如何对变换进行建模并使用当前变换。

5.6 节讨论三维场景的建模并使用 OpenGL 将其绘制出来。一台摄像机由它的位置和朝向定义，以便它能够获得想要的场景截图。一些人(包括大部分写计算机图形学教材的人)喜欢将它想象成一个针孔摄像机，但是我们将会使用一个更加生动的方法，它可以让人感觉截图是从一架在场景中飞驰而过的飞机上拍摄的。实际上，在后面的讨论中我们将会

使用一个更加具有描述性的词语,即 Jib 摄像机。这是借用视频制作工业中的术语,在视频制作中制作人会指挥控制 Jib 摄像机在场景中的运动轨迹。应用程序的编写人员可以指定一个特殊视点用来产生特殊的视觉效果(如果想要的视觉效果已经得到了,那么下一步的计划就是要让场景中的物体看起来更加真实,好像物体由某种特殊的材料做成,并且要打上光照)。对于动画的情形,视点可能在连续的帧中进行变化,就好像一台摄像机飞过了一个场景。本节讨论了场景中的变换是怎样对物体的位置和大小进行调整的。我们建模并绘制了一些三维场景,同时设计了完成此项工作所必需的程序代码。本节还要介绍场景描述语言(SDL),同时展示如何写一个应用程序,使其可以绘制 SDL 描述的任何场景。这就需要一定数量的类(SDL 的编译器已经提供了)用来支持读取和分析 SDL 文件,创建可以绘制的物体列表。这些类已经经过测试和编译,可以从本书的网站获得。

本章最后以一些精心制作的案例作为结束,提供给读者一个可以在图形程序中使用仿射变换的实践机会。此外还描述了如何将一个仿射变换分解为一些基本的仿射变换,并且利用一次旋转操作等价于三次连续剪切操作的原理开发一段应用程序用来快速地绘制出圆弧。

5.1 概　　述

仿射变换是计算机图形学的基石,同时也是 OpenGL 的中心内容。它也是很多程序员公认的难点,因为正确运用它们的确很困难。一个特别微妙的地方是关于点和向量的混淆。点和向量非常相似,在程序中经常使用相同的数据结构表示它们,比如在当前坐标系下三元组(3.0, 2.5, −1.145)既可能表示一个点也可能表示一个向量。这就有可能使得程序产生错误,而且很难被察觉出来,主要原因是点和向量的变换方式并不相同。为了清晰地把它们区分出来,我们需要使用第 4 章介绍过的齐次坐标系统。

5.2 几何变换初步

我们至少已经看过一些二维变换的例子了。比如在第 3 章中,窗口和视口的转换就是使用缩放和平移,从而让窗口中的物体适应它们在视口中的最终尺寸和位置。

我们想要以这些想法为基础,实现对目标物体尺寸、方向和位置的更灵活的控制。下一步,我们要基于功能强大的仿射变换开发需要的工具。这些工具在二维和三维都可以进行操作(下面我们看一些使用仿射变换的例子)。

在每个例子中,读者都应该注意变换是如何节省编程者的精力的。要知道一般来说,构建一个场景最难的地方在于设计出编程者想要出现在场景中的各种物体。

图 5.1(a)展示了一个房子模型变换前后的两个版本。图 5.1(a)展示的是一个二维例子,图 5.1(b)部分展示的是一个三维例子。房子模型尺寸被缩小了,并且旋转了一个小角度,同时向右上移动了一段距离。整个变换是 3 种以上基本变换的组合:旋转、缩放和平移。图 5.1(b)展示了一个三维房子模型做相似的变换前后的位置:三维模型上面每一个

点都经过了缩放、旋转和平移。

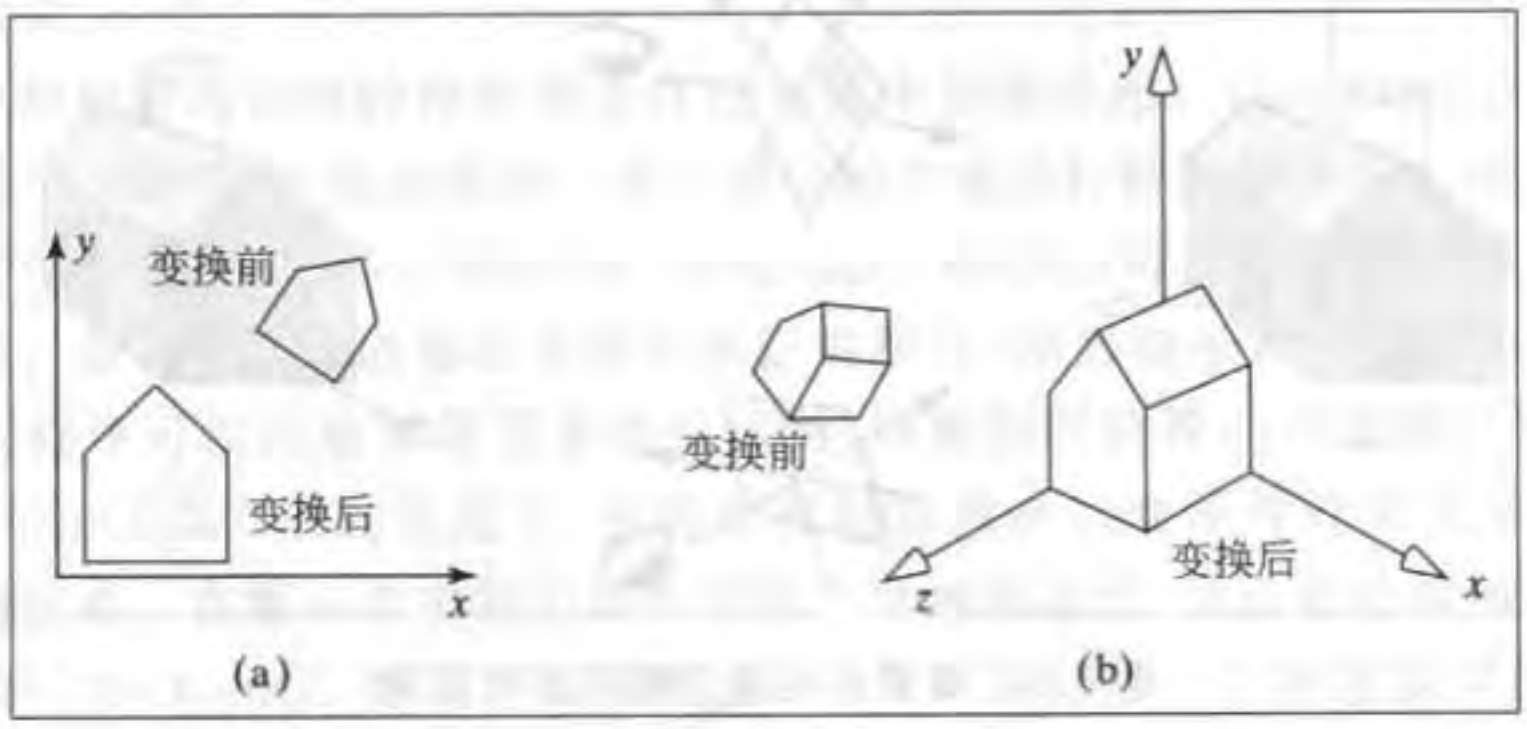


图 5.1　物体变换前和变换后的图像

变换在很多情况下是很有用的：

a. 我们可以用很少的物体组成一个场景，如图 5.2 所示。每一个拱形在自己的坐标系下只需要被设计一次。然后就可以使用适当的变换，在场景中放置很多大小位置不同的拱形例子。图 5.3 是一个三维例子，整个场景是由很多经过缩放和移位的立方体组成的。

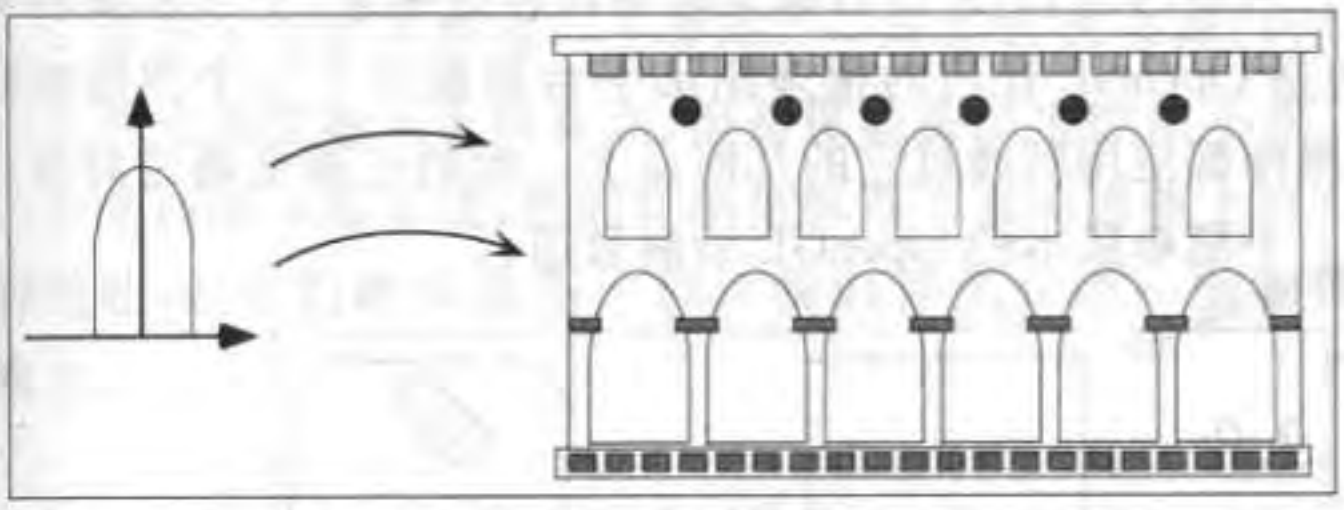


图 5.2　一幅简单实例重复组成的图像

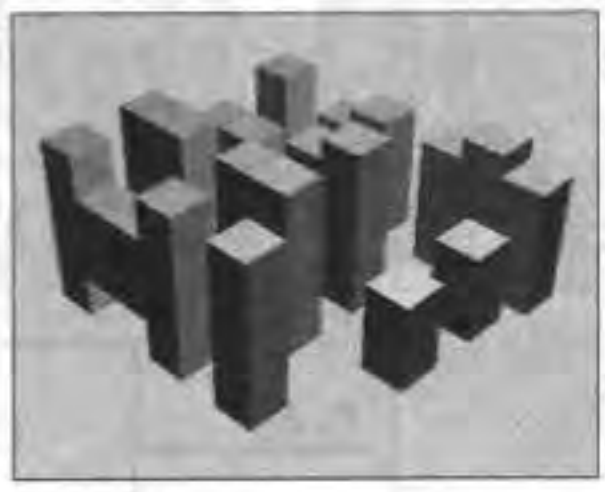


图 5.3　由基本图元构成的三维场景

b. 一些物体表现出对称性，如图 5.4 中的雪花。我们可以设计一个基本的图案，然后经过反射、旋转和平移组合出整个雪花图形(回想 3.2 节的例子)。

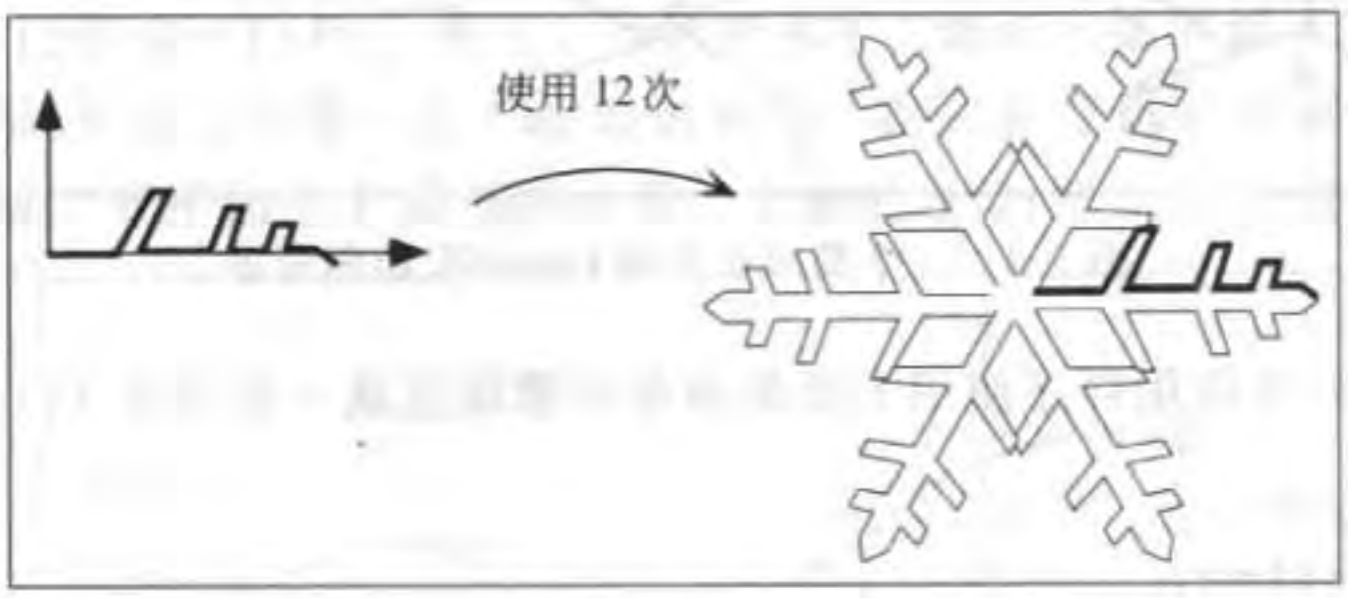


图 5.4　一个“基本图案”和它构成的图形

c. 设计人员可能想从不同的有利位置观察物体，并且得到每个位置的图片。可以使用相同的摄像机旋转观察场景。但是如图 5.5 中提到的，为了拍摄快照，如果不动场景，只是移动摄像机到不同的方位将会显得更加自然。改变摄像机的方位可以通过仿射变换实现。

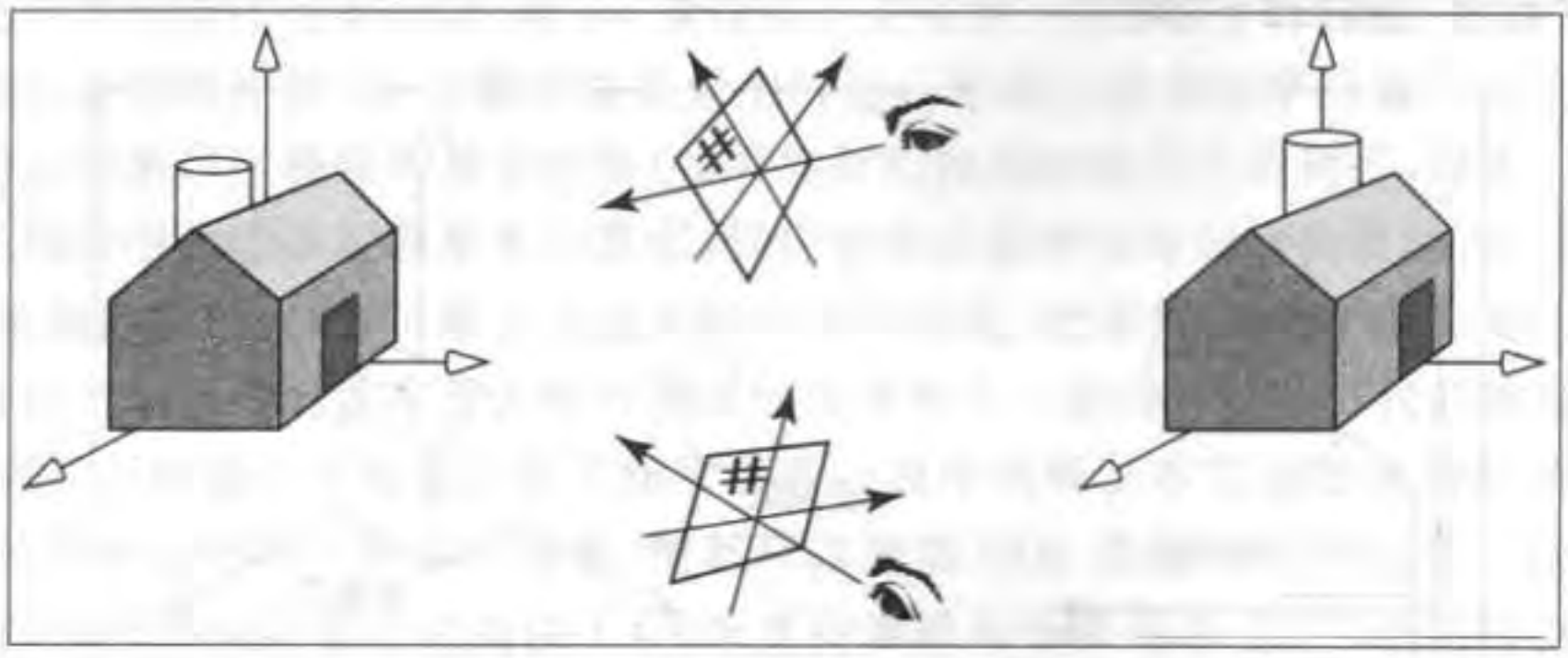


图 5.5 场景从不同角度观察的效果

d. 在计算机动画中,经常有几个物体之间的相对运动。我们可以通过平移和旋转这些物体的局部坐标系,得到这种动画效果。

我们的主要目的是什么? 使用 OpenGL 进行变换

本章前面几节主要讲述了一些仿射变换的基本概念,以及在二维和三维情况下怎样使用缩放、旋转、平移得到某种几何效果。当然,我们的终极目标是绘制出经过缩放、平移和旋转等变换后的物体,生成想要的场景。包括 OpenGL 在内的很多图形平台都提供了一个图形绘制管道,或者说是一个操作顺列,这些操作都应用到通过它的几何点上。绘制过程是通过对每一个点进行操作来实现的。图 5.6 是一个简单版本的 OpenGL 绘制管道。

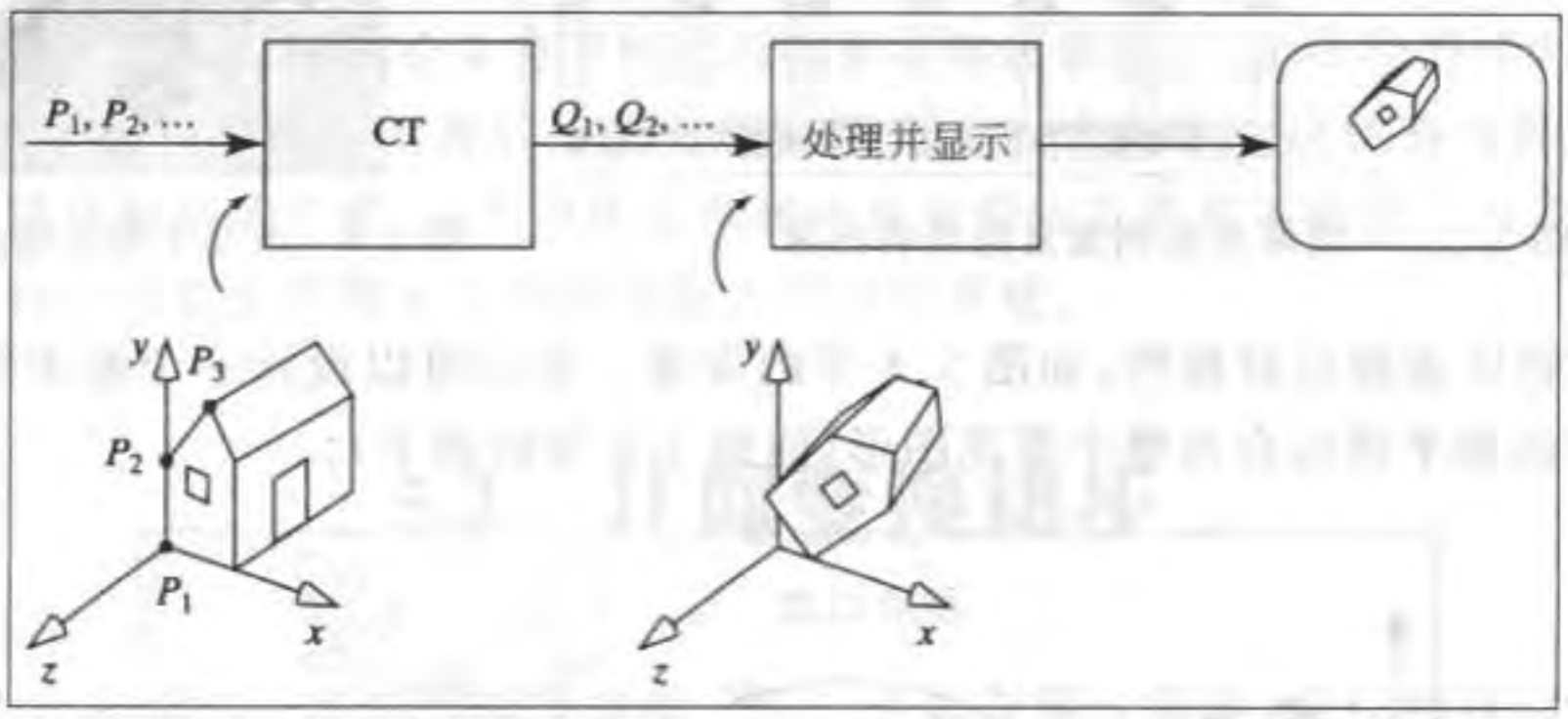


图 5.6 一个简单版本的 OpenGL 绘制管道

如图中所示,一个应用程序使用下面的命令向管线发送一系列点 P_1, P_2, P_3,…:

```
glBegin(GL_LINE);
    glVertex3f(…);
    glVertex3f(…);
    glVertex3f(…);
    …
glEnd();
```

如图中所示,这些点首先进行一次当前变换(The Current Transformation, CT),这个变换将会把这些点变成另一些点 Q_1、Q_2、Q_3。如果 P_1 描述了一些几何体,那么 Q_1 就描述

了这个几何体经过变换以后的几何体。然后这些点再被送到其他步骤里，最后在显示器上绘制最终的图像。

图 5.6 中绘制管道省略的部分将会在第 8 章中详细介绍。OpenGL2.0 以前的版本中绘制管道是固定功能的，也就是说每一步以特定的方式进行特定的操作。使用 OpenGL2.0 的新特性和着色语言(OpenGL Shading Language，GLSL)可以在某种程度上重新对绘制管道进行布置。编程者不仅能够改变操作执行的顺序，而且能够使得操作变得可编程。这就意味着应用程序可以向绘制管道发送数据，然后根据其特殊的用途进行自定义的处理。这样在遵循 OpenGL2.0 版的规范下，使得富有创新精神的软硬件开发人员可以开发出新的算法和绘制技术。在第 8 章中我们将介绍两个可编程单元，顶点着色器和片段着色器，同时介绍它们如何与 OpenGL 绘制管线相配合(一个片段包括一个给定像素的深度、颜色和纹理特性，第 8、9 章将介绍更多关于片段的知识)。

因此当前变换在对物体操作的过程中起了至关重要的作用，同时应用程序员必须知道怎样调整当前变换 CT 来得到想要的变换操作。在描述完仿射变换的基本理论后，我们会转到 5.5 节去看看它在实际中是如何操作的。

物体变换和坐标变换

有两种方式去看待一个变换：一种是物体变换，另一种是坐标变换。物体变换使用同一个规则改变物体上所有点，但是保证底层坐标系不变。坐标变换按照原坐标系统定义了一个全新的坐标系统，然后在新坐标系下表示物体上所有的点。两种变换紧密联系，各有各的长处，但它们的实现不一样。我们首先介绍一些物体变换的思想，然后过渡到坐标变换上。

5.2.1 点和物体变换

这里我们先介绍变换的一般性概念，然后再专门探讨仿射变换。

一个变换(叫做 $T()$)，将空间中(二维或者三维)一点 P 按照某一指定的等式或算法变为一个新的点 Q，记作 $Q=T(P)$。图 5.7 分别展示了二维和三维的例子。

如图 5.7 所示，平面上任意一点 P 被映射到 Q。我们说 Q 是 P 在映射函数 T 下的像。图 5.7(a)表示了在二维平面上 P 点被映射到一个新的点 Q；图 5.7(b)表示了在三维空间中

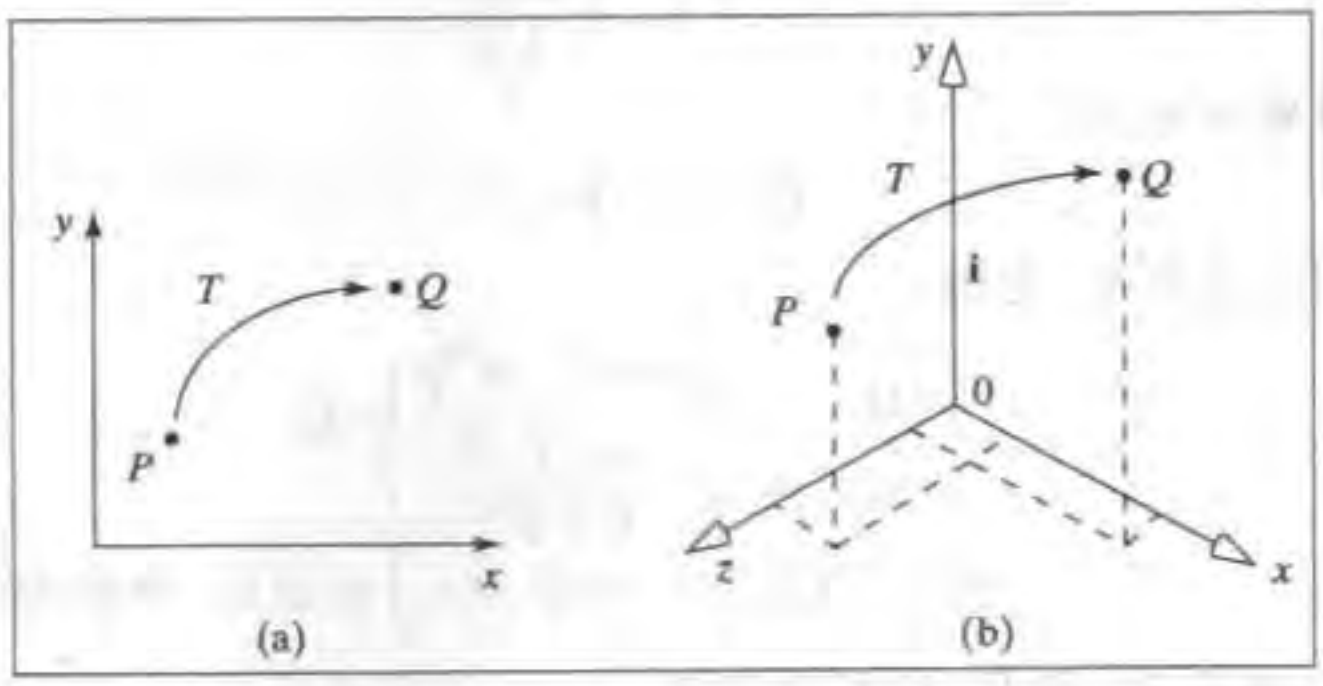


图 5.7 一个点映射到新的位置

一点 P 被映射到一个新的点 Q。我们变换一个物体是通过变换物体上面每一个点实现的，当然每个点都使用相同的函数 $T()$。我们可以一次映射整个点集，这个点集可能是一条线或者是一个圆上的所有点。直线 L 在 T 下的像是由 L 上所有点的像组成的。

大部分映射是连续的，因此一条直线的像仍然是某种形状的连续曲线，尽管它可能不是一条直线。我们将会看到，仿射变换将仍然保持是直线。也就是说在变换 T 下，直线仍然是直线，这就限制了使用仿射变换得到的形状的丰富性。但是和仿射变换其他性质比起来，这个性质更加重要。本章大部分内容将会集中在仿射变换上面，但是如果想要得到特殊的效果使用其他变换也是可以的。例如，图 5.8中的极其扭曲的图像就不能通过仿射变换得到(原始图形是在圆中的莱昂纳多)。这种高度扭曲的变换可能会用在特殊的视觉效果中或者用于强调物体的某个重要特征。

图 5.8 一幅图像的扭曲效果

为了清晰起见，我们在执行变换操作的时候使用一个显式坐标系。回忆第 4 章的内容，一个坐标系包含一个特殊的点 ϕ 称为**原点**，和某些作为坐标轴的相互垂直的向量(二维情况为下 $\mathbf{i}$ 和 $\mathbf{j}$，三维情况下为 $\mathbf{i}$、$\mathbf{j}$ 和 $\mathbf{k}$)。

首先来看二维情况，它很容易进行可视化。无论我们使用什么坐标系，P 和 Q 都可以用齐次坐标 $\tilde{P}$ 和 $\tilde{Q}$ 表示为：

$$\tilde{P} = \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix} \quad \tilde{Q} = \begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix}$$

回忆一下，这就表示点 $\tilde{P}$ 的位置是 $\tilde{P} = P_x\mathbf{i} + P_y\mathbf{j} + \phi$，$\tilde{Q}$ 同理。P_x 和 P_y 是 $\tilde{P}$ 的齐次坐标分量。现在要从原点移动到 $\tilde{P}$ 点，就要沿着 i 轴移动距离 P_x，沿着 j 轴移动距离 P_y。

假设一个点 $\tilde{P}$ 经过变换 T 得到点 $\tilde{Q}$：

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = T \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix} \tag{5.1}$$

或者更加简单地表示为：

$$\tilde{Q} = T(\tilde{P})。 \tag{5.2}$$

函数 $T()$可能会很复杂，比如：

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(P_x)e^{-P_y} \\ \dfrac{\ln(P_y)}{1+P_x^2} \\ 1 \end{bmatrix}$$

因此 Q_x 和 Q_y 都是 P_x 和 P_y 的复杂组合。

这些变换或许会产生有趣的几何效果，但是建立仿射变换时我们将自己局限在一些简

单的函数族上面，它们都是 P_x 和 P_y 的线性函数。这个特性刻画出了仿射变换。

5.2.2 仿射变换

仿射变换的一些性质、优点及缺点

仿射变换是计算机图形学中最普通的一种变换，它有很多很有用的性质。其中之一是仿射变换可以很容易地对物体实施缩放、旋转和平移操作。一系列仿射变换可以组合成一个单一的仿射变换，更重要的是，仿射变换可以使用一个紧凑的矩阵形式表示出来。后面我们会看到，对一个二维向量或者点实行仿射变换的矩阵是一个 3×3 矩阵。之所以是 3×3 矩阵是由于向量和点使用了齐次坐标的表示形式。

仿射变换有一个简单的形式。因为 $\tilde{Q}$ 的坐标是 $\tilde{P}$ 坐标的线性组合，因此变换得到的点 $\tilde{Q}$ 可以写成：

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11}P_x + m_{12}P_y + m_{13} \\ m_{21}P_x + m_{22}P_y + m_{23} \\ 1 \end{bmatrix} \tag{5.3}$$

其中有 6 个给定的常量 m_{11}、m_{12} 等。Q_x 的坐标包含 P_x 和 P_y，Q_y 也是如此。后面我们会看到，这种在 x 和 y 分量之间的交叉作用会带来旋转和剪切的效果。

等式(5.3)所示的仿射变换有一个很有用的矩阵表示，这会帮助你更好地思考：

$$\begin{bmatrix} Q_x \\ Q_y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ 1 \end{bmatrix} \quad \text{或者 } \tilde{Q} = \widetilde{MP}\text{（显式表达形式）} \tag{5.4}$$

这里的 Q 和等式(5.3)中的是一样的，都是 M 和 P 相乘的结果。在实际应用中，请注意矩阵的第 3 行是怎样使 $\tilde{Q}$ 的第三分量置为 1 的。对于一个仿射变换矩阵，其第 3 行经常都是(0，0，1)。

例 5.2.1 变换的使用

使用矩阵表示一个仿射变换：

$$\begin{bmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix}$$

计算 $P=(1, 2, 1)$的像 Q。

解：

$$Q = \begin{bmatrix} 8 \\ 2 \\ 1 \end{bmatrix} = \begin{bmatrix} 3 & 0 & 5 \\ -2 & 1 & 2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix}$$

直接用 P 和矩阵相乘，请留意，正如所期望的那样，Q 是一个点。

5.2.3 二维基本仿射变换的几何效果

仿射变换会产生怎样的几何效果呢？它产生四种基本变换的组合效果：(a)平移，(b)缩放，(c)旋转，和(d)剪切。因为这些变换每次只是产生了一种变换效果，因此它们被称为基本变换。图 5.9 展示了 3 种变换单独使用的效果。

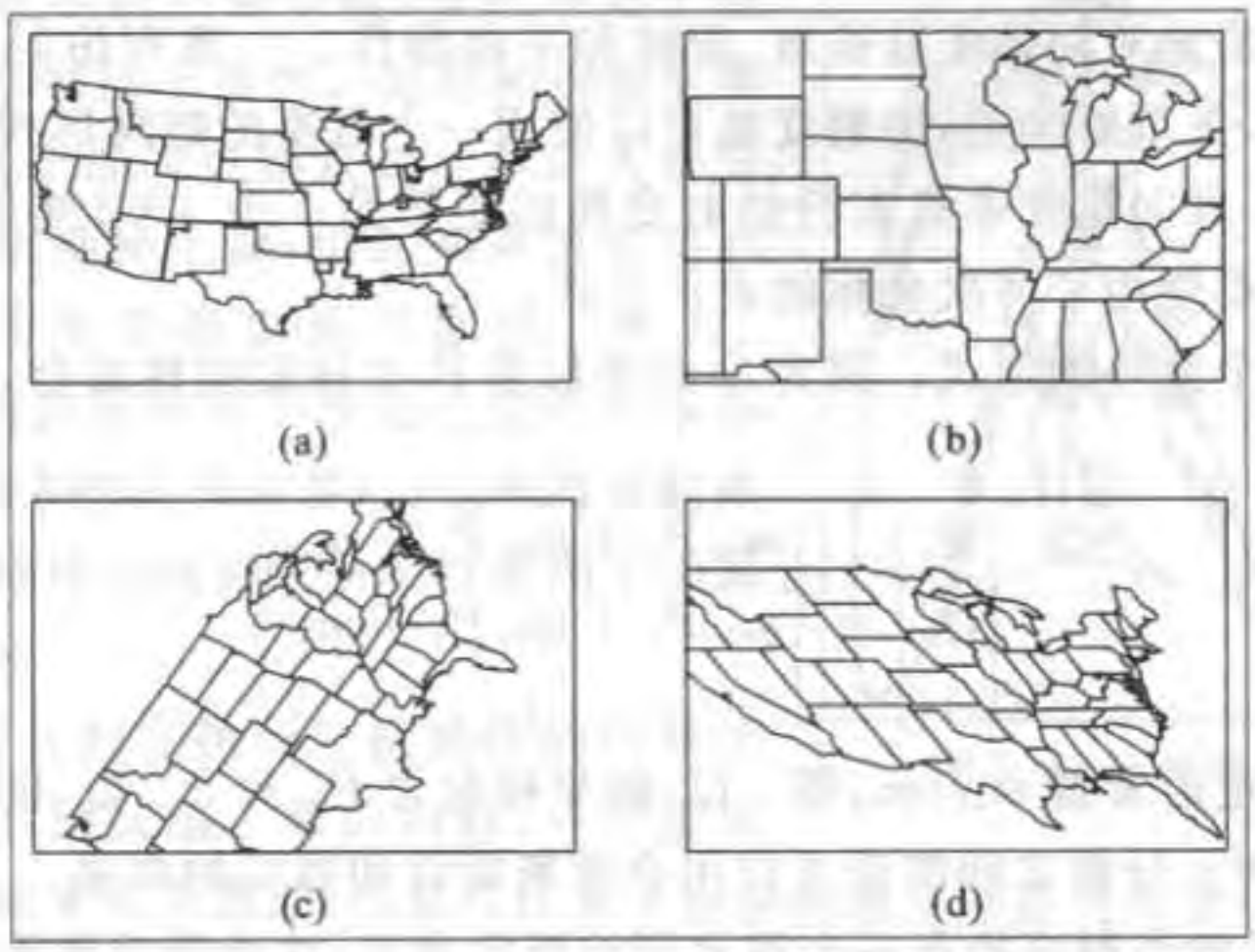


图 5.9 一幅地图的变换操作

(a) 原始图像；(b) 缩放效果；(c) 旋转效果；(d) 剪切效果

平移

当显示一张图片时，你经常会想要把图片移动到另一个位置。平移操作的一个关键性质，反映在数学上是点 P 移动的距离和 P 点预先位置无关。为了将 P 点沿着 x 方向平移距离 a，沿着 y 方向平移距离 b，我们可以使用下面的矩阵：

$$\begin{pmatrix} Q_x \\ Q_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & a \\ 0 & 1 & b \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix} \quad \text{得到：} \begin{pmatrix} P_x + a \\ P_y + b \\ 1 \end{pmatrix}$$

例如，如果 a 是 2，b 是 3，那么每一个点将会平移到原来位置向右两个单位、向上三个单位的位置。例如点(1，−5)会变换到(3，−2)，点(0，0)会变换到(2，3)。请注意，如果这个矩阵应用到一个向量(它的第三个坐标是 0)，平移将会没有任何效果。

使用齐次坐标系的进一步讨论

我们似乎没有必要去论证使用齐次坐标系的必要性，但是多少了解一些对将来的学习是有好处的。如果不使用齐次坐标，那么将没有办法使用矩阵操作进行平移。例如，如果我们使用通常的坐标表示 P 点和一个 2×2 矩阵：

$$M = \begin{pmatrix} 1 & b \\ a & 1 \end{pmatrix} \times \begin{pmatrix} p_x \\ p_y \end{pmatrix}$$

我们将会得到变换后的点：

$$\begin{pmatrix} p_x + bp_y \\ ap_x + p_y \end{pmatrix}$$

这对于任意 a 和 b 来说几乎不是一个平移，主要原因是 a 和 b 是与 p_x、p_y 是相乘的，而不是简单的相加。但是在齐次坐标系下，我们把第三项置为 1 或者 0，从而会加到点或者向量的其他项中。

缩放

缩放能够改变一个图像的大小，它包含两个缩放因子 S_x 和 S_y，分别对应 x 轴和 y 轴，表示如下：

$$(Q_x, Q_y) = (S_x P_x, S_y P_y)$$

因此，缩放的矩阵就可以简单地表示为：

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

这里的缩放更准确地说是关于原点的缩放，这是因为点 P 是从原点沿着 x 轴移动 S_x 倍，沿着 y 轴移动 S_y 倍。如果缩放因子是负数，那么就会有一个沿某一轴的反射。图 5.10 就是一个例子，将缩放因子$(S_x, S_y)=(-1, 2)$应用到一个点集，每一个点是关于 y 轴的反射，并且放大 2 倍。也有纯粹的反射，其缩放因子是+1 或者−1，一个例子就是

$$T(P_x, P_y) = (-P_x, P_y) \tag{5.6}$$

它通过用$-x$ 代替 x，使得图像沿着 y 轴垂直折叠过去(它的变化矩阵是什么呢?)。

如果两个缩放因子一样：$S_x = S_y = S$，那么此变换叫做均匀变换，或者叫做关于原点的放大，放大系数是$|S|$。如果 S 是负数，那么变换就是关于两个轴的反射。一个点从原点向外移动$|S|$倍远的距离，如果$|S|<1$，那么该点将会靠近原点，变得缩小。如果缩放因子不一样，那么变换叫做非均匀缩放。

旋转

一个基本的图形操作就是绕着某一点旋转一定角度。图 5.11 展示了一个点集绕着原点 旋转了一个角度 $\theta=60°$。

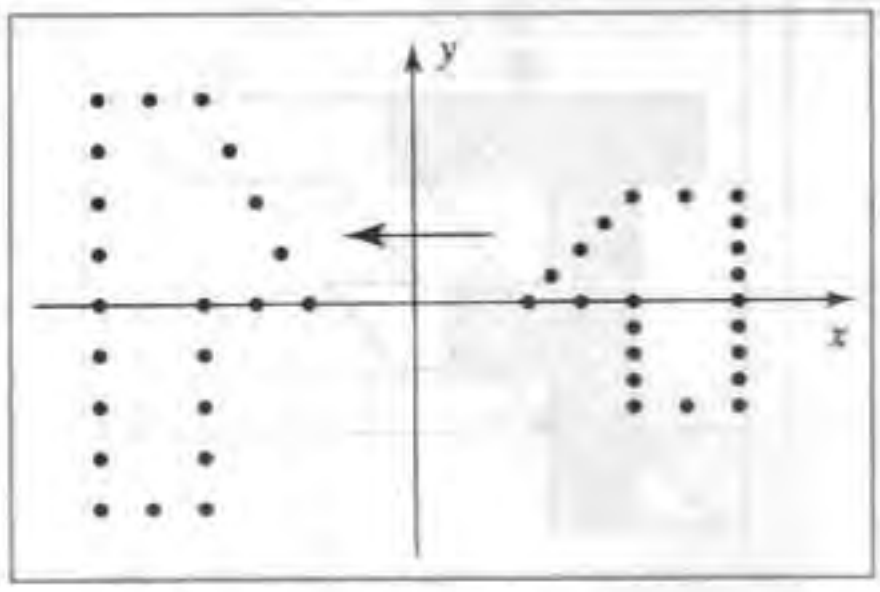


图 5.10　一个缩放和反射的组合

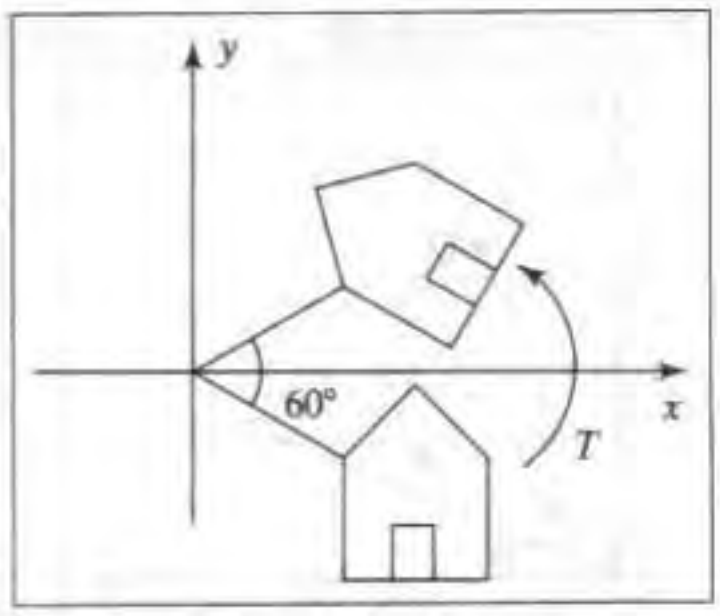


图 5.11　将一个点集旋转 60°

当 $T()$ 表示关于原点的旋转时，$Q=T(P)$ 有自己的形式，运用我们高中学的三角学知识有：

$$Q_x = P_x\cos(\theta) - P_y\sin(\theta)$$
$$Q_y = P_x\sin(\theta) - P_y\cos(\theta) \tag{5.7}$$

后面我们会得到，如果旋转角度 θ 是一个正值，那么旋转方向将是逆时针(CCW)。一个纯粹的绕原点的旋转可以用矩阵表示为：

$$\begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.8}$$

例 5.2.2

找出 P 点绕原点旋转 60°之后的点 Q。

解：对于 60°，实际上是 1.047 弧度，$\cos(\theta)=0.5$，$\sin(\theta)=0.866$。等式(5.7)变为 $Q_x=(3)(0.5)-(5)(0.866)=-2.83$，$Q_y=(3)(0.866)+(5)(0.5)=5.098$。可以使用画图工具从(3，5)旋转 60°，得到映射后的点的位置。同样可以检查 Q 和 P 到原点的距离是一样的(这个距离是多少？答案是 5.83082，和另一个是一样的)。

旋转映射等式的推导 我们想要表明等式(5.7)的正确性。图 5.12 显示了如何得到 P 点绕原点旋转角度 θ 后 Q 点的坐标。如果 P 点到原点的距离为 R，角度为 ϕ，那么 $P=(R\cos(\phi),R\sin(\phi))$。现在 Q 点和原点的距离与 P 点一样，角度为 $\theta+\phi$。使用三角函数，Q 点的坐标为

$$Q_x = R\cos(\theta+\phi)$$
$$Q_y = R\sin(\theta+\phi)$$

使用三角关系进行替换：

$$\cos(\theta+\phi) = \cos(\theta)\cos(\phi) - \sin(\theta)\sin(\phi)$$
$$\sin(\theta+\phi) = \sin(\theta)\cos(\phi) + \cos(\theta)\sin(\phi)$$

同时使用 $P_x=R\cos(\phi)$ 和 $P_y=R\sin(\phi)$ 就可以得到等式(5.7)。

剪切

一个在 x 方向上或者是沿着 x 方向的剪切操作如图 5.13 所示。在这个例子中，每一点的 y 坐标不受影响，而 x 坐标平移一个 y 坐标的线性量。一个在 x 方向上的剪切表

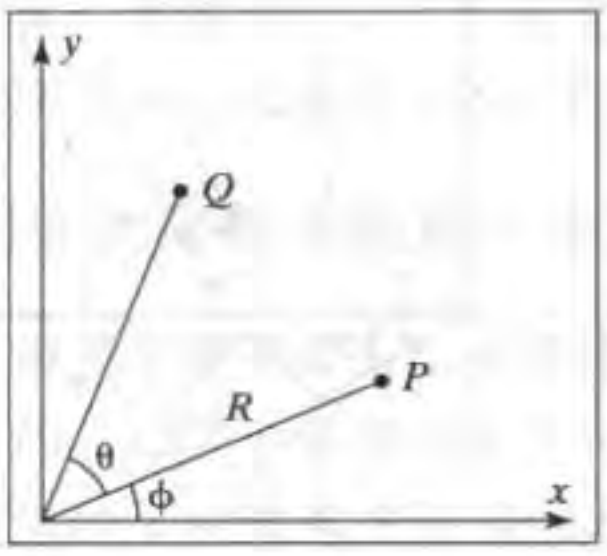


图 5.12 旋转映射的推导

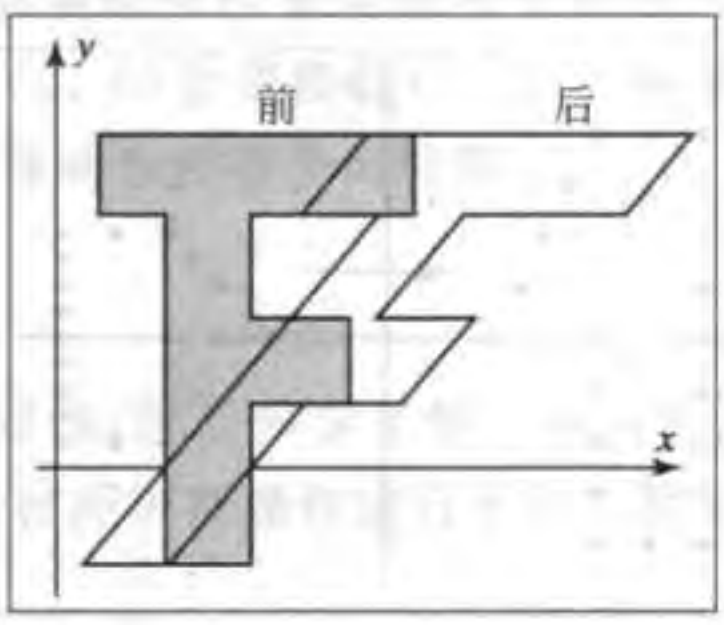


图 5.13 一个剪切的例子

示成：

$$Q_x = P_x + hP_y$$
$$Q_y = P_y$$

系数 h 表示了 P 点的 y 坐标有多少被加到了 x 坐标上。h 可正可负。裁剪有时候被用来从正规字体制作 *italic* 样式的字体。和这个剪切相关联的矩阵是：

$$\begin{pmatrix} 1 & h & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.9}$$

我们也可以在 y 轴上面进行裁剪，这种情况下对于某些 g 有 $Q_x = P_x$ 和 $Q_y = gP_x + P_y$，用矩阵表示为：

$$\begin{pmatrix} 1 & 0 & 0 \\ g & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

例 5.2.3

当等式(5.9)中 $h=0.3$ 时，点(3，4)变换到哪一点？

解：

$$Q = (3 + (0.3) \times 4, 4) = (4.2, 4)$$

例 5.2.4

当等式(5.10)中 $g=0.2$ 时，点(6，−2)映射到哪一点？

解：

$$Q = (6, 0.2 \times 6 - 2) = (6, -0.8)$$

剪切的一个值得注意的性质是它的矩阵行列式是 1。后面我们会看到，这保证了图形的面积不变。

练习

练习 5.2.1 缩放的效果

一个纯缩放的仿射变换系数为 $S_x=3$ 和 $S_y=-2$。画出图 5.14 中三个物体在此变换下的结果(使用一个事实——后面会证明——仿射变换将直线映射为直线，椭圆映射为椭圆)。

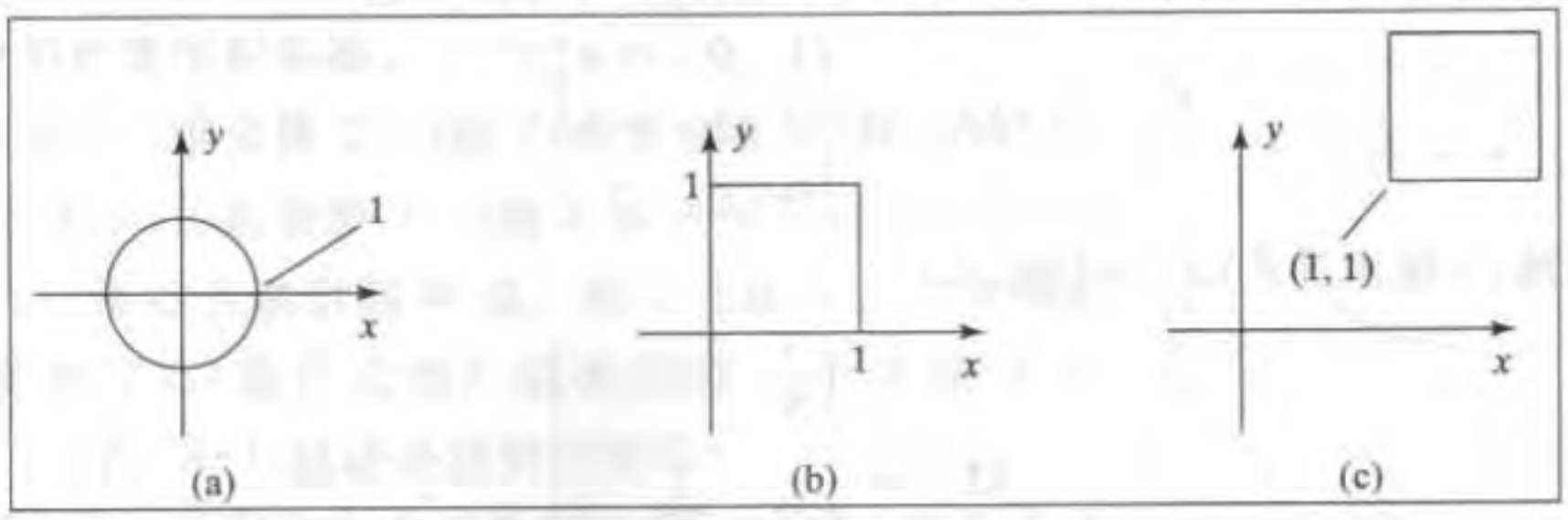


图 5.14 被缩放的物体

练习 5.2.2 旋转一个点

使用等式(5.7)求出下面各点绕原点旋转以后的结果：

a. (2, 3),旋转 $-45°$

b. (1, 1),旋转 $-180°$

c. (60, 61),旋转 $4°$

请使用图形工具检查每个结果,比较旋转以后到原点的距离和没有旋转以前到原点的距离。

练习 5.2.3 裁剪

考虑剪切变换中 $g=0.4$ 和 $h=0$。实验证明各种不同的三点共线经过裁剪以后还是三点共线。那么,假设直线裁剪以后还是直线,计算下面的线段经过裁剪以后是什么图形?

a. 水平线段(−3, 4)和(2, 4)

b. 水平线段(−3, −4)和(2, −4)

c. 垂直线段(−2, 5)和(−2, −1)

d. 垂直线段(2, 5)和(2, −1)

e. 线段(−1, −2)和(3, 2)

5.2.4 仿射变换的逆变换

你可能经常使用变换 T 对某一个点进行变换,然后你又想通过另一个变换去除之前的变换得到原来的点,这个变换就叫做**逆变换**,记作 T^{-1}。因此通过原始变换怎样简单计算 T^{-1} 就变得很有意义。

使用仿射变换就可以达到上述目的。下面的定理显示了仿射变换的一个很好的性质。

定理：仿射变换的逆变换是另一个仿射变换,它的矩阵是原仿射变换矩阵的逆矩阵。

如果一个矩阵的行列式不为 0,那么它的逆矩阵一定存在。

为了说明这一点,我们假设仿射变换 T 和矩阵 M 相关联,对点 P 实施变换 T 就可以得到点 Q。因此 $Q=T(P)$ 或者 $Q=MP$(假设我们的点是矩阵的一列,为什么?)。对 Q 使用逆变换 T^{-1} 得到 $P=T^{-1}(Q)$。或者是在 $Q=MP$ 两边简单乘以 M 的逆,写作 M^{-1},得到

$$P = M^{-1}Q \tag{5.11}$$

因此我们可以得到基本变换的逆变换：

- **平移**：如果变换是在 x 坐标增加 a,在 y 坐标增加 b,那么逆变换就是简单地减去这些值而不是加上,所以有矩阵

$$M^{-1} = \begin{bmatrix} 1 & 0 & -a \\ 0 & 1 & -b \\ 0 & 0 & 1 \end{bmatrix}$$

- **缩放**：(使用等式(5.5)中的 M)

$$M^{-1} = \begin{bmatrix} \frac{1}{S_x} & 0 & 0 \\ 0 & \frac{1}{S_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- **旋转**：(使用等式(5.7)中的 M)

$$M^{-1}=\begin{bmatrix}\cos(\theta) & \sin(\theta) & 0\\ -\sin(\theta) & \cos(\theta) & 0\\ 0 & 0 & 1\end{bmatrix}$$

注意到这和最初的设想是一样的，一个旋转 θ 度的逆变换很简单就是旋转 $-\theta$ 度。

- **剪切**：(使用等式(5.8)中的 M)

$$M^{-1}=\begin{bmatrix}1 & -h & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

练习

练习 5.2.4 剪切变换的逆变换

一个剪切变换的逆变换是否还是一个剪切变换？为什么？

练习 5.2.5 逆矩阵

计算下面矩阵的逆矩阵：

$$M=\begin{bmatrix}3 & 2 & 1\\ -1 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

5.2.5 组合一个仿射变换

我们很少只需要单一的基本变换，通常要构造一个几种基本变换的组合变换。例如我们想要：

- 平移(3，−4)
- 然后旋转 30°
- 然后缩放(2，−1)
- 然后平移(0，1.5)
- 最后旋转−30°

这些单独的变换怎样才能组合成一个整体的变换呢？这种将一系列的几个变换作用构造为一个变换的过程叫做组合(或者连接)变换。我们将会看到，两个仿射变换组合以后，得到的变换仍然是仿射变换。

考虑两个二维变换 $T_1()$和 $T_2()$组合在一起时的情形。如图 5.15 所示，仿射变换 $T_1()$将 P 点映射到 Q 点，同时仿射变换 $T_2()$将 Q 点映射到 W 点。那么直接将 P 点映射到 W 点的变换 $T()$是什么呢？它是这样一个变换 $W=T_2(Q)=T_2(T_1(P))$，这还是仿射变换吗？

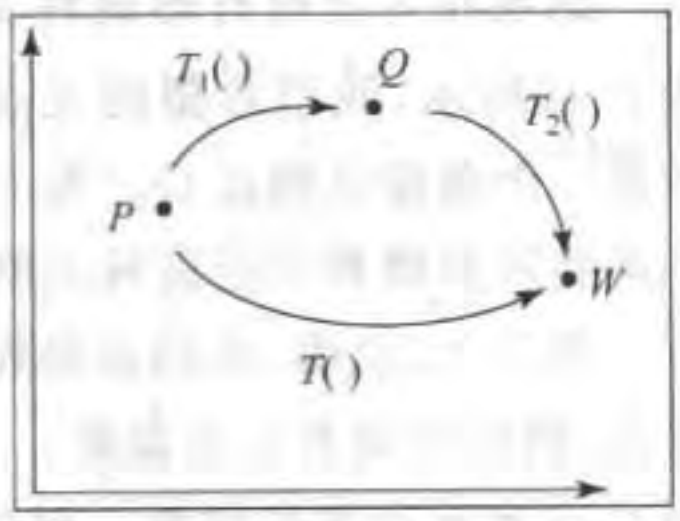


图 5.15 两个变换的组合

假设两个变换用矩阵表示分别是 $\widetilde{M}_1$ 和 $\widetilde{M}_2$，那么 $\widetilde{P}$ 首先变换到点 $\widetilde{M}_1\widetilde{P}$，然后又变换到 $\widetilde{M}_2(\widetilde{M}_1\widetilde{P})$。通过结合性可

得$(\widetilde{M}_2\widetilde{M}_1)\widetilde{P}$,因此我们有

$$\widetilde{W} = \widetilde{M}\widetilde{P} \tag{5.12}$$

其中整体变换使用一个单一的矩阵

$$\widetilde{M} = \widetilde{M}_2\widetilde{M}_1 \tag{5.13}$$

因此,两个仿射变换的组合是另一个仿射变换,它的矩阵是两个单独变换矩阵的乘积。

当使用齐次坐标时,仿射变换的组合仅仅通过一个简单的矩阵乘法就可以实现。请注意矩阵的排列顺序与变换的操作顺序相反。如果我们首先用矩阵 $\widetilde{M}_1$ 进行变换 T_1,然后用矩阵 $\widetilde{M}_2$ 进行变换 T_2,那么最终的变换的矩阵就是 $\widetilde{M}_2\widetilde{M}_1$,当你从左向右读这个结果时,第二个矩阵首先出现(当我们变换坐标系时顺序将会相反)。

同理,我们可以组合任意数量的仿射变换,这仅仅需要把它们的相关矩阵进行相乘。在这种情况下,包含旋转、缩放、剪切和平移的任意变换组合都能变成一个单一的矩阵表示。

例 5.2.5

构造一个变换能够实现:

a. 旋转 45°。

b. 然后在 x 方向上缩放 1.5 倍,在 y 方向上缩放 -2 倍。

c. 然后平移(3, 5)。

找出点(1, 2)在此变换后的像。

解:以恰当的顺序对三个矩阵进行相乘(第一个在最后,依次类推),我们得到:

$$\begin{pmatrix} 1 & 0 & 3 \\ 0 & 1 & 5 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1.5 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0.707 & -0.707 & 0 \\ 0.707 & 0.707 & 0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1.06 & -1.06 & 3 \\ -1.414 & 1.414 & 5 \\ 0 & 0 & 1 \end{pmatrix}$$

现在变换点(1, 2),先把它扩展为齐次坐标表示(1, 2, 1),然后把它和上述矩阵相乘,得到的结果是(1.94, 0.758, 1),省略最后的分量得到(1.94, 0.758)。可以使用图形计算软件来验证这一结果。

5.2.6 二维组合变换的实例

下面我们考察二维组合变换的若干重要实例,看看它们的具体性质。

例 5.2.6 绕任意一点旋转

到现在为止所有的旋转都是绕着原点的,但是我们想要绕着平面上其他点旋转。如图 5.16所示,我们希望的支点是 $V=(V_x, V_y)$,我们想要旋转 P 点一个角度 θ 到点 Q。为了实现这个旋转,我们需要把绕着 V 点旋转和绕着原点旋转关联起来。

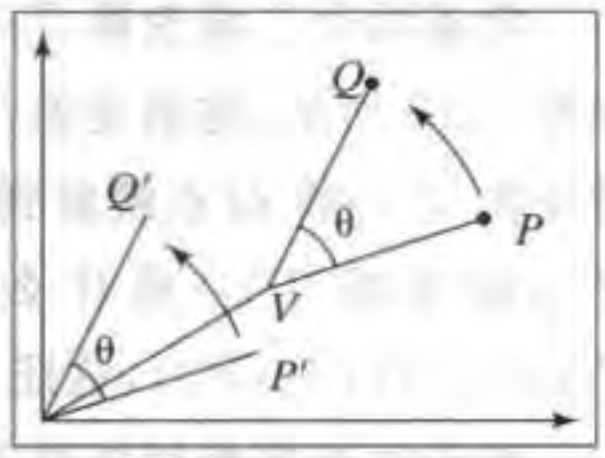


图 5.16 绕一个点旋转

图 5.16 说明,我们如果先平移所有的点使得 V 点和原点重合,然后再绕着原点旋转。完成之后,整个平面再平移回去,使 V 点回到原来的位置。因此这个旋转就包含以下 3 个基本变换:

1. 使 P 点平移 $v=(-V_x,-V_y)$。
2. 绕着原点旋转 θ。
3. 使 P 点平移回去 v。

为每一个变换创建一个矩阵，然后把它们乘起来：

$$\begin{bmatrix}1 & 0 & V_x\\0 & 1 & V_y\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}\cos(\theta) & -\sin(\theta) & 0\\\sin(\theta) & \cos(\theta) & 0\\0 & 0 & 1\end{bmatrix}\begin{bmatrix}1 & 0 & -V_x\\0 & 1 & -V_y\\0 & 0 & 1\end{bmatrix}=\begin{bmatrix}\cos(\theta) & -\sin(\theta) & d_x\\\sin(\theta) & \cos(\theta) & d_y\\0 & 0 & 1\end{bmatrix}$$

整体的平移分量 d_x 和 d_y 分别是：

$$d_x=-\cos(\theta)V_x+\sin(\theta)V_y+V_x$$
$$d_y=-\sin(\theta)V_x-\cos(\theta)V_y+V_y$$

因为 $\sin(\theta)$ 和 $\cos(\theta)$ 同时出现在结果矩阵和绕原点旋转的变换矩阵中，所以我们可知绕任意一点旋转实际上等价于先绕原点旋转然后再经过一个如上所示的距离为 (d_x,d_y) 的平移。

来看一个具体的例子，有一个变换是绕着(−2，3)旋转 30°，点(1，2)在此变换下会映射到哪一点？旋转 30°我们使用 $\cos(\theta)=0.866$、$\sin(\theta)=0.5$。平移向量是(1.232，1.402)，对任意一点 (P_x,P_y) 使用变换，结果是：

$$Q_x=0.866P_x-0.5P_y+1.232$$
$$Q_y=0.5P_x+0.866P_y+1.402$$

把这个作用应用到(1，2)上得到(1.098，3.634)。这是正确结果，可以使用图形工具验证。

例 5.2.7 关于任意支点的缩放和剪切

同样地，我们也想做关于任意点的缩放而不仅仅是原点。等式(5.5)所示的缩放操作是关于原点的，就像上述的旋转一样我们要依次进行平移-变换-逆平移等一系列操作。缩放操作和剪切操作会在后面的练习中进行探讨。

例 5.2.8 关于直线的反射

考虑一条经过原点的直线，和 x 轴成 β 角度，如图 5.17 所示。点 A 反射到点 B，每一个房子模型反射到另一个。我们想要建立一个关于这个轴可以反射任意点 P 到点 Q 的变换。这还是一个仿射变换吗？

为了说明它是一个仿射变换，我们通过三个部分建立它：

- 旋转角度 $-\beta$（因此，旋转轴可以看作是 x 轴）。
- 关于 x 轴反射。
- 旋转 β 度使轴回到原位。

这些变换每一个都可以用一个矩阵表示，因此这是一个仿射变换并且最终变换可以用三个矩阵的乘积表示。读者可以检查以下每一个变

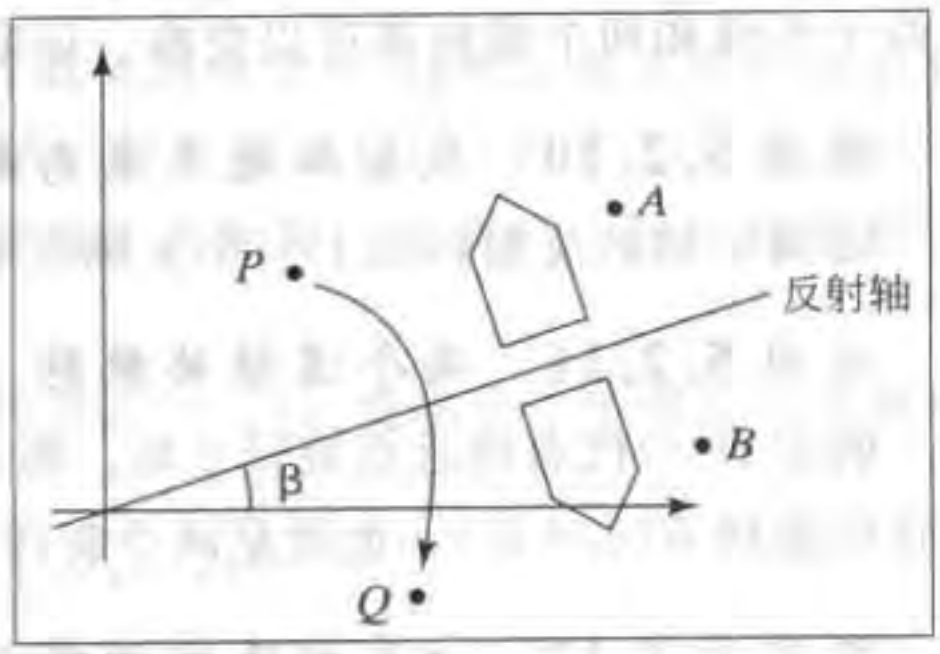


图 5.17 轴反射

换都对应下面一个矩阵,最终矩阵也是正确的:

$$\begin{pmatrix} c & s & 0 \\ -s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} c & -s & 0 \\ s & c & 0 \\ 0 & 0 & 1 \end{pmatrix}=\begin{pmatrix} c^2-s^2 & -2cs & 0 \\ -2cs & s^2-c^2 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

其中 c 代表 $\cos(\beta)$,s 代表 $\sin(\beta)$。使用三角恒等式,最终矩阵可以被写作:

$$\begin{pmatrix} \cos(2\beta) & \sin(2\beta) & 0 \\ \sin(2\beta) & -\cos(2\beta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{(关于角度为 } \beta \text{ 的轴的反射)} \tag{5.14}$$

这看起来很像一个旋转矩阵,除了角度是 2 倍,并且负号变到了第 2 行。事实上它是关于角度为 β 的轴的反射。

练习

练习 5.2.6 仿射变换的不动点

如果一个点 p 是仿射变换 $T(p)=Mp$ 的不变点,那么满足 $T(F)=F$ 或者 $FM=F$。

a. 证明当矩阵 M 的第三列是(0, 0, 1)时,就会没有平移,所以原点总是一个不动点。

b. 证明对一些向量 $\mathbf{d}$,F 必须满足 $F=\mathbf{d}(1-M)^{-1}$。是不是每一个仿射变换都有不动点呢?

c. 关于 V 点的缩放,其系数是 S_x 和 S_y,求它的不动点。

练习 5.2.7 找出矩阵

显式地给出 3×3 形式的矩阵,分别描述下面各个变换:

a. 在 x 轴方向上放大 2 倍,然后关于点(2, 1)旋转。

b. 关于(2, 3)进行缩放,然后平移(1, 1)。

c. 在 x 方向上剪切 30%,然后再在 x 方向上放大 2 倍,最后绕点(1, 1)旋转 30°。

练习 5.2.8 规则化

求一个仿射变换,可以把以(0, 0)、(2, 1)、(0, 5)和(−2, 4)组成的四边形映射成一个正方形,正方形的四个顶点分别是(0, 0)、(1, 0)、(1, 1)和(0, 1)。画出这个四边形。

练习 5.2.9 一些变换的顺序可以交换

证明均匀缩放和旋转操作可以交换,结果变换不依赖于各个单独变换作用的顺序。证明两个平移和两个缩放都可以交换。证明不均匀缩放和旋转不能交换。

练习 5.2.10 反射加起来成为旋转

证明 x 轴的反射再加上一个 y 轴反射和一个 180°的旋转相等价。

练习 5.2.11 两个连续的旋转

假设 $R(\theta)$代表绕远点旋转 θ 角。那么证明旋转 $R(\theta_1)$后面跟旋转 $R(\theta_2)$,等价于一个单独的旋转 $R(\theta_1+\theta_2)$。也就是两个旋转是可加的。

练习 5.2.12 一系列连续的剪切

找出一系列沿着 x 轴的纯剪切后面跟一系列沿着 y 轴的纯裁剪的组合变换。它还是

裁剪变换吗？可以使用绘图工具，看看一个以圆点为中心的正方形经过一个两轴同时裁剪和经过两个轴一系列裁剪后，发生了什么？

5.2.7 仿射变换的一些有用的性质

我们已经学习了如何用矩阵表示二维变换，如何将很多复杂的变换组合成一个变换，以及二维仿射变换的几何效果。在扩展到三维变换之前总结一些仿射变换的通用性质是很有意义的。这些性质是很好扩展的，因为变换的时候没有涉及到维度信息，因此在三维仿射变换情况下仍然适用。三维的仿射变换有一点我们应该知道，与二维情况相似，也可以在齐次坐标下用一个矩阵表示。

仿射变换保持点的仿射组合

我们知道，两个点 P_1 和 P_2 的仿射组合就是点 $W=a_1P_1+a_2P_2$，其中 $a_1+a_2=1$。当对点 W 应用仿射变换 $T()$时会发生什么？我们指出 $T(W)$等于各点仿射变换的组合：

$$T(a_1P_1+a_2P_2)=a_1T(P_1)+a_2T(P_2) \tag{5.15}$$

（想想这个值得注意的结果：a 就是之前等式中的值）

例如 $T(0.7(2,9)+0.3(1,6))=0.7T((2,9))+0.3T((1,6))$。这其实就是简单的线性性质。使用齐次坐标，点 $T(\widetilde{W})$就是 $\widetilde{M}\widetilde{W}$，使用矩阵乘法的线性性质，我们有：

$$\widetilde{M}\widetilde{W}=\widetilde{M}(a_1\widetilde{P}_1+a_2\widetilde{P}_2)=a_1\widetilde{M}\widetilde{P}_1+a_2\widetilde{M}\widetilde{P}_2$$

用普通坐标表示就是 $a_1T(P_1)+a_2T(P_2)$，如前述的那样。仿射变换保持点的仿射组合的性质是相当基本和抽象的，但是它被证明是很关键的，甚至和仿射变换的定义一样重要。

仿射变换保持共线和共面关系

仿射变换的一个重要性质就是保持共线性和共面性，因此直线的像是另一条直线。为了证明这一点，我们回忆一下通过 A 和 B 两点的直线的参数表示 $L(t)$，它是 A 和 B 的一个仿射组合：

$$L(t)=(1-t)A+tB$$

这是点的一个仿射组合，从前面我们可以知道，$L(t)$的像是 A 和 B 的像的仿射组合：

$$Q(t)=(1-t)T(A)+tT(B) \tag{5.16}$$

这是另外一条通过 $T(A)$和 $T(B)$的直线。在计算机图形学中，这可以极大地减轻绘制经过变换的线段的工作量：我们只需要计算两个端点的变换结果 $T(A)$和 $T(B)$，然后绘出它们之间的线段就可以了！这样就避免了去变换线段上面的每一个点——这样做显然很荒唐。

同理，一个平面也被转换成另外一个平面。回忆等式(4.45)，表示一个平面可以写成点的仿射组合：

$$P(s,t)=sA+tB+(1-s-t)C$$

当每一个点都做变换后，等式变成了：

$$T(P(s,t))=sT(A)+tT(B)+(1-s-t)T(C)$$

这就是变换后平面的参数表示。

共线性和共面性的保持保证了多边形经过变换仍然是多边形,平面多边形(它们的顶点都在一个平面内)也会变换到另一个平面多边形。特别地,三角形变换的结果也是三角形。另一个重要性质在后面也会提到,就是保持内在性:也就是说如果 A 点在 B 和 C 之间,那么点 $T(A)$ 也在 $T(B)$ 和 $T(C)$ 之间。

保持直线和平面的平行性

如果两条直线或者两个平面是平行的,那么它们在仿射变换下的像也是平行的。这个很容易证明。我们首先证明直线情况。取一条方向为 $\mathbf{b}$ 的直线 $A+\mathbf{b}t$。在齐次坐标下它将变换到 $\widetilde{M}(\widetilde{A}+\widetilde{\mathbf{b}}t)=\widetilde{M}\widetilde{A}+(\widetilde{M}\widetilde{\mathbf{b}})t$,它的方向是 $\widetilde{M}\widetilde{b}$。这个新的方向和 A 没有关系。因此两个不同的但有相同方向的直线 $A_1+\mathbf{b}t$ 和 $A_2+\mathbf{b}t$ 经过变换以后同样具有一个方向 $\widetilde{M}\widetilde{\mathbf{b}}$,因此它们还是平行的。一个重要例证就是平行四边形经过变换仍然是平行四边形。

同理,对于平面也适用:它的方向向量(参考等式(4.43))会变换到一个新的不依赖于平面位置的方向。一个重要的例证是平行六面体变换到另一个平行六面体。

例 5.2.9 网格是怎么变换的呢?

因为仿射变换把平行四边形依然映射为平行四边形,所以这就限制住了仿射变换能够在多大程度上改变物体的形状。为了说明这一点,我们将二维仿射变换 T 应用到单位垂直网格线上,如图 5.18 所示。因为网格线是由两组平行线组成的,因此经过 T 变换后单位垂直网格线映射到另一个网格线,这个网格线由另一组新的平行线组成。考虑定义在网格中的物体,想想它们是怎么扭曲变形的。仿射变换可以做到:以扭曲网格线的方式去扭曲图形。新的网格线可以扭曲成任何角度,也可以伸缩为任意距离,变换后的轴也不一定垂直。当然整个网格也可能被安放在平面上任意位置。

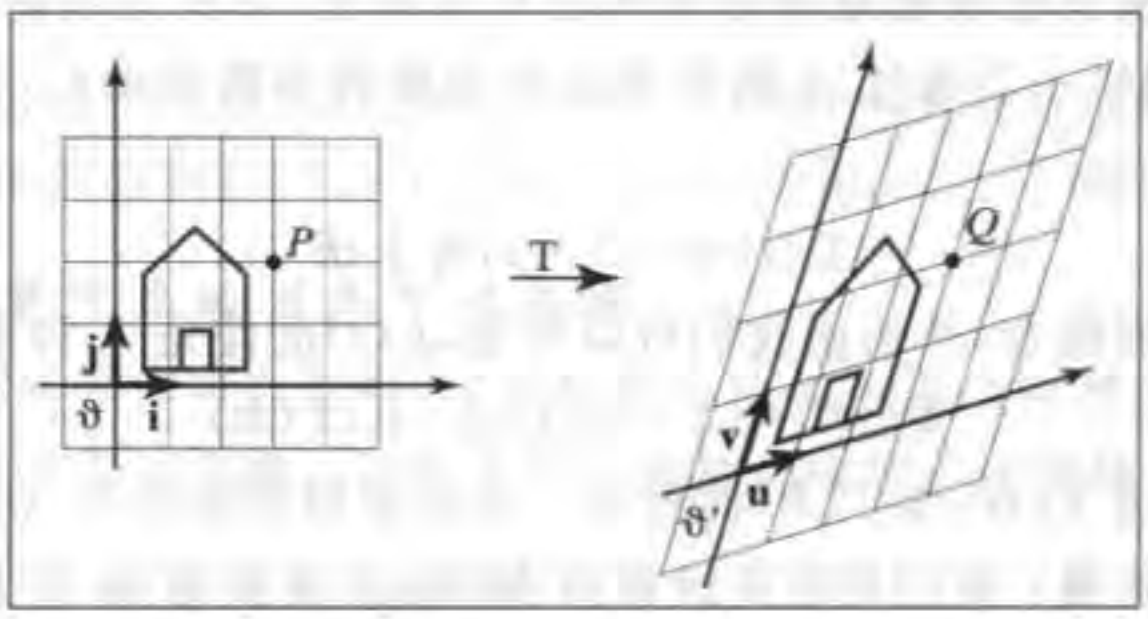


图 5.18 网格变换

三维中有同样的结果:三维仿射变换将立方体网格映射为一个平行六面体网格。

矩阵的列揭示出了变换后的坐标系

检验变换矩阵的列是很有意义的,因为它们表示了坐标系是怎么被变换的。假设矩阵已经给定:

$$M=\begin{pmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ 0 & 0 & 1 \end{pmatrix}=(\mathbf{m}_1 \vdots \mathbf{m}_2 \vdots m_3) \tag{5.17}$$

因此，它的列是 $\mathbf{m}_1$、$\mathbf{m}_2$ 和 m_3。前两列都是向量（第三个元素是 0），最后一列是一个点（它的第三个元素是 1）。一般来说，我们的坐标系定义是由原点 ϕ，和两个基本向量 $\mathbf{i}$ 和 $\mathbf{j}$：

$$\phi=\begin{pmatrix}0\\0\\1\end{pmatrix},\quad \mathbf{i}=\begin{pmatrix}1\\0\\0\end{pmatrix},\quad \mathbf{j}=\begin{pmatrix}0\\1\\0\end{pmatrix}$$

注意，向量 $\mathbf{i}$ 变换到了向量 $\mathbf{m}_1$（检验之）：

$$\mathbf{m}_1=M\mathbf{i}$$

同理，$\mathbf{j}$ 变换到了 $\mathbf{m}_2$，同时 ϕ 映射到了点 m_3。图 5.19(a)说明这一点。坐标系从（$\mathbf{i}$，$\mathbf{j}$，ϕ）变换到了坐标系（$\mathbf{m}_1$，$\mathbf{m}_2$，m_3），这正好是矩阵的列。

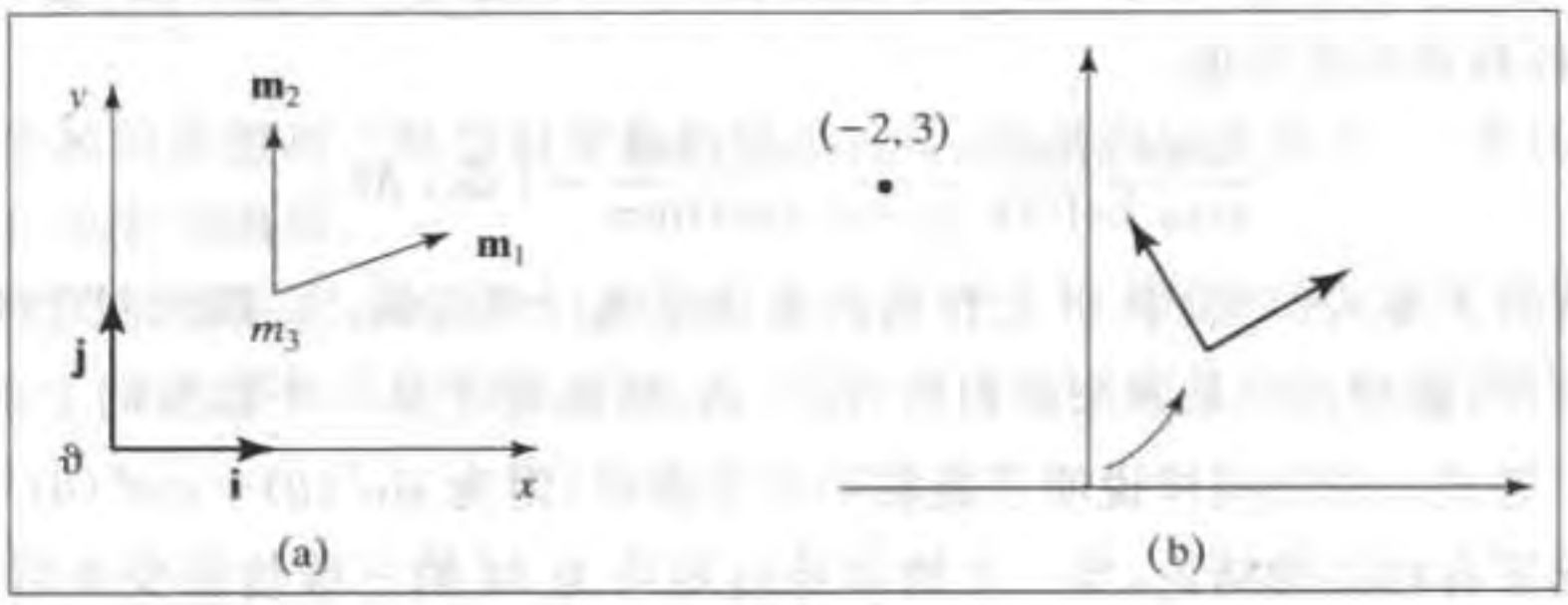


图 5.19　坐标变换

新的坐标系的轴不一定是垂直的，也不一定是单元长度。（如果变换只有旋转和缩放，那么它们仍然垂直）。任意点 $P=P_x\mathbf{i}+P_y\mathbf{j}+\phi$ 变换到 $Q=P_x\mathbf{m}_1+P_y\mathbf{m}_2+m_3$。有时候以这种方式看待一个变换矩阵是很有启发作用的。

例 5.2.10　关于点的旋转

例 5.2.5 中的变换是关于点（−2，3）旋转 30°，得到矩阵：

$$\begin{pmatrix} 0.866 & -0.5 & 1.232 \\ 0.5 & 0.866 & 1.402 \\ 0 & 0 & 1 \end{pmatrix}$$

如图 5.19(b)所示，坐标系因此变换到了一个新的坐标系，它的原点是（1.232，1.402，1），坐标轴由向量（0.866，0.5，0）和（−0.5，0.866，0）定义。注意这些坐标轴仍然是垂直的，因为只使用了旋转变换。

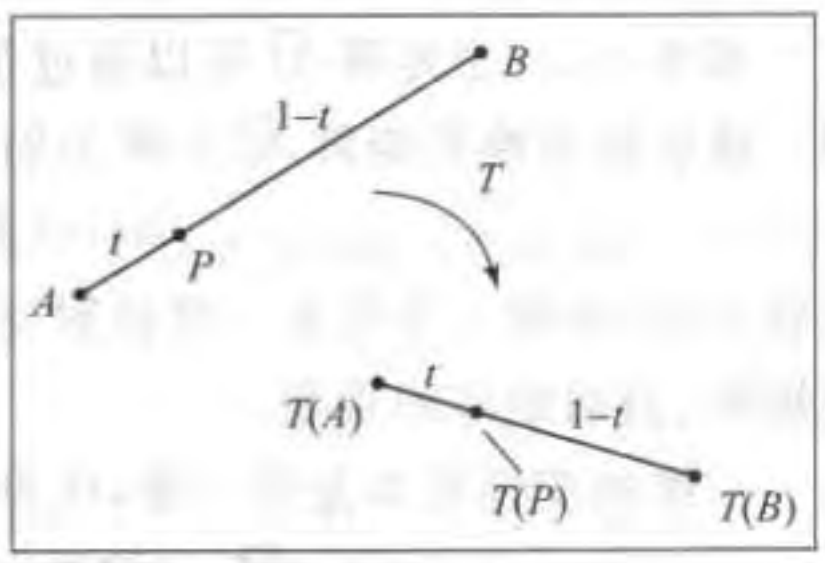


图 5.20　等比例变换

保持相互比例

仿射变换还有另一个有用的性质。考虑点 P 位于点 A 和 B 之间的 t 比例处，如图 5.20 所示。对三个点进行仿射变换 $T()$。我们可以指出经过变换的

点 $T(P)$ 仍然以比例 t 位于 A 和 B 的像 $T(A)$ 和 $T(B)$ 之间。这个不难说明(参见练习)。

一个特殊的例子,线段的中点仍然映射到线段的中点。这个引出了一个漂亮的几何结果:任何平行四边形的对角线均分这个四边形。(证明:任何平行四边形都是正方形的仿射变换,而正方形的对角线均分它,那么平行四边形的对角线也均分自己。)三维情况也一样:任何平行六面体的对角面也均分它。

除了保持线性、平行性和相关比例以外,仿射变换还保持椭圆和椭球。见后面第 8 章。

仿射变换怎样影响一个图形的面积

在 CAD 应用中,计算一个物体的面积和体积是很重要的。例如,当一个多边形的所有定点经过仿射变换后它的面积受到了什么影响?很明显平移和旋转不会影响到图像的面积,但是缩放肯定会的,剪切也可能会。

结论很简单,练习中会得到:当二维变换矩阵 M 作用到一个物体后,它的面积就会乘以矩阵 M 的行列式的绝对值:

$$\frac{\text{area after transformation}}{\text{area before transformation}} = |\det M| \tag{5.18}$$

在二维情况下等式(5.6)中 M 的行列式是 $|m_{11}m_{22}-m_{12}m_{21}|$。因此对于等式(5.10)所示的纯缩放变换,新的面积是原来面积的 S_xS_y 倍,然而对于某一个轴方向上的裁剪来说面积没有变化。等式(5.21)同样说明了旋转不改变面积,因为 $\sin^2(\theta)+\cos^2(\theta)=1$。

三维情况下有相似的结论,当一个物体经过矩阵为 M 的三维仿射变换后,体积也就乘以了 $|\det M|$。

例 5.2.11 椭圆的面积

正好在宽为 W 高为 H 的四边形中的椭圆的面积是多少?

解:这个椭圆可以通过下面的方法获得:将一个单位圆 $x^2+y^2=1$,然后缩放因子为:$S_x=W$ 和 $S_y=H$,这个变换的矩阵是 M,它的行列式是 WH。单位圆的面积是 π,那么椭圆的面积就是 πWH。

每一个仿射变换都由基本仿射变换组成

我们可以通过组合一系列基本仿射变换来获得一个复杂的仿射变换。那么反过来问,一个仿射变换都由哪些基本仿射变换组成呢?

基本上,一个矩阵 $\widetilde{M}$ 可以通过很多方式分解成为一些基本矩阵的乘积。特别地,可以用二维仿射变换把矩阵 $\widetilde{M}$ 分解为如下结果:

$$\widetilde{M}=(\text{剪切})(\text{缩放})(\text{旋转})(\text{平移})$$

也就是说,任何一个代表二维仿射变换的 3×3 矩阵 $\widetilde{M}$ 都可以写成平移矩阵、旋转矩阵、缩放矩阵、剪切矩阵的乘积。

三维的变换更加复杂一些,代表三维变换的 4×4 矩阵可以写作:

$$\widetilde{M}=(\text{缩放})(\text{旋转})(\text{剪切 1})(\text{剪切 2})(\text{平移})$$

即平移矩阵、剪切矩阵、另一个裁剪矩阵、旋转矩阵和缩放矩阵的乘积(从右向左的顺序)。

练习

练习 5.2.13　参数一般化

如果 W 是 N 个点 $P_i, i=1,\cdots,N$ 的仿射组合，$T()$ 是一个仿射变换，那么 $T(W)$ 也是 N 个点 $T(P_i), i=1,\cdots,N$ 的仿射组合。

练习 5.2.14　证明比例关系是保持的

考虑 P 由 $A+\mathbf{b}t$ 定义，其中 $\mathbf{b}=B-A$。求出 P 到 A 和 B 的距离 $|P-A|$ 和 $|P-B|$，特别地，证明它们的比例是 t 和 $1-t$。那么如果 t 不在 0 和 1 之间呢？同样求出经过仿射变换 T 以后的距离 $|T(P)-T(A)|$ 和 $|T(P)-T(B)|$。

5.3　三维仿射变换

三维仿射变换的思想和二维仿射变换的情形一样，但是表达更复杂了，并且对三维仿射变换进行可视化也更加困难。

我们再一次使用坐标系，假设我们有原点 ϕ 和三个互相垂直的轴，方向分别是 $\mathbf{i}$、$\mathbf{j}$ 和 $\mathbf{k}$（如图 5.8 所示）。在此坐标系中的点 P 定义为 $P=\phi+P_x\mathbf{i}+P_y\mathbf{j}+P_z\mathbf{k}$，因此我们有如下的表达形式：

$$\widetilde{P}=\begin{bmatrix}P_x\\P_y\\P_z\\1\end{bmatrix}$$

假设 $T()$ 是一个仿射变换，把点 $\widetilde{P}$ 变换到 $\widetilde{Q}$。那么与二维情况类似，变换 $T()$ 可以用一个 4×4 的矩阵表示：

$$\widetilde{M}=\begin{bmatrix}m_{11}&m_{12}&m_{13}&m_{14}\\m_{21}&m_{22}&m_{23}&m_{24}\\m_{31}&m_{32}&m_{33}&m_{34}\\0&0&0&1\end{bmatrix}\tag{5.19}$$

那么，我们就可以说点 $\widetilde{Q}$ 是点 $\widetilde{P}$ 和矩阵 $\widetilde{M}$ 相乘的结果：

$$\begin{bmatrix}Q_x\\Q_y\\Q_z\\1\end{bmatrix}=\widetilde{M}\begin{bmatrix}P_x\\P_y\\P_z\\1\end{bmatrix}\tag{5.20}$$

请再次注意，仿射变换矩阵的第四行是一串 0 外加一个 1 组成的（这一点在第 7 章检验投影矩阵时将不适用）。

5.3.1　基本三维变换

我们先单独考虑基本三维变换的特性，然后把它们综合到一般三维仿射变换中去。

平移

对于一个纯平移变换,矩阵 $\widetilde{M}$ 有下面的简单形式:

$$\begin{bmatrix} 1 & 0 & 0 & m_{14} \\ 0 & 1 & 0 & m_{24} \\ 0 & 0 & 1 & m_{34} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

请验证 $\widetilde{Q}=\widetilde{M}\widetilde{P}$ 是简单地平移到 $\widetilde{Q}$,其中的平移向量 $\mathbf{m}=(m_{14},m_{24},m_{34})$。

缩放

三维缩放是二维缩放的直接扩展,有下面的矩阵:

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.21}$$

3 个常量 S_x、S_y 和 S_z 决定了相应坐标的缩放。缩放和二维一样是关于原点的。图 5.21展示了在 z 轴缩放 0.5 和在 x 轴缩放 2 的效果。

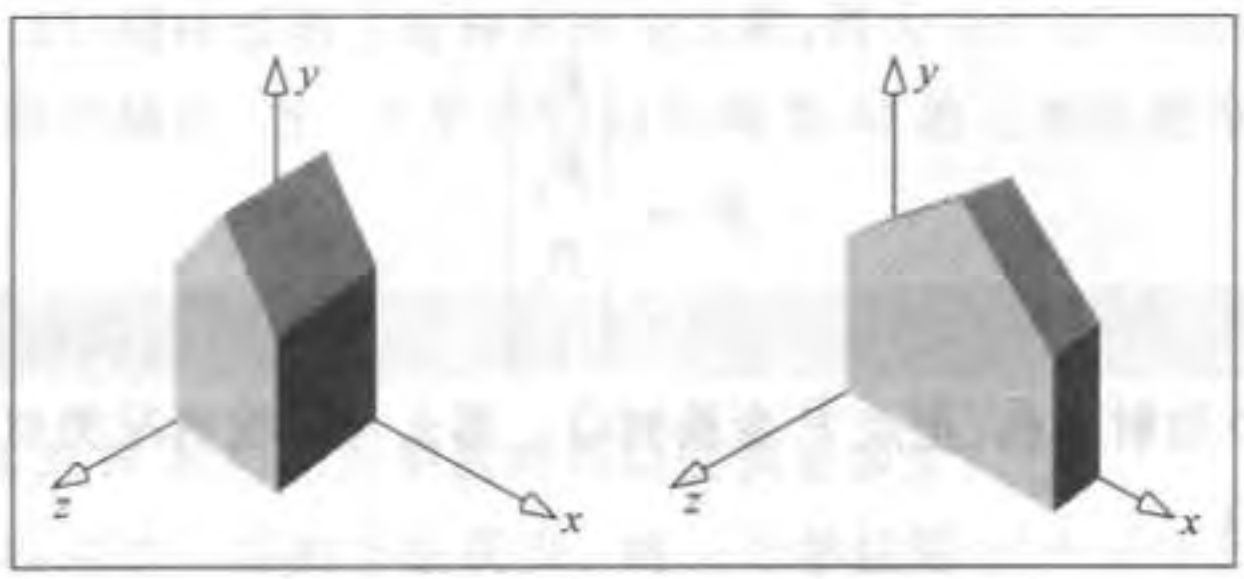


图 5.21 缩放

剪切

三维剪切和二维比较起来相差就很大了。最基本的剪切变换的矩阵是单位阵中某一个 0 被另一个值替换,如:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ f & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.22}$$

生成的 $\widetilde{Q}=(P_x, fP_x+P_y, P_z, 1)$;也就是说 P_y 加上 P_x 的某个倍数,而其他分量不变。

旋转

三维旋转在图形学中很常见,因为我们总是要通过旋转一个物体或者是镜头得到一个新的视图。和二维旋转比起来三维旋转有很大的变化,因为我们必须确定一个旋转所绕的

轴，而不是一个点。一种近似的方法是将旋转分解成一组简单的旋转的组合。

关于某一坐标轴的基本旋转　最简单的旋转是绕着某一个坐标轴的旋转。我们把绕 x 轴的旋转叫做 x-roll，把绕 y 轴的旋转叫做 y-roll，绕 z 轴的旋转叫做 z-roll。我们将逐个表达出 x-roll、y-roll和 z-roll 所对应的矩阵。对于每种给定坐标轴，都旋转一个角度 β。关于正角度的定义我们都是用向里看的规则：

β 的正值表示当逆着轴的方向向里望去时方向是逆时针(CCW)方向。

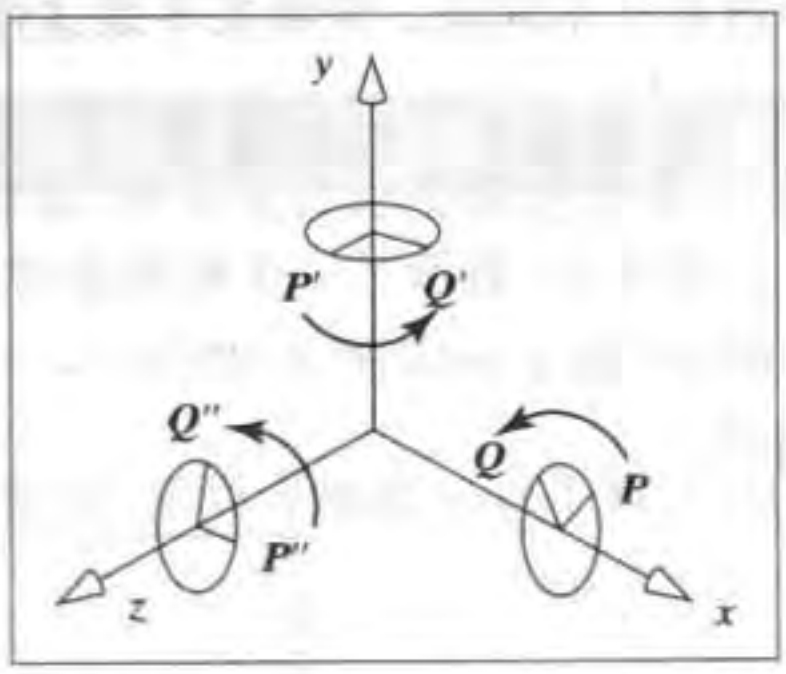


图 5.22　围绕 3 个轴的位置旋转

图 5.22 显示了 3 种基本的正向旋转。

这个等式也符合我们的二维旋转情形：二维空间的正向旋转等价于当我们在 z 轴的正半轴上某点看 xy 平面时的 z-roll 旋转。

特别的，请注意在上述规定下当旋转 90°时会发生什么情况：

- 对于 z-roll，x 轴转到了 y 轴。
- 对于 x-roll，y 轴转到了 z 轴。
- 对于 y-roll，z 轴转到了 x 轴。

下面 3 个矩阵表示了一个点绕某一轴旋转 β 度的变换。我们使用如下有提示作用的记法 $R_x()$、$R_y()$和 $R_z()$来表示 x-roll、y-roll 和 z-roll。参数就是点旋转的角度，以弧度为单位，c 代表了 $\cos(\beta)$，s 代表了 $\sin(\beta)$。

1. 一个 x-roll：

$$R_x(\beta)=\begin{bmatrix}1&0&0&0\\0&c&-s&0\\0&s&c&0\\0&0&0&1\end{bmatrix}\tag{5.23}$$

2. 一个 y-roll：

$$R_y(\beta)=\begin{bmatrix}c&0&s&0\\0&1&0&0\\-s&0&c&0\\0&0&0&1\end{bmatrix}\tag{5.24}$$

3. 一个 z-roll：

$$R_z(\beta)=\begin{bmatrix}c&-s&0&0\\s&c&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\tag{5.25}$$

注意每个矩阵中有 12 项是单位阵中的 0 和 1。它们位于与旋转坐标轴相关联的行和列中(比如：x-roll 中的第一行和第一列)。这保证了经过变换的点的对应坐标分量不会发生变化。c 和 s 总是出现在矩阵的其他行和列上。

另外，为什么 y-roll 与其他两个不太相同呢？请注意在前面的基本变换矩阵中 c 和 s 或

者 c 和 $-s$ 所出现的行。对于 x-roll 和 z-roll 来说,包含 $-s$ 的行出现在包含 s 的行上面,对于 y-roll 来说,包含 s 的行出现在包含 $-s$ 的行上面。这似乎表明了 y-roll 与另两个存在某种内在的不同点。但事实不是这样子的,后面的练习中我们会进行说明。

例 5.3.1 旋转房子

图 5.23 显示了一个在原点的房子(图 5.23(a)),经过一个 $-70°$ 的 x-roll(图 5.23(b)),一个 30°的 y-roll(图 5.23(c)),一个 $-9°$ 的 z-roll,每个变换单独进行。请仔细查看这个例子。

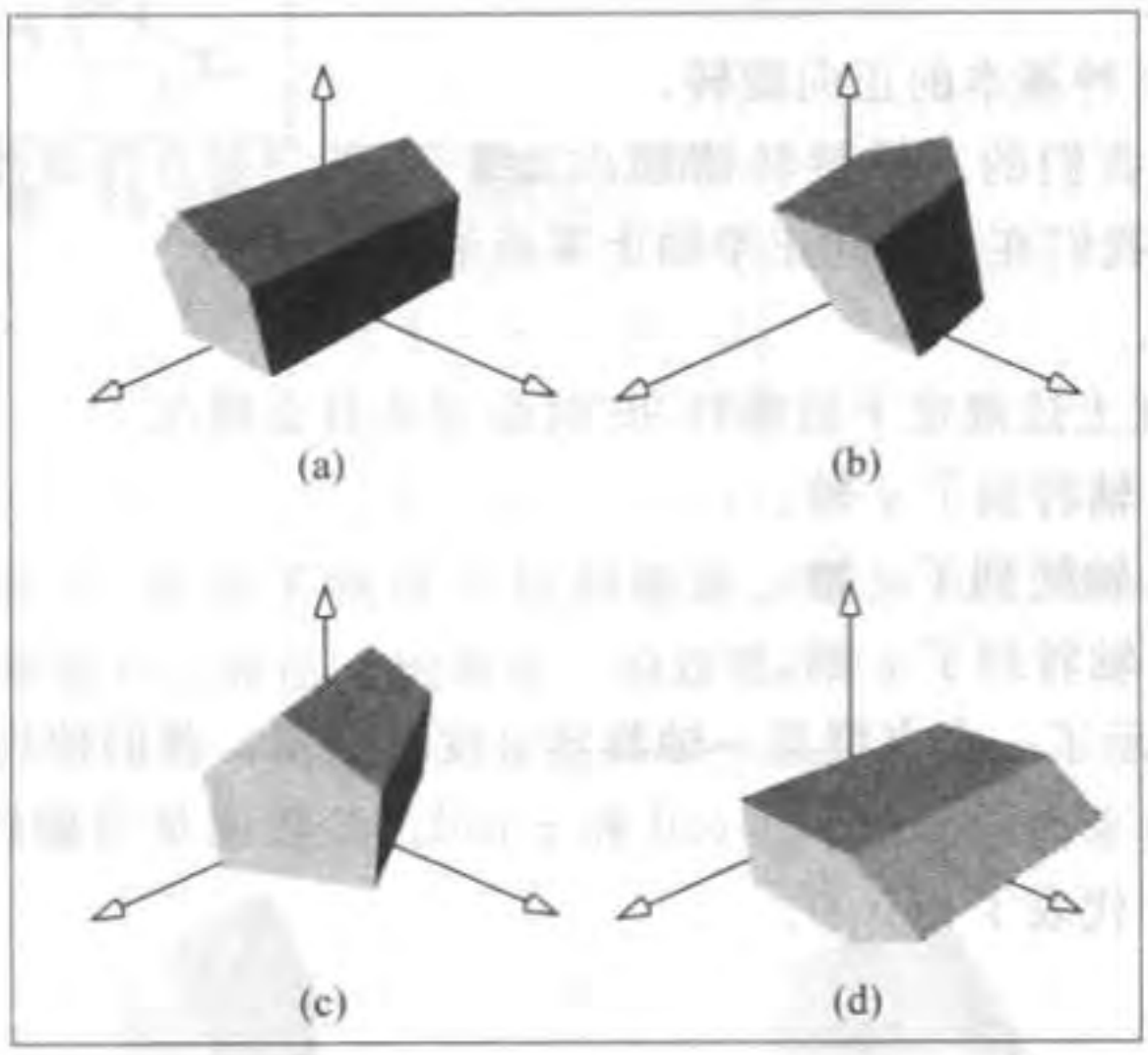


图 5.23 房子旋转

(a) 房子;(b) $-70°$ x-roll;(c) 30° y-roll;(d) $-90°$ z-roll

例 5.3.2

求 $P=(3, 1, 4)$ 经过 y-roll 旋转 30°后得到的点 Q。

解:使用等式(5.24):

$$c = 0.866$$

$$s = 0.5$$

P 变换到:

$$Q = \begin{pmatrix} c & 0 & s & 0 \\ 0 & 1 & 0 & 0 \\ -s & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 3 \\ 1 \\ 4 \\ 1 \end{pmatrix} = \begin{pmatrix} 4.6 \\ 1 \\ 1.964 \\ 1 \end{pmatrix}$$

和期望的一样,y 坐标没有变。

练习

练习 5.3.1　对 90°旋转的情形进行可视化

绘制一个右手三维系统，请充分理解如下规律：绕一个坐标轴旋转 90°（向原点方向看是 CCW）就是将另一个坐标轴旋转到第三个坐标轴的位置，就像我们之前讲述的那样。那么 x 轴上面的点绕 x 轴旋转将会是什么效果？

练习 5.3.2　旋转简单的房子模型

请画出对每一顶点实施一个 45°的 x-roll 后的房子模型，然后再画出 y-roll 和 z-roll 的情形。

练习 5.3.3　进行旋转

找出 $P=(1, 2, -1)$经过一个 45°的 y-roll 后的像 Q。在三维坐标中画出 P 和 Q，并解释你的结果的合理性。

练习 5.3.4　测试坐标轴的 90°旋转

本练习提供了一些记住变换矩阵的小窍门。将 3 个旋转矩阵作用到每个单位位置向量 **i**、**j** 和 **k**，旋转 90°。讨论每种情况的结果。

练习 5.3.5　一个 y-roll 确实不一样吗？

等式(5.24)中的负号似乎放错了位置，好像应该是在下面一行而不是上面一行。但事实上等式(5.23)～等式(5.25)是一致的。这仅仅是排序上的问题。考虑 3 个坐标轴 x、y 和 z 循环出现 $x\to y\to z\to x\to y\cdots$。如果我们正在讨论的旋转是围绕着当前某一坐标轴（$x$、$y$ 或 z），那么我们可以指出它之前和之后的轴。比如 x 是当前轴，那么 z 是之前的轴，y 是之后的轴。在这种命名规则下，3 种旋转使用相同的等式：$Q_{curr}=P_{curr}$，$Q_{next}=cP_{next}-sP_{prev}$ 和 $Q_{prev}=sP_{next}+cP_{prev}$。对可能的 3 种当前轴写出上述等式。

5.3.2　组合一个三维仿射变换

很明显，三维仿射变换可以进行组合，其结果也是另一个仿射变换。这个想法和等式(5.13)中二维情况是一致的：在变换的过程中先进行 M_1 操作再进行 M_2 操作，那么最后的整体变换矩阵是两个单独变换的乘积，M_2 乘以 M_1：

$$M = M_2 M_1 \tag{5.26}$$

任意数量的仿射变换都可以通过这种方式组合，最终结果矩阵表示整个变换。

图 5.24 显示了一个结果，一个房子首先经过变换 M_1，然后经过变换 M_2。最终结果和变换 M_2M_1 的结果一样。

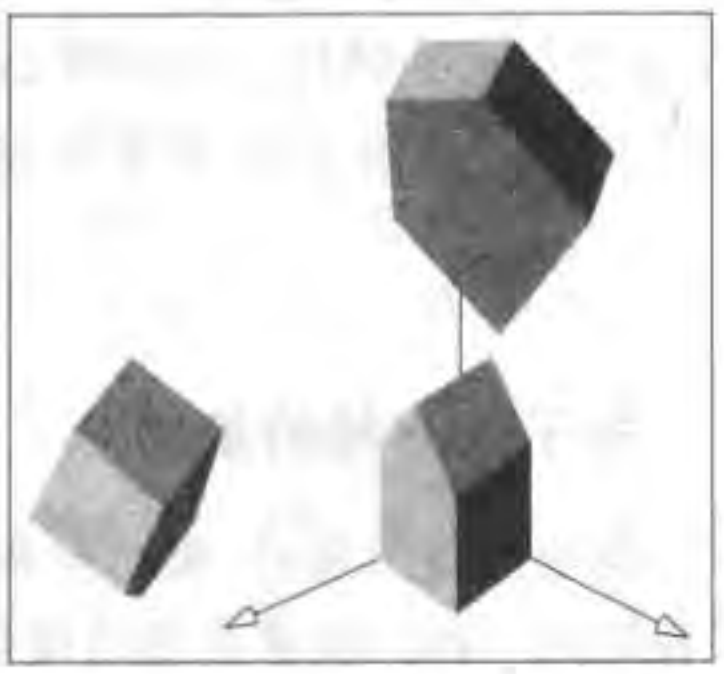


图 5.24　三维仿射变换组合

5.3.3 旋转的组合

三维变换和二维变换一个重要的区别就是旋转变换的组合方式不相同。二维情况下两个旋转变换 $R(\beta_1)$ 和 $R(\beta_2)$ 组合成变换 $R(\beta_1+\beta_2)$，它们组合的顺序是不影响结果的。在三维情况下由于旋转可以绕着不同的轴，情况就复杂得多。对于两个绕着不同轴的旋转变换，它们的组合顺序对结果影响很大：也就是说三维旋转矩阵不能交换。我们在这里将要讨论三维旋转的一些性质，了解一个旋转可以表示成的各种不同形式，然后看看怎样创建一个旋转变换用于完成某项特定任务。

在实际应用中经常要把 3 种基本的旋转变换组合成一个旋转变换：一个 x-roll 接着一个 y-roll，最后再加上一个 z-roll，每一个都是关于某个坐标轴的旋转。每一个单独的变换有等式(5.23)～(5.25)，所以整个变换就是：

$$M = R_z(\beta_z)R_y(\beta_y)R_x(\beta_x) \tag{5.27}$$

这里的角度 β_x、β_y 和 β_z 统称为欧拉角。我们将会看到任何绕一个轴(它肯定过原点)的三维旋转都可以通过选择三个合适的欧拉角的变换矩阵相乘得到，我们也会讲到一个构造性的方法用来构造所需的矩阵。上述表明需要 3 个(且仅仅 3 个)参数就可以完全确定一个旋转变换。

有人使用别的变换顺序去得到一个复杂的旋转。例如：他们或许把旋转表示为 $R_x(\beta_x)R_z(\beta_z)R_y(\beta_y)$：首先一个 y-roll，然后一个 z-roll，最后一个 x-roll。因为三维旋转不能交换顺序，所以必须选择不同的角度 β_x、β_y 和 β_z 才能得到相同的旋转。变换顺序不同，所使用的欧拉角也不相同。

例 5.3.3

求和下面变换相关的矩阵，该变换首先经过 45°的 x-roll，然后是 30°的 y-roll，最后跟一个 60°的 z-roll。直接将三个矩阵相乘可以得到：

$$\begin{pmatrix} 0.5 & 0.866 & 0 & 0 \\ 0.866 & 0.5 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 0.866 & 0 & 0.5 & 0 \\ 0 & 1 & 0 & 0 \\ -0.5 & 0 & 0.866 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.707 & -0.707 & 0 \\ 0 & 0.707 & 0.707 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$=\begin{pmatrix} 0.433 & -0.436 & 0.789 & 0 \\ 0.75 & 0.66 & -0.047 & 0 \\ -0.5 & 0.612 & 0.621 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

关于任意轴的旋转

当使用欧拉角时，我们要指定绕坐标轴旋转即 x-roll、y-roll 和 z-roll 的某个顺序。如果我们用一种方法来构造点绕任意方向的轴的旋转，那么我们就可以更加容易地使用旋转操作。想象一下，地球仪或其他玩具绕着某个倾斜的轴旋转的样子。实际上，欧拉理论表明任意的旋转都有如下形式：

欧拉定理：关于某个点的任何旋转(或者是一系列旋转)等价于绕着通过此点的某个轴的单一旋转。

这样的旋转的矩阵是什么呢？我们可以方便使用它吗？

在图 5.25 中，有一个用向量 **u** 表示的轴，以及一个点 P，它将绕着 **u** 旋转 β 角度得到点 Q。

因为 **u** 可以是任意方向，所以初看上去很难找到一个矩阵用来表示这样的一个变换。但事实上可以用两种方式找到这样的矩阵，一种是经典的方式，一种是构造性的方式。

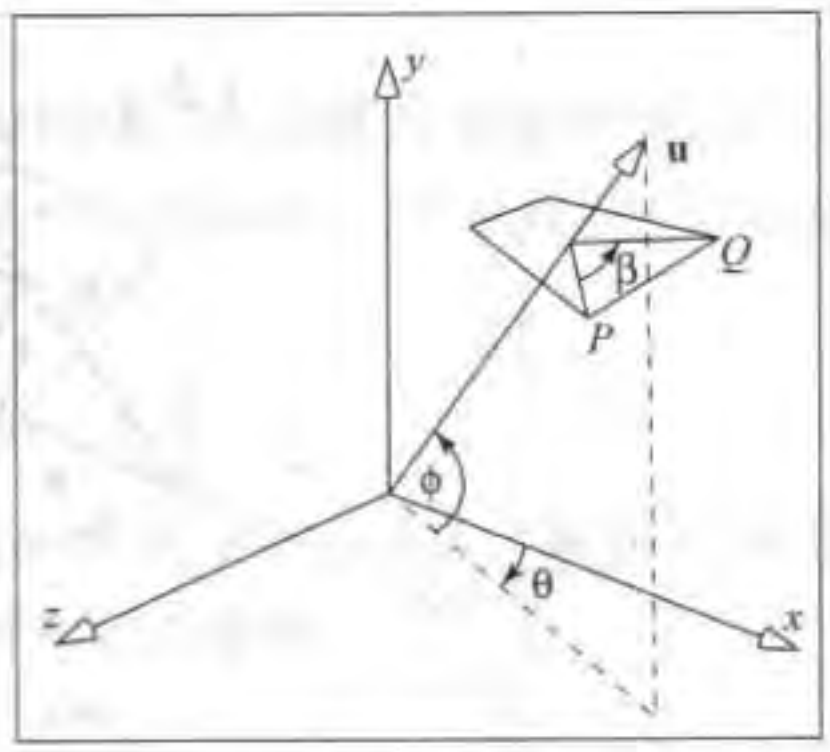


图 5.25 绕某轴旋转一定角度

经典方式

将所需要的旋转按照一固定的步骤进行分解：

1. 进行两个旋转，使得 **u** 和 z 轴重合。
2. 以 β 角度进行 z-roll 旋转。
3. 做步骤 1 中的两个旋转的逆操作，使得 **u** 回到原来的方向。

这里回忆一下二维情况下绕一个点的旋转：第一步进行预备变换；然后是完成简单变换；最后是完成预备变换的逆变换。结果是(练习中讨论过)变换需要五个矩阵相乘：

$$R_u(\beta) = R_z(-\theta)R_y(-\phi)R_z(\beta)R_y(\phi)R_z(\theta) \tag{5.28}$$

其中每一个都是关于一个坐标轴的旋转。手工操作是很繁琐的，但是编程实现是很简单的。尽管如此，对于上述乘积得到的矩阵也很难看出其各个分量是受哪些因素影响的。

构造性方法

使用某些向量工具，我们可以得到矩阵 $R_u(\beta)$ 的某种更有启发意义的表达方式。最近这种方式变得流行起来，在 GEMS I【Classner90】中有几个作者的若干不同的版本描述。我们选择 Maillot 的推导过程【Maillot90】。

图 5.26 中有一个旋转轴 **u**，我们想要得到 P 点经过角度 β 旋转到 Q 的操作。这个方法(案例 5.5 中详细讲述)通过一种有效的方式在旋转平面建立一个二维坐标系统，如图所示。如图 5.26(b)所示，定义了平面内的两个正交向量 **a** 和 **b**，点 Q 可以用它们的线性组合表示。Q 的表达式中包括向量点乘和叉积运算。但是因为每一项都是点 P 的坐标分量的线性组合，所以结果可以写成 P 乘以一个矩阵的形式。

最终结果是矩阵：

$$R_u(\beta) = \begin{bmatrix} c+(1-c)\mathbf{u}_x^2 & (1-c)\mathbf{u}_y\mathbf{u}_x - s\mathbf{u}_z & (1-c)\mathbf{u}_z\mathbf{u}_x + s\mathbf{u}_y & 0 \\ (1-c)\mathbf{u}_x\mathbf{u}_y + s\mathbf{u}_z & c+(1-c)\mathbf{u}_y^2 & (1-c)\mathbf{u}_z\mathbf{u}_y - s\mathbf{u}_x & 0 \\ (1-c)\mathbf{u}_x\mathbf{u}_z - s\mathbf{u}_y & (1-c)\mathbf{u}_y\mathbf{u}_z + s\mathbf{u}_x & c+(1-c)\mathbf{u}_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{5.29}$$

其中 $c=\cos(\beta)$，$s=\sin(\beta)$，并且(u_x, u_y, u_z)是单位向量 **u** 的分量形式。这个矩阵看起

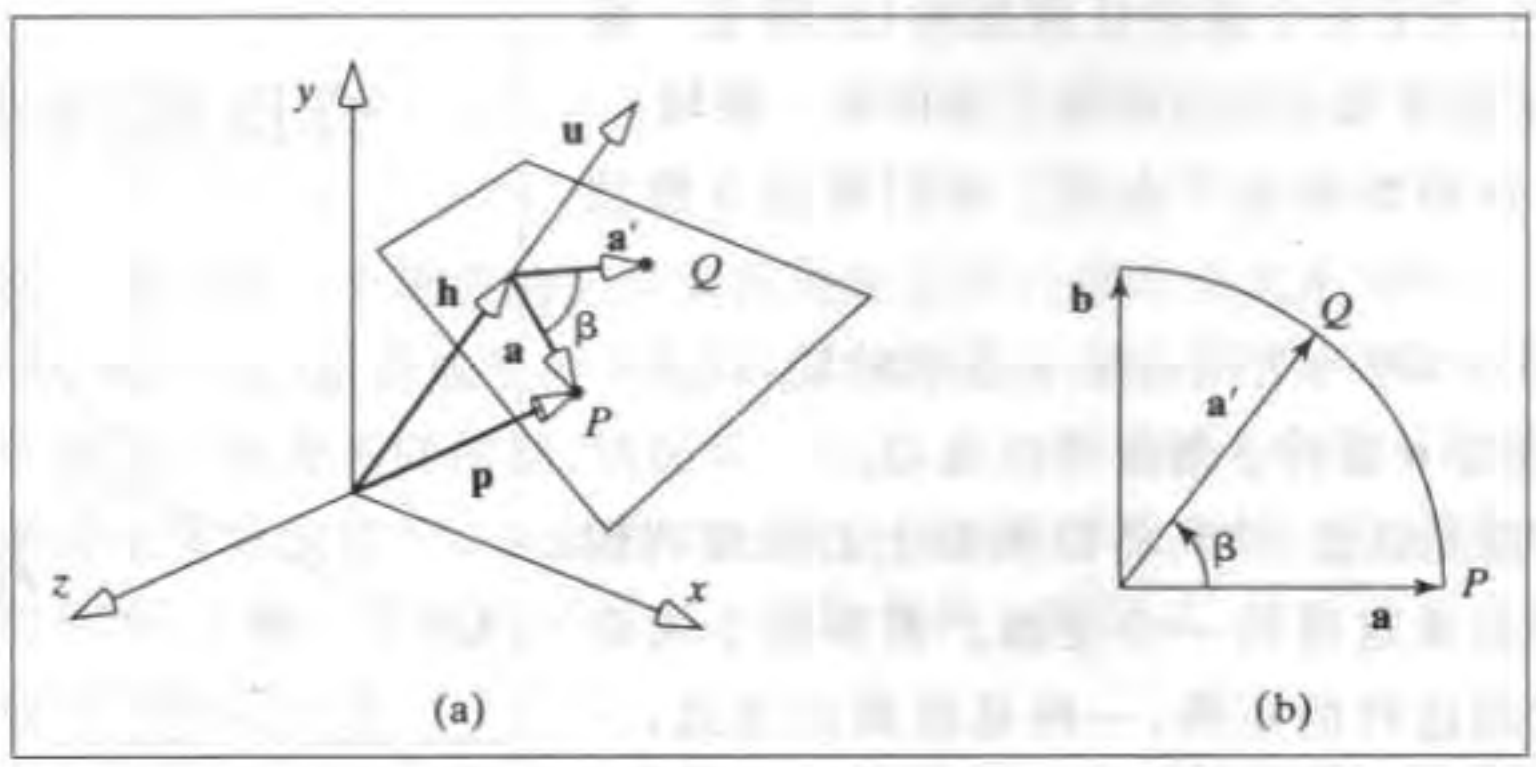


图 5.26 P 旋转到 Q

来很复杂,但是事实上我们后面会看到,它的组织形式是很有结构的,给定任意一个这样的旋转矩阵,我们都可以找出它的旋转轴和旋转角度(这也就证明出了欧拉定理)。

后面我们会看到,OpenGL 提供了一个函数可以创建绕任意轴的旋转,叫做:

```
glRotated(angle, ux, uy, uz);
```

这展示了 OpenGL 的强大功能,可以帮助编程人员得到想要的结果。

例 5.3.4 关于轴旋转

找出绕轴 $\mathbf{u}=(1,1,1)/\sqrt{3}=(0.577,0.577,0.577)$ 45°度的旋转矩阵。

解:对于 45°的旋转来说,$c=s=0.707$,然后填入等式(5.29),我么可以得到:

$$R_{\mathbf{u}}(45^\circ)=\begin{bmatrix} 0.8047 & -0.31 & 0.5058 & 0 \\ 0.5058 & 0.8047 & -0.31 & 0 \\ -0.31 & 0.5058 & 0.8047 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

这个矩阵的行列式为 1。图 5.27 显示了房子的基本变换,从原来的位置(底部)绕着 **u** 轴旋转 22.5°(中间),和旋转 45°(上面)后的结果。

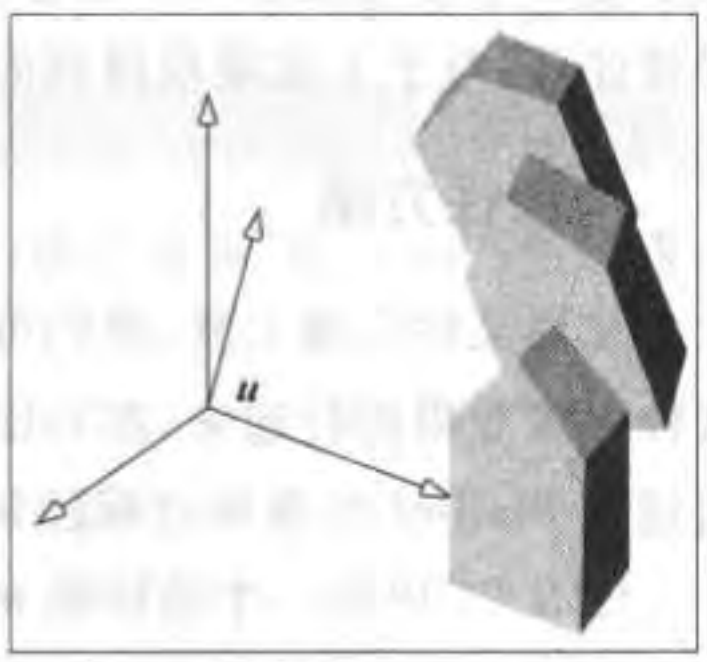


图 5.27 房子绕轴 **u** 的基本变换

找出旋转的轴和角度

欧拉定理保证了任何一个旋转等价于一系列绕坐标轴的旋转。当给定某个旋转矩阵时,计算出它的旋转轴和旋转角度是很有用处的。

假设矩阵的 9 个值 m_{ij} 已经给定:

$$R_u(\beta)=\begin{bmatrix} m_{11} & m_{12} & m_{13} & 0 \\ m_{21} & m_{22} & m_{23} & 0 \\ m_{31} & m_{32} & m_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

我们怎样才能提取旋转角度 β 和所绕的轴 **u** 呢?

通过检查等式(5.29)将会很容易做到这一点【Watt92】。首先请注意矩阵 $R_{\mathbf{u}}(\beta)$ 的迹——也就是说四个对角线元素的和——是 $1+(1+2c)=2+2\cos(\beta)$。所以我们可以直接得到 $\cos(\beta)$ 为:

$$\cos(\beta)=\frac{1}{2}(m_{11}+m_{22}+m_{33}-1)$$

通过对此值求反余弦可以得到角 β 进而还可以计算出 $s=\sin(\beta)$。现在观察等式(5.29),看一看如何通过矩阵中的某一对分量组合生成 **u** 的分量的:

$$\begin{aligned} u_x &= \frac{m_{32}-m_{23}}{2\sin(\beta)} \\ u_y &= \frac{m_{13}-m_{31}}{2\sin(\beta)} \\ u_z &= \frac{m_{21}-m_{12}}{2\sin(\beta)} \end{aligned} \tag{5.30}$$

例 5.3.5 找出旋转轴和旋转角度

假设你不知道例 5.3.3 中构成旋转矩阵的旋转轴和旋转角度,那么请求出它。矩阵的迹是 2.414,所以 $\cos(\beta)=0.707$,所以 β 一定是 45°,$\sin(\beta)=0.707$。现在计算等式(5.30)中的每一项:所有的都是 0.577,因此 $\mathbf{u}=(1,1,1)/\sqrt{3}=(0.577,0.577,0.577)$,这和预期的一样。

练习

练习 5.3.6 那些可以交换?

考虑两个仿射变换 T_1 和 T_2,下面几种情况那些 $T_1T_2=T_1T_1$ 成立:

a. 两个都是纯粹的平移

b. 两个都是缩放

c. 两个都是剪切

d. 一个是旋转,另外一个是平移

e. 一个是旋转,另外一个是缩放

f. 一个是缩放,另外一个是剪切

练习 5.3.7 关于绕一般轴 u 旋转的特殊情况

复杂的结果在特例时通常会退化为我们熟悉的形式,这一点很有帮助。当 **u** 是如下情况时请检验等式(5.30)的结果:

a. x 轴 **i**

b. y 轴 **j**

c. z 轴 **k**

特别地,求出其旋转矩阵并且通过它计算出一些相关的项,比如 β 等等。

练习 5.3.8 正交矩阵

如果矩阵的列是相互正交的单位向量,那么这个矩阵是正交阵。证明等式(5.23)~等式(5.25)给出的旋转矩阵都是正交的。正交矩阵的行列式是什么呢?正交矩阵有一个极好的性质:它的逆和转置相等(参见附录 2)。请证明为什么列的正交性就可以保证如上结论。找出前面所述的 3 个矩阵的逆,并请说明一个旋转的逆其实很简单,就是向相反方向的旋转。

练习 5.3.9 矩阵是正交的

证明等式(5.30)所示的复杂旋转变换也是正交的。

练习 5.3.10 旋转矩阵的结构

证明 3×3 的旋转矩阵的列是两两正交的,第三个列是前两个列的叉积。

练习 5.3.11 如果旋转轴不通过原点会怎样?

如果轴不通过原点而是通过 $S+\mathbf{u}t$ 给出的,其中 S 为某一个点,那么我们必须平移 $-S$ 到原点,然后进行适当的旋转,最后再平移 S。请推导出结果矩阵。

5.3.4 总结三维仿射变换的性质

5.2.8 节所讲述的二维仿射变换的性质可以用在三维仿射变换上面。任意一个三维仿射变换 $T()$ 都有一个矩阵 M 相对应,它们是:

- **保持点的仿射组合**。如果 $a+b=1$,那么 $aP+bQ$ 表示一个三维点,有 $T(aP+bQ)=aT(P)+bT(Q)$。
- **保持共线和共面关系**。保持共线关系:在三维情况下直线 L 的像 $T(L)$ 是另一条直线;三维情况下平面 W 的像 $T(W)$ 是另一个平面。
- **保持直线和平面的平行性**。如果 W 和 Z 是平行的两条直线(或平面),那么 $T(W)$ 和 $T(Z)$ 也是平行的。
- **矩阵的列表示出了变换后的坐标系**。如果矩阵的列是 $\mathbf{m}_1$、$\mathbf{m}_2$、$\mathbf{m}_3$ 和点 m_4,整个变换把坐标系 $(\mathbf{i},\mathbf{j},\mathbf{k},\phi)$ 变换为 $(\mathbf{m}_1,\mathbf{m}_2,\mathbf{m}_3,m_4)$。
- **保持相关比例**。如果 P 点是 A 点到 B 点的一部分,比例为 f,那么 $T(P)$ 也是 $T(A)$ 和 $T(B)$ 的一部分,比例也为 f。
- **变换对物体体积的影响**。如果三维物体 D 的体积是 V,那么变换以后的像 $T(D)$ 的体积为 $|\det M|V$,其中 $|\det M|$ 是矩阵的行列式的绝对值。

每一个仿射变换都由基本变换组成。一个三维仿射变换可以分解成一系列的基本变换。参见案例分析 5.2,有很多方法可以达到此目的。

5.4 如何实现坐标系变换

有另一种方法去考虑仿射变换,在很多方面它可以更加自然地对一个场景建模。不同于将仿射变换看作是在一个固定坐标系中生成不同点的观点,你也可以把它想象成生成一

个新的坐标系去表示点。

题外话：说一下表示的方法。为了适应印刷出来的页面，我们使用下面的记号：

$$\text{用}(P_x, P_y, 1)^T \text{ 代替} \begin{pmatrix} P_x \\ P_y \\ 1 \end{pmatrix}$$

（也可以参见附录 2）角标 T 表示转置，因此我们可以把列向量写成行向量转置的形式。

假设我们有一个二维的坐标框架 #1 如图 5.28 所示，它具有原点 ϕ 和轴 $\mathbf{i}$、$\mathbf{j}$。进一步假设我们有一个仿射变换 $T()$，可以用矩阵 M 表示，此变换将坐标系统 #1 变换为 #2，新的原点是 $\phi' = T(\phi)$，新的轴是 $\mathbf{i}' = T(\mathbf{i})$ 和 $\mathbf{j}' = T(\mathbf{j})$。

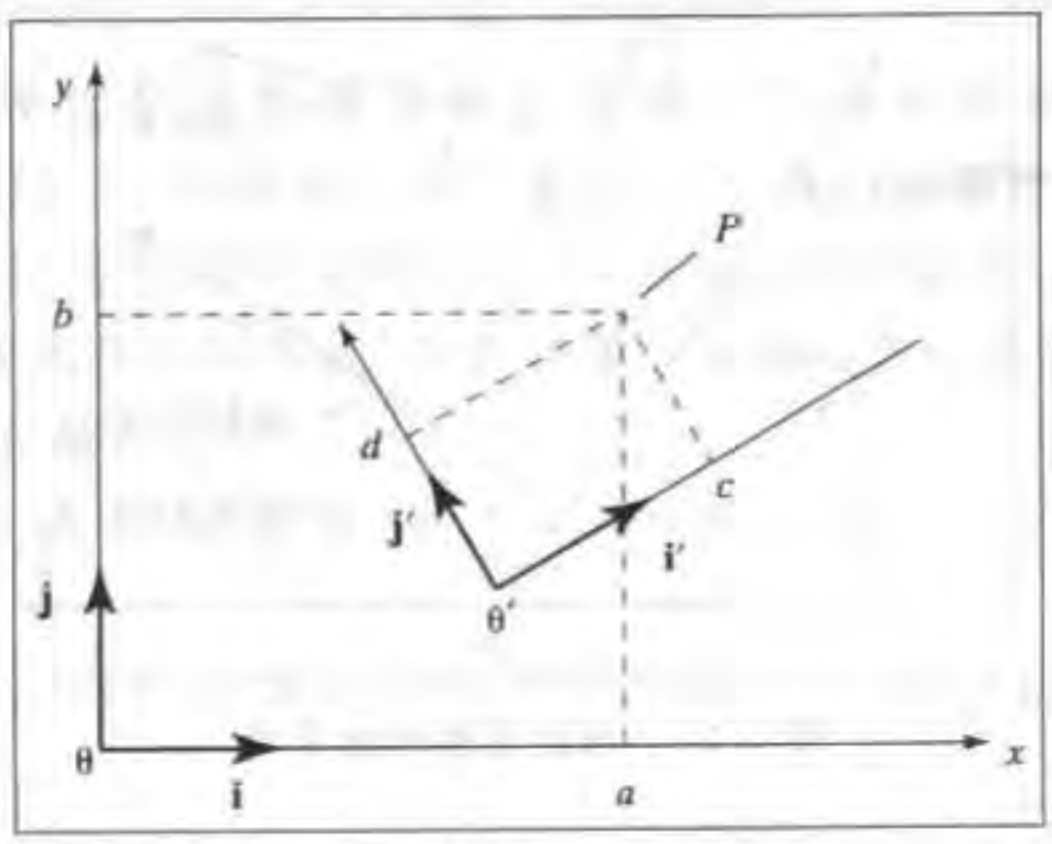


图 5.28　坐标变换

现在假设点 P 在坐标系 #2 下有一个新的表示 $(c, d, 1)^T$。那么在坐标系 #1 下，其表示 $(a, b, 1)^T$ 中 a 和 b 的值都是多少？答案很简单：用 M 和 $(c, d, 1)^T$ 相乘：

$$\begin{pmatrix} a \\ b \\ 1 \end{pmatrix} = M \begin{pmatrix} c \\ d \\ 1 \end{pmatrix} \tag{5.31}$$

作为结论，我们有如下的定理：

假设坐标系 #2 是坐标系 #1 经过仿射变换 M 得到的。进一步假设 $(P_x, P_y, P_z, 1)$ 是点 P 在坐标系 #2 下的表示。然后，在坐标系 #1 下 P 的坐标是 MP。

这个定理对某些读者来说是显而易见的，但是如果有读者不太清楚的话，可以在本节最后的练习中进行推导。当然这个结果在三维系统中同样可以应用，在计算三维点通过 OpenGL 管线如何进行变换时经常要用到它。

例 5.4.1　坐标系旋转

再一次考虑例 5.2.5 中的变换，把点绕 $(-2, 3)$ 旋转 30°。如图 5.23 所示，这个变换把原点 ϕ 和轴 $\mathbf{i}$、$\mathbf{j}$ 映射到坐标系统 #2 中。下面考虑在新的坐标系中坐标为 $(P_x, P_y, 1)^T$ 的点 P。这个点在原坐标系 #1 中的坐标是什么？答案就是 MP。例如，新坐标系中的点 $(1, 2, 1)^T$ 在原坐标系中的坐标是 $M(1, 2, 1)^T = (1.098, 3.643, 1)^T$（请在图中画出来）。注意到变换的中心点 $(-2, 3, 1)^T$ 是变换的不动点：$M(-2, 3, 1)^T = (-2, 3, 1)^T$。因此，如果我们

在新坐标系中取 $P=(-2,3,1)^T$,那么它在原坐标系中的像是$(-2,3,1)^T$。

坐标系的连续变化

考虑接连进行两次坐标系变换会形成一个怎样的变换,整体效果又怎样呢? 如图 5.29 中所示,坐标系 #1 经过变换 $T_1()$变换到坐标系 #2,然后坐标系 #2 又经过变换 $T_2()$变换到 #3。注意,#3 是和 #2 相关的。

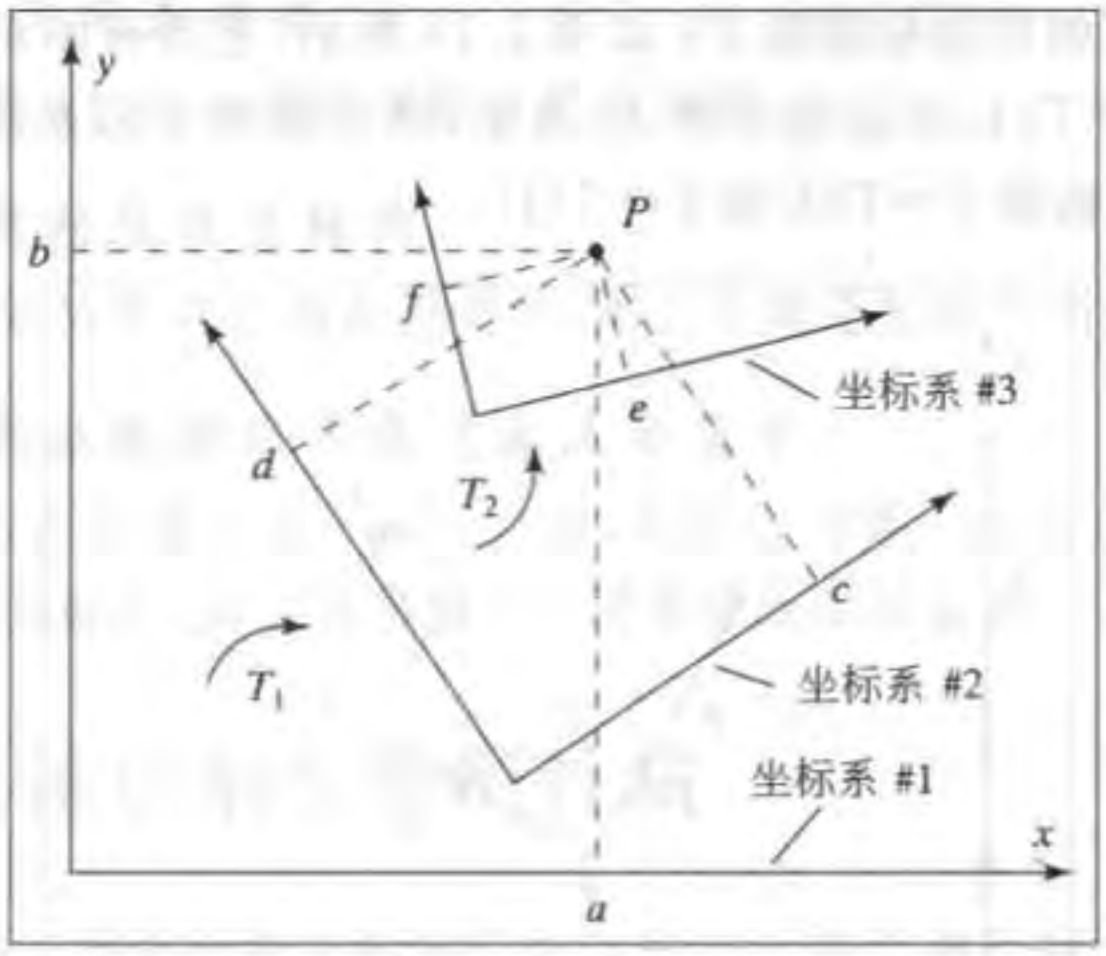


图 5.29 两次变换坐标系统

问题又来了:如果点 P 在坐标系 #3 中的坐标是$(e,f,1)^T$,那么它在坐标系 #1 下的表示$(a,b,1)^T$ 是多少呢?

为了回答这个问题,只需要逆向工作,然后集合每一个变换的效果。在坐标系 #2 下,P 点的坐标是$(c,d,1)^T=M_2\ (e,f,1)^T$。在坐标系 #1 下点$(c,d,1)^T$ 有坐标$(a,b,1)^T=M_1\ (c,d,1)^T$,将这些变化放在一起:

$$\begin{bmatrix} a \\ b \\ 1 \end{bmatrix} = M_1 \begin{bmatrix} c \\ d \\ 1 \end{bmatrix} = M_1 M_2 \begin{bmatrix} e \\ f \\ 1 \end{bmatrix} \tag{5.32}$$

要点就是当我们从$(e,f,1)^T$ 去求$(a,b,1)^T$ 时,我们首先作用 M_2,然后 M_1,和变换的作用顺序相反。

我们用三次连续变换的情形来总结这个事实,它可以很快地归纳到任意数量变换的情形:

变换点

应用一系列变换 $T_1()$、$T_2()$、$T_3()$(按顺序)到一个点 P,其中变换 $T_i()$可以使用 M_i 表示,构造矩阵:

$$M = M_3 \times M_2 \times M_1$$

那么 P 点被变换到 MP。也就是说每一次变换 M_i,就在前面乘以 M_i。

变换坐标系

应用一系列变换 $T_1()$、$T_2()$、$T_3()$(按顺序)到一个坐标系,构造矩阵:

$$M = M_1 \times M_2 \times M_3$$

然后，在经过变换的坐标系中的点 P，在原坐标系中有坐标 MP。每增加一个变换 M_i，就在后面乘以 M_i，那么乘以 P 的矩阵就是 $M_1M_2M_3\cdots M_i$。

OpenGL 怎么操作？

下一节我们会看到 OpenGL 提供的工具完成连续的变换，得到最终的整体变换。事实上，OpenGL 是用后面乘的方法去把每一个变换和当前的变换组合起来的。因此，对于建模人员来说，用连续坐标系的变换方式考虑问题会显得更加自然，这是因为这些变换作用的顺序和 OpenGL 的计算顺序是一致的。

练习

练习 5.4.1 变换一个坐标系怎么和变换一个点联系起来

我们希望说明等式(5.31)的结果。为了说明这个，我们需要首先说明下面每一步：

a. 说明在坐标系 #2 中的点 P 坐标为 $(c,d,1)^T$，它的位置是 $c\mathbf{i}'+d\mathbf{j}'+\phi'$。

b. 我们想找出这个点在坐标系 #1 中的位置。说明在坐标系 #1 中，$\mathbf{i}'$ 是 $M(1,0,0)^T$，$\mathbf{j}'$ 是 $M(0,1,0)^T$，原点 ϕ' 是 $M(0,0,1)^T$。

c. 因此说明这个点的表示为 $ci'+dj'+\phi'=cM(1,0,0)^T+dM(0,1,0)^T+M(0,0,1)^T$。

d. 说明它与 $M(c,d,1)^T$ 的表达形式 $M(c,0,0)^T+M(0,d,0)^T+M(0,0,1)$ 是等价的。

练习 5.4.2 基本的例子

图 5.30 显示了四个基本坐标系变换的效果。每一种情况下，原始坐标系的坐标轴 x 和 y 变到新坐标系下的坐标轴 x' 和 y'。

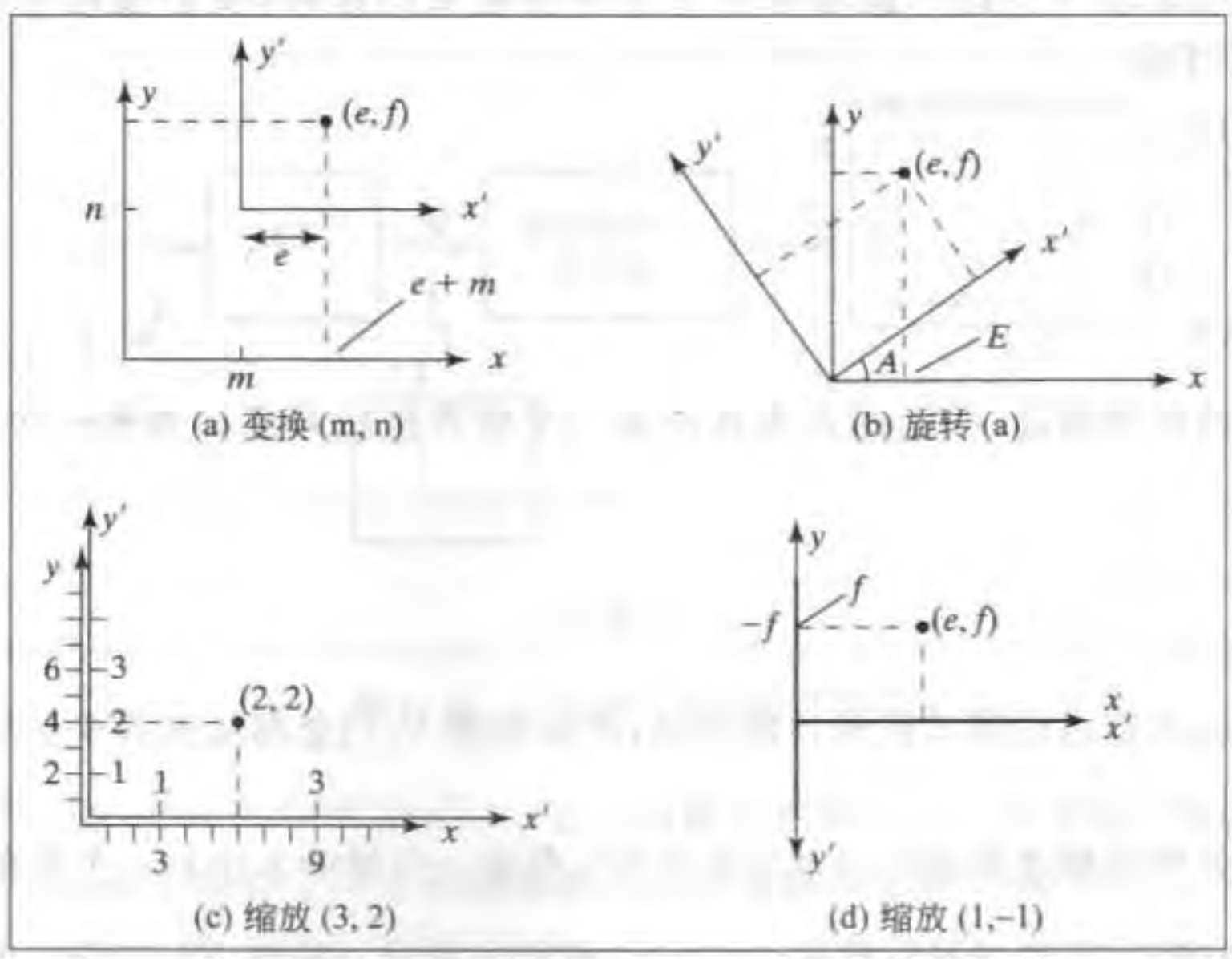


图 5.30 四个基本坐标系变换的效果

5.5 在程序中使用仿射变换

我们想看看怎样将仿射变换的理论应用到实际中,以得到物体的缩放、旋转和平移。我们将探究 OpenGL 是怎样实现这些的。我们先来看看二维实例,因为它们比较容易可视化,然后我们再转到三维实例。

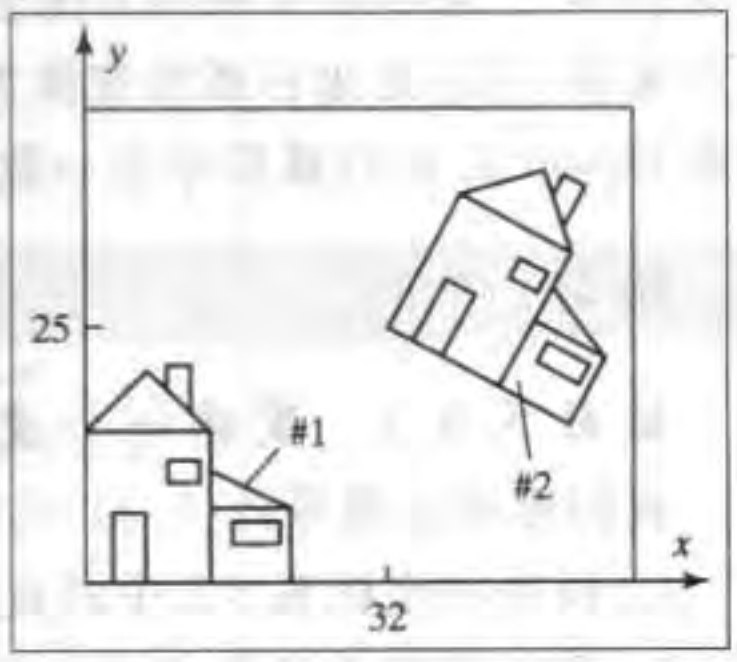


图 5.31 房子的旋转与变换

假设你有一个程序 house()可以绘出图 5.31 中的房子#1。但是你想画经过旋转 $-30°$,然后又平移了(32,25)的#2 视图。这是很普遍的情况:我们以恰当的位置和大小定义一个物体,但是我们却想在不同的大小、角度和位置绘制它。

就像我们在第 3 章中讨论的,house()可以画出各种不同的图像。如果用 OpenGL 实现,它肯定包含很多像这样的片断:

```
glBegin(GL_LINES);
    glVertex2d(V[0].x,V[0].y);
    glVertex2d(V[1].x,V[1].y);
    glVertex2d(V[2].x,V[2].y);
    … //其他点
glEnd();
```

这些都是基于点数组 V[]的。或者像第 3 章中所定义的那样,我们使用 moveTo()或者 lineTo(),可以写成:

```
moveTo(V[0]);
lineTo(V[1]);
lineTo(V[2]);
… //其他点
```

对于上述的两种情况,我们都会为其设置一个世界坐标系窗口和和一个视口,有如下形式:

```
setWindow(…);
setViewport(…);
```

同时我们还知道通过窗口到视口的变换,所有的顶 V[i]全都从世界坐标变换到了屏幕坐标。

但是我们如何安排才能够绘出#2 房子呢?存在一个难的方法和一个简单的方法。

困难的方法

使用这种方式我们要构造出所需的变换矩阵,称为 M,同时构造一个函数,称为

transform2D()，它可以把一个点变换到另一个点，比如：

```
Q=transform2D(M,P);
```

程序会计算出 $\widetilde{Q}=\widetilde{M}\widetilde{P}$。为了在 house()将这个变换应用到每个点 $V[i]$，我们需要修改上面的源代码，例如：

```
moveTo(transform2D(M,V[0]));
lineTo(transform2D(M,V[1]));
lineTo(transform2D(M,V[2]));
```

这样经过变换的点就被传递给 moveTo()和 lineTo()。如果我们有 house()的代码，这个方法肯定可行。但是即使在最好的情况下它仍然很麻烦，并且如果不能得到 house()的代码的话就根本不可能了。并且还首先需要有工具去创建矩阵 M。

简单的方式

我们希望所需的变换可以自动作用到每一个顶点上。我们知道，窗口到视口的点变换被悄悄地放到了 moveTo()和 lineTo()中进行，同样我们也可以用另一个附加的变换。它通常叫做**当前变换**(CT)。这样 moveto()和 lineTo()的功能就被我们增强了，它们先对顶点实施这个变换，然后再执行窗口到视口的变换(裁剪发生在世界窗口边界)。

图 5.32 扩充了图 5.6 中曾经介绍过的图形绘制管道的内容。当参数 V 被传递给 glVertex2d()时，V 首先被 CT 变换到点 Q。(图中标上“世界窗口”的框图应该是紧跟着 CT 变换的)Q 点然后经过窗口到视口的变换得到屏幕窗口上的一点 S(后面我们也会看到，在这个映射过程的最后阶段会存在裁剪操作)。

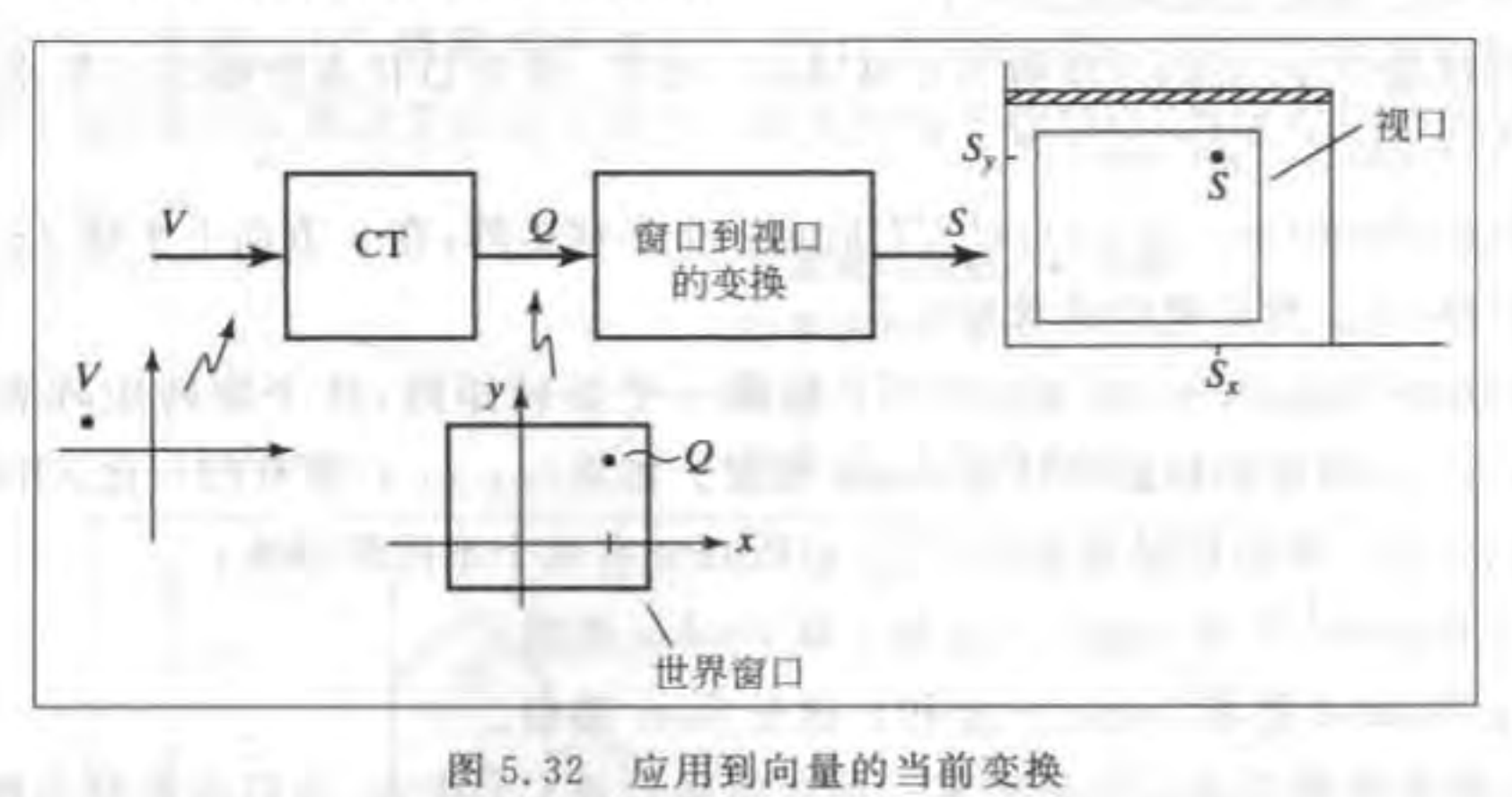


图 5.32　应用到向量的当前变换

我们怎样扩展 moveTo()和 lineTo()去实施额外的映射呢？很幸运，OpenGL 会自动完成这个变换！OpenGL 保存了一个所谓的**模型视图矩阵**(modelview matrix)，每一个通过图形管线的点都会乘以这个矩阵。只需要设置一次这个模型视图矩阵就可以实施我们所需的变换。

OpenGL 完全工作在三维环境下,因此它的模型视图矩阵进行的是三维变换。我们在这里对模型视图矩阵进行一下限制,使其工作在二维变换下。后面我们将会使用它的全部能力。图 5.33 说明了我们怎样对三维变换进行限制,使其能够执行二维变换。主要的想法就是把二维的绘制限制在 xy 平面:z 坐标为 0。因此当我们变换二维点时,我们把底层的三维变换中能够影响到 z 坐标的部分进行一下设置,使其对 z 坐标不发生影响。例如,二维的关于原点的旋转相当于关于 z 轴的三维旋转,如图所示。进一步,尽管三维缩放有 3 个参数 S_x、S_y 和 S_z 分别进行 x、y 和 z 轴上的缩放,但是我们设置 $S_z=1$。

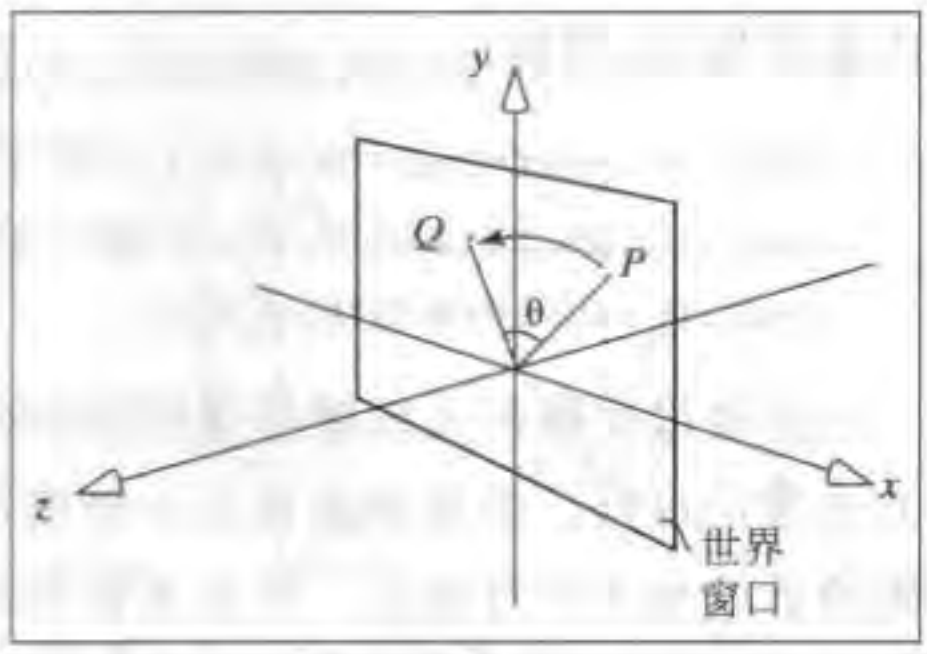


图 5.33 在 xy 平面上的二维绘图

改变模型视图矩阵的基本程序函数是 glRotated()、glScale 和 glTranslated()。这些都不直接设置 CT;相反每一个都是在 CT(模型视图矩阵)后面乘上一个特殊的矩阵 M,然后把结果又返回到 CT 中。也就是说这些变换根据需要生成一个矩阵 M,然后执行:

$$CT = CT * M \tag{5.33}$$

顺序很重要。像前面所讲的,对一个点应用 $CT * M$ 等价于首先进行 M 表示的变换,然后跟着进行直接由 CT 规定的变换。或者我们考虑坐标系变换的情形,这就是说对当前的坐标系再进行一次附加的变换。

OpenGL 程序进行二维变换的例子是:

- glScaled(sx, sy, 1.0);对 CT 后乘一个缩放矩阵,在 x 方向上缩放 sx,在 y 方向上缩放 sy。然后把结果放回 CT。

  最好试着将'gl'scaled'读做'GL SCALE DEE',因为 D 代表数据类型是 double,就像"glVertex2f","glColor3f"。
- glTranslated(dx, dy, 0);对 CT 后乘一个平移矩阵,在 x 方向上平移 dx,在 y 方向上平移 dy。然后把结果放回 CT。
- glRotatef(angle, x, y, z);对 CT 后乘一个旋转矩阵,这个旋转矩阵表示了绕着 (x, y, z) 所定的轴旋轴转动 angle 角度。如果 (x, y, z) 没有归一化,OpenGL 会将它归一化。然后把结果放回 CT。glRotate 有两个不同的版本:

  a. glRotated 要求 angle、x、y 和 z 取 double 类型。

  b. glRotatef 要求 angle、x、y 和 z 都是 float 类型。

由于这些函数的功能仅仅是把某一种变换融合进 CT 中去,我们还需要某种方式来开始:把 CT 初始化为单位矩阵。OpenGL 提供了 glLoadIdentity()函数,因为此函数可以对 OpenGL 支持的任意矩阵进行设置,所以我们还必须告诉 OpenGL 到底要对哪个矩阵进行操作。这就需要使用语句 glMatrixMode(GL_MODELVIEW)。

图 5.34 展示了 4 个用户编写的函数来管理 CT,这就允许我们可以建立任意的二维变换。它们的形式看上去很简单,这是因为困难的工作都交给了 OpenGL。

```
//<<<<<<<<<<<<<<<<initCT>>>>>>>>>>>>>>>>>>
void initCT(void)
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();                    //set CT to the identity matrix
}
//<<<<<<<<<<<<<<<<scale2D>>>>>>>>>>>>>>>>
void scale2D(double sx, double sy)
{
    glMatrixMode(GL_MODELVIEW);
    glScaled(sx,sy,1.0);                 // set CT to CT * (2D scaling)
}
//<<<<<<<<<<<<<<<<translate2D>>>>>>>>>>>>>>>>
void translate2D(double dx, double dy)
{
    glMatrixMode(GL_MODELVIEW);
    glTranslated(dx,dy,1.0);             //set CT to * (2D translation)
}
//<<<<<<<<<<<<<<<<rotate2D>>>>>>>> >>>>>>>>
void rotate2D(double angle)
{
    glMatrixMode(GL_MODELVIEW);
    glRotated(angle,0.0, 0.0, 1.0);  //set CT to CT * (2D rotation)
}
```

图 5.34 二维变换的 *CT* 管理函数

我们现在开始使用二维变换。下面我们要展示一些代码用于画图 5.35 中房子 #3 的视图，首先让房子旋转 −30°，然后再平移(32, 25)，使用的都是前面定义过的函数。注意，要保证顺序正确，那么应该以反向进行操作：首先使用平移操作，然后再进行旋转操作。

```
setWindow(…);
setViewport(…);                          //设置窗口到视口的变换
initCT();                                //以单位阵开始
house();                                 //绘制没有经过变换的房子
translate2D(32,25);                      //现在 CT 包含了平移
```

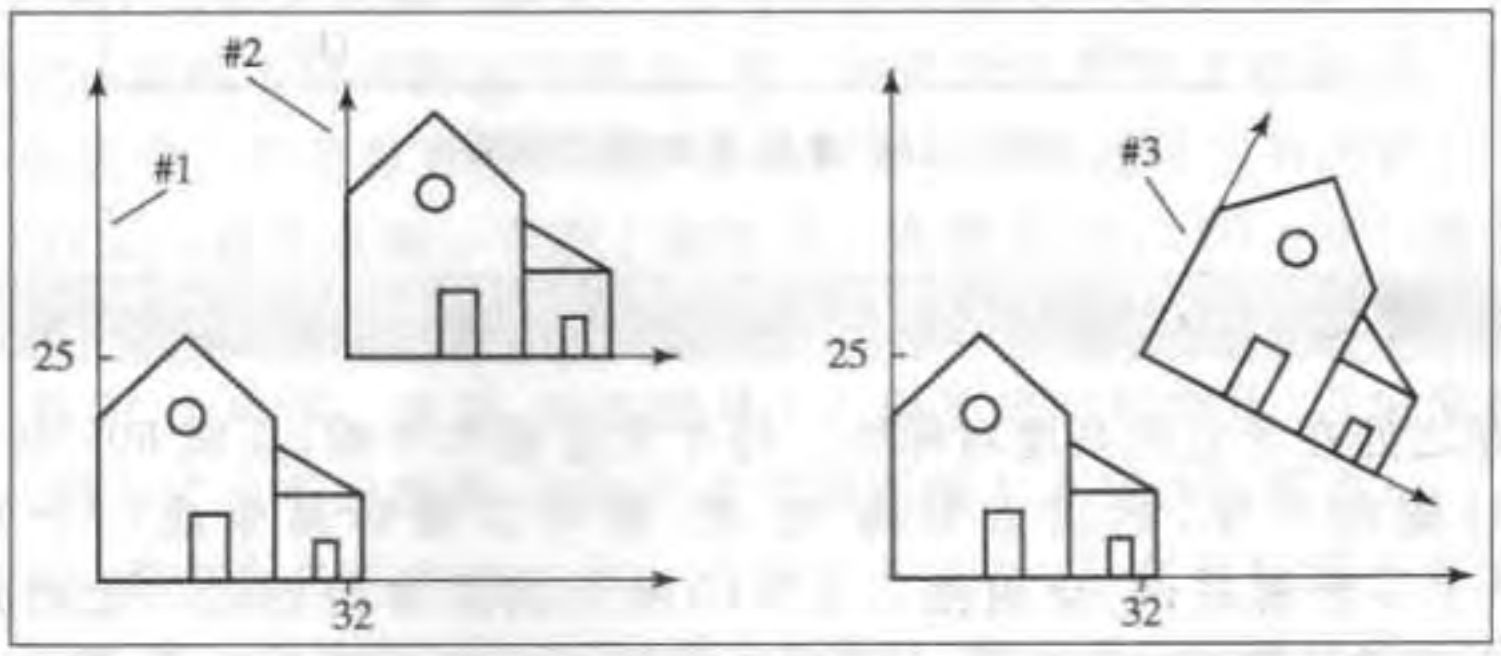


图 5.35 从坐标系的变换角度考虑

```
rotate2D(-30.0);                    //现在 CT 包含了平移和旋转
house;                              //绘制经过变形的房子
```

注意,我们可以以任意方式进行缩放、旋转和平移,并且不需要修改函数 house()的内部的程序。

有人发现从坐标系变换的角度考虑更加自然。如图 5.35 所示,他们会想首先将坐标系统平移(32, 25)形成新的坐标系 #2,然后对它旋转 −30°,得到新的坐标系 #3。因为 OpenGL 应用变换的顺序是坐标系变化的顺序,这部分代码是首先调用 translated2D(32, 25),然后调用 rotate2D(−30, 0)。当然,这和使用另一种方法得到的代码是一样的,只不过思考的角度不同。

下面我们给出一些例子,来说明如何对 *CT* 进行简单操作就可以获得想要的效果。

例 5.5.1 利用旋转对称性

图 5.36(a)展示了一个用条带构成的星星,其中的条带是相互交织在一起的。使用 rotate2D()将会很容易绘出此形状。假设函数 starMotif()绘制这个星星的一部分,如图 5.36所示的多边形。(确定这个多边形的顶点是很难的,详见后面的案例分析(5.1)为了绘出整个星星,我们需要将这个图形绘制 5 次。每次旋转这个图形 72°或五分之一的圆:

```
for( int count=0; count<5; count++)
{
    starMotif();
    rotate2D(72.0);
}
```

看看每一步会发生什么。

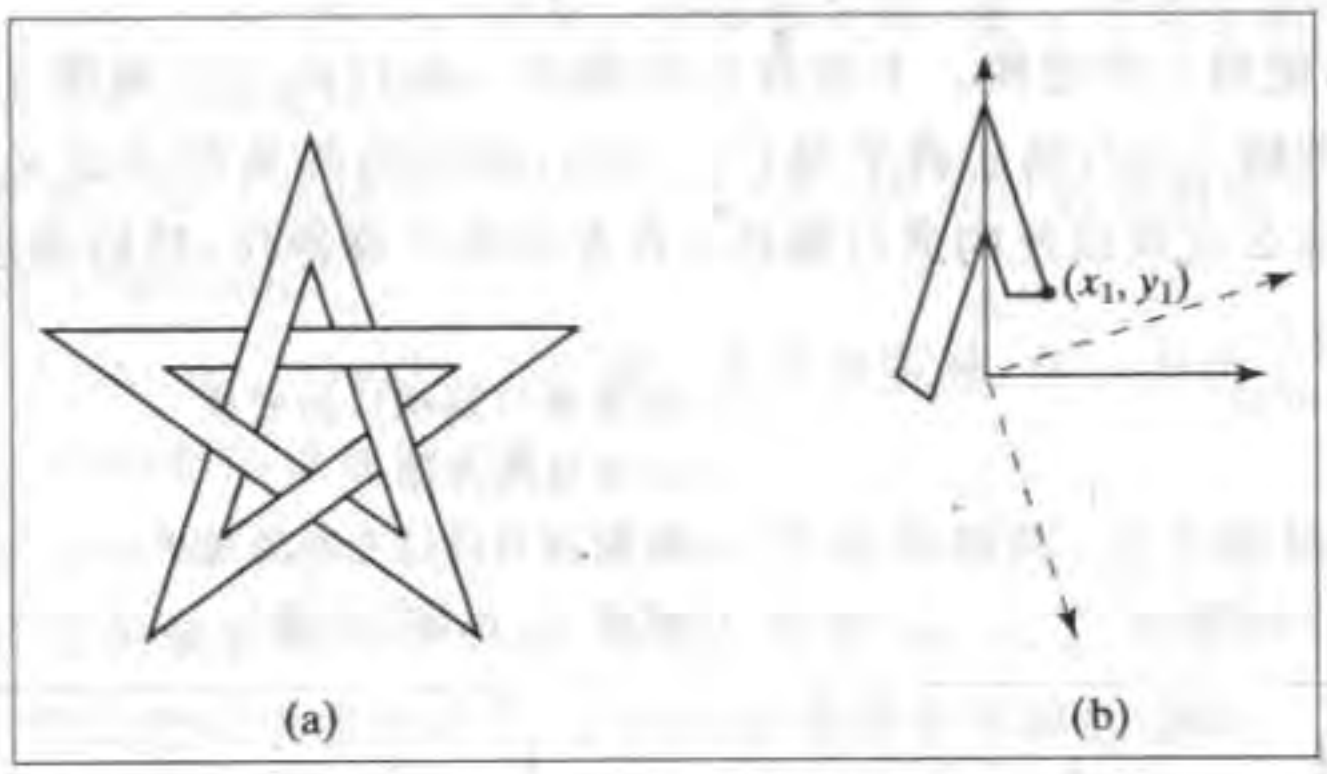


图 5.36 坐标系的连续旋转

例 5.5.2 雪花

雪花的漂亮之处在于它的高度对称性。每个雪花有六个瓣,互成 60°,每个瓣本身也是对称的。只设计瓣的一半,然后重复画 12 次,就可以很容易生成一个复杂的雪花。图 5.37(a)显示了一个雪花,它是由图 5.37(b)所示的图案得到的。此图案是一个位于正 x 轴方向上方的多边形(如图所示,为了避免和雪花的其他部分重叠,我们要保证这个多边形在 30°那条线下方)。

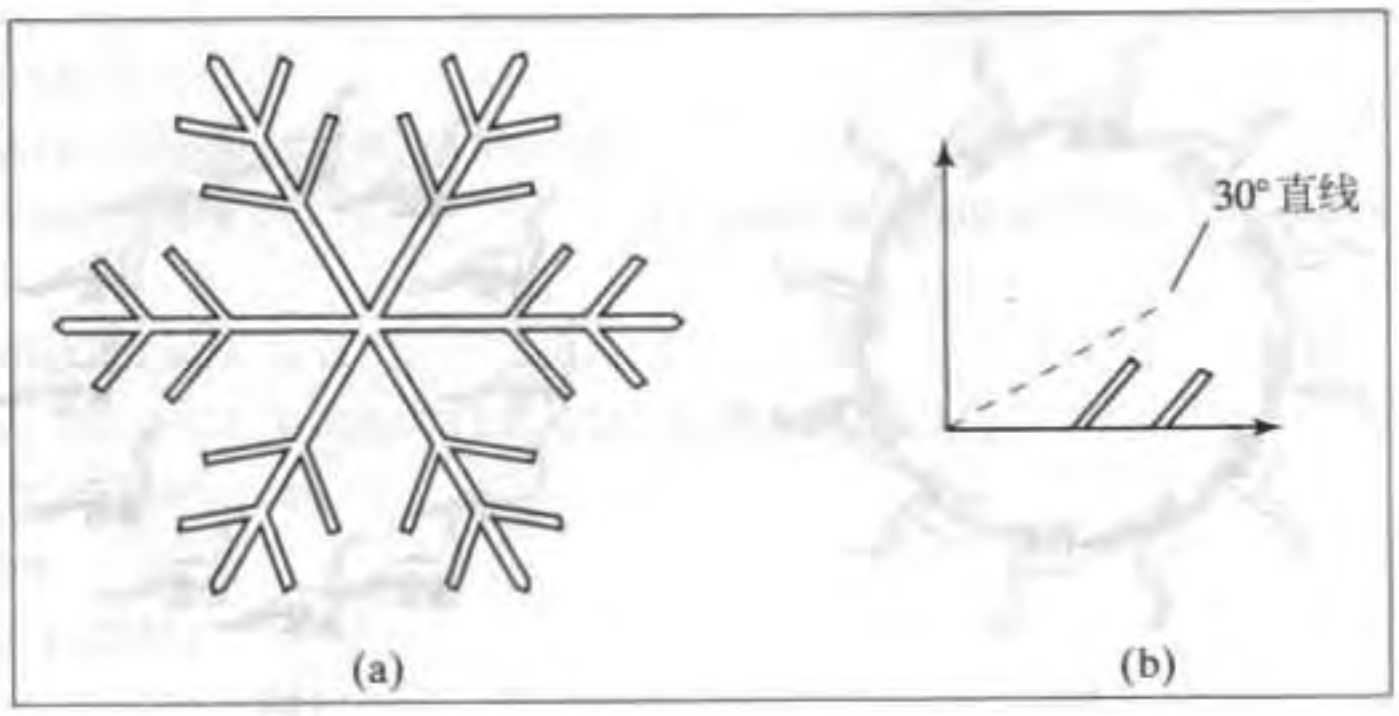


图 5.37 设计一片雪花

雪花的每一个瓣都由上述的图案和它的反向版本组成。通过 scale2D(1, −1)可以得到这个图案关于 x 轴的反向版本。所以此图案和它的反向可以如下绘制：

```
flakeMotif();
scale2D(1.0,-1.0);
flakeMotif();
scale2D(1.0,-1.0);
```

为了画出整个雪花，需要重复 6 次，每次之间还要旋转 60°：

```
void drawFlake()
{
    for (int count=0; count<6; count++)
    {
        flakeMotif();
        scale2D(1.0,-1.0);
        flakeMotif();
        scale2D(1.0,-1.0);
        rotate2D(60.0);
    }
}
```

例 5.5.3 利用恐龙组成图案

图 5.38 是两个由恐龙组成的图案。在两幅图中恐龙都是分布在一个圆上，但是一个是每个恐龙都经过一定的旋转使脚正对原点，另一种是所有的恐龙都朝上。两种情况都是使用平移和旋转生成的。但是看看操作顺序是如何影响图案还是很有趣的。

假设 drawDino()在原点画一个朝上的恐龙。在图 5.38(a)中，每个恐龙的坐标系都先绕原点旋转一定的角度(这样 y 轴的点直接向上穿过恐龙的背部)。然后坐标系沿着自己的 y 轴向上平移 H 个单位。注意，每次循环 CT 都要进行初始化，以免变换会累计在一起(或者用你自己的方式去考虑变换，你也许会使用图案上点变换的观点)。

像图 5.38(b)那样保持图案朝上有一个简单的方法，是在坐标系平移它之前提前旋转每一个图案。如果有一个图案要出现在 120°的位置，那它首先旋转 −120°(仍然在原点)，然后平移 H 个单位，最后旋转 120°。

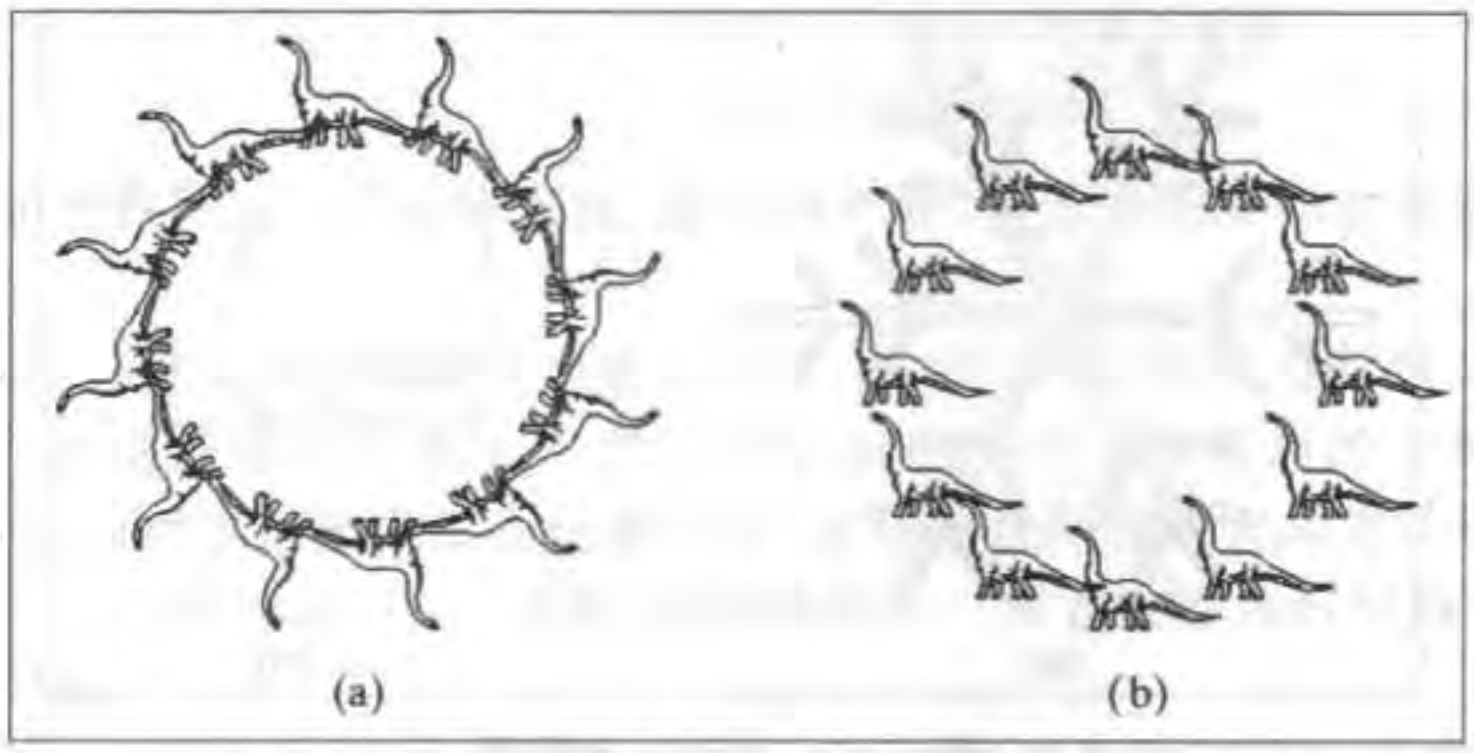


图 5.38 两个由恐龙组成的图案

(a) 每个恐龙的坐标系都先绕原点旋转；(b) 保持图案朝上

5.5.1 为后面的使用保存 CT

一个程序也许会多次调用 rotate2D()、scale2D()和 translate2D()形成一个冗长的序列。这些函数会对 *CT* 做一些附加的或相关的改变，但是我们有时候需要使用之前的 *CT*，以便为绘制新的图片开辟一条不同的变换路径。为了保存所需的 *CT*，我们复制一份 *CT* 并且放到一个方便的地方。然后在后面某个时刻，我们可以把存储的矩阵恢复为 *CT* 矩阵，这样可以高效地回到此矩阵有效时的变换状态。或许我们还想存储先前使用过的 *CT* 的一个集合，然后在某个时刻选择其中的一个。

为了达到这个目的，我们可以使用一个**变换栈**，如图 5.39 所示。在栈顶的矩阵就是实际的 *CT*，像 rotate2D()这样的操作就以先前介绍的方式与它进行融合。为了保存这个 *CT* 以备后用，需要复制一份，然后使用图 5.40 将要定义的 pushCT()将它压入栈顶。这就使得栈顶的两个矩阵是一样的。现在就可以调用其他的变换 scale2D()等变换来改变栈顶的矩阵了。要恢复之前的矩阵，就只需要调用 popCT()将栈顶矩阵弹出栈，然后丢弃之。这种后进先出的方式可以让我们恢复出上一个矩阵，上上个矩阵，甚至更早以前的。

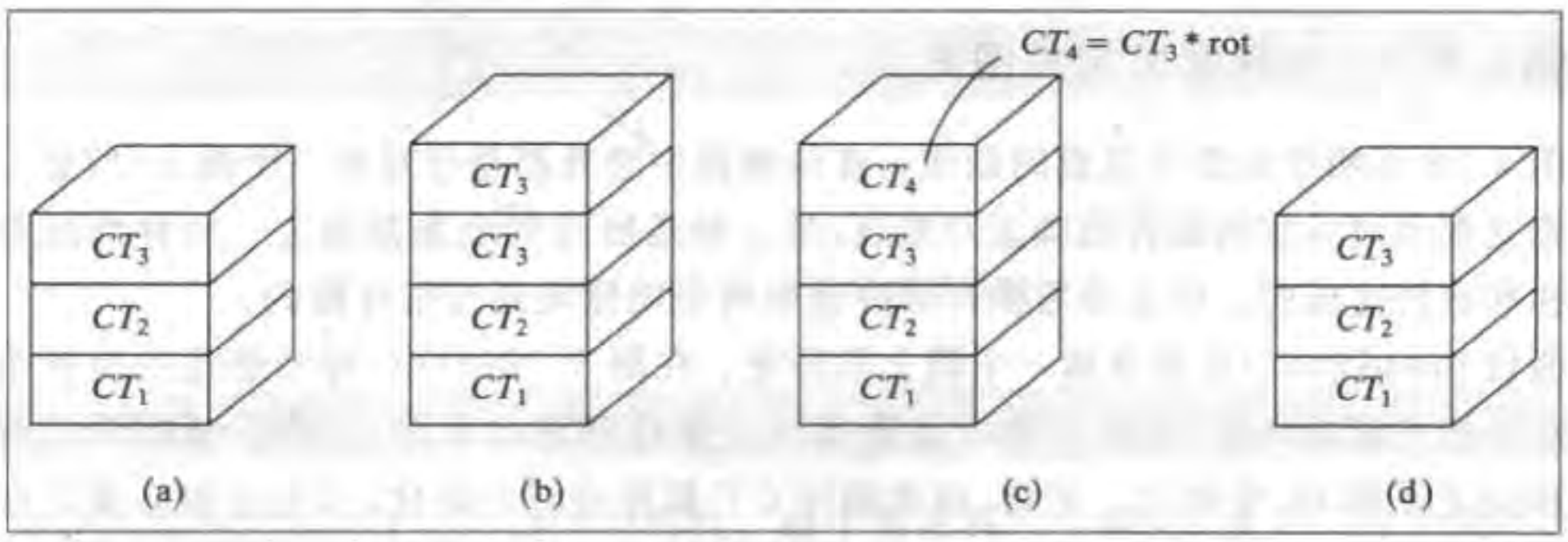


图 5.39 CT 栈

因为 OpenGL 提供了语句 glPushMatrix()和 glPopMatrix()来管理几个不同的栈，所以 popCT()和 pushCT()的实现很简单。图 5.40 是我们想要的函数例程。注意：每个函数

```
void pushCT(void)
{  glMatrixMode(GL_MODELVIEW);
    glPushMatrix();              // push a copy of the top matrix
}
void checkStack(void)
{  if (glGet (GL_MODELVIEW_STACK_DEPTH)≤1))
    // do something
    else
      popCT();
}
void popCT(void)
{  glMatrixMode(GL_MODELVIEW);
    glPopMatrix();               // pop the top matrix from the stack
}
```

图 5.40 CT 栈的函数例程

都必须告诉 OpenGL 要作用的是那个栈。另外，在操作栈的时候，会遇到的一个问题就是从一个空栈中弹出元素。在 OpenGL 中，从只含有一个矩阵的栈中弹出元素是错误的，因此在弹出之前必须确认矩阵的数量。幸运的是 OpenGL 提供了一个查询函数 glGet(GL_MODELVIEW_STACK_DEPTH)去确认栈中元素的个数。因此，图 5.40 的伪代码可以防止出现栈下溢的严重错误。

例 5.5.4 拼贴画其实很容易制作

有一种被称为拼贴画的设计在世界各地都很常见，它们出现在墙上、瓷器上或者纺织物上面。它们是由某些基本图形在水平方向和垂直方向进行重复得到的。如图 5.41 显示的那样，考虑在窗口中用此基本图案制作拼贴画。基本图案绘制在自己的局部坐标系中心，如图 5.41(a)所示，我们可以使用函数 motif()绘制这个图形。然后如图 5.41(b)所示，在 x 方向上每隔 L 个单位，在 y 方向上每隔 D 个单位，绘出拼贴画。

图 5.42 展示了如何简单地操作坐标系以便在两个循环内得到拼贴画。*CT* 在画完每一行后被重置，因此就回到了本行的起点，然后上移一行准备绘制下一个新行。另外，整段代码由 pushCT()开头并且由 popCT()结尾，因此绘制完拼贴画之后 *CT* 就被置为了初始值，为后面更多的绘制提供方便。

练习

练习 5.5.1 六边形拼贴画

六边形为拼贴画提供了更丰富的设置，因为规则六边形可以很规整地组合在一起就像蜂窝那样。图 5.43 是 9 列堆嵌在一起的六边形。这里的六边形都是空的，但是我们可以在其中绘制有趣的图形。写出能够绘制这个图案的代码。

练习 5.5.2

重复前面的练习，但是要提示使用者输入图案中六边形的行数和列数。

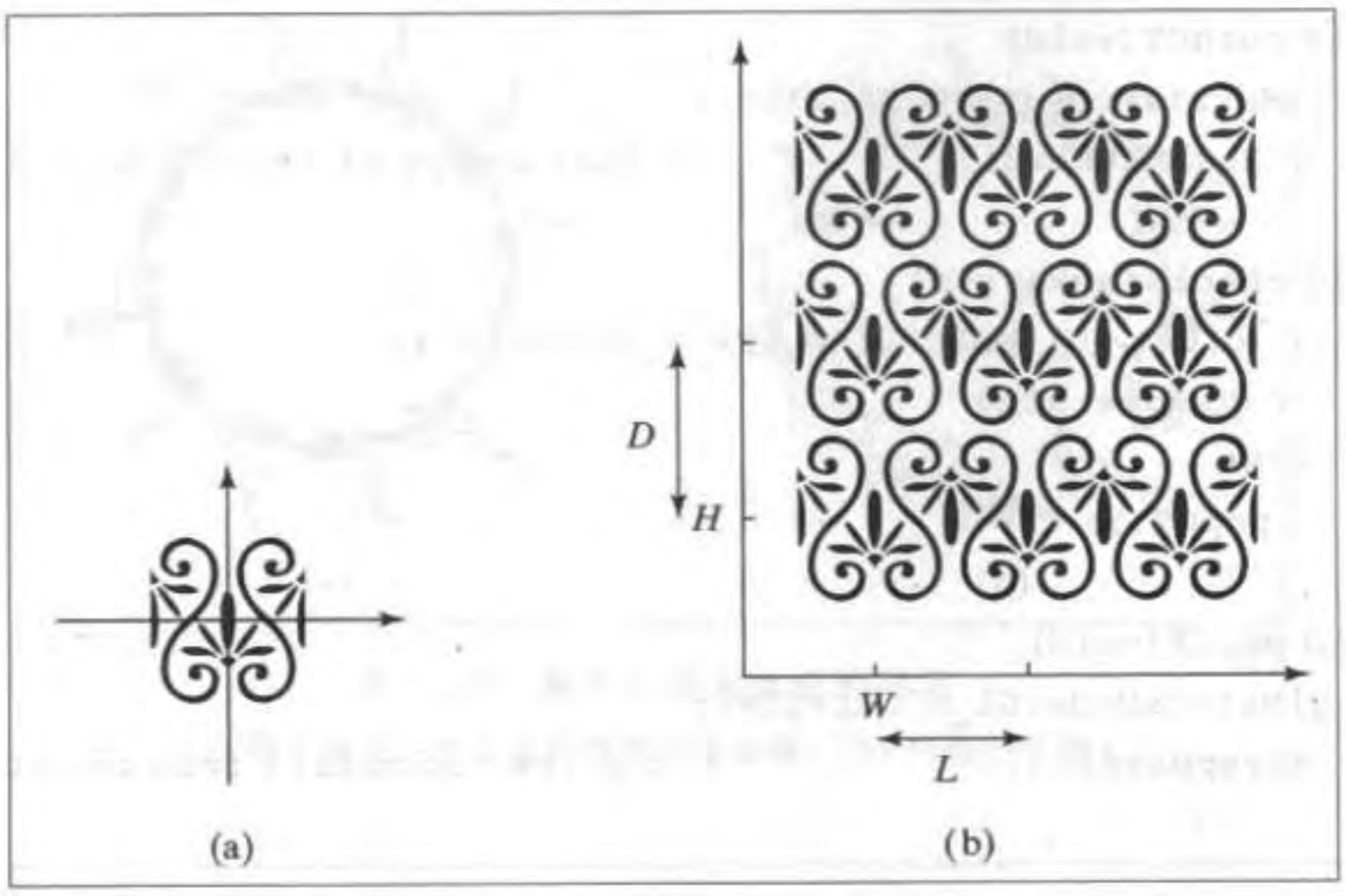


图 5.41 用基本图案制作拼贴

```
pushCT();                          //so we can return here
translate2D(W, H);                 //position for the first motif
for (row=0, row<3, row++)          //draw each row
{
        pushCT();
        for (col=0, col<4, rol++) //draw the next set of columns
        {
            motif();
            translate2D(L, 0);     //move to the right
        }
        popCT();                   //back to the start of this row
        translate2D(0, D);         //move up to the next row
}
popCT;                             //back to where we started
```

图 5.42 绘制拼贴画的代码

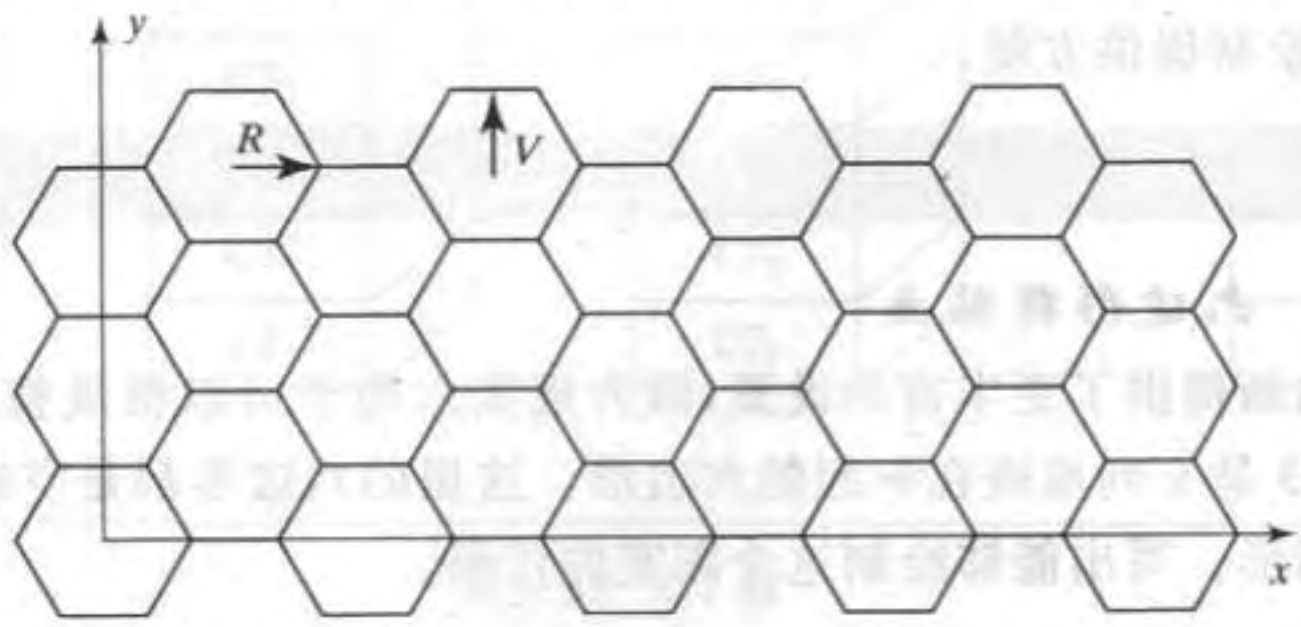


图 5.43 六边形的拼贴画

练习 5.5.3

重复前面的练习，绘制一个由紧密排列的正方形而不是六边形组成的图案，使用用户提供的行数和列数，同时在每个正方形中绘制简单的图案。

5.6　使用 OpenGL 绘制三维场景

我们在第 3 章中介绍了二维物体的绘制。这些函数创建一个窗口和视口，然后使用 moveTo()和 lineTo()绘制直线。到现在为止，本章已经引入了 *CT* 的概念，同时提供了函数进行二维的旋转、缩放和平移。这些二维变换实际上仅仅是三维变换的特殊形式：它们忽略了第三维分量。

在这一节，我们将检验三维变换是如何在 OpenGL 程序中被使用的。主要强调的是在三维场景中如何把物体变换到所需的位置和朝向。这些工作全部使用矩阵完成，这一点是不足为奇的，同时 OpenGL 也提供了函数用来创建和使用所需要的矩阵。并且 OpenGL 还对矩阵栈进行管理，这样一来就很容易完成如下操作：对一个物体设置变换，然后再恢复到之前的变换，以便为下一次变换做准备。

编写程序使用三维变换来绘制不同的场景，是一件很不错的事情。利用这样的程序有助于提高你对各种三维变换的直观想象能力(这些涉及到旋转和平移的三维变换向来都是很难可视化的，因此读者可以从 OpenGL 中获取所需的帮助)。

程序员使用 OpenGL 可以很方便地设置一个摄像机，从特殊视角获取场景截图。伴随着摄像机的建立也生成了一个矩阵，在第 7 章我们将学习有关的细节。这里我们仅仅使用 OpenGL 工具去设置一个合理的摄像机，这样精力就可以集中在物体变换上。虽然我们还未确切了解一个工具的运行机制就开始使用它，但是这样是有很大好处的：调用一些简单的 OpenGL 函数可以帮助你绘制出令人印象深刻的三维场景图像。

5.6.1　观察过程和图形绘制管道概述

到目前为止我们所有的二维绘制实际上都在使用三维观察过程的一个特例，它是基于简单的平行投影的。我们一直使用图 5.44 介绍的摄像机。摄像机有一个镜头，连同另一个特殊的点叫做视点(因为程序员对场景很熟悉，所以可以很容易地把一个特殊点设定为视点)。请记住，尽管我们画出了眼睛好像它是属于某个人或动物的，但这纯粹是为了有助于直观想象。事实上，眼睛就是三维场景中的一点。我们说它朝向某一特殊方向；这也是为了

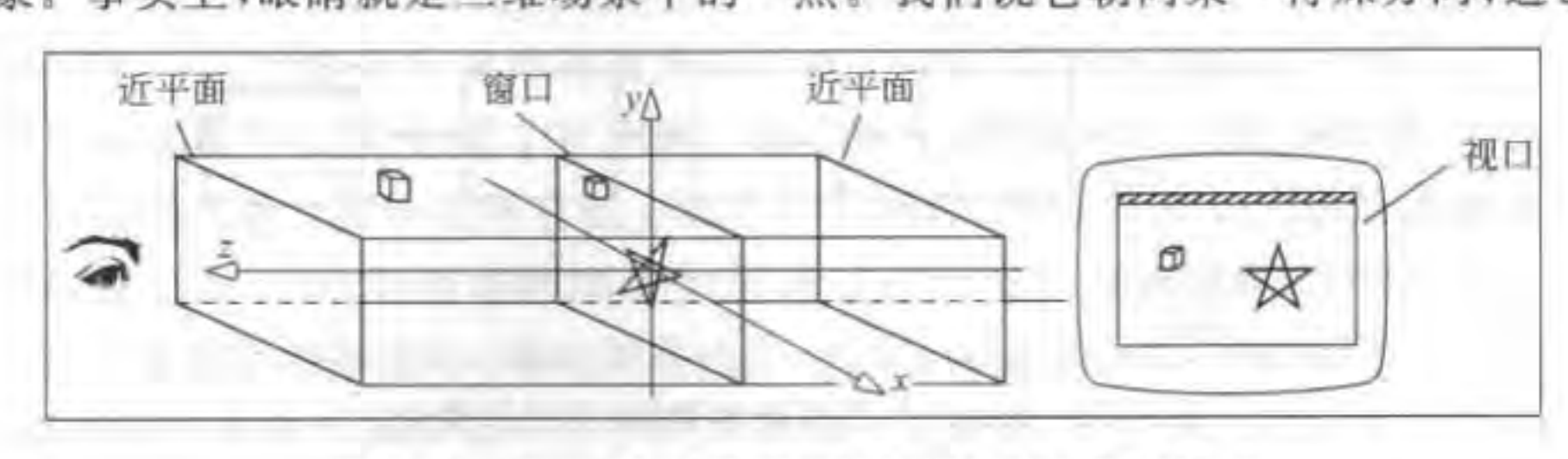


图 5.44　OpenGL 中用于二维绘图的简单观察过程

帮助读者更好的理解观察过程。摄像机的视景体(第 7 章会深入讨论)是一个平行六面体,它的四个侧面是由窗口的边界决定的,它的其他两个面是近平面和远平面。在视景体中的点沿着与 z 轴平行的直线投射到窗口上:3D 点(x_1,y_1,z_1)投射到点(x_1,y_1,0)。在视景体外的点就被裁掉了。一个独立的**视口变换**把投射的点从窗口映射到显示设备的视口上。

现在我们转向三维图形,并将三维物体放进一个场景。对于这些例子,我们将继续使用平行投影(第 7 章介绍更加真实和熟悉的透视投影,在这种投影中,远的物体看起来比近的物体要小)。我们使用和图 5.44 相同的摄像机,但是允许它的位置和朝向更加灵活,以便于对场景生成更好的视图。

图 5.45 展示了这样一个在某场景中的摄像机;这个场景由方块组成,有些方块处于视景体之外,同时展示了被这个摄像机拍摄到的图像。

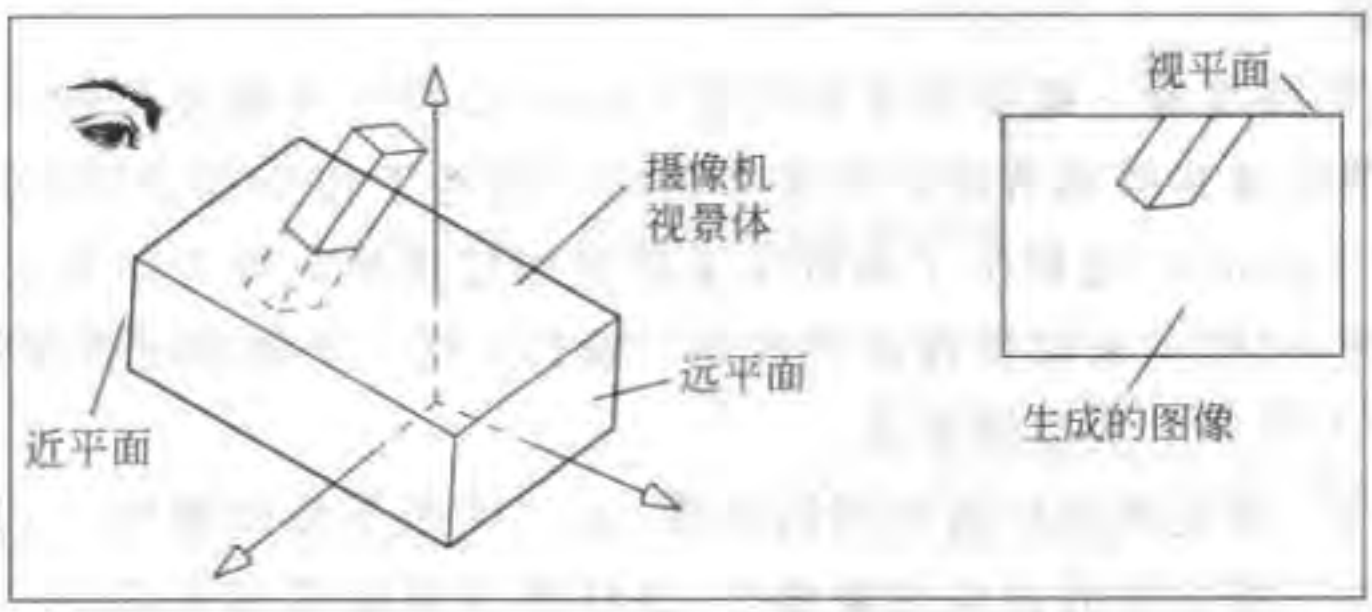


图 5.45 生成某个场景的平行视图的摄像机

我们在前面几节看到,OpenGL 提供了 3 个函数用于对一个形状进行模型变换。图 5.45中的积木块实际上就是一个正方体经过拉伸、旋转和平移得到的。OpenGL 还提供了函数用于在场景中定义视景体和它的位置。

由 OpenGL 实现的图形绘制管道是通过矩阵变换完成它的主要工作的,因此我们首先考察一下管线中的每个矩阵。这里我们重点看看矩阵操作的基本思想,第 7 章我们再详细讨论。图 5.46 展示了最简单的图形管线。通过调用 glVertex3d(x, y, z)使得物体的每一个顶点通过这个管线。顶点和下面所示的各种矩阵相乘,如有必要,该点会被裁剪掉,如果这个点未被裁剪它最终会被映射到视口上。每个顶点会通过 3 个矩阵:

- 模型视点矩阵
- 投影矩阵
- 视口矩阵

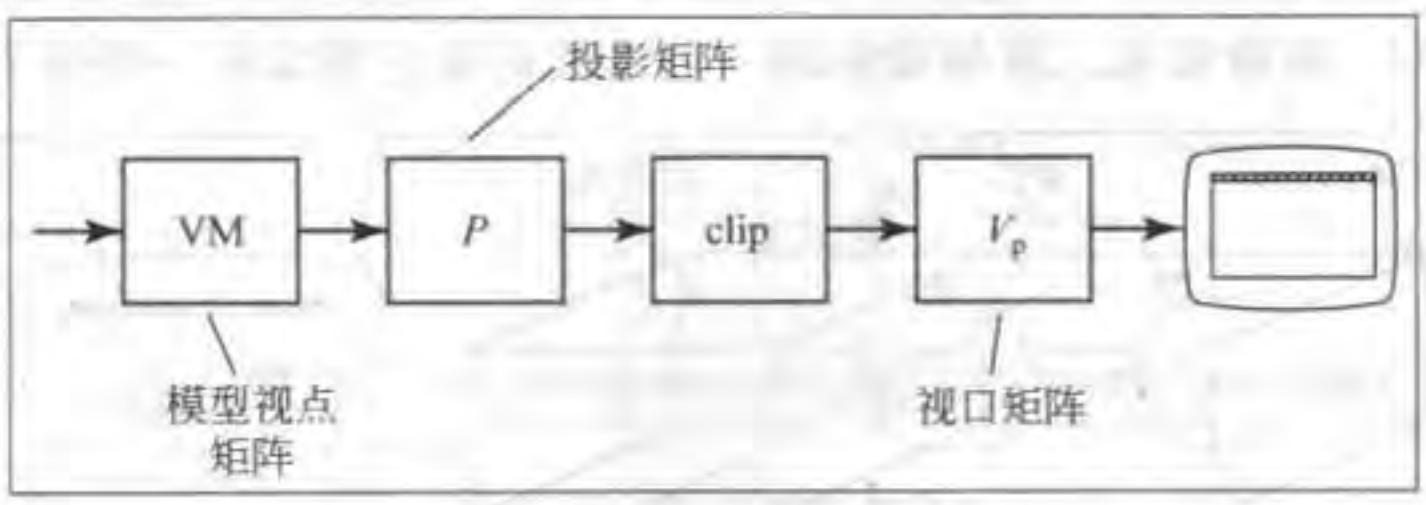


图 5.46 OpenGL 图形绘制管道(已稍作简化)

模型视点矩阵基本上就是我们所说的 CT。它结合了两种效果：对物体的一系列模型变换，和对空间中摄像机位置和朝向的变换（这就是模型视点矩阵这个特定称谓的由来）。尽管实际上在管线中它是一个单独的矩阵，但是很容易把它想成两个矩阵的乘积——一个模型矩阵 M 和一个视点矩阵 V。模型矩阵首先被作用，然后是视点矩阵，所以模型视点矩阵实际上是 VM（为什么？）。

对于图 5.45 所展示的一个摄像机朝下观察有一个积木块的场景，图 5.47 说明了其中的 M 和 V 矩阵是怎么工作的。图 5.47(a)显示了一个位于原点的单位立方体。一个 M 定义的模型变换对这个立方体进行缩放，旋转和平移，图 5.47(b)所示。图 5.47(b)也说明了摄像机视景体的相对位置。

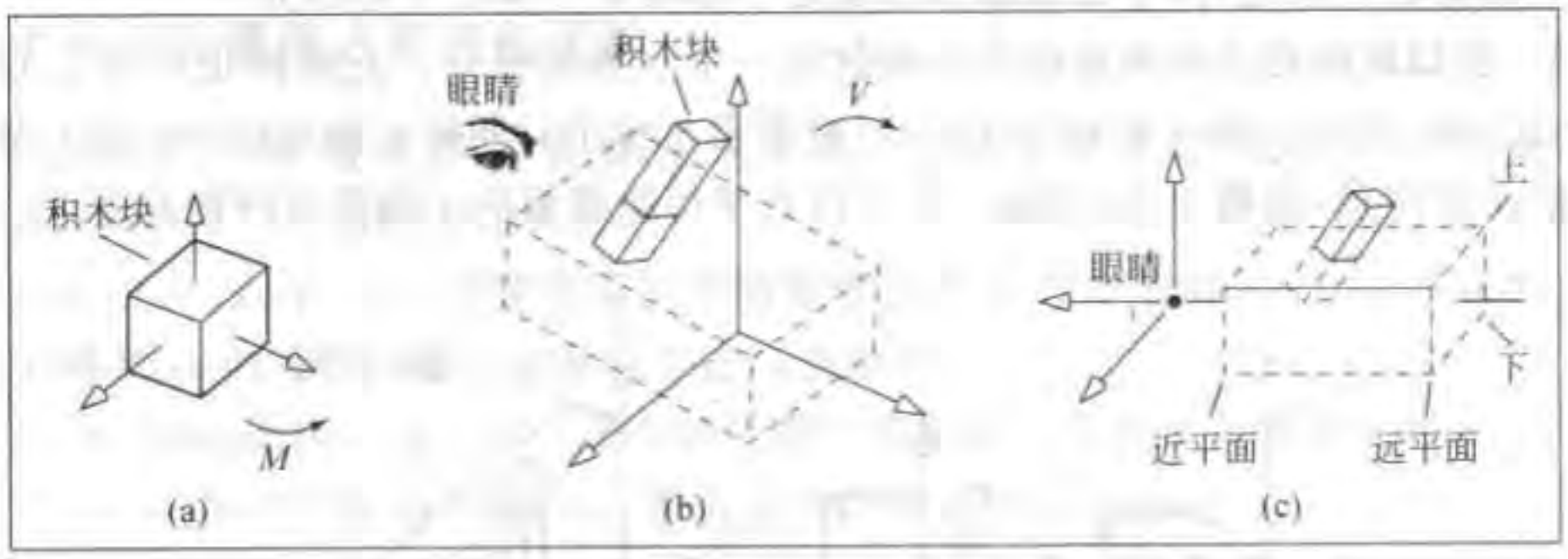


图 5.47 图形绘制管道中模型视点矩阵的效果

(a) 变换前；(b) 模型变换后；(c) 模型视点变换后

现在使用 V 矩阵把积木块旋转平移到一个新的位置。然后特定的变换把摄像机从它原来的位置变换到一个一般的位置，这个一般的位置使得摄像机的视点处在原点，并且视景体和 z 轴对齐，如图 5.47(c)所示。现在积木块的顶点已经就位（也就是说他们的坐标有恰当的值），可以把它们投影到一个平面上比如近平面，得到恰当的值用来显示投影图像。所以实际上 V 矩阵把场景中的顶点坐标变换成摄像机坐标系中的坐标（摄像机坐标有时也叫做视点坐标）。为了告诉 OpenGL 我们要操作模型视点矩阵，必须调用函数 glMatrixMode(GL_MODELVIEW)。

在摄像机坐标系中，视景体的边界平行于 x、y 和 z 轴。如图所示视景体在 x 方向从右到左扩展，在 y 方向从上到下扩展，在 z 方向从近到远扩展。当原立方体的所有顶点通过整个模型视点矩阵变换之后，它的位置就如图 5.47(c)所示的那样。

投影矩阵以一种特殊的方式旋转平移每一个顶点，使得在视景体中的所有点将来出现在一个每一维都从 -1 到 1 的标准立方体中（如果使用透视投影，这个矩阵的作用更大，第 7 章会详细讲述）。这个矩阵有效地把视景体压缩为中心位于原点的立方体。这个立方体的各个面可以有效裁剪物体，如第 7 章所讲。以这种方式缩放方块会使其严重失真，但是视口变换会抵消这种失真。投影矩阵也把 z 轴进行了翻转，所以增加 z 分量的值就表示增加点到视点的深度。图 5.48 显示了积木块如何通过这个变换变到不同的积木块。为了告诉 OpenGL 我们要操作投影矩阵，必须调用函数 glMatrixMode(GL_PROJECTION)。

（注意观察体从来不需要像一个平行六面体一样创建和建模。仅仅当一个特殊的形状被投影变换到一个标准立方体时定义）。

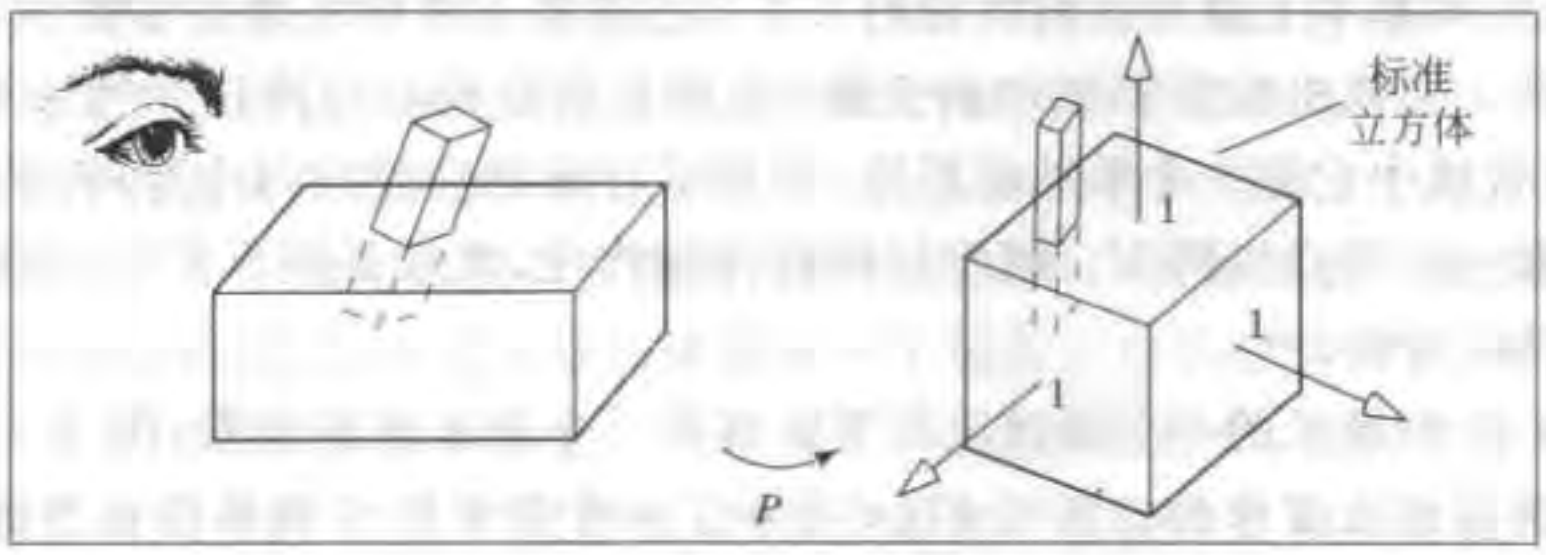


图 5.48 投影矩阵的效果

现在裁剪会自动进行,它会消除积木块处于标准立方体之外的部分。

最后,**视口矩阵**把方块剩余的部分映射到一个三维的视口。此矩阵把标准立方体映射为一个立方体,它的 x 和 y 坐标等于视口(在屏幕坐标中),同时 z 轴坐标从 0 到 1 并且保持每一点的深度信息,如图 5.49 所示。深度信息使得隐藏面的去除变得特别有效。

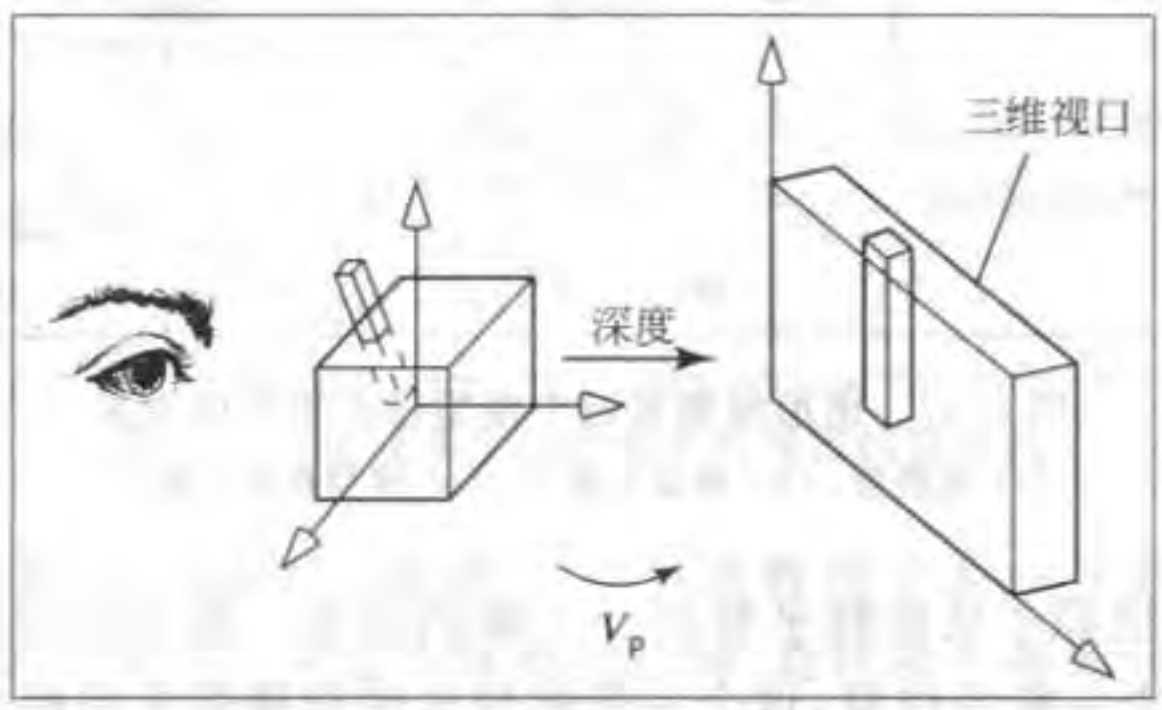


图 5.49 视口矩阵的效果

模型视点变换总结

这个概述阐述了 OpenGL 绘制管道是如何运作的,展示出变换的中心地位。对象的每一个顶点都要经过一系列变换,从世界坐标系变换到视点坐标系,然后再到一个为裁剪设计的过渡坐标系,最后到一个适宜于正确显示的坐标系。由于矩阵乘法的使用,每一次的变换都很高效。

注意:绘制管道的一些重要细节在这个概述中没有涉及。当使用透视变换时,我们需要包括一个透视除法,它是利用了齐次坐标的有关性质。当然很多有趣的细节都隐藏在绘制一个图像的最后几步中,比如计算顶点之间像素的颜色值。

5.6.2 OpenGL 中的建模和视点工具

现在我们介绍一些 OpenGL 的函数,用来建模和设置、使用摄像机。已经证明,使用类似在航空技术中出现的概念可以提高你对摄像机定位和朝向的直观印象。在那里不再使用摄像机,而是使用一个飞行器让它飞过一些点并朝向特定的方向来抓取图像。图 5.51 展示了一个这样的飞行器,它可以帮助程序员想象摄像机定位和朝向的样子。因为描述飞行器

需要使用很多参数，而我们要避免使用过多的参数；摄像机也是这样的。因此我们再介绍一个不同的设备，它在电视工业领域已经很熟悉：jib 摄像机。图 5.50 展示了一个 jib 摄像机，它吊在一个很长的机器的一端；这个机器由一个动力的三脚架带动，三脚架可以被摄影师滑过整个场景，摄影师在三脚架顶上进行操控，不仅可以控制三脚架还可以控制摄像机的精确位置。当三脚架滑过场景时摄影师就在三脚架的顶部 jib 摄像机的周围，摇上摇下拍摄想要的场景。

图 5.50 用于电视制作的 jib 摄像机

三个用于设置视点变换的函数

下面的函数一般用来修改模型视点矩阵，因此如果要使用它们就需要先调用 glMatrixMode(GL_MODELVIEW)。

- glScaled(sx,sy,sz)：用矩阵后乘当前矩阵执行 x 方向缩放 sx，y 方向缩放 sy，z 方向缩放 sz 的操作。最后结果保存进当前矩阵。
- glTranslated(dx, dy, dz)：用矩阵后乘当前矩阵执行执行 x 方向平移 dx，y 方向平移 dy，z 方向平移 dz 的操作。最后结果保存进当前矩阵。
- glRotated(angle, ux,uy, uz)：用矩阵后乘当前矩阵执行旋转 angle 角度，围绕的轴通过原点和点(ux, uy, uz)。最后结果保存进当前矩阵。

在 OpenGL 中设置摄像机(平行投影)

glOrtho(left, right, bottom, top, near, far)创建了一个平行六面体的视景体，它在 x 方向上从 left 到 right 延伸，在 y 方向上从 bottom 到 top 延伸，在 z 方向上从 near 到 far 延伸(因为这个操作是在眼睛坐标系下进行的，所以摄像机的眼睛在原点，朝向 z 轴负方向)。这个函数创建了一个矩阵，并且后乘当前矩阵。(我们在第 7 章详细介绍这个矩阵含有哪些值)

使用下面的代码设置投影矩阵：

```
glMatrixMode(GL_PROJECTION);                    //设置投影矩阵为当前矩阵
glLoadIdentity();                               //设置为单位阵
glOrtho(left,right,bottom,top,near,far);        //乘以新矩阵
```

因为默认的摄像机安放在原点，朝向 z 轴负方向，当 near 的值为 2 时，就是说把近平面放在 $z=-2$，即眼睛前面 2 个单位。类似地，当 far 的值为 20，把远平面放在在眼睛前 20 个单位。

摄像机的定位和瞄准

OpenGL 提供了一个函数，可以很容易地设置一个基本摄像机的姿态：

```
gluLookAt( eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y, up.z);
```

这个函数创建视点矩阵，然后后乘当前矩阵。它的最终目的是建立一个坐标系统——也就是计算 3 个相互正交的单位向量：**u**、**v** 和 **n**. **n** 就是视点(眼睛位置)——观察点(观察的

位置)的归一向量;它由飞行器的喷气入口指向排气口。我们把向量 **n** 叫做纵向轴,把通过飞行器两个机翼的轴称为横向轴。这些向量所基于的信息必须简单合理,以方便程序员将它们输入程序。

这个函数以摄像机视点的位置 eye 和注视的点 look 为参数。它还有一个近似的向上的 up 向量作为参数。前两个很简单,因为程序员知道他感兴趣的场景处于那个位置,也可以直接地为摄像机开始的位置选取合理的 eye 和 look 的值。图 5.51 展示了一架飞机相对于世界的朝向,其中要用到 u、v 和 n 向量。航空术语:倾斜度、翻转度和偏离度用来描述飞机的方向。

- 倾斜度——纵向轴和世界坐标水平方向的夹角(图 5.51(a))。
- 翻转度——横向轴和世界坐标的夹角(图 5.51(b))。
- 偏离度——纵向轴和世界坐标中某一基本"罗盘"方向的夹角,纵向轴引起飞机飞行方向的改变(图 5.51(c))。

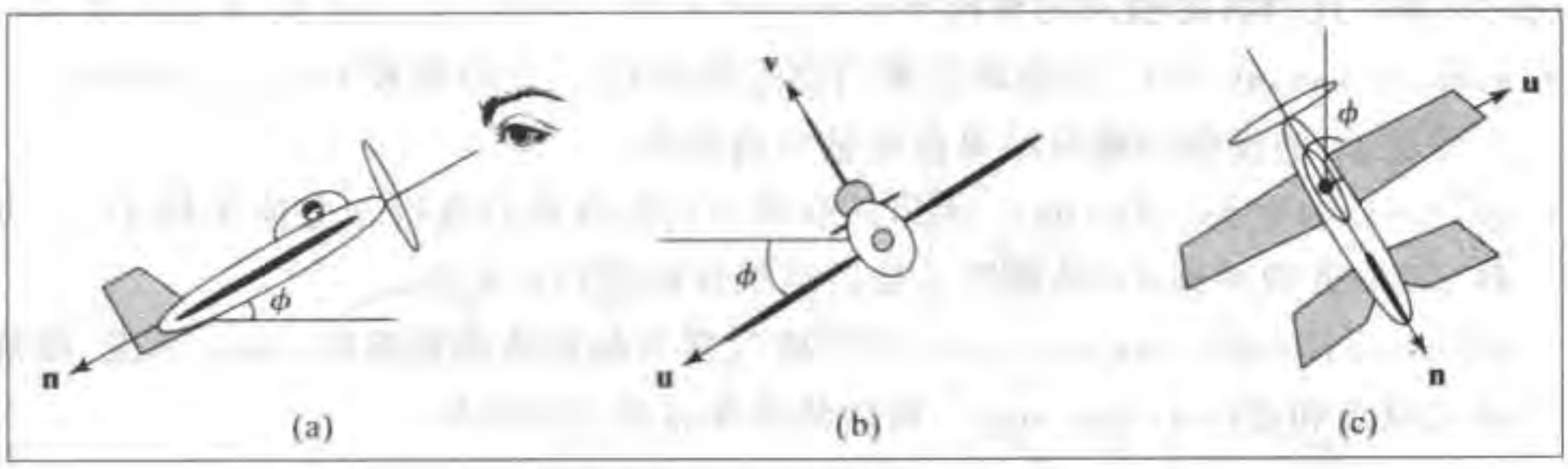


图 5.51 飞机旋转的描述

(a)倾斜;(b)翻转;(c)偏离

为 **up** 选择一个方向很困难。当代码已经被写好后,程序员也许不知道程序中飞机的方向,因为飞机可能已经飞了几圈并翻了几次跟头。初次猜测 up_{guess} 通常设置为(0, 1, 0);也就是 up 和 y 轴平行。另外给定飞机的任何方向,比如向量 $\boldsymbol{n}$ 从飞机喷气入口到排气口,那么我们必须能够计算出两个相互正交向量 **u** 和 **v**,用来构建飞机的坐标系统(如图 5.51)。对于一个飞机,把 **n** 想象成是从飞机入口(飞机前面)指向排气管。如图所示,每次旋转都经过角度 ϕ。图中展示了每次旋转都经过角度 ϕ。

仅仅知道 up 向量和 **n** 向量正交程序员很难对 up 向量进行估算。**n** 向量就是一个 eye-look 向量的简单归一化。我们下面来看看怎样实现一个真实的 up 向量 up_{true}。我们首先把它设置成和 y 轴平行:$up_{guess}=(0, 1, 0)$。

我们想要函数 gluLookAt(…)来设置矩阵 VM 中的 V 部分因此它在增加其他变换之前就被调用了,因为模型变换后会乘以模型视点矩阵。结果就是如下使用 glulookAt():

```
glMatrixMode(GL_MODELVIEW);                          //设置当前模型视点矩阵
glLoadIdentity();                                    //以单位阵开始
glulookAt(eye.x,eye.y,eye.z,look.x,look.y,look.z,0,1,0);   //"看向"点近似 up_true
```

例 5.6.1 设置一个传统的摄像机

摄像机通常从某一个较近的位置俯视整个场景。图 5.52 所示的摄像机的视点在

eye=(4, 4, 4),朝向原点方向盯着 look = (0, 1, 0)。up_{guess}方向被初始设置为 up_{guess} = (0, 1, 0)。假设我们同样需要观察体宽为 6.4,高为 4.8(因此这个比例为 640/480)。同时设置 near 为 1,far 为 50。摄像机就用下面的代码建立:

```
glMatrixMode(GL_PROJECTION);                    //设置观察体
glLoadIdentity();
glOrtho(-3.2,3.2,-2.4,2.4,1,50);
glMatrixMode(GL_MODELVIEW);                     //设置安放摄像机
glLoadIdentity();
gluLookAt(4,4,4,0,1,0,0,1,0);
```

这个练习展示了模型视点矩阵中对应位置的特殊值。因为视点在原点上方,朝向 z 轴负方向,因此 up_{true}不会平行于 y 轴,但是相反标记为向前。可视化飞机直接朝下的情况,这时 up_{true}平行于地面。在第 7 章我们讨论如何操作这个函数建立灵活的设计摄像机的工具。如果好奇想知道 gluLookAt()的参数在模型视点矩阵中如何放置,请参见练习。

练习

练习 5.6.1 gluLookAt 干了些什么?

我们知道 gluLookAt()建立了一个矩阵,把世界坐标转换成视点坐标。图 5.53 展示了一个摄像机作为一个悬挂在真实世界中的坐标系统,它的眼睛在原点处,方向由三个相互正交的向量 **u**、**v** 和 **n** 决定。眼睛朝向 −**n** 方向。gluLookAt()使用参数 eye、look 和 up_{guess} 去创建 **u**、**v** 和 **n**。具体等式见第 7 章。

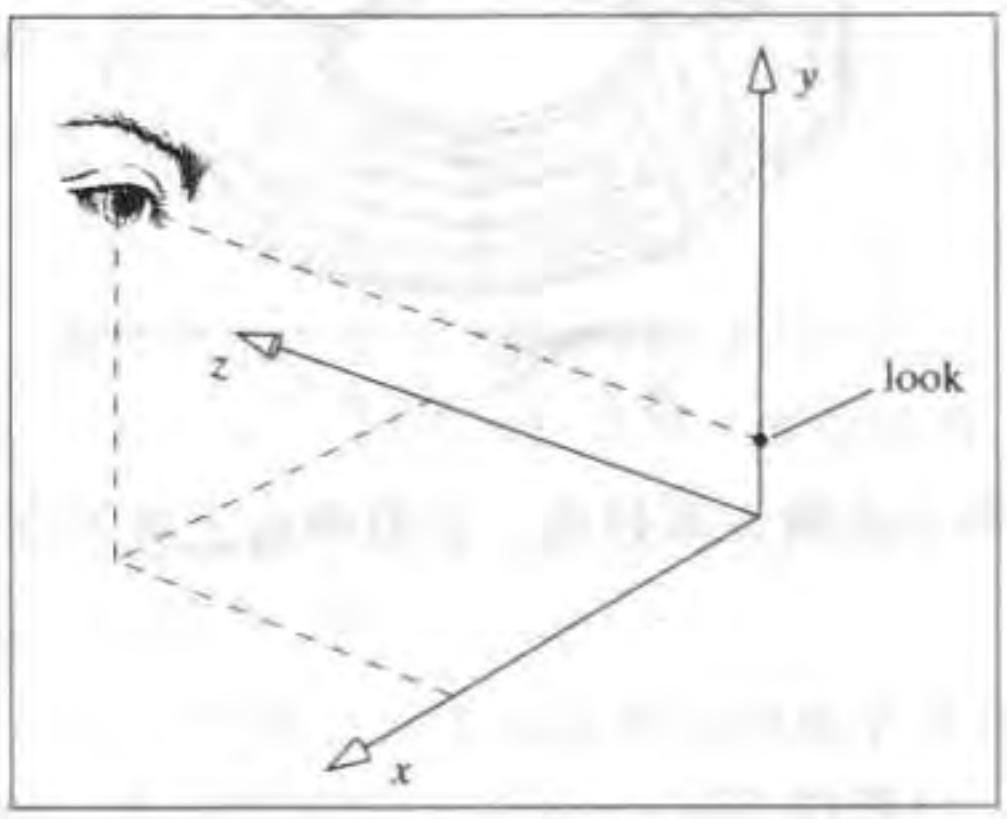


图 5.52 用 gluLookAt 设置一个摄像机

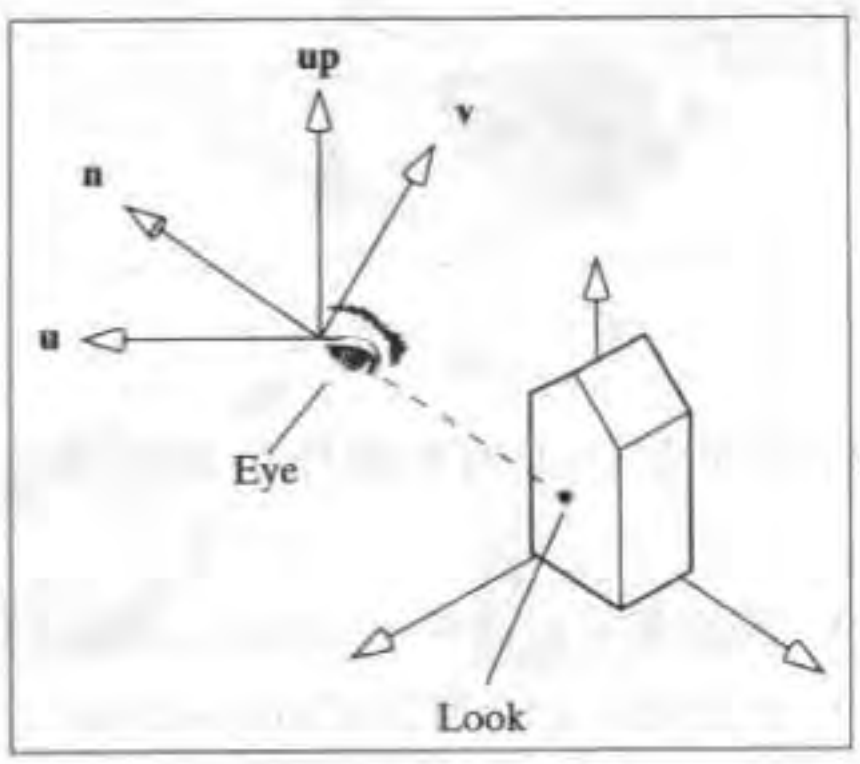


图 5.53 在摄像机坐标中转换世界坐标系统

5.6.3 用 OpenGL 绘制基本形状

下一章将会看到如何创建我们自己的三维物体,不过这里为了实验摄像机的设置和使用我们需要绘制一些三维物体。GLUT 提供了一些现成的物体(如图 5.54)。包括了一个球,一个圆锥,一个圆环,五个柏拉图立体(见第 6 章讨论),和著名的茶壶。每一个都是一个

网格模型(就好像是很多个端与端连接的线的集合),也可以作为实心物体,还能为每个面进行着色。

- **cube**: glutWireCube(GLdouble size) **Each side is of length** size
- **sphere**: glutWireSphere(GLdouble radius, GLint nSlices, GLint nStacks)
- **torus**: glutWireTorus(GLdouble inRad, GLdouble outRad, GLint nSlices, GLint nStacks)
- **teapot**: glutWireTeapot(GLdouble size)

图 5.54 GLUT 的一些物体

还有函数 glutSolidCube(),glutSolidSphere()等等,后面我们会用到。圆环的形状由它的内直径 inRad 和外直径 outRad 决定。球和圆环都由多边形面去近似的。图 5.55 就是沿着 z 轴方向,x 轴向右,y 轴向上。想象一个比萨或者甜甜圈;nSlides 是你围绕着 z 轴切割比萨的块数。图中 nSlides 为 8。

如果你旋转圆环,使得 z 轴朝上,每个边都是一个近似的圆,形式是一定数量的线段组成的多边形。这个数量叫做 nStacks,图 5.56 中的 nStacks 是 16。

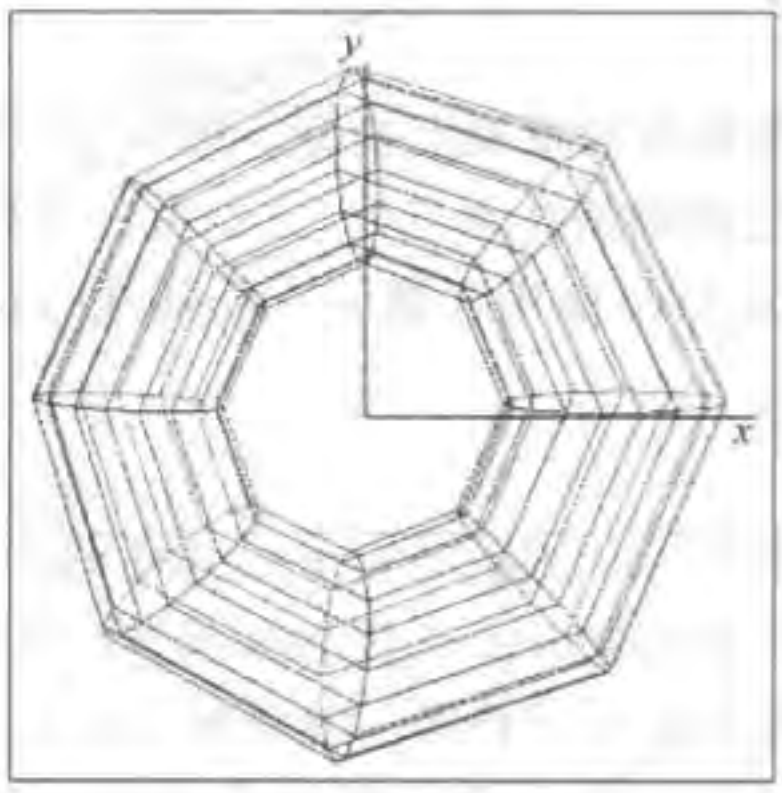


图 5.55 一个花环

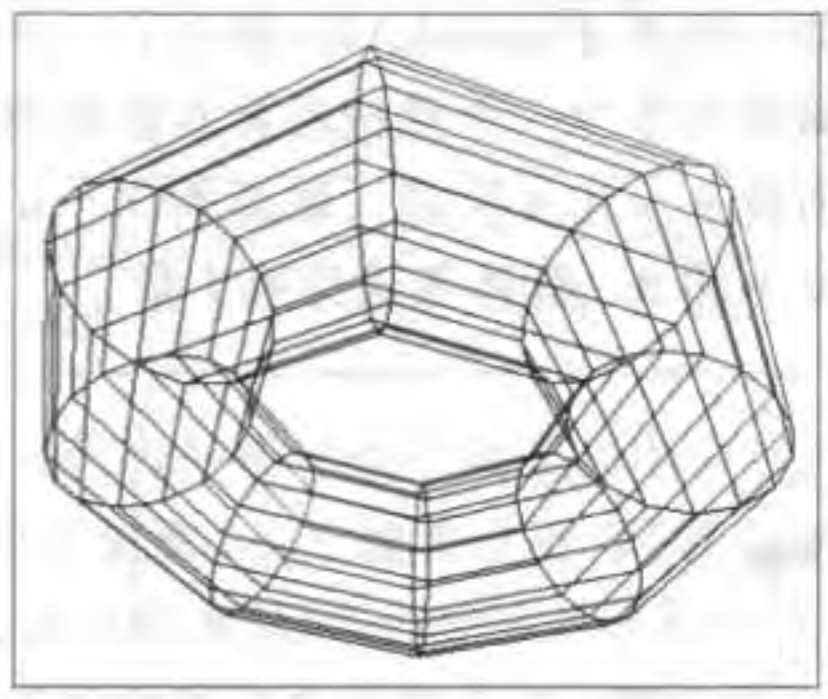


图 5.56 一个花环

GLUT 提供了四个额外的柏拉图实体,这些会在第 6 章讨论。它们都由下面的方程产生:

- 四面体:glutWireTetrahedron()
- 八面体:glutWireOctahedron()
- 十二面体:glutWireDodecahedron()
- 二十面体:glutWireIcosahedron()

其他由 GLUT 和 GLU 生成的也很有趣:

- 圆锥:glutWireCone(GLdouble baseRad, GLdouble height, Glint nSlides, Glint nStaks)
- 截锥:gluCylinder(GLUquadricObj * qobj, GLdouble baseRad, GLdouble topRad, GLdouble height, Glint nSlides, GLint nStacks)

圆锥和截锥的轴就是 z 轴。它们的基点位于 $z=0$ 的平面,它们沿着 z 轴扩展到 $z=$

height。在 $z=0$ 处圆锥和截锥的半径有 baseRad 给定。截锥在 $z=$ height 处的半径由 topRad 给定。和圆环、球一样，nSlides 关于 z 轴细分圆锥和截锥，nStacks 个多边形沿着这些物体的 z 轴。

实际上**截锥**是一组形状，有 topRad 的值区分。当 topRad 等于 1 时，没有锥形，这是基本的圆柱。当 topRad 为 0 时，截锥就是圆锥体。

注意在 OpenGL 中绘制截锥需要一些额外的工作，因为它是二次曲面的特殊情况，如第 6 章所述。为了绘制它必须(1)定义一个新的二次曲面对象；(2)设置绘制类型(GL_LINE 绘制网格，GL_FILL 绘制实体)；(3) 绘制物体：

```
GLUquadricObj * qobj=gluNewQuadric();                    //生成一个二次对象
gluQuadricDrawStyle(qobj, GLU_LINE);                     //设置为绘制网格
gluCylider(qobj, baseRad, topRad, nSlides, nStacks);  //绘制圆柱体
```

在下面的两个例子中，我们将用到其中的一些形状，这两个例子主要关注使用仿射变换进行建模和观察三维场景。同时给出了绘制每一个场景的程序。如果你直接输入这些程序，生成图像，然后根据效果调整各个参数，你就可以得到很好的直观认识。

例 5.6.2 由网格物体组成的场景

图 5.57 展示了一个场景，这个场景中有几个物体位于一个单位立方体的角上。这个立方体有一个角在原点处。7 个物体出现在立方体不同的角上，所有的都是网格。

摄像机给出的观察体在 y 方向上扩展到从 $-2\sim2$，具有比例为 640/480。它的近平面在 $N=0.1$，原平面在 $F=100$。这个是用下面调用完成的：

```
glOrtho(-2.0 * aspect, 2.0 * aspect,
-2.0, 2.0, 0.1, 100);
```

摄像机被安放在 eye=(2, 2, 2)，look=(0, 0, 0)，up=(0, 1, 0)，是用下面的调用：

```
gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0,
0.0, 1.0, 0.0);
```

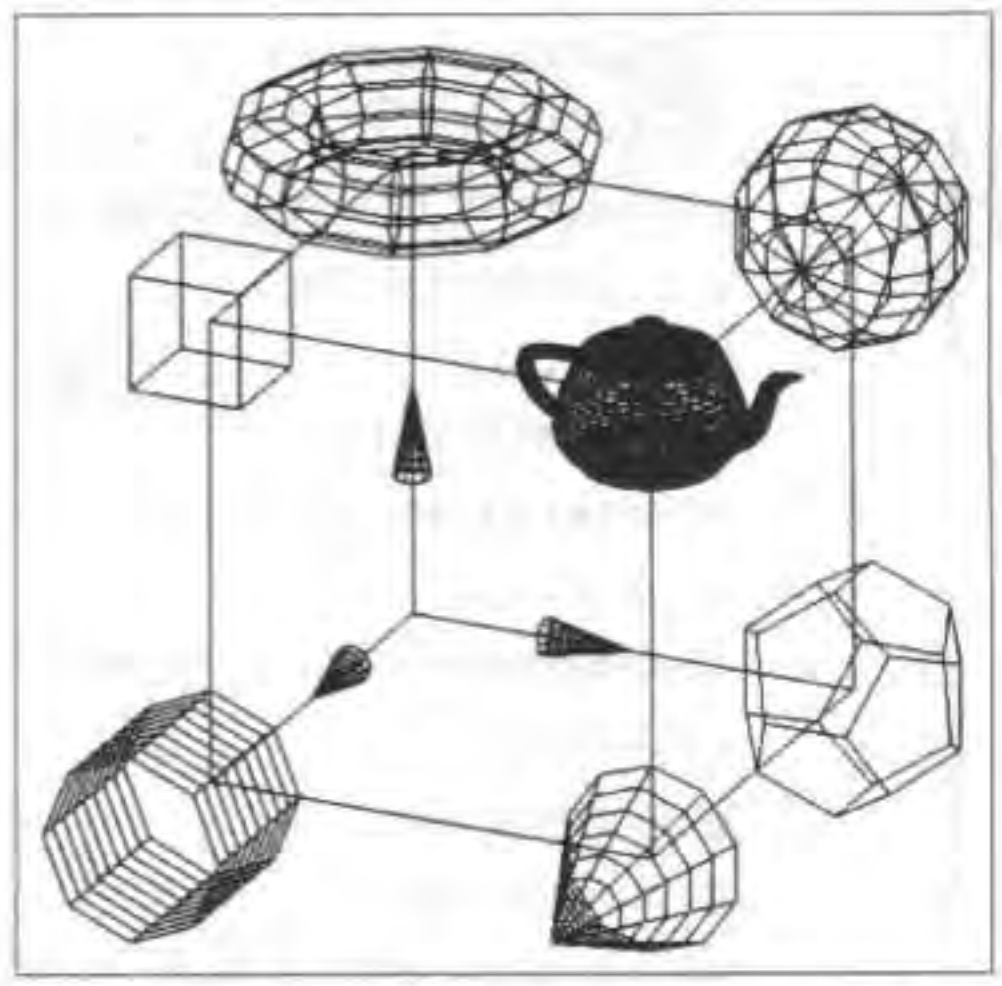


图 5.57 几个物体位于一个单位立方体的角

图 5.58 给出了生成这幅图的整个程序。main()函数初始化一个 640×480 的窗口，设置视口和背景颜色，然后定义了displayWire()函数成为完成绘制调用的函数。在 displayWire()中首先创建摄像机的形状和位置。然后依次绘出每个物体。大部分物体需要它们自己的模型矩阵，以便如期地旋转和放置它们。在每一个物体的变换创立之前，要调用 glPushMatrix()保存当前矩阵，当物体被绘制完成后，调用 glPopMatrix()恢复之前的变换矩阵。这样绘制每一个物体的代码就处于 glPushMatrix()和 glPopMatrix()之间。仔细检查每一个变换，看看每个物体是否放置在正确的位置上。

```
# include<windows.h>//suitable when using Windows 95/98/NT/2000/XP
# include<gl/Gl.h>
# include<gl/Glu.h>
# include<gl/glut.h>
//<<<<<<<<<<<<<<<<<<<<<axis >>>>>>>>>>>>>>>>
void axis(double length)
{ // draw a z-axis, with cone at end
    glPushMatrix();
    glBegin(GL_LINES);
    glVertex3d(0, 0, 0); glVertex3d(0,0,length); // along the z-axis
    glEnd();
    glTranslated(0, 0,length -0.2);
    glutWireCone(0.04, 0.2, 12, 9);
    glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<displayWire >>>>>>>>>>>>>>>>>>>>>>
void displayWire(void)
{
    glMatrixMode(GL_PROJECTION);     // set the view volume shape
    glLoadIdentity();
    glOrtho(-2.0 * 64/48.0, 2.0 * 64/48.0, -2.0, 2.0, 0.1, 100);
    glMatrixMode(GL_MODELVIEW);      //position and aim the camera
    glLoadIdentity();
    gluLookAt(2.0, 2.0, 2.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);    //clear the screen
    glColor3d(0,0,0);                // draw black lines
    axis(0.5);                       // x-axis
    glPushMatrix();
    glRotated(90, 0,1.0, 0);
    axis(0.5);                       // y-axis
    glRotated(-90.0, 1, 0, 0);
    axis(0.5);                       // z-axis
    glPopMatrix();
    glPushMatrix();
    glTranslated(0.5, 0.5, 0.5);     // big cube at (0.5, 0.5, 0.5)
    glutWireCube(1.0);
    glPopMatrix();
    glPushMatrix();
    glTranslated(1.0,1.0,0);         // sphere at (1,1,0)
    glutWireSphere(0.25, 10, 8);
    glPopMatrix();
```

图 5.58 使用 OpenGL 绘制图 5.57 的完整程序

```
    glPushMatrix();
    glTranslated(1.0,0,1.0);                    // cone at (1,0,1)
    glutWireCone(0.2, 0.5, 10, 8);
    glPopMatrix();
    glPushMatrix();
    glTranslated(1,1,1);
    glutWireTeapot(0.2);                        // teapot at (1,1,1)
    glPopMatrix();
    glPushMatrix();
    glTranslated(0, 1.0 ,0);                    // torus at (0,1,0)
    glRotated(90.0, 1,0,0);
    glutWireTorus(0.1, 0.3, 10,10);
    glPopMatrix();
    glPushMatrix();
    glTranslated(1.0, 0 ,0);                    // dodecahedron at (1,0,0)
    glScaled(0.15, 0.15, 0.15);
    glutWireDodecahedron();
    glPopMatrix();
    glPushMatrix();
    glTranslated(0, 1.0 ,1.0);                  // small cube at (0,1,1)
    glutWireCube(0.25);
    glPopMatrix();
    glPushMatrix();
    glTranslated(0, 0 ,1.0);                    // cylinder at (0,0,1)
    GLUquadricObj * qobj;
    qobj=gluNewQuadric();
    gluQuadricDrawStyle(qobj,GLU_LINE);
    gluCylinder(qobj, 0.2, 0.2, 0.4, 8,8);
    glPopMatrix();
    glFlush();
}
//<<<<<<<<<<<<<<<<<<<<<main >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char ** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB );
    glutInitWindowSize(640,480);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Transformation Test -wireframes");
    glutDisplayFunc(displayWire);
    glClearColor(1.0f, 1.0f, 1.0f,0.0f);  // background is white
    glViewport(0, 0, 640, 480);
    glutMainLoop();
}
```

图 5.58（续）

图中还展示了使用箭头绘制的 x、y 和 z 三个轴,显示出底层的坐标系可以帮助我们选择观察的方向。因为我们的函数 axis()绘制出 z 轴,要从观察点看到 x 轴很不容易。如果我们事先让 x 轴和 z 轴绕着 y 轴旋转 $90°$,使得旋转后的坐标系统很容易可视化,同时坐标轴也在新位置上进行了重绘。注意,坐标轴的绘制没有放在 glPushMatrix()和 glPopMatrix()之间,因此下一次旋转生成的 y 轴将会代替已经旋转的坐标系统。

注意三维中平行的立方体的边,显示出来也是平行的。这是使用平行投影的结果。这个立方体看起来有些不自然,因为我们习惯于使用透视投影观察世界。第 7 章会看到,如果我们透视投影,这些平行的边看起来就不平行了。

例 5.6.3 着色的三维场景绘制

我们构建一个更加复杂的场景来说明模型变换以后的用途。我们还要展示 OpenGL 如何使用插值着色和适当的消除隐藏面绘制更加真实的实心物体。

图 5.59 展示了一个场景的两个视角。两个视角都是使用摄像机设置 gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0, 1.0, 0.0)。图 5.59(a)使用一个比较大的观察体包括了整个场景;图 5.59(b)使用了一个小的观察体仅仅包含了场景的一小部分,因此产生了一个特写视角。

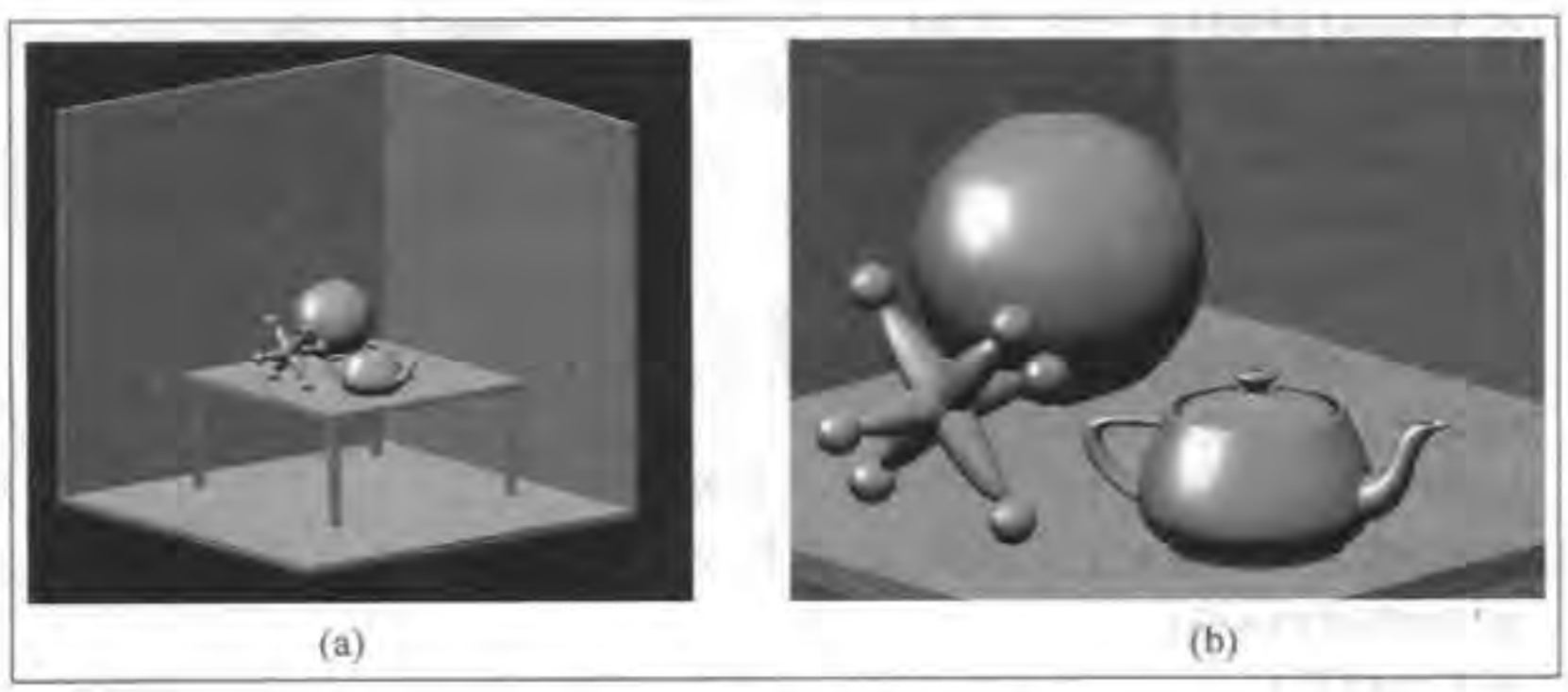


图 5.59 一个简单的三维场景

(a) 使用一个较大的观察体;(b) 使用一个较小的观察体

这个场景包含 3 个物体位于一个角落处的桌子上。每个墙都是立方体压成片,然后移动到位置上(我们会再次看到不自然的平行映射)。小铁叉是由 3 个拉伸的球体以固定的角度外加 6 个小球位于端点处组成。

桌子包含一个桌面和四个腿。这五个块中每一个都是一个立方体经过缩放得到想要的大小和形状。图 5.60 是一个桌子的轮廓。它有四个参数决定每个部分的大小:topWidth、topThick、legLen 和 legThick。函数 tableLeg()画出每条腿,在函数 table()中被调用四次绘制出四条不同位置的腿。tableLeg()使用不同的参数生成不同的模型变换。同样,glPushMatrix()和 glPopMatrix()分别在函数两端来分隔它们的效果。

这个程序的完整代码如图 5.61 所示。注意,每个形状的实心绘制函数比如 glutSolidSphere()会在这里使用,而不是使用它们的线框函数。检查一下变换中每个物体所用的方向和位置是否合适。特别检查小叉子模型。这个例子是用来激发你的兴趣去亲自练习这些变换的。

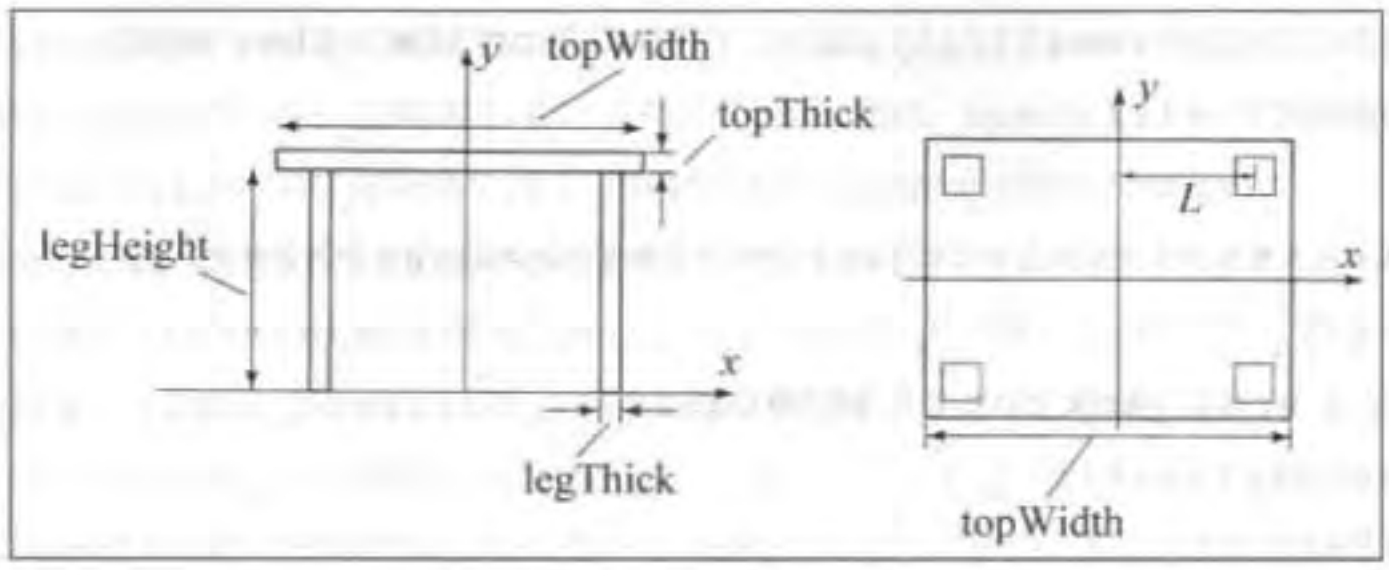


图 5.60 桌子的设计

```
#include<windows.h>
#include<iostream.h>
#include<gl/Gl.h>
#include<gl/Glu.h>
#include<gl/glut.h>
//<<<<<<<<<<<<<<<<<<<wall >>>>>>>>>>>>>>>>>>>
void wall(double thickness)
{ // draw thin wall with top=xz- plane, corner at origin
    glPushMatrix();
    glTranslated(0.5, 0.5* thickness, 0.5);
    glScaled(1.0, thickness, 1.0);
    glutSolidCube(1.0);
    glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<<<<<<tableLeg >>>>>>>>>>>>>>>>>>>>>>>
void tableLeg(double thick, double len)
{
    glPushMatrix();
    glTranslated(0, len/2, 0);
    glScaled(thick, len, thick);
    glutSolidCube(1.0);
    glPopMatrix();
}
//<<<<<<<<<<<<<<<<<<<<<<<jack part >>>>>>>>>>>>>>
void jackPart()
{ // draw one axis of the unit jack-a stretched sphere
    glPushMatrix();
    glScaled(0.2,0.2,1.0);
    glutSolidSphere(1,15,15);
    glPopMatrix();
    glPushMatrix();
    glTranslated(0,0,1.2); // ball on one end
    glutSolidSphere(0.2,15,15);
    glTranslated(0,0, -2.4);
```

图 5.61 程序的完整代码

```
        glutSolidSphere(0.2,15,15); // ball on the other end
        glPopMatrix();
    }
    //<<<<<<<<<<<<<<<<<<<<jack >>>>>>>>>>>>>>>>>>>>>
    void jack()
    { // draw a unit jack out of spheroids
        glPushMatrix();
        jackPart();
        glRotated(90.0, 0, 1, 0);
        jackPart();
        glRotated(90.0, 1,0,0);
        jackPart();
        glPopMatrix();
    }
    //<<<<<<<<<<<<<<<<<<<<<<<table >>>>>>>>>>>>>>>>>>>>>
    void table(double topWid, double topThick, double legThick, double legLen)
    { // draw the table-a top and four legs
        glPushMatrix(); // draw the table top
        glTranslated(0, legLen, 0);
        glScaled(topWid, topThick, topWid);
        glutSolidCube(1.0);
        glPopMatrix();
        double dist=0.95* topWid/2.0-legThick / 2.0;
        glPushMatrix();
        glTranslated(dist, 0, dist);
        tableLeg(legThick, legLen);
        glTranslated(0, 0, -2 * dist);
        tableLeg(legThick, legLen);
        glTranslated(-2 * dist, 0, 2*dist);
        tableLeg(legThick, legLen);
        glTranslated(0, 0, -2*dist);
        tableLeg(legThick, legLen);
        glPopMatrix();
    }
    //<<<<<<<<<<<<<<<<<<<<displaySolid >>>>>>>>>>>>>>>>>>>>>
    void displaySolid(void)
    {
        // set properties of the surface material
        GLfloat mat_ambient[] ={ 0.7f, 0.7f, 0.7f, 1.0f}; // gray
        GLfloat mat_diffuse[] ={0.6f, 0.6f, 0.6f, 1.0f};
        GLfloat mat_specular[] ={1.0f, 1.0f, 1.0f, 1.0f};
        GLfloat mat_shininess[] ={50.0f};
        glMaterialfv(GL_FRONT,GL_AMBIENT,mat_ambient);
```

图 5.61 (续)

```
glMaterialfv(GL_FRONT,GL_DIFFUSE,mat_diffuse);
glMaterialfv(GL_FRONT,GL_SPECULAR,mat_specular);
glMaterialfv(GL_FRONT,GL_SHININESS,mat_shininess);
// set the light source properties
GLfloat lightIntensity[]={0.7f, 0.7f, 0.7f, 1.0f};
GLfloat light_position[]={2.0f, 6.0f, 3.0f, 0.0f};
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glLightfv(GL_LIGHT0, GL_DIFFUSE, lightIntensity);
// set the camera
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
double winHt =1.0;          // half-height of the window
glOrtho(-winHt * 64/48.0, winHt * 64/48.0, -winHt, winHt, 0.1, 100.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0,1.0,0.0);
// start drawing
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT); // clear the screen
glPushMatrix();
glTranslated(0.4, 0.4, 0.6);
glRotated(45,0,0,1);
glScaled(0.08, 0.08, 0.08);
jack(); // draw the jack
glPopMatrix();
glPushMatrix();
glTranslated(0.6, 0.38, 0.5);
glRotated(30,0,1,0);
glutSolidTeapot(0.08);          // draw the teapot
glPopMatrix();
glPushMatrix();
glTranslated(0.25, 0.42, 0.35); // draw the sphere
glutSolidSphere(0.1, 15, 15);
glPopMatrix();
glPushMatrix();
glTranslated(0.4, 0, 0.4);
table(0.6, 0.02, 0.02, 0.3);    // draw the table
glPopMatrix();
wall(0.02);                     // wall #1: in xz-plane
glPushMatrix();
glRotated(90.0, 0.0, 0.0, 1.0);
glPopMatrix();
glPushMatrix();
glRotated(-90.0,1.0, 0.0, 0.0);
```

图 5.61（续）

```
    wall(0.02);                          // wall #3: in xy-plane
    wall(0.02);                          // wall #2: in yz-plane
    glPopMatrix();
    glFlush();
}
//<<<<<<<<<<<<<<<<<<<<main>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char ** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB| GLUT_DEPTH);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(100, 100);
    glutCreateWindow("Shaded example-3D scene");
    glutDisplayFunc(displaySolid);
    glEnable(GL_LIGHTING);               // enable the light source
    glEnable(GL_LIGHT0);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_DEPTH_TEST);             // for hidden surface removal
    glEnable(GL_NORMALIZE);   // normalize vectors for proper shading
    glClearColor(0.1f,0.1f,0.1f,0.0f); // background is light gray
    glViewport(0, 0, 640, 480);
    glutMainLoop();
}
```

图 5.61 (续)

这段代码也展示了很多创建着色图像要做的事情。光源的位置和方向必须初始化,还有某个物体表面的性质,这是用来描述物体如何反光的。因为我们在第 8 章讨论着色过程,这里只是给出各个函数的调用;使用它们将会生成着色效果。

练习

练习 5.6.2 调查 OpenGL 中矩阵的值

通过给模型视点矩阵赋特殊值的方法,测试上面提到的定理。你可以看看 OpenGL 中模型视点矩阵里面到底存储的是什么,首先定义一个数组 GLfloat mat[16],然后使用 glGetFloatv(GL_MODELVIEW_MATRIX, mat),这个函数把模型视点矩阵中的 16 个值复制进 mat 数组里面。M[i][j]被拷进了 mat[4 * j+i],其中 i, j=0, 1,…,3。

5.6.4 使用 SDL 从文件中读取一个场景的描述

在前面的例子中,场景是通过特殊的 OpenGL 调用来描述的,这些调用变换和绘制每一个物体的,如下面的代码所示:

```
glTranslated(0.25,0.42,0.35);
glutSolidSphere(0.1,15,15);          //绘制一个球
```

因此物体是被"硬编码"到程序中的。用这种方式去设计一个场景是笨拙和容易产生错误的。如果设计人员能通过一种简单的语言来描述场景中的物体，然后把描述放进一个文件里，将会是很实用的。那么这个绘制程序就成了一个(更加简单的)通用程序：它在运行时读取一个场景文件，然后绘制出文件中所描述的任何物体。

附录 3 讲述的场景描述语言(SDL)提供了这样一个工具。我们在附录 3 中定义了一个 Scene 类，也可以在本书网站上面找到，它支持读取 SDL 文件和绘制文件中描述的物体。使用下面的一系列命令，可以在实际程序中很容易地应用 Scene 类。创建一个全局 Sence 对象：

```
Scene scn;                           //创建一个 Scene 对象
```

使用 Scene 类中 read 方法读取一个场景文件：

```
scn.read("example.dat");             //读取场景文件，建立物体列表
```

图 5.62 展示了读取简单的 SDL 文件以后 scn 对象的数据结构。

```
!example.dat：一个简单的场景，有一个光源和四个形状
background 0 0 1                     !给场景一个蓝色背景
light 2 9 8 1 1 1                    !放一个白色光源在(2,9,8)
diffuse .9 .1 .1                     !使得下面的物体略微带红色
translate 3 5 - 2 sphere             !在 3 5-2 放置一个球
translate -4-6 8 cone                !在场景中放一个圆锥
translate 1 1 1 cube                 !放一个立方体
diffuse 0 1 0                        !使得下面的物体呈绿色
translate 40 5 2 scale .2 .2 .2 sphere  !加一个小球
```

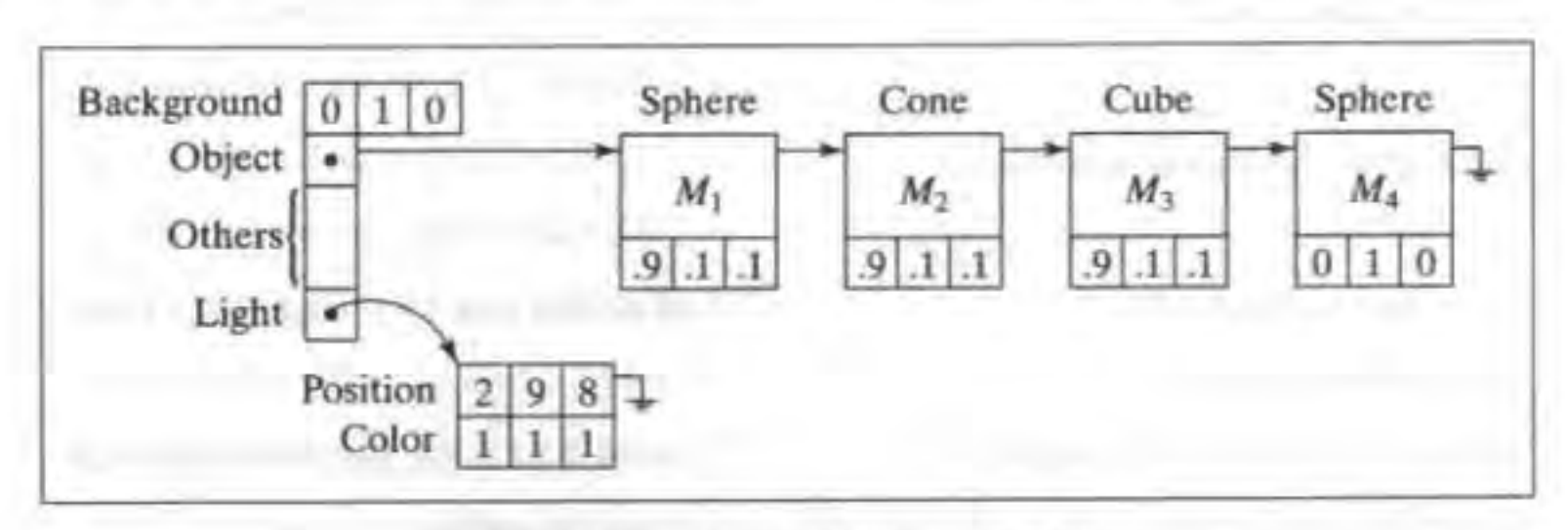


图 5.62 Scene 类的一个对象

第一行是注释；注释一直到行尾。这个场景有一蓝色的背景颜色(red, green, blue)=(0, 0, 1)，一个白色(1, 1, 1)的光源位于(2, 9, 8)，四个物体包括两个球、一个圆锥和一个立方体。Light 域指向光源的列表，同时 obj 域指向物体列表。每一个物体有自己的仿射变换 M，描述它在场景中怎么缩放、旋转、以及平移。同时还含有许多数据域来制定其特殊的材质，这些都很重要，我们会在第 8 章中讨论如何使绘制更加真实。图中只是给出了漫反射的数据域。

一旦光源和物体列表建立了，程序就可以绘制场景了：

```
scn.makeLightsOpenGL();
scn.drawSceneOpenGL();
```

第一条指令将光源描述传递给 OpenGL。场景使用 drawSceneOpenGL()取绘制物体列表中的每一个物体。这个方法的代码非常简单：

```
void Scene::drawSceneOpenGL()
{
    for(GeomObe * p=obj; p; p=p->next)
        p->drawOpenGL();
}
```

这个函数在物体列表中移动一个指针，依次为每个物体调用函数 drawOpenGL()。这是一个使用多态性的好例子，也是面向对象程序的一个里程碑。每一个不同的形状知道怎样去绘制它自己：它有一个函数 drawOpenGL()可以调用恰当的程序画出自己的形状。所以当 p 指向一个球时，drawOpenGL()自动调用关于球的程序；当 p 指向一个圆锥时，drawOpenGL()自动调用关于圆锥的程序；等等。图 5.63 展示了关于球和圆锥的类；它们仅仅在最后调用不同的 OpenGL 绘制程序。每次首先将物体的材质传给 OpenGL，然后使用物体的特殊仿射变换更新模型视点矩阵。原始的模型视点矩阵压入栈中，然后重新取出，防止它被当前物体的绘制所影响。在图 5.63 和图 5.64 的程序中可以看到在 OpenGL 应用程序中材质、颜色、光源是如何被设置、使用的；这些话题稍后会在第 8 章中讨论。

```
void Sphere :: drawOpenGL()
  tellMaterialeGL():                //pass material data to OpenGL
  glPushMatrix():
  glMultMatrixf(trangf.m);          //load this object's matrix
  glutSolidSphere(1.0,10,12):       //draw a sphere
  glPopMatrix();
}
void Cone :: drawOpenGL()
{
  tellMaterialsGL();                //pass material data to OpenGL
  glPushMatrix();
  glMultMatrixf(transf.m);          //load this object's matrix
  glutSolidCone(1.0,1.0,10,12);//draw a cone
  glPopMatrix();
}
```

图 5.63 球和圆锥的 drawOpenGL()方法

图 5.64 展示了读取 SDL 文件和绘制场景的程序，它非常短。但是 Scene、Shape 等类必须在编译的时候能够在头文件中得到。该程序读取特殊的 SDL 文件 myScene1.dat，这个文件重建图 5.59 中的场景。注意通过简单地修改 SDL 文件，程序可以绘制 SDL 文件中的任何场景，而不需要修改程序代码。

```
#include "basicStuff.h"
#include "Scene.h"

Scene scn;          // construct the scene object
//***************display SDL*****************************************
void displaySDL()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    double winHt=1.0;
    glOrtho(-winHt * 64 / 48.0, winHt * 64 / 48.0,
    -winHt, winHt, 0.1, 100.0);
    glMatirxMode(GL_MODELVIEW);
    glLoadIdentity();
    glLookAt(2.3, 1.3, 2, 0, 0.25, 0, 0.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    scn.drawSceneOpenGL();
}
//*****************Main*****************************************
void main( int argc, char ** argv )
{
    glutInit(&argc, argv);
    glutInitDisplayMode( GLUT_DEPTH | GLUT_RGB );
    glutInitWindowSize( 640, 480 );
    glutInitWindowPosition( 100, 100 );
    glutCreateWindow( "read and draw an SDL scene" );
    glutDisplayFunc( displaySDL );
    glShadeModel( GL_SMOOTH );
    glEnable( GL_DEPTH_TEST );
    glEnable( GL_NORMALIZE );
    glViewport( 0, 0, 640, 480 );
    scn.read("myScene1.dat");
    glEnable( GL_LIGHTING );
    scn.makeLightsOpenGL();
    glutMainLoop();
}
```

图 5.64 读取 SDL 文件程序

描述图 5.59 所示场景的 SDL 文件如图 5.65 所示。它首先定义了一个 jackPart，然后使用它 3 次，就得到了有 9 个球的小铁叉，如附录 3 所示。同样，首先定义了一个桌子腿，然后使用了 4 次。

```
!myScene1.dat
light 20 60 30 .7 .7 .7 !put a light at (20,60,30),color:(.7, .7, .7)
ambient .7 .7 .7 !set material properties for all of the objects
diffuse .6 .6 .6
specular 1 1 1
exponent 50
def jackPart{ push scale .2 .2 1 sphere pop
    push translate 0 0 1.2 scale .2 .2 .2 sphere pop
    push translate 0 0 -1.2 scale .2 .2 .2 sphere pop
}def jack{ push use jackPart
    rotate 90 0 1 0 use jackPart
    rotate 90 1 0 0 use jackPart pop
}
def wall{push translate 1 .01 1 scale 1 .02 1 cube pop}
def leg {push translate 0 .15 0 scale .01 .15 .01 cube pop}
def table{
    push translate 0 .3 0 scale .3 .01 .3 cube pop ! table top
    push
    translate .275 0 .275 use leg
    translate 0 0-.55 use leg
    translate-.55 0 .55 use leg
    translate 0 0-.55 use leg pop
}
!now add the objects themselves
push translate .4 .4 .6 rotate 45 0 0 1 scale .08 .08 .08 use jack pop
push translate .25 .42 .35 scale .1 .1 .1 sphere pop
push translate .6 .38 .5 rotate 30 0 1 0 scale .08 .08 .08 teapot pop
push translate 0.4 0 0.4 use table pop
use wall
push rotate 90 0 0 1 use wall pop
push rotate-90 1 0 0 use wall pop
```

图 5.65 一个简单的 SDL 文件

使用像 SDL 一样的场景描述语言,以及读取分析它的工具,设计人员可以专注于设计复杂的场景而不用考虑代码层。场景可以被创建,然后经过一次一次的修改,直到满意为止。而开发人员则致力于构建一个可以绘制任何用 SDL 描述的场景的应用程序。

本章小结

仿射变换是计算机图形学的一个主要部分,因为它提供了处理图形物体的统一工具。仿射变换由一个矩阵定义,对一个点或者向量进行仿射变换等价于一个乘法。设计者经常需要去缩放、定向和定位物体,以便于组成场景,和选择合适的场景视角,仿射变换使得这一切在程序中变得简单。

仿射变换把一个坐标系变换成另一个，当使用齐次坐标时，仿射变换可以使用单一矩阵表示。一系列这样的变换可以组合成一个单一的仿射变换，它的矩阵就是各个矩阵的乘积。重要地，仿射变换保持共线性，因此直线的像仍然是直线，平面的像仍然是平面。由于直线（它的两个端点）和平面（3 个点或者 4 个参数）有简单的表示方法，这大大简化了对直线和平面的操作。除此之外，平行性也被保持下来，因此平行四边形映射为平行四边形，三维平行六面体映射为平行六面体。这样的话，想象仿射变换的几何效果就简单了一些。

三维仿射变换比二维仿射变换复杂的多，尤其是当可视化一个旋转的组合时。一个给定的旋转可以看作是 3 个转过欧拉角度的基本旋转，或者是一个绕着某一轴的旋转，或者是一个简单的具有某种性质的矩阵（它的列都是相互正交的单位向量）。这 3 种形式之间的变换是很重要的。

OpenGL 和其他图形包为操作和应用变换提供了强大的工具。在 OpenGL 中所有顶点在显示之前都经过了好几个变换。程序员可以开发程序定义和操作摄像机，同时操作放入场景中的物体的大小和位置。OpenGL 使用的两个矩阵（模型视点矩阵和视口矩阵）定义了仿射变换，但是投影变换定义了一个透视变换，第 7 章中将会介绍。OpenGL 中还保持了一个变换栈，它使场景设计者很容易控制一个物体的位置和另一个物体的依赖性，同时创建多个相关部分组成的物体。

SDL 语言连同 Scene 和 Shape 类使编程和场景设计分开变得更加简单。一个应用程序只设计一次，就可以绘制任何由一列光源和一列物体描述的场景。这个程序在之后被不同的场景文件一次一次使用。场景设计中的一个关键任务是为每一个物体应用合适的几何变换。因为一些错误是不可避免的，因此使用一种精确可读的方式表达变换是非常有好处的。

下一节将会展示一些学习实例，详细阐述本章的主要思想并给出了一些使用程序练习仿射变换的建议。这些实例涵盖了从深入研究变换理论到实际建模和渲染的方面，例如绘制 CAD 电路和机器人。

案例分析

案例 5.1 使用多个选择旋转绘制图 5.36 所示的星星

（难度：Ⅱ）编写一个函数绘制图 5.36(b)所示的多边形，它是五角星的一个角。然后使用旋转变换绘制整个星星。

案例 5.2 分解一个三维仿射变换

（难度：Ⅲ）这个案例检查几个显著的仿射变换族。

一个三维仿射变换中有什么？

我们再一次忽略仿射变换的平移部分，专注于由 3×3 矩阵表示的线性变换部分。哪种变换藏在其中？这个要比二维的情况复杂的多。Goldman【GEMS Ⅲ，p. 108】展示了每一个 M 都是缩放 S、旋转 R 和两个裁剪 H_1 和 H_2 的乘积：

$$M = SRH_1H_2 \tag{5.34}$$

每一个仿射变换都可以看作是一系列基本的操作,最后跟一个平移。在这个案例中,我们研究这种形式背后的数学,同时看看实际的分解是怎样进行的。

使用变换的类

把仿射变换进行分类是很有用的,依照变换时它们对物体的某种性质能做什么或者不能做什么。都知道它们统称保持物体边的平行性,但是哪种变换同样保持物体每个边的长度,哪个保持每一对边之间的角度?

刚体运动

很明显旋转或者平移都不会改变物体的形状和大小。另外,光用一个平面反射也不会改变物体的形状和大小。因为物体的形状不会被这些变换中单独一个变换影响,同样也不会被它们之间的任意组合影响。我们把所有包括任意旋转、反射和平移的仿射变换的组合记作:

$$T_{\text{rigid}} = \{\text{rotations}, \text{reflections}, \text{translations}\}$$

它们就是大家知道的刚体运动类,因为一个物体刚性地从一个位置和方向移动到另一个位置。这样的变换在齐次坐标下有正交矩阵,它们都是逆和转置相等的矩阵:

$$\widetilde{M}^{-1} = \widetilde{M}^{T}$$

保持角度的变换

一个一致缩放(有相等的缩放因子 $S_x = S_y = S_z$)放大或者缩小一个物体,但是都是一致完成的,因此不会对物体的形状有影响。这样任何两个边之间的角度都不会受到影响。我们把这类记作:

$$T_{\text{angle}} = \{\text{rotations}, \text{reflections}, \text{translations}, \text{uniform scalings}\}$$

这一类比刚体运动大,因为它包括了一致缩放。稍后第 8 章我们会看到,这是一个重要的类,因为光照和着色计算依赖于各个向量之间的点乘。如果某一个变换不改变角度,那么它也不会改变点积,同时光照计算可以在变换和没经过变换的空间相互代替。

(估计时间:3 小时)给定一个三维仿射变换$\langle M, \mathbf{d}\rangle$,我们希望看到它是怎样由一系列变换组合的。依照 Goldman【GEMS III, p. 108】我们建立步骤把 3×3 矩阵 M 分解成缩放 S、旋转 R 和两个裁剪 H_1 和 H_2 的乘积:

$$M = SRH_1H_2 \tag{5.35}$$

请依照这种方式修改每一步,完成一个程序把矩阵 M 分解成单独的矩阵 S、R、H_1、H_2。

假设矩阵 M 的行是 $\mathbf{u}$、$\mathbf{v}$、$\mathbf{w}$,每一个都是三维向量:

$$M = \begin{bmatrix} \mathbf{u} \\ \mathbf{v} \\ \mathbf{w} \end{bmatrix}$$

Goldman 的假设基于古典的 Gram-Schmidt 正交,所以矩阵 M 的行以这种方式组合,这使得它们变得互相垂直,都是单位长度。由这些行组成的矩阵因此是正交的,同时代表一个旋转(或者一个旋转加反射)。Goldman 说明了正交过程实际上是两个裁剪。下面是细节。

下面是每一个步骤。仔细跟随每一步,同时完成括号中的任务。

1. 归一化 $\mathbf{u}$ 到 $\mathbf{u}^* = \mathbf{u}/S_1$,其中 $S_1 = |\mathbf{u}|$。

2. 从 $\mathbf{v}$ 中减去 $\mathbf{u}^*$ 的一部分，使结果和 $\mathbf{u}^*$ 正交。记作 $\mathbf{b}=\mathbf{v}-d\mathbf{u}^*$，其中 $d=\mathbf{v}\cdot\mathbf{u}^*$［证明 $\mathbf{b}\cdot\mathbf{u}^*=0$］。

3. 正交化 $\mathbf{b}$：设置 $\mathbf{v}^*=\mathbf{b}/S_2$，其中 $S_2=|\mathbf{b}|$。

4. 设置一些中间值：$m=\mathbf{w}\cdot\mathbf{u}^*$ 和 $n=\mathbf{w}\cdot\mathbf{v}^*$，$e=\sqrt{m^2+n^2}$，和 $\mathbf{r}=(m\mathbf{u}^*+n\mathbf{v}^*)/e$。

5. 从 $\mathbf{w}$ 中减去 $\mathbf{r}$ 的一部分，使得结果和 $\mathbf{u}^*$、$\mathbf{v}^*$ 正交。记作 $\mathbf{c}=\mathbf{w}-e\mathbf{r}$［证明 $\mathbf{c}\cdot\mathbf{u}^*=\mathbf{c}\cdot\mathbf{v}^*=0$］。

6. 正交化 $\mathbf{c}$：这是 $\mathbf{w}^*=\mathbf{c}/S_3$，其中 $S_3=|\mathbf{c}|$。

7. 矩阵

$$R=\begin{bmatrix}\mathbf{u}^*\\ \mathbf{v}^*\\ \mathbf{w}^*\end{bmatrix}$$

因此是正交的，同时表示一个旋转（计算它的行列式：如果是 -1，那么用 $-\mathbf{w}^*$ 代替 $\mathbf{w}^*$）。

8. 定义裁剪矩阵 $H_1=I+\frac{d}{S_2}(\mathbf{v}^*\otimes\mathbf{u}^*)$，其中 $(\mathbf{v}^*\otimes\mathbf{u}^*)=(\mathbf{v}^*)^T\mathbf{u}^*$ 叫做 $\mathbf{v}^*$ 和 $\mathbf{u}^*$ 的外乘（详见附录 2）。

9. 定义裁剪矩阵 $H_2=I+\frac{e}{S_3}(\mathbf{w}^*\otimes\mathbf{r})$，其中 $(\mathbf{w}^*\otimes\mathbf{r})=(\mathbf{w}^*)^T\mathbf{r}$。

10. 证明 $\mathbf{u}^*H_1=\mathbf{u}^*$，$\mathbf{v}^*H_1=\mathbf{v}^*+d\mathbf{u}^*/S_2=\mathbf{v}/S_2$，和 $\mathbf{w}^*H_1=\mathbf{w}^*$。首先证明外乘的性质，对于任意向量 $\mathbf{a}$、$\mathbf{b}$、$\mathbf{c}$：$\mathbf{a}(\mathbf{b}\otimes\mathbf{c})=(a\cdot b)c$。然后对三个正交向量 $\mathbf{u}^*$、$\mathbf{v}^*$ 和 $\mathbf{w}^*$ 应用这个性质，证明上述三个性质。

11. 证明 $\mathbf{u}^*H_2=\mathbf{u}^*$，$\mathbf{v}^*H_2=\mathbf{v}^*$，和 $\mathbf{w}^*H_2=\mathbf{w}^*+e\mathbf{r}/S_3=\mathbf{w}/S_3$。

12. 把这些综合起来证明 $M=SRH_1H_2$。S 定义为：

$$S=\begin{bmatrix}S_1 & 0 & 0\\ 0 & S_2 & 0\\ 0 & 0 & S_3\end{bmatrix}$$

注意这个分解不是唯一的，因为向量 $\mathbf{u}$、$\mathbf{v}$ 和 $\mathbf{w}$ 可以在另一个顺序下正交。比如我们可以首先形成 $\mathbf{w}^*$ 是 $\mathbf{w}/|\mathbf{w}|$，然后从 $\mathbf{w}^*$ 中减去一个 $\mathbf{v}$，使得结果正交于 $\mathbf{w}^*$，然后减去一个向量 $\mathbf{u}$，使得结果和前两个向量都垂直。写出程序：

```
void decompose( DBL m[3][3], DBL S[3][3], DBL R[3][3],
                DBL H1[3][3], DBL H2[3][3] )
```

其中 DBL 定义为 double，它输入 M，把结果返回在 S、R、H_1 和 H_2 中。使用多个 M 测试你的程序保证它们的积和原矩阵 M 一样。

分解二维变换的其他方式，可以参考【Thomas，GEMS，p. 320】和【Shoemaker，GEMS IV，p. 207】中的例子。

案例 5.3 在场景中创建一个物体并与其交互

（难度：Ⅲ）程序中的交互功能可以增加用户的兴奋度，尤其是在游戏中，这个工具能在

场景中增加或者去除某个物体。比如使用鼠标或者键盘在场景中增加一个桌子或者是机器人,或者是让这些东西突然消失。还有一个有趣的是让机器人的头左转右转,或者让机器人挥舞它的前臂。在 OpenGL 三维程序中怎么实现呢?需要写键盘控制去控制创建或者删除场景中的物体,同时还要写鼠标控制去控制创建或者删除场景中的物体。机器人的头部需要很细致,可以是一个简单的圆柱或者圆锥。处理鼠标反馈函数,使得鼠标左键在物体上面点击可以使程序交替显示网格和实体多边形。

案例 5.4 飞过主要形状

(难度:Ⅱ)使用图 5.57 作为指导,创建一个应用程序允许用户控制摄像机飞过整个场景。飞机一开始没有按键的时候应该是固定的。

进一步阅读

有很多关于变换的图书。George Martin 的"Transformation Geometry"【Martin82】,Yaglom 的"Geometric Transformations"【Yaglom62】。

第6章 使用多边形网格建模

本章学习目标

- 开发用于处理3D空间中的物体的工具
- 使用多边形网格表示实体对象
- 绘制网格对象的简易线框视图
- 介绍粒子系统并说明它的运行机制
- 介绍基于物理的系统并阐述应用程序如何生成并显示它们

预览

6.1节概述3D建模过程。6.2节对多边形网格进行描述，可以让你使用简单的数据结构对复杂3D物体进行绘制。我们要探讨这种网格的性质，并给出显示它们的算法。

6.3节给出了许多有趣的3D形体，包括柏拉图多面体、Bucky球、穹顶以及棱柱。6.4节探讨拉伸或扫掠形体，并且展示了如何生成动感图标中的3D字母、在空间中起伏荡漾的管状与蛇形物体以及旋转曲面。

6.5节讨论如何对具有光滑曲面的实体进行网格建模。这些网格对某些光滑的构成面(underlying surface)进行了近似，其关键要素是对构成面上的每一点计算法向量。一些有意思的光滑实体将要被研究，包括二次曲面、直纹面和Coons面片、双变量参数显式表示的曲面以及旋转曲面。

6.6节对粒子系统这一激动人心的领域进行了描述，它在动画中的应用尤其使人着迷。在这种系统中，场景包含了成百上千的基本粒子，这些基本粒子通常简单地表示为圆点。这些圆点存在于3D空间中，并且具有诸如颜色、生命周期、速度和大小等属性。在动画中，它们在某一时刻诞生然后在其后的某一时刻消亡。粒子系统的实例包括草地、喷泉喷射的水滴、掠过观察者的星团、云彩和雾气。图6.1(a)描绘了一个喷泉，其中的粒子是喷向空中的水滴。在对应的动画中，这些水滴又回落进水池，泛起层层涟漪。图6.1(b)是星团掠过观察者的画面，我们经常可以从电视与电影中看到这种场面。本书的网站上可以得到上述两个程序。

6.6.2节也同时介绍了基于物理的系统，在这种系统中场景里的各种对象是通过弹簧、小装置部件、电磁力、重力或者其他物理机制建模连接起来。编写应用程序的程序员通过编写一些子程序，使得各种对象可以依据这些力的作用进行合理的反应。基于物理的系统在动画中的表现最为生动，我们可以看到各种物体上下摆动、前后摇曳、来回旋转，就好像被无形但可知的力量控制一样。

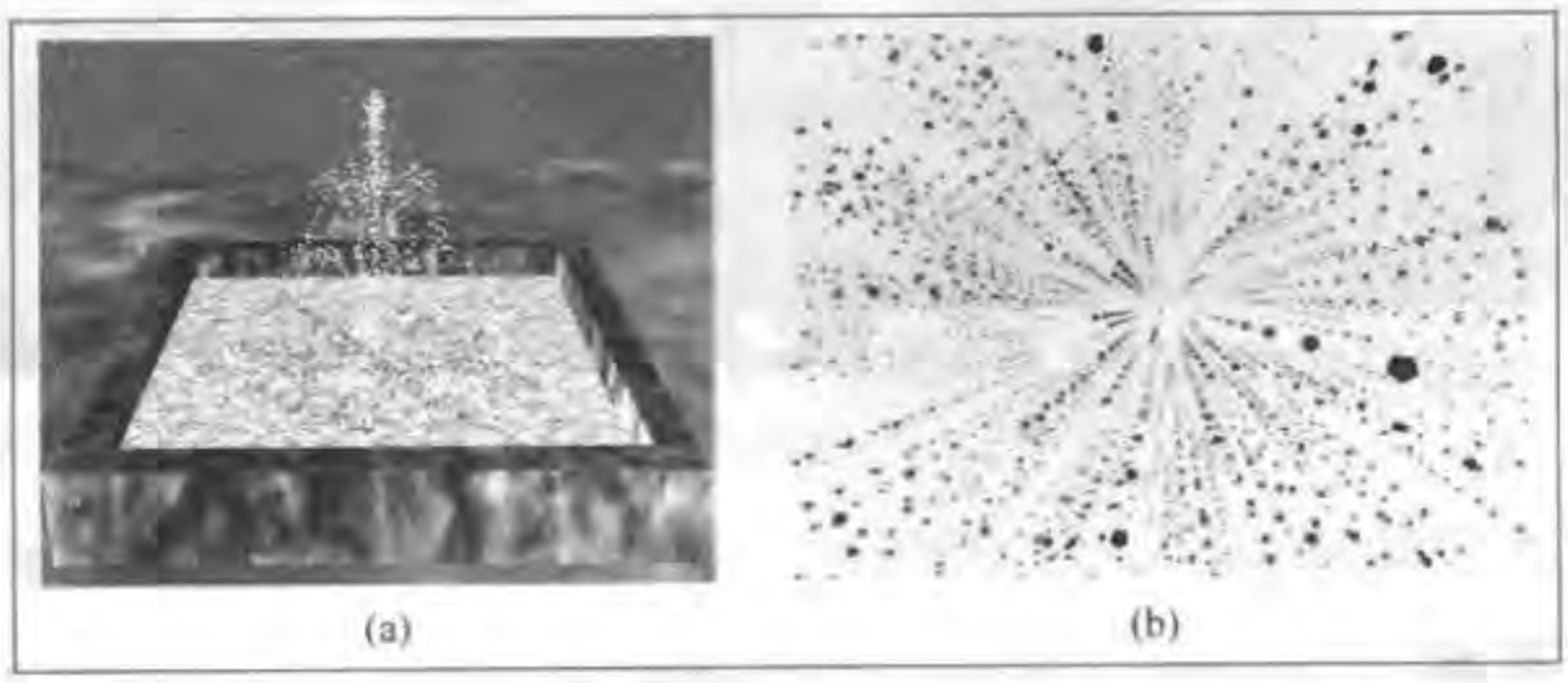


图 6.1

(a) 用粒子系统做的喷泉;(b) 星团模拟

本章结尾的案例分析对一些思想近一步进行了探索,并且要求读者编写应用程序对它们进行验证。其中的一些案例是十分实用的,例如从一个文件中读出网格数据以及生成诸如管状体和拱门等漂亮的 3D 形体。

6.1 概 述

本章探讨用多边形网格对 3D 物体进行描述的方法。多边形网格是多边形或面片的简单集合,这些面片拼接在一起形成物体的表面。在图形学领域中,它们已经成为表示一大类实体形状的标准方法。我们在前面已经看到了一些多边形网格的例子,比如立方体和二十面体以及对球面、柱面、锥面等光滑形体的网格近似表示。在本章中我们会看到更多的例子。由于使用多边形很简便,使得多边形网格在图形学领域占有很重要的地位。多边形易于表示(通过使用一系列的顶点)和变换,它们有简单的性质(单一的法向量、明晰的内部与外部等),并且容易绘制(使用多边形填充算法或把纹理映射到多边形上)。

包括 OpenGL 的许多渲染系统都是通过绘制一系列多边形来对物体进行绘制的。每一个多边形面片被送到图形管线,在那里,面片上的顶点要经历各种变换,最后对面片中未被裁剪的部分进行上色或渲染处理并在显示设备中进行显示。

我们想要了解如何通过指定合适的面片集来设计复杂的 3D 形体。使用多边形网格可以精确地表示一些物体,而其他物体只能被近似地表示。例如图 6.2(a)中的仓房,它本来就是由一些平面构成的,所以正如所期望的那样,在渲染结果中可以清晰地看到面与面之间的棱边。但是对于图 6.2(b)图中的圆柱,情况又变得不同了。图 6.2(b)中的圆柱原本看上去应该具有圆滑的表面,但是这种圆滑性仅仅使用多边形是无法实现的:各个平面很容易区分开来,面与面之间的棱边也很明显。我们必须开发出一些渲染技术使得网格看上去很光滑,正如图 6.2(c)所反映的那样。将在第 8 章探讨这种技术的具体细节。

我们先概述性地描述多边形网格,然后了解如何使用 OpenGL 在程序中对它们进行定义和操作。我们运用这些思想来建模多面体——它们天然就由平面构成,并且研究一些有意思的多面体。然后我们解决如何使用网格来近似一个光滑弯曲的形状,还要开发出一些必要的工具生成并操作这些模型。

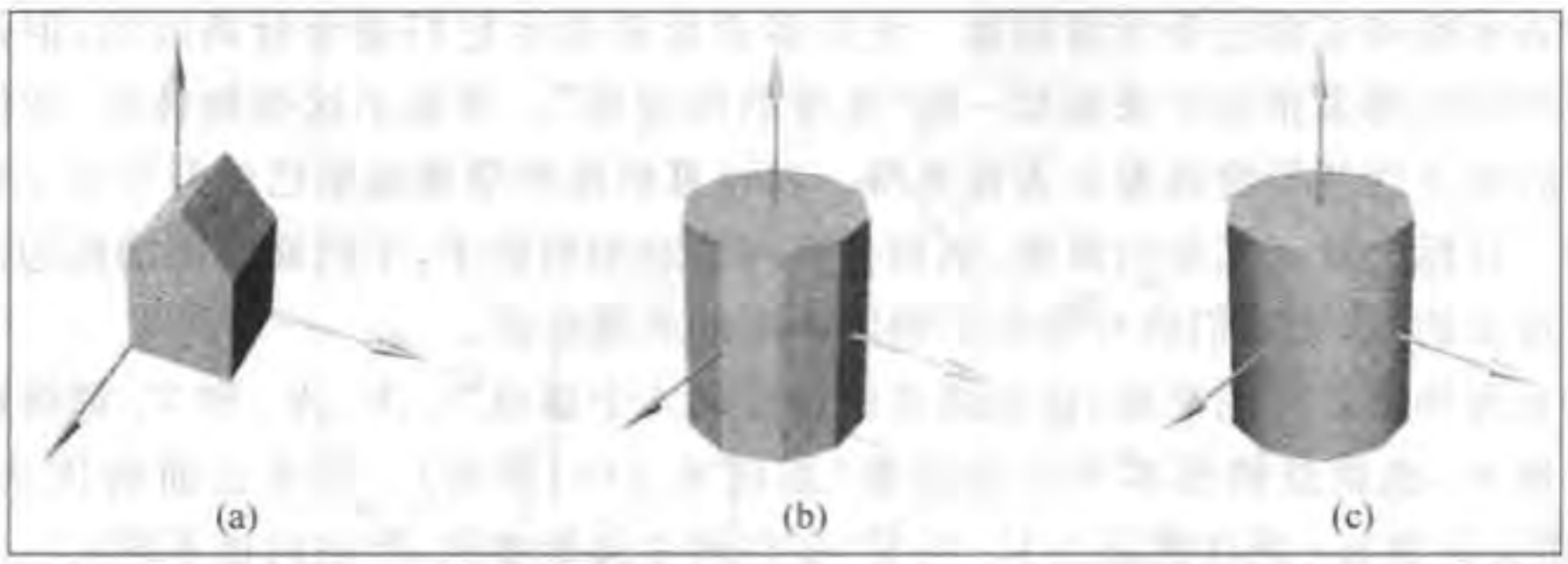


图 6.2　用 Mesh 表示的各种形体

6.2　多边形网格实体建模初步

无论是立体的形状还是薄薄的表皮我们都将使用网格进行建模。一个物体是立体的，如果它的多边形面片拼接在一起围成了一块封闭的空间。对于其他情况，各个拼接的面片并没有形成封闭的空间，只是组成了一张很薄的面。上述两种情形的多边形集合都称为多边形网格(或简称为网格)。

多边形网格由一系列多边形定义并且具有其中每一个多边形的朝向信息。这个重要的朝向信息通常就是多边形面片所在平面的简单的外法线向量。在渲染过程中使用这种朝向信息可以确定从光源射到面片上的光线沿不同方向反射的量，我们将在第 8 章讨论这个问题。图 6.3 展示了仓房各个面上的法向量。正如将在第 8 章讨论的那样，一个面的明暗程度与其外法线向量和朝向光源向量的夹角(在图中用 θ 表示)的余弦值成正比。因此在最后的绘制和纹理贴图的过程中，曲面相对于光源的朝向信息起了很重要的作用。我们将在第 8 章讨论 OpenGL 渲染语言背后的新思想。

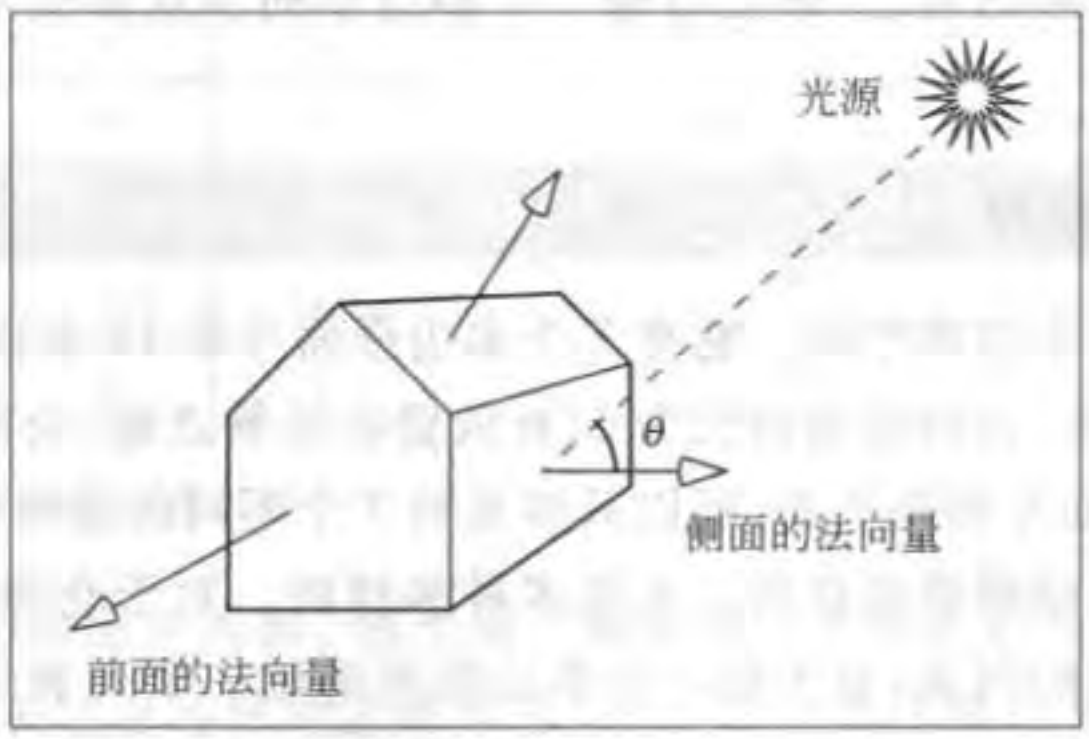


图 6.3　法线方向决定了面的明暗程度

顶点法线向量与面法线向量

对于一些形体，给面片中的每一顶点单独指定一个法向量比单纯地给整个面片指定一个法向量效果要好。这种方法有利于加快光滑弯曲物体裁剪与渲染的过程。我们经常要使

用网格来表示圆球或圆柱等光滑曲面。使用多边形来表示它们是十分高效的,但是我们心里要明白我们是用多张面片来近似一张"光滑的构成面"。当显示这些物体时,我们要弱化面片之间的独立性使得物体看上去很光滑。在计算机图形学领域里已经开发出了许多算法来实现这一目标。对于其他的形体,例如仓房或游戏用的骰子,不同面之间的棱边本来就应该是清晰可见的,那么我们将对每个面指定不同的光强度值。

考虑仓房外墙这样的平面,这也就是说侧面的 4 个顶点 V_1、V_2、V_3 和 V_4 都将被赋予相同的法向量 $\mathbf{n}_1$,也就是侧面本身的法向量(如图 6.4(a)所示)。但是正面的顶点,比如说 V_5,将使用法向量 $\mathbf{n}_2$(请注意顶点 V_1 与 V_5 在空间中位置相同,但法向量不同)。

考虑另一种情形,对于图 6.4(b)中圆柱面这样的光滑曲面,面 F_1 的顶点 V_1 和面 F_2 的顶点 V_2 却共用相同的法向量 $\mathbf{n}$,这个法向量垂直于光滑的原始构成面。在 6.2.2 节中会看到如何方便的计算这个向量。

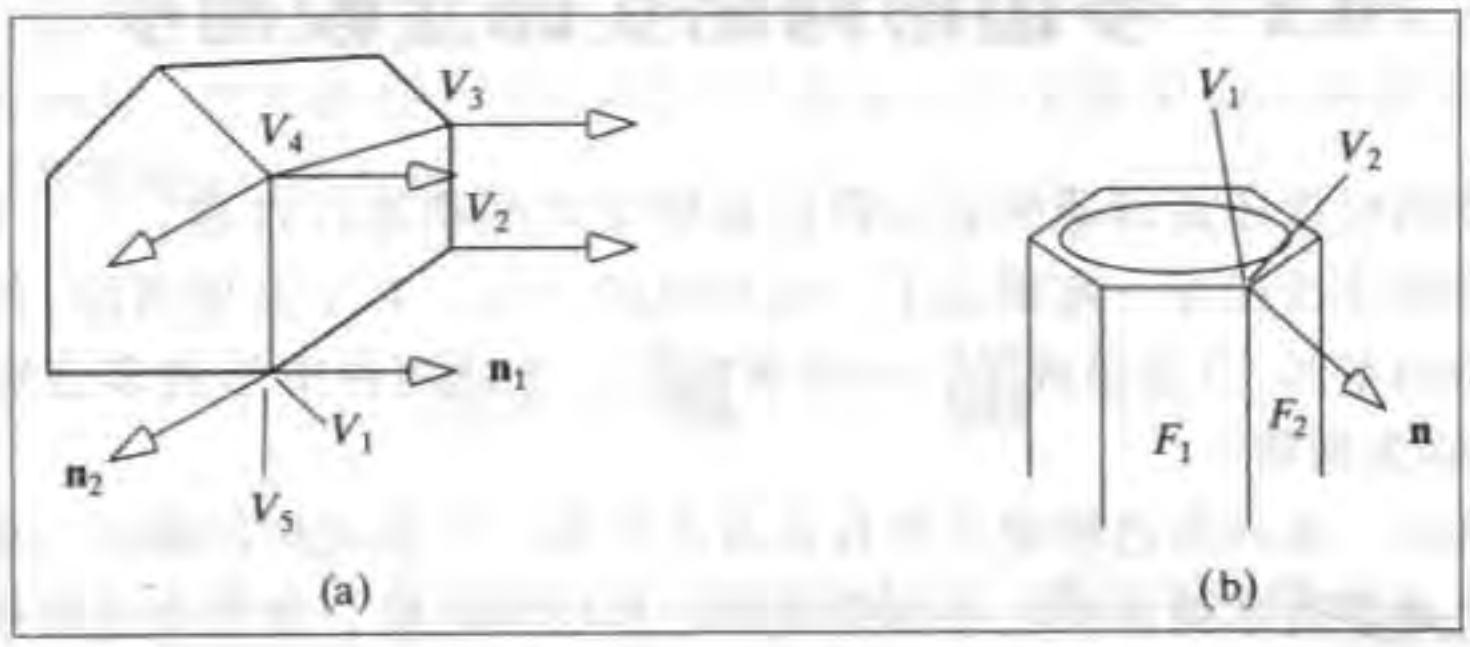


图 6.4 在面的每个顶点指定一个法线

6.2.1 定义多边形网格

多边形网格是多边形的集合,并且对每一个多边形的顶点都指定一个法向量。我们先来看一个例子。

例 6.2.1 仓房模型

图 6.5 是仓房的一个简单形体。它有 7 个多边形面片和 10 个顶点(每一个顶点被 3 个面片共用)。为方便起见,它的底面边长为 1(被放置在场景之前,仓房可以进行合理的放缩与定位)。因为仓房的面片都是平面,所以只涉及到 7 个不同的法向量,每个面片对应一个。

在文件或程序中存储网格信息的方式是多种多样的。对于仓房这个例子,你可以构建一个由 7 个多边形组成的列表,对于每一个多边形再构建一个列表记录其所有顶点的位置与顶点的法线信息(一共有 30 个顶点与 30 个法线)。因为总共只有 10 个不同的顶点与 7 个不同的法线,因此这种表示方法是十分冗余的。

一种更高效的方法是使用顶点列表、法线列表和面片列表这样 3 个分离的列表。顶点列表记录网格中不同顶点的位置。法线列表记录模型中存在的不同的法线向量。面片列表仅仅记录顶点列表与法线列表中的索引。我们将会看到,采用这种方法使用 10 个顶点、7 个法线和由 7 个简单的面描述符组成的列表就可以刻画仓房模型。

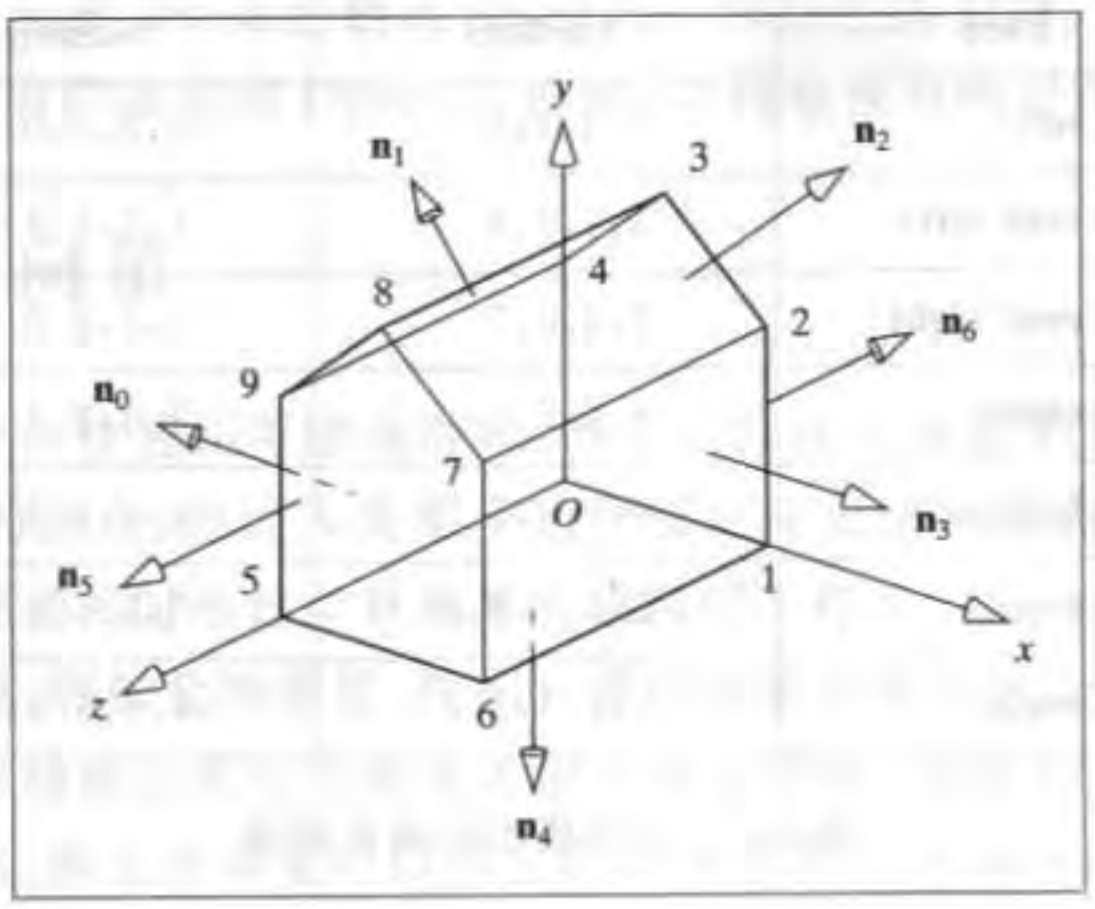


图 6.5　仓房模型

上述 3 个列表是协同工作的：顶点列表包含位置或几何信息，法线列表包含方向信息，面片列表包含连通性或拓扑信息。

图 6.6 是仓房模型的顶点列表。图 6.7 是 7 个不同法线组成的列表。顶点的序号为 0～9，法线的序号为 0～6。图中的法向量都已经被单位化，因为大多数的渲染算法要求法向量是单位长度（使用两个单位向量的点积可以计算出余弦值）。

顶点	x	y	z
0	0	0	0
1	1	0	0
2	1	1	0
3	0.5	1.5	0
4	0	1	0
5	0	0	1
6	1	0	1
7	1	1	1
8	0.5	1.5	10
9	0	1	1

图 6.6　仓房模型的顶点列表

normal	n_x	n_y	n_z
0	−1	0	0
1	−0.707	0.707	0
2	0.707	0.707	0
3	1	0	0
4	0	−1	0
5	0	0	1
6	0	0	−1

图 6.7　法向量列表

图 6.8 是仓房模型的面片列表：每个面片都拥有一个顶点列表和顶点法向量列表。为了节省空间，只存贮相关顶点与法向量的索引（由于每个面片都是平面，面片中的所有顶点共用同一个法向量）。选择面片中任意一个顶点作为开始，然后在面中逐点进行遍历，直到组成一个闭合回路，以上述方式形成此面片的顶点列表。遍历多边形的方式有两种：顺时针方向与逆时针方向。例如，面 5 可以表示为(5，6，7，8，9)或(9，8，7，6，5)。尽管可以使用上述的任意一个遍历方向，但是我们遵守一个在实际应用中简便可行的准则：

站在物体外进行观察，以逆时针的方向遍历多边形。

Face	Vertices	Normal
0(left)	0,5,9,4	0,0,0,0
1(roof left)	3,4,9,8	1,1,1,1
2(roof right)	2,3,8,7	2,2,2,2
3(right)	1,2,7,6	3,3,3,3
4(bottom)	0,1,6,5	4,4,4,4
5(front)	5,6,7,8,9	5,5,5,5,5
6(back)	0,4,3,2,1	6,6,6,6,6

图 6.8 仓房模型的面片列表

在这个准则之下,如果你沿着面片的边界逐点进行遍历,那么面的内部始终在你的左边。后面我们还会设计算法实现这种排序方式。基于这个准则,程序可以轻易地判别出面片的正面与背面。

仓房是一个数据密集模型的例子,其每一个顶点由设计人员手工输入。作为对比,稍后我们会见到一些由算法自动生成的模型。比如未来的房屋主人想要设计一下他们心目中理想的房屋,并且还要让 CAD 软件或建筑设计人员把房屋的 3D 轮廓图或全彩图展示出来。但是要把诸如房屋、医院或华盛顿的五角大楼这样的建筑用手工的方式填入上述列表将是一项十分繁重的工作。一个可能的方案是把列表的数据存储在文件中。有了这些文件,对仓房或房屋的各种操作就容易得多。一些 CAD 或 3D 图形程序,如 AutoCAD 和 3D Studio Max,可以自动生成先前描述的那些列表。设计人员使用一个简单的命令就可以把医院、工厂或教堂的所有数据保存在一个文件中,并且可以稍后在应用程序的渲染过程中使用它们。

6.2.2 3D 文件格式举例

你也许对一些流行的 CAD 与 3D 图形软件如何在文件中保存 3D 网格信息很感兴趣。尽管没有一种标准的文件格式,但是一些文件格式已经被证实是高效而且易用的。例如,图 6.9 是从 3 个不同角度分别观察同一个 3D 网格模型文件得到的结果,使用的是斯坦福大学计算机图形实验室开发的 qs 文件格式。这个网格模型有 566 098 个顶点[①],基于这些顶点又得到 2 748 318 个点(大约 5 500 000 个三角形)。请想象一下如果用手工方式绘制那些点。

图 6.9 一个 3D 网格模型的三个视图

① 资料来源:斯坦福大学计算机图形实验室

OpenGL 可以读取许多 3D 文件格式，其中的一部分包括 3DS、VRML、PLY、MS3D 和 OBJ。本书的配套网站有许多使用 OpenGL 读取 3D 网格模型的资源。

6.2.3 计算法向量

手工方式设置顶点的位置还可能做得到，但是计算法向量就不那么容易了。通常每个面片要有三个或更多的顶点，设计人员用手工方式标注出法向量会是十分具有挑战性的。最好可以在构造网格模型的时候让计算机来完成这项工作。

如果是平面的情形，例如仓房模型，我们只需找到面片本身的法向量然后把它附给面片中的所有顶点。一种直接的方式是用向量叉积计算法向量，如图 4.14 所示。设 V_1、V_2、V_3 为面片中相邻的三个点，那么法向量可以用它们的叉积表示为 $\mathbf{m}=(V_1-V_2)\times(V_3-V_2)$。还可以把得到的这个向量规范化为单位长度。

这个简便的方法却会带来两个问题。

(1) 如果两个向量 V_1-V_2 和 V_3-V_2 近乎于平行的话，叉积后的结果会非常小(为什么)，这样就会带来数值不稳定性的问题。

(2) 我们稍后会看到，多边形有可能并不是理想中的那么平直，也就是说其所有的顶点并不落在一张平面上。因此用这样的顶点表示的面片并不是真正的平面。为了得到它的法向量，我们需要把所有的顶点因素考虑进去然后求某种平均值。

案例 6.2 会讲述并推导一个鲁棒的方法来解决上述问题，这个方法是由 Martin Newell【Newll79】得到的。它通过如下等式计算法向量 $\mathbf{m}$ 的分量 m_x、m_y、m_z：

$$m_x=\sum_{i=0}^{N-1}(y_i-y_{next(i)})(z_i-z_{next(i)})$$

$$m_y=\sum_{i=0}^{N-1}(z_i-z_{next(i)})(x_i-x_{next(i)})$$

$$m_z=\sum_{i=0}^{N-1}(x_i-x_{next(i)})(y_i-y_{next(i)}) \tag{6.1}$$

这里的 N 是面片中的顶点总数，(x_i, y_i, z_i) 是第 i 个顶点的位置坐标，$next(j)=(j+1)\bmod N$ 是面中顶点 j 的下一个顶点序号。取模运算保证了第 $(N-1)$ 号顶点的后一个顶点是顶点 0。这种方法在计算法向量的每一个分量过程中，只需对每一条边使用一次乘法运算，而且不需要做共线测试。

用 Newell 法计算得到的向量 $\mathbf{m}$ 既可能指向多边形的内部也可能指向多边形的外部。如果从外部观察多边形以逆时针遍历其顶点(以 i 递增的顺序)，那么 $\mathbf{m}$ 指向此面的外部。

例 6.2.2

考虑此多边形，其顶点为 $P_0=(6, 1, 4)$，$P_1=(7, 0, 9)$，$P_2=(1, 1, 2)$。使用 Newell 法计算它的法向量。因为这个多边形是三角形，所以它一定是平直的。

解：直接使用叉积计算 $((7, 0, 9)-(6, 1, 4))\times((1, 1, 2)-(6, 1, 4))=(2, -23, -5)$。应用 Newell 法可以得到相同的结果：$(2, -23, -5)$。

练习

练习 6.2.1 使用 Newell 法

对于顶点(6, 1, 4)、(2, 0, 5)、(7, 0, 9),比较一下用 Newell 法和通常的叉积法得到的法向量是否相同。然后使用 Newell 法计算如下多边形的(n_x, n_y, n_z),这个多边形的顶点为(1, 1, 2)、(2, 0, 5)、(5, 1, 4)、(6, 0, 7)。此多边形是否是平面?如果是的话,使用叉积法得到它的法向量,然后将其与 Newell 法的结果对比一下。

练习 6.2.2 非平面多边形又如何

考虑图 6.10 中的四边形,其顶点为(0, 0, 0)、(1, 0, 0)、(0, 0, 1)、(1, a, 1)。当 a 不等于 0 时,这是一个非平面多边形。使用 Newell 法计算它的法向量,并且讨论当 a 取不同值时,这个估计值的好坏程度。

练习 6.2.3 标准立方体的表示

请生成标准立方体的顶点、法线与面片列表。标准立方体的中心在原点,它的边与坐标轴平行,并且每条边的长度等于 2。因此它的 8 个顶点坐标分别为(−1, −1, −1)、(−1, −1, 1)、(−1, 1, −1)、(−1, 1, 1)、(1, −1, −1)、(1, −1, 1)、(1, 1, −1)和(1, 1, 1)。

练习 6.2.4 带洞的面

图 6.11 要说明的是如何把一个带洞的面片保存进面片列表。正如图中所示,添加了一对本不存在的边用来连接这个面片的内外边界。

面片的遍历方式是当在面片外部行走时面片的内部要在你的左侧。所以洞的遍历方向是顺时针。假设我们从外部观察图中的面,顶点列表应该是:5 4 3 8 9 6 7 8 3 2 1。在此面中再随便画上另一个洞,然后给出面相应的顶点列表。与每个顶点关联的法线又是什么?

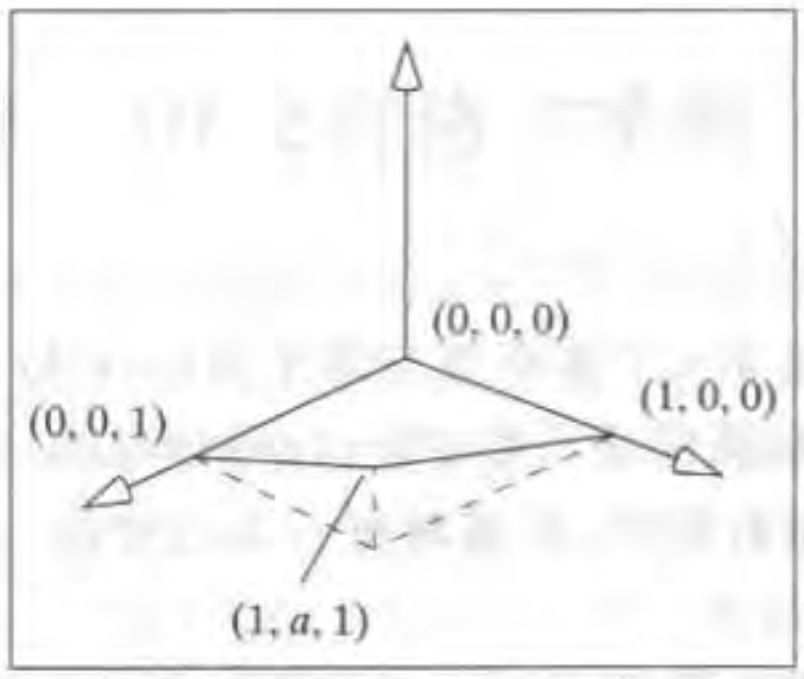


图 6.10 非平面多边形

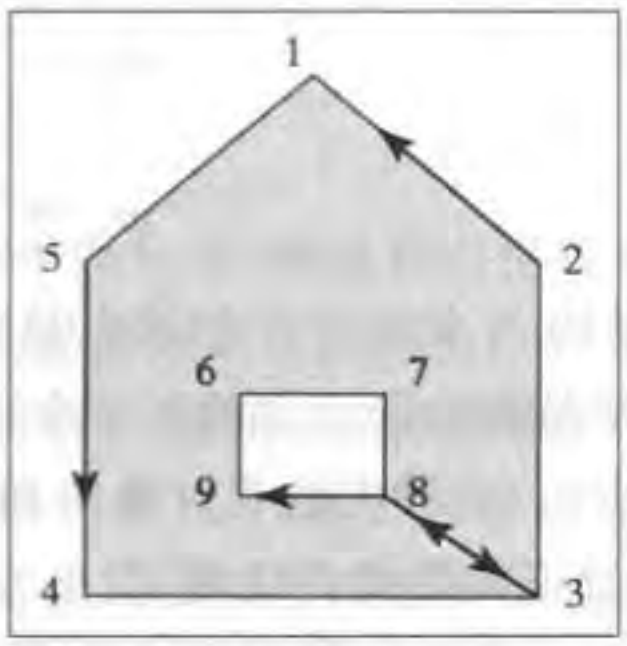


图 6.11 带洞的面

6.2.4 网格的性质

给出一个已经指定了顶点、法向量以及面片列表的网格,我们想要知道它表示的是何种物体。我们关心的一些性质如下。

- 实体性:先前已经提到过,如果一个网格的面片包围一块有限体积的空间,那么它表示的是一个实体对象。

- 连通性：网格是连通的，如果它的每一个面片至少与其他的面片共享一条边。（如果一个网格不是连通的，那么通常它表示的对象数多于一个。）
- 简单性：一个网格是简单的，如果它表示的是一个实体并且其中不含有洞；它可以不经撕裂地变化为一张球面（注意，这里的“简单”与简单多边形的“简单”意义差别很大）。
- 平直性：网格是平直的，如果它的每一个面片都是平面多边形：面片的顶点都落在同一张平面上。一些图形算法对于平直的面片处理起来更加高效。三角形天生就是平直的，所以基于这个优点一些建模软件只使用三角形。另一方面，四边形有可能是平直的，也有可能不是。例如图 6.10 的四边形是平直的，当且仅当 a=0。
- 凸性：一个网格表示的是一个凸体，如果连接此物体任意两点的线段也全部包含在物体中。多边形的凸性首次在 2.3.6 节中提到过。图 6.12 是一些凸物体和一些非凸物体。对于每个非凸物体，图中画出了一根线段，它的端点落在物体上但它本身并不被物体所包含。

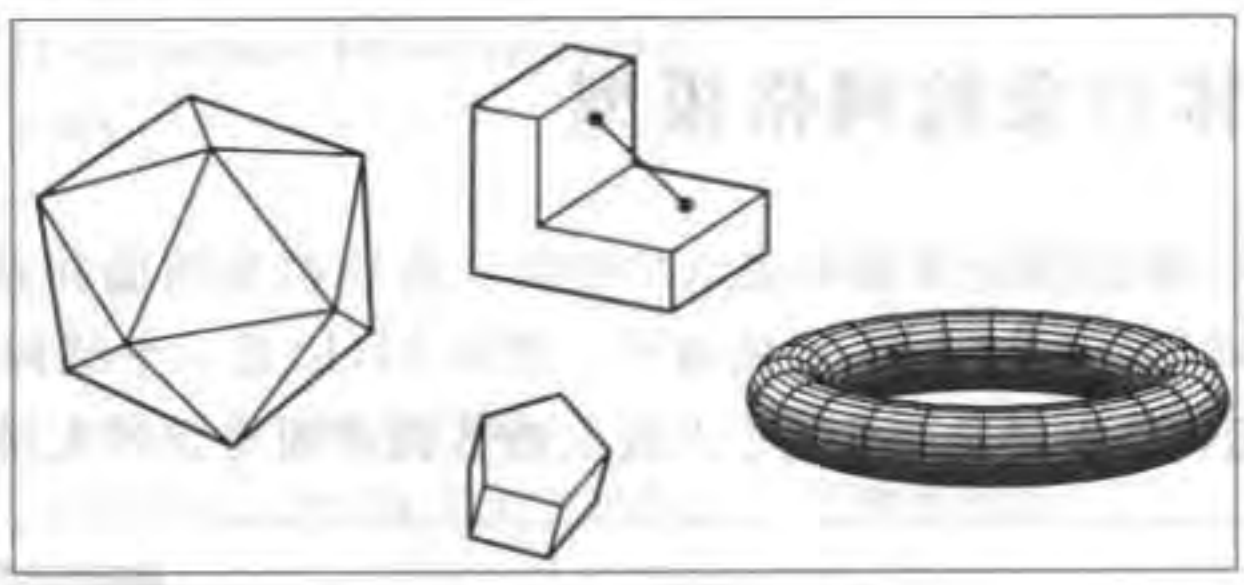


图 6.12　凸的以及非凸的 3D 物体

练习

练习 6.2.5

请识别出图 6.12 中哪些物体是凸的，哪些是非凸的。

简单的仓房模型具有前述的所有性质（请验证）。给出了一个网格，有些性质很容易用程序进行检验，也就是说存在一个简单的算法来完成此项工作（我们稍后会讨论一些）。另一些性质，例如实体性，就很难检验。

图形学领域，我们建模的对象所选择的多边形网格可能具有上述的某些或全部性质：具体的选择跟网格的用途有关。如果一个网格表示的是具有某种材质的物理对象，也许要计算它的质量或重心，那么至少应该保证网格的连通性与实体性。如果我们只是打算绘制物体，那么要求就宽松的多，因为即使是一些非物理的对象，它也可以被绘制出来。

图 6.13 是一些用网格表示的物体。金字塔由三角形构成，它的面显然都是平直的。它不仅是凸的；事实上它满足上述的所有性质。

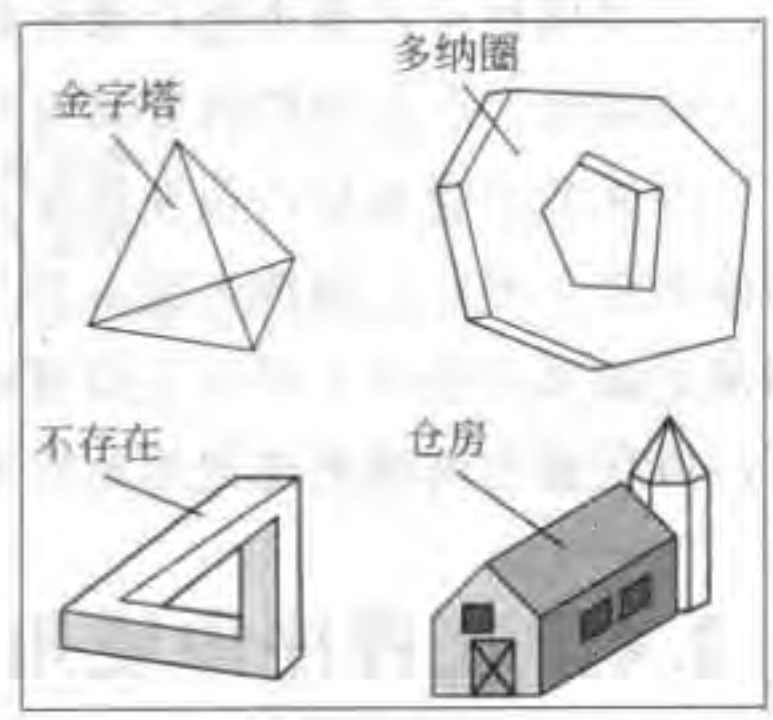


图 6.13　用网格描述的实体

多纳圈模型具有连通性和实体性,但是它不是简单的(有一个洞)也不是凸的。它具有两个带洞的面片。单从图中看不出它的面片平直与否。稍后我们给出一个算法可以通过面片列表与顶点列表判断它的平直性。

被标上"不可能"的那个图看上去是不可能的但实际并非如此。这个物体甚至可以用网格表示出来。Gershon Elber 的研究网站(http://www.cs.technion.ac.il/~gershon/EscherFor-Real/)上展示了一些看上去不太可能的物体,但是却描述了如何对它们进行建模的方法。

仓房模型看上去好像由两个部分组成,但是它可以包含进一个网格之中。它是否是连通的取决于筒仓与主仓在连接处面片的定义方式。仓房模型也表达了在图形学中经常遇到的一种情形。为了表示窗户和门,一些面片被添加到网格中用来提供对象的纹理。例如,仓房的侧面是一个不含洞的长方形,但是却加入了两个正方形来表示窗户。这些正方形和仓房的侧面在同一个平面中。这时网格就不再是连通的,但是它仍然可以在图形设备上显示出来。

6.2.5 非实体对象的网格模型

图 6.14 是其他一些用多边形网格表示的物体。这些都是曲面并且最好把它们看成是无限薄的壳。图 6.14(a)是一个开了盖的盒子。图 6.14(b)是一个结构件,它的面用来对一张光滑曲面进行逼近。图 6.14(c)是基于人脸上的数据点而得到的人脸模型。

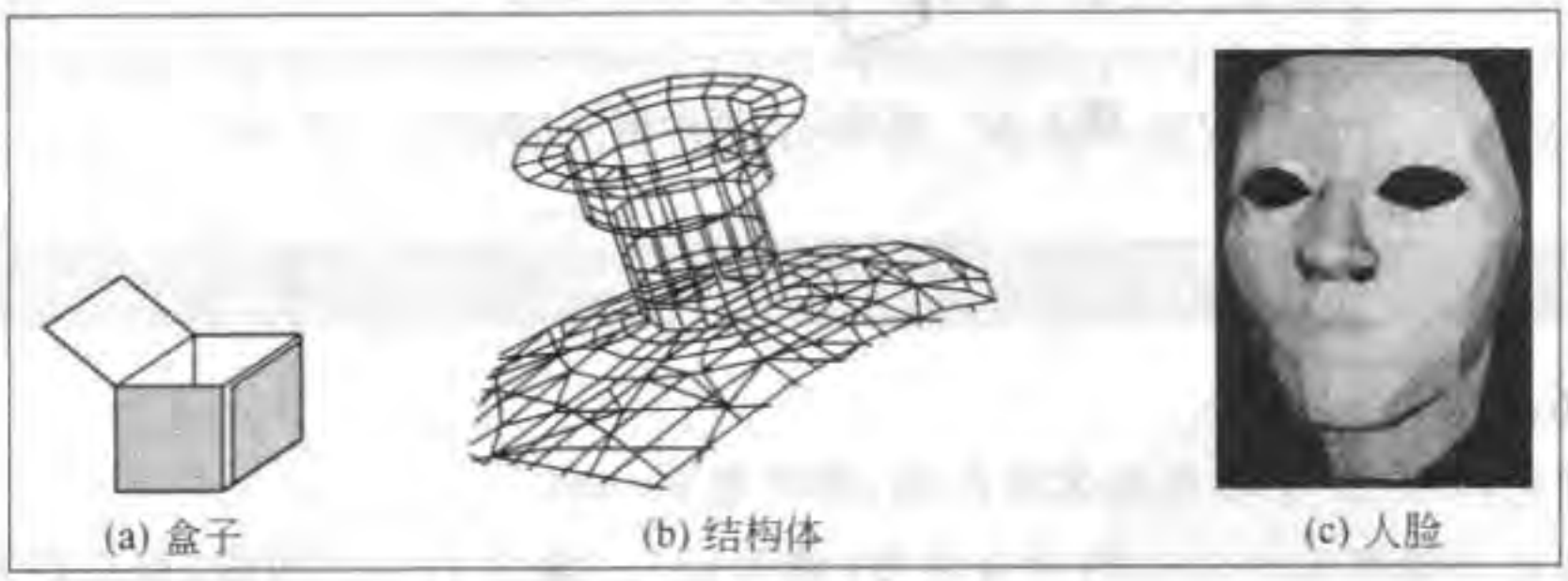


图 6.14 用网格描述的曲面

如果网格中的每个面片都绘制为着色多边形,那么图片看上去就不十分自然,例如上述的人脸模型。后面我们将要研究用网格模型绘制光滑曲面的工具。

许多几何建模软件包使用多边形网格来构造某些物体模型(实体或曲面),力图捕捉到物体的真正形状。构造生成各种列表是一个十分困难的问题。例如,考虑设计出一个算法用来生成顶点和面片列表近似逼近引擎、假肢和建筑等形状。通过使用足够多的面片,网格可以以任意的精确度逼近原始曲面。

6.2.6 在程序中使用网格

我们想要一种高效的方法使得在程序中控制网格用来便捷地生成和绘制物体。由于经

常使用文件来存储网格数据，所以我们还需要能够存取网格文件的简单方法。

很自然地，可以定义一个 Mesh 类并且在其中放入我们想要的功能。图 6.15 是 Mesh 类的声明，除此之外还有两个辅助的类 VertexID 和 Face[①]。一个网格物体拥有一个顶点列表、一个法线列表以及一个面片列表，分别用数组 pt、norm、face 表示。当知道了这些数组的规模大小后就可以在程序运行时动态给它们分配空间。数组的大小分别保存在 numVerts、numNormals 和 numFaces 中。还可以在稍后添加其他的数据域用来描述物体的各种物理属性，例如重量和材质。

```
//############VertexID #############
class VertexID{
public:
int vertIndex;                          // 此顶点在顶点列表中的索引
int normIndex;                          // 顶点的法线索引
};
//#############Face #############
class Face{
  public:
int nVerts;                             // 面片中的顶点数
VertexID * vert;                        // 顶点和法线索引组成的列表
Face(){nVerts=0; vert=NULL;}            // 构造函数
~Face(){delete[] vert; nVerts=0;}       // 析构函数
};
//#############Mesh #############
class Mesh{
  private:
int numVerts;                           // 网格中的顶点数
Point3 * pt;                            // 3D 顶点组成的数组
int numNormals;                         // 网格中的法线数目
Vector3 * norm;                         // 法线数组
int numFaces;                           // 网格中的面片数
Face * face;                            // 面片列表
//…others to be added later
  public:
Mesh();                                 // 构造函数 r
  ~Mesh();                              // 析构函数
int readFile(char * fileName);          // 从文件中读入网格数据
//… 其他…
};
```

图 6.15 网格的数据结构

① 前面已经提到过基本类 Point3 和 Vector3 的定义，它们在附录 3 中也会提及。

数据结构 Face 基本上可以说是面片的顶点及其法向量组成的列表。在这里它被组织为索引对的数组：第 f 个面中第 v 个顶点的法线取值为 norm[face[f]. vert[v]. normIndex]。初看上去这显得很繁琐，但是这种索引机制很有条理也很容易操作，并且它还具有高效的优势可以快速随机地读取数组 pt[]。

例 6.2.3 正四面体的数据

图 6.16 展示了正四面体的数据结构，它的顶点为(0, 0, 0)、(1, 0, 0)、(0, 1, 0)和(0, 0, 1)。验证各个域中给出的值(我们稍后会讨论如何计算法向量)。

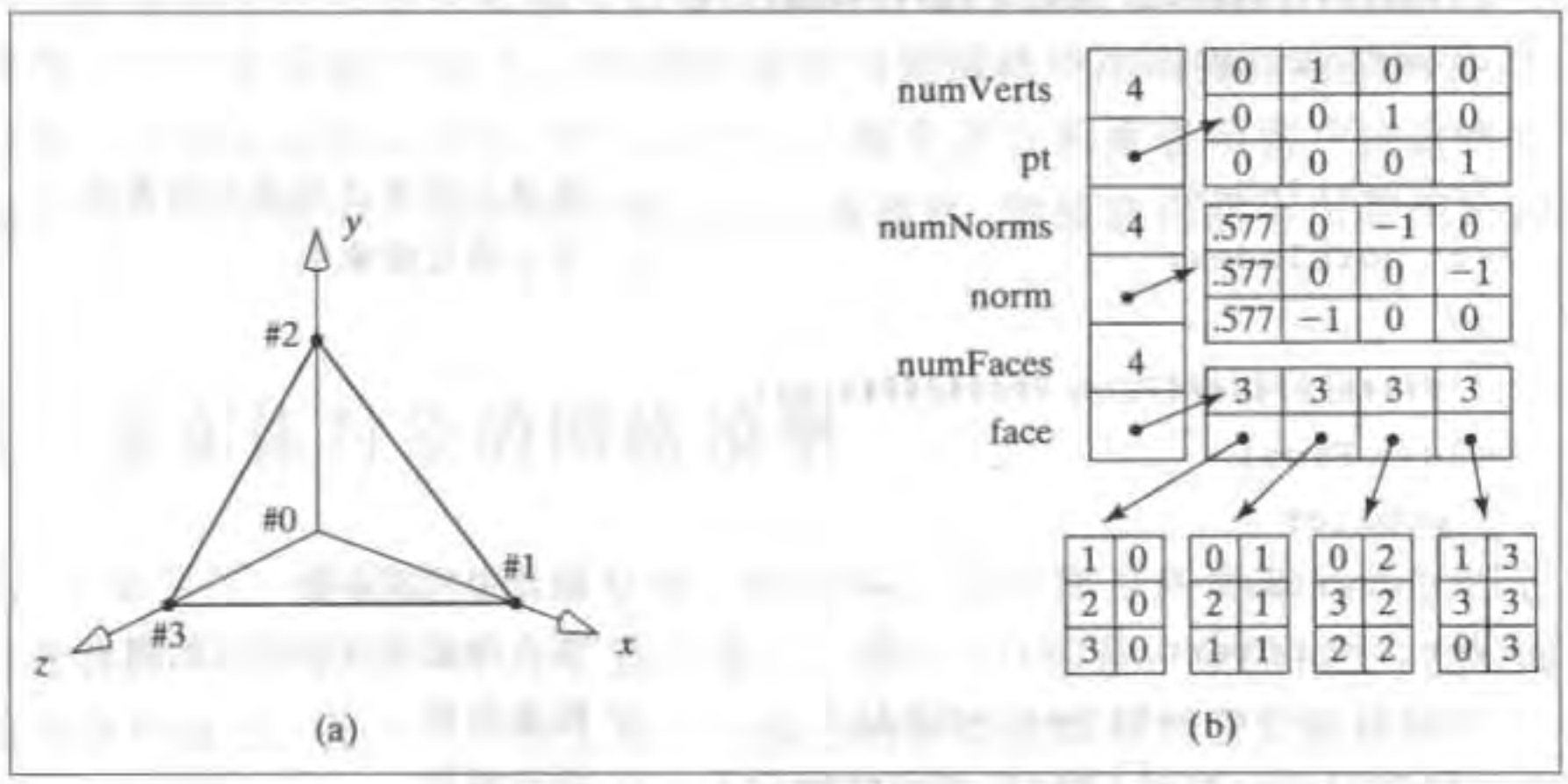


图 6.16 四面体的数据

首要的任务是开发出绘制这种网格对象的方法，即绘制出它的每一个面片。基于 OpenGL 的实现方法 Mesh::draw()必须要遍历网格对象的面片数组，然后把每个面片中所有的顶点和法线送到图形管线。在 OpenGL 中，使用 glNormal3f(m. x,m. y, m. z)[①]把法向量 **m** 指定给其后的顶点。所以 Mesh::draw()的基本流程为：

```
for(网格中的每个面片 f)
{
    glBegin(GL_POLYGON);
    for(面片 f 中的每个顶点 v)
    {
        glNormal3f(顶点 v 处的法线);
        glVertex3f(顶点 v 的位置);
    }
    glEnd();
}
```

图 6.17 是具体的实现方法。

① 为了进行合理的渲染，这些向量必须要归一化。这可以在 Init 函数中添加 glEnable(GL_NORMALIZE)完成，它会让 OpenGL 自动地对所有法向量实行归一化。

```
void Mesh:: draw()                          // 使用 OpenGL 绘制此网格
{
    for(int f=0; f<numFaces; f++)           // 绘制每个面片
    {
      glBegin(GL_POLYGON);
      for(int v=0; v<face[f].nVerts; v++)     // 遍历每个顶点.
      {
        int in=face[f].vert[v].normIndex ;    // 法线索引
        int iv=face[f].vert[v].vertIndex ;    // 顶点索引
        glNormal3f(norm[in].x, norm[in].y, norm[in].z);
        glVertex3f(pt[iv].x, pt[iv].y, pt[iv].z);
      }
      glEnd();
    }
}
```

图 6.17 使用 OpenGL 绘制一个网格

我们还需要构造特定网格的方法以及把事先定义好的网格从文件读入内存的方法。我们在案例 6.1 中会考虑文件的读写方法。下面会考察可以用网格存储的一系列有趣的形状,并且看一看如何生成它们。

使用 SDL 构造并绘制网格物体

在图形应用程序中使用第 5 章介绍的 SDL 语言读取网格的描述是很方便的。为了使用 SDL,只需要从 Shape 类(SDL 已经实现了)中派生出 Mesh 类然后添加方法 drawOpenGL()。本书的配套网站中有 Shape 类的详细描述以及 SDL 的一些支持类。这种思想在第 12 章中关于光线跟踪的讨论中变得尤为重要。

负责读取 SDL 文件的 Scene 类已经被设置为可以接受关键字 mesh,其后要跟上包含网格描述的文件名。因此,如果要构造并绘制 pawn 模型同时要伴随适当的平移与放缩,只需要如下操作:

```
push translate 3 5 4 scale 3 3 3 mesh pawn.3vn pop
```

(本书的配套网站有许多后缀名.3vn 的文件)文件 pawn.3vn 被用在 tellMaterialsGL()中;顶点列表、顶点法线列表和各种各样的反射系数将在第 8 章讨论。

6.3 多 面 体

我们可以方便地通过对网格数据进行约束,使得它们表示一个多面体。我们感兴趣的一大类实体对象实际上都是多面体,并且如果网格表示的仅仅是多面体的话,那么对其处理的算法就可以大大化简。

针对不同的情况,多面体的定义也稍有不同【Coxeter69, Hilbert50, Foley90】,我们遵

循如下定义：

定义：多面体是一个连通的网格，它由简单的平面多边形组成并且包围了一块有限体积的空间。

由定义可知多面体表示了单个的实体对象。这就要求：

(1) 每条边恰好由两个面共享。

(2) 在每个顶点处至少有 3 条边相交于此。

(3) 面之间不能相互贯穿：两个面要不根本不接触，要不只沿着公共边相接。

图 6.13 中的金字塔显然是多面体。多纳圈很明显形成一个封闭的空间，所以如果它的面片都是平面的话它就是多面体。但它不是简单多面体，因为它之中有一个洞。另外，它其中的两个面片自身也含有洞。“不可能物体”是一个多面体吗？为什么？如果忽略纹理面片(窗户和门)的话，仓房模型可以由两个多面体构成，一个主仓一个筒仓。

欧拉公式

欧拉公式(可以很容易证明，例如参见【Courant 和 Robbins，61，p236】)表明了一个简单多面体其面数、边数以及顶点数(分别用 F、E 和 V 表示)之间的基本关系：

$$V + F - E = 2 \tag{6.2}$$

例如，立方体的 $V = 8$，$F=6$，$E=12$。

通过 Schlegel 图可以很漂亮地揭示出多面体的结构。它是基于从其中一个面的中心外一点观察多面体的结果，参见图 6.18(a)。如此观察立方体得到的 Schelegel 图如图 6.18(b)所示。最前端的面显示为一个大多边形，它包围了其他的面。

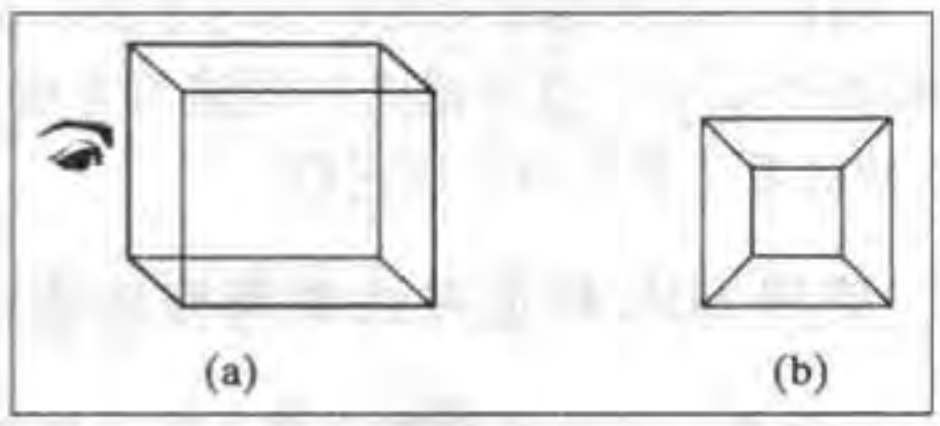


图 6.18 立方体的 Schlegel 图

图 6.19 是一些其他的例子。图 6.19(a)是图 6.13 中金字塔的 Schlegel 图，图 6.19(b)图和图 6.19(c)图是仓房的两张不同的 Schlegel 图(离眼睛最近的是哪些面?)。

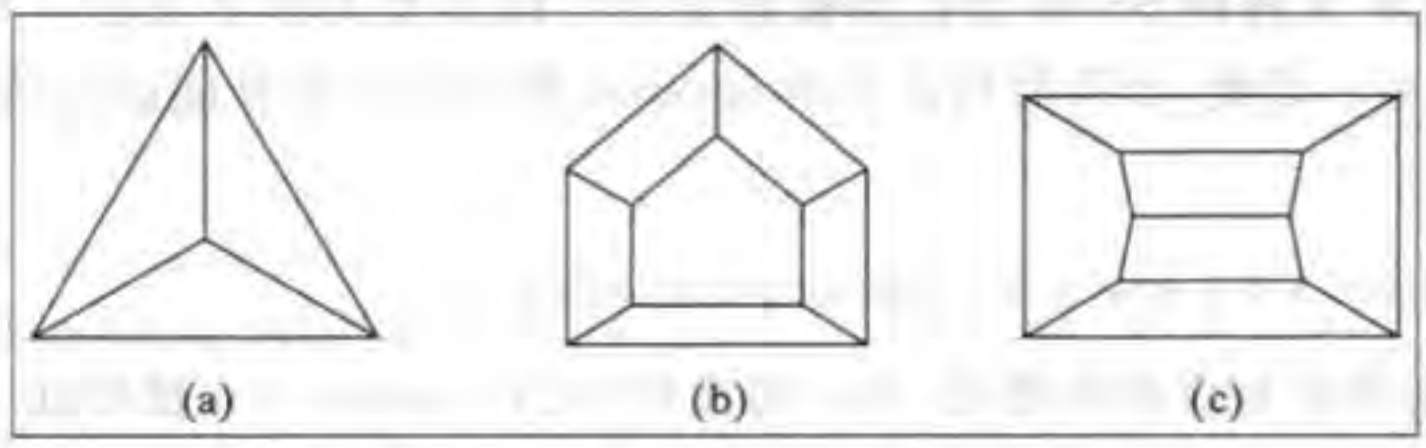


图 6.19 金字塔和仓房的 Schlegel 图

6.3.1 棱柱

棱柱是多面体的一种特例，由于它具有某种对称性因而很容易进行描述。像图 6.20 所示的那样，棱柱是把一个多边形沿着一条直线扫掠(或拉伸)的结果，把 2D 多边形转化为了 3D 多面体。图 6.20(a)中的多边形 P 沿着向量 $\mathbf{d}$ 扫掠构成了图 6.20(b)图中的多面体。当 $\mathbf{d}$ 垂直于 P 所在的平面时，棱柱为直角棱柱。图 6.20(c)展示了几个字母扫掠生成的棱柱。

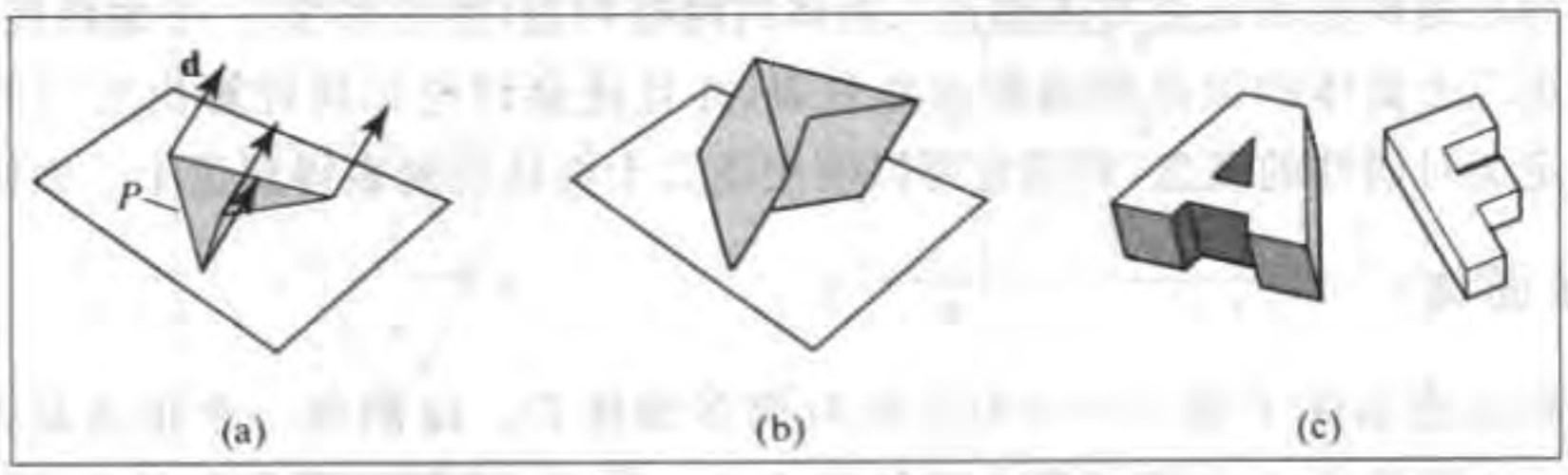


图 6.20　使用拉伸操作得到棱柱

练习

练习 6.3.1　构造棱柱的列表

构造图 6.21 中棱柱的顶点、法线与面片列表。假设棱柱的底面(面片 4)在 xy 坐标平面上,顶点 2 在 z 坐标轴且 $z=4$,顶点 5 在 x 坐标轴且 $x=3$。底面是一个等边三角形。

练习 6.3.2　构造网格

考虑图 6.13 中的多纳圈和仓房模型,对顶点指定编号然后构造其面片列表。

6.3.2　柏拉图实体

如果多面体的每一个面片都相同并且为正则多边形,那么此多面体称为正则多面体。上述对称性的要求十分苛刻,所以只存在 5 种这样的物体,即图 6.22 中所示的柏拉图实体[①]【Coxeter61】。柏拉图实体展示出完美的对称性并且拥有许多漂亮的性质。它们是计算机图形学领域有趣的研究对象,同时又经常在 CAD 实体建模中得到应用。

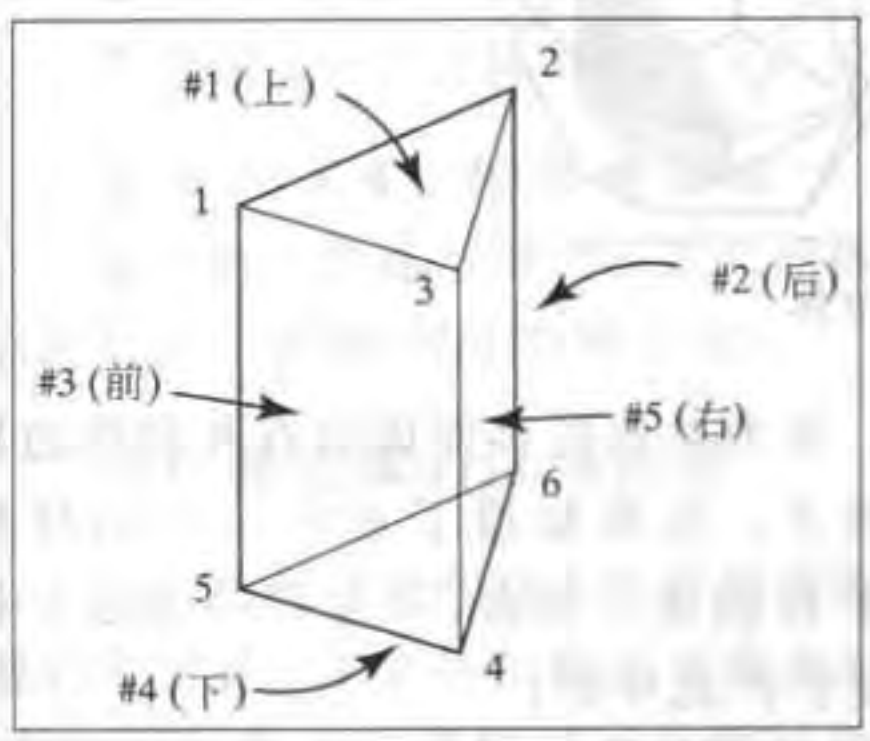


图 6.21　多面体物体范例

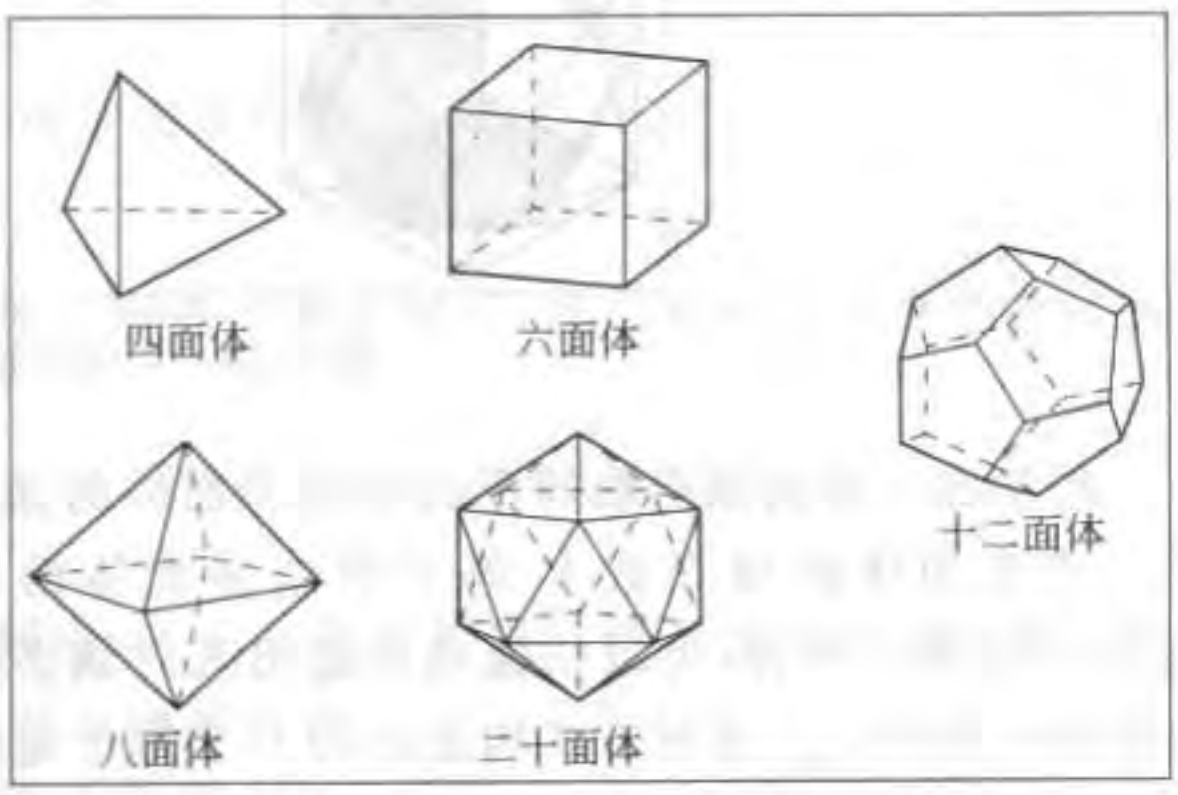


图 6.22　五种柏拉图立体

面片是等边三角形的柏拉图实体有 3 种,面片是正方形的柏拉图实体有 1 种,正十二面体的面片是正五边形。立方体和正八面体是正则棱柱体的变种。

---

① 是为了纪念柏拉图(427～347B.C.)而命名的,他在《蒂迈欧篇》对这些立体有所论述。但实际上它们在更早之前就被发现了,例如曾经从公元前 500 年的位于 Padua 的伊特鲁里亚废墟中发现过正十二面体的小物件。

可以很容易地构造出正立方体和正八面体的网格列表(参见练习)。下面将展示如何构造正四面体和正二十面体的顶点列表和面片列表,并且还会讨论如何计算出它们的法线列表。下一小节会定义对偶性的概念,利用它可以使用正二十面体的列表得到正十二面体的列表。

对偶多面体

每一个柏拉图实体 P 都有一个对应的对偶多面体 D。D 的每一个顶点是 P 的面片中心,所以 D 的边连接了 P 中相邻两个面片的中心。图 6.23 展示了每种柏拉图实体的对偶多面体,对偶多面体内嵌在原柏拉图实体之中。

- 正四面体的对偶还是正四面体。
- 正立方体与正八面体互为对偶。
- 正十二面体与正二十面体互为对偶。

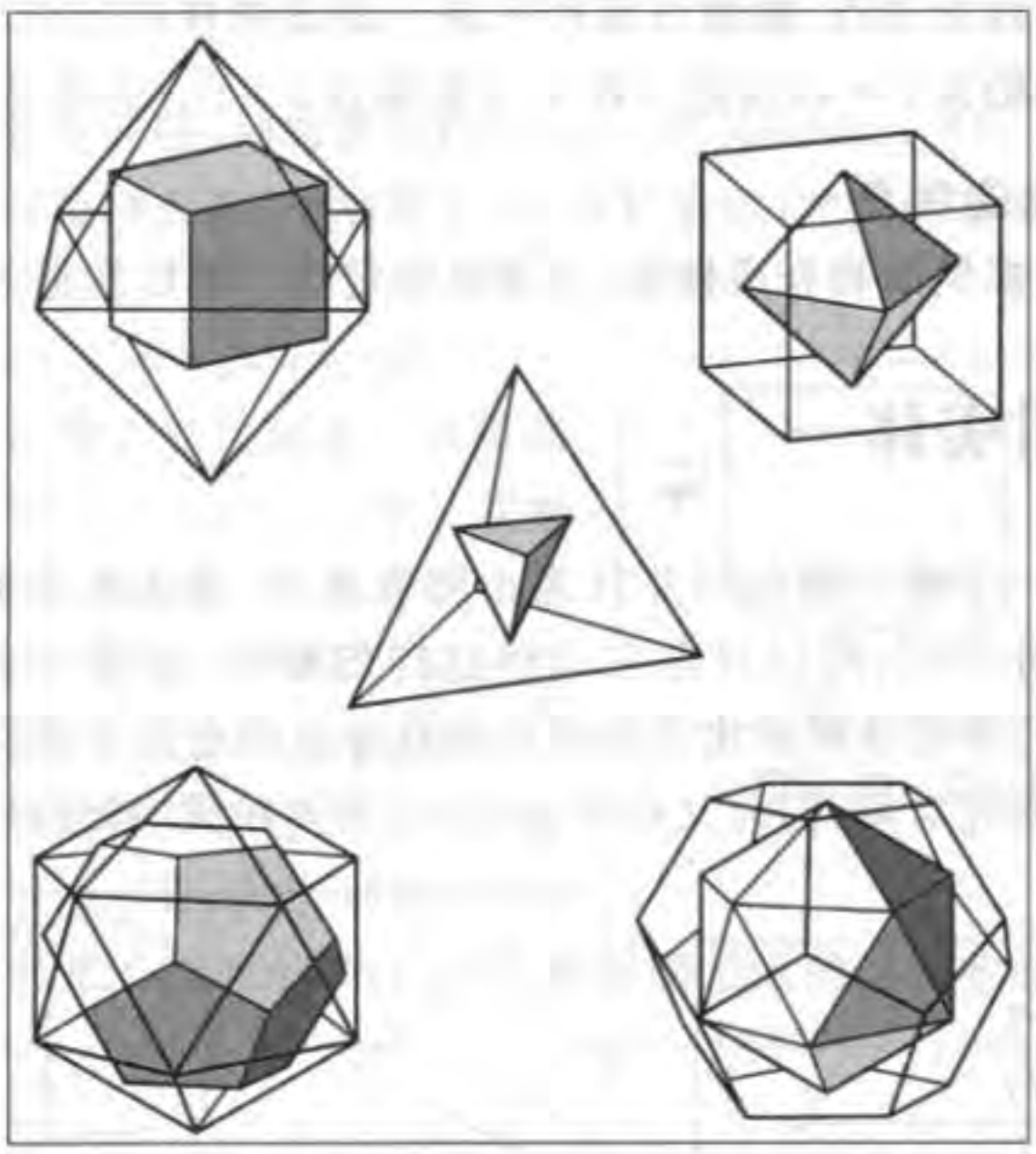


图 6.23 对偶柏拉图立体

柏拉图实体的顶点数等于其对偶多面体的面数。互为对偶的多面体具有相同的边数 E,一个多面体的顶点数 V 等于对偶多面体的面数 F。如果知道了其中一个柏拉图实体 P 的顶点列表,可以快速地构造出其对偶多面体 D 的顶点列表,因为 D 中顶点 k 是 P 中面 k 的中心。这种方式构造出的 D 实际上是内嵌于 P 之中的。

为了给每个顶点和面片编上序号,我们使用一种称为模型(model)的方法,沿着物体的边将其裁开然后展开铺平,此时从外部就可以观察到物体全部的面了。图 6.24 是三种柏拉图实体的模型。然后正如我们所希望的,就可以对模型的每一个顶点和面片进行编号。但是我们将会看到,对偶多面体模型的编号工作必须要多加小心。考虑正立方体及其对偶正八面体。正立方体的面 4 分别被编号为 1、5、6、2 的顶点包围。借助对偶性正八面体的顶点 4 就应该被编号为 1、5、6、2 的面包围。注意到正四面体的自对偶性,包围面 k 的顶点编号应该和包围顶点 k 的面编号相同。

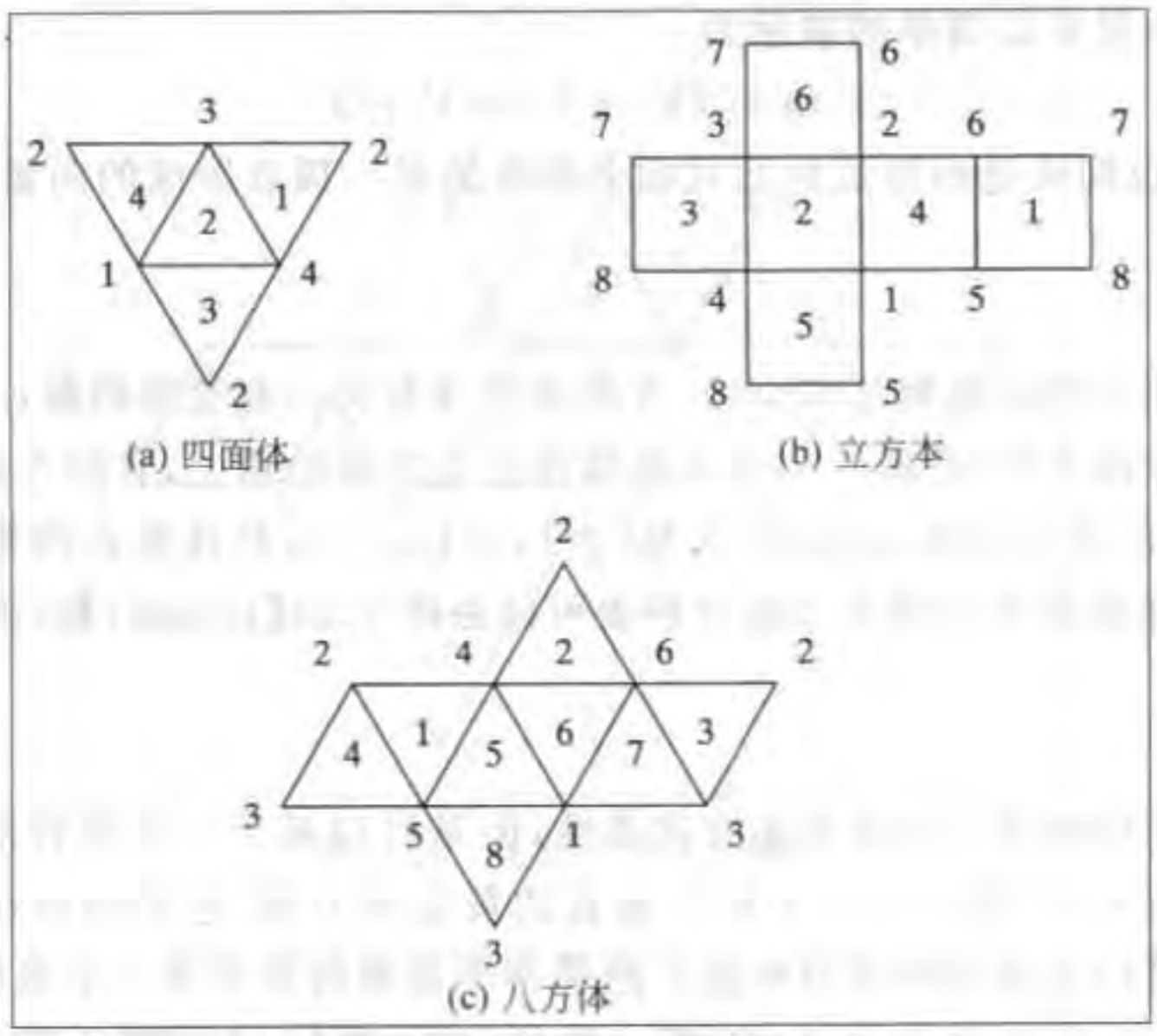


图 6.24　四面体、立方体和八面体的模型

正八面体的顶点 4 的位置是正立方体的面 4 的中心。回忆第 5 章的内容，一个面的中心是其顶点位置的平均。所以，如果知道了立方体面 4 的顶点 V_1、V_5、V_6 和 V_2，那么可以快速计算出正八面体的顶点 4。

$$V_4 = \frac{1}{4}(V_1 + V_5 + V_6 + V_2) \tag{6.3}$$

练习

练习 6.3.3　正八面体

考虑作为正立方体对偶的正八面体，构造其顶点列表和面片列表。

练习 6.3.4　验证对偶性

基于前一个练习得到的正八面体的顶点列表和面片列表，找出其对偶多面体并且验证它就是正立方体(尺寸有所变化)。

柏拉图实体的法线向量

如果我们希望构造出柏拉图实体的网格，我们必须计算出它每个面片的法向量。由于我们希望显示出来的柏拉图实体是拥有平坦的面并且棱边分明，所以应该使用面片的法向量而不是顶点的法向量。用通常的 Newell 法就可以实现上述目标，但是因为柏拉图实体具有高度的对称性，所以还有一个更加简便的方法。假设柏拉图实体的中心在原点，则它每个面片的法向量是由原点到这个面片中心的形成的向量，这个向量可以由面的所有顶点求均值得到，显然它是点集的一个仿射组合。图 6.25 是正八面体的

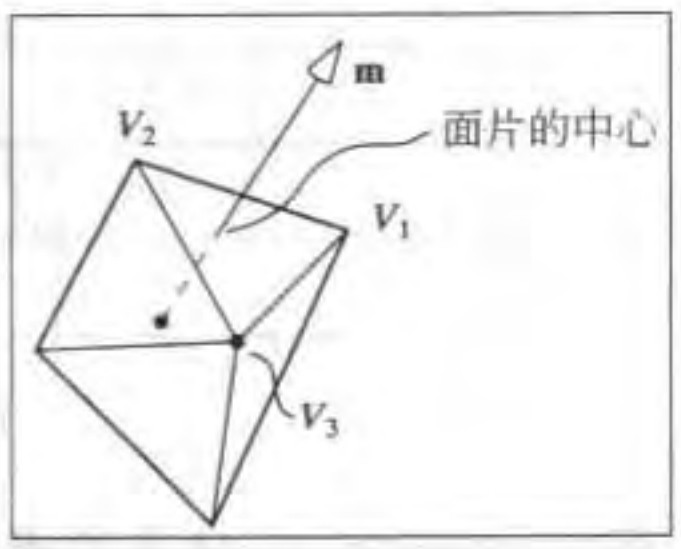


图 6.25　利用对称性计算面片的法线

情形。面片的法向量可以简单的表示为

$$\mathbf{m} = (V_1 + V_2 + V_3)/3 \tag{6.4}$$

(注意：这个向量也同样是由原点到其对偶多面体的某一顶点形成的向量。)

正四面体

正四面体的顶点列表显然与其位置、方向和尺寸有关。有意思的是，一个正四面体可以内嵌进一个正立方体之中(它的 4 个顶点坐落在正立方体的角上，它的 6 条边落在正立方体的面上)。考虑单位正立方体，它的顶点为(±1，±1，±1)，并且令正四面体的一个顶点为(1，1，1)。那么正四面体的顶点与面片列表可以由图 6.26【Blinn87】给出。

正二十面体

求出正二十面体的顶点列表更加有挑战性，但是可以基于一个独特的事实来让这项工作变得很简单。图 6.27 展示了三个相互垂直的黄金长方形，它们内嵌在一个正二十面体中。通过这张图可以直接得到顶点列表。把每一个黄金长方形与一个坐标轴对齐。为方便起见，规定其中一个长方形的长边沿着 x 轴从 −1 到 1，它的短边从 $-\phi$ 到 ϕ，这里的 $\phi=(\sqrt{5}-1)/2=0.618$是黄金分割比例。

顶点列表					面片列表	
顶点	x	y	z		Face#	顶点
0	1	1	1		0	1，2，3
1	1	−1	−1		1	0，3，2
2	−1	−1	1		2	0，1，3
3	−1	1	−1		3	0，2，1

图 6.26　四面体的顶点列表和面片列表

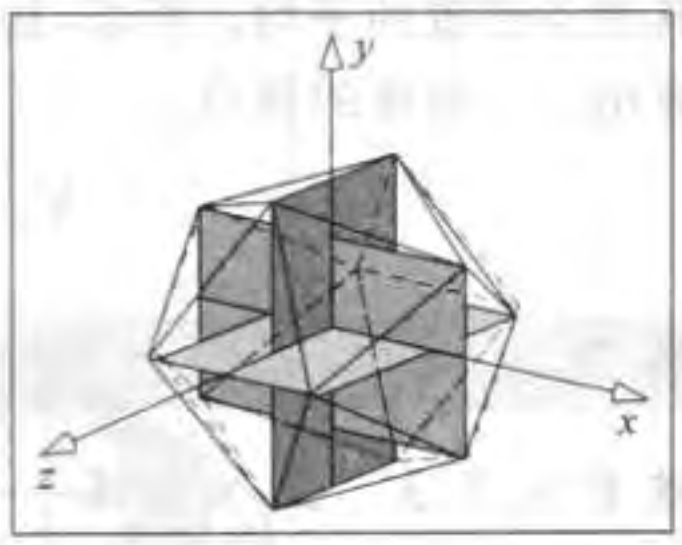


图 6.27　用黄金长方形定义正二十面体

然后就可以列出所有顶点的位置了，如图 6.28 所示。

顶点	x	y	z	顶点	x	y	z
0	0	1	ϕ	6	ϕ	0	1
1	0	1	$-\phi$	7	$-\phi$	0	1
2	1	ϕ	0	8	ϕ	0	−1
3	1	$-\phi$	0	9	$-\phi$	0	−1
4	0	−1	$-\phi$	10	−1	ϕ	0
5	0	−1	ϕ	11	−1	$-\phi$	0

图 6.28　正二十面体的顶点列表

图 6.29 是正二十面体的模型。可以直接得到它的面片列表(问题：面 8 的法向量是什么?)。

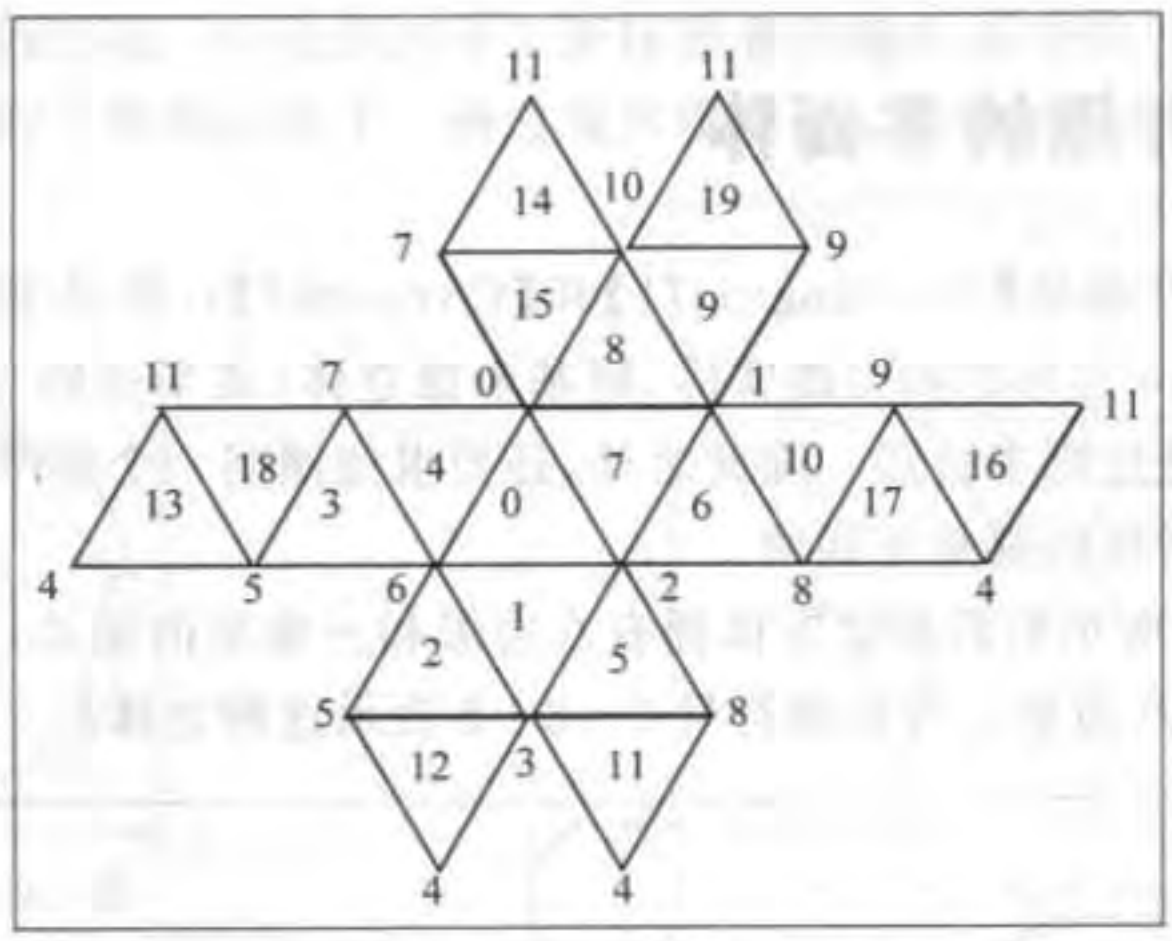


图 6.29　正二十面体的模型

有些人喜欢把正二十面体的模型稍微调整成图 6.30 所示的样子。通过它可以很明显地看出正二十面体由一个棱柱(阴影表示)和上下两个五棱锥组成。

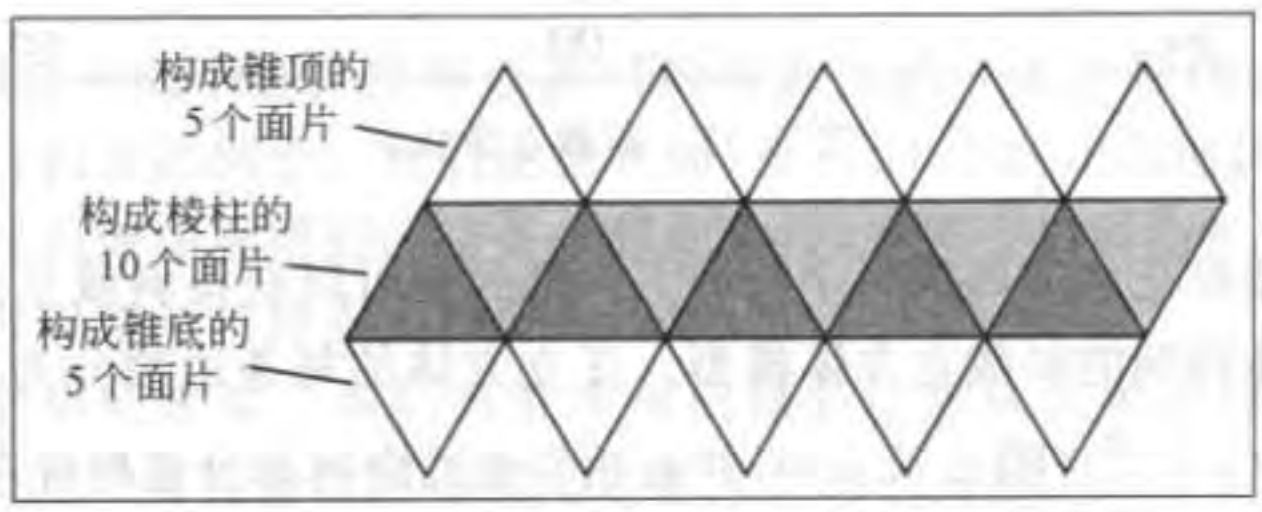


图 6.30　正二十面体是带有上下盖的棱柱

正十二面体

正十二面体是正二十面体的对偶，所以构建其列表的所有信息都可以通过正二十面体的列表得到。但是看一看正十二面体模型的展开图也是有益的，如图 6.31 所示。

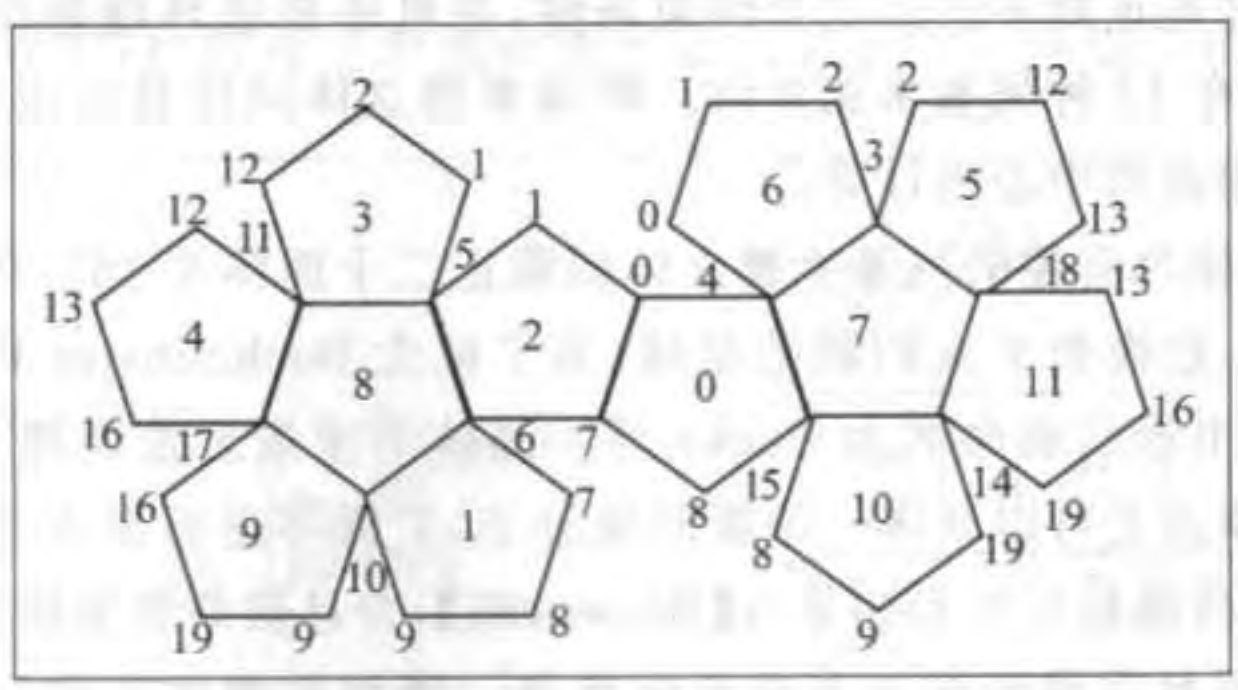


图 6.31　正十二面体的模型

再次利用对偶性，我们可以知道正十二面体的顶点 k 处在正二十面体的面 k 的中心：也就是面 k 的三个顶点的平均值。用这种方法可以很容易地计算出正十二面体的所有顶点。

6.3.3 其他有趣的多面体

多面体种类很多(参见【Wennington71】和【Coxeter63】),但是其中的一类特别有趣。不同于每一个面都是 n 边形的柏拉图实体,阿基米德立体(或半正则立体)的面片类型多于一种,但它们依旧还是正则多边形。除此之外,还要求按照同一个顺序遍历环绕每个顶点的面,这样得到的多边形序列都是相同的。

例如,图 6.32(a)所示的斜截立方体拥有八边形和三角形的面片,并且环绕每个顶点的是一个三角形和两个八边形。可以用符号 3·8·8 表示这种立体。

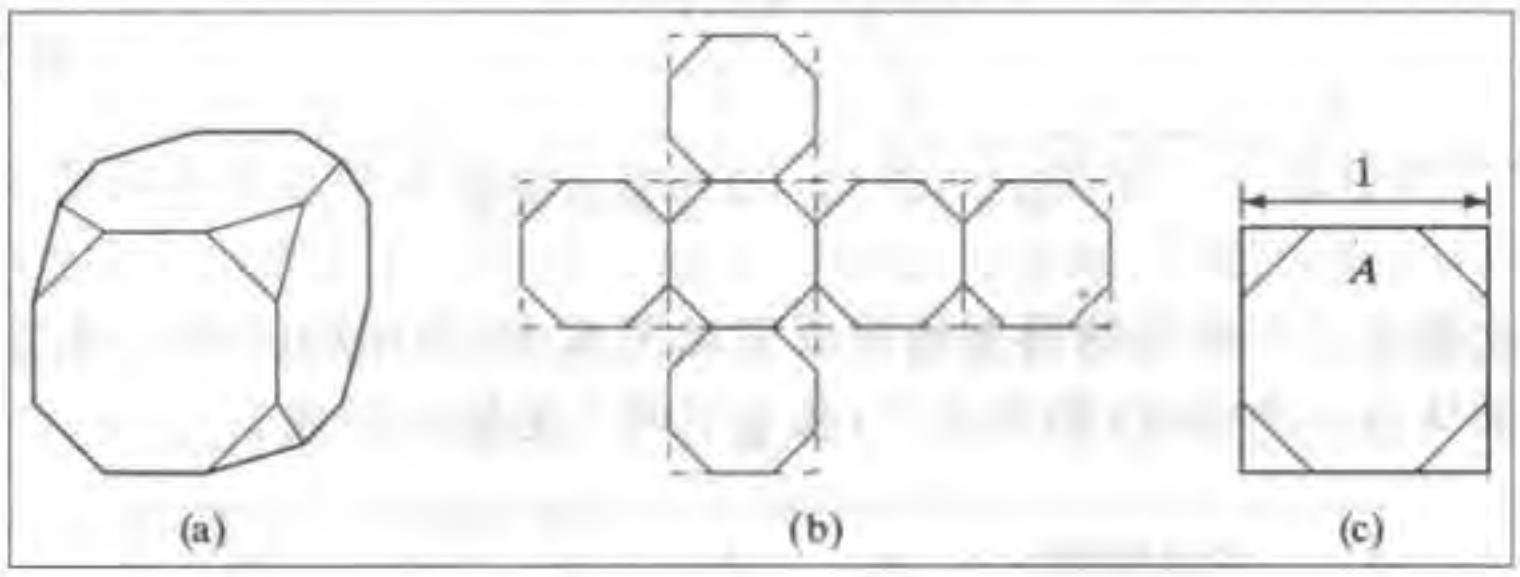


图 6.32 斜截立方体

使用适当的方法把正立方体的每个角切下一部分就可以获得斜截立方体。图 6.32(b)就是通过一个正立方体得到的斜截立方体模型。正立方体的每条边被分为三个部分,中间的部分 A 其长度等于 $1/(1+\sqrt{2})$(图 6.32(c)),并且相邻边的中间部分也相合在一起。因此如果正立方体的一条边的端点为 C 和 D,那么通过仿射组合可以得到两个新的顶点 V 和 W。

$$V = \frac{1+A}{2}C + \frac{1-A}{2}D \tag{6.5}$$

$$W = \frac{1-A}{2}C + \frac{1+A}{2}D \tag{6.6}$$

利用上述方法可以直接构造出斜截立方体的顶点与面片列表。

因为有每个面都是正则多边形这个约束条件,并且还要求环绕每个顶点的面的排列方式也一样,所以只存在 13 种阿基米德立体。阿基米德立体同样具有很强的对称性,它的每个法向量都可以使用面的中心来计算。

图 6.33 是一种相当有趣的阿基米德立体斜截正二十面体 $5 \cdot 6^2$,它是由正则六边形和正则五边形组成的。它类似于人们踢的足球,为了纪念 Buckminster Fuller 对其结构进行的研究工作,最近这个形状被命名为 Bucky 球。晶体学家最近发现把 60 个碳原子安排在斜截正二十面体的顶点上可以获得一种新的碳分子,它既不是石墨也不是金刚石。这种物质有很多特性,例如高温稳定性和超导性【Browne90】,并且被命名为福勒烯(Fullerene)。

构造 Bucky 球并且在图形设备上显示出来是一项很有趣的工作。为了构造它的顶点与面片列表,先画出图 6.29 中正二十面体的模型图,然后把每条边等分为三个部分。这样,在每条边上生成了两个新的顶点,它们的位置很容易计算。然后,按个人喜好对得到的 60 个新顶点进行编号构造出 Bucky 球的顶点列表。图 6.34 是正二十面体模型的一部分,

新生成的顶点用边进行连接。注意到正十二面体原有的每个面变成了六边形，原有的顶点在生成五边形面的过程中被裁减掉了。通过观察模型可以很容易地构造出面片列表。

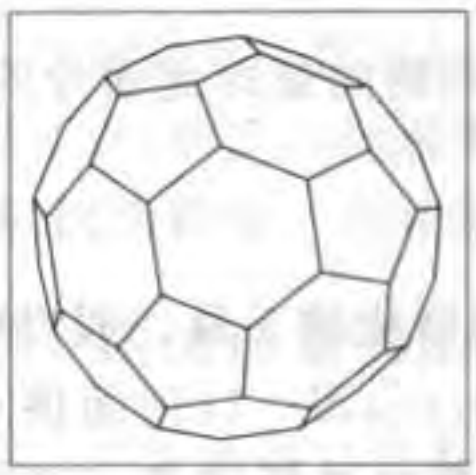


图 6.33 Bucky 球

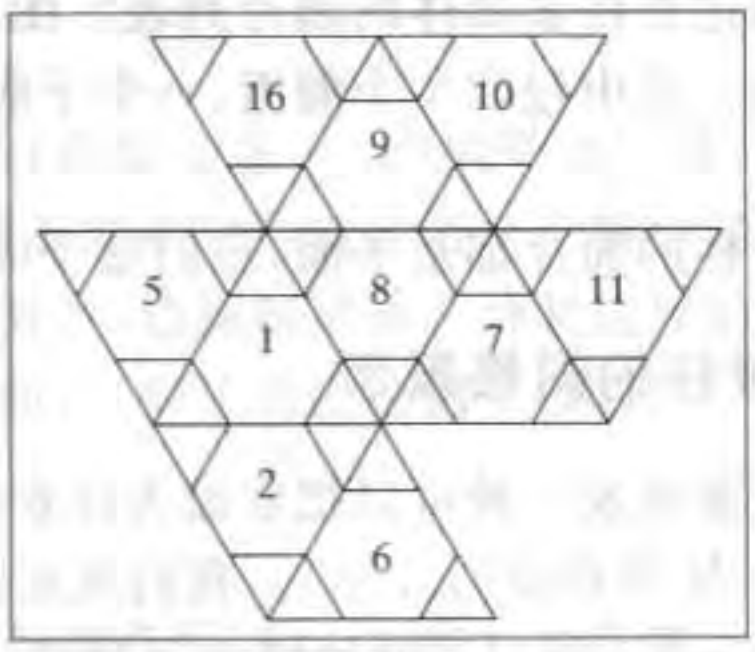


图 6.34 构造 Bucky 球

6.4 拉伸形体

通过把 2D 图形在空间中进行拉伸或者扫掠可以生成一大类形体。图 6.20 中的棱柱是一个沿着直线进行扫掠的例子。我们还会看到，图 6.24 中的正四面体和正八面体同样是在空间中以某种方式拉伸图形得到的结果。除此之外，如果我们稍加扩展一下拉伸的定义，那么旋转曲面同样可以通过拉伸多边形进行近似。

在本节中我们会研究通过一系列离散步骤扫掠多边形从而生成网格的几种方法。在 6.5 节还会开发类似的工具构造出网格用来近似光滑的扫掠体。

6.4.1 生成棱柱

下面开始探讨棱柱，它是多边形沿着一条直线扫掠得到的。图 6.35 是通过箭头状多边形 P 得到的棱柱，它在 xy 坐标平面有 7 个顶点。P 沿着 z 轴扫过一段距离 H 得到图 6.35(b)所示的箭头状棱柱(更一般的讲，还可以沿着向量 **d** 进行扫掠，如图 6.19 所示)。在 P 进行扫掠的过程中，P 的每一个顶点在 z 方向生成了一条边。由此又生成了 7 个顶

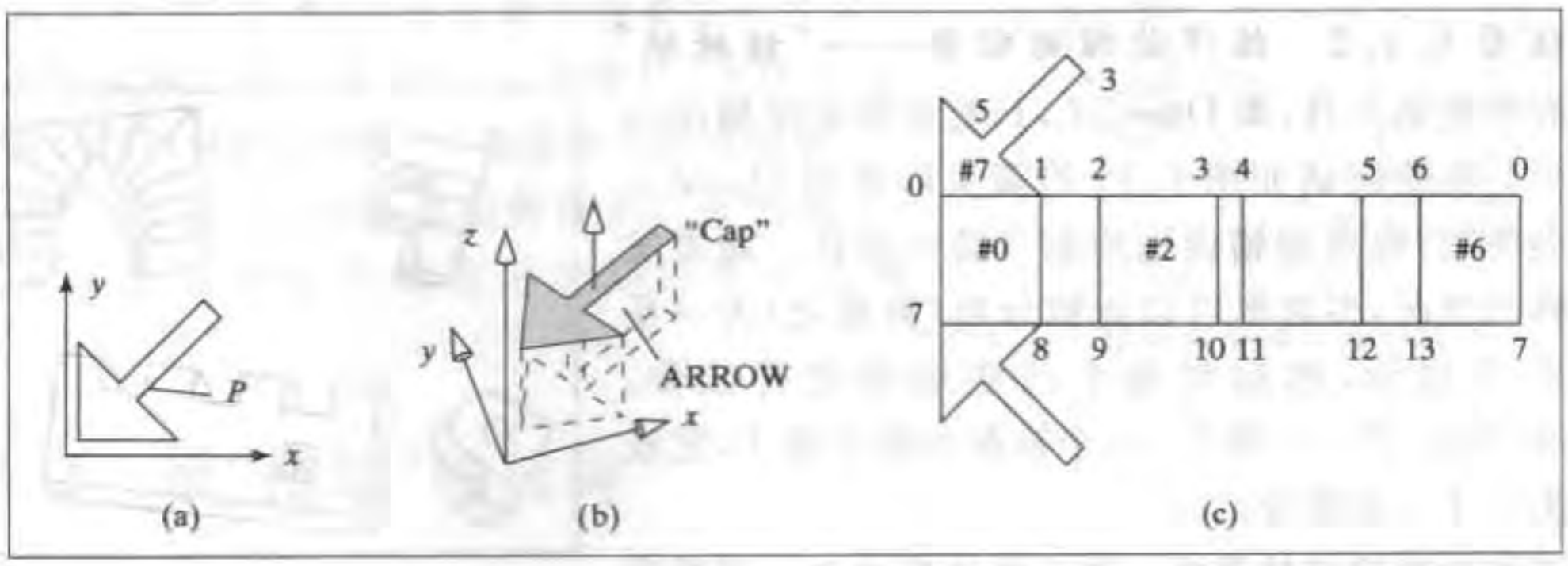


图 6.35 棱柱范例

点,所以棱柱共有 14 个顶点。这些顶点都是成对出现的:如果底面多边形有一个顶点为$(x_i, y_i, 0)$,则棱柱就还包含一个坐标为(x_i, y_i, H)的顶点。

如何描述棱柱多面体的面片列表?图 6.35(c)是把棱柱展开成模型的样子,可以看到它有 9 个面。其中包含 7 个侧面、一个下底面 P 和一个顶面。例如,面 2 是由顶点 2、9、10、3 定义的。

由于棱柱的面片都是平面,我们赋予面片中所有顶点以相同的向量即此面片的法向量。

构造棱柱的网格表示

我们需要开发一种可以把底面为任意多边形的棱柱进行网格化的工具。假设棱柱的底面多边形有 N 个顶点(x_i, y_i)。我们规定底面顶点的序号为 $0,\cdots,N-1$,顶面顶点的序号为$N,\cdots,2N-1$。所以在此种规定下顶点 i 和顶点 $i+N$ 有一条边进行连接。这样顶点列表就可以很容易地构造出来,这张表中包含有点$(x_i, y_i, 0)$和点(x_i, y_i, H),其中下标 i 为 $0,1,\cdots,N-1$。

构造面片列表的方法也同样很简单。我们先把侧面或外墙构造出来然后再添加顶面和底面。对于第 $j(j=0,\cdots,N-1)$个侧面,它的 4 个顶点的指标为 j、$j+N$、$next(j)+N$ 以及 $next(j)$。在这里当 j 等于 $N-1$ 时 $next(j)$等于 0,其他情况 $next(j)$等于 $j+1$。这样就可以保证 0 号顶点接在 $N-1$ 号顶点之后。用代码可以表示如下:

```
if (j<n-1)
  next=j+1;
else
  next=0;
```

把这样生成的每个面片插入面片列表。用先前介绍的 Newell 法可以方便地计算每个面片的法向量。最后我们构造底面与顶面并把它们插入面片列表。案例 6.2 还会提供构造棱柱网格更多的细节。

练习

练习 6.4.1 构造网格模型

构造图 6.35 中棱柱的网格模型。以你喜欢的方式给箭头图形指定坐标,然后在 z 轴进行扫掠,计算所有顶点的位置。

练习 6.4.2 拉伸棱柱的组合——"铺砖块"

有些渲染工具,如 OpenGL,只能可靠地绘制出凸多边形。在绘制诸如图 6.35 的箭头形状时 OpenGL 可能会失败,也可能错误地绘制出箭头形状。如果发生这样的情况,多边形可以先被分解(离散化)为一系列的凸多边形,然后对每个凸多边形进行拉伸。图 6.36 展示了一个例子:一个被拉伸的字母 F,它被分解为 4 个凸的部分。

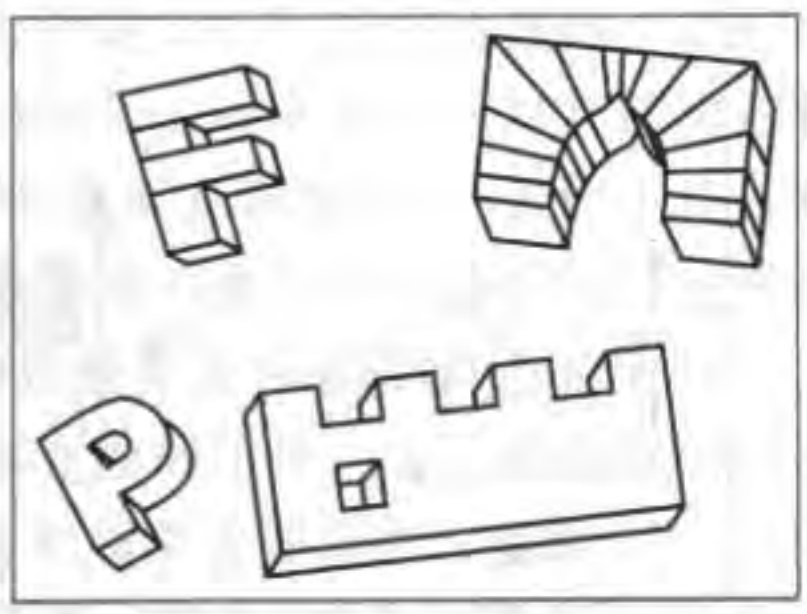


图 6.36 基于凸棱柱集合的拉伸体

诸如此类的棱柱是由一些凸棱柱组成的。有的凸棱柱之间彼此相接,共用相邻面的全部或者一部分。

由于顶点位置是通过高精度计算得到的，相接面之间的缝隙通常是无法观察到的。

对于这一类的形状，需要一种方法，可以通过棱柱组成的数组构造出网格，也就是

```
void Mesh::makePrismArray(…)
```

它以构成底面的某(凸)多边形列表作为参数(假设其在 xy 坐标平面)，除此之外，还可以使用一个向量 **d** 来描述拉伸的方向和程度。顶点列表应该包含每一个棱柱的顶面以及底面多边形顶点，而面片列表要存储每一个棱柱的侧面、底面和顶面。绘制这样的网格浪费了一些开销，因为即使一些相接的侧面不会被观察到，它们还是被绘制了(两次)。

特例：四边形条带的拉伸体

这是一类简单但又很有趣的棱柱，它的构造和操作可以设计得十分高效。这一类的棱柱的底面是四边形条带。四边形条带是四边形排列成链状而成的，类似排成一排的砖块，这样相邻的面片就完全对齐了，如图 6.37(a)。请回想图 2.39 的内容，四边形条带是 OpenGL 基础的几何图形。四边形条带可以由一列顶点描述：

$$\text{四边形条带} = \{p_0, p_1, p_2, \cdots, p_{M-1}\} \tag{6.7}$$

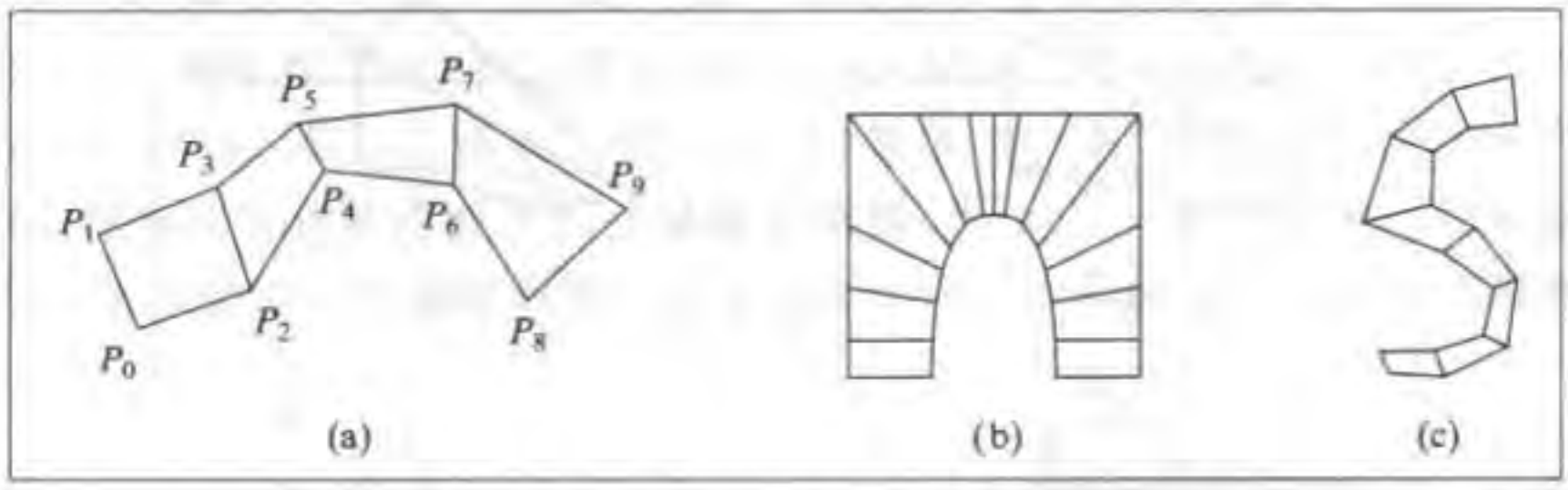


图 6.37　四边形条带与用四边形条带构造的棱柱

顶点是要成对出现的，奇数号顶点组成四边形条带的一条边，偶数号顶点组成其另一条边。每个“砖块”重复使用先前的一对顶点，再从列表中加入两个新顶点。并不是每个多边形都可以表示为四边形条带(图 6.36 中哪个多边形不是四边形条带？字母表中的哪个字母可以表示为四边形条带？)。

当用网格表示一个四边形条带的拉伸体时，顶点列表只有 $2M$ 个顶点，并且面片列表也只包含形成外表的面片。总共有 $2M-2$ 个面片(为什么？)。所以在绘制网格的过程中，没有多余的面被绘制。

图 6.38 是四边形条带拉伸体的有趣的例子。案例 6.4 还会更加详细地讨论如何生成这种网格。

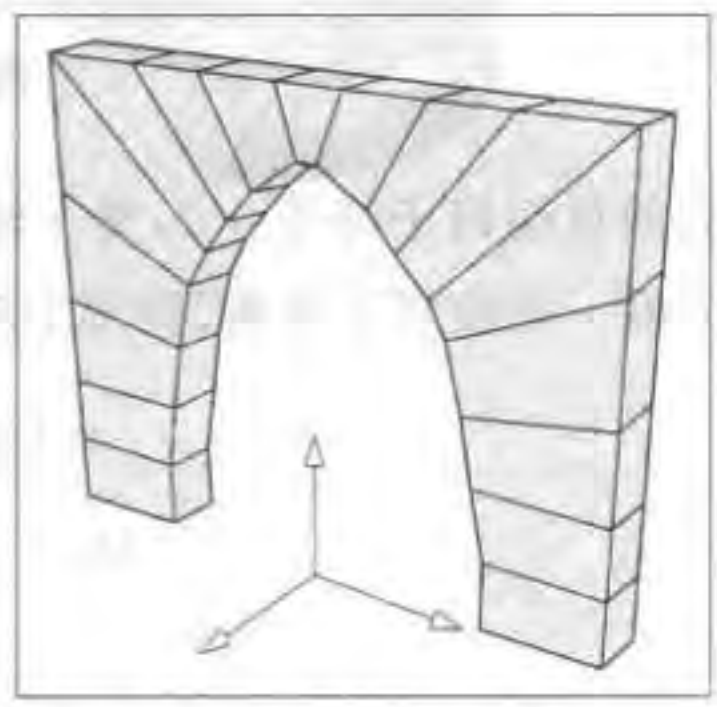


图 6.38　把四边形条带进行拉伸——拱门

6.4.2　带“扭结”的拉伸

到目前为止，拉伸还仅仅是把底面多边形平移到一个新位置从而得到顶面多边形。可

以很容易地对这种方式进行扩展来产生更加广泛的形体;可以通过对底面多边形进行放大或缩小甚至旋转来生成顶面多边形。特别地,如果底面多边形为 P,其顶点为$\{p_0, p_1, p_2, \cdots, p_{N-1}\}$,而顶面多边形的顶点为

$$P' = \{M_{p_0}, M_{p_1}, \cdots, M_{p_{N-1}}\} \tag{6.8}$$

这里的 M 是一个 4×4 的仿射变换矩阵。图 6.39 展示了一些例子。

图 6.39(a)和图 6.39(b)显示的是棱锥,或者说是一端逐渐变细的柱面(也可以叫截锥),它们的顶面是底面的缩小版本。变换矩阵可以如下表示

$$M = \begin{bmatrix} 0.7 & 0 & 0 & 0 \\ 0 & 0.7 & 0 & 0 \\ 0 & 0 & 1 & H \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

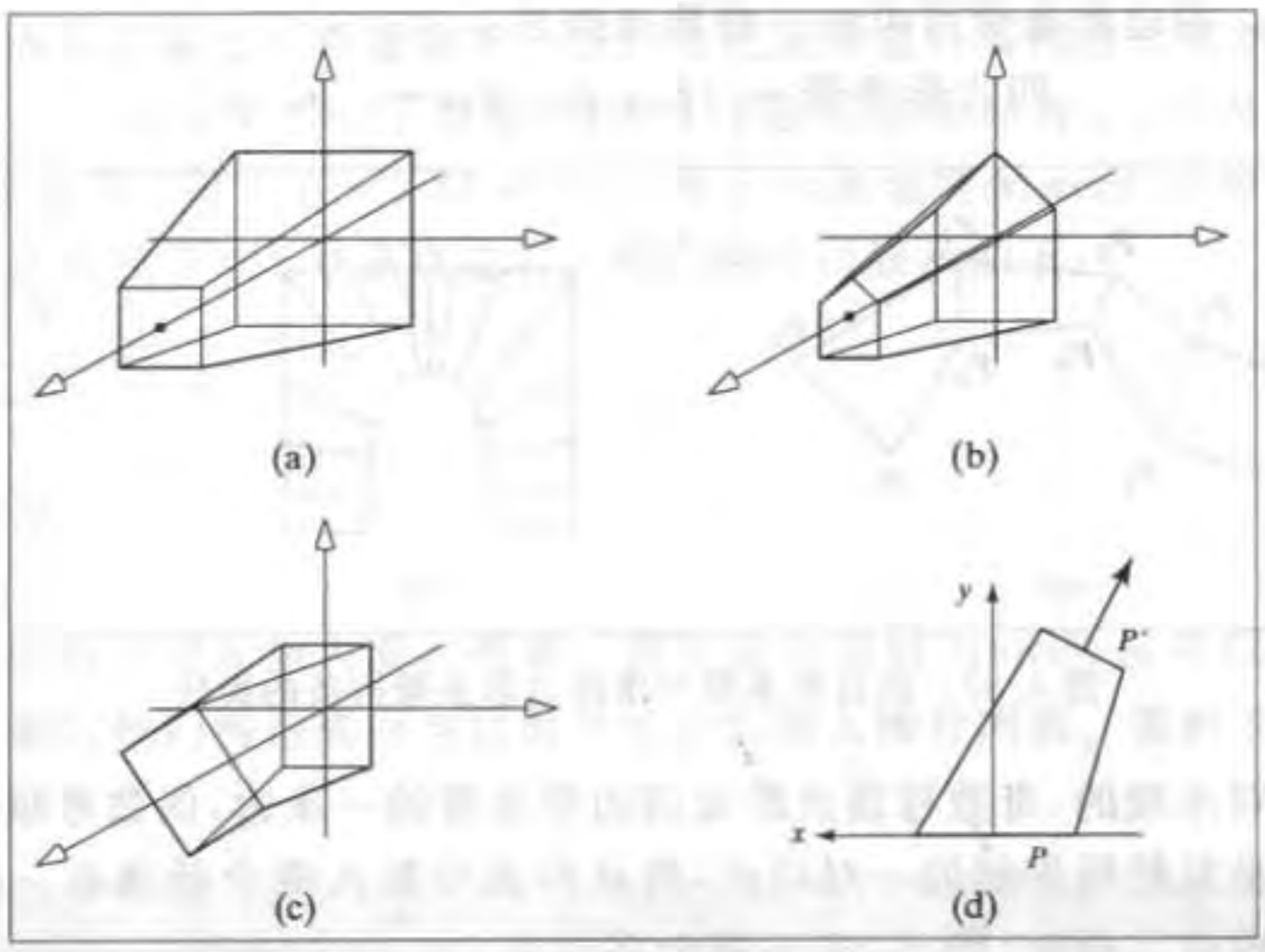


图 6.39 锥体和扭转的棱柱

它的放缩因子等于 0.7,沿 z 轴平移距离为 H。图 6.39(c)展示的柱面,其顶面在平移之前先沿 z 轴旋转了 θ 角,使用矩阵可以表示如下

$$M = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 & 0 \\ -\sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & H \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

图 6.39(d)利用截面图展示了顶面 P'在移到相应位置之前如何进行的旋转。

上述这些棱柱的构造方法与 M 代表的仅仅是平移操作来构造棱柱的方法一样简便:面片列表与原方法得到的一模一样,只是顶点位置(还有法向量值)有所改变。

练习

练习 6.4.3　一端逐渐变细的柱面

对于一端逐渐变细的柱面详细描述如何构造其顶点、法线以及面片列表。它的底面与顶面都是正五边形，但是顶面是底面的一半大小。

练习 6.4.4　把四面体当作一端逐渐变细的柱面

请描述如何把一个四面体建模为底面是三角形的一端逐渐变细的柱面。以这种方式构造四面体的网格是否高效？

练习 6.4.5　构造逐段拉伸的形体——管状体与蛇形体

有一类丰富的形体可以通过一系列的拉伸操作构造出来，每一次拉伸都使用其自己的变换，然后把它们端面相接形成一个管状体。图 6.40(a)是把一个正方形 P 拉伸 3 次得到的管状体，每一次拉伸操作的方向、放缩和扭曲程度都不相同。第一段拉伸部分其两端的多边形为 M_0P 和 M_1P，其中初始矩阵 M_0 给出了管状体起始面的位置和朝向。第二段其两端的多边形为 M_1P 和 M_2P，后面的依此类推。我们把那些被变换了的多边形称为管状体的“腰”。在此例中，网格包含有 16 个顶点，即 M_0p_0、M_0p_1、M_0p_2、M_0p_3、M_1p_0、M_1p_1、M_1p_2、M_1p_3，…，M_3p_0、M_3p_1、M_3p_2、M_3p_3。图 6.40(b)展示了一条“蛇”，之所以这样称呼是因为通过矩阵 M_i 的作用管状物逐渐生长并且越来越细，形成了一条蛇的身子和头部。请留意蛇形体其每一部分被渲染的明暗效果。将在第 8 章讨论如何使用算法来进行着色处理。

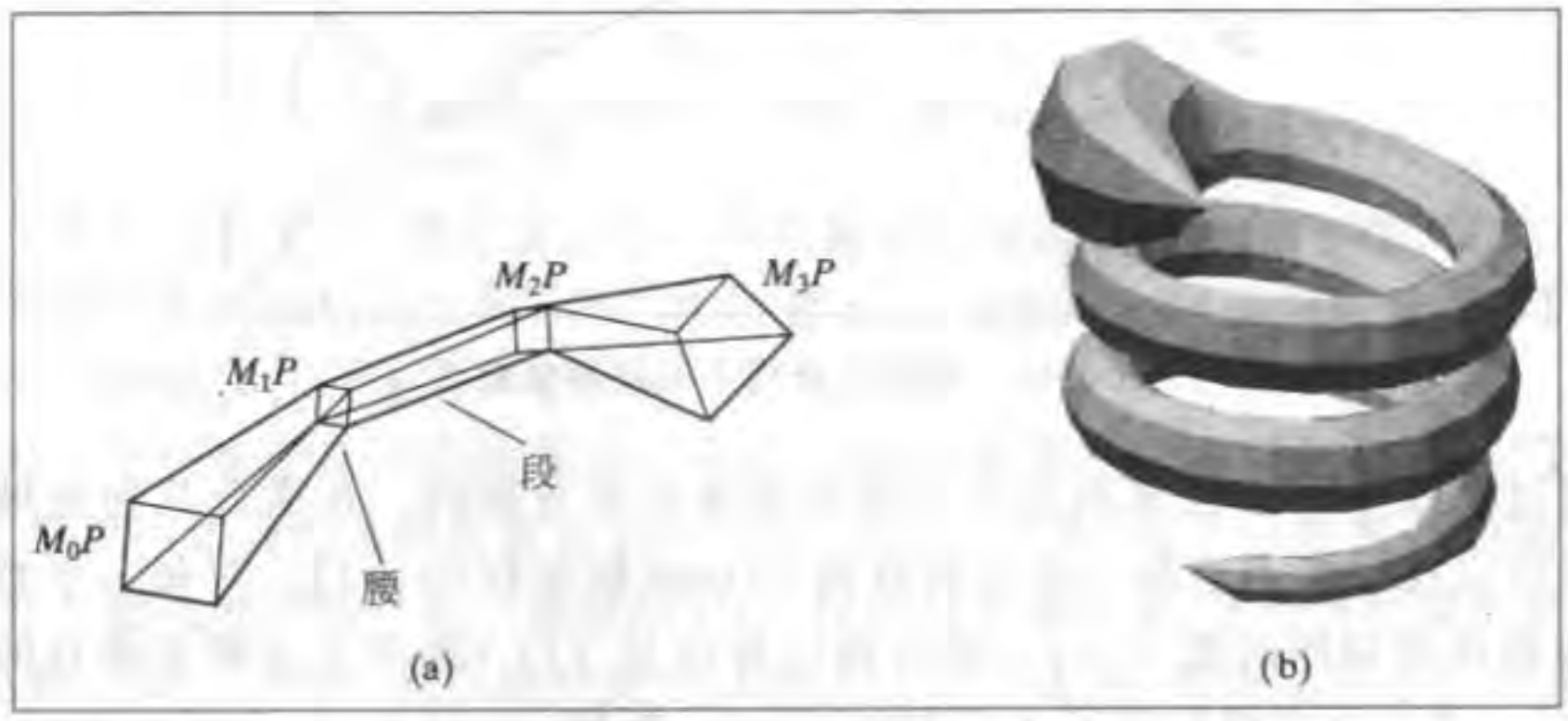


图　6.40

(a) 管状体；(b) 逐段拉伸而成的蛇形体

基于 3D 曲线的管状体——Frenet 标架

如何设计有趣且实用的管状体和蛇形体呢？可以使用手工方式选择各个独立的矩阵 M_i，但是这个方法太笨拙了。在 CAD 领域中，完美的境界是使此种任务自动化。不难想象，管状体是包裹在一根空间中起伏荡漾的曲线周围，这根曲线称为管状物的脊线。我们将用参数化方式把这根曲线表示为 $C(t)$。例如，图 6.41(a)所示的螺旋线(回想 3.8 节)可以用参数表示为

$$C(t) = (\cos(t), \sin(t), bt) \tag{6.9}$$

其中,b 为某个常数。

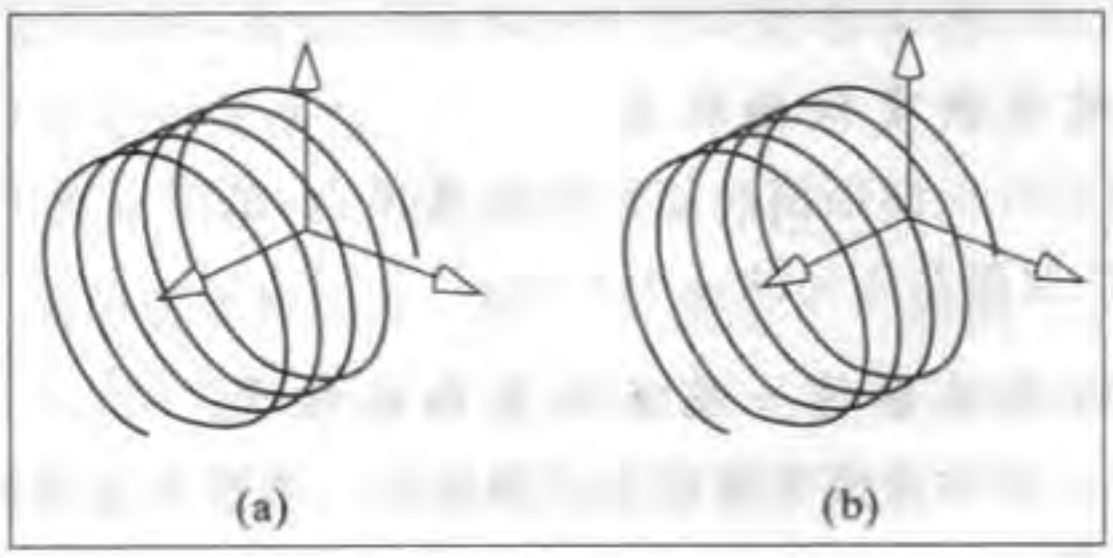


图 6.41 一对螺旋线

为了构造出管状体的腰多边形,我们对 $C(t)$ 在一些 t 值 $\{t_0, t_1, \cdots\}$ 进行采样,然后在点 $C(t_i)$ 与曲线垂直的平面内构造出被变换过的多边形,如图 6.42 所示。为便于理解,可以想象在脊线的每一个指定点上建立起一个局部坐标系:坐标系的 z 轴沿着曲线的方向,x 和 y 轴指向于 z 轴垂直的方向(并且相互正交)。腰多边形被置于局部坐标系的 xy 坐标平面内。我们所需要的就是一种简单易懂的方法来计算出每个腰多边形的顶点。

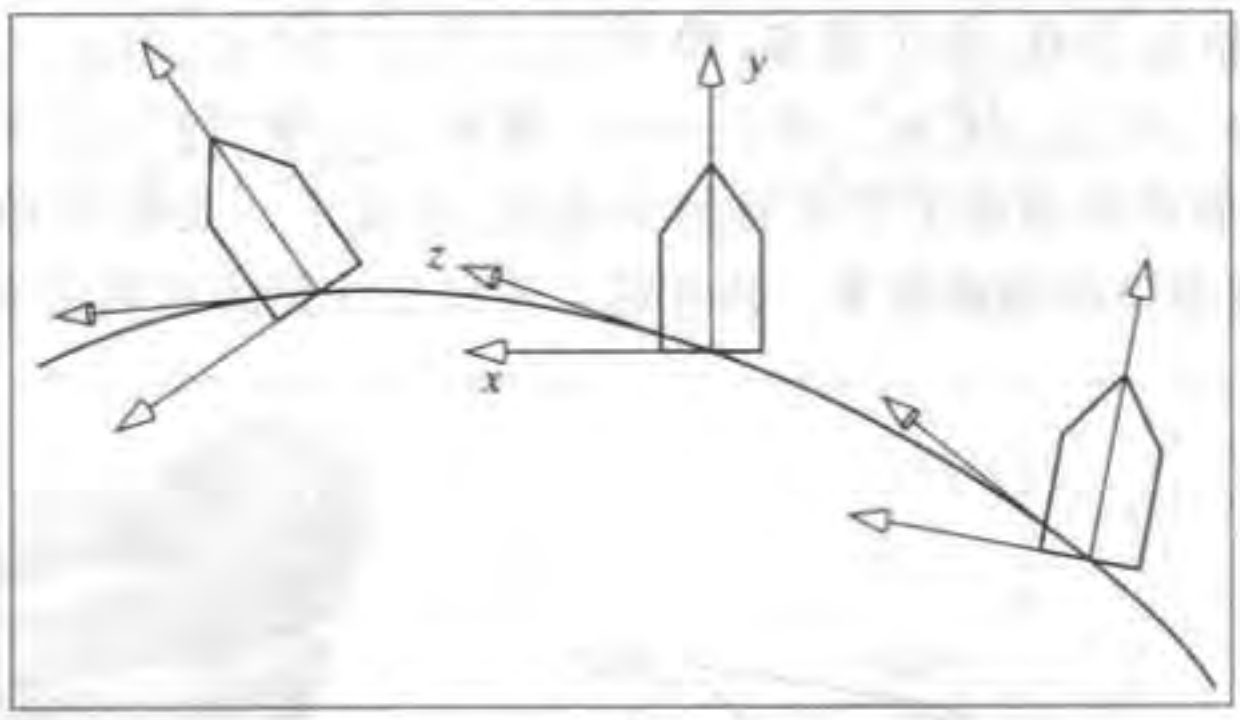


图 6.42 构建在脊线上的局部坐标系

仅仅通过曲线 $C(t)$ 自己来决定出局部坐标系是最方便的。在微分几何领域中有一个熟知的方法可以在脊线上的每一点上构造出 Frenet 标架【Gray93】。对每一个感兴趣的值 t_i,计算出与曲线相切的向量 $T(t_i)$。然后再计算出与 $T(t_i)$ 垂直且又相互垂直的两个向量 $a(t_i)$ 与 $b(t_i)$。这三个向量构成了 t_i 点处的 Frenet 标架。

一旦计算得到了 Frenet 标架,就可以很容易地找到变换矩阵 M 通过它把管状体的底面多边形变换到相应的位置与朝向。这个变换把世界坐标系转换到新的坐标系。(理由与练习 5.6.1 很相似,它是把摄像机坐标系转换到了世界坐标系。)矩阵 M_i 必须要把 $\mathbf{i}$、$\mathbf{j}$、$\mathbf{k}$ 相应地变换为 $\mathbf{b}(t_i)$,$\mathbf{a}(t_i)$,$\mathbf{T}(t_i)$,同时还要把世界坐标系的原点变换为脊线上的点 $C(t_i)$。因此用齐次坐标表示这个矩阵时它的列向量直接就是 $\mathbf{N}(t_i)$,$\mathbf{B}(t_i)$,$\mathbf{T}(t_i)$ 和 $C(t_i)$。

$$M_i = (\mathbf{N}(t_i) \mid \mathbf{B}(t_i) \mid \mathbf{T}(t_i) \mid C(t_i)) \tag{6.10}$$

构造 Frenet 标架

曲线上每一点处的 Frenet 标架取决于曲线弯曲扭转的方式。它是由 $C(t)$ 的某些导数计算出来的,所以如果可以得到这些导数的话就能很容易生成 Frenet 标架。

特别地，如果 $C(t)$ 可微的话，可以对它计算一阶微分从而得到曲线每一点处的切向量 $\dot{C}(t)$（假设 $C(t)$ 拥有分量 $C_x(t)$、$C_y(t)$、$C_z(t)$，那么导数可以简单地表示为 $\dot{\mathbf{C}}(t)=(\dot{C}_x(t),\dot{C}_y(t),\dot{C}_z(t))$。这个向量指向曲线在 t 值时的前进方向，也就是曲线的切线方向。我们对其进行归一化得到 t 点处的单位切向量。例如，螺旋线等式(6.9)的单位切向量可以如下表示

$$\mathbf{T}(t)=\frac{1}{\sqrt{1+b^2}}(-\sin(t),\cos(t),b) \tag{6.11}$$

t 在 4 个点处取值得到的切向量如图 6.43(a)所示。请设想一下当沿着曲线运动时标架是如何转动的。

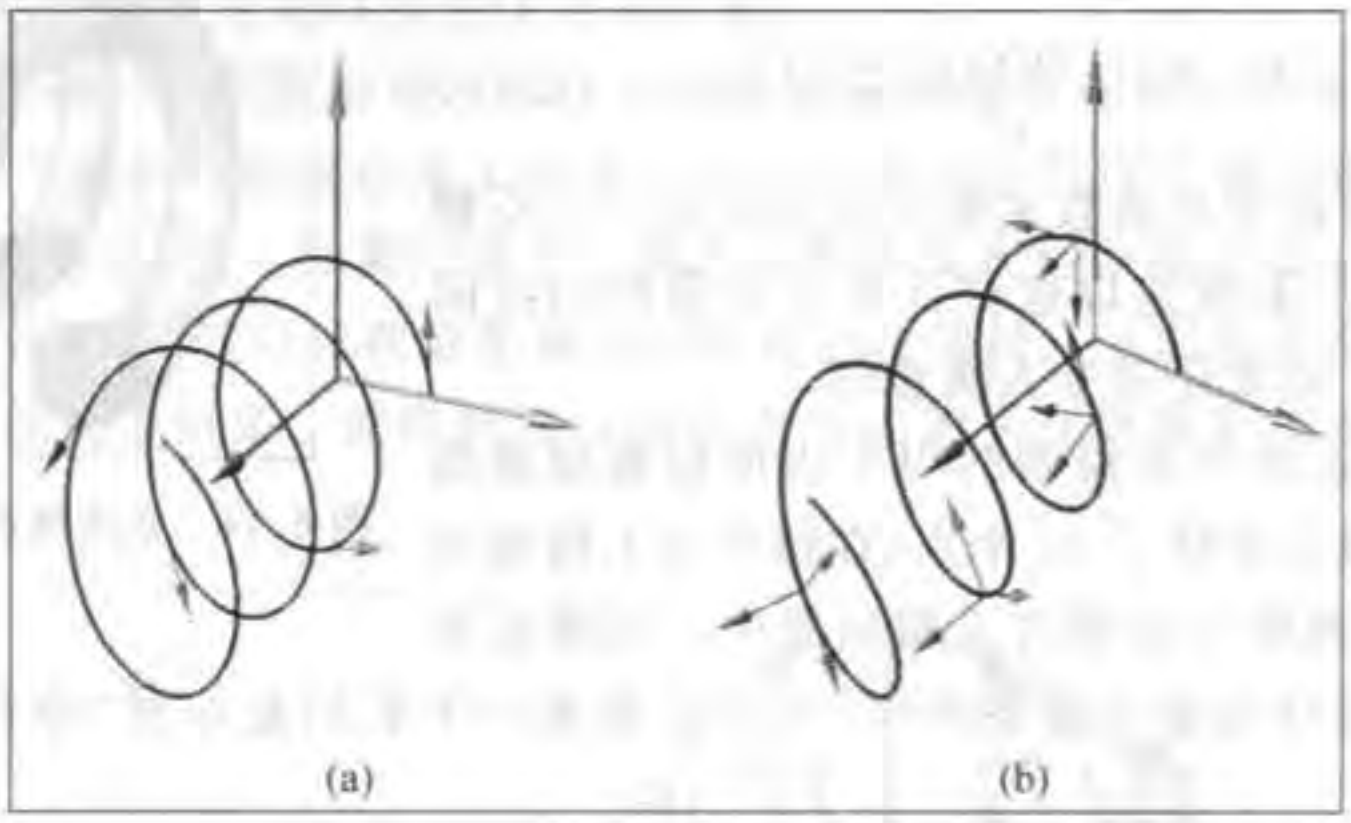


图　6.43

(a) 螺旋线的切线；(b) 不同 t 处的 Frenet 标架

已知一个单位向量 $\mathbf{T}(t)$，要计算出另一个向量 $\mathbf{a}(t)$ 使得其与 $\mathbf{T}(t)$ 正交。有了这两个向量就很容易把坐标系彻底构造出来：第三个向量 $\mathbf{b}(t)$ 与前两个向量都正交，这可以通过求两个向量的叉积得到，$\mathbf{b}(t)=\mathbf{a}(t)\times\mathbf{T}(t)$。

怎样找到一个正交于 $\mathbf{T}(t)$ 的向量呢？构造 Frenet 标架的经典方法【Gray93】是计算一个二阶导数，然后用适当的叉积运算得到 $\mathbf{a}(t)$ 和 $\mathbf{b}(t)$。但是，对于一些复杂的曲线计算二阶导数的工作可能比较困难。在二维空间中，可以获得垂直向量，它是与已知向量垂直的唯一向量（参见第 4 章）。但是，在三维空间中这种方法却不适用，那该怎样办？注意到，在这里并不需要一个独一无二的向量，只要是与 $\mathbf{T}(t)$ 垂直的某一个向量就可以。能否通过简便易行的方法得到这样的向量？

假设 $\mathbf{T}(t)$ 的分量为 a、b、c（它们当然都是 t 的函数）。通过观察，向量

$$\mathbf{n}=(-b,a,0) \tag{6.12}$$

与 $\mathbf{T}(t)$ 垂直（请验证）。把 $\mathbf{n}$ 和 $\mathbf{T}(t)$ 进行归一化，然后对它们进行叉积得到第三个向量 $\mathbf{a}$，这样就得到了所期望的坐标系：三个相互垂直的单位向量。

$$\mathbf{a}=n\times\mathbf{T} \tag{6.13}$$

一些读者也许会对把 $\mathbf{n}$ 的第三个分量强制设为 0 持有异议。对那些不喜欢此种方式的人也可以放松这个限制，把第三个分量设为一个不同的值 d，然后再调整第二个分量(a)以使得 $\mathbf{n}$ 和 $\mathbf{T}$ 的点积仍然为 0（如果读者对此感兴趣，可以自行探索其中的细节）。对于螺旋线这个例子，向量可以如下给出

$$\mathbf{n}(t) = \frac{1}{\sqrt{1+b^2}}(b\sin(t), -b\cos(t), 1) \tag{6.14}$$

图 6.43(b)是螺旋线上取不同 t 值得到的 Frenet 标架。

附：用数值方法计算 Frenet 标架

当 $\mathbf{C}(t)$的等式很复杂时，得到它连续阶导数的封闭形式可能比较麻烦，这样就需要把 $\mathbf{T}(t)$、$\mathbf{B}(t)$和 $\mathbf{N}(t)$的等式硬编码进程序中去。另一种解决方案是用数值方法近似计算那些导数，等式如下

$$\dot{\mathbf{C}}(t) = \frac{C(t+\varepsilon) - C(t-\varepsilon)}{2\varepsilon}$$

$$\ddot{\mathbf{C}}(t) = \frac{C(t-\varepsilon) - 2C(t) + C(t+\varepsilon)}{\varepsilon^2} \tag{6.15}$$

使用者心里应该清楚求数值微分本身就是一个不稳定的过程【Burden85】，但是以这种计算方式得到的方向 $\mathbf{T}(t)$、$\mathbf{B}(t)$和 $\mathbf{N}(t)$ 通常还是令人满意的。

图 6.44 包裹螺旋线而成的管状体

图 6.44 是以这种方式得到的用十边形包裹螺旋线的结果。在螺旋线上采样了 30 个点，在每个点上构造出 Frenet 标架，然后再把十边形放入新标架中。试着想象一下沿着脊线前进时局部坐标系的样子，并且留意一下它们是如何“漂移”和围绕脊线旋转的。

图 6.45 展示了其他一些有意思的例子，它们是基于圆环螺线的(在第 3 章的 3.8 节首次见过它们)【Gray93, p. 212】。其中，各个面的边都被绘制出来用以揭示此管状物是如何扭转前进的。绘制网格的边的问题我们会在案例 6.7 中考虑。圆环螺线是把一个圆环包裹在一条螺旋线上形成的(试着想象一下在这里看不到的那个圆环)，可以用如下等式给出

$$C(t) = ((a + b\cos(qt))\cos(pt), (a + b\cos(qt))\sin(pt), c\sin(qt)) \tag{6.16}$$

a、b、p 和 q 是某些指定的常数。图 6.45(a)中的参数 p 和 q 分别取 2 和 5，图 6.45(b)中它们取 1 和 7。

图 6.46 展示的是一个海螺形状，它是在螺旋线外包裹一个半径逐渐变大的圆环而得到的。为了达到这个目的，等式(6.10)中的矩阵应该再被乘以一个放缩矩阵，这里的放缩比例

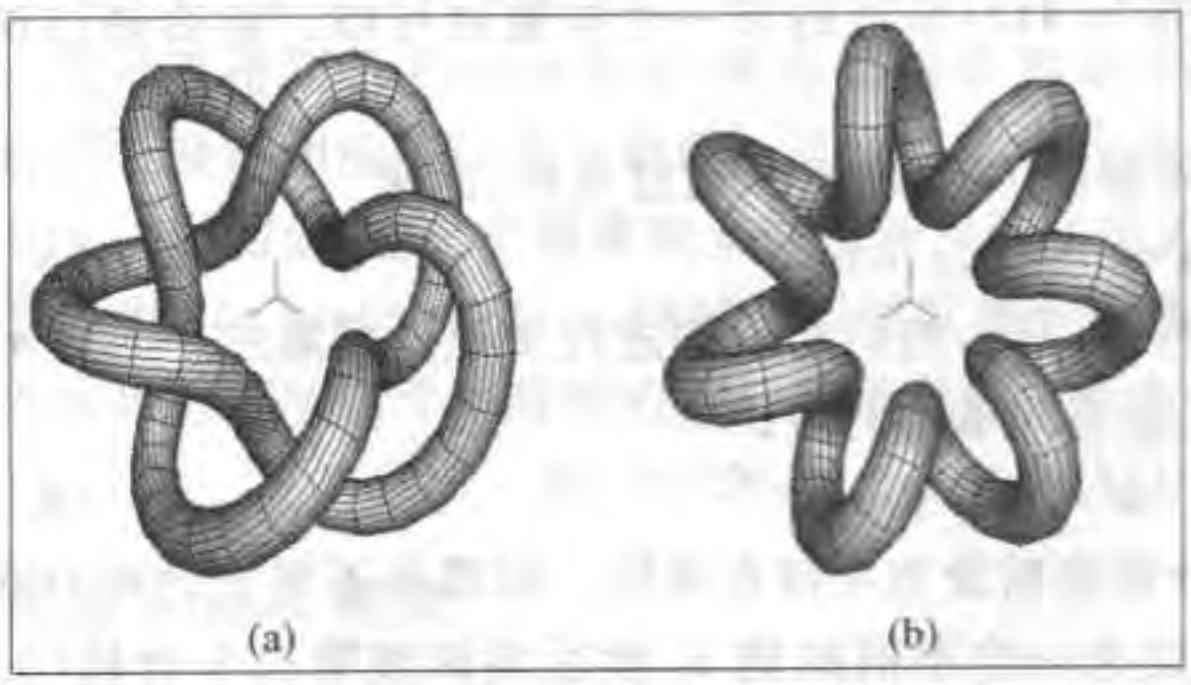


图 6.45 基于圆螺旋脊线的管状体

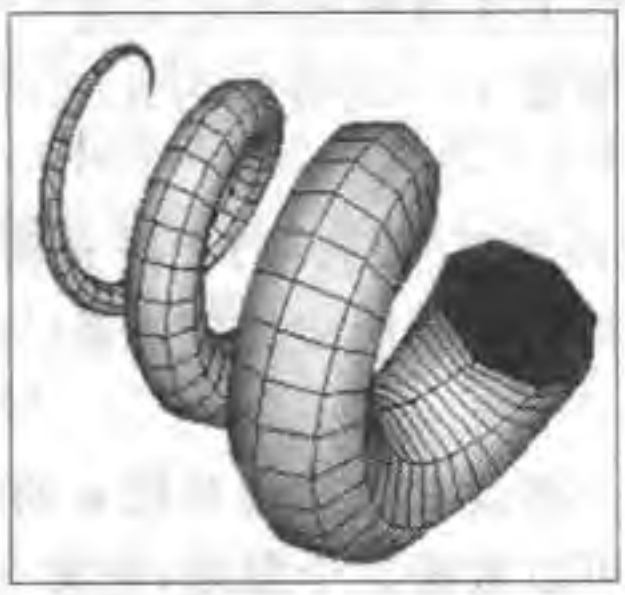


图 6.46 海螺形状

与 t 有关。

$$M' = M\begin{pmatrix} g(t) & 0 & 0 & 0 \\ 0 & g(t) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

此处的 $g(t)=t$。还可以再把旋转运动添加进矩阵中，这样的话沿着脊线观看时管子扭曲的就更加剧烈。

使用 Frenet 标架扫掠曲线引出的一个问题是局部标架的扭转有时会在生成的曲面中引进并不希望看到的扭曲。最近的研究工作，如【Wang97】，找到了 Frenet 标架的一种替代方法可以产生较小的扭曲进而得到更优美的曲面。

Frenet 标架的另一种有趣的应用是对过山车的运动进行分析。图 6.47 是一个过山车的模型，可以看到车辆的运动路线是十分复杂的。如果我们假设车辆内的发动机可以在任何情况下控制它的速度，那么如果知道了车辆的运行路线就可以确定出 $\boldsymbol{C}(t)$。现在若得到了 $\boldsymbol{C}(t)$适当的导数，那么就可以获取车辆运动轨迹的法向量以及副法向量，并且每个 t 值时的 Frenet 标架也能构造出来。这样就可以推导出等式用来计算车轮以及车辆中乘客的受力情况。

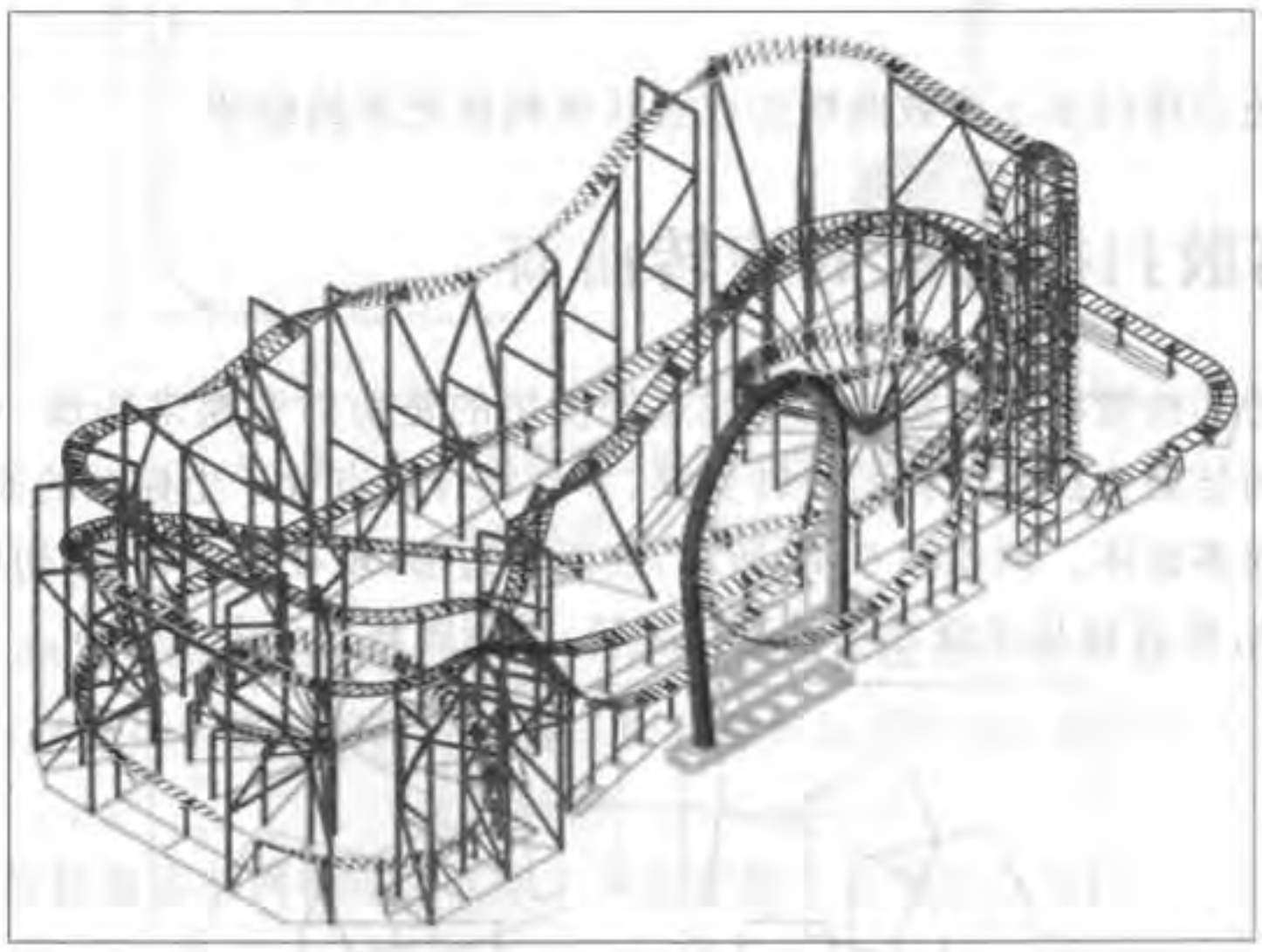


图 6.47　过山车模型

练习

练习 6.4.6　什么是 $\mathbf{N}(t)$?

请证明 $\mathbf{N}(t)$平行于 $\ddot{\mathbf{C}}(t)-(\dot{\mathbf{C}}(t)\cdot\ddot{\mathbf{C}}(t))\dot{\mathbf{C}}(t)/|\dot{\mathbf{C}}(t)|^2$，因此当 t 时刻的速度与加速度垂直时 $\mathbf{N}(t)$指向加速度的方向。

练习 6.4.7　螺旋线的标架

请考虑先前那个圆形螺旋线的例子。验证前面那些给出的单位切向量、副法线向量以

及法线向量的等式都是正确的。同时验证它们都是单位长度且相互垂直的。描述一下当你在曲线上运动的时候局部坐标系的朝向如何变化。

图 6.48 是附加的一些例子。图 6.48(a)是一个六边形裹在一条椭圆脊线外扫掠形成的某种椭圆管状体,图 6.48(b)是几段形体按照 Lissajous 图形的样子拼接成的一个扭曲,Lissajous 图形由下列等式给出

$$C(t) = (r\cos(Mt + \phi), 0, r\sin(Nt)) \tag{6.17}$$

这里 $M=2, N=3, \phi=0$。

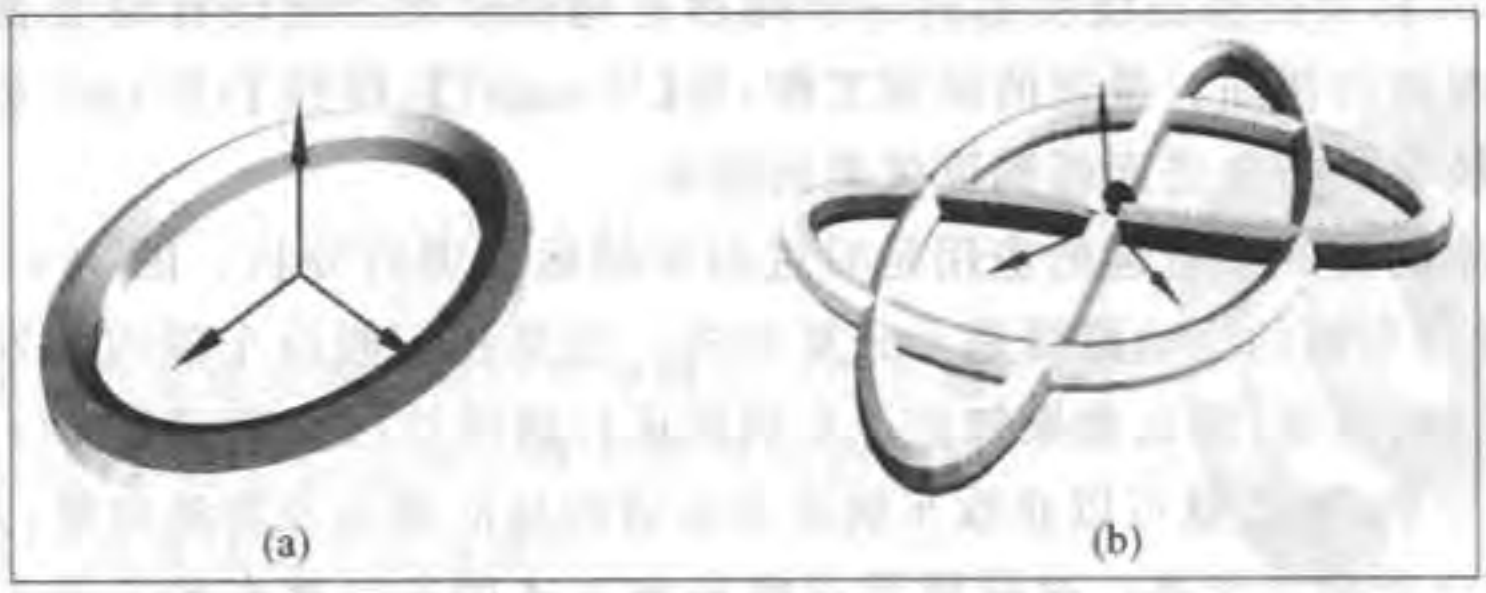


图 6.48

(a) 用六边形包裹而成的椭圆管状体;(b) 用七边形包裹 Lissajous 图形

案例 6.5 还会探讨基于参数曲线生成管状体网格更多的细节。

6.4.3 离散扫掠而成的旋转曲面

前面介绍的那些管状体在每一个脊线点处使用的是仿射变换来构造一个新的坐标系。如果我们使用的仿射变换是纯粹的旋转变换,并且把脊线点置于坐标系的原点,那么就可以得到一类丰富的多面体。例如图 6.49 所示,底面多边形(现在称为轮廓)初始位于 x 轴正向 3 个单位长度处,然后分几步绕着 y 轴进行旋转,得到圆环的一个近似体。

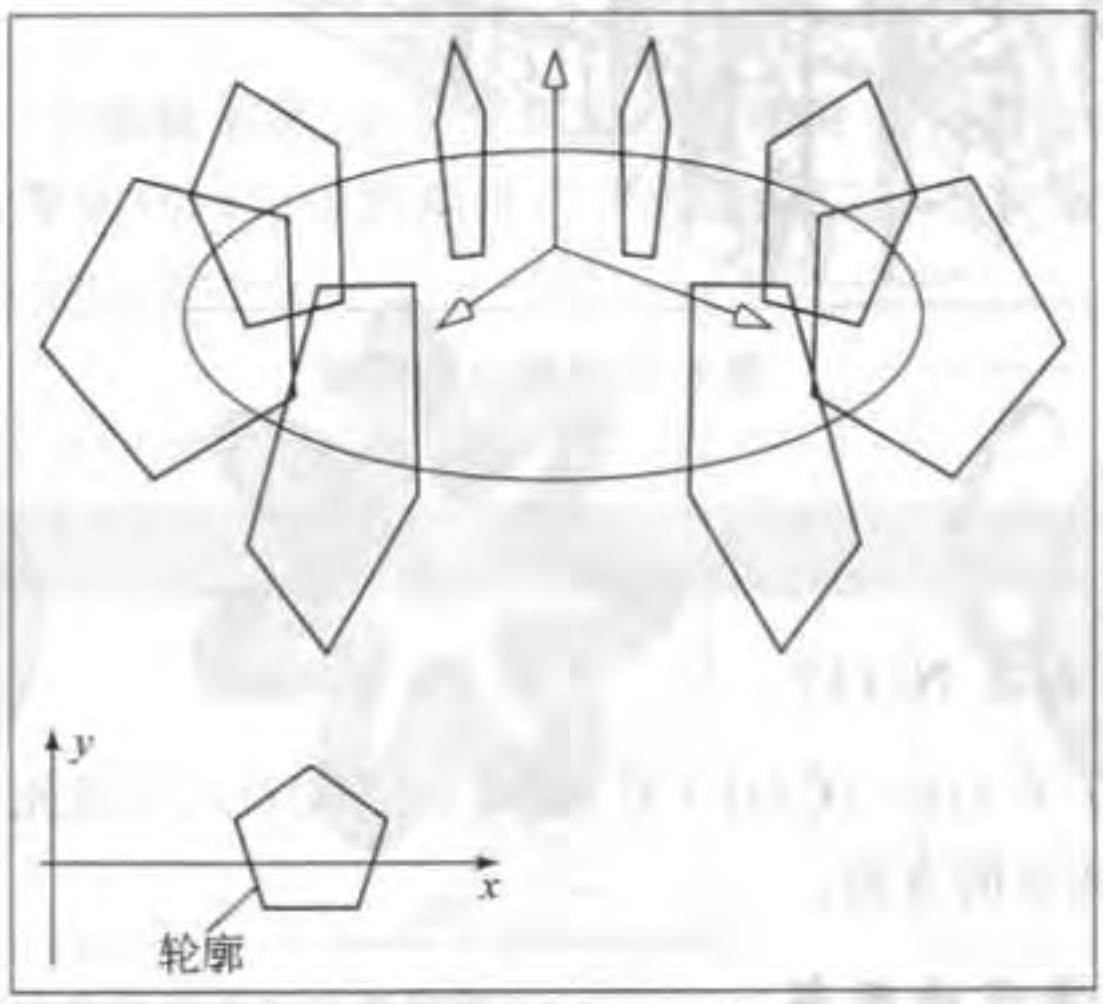


图 6.49 分步骤的旋转扫掠

这等价于把一个图形绕着坐标轴进行圆状扫掠，所得的形体通常称为旋转曲面。我们在 6.5 节会研究真正的旋转曲面；这里生成的仅仅是它们的离散逼近，因为我们是在离散步骤里完成扫掠工作的。

图 6.50 是一个生成马蒂尼酒杯形状的例子。这里的轮廓不再是一个封闭的多边形，而是基于一些点 $P_j=(x_j, y_j, 0)$ 得到的简单折线。如果要把折线围绕 y 轴放在 K 个等间隔角度处，那么变换矩阵应该为

$$\widetilde{M}_i = \begin{bmatrix} \cos(\theta_i) & 0 & \sin(\theta_i) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_i) & 0 & \cos(\theta_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

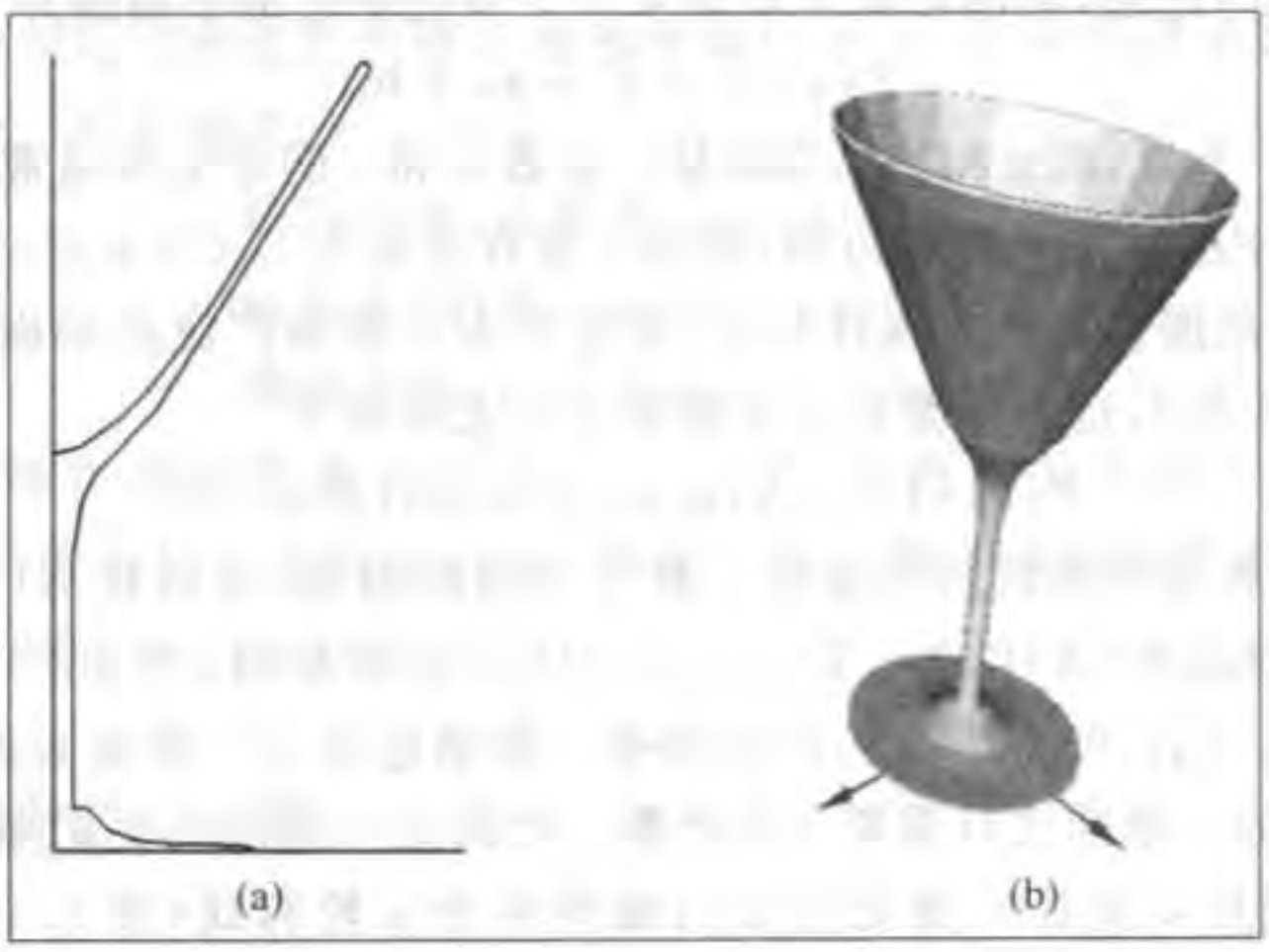


图 6.50　离散扫掠近似生成马蒂尼酒杯

(a) 轮廓；(b) 扫掠曲面

这里，$\theta_i=2\pi i/K$，$i=0, 1, \cdots, K-1$。注意，这里并没有牵扯到旋转操作。由于这个变换很简单，可以直接写出顶点的位置坐标。旋转操作生成的第 i 条折线为

$$(x_j\cos(\theta_i), y_j, x_j\sin(\theta_i)) \tag{6.18}$$

如何构造旋转曲面的网格将在 6.5.7 节和案例 6.6 中进一步讨论。

6.5　用网格逼近光滑物体

截止到现在，已经可以构造网格来表示多面体，这是一种由一堆平面多边形组成的形体。它们趋向于数据密集型，需要对每个面单独指定顶点。现在尝试用网格去逼近天生就光滑的形体，例如球面或环面。通常这些形体是由等式而不是数据定义的。我们还希望这些网格可以被光滑地绘制出来：即使它们也是由一些平面的集合所表示。适当的算法(Gouraud 渲染算法)可以用光滑渐变的方式绘制它们，并不能明显看出面片与面片之间的界限(如图 6.1所示)。这就要求在所有面片的每个顶点找到恰当的法向量。特别地，我们

可以计算原有的那张光滑曲面的法向量。我们会在第 8 章讨论 Gouraud 渲染算法。

处理任意类型曲面的一个基本方法是把它多边形化(或离散化),即表示为一堆平面的集合。如果各个平面足够小并且从一个平面到另一个平面它们的方向也变化得比较平缓,那么得到的网格就可以很好地逼近原始曲面。面片的顶点可以通过对曲面参数表达式进行离散采样得到。网格的顶点列表与面片列表是采用通常的方法构造出来的,只是这里的顶点是由等式计算得到。顶点的法向量也同样如此:它们是通过计算原始曲面法线的等式在各离散点处的取值得到的。

6.5.1 曲面的表示

为了展开下面的论述,请回忆第 4 章的 4.5.5 节讨论过的平面面片,用参数法表示如下

$$P(u,v) = C + \mathbf{a}u + \mathbf{b}v \tag{6.19}$$

这里,C 代表一个点,而 **a** 和 **b** 都是向量。参数 u 和 v 的定义域通常限制为[0, 1],这样面片就是 3D 空间中的一个平行四边形,其四个角点分别为 C、$C+\mathbf{a}$、$C+\mathbf{b}$ 以及 $C+\mathbf{a}+\mathbf{b}$。

这里把研究的范围扩大到非线性形式,用它来表示更加广泛的曲面形状。我们引进三个函数 $X()$、$Y()$和 $Z()$,这样曲面可以用参数形式表示如下

$$P(u,v) = (X(u,v), Y(u,v), Z(u,v)) \tag{6.20}$$

这里的 u 和 v 在适当的区间内取值。对于不同的曲面,其函数 X、Y、Z 也不相同。当 u 和 v都等于 0 时曲面在$(X(0,0), Y(0,0), Z(0,0))$处取值,当 $u=1$ 而 $v=0$ 时曲面在$(X(1,0), Y(1,0), Z(1,0))$处取值,以此类推。要铭记表示一张曲面需要使用两个参数,而表示 3D 空间中的一条曲线只需要一个参数。当固定 v 而让 u 变化得到的曲线我们称为 v 轮廓线,同样的固定 u 而让 v 变化生成的曲线称为 u 轮廓线(图 6.51 中有 u 轮廓线以及 v 轮廓线的例子)。

曲面的隐函数形式

尽管我们主要关心不同曲面的参数表达式,但是了解描述曲面的另一种方法即隐函数形式也是很有帮助的。请回忆一下第 3 章的 3.8 节所讲的内容,2D 平面中的曲线具有隐函数形式 $F(x,y)$,对于且仅对于曲线上的点,此函数取值为 0。3D 空间中的曲面也存在一个相似的函数 $F(x,y,z)$,它只在曲面上的点处取值为 0。因此曲面可以由如下的隐函数等式表示

$$F(x,y,z) = 0 \tag{6.21}$$

曲面上所有的点有且仅有这些点满足此等式(请注意"隐函数等式"与"隐函数形式"的区别)。此等式限制了 x、y 和 z 的值必须是所考察曲面上某一点的坐标分量。例如,回忆(见第 4 章)通过一点 B 且法向量为 **n** 的平面可以用等式表示为 $n_x x + n_y y + n_z z = D$(这里的 $D = \mathbf{n} \cdot B$),所以,此平面的隐函数形式为 $F(x,y,z) = n_x x + n_y y + n_z z - D$。有时把 F 看成是点 P 而不是 x、y、z 三个变量的函数更加方便一些,由此曲面上所有的点可以用 $F(P)=0$ 简洁地表示出来。例如平面的这个例子,可以定义 $F(P) = \mathbf{n} \cdot (P - B)$并且当且仅当 $F(P) = \mathbf{n} \cdot (P - B)$等于 0 时点 P 在此平面上。如果希望使用齐次坐标系(回忆第 4 章 4.5 节的内容),那么 $\tilde{P}$ 就可以表示为一个四元组 $\tilde{P} = (x,y,z,1)^T$,这样平面的隐函数形式就可

以变得更加简单：$F(\widetilde{P})=\widetilde{n}\cdot\widetilde{P}$，这里的 $\widetilde{n}=(n_x,n_y,n_z,-D)$ 同时含有法线向量以及 $-D$ 的信息。

已知参数形式，要推导出 $F(x,y,z)$ 或 $F(P)$ 通常并不容易（给出 $F(x,y,z)$ 要找到其参数形式同样也不容易）。但是，如果能同时获取参数形式和隐函数形式，就可以很容易地判别它们是否描述的是同一张曲面。只需要把 $F(x,y,z)$ 中的 x、y 和 z 替换为 $X(u,v)$、$Y(u,v)$、$Z(u,v)$，然后验证 F 是否在 u、v 相应的定义域内恒等于 0。

对于球面这种封闭曲面，可以很容易地定义出内部区域与外部区域。其他一些类似于平面的曲面很明显地把 $3D$ 空间划分为两个部分，但是还要根据具体的应用环境确定哪一个半空间是内部哪一个是外部。而对于诸如糖果盒上的彩带这样的曲面，定义内部区域和外部区域就没有什么意义了。

当能够有意义地指定出曲面的内部区域和外部区域时，曲面的隐函数形式 $F(x,y,z)$ 也被称为内外部函数。我们说点 (x,y,z) 在

$$\begin{aligned}&\text{曲面内部，如果 } F(x,y,z)<0\\&\text{曲面上，如果 } F(x,y,z)=0\\&\text{曲面外部，如果 } F(x,y,z)>0\end{aligned}\tag{6.22}$$

这就提供了一种快速简便地判别点 (x',y',z') 与曲面关系的方法：只需要求得 $F(x',y',z')$ 的值然后看它是正是负还是 0。最近又有人做了许多出色的工作使用隐函数形式来渲染曲面，参见【Bloomenthal97】。

6.5.2　曲面的法向量

前面已经讲过，我们需要计算出的曲面任意点处法向量。下面会给出一种基于曲面参数表达式的方法，还会给出一种基于隐函数形式的方法。然后对每一种类型的曲面进行探讨，找出计算其法向量的合适方法。

通过考虑曲面在点 $P(u_0,v_0)$ 周围的一小块邻域，可以定义出点 $P(u_0,v_0)$ 的法向量方向。如果这个邻域足够小并且曲面在附近变化得也很平稳，那么这个邻域实质上可以看作是平坦的。因此，在局部范围内它和一小块平面很相像，并且有一个明确定义的法线方向。图 6.51 在一张曲面面片上画出了不同点处的法向量。看上去这张曲面上不同的点其法向量也各不相同。

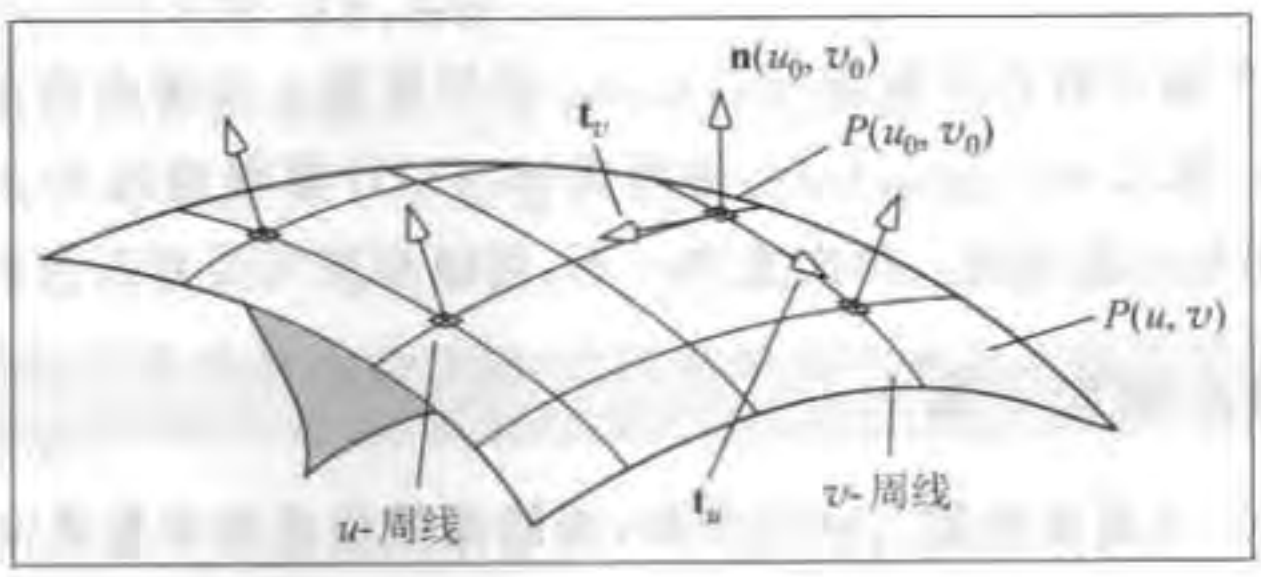


图 6.51　曲面的法向量

用 $\mathbf{n}(u,v)$ 表示(u,v)处的法向量。现在开始讨论如何对它进行计算。

参数曲面的法向量

不难知道,$\mathbf{n}(u,v)$可以用(u_0,v_0)处平面邻域上两个向量的叉积得到。作为两个向量叉积得到的结果,它可以保证与那两个向量都垂直。平面中的那两个向量(在图 6.51 中用 $\mathbf{t}_u$ 和 $\mathbf{t}_v$ 表示)都是某种切向量。从微积分教科书中可以看到它们仅仅是 $\mathbf{p}(u,v)$(由原点到曲面点 $P(u,v)$的向量[①])的偏导数在目标点处的取值【Thomas53】。因此法向量的表达式可以写为

$$\mathbf{n}(u_0,v_0)=\left(\frac{\partial\mathbf{p}}{\partial\mathbf{u}}\times\frac{\partial\mathbf{p}}{\partial\mathbf{v}}\right)\bigg|_{u=u_0,v=v_0} \tag{6.23}$$

这里的竖线|表示偏导数在 $u=u_0,v=v_0$ 处取值。这种方式得到的 $\mathbf{n}(u,v)$并不是单位长度的向量,但是可以根据需要对它单位化。

例 6.5.1 上述方法适用于平面吗?

考虑用参数形式给出的平面 $P(u,v)=C+\mathbf{a}u+\mathbf{b}v$。对 u 求偏导数得到 $\mathbf{a}$,对 v 求偏导数得到 $\mathbf{b}$。根据等式(6.23),$\mathbf{n}(u,v)=\mathbf{a}\times\mathbf{b}$,可以看出这个结果是正确的。

一般来说,如果曲面足够光滑的话 $\mathbf{p}(u,v)$的偏导数就存在。在场景建模中我们遇到的大部分曲面都具有必要的光滑性和足够简单的数学表达式,所以偏导数并不难获得。因为 $\mathbf{p}(u,v)=X(u,v)\mathbf{i}+Y(u,v)\mathbf{j}+Z(u,v)\mathbf{k}$,而向量的偏导数是其各个分量的偏导数组成的向量,所以有

$$\frac{\partial\mathbf{p}(u,v)}{\partial u}=\left(\frac{\partial X(u,v)}{\partial u},\frac{\partial Y(u,v)}{\partial u},\frac{\partial Z(u,v)}{\partial u}\right) \tag{6.24}$$

后面会直接对每一种曲面使用这个等式。

隐式曲面的法向量

如果曲面是以隐函数的形式 $F(x,y,z)=0$ 给出的话,法向量可以有另外一种表达方式。曲面点(x,y,z)上的法向量可以用三个偏导数生成——通常叫作 F 的梯度,用∇F 表示,等式如下【Thomas53】

$$\mathbf{n}(x_0,y_0,z_0)=\nabla F\,|_{x=x_0,y=y_0,z=z_0}=\left(\frac{\partial F}{\partial x},\frac{\partial F}{\partial y},\frac{\partial F}{\partial z}\right)\bigg|_{x=x_0,y=y_0,z=z_0} \tag{6.25}$$

这里要计算每个偏导数在目标点(x_0,y_0,z_0)处的取值。如果曲面上点(x_0,y_0,z_0)的参数形式为 $P(u_0,v_0)$,那么 $\mathbf{n}(x_0,y_0,z_0)$与用等式(6.17)计算得到的 $\mathbf{n}(u_0,v_0)$它们的方向都是相同的,只是长度有可能不同。再次重申一下,可以根据需要对其进行单位化。

例 6.5.2 再次研究平面

再次考虑通过点 A 且法向量为 $\mathbf{n}$ 的平面,用隐函数形式表示为 $F(x,y,z)=n\cdot((x,y,$

① 由于 $\mathbf{p}(u,v)$就是 $P(u,v)-(0,0,0)$,所以 $\mathbf{p}(u,v)$的导数等与 $P(u,v)$的导数。

$z)-A)=0$，或 $n_x x+n_y y+n_z z-\mathbf{n}\cdot A=0$。它的梯度 $\nabla F=\mathbf{n}$，这与我们期望的一致。

注意到用梯度形式给出的法向量是关于 x、y 和 z 的函数，并不是用 u 和 v 表出的。有时候对于同一张曲面，我们同时知道其内外部函数 $F(x,y,z)$ 以及参数形式 $\mathbf{p}(u,v)=X(u,v)\mathbf{i}+Y(u,v)\mathbf{j}+Z(u,v)\mathbf{k}$。对于这种情形，可以很简单地找出法线的参数形式 $\mathbf{n}(u,v)$，方法分为两步：(1)使用等式(6.20)得到用 x、y 和 z 表示的法向量，然后(2)用 $X(u,v)$ 替换 x，用 $Y(u,v)$ 替换 y，用 $Z(u,v)$ 替换 z。稍后会在一些例子中用到这个方法。

6.5.3 仿射变换的影响

有时候我们需要处理已经进行过仿射变换的隐函数曲面或参数曲面。我们想要了解仿射变换如何影响曲面的法向量。

假设变换是由一个 4×4 的矩阵 M 表示的，原始曲面具有隐函数形式(点都用齐次坐标表示)$F(\widetilde{P})$ 并且参数形式 $\widetilde{P}(u,v)=(X(u,v),Y(u,v),Z^{*}(u,v),1)^{T}$。那么很明显变换后曲面的参数形式为 $M\widetilde{P}(u,v)$(为什么?)。同样，很容易验证(参见稍后的练习)变换后的曲面其隐函数形式为 $F'(\widetilde{P})=F(M^{-1}\widetilde{P})$。

进一步，如果原始曲面的法向量为 $\mathbf{n}(u,v)$，那么变换后的曲面具有：法向量：$M^{-T}\mathbf{n}(u,v)$。

例如，先前讨论的那个具有法向量 $\bar{n}=(n_x,n_y,n_z,-D)$ 的平面 $F(\widetilde{P})=\bar{n}\cdot\widetilde{P}$，假设要对它实施变换。变换后平面的隐函数形式 $F'(\widetilde{P})=\bar{n}\cdot(M^{-1}\widetilde{P})$。这也可以写作 $(M^{-T}\bar{n})\cdot\widetilde{P}$(参见练习)，所以变换后的平面法向量包含转置矩阵的逆矩阵，这与一般的曲面其法向量受到的影响相同。

练习

练习 6.5.1 变换后曲面的隐函数形式

假设曲面上的所有点满足 $F(P)=0$，并且 M 把 $\widetilde{P}$ 变换到 $\widetilde{Q}$ 即 $\widetilde{Q}=M\widetilde{P}$。那么请说明变换后的曲面上任意一点 $\widetilde{Q}$ 都是通过 $M^{-1}\widetilde{Q}$ 变换而来的，并且它满足 $F(M^{-1}\widetilde{Q})=0$。这也就证明了变换后曲面的隐函数形式为 $F(\widetilde{Q})=F(M^{-1}\widetilde{Q})$。

练习 6.5.2 法向量怎样被影响?

设 $\mathbf{n}=(n_x,n_y,n_z,0)^{T}$ 为 P 点处的法向量并且设 v 为曲面 P 点处任意一条切向量。那么 $\mathbf{n}$ 必然与 $\mathbf{v}$ 垂直，可以写作 $\mathbf{n}\cdot\mathbf{v}=0$。

a. 验证点积可以用矩阵乘法表示为 $\mathbf{n}^{T}\mathbf{v}=0$(见附录 2)。

b. 验证当插入矩阵乘积 $M^{-1}M$ 后点积仍然为 0：$\mathbf{n}^{T}M^{-1}M\mathbf{v}=0$。

c. 验证上式可以写为 $(M^{-T}\mathbf{n})(M\mathbf{v})=0$，所以 $M^{-T}\mathbf{n}$ 垂直于 $(M\mathbf{v})$。由于切向量 $\mathbf{v}$ 被变换为 $M\mathbf{v}$，而 $M\mathbf{v}$ 又与变换后的曲面相切，这就是说 $M^{-T}\mathbf{n}$ 必须要正交于变换后的曲面，也就是我们所希望验证的。

d. 曲面的法向量还可以由隐函数形式的梯度表出，所以变换后的曲面在 P 点处的法线一定是 $F(M^{-1}P)$ 的梯度。请验证(使用微积分中的链式法则)此函数的梯度等于 M^{-T} 乘以 $F()$ 的梯度。

练习 6.5.3 变换后曲面的切平面

为了了解切向量如何变换,可以先看一看切平面在曲面变换前后的对应关系。假设原始曲面在 P 点处的切平面有参数形式 $P+\mathbf{a}u+\mathbf{b}v$,这里的 $\mathbf{a}$ 和 $\mathbf{b}$ 是平面中的两个向量。因此曲面的法线 $\mathbf{n}=\mathbf{a}\times\mathbf{b}$

a. 验证变换后曲面的参数形式是 $MP+M\mathbf{a}u+M\mathbf{b}v$,并且此平面的法向量 $\mathbf{n}'=(M\mathbf{a})\times(M\mathbf{b})$。

b. 参照附录 2,验证下列等式:

$$(M\mathbf{a})\times(M\mathbf{b})=(\det M)M^{-T}(\mathbf{a}\times\mathbf{b})$$

上式把变换后向量的叉积与变换前向量的叉积联系了起来。

c. 由此验证 $\mathbf{n}'$ 与 $M^{-T}\mathbf{n}$ 平行。

6.5.4 三个"通用"形体:球面、柱面和锥面

我们开始讨论三个经典形体,球面、柱面和锥面的"通用"版本。我们会推导出它们的隐函数以及参数形式。通用的意思是说形体的大小、朝向、位置都是基本的或基础的。换成一种粗略的说法是对它们进行描述只需要用到最少的参数:绝大多数的参数都等于 0。例如,通用球面的圆心在原点并且半径为单位长度;通用锥面的底面是 xy 坐标平面上的单位圆,并且它的轴与 z 坐标轴重合也是单位长度。稍后还会讨论另两大类的通用形体,通用柱面和一端逐渐变小的柱面。我们还会推导出这些物体每一点处的法线方向。请留意我们在第 5 章已经使用过 OpenGL 函数来绘制这些形体。然而我们自己开发的工具有如下优势:(a)对于生成的形体,可以对其细节部分进行更多的控制;(b)可以得到物体真正的网格表示,这样就可以用 Mesh 类中的方法对它进行操作。

通用球面

我们把球心在原点半径为 1 的球面称为通用球面(如图 6.52(a)所示)。它构成了我们使用的所有球状形体的基础。它的隐函数形式我们并不陌生:

$$F(x,y,z)=x^2+y^2+z^2-1 \tag{6.26}$$

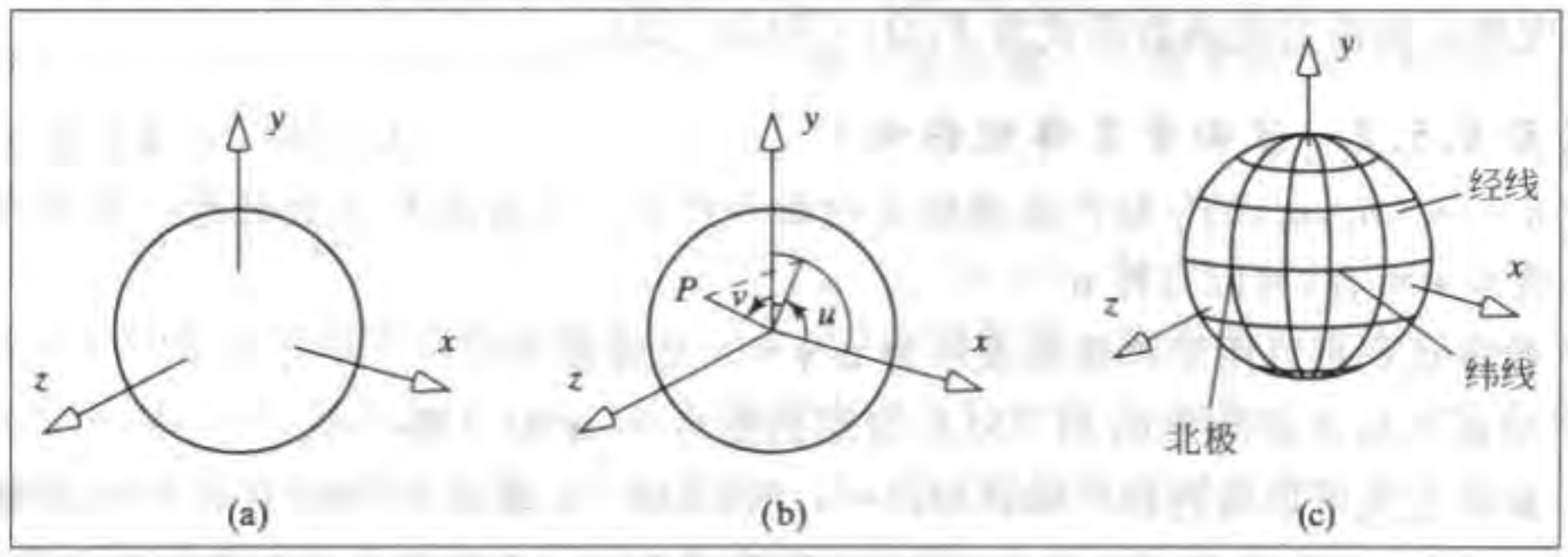


图 6.52

(a) 通用球面;(b) 参数形式;(c) 纬线和经线

它还有一种更加优美的表达方式 $F(P)=|P|^2-1$(如果球面半径是 R,表达形式又如何?)。

通过把点用球坐标(见附录 2)表示可以立即获得此球面的参数表达式。我们用 u 表示经度,用 v 表示纬度。那么球面上任意一点 $P=(x,y,z)$ 在球坐标系下的参数形式为 $(\cos(v)\cos(u),\cos(v)\sin(u),\sin(v))$(如图 6.52(a)所示)。

$$P(u,v)=(\cos(v)\cos(u),\cos(v)\sin(u),\sin(v)) \tag{6.27}$$

为了覆盖所有的点,令 u 的定义域为 $(0,2\pi)$,v 的定义域为 $(-\pi/2,\pi/2)$。

可以很容易地验证上述参数形式与隐函数形式是一致相容的:把等式(6.27)的项带入到等式(6.26)中去,可以看到对任意的 u 和 v 结果都为 0。

(**问题**:如果球面的球心在 (a,b,c) 半径为 R,那么参数形式如何?)

可以采用地理学的名词对球面上的某些轮廓线命名。参数 u 轮廓线称为经线,参数 v 轮廓线称为**纬线**,如图 6.52(c)所示。

问题:u 和 v 取何值时点 $P(u,v)$ 在赤道上?南极点处的 u 和 v 取何值?北极呢?

一个给定的形状可能有不同的参数形式。球面的另一种参数形式将在练习中讨论。

曲面的法向量

对于球面上任意一点,如何用参数 (u,v) 表示此处的法向量 $\mathbf{n}(u,v)$?凭直觉法向量都是沿着半径指向外侧的,所以法向量必须平行于从原点到球面点所构成的向量。等式(6.19)肯定了上述说法:梯度可以简单地表示为 $2(x,y,z)$,它是与 P 点坐标成比例的。对于参数形式,通过等式(6.27)可以得到 $\mathbf{n}(u,v)=-\cos(v)\ p(u,v)$,所以正如期望的那样 $\mathbf{n}(u,v)$ 平行于 $\mathbf{p}(u,v)$。当对 $\mathbf{n}$ 进行归一化后可以消去我们不感兴趣的尺度因子 $-\cos(v)$。我们必须要用 $\mathbf{p}(u,v)$ 而不是 $-\mathbf{p}(u,v)$ 来表示法线,这样就确保它是沿着半径指向外侧而不是内侧。当你的船停留在北纬 50 度 25 分、西经 4 度 26 分时,船的法向量的经度和纬度是什么(甲板上桅杆顶端所指方向)?首先这个问题是否是定义良好的,其次如果是的话答案是什么?

通用柱面

我们所说的通用柱面,它的中心轴与 z 坐标轴重合,横截面是半径为 1 的圆形,并且沿着 z 轴从 0 拉伸到 1,图 6.53(a)是它的图形。

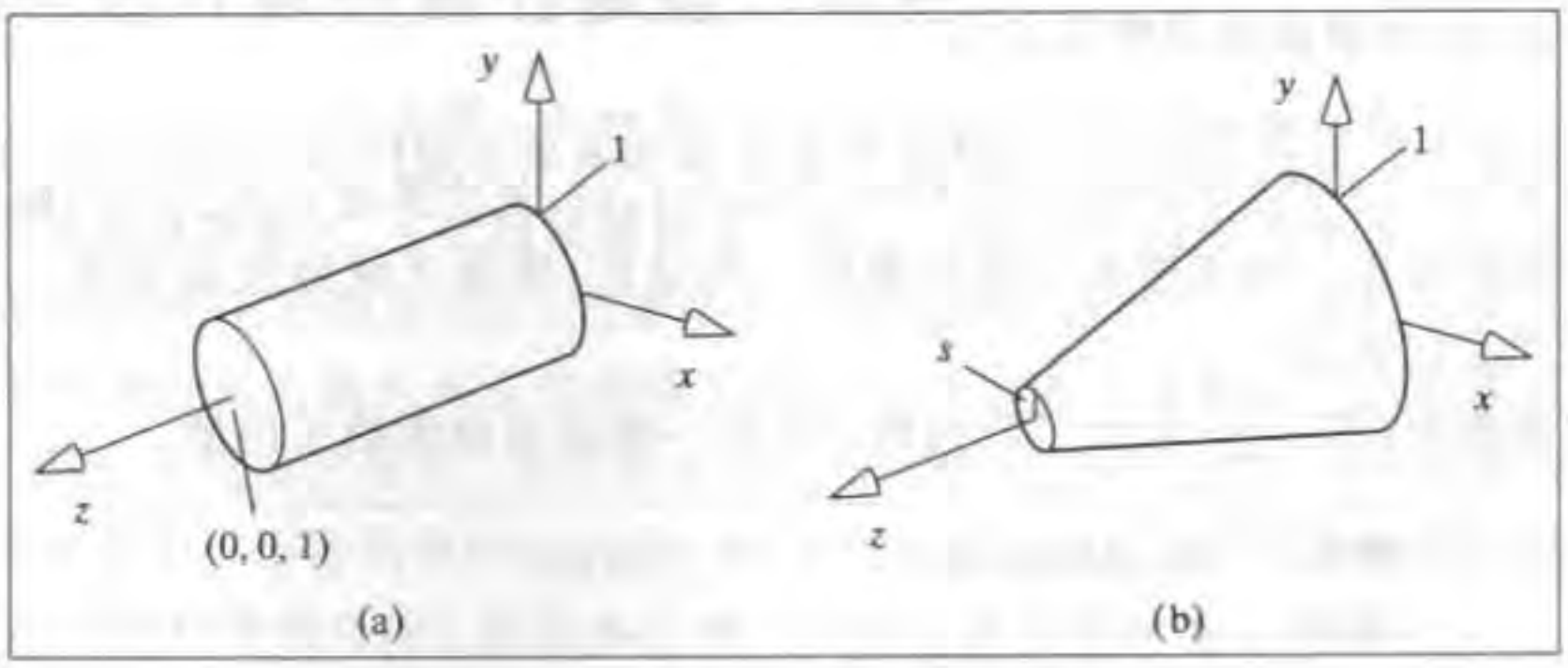


图 6.53　通用柱面和一端逐渐变小的柱面

像我们在第 5 章中所做的那样,把上述柱面看成是一端逐渐变小的柱面的特例是十分方便的。这样只需引进一个参数就可以表示无数种形状。图 6.53(b)是一个通用的一端逐

渐变小的柱面,在 $z=1$ 时圆半径为 s。参数 s 通常小于 1,但是它也可能大于 1(想象这两种情况)。通用柱面是一端逐渐变小的柱面当 $s=1$ 时的特例。更近一步,下面要研究的通用锥面是一端逐渐变小的柱面当 $s=0$ 时的形状。我们会推导当 s 为任意值时一端逐渐变小的柱面的方程。这也就可以得到通用柱面和锥面的等式,只须相应地把 s 设为 1 或 0。

如果把一端逐渐变小的柱面想象为一个中空的薄壳,它的外壁(暂时不考虑柱面的顶面和底面)的隐函数形式为:

$$F(x,y,z) = x^2 + y^2 - (1+(s-1)z)^2 \quad 当\ 0 < z < 1 \tag{6.28}$$

参数形式为

$$P(u,v) = ((1+(s-1)v)\cos(u),(1+(s-1)v)\sin(u),v) \tag{6.29}$$

其中,u、v 在适当的范围内取值(是什么?)。当 $s=1$ 时的通用柱面的表达形式如何?

当要把一端逐渐变小的柱面表示为实体对象时,要在它的两端添加两个圆盘:一个底面和一个顶面。顶面是平面 $z=1$ 上的一块圆形部分,可以用不等式表示为 $x^2+y^2<s^2$,或用参数形式表示为 $P(u,v)=(v\cos(u),v\sin(u),1)$,其中 v 在$[0,s]$中(底面的参数形式是什么?)。

一端逐渐变小的柱面其外壁的法向量可以用等式(6.25)推导出来(请验证)。结果为:

$$\mathbf{n}(x,y,z) = (x,y,-(s-1)(1+(s-1)z)) \tag{6.30}$$

或者用参数形式表为 $\mathbf{n}(u,v)=(\cos(u),\sin(u),1-s)$。通用柱面的法向量可以更简单地写为$(\cos(u),\sin(u),0)$。这与我们的直觉是一致的:法线是从柱面的中心轴沿着半径方向向外辐射。一端逐渐变小的柱面其外壁法线也是沿着半径方向的,但是它的 z 分量有一个常值偏移量(顶面和底面的法线是什么?)。

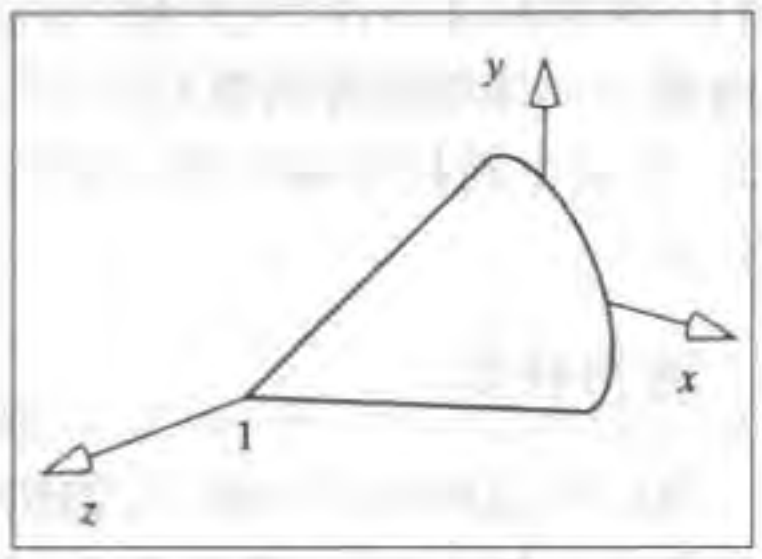


图 6.54 通用锥面

通用锥面

我们指的通用锥面是中心轴与 z 坐标轴重合,圆截面的最大半径为 1 的锥面并且沿着 z 方向从 0 拉伸到 1,如图 6.54 所示。它是一端逐渐变小的柱面当 $s=0$ 时的特例。因此,其外壁的隐函数形式为

$$F(x,y,z) = x^2 + y^2 - (1-z)^2 = 0 \quad 当\ 0 < z < 1 \tag{6.31}$$

参数形式为 $P(u,v)=((1-v)\cos(u),(1-v)\sin(u),v)$,其中弧度 u 在$[0,2\pi]$取值,v 在$[0,1]$中取值。再次使用一端逐渐变小的柱面的一些结论,锥面外壁的法向量是$(x,y,1-z)$。它的参数形式如何表达?

为了方便起见,图 6.55 给出了我们所讨论的一些通用曲面的法向量。

曲面	$\mathbf{n}(u,v)$ at $\mathbf{p}(u,v)$	$\nabla \mathbf{F}(x,yz)$
球形	$\mathbf{p}(u,v)$	(x,y,z)
锥形	$(\cos(u),\sin(u),1-s)$	$(x,y-(s-1)(1+(s-1)z))$
圆柱形	$(\cos(u),\sin(u),0)$	$(x,y,0)$
圆锥形	$(\cos(u),\sin(u),1)$	$(x,y,1-z)$

图 6.55 通用曲面的法向量

练习

练习 6.5.4 另一种表示通用球面的方法

可以通过把参数 u、v 表示为不同的几何量从而得到球面不同的参数形式。我们还是用 u 表示弧度，但是用 v 表示点相对于 xy 坐标平面的高度。所有高度为 v 的点落在半径为 $\sqrt{1-v^2}$ 的圆上，所以参数形式可以写为

$$P_2(u,v)=(\sqrt{1-v^2}\cos(u),\sqrt{1-v^2}\sin(u),v) \tag{6.32}$$

其中 u 的区间为 $[0,2\pi]$，v 的区间为 $[0,1]$。请验证对于所有的 u、v，P_2 离原点距离为 1。

练习 6.5.5 下列曲面是什么？

设 A 的位置是 $\mathbf{a}$，P 的位置是 $\mathbf{p}$。用语言描述并且大致绘出下列曲面：

(1) $\mathbf{p}\cdot\mathbf{a}=0$；

(2) $\mathbf{p}\cdot\mathbf{a}=|\mathbf{a}|$；

(3) $|\mathbf{p}\times\mathbf{a}|=|\mathbf{a}|$；

(4) $\mathbf{p}\cdot\mathbf{a}=\mathbf{p}\cdot\mathbf{p}$；

(5) $\mathbf{p}\cdot\mathbf{a}=|\mathbf{a}||\mathbf{p}|/2$。

练习 6.5.6 计算通用柱面和锥面的法向量

用两种方法计算一端逐渐变小的通用柱面和通用锥面在任一点处的法向量：

a. 使用参数形式。

b. 使用隐函数形式，然后把结果转化成参数形式。

练习 6.5.7 变换球面

把通用球面在 x 方向放大 2 倍，在 y 方向放大 3 倍，然后绕 z 轴旋转 30°，计算变换后图形的隐函数形式。

6.5.5 构造弯曲曲面的多边形网格

现在我们讨论如何构造网格对象来近似逼近诸如球面、柱面或锥面等光滑曲面。这个过程称为多边形化或离散化，它把曲面替换为一堆三角形和四边形的集合。这些多边形的顶点位于曲面自身，并且用线段相连（它通常并不完全落在曲面上）。我们选择好一系列的 u、v 值，然后利用曲面的参数形式在这些值上进行采样得到顶点集合。这些顶点被放进顶点列表之中。然后生成面片列表：每个面片包含三个或四个索引指向顶点列表中的相应顶点。给面片中每个顶点指定曲面的法向量。这个法向量是在此顶点处原始曲面的法线方向（请留意与表示多面体时的区别：那里是把面片的法线指定给面片中的顶点）。

图 6.56 展示了上述方法如何对通用球面进行处理，观察方向是迎着球的南极。我们沿着经度和纬度线对球面进行剖分。使用 OpenGL 中“条”和“层”的概念（见第 5 章的 5.6.3 节），我们把球面沿着赤道方向切成 nSlices 条，而从南极到北极方向分成 nStacks 层。图中是 12 条 8 层的例子。nSlices 和 nStacks 越多，网格逼近球面的效果越好。

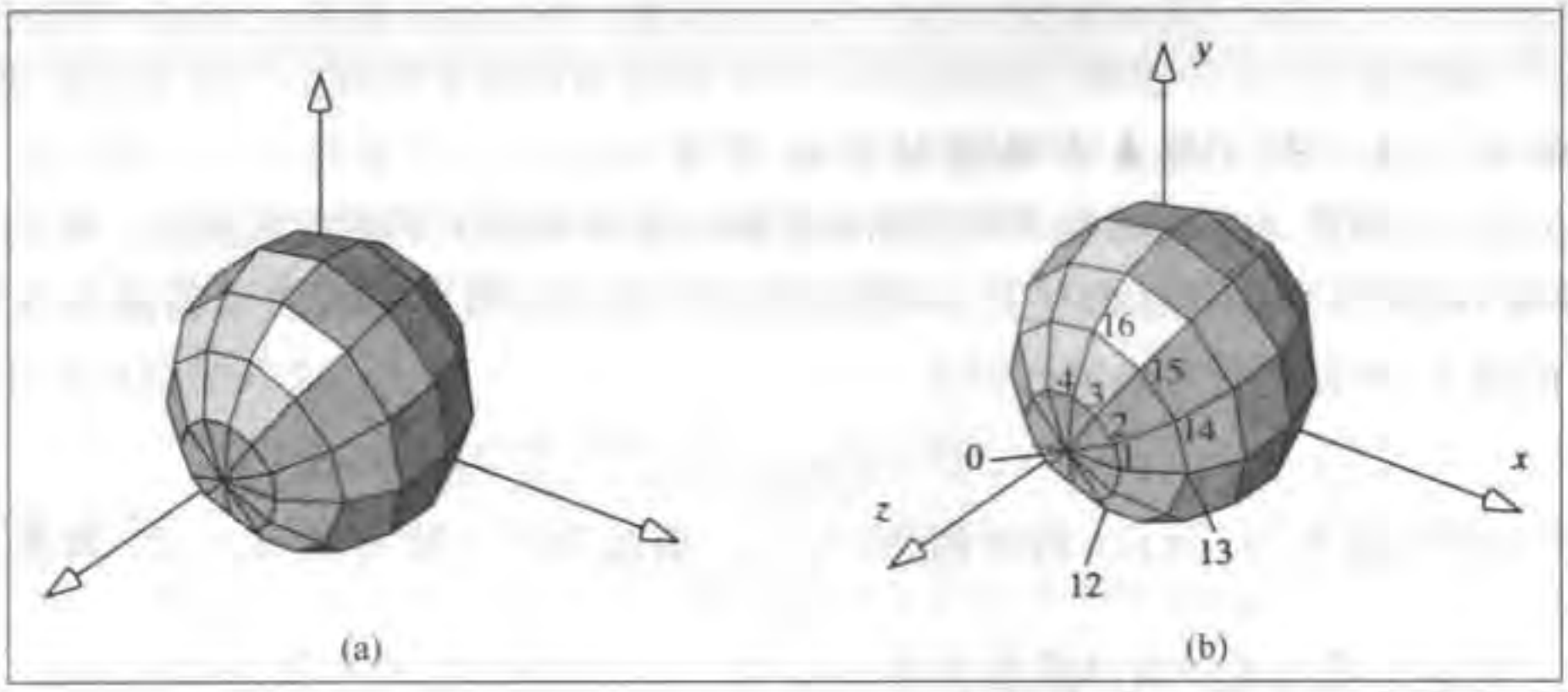


图 6.56

(a) 近似逼近通用球面的网格；(b) 给顶点编号

切条时，需要沿着赤道方向从 0 到 2π 选出 nSlices 个 u 值。通常这些值是等距分布的：$u_i = 2\pi i/\text{nSlices}, i = 0, 1, \cdots, \text{nSlices}-1$。分层时，通常把一半数量的层放在赤道之上，一半放在下面。最顶层和最底层是由三角形构成的；其他层中的面都是四边形。这样就需要在沿着纬度方向定义(nStacks+1)个值：它们都在 $-\pi/2 \sim \pi/2$ 间取值，$V_j = \pi/2 - \pi j/\text{nStacks}, j = 0, 1, \cdots, \text{nStacks}$。

现在就可以生成顶点列表了。图中展示了我们对顶点标号的方式(要选用方便的方式排序)。令南极点为 pt[0]，其后面依次的 12 个顶点为其下一层中的点 $v_j = -\pi/2 + \pi \times j/\text{nStacks}$，依此类推。由于有 12 条 8 层，所以一共有 98 个点(为什么?)。

法向量列表也比较好构造：norm[k]用来保存球面顶点 pt[k]处的法线。norm[k]是通过在参数形式下计算点(u, v)处的 $\mathbf{n}(u, v)$得到的。对于球面的情形尤为简单，因为 norm[k]等于 pt[k]。

这个例子的面片列表有 96 个面，其中 24 个是三角形。我们可以令顶部的三角形为最初的 12 个面，下一层中的 12 个四边形为其后的 12 个面，依此类推。前面的几个面包含数据如下：

顶点数：333…

顶点索引：0 1 20 2 30 3 4…

法线索引：0 1 20 2 30 3 4…

注意到，所有用来表示光滑形状的网格，它的法线索引 normIndex 总是等于顶点索引 vertIndex，所以数据结构中含有冗余信息(因此，对于这样的网格可以使用更加高效的数据结构。它可能是什么样子?)。以此种方式对球面进行多边形化还是比较简单的，但是要把更加复杂的形状进行多边形化就需要一些技巧了。本章末尾的进一步阅读材料中有更多的探讨。

最终我们需要一个方法，例如 makeSurfaceMesh()，对给出的曲面 $P(u, v)$构造出合适的网格。

请注意一些图形程序包的程序对三角形操作是高度优化过的。为了利用这一点，可以把球面离散化为三角形的集合，即把每个四边形再分成两个三角形。

这个简便的方法没有改变顶点但改变了面片列表，把每个四边形替换成两个三角形。比如，顶点为 2、3、15、14 的面被分割为两个三角形，其中一个的顶点是 2、3、15 另一个的顶

点是 2、15、14。

球面是旋转曲面的一个特例。一端逐渐变小的柱面同样是旋转曲面。可以很容易地构造出一端逐渐变小的柱面的网格。图 6.57 展示的逼近后的一端逐渐变小的柱面，它的 nSlices=8 并且 nStacks=1。它的顶面和底面是八边形(如果希望在网格中只使用三角形，也可以把它的外壁、顶面和底面分割成三角形(如何做呢?))。

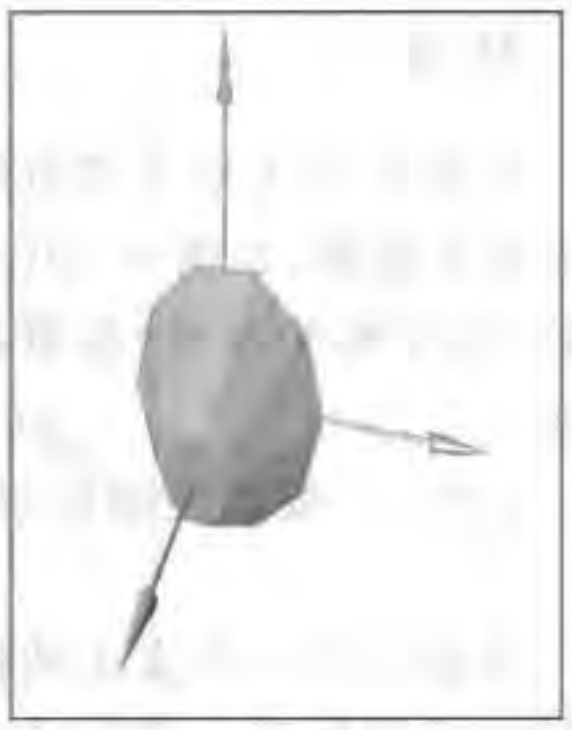


图 6.57　用网格近似逼近一端逐渐变小的柱面

练习

练习 6.5.8　构造球面网格

构造出通用球面的顶点、法线以及面片列表，这里 nSlices=6，nStacks=4。请选择方便的排序方式。

练习 6.5.9　只使用三角形构造网格

修改前面的练习让所有的面都为三角形。

练习 6.5.10　柱面和锥面的网格

构造出一端逐渐变小的柱面的顶点、法线以及面片列表，这里 nSlices=4，nStacks=2。

6.5.6　直纹面

下面定义和研究两类重要的曲面，这样就能对更多的曲面进行建模和分析。其中一类是在现实世界中经常遇到的“直纹面”，另一类是旋转曲面。直纹面被定义为如下形式的曲面：在其上的任意一点都可以找到一条通过它的直线使之完全落在曲面上。例如，锥面和一端逐渐变小的柱面，它们都是线性拉伸的形体所以肯定都是直纹面。但是球面就不是直纹面(为什么?)。

锥面

锥面是直纹面，这里它的一条曲线 $P_0(u)$ 退化成一个点 $P_0(u)=P_0$，即锥面的顶点，如图 6.58 所示。

在这种参数化形式下，所有的直线在 $v=0$ 时都经过 P_0，在 $v=1$ 时经过 $P_1(u)$。我们对锥面的一些特例应该比较熟悉。当 $P_1(u)$ 是圆时得到的是圆锥面，又当此圆所在的平面垂直于圆心与 P_0 的连线时得到的是直角圆锥面。图 6.58 使用的 $P_1(u)=(r(u)\cos u, r(u)\sin u, 1)$，这里的半径 $r(u)$ 按照正弦方式变化：$r(u)=0.5+0.2\cos(5u)$。

由于锥面是把一条曲线线性拉伸的结果，而在 $v=0$ 时曲线 $P_0(u)$ 收缩为一个点，因此锥面上的点 $P(u,v)$ 可以用等式(6.33)表示为

$$P(u,v)=(1-v)P_0+vP_1(u)\quad\{\text{一般的锥面}\}\tag{6.33}$$

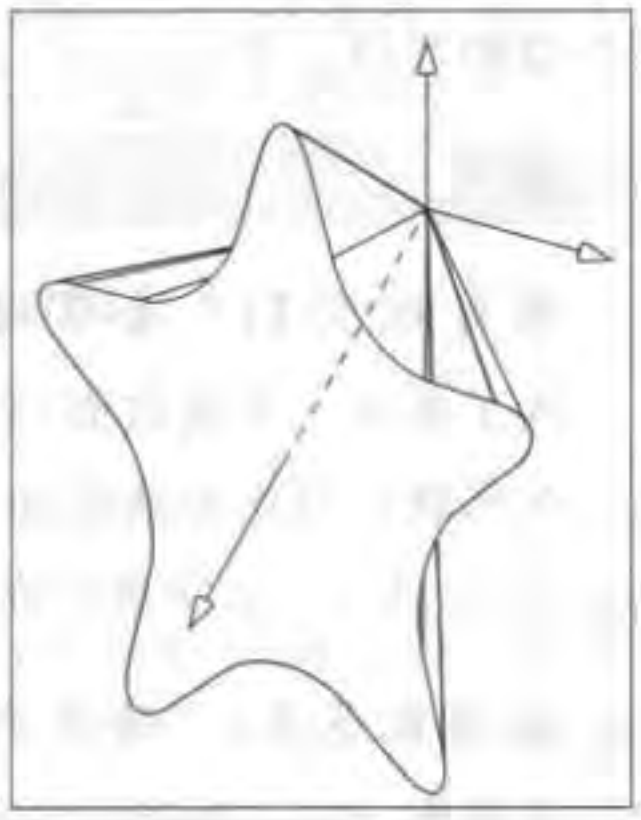


图 6.58　锥面

柱面

柱面是如下形式的直纹面,它的 $P_1(u)$ 是 $P_0(u)$ 平移后的版本:$P_1(u)=P_0(u)+\mathbf{d}$,$\mathbf{d}$ 是某个向量,如图 6.59(a)所示。有时柱面也可以描述为用端点是 $P_0(0)$ 和 $P_0(0)+\mathbf{d}$ 的线段(通常称为母线)沿着曲线 $P_0(u)$(通常称为准线)在不改变线段方向的前提下扫掠的结果。

因此,一般的柱面可以表达为参数形式

$$P(u,v)=P_0(u)+\mathbf{d}v \tag{6.34}$$

要想成为一个真正的柱面,曲线 $P_0(u)$ 要被限制在一张平面内。如果 $P_0(u)$ 是一个圆,得到的是圆柱面。方向 $\mathbf{d}$ 并不需要与平面垂直,但是如果垂直的话曲面称为直角柱面。通用柱面是直角柱面的一个例子。图 6.59(b)画的是"糖果彩带柱面",这里的 $P_0(u)$ 像一条彩带上下摆动。在练习中会进一步探讨彩带形状。

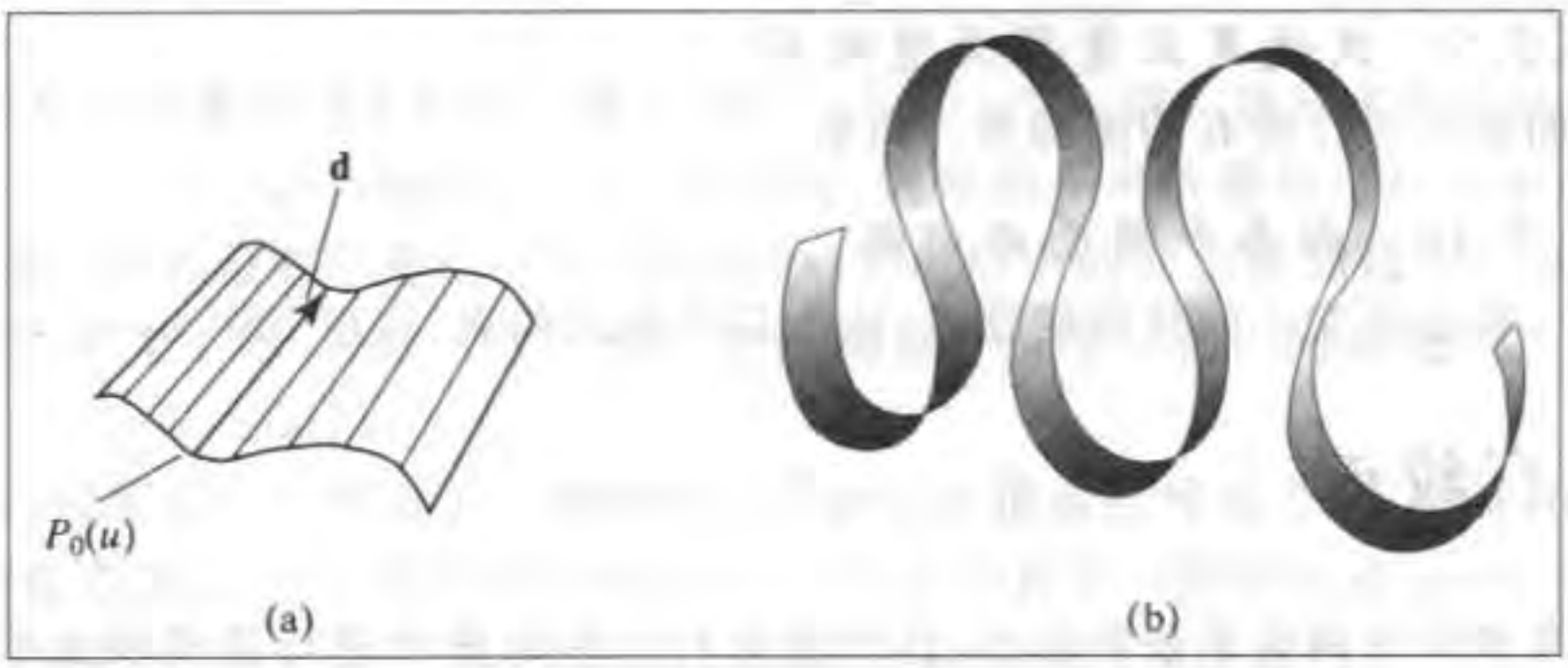


图 6.59
(a) 柱面;(b) 糖果彩带柱面

其他直纹面

还有许多其他有趣的直纹面。图 6.60(a)是一个双螺旋结构,两条螺旋线 $P_0(u)$ 和 $P_1(u)$ 相互缠绕。图 6.60(b)中的 Möbius 十分新奇,它仅有一条边界。练习中有这些曲面的参数表达形式。图 6.60(c)用 4 个直纹面搭建起一座穹顶。案例 6.8 考察了如何构建一座教堂的穹顶。图 6.60(d)是 Coons 面片,它以著名的图形学专家 Coons 命名。

练习

练习 6.5.11 金字塔是锥面

为了生成一个直纹面,它是一个底座为正方形的金字塔(如 Gizeh 金字塔),那么 $P_1(u)$ 是什么形状?写出此曲线和点 P_0 的表达式,使得金字塔底面位于 xy 坐标平面并且中心在原点,边长为 2。金字塔的高度是 1.5。

练习 6.5.12 糖果彩带柱面

请找出 $P_0(u)$ 的参数形式,使得可以较好的逼近图 6.59(b)中的糖果彩带柱面。可以考虑彩带是缠绕在相接的一系列半径为 1 的圆柱面之上的。第 i 个柱面的中心位于

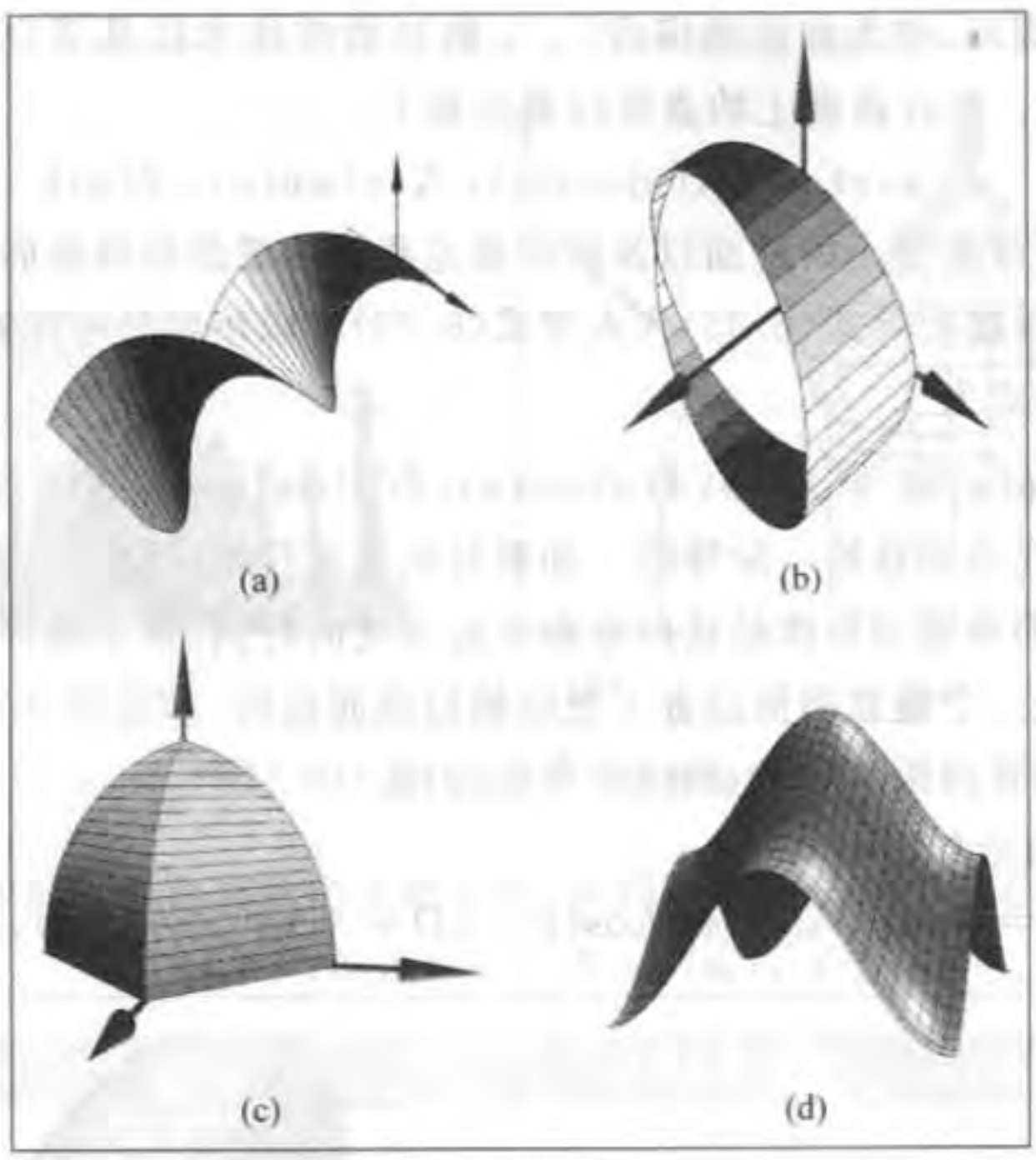


图　6.60

(a) 双螺旋面；(b) Möbius 带；(c) 穹顶；(d) Coons 补片

$(x_i, y_i)=(id, \pm r)$，这里交替的取＋、－，并且 $d^2=1-r^2$。r 取 0、1 之间的某个值。

练习 6.5.13　双螺旋

单条螺旋线的参数形式为$(\cos t, \sin t, t)$。请找出两条螺旋线 $P_0(u)$和 $P_1(u)$的表达式，它们都绕着 z 坐标轴旋转，但并不同步，角度相差 180°。请写出这两条曲线组成直纹面的参数形式。

练习 6.5.14　Möbius 带

请找出图 6.60 中 Möbius 带的参数形式。

提示：把一条线绕着 z 坐标轴旋转一圈，在此过程中同时对线实行一次扭转。$P_0(u)=(\cos(2pu), \sin(2pu), u)$，$P_1(u)=(\cos(2pu), \sin(2pu), 1-u)$，上述形式可以吗？

练习 6.5.15　是真正的插值吗？

请验证等式(6.34)中的 $P(u,v)$插值 4 条边界曲线，并且由此插值 4 个角点。

6.5.7　旋转曲面

前面已经说过，旋转曲面是把轮廓曲线 C 绕着一条轴进行旋转扫掠得到的。假设在 xy 坐标平面有一条轮廓线，用参数表示为 $C(v)=(X(v), Z(v))$。为了生成旋转曲面，我们把轮廓线围绕 z 坐标轴进行旋转并且用参数 u 控制旋转的角度。像先前一样，曲线绕着轴旋转从而在不同位置形成的曲线称为经线。完全扫掠一周会生成一个封闭的环，所以 v 为

常数的轮廓线都是圆环,称为曲面的纬线[①]。v 值处的纬线半径是 $X(v)$ 并且距离 xy 坐标平面的高度为 $Z(v)$。因此曲面上的点可以表示如下

$$P(u,v) = (X(v)\cos(u),\ X(v)\sin(u),\ Z(v)) \tag{6.35}$$

通用球面、一端逐渐变小的柱面以及锥面都是我们熟悉的旋转曲面的特例(它们的轮廓线分别是什么?)。直接把等式(6.35)代入等式(6.23)可以很容易地计算出旋转曲面的法向量(参见练习),其等式为

$$\mathbf{n}(u,v) = X(v)(\dot{Z}(v)\cos(u),\dot{Z}(v)\sin(u),-\dot{X}(v)) \tag{6.36}$$

这里的字母上的点代表函数的一阶导数。如果对向量实行归一化可以消去比例因子 $X(v)$。我们之前推导出的简单通用形体的法向量都是此等式的特例(参见练习)。

例如,圆环面是一个偏置圆周绕着 z 坐标轴扫掠而成的,参见图 6.61(a)。圆周的半径是 A,在 x 轴方向偏置的距离是 D,所以轮廓线为 $C(v)=(D+A\cos(v),\ A\sin(v))$。由此圆环面(图 6.61(b))表示如下:

$$P(u,v) = ((D+A\cos(v))\cos(u),\ (D+A\cos(v))\sin(u),A\sin(v)) \tag{6.37}$$

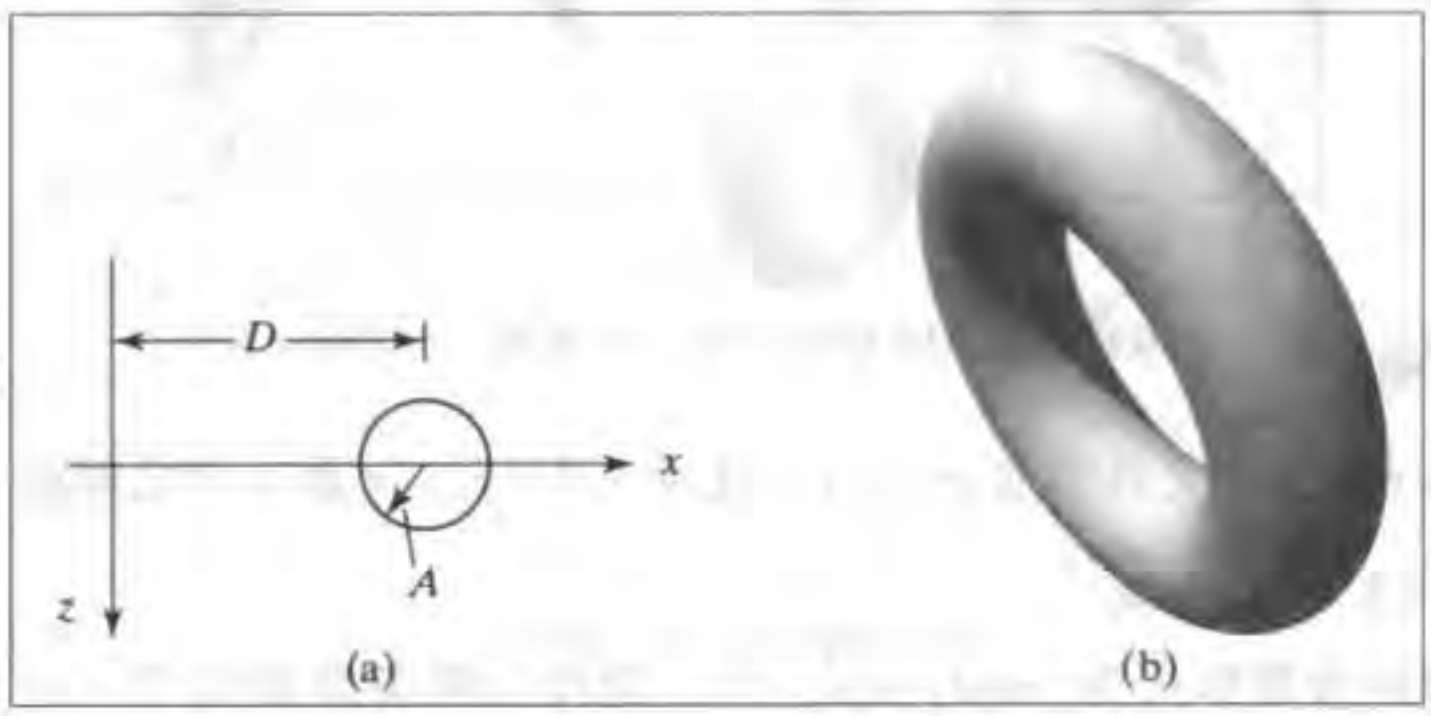


图 6.61 环面

我们会在练习中推导它的法向量等式。

通常我们扫掠平面曲线所使用的轴也位于这个平面,但是旋转曲面也可以围绕任意轴扫掠生成。一给定轮廓曲线围绕不同的轴可以扫掠出许多有趣的曲面。绕任意轴生成的旋转曲面其一般形式 $P(u,v)$ 将在练习中推导。

在程序中采用通常的方法可以构造出旋转曲面的网格(参加 6.5.4 节)。我们选择 u、v 值的点集 $\{u_i\}$ 和 $\{v_i\}$,然后通过 $P(u_i,\ v_i)$ 计算出顶点,通过 $\mathbf{n}(u_i,\ v_i)$ 计算出法线方向。通过把相邻的 4 个顶点用直线连接起来构造多边形面片。案例 6.13 会讨论这种方法。

图 6.62 是另一个例子,这里我们建模近似构造出位于印度 Agra 邦的泰姬陵穹顶,图 6.62(a)是这个穹顶的照片。图 6.62(b)是 xz 坐标平面上的轮廓曲线,图 6.62(c)是生成的旋转曲面。由于没有合适的参数等式,我们使用一系列数据点 $C_i=(X_i,\ Z_i)$ 的集合来描述轮廓(在第 8 章我们会纠正这个限制,在那里会使用 B 样条曲线生成基于给定数据点集的光滑参数曲线)。

---

① 更正式的讲,经线是曲面和过旋转轴的平面形成的交线,纬线是曲面和垂直于旋转轴的平面形成的交线。

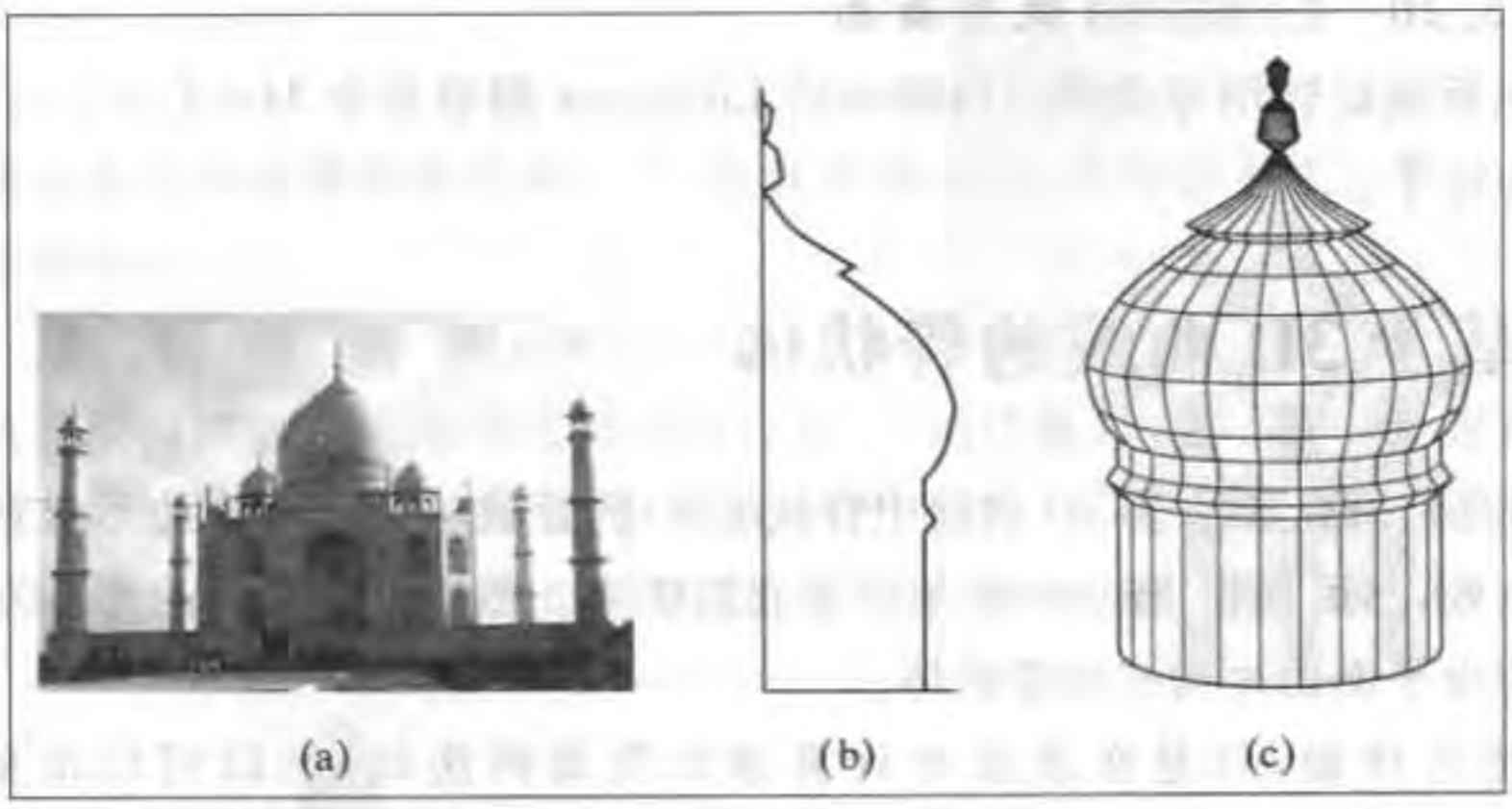


图 6.62　旋转曲面——泰姬陵穹顶

当旋转曲面的轮廓线是由离散的点组成时，得到的顶点 ij 与等式(6.37)稍有不同：

$$P_{i,j} = (X_j\cos(u_i),\ X_j\sin(u_i),\ Z_j)$$

练习

练习 6.5.16　作为旋转曲面的通用形体

描述通用球面、柱面、锥面的轮廓线的参数形式，然后再把它们表示为旋转曲面。

练习 6.5.17　围绕其他轴的旋转

考虑在 xz 平面内的轮廓曲线 $C(v)=(X(v),\ Z(v))$，和通过原点用单位向量 $\mathbf{r}$ 表示的任意轴。从等式(5.31)我们可以知道矩阵 $R_{\mathbf{r}}(\theta)$ 表示把一点绕轴 $\mathbf{r}$ 旋转 θ 弧度。

a. 由此验证把 $C(v)$ 绕轴 $\mathbf{r}$ 扫掠生成的旋转曲面为

$$(X(u,v),Y(u,v),Z(u,v),1) = R_{\mathbf{r}}(u)\begin{bmatrix} X(v) \\ 0 \\ Z(v) \\ 1 \end{bmatrix}$$

b. 用上式检验围绕 z 坐标轴旋转的这个特例。

c. 重复 b 步骤，围绕 x 坐标轴和 y 坐标轴旋转。

练习 6.5.18　计算法向量

(a)把等式(6.35)代入等式(6.23)推出旋转曲面法向量的等式(6.36)；(b)用这个结果计算通用球面、柱面以及锥面的法向量并且验证其与 6.5.2 节中的计算结果一致。验证圆环面的法向量为

$$\mathbf{n}(u,v) = (\cos(v)\cos(u),\cos(v)\sin(u),\sin(v))(D + A\cos(v))$$

并且推导出圆环面的内外部函数，然后用它的梯度计算法线。

练习 6.5.19　椭圆环面

计算如下两张旋转曲面的参数形式：(a)先把椭圆($a\cos(v)$，$b\sin(v)$)沿着 x 坐标轴偏置距离 R，然后围绕 y 坐标轴旋转。(b)同样的椭圆围绕 x 坐标轴旋转。

练习 6.5.20 Lissajous 旋转曲面

围绕 y 坐标轴旋转用等式(6.17)表示的 Lissajous 图形其中 $M=2, N=3, \phi=0$,请草绘出生成曲面的样子。

6.5.8 基于 3D 曲线的管状体

6.4.4 节我们研究了基于 3D 空间中脊线 $C(t)$ 的管状体。一个多边形被放置于每个有选择的脊线点处,然后根据 Frenet 标架计算它的朝向。然后把相邻多边形的对应点相连生成沿着脊线的由平面面片构成的管状体。

这里依然这样做,只是在顶点处计算原始曲面的法线,所以可以渲染得很光滑。图 6.63是包裹在螺旋线之外的管状体。请用它与图 6.40 比较一下。

如果把圆环$(\cos(u), \sin(u), 0)$包裹在脊线 $C(t)$之外,那么得到曲面的参数形式是

$$P(u,v) = C(v) + \cos(u)\mathbf{a}(v) + \sin(u)\mathbf{b}(v) \tag{6.38}$$

这里的向量 $\mathbf{a}(v)$垂直于曲线,并且 $\mathbf{b}(v)$同时垂直于曲线和 $\mathbf{a}(v)$。然后用通常的方式构造管状体的网格,对 $P(u,v)$进行采样,生成顶点、法线和面片列表。

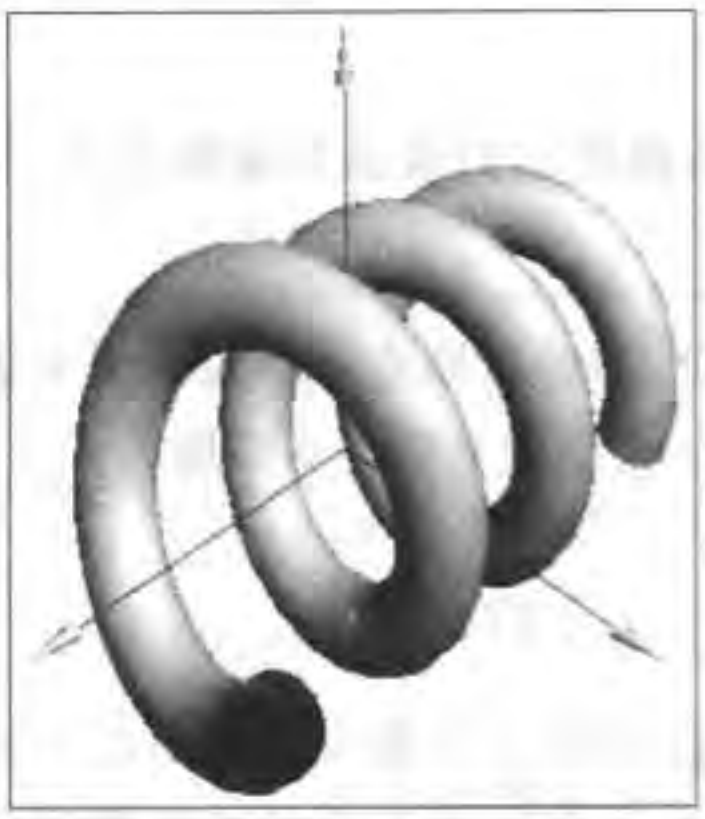


图 6.63 空间中的螺旋管状体

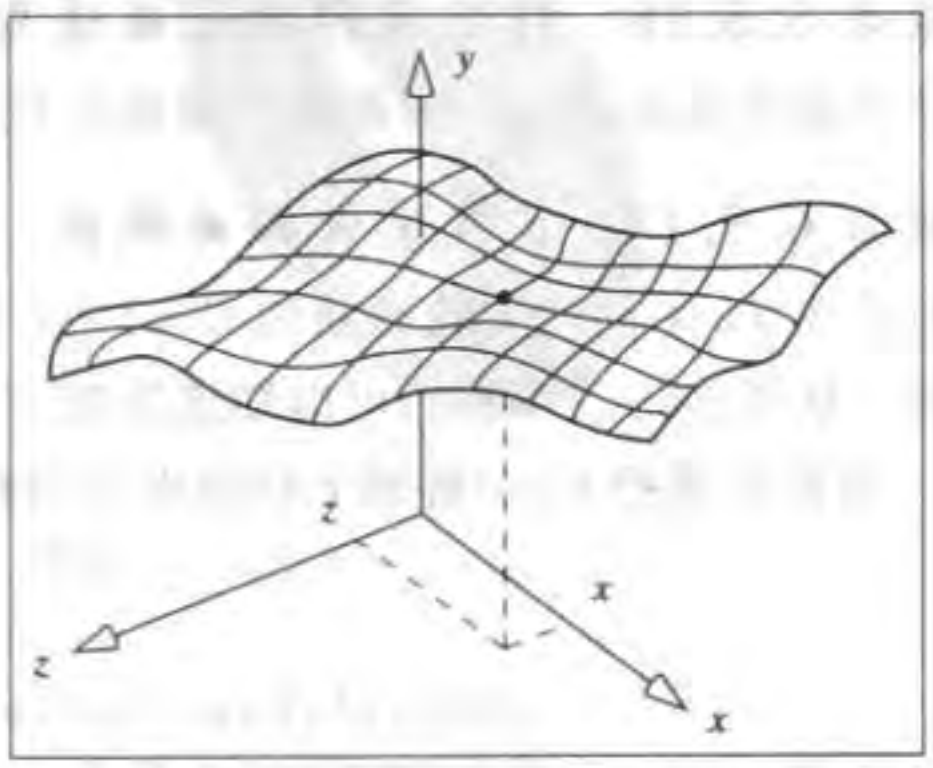


图 6.64 xz 坐标平面上的单值高度场

6.5.9 基于双变量显式函数的曲面

许多曲面形状在其中一个维度上是单值的,所以可以用双独立变量显式函数来表示它们。例如,图 6.64 所示的 xz 坐标平面上的单值高程面。曲面在(x,z)点的高度是某个 $f(x,z)$。这样的函数有时称为高度场【Bloomenthal97】。高度场经常用等式描述,例如高斯函数为等式(6.39):

$$f(x,z) = e^{-ax^2-bz^2} \tag{6.39}$$

(a 和 b 是常数),或者圆对称正弦函数

$$f(x,z) = \frac{\sin(\sqrt{x^2+z^2})}{\sqrt{x^2+z^2}} \tag{6.40}$$

以及波浪面

$$f(x,z) = e^{-2|x|-5|z|}\cos(4x-12z)$$

请用上述曲面与球面等曲面比较一下，后者在点(x,z)处对应不止一个 y。单值函数有一个简单的参数形式：

$$P(u,v) = (u,\ f(u,v),\ v) \tag{6.41}$$

它们的法向量可以直接应用等式(6.23)计算。(请计算)这里 u 和 v 是直接作为参数函数的分量出现的。因此 u 轮廓线位于常值 x 平面内，v 轮廓线位于常值 z 平面内。图 6.65(a)是等式(6.39)的图形，图 6.65(b)是等式(6.40)的图形。

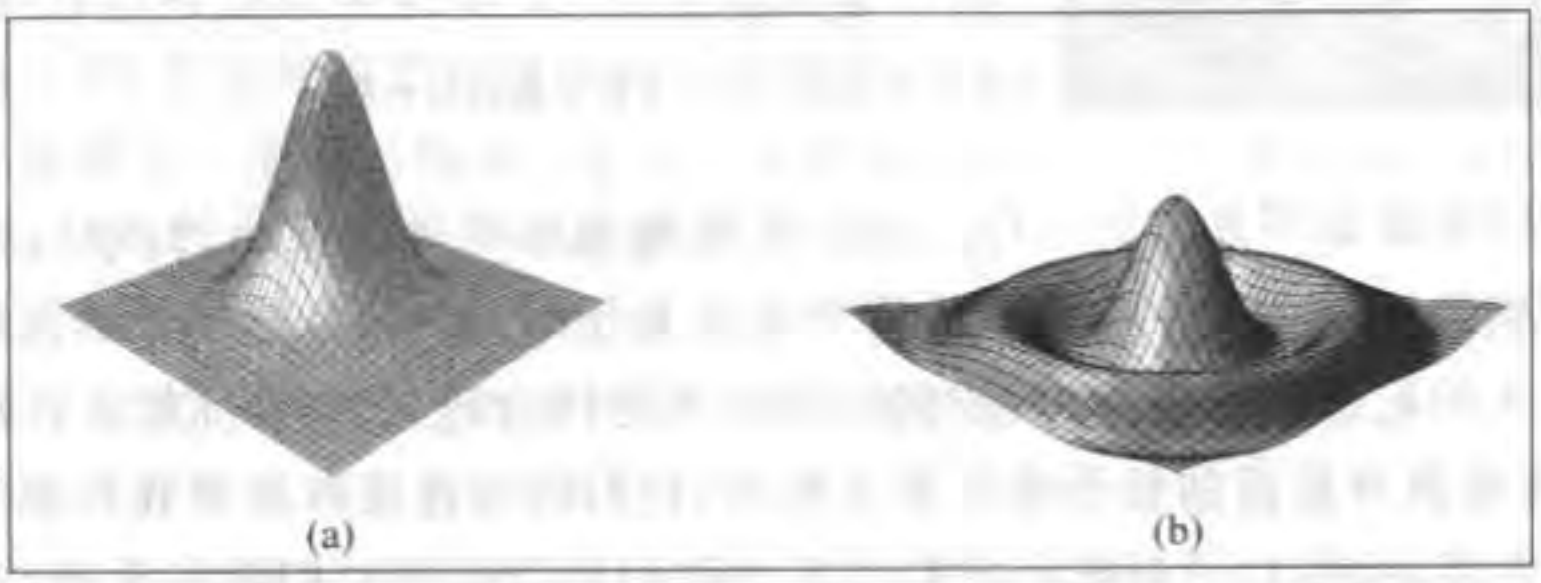


图 6.65　两种高度场

(a) 高斯函数；(b) 正弦函数

图中的每条线是 k 取某些值时平面 $x=k$ 或 $z=k$ 与曲面的交线。这样的绘制方法是为了阐明数学函数的一些性质。

练习

练习 6.5.21　显函数表示的二次曲面

椭圆抛物线可以写为 $z=f(x,y)$，所以它有另一种参数形式$(u,\ f(u,v),\ v)$。$f()$是什么？参数形式在什么情形中有用？还有什么二次曲面可以如此表示？

6.6　粒子系统与基于物理的系统

6.6.1　粒子系统

图 6.66(a)是一幅截图，取自使用粒子系统模拟火焰的动画。本章的前言提到，粒子系统有能力对大量的粒子进行跟踪。在场景中每个粒子都拥有位置和速度，除此之外还可能拥有色彩、生命期，在一些场合甚至包含尺度、透明度和形状等属性。根据应用需要，会使用随机数生成器对每种属性赋值。

通常要使得表现更加逼真，应用程序会使用自己的属性集合对每个粒子建模；特别是你希望对每个粒子的位置和速度进行跟踪和更新使得它看上去更加自然。这样水滴从喷泉中

图 6.66

(a) 用粒子模拟的火焰；(b) 用粒子模拟的云彩

喷涌并回落的过程就如同真实的一样。由于要精确地跟踪每个粒子的运动，所以构建一个成功的粒子系统的困难在于存储信息所需要的巨大开销。一种自然的做法是把粒子的状态存储在一个庞大的记录数组中，然后依据时间的流逝用合理的规则和方法更新每个粒子的状态。例如，有些具有重量的粒子会被重力作用，它们的加速度取决于我们熟知的牛顿定律 $f=ma$。另一方面，有些粒子系统会把粒子考虑得很小以至于它们没有重量，在这种情况下重力就不起作用了。图 6.66(b)场景中的云彩就没有重量。一些很好的粒子系统教程在网上都可以获取，其中一个优秀的资源是 www.nehe.com。

6.6.2 基于物理的系统

建模系统有一个分支被称为基于物理的系统，它是利用数学方式描述场景中各种物体间的受力行为。例如，基于物理建模的系统可以用于布料模拟，参见图 6.67(a)中的旗帜。图 6.67(b)中一端带有金属球的小棍受三块磁铁的吸引而来回摇摆。摇摆运动取决于所受的磁力和牛顿定律，这样就可以预测出小棍在合力作用下的加速度($f=ma$)。

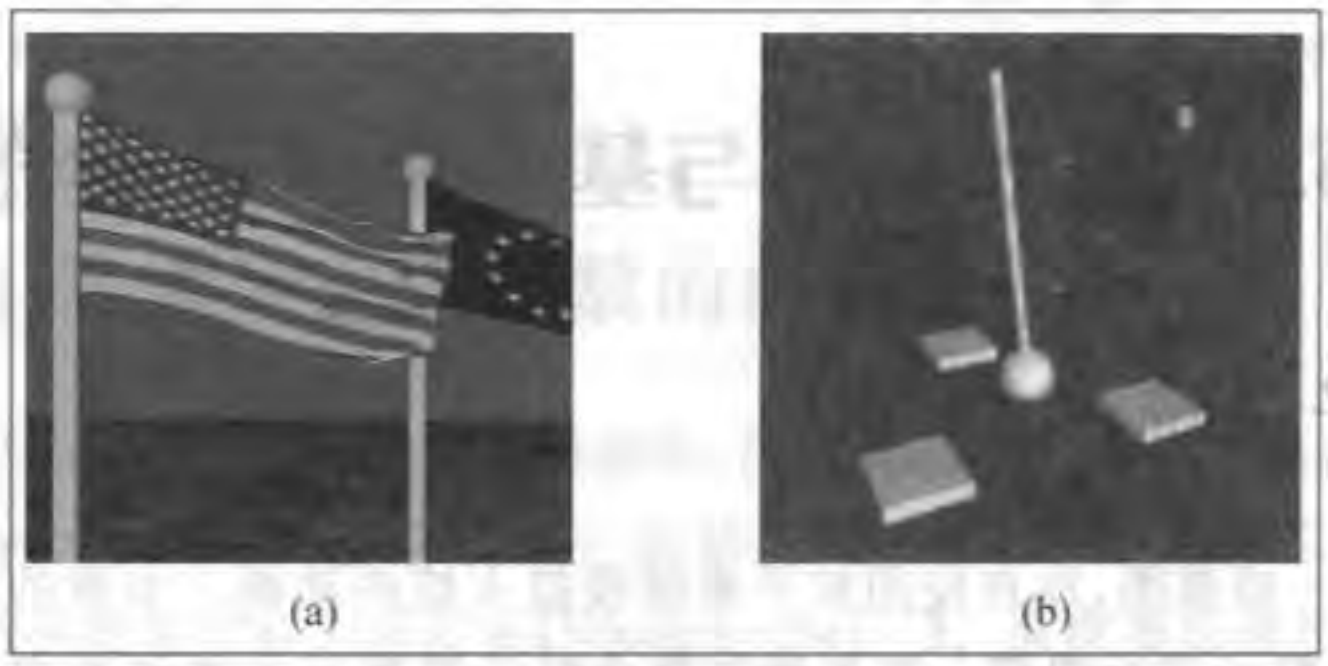


图 6.67

(a) 风中飘扬的旗帜；(b) 被磁铁吸引的摆锤

其他的例子还包括相互碰撞变形的胶质液滴；云彩、海浪和其他涌动的物体；甚至包括有生命的物体。在这些例子中，一个关键因素是不同的物体相互碰撞过程中可能发生的变

形。为了表达出系统的这种交互作用过程需要用到很复杂的数学，并且事实上基于物理的系统一个很明显的特征就是牵扯到使用相关数学的困难性（常微分等式、偏微分等式等）。它们必须通过数值方法求解，这就使得相关算法的运行速度很慢。

一个很吸引人的、在一些电脑游戏玩家里也很受欢迎的例子就是地形的建模与观察。伴随着动画的进行，观察者可以飞跃模拟地形，看到下面海洋或瀑布泛起的层层波浪、从火山口喷出的烟尘碎屑以及其他各种真实的物理现象。这些地形通常包含海洋中屹立的山脉，每座山都有山峰和巨大的山谷，海水把整个山脉环绕起来。图 6.68 是这种模拟动画中的一幅截图。

图 6.68　地形模拟

地形通常建模为一个网格物体：定义一张曲面，它可以是基于类似等式（6.39）给出的高度场，把其中某些四边形面片进一步分割，用这种方法持续地对它进行细化，直到满足我们需要的层次细节。然后按照第 9 章的方法把这些面片作为填充着色多边形进行渲染。

虚拟地形很具有视觉冲击力，当飞过伴随着上下翻滚的波浪的地形时就更加引人入胜。我们在第 7 章将探讨如何让观察者飞过地形。本书的配套网站有许多这样的动画。

本章小结

本章关心的是建模和绘制 3D 对象的曲面。这就要找到曲面形状合适的数学描述，并且创建高效的数据结构来保存曲面足够多的细节，加快曲面的渲染过程。我们开发出了 Mesh 类，它有三个列表：顶点、法向量以及面片列表。这种数据结构可以高效地保存诸如多面体这种由平面构成的物体的相关几何数据，并且还可以保存充足的数据来建模多边形表皮用以近似逼近光滑弯曲曲面。

我们指出一旦构造出了网格数据结构，那么在 OpenGL 环境下进行绘制就变得十分容易。也可以很容易地把网格存储在一个文件中，然后在程序中再把它读取出来。

现代的渲染算法使用面中每个顶点的法向量来决定面中不同点处绘制的明暗程度。如果要把面绘制得比较单调，那么就把同一个法向量—面自身的法向量—赋予面的每个顶点。如果网格要表示一个光滑弯曲的曲面，那么面片每个顶点的法向量被设置为原始曲面在此点处的法向量，然后渲染算法使用插值来产生图片中光滑渐变的效果（将在第 10 章讨论）。因此如何保存网格中的法向量取决于设计者希望物体所表达出来的样子。

我们探讨了大量出现在流行应用程序中的多面体形体，并且开发了生成某些多面体网格的技术。这里要特别注意面顶点的法向量使用的是面自身的法向量。我们也研究了许多类光滑物体，包括经典的二次曲面、柱面以及锥面，然后探讨了如何利用曲面参数形式合适的导数来计算每点处的法线方向。

下一节的案例分析会详细阐述其中的某些思想，读者不应该错过。其中有一些我们会从理论高度进一步分析。我们会大致描述计算法向量的 Newell 法的推导方法，然后要求读者补齐一些细节。其他的案例分析会针对一些更加有趣的形体要求读者设计出方法生成并绘制它们。

案例分析

案例 6.1 用文件存储网格

(难度：Ⅱ)我们打算让 Mesh 类支持使用文件存储与读取 Mesh 对象的功能。我们尽量把文件格式设计得简单一些。文件的第一行列出了网格中顶点、法线以及面片的数目。然后列出每个顶点的值，它是一个浮点数三元组(x_i，y_i，z_i)。每一行会有若干个顶点。然后依次列出每个法向量，它仍然是浮点数三元组。最后列出的是每个面片的信息，格式如下：

- 面片中顶点的个数
- 面片中顶点的索引列表
- 面片中顶点法线的索引列表

例如，图 6.5 中的仓房模型可以存储为：

```
10 7 7
0 0 0    1 0 0    1 1 0    1 1.5 0    0 1 0
0 0 1    1 0 1    1 1 1    1 1.5 1    0 1 1
-1 0 0  -0.477 0.8944 0    0.447 0.8944 0
1 0 0    0 -1 0    0 0 1    0 0 -1
4    3 4 9 8    1 1 1 1
4    2 3 8 7    2 2 2 2
4    1 2 7 6    3 3 3 3
4    0 1 6 5    4 4 4 4
5    5 6 7 8 9    5 5 5 5 5
5    0 4 3 2 1    6 6 6 6 6
```

它的第一个面片是四边形，其顶点编号为 0、5、9、4，最后的两个面片是五边形。

在程序中为了把网格从文件中读取出来，你也许用得上图 6.69 中的代码，给它传递文件名，它就会打开相应的文件并且把数据读取到 Mesh 对象中，如果操作成功就返回 0。如果返回了非 0 值，就表明有错误发生，例如无法找到文件(在程序中还应该对格式是否错误做一些附加测试，例如应该出现整数的地方却出现了浮点数)。

因为在取得 numVerts、numNorms 和 numFaces 之前并不知晓网格的大小，所以保存顶点、法线和面片的数组是在运行时动态分配得到合适的空间。

本书的网站上有很多此种格式的文件(它们的后缀是.3vn)。

同样，可以很容易地编写函数 int Mesh::writeMesh(char * filename)用来把网格对象写入文件。

编写一个应用程序从文件中读取网格对象，然后采用图 6.17 的方法把它们绘制出来，并且还允许用户把网格对象写入文件。初步的时候可以先让应用程序生成四面体和仓房的网格。

```
int Mesh:: readmesh(char * fileName)
{
    fstream infile;
    infile.open(fileName, ios::in);
    if(infile.fail()) return-1;                //错误-无法打开文件
    if(infile.eof()) return-1;                 // 错误-文件为空
    infile>>numVerts>>numNorms>>numFaces;
    pt=new Point3[numVerts];
    norm=new Vector3[numNorms];
    face=new Face[numFaces];
    //检查内存是否充足:
    if( !pt || !norm || !face)return-1;        // 内存分配失败
    for(int p=0; p<numVerts; p++)              //读取顶点
        infile>>pt[p].x>>pt[p].y>>pt[p].z;
    for(int n=0; n<numNorms; n++)              // 读取法线
        infile>>norm[n].x>>norm[n].y>>norm[n].z;
    for(int f=0; f<numFaces; f++)              // 读取面片
    {
        infile>>face[f].nVerts;
        face[f].vert=new VertexId[face[f].nVerts];
        for(int i=0; i<face[f].nVerts; i++)
            infile>>face[f].vert[i].vertIndex
                >>face[f].vert[i].normIndex;
    }
    return 0;                                  //成功
}
```

图 6.69　把网格文件读取到内存中

案例 6.2　Newell 法的推导

(难度：Ⅱ)Newell 法可以基于多边形的顶点利用等式(6.1)来计算出它的法线，现在我们来揭示 Newell 法背后的理论基础。在我们展开讨论的时候会根据需要讲述一些必要的数学知识；同时要求读者证明一些过渡的结论。请注意，你只需要证明标出了“作为本案例分析的一部分，请证明”字样的那些步骤。其他的步骤供你参考。

本讨论中，要处理的多边形面片 P(凸的或非凸的)由 N 个 3D 顶点给出：

$$P=\{P_0,P_1,\cdots,P_{N-1}\} \tag{6.42}$$

我们打算要证明出为什么等式(6.1)在 P 是平面的时候可以精确地计算出它的法向量 $\mathbf{m}=(m_x,\ m_y,\ m_z)$，而当 P 不是平面时可以提供一个比较好的法线均值。

推导

图 6.70 把 P 投影到(正交投影——沿着主轴方向，参见第 5 章)每个主平面上：即平面 $x=0$、$y=0$ 和 $z=0$。每个投影都是 2D 多边形。我们首先要证明 $\mathbf{m}$ 的分量与这些投影多边形的面积 A_x、A_y 和 A_z 成正比。

为了简单起见,这里的 P 是一个三角形我们称为 T,参见图 6.71。假设它的单位法向量是 $\mathbf{m}$。进一步令 T' 表示 T 在平面上的投影,它的单位法线为 $\mathbf{n}$。我们证明 T' 的面积,用 $\mathrm{Area}(T')$ 表示,是 T 的面积 $\mathrm{Area}(T)$ 的一部分,并且这部分可以简单地用点积表示为

$$\mathrm{Area}(T') = (\mathbf{m} \cdot \mathbf{n})\ \mathrm{Area}(T)$$

如图所示,三角形 T 有两条边用 $\mathbf{v}$ 和 $\mathbf{w}$ 表示。

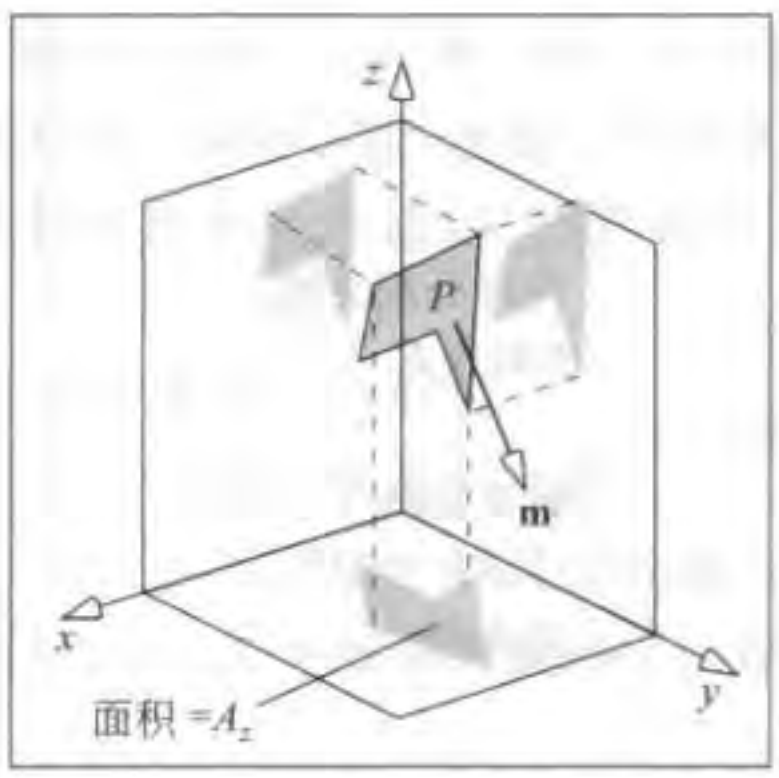


图 6.70 使用投影面积计算法向量

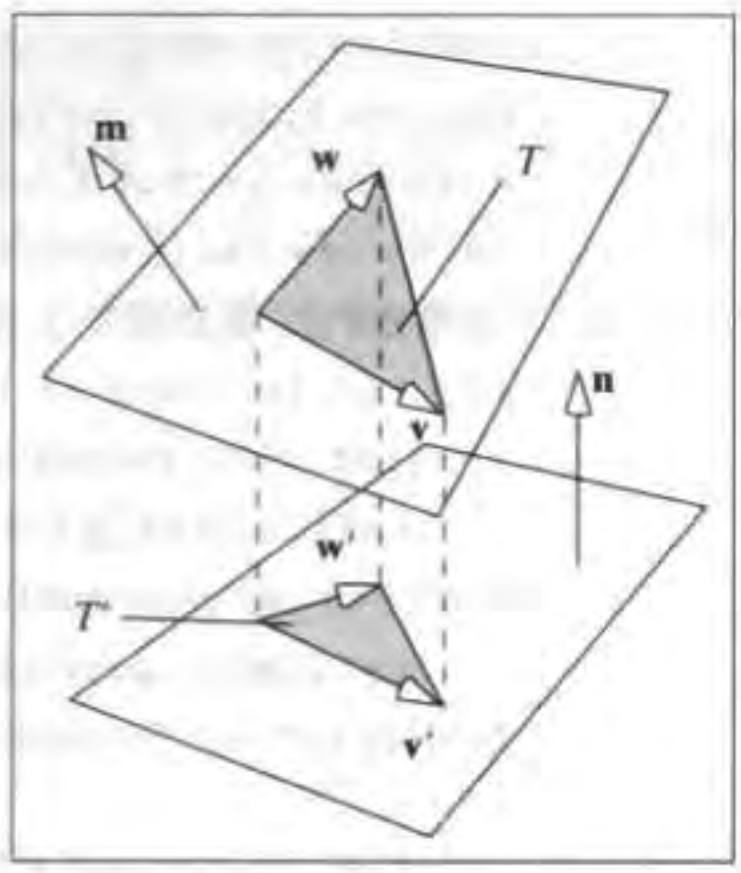


图 6.71 正交投影对面积的影响

作为本案例分析的一部分,请证明 T 的面积 $\mathrm{Area}(T)=\frac{1}{2}|\mathbf{w}\times\mathbf{v}|$,这里要用到叉积的大小与方向的知识。另外作为本案例的一部分,还请证明 $\mathbf{v}\times\mathbf{w}=2\mathrm{Area}(T)\mathbf{m}$。现在因为三角形 T 由向量 $\mathbf{v}$ 和 $\mathbf{w}$ 定义,所以投影三角形 T' 可以用投影向量 $\mathbf{w}'$ 和 $\mathbf{v}'$ 定义。作为本案例的一部分,请推出用 $\mathbf{w}'$ 和 $\mathbf{v}'$ 的叉积表示的 T' 面积的表达式。但是我们的目标是使用原始的 $\mathbf{v}$ 和 $\mathbf{w}$ 而不是它们的投影来表示 T' 的面积。利用第 4 章中生成正交投影的知识可以比较容易地解决上述问题。

作为本案例分析的一部分,用 $\mathbf{v}$ 和 $\mathbf{w}$ 来表示投影向量 $\mathbf{w}'$ 和 $\mathbf{v}'$,并且证明 T' 的面积等于 $\mathrm{Area}(T)\mathbf{m}\cdot\mathbf{n}$。

作为本案例分析的一部分,请证明上述结果可以推广到任意的平面多边形 P 和它的投影 P'。

由于点积与某一角度的余弦值成正比,作为本案例分析的一部分,请证明 $\mathrm{Area}(T')=\mathrm{Area}(T)\cos\phi$,并且说明角度 ϕ 的含义。

作为本案例分析的一部分,请证明上述定义的面积 A_x、A_y 和 A_z 分别等于 Km_x、Km_y 和 Km_z,K 是某个常数。因此面积 A_x、A_y 和 A_z 与 m_x、m_y 和 m_z 成相同的比例。

所以为了计算 $\mathbf{m}$,只需要计算出向量 (A_x, A_y, A_z) 然后把它归一化。现在我们证明如何直接使用顶点信息计算由等式(6.42)给出的 P 在 xy 坐标平面上的投影面积。其他两个方向的投影也同理。

每个 3D 顶点 $P_i=(x_i, y_i, z_i)$ 正交投影到 xy 坐标平面上形成 $V_i=(x_i, y_i)$。图 6.72 是投影多边形 P' 的例子。

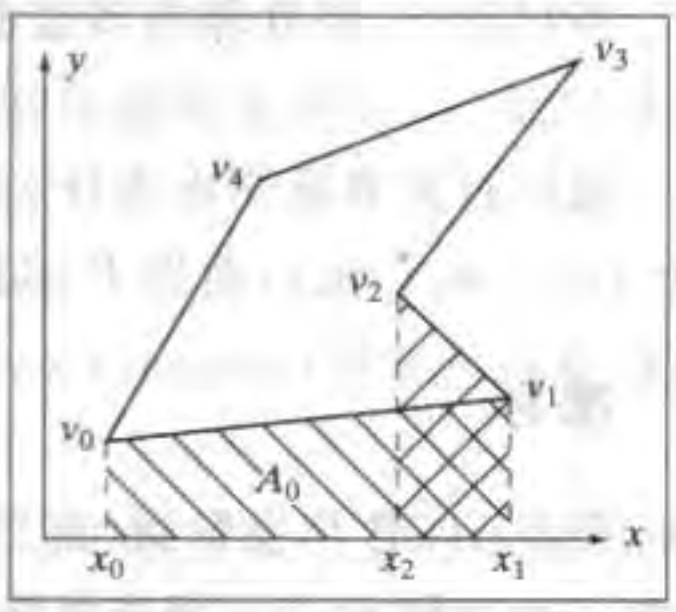


图 6.72 计算多边形的面积

P'的每一条边与 x 坐标轴可以组成一个梯形区域。

这种梯形的面积可以用底边乘以多边形的边中点得到。例如，图中 A_0 的面积等于 $A_0 = 0.5(x_0 - x_1)(y_0 + y_1)$。如果 x_0 在 x_1 的左侧那么上式取负值，否则取正值。我们对每一条边都使用这个等式。对第 i 条边我们定义

$$A_i = \frac{1}{2}(x_i - x_{\mathrm{next}(i)})(y_i + y_{\mathrm{next}(i)})$$

这里当 i 等于 $N-1$ 时，next(i)等于 0，其他情况等于 $i+1$。由于它实际上就等于等式(6.1)中多边形 P 的法线表达式，这也就完成了 Newell 法的推导。

即使多边形的两条相邻边共线(此时它们的叉积为 0)，它们依然可以贡献出的相应的面积。把所有 A_i 的面积，包括正的和负的，相加起来得到一个值，这个值就是多边形的面积或者是其负值。

现在我们要问，**m** 指向两个方向中的哪一个？也就是说，当你把右手的指头环绕多边形的顶点从 P_0 到 P_1 再到 P_2 以此类推的形式蜷缩起来，如图 6.73 中的箭头方向，那么 **m** 是大拇指指向的方向还是其反向？

m 的确就是图 6.73 所示的方向。因此对于一个具有明确内外部的网格，如果从外部观察其顶点是逆时针方向排列，那么我们就说此时的 **m** 是外法线。

案例 6.3　棱柱

(难度：Ⅲ)编写一个应用程序可以让用户使用鼠标绘出棱柱的底面多边形。然后程序生成棱柱的顶点、法线以及面片列表并把棱柱显示出来。

图 6.74(a)是用户的绘制区域：屏幕上的一块正方形。用户在正方形内点击鼠标左键指定一系列的点，结束此过程可以点击右键。

在 3D 空间中，这个正方形是 xy 坐标平面上的单位正方形如图 6.74(b)，棱柱的底面在其内部。这样我们就可以确定 3D 空间中底面的大小。棱柱是底面沿 z 坐标轴扫掠单位长度生成的。用这个程序生成若干个用户指定的棱柱。看一看你是否可以用 OpenGL 实现正确绘制非凸底面多边形。

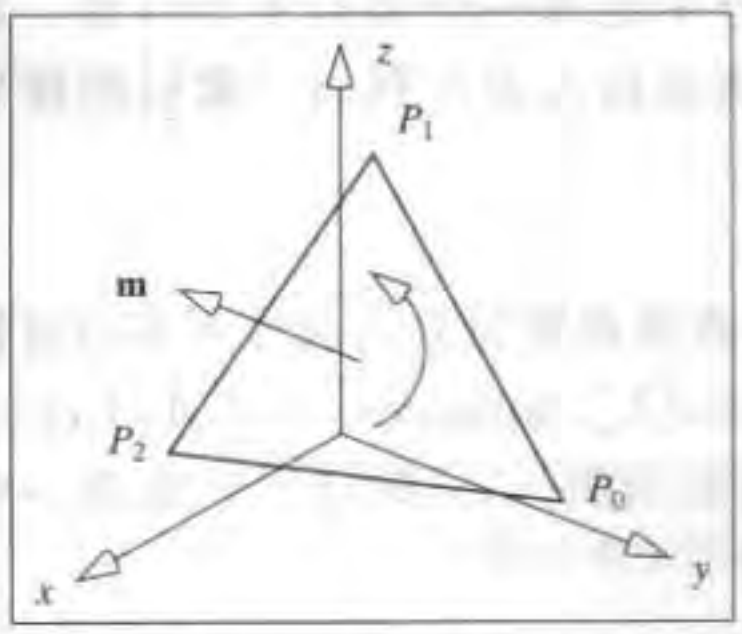


图 6.73　Newell 法计算的法线方向

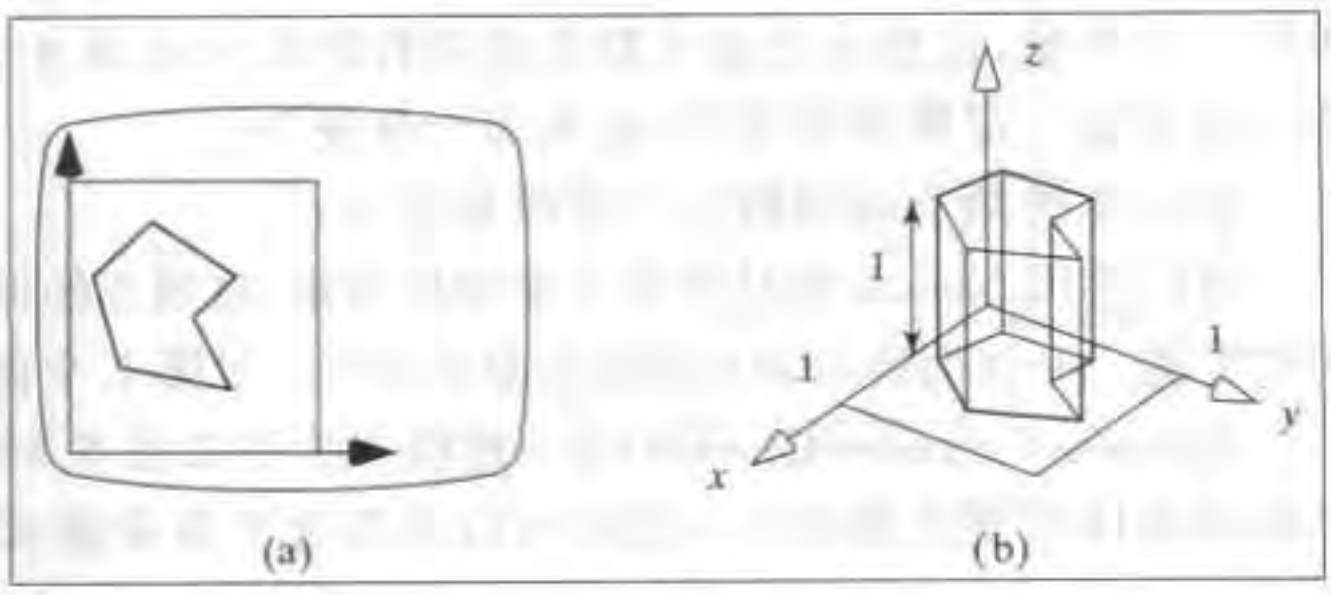


图 6.74　设计并生成一个棱柱网格

案例 6.4　棱柱数组与四边形条带拉伸体

(难度：Ⅲ)编写 6.4.2 节中讲述的两个方法：

```
void Mesh::makePrismArray(<…suitable argument…>);
```

```
void Mesh::makeExtrudedQuadStrip(Point2 p[], int numPts, Vector3 d);
```

它们用来生成棱柱数组以及四边形条带拉伸体的网格。

(a) 棱柱数组：选择一种恰当的数据结构来表示棱柱数组。注意到 makePrismArray() 与生成单个棱柱网格的方法很相似。练习用这个方法构造至少两个字母形状'K'和'W'(如果你愿意也可以试着生成'D')。

(b) 用来表示管状体的四边形条带拉伸体：构造网格顶点、法线与面片列表的过程实际上就是向这些数组列表中放入恰当的索引序号。为了有助于编写这个方法，请考虑用等式(6.7)描述的底面四边形条带，其顶点为

$$四边形条带 = \{p_0, p_1, \cdots, p_{M-1}\}$$

这里的 $p_i=(x_i, y_i, 0)$ 位于 xy 坐标平面，如图 6.75(a)所示。当拉伸后，每两个连续的顶点对构成了管状体的一个腰，如图 6.75(b)所示。管状体中共有 $num=M/2-1$ 个部分。

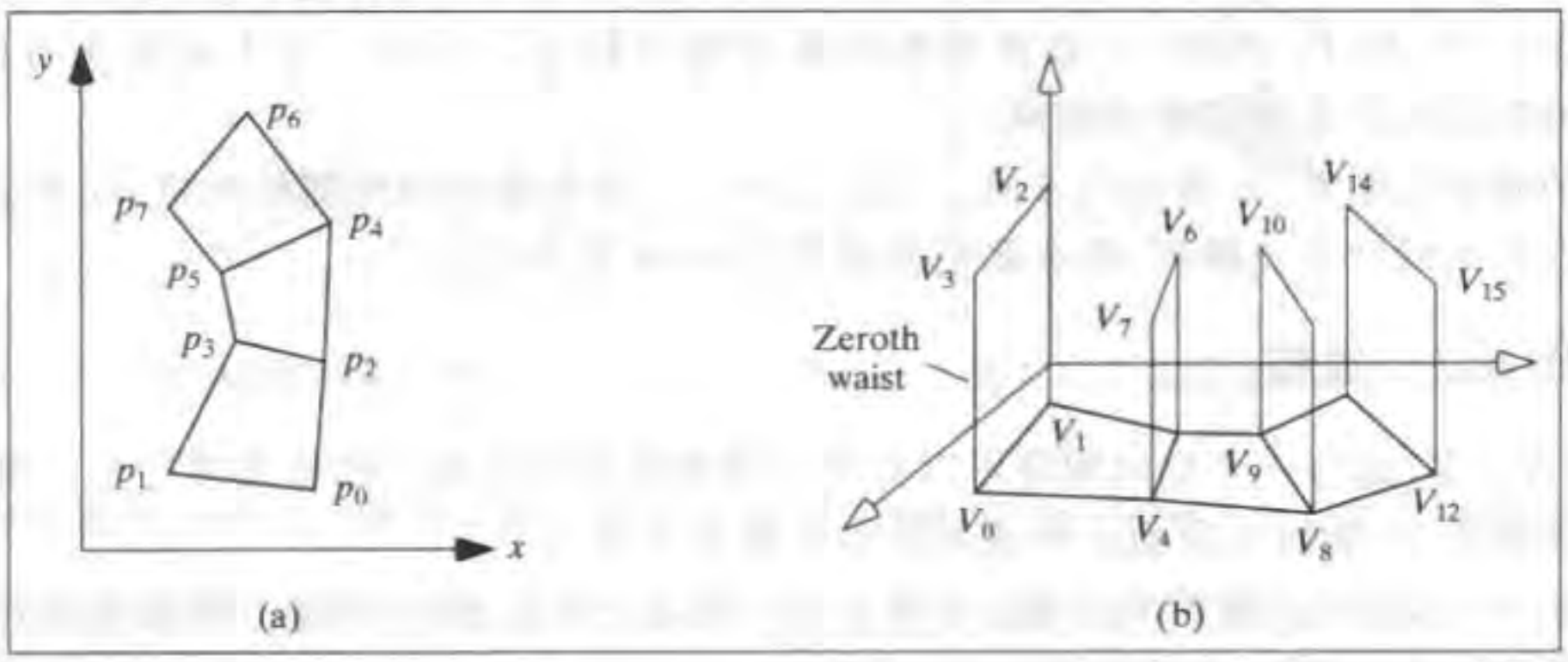


图 6.75 设计并生成一个棱柱网格

0 号腰含有顶点 p_0、p_1、$p_1+\mathbf{d}$ 和 $p_0+\mathbf{d}$，这里的 $\mathbf{d}$ 是拉伸向量。我们使用图 6.75(b)的方式把顶点添加进顶点列表中。

现在我们来处理面片列表。我们先添加管状体中每一部分的外壁，然后再加入它的端面(例如，第一个端面要用到第一个腰的顶点)。num 个部分中的每一个都有 4 个外壁。对于每一个外壁，依照从外面观察是逆时针的方向把其 4 个顶点加入面片列表。索引的排列形式还是有一定规律模式的，但是却十分复杂。

管状体的两个端面的顶点索引是什么？

可以使用 Newell 法计算每个面的法向量，这项工作可以在顶点索引放入面片列表的时候同时完成。一张面所有顶点的法向量都相同：对第 L 个面，face[L]. normindex={L,L,L,L}。

用 makeExtrudedQuadStrip()建模并绘制如图 6.38 所示的拱门，同时还可以处理一些字母状形体只要它的底边多边形可以用四边形条带表示。

案例 6.5 基于参数曲面的管状体和蛇形体

(难度：Ⅲ)编写并且测试以下函数

```
void Mesh::makeTube(Point2 P[], int numPts, float t[], int numTimes)
```

它把顶点为 $P_0, P_1, \cdots P_{N-1}$ 的多边形包裹在脊线 $C(t)$ 外生成由平面组成的网格。在脊线的 t_0、$t_1 \cdots t_{M-1}$ 处生成管状体的腰，在每个 $C(t_i)$ 点处构造出 Frenet 标架。函数 $C(t)$ 在程

序中是作为等式硬编码进去的。

请至少用螺旋线和 Lissajous 图形作为脊线进行测试。

案例 6.6　构造离散的旋转曲面

(难度：Ⅲ)编写一个应用程序允许用户用鼠标指定一个物体的轮廓，如图 6.76。然后它生成其旋转曲面的网格并且进行显示。此程序还要把网格数据按照案例分析 6.1 中的格式写入文件。

图 6.76 展示了用户的绘制区域：屏幕上的一个正方形。用户用鼠标在此正方形中指定一系列的点。

把这个正方形看成是 3D 空间中 xz 坐标平面上的单位正方形。这样轮廓线在 3D 空间中就有了尺寸。然后按照用户指定的步数，围绕着 z 坐标轴扫掠轮廓线生成旋转曲面。

用此程序生成若干个用户输入的旋转曲面。

案例 6.7　边列表与线框图

(难度：Ⅱ)通过只绘制出网格的每条边可以得到网格的线框图。请编写完成此项工作的函数 void Mesh::drawEdges(void)。它仅需要遍历每个面，然后用线段连接起相邻的顶点。这样每条线段会被绘制两次(为什么?)。

在某些苛刻的条件下，这种低效的方式是不能被容忍的。根据这种情形，可以构造出网格的边列表，网格的每条边只在此列表中出现一次。边列表是索引对组成的数组，索引对中的两个索引指向边的两个端点。请写出构造任意网格边列表的算法。它遍历网格中的每个面，找到其上的每一条边，如果此边没有出现在列表中就将其加入。

请注意通常并不能够依据边列表和顶点列表生成面片列表。图 6.77 是一个经典例子。仅仅通过边列表没有办法说明哪些是面而哪些不是面：甚至单从立方体的线框图无法获知盒子是封闭的还是打开了的。因此面片列表比边列表拥有更多的信息。

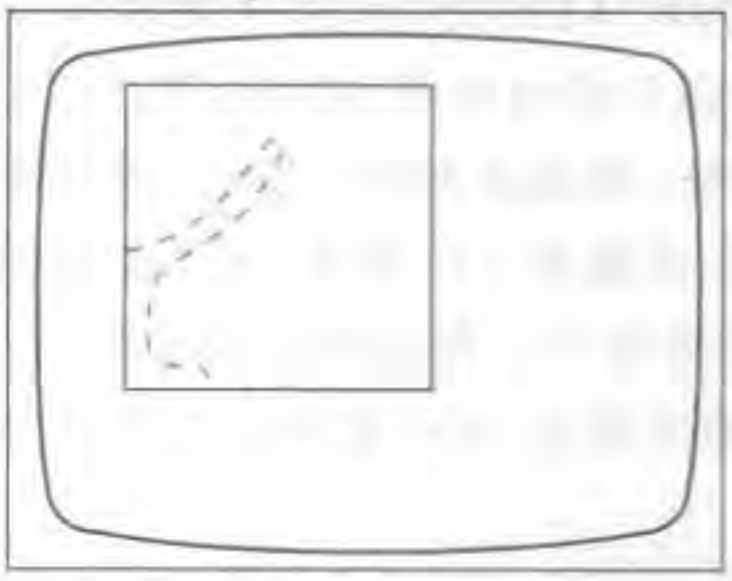


图 6.76　设计旋转曲面的轮廓

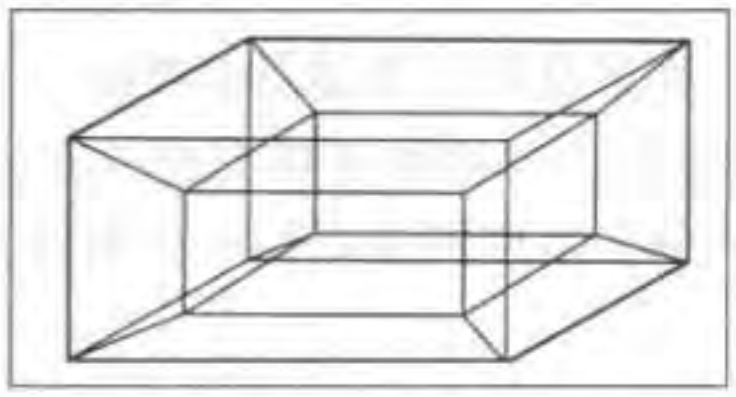


图 6.77　由二义性的物体

案例 6.8　穹顶

(难度：Ⅲ)许多经典建筑的屋顶都成拱形称为穹顶。图 6.78(a)是构造在正方形底座上的穹顶【Fleming66】。它由 4 张曲面构成，每张都是直纹面，在顶端交汇在一起。图 6.78(b)是构造在八边形底座上的穹顶，由 8 张曲面构成。针对建在正方形和八边形上的穹顶，编写一个函数用来生成它们的网格模型并绘制出线框图。

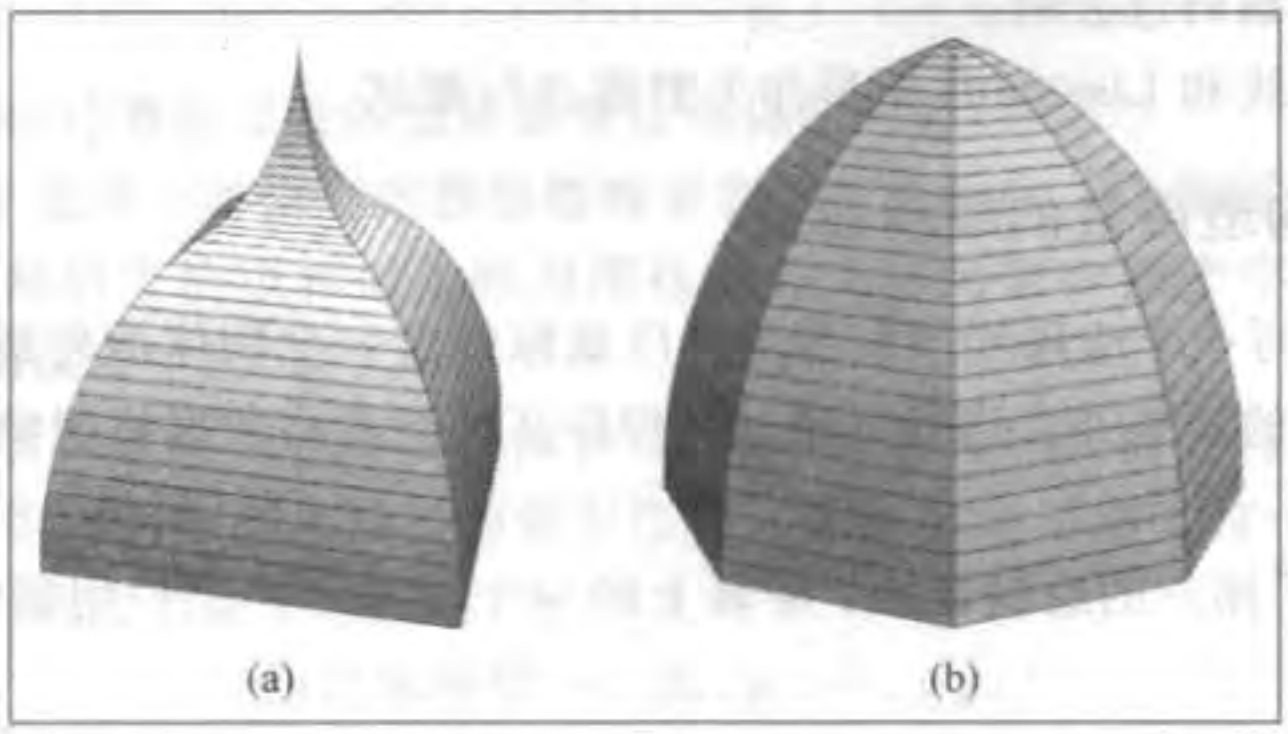


图 6.78 穹顶范例

案例 6.9 柏拉图实体

(难度：Ⅰ)(按照案例 6.1 中的格式)生成每一种柏拉图实体的网格文件。然后在应用程序中读取这些文件并绘制出相应的物体。

进一步阅读

有许多关于曲面和实体定义与生成的书。Rogers 和 Adams 著的"Mathematical Elemnets for Computer Graphics"【Rogers90】对曲线和曲面进行了清晰地介绍，同样经典但年代稍久远的有 Faux 和 Pratt 著的"Computational Geometry for Design and Manufacture"【Faux79】。Gray 著的"Modern Differential Geometry of Curves and Surfaces with Mathematics"【Gray93】对曲线和曲面进行了严格的数学处理，并且附有 Mathematica 程序代码对它们进行绘制。Mortenson 著的"Geometric Modelling"【Mortenson85】也提供了 CAD 工业中有关实体建模的出色论述。Hearn 和 Baker【Hearn03】对计算机图形学进行了概述并且阐述了当代文献中的许多主题。

第7章 三维观察

本章学习目标

- 创建和操作摄像机的开发工具，以便在各种视角生成三维场景图像
- 介绍如何让一个摄像机交互地飞过一个场景，同时获取动画
- 学习描述各种投影的数学知识
- 介绍 OpenGL 图形绘制管道中的操作是怎样执行的，同时为什么要用这些操作
- 建立一个强大的二维物体裁剪算法
- 设计一种生成立体视图的方法

预览

7.1 节概述了建立一个应用程序使摄像机飞过场景所额外需要的工具。请回忆前面我们使用短语"摄像机"来描述的像飞行器那样的假想的摄像机，正如我们经常在电视制片中使用的摄像机。7.2 节定义了一个可以生成透视视图的摄像机，同时介绍怎样使用 OpenGL 生成这样一个摄像机。此外还介绍了很多航空术语：如"倾斜、翻转和偏转"，这些术语帮助我们以一种直观的方式描述操作摄像机的方式。在本节中还建立了用矩阵方式描述摄像机旋转的数学工具。7.3 节定义了一个 Camera 类，这个类封装了摄像机的信息以及在场景中建立和调整摄像机的方法。

7.4 节详细讲述了透视投影的几何特性，同时描述了表现透视性质的数学工具。本节还介绍了怎样把透视投影加到图形绘制管道中，同时描述了 OpenGL 怎样做到这一点。为达到这些目的，介绍了齐次坐标的一些额外性质。本节同样建立了一个强大的、在齐次坐标中使用的裁剪算法，并阐明了在裁剪之前对各点进行恰当变换可以显著地提高效率。

7.5 节展示了怎样生成一个场景的立体视图以使得它们更加容易理解和更吸引人。本章以一些案例学习作为结尾，这些案例主要集中在如何创建程序以检验我们讨论的内容。

7.1 概　　述

本章主要讨论如何为精心设计的三维物体场景生成图像。和我们在第 5 章中看到的一样，OpenGL 提供了在场景中建立 jib 摄像机的工具，同时也提供了把场景投影到 jib 摄像机视平面的工具和把投影绘制到视口的工具。到现在为止，我们的 jib 摄像机只能生成平行投影。在第 6 章中我们讨论了好几类有趣的三维形状，这些可以被用来在场景中建模我们想要的物体，并且通过 Mesh 类我们可以使用基本的明暗法绘制它们。

我们还可以再做点什么呢？我们想要更深层次地了解场景背后的运行机制。这种更深层次的理解可以帮助我们很快地编写出没有 bug 的程序。更加深刻地理解等式背后的数学原理可以为程序员提供新的思路。更加现实一点，我们想要控制摄像机生成透视投影。我们希望有一种方式可以让用户对摄像机的位置和方向进行更多的控制，使得在动画中能够控制摄像机飞过整个场景。这就要求开发出 OpenGL 也未提供的控制工具，同时希望它是基于直观方式的。我们同样也需要对摄像机视景体进行精确的控制，在透视情况下尤其如此，就像平行投影那样，这取决于一个矩阵。上述要求需要对齐次坐标有一个更深层次的利用，因此我们首先要开发出透视投影的数学工具，然后看看怎么把它和 OpenGL 绘制管道结合起来。同样我们也会描述怎样针对摄像机的视景体进行剪切，这同样需要在深层次上使用齐次坐标的一些细节。在最后我们将会看到一个从头到尾运行的全过程。

7.2 再次研究摄像机

第 5 章中我们曾经使用摄像机生成平行投影。它的视景体是一个由 6 个面围起来的平行六面体，包括一个近平面和一个远平面。OpenGL 同样支持摄像机生成一个三维场景的透视视图。这和之前使用的摄像机类似，只不过使用的视景体不同。

图 7.1 展示了这种摄像机的一般形式。它有一个视点(或者叫观察参考点，VRP)指向空间中的某个点。它的视景体是塔尖在视点处的金字塔。从点 P 到视点的直线叫做 P 点的投影线(所有点的投影线都交汇于视点)。视景体的轴叫做视平面法线(VPN)。金字塔的张角由观察角度 θ(参见图 7.1(b))决定。定义 3 个垂直于 VPN 的平面：近平面、视平面、远平面，这些平面和 VPN 相交，形成四边形状的窗口。窗口有某个长宽比例，它可以在程序中设定。OpenGL 会把场景中处于视景体外的点都剪切掉。在视景体中的点 P 会投射到视平面上的一个对应点 P'，如图 7.1(c)所示(我们总是可以以投影点的方式进行思考：当投影比较复杂的物体时，它的每一个点都会被投影)。

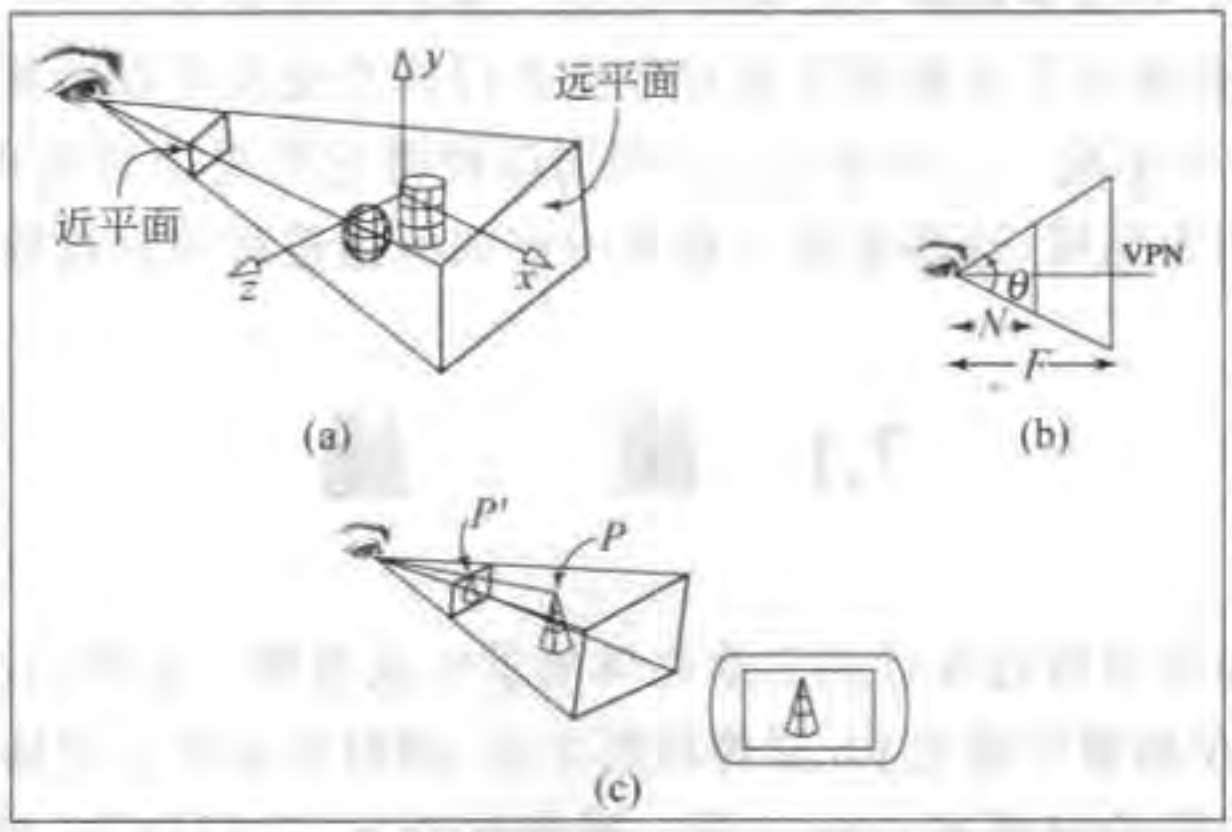


图 7.1 构造场景透视观察的摄像机

要点：点 P 投影到点 P'，点 P' 是从 P 点出发的投影线和视平面的相交点。

建议读者：熟悉并记住这些各种术语；这些将在后面章节很有用。

最后，在视平面上形成的图像会映射到视口上，如图 7.1(c)所示，这样就可以在显示器上进行显示了。

7.2.1 设置视景体

图 7.2 显示的摄像机处于默认位置，它的视点在原点，VPN 与 z 轴对齐。一些学生发现把视点考虑成"正在看"某些方向很有用，这有助于我们形成对摄像机的直观印象。尽管如此，请牢记在心：视点仅仅是一个点，并没有朝向信息。相反，程序员定义一个 look 点(如图 7.1(c)所示)，这是场景中我们感兴趣的点，同时连接视点和 look 点便定义出了 VPN 即 eye-look，它随后经过归一化成为向量 **n**。摄像机的设置是否合理，**n** 是很关键的因素(我们随后所指的向量 *VPN* 和 **n** 都是从 look 点到视点的)。

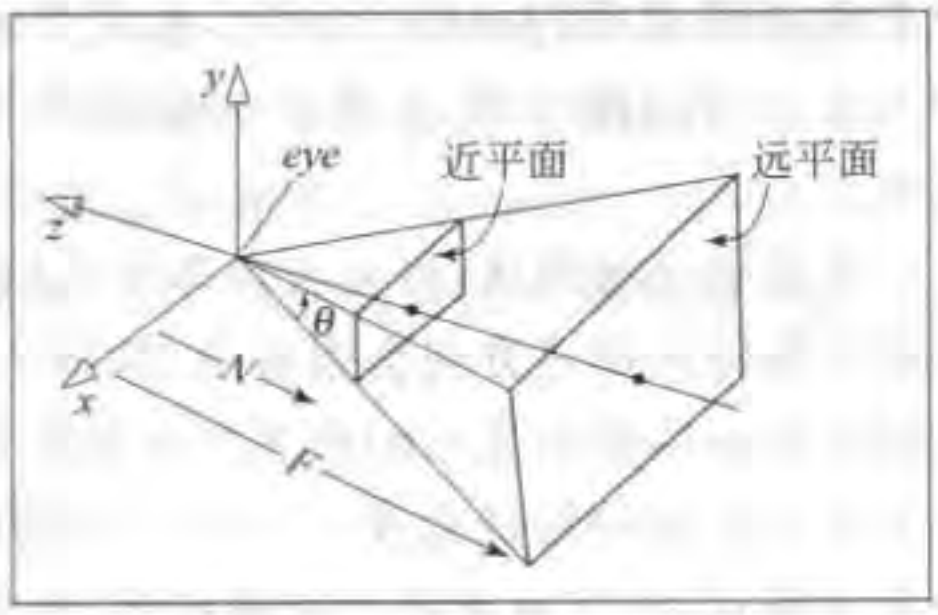


图 7.2 摄像机的默认位置

OpenGL 为在程序中设置视景体提供了简便的方法，在后面的章节会详细介绍。

7.2.2 摄像机的定位和定向

为了获得场景的某一个想要的视图，我们把摄像机从图 7.2 所示的默认位置进行移动，并让其指向特定的方向。我们通过平移和旋转来完成这个操作；这些变换成为**模型视点矩阵**的一部分，在后面章节会讨论到。

令人欣慰的是，我们设置摄像机的位置和方向的方式与我们设置平行投影的方式是完全一样的。(平行投影和透视投影的唯一区别就在于投影矩阵，它定义出了视景体的形状)。最简单的方式还是使用 gluLookAt()，如下所示：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt( eye.x, eye.y, eye.z, look.x, look.y, look.z, up.x, up.y, up.z );
```

和以前一样，这个函数移动摄像机使得它的视点位于 eye，它看向的点是 look。向上的方向由向量 **up** 决定，通常设置为(0,1,0)。在第 5 章中我们把这些参数和设置摄像机的整个过程看成自然而然的而没有进行详细介绍，本章我们会更加深入地了解它是如何完成工作的，以及如何更好地去设置摄像机。我们还要开发一些工具，使得摄像机的朝向可以进行相对变化，比如向左旋转、向上翘起或者向前滑行等。

具有任意方向和位置的摄像机

一个摄像机可以位于场景中的任意位置并且具有任意朝向，可以想象对图 7.2 中的摄像机加一个变换，使其移动到空间中的某个位置，然后旋转到某个想要的方向。我们需要一

种精确描述这种变换的方式,并且得到最后的模型视点矩阵。

如图 7.3 所示,给摄像机加上一个显式坐标系对我们很有帮助。

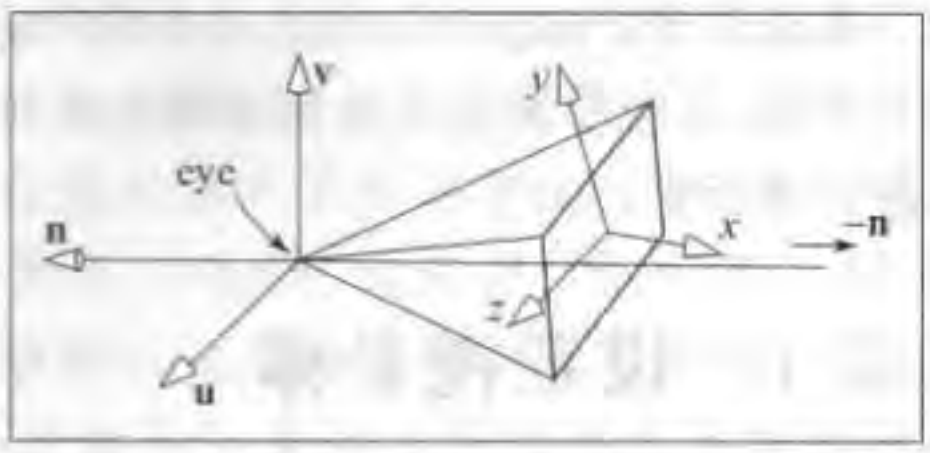


图 7.3 绑定摄像机到某个坐标系中

这个坐标系的原点在摄像机的视点处,有 3 根坐标轴,叫做 **u** 轴、**v** 轴和 **n** 轴,用来定义摄像机的朝向。各个轴的指向由向量 **u**、**v** 和 **n** 给定,如图 7.3 所示,注意这是一个右手坐标系统。摄像机的默认朝向是沿着 z 轴的负方向,因此一般来说摄像机朝向 n 轴负方向,也就是说方向是 $-\mathbf{n}$。方向 **u** 指向摄像机右侧,方向 **v** 指向上方。可以把 u 轴、v 轴和 n 轴看作是图 7.2 中的 x 轴、y 轴和 z 轴经旋转平移后的版本。

位置很容易描述,可是方向却很难给出。如果我们借用飞行术语中的倾斜度、航向、偏离度和翻转来描述方向会很有用,如图 7.4 所示。一个飞机的倾斜度就是飞机的纵向轴(从机尾到机头的朝向是 $-\mathbf{n}$)和水平面的夹角。飞机的翻转是指关于纵向轴的旋转;它的翻转是关于水平面的旋转角度。一个飞机的航向是指它的头所指的方向(也叫做方位角)。如图 7.5 所示,给定 **n** 求航向和倾斜度,很简单只需要把 $-\mathbf{n}$ 表示成球面坐标即可(参考附录 2 中球面坐标系的回顾)。向量 $-\mathbf{n}$ 具有经度和纬度,分别是 θ 和 ϕ。飞机的朝向由 $-\mathbf{n}$ 的经度给定,倾斜度由 $-\mathbf{n}$ 的纬度给定。翻转、倾斜度和朝向的关于 **u** 和 **n** 的表达式请参见练习。

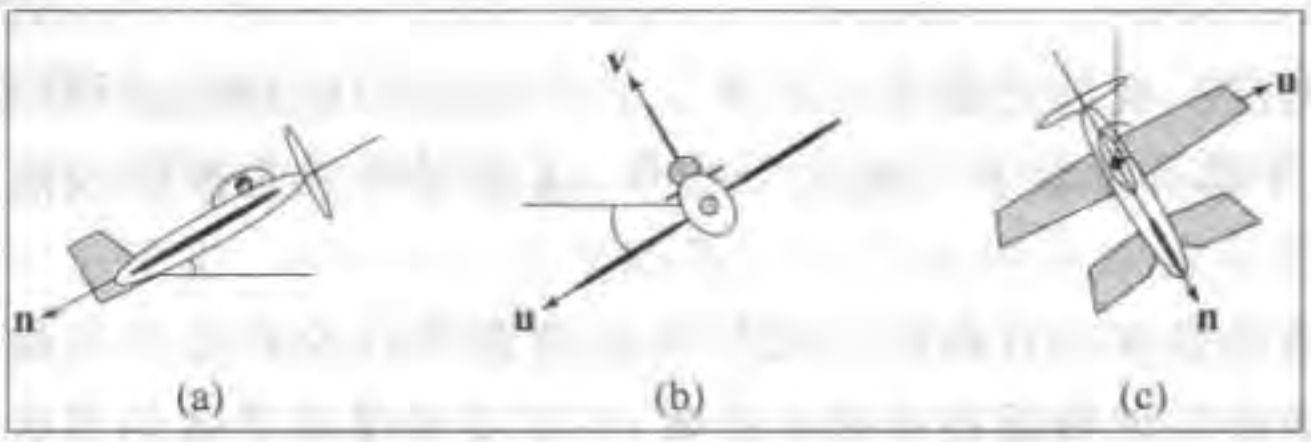


图 7.4 一架飞机相对于世界坐标系的朝向图示

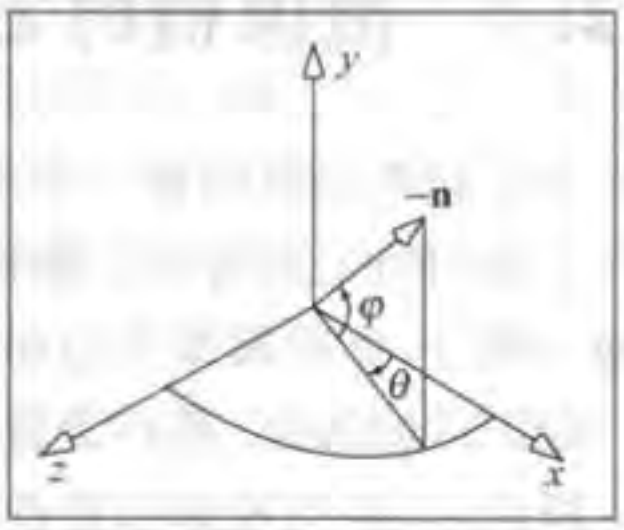


图 7.5 飞机的朝向和倾斜度

倾斜(度)和翻转(度)既可以当名词又可以当动词:当用作动词时,描述的是飞机朝向的变化。当一个飞机的倾斜度增加时,你可以说它在仰起(关于 u 轴旋转);当它关于 n 轴旋转时,你可以说它在翻转。关于改变航向的航空术语一般是偏离:向左或向右的偏离实际上都是关于 v 轴的旋转。

这些术语同样也可以用到摄像机上面。图 7.6(a)是一个我们很熟悉的摄像机,其上附有一个坐标系:有 u、v 和 n 轴,这个坐标系的原点也在摄像机的视点处。图 7.6(b)中的摄像机进行了一次非零翻转,图 7.6(c)中的摄像机进行的则是零翻转。我们通常设置的是零翻转摄像机并且称之为无翻转摄像机。无翻转摄像机的 u 轴是水平的——也就是说和世界坐标的 y 轴垂直。注意,无翻转摄像机仍然可能拥有一个任意的 **n** 方向,所以它可以具有任意的倾斜度和航向。

我们怎么控制摄像机的翻转、倾斜和航向呢?用 gluLookAt()很容易设置一个初始摄

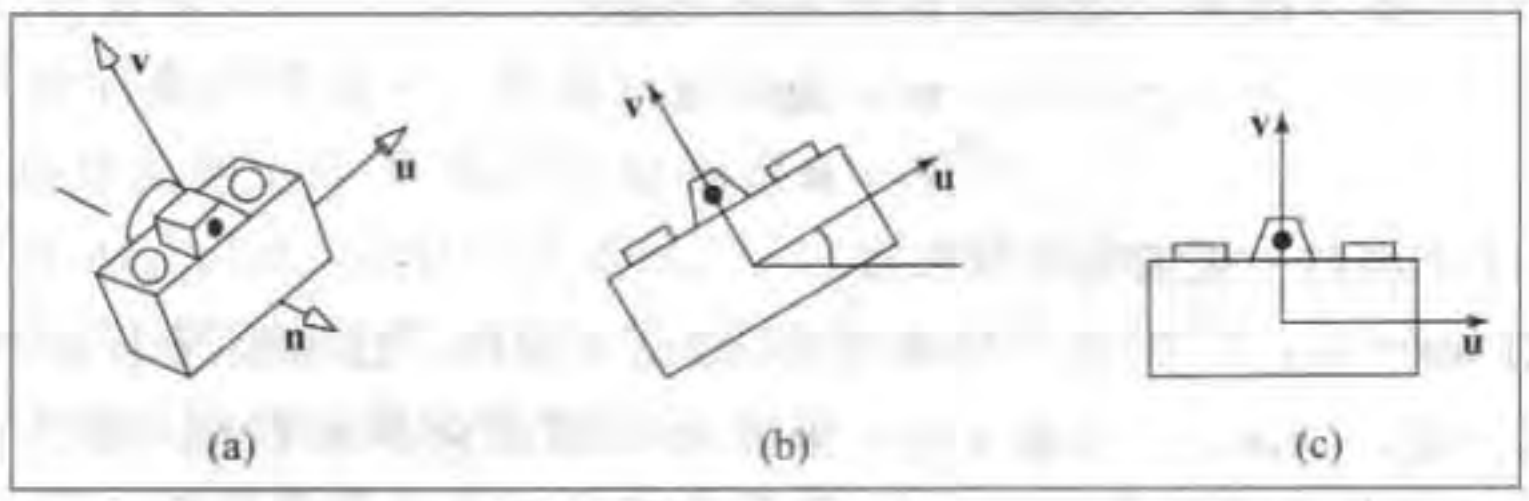


图 7.6 不同的摄像机姿态

像机，这是由于设置视点和 look 点对于我们来讲不太困难。但是怎样选择 **up** 去得到一个翻转是很困难的，同时在后面仅仅使用 gluLookAt()来调整摄像机也是很困难的(摄像机在笛卡儿坐标系下面工作，然而朝向取决于相对于轴的角度和旋转)。OpenGL 没有提供工具直接操作 u、v、n 轴，因此在程序中我们要自己去设置它们，从而使描述和调整摄像机容易很多。

gluLookAt()能干什么：一些数学基础理论

当我们把参数 *eye*、*look* 和 **up** 传给 gluLookAt()运行的时候，得出的 **u**、**v** 和 **n** 方向都是什么呢？让我们详细看看 gluLookAt()做了什么，以及为什么这么做。

如图 7.7(a)所示，我们给定了 *eye* 和 *look* 的位置和 **up** 方向。我们知道 **n** 平行于向量 eye－look，如图 7.7(b)所示，因此 **n**＝*eye*－*look*(后面如有需要，我们会归一化此向量和其他向量)。

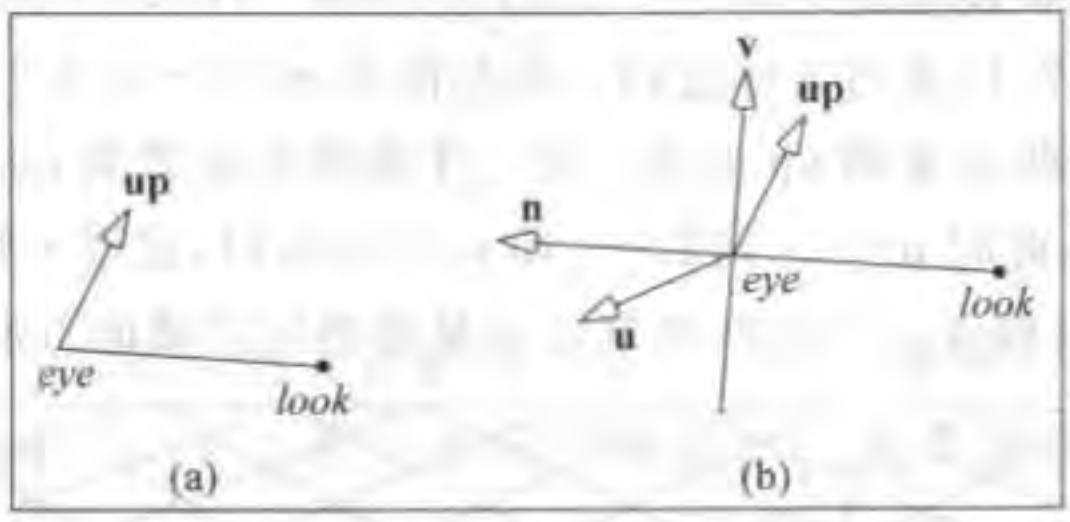


图 7.7 构建向量 **u**、**v** 和 **n**

现在我们需要得到 **u** 和 **v**，它们都垂直于 **n** 并且互相垂直。**u** 向量指向摄像机的侧边，因此自然(合理的)使它垂直于 **up**，**up** 就是用户设置的向上的方向。这是 gluLookAt()在任何情况下的假设，因此 **u** 垂直于 **n** 和 **up**。一个计算出垂直于两个向量的向量的好方法是求这两个向量的叉积，因此我们有 **u**＝**up**×**n**(用户不能为 **up** 选择一个平行于 **n** 的向量，如果这样 **u** 的长度将会为 0，为什么?)。我们之所以选择 **u**＝**up**×**n** 而不是 **n**×**up**，是因为 **u** 应该指向－**n**。

当 **u** 和 **n** 求出来以后，最后就是计算 **v** 了；它必须同时垂直于 **u** 和 **v**，因此使用一个乘法：**v**＝**n**×**u**。注意 **v** 通常不会与 **up** 重合：**v** 一定是指向垂直于 **n** 的方向，然而 **up** 只是用户给出的一个向上的示意性向量。

总结：给定 *eye*、*look* 和 **u**，我们得到：

$$\begin{aligned}\mathbf{n} &= eye - look \\ \mathbf{u} &= \mathbf{up} \times \mathbf{n} \\ \mathbf{v} &= \mathbf{n} \times \mathbf{u}\end{aligned} \tag{7.1}$$

然后把这 3 个都归一化成为单位长度。

请注意,对 **up**=(0, 1, 0)这个特殊情形,结论又如何。这种情况下 $\mathbf{u}=(n_z,0,-n_x)$, $\mathbf{v}=(-n_xn_y,n_x^2+n_z^2,-n_zn_y)$。注意 **u** 的 y 坐标为 0,因此它是水平的。更进一步,**v** 的 y 分量是正值,因此它正好指向上方。

例 7.2.1 找出摄像机的坐标系

考虑一个摄像机,其 $eye=(4,4,4)$,$look=(0,1,0)$。更进一步设向量 **up** 为(0,1,0)。求出 **u**、**v** 和 **n**。然后当 **up** 为(2, 1, 0)时再求一次。

解:从等式(7.1)我们知道:**u**=(4,0,−4),**v**=(−12,32,−12),**n**=(4,3,4),这几个向量都很容易归一化成为单位长度(画出这个结论)。注意,**u** 确实是水平的。检验它们是不是都互相垂直。在 **up**=(2,1,0)的情况下(在结果计算出来之前尽量在头脑中想象一下这个摄像机。还有,我们计算出的结果也没有归一化),我们有 **u**=(4,−8,2),**v**=(38,8,−44),**n**=(4,3,4)。画出这个结果。检查这些结果是否相互垂直。

例 7.2.2 建立摄像机的直观印象

在建立摄像机时,为了有助于提高直观的几何直觉,图 7.8 展示了在世界坐标系上面的两个摄像机例子——每一个都具有视景体和坐标系——为了清晰表现,我们在 xz-平面上画出网格。其中一个摄像机设置为 $eye=(-2,2,0)$,$look=(0,0,0)$和 **up**=(0,1,0)。对于这个摄像机,我们从等式(7.1)求出未经过归一化的向量 **n**=(−2,2,0),**u**=(0,0,2)和 **v**=(4,4,0)。此图显示出了这些向量和 **up** 向量。第二个摄像机设置为 $eye=(2, 2, 0)$,$look=(0, 0, 0)$和 **up**=(0, 0, 1),此时 **u**=(−2,2,−)和 **v**=(0,0,8),这里 **v** 和 **up** 平行。注意这个摄像机看起来是侧着的:(检查一下这些向量是不是都指向正确的方向)。

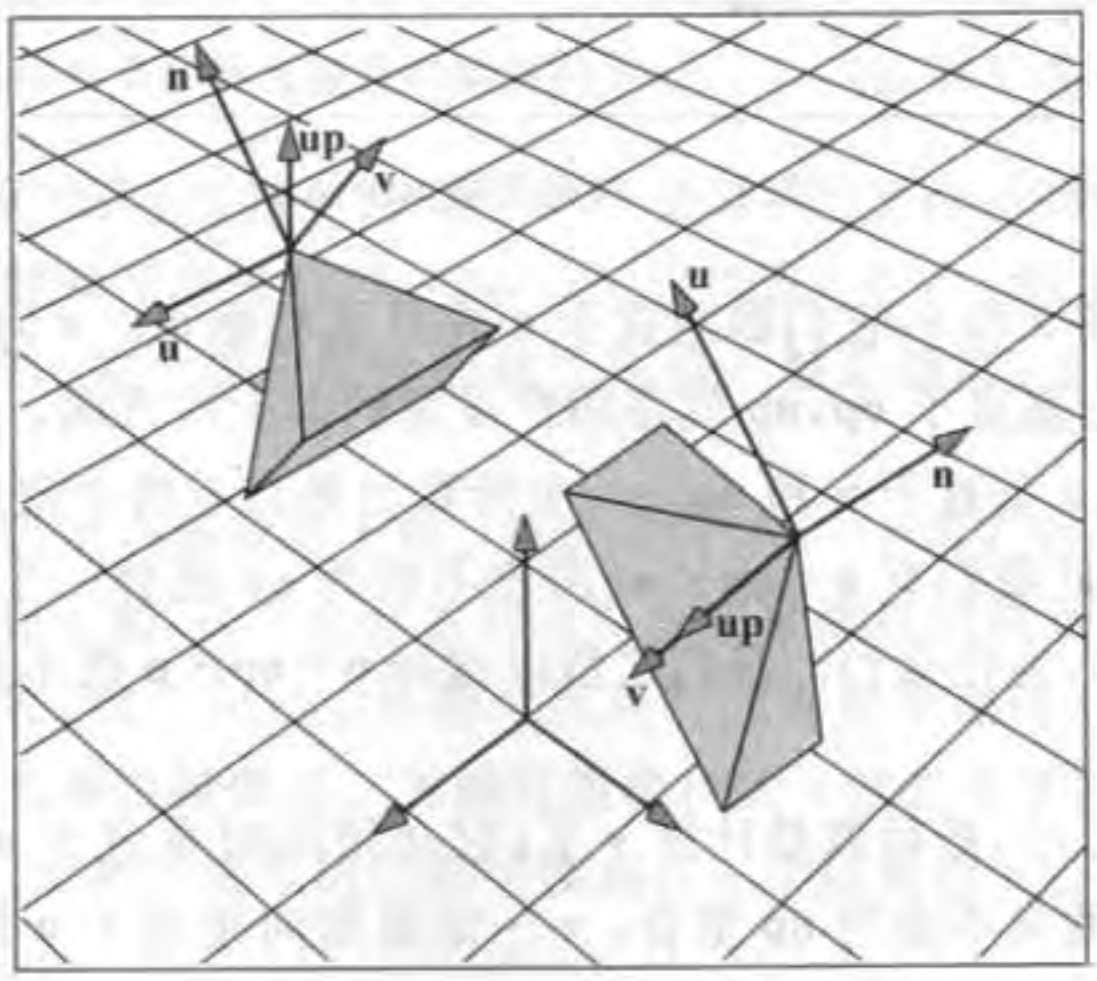


图 7.8 两个摄像机设置的图示

最后，我们想看看 gluLookAt()在模型视点矩阵中的值。从第 5 章我们知道模型视点矩阵由两个矩阵的乘积得到——处理从世界坐标到摄像机坐标转换的矩阵 V 和对模型所有点进行变换的矩阵 M。gluLookAt()由等式(7.1)建立矩阵 V，同时后乘当前矩阵。因为 V 矩阵的作用是把世界坐标系转换成摄像机坐标系，所以如图 7.9 所示，它必须把摄像机坐标系统变换到一般的位置。也就是说它必须把 *eye* 变换到原点，**u** 变换到向量 **i**，**v** 到 **j**，**n** 到 **k**。有好几种方式可以得到 V 的值，但是最精确的方法是使用下面的矩阵：

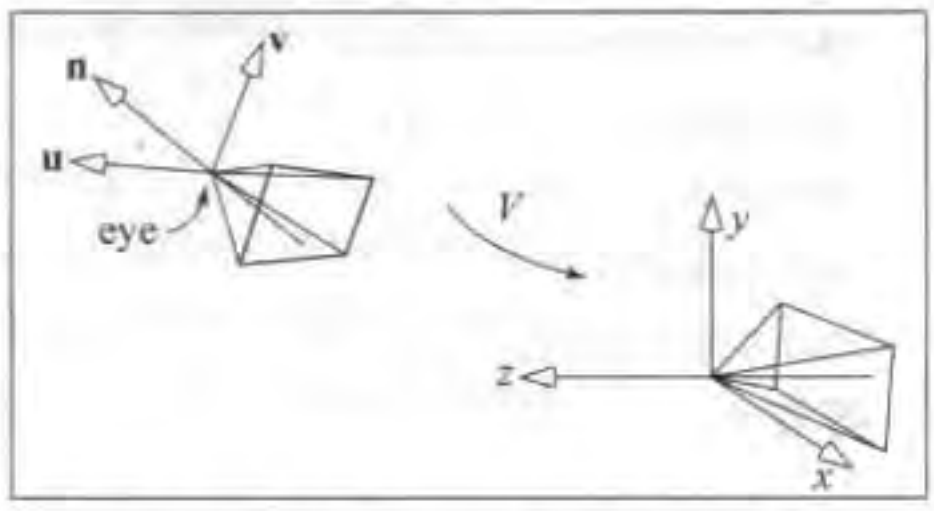


图 7.9 gluLookAt()函数设置的变换

$$V=\begin{bmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.2}$$

其中$(d_x,d_y,d_z)=(-eye\cdot\mathbf{u},-eye\cdot\mathbf{v},-eye\cdot\mathbf{n})$[①]。事实上我们想要的结果是

$$V\begin{bmatrix} eye_x \\ eye_y \\ eye_z \\ 1 \end{bmatrix}=\begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

其中我们把 *eye* 扩展到齐次坐标。同样检查

$$V\begin{bmatrix} u_x \\ u_y \\ u_z \\ 0 \end{bmatrix}=\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

同时 V 把 **v** 映射到$(0,1,0,0)^T$，把 **n** 映射到$(0,0,1,0)^T$。矩阵 V 是由 gluLookAt()创建的，并且后乘到当前矩阵。后面当我们在程序中维护这个摄像机时有机会进行这样的操作。

7.3 在程序中指定一个摄像机

为了更好地控制摄像机的移动，我们要在程序中创建和维护自己的摄像机。当每一个变化作用于这个摄像机时，摄像机会告诉 OpenGL 新的摄像机姿态是什么。我们创建 Camera 类，这个类知道摄像机应该完成什么工作。因为要创建一个摄像机需要很多的变量和向量，我们需要一个有组织的方式去创建和维护摄像机的各组成部分。答案很明显，就像前述的那样创建一个 Camera 类，这样做很简单但是代价很大。在程序中我们创建一个

① 技术点：一个点不能和一个向量进行点积，因此此处 eye 应换为向量(eye−(0,0,0))

Camera 的对象,叫做 cam,然后调用它的函数:

```
cam.set(eye,look,up);
cam.slide(-1,0,-2);
cam.roll(30);
cam.yaw(20);
cam.pitch(15);
等等
```

图 7.10 展示了 Camera 类定义的一部分。Camera 类包括域 eye、look、up 和其他相关的点,它们都是 Point3 类型,向量 **u**、**v** 和 **n** 都是 Vector3 类型。Point3 和 Vector3 之前已经定义过了,参见附录 3。

```
class Camera{
  private:
    Point3 eye, look, up;
    Vector3 u, v, n;
    double viewAngle, aspect, nearDist, farDist;
    void setModelViewMatrix();
  public:
    Camera();
    void set(Piont3 eye, Point3 look, Vector3 up);
    void roll(float angle);
    void pitch(float angle);
    void yaw(float angle);
    void slide(float du, float dv, float dn);
    void setShape(float viewAngle, float aspect, float Near, float Far);
    void getShape(float& viewAngle, float& aspect, float& Near, float& Far);
};
```

图 7.10 Camera 的类定义

同样还有数据域用来描述视景体的形状:viewangle、aspect、nearDist 和 farDist。

函数 setModelviewMatrix()用来把模型视点矩阵传递给 OpenGL。它只能被这个类的成员函数调用。任何对摄像机位置和方向的改变都要更新模型视点矩阵。图 7.11 展示了一个可能的实现版本。它根据 *eye*、*look*、**u**、**v** 和 **n** 当前的值计算等式(7.2)所示的矩阵,然后使用 glLoadMatrixf()把这个矩阵直接加载到模型视点矩阵中。

方法 *set*()和 *gluLookAt*()的作用相似:它把 eye、look 和 **up** 的值代入等式(7.1)计算 **u**、**v** 和 **n**。它会把这些信息通知给摄像机对象,同时把这些值传递给 OpenGL。

函数 setShape()甚至更简单:它把四个参数值放进摄像机类的适当的域,然后调用函数 gluPerspective(viewAngle, aspect, nearDis, farDist)(同时还有 glMatrixMode(GL_PROJECTION)和 glLoadIdentity())去设置投影矩阵。

摄像机的核心方法是 slide()、roll()、yaw()和 pitch(),它们都会对摄像机的位置和方向做相关调整。函数slide()移动摄像机但是不改变它的方向(在Camera数据结构里面保

```
void Camera::setModelViewMatrix(void)
{//load modelview matrix with existing camera values
    float m[16];
    Vector3 eVec(eye.x, eye.y, eye.z);
    m[0]=u.x; m[4]=u.y; m[8]=u.z;  m[12]=- eVec.dot(u);
    m[1]=v.x; m[5]=v.y; m[9]=v.z;  m[13]=- eVec.dot(v);
    m[2]=n.x; m[6]=n.y; m[10]=n.z; m[14]=- eVec.dot(n);
    m[3]=0;   m[7]=0;   m[11]=0;   m[15]=1.0;
    glMatrixMode(GL_MODELVIEW);
    glLoadMatrixf(m);
}
void Camera :: set(Point3 Eye, Point3 look, Vector3 up)
{//make u, v, n vectors
    eye.set(Eye);
    n.set(eye.x-look.x, eye.y-look.y,eye.z-look.z);
    u.set(up.cross(n));
    v.set(n.cross(u));
    u.normalize(); v.normalize(); n.normalize();
    setModelViewMatrix();
}
```

图 7.11　函数 set()和 setModelviewMatrix()

存 *eye*、**u**、**v**、**n** 的原因是我们可以保存一份当前摄像机的记录以便对它进行交互式的修改)。下面我们就来考察摄像机的成员函数是怎么运作的。

7.3.1　交互地移动摄像机

用户通过按键或者单击鼠标控制摄像机飞过整个场景。例如,按下'u'键可以控制摄像机向上移动一段距离,按下'y'键可以使摄像机向左偏,按下'f'键可以使摄像机向前移动。用户通过一个视点观察场景,还可以改变摄像机以便得到一个更好的视点和朝向来生成图像。或者用户可以让摄像机绕着场景飞行得到快速的连续画面。如果这些截图被保存下来然后快速回放,那么就产生了一个绕场景飞行的动画。

我们有 6 个自由度去调整摄像机:它可以在三维上变动其位置,同时还可以绕 3 个坐标轴中的任意一个进行旋转。首先我们来建立 slide()函数来移动摄像机。

移动摄像机

移动摄像机就是指沿着它的其中一个轴对摄像机进行移动——也就是说在 **u**、**v** 和 **n** 方向上,但是没有涉及旋转操作。因为摄像机视点的朝向是 **n** 轴负方向,那么沿着 **n** 方向就是向前或者向后移动。类似地,沿着 **u** 轴是向左或者向右移动,沿着 **v** 轴是向上或者向下移动。

控制摄像机沿着一个轴运动很简单。如果要沿着 **u** 轴把摄像机移动距离 D,那么就设

置 *eye* 为 *eye*+*D***u**,同时以相同的方式改变 *look*。不用改变 **up**。为了方便起见,我们可以把 3 个可能的移动组合到一个函数里面。Slide(delU, delV, delN)使摄像机在 *u* 轴移动 delU,在 **v** 轴移动 delV,在 **n** 轴移动 delN:

```
void Camera slide( float delU, float delV, float delN )
{
    eye.x+=dulU*u.x+delV*v.x+delN*n.x;
    eye.y+=dulU*u.y+delV*v.y+delN*n.y;
    eye.z+=dulU*u.z+delV*v.z+delN*n.z;
    look+=dulU*u.x+delV*v.x+delN*n.x;
    look.y+=dulU*u.y+delV*v.y+delN*n.y;
    look.z+=dulU*u.z+delV*v.z+delN*n.z;
    setModelViewMatrix();
}
```

旋转摄像机

我们想要翻转、倾斜或者偏移摄像机,这些都要求摄像机绕着自己的一个轴进行旋转。接下来我们将详细讨论翻转的情况;另外两种旋转的情况类似。

翻转摄像机实际上就是绕着它自己的 **n** 轴旋转。也就是说 **u** 和 **v** 方向必须旋转,如图 7.12 所示。当 **u** 和 **v** 已经旋转过 α 度后,在示例平面上会形成两个新轴 $\mathbf{u}'$ 和 $\mathbf{v}'$。

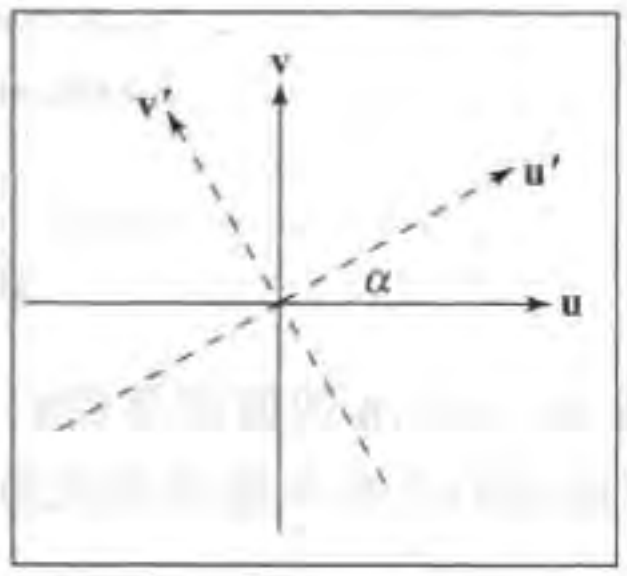


图 7.12 摄像机翻转

因此我们仅仅需要用 **u** 和 **v** 的线性组合表示 $\mathbf{u}'$,$\mathbf{v}'$ 的表示法类似:

$$\mathbf{u}' = \cos(\alpha)\mathbf{u} + \sin(\alpha)\mathbf{v}$$
$$\mathbf{v}' = -\sin(\alpha)\mathbf{u} + \cos(\alpha)\mathbf{v} \qquad (7.3)$$

然后把新轴 $\mathbf{u}'$ 和 $\mathbf{v}'$ 加入摄像机,这个实现起来很直接。

```
void Cmaera::roll( float angle )
{
    float cs=cos(3.14159265/180*angle);
    float sn=sin(3.14159265/180*angle);
    Vector3 t(u);
    u.set(cs*t.x-sn*v.x, cs*t.y-sn*v.y, cs*t.z-sn*v.z);
    v.set(sn*t.x+cs*v.x, sn*t.y+cs*v.y, sn*t.z+cs*v.z);
    setModelViewMatrix();
}
```

函数 pitch()和 yaw()实现方式类似。参见练习。

集成

我们在图 7.13 中展示了如何在 OpenGL 中使用 Camera 类来控制摄像机飞过整个场景。场景中只包含一个茶壶。摄像机是一个全局对象,在 main()函数中设置好它的起始视点和形状。当一个键被按下时,myKeyboard()函数被调用,摄像机根据你按下的键进行平

移或者旋转。例如：如果你按下了'P'键，那么摄像机就会仰起 1 度。如果是 CTRL F 键[①]（按着 control 键然后按'f'键）被按下了，那么摄像机向下倾斜 1 度。在按键被处理完以后，glutPostRedisplay()被函数 myDisplay()调用绘制新的图像。

注意调用函数 glutSwapBuffer()，同时注意调用函数 glutInitDisplayMode()[②]时的参数 GLUT_DOUBLE。应用程序使用**双缓冲**（见第 3 章）来达到当前帧和下一帧之间快速平滑的转换。有两个内存缓冲区用来存储生成的图像。在函数 glutSwapBuffer()的控制下显示不断从一个缓冲区转换到另一个缓冲区。每一个新图像在不可见的缓冲区中进行绘制，当绘制结束以后，显示才切换到这个图像。因此，用户不会看到显示器的清屏操作和新图像被一行一行显示出来的这种烦人效果。相反，在后台合成新图像时，旧图像还是很稳定地进行显示，然后显示很快切换到新完成的图片上面。

```
#include "windows.h"
#include "GL/gl.h"
#include "GL/glu.h"
#include "GL/glut.h"
#include "camera.h"


Camera cam;                                  //make a global camera object

//<<<<<<<<<<<<<<<<<<<<<<<<<myKeyboard>>>>>>>>>>>>>>>>>>>>>>>>
void myKeyboard(unsigned char key, int x, int y)
{
  switch(key)
  {

    //controls for camera
   case 'L':     cam.slide(.2, 0, 0); break;          //slide camera right
   case 'L'-64: cam.slide(-0.2, 0, 0); break;         //slide camera left

   case 'U':     cam.slide(0, .2, 0); break;          //slide camera up
   case 'U'-64: cam.slide(0,-0.2, 0); break;          //slide camera down
   case 'F':     cam.slide(0,0, 0.2); break;          //slide camera forward
   case 'F'-64: cam.slide(0,0,-0.2); break;           //slide camera back
   //add up/down and left/right controls
   case 'P':     cam.pitch(-1.0); break;
   case 'P'-64: cam.pitch( 1.0); break;
   // add yaw controls
   case 'Y':     cam.yaw(-1.0); break;
   case 'Y'-64: cam.yaw( 1.0); break;
```

图 7.13 使用 Camera 类围绕茶壶进行视点漫游的完整程序代码

① 在大多数键盘中 CTRL 和字母键组合会返回一个小于 64 的 ASCII 码。

② glutInitDisplayMode()函数必须使用 GLUT_DOUBLE 参数来开启双缓冲。

```
        //add roll controls
        case 'R':     cam.roll(1.0); break;
        case 'R'-64: cam.roll(-1.0); break;
    }
    glutPostRedisplay();                                //draws it again
}

//<<<<<<<<<<<<<<<<<<<<<<<<myDisplay>>>>>>>>>>>>>>>>>>>>>>>>>>
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutWireTeapot(1.0);                                //draw the teapot
    glFlush();
    glutSwapBuffers();        //display the screen just made
}

//<<<<<<<<<<<<<<<<<<<<<<<<main>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char**argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);    //double buffering
    glutInitWindowSize(640,480);
    glutInitWindowPosition(50, 50);
    glutCreateWindow("fly a camera around a teapot");
    glutKeyboardFunc(myKeyboard);
    glutDisplayFunc(myDisplay);
    glClearColor(1.0f,0.0f,0.0f,0.0f);                  //background is red
    glColor3f(0.0f,0.0f,0.0f);                          //set color of stuff
    glViewport(0, 0, 640, 480);
    cam.set(4, 4, 4, 0, 0, 0, 0, 1, 0);                 //make the initial camera
    cam.setShape(30.0f, 64.0f/48.0f, 0.5f, 50.0f);
    glutMainLoop();
}
```

图 7.13(续)

使用摄像机绘制 SDL 场景

如第 5 章描述的一样,把一个摄像机加到能够读取 SDL 文件的程序中很容易。有两个全局对象:

```
Camers cam;
Scene scn;
```

同时在 main()中 scn. read("myScene. dat")读取和处理一个 SDL 文件。最后,在

myDisplay(void)中，用 scn. drawSceneOpenGL()替换掉原有的绘制函数。

练习

练习 7.3.1　实现 pitch()和 yaw()

写出函数 void Camera::pitch(float angle)和 void Camera::yaw(float angle)，这两个函数实现了摄像机的倾斜和偏移。偏移为正，摄像机向左偏；倾斜为正，摄像机向上翘。

练习 7.3.2　编写一个通用的 rotate()函数

写一个函数 void Camera::rotate(Vector3 axis, float angle)使得摄像机绕轴 axis 旋转 angle 角度。这个函数使 **u**、**v** 和 **n** 三个轴围绕 *eye* 进行旋转。

7.4　三维物体的透视投影

我们已经有了 Camera 类，因此我们可以绕着三维场景飞行，并且很容易地生成图像。使用 OpenGL，每张图像都是通过把物体的所有顶点都传给图形绘制管道生成出来的，如第 5 章所描述的那样。图 7.14 又一次展示了图形绘制管道，但是出现了一个新元素。

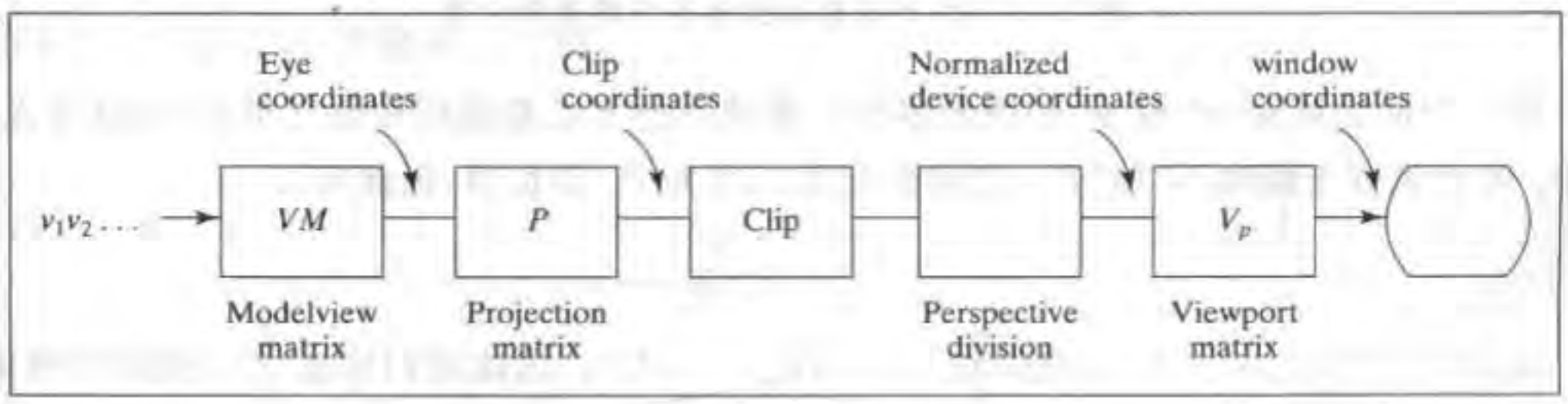


图 7.14　图形绘制管道

回忆一下每一个顶点 v 都是右乘模型视点矩阵(VM)。建模部分(M)包括了物体的所有模型变换；视点部分(V)说明了摄像机位置和朝向的变换。当一个顶点被此矩阵作用后，它就在处于**视点坐标系**中——也就是说在视点的坐标系统中。图 7.15 展示了一个这样的坐标系统，其中视点位于原点，近平面在 $z=-N$ 处垂直于 z 轴。位于 P 处的顶点的视点坐标被传递给绘制管道的下一个阶段，在此阶段点被投影到近平面上一点(x^*, y^*)，然后执行裁剪操作，最后未被裁剪掉的点映射到显示器的视口中。

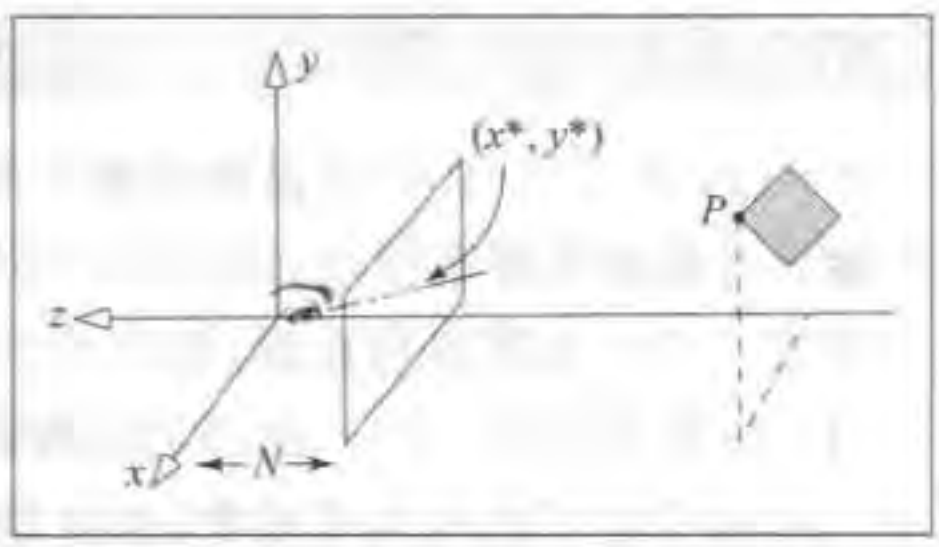


图 7.15　在视点坐标系中描述的一个顶点的透视投影

此时我们需要更深入地研究形成透视投影的过程，并且需要回答一些问题。哪些操作构成一个透视投影，绘制管道又是如何使用这些操作的？矩阵和透视投影的关系是什么？投影是怎么把视景体变换到一个用于裁剪的规范视景体？裁剪是怎么实现的？齐次坐标起到什么作用？为了消除隐藏面，要怎样存储视点到一个顶点的深度信息？透视除法的步骤又是

什么呢?

我们首先讲述与特定图形绘制管道无关的透视投影的一些特性。然后看看图形绘制管道是怎样精心安排步骤来生成透视投影所需要的各种数值的。

7.4.1 点的透视投影

透视投影的基本操作是把一个三维点投影到一个二维平面上。图 7.16 是图 7.15 的进一步说明,在图中点 $P=(P_x,P_y,P_z)$被投影到近平面上一点(x^*,y^*)。我们在近平面上建立一个坐标系统,它的原点位于摄像机的 z 轴。这样的话讨论距离原点向右 x^* 个单位,向上 y^* 个单位的 P 点投影坐标才有意义。

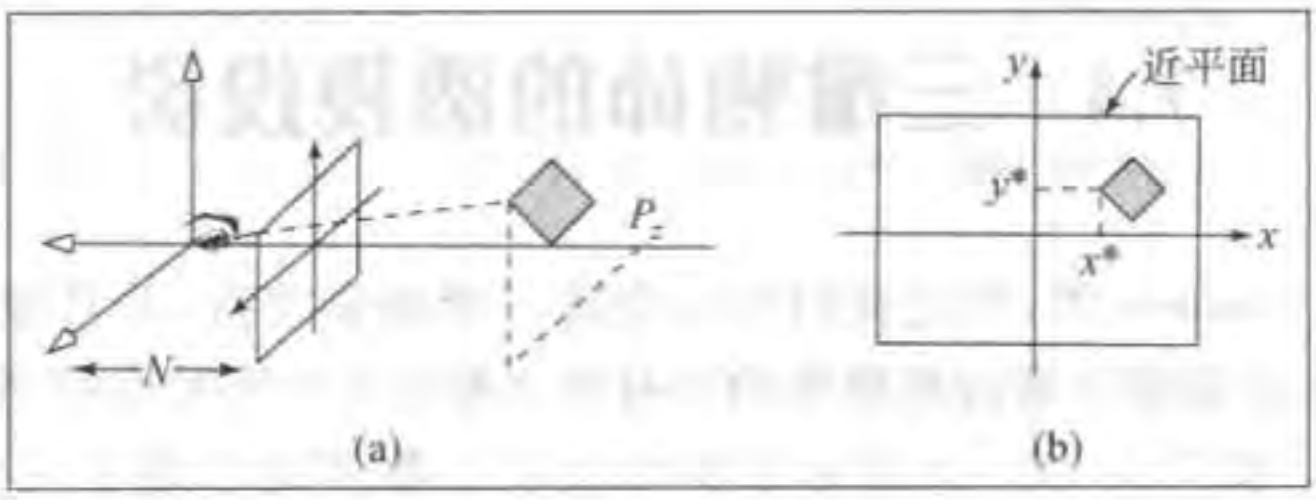


图 7.16 点 P 在视点坐标系中的投影位置

第一个问题就是 x^* 和 y^* 的值是多少?最简单的方法是使用相似三角形,也就是说 x^* 和 P_x 的比值等于距离 N 和$|P_z|$之间的比值。因为 P_z 是负值,因此有:

$$\frac{x^*}{P_x}=\frac{N}{-P_z}$$

或者 $x^*=NP_x/(-P_z)$,类似地 $y^*=NP_y/(-P_z)$。因此我们知道 P 点投影到视平面上一点:

$$(x^*,y^*)=\left(N\frac{P_x}{-P_z},N\frac{P_y}{-P_z}\right)\quad(P\text{ 的投影})\tag{7.4}$$

练习中给出了得到这一结果的另一个例子。

例 7.4.1

$P=(1,0.5,-1.5)$会投影到视平面上哪一点?近平面在 $N=1$ 处。

解:直接使用等式(7.4)得到$(x^*,y^*)=(0.666,0.333)$。

关于一个点如何进行投影,我们可以做一些预先观察:

(1) 注意到分母$-P_z$。越远的点此值越大(沿着 z 轴负方向),所以使得 x^* 和 y^* 的值减小。这就引起了透视缩短现象,也就是物体上距离远的部分看起来比较小。

(2) 分母可能为 0,当 P 点和眼睛处于同一平面时 P_z 就会为 0。一般来说我们会使用裁剪去掉这些后面会引起麻烦的点。

(3) 如果 P 点处于眼睛后面会有一个正的 P_z,这将会引起问题,后面我们会看到。这些点也同样要被裁剪掉。

(4) 近平面的距离 N 的影响很简单就是缩放图像(x^* 和 y^* 都与 N 成比例)。如果我

们选择其他平行于近平面的面作为视平面，那么投影到这个平面的图像仅仅只是大小与投影到近平面的图像不同。因为最后我们还是要把投影映射到一个固定大小的显示器视口上，因此并不会影响到投影图像的大小。这就是说任何平行于近平面的视平面都可以，因此我们也可以使用近平面作为视平面。

(5) 直线投影为直线。图 7.17 提供了最简单的证明。考虑三维空间中点 A 和点 B 之间的直线。A 投影到 A'，B 投影到 B'。但是 A 和 B 之间的点是不是直接投影到 A' 和 B' 之间的点呢？是的：只需考虑 A、B 和原点形成的平面。因为任何两个平面相交都是一条直线，因此这个平面和近平面的相交部分就是直线，所以直线段 AB 投影到直线段 $A'B'$。

请留意我们这样做的目的是为 P 矩阵计算出合适的值，使得 OpenGL 可以在需要的时候使用它们，这个过程会发生在图形绘制管道中的几个不同时刻。如图 7.17 所示，它首先要显示出线段 AB 上面的哪个点需要被投影。每一个投影点被转换成一个算术表达式，然后被加载到矩阵 P 中。特别重要的是计算投影点需要一个除法操作，也就是透视除法，齐次坐标表示在这里展现了它的强大力量，后面详细讲述其步骤。

例 7.4.2　仓房的 3 个投影

通过不同的摄像机观察一个简单的物体可以获得大量直观信息。这里我们将查看第 6 章曾经介绍过的并且在图 7.18 中重复出现的仓房模型的各条边是怎样投影到 3 个摄像机的。仓房有 10 个顶点 15 个边和 7 个面。

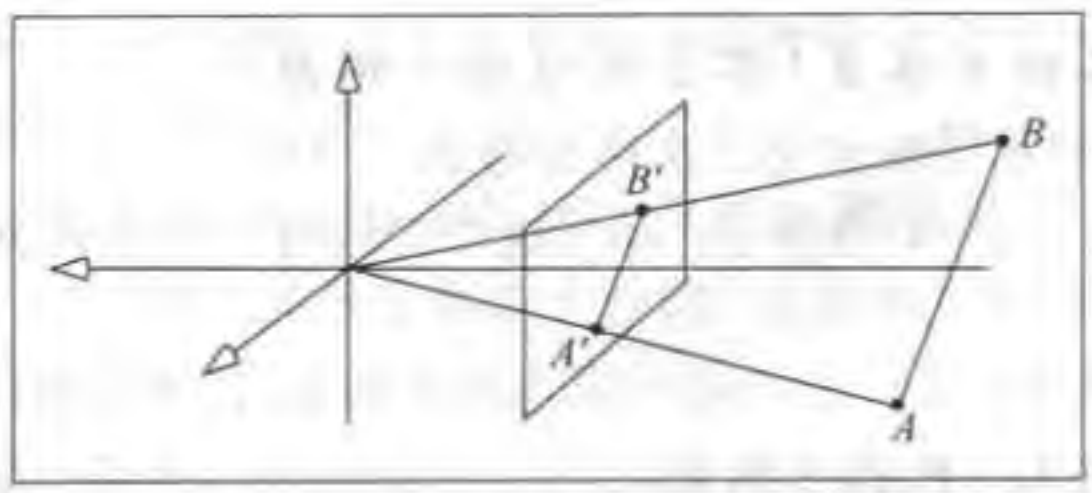


图 7.17　直线投影为直线的证明图示

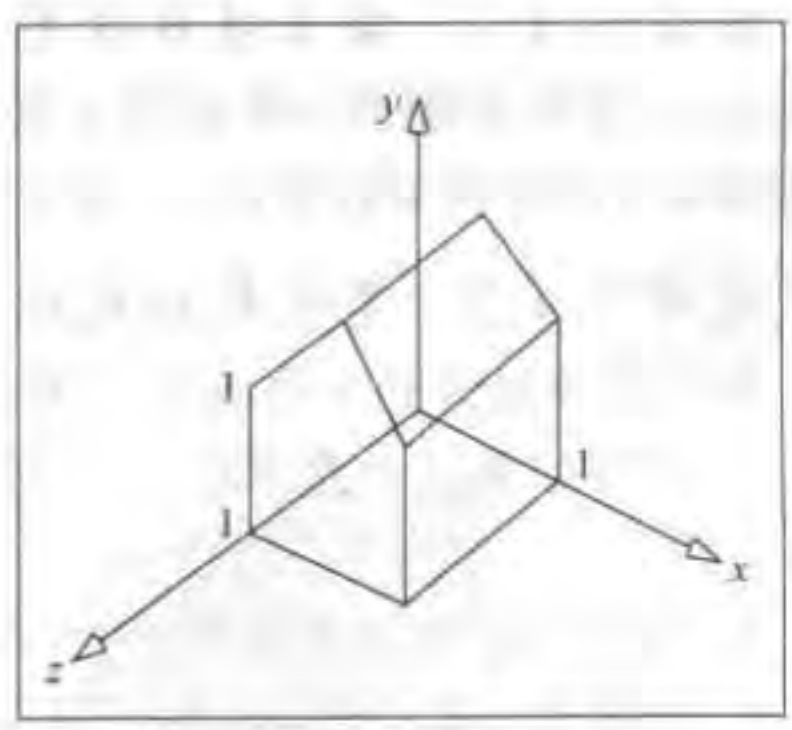


图 7.18　仓房的图示

视图 #1：我们首先把摄像机的 *eye* 设置在(0，0，2)，*look* 设置在(0，0，)，**u**=(1，0，0)和 **v**=(−1，0，0)。我们把近平面设置在距 *eye* 为 1 的地方(因此近平面恰好与仓房的前壁重合)。在摄像机坐标系，所有仓房前壁的点都有 $P_z=-1$，后壁上的点有 $P_z=-2$。因此由等式(7.4)可知，在前壁上面的任意点(P_x，P_y，P_z)都投影到：

$$P' = (P_x, P_y) \quad \text{\{前壁点的投影\}}$$

任意后壁上的点投影到：

$$P' = (P_x/2, P_y/2) \quad \text{\{后壁点的投影\}}$$

后壁上点投影的透视缩短系数都是 2。图 7.19(a)展示了这种视图下仓房的投影。注意后壁上面的边的投影长度为原来的一半。还注意到在三维情况下仓房的平行边投影后不一定还是平行的(我们会看到和视平面平行的平行边投影后还是平行的，不平行于视平面的

平行边的投影是不平行的,它们相交于"灭点")。

视图#2:这里的摄像机已经经过平移了,使得 $eye=(0.5,0,2)$,但是假设 **u** 和 **n** 都是不变的。图 7.19(b)展示了这个投影。

视图#3:这里我们使用的摄像机 $eye=(2,5,2)$,$look=(0,0,0)$,结果如图 7.20 所示。加进的世界坐标系可以起到一个指导作用。这个视图从一个观察点展示出了仓房模型,但是很难从线框图中分辨出每一个面的真正位置。

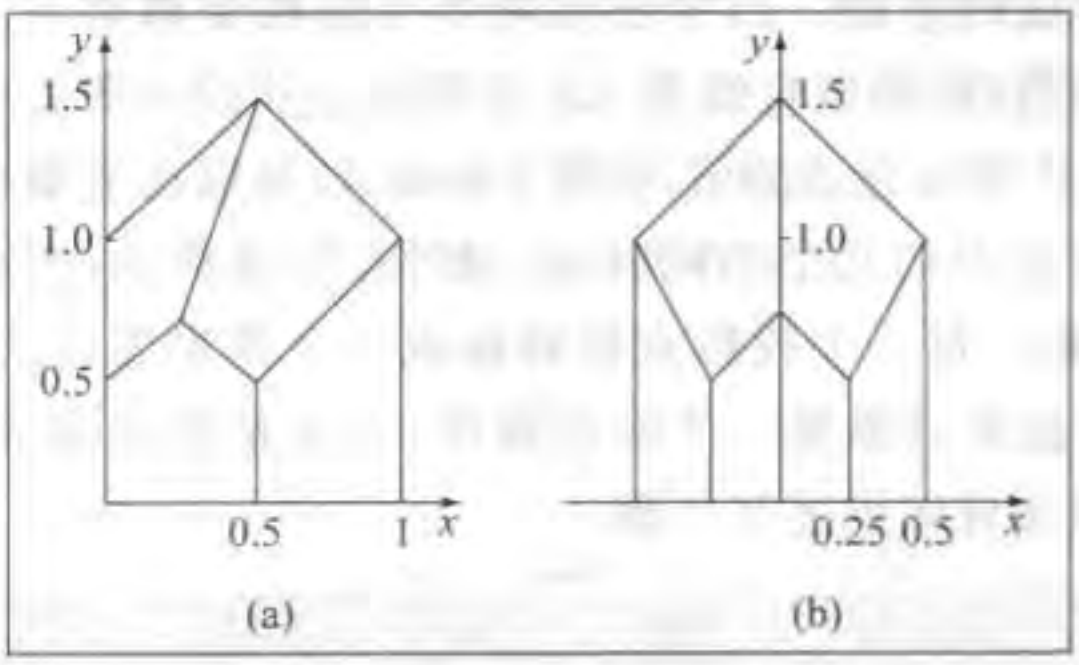


图 7.19 在视图#1 和视图#2 下的仓房投影

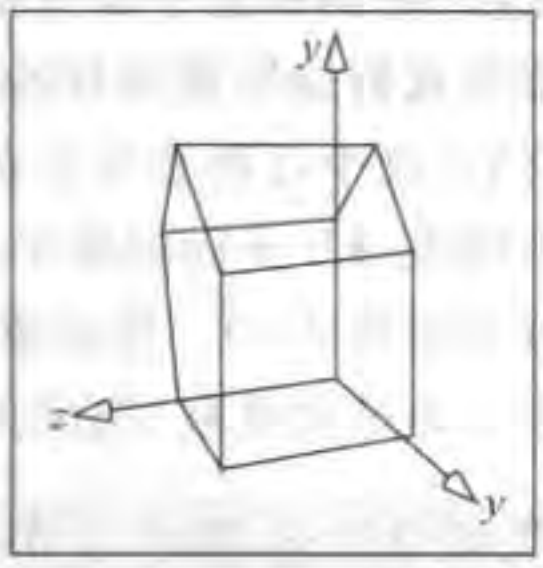


图 7.20 第三个视角下的仓房图像

练习

练习 7.4.1 绘出透视视图下的立方体

绘制(在你的图形计算器或者纸上)透视视图下的立方体 C(沿着坐标轴,中心在原点,边长为 2),眼睛在 z 轴上 $E=5$ 处。如果立方体中心移动到(1,1,1),请重复上面的过程。

练习 7.4.2 光线投射到视平面上什么位置(不要跳过这个问题)

我们想通过找到从原点到 P 点的光线和视平面的交点来得到等式(7.4)。

a. 证明如果这个光线在 $t=0$ 时在原点,$t=1$ 时在 P 点,那么这个光线有一个参数方程 $r(t)=Pt$。

b. 证明光线和视平面交于 $t=N/(-P_z)$

c. 证明交点是 $(x^*,y^*)=(NP_x/(-P_z),NP_y/(-P_z))$

7.4.2 直线的透视投影

这里我们通过考察直线的投影方式来获得透视投影的一些有趣性质。

(1) 我们知道三维投影把平行直线投影到直线,但是这些直线不一定平行。如果不平行,它们相交于灭点。

(2) 通过眼睛后面的直线会穿越无穷远处(这样的直线会被裁剪掉)。

(3) 透视投影通常会生成具有真实感的几何图像。但是对于平行于视平面的很长的直线而言,这种真实感会不太自然。

平行直线的投影

我们假设三维空间中一条直线经过点 $A=(A_x,A_y,A_z)$,并且具有方向 $\mathbf{c}=(c_x,c_y,c_z)$。

因此它的参数表达式为 $P(t)=A+\mathbf{c}t$。使用这个等式代替等式(7.4)中相关的项，得到这条投影直线的参数表达式：

$$p(t)=\left(N\frac{A_x+c_xt}{-A_z-c_zt},N\frac{A_y+c_yt}{-A_z-c_zt}\right) \tag{7.5}$$

(这或许看起来不像是一条直线的参数表达式，但是事实上它是。参见练习)这样在三维空间中点 A 投影到点 $p(0)$，当 t 变化时，投影点 $p(t)$ 移动划过屏幕形成一条直线。我们从这个等式可以发现很多重要的性质。

假设直线 $A+\mathbf{c}t$ 平行于视平面。那么 $c_z=0$，并且投影直线表示为：

$$p(t)=\frac{N}{-A_z}(A_x+c_xt,A_y+c_yt)$$

这个是直线的参数表达式，斜率为 c_y/c_x。这个斜率不依赖于直线的位置，仅仅和方向 $\mathbf{c}$ 有关。那么在三维空间中所有拥有方向 $\mathbf{c}$ 的直线都会投影到这个斜率，所以它们的投影也是平行的。我们可以总结：

如果三维空间中两个直线相互平行并且平行于视平面，那么它们的投影是两条平行线。

反之，考虑方向 $\mathbf{c}$ 不平行于视平面的情况。为方便起见假设 $c_z<0$，因此当 t 增加时，直线向后退离眼睛越来越远。当 t 非常大时，等式(7.5)变为(因为乘 t 的项比其他项要重要)：

$$p(\infty)=\left(N\frac{c_x}{-c_z},N\frac{c_y}{-c_z}\right)\quad\text{(直线的灭点)} \tag{7.6}$$

这个点叫做这条直线的**灭点**：也就是当 t 变得越来越大时直线不断接近的点。注意这个点只依赖于直线的方向 $\mathbf{c}$ 而与它的位置无关(在 A 中包含)。这样，所有的平行线共享一个灭点。特别地，由于这些被投影的线相交于一点，所以它们是不平行的。

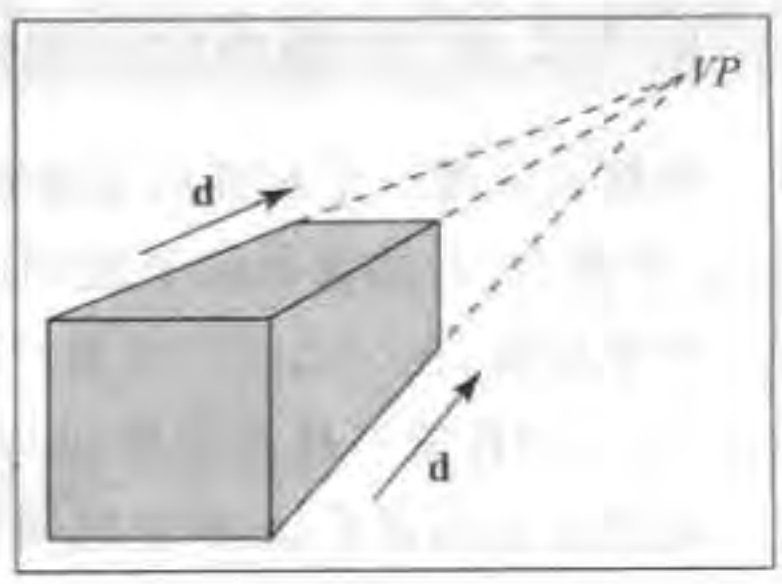


图 7.21　平行直线的灭点

图 7.21 中立方体的例子更加生动地说明了这一点。立方体的几条边都是平行的：比如水平的那些边，垂直的那些边，从视点向后退的那些边。这个图像是在摄像机朝某一方向使得它的近平面平行于立方体的前面时得到的。这样在摄像机坐标系中水平和垂直边 $\mathbf{c}$ 的 z 分量都是 0。因此水平边和垂直边都投影到平行线。那些向后退的边因为不平行于视平面，因此都聚集于一灭点(VP)。艺术家经常这样建立一幅绘画：选择灭点，构建指向 VP 的平行线。接下来我们应该了解更多关于灭点的知识。

图 7.22 从几何学角度对灭点进行了示意。从上向下看摄像机的 xz 平面，我们看到眼睛正在观察直线 AB 上面的不同的点。A 投影到 A'，B 投影到 B'，等等。非常远的点投影到 VP，如图所示。从点到 VP 的直线必须平行于 AB(为什么?)。

从视点后面经过的直线

之前我们已经看到投影视点平面上的一点会得到分母 0，并且如果一定要投影它肯定

会引起错误。我们现在投影一条线段,它的一端在眼睛前面,另一端在眼睛后面。

图 7.23 中再一次从上向下看摄像机。在视点前面的点 A 以一种合理的方式投影到点 A'。在另一端的位于眼睛后面的点 B 同样投影到 B',它似乎位于了视平面错误的一边。考虑点 C 从 A 移动到 B 时,画出它是怎么投影的。当 C 向着视点平面往回移动时,它的投影点沿着视平面越来越向右滑动。当 C 接近视点所在的平面时,它的投影点到了无穷远处,当 C 在视点后面运动时,它的投影点"绕过无穷远"从相反的方向出来了【Blinn96】。如果我们在算法中试图绘制出这样的一条直线,那么结果可能是一片混乱。所以一般来说直线中比近平面更靠近视点的那一部分会在投影之前就被裁剪掉。

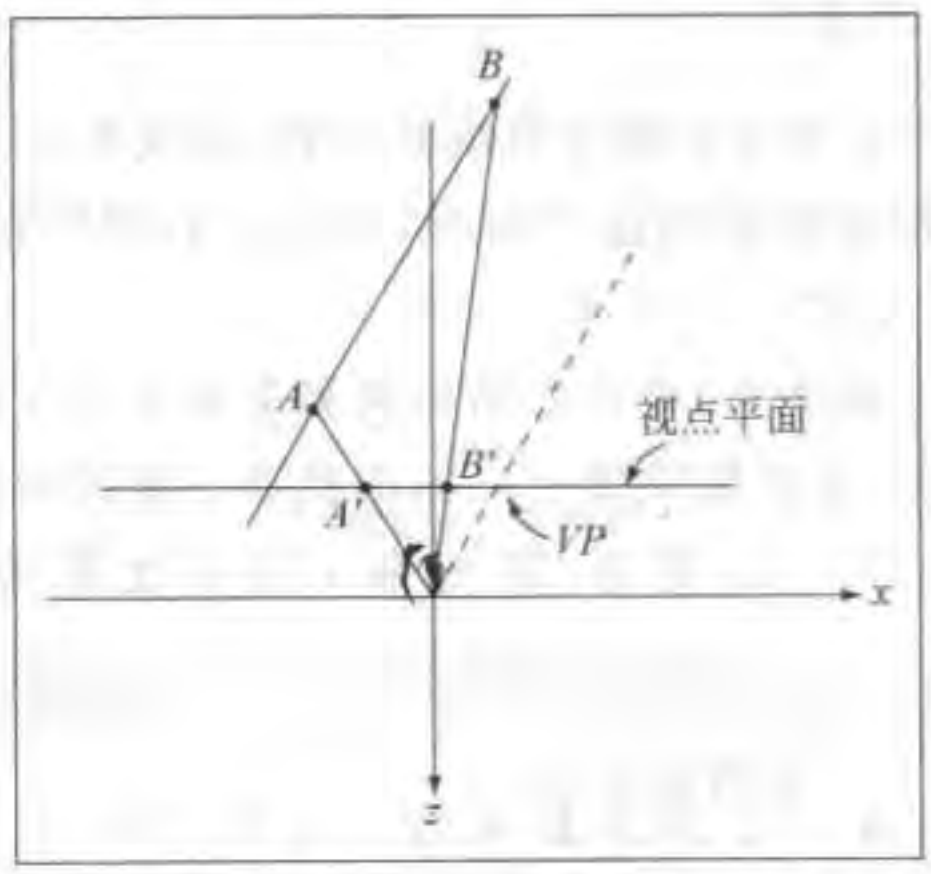


图 7.22 灭点的几何示意图

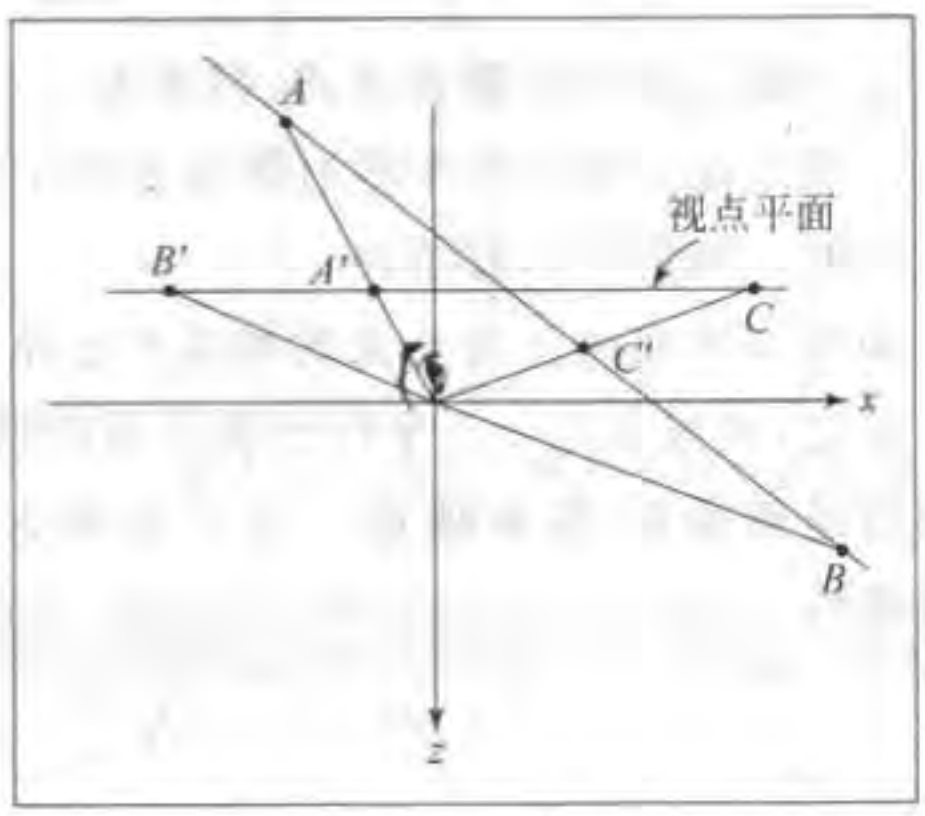


图 7.23 直线 AB 的投影,其中点 B"在视点后"

例 7.4.3 透视中的经典水平平面

观察灭点的一个好办法是观察透视情况下的直线网格,如图 7.24 所示。这里网格线处在 xz 平面上,且均为单位长度间隔。眼睛位于平面上方一个单位处,在(0,1,0),沿着 $-\mathbf{n}$ 的方向看,其中 $\mathbf{n}=(0,0,1)$。和往常一样我们取 $\mathbf{up}=(0,1,0)$,N 定为 1。

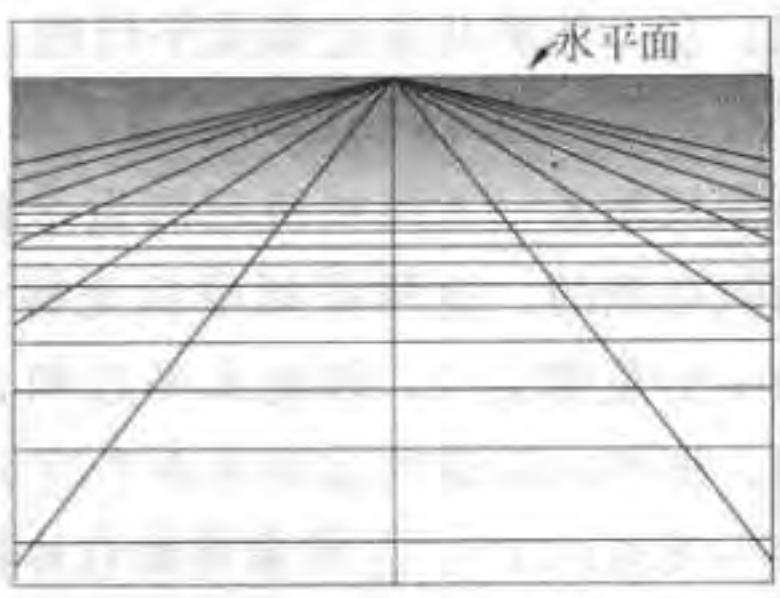


图 7.24 在 xz 平面的水平栅格的透视效果

在视点坐标系下,x 不变的网格线的参数表达式为 $(i,0,t)$,其中 $i=\cdots,-2,-1,0,1,2,\cdots$,$t$ 从 0 到 ∞。通过等式(7.4),第 i 行直线投影到 $(-i/t,2/t)$,是一条经过灭点(0,0)的直线。所有这些直线如预期一样都相交于这个灭点。

在视点坐标系下,z 不变网格线的参数表达式为 $(t,0,-i)$,其中 $i=1,2,\cdots,N$,t 从 $-\infty$ 到 ∞。这些直线投影到 $(-t/i,-2/i)$,这些都是水平直线。它们的投影直线都是平行的,因为网格线本身平行于视平面,越远的直线相互靠得越近,这为我们提供了一个生动的透视缩短的实例。很多遥远的线没有画出来,因为它们已经密得看不清楚了。地平线由所有的 z 是很大的负值的线组成;它位于 $y=0$ 处。

观察很长的平行线时的异常情况

透视投影应该是我们观察方式的合理模型。但是有很多异常情况，这主要由于我们的眼睛没有平坦的观察屏幕。这个问题发生在非常长的物体上面，例如你向上看一对平行的电话线，如图 7.25(a)所示。

从透视视角，如果我们把视平面设置得和电话线平行，我们知道图像将会显示两条平行直线(见图 7.25(b))。但是我们看到的却是不同的情况：电线好像是弯曲了，并且两边都集中于灭点(见图 7.25(c))。实际上这个异常几乎是不可见的，因为窗口或者你的眼睛把观察到的区域限制在一个合理的区域(为了看到电线不同的部分，你需要向上向下翻转你的眼睛，这当然会旋转视平面)。

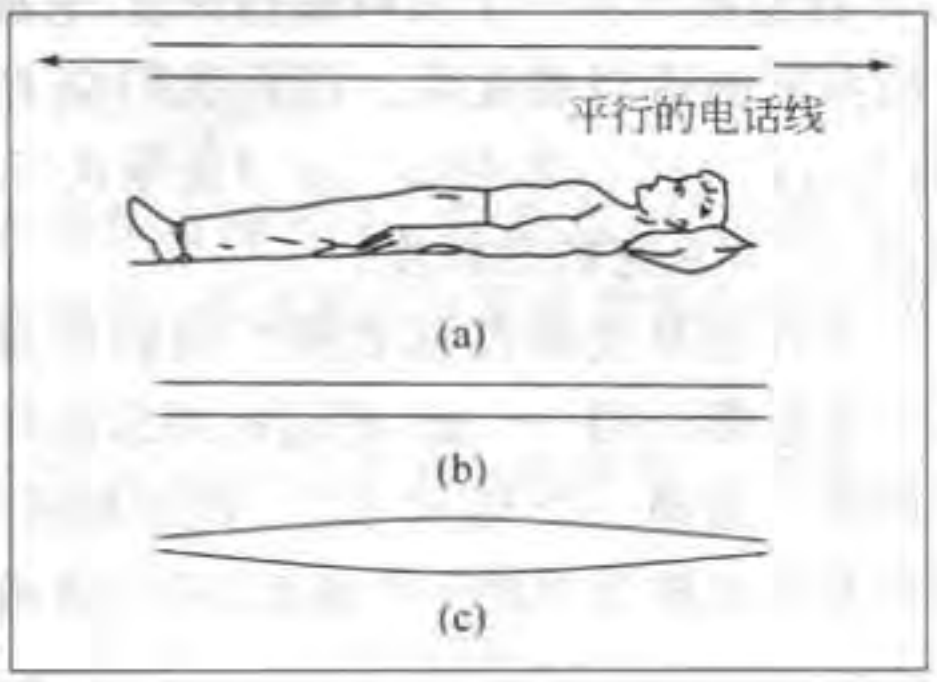


图 7.25 非常长的两条平行线的观察效果

练习

练习 7.4.3 直线投影成直线：参数形式

证明等式(7.5)中的参数表达式是一条直线。提示：对于 x 分量，把分母除到分子上得到 $-A_xN/A_z+Rg(t)$，其中 R 依赖于 A 和 $\mathbf{c}$ 的 x 分量，但不依赖于 y 分量，同样 $g(t)$ 是 t 的函数，不依赖于 x 或者 y 分量。y 分量重复上述过程，得到 $-A_yN/A_z+Sg(t)$ 具有相似的性质。讨论这个为什么是一条直线的参数表示(虽然当 t 变化时点的移动速度并不恒定)。

练习 7.4.4 推导水平网格的结果

推导例 7.4.3 中水平网格投影的参数表示形式。

7.4.3 在图形绘制管道中加入透视投影

我们需要图形绘制管道把物体的顶点投影到近平面上，然后再映射到视口中。在通过模型视点矩阵以后，顶点就变为在摄像机坐标系中的表示，同时等式(7.4)展示了我们所要计算的恰当的投影值。我们需要作裁剪，然后把未被裁剪的点映射到视口。但是除此之外，我们还需要做一些额外的工作。

增加伪深度

在进行投影时我们会丢弃深度信息——也就是说点离眼睛的距离。但是我们一定不能完全丢弃这些信息，否则后面无法进行隐藏面消除。

在视点坐标下，一个点到眼睛的实际距离是 $\sqrt{P_x^2+P_y^2+P_z^2}$，对感兴趣的每一个点计算这个值是烦人和耗时的。我们真正想要知道的只是一种距离的度量方法，使得当两个点投

影到近平面上同一个点时能够告诉我们那一个点更近一些。图 7.26 展示了位于同一条从眼睛出来的直线上的两点 P_1 和 P_2,因此它们投影到同一个点。我们必须能够检验到底 P_1 是遮挡 P_2 还是 P_2 遮挡 P_1。因此对于投影的每一个点 P 我们都计算一个值叫做伪深度,它提供一个 P 的深度的适当的度量。以后我们说 P 点投影到(x^*, y^*, z^*),其中(x^*, y^*)是等式(7.4)提供的值,z^* 就是它的伪深度。

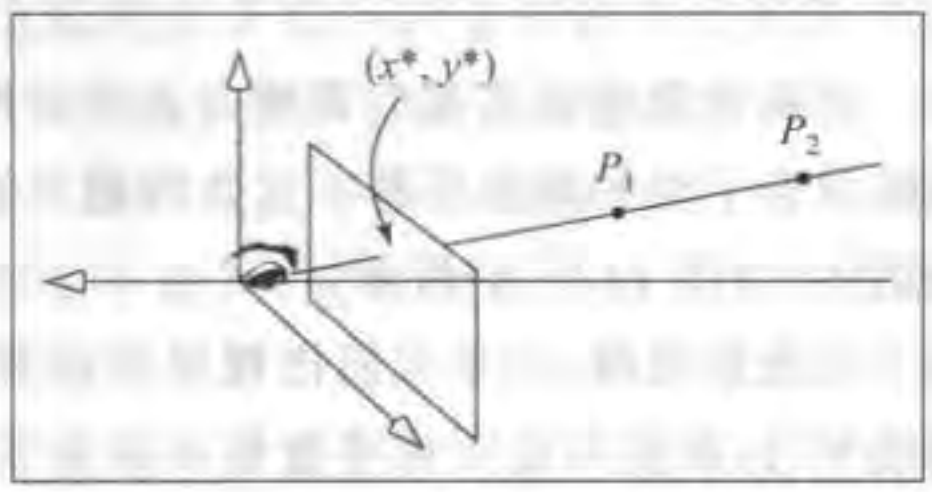


图 7.26 P_1 和 P_2 哪个点更近?

怎样选择伪深度函数呢?我们注意到如果两个点投影到同一个点,更远的点总是有一个更大的负值 P_z,因此我们使用 $-P_z$ 作为伪深度的值。选择一个与 x^* 和 y^* 具有相同分母 $-P_z$ 的函数是很协调和高效的,因此我们尝试使用具有这样分母的一个函数,同时该函数的分子与 P_z 成线性关系,我们说 P 投影到:

$$(x^*, y^*, z^*) = \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, N\frac{aP_z+b}{-P_z}\right) \tag{7.7}$$

选择常量 a 和 b。尽管很多 a 和 b 都很好用,但是我们应该选择它们使得伪深度处于 -1 和 1 之间(后面会看到这个选择为什么好)。由于当点沿着 z 轴向下移动得更远时深度不断增加,我们决定当 $P_z=-N$ 时伪深度为 -1,当 $P_z=-F$ 时伪深度为 1。在这种情况下,我们很容易计算出 a 和 b,得到:

$$a=-\frac{F+N}{F-N},\quad b=\frac{-2FN}{F-N} \tag{7.8}$$

图 7.27 绘制出了伪深度与 $-P_z$ 的相对关系。它从近平面上点的值 -1 增长直到远平面上点的值 1。当 P_z 靠近 0 时(此时点就在眼睛前),伪深度接近 $-\infty$。对于在眼睛后的点,伪深度是一个很大的正值。但是我们会裁剪掉所有比近平面还近的点,因此这种灾难性的行为不会出现。

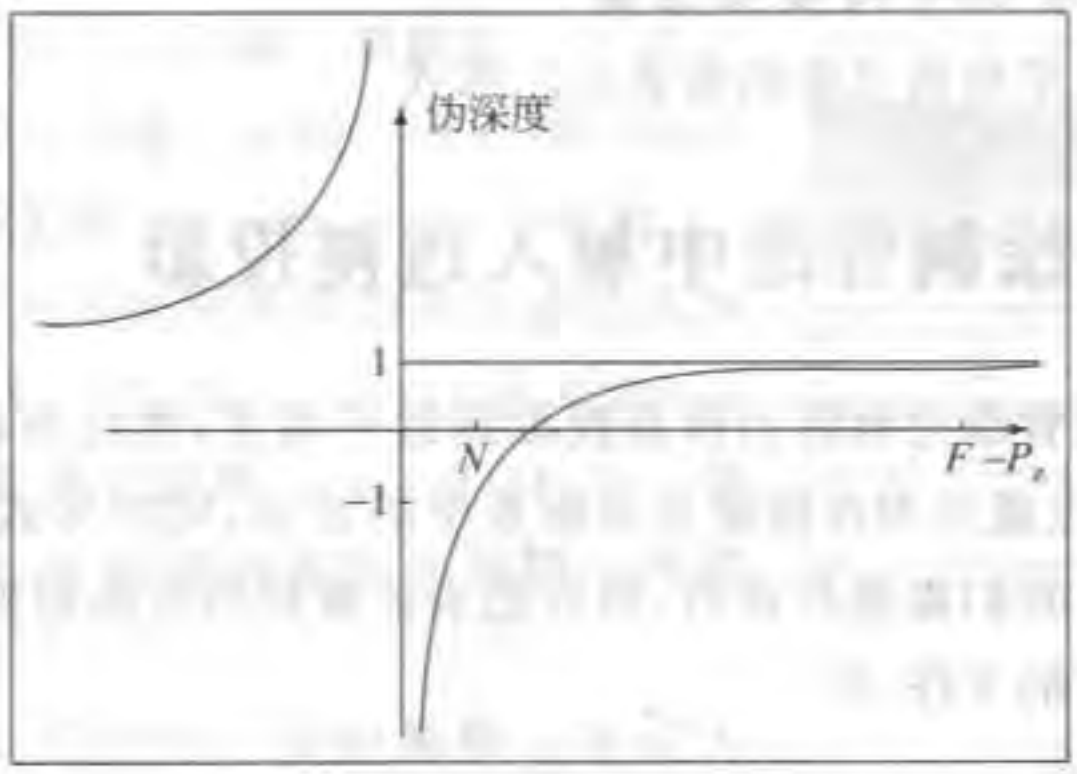


图 7.27 伪深度随着 Pz 增加

另外我们也注意到当$(-P_z)$越来越接近 F 的时候,伪深度变化的速度却越来越慢。由于计算机做的是有限精度的代数运算,所以当进行隐藏面消除的时候,要区分出两个点的伪深度就会产生问题:尽管两点距离视点的实际深度并不相同,但是它们的伪深度值却相同。

留意到按照等式(7.7)中定义,当 P_z 越往负方向变化那么伪深度就越往正方向变化。

这看上去是十分合理的，因为当 P_z 沿负方向越走越远时，距离视点的深度越来越大。

例 7.4.4　当($-P_z$)越来越接近 F 时，伪深度变化得越来越慢

假设 $N=1$，并且 $F=100$。这样就有 $a=-101/99$ 和 $b=-200/99$，由此可以计算出伪深度：

$$\text{pseudodepth}\Big|_{N=1,F=100}=\frac{101P_z+200}{99P_z}$$

这样 $P_z=-N$ 时伪深度等于 -1，$P_z=-F$ 时伪深度等于 1。但是当 P_z 越接近 $-F$ 时，伪深度变化的越来越慢。比如 P_z 分别取 97、98 和 99，那么可以计算出伪深度取值相应为 1.041028、1.040816 和 1.040608。

通过少量的代数运算(参见练习)就可以看出当 N 远小于 F 时，伪深度有如下近似等式：

$$\text{psedodepth}\approx 1+\frac{2N}{P_z} \tag{7.9}$$

再一次可以看出当($-P_z$)接近 F 的时候，伪深度变化得越来越慢。但是当 N 取较大的值时，这种变化速度是可以加快的。所以 N 应该尽可能地取较大的值(当然取值不能任意的大，以至于把距离照相机最近的物体都裁剪掉了)。

使用齐次坐标

在等式(7.7)中我们为什么要在每一项中添加同一个分母？下面我们将要讲到，这可以使得图形绘制管道中截至目前的所有操作步骤都使用矩阵乘法来完成，处理起来更加一致和高效。(一些图形卡上的芯片可以使用硬件来完成点与矩阵相乘的操作，运算速度相当快!)这样做还为高效可靠的裁剪步骤提供了条件。

这个新方法要求我们使用齐次坐标来表示点。因为在齐次坐标下使用 modelview 矩阵对顶点实行变换很简单，事实上我们以前已经这样做过。但是我们不会仅仅局限于此，而是要进一步探讨齐次坐标表示的概念，以发掘它新的能力。特别地，矩阵不仅可以表示仿射变换，还可以表示**透视变换**。

截止到目前，我们所说的点 $P=(P_x,P_y,P_z)$ 的齐次坐标形式为 $(P_x,P_y,P_z,1)$，而向量 $\mathbf{v}=(v_x,v_y,v_z)$ 的齐次形式是 $(v_x,v_y,v_z,0)$。也就是在普通坐标形式后面添上 1 或 0 就可以得到相应的齐次坐标形式。这样我们就有可能使用坐标系作为一组基来表示感兴趣的点和向量，并且能够用矩阵来表示仿射变换。

现在我们把上述思想进行扩展，定义点 $P=(P_x,P_y,P_z)$ 有一族齐次表示 (wP_x,wP_y,wP_z,w) 这里 w 可以取不为 0 的任意值。例如点(1,2,3)可以表示为(1,2,3,1)、(2,4,6,2)、(0.003,0.006,0.009,0.001)、(−1,−2,−3,−1)等等。如果某人给你一个齐次形式表示的点，比如(3,6,2,3)，并且要你说出这个点是什么，这样只需要用它除以其最后一个分量得到 $\left(1,2,\frac{2}{3},1\right)$，然后再把最后一个分量抹去：点的普通坐标形式为 $\left(1,2,\frac{2}{3}\right)$。这就是说：

- 要想把点的普通坐标转化为齐次坐标，在其后面添上 1。
- 要想把点的齐次坐标转化为普通坐标，需要用最后一个分量去除所有的分量，然后

再把第四个分量抹去。

把一个点的所有分量同等程度地放缩后依然表示的是这个点,这个额外的性质正是"齐次"名字的由来。到目前为止,我们处理的都是最后的分量等于 1 的情形。

在练习中我们会继续研究齐次坐标,但是现在我们把精力集中于在点变换过程中齐次坐标的变化规律。请回忆仿射变换矩阵的第四个行向量总是等于(0,0,0,1)。因此我们用这样的矩阵 M 乘上点 P 的齐次坐标形式,有 $MP=Q$(回顾等式(5.24)),正如下面的例子所示

$$\begin{bmatrix} 2 & -1 & 3 & 1 \\ 6 & 0.5 & 1 & 4 \\ 0 & 4 & 2 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{bmatrix} = \begin{bmatrix} wQ_x \\ wQ_y \\ wQ_z \\ w \end{bmatrix}$$

Q 的最后一个分量总不会改变:依然等于 w。因此我们可以用通常的方法把 Q 变回到普通坐标形式。

但是如果我们把第四个行向量(0, 0, 0, 1)进行变动的话,那么就会产生一些新情况。考虑一个重要的例子,它的第四个行向量是(0, 0, −1, 0),(这与我们将要讲到的"投影矩阵"很相似):

$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \quad \text{(投影矩阵 —— 版本 1)} \tag{7.10}$$

其中的 N、a、b 为任意值。用它乘上一个齐次坐标表示的点,其中 w 是任意值:

$$\begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} wP_x \\ wP_y \\ wP_z \\ w \end{bmatrix} = \begin{bmatrix} wNP_x \\ wNP_y \\ w(aP_z+b) \\ -wP_z \end{bmatrix}$$

它对应了哪一个普通坐标下的点?先除以第四个分量,然后再将它抹去,可得

$$\left(N\frac{P_x}{-P_z},\ N\frac{P_y}{-P_z},\ \frac{aP_z+b}{-P_z}\right)$$

这就是我们使用等式(7.7)得到的结果。因此使用了齐次坐标后矩阵乘法也可以表示透视操作。当然我们还必须把结果除以第四个分量,这一个步骤也称为**透视除法**。

第四个行向量不是(0,0,0,1)的矩阵执行的并不是仿射变换。它代表的是范围更广的一类变换,称为**透视变换**。它属于变换操作而非投影。投影会降低点的维数,把它变为一个 3 元组或 2 元组,而透视变换把 4 元组依然变成 4 元组。

请思考一下,把非零值比如(A,B,C,D)添加到等式(7.10)中矩阵的第四行,会在代数运算方面产生什么效果。当你用矩阵乘以(P_x,P_y,P_z,1)(或它的倍数),结果中的第四项为 $AP_x+BP_y+CP_z+D$,它是与 P 的每一分量线性相关的。当做完透视除法后,这一项会出现在点的分母中。这样的分母也正是我们想要的,它在几何上产生的效果就是往一般平面上的透视投影,我们在练习中会证明这一点。

因此透视变换把一个 3D 点 P 变为另一个 3D 点 P',相应的等式为

$$(P_x, P_y, P_z) \rightarrow \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z+b}{-P_z}\right) \quad \text{“透视变换”} \tag{7.11}$$

那么投影这一部分操作在哪里执行呢？在图形绘制管道中这个点的前两个分量被用于绘制：计算点在屏幕坐标系下的绘制位置。被剥离开的第三个分量用于深度测试。如果要计算点在屏幕上的位置，就要忽略掉第三个分量，这可以通过把它置为 0 实现：

$$\left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, \frac{aP_z+b}{-P_z}\right) \rightarrow \left(N\frac{P_x}{-P_z}, N\frac{P_y}{-P_z}, 0\right) \quad \text{“投影”} \tag{7.12}$$

这也正是我们在第 5 章中曾经做过的，那里我们放置好照相机进行 3D 场景绘制的时候要对点实施正交投影（即与视平面垂直）。后面我们还会详细研究正交投影。现在我们可以总结出：

（透视投影）=（透视变换）+（正交投影）

这种把透视投影分解为一个特殊变换外加一个（平凡）投影的做法被证明是十分有用的，这不仅体现在算法方面，而且还有助于理解每个点在通过图形绘制管道时实际上都经历了哪些操作。OpenGL 把变换操作的步骤和投影操作的步骤分离开来。事实上，它还会在这两个步骤中插入裁剪、透视除法，以及一个附加映射的操作。

最后还要强调一点。当我们打算显示一个网格模型的时候，必须把几千个甚至几百万个顶点发送到图形绘制管道。显然，如果我们只让每个顶点进行一次矩阵乘法操作而非一系列的多次矩阵乘法操作，那么效率就会大大的提升。所以把一个顶点所要经历的所有矩阵乘法合成为一个单一的矩阵会显著地提升效率。这也就是 OpenGL 所做的：它一次把所有要用到的矩阵乘起来合成为一个单一矩阵，然后再用这个合成矩阵去乘以每个顶点。

下面我们会更深一步地了解上述过程中变换操作的部分。

透视变换依据等式(7.11)把 3D 点 P 变成另一个 3D 点，为投影做好准备。把它理解为 3D 空间的变形并且了解一个形状如何变形到另一个形状是十分有帮助的。更重要地，它具有保直线性和保平面性，也就是直线被变换到直线，平面被变换到平面，多边形面片被变换到另一个多边形面片。透视变换还具有保持包含关系的性质，因此如果点 a 在一个物体内，那么变换后的点也在变换后的那个物体内（为了保持这些性质，我们要选择好一个适当的伪深度函数）。在练习中会证明上述性质。

特别地，我们想要看一看照相机的视景体是如何被变换的，因为如果我们打算在变形的空间内进行裁剪，那么就要使用变形的视景体。透视变换在这一点表现得十分出众：变形后的视景体是可以进行简单高效裁剪操作的理想形体！图 7.28 说明了视景体和其他的一些形体是如何被变换的。位于 $z=-N$ 处的近平面被映射到 $z=-1$ 处的平面 W'，并且远平面被映射到 $z=+1$ 的平面。把顶面 T 倾斜，将其映射到水平面 T'，使得它与 z 坐标轴平行。底面 S 变为了水平面 S'，两个侧面也都平行于 z 坐标轴。照相机的视景体被变换为一个平行六面体！

透视变换的几何性质

如果平面中的直线或者平行于近平面或者通过视点，那么这些平面的映射关系就很容易证明。请仔细考虑下面的命题：

命题：通过视点的直线被映射为平行于 z 坐标轴的直线。

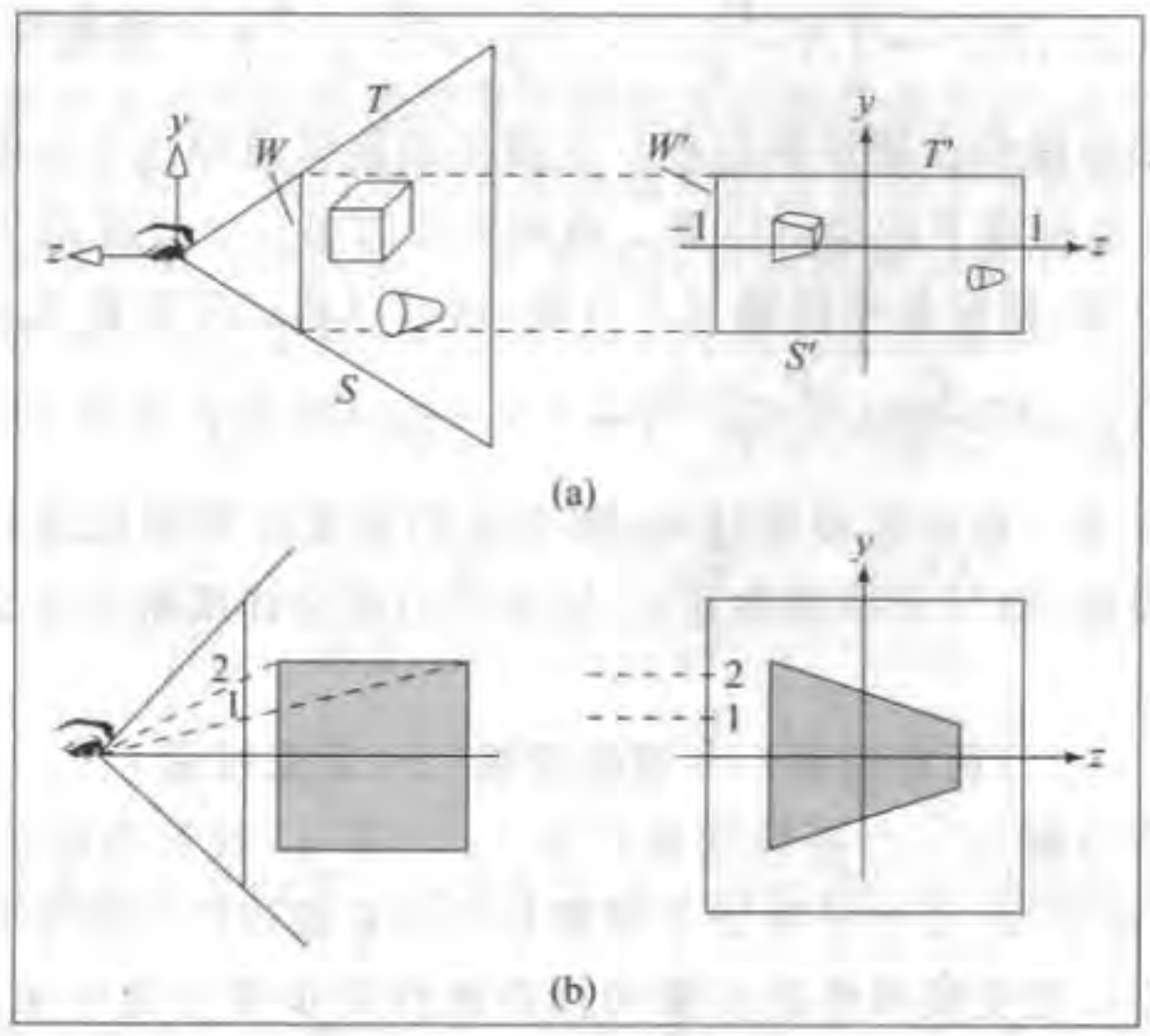


图 7.28 使用透视变换对视景体进行变形

证明：这种直线上的所有点都被投影为视平面上的同一个点，不妨设为(x^*, y^*)。因此直线上的点被变换到点(x, y, z)，其中 $x=x^*$，$y=y^*$，并且 z 取遍 -1 和 1 之间的所有伪深度值。

命题：与 z 坐标轴垂直的直线映射后依然垂直于 z 坐标轴

证明：这种直线上的所有点都具有相同的 z 坐标值，所以它们映射后的点也具有相同的伪深度值。

有了以上事实，就很容易得出变形后的视景体的精确形状和大小。

透视变换也把物体，比如说图中的积木块，变形为其他的形状。图 7.28(b)中一个积木块被投影到了近平面。假设其正面的顶边被投影到 $y=2$，并且背面的顶边被投影到 $y=1$。当对这个积木块实施变换后，它变成一个截断锥体：它正面的顶边位于到 $y=2$，背面的顶边位于 $y=1$。越接近近平面变得越大，越远离近平面变得越小。根据投影规律，变换后的物体它的背面要比正面更小一些。变换后物体的 x 和 y 坐标就是原始物体被投影后的 x 和 y 坐标。当你对变换后物体实行正交投影时会用到这些坐标。简单地说：

物体经透视变换变形后再用正交投影进行绘制，其结果等同于原始物体直接进行透视投影。

因此所有的物体都遵循透视投影的规则被适当地变形为依透视规律缩小的形体。然后再用正交投影进行绘制就可以得到正确的结果。

下面我们深入地研究一下视景体变换后的具体形状和大小。

深入研究变换视景体与正则视景体映射

我们打算给变形前后的视景体分别标出尺寸数值。考虑顶面，如图 7.29 所示它在 $z=-N$ 处通过点$(left, top, -N)$。因为组成顶面的直线通过视点以及近平面中 y 坐标值为 top 的点，所以它必须被变换为平面 $y=top$。同样，

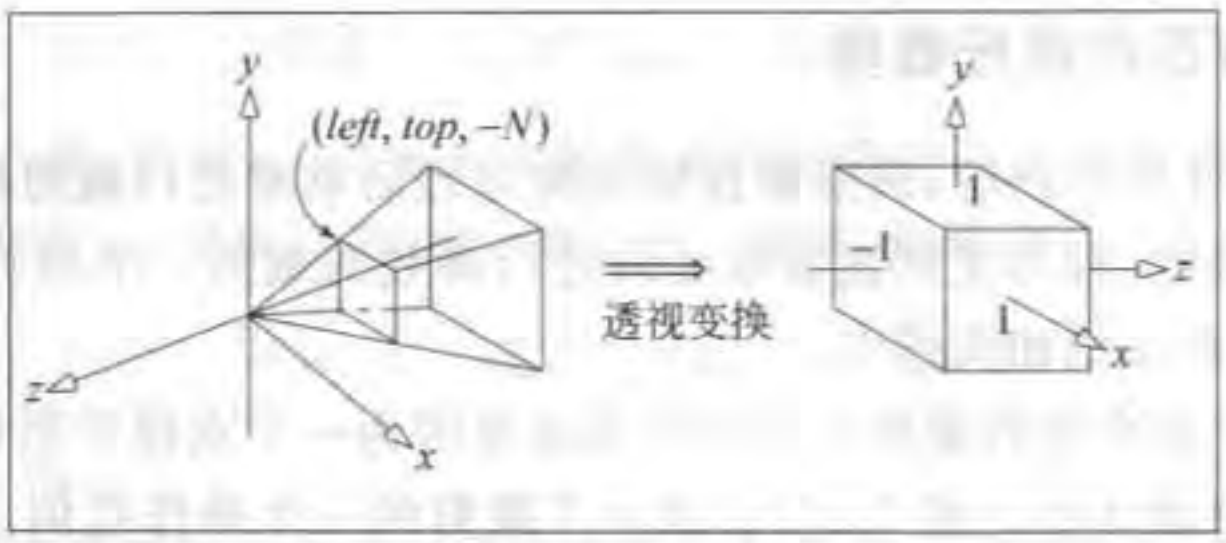


图 7.29 透视变换的细节

- 底面被变换为平面 $y=bott$。
- 左侧面被变换为平面 $x=left$。
- 右侧面被变换为平面 $x=right$。

现在我们对变换后的视景体有了更准确的认识：它是一个平行六面体，它的尺寸大小与照相机的性质有关。我们将会看到，这是一个用于裁剪的绝佳形体，因为它的各个面与坐标面平行。但是如果它的尺寸不依赖于特定的照相机就更好了。OpenGL 在透视变换中又加入了伸缩和平移操作使得上述的平行六面体变成了一个**正则视景体**，它是一个正立方体，每一维的坐标都从 −1 到 1。由于在 x 维和 y 维实行了不同程度的拉伸，把任意场景都变换为一个固定体积的空间，这就势必使形状产生了失真，但是这种失真可以在最终的视口变换后得到消除。

变换后的视景体的 z 维本来就是从 −1 到 1，所以只需要在其他两个维度上进行伸缩操作。因此在 x 和 y 方向加入伸缩和平移操作就可以把平行六面体变换为正则视景体。我们先在 x 方向平移 $-(right+left)/2$，在 y 方向平移 $-(top+bott)/2$。然后在 x 方向乘上尺度因子 $2/(right-left)$，在 y 方向乘上尺度因子 $2/(top-bott)$。当这些矩阵乘法操作完成后，我们就得到了最终的矩阵形式：

$$R=\begin{pmatrix} \frac{2N}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\ 0 & \frac{2N}{top-bott} & \frac{top+bott}{top-bott} & 0 \\ 0 & 0 & \frac{-(F+N)}{F-N} & \frac{-2FN}{F-N} \\ 0 & 0 & -1 & 0 \end{pmatrix} \quad \text{（透视矩阵）} \tag{7.13}$$

这个矩阵被称为**透视矩阵**，它除了完成透视变换之外，还完成了伸缩和平移操作，把照相机的视景体变换为正则视景体。当使用函数 glFrustum(left, right, bott, top, N, F)的时候，OpenGL 生成的(要与当前矩阵进行相乘)正是这个矩阵。回想我们以前倒是经常使用函数 gluPerspective(viewAngle, aspect, N, F)，因为它的参数更加直观一些。它建立的也是同一个矩阵，但是事先要用如下等式计算出 top、$bott$ 等变量的值

$$top = N\tan\left(\frac{\pi}{180}viewAngle/2\right)$$

$$bott=-top,\quad right=top*aspect,\quad \text{以及 } left=-right$$

使用视景体对面片进行裁剪

回想一下图 7.14 中的内容,顶点被投影矩阵处理后就要进行裁剪操作。这种操作是在变形后的空间内完成的,因为正则视景体可以进行高效的裁剪。下面我们开始研究它的机制,并且深入到裁剪算法的相关细节。

在已被变形的空间中进行裁剪之所以行得通是因为一个点位于照相机的视景体中当且仅当它也位于正则视景体中。图 7.30(a)展示了裁剪的一个操作范例。三角形的三个顶点是 v_1、v_2 和 v_3。顶点 v_3 位于正则视景体 CVV 之外。裁剪操作作用在边上:它首先对边 v_1v_2 进行裁剪但是此边全体都位于 CVV 内部。然后它裁剪边 v_2v_3 并且纪录下新的顶点 a,这是边离开 CVV 的位置。最后它裁剪边 v_3v_1 并且记录下此边进入到 CVV 的位置。经过上述过程原始的三角形就变成了顶点为 v_1v_2ab 的四边形(后面我们还会看到,除了计算新顶点的位置,图形绘制管道还会计算这些新顶点的颜色和纹理参数)。

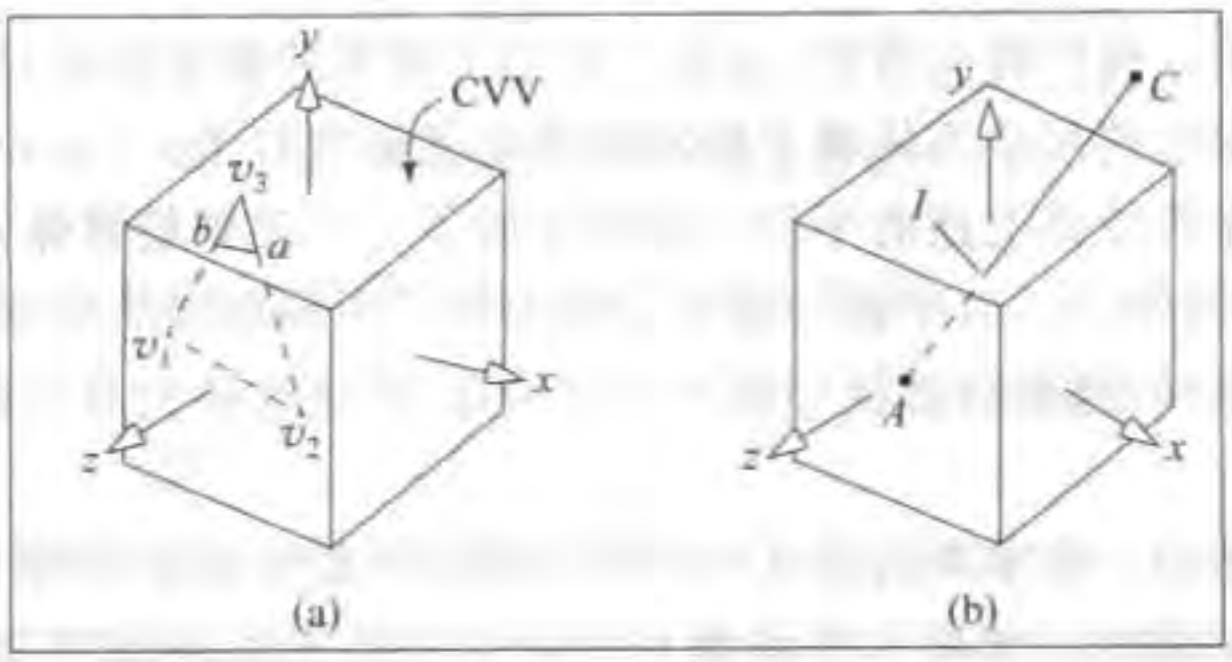


图 7.30 用正则视景体进行裁剪

裁剪的问题基本就是使用 CVV 对线段进行裁剪的问题。在 4.8.3 小节中我们已经讨论过这样的一个算法,Cyrus-Beck 裁剪算法。我们这里要开发的裁剪算法与它很相似,但是当然用于 3D 而不是 2D 的环境。

实际上,它是处理的 4D 情形。我们的裁剪操作作用在 4D 齐次坐标空间,称之为"裁剪坐标",如图 7.14 所示。这比看上去的还要简单,并且可以很好地区分出一个点是在视点之前还是在视点之后。

在图 7.30(b)中假设我们打算用 CVV 裁剪线段 AC。这就是说给出齐次坐标表示的两个点 $A=(a_x,a_y,a_z,a_w)$和 $C=(c_x,c_y,c_z,c_w)$,我们要判断出这条线段的哪一部分位于 CVV 内。如果线段和 CVV 的边界相交,我们需要计算出交点 $I=(I_x,I_y,I_z,I_w)$。

像 Cyrus-Beck 算法一样,我们把 CVV 看成是六张无限延伸的平面,然后逐个地对每个平面考虑给定的边与它形成的位置关系。我们把边表示为参数形式 $A+(C-A)t$。A 点的参数为 0,C 点的参数为 1。针对 CVV 的每个面,我们首先测试 A 和 C 是否位于该面的同一侧:如果是的话就没有必要计算边与面的交点了。如果两点分属不同的两侧,那么我们计算出交点的位置然后裁剪掉位于平面外部的那一部分。

所以我们必须能够判断出一个点是在一张平面的内部还是外部。例如,考虑平面 $x=-1$。如果对于点 A 有

$$\frac{a_x}{a_w} > -1 \quad \text{或者} \quad a_x > -a_w \quad \text{或者} \quad (a_w + a_x) > 0 \tag{7.14}$$

则点 A 位于平面的右侧(内部)。当不等式的两端同时乘上一个负项时,不等号的方向必须改变。但是我们处理的 a_w 只取正值(参见练习)。同样,如果有

$$\frac{a_x}{a_w} < 1 \quad \text{或者} \quad (a_w - a_x) > 0$$

则 A 位于 $x=1$ 的内部。

Blinn【Blinn96】把这些值称为点 A 的"边界坐标",并且他列出了我们要用到的六个这样的值,如图 7.31 所示。

边界坐标	齐次坐标	裁剪面
BC_0	$w+x$	$x=-1$
BC_1	$w-x$	$x=1$
BC_2	$w+y$	$y=-1$
BC_3	$w-y$	$y=-1$
BC_4	$w+z$	$z=-1$
BC_5	$w-z$	$z=1$

图 7.31 边端点的边界坐标

我们依次为点 A 和点 C 计算上述的六个值。如果一个点的所有六个值都为正,那么该点就位于 CVV 内部。如果都为负,则该点位于外部。如果两个端点都位于内部,我们称这种情况为"平凡的接受",在 3.3 节中的 Cohen-Sutherland 算法中处理过相同的情况。如果点 A 和点 C 都位于同一个平面的外侧(相应的边界坐标 BC 为负),那么这条边必定全部位于 CVV 之外。

平凡的接受:两个端点都位于 CVV 之内(所有 12 个 BC 都为正)。

平凡的拒绝:两个端点都位于 CVV 的同一个面的外部。

如果不是上述的两种情况,那么我们就必须单独使用每一张平面对线段 AC 进行裁剪。和 *Cyrus-beck* 算法一样,我们需要记录下**候选区间**(CI)(参见图 4.45),它是由 t 值组成的区间,边只有在这一部分取值才可能位于 CVV 内部。我们知道它的相反情形:如果 t 在 CI 外取值,那么可以肯定它不在 CVV 内部。CI 的取值范围从 $t=t_{in}$ 到 t_{out}。

我们使用每一张面轮流地对边进行测试。如果相应的边界坐标符号相反,那么我们可知边与平面相交于某一 t_{hit},然后需要把它计算出来。若边是进入方向的(当 t 值增加时由平面外部进入到平面内部)那么我们更新 $t_{in} = \max(\text{old}t_{in}, t_{hit})$,这样做是因为进入到 CVV 时的 t 不可能早于 t_{hit}。同样的,如果边是退出方向的,我们更新 $t_{out} = \min(\text{old}t_{out}, t_{hit})$。若在某一时刻 CI 变成了空区间($t_{out} > t_{in}$),那么我们就可以把整个边都裁剪下去然后"提前退出"算法程序,节省不必要的计算。

很容易计算边与平面相交时的 t 值。把边用参数表示为齐次坐标形式:

$$\text{edge}(t) = (a_x + (c_x - a_x)t, a_y + (c_y - a_y)t, a_z + (c_z - a_z)t, a_w + (c_w - a_w)t)$$

例如计算边与平面 $X=1$ 的交点,那么当 $A+(C-A)t$ 的 x 坐标为 1 时有:

$$\frac{a_x + (c_x - a_x)t}{a_w + (c_w - a_w)t} = 1$$

从中可以很容易解出 t,

$$t = \frac{a_w - a_x}{(a_w - a_x) - (c_w - c_x)} \tag{7.15}$$

留意到 t_{hit} 只取决于两个边界坐标。与其他平面的相交也有类似的等式。

很容易把上面所说的内容写成程序,参见图 7.32 所示的伪代码。它基本上就是 Liang-Barsky 算法【Liang84】,然后再用 Blinn【Blinn96】的一些建议进行了细化。程序 clipEdge(Point4 A, Point4 C)需要的参数是两个齐次坐标表示的点(分量是 x、y、z 和 w),当 AC 与 CVV 无交点时结果返回 0,其他情况返回 1。它同时也对 A 和 C 进行修改,这样当程序结束时,A 和 C 就是裁剪后的边的端点。

```
int clipEdge(Point4& A, Point4& C)
{
    double tIn=0.0, tOut=1.0, tHit;
    double aBC[6], cBC[6];
    int aOutcode=0, cOutcode=0;
    <..计算 A 和 C 边界坐标 BC..>
    <..生成 A 和 C 的 outcode..>

    if((aOutcode & cOutcode) !=0)                    //平凡的拒绝
            return 0;
    if((aOutcode | cOutcode)==0)                     //平凡的接受
            return 1;

    for(int i=0; i <6; i++)                          //每个面都进行裁剪
    {
        if(cBC[i=<0)                                 //退出方向:C 在外部
        {
            tHit=aBC[i]/(aBC[i]-cBC[i]);
            tOut=MIN(tOut,tHit);
        }
        else if(aBC[i] <0)                           //进入方向:A 在外部
        {
            tHit=aBC[i]/(aBC[i]-cBC[i]);
            tIn=MAX(tIn, tHit);
        }
        if(tIn >  tOut) return 0;                    //CI 为空,提前退出
    }
    // update the end points as necessary
    Point4 tmp;
    if(aOutcode !=0)                                 //A 在外部:tIn 被修改过
    {//计算 A 新的值,(但暂时不修改它)
        tmp.x=A.x+tIn * (C.x-A.x);
```

图 7.32 边的裁剪程序

```
        tmp.y=A.y+tIn * (C.y-A.y);
        tmp.z=A.z+tIn * (C.z-A.z);
        tmp.w=A.w+tIn * (C.w-A.w);
    }
    if(cOutcode !=0)                          //C 在外部: tOut 被修改过
    {    //更新 C(使用 A 的原始值)
        C.x=A.x+tOut * (C.x-A.x);
        C.y=A.y+tOut * (C.y-A.y);
        C.z=A.z+tOut * (C.z-A.z);
        C.w=A.w+tOut * (C.w-A.w);
    }
    A=tmp;                                    //现在更新 A
    return 1;                                 //边的一部分在 CVV 中
}
```

图 7.32（续）

程序计算每个端点的六个边界坐标并且把它们存储在 aBC[]和 cBC[]中。为了效率起见，它还为每个点构造了一个 outcode 变量用于保存此点的六个边界坐标的符号。如果 aBC[i]>0（A 位于第 i 个面内部）则 A 的 outcode 的第 i 位置 0 否则置 1。有了这些，当 aOutCode 和 cOutCode 都为 0 就是“平凡的接收”。当两个 outcode 逐位进行“与”操作结果非 0 时就是“平凡的拒绝”。

在循环语句中轮流使用每个面对边进行测试，BC 中最多只能有一个为负（为什么?）。如果点 A 有一个负的 BC 值，那么此边在交点处一定是进入方向的；如果点 C 有一个负的 BC 值，那么此边在交点处一定是退出方向的。（Blinn 的方法要稍微快一些，他使用了一个掩码对 outcode 变量的位进行测试。每次 tIn 和 tOut 更新后，如果 tIn 大于了 tOut 那么就提前退出程序。

当测试过所有平面后，tIn 和 tOut 中的一个或者全部会被修改。如果 tIn 被改动了那么把 A 更新为 $A+(C-A)tIn$，如果 tOut 被改动了那么把 C 更新为 $A+(C-A)tOut$。

Blinn 建议在处理之间提前计算好每个点的 BC 值和 outcode 值。这样当一个顶点为多个边的端点时（经常发生这种情况）就不需要再重复计算这些值了。

我们为什么要用正则视景体进行裁剪?

我们现在已经看出使用正则视景体进行裁剪十分方便，所以对物体实施裁剪之前先将其变换到裁剪坐标就有很大的价值。CVV 的两个重要特性包括：

1. 它无须设定参数：算法不需要额外的信息来描述裁剪视景体。它使用的值无非就是−1 和 1。所以程序代码可以进行高度优化获取最大的效率。

2. CVV 的平面与坐标轴是对齐的（在透视变换执行之后）。这就意味着我们只需要使用一个坐标值就可以判断点位于平面的哪一侧，比如 $a_x>-1$。如果平面不是与坐标轴对齐的话，就还需要做开销比较大的点积运算。

我们为什么要使用齐次坐标进行裁剪,而不是在透视除法之后进行裁剪?

这并不是完全必要的,但是这样做可以使裁剪算法更加简洁快速。做完透视除法之后会破坏掉一些信息:如果你知道了 a_x 和 a_w 的值,当然也就知道了它们的符号。但是如果只给出 a_x/a_w 的商,那么你只能了解到 a_x 和 a_w 是否同号以及它们之间的相对比例关系。使用齐次坐标的值并且裁剪掉比近平面离视点更近的点,这样那些位于视点背面的点就自动被排除掉了,例如图 7.23 中的 B。

还存在一些古怪的情形必须要在齐次坐标下进行裁剪,这在【Blinn96, Foley90】中有所描述。它们牵扯到物体的一些特殊变换或某些曲面的构造问题,即使原始点(a_x, a_y, a_z, a_w)位于视点之前,它的第四个分量也取负值。我们在这里讨论的物体建模问题不会遇到这种情况。我们可以得出结论,使用齐次坐标进行裁剪,即使通常并不是必须的,但是这可以使算法更加快速和简单,而且几乎不产生什么开销。

执行完裁剪操作,最后执行透视除法(如图 7.14 所示),然后三元组(x, y, z)就要经历视口变换。我们后面会讨论,这种变换对 x 和 y 值实行尺寸变换以及平移操作,以便它们可以被恰当地放置在视口之中,同时对 z 分量(伪深度)也做一些小的调整使之更加有利于深度测试。

视口变换

我们已经看到,透视变换把场景都变换到正则立方体中,如图 7.33。如果照相机视景体的宽高比(也就是近平面上的窗口的宽高比)等于 1.5,那么透视变换把物体放到宽高比为 1 的窗口后就会产生明显的失真。但是视口变换通过把正方形映射到宽高比为 1.5 的视口可以把这种失真消除掉。

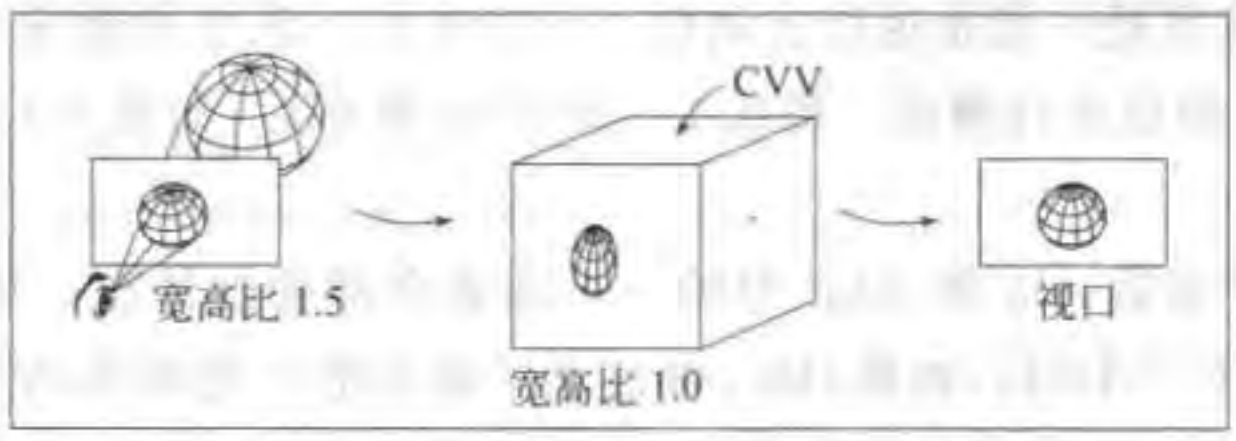


图 7.33 视口变换恢复了原始的宽高比

我们以前经常使用 OpenGL 函数 glViewport(x, y, wid, ht)。它指定视口左下角的屏幕坐标为(x, y),视口的宽等于 wid 个像素,视口的高等于 ht 个像素。因此它指定的视口宽高比等于 wid/ht。视口变换同时也把伪深度由区间[−1, 1]变换到区间[0, 1]。

重新温习图 7.14,它揭示了完整的图形绘制管道。在 OpenGL 中,每个点 P(通常是多边形的一个顶点)要经历如下步骤:

- 通过给 P 添加分量 1 将其**扩展**为齐次 4 元组;
- 用这个 4 元组乘上**矩阵 modelview**,得到视点坐标系下的 4 元组;
- 再将点乘上**投影矩阵**,得到裁剪坐标下的 4 元组;
- 对以此点为端点的边实施**裁剪操作**;

- 进行隐藏面消除操作(参见 8.4 节)；
- 执行**透视除法**，得到一个 3 元组；
- 在视口变换过程中 3 元组被乘上一个矩阵：得到的结果(sx,sy,dz)被用于绘制和深度计算。(sx,sy)是点在屏幕上的显示位置；dz 是用某种尺度表示的原始点距离照相机视点的深度。

练习

练习 7.4.5　点 P 的投影在何处？

假设视平面在照相机坐标系中的方程为 $Ax+By+Cz=D$。请证明任意一点 $P(P_x,P_y,P_z)$在平面上的投影可用齐次坐标表示为

$$P'=(DP_x,DP_y,DP_z,AP_x+BP_y+CP_z)$$

练习 7.4.6　伪深度的近似形式

证明当 N 远小于 F 时，等式(7.9)可以很好的逼近伪深度 $a+b/(-P_z)$，这里的 a 和 b 分别由等式(7.8)给出。

练习 7.4.7　无穷远处的点的齐次坐标

当 w 变得越来越小时，请考虑齐次坐标表示的点(x,y,z,w)的性质。比如当 $w=0.01$ 时，它为$(100x,100y,100z,1)$；当 $w=0.0001$ 时，它为$(10000x,10000y,10000z,1)$，以此类推。点会沿着方向(x,y,z)"趋向于无穷"。事实上，用$(x,y,z,0)$表示的点被称为"无穷远点"。齐次坐标的优点之一就是这种理想点也可以拥有一个完美的有限表达形式：在某些数学推导过程中，这可以消除许多棘手的特例情形。例如，两条直线总是会相交，即使它们相互平行【Ayers67，Semple52】。但是针对一些其他情况，齐次坐标就并不很胜任。两个齐次坐标表示的点的差是什么？

练习 7.4.8　透视变换如何影响直线和平面？

我们必须要证明透视变换具有保平面性和保包含性。

a. 请说明为什么证明了如下情形上述命题就可以成立：直线上一点 P 位于 A、B 两点之间，那么变换后的 P'位于变换后的 A 和 B 之间。

b. 请证明透视变换生成的点 P'确实具有上述性质。

c. 请证明通过视点的每个平面都被映射为平行于 z 坐标轴的平面。

d. 请证明垂直于 z 坐标轴的平面映射后依然垂直于 z 坐标轴。

e. 请证明透视变换保持深度的相对关系。

练习 7.4.9　变换后的视景体的细节

证明变形后的视景体的尺寸可以通过先前练习中的五个点给出。你可以使用先前练习中证明的结论。

练习 7.4.10　证明投影矩阵的最终形式

投影矩阵基本上就是等式(7.10)的形式，但是还需要加上平移和伸缩变换。如果用 M 表示等式(7.10)中的矩阵，T 表示平移矩阵，S 表示伸缩矩阵，请证明矩阵的乘积 STM 就是等式(7.13)中所给出的。

练习 7.4.11 视点后面的点如何变换?

如果透视变换把视点变到负无穷远处,那么原先位于视点后面的点会如何变换?请考虑这样一条直线 $P(t)$,当 $t=0$ 时它的起点位于视点之前,当 $t=1$ 时直线位于视点之后。

a. 找出它的齐次坐标形式

b. 经历过透视变换之后,写出它的参数形式

c. 用几何的观点对上述情形进行解释。请特别指出第四个分量的几何含义。【Blinn78】中有关于此现象的重要论述。

7.5 生成立体视图

我们先简单地旁述一下如何使用照相机得到一个场景的立体视图。立体视图可以使图像表达得更加明晰;当它被正确地观看时,可以让观察者对图像产生深度感,这样不仅真实有趣,更重要的是可以减少图像的视觉歧义(比如哪些线段是在另一些线段的前方)。本书所有的立体图像都是利用下面的方法生成的。

到目前为止,我们所说的照相机也许更应该称为"独眼"照相机;在希腊神话中 Poseidon 的儿子 Polyphemus 就是一位独眼怪人。如果要想体验一下这种照相机的局限性,你可以闭上一只眼睛在房间里来回走动并且试着做一些简单的工作。人们与生俱来的眼-脑立体感系统可以提供大量的视觉深度信息。我们希望让计算机产生的图形图像也获得此种能力。

要想制作立体视图,一左一右的两幅图像要分别用略有差异的照相机生成,如图 7.34 所示。所有照相机的 lookAt 点都是同一个,但是视点位置各不相同。在图 7.34(b)中,显示器里并排显示了两个视口。左图像在左侧的视口,右图像在右侧的视口。

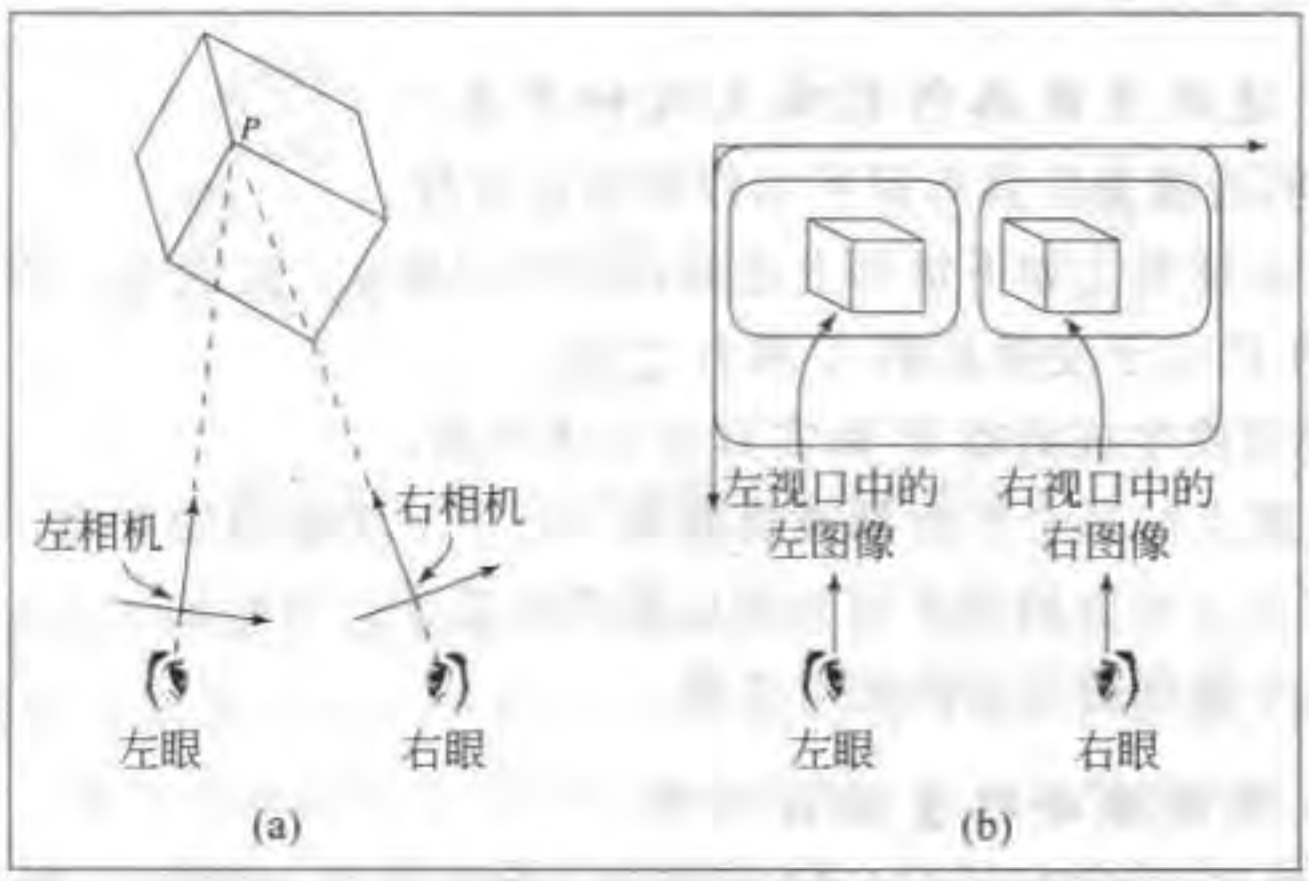


图 7.34 生成立体视图

要想观看立体视图,用你的左眼盯住左图像而右眼盯住右图像。当你方法得当的话,两幅图像就会融为一体并且产生深度信息。这也许需要做一定的练习(前言中描述了一种练习的方法)。

图 7.35 展示了第 6 章提及的 Bucky 球的立体线框视图。两幅图像有很明显的不同,并

且如果只观察其中的某一幅图像就会产生明显的视觉歧义(哪些边在前,哪些边在后?)。但是立体视图就可以消除了这些边的歧义,使得图像表达得简单明了。

图 7.36 是仓房的立体视图。图 7.36(a)中的照相机被转动了 40°。注意到如果没有了立体效果,就很难判别出仓房的朝向。图 7.36(b)是仓房一个角落的特写,可以明显地看出其严重的透视失真。

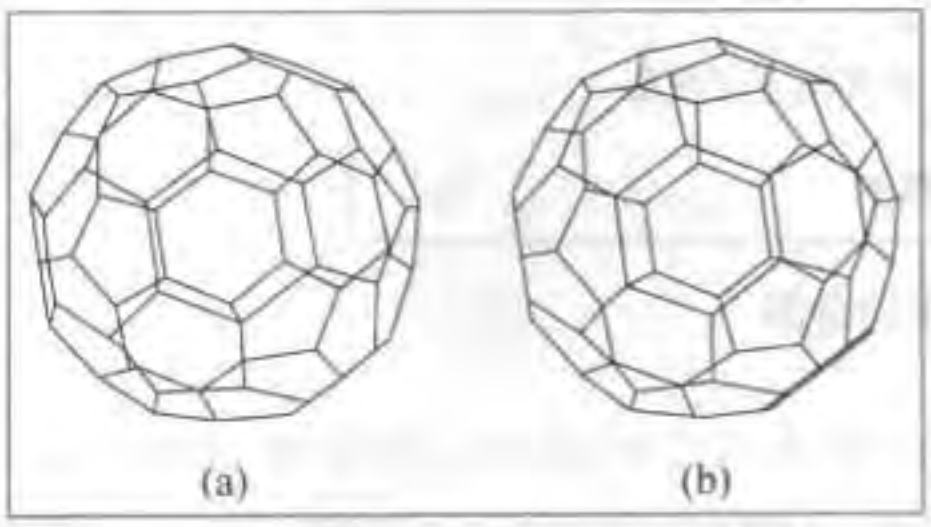


图 7.35 Bucky 球的立体视图

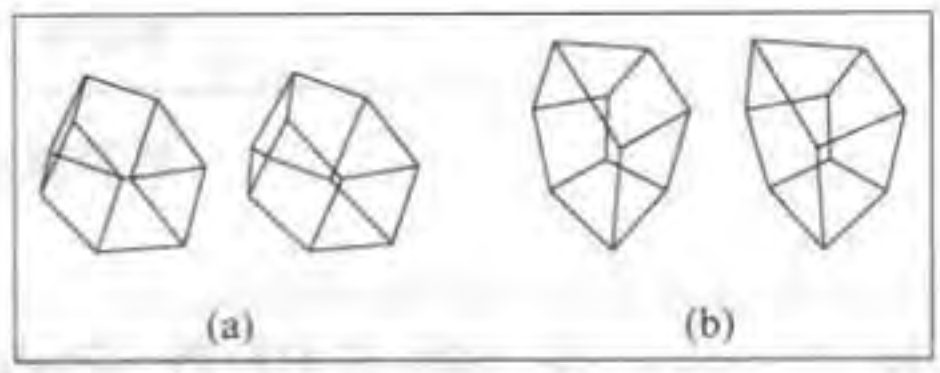


图 7.36 仓房视图特写

为了放置好两台照相机,我们必须要决定左右视点的位置。一个简单的方法是先基于单个 LookAt 点和一个初始视点放置好一台照相机,如图 7.37 所示。选择好 **up** 向量,这就构造好一台"独眼"照相机,它具有三个方向**u**、**v**、**n**。

在初始视点处,沿着 $-\mathbf{u}$ 和 **u** 方向分别偏移距离 D 放置左右视点。D 的选择取决于具体应用中的度量单位。如果所有的长度和距离都用英寸表示,那么用户就应该相应地把照相机放在距离 LookAt 点的若干英寸处。人类两眼大概的距离是 3 英寸,所以 D 最好设为 1.5。如果使用"米"作为度量单位,那么两眼间的距离也应该用"米"来表示。对于一些情形,它的场景比较虚幻怪异,并且不具有内在的尺度,那么为了获得立体效果就要进行一些实验。案例 7.2 是一个生成立体视图的研究项目。

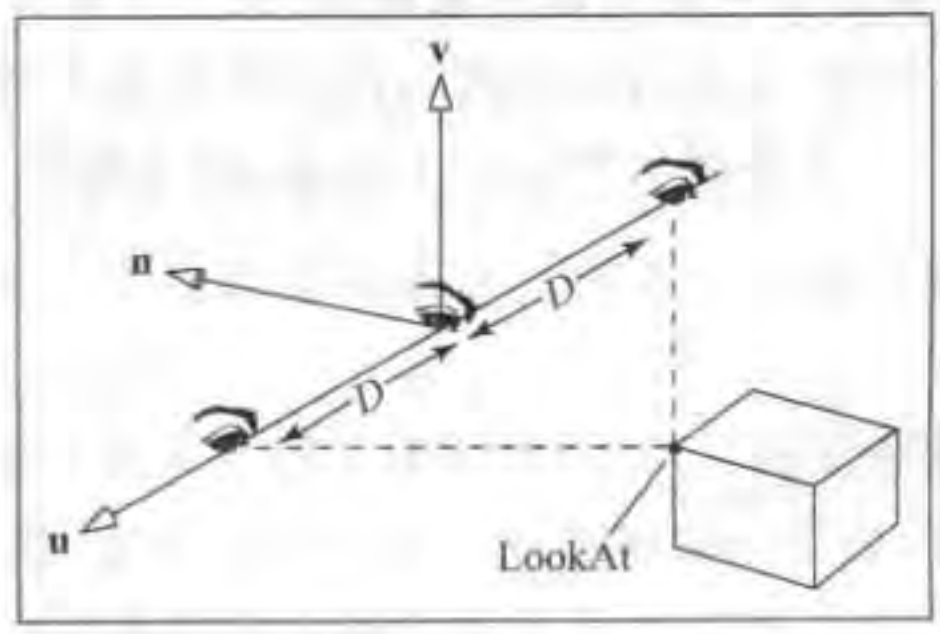


图 7.37 为生成立体视图而放置两个视点

7.6 投影的分类

我们已经探讨过平面投影的一些基本思想,它是把所有的点以某种方式投射到一张平面上。我们在第 5 章看到了平行投影,而在本章又看到了透视投影。在艺术、建筑以及工程绘图领域要涉及到许多投影的特例,下面我们分析一下它们的特点,并且看一看它们之间的关系。

如图 7.38 可以很自然地把平面投影组织成树型结构。在这棵树中,投影类型的每一个子节点都是其父节点的一种特例。在平行投影与透视投影之间产生了第一层重要的分类。我们先来探讨透视投影的各种类型。

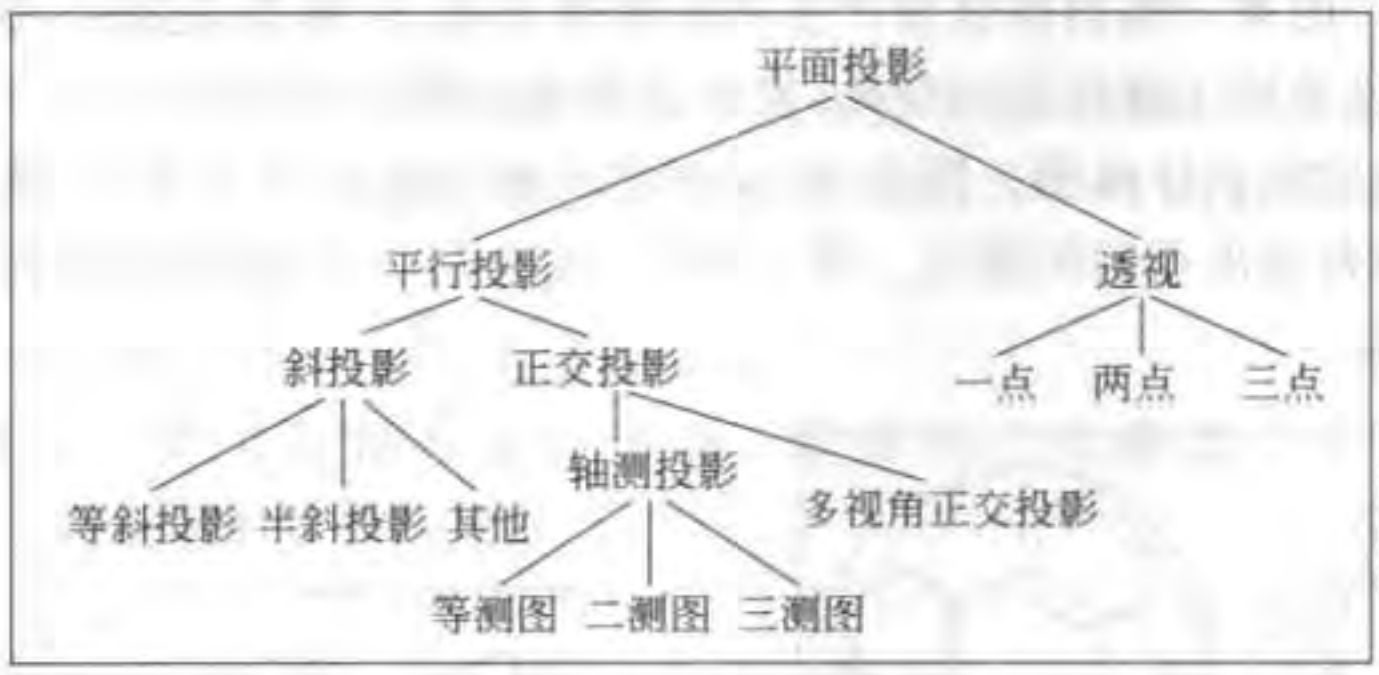


图 7.38 常用投影的分类

7.6.1 一点、两点以及三点透视

透视投影恰好可以分成三类：一点、两点以及三点透视。分类依据的是照相机的朝向与世界坐标系的相对关系。名字来源于观察图 7.39 中单位立方体的情形。这个单位立方体紧靠坐标系的第一象限，并且一个角点位于原点处。更重要的是，它的边与世界坐标系的轴对齐，在本讨论中我们称世界坐标系的轴为**主轴**。主轴的单位方向向量是 **i**、**j** 和 **k**。同样，三个平面 $x=0$、$y=0$ 以及 $z=0$ 被称为**主平面**，立方体的六个面与它们是对齐的。

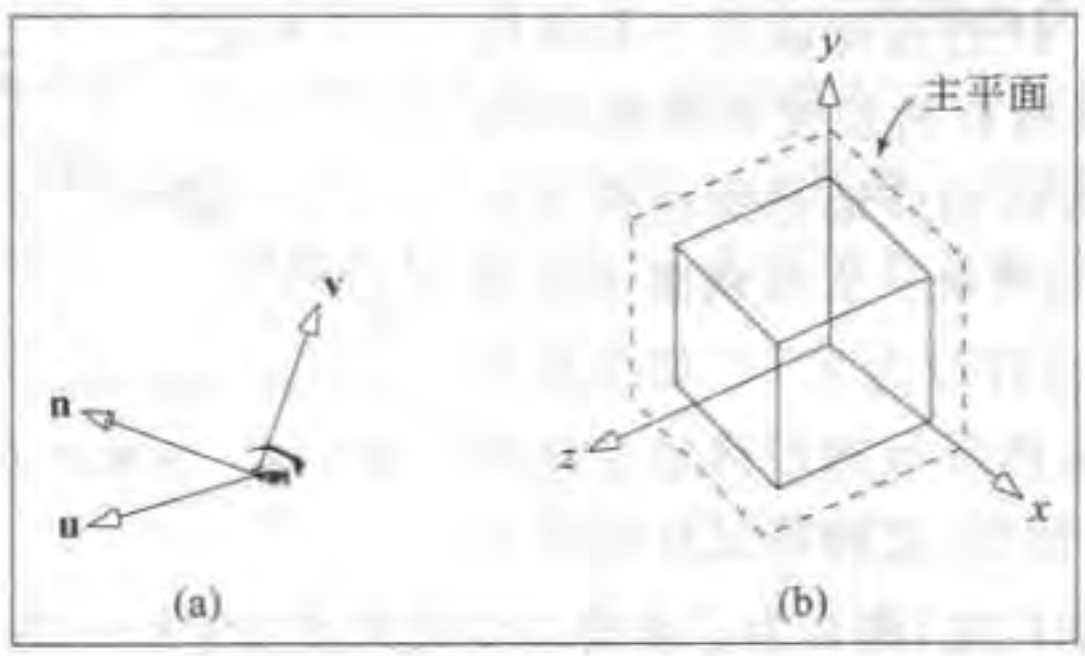


图 7.39 单位立方体、主轴以及主平面

相对于坐标系，照相机可以有无数种朝向方式。对于其中的一些情形，照相机的 $\boldsymbol{n}$ 轴与某个主轴是垂直的。通常，对透视投影的分类可以依据主轴所产生的**有限远处的灭点**数量。回想一下，当某条直线与 **n** 垂直时，其灭点在无穷远处；否则其灭点在有限远处。所以我们同样也可以计算与 **n 不垂直**的主轴数量。这其实也是**穿透**照相机的视平面的主轴数量。(为什么?)

1. **一点透视**：只有一条主轴产生有限远处的灭点。也就是 **n** 只与 **i**、**j**、**k** 三个方向之中的一个不垂直。但由于 **n** 与其他两个方向垂直，所以它垂直于一张主平面。n_x、n_y、n_z 三个分量中的两个必须为 0。

图 7.40(a)是一张一点透视图，其中的照相机视平面与 xy 坐标平面平行。立方体的几条延伸线在有限远处汇聚成一个灭点。这里，照相机的 **n**=(0,0,1)。对于照相机坐标系，延伸线的方向 **c**=(0,0,−1)，所以通过等式(7.6)可得灭点位于(0,0)。另一方面，与 x 轴

和 y 轴平行的直线其灭点位于无穷远处。

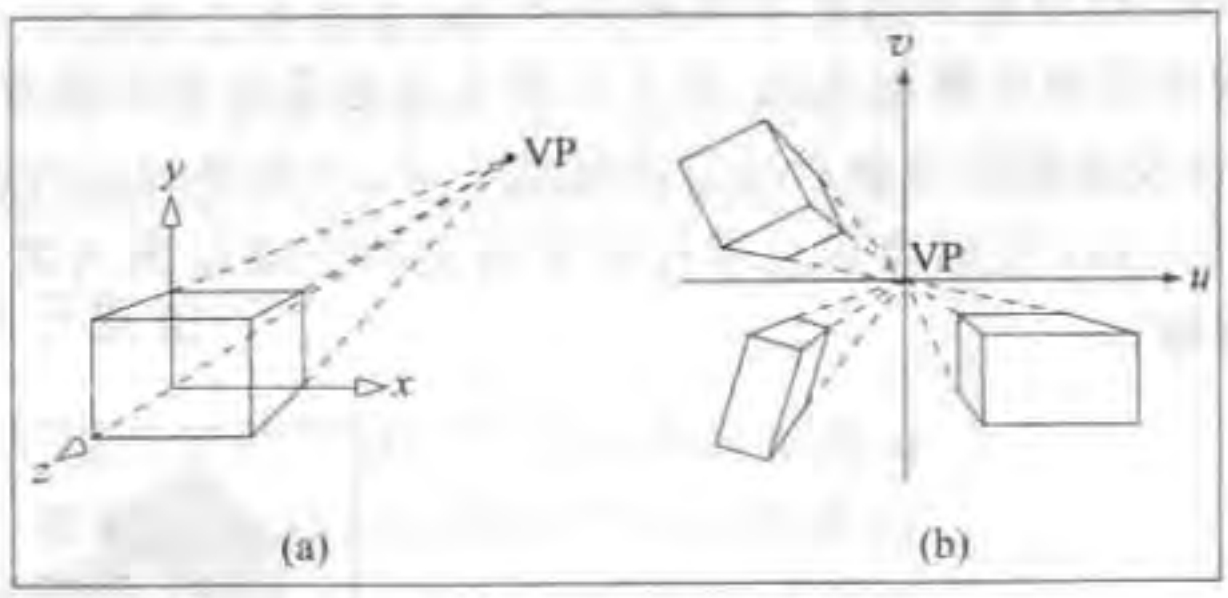


图 7.40 一点透视图

灭点的位置与照相机相对于立方体的位置无关。图 7.40(b)是用一点透视法画出的若干块积木。每一块积木都有其自身的主轴，并且在图中每一块积木的前端面与视平面平行。所有的延伸线汇聚成同一个灭点(0，0)。

重新观察图 7.24 中水平面上的两套网格线。网格线平行于主轴(世界坐标系的坐标轴)。还有另一套垂直的网格线并没有画出来，它与世界坐标系的 y 坐标轴平行。这张图看上去像是一点透视图，因为在水平面上只有一个灭点。但是，如果照相机是向下瞄准的，那么它就是我们将要讨论的两点透视。单从这张图你无法区分到底是哪一种透视(如果你被告知水平面被投影到 $y=0$，那么就可以肯定照相机是水平放置的并且为一点透视)。

2. **两点透视**：恰好两个主轴的灭点在有限远处。因此照相机的 **n** 方向不与这两个轴垂直；它只垂直于另一个轴。**n** 的三个分量中有一个必须为 0。

图 7.41(a)是一个立方体的两点透视图：由于轴 **i** 和 **k** 穿透了视平面，所以有两个有限远处的灭点。照相机按照图 7.41(b)所示的位置放置，它的 **n** 方向与 z 坐标轴成 θ 度角，因此可知 $\mathbf{n}=(\sin(\theta),\cos(\theta))$。这里的 **n** 垂直于 **j**，所以垂直方向的主轴其灭点位于无穷远处。不难计算出有限远处的灭点位置(参见练习)。

如果用两点透视表达图 7.24 中初次见到的网格线场景会是一件很有意思的事情。在图 7.42 中，视点依然位于 $y=1$ 处，其朝向为水平方向，但是照相机向左转动了一定角度使得 $\mathbf{n}=(0.74,0,0.67)$。现在通过在水平面上把那两套直线进行延伸，可以得到水平面上的两个灭点，并且两个灭点之间间隔很远。(如何数值计算出灭点?)由于远处的线相互交织在一起十分拥挤，为了清晰起见这里并没有绘出。

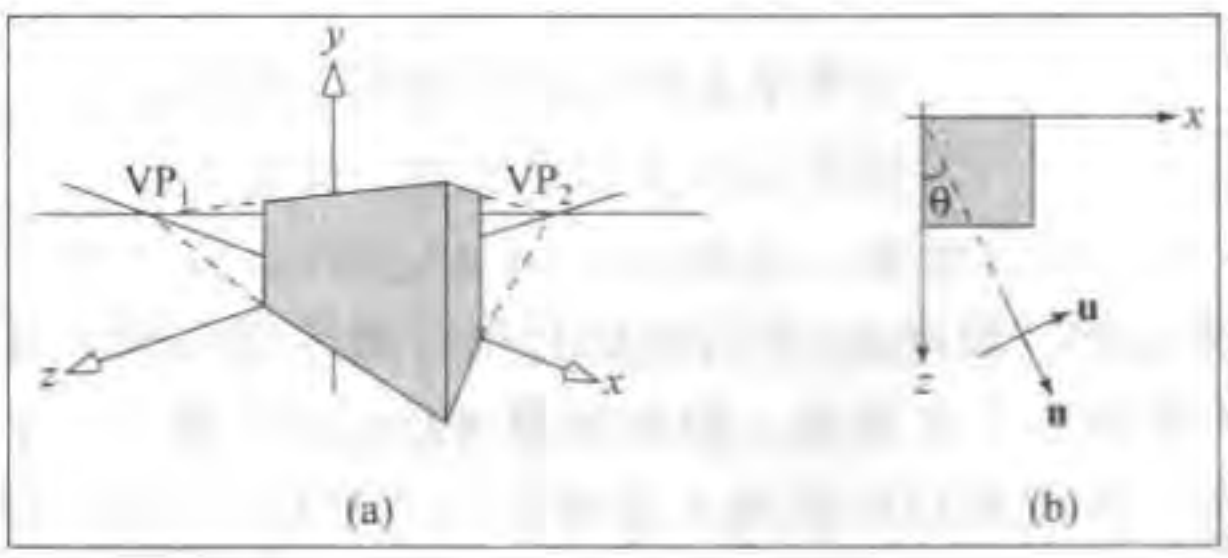


图 7.41 两点透视图

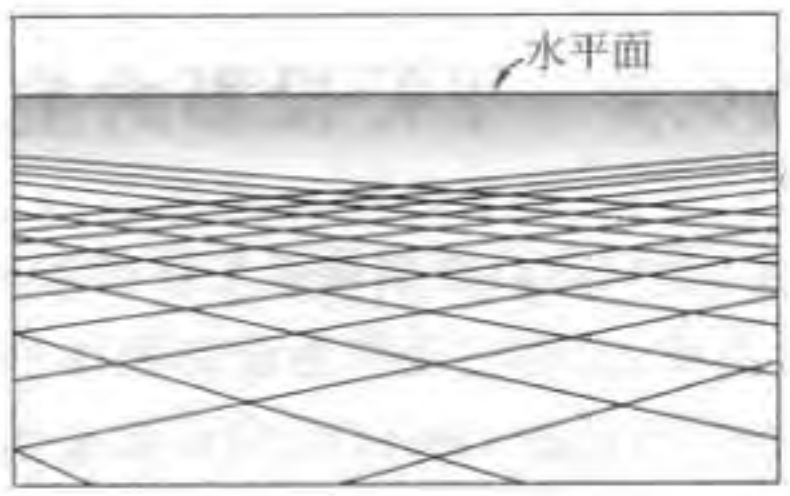


图 7.42 无穷网格线的两点透视图

3. **三点透视**：所有的三个主轴其灭点都位于有限远处：三个主轴都穿透视平面。由于 **n** 不与任何一个轴垂直，所以它所有的分量均不为 0。

艺术家经常在透视绘画中使用灭点，它可以突出表达某些特征或者增加戏剧般的效果。图 7.44 中用三点透视表达的建筑场景可以说明这一点。(画中的三个灭点位于何处?)著名的荷兰艺术家 M. C. Escher 凭借奇异且带有视觉错觉的三维绘画闻名于世，他创作的一些作品就使用了三点透视。

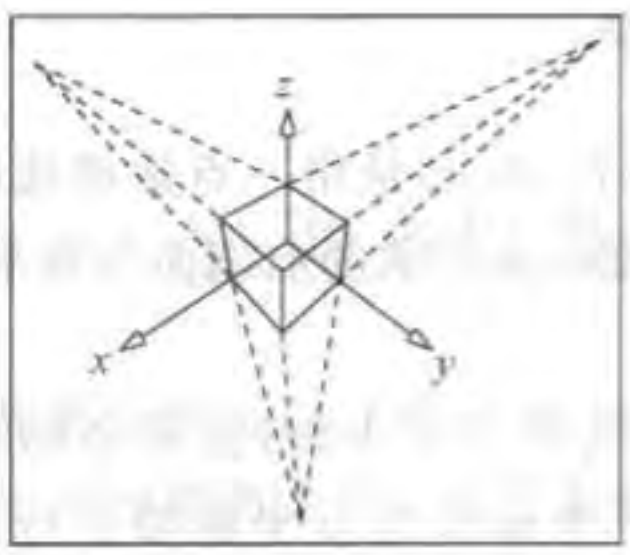


图 7.43 三点透视

图 7.44 三点透视范例

练习

练习 7.6.1 生成一点透视

在应用程序中，为放置理想的照相机我们需要选择好 eye 和 look 的值，此时 **n** 就是 eye−look。选择三套 eye 和 look 值，生成单位立方体的一点透视图。

练习 7.6.2 灭点的位置

用手绘出单位立方体和世界坐标系轴的一点透视图，其中 eye=(2, 3, 4)。

练习 7.6.3 计算灭点

在图 7.41(a)中，对于任意的值 θ 计算图中两个灭点的位置。

练习 7.6.4 生成两点透视

请选择三套照相机的 eye 和 look 值，然后分别生成单位立方体的有趣的两点透视图。对于每个例子请计算灭点的实际位置。

7.6.2 平行投影的类型

下面我们继续研究平行投影的分类。到目前为止，我们仅仅在第 5 章简要地提及过平行投影，那里它是作为绘制物体的一种简单方式出现的。利用函数 glOrtho()建立一个视景体，然后让 OpenGL 完成其余的工作。现在我们稍微深入地研究一下平行投影的性质，并且区分出日常使用中平行投影的各种类型。

我们知道在透视投影中场景里所有的点 P 都沿着**投影线**投射到视平面上，各条投影线

汇聚于视点位置：每条投影线由点 P 自身和视点决定，所以有各种各样的投影方向。相反，作为平行投影，它的所有投影线都是同一个方向，不妨设为 $\mathbf{d}$。图 7.45 展示了两个将要被投影的点。点 P 投影到视平面上的点 p，点 Q 投影到点 q，它们的投影方向都为 $\mathbf{d}$。

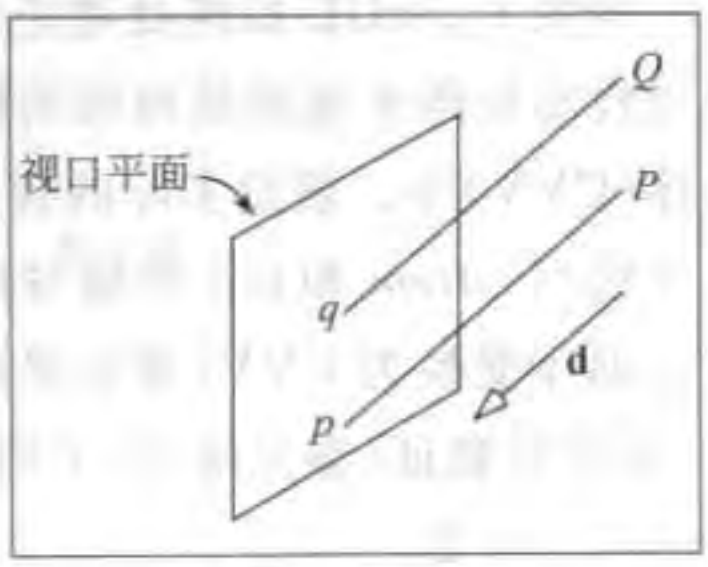


图 7.45　平行投影

平行投影的若干理论

点 P 的投影点 p 位于何处？假设平面的法向量是 $\mathbf{n}$ 并且平面通过点 B。那么由点 P 出发沿着方向 $\mathbf{d}$ 形成的射线与平面的交点就是点 p。先前我们已经多次做过这样的计算。如果射线的参数形式为 $P+\mathbf{d}t$，那么把它代入平面方程 $\mathbf{n}\cdot(P-B)=0$，然后求出 t。然后再把这个 t 值代回 $P+\mathbf{d}t$ 得到交点：

$$p = P + \mathbf{d}\frac{\mathbf{n}\cdot(B-P)}{\mathbf{n}\cdot\mathbf{d}} \tag{7.16}$$

这与我们在透视投影中得到的表达式有很大的不同；特别的，它不会因距离而产生透视缩短的效果：分母里没有出现包含点 P 的项（另外请注意可以用 $-\mathbf{d}$ 替换 $\mathbf{d}$ 而不影响得到的投影结果）。

我们专门使用熟悉的照相机模型来看一看上述等式到底可以揭示出什么信息。假设我们被置于照相机坐标系中，所以 $\mathbf{n}=(0,0,1)$，并且投影平面是 xy 坐标平面那么 $B=(0,0,0)$。（这比投影到近平面要稍微简单一些。）因此可以求得点 p（请验证）：

$$p = \left(P_x - d_x\frac{P_z}{d_z}, P_y - d_y\frac{P_z}{d_z}, 0\right) \tag{7.17}$$

平行投影可以划分为两个主要类型，如图 7.46 所示。

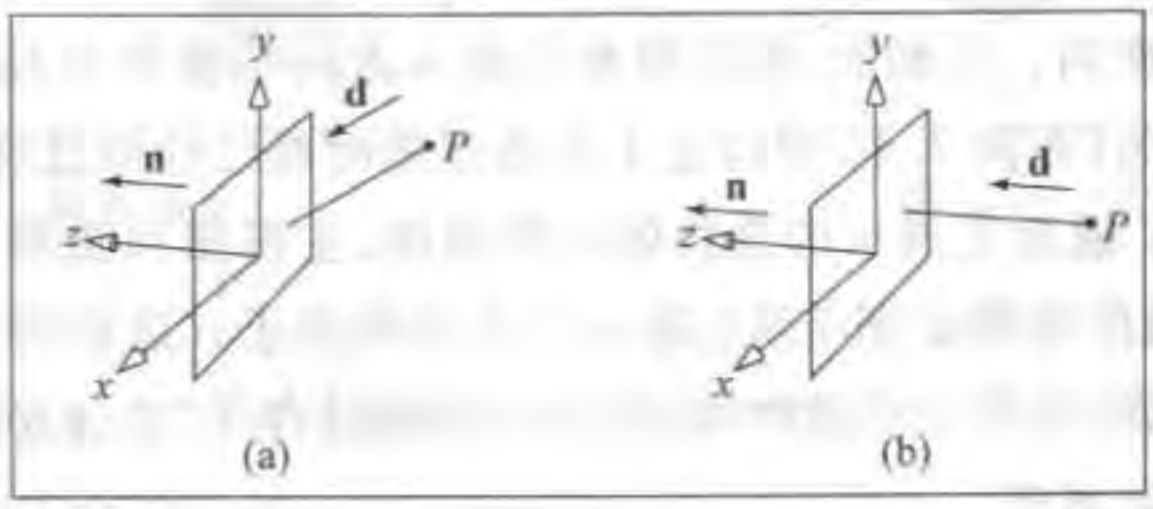


图 7.46　在照相机坐标系中的平行投影

(a) $\mathbf{d}$ 取一般的值；(b) $\mathbf{d}$ 与 $\mathbf{n}$ 平行

1. 斜投影：投影方向 $\mathbf{d}$ 与 $\mathbf{n}$ 不平行。
2. 正交投影：投影方向 $\mathbf{d}$ 与 $\mathbf{n}$ 平行。

图 7.46(a)中的方向 $\mathbf{d}$ 取的是一般情形并不平行于 $\mathbf{n}$（当然我们不会让 d_z 等于 0）。图 7.46(b)是 $\mathbf{d}$ 平行于 $\mathbf{n}$ 的情形，所以这里的 d_x 和 d_y 等于 0。下面我们仅讨论正交投影，因为在日常应用中斜投影并不经常使用。

正交投影

在这种情况下 d_x 和 d_y 都等于 0。那么此时的 $p=(P_x,P_y,0)$。因此正交投影无非就

是抹去点 P 在照相机坐标系下的第三个坐标分量。

研究 OpenGL 如何处理正交投影是一件很有意思的事情。回忆以前的内容,OpenGL 使用投影矩阵来描述照相机的视景体。特别地,投影矩阵指明了如何把顶点变换到正则视景体(CVV)中。假设实际的视景体在 x 轴方向从 l 到 r($left$ 和 $right$ 的缩写),在 y 轴方向从 b 到 t($bottom$ 和 top 的缩写),在 z 轴方向从 n 到 f($near$ 和 far 的缩写)。为了把这个平行六面体变换为 CVV,我们必须进行平移和放缩操作使得 CVV 在每个维度上都从 -1 到 1。很容易验证(参见练习)下列矩阵

$$\begin{bmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \text{(用于正交投影的 OpenGL 投影矩阵)} \quad (7.18)$$

可以把上述视景体变换为 CVV。此矩阵可以由函数 glOrtho()生成。如果你用它乘上点 P(表示成齐次坐标形式),事实上就得到了点 p,OpenGL 会对它进行裁剪测试并且映射到视口。请注意 OpenGL 并不像我们前面讲到的那样把 z 分量设置为 0。相反,它进行的是变换操作而非投影。最终它还是会进行真正的投影操作,分离出 z 分量,使用 x 和 y 分量在屏幕坐标系上进行绘制,z 分量的作用是进行深度测试。

正交投影的分类

在前面的分类图表中我们可以看到正交投影有很多类型。不同类型的分类依据是照相机在世界坐标系内的朝向。特别的,根据照相机的 **n** 方向与世界坐标系中坐标轴的关系可以给各种类型进行命名(在图 7.40 中讨论 k 点透视的时候已经做过这种尝试)。

多视角正交投影:通常它展示的是物体的俯视图、主视图和侧视图。**n** 分别于 **k**、**i** 和 **j** 平行,然后绘制出想要的物体。图 7.47 是一个展示的例子。这种视图十分适合于工程制图,因为你可以直接从绘图纸上测量物体的尺寸,尤其适合于"立方形"的物体,因为它的面可以与世界坐标系的轴对齐。

轴测视图:在轴测视图中,为了使物体形状的 3D 感表达得更强,要对 **n** 进行一定的选择。通常在此种视图中,**n** 并不与任何主轴平行;更确切地说,要调整好朝向使得(立方形)物体三个相邻面都可见。选择物体中的哪些面作为可见面取决于面的重要程度。除此之外,其中的一个主轴通常是垂直方向的。物体中的平行线在视图中当然依然平行,只不过线会有一定程度的**缩短**。在图 7.48 中 **n** 与 x 坐标轴成 α 角度。

与视平面的法线成 α 弧度的单位线段在视图中的长度为 $\sin(\alpha)$,所以可知**缩短因子**等于 $\sin(\alpha)$。

依据有多少个主轴被同等程度地缩短,可以把轴测图分为三类:

(1) 等测图("同等程度"):所有的主轴都被同等程度地缩短。

(2) 二测图("两种程度"):有两个主轴被同等程度地缩短。

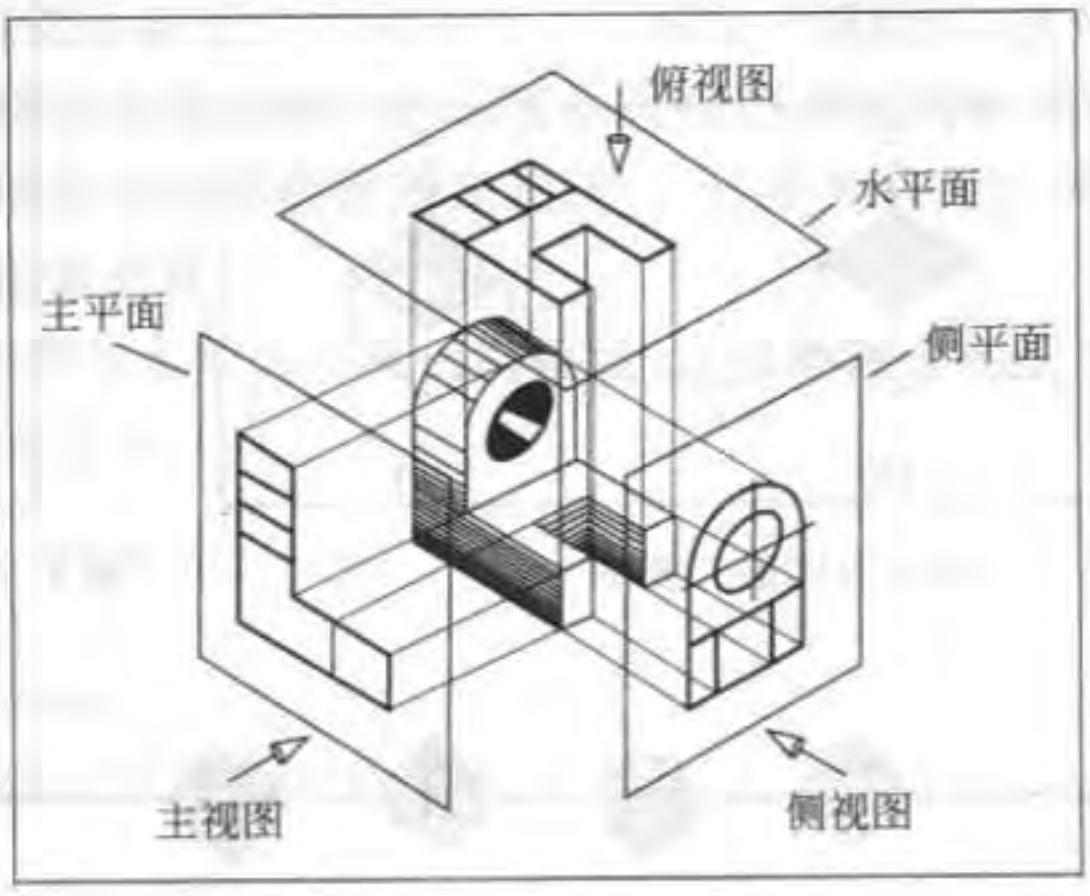


图 7.47　多视角正交投影

(3) 三测图("三个程度")：三个主轴的缩短程度各不相同。

等测图：图 7.49(a)是立方体的一个等测图，它是沿着立方体八个对角线方向中的一个观察得到的。所有的三个主轴被同等程度地缩短。从图 7.49(b)可以看出透明立方体的等测图是一个包含三条对角线的正则六边形。

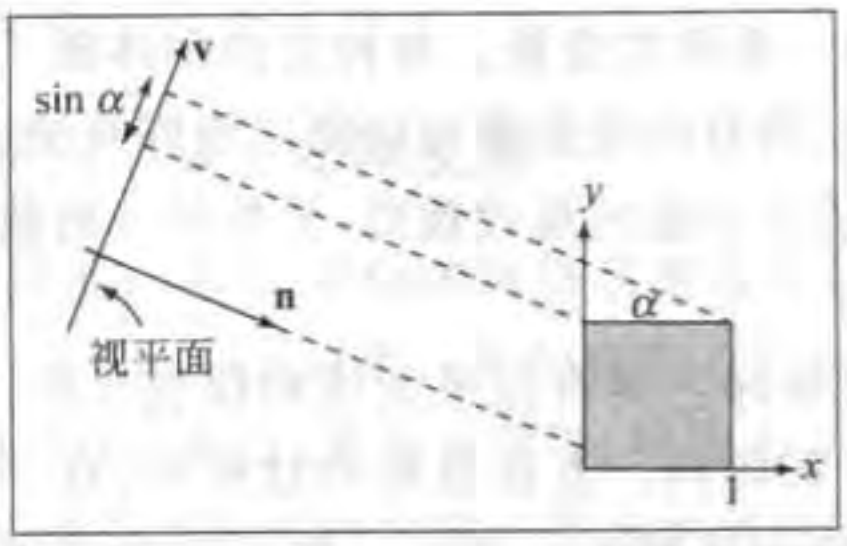


图 7.48　缩短因子

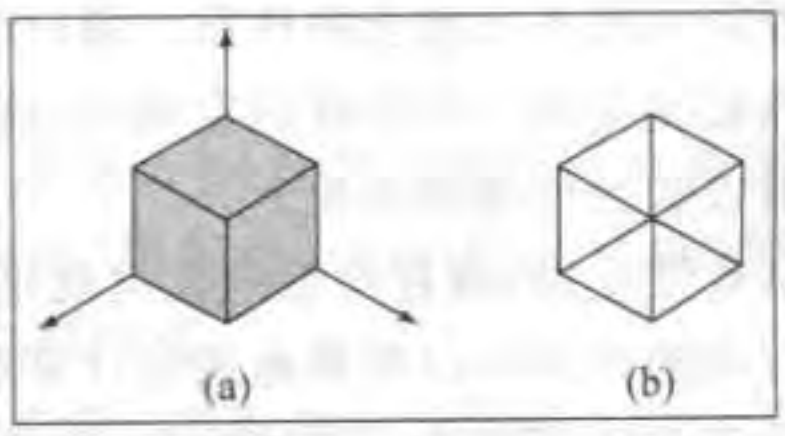


图 7.49　立方体的等测图

获得等测图的条件是 $n_x = \pm n_y = \pm n_z$。在等测图中主轴投影之间的夹角都相等。主平面内线段的长度可以从绘图中直接测量得到，把这个长度乘上一个固定的值(是多少?)就可以获得那条线段的真实长度。

二测图：当只有两条轴与 **n** 夹相同角度时，那么得到的视图为**二测图**。对于二测图，必须有两个方向余弦相同，因此有 $n_x = \pm n_y$，$n_x = \pm n_z$ 或 $n_y = \pm n_z$。图 7.50 展示了一系列的二测图；其中的第三个图几乎就是等测图。对于每一个视图都有 $n_x = n_y$，并且取值都是负值(n_z 在第一张图中等于多少？在最后一张图中又等于多少?)。不同的朝向反映了各种面不同的要强调的程度。

三测图：最后，如果三个主轴与 **n** 的夹角各不相同，那个生成的视图为**三测图**。图 7.51 是三测图的一个例子，其中 **n** 的分量在选择上几乎是任意的。当朝向选择恰当的话，三测图看上去会十分自然。

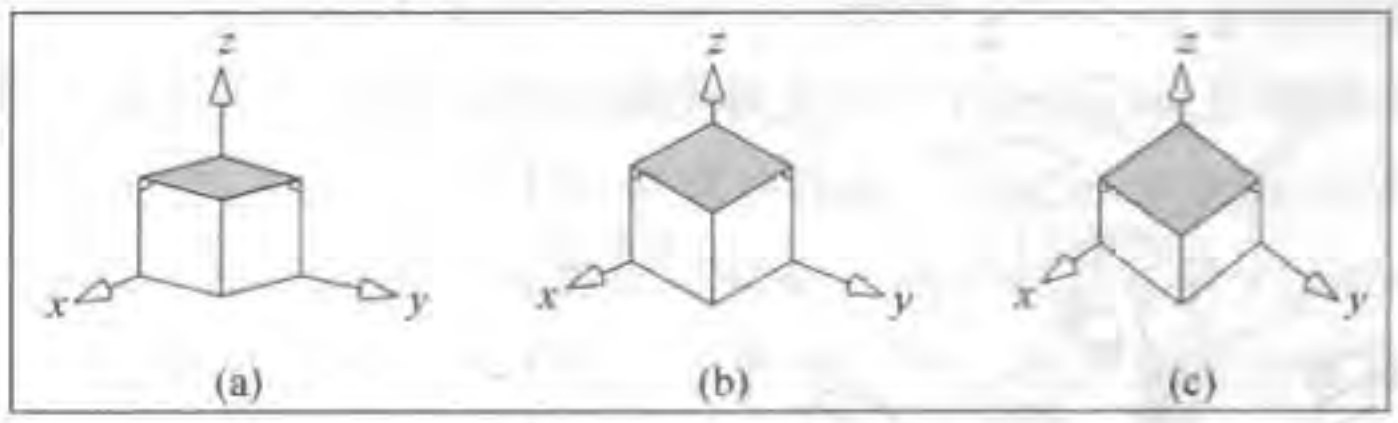


图 7.50 一些立方体的二测图

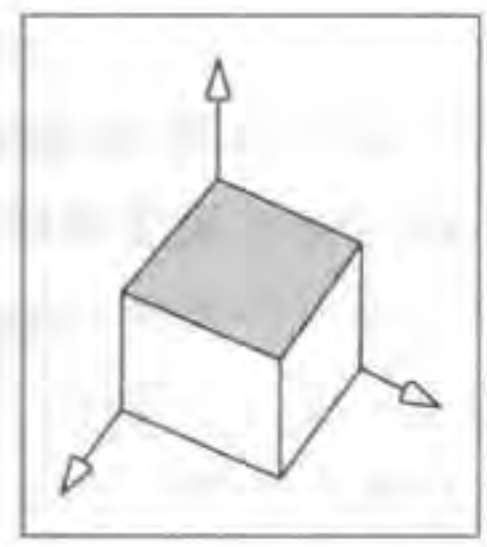


图 7.50 一些立方体的二测图

本章小结

本章中我们学习了如何在程序中定义并控制一个“观察系统”,这样我们就能够生成从不同视点观察 3D 场景所得到的图像,并且还可以在动画中让照相机游走于场景之中。一个视点外加一个视景体就建模构造出一台照相机,它拥有自己的坐标系。一般地,它与针孔照相机很相像,把点投影到一张视平面上。视景体内的点被投影到视平面,而落在视景体之外的点则被裁剪掉。

观察过程要用到的许多数学知识都牵扯到点在不同空间中的几何变换问题。OpenGL 所使用的图形绘制管道对进入绘制管道的顶点实施一系列的变换。每种变换都体现了照相机和投影机制各自独立的性质。通过使用齐次坐标,所有的变换都可以使用矩阵乘法完成。最终点被映射到一个特殊的空间中,这个空间十分适合于边的高效裁剪,另外单一的裁剪算法就可以满足任何照相机。

从本质上讲,透视投影的几何就是某种相似性,幸好它具有保平面性和保包含性,所以整条直线的投影可以简单地通过计算线上两点的投影得到。点在投影的过程中,有一个变换要使用点的深度进行除法操作,这样就使得远处的物体看上去更小一些。因为要用到空间坐标除以一个公共除数这样一个除法操作,所以这种变换就十分适合于采用齐次坐标形式,这里的除数就是第四个分量。于是使用矩阵乘法后面再跟一个除法操作就可以实现这个变换,并且一个投影与另一个投影之间不同的几何细节可以通过矩阵的参数辨别出来。

现在整条图形绘制管道就可以表达为:先进行一次矩阵乘法,接着是裁剪操作,然后进行透视除法,最后通过一次矩阵乘法变换到屏幕坐标系。第一个矩阵通常分成 modelview 矩阵和投影矩阵两部分,这样就可以单独指定出照相机的位置和朝向而其视景体无关。我们前面已经看到 modelview 矩阵包含两个变换:一个变换是把场景中的物体变换到合适的位置和尺寸,另一个变换反映了场景中的照相机位置和朝向。

案例分析

案例 7.1 使照相机在场景中进行游走

(难度:Ⅱ)编写一个允许照相机在场景中进行游走的程序。用户使用键盘来控制照相

机。每按一次键，照相机就沿着三个方向中的一个移动一段距离，或是绕着某一个轴旋转一定角度（与图 7.13 中调用的函数类似），然后场景就在照相机的新视点处重新绘制一次（你来决定哪些键用来移动照相机而哪些键用来旋转）。还要允许用户使用其他的一些键来改变可视角度以及照相机的宽高比。

如果你像图 7.24 那样在水平面上画出**网格线**，这样绘制出的图形就更易于理解。这很容易通过下列的循环语句实现：

```
for(int x=-100; x<100; x++)
{
    glBegin(GL_LINES);
      glVertex3d(x,0,100);glVertex3d(x,0,-100);
    glEnd();
}
```

它绘制了 200 条 x 坐标线。也可以类似绘制出 z 坐标线。增加一个控制键使得用户可以选择网格线的显示与隐藏。

OpenGL 提供了许多图元用来构建有趣的场景。请使用第 5 章所介绍的一些物体，例如球面、锥面、十二面体和茶壶，构造生成这样的场景。请绘制出它们的线框图。稍后我们还会讨论如何绘制它们的实体图，这需要建立好光源并且给物体附上材质。请参考第 5 章中的例子。

案例 7.2　立体视图

（难度：Ⅱ）扩展你的网格观察程序使得它可以生成图 7.34 描述的物体立体视图。要构造三个 Camera 对象。先用标准方法构造出一个“独眼”照相机，然后使用“独眼”照相机的相关信息（视点、方向 **u** 等等）构造出左视点和右视点照相机。

并排在屏幕的两个视口里显示照相机生成的相应图像。要锻炼你的眼睛可以分别盯住两幅图像，这样才能获得立体效果。

还要允许用户能够在场景中移动左右照相机（两台照相机是固定在一起的）获得更佳视图。请试着找出两台照相机之间的最佳距离，这样观看近处的和远处的物体都可以获得十分自然的效果。

案例 7.3　可以极大提升效率的背面剔除操作

（难度：Ⅱ）当我们渲染一个网格物体时，它的每一个面片都被输送到图形绘制管道，其中的顶点要经历变换操作，最后面片被绘制出来。但是对于物体的任意一种朝向，都存在许多背离观察方向的面片。它们被称为**背面**，并且没有必要被输送到图形绘制管道，因为它们都是不可见的：会存在比它们更接近视点的面片把它们挡住。图 7.52(a)是经典的仓房模型。对于任意一个照相机位置，仓房的面片中都存在背面：图中标出了两个。

我们说一个面片是背面，如果它是背离视点方向的。这也就是说它的外法向方向 **m** 与视点与面片任意一点 P 所成向量的夹角小于 90°（参见图 7.52(b)）。若两个向量的夹角小于 90°，那么它们之间的点积为正。所以针对一个面片是否是背面可以有如下的测试：

F 是一个背面，如果 $(P-eye)\cdot\mathbf{m}>0$

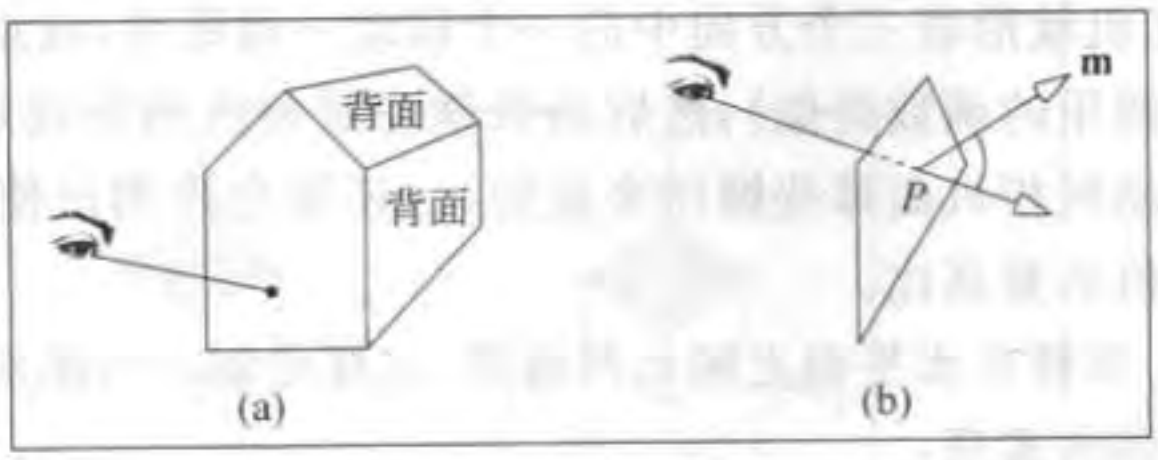


图 7.52

(a) 仓房的背面;(b) 背面的定义

这个测试的开销并不大。所以可以先测试一个面片是否是背面,如果是就将其丢弃,这样就可以显著提升绘制复杂网格的速度。

修改图 6.15 中 Mesh::draw()的代码,让它只绘制网格物体的前面,跳过那些被测试为背面的面片,这可以通过插入语句 if(isBackFace(f,...))continue 实现。开发出函数 int isBackFace(f,...)的代码,当 f 是背面时它返回 1,否则返回 0,只把那些必要的参数传递给它。开发一个可以基于任意视点绘制网格的应用程序,比较一下背面剔除法使用与不使用时的速度。

进一步阅读

贯穿整章都提到过,"OpenGL Programming Guide"【Woo04】为建立和控制 3D 图形观察系统提供了一个清晰的介绍。Foley 等著的【Foley03】涉及到了更多的细节。"Jim Blinn's Corner-A Trip Down the Graphics Pipeline"【Blinn96】中的若干文章针对投影、视角变换、裁剪方法提供了清晰有趣的论述。Angel【Angel03】中也含有照相机修正和观察系统方面很好的描述。

第 8 章

几何体的真实感渲染

本章学习目标

- 增加三维场景绘制的真实感
- 探讨光在物体表面的反射性质
- 在有光源的环境下渲染多边形网格
- 探讨如何使多边形网格的渲染结果看上去平滑
- 为物体表面增加纹理
- 为场景中的物体绘制阴影

预览

8.1 节讲述了增加三维物体渲染的真实感的必要性。

8.2 节介绍了计算机图形学中几种不同的着色(或者光照)模型,以及计算环境光、漫反射和高光对物体最终颜色贡献的工具。另外这节还讲述了在 OpenGL 中,如何建立光源,如何描述面片的材质属性,和渲染多边形网格时 OpenGL 图形管线如何运行。

8.3 节侧重于渲染多边形网格表示的三维模型。介绍了平面着色、Gouraud 着色和 Phong 着色方法。8.4 节介绍了一种简单的基于深度缓存的隐藏面消隐技术,这也是 OpenGL 所采用的方法。适当的隐藏面消隐能够显著地增加结果的真实感。

8.5 节介绍了在物体表面绘制纹理(比如图片,剪切画或者标签)的方法。通过绘制纹理能够使得物体看上去就像表面贴上了标签一样,或者让人觉得就是用木头、砖等真实材料制成的一样。本章也会介绍纹理映射的工作流程。8.5.4 节展示了使用 OpenGL 为物体添加纹理的一个完整程序。8.5.6 节讨论了在反光的曲面上映射纹理,使得物体看上去就像置身于所在场景中一样,基于这种原理的很多方法能够使得三维场景看上去更加真实。

8.6 节介绍了阴影的绘制技术,并且和 OpenGL 所用(功能很弱)的方法进行了比较。8.7 节介绍了凹凸贴图,它可以产生物体表面粗糙或者凹凸的不平的效果,从而使得程序员不用花很大力气去修改模型的真实表面数据(比如一个网格数据)来实现粗糙和凹凸不平的效果。

本章结尾有一些案例分析,对以上提到的内容作更深入的介绍,供有兴趣的读者钻研和实现。

8.1 概　　述

在前面的章节中,我们开发了一些工具进行物体的网格建模,通过操作照相机拍摄这些模型生成照片。现在我们想增加一些工具使得这些物体看上去更加有趣和真实。第 5 章的

一些例子使用了一些 OpenGL 函数渲染出闪闪发光的茶壶和有光源场景中的球体,但是没有涉及到这些实现背后的理论依据。这章我们就要介绍**渲染**一幅三维物体图像所涉及的知识。实际上这就是计算图像中每个像素看上去应该是什么颜色的问题。这很大程度上取决于不同的**着色模型**(**shading model**)。着色模型就是试图描述光源发出的光线和场景中的物体是如何相互作用的。因为实现上的限制,我们通常不会模拟有关光散射和反射的全部物理原理,因为这是非常复杂的,而且会导致出现很长的软件开发周期(一个开发过程中最耗时的部分)和运行起来非常慢的算法。但是有一些近似的模型已经被提出,它们能够很好地提供不同级别的真实感效果。

为了介绍涉及的基本问题,我们开始将由浅入深地介绍实现各个级别真实感效果的技术,然后介绍如何把这种技术应用到实际中去,同时看看 OpenGL 是如何替我们完成这些繁重的工作的。

在各种级别的真实感技术中,处于最底端的是**线框**(**wireframe**)渲染。图 8.1 是用线框方法渲染的一幅 540 个立方体混乱摆放的效果图。每个物体只有边被画了出来,同时你的视线能够透过物体看到后面的东西。这让我们几乎不可能看出来这是什么东西。(立体的图像只对我们有一点点帮助)。

图 8.2 没有绘制在面后部而被遮挡的边,使得结果有了明显的改善。我们可以把这种方法称作"线框加隐藏面消隐"技术(真正准确的说法是:可见线框渲染技术)。虽然只有边被画出来,但是每个物体看上去都有立体感,而且能很容易地看出来每个物体从哪开始到哪结束。注意图中的某些边只画出了一部分,因为它们斜插入遮挡面的后面从而被截断。

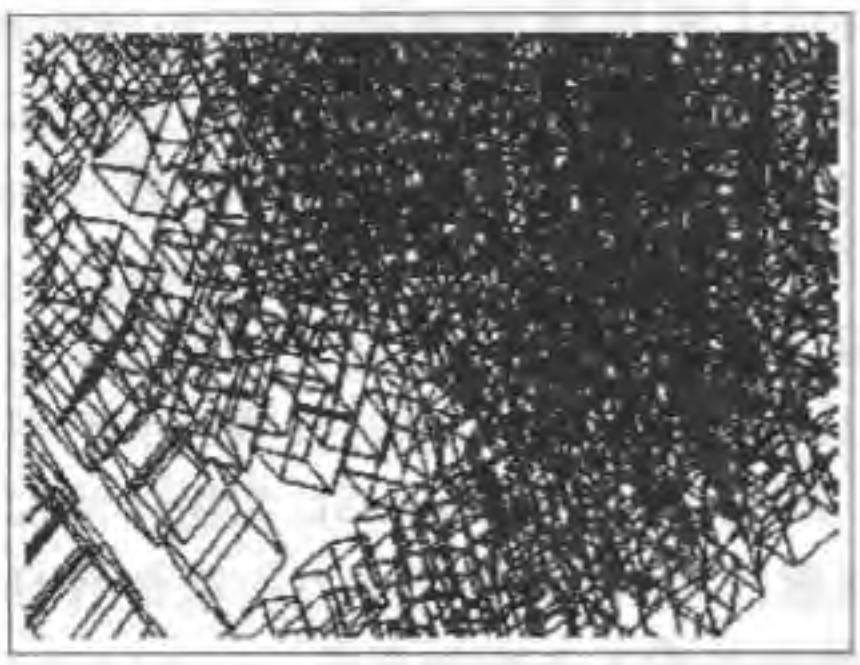


图 8.1 使用线框渲染的场景

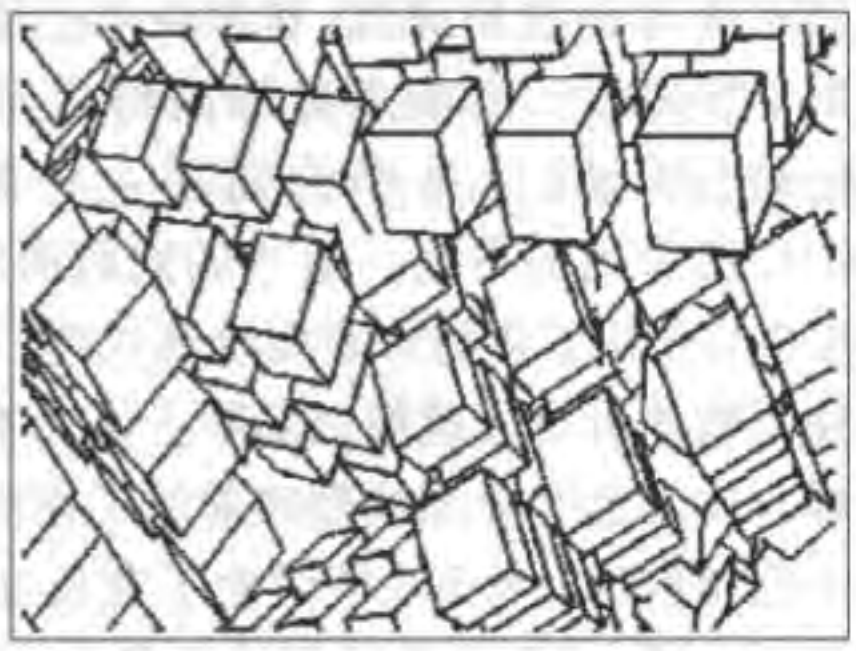


图 8.2 边可见部分的线框渲染

为满足读者的好奇,这里提示一下,这张图是使用 OpenGL 并且打开深度缓存功能绘制出来的——对于每一个立方体,用 drawMesh()画白色的面,再用 drawEdges()画黑色的边。具体的过程我们已经在第 6 章中讨论过。

下一步我们把真实感效果提高一个层次,希望渲染出的物体象是处于有若干光源照明的场景中。根据物体表面性质和光源的位置,每个物体不同的部分反射光的多少也不同。渲染这样的场景需要有一些自动的方法计算每个面上各个部分的颜色,这个方法就是着色模型。

图 8.3 是一个多边形网格场景:权架(见例 5.6.3),在一个圆柱和圆台上的多面球体。图 8.3(a)是**线框**显示的结果,图 8.3(b)是着色模型显示的结果(附加隐藏面消隐)。面向光源的面(光源处于物体的上方)比背向光源的面更亮(请仔细观察这个差异)。这幅图还展示了平着色(flat shading):每个面上反射的光量只根据其上某点计算一次。因此同一个面上

所有点的灰度是一样的。

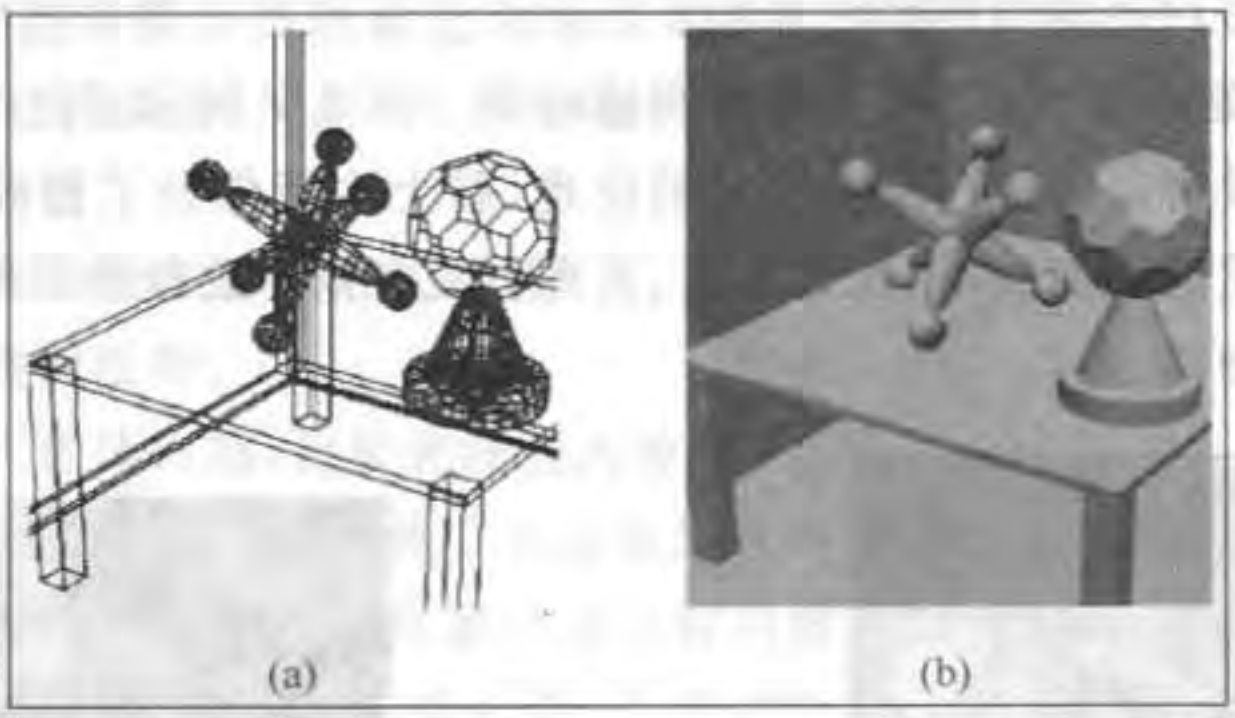


图 8.3 使用着色模型的网格

(a) 线框示图；(b) 平着色

更复杂一些，我们应该体现出颜色的使用。颜色这部分会在 8.2.6 节介绍。

在第 6 章我们讨论过使用网格来逼近平滑曲面的物体。这样的网格模型渲染的结果也应该是平滑的，更像平滑的曲面而不是一个一个的多边形。在图 8.4 中，权架上的球和圆柱是平滑的物体，而多面球体和桌子不是。图 8.3(b)使用的是平面明暗处理，而图 8.4 是同一场景使用**平滑着色**(**smooth shading**)的渲染结果。这里同一个面上不同点的灰度不同是通过插值实现的，这就是 **Gouraud 明暗处理**(**Gouraud shading**)。这样通过不同灰度值使得本来平滑的物体看上去更加平滑(虽然不是完全的平滑)，多边形的边界消失了，从而给我们这种平滑的感觉，而不是感到曲面是由一些小面片组成的。因为多面球体本身就不是一个平滑曲面的物体，所以即使使用了平滑的渲染方法，也会显现出一个个单独的可见表面。我们会在第 8.3 节介绍 Gouraud 明暗处理。

高光(highlight)可以使得物体看上去更加有光泽。图 8.5 展示了加入**镜面反射光**(**specular light**)后的效果。有关高光的讨论放在了 8.2 节。越有光泽的物体应该看上去越明亮，但是越多局部的高光就越会给人一种像塑料材质的感觉。

图 8.4 使用平滑着色的场景

图 8.5 加入镜面反射高光(specular highlight)的效果

阴影是另外一个能够增加图像真实感的渲染效果。图 8.6 展示了在图 8.5 同样的场景

中适当增加了阴影后的渲染效果。其中物体的阴影投射到了与其相邻的其他物体上。我们将在第 8 章的 8.6 节讨论实现阴影的算法。

在物体上增加纹理可以显著地提高真实感效果。图 8.7 展示了把不同纹理绘制到不同表面后的效果。场景背后的墙壁贴上重复的纹理,看上去就像贴了瓷砖的墙一样。使用纹理可以产生不同材质的表面效果,比如木头、大理石或者铜。纹理的另外一个用途是把图片像转印一样包裹在物体上。

图 8.6 带有阴影的场景

图 8.7 映射到表面的纹理

此外还有一些技术能够增加渲染的真实感。在第 12 章我们将更加深入地研究**光线跟踪**(ray tracing)。虽然光线跟踪需要的计算量非常大,但是算法本身易于实现并且能够渲染出许多效果,比如阴影、镜面反射和光线穿透透明物体。

在本章将介绍一些能够渲染出不同级别真实感效果的方法。我们首先从图形学中经典的照明模型开始,它能够使得物体看上去像处于一个或者多个光源的场景之中,同时还会介绍如何绘制多边形网格,使得效果看上去像是平滑的曲线曲面。其次我们会学到一种隐藏面消隐方法,看它如何在实际的渲染过程中发挥作用,这种方法也是 OpenGL 所使用的消隐方法。一般而言,我们将看到这样一个事实:为了实现更真实的效果,选择使用一种渲染模型而不是另外一个模型,总是要付出一定的代价——程序运行得更慢。不过当你使用 OpenGL 编写应用程序的时候,这点或许重要或许不那么重要。

8.2 渲染模型

真实表面上光的反射机制是非常复杂的,受很多因素的影响。有些因素是和几何有关的,例如光源、观察者和物体表面法向的相对方向。另外一些因素与物体表面的性质有关,比如粗糙程度和颜色。

渲染模型描述了光如何在物体表面发生散射和反射。本节我们将会看到一些简单的渲染模型,这些模型假定使用黑白光。这里**黑白**光指只有明暗不同而没有颜色,实际上就是灰度。因此使用一个数值就可以描述它:亮度。我们将会看到如何计算从物体各个部分到达相机的光的亮度。这些计算方法对于普通的光几乎是一样的,除了把红、绿、蓝三个分量分

别单独计算以外。

图形学中常用的渲染模型都假设把为场景照明的光源分为两种：点光源和**环境**光源(ambient light)。这些光源会"照射"到不同物体的表面。入射光和表面的相互作用可以分为三种：

- 一些被物体表面吸收而转化成热能。
- 一些在物体表面反射。
- 一些折射进入物体内部，比如光线进入玻璃。

如果所有的光都被吸收了，那物体看上去是黑色的，这就是物理中所谓的**黑体**。如果所有的入射光都被传导出去，那么物体只能通过反射的效果(或者**折射**的效果，如果把相机置于物体背光面)来看到，关于折射我们将在第 12 章讨论。

这里我们只关心在表面被反射或者散射的这部分光。一部分这样的光正好入射到眼睛，使得我们能够看到物体。到达眼睛的光量的多少取决于物体表面的朝向、光源和观察者。我们假设入射光的反射可以分为两种：**漫反射**(diffuse scattering)和**镜面反射**(specular reflection)。

- **漫反射**发生时入射光浅浅地穿透物体表面然后向各个方向均匀散射。因为散射光和表面有强烈的相互作用，所以光的颜色通常被表面材质的属性所影响。
- **镜面反射**更像镜子和光的作用那样具有很强的方向性。入射光不穿透物体表面，而是直接在外表面反射。结果是产生高光，并且是表面看上去有光泽。在刻画镜面反射的最简单的模型中，反射光颜色和入射光相同，这个性质使得材质看上去趋近于塑料。在镜面反射的更复杂的模型中，颜色在高光的不同区域会有变化，这种模型更好地逼近了金属表面的光泽。我们对这两种镜面反射模型都会进行探讨。

更多的表面是以上两种反射的结合，根据表面特性的不同比如粗糙程度和材质类型，两种反射所占比例也不同。我们可以说从物体表面反射向某一方向的全部反射光是漫反射分量和镜面反射分量之和。对于我们关心的表面上的每一个点，我们需要计算到达眼睛的反射光的每个分量。下面将要介绍的算法就是来完成这个过程。

8.2.1　影响反射光的几何因素

我们需要找到用来计算漫反射和镜面反射分量的三个向量。图 8.8 画出了用来计算从 P 点到眼睛的光量所需要的这三个关键向量($\mathbf{s}$、$\mathbf{m}$ 和 $\mathbf{v}$)。

(1) 法向量 $\mathbf{m}$ 指向 P 点所在表面的法方向。

(2) 向量 $\mathbf{v}$ 从 P 点指向观察者的眼睛($\mathbf{v}$ 是观察者英文 viewer 的首字母)。

(3) 向量 $\mathbf{s}$ 从 P 点指向光源($\mathbf{s}$ 是光源英文 source 的首字母)。

这三个向量之间的角度是计算光强的基本因素。这些角度通常都是在世界坐标系下计算，因为一些变换(比如投影变换)不能保持夹角不变。

网格物体的每个面片拥有两个面。如果物体是实心的，两个面通常称为里面和外面。人的眼睛只能够看到外面(除非眼睛可以进入到物体内部!)，我们计算光的贡献也只考虑外面。但是对于有些物体，比如图 8.9 中的这个敞开的盒子，眼睛有可能看到盒盖的内表面。

能够看到取决于这个面的法向 $\mathbf{m}_2$ 和视点方向 $\mathbf{v}$ 的夹角。如果角度小于 90°则这个面可以被看到。因为这个角度的余弦值正比于两个向量的点乘 $\mathbf{v}\cdot\mathbf{m}_2$,所以只有当 $\mathbf{v}\cdot\mathbf{m}_2>0$ 时眼睛能够看到这个面。

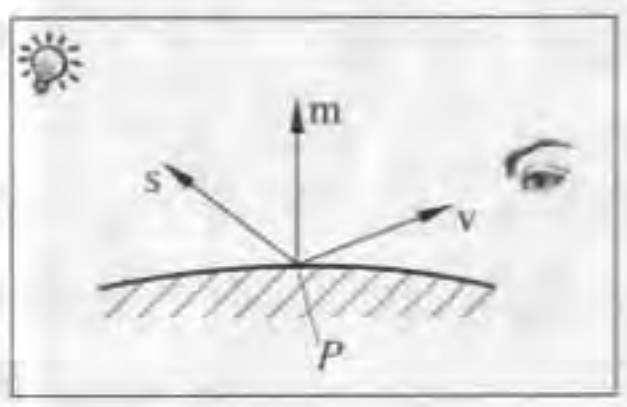


图 8.8 计算反射光时的重要向量

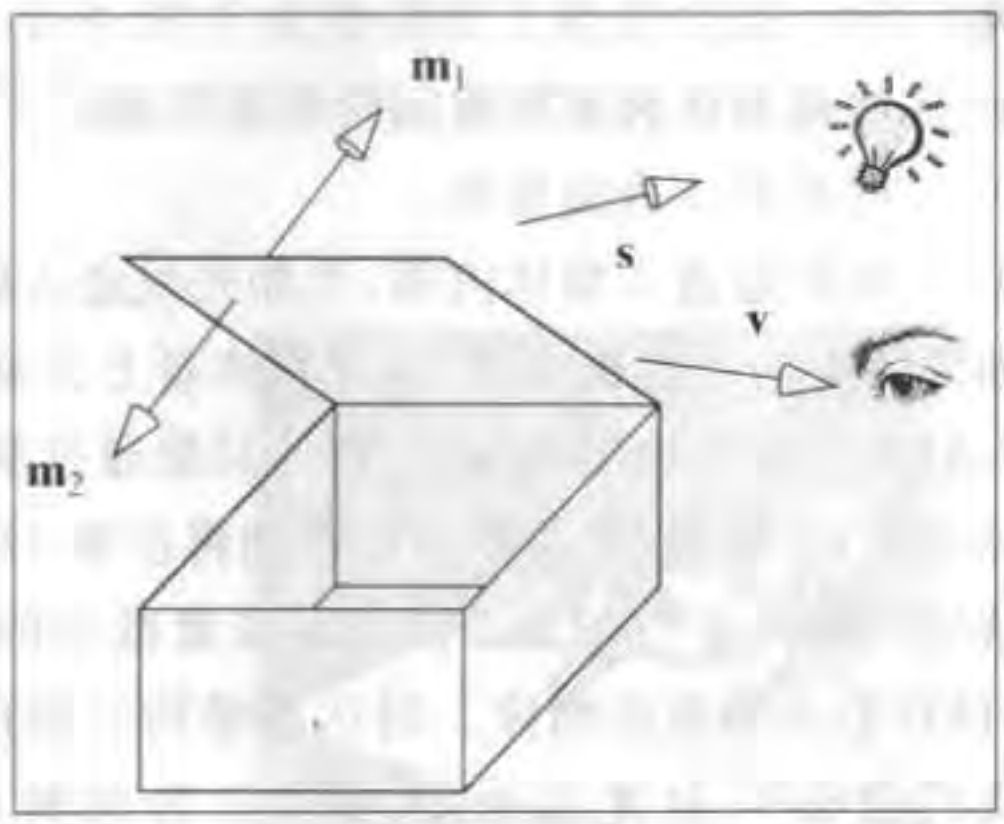


图 8.9 只为面片的一个面计算光照

对于给定面片的某个面我们将推导渲染模型。如果这个面背向视点,则没有任何反射光的贡献。在实际的应用中,必须告知渲染算法所计算的光的贡献来自面片的哪一面或者两面都计算。我们将看到 OpenGL 支持这一点。

8.2.2 如何计算漫反射分量

假设光从一个点光源投向一个面片(曲面的一个小面)。一部分光从这面向各个方向均匀的漫反射出来。一部分漫反射光到达了眼睛,把这部分光的强度设为 I_d。那么 I_d 和 $\mathbf{m}$、$\mathbf{v}$、$\mathbf{s}$ 方向的关系是什么呢?

漫反射的一个重要特点是它独立于视点方向。这点也经常被称为**全方位散射**(omnidirectional scattering)。因为散射向各个方向是均匀的,所以面片 F 相对于视点的朝向没有意义。因此,I_d 与 $\mathbf{m}$ 和 $\mathbf{v}$ 的夹角无关(除非 $\mathbf{v}\cdot\mathbf{m}<0$ 时 I_d 等于 0)。另外一方面,照亮面片的光量和面片相对光源的朝向相关:光量正比于面片能被看到的面积,也就是面片投影的面积。

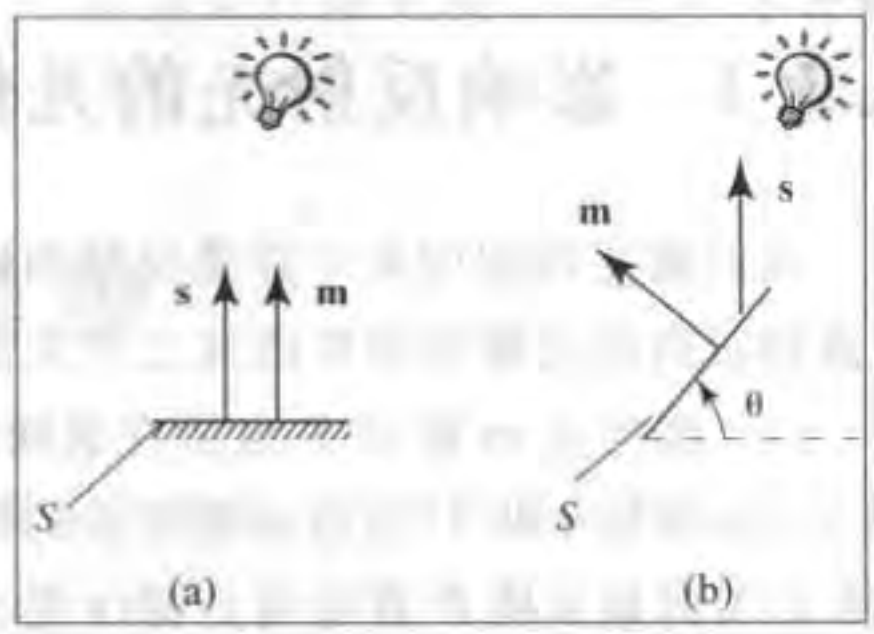


图 8.10 亮度取决于面片的投影面积

图 8.10(a)斜线的部分是一个点光源照在面 S 上,$\mathbf{m}$ 和 $\mathbf{s}$ 平行。图 8.10(b)中面旋转了一个角度 θ。投影的面积只有原先的 $\cos(\theta)$,所以 S 的亮度也成比例的下降。这种亮度和表面朝向的关系常被称作 **Lambert 法则**【Johann Heinrich Lambert, 1782-1777】。需要注意的是,当 θ 接近于 0 的时候,亮度随角度变化很小,这是因为余弦函数在这附近变化很小。但当 θ 接近 90°时,亮度快速地下降到 0。

我们知道 $\cos(\theta)$ 是单位化后 $\mathbf{s}$ 和 $\mathbf{m}$ 的点积,因此我们可以得到计算漫反射分量的强度

等式：

$$I_d = I_s \rho_d \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}$$

式中，I_s 是光源的亮度，ρ_d 是漫反射系数。需要注意的是如果面片背向视点，式子中的点积为负，这时我们希望为 0。所以一个更加准确的漫反射分量计算等式是：

$$I_d = I_s \rho_d \max\left(\frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}, 0\right) \tag{8.1}$$

图 8.11 显示了一个球体在不同光照下的漫反射效果，入射光来自上方，反射系数依次是：0、0.2、0.4、0.6、0.8 和 1。需要注意的是当 ρ_d 等于 0.0 的时候，球体是全黑的。底部的阴影部分（这里点积为负）也都是黑的。

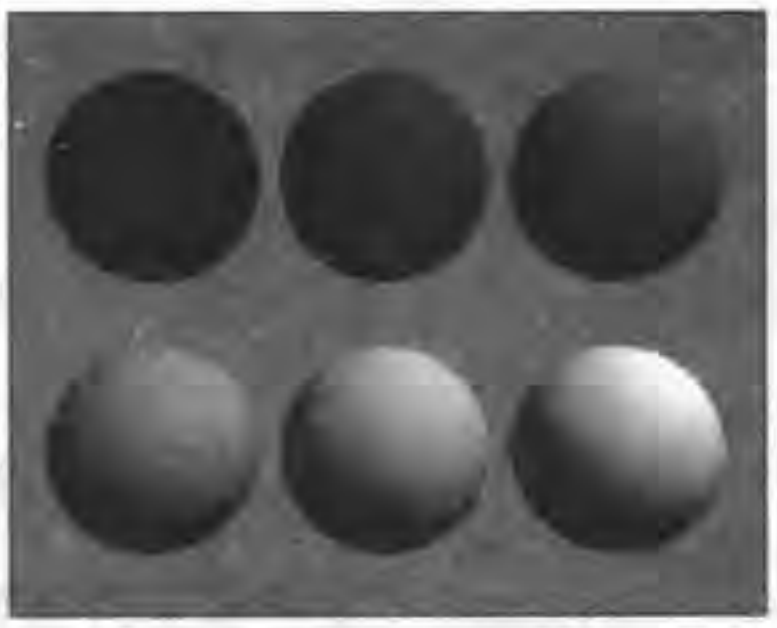


图 8.11　漫反射光下一些拥有不同反射系数的球体

在真实的世界中，漫反射背后的过程要比我们这个简单的模型复杂得多。反射系数 ρ_d 和入射光的波长（颜色）、夹角 θ 和不同的表面物理特性有关。但是为了简化和减少计算时间，渲染图像时这些效果都忽略了。为每个表面选择“合理”的 ρ_d 值，有时需要通过比较结果图片和真实观察效果之间的差别反复试验确定。

在某些渲染模型中距离对效果的影响被考虑其中，虽然这点有一定程度的争议。图 8.10 中表面 S 的反射光强度与 S 到光源的距离的三次方成反比。但是实验结果显示用这种方法生成的图片有夸张的深度效果。另外的一点是，有时候为了方便光源建模，会假设把光源放置在“无穷远”的位置。在这种情况下使用上述反比定理会使得光完全衰减！这里模型中的一个问题是，为了简单我们把光源用点光源建模，但是许多场景的真实照明来自周围物体的反射，这点很难用模型描述（这个效果用环境光分量描述）。所以结果就并不奇怪了，因为在一个不真实的模型基础上严格地附加物理定律最后只能导致不真实的结果。

引进距离项对于大多数图片来说只能增加很小很小真实感。一些过程的确强制光强与视点到物体的距离成反比，但是这不是基于物理原理。实验这个效果虽然很有意思，而且在 8.2.8 节中会看到 OpenGL 提供了一些接口，但是在下面的介绍中我们将不考虑距离因素。

8.2.3　镜面反射

真实物体的散射光并不是各向均匀的，所以镜面反射分量也需要加到渲染模型中。镜面反射能引起高光，当物体有光泽的时候这能够显著的增加渲染图片的真实性。在这节我们将讨论由 Phong【Phong75】提出的一个镜面反射光的简单模型，它非常容易实现。并且由 Phong 镜面反射产生的高光能给物体塑料一样的外表，所以当你打算渲染有光泽的塑料或玻璃的物体时 Phong 模型是不错的选择。对于拥有闪亮金属表面的物体，虽然可以通过在 OpenGL 中精心的选取某些颜色参数粗略的逼近，但是 Phong 模型并不适用于此。一些更复杂的镜面反射光模型对闪亮的金属有更好的表现。但因为 OpenGL 的渲染过程不直

接支持这些模型,所以我们在后面第 12 章光线跟踪的部分再具体介绍它们。

图 8.12(a)给出了光线从光源出发遇到物体表面后向不同方向反射的示例。在我们当前讨论的 **Phong 模型**中,反射光能量最大的方向是完全镜面反射(见第 4 章)的方向 **r**,这个方向反射角和入射角 θ 相等。如果表面是完全的镜面,这个方向也是全部光线反射的方向。附近其他的方向上反射光快速衰减,我们在图中用反射向量的相对长度表示。图 8.12(b)中我们用雷达圈常见的"光束模式"来表示这点。P 点到光束外边缘的长度表示在这个方向上散射光的相对强度。

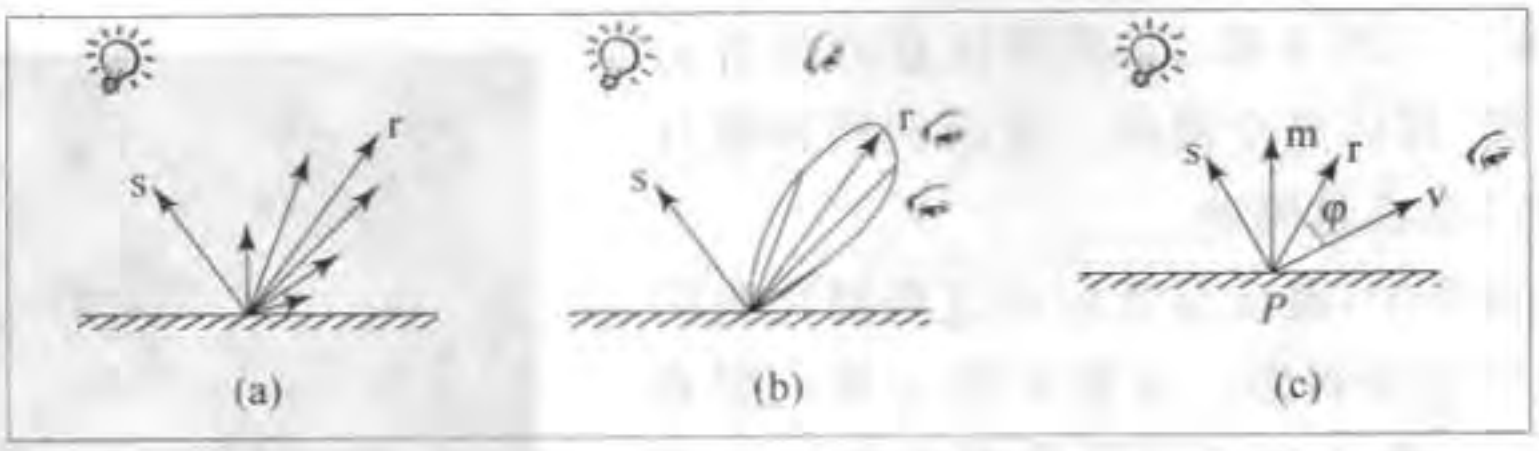


图 8.12 有光泽物体的镜面反射

图 8.12(c) 画出了如何量化这种光束模式。我们从第四章知道完全反射方向 **r** 依赖于 **s** 和表面法向 **m**:

$$\mathbf{r} = -\mathbf{s} + 2\frac{(\mathbf{s}\cdot\mathbf{m})}{|\mathbf{m}|^2}\mathbf{m} \quad \text{(镜面反射方向)} \tag{8.2}$$

对于很光滑但还不是完全镜面的表面,反射光强度随着 **r** 和 **v** 的夹角 ϕ 的增加而衰减。真正的衰减量是一个关于 ϕ 的复杂函数,但是在 Phong 模型中它被表示为 ϕ 余弦值的 f 次幂,也就是说$(\cos(\phi))^f$,其中 f 的值通常在 1~200 之间通过实验确定。

图 8.13 画出了不同 f 值下光强度随 ϕ 值变化的函数。可以发现 f 值越大,反射光越集中于 **r** 方向,也就是接近镜面。理想的镜面可以通过 $f=\infty$ 表示,但是完全反射通常用另外的方式表示,这将在第 12 章介绍。

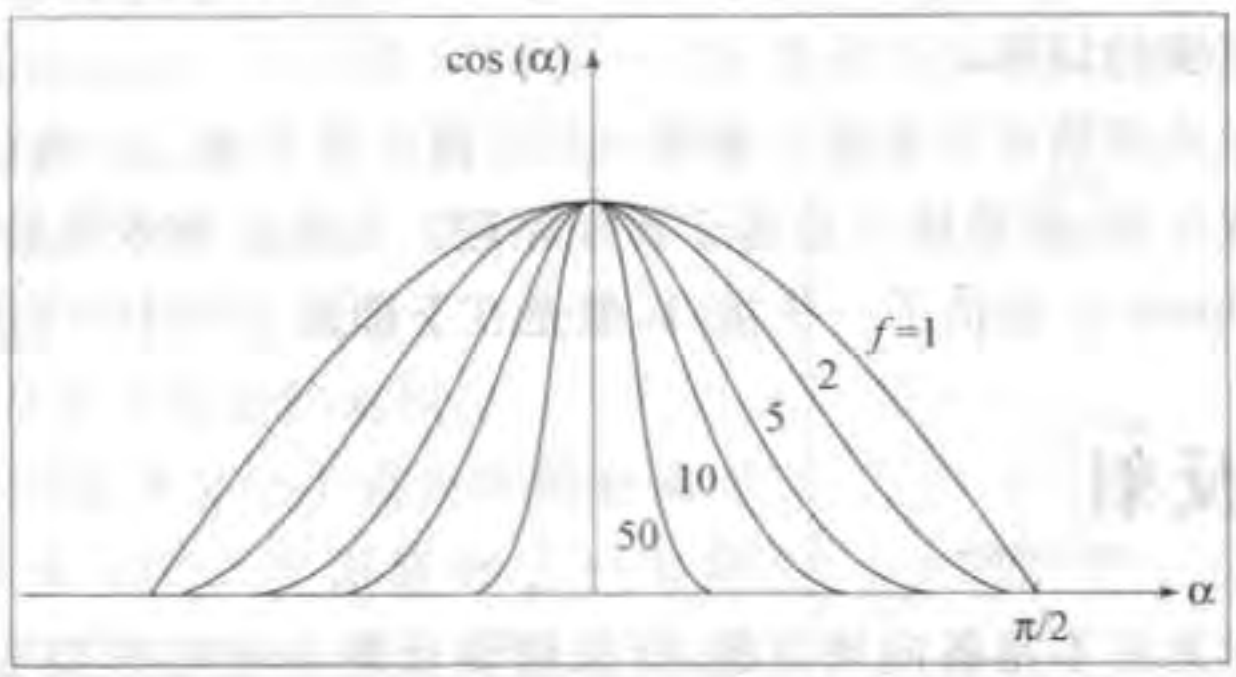


图 8.13 镜面反射光随角度的衰减

用 **r** 和 **v**(单位化后)的点积等价的替换 $\cos(\phi)$,镜面反射的光强度 I_{sp} 可以表示为

$$I_{sp} = I_s\rho_s\left(\frac{\mathbf{r}}{|\mathbf{r}|}\cdot\frac{\mathbf{v}}{|\mathbf{v}|}\right)^f \tag{8.3}$$

其中新的参数 ρ_s 是**镜面反射系数**。和其他在渲染模型中的参数一样,它通常也是通过实验的方法确定(和漫反射一样,如果 $\mathbf{r}\cdot\mathbf{v}$ 为负则 I_{sp} 为 0)。

我们可以看看这个方法计算镜面反射分量的复杂度。它包含了若干点乘、向量的单位化和指数函数。这就自然地让我们去寻找一个简单一些的方法。

通过半角向量加速

通过等式(8.3)计算镜面反射的方法效率比较低，因为它首先需要计算出向量 **r**，并且进行单位化。使用半角向量可以在一定程度上改进这个不足。在实际的应用中 **r** 用另外的一项代替从而加速计算，这个方法由 Blinn【Blinn77】首先提出。为了避免使用 **r** 和 **v** 夹角的余弦值，需要首先计算出 **s** 和 **v** 的半角向量 $\mathbf{h}=\mathbf{s}+\mathbf{v}$，如图 8.14 所示。如果表面法向是沿 **h** 方向，则观察者能看到最强的反射光。因此 **m** 和 **h** 的夹角 β 可以用来表示观察者看到的反射光强度的衰减。β 和 ϕ 不相等(事实上如果共面的变化 **s** 和 **v**，β 可以是 ϕ 的两倍，见习题)，但是这个差别可以通过选取不同的指数 f 来弥补。(无论如何，这个镜面反射项不是基于物理原理，所以这种通过调整 f 产生满意结果的做法是可以接受的)因而在一般的应用中镜面反射都使用基于 $\cos(\beta)$ 的方法，即使用 **h** 和 **m** 的点积：

$$I_{sp} = I_s\rho_s \max\left(0, \left(\frac{\mathbf{h}}{|\mathbf{h}|}\cdot\frac{\mathbf{m}}{|\mathbf{m}|}\right)^f\right) \quad \text{\{调整后的镜面反射分量\}} \tag{8.4}$$

值得注意的是通过这个变化就不需要计算反射向量 **r**，从而节省了计算时间。另外，如果光源和视点都非常远，则 **s** 和 **v** 对于物体上不同的表面都是不变的，所以(对于每个面)只需计算一次 $\mathbf{h}\cdot\mathbf{m}$ 就可以了。

图 8.15 比较了一个球体反射不同强度的镜面反射光。从上到下镜面反射系数 ρ_s 依次是 0.25、0.50 和 0.75，从左到右指数 f 依次是 3、6、9、25 和 200(对于所有的球体环境光和漫反射系数都分别是 0.1 和 0.4。光源在所有的例子中都是位于同样的位置)。

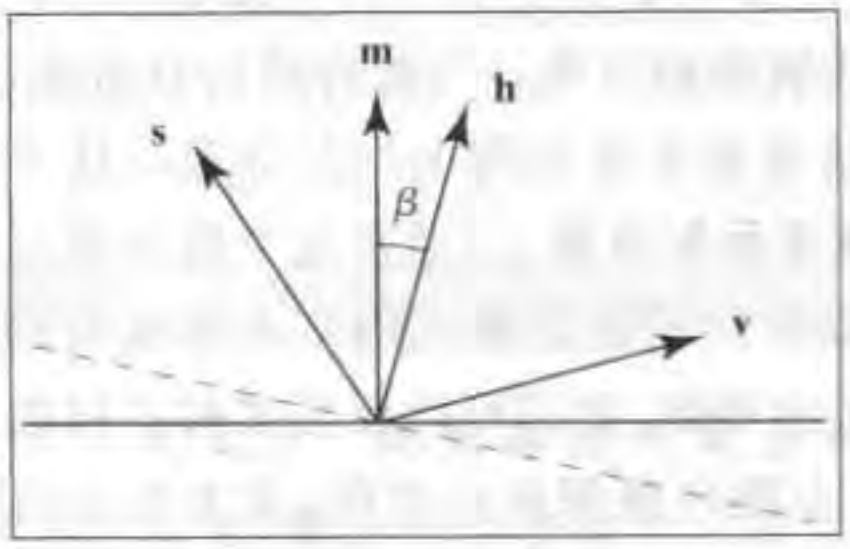


图 8.14　半角向量

图 8.15　光泽表面的镜面反射

事实上，镜面反射的物理机制要比 Phong 模型的描述复杂得多的多。一个更加真实的模型会假定镜面反射系数依赖于入射光波长 λ(比如颜色)和入射角度 θ_i(图 8.10 中向量 **s** 和 **m** 的夹角)。这种模型试图描述光如何从不同的材质(比如锌和锡铅合金)中反射，它的效果将在介绍等式(8.9)时做简要的讨论。我们同样会在第 12 章探讨彩虹和三棱镜的原理时解决这些问题。

练习

练习 8.2.1　半角向量

根据图 8.14 中的几何关系，证明当相关向量共面时，可能出现 $\beta=2\phi$ 的情况。

练习 8.2.2 一种镜面反射加速

Schlick【Schlick94】计算镜面反射项的另外一种方法替代指数函数的使用。令 D 表示等式(8.3)中的点积 $\mathbf{r}\cdot\mathbf{v}/|\mathbf{r}||\mathbf{v}|$。Schlick 提出用下式代替 D^f：

$$\frac{D}{f-fD+D}$$

这个式子的计算速度更快。画出在不同的 f 值下，比较两个函数在 D 在区间[0,1]上的函数图像。另外特别要注意 D 在 1 附近时的情况，因为这是反射光最强的区域。

8.2.4 环境光的作用和人类的视觉感知

通过简化现实中光线和物理表面的反射过程，我们得到了漫反射和镜面反射分量的计算方法。与简单的在线框上填充颜色相比，这种依赖于视点位置、模型和光源位置计算颜色的方法极大地增强了图像的真实感。

为了追求一个简单的反射模型，我们远离了场景渲染的完美结果。比如我们得到的阴影，颜色显得过于深和生硬。为了柔化这些阴影，我们可以增加第三个光分量，称为环境光。

如果只考虑漫反射和镜面反射分量，物体表面任何背向点光源的阴影部分一点光都接收不到，所以只能画为黑色！但在现实的经验告诉我们不是这样。我们身边常见的场景都会接受一些间接光源的照射。这些光从环境中的光源发出后经过周围不同物体的多次反射，最后到达物体，比如透过窗户中射入的光、荧光灯发出的光，还有其他类似的。但是想用模型精确地描述这种光是非常吃力的，会大量增加计算量。

环境光源和环境反射

为了解决阴影完全黑色的问题，我们可以假设环境中均匀地分布着昏暗的背景光，称为**环境光**。这种环境光的光源并不处在某个位置，并且光在各个方向上均匀分布。这个光源用光强度 I_a 表示。给模型的每个面片赋予一个**环境光反射系数** ρ_a(通常这个值和漫反射系数 ρ_d 相等)，则可以简单的把 $I_a\rho_a$ 这项加上原来漫反射和镜面反射光用来表示从面片上 P 点到达视点的全部光。I_a 和 ρ_a 的值通常也是经过实验得到，我们可以反复实验不同的值选择看上去效果最好的。环境光太小会使得阴影看上去颜色很深而且生硬；太大会使得图像看上去像浸泡过一样——平淡缺少变化。

图 8.16 画出了在漫反射光的基础上添加不同强度的环境光后，球体模型的渲染效果。每个例子中漫反射光源强度和环境光源强度都是 1.0，漫反射系数都是 0.4。从左到右环境光反射系数依次是 0.0、0.1、0.3、0.5 和 0.7。只有添加了最恰当的环境光量，原来球体底部生硬的阴影才会被柔化从而看上去更加真实。另一方面过多的环境光会过于影响阴影效果。

图 8.16 环境光的效果

8.2.5　如何综合光的贡献

我们现在可以把漫反射、镜面反射和环境光三种光的贡献加起来得到从 P 点到达视点总的光量 I：

$$\begin{aligned} I &= \text{环境光} + \text{漫反射光} + \text{镜面反射光} \\ I &= I_a\rho_a + I_d\rho_d \times lambert + I_{sp}\rho_s \times phong^f \end{aligned} \tag{8.5}$$

其中：

$$lambert = \max\left(0, \frac{\mathbf{s} \cdot \mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right) \quad phong = \max\left(0, \frac{\mathbf{h} \cdot \mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right) \tag{8.6}$$

I 的值与三种光源强度和反射系数有关，同时也和点 P、视点和光源的相对位置有关。这里我们给予漫反射光源强度和镜面反射光源强度不同的名称 I_d 和 I_{sp}，这是因为在 OpenGL 中你可以分别单独设置这两个值。这点我们在后面会看到，在实际使用中，这两个值通常相等。

为了更加深入地理解光强度 I 和 P 点位置的关系，请再次看一看图 8.10。I 是根据面片上不同位置的点 P 计算出来的。环境光分量在整个面片上没有变化；**m** 对于面片上的所有点 P 值都相同，但是 **s** 和 **v** 的方向却依赖于 P。（比如，$\mathbf{s}=S-P$，其中 S 是光源的位置。那 **v** 是如何依赖于 P 和视点的呢？）如果光源的位置相对很远（常见的情况），**s** 因 P 的改变只会有很小的变化，所以漫反射分量对于不同的 P 点变化也会很小。特别是当 **s** 和 **m** 几乎平行的时候更是这样，因为当夹角很小的时候 cos() 的值变化很小。对于远处的光源，不同点 P 的半角向量 **h** 也是变化很小。另外一方面，如果光源里面片很近，不同的 P 点对应的 **s** 和 **h** 会有很大的变化。这时镜面反射项在整个面片上变化的非常明显，所以明亮的高光只出现在面片的一小部分上面。同样的效果也会出现在视点和面片距离很近的时候——因为引起了 **v** 方向的大范围变化，还有就是指数 f 值很大的时候也是这样。

练习

练习 8.2.3　视点距离对效果的影响

（基于直觉）描述 P 点在面片上变化时各种光的分量如何变化，(a) 当视点距离面片很远和 (b) 当视点离面片很近时各是怎样？

8.2.6　增加颜色

我们可以很直接地将这个渲染模型推广到有颜色的表面反射彩色光的情况。虽然这个推广很简单，但是最后的效果很好且很合理。

第 11 章有更多的关于颜色本质的背景和具体介绍，这里我们只需知道彩色光是由红、绿、蓝三种光通过不同比例混合组成的。处理彩色光源和彩色的物体表面的时候，我们单独的计算每个颜色分量，简单地将它们加在一起成为最后反射光的颜色。所以等式(8.5)应被使用三次：

$$
\begin{aligned}
I_r &= I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times lambert + I_{spr}\rho_{sr} \times phong^f \\
I_g &= I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times lambert + I_{spg}\rho_{sg} \times phong^f \\
I_b &= I_{ab}\rho_{ab} + I_{db}\rho_{db} \times lambert + I_{spb}\rho_{sb} \times phong^f
\end{aligned} \tag{8.7}
$$

(其中 $lambert$ 和 $phong^f$ 在等式(8.6)中给出)来计算反射光的红、绿、蓝分量。注意我们提到过光源有三种颜色类型：环境光=(I_{ar}，I_{ag}，I_{ab})，漫反射=(I_{dr}，I_{dg}，I_{db})和镜面反射=(I_{sr}，I_{sg}，I_{sb})。通常漫反射和镜面反射的光源颜色相同。另外还应该注意到 $lambert$ 和 $phong^f$ 项与要计算的颜色分量无关(除了通过反射系数来影响)，所以这些项只需要被计算一次。为了使用这个的模型，我们还需要定义 9 个反射系数：

环境光反射系数：ρ_{ar}，ρ_{ag} 和 ρ_{ab}

漫反射系数：ρ_{dr}，ρ_{dg} 和 ρ_{db}

镜面反射系数：ρ_{sr}，ρ_{sg} 和 ρ_{sb}

环境光和漫反射系数依赖于表面本身的颜色。这里"表面的颜色"指当其被白光照射时所反射的漫反射颜色：一个红色的表面在白光照射下就显示出红色。如果在其他一些颜色光的照射下，它可能会显出完全不同的颜色。下面的示例说明了这一点：

例 8.2.1 物体的颜色

如果我们说一个球体的颜色是由 30%的红色、45%的绿色和 25%的蓝色组成的。那么分别设置环境光和漫反射系数为(0.3K，0.45K，0.25K)是合理的，其中 K 是一个比例因子，用来决定球体反射入射光的整体比例。如果这个球体放在红、绿、蓝比例相等($I_r = I_g = I_b = I$)的白光中，则漫反射光强度的三个分量是 $I_r = 0.3KI$，$I_g = 0.45KI$，$I_b = 0.24KI$，所以正和期望的一样，我们可以看到它的颜色是 30%红、45%绿和 25%蓝。

例 8.2.2 一个亮红色物体置于亮绿色光中

假设一个球体的环境光和漫反射系数是(0.8，0.2，0.1)，它在白光中看上去几乎是红色。我们使用一个亮绿色光源 I_s=(0.15，0.7，0.15)，则反射光的颜色是(0.12，0.14，0.015)。这是一个红绿分量几乎相等的混合，最终呈现的颜色浅黄色(我们将在 11 章讨论)。

镜面反射光颜色 因为镜面反射光更像镜子，即反射光的颜色通常和光源颜色相同。例如，平常的经验告诉我们在黄色光中，一个红色的苹果高光部分的颜色是黄色而不是红色。同样的情况也可以在塑料材质的物体上观察到。为了制造塑料表面的镜面高光，等式(8.7)中的镜面反射系数 ρ_{sr}，ρ_{sg} 和 ρ_{sb} 同城都被设置成一样的值 ρ_s，所以反射系数本身都是"灰度"值，不会改变入射光的颜色。设计者可以为一个不太光泽的塑料表面选择 ρ_s=0.5，或者为一个非常有光泽的表面选择 ρ_s=0.9。

不同材质制造的物体

通过精心地选择反射系数，可以使物体呈现特定材质的效果，比如铜、金或者锡铅合金，至少能在效果上做到比较接近。McReynolds 和 Blythe【McReynolds97】建议使用图 8.17 中列出的反射系数。图 8.18 中的球体就是使用了这些系数渲染后的结果(在配套的网站上，有一个彩色版本，其中的球体相当真实地还原了相应材质的外观)。值得注意的是镜面反射系数拥有不相等的红、绿、蓝分量，所以镜面反射光不是简单地和入射光一样。但是

McReynolds 和 Blythe 提醒使用者注意，由于 OpenGL 的渲染算法结合了 Phong 镜面反射分量，所以最终的视觉效果并不是完全的真实。我们将在第 12 章里回顾这个问题并且介绍更加真实的 Cook-Torrance 渲染方法。

材料	环境：ρ_{ar}、ρ_{ag}、ρ_{ab}	扩散：ρ_{dr}、ρ_{dg}、ρ_{db}	反射：ρ_{sr}、ρ_{sg}、ρ_{sb}	指数：f
黑塑料	0.0 0.0 0.0	0.01 0.01 0.01	0.50 0.50 0.50	32
黄铜	0.329412 0.223529 0.027451	0.780392 0.568627 0.113725	0.992157 0.941176 0.807843	27.8974
青铜	0.2125 0.1275 0.054	0.714 0.4284 0.18144	0.393548 0.271906 0.166721	25.6
铬	0.25 0.25 0.25	0.4 0.4 0.4	0.774597 0.774597 0.774597	76.8
铜	0.19125 0.0735 0.0225	0.7038 0.27048 0.0828	0.256777 0.17622 0.06014	12.8
金	0.24725 0.1995 0.0745	0.75164 060648 022648	0.628281 0.555802 0.366065	51.2
白腊	0.10588 0.058834 0.113725	0.427451 0.470588 0.541176	0.3333 0.3333 0.521569	9.84615
银	0.19225 0.19225 0.19225	0.50754 0.50754 0.50754	0.508273 0.508273 0.508273	51.2
抛光银	0.23125 0.23125 0.23125	0.2775 0.2775 0.2775	0.773911 0.773911 0.773911	89.6

图 8.17　常见材质的参数

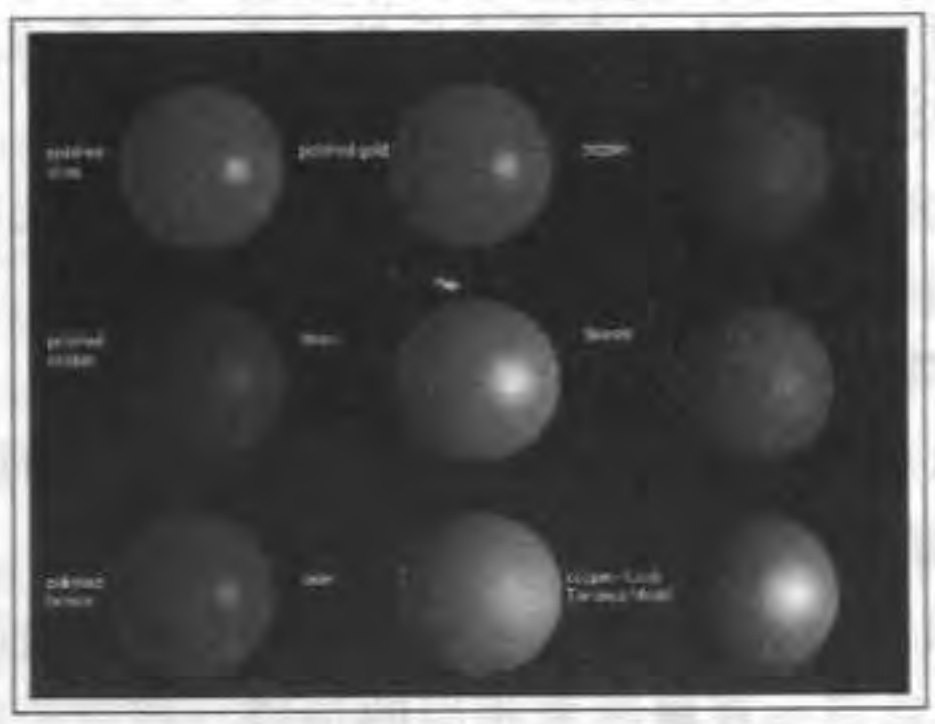


图 8.18　不同材质的光泽球体

8.2.7 渲染和图形管线

着色位于图形管线的哪一步？又是如何完成的？图 8.19 再次给出了流水线。其中关键的一点是网格的顶点与其相应的顶点法向在顺着流水线前进，并且所有的着色计算都是在顶点上完成的。(回顾 Mesh 类中的 draw()方法传递每个顶点与其对应的法向给 OpenGL，见图 6.15。)

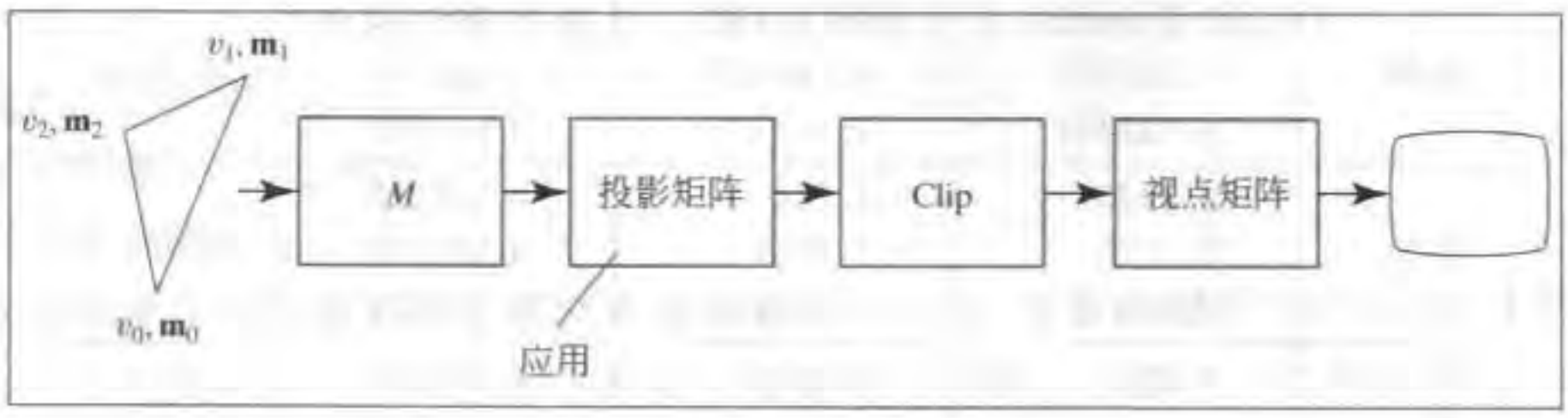


图 8.19 修正后的图形管线

图中画出了一个顶点是 v_0, v_1, v_2 的三角形被渲染的过程。顶点 $\mathbf{v}_i$ 对应的法向是 $\mathbf{m}_i$。这些量通过如下调用传送给流水线：

```
glBegin(GLPOLYGON);
    for(int i=0; i<3; i++)
    {
      glNormal3f(m[i].x, m[i].y, m[i].z);
      glVertex3f(v[i].x, v[i].y, v[i].z);
    }
glEnd();
```

调用 glNormal3f()设置了“当前法向量”，这个法向被应用于随后的通过 glVertex3f()设置的全部顶点。当前法向量直到另外一个 glNormal3f()调用的时候才会改变。所以上面的代码中每一个顶点都被设置了一个不同法向。

顶点通过模型视点矩阵(modelview matrix)$\boldsymbol{M}$ 变换到相机(视点)坐标系。法向量也会变换，但是向量的变换和顶点的变换不同。回顾在 6.5.3 节的介绍，通过矩阵 $\boldsymbol{M}$ 变换表面上的顶点，原来表面上点对应的法向 $\mathbf{m}$ 在变换后的表面上会变成 $\boldsymbol{M}^{-T}\mathbf{m}$，其中 $\boldsymbol{M}^{-T}$ 是 $\boldsymbol{M}$ 的逆矩阵再转置。OpenGL 会自动而正确地处理法向量。

在后面的小节中我们会看到，OpenGL 允许你设定不同的光源和它们的位置。光源也是物体，并且光源的位置同样会被模型视点矩阵所变换。

所有通过模型视点矩阵变换后的量都是处于相机坐标系下。此时应用等式(8.7)表达的渲染模型，这样每个顶点就会被添加上了颜色。计算颜色要知道向量 $\mathbf{m}$、$\mathbf{s}$ 和 $\mathbf{v}$，这些量在流水线处理的这个时刻都是已知的(请相信这一点)。

在随后的处理流水线上，伪深度值被计算出来，顶点也会通过投影矩阵(projection matrix)进行变换。颜色信息则一直跟随着相应的顶点。随后的剪裁过程是在齐次坐标系下完成的，这步可能会改变一些顶点。图 8.20 就给出了这样一个例子：三角形的顶点 v_1 被

剪裁掉，增加了两个新的顶点 a 和 b，三角形变成了四边形。新顶点的颜色也必须要计算，因为这是渲染时需要的。

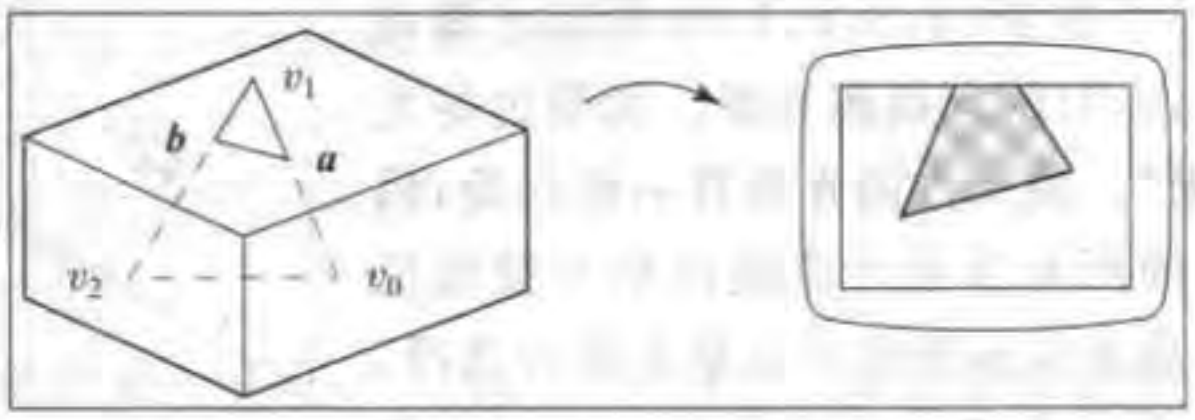


图 8.20　用视锥剪裁一个多边形

新顶点的颜色通常经过插值计算得到，就像在 Gouraud 明暗处理时做的一样。例如，假设 v_0 的颜色是(r_0, g_0, b_0)，v_1 的颜色是(r_1, g_1, b_1)。如果点 a 位于从 v_0 到 v_1 的 40%的位置，则 a 的颜色是(r_0, g_0, b_0)的 60%和(r_1, g_1, b_1)的 40%的混合。这可以用等式表示：

$$a\text{ 点颜色} = (\mathrm{lerp}(r_0, r_1, 0.4), \mathrm{lerp}(g_0, g_1, 0.4), \mathrm{lerp}(b_0, b_1, 0.4)) \tag{8.8}$$

其中我们用使用了函数 lerp()(线性插值 liner interpolation 的简称)，这个函数在第 4 章介绍动画插值帧的时候也用到过。

定义为：

$$\mathrm{lerp}(G, H, f) = G + (H - G)f \tag{8.9}$$

它的值等于 G 和 H[1] 以 f 为比例插值的结果。

顶点最后通过视窗变换(view port transformation)映射到屏幕坐标(screen coordinates)(同时附带着 0 到 1 之间的伪深度值)。这个四边形最终被渲染出来(在隐藏面消隐之后)，如图 8.20 所示。我们将要对实际的渲染步骤做更多的介绍。

8.2.8　在 OpenGL 中使用光源

OpenGL 提供了许多函数用来设置光源和设置物体表面材质属性。理解并掌握其中所有的变化和细节会比较乏味，所以我们在这只介绍基本的用法。我们会讨论如何在场景中设置不同类型的光源。下一节中我们再看一看设置物体表面反射属性的方法。

创建光源

OpenGL 允许你定义八个光源，依次用 GL_LIGHT0、GL_LIGHT1 等命名。每个光源可以设置不同的属性，也可以被开启或关闭。每个属性都有默认的赋值。例如，建立一个位置在世界坐标系下(3,6,5)的光源，可以使用[2]：

```
GLfloat myLightPosition[]={3.0, 6.0, 5.0, 1.0};
glLightfv(GL_LIGHTO, GL_POSITION, myLightPosition);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHTO);
```

数组 myLightPosition[](这个数组也可以使用其他你喜欢的名字)指定了光源的位置。这个位置连同光源名 GL_LIGHT0 一起传递给 glLightfv()来设置光源 GL_LIGHT0。

一些光源，比如桌上的台灯，位于场景中，相比而言另外一些光源，比如太阳，处于无限远的位置。OpenGL 允许你通过使用齐次坐标设置这两种类型光源的位置：

$(x,y,z,1)$：位置在(x,y,z)的局部光源

$(x,y,z,0)$：位于(x,y,z)方向上无限远的光源

图 8.21 画出了一个位于(0,3,3,1)的局部光源和一个"位于"方向(3,3,0,0)的远距离光源。无限远的光源也经常称为"**方向光**"。使用方向光源有一些好处，因为方向 s 对于场景中的所有顶点计算漫反射和镜面反射时都是不变的。但是方向光源也不总是正确的选择：一些视觉效果只有在使用靠近物体的局部光源时才能实现。

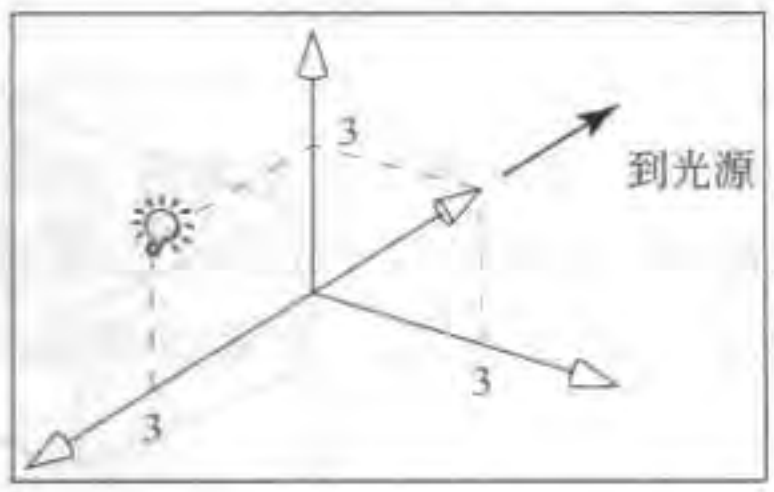


图 8.21 局部光源和无限远处的光源

你也可以非常直观地设置不同颜色的光源。OpenGL 允许你为一个光源发出的三种不同光——环境光、漫反射和镜面反射，设置不同的颜色。一个光源发出环境光听上去可能有些奇怪。它只是根据等式(8.7)来处理：一个整个场景中全局的全方位的光源。将它和一个光源联系在一起的好处是这样可以在应用过程中方便地打开和关闭它。(OpenGL 也提供一个真正的环境光，不与任何光源绑定在一起，我们将在后面有关"光照模型"中介绍。)

光源的颜色使用数组来储存，然后传递给 glLightfv()。

```
GLfloat amb0[ ]={0.2, 0.4, 0.6, 1.0};          //define some colors
GLfloat diff0[ ]={0.8, 0.9, 0.5, 1.0};
GLfloat spec0[ ]={1.0, 0.8, 1.0, 1.0};
glLightfv(GL_LIGHT0, GL_AMBIENT, amb0);        //attach them to LIGHT0
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff0);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec0);
```

颜色使用所谓的 **RGBA** 格式，也就是红、绿、蓝和"透明度(alpha)"。透明度只当在屏幕上混合两种颜色的时候使用。我们在第 9 章讨论这个。对于我们现在的应用它通常设为 1.0。

光源不同的参数都有默认的数值。对于所有的光源：

默认环境光=(0,0,0,1)；　←可能最昏暗：黑色

对于光源 LIGHT0：

默认漫反射=(1,1,1,1)；　←可能最亮：白色

默认镜面反射=(1,1,1,1)；　←可能最亮：白色

同时对于其他光源的漫反射和镜面反射默认都是黑色。

探照灯

光源默认都是点光源，即发出的光均匀地射向各个方向。但是 OpenGL 允许你把它们设置为探照灯，这样它们发出的光就集中在一些特定的方向。图 8.22 画出了一个朝向方向 **d**，"切角"为 α 的探照灯。

在圆锥之外的点无法看到光。对于圆锥之内的顶点 P，到达 P 点的光强以因子 $\cos^{\epsilon}(\beta)$ 衰减，其中 β 是 **d** 与光源和 P 点连线的夹角，epslon 是用户指定的光源根据角度衰减的指数。

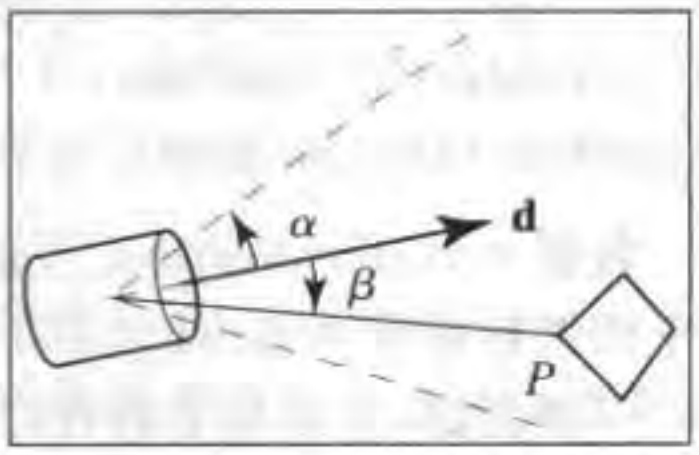


图 8.22 OpenGL 探照灯的属性

设置探照灯的参数，使用 glLightf()设置单个值，使用 glLightfv()设置一个向量：

```
glLight(GL_LIGHTO, GL_SPOT_CUTOFF, 45.0);
glLight(GL_LIGHTO, GL_SPOT_EXPONENT, 4.0);
Glfloat dir[]={2.0, 1.0, - 4.0};
glLight(GL_LIGHTO, GL_SPOT_DIRECTION, dir);
direction vector, dir;
```

这些参数的默认值是 $\mathbf{d}=(0,0,-1)$，$\alpha=180^\circ$，$\varepsilon=0$，这些参数使得光源是一个全方向的点光源。

光随距离的衰减

OpenGL 还允许你设置光随光源发出的距离以多快的速度衰减。虽然我们已经说过这种依赖关系不是很重要，但通过实验不同的衰减速率来仔细地调整图像也还是有趣的。OpenGL 中位置光源的强度衰减是根据如下衰减因子：

$$atten=\frac{1}{k_c+k_lD+k_qD^2} \tag{8.10}$$

其中 k_c、k_l 和 k_q 是系数，D 是光源到顶点的距离。这个表达式非常通用，它使你能够实现关于光源距离的常数、线性和二次函数(反平方定律?)。这些参数被如下函数调用控制：

```
glLightf(GL_LGIHT0, GL_CONSTANT_ATTENUATION, 2.0);
```

同样对于 GL_LINEAR_ATTENUATION 和 GL_QUADRATIC_ATTENUATION。参数的默认值是 $k_c=1$、$k_l=0$ 和 $k_q=0$，也就是默认没有衰减。

光照模型

OpenGL 通过设置三个参数来指定光照模型的一般规则。这些参数传递给不同版本的 glLightModel 函数。

a. 全局环境光颜色

你可以在场景中建立一个全局环境光源，它独立于任何一个普通的光源。建立这个光源，制定它的颜色可以使用：

```
GLfloat amb[]={0.2, 0.3, 0.1, 1.0};
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, amb);
```

这样就设置了漫反射光源的颜色为(0.2,0.3,0.1)。默认的值是(0.2,0.2,0.2,1.0)，所以这个漫反射光源始终存在，除非你有意改变它。这使得即使你没有调用任何光照函数，场景中的物体也可以被看见。

b. 视点是局部还是无限远?

在第 8.2.3 节我们介绍过 OpenGL 计算镜面反射光使用“半角向量”

$$\mathbf{h}=\mathbf{s}+\mathbf{v} \tag{8.11}$$

对于网格上的不同顶点对应的 **s** 和 **v** 真实的方向一般不同(画图说明这点)。如果光源是方向光，则 **s** 是常数，但是 **v** 对于不同的顶点仍然变化。如果 **v** 对于所有的顶点也是常

数,则渲染速度能够增加。在默认的情况下,OpenGL 使用 v=(0,0,1),这是在相机坐标系下 z 轴的正方向。你可以强制渲染管线使用对于每个顶点真实的 v 值:

```
glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
```

c. 多边形的两面是否都适当地着色?

模型中的每个多边形面片都有两个面。建模的时候,我们一般把它们想象成"内表面"和"外表面"。列出面片各个顶点的排列顺序时,惯例上是按照从物体外面观察的逆时针方向。大多数的网格模型表示的物体都是封闭空间的固体,这就能很好地定义里面和外面。对于这样的物体,相机只能够看到它们外表面(假定相机不会放置在物体内部)。在适当进行隐藏面消隐之后,从视点处看每个面片的内表面肯定会被更近的面片遮挡住。

OpenGL 中没有里面和外面的概念。它只能区分"正面"和"背面"。一个面是**正面**,如果它的顶点从视点看过去是按照逆时针顺序(CCW)列出的。图 8.23(a)画出了从视点观察到的一个立方体,我们假定建模使用了逆时针顺序的惯例。箭头表明了每个面片的顶点传递给 OpenGL 时的顺序(在 glBegin(GL_POLYGON);…;glEnd()之间)。对于一个空间封闭的物体,所有的可见面片因此都是正面。同时 OpenGL 会适当而正确地画出它们。OpenGL 也会画出背向面片,但是它们最终被一些更近的正面所遮挡。

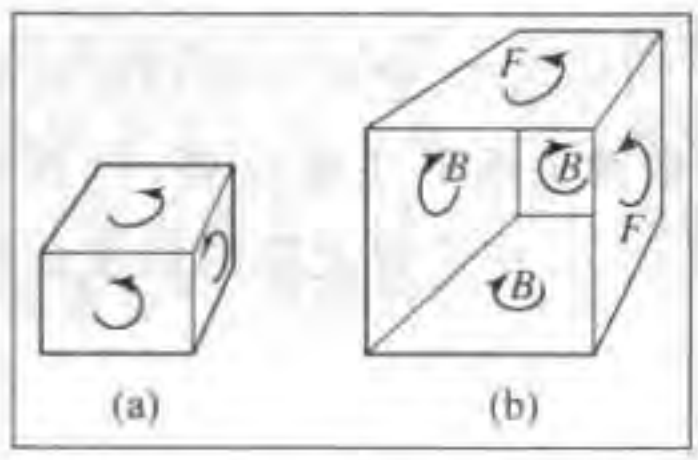


图 8.23 OpenGL 中正面的定义

在图 8.23(b)中情况就不一样了,盒子的一个面被移去。箭头依旧列出了各个面片的顶点传递给流水线时的顺序。这次,三个可见的面片是背面。在默认的情况下,OpenGL 不会对这些面片适当地着色。为了强制 OpenGL 适当地对背面着色,使用:

```
glLightModeli(GL_LIGHT_MDOEL_TWO_SIDE, GL_TRUE);
```

这样 OpenGL 会对所有背面的法向取反,这样它们就指向了观察者,并且可以适当地计算颜色。用 GL_FALSE(默认值)替换 GL_TRUE 就可以关闭这个功能。

注意,OpenGL 画的面片不会投阴影,所以背面会收到光源同样的光,即使可能有其他的面片位于它和光源之间。

光源的移动

回顾光源通过模型视点矩阵的变换和普通的顶点是一样的。因此光源位置也可以通过适当地使用 glRotated()和 glTranslated()重置。数组 position 通过调用 glLightfv(GL_LIGHT0, GL_POSITION, position)设置的位置会被模型视点矩阵改变,这个效果是在 glLightfv()调用的时候完成的。所以为了通过变换改变光源位置,并且还要独立于相机的运动,要把光源位置变换的命令置于一对压栈/退栈之间,例如:

```
void display()
{  GLfloat position[ ]={2, 1, 3, 1};                    //initial light position
    <..clear color and depth buffers..>
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
```

```
    glPushMatrix();
    glRotated(...);     glTranslated(...);                //move the light
    glLightfv(GL_LIGHT0, GL_POSITION, position);
    glPopMatrix();
    gluLookAt(...);                                       //set the camera position
    <..draw the object..>
    glutSwapBuffers();
}
```

另外一方面，为了使光源随着相机一起移动，使用：

```
GLfloat pos[ ]={0,0,0,1};
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glLightfv(GL_LIGHT0, GL_POSITION, position);          //light at (0,0,0)
gluLookAt(...);                                       //move light and camera
<..draw the object..>
```

这样建立的光源放置在视点的位置(就像矿工安全帽上的灯)，光源会随着相机移动。

8.2.9　在 OpenGL 中使用材质属性

你只能通过光从物体表面反射看到光源的效果。OpenGL 提供了设定在等式(8.7)中出现的各种反射系数的方法。这些参数通过调用函数 glMaterial 的不同版本进行设置，并且还可以单独地对正面和背面的参数进行设置(见关于图 8.23 的讨论)。例如，

```
GLfloat myDiffuse[ ]={0.8, 0.2, 0.0, 1.0};
glMaterialfv(GL_FRONT, GL_DIFFUSE, myDiffuse);
```

为随后所有正面设置了漫反射系数($\rho_{dr}, \rho_{dg}, \rho_{db}$)=(0.8,0.2,0.0)。反射系数通过 RGBA 格式的四元组设置，这点和颜色一样。第一个传递给 glMaterialfv()的参数可以选择：

- GL_FRONT：为正面设置反射系数。
- GL_BACK：为背面设置反射系数。
- GL_FRONT_AND_BACK：为两面同时设置反射系数。

第二个参数可以选择：

- GL_AMBIENT：设置环境光反射系数。
- GL_DIFFUSE：设置漫反射系数。
- GL_SPECULAR：设置镜面反射系数。
- GL_AMBIENT_AND_DIFFUSE：设置环境光和漫反射系数为相同的值。这个参数方便了一般的使用，因为环境光和漫反射系数通常都会设置成一样的值。
- GL_EMISSION：设置表面的辐射颜色。

最后的一个参数是设置面片的**辐射颜色**(**emissive color**)，就是面片发出“辉光”的颜色，它独立于所有的光源。

把所有的放在一起

我们现在扩展等式(8.7)把 OpenGL 实际计算光的所有附加贡献添加进来。全部的红色分量这样计算：

$$I_r = e_r + I_{ar}\rho_{ar} + \sum_i atten_i \times spot_i \times (I_{ar}^i\rho_{ar} + I_{dr}^i\rho_{dr} \times lambert_i + I_{spr}^i\rho_{sr} \times phong_i^f) \tag{8.12}$$

对于绿色和蓝色分量的表达式类似。辐射光是 e_r，I_{ar} 是全局环境光，光照模型中介绍过它们。求和式说明所有光源的全局光、漫反射和镜面反射的贡献最后是要相加在一起。对于第 i 个光源，$atten_i$ 是等式(8.10)中的衰减因子，$spot_i$ 是探照灯因子(见图 8.22)，$lambert_i$ 和 $phong_i$ 是我们熟悉的漫反射和镜面反射的点积。所有这些项对于每个光源都要被重新计算。

注意：如果 I_r 出现了大于 1.0 的值，*OpenGL* 会把它剪裁到 1.0：所有最亮的光分量只能是 1.0。

8.2.10 渲染 SDL 设定的场景

场景描述语言(scene description language，SDL)在第 5 章和附录 3 做过介绍，它支持读取某个材质属性到物体，这样物体就能够被适当地着色。例如，

```
Light 3 4 5 0.8 0.8 0.8 ! bright white light at (3, 4, 5)
background 1 1 1 ! white background
globalAmbient 0.2 0.2 0.2 ! a dark gray global ambient light
ambient 0.2 0.6 0
diffuse 0.8 0.2 0.1 ! red material
specular 1 1 1 ! bright specular spots-the color of the sourse
exponent 20 ! set the Phong exponent
scale 4 4 4 sphere
```

描述了一个包含球体的场景，球体的材质属性是(见等式(8.7))：

- 环境光反射系数：$(\rho_{ar}, \rho_{ag}, \rho_{ab}) = (0.2, 0.6, 0)$
- 漫反射系数：$(\rho_{dr}, \rho_{dg}, \rho_{db}) = (0.8, 0.2, 1.0)$
- 镜面反射系数：$(\rho_{sr}, \rho_{sg}, \rho_{sb}) = (1.0, 1.0, 1.0)$
- 和 *Phong* 模型指数：$f = 20$

光源的漫反射和镜面反射分量的颜色都是(0.8，0.8，0.8)。还有全局环境光(I_{ar}，I_{ag}，I_{ab})=(0.2，0.2，0.2)。

当物体被创建时，当前材质属性被读入到物体的 mtrl 域(参见附录 3 和网上的源代码 Shape.cpp 中 Scene::getObject()的结尾部分)。当一个物体通过自身的 drawOpenGL()方法画其自己的时候，它首先将自己的材质属性传递给 OpenGL(参见 Shape::tellMaterialsGL()),所以当它真的被进行绘制的时候，OpenGL 的当前状态就是这些属性值。

我们在第 12 章介绍光线跟踪的时候，也会类似地使用每个物体的材质数据，查询这些

材质属性并且做适当的渲染。

8.3　平面着色与平滑着色

不同的物体需要不同的着色效果。在第 6 章中我们使用多边形网格来为许多不同的形状建模。对于其中的一些，像谷仓或者多面球体，我们希望在图像中看到每个单独的面片。但是对于另外的一些，像球体或者国际象棋中的卒，我们希望看到用面片逼近的表面，同时不想看到其中的边。

在建模过程中，对于每个面片的每个顶点我们都附加上了法向量。重复一下其中的要点，如果某个面片被看成一个独立的多边形，我们给它所有的顶点都附加上了同样的法向量。该法向量就是该面片所在平面的法方向。另外一方面，如果面片是用来拟合一个潜在的曲面，我们为每个顶点附加被拟合曲面在该点处的法向。

我们现在看看每个顶点上的法向量信息是如何在各种不同的渲染中被使用的。其中最主要的区别是一个方法强调单个的多边形（平面着色），另一个方法通过融合面片来降低边的突出程度（平滑着色）。平滑着色又有两种方法，称为 Gouraud 和 Phong 明暗处理。我们两种方法都要讨论。注意一些作者更喜欢称其为“照明模型（illumination model）”而不是“着色模型（shading model）”。

对于两种着色方法，顶点都是通过图形管线和光照的计算为每个顶点附加上颜色信息，最终面片顶点转化为屏幕坐标，并且面片被逐像素地绘制出适当的颜色。

画一个面片

面片是在多边形填充过程中被着色的。填充一个多边形非常简单，虽然一个精心设计的高效率填充算法可能会非常复杂（见第 9 章）。这里我们只看基本的内容，侧重于每个像素的颜色是如何设定的。

多边形填充程序有时也被称为**填充器（tiler）**，因为它在多边形上一个像素一个像素地移动，同时为每个像素着上适当的颜色，就像人在地板上铺瓷砖一样。具体地讲，多边形内的像素是按照规定顺序被访问的，通常是多边形底部到顶部的扫描线顺序，每个扫描线内是从左到右。

我们假定这里所处理的多边形是凸多边形。只处理凸多边形的情况下，填充算法可以设计得很高效，因为每个扫描线上只有一段连续的像素在多边形中。大多数 OpenGL 的实现都利用了这一点，可以正确地填充凸多边形，但是并不保证对于非凸多边形也能正确地填充。参见练习中有关多边形凸性有关的内容。

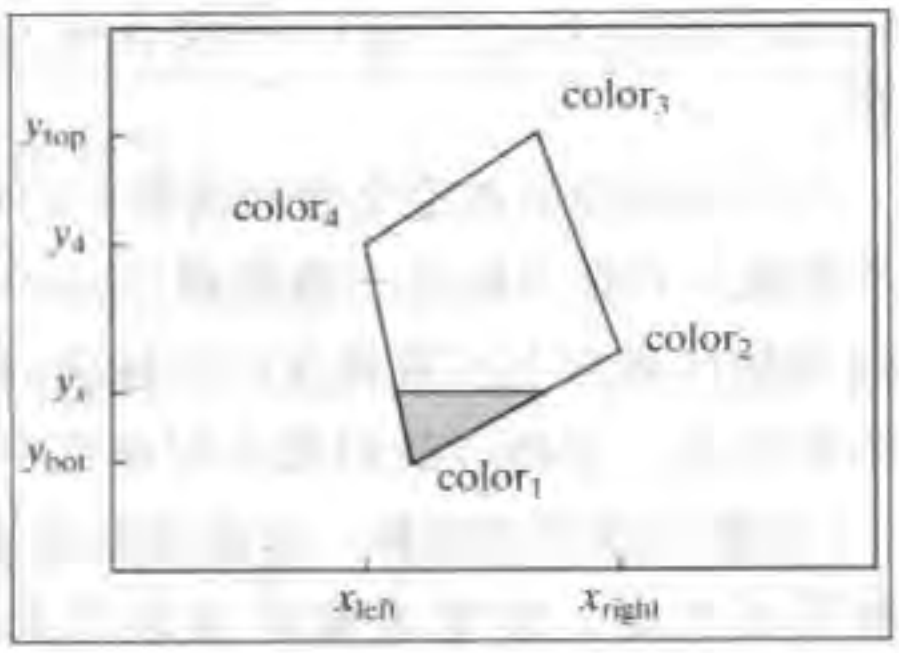


图 8.24　带颜色地填充多边形面片

图 8.24 是一个四边形面片的例子。每个顶点的屏幕坐标也标注了出来。面片的最低和最高点分别是 y_{bot} 和 y_{top}。填充过程先是填充 $y=y_{bot}$ 这行（这种情况下只是一个像素），然后是 $y_{bot}+1$ 行，以

此类推。在处理每个扫描线时,比如在图中的 y_s 这行,这有一个最左边的像素 x_{left} 和最右边的像素 x_{right}。填充算法从左 x_{left} 到 x_{right} 的移动把每个像素应有的颜色画上。所以填充算法的实现就是一个简单的双重循环,伪代码大致如下:

```
for (int y=ybott; y<=ytop; y++)          //for each scan-line
{
    <..find xleft and xright..>
  for (int x=xleft; x <=xright; x++)
      //for each relevant  pixel across this scan-line
    {
        <..find the color c for this pixel..>
        <..put c into the pixel at (x, y)..>
  }
}
```

在双重循环中隐藏面消隐也可以很容易地实现。平面着色和平滑着色的主要区别是每个像素颜色 c 的决定方法。

8.3.1 平面着色和马赫带

当面片是平的(比如谷仓的屋顶)并且光源相距很远的时候,屋顶上不同点的漫反射分量变化很小。等式(8.6)中的 *lambert* 项在面片的各个点上几乎相等,因为从每个点指向光源的向量几乎一样。在这种情况下为凸面片上的每个像素点都使用相同的颜色是合理的。OpenGL 提供了一种渲染模式可以为整个的面片渲染相同的颜色。虽然面片的每个顶点都会在流水线中携带一个颜色,但是绘制算法只会使用其中的一个(通常是面片的第一个顶点的颜色)。所以伪代码中的命令,<为这个像素计算颜色 c>,并不是在双重循环的内部,而是位于内重循环之前(译者注:这里应该是位于双重循环的外面),还要有"把 c 设置成某一个顶点的颜色"。(因为对每个像素使用同样的颜色,所以使得平面着色非常的快)。

OpenGL 中指定平面着色要使用:

```
glShadeModel(GL_FLAT);
```

图 8.25 就是使用平面着色渲染出的多面球体和球体。多面球体上每个单独的面片都像希望的那样清晰可见。而与此相反地,球体应该是一个平滑的物体,但是使用 glShadeModel(GL_FLAT)并没有使其平滑,因为整个面片都只用其中一个顶点的颜色着色。

有一种还没有被完全解释清楚的幻觉,使得面片之间的边看上去比其实际情况更加明显,这种现象在视觉中称为**一侧抑制**(lateral inhibition)。Ernst Mach 首先描述了这种现象:当光强在物体表面上出现梯度的不连续,视觉就会在梯度不连续的地方产生**马赫带**,人就能看到清晰的边。这种边的幻觉不用很集中精神就能被看到。

(观察马赫带的实验:选择多面球体的一个面片和一个边,要求其左边相邻面片比其右边相邻面片暗。你能否看到靠近边并且与边平行的窄带,左边的这个颜色——很奇怪地——要比右边的深?这可能不太容易看出来)这个效应夸大了使用平面着色的渲染结果,

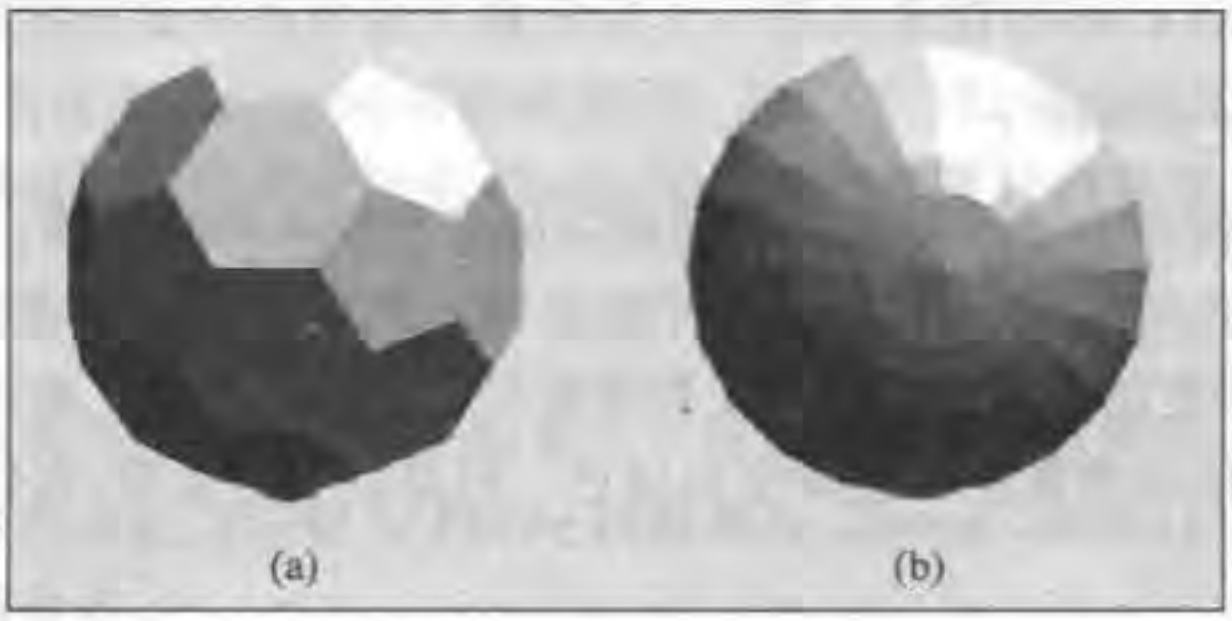


图 8.25　使用平面着色渲染的两个网格
(a) 多面球体；(b) 球体

使其看上去更加像网格模型。马赫带在图 b 中的球体上也能被很清楚地看到。

使用平面着色镜面高光的渲染结果就更差了，还是因为整个面片都是使用一个顶点计算出的颜色。如果恰好一个很强的镜面反射分量处于填充所选取的顶点上，则整个面片都会填充上这个亮度。如果镜面高光没有落在代表顶点上，那整个面片都没有高光。正是这个原因，一般不会在计算着色的时候考虑镜面反射分量。

8.3.2　平滑着色

平滑着色试图通过计算面片上多个点的颜色来弱化面片之间的边。这有两个主要的平滑着色方法，Gouraud 明暗处理和 Phong 明暗处理【Gouraud71，Phong75】。这些技术更准确地应该被称作“Gouraud 和 Phong 插值”，因为它们对于不同的点都是在混合颜色。OpenGL 只使用了 Gouraud 明暗处理，但是我们两种方法都会介绍。

Gouraud 明暗处理对于每个像素计算不同的颜色。对于在 y_s 处的扫描线（见图 8.24），首先通过左边最上方和最下方顶点的插值，找到最左边像素点的颜色 $color_{left}$。对于扫描线 y_s 来说，上方顶点的颜色是 $color_4$，下方顶点的颜色是 $color_3$，所以 $color_{left}$ 通过下式来计算（回顾等式(8.9)）：

$$color_{left} = \mathrm{lerp}(color_1, color_4, f) \tag{8.13}$$

其中 f 是 y_s 在 y_4 和 y_{bot} 之间距离所占的比例。

当 y_s 从 y_4 变化到 y_{bot}，f 值从 0 变到 1。注意等式(8.13)实际上包含三个计算对应红、绿和蓝三个分量。

类似地，再通过最上方和最下方的顶点颜色插值出 $color_{right}$。则填充过程就可以沿着扫描线，在 $color_{left}$ 和 $color_{right}$ 上线性插值得到像素 x 的颜色：

$$c(x) = \mathrm{lerp}\left(color_{left}, color_{right}, \frac{x - x_{left}}{x_{right} - x_{left}}\right) \tag{8.14}$$

为了提高效率，每个像素的颜色采用增量计算。也就是说 $c(x+1)$ 和 $c(x)$ 之间的差是一个常数，所以

$$c(x+1) = c(x) + \frac{color_{right} - color_{left}}{x_{right} - x_{left}} \tag{8.15}$$

Gouraud 明暗处理比平着色要耗费多一些的计算。OpenGL 中设置 Gouraud 明暗处理

使用：

```
glShadeModel(GL_SMOOTH);
```

图 8.26 展示了使用 Gouraud 明暗处理渲染的多面球体和圆球体。多面球体看上去和图 8.25 中用平面着色的结果一样，因为每个面片顶点的颜色都一样，所以插值不做任何改变。但是球体看上去更加平滑了。颜色在相邻面片之间没有突变。面片的边(还有马赫带)都不见了，取而代之的是物体上平滑渐变的颜色。但是沿着外轮廓，你仍然能够看到一些面片的外边缘。

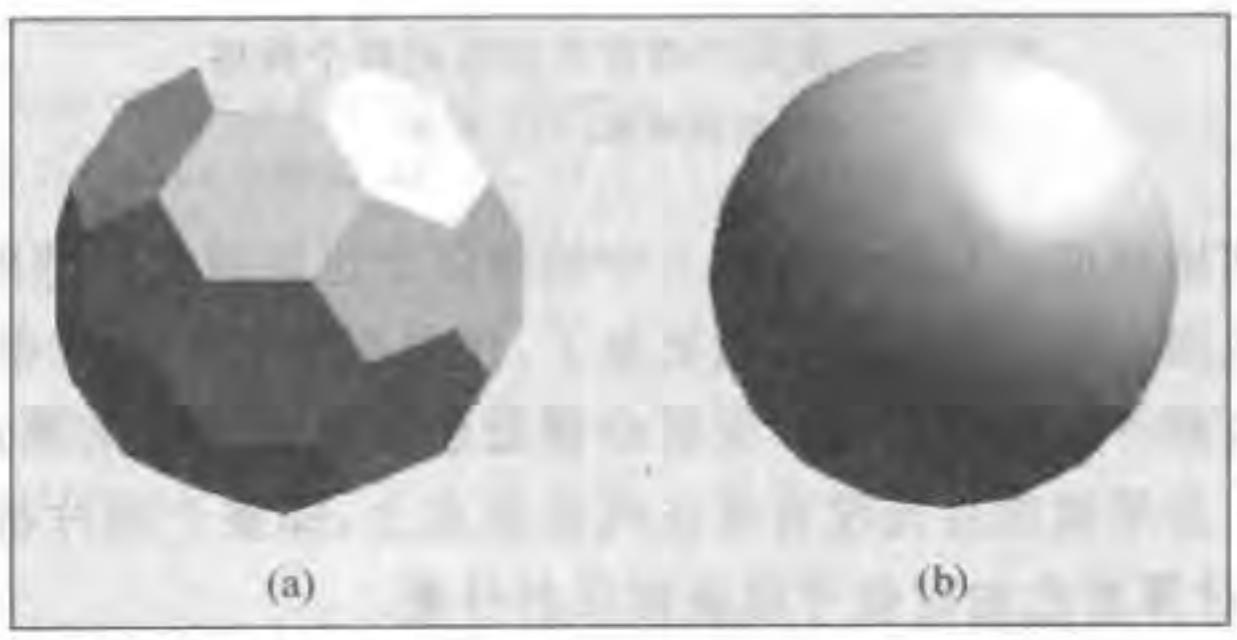


图 8.26 使用平滑着色渲染的两个网格

(a) 多面球体；(b) 球体

为什么使用这个技术后便消失了？图 8.27(a)画出了两个面片 F 和 F'，它们共享一个边而且假定物体想渲染出单独的面片。当渲染 F 的时候，使用了颜色 c_L 和 c_R，当渲染 F' 的时候，使用的是颜色 c'_L 和 c'_R。但是因为 c_R 等于 c'_L，所以在边的附近沿扫描线出现了颜色明显的变化，这也是我们希望的结果。计算 c_L 和 c_R 时所用的法向是通过平均与这个边相邻的面的法向得到的。

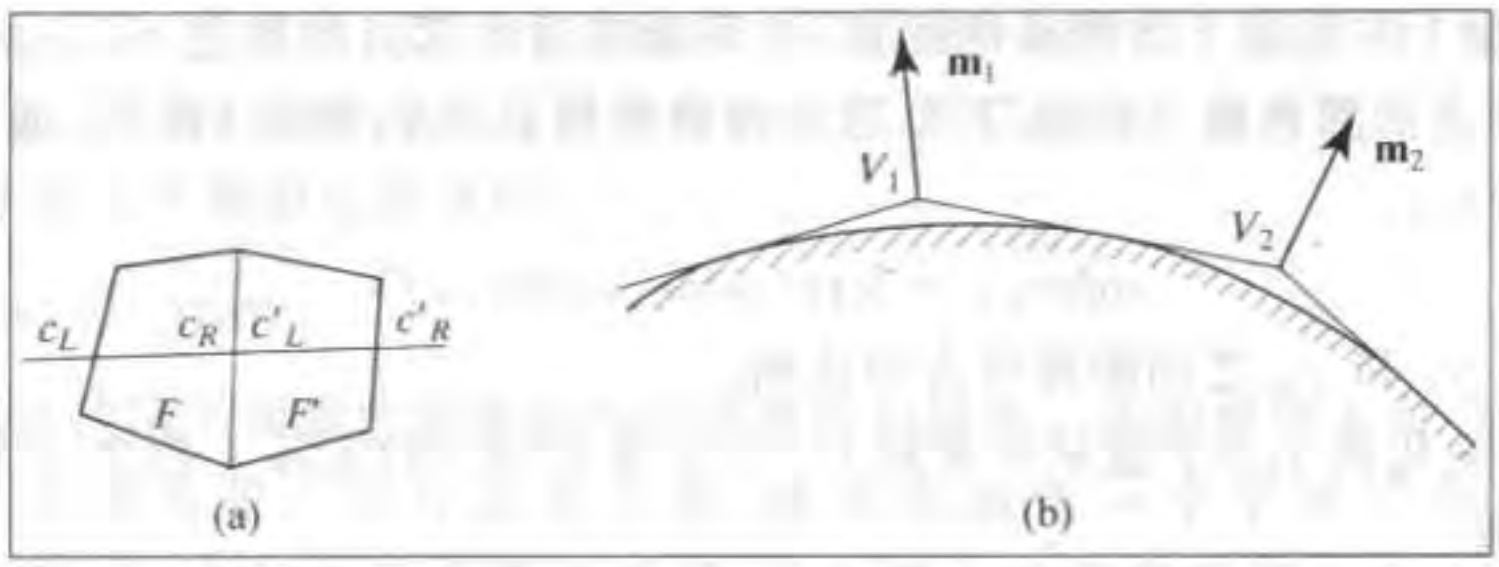


图 8.27 多边形两侧颜色的连续性

(a) 有明显边缘的物体；(b) 平滑曲面的物体

图 8.27(b)解释了这个技术如何显示用网格逼近的潜在曲面，这个时候我们想得到一个平滑物体的图像。图中有一个标注了 V_1 和 V_2 等的多边形表面，同时用这个多边形面片逼近的假象的平滑曲面也画了出来。顶点法向 $\mathbf{m}_1$、$\mathbf{m}_2$ 等的计算要使其垂直于该点所在的假象曲面，这样在每个顶点上用这个法向进行渲染就是正确的，因此计算出的颜色也是正确的。这样产生的顶点颜色就非常平滑，这只是一个简单的基于插值的数学定律而不是物理定律。

因为颜色是通过插值而不是在每个像素点上计算出来的，Gouraud 明暗处理不能很好地处理高光。因此，当使用 Gouraud 明暗处理时，等式(8.12)中光强的镜面反射分量经常被去掉。高光可以通过下面将要介绍的 Phong 明暗处理来更好的重现。

Phong 明暗处理

通过更好地逼近面片上像素处的法向量可以得到更加真实的效果——特别是对于高光和有光泽的物体。这种着色方法称为 **Phong 明暗处理**(Phong Shading)，以其发明者 Phong Bui-tuong【Phong75】命名。

计算 Phong 明暗处理时，我们要找到面片上每个点处的法向量，然后再在那里使用着色模型计算颜色。我们计算每个像素点处的法向量的方法是插值多边形顶点的法向量。

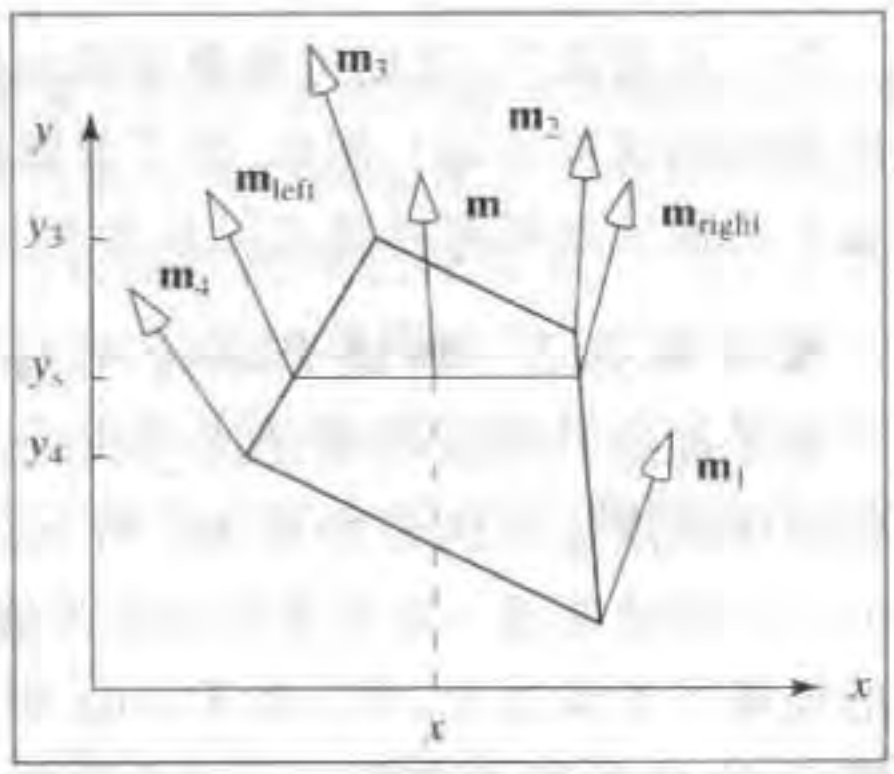


图 8.28　法向插值

图 8.28 画出了一个投影后的面片，四个顶点处标注了法向量 $\mathbf{m}_1$、$\mathbf{m}_2$、$\mathbf{m}_3$ 和 $\mathbf{m}_4$。对于所示的扫描线 y_s，向量 $\mathbf{m}_{left}$ 和 $\mathbf{m}_{right}$ 通过线性插值得到。

插值后的向量必须要标准化为单位向量，因为它要被用在渲染等式中。一旦 $\mathbf{m}_{left}$ 和 $\mathbf{m}_{right}$ 已知，它们就可以被插值得到扫描线上每个位置 x 处的法向量。这个向量被单位化后就可以在着色计算中使用，以得到这个像素的颜色。

图 8.29 对比 Gouraud 和 Phong 明暗处理的效果。因为各点之间法向量方向的变化非常平滑，也更接近于潜在的平滑曲面，所以和 Gouraud 明暗处理相比，Phong 明暗处理产生的高光更加真实，最终的渲染结果也更真实。

图 8.29　对比

(a) Gouraud 明暗处理；(b) Phong 明暗处理

Phong 明暗处理的主要缺点是每个像素上的计算量，因此 Phong 明暗处理所用的时间可能是 Gouraud 明暗处理的 6～8 倍。一些加速这个处理的方法也相继被提出来【Bishop86，Claussen90】。

OpenGL 不能够设置使用 Phong 明暗处理，因为它只在模型视点变换后的每个顶点上使用一次光照模型。法向量这个信息并不会传递到投影变换和投影剪裁步骤之后的渲染这

步。但是,在 8.5 节我们将看到一种逼近 Phong 明暗处理的方法,它使用环境贴图技术将高光纹理贴到物体上。

练习

练习 8.3.1 填充面片

完成多边形填充算法的操作细节,多边形顶点$(x,y)=(23,137),(120,204),(200,100),(100,25)$,对于扫描线 $y=136$, $y=137$ 和 $y=138$。写出每种情况下 x_{left} 和 x_{right} 的值。

练习 8.3.2 *Gouraud* 明暗处理中保留边

在一些情况下,我们可能希望画出模型中特定的褶皱和边。讨论如何通过选择顶点的法向量控制这样的边?例如,为了保留图 8.27 中的面片 F 和 F'之间的边,顶点法向应该是什么?其他的技巧和问题可以在参考文献中找到,如【Rogers85】。

练习 8.3.3 加速 *Phong* 明暗处理的篱笆明暗处理(*fence shading*)

为了加速 Phong 明暗处理,Behrens【Benhrens94】提出了一种插值方法,在每个扫描线上仍然用顶点法向插值得到 $\mathbf{m}_L$ 和 $\mathbf{m}_R$,然后计算这个最左和最右像素点的颜色,随后像 Gouraud 明暗处理一样沿着扫描线插值这两个颜色。这种称为"篱笆明暗处理"的方法大幅度的提高了渲染速度,而且和 Phong 明暗处理相比高光的处理效果只是略微下降。描述一下图 8.28 中的顶点法向 $\mathbf{m}_1$、$\mathbf{m}_2$、$\mathbf{m}_3$ 和 $\mathbf{m}_4$ 的方向,使得

a. 篱笆明暗处理产生和 Phong 明暗处理一样的高光

b. 篱笆明暗处理和 Phong 明暗处理产生的高光非常不一样

练习 8.3.4 *Phong* 明暗处理算法

在逐像素着色代码上做必要的修改,使其加入 Phong 明暗处理。假设每个面片的顶点法向量已知。同时讨论使用 OpenGL 的平滑着色算法如何逼近 Phong 明暗处理。提示:增加模型中的面片数量。

8.4 增加隐藏面消隐

如果有足够的内存作为**深度缓存**(depth buffer)(也称为"z-缓存","z-buffer"),在我们前面讨论的渲染过程中加入隐藏面消隐是非常简单的。因为它能够很容易地融入我们讨论的渲染机制,所以我们到这步才介绍它。

8.4.1 深度缓存方法——OpenGL 使用的方法

深度缓存(或者 z-buffer)算法是一个最简单最容易实现的隐藏面消隐算法,它也是 OpenGL 所使用的方法。它的主要局限是需要大量的内存,还有经常绘制那些会被随后的较近物体遮挡的物体(所以绘制较远物体的时间就被浪费了)。

图 8.30 画出了一个深度缓存和与其对应的帧缓存(frame buffer)。对于屏幕上每个像

素 p[i][j]，深度缓存一个 b 比特的量 $d[i][j]$。b 的值通常在 12～30 的范围内选取。

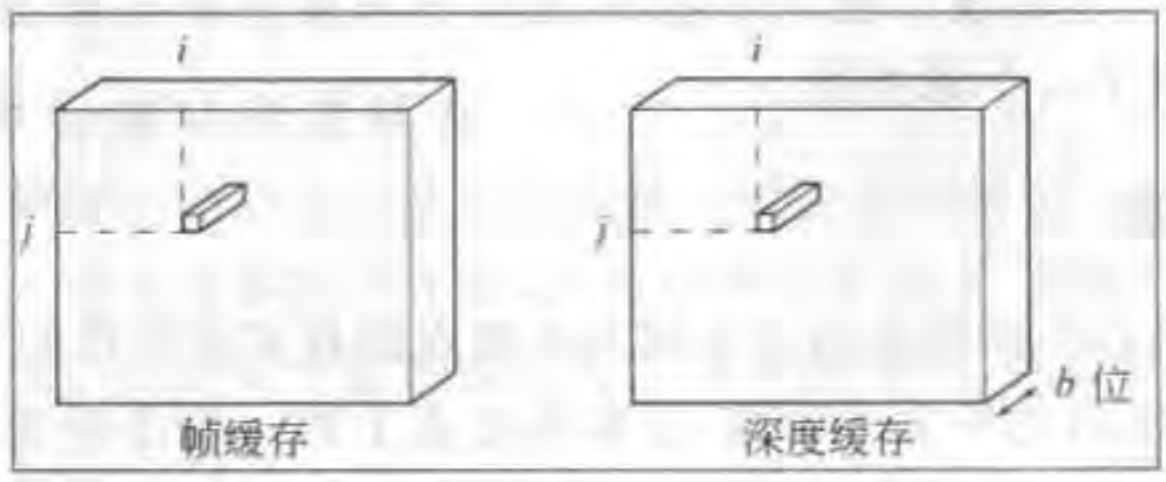


图 8.30　深度缓存示意

在渲染过程中，深度缓存值 $d[i][j]$ 存储这个像素（当前）遇到的最近物体的伪深度。随着面片扫描线上的像素一个个地被处理，一起测试当前面片的伪深度是不是小于该点深度缓存中存储的伪深度值 $d[i][j]$。如果小于，则用当前面片的颜色替代 $p[i][j]$，并且用这个更小的伪深度替换 $d[i][j]$ 中旧的深度值。面片可以以任意顺序绘制。如果远一些的面片先被绘制，该面片的一些像素将被随后更近面片的颜色所替代，因此渲染这个较远面片的时间就浪费了。需要注意的是这个算法对于任何形状的物体都有效，包括曲面在内，因为它寻找最近的表面是基于逐点的测试。

综上所述

- 对于每个像素点 ij，$p[i][j]$ 保存屏幕上该点目前最近面片的颜色。
- $d[i][j]$ 保存屏幕上该点目前最近面片的伪深度值。

数组 $d[][]$ 初始值是 1.0，这是伪深度值可能的最大值。帧缓存初始值是背景颜色。

计算每个像素的伪深度值

我们需要一个快速计算每个像素伪深度的方法。回顾面片的每个顶点 $P=(P_x,P_y,P_z)$ 传入图形管线，经过一系列变换，每个顶点在视窗变换后可用的信息是三元组。这是下式的一个缩放和平移后的结果（见等式(7.2)）：

$$(x,y,z)=\left(\frac{P_x}{-P_z},\frac{P_y}{-P_z},\frac{aP_z+b}{-P_z}\right)$$

其中第三个分量就是伪深度值。常数 a 和 b 的选择方法，是使第三个分量对于近平面上的点 P 等于 0，对于远平面上的点 P 等于 1。为了更高的效率，我们希望能够增量地计算每个像素的伪深度值，这也是使用插值的方法，类似我们用等式(8.15)计算颜色时所做的。

图 8.31 中是一个正在沿扫描线 y 填充的面片。标注了各个点的伪深度值。顶点处的伪深度值 d_1、d_2、d_3 和 d_4 是已知的。我们要计算扫描线 y_s 上 d_{left} 的值 lerp(d_1，d_4，f)，和 d_{right} 的值 lerp(d_2，d_3，h)。我们还要通过 lerp(d_{left}，d_{right}，k)计算扫描线上像素(x，y)的伪深度值 d(h，k 的值是什么?)。问题是这样算出来的三维面片上相应点的伪深度值是"真"的么。

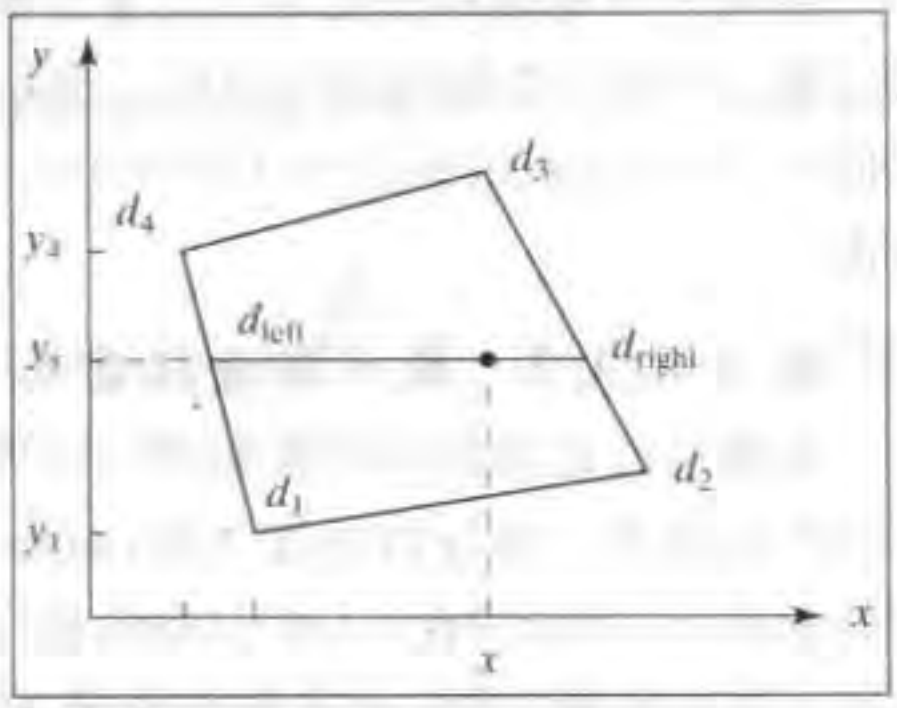


图 8.31　伪深度值的增量计算

答案是不完全地有效。如果原始的三维面片是平的,则计算就是对的,因为投影变换保证变换后还是平的。但是回顾以前介绍过的,这个伪深度值并不是随物体距视点的真实距离线性的变化,而是趋于一个渐近线。

远处的深度压缩

回顾 7.4 节,一个点的伪深度值不是随其距视点的真实值线性的变化,而是趋于一个渐近线。这意味着当深度值比较大的时候,真实深度值上的一个小的变化映射到伪深度值后变为极小的变化。因为表示伪深度值的比特数有限,两个接近的深度值有可能会被映射为相同的伪深度值,这就可能在进行 d<d[x][y]比较的时候导致错误。使用更多的比特来表示伪深度值可以解决这个问题,但是需要更多的内存。而且对于近平面距离视点比较远的这种可能情况,这样做的效果也很小。

OpenGL 支持一个深度缓存并且使用上述算法实现隐藏面消隐。你必须在初始化显示模式的时候指明使用深度缓存:

```
glutInitDisplayMode(GLUT_DEPTH| GLUT_RGB);
```

并且启用深度测试:

```
glEnable(GL_DETPH_TEST);
```

渲染每一幅图像之前要初始化深度缓存:

```
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);          //清屏
```

其他许多的隐藏面消隐方法可以在本书的配套网站上找到,可以作为深入阅读的资料。和本算法相比,那些算法复杂得多,并且必须被仔细地检查以保证对所有的情况都正确。

练习

练习 8.4.1 补充

补充从已知的顶点伪深度值计算 d_{left}, d_{right} 和 d 值的细节。

练习 8.4.2 深度值编码

假设为深度缓存中的每个元素分配 b 个比特。这 b 个比特用来记录从 0 到 1 之间的伪深度值。0 和 1 之间的值可以用二进制数 $.d_1d_2d_3\cdots d_b$ 表示,其中 d_i 是 0 或者 1。例如,伪深度值 0.75 可以被编码为 0.1100000000…这是不是一个使用 b 比特的方法?从正反两方面讨论。

练习 8.4.3 减少深度缓存的大小

如果没有足够的内存来实现一个完全的深度缓存,我们可以一片一片地生成图像。深度缓存用来为一部分扫描线工作,对每一部分图像反复调用算法。比如,在一个 512×512 的显示屏上,只能分配 64 个扫描线相应的深度缓存,所以要反复执行 8 次算法。每次所有的面片都被扫描,计算相应扫描线覆盖区域的面片深度,并且和当时的深度缓存比较。当然,所有的面片的 8 次扫描使得这个算法的运行很慢。假设一个场景有 F 个面片,每个面

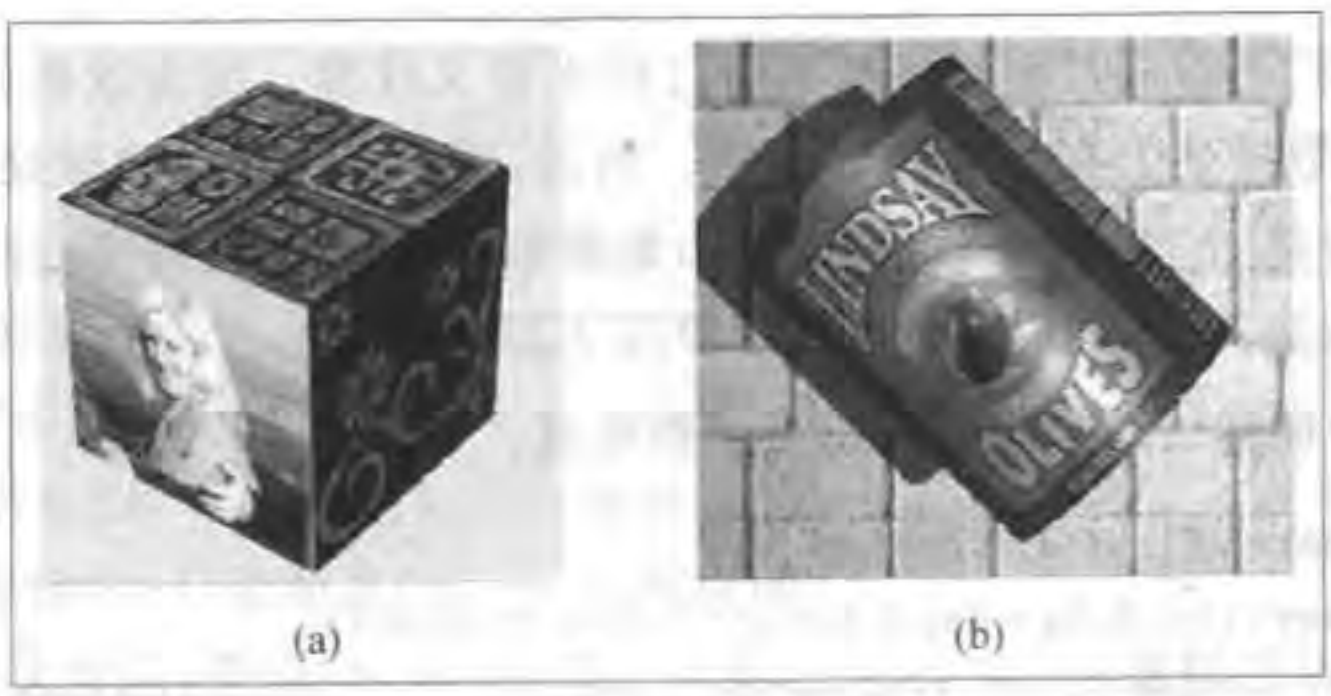
片平均覆盖 L 个扫描线。当分配 $nRows/N$ 扫描线的内存给深度缓存时，计算这个方法将多用多少时间？

练习 8.4.4　单扫描线深度缓存

上一个练习中把帧缓存的分块方法可以被推广到极限的情况：深度缓存只存储一行扫描线的深度。这看上去需要更多的计算量，因为每个面片因为"更新"需要被处理多次，每个扫描线都需要处理一次。讨论对于这种情况算法将如何修改，并且估计和一个全屏深度缓存相比这将要多用多长的时间？

8.5　为面片添加纹理

图像的真实感连同其对于观察者的吸引力，能够通过为网格物体的各个表面贴纹理而被显著增强。图 8.32 给出了一些例子。在图 8.32(a)中，一些图片被贴在立方体的表面上。在图 8.32(b)中，一个标签被弯曲地贴在圆柱罐上，背后的墙看上去也是由砖块砌成的。



(a)　(b)

图 8.32　物理映射示例

(a) 纹理映射到立方体表面；(b) 纹理包裹在一个罐头上，还有贴在罐头后面的墙上

基本的技术始于在"**纹理空间**(texture space)"中定义的一组纹理函数，比如图 8.33 所示的那些。纹理空间中的参数按习惯以 s 和 t 命名。纹理就是一个函数 texture(s,t)，对于 s 和 t 在 0 到 1 之间的取值产生一个颜色或者亮度值。

我们要看如何把纹理贴到物体的各个点上，但是首先我们先看看不同纹理的来源。

不同的纹理来源

纹理最普遍的来源是位图(矩阵一样的像素数组)和生成函数。图 8.33(a)是一个位图的例子，对应的图 8.33(b)是一张通过一些函数计算生成的(人造)图片。

位图纹理

位图经常来源于位图表示的图片(比如数字照片、数码相机图片、剪切画或者用一些程序生成的图片)。另外，互联网也提供了数以万计的有趣纹理。这种纹理由二维数组组成，

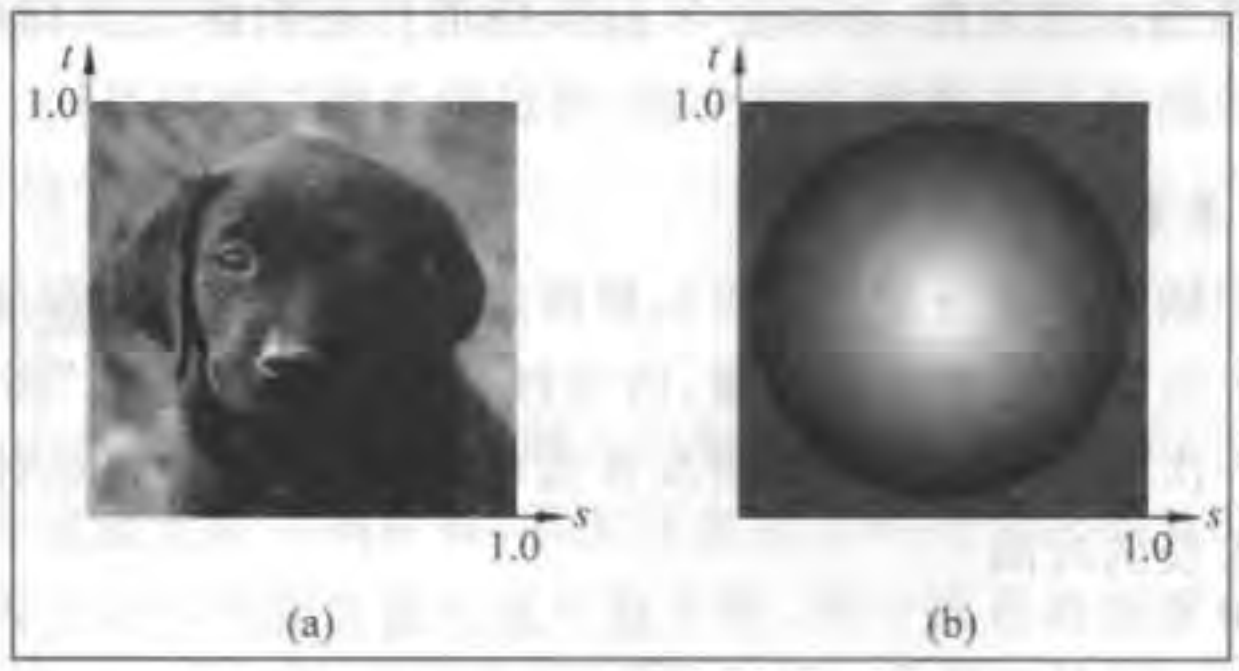


图 8.33 纹理示例

(a) 图片纹理;(b) 生成的纹理

记为 txtr[r][c],存储着颜色值(经常称为"**纹理元素(texels)**")。我们的第一个任务是把数组 txtr[r][c]中的值适当地和 0 到 1 之间的 s、t 值联系起来。假设数组 *txtr*[][]有 c 列和 r 行。

纹理映射的过程化生成

除此之外,我们可以通过数学函数或者过程来定义纹理。因为这些函数通常都由编程人员创建,所以函数描述上有很大的自由度。例如,图 8.33(b)中的球处于图片中心,圆心位于二维坐标的$(s,t)=(0.5,0.5)$,半径是 0.5,所以从点(s,t)到其中心的距离是:

$$\sqrt{(s-0.5)^2+(t-0.5)^2}$$

图 8.31(b)中的球纹理可以通过以下函数生成:

```
float fakeSphere (float s, float t){
    float r=sqrt((s-0.5) * (s-0.5)+(t-0.5) * (t-0.5));
    if (r>0.3) return 1-r/0.3; else return 0.2;}        //0.2 is the background
```

这个函数值域从位于中心的 1(白色)到位于(顺着轴线)边缘的 0(黑色),组成一个圆。在处理函数中的值 0.3 使得球内最暗的亮度能和背景亮度平滑地融合。练习中给出了另外一个模仿国际象棋棋盘的纹理。任何输入是各个 s 和 t 值的函数都可以被称为纹理:平滑融合和卷曲的颜色,Mandelbrot 集合(见附录),固体的线框绘制等等。

我们后面会看到有多种方法来使用 texture(s,t)的值:如果面片发光可以被用来当面片的颜色;可以被用来当做环境光、漫反射或者镜面反射系数去调和面片反射光的比例;可以用来改变表面的法向量而使物体产生凹凸的外表。

练习

练习 8.5.1 经典的棋盘纹理

图 8.34 给出了一个棋盘,它是由明暗相间的 5×5 的方格组成,亮度是 0(黑色)和 1(白色)。

为这个纹理写出函数 float texture(float s, float t)(参见练习 2.3.1)。

在有了纹理函数之后,下一步是如何把它适当地映射到物体表面,然后从相机中看到

它。图 8.35 中的例子展示了所有的问题。这里一个简单的纹理样例被映射到两个不同的物体：一个平面多边形和一个圆柱。对于每个物体，都需要有个变换，称为 T_{tw}（从纹理坐标到世界坐标）。这个变换把纹理(s, t)的值映射到物体表面的点(x, y, z)上。相机从某个角度拍摄场景，产生最后看到的结果。我们把点从三维坐标变换到屏幕的变换称为 T_{ws}（从世界坐标到屏幕坐标），则物体表面上一点(x, y, z)被看到的像素位置是$(s_x, s_y)=T_{ws}(x, y, z)$。所以综合起来，纹理上的值$(s^*, t^*)$最后到达像素是$(s_x, s_y)=T_{ws}(T_{tw}(s^*, t^*))$。

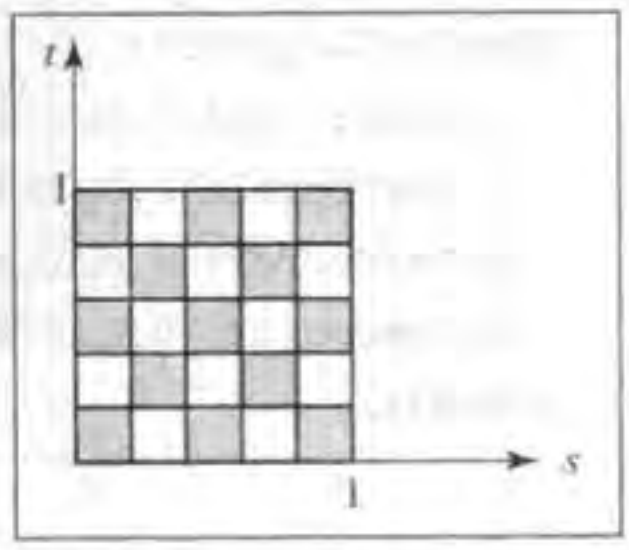


图 8.34　经典的棋盘纹理

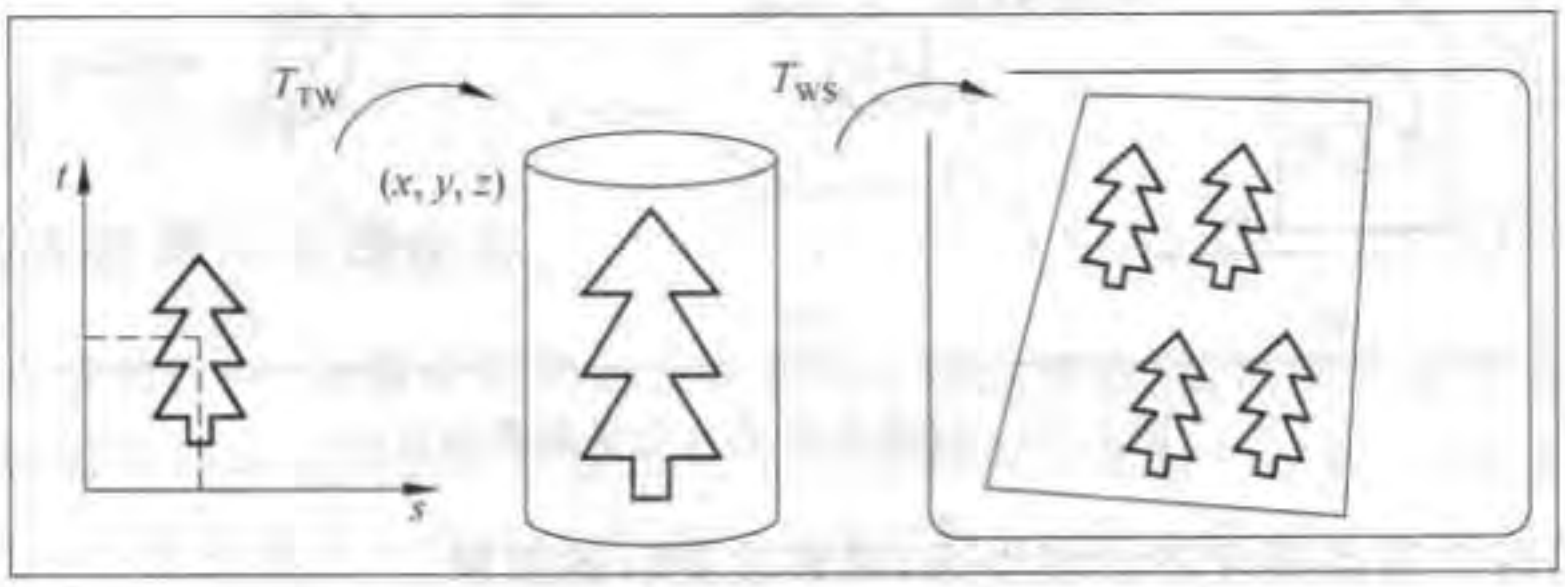


图 8.35　纹理贴在一个圆柱体和一个平面上

渲染过程实际上是另一个途径：对于像素(s_x, s_y)这有一系列的问题：

a. 从(s_x, s_y)看到的最近曲面是什么？回答这个问题需要解决隐藏面消隐问题，这样才能决定哪个纹理才是相关的。

b. 曲面上哪个点(x, y, z)和(s_x, s_y)是对应的？

c. 哪个纹理坐标(s, t)是和点(x, y, z)对应的？

所以我们需要反变换，类似$(s, t) = T_{tw}^{-1}(T_{ws}^{-1}(s_x, s_y))$，这样给出像素的坐标就能够得到$(s, t)$的坐标。反变换依据曲面的不同可能很容易得到也可能很难得到。

8.5.1　在平面上贴纹理

我们首先看看最重要的情况：映射纹理到一个平面。这是一个建模任务。然后我们在看过纹理是如何真正地被渲染出来的以后，解决视角的问题。最后我们讨论把纹理映射到更加复杂的形状表面。

把纹理贴到平面上

因为纹理空间本身是平面，所以把纹理贴到平面上是最简单的。图 8.36 给出了一个（在五边形中的）纹理图像映射到一个平面多边形（另外的一个五边形）F。我们必须指定 F 上每个点在纹理上的对应点。在 OpenGL 中，指定纹理空间中的点 $P_i=(s_i, t_i)$和面片上的顶点 V_i 的对应关系，使用函数 glTexCoord2f()。函数 glTexCoord2f(s,t)设置当前的纹理坐标为(s,t)，这个坐标就和随后定义的顶点联系到一起。通常每个 glVertex2f()调用前都要调用 glTexCoord2f()，这样每个顶点都有一对不同的纹理坐标。例如，为了定义一个四边形并且在

其上设置一个纹理,我们通过如下程序传给 OpenGL 4 个纹理坐标和 4 个三维顶点:

```
gBegin(GL_QUADS);        //define a quadrilateral and position our texture on it
  glTexCoord2f(0.0, 0.0); glVertex3f(1.0, 2.5, 1.5);
  glTexCoord2f(0.0, 0.6); glVertex3f(1.0, 3.7, 1.5);
  glTexCoord2f(0.8, 0.6); glVertex3f(2.0, 3.7, 1.5);
  glTexCoord2f(0.8, 0.0); glVertex3f(2.0, 2.5, 1.5);
glEnd();
```

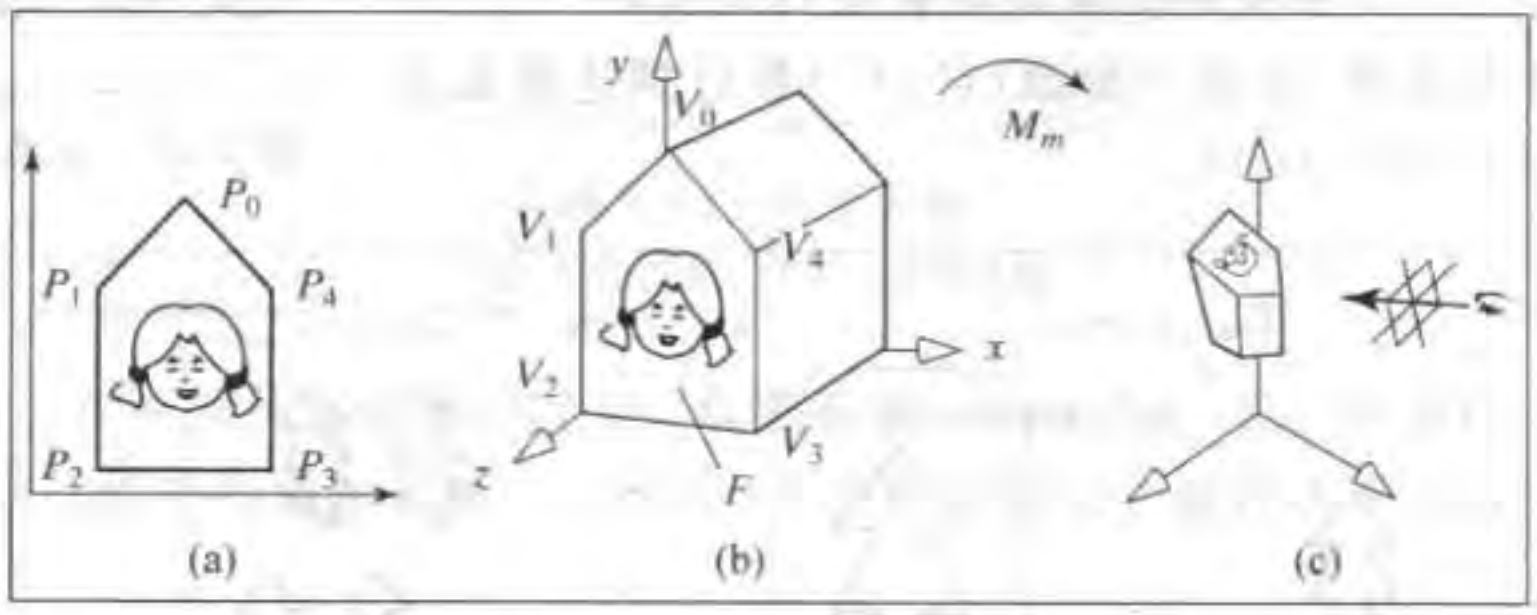


图 8.36 纹理映射到一个平面多边形

改变上面四个顶点相应的纹理坐标,观察改变后的效果。

为每个 V_i 设定 P_i 等价于在纹理空间定义了一个多边形 P,它拥有和 F 一样多的顶点。通常 P 和 F 的形状也是一样的:在 P 中的这部分纹理被无扭曲地贴到整个 F 上面。当 P 和 F 有相同的形状时,这个映射无疑是仿射变换:缩放,有可能伴随着旋转和平移。

图 8.37 给出了一个普通的情况,矩形纹理的四个角对应到三维场景中面片的四个角。(纹理坐标(s, t)赋值到对应的三维面片的拐角上。)在这个例子中,纹理是一个 640×480 像素的位图,并且被贴到长宽比为 640/480 的矩形上,所以看上去没有扭曲(注:纹理坐标 s 和 t 的变化范围依旧是 0 到 1)。图 8.38 使用的纹理坐标产生了反复平铺纹理的效果。为了做到这一点,一些纹理坐标的值处于[0,1]区间之外。当渲染器遇到 s 和 t 的值处于单位正方形之外的时候,比如 s=2.76,它忽略掉整数部分而只使用小数部分 0.76。这样面片上使用(s, t)=(2.6,3.77)为其纹理的点所帖的纹理是 texture(0.6, 0.77)。默认情况下 OpenGL 就是这样平铺纹理的。如果需要,它也可以替代设置使用"剪裁"纹理值的方法,参见练习。

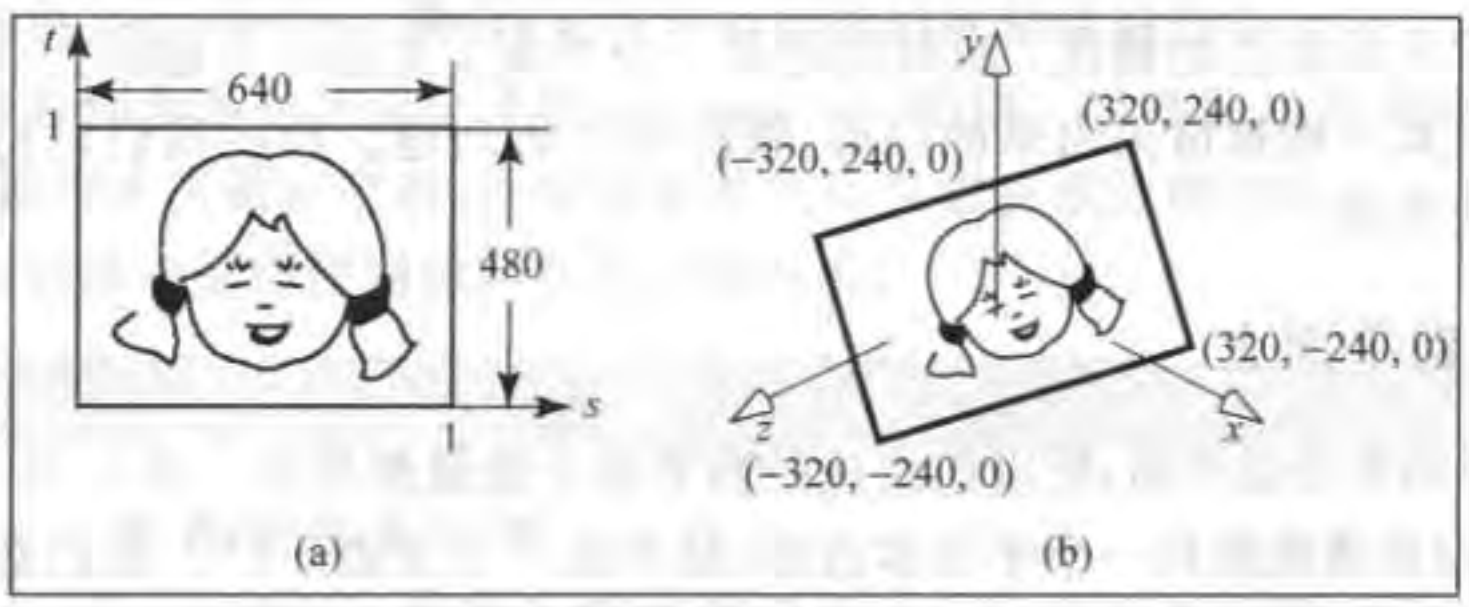


图 8.37 正方形映射到矩形

这样一个纹理坐标(s, t)跟随着面片的顶点沿流水线传递。下一节我们会介绍,F 中的点将被填充上 P 内对应的纹理值,而且使用插值计算所得到的纹理坐标(s, t)值。这个插

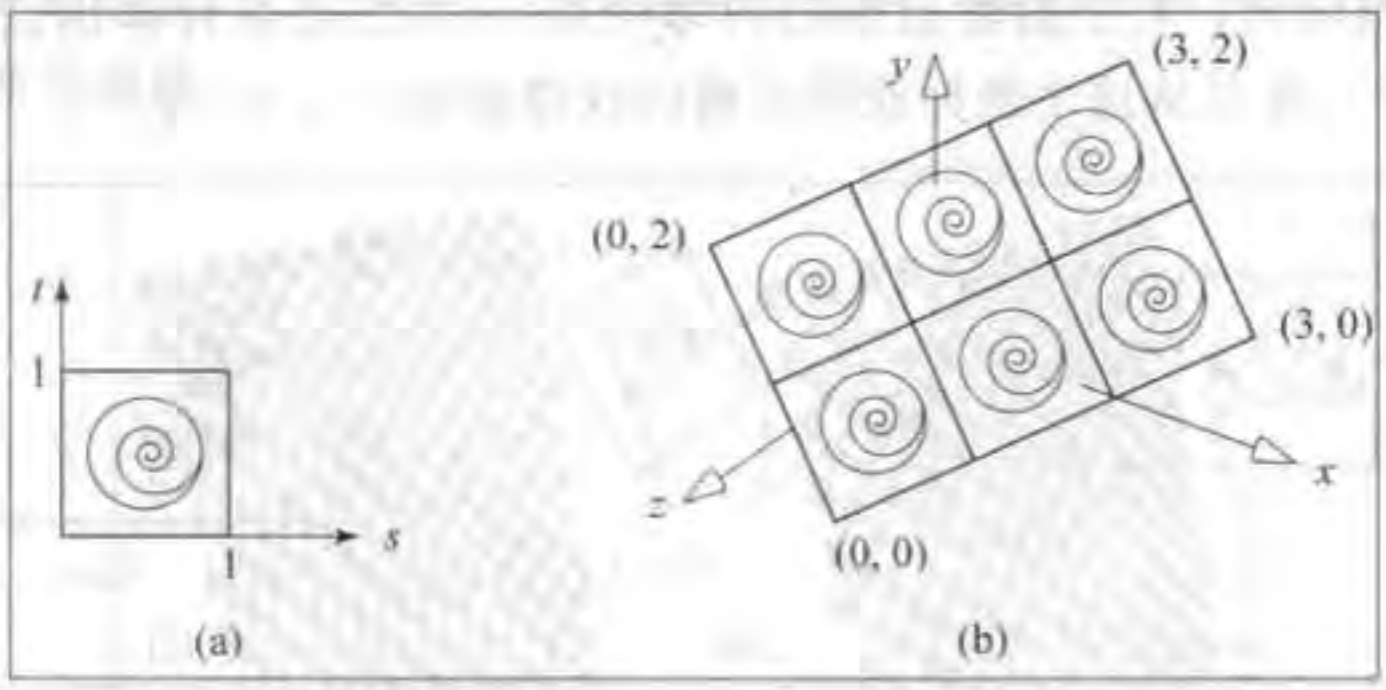


图 8.38　产生反复出现的纹理

值过程将在下一节介绍。

为网格模型添加纹理坐标

回顾第 6 章的代码，网格模型拥有三个列表：顶点列表、法向量列表和面片列表。我们必须在其中添加纹理坐标列表以存储不同的顶点对应的坐标(s_i, t_i)。为这目的我们添加一个数组，每个元素是类 TxtrCoord 中的一对浮点数，类定义是

```
Class TxtrCoord { public: float s, t; };
```

数组存储了所有网格的纹理坐标对。这有几种不同的处理模型纹理的方法，每种方法都要指明模型中纹理信息是如何组织的。最重要的两种方法如下。

1. 网格模型是由一些平面片组成，并且每个面片应用不同的纹理。这样每个面片只有一个单独的法向值，但拥有其自己的纹理坐标列表。所以每个面片的数据组织可以是：

- 面片顶点的数量
- 面片法向的索引
- 顶点索引的列表
- 纹理坐标索引的列表

2. 网格表示一个潜在的平滑物体，并且单个纹理包裹着这个物体(或者其一部分)。这样每个顶点都对应一个指定的法向量和一个特定的纹理坐标。因此使用一个顶点、法向和纹理坐标索引的列表就可以了。每个面片的数据组织可以是：

- 面片顶点的数量
- 顶点索引的列表

练习中会更加深入地探讨这两种类型网格所需要的数据结构。

8.5.2　渲染纹理

渲染面片 F 中的纹理和 Gouraud 明暗处理相似：渲染器在面片中一个像素一个像素地移动。对于每个像素它必须决定对应的纹理坐标(s, t)，访问纹理并且设置像素为适当的纹理颜色。我们将看到必须非常小心地寻找坐标(s, t)。

图 8.39 展示了相机拍摄帖有纹理的面片 F，并且正在进行渲染的过程。扫描线 y 正在

从 x_{left} 到 x_{right} 地被填充。对于沿着扫描线的每个点 x,我们必须计算出面片上正确的坐标(用 $P(x, y)$ 表示),并且从这个坐标获得正确的纹理坐标(s^*, t^*)。

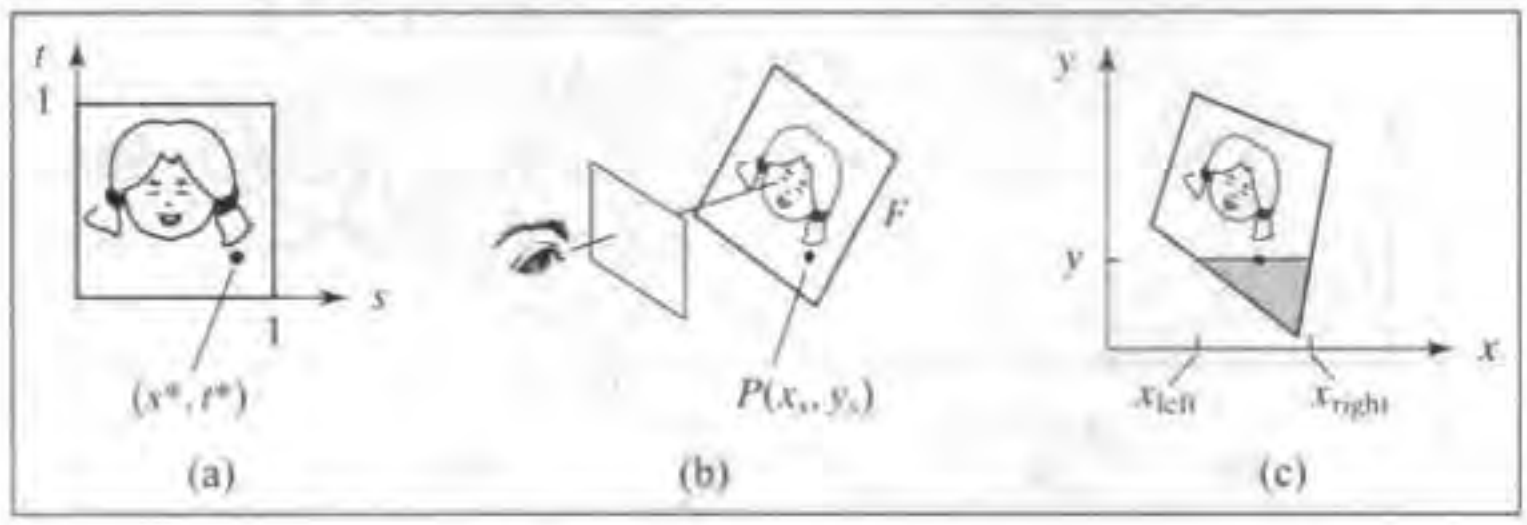


图 8.39 从视点看到的头像

在建立纹理到物体的映射之后,我们就知道面片 F 上每个顶点的纹理坐标,例如图 8.40 中的标注。自然的想法是为每个扫描线使用增量方法快速地计算(s_{left}, t_{left})和(s_{right}, t_{right}),然后在扫描线上随着位置移动插值这些值。但我们必须小心:当我们在扫描线 y 上从 x_{left} 到 x_{right} 移动,简单地使用增量方法从 s_{left} 变到 s_{right} 是不行的。因为在投影后的面片上相等的间隔并不对应于三维面片上相等的间隔。

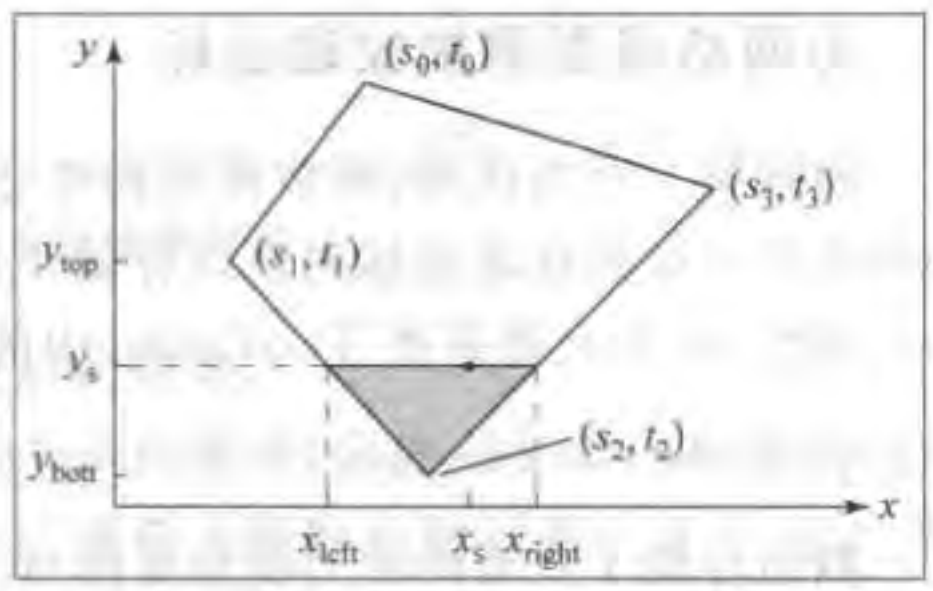


图 8.40 纹理坐标的增量计算

图 8.41 说明了这个问题。图 8.41(a)是一个左边比右边更接近观察者的面片 F。图 8.41(b)是这个面片在屏幕上的投影 F'。在扫描线 $y=170$ 上,我们等距地标出 F' 上的点代表面片上连续像素的位置。这些标记点对应到真正面片上的位置在图 8.41(a)中标出。可以看出这些点在距离 F 的远端会靠得更近,这就是立体视觉中简单的近大远小现象。

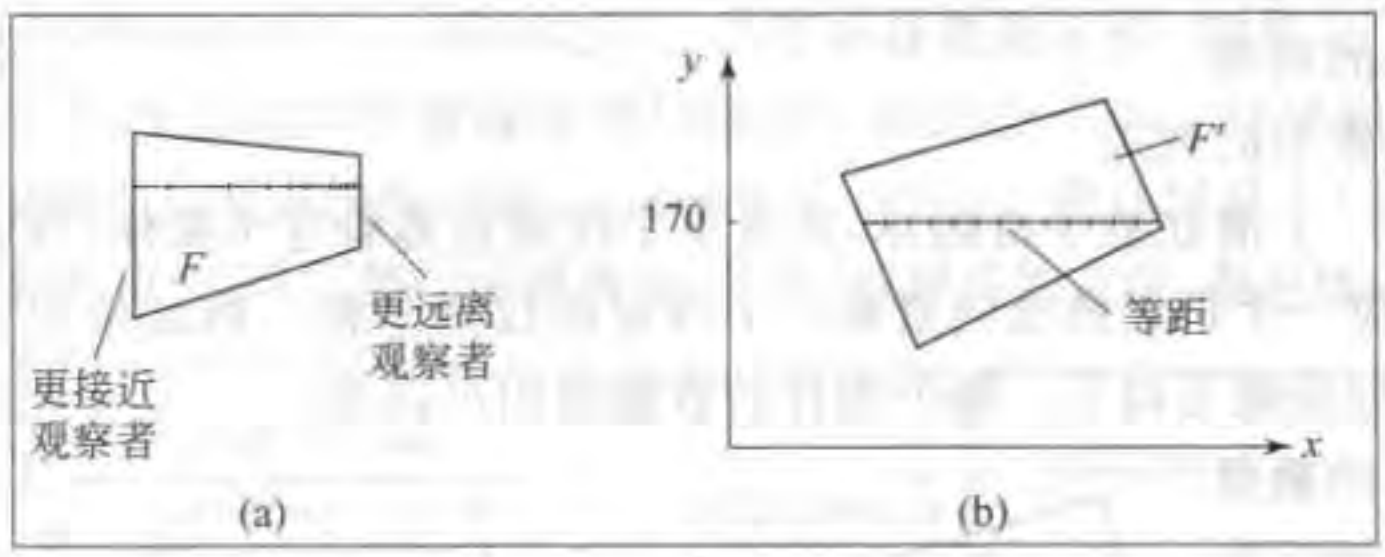


图 8.41 线性插值点的间距

如果我们使用简单的线性插值,等距地划分 s 和 t 的间隔来计算纹理坐标,我们就在错误的位置进行了纹理采样,得到一个图像扭曲的结果。图 8.42 说明了这样简单地映射棋盘纹理到一个矩形上会得到什么结果。图 8.42(a)中使用的线性插值产生了纹理上明显的扭曲。这个扭曲在多边形旋转的动画中会显得特别的混乱,因为纹理动态出现卷绕和拉伸。图 8.42(b)中使用了正确的插值,棋盘看上去就像其应该的那样。在动画中,这个纹理就会像紧紧地附着在旋转的面片上一样。一个完整的程序可以在网上(RotatingCube.cpp)找到,展示了一个 6 个面帖有不同纹理的旋转立方体。图 8.32 是这个旋转立方体程序的一个

截图。因为有正确的插值，所以每个面都没有扭曲变形。我们下面将要讨论如何写出正确的代码来实现正确的插值。

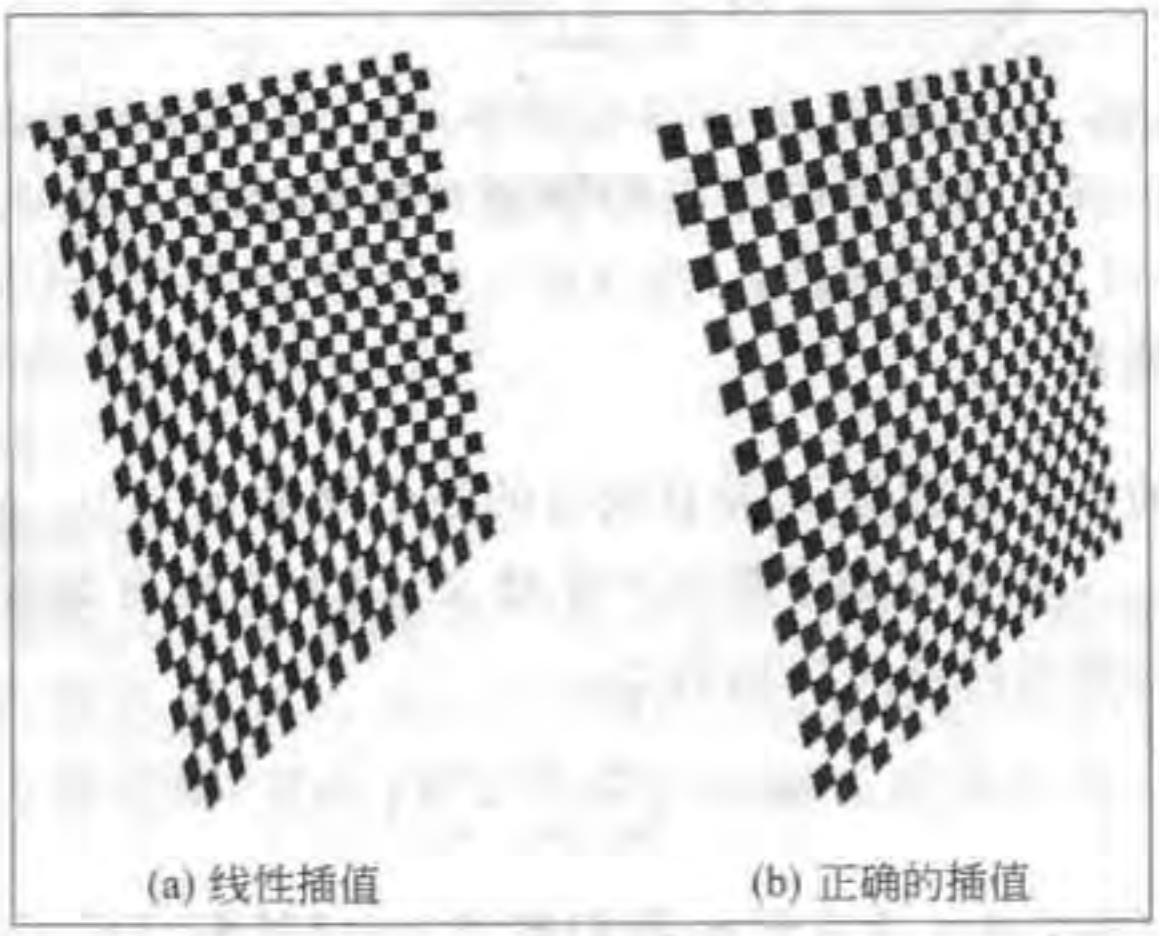


图 8.42 线性插值和正确插值的渲染结果

一些研究正确插值方法的文章已经被发表。Heckbert、Moretn【Heckbert91】和 Blinn【Blinn92】描述了一个基于仿射变换和投影变换本质的简洁推广。Segal 等人【Segal92】使用基于线段参数表达的更加代数化的推导得到了同样的结果。我们在此介绍后一种推导。

图 8.43 给出了我们将要分析的情况。我们知道仿射和投影变换保持直线性，所以在视点坐标系内直线 L_e 投影为平面坐标系下的直线 L_s。相似地，我们希望画在直线 L_t 上的纹理元素，在纹理坐标下是沿着直线 L_t 分布并且映射为 L_s。关键的问题是：如果我们在屏幕直线 L_s 上移动相等的距离，在纹理空间上纹理元素将沿 L_t 怎样移动？

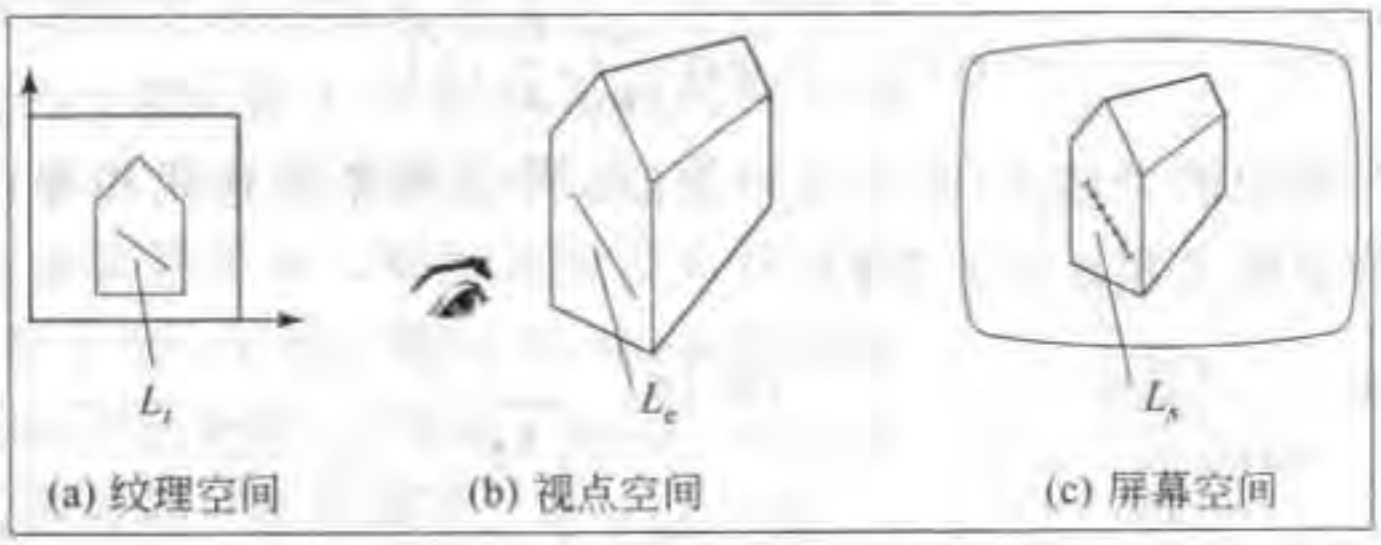


图 8.43 直线从一个空间映射到另一空间

我们在后面推导了一个一般的结果，总结如何进行插值：所做的都是和立体视觉分割的效果有关。然后我们把这个结果和图形管线中执行的变换联系起来，看看为了完成正确的纹理映射所需要的额外步骤在哪里。

图 8.44 画出了三维中的直线 AB 经过矩阵 M 的变换后变为三维中的直线 ab（M 可以表示一个仿射变换，或者是更一般的透视变换）。点 A 映射到 a，B 映射到 b。考虑点 $R(g)$ 位于由 A 到 B 中比例为 g 的位置。它被映射到位于由 a 到 b 中比例为 f 的某点 r(f)。我们将会看到比例 f 和 g 的值并不相等。问题是当 f 值从 0 变到 1，g 值将如何变化？这也就是沿 ab 的运动如何对应沿 AB 的运动？

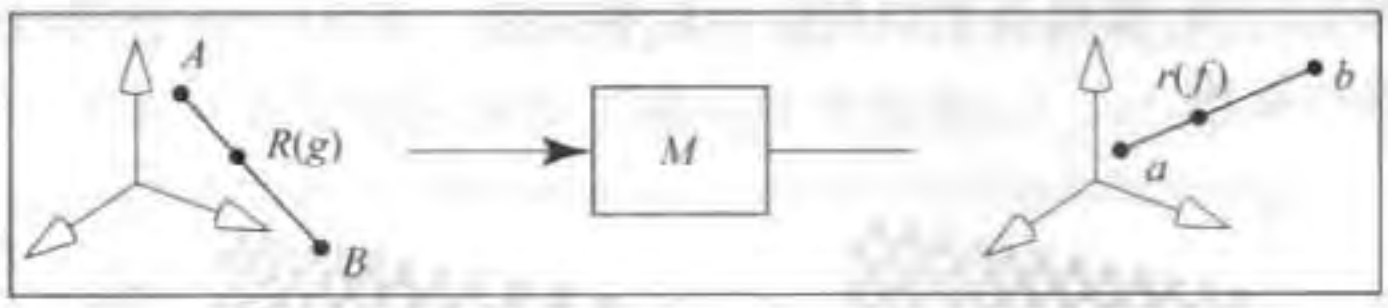


图 8.44 对应直线的运动如何变换?

推导 g 和 f 的关系

我们假定坐标 a 的齐次坐标是 $\tilde{a}$,并且设它的各个分量 $\tilde{a}=(a_1, a_2, a_3, a_4)$。(我们使用下标 1、2、3、4 替代 x、y 等等是为了防止产生歧义,因为这里有太多的"x, y, z"了)。所以通过对 $\tilde{a}$ 进行透视变换的除法可以得到点 a:

$$a = \left(\frac{a_1}{a_4}, \frac{a_2}{a_4}, \frac{a_3}{a_4}\right)$$

因为 M 映射 $A=(A_1, A_2, A_3)$ 到 a,我们知道 $\tilde{a}=M(A, 1)^T$,其中 $(A, 1)^T$ 是分量为 A_1、A_2、A_3 和 1 的列向量。类似地,$\tilde{b}=M(B,1)^T$(仔细检验这些关系)。为了保持简洁我们使用符号 lerp(),我们已经定义了 $R(g)=\text{lerp}(A, B, g)$,这点映射到 $M(\text{lerp}(A, B, g), 1)^T=\text{lerp}(\tilde{a}, \tilde{b}, g)=(\text{lerp}(a_1, b_1, g), \text{lerp}(a_2, b_2, g), \text{lerp}(a_3, b_3, g), \text{lerp}(a_4, b_4, g))$(检验这些关系)。这是点 $r(f)$ 的齐次坐标版本 $\tilde{r}(f)$,所以能通过透视变换的除法恢复真实的 $r(f)$ 的分量。为了简洁,我们只写出第一个分量 $r_1(f)$:

$$r_1(f) = \frac{\text{lerp}(a_1, b_1, g)}{\text{lerp}(a_4, b_4, g)} \tag{8.16}$$

但是因为定义 $r(f)=\text{lerp}(a, b, f)$,我们有第一个分量 $r_1(f)$ 的另外一个表达式:

$$r_1(f) = \text{lerp}\left(\frac{a_1}{a_4}, \frac{b_1}{a_4}, f\right) \tag{8.17}$$

$r_2(f)$ 和 $r_3(f)$ 相应的表达式(是什么?)类似。令这两个表达式相等,再做一些代数变换可以得到我们希望的 f 和 g 的关系:

$$g = \frac{f}{\text{lerp}\left(\frac{b_4}{a_4}, 1, f\right)} \tag{8.18}$$

因此点 $R(g)$ 映射到了 $r(f)$,但是 g 和 f 不相等。g 只有在 $f=0$ 和 $f=1$ 时与 f 相等,随后 g 随 f 因除以一个和比例 b_4/a_4 有关的分母而变化。如果 a_4 等于 b_4 则 g 和 f 相等(检验这个关系)。图 8.45 给出了对于不同的 b_4/a_4 的值,g 随 f 的变化关系。

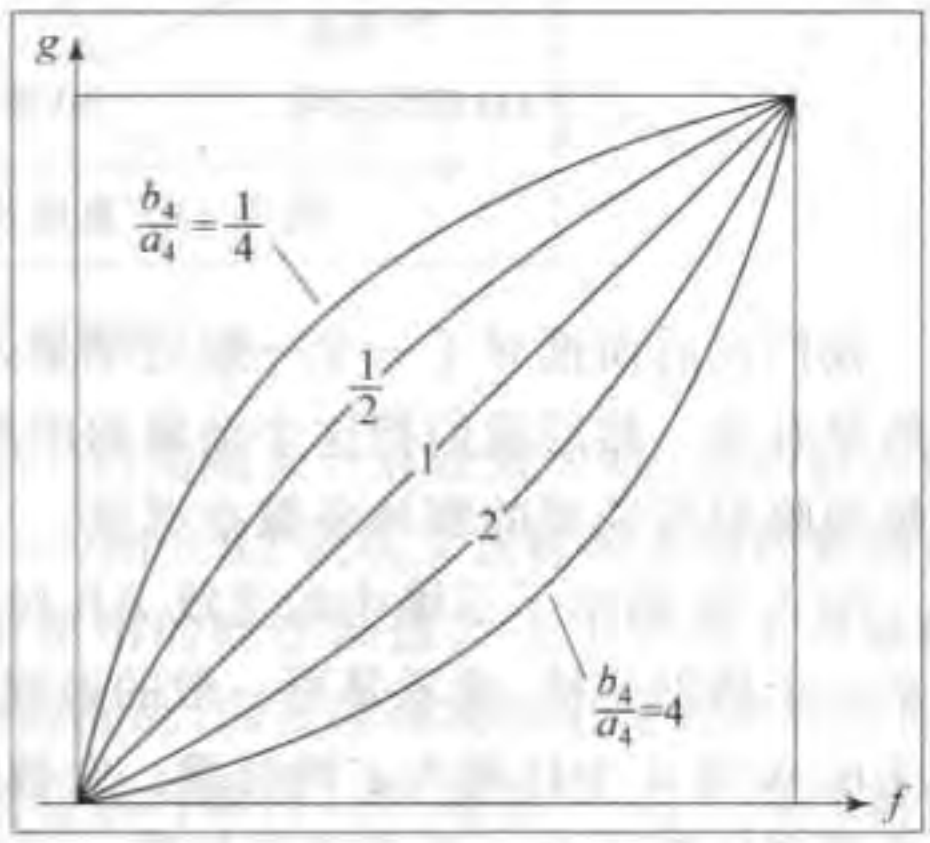


图 8.45 g 和 f 的关系

这样我们来到最后一步,看看 $r(f)$ 映射到三维面片上的点 $R(g)$ 在哪里。把等式(8.17)简写成 $R(g)=A(1-g)+Bg$,然后代数化简得到第一项(请自行推导):

$$R_1 = \frac{\mathrm{lerp}\left(\frac{A_1}{a_4}, \frac{B_1}{b_4}, f\right)}{\mathrm{lerp}\left(\frac{1}{a_4}, \frac{1}{b_4}, f\right)} \tag{8.19}$$

对于分量 R_2 和 R_3，有相似的等式(其中的分母和 R_1 的相同)。这是个关键的结果。它说明了三维中哪个点(R_1, R_2, R_3)和屏幕坐标系中位于两点 a、b 之间(比例为 f)的点对应。所以任何三维面片顶点上的随其线性变化的属性值(比如纹理)就可以依旧线性地变化。

我们关心的变换矩阵 M 的两种分类是：

- 变换是仿射变换。
- 变换是透视投影变换。

a. 当变换是仿射变换，a_4 和 b_4 都等于 1(为什么?)，所以上述的等式可以简化。f 和 g 一致并且上面提到的 R_1 等于 $\mathrm{lerp}(A_1, B_1, f)$。我们可以直接总结这点为：

事实：如果 M 是仿射变换，直线 ab 上等距的间隔对应到直线 AB 上依旧是等距的间隔。

b. 当 M 表示从视点坐标系到剪裁坐标系的透视投影变换时，第四个分量 a_4 和 b_4 不再是 1。我们在第 7 章推导过矩阵 M，它的基本形式在等式(7.10)中给出：

$$M = \begin{bmatrix} N & 0 & 0 & 0 \\ 0 & N & 0 & 0 \\ 0 & 0 & c & d \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

其中 c 和 d 是使伪深度值正确的常量。对于这个矩阵，向量 $M(A,1)^T$ 是什么？是 $\tilde{a} = (NA_1, NA_2, cA_3 + d, -A_3)$，关键的部分是 $a_4 = -A_3$。这也就是在相机坐标系下点在 z 轴方向上的位置——也就是点在视点前方的深度。

所以 a_4 和 b_4 的相对大小在线段近大远小的透视中处于核心位置：它们分别代表了 A 和 B 沿相机视平面法向的"深度"。如果 A 和 B 有相同的深度(比如它们所在的平面平行于相机视平面)，则沿线段没有透视扭曲，所以 g 和 f 实际上是一样的。图 8.46 中，从视点发出连接视平面上等距点的光线(以等距 f 增加)，到达原始三维面片上对应的就是不等距的点。对于这个例子，A 比 B 更近，导致 $a_4 < b_4$，所以 g 的增量会随着 A 到 B 而变大。

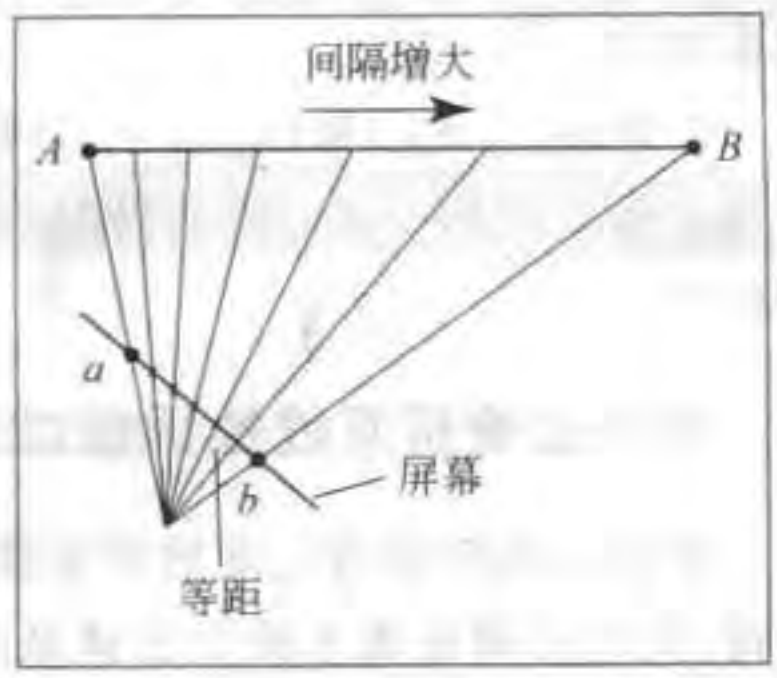


图 8.46　a_4 和 b_4 的值点的深度有关

增量渲染

我们现在把所有的这些放到一起，来为要被渲染的面片上的每个点计算正确的坐标(s, t)。图 8.47 给出了一个正在被渲染的谷仓模型的面片。面片左边端点是 a 和 b。面片在扫描线 y 上的范围是从 x_{left} 到 x_{right}。我们需要分别为 x_{left} 和 x_{right} 找到相应的纹理坐标(s_{left}, t_{left})和(s_{right}, t_{right})，这样它们就可以在扫描线上被插值。考虑要计算扫描线 y 上 s_{left} 的值 $s_{\mathrm{left}}(y)$。我们知道点 a 的纹理坐标 s_A 和点 b 的纹理坐标 s_B，因为这两个值是随顶点 A 和 B 经流水线传递过来的。如果扫描线 y 的位置在 y_{bott} 和 y_{top} 之间的比例是 f(也就是 $f = (y-$

$y_{bott})/(y_{top}-y_{bott})$),则从等式(8.19)中我们知道正确的纹理坐标是:

$$S_{left}(y)=\frac{\mathrm{lerp}\left(\frac{S_A}{a_4},\frac{S_B}{b_4},f\right)}{\mathrm{lerp}\left(\frac{1}{a_4},\frac{1}{b_4},f\right)} \tag{8.20}$$

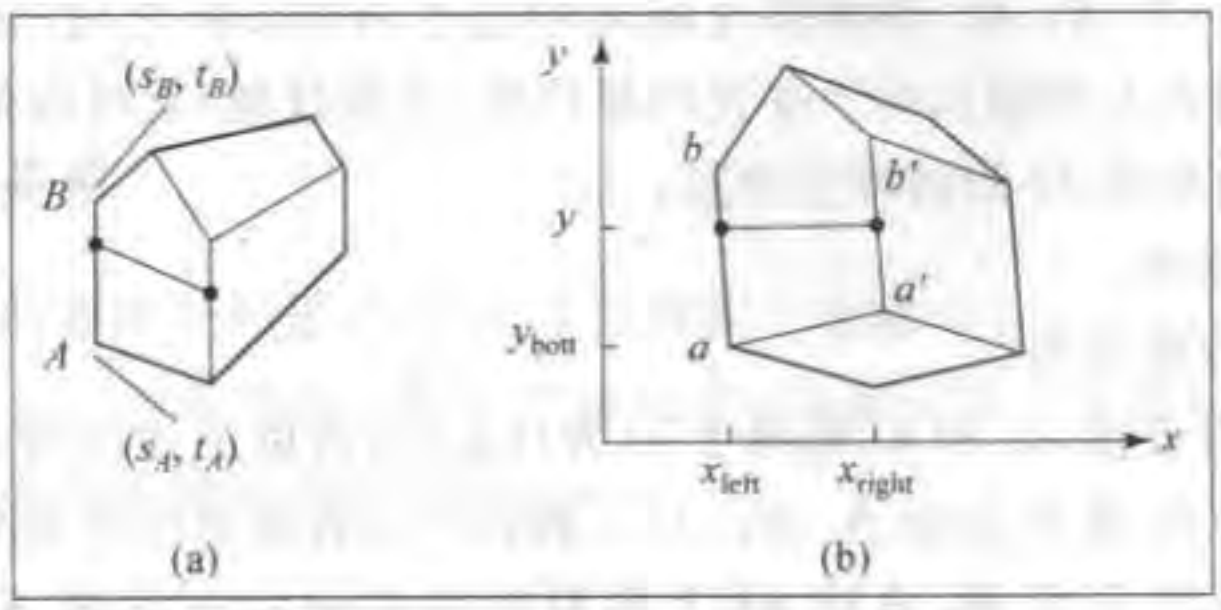


图 8.47 渲染面片上的纹理

相应地我们也能求出 t_{left} 的纹理坐标。注意 s_{left} 和 t_{left} 的式子中有相同的分母——$1/a_4$ 和 $1/b_4$ 的一个线性插值。分子是纹理坐标除以 a_4 和 b_4 后的线性插值。这有时也称作"理性线性绘制(rational linear rendering)"【Heckbert91】或者"双曲线插值(hyperbolic interpolation)"【Blinn92】。当 f 增加的时候为了高效的计算(s,t),我们需要存储以下的值 s_A/a_4, s_B/b_4, t_A/a_4, t_B/b_4, $1/a_4$ 和 $1/b_4$,因为随着像素的变化这些值不会改变。分子和分母项都可以找到基于 y 的增量计算方法,就像我们在 Gouraud 明暗处理(参见等式(8.15))中做的。但是为了求出每个 y 对应的 s_{left} 和 t_{left},等式中的除法是必须要做的。

(s_{right}, t_{right})也是用类似的方法计算出来的。它们的分母是基于点 a' 和 b' 所产生的 a'_4 和 b'_4 的值。

一旦(s_{left}, t_{left})和(s_{right}, t_{right})被找到,扫描线就可以被填充了。对于 x_{left} 到 x_{right} 中的每个点 x 所对应的 s 和 t 值,同样地再次使用双线性插值计算得到(求 x 的 s 值的表达式是什么?)。

为什么使用双线性插值以及这样计算的代价多大?

使用双线性插值正确的渲染纹理意味着什么?并且剪裁这步是否需要改进?我们将会看到,我们必须在流水线上传递某些附加的信息,并且计算一些不同于上面看到的值。

图 8.48 给出了流水线的一个改进。不同的点都标注了在那一步可用的信息。每个顶点 V 附带着纹理对(s,t)和一个顶点法向。顶点通过模式视点矩阵(法向乘以这个矩阵的逆的转置)变换,产生视点坐标系下的顶点 $A=(A_1, A_2, A_3)$和法向 $\mathbf{n}'$。计算着色时使用这个法向,产生颜色 $\mathbf{c}=(c_r,c_g,c_b)$。纹理坐标(s_A,t_A)(等于(s,t))一直跟随着顶点 A。顶点 A 随后经过透视变换,产生 $\bar{a}=(a_1, a_2, a_3, a_4)$。纹理坐标和颜色 $\mathbf{c}$ 并没有改变。

如第 7 章的讨论,处理完根据**视椎**(view volume)的剪裁。正像图中所提到的,这步有可能使一些顶点消失同时生成其他的顶点。比如当创建了一个顶点 D,我们必须决定它的位置(d_1,d_2,d_3,d_4),并且还有相应的颜色和纹理坐标。依据剪裁算法的原理,位置 d_i 通过线性插值得到:$d_i=lerp(a_i,b_i,t)$,$i=1,\cdots,4$,对于某个 t。请注意第四个分量 d_4 也是这样

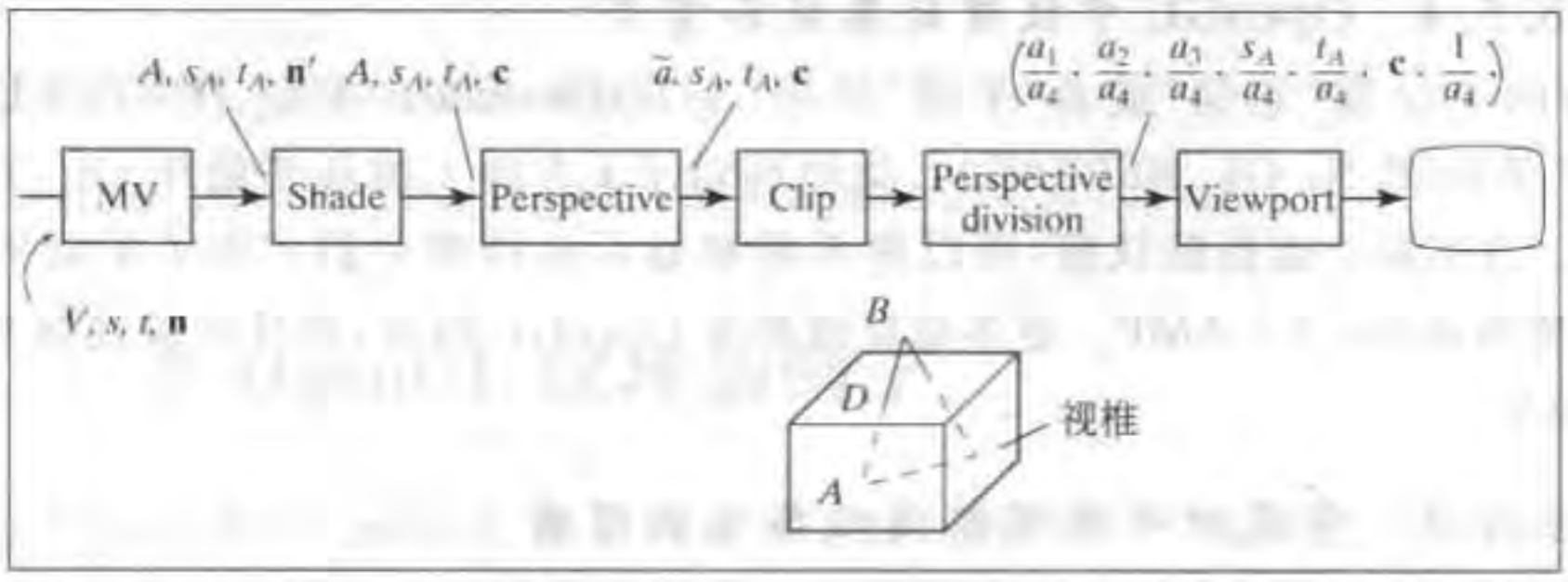


图 8.48　包含双线性插值的图形管线

形成的。很自然地，在这里同样使用线性插值来生成颜色分量和纹理坐标(这样做的原因在练习中有讨论)。因此，在剪裁之后，面片仍然由一些附带着颜色和纹理坐标的顶点组成。对于点 A，这些信息可以存为一个数组$(a_1, a_2, a_3, a_4, s_A, t_A, \mathbf{c}, 1)$。最后的一项 1 是被添加进去的，我们将在下一步用到它。

现在做完了透视除法之后，为了做双曲线插值我们需要一些项，比如 s_A/a_4，$1/a_4$ 和 t_A/a_4(见等式(8.20))，为了做双曲线插值我们把数组中的每一项都除以 a_4 得到了数组$(x, y, z, 1, s_A/a_4, t_A/a_4, \mathbf{c}, 1/a_4)$。(我们也会除以颜色分量以获得更加真实一些 Gouraud 明暗处理。参见练习)。前三项$(x, y, z) = (a_1/a_4, a_2/a_4, a_3/a_4)$表示了在标准化设备坐标系上点的位置。第三个分量是伪深度值。前两个分量将会通过视窗变换缩放和平移。为了简化符号，我们将继续称其屏幕坐标为(x, y, z)。

所以最后渲染器对于每个要渲染面片的顶点收到了数组$(x, y, z, 1, \cdots)$。现在使用等式(8.20)中的双线性插值渲染纹理就很简单了：每个顶点所需要的值 s_A/a_4 和 $1/a_4$ 都已知。

练习

练习 8.5.2　带纹理的网格模型的数据结构

讨论表示以下两种类型的网格模型所需要指定的数据类型：

a. 每个表面应用不同的纹理。

b. 一个纹理被卷绕包裹于整个网格。

写出每种数据类型所需要的模板，当网格表示一个立方体时给出每种类型表示下的一个数据样例。

练习 8.5.3　伪深度值的计算是正确的

证明在 8.4 节宣称的在计算伪深度值时使用线性插值(而不是双曲线插值)是正确的。假设点 A 投影到 a，点 B 投影到 b。用线性插值我们为投影后的点计算其伪深度值 $\mathrm{lerp}(a, b, f)$，把它作为这个点的第三个分量。只有当这样得到的结果等于插值点 $\mathrm{lerp}(A, B, g)$(对于某个适当的 g 值)投影后的伪深度值时，我们上面的做法才是对的。证明实际情况下这种做法是对的(提示：对被投影点的第三个分量使用等式(8.16)和等式(8.17))。

练习 8.5.4 OpenGL 中纹理的卷绕和剪裁

在 s 方向上设置"卷绕"或者"平铺"使用：glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)。类似地对于 t 方向上的环绕使用 GL_TEXTURE_WRAP_T。这实际上也是默认值，所以你不需要显示地指明它们。为了关掉平铺把 GL_REPEAT 替换成 GL_CLAMP。更多信息请参考 OpenGL 网站，并且实验观察 OpenGL 不同设置的效果。

练习 8.5.5 剪裁时对纹理做线性插值的理由

用视锥剪裁面片时经常会产生新的顶点。我们必须为每个点设定纹理坐标。假设新点 V 形成在顶点 A 到顶点 B 中间比例为 f 的位置。另外假设 A 点纹理坐标是 (s_A, t_A)，B 点类似。证明假设纹理被贴在一个平的面片上，设定 V 的纹理坐标为 $(\mathrm{lerp}(s_A, s_B, f), \mathrm{lerp}(t_A, t_B, f))$ 是合理的。

练习 8.5.6 双曲线插值的计算量

比较纹理坐标的双曲线插值和线性插值所需要的计算量。假设乘除法所需要的计算量 10 倍于加减法。

8.5.3 什么是纹理调和

纹理映射得到的数值在渲染计算中如何使用？为了产生不同的视觉效果，我们将看到三种常见的使用方法。我们简单地使用等式(8.5)中计算亮度灰度值的情况来学习。对于彩色情况只是同样的计算，分别应用于红色、绿色和蓝色分量。

1. 创建一个发光体

这是最简单的计算方法。设置可见的亮度 I 等于该点的纹理值：

$$I = \mathrm{texture}(s,t)$$

(或者再乘以某个常量)。物体看上去像发光或者荧光：小的纹理值表示发出很少的光，大的纹理值表示更多的光。不需要其他的光照计算。

(对于彩色光，红、绿、蓝分量分别地赋值：例如，红色分量 $I_r = \mathrm{textrue}_r(s,t)$)

为了让 OpenGL 这样贴纹理，设定：

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

2. 调和反射系数画纹理

我们先前知道一个物体的颜色就是其漫反射光的颜色(当处于白光中)。因此我们可以通过变化漫反射系数，也许还有环境光反射系数，来使得纹理看上去就像被画在物体表面上一样。我们称为纹理函数逐点地调和反射系数。这样对于适当的 s 和 t 值，我们替换等式(8.5)为：

$$I = \mathrm{texture}(s,t)[I_a\rho_a + I_d\rho_d \times lambert] + I_{sp}\rho_s \times phong^f$$

因为 Phong 镜面反射光的颜色是光源的颜色而不是物体的颜色，所以高光不依赖于

纹理。

为了让 OpenGL 这样贴纹理，设定：

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

8.5.4　一个 OpenGL 纹理的例子

为了说明如何调用 OpenGL 提供的纹理工具，我们用一个应用程序来做说明，这个程序显示一个旋转的立方体，立方体的每个面上都贴有不同的图片。图 8.49 是这个程序所实现的动画的截图。纹理映射到每个面片使用双曲线插值。

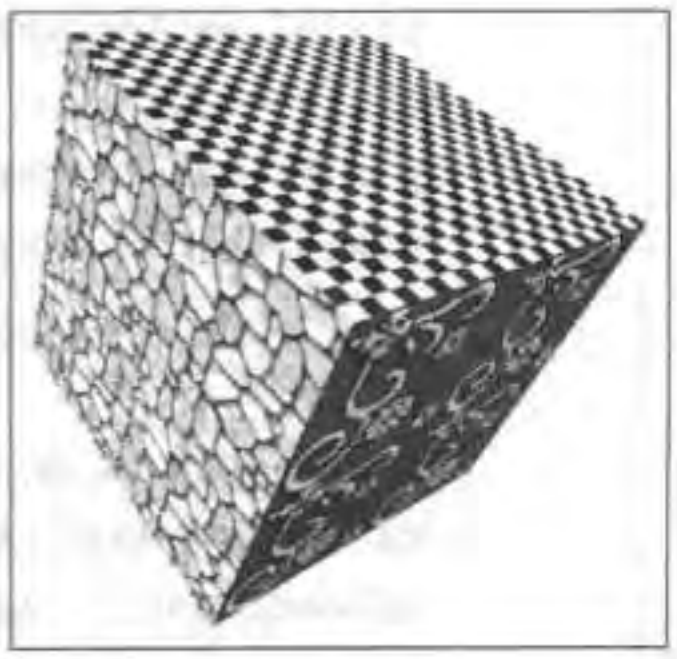


图 8.49　演示程序生成的贴有纹理的立方体

图 8.50 给出了这个程序的代码。它使用了一些 OpenGL 函数建立纹理并且把它们贴到立方体的面上。这里还列出了许多不同的参数可供映射纹理的时候使用。图中给出的参数可以很好地工作，小心调整一些参数（参见 OpenGL 的手册）可以改善图像质量或者提高速度。我们这里只介绍基本的关键步骤。

添加纹理的第一个任务是在内存中创建纹理的**像素图像**（pixmap）。OpenGL 实用的纹理存储在**像素图像**（pixel maps，或者简称为 pixmaps）中。这点将在第 9 章有深入的讨论，并且类 RGBpixmap 的设计就是为了提供建立和操作纹理图像的工具。这里我们把一个纹理图像看成一个简单的像素值的数组，每个像素值都是一个存储红、绿、蓝颜色值的三元组。

```
#include "windows.h"
#include "GL/gl.h"
#include "GL/glu.h"
#include "GL/glut.h"
#include "RGBpixmap.h"

//##########################GLOBALS##########################
RGBpixmap pix[6];     //make six (empty) pixmaps
float xSpeed=0, ySpeed=1,  xAngle=0.02, yAngle=0.02;
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<myinit>>>>>>>>>>>>>>>>>>>>>>>>>>>>.
void myInit(void)
{
    glClearColor(1.0f,1.0f,1.0f,1.0f);          //background is white
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);

    pix[0].makeCheckerboard();                  //make pixmap procedurally
    pix[0].setTexture(2001);                    //create texture
```

图 8.50　旋转的纹理立方体程序

```
    pix[1].readBMPFile("Mandrill.bmp");          //make pixmap from image
    pix[1].setTexture(2002);                     //create texture

    pix[2].readBMPFile("Mandrill.bmp");          //make pixmap from image
    pix[2].setTexture(2003);                     //create texture

    pix[3].makeCheckerboard();                   //make pixmap from image
    pix[3].setTexture(2004);                     //create texture

    pix[4].readBMPFile("red.bmp");               //make pixmap from image
    pix[4].setTexture(2005);                     //create texture

    pix[5].makeCheckerboard();                   //make pixmap from image
    pix[5].setTexture(2006);                     //create texture


    glViewport(0, 0, 640, 480);                  //set up the viewing system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, 640.0/ 480, 1.0, 30.0);//set camera shape
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslated(0.0, -0.2, -4);                 //move camera back
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<display>>>>>>>>>>>>>>>>>>>>>>>>>>
void display(void)
{
      glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
    glPushMatrix();
    glRotated(xAngle, 1.0,0.0,0.0);
    glRotated(yAngle, 0.0,1.0,0.0); //rotate

    glBindTexture(GL_TEXTURE_2D,2001);   //top face: 'fake' checkerboard
    glBegin(GL_QUADS);
    glTexCoord2f(-1.0, -1.0); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(-1.0, 2.0); glVertex3f(-1.0f, 1.0f,  1.0f);
    glTexCoord2f(2.0, 2.0); glVertex3f( 1.0f, 1.0f,  1.0f);
    glTexCoord2f(2.0, -1.0); glVertex3f( 1.0f, 1.0f, -1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2002);   //right face: mandrill
    glBegin(GL_QUADS);
```

图 8.50 (续)

```
    glTexCoord2f(0.0, 0.0); glVertex3f(1.0f, -1.0f,  1.0f);
    glTexCoord2f(0.0, 2.0); glVertex3f(1.0f, -1.0f, -1.0f);
    glTexCoord2f(2.0, 2.0); glVertex3f(1.0f,  1.0f, -1.0f);
    glTexCoord2f(2.0, 0.0); glVertex3f(1.0f,  1.0f,  1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2003);      //left face: mandrill
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0); glVertex3f(-1.0f, 1.0f,  -1.0f);
    glTexCoord2f(2.0, 0.0); glVertex3f(-1.0f, 1.0f, 1.0f);
    glTexCoord2f(2.0, 2.0); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(0.0, 2.0); glVertex3f(-1.0f,  -1.0f,  -1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2004);      //bottom face: 'fake' checkerboard
    glBegin(GL_QUADS);
    glTexCoord2f(1.0, 1.0); glVertex3f(1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0, -2.0); glVertex3f(1.0f, -1.0f,  -1.0f);
    glTexCoord2f(-2.0, -2.0); glVertex3f( -1.0f, -1.0f,  -1.0f);
    glTexCoord2f(-2.0, 1.0); glVertex3f( -1.0f, -1.0f, 1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2005);      //box front: red
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 2.0); glVertex3f(1.0f, 1.0f, 1.0f);
    glTexCoord2f(2.0, 0.0); glVertex3f(1.0f, -1.0f, 1.0f);
    glTexCoord2f(0.0, -2.0); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(-2.0, 0.0); glVertex3f(-1.0f, 1.0f, 1.0f);
    glEnd();

    glBindTexture(GL_TEXTURE_2D,2006);   //box back: 'fake' checkerboard
    glBegin(GL_QUADS);
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0f, 1.0f, -1.0f);
    glTexCoord2f(1.0, 0.0); glVertex3f(1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0, 0.0); glVertex3f(-1.0f, -1.0f, -1.0f);
    glEnd();

    glFlush();
    glPopMatrix();
    glutSwapBuffers();
    glutPostRedisplay();
}
```

图 8.50（续）

```
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<spinner>>>
```

然后为每个对象读入所需的纹理图像。最后把每个对象传递给 OpenGL 来定义纹理。

生成函数纹理

我们首先使用方法 makeCheckerBoard()创建一个棋盘纹理。棋盘的样子为我们所熟悉而且非常容易创建，同时方方正正的几何形状也使其成为测试正确性的好纹理。程序为 pix[0]生成棋盘像素图像使用：pix[0].makeCheckerboard()。方法本身的实现如下：

```
void RGBpixmap:: makeCheckerboard()
{   //make checkerboard pattern
    nRows=nCols=64;
    numPixel=new RGB[3*nRows*nCols];
    if (!numPixel) {cout <<"out of memory!"; return;}
    long count=0;
    for(int i=0; i <nRows; i++)
        for(int j=0; j <nCols; j++)
        {
            int c=(((i/8)+(j/8))%2)*255;
            numPixel[count].r=c;                    //red
            numPixel[count].g=c;                    //green
            numPixel[count++].b=0;}                 //blue
        }
}
```

它创建了 64×64 的数组，每个像素都是 RGB 的三元组。OpenGL 一直到版本 1.5 都要求纹理像素图像的宽和高都必须是 2 的整数次幂。OpenGL 2.0 已经去除了这个限制。像素图像在内存中的布局就是一个长的字节数组：从下到上一行接一行，每行中从左到右。每个像素都被赋值为(c,c,0)，其中每 8 个像素 c 值在 0 和 255 中跳变一次(我们在练习 2.3.1 中使用了类似的跳变方法)。棋盘的两个颜色是黑(0,0,0)和黄(255,255,0)。函数保存了像素图像中第一个像素的地址，它在后面将会传递给 glTexImage2D()来为 OpenGL 创建真正的纹理。

一旦生成了像素图像，我们必须把它和一个单独的正整数值绑定到一起(作为它的名字)，这样在 OpenGL 中就能够准确地引用它。我们随意地为示例中的 6 个纹理命名为 2001,2002,…,2006。通过执行一些 OpenGL 的调用来创建纹理，我们把这些调用封装到方法中：

```
void RGBpixmap::setTexture(GLuint textureName)
{
    glBindTexture(GL_TEXTURE_2D, textureName);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,GL_NEAREST);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, nCols, nRows, 0, GL_RGB,
                 GL_UNSIGNED_BYTE, pixel);
}
```

通过调用 glBindTexture()为要生成的纹理绑定名字。正像我们将要看到的，当这个调

用执行之后,它就会设定这个纹理为活动的纹理。

调用 glTexParameteri()指定了,当纹理需要被放大或者缩小时,使用坐标距离像素中心最近的那个纹理元素来填充像素。这样做速度快但会导致走样。我们将在第 9 章更深入地讨论图像滤波和反走样。最后,调用 glTexImage2D()将纹理图像和当前的纹理联系起来。这个调用描述了纹理是由 RGB 字节三元组组成的二维图像,还给出了宽和高以及位图首字节(像素)的内存地址。

从已有图像生成纹理

OpenGL 不支持读取图像文件并且在内存中创建像素图像。方法 readBMPFiel()给出了一个简单的读取 BMP 图像到像素图像的方法,具体实现可以在本书配套网站上找到。例如:

```
pix[1].readBMPFile("mandrill.bmp);
```

读取文件 mandrill. bmp 并且创建像素图像 pix[1]。

一旦像素图像被建立,就可以用 pix[1]. setTexture()函数把像素图像传递给 OpenGL 以生成纹理。

纹理映射必须通过 glEnable(GL_TEXTURE_2D)启用。另外,还需要使用 glHint(GL_PERSPECTIVE, GL_NICEST)要求 OpenGL 正确的(使双曲线插值)渲染纹理,这样当面片相对观察者旋转的时候,纹理才会正确的贴在面片之上。

纹理的创建、启用和设定插值方法只需在初始化过程中做一次。随后每次显示的时候,纹理都会被正确的处理。在 display()中,立方体通过 xAngle 和 yAngle 被旋转,并且画出其六个面。这只需要简单的为每个面绑定正确的纹理,并且在 glBegin()/glEnd()中对应好面片顶点的三维坐标和其纹理坐标,就像我们在代码中展示的一样。

当立方体的渲染(在屏幕外)完成了,glutSwapBuffers()被调用使得新的帧变为可见。控制动画是使用回调函数(callback function)spinner()作为"空闲函数"。当系统空闲的时候——没有需要相应的用户输入,spinner 就会被自动的调用。它小小地改变立方体的旋转角度然后再次调用 display()。最终实现的动画效果就是立方体在旋转,这样各个面片就会反复的出现在视野中或者转出视野。

8.5.5 在曲面上卷绕纹理

我们已经看到了如何把纹理贴到平面上。现在我们将看看如何在曲面上卷绕纹理,比如一个啤酒罐或者一个象棋子。我们假定物体是网格模型,它是由大量小的平的面片组成。正如 8.5.1 节末尾提到的,网格的每个顶点有其相应的纹理坐标(s_i, t_i),虽然我们可能不知道它们。所以主要的问题是为网格的每个顶点找到适当的纹理坐标(s, t)。

我们给出映射纹理到圆柱状物体上和球状物体上的例子,看看对于每种情况都是如何处理的。

例 8.5.1 在罐头上包裹标签

假设我们想把一个标签裹在一个圆柱体上,如图 8.51 所示。我们自然地会想到圆柱坐

标系。标签位于方位角 θ_a 到 θ_b、z 轴 z_a 到 z_b 之间。圆柱体是用多边形网格表示，所以它的表面是矩形的条带，见图 8.51(b)。对于每个面片的顶点 V，我们必须找到合适的纹理坐标 (s_i, t_i)，保证纹理上正确的"条带"被映射到表面。

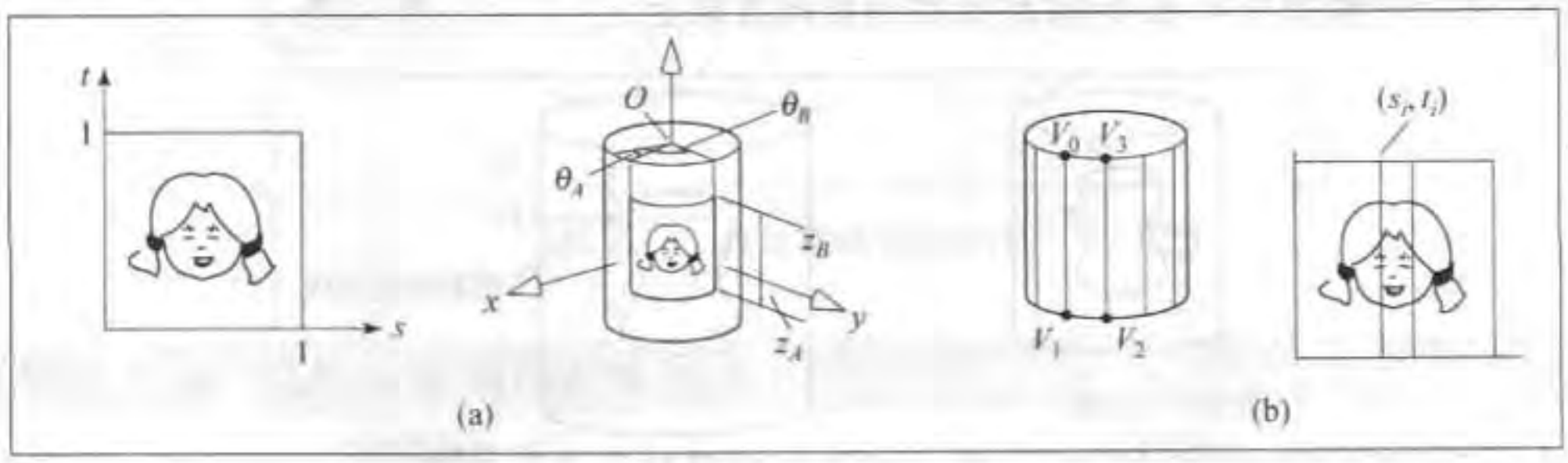


图 8.51　将标签包裹在圆柱上

该几何体足够简单以至于解决方法可以直观地得到。这里 (s,t) 和圆柱体上一点的方位角、高度 (θ,z) 之间存在一个直接的线性关系：

$$s = \frac{\theta - \theta_a}{\theta_b - \theta_a}, \quad t = \frac{z - z_a}{z_b - z_a} \tag{8.21}$$

因此如果有 N 个面片围绕成圆柱体，第 i 个面片左边的方位角是 $\theta_i = 2\pi i/N$，则它左上角顶点的纹理坐标 $(s_i, t_i) = ((2\pi i/N - \theta_a)/(\theta_b - \theta_a), 1)$。其他三个顶点的纹理坐标用类似的方法可以得到。将每个面片的顶点和 (s,t) 值联系起来只需要简单地在建模过程中添加一些代码即可(见练习)。

当物体不是一个简单的圆柱体时，这个过程会变得更加复杂。我们后面将看到如何把纹理映射到更一般的旋转曲面上。

例 8.5.2　把标签收缩包裹到一个旋转曲面上

回顾第 6 章，旋转曲面是通过一个特征曲线 $(x(v), z(v))$ 定义的，如图 8.52(a)，曲面的结果——一个花瓶，是通过参数 $P(u,v) = (x(v)\cos u, x(v)\sin u, z(v))$ 给出。物体的曲面是通过一系列沿着轮廓线分布的面片表示的(见图 8.52(b))。所以给定的面片 F_i 有四个顶点 $P(u_i, v_i)$，$P(u_{i+1}, v_i)$，$P(u_i, v_{i+1})$，$P(u_{i+1}, v_{i+1})$。我们需要为这些顶点找到对应的纹理坐标 (s,t)。

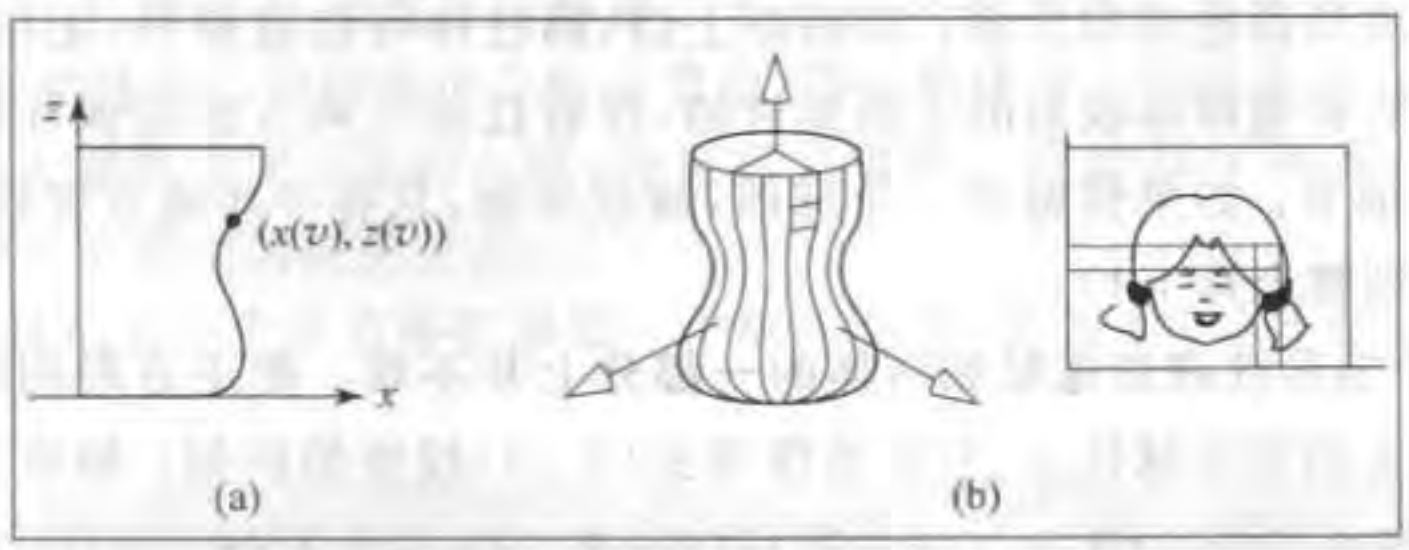


图 8.52　将标签包裹在花瓶上

和上个示例一样，一种自然的做法是根据等式(8.23)使 s 和 t 线性的随 u 和 v 变化。这等价于把纹理卷绕贴在一个假想的包裹着花瓶的橡胶圆柱体上(见图 8.53(a))，然后让

圆柱体收缩,这样纹理就被水平和竖直地分割直到它们接触到花瓶的表面。这种方法称作"收缩包裹",由 Bier 和 Sloane【Bier86】提出,文章中他们还讨论了把纹理映射到不同类型形状上的几种方法。可以把收缩映射看成沿着假想圆柱体的法向(见图 8.53(b)):纹理上的顶点 P_i 对应的模型点 V_i 位于从 P_i 出发的法向方向上。

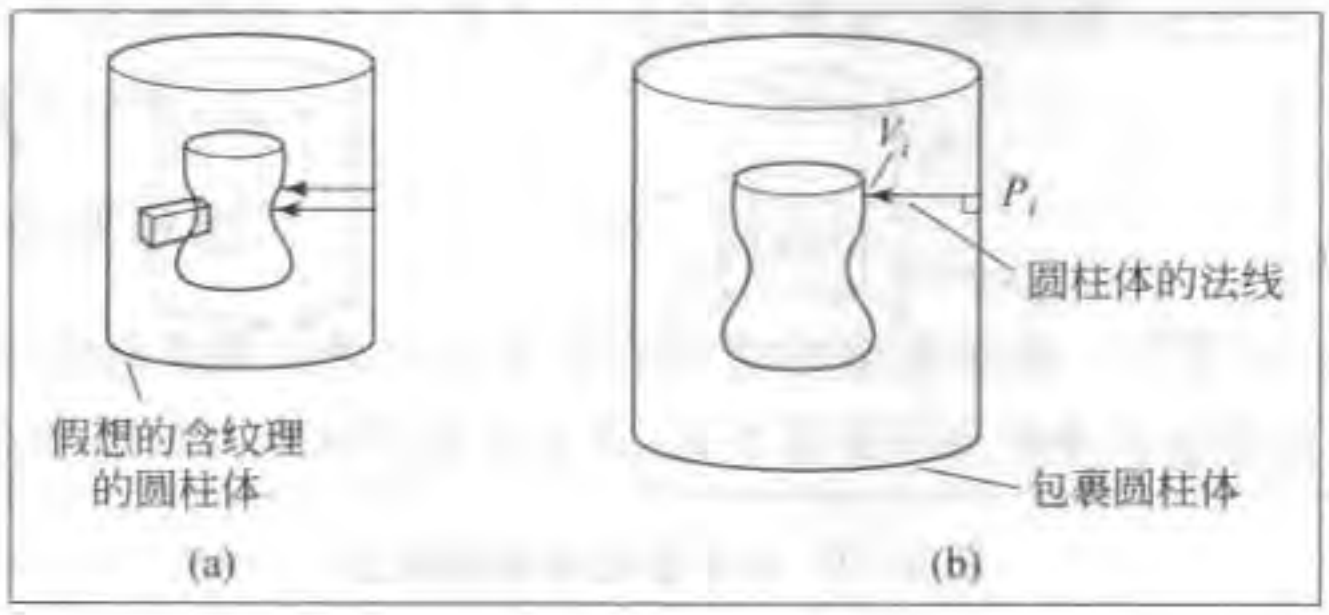


图 8.53 将纹理收缩包裹到花瓶上

收缩包裹对圆柱形状的物体有效,虽然如果特征曲线是复杂的形状,纹理的图像有可能被扭曲。

Bier 和 Sloane 提出了联系假想圆柱上的纹理点和物体顶点的几种其他方法。图 8.54 给出了其中的两种。

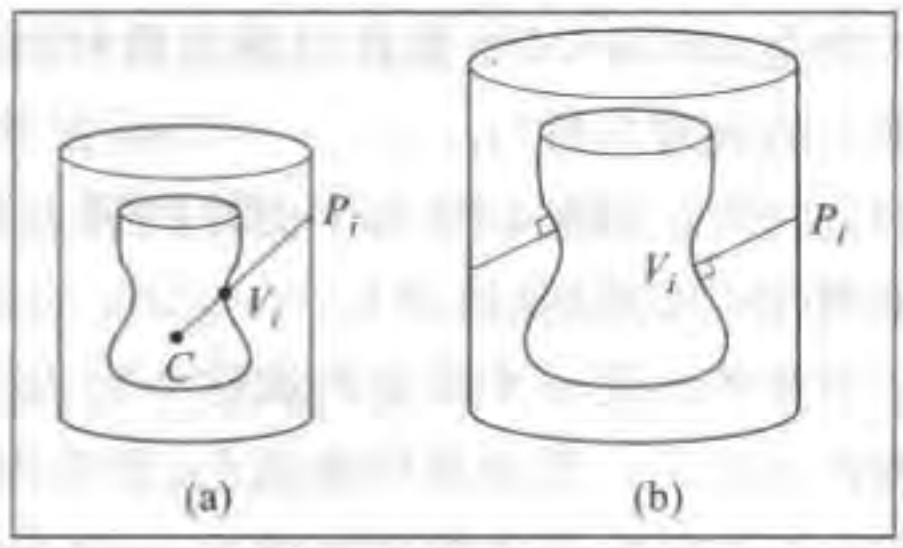


图 8.54 从假想圆柱上到物体的不同映射方法

图 8.54(a)中,一条直线从物体的中心点 C,通过顶点 V_i 和圆柱体相交于 P_i 点。图 8.54(b)中,使用了 V_i 点处的物体表面法向:P_i 是从 V_i 出发沿这个法向与圆柱的交点。要注意的是,这三种联系纹理和物体顶点的方法可能会产生非常不同的结果,因为效果依赖于物体的形状(见练习)。设计者需要根据物体的形状和纹理的属性选取最合适的方法(对于象棋棋子的兵使用哪种方法合适?)。

例 8.5.3 映射纹理到球体上

映射矩形纹理到圆柱体很简单:从拓扑上讲,圆柱体可以被剪开,无扭曲地展成平面。球体却不同。像所有做球体映射的人所知道的,没有任何一种方法能保证够把球体展成平面后保持所有的细节:如果你剪开一个球体,铺成平面,总有一些地方要被严重的拉伸(想象一个棋盘映射到整个球上!)。

但是,把一个矩形纹理图像贴到球体的**一部分**上并不难。把正方形的纹理映射到经度 θ_a 到 θ_b 维度 ϕ_a 到 ϕ_b 的部分球体上,只需要像等式(8.23)线性的映射:如果顶点 Vi 位于(θ_i, ϕ_i),其对应的纹理坐标$(s_i, t_i) = ((\theta_i-\theta_a)/(\theta_b-\theta_a), (\phi_i-\phi_a)/(\phi_b-\phi_a))$。图 8.55 给出了一个图片贴到部分球体上的结果,只能看到少量的扭曲。

图 8.55(b)说明了如何把纹理覆盖到整个球体上:把八个三角形纹理映射到球体的八个部分。

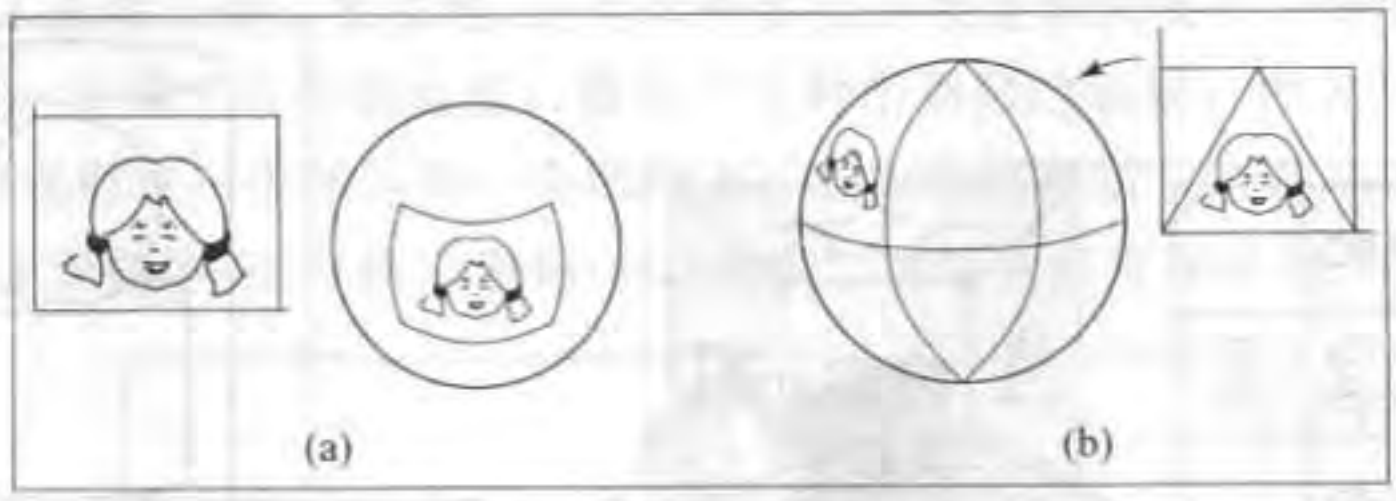


图 8.55　将纹理映射到球体

例 8.5.4　映射纹理到球状物体上

我们在前面的几段中介绍了为圆柱状物体添加纹理。但是一些物体更像球体而不是圆柱。图 8.56(a)给出了一个多面球体，它的表面是五边形和六边形。我们可以设计一些五边形和六边形的纹理，并且手工把它们贴在每个面上，但是对于一些场景我们更希望用一个纹理包裹住整个多面球体。

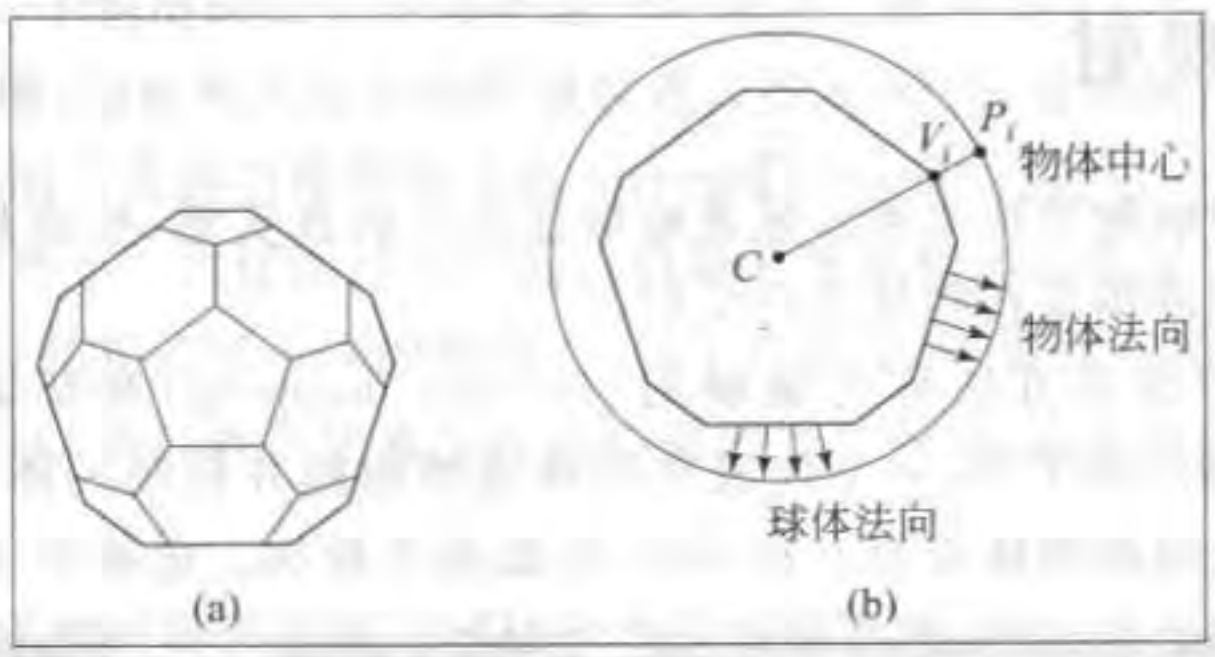


图 8.56　球状物体

自然地，我们可以在球状物体外包裹一个拥有纹理的假想球体(而不是一个假想圆柱)，使用上面讨论的某个建立关联的方法进行贴图。图 8.56(b)给出了被这样一个球体包裹的多面球体。这里列出了三种联系纹理点 P_i 和物体顶点 V_i 的方法：

- 物体中心：P_i 在物体中心 C 到顶点 V_i 的连线上。
- 物体法向：P_i 是从 V_i 出发沿面片法向与球体的交点。
- 球体法向：V_i 是从 P_i 出发言沿球面上 P_i 点法向与物体的交点。

(**问题**：如果物体中心和球体中心恰巧重合，那么物体中心和球体法向两种方法是否一样?)物体中心法大多数情况下更好一些，并且容易实现。正如 Bier 和 Sloane 所说的，其他两种方法经常产生不希望的渲染结果。

图 8.57(a)显示了一个立方体扩展到一个纹理图上，图 8.57(b)显示了该纹理包裹该立方体的情况。

练习

练习 8.5.7　如何联系 P_i 和 V_i

图 8.58 中的旋转曲面 S 由一个圆柱体和其上的一个球体组成。整个物体被一个拥有棋盘纹理的圆柱体包围。画出使用以下三种对应方法得到的最终贴图结果：

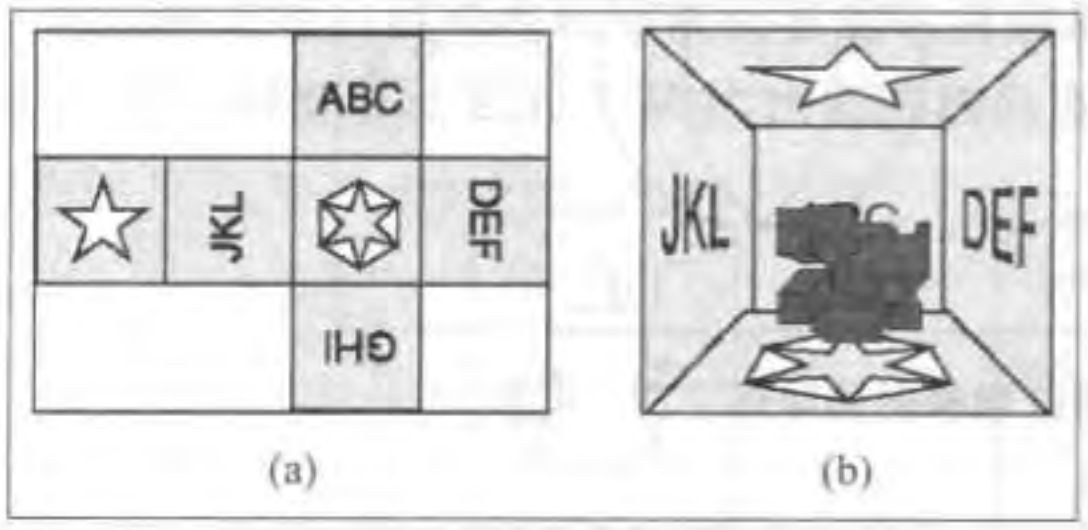


图 8.57 使用包围立方体

图 8.58 假想圆柱体包裹的旋转曲面

- 收缩包裹
- 物体中心
- 物体法向

8.5.6 反射映射

一类被称为**反射映射**的技术可以显著地增强图片的真实感,特别是在动画中。基本的想法是看到物体周围的场景在物体上的反射。

两种主要的反射映射方法称为**铬映射**(chrome mapping)和**环境映射**(environment mapping)。在**铬映射**的例子中,一个代表周围环境的粗糙并模糊的图像被反射在物体上,就像你可能在电镀上铬的物体表面上看到的,比如拖车栓钩。电视中大量出现的闪亮的字母和标志在空间中移动的动画,都是使用了铬映射加上偶尔出现的高光后产生的奇特效果。图 8.59 给出了这样一个例子。图 8.59(a)是一个铬映射的例子;整个图像清晰呈现了一个闪亮的机器人坐在一个凳子上。机器人上有暗和亮的区域,但是我们并不能看出来反射本身的图像,反射只是提供了围绕在物体周围的大概的情况。

图 8.59 举例

(a) 铬映射;(b) 环境映射

在**环境映射**(首先由 Blinn 和 Newell【Blinn76】提出)的例子中,我们在物体的反射中可以看到一个可辨识的周围环境的图像。从这样的反射中,我们得到了更多的视觉线索,特别是当物体移动的时候。每个人想必都看过这个经典的照片,一个宇航员在月球上行走,在他

的面罩上反射出了月球飞船。相似地，在电影中你有时候会看到演员带着墨镜的特写（比如黑暗武士，译者注：星球大战中的角色），墨镜中反射出周围的场景。图 8.60 所给的两个例子中，一个餐馆被反射在一个球体和一个圆环上。餐馆的纹理被包裹在一个巨大的球体上，这个球体内部包含着物体，这样纹理坐标(s,t)对应于球体的经度和纬度。

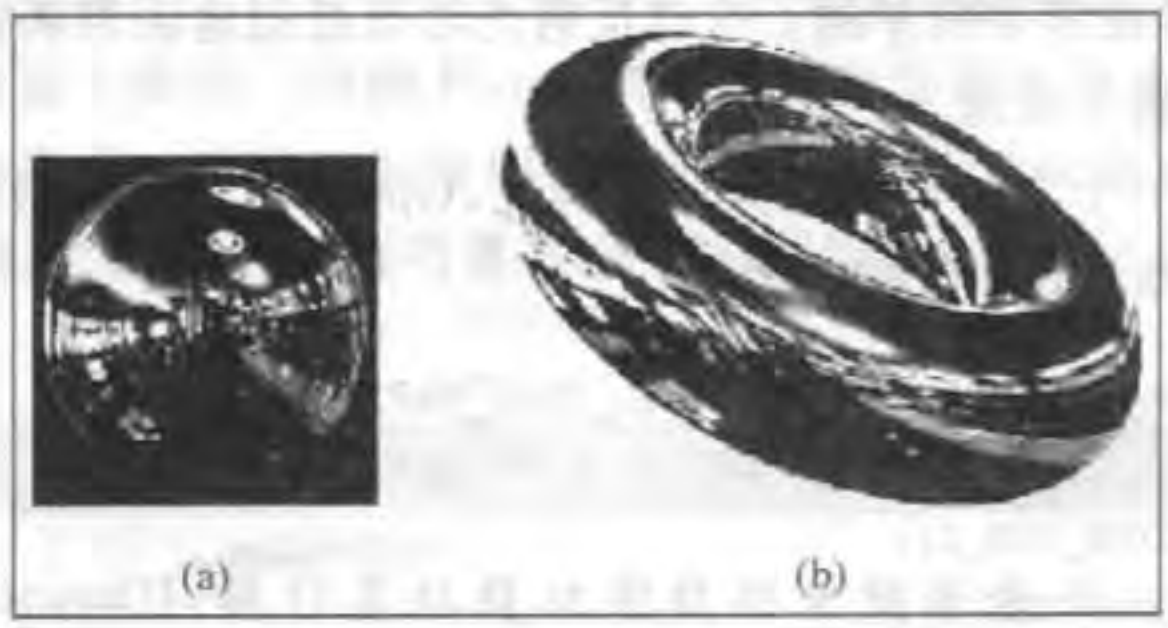


图 8.60　环境映射举例

图 8.61 使用了一个包围的立方体而不是一个球体。图 8.61(a)给出了这个环境，它由 6 张从室内观察到的墙、地板和天花板的图像组成。图 8.61(b)给出的一个光泽物体反射了房间的不同部分。包围立方体的使用首先由 Greene【Greene86】提出，一般来讲它的效果的扭曲将会比包围球的更小。六张图片可以通过渲染 6 张从物体的位置分别向不同方向观察的图片生成（当然这时候先去掉中间的物体）。对于每张图片，要摆好相机的位置，设好窗口的大小。另外，纹理可以通过数字化真实相机拍摄的照片获得，真实相机从各个主要方向拍照房间或场景。

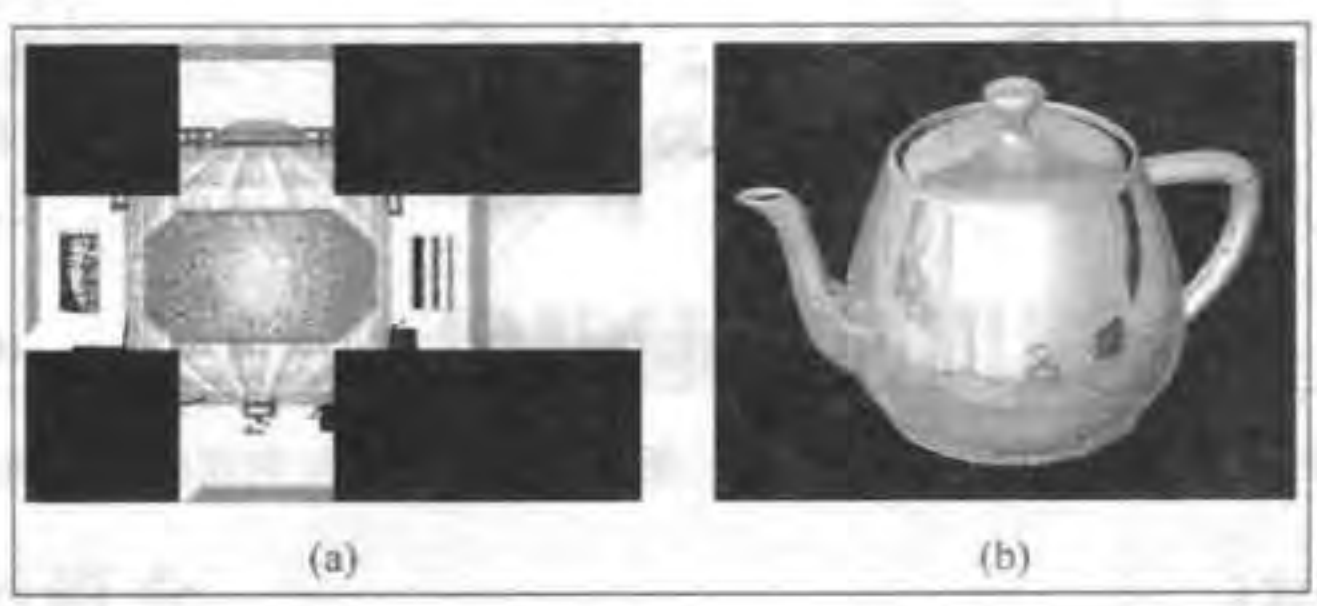


图 8.61　基于包围立方体的环境映射

在带光泽物体移动的动画中，铬映射、环境映射与法向纹理映射有很大的区别。前者反射的图像会在移动的物体上流动，而法向纹理贴图将会被固定在物体上和其一起移动。如果一个带光泽的球体旋转一个固定角度，法向纹理贴图也会随着球体旋转一个角度，而反射映射确保持不变。

如何实现铬映射和环境映射

环境映射是如何工作的？你看到的光泽物体上的点 P 是环境中正好在 P 点正确反射到你眼睛的颜色。为了找到这个方向，跟踪从视点到 P 的光线，并且计算反射光的方向，再

沿着这个光线找到与它(在包围立方体或者球体上)相交的纹理。图 8.62 画出了一个光线从视点出发射向点 P。如果这个光线的方向是 $\mathbf{u}$ 并且 P 点的单位法向是 $\mathbf{m}$,我们根据等式(8.2)可以知道反射光的方向是 $\mathbf{r}=\mathbf{u}-2(\mathbf{u}\cdot\mathbf{m})\mathbf{m}$(回顾第 4 章)。反射光沿 $\mathbf{r}$ 方向前进直到与有纹理的假想曲面相交。如果假设光泽物体位于正中心,并且和包围立方体或球体相比非常小,则这个计算是非常简单的。这样反射光可以近似地认为从物体中心射出,它的方向 r 可以被直接用来索引纹理。

对于纹理包裹在一个大的包围立方体的情况,OpenGL 提供了进行这种近似环境映射的工具。通过设定 s 和 t 的映射模式实现:

```
glTexGenf(GL_S,GL_TEXTURE_GEN_MODE, GL_CUBE_MAP);
glTexGenf(GL_T,GL_TEXTURE_GEN_MODE, GL_CUBE_MAP);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
```

这样当顶点 P 和其单位法向 $\mathbf{m}$ 传入流水线时,OpenGL 计算一个适合的纹理坐标(s,t)用来索引在包围立方体上的纹理。这只在物体面片的每个顶点上计算一次,绘制面片时依旧为中间的其他点插值计算纹理坐标(s,t)。

OpenGL 是如何快速计算出正确的纹理坐标(s,t)的呢?如图 8.63(a)所示,它首先(使用上面的等式)找到(在视点坐标系中的)反射方向 $\mathbf{r}$,其中 $\mathbf{u}$ 是(视点坐标系中的)从视点指向物体上顶点 V 的单位向量,$\mathbf{m}$ 是 V 点的法向。

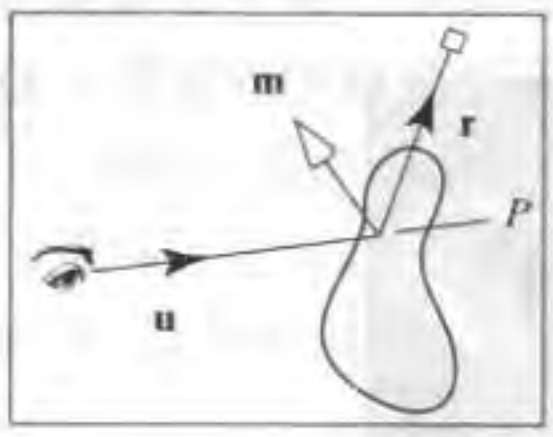


图 8.62 计算反射光的方向

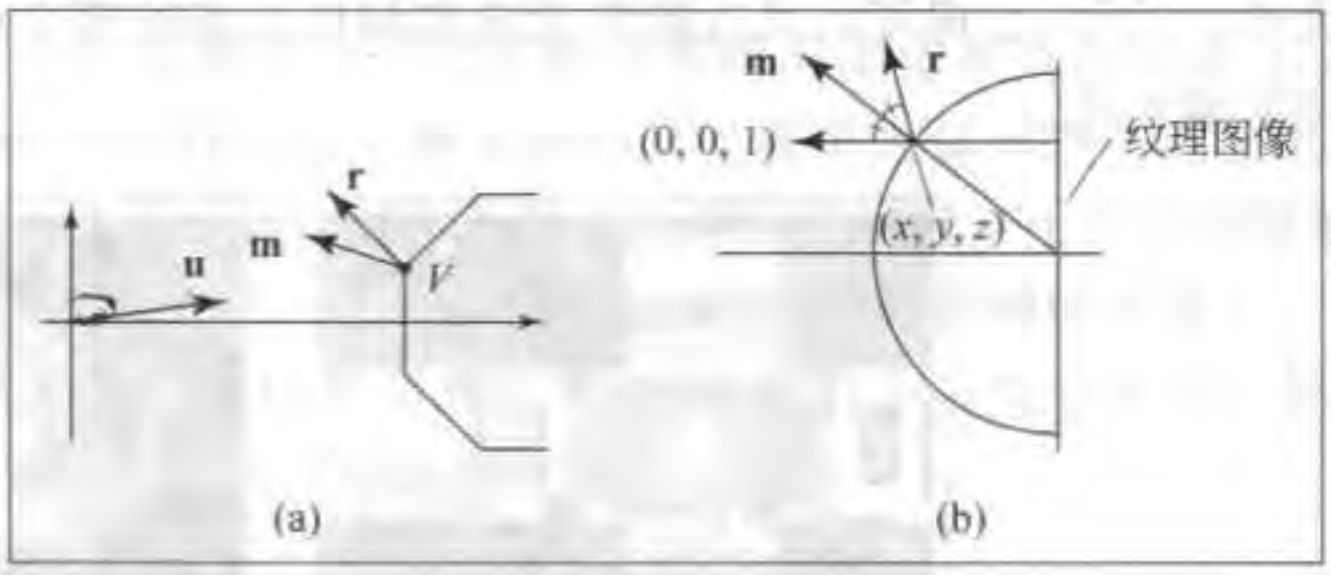


图 8.63 OpenGL 中纹理坐标的计算

然后它使用等式:

$$(s,t)=\left(\frac{1}{2}\left(\frac{r_x}{p}+1\right),\quad \frac{1}{2}\left(\frac{r_y}{p}+1\right)\right) \tag{8.22}$$

其中 p 是一个神奇的缩放因子 $p=\sqrt{r_x^2+r_y^2+(r_z+1)^2}$。这项的推导放在练习中。我们还必须预计算一个纹理表示所在的环境。这个纹理相当于把一个完全反射的球体(译者注:原著中是立方体)置于环境中,视点从无限远处观察球体上的图像【Haeberli93】。这样就把视点背后的半球环境映射到了纹理中央的圆,视点前方的半球环境映射到围绕这个圆的圆环(画图说明这点)。如果视点变化了位置,这个纹理必须要被重新计算。

回顾在前言中我们介绍过 OpenGL 中**扩展**的概念;OpenGL 中实现的许多扩展可以使得环境映射变得非常简单。

使用环境映射模拟高光

OpenGL 中可以使用反射映射在物体上产生镜面反射高光。可以创建拥有一个集中的高亮度的斑点的纹理。反射映射把这个高光画到物体表面使它看上去像环境中某处有一个真正的光源。这样产生的高光可以比 Phong 镜面反射项附加 Gouraund 明暗处理产生的高光更集中更富于细节。回顾 Phong 模型项只会在面片顶点处计算，很容易丢掉落在两个顶点之间的高光。反射映射使用的纹理坐标(s,t)是在顶点形成然后进行插值的，所以即使顶点索引的纹理坐标恰巧落在亮点的周围，这个亮点也会在面片上恰当地被渲染出来。

练习

练习 8.5.8　OpenGL 的计算环境映射纹理坐标的方法

推导等式(8.22)。图 8.36(b)给出了(视点坐标系下)相关的向量。视点位于方向(0,0,1)的远处。球体的半径是 1，位于 z 的负半轴。假设光从方向 r 射过来，击中球体上的点(x,y,z)。球面上这点的法向是(x,y,z)，并且沿方向 **r** 过来的光正好反射到方向(0,0,1)。这意味着法向必须等分 **r** 和(0,0,1)，或者说必须是这两个向量相加后的方向，所以$(x,y,z)=K(r_x,r_y,r_z+1)$，对于某个 K：

a. 证明当 $K=1/p$ 时，法向量是单位长度。其中 p 在等式(8.22)中给出。

b. 证明因此$(x,\ y)=(r_x/p,r_y/p)$；

c. 假设当前纹理图像坐标 x 和 y 的范围是 -1 到 1。证明为什么我们希望在点(x,y,z)反射后所看到的等于纹理图像在(x,y)的值。

d. 证明，现在纹理坐标的范围是 0～1——也就是 *OpenGL* 使用的，我们希望在(x,y)看到的值在纹理图像上的坐标(s,t)就是等式(8.22)给出的。

8.6　给物体添加阴影

8.6.1　阴影简介

阴影使图像看上去更真实。由日常经验可知，一个物体投射阴影到另一个物体的方式，对于判断它们的位置提供了重要的视觉线索。图 8.64 包括了两个图像，悬浮在平面上的一个立方体和一个球体。图 8.64(a)的图没有阴影，就不可能看出立方体和球体悬浮在平面上多远的距离。作为对比，图 8.64(b)可以看到阴影，对判断物体的位置提供了有用的提示。一个阴影能够传达很多信息；近似于第二次观察物体(从接近光源的视点的角度)。

正如我们将要看到的，有一些在图像中生成阴影的方法。不幸的是，本章中所使用的工具只适用于点光源投影到平面上时产生的阴影。在第 12 章中，光线跟踪工具对于任何光源投射到任何形状表面上的情况，都能计算出准确的阴影形状。

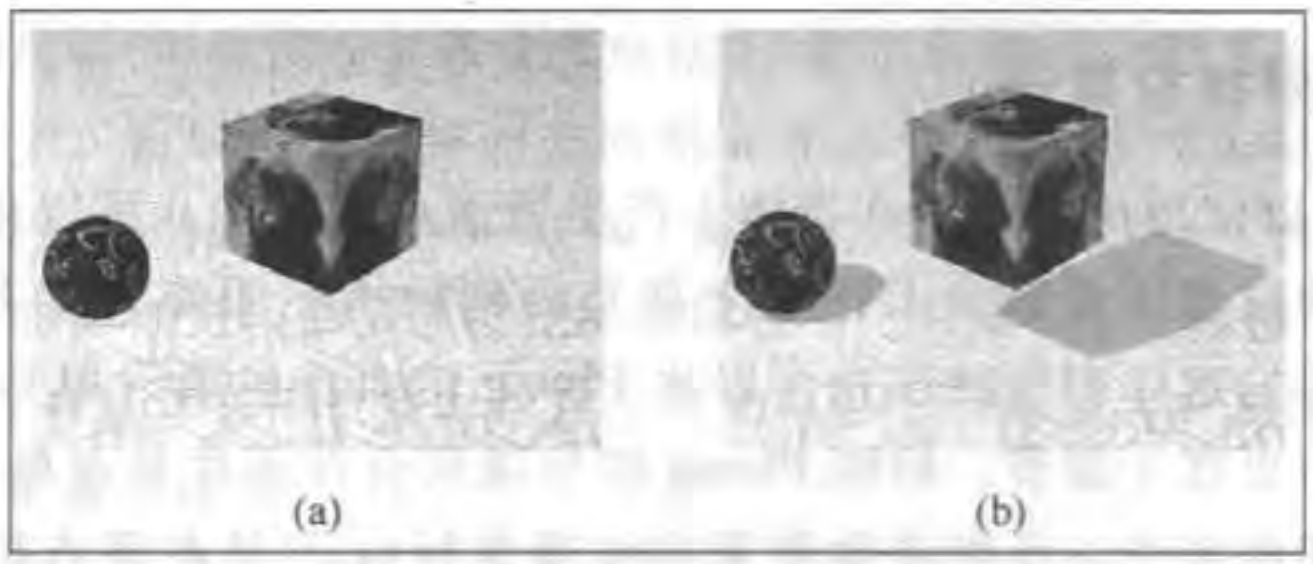


图 8.64 阴影的效果

(a) 悬浮的立方体和球体,没有阴影;(b) 有阴影的效果

作为纹理的阴影

这种技术显示了通过一个点光源投射到平面的阴影。问题是如何计算出阴影的形状。图 8.65(a)显示了一个盒子投射阴影到地板上。把光源作为投影中心,阴影的形状由盒子的每个面到地板平面的投影决定。事实上阴影是 6 个面投影的集合。图 8.65(b)显示了其中两个面的叠和:顶面投影到 top',前面投影到 $front'$(勾画出其他四个面的投影,可以看出它们的集合就是所需要的阴影)。

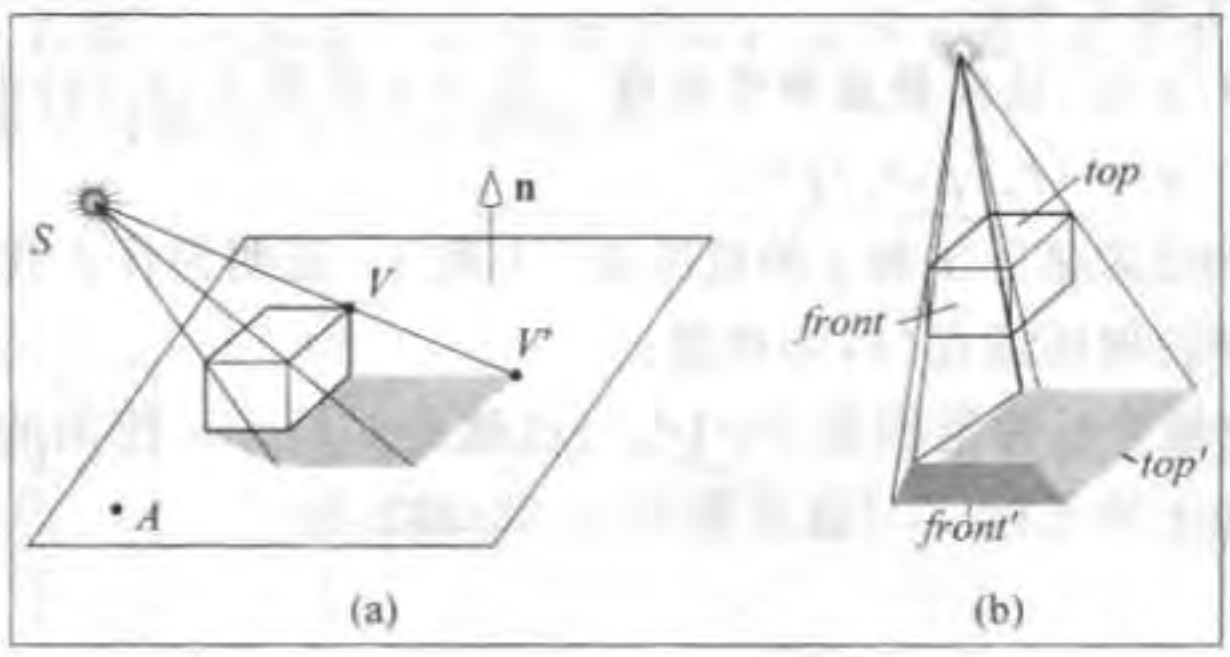


图 8.65 计算阴影的形状

这是画阴影的关键。在使用环境光、漫反射、和镜面反射光绘制平面后,只采用环境光重新绘制 6 个面在平面上的投影,就能得到阴影的正确形状和色彩。最后画盒子。(如果盒子接近平面,盒子的一部分将挡住部分阴影。)

构建"投影"面

通过 F 生成新面 F',投影它的每个顶点到平面上。我们需要一种计算出这些顶点在平面上位置的方法。假设,如图 8.65 所示,平面经过点 A,其法向量为 $\mathbf{n}$。认为投影顶点 V,产生点 V'。这里的数学原理是熟悉的:点 V' 是在 S 的光源发射的光线经过 V 到达平面的点。就像练习中所做的,这个点是:

$$V' = S + (V - S)\frac{\mathbf{n} \cdot (A - S)}{\mathbf{n} \cdot (V - S)} \tag{8.23}$$

这个练习说明怎样在同一坐标系中写成一个 V 次矩阵,这对于绘制引擎有利,就像支持矩阵乘法的 OpenGL。

练习

练习 8.6.1　阴影形状

假设一个立方体悬浮在一个平面上。这个立方体的阴影是什么形状的？如果点光源位于(a)直接在顶部上面？(b)沿着立方体的主对角线(看作是各向同性的视图)？勾画出悬浮在平面上的一个球体和一个圆柱体在各种位置的光源中的阴影。

练习 8.6.2　制作阴影面

(a) 证明：从 S 点的光源发出的光线经过顶点 V 到达平面 $\mathbf{n}\cdot(P-A)=0$ 在点 $t^*=\mathbf{n}\cdot(A-S)/\mathbf{n}(V-S)$。

(b) 证明：这就是等式(8.25)给出的点 V'。

练习 8.6.3　它等价于一个矩阵乘法

(a) 证明：V'在等式(8.23)中的表达式能够被写成矩阵乘法：$V'=M(V_x,V_y,V_z,1)^T$，这里 M 是一个 4×4 的矩阵。(b) 用 A,S 和 $\mathbf{n}$ 表达 M。

8.6.2　利用阴影缓冲区绘制阴影

另一种完全不同的绘制阴影的方法是采用深度缓冲区完成隐藏面的移除。对于每个光源，它使用一个辅助的二次深度缓冲区，称为**阴影缓冲区**。这需要大量的存储空间，但这种方法并不仅限于投射阴影到平面上。

这种方法是基于隐藏在光源后的任何点都一定位于阴影中的原则。在另一方面，如果没有物体位于点和光源之间，那么这个点不在阴影中。阴影缓冲区包含视点在光源处的场景的“深度图片”：它的每个元素记录了从光源到最近物体的连接方向的距离。

绘制分如下两个步骤。

1. 阴影缓冲区装载。

阴影缓冲区每个元素初始化为 1.0，这是可能的最大伪深度。然后，假设照相机定位在光源，扫描转换场景中的每个面，但是只检测面上的点的伪深度。阴影缓冲区的每个元素记录下能被看到的最小伪深度。

更特殊的情况时，图 8.66 显示了位于光源的“人眼照相机”看到的场景。假设点 P 在从光源到阴影缓冲区“像素”$d[i][j]$ 的光线上，金字塔上的点 B 也在这条光线上。如果金字塔在，则 $d[i][j]$记录的是到 B 的伪深度；如果金字塔不在，则 $d[i][j]$记录的是到 P 的伪深度。

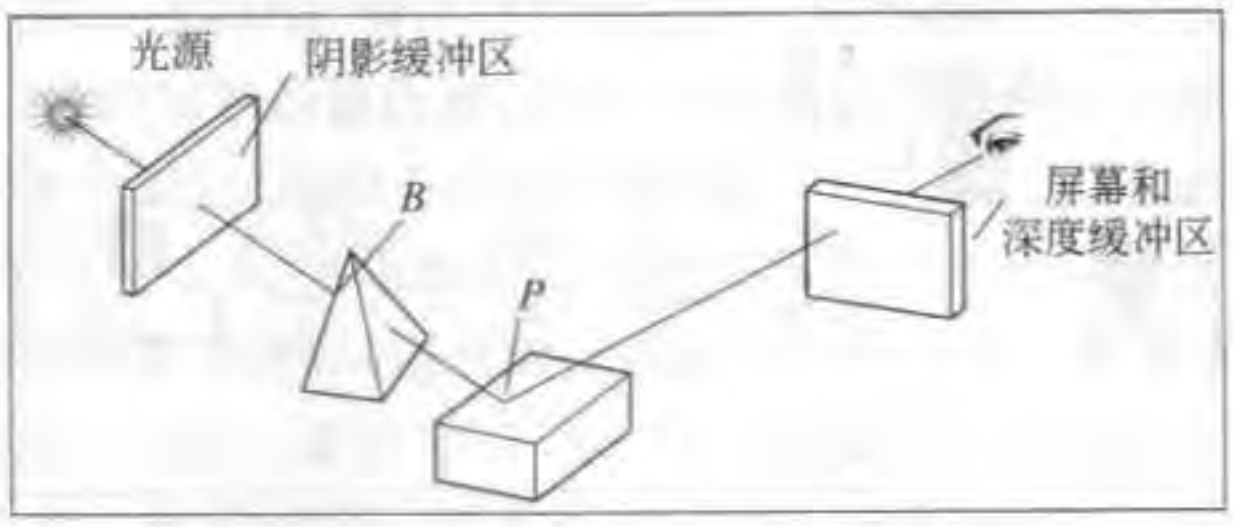


图 8.66　使用阴影缓冲区

注意阴影缓冲区的计算是与眼睛的位置无关的,因此在一个动画中若只有眼睛移动,阴影缓冲区只需要被装载一次。但是只要物体相对于光源移动,阴影缓冲区就必须被重新计算。

2. 绘制场景。

绘制场景中的每个面通常采用人眼相机。假设相机穿过 $p[c][r]$看到 P 点。当绘制 $p[c][r]$时,我们必须要找到:

- 从光源到 P 的伪深度 D;
- 在深度缓冲区被测试的索引位置$[i][j]$;
- 储存在深度缓冲区中的值 $d[i][j]$。

如果 $d[i][j]$小于 D,则点 P 在阴影中,$p[c][r]$只需要利用环境光设置。反之则点 P 不在阴影中,$p[c][r]$要使用环境光、漫反射光、和镜面反射光设置。

这些步骤怎样实现?如练习中所描述的,人眼相机视平面的每个点对应于光源相机视平面的一个点。对于每个屏幕像素,找出光源到 P 的伪深度,也包括索引$[i][j]$计算出存储在阴影缓冲区的最小伪深度。

练习

练习 8.6.4 计算来自光源的伪深度

假设矩阵 M_c 和 M_s 分别将场景中的点 P 变换到人眼相机视平面和到光源相机视平面上合适的(3D)点:

a. 叙述如何建立一个光源相机和怎样找到结果矩阵 M_s。

b. 转换:给定人眼相机视平面上的点(x,y),找到在光源相机视平面上的位置(i,j)和深度。

c. 给定(i,j),怎样确定光源相机上的索引$[i][j]$和 P 的伪深度。

练习 8.6.5 扩展光源

本章我们只考虑点光源的情况。而现实中更多的是各种类型的其他光源。如图 8.67(a)所示,这样的光源投射更复杂的阴影,有一个任何光源看不到的全影区,和一个稍亮点的部分光源可见的半影区。在图 8.67(b)部分,一个炽热的半径为 2 的球形光源照射在单元立方体上,从而投射一个阴影到墙面 W 上,准确地画出墙上的全影和半影。如你所想的,绘制阴影的算法因为扩展的光源变得复杂。参阅【Watt92】,可获得完整的介绍。

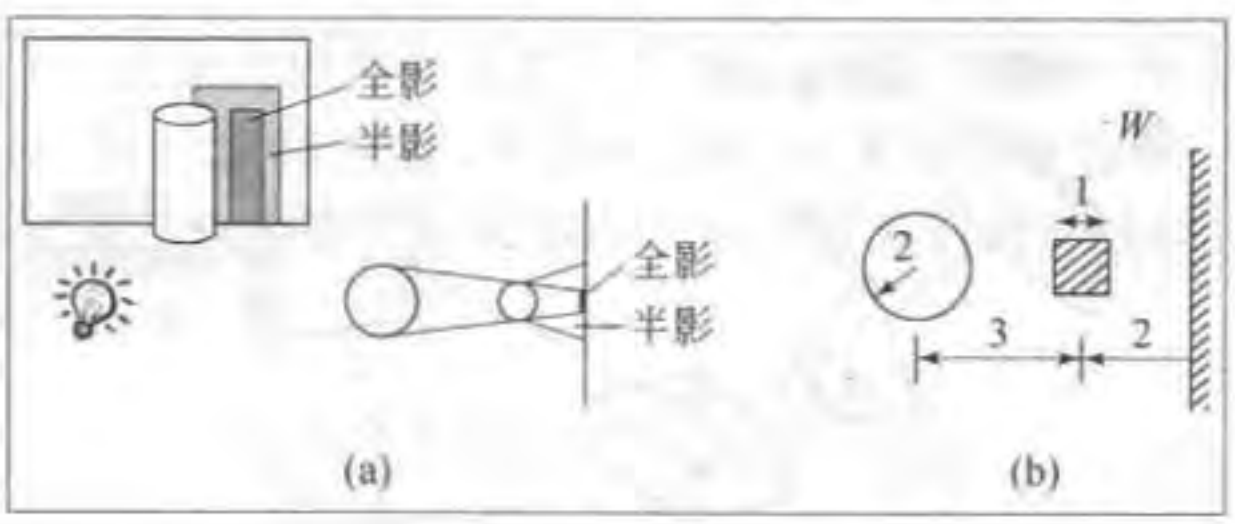


图 8.67 扩展光源的全影和半影

接下来我们将介绍图像制作的辐射度方法，并与光线跟踪算法作对比。这里我们只给出辐射度方法的简要介绍，因为大量需要解释的细节超出了本书的范围。

8.6.3　辐射度的简要介绍

回顾一下，环境光包含从物体的每个面上反射的光分量的计算，可称为“包罗万象”。它试图包括场景中许多光束在各种表面上产生的多重反射。一种可行的绘制真实感改进方法是形成一个精确的关于到达每个表面的光能量与离开表面光能量的对比的模型。然而由于跟踪每条光线的代价太大了，所以把它们合并成一个环境光分量。环境光是一种假想的光属性，现实生活中不存在。图 8.68 展示了一个光线跟踪的场景的例子：图 8.68(a)展示了一个用辐射度方法生成的场景；相对的是，图 8.68(b)采用光线跟踪，而没有用辐射度。

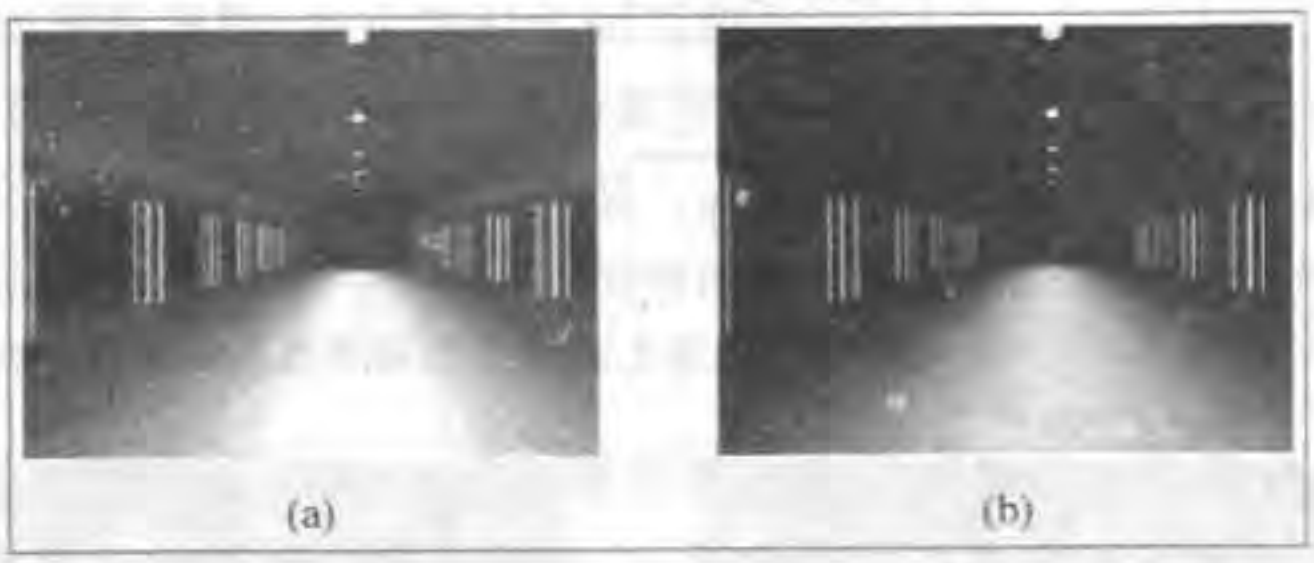


图　8.68

(a) 用辐射度；(b) 没有用辐射度

第 12 章我们将看到采用光线跟踪算法能显著提高真实感的水平，可以生成一些特殊的效果。特别是当辐射度取代光线跟踪会制造出更高等级的真实效果，但代价是增加了计算量。

一些资源可在本书网站上获取。也有一篇 Paul Nettle[18] 写的文章，对辐射度的算法及其复杂度作了精彩总结。

8.7　OPENGL2.0 和着色语言(GLSL)

回顾前言，有一个用 *OpenGL* 提高核心功能的机制：

- **扩展**—这允许对 OpenGL 做出改变和提高。那些想测试和提高 OpenGL 的公司，定义和创造扩展，例如新的算法集，函数，和有权使用新硬件的能力。

例如，可编程的着色部分(不要与阴影混淆)是一个扩展，它提供程序员一些直接调用图形卡的函数。在 OpenGL 2.0 中，可编程着色扩展成为 OpenGL 的核心。这些新功能的加入，允许程序员直接控制着色操作，避开固定的着色函数管道。最复杂的着色方法包括凹凸贴图和 3D 纹理，但是 OpenGL 着色语言(GLSL)使得程序员能轻易实现这些工作的大部分。图 8.69 是一个简化的 OpenGL 管道外加 GLSL 的简单版本。管道开始先输入多边形(顶点)的集合，接着是**顶点处理器**。顶点处理器运行**顶点着色代码**。顶点着色代码是处理

输入顶点数据的代码;数据包括位置,颜色,法线,等等。通过顶点着色代码,一个程序可完成如下任务:

- 通过视图矩阵和投影矩阵控制顶点位置转换;
- 转换法向量,如果需要的话将它们归一化;
- 生成并转换纹理坐标;
- 对每个顶点应用光照模型(环境光,漫反射,和镜面反射);
- 计算颜色。

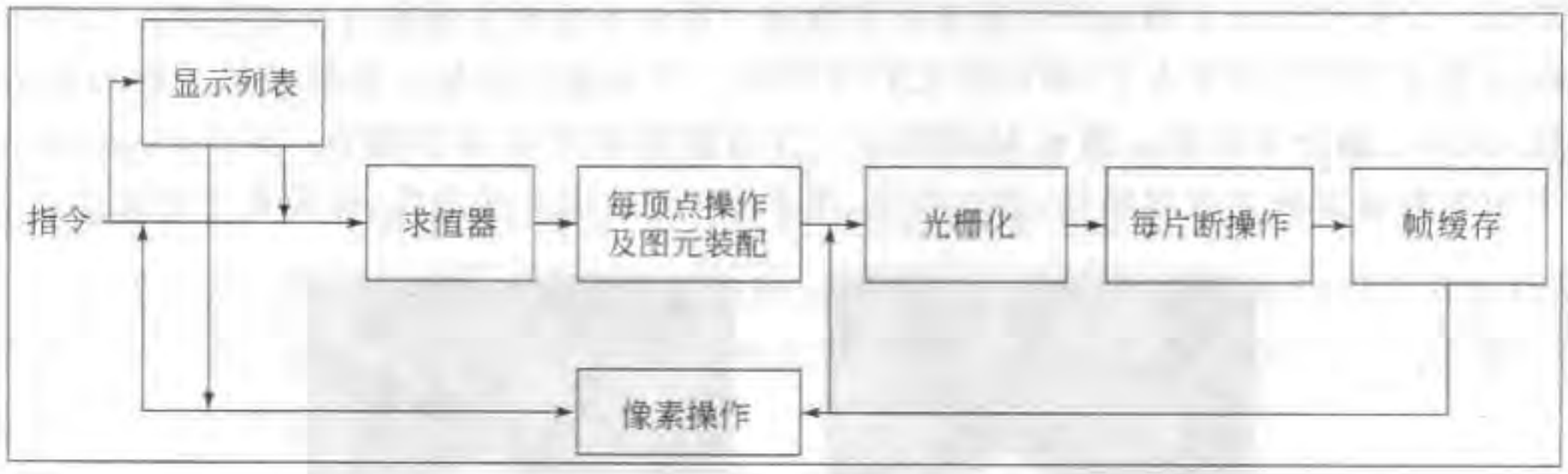


图 8.69 OpenGL 管道,加上顶点处理器和像素处理器

注意:当程序员设计一个定制的顶点着色器时,OpenGL 的顶点处理器的固定功能性被取代。如果定制的顶点着色器被激活的话,它将负责顶点处理的所有需要功能。

一旦这些顶点被转换成视平面,图元就被光栅化,并形成片断。例如,一个图元就像一个点可能覆盖屏幕上 1 个像素,或者一条线可能覆盖屏幕上 5 个像素。由点产生的片断包含一个窗口的坐标和深度信息,及与之相关联的属性包括颜色,纹理坐标,深度,等等。光栅化也确定了片断上的像素值。每个这样的属性值通过对顶点上对应属性值插值得到。一个跨越 5 个像素的线图元生成 5 个片断,这些片断的属性通过线段端点之间的对应值插值而得到。一些图元从不产生任何片断,反之另一些图元产生大量的片断。关于 OpenGL 管道在图 8.69 中"光栅化"阶段的输出,是一个片断流(这里的管道实际上是指 OpenGL 怎样操作的建议)。

8.7.1 凹凸贴图

凹凸贴图技术首次是被 Jim Blinn【Blinn88】提出的。它是一种给物体添加复杂纹理的高效方法,不需要改变物体的基本几何结构。例如,图 8.70 显示了一幅桔子粗糙表面的放大视图。然而,桔子的真实几何结构仍是一个光滑的球体,只是在球体上绘制了纹理。在每一点,纹理扰动了表面的原始法向量,从而扰动了在镜面高光计算中起重要作用的法方向。

重要的是要注意到物体的几何形状仍是一个简单的球形(很容易建模),与此相反的是扰动纹理可以是一个图像或者其他的随机纹理源,正如我们前面讨论的那样。

图 8.71 显示了另一个例子,表面看上去像是雕刻了数字,"1914—1918"。但是,物体实际上是一个光滑的矩形,刻痕通过扰动法向量产生。可以想象如果通过改变原始几何体(可能是个复杂的网格)来取得同样的效果将是多么的复杂。

图 8.70　桔子的凹凸贴图

图 8.71　平面表面上的浮雕

凹凸贴图是怎样制作的呢？给一个原始的扰动贴图，我们打算用它来扰动物体表面的法向量，我们必须一个一个像素地计算它对真实表面法向的影响。如 Blinn 所描述的那样，法向扰动要求重复使用偏导数，并在它们之间叉积。

这些计算超出了本书的范围。幸运的是某些 OpenGL 的扩展对凹凸贴图会有所帮助，使程序员免于原始的数学计算（偏导数，等等）。在 www. paulsprojects. net 有 Paul Baker 关于这方面的精彩指南。图 8.72 举了一个圆环的例子，分别采用和不采用它在指南中仔细讲解的凹凸贴图的技术。

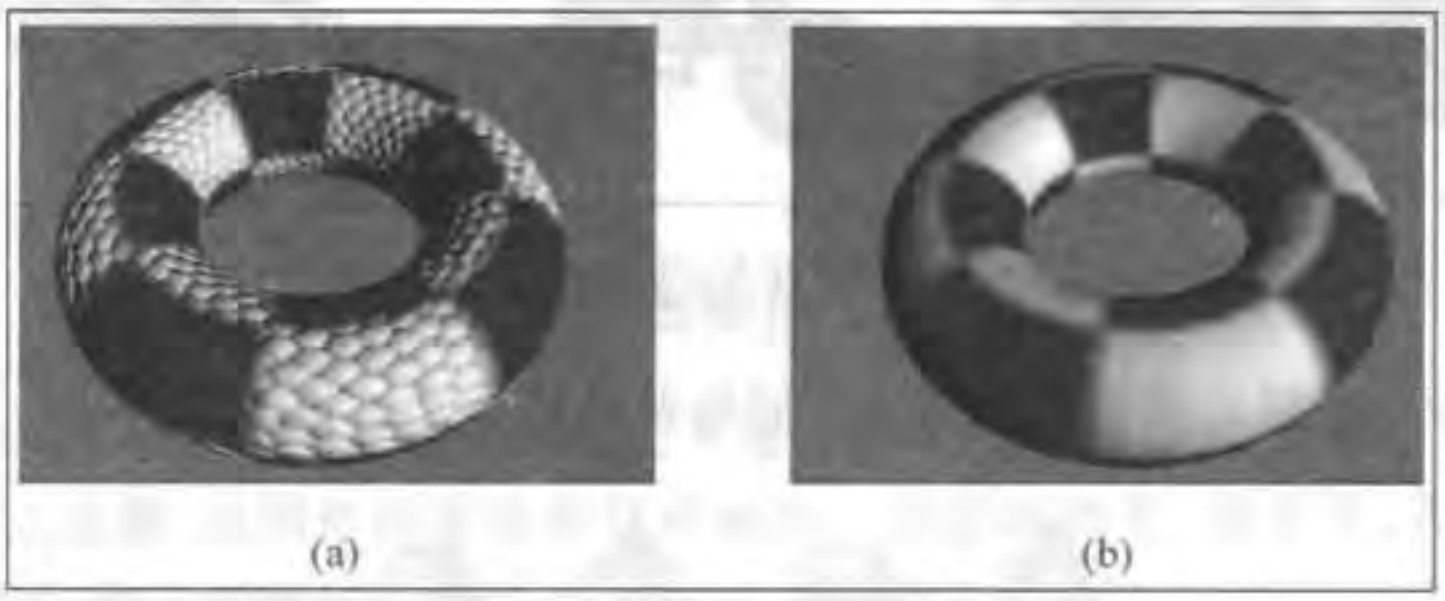


图　8.72

(a) 圆环的凹凸贴图例子；(b) 没有凹凸贴图的圆环

另一篇关于凹凸贴图的精彩文章是 NVIDIA 公司的 Mark Kilgard 写的，他原从事于 GLUT 的应用后来继续 OpenGL 的研究。Mark 的文章，"A Practical and Robust Bump-Mapping Technique for Today's GPUs"（能够在 www. nvidia. com 中找到），是一篇涉及凹凸贴图及其原理的精彩文章。

8.7.2　非真实感绘制

本书的重点是场景的真实感绘制，以求最终的图像看上去尽可能的真实。但在有些情况，真实不是图像最想得到的效果。相反，有人也许想强调一幅图像中的特殊信息，或者故意忽略一些绘制成本高和用户不感兴趣的细节。例如，有人想要制作一个类似卡通的发动机图片，如图 8.73 所示。这幅图像给人的印象是它像是手工用画笔画出来的。

在另外一些情况下有人也许想制作非常详细和准确的某一机器的技术图或一幅蓝图，不需要考虑着色、阴影、和反锯齿。这种技术被称为**笔墨绘制**（也可称作雕刻）。图 8.74 显

示了笔墨绘制帆船的画。

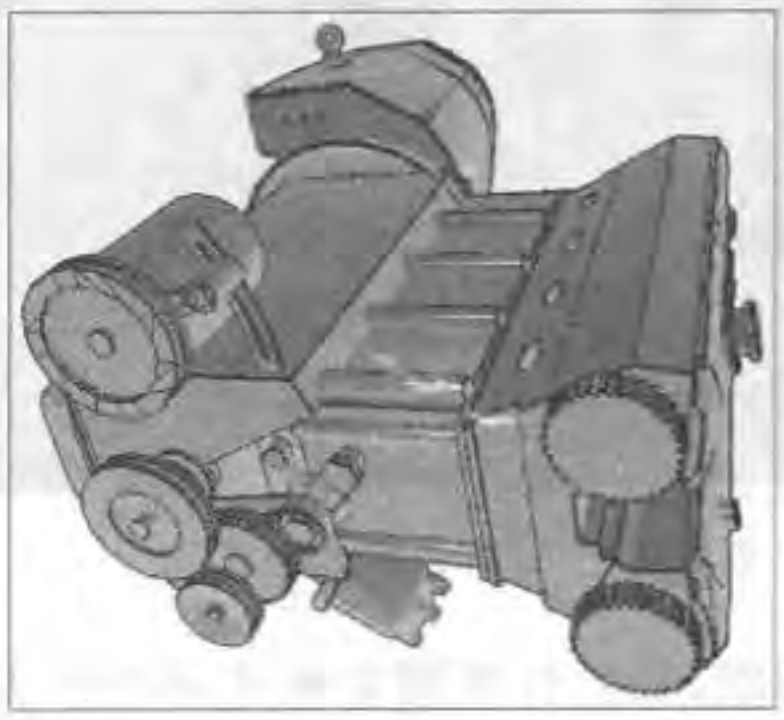


图 8.73 一个近似卡通的发动机图片

图 8.74 帆船的笔墨绘制

图 8.75 举了非真实感绘制的另外 3 个例子。注意这些图像必须非常像它们所代表的东西,尽管它们明显不像物体的真实照片。

图 8.75 拖拉机、摩托车、飞机的非真实感绘制

查阅本书的网站,可以找到关于非真实感绘制的资源和链接。

本章小结

计算机图形学从发展之初就对绘制 3D 场景的真实感有着很高的要求。物体线框表示可以被非常快速的绘制,但场景很难分辨,特别是几个物体相互重叠的情况。当用颜色填充面片和面消隐后,真实感有了很大的提高,但是这种方法绘制的图像在有光源照明的情况下依然不能给场景中的物体以突出印象。

所需要的是着色模型,以描述光线在表面的反射规律,这取决于表面的属性以及它与光源和摄像机的位置关系。光反射的物理原理是非常复杂的,所以程序员想出了一些近似方法和技巧,在大部分情况下在效果和速度上都可以接受。漫反射分量的模型是最接近现实物理效果的,但当考虑越来越多的因素时它变得极端复杂。高光反射根本不是基于物理原则的,但它们对于闪亮物体上的高光可以做恰当的近似。环境光非常的抽象,是避开处理物体之间多重反射的捷径,使得我们可以回避复杂的光线追踪计算。

即使是简单的着色模型也包含一些参数比如反射系数、表面粗糙度的描述和光源的色彩,OpenGL 提供了许多设置这些参数的方法。对于设计者,很少有如何选择这些参数取值的指导;往往通过反复试验直到最终达到满意的图像绘制效果来确定它们的取值。

在本章我们重点讨论多边形网格模型,最基本的任务就是绘制一个多边形。多边形的面

非常简单，用少量的数据就可以描绘，比如顶点位置、顶点法线、表面颜色和材质。另外还介绍了一个给多边形填充色彩的高效算法，特别是对凸多边形的情况。算法还能利用一个多边形的平整度来修改增加样式的深度值，使得深度-缓冲器隐藏面消隐算法更简单和有效。

当一个网格模型用来近似成一个原本平滑的表面时，表面上显现的边线效果是我们不希望看到的。Gouraud 和 Phong shading 提供了绘制平滑表面的方法（除了轮廓线）。Gouraud shading 速度快，但不能如实地重现高光；Phong shading 效果更真实但计算代价高。进一步研究 OpenGL 体系，Review Board（ARB）设计了一种着色语言来制作更真实的图像。更多细节可参阅 www.opengl.org。

场景的真实感程度可以通过在物体表面贴纹理得到极大的提高。纹理可以使物体看上去就像用某种材料比如砖块或木头制作的，标签或者其他外形也可以贴到表面上。环境贴图带给观众一个环绕闪亮物体环境的印象，这使得场景更真实，特别是在动画中。纹理贴图必须要认真制作，然而，需要采用正确的插值和反锯齿方法（如在第 9 章中所述）。

本章进一步讨论了物体阴影制作的效果。这是一个复杂的课题，现已经发展出了许多技术。通过更精细的技术比如光线跟踪可以得到更强的真实感。第 12 章阐述了这些技术的主要思想。

案例分析

案例 8.1　利用 OPENGL 制作着色物体

（难度：Ⅱ）设计一个程序：一个照相机飞越空间观看各种多边形网格物体。然后，在场景中设置一个点光源，给网格设置各种材质属性。包括环境光、漫射光、和镜面反射光分量。提供一个按键，能够在线框图和光滑着色图之间切换。

案例 8.2　纹理绘制

（难度：Ⅱ）升级案例 8.1 的程序，在网格物体表面绘制纹理。编写一个程序：能够读一个 BMP 图像文件，并把它贴到一个 OpenGL 纹理物体上。试验放五种不同的图像纹理和一种程序纹理到一个立方体的各个面，然后设置一个动画旋转立方体。提供一个按键，让用户能够在线性插值和校正插值之间切换来绘制纹理。

案例 8.3　升级 SDL 使其包含纹理

（难度：Ⅲ）SDL 场景描述语言不包括给物体表面指定纹理的方法。关键词纹理经常出现在 SDL 中，但它不做任何工作只是出现在文件中。认真研究 Scene 和 Shape 类的代码，可参阅本书网站，设计一种语法比如

```
texture giraffe.bmp p1 p2 p3 p4
```

从存储图像中制作纹理（这儿 giraffe.bmp），把它贴到后来定义物体的某一面上。确定纹理需要多少参数，和它们怎样使用。扩展 drawOpenGL()到 2 或 3 个形状，这样便于它准

确地对要求的物体贴图。

进一步阅读

要想更深入地了解最近的 OpenGL 着色语言,可阅读 Randi J. Post【Post02】所著的"OpenGL Shading Language"。还可阅读另一本最近的关于顶点和片断着色的书 Matt Pharr【Pharr05 所著的"GPU Gems 2"。另外,在图形学的经典著作 Jim Blinn 的"A Trip Down the Graphics Pipeline"【Blinn96】和"Dirty Pixels"【Blinn98】中,提供了众多的文章详细地介绍了阴影绘制和纹理绘制的双曲线插值的相关问题。

第 9 章

光栅显示工具

本章学习目标

- 讲述像素映射的概念和针对像素映射的操作
- 开发像素映射的复制、缩放、旋转工具
- 讨论各种绘制模式，例如 XOR 模式
- 开发图像合成的工具
- 介绍定义和处理区域的方法
- 讲述 Bresenham 直线绘制算法
- 构建区域填充工具——特别是多边形区域填充
- 讨论走样的产生和反走样算法
- 介绍用来产生更多颜色的抖动和误差扩散工具

预览

9.1 节和 9.2 节回顾了作为图像存储与处理的基本对象的像素映射，并讲述了针对像素映射的几种操作。9.3 节讲述像素映射合并的方法。像“异或”这种特殊的绘制模式也将在本节讨论。

9.4 节讲述 Bresenham 直线绘制算法。9.5 节到 9.7 节详细讲述像素映射中区域的描述和使用颜色或图案的区域填充算法。9.8 节讨论在光栅设备上显示图像时固有的走样（或锯齿）现象，和针对其开发的改善视觉效果的技术。

9.9 节介绍了抖动和误差扩散方法使得光栅显示器看起来比实际拥有更多的颜色。案例分析对上述主题进行了详尽讨论，并提出了一些重要的程序设计项目。

9.1 概　述

在本章中我们详细介绍如何从图元形成图像，以及如何处理图像以获得各种视觉效果。这里提到的图像由像素阵列组成，通常在光栅显示器上显示。

生成图像的方法主要有两种：

1. 扫描（和数字化）一幅已经存在的照片或者电视图像。
2. 用程序计算像素值，例如场景绘制。

在前面的章节中，我们主要关注于第二种生成方法：先生成如直线和多边形这样的图元，然后考虑如何为每个图元设置颜色和纹理坐标。利用 OpenGL，我们只需把它们发送到

"视口",这些图元就会显示在屏幕上。我们掩盖了为直线或多边形内部的每个像素确定颜色这一关键步骤,这件事由 OpenGL 来完成。

这种通过帧缓冲区一块区域的顶点位置和颜色等高级信息确定区域内部像素颜色的过程称为**扫描转换**(scan conversion)或**光栅化**(rasterization)。图 9.1 是图形绘制流水线的后端部分,光栅化就是其中一个过程。只要顶点经过视点变换后仍然可见,OpenGL 便将他们组合成图元(通过 glBegin (GL_POLYGON)或类似语句决定),然后对图元进行光栅化,即确定图元内部的所有像素的属性。在本章中我们将详细讲述这一过程是如何实现的,并介绍直线绘制以及使用颜色和纹理进行多边形填充的方法。

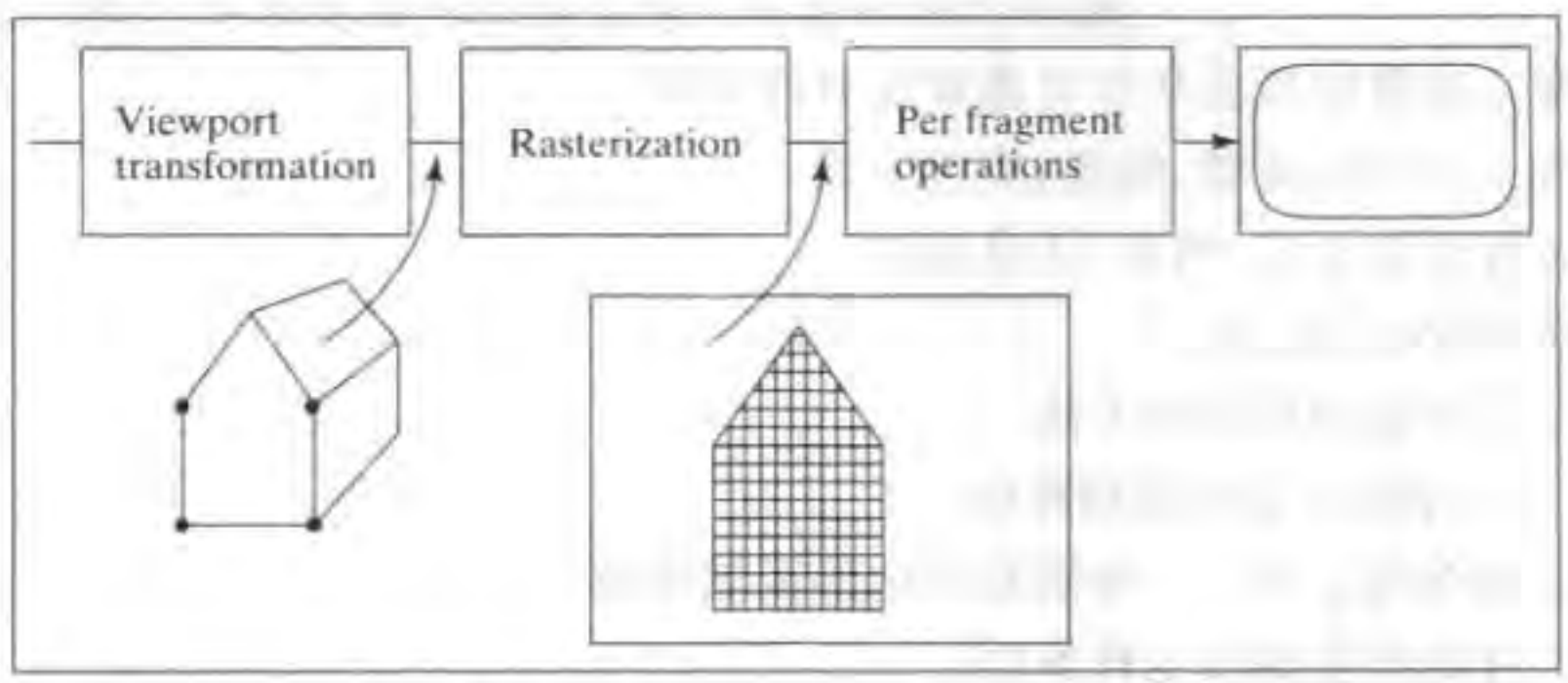


图 9.1 图形绘制流水线中的光栅化过程

事实上,光栅化过程生成的并非简单的像素,而是生成片段(fragment)。片段由颜色、深度和纹理坐标对组成。在把片段作为简单像素值写入帧缓冲区之前要对其进行一系列的操作和测试(见图中的"逐个片段操作")。我们将详细讲述各种可能的操作,并讨论它们的作用。

我们还要介绍在图像生成后如何对其进行各种处理,这包括基于一幅现存图像中(或要合并的几幅图像中)相关像素值计算新的像素值的操作。OpenGL 把这些操作放在图像处理流水线的"逐个片段操作"部分执行。

第 1 章介绍了光栅图像和一些相关的显示设备,在那里图像以像素映射(pixmap,pixel map 的缩写)的形式存储在内存中——一个矩形的数值数组。像素映射可以存储在计算机内存的任意区域,并且可以从一个地方复制到另一个地方。当像素映射复制到帧缓冲区后,扫描控制器将像素值转换为彩色亮度,像素映射就在显示设备上显示出来了。此时我们在屏幕上看到的实际上是帧缓冲区中所存数据的一个"画像"。因此,"像素"一词通常既指像素映射中保存的数值,也指显示器上这些亮点本身。

在前面章节中讲到像素映射中的每个像素值是定长的,而表示一个像素数据的比特长度被称为**颜色深度**(color depth)。如果一个像素具有 b 比特的颜色深度,那么它可以取 2^b 种不同的数值,因此最多可以表示 2^b 种不同的颜色。如果 $b=1$,则只有两种可能的颜色,例如黑与白。但是由于造价降低,现在很多视频显示器使用 32 位颜色深度,因此拥有 2^{32}(多达 8.1×10^{16})种可能的颜色。

光栅图像和光栅显示设备在处理空间和颜色数据时都是离散的。这种离散性,加上程序可以直接对内存像素操作的能力,导致一类特殊的图像生成和处理工具的出现。

9.2　处理像素映射

本节详细介绍像素处理的工具(包括在屏幕上,即处理帧缓冲区数据,也包括不在屏幕上,即在常规内存中),并描述这些处理过程产生的视觉效果。我们还会介绍 OpenGL 为处理像素映射提供了哪些工具。

9.2.1　对像素映射的操作

首先概括一下对像素映射都有哪些操作。在后续小节中,我们讨论这些操作如何实现,以及它们在不同情况下的效果如何。

绘制图像

通过绘制到帧缓冲区进行渲染的操作会改变显示设备上可见的内存映射[①]。当使用 OpenGL 绘制场景时,写操作会直接在帧缓冲区的像素映射上进行。

复制内存映射

像素映射可以从一段内存复制到另一段内存中。图 9.2 显示了四种复制操作,图中给出了四个可见的像素映射和它们的不可见副本。复制(copy)操作将图像从显示设备的一处复制到另外一处。读取(read)操作将显示图像的一部分复制到常规内存。绘制(draw)操作将常规内存的像素映射复制到帧缓冲区显示出来。还有一类操作称为内存复制(memCopy),它将一幅图像在常规内存中从一处复制到另外一处。

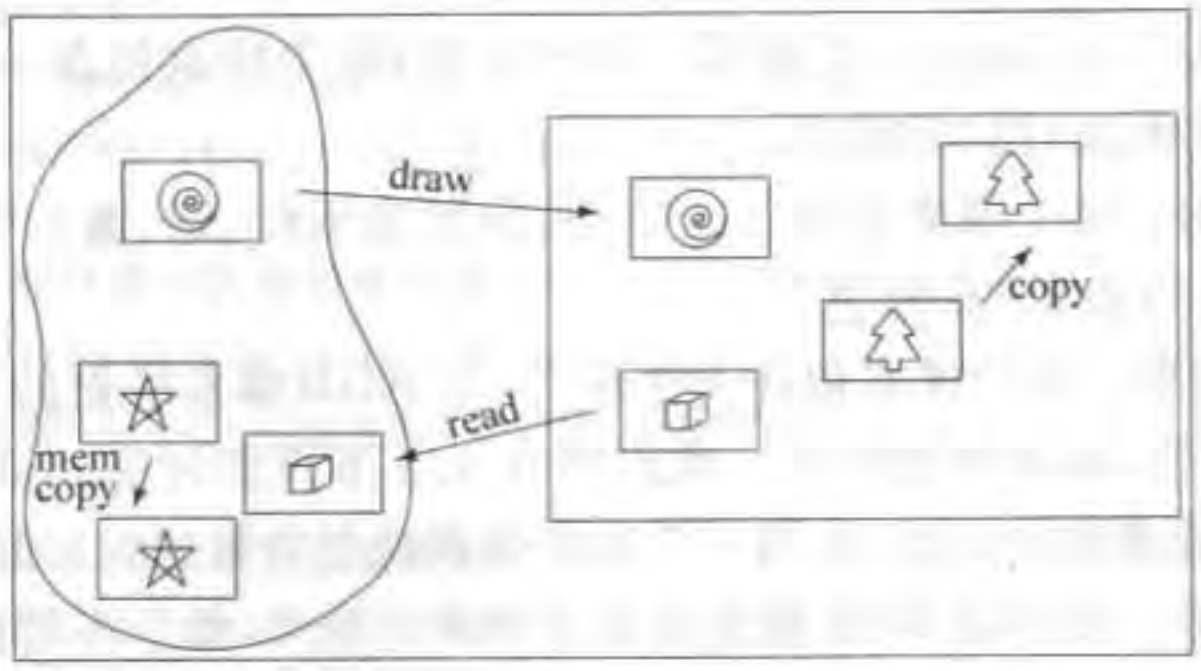


图 9.2　像素映射各种不同的复制操作

OpenGL 提供了一些函数来支持这些复制操作:

- glReadPixels();←读取帧缓冲区中的一个区域到常规内存。
- glCopyPixels();←在帧缓冲区中把一块区域复制到另外一个位置。
- glDrawPixels();←将给定的内存映射绘制到帧缓冲区中。

---

① 帧缓冲区内存有时也称为"屏幕上"内存,因为这些像素值和屏幕上显示的内容关系紧密。

缩放和旋转内存映射

放大、缩小和旋转图像有时是很必要的。根据缩放因子和旋转角度的不同,实现这些操作可能很简单,也可能相当复杂。

稍后我们会详细讨论。

比较两个内存映射

另一个重要的任务就是比较两个像素映射的差异。例如,通过比较两幅 X 光图像可以发现肿瘤是否发生变化。通常两个像素映射是逐个像素比较,同时在对应的像素对上执行某些数学操作。

在内存映射中表示一个区域并着色

一些像素映射包含的是易于识别的物体,例如圆、多边形和某些方面看上去一致的像素组成的区域(参考心脏、脑部、下颌的 X 光图像)。我们需要描述像素映射中的这些区域,并使用更高级、更抽象,或符号化的形式对它们进行表示。另外,我们还经常需要使用单一颜色或图案,为这些区域的内部像素进行着色,这被称为**区域填充**(region filling)。

9.2.2 对像素映射有用的数据类型

为像素映射的操作而设计一些类是很自然的。一个像素映射具有特定的行数和列数,其中每个像素又保存着一些特定的数据。下面是一些重要的像素映射实例:

- 位图(bitmap):每个像素占一位(bit),因此每个像素不是"开"就是"关"。
- 灰度级(grayscale)像素映射:每个像素占一个字节(byte),表示 0~255 的灰度级别。0 代表黑色,255 代表白色。
- LUT 索引(LUT indices):正如第一章的介绍,每个像素包含一个数字,这个数字是一个颜色查找表(LUT)的索引。
- RGB 像素映射:每个像素包含 3 个字节,分别表示红、绿、蓝三个颜色分量。通常认为这种方式可以表示"真彩色"。
- RGBA 像素映射:每个像素包含 4 个字节,与 RGB 像素映射的不同之处在于第4 个字节是 alpha 值,表示不透明度。我们将在 9.3 节详细讨论 alpha 通道的使用。

为了方便地操作像素映射,我们使用一个类将像素映射的数据封装起来,并提供一些成员方法来管理和操作图像。我们以 RGB 像素映射为例讲述细节,然后将其扩展到其他类型。为了充分利用 OpenGL 的图像处理工具,像素映射的数据格式应与 OpenGL 的格式保持一致[①]。

我们首先定义 RGB 类型,来保存一个像素值的 RGB 三元组:

```
struct RGB {unsigned char r, g, b; };
```

图 9.3 显示了 RGBpixmap 类的定义。我们遵循 OpenGL 的格式,将像素映射表示为一维的像素数组,像素的顺序从底到顶按行排列,每行之内按从左到右顺序排列。

---

① 实际上 OpenGL 提供了一系列方法来封装内存中的像素映射数据。我们在这里使用最常用的格式。

```
class RGBpixmap {
private:
    int nRows, nCols;                          //像素映射的尺寸
    RGB * pixel;                               //像素数组
public:
    RGBpixmap() { nRows=nCols=0; }
    RGBpixmap(int r, int c)                    //构造器
    {
        nRows=r;
        nCols=c;
        pixel=new RGB[r * c];
    }
    void setPixel(int x, int y, RGB color)
    {
        if (x>=0 && x<nCols && y>=0 && y<nRows)
            pixel[nCols * y+x]=color;
    }
    RGB getPixel(int x, int y)
    {
        return pixel[nCols * y+x];
    }
    //***在当前光栅位置(CRP)绘制像素映射
    void draw() {
        glDrawPixels(nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel);
    }
    //***将矩形区域像素值读入像素映射
    void read(int x, int y, int wid, int ht) {
        nRows=ht;
        nCols=wid;
        pixel=new RGB[nRows * nCols]; if (!pixel) exit(-1);
        glReadPixels(x, y, nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel);
    }
    //***复制部分显示区域回显示器上
    void copy(int x, int y, int wid, int ht) {
        glCopyPixels(x, y, wid, ht, GL_COLOR);
    }
    //***将 BMP 文件读取到像素映射中
    int readBMPFile(char * fname);
    //***将像素映射写入 BMP 文件
    void writeBMPFile(char * fname);

    //…其他内容…
};
```

图 9.3　RGBpixmap 类定义

图中定义了一些方法。默认构造函数创建一个空的像素映射，另一个构造函数创建一个 r 行 c 列的像素映射。setPixel()和 getPixel()的作用分别是设置和读取一个特定的像素值。

draw()、read()和 copy()方法直接通过 OpenGL 函数实现。

- draw()方法将像素映射复制到帧缓冲区,使得映射的左下角放在"当前光栅位置",这个变量可以通过 glRasterPos2i(x, y)来设置。
- read()方法执行反方向的复制,将一块矩形的帧缓冲区数据复制到像素映射中。区域的左下角位于"当前光栅位置"pt,wid 和 ht 指定区域的大小。read()首先分配保存这些像素所需的内存,由 glReadPixels()来完成真正的复制操作。
- copy()方法在帧缓冲区内将一个区域复制到另外一个位置,效果相当于依次执行了一次 read()和 draw(),只是没有生成中转的像素映射。被复制的区域左下角在(x, y),大小由 wid 和 ht 决定。复制到的新位置仍在帧缓冲区中,左下角在当前光栅位置。这是 9.3.4 节将要介绍的 bitblt 操作的一种形式。

利用 readBMPFile()和 wrtieBMPFile()可以很容易的创建和存储像素映射。readBMPFile()将保存在 BMP 文件中的图像读入像素映射,同时为其分配所需的内存。wrtieBMPFile()创建一个 BMP 文件保存像素映射。这两个方法的代码可以从图书公司的网站上获取。也可以从 http://nehe.gamedev.net 获取 loadBMP()函数,该网站还提供了一些例程,但与像素映射的函数稍有不同。

这个类虽然简单但是功能强大,下面的例子将说明如何使用这个类。

例 9.2.1 处理像素映射的测试平台

图 9.4 给出了一个使用 pixmap 类的应用程序,使用鼠标和键盘控制图像映射的读取和绘制。使用这个应用程序进行实验可以获得很多信息。这个程序开始运行时创建了两个像素映射 pic[0]和 pic[1],然后为其加载了两幅 BMP 图像。其中一个图像显示在初始的光栅位置。单击鼠标左键调用 draw()方法在鼠标点击位置再次绘制图像。如果按住鼠标左键在屏幕上拖动,像素映射则随着鼠标运动位置在屏幕上绘制出来。按下 s 键,在两种像素映射间切换。按下 r 键,调用 read(),读取屏幕上一块 200×200 大小的区域的内容,写入 pic[0],覆盖掉原来的内容。单击鼠标右键清屏。

```
//必须的头文件
RGBpixmap pic[2];                                   //创建两个(空的)全局像素映射
int screenWidth=640, screenHeight=480;
IntPoint rasterPos(100,100);
int whichPic=0;                                     //当前显示的像素映射
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<myMouse>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myMouse(int button, int state, int mx, int my)
{     //左键单击设置当前光栅位置
    if (button==GLUT_LEFT_BUTTON && state==GLUT_UP)
    {
        rasterPos.x=mx; rasterPos.y=screenHeight-my;
        glRasterPos2i(rasterPos.x, rasterPos.y);
        glutPostRedisplay();
    }
    else glClear(GL_COLOR_BUFFER_BIT);        //右键清屏
}
```

图 9.4 一个像素映射操作的测试程序

```
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<mouseMove>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void mouseMove(int x, int y)
{    //随着鼠标移动设置当前光栅位置
    rasterPos.x=x; rasterPos. y=screenHeight-y;
    glRasterPos2i(rasterPos.x, rasterPos.y);
    glutPostRedisplay();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<myReshape>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myReshape(int w, int h)
{
    screenWidth=w; screenHeight=h;
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<myDisplay>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myDisplay(void)
{
    pic[whichPic].draw();          //在当前光栅位置绘制图像
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<myKeys>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void myKeys(unsigned char key, int x, int y)
{
    switch(key)
    {
    case 'q': exit(0);
    case 's': whichPic=1-whichPic; break;          //切换图像
    case 'r': pic[0].read(0,0,200,200);break;      //屏幕抓图
    }
    glutPostRedisplay();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<main>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void main(int argc, char ** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowPosition(30, 30);
    glutCreateWindow("Experiment with images");
    glutKeyboardFunc(myKeys);
    glutMouseFunc(myMouse);
    glutMotionFunc(mouseMove);
    glutDisplayFunc(myDisplay);
    glutReshapeFunc(myReshape);
    glClearColor(0.9f, 0.9f, 0.9f, 0.0);           //背景色
    glClear(GL_COLOR_BUFFER_BIT);
    pic[0].readBMPFile("CokeCan2.bmp");            //创建一个像素映射
    pic[1].readBMPFile("Mandrill.bmp");            //创建另一个像素映射
    glutMainLoop();
}
```

图 9.4（续）

例 9.2.2 在屏幕上写文本

在光栅显示器中绘制字符的一种常用方法是为每个字符创建各自的像素映射。图 9.5 给出了一些例子。这些像素映射提供了字符的"图画",它通常定义在某一特定尺寸的像素阵列内,如 12 像素×8 像素,这些像素取值为 0 或 1。因此如果点阵尺寸较大,字符的形状容易控制,可以获得更加美观的字体。

程序初始化时,可以从硬盘加载各种字体到屏幕外内存,随时准备使用。当绘制字符串时,每个字符依次通过 draw()函数绘制到帧缓冲区的适当位置,同时根据该字符的宽度递增 x 的位置。对于按比例间隔的字体,每个字符都有一个特有的宽度(通常"i"间隔最小,"W"间隔最大)。这样字符显示就获得了最美观的效果。

例 9.2.3 窗口卷动

对于类似文字处理软件的应用程序中,屏幕上经常布满文字,一行接着一行。如图 9.6 所示,为了给屏幕底下新建的一行腾出位置,上方所有文字必须向上卷动一行(当然最上一行文字也随之消失)。为此,可以调用 copy()方法将除第一行之外的所有行的矩形,向上复制一行,并覆盖原来的内容。然后调用 draw()方法绘制一个空行到最下一行。

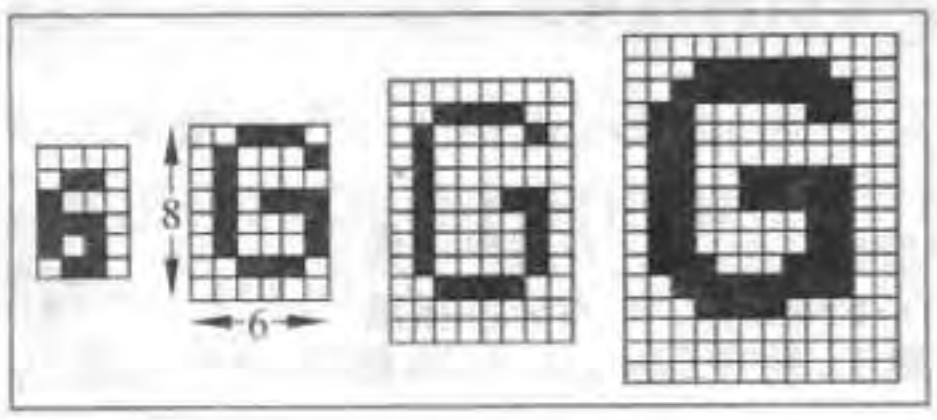


图 9.5 字符可以由二进制点阵构成:大的点阵可以构造更美观的字体从左到右点阵的大小分别是 4×6、6×8、8×12、12×16

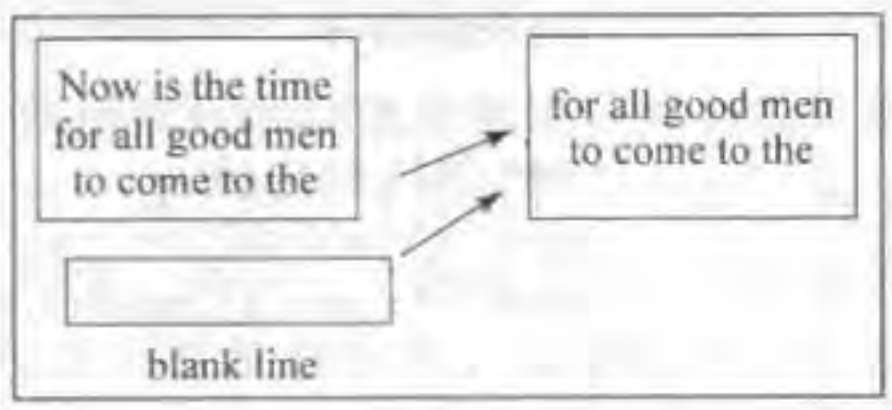


图 9.6 窗口内文字卷动

例 9.2.4 弹出菜单、下拉菜单和对话框

正如第 2 章所述,当今人们与计算机交互大多通过图形用户界面(GUI),它提供了多种菜单和对话框,用户通过鼠标选择要执行的动作。

当菜单被激活时,会弹出到视图中,暂时覆盖了部分屏幕。图 9.7(a)显示了屏幕被下拉菜单覆盖的结果。当用户做出了选择后,菜单消失,被菜单覆盖的区域就会重新出现(图 9.7(b))。当然,被遮挡的部分必须事先保存到常规内存,稍后才能被恢复。菜单覆盖区域恢复的流程如下:

1. 用户选择菜单标题。
2. 程序计算将要覆盖的矩形。
3. 调用 read(),将该区域复制到屏幕外内存。
4. 调用 draw(),在合适的位置绘制菜单。
5. 用户移动鼠标指针,点击需要的菜单项。
6. 程序将屏幕外内存中的像素映射复制到原来的位置,从而恢复被覆盖区域。

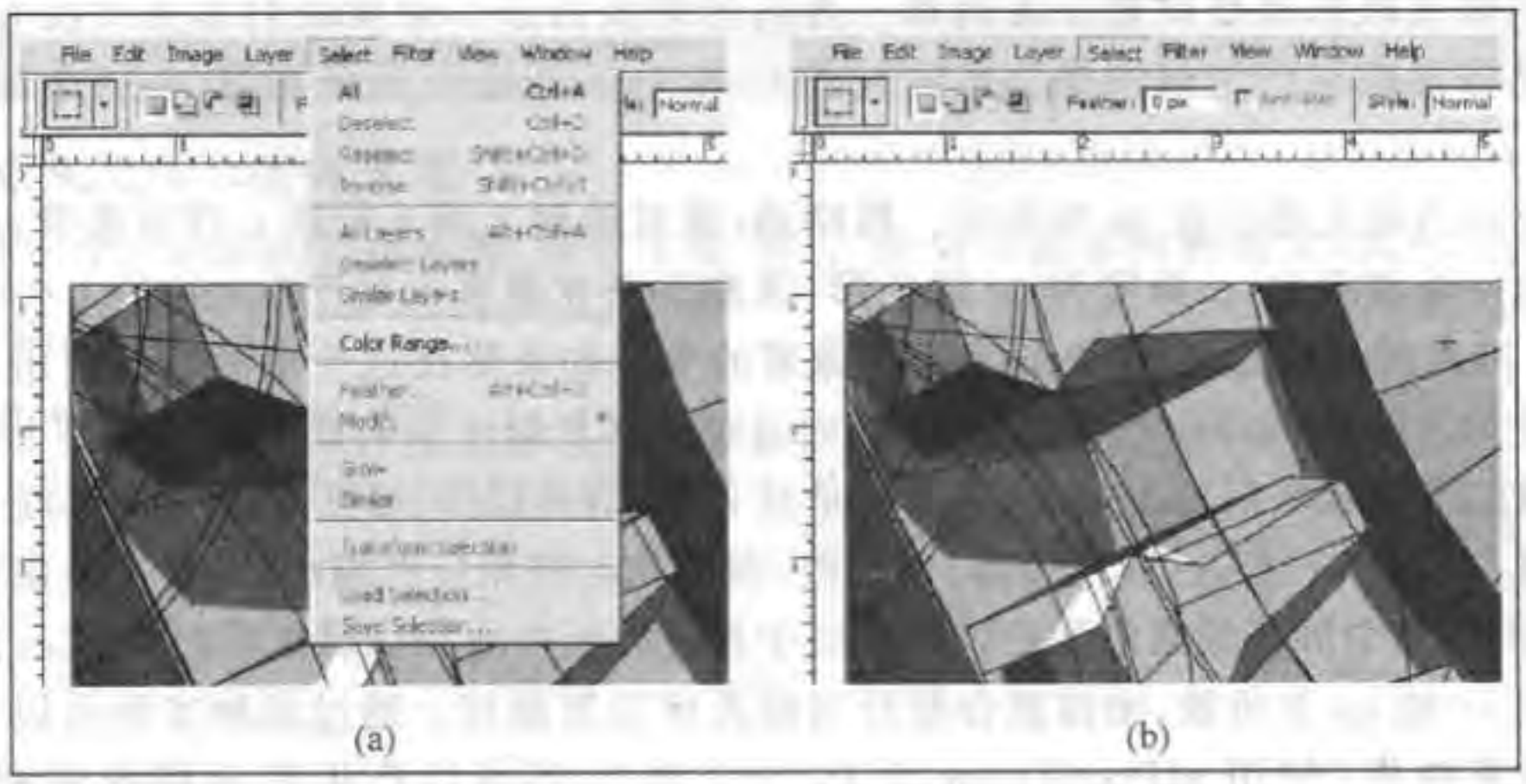


图 9.7　下拉菜单

9.2.3　图像缩放和旋转

图像的缩放和旋转操作是经常被用到的。例如，一个人造卫星拍摄的欧洲的图像可能是以 1600×1200 的像素映射格式显示，而我们要将其与以前扫描过 1540×1800 像素的图像进行比较。在比较之前，需要将小的那个图像放大，使两幅图像对应的部分能够正确的重叠起来。类似地，我们可能要比较一组数字化的指纹，但是发现一些指纹图像和其他图像比起来有轻微的旋转。为了对比不同的指纹，我们必须旋转图像使他们对齐。下面介绍一些简单的缩放和旋转操作。

以某个系数 s 对一个像素映射进行缩放的概念就是生成一个新的像素映射，其 x 和 y 方向的像素个数均为原来的 s 倍。用摄影术语说，如果 s 大于 1，就说这个像素映射被放大(enlarged)了；否则称这个像素映射被缩小(reduced)。当然像素本身的大小是不变的。当 s 是整数时，很容易通过像素复制(pixel replication)来放大像素映射。图 9.8 显示了一个字符位图两倍放大的例子。小位图中每个像素在大位图中生成一个 2×2 的像素阵列。对于其他整数 s，每个像素生成一个 $s\times s$ 的像素阵列。

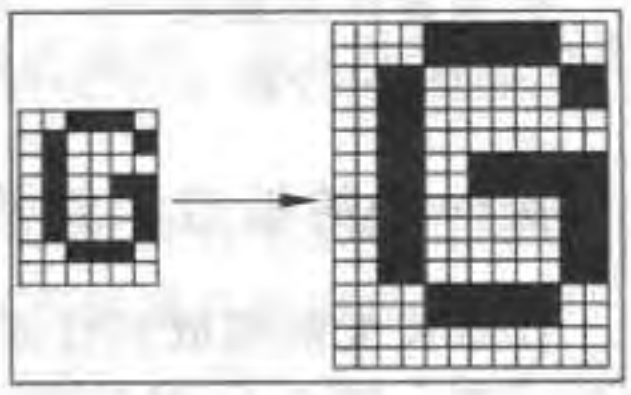


图 9.8　字符的两倍放大

以某个整数系数(即 $s=1/n$，n 是整数)缩小像素映射的过程也是很直观的。例如，对于 $s=1/3$ 的情况，可以通过在原始像素映射中每 3 行和每 3 列中保留一个像素来建立缩小的像素映射。按这种方式对原像素映射进行的“采样”会丢失信息，因此缩小的图像可能比原始图像的质量差很多。通常计算这 9 个像素的均值，并将其放在保留的位置上作为代表像素，会得到更好的效果。我们会在 9.7 节讨论包括这种思路的反走样技术。

OpenGL 提供了一个简单的机制来进行图像的缩放，这些图像使用 glDrawPixels()和 glCopyPixels()绘制(因此 draw()和 copy()函数可以看到缩放的效果)。函数

```
glPixelZoom(float sx, float sy);
```

为后续的像素映射函数设置 x 方向和 y 方向的缩放因子。像素映射本身并没有缩放;缩放的是从像素映射产生的图像。缩放因子 sx 和 sy 可以是任意浮点数,包括负值,默认值是 1.0。

缩放以当前光栅位置 pt 为基准。粗略讲,像素映射上第 r 行第 c 列的像素,在屏幕上被绘制成一个宽度为 sx 高度为 sy 的矩形,其左下角在屏幕上处于($pt.x+sx*r$, $pt.y+sy*c$)。准确的说法是,屏幕上任意一个像素的中心如果落在这个矩形内就使用该像素点的颜色表示。例如,如果 sx 是 2,sy 是 3,则通过像素复制生成的要绘制在屏幕上的像素映射的宽度是原像素映射宽度的 2 倍,高度是其 3 倍。它的左下角位置保持不变,还是在当前光栅位置。如果 sx 或 sy(或两者都)是小数,图像就会被相应的缩小,并且有一定程度的质量损失,因为每个屏幕像素的颜色仅仅取决于其中心所在的"矩形"像素的颜色。

如果 sx 或 sy 是负数,图像就会相对当前光栅位置翻转。通过这种方法可以显示一个倒置的像素映射(使用 glPixelZoom(1.0, −1.0)),还可以产生其他特殊的视觉效果。图 9.9 是测试平台的一个截图,并排显示了一幅图像的四个缩放后的结果(使用缩放系数分别为 sx=1.0, −1.0, 0.5, 1.5)。生成每幅图像时首先设置 sx,然后执行 glPixelZoom(sx, 1); glutPostRedisplay()。

图 9.9 通过修改 x 方向缩放尺度,生成一个像素映射的多个不同版本

1/4 周旋转

把像素映射旋转 90°、180°和 270°的过程非常简单:只要生成一个新的像素映射,然后使用 getPixel()和 setPixel()将像素从一个映射复制到另外一个。参见练习。

更一般的缩放和旋转

任意尺度的缩放和任意角度的旋转情况会更加复杂。假定我们要生成某个像素映射的变换结果。图 9.10 显示了一个像素映射 S(源),我们要对其施加变换 T,T 可能是一个混合了旋转和缩放的仿射变换。变换的结果是像素映射 D(目标)。现在的问题是如何正确计算 D 中每个像素 p 的颜色值。最简单的方法是对 D 中的每个像素 p,寻找 S 中覆盖了 $T^{-1}(p)$ 的像素的颜色,这个颜色就作为 D 中像素 p 的颜色值。然而仅仅使用这样的"采样"方法可能产生严重的锯齿,变换结果的质量可能会很差,这也是反走样部分要解决的问题之一。

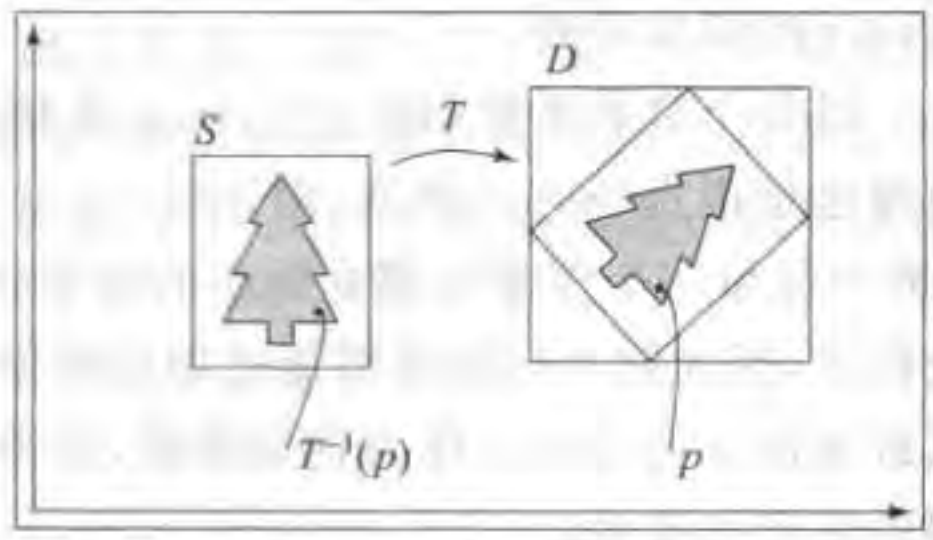


图 9.10 计算变换后的图像

练习

练习 9.2.1　设计字符'R'

设计一个 8×16 的二值图像，显示字符'*R*'。使用 1 和 0 两种模式，其中 0 表示白色，1 表示黑色。

练习 9.2.2　设计字符

在 r 行 c 列的点阵内设计最好看的字符：a、A、r、R、& 和 3。按照下列点阵尺寸进行设计：

a. $(r, c)=(6, 4)$；

b. $(r, c)=(10, 8)$；

c. $(r, c)=(14, 12)$。

练习 9.2.3　缩小字符

一种把矩形像素映射中定义的字符缩小的方法是把阵列的行数和列数分别缩小为原来的一半，缩小的像素映射中每个像素必须模拟原来单元中对应的 2×2 的像素阵列。给出一个算法，从 12×10 的像素映射生成 6×5 的像素图案。一种方法是如果 2×2 的像素阵列中有两个或更多的像素是 1，就把对应的新像素设置为 1。这个规则是不是最好的呢？

练习 9.2.4　旋转 90°

考虑一个正方形的像素映射，我们使用数组 $A[i][j]$表示，i 和 j 的范围是 $0\sim N-1$。给出 A 逆时针旋转 90°时形成的像素映射 $B[col][row]$的表达式。

9.3　合并像素映射

在某些情况下我们需要通过合并两个像素映射生成一个新的像素映射。这个操作可以用于屏幕上移动鼠标、比较两幅图像、让一幅图像渐变到另一幅图像等。我们先看几个有重要实际意义的例子。

像素映射合并操作通常是逐像素(pixelwise)的——也就是说，在新旧像素映射的对应像素执行某种操作。明确地说，两个像素映射 A 和 B 根据下式合并形成第三个像素映射 C。

$$C[i][j] = A[i][j] \otimes B[i][j] \quad \text{对于每一个 } i, j$$

其中⊗表示某种操作。下面是几种不同的操作的示例：

- 计算两幅图像的平均值。这里⊗表示取 A 的一半和 B 的一半的和：

$$C[i][j] = \frac{1}{2}(A[i][j] + B[i][j])$$

- 计算两幅图像的差，用来确定两幅图像的差别。这里⊗表示取减：

$$C[i][j] = A[i][j] - B[i][j]$$

- 寻找一幅图像比另一幅图像亮的位置。这里⊗意味着"前者大于后者"：

$$C[i][j] = A[i][j] > B[i][j]$$

如果 A 中某一像素比 B 中对应像素亮，则 C 中对应像素取值为 1，否则取值为 0。

计算两幅图像平均值的一般形式是计算两幅图像的加权平均(weighted average)。对

于某个小数 f,像素映射 A 的权重为$(1-f)$,B 的权重为 f,则:

$$C[i][j]=(1-f)A[i][j]+fB[i][j]$$

举例来说,如果 $A[i][j]$的 RGB 分量是(14,246,97),$B[i][j]$的 RGB 分量是(82,12,190),则当 $f=0.2$ 时,有 $C[i][j]=(27,199,115)$。可以通过修改 setPixel()函数实现两个 RGBpixmaps 的加权平均。

例 9.3.1 从一幅图像渐变到另一幅图像

加权平均的一个有趣应用是两幅图像的渐变。开始的时候图像 A 完全显示。随着时间流逝,A 逐渐淡出,同时图像 B 逐渐出现在 A 上,直到最后显示内容只有图像 B。如果使用 t 代表时间,随着 t 从 0 平滑过渡到 1,在时刻 t 显示的图像为:

$$A(1-t)+Bt \tag{9.1}$$

这个过程非常类似于第四章介绍的"插值"。图 9.11 显示了 $t=0$、0.25、0.5、0.75、1 等五个时刻显示的图像。案例分析 9.2 讨论了一种利用 OpenGL 的 alpha 通道功能实现两幅图像渐变的方法。9.3.2 节将详细介绍 alpha 通道。

图 9.11 两幅图像的渐变

9.3.1 读-改-写循环

在把像素映射 D 和 S 合并形成一个新像素映射 C 的过程中,有一种特殊情况就是 C 和 D 相同,即合并的结果又放回到 D 中。这个合并过程可以表示为 $D=D\otimes S$:像素映射 D 中的像素与 S 中的对应像素合并,得到的结果放回到 D 的该像素位置。这个过程被称为读-改-写循环,因为 D 中的像素首先从内存中读出,然后与 S 中对应像素合并而发生改变,最后结果写回 D。图 9.12 用符号说明了 D 本身是帧缓冲区时,这个循环的工作流程。合并过程就是依次对 D 中每个像素应用这个循环的过程。采用一些硬件配置能使这个操作效率非常高。后面将会看到,OpenGL 提供了一个指令可以完成整个像素映射的合并操作。

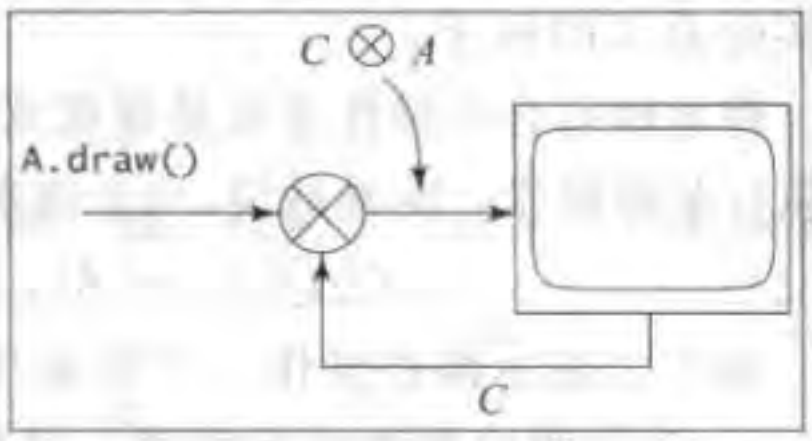


图 9.12 应用于帧缓冲区的读-改-写循环

9.3.2 alpha 通道和图像融合

两幅图像加权求和实际上是图像融合(blending)或合成(compositing)的一个特例。融合操作可以在一幅图像上绘制另外一幅半透明的图像,其关键是为 RGB 颜色模型添加第四

个分量——alpha 值。参照前面表示像素的数据类型，我们将 RGB 类型扩展为 RGBA 类型：

```
struct RGBA {
    unsigned char r, g, b, a;
};
```

alpha 值 a 的取值范围是 0～255，通常表示像素的“不透明”程度：值为 0 表示完全透明，值为 255 表示完全不透明。需要注意的是，RGBA 类型的像素映射需要的内存是同样的 RGB 类型像素映射的 4/3。像素映射中所有 alpha 值放在一起的集合通常称为 alpha 通道。

后面将会看到，alpha 分量最常见的用法是作为一个在 0～1 之间的缩放系数，因此实际使用的值是一个小数，$a/255$。

例 9.3.2　半透明图像覆盖

图 9.13 给出了一个在背景图像 D 上覆盖一个由面具和龙组成的图像 S 的例子。当源图像 S 创建时，龙所在像素的 alpha 值设为 255(完全不透明)，面具所在像素的 alpha 值设为 128(半透明)，所有其他像素的 alpha 值设为 0。当 S 和 D 融合时，结果显示出龙在背景 D 上，同时有一些背景颜色从面具中“显露出来”。

图 9.13　把图像 S 合成到图像 D 上

alpha 融合是一个源像素和目标像素加权平均，并写回目标像素的过程，也就是一个读-改-写循环。用表达式表示就是 $D=aS+(1-a)D$，其中 a 是源像素的 alpha 值，为某个 0～1 之间的小数。但实际上每个像素的 alpha 值可能都是不同的，而且 S 和 D 都有红、绿、蓝三个分量，因此更精确的写法是这样的，以第 j 行第 i 列上的最终目标像素的绿色分量为例：

$$D[i][j]\cdot g = aS[i][j]\cdot g + (1-a)D[i][j]\cdot g \tag{9.2}$$

其中 $a=\frac{S[i][j].a}{255}$。特别注意的是，a 值与 i 和 j 有关，即不同像素的 alpha 值也是不同的。红色和蓝色分量融合的表达式也是类似的。

要在程序中实现这种融合操作，我们首先要把前面介绍的 RGBpixmap 类扩展成 RGBApixmap(参见练习)，然后添加 blend()方法，通过一个读-改-写循环实现这种覆盖功能。像素映射 S 和 D 的融合便可以使用下面的代码完成：

```
D.draw();                    //和以前一样使用不透明方式绘制 D
S.blend();                   //使用 S 的 alpha 值,与 D 进行加权平均
```

OpenGL 提供了工具,使得 blend()更容易实现。只需设置一个"融合模式",指定源缩放系数和目标缩放系数。要把 a 设置为源 alpha 值,只需用如下方式指定:

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

该命令设置源缩放系数为源本身的 alpha 值,设置目标缩放系数为 1-源 alpha 值。图 9.14 给出了 blend()方法的一个简单实现。

```
void RGBApixmap::blend()
{
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_BLEND);              //启用融合
    draw();                          //绘制像素映射,并与目标融合
}
```

图 9.14 使用源图的 alpha 通道融合源图像和目标图像

例 9.3.3 模拟色度键:将某种颜色强制设为透明色

我们在电视上经常看到天气预报员站在一幅地图前讲解各种天气状况。实际上,这个预报员是站在一幅蓝色的背景前(图 9.15),同时一幅单独的气象图在同步的播放。电视信号在人的图像与地图图像之间切换:当线扫描中遇到蓝色就切换到地图的信号,否则显示人的画面。

图 9.15 在电视中使用的色度键

我们可以利用像素映射模拟这种效果,通过指定某种颜色的 alpha 值为 0 ,使源像素映射中这种颜色透明。下面程序在一个像素映射中扫描,如果当前像素的颜色与指定的颜色一致,就把它的 alpha 值设为 0,否则设为 1。

```
void RGBApixmap::setChromaKey(RGB c)
{
    long count=0;
    for(int row=0; row<nCols; row++)
        for(int col=0; col<nRows; col++)
        {
            RGBA p=pixel[count];
```

```
            if(p.r==c.r && p.g==c.g && p.b==c.b)
                pixel[count++].a=0;
            else pixel[count++].a=255;
        }
}
```

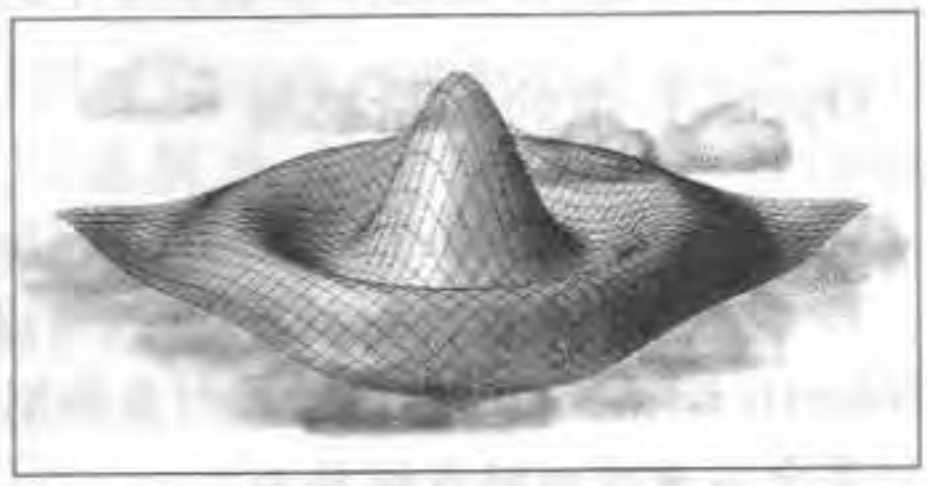


图 9.16　使像素映射的背景透明

图 9.16 显示了像素映射 S，它有单一背景颜色 c。应用 setChromaKey(c)方法可以使得该像素映射所有颜色为 c 的像素的 alpha 值为 0。

例 9.3.4　用画笔绘制

在画图程序中，鼠标起到了画笔的作用。用户可以为每条“笔划”设置颜色样本。这个功能可以通过 blend()函数来实现，将画笔的像素映射的 alpha 值设为 10%左右，并与目标图像融合。每画一笔，就有一些画笔的颜色融合到目标图像上(参见练习，了解如何估算每次下笔时新颜色增加的数量)。接近画笔中心像素的 alpha 可以比画笔边缘处像素的 alpha 设置得高一些，使得中心的颜色增加的快一些。

例 9.3.5　光标管理

移动鼠标时，鼠标的光标也在屏幕上掠过。光标移动到每一个位置，它都会遮住一部分显示的内容，而当鼠标移开，这部分内容也必须恢复回来。

实现这一任务的处理过程与显示下拉菜单类似，正如图 9.17 所示。在屏幕上绘制光标之前，先生成一个大致能够覆盖光标的矩形区域的图像副本(图中的像素映射 $Pix1$)。然后把光标的像素映射在此处与图像融合。在光标的像素映射中，1 表示光标箭头部分，0 表示其余部分。因此，当融合发生时，箭头看上去是浮在背景图像之上。当光标移动到一个新的位置时，有三件事情发生：

(1) 调用 draw()方法绘制 $Pix1$(alpha 值为 1)，覆盖到当前光标位置，也就是恢复原来的图像。

(2) 调用 read()方法将要被光标遮挡的部分($Pix2$)读入屏幕外的内存。

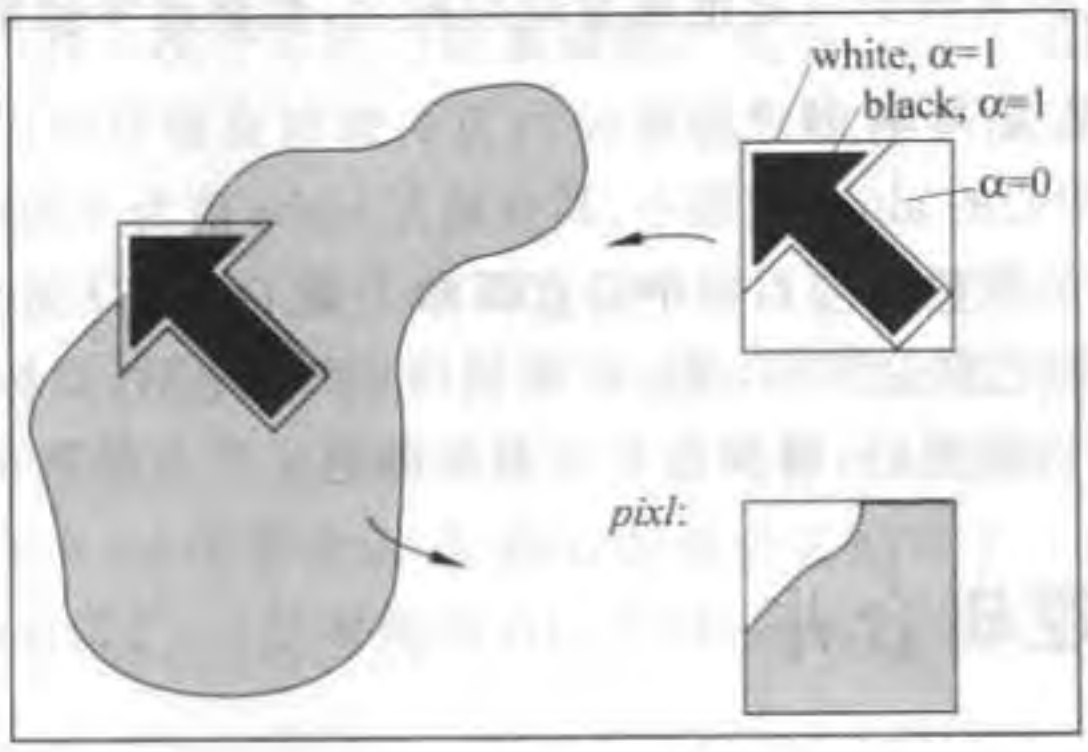


图 9.17　管理移动光标

(3) 在新的鼠标位置对光标和背景图像进行融合。

要注意的是,即使光标移动非常小的距离,也要进行这个过程。

OpenGL 中的可用扩展

OpenGL 提供了几个设置和使用 alpha 通道的工具。

1. **设置 alpha 值** 绘制场景时,可以显式的设置后面要绘制的图元的 alpha 值。使用 glColor4f(r, g, b, a)指定这些对象的第四个颜色分量,a 的取值范围为 0.0(完全透明)到 1.0(完全不透明),缺省值是 1.0。

我们也可以给三维物体设置"半透明材质",此时绘制这个物体的所有像素都具有一个特定的 alpha 值。这可以通过为物体每个定点的漫反射系数指定一个 alpha 值来实现:

```
glMaterialfv(GL_FRONT, GL_DIFFUSE, ref1) ;
```

其中 ref1 是一个四元组,它给出漫反射分量的颜色。例如 ref1[4]={0.5,0.3,0.8,0.4},将后面定义的顶点的 alpha 值设置为 0.4。

2. **融合设置方法** 我们已经介绍了使用 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)生成源像素映射和目标像素映射的加权平均。通过为 glBlendFunc()选取不同的参数,OpenGL 允许为源缩放系数和目标缩放系数设置不同的值,这些参数包括 GL_ZERO、GL_ONE、GL_DST_ALPHA、GL_SRC_COLOR 等。

例如,glBlendFunc(GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA)设置了源像素和目标像素的权值分别为 α_D 和 $1-\alpha_D$,其中 α_D 是目标像素的 alpha 值,而非源像素的。glBlendFunc(GL_DST_COLOR, GL_ZERO)设置了用目标像素的相应分量调整(或乘)源像素的每个颜色分量。

练习

练习 9.3.1 扩展 RGBpixmap 类使其包含 alpha 分量

调整 9.2.2 节的 RGBpixmap 类定义,使像素映射支持 alpha 通道。添加 blend()方法和 setChromaKey()方法,并通过以下途径进行测试:(a)将一个 BMP 图像读取到像素映射中(如果图像具有单一颜色背景,测试效果更好);(b)使用 setChromaKey()方法设置某一颜色为透明色;(c)将此像素映射与其他像素映射融合,观察透明效果。

练习 9.3.2 画笔应用新颜色

考虑一个靠近笔刷中心的 alpha 值很小、其余地方 alpha 值为 0 的画笔的像素映射。每一次使用这个画笔时,画笔的颜色 C 与目标中任意图像 D 融合,则 D 变为 $\alpha C+(1-\alpha)D$。重复使用目标图像上画笔的颜色就会加深。(a)证明使用 i 次画笔后,目标颜色为 $D_i=\alpha C+(1-\alpha)D_{i-1}$;(b)计算使用 8 次画笔后,源颜色 C 在目标颜色中所占的比例。

9.3.3 像素的逻辑合并

还有一些合并像素的方法是将每个像素值简单的看作比特位的集合来处理,而不关心比特位所表示的数值。同以前一样,像素映射还是逐个像素合并,但每个像素是逻辑地

(logically)合并，逐位运算，例如使用或(OR)、与(AND)、异或(EXCLUSIVE OR)等运算来实现像素映射的合并。

举例说明，像素 A 的 RGB 颜色分量为(21,127,0)，写成二进制形式是(00010101,01111111,00000000)。假定 B 的颜色分量的二进制形式是(01010101,11110000,10000101)。如果对它们进行或操作，结果 C 在下面给出：

A：(00010101,01111111,00000000)
B：(01010101,11110000,10000101)
C：(01010101,11111111,10000101)　←A 和 B 的或运算结果

如果 A 和 B 上下对齐的位中有一个(或者都)是 1，则 C 中相应的位也是 1。如果对 A 和 B 通过异或运算(EXCLUSIVE OR 或者 XOR)合并，就得到

A：(00010101,01111111,00000000)
B：(01010101,11110000,10000101)
C：(01010101,11111111,10000101)　←A 和 B 的或运算结果

这种情况下，如果 A 和 B 对应位上恰好有一位是 1，即只有一个而不是两个都是 1，那么 C 中相应位也是 1。颜色 $C=A$ XOR B 依赖于 A 和 B，但结果并不像与和或运算那样直观(参见练习)。如果 B 是黑色(0,0,0)，那 $C=A$ XOR B 结果是什么呢？如果 B 是白色(255,255,255)，结果又是什么呢？异或运算有一个有趣的性质：应用两次异或运算等于根本没进行任何运算。也就是说，如果 $C=A$ XOR B，然后将 C 与 B 再次进行异或，即(A XOR B)XOR B，这个结果还是 A！(参考练习)。后面在实现"橡皮筋"(rubber band)时会充分利用这个性质。

像 C++ 这样的语言可以直接支持按位进行的逻辑运算，可以在程序中直接完成这样的操作。例如，对 A 和 B 的绿色分量 OR 运算，可以简单的执行 $C \cdot g = A \cdot g | B \cdot g$。类似地，对于红色分量的 XOR 运算，则执行 $C \cdot r = A \cdot r \wedge B \cdot r$。

其他运算还包括"与"(AND)、"非"(NOT)，以及这些运算的组合。如果对两个量进行 AND 运算，用 C++ 语言表示绿色分量的运算为 $C \cdot g = A \cdot g \& B \cdot g$，只有当 A 和 B 对应位都为 1 时，$C \cdot g$ 的相应位才是 1。如果对某个值进行非(或补)运算，用 C++ 语言表示是 $C \cdot g = \sim A \cdot g$，当且仅当 A 中的位为 0 时，C 中的对应位才为 1。

我们可以使用 C++ 这种逻辑运算对两个像素映射进行逐个像素的合并。不过更幸运的是，OpenGL 提供了一种一次合并两个像素映射的机制。一旦选择并启用了一种逻辑运算，后面所有绘制都对当前存储在帧缓冲区内的源像素执行一次读-改-写的循环来应用这种逻辑运算。

执行命令 glEnable(GL_COLOR_LOGIC_OP)可以启用逻辑运算。使用 glLogicOp()选择当前的运算模式，可用的参数如图 9.18 所示，图中 S 表示源像素，D 表示目标像素。默认值是 GL_COPY，这个参数表示使用源像素简单地代替目标像素。

图 9.19 显示了两个 8×8 像素映射 A 和 B 逻辑合并的例子。这里像素值与颜色的关系是：0 表示黑色，1 表示白色。(如果使用 GL_AND 和 GL_EQUIV，得到的像素映射是什么样的呢？)

下面我们介绍 XOR 绘制模式的经典应用——"橡皮矩形"(rubber rectangle)，并介绍如何使用OpenGL实现这个应用。它很好的利用下面这个原理：在XOR模式下对一个物

参 数 值	结 果	参 数 值	结 果
GL_CLEAR	0	GL_OR	S 或 D
GL_COPY	S	GL_NAND	(S 和 D)的非
GL_NOOP	D	GL_NOR	(S 或 D)的非
GL_SET	1	GL_XOR	S XOR D
GL_COPY_INVERTED	非 S	GL_EQUIV	(S XOR D)的非
GL_INVERT	非 D	GL_AND_INVERTED	S 的非和 D
GL_AND_INVERT	S 或非 D	GL_OR_INVERTED	S 的非或 D
GL_AND	S 和 D		

图 9.18 OpenGL 中可用的逻辑运算

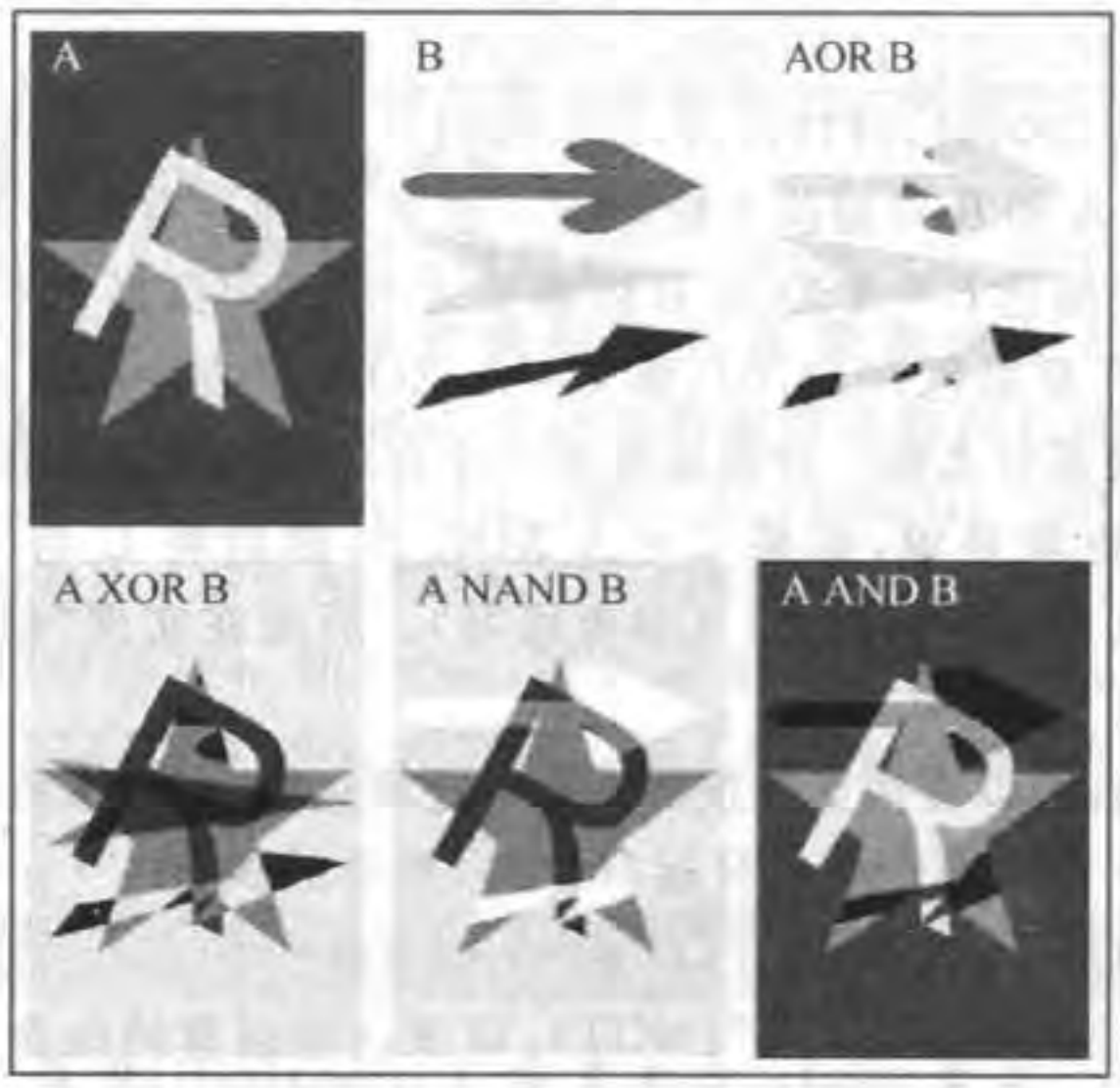


图 9.19 各种逻辑运算的效果

体连续绘制两次就会擦去这个物体,恢复成原来的图像。练习中还会介绍其他几个例子。

例 9.3.6 绘制"橡皮矩形"

当用户希望用鼠标在屏幕上绘制精确的图形时,"橡皮矩形"和"橡皮筋"能够提供很有价值的反馈。图 9.20 显示了几个例子,要绘制图 9.20(a)中的橡皮矩形,用户先把鼠标移动到要绘制的矩形的一个角点处(这个点称为中枢点(pivot point)),然后按下鼠标。当用户按住鼠标拖动时,就会看到一个弹性的矩形,它的一角始终固定在中枢点,对角位于鼠标指针当前位置。这个矩形变化时,它看起来就像掠过屏幕上绘制的图像而不改变它们。用户可以来回调整矩形,直到精确地覆盖住想要的区域。当鼠标键抬起以后,最终的矩形就在屏幕上保持可见了,它的坐标可供程序以后绘图使用。"橡皮筋"(图 9.20(b))是一条从中枢点延伸到鼠标位置的弹性直线,并随着鼠标移动拉长或缩短。实际上它仅仅是"橡皮矩形"的一个变种,通过绘制矩形的对角线就可生成。

橡皮矩形(或橡皮筋)需要不停地擦除和重绘,即使位置有一点点变化。而且不能用绘制背景色的方法擦除这个矩形,这会破坏线下面的图像。解决方案是利用 XOR 绘制模式。

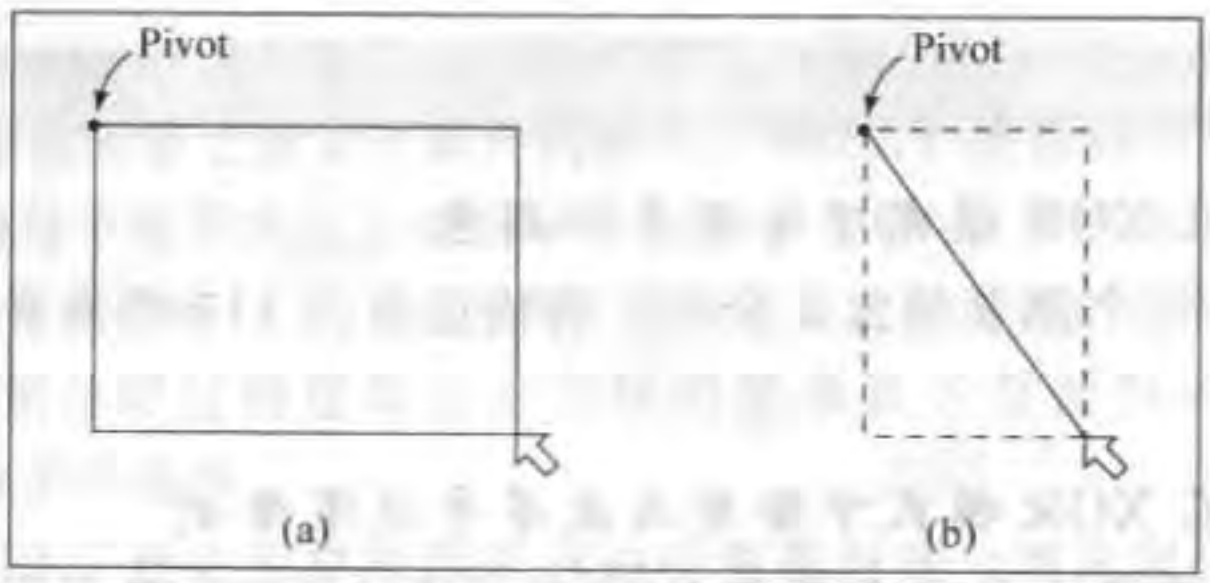


图 9.20　橡皮矩阵和橡皮筋

要擦去这个矩形，只须将其再绘制一次，同时原来的图像也会恢复。

图 9.21 给出了创建和管理橡皮矩形的鼠标回调程序。这些回调例程按照通常的方式在 main()中注册：调用 glutMouseFunc(myMouse)使 myMouse()成为鼠标按下或抬起事件的回调函数，调用 glutMouseMotionFunc(mouseMove)使 mouseMove()成为鼠标拖动 move 事件的回调函数。定义了一个全局的矩形对象 rr，并在两个回调函数中引用。初始矩形(大小为 0)在 myMouse()中建立。绘制矩形时不要使用黑色(全 0)作为矩形的绘制颜色！(为什么?)

值得注意的是，在 XOR 绘图模式下这个橡皮矩形的颜色并不是一致的。矩形上要绘制的每个像素的颜色是矩形的真实颜色和背景该处像素颜色进行 XOR 运算后的结果。这种效果通常并不令人讨厌，实际上是非常有用的，这使得矩形下的任何颜色的物体都能清楚的看到。

```
IntRect rr;                                    // 全局矩形变量

//<<<<<<<<<<<<<<<<myMouse>>>>>>>>>>>>>>>>
void myMouse(int button, int state, int mx, int my)
{
if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN)
    {
            glEnable(GL_COLOR_LOGIC_OP);       //启用逻辑运算
            glLogicOp(GL_XOR);                 //设置当前运算模式为 XOR
            rr.left=rr.right=mx; //set the pivot
            rr.top=rr.bott=screenHeight ?my;
    }
    if(button==GLUT_LEFT_BUTTON && state ==GLUT_UP)
        glDisable(GL_COLOR_LOGIC_OP);          //禁用逻辑运算
}
//<<<<<<<<<<<<<<<<mouseMove>>>>>>>>>>>>>>>>
void mouseMove(int mx, int my)
{
        rr.draw();                             //清除旧的矩形，仅在 XOR 模式有效
        rr.right=mx;                           //设置新的对角点
        rr.bott=screenHeight-my;               //翻转 y 坐标
        rr.draw();                             //绘制新的矩形
}
```

图 9.21　橡皮矩形的鼠标回调例程

练习

练习 9.3.3 在 XOR 模式下绘制多种颜色

假定帧缓冲区的每个像素包含 3 个位。请描述值为 110 的像素与各种可能像素值 XOR 的结果。

练习 9.3.4 在 XOR 模式下绘制两次等于没有绘制

对于任意颜色的像素 A 和 B,证明在 XOR 模式下绘制两次 B 得到的是没有改变的 A,也就是说(A XOR B) XOR B 的结果是 A。

练习 9.3.5 对称运算符

我们注意到,OR 运算符是对称的,即 A OR B 和 B OR A 是相等的。那么 16 个运算符$\oplus$中还有哪些在 $a\oplus b=b\oplus a$ 意义下是对称的?

练习 9.3.6 原地交换两幅图像

证明两个像素映射 A 和 B 可以通过执行 3 次 XOR 运算交换位置。当这个过程结束时,A 中的是原来 B 的像素值,B 中是 A 原来的像素值:

A=A XOR B;

B=A XOR B;

A=A XOR B。

练习 9.3.7 反转模式绘制

使用反转模式绘制图案也可以用来擦除一条线并恢复原来的像素值。一个 b 位的帧缓冲区有 $N=2^b$ 可能的像素颜色值,范围从 0 到 2^b-1。使用反转模式绘制像素的意义是将当前像素值 d 替换为 $f(d)=N-d$。需要注意的是这个函数是"循环"的:绘制两次就可以恢复原来的数值,因为 $f(f(d))=N-f(d)=N-(N-d)=d$。说明如何使用这种技术进行橡皮筋的绘制,并说明当橡皮筋跨越不同颜色的像素时,橡皮筋上的颜色是什么,是否和 XOR 模式下一样?对于一位($b=1$)的平面,反转模式与 XOR 模式有什么区别?

9.3.4 BitBLT 操作

可以把 draw()、read()和 copy()方法集成为单个函数,并在其中添加一些能够显著提高其能力和适用性的功能。这个操作被称为 BitBLT(发音为"bitblit"),意思是位块传输(bit boundary block transfer),还可以被称为 raster op【Inga78, Newell79】。当每个像素超过 1 位时,有时也把 BitBLT 称为 pixelBLT。这里我们把这两种情况都用 BitBLT 表示。

BitBLT 有时使用软件实现,但由于这种操作已经非常普遍,因此出现了专门设计的 VLSI 芯片可以高速执行 BitBLT(每秒处理几亿像素)。只需向 BitBLT 芯片中写入几个参数,该芯片就可以接管过来,并直接与系统总线交互,进行数据传输。

BitBLT 操作的定义

BitBLT 有几个不同的版本,它们在一些细节上有所差别。BitBLT 最简单的形式是把

源矩形(source rectangle)的像素复制到高度和宽度相同的目标矩形(destination rectangle)中。源和目标可以是在屏幕上或者屏幕外内存中。BitBLT 处理器以一种简单的方式描述这两个矩形：它存储每个矩形的左上角的 x 坐标和 y 坐标，以及矩形的高度和宽度(以像素为单位)。处理器来处理所有(x，y)对与内存对应地址之间所需的变换。

BitBLT 使用前面介绍过的逻辑合并方法把源像素 S 复制到目标矩形 D 中：$D=D\otimes S$。其中$\otimes$可以有多种选择。

有些版本的 BitBLT 允许用户对源矩形中的图像与某个预先定义好的像素映射进行“预混合”，这个预先定义好的像素映射被称为半色调图案(halftone pattern)。然后把这个预混合后的图像作为源 S 复制到目标。半色调图案保存在某个大小的像素映射中，比如 16×16 像素块，它可能是用 0 和 1 表示的棋盘图案。如果模板的大小小于源矩形，就把它复制拼接起来，直到填满矩形。对源像素映射 S 的选择有 4 种：

- 对源和半色调图案进行 AND 运算。
- 只使用源(忽略半色调图案)。
- 只使用半色调图案。
- 纯黑色。

多数 BitBLT 处理器还要维护一个裁剪(clipping)矩形。在绘制目标的每个像素之前，它们的坐标要先与裁剪矩形的边界比较，只有落在边界内部的像素才显示出来。裁剪过程可以在处理器内由硬件快速实现。裁剪矩形有效地限制了目标中实际绘制的范围。

9.4　自己实现直线绘制：Bresenham 算法

大多数图形环境都有绘制直线的内建工具。每种环境都至少有某种形式的 line()或者 lineto()函数。OpenGL 当然提供了更多的工具。因此我们通常不太关心这些工作的原理和细节——这些工具如何确定直线两个端点之间哪些像素要被绘制。

但是直线绘制是计算机图形学中最基本的算法，掌握它可以有助于了解上述程序的运行过程。实际上，程序员在开发或优化新的商业图形包时，仍然需要编写这些程序。

我们先从一种直观但是效率非常低的方法开始，以便提出我们的主要目的。然后再提出一种速度快得多的方法，这就是著名的 Bresenham 直线绘制算法。

假定我们要设置一些像素值，使得两个整数坐标(a_x，a_y)和(b_x，b_y)之间出现一条直线。图 9.22 显示出沿着理想直线(a_x，a_y)和(b_x，b_y)有某些像素被设置为“开”。如果这些像素被有效的选择出来，就可以在屏幕上显示出一条直线的样子(尽管不可避免的会有一些“锯齿”)。

根据一些基本代数知识，理想直线应满足下面的方程：

$$y = m(x - a_x) + a_y \tag{9.3}$$

其中 x 在 a_x 和 b_x 之间变化，m 是直线的斜率，有

$$m = \frac{b_y - a_y}{b_x - a_x} \tag{9.4}$$

例如，取(a_x，a_y)=(23，41)、(b_x，b_y)=(125，96)，则 m=55/102=0.5392。斜率只对非竖直的直线有意义，即 a_x 与 b_x 不同的时候。直线是水平或竖直时，可以很清楚确定哪些像

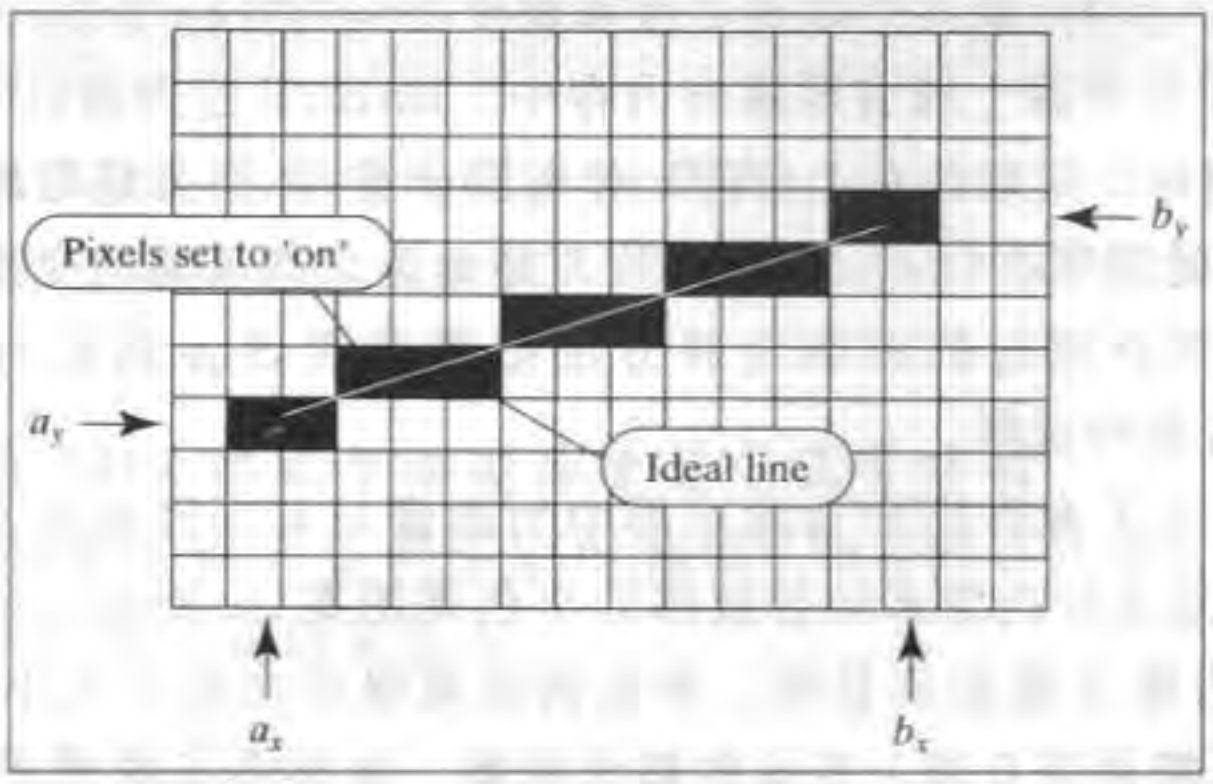


图 9.22 绘制一条直线段

素要被打开。对于其他类型的直线我们就需要一种算法来确定。

一种简单但效率低下的方法就是,沿着 x 方向从 a_x 增加到 b_x,递增的步长为一个像素。在每一步计算 $m(x-a_x)+a_y$,并取最接近此值的整数。在 a_x 小于 b_x 的情况下,可以使用以下代码实现:

```
float y=a.y                          //初始数值
for (int x=a.x; x<=b.x; x++, y+=m)
    setPixel(x, round(y));
```

其中 setPixel(x, y)和以前一样,把当前颜色写入帧缓冲区第 x 行第 y 列的像素中。本算法中需要对 y 值取最接近的整数值,这是一个非常费时的操作。

9.4.1 Bresenham 直线绘制算法

Bresenham 算法与前述方法相比有很大优势,它避免了浮点运算和舍入操作。它背后的思想也非常重要,其他类型的图元,如圆和椭圆的绘制算法也都应用了同样的思路。Bresenham 算法是增量算法的经典例子,它利用前一个像素的信息来计算当前像素的位置[①]。该算法只使用整数运算并且不用任何乘法运算,生成像素的内层循环紧密而且高效。

Bresenham 算法有几种变体,它们的计算方法略有不同。我们介绍的版本被称为中点算法(midpoint algorithm)。对于直线,该算法与 Bresenham 算法产生同样的像素,而且还可以扩展到更复杂的图形绘制算法,如圆和椭圆。

和以前一样,假定给定直线的两个端点(a_x,a_y)和(b_x,b_y),要确定直线上两端点之间像素的最优序列。

为了简化讨论,我们首先考察 $a_x<b_x$(也就是说 b 点在 a 点右侧),同时直线斜率在 0 和 1 之间的情况(稍后我们将取消这个限制)。为方便起见,我们定义线段在 x 方向和 y 方向上的长度,即 W 代表"宽度",H 代表"高度":

$$W = b_x - a_x$$

① 这种算法有时也称为 DDA 算法,由一种使用增量方式求解微分方程的机械设备而得名。

$$H = b_y - a_y \tag{9.5}$$

根据我们前面的假设可以得到，W 和 H 都为正，且 $H<W$。因此，当 x 从 a_x 增加到 b_x，对应的 y 值也从 a_y 增加到 b_y，但 y 增加的速度慢。当 x 的步长为一个单位时，y 值的最优结果一定是有时保持不变，有时增加 1。中点算法能够快速地确定到底会发生哪一种情况。（y 永远不会减少或者增加幅度大于 1，为什么？）

根据第 4 章的介绍，我们知道通过(a_x, a_y)和(b_x, b_y)的理想直线方程为：

$$-W(y-a_y)+H(x-a_x)=0$$

对于所有落在理想直线上的点(x, y)，代入方程的左侧始终为 0。

我们为这个表达式起个名字以备后用。事实上先把这个表达式乘以 2 再命名更为重要（稍后会看到，这样做是为了避免在关键公式中出现一个麻烦的系数 1/2）。现在我们定义下面函数：

$$F(x,y) = -2W(y-a_y)+2H(x-a_x) \tag{9.6}$$

$F(x, y)$的重要性质就是，它的符号表明了(x, y)是在理想直线的上方还是下方：

- 如果 $F(x,y)<0$，则(x, y)在直线上方。
- 如果 $F(x,y)>0$，则(x, y)在直线下方。

注　推理提示：假设(x, y)在直线上，那么我们知道 $F(x,y)=0$。把 y 向上稍微调整，并且保持 x 不变，这使得函数 F 的数值变小。因此把 y 增加到直线上方会使 F 变为负值。

练习

练习 9.4.1　Bresenham 算法实验

对于两个端点(3, 7)和(9, 11)之间的直线段，有

$$F(x,y) = (-12)(y-7)+(8)(x-3)$$

直线上的点，例如(7, 29/3)满足 $F(x,y)=0$。而点 $A=(4, 4)$和 $B=(5, 9)$处函数 $F(x,y)$的符号是什么，它们与直线的相对位置关系是什么样的？

答案：A 处在直线下方，F 值为 44。B 处在直线上方，F 值为-8。

现在如何确定哪些像素在直线上呢？图 9.23 给出了一些接近理想直线的像素。位于格子交叉点的圆圈表示像素的中心。假设我们已知对应 p_x 的最优 y 值是 p_y，现在要确定下一个 x 即 p_x+1 对应的最优 y 值。我们希望知道直线在 p_x+1 处是距离点 $L=(p_x+1, p_y)$更近，还是距离点 $U=(p_x+1, p_y+1)$更近。图中只是给出了理想直线上一个可能位置，它可能低于 L 或者高出 U 么？（为什么？）。

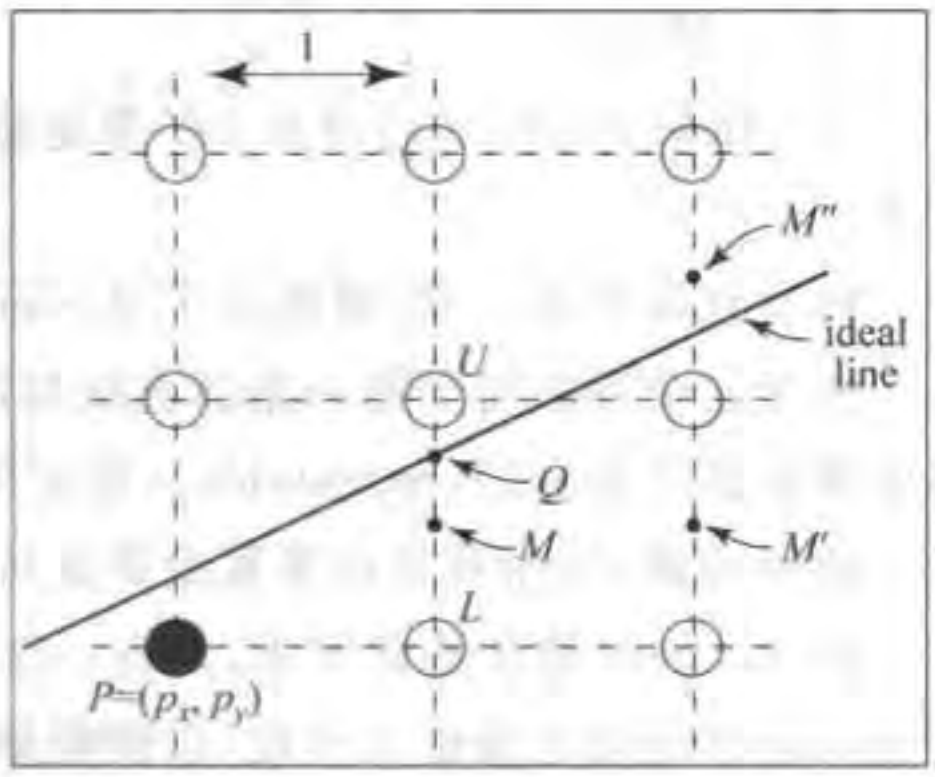


图 9.23　中点算法推导图示

我们可以通过判断直线在 U 和 L 的中点 M 的上方还是下方来确定选择像素 U 还是像素 L。$M=(p_x+1, p_y+1/2)$。通过计算 M 点处的函数 F 的符号，我们可以确定理想直线与 M 的相对关系：

- 如果 $F(M_x,M_y)<0$,则 M 在理想直线上方,所以选择 L。
- 如果 $F(M_x,M_y)>0$,则 M 在理想直线下方,所以选择 U。

因此,如果 $F(M_x,M_y)>0$,要选择的像素比前一像素高一个点,我们增加 y 值,否则就不增加 y 值。剩下的就是代数问题了:如何快速计算 $F(,)$,关键是根据这个函数每一步的增量来递增地计算函数值。

我们计算出 M 点的函数值 $F(M_x,M_y)$以备后用:

$$F(M_x,M_y)=-2W(p_y+1/2-a_y)+2H(p_x+1-a_x) \tag{9.7}$$

当 x 从 p_x+1 移动到下一个 x 值 p_x+2 时,考虑函数 $F(\ )$的变化。如图 9.23 所示,M 可能是 M',也可能是 M''。如果在前一步 y 没有增加,那么结果是 $M'=(p_x+2,p_y+1/2)$;如果前一步 y 增加了 1,那么结果就是 $M''=(p_x+2,p_y+3/2)$。

情况 1:如果前一步计算 F 是负值(因此这一步中 y 值不增加),则

$$F(p_x+2,p_y+1/2)=-2W(p_y+1/2-a_y)+2H(p_x+2-a_x)$$

减去(9.7)式,得到与 $F(M_x,M_y)$的差异:

$$F(p_x+2,p_y+1/2)=F(M_x,M_y)+2H \tag{9.8}$$

情况 2:如果前一步计算 F 是正值(此步 y 增加 1),则

$$\begin{aligned}F(p_x+2,p_y+3/2)&=-2W(p_y+3/2-a_y)+2H(p_x+2-a_x)\\&=F(M_x,M_y)-2(W-H)\end{aligned} \tag{9.9}$$

不管在哪种情况下,测试量的增量都是常数的:如果不增加 y,增量为 $2H$;如果增加 y,则增量为 $-2(W-H)$。

现在剩下的唯一问题是这个过程如何开始。当 $x=a_x$ 时,我们知道 $y=a_y$。因此 M 的第一个值是 $M=(a_x+1,a_y+1/2)$,而

$$\begin{aligned}F(x,y)&=-2W(a_y+1/2-a_y)+2H(a_x+1-a_x)\\&=2H-W\end{aligned} \tag{9.10}$$

注:如果前面没有把函数加倍,这个结果就会是 $H-0.5W$,这样就无法做到对所有量都使用整数值。

总结 把 F 初始化为 $2H-W$,把 x 初始化为 a_x,y 初始化为 a_y。然后对于每一步:

(1) 将(x,y)处的像素设置为需要的颜色。

(2) x 增加 1。

(3) 如果 $F<0$,通过增加 $2H$ 来更新 F;否则将 y 增加 1,并通过增加 $-2(W-H)$来更新 F。

将所有综合在一起,就得到了 Bresenham 算法(在前面所述的约束条件下)。

由于这个算法对于每一条新直线都需要重新初始化,并且只依赖于直线的端点数据,因此该算法是可重复的(repeatable),即使用不同的颜色重新绘制直线每个像素都可以完全覆盖。换句话说,使用背景色重复绘制就可以完全擦去原来绘制的直线。

Bresenham 算法非常简单,只有一个内部循环,在循环内也只有几次加法和几次比较。它可以很容易用汇编语言实现,以获得最大速度。通常也有一些特殊用途的图形硬件实现此算法以获得更高的运行速度。

```
bresenham(IntPoint a, IntPoint b)
{//约束条件: a.x<b.x, 同时 0<H/W<1
    int y=a.y, W=b.x-a.x, H= b.y-a.y;
    int F=2*H-W;                    //当前误差项
    for(int x=a.x; x<=b.x; x++)     //内层循环
    {
        setPixel(x, y);
        if(F<0)
            F+=2*H;                 //为下一个像素更新函数值
        else{
            y++;
            F+=2*(H-W);
        }
    }
}
```

图 9.24　中点算法(带约束条件下)

例 9.4.1　示例

观察一个具体的例子中 F 的变化,非常有助于理解算法。令 $(a_x,a_y)=(4,1)$,$(b_x,b_y)=(16,4)$,则有 $W=12$,$H=3$。因为这条直线的斜率是 1/4,所以期望的结果应该是每个四步左右 y 值递增一次。F 初始化为 -6。每次 x 增加,都要做下面两件事情中的一件:如果 F 是负数,就把它加 6;否则把它减 18,同时 y 递增 1。

得到的结果序列(可以执行 setPixel(x, y)来观察)如下:

x:	4	5	6	7	8	9	10	11	12	13	14	15	16
y:	1	1	2	2	2	2	3	3	3	3	4	4	4
F:	−6	0	−18	−12	−6	0	−18	−12	−6	0	−18	−12	−6

图 9.25 显示了中点算法应用于此例的情况,图中给出了结果直线和 F 的变化。从图中可以很清楚看到直线的"锯齿"出现在每个短的水平线段跳到下一个高一点的水平线段。

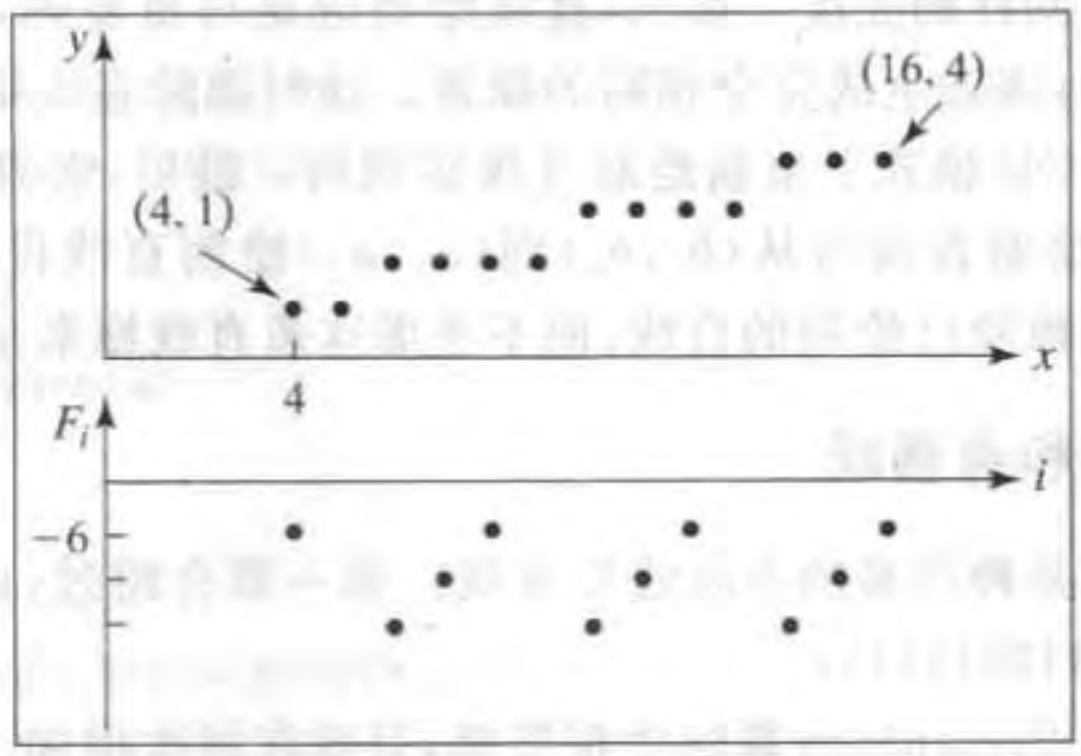


图 9.25　中点算法应用示例

去掉对 Bresenham 算法的限制

前面讲述的算法只能处理 $a_x<b_x$,且斜率在 0 和 1 之间的情况。通过添加处理各种情况的代码,其余的情况也不难处理。

当 $a_x>b_x$ 时得到同样直线 中点算法在每一步都会在两种情况中选择一个,把像素设置在中点的正确的一侧。因此我们只须调整图 9.24 的算法,实现在 $a_x>b_x$ 情况下的中点算法,这不需要很大的改动(参见练习)。需要注意的一个细节是:图中的代码分为 $F<0$ 和 $F\geqslant 0$ 两种情况,其中 $F=0$ 的情况与 $F>0$ 合并在了一起(此时 y 增加)。所以我们必须确保相反方向跟踪直线时,要把 $F=0$ 包含在 y 增加的情况中。

另一种方法可以通过检查是否有 $a_x>b_x$,如果是就交换 a_x 和 b_x,并交换 a_y 和 b_y,然后使用图 9.24 的算法。这种方法永远不会绘制"向左"的直线,而是重新定义直线的端点,使它不必绘制这样的直线。然而如果要用这种算法绘制折线段,这种技术就并不十分成功。因为交换顶点会损失折线中端点的自然顺序,而在把折线绘制成点线或虚线时,这个顺序是非常重要的。

斜率大于 1 的直线 这种情况下,只需交换 x 和 y 的角色,将 y 从 a_y 到 b_y 递增,并且使用同样的方法测试并决定何时增加 x(此时 x 的变化比较慢)。

斜率为负数的直线 如果斜率在 0 和 -1 之间,H 自动具有正确的符号。在 x 上递增并使用同样的测试,但另一个变量不再是递增,而是递减。如果斜率小于 -1,则用 $-W$ 代替 W,交换 x 和 y 的角色,就像处理斜率大于 1 的情况一样。

水平线和竖直线 这种情况在图形中经常出现,因此必须提高对这些直线的检测效率(例如使用代码 if(a. x= = b. x)…)和使用简化的绘制算法。对于竖直线,改进是有限的,因为大部分时间用在 setPixel()上。对于水平线,可以使用一个函数沿着扫描线一次写入多个位,从而显著提高其绘制速度。

参见练习了解如何实现这些一般化过程。

绘制出的直线应具有的性质

我们需要直线绘制算法应具有哪些性质呢?首先,直线应该尽可能的直,并且确实穿过给定的两个端点。第二,这条直线应该尽量"平滑"并且在直线各个部分具有一致的亮度,不同斜率的直线应该具有同样的亮度。第三,直线绘制应是可重复的,也就是说,如果使用同样的端点绘制直线两次,应该生成完全相同的像素。这对擦除直线很重要,因为擦除就是通过使用背景颜色或在 XOR 模式下重新绘制直线实现的。最后,它应该与绘制的方向无关,即从(a_x,a_y)到(b_x,b_y)绘制直线与从(b_x,b_y)到(a_x,a_y)绘制直线得到的结果应完全一样。这样应用程序可以完全擦除已绘制的直线,而不考虑这条直线原来是怎么绘制的。

用图案绘制直线和点画线

我们有时需要绘制某种图案的点画线和虚线。第一章介绍过,虚线的图案以位序列的形式存储,比如 0011111100111111。

把这种图案集成到 Bresenham 算法中很简单,只要在每次增加 x 的同时递增一个指向图案的指针,然后使用指向相应位的比特值设置 setPixel()所使用的当前绘制颜色。对于长直线,从图案末尾折回起始处循环递增地重复使用图案。

当使用某种图案的虚线绘制折线时，虚线的图案需要在线段与线段间保持连续。为此，需要把这个图案和指向图案的指针设置为 Bresenham 算法的全局变量，这样在算法连续调用时能够不断访问和更新这些变量，在算法结束时保证其数据不丢失。参见练习，了解关于图案绘制的其他问题。

OpenGL 使得使用点画线绘制直线和多边形变得简单。glEnable(GL_LINE_STIPPLE)命令为后面绘制的所有直线启用了点画线绘制模式。点画线模式可以通过 glDisable(GL_LINE_STIPPLE)命令关闭。一旦点画线模式启用，后面绘制的直线就按照 glLineStipple(factor, pattern)命令定义的图案绘制方式绘制。参数 factor 是 int 类型，它决定了每个图案重复的次数。例如，factor=2 表示图案的每个比特位使用 2 次以后才切换到下一个比特位。factor 的范围是 1～256。图 9.26 显示了三条直线，factor 分别为 1、2 和 3。

图 9.26　使用不同 factor 的三条点画直线

参数 pattern 是一个 gluByte 类型的数组，使用 0 和 1 的串(1 表示像素开，0 表示像素关)表示图案。为了简化，程序员通常使用十六进制方式表示图案的元素，每个十六进制数值可转换为 4 个比特位。例如一个图案的十六进制表示是 0x0000，等同于二进制的 0000000000000000，即 16 个像素均为黑色。图案 0x0101(二进制为 0000000100000001)可以生成一个点画线。图案 0x0F0F(二进制为 0000111100001111)可以生成一个较宽的虚线。图 9.27 给出了“OpenGL Red Book”【Woo04】中一个完整的程序。

```
#include<GL/glut.h>
#include<stdlib.h>

#define drawOneline(x1,y1,x2,y2) glBegin(GL_LINES); \
    glVertex2f((x1),(y1)); glVertex2f((x2),(y2)); glEnd();

void init(void)
{
    glClearColor(1.0,1.0,1.0.0,0);                //使用白色不透明背景
    glShadeModel(GL_FLAT);
}

void display(void)
{
    int i;

    glClear(GL_COLOR_BUFFER_BIT);

/* 选择黑色绘制直线 */
```

图 9.27　一个完整的演示点画线绘制的程序【Woo04】

```
    glColor3f(0.0,0.0,0.0);

/* in first row, 3 lines, each with a different stipple  */
    glEnable(GL_LINE_STIPPLE);

    glLineStipple(1,0x0f0f);            /*  点画线  */
    drawOneline(50.0, 125.0, 150.0, 125.0);
    glLineStipple(3, 0x0f0f);           /* 虚线 */
    drawOneline(250.0, 125.0, 350.0, 125.0);

    glDisable(GL_LINE_STIPPLE);
    glFlush();
}

void reshape(int w, int h)
{
    glViewport(0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, (GLdouble) w, (GLdouble) h);
}

void keyboard(unsigned char key, int x, int y)
{
    //ESC 键退出
    swith(key){
        case 27:
            exit(0);
            break;
    }
}

int main(int argc, char**argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GULT_SINGLE | GLUT_RGB);
    glutInitWindowSize(400,150);
    glutInitWindowPosition(100,100);
    glutCreateWindow(argv[0]);
    init();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutKeyboardFunc(keyboard);
    glutMainLoop();
    return 0;
}
```

图 9.27 (续)

多边形点画显示

OpenGL 也为多边形点画显示提供了相应机制。多边形的点画显示与直线类似。首先也要使用 glEnable(GL_POLYGON_STIPPLE)来启用多边形点画模式；然后使用 glPolygonStipple(bug)语句定义图案，该图案由一个十六进制数值的数组构成。bug 是我们定义的图案名称，如图 9.28(b)所示。9.28(c)显示的是一个半色调图案，程序员经常用它来达到一种平滑的灰色效果。

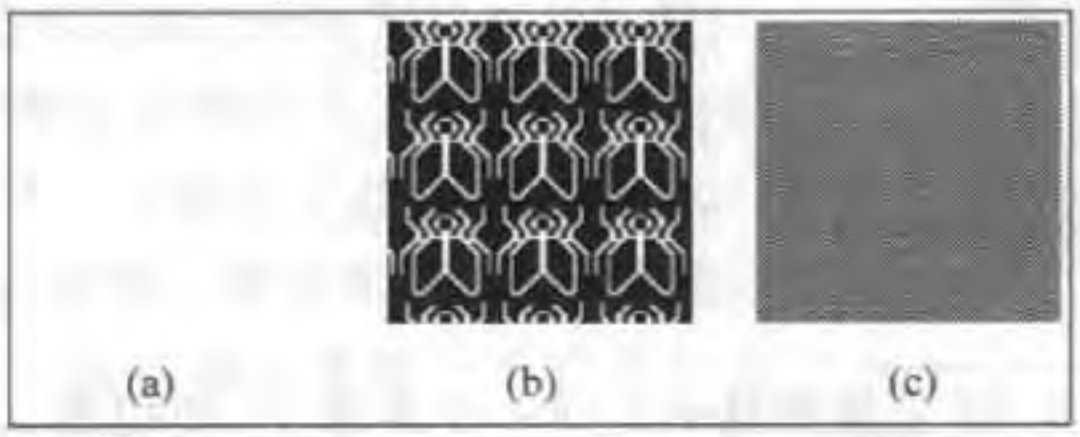


图 9.28　三个矩形

(a) 无图案；(b) "bug"图案；(c) 半色调图案

bug 和半色调图案由下面的数组定义：

```
Glubyte bug[]={
    0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00, 0X00,
    0X03, 0X80, 0X01, 0XC0, 0X06, 0XC0, 0X03, 0X60,
    0X04, 0X60, 0X06, 0X20, 0X04, 0X0C, 0X30, 0X20,
    0X04, 0X18, 0X18, 0X20, 0X04, 0X0C, 0X30, 0X20,
    0X04, 0X06, 0X60, 0X20, 0X44, 0X03, 0XC0, 0X22,
    0X44, 0X01, 0X80, 0X22, 0X44, 0X01, 0X80, 0X22
    0X44, 0X01, 0X80, 0X22, 0X44, 0X01, 0X80, 0X22
    0X44, 0X01, 0X80, 0X22, 0X44, 0X01, 0X80, 0X22
    0X66, 0X01, 0X80, 0X66, 0X33, 0X01, 0X80, 0XCC
    0X19, 0X81, 0X81, 0X98, 0X0C, 0XC1, 0X83, 0X30
    0X07, 0XE1, 0X87, 0XE0, 0X03, 0X3F, 0XFC, 0XC0
    0X03, 0X31, 0X8C, 0XC0, 0X03, 0X33, 0XCC, 0XC0
    0X06, 0X64, 0X26, 0X60, 0X0C, 0XCC, 0X33, 0X30
    0X18, 0XCC, 0X33, 0X18, 0X10, 0XC4, 0X23, 0X08
    0X10, 0X63, 0XC6, 0X08, 0X10, 0X30, 0X0C, 0X08
    0X10, 0X18, 0X18, 0X08, 0X10, 0X00, 0X00, 0X08
};
Glubyte halftone[]={
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
```

```
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
    0XAA, 0XAA, 0XAA, 0XAA, 0X55, 0X55, 0X55, 0X55,
};
```

如果程序员想把图 9.28 三幅图显示在同一个屏幕上，使用图 9.29 给出的代码片段可以完成。9.28(a)使用熟悉的函数 glRectf(...)进行单一颜色绘制，然后启用点画模式，并使用 gluPolygonStipple(bug)设置 bug 为当前图案；然后使用另一个 glRectf 调用绘制。最后把当前图案切换到半色调图案，并再次绘制矩形。

```
glRectf (25.0, 25.0, 125.0, 125.0);          //白色矩形
glEnable(GL_POLYGON_STIPPLE);
glPolygonStipple(bug);
glRectf (125.0, 25.0, 225.0, 125.0);         //苍蝇图案
glPolygonStipple(halftone);
glRectf (225.0, 25.0, 325.0, 125.0);         //半色调图案
glDisable(GL_POLYGON_STIPPLE);
```

图 9.29 一个多边形点画显示的代码片段

练习

练习 9.4.2 去掉 Bresenham 算法的约束

把 Bresenham 算法的变体应用于以下情况：(a)$a_x > b_x$；(b)直线斜率大于 1；(c)直线斜率为负。把各种情况的处理方法合并成一个单独的算法，使其可以处理任意线段，尽量提高程序的效率。

练习 9.4.3 数值例子

对于端点为(8,23)和(21,11)的直线，应用 Bresenham 算法，给出 x、y、F 值序列。证明算法从另一个点开始时得到的结果相同。

练习 9.4.4 按图案绘制

扩展 Bresenham 算法，使其能够绘制 16 位图案的直线。进行进一步的调整，使得绘制折线时该图案从一条线段接到另一条线段时保持连续不中断。

9.5 定义和填充像素区域

光栅显示器能够显示纯色或图案填充(fill)的像素区域。在这里区域(region)一词的意思是以某种方式彼此“相邻”或由某种共同性质联系起来的一组像素。图 9.30 显示了一些使用不同颜色和图案填充的区域。

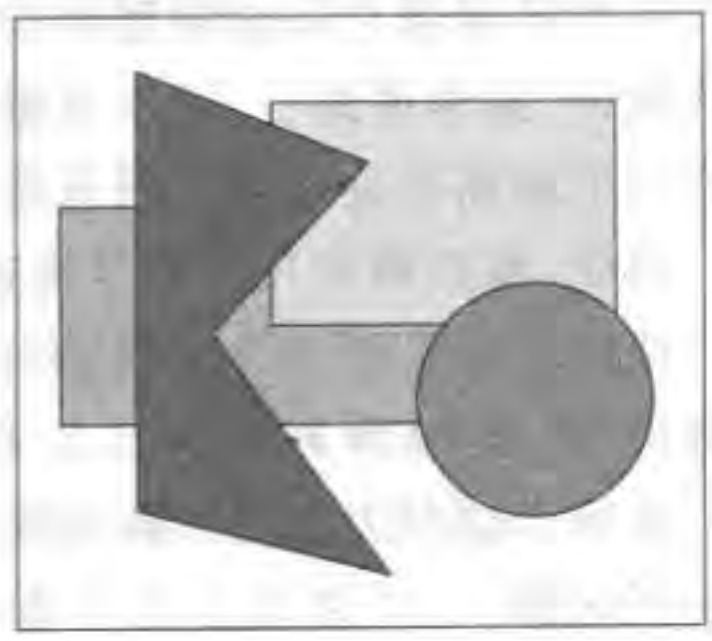


图 9.30　一些使用图案绘制的区域

现在许多绘图程序都允许用户交互的生成一幅图像。所有这些程序都包含一个填充工具，这个工具允许用户单击一个区域，然后选择一种图案，这个区域随之马上被新的图案填充。下面让我们来看一下它是如何实现的。

9.5.1 区域定义

有多种定义区域的方法。这些方法的主要区别是：“像素定义”还是“符号表示”。

- “像素定义的”(pixel-defined)区域是用像素映射中的实际像素描述的。对一个区域 R 的描述可能要列出 R 内部的所有像素：(34,12)、(34,13)、(34,14)等。或者可以定义 R 为由所有像素值为 77，且按某种方式与像素(43,129)连通的像素组成。连通的概念需要被仔细定义。想知道区域 R 的样子，需要仔细查看像素映射，观察在这个定义下，哪些像素属于 R。不管怎么说，列举区域的所有像素都是令人厌烦的，并且容易出错。
- “符号表示”这种描述方式并不列举像素，而是提供区域 R 中所有像素具备的某种性质。这样的描述通常比纯粹的枚举像素更高级，或者说更抽象。下面是一些符号描述区域的方法：
  - 所有到给定点(23, 47)的距离比到点(12, 14)、(22, 56)、(35, 45)中任何一个距离都小的像素。
  - 以(5, 23)为圆心，8 为半径的圆形内部的所有像素。
  - 所有位于以(32, 56)、(120, 456)、(345, 1129)、(80, 87)为顶点的多边形内部的点。这种“多边形定义”(polygon-defined)区域是一种非常重要的形式，我们将在后面详细讨论。

根据要研究的区域描述方法不同，程序员可以选择完全不同的数据结构和算法对区域进行处理和分析。

我们不会对像素定义的区域进行过多的关注，而是着重讨论符号定义区域，特别是多边形定义区域的各种处理方法。

9.6 处理符号定义的区域

一些区域填充算法要逐个像素地读取帧缓冲区中的像素,以确定每个像素的颜色,从而在区域中"探索道路"。可以设想,如果填充算法处理的是一个用更高级的更符号化的描述定义的区域而不是靠像素列举描述的区域,那么这个填充算法的效率会高很多。

因此,我们研究几种符号描述区域的方法,这几种方法各有优缺点。在图形学中使用最为广泛的方法是把一个区域定义为一个多边形的内部,这也是 OpenGL 所采用的方法。我们将详细介绍这种方法。

这种方法分为两类:第一类是用矩形的集合表示多边形,第二类用定义边界的路径确定一个区域。

9.6.1 矩形定义的区域

这种方法把一个区域描述为一组矩形。这些矩形可以小到由单个像素构成,大到整个像素映射。这个区域可以有空洞甚至有不连通的小区域。

图 9.31 中显示了一个简单的例子,其中每个像素的值是黑色或白色。我们把这个区域分解为一组对齐的矩形,这种表示正好是找到的所有矩形的列表(用眼睛快速扫视这个图形可以发现约有 11 个矩形满足这个要求,但是有多种分解方式)。

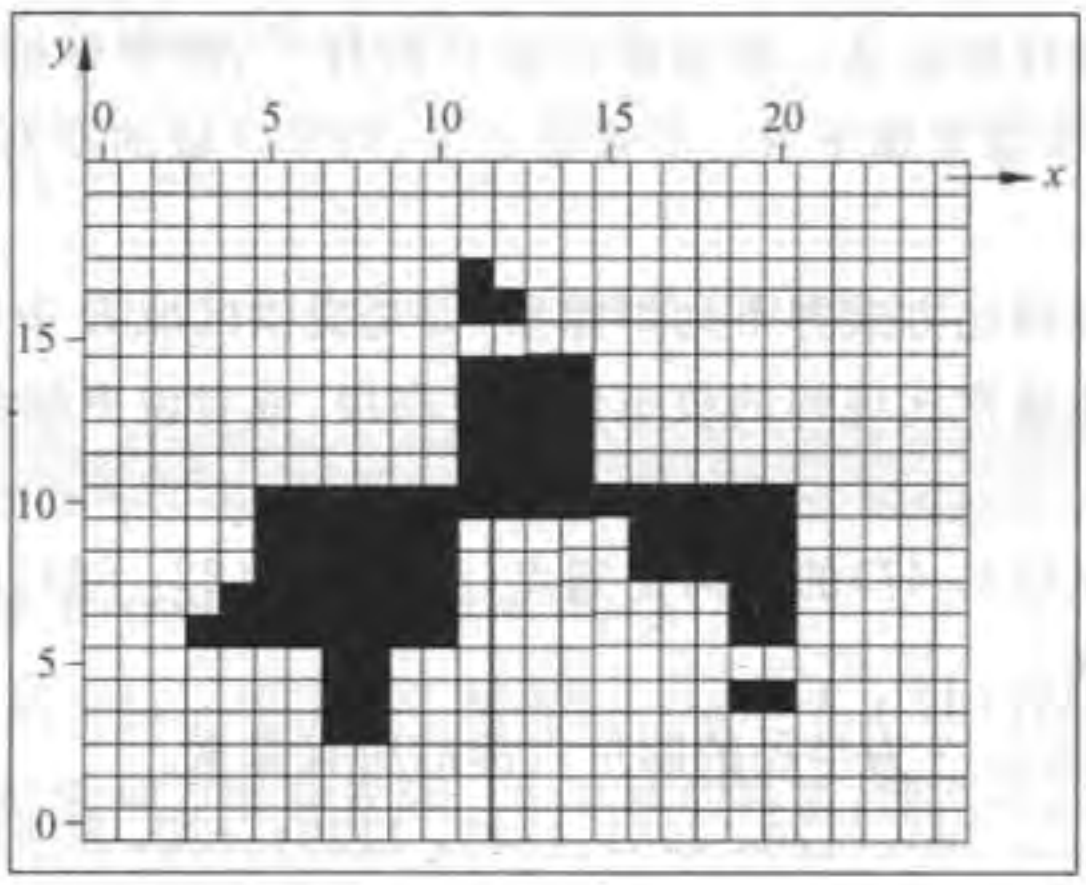


图 9.31 用矩形表示的区域

我们介绍一种规范的将像素映射分解为矩形列表的方法。该方法简单有效,它可以快速的从像素映射构建形状(shape),也可以快速填充这个形状描述的区域。

该方法从像素映射的最上端开始,一行一行地向下扫描来识别矩形。只要某个扫描线中的像素串集合与上一行扫描线中的像素串集合不同,就会出现一组新的矩形。其中,像素串(run)表示同一扫描线上相同颜色的相邻的像素集合。

像素映射的这种表示方法有时也称为它的形状(shape)表示【Steinhart92】。这样的表

示方法可以描述像素映射中的任意像素集合，包括不连通区域组成的集合。

当区域具有横向跨度一致性和扫描线一致性时，用这种方法表示最紧凑。横向跨度一致性(span coherence)是一条扫描线内的属性，指的是沿着一条扫描线多数像素具有相同数值。扫描线一致性(scanline coherence)是扫描线之间的属性，指的是一条扫描线的图案与相邻扫描线的图案类似。如果区域具有高度的一致性，那么用形状表示时，平均每个像素需要的数据更少。

缩放和平移区域

当我们想对区域进行一些处理时，"形状"表示法也很有用，例如：

- **区域平移**　假设把一个用形状表示法表示的区域在 x 方向平移 30 个像素，y 方向平移 55 个像素，只需遍历它的列表，把所有矩形的 x 值增加 30，y 值增加 55。
- **区域扩大一倍**　首先平移这个区域，使得第一个像素串位于原点(0，0)上，然后把数据结构中的所有值扩大一倍。最后再把这个区域平移回所需的位置上。

9.6.2　路径定义的区域

很自然会想到使用区域的边界来定义这个区域，这个边界通常是按照某种顺序排列的路径。描述路径的方法有很多，下面是其中一些：

- 通过数学公式。例如 $(x-122)^2+(y-36)^2=25$ 确定了一个直径为 10 像素的圆形路径。
- 通过折线。通过像素位置序列 $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ 可以确定一条折线路径。如果这条路径是封闭的，即 $(x_1, y_1)=(x_n, y_n)$，就确定了一个多边形。
- 通过相邻像素序列。链码(chain code)是一种特别值得注意的表示邻接像素路径的方法。先为路径指定一个起始像素，比如(34，67)。然后记录从像素到像素移动的序列，比如"向上、向右、向下"等等。为可能的移动方向集合编码的方式有很多。

图 9.32 中显示了两种编码方式。第一种方法(图 9.32(a))记录 4 个"罗盘"方向的移动，每个方向与 0 到 3 中的一个数字关联。于是这样的移动序列定义了一个连续的路径。图中的路径 A 由它下面的数字序列表示，该序列从右下角开始记录。这条路径恰好是封闭的，但是一般说来，路径并不一定是封闭的(序列中的 0、1、2、3 的个数满足什么条件时，路径是封闭的?)。值得注意的是，由于使用了 4 个可能的方向，所以每个方向可以只用两个比特位表示。

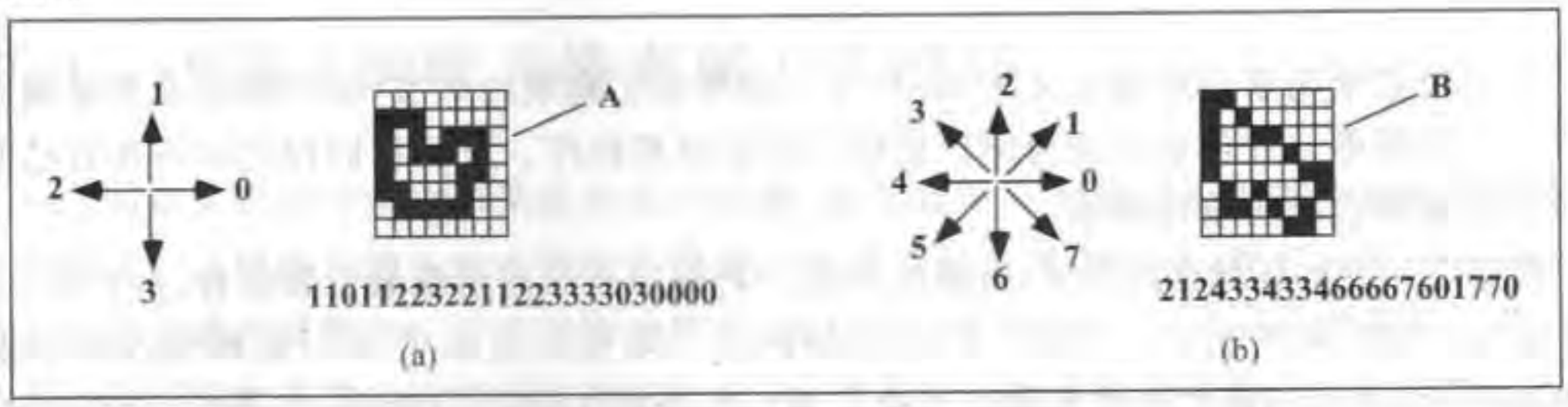


图 9.32　通过链码定义的路径

图 9.32(b)中给出的方法允许有 8 种可能的方向。其中,"1"表示路径中的下一个像素在上面一行偏右一个像素。另外 3 个对角线方向的表示方法与此类似。(图中所示序列的起始点在哪里?)链码与第 3 章中的相对绘图很类似,我们会在附录 5 中进一步讲述。

图 9.33 显示了一种用来存储路径的数据结构。最开始的两个元素给出路径的起始点坐标,接下来是要移动的步数,然后就是移动序列。

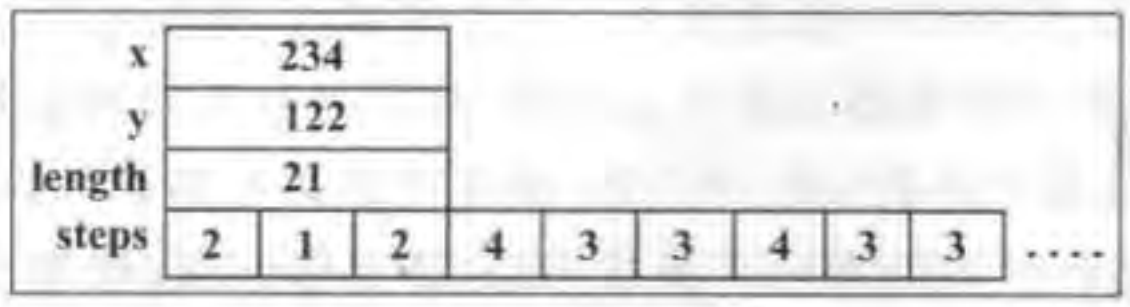


图 9.33 适用于链码编码路径的数据结构

链码能够为某些路径和区域提供非常简洁的表示。(对于 8-方向链码,每个方向需要用几个比特位表示?)此外,还可以很方便地执行某些处理过程:

(1) 平移路径:非常简单,只需改变起始点坐标即可。

(2) 以系数 k 放大路径:将每个符号重复 k 次。

(3) 把路径逆时针旋转 90°:对于 4-方向链码,只需把每步的数值增加 1,如果是 3 就增加为 0。8-方向链码的处理也是同样道理。

而另一方面,有些操作处理起来会变得困难。例如填充一个用链码表示的区域就很复杂,这是因为从链码确定扫描线上的像素串并不容易。所以最好是先把链码表示变换为"形状"表示。

练习

练习 9.6.1 链码路径

编写代码,使用 3 比特移动编码表示一个对齐的正方形,并选择合适的起点使得链码总长最小。

练习 9.6.2 旋转的正方形的链码

重复练习 9.6.1,表示一个旋转 45°的正方形。如果不能精确的表达,就寻找最佳的近似。

9.7 填充多边形定义的区域

人们经常使用多边形来定义区域,并且已经开发了高效的算法,使用纯色或图案填充多边形。有些图形软件包也已经为用户提供了高效填充程序。下面我们讨论如何在自己的软件包中实现多边形的填充操作。

假定要填充的区域多边形 P,由描述为指定 P 的顶点序列的像素位置集合,$p_i=(x_i,y_i)$,$i=1\cdots N$。图9.34 给出了一个有 7 个顶点的例子。填充多边形 P 可以在帧缓冲区中逐条扫描线进行,通过扫描线与边界的交点决定填充每条线上的适当部分(如图 9.34 中扫描线 $y=3$ 的情况)。因为边界对之间的像素串一定在 P 内部,可以用适当颜色对其进行填充。

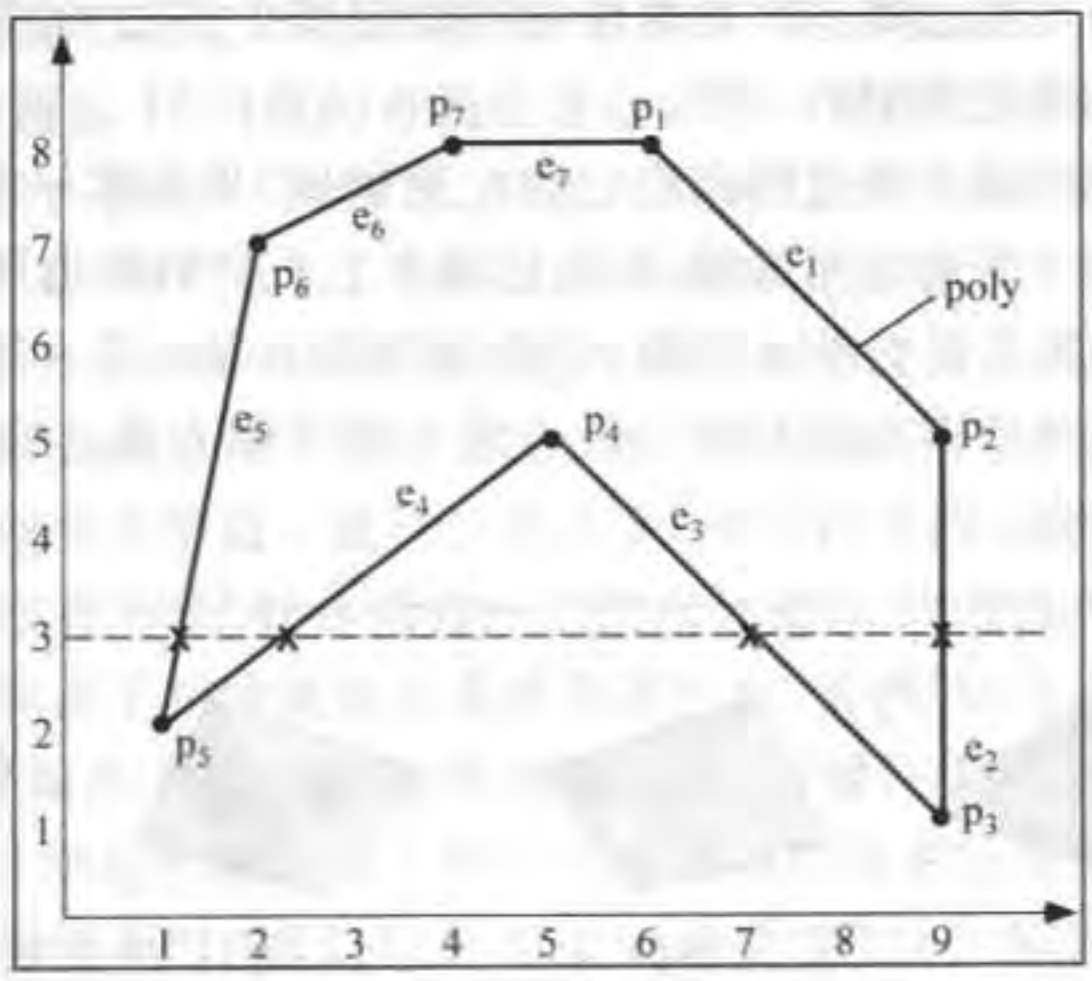


图 9.34　填充扫描线上的适当部分

下面的伪代码给出了填充的过程：

```
for (每条扫描线 L)
{
    计算 L 与 P 的所有边的交点
    按 x 大小为这些交点排序
    填充所有交点对之间的像素串
}
```

例如，图中的扫描线 $y=3$ 与 4 条边 e_2, e_3, e_4, e_5 相交。对这 4 个交点的 x 值舍入取整（取整的方法后面会介绍），然后排序得到序列 1，2，7，9。接着填充了两个像素串：第一个从第 1 列到第 2 列，第二个从第 7 列到第 9 列。

需要注意的是，为排序后的边交点配对的过程中使用了内部-外部测试（inside-outside test）。沿着扫描线移动，在每个交点处或者是进入 P 或者是离开 P，也就是说“内部性”（insideness）会在交点处发生改变（内部性有时也称作奇偶性（parity），也可以说在每个交点处奇偶性发生变化）。如果是进入多边形，那么要填充接下来的像素；如果是离开多边形，接下来的像素就不应该填充。如果整个多边形 P 都在扫描线起始处的右侧，那么奇偶性的初始值为“出”。算法充分利用了横向跨度的一致性，扫描线上与一个内部像素相邻的几个连续像素常常也是内部像素。这样可以通过最少的计算填充整个像素串。

9.7.1　边界上的哪些像素属于多边形

一个场景包含几个多边形是很常见的事情，而且其中几个多边形可能会彼此相邻共用一条公共边。这样的多边形称为彼此邻接的。如果不认真考虑的话，我们的算法可能会把公共边上的像素先设置为一个多边形的颜色，然后又设置为另一个多边形的颜色。像这样把一条边绘制两次可能会导致混乱的视觉效果：如果应用程序恰好是在 XOR 模式下绘图，那么这条边可能会呈现出奇怪的颜色。如果绘制结果是传送给自动记录照相机，那么绘制

两次可能导致这条边的亮度过高。因此算法必须确定哪个多边形"拥有"这条边,这样每条邻边才会只属于这两个多边形中的一个。

一种效果较好的规则是令多边形拥有它的左侧边(如果是水平边,就是下面的边)。所以如图 9.35 所示,当两个多边形相邻时,公共边属于它右侧的多边形,因此只会用这个多边形的颜色绘制一次。如果公共边是水平的,它将使用它上方的多边形的颜色绘制。图 9.35 用简单的三角形显示了两个多边形共有一条边的 4 种可能方式。每种情况下,公共边都应使用多边形 B 的颜色绘制。

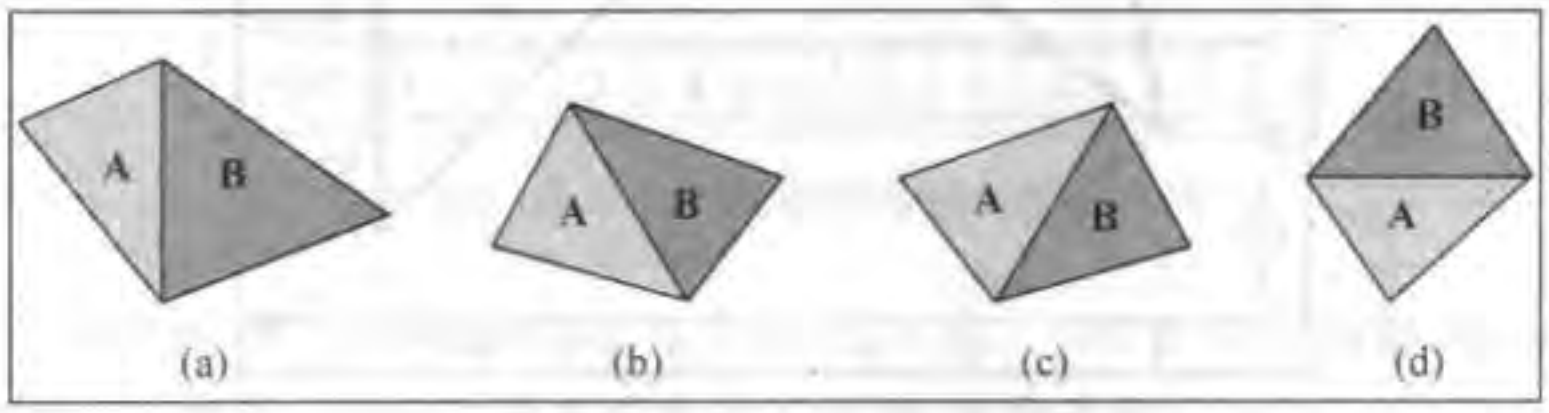


图 9.35 属于多边形 B 的公共边

如何在执行"填充所有交点对之间的像素串"这个命令时应用这条规则呢?图 9.36 给出了一个有 9 个顶点的多边形 P 的例子。中心在 P 内部的像素已经着色,但是哪些中心在边界上的像素应该着色呢?由于左侧和下面的边属于 P,所以这些边上的像素要着色,而那些位于右侧和上面边上的像素不着色。

因此在沿着扫描线填充两个交点之间这部分的过程中,如果最左边的像素位于边界上,那么填充的部分应包括这个像素,而不包括最右边的像素。这样就不会影响那些属于多边形 P 右边或上边的其他多边形的任何像素(参看图 9.36)。

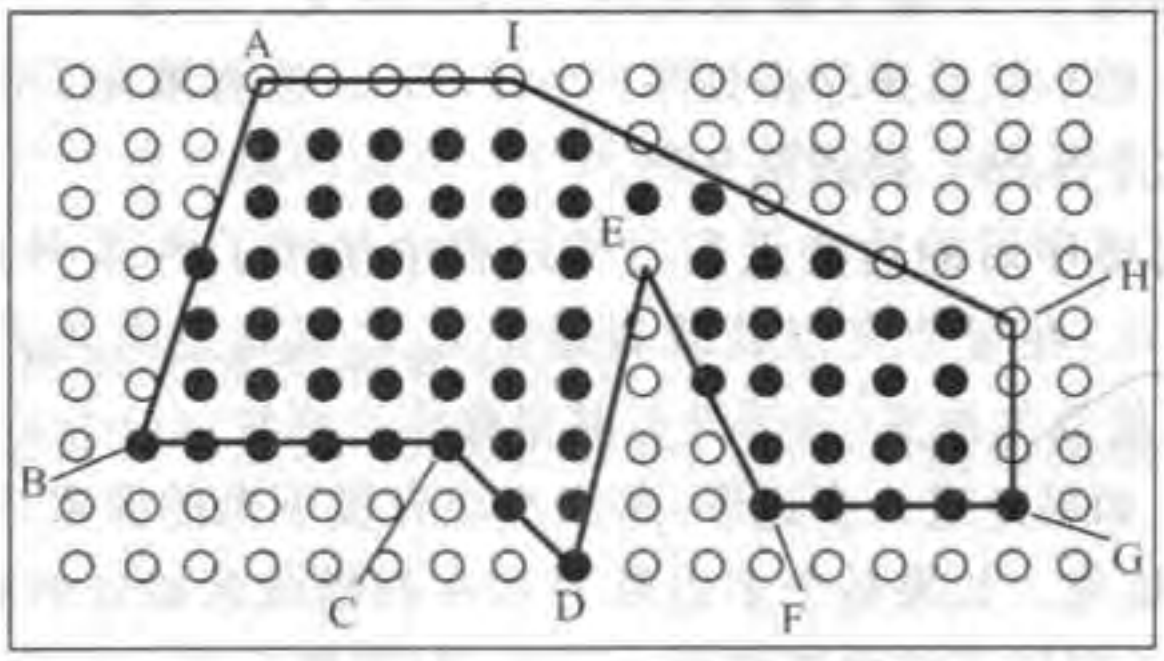


图 9.36 填充内部像素之间的部分

由于扫描线与边界交点一般在两个整数之间,所以要对其上取整或下取整,以得到填充部分正确的起点和终点。例如,假设一个像素串的左交点为实数值 xLeft,右交点为 xRight。我们要分别计算这个像素串中的起始像素 xFirst 和终止像素 xLast。根据前述规则,xFirst 是大于等于 xLeft 的最小整数。类似地,xLast 是严格小于 xRight 的最大整数。

处理扫描线相交于边界端点的情况

经常会出现一条扫描线直接穿过一条边的端点的情况(因为边的端点为整数)。为了获得正确的奇偶性,需要把这类穿越当作相交来处理。

例如在图 9.36 中，穿过顶点 H 的扫描线“看到了”两个交点（一个是与边 GH 的交点，一个是与 HI 的交点），所以 H 两侧的奇偶性看起来是一样的。但是这样会使我们将 H 的右侧填充，这显然是错的。另外一种情况，多边形拥有位于下面的水平边线 BC，所以我们希望在 B 处看到一个交点（为什么？）。但多边形不拥有上面的水平边 AI，所以希望在 A 处的交点数为偶数个。有这么多种不同的情况似乎需要相当复杂的规则集合！

然而，有一种简单的规则适用于各种情况：在交点计算中，忽略扫描线与边界的上面端点产生的交点，并完全忽略水平边。这样在图 9.36 中可以看到，正如我们所希望的，顶点 A 和 I 处没有出现交点，在 B 和 H 处各只有一个交点。图 9.37 中显示了扫描线求交时可能会遇到的其他情况，并标出了扫描线通过多边形顶点时“看到”的交点数目。在每种情况下，对于交点右侧的奇偶性都是正确的数值（验证一下）。需要注意的是，这个规则产生了一个不合理的效果：图 9.36 中处于两条边上端点处的 E 点的像素没有画出。

综上，我们对填充算法中“计算 L 与 P 所有边的交点”这一部分进行调整，使其包含以下改进：

- 计算扫描线与 P 的所有边的交点；
- 丢弃与水平边的交点和与任意边上端点的交点。

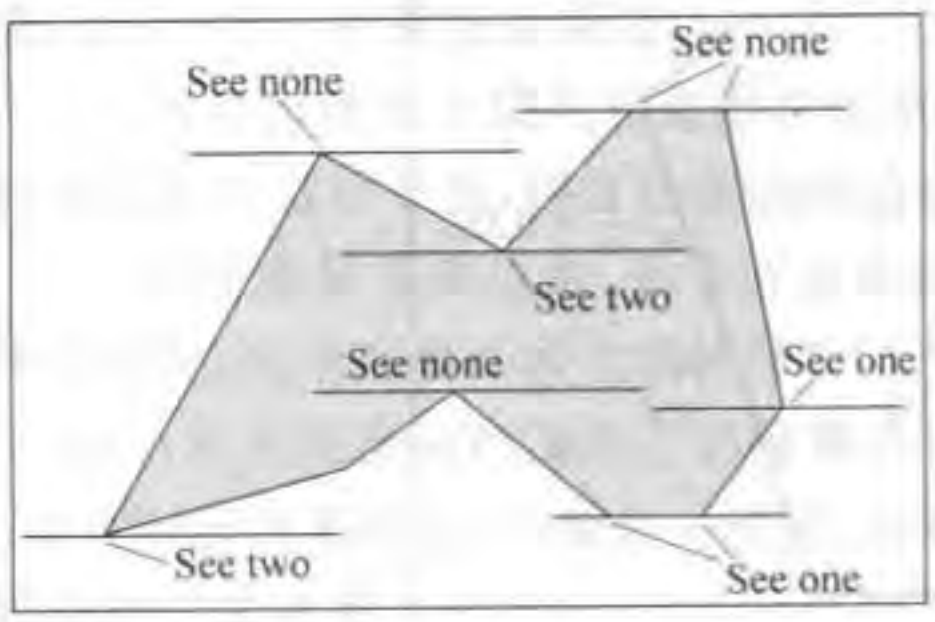


图 9.37　在多边形边界上看到的交点个数

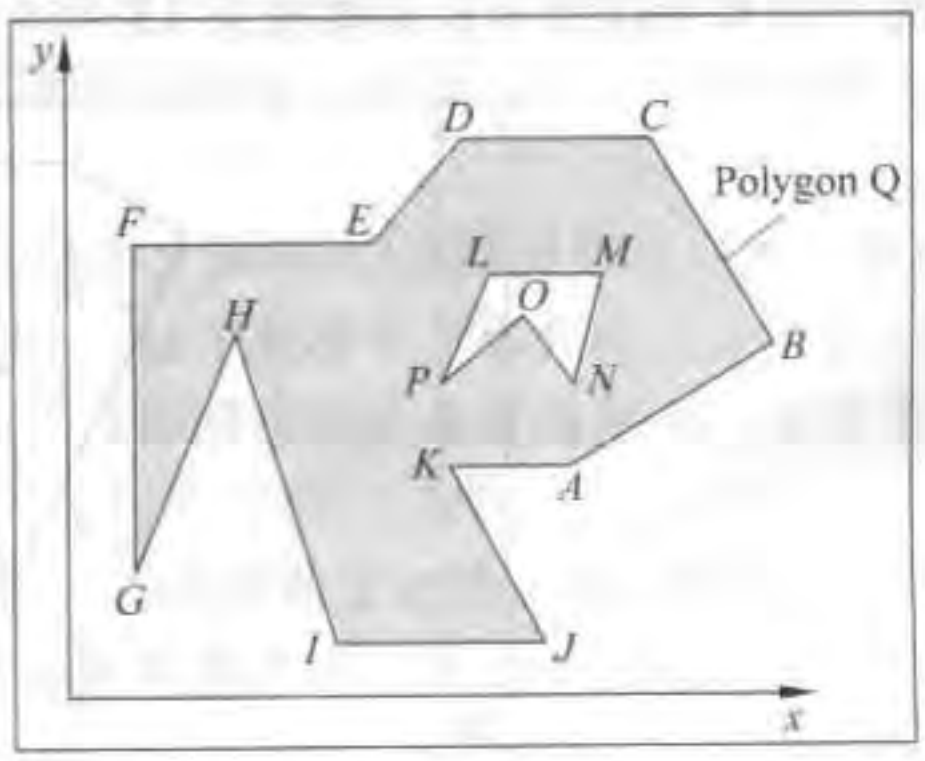


图 9.38　要填充的多边形示例

练习

练习 9.7.1　对方法进行测试

考虑图 9.38 中多边形 Q 的例子，这个多边形有如下 16 个顶点。

A：(52,30)，B：(74,43)，C：(60,60)，D：(38,60)，E：(30,50)，

F：(10,50)，G：(10,28)，H：(22,41)，I：(33,10)，J：(50,10)，

K：(39,30)，L：(40,44)，M：(54,44)，N：(46,34)，O：(45,42)，

P：(33,34)。

最后 5 个点构成了 Q 的一个空洞。画出这个多边形，然后判断前面我们讲述的方法能否正确填充这个多边形。

9.7.2 改进算法性能

要改进前一节中算法的性能,我们需要寻找其中最耗时的部分。这部分就是大量的扫描线与边交点的计算。为了减轻这个负担,可以建立并维护一个简单的列表以便于快速地确定交点。这个列表称为活性边表(Active-Edge List,AEL),有了它算法便能够充分利用边一致性(edge coherence)。边一致性包含两部分内容:

- 与扫描线 y 相交的边中有许多也与扫描线 $y+1$ 相交;
- (直线的一条性质)从一条扫描线到下一条扫描线,交点的 x 值增量相等。

在填充过程中,可以很简单的参考 AEL 确定要在每条扫描线上填充的像素串。AEL 中保存着当前扫描线(正在被填充的扫描线)与所有边的交点的 x 值。x 值按顺序保存,所以根据奇偶性规则,前两个 x 值确定了第一个像素串,接下来的两个值确定第二个像素串,依此类推。

例如,假定正在填充图 9.39 所示的多边形,已经到达扫描线 $y=50$。这条扫描线与多边形的四条边有交点。这些交点的 x 值很容易计算,按顺序分为 45、56.66、70 和 100。根据这个结果可以很容易地填充从 45~56 和从 70~99 这两段区间。

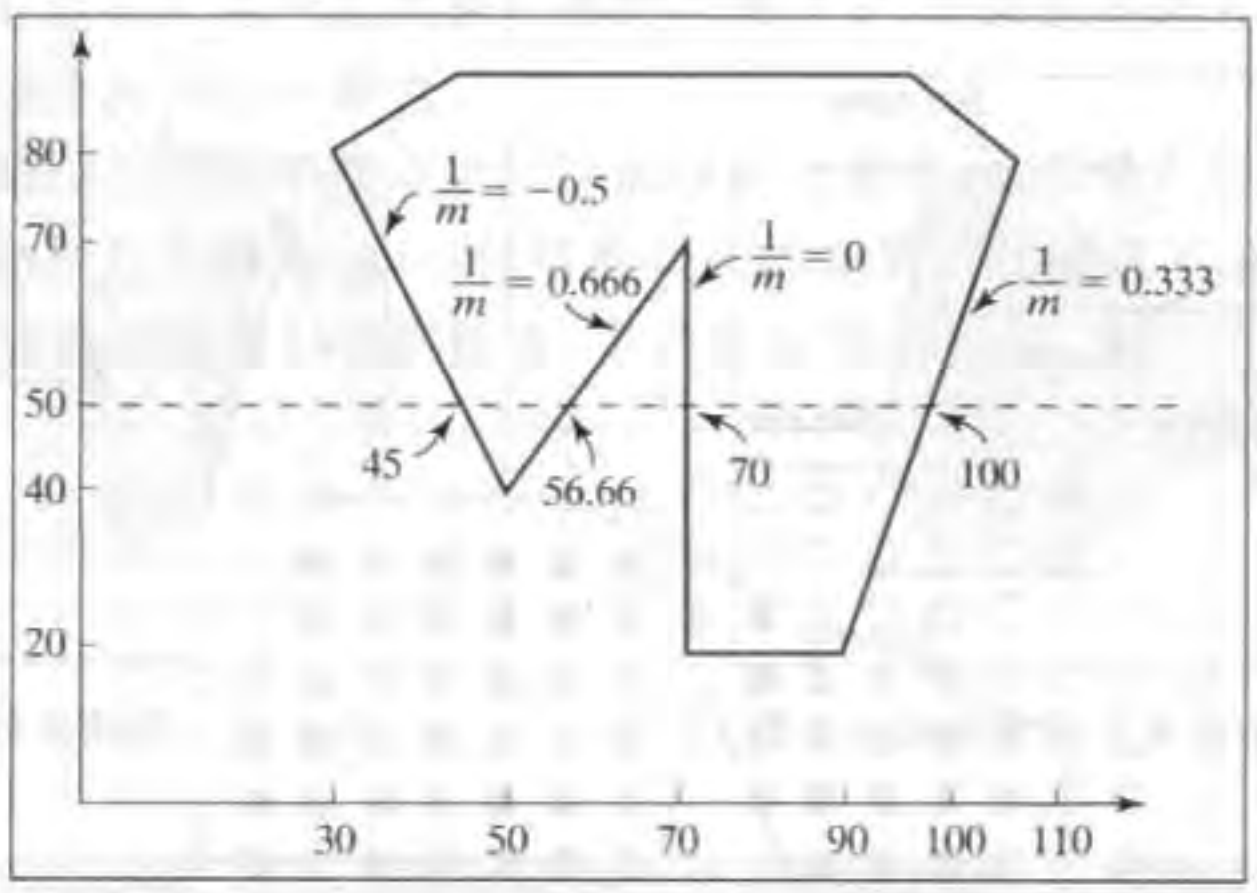


图 9.39 填充多边形示例

图 9.40 中提出了在这种情况下 AEL 的一种链表形式,图中显示了按顺序保存的当前的四个交点以及其他一些辅助数据,这些辅助数据能够使这种算法快速更新 AEL,以便于下一条扫描线使用。

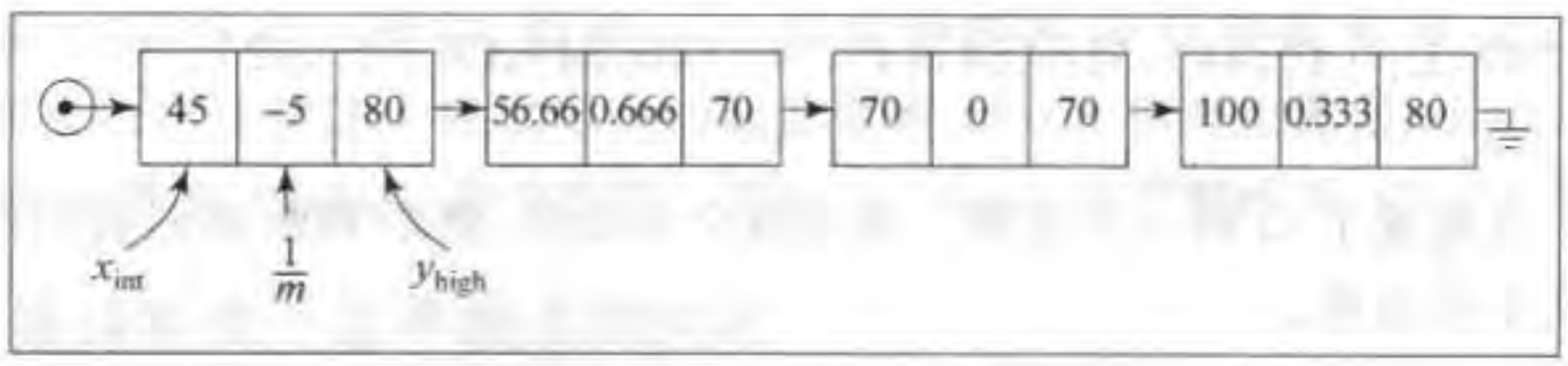


图 9.40 扫描线 y=50 的 *AEL*

确切的说，对于每条与当前扫描线相交的边，AEL 中都包含下面 3 项数据：

(1) 这条边与当前扫描线交点的 x 值 x_{int}。

(2) 这条边斜率 m 的倒数 $1/m$。

(3) 这条边的上端点 y 值 y_{high}。

其中第二项用于确定扫描线 $y=51$ 与各条边交点的位置。需要注意的是，如果一条边的斜率是 m，那么 y 向上移动一个单位将导致交点的 x 值增加 $1/m$（验证这一点）。因此每条边的 $1/m$ 值直接存储在该边的 AEL 中（如图 9.40 所示）。（例如，最左侧的边在 x 方向有20 个单位的跨度，在 y 方向有 40 个单位的跨度。因此其斜率的倒数为 20/40。检验图中给出的其他数据）。于是只需对 x_{int}加上 $1/m$ 就得到了这条边与其上方下一条扫描线的交点，这样就充分利用了边一致性。

需要注意的是，增加 $1/m$ 并舍入正是 Bresenham 算法沿着一条线一步一步移动时所做的事情。实际上，在此处使用 Bresenham 方法（根据某个误差项的符号递增，然后更新这个误差项）计算交点可以使得效率达到最高。

在填充过程中移动到下一条扫描线时，除了沿着一条边递增之外，还可能发生另外几种情况：

(1) 新的扫描线可能刚好超出（在其之上）AEL 表示的一条边。AEL 中保存着每条边的上端点 y 值，这使得这种情况很容易确认。如果新的扫描线 y 的值超出了一条边的上端点值，就从 AEL 中删去这条边。

(2) 如果扫描线 y 值与多边形中的某些边的下端点 y 值相等，这就说明它遇到了新边。此时通过查找另一个表，将这些新边的信息加入 AEL。这个表将在下面介绍。

(3) 对于非简单多边形，如果两边相交，那么边交点的顺序可能会发生颠倒。如果发生这样的情况，记录列表必须重新排序。

所以，填充完当前扫描线上的像素串之后，递增 y，同时更新交点的 x 值，必要的话还要从 AEL 中删除某些边，添加另一些边，并将交点按 x 值重新排序。下一条扫描线也可以按照同样的方式处理。

边表

处理下一条扫描线时，为了确定要添加哪些边，一种方法是检测 P 的每个顶点，以确定当 y 增加时遇到了哪些边。但是这种方法效率不高。我们可以提前获取 P 中每条边的信息，然后将其巧妙的放置在一个边表（Edge Table，ET）中。在更新 AEL 时，利用 ET 可以快速获取所需的信息。

边表由一个链表的数组 edgetable[]构成。对应每一条扫描线有一个链表。图 9.41 给出了图 9.38 中多边形 Q 的 ET 中的一部分。边表可以在最初遍历多边形以去掉平行边和处理其他边时构造。

每条扫描线的链表中包含 Q 中所有其下端点在这条扫描线上的边的信息（这种方法有效地根据其下端点的 y 值对 Q 的所有边进行了排序；有一种说法是 ET 为多边形的边执行了一次桶排序（bucket sort））。可以看到，实际上 ET 中的许多表都是空的。

每条边的表示方法与 AEL 中边定义的类型相同，并写入正确的 y_{high}和 $1/m$ 字段，x_{int}字段的值为该边底端点的 x 值。

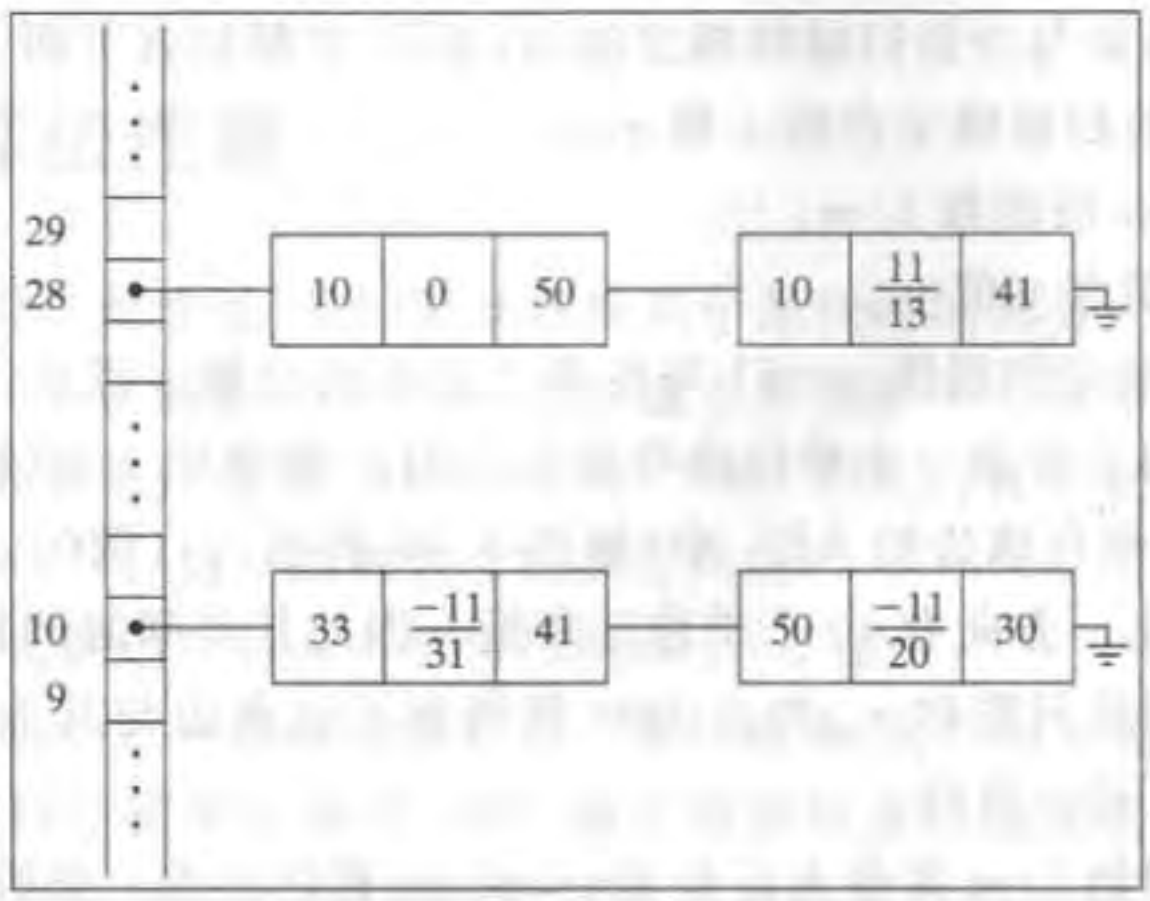


图 9.41 表示多边形 Q 的边表

更新 AEL 时,利用边表可以很容易确定在 AEL 中要添加哪些边。对于位于第 y 行的新的当前扫描线,edgetable[y]保存的边都需要添加到 AEL 中。x_{int}值自动包含最初的交点 x 值,其他字段也被正确地加载。因此要在 AEL 中添加新记录,只需调整少量几个指针即可。在填充像素串之前,对 AEL 按照 x_{int}的升序重新排列,以维护正确的奇偶性。

图 9.42 给出了整个算法的框架。AEL 开始为空,然后算法从 $y=0$ 开始依次填充每条扫描线。开始 y 值随着主循环快速递增,直到在 AEL 中添加第一批边,在这之前都不进行绘制。案例分析 9.7 给出了这个算法的更多细节。

```
AEL=NULL;
for(y=0; y<=maxRow; y+1)
{  // APE 最初为空
    <将 edgetable[y]中的所有边加到 AEL 中>
    if( AEL!=NULL)                          //判断是否有边处理?
    {
      根据 xInt 值对 AEL 排序
      使用 AEL 信息沿着 y 填充像素值
      将 AEL 中满足 yupper==y 的记录删除
      用斜率的倒数更新 xInt 的值
    }
}
```

图 9.42 多边形填充算法框架

Lui 等人【Lui00】最近提出了一个非常高效的多边形填充算法,该算法使用一个基于"三元组"的数据结构来描述多边形的每条线段(三元组由一个 y 值、一个线段左侧端点的 x 值和一个线段右侧端点的 x 值构成)。

练习

练习 9.7.2 表填充算法

表填充算法能够快速填充某一类多边形。首先,对多边形上的每条边进行扫描转换(可

以使用 Bresenham 算法)，生成沿着每条边的(x,y)对。然后将其与两个数组 min[y]和 max[y]作比较测试，这两个数组中保存对于每个 y 值目前所遇到的最小 x 值和最大 x 值。如果 $x<$min[y]，就把 min[y]更新为该 x 值，max[y]的处理方法类似。最后对于每条扫描线 y，通过绘制从 min[y]到 max[y]的像素串来填充这个多边形。由于不需要排序，所以这个算法的速度很快。

请问多边形要满足什么几何条件才能确保使用该算法正确填充？针对该算法，给出几个能够正确填充的多边形的例子，再给出几个不能正确填充的多边形的例子。

练习 9.7.3　栅栏填充算法

对多边形的栅栏填充(fence-fill)方法利用了对图像中像素求"补"或求"反"这个概念。对于一个二值图像来说，求补就是把值为 1(白色)的像素设置为 0(黑色)，反之亦然。求补的一个基本性质是求补两次就恢复原来的值。

该填充算法的流程是这样的：建立一个垂直的"栅栏"——它可能通过要填充的多边形的某个顶点。把所有像素设置为 0。然后对于多边形的每条边，在通过这条边的每条扫描线上，对从该边到栅栏之间的所有像素求补。一条扫描线上位于多边形之外的所有像素会被求补偶数次，因此值为 0。那些位于多边形内部的像素被求补了奇数次，所以值为 1(要被填充)。这个算法与栅栏的位置无关。

在纸上画出几个多边形，手算使用栅栏填充方法填充这些多边形。证明这种方法对于多个多边形和有空洞的多边形也适用。如果将这种方法应用于一条线，而不是有着真实内部和外部的多边形，会产生什么效果？

9.8　走样和反走样技术

前面的讨论中多次出现"锯齿"这个词，它是"走样"(aliasing)的一种形式。而走样是光栅显示的一种固有性质。产生走样现象的原因是像素本质上的离散性，也就是说，像素是以某一固定尺寸的矩形阵列的形式出现在显示器上的。图 9.43 显示了字符'a'的走样图像。

以图 9.44(a)所示的黑色矩形为例，如果这个矩形覆盖了大量的像素，它的边界尽管还是有锯齿，但是看起来会相对平滑一些。但是如果这个矩形只覆盖了少数几个像素，如图 9.44(b)所示，那么"锯齿"就会非常明显，以至于能够扰乱视线。在本图中，像素设置为黑色的依据是该矩形是否覆盖了一个特殊的点，即像素的中点。使用像素的中点对矩形进行采样，看它是否在矩形上，这是一种非常有效的方法。以这种采样为依据，整个像素区域的颜色可以设置为黑色或白色两种颜色。

图 9.43　字符'a'的走样图像

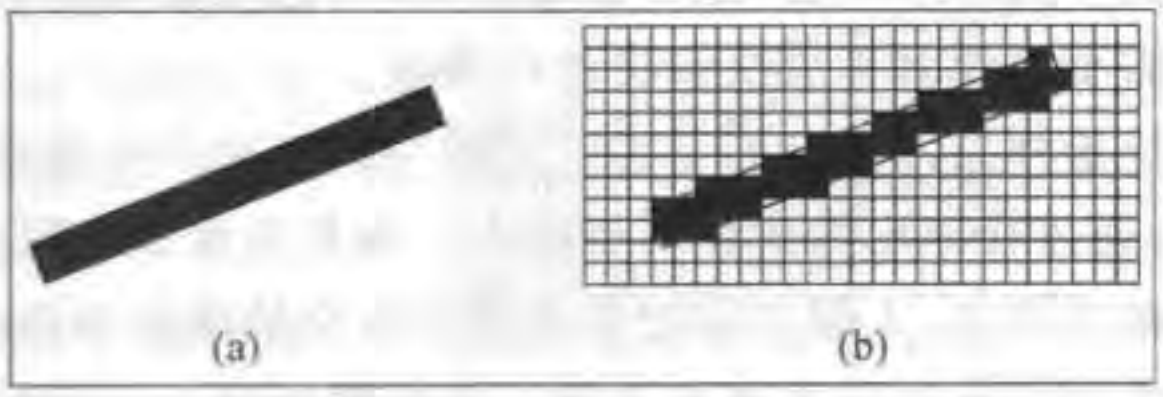


图 9.44　一个矩形的走样现象

(a) 图形覆盖了较多像素；(b) 图形覆盖像素相对较少

但这种采样策略可能会导致小物体(比如 3D 场景中远处的物体)完全消失(见图 9.45(a))。如果这个物体处于像素的中心之间,就完全不会被显示。图 9.45(b)演示了动画中这样的物体如何会一会儿出现、一会儿消失地闪烁。该物体可能在某一帧中覆盖了一个像素的中心,而到了下一帧就不覆盖了,这样产生的结果是无法让人接受的。

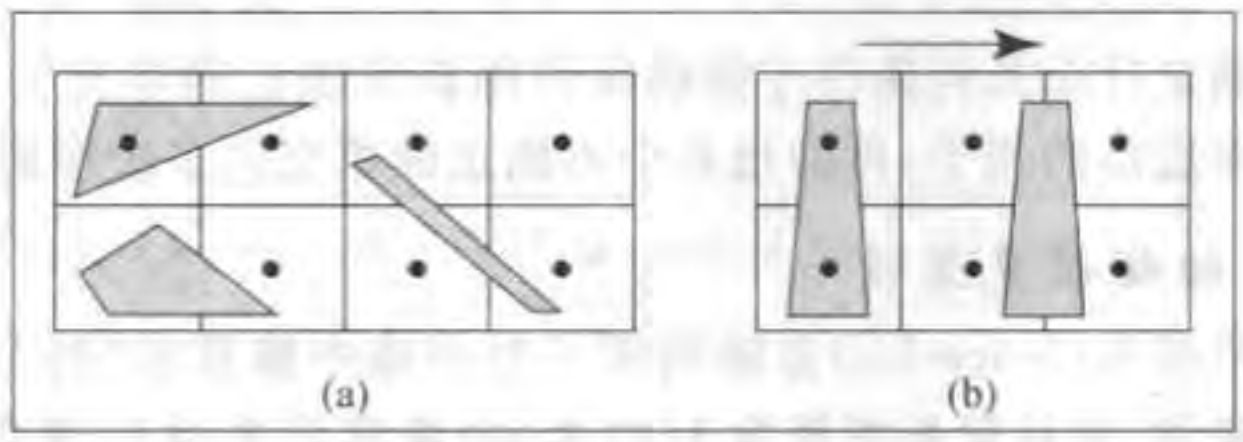


图 9.45 小物体由于走样而消失

粗略地说,如果对一个快速变化的信号采样频率过低,所得样本表示的会是低频变化的信号:原始信号的频率看起来被较低的"走样"频率所代替。图 9.46(a)显示了一个快速变化的方波,以图中黑点标出的等间距采样。如果只从这些采样结果看(图 9.46(b)),这个信号看上去是它的走样结果:是一个频率较低的方波。

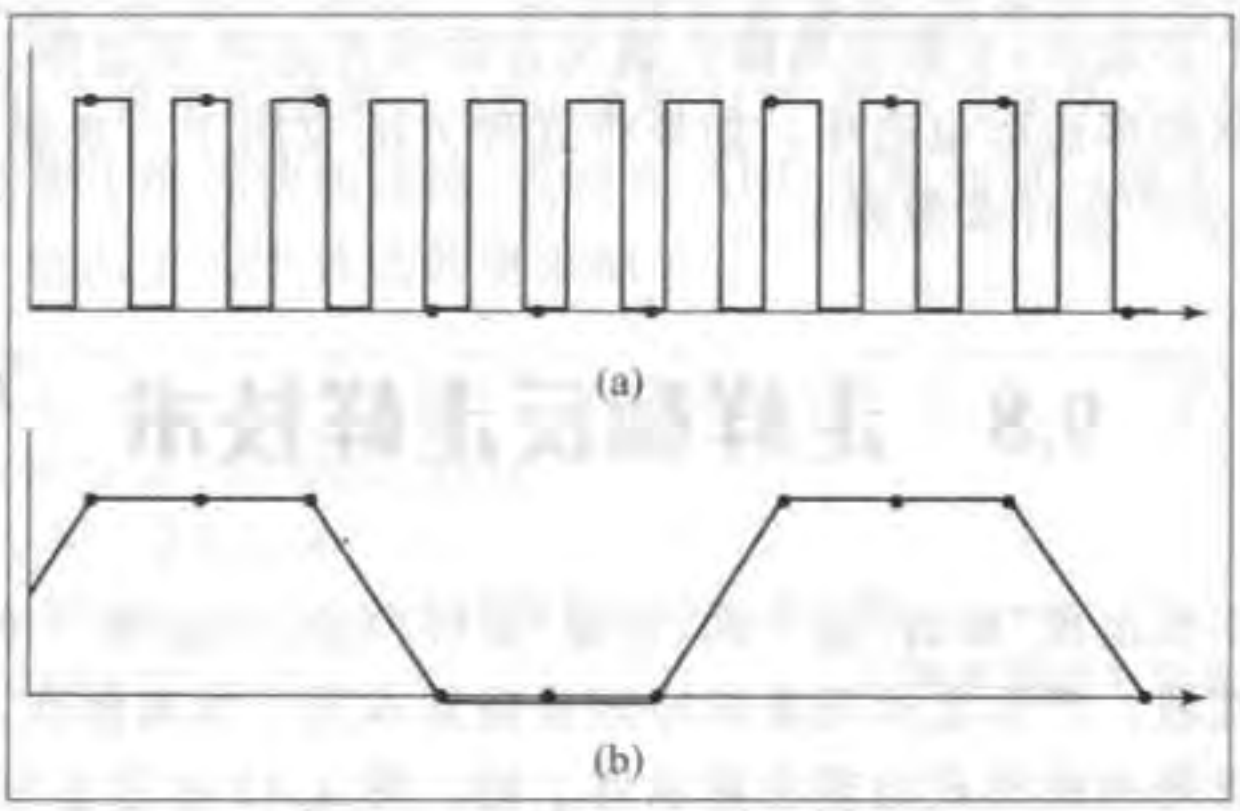


图 9.46 采样频率过低使得信号看上去走样

9.8.1 反走样技术

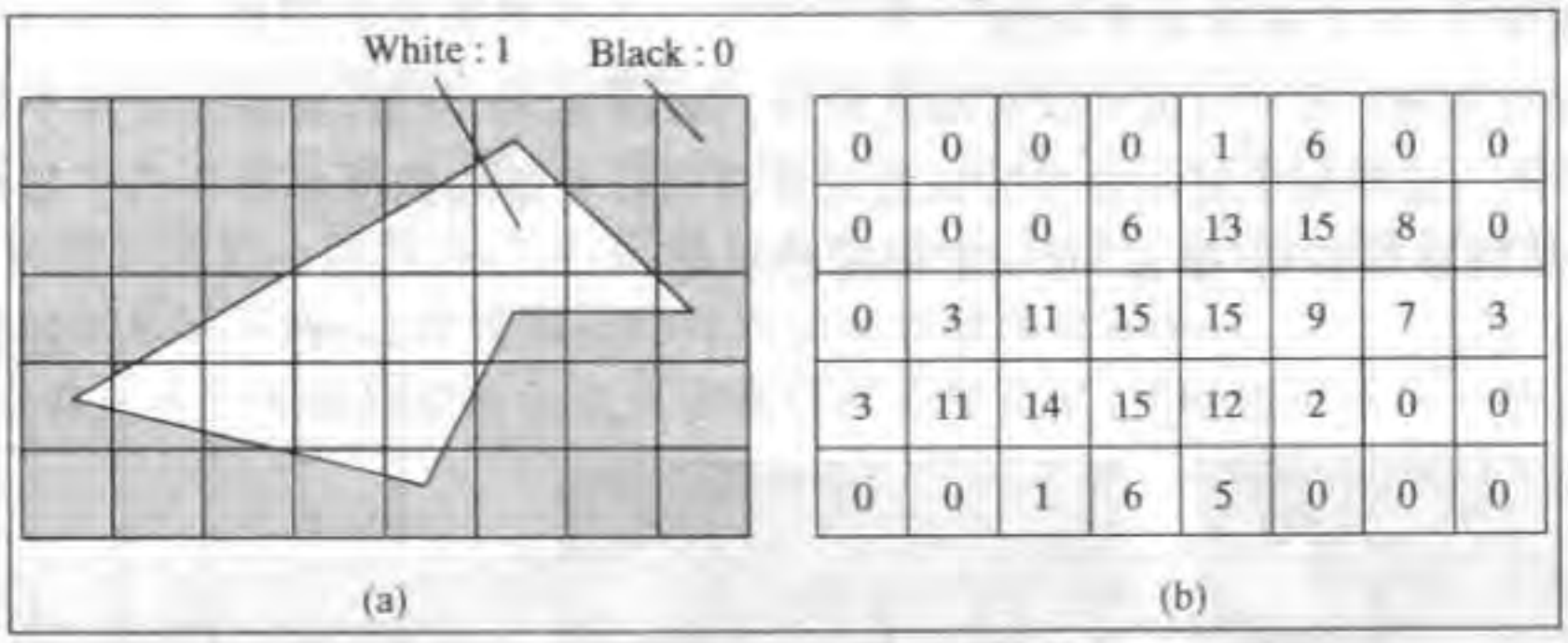
如何降低由于采样不足而产生的走样现象呢?采用分辨率更高的显示设备,配以更好的绘制算法,对解决问题有所帮助,因为这可使锯齿相对物体会更小一些,但仍存在。所以我们需要寻找其他方法解决走样现象。

反走样技术涉及到某种形式的"模糊"来产生更平滑的图像。对于在白色背景中的黑色矩形,通过在矩形的边界附近掺入一些灰色像素,可以柔化从黑到白的尖锐变化。从远处观察这幅图像时,人眼能够把这些缓和变化的暗影融合在一起,从而看到了更加平滑的边界。

常用的反走样方法有 3 种:即前置滤波(prefiltering)、过采样(supersampling)和后置滤波(postfiltering)。

前置滤波

前置滤波技术根据物体的覆盖率(coverage)计算像素的颜色。覆盖率是指某个像素区域被物体覆盖的比例。考虑图 9.47(a)所示的在黑色背景上的白色多边形的扫描转换结果。假定亮度值 0 表示黑色,1 表示白色。把这个多边形放在方格线中,其中每个正方形的中心对应显示器上一个像素的中心。被多边形覆盖了一半的像素的亮度值赋为 1/2;被覆盖了三分之一的像素赋值为 1/3;依此类推。如果帧缓冲区的每个像素有 4 个比特位,那么 0 表示黑色,15 表示白色。一个像素如果被多边形覆盖了四分之一,它应该赋值为(1/4)15,舍入之后结果为 4。图 9.47(b)给出了根据每个像素的覆盖率计算得到的值。(如果只使用像素中心对多边形采样,当多边形覆盖像素中心时取值 15,否则为 0,则该像素阵列的值会是什么?)



0	0	0	0	1	6	0	0
0	0	0	6	13	15	8	0
0	3	11	15	15	9	7	3
3	11	14	15	12	2	0	0
0	0	1	6	5	0	0	0

图 9.47　利用像素区域被覆盖的比率

当然,求出每个像素覆盖率的几何计算可能非常耗时。但是现在已经有很多有效的方法被研究出来,例如 Pitteway 和 Watkinson 提出的方法【Pitteway80】,还有最近 Xiaolin Wu 提出的算法【Wu91】。这些算法都采用增量的思路计算每个像素的覆盖率,并且都使用整数计算。

综上所述,前置滤波对扫描转换后的各种物体的详细几何形状进行操作,并根据这个物体对每个像素区域的覆盖率计算每个像素的平均亮度。对于非多边形的形状,前置滤波的计算代价可能非常大,所以我们需要寻找其他反走样的方法。

过采样

既然走样是由于对物体采样点数过少,那么我们可以尝试对每个像素多点采样来降低走样现象。这种技术称为过采样(supersampling),即对场景进行超过显示像素数目的亮度采样,然后取几个样本的平均值来形成要显示的像素值。

图 9.48 给出了一个双倍采样的例子。该物体(这里是一个倾斜的条状物)在 x 方向和 y 方向的采样密度都是显示像素密度的 2 倍。图中的正方形表示显示出来的像素,'x'表示场景的采样位置。

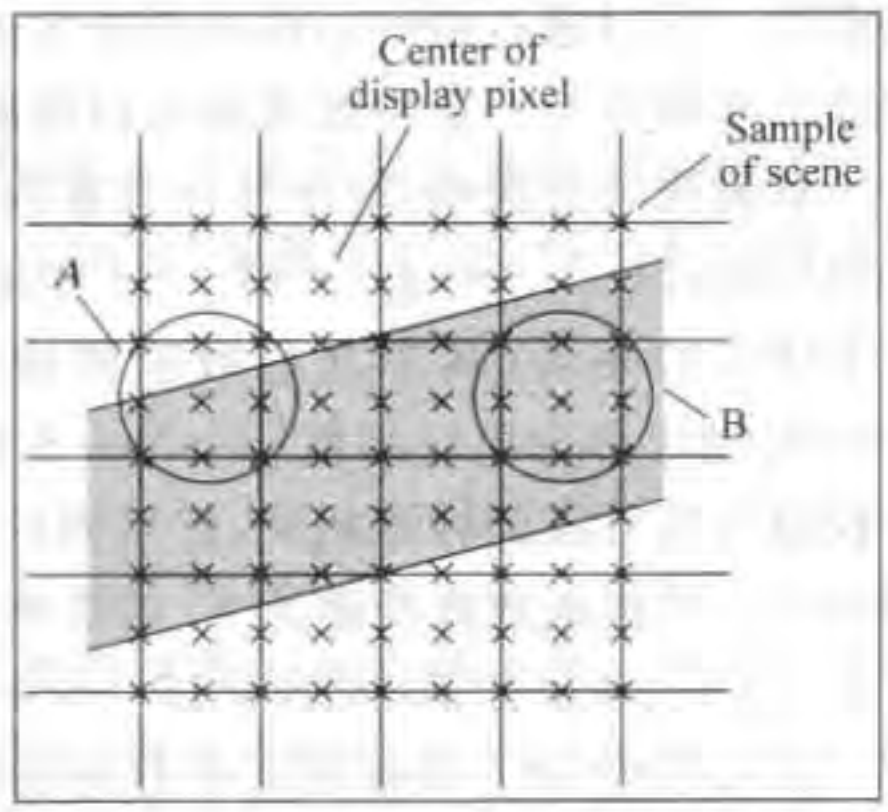


图 9.48　采用过采样进行反走样

每个最终显示的像素值可以通过 9 个"相邻"样本点的平均值构成,即该显示像素的中心位置的样本点以及围绕它的 8 个样本点。可以看到,有些样本点是被几个像素重复利用了。(哪些样本点?)中心在 A 的显示像素的数值基于条状物内部的 6 个样本和背景的 3 个样本。因此它的颜色设置为条状物颜色的 2/3 加上背景颜色的 1/3。处在 B 点的显示像素数值基于 9 个条状物内部的样本,因此它的颜色设置为条状物的颜色。

图 9.49 左图显示了一个分辨率为 300×400 像素的场景,场景的"锯齿"很明显,特别是在物体的侧面处。图 9.49 右图显示了双倍采样得到的结果。对于同样的场景,以 600×800 的分辨率采样,300×400 显示像素中的每个像素取过采样数据中相应的 9 个相邻样本点的均值。虽然有一定程度的模糊,但可以看到"锯齿"被明显地柔化了。

一般来讲,过采样方法对于每个显示像素在 x 方向和 y 方向各计算 N_s 个样本,然后取一定数量的相邻样本的平均值作为每个显示像素的值。例如,对于 $N_s=4$ 的过采样,每个显示像素需要求 16 个样本的平均值。

不使用过采样($N_s=1$)也可以实现反走样。如图 9.50 所示,对场景在每个显示像素的 4 个角点采样。这样每个显示像素的亮度设置为在其 4 角取到样本的均值。虽然没有采用过采样,仍然能够观察到"锯齿"有一定程度的柔化。

图 9.49 使用两个不同的采样数绘制的物体
左侧为未进行反走样操作效果;右侧为双倍采样效果

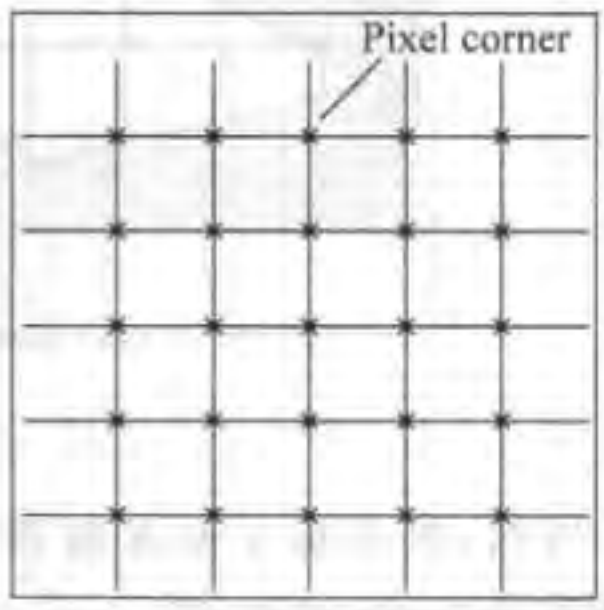


图 9.50 采用角采样的反走样方法

后置滤波

双倍采样方法取 9 个相邻样本的平均值作为显示像素的亮度值,每个样本具有同样的重要性。可以通过给中心样本以更大的权重,给 8 个相邻样本较小的权重来改善这种方法造成的模糊效果。这种改变还可以使更多的相邻样本加入到均值计算中来。

后置滤波方法中,每个显示像素的数值是场景中一个适当规模的相邻样本集合的加权平均(weighted average)。图 9.51 中给出了双倍采样时的情况。图中每个数值表示场景中一个样本的亮度,其中灰色数值的位置表示显示像素的中心。表示权重的正方形模板(mask)也可称为窗口函数(window function)依次覆盖每个灰色像素。然后把窗口中的每个权值与其对应的样本相乘,把得到的 9 个乘积相加就是这个显示像素的亮度值。例如,当图中所示的模板放在亮度为 30 的样本上时,加权平均的结果为:

$$(30)/2+(28+16+4+42+17+53+60+62)/16=32.625$$

舍入后为 33。该模板将中心的权重设为其 8 个相邻像素权重的 8 倍,而权重之和总是为 1。

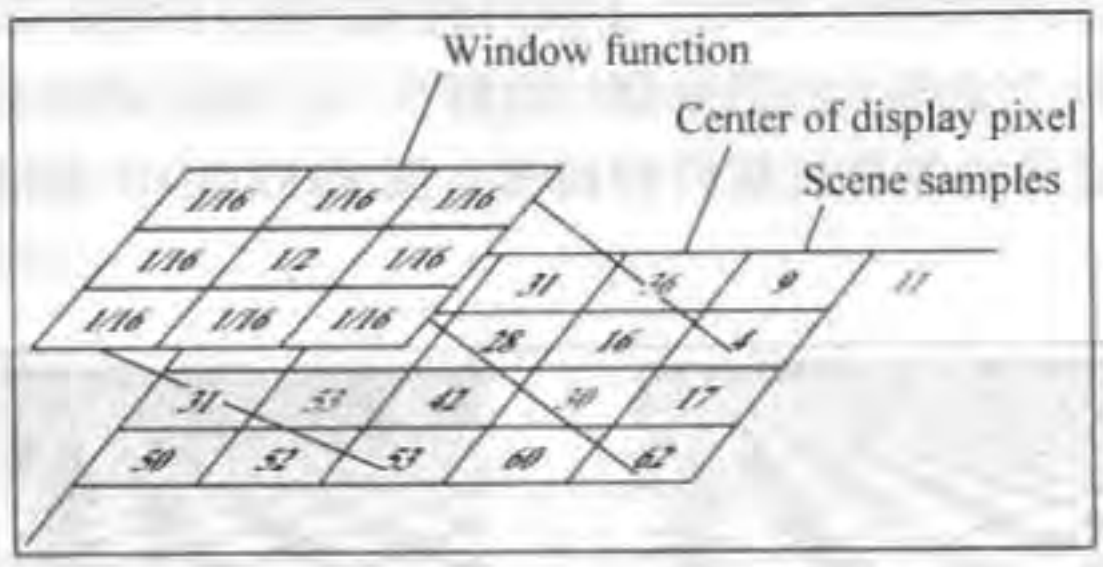


图 9.51　对以副图形进行后置滤波处理

值得注意的是，前面介绍的过采样仅仅是后置滤波的一种特例，只不过所有的权值都是 1/9。从信号处理领域引入的采样和滤波理论为研究作为后置滤波器的各类不同的窗口函数提供了分析方法。有时还会使用更大一些的模板，比如 5×5，甚至 7×7 的模板，这些模板考虑了距离中心样本范围更大的邻居样本，能够提供更好的平滑效果。

后置滤波法可以在任意 N_s 值的过采样中使用。如果 $N_s=4$，使用 5×5、7×7 甚至 9×9 模板都是比较合适的。如果 $N_s=1$，就像角采样，可以采用中心权值比较大的 3×3 模板。根据所渲染的场景的不同，这种模糊可能有效果也可能没有效果。

第 12 章将讨论与光线跟踪有关的更多高级反走样技术，同时也可以从一些联机资料获得相关信息。

练习

练习 9.8.1　过采样

对于 $N_s=1$ 的情况，画出一个与图 9.50 类似的像素栅格图，在每个显示像素中心位置防止'X'，在每个像素的四角放置's'表示场景的采样点。

练习 9.8.2　多边形反走样

设帧缓冲区像素的取值 0～15。对一个顶点为(1,3)、(6,7)、(15,4)、(11,15)和(1,8)的白色多边形的每个边进行扫描变换同时进行前置滤波，计算结果像素的值。假定背景为黑色，使用边界填充算法用白色填充这个多边形。如果多边形的边上有一个空洞，那么这个填充算法是否可能失败？如果可能，应该如何弥补这个缺陷？

练习 9.8.3　角采样

对于一个有 R 行 C 列的光栅显示器，使用角采样时，一个场景要计算多少个样本？也就是说，这个光栅显示器有多少个"角"？把这种采样方法与"像素中心"采样法和双倍采样法进行对比。

9.8.2　纹理的反走样

第 8 章介绍了如何把纹理映射到各种表面上，以获得更强的真实感效果。因为通常总希望纹理能够真实地反映某种图案和图像，所以映射的纹理对走样效应更加敏感。另外，纹理本身通常也是像素映射的形式，而显示出来的像素和像素映射中的像素通常有着复杂的

几何关系。我们将介绍几个绘制纹理时会出现的特殊问题,并详细讲述几种解决方案。

图 9.52(a)给出了一个走样的经典示例:绘制在一个倾斜表面上的纹理。远处的棋盘方格已经严重地走样,呈现出各种波纹状的图案。图 9.52(b)中的图像是经过反走样处理的,结果有本质的改善。

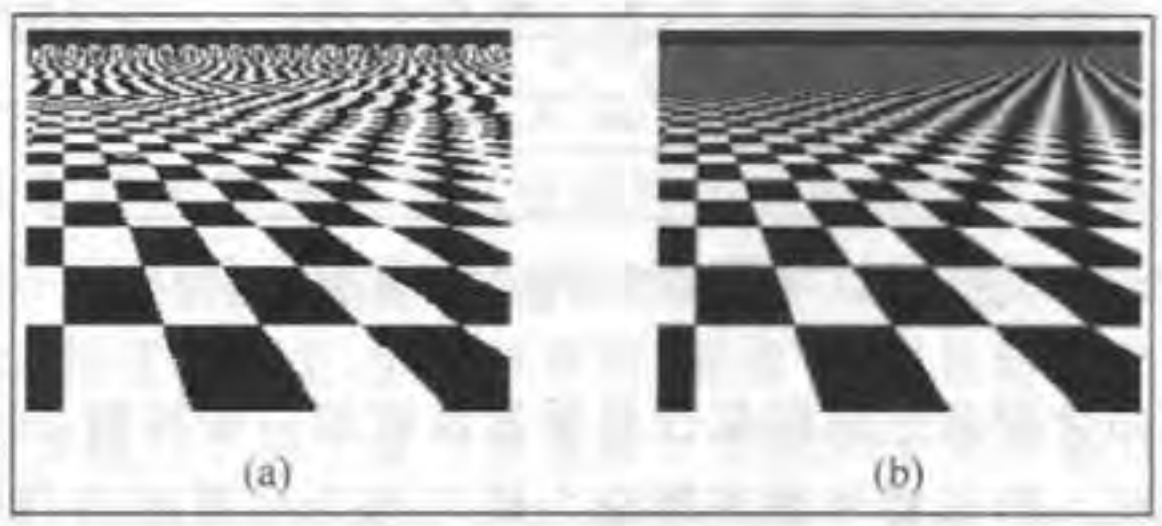


图 9.52 走样和用反走样技术改进的示例

回顾一下以前的内容,纹理定义为纹理空间中的一个纹理函数 texture(s,t),在最终绘制到显示器之前会进行一系列复杂的映射。而渲染任务却是反方向进行工作,对每个给定的显示像素坐标(x,y),计算它在纹理函数 texture()中对应的颜色。

图 9.53 显示了一个正被渲染的特定像素(x,y)和纹理空间中对应的纹理坐标(s^*,t^*)。为方便起见,我们用 $T()$表示从像素空间到纹理空间的所有映射,于是有$(s^*,t^*)=T(x,y)$。(这个映射的各个分量已经在第 8 章中介绍)。

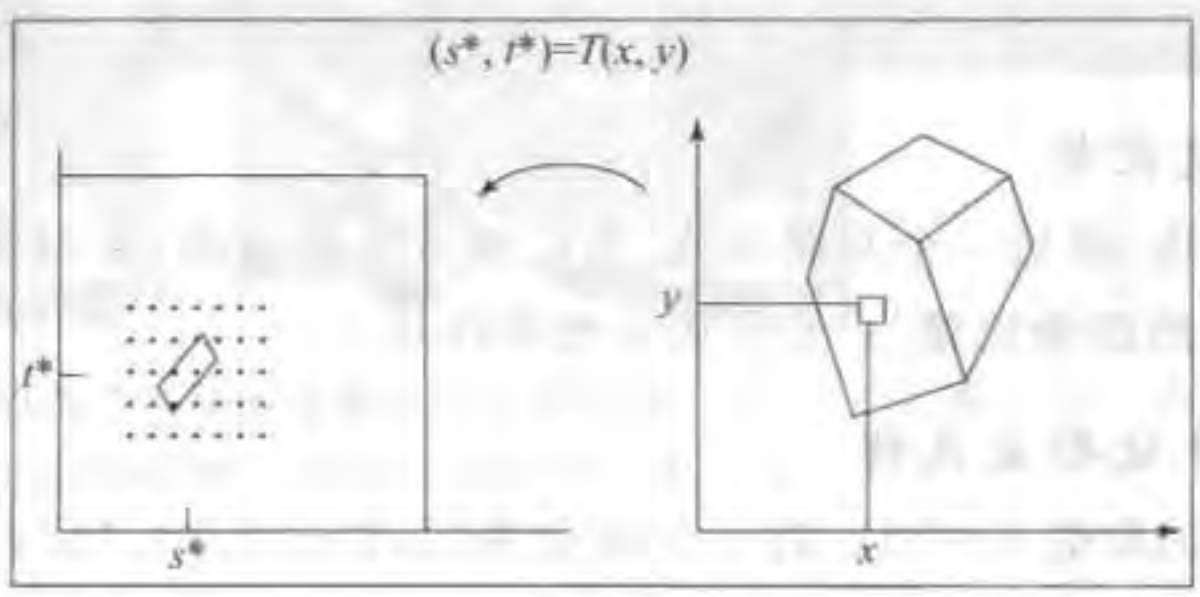


图 9.53 纹理渲染过程走样的原因

由于像素不是一个点而是有一定面积的,因此我们必须认真考虑怎样把以(x,y)为中心的方形像素映射到纹理空间。图 9.53 中也以四边形的形式画出了这个像素的映射[①]。后面我们把这样的四边形称为屏幕像素的纹理四边形。想像纹理空间由这样的纹理四边形覆盖,每个纹理四边形都来自于一个屏幕像素。纹理四边形的大小和形状取决于 $T()$的性质,要计算起来代价很高。如果纹理函数 texture(,)在纹理四边形内部有变化,而屏幕像素却只用一个采样结果 texture(s^*,t^*)着色,那么就会丢失重要信息,出现走样现象。

为了减小走样效果,我们为每个屏幕像素着色时,应使用位于对应的纹理四边形内部的若干颜色均值作为该像素的颜色值。图 9.54 显示了叠加在定义 texture(s,t)的像素映射 txtr[][]中各个纹理元素(texel)之上的一个纹理四边形。如果有一种简单的方法可以计算

① 如果显示的 3D 表面是曲面,那么四边形的边也是弯曲的。

每个纹理元素位于这个纹理四边形之内的面积，就很容易求出平均值：按照面积对每个 txtr[r][e]计算加权求和，然后除以这个四边形的面积即可。这个过程类似于前面介绍的用于反走样的前置滤波技术。不幸的是，这样的计算严重影响性能，因为很难计算出每个纹理元素被四边形覆盖的面积。

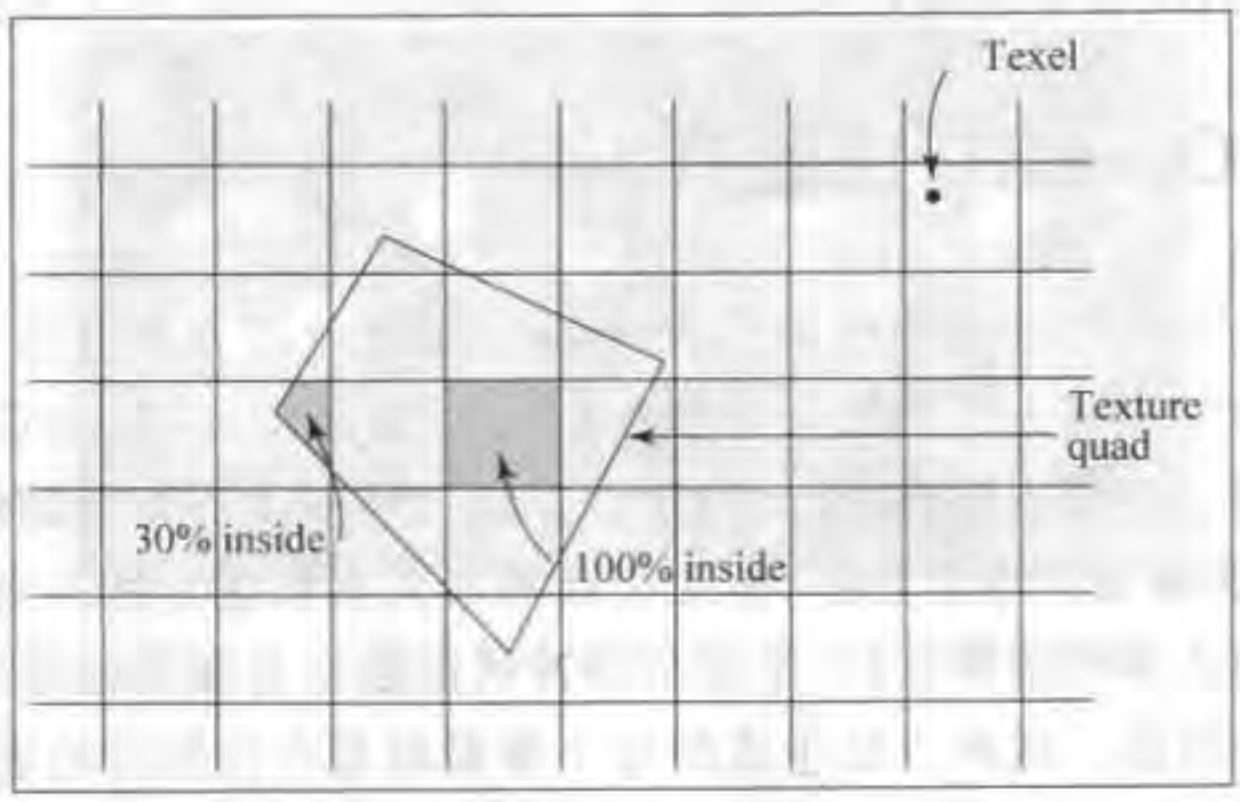


图 9.54　计算用面积加权平均的颜色

Heckbert 调研了研究人员提出的各种近似前置滤波的方法【heckbert86】，发现一种有效的方法——椭圆加权平均(EWA)滤波。如图 9.55 中表明，这种方法把每个屏幕像素想像成被一个圆对称的滤波器函数覆盖。图中的同心圆表示不同的权重等级(就像地形图)，并把这个滤波器函数映射到纹理空间中。在纹理空间中，这些权重等级变成一系列的椭圆，并能够大致表示纹理四边形的形状。我们将滤波函数采样，并保存在查找表(LUT)中，记录椭圆内部的不同位置的权值，以此对椭圆内的纹理元素进行加权平均。这个过程可以完全通过增量方法实现，对每个像素只需要少量的计算，所以效率很高(利用了滤波器函数从一个空间到另一个空间的可控变换)。图 9.52(b)就是用 EWA 滤波器渲染的向后倾斜的棋盘图案。

另一种方法称为随机采样，在【Watt&Watt92】中有详细介绍。该方法避免了困难的几何计算，通过在纹理元素中随机采样求均值来计算纹理颜色。图 9.56 显示了与某个屏幕坐标相对应的以(s^*,t^*)为中心的纹理四边形。我们并不是用这个纹理四边形本身，而是在(s^*,t^*)周围的纹理元素中随机采集一定数量的样本，最终纹理颜色就是这一些采集的样本点颜色的均值：

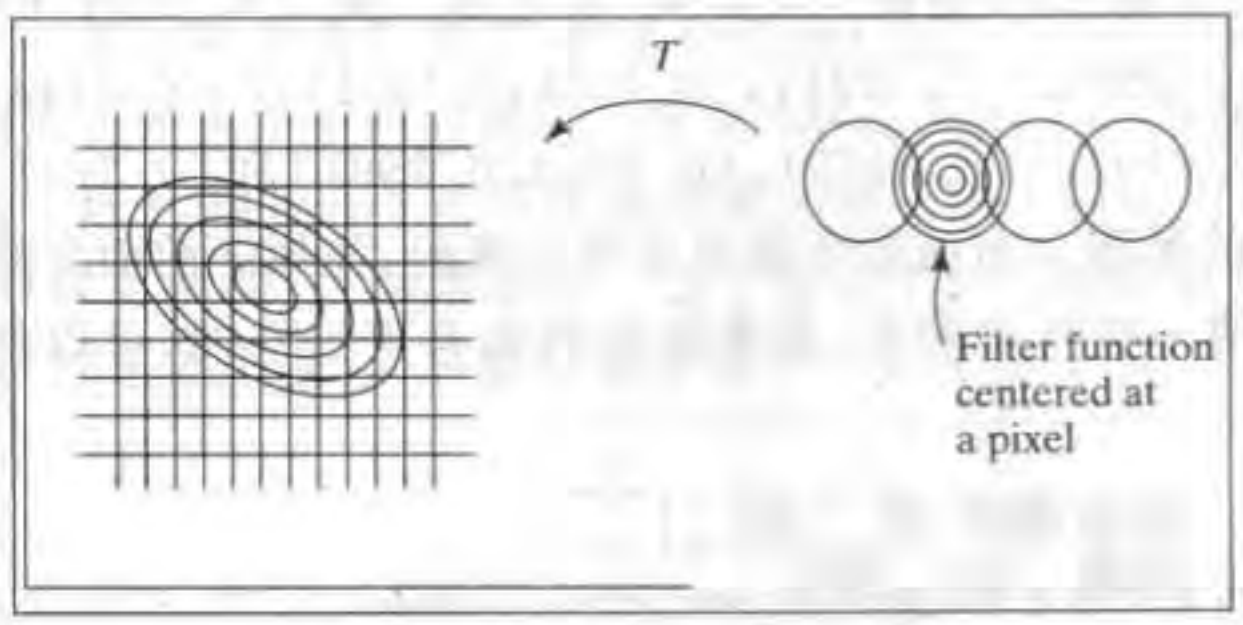


图 9.55　椭圆加权面积滤波

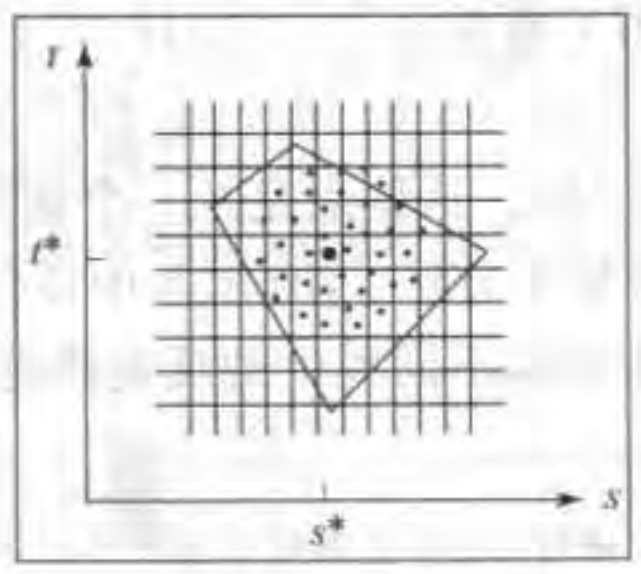


图 9.56　采用随机采样方法的反走样

$$\text{average} = \frac{1}{N}\sum_{k}\text{texture}(s^* + \alpha_k, t^* + \beta_k) \tag{9.11}$$

其中 α_k 和 β_k 为小的随机量,可以很容易的使用随机数生成器生成。它们的分布可以根据纹理四边形的大致尺寸进行调整。

在第 12 章中我们还会看到随机采样作为一个强大的工具应用于光线跟踪。

9.8.3 使用 OpenGL 反走样

OpenGL 提供了一些执行反走样操作的工具。最简单的方法是采用一个累积缓冲区(accumulation buffer),这是一个与帧缓冲区类似的附加存储区,OpenGL 可以生成累积缓冲区并对其进行绘制。类似于随机采样法,这种方法在略微不同视点的位置(相差只有一个像素的几分之一)多次绘制同一个场景,然后把结果加入累积缓冲区。当把所有这些位置经过扰动的绘制结果加入累积缓冲区后,把累积缓冲区的数据复制到帧缓冲区中,显示结果就是经过反走样处理的图像。这种方法生成的每个像素都是在投影后的场景中该像素邻域颜色的均值。

下面的代码显示了当使用摄像机对 3D 场景中拍照时反走样是如何实现的。首先在程序运行时创建累积缓冲区[①],并将其初始化为全 0(使用 glClear(GL_ACCUM_BUFFER_BIT))。然后把这个场景绘制 8 次,每次绘制之前把摄像机根据存储在数组 jitter[]中的位移量,在 x 方向和 y 方向上"平移"一个微小的距离(参见第 5 章和第 7 章)。使用 glAccum(GL_ACCUM, 1/8, 0)语句,把每次新绘制结果的数值乘以 1/8 后逐个像素地加到累积缓冲区中。绘制完第 8 次之后,使用 glAccum(GL_RETURN, 1.0)把累积缓冲区复制到帧缓冲区中。下面是完整的代码:

```
glClear(GL_ACCUM_BUFFER_BIT);                          //清空累积缓冲区
for (int i=0; i <8; i++)
{
    cam.slide(f * jitter[i].x, f * jitter[i].y, 0);   //平移摄像机
    display( );                                        //绘制场景
    glAccum(GL_ACCUM, 1/8, 0);                         //加入到累加缓冲区
}
glAccum(GL_RETURN, 1.0);                               //把累积缓冲区数据复制到帧缓冲区
```

jitter 向量中包含 8 个点,这些点的 x 值和 y 值在 $-0.5 \sim 0.5$ 之间。头文件 jitter.h[②]中使用了数值是(−0.3348,0.4353),(0.2864,−0.3934),(0.4594,0.1415),(−0.4144,−0.1928),(−0.1837,0.0821),(−0.0792,−0.3173),(0.1022,0.2991)和(0.1642,−0.0549)。这些点是从一个圆形对称的概率分布函数中随机选择出来的,类似于前面介绍过的 EWA 方法。jitter.h 中还包括其他一些抖动向量,有更短些的也有更长些的,可以用它们来尝试不同程度的反走样处理。

---

① 通过在初始化函数中加入 GL_ACCUM 参数来创建累积缓冲区:glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB | GLUT_ACCUM | GLUT_DEPTH)。

② 可以在 GLUT 软件包中获得。

图 9.57 给出了用两种方法渲染的一个 3D 场景：左图中的场景没有使用反走样技术；右图显示了在累积缓冲区中取 8 个场景“抖动(jittered)”绘制结果的均值得到的改进效果。可以看到经过反走样处理后的“锯齿”明显减少了。因为对于每一帧场景都要绘制 8 次，所以这种方法会降低性能。

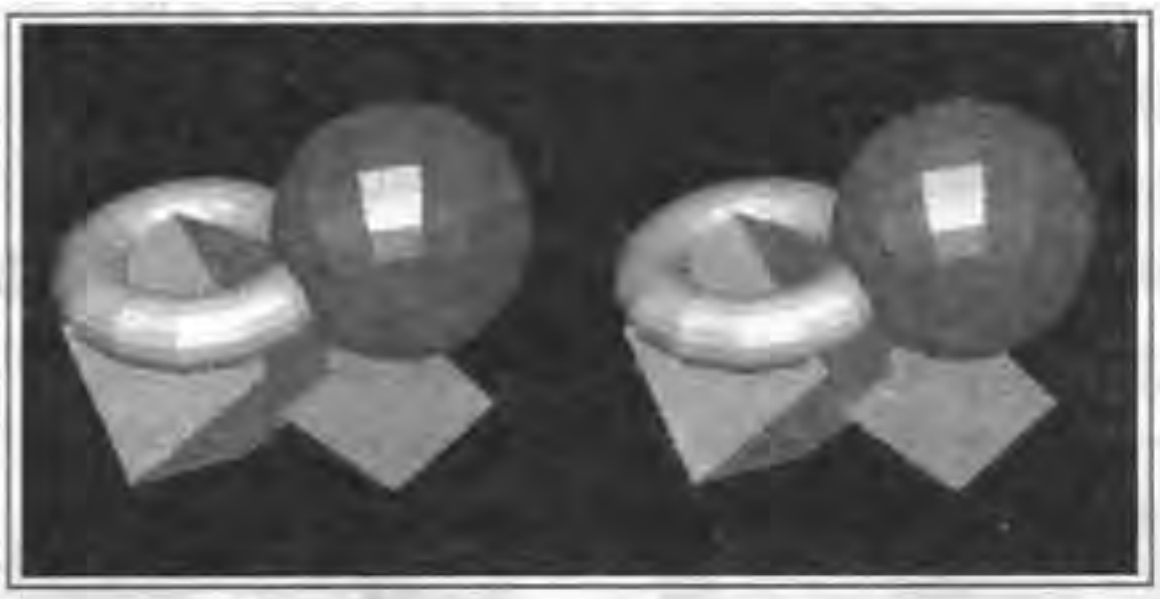


图 9.57　未采用和已采用反走样技术渲染的场景

9.9　生成更多的明暗效果和色彩

早期的图形系统中，显示设备是非常昂贵的，特别是那些支持真彩色的显示器。当然在当今，这些显示器不管是工作用途还是家用，支持真彩色(每个像素占 32 比特位)已是稀松平常，且价格也便宜了很多。因此，在 20 世纪 80 年代末，大量研究人员就研究出一些算法，使得每像素一比特的图像看上去具有每个像素多种明暗的效果。

本节我们会简要介绍其中的一两种方法，包括该问题的历史，以及人的感知系统与图像的交互方式。

一种方法称为半色调(halftone)，这种方法以空间分辨率换取色彩分辨率【Ulichney87，Knuth87】。采用了这种方法的一个熟悉的例子就是报纸。虽然只用了黑色的油墨，报纸的图像看上去却有很多灰度等级。这种效果是通过使用紧密排列的、大小不一的黑色油墨块获得的。多数墨块都较大时其覆盖的区域就显得颜色较深，因为这里深度的平均值更高一些。而墨块都较小时看起来就较浅。眼睛能够把墨块合并，来感受局部区域内的平均深度。然而，报纸图片的空间分辨率要比照片小得多，因为报纸图片是由独立的墨块组成，而这些墨块不可能达到任意小。

在计算机图形学中，数字半色调(digital halftone)或图案法(patterning)采用了点的阵列而不是大小不同的墨块。图 9.58(a)给出了一个例子，使用 2×2 的点阵列(每个点的值为 1 或 0)模拟有 5 个可能亮度级的块。为了讨论方便，我们采用打印业习惯，用 0 级表示白(没有油墨)，4 级表示黑。这个想法之所以能够实现，是因为眼睛看到的是每个 2×2“小块”的平均亮度，从而能够感受到 5 个亮度级。

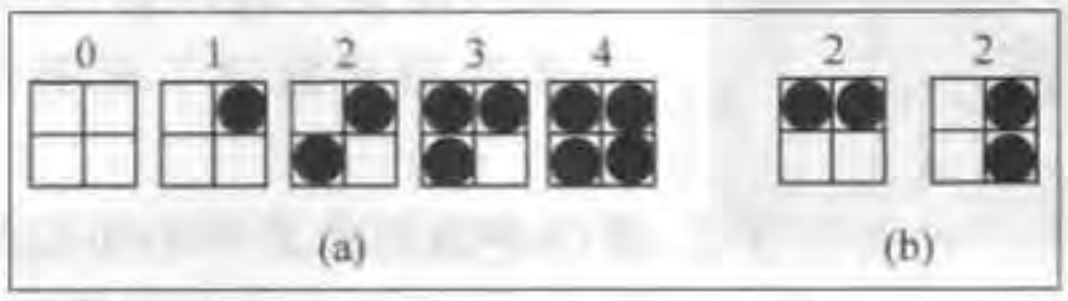


图 9.58　2×2 图案

为了对前面的讨论有直观的理解,考虑一个 100×100 像素的原始灰度图像,亮度值范围是从 0～4。而我们只有一台二值显示器。为了显示这个灰度图像的内容,我们可以使用 200×200 像素显示该图像,然后给每个 2×2 的像素小块设置合适的图案,用来表示灰度级 0,…,4。这又是一个用空间分辨率换取亮度分辨率的例子。

选取图 9.58(a)中黑色元素的位置时,要尽可能做到"不规则",从而避免图像中产生条纹或其他不自然的现象。相反的,如果只把图 9.58(b)中两种图案之一用于表示灰度级 2,那么在某些图案中,图像上可能会出现水平或垂直的条纹。

图 9.59 给出了两幅图:左侧是 256 级灰度图。右侧左图在二值显示器上使用 2×2 图案表示的结果。从图中可以清楚地看到 5 级灰度产生的效果。

图 9.59 使用 2×2 图案化示例

左图:256 级灰度显示没有图案化;右图:2×2 图案化图像在二值显示器上的结果

还可以使用更大的单元尺寸产生更多的灰度等级。图 9.60 显示了使用 3×3 单元的图案,这样能够获得 10 个灰度级。一般 $n \times n$ 个单元 0 和 1 的图案可以产生 n^2+1 种灰度等级(为什么?)。图案化方法在原始图像的分辨率低于所使用显示设备的分辨率时是最适合使用的。

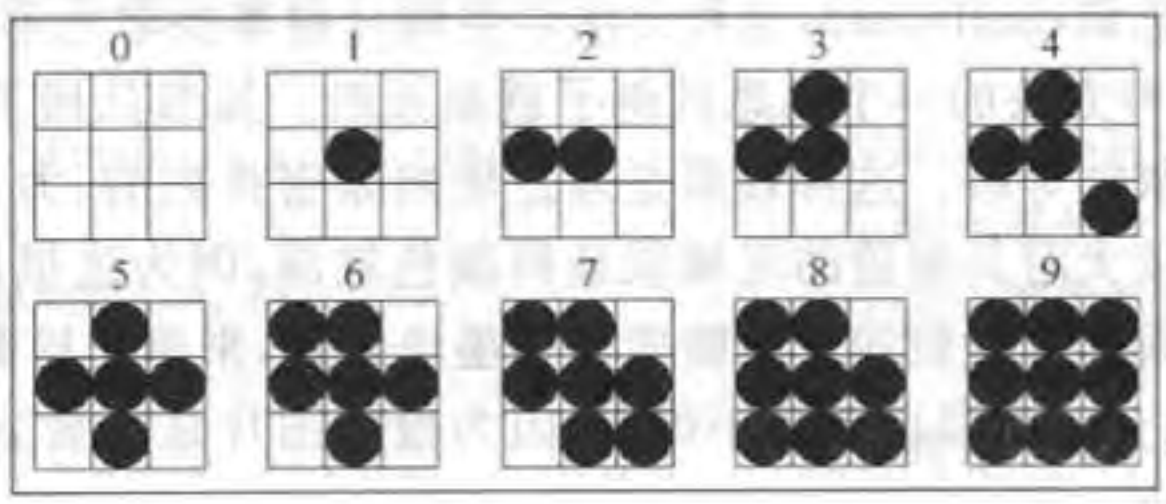


图 9.60 3×3 图案

请参阅【Knuth87】获取数字半色调的更多信息。

前面我们简要介绍了使眼睛看上去比实际更多灰度级的半色调技术。下面我们简单介绍另一种技术:**误差扩散**(error diffusion)

9.9.1 误差扩散

误差扩散提供了另外一种阈值方法,使得多级的像素映射可以显示在只支持较少颜色的显示设备上(虽然现在这种显示器已不多见)。这里再次假定原始像素映射中的每个像素

亮度范围为 0～255，要用适当的方法把每个像素替换为 0 或 1。

如果一个像素的亮度为 A，简单的阈值法就是，如果 $A<128$ 就将其替换为 0，如果 $A\geqslant 128$ 就替换为 1。只要 A 不是 0 或 255，这种方法会产生显示值相对于"真实"值的误差。例如，如果 $A=42$，就会把这个像素设置为 0，这个值比起 42 来太小了。如果 $A=167$，就会把这个像素设置为 1(最高亮度，对应像素值 255)，这个值又太大了，比真实值大了 $255-167=88$。如何处理这个误差呢?

误差扩散方法通过从像素映射中的某些相邻像素减去不可避免的误差来补偿这些误差。我们把产生的误差分成几份传送给还没有经过阈值化的相邻像素，当这些像素阈值化时，就使用调整以后的新值与阈值相比较。按照这种方式，误差在图像中"扩散"开来，以保持正确的平均亮度值。

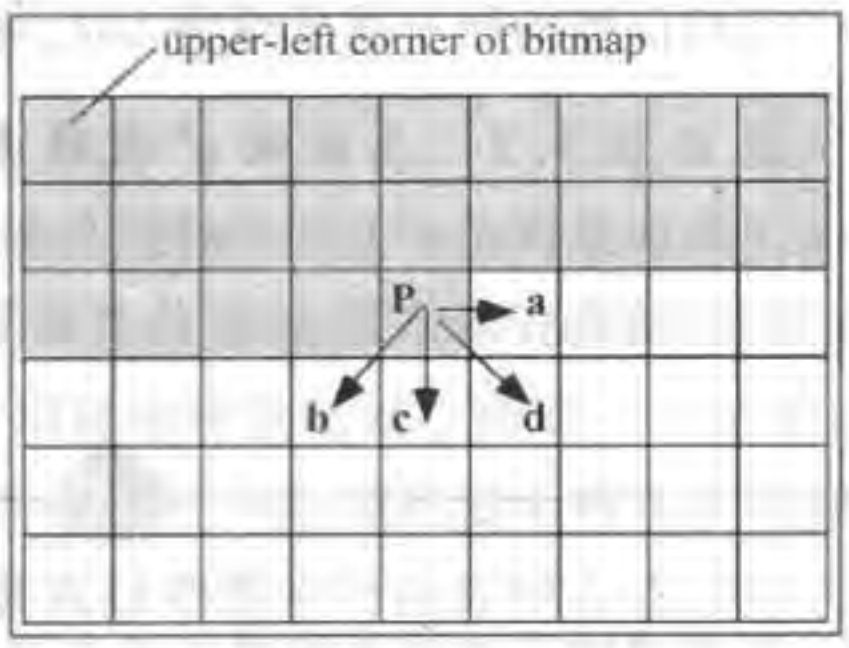


图 9.61　误差扩散过程

图 9.61 显示原(多级灰度)像素映射的一部分。我们假定仍使用通常的顺序从上到下，从左到右处理。假设灰色的像素是已经处理过的。像素 p 刚刚与 128 比较过，并设置为 0 或者 1。

设 A 为该点的像素值。如果 A 小于 128，则显示的像素值为 0，同时误差 E 是 $-A$(显示的亮度值太低了)。如果 A 大于等于 128，则显示的像素值为 1，这时误差 E 是 $255-A$(显示的亮度值高了 $255-A$)。

现在把二值化产生的误差分为几份传给像素 a、b、c 和 d(如图 9.61 所示)。这些像素的值变为：

$$\begin{aligned} a &= a - f_a E \quad \{adjust\ pixel\ to\ the\ right\} \\ b &= b - f_b E \quad \{adjust\ pixel\ to\ lower\ left\} \\ c &= c - f_c E \quad \{adjust\ pixel\ to\ below\} \\ d &= d - f_d E \quad \{adjust\ pixel\ to\ lower\ right\} \end{aligned} \tag{9.12}$$

其中 f_a，f_b，f_c，f_d 是常量。一种常用的比例选择是 $(f_a, f_b, f_c, f_d)=(7/16, 3/16, 5/16, 1/16)$。这些数值相加等于 1，也就是说产生的误差全部传递到相邻像素中，从而保持了局部的亮度均值不变。

当处理到扫描线最末端时，传递给 a 和 d 的误差就不要传送给下一条扫描线的开始(为什么?)，可以把它们简单忽略掉，也可以将所有误差传递给 b 和 c。经验表明最好在处理下一条扫描线时反方向处理，即第一条从左向右，第二条就从右向左处理。这种蛇形扫描的方式被称为蛇纹光栅模式。

图 9.62　经过误差扩散处理后的一幅像素映射

图 9.62 给出了一幅经过误差扩散的 512×512 像素映射在一台过时的显示器上显示的效果。这个误差扩散方法使用了蛇纹光栅和等式(9.12)中的系数。

将此技术扩展，使其支持两个以上颜色的显示设备是很容易的。对每个像素计算最接近

的显示灰度级,把生成的误差按照刚才介绍的方法传递出去。对于彩色图像,可以分别对三个颜色分量进行独立的误差扩散。

Knuth 将抖动技术和误差扩散技术合并为一种称为"点扩散"的方法,这种方法只比单独每种方法稍微复杂一点,但是能够产生更好的效果。算法的细节可以参看【Knuth87】。

练习

练习 9.9.1 手算

对一幅图像最上面一行扫描线执行误差扩散处理,这行扫描线上的所有像素值为 130,下一行扫描线的所有像素值为 132。讨论误差扩散后的结果。

练习 9.9.2 其他的扩散误差量

说明如果误差扩散处理时把全部误差都传送给右边像素得到的效果。再说明如果把一半误差传送给右边像素,一半传送给下面像素得到的效果。

本章研究了使用光栅显示设备显示图像时出现的问题和一些解决方法。光栅显示的基本性质是离散性,即显示出来的图像是以许多离散行和列方式排列的发光像素组成的,每个像素发出的颜色是几个离散的颜色值之一。图像在三个维度上都是离散的——水平方向、垂直方向、颜色值。但是由于像素点的排列非常紧密,而且(通常)有许多种颜色值,所以眼睛-大脑系统会融合临近的像素点而看到它们的"平均"颜色,因此能够把发光点的阵列融合为可以识别的图案。

这种融合使得光栅显示器具有一种基本属性,即可以使区域看上去像是用纯色或图案填充的,基于线绘制的设备如绘图仪很难达到这一点。我们讨论了几种填充区域的方法,特别关注了用多边形定义的区域的填充。

离散性既有好处也有坏处。空间离散性(水平的和竖直的)导致了走样现象的发生("锯齿"):一条对角的直线看起来是有锯齿的,这样观看起来会很不舒服。所以开发了很多反走样技术来改善走样的视觉效果。过分的颜色离散化——比如只有两种颜色值的显示设备——会产生不自然的"色带"或"色岛",而且丢失了原始图像中的许多信息。所以人们又研究出抖动技术,充分利用了人眼融合邻近点这个特点,使眼睛"看到"比实际更多的颜色。

光栅显示还有另外一个基本性质——用数值表示像素,而数值可以保存在内存中。光栅显示器可以看作是一个巨大内存区域对外开放的一个窗口,这个内存区域就是像素映射。显示系统将数字映射为颜色,使得像素映射可以被人们感知。计算机对数值操作的便捷特性打开了一扇通往新技术的大门。光标可以在鼠标的操纵下在屏幕上扫过,窗口可以卷动,图像可以在屏幕上内存和屏幕外内存之间来回切换等。

现代光栅显示器的特点就是可以处理非常多的像素,有时甚至数百万。这使图像能够产生丰富的细节,但是也增加了执行某些操作的时间。复制大量像素值通常要由通过特制的硬件来实现,比如 bitBLT 芯片。这也促使人们开发出了更高效的算法以改进性能。其中许多算法都利用了一致性这个贯穿于图形学中的概念。比如,横向跨度一致性

(span coherence)是指扫描线上许多相邻的像素常常具有同样的颜色；还有扫描线一致性(scan line coherence)，是指相邻扫描线上的像素常常是类似的。

本章讨论的各种技术都是在像素级上操作，以提高光栅显示器上的图像效果。这些技术或者是利用了离散性的优势，或者是弥补了离散性的缺点。因为一幅图像中的像素个数如此众多，所以对于每种算法来说效率都很重要。

案例分析

案例 9.1　读取和显示 BMP 图像文件

(难度系数：Ⅱ)编写一个应用程序，给出 RGBpixmap 类的方法 draw()、read()、copy()、readBmpFile()和 writeBmpFile()的细节，并允许用户读取存储在 BMP 文件中的图像，并利用 OpenGL()显示出来。另外程序还应允许用户用鼠标在显示器上指定一个矩形，并将该矩形内部的像素映射保存到 BMP 文件中。如果用户按下 f 键，就把显示的像素映射围绕其水平中心线反转，显示出上下颠倒的图像。按 v 键就围绕图像的垂直中心线反转。选择多个 BMP 图像进行实验。

案例 9.2　用 OpenGL 融合两幅图像

(难度系数：Ⅱ)9.3 节介绍过，很容易让一幅图像渐变为另一幅图像。图 9.12 中还给出了一个示例。如果这两幅图像分别存储在像素映射 A 和 B 中，那么只需对一系列 t 值绘制这两幅图像的加权平均 A(1－t)＋Bt 即可。

本案例使用 OpenGL 的 alpha 融合功能和图 9.15 中的 blend()方法形成加权平均，基本步骤如下：

a. 读取两个图像文件生成两个大小相同的像素映射 *A* 和 *B*

b. 设置适当的融合函数 glBlendFunc()

c. 清屏

d. 将 *t* 设为值分别为 0，0.2，0.4，0.6，0.8，1

e. 设置 *A* 的 alpha 值为 t

f. 绘制 *B*，完全不透明

g. 绘制 *A*，与背景融合

h. 暂停以观察融合的图像

需要为图像设置一个“全局”alpha 值，即把每个像素的 alpha 值设置为同一个值。这很容易通过在 RGBpixmap 类中添加一个方法 setAlpha(float alpha)实现：

1. 编写方法 setAlpha(float alpha)。只需遍历像素映射中的所有像素值，把其中每个像素的一个分量设置为最大值(255)的 alpha 倍。

2. 以示例 9.3.2 为范例，编写一个应用程序实现两幅图像 *A* 和 *B* 的渐变，使得按 d 键时开始从 *A* 到 *B* 的渐变，按 b 键时开始从 *B* 渐变到 *A*。通过几对图像验证这个程序。

案例 9.3 处理"shape"数据结构

(难度系数:Ⅲ)9.6.1 节介绍了一个用矩形集合表示区域的数据结构。本案例分析中,要求读者开发出实际处理这种结构的程序:

1. 生成一个 shape。编写例程:

```
void pixmap2shape(RGBpixmap& pixmap, shape& shape, Color3 color);
```

该例程从像素映射生成一个 shape 数据结构。这个 shape 结构精确地表示了这个像素映射内颜色为 color 的所有像素组成的所有区域。

2. 从 shape 中生成像素映射。这是前面操作的逆操作,生成一个像素映射,其中包含一个 shape 中描述的全部区域。

```
void pixmap2shape(shape& shape, RGBpixmap& pixmap, Color3 fore, Color3 back);
```

这个例程生产一个足够大的像素映射,使之可以覆盖 shape 描述的区域。这个像素映射用颜色 fore 填充所有 shape 内的像素,其余的所有像素设置为 back 颜色。

案例 9.4 一般的多边形填充

(难度系数:Ⅲ)编写例程:

```
short fillPoly(IntPolyArray& poly);
```

该例程填充多边形 poly。如果多边形是退化或病态的,或者填充过程不成功则返回-1,否则返回 0。按照 9.7 节提出的算法编写,并使用边表和活性边表来提高效率。

测试多边形可以简单的随机生成。用户输入多边形的顶点个数,顶点坐标由程序随机产生,并确保多边形闭合。多边形生成以后,便可进行填充。提高要求:使用图案填充多边形。

案例 9.5 误差扩散

(难度系数:Ⅱ)编写例程:

```
errorDiffuse(IntRect r);
```

该例程对矩形 r 内部的屏幕图像执行方程 9.2.2 所述的误差扩散算法。感兴趣的矩形由案例分析 9.1 中的方法确定。观察误差扩散前后的图像差别。该结果在仅支持少数亮度或颜色的显示器上观察效果会更明显。例程使用扇形模式对相邻扫描线进行扫描。通过几幅图像验证这个程序。

进一步阅读

Jim Blinn 在 Jim Blinn'Corner, Dirty Pixels【Blinn98】中针对本章的主题给出了一个十分通俗的论述,其中包括反走样、抖动和合成。Ulichney 的 Digital Halftoning【Ulichney87】针对抖动给出了一个透彻的论述,同时 Knuth 也对此问题提出了很广泛的看法【Knuth87】。"A New Polygon Based Algorithm for Filling Regions"【Lui00】是 Lui 等人发表的一个区域填充的新方法。OpenGL 红宝书也针对如何控制程序以生成本章中讨论的各种效果进行了详细阐述【Wu04】。

第10章 曲线和曲面设计

本章学习目标

- 开发用于表示和设计曲线的工具
- 描述常用曲线的重要数学性质，例如光滑性
- 研究贝塞尔曲线和B样条曲线的数学性质
- 开发用于设计贝塞尔、B样条和NURBS曲面的工具

预览

10.1节介绍曲线参数化表示的一些重要性质，提出用于度量这些曲线光滑性的方法。10.2节讨论用多项式和有理多项式来表示的曲线类型。

10.3节介绍交互式曲线设计的主要方法，用户通过鼠标交互指定一些控制顶点，利用曲线生成算法预览曲线，然后编辑控制顶点来改变曲线的形状。重点讨论插值控制顶点的曲线和逼近控制顶点的曲线之间的区别。

10.4节介绍贝塞尔曲线，10.5节讨论其性质，这些性质在计算机辅助设计(CAD)中得到广泛应用。10.6节讨论贝塞尔曲线的局限性并且寻找曲线设计的更好的方法，主要介绍分段多项式。从而引出样条函数的讨论。

10.7节介绍B样条，10.8节讨论它们的一些有用的性质。10.9节描述非均匀有理B样条曲线(NURBS)。10.10节讨论一些插值控制顶点的曲线的计算方法。

10.11节处理复杂曲面形状设计的问题，并且把第6章的方法推广到基于贝塞尔，B样条和NURBS曲线的曲面构造。主要涉及将两个曲面片无缝拼接的问题，并且用经典的茶壶模型作为混合曲面片的例子。另外还给出了一些其他曲面设计的例子。

案例展示了用于描述和绘制多种曲线和曲面的工程，此外还介绍了ElliptiPool游戏和绘制贝塞尔，B样条和NURBS曲线和曲面的工程。

10.1 概　　述

到目前为止，我们主要讨论了两种简单的几何物体：由点和直线组成的多点体(Polypoints)和由面集合组成的网格(回顾第6章)。我们想以一种有组织的方式来描述和表达出现在计算机图形学和计算机辅助设计程序中的更加丰富的形状类型。

某些形状会特别引起人们的兴趣，因为它们从审美角度看上去非常愉悦或者它们表示了自然界中一些实际存在的物体的外形。有些形状是专为某种功能需求需要仔细计算得

到,例如如何设计机翼的扫掠方式将给出飞机的最大提升力。还有其他的一些形状,例如汽车的挡泥板曲线,需根据其特定的功效,制造的简易性及用户实际需求来组合设计。

某些形状,例如对数螺旋和行星绕太阳运行的路径,有简明的数学公式使得它们易于分析,但是当我们想写个程序来绘制它们时,这些公式却帮助不大。因此我们需要从一种表示形式转换成另一种更适合于特定任务的表示形式的方法。其他的一些形状可直接由给定的数据来表示,而不是由数学表达式来表示。

10.1.1 作为运动轨迹的参数曲线

曲线参数化表示的重要应用源于描述物体随时间移动的轨迹。例如,当设计一个动画,场景中摄像机的轨迹需要在任意时刻确定。图 10.1 显示了一个在场景中移动的摄像机。它在 t 时刻位于位置 $P(t)$。设计者选择一个合适的函数 $P(t)$使得摄像机按要求移动,或许在 $t=0.1$ 时拍摄快照#1,在 $t=0.2$ 时拍摄快照#2,等等。摄像机的视线方向在任意时刻也都必须被指定(见练习)。

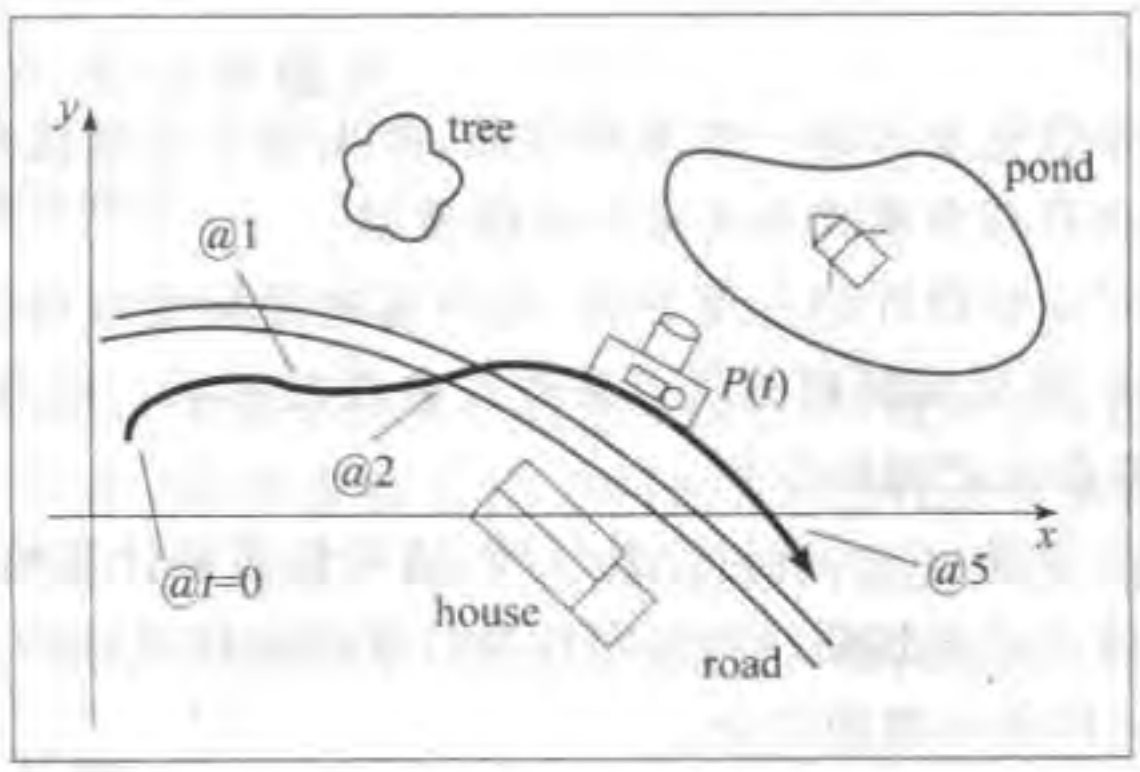


图 10.1 在动画中指定摄像机的位置

除了使得摄像机在指定的时间处于指定的位置,设计者必须确保摄像机沿着 $P(t)$前进时光滑地移动,当动画被回放时没有任何扰乱性的抖动。如下面我们所考虑的,这对摄像机的速度施加了某些条件。其他的物体也可能在动画中移动:在图 10.1 中汽车可能会沿着道路移动,小船可能会改变它的航线,一个人也可能从房子里出来。如在第 6 章中讨论的,人的胳膊和腿可能也会有它们自己的运动。所有这些物体的运动必须用一个适当的方法来描述。

通常,动画在 3D 场景中生成,其中摄像机沿着 3D 的路径运动且每个快照都是通过把场景投射到摄像机的胶卷上形成影像。

10.1.2 运动的光滑性

把参数 t 当成是时间的变量,当 t 增加时把点 $P(t)$看作是沿曲线运动。很自然要问在任意时刻点 $P(t)$沿曲线运动的速度是什么。速度 $\mathbf{v}(t)$是描述 $P(t)$经过曲线时的速率和方向的向量。通过初步的计算,对于 2D 中的曲线有

$$\mathbf{v}(t)=\frac{dP(t)}{dt}=\left(\frac{dx(t)}{dt},\frac{dy(t)}{dt}\right) \tag{10.1}$$

例如，由 $P(t)=(W\cos(t),H\sin(t))$ 给出的椭圆在 t 时刻的速度由 $\mathbf{v}(t)=(-W\sin(t),H\cos(t))$ 给出。$\mathbf{v}(t)$ 的大小通常被称为在 t 时刻的速率。图 10.2 显示了在椭圆上一些点的速度。当 t 改变时，大小和方向都会改变。（哪个位置的速率最大?）

曲线 $P(t)$ 在 $t=t_0$ 时刻的切线可以由含有参数 u 的参数形式 $L(u)$ 给出，如下所示。在 $u=t_0$ 时它明显通过 $P(t_0)$ 且在方向 $\mathbf{v}(t_0)$ 上运动。因此它由下式给出

$$L(u)=P(t_0)+\mathbf{v}(t_0)u \tag{10.2}$$

这些公式使得在各种应用中计算和画出曲线的切线变得很直观。

曲线的**法线**方向也可以在每个点处计算。它在所关心的点处被定义为垂直于切线的方向。因此如上在时刻 t_0 有切线方向 $v(t_0)$，法向方向是第 4 章中向量 v 的任意倍数。用 K 表示常量倍数，法向方向为：

（注意：倍数 K 符号的选择将会区分开**内向法向**和外向法向，在实际分析中根据需要改变 K 的符号。）

$$\mathbf{n}(t_0)=\mathbf{v}^{\perp}(t_0)=(-dy/dt,dx/dt)\mid t=t_0 \tag{10.3}$$

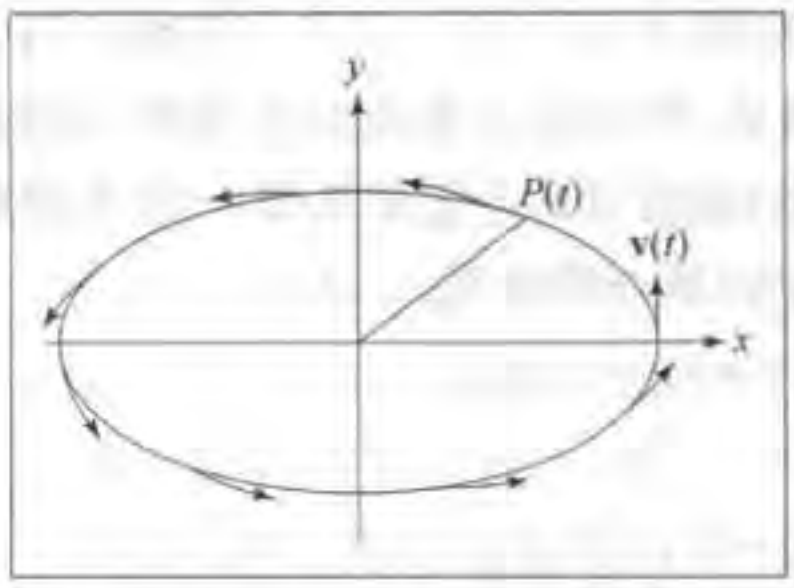


图 10.2　一条曲线上 $P(t)$ 的速度

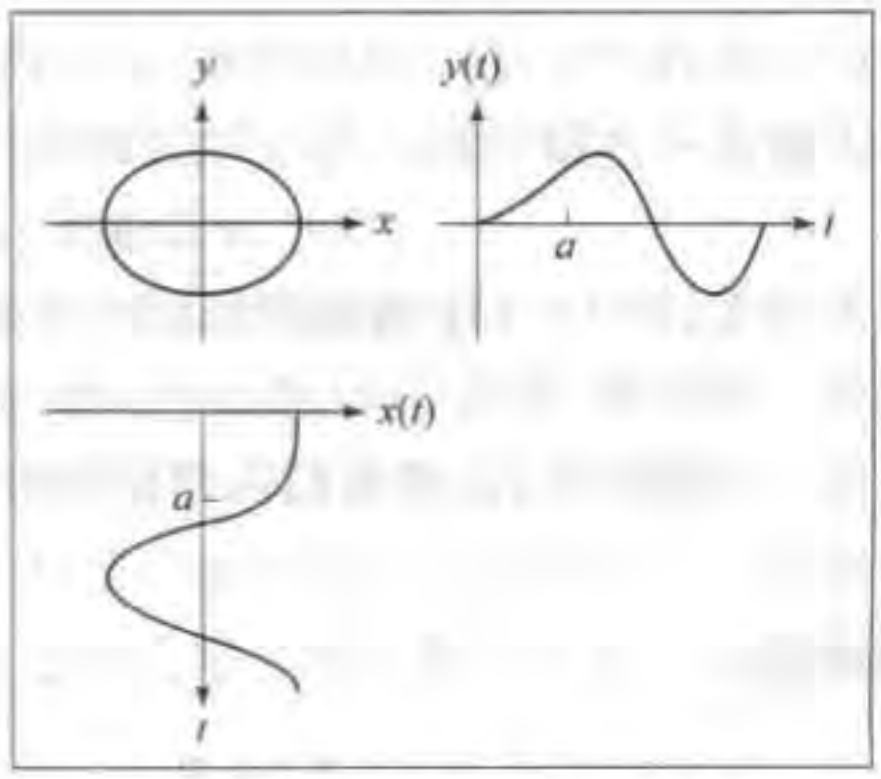


图 10.3　运动速度突然改变的轨线

因此上述示例的等式(10.1)的椭圆，作为例子，其法向向量为 $(-H\cos(t),-W\sin(t))$ 或其任意倍数。这是一个内向法向：乘以 -1 可以得到外向法向。在圆的特殊例子中，$\mathbf{n}(t_0)$ 是 $P(t_0)$ 自身的倍数，因此法向与从 0 到点 $P(t_0)$ 的径向向量方向相同（注意对于椭圆不是这样）。

当曲线 $P(t_0)$ 被用来描述一个物体例如像摄像机如何实时移动时，问题产生：物体沿着轨线是颠簸地移动还是光滑地移动？沿着给定的路线物体可以无穷种方式移动，开始，停止，加速，暂时在一部分曲线的折回，等等。如果我们只绘制曲线，则运动无关紧要：所有的轨线具有相同的图像。但是用参数函数来表示摄像机的运动，情况就会变得非常复杂：沿着轨线运动的细节被记录在一系列得到的快照中。突然的加速或跳跃通常是不希望出现的。

作为例子，考虑一个沿着椭圆轨线移动的摄像机，但有一个替换的参数表示：在 $t=a$ 时刻运动的速度突然被加速 3 倍：

$$P(t)=(x(t),y(t))=\begin{cases}(W\cos(t),H\sin(t)) & 0<t<a\\ (W\cos(3t-2a),H\sin(3t-2a)) & a<t<\dfrac{2(\pi+a)}{3}\end{cases} \tag{10.4}$$

图 10.3 展示了这种情况。轨线依然是椭圆，但是在 $t=a$ 时刻 $x(t)$ 和 $y(t)$ 开始以 3 倍的速度加速变动，所以导数在 $t=a$ 时刻有个不连续点。在这个时刻，时间轴好像以 3 倍比

例被压缩了。

参数曲线的形状不受这个跳跃影响,但是沿曲线点 $P(t)$ 的运动的本质在 $t=a$ 时刻前后是有显著不同的。

练习

练习 10.1.1 沿着椭圆上点的速度公式

根据等式(10.4)中的曲线给出在 a 时刻前后的一阶导数公式 $P'(t)=(x'(t),y'(t))$。速度相同吗?速率相同吗?

$$P'(t)=(x'(t),y'(t)) \tag{10.5}$$

当我们研究曲线和摄像机运动的光滑性,很容易想到描述曲线的不同种类的光滑性。我们将对两种光滑性有兴趣,它们分别被称为**参数连续性**和**几何连续性**。

参数连续性

我们说曲线 $P(t)$ 在 t 时间间隔 $[a,b]$ 内处处 k 一阶参数连续如果在 $[a,b]$ 内曲线所有点有直到 k 阶的一连续导数。为了简洁的表达,我们说:

$$P() \text{ 在}[a,b]\text{ 内 } k \text{ 阶光滑} \tag{10.6}$$

作为例子,图 10.4 中的椭圆处处 0 阶光滑,因为函数 $P(t)$ 自身是处处连续的。但是它不是处处一阶光滑,因为 $P'(t)$ 在 $t=a$ 有一个不连续点(除了 $t=a$ 它是处处一阶光滑的)。为了避免抖动的动画,通常我们强调摄像机的运动必需的是一阶光滑。

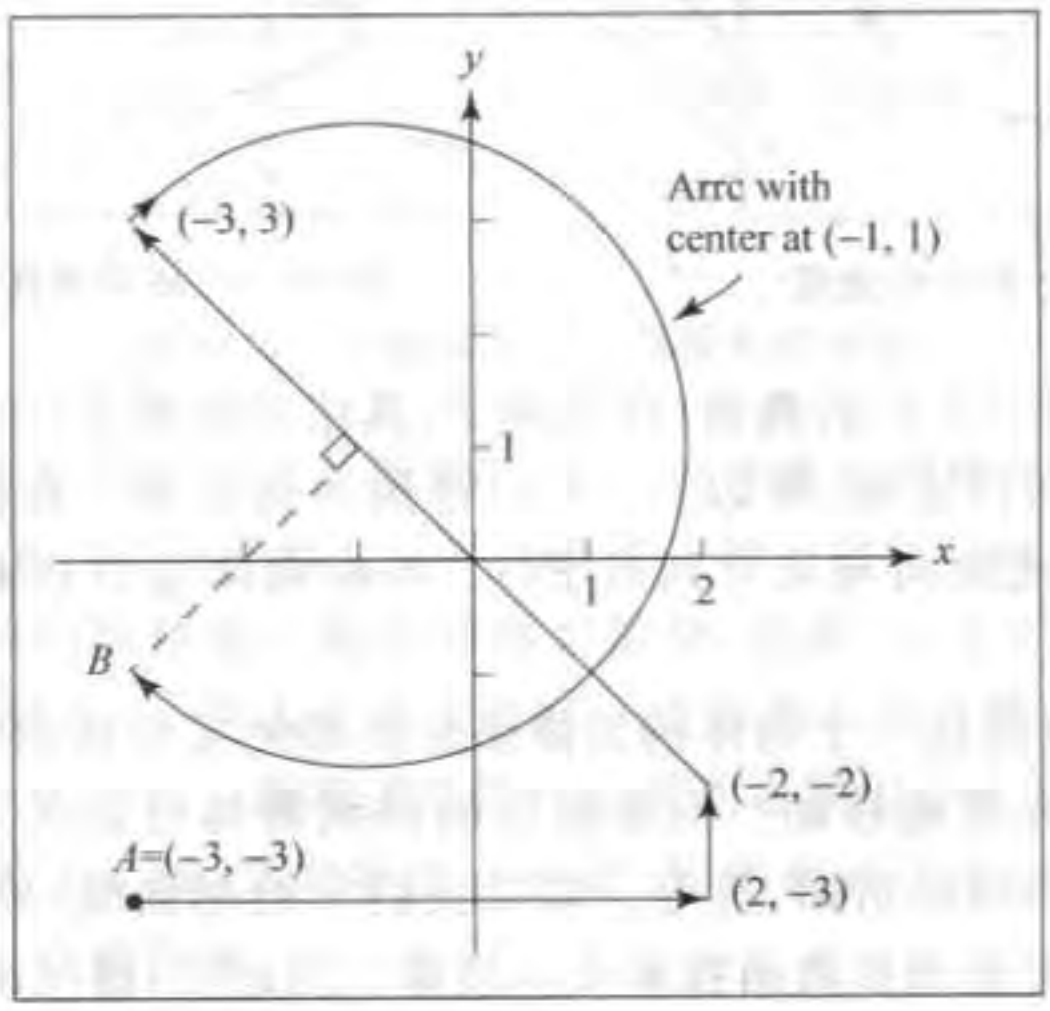


图 10.4 什么是参数表示?

有连续的速度和加速度的曲线是二阶光滑的。注意 $[a,b]$ 上二阶光滑函数必定是一阶和 0-阶光滑的。但一阶光滑函数未必是二阶光滑的。

回顾在第 3 章中讨论的三种类型的螺线(阿基米德螺线、对数螺线和等角螺线)。我们需要研究这些曲线的光滑阶。例如,对于对数螺线,其参数形式为 $f(\theta)=Ke^{a\theta}$,显然为 0-阶光滑的,且其速度向量为:

$$f'(\theta)=(-Ke^{a\theta}\sin(\theta)+Ke^{a\theta}\cos(\theta)ae^{a\theta}$$

$$f'(\theta) = Ke^{a\theta}\cos(\theta) + Ke^{a\theta}\sin(\theta)ae^{a\theta})$$

因为上述表达式中所有的项都是连续的，所以对数螺旋是一阶光滑的。图 10.5 显示了对数螺线的图像，其外形很像鹦鹉螺的外壳轮廓线。

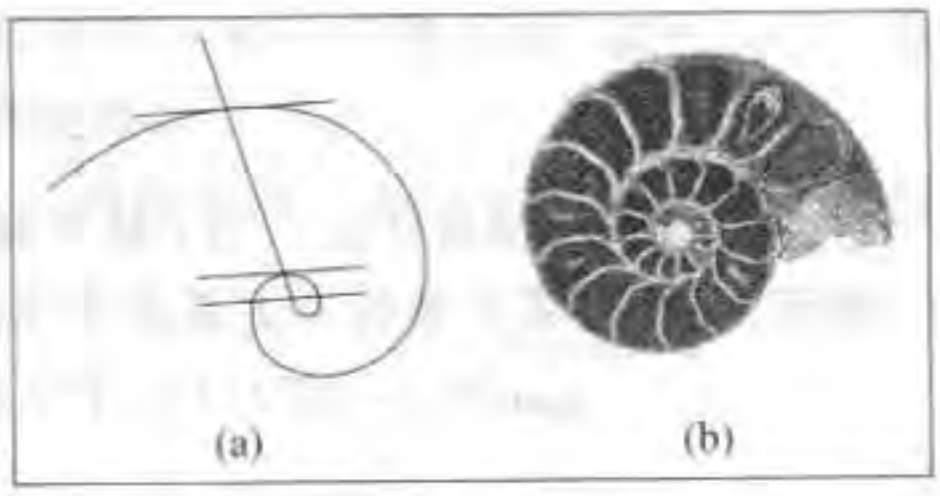


图 10.5　对数螺线和鹦鹉螺

在这节中，我们基于曲线的视觉光滑性而非沿着曲线的运动光滑性，讨论另一种光滑性的不同次数。

- 如果曲线在一个区间是连续的，我们称它为 0 阶光滑。
- 如果曲线在一个区间存在一阶连续导数，我们称它为一阶光滑。
- 如果曲线在一个区间存在一阶及二阶连续导数，我们称它为二阶光滑。
- 如果曲线在一个区间存在一阶、二阶及三阶连续导数，我们称它为三阶光滑。

几何连续性(G^k 连续)

一般地，曲线的几何连续性要求曲线的各阶导数向量的方向是连续的，即使它们在速率上有不连续点。等式(10.4)中椭圆除了在 $t=a$ 之外有连续导数。

等式(10.4)中的例子还表明，正是这个奇怪的参数化导致曲线的这种类型的光滑性得不到满足(在 $t=a$)。这是容易理解的。为了讨论这种连续性，我们需要如下的一些定义。

- G^0Z 连续和 0 阶光滑相同：它仅仅表示 $P(t)$ 关于 t 在区间$[a,b]$上连续。
- 在$[a,b]$上 G^1 连续表示对于区间$[a,b]$上每个 c 有 $P'(c-)=kP'(c+)$其中 k 为常数。因此在此区间上任一点(c)的速度向量的长度可能以某个数量 k 的倍数跳跃变化，但是其方向是连续的。
- 在$[a,b]$上 G^2 连续表示一阶和二阶导数有连续方向：对区间$[a,b]$上任一点 c 有 $P'(c-)=kP'(c+)$和 $P''(c-)=mP''(c+)$，其中 k 和 m 为常数。

注意：从这些定义中可知，曲线的几何连续性实际上是指曲线的速率或速度向量的长度(和它们的导数)是连续的。

练习

练习 10.1.2　绘制椭圆的切线

写一个程序画出在参数值 t 由图 10.3 生成的切于椭圆的短线段。

练习 10.1.3　圆锥曲线的切线和法线

在任意值 t 处计算抛物线和双曲线的切线和法向量的表达式。

练习 10.1.4　求参数表示

求图 10.4 所示曲线的参数表示。曲线在 $t=0$ 时刻始于点 A 且在所示的整个形状上以常速率移动，在 $t=20$ 时刻到达点 B。

练习 10.1.5　圆的另一种参数表示

除了由时间变换产生的表示的多样性，某些曲线允许在特征上有很大差别的简单表示，

例如下面这个。证明下列形式在 t 从 0 变化到无穷时生成圆的一部分。

$$x(t) = a\frac{1-t}{1+t}$$

$$y(t) = 2a\frac{\sqrt{t}}{1+t} \tag{10.7}$$

它的圆心和半径是什么？当 t 从 0 到 1 变化时生成圆的哪部分？

提示：回顾对某个参数 b 关系式 $T=\tan(b/2)$。然后

$$\sin(b) = 2T/(1+T^2) \text{ 和 } \cos(b) = (1-T^2)/(1+T^2)。 \tag{10.8}$$

10.2 用多项式描述曲线

多项式是基本的数学对象且在计算机图形学中经常使用，是因为它们能有效计算且具有很好的性质。在本章的剩余部分我们将关注多项式的特殊形式：这里我们研究由简单多项式构成的隐式形式和参数形式的相互关系。

首先介绍一个概念：一个 t 的 **L 次多项式**是一个下式给出的函数

$$a_0 + a_1 t + a_2 t^2 + \cdots + a_L t^L \tag{10.9}$$

其中常数 $a_0, a_1, \cdots a_L$ 是它的**系数**，每个都与 t 的某个次幂对应。多项式的**次数**是当 t 升幂时的最高次幂的指数。为此要成为 L 次多项式，我们强调 a_L 不等于 0。**阶**是多项式中的系数个数(这里是 $L+1$)，它比次数大 1。

一次多项式曲线

我们已经研究过线性多项式，知道 $x(t)$ 和 $y(t)$ 的线性参数形式生成一条直线，并且对应的隐式形式对于 x 和 y 是线性的。例如，参数表示为 $P(t)=a_0+a_1 t$ 的曲线是一条在时刻 0 通过 a_0 且在时刻 $t=1$ 通过 a_0+a_1 的直线。注意关于 $P(t)$ 的式子实际上是两个式子，一个是 $x(t)$ 另一个是 $y(t)$。当然，在 3D 中会有第三个式子是 $z(t)$。

二次多项式曲线

我们自然要问哪些曲线形状可以用二次多项式 $x(t)$ 和 $y(t)$ 表示：

$$x(t) = at^2 + 2bt + c$$

$$y(t) = dt^2 + 2et + f \tag{10.10}$$

其中 a、b，等等是常数。答案是简单的：对于任意选择的常数 $a, b, \cdots, f$，这条曲线总是一条抛物线。因此没有办法用这个形式来表示一条椭圆或双曲线。可以试着画一下如下两种情况时的抛物线的形状：

- 情况 #1：$a=b=0, c=d=-2$，和 $e=f=-6$
- 情况 #2：$a=b=3, c=d=8$，和 $e=f=4$

二次隐式形式

通过另一种方式，我们来看二次隐式形式，也就是，关于 x 和 y 的二次多项式。回想解

析几何中一般的二次隐式形式为：

$$F(x,y) = Ax^2 + Cy^2 + Dx + Ey + F \tag{10.11}$$

（注：某些作者写这个二次隐式形式时可能会有关于 x 和 y 的乘积项，如 $+Bxy$。事实上，通过一个适当的坐标轴的旋转变换，这项可以被去掉。）

其中 $A,C,\cdots,F$ 为常数。假定 A 和 C 不全为 0，否则生成一条退化曲线。同样回想起 $F(x,y)=0$ 描述的曲线形状是一个圆锥曲线。所表示的特殊的圆锥曲线可以通过检查系数 A 和 C 的符号来简单地确定，如下：

如果 $AC > 0$，它是一个椭圆　(10.12)

如果 $AC = 0$，它是一个抛物线

如果 $AC < 0$，它是一个双曲线　(10.13)

因此如果 A 或 C 有一个为 0 但不都为 0 时它是抛物线，如果 A 和 C 有相同符号它是椭圆，且如果 A 和 C 有相反符号它是双曲线。

它描述的圆锥曲线依赖于离心率[①]的值，ε，如图 10.6 所示。

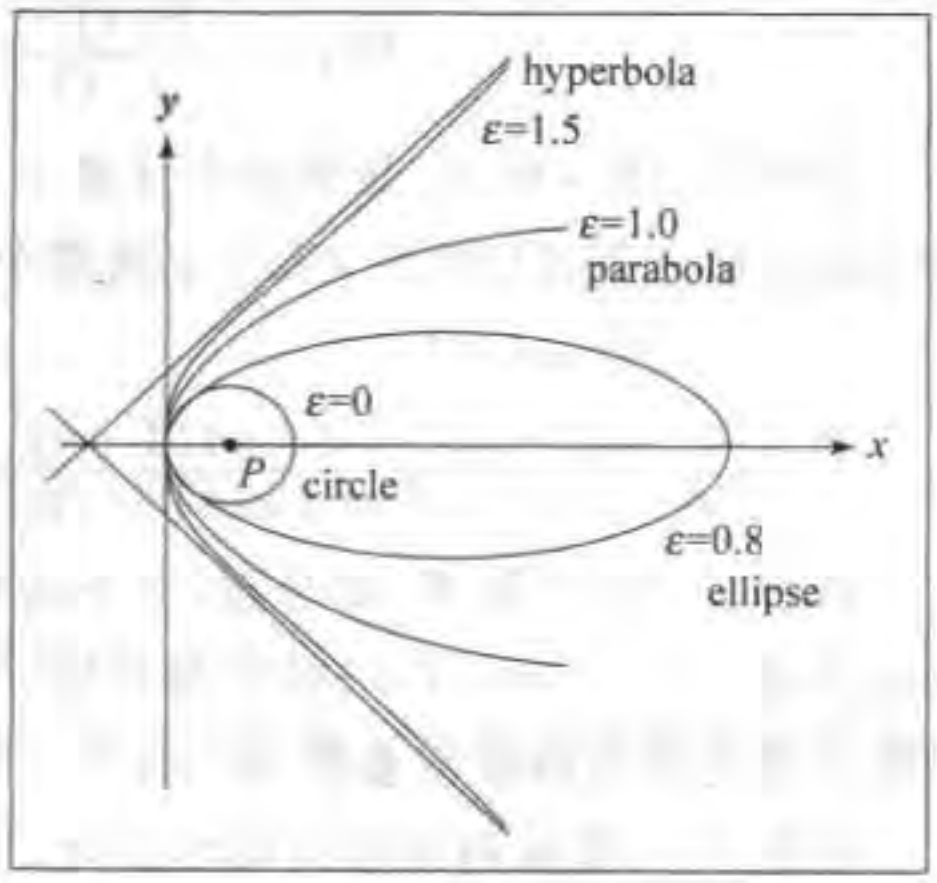


图 10.6　具有不同离心率的二次曲线

练习探究了为了画这批曲线的可能的参数化选择。注意在抛物线之外，我们无法为圆锥曲线找到多项式参数化形式。为此我们必须等到下一小节和**有理**多项式的介绍。

三次多项式曲线和更高次多项式曲线

参数形式到隐式形式与隐式形式到参数形式

如果我们把自己限制到由一次和二次多项式参数化的曲线，我们看到这种情形是很好理解的。然而，当多项式是更高次时事情变得比较复杂。在早期的曲线设计研究中，Sederberg 等【Sederberg85】证明对于给定的多项式函数 $x(t)$ 和 $y(t)$ 总有可能找到其隐式形式，但是只有对一次或二次的隐式形式才总能找到其参数形式。

本章中关于贝塞尔和 B 样条曲线我们主要的工具是三次多项式，并且我们将看到它为曲线设计提供了一个强有力的方法。

但是这些方法将不会从一个隐式形式出发且试图寻找其参数化形式，而是从由设计者设定的控制顶点集合开始且应用某个特定的算法沿曲线产生一些点。因此设计者如果有必要可以编辑控制点的位置且重新查看曲线。在许多方式中，与纯粹的数学方法相比，这是一种自然的曲线设计方法，因为它比较形象，当程序生成形状时允许设计者查看曲线设计的进程。

在结束当前对曲线设计的解析方法研究之前，我们先简要地预览将在后面研究的**有理多项式**函数类（如我们将要看到的，它们是 NURBS 的基）。下面对有理多项式的简要介绍将说明一些点的定义如何确定一个曲线的形状。它还产生了一个重要的结果：圆锥曲线可以用二次多项式的比精确表示。

---

① 作为感兴趣的一点，地球轨道的离心率是 0.0167，冥王星的是 0.25。

有理参数形式

考虑参数化,其中 $x(\cdot)$ 和 $y(\cdot)$ 被分别定义为两个多项式的比。线性情形在练习中探究过。等式(10.14)显示了由两个二次多项式的比组成的参数形式。我们看这里的二次多项式的情形且关注特殊的参数化形式:

$$P(t)=\frac{P_0(1-t)^2+2wP_1t(1-t)+P_2t^2}{(1-t)^2+2wt(1-t)+t^2} \tag{10.14}$$

其中 P_0,P_1,和 P_2 是平面中任意三点。它们在本书中被称为**控制顶点**,因为它们控制曲线的形状。等式(10.14)实际上是两个式子,是如下曲线的简化形式:

$$\begin{aligned}&(x(t),y(t))\\&=\left(\frac{x_0(1-t)^2+2x_1wt(1-t)+x_2t^2}{(1-t)^2+2wt(1-t)+t^2},\frac{y_0(1-t)^2+2y_1wt(1-t)+y_2t^2}{(1-t)^2+2wt(1-t)+t^2}\right)\end{aligned} \tag{10.15}$$

其中 x_0 和 y_0 是 P_0 的分量,对于另外两个点类似。分子中二次多项式的系数是控制顶点的分量。$x(\cdot)$ 和 $y(\cdot)$ 的分母相同。它们也是二次的但不依赖于控制顶点。然而,它们依赖于将要讨论的**权重参数**,即 w。

注意 $P(t)$ 是控制顶点的线性组合。如我们在 4.5.2 节中所看到的,为了使 $P(t)$ 成为一个有意义的点,它必须是这些点的仿射组合。

注意如果我们在等式(10.15)取 $t=0$,右端项简单地趋于 (x_0,y_0),因此这个曲线通过称为点 P_0。类似的,它在 $t=1$ 通过点 P_2。对于中间的 t 值 $P(t)$ 以一种复杂的方式依赖于所有三点。

图 10.7(a)图示了三个示例的控制点以及曲线当 t 从 0 增加时如何从 P_0 形成和当 t 到达 1 时如何在 P_2 点结束。图中询问中间的形状是什么样的?图 10.7(b)说明了结果:曲线是二次曲线段的一种且其类型依赖于 w 的值:

- 如果 $w<1$,它是一个椭圆。
- 如果 $w=1$,它是一个抛物线。
- 如果 $w>1$,它是一个双曲线。 (10.16)

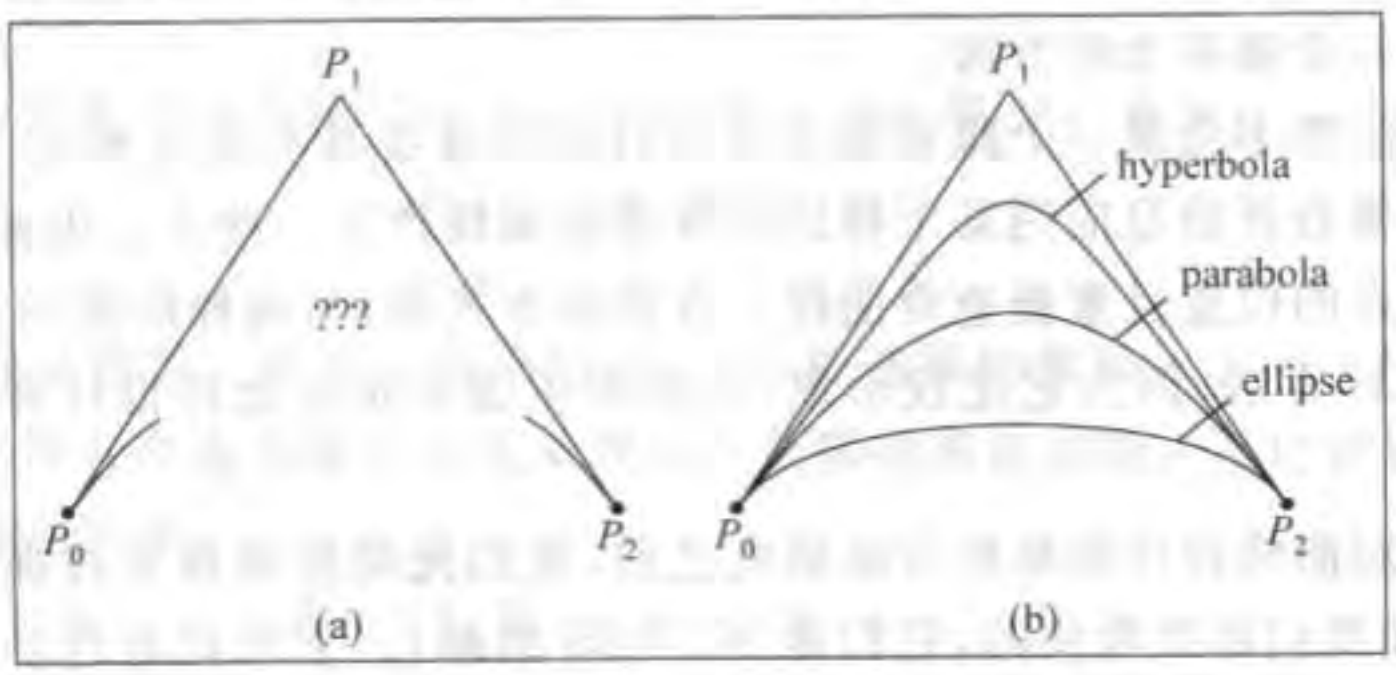


图 10.7 利用二次有理多项式生成圆锥曲线

因此我们有个参数化的生成圆锥曲线的方法,不用求助于之前使用的三角函数。练习说明了怎样生成这些曲线的另一半和怎样用这个参数形式生成圆。

练习

练习 10.2.1　退化的二次方程式

给出等式(10.11)中系数 $A,B,\cdots,F$ 的例子使得曲线 $F(x,y)=0$ 是：

(a) 不存在(例如，没有点满足 $F(x,y)=0$)；

(b) 单个点；

(c) 一条直线；

(d) 两条平行线。

练习 10.2.2　线性有理参数化

讨论由下列参数形式生成的曲线类：

$$x(t)=\frac{a+bt}{e+ft}$$

$$y(t)=\frac{c+dt}{g+ht} \tag{10.17}$$

其中 $a,b,\cdots,h$ 为常数。能否生成非直的曲线？给出一些例子说明这类曲线有多丰富。

练习 10.2.3　是直的吗？

从图 10.7 中看出如果 P_0、P_1 和 P_2 位于一条直线上，整个曲线应该是一条直线。证明这是否是真的。

练习 10.2.4　用有理二次多项式画圆锥曲线

图 10.8(a)显示了一个内切于等边三角形的圆。圆的三分之一可以利用等式(10.14)的参数表示基于所示的点 P_0，P_1，和 P_2 来绘制。因此整个圆可以被当作三个圆弧来画，每段用三个点来生成。可以证明【Farin90】对于圆要用的适当的权重 w 为 $w=\cos(a)$，其中 a 是角 $P_2P_0P_1$。对于等边三角形这个角是 60°。

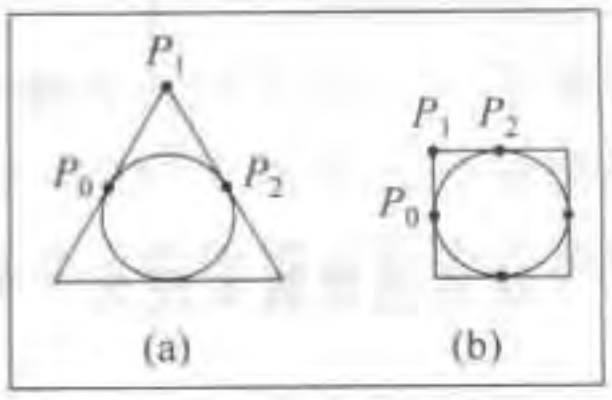


图 10.8　圆可看作三段或四段圆弧

(a) 对于五个 t 值画出等式(10.7)中点 $P(t)$，使得这些点沿圆弧从 P_0 到 P_2 接近均匀分布。

(b) 这与用($\cos(\cdot)$，$\sin(\cdot)$)的样本点画圆相比是否是更有效的方式？讨论。

(c) 对于 10.8(b)中所示四个圆弧重复上述(a)和(b)的工作，在这种情况下角 $P_2P_0P_1$ 是什么？

10.3　关于交互式曲线设计

目前为止所讨论的曲线是基于相对简单的数学公式。现在我们想扩大任务，研究适合于某个特定目标的更加复杂的曲线。特别地，我们希望开发一些允许设计者通过简单地指定控制顶点的小集合来生成大量形状的工具。

例如，假定设计者希望生成图 10.9(a)中曲线的计算机表示。这条曲线可能是汽车挡

泥板,涡轮机叶片或电钻外壳设计的一部分。或者它也可能是正在拍摄快照的摄像机从场景中掠过的轨线。目的是以某种形式捕获曲线的形状允许它任意重建,按要求调整形状和大小,将这些信息发送给机器用于自动切割和造型,等等。

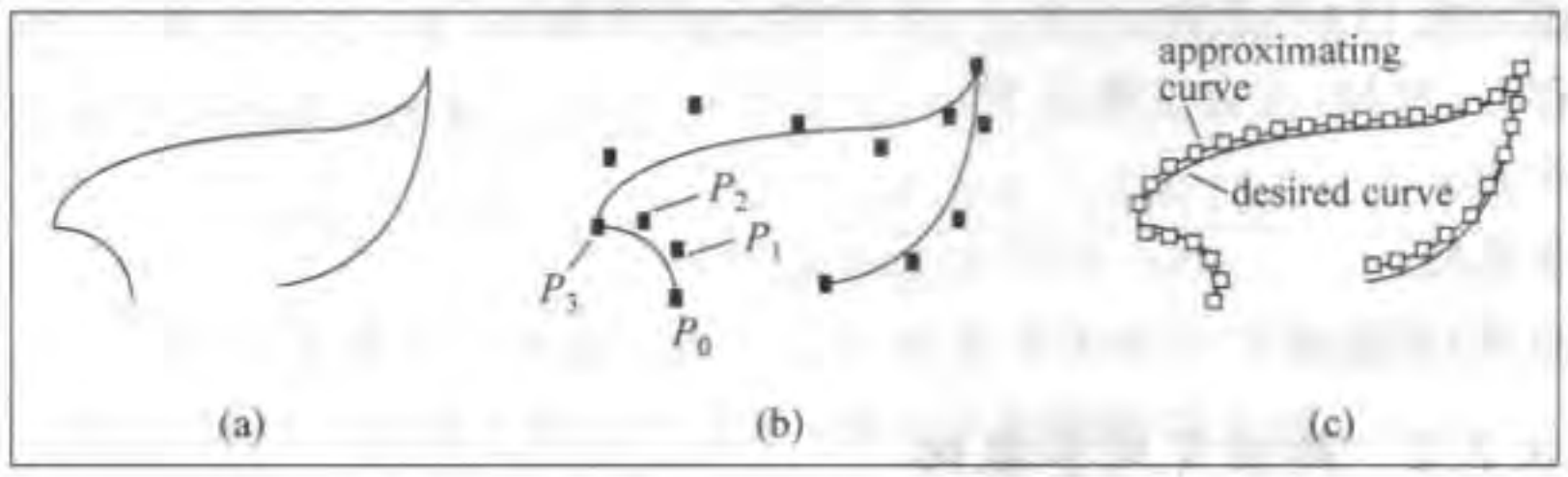


图 10.9 一个曲线设计流程

为了输入曲线,设计者把曲线的草图录制到绘图板上,然后沿着曲线移动光标在一组接近曲线的**控制顶点** $P_0, P_1, \cdots$ 处点击,如图 10.9(b)所示。控制顶点序列通常被叫做**控制多边形**[①]。设计者根据丰富的经验和对曲线生成算法特征的深入理解来输入控制顶点-这个算法将在后面用来根据数据点重新生成曲线。

如图 10.10 所示,算法的作用是对于任意给定的 t 值生成点 $P(t)$。算法的输入数据是一起确定点 $P(t)$所在曲线的控制顶点集合。

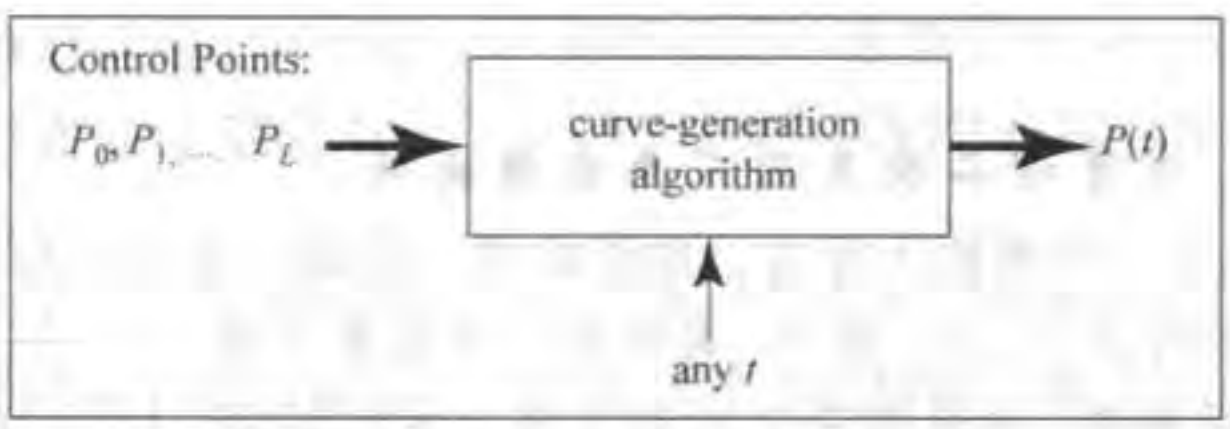


图 10.10 曲线生成算法

算法通常被实现为一个函数

```
Point2 CurvePt(double t, RealPointArray pts)
```

对于某个区间内任意 t 值返回一个点。为了绘制这条曲线用户可以选择一系列 t 值,在每个 t 值处计算 CurvePt(),然后用线段把这些点连接起来形成一个多边形折线。图 10.9(c)显示了一些小正方形的集合,表示给定图 10.9(b)中控制顶点序列由一个合理的算法可能返回的点。

由这些点所定义的折线很有希望能逼近当设计者输入控制顶点时头脑中的原始曲线。如果生成的曲线不能对原始曲线提供合适的近似,设计者将编辑控制顶点,大概用鼠标以这种和那种方式移动控制顶点且通过多次调用 CurvePt()来重新生成曲线。这个迭代过程继续进行直到设计者满意为止。因此交互设计由以下步骤组成:

(1) 确定初始控制顶点。

(2) 使用算法生成曲线。

---

① 控制顶点序列通常指定一个多边形折线形而非一个封闭的多边形,但是我们将继续使用"多边形"

(3) 如果对曲线满意,停止。

(4) 调整某些控制顶点。

(5) 回到步骤 2。

插值与逼近

图 10.11 区分了主要的两类曲线生成算法。图 10.11(a)显示了生成**插值**控制顶点的曲线 $P(t)$ 的算法:$P(t)$ 精确地通过这些控制顶点且在这些点中间形成光滑的曲线。图 10.11(b)使用了生成逼近控制顶点的曲线 $R(t)$ 的算法。$R(t)$ 依次被每个控制顶点所吸引,但实际上并不经过所有控制顶点。从随后各种算法的学习中我们将会看到每种算法都有优点和缺点。

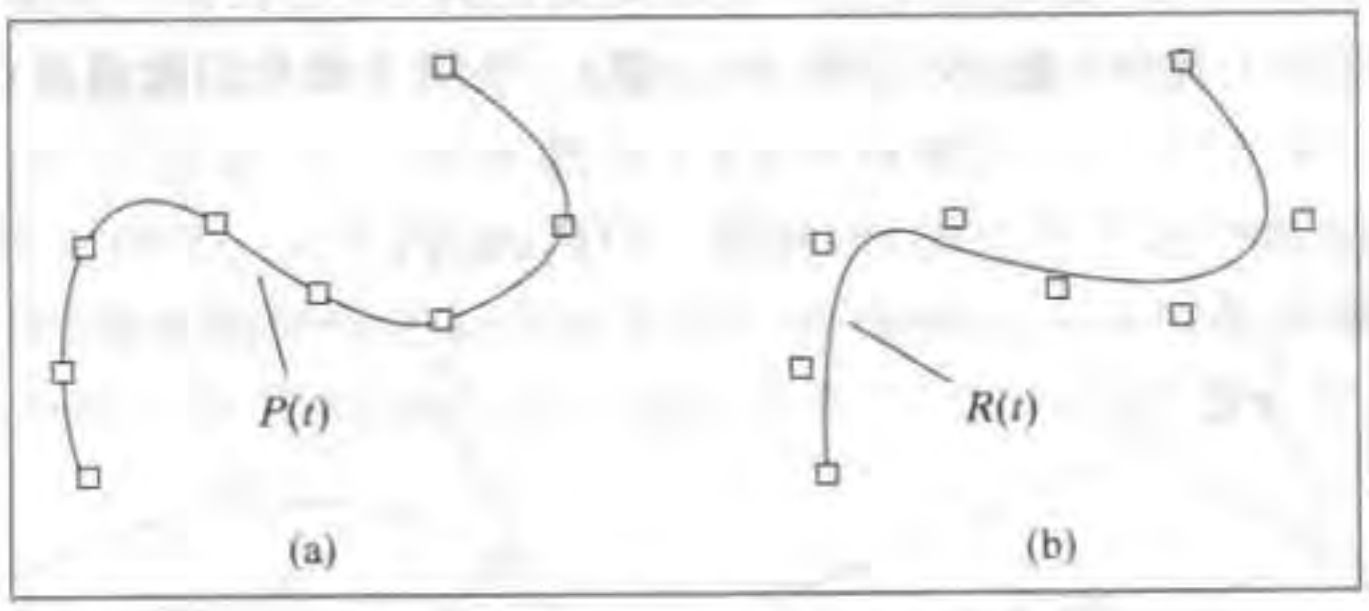


图 10.11　插值和逼近曲线生成算法

迭代曲线设计的流程是计算机辅助设计(CAD)领域中一个重要方面且当设计一个要加工的物体时经常被使用。

下面几节我们将推出一系列曲线和曲面设计的技术。在许多可以采用的方法中选择时,重点放在用贝塞尔和 B 样条的交互曲线设计。这些曲线族在 CAD 应用中已经变得很流行。我们的介绍将会比较简要,但是我们会提供足够的细节使读者能够编写程序来执行交互曲线设计和绘制所设计的物体。

我们首先介绍几种曲线逼近的技术,主要研究贝塞尔曲线和 B 样条曲线。尽管设计者似乎总是希望得到的曲线能够插值控制顶点,但是我们将看到逼近方法是有优点的。接下来我们再研究如何调整算法确保对控制顶点的插值。

10.4　用贝塞尔曲线进行曲线设计

我们以简洁精致的生成贝塞尔曲线的**德卡斯特里奥**算法开始,这个算法是 CAD 的基础。贝塞尔曲线由 Paul de Casteljau 于 1959 年发明,后来又由 Pierre Bézier 在 1962 年左右独立发明。这些算法都成为两个汽车公司(雪铁龙和雷诺)中作为汽车车身形状设计的 CAD 系统的重要部分。

10.4.1 德卡斯特里奥算法

德卡斯特里奥算法用一系列点 $P_0, P_1, P_2, \cdots$,对于 0 到 1 之间的每个 t 值为点 $P(t)$ 构造一个定义好的值。因此这个算法提供了一种由一组点生成一条曲线的方法。改变这些点就改变了这条曲线。构造过程基于一系列熟悉的易于实现的内插步骤(回顾第 5 章)。因为内插是一个很稳定的过程,对其生成的曲线可能会推出许多有价值的性质。

在 3 点之间内插生成抛物线

从三个点 P_0、P_1 和 P_2 开始,如图 10.12(a)所示。在 0 和 1 之间选择某个 t 值,比如 $t=0.3$,把点 A 放在从 P_0 到 P_1 的线段上占两点距离的比例为 t 的位置。类似地,把点 B 放在从端点 P_1 到 P_2 的线段上比例 t 处(使用同一个 t 值)。从第 5 章我们知道新点可以被表示为

$$A(t) = (1-t)P_0 + tP_1$$
$$B(t) = (1-t)P_1 + tP_2 \tag{10.18}$$

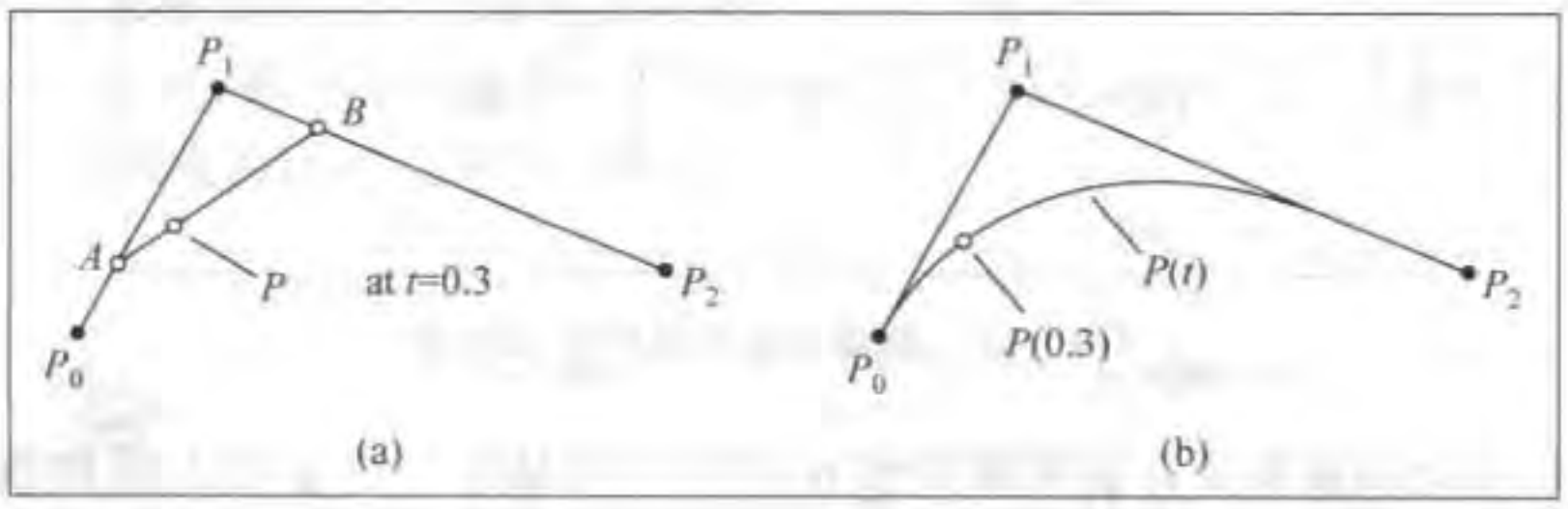


图 10.12 基于三点的德卡斯特里奥算法

现在对这些点重复线性插值步骤(还使用同一 t 值)。寻找位于从 A 到 B 线段上比例为 t 处的点 $P(t)$:

$$P(t) = (1-t)A + tB \tag{10.19}$$

如图 10.12 所示。例如,对 $t=0.5$,$P(0.5)$ 就是给定的三点的"中点的中点"。如果对于 0 和1 之间的每个 t 值执行这个过程,曲线 $P(t)$ 将被生成,如图 10.12(b)所示。这条曲线的参数形式是什么?通过直接把等式(10.18)代入等式(10.19),我们得到

$$P(t) = (1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2 \tag{10.20}$$

它可以被表示为第 5 章中函数 Tween()项的形式(见练习)。

$P(t)$ 的参数形式是 t 的二次多项式,所以从 10.2 节中知道这条曲线是抛物线。即使 t 从 $-\infty$ 到 ∞ 变化,这条曲线依然是抛物线。显然,在 $t=0$ 处曲线通过点 P_0,在 $t=1$ 处曲线通过点 P_2。(为什么?)因此我们得到一种定义好的过程能够基于给定的三点生成一条光滑的抛物线。

如果使用的控制顶点多于三个将会怎样?最常用的贝塞尔曲线族是那些基于四个控制顶点的。图 10.13(a)显示了如何对点 P_0、P_1、P_2 和 P_3 应用德卡斯特里奥算法。对于给定的 t 值,点 A 位于从 P_0 到 P_1 的线段上比例为分数 t 的位置上,点 B 和点 C 类似。然后点 D 位于从 A 到 B 线段上分数 t 处,E 点类似。最后,要求的点 P 位于从 D 到 E 线段上分

数 t 处。如果对0 到 1之间的每个 t 值都进行计算，就形成了从 P_0 开始，被 P_1 和 P_2 吸引，终止于 P_3 的曲线$P(t)$。这就是由四点确定的贝塞尔曲线。图 10.13(b)显示了不同位置的四点所确定的贝塞尔曲线。

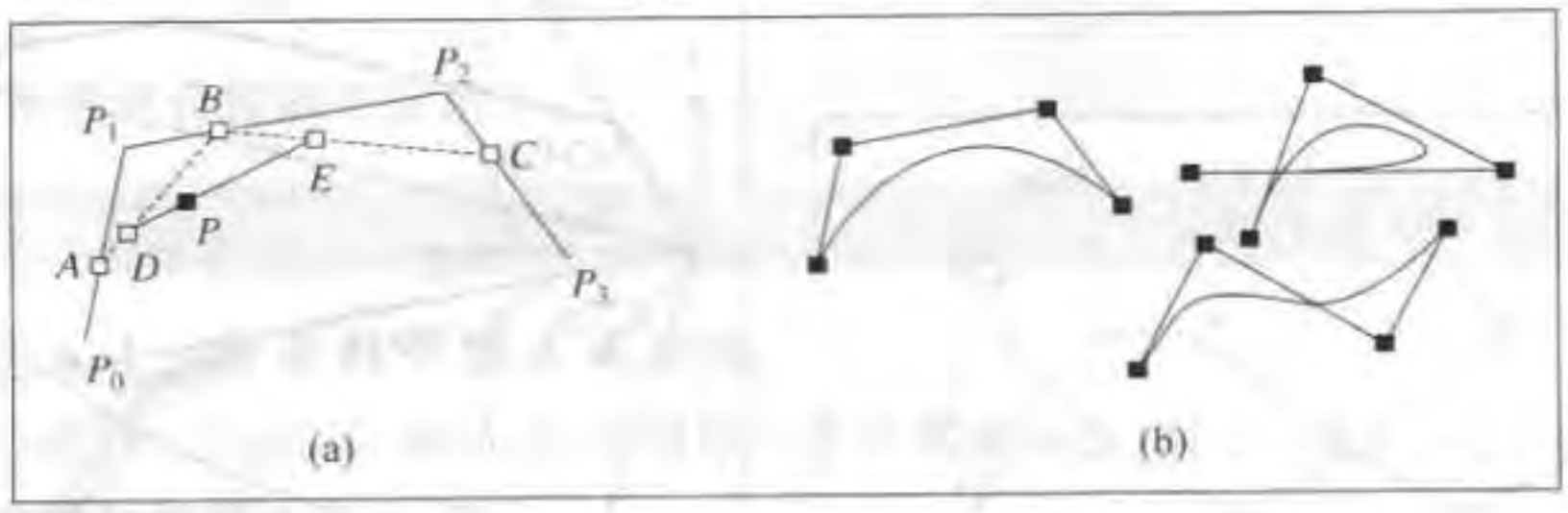


图 10.13　基于四点的贝塞尔曲线

很容易证明(见练习)基于四点的贝塞尔曲线有如下参数形式：

$$\mathbf{P}(t) = \mathbf{P}_0(1-t)^3 + \mathbf{P}_1 3(1-t)^2 t + \mathbf{P}_2 3(1-t)t^2 + \mathbf{P}_3 t^3 \tag{10.21}$$

它是 t 的三次多项式。从等式(10.21)注意到 $P(t)$包含了四个控制顶点，每项都被一个三次多项式加权，然后把加权之后的项相加。这里所包含的项都被看作 **Bernstein 多项式**。这四个三次 Bernstein 多项等式(它们的阶是多少？它们的系数是多少？)是：

$$\begin{aligned} B_0^3(t) &= (1-t)^3 \\ B_1^3(t) &= 3(1-t)^2 t \\ B_2^3(t) &= 3(1-t)t^2 \\ B_3^3(t) &= t^3 \end{aligned} \tag{10.22}$$

三次 Bernstein 多项式很容易被记为展开表达式 $a(t)=(1-t+t)$到三阶所得到的项，这个表达式显然对所有的 t 值都简单地为 1。现在展开 $a^3(t)$然后按照$(1-t)$和 t 的各次幂合并得到的项。

$$((1-t)+t)^3 = (1-t)^3 + 3(1-t)^2 t + 3(1-t)t^2 + t^3$$

这立即产生 Bernstein 多项式的一个重要性质：对于任意的 t 值这些多项式加起来和为 1。(为什么？)

$$\sum_{k=0}^{3} B_k^3(t) = 1 \tag{10.23}$$

为此等式(10.21)中 $P(t)$显然是点的仿射组合，所以把加权后的点加起来是合理的。(回顾第 5 章中添加点的问题。)

图 10.14 画出了 t 从 0 到 1 变化时四个三次 Bernstein 多项式的形状。可以看到这些多项式随 t 光滑地起伏。接下来我们将看到它们有多么“光滑”。

图 10.15 从几何上解释了等式(10.21)中点 $P_0,\cdots,P_3$ 是如何被融合在一起形成 $P(t)$。把这些点看作是从原点出发的向量(因此我们把 P_0 写为 $\mathbf{p}_0$，其他类似)且令 $t=0.3$。则等式(10.21)变为

$$\mathbf{p}(0.3) = 0.343\mathbf{p}_0 + 0.441\mathbf{p}_1 + 0.189\mathbf{p}_2 + 0.027\mathbf{p}_3$$

图中四个向量被加权且加权结果用平行四边形法则加起来形成向量 $\mathbf{p}(0.3)$。可以看到，在 $t=0.3$ 处做主要贡献的是 $\mathbf{p}_0$ 和 $\mathbf{p}_1$。虽然第三项和第四项的权值很小，但是这些项对 $\mathbf{p}(0.3)$依然有影响因为向量 $\mathbf{p}_2$ 和 $\mathbf{p}_3$ 相对较大。

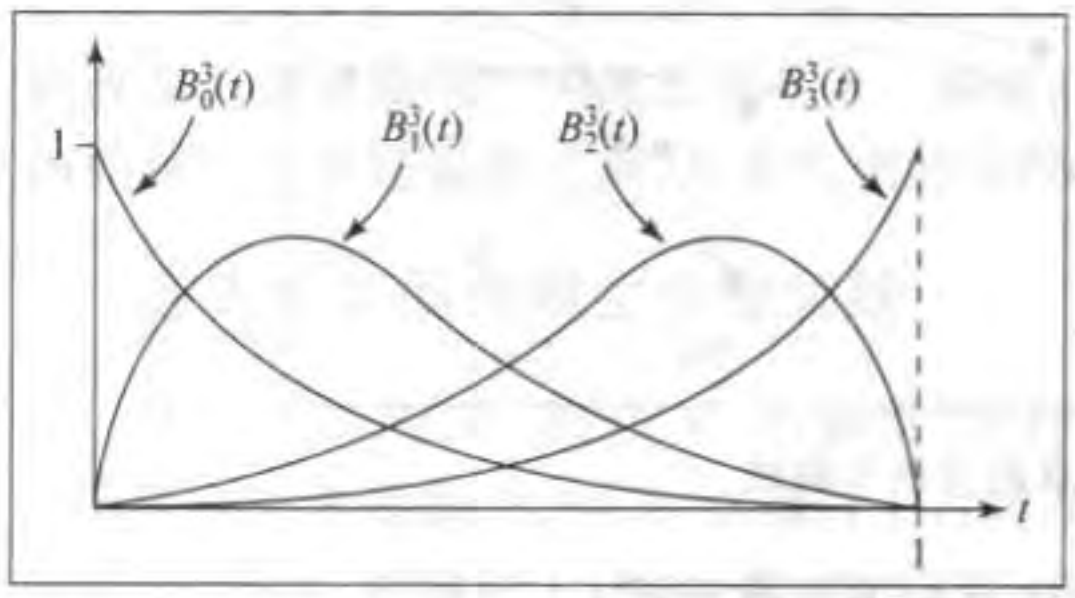


图 10.14 三次 Bernstein 多项式

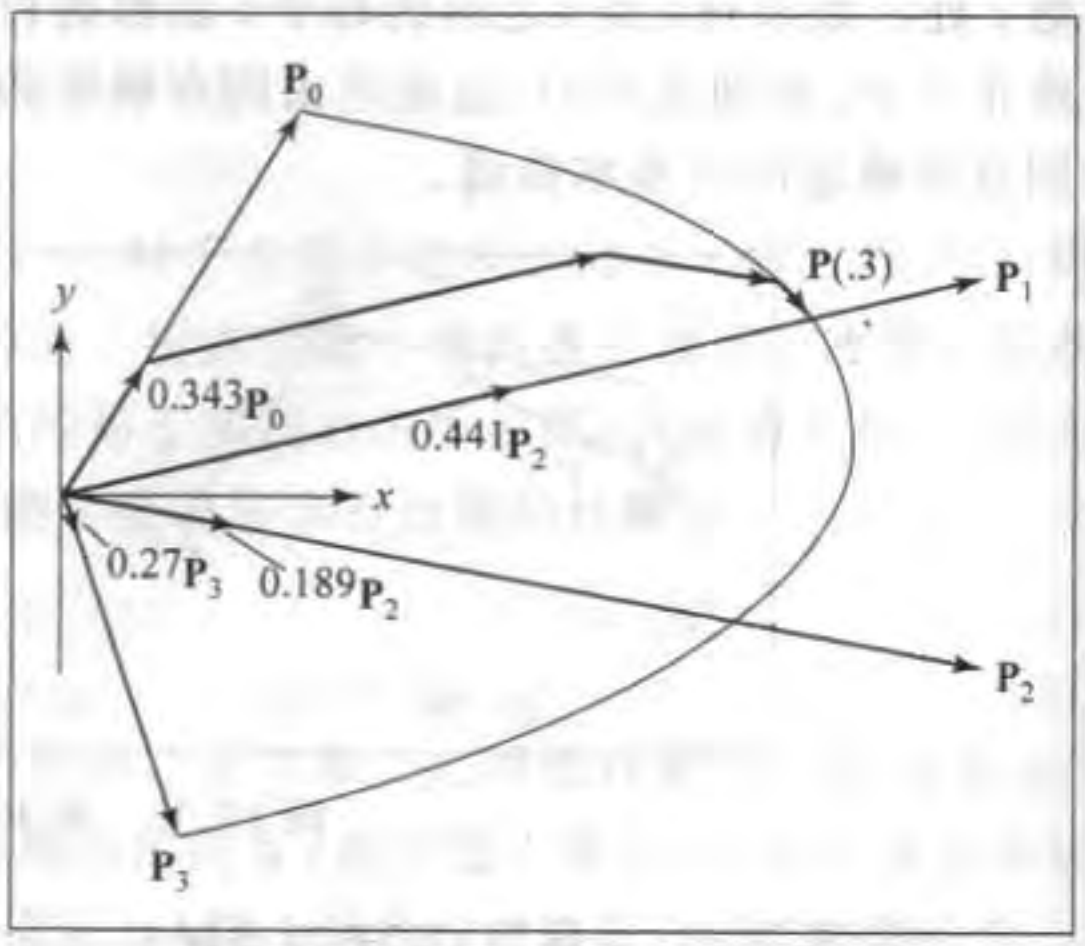


图 10.15 用 Bernstein 多项式混合四个向量

随着 t 的变化,这四个向量的权值也发生变化,且 $\mathbf{p}(t)$转移到不同的位置。试着看出在 $t=0.5$时各个向量的权重以及在形成向量 $\mathbf{p}(0.5)$时各个向量的贡献。

把德卡斯特里奥算法扩展到任意多个点

我们已经看到德卡斯特里奥算法利用内插,当使用三个点时生成二次参数表达式,当使用四个点时生成三次参数表达式。这个算法可以自然地推广到使用 $L+1$ 个控制顶点 P_0, $P_1,\cdots,P_L$的情形。对于每个 t 值,可以连续类似计算多次,每次计算都是通过插值前一次计算生成的相邻点得到:对于 $i=0,\cdots,L$ 有

$$\mathbf{P}_i^4(t) = (1-t)\mathbf{P}_i^3(t) + t\mathbf{P}_{i+1}^3(t)$$

$$\cdots$$

$$\mathbf{P}_i^L(t) = (1-t)\mathbf{P}_i^{L-1}(t) + t\mathbf{P}_{i+1}^{L-1}(t) \tag{10.24}$$

$\mathbf{P}_i^k(t)$中上标 k 表示第 k 次计算。这个过程以 $\mathbf{P}_i^0(t)=P_i$ 开始,结束于最后的贝塞尔曲线 $P(t)=P_i^L(t)$。得到的贝塞尔曲线可以用 Bernstein 多项式表示为

$$P(t) = \sum_{k=0}^{L} P_k B_k^L(t) \tag{10.25}$$

其中第 k 个 L 次 Bernstein 多项式被定义为[①]

$$B_k^L(t) = \binom{L}{k}(1-t)^{L-k}t^k \tag{10.26}$$

这里$\binom{L}{k}$是**二次项系数函数**,由下面给出

$$\binom{L}{k} = \frac{L!}{k!(L-k)!} \quad 对\ L \leqslant k \tag{10.27}$$

① 熟悉概率论的读者会注意到这种形式与二次项概率分布非常相似。

如果 $L<k$，这项值被定义为 0。每个 Bernstein 多项式都被看作是 L 次的。如以前，Bernstein 多项式是展开$[(1-t)+t]^L$ 所得到的各项，因此我们确保

$$\sum_{k=0}^{L} B_k^L(t) = 1 \quad \text{对所有的 } t \tag{10.28}$$

且 $P(t)$是各点合理的仿射组合。

练习

练习 10.4.1　用图形计算生成曲线

对于三个点(0,0)、(2,4)和(6,1)应用德卡斯特里奥算法，对于 t 值 0、0.2、0.4、0.6、0.8 和 1 分别在绘图计算器上画出 $P(t)$。

练习 10.4.2　二次曲线必是平面曲线

证明过三个(不同的)点存在唯一的一条二次曲线。回想三个(非共线的)点确定一个平面，证明它们所确定的二次曲线不会偏离这个平面。因此抛物线永远不会偏离一个平面。作为额外的收获，证明三次曲线可能是非平面的。

练习 10.4.3　由 Tween()表示的贝塞尔曲线

回顾 4.5.3 节中的函数 Tween()。它用两个点作为参数，且对于给定的 t 值返回这些点的内插值。证明等式(10.20)中 $P(t)$可以被写为 Tween(Tween(P0,P1,t), Tween(P1,P2,t),t)，或者两个内插值的内插值。写出三次贝塞尔曲线的类似形式。

练习 10.4.4　三次贝塞尔曲线

通过写出类似于前一个练习中的连续的内插值，证明等式(10.21)给出的贝塞尔曲线的参数形式是正确的。

练习 10.4.5　建立贝塞尔曲线的直观图

利用等式(10.21)，当四个控制顶点为(2, 3)、(6, 6)、(8, 1)和(4, −3)时在时间 $t=0.2,0.5,0.9$计算 $P(t)$的位置。如图 10.15 所示，画出对最终的 $P(t)$值做贡献的加权后的向量的草图。

练习 10.4.6　四次和五次贝塞尔曲线

写出 $L=4$ 情形下等式(10.26)中的 Bernstein 函数。证明这些函数是展开$[(1-t)+t]^4$ 所得到的各项。对 $L=5$ 重复这个过程。

练习 10.4.7　Bernstein 多项式的递推关系

证明 n 阶 Bernstein 多项式总可以由 $n-1$ 阶 Bernstein 多项式利用下式推出

$$B_i^n(t) = (1-t)B_i^{n-1}(t) + tB_{i-1}^{n-1}(t) \tag{10.29}$$

其中 $B_0^0(t)=1$，且当 j 不在 $0,\cdots,n$ 的范围内时 $B_j^n(t)=0$。

提示： $\begin{pmatrix} n \\ i \end{pmatrix} = \begin{pmatrix} n-1 \\ i \end{pmatrix} + \begin{pmatrix} n-1 \\ i-1 \end{pmatrix}$.

练习 10.4.8 贝塞尔曲线插值两端点

证明一般的 L 次贝塞尔曲线通过两个最外面的控制顶点：$P(0)=P_0$ 和 $P(1)=P_L$。通过在 $t=0$ 和 $t=1$ 处计算 Bernstein 多项式的值来证明，并证明在这两个端点处除一项外其他所有项都为 0。

10.5 贝塞尔曲线的性质

贝塞尔曲线具有一些重要的性质使其很适合于 CAD。我们将在后面看到这些性质也适合于 B 样条。学习这些性质及其证明提供了对贝塞尔曲线更深入的理解。

端点插值

基于控制顶点 $P_0,P_1,\cdots,P_L$ 的贝塞尔曲线 $P(t)$ 通常并不通过或插值所有控制顶点。但是我们已经看到，它总是插值 P_0 和 P_L。这是一个很重要的性质，因为输入一系列点的设计者精确地知道贝塞尔曲线从哪里开始到哪里终止。

仿射不变性

为了缩放贝塞尔曲线，改变其方向或其位置以备后用，需要其满足一个仿射变换。假定我们希望利用仿射变换 T(T 在 $2D$ 情况下用 3×3 矩阵表示，在 $3D$ 情况下用4×4矩阵表示)把等式(10.25)中贝塞尔曲线上点 $P(t)$变换到新点 $Q(t)$。因此 $Q(t)=T(P(t))$。看起来对于任意给定的 t 值为了计算 $Q(t)$我们必须先计算 $P(t)$，然后对它进行变换，对于每个新的 t 值重复这个计算过程。实际上并不需要这样。我们只需变换控制顶点(一次)，对任意 t 在同样的 Bernstein 形式中用这些新的控制顶点来重新生成变换后的贝塞尔曲线！即：

$$Q(t)=\sum_{k=0}^{L}T(P_k)B_k^L(t) \qquad (10.30)$$

仿射不变性就是变换后的曲线与基于变换后的控制顶点的曲线是相同的。

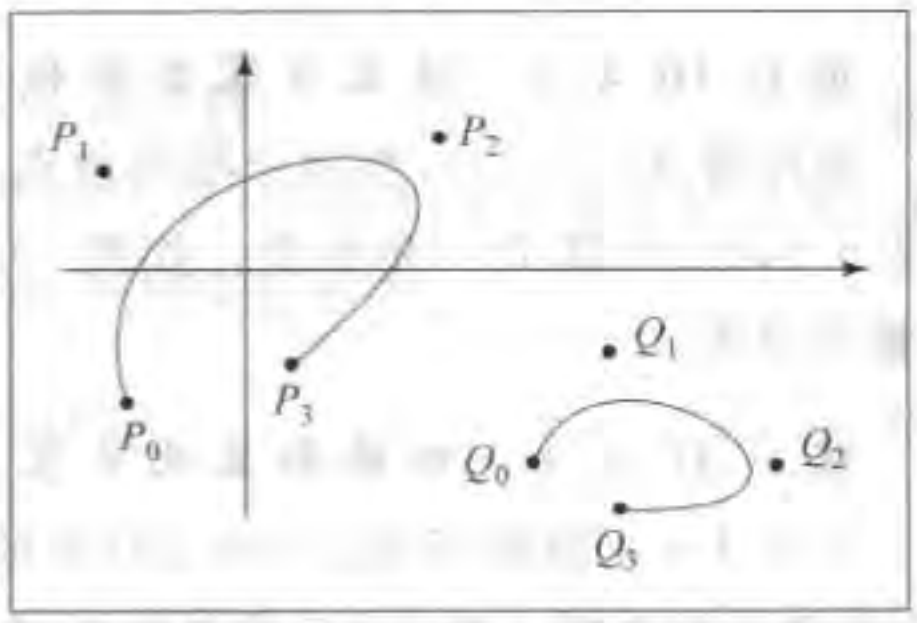


图 10.16 显示仿射不变性

图 10.16 显示了一条基于四个控制顶点 $P_0,\cdots,P_3$的贝塞尔曲线。这些点被旋转，伸缩和平移到新的控制顶点 Q_k，且由它们确定的贝塞尔曲线也画出来了。这条曲线与变换原始的贝塞尔曲线所得的结果相同。

贝塞尔曲线具有仿射不变性的原因很简单：它们是由点的仿射组合形成，且从 5.2 节中我们知道仿射变换保持仿射组合。

凸包性

设计者常常依赖的另一个性质是贝塞尔曲线，$P(t)$，不会超出其凸包。回顾第 5 章中点集 $P_0,P_1,\cdots,P_L$ 的凸包是所有这些点的凸组合的集合，即由下式给出的所有点的集合

$$\sum_{k=0}^{L} \alpha_k P_k \tag{10.31}$$

其中每个 α_k 是非负的，且它们的和为 1。

等式(10.25)中的 $P(t)$ 对每个 t 都是它的控制顶点的凸组合，因为 Bernstein 多项式不会为负数且其和为 1。因此贝塞尔曲线上的每个点都是其控制顶点的凸组合，所以贝塞尔曲线必定位于其控制顶点的凸包内。

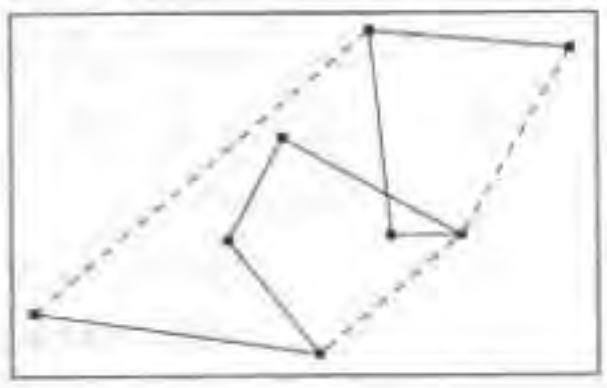


图 10.17　利用凸包性

凸包性也可由以下事实推出：曲线上的每个点都是两个内插点的内插值，且两个点的内插值形成它们的凸组合。图 10.17 阐明了设计者如何利用凸包性质。即使图中的 8 个控制顶点形成了一个锯齿状的控制多边形，设计者也知道贝塞尔曲线会在两个端点之间光滑移动，而且永远不会超出凸包的范围。

贝塞尔曲线的导数

因为当曲线关于 t 的导数有不连续点时曲线上会出现尖角或突变，所以我们需要研究等式(10.25)中 $P(t)$ 的各种导数。

对于贝塞尔曲线可以证明其一阶导数为

$$\mathbf{p}'(t) = L \sum_{k=0}^{L-1} \Delta P_k B_k^{L-1}(t) \tag{10.32}$$

其中

$$\Delta P_k = P_{k+1} - P_k \tag{10.33}$$

(见练习)速度曲线是建立在一组新的控制顶点 ΔP_k 上的另一条贝塞尔曲线。我们简单地对原始控制顶点两两进行差分来形成速度曲线的控制向量。从形式 $B_k^{L-1}(t)$ 注意到求导把曲线阶数降低一阶。例如三次贝塞尔曲线的导数是二次贝塞尔曲线。贝塞尔曲线的光滑性在练习中讨论。

生成并绘制贝塞尔曲线

假定我们希望建立一个应用程序使其通过一系列控制顶点绘制一条贝塞尔曲线。如何组织这样的应用程序呢？等式(10.25)中贝塞尔曲线 $P(t)$ 将用一条近似折线形来绘制。在密集分布的 t 值上对函数 $P(t)$ 采样－比如 $t_i = i/N$ 对于 $i = 0, 1, \cdots, N$－且用直线连接点 $P(t_i)$。唯一的问题是在每个要求的 t 值处计算 $P(t)$，我们假定用如下的函数来执行：

```
Point2 bezier(RealPointArray poly, double t);
/* 根据控制多边形计算贝塞尔曲线上点的位置 P(t) */
```

这个程序使用存储在 poly 中的点数组(称为控制多边形)对于给定的 t 值计算等式(10.25)的结果并返回所得到的点。注意 Bernstein 多项式的次数存储在 poly.num 中。程序的实现留到练习中。图 10.18 显示了一个完整的可执行程序其中用户可以指定附加的控制顶点。每组三个点指定后，另一条贝塞尔曲线就被画出来了。这个程序的深层扩展将在案例分析中讨论。

```
#include<windows.h>
#include<math.h>
#include<gl/Gl.h>
#include<gl/glut.h>

int SCREEN_HEIGHT=480;
int NUMPOINTS=0;
class Point {
public:
    float x, y;
    void setxy(float x2, float y2) { x=x2; y=y2;}
    cont Point & operator (const Point &rPoint) {
        x=rPoint.x;
        y=rPoint.y;
        return * this;
    }
};
Point abc[3];
void myInit() {
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0, 0.0, 0.0);
    glPointSize(4.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIndentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
void drawDot(int x, int y) {
    glBegin(GL_POINTS);
    glVertex2i(x, y);
    glEnd();
    glFlush();
}
void drawLine(Point p1, Point p2) {
    glBegin(GL_LINES);
    glVertex2f(p1.x, p1.y);
    glVertex2f(p2.x, p2.y);
    glEnd();
    glFlush();
}
Point drawBezier(Point A, Point B, Point C, double t) {
    Point P;
    P.x=pow((1-t), 2) * A.x+2 * t * (1-t) * B.x+pow (t, 2) * C.x;
    P.y=pow((1-t), 2) * A.y+2 * t * (1-t) * B.y+pow (t, 2) * C.y;
    return P;
}
```

图 10.18　一个完整的贝塞尔曲线应用程序练习

```
void myMouse (int button, int state, int x, int y) {
    if(button==GLUT_LEFT_BUTTON && state==GLUT_DOWN) {
      abc[NUMPOINTS].setxy((float) x, (float)(SCREEN_HEIGHT-y));
      NUMPOINTS++;
      drawDot(x, SCREEN_HEIGHT-y);
      if(NUMPOINTS==3) {
          glColor3f(1.0, 0.0, 1.0);
          drawLine(abc[0], abc[1]);
          drawLine(abc[1], abc[2]);
          Point POld=abc[0];
          for (double t=0.0; t<=1.0; t+=0.1){
              Point P=drawBezier(abc[0], abc[1], abc[2], t);
              drawLinw(POld, P);
              POld=P;
          }
          glColor3f(1.0, 0.0, 0.0);
          NUMPOINTS=0;
      }
    }
}
void myDisplay() {
    glClear(GL_COLOR_BUFFER_BIT);
    glFlush();
}

int main(int argc, char * agrv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | CLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(100, 150);
    glutCreateWindow("Bezier Curve");
    glutMouseFunx(myMouse);
    glutDisplayFunc(myDisplay);
    myInit();
    glutMainLoop();
        return 0;
}
```

图 10.18（续）

练习

练习 10.5.1　画出速度

在绘图计算器上或绘图纸上画 4 个点，画出对应的贝塞尔曲线，仔细的计算它的速度和加速度向量关于 t 的函数。

10.6 寻找更好的混合函数

似乎贝塞尔曲线为曲线设计提供了终极工具。通过在平面上适当地放置控制顶点就能够生成各种各样的光滑曲线。

但在后面将看到,在曲线设计中贝塞尔曲线自身并未提供足够的灵活性。一个问题是所用的 Bernstein 多项式的次数与控制顶点的个数相关:基于 $L+1$ 个控制顶点的贝塞尔曲线是 L 次多项式的组合。高次多项式计算起来代价很高,且易有数值舍位误差。我们希望设计者可以随意选取任意多个控制顶点——甚至 40 个或更多。

10.6.1 局部控制问题

一个更加重要的问题是贝塞尔曲线并未提供足够的对曲线形状的局部控制。图 10.19 给出了使用五个控制顶点生成一条贝塞尔曲线(实线)的情形,其在 $t=1$ 附近偏离了所期望的曲线(虚线)。

为了纠正这个偏离,用户需要向上移动 P_2 和 P_3 以使贝塞尔曲线更接近所期望的曲线。但是,如图 10.19 所示,这也影响了曲线的前半部分,迫使它偏离所期望的曲线。

问题是任意一个控制顶点的改动都会改变整个曲线。这起因于 Bernstein 多项式的本质(回顾图 10.14):在整个区间[0,1]上每个都是"活动"(意思是不为 0)的。函数值不为 0 的区间通常叫做它的**支撑**。因为每个 Bernstein 多项式在整个区间[0,1]上都有支撑,且曲线是这些函数的混合,所以每个控制顶点对 0 到 1 之间 t 值处的曲线都有影响。因此调整任何控制顶点都会影响曲线各处的形状,缺少局部控制。

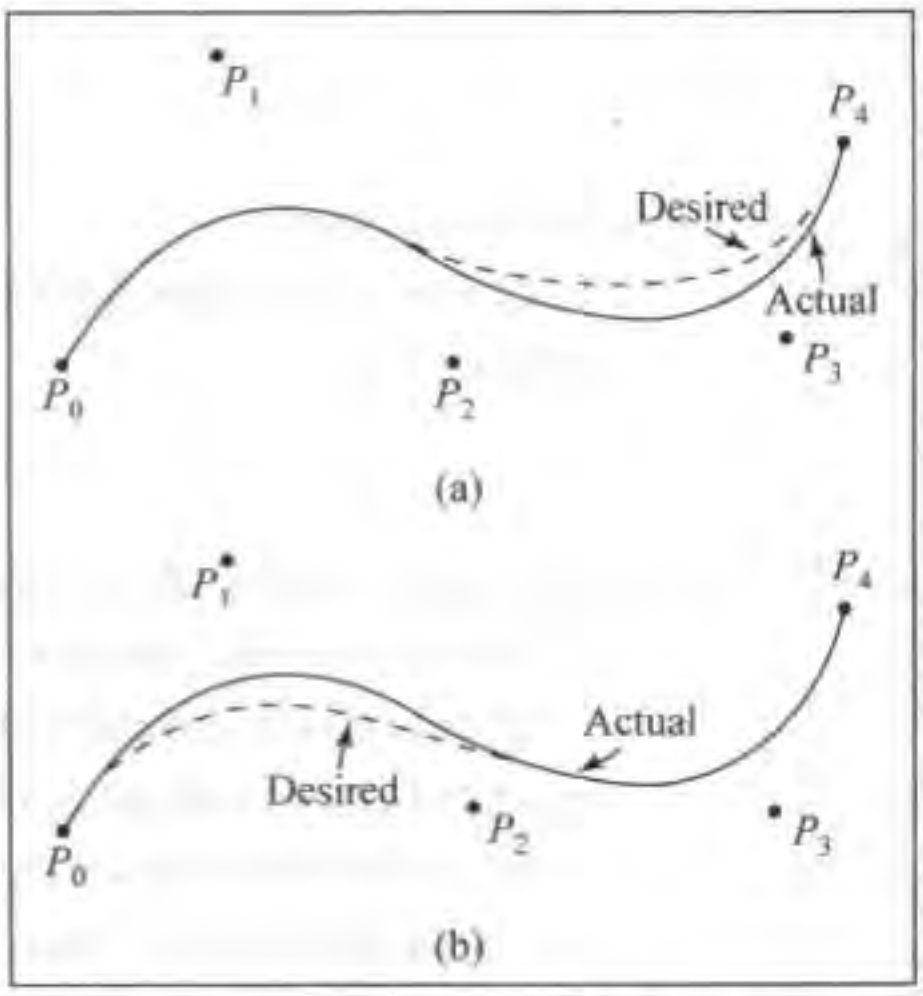


图 10.19 编辑曲线的某些部分

把上述情况与图 10.20 中给出的一组有用的猜想的混合函数相比较。图中显示了六个混合函数,$R_0(t), R_1(t), \cdots, R_5(t)$(后面将定义①),每个函数只在区间[0,1]的一部分上有支撑。例如,$R_0(t)$的支撑是[0,.25],而 $R_3(t)$的支撑是[.25,1]。事实上,对于任意 t 值这些混合函数中活动的不会超过 3 个。

考虑使用这些混合函数,基于六个控制顶点 $P_0, P_1, \cdots, P_5$ 建立一条曲线 $V(t)$。我们使用与贝塞尔曲线相同类型的参数形式:

$$V(t) = \sum_{k=0}^{5} P_k R_k(t) \tag{10.35}$$

① (预览):这些实际上是后面要研究的二次 B 样条。

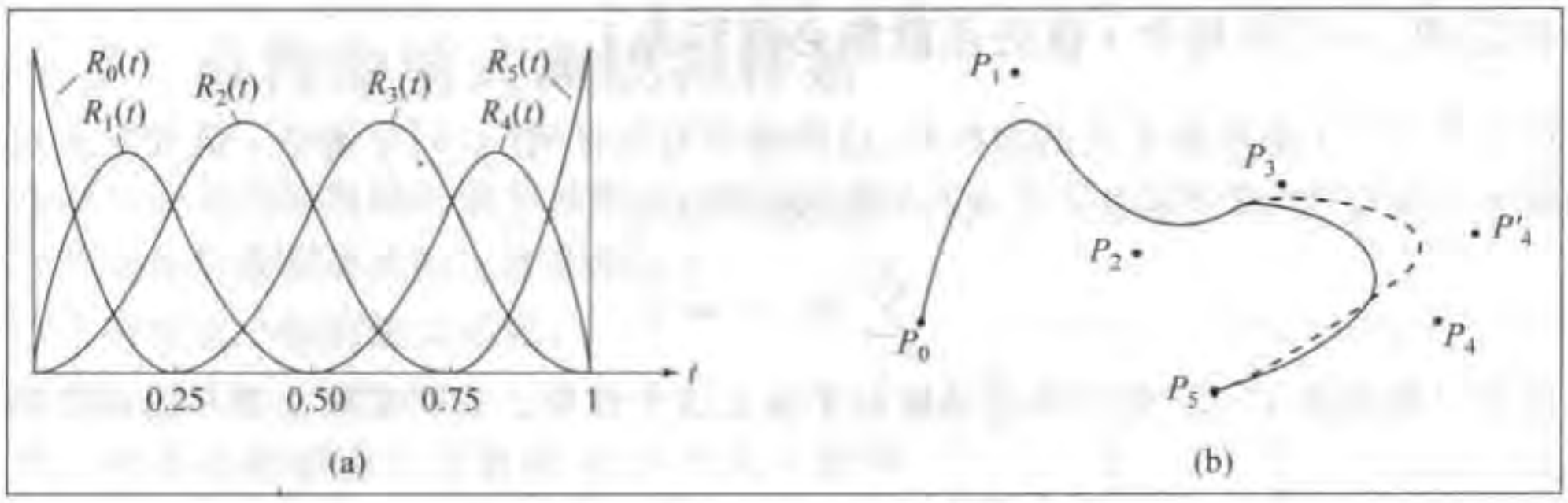


图 10.20　具有集中支撑集的混合函数

(a) 比较有希望的具有集中支撑集的猜想的混合函数集；

(b) 一个控制顶点调整前(实线)后(虚线)的示例曲线

但寻找了一组较好的函数 $R_k(t)$。图 10.20(b)显示了基于一组示例控制顶点的曲线。在每个 t 值处位置 $V(t)$所依赖的控制顶点不会超过 3 个。特别地，在$[0.75,1]$中的 t 值处只有点 P_3，P_4，和 P_5 控制曲线的形状。如果单个控制顶点 P_4 被移动到 P'_4，只有曲线中用虚线表示的那部分会发生变化。因此这组混合函数确实使得控制顶点具有某些局部控制力。

10.6.2　混合函数集的性质

图 10.20 中混合函数的形状经过了调整只是为了说明局部控制的性质。但是它们事实上是基于曲线设计中常用的函数。它们的详细性质将在后面描述，但是我们在这里对它们进行深入研究是为了对混合函数集应该具有的性质有个印象。

和往常一样，我们假定要使用的曲线生成算法依照下式混合控制顶点 $P_0,\cdots,P_L$：

$$V(t)=\sum_{k=0}^{L}P_kR_k(t),\quad t\text{ 在}[a,b]\text{ 区间内} \tag{10.36}$$

其中混合函数 $R_0(t),\cdots,R_L(t)$具有某些性质使其能够生成更好的曲线或者能使设计过程更加直观。混合函数应该具有如下的这些性质：

- 易于计算且数值稳定；
- 在区间$[a,b]$中每个 t 值处和为 1；
- 具有小的支撑集来提供局部控制；
- 插值由设计者选定的某些控制顶点；
- 足够光滑。

我们分别考虑各个性质。

函数集应该易于计算且数值稳定

为了快速生成曲线我们希望混合函数计算起来很简单。我们还希望它们受数值舍位误差的影响最小。这些考虑引导我们选择多项式作为混合函数，选择次数尽可能低且能连续生成所需要的曲线的多项式。其他类型的函数，例如正弦和余弦函数，其代价过高而不能使用。

在区间$[a,b]$中每个 t 值处函数集必须和为 1

$V(t)$在每个 t 值处都是点的加权和,这使得只有在区间$[a,b]$中每个 t 值处它是点的仿射和时才有意义(否则它不是一个点)。因此我们坚持

$$\sum_{k=0}^{L} R_k(t) = 1 \tag{10.37}$$

你可以看到图 10.20 中的候选函数似乎满足这个性质。我们实际上要用的函数确实和为 1。

函数集应该具有小支撑集用于局部控制

之前我们已经讨论过:为了达到局部控制我们希望每个混合函数集中在某些 t 值上,只在区间$[a,b]$的一小部分上有支撑。

函数集应该插值某些控制顶点

设计者可能希望 $V(t)$通过一些控制顶点而只被另外一些控制顶点所吸引。图 10.20 中的形状只插值了第一个和最后一个控制顶点。这是显然的因为在 $t=0$ 和 $t=1$ 处除了一个混合函数其他的都为 0,迫使剩下的那个为 1(为什么)使得能够插值。(为什么?)我们将在不久后看到怎样提供一个机制来调整混合函数使得另外的一些控制顶点被插值。

函数集应该有足够的光滑性

设计者通常希望对于任意控制顶点集 $V(t)$都是一条光滑曲线。典型地,为了生成所期望的曲线形状的光滑性 $V(t)$至少应该是一阶光滑的,可能甚至是二阶光滑的。$V(t)$的光滑性依赖于混合函数的光滑性。特别地,如果每个混合函数在$[a,b]$上是一阶光滑的,则 $V(t)$在$[a,b]$上也是一阶光滑的。

注意图 10.20 中的混合函数似乎在其内部是光滑的。起始和结束的平稳也是很重要的。图 10.21显示了一个混合函数及其一阶导数。

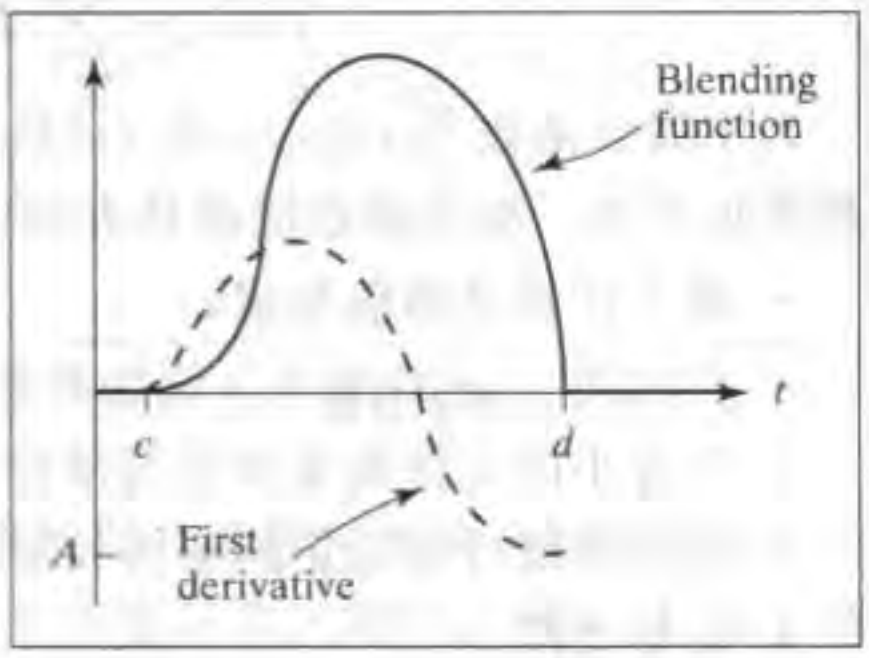


图 10.21 一个候选混合函数及其导数

函数的导数在起始处 $t=c$ 从 0 连续变化。但它的导数在结束处 d 不连续,从一个负数突变到 0。这个函数在除了 $t=d$ 处处是一阶光滑的。使用这个混合函数的曲线在 $t=d$ 不会是一阶光滑的。因此要求混合函数在其内部光滑,且在起始和终止处导数为 0。

图 10.20 中某些形状在起始和终止处导数为 0,而其他的不是这样。那些起始点和终止点在区间[0,1]的形状满足起始和终止点处导数为 0,所以 $V(t)$在区间(0,1)内是一阶光滑的。在区间[0,1]的端点处导数不连续是可以接受的因为我们永远不会使用超过 0 和 1 的 t 值。

10.6.3　分段多项式曲线和样条

我们开始寻找好的候选混合函数，比如低次多项式。例如，是否存在具有图 10.22 所示形状且满足我们所有要求的三次多项式？

为了研究这个我们定义函数：

$$R(t) = at^3 + bt^2 + ct + d$$

然后看是否能够选到参数使 $R(t)$ 及其一阶导数在 $t=0$ 和 $t=1$ 处都为 0。这导出关于系数的四个条件：

$$R(0) = d = 0$$
$$R(1) = a + b + c + d = 0$$
$$R'(0) = c = 0$$
$$R'(1) = 3a + 2b + c = 0$$

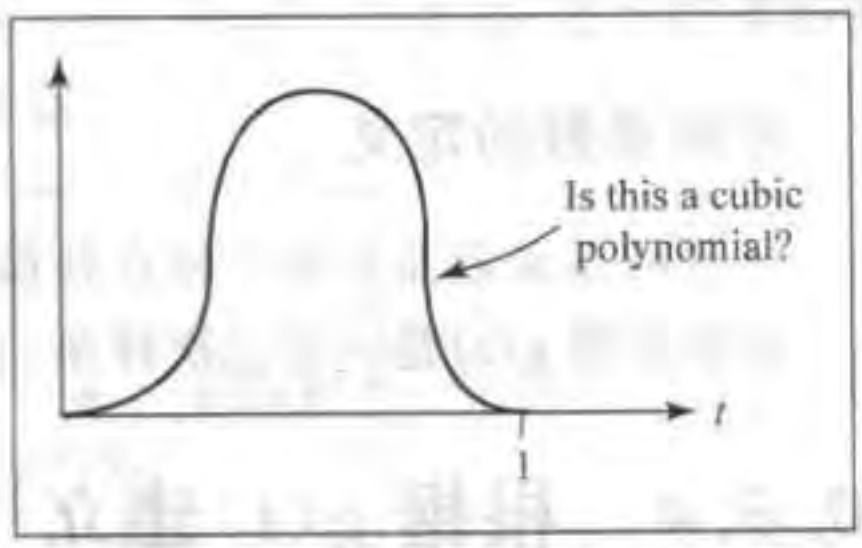


图 10.22　能否找到这样的三次多项式

不幸的是，这些条件使得 $a=b=c=d=0$，所以不存在这样的形状。也就是说三次曲线不具有足够的灵活性来实现我们所需要的混合（练习中会问四次多项式是否满足这个要求）。

为了获得更多的灵活性我们试着把几个低次多项式分段连接起来。这样在不同的 t 分段区间上由不同的多项式定义的曲线称为**分段多项式**。

图 10.23 给出的示例形状 $g(t)$ 帮助建立几个术语。我们看到 $g(t)$ 由三个多项式**段**，$a(t)$、$b(t)$ 和 $c(t)$ 组成，如下定义

$$a(t) = \frac{1}{2}t^2$$

$$b(t) = \frac{3}{4} - \left(t - \frac{3}{2}\right)^2$$

$$c(t) = \frac{1}{2}(3-t)^2 \tag{10.38}$$

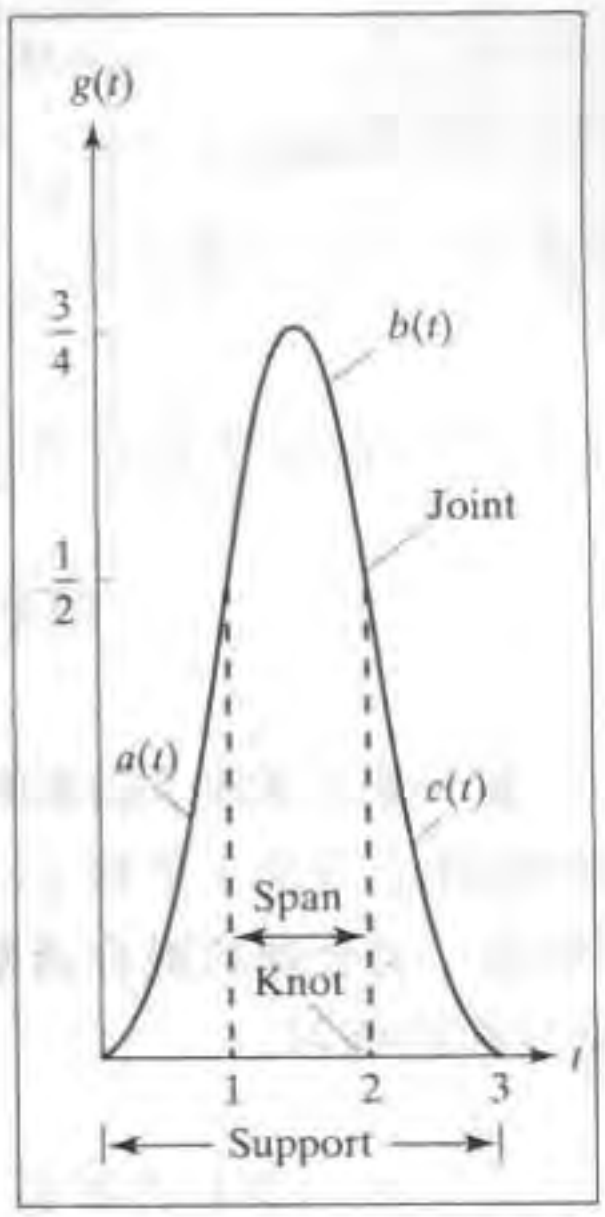


图 10.23　分段多项式成分

$g(t)$ 的支撑区间是 $[0,3]$；$a(t)$ 定义在区间 $[0,1]$，$b(t)$ 定义在区间 $[1,2]$，$c(t)$ 定义在区间 $[2,3]$。一对分段多项式相遇的点叫做**连接点**，发生相遇时的 t 值叫做**结点**。本例中有四个结点：0、1、2 和 3。

再一次强调：

- 结点是分段相遇时的 t 值；
- 连接点是相邻的分段函数值。

$g(t)$ 在其支撑区间上是否处处连续呢？因为它是由多项式定义的，在每个区间内部它肯定是连续的，因此我们只需检查这些分段在连接点的情况。从等式(10.37)中这很容易检查：$a(1)=b(1)=1/2$ 和 $b(2)=c(2)=1/2$。

进一步，$g(t)$ 的导数是处处连续的，所以 $g(t)$ 在 $[0,3]$ 上是 1-光滑的。为此，注意到导

数在每段区间内部必定是连续的(为什么?),所以只需在结点处检查它的连续性。直接计算表明 $a'(1)=b'(1)=1$ 和 $b'(2)=c'(2)=-1$。因此当我们从一个多项式段移到下一个时,斜率不会发生突变。

二阶导数是不连续的,但是在结点(哪个?)处的两个值是否发生跳跃?

这里的形状 $g(t)$ 是**样条函数**的一个例子,分段多项式函数具有足够的光滑性。特别地,我们有

样条函数的定义

M 次样条函数是在每个结点处都是 $M-1$ 阶光滑的 M 次分段多项式。

显然示例 $g(t)$ 是一个二次样条:它是一个二次分段多项式且有处处连续的一阶导数。

10.6.4 根据 $g(t)$ 建立一组混合函数

如何将上述的样条函数 $g(t)$ 作为混合函数来表示曲线呢?一种方法是使用平移后的 $g(t)$,其中每个混合函数 $g_k(t)$ 是用某个数量平移基本形状 $g(t)$ 形成的。图 10.24 显示了由将 $g(\cdot)$ 平移一个整数位置所形成的 7 个混合函数 $g_0(t),\cdots,g_6(t)$:

$$g_k(t) = g(t-k), \quad 其中\ k = 0,1,\cdots \tag{10.39}$$

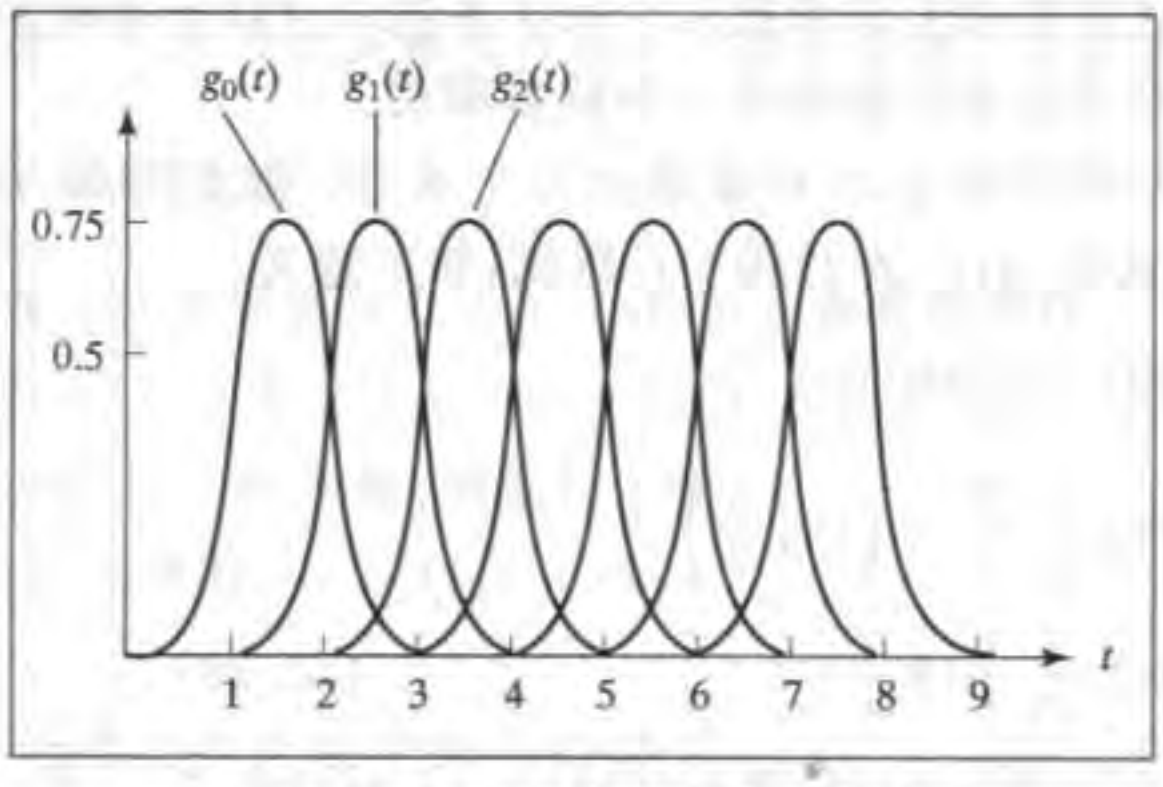


图 10.24 表示样条曲线的混合函数

由于 $g(\cdot)$ 各个版本的结点都出现在整数位置,所以平移曲线使其结点叠加在下个曲线的结点上等价于平移 $g(\cdot)$。只有当这些平移后的版本在任意 t 值处和为 1 时,它们才能形成一组合理的混合函数。然而对于 2~7 之间的所有 t 值这些函数确实满足这一点!(见后面的练习。)

$$\sum_{k=0}^{6} g(t-k) = 1 \quad t\ 在[2,7]内 \tag{10.40}$$

这似乎看起来很神奇,但是我们将看到这是我们最终得到的混合函数的一般属性。这里的关键是把每个函数平移一个整数,适当地把这些形状连接起来使其和为 1。

设计者选定七个控制顶点且利用如下算法生成曲线:

$$V(t) = \sum_{k=0}^{6} P_k g(t-k) \tag{10.41}$$

只能用 2 和 7 之间的 t 值(为什么?)。在这个范围内,注意到在任意 t 值处恰有三个混合函数是活动的,所以对曲线形状有很好的局部控制。另外注意到在特殊时刻 $t=2$, 3,…,7处只有两个函数是活动的且其值都为 0.5。因此在这些 t 值处 $V(t)$将位于其中两个控制顶点所形成线段的中点上。

图 10.25 给出了七个控制顶点和所得到的曲线的一个示例。曲线在 $t=2$ 处从 P_0P_1 的中点起始,且光滑地波动依次通过控制多边形各个边的中点。随着 t 的增加,各个混合函数起起落落,且对曲线的主要影响从一个点传递到另一个点。

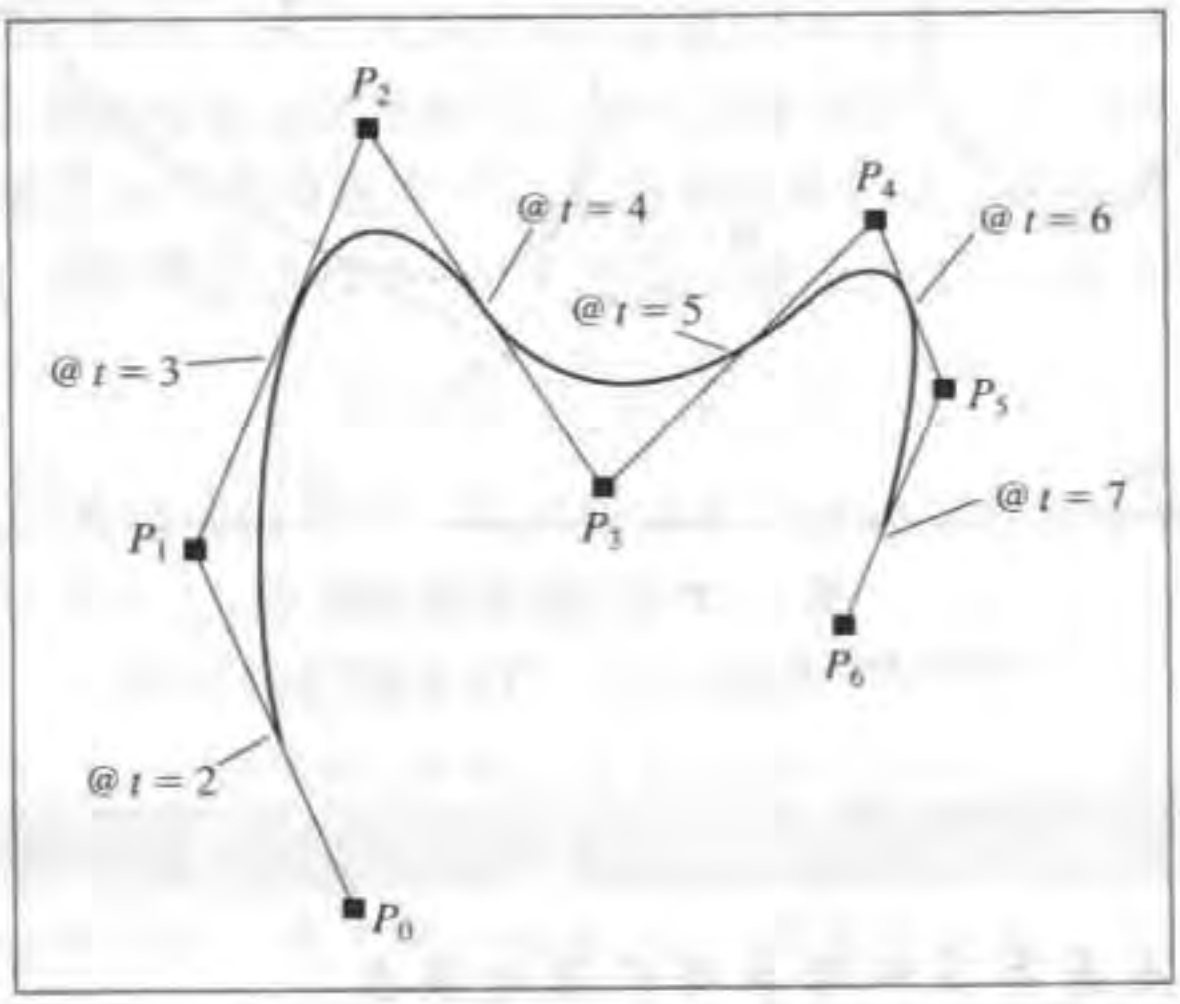


图 10.25　利用 $g(\cdot)$的平移设计曲线

基于这组混合函数的曲线具有哪些性质?

- 设计者对曲线形状具有很好的局部控制,因为每个混合函数的支撑区间的长度都不超过 3。
- 设计者放置控制顶点时可以利用曲线必定通过控制多边形各边中点的这一特点。因此这个算法具有某些直观的几何性质。
- 因为每个混合函数都是 1-光滑的,所以整个曲线是 1-光滑的。
- 除了 P_0 和 P_6 曲线上任何控制顶点都不会被插值。
- 所有多项式都是二次的,所以它们计算起来既快速又稳定。多项式的次数不依赖于控制顶点的个数。这种方法适用于任意多个控制顶点。

刚描述的曲线生成算法可以用下面的程序很容易地实现

```
Point2 curvePt(double t, RealPointArray pts)
```

对于每个输入的 t 值返回点 $V(t)$。这个将在案例 10.4 中更加深入讨论。

例 10.6.1　扩展到绘制封闭曲线

不难扩展这种方法使其能够生成如图 10.26(a)所示的封闭曲线。图 10.26(b)显示了图 10.25 中曲线的封闭版本。为了实现这个目标,你所需做的就是通过复制控制顶点 P_0 和 P_1,从而增加了两个额外的控制顶点,使得方程(10.41)的和式中添加两项。我们知道在

$t=7$ 前面的曲线通过线段 P_5P_6 的中点。因此在 $t=8$ 时曲线通过 P_6P_7 的中点,且在 $t=9$ 时曲线通过 P_7P_8 的中点,于是封闭了曲线。另一种方法在练习中给出。

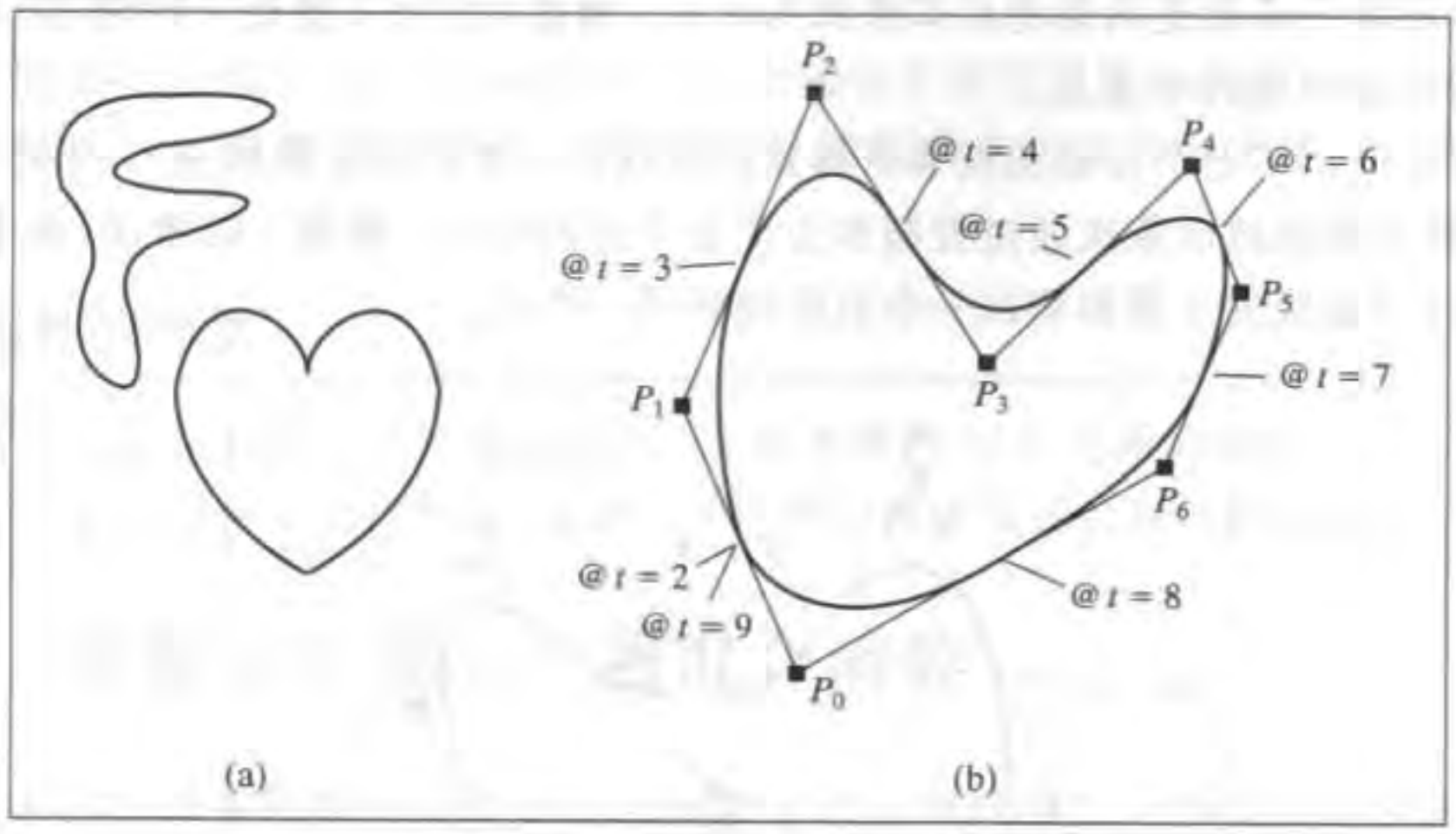


图 10.26 生成封闭曲线

(a) 封闭曲线的例子;(b) 一种生成封闭曲线的方法

练习

练习 10.6.1 是否存在四次多项式混合函数?

考虑四次多项式 $r(t)=at^4+bt^3+ct^2+dt+e$。是否存在这样的 a、b、c、d 和 e 使得函数通过 0 且其一阶导数在 $t=0$ 和 $t=1$ 处都等于 0? 如果存在,找到合适的值且画出曲线的草图。

练习 10.6.2 证明 g(·)的平移版本在任意 t 值处和为 1

考虑平移的和 $g(t)+g(t-1)+g(t-2)$,其中 $g(t)$是等式(10.38)中定义的分段多项式。证明对于 2 和 3 之间的任意 t 值这个和总为 1。

练习 10.6.3 Bernstein 多项式是样条吗?

证明 Bernstein 多项式确实是样条。形成每个多项式需要多少个分段多项式? 结点位于何处? 这个样条的次数是多少? 是否足够连续可微?

练习 10.6.4 试验二次样条曲线的直观性

以某种复杂的模式画出 12 个控制顶点,然后画出由等式(10.38)的二次样条函数生成的曲线的粗略草图。利用曲线通过控制多边形各边中点这个事实。

练习 10.6.5 没有额外控制顶点的情况下生成封闭曲线

证明对等式(10.41)做稍微变动

$$V(t)=\sum_{k=0}^{6}P_k g((t-k)\bmod 7) \tag{10.42}$$

随着 t 从 0 到 7 变化时能够生成基于七个(各不相同的)控制顶点的封闭曲线。不需要控制顶点的副本。取模函数 mod 把混合函数有效地折叠到区间[0,7]内部,使这些函数在

这个区间的不同部分活动。为了了解这种方法是怎么实现的，画出 $L=4$ 时当 t 从 0 变化到 5 对于 $k=0,1,2,3,4$ 的 5 个函数 $g((t-k)\bmod(L+1))$。

10.6.5　样条曲线和基函数

上面这种包含了二次函数 $g(t)$ 平移的方法似乎给我们提供了一种很好的曲线设计工具。为什么还要深入研究呢？问题是我们需要对曲线形状有更多的控制：它必须更加弯曲且光滑度高于一阶。这使得我们转向研究三次多项式。我们还希望设计者能够指定哪些控制顶点被插值。如果有单个算法能够包含上面所描述的所有设计方法——包括贝塞尔曲线，就好了。

因此我们想寻找满足前面期望表中列出的所有性质的更一般的混合函数族。我们继续使用同样的基于 $L+1$ 个控制顶点和 $L+1$ 个混合函数 $R_0(t),\cdots,R_L(t)$ 的参数形式：

$$P(t)=\sum_{k=0}^{L}P_kR_k(t) \tag{10.43}$$

我们依然要求 $P(t)$ 是点的仿射和。我们继续使用分段多项式作为混合函数，但现在它们定义在更一般的结点序列上，称为**结点向量**，即 $\mathbf{T}$：

$$\text{结点向量}:\mathbf{T}=(t_0,t_1,t_2,\cdots) \tag{10.44}$$

也就是结点值是 $t_0,t_1,\cdots$ 的序列，且假定是非递减的：$t_i\leqslant t_{i+1}$。某些结点可能具有相同的值但仍然给予不同的名字。T 中结点的名字将在下面讨论。

图 10.27 说明了这种情形。每个混合函数 $R_k(t)$ 都是在时刻 t_k 之前为 0，在结点向量中的几个分段区间上非 0，然后又回到 0 的混合函数。我们还要求每个 $R_k(t)$ 都是样条函数，于是这个函数在其支撑集的所有 t 处具有一定的光滑性。

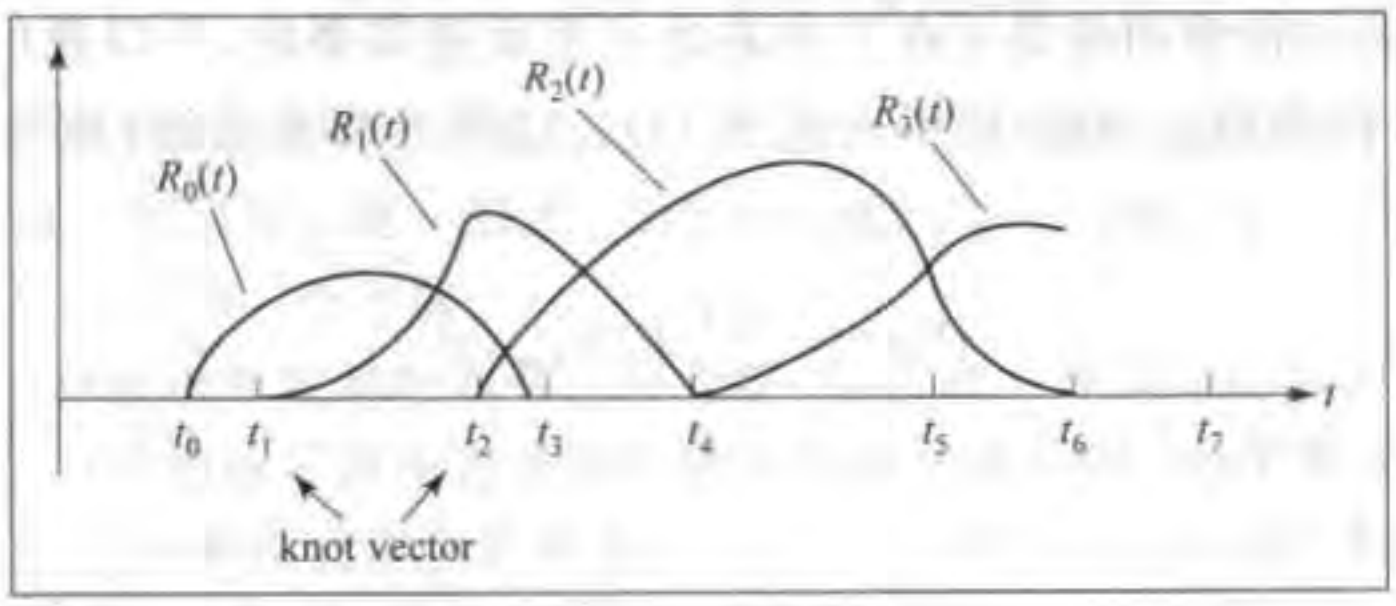


图 10.27　结点向量和混合函数的推广

注意贝塞尔曲线和 $g(t)$ 的平移公式都满足这个更一般的形式：

- 贝塞尔：有一个从 0 到 1 的区间，所有的 Bernstein 多项式都是 L 次的，在 $t=0$ 处有 $L+1$ 个结点且在 $t=1$ 处有 $L+1$ 个结点。Bernstein 多项式是样条，因为它们具有连续的 L 阶导数。
- $g(t)$ 的平移：结点值是整数 $0,1,\cdots,L+2$；每个平移函数 $g(t-k)$ 都是一个样条，由三个二次多项式组成且覆盖长度为 3 的区间。

因为每个 $R_k(t)$ 都是一个分段多项式，所以整个曲线 $P(t)$ 是用控制顶点加权的分段多项式的总和。例如，在某个区间曲线可能由下式给出

$$P(t)=P_0(3t^2-4t+2)+P_1(8t^2-7.3t-7.99)+\cdots \tag{10.45}$$

在不同的分段区间内,它由不同的多项式之和给出,但是我们知道所有分段都能够使曲线连续。这样的曲线称为样条曲线【Farin88】[①]。

现在提出的问题是:给定一个结点向量,是否存在某组混合函数使其能够生成所有定义在这个结点向量上的样条曲线?这样一组函数称为样条基,意味着任何样条曲线通过选取适当的控制多边形都可以于等式(10.43)的和相匹配。

答案是存在很多这样的函数族,但是只有一组特别的基函数其混合具有最小的支撑集并因此提供最大的局部控制。它们就是 B 样条,其中'B'源于单词 basis。

10.7 B 样条基函数

我们希望能用一种直观的方式去定义 B 样条混合函数 $R_k(t)$,这样有利于计算机实现。虽然文献给出了很多定义 B 样条的方法,但是还存在一个公式可以定义任意阶的 B 样条函数。它是递归定义的,容易编程实现,而且不存在数值问题(其他方法具有更高的计算效率,见练习)。

每个 B 样条函数都基于某个阶数 m 的多项式。如果 $m=3$,多项式的阶数为 3,度数为 2,所以它定义 2 次 B 样条。如果阶数 $m=4$,也就是度数为 3 或者立方。这是两种最重要的情况,尽管公式允许我们定义任意阶的 B 样条。

10.7.1 B 样条函数的定义

让 B 样条函数的阶数明确显示在它的表达式里面是需要的,所以我们用 $N_{k,m}(t)$ 来表示第 k 个 m 阶 B 样条混合函数,而不再用 $R_k(t)$。这样 B 样条曲线(见等式(10.43))就可以表示为:

$$P(t) = \sum_{k=0}^{L} P_k N_{k,m}(t) \tag{10.46}$$

给定以下几个条件:

- 节点向量 $\mathbf{T}=(t_0,t_1,t_2,\cdots)$。
- $(L+1)$个控制顶点 P_k。
- B 样条函数的阶 m。

则 B 样条函数 $N_{k,m}(t)$可以这样得到:

$$N_{k,m}(t) = \left(\frac{t-t_k}{t_{k+m-1}-t_k}\right)N_{k,m-1}(t) + \left(\frac{t_{k+m}-t}{t_{k+m}-t_{k+1}}\right)N_{k+1,m-1}(t) \tag{10.47}$$

其中 $k=0,1,\cdots L$。它是一个递归式即如何通过两个 $m-1$ 阶 B 样条函数来定义 m 阶 B 样条函数。要想完备这个定义,还需要给定一初始条件,它是一个区间上值恒为 1 的函数:

---

① 注意 10.6.3 节中定义的样条函数和样条曲线的区别。样条函数只是具有某种程度光滑性的分段多项式。样条曲线是使用分段多项式混合函数得到的点的仿射组合。在结点处样条曲线一定是连续的但不一定具有连续的导数。

$$N_{k,1}(t)=\begin{cases}1 & t_k<t\leqslant t_{k+1}\\ 0 & \text{其他}\end{cases} \tag{10.48}$$

注意到所有这些函数的和在任意点 t 处均为 1，所以把它们表示成点的线性组合是完全合理的。

例 10.7.1　线性 B 样条

当节点向量均匀分布如 $T=(t_0=0,t_1=1,t_2=2,\cdots)$ 时，第一个 2 阶 B 样条函数 $N_{0,2}(t)$ 会是什么形状呢？在这个条件下，等式(10.47)变为：

$$N_{0,2}(t)=\frac{t}{1}N_{0,1}(t)+\frac{2-t}{1}N_{1,1}(t) \tag{10.49}$$

图 10.28(a)给出了线性系数 t 与一阶 B 样条函数 $N_{0,1}(t)$ 相乘以及 $2-t$ 与 $N_{1,1}(t)$ 相乘的图，它们相加得到一个三角形峰线。注意到与此相同的，二阶 B 样条函数 $N_{0,2}(t)$ 是 $N_{0,1}(t)$ 与 $N_{1,1}(t)$ 的仿射组合，所以，当 $0\leqslant t\leqslant 1$ 时 $N_{0,2}(t)=t$；当 $1\leqslant t\leqslant 2$ 时 $N_{0,2}(t)=2-t$；其他情况下 $N_{0,2}(t)=0$。

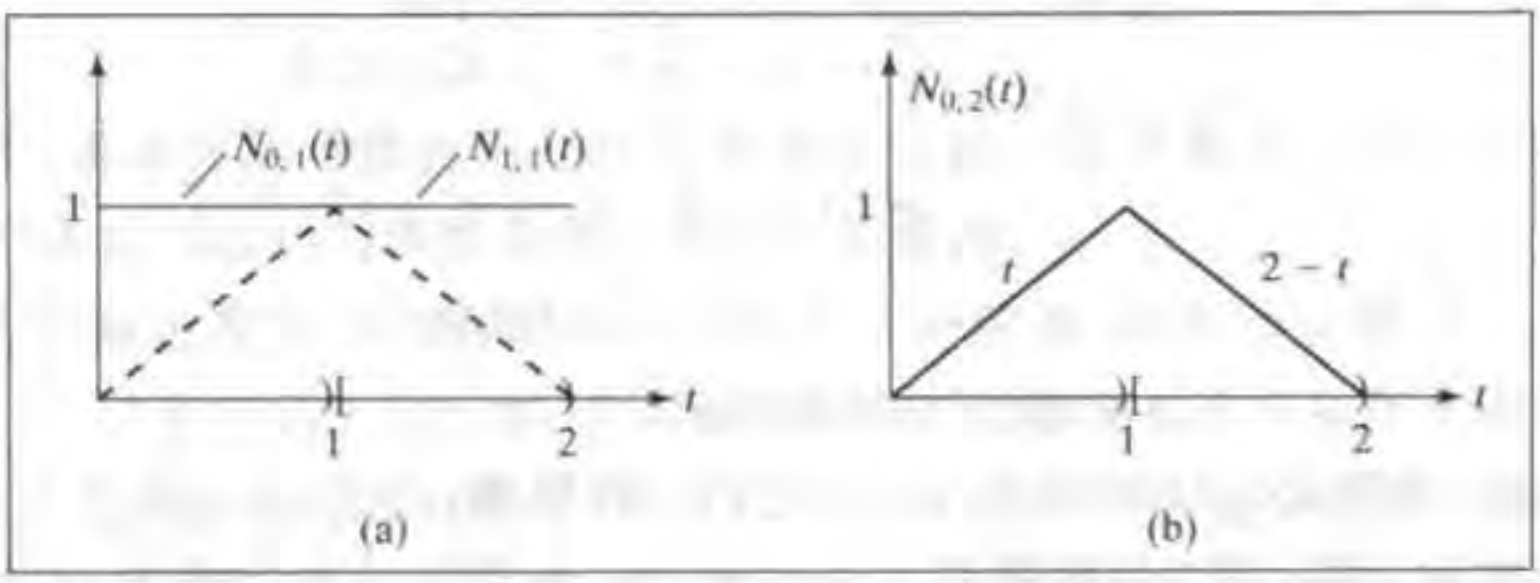


图 10.28　线性 B-样条函数的构造

其余 B 样条函数的构造类似，比如 $N_{1,2}(t)$ 也是一个三角峰线起始于 $t=1$ 结束于 $t=3$，只要将 $N_{0,2}(t)$ 平移一下即可。更一般地，当节点向量均匀分布时，每一个二阶 B 样条函数都只是第一个即 $N_{0,2}(t)$ 的一个平移。

注意到线性 B 样条曲线就是控制多边形本身(但是如果节点向量不均匀分布是不是也这样呢?)由于线性 B 样条除了简单直线没有提供任何其他东西，所以它一般不会被用在曲线设计中，但是它可以用来构造高阶 B 样条。

例 10.7.2　二次 B 样条

假如我们想确定在均匀节点条件下二次($m=3$)B 样条函数 $N_{i,3}(t)$ 的形状，我们只需看 $N_{0,3}(t)$ 的形状就可以了，因为其他都只需将它平移即可。由等式(10.47)得出：

$$N_{0,3}(t)=\frac{t}{2}N_{0,2}(t)+\frac{3-t}{2}N_{1,2}(t) \tag{10.50}$$

上述等式中两项的第一项是一上升系数与第一个三角峰线相乘，第二项是一下降系数与第二个三角峰线相乘，见图 10.29(a)，一系数与三角峰线相乘得到两条抛物线，它们相交形成一个角点，但是当上面等式(10.50)中两项相加时，角点消失得到一条连续曲线 $N_{0,3}(t)$，见图 10.29(b)。

我们可以由等式(10.50)来确定二次 B 样条函数 $N_{0,3}(t)$ 每一部分的代数形式，其结

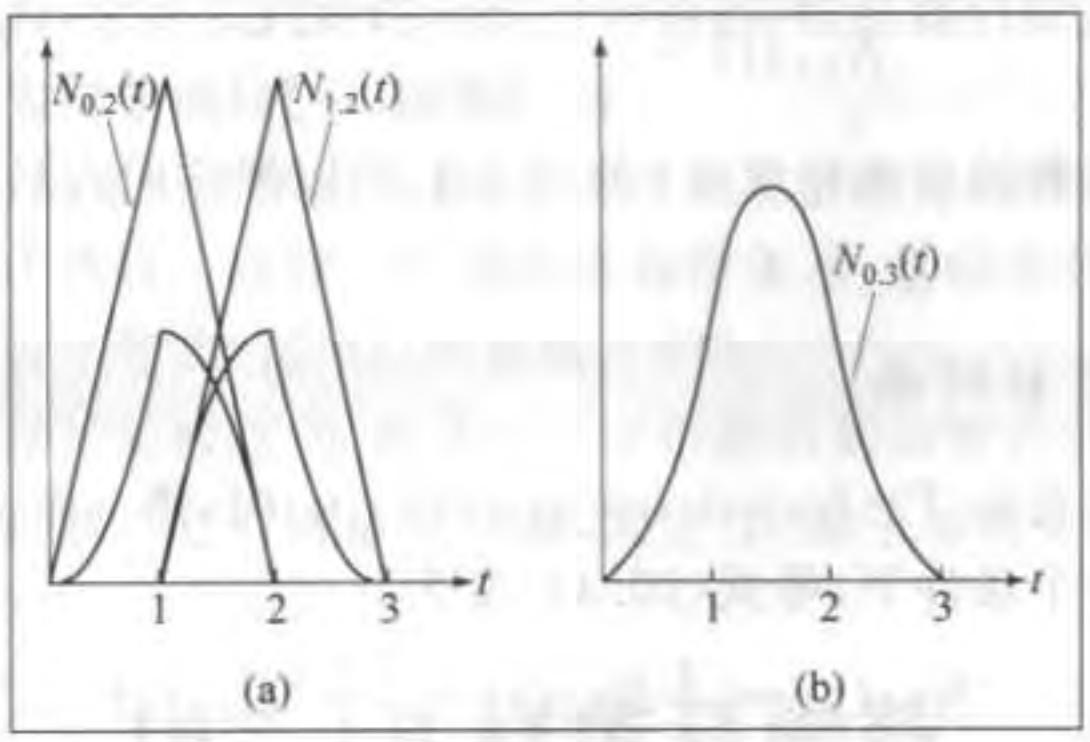


图 10.29 第一个二次 B 样条的形状

果为：

$$N_{0,3}(t)=\begin{cases}\frac{1}{2}t^2 & 0\leqslant t\leqslant 1\\ \frac{3}{4}-(t-\frac{3}{2})^2 & 1\leqslant t\leqslant 2\\ \frac{1}{2}(3-t)^2 & 0\leqslant t\leqslant 1\\ 0 & \text{其他}\end{cases} \tag{10.51}$$

它取决于四个节点 0、1、2、3，其支撑区间为[0，3]。

比较 $N_{0,3}(t)$ 与等式(10.39)中的 $g(t)$，它们一样精确，因为 $g(t)$ 也是 B 样条函数。它的一阶导数存在且连续，但二阶导数就不是这样了。在均匀节点向量下，二次 B 样条确实是一曲线。

当节点向量均匀分布时，其他二阶 B 样条的形状是很容易确定的，因为第一个二次 B 样条函数是其他二次 B 样条的一个简单平移，而且等式(10.47)中每一项系数的不同之处仅在节点向量上，也说明了这一点。事实上，此结论对均匀节点条件下的任意阶 B 样条都是成立的，因为：

$$\text{若节点 } t_k=k,\quad \text{则 } N_{k,m}(t)=N_{0,m}(t-k) \tag{10.52}$$

当节点向量均匀分布时，此形式可以直接替换到等式(10.46)中。

事实上，即使节点不均匀分布，等式(10.47)中所有二次 B 样条的和在任意点 t 处均为 1，详见练习。

例 10.7.3 三次 B 样条

三次 B 样条是最常用的，图 10.30(a)给出了 $N_{0,4}(t)$ 的形状，它的形式与前面类似(见练习)。$N_{0,4}(t)$ 关于 $t=2$ 对称，而且它可以写成：

$$N_{0,4}(t)=\begin{cases}u(1-t) & 0\leqslant t\leqslant 1\\ v(2-t) & 1\leqslant t\leqslant 2\\ v(t-2) & 2\leqslant t\leqslant 3\\ u(t-3) & 3\leqslant t\leqslant 4\\ 0 & \text{其他}\end{cases} \tag{10.53}$$

$$u(t)=\frac{1}{6}(1-t)^3, v(t)=\frac{1}{6}(3t^3-6t^2+4) \tag{10.54}$$

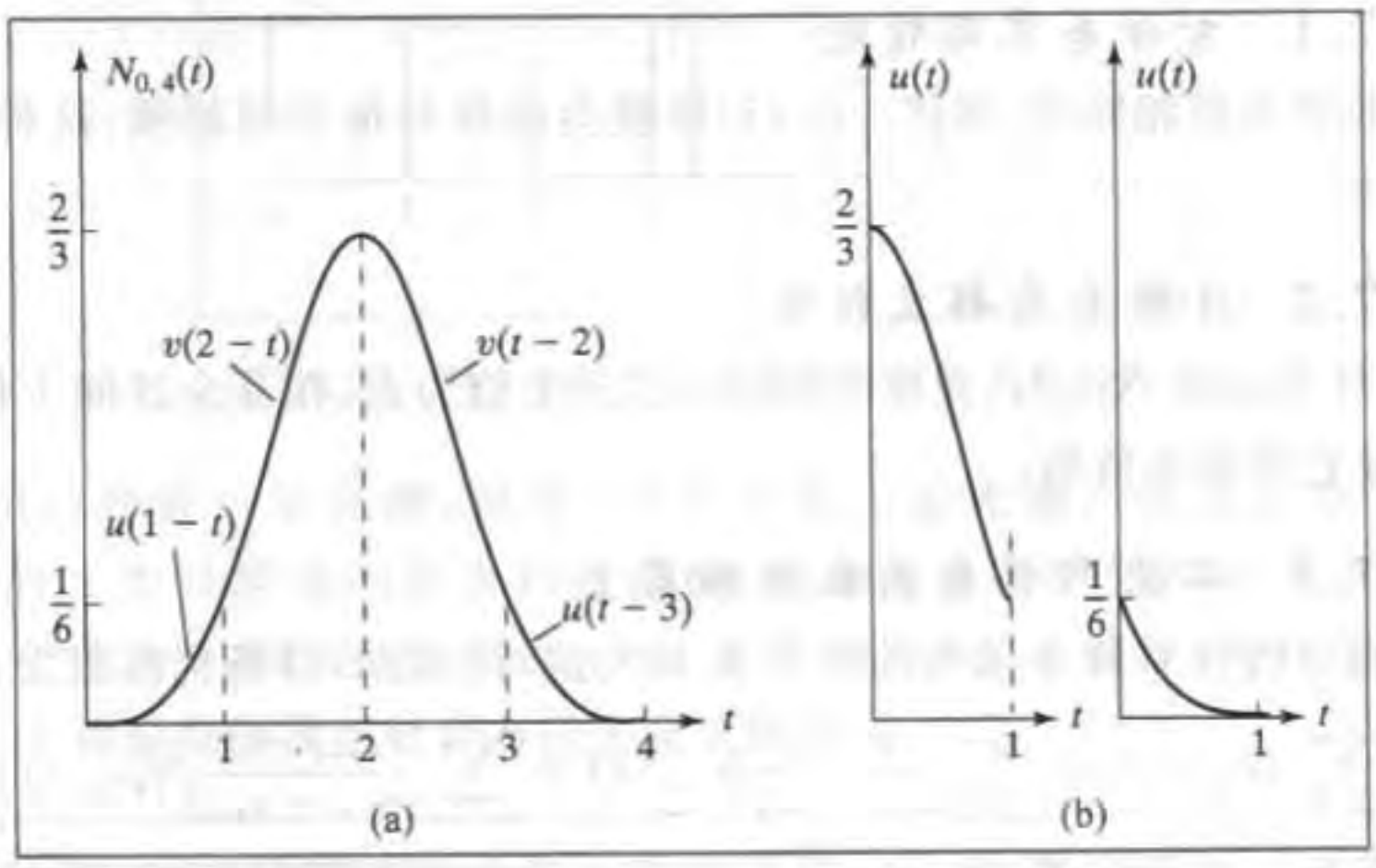


图 10.30　均匀节点下的三次 B 样条

通过直接计算求得的导数表明三次 B 样条曲线的一阶导数及二阶导数都是处处连续的，所以三次样条曲线在均匀节点条件下是二阶光滑的。

基于这里的结论我们还可以得出（见练习），一般地，B 样条函数 $N_{k,m}(t)$ 只在区间 $[t_k, t_{k+m}]$ 上为正，其余区间上均为零，它在整个区间上是非负的。

我们还应注意到除数为零的情况，即等式(10.47)的两项中的分母都可能因为选择特殊的节点向量而变为零，但是当这种情况发生时，其对应的低阶函数即 $N_{k,m-1}(t)$ 和 $N_{k+1,m-1}(t)$ 也总是相应地为零，所以任何一项若它的分母为零，我们便认为这一项为零。

由于 B 样条函数 $N_{k,m}(t)$ 是递归形式定义的，所以它很容易编程实现，容易计算它在给定节点向量条件下任意点 t 处的函数值，图 10.31 给出的代码就是等式(10.47)的实现。

```
float bSpline(lnt k, int m, float t, float knot []
{
      float denom1, denom2, sum=0.0;
      if(m==1)
        return(t>=knot[k] && t<knot[k+1]);        //1 or 0
      //m exceeds 1.. use recursion
      denom1=knot[k+m-1]-knot[k];
      if (denom1 != 0.0)
        sum=(t-knot[k]) * bSpline(k, m-1, t, knot)/denom1;
      denom2=knot[k+m]-knot[k+1];
      if (denom2 != 0.0)
        sum+=(knot[k+m]-t) * bSpline(k, m-1, t, knot)/denom2;
      return sum;
}
```

图 10.31　计算 B 样条函数

练习

练习 10.7.1 分母为零的情况

当两个相邻节点值相同时,等式(10.47)中就会出现分母为零的项,这种情况下我们将此项直接视为零。

练习 10.7.2 B 样条局部支柱性

一般地,B 样条函数 $N_{k,m}(t)$ 只在区间$[t_k, t_{k+m}]$上值为正,在其余区间上值为零,也说明了在任意点 t 处它都是非负的。

练习 10.7.3 二次 B 样条函数的和为 1

我们已经知道线性 B 样条函数的和为 1,其实这对二次 B 样条仍然成立。我们想证明

$$\sum_{k=-\infty}^{\infty} N_{k,3}(t) = \sum_{k=-\infty}^{\infty} \frac{t-t_k}{t_{k+2}-t_k} N_{k,2}(t) + \sum_{k=-\infty}^{\infty} \frac{t_{k+3}-t}{t_{k+3}-t_{k+1}} N_{k+1,2}(t)$$

在任意点 t 处值为 1,应用完等式(10.47)后将第二项中的 k 变化为 $k-1$,然后两项合并为一项,再利用一下线性 B 样条函数和为 1 的条件,就可以得到二次 B 样条函数的和在任意点 t 处为 1。

练习 10.7.4 三次 B 样条函数的计算

对均匀节点下的三次 B 样条,验证等式(10.53),并计算这个三次 B 样条的一阶及二阶导数,并证明它们处处连续。

练习 10.7.5 周期性 B 样条曲线

证明下面基于 $L+1$ 个控制顶点及 m 阶 B 样条函数的曲线

$$P(t) = \sum_{k=0}^{L} P_k N_{0,m}((t-k) \bmod (L-1)) \tag{10.55}$$

是封闭的,因此它关于 t 具有周期性。

10.7.2 如何利用节点向量中的多重节点

到现在为止我们还只是讨论了均匀节点下的 B 样条函数。通过改变相邻两个节点之间的距离,曲线设计者更容易控制最终曲线的形状。

核心问题是当两个相邻节点离得很近时,B 样条混合函数会是什么形状呢?图 10.32 给出了节点向量为 $\mathbf{T}=(0,1,2,3,3+\varepsilon,4+\varepsilon,\cdots)$时几个一阶 B 样条函数的形状,这里 ε 是一个很小的正数。

这时每一个分段多项式在区间$[3,3+\varepsilon]$的部分被挤成一小段,每一个 B 样条混合函数不再是其他同阶函数的一个简单平移。若将 ε 设为零,这一小段将不存在,而且节点 $t=3$ 变为多重节点,重数为 2。

图 10.33 给出了二重节点下 B 样条混合函数的形状,其中两个线性 B 样条函数是不连续的(图 10.33(b)),一个二次 B 样条函数在重节点 $t=3$ 处一阶导数是不连续的(图 10.33(c))。一般地,在重节点处 i 阶光滑曲线退化为 i−1 阶光滑曲线。三次 B 样条曲

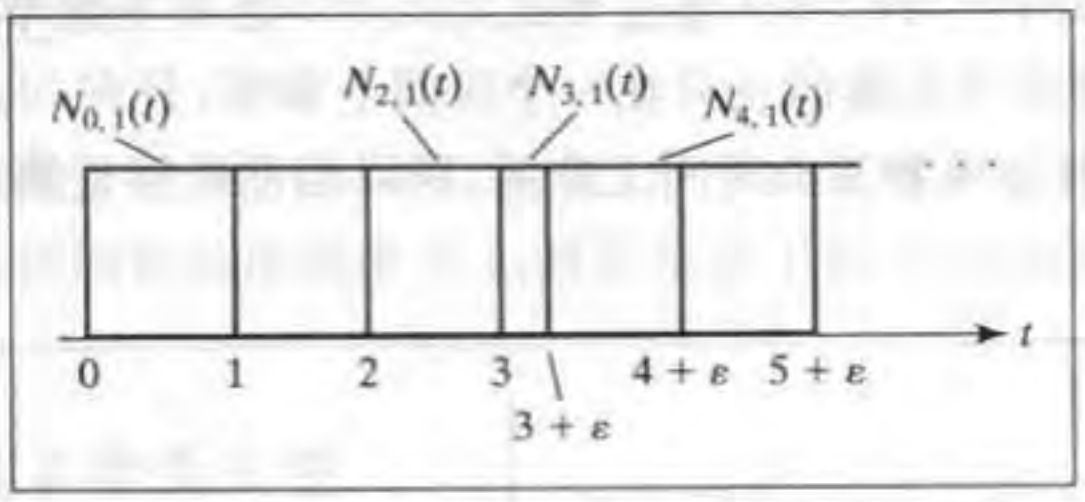


图 10.32　两个相邻节点距离很近时的 B 样函数

线(图 10.33(d))处处一阶光滑，但在 $t=3$ 处非二阶光滑。但是注意到如果我们用图 10.33(c)中的二次 B 样条函数去构造曲线，则曲线将会插值控制顶点 P_2，因为函数 $N_{2,3}(t)$在 $t=3$ 处值为 1，而其他函数在 $t=3$ 处值均为 0。一般地，当 t 接近一个重数大于 1 的节点时，在相应控制顶点处会有很多重要的性质。

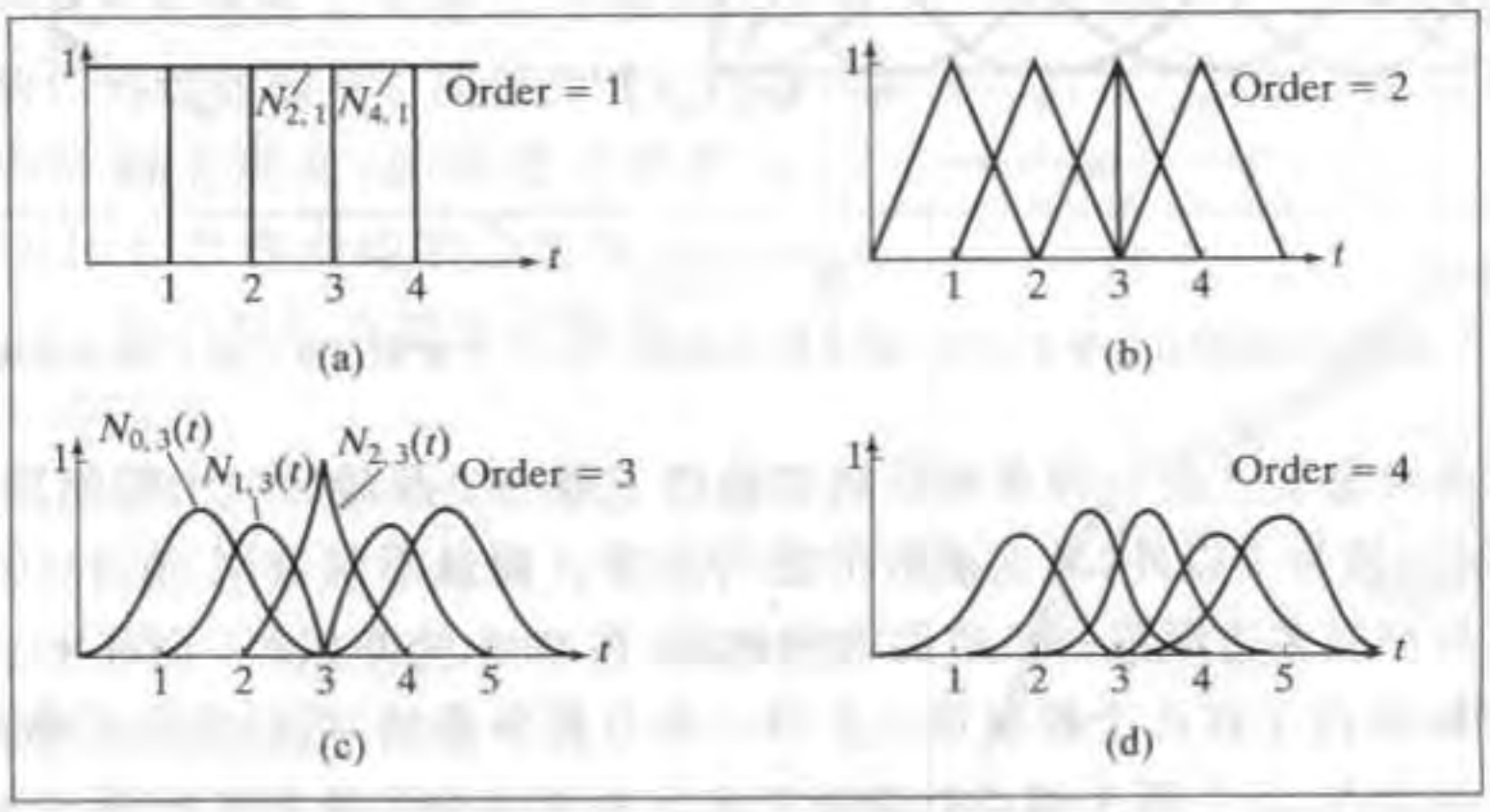


图 10.33　二重节点下 B 样条的形状

另外，二次样条在三重节点处不连续，三次样条在三重节点处一阶导数不连续，但是它们仍然插值其中一个控制顶点。通过调整每一个节点的重数，曲线设计者可以很好得控制曲线的形状。

不过当节点向量包含重节点时，无论等式(10.47)还是图 10.31 中的代码部分都不需要改变，至于等式(10.47)我们已经提到会出现分母为零的情况，图 10.31 中的代码也无需修改，因为它会自动处理这种情况的。

10.7.3　非封闭 B 样条曲线：标准节点向量

在曲线设计中，一类特殊的节点向量已成为标准。在此节点向量下，曲线插值第一个和最后一个控制顶点，使得设计者很容易预测曲线的大致位置。

m 阶 B 样条曲线的标准节点向量为首末两个 m 重节点，其余节点距离相差 1，我们先看一个例子再看这个节点向量是怎么提出的。假设有八个控制顶点，我们用三次($m=4$)B 样条函数来构造曲线，则标准节点向量为：

$$\mathbf{T}=(0,0,0,0,1,2,3,4,5,5,5,5)$$

B 样条函数 $N_{0,4}(t),\cdots,N_{7,4}(t)$ 通过等式(10.47)定义在此节点向量上,见图 10.34(a)。$N_{0,4}(t)$,$N_{7,4}(t)$ 都是不连续的且只在一个区间上非零,只有 $N_{3,4}(t)$,$N_{4,4}(t)$ 在四个区间上非零,其余都是在两个或者三个区间上非零,所以当曲线接近第一个和最后一个控制顶点时形状被严重扭曲。

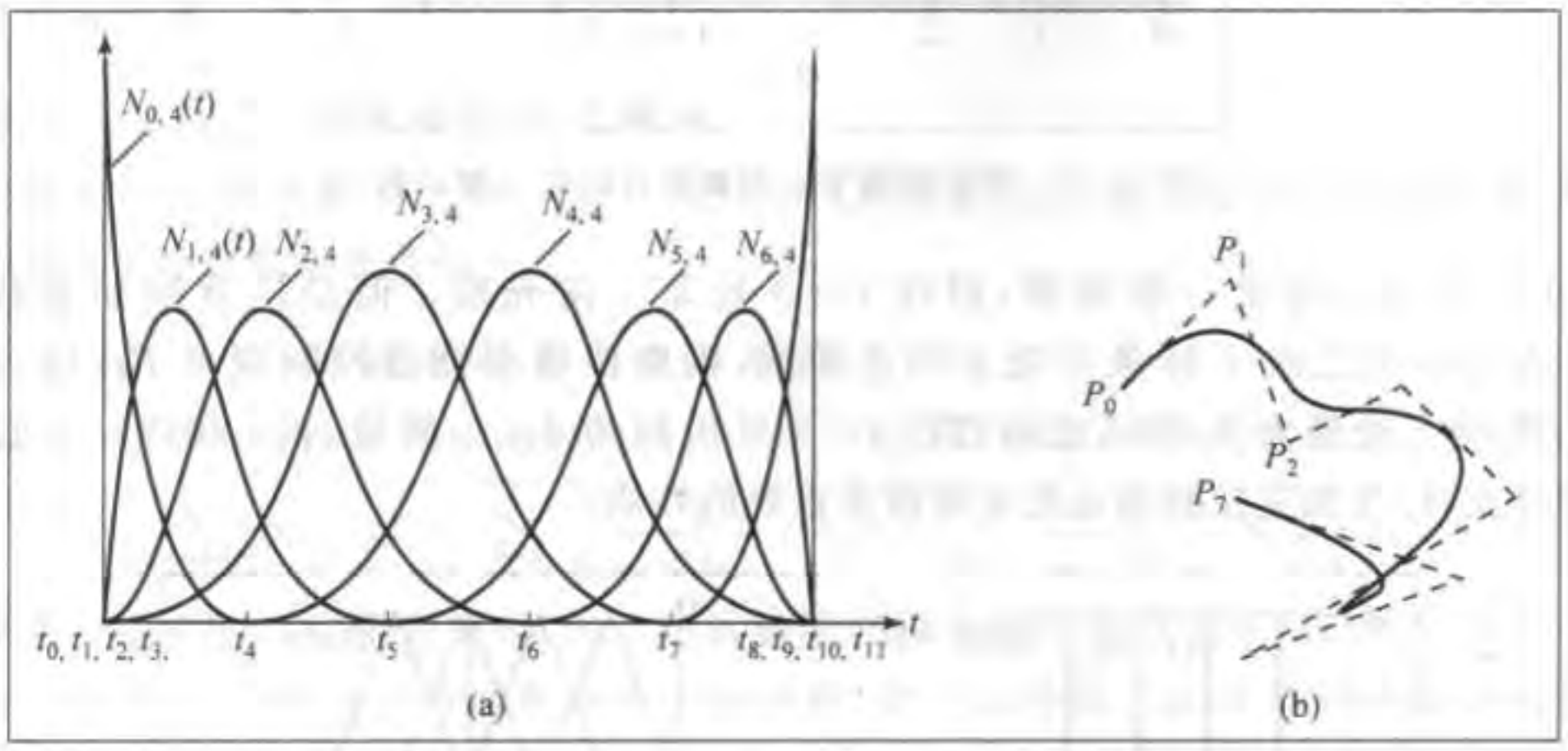


图 10.34

(a) 定义在标准节点向量下 8 个三次 B 样条函数;(b) 八个控制顶点下的 B-样条曲线

但是此节点向量下这些 B 样条函数确实插值了第一个及最后一个控制顶点。比如 $t=0$ 处,除 $N_{0,4}(t)$ 的值为 1 以外,其他函数的值均为零。而且不难证明,在 $t=0$ 处,B 样条曲线的切线方向与控制多边形第一条边的方向相同,在 $t=5$ 处亦如此(见练习)。

图 10.34(b)给出了在八个控制顶点下的一条 B 样条曲线,可以看出它确实插值第一个和最后一个控制顶点,且在两个端点处切线方向与控制多边形的方向相同。

注意到当控制多边形自交时,B 样条曲线也会出现自交。

$L+1$ 个控制顶点下的 m 阶 B 样条曲线的标准节点向量应满足如下条件:

(1) 首先要有 $L+m+1$ 个节点,记为 $t_0,\cdots,t_{L+m}$。

(2) 前 m 个节点即 $t_0,\cdots,t_{m-1}$ 的值均为零(第一个 m 阶 B 样条函数从 $t=0$ 开始定义)。

(3) 节点 $t_m,\cdots,t_L$ 的值从 1 开始依次增加 1,直到 $L-m+1$(最后一个 m 阶 B 样条函数 $N_{L,m}(t)$ 从 $t_L=L-m+1$ 开始定义,只在一个区间上值非零)。

(4) 最后 m 个节点即 $t_{L+1},\cdots,t_{L+m}$ 的值均为 $L-m+2$。

有了上述规则,给定值 m 和 L,就很容易编写程序来产生一个标准节点向量,代码如下:

```
void buildKnots(int m, int L, double knot[])
{
    // 建立 L+1 个控制顶点下的 m 阶 B-样条曲线的标准节点向量
    int i;
    if(L<(m-1)) return;                    // 控制顶点太少
    for(i=0; i<=L+m; i++){
        if (i<m) knot[i]=0.0;
        else if (i<=L) knot[i]=i-m+1;      // i 不小于 m
```

```
        else knot[i]=L-m+2;}                    // i 大于 L
}
```

注意代码中的判断条件取决于 m 和 L 的值，对于给定的阶数 m 必须有一定数目的控制顶点使得曲线的定义区间的长度至少为 1，所以便有了如下的约束条件：阶数 m 不能超过控制顶点的个数 $L+1$。

贝塞尔曲线也是 B 样条曲线

前面我们通过两种方式引入了贝塞尔曲线，即：德卡斯特里奥算法和 Bernstein 多项式。现在我们给出第三种方式。贝塞尔曲线也是一特殊的 B 样条曲线，这是因为当 $m=L+1$ 时定义在标准节点向量上的 B 样条函数其实是一 Bernstein 多项式，即：

$$N_{k,L+1}(t) = B_k^L(t), \quad k = 0,\cdots,L$$

我们需要注意的是当阶数 m 增加到 $L+1$ 时，标准节点向量会发生什么样的变化（见练习），前面 m 个节点的值为 0，后面 m 个节点的值为 1，t 只在区间[0,1]上变化。例如，如果 $L=5$，$m=6$，我们得到标准节点向量为 $\mathbf{T}=(0,0,0,0,0,0,1,1,1,1,1,1)$，这样每个分段多项式只在一个区间上定义，且多项式的阶 $m=L+1$，这就是为什么 B 样条函数会变为 Bernstein 多项式。事实上，我们也可直接通过等式(10.47)得到 Bernstein 多项式。

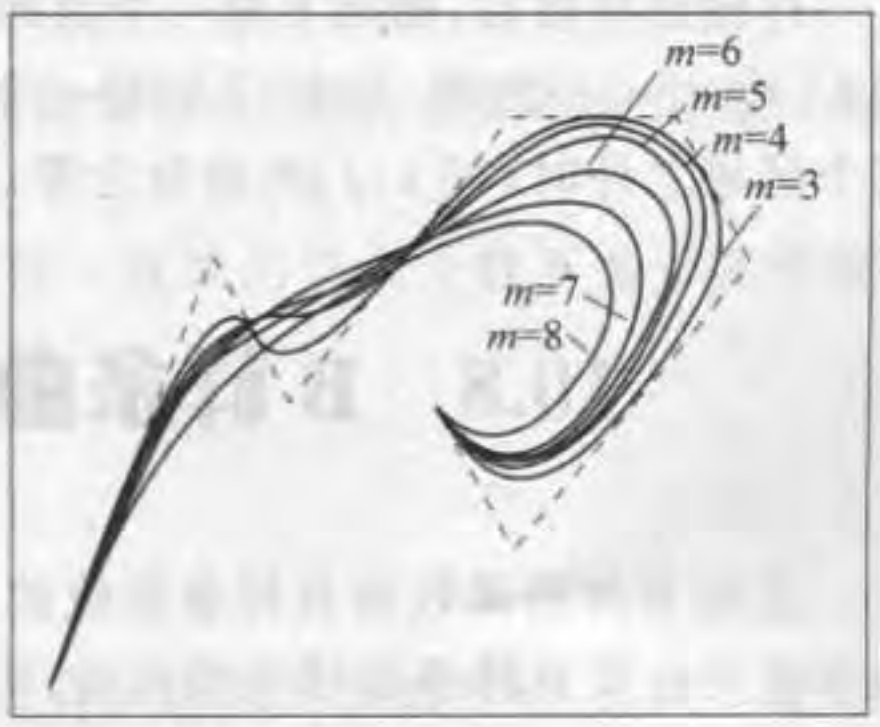


图 10.35　不同阶数的 B 样条曲线

我们最初引入 B 样条曲线的原因是贝塞尔曲线不能很好地对曲线进行局部控制与调整。当 B 样条函数的阶数增加 1 时，它的非零区间就增加 1 个，曲线局部控制就变得不那么灵活，当 m 增加到 $L+1$ 时，就出现了 B 样条曲线的情况，失去了局部控制灵活性。图 10.35 给出了阶数逐渐增加时 B 样条曲线的变化。图中有八个控制顶点，所以当阶数 $m=8$ 时，曲线就变成了贝塞尔曲线。图中的曲线都是由代码 10.31 产生的。（$m=2$ 时的曲线是？）

练习

练习 10.7.6　二次 B 样条曲线的标准节点向量

八个控制顶点下的三阶 B 样条曲线的标准节点向量为 $\mathbf{T}=(0,0,0,1,2,3,4,5,6,6,6)$。

练习 10.7.7　当阶数 m 增加时的标准节点向量

当阶数分别为(a)$m=3$；(b)$m=4$；(c)$m=5$；(d)$m=6$；(e)$m=7$ 在七个控制顶点下的标准节点向量分别是？

练习 10.7.8　标准节点向量下的二次 B 样条曲线

通过等式(10.47)，计算出在标准节点向量下二次 B 样条函数的显式表达式（分段多项式的形式）

练习 10.7.9 三次 B 样条

计算出前四个标准节点向量下的三次 B 样条函数,并求出曲线在 $t=0$ 时的导数,看它与控制多边形的第一条边的方向是否相同?

练习 10.7.10 导出 Bernstein 多项式

当 $m=L+1=4$ 时,通过等式(10.47)求得的 B 样条函数就是 Bernstein 多项式。

练习 10.7.11 B 样条的高效计算

当绘制一条 B 样条曲线时,需要产生许多点,也就是要计算出 $N_{k,m}(t)$ 在很多不同的 t 点处的值。用图 10.58 递归的方法计算会导致许多多余的计算量即低阶 B 样条函数的值。这里我们提供一个高效的方法,每个 $N_{k,m}(t)$ 的值只需计算一次,然后把它们存为数组,比如 $N[k,i]$ 中保存 $t=i\Delta t$ 处计算得到的 $N_{k,m}(t)$ 的值,其中 Δt 可以按照需要设定。这样就只需要在 B 样条函数的非零区间上计算即可,而且只需要计算几种不同型形状的 $N[k,i]$,因为其他都是某个函数平移得到的。那么若有 55 个控制顶点,三次 B 样条函数有多少种不同的形状呢?

存储这些值后,曲线上每一个点的坐标都可以通过访问数组的某个值得到:$P(t_i)$ 是各项 $C[k]N[n,j]$ 的和,其中 $C[k]$ 是一个控制顶点,需要选择适当的 n,j 的值。注意当 j 处于某个区域之外时,$N[k,j]$ 的值全为零,就不需要再访问这个数组了。

10.8 B 样条曲线在设计中的重要性质

总结 B 样条函数和 B 样条曲线的基本性质是很重要的。我们应该知道贝塞尔曲线的很多重要性质 B 样条曲线也都具备。

(1) m 阶 B 样条函数事实上就是一 m 阶分段多项式,且它们是 $m-2$ 阶光滑的,也就是说它们的 $m-2$ 阶导数是连续的。它们形成一组样条基,就是说任何样条曲线都可以表示为它们的线性组合。在所有的样条基中,B 样条基是局部支柱性最好的,也就说 B 样条基的非零区间是最短的。

(2) B 样条混合函数 $N_{k,m}(t)$ 的非零区间为 $[t_k,t_{k+m}]$,起始于 t_k,终止于 t_{k+m}。函数集 $\{N_{k,m}(t)\mid k=0,1\cdots,L\}$ 的非零区间为 $[t_0,t_{m+L}]$。

(3) 由等式(10.55)可以得到一条 $L+1$ 个控制顶点下的封闭 B 样条曲线(等式(10.55)是在节点均匀分布条件下的)。

(4) 当节点取为标准节点向量时,B 样条曲线会插值第一个及最后一个控制顶点,且在两个端点处的切线方向分别与控制多边形的第一条边及最后一条边的方向相同。

(5) B 样条函数 $N_{k,m}(t)$ 在任意点 t 处都是非负的。且对任意的 $t\in[t_0,t_{m+L}]$,均有

$$\sum_{k=0}^{L} N_{k,m}(t) = 1 \tag{10.56}$$

前面我们已经证明了低阶的情况,高阶的情况可以由等式(10.47)归纳得到。

(6) B 样条曲线具有仿射不变性。要想变换一条 B 样条曲线,只需同样变换所有控制顶点即可,这是因为 B 样条曲线是所有控制顶点的仿射组合。

(7) 根据性质 5，B 样条曲线是它的所有控制顶点的凸组合，所以整条曲线落在控制顶点集的凸包内。但是注意到在任意点 t 处，只有 m 个 B 样条函数是非零的，所以在 t 处，曲线落在 m 个连续的活动控制顶点组成点集的凸包内。图 10.36 给出了在标准控制顶点下的一个二次 B 样条曲线，在任意点 t 处最多只有 3 个控制顶点是活动的，所以其凸包是三角形。随着 t 的逐渐增加，$P(t)$ 渐进地跳出一个三角形凸包进入下一个，因为 B 样条函数是轮流活动的。请问图中曲线进入及离开阴影凸包的 t 值分别是多少？

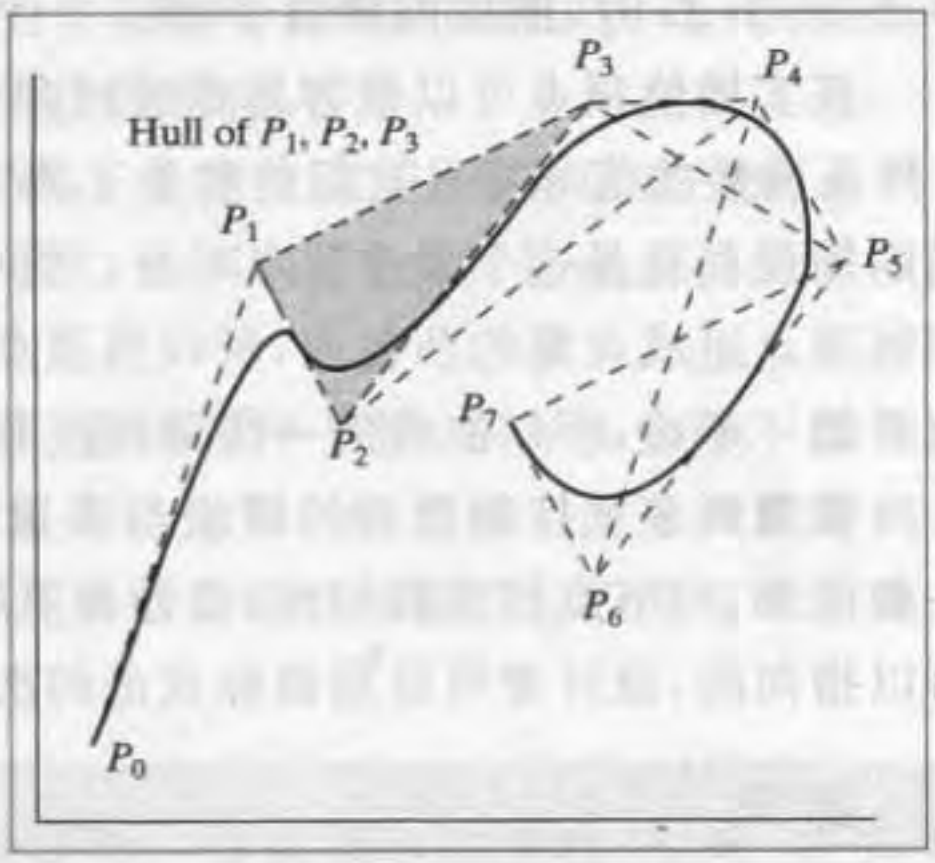


图 10.36　二次 B-样条曲线的凸包

m 个控制顶点组成点集的凸包比所有控制顶点组成点集的凸包要小得多，所以与贝塞尔曲线相比，B 样条曲线被限制在了一个更小的区域，这个性质使得曲线设计者更容易局部调节控制曲线的形状。

(8) B 样条曲线可以呈现直线型：如果 m 个连续控制顶点共线，它们的凸包是一条直线，所以曲线呈直线型。

(9) B 样条曲线具有变差缩减性：B 样条曲线与任一直线的交点个数不多于其控制多边形与该直线的交点个数【Farin88】。

10.8.1　多重控制顶点

曲线设计者还可以利用多重控制顶点更有效地调节 B 样条曲线的形状，所谓多重控制顶点就是使几个连续控制顶点重合于一点，这会使得曲线更加逼近控制多边形。

图 10.37 给出了三次 B 样条曲线的例子。当顶点 D 为单重顶点即控制顶点为 A、B、C、D、E、F、G 时，曲线呈现出通常三次 B 样条的形状；当顶点 D 为二重顶点即控制顶点为 A、

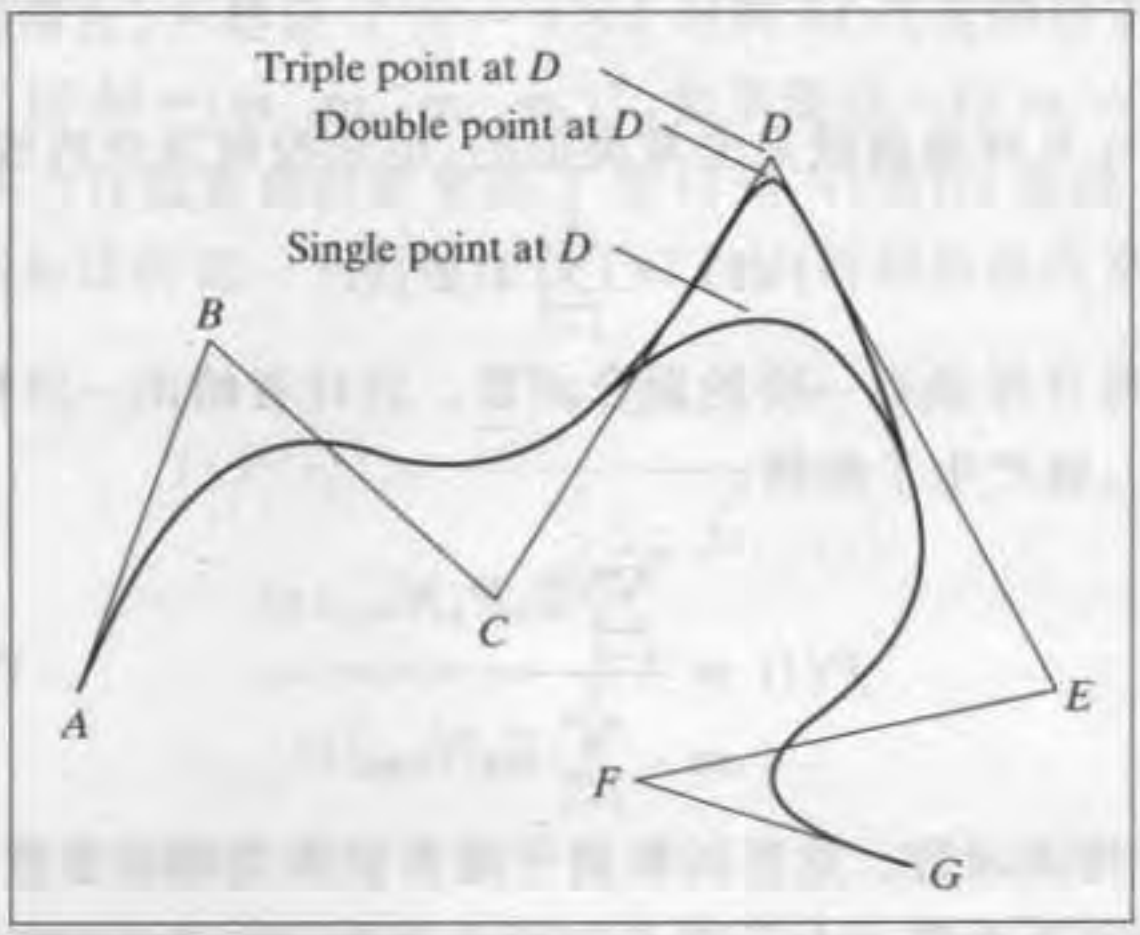


图 10.37　多重控制顶点下的 B-样条曲线

B、C、D、D、E、F、G 时,曲线更加靠近 D 点;当顶点 D 为三重顶点即控制顶点为 A、B、C、D、D、D、E、F、G 时,曲线则插值 D 点。

其实插值顶点可以很容易地通过图 10.34(a)来解释,注意到在节点 t_5,t_6 处,只有三个 B 样条函数的值非零且它们的和为 1,所以如果这三个相应的控制顶点重合为一点,那么它们的加权和就是这个多重顶点本身。更一般地,一个三次 B 样条曲线会被限制在四个连续控制顶点组成点集的凸包内,所以当顶点 D 为三重顶点时,包含 D 的凸包就变成了控制多边形的一条边,所以曲线的一段落在控制多边形的那两条边上,插值了多重顶点 D。

注意到多重控制顶点的用法与多重节点的用法是不同的,尽管它们的效果是一样的。一般说来,与节点的重数相比,设计者更容易增加控制顶点的重数,因为控制顶点是可见且可以指向的,设计者可以用鼠标点击的次数来指定控制顶点的重数。

练习

练习 10.8.1 凸包性质

画出几种八个控制顶点组成的控制多边形,假设二次 B 样条函数定义在标准节点向量下,画出其连续凸包,并计算出曲线进入和退出每个凸包的 t 值。然后大致勾画出 B 样条曲线,确保它通过其控制多边形内部边的中点。

练习 10.8.2 多重控制顶点下的二次 B 样条曲线

解释双重控制顶点在二次 B 样条曲线中的作用。先画出一个没有多重顶点的控制多边形,画出它的每个相应凸包,大致勾画出 B 样条曲线。然后逐次增加其中一个控制顶点的重数后重复上述过程。

10.9 有理曲线和非均匀有理 B 样条曲线

在 10.2 节我们大致介绍了贝塞尔曲线的有理参数形式,知道了圆锥曲线(如椭圆,双曲线,抛物线)可以由有理二次贝塞尔曲线(等式(10.14))精确产生,这里我们详细介绍有理 B 样条曲线。

有理 B 样条曲线与 B 样条曲线是非常类似的,也是控制顶点的线性组合,即:

$$P(t) = \sum_{k=0}^{L} P_k P_k(t) \tag{10.57}$$

但是有理 B 样条使用与 B 样条不一样的混合函数。设计者给出一组权因子$\{\overline{\omega_0},\overline{\omega_1},\cdots,\overline{\omega_L}\}$和控制顶点 $P_0,\cdots,P_L$,就产生了曲线:

$$P(t) = \frac{\sum_{k=0}^{L} \overline{\omega}_k P_k N_{k,m}(t)}{\sum_{k=0}^{L} \overline{\omega}_k N_{k,m}(t)} \tag{10.58}$$

其中 $N_{k,m}(t)$是 k 阶 B 样条函数。这里的权因子通常被称为形状参数,且通常被设为非负以保证等式(10.58)中分母不为零。由于节点向量通常是非均匀分布的,所以此时的曲线被称为非均匀有理 B 样条(即:NURBS)。

观察等式(10.58),$P(t)$的分子是所有控制顶点的线性组合。在第 4 章我们已经知道,为了使得它有意义,我们需要把它表示成控制顶点的仿射组合,所以同除以分母,使得所有系数的和为 1,这样 $P(t)$就表示成了所有控制顶点的仿射组合。

由等式(10.58)我们还可以看出,如果其中的权因子是完全相同的,分母就变成了一个常数(为什么?),$P(t)$就退化为最初的 B 样条曲线等式(10.46)了,也就是说当权因子不同时,$P(t)$就是等式(10.46)的推广。

我们再来看一下这里的混合函数是怎么得来的。像练习中一样,在齐次坐标下给每个控制顶点一个权因子,即给顶点 P_k 权因子 ω_k,然后将其转化为一般坐标下的形式就得到了 NURBS 曲线,NURBS 曲线通过控制顶点的权因子在高维空间中保留了它们原有的性质。

NURBS 曲线的两大重要优点

1. 通过合适地选择控制顶点和权因子,$P(t)$就成为一条精确圆锥曲线,见例 10.9.2。这不同于非有理 B 样条曲线,它只能逼近而不能精确得到圆锥曲线。

2. NURBS 曲线在投影变换下保持不变,所谓投影变换我们已经在第七章中讨论过了,它是仿射变换的推广。仿射变换矩阵的第四行固定为(0,0,0,1),但是投影变换矩阵的第四行更一般化。一般的 B 样条曲线具有仿射变换不变性,所谓仿射变换就是旋转,伸缩,平移以及它们的线性组合,而 NURBS 曲线在更宽范围的变换下有不变性。

这就是说如果想得到 NURBS 曲线在某个方向上的投影,只需要先得到每个控制顶点在此方向上的投影,然后通过等式(10.57)得到新的曲线就是原曲线在此方向上的投影(当然权因子也要做相应的调整),这样比找出曲线上每一个点的投影要简单得多。但是相应地 B 样条就不能这样,因为它只在仿射变换下具有不变性。

投影不变性

这里我们来详细介绍一下 NURBS 曲线的投影不变性。投影不变性就是当一个 NURBS 曲线通过一个 4×4 矩阵变换时,其结果仍然是一 NURBS 曲线,它的控制顶点是原曲线的控制顶点通过上面 4×4 矩阵变换得到的,当然这里的权因子要作适当调整。

我们来考虑 3D 的情况,不妨设 T 为一 4×4 矩阵 M 代表的投影变换,$\mathbf{m}_1$、$\mathbf{m}_2$、$\mathbf{m}_3$、$\mathbf{m}_4$ 分别代表 M 的每一行,即 $M=(\mathbf{m}_1,\mathbf{m}_2,\mathbf{m}_3,\mathbf{m}_4)^T$,如果最后一行 $\mathbf{m}_4=(0,0,0,1)$,则 T 就是仿射变换。所谓投影不变性就是将投影变换 T 作用到 NURBS 曲线 $P(t)$的每一个点上即 $T(P(t))$与将它作用到曲线的每一个控制顶点上 $T(P_k)$得到的曲线是相同的【Pieg191】。

$$T(P(t))=\frac{\sum_{k=0}^{L}\overline{\omega}_k T(P_k)N_{k,m}(t)}{\sum_{k=0}^{L}\overline{\omega}_k N_{k,m}(t)} \tag{10.59}$$

其中权因子调整为:

$$\overline{\omega}_k=\omega_k(\widetilde{P}_k\cdot\mathbf{m}_4) \tag{10.60}$$

$\widetilde{P}_k=(P_x,P_y,P_z,1)^T$ 为控制顶点 P 的齐次坐标形式,齐次坐标为 1。注意到权因子的调整依赖于控制顶点的位置,即矩阵 M 的第四行,所以如果 T 是一仿射变换,则 $(P_k\cdot\mathbf{m}_4)=1$,权因子不用调整。

练习

练习 10.9.1 一般曲面如何通过 NURBS 曲线产生?

讨论以下曲面如何通过 NURBS 曲线产生?

a. 圆柱

b. 圆锥

c. 圆盘

d. 球

详细参考案例 10.10。

练习 10.9.2 当权因子完全相同时会如何?

当 NURBS 曲线的权因子完全相同时,它就退化一 B 样条曲线,这时 NURBS 曲线被一矩阵 M(M 是一个 4×4 矩阵)变换后是否还是一 NURBS 曲线? 是否还是一 B 样条曲线?

练习 10.9.3 在齐次坐标下产生 NURBS 曲线

对一 B 样条曲线,我们给曲线的每一个控制顶点 P_k 加上一个权因子 ω_k,则曲线在齐次坐标下表示为:

$$\widetilde{R}(t)=\sum_{k=0}^{L}\omega_k\widetilde{P}_kN_{k,m}(t) \tag{10.61}$$

这里 $\widetilde{P}_k$ 表示顶点 P_k 的齐次坐标形式(即:$\widetilde{P}_k=(P_x,P_y,P_z,1)^T$)。

a. 分别写出 $\widetilde{R}(t)=(x(t),y(t),z(t),\omega_k)^T$ 的每一个组成部分即 $x(t),y(t),z(t),\omega_k$,并注意观察 ω_k 的形式。

b. 除以 ω_k,把上述 $\widetilde{R}(t)$ 转化为一般坐标下的形式,得到:

$$P(t)=\frac{\sum_{k=0}^{L}\omega_kP_kN_{k,m}(t)}{\sum_{k=0}^{L}\omega_kN_{k,m}(t)} \tag{10.62}$$

c. 可以看出它与等式(10.57)或等式(10.58)的形式是完全相同的。

例 10.9.1 怎样得到圆锥曲线?

对于等式(10.14)的特殊情况,令 $m=3$(二次 B 样条),并选择合适的节点向量使得B 样条为 Bernstein 多项式(如何选择?),然后令权因子 $\omega_0=\omega_2=1,\omega_1=\omega$,这样就可以得到等式(10.14)。

例 10.9.2 怎样得到一个精确的圆?

由 NURBS 曲线可以通过多种方式精确得到一个圆【Pieg189】,我们在这里给出怎样通过七个控制顶点下的二次 B 样条得到圆。图 10.38(a)给出了控制顶点 $P_0,\cdots,P_6$ 围成一个

正方形，圆就围绕着这个正方形产生，权因子的值也在图中标出，它们要么为 1，要么为$\frac{1}{2}$，节点向量为 $T=\left\{0,0,0,\frac{1}{4},\frac{1}{2},\frac{1}{2},\frac{3}{4},1,1,1\right\}$，注意这里 $t=\frac{1}{2}$为双重节点。图 10.38(b)给出了七个基函数，其中所有 $R_k(t)$的共同分母的曲线图中用虚线画出。本节后面练习给出了确定每个基函数在非零区间内实际等式的过程，并验证在所有 t 处均有$x^2(t)+y^2(t)=1$。

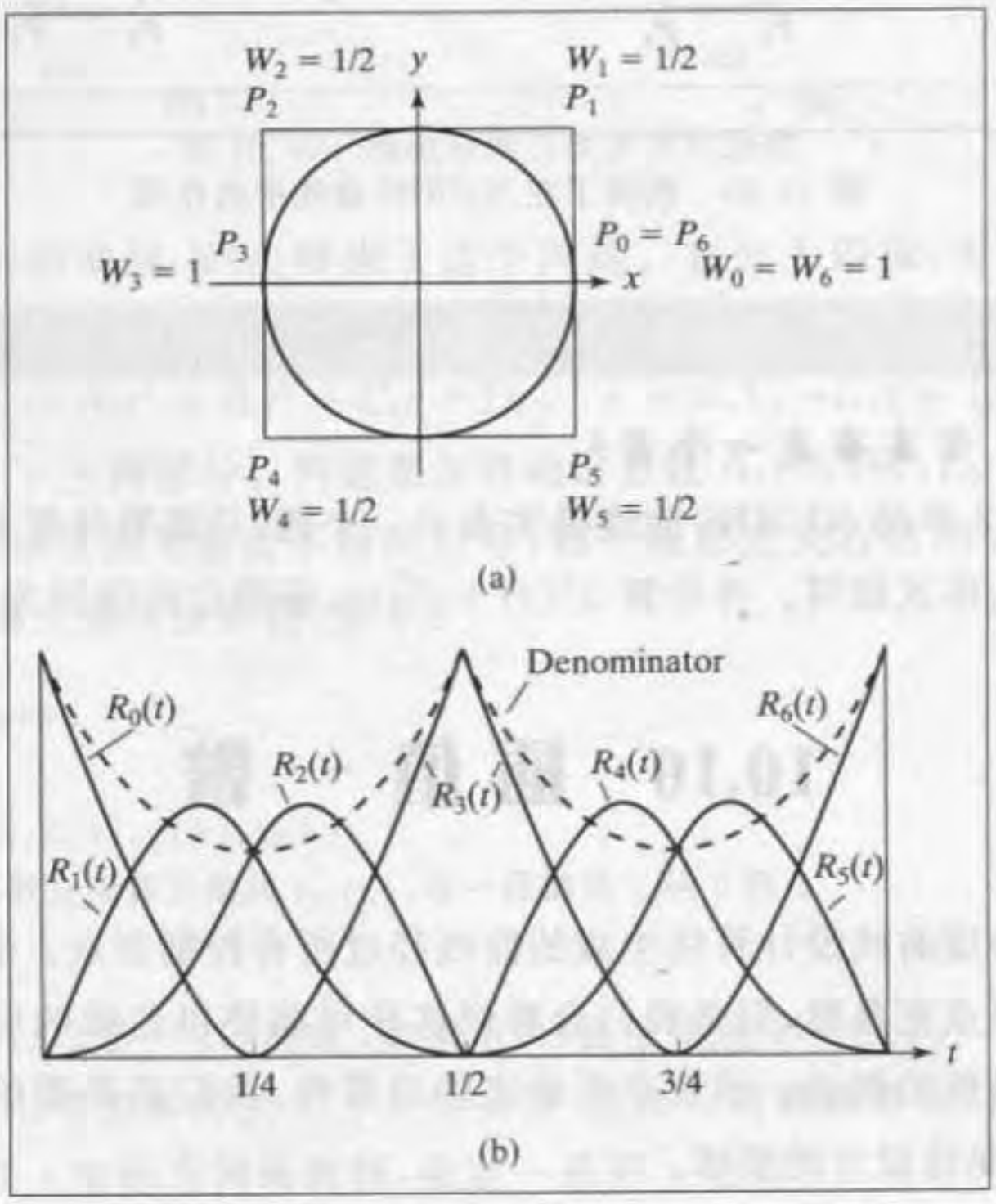


图 10.38　通过多重控制顶点调整曲线

(a) NURBS 曲线得到圆；(b) 对应的基函数

例 10.9.3　权因子的作用

下面图 10.39(a)给出了 6 个控制顶点下的三次 B 样条曲线(在均匀节点下)，它是通常情况下的非有理 B 样条(等价于权因子相同时的 NURBS 曲线)。图 10.39(b)显示出了当权因子不一样时曲线发生的变化，即曲线更接近于权因子较大的控制顶点(当权因子为零时会怎样呢?)。

NURBS 曲线的普遍性和灵活性使得它在曲线设计中越来越流行。由于它包含了 B 样条曲线这种特殊情况，所以一个单一的曲线算法可以用于生成较宽范围的曲线族，包括圆锥曲线。所以设计者只需一个算法即可，而不需要一个包含多种算法的库。

在曲线设计中还存在许多其他方法，其中一些可以让用户更好的控制曲线的形状。比如用户可以改变节点之间的距离或者增加控制顶点等来控制曲线的形状。读者如果对此感兴趣可以上网搜索关于 B 样条和 NURBS 曲线的其他资源。

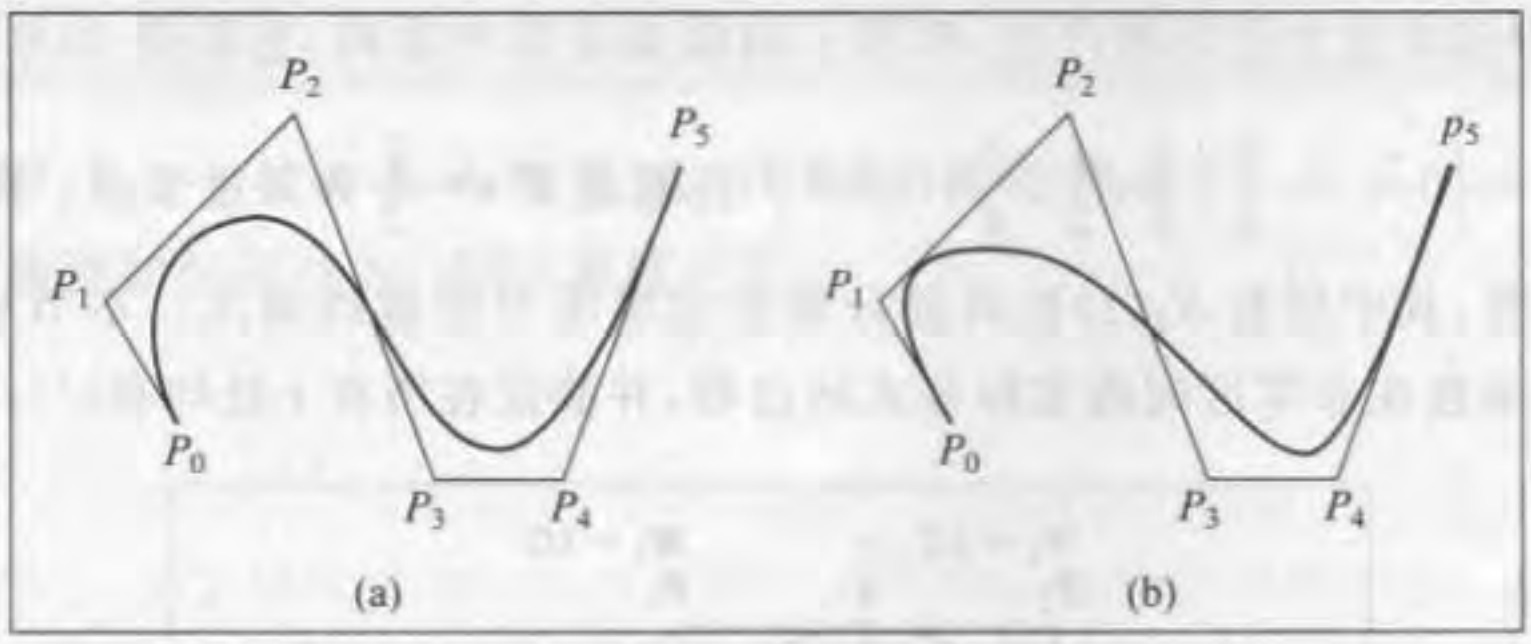


图 10.39 权因子在 NURBS 曲线中的作用

练习

练习 10.9.4 它是否是一个圆?

证明例子 10.9.2 中的 NURBS 曲线确实表示一个圆,只需要计算出在每个节点区间上基函数的多项式表达形式即可。并计算 $x^2(t)+y^2(t)$,证明它的值恒为 1。

10.10 插值一瞥

有时候设计者希望曲线设计算法生成的曲线经过所有控制顶点。表面上好像这样比让曲线逼近每个控制顶点更自然,但是我们会看到这样可能使得曲线的形状变得很局限或者控制顶点有不希望出现的摆动。尽管存在着这些局限性,我们还是要给出各种情况下的插值算法,满足现实中曲线设计的要求。

设计者已经有了一种通过 B 样条得到的插值算法,即:给定每一个控制顶点足够大的重数,就可以使得得到的 B 样条曲线插值它们。

在案例 10.6 中我们将看到另外一种插值方法,它也是建立在 B 样条基础上,设计者给出一系列点,此算法将会产生另外一系列顶点,以此作为控制顶点产生的 B 样条曲线将会插值设计者最初给出的点。

这一节我们大致介绍一下曲线插值算法,即给定一系列点,如何得到一曲线使得它经过这些点。这是一个很早就提出的问题,我们在这里仅介绍一部分算法,Farin【Farin88】详细讨论了曲线插值这个问题。

10.10.1 利用分段三次多项式插值

这里我们介绍如何利用分段三次多项式来插值。我们只用分段三次多项式是因为它虽然简单但是足够灵活有效。

问题:图 10.40(a)解释了这个基本问题,即:用户给定一系列控制顶点 $P_0, P_1, \cdots, P_L$(这里 $L=4$),得到一条足够光滑的曲线依次经过这些点。

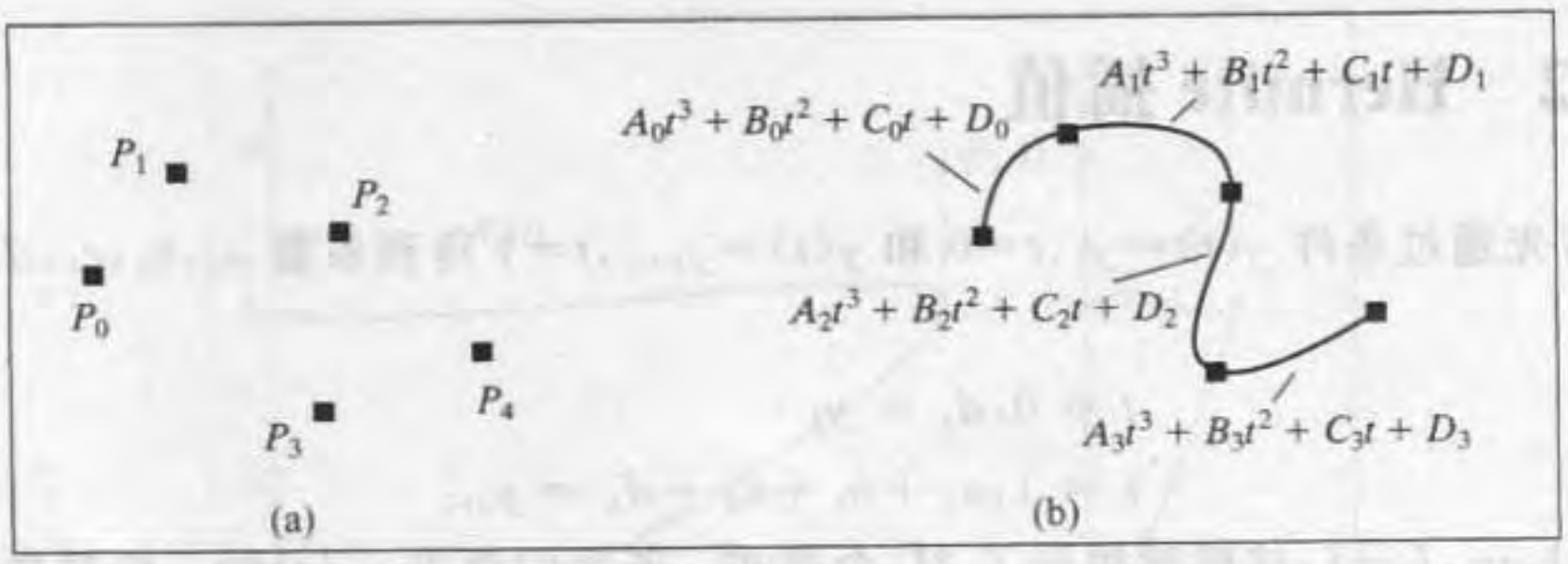


图 10.40　利用分段三次多项式插值

图 10.40(b)所示的曲线 $R(t)$ 解决了这个问题。它分为四段，每段都是一个三次多项式：

$$R_k(t) = A_k t^3 + B_k t^2 + C_k t + D_k, \quad k = 0,1,\cdots L, t \in [0,1] \tag{10.63}$$

当然每段都包含 x、y 两部分。问题是怎样确定系数 A_k, B_k, C_k, D_k 呢？

这里我们为了简单采用与前面不同的记号：每一段都定义在区间 $t \in [0,1]$ 上，这样画这条曲线时，就需要独立画出这四段，如下：

```
for(k=0; k<L; k++)
{
    求出系数 A[k],B[k],C[k],D[k]
    根据求得的系数分段画出曲线 Rk(t),每一段都是 t 从 0 到 1
}
```

$R_k(t)=(x(t),y(t))$ 是一 2D 曲线，如果 $R(t)$ 插值顶点 $P=(x,y)$，则 $x(t)$ 插值 x，$y(t)$ 插值 y。所以我们可以考虑如何使 x、y 分别插值，这样问题就被简化，且形象化了。

图 10.41(a)就以 y 坐标为例来解释，给定一系列 y 值即 $y_0, y_1, \cdots y_L$，我们要得到一个分段三次多项式 $y(t)$ 经过这些 y 值且一阶光滑(有时候还会要求二阶光滑，也就是说是一样条)。

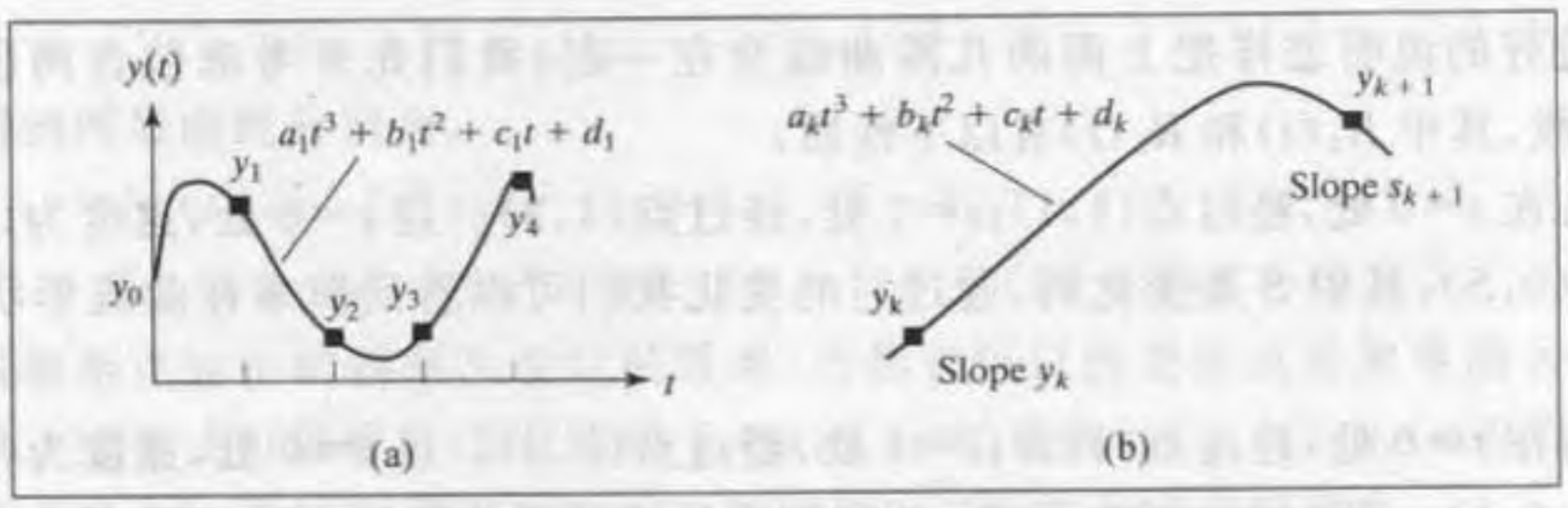


图 10.41　插值曲线的 y 部分

图 10.41(b)画出了曲线的第 k 段，它表示为：

$$y_k(t) = a_k t^3 + b_k t^2 + c_k t + d_k, \quad k = 0,1,\cdots L, t \in [0,1] \tag{10.64}$$

我们用 s_k 来表示它的一阶导数，即 $y_k'(0)=s_k$，有时候 s_k 是给定的，有时则是通过曲线需要满足的其他性质计算出来的。再问一次，系数如何确定？

10.10.2 Hermite 插值

我们首先通过条件 $y(t)=y_k, t=0$ 和 $y(t)=y_{k+1}, t=1$ 得到系数 a_k, b_k, c_k, d_k 应满足以下条件：

$$t=0: d_k=y_k$$
$$t=1: a_k+b_k+c_k+d_k=y_{k+1}$$

其中 $k=0,1,\cdots,L-1$，这样就得到了 $2L$ 个等式。若我们要求 $y_k(t)$ 的一阶导数 $y_k'(t)$ 在端点处等于给定的值，即满足条件 $y_k'(t)=s_k, t=0$ 和 $y_k'(t)=s_{k+1}, t=1$，因为 $y_k(t)$ 的导数为 $y'_k(t)=3a_kt^2+2b_kt+c_k$，得到：

$$t=0: c_k=s_k$$
$$t=1: 3a_k+2b_k+c_k=s_{k+1}$$

其中 $k=0,1,\cdots,L-1$，又得到了 $2L$ 个等式。这样对 $4L$ 个未知系数我们就得到了 $4L$ 个等式，而且通过给定每一段端点处的一阶导数使得曲线在每一段接点处保持一阶光滑。

通过上面 $4L$ 个等式我们可以直接求解得到系数 a_k, b_k, c_k, d_k（通过 y_k, s_k 表示）：

$$a_k=s_{k+1}+s_k-2(y_{k+1}-y_k)$$
$$b_k=3(y_{k+1}-y_k)-2s_k-s_{k+1}$$
$$c_k=s_k$$
$$d_k=y_k$$

其中 $k=0,1,\cdots,L-1$。这样就确定了每一段 $y_k(t)$，同样可以确定 $x_k(t)$，就可以画出分段三次多项式曲线 $R_k(t)$ 了。

给定每一个 y_k 点的斜率 s_k，曲线的坐标就可以完全确定，方法有多种。我们首先看到了 s_k 在曲线形状中所起的作用，接下来我们来研究怎样使得所得曲线二阶光滑，引出了三次样条。

例 10.10.1 正切和斜率

为了较好的说明怎样把上面的几段曲线合在一起，我们先来考虑包含两段 $R_0(t)$ 和 $R_1(t)$ 的曲线，其中 $R_0(t)$ 和 $R_1(t)$ 有以下性质：

$R_0(t)$：在 $t=0$ 处，经过点(1,1)；$t=1$ 处，经过点(4,3)。且 $t=0$ 处，速度为(1,0)；$t=1$ 处，速度为(0,S)，其中 S 是变化的，通过它的变化我们可以看到斜率在曲线形状中所起的作用。

$R_1(t)$：在 $t=0$ 处，经过点(4,3)；$t=1$ 处，经过点(0,3)。且 $t=0$ 处，速度为(0,S)；$t=1$ 处，速度为(0,1)。图 10.42 给出了 $S=1$ 这种特殊情况下的结果，不仅画出了原始曲线，还分别画出了曲线的四个组成成分 $x_0(t), x_1(t), y_0(t), y_1(t)$。

仔细观察上面两段曲线是怎样合为一段的，我们可以看到两段曲线是光滑过渡的，因为它们的速度是连续的。这是因为 $x(t)$ 和 $y(t)$ 都是连续的，且在接点处有连续的斜率，所以曲线是一阶光滑的。由于 $x(t)$ 的两段在接点处斜率都是零，所以它在接点处是水平地从左端过渡到右端的，$y(t)$ 也是光滑地从左到右的，在接点处斜率都为 1。

为了更清楚地了解每一部分的性质，我们给等式(10.63)以具体的数据，先以 y 一来

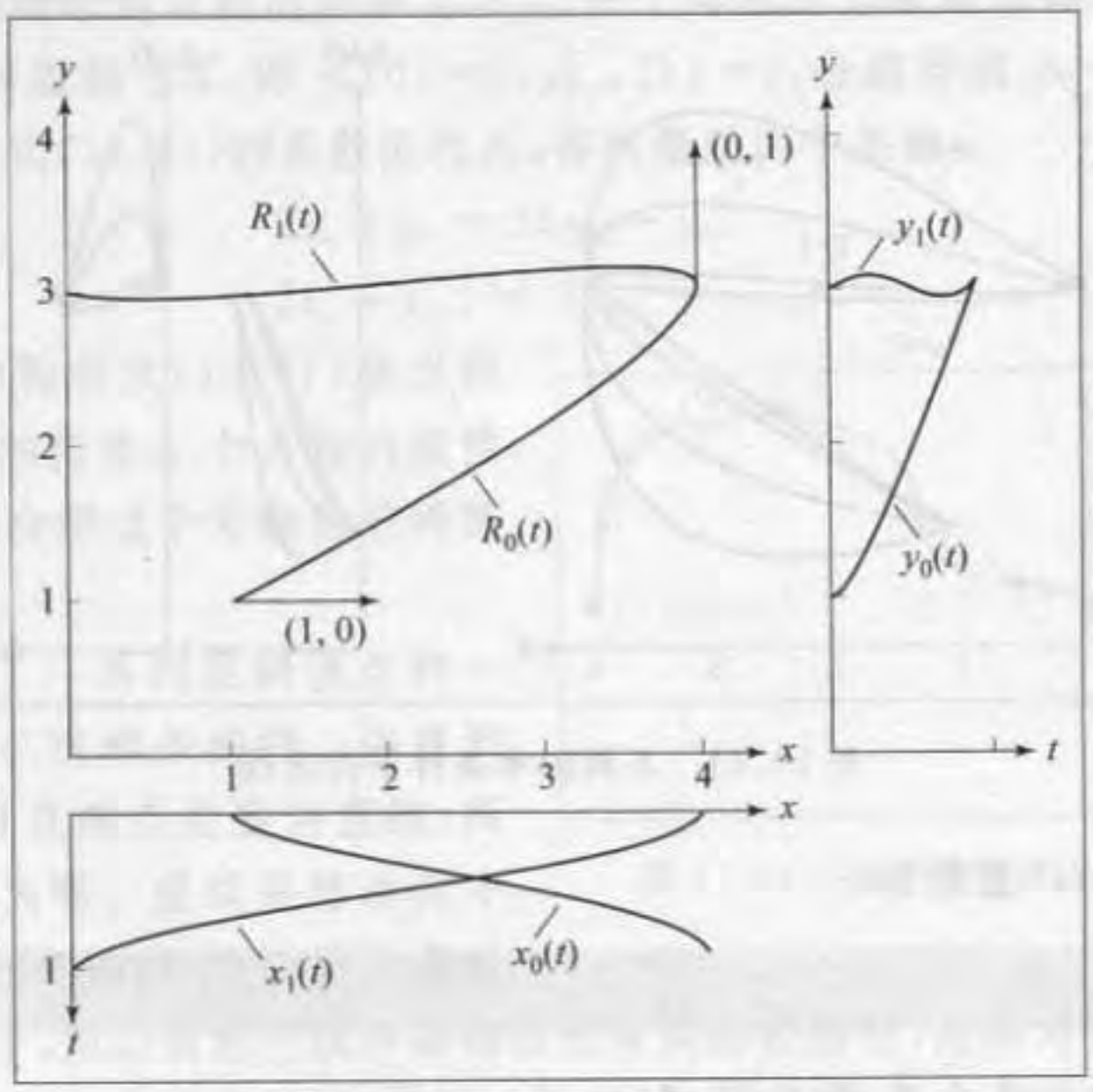


图 10.42　两段三次曲线的合成

说明：

第一段：$y_0=1, y_1=3, s_0=0, s_1=S$，得到三次多项式：

$$y_0(t)=(S-4)t^3+(6-S)t^2+1$$

第二段：$y_1=3, y_2=3, s_1=S, s_2=1$，得到三次多项式：

$$y_1(t)=(S+1)t^3-(2S+1)t^2+St+3$$

对于 x 一同样有：

第一段：$x_0=1, x_1=4, s_0=1, s_1=0$，得到三次多项式：

$$x_0(t)=-5t^3+7t^2+t+1$$

第二段：$x_1=4, x_2=0, s_1=0, s_2=0$，得到三次多项式：

$$x_1(t)=8t^3-12t^2+4$$

这样得到两段曲线分别为：

$$R_0(t)=-5t^3+7t^2+t+1,\quad (S-4)t^3+(6-S)t^2+1$$

$$R_1(t)=8t^3-12t^2+4,\quad (S+1)t^3-(2S+1)t^2+St+3$$

仔细观察接点处 y 的斜率 S 变化的效果，当然它仅仅改变接点处速度的大小，而不改变方向。从上述等式可以看出，S 仅影响 $y_0(t)$，$y_1(t)$ 的系数，对 $x_0(t)$，$x_1(t)$ 没有任何作用。图 10.43 给出了多个不同的 S 下函数 $y(t)$ 的图像以及整条曲线的形状。

从图上可以看出，斜率 S 越大，曲线在接点处的速率就越大。为了适应这个速率的变化，曲线需要在接近及离开接点时凸出以便在垂直方向上能快速通过接点，这也说明了三次曲线的“弹性”程度：因为没有足够的自由度，不能任意快的弯曲，所以会调节整个区间上的形状以满足端点处的斜率约束。

当 $S=0$ 时，曲线在接点处产生一个角点（见练习），$R_0(t)$ 和 $R_1(t)$ 在接近接点时速度都会变为零，所以即使它们的速度在接点处突然改变方向也仍然是连续变化的，所以曲线是一

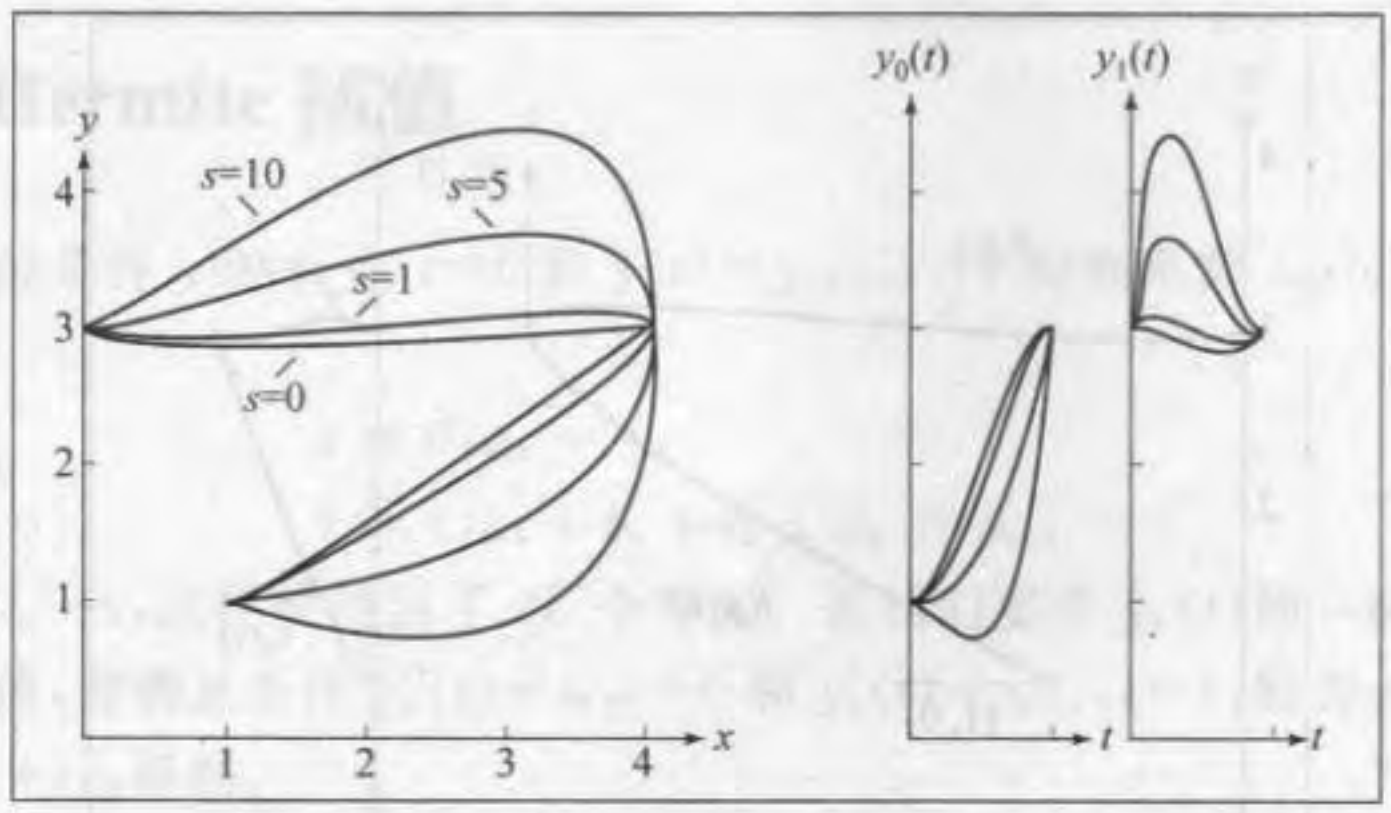


图 10.43 不同斜率条件下的曲线

阶连续的(问它是否 G^1 连续?)。

练习

练习 10.10.1 速度为零的情况

证明若三次曲线的两段在端点处速度均为零,则曲线就变成了直线。

练习 10.10.2 观察曲线

试写一个小程序来产生图 10.41 和图 10.42 中的曲线,然后改变 S 的值看曲线有什么变化。可以如下这样来定义函数:

```
Point2 y(float t, float S, int seg)
{
    //return y(t)
}
```

对 $x(.)$同样。

练习 10.10.3 环

若令 $S=-1$,则图 10.42 中的曲线将会出现一个环。

10.10.3 自然三次样条

上节我们已经知道,只要给定曲线在每个控制顶点处的斜率 s_k,就可以确定曲线的形状,称为 Hermite 插值。但是如果我们指定特殊的 s_k,就可以使得 $y(t)$的二阶导数在每一个内部点处连续(在端点处导数连续没有意义),其中 $k=1,2,\cdots,L-1$。每一段 $y_k(t)$的二阶导数为 $y_k''(t)=6a_kt+2b_k$,所以让 $y_{k-1}''(1)$和 $y_k''(0)$相等,得到:

$$6a_{k-1}+2b_{k-1}=2b_k,\quad k=1,2,\cdots,L-1 \tag{10.65}$$

用等式(10.63)的系数值代入并简化,得到关于斜率的条件:

$$s_{k-1}+4s_k+s_{k+1}=3(y_{k+1}-y_{k-1}),\quad k=1,2,\cdots,L-1 \tag{10.66}$$

可以看出相邻控制顶点的斜率具有上式的关系才能保证曲线在内部点处是二阶连续

的，但我们需要指定两个端点处的斜率 s_0, s_L。一个经典的方法是这样确定它们的，就是让两个端点处的二阶导数都为零，即 $y_0''(0)=0, y_{L-1}''(1)=0$，分别得到：$b_0=0, 3a_{L-1}+b_{L-1}=0$（为什么？）。也用等式(10.63)的系数值代入，得到最后两个条件：

$$\begin{aligned} 2s_0 + s_1 &= 3(y_1 - y_0) \\ 2s_L + s_{L-1} &= 3(y_L - y_{L-1}) \end{aligned} \tag{10.67}$$

将等式(10.66)和等式(10.67)联立得到一个关于斜率 s_k 的包含 L 个方程的线性方程组，后面练习将介绍这个方程组的特殊性质及它的求解。

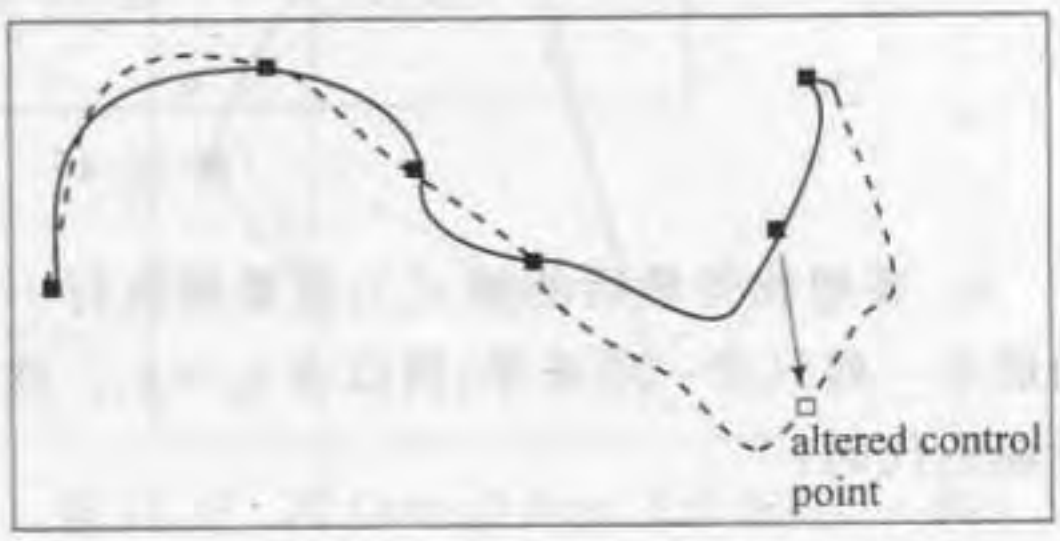


图 10.44　插值控制顶点的自然三次样条

图 10.44 给出了一系列控制顶点和一条插值它们的自然三次样条曲线。注意到曲线处处连续，而且在端点处变为直线，所以端点处二阶导数为零。虚线是移动其中一个控制顶点后得到的曲线，可以看出曲线的位置处处都移动了，所以自然三次样条曲线没有局部控制性，是因为它不仅固定了端点处的一阶导数，还要使得二阶导数连续，所以一个控制顶点的变化会引起整条曲线的变动。

练习

练习 10.10.4　求解自然三次样条中关于斜率的线性方程组

将由等式(10.66)和等式(10.67)组成的线性方程组写成矩阵的形式，比如当 $L=4$ 时，我们有：

$$(s_0, s_1, s_2, s_3, s_4)\begin{pmatrix} 2 & 1 & 0 & 0 & 0 \\ 1 & 4 & 1 & 0 & 0 \\ 0 & 1 & 4 & 1 & 0 \\ 0 & 0 & 1 & 4 & 1 \\ 0 & 0 & 0 & 1 & 2 \end{pmatrix} = 3(y_1 - y_0, y_2 - y_0, y_3 - y_1, y_4 - y_2, y_4 - y_3)$$

系数矩阵是三对角矩阵，即只有对角线及对角线两侧的值非零，这样的线性方程组可以分两步求解，方法如下：

a. 首先对这个方程组执行向前消去法，以消去上面一斜线上的 1。具体步骤为：首先第一个方程两边同除以 2，使其第一项系数为 1；其次从第二个方程开始，依次减去上一个方程的适当倍数，以消去其中的 1；然后缩放这个方程式使其对角线上的系数为 1。这样就把这个方程组转化成了“下三角”形式（即主对角线上面都为零）：

$$(s_0, s_1, s_2, s_3, s_4)\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ v_0 & 1 & 0 & 0 & 0 \\ 0 & v_1 & 1 & 0 & 0 \\ 0 & 0 & v_2 & 1 & 0 \\ 0 & 0 & 0 & v_3 & 1 \end{pmatrix} = (q_0, q_1, q_2, q_3, q_4) \tag{11.68}$$

其中的 $v_i, q_i, i=0,1,2,3,4$ 容易计算出来的。

b. 图 10.45 给出了前面向前消去法的执行代码，其中使用数组 $v[]$ 和 $q[]$。

```
v[0]=0.25;
q[0]=3.0*(y[1]-y[0])*v[0];
for(i=1; i<L; i++)
{
    v[i]=1.0/(4.0-v[i-1]);
    q[i]=(3.0*(y[i]-y[i-1])-q[i-1])*v[i];
}
```

图 10.45 向前消去法

c. 要想获得最后的解 $s[]$,需要再执行一个反向代入步骤,注意到等式(11.79)中矩阵的最后一列只有一项非零,所以有 $s_4=g_4$。然后依次向前代入每一个方程即可得到各个 s_i,下面是代码:

```
s[L]=q[L];
for(i=L-1; i>0; i--) s[i]=q[i]-v[i]*s[i+1];
```

10.10.4 三次样条插值中斜率的计算

对于三次样条插值中斜率的确定,除了上一节介绍的方法外,还有很多其他方法,这里我们只大致介绍目前最常见的一种,**Catmull-Rom** 方法。

我们仍然要求曲线在内部点一阶光滑,但是不再要求二阶光滑,主要是希望将条件放宽可以使得曲线设计者能够更好的局部控制曲线的形状。

上节的方法是通过曲线的二阶导数连续来确定它在控制顶点处的斜率,相反这里我们通过它的相邻控制顶点的位置来确定。

Catmull-Rom 方法是很简单的,就是让点 P_k 的速度向量 $P'(t_k)$ 与它的前一个及后一个顶点的位置相关,如图 10.46 所示,简单的说,就是与顶点 P_{k-1} 到 P_{k+1} 的向量成比例,这样使得曲线在顶点 P_k 的切线平行于它的前一个顶点到后一个的方向,即:

$$\mathbf{P}'(t_k)=m(P_{k+1}-P_{k-1}) \tag{10.69}$$

其中 m 是一常数(我们下面会来分析系数 m 的作用,通常情况下取 $m=1/2$)。按照以前的方式,只需要让函数 $y(t)$ 的斜率 $s_k=m(y_{k+1}-y_{k-1})$,对函数 $x(t)$ 也同样。

这样我们就确定了曲线在每一个内部控制顶点的斜率 s_k,进而确定了等式(10.32)中当 $k=1,2,\cdots,L-2$ 时三次多项式的系数,但是我们还需要两个条件来确定曲线在两个端点处的斜率即 s_0,s_L,可以通过使得端点处的二阶导数为零来得到等式(10.35),其他方法见练习。图 10.47 给出了通过此方法确定的一条曲线,注意到曲线在端点处是伸直的正是因为二阶导数为零。

增加张力控制

为了设计者可以更好的控制曲线在每一个接点处的形状,Catmull-Rom 方法还引入了张力参数 v_k,这可以使设计者改变等式(10.69)中的参数 m,进而在不改变 $\mathbf{P}'(t_k)$ 方向的前提下调整它的大小。

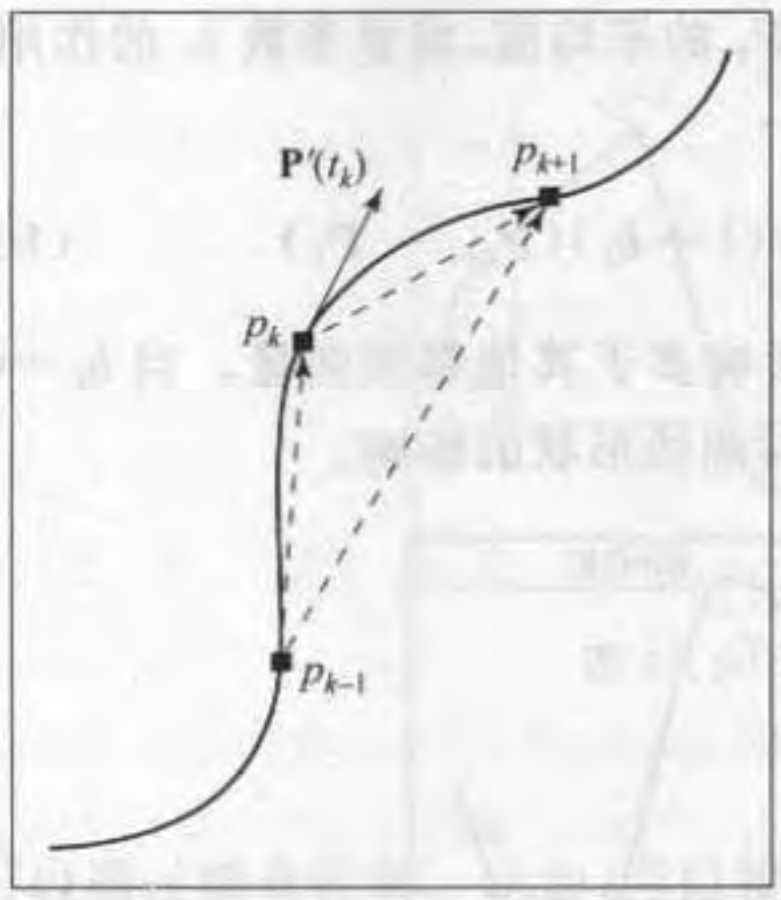


图 10.46　通过两个相邻顶点确定斜率

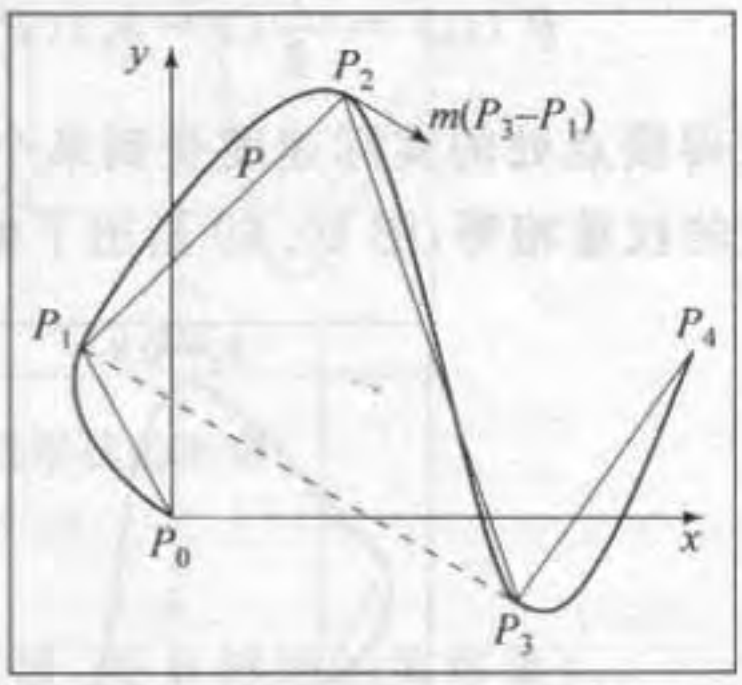


图 10.47　用 Catmull-Rom 方法得到的一条插值曲线($m=1/2$)

这样第 k 个控制顶点的速度向量为：

$$\mathbf{P}'(t_k)=\frac{1}{2}(1-v_k)(P_{k+1}-P_{k-1}) \tag{10.70}$$

其中 $k=1,2,\cdots,L-1$。(这样斜率 s_k 的值会有什么变化?)

参数 v_k 通常被控制在区间[$-1,1$]上，如当 $v_k=0$ 时，对应着等式(10.69)中 $m=\frac{1}{2}$。图 10.48 给出了顶点 P_2 处的张力参数对曲线形状的影响，图 10.48(a)中曲线的张力参数是 1，所以在接点处速度为零，而且在接近接点处曲线变直(为什么?)。图 10.48(b)中曲线的张力参数是 -1，所以曲线在接点处"松弛"。

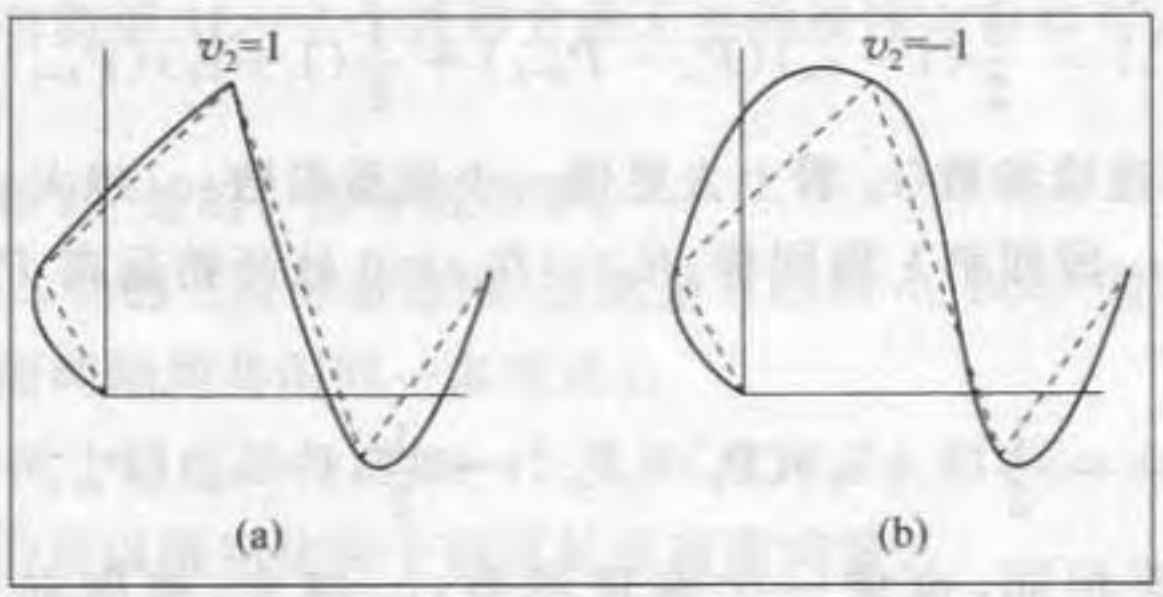


图 10.48　张力参数对曲线形状的影响

(a) 张力参数=1；(b) 张力参数=-1

增加偏重控制

经典曲线 Kochanek-Bartels 样条就是额外增加了一个参数以更好的控制曲线的形状【Kochanek84】，其中之一就是"偏重"。我们可以将等式(10.69)改写一下$\left(m=\frac{1}{2}\right)$：

$$\mathbf{P}'(t_k)=\frac{1}{2}(P_k-P_{k-1})+\frac{1}{2}(P_{k+1}-P_k) \tag{10.71}$$

即表示成与它相邻的两个向量 P_k-P_{k-1}，$P_{k+1}-P_k$ 的平均值，偏重参数 b_k 的作用就是使得这两项不再平均，即：

$$\mathbf{P}'(t_k)=\frac{1}{2}(1-b_k)(P_k-P_{k-1})+\frac{1}{2}(1+b_k)(P_{k+1}-P_k) \tag{10.72}$$

这使得接点处的实际速度受到某个邻接向量的影响多于其他邻接向量。当 $b_k=0$ 时，两个向量的权重相等，图 10.49 给出了偏重参数变化对曲线形状的影响。

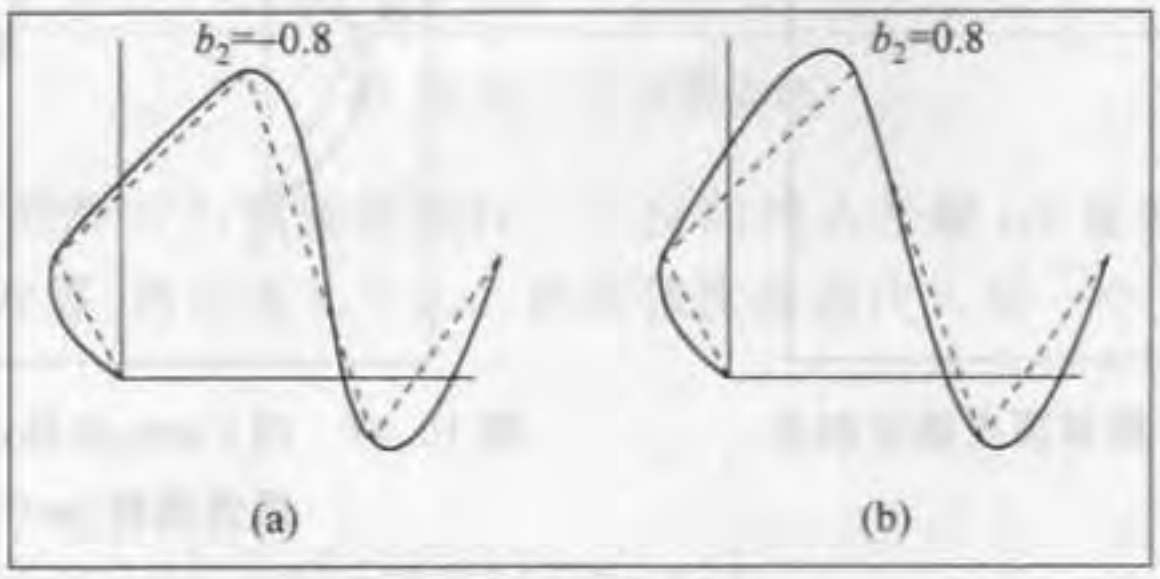


图 10.49 偏重参数对曲线形状的影响

(a) 偏重参数=-0.8；(b) 偏重参数=0.8

增加连续性控制

Kochnaek 和 Bartels 引入另一参数，设计者通过这个参数，可以控制接点处之前的速度不同于接点之后的，而不同于以前让速度向量在接点处连续。以接点 P_k 来说，它的前一段即第 $k-1$ 段，$R_{k-1}(t)$(见等式 10.59)在 $t=1$ 处到达 P_k，我们在此处设置它的速度向量为：

$$\mathbf{R}'_{k-1}(1)=\frac{1}{2}(1-c_k)(P_k-P_{k-1})+\frac{1}{2}(1+c_k)(P_{k+1}-P_k) \tag{10.73}$$

这里的 c_k 就称为连续参数(c_k 看上去更像一个偏重参数：c_k 越大，速度向量越接近向量 $P_{k+1}-P_k$)。对 P_k 后一段即第 k 段同样，$R_k(t)$ 在 $t=0$ 处开始远离 P_k，我们在此处设置它的速度向量为：

$$\mathbf{R}'_k(0)=\frac{1}{2}(1+c_k)(P_k-P_{k-1})+\frac{1}{2}(1-c_k)(P_{k+1}-P_k) \tag{10.74}$$

这里的 c_k 与前面相同，也像一个偏重参数：c_k 越大，速度向量越接近向量 P_k-P_{k-1}。如果 $c_k=0$，这两个速度向量相同，曲线在接点处一阶光滑。当 c_k 不等于零时，这两个向量具有不同的大小和方向。图 10.50 给出了这个连续参数对曲线形状的影响。

张力、偏重、连续性这三个参数可以同时被应用到实际中(见练习)，设计者可以在接点处分别设置这三个参数的值。典型应用中设计者遵守以下步骤：

1. 用鼠标点击确定初始控制顶点。

2. 检查这样得到的曲线，符合条件就停止。

3. 不符合条件的话，编辑控制顶点，调整每个控制顶点的张力，偏移量和连续性参数(设计者可以点击一个控制顶点，将它拖动到新的位置。然后在这个顶点调整它的各个参

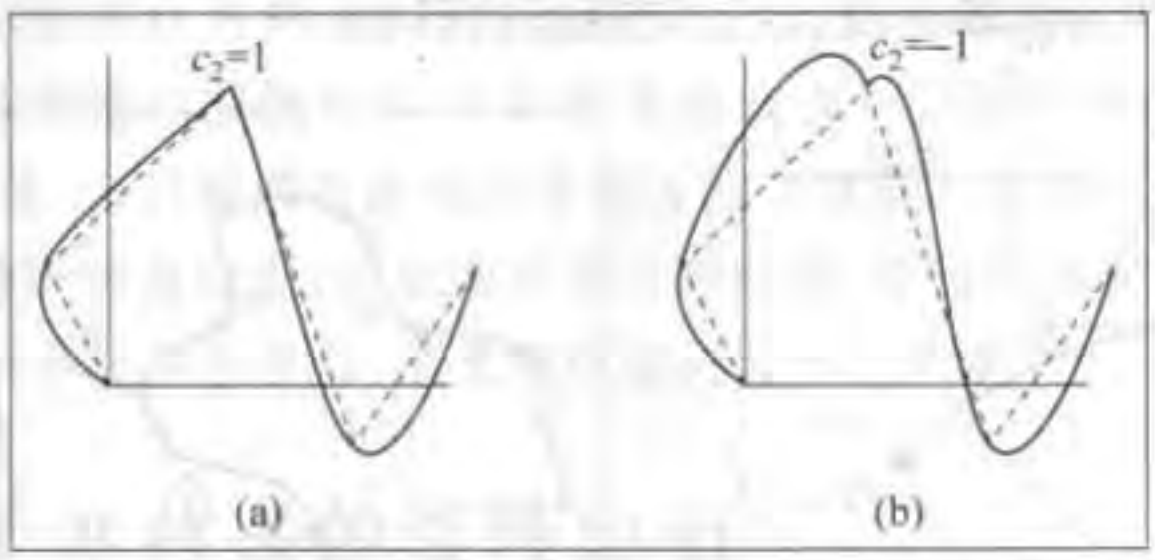


图 10.50　连续参数对曲线形状的影响

(a) $c_2=1$；(b) $c_2=-1$

数，可以通过键盘控制。比如，可以按 b 键减少偏移量，按 B 键增加偏移量）。

4. 返回到第二步。

练习

练习 10.10.5　综合的 Kochnaek-Bartels 样条

综合考虑张力、偏移量和连续性三个参数的影响，来写出 Kochnaek-Bartels 样条 $\mathbf{R}'_{k-1}(1)$ 和 $\mathbf{R}'_k(0)$ 的等式。

练习 10.10.6　选择端点条件

前面在 $L+1$ 个控制顶点上三次插值的时候，我们需要确定 $4L$ 个系数可，其中 $4L-2$ 个可以通过插值条件和斜率约束来确定，有 2 个系数待定。等式(10.66)是通过要求两个端点的二阶导数为零来确定这 2 个系数的，但是还有很多其他条件可以用来确定它们，只需要有两个关于系数的且与前面 $4L-2$ 个线形方程无关的线性方程即可，以下几个条件均满足此要求：

a. 两个端点 P_0 和 P_L 处的一阶导数为零。

b. 顶点 P_1 和 P_{L-1} 处的三阶导数连续（这就是有名的 de Boor 的“非节点”约束，它使得曲线的前面两段和后面两端均共用同一多项式）。

c. 在控制多边形两个端点处各增加一个顶点（记为 P_{-1} 和 P_{L+1}），曲线仍然从 P_0 开始，P_L 结束，但是这样我们可以得到这两个顶点处的速度向量。

练习 10.10.7　通过控制顶点的抛物线

为了计算顶点 P_k 处的斜率，我们可以先求出经过点 P_{k-1}，P_k，P_{k+1} 的抛物线，然后据此计算出点 P_k 处的速度，对于任意这样的三个顶点，都可以求出速度的表达式。

10.10.5　交互指定切向量

一个 CAD 绘图程序可能需要给设计者提供可以控制切向量的可视“控制柄”，如图 10.52 所示。用户指定控制顶点后，就会有一条初始插值曲线，这时用户可以通过拖动鼠标来限定某个顶点处的切向量方向，然后就会产生一条新曲线取代原来的曲线。这样的可

视化交互让用户可以快速编辑曲线,以达到理想的形状。

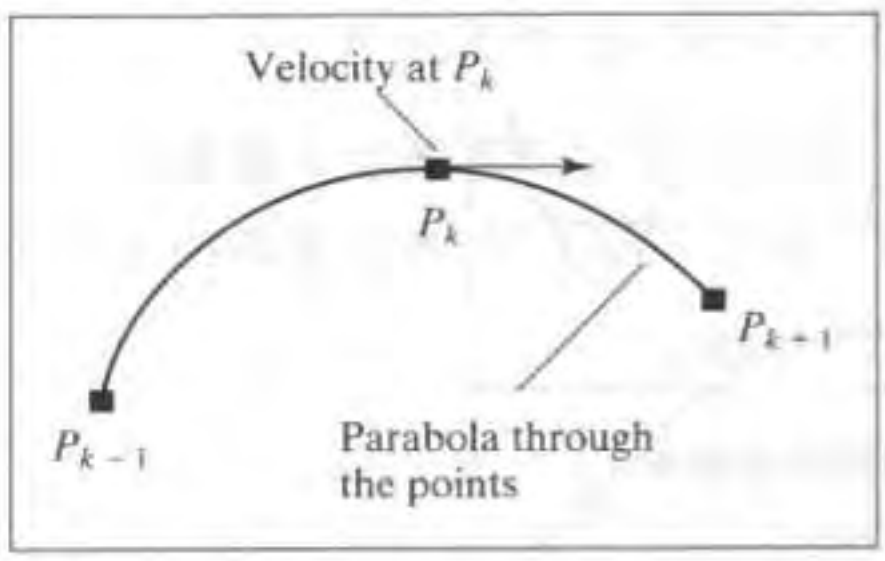


图 10.51 由经过三个相邻顶点的抛物线来确定中间顶点处的速度

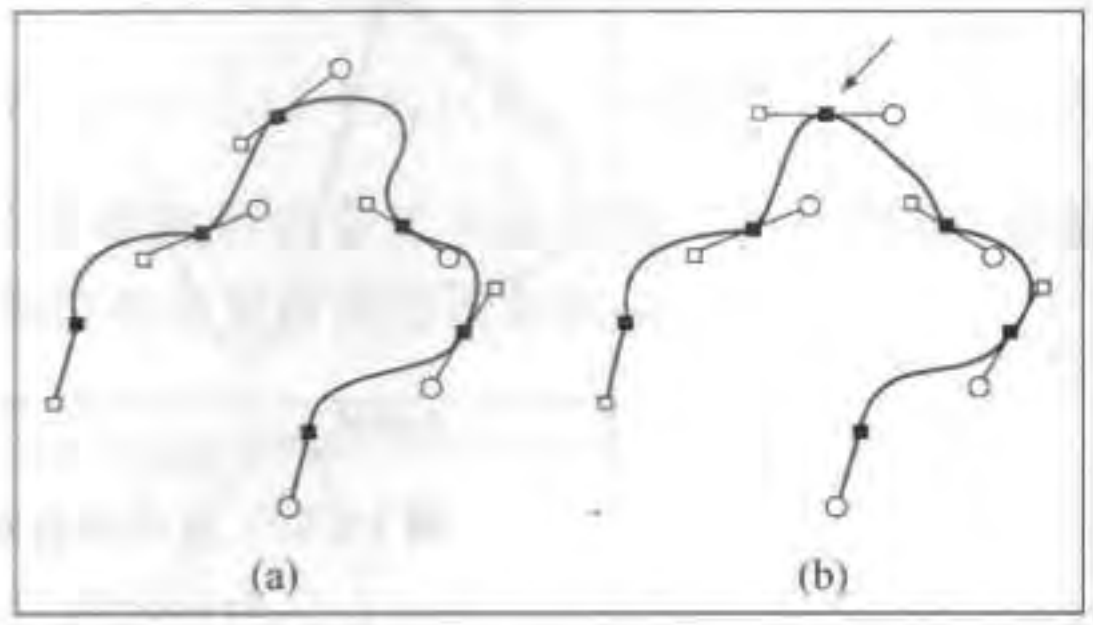


图 10.52 三次曲线段的交互设计

10.11 自由曲面造型

迄今为止,我们已经研究了如何表示和构造二维及三维上的曲线,那么显然对这些思想的推广可以得到曲面的构造方法。在第 6 章我们介绍了一系列不同类型的曲面,如直纹面、双线性曲面片、Coons 曲面片以及旋转曲面等。现在我们来研究如何用贝塞尔曲线和B样条曲线来设计生成这些曲面,从而为这些曲面的构造提供强有力的工具。

10.11.1 基于 B 样条的直纹面

直纹面是一类非常便于使用的曲面。回忆一下 6.5.6 节中的内容,直纹面由两条末端曲线 $P_0(u)$和 $P_1(u)$决定,且每一个 u 所确定的两点用直线相连后就生成了相应直纹面。因此直纹面的参数表达式就是两条曲线上相应点的线性插值(或内插)。为方便起见,重复等式(6.35),我们有

$$P(u,v) = (1-v)P_0(u) + vP_1(u) \tag{10.75}$$

这里我们把 $P_0(u)$和 $P_1(u)$扩展为 B 样条(或贝塞尔)曲线。图 10.53 显示了末端曲线 $P_0(u)$和$P_1(u)$为三次贝塞尔曲线的直纹面。$P_0(u)$的 4 个控制顶点为 $P_0^0, P_1^0, P_2^0, P_3^0$,$P_1(u)$的 4 个控制顶点为 $P_0^1, P_1^1, P_2^1, P_3^1$。根据贝塞尔曲线的等式(10.25),可以把这个曲面写作:

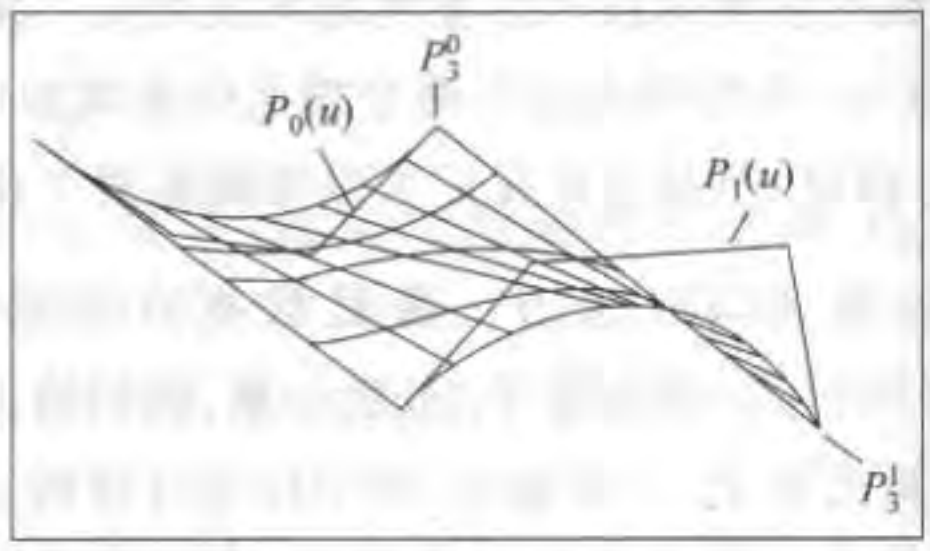


图 10.53 基于贝塞尔曲线的直纹面

$$P(u,v) = \sum_{k=0}^{3}((1-v)P_k^0 + vP_k^1)B_k^3(u) \tag{10.76}$$

现在来看看这个曲面的特性,u 线是连接两条对边贝塞尔曲线上相应两点的直线,v 线

是贝塞尔曲线，它的控制顶点为两条对边贝塞尔曲线上相应控制顶点的内插$(1-v)P_k^0+vP_k^1$。上述的两条末端贝塞尔曲线可以用 B 样条曲线或 NURBS 曲线代替，甚至它们可以定义在不同的节点向量上，只要两条曲线的参数 u 定义在同一个区间即可。

这样，所有比较特殊的直纹面，比如圆锥面和圆柱面，都可以被简单的表达。例如，如果 $P_0(u)$是 $P_1(u)$在空间中的一个平移，那么直纹面会成什么形状？

10.11.2　基于 B 样条的旋转曲面

回忆 6.5.7 节中的内容，轮廓线绕着轴线旋转扫描形成的扫掠面就是旋转曲面。若 $C(v)=(X(v),Z(v))$绕 z 轴扫掠，生成的旋转曲面其参数表达式为等式(10.77)。我们假设轮廓线 $P(u)=(x(u),y(u))$在 x 轴和 y 轴上有分量，那么在它绕着 y 轴扫掠后生成的旋转曲面其参数表达式为：$P(u,v)=(x(u,v),y(u,v),z(u,v))=(x(u)\cos(v),y(u),x(u)\sin(v))$。

$$P(u,v)=(X(v)\cos(u),X(v)\sin(u),Z(v)) \tag{10.77}$$

为方便起见，轮廓线通常用贝塞尔曲线或 B 样条曲线表示。我们选择 $L+1$ 个控制顶点(X_k,Z_k)来构造曲线：

$$(X(v),Z(v))=\sum_{k=0}^{L}(X_k,Z_k)N_{k,m}(v) \tag{10.78}$$

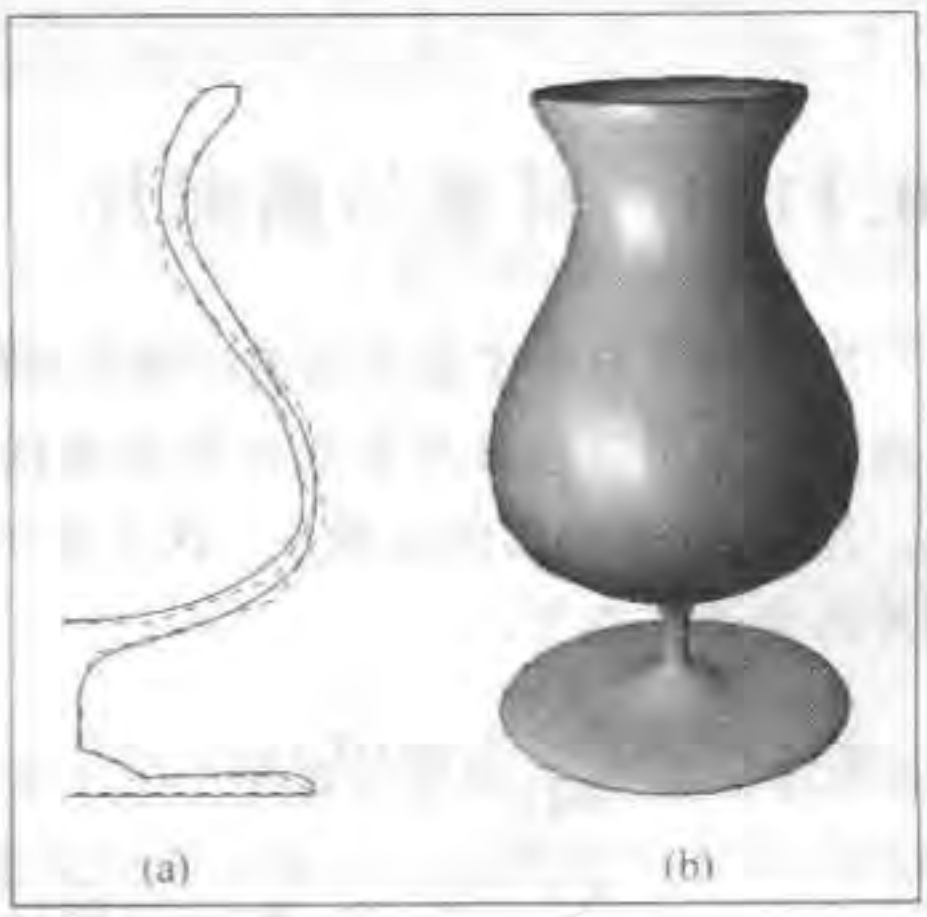


图 10.54　轮廓线为 B 样条曲线

(a) 高脚玻璃杯的轮廓线；

(b) 旋转曲面生成的高脚玻璃杯

图 10.54(a)中用三次 B 样条曲线定义了一个高脚玻璃杯的轮廓线，图 10.54(b)显示了由此生成的旋转曲面(在第 6 章中介绍了旋转曲面的网格对象，则这个网格的绘制可以应用 Mesh::draw()方法)。

例 10.11.1　经典的茶壶

轮廓线也可以用贝塞尔曲线表示。图 10.55(a)显示了最初由 Martin Newell【Blinn96】设计的茶壶壶体的轮廓线。此轮廓线由 3 条贝塞尔曲线组成，其 10 个控制顶点如图 10.56 所示。第一段贝塞尔曲线的控制顶点为点 0、1、2、3，第二段为点 3、4、5、6，第三段为点 6、7、8、9。注意到每条曲线最后一部分与下一段曲线第一部分共线(译者注：应为“每条曲线段控制顶点的最后一部分与下一段曲线控制顶点的第一部分共线”)，这个条件保证了不同的贝塞尔曲线合成一条曲线时满足导矢向量(译者注：应为“向量方向”)连续(见 10.1.2 节)。

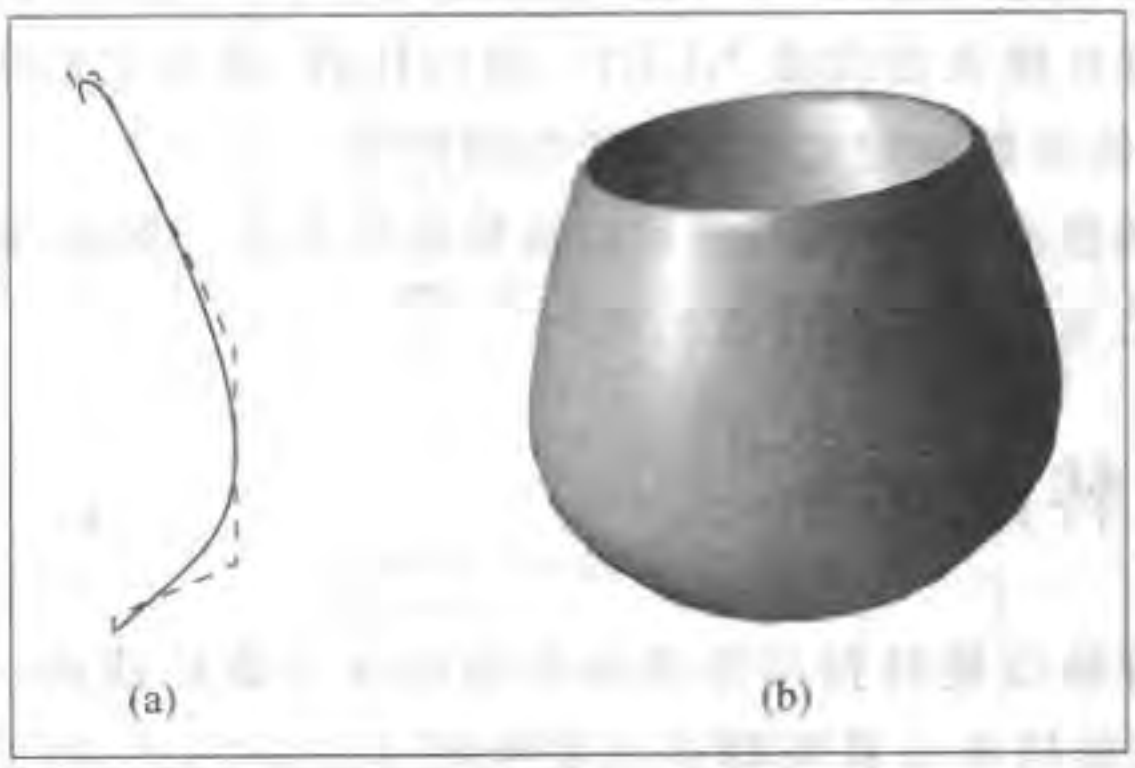


图 10.55 茶壶壶体轮廓线的贝塞尔曲线表示

(a) profile of body; (b) the teapot

i	x	z
0	1.4	2.25
1	1.3375	2.38125
2	1.4375	2.38125
3	1.5	2.25
4	1.75	1.725
5	2	1.2
6	2	0.75
7	2	0.3
8	1.5	0.075
9	1.5	0

图 10.56 茶壶壶体轮廓线数据点

10.11.3 贝塞尔曲面片

图 10.57 显示了基于两条贝塞尔曲线的直纹面。为了设计时更加灵活,等式(10.76)中线性的 u 线可以用贝塞尔或 B 样条曲线代替。贝塞尔曲面片的 u 线和 v 线都是贝塞尔曲线。比如,如果曲面的 u 线是二次贝塞尔曲线,而 v 线是三次贝塞尔曲线,那么此贝塞尔曲面片的表达式如下:

$$P(u,v)=\sum_{k=0}^{3}P_k(v)B_k^3(u)$$
$$=\sum_{k=0}^{3}\left(\sum_{i=0}^{2}P_{i,k}B_i^3(v)\right)B_k^3(u) \quad (10.79)$$

这里,u、v 值在 0 到 1 之间变化。图 10.57 给出了一个例子。

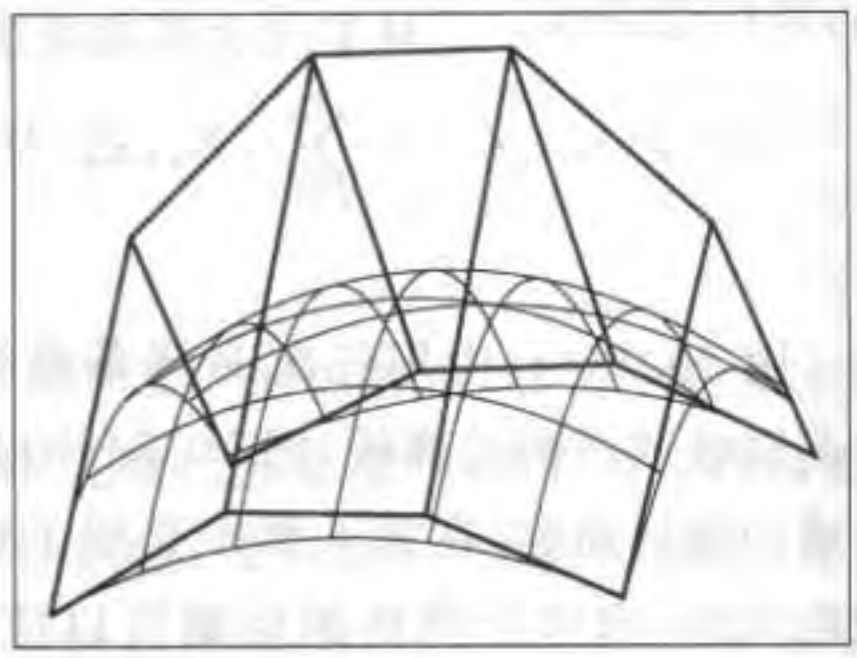


图 10.57 贝塞尔曲面片的一个实例

每一条 v 线都是关于 u 的贝塞尔曲线,它的 4 个控制顶点都位于一条二次贝塞尔曲线上。(这些 u 线该如何描述?)这 12 个控制顶点一起构成了**控制多面体**,它们决定了一个贝塞尔曲面片的形状。贝塞尔曲面片的表达式如等式(10.80):

$$P(u,v)=\sum_{k=0}^{L}\sum_{i=0}^{M}P_{i,k}B_i^M(u)B_k^L(v) \quad (10.80)$$

通常这个控制多面体一般就是由$(M+1)(L+1)$个顶点组成的控制网格。为了生成一个贝塞尔曲面片,设计者必须仔细设置各个顶点的位置来决定曲面的形状。

图 10.58 显示了一个**双三次**贝塞尔曲面片(即 L 和 M 都是 3)及其控制多面体。关于贝塞尔曲面的更多细节,可参看【Rogers90】。ATI(www.ati.com)和 NVIDIA(www.nvidia.com)也都提供关于贝塞尔曲面建模的既有意思又有深度的讨论。

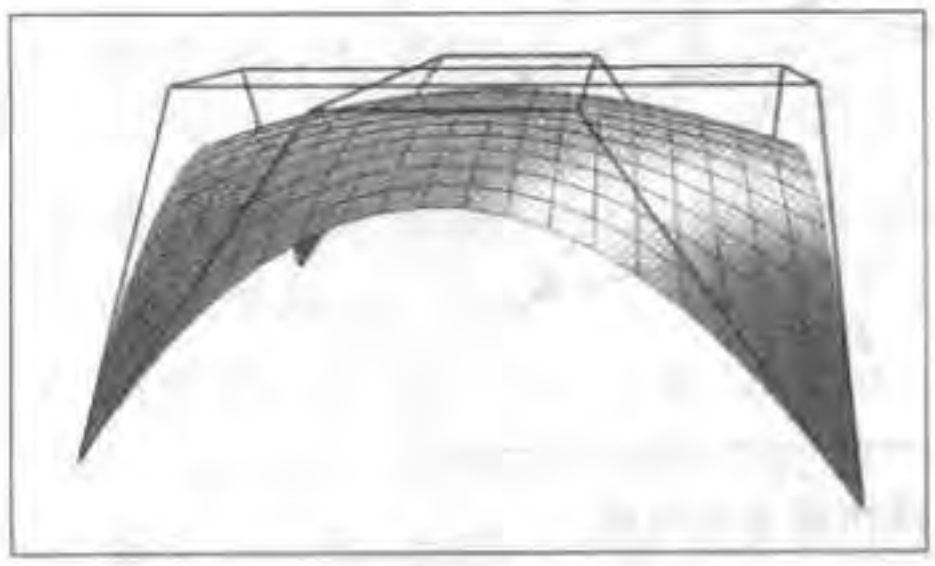


图 10.58　双三次贝塞尔曲面片及其控制多面体

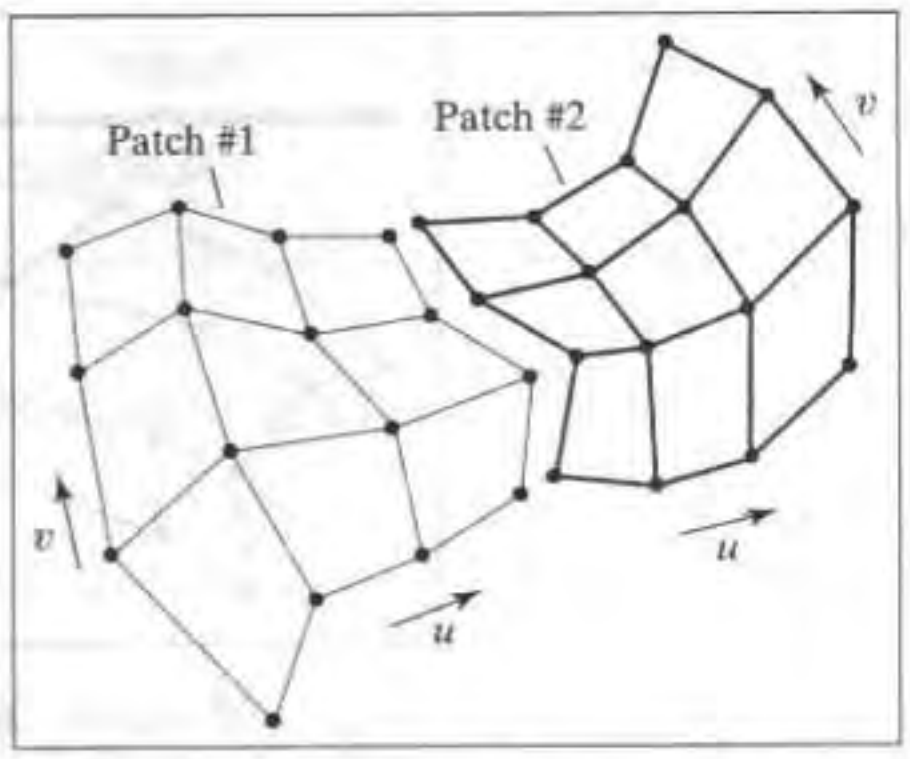


图 10.59　两个连续过渡的贝塞尔曲面片

练习

练习 10.11.1　贝塞尔曲面片的法线

把贝塞尔曲面片的参数表达式代入等式(6.25)，得到一个复杂的关于贝塞尔曲面法向量的表达式。请从形式上尽可能的进行化简。问法向量是否在整个(u,v)上连续变化？

10.11.4　贝塞尔曲面片的拼接

设计人员在对复杂外形进行造型时，也许需要用到多个贝塞尔曲面片，并且它们在公共边界上能够光滑拼接。图 10.59 显示了两个用于定义贝塞尔曲面片的控制多面体，分别为黑色和灰色。对两个控制多面体都应用等式(10.80)，当 u 和 v 都从 0 变化到 1 时就生成了相应的贝塞尔曲面片。这时它们只有控制网格不同而已。

为了两个曲面片能够无缝拼接，两个相应控制多面体必须满足怎样的条件？事实上很简单，只要两个曲面片的边界重合，即只需两个控制网格在边界处的点重合。这是由于边界上的曲线形状只由控制多面体上的相应边界点决定(可考虑等式(10.80)在 $u=0$ 时的情况)。所以设计者只需取两个曲面片的边界有相同的控制多边形即可。

相比较而言，在连接处保持切线(译者注：应为“切线方向”)连续更为困难(但是一旦切线方向连续，那么也就保证了曲面法线方向在连接处连续——这是为什么？)一个充分条件如图 10.60 所示：两个控制多面体中两条相交且交点在边界上的边，比如 E 和 E1，必须共线。但设计者很难做到这一点，更多难度稍低的要求详见 Faux 和 Pratt【Faux79】。

例 10.11.2　茶壶把手的设计

我们已知茶壶的壶身和壶盖是基于贝塞尔曲线的旋转曲面，而茶壶的把手和壶嘴，每一个却都是由四个贝塞尔曲面片构成的。这里我们来示范把手是如何构造的【Blinn96】，壶嘴的构造详见案例 10.8。

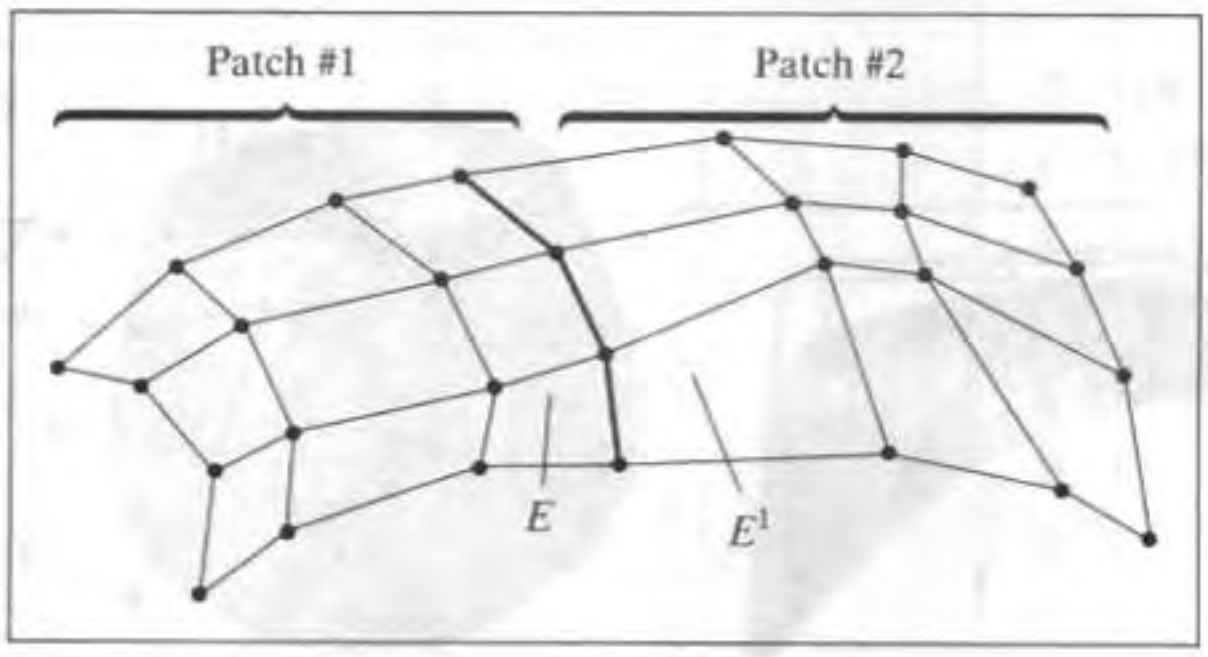


图 10.60 在连接处保持切线方向连续

图 10.61(a)从三维的视角显示了茶壶的把手,图 10.61(b)显示了它的横截面,以及横截面处相应的控制多面体。

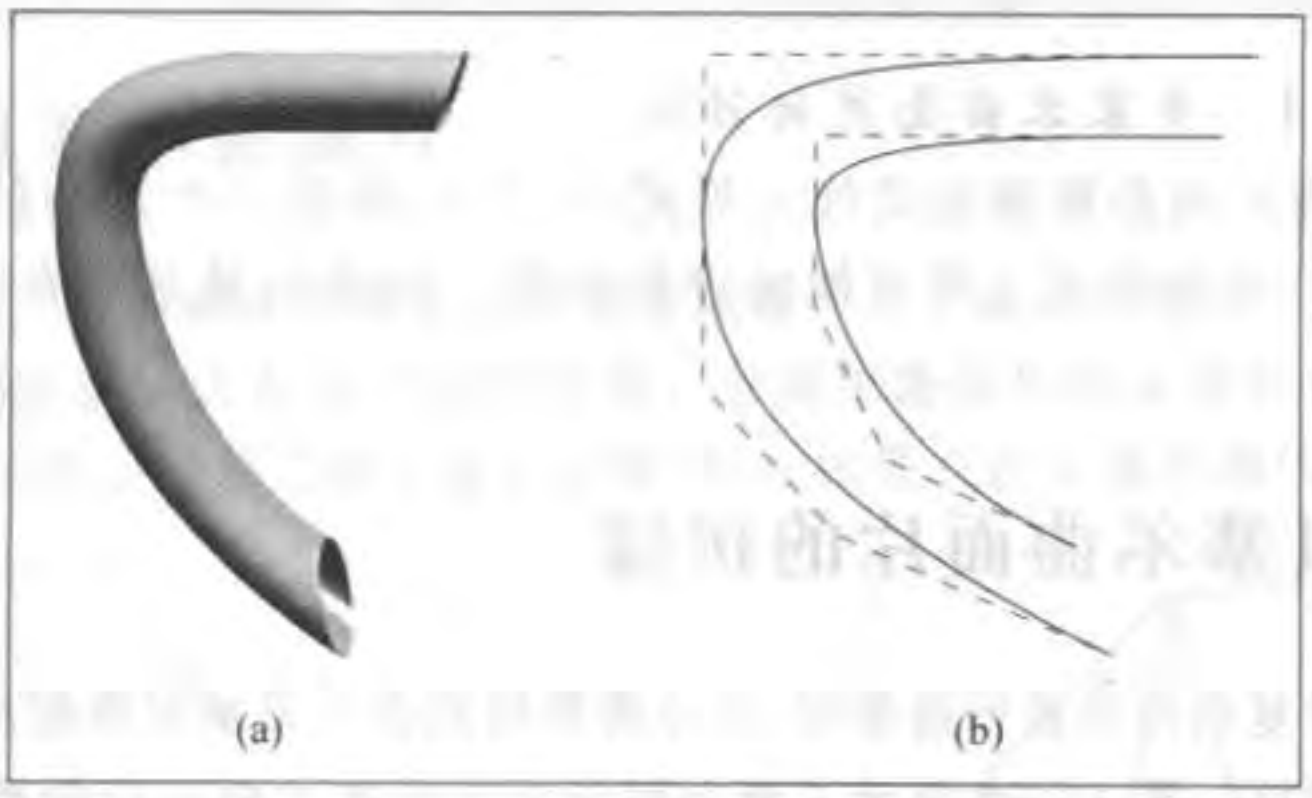


图 10.61 茶壶把手的设计

把手曲面是关于 xz 平面对称的,在 xz 平面的 y 轴正方向一侧有上下两个曲面片,同时类似镜面反射,xz 平面的 y 轴负方向一侧也有上下两个曲面片。对于 y 轴正方向一侧的上曲面片的控制多面体,是由 4 个竖立在 xz 平面上的矩形组成,这些矩形延伸到正八分圆 0.3 个单位距离处。对于第一个矩形,在 $y=0$ 和 $y=0.3$ 时使用 A_0 点各一次,同样,$y=0.3$ 和 $y=0$ 时使用 B_0 点各一次。当 $A_0=(-1.65, 1.875)$ 和 $B_0=(-1.5, 2.1)$ 时,我们得到四个控制顶点(−1.65,0.0,1.875),(−1.65,0.3,1.875),(−1.5,0.3,2.1),(−1.5,0.0,2.1)。

整个在 y 轴正方向一侧的上曲面有 16 个控制顶点:

- 基于 A_0,B_0 点的:(−1.6,0.0,1.875),(−1.6,0.3,1.875),(−1.5,0.3,2.1),(−1.5,0.0,2.1)
- 基于 A_1,B_1 点的:(−2.3,0.0,1.875),(−2.3,0.3,1.875),(−2.5,0.3,2.1),(−2.5,0.0,2.1)
- 基于 A_2,B_2 点的:(−2.7,0.0,1.875),(−2.7,0.3,1.875),(−3.0,0.3,2.1),(−3.0,0.0,2.1)

- 基于 A_3，B_3 点的：(−2.7，0.0，1.65)，(−2.7，0.3，1.65)，(−3.0，0.3，1.65)，(−3.0，0.0，1.65)

同样，整个在 y 轴正方向一侧的下曲面有 16 个控制顶点：

- 基于 A_3，B_3 点的：(−2.7，0.0，1.65)，(−2.7，0.3，1.65)，(−3.0，0.3，1.65)，(−3.0，0.0，1.65)
- 基于 A_4，B_4 点的：(−2.7，0.0，1.425)，(−2.7，0.3，1.425)，(−3.0，0.3，1.2)，(−3.0，0.0，1.2)
- 基于 A_5，B_5 点的：(−2.5，0.0，0.975)，(−2.5，0.3，0.975)，(−2.65，0.3，0.7875)，(−2.65，0.0，0.7875)
- 基于 A_6，B_6 点的：(−2.0，0.0，0.75)，(−2.0，0.3，0.75)，(−1.9，0.3，0.45)，(−1.9，0.0，0.45)

案例 10.8 中要求读者使用这些数据点写一个程序，从不同视角表示这个茶壶。

10.11.5　B 样条曲面片

在曲面设计中，为了得到更好的局部可控性，我们可以用 B 样条函数来代替 Bernstein 多项式。其等式为

$$P(u,v)=\sum_{i=0}^{M}\sum_{k=0}^{L}P_{i,k}N_{i,m}(u)N_{k,n}(v) \tag{10.81}$$

这里 $N_{i,m}(u)$ 和 $N_{k,n}(v)$ 是由等式(10.47)定义的 B 样条基函数(可以有不同的阶数)。通常我们为这两个 B 样条形式选择标准节点向量，这样就可以插值控制多面体的角点了。如果控制顶点反复出现或者有如同等式(10.55)那样的周期性，则曲面封闭(关于 u 或 v 封闭或两者皆封闭)。控制多面体由 $(L+1)(M+1)$ 个控制顶点组成，u 和 v 的取值范围从 0 到其相应节点向量中节点最大值。我们还是选择三次 B 样条(即 $m=n=4$)。由于控制顶点的数目不受限制(顶点数并不像贝塞尔曲线那样影响多项式的阶数)，它可以对极其复杂的曲面进行很好的造型。如前文所示，为了设计出需要的曲面，设计者必须设置相应的节点向量和控制多面体。

图 10.62 显示了 B 样条曲面的一个例子。它在某一个参数上具有周期性(见习题)，而且控制多面体和 u 线 v 线清晰可见。

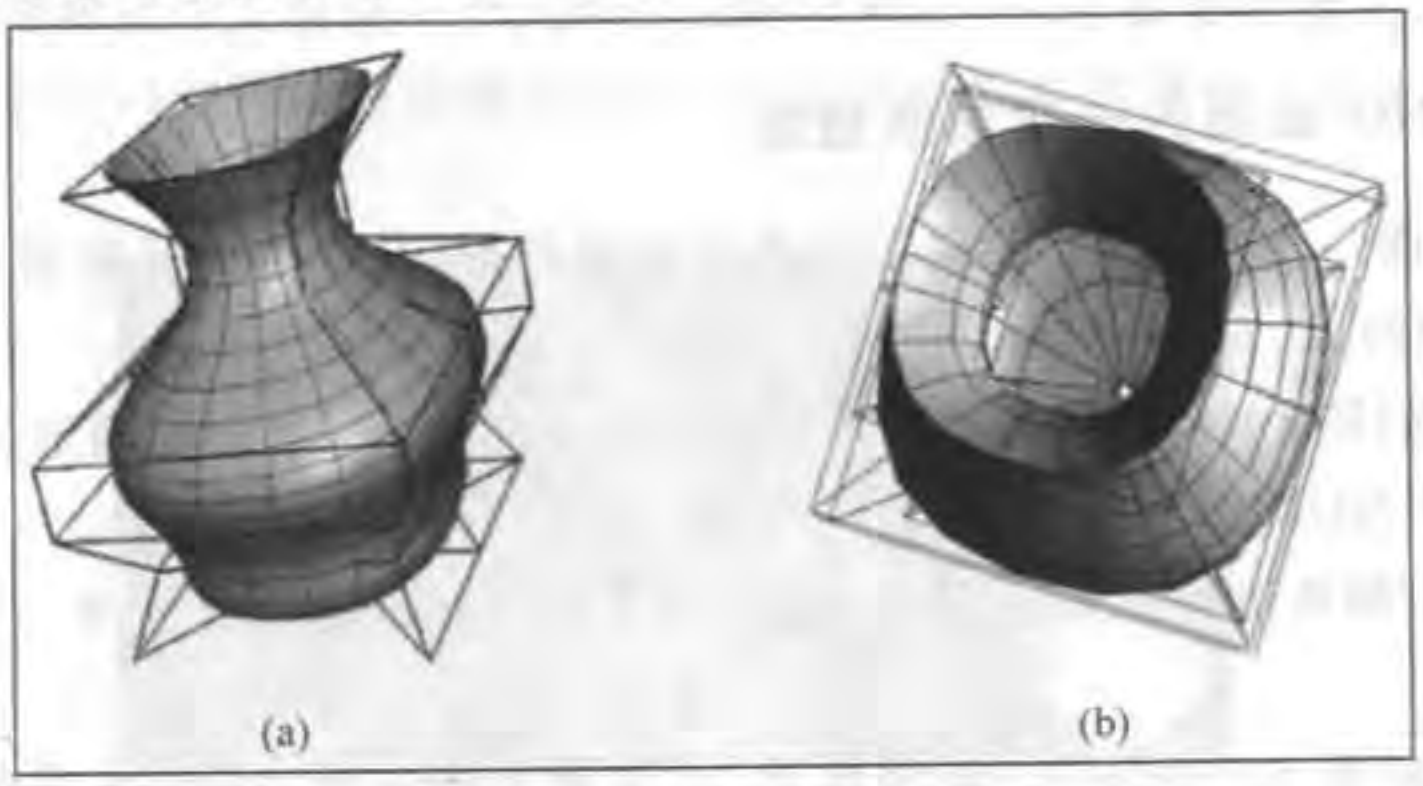


图 10.62　B 样条曲面的实例

练习

练习 10.11.2 周期 B 样条曲面

修改等式(10.81)使得曲面在参数 u 上具有周期性。你会发现等式(10.55)有助于此。那么对于这里的两类 B 样条,怎样的节点向量才有用?

10.11.6 NURBS 曲面

在 10.9 节中,我们简要地介绍了有理参数 B 样条曲线。如同等式(10.57)所示,这类曲线的形状同时取决于一组控制顶点和一组权因子 ω_i。设计者通过设置这些参数来得到所需曲线。

如同等式(10.81)那样处理 B 样条曲线,我们同样可以把 NURBS 曲线扩展成 NURBS 曲面,【Rogers00】使用如下形式:

$$P(u,v)=\frac{\sum_{i=0}^{M}\sum_{k=0}^{L}\omega_{i,k}P_{i,k}N_{i,m}(u)N_{k,n}(v)}{\sum_{i=0}^{M}\sum_{k=0}^{L}\omega_{i,k}N_{i,m}(u)N_{k,n}(v)} \tag{10.82}$$

这里 $P_{i,k}$ 是**控制多面体**的控制顶点,$\omega_{i,k}$ 是权因子,$N_{i,k}$ 是 B 样条函数。如同 B 样条曲线,如果所有的权因子相等,则这个曲面就可以化简为等式(10.81)中的 B 样条曲面(请验证)。

如同前面所讨论的,NURBS 曲面有两个主要优点:

1. 选择适当的控制顶点和权因子,$P(u,v)$ 可以精确表示二次曲面。这与非有理 B 样条曲面形成对比,后者只能逼近二次曲面。后面我们会举例构造圆锥曲面。

2. NURBS 曲面具有**投影不变性**。这个不变性即若要得到一个 NURBS 曲面的投影曲面,只需计算原曲面控制顶点的投影点,适当调整相应权因子,然后应用等式(10.82)即可。与此相比,非有理 B 样条曲面只有仿射不变性,而没有投影不变性。

由于 NURBS 曲面表现出来的一般性和灵活性,它在曲线曲面设计中有着广泛的应用。

基于 NURBS 曲面的一般曲面造型

NURBS 曲面一族在设计中提供了种类繁多的外形,其中包括了前面章节中讨论的曲面类型。实例如图 10.63 所示。

- 拉伸曲面(图 10.63(a))。棱柱外形由关于 u 的 NURBS 曲线确定,直线边界由关于 v 的一阶 NURBS 曲线决定。
- 直纹面。两条边界曲线是关于 u 的 NURBS 曲线,母线是关于 v 的一阶 NURBS 曲线。
- 旋转曲面(图 10.63(b))。轮廓线是 xz 平面上的一条 NURBS 曲线,鉴于 NURBS 曲线可以精确表示圆锥曲线,则用平行于 $z=0$ 的平面截得部分为圆弧。

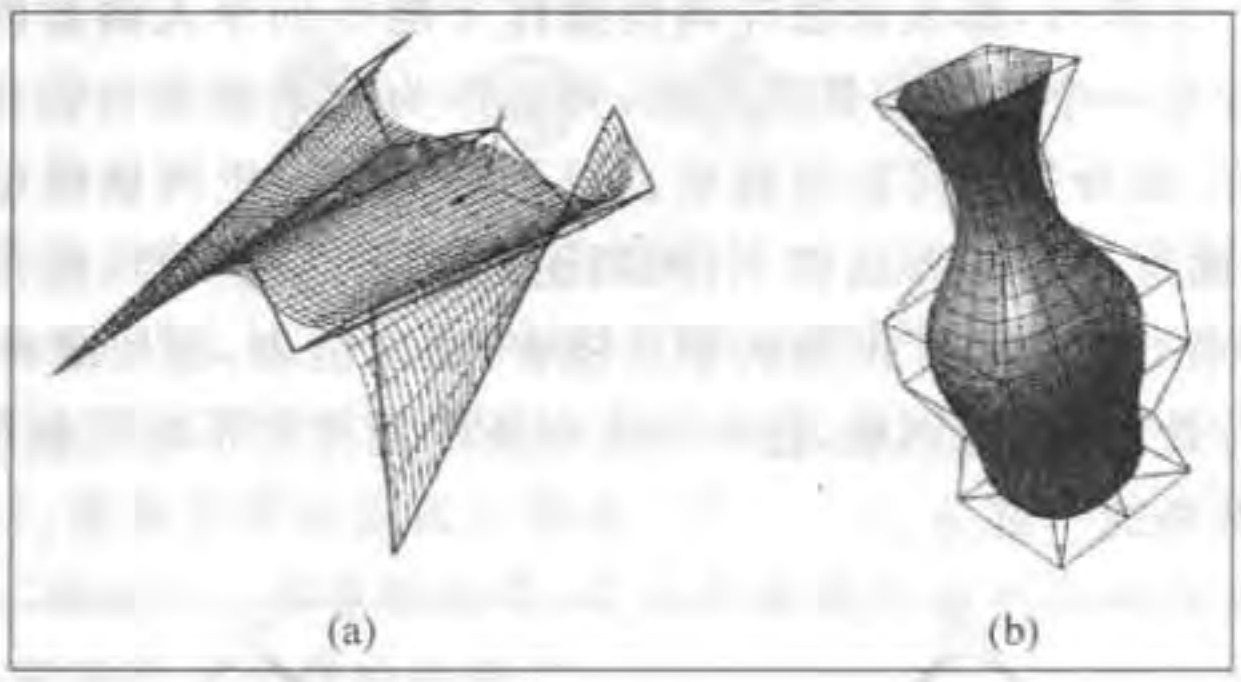


图 10.63　NURBS 曲面的实例

- 二次曲面。旋转曲面若以圆锥曲线为其轮廓线，则生成二次曲面。若对它进行仿射变换，可使其发生扭曲而不再成圆对称。

我们在此强调仅仅依靠 NURBS 设计工具所展现的灵活性，就足以对以上所有类型曲面进行造型。案例 10.10 展示了在设计和造型这些曲面时所遇到的细节。

许多用于设计和处理曲面的其他技术也已经有所发展。关于这些技术具体实例及细节的资源和链接可以在本书所属网站上找到。

练习

练习 10.11.3　一般曲面造型

讨论如何用 NURBS 曲面对以下曲面进行造型：(a)圆柱；(b)圆锥；(c)平面；(d)球面。更多细节参见案例 10.8。是否可以对任意的 Coons 曲面片进行造型？

10.11.7　细分曲面

对于一些网格，尤其是从第三方得到的以及能够在线使用的网格，它们只提供粗糙的几何模型，对一些应用而言缺少细节。比如一些地形模型如果面片数量不足，就不能遮盖个别的表面，因而也不容易区别光滑波动的山丘(从远处望去)和山脉上陡峭的山峰(从飞机上近距离观察)。或者你希望手工制成一个光滑的国际象棋棋子，比如象或者兵，如图 10.64 所示，但是所能使用的锉刀却只能锉成很少的面片(如图中最左边模型)，从而不能令人感到满意。

图 10.64　网格线的两层改进，以及明暗显示模式下的棋子

如果得到这样一个锉刀,那么该怎样调整操作才能得到令人满意的模型?细分网格面片从而得到新的面片是一个极有前景的方法,并且作为一个课题它仍在被研究中。如同我们接下来描述的那样,细分过程可以很简单。对于图 10.65 里网格模型中的一个特定的三角形面片,一个增加面片的简单方法如下:两两连接三条边的中点,使得一个三角形生成四个三角形(谨记第 6 章介绍的关于网格的顶点链表,法向链表,面片链表等数据结构,如何利用它们构造算法进行处理)。特别地,注意到生成新的面片并不需要新的数据——改进后的网格实质上是原始网格的一部分。

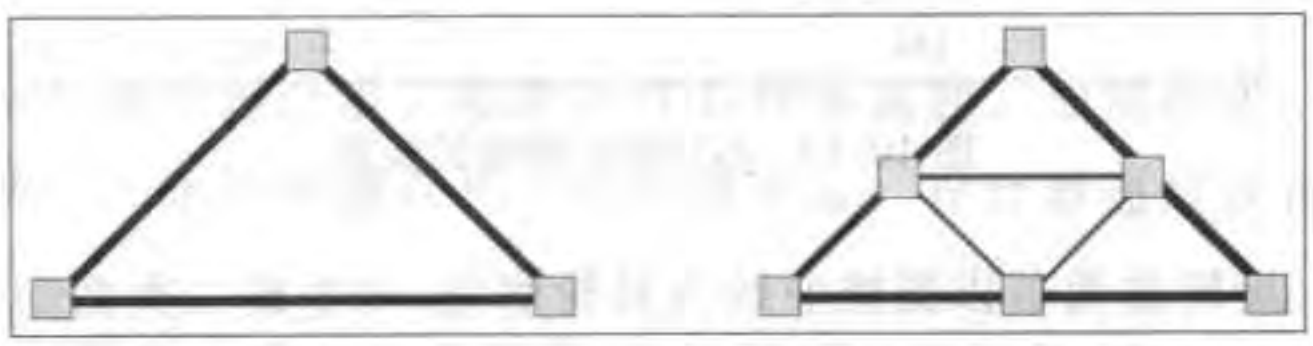


图 10.65 左侧:原始网格;右侧:经过一次细分后的网格

由一个面片生成四个面片,称为"单层改进"。显然把这个过程直接继续下去可得到更多层次的改进。图 10.66 显示了线框显示模式下一个女性躯干模型的三层改进,其中左侧模型的面片清晰可见。

图 10.66 在线框显示模式下的三层改进

图 10.67 中在明暗显示模式下同样的三层改进。此时从左至右各个模型在质量上的差别十分明显。

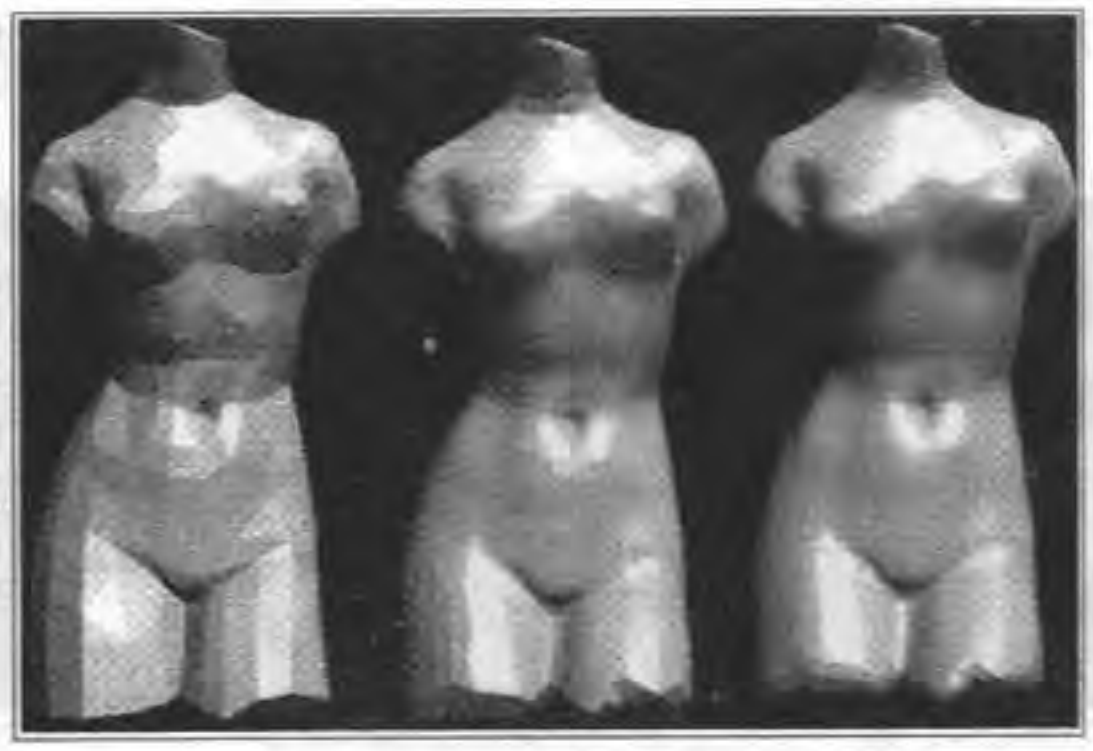


图 10.67 在明暗显示模式下的三层改进

本章小结

本章介绍了几类表示和处理复杂曲线曲面的技术。这类曲线主要有两个用处，一个是用于图形应用程序中的绘制，另一个是对物体的移动进行建模，比如动画中的照相机。大多数的曲线设计是基于在第 3 章首次出现的参数表达形式。在处理和绘制在特定点自交的或是垂直移动的曲线时，参数方程比隐式方程更自然。

我们已经了解了曲线的一些重要性质，尤其是曲线的速率和曲线上每点的法向量。另外，还给出了曲线的光顺性这个微妙的定义。

我们举了一系列曲线的例子，如圆锥曲线，多项式曲线，以及按比例混合的多项式曲线。

我们还讨论了如何基于一组控制顶点来构造平滑变化的曲线，这种方法在计算机辅助几何设计(CAGD)中很常见。设计者可以设置一组数据点，混合生成曲线，从而控制曲线的形状。必须强调插值点的曲线和逼近点的曲线是有区别的。无论哪一种情况，几个控制顶点和算法可以构造曲线上无限个点，每个点对应参数 t 的一个值。

由于贝塞尔曲线的简洁性，它们最先被介绍。这类曲线来自迭代的 de Casteljau 内插过程，这使得它们有许多直观的性质。这类曲线有几类我们所需要的性质，使得它的形状可被预测，因此设计者在设置控制顶点的位置时即可大致确定曲线的形状。

贝塞尔曲线虽然应用在许多设计环境中，但由于作为基础的 Bernstein 多项式是定义在整个区间上的，使得贝塞尔曲线失去了局部可控性。另一个缺点就是随着控制顶点个数的增加，多项式的阶数也随之增加。这会导致曲线不能随心所欲地变化形状，并且计算代价较大和稳定性较差。

因此我们介绍了一类内容更为丰富的基于样条的混合函数。所谓样条即是分段多项式的组合，同一类型的曲线在各点有相同的连续阶。B 样条作为一类特殊的基函数，可以生成任何样条，在造型中应用最为广泛。因此它们为设计者提供了强有力的局部可控性，同时还具有像贝塞尔曲线那样所期望的性质。当 B 样条多项式的阶数增加到与控制顶点个数相同时，B 样条就是 Bernstein 多项式。

NURBS 曲线有更多的方法来控制外形。尽管它更为复杂，但是却使 B 样条成为它的一个特殊形式，而且能够精确表示圆锥曲线。支持 NURBS 算法的 CAD 系统能够使设计者运用一个统一的工具来生成各种不同形状的曲线。

我们讨论了用于插值而不仅仅是逼近控制顶点的曲线算法。我们着重介绍了分段三次多项式曲线，而且研究了当曲线不仅插值而且在各个连接点处有预定速率时，各个系数需要满足的条件。这个速率可由用户指定，或者由为得到在连接处更大的光顺性的约束指定，或由相邻控制顶点的局部几何信息指定。设计者也可以利用额外的参数，比如张力和偏移量，来更加有力地控制曲线的最终外形。

B 样条也可用于插值，但是它通常只能插值首末端点而不是所有控制顶点。这里有一个技巧，即计算一组新的控制顶点，使得由此生成的 B 样条曲线恰好通过所有原始控制顶点。

我们扩展了对曲线设计技术以设计不同类型的曲面，包括直纹面，旋转曲面，二次曲面。我们也讨论了使用贝塞尔，B 样条，有理 B 样条函数来设计曲面。可以把贝塞尔曲面片的生

成想象成一条形状不断变化的贝塞尔曲线在空间中的扫掠。这条移动中的贝塞尔曲线上的每一点其轨迹本身也是一条贝塞尔曲线。若几个贝塞尔曲面片的控制多面体的边界满足特定条件,则这几个贝塞尔曲面片就可以拼接起来。我们也讨论了 B 样条曲面片,它给设计者提供了更大的灵活性。由于 B 样条曲面多项式的阶数不随着控制顶点的增加而增加,所以可以生成非常复杂的曲面。

我们最后也扩展到了 NURBS 曲面族,设计者可以通过改变一组权因子来改变曲面片的形状,从而得到更大的灵活性。NURBS 曲面具有仿射不变性和投影不变性。此外,可以特殊化 NURBS 曲面来生成其他类型的曲面,因此设计者可以运用一种 NURBS 曲面算法来构造多种不同类型的曲面。

曲面细分法可以改进简单的和有组织结构的网格模型。利用这种方法对网格进行优化,可以生成新的面片,从而使网格模型在渲染时显得更加光滑。

本章只涉及到最基础的曲面设计方法,它们也已经发展出许多新的技术,大型计算机辅助设计和 3D 模型软件包中常常包含这些技术。设计者可以从中选择,反复调整,直到得到所需结果。一些外形,比如飞机的机翼和帆船的船体,是根据科学原理,美学,直觉,经验等各方面的综合进行造型的。

案例分析

案例 10.1 几类有趣的参数曲线

(难度等级:Ⅱ)**A. 椭圆的一般形式** 椭圆的参数表达方程可用一个正弦函数和一个余弦函数表达。通过叠加几条不同速度的椭圆曲线可以得到一组有趣的曲线。用这种方法计算谐波之和,与第 3 章的傅立叶级数曲线相似,只是这里是在二维上进行。我们先从两项开始,然后再考虑一般情况。设这簇曲线表达式如下:

$$x(t) = X_1\cos(2\pi t) + X_2\cos(2\pi kt)$$
$$y(t) = Y_1\sin(2\pi t) + Y_2\sin(2\pi kt) \tag{10.83}$$

每个等式的第一项表示椭圆,后面添加的是第二个椭圆,其速度是第一个的 k 倍。当 t 从 0 变到 1,第一个椭圆绕了一圈,另一个绕了 k 圈。如果 k 是整数,那么生成的图形是完全闭合的。编程画出这样的周期图形,其输入变量为 X_1, X_2, Y_1, Y_2, k。把更多的项添加到 $x(t), y(t)$ 中,可得到更一般的结果。

B. 圆的渐开线 抓住缠在扫帚柄周围绕圈的细线上的一点。拉紧细线绕着扫帚柄转动,使之展开。P 点所经过的螺旋型轨迹就是一条圆的渐开线。圆(扫帚柄)和 P 的连线就是圆的切线。显然这条切线总是与螺旋线垂直。此外,螺旋线上的各圈是相互平行的,且间距相等(间距多少?)通过旋转这个图形(或选择细线上不同的点 P)可以得到一族这样的螺旋线。每条螺旋线与圆所有的切线正交。曲线的参数方程如下:

$$x(t) = \cos(2\pi t) + 2\pi t\sin(2\pi t) \quad y(t) = \sin(2\pi t) - 2\pi t\cos(2\pi t) \tag{10.84}$$

编程来画出圆的渐开线。

C. 其他正弦类型曲线　对前面练习中的 $x(t)$，$y(t)$ 进行轻微的调整也会生成完全不同的形状。例如，图 10.68 中的曲线（感谢 Mount Holyoke 学院的 Robert Weaver 教授）可由下面等式得到：

$$x(t) = \cos(t) + \sin(8t)$$
$$y(t) = 2\sin(2\pi t) + 7\sin(7t) \qquad (10.85)$$

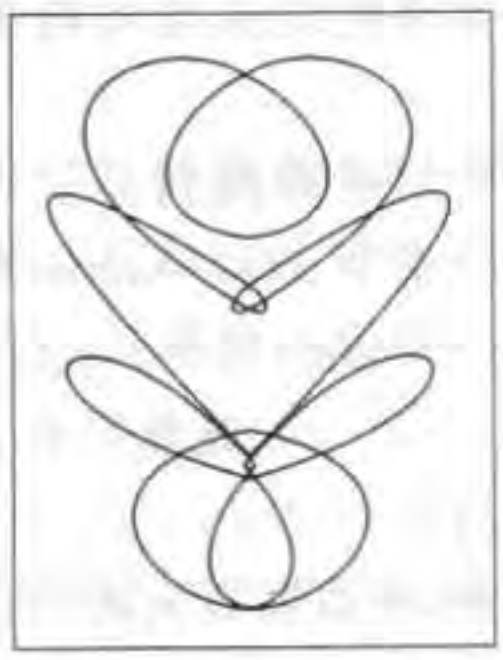


图 10.68　形状古怪的曲线

编程生成这条曲线，并且尝试一下其他变动。若要生成封闭的（周期的）曲线，这些三角函数必须满足怎样的条件？

D. Lissajous 曲线　如果允许两个正弦函数的频率不同，那么将会生成椭圆的一个变体：

$$x(t) = \cos(2\pi Mt + angle) \quad y(t) = \sin(2\pi Nt) \qquad (10.86)$$

这里 M 和 N 是新的频率，*angle* 是两个分量之间的"相位偏移"。这些生成的形状被称为"Lissajous 曲线"，在测试闭合电路时偶尔会出现在示波器上。编程显示 Lissajous 曲线，其输入参数为 M，N 和 *angle*。在实验中选择曲线上的点并大幅度增加 t 的值，观察其形状的变化。如果在曲线绘制完成后互相交换 x 和 y 变量的值，将可以观察到有趣的对称性。

案例 10.2　椭圆形台球桌

（难度等级：Ⅲ）在案例 4.4 中，我们研究了光线如何在多边形房间的墙壁上进行反射。可以考虑其他形状的房间又会怎样。被称为"Elliptipool"的椭圆形台球桌于 1964 年在美国开始销售【Gardner71，Steinhaus69】。如图 11.65 所示，通过跟踪光线在椭圆形房间内的反射情况，可以模拟出椭圆形台球桌。

给定初始光线 $S+\mathbf{c}t$，其起始点 S 在椭圆内。计算光线与椭圆的交点 P 以及反射光线方向 $\mathbf{r}$。从 S 到 P 绘制一条直线用于跟踪这条反射光线。然后把下一条光线设为 $P+\mathbf{r}t$，并重复这个过程。

不妨假设椭圆方程如下：

$$F(P) = \left(\frac{x}{a}\right)^2 + \left(\frac{y}{b}\right)^2 - 1 = 0$$

且假设给定光线为 $S+\mathbf{c}t$。

光线与椭圆的交点：

如果光线与椭圆相交，那么一定存在一个 t 值使得点 $S+\mathbf{c}t$ 在曲线 $F(P)=0$ 上。即 t 满足条件 $F(S+\mathbf{c}t)=0$。所以对于椭圆我们有如下方程：

$$\left(\frac{S_x + c_x t}{a}\right)^2 + \left(\frac{S_y + c_y t}{b}\right)^2 - 1 = 0$$

这是一个关于 t 的二次方程。二次方程很容易求解，可能有零个，一个或两个解。

- 无解：光线不与椭圆相交。
- 一个解：光线与椭圆相切。
- 两个解：光线进入椭圆后又离开椭圆。

如果光线从房间内部出发,那么两个解一正一负(为什么?)。我们使用正值解,记为 t_{hit}。

我们还需找到反射光线在交点处的方向。在等式(10.3)中令 $x(t)=a\cos(t)$,$y(t)=b\sin(t)$即可得到其法向量$(-b\cos(t),-a\sin(t))$,若用 x,y 的形式可写作$(-bx/a,-ay/b)$。既然只有方向才是重要的,那么经过放缩可化简为$(-b^2x,-a^2y)$。由于光线是从椭圆的内壁反射回来,所以需要向内的法向量。通过观察图 10.69,可得当 x 和 y 都为正时,向内的法向量其 x 和 y 分量都是负的,则法向量有如下形式:

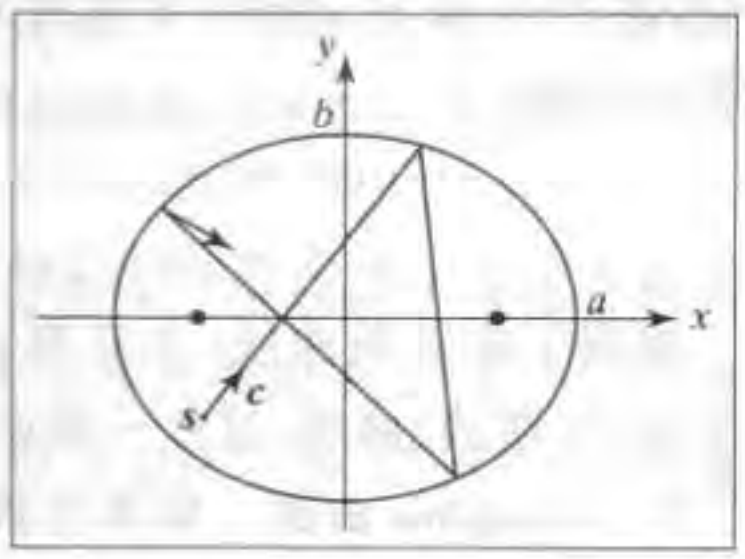


图 10.69 椭圆形台球桌的模拟

$$\mathbf{n}=(-b^2x,-a^2y) \qquad (10.87)$$

它确定了向内法向量的方向。把它代入等式(4.27),可以得到反射光线 r。

对椭圆形台球桌的模拟揭示了一些有趣的现象:光线的路径只有三种【Steinhaus69】。

- 如果光线通过任意一个焦点,则经过反射它会通过另一个焦点。这显然是由于椭圆的反射性质:从一个焦点发出的光线经过椭圆反射后会通过另一个焦点。这样这条光线会一直反复交替地通过两个焦点。几次反射后,它经过的路径就会和 x 轴难以区分了。
- 如果对于光线其初始路径不在两个焦点之间,那么它以后将再也不会经过两个焦点之间了。如图 10.70 所示,它的路径与一个较小椭圆相切,这个小椭圆和原始椭圆有相同的焦点。
- 如果光线的初始路径在两个焦点之间,那么它经过路径将不会终止,而且这条路径与焦点的距离不小于与拥有同椭圆一样焦距的双曲线的距离,如图 10.71 所示。

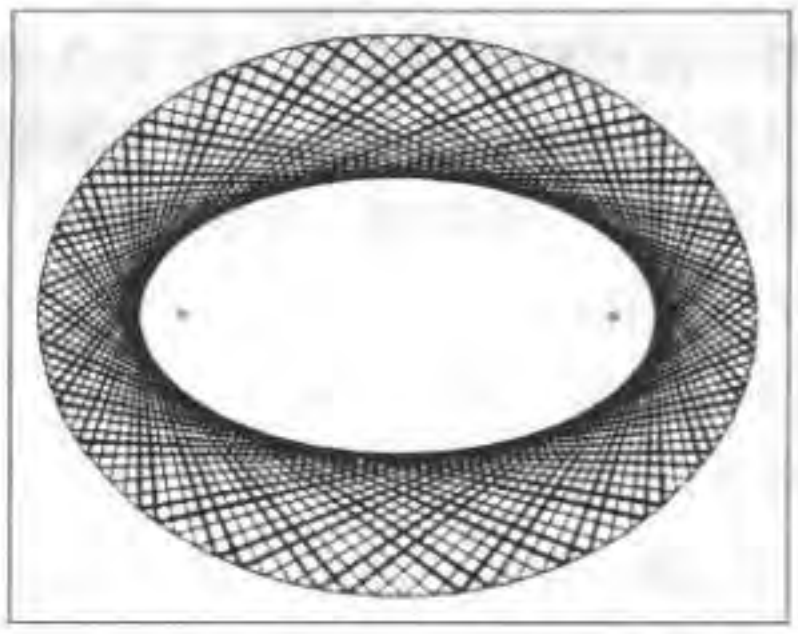


图 10.70 光线与第二个椭圆相切

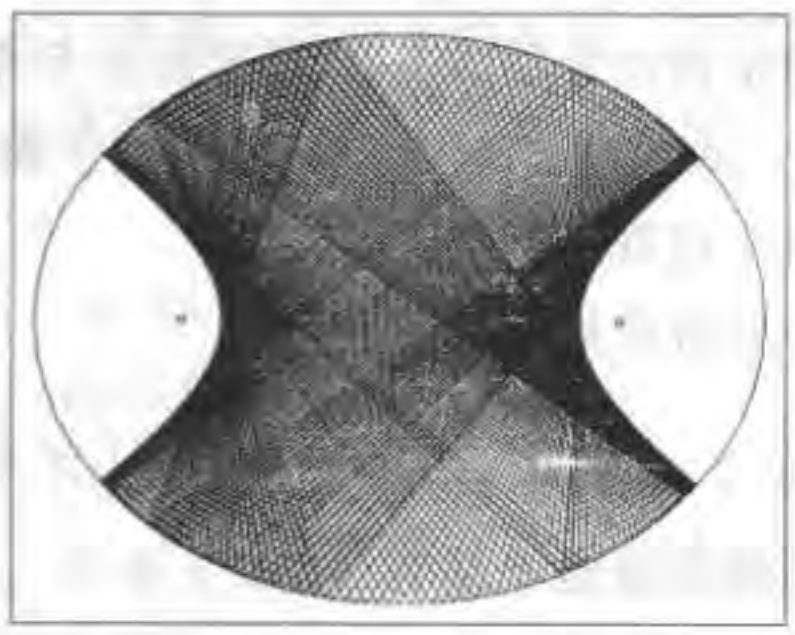


图 10.71 光线与一条双曲线相切

编写并执行一段程序来模拟椭圆形台球桌。用户在某些地方指定光线的初始位置和方向,然后对光线的多次反射进行跟踪。要允许用户指定光线通过一个焦点。(注意此时数学上的舍入误差会产生一些不可预料的影响,如何消除它?)偶尔改变路径的颜色,这样即使显示器被已有路径填满,当前路径仍可被区别显示。在实验中使用包括圆形在内的不同离心率的椭圆形台球桌。

(选做)通过在椭圆形台球桌内放置圆或者其他障碍物(椭圆?)来增强实验。这时是否会有可辨认的图案出现?

案例 10.3　贝塞尔曲线

(难度等级：Ⅱ)编写一个程序来绘制一条贝塞尔曲线，其一系列控制顶点由用户操作鼠标输入。计算曲线上参数值间隔密集的点，并用直线段依次连接。实验中可用不同数量的控制顶点。注意随着控制顶点个数的增加，多项式阶数也随之增加，计算就会变慢。

案例 10.4　二次样条的生成

(难度等级：Ⅱ)等式(10.41)表示了利用混合函数 $g(t-k)$ 的加权之和生成的曲线：

$$V(t) = \sum_{k=0}^{6} P_k g(t-k)$$

这里是由等式(10.38)定义的二次样条函数。编写并执行一段程序来绘制上式中的曲线 $V(t)$，其中的一系列控制顶点 P_k 由用户操作鼠标输入。此时可以编写一个辅助函数 double g(double t)来返回 g(t)的值。特别的，当 t 在区间[0，3]外时返回 0。

案例 10.5　建立样条曲线编辑器

(难度等级：Ⅲ)设计并执行一段程序，使得用户能够操作鼠标生成一个控制多边形 **P**。要求程序能够绘制由 **P** 定义的 B 样条曲线。这个程序可以执行以下命令，用户只需按下键盘上适当的键即可执行这些命令("b"代表 begin，"d"代表 delete，等等)：

- b)egin：(开始一个新的控制多边形 **P**)。
- d)elete：(删除 **P** 中离鼠标所指位置最近的点)。
- m)ove：(把鼠标所指的点拖到新的位置)。
- r)efresh：(绘制 **P**)。
- o)rder('1',…,'9')：(绘制阶数给定的基于 **P** 的样条曲线)。
- c)losed('−1',…,'−9')：(绘制基于 **P** 的封闭 B 样条曲线)。
- e)rase：(清屏)。
- q)uit：(退出程序)。

1. (选做)多个控制多边形　扩充程序的功能，使得它每次可以显示 10 个不同的控制多边形，并且能对每一个进行任意编辑。

2. (选做)能用 B 样条得到圆吗？　选择共圆的四点或八点，试验基于这些点的封闭 B 样条对圆的逼近可以到何种程度？设计一种合理的标准来评估曲线和圆之间的数值误差，并尝试不同的配置来确定最佳的曲线。

3. (选做)B 样条的变换　扩充程序的功能，使得用户可以对曲线进行仿射变换(可以通过预先设置的菜单进行)，并得到一个新的控制多边形。基于新控制多边形的 B 样条就是仿射变换后的 B 样条。

案例 10.6　用 B 样条插值控制顶点

(难度等级：Ⅲ)基于 B 样条混合函数和标准节点向量的曲线只能插值首末控制顶点。但是，通过对控制顶点的预处理，可以使 B 样条曲线插值所有控制顶点。在预处理中，根据原有的控制顶点产生新的控制顶点。根据新控制顶点生成的 B 样条曲线将通过所有给定

的原始控制顶点。

我们通过一个特定的例子来揭示其核心思想。用三次 B 样条函数来插值 6 个点 $y_0,\cdots,y_5$,这些点对应的 t 值间距相等,如图 10.72(a)所示。

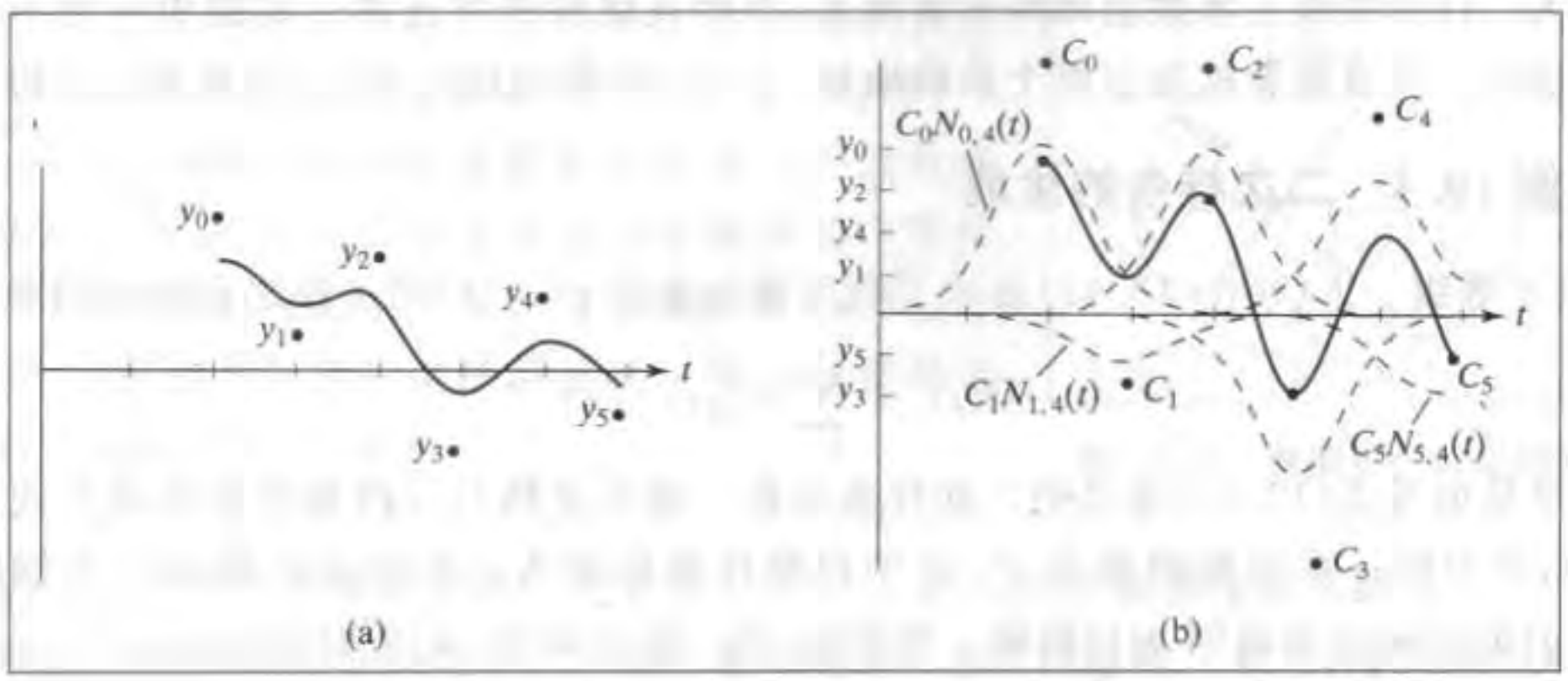


图 10.72 试着用三次 B 样条插值 6 个控制顶点

我们使用等距节点而不用标准节点向量,这样 $N_{0,4}(t)$ 在 $t=0$ 处开始,在 $t=2$ 处"凸起",$N_{1,4}(t)$ 在 $t=1$ 处开始,在 $t=3$ 处"凸起",依次类推。因此我们试着在 $t=2$ 时插值 y_0,$t=3$ 时插值 y_1,依次类推。

如图 10.72(b)所示,下面的等式

$$y(t)=\sum_{i=0}^{5}y_iN_{i,4}(t)$$

不经过所有控制顶点。这是由于 B 样条函数叠加后和式中各项互相影响,导致 $y(t)$ 值不能达到数据点的值。为了纠正这个错误,我们用一组新的值 $c_0,\cdots,c_5$ 来代替 y_i,这样基于如下等式的曲线确实能够插值 y_i,如图 10.72(b)所示。

$$p(t)=\sum_{i=0}^{5}c_iN_{i,4}(t) \tag{10.87}$$

通过求解一个线性方程组,就可以得到这组正确的 c_i 值。插值 6 个点满足条件 $p(2)=y_0,p(3)=y_1,p(4)=y_2,\cdots p(7)=y_5$。由于 t 在取整时,B 样条函数只能在 0,1/6,4/6上取值。所以这 6 个条件有如下形式:

$$\begin{aligned}4c_0+c_1&=6y_0\\c_0+4c_1+c_2&=6y_1\\c_1+4c_2+c_3&=6y_2\\\cdots&=\cdots\\c_4+4c_5&=6y_5\end{aligned} \tag{10.88}$$

这些等式几乎与等式(10.65)和等式(10.66)相同,所以可用 10.10.3 节中的方法求解。

选择一个合适的 L,上述方法可以立刻推广到任意个数的控制顶点 y_0, $y_1,\cdots,y_L$。编写一个函数 void adjust(double y[], double c[], int L),用给定的数组 y[]生成数组 c[]。

插值点 $p_i=(x_i,y_i)$时,对 x 分量和 y 分量各做一次预处理,得到两个数组 x_new[]和 y_new[]。这时插值曲线如下:

$$P(t) = \sum_{i=0}^{L} W_i N_{i,4}(t)$$

这里 W_i=(x_new[i], y_new[i])。实例见图 10.73。

编写一个程序，允许用户操作鼠标输入($L+1$)个控制顶点，由此绘制一条三次 B 样条插值曲线。尝试用不同的 L 进行实验。

(选做)扩展这个程序的功能，使得生成的插值曲线为闭合曲线。那么，为此等式(10.88)需要作何修改？

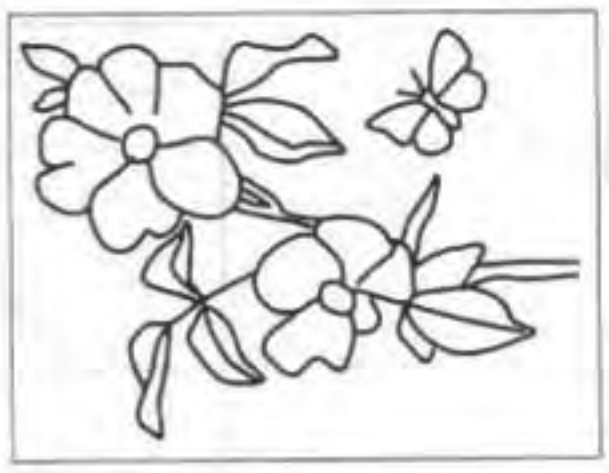


图 10.73　用 B 样条在 2D 上插值的实例

案例 10.7　用三次多项式插值

(难度等级：Ⅲ)编写一个程序，允许用户操作鼠标输入($L+1$)个控制顶点，由此绘制一条基于三次多项式的插值曲线。根据事先指定的张力值，偏移量，连续性，用 Kochanek-Bartels 方法设置内部连接点的速率。对于剩下的两个三次系数的条件，可以强制两个端点处二阶导数为 0。

允许用户使用键盘调整每个内部控制顶点的张力值，偏移量，连续性。比如，当用户单击控制顶点选定后，按“v”键张力将减少一个小的定量，或者按“V”键增加。同样的，按“b”或“B”键调整偏移量，按“c”或“C”键调整连续性。

案例 10.8　经典的茶壶

(难度等级：Ⅱ)用 OpenGL 编程，从不同视角绘制经典的茶壶。不要用 GLUT 版本中的茶壶：用曲面片造型一个属于自己的茶壶。

茶壶有 4 个主要部分组成。**壶体**是一张旋转曲面，其轮廓线由 xz 平面上 3 条贝塞尔曲线组成，详见 10.10.2 节。**把手**由 4 张贝塞尔曲面片组成，详见 10.10.3 节。**壶盖**也是一张旋转曲面，其轮廓线由 2 条贝塞尔曲线组成，如图 10.74，其数据点见图 10.75。

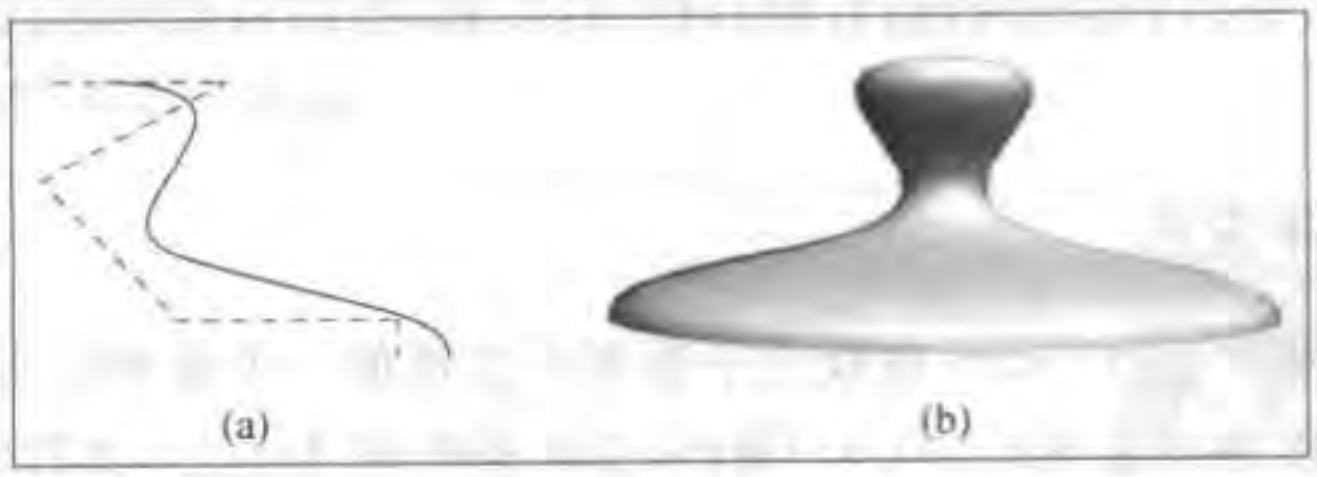


图 10.74　茶壶的壶盖

i	x	z
0	0	3
1	0.8	3
2	0	2.7
3	0.2	2.55
4	0.4	2.4
5	1.3	2.4
6	1.3	2.25

图 10.75　茶壶壶盖轮廓线的数据点

壶嘴与把手类似，也是由 4 张贝塞尔曲面片组成。图 10.76(a)显示了它的横截面，以及横截面处相应的控制多面体。

与把手类似，壶嘴曲面也关于 xz 平面对称。整个 y 正方向一侧的上曲面片有以下 16 个控制顶点：

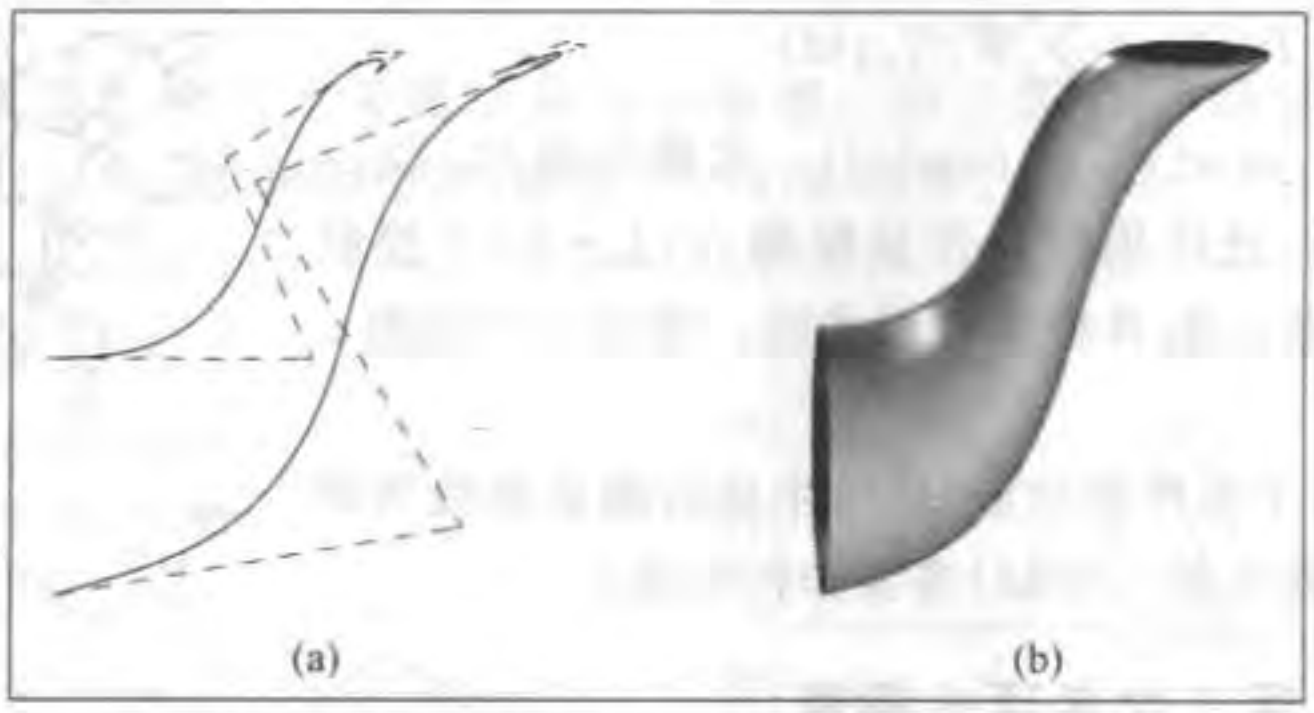


图 10.76 茶壶壶嘴的设计

- 基于 C_0, D_0: (1.7, 0.0, 0.45),(1.7, 0.66, 0.45),(1.7, 0.66, 1.275),(1.7, 0.0, 1.275)
- 基于 C_1, D_1: (3.1,0.0,0.675),(3.1,0.66,0.675),(2.6,0.66,1.275),(2.6,0.0,1.275)
- 基于 C_2, D_2: (2.4,0.0,1.875),(2.4,0.25,1.875),(2.3,0.25,1.95),(2.3,0.0,1.95)
- 基于 C_3, D_3: (3.3,0.0,2.25),(3.3,0.25,2.25),(2.7,0.25,2.25),(2.7,0.0,2.25)

在 y 轴正方向一侧的下曲面有 16 个控制顶点:

- 基于 C_3, D_3: (3.3,0.0,2.25),(3.3,0.25,2.25),(2.7,0.25,2.25),(2.7,0.0,2.25)
- 基于 C_4, D_4: (3.525,0.0,2.34375),(3.525,0.25,2.34375),(2.8,0.25,2.325),(2.8,0.0,2.325)
- 基于 C_5, D_5: (3.45,0.0,2.3625),(3.45,0.1,2.3625),(2.9,0.1,2.325),(2.9,0.0,2.325)
- 基于 C_6, D_6: (3.2,0.0,2.25),(3.2,0.15,2.25),(2.8,0.15,2.25),(2.8,0.0,2.25)

案例 10.9 NURBS: 投影不变性

(难度等级: Ⅰ)A. 这个案例提供了 NURBS 曲线关于投影不变性的一个逐步证明。其中的细节即具体步骤。设 NURBS 曲线由等式(10.57)给出,变换矩阵 M 为 4×4,其行向量分别是 $\mathbf{m}_1, \mathbf{m}_2, \mathbf{m}_3, \mathbf{m}_4$。

a. 经过变换 T,等式(10.57)在齐次坐标下的曲线 $\widetilde{P}(t)$ 成为如下的 $M\widetilde{P}(t)$:

$$M\widetilde{P}(t) = \sum_{k=0}^{L} \omega_k (\widetilde{P}_k \cdot \mathbf{m}_1, \widetilde{P}_k \cdot \mathbf{m}_2, \widetilde{P}_k \cdot \mathbf{m}_3, \widetilde{P}_k \cdot \mathbf{m}_4)^T N_{k,m}(t)$$

这里 $\widetilde{P}_k = (P_x, P_y, P_z, 1)^T$ 是齐次坐标下的 P_k。

b. 在一般坐标下,上式即为:

$$T(p(t)) = \frac{\sum_{k=0}^{L} \omega_k (\widetilde{P}_k \cdot \mathbf{m}_1, \widetilde{P}_k \cdot \mathbf{m}_2, \widetilde{P}_k \cdot \mathbf{m}_3)^T N_{k,m}(t)}{\sum_{k=0}^{L} \omega_k (\widetilde{P}_k \cdot \mathbf{m}_4) N_{k,m}(t)} \tag{10.89}$$

c. 显然每一个控制顶点 P_k 变换为：

$$T(P_k) = \left(\frac{\widetilde{P}_k \cdot \mathbf{m}_1}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_2}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_3}{\widetilde{P}_k \cdot \mathbf{m}_4} \right)^T$$

d. 用以上控制顶点和一组权因子，比如 v_k，生成的 NURBS 曲线如下：

$$\frac{\sum_{k=0}^{L} v_k \left(\frac{\widetilde{P}_k \cdot \mathbf{m}_1}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_2}{\widetilde{P}_k \cdot \mathbf{m}_4}, \frac{\widetilde{P}_k \cdot \mathbf{m}_3}{\widetilde{P}_k \cdot \mathbf{m}_4} \right)^T N_{k,m}(t)}{\sum_{k=0}^{L} v_k N_{k,m}(t)}$$

e. 如果 v_k 取等式(10.60)中的 $\bar{\omega}_k$，那么上式就和等式(10.59)相同。

B. 为什么 B 样条没有投影不变性　说明前面的推导步骤中哪一步对 B 样条不成立。

C. 扩展到曲面　用与上面类似的步骤，证明 NURBS 曲面也具有投影不变性。

案例 10.10　NURBS 曲线的绘制

（难度等级：Ⅱ）

a. 编写并执行一个应用程序，用于绘制 NURBS 曲面片的 u 线和 v 线。它的核心内容是一个计算曲面 $P(u,v)$ 上点的函数，其原型可能是：

```
Point3 nurbsPoint(Point3 P[][],          //控制顶点的矩阵
        int L, int M,                    //控制顶点个数为(L+1)(M+1)
        float w[],                       //权因子向量
        float knot[],                    //节点向量
        int m, int n,                    //B 样条的阶数
        float u, v);                     //参数 u, v 的值
```

b. 用图 10.77 平面视角中的控制多面体进行实验，这里的 0、a 或 b 表示各点在 xy 平面上方的高度。图 10.77(b)显示了在选择特定权因子以及高度 a 和 b 后所生成的“穹顶状”NURBS 曲面。

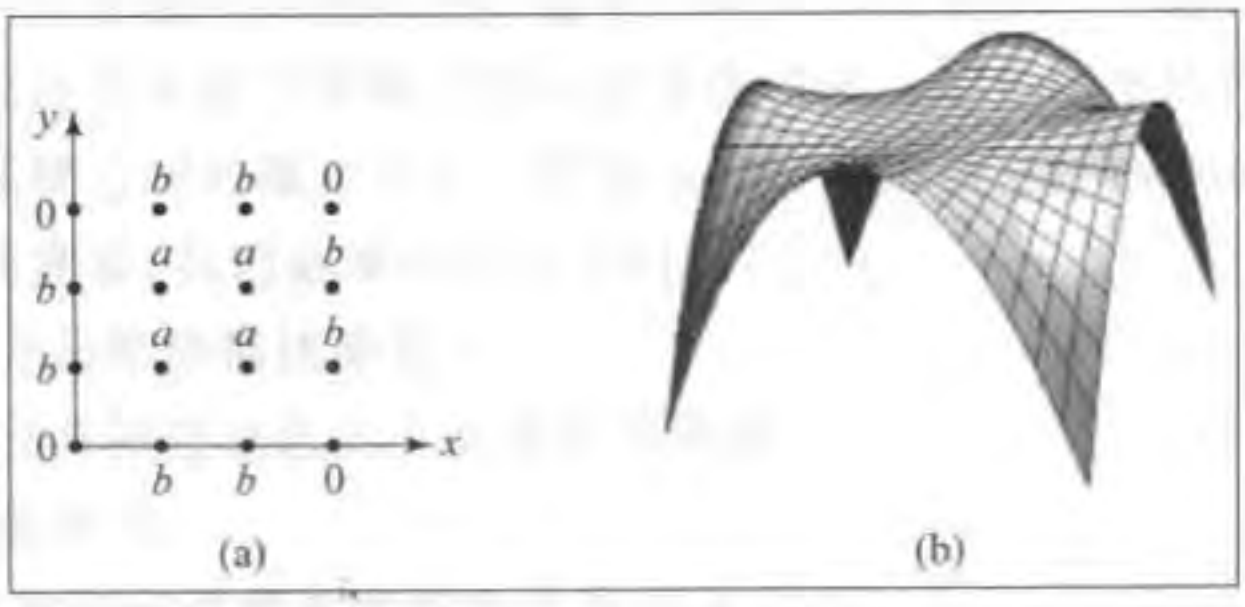


图 10.77　设计一个屋顶形状

在用户指定 a 和 b 后，按照上述方法构造和绘制曲面片。试用以下类型的权值：

i. 所有权因子相等。

ii. 高度为 0 的权因子为 W,其余点权因子为 sW,其中 s 值由用户输入。

iii. 其他感兴趣的权因子组合。

以上组合能否生成一个接近于半球的屋顶曲面?怎样调整才能使屋顶曲面更接近于半球?

c. 构造拉伸曲面和直纹面。尝试用 NURBS 曲面构造拉伸(线性扫描)曲面。以 NURBS 曲线 $C(u)$ 为轮廓线,沿着着垂直于轮廓线所在平面的方向扫掠,就可得到拉伸曲面。直线侧边是由 v 上的一阶 NURBS 曲线构成。再构造直纹面,其两条边界曲线都是 u 上的 NURBS 曲线,母线是 v 上的一阶 NURBS 曲线。

进一步阅读

Faux 和 Pratt【Faux79】在早年写了一本十分优秀的书,介绍了曲线曲面设计中的数学基础。Rogers 和 Adams【Rogers90】的书提供了大量在曲面设计中用到的方法,Farin【Farin90】也仔细地介绍了曲面中用到的数学。Bartels【Bartels87】讨论了曲线曲面设计中种类繁多的样条,以及它们各自不同的性质。由 Bloomenthal【Bloomenthal97】编辑的书中几个关于曲线曲面设计的章节很精彩,包括一章 Blinn 所写的关于 2 阶曲面和一章 Bajaj 所写的隐式曲面。Rogers【Rogers00】从历史的观点很好地介绍了 NURBS 曲线曲面。还可以看最近 N. Bunnel 在"GPU Gems2"一书中的"Adaptive Tesselation of Subdivision Surfaces with Displacement Mapping"以及 R. Pajarola 与 J. Rossignac【Pajarola00】在期刊"IEEE CG&A"中的 Compressed Progressive Meshes,这些都是关于细分曲面的。

第11章 颜色理论

本章学习目标

- 研究颜色的性质及其数值描述
- 分析颜色描述的一些标准
- 定义和使用各种颜色空间
- 学习几种减少图像颜色种数的方法
- 研究对颜色查找表进行编程的方法

预览

11.1 节介绍人类复杂的颜色视觉系统，及如何用可靠且可复制的数值方法描述颜色。11.2 节讨论颜色的匹配过程，及如何用三原色的线性组合来表示任何颜色，此外还将说明如何选择三原色最合适。11.3 节介绍 CIE 标准色度图的主要思想，并说明其在颜色计算中的应用，另外还将讨论色域的概念。11.4 节介绍几种不同的颜色空间，并给出几个工具用于在不同颜色空间之间转换颜色。11.5 节介绍几种颜色量化算法，颜色量化是指在不降低视觉质量的条件下减少一幅图像的颜色数。

11.1 概　述

前面章节中介绍的光照模型通过分别计算红、绿、蓝三原色分量从而得到最终颜色。这种方法是可行的，下面将证明事实上这和人眼的工作原理有关。此外，这也是和现今大多数普通显示设备混合不同数量的内置红、绿、蓝色从而得到任意颜色的过程一致。

但是颜色的形成过程比这个简单方法远远复杂的多。颜色的形成依赖于光线的物理辐射和人眼——大脑系统之间的微妙交互。更深入的研究请参见【Feynman63】。从开发计算机图形应用程序的角度看，我们必须回答以下问题：

- 如何用数值方法准确描述颜色？
- 这些方法和日常描述颜色的方法有什么关系？
- 如何比较颜色异同？
- CRT 显示器和打印机能显示的颜色范围是多少？
- 怎样加载颜色查找表才能生成所需颜色？
- 当某个设备能显示的颜色数较少，比如说只有 256 色时，应该如何处理颜色范围？

为了开发能正确显示颜色的程序,需要方便的工具来描述和控制颜色,故本章将着手处理这个棘手的问题。

和长波、红外线和 X 射线一样,光本身也是一种电磁现象。所谓光,就是波长[①]在很窄的带宽范围,即可见光谱,内的电磁波。图 11.1 展示了可见光谱(对人类而言)在整个电磁波谱内的位置,图中同时还显示了一些其他常见电磁波的光谱。电磁波振动频率 f 向右边增大,而波长 λ 向左边增大[②]。人眼可见光的波长大约在 400nm～700nm(纳米)之间。

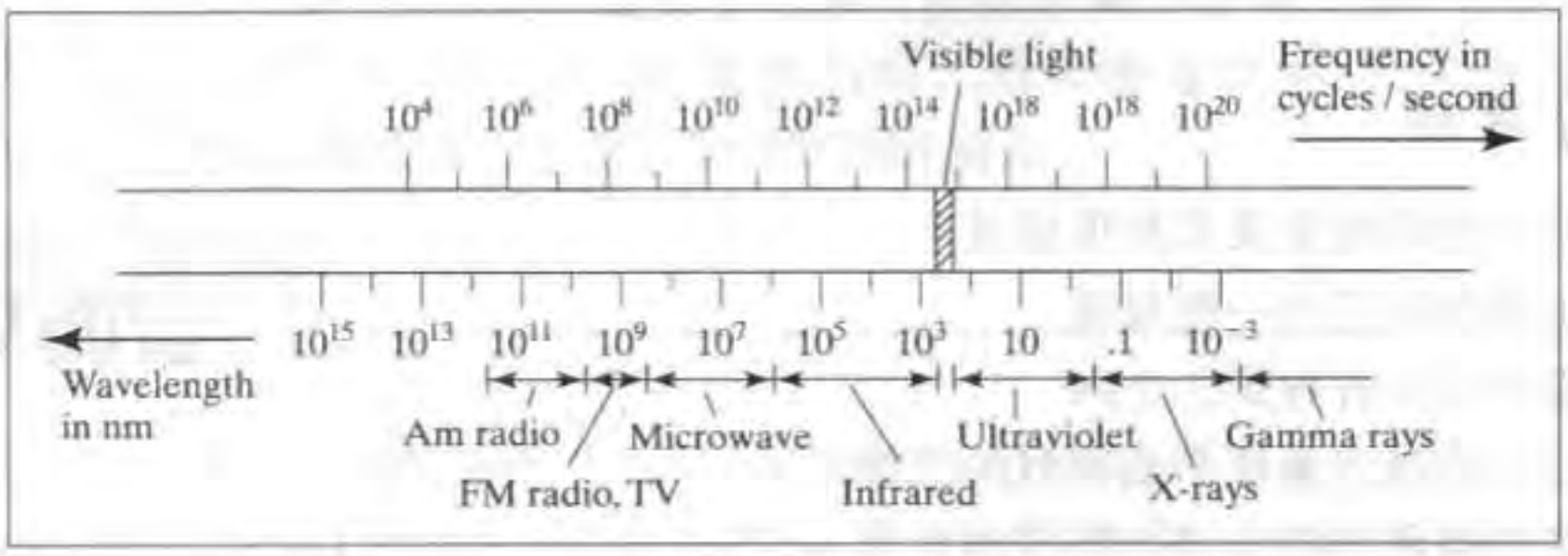


图 11.1 电磁波谱

11.1.1 眼睛:人类颜色感知的生理学基础

视网膜是人眼中感受光的一层薄膜。它位于眼球后部,包含两类神经感知细胞:视锥细胞和视杆细胞。

视锥细胞是对颜色敏感的细胞,每个视锥细胞感应某种特定颜色:红色、绿色或蓝色。根据三色刺激理论(将在后面分析说明),我们看到的颜色是视锥细胞对红、绿和蓝光的相对响应的结果。人眼大概能区别 200 种强度的红色、绿色或蓝色。每个眼睛有 600 万～700 万个视锥细胞,主要集中在视网膜上称为中央凹的一小块区域内。每个视锥细胞有它自己的神经细胞,从而允许眼睛分辨很小的细节。为了看清物体的细节,人眼将直视目标,从而使物体成像于中央凹。

相反,视杆细胞既不能辨别不同颜色,也不能看清细节。视网膜上围绕中央凹集中了 7500 万～15000 万个视杆细胞。此外,许多视杆细胞只连接到单个神经细胞,因而妨碍它们辨别微小细节【Gonzalez 和 Wintz87】。那么视杆细胞有什么作用?它们对低强度光线特别敏感,可以看到暗淡光线下视锥细胞看不到的物体。因此,晚上最好在稍远处注视物体,从而使物体成像于视网膜的中央凹之外。虽然丢失了细节和颜色,但至少看到了物体的大致轮廓。事实上,人眼在暗淡光线下对周围景象的视觉敏感度对人类的进化是很重要的。

有些光源,例如激光,发射单一波长的光,也称"纯光谱"光。人眼感知 400nm 光为紫色,620nm 光为红色,其他纯光谱色的波长在这两个极值之间。图 11.2 显示了一些纯光谱

① 波长是电磁波在一个振动周期内传播的距离。

② 波长 λ 和频率 f 成反比:$\lambda = v/f$,其中 v 为光在相应介质中的传播速度。在空气(或真空)中 v=300000 千米/秒;在玻璃中的速度为空气中的 65%。

色的光谱密度 $S(\lambda)$(单位波长功率)及其视觉颜色的常用名称。

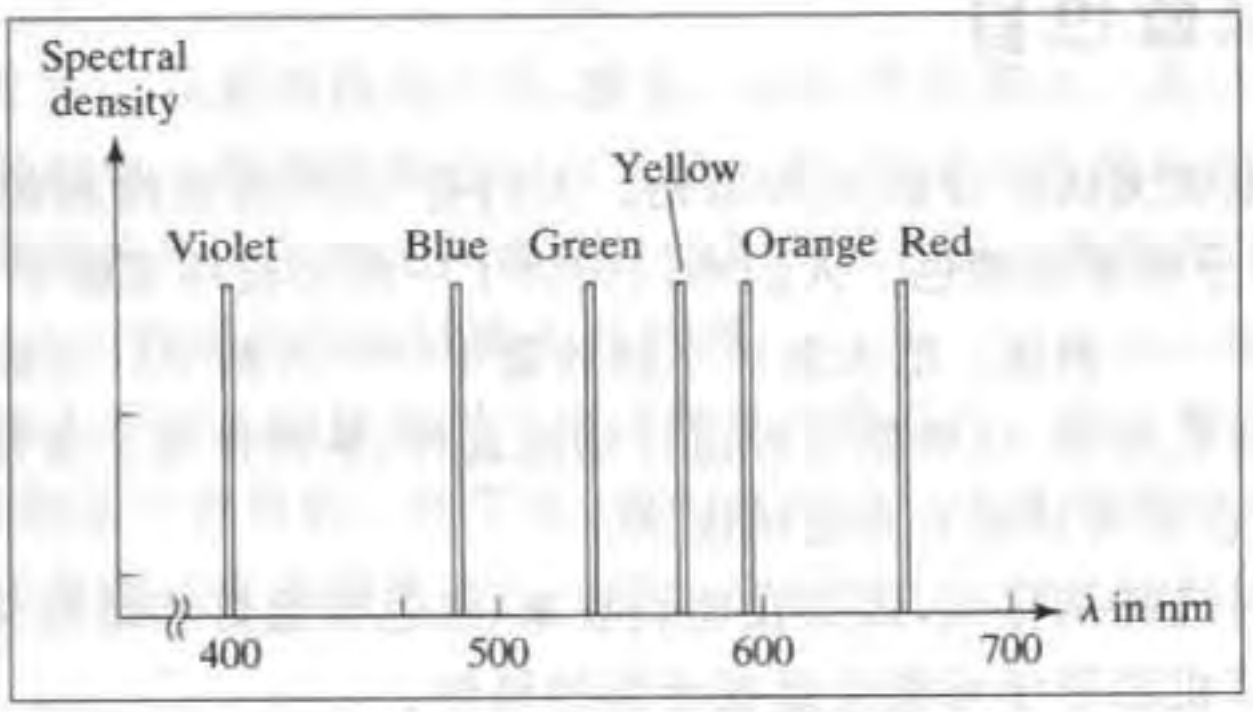


图 11.2 一些纯光谱色的光谱

大多数光源的光不是单一波长的,它们包含某一连续范围内的不同波长,相应每种波长的总能量也互不相同。它们的光谱密度集合(或称"频谱")覆盖一个波长带。光在任何波段范围内的总能量为此波段内光谱密度曲线下(和横坐标轴围成)的面积。图 11.3 展示了几种常见颜色的光谱密度曲线。

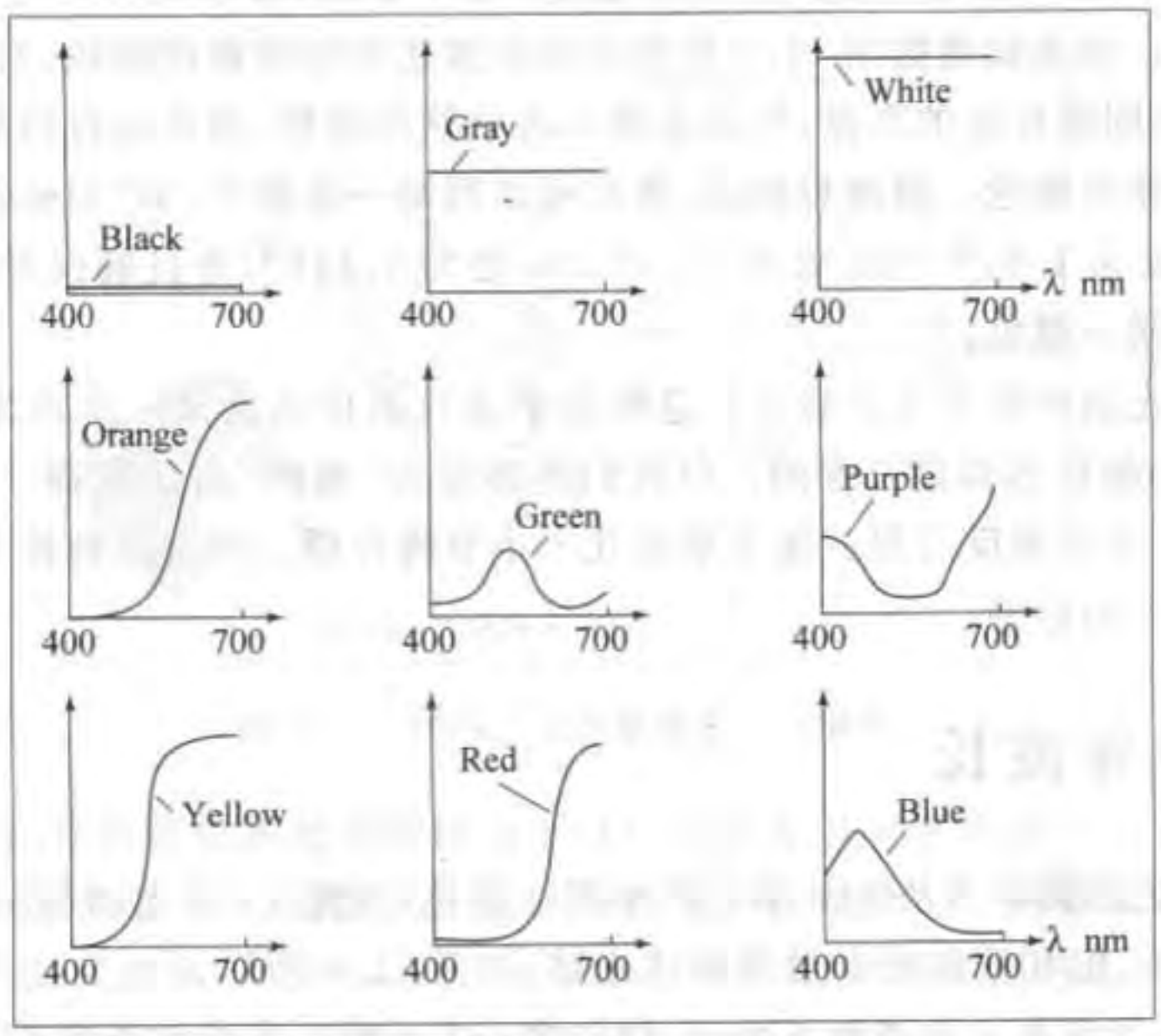


图 11.3 一些典型光线的光谱及其视觉颜色

每条曲线标注有人眼看到该种光时感知到的颜色。注意到白光在所有频率处包含近似等量的能量,而红光的能量集中于长波长处。灰色光也有较"平坦"的光谱密度,但是其强度较低。这些例子都突出了用数值方法描述颜色的困难之一:许多不同的光谱密度函数在人眼看来有相同颜色。例如,给定任何颜色样本,可以和很多不同的光谱密度形状相匹配,也就是说把它们并排放在一起时,样本颜色和其中任一种光谱密度对应的颜色是人眼无法分辨的。

11.1.2 红绿蓝色盲

有些人的视觉系统无法区分红光和绿光。大约有 10%的男性和相对少得多的女性是“色盲”,即他们无法分辨某些颜色。人们专门设计了一些实验用来确定被测试者是否色盲。本书主页展示有这样一个测试:把大量的小圆放置于一个大圆中,(你能看见数字吗?)小圆分别用稍微不同的颜色绘制,这些颜色经过仔细地选择,从而使有正常视觉的人可以看到大圆中的数字图案,而色盲者只能看到随机图案。

色盲者过马路时应特别仔细,因为在他们看来,红色交通灯和绿色交通灯是一样的!同样,在他们看来,计算机图形学可能会是很奇怪的科学。

11.2 颜色描述

假设要在电话里精确地描述一种颜色,所有描述必须都是口头的。对方可能是染料生产商或出版公司的生产经理。仅仅说“亮紫蓝色”是不够的,必须确保对方精确领会我们对颜色特征的描述。如果知道如图 11.3 所示的目标颜色的频谱密度曲线,就可以试着描述曲线在近似 12 个不同波长处的高度,但是这种方法显然很糟糕,而且太过精确,因为许多不同频谱曲线生成同样的颜色。最理想的话,我们可以列举一些数字,如“目标颜色在 400nm 处为 3.24,500nm 处为 1.6,600nm 处为 85,680nm 处为 1.117”,并且确保对方根据此描述能精确地重新生成同一颜色。

那么需要从光谱中取出多少数字?这些数字又代表什么含义?值得注意的是,答案是 3 个数字:人眼的颜色感知是三维的。但我们还需要对“编码”方法取得一致意见:如何将颜色映射为数字,及其相反过程。接下来的几个小节将介绍一些常用转换方法,然后讨论当前国际标准所包含的思想。

11.2.1 主导波长

一种描述颜色的简单方法是列举能产生同一颜色(视觉上)的光谱范围。这种方法用 3 个数字:主导波长、饱和度和光通量来描述光谱,如图 11.4 所示。

在此例图中,光谱在主导波长 620nm 处包含一个峰值。主导波长的位置规定了颜色的色调,此例中为红色。此外还有一定量白光,表现为矩形“底座”,降低红色的饱和度,使之呈现粉红色。

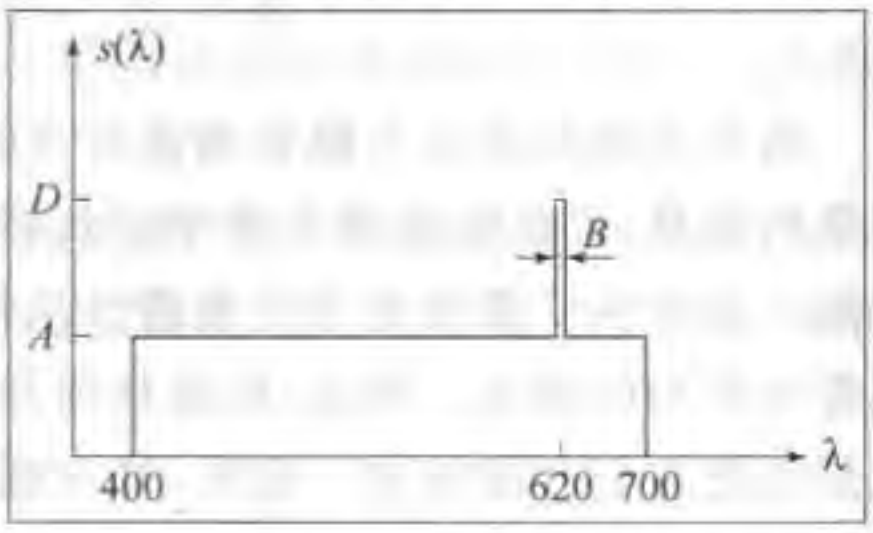


图 11.4 一种有主导波长的颜色的光谱

光的总能量,即光通量,由整个光谱曲线下的面积 L 给出。由于这里所说的各种光谱都是长方形状的,所以这个计算很简单:$L=(700-400)A+DB$。光的饱和度(或称纯度)定义为光的主导成分的光通量占总光通量的百分比【Billmeyer81】:

$$纯度 = \frac{(D-A)B}{L} \times 100\% \tag{11.1}$$

若 $D=A$，则纯度为 0，视觉颜色为白色，没有一点红色的痕迹。若 $A=0$，则一点白色都没有，人眼看到的是纯红色。柔和的颜色包含大量白色，称之为不饱和的。当两种颜色只有色调不同时，人眼大概能辨别 128 种不同颜色。当两种颜色只有饱和度不同时，人眼大概能辨别 20 种不同颜色，这个数字具体还依赖于其色调。

饱和度，光通量和主导波长的概念对于描述颜色非常有用。但是对于具体的一种颜色，如何准确度量这些值却是不容易的。接下来，我们将介绍一些更有效的方法用于描述颜色。首先，我们需要方法用来测试两种颜色是否一样。这就引出了颜色的匹配问题，这是描述颜色的基础。

11.2.2　颜色感知和匹配

常用描述颜色的方法是将其与一系列标准颜色样本比较，从中找出最匹配的颜色。人们设计了许多这类标准颜色集合，并广泛用于染料和印刷业【Munsell41】。如图 11.5 所示，也可以通过适当地组合一些测试光，然后和目标颜色做比较。

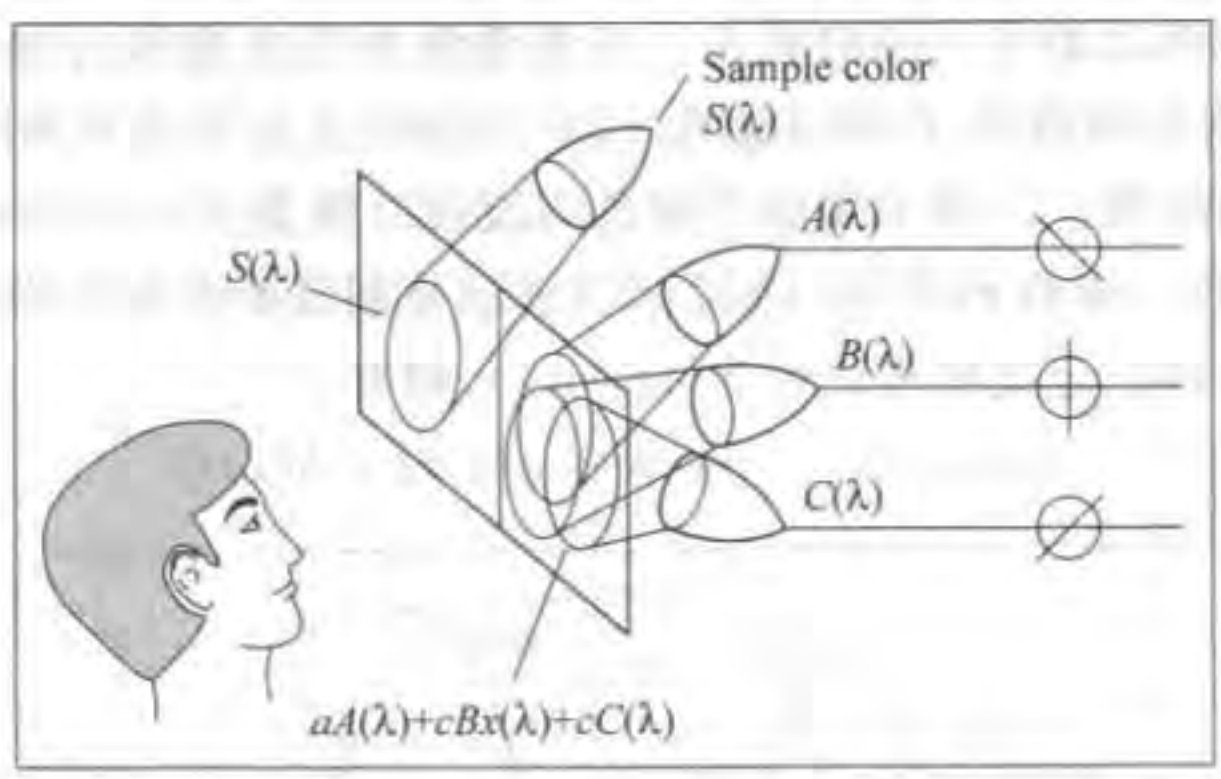


图 11.5　用测试色的叠加来匹配颜色

如上图所示，记样本颜色的光谱密度为 $S(\lambda)$，它被投射至屏幕的一个区域。三种光谱密度分别为 $A(\lambda)$、$B(\lambda)$和 $C(\lambda)$的测试光在屏幕的另一区域叠加。观察者调节测试光的强度(a、b 和 c)，直到测试颜色 $T(\lambda)=aA(\lambda)+bB(\lambda)+cC(\lambda)$和样本颜色无法区别，即使两个光谱 $S(\lambda)$和 $T(\lambda)$差别很大也没关系。这时就可以说样本颜色由 a、b 和 c 单位的 3 个测试颜色组成。在什么意义下，这么说是有道理的？

人脑有着非同寻常的颜色叠加代数机制【Feynman63】。若 $S(\lambda)$和 $P(\lambda)$两个光谱形状，在人眼感知下是同一颜色，我们记这一事实为 $S=P$。现在给这两个颜色同时添加第三种颜色 N，即把光谱 $N(\lambda)$同时叠加至前两者的光谱。实验证明，新生成的两种颜色仍然是人眼无法区别的！

除了符号＝表示两种颜色不可分辨之外，我们还定义符号＋，用 $S+N$ 表示光谱 $S(\lambda)$和 $N(\lambda)$相加产生的颜色。因此前述实验结果可以写为：

$$若\ S=P,\quad 则\ N+S=N+P \tag{11.2}$$

同样可以定义颜色的缩放,即对其光谱密度或总体亮度进行缩放。若 S=P,则对任何(正)标量 a 有 $aS=aP$。从而对两种颜色 A 和 B 进行线性组合是有意义的,例如 $T=aA+bB$,其中 a 和 b 为标量(回忆第四章中对向量的类似操作)。因此可以设计实验验证颜色的向量代数。在这个实验中,我们把颜色当作向量,对其进行加减、缩放、分解成不同分量等各种操作。

如前所述,关于人眼颜色感知系统的另一个重要事实是它是三维[①]的:任何颜色 C 都可以由三种原色,如红、绿和蓝色按下式叠加构造:

$$C = rR + gG + bB \tag{11.3}$$

其中 r、g 和 b 三个数分别描述每种原色在 C 中所占数量。符号 R、G 和 B 分别表示红、绿和蓝色,它们经常被作为三原色讨论。选用红、绿和蓝色作为三原色的原因是:人眼的视锥细胞对这三种颜色特别敏感。当然,等式(11.3)对其他三原色也适用,只要任一原色不是另外两种原色的线性组合即可。

给定一组三原色:R、G 和 B,任何其他颜色 $C=rR+gG+bB$,可以用三维空间中的一个点 (r,g,b) 表示。例如,若 R、G 和 B 分别对应红、绿和蓝色,则(0,1,0)表示单位亮度的纯绿色,(0.2,0.3,0.5)表示黄色。如果把每个分量增加一倍,则得到的颜色色调不变而亮度加倍。

有人做过一些实验测试人眼如何匹配颜色。其中特别有趣的是,对 R、G 和 B 进行线性组合得到(看起来)和纯光谱色相同的颜色。纯光谱色是完全饱和的单色,其能量集中于单一波长 λ。(用主导波长的话说,此纯光谱色 100%饱和,且主导波长为 λ)。图 11.6 显示了对大量测试者的试验结果。其中采用的三原色为波长分别为 700nm、546nm 和 436nm 的纯单色红色、绿色和蓝色。函数 $r(\lambda)$,$g(\lambda)$ 和 $b(\lambda)$ 表示分别需要多少这样的红、绿和蓝色来匹配波长为 λ 的纯光谱色。记此纯光谱色为 $mono(\lambda)$,则有:

$$mono(\lambda) = r(\lambda)R + g(\lambda)G + b(\lambda)B \tag{11.4}$$

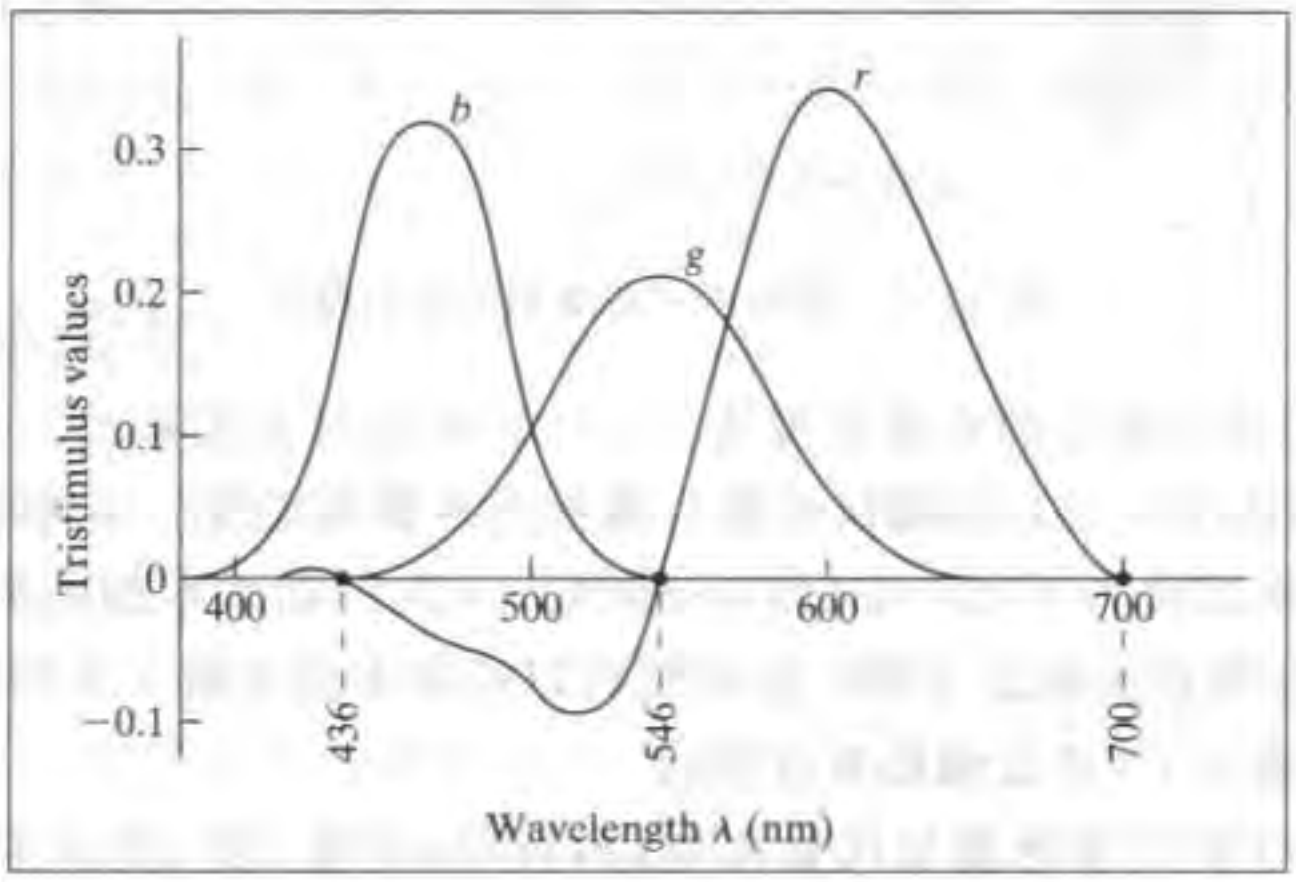


图 11.6 RGB 三原色的颜色匹配函数

例如,纯橙色 $mono(600)$ 在一般人看来和 $0.37R+0.08G$ 合成的颜色是一样的。显然,两者的光谱是不同的,但是两者看起来却完全相同。

① 换种说法就是:任何四种颜色总是线性相关的,即其中任一种颜色总是可以表示为另外三种的线性组合。

特别要提一点：对于前面使用的原色 R、G 和 B，数值 r、g 和 b 中某些可能必须是负值，才能使等式(11.4)成立！例如 $r(\lambda)$ 在 $\lambda=520$ 处是负值。由于我们不能去除尚不存在的颜色，那么颜色线性组合等式中的负号，例如 $C=0.7R+0.5G-0.2B$，代表什么物理意义？幸运的是，如果我们将上述等式重写为 $C+0.2B=0.7R+0.5G$，则前述矛盾消失了。这是说 C 本身不能由三原色的正叠加来构造，但是颜色 $C+0.2B$ 可以和 R 与 G 的适当的凸线性组合匹配。当我们取任何可见色 R、G 和 B 作为三原色时，这种现象就会发生。大部分颜色可以由三原色构造(使用正系数 r、g 和 b)，而其他颜色则不行，必须将其中一种原色放在等式的左边。简单地说，原因是两种颜色相加的和是不饱和色，故不可能由两种颜色的叠加来构造高度饱和的颜色。这对任何纯光谱色特别明显，因为它们本身是完全饱和的。

将颜色匹配等式的系数进行归一化(统一缩放使其和为 1)是很有用的，故定义：

$$\bar{r}(\lambda)=\frac{r(\lambda)}{r(\lambda)+g(\lambda)+b(\lambda)},\quad \bar{g}(\lambda)=\frac{g(\lambda)}{r(\lambda)+g(\lambda)+b(\lambda)},$$

$$\bar{b}(\lambda)=\frac{b(\lambda)}{r(\lambda)+g(\lambda)+b(\lambda)}$$

则显然有 $\bar{r}(\lambda)+\bar{g}(\lambda)+\bar{b}(\lambda)=1$。这些相对权重被称为 $mono(\lambda)$ 的色度值。它们给出了匹配单位亮度 λ 波长的颜色所需要的三原色的量。固定亮度不变，则可以只用两个数值，例如 $(\bar{r}(\lambda),\bar{g}(\lambda))$，来描述颜色，因为总是可以根据 $\bar{b}(\lambda)=1-\bar{r}(\lambda)-\bar{g}(\lambda)$ 确定 $\bar{b}(\lambda)$。

令 λ 在可见光谱内变化，我们可以绘制出三维点 $(\bar{r}(\lambda),\bar{g}(\lambda),\bar{b}(\lambda))$ 的轨迹，如图 11.7 所示。由于经过了归一化，曲线上所有点位于 $r+g+b=1$ 平面内。注意到，当 λ 取某些值时，曲线上的点的一些坐标为负值，即此曲线不完全位于第一卦限内。接下来讨论的 CIE 标准，对此曲线进行一些变换，从而使它完全位于第一卦限内，即三个坐标值处处为正。

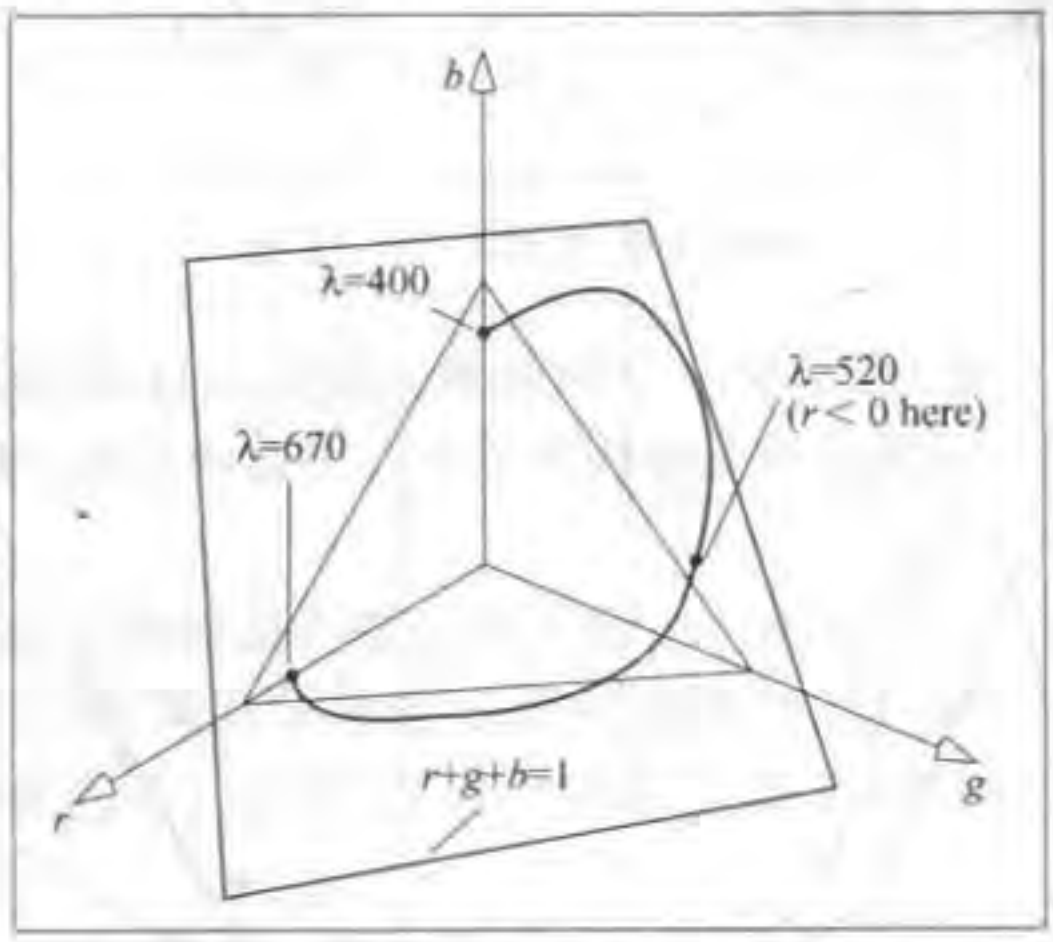


图 11.7　RGB 三原色的纯光谱色曲线

11.3　CIE 标准

怎样才能精确地描述颜色，并且使任何人都同意？由于颜色感知是三维的，只要对三原色达成一致意见，就可以用适当的三元数如 (r,g,b) 来描述颜色。那么该采用哪三种原色？

遗憾的是,使用任何可见色作为原色,都会导致描述有些可见色需要负系数。

为了解决这个难题,国际照明委员会(CIE)于 1931 年制定了 CIE 标准。CIE 标准定义了三种特殊的"过饱和"原色 X、Y 和 Z。CIE 标准经常被称为 **XYZ 模型**。其中的三原色不对应任何真实存在的可见色,但它们却有个性质:所有可见色可以表示为它们的凸线性组合。它们由类似图 11.6 所示的颜色匹配函数定义。

图 11.8 展示了 CIE 标准采用的颜色匹配函数。其概念和等式(11.4)是一样的:波长为 λ 的单色光和三原色的某个特定的线性组合相匹配:

$$mono(\lambda) = x(\lambda)X + y(\lambda)Y + z(\lambda)Z \tag{11.5}$$

注意到三个函数在任意 λ 值处均大于零,所以 $mono(\lambda)$ 总是三原色的凸线性组合。

那么 X、Y 和 Z 是怎么确定的?它们是通过对等式(11.4)中的颜色匹配函数即 $r(\lambda)$、$g(\lambda)$、$b(\lambda)$ 进行仿射变换来定义的,$x(\lambda)$、$y(\lambda)$ 和 $z(\lambda)$ 都是这些曲线的特殊的线性组合。为了方便起见,要求 $y(\lambda)$ 和**光效函数**有同样形状,**光效函数**即人眼对单位辐射亮度不同波长的单色光的反应函数。这就使得任意颜色包含的 Y 原色的量和它的总体亮度相等。

再次将色度值归一化以保持单位亮度是很有用的,因此我们定义:

$$\bar{x}(\lambda) = \frac{x(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)}, \quad \bar{y}(\lambda) = \frac{y(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)},$$

$$\bar{z}(\lambda) = \frac{z(\lambda)}{x(\lambda)+y(\lambda)+z(\lambda)}$$

显然有 $\bar{z}(\lambda)=1-\bar{x}(\lambda)-\bar{y}(\lambda)$。图 11.9 展示了参数曲线 $\mathbf{s}(\lambda)=(\bar{x}(\lambda),\bar{y}(\lambda),\bar{z}(\lambda))$,它完全位于 xyz 空间的第一卦限内。

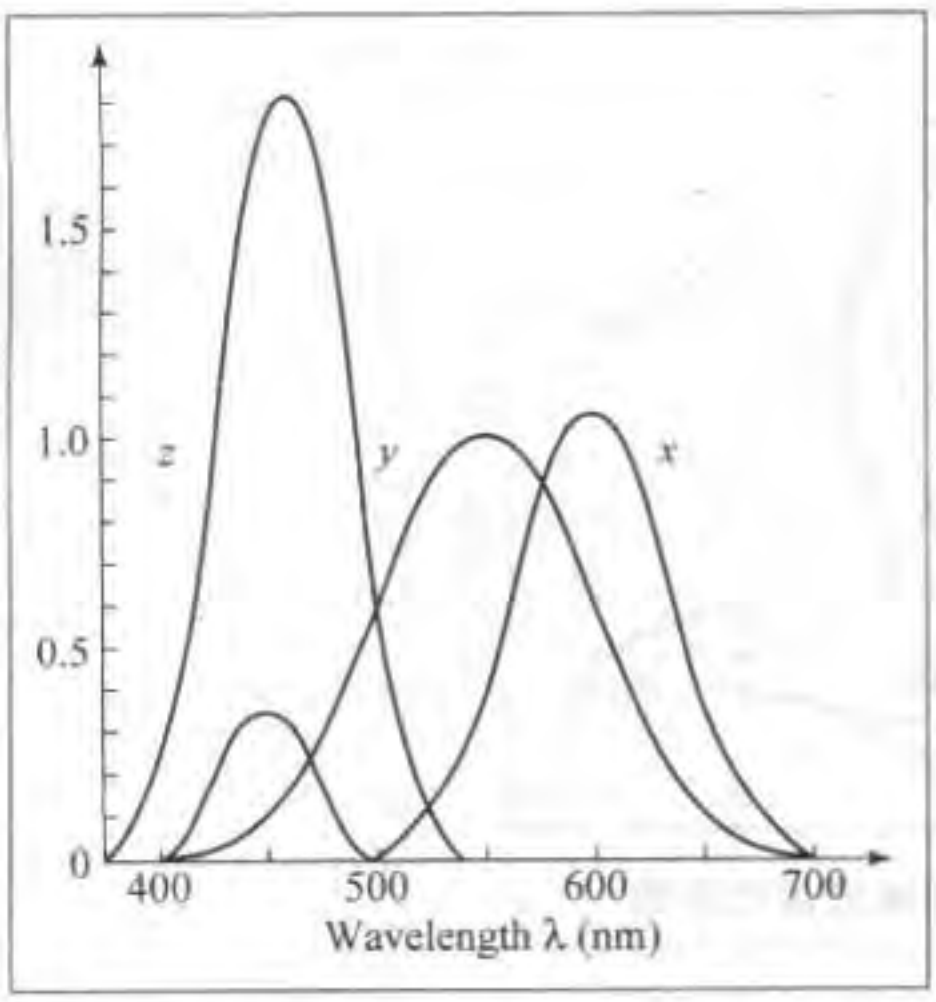


图 11.8 X、Y、Z 三原色的颜色匹配函数

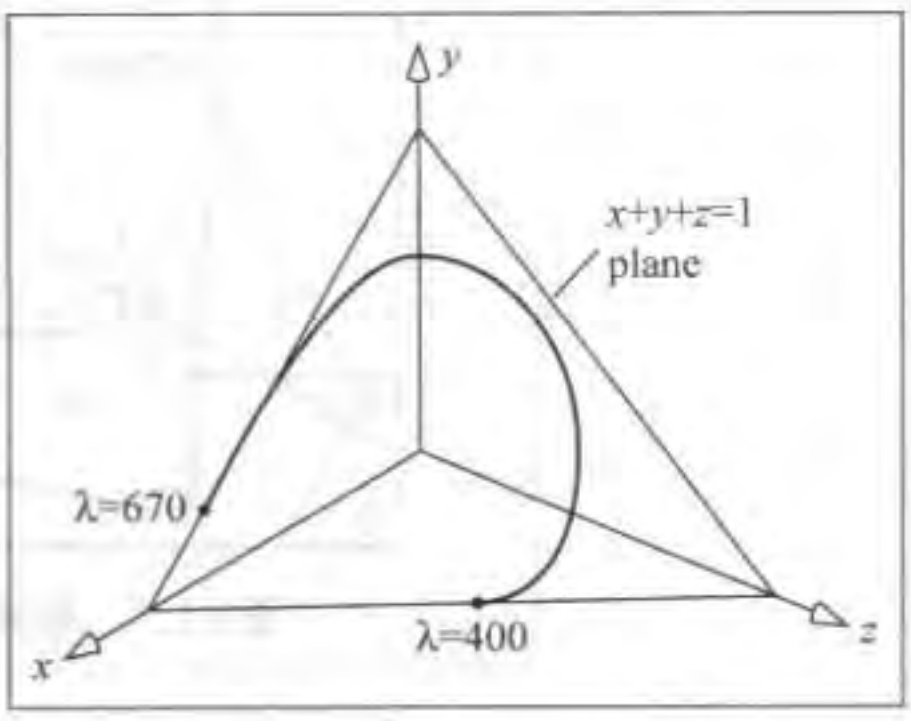


图 11.9 构造 CIE 标准

这三种原色的特定性质及该仿射变换的计算细节,可以从各种参考书中找到,如【Billmeyer81】,【Conrac85】。但是即使不知道这些,也可以理解和使用 CIE 标准。

11.3.1 构造 CIE 色度图

虽然光谱颜色曲线 $s(\lambda)$ 是三维的，但是它位于平面 $x+y+z=1$ 内，所以可以很容易用二维图形表示它的形状，而且可以打印出来用于查阅。描述一种（单位亮度）颜色只需要指定 x 和 y 即可，因为给定 (x,y) 可以很容易的计算 z（如何计算？）。

所谓标准 **CIE 色度图**，就是图 11.10 中所示的曲线 $s'(\lambda)=(\bar{x}(\lambda),\bar{y}(\lambda))$。

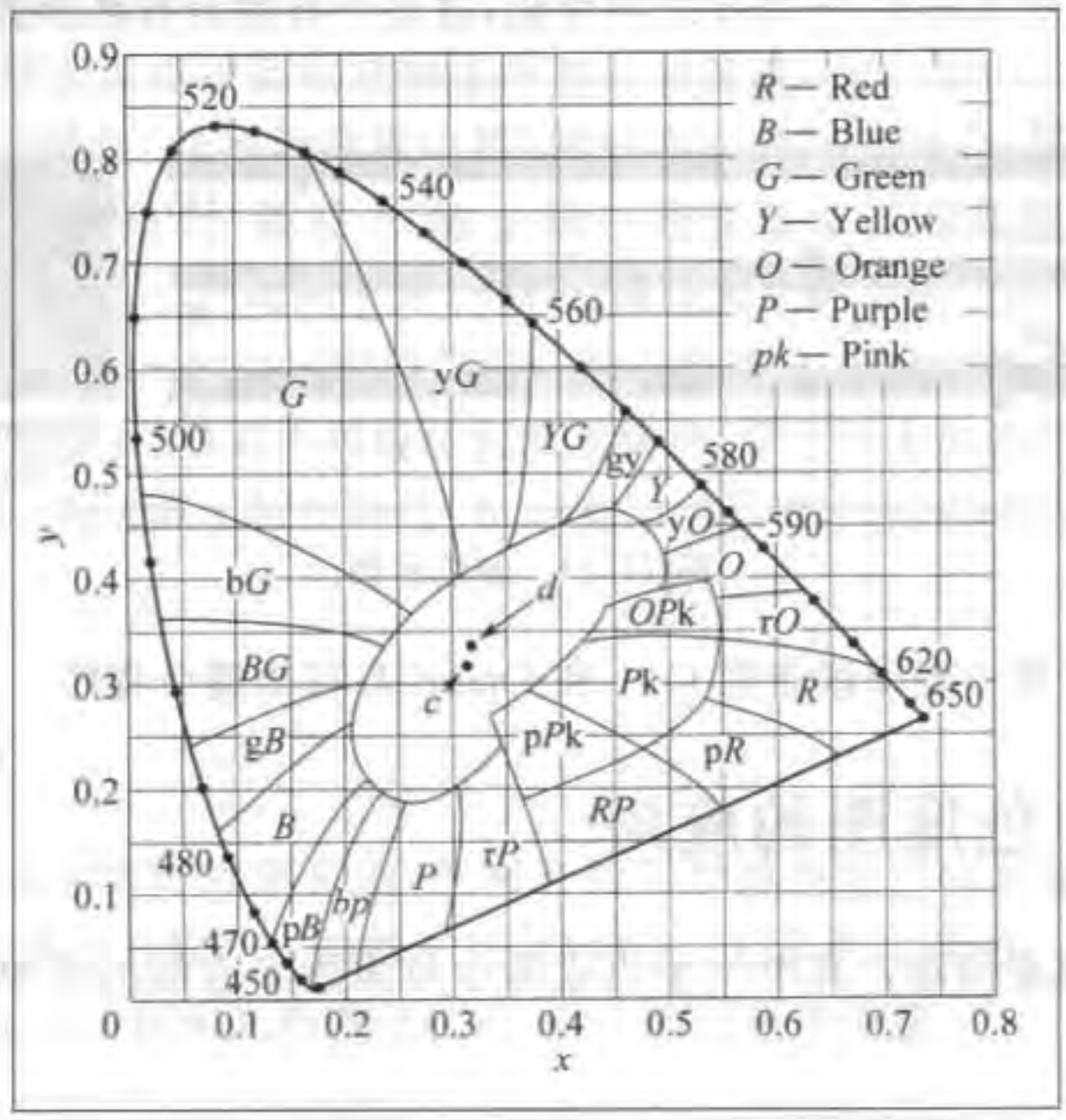


图 11.10 CIE 标准色度图

（想象沿 z 轴方向查看图 11.9 中的三维曲线的正交投影）这个图展示了纯光谱色的马蹄形轨迹，其中标签为相应颜色的波长。马蹄形内部是所有其他可见色，马蹄形以外区域不对应任何可见色。

图中各个区域分别用常用的颜色描述词汇标记。例如（0.6，0.3）点附件看起来是红色。遗憾的是，图中相同的距离并不对应相同的视觉色差。例如，区域 G 内的微小位置变化只会引起微小的视觉色差。相反，区域 B 和 Y 内很小的位置变化就会导致很大的视觉色差。

CIE 色度图定义了一些特殊点。位于 $(x,y)=(0.310,0.316)$ 的 c 点是被称为 **C 光源**的白色，它被认为是“完全不饱和”色。它经常被作为基准白色用于校准某些图形显示器。C 光源的颜色类似于正午多云的天空。位于（0.313，0.329）的 d 点是理想黑体加热到“白热”温度，即 6504K 时发射出的光。这是比 C 光源稍微带一点绿的颜色。许多其他颜色，如白炽灯的钨丝、月光、红热的铁等，在特定温度下发射出的光都经过了准确度量（【Conrac85】和【Rogers85】）。

CIE 色度图的重要价值在于提供了一个描述颜色的世界标准。人们设计了专门仪器用

来生成马蹄形内部任意点(x,y,z)表示的颜色,因而还可以通过仔细匹配来度量任意颜色。人们还设计出设备用来给指定颜色样本自动计算(x,y,z)。接下来可以看到,这个图还可用来对颜色进行一些重要计算。

DIY:颜色控制测试

接下来我们介绍一个程序,可以用鼠标直接控制颜色的 r、g、b 值,同时可以看到最后的颜色。图 11.11 展示了这个程序使用的颜色表,同时还有三个滑块用于调节 r、g 和 b 的值。缓慢调节每个滑块,使其值在 0～255 之间变化,这是个有趣而有启示意义的过程。

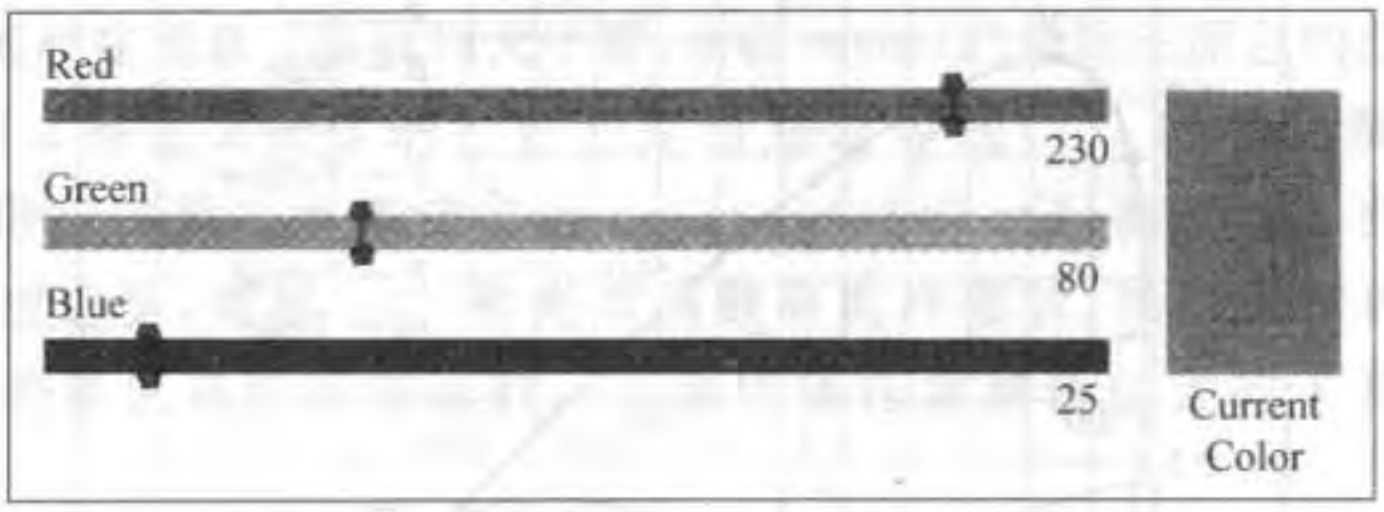


图 11.11 颜色示例

见练习 11.3.1,其中要求读者用 C++ 和 OpenGL 写出整个程序。

11.3.2 CIE 色度图的应用

CIE 色度图有很多用途。其中一些用途源自对该图中直线的简单解释,就像图 11.12 中所提示的。

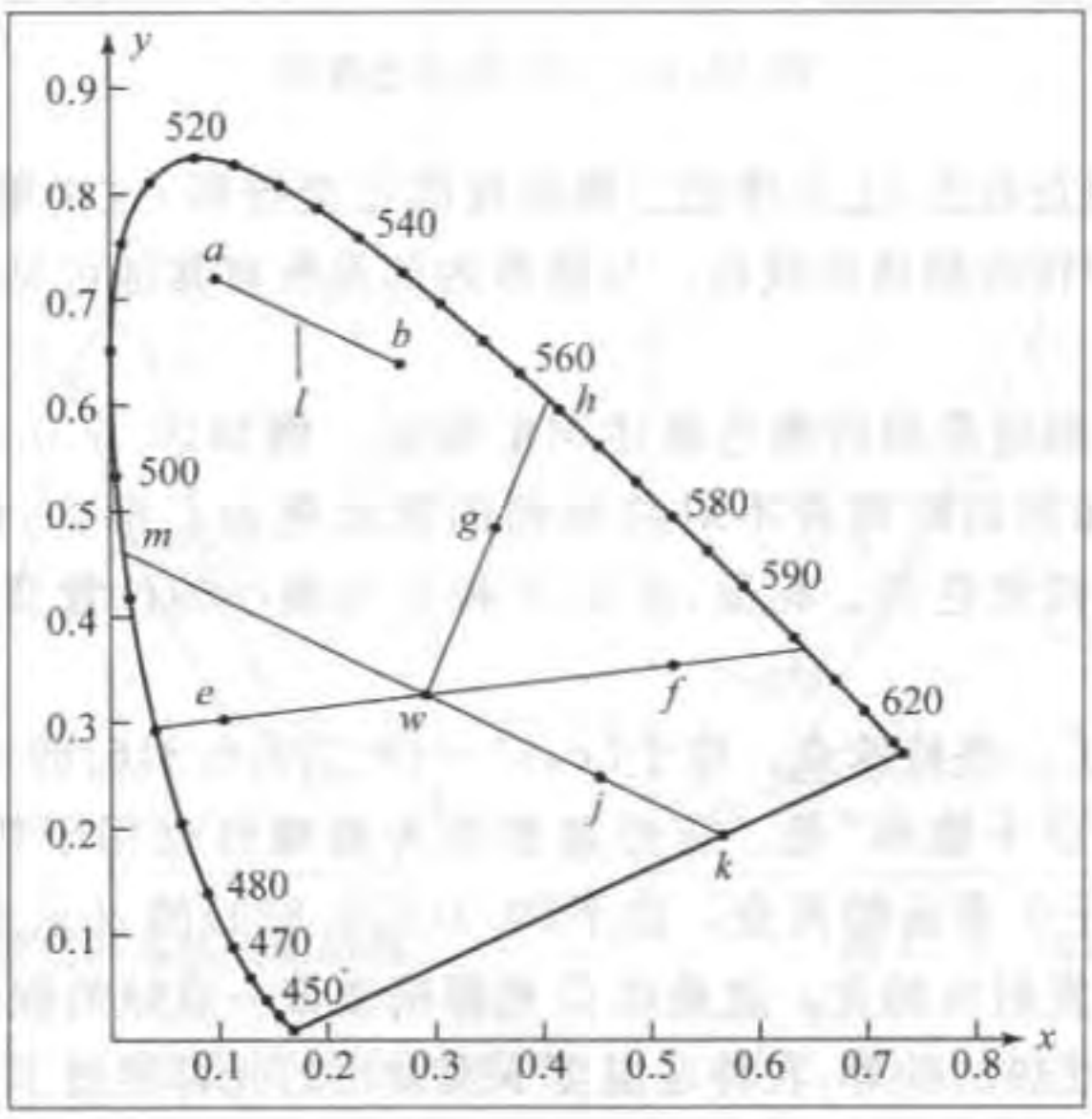


图 11.12 CIE 色度图的应用

考虑两个颜色 a 和 b 之间的线段 l。l 上任意点都是 a 和 b 的凸组合(回顾第 4 章),其形式为 $(\alpha)a+(1-\alpha)b$,其中 $0\leqslant\alpha\leqslant1$。每个点都代表合法的颜色(其 x、y 和 z 分量值和为 1,为什么?),因此可以断言,线段上任意点(并且只有这些点)的颜色可以通过在屏幕上叠加一定量的颜色 a 和 b 来产生。

当两种颜色相加,和为白色时,我们说这两个颜色是互补色(和白色的选择有关)。所以在图 11.12 中 e(蓝-绿)和 f(桔-粉红)关于 w 互补,因为一定量的这些颜色相加产生白色 w。图 11.13 列出了一些常见的互补色对,后面将进一步讨论。

红色—青色
绿色—洋色
蓝色—黄色

图 11.13　互补色对

CIE 色度图也可用于度量给定颜色的主导波长和纯度,例如图中的 g。g 一定是某纯光谱色(位于马蹄形边界)和标准白的线性组合。为了确定具体是哪种光谱色,只要从 w 向 g 画一条直线得到和马蹄形的交点(这里是 h 点),然后测量 h 处的波长——此例中为 564nm,微带黄色的绿色。类似地,饱和度或称纯度即为两个距离的比值:gw/hw。j 点的颜色没有主导波长,因为延长线段 wj 和马蹄形的交点 k 位于所谓的紫色线上,而它不对应任何纯光谱色(紫色线上的颜色是由红色和紫色组成的)。在这种情况下,先计算 j 的互补色(在 m 处),然后用互补色的波长加后缀 c 表示主导波长:498_c。

11.3.3　色域

CIE 色度图在定义色域时特别有用,色域是指某个设备能产生的颜色的范围。例如,CRT 显示器只能显示其荧光粉产生的基本红、绿和蓝三原色的组合色。图 11.14 显示了一种常见的彩色 CRT 显示器的荧光粉颜色红色、绿色和蓝色的位置。

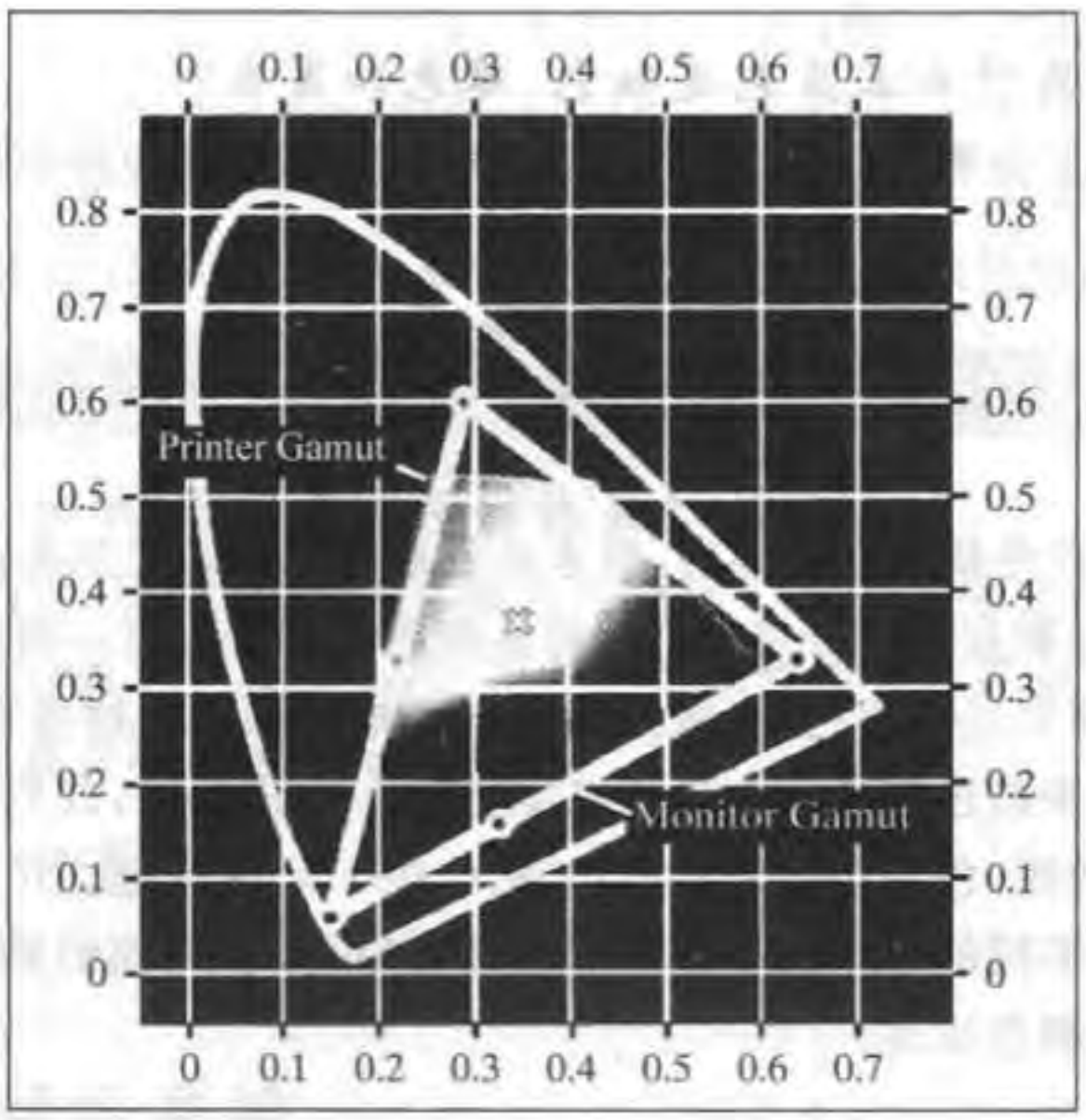


图 11.14　普通 CRT 显示器的色域

图 11.15 显示了某个 CRT 显示器的三原色的具体位置(见【Stone88】)。

原色	x	y
红	0.628	0.330
绿	0.258	0.590
蓝	0.1507	0.060

图 11.5 普通 CRT 显示器的 CIE 坐标

由图可见,这三个点定义了一个三角形区域。三角形内部任何点的颜色是这三原色的凸线性组合,所以都能显示[①]。三角形以外的颜色不属于此显示器的色域,所以无法显示。白色属于此色域,反映了一个众所周知的事实,就是适当数量红、绿、蓝相加产生白色。

同时显示的还有一个彩色打印机的色域。由于颜色在白纸上的形成机制,颜色不是直接的加法系统(见 11.4.2 节),所以其色域不是简单的三角形。这个色域比 CRT 显示器的要小,因此有些显示器能显示的颜色,这个打印机不能产生。另一方面,打印机的色域包含显示器的色域之外的一些点,因此有些颜色打印机可以产生,CRT 显示器却不能。

注意到无论如何选择三原色,即使选择为马蹄形边界上的点,三角形色域总是不能覆盖所有可见色,因为马蹄形总是会凸出到它内部任何三点构成的三角形之外。红色、绿色和蓝色是三原色很自然的选择,因为它们在 CIE 图中距离比较远,从而其色域可以覆盖色度图的大部分面积。相反,比如取黄色、青色和洋红色作为三原色,色域就会小得多(还能产生白色吗?)。

练习

练习 11.3.1 控制颜色表

以图 11.11 为指导,用 C++ 和 OpenGL 开发一个程序,允许用户调节样本颜色的 r、g 和 b 三分量的值。

练习 11.3.2 为什么三原色是红色、绿色和蓝色?

从物理学和哲学上说明为什么人眼中的视锥细胞对红、绿和蓝色最敏感。

11.4 颜色空间

CIE 对颜色的描述是精确而标准的,但未必是最自然的。尤其是在计算机图形学中,最自然的方法是对红、绿和蓝色进行组合得到目标颜色。有些人更习惯于用色调、饱和度和亮度来描述颜色,而画家则经常用色泽,浓淡和色调来描述颜色。这些都是**颜色模型**的例子,即选择 3 个"描述符"来描述颜色。如果可以度量这 3 个描述符,我们就可以用三元数来描述颜色,例如(色泽,浓淡,色调)=(0.125,1.68,0.045)。这就建立了一个用于描述颜色的三维坐标系统。选择不同的坐标架导致不同的**颜色空间**,因此我们需要算法实现在不同颜色空间之间互相转换颜色描述。

---

① 美国通信标准委员会(NTSC)定义了 CIE 坐标下的标准三原色:红色=(0.670,0.330,绿色=(0.210,0.710),蓝色=(0.140,0.080)【Rogers85】。

11.4.1 RGB 颜色空间

RGB(红、绿、蓝的简称)颜色模型描述颜色为适当定义的红、绿和蓝三原色的凸线性组合,如等式(11.3)所示。若限制 r、g 和 b 三个数值都在 0 和 1 之间变化,则所有可定义的颜色位于图 11.16 所示的立方体内部。

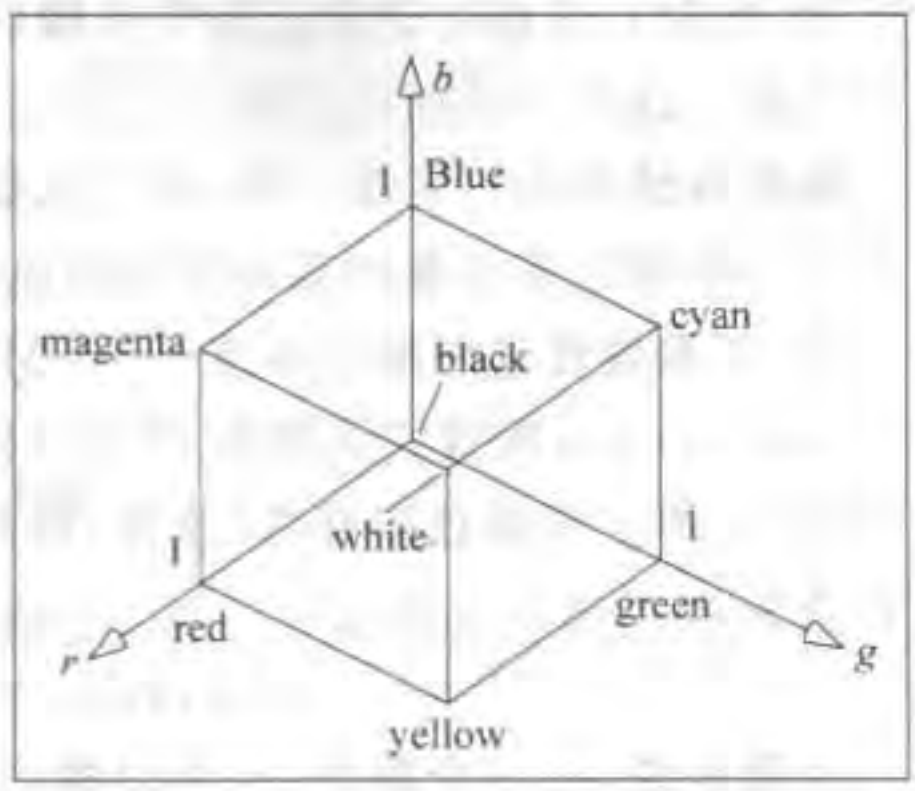


图 11.16 RGB 颜色立方体

和 CIE 色度图不同,RGB 模型不对颜色亮度进行归一化。靠近(0,0,0)的点较暗,而远离它的点较亮。例如(1,1,1)对应纯白色。对计算机图形学而言,这个颜色空间是最自然的,因为颜色描述如(0.3,0.56,0.9)可以直接翻译为颜色查找表(LUT)中的值。注意到标记洋红色的角落表示红色加蓝色可以得到洋红色,标记黄色和青色的角落可以同样解释。在体对角线上的颜色是互补色。

当归一化至单位亮度时,这里定义的所有颜色在 CIE 色度图上都有对应位置,只要提供合适的红、绿和蓝色分量值。(这个色域是什么形状?)不难将 CIE 坐标(x,y,z)描述的颜色转化至(r,g,b)空间,反之亦然。由于 R、G、B 三原色都是 CIE 标准中 X、Y、Z 三原色的线性组合,所以线性变换即可完成上述任务。这个变换由 R、G、B 三原色及白色的定义确定。对于图 11.14 中给出的三原色和图 11.10 中 d 点指定的白色,要求的线性变换如下(详见【Rogers85】):

$$(r,g,b)=(x,y,z)\begin{pmatrix} 2.739 & -1.110 & 0.138 \\ -1.145 & 2.029 & -0.333 \\ -0.424 & 0.033 & 1.105 \end{pmatrix} \tag{11.6}$$

显然,用此矩阵的逆,即可从 RGB 颜色空间变换到 XYZ 颜色空间。

练习

练习 11.4.1 从 RGB 颜色空间转化至 CIE 空间

计算前述矩阵的逆,从而得到 RGB 颜色空间到 CIE 空间的变换。CIE 空间的一个重要性质是所有颜色可以表示为 X、Y、Z 三原色的凸线性组合。这会导致逆矩阵有何性质?你计算得到的逆矩阵是否满足这个条件?

练习 11.4.2 计算分量

推导计算等式(11.6)中矩阵的各个元素并计算其逆。

11.4.2 加/减色系统

到目前为止,我们讨论的都是由有色光各分量的叠加来得到新颜色,这是一个加色过

程。**加色系统**用一定量的三原色的和,通常为红、绿和蓝,来表示一种颜色 D:$D=(r,g,b)$。加色系统可以用任何三种颜色作为三原色,但是由于红、绿、蓝在 CIE 图上相距较远,它们提供了较大的色域。

当用颜色消除法表示颜色更自然时,我们用减色系统来描述颜色。当光线在物体表面上反射(散射),或在半透明介质中传输时(例如使用摄影滤光器时),有些颜色就会被介质吸收而去除。这是一个减色过程。

减色系统和加色系统一样,用三元数表示一种颜色 D,但是 3 个数中每一个指定需要从白色中去除相应多少量的某种特定颜色(相应原色的互补色),从而生成颜色 D。为了说明这一点,考虑最普通的减色系统——CMY 系统,它使用减法三原色:青、洋红和黄色。如果 $D=(c,m,y)_{CMY}$,则我们说颜色 D 可以由白色去除 c 份青色的补色(即红色),m 份洋红的补色(绿色),和 y 份黄色的补色(蓝色)得到。因此,可以立即得到下述 RGB 系统和 CMY 系统的关系:

$$(r,g,b)_{RGB}=(1,1,1)-(c,m,y)_{CMY} \tag{11.7}$$

也就是说,在一个颜色中,可以通过增加 y 来减少蓝色分量 b,因为 y 指定了从白色中去除黄色的补色的数量。

图 11.17 举例说明了减色过程。三个玻璃片在 CMY 系统中被描述为 $(0.4,0.5,0.2)_{CMY}$。当 RGB 加色系统中的白色光 $(1,1,1)_{RGB}$,透过青色玻璃片后,40%红色成分被吸收,产生了青色光 $(0.6,1,1)_{RGB}$。当光又透过洋红色玻璃片后,50%绿色成分被吸收,生成颜色 $(0.6,0.5,1)_{RGB}$。(这是什么颜色?)最后,当这个光透过黄色玻璃片后,20%蓝色成分被吸收,产生颜色 $(0.6,0.5,0.8)_{RGB}$。

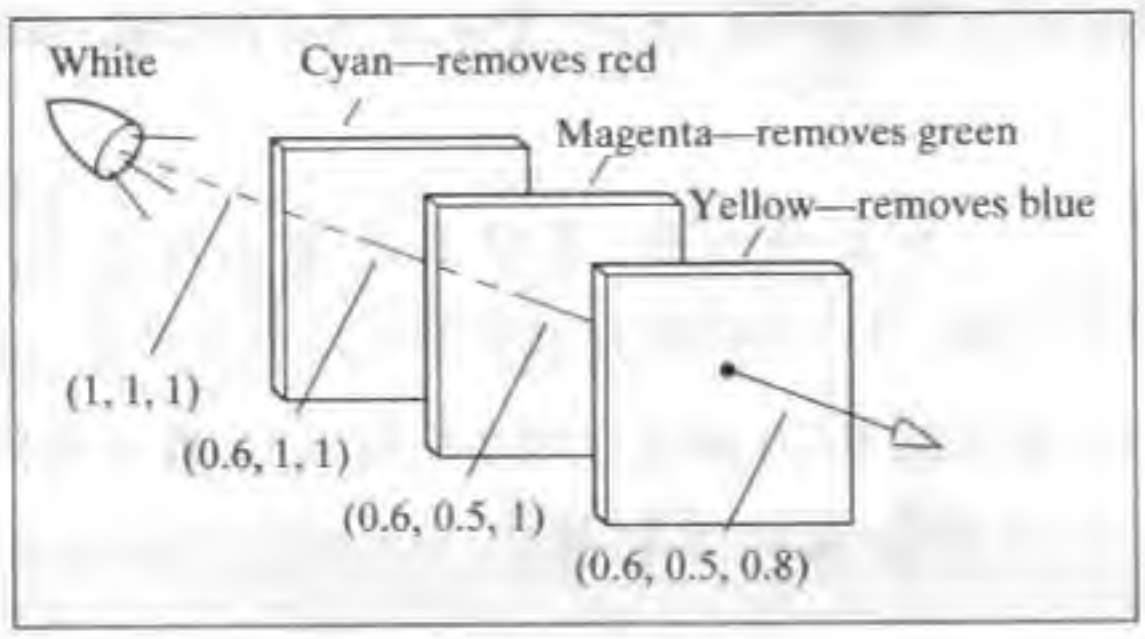


图 11.17 减色过程

现在有色表面的散射可以类似地描述。例如在三色打印过程中,青、洋红和黄色颜料和透明颜料混合在一起。添加任一种颜色都会导致相应的互补色减少。例如,若洋红色颗粒和无色颜料混合,则此颗粒会去除白光中的绿色成分而只反射红色和蓝色成分。通过 CMY 三原色的混合,此减色系统在颜色硬拷贝设备中用来产生目标颜色。

图 11.18 显示了加/减色系统的三原色及其相互作用。图 11.18(a)中,红、绿和蓝色光线共同照射到白色表面上。当这些光线重叠时,它们就会组合形成新的颜色。例如,红色和绿色相加形成黄色。当三个都重叠时,白色就形成了。

图 11.18(b)展示了相反的情况。每个圆是通过加入图示颜色的墨水形成的。在白光下观察此图,有个圆盘呈现黄色,因为其颜料从白色中减去了其补色(蓝色)。当黄色和青色

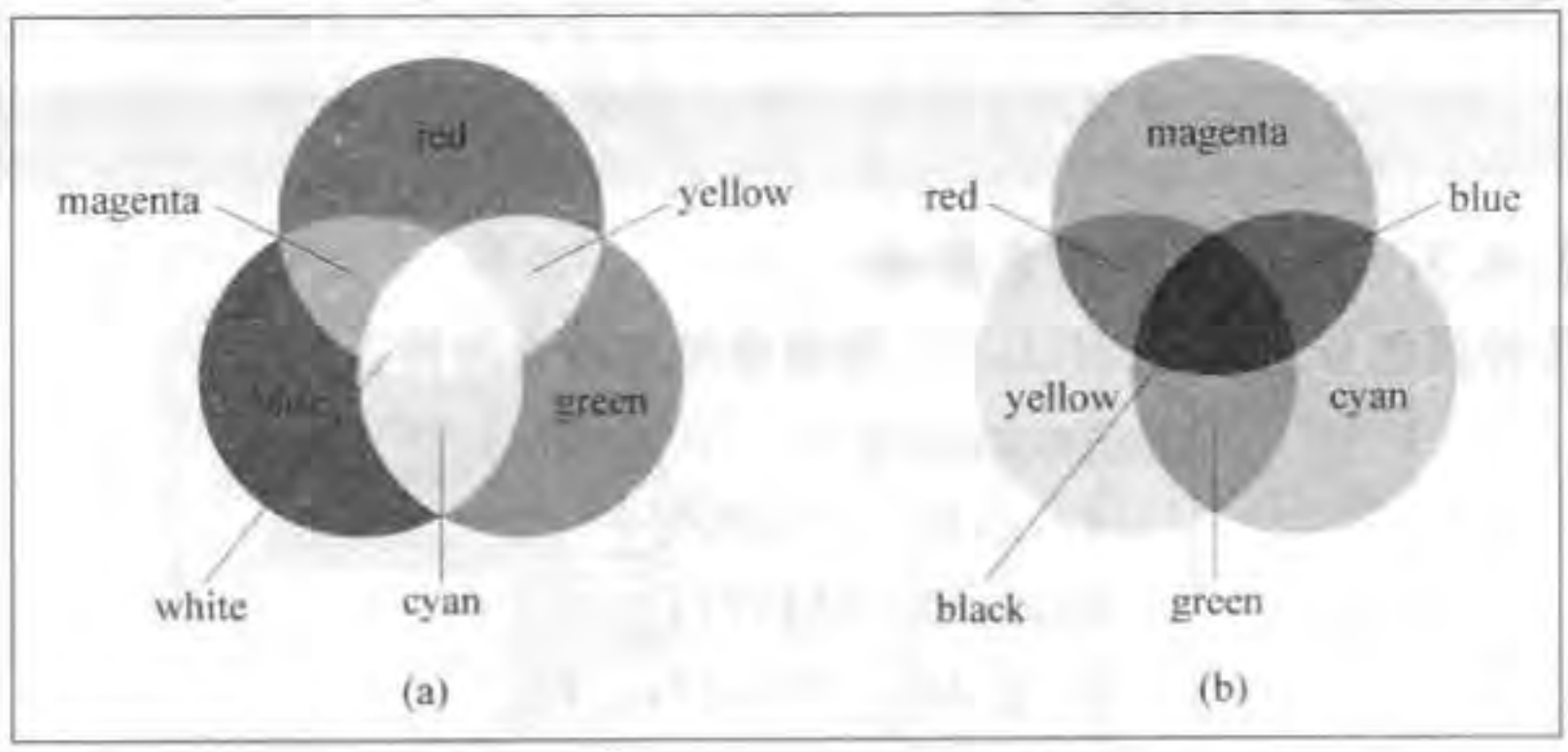


图 11.18 加/减色系统

颜料混合时，我们看到了绿色，因为蓝色和红色成分都已去除。中心部分为黑色是因为所有颜色成分都已去除。

11.4.3 HLS 颜色模型

一个更自然的颜色模型用色调(H)、亮度(L)和饱和度(S)来描述颜色，因为这些量是人眼容易感知和区分的。此模型由将 RGB 立方体扭曲为底部相连的两个六棱锥得到，如图 11.19 所示。

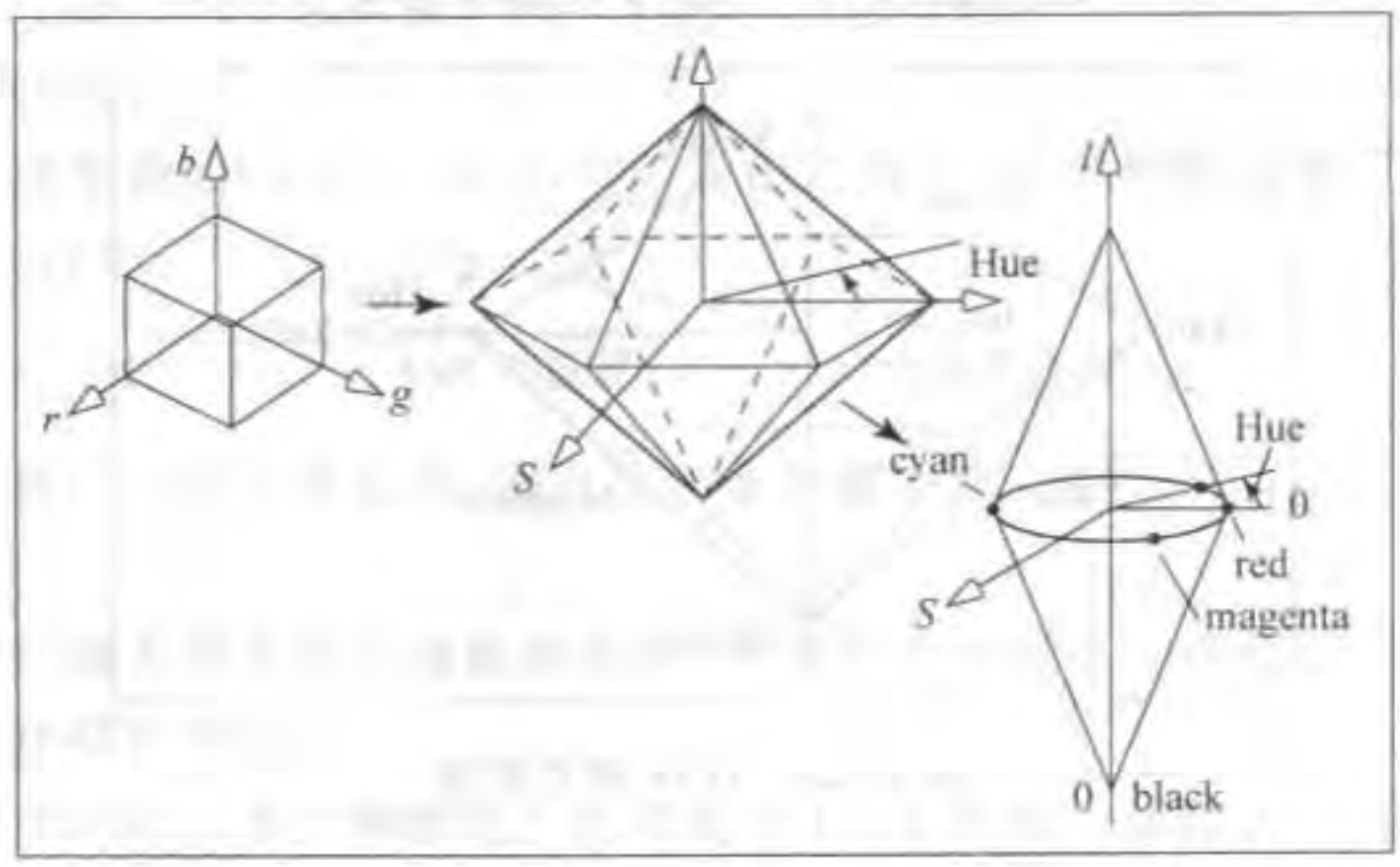


图 11.19 把 RGB 系统变形为 HLS 系统

沿从(1,1,1)到(0,0,0)的体对角线看 RGB 正方体，我们看到六个主要色调(红、绿、蓝及其互补色)分别位于六边形的顶点。故色调可以和 0～360 的角度关联。习惯上 0 度对应红色。当 RGB 三个分量都在 0 和 1 之间变化时，亮度相应在 0 和 1 之间变化。这刚好和 RGB 立方体中白色到黑色的体对角线相对应。一种颜色的饱和度，大致上为其到 RGB 立方体的体对角线的距离，映射到 HLS 锥中为其到亮度中轴的径向距离

HLS 颜色空间是基于 RGB 空间的变形，这引导我们寻找一种算法把颜色从一个系统转化至另一个系统。这里的变形非常复杂，因此我们不必坚持精确的几何变换。把颜色从 RGB 空间转换至 HLS 空间的算法最关心对 R、G、B 分量的最大值和最小值的转换。该算

法提供了一个有用且可逆的转换工具。

练习

练习 11.4.3 完成一些颜色转换

对下列几种颜色分别计算(H,L,S),并解释结果的合理性:

$$(r,g,b)=(0.2,0.8,0.1)$$
$$(r,g,b)=(0,0,0.8)$$
$$(r,g,b)=(1,1,1)$$
$$(r,g,b)=(0,0.7,0.7)$$

练习 11.4.4 HLS-to-RGB 转换

写一个算法 HLS-to-RGB(),将颜色从 HLS 空间转换至 RGB 空间。要求提供前述 RGB-to-HLS 变换的反变换:对 RGB-to-HLS 变换得到的 H、L、S 值应用此变换能恢复原来的 R、G、B 值。

练习 11.4.5 HSV 颜色模型

另一种颜色模型,色调(H)、饱和度(S)和值(V)系统也是基于对 RGB 立方体的变形,但它是单锥体,而非双锥体,如图 11.20 所示。和 HLS 系统一样,在此模型中,色调也被映射为角度,六边形变形为圆,饱和度的含义也是一样的。值 V 记录颜色的亮度,取值范围为 0~1,如图所示。写一个算法把颜色从 RGB 空间转换至 HSV 空间。

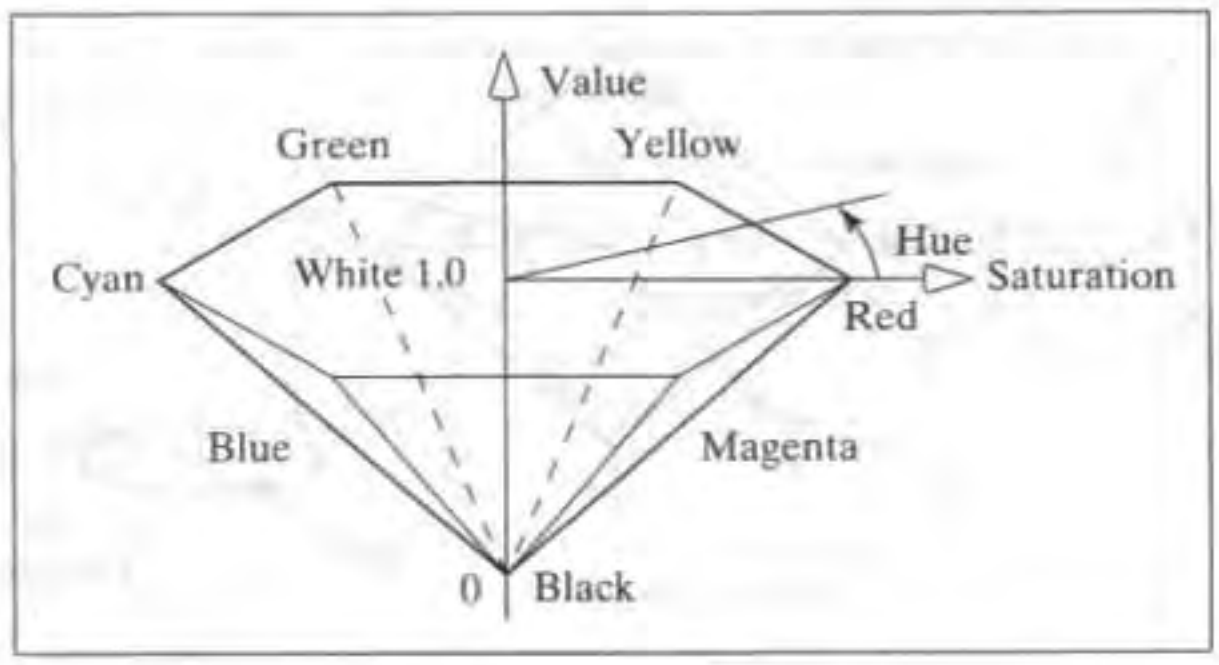


图 11.20 HSV 颜色模型

11.5 索引颜色和颜色查找表

有些图形系统用间接方法关联像素值和颜色。它们使用颜色查找表(简称 LUT),像素值和最后颜色之间的关联可由编程确定。图 11.21 展示了一个简单的例子。这里色深为 6,但是每个像素的 6 位数值在驱动 CRT 之前需要经过一个中间步骤。这些像素值是一个表格的索引,这个表格有 64 个单元格,LUT[0],…,LUT[63]。(为什么正好是 64 项?)例如,若像素值为 39,LUT[39]中存储的值就被用来驱动 DAC(数模转换器),而不是直接用

数值 39 驱动。如图所示，LUT[39]中为 15 位数 01010 11001 10010，其中前 5 位(01010)用来驱动“红色 DAC”，中间 5 位驱动“绿色 DAC”，最后 5 位驱动“蓝色 DAC”。

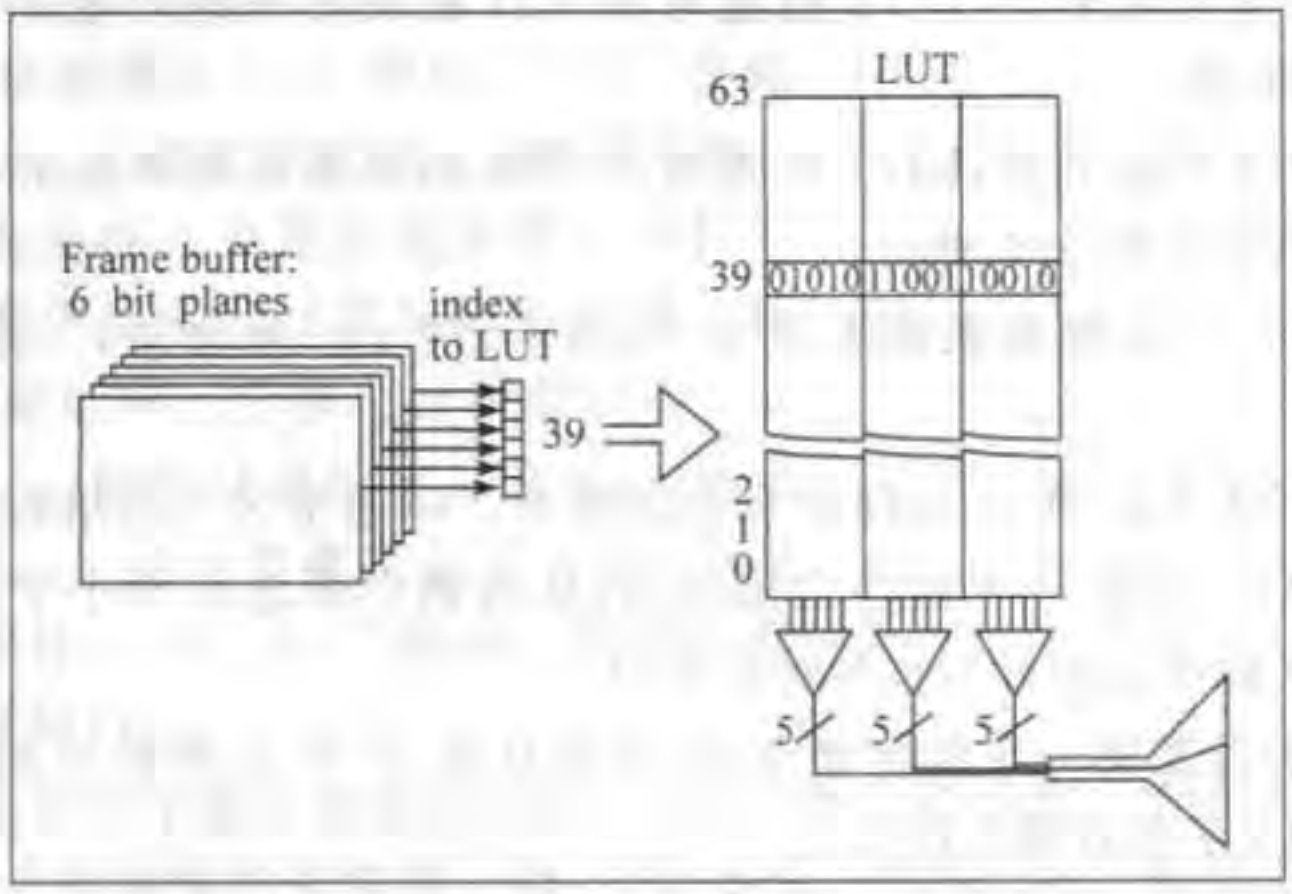


图 11.21 一个采用 LUT 的彩色显示系统

每个 LUT[]单元格可由程序用某些操作系统编程接口如 setPallette()设置。例如，指令：

```
setPalette(39, 17, 25, 4);
```

将设定 LUT[39]为 15 位数值 10001 11001 00100(因为 17、25、4 的二进制数值分别为 10001、11001、00100)。

为了使某个像素例如(x,y)=(479,532)发出此颜色，可以用预先定义的 drawDot()把数值 39 存入帧缓存中：

```
drawDot(479, 532, 39);          //设置(479,532)处像素值为 39
```

每次帧缓存被“扫描”至显示器，读出这个像素值为 39，接着 LUT[39]中的数值被用来驱动 DAC。

上述可编程性极大地便利了颜色的选择，但是自然要牺牲一些代价：程序(或程序员)必须计算出要使用哪种颜色！

这个系统最多能显示多少种颜色？此系统中 LUT 的每一项包含 15 位，故每项可以设为 2^{15}=32K=32 768 种颜色之一。系统可显示的 2^{15} 种颜色集合被称为调色板，故我们称这个显示器“有 32K 色调色板”。

问题是每个像素值在 0～63 范围内，所以 LUT 一次只能存储 64 种不同颜色。因此这个系统一次最多只能显示 64 种不同颜色。这里的“一次”是指描扫完整个帧缓存的过程——类似于 1/60 秒左右。在扫描一幅图像的过程中，LUT 的内容是不能改变的，所以整个扫描过程使用固定的 64 种调色板颜色。虽然程序可以在两次连续扫描的极短间隔内改变 LUT 的内容，但是通常 LUT 的内容在多次扫描过程中都保持不变。

概括地说，假设某光栅显示系统色深为 b(则其帧缓存有 b 个位平面)，每个 LUT 单元格位宽为 w，那么这个系统最多可以显示 2^w 种颜色，一次最多 2^b 种颜色。

例 11.5.1

(1) 一个有 $b=8$ 个位平面,LUT 位宽为 $w=12$ 的图形系统可显示 4096 种颜色,一次可显示其中任何 256 种。

(2) 一个有 $b=8$ 个位平面,LUT 位宽为 $w=24$ 的图形系统可显示 $2^{24}=16\ 777\ 216$ 种颜色,一次可显示其中任何 256 种。

(3) 若 $b=12$,$w=18$,则此系统可显示 $256K=262\ 144$ 种颜色,一次可显示 $2^{12}=4096$ 种颜色。

位平面数 b 和 LUT 位宽 w 之间没有强制关系。w 通常为 3 的倍数,从而每个 DAC 由同样多位驱动($w/3$)。另外,b 永远不会超过 w,从而调色板总是不小于系统一次能显示的颜色数。(为什么永远不会设计 $w<b$ 的系统?)

注意到 LUT 本身需要一定的存储空间,虽然只有 2^b 个 w 字长空间。例如,若 $b=12$,$w=18$,则需要 9 216 字节存储 LUT。

光栅显示系统采用 LUT 的动机是什么?一般程序都要尽量减少内存占用。增大 b 会大量增加帧缓存需要的内存,主要是因为图像中像素太多。大量的内存占用会显著增加整个系统的代价。

11.6 颜色量化

渲染场景时,会产生大量浮点颜色三元数(I_r,I_g,I_b),每个三元数对应图像中一个像素。第 9 章讨论了把这种颜色三元数转化为二进制的方法,从而可以传输至显示器或存档为文件以备后用。一种常用方法是把(三元数的)每个浮点分量转化为一个字节,从而图像就可以用"真彩色"——24bit/pixel(位每像素)来表示。扫描仪将照片数字化时,通常也生成真彩色图像文件。

但是有两个重要问题必须解决:

- 有些设备(如 PDA 或 GPS 系统)不能处理 24 位颜色。虽然不断下降的内存价格使得普通工作站的显示器有足够的色彩分辨率,但还是有许多设备色彩分辨率有限。它们可能只能处理 5 位的红、绿或蓝色,或者它们使用只有 256 项的颜色查找表(例如飞机场或超市的自助服务机)。
- 真彩图像可能很大,需要很大的磁盘空间和很长的网络传输时间

因此我们需要一些方法,减小彩色图像大小并减少其中包含的颜色数。例如我们需要这样一种方法,它能选择真彩图像中"最好"的 256 色,并对任何其他颜色,从这 256 色中找到"最接近的"一个代替。此过程通常称为**颜色量化**。

问题描述

给定 N 个颜色三元数,找到能最好地代表初始(N 种)颜色的 K(K 一般比 N 小得多)种颜色。对每种初始颜色,找到其"最接近"的替代色。有需要的话,将原始颜色替换为相应的最佳替代色的索引 $0,1,\cdots,K-1$。

这里有许多问题要解决，例如“最好”和“最接近”是什么意思？另外，由于 N 如此之大，任何好算法都必须足够高效。

我们接下来讨论四种减少图像颜色数的方法。在每种方法中，假设图像用 RGB 颜色三元数[①]存储于文件中——浮点三元数或者整数三元数都可以。

很多时候会先进行一个预处理步骤。即扫描图像文件确定图像中像素三元数的取值范围。特别地，此扫描过程要计算三个颜色分量的极值：红色分量的 r_{min} 和 r_{max}，绿色分量的 g_{min} 和 g_{max}，及蓝色分量的 b_{min} 和 b_{max}。这些数值定义了 RGB 空间中的一个平行六面体，如图 11.22 所示。则所有颜色全体就位于此区域内。

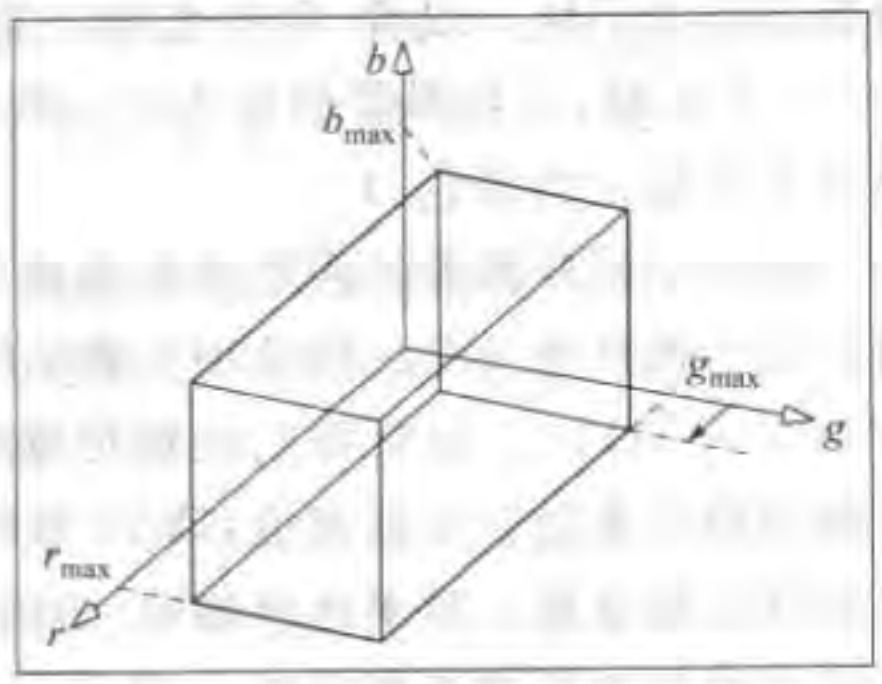


图 11.22　位于一个颜色块内的颜色全体

设想有大量点散布于此六面体的不同区域内，每个点代表一种颜色。有些颜色可能会重复很多次，有些可能紧紧的聚集在一起，可能还有一些孤立点，在这个六面体的一个很大区域内都没有相邻点。这就导致了很多问题：如果许多点聚集在很小的区域内，我们是否应该用该区域中心替代区域内所有点？如果有一种颜色重复出现多次，但是没有比较近的相邻点，我们是否该用高精度来表示它？

11.6.1　均匀量化

最简单的方法是沿坐标轴把颜色块分成一些互不重叠的子块，并选择每个子块接近中心的颜色值作为该子块的代表色。

那么如何细分颜色块？这里我们介绍一个最简单的方法。如图 11.23 所示，分别沿三个颜色轴将颜色块切片。这样就可以独立的选择代表色的红、绿和蓝色分量。例如，若

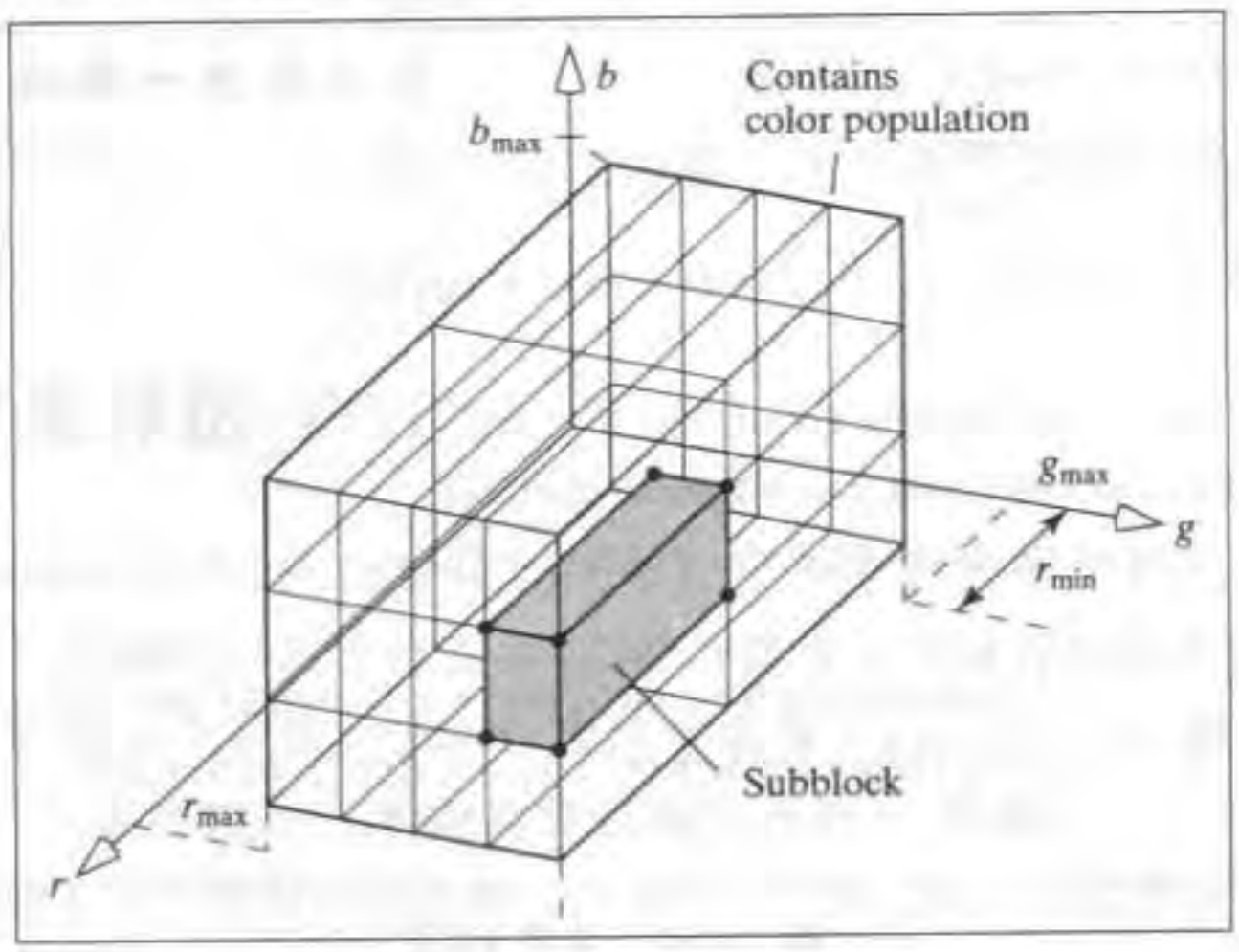


图 11.23　切分颜色块

① 为了改进量化结果，也可以先把 RGB 颜色转化至其他颜色系统，如 HSV 或 CIE，然后在新系统中进行量化。

$K=256$,则红色范围 $r_{min},\cdots,r_{max}$ 可分成 8 片,绿色也分为 8 片,而蓝色分为 4 片,从而正好得到 256 个子块。或者,我们也可以把红色和蓝色都分成 6 片,而绿色分成 7 片,从而产生 252 个子区域。(如果要使用 LUT,则有 4 个条目可以留空不用,或者它们可以用来存储边界或文本标记的颜色。)

还有个同样重要的问题是如何选择代表色,它们的数值必须可以用 LUT 记录。例如假设每个条目有 4 个比特位用于表示绿色分量,从而允许绿色的强度范围为 0～15(二进制为 0000～1111)。如果我们决定把绿色分成 6 块,那么该从 0～15 之间选择哪 6 个数值?注意到对大多数显示器来说,第 15 级绿色的实际亮度可以通过调节其前置面板上的控制旋钮来独立地设置。对于这种终端,必须对相对强度进行编码。下面的例子将帮助说明不同因素之间是如何相互影响的。

例 11.6.1

考虑一种典型情况:LUT 有 $K=256$ 个条目,每个条目分别用 4 位表示红色、绿色和蓝色的强度。扫描三元数图像文件后,发现 r_{min}、g_{min}、b_{min} 都很接近 0,而 $r_{max}=4.0$,$g_{max}=7.0$,$b_{max}=3.0$,那么该如何将颜色三元数映射为 LUT 值?

我们决定用 7 个代表描述绿色成分,用 6 个代表描述红色和蓝色成分。为了简单起见,我们在 16 个候选 LUT 值中尽可能均匀的选择代表色。对于红色和蓝色,6 个值分别设为 0、3、6、9、12 和 15 即可满足要求。对于绿色,我们不能达到完全等间隔的分割,但是把值设为 0、3、5、8、11、13 和 15 是个合理选择。

接下来,如何把这些值存储至 LUT 条目?创建一个整数数组 int r[6]存储 6 个红色值,类似地创建数组 g[]和 b[]。以 r[]为例,它保存 6 个数值 0、3、6、9、12 和 15。在下面的源代码中,三重嵌套的循环遍历了这三个数组中颜色的所有可能组合,并对每种组合生成一个 12 位数值并存储于 LUT[]数组:

```
#define numRed 6
#define numGreen 7
#define numBlue 6
#define pack(r,g,b) (256*(r)+16*(g)+(b))

for(rd=0; rd <numRed; rd++)
    for(grn=0; grn <numGreen; grn++)
        for(blu=0; blu <numBlue; blu++)
        {
            index=numRed * numGreen * rd +numBlue * grn +blu;
            LUT[index]=pack(r[rd], g[grn], b[blu]);
        }
```

图 11.24 装载 LUT

宏定义 pack 将红色值乘以 256,即将其左移至 LUT 条目的最高 4 位,对绿色进行类似位移处理。例如,当 rd、grn、blu 分别为 3、2、5 时,对应 LUT 中索引为 133,由 $r[3]=9$,$g[2]=5$ 和 $b[5]=15$ 三个数根据要求组合得到的数为 2399,对应 16 进制数为 $95f$。

当重新扫描图像文件时，三元像素值序列(r_i, g_i, b_i)中每个r_i分别和 6 个红色代表色比较，找出最接近的一个：找到j_r使得r_i离$r[j_r]$最近。例如，若$r_i=8.23$，则最接近的值为 9，故$r_i=8.23$。同样地计算j_g和j_b，然后可以计算索引值为$j=42*j_r+6*j_g+j_b$。

均匀量化方法有其缺陷：不考虑颜色的总体分布，并且扫描原始图像文件时没有统计任何信息（除了r_{max}、g_{max}和b_{max}三个数值）。然而这种方法可适用于某些图像，特别是当每个像素的比特位数较大且 *LUT* 提供的调色板也很大时。

图 11.25 显示的是一幅原始图像，图 11.26 显示了对它进行均匀量化的结果（真彩色版本可从此书主页找到）。注意到大量单色带出现在原图像中一些有颜色梯度的区域（但是变化很平缓）：许多相邻的近似颜色由单一代表色替代，从而导致这些区域颜色单一，缺少平滑过度。

图 11.25　本书作者的原始照片

图 11.26　量化至 256 色后的作者照片

练习

练习 11.6.1　试验一些颜色值

在例 11.5.1 中，试说出 LUT 的前 20 项的内容。这个排列是否有序？LUT[29]中存储的是什么？LUT[231]呢？

11.6.2　流行度算法

流行度算法【Heckbert82】和均匀量化方法相比至少考虑到有些颜色会经常在图像中出现。因此此算法给这些颜色以较大优先级，即使可能很多最流行的颜色会紧凑在一块。

它的基本方法是构造一个表格记录每种颜色在图像中出现的次数，然后对这个表格进行排序。那么，排序后的表格的前 K 种颜色就是最流行的 K 种颜色。

因为表示真彩色的 3 字节整数有很多种可能取值（2^{32}种），所以我们首先将红、绿和蓝色每个值截短为 5 位以简化问题，这样就只剩下 $2^{15}=32K$ 种可能的颜色值。于是分配一个长度为 32K 的数组并将所有元素初始化为 0。扫描图像，每读入一个颜色，就将数组中相应元素增一。读完所有 N 种颜色后，对这个数组进行排序，然后取 K 个最流行颜色为代表色。

接着重新扫描图像，将每个颜色用流行度表格中最接近的代表色$(r,g,b)[i]$替换。通

常采用最小平方距离作为"最接近"的度量：

求 i 使得 $(rd-r[i])^2+(grn-g[i])^2+(blu-b[i])^2$ 达到最小值

这是个代价昂贵的操作。

11.6.3 中值切分算法

中值切分算法由 Heckbert【Heckbert82】首先提出。这个算法把色块分割为 K 个子块，每个子块包含大致相同的颜色数。图 11.27 显示了二维情况下的基本步骤。(假设颜色只有红色和绿色成分。)在图 11.27(a)中，原始颜色立方体被首先沿最长的一维细分：找到中值 r_1 使两个子块分别包含 $N/2$ 种颜色。接着，类似地处理两个子块——在最长一维的中值处切割为两部分。持续这个过程，直到分割成 K 个子块。每个子块的代表色取为其中心。图 11.27(b)显示了一个经过九次切分的例子。

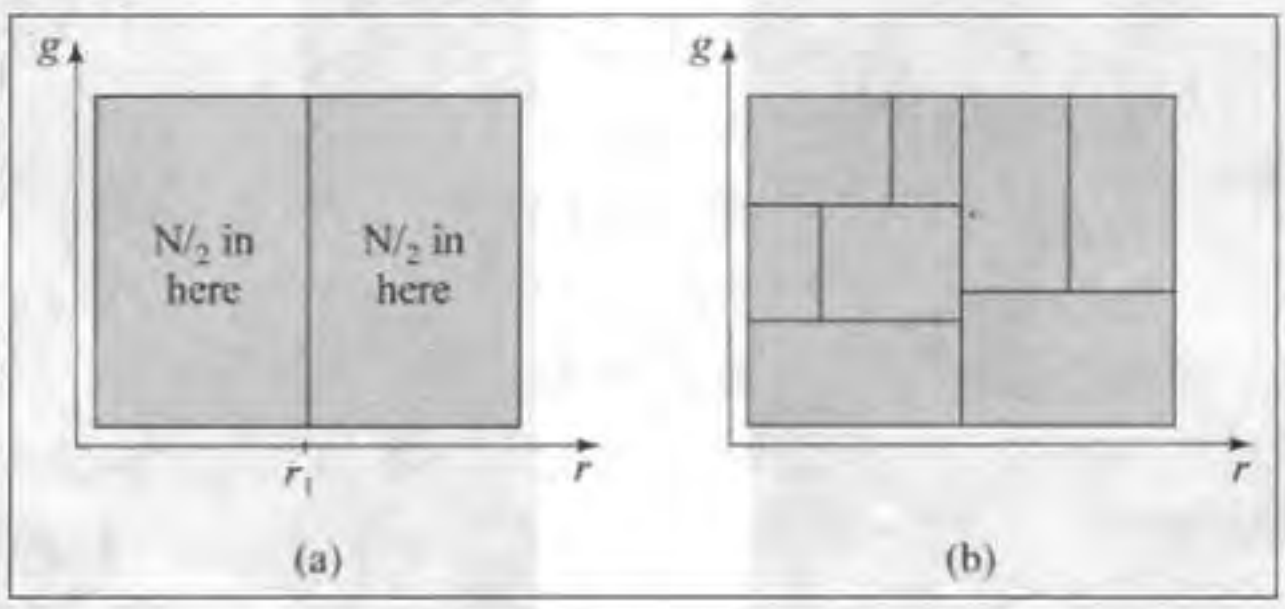


图 11.27 中值切割过程(对于二维颜色)

进行切分过程的一个简单方法是使用子块队列。定义合适的数据结构，存储色块维数及指向其包含的颜色数组的指针。每次出列一个子块，在最长维的中值处把它切分成两个子块，再把两个子块都加入队列等待下次处理。

当我们得到 K 个子块后，将其分别用 $0,1,\cdots,K-1$ 标记，然后重新读取整个图像，检查每个颜色位于哪个子块。Heckbert【Heckbert82】提出了高效算法实现此过程。

图 11.28 展示了图 11.25 经过中值切割算法处理，并把颜色数减少至 256 时的结果。仍然可以看到一些单色带。

图 11.28 用中值切割算法量化至 256 色后的图像

11.6.4　八叉树量化

Gervautz【Gervautz90】提出了一种基于八叉树(一种树形数据结构,其中每个节点最多有 8 个子节点)的方法,用来有效地将图像颜色数减少至 K 种。事实上此方法读取彩色图像,并建立一个颜色八叉树,准确记录颜色直至 K 种不同颜色被加入。然后,每读入一种新颜色,就将其加入八叉树中,但是紧接着又"简化"八叉树使它再次只包含最多 K 种不同颜色。简化过程强制一些颜色被合并到一起(变为它们的代表色),并用相应的近似颜色代替被合并的颜色。当图像中 N 种颜色全部读入后,八叉树包含 K(或略少)种代表色。然后再次遍历此八叉树,并赋给每个代表色一个索引值。

接着再次遍历图像,每遇到一种颜色,就把它插入八叉树中并很快的找到位置,或者正好位于树的底部(被叶节点精确描述),或者位于其最佳替代色所在的某些中间节点。然后返回此节点索引,说明它应该替换为哪种颜色。

如何利用八叉树结构使上述过程有效执行,并且不占用过多内存,这是个很有意思的问题。图 11.29 展示了八叉树的一部分:每个节点包含加入此树的颜色的一些相关信息,及 8 个指向其子节点的指针。

当一个颜色被插入八叉树时,其存在性通过"增加"一条直到第 8 级的路径来记录其存在。图 11.30展示了一个例子,表中显示了红、绿和蓝色值(二进制)。检查(r,g,b)三元数的每一位,分别从 R、G 和 B 的最高位开始,这里最高位分别为 101,它代表二进制的 5,所以路径指向 5 号子节点。若 5 号子节点尚不存在,则创建一个新节点并连接至此位置。继续此过程,处理第二高位三元数(110),路径指向 3 号子节点。经过 5 次迭代之后,检查颜色的最低位(110),路径指向 6 号子节点。如果此处有一叶节点(因为之前图像中已出现过此颜色),则将该颜色的计数(节点的一个域)增一。否则创建一个颜色计数为 1 的叶节点。注意到八叉树的深度不超过 8,插入一个颜色的过程是很快的:只需要不超过 8 次的子节点索引计算。

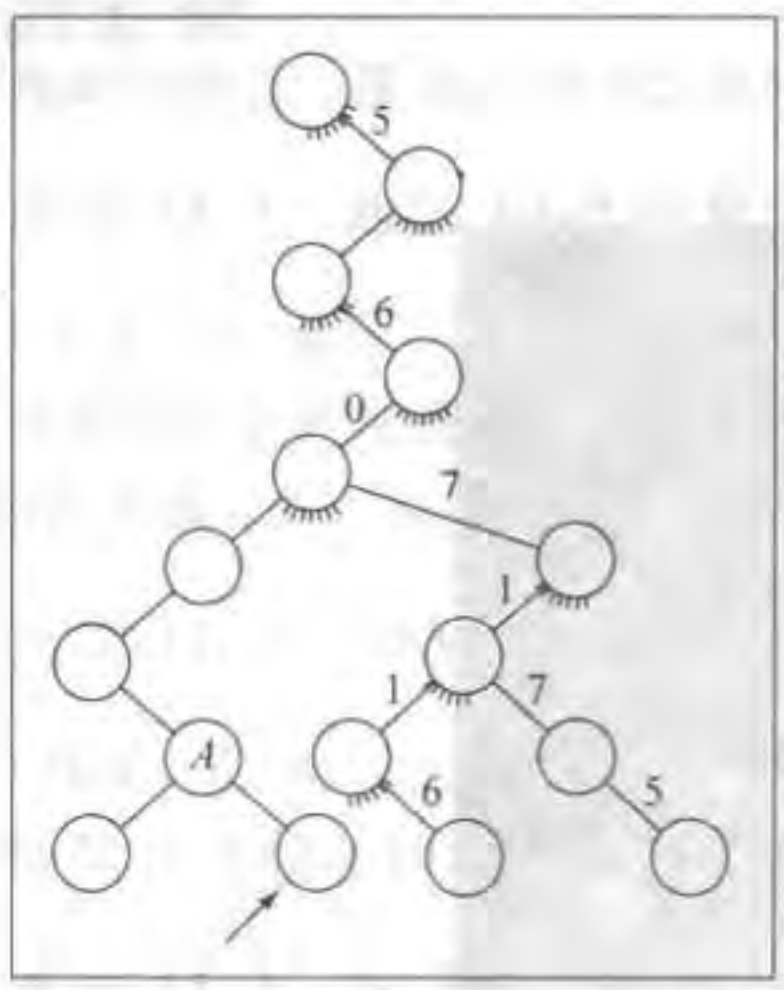


图 11.29　一个包含 4 种颜色的八叉树

R=	1	0	1	0	1	0	0	1
G=	0	1	1	0	1	0	0	1
B=	1	1	0	0	1	1	1	0
child:	5	3	6	0	7	1	1	6

图 11.30　从根节点到颜色叶节点的路径示例

每个深度为 8 的节点精确代表一种颜色(精度为 8 位),而中间节点代表颜色子块。例如,一个 3 级节点表示所有 R、G、B 分量的高三位相同,而其余位为任意值的颜色。

当八叉树包含颜色数超过 K 时,树就要被简化。从尽可能低的深度寻找一个"可简化"节点(至少有 2 个分支)。(图 11.29 中 A 节点就是一个可简化节点。)为了简化八叉树,先删除可简化节点的子节点(释放空间用于存储新节点),并将该简化节点标记为叶节点。此节点已经包含其子节点中出现颜色的汇总信息。每次八叉树简化过程减少的颜色种数比删除的子节点数小 1(为什么?)。现在,新的颜色又可以被插入八叉树中了。

如案例 11.7 中所详细描述的,每个节点要跟踪一些信息的变化,包括:

```
long numTimes;                    //这个子块中出现的颜色个数
long rSum, gSum, bSum;            //颜色值的累加和
int isLeaf;                       //1,如果此节点为叶节点
```

每个颜色(R,G,B)被插入八叉树时,更新其路径经过的所有节点的信息:

```
numTimes++;                       //这个子块又多了一种颜色
rSum+=R;                          //红色值总和
gSum+=G;
bSum+=B;
```

因此当简化一个节点时(其子节点被删除),它已经包含关于其子树可能发生任何变化的所有信息。稍后扫描八叉树确定每个叶节点的索引时,叶节点的代表色取为平均颜色:

$$代表色 = \left(\frac{rSum}{numTimes}, \frac{gSum}{numTimes}, \frac{bSum}{numTimes}\right)$$

最后,当再次从图像中读入 N 个颜色时,跟踪每个颜色的路径直至找到叶节点,检索得索引值。由于每条路径不超过 8 个节点,故此过程是很快的。

Gervautz 和 Purgathofer 关于颜色量化提出了一些建议,并估计了每种方法需要的计算资源。他们认为用八叉树方法得到的结果在质量上可以和中值切割方法相媲美,而且前者的处理时间和内存占用要低得多。

图 11.31 是图 11.25 经过八叉树量化,颜色数减少至 256 时的结果。仍然可以看到一些单色带,但是再现的颜色明显要比图 11.26 好得多。

图 11.31 用八叉树量化至 256 色后的图像

本章小结

人眼——大脑系统感知颜色的过程是很复杂的。一种光的颜色可以通过描述其光谱密度函数来精确指定，但是这种方法用于交流颜色是很不方便的。另外，人眼看起来一样的颜色，它们的光谱密度可以是不同的。事实上，我们对颜色的感知是三维的，所以我们需要一种方法来用三元数描述颜色。

CIE 标准提供了一个精确描述颜色的方法。任何具有单位亮度的颜色 S，可由 2 个数值(x,y)确定。这个表示法是由于如标准中定义，可以通过对适量的 X、Y 和 Z 进行相加来产生颜色 S。确切的说，$S=xX+yY+(1-x-y)Z$。其中特殊的超饱和三原色实际上是人眼不可见的，选择它们是为了使任何纯光谱色（单一波长光）都可以用正系数 x 和 y 来组合生成。CIE 色度图提供了一个描述颜色和计算给定颜色的组成成分的世界通用标准。CIE 色度图也用于显示色域，色域可以通过列举三原色的任意正线性组合得到。

有些算法可以把一个颜色系统描述的颜色转换至其他颜色系统，所以我们可以在 CIE 坐标和更熟悉的 RGB 坐标之间互相转换。其他颜色空间，如色调、饱和度和亮度，也经常被使用。

计算一幅彩色图像时（例如根据有光照的三维场景），只有绘制完成后，才知道要绘制的颜色范围，所以边绘制边显示图像是很困难的。一种较好的方法是，绘制时把所有颜色三元数存档至一个文件，然后扫描此文件确定颜色的总体分布。一旦知道颜色的总体分布，就可以在颜色三元数和显示设备可以处理的数值之间建立合适的对应。这其中可能包括要选择适当的颜色并装入 LUT。由于有些设备能显示的颜色数很有限，所以经常需要减少图像中的颜色数。人们提出了几种不同方法用于颜色量化，这些方法在程序实现的复杂度、计算效率、最后产生图像的质量等各方面都有不同改进。

案例分析

案例 11.1　绘制 RGB 颜色空间

（难度：Ⅱ）编写一个程序显示图 11.16 中 RGB 颜色立方体的一个“切片”。用适当的颜色显示切片上所有点。要求用户可以用键盘确定切平面位置（即确定平面 $aR+bG+cB=d$的系数 a、b、c 和 d）。

案例 11.2　HSV-RGB 转换

（难度：Ⅱ）编写一个带 GLUI 菜单的程序，包含第 2 章中的微调输入旋钮，允许用户用鼠标选择 H、S 和 V 的值，然后程序显示特别着色的彩色长方形中相应的 RGB 代表色。

案例 11.3　均匀颜色量化

（难度：Ⅲ）编写一个程序试验均匀颜色量化算法。从文件中读取一幅图像，然后在屏

幕上并排显示图像的 2 个版本：左边为全彩色版本，右边为量化版本。用户确定不同的红、绿和蓝色种数(例如分别为 8、7 和 4 种)，然后程序计算量化后的颜色并重新计算要显示的图像。

案例 11.4 流行度颜色量化

(难度：Ⅲ)编写一个程序试验流行度颜色量化算法。从文件中读取一幅真彩图像，然后用户确定允许使用的不同颜色种数 K。接着程序扫描图像，将 RGB 三元数量化为 15 位，计算颜色直方图，将数组排序，并得到 K 个最流行颜色。最后再次扫描图像，用 K 个代表色中最匹配的一个显示每个像素。屏幕上并排显示两个版本的图像：左边为全彩色版本，右边为量化版本。

案例 11.5 中值切割颜色量化

(难度：Ⅲ)编写一个程序试验中值切割颜色量化算法。代码可从网上或者【Lindley92】等书中获取。从文件中读取一幅真彩图像，然后用户确定允许使用的不同颜色种数 K。接着程序扫描图像，计算由中值切割算法确定的 K 个代表色。最后再次扫描图像，用 K 个代表色中最匹配的一个显示每个像素。屏幕上并排显示两个版本的图像：左边为全彩色版本，右边为量化版本。

案例 11.6 八叉树颜色量化

(难度：Ⅲ)编写一个程序试验八叉树颜色量化算法。可以自己实现算法，也可以像前一实例一样获取代码。从文件中读取一幅真彩图像，然后用户确定允许使用的不同颜色种数 K。接着程序扫描图像，建立一个有 K 个代表色的八叉树，并遍历此树给每个叶节点一个索引值。最后再次扫描图像，找到每个像素在八叉树中的索引，并用相应颜色显示。屏幕上并排显示两个版本的图像：左边为全彩色版本，右边为量化版本。

一个有用的八叉树节点数据结构如下：

```
class Node{
    public:
    int isLeaf;                          //值为 1 若此节点为叶节点
    int whichLevel;                      //此节点的深度
    int index;                           //LUT 索引，在 Pass 1 中确定
    int numChildren;                     //此节点的子节点数
    long numTimes;                       //这个子块中出现的颜色个数
    long Rsum, Gsum, BSum;               //颜色值的累加和
    Node * nextCand;                     //在此深度进行简化的候选节点
    Node * child[8];                     //此节点的 8 个子节点
};
```

一些主要函数及主要流程如下：

```
insertTree(Node& node, BYTE rgb[])
{ //将 24 位颜色 rgb 插入子树
    if( node==NULL )
```

```
    makeNewNode(node);                    //构造并初始化一个新节点
if( node.isLeaf )
{
    node.NumTimes++;                      //代表的颜色数增一
    addColors(node, rgb);                 //颜色值累加
}
else
{
    childIndex=4*rgb[0]+2*rgb[1]+rgb[2];
        insertTree(childIndex, rgb);
    }
}
```

如何寻找可简化节点？ 维护一个数组 reducible[]，它有 7 个栈，每个栈对应树的一层。reducible[i]包含节点指针，指向树中第 i 层可简化节点(其包含子节点数超过 1)。每次向八叉树中插入一种颜色时，沿一条特定路径到达一个叶节点，可能还会建立新的节点。在经过的每个节点处(除了最后的叶节点)，将子节点数增一并测试此数值，如果恰为 2，则此节点变为可简化节点，一个指向它的指针被压入相应的栈中。

当八叉树必须被简化时，从 $i=7$ 开始减小 i 值逐个测试这些栈：找到第一个非空栈 reducible[i]，栈头指向一个可简化节点，将此节点从栈中弹出并简化。

```
void reduceTree(void)
{
    thisNode=findReducibleNode();
    thisNode.isLeaf=1;
    numLeaves-=(thisNode.numChildren-1);
    freeChildren(thisNode);
}
```

进一步阅读

Feynman【Feynman63】关于颜色的章节写的很精彩，是介绍颜色理论的经典教材。Bill Meyer【Meyer81】全面讨论了颜色技术。Rogers【Rogers90】也完整地介绍过颜色理论及其在计算机图形中的管理。Hall 的“Illumination and Color in Computer Generated Imagery”【Hall88】更详细地介绍了颜色的产生和转换。

第12章 光线跟踪基础

本章学习目标

- 认识光线跟踪的基本概念
- 理解光线跟踪的数学和算法基础
- 建立并渲染包含球体、圆锥体、圆柱体、凸多面体以及其他实体的场景
- 创建包含光线折射和透明度处理的高真实感图像
- 建立处理 3D 实体纹理和位图的工具
- 对场景中物体的表面添加纹理，使之更加生动逼真
- 学习如何创建物体的 3D 实体纹理，如木质纹理和大理石纹理等，并学习如何光线跟踪这些物体
- 利用构造实体几何(CSG)方法创建更多几何形状的物体，并学习如何光线跟踪由这些物体构成的场景

预览

12.1 节介绍光线跟踪技术，12.2 节介绍在执行光线跟踪算法前需要设定的照相机的位置和场景的几何信息。12.3 节给出了光线跟踪的一个实例。12.4 节介绍了一些方便光线跟踪的基本几何形状，并引出了光线跟踪算法的核心技术，就是一条光线如何和物体求交。在该节中还讨论了如何将光线变换到物体的一般坐标系中。

12.5 节介绍了一些面向对象编程的思想和方法，并设计了一些类和程序来构成一个光线跟踪器，还讨论了不同类对象之间的相互作用关系。一个基本的类是 polymorphism，在该节中将讨论类 polymorphism 的设计不仅避免了大量代码的编写，而且使得程序更加可靠和灵活。该节实现了一个简单但完整的光线跟踪器。12.6 节考虑光线如何与扩展集合中的几何物体求交，如圆台和凸多面体等。

12.7 节利用第 8 章中提出的渲染模型建立了一个全彩色的光线跟踪器，该光线跟踪器能处理物体表面的环境光、漫反射光以及镜面反射光等。12.8 节讨论如何在物体表面绘制纹理，其中包括有图片生产的 2D 纹理，以及如木质纹理和大理石纹理等 3D 纹理。关于光线跟踪的反走样问题将在 12.9 节中处理。12.10 节讨论如何显著地加速光线跟踪算法。我们可以利用包围盒或其他包围区域技术避免大量的耗时的光线求交计算。

12.11 节描述如何生成二级光线来加强光线跟踪器的真实感。该节还描述了一个生成真实阴影的方法。12.12 节描述了具体生成二级光线的方法来模拟有光泽物体表面的反射光线，以及穿过透明物体的折射光线。

12.13 节扩充了可以光线跟踪的物体集合，并通过构造实体几何(CSG)方法来生成新形

状的物体；新的适合任意形状物体的光线跟踪方法也随之给出。在本章的结尾给出了许多案例分析，通过这些案例分析能测试光线跟踪器的效果，并引导我们对光线跟踪器进行改善。

12.1 前　言

在第 6～8 章中，我们介绍了由多边形网格构成的场景的渲染方法，包括一个描述（近似描述）了光线如何在多边形表面反射的渲染模型。另外，Gouraud 和 Phong 插值渲染方法防止了多边形网格模型表面的面片化现象。因此渲染得到的图像显示过渡光滑的表面（轮廓边界除外），即使模型的表面由离散面片组成。

光线跟踪提供了一个渲染场景的强劲方法。图 12.1 显示了光线跟踪的基本想法。设想屏幕缓存为由空间中的像素排列成的数组，人眼透过每个像素点观察场景。对于屏幕缓存上的每一个像素点，考虑一个问题：人眼透过这个像素点会观察到什么？我们可以设想场景中的 P 点将会发射一条光线穿过该像素点到达人眼，于是像素点的颜色就设为该光线的颜色。

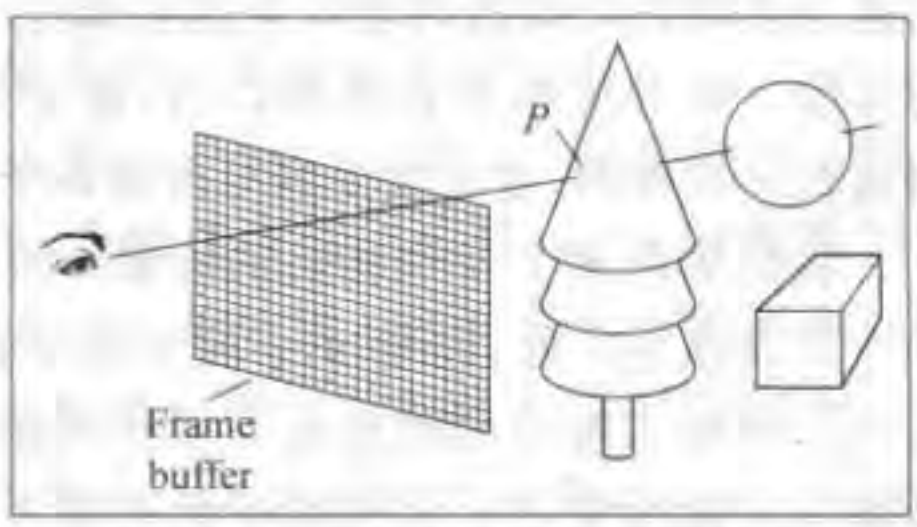


图 12.1　通过屏幕上一个像素点观察场景中的一点

实际上光线跟踪的过程和上面描述的过程有些区别，这一点常常困扰很多学生，我们将在本章中详细讨论。光线被设定为由眼睛发射出来，并穿过每个像素点的中心进入场景。光线进入场景后，检测场景中每个物体和光线的相交情况。如果与物体有相交，记录第一个和光线相交的物体，并记录相交的点。这里我们引进参数 t 来关联抽象的时间描述和光线的位置，这样一来，我们就可以说“光线现在在这里”或者“光线先到达这一点”。这里，我们应该看到光线跟踪的一个显著性质：它自动解决了隐藏面剔除问题，因为与光线相交的第一个曲面总是更接近人的眼睛，许多远处的曲面被自动忽略了。在给场景设定光源后，前面描述过的渲染模型将运用到第一个光线与场景相交的点，并计算该点处的环境光、漫反射光和镜面反射光。最后计算得到的颜色设为该像素点的颜色。

因为我们跟踪光线在场景中的传播路径，所以一些有趣的视觉效果很容易添加进来，例如阴影、光线的反射和折射效果等，渲染出用其他方法很难得到的高真实感的场景。光线跟踪的另一个优点是它可以处理更多类型的几何物体而不仅仅是多边形网格模型。几何实体可以由一些基本的几何形状构造得到，如球体、圆锥体和圆柱体等。这些基本几何形状都可以用数学公式精确表示出来，当然我们不要求每个形状都一定要写出它的数学表达式，但是如果需要的时候我们可以利用这些数学表达式。在把几何物体添加到场景中之前，我们也可以对它作用一个仿射变换来改变它的大小和方向，使得最后得到的场景更加复杂。在描述场景中的物体时，我们将会看到本章中提供的**场景描述语言**（Scene Description Language）是多么有用。

在本章中，我们将会描述生成复杂场景的高质量渲染图像的光线跟踪算法基础。我们采用渐进式的学习过程，因此在几个小节后我们就能了解到如何通过少量的编程得到一个

简单场景的渲染图像。其他复杂的算法将逐个介绍来建立更加高级的光线跟踪技术。为了激励读者更加深入地学习和实现本章中的算法,我们提供了一些精美的插图(如图 12.52)。从这些插图可以看出,光线跟踪技术可以渲染出极具真实感的图像。

12.2 建立光线跟踪的几何场景

为了跟踪穿过某个屏幕像素的光线的路径,我们需要一种光线的简单表示形式。参数表示法是目前已知的最易用的表示光线的方法。为了便于参考,我们将需要将第 6 章和第 7 章中涉及取景过程的内容集中到一起。需要事先说明的是,为了精确描述光线跟踪的几何信息我们需要设定大量的参数。大量的参数往往让人难以接受,所以我们建议读者在开始的时候最好能在纸上记下一些关键的参数。到对这些参数熟悉以后,你会发现很多参数都是相互关联的,这样就更容易理解各个参数的意义。

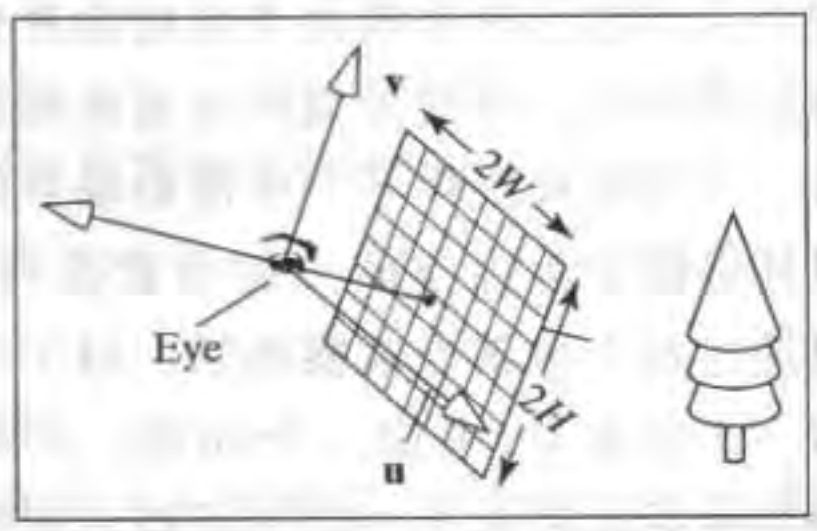


图 12.2 设定光线跟踪的相机位置

我们采用第 7 章中相机的设置:如图 12.2 所示,视点设在点 *eye* 处,向量 **u**、**v** 和 **n** 确定相机三个轴的方向。视域四棱锥的前截面与视点的距离为 N,投影屏幕与前截面重合。相机的形状和第 7 章中描述的相同:如图 12.3 所示,相机的视角为 θ,前截面上的投影屏幕的宽高比为 *aspect*。于是投影屏幕在 v 方向上的范围为 $-H$ 到 H,**u** 方向上的范围为 $-W$ 到 W,其中 H 和 W 由下面的等式给出:

$$H = N \times \tan(/2)$$
$$W = H \times aspect \tag{12.1}$$

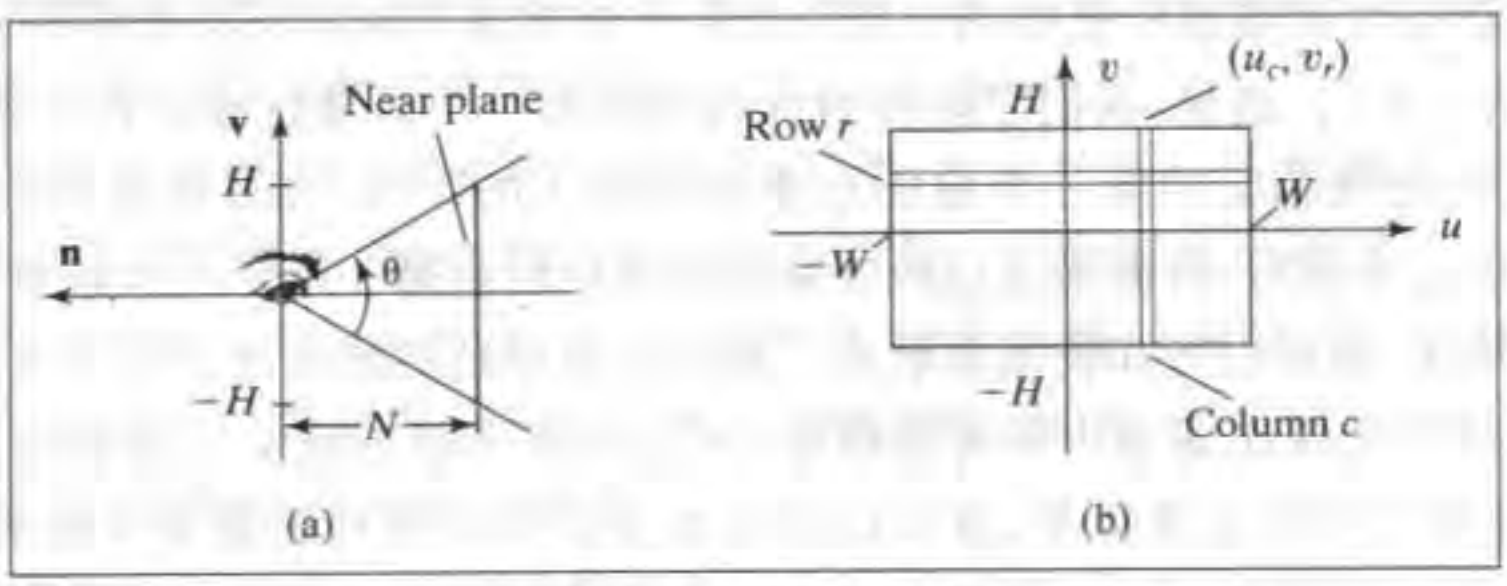


图 12.3 相机的形状

(a) 相机的形状;(b) 屏幕上的像素点

在第 7 章中我们曾提到视口变换决定了前截面上的点和视口上像素点之间的一一对应关系,这里我们继续采用这样的对应关系,但没有显式作用视口变换。事实上,我们可以简单地认为视口和前截面上的投影屏幕窗口重合,这样一来,人眼就通过视口上每个像素点来观察场景。假设视口上有 $nCols \times nRows$ 个像素,考虑第 r 行第 c 列上的像素,这里 r 的范围为 0 到 $nRows-1$,c 的范围为 0 到 $nCols-1$,以后我们称该像素为"第 rc 个像素"。一般情况下,我们按从下到上,从左到右的顺序遍历所有视口上像素,如图 12.3(b) 所示。有一

个问题是：每个像素点的坐标是多少？

如图 12.3 所示，设每个像素的左下角的坐标为(u_c, v_r)，则

$$u_c = -W + W\frac{2c}{nCols}, \quad 其中\ c = 0,1,\cdots,nCols-1$$

$$v_r = -H + H\frac{2r}{nRows}, \quad 其中\ r = 0,1,\cdots,nRows-1 \tag{12.2}$$

为了用数学公式表示一条光线在 3D 空间的位置，我们需要知道光线经过前截面上的像素的坐标。实际上只要知道该像素点离视点 *eye* 的距离就可以了。从 *eye* 点出发，计算在 **u**、**v** 和 **n** 方向上需要移动多少距离才能到达该像素的左下角点。读者们可能会觉得光线的路径应该经过每个像素的中心点，而不是左下角点，这样的想法当然是合理的，但是一般像素都非常小，这两种方法的结果没有明显区别。我们设在 **n** 的反方向上移动距离 N，在 **u** 方向上移动距离 u_c，在 **v** 方向上移动距离 v_r，则光线经过的像素左下角点的三维坐标由下式给出：

$$eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v} \quad \leftarrow 像素左下角点在三维空间中的坐标 \tag{12.3}$$

沿着这条光线发射方向上的每一点，我们可以关联一个时间参数 t。随着时间 t 的增加，光线上的点距离视点越来越远。我们设定 $t=0$ 时光线在视点的位置，$t=1$ 时光线到达像素的左下角点。光线跟踪中涉及到的基本计算就是当 t 在 0 到 1 之间变化时确定光线的位置，特别是要找出场景中有哪些物体和光线相交。光线沿着一条射线的方向匀速向前传播，我们要确定光线的传播路径是否和场景中的物体相交。如果用伪代码表示的话，光线传播路径的表达式可以写成：

$$r(t) = eye(1-t) + pixelcorner\ t \tag{12.4}$$

把 *pixelcorner* 的坐标代入上式，我们得到第 rc 光线参数表示形式：

$$r(t) = eye(1-t) + (eye - N\mathbf{n} + u_c\mathbf{u} + v_r\mathbf{v})t \tag{12.5}$$

稍作计算就会发现，该光线当 $t=0$ 时在视点位置，当 $t=1$ 时在第 rc 个像素的左下角点处。对于一条光线来说，它的起点和传播方向是我们比较关心的，所以对等式(12.5)作简单处理我们得到经过第 rc 个像素的光线如下式：

$$r(t) = eye + \mathbf{dir}_{rc}t$$

其中

$$\mathbf{dir}_{rc} = -N\mathbf{n} + W\left(\frac{2c}{nCols} - 1\right)\mathbf{u} + H\left(\frac{2r}{nRows} - 1\right)\mathbf{v} \tag{12.6}$$

在本章中，我们将会用上面的等式表示一条光线，而且在用程序实现光线跟踪算法是会直接利用到上述等式，所以读者有必要对光线的表达式深刻理解。

用上述等式表示的这簇光线有一个重要的性质，那就是当 t 不断增加时光线的位置离视点越来越远。你可以计算一下向量 $r(t)-eye$ 的长度来验证这条性质。如果光线分别在 t_a 和 t_b 时刻与两个物体相交，那么离视点近的物体与光线相交时对应的 t 值比较小。因此，在跟踪每条光线并检查它与场景中物体的相交情况时，我们只要记录光线第一次和物体相交的时间就可以了，光线通过的像素也最终根据这个记录来赋予颜色值。这样一来，我们自动解决了图形学中多年以来的一个重要问题，那就是“隐藏面剔除问题”，远处的曲面被自动忽略了。

练习

练习 12.2.1 补充算法细节

详细推导等式(12.1)～等式(12.6),证明第 rc 条光线的传播方向确实如文中所述。计算当 $t=2$ 和 $t=-1$ 时光线的位置。

练习 12.2.2 关于光线的数值计算

假设相机的设置是 $eye=(0,0,-1)$,$\mathbf{u}=(1,0,0)$,$\mathbf{v}=(0,1,0)$和 $\mathbf{n}=(0,0,-2)$,视角 θ 为 30°,前截面的宽高比为 1.5。又设 $nRows=480$,$nCols=640$,请给出第 rc 条光线的参数表达式,这里 $r=100$,$c=200$。

练习 12.2.3 像素点的位置?

请写出第 rc 个像素的中心点的坐标等式,而不是像素的左下角点坐标。

练习 12.2.4 关于光线的渐进计算

由等式(12.6)可知,如果按扫描线的顺序遍历所有光线,那么当前光线的方向可以由前一条光线的方向经过一个向量加法得到:请将 $\mathbf{dir}_{r,c+1}$ 写成关于 $\mathbf{dir}_{rc}$ 的形式。

12.3 光线跟踪过程纵览

我们首先给出一个光线跟踪器的基本框架,描述其中涉及到的一些基本计算。然后,我们介绍如何实现光线跟踪算法中的关键部分。

图 12.4 给出了实现光线跟踪器的基本步骤。需要光线跟踪的场景一般由一些几何物体与光源构成,比如场景中会包含球体、圆锥体、长方体和圆柱体等,每个几何物体都有确定的形状、大小和位置信息等。我们用一定的形式描述这些几何物体,并把它们存放到一个物体列表中。我们还需要如上一节所述对场景设定相机参数,然后对每一个像素都建立一条

```
<定义场景中的物体和光源>
<设定相机位置>
for(int r=0; r<nRows; r++)
  for(int c=0; c<nCols; c++)
  {
   <1. 建立第 rc 条光线>
   <2. 求出场景中所有与第 rc 条光线相交的物体>
   <3. 找出离视点最近的与光线相交的物体>
   <4. 计算光线与物体的交点坐标,并得到物体表面在该点的法向>
   <5. 计算在交点处人眼看到的光线的颜色>
   <6. 第 rc 个像素被赋予上述颜色>
  }
```

图 12.4 光线跟踪器构架的伪代码

从视点出发穿过该像素左下角点的光线，这里涉及到计算第 rc 条光线的发射方向 $\mathbf{dir}_{rc}$。整个光线跟踪的过程已经在图 12.4 中用伪代码的形式给出。

图 12.4 中的步骤 3～5 是之前没有提到过的，我们将在以后几节中详细介绍。我们先检测第 rc 条光线是否与物体列表中的物体相交，如果有相交，则记录光线 $r(t)$ 到达物体表面时的时间 t。当物体列表中的所有物体都已经被检测过，那么与光线相交时对应的 t 最小的物体离视点最近。计算该物体与光线的交点坐标，并计算物体表面在交点处的法向量。于是，在交点处人眼看到的光线的颜色就存储为像素的颜色。

图 12.5 显示了由一些圆柱体、球体以及三个圆锥体组成的简单场景，其中雪人主要由球体组成(观察雪人的帽子是由那些基本几何形状组成的?)，该场景中也包含了两个光源。注意到场景中的物体是可以相互贯通的，当把场景渲染成图像时，我们只关心最外层的曲面。

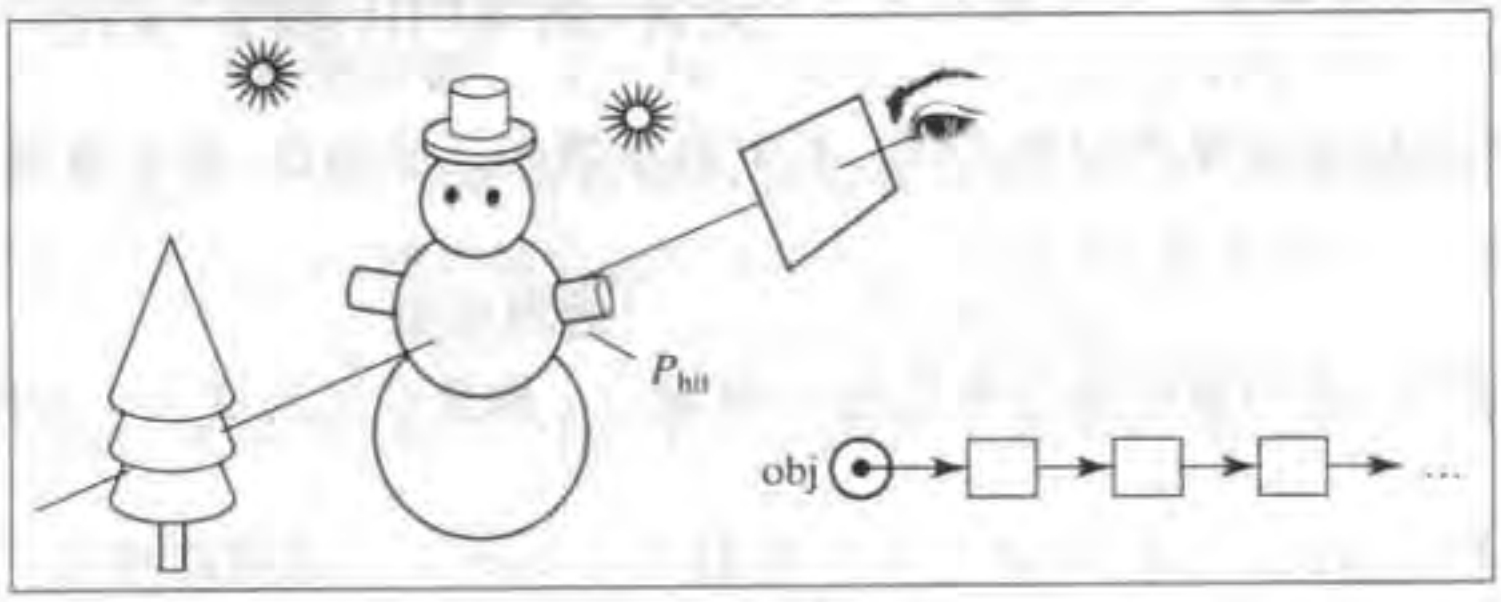


图 12.5　光线跟踪一个场景

场景中所有物体的描述都被保存在一个物体列表中，如图 12.5 所示。图中所画的光线与一个球体、一个圆柱体以及两个圆锥体相交，与其他的物体都不相交。在这个例子中，容易发现圆柱体是最先与光线相交的物体。设交点为 P_{hit}，则计算 P_{hit} 点的坐标只要把光线到达 P_{hit} 点时的时间 t_{hit} 代入光线的表达式(12.6)就可以了：

$$P_{hit} = eye + \mathbf{dir}_{r,c} t_{hit} \quad \langle 交点 \rangle \tag{12.7}$$

(请务必理解上述等式，因为它会在本章中多次出现。)对于某些像素，穿过它的光线可能与场景中的任何物体都不相交，那么这些像素的颜色就设为场景的背景颜色。

经过上面的描述，我们容易知道光线跟踪中的基本计算就是寻找光线与给定物体的交点。在本章中，我们将用一定的篇幅来介绍求交算法，并讨论如何使算法更加快速有效。

12.4　光线与物体求交

回顾一下第 5 章，我们曾建立了一个场景类 Scene，它能读入一个用 SDL 语言编写的文件并建立场景中的物体列表。在光线跟踪时，我们仍然用这个类来建立场景：

```
Scene scn;                    //实例化一个场景对象
Scn.read("myScene.dat");      //读入一个 SDL 场景文件
```

场景中的物体被创建后都被存放在物体列表中，每一个物体都是通用形状的一个实例，

并附加一个仿射变换决定物体的大小、方向以及在场景中的位置等。这里的通用形状指球体、圆锥体等,(后面我们将讨论关于每类形状的通用的精确意义)。在第 6 章中,我们介绍了一些通用形状,图 12.6 显示了我们将光线跟踪的通用形状,如球体、圆锥体、圆锥体、正方体、正方形以及四面体等。

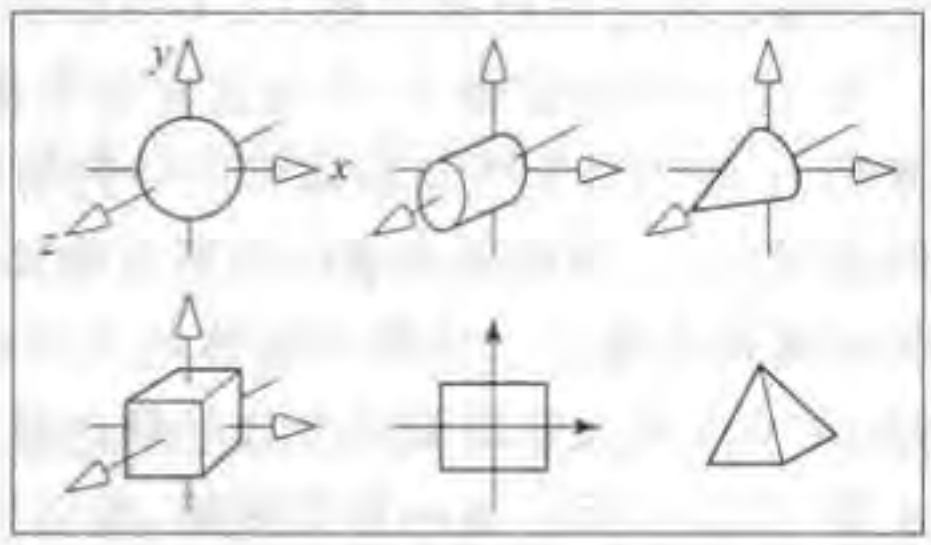


图 12.6 一些光线跟踪的基本通用形状

在第 5 章中我们利用 OpenGL 绘制物体(对每一类形状我们编写了 drawOpenGL()绘制方法),在本章中我们将利用光线跟踪的方法绘制物体。这样就涉及到光线与场景中物体的求交问题。如果我们对每个形状采用第 6.5.1 节中介绍的隐式表示方法,那么会使求交计算简单很多。比如,**通用球面**是一个半径为 1 中心在原点的球面,它的隐式表达式为:

$$F(x,y,z) = x^2 + y^2 + z^2 - 1 \quad \text{〈通用球面〉} \tag{12.8}$$

如果我们采用更为简单的记号 F(P),其中的参数为三维的点,那么通用球面的隐式表达式可以写成:

$$F(P) = |P|^2 - 1 \quad \text{〈通用球面〉} \tag{12.9}$$

如图 12.6 所示,通用圆柱面是半径为 1,纵轴与 z 轴重合且长度为 1 的圆柱面,它的隐式表达式为:

$$F(x,y,z) = x^2 + y^2 - 1 = 0, \text{且 } 0 < z < 1 \quad \text{〈通用圆柱面〉} \tag{12.10}$$

在实际场景中,每个通用形状都经过一个仿射变换得到一个新的形状,这时它的隐式表达式与通用形状的隐式表达式有很大的不同。但是,很快我们就会发现在计算光线和物体的交点时,只要求解光线和通用形状的交点就可以了。因此,通用形状的隐式表达式在光线跟踪算法中显得十分重要。

那么,我们如何求解一条光线和一个隐式表达式为 F(P)通用形状的交点呢?假设光线的起点为 S,传播方向为 **c**。采用这样的符号比之前我们采用的符号起点 *eye* 和方向 $\mathbf{dir}_e$ 更为简洁,于是光线的表达式由下式给出:

$$r(t) = S + \mathbf{c}t \tag{12.11}$$

我们容易知道:物体表面上所有的点满足 $F(P)=0$;且当光线上某一点 $r(t)$ 落在物体表面时,光线和物体相交。于是,点 $r(t)$ 在物体表面上的充要条件是 $F(r(t))=0$。假设在时间 t_{hit} 时,光线和物体相交,那么 t_{hit} 可以通过求解下面的等式得到:

$$F(S + \mathbf{c}t_{hit}) = 0 \tag{12.12}$$

光线跟踪算法的大部分计算都集中在如何高效地求解上述等式。下面我们将介绍当物体具有比较简单的表面形状时,如平面或球面等,上述等式的求解方法。

12.4.1 光线与通用平面求交

在场景中经常需要用一张平面来表示房间的地板和墙壁等,它们往往具有比较均一的颜色,或者被贴上一些纹理,如黑白棋盘纹理或扫描的图片等。

通用平面是指 xy—平面,或 $z=0$ 的平面,所以它的隐式表达式是 $F(x,y,z)=z$。当光线 $S+ct$ 与该平面相交时满足 $S_z+c_zt_h=0$,这是一个关于 t_h 的线性等式,它的解为:

$$t_h = -\frac{S_z}{c_z} \tag{12.13}$$

如果 $c_z=0$，光线和平面平行，此时光线和平面没有交点（除非 S_z 也为 0，这时光线嵌入到平面，也看不到任何反射光线）。否则，光线和平面的交点为 $P_{hit}=S-\mathbf{c}(S_z/c_z)$。

例 12.4.1　求解光线和平面的交点

请问光线 $r(t)=(4,1,3)+(-3,-5,-3)t$ 与通用平面的交点坐标是什么？

解：

利用等式(12.13)，求得 $t_h=-3/-3=1$；则交点坐标为 $S+\mathbf{c}=(1,-4,0)$。检验发现该点确实落在 $z=0$ 的平面上。

12.4.2　光线与通用球面求交

给定光线的表达式 $S+\mathbf{c}t$ 和通用球面的隐式表达式(12.9)，如何求解它们的交点呢？把 $S+\mathbf{c}t$ 代入等式 $F(P)=0$，我们得到 $|S+\mathbf{c}t|^2-1=0$，或者（参考等式(4.10)）[①]

$$|\mathbf{c}|^2t^2+2(S\cdot\mathbf{c})t+(|S|^2-1)=0 \tag{12.14}$$

这是关于 t 的二次等式：$At^2+2Bt+C=0$，其中 A、B 和 C 是

$$\begin{aligned} A &= |\mathbf{c}|^2 \\ B &= S\cdot\mathbf{c} \\ C &= |S|^2-1 \end{aligned} \tag{12.15}$$

我们用一元二次方程的求根公式求解上述等式：

$$t_h = -\frac{B}{A} \pm \frac{\sqrt{B^2-AC}}{A} \tag{12.16}$$

如果判别式 B^2-AC 的值为负，那么等式有实数解，光线和球面无交点。如果判别式的值为零，那么光线在球面上某一点擦过球面，和碰触到球面的时间为 $-B/A$。如果判别式的值为正，那么光线在两个时间点 t_1、t_2 和球面相交，分别对应等式(12.16)中的“+”和“-”（关于数值计算中的精度问题将会在练习中讨论）。

求解光线和平面或球面的交点问题相对比较简单，因为最终我们只要求解一个线性或者一元二次等式就可以了。在处理光线和其他基本形状求交问题时，往往导出的等式不易求解，我们将在练习中进一步讨论这个问题。

例 12.4.2　求解光线和通用球面的交点

请问光线 $r(t)=(3,2,3)+(-3,-2,-3)t$ 与通用球面的交点是什么？

解：

利用等式(12.15)得，$A=22$，$B=-22$ 和 $C=21$；代入等式(12.16)得，$t_1=0.7868$ 和 $t_2=1.2132$；于是两个交点分别是 $S+ct_1=(3,2,3)(1-0.7868)=(0.6393,0.4264,$

---

① 注意，为了表达简洁，我们并没有严格遵守符号的用法。S 是点而不是向量，所以严格来说我们不能用它来做点积。$|S|^2$ 是 $S_x^2+S_y^2+S_z^2$ 的简略表示方法。

0.6396)和 $S+ct_2=(3,2,3)(1-1.2132)=(-0.6393,-0.4264,-0.6396)$。容易验证发现这两个点确实落在通用球面上,与原点的距离为 1。读者还会发现两个交点原点对称,这是因为光线经过原点而且通用球面的中心落在原点上。

12.4.3 光线与仿射变换后的物体求交

在构造场景时,设计者会对每个物体施加一个仿射变换 T,来确定物体在场景中的大小、方向和位置。对物体施加仿射变换后,我们之前得到的等式和结果会产生什么样的变化?如果仿射变换后物体的隐式表达式变得非常复杂,那么等式(12.12)就很难去求解,这是我们不愿看到的。下面我们将证明,这种情况不会发生。

图 12.7 显示了变换 T 把通用球面 W' 映射为一个椭球面 W。光线 $S+\mathbf{c}t$ 何时和 W 相交呢?假设我们已经求得变换后的物体 W 的隐式表达式 $G()$,那么只要求解等式 $G(S+\mathbf{c}t)=0$ 来求出碰撞时间。

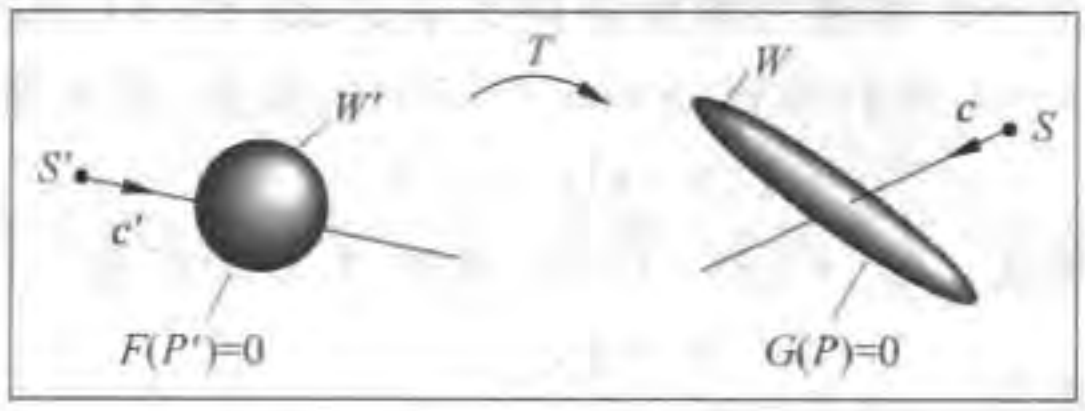


图 12.7 光线与一个椭球面相交

这样一来,问题就转化为求解变换后物体的隐式表达式。在第 6.5.3 节中我们看到,如果通用物体的隐式形式为 $F(P)$,那么变换后物体的隐式形式为 $F(T^{-1}(P))$,其中 T^{-1} 为 T 的逆变换(也就是说,如果 T 的变换矩阵为 M,那么 T^{-1} 的变换矩阵为 M^{-1})。于是,我们必须通过求解下面等式得到碰撞时间 t:

$$F(T^{-1}(S+\mathbf{c}t))=0 \tag{12.17}$$

这个等式看起来有点凌乱,但实际上它的几何意义非常简单:求解逆变换光线 $T^{-1}(S+\mathbf{c}t)$ 与原来的通用物体的碰撞时间!因为仿射变换 T 是线性的,所以逆变换光线为:

$$T^{-1}(S+\mathbf{c}t)=(T^{-1}S)+(T^{-1}\mathbf{c})t$$

在图 12.7 中,我们用 $S'+\mathbf{c}'t$ 表示逆变换光线(明显,逆变换光线还是一条光线,它的起点为 S',传播方向为 $\mathbf{c}'$)。假设仿射变换 T 的矩阵为 M,采用齐次坐标表示,我们得到逆变换光线的参数表达式:

$$\tilde{r}(t)=M^{-1}\begin{bmatrix}S_x\\S_y\\S_z\\1\end{bmatrix}+M^{-1}\begin{bmatrix}c_x\\c_y\\c_z\\0\end{bmatrix}t=\tilde{S}'+\tilde{c}'t \quad \text{〈逆变换光线〉} \tag{12.18}$$

这里 M^{-1} 在一个等式中与一个点相乘,在另一个等式中与一个向量相乘。点 S' 的坐标由 $\tilde{S}'$ 舍弃 1 得到;向量 c' 的坐标由 $\tilde{c}'$ 舍弃 0 得到。

因此,我么可以求解逆变换光线与原物体的交点,来代替求解光线与变换后物体的交点。场景物体列表中的每一个物体都关联了一个仿射变换,求解光线 $S+\mathbf{c}t$ 与变换后物体

的交点需要下面几个步骤：

(1) 求出逆变换光线 $S'+\mathbf{c}'t$。

(2) 求出逆变换光线与通用物体的碰撞时间 t_h。

(3) 把碰撞时间 t_h 代入等式 $S+\mathbf{c}t$ 得到实际的交点坐标。

这个方法的优点是我们只要设计代码实现光线与通用物体的求交算法就可以了。这是个双赢的结果：程序员只要编写更简单的代码；计算机只需更少的计算。我们在构造场景时可以随意对物体施加任何仿射变换来得到更为复杂有趣的场景，而不用担心在光线跟踪时需要编写更为复杂的代码。对物体的变换转化为对光线的变换，由此带来的好处主要是得益于光线是一条直线。

例 12.4.3　求解光线和变换后物体的交点

假设椭球面 W 是由通用球面变换得到，用 SDL 语言描述如下：

```
translate 2 4 9
scale 1 4 4
sphere
```

可知，球面先经过一个放缩然后平移变换为椭球面。它的仿射变换和逆变换矩阵分别为(请验证)：

$$M=\begin{bmatrix}1&0&0&2\\0&4&0&4\\0&0&4&9\\0&0&0&1\end{bmatrix},\quad M^{-1}=\begin{bmatrix}1&0&0&-2\\0&\frac{1}{4}&0&-1\\0&0&\frac{1}{4}&-\frac{9}{4}\\0&0&0&1\end{bmatrix}$$

请求出光线$(10,20,5)+(-8,-12,4)t$ 与 W 的交点。

解：

逆变换光线为$(8,4,-1)+(-8,-3,1)t$(请验证)。利用等式(12.15)得到$(A,B,C)=(74,-77,80)$。于是判别式的值为 9，光线和物体有两个交点。从等式(12.16)我们得到碰撞时间为 1.1621 和 0.9189。把比较小的碰撞时间代入光线的表达式就得到所求的交点坐标，$(10,20,5)+(-8,-12,4)0.9189=(2.649,8.97,8.67)$。你将如何验证该点确实落在变换后的物体上？

练习

练习 12.4.1　求交点

请求出中心在$(1,2,1)$，半径为 5 的球面和光线$(3,5,8)+(-4,-2,-6)t$ 的碰撞时间以及交点坐标。

练习 12.4.2　它和平面相交吗？

已知一个平面由 SDL 描述如下：translate 4 5 6 rotate 90 1 0 0 plane；请问光线$(10-t,8-2t,3+t)$于何时何处与该平面相交？

12.5 一个光线跟踪器的应用

有了前面介绍的光线与物体的求交算法,我们现在可以根据图 12.4 描述的框架构造一个光线跟踪器。我们将会利用场景类 Scene 和第 5 章中介绍的 SDL 语言,因为它们是描述复杂场景的有用工具。我们假定场景已经给定,场景中的物体保存在一个物体列表中,而且每一个物体都关联一个仿射变换。

一个光线跟踪器需要处理一些相互影响的对象,比如:对场景取景的照相机;场景的投影屏幕;从视点发射出来并穿过场景的光线;以及包含许多几何物体和光源的场景。在用面向对象的思想实现光线跟踪器时,我们要精心设计哪些类是需要的,每个类又要具有哪些功能,以及每个类要具有什么样的属性来支持它的功能。这里我们采用自上而下的设计方法,当然你也可以采用其他方法来解决这个问题。

在我们的设计方法中,我们把光线跟踪的任务分配给照相机,于是我们在已有的 Camera 类里添加一个方法:

```
Void Camera::raytrace(Scene& scn, int blockSize);
```

我们传递给照相机两个参数,一个需要光线跟踪的场景对象,另一个是 blockSize,它的意义我们下面会介绍。该方法产生从视点出发并经过每个像素左下角点进入场景的光线,并求出人眼在每条光线上看到的颜色值,然后把求得的颜色值赋给相应的像素。

我们将用 OpenGL 来实现像素的绘制,并在 raytrace()中设定模型视图和投影矩阵来直接在视图上绘制,具体的过程将在下面叙述。这样一来,程序主循环中 display()方法就显得比较简单:它只要清除屏幕并告诉相机类对象 cam 去光线跟踪场景就可以了,如下面代码所示:

```
void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);          //清除屏幕
    cam.raytrace(scn, blockSize);          //光线跟踪场景
}                                          //适当地设置像素的颜色
                                           //用正确的颜色填充该像素
```

读者在设计自己的光线跟踪器时,对上面代码的微小改变可能会带来很大的变化。在前面我们已经实现了用 OpenGL 函数来渲染 SDL 语言描述的场景,所以在用光线跟踪算法渲染场景前我们不妨给出场景的一个粗糙预览图。场景的预览图可以快速地被渲染出来,它能让用户知道相机的取景正确并且场景中的物体都在它们合适的位置上。无论是相机或者是场景的设置有问题,用户都可以进行调整,然后再次查看预览结果。如果所有的设置都正确,那么光线跟踪的渲染结果将和场景的预览结果吻合。因此,通过对比预览结果,用户可以很明显地发现光线跟踪器中的错误。

为了添加对场景的预览,可以把 display()方法简单扩展为:

```
void display(void)
```

```
{
    //清除屏幕并重置深度缓存
    glClear(G L_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    cam.drawOpenGL(scn);
      //绘制场景的预览结果
    cam.raytrace(scn, blockSize);
      //用光线跟踪的渲染结果覆盖预览结果
}
```

图 12.8　在预览场景上绘制光线跟踪结果

图 12.8 显示了光线跟踪的一个中间结果：首先绘制场景的一个粗糙的预览结果，在预览结果上再绘制光线跟踪的结果，在该图中只有场景的下半部分已经用光线跟踪绘制。这里需要强调的是，光线跟踪的结果和预览场景完全重合。

整合：raytrace()的构成

Camera 类中的 raytrace()方法要实现图 12.4 中描述的光线跟踪器的框架。对于每一个在第 r 行第 c 列上的像素，需要产生一条从视点出发并穿过第 rc 个像素的左下角点进入场景的光线。因此，我们需要构造一个光线类 Ray，它的开头部分代码如图 12.9 所示。Ray 类需要一个属性来保存光线的起点，以及另一个属性保存光线的发射方向，以后我们将会添加 Ray 类的其他属性。另外有两个方法用于设置光线的起点和发射方向。

```
class Ray{
public:
    Point3 start;
    Vector3 dir;
    void setStart(Point3& p){start.x=p.x; etc..}
    void setDir(Vector3& v){dir.x=v.x; etc..}
    //其他属性和方法
};
```

图 12.9　Ray 类的开头部分代码

图 12.10 描述了 raytrace()方法的框架：它设置光线的起点为视点，然后对应于每一个像素计算并设置第 *rc* 条光线的发射方向(根据等式(12.3))。Scene 类对象 scn 将计算沿着发射光线返回的颜色的值：crl=scn. shade(theRay)。方法 shade()做了核心的工作：它生成一条射入场景的光线，求出光线与场景中物体的交点，计算沿着发射光线返回的颜色值。在本章的后面部分，我们将具体实现方法 shade()的功能。

绘制用于预览的像素块

参数 blockSize 决定了在每一步绘制时像素块的大小。在实现光线跟踪器时，绘制像素块而不是每个像素点是一个简单的加速方法。此时得到的渲染图像比较粗糙但渲染速度很快：我们仅跟踪穿过像素块左下角点的光线；如果跟踪穿过屏幕上每个像素的光线的话，图

```
void Camera::raytrace(Scene& scn, int blockSize)
{
    Ray theRay;
    theRay.setStart(eye);
    //设置 OpenGL 显示简单的预览结果
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0,nCols,0,nRows);          //整个投影屏幕设为视口
    glDisable(GL_LIGHTING);               //于是 glColor3f()能正确着色
    //开始光线跟踪
    for (int row=0; row<nRows; row+=blockSize)
    for (int col=0; col<nCols; col+=blockSize)
    {
        计算光线的发射方向
        theRay.setDir(<direction>);       //设置光线的发射方向
        Color3 clr=scn.shade(theRay);     //计算颜色值
        glColor3f(clr.red,clr.green.clr.blue);
        glRecti(col,row,col+blockSize,row+blockSize);
    }
}
```

图 12.10 raytrace()的框架

像将由逐个像素渲染得到,渲染速度比较慢。

图 12.11(a)显示了一个简单的例子来说明像素块的构成。屏幕上实际的像素个数是 16 行 32 列,这里设置 blockSize 的值为 4,于是每个块由 16 个像素组成。计算穿过该块的左下角点的光线的颜色值,那么块中所包含的 16 个像素都被设置为该光线的颜色值。此时得到的渲染图像是分辨率为 4×8 的光栅化结果,是光线跟踪的全分辨率渲染结果的一个粗糙近似,但是它的渲染速度很快。如果粗糙的渲染图像是正确的,那么用户可以设置 blockSize 的值为 1 来得到场景的全分辨率渲染结果。图 12.11(b)显示了设置不同的块的大小时一个简单场景渲染结果,对应的块大小分别为 4、2 和 1。第一种设置的渲染速度是最后一种设置的 16 倍。注意到当块的大小为 4 时比块的大小为 1 时得到渲染图像的效果粗糙很多,但是地球仪和地板上的纹理还是清晰可见的。

注意到 raytrace()中设置了 OpenGL 的模型视图矩阵和投影矩阵来正确绘制像素块。模型视图矩阵被设为单位阵,投影矩阵也只是简单地把窗口缩放为视口大小。这样设置使得 OpenGL 的绘制过程显得更加明晰,于是我们可以利用 glRecti()方法直接将一个正方形绘制到视口上。这里,我们假设在程序开始运行的时候,已经利用 glViewport(0,0,nCols,nRows)方法将视口设置为整个投影屏幕。OpenGL 的光照状态应该关闭,否则 glColor3f()不能得到正确的着色结果。

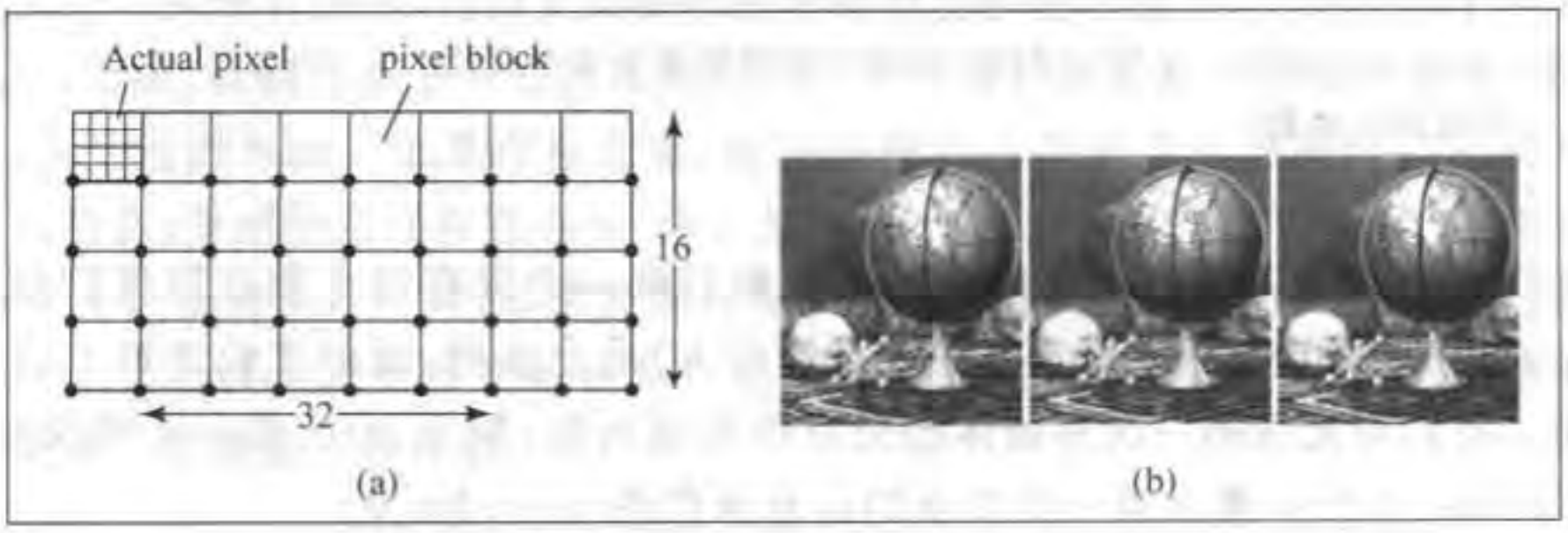


图　12.11

(a) 设置像素块的大小用于预览光线跟踪结果；(b) 块的大小分别为 4、2 和 1 时光线跟踪的结果

shade(ray)的基本构成

光线跟踪器的核心内容在 shade()方法内，图 12.12 中用伪代码描述了它的算法框架。它的第一部分工作是判断光线是否和物体相交，如果是，找出相交的物体。为了完成这个工作，它需要一个 getFirstHit(ray, best)，该方法将光线与物体第一次相交的数据保存到名字为 best 的相交记录中。我们将在下面更详细地介绍 getFirstHit()和相交记录的结构。一旦光线的相交信息已经得到，shade()将计算光线的颜色值。如果没有物体和光线相交，那么 shade()简单地返回背景的颜色。如果光线确实和一个物体相交，那么 shade()将根据第 8 章中介绍的经典渲染模型计算光线颜色的各个组成部分，其中包括物体发射出的光线(如果物体是发光体)，以及环境光线、漫反射光线和镜面反射光线。另外，shade()还考虑有光泽的物体表面的反射光线，以及穿过透明物体的折射光线。

```
Color3 Scene::shade(Ray& ray)
{       //返回该光线的颜色
    Color3 color;                          //返回的综合颜色
    Intersection best;                     //用于保存目前为止最佳的碰撞记录
    getFirstHit(ray, best);                //填充"best"记录
    if (best.numHits==0)
        return background;
    color.set(the emissive color of the object);
//添加更多的构成光线
    color.add(ambient, diffuse, and specular components);
    color.add(reflected and refracted components);
    return color;
}
```

图 12.12　shade()的算法框架

getFirstHit()寻找最早和光线相交的物体，并把相关信息记录在相交记录 best 中。我们用 Intersection 类来实现相交记录，它的结构由下面的伪代码描述：

```
class Intersection{
public:
```

```
    int numHits;        //有正的相交时间的交点的数目
    HitInfo hit[8];     //交点列表-以后可能需要多于 8 个交点
    …其他成员函数…
};
```

它有两个属性域-有正的相交时间的交点数目和一个保存每个交点信息的数组。我们认为有正的相交时间的交点(位于光线起点的前方)是合法的,而对于位于视点后面的交点不感兴趣。我们对光线第一次和物体的交点特别感兴趣,如果 inter 是一个相交记录,而且 inter. numHits 大于 0,那么第一个交点的信息保存在 inter. hit[0]。

为什么我们要保存光线与物体的所有交点信息,而不是仅仅保存第一个交点信息?这样做一个很大的好处是能渲染布尔物体(见第 12.12 节)。为了能处理布尔物体,我们必须保存光线与物体的所有交点信息,所以为了以后处理方便我们先要花精力保存所有需要的信息。一般来说,视点在所有物体的外部,而且一条光线和物体相交两次:一次相交时进入物体,另一次相交时从物体中出来。在这种情况下,inter. numHits 等于 2,inter. hit[0]描述光线进入物体的位置,inter. hit[0]描述光线从物体中出来的位置。但是有些物体,如图 12.13 中所示的圆环和布尔物体,与光线有多于两个交点。图 12.13(a)所示的物体与光线有四个交点,所以 inter. numHits 为 4,我们把交点信息保存在 inter. hit[0],……, inter. hit[3]。图 12.13(b)中的物体(假设是透明的)与光线有八个交点,但是视点在物体内部,只有第 6 次、第 7 次和第 8 次相交在正的时间发生(所以 inter. numHits 为 3)。

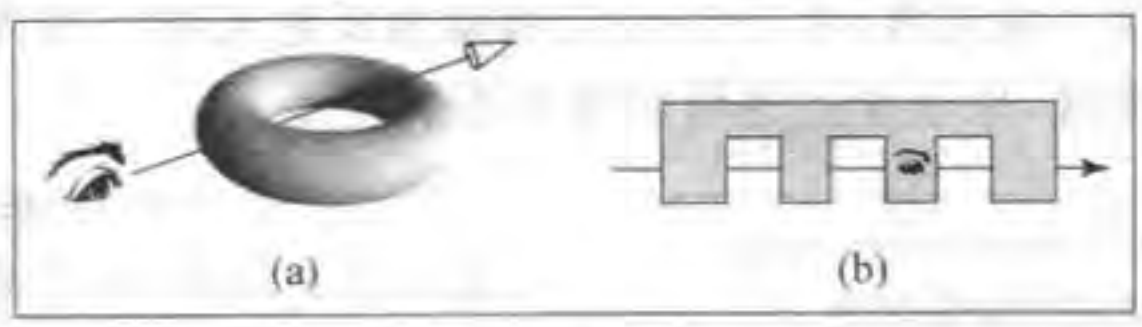


图 12.13 物体与光线有多个交点,有些相交在负的时间发生

每个交点的信息保存在相交记录类 HitInfo 中,它的结构如下所示:

```
class HitInfo{                          //每个交点的信息
    public:
    double hitTime;                     //相交时间
    GeomObj * hitObject;                //相交的物体
    bool isEntering;                    //光线是进入还是出来
    int surface;                        //相交于哪个表面
    Point3 hitPoint;                    //交点的坐标
    Vector3 hitNormal;                  //交点处曲面的法向
    …其他成员变量和方法…
};
```

一个相交记录包含交点的多个属性:相交时间,一个指向相交物体的指针,光线在交点处是进入物体还是从物体中出来,交点的坐标,以及物体表面在交点处的法向。所有这些属性在光线跟踪过程中都是非常重要的。

图 12.14 中用伪代码描述了 getFirstHit()的算法框架。它用指向物体的指针 pObj 遍

历了物体列表中的所有物体，检测光线是否和物体相交。为了检测相交情况，它调用了每个物体的 hit()方法，我们将在下一节的开头介绍这个方法。如果存在有效的交点，每个 hit()返回 true，否则返回 false。如果存在交点，该方法建立一个相交记录来保存光线与该物体的所有交点信息，并把信息保存在 inter 中。然后 getFirstHit()比较 inter 中的第一个交点时间和 best 中记录的交点时间，如果发现有更早的交点时间，那么把 inter 中的数据复制到 best 中。best.numHits 的值被初始化为 0，这样当 pObj 遍历物体列表时，第一个有效的相交记录的交点个数将被正确计数。

```
void Scene::getFirstHit(Ray& ray, Intersection& best)
{
    Intersection inter;                     //实例化相交记录类
    best.numHits=0;                         //还没有交点
    for(GeomObj * pObj=obj; pObj!=NULL; pObj=pObj->next)
    {    //检测场景中的每一个物体
        if(!pObj->hit(ray, inter))          //光线是否和 pObj 相交?
            continue;                       //无交点：检测下一个物体
        if(best.numHits==0 ||               //best 中还没有交点信息
            inter.hit[0].hitTime<best.hit[0].hitTime)
            best.set(inter);                //复制 inter 到 best
    }
}
```

图 12.14　getFirstHit()函数

注意到 getFirstHit()把复杂的光线与物体的求交计算交给了每个物体的 hit()方法。我们将为每种类型物体实现一个 hit()方法。为了提高效率，我们将充分利用每种通用物体的形状特征设计适当的 hit()方法。这是 C++ 语言**多态性**特征的一个典型应用，利用多态性可以简化代码，并且使程序更加稳定高效。多态性特征允许我们为每种物体编写 hit()方法时用同一个函数名“hit”。编译器不知道调用哪个 hit()，因为它甚至不知道物体列表中物体的类型是什么。实际上正确的 hit()将在程序运行时被系统调用，从而得到正确的求交结果。

物体类中的 hit()方法的主要任务是对给定的一条光线建立一个相交记录，并把光线和该物体的所有交点信息填充到相交记录中。下一节我们将实现球体的 hit()方法。

12.5.1　光线和球体的求交程序

图 12.15 显示了 Sphere 类的 hit()方法。它首先利用球体的逆变换将光线 r 变换到球体的通用坐标系，xfrmRay()方法根据等式(12.18)实现了这个变换(见练习)。然后计算等式(12.12)中的系数 A、B 和 C，以及判别式 B^2-AC。如果判别式的值为负，此时等式没有实数解，可知光线和物体没有交点。于是 hit()方法的返回值为 false，inter 变量也将不会被使用到。

```
bool Sphere::hit(Ray &r, Intersection& inter)
{
    Ray genRay;                          //用于生产泛型光线
    xfrmRay(genRay,invTransf,r);
    double A, B, C;
    A=dot3D(genRay.dir, genRay.dir);
    B=dot3D(genRay.start, genRay.dir);
    C=dot3D(genRay.start, genRay.start)-1.0;
    double discrim=B*B-A*C;
    if(discrim<0.0)                      //没有交点
        return false;
    int num=0;                           //目前的交点个数
    double discRoot=sqrt(discrim);
    double t1=(-B-discRoot)/A;           //第一次相交时间
    if(t1>0.00001)                       //交点是否在视点前方?
    {
        inter.hit[0].hitTime=t1;
        inter.hit[0].hitObject=this;
        inter.hit[0].isEntering=true;
        inter.hit[0].surface=0;
        Point3 P(rayPos(genRay, t1)); //交点坐标
        inter.hit[0].hitPoint.set(P);
        inter.hit[0].hitNormal.set(P);
        num=1;                           //一个有效交点
    }
    double t2=(-B+discRoot)/A;           //第二次相交时间
    if(t2>0.00001)
    {
        inter.hit[num].hitTime=t2;
        inter.hit[num].hitObject=this;
        inter.hit[num].isEntering=false;
        inter.hit[num].surface=0;
        Point3 P(rayPos(genRay, t2)); //交点坐标
        inter.hit[num].hitPoint.set(P);
        inter.hit[num].hitNormal.set(P);
        num++;                           //另一个有效交点
    }
    inter.numHits=num;
    return (num>0);                      //true 或者 false
}
```

图 12.15 Sphere 类的 hit()函数

如果判别式的值为正，由等式(12.16)得到两个交点时间。我们记值较小的交点时间为 t_1，值较大的交点时间为 t_2。如图 12.16 所示，这里有三种情况。当 t_1 和 t_2 的值都大于零时，球体在视点前方；当 t_1 的值为负 t_2 的值为正时，视点在球体内部；当 t_1 和 t_2 的值都为负时，球体在视点的后方。

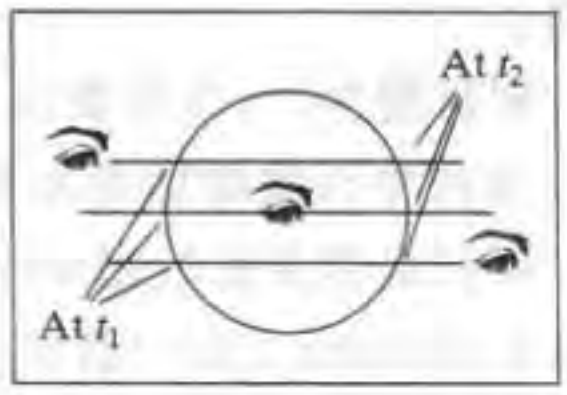


图 12.16　哪些交点在视点前方？

如果 t_1 的值严格大于 0，那么 t_1 就是有效的第一次相交时间，我们把交点信息保存在 inter.hit[0]中，并设变量 num 的值为 1，显示已经有一个交点存在。如果 t_2 的值也为正，那么第二个交点的信息保存在 inter.hit[num]中。(如果第一个交点时间为负，第二个交点时间可能为正也可能为负，相应的交点信息被自动保存在 inter.hit[0]中。)

在保存交点信息时，我们利用了球体的特征。比如说，我们知道球体是凸物体，光线必然先由一个交点进入球体，然后从另一个交点出来。变量 surface 的值设为 0，因为球体只有一个表面：我们以后将处理光线和多个表面相交的情况。在通用坐标系下，光线和球体的交点坐标也保存在 hitPointfield 中。根据定义可知，交点始终落在光线上位于给定的交点时间处，交点的坐标由下面方法得到(见练习)：

```
Point3 rayPos(Ray& r, float t);      //返回光线在 t 时刻的位置
```

计算球体表面在交点处的法向也非常简单：因为球面上点的法向由球心指向该点，法向的坐标和该点的坐标一致。

练习

练习 12.5.1　把场景中的光线转换到通用坐标系中

hit()方法中调用 xfrmRay()方法来把光线变换到球体的通用坐标系中，请写出 xfrmRay()方法的实现细节。请特别注意对点做变换和对向量做变换的区别。

练习 12.5.2　rayPos()方法

请实现文中提到的 rayPos()方法，它的输入参数为一条光线和一个时间，返回为在给定时刻光线的位置。

练习 12.5.3　光线何时与球面相切

讨论当光线和球面相切时，我们应该计算和保存哪些值。

练习 12.5.4　关于计算的复杂度

光线和球体求交需要多少计算量？在光线和球体求交计算中，需要进行多少次乘法、除法和开根号运算？

12.5.2　一个完整的光线跟踪器

现在我们手上已经有足够的工具来为由球体和椭球体组成的场景构建一个简单的光线跟踪器。在处理更复杂的场景前，我们要先确认前面介绍的光线跟踪器的各个部分在整合

到一起后仍能正常工作。在这个过程中的每个部分都没有浪费,因为所有的工具都在以后用到。

到目前为止,如果要设置物体的颜色,我们只能设物体为发光物体(回顾第 8 章),在 SDL 文件中设置物体的发光颜色:emissive 0.3 0.6 0.2。

我们很容易修改图 12.12 中的方法 Scene::shade()让它仅处理发光物体的发射光线:只要删除 color.add()语句,该语句添加其他组成光线:

```
Color3 Scene::shade(Ray&ray)
{
    Color3 color;
    Intersection best;
    getFirstHit(ray, best);
    if(best.numHits==0) return background;
    Shape * myObj=(Shape * )best.hit[0].hitObject;    //相交物体
    color.set(myObj->mtrl.emissive);
    retrun color;
}
```

图 12.10 中方法 Camera::raytrace()唯一需要修改的部分是计算相应于每个像素块的光线的发射方向[①]。等式(12.2)可以帮助我们计算经过第 ik 个像素块的坐下角点的光线的相关参数。案例分析 12.1 将更详细的讨论一个球体场景的光线跟踪器实现细节。

12.6 光线和其他基本几何体求交

我们需要实现其他形状类的 hit()方法。所有的 hit()都是相似的:光线首先被变换到物体的通用坐标系,然后计算光线和通用物体的角点。我们只需要处理一下每个物体和光线求交时的计算细节。

12.6.1 与正方形求交

正方形是经常用到的通用物体之一。通用正方形落在 $z=0$ 的平面上,x 和 y 的范围都是 -1 到 1。它的隐式表达式为 $F(P)=P_z$,$|P_x|\leqslant 1$ 且 $|P_y|\leqslant 1$。它可以变换到空间中任何位置上的平行四边形,所以经常用来表示场景中的墙壁和窗户物体等。hit()首先计算光线是否和通用正方形相交(见等式(12.10)),然后再检测交点是否落在正方形内,如图 12.17 所示。

① 注意,为了提高效率,一些变量的计算可以从内循环中去除。

```
//<<<<<<<<<<<<<<<<<<<与正方形求交>>>>>>>>>>>>>>>>>>> .
bool Square::hit(Ray &r, Intersection& inter)
{
    Ray genRay;                                          //用于生成泛型光线
    inter.numHits=0;                                     //初始化
    xfrmRay(genRay, invTransf, r);
    double denom=genRay.dir.z;                           //分母
    if(fabs(denom)<0.0001) return false;                 //光线和平面平行：无交点
    double time=-genRay.start.z/denom;                   //相交时间
    if(time<=0.0) return false;                          //交点落在视点后方
    double hx=genRay.start.x+genRay.dir.x*time;          //交点的 x 坐标
    double hy=genRay.start.y+genRay.dir.y*time;          //交点的 y 坐标
    if((hx>1.0) || (hx<-1.0)) return false;              //x 不在范围内
    if((hy>1.0) || (hy<-1.0)) return false;              //y 不在范围内
    inter.numHits=1;                                     //有一个有效交点
    inter.hit[0].hitObject=this;
    inter.hit[0].hitTime=time;
    inter.hit[0].isEntering=true;
    inter.hit[0].surface=0;
    inter.hit[0].hitPoint.set(hx,hy,0);
    inter.hit[0].hitNormal.set(0,0,1);
    return true;
}
```

图 12.17　Square 类的 hit()函数

12.6.2　与锥形柱体求交

图 12.18 显示了一个通用锥形柱体和几条光线。该柱体的侧面是一个“无限长围墙”的一部分，它在 $z=0$ 处的半径为 1，在 $z=1$ 处有个小于 1 的半径 s。以后将“无限长围墙”简称为围墙。由等式(6.30)可知，锥形柱体的侧面的隐式形式为

$$F(x,y,z)=x^2+y^2-(1+(s-1)z)^2,\quad 0<z<1$$

如果 $s=1$，这个形状变成通用圆柱体；如果 $s=0$，则变成通用圆锥体。我们实现锥形柱体的 hit()方法，自然也就得到了圆柱体和圆锥体的 hit()方法。

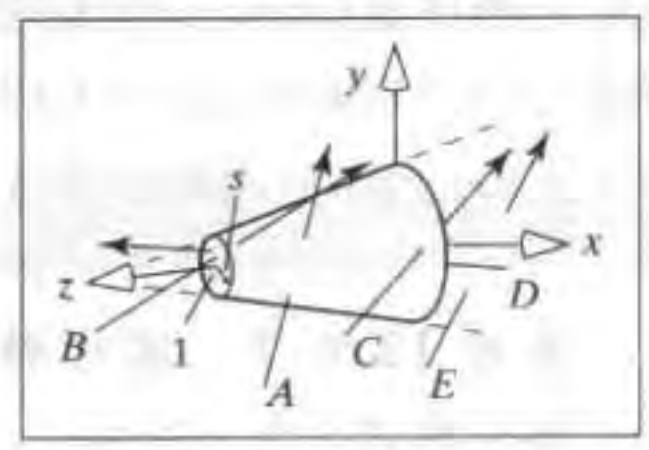


图 12.18　泛型圆台

图中还显示了几条光线，用来说明光线与物体相交或者不相交时的不同情况。光线 A 与柱体(简便起见，我们通常把锥形柱体简称为“柱体”)的侧面相交两次，而光线 B 先与围墙相交，然后从柱体的上底面进入柱体，最后从侧面穿出。光线 C 首先与围墙相交，然后从柱体的下底面穿出，而光线 D 从柱体的下底面进入，然后从上底面穿出。光线 E 虽然和围墙相交，但它处于柱体的外部，所以和柱体没有交点。图中没有显示与围墙完全分离的

光线。

由于存在如此多的情况,我们需要一个好的方法来避免多次的 if(),.else 检测。解决的办法是:按照任何一种方便的顺序来区分交点,而且将交点保存在 inter.hit[]列表中而不去考虑它们的顺序。最后一步,如果 inter.hit[]中有两次命中顺序颠倒,就交换这两项的位置。事实上这很容易做到,因为我们已经有一个列表来保存交点记录。

单个的相交关系测试是简单的。为了决定光线是否和围墙相交,我们只要把 $S+ct$ 代入锥形柱体的隐式形式,得到二次等式 $At^2+Bt+C=0$。容易知道,

$$\begin{aligned} A &= c_x^2 + c_y^2 - d^2 \\ B &= S_x c_x + S_y c_y - Fd \\ C &= S_x^2 + S_y^2 - F^2 \end{aligned} \tag{12.19}$$

其中 $d=(s-1)c_z$,且 $F=1+(s-1)S_z$。对于圆柱体和圆锥体,这些参数又变成怎样的形式?

我们使用的系数和球体情况时使用的系数形式相同。如果判别式 B^2-AC 的值为负,则光线在围墙外。如果判别式的值为非负,那么光线和围墙相交,且相交时间可以通过求解这个二次等式得到。为了检测光线是否和柱体相交,就要求出交点坐标的 z 分量。仅当 z 分量落在 0 和 1 之间时,光线和柱体的侧面相交。

为了检测光线和柱体下底面的相交情况,我们求解光线和平面 $z=0$ 的交点。假设交点坐标为$(x,y,0)$,那么当 $x^2+y^2<1$ 时交点落在下底面内。相似地,为了检测光线和上底面的交点,我们求解光线和平面 $z=1$ 的交点。假设交点坐标为$(x,y,1)$,那么当 $x^2+y^2<s^2$ 时交点落在上底面内。

一个柱体有多个表面,接下来我们想知道交点落在哪个表面上。例如,我们可能会给侧面和底面贴上不同的纹理。我们采用如下的编号方法:侧面编号为 0,下底面编号为 1,上底面编号为 2。然后,在相交记录的 surface 字段中保存相应的曲面编号。

将上述想法整合到一起并编写光线与锥形柱体的求交代码并不困难,但还是有些繁琐。处于简洁考虑,我们把细节留在练习里,要求读者补充完整。

还有一个信息我们需要得到,那就是交点处的法向量。如第 6 章中所述,可以简单得到柱体侧面在点(x,y,z)处的法向量为$(x,y,-(s-1)(1+(s-1)z))$。上下底面的法向量分别为$(0,0,1)$和$(0,0,-1)$。hit()将为 HitNormal 字段附上相应的值。

练习

练习 12.6.1 实现锥形柱体的 hit()方法

验证等式(12.19)中二次等式系数的正确性。将 TaperedCylinder::hit(Ray& ray, Intersection& inter)的代码补充完整。

练习 12.6.2 实现圆锥体的 hit()方法

如果你想编写一个专门用于 Cone 类的 hit()方法,那么它与 TaperedCylinder 类中的 hit()方法会有什么区别?说明如果要提高效率,可以修改哪几行代码。

练习 12.6.3 关于计算的复杂度

在求解光线与正方形或锥形柱体的交点时,分别需要进行多少次的加、减、乘、除运算?

12.6.3　与立方体(或任何凸多面体)求交

在多种图形环境中,凸多面体都被认为是非常有用的。我们也在本书中多处讨论到凸多面体的应用(例如第 6 章中对柏拉图立方体的讨论)。因为凸多面体是用它的边界平面来表示的,所以很容易实现光线与它求交的算法。我们将在本节的后面讨论这个问题。

通用立方体作为凸多面体的一个特例,应该引起我们的特别关注。它以原点为中心,顶点坐标为(±1,±1,±1),包含了所有+1 和−1 的所有 8 种组合。因此它的边与坐标轴平行,而且六个面落在图 12.19 所示的平面上。为了表述方便,我们给每个平面指定了一个名字。每个平面的名字根据它与观察者的位置关系指定,设观察者的位置在点(0,0,10),则名字为 top 的平面就位于立方体的上方,其他平面的位置以此类推。图中也给出了每个平面向外的法向量,以及平面中具有代表性的一个点 spot。

平面	名称	等式	向外法向	spot 点
0	top	y=1	(0,1,0)	(0,1,0)
1	bottom	y=−1	(0,−1,0)	(0,−1,0)
2	right	x=1	(1,0,0)	(1,0,0)
3	left	x=−1	(−1,0,0)	(−1,0,0)
4	front	z=1	(0,0,1)	(0,0,1)
5	back	z=−1	(0,0,−1)	(0,0,−1)

图 12.19　构成泛型立方体的六个平面

通用立方体之所以重要,有两个原因:

(1) 通过对立方体作用仿射变换,我们可以在场景中添加各种各样的有趣的"盒子"。然后在光线跟踪过程中,通过逆变换将每条光线变换到通用立方体的坐标系中,这样我们就可以利用光线与通用立方体的求交程序快速求出交点。

(2) 通用立方体可以作为场景中其他通用形状的**包围盒**:每个通用形状都内切于一个通用立方体。后面我们将会看到,在检测光线与物体的相交情况前先检测光线与物体包围盒的相交情况,将会大大提高程序的效率,特别是在光线与物体的求交计算很费时的情况下,提速的效果更加明显。如果光线与物体的包围盒不相交,那么光线必然与物体也不相交。因此,拥有一个光线与通用立方体的高效求交算法将会带来巨大的好处。

通用立方体的求交算法

在第 4.8.3 节中我们描述过的 Cyrus-Beck 算法,它处理 2D 的直线在凸窗口中的裁剪问题。光线与立方体的求交过程实质上就是 Cyrus-Beck 算法。该算法也可用于第 8 章中描述的直线与相机视景体的裁剪问题。它的基本思想是:立方体的每个平面把空间分为了"内"半空间和"外"半空间;光线上的一个点在立方体内,当且仅当它位于立方体的所有内半空间内。因此,光线与立方体的求交问题就转换为确定一个时间段的问题,在这个时间段内光线落在立方体所有平面的内侧。

设立方体为 P。对 P 的每个平面依次检测光线,计算光线进入或是离开平面内半空间

的时间。我们跟踪一个"候选"时间段 CI：该时间段基于目前为止的检测结果，在该时间段内光线有可能落在立方体内部。这个区间是 t_{in} 和 t_{out} 之间的部分，即 $CI=[t_{in},t_{out}]$。在时间段 CI 内，光线可能位于指定的半空间内；或者说，在时间段 CI 外，光线肯定不在半空间内。如果光线是进入指定的半空间且 $t<t_{in}$，或者光线是从内半空间中出来且 $t>t_{out}$，那么光线在半空间相关的平面的外侧。因为 P 的每个平面都要检测，所以我们对该区间进行"砍断"，或者是增大 t_{in} 或者是减小 t_{out}。在任何时候如果 CI 为空，那么光线必定与物体没有交点，提前结束光线与其他平面的相交检测。如果检测完 P 的所有平面后，CI 非空，那么说明光线在 t_{in} 时刻进入物体，在 t_{out} 时刻离开物体。

为了使这一过程更加形象，图 12.20 描述了一个光线在 $t=3.6$ 时进入通用立方体，在 $t=4.1$ 时离开。在这个例子中，检测结束后 CI 的结果为[3.6,4.1]，且光线确实和立方体相交。为了简便起见，图 12.21 给出了这一过程的一个 2D 版本，就是找出光线与正方形的交点。检测过程会因为立方体各个面的检测顺序不同而有所变化，但最终检测结果都是一样的。我们假设正方形的边的检测顺序与图中标的序号一致，由 1 号到 4 号。CI 初始化为 $CI=(-\infty,+\infty)$。光线从第一条边穿出的时间为 2.9，所以可知在 $t=2.9$ 以后光线必然不在正方形内，于是 CI 变成 $CI=(-\infty,2.9]$。检测第 2 条边可知，光线在更早的时候，$t=2.7$，就离开了正方形，所以 t_{out} 变为 2.7。检测第 3 条边可知，光线不可能在前进入正方形，所以 t_{in} 变成 1.6。最后检测第 4 条边，得到 t_{in} 为 1.8。于是，CI 的最终结果为[1.8,2.7]。

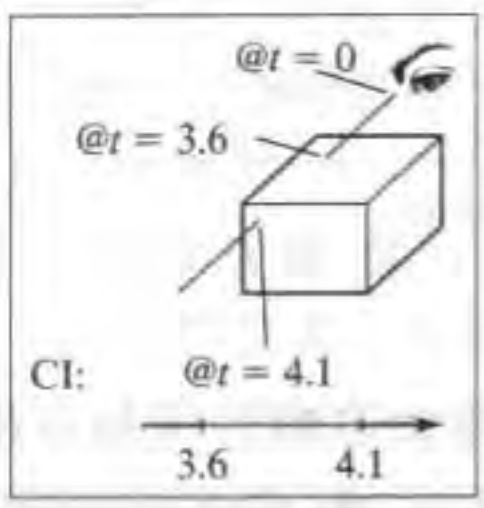


图 12.20 一条光线穿过泛型立方体

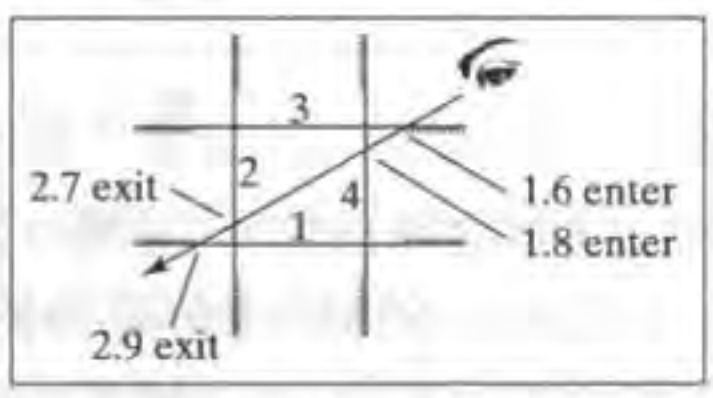


图 12.21 二维情况

在上述的每一步中，t_{in} 和 t_{out} 的值都是按照下面的伪代码进行调整：

```
初始化 tin为-∞，tout为∞ (对每个 CI)
if(光线在 thit进入)
        tin=max (tin, thit)
else if(光线在 thit离开)
        tout=min (tout, thit)
```

进一步具体化，令光线为 $S+\mathbf{c}t$，待检测的平面经过点 B 且具有向外的法向量 $\mathbf{m}$。该平面的隐式形式为 $F(P)=\mathbf{m}\cdot(P-B)$，所以当光线和平面相交时 t 满足 $\mathbf{m}\cdot(S+\mathbf{c}t-B)=0$，可得

$$t=\frac{numer}{denom}$$

其中

$$\begin{aligned} numer &= \mathbf{m}\cdot(B-S) \\ denom &= \mathbf{m}\cdot\mathbf{c} \end{aligned} \tag{12.20}$$

如果 $denom>0$，说明光线进入平面的外半空间（因为此时 **m** 和 **c** 的夹角小于 90°）；如果 $denom<0$，说明光线进入平面的内半空间。如果 $denom=0$，说明光线与平面平行，此时 *numer* 的值决定了光线是完全位于内半空间还是外半空间，正如第 4.8.3 节所述。图 12.22 归纳了所有这 4 种情况。

位置	条件
进入内半空间	$denom<0$
进入外半空间	$denom>0$
完全位于内半空间	$denom=0$，$numer>0$
完全位于外半空间	$denom=0$，$numer<0$

图 12.22　光线与平面内半空间的相交情况

所有这些思想对 3D 和 2D 情况都适用。对于通用立方体，*numer* 和 *denom* 的值可以很快计算得到：其中的点积计算得到很大地简化，因为在立方体的通用坐标系中，向量 **m** 有两个分量都是 0。图 12.20 中的平面产生了下列值（请检验它们）：

平面	*number*	*denom*
0	$1-S_y$	c_y
1	$1+S_y$	$-c_y$
2	$1-S_x$	c_x
3	$1+S_x$	$-c_x$
4	$1-S_z$	c_z
5	$1+S_z$	$-c_z$

图 12.23 给出了 Cube 类的 hit()方法的伪代码，它实现了光线和立方体的交点检测算法。和其他的 hit()方法一样，光线先被变换到物体的通用坐标系中，然后所有的检测都在光线和通用物体间进行。在一个循环语句中逐个检测六个平面，并适当改变 numer 和 denom 的值。CI 由最初的（$-\infty$，$+\infty$）砍断为最终的（tIn，tOut），除非在某次检测后它为空，此时说明光线和立方体没有交点。因为确定哪个平面和当前的 tIn 和 tOut 相关联是非常重要的，所以我们把这个信息保存在变量 SurfIn 和 SurfOut 中。

当所有的平面都经过测试，就可以确定相交时间 tIn 和 tOut。如果 tIn 为正，那么就将 tIn 时刻的交点信息加载到 inter.hit[0]中，将 tOut 时刻的交点信息加载到 inter.hit[1]中。如果只有 tOut 为正，就将该时刻的交点信息加载到 inter.hit[0]中。

与光线相交的平面的法向量可以通过一个辅助方法 cubeNormal(i)得到，该方法返回如图 12.20 所示的平面的向外法向量。例如 cubeNormal(0)返回(0,1,0)，cubeNormal(3)返回（−1,0,0）。参见本节后面的练习。

任意凸多面体的求交算法

拓展 hit()方法来适合任意凸多面体是很直接的。假设凸多面体有 N 个面，第 i 个平面包含点 B_i，且具有（向外）法向量 $\mathbf{m}_i$。除了利用等式(12.20)计算 *numer* 和 *denom* 外，立方体的 hit()基本不需要做任何修改。此时，需要两个费时的点积计算，而 for 循环变为：

```
bool Cube::hit(Ray& r, Intersection& inter)
{
    double tHit, numer, denom;
    double tIn=-100000.0, tOut=100000.0;        //正负无穷大
    Ray genRay;
    int inSurf, outSurf;                        //6个面中的哪个面
    xfrmRay(genRay, invTransf, r);
    for (int i=0; i<6; i++)
    {
        switch(i)                               //待检测的平面
        {
        case 0: numer=1.0-genRay.start.y; denom=genRay.dir.y; break;
            case 1, case 2, case 3, case 4 similarly
        case 5: numer=1.0+genRay.start.z; denom=-genRay.dir.z; break;
        }
        if(fabs(denom)<0.00001)                 //光线和平面平行
        {
            if(numer<0) return false;           //光线在外部
            else;                               //光线在内部,不改变 tIn 和 tOut 的值
        }
        else                                    //光线不和平面平行
        {
            tHit=numer/denom;
            if(denom>0){                        //射出
                if(tHit<tOut){                  //一个更早的射出时刻
                    tOut=tHit; outSurf=i;
                }
            }
            else {                              //denom 的值为负:进入
                if(tHit>tIn){                   //一个更迟的进入时刻
                    tIn=tHit; inSurf=i;
                }
            }
        }
        if(tIn>=tOut) return false;             // 不相交-提早结束
    }                                           //for 循环语句结束
    int num=0;                                  //仍未有有效的交点
    if(tIn>0.00001)                             //第一次相交在视点前方吗?
    {
        inter.hit[0].hitTime=tIn;
        inter.hit[0].surface=inSurf;
        inter.hit[0].isEntering=1;              //进入
        inter.hit[0].hitObject=this;
        inter.hit[num].hitPoint.set(rayPos
                (genRay.start,genRay.dir,tOut));
        inter.hit[num].hitNormal.set(cubeNormal(outSurf));
        num++;
    }
    inter.numHits=num;                          //在视点前方的交点个数
    return (num>0);
}
```

图 12.23 Cube 类的 hit()方法

```
for (int i=0; i<N; i++)     //对多面体的每个面
{
    numer=dot3(m_i, B_i-S);
    denom=dot3(m_i, c);
    if (fabs(denom)<eps)… 和以前一样
    … 同前 …
}
```

网格物体的求交算法

从第 6 章开始，我们就使用了各种各样的网格物体，而且已经有了一个可以用于任何网格物体的 drawOpenGL()方法，因此自然想到对网格物体进行光线跟踪。我们利用前面介绍的方法实现一个网格物体的 hit()方法。回忆前面的内容，我们用面列表描述 Mesh 对象，每个面又是一些顶点以及顶点处的法向量组成的列表。我们依次取网格的每个面，每个平面都是一个有界平面，并称该平面所在的无界平面为"面平面"(face plane)。因此，我们要跟踪的网格物体的形状就是所有"面平面"的内半空间的交。

图 12.24 给出了 2D 平面上的两个形状，并标记了它们的面平面(直线)。图 12.24(a)中的图形具有凸形状，所以它的面平面和它的边界平面一致。这个图形将会被正确的光线跟踪。图 12.24(b)中的图形具有一个非凸形状，我们用阴影标记了包含在所有面平面内半空间的区域，该区域是光线跟踪器将会绘制的形状。因此，对于非凸形状的物体，光线跟踪器利用 Cyrus Beck 算法将会得到一个错误的结果，Cyrus Beck 算法只适用于具有凸形状的网格物体。

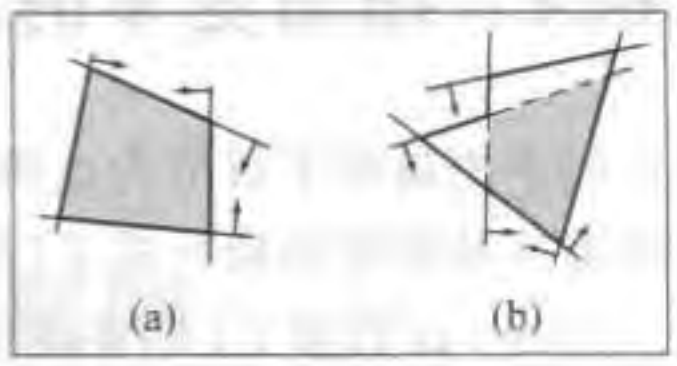


图 12.24　2D 的物体以及它的面平面
(a) 凸形状；(b) 非凸形状

为了实现网格对象的 hit()方法，我们依次为每个面设置两个向量 numer 和 denom，并把面的第 0 个顶点 pt[face[f].vert[0].vertIndex]设置为面的代表点，该点的法向量与面平面的法向量 norm[face[f].vert[0].normIndex]一致。

练习

练习 12.6.4　设置立方体交点处的法向量

编写方法 Vector3 cubeNormal(int which)，该方法返回立方体的面 which 的向外法向量，如图 12.20 所示。最具可读性的代码可能会是使用一个具有 6 种判断条件的 switch 语句，然后将向量设置为 6 个已知法向量中的一个。稍微简单一点的判断并设置法向量的方法如下所示：

```
int m=which/2, n=(which%2)?-1:1;
if(m==0) v.set(0, n, 0);
else if(m==1) v.set(n, 0, 0);
else v.set(0, 0, n);
```

说明这种方法是如何运作的，并利用此方法重新编写方法 cubeNormal()。

练习 12.6.5 四面体的面

根据图 6.27,说明正四面体的面列表中包含的内容。

练习 12.6.6 修改 hit(),使之适合于凸多面体

补充光线与凸多面体求交的 ConvexPolyhedron :: hit ( ) 方法的细节。关于 ConvexPolyhedron 类的细节,请参考本书相关网站上的关于 Shapes 类的说明。

练习 12.6.7 与四面体相交

对于顶点为(1, 0, 0),(−1, 0, −1),(−1, 0, 1)和(0, 1, 0)的四面体,请确定光线(0, 0, 0)+(1, 2, 3)t 在其内部的时间段。

练习 12.6.8 关于计算的复杂度

在对凸多面体的每个面进行交点检测时,需要进行多少次的乘法/除法运算?有时可能只检测了几个平面后,就因为 tIn 超过了 tOut 而提前结束检测。试问,当随机选择一条光线进行相交检测时,平均需要检测多少个平面?试将此计算的复杂度与光线和球体的求交计算的复杂度作比较。

12.6.4 添加更多的基本形状

人们不用局限于目前为止所考虑的通用形状,而可以涉及一些其他类型的形状。我们只要知道这些形状的隐形式 $F(P)$,然后就可以像以前那样,确定光线 $S+\mathbf{c}t$ 在何处与表面相交。用 $S+\mathbf{c}t$ 代替 $F(P)$中的 P,形成了以下对时间 t 的函数:

$$d(t) = F(S + \mathbf{c}t) \tag{12.21}$$

该函数具有以下性质:

- 当光线上的点在物体的外部,函数值为正;
- 当光线上的点在物体的表面上,函数值为零;
- 当光线上的点在物体的内部,函数值为负。

在计算交点的时候,我们寻找满足 $d(t)=0$ 的 t 值,所以与一条光线求交时相当于求解这个等式。对于球面以及其他二次曲面,我们已经知道这个等式是简单的二次等式。

但对于圆环面就不同了。通用圆环面具有如下隐式形式

$$F(P) = \left(\sqrt{P_x^2 + P_y^2} - d\right)^2 + P_z^2 - 1 \tag{12.22}$$

所以等式 $d(t)=0$ 是四次的,这使得求解比较困难,虽然可以在很多数学手册中找到它的显式解。

更一般地,我们考虑函数 $d(t)$的曲线形状。如果光线的起点 S 位于物体的外部,并且发射方向指向物体,那么 $d(t)$的形状与图 12.25 中的形状类似。

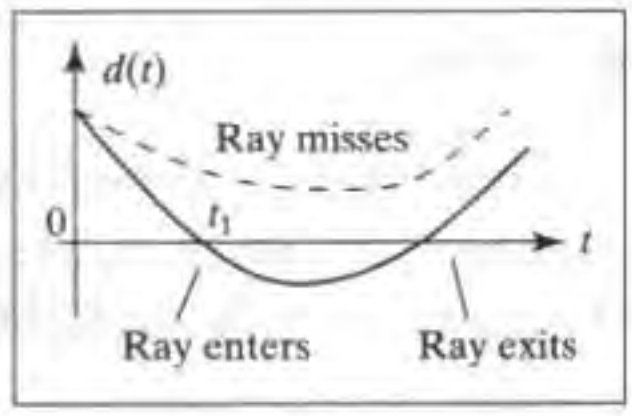


图 12.25 对于给定光线和物体的 $d(t)$函数的形状

因为光线的起点 S 位于物体的外部,所以 $d(0)$的值为正。当光线靠近物体时,$d(t)$的值逐渐减小,并且 $d(t)=0$ 时光线到达物体表面。然后一段时间光线位于物体内部,且 $d(t)<0$。当光线从物体中出来,$d(t)$经过零点,然后逐渐增

大。对圆环这样的物体，光线可能再次进入物体内部（请勾画一个具有这样性质的物体）。相反地，如果光线和物体没有交点（图中用虚线表示的情况），那么 $d(t)$ 减小一段时间后，在未到达零之前，就会一致增大下去。

对于像球面这样的二次曲面，$d(t)$ 是抛物线形状；对于圆环，$d(t)$ 是一个四次曲线形状。对于其他曲面，$d(t)$ 的形状可能会非常复杂，此时就需要利用数值计算的方法来求解等式 $d(t)=0$。

为了找到满足 $d(t)=0$ 的 t 的最小正值，我们需要逐步求解 $d(t)$ 的值，使 $d(t)$ 不断逼近于零。很多数值方法，如牛顿迭代法【Acton70，Conte 和 deBoor80】，都致力于得到更优的 t 值，但是这些方法都需要多次迭代计算，因此也大大降低了光线跟踪的速度。

人们还开发出了一些其他类别物体的光线跟踪方法，特别是分形曲面、旋转曲面和拟圆柱面【Kajiya83】。参见案例分析中对这些物体进行光线跟踪的方法。寻找由超大数量的物体组成的场景的高效光线跟踪算法，仍然是当前图形学研究中的一个课题。

练习

练习 12.6.9　与圆环相交

根据数学手册，确定如何得到一个四次等式的解，并设计必要的步骤来求出光线 $S+ct$ 与圆环等式(12.22)的交点。

12.7　绘制场景的明暗图

设计一个发光物体的光线跟踪器是开始学习光线跟踪算法的好方法，但是当要绘制一幅具有强烈真实感的场景时，你会发现这样的光线跟踪器是远远不够的。为了产生逼真的图像，我们必须确定在交点处反射到人眼的光线的性质。为了推导出整个计算过程，我们将回顾一下第 8 章中讨论的 OpenGL 明暗着色模型，然后讨论如何改进现在的模型以得到更逼真的效果。

OpenGL 在绘制场景时考虑了顶点处的环境光线、漫反射光线和镜面反射光线，并利用 Gouraud 模型计算得到在曲面上光滑变化的漫反射光线。图 12.26 归纳了光线的主要成分。图中，第 rc 条光线与一个椭球体在 P_h 点处相交，光源设在 L 处。我们希望计算有多少光在 P_h 点处返回到眼睛里。计算将涉及到的几个主要向量为 **s**、**v** 和 **m**。向量 **s** 指向光源，所以可以由 $\mathbf{s}=L-P_h$ 式得到；向量 **v** 指向观察者，所以与光线的方向 **dir** 相反；**m** 是 P_h 点在表面上的法向量。

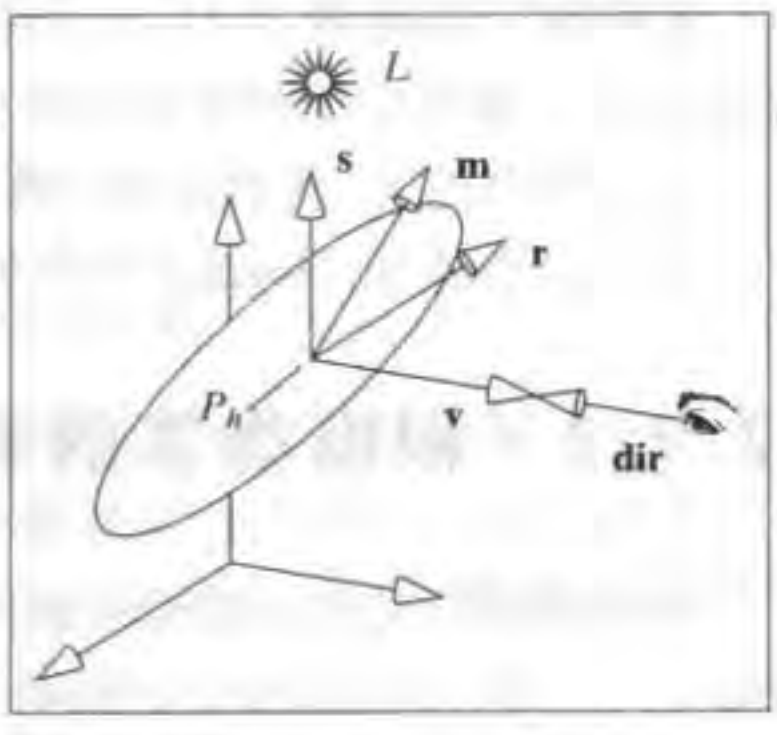


图 12.26　明暗着色模型

正如第 8 章中所述，眼睛观察到的光的总强度由环境光、漫反射光和镜面反射光三部分组成，即

$$I = I_a\rho_a + I_d\rho_d \times lambert + I_{sp}\rho_s \times phong^f \tag{12.23}$$

其中,我们定义

$$lambert = \max\left(0, \frac{\mathbf{s}\cdot\mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right) \quad 和 \quad phong = \max\left(0, \frac{\mathbf{h}\cdot\mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right) \tag{12.24}$$

I_a 是环境光强度,I_{sp} 是光源强度。ρ_a、ρ_b 和 ρ_s 分别是环境光、漫反射光和镜面反射光的反射系数。向量 **h** 是由 **h**=**s**+**v** 定义的中间向量。

如果场景中有多个光源,那么每个光源都会产生漫反射光和镜面反射光线。这些光线都将添加到等式(12.23)中。回顾环境光的定义,它实际上是场景中相邻物体间多重反射光的近似合成。后面将看到,当一个物体足够亮时,就必须跟踪它的经过多次反射后的光线。场景中也有些光线数量多强度弱,很难分别对它们进行光线跟踪,I 中环境光的部分有助于模拟这些光线的效果。

和第 8 章中的做法一样,我们把这个等式推广到彩色光源和物体上。此时,到达眼睛的光线包含红、绿、蓝三种成分,它们分别由下式给出:

$$\begin{aligned} I_r &= I_{ar}\rho_{ar} + I_{dr}\rho_{dr} \times lambert + I_{spr}\rho_{sr} \times phong^f \\ I_g &= I_{ag}\rho_{ag} + I_{dg}\rho_{dg} \times lambert + I_{spg}\rho_{sg} \times phong^f \\ I_b &= I_{ab}\rho_{ab} + I_{db}\rho_{db} \times lambert + I_{spb}\rho_{sb} \times phong^f \end{aligned} \tag{12.25}$$

其中每组 I 和 ρ 都是相应颜色成分的强度和反射系数。值得注意的是,这 3 种颜色成分使用了同样的项 *lambert* 和 *phong*(以及同样的向量 **s**、**v** 和 **m**),因此对一条光线只需要跟踪一次,而 *lambert* 和 *phong* 也只需要计算一次。

明暗着色计算要求知道交点 P_h 的坐标以及向量 **s**、**v** 和 **m**。下面讨论如何计算 **m**,这将利用到一些信息,这些信息都已经保存在相交记录中。

12.7.1 计算交点处的法向量

向量 **m** 是物体表面在交点处的法向量,如何确定它呢?首先我们在通用坐标系下可以相对容易计算它,然后再把它变换到世界坐标系中。在第 6.5.3 节中,我们曾说明如果物体经过矩阵 M 变换到另一个物体,那么法向量 $\mathbf{m}'$ 将会变换成法向量

$$\mathbf{m} = M^{-T}\mathbf{m}' \tag{12.26}$$

其中 M^{-T} 是矩阵的 M 的逆转置矩阵。于是,通过计算通用物体在交点处的法向量,然后乘以 M^{-T} 就可以得到要求的法向量。

令人高兴的是,我们已经计算并保存了通用坐标系中交点处的法向量;该法向量信息保存在相交记录的 hitNormal 字段中。

12.7.2 根据物体的表面材质为其着色

我们看到的交点处的颜色取决于环境光、漫反射光和镜面反射光成分的合成。反射系数如 ρ_{ar}、ρ_{dg} 等保存在相交记录的 mtrl. ambient、mtrl. diffuse 和 mtrl. specular 字段中。当首次读入 SDL 文件时,这些字段将被加载到内存里。

在实践中一个普遍的做法是将 Phong 镜面反射光设置为与光源一样的颜色。这样只

要把反射系数 mtrl. specular 的红、绿、蓝三个分量设置为相等，例如(0.9，0.9，0.9)，那么用此系数乘以光源的颜色得到的仍然是相同的颜色，只是亮度减弱。这样的设置使物体看起来像有光泽的塑料。

在光线跟踪器中增加环境光、漫反射光和镜面反射光成分并不困难。图 12.27 用伪代码描述了如何修改图 12.12 中的 shade()方法。

```
Color3 Scene::shade(Ray& r)
{
    调用 getFirstHit(r, best)得到第一个交点;
    生成副本 h=best.hit[0];                          //第一次相交记录
    根据 h.hitTime 计算 hitPoint
        Form v=-ray.dir;                             //向着观察者方向
    v.normalize();
    Shape* myObj=(Shape*)h.hitObject;                //指向相交物体
    Color3 color(myObj->mtrl.emissive)①;             //从发光部分开始
    color.add(ambient contribution);                 //计算环境光颜色
    Vector3 normal;
    //将一般法向量转换为世界法向量
    xfrmNormal(normal, myObj->invTransf, h.hitNormal);
    normal.normalize();                              //归一化
    for(each light source, L)                        //累加所有光源
    {
        if(isInShadow(…)) continue;      //如果在阴影中,跳到下一个光源
        Form s=L.pos-hitPoint;                       //由交点指向光源的向量
        s.normalize();
        float mDotS=s.dot(normal);                   //Lambert 系数
        if(mDotS>0.0)                                //交点面向光源
            Form diffuseColor=mDotS*myObj->mtrl.diffuse *L.color;
        color.add(diffuseColor);                     //添加漫反射光部分
        Form h=v+s;                                  //中间向量
        h.normalize();
        float mDotH=h.dot(normal);                   //phong 系数
        if(mDotH<=0)continue;                        //无镜面发射光
        float phong=pow(mDotH, myObj->mtrl.specularExponent);
        specColor=phong*myObj->mtrl.specular*L.color;
        color.add(specColor);
    }
    return color;
}
```

图 12.27　伪代码，在 shade()中添加明暗计算

检查 getFirstHit()方法中得到的最佳相交记录 best，得到光线与物体的第一个交点信息。复制 best. hit[0]到 HitInfo 记录 h 中，交点 hitPoint 的坐标可以利用光线等式和时间 h. hitTime 得到。通用坐标系中交点的法向量保存在 h. hitNormal 中，然后利用等式

① 我们假设 Vector3 类和 Color3 类已经具有了一些简单的构造函数：Vector3 s(A,B) 构造由点 A 指向点 B 的向量；Color3 col(f, d, c)构造一个颜色，它的红色部分为 f * d. red * c. red，蓝、绿部分类似。

(12.26)变换到世界坐标系中。接着,沿着光线返回的各种颜色成分保存在 color 中。这样就求出了放射光线和环境光线部分,同时对于该光源,求出它们所产生的漫反射光和镜面反射光成分并将其添加到 color 中。如果光源在交点处被某个对象遮住,就不用考虑该光源发出发出的光线。在第 12.10 节中我们曾讨论过阴影效果。现在只要删除其中"if (isInshadow (…)) continue;"这一行,就可以不进行任何阴影测试。

根据等式(12.23)确定镜面反射光部分,这要求计算向量 **v** 和 **h** 并利用保存在 myObj->mtrl.specular 中的镜面反射系数(和指数 f)。

图 12.28 显示了一个场景,其中的物体已经用环境光、漫反射光和镜面反射光进行了渲染,但是没有计算阴影。镜面高光使物体看起来像是由有光泽的塑料制成的。

图 12.28 综合环境光、漫反射光和镜面反射光渲染场景

练习

练习 12.7.1 变换法向量

编写一个函数 void xfrmNormal(Vector3& res, Affine4& aff, Vector3& v),用来计算向量 v 和矩阵 aff 的转置的乘积,并把结果保存在 res 中。在光线跟踪器中,我们会调用函数 xfrmNormal(normal, myObj->invTransf, h.hitNormal)。也就是说,该函数利用保存在相交对象中的逆矩阵 myObj->invTransf,对通用坐标系中的法向量 h.hitNormal 进行变换,最后得到世界坐标系中的法向量。

12.7.3 基于物理的明暗着色模型—Cook Torrance 明暗处理

Phong 高光很容易产生,但是总使物体看上去像有光泽的塑料。如果使用 OpenGL 为场景明暗着色,就必须解决 Phong 高光问题。但是如果通过光线跟踪算法渲染场景,就有了多种选择,并且在计算镜面反射高光时可以考虑采用不同的算法。

为了追求更加逼真的视觉效果,许多研究人员已经开发出了更精确的明暗着色模型。这些模型从物理原理出发,分析了光如何从一个实际表面反射,并建立一些数学表达式来表示到达人眼的反射光的强度和颜色。这些模型主要关心物体表面上光能的"守恒":入射光的一部分转化为热能被材料吸收;一部分与表面发生相互作用,变成漫反射光;还有一部分则转化为镜面反射光从表面反射出去。不同材料对入射光的分裂作用是不同的:例如,粗糙的表面比光滑、有光泽的表面产生的漫反射光更多,而镜面反射光更少。

在 Torrance 和 Sparrow【Torr67】以及 Trowbridge 和 Reitz【Trowbridge75】的早期工作中,粗糙表面被概念化地模拟为许多"微平面"(microfacet)的集合,这些微平面有着不同的方向,如图 12.30 所示。入射光线与平均法线方向 **m** 的夹角为 ϕ,当入射光到达不同的微平面时,就会向不同的方向反射出去,其中有一部分光线反射到人眼。

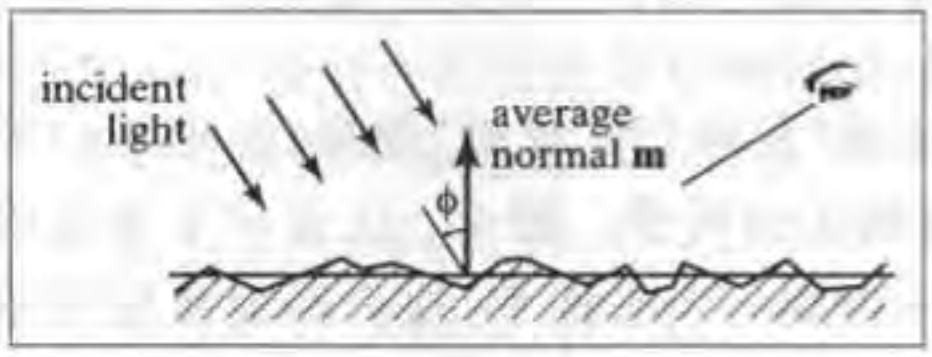


图 12.29　粗糙表面模拟为随机取向的微切面的集合

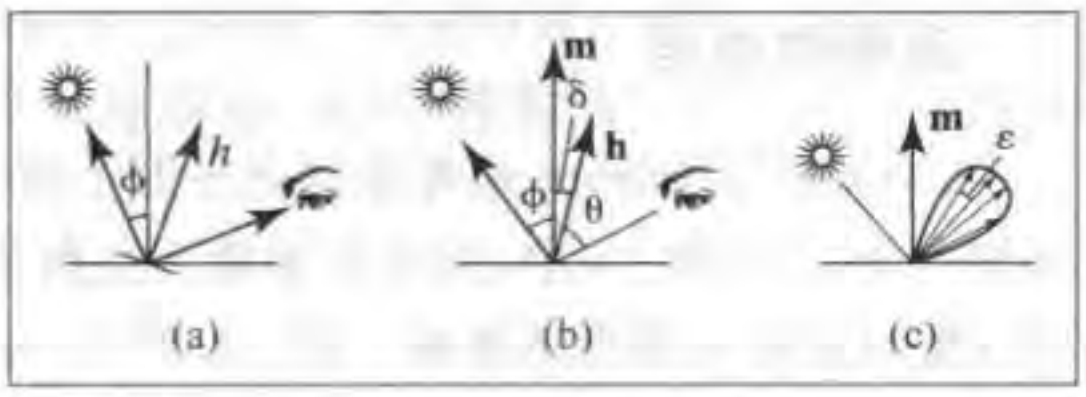


图 12.30　确定方向为 δ 角的微平面所占的比例

Blinn【Blinn77】采用这个模型开发了一种适用于计算机图形学应用的算法，并显示了该算法产生的高光效果与 Phong 高光有很大的不同。Cook 和 Torrance【Cook82】发展和改进了 Blinn 的模型，在反射高光中引入了“颜色偏移”(color shift)，这样就与真实材料上的光反射效果更加匹配。

下面我们介绍 Cook-Torrance 模型的主要部分，并说明如何将它应用于光线跟踪器中。讨论这个模型的 3 个基本方面以及它们对镜面反射光数量的影响。每个方面都试图通过建立相应的算法来准确表达光反射所涉及的物理现象。

微平面法向量的分布

Cook-Torrance 模型假设每个微平面都相当于一个理想的镜面，而且只有那些方向恰好吻合的平面才对特定方向上的反射光线有贡献。如图 12.31(a)所示，只有那些以 $\mathbf{h}=\mathbf{s}+\mathbf{v}$ 为法方向的平面才对在方向 $\mathbf{v}$ 上见到的光有贡献(方向 $\mathbf{v}$ 指向观察者)。因此，有必要知道哪些微平面具有此法方向。人们已经对某些特定材料制成的表面上的微平面法方向的分布情况进行了统计研究。研究得到了一个分布函数 $D(\delta)$，它反映了法方向和曲面的法向量成 δ 角的微平面在所有微平面中占的比例。如图 12.30(b)所示，如果入射角为 ϕ，观察者位于偏离微平面法向量 θ 角的位置，那么只有微平面的法方向和曲面法向 $\mathbf{m}$ 成角度 $\delta=(\theta-\phi)/2$ 时，微平面才能将入射光线反射到观察者(参见练习)。具有此法方向的微平面所占的比例为 $D(\delta)$。

人们已经发现了多种分布函数，Cook 和 Torrance 采用了 Beckmann 分布函数【Beckmann63】：

$$D(\delta)=\frac{1}{4m^2\cos^4(\delta)}e^{-\left(\frac{\tan(\delta)}{m}\right)^2} \tag{12.27}$$

其中参数 m 度量曲面的粗糙程度。特别地，可以取 m 为微平面斜率的均方根。对于相对光滑的表面 m 的值大约为 0.2；而对于粗糙的表面，m 的值大约为 0.6。实验表明，Beckmann 分布是很多粗糙物体表面很好的数学近似。$D(\delta)$在 $\delta=0$ 处有最大值(最大值是多少?)，随着 δ 远离 0，$D(\delta)$的函数值也逐渐减小；m 的值越小，$D(\delta)$的函数值下降越快。

Cook 和 Torrance 将 $D(\delta)$作为不同方向上镜面反射作用的比例因子，其中 δ 为 $\mathbf{h}$ 和 $\mathbf{m}$ 之间的夹角。图 12.30(c)显示了 $m=0.3$ 时反射光强度随观察方向的变化。图中给出的是与图 8.12 类似的“光束方向图”。每个方向上反射光的相对强度用向量的长度表示。在全反射方向上(角 ϕ 处)反射光强度最大，这是因为该平面的法线最可能与 $\mathbf{m}$ 平行。如果偏离了这个理想镜面的方向(例如偏离了 ε)，反射光强度就变小了，这是因为该平面的法线方向偏离向量 $\mathbf{m}$ 角 ε 后，分布函数在此处的值也变小，于是只有较少的平面具有此法线方向(参见本节后面的练习)。

阴影和遮挡

Cook 和 Torrance 还考虑到微平面上将会产生的"阴影"和"遮挡"效果,并把它们都归结到一个"几何项"G 中,用来作为镜面反射光强度的比例因子。图 12.31 显示了平面的几何位置对反射光强度的影响。图 12.31(a)中,光入射到一个平面上,这个平面所处的角度使之能够完全被照亮而且所有的反射光都能从该平面上"逃逸"出去。这种情况下,G 等于 1。图 12.31(b)中,反射光离开平面时,有一部分被相邻平面遮挡住了,此时相应地减小了 G 的值。而在图 12.31(c)中,平面只有一部分被照亮,部分入射光被相邻平面遮挡住了而在平面上产生阴影,此时 G 小于 1。

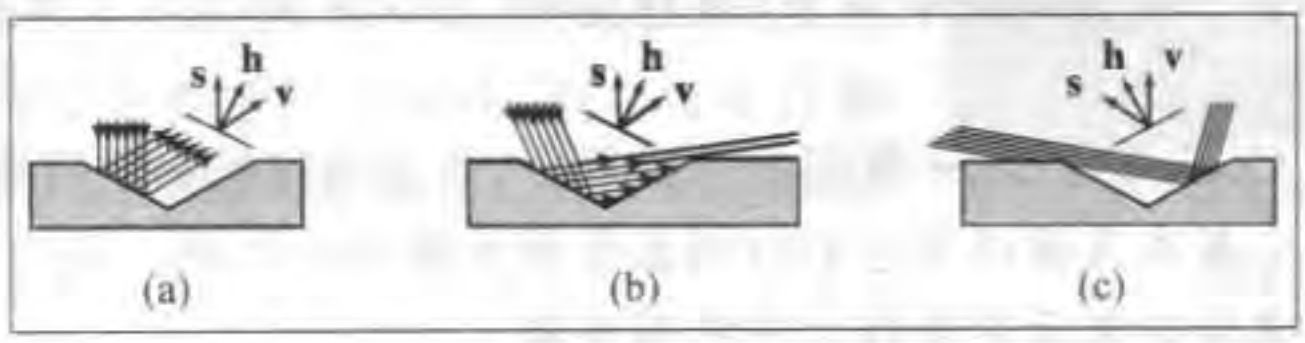


图 12.31 微平面上光的遮挡和阴影

Blinn 引进了几何因子来表示这些几何现象,在这些等式中只有涉及到简单的点积的运算。几何因子取 1、G_s 和 G_m 三者中的最小值:

$$G = \min(1, G_m, G_s) \tag{12.28}$$

其中

$$G_m = \frac{2(\mathbf{m}\cdot\mathbf{h})(\mathbf{m}\cdot\mathbf{s})}{\mathbf{h}\cdot\mathbf{s}}, \quad G_s = \frac{2(\mathbf{m}\cdot\mathbf{h})(\mathbf{m}\cdot\mathbf{v})}{\mathbf{h}\cdot\mathbf{s}} \tag{12.29}$$

分别表示没有被遮挡住的那部分反射光和没有被遮挡住的那部分入射光。

菲涅耳(Fresnel)系数

在 Torrance-Sparrow 模型中的第三个因素也可以影响镜面反射光的数量。微平面并不是理想的镜面,但是它像实际材料那样,只反射部分的光线,其余部分则被表面吸收。被反射的那部分光由菲涅耳系数[①] $F(\phi,\eta)$决定,其中 ϕ 是图 12.30 所示的光线的入射角($\mathbf{m}$ 和 $\mathbf{s}$ 之间的夹角),η 是材料的折射率。在后面的章节 12.11 中,当光线跟踪透明材料[②]时将会用到折射率,但是对折射率的使用其实是从金属这样的不透明材料开始的。

对于折射率为 η 的表面,菲涅耳系数 $F(\phi,\eta)$可以由电磁反射第一定律推导出来。计算等式为

$$F = \frac{1}{2}\frac{(g-c)^2}{(g+c)^2}\left\{1+\left(\frac{c(g+c)-1}{c(g-c)+1}\right)^2\right\} \tag{12.30}$$

其中 $c=\cos(\phi)=\mathbf{m}\cdot\mathbf{s}$, $g^2=\eta^2+c^2-1$。注意,中间项 g 由材料的折射率决定。对于许多材料来说,折射率随光波波长的变化非常复杂,这就使得 F 与光波波长的关系也非常复杂。

---

① 菲涅耳(Augustin Jean Fresnel, 1788—1827),法国物理学家,提出很多关于光反射、折射以及偏振现象的基本理论。

② 材料的折射率是光在真空中的传播速度与光在材料中传播速度的比值。它一般和光的波长有关系。在本章后面我们会介绍光的波长和它颜色之间的关系。

图 12.33 显示了对于不同的 η 值，F 相对 ϕ 的变化曲线。从图中可以看出，菲涅耳系数在“垂直入射”处($\phi=0$)的值与材料相关，然后平滑地变化，直到“掠射角”($\phi=\pi/2$)处到达 1.0。菲涅耳系数不会超过 1(为什么?)

- 在垂直入射时，$F(0,\eta)<1$；
- 在掠射时，$F(\pi/2,\eta)=1$。

Torrance 和 Sparrow【Torrance67】在微平面模型中综合考虑了上述因素，得到了一个基于物理原理的镜面反射模型。他们把镜面反射光的强度定义为 F、D 和 G 这 3 项的乘积与 $\mathbf{m}\cdot\mathbf{v}$ 的商，即

$$spec = \frac{F(\phi,\eta)DG}{(\mathbf{m}\cdot\mathbf{v})} \tag{12.31}$$

其中的分母 $\mathbf{m}\cdot\mathbf{v}$(就是 $\cos(\theta)$，参见图 12.31(b))是根据下面的论述推导出来的：以掠射角(θ 大余弦值就小)观察表面与以一个小角 θ 观察表面相比，在给定的立体角范围内看到更多的微平面。看见的微平面的数量越多，产生的光也越多。因此 $\mathbf{m}\cdot\mathbf{v}$ 的值越小，得到光的强度也越大。

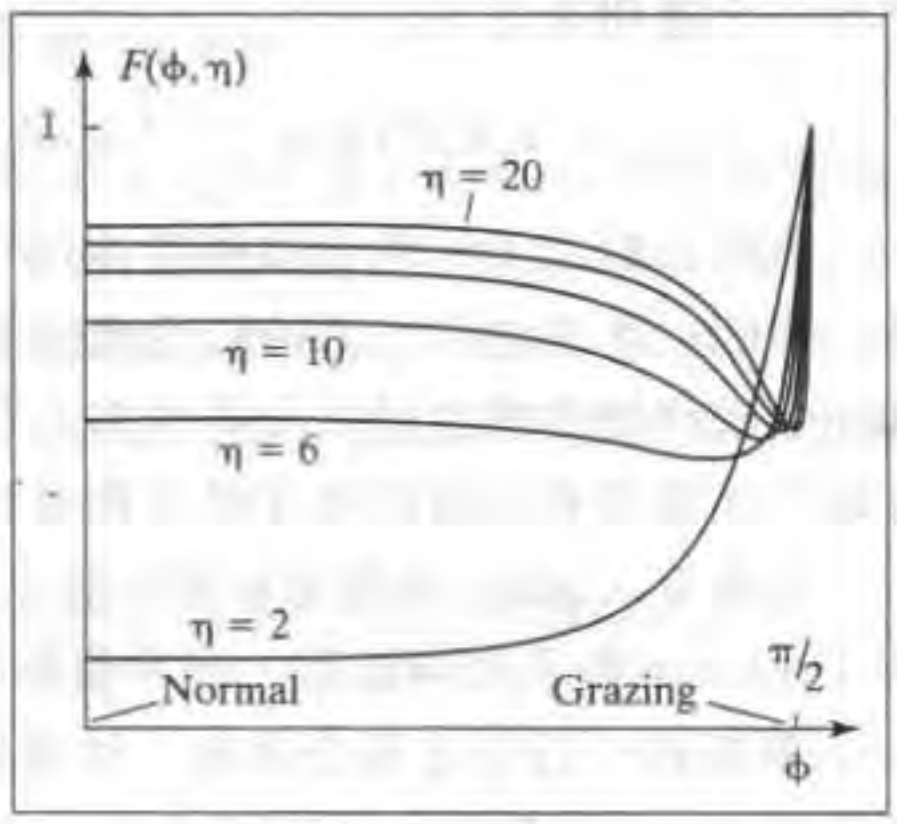


图 12.32　菲涅耳系数与入射角的关系

Blinn【Blinn77】对这个模型做了适当的改进以适合于计算机图形学的应用，并将它与 Phong 高光做了比较。尤其令人感兴趣的是，以不同的入射角 θ 得到的镜面反射光的作用方法也不同。根据 Phong 模型，镜面反射高光在“理想射入”方向上的光强总是最强的，而当视角稍微偏离这个方向之后，反射高光就会骤然下降。图 12.33(a)显示了“光束方向图”，描述了 Phong 高光的光强如何随着角度的变化而减弱。图中显示了两种入射角的光束方位图，无论入射角 ϕ 的大小如何，光束方向图的形状总是一样的。微平面模型的光束方向图见图 12.33(b)(本节后面的练习中推导了光束方向图的表达式)。当 ϕ 较小时，光束方向图的形状类似于 Phong 高光。但是对于大掠射角，不仅光束方向图的形状不一样，而且最大强度的方向也不再是镜面反射的方向！这使镜面反射高光周围的光环与 Phong 高光周围的光环完全不同，这更接近真实材料表面的反射光所观察到的效果。

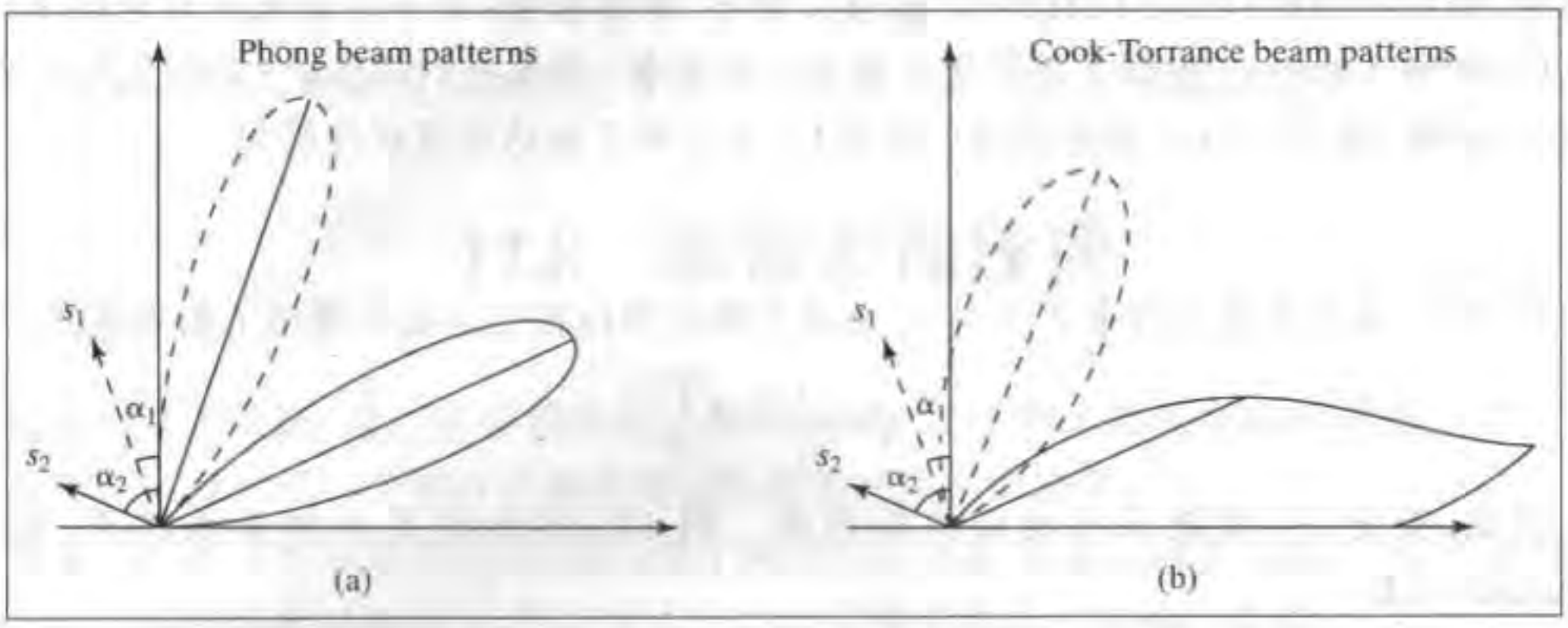


图 12.33　不同入射角的光线产生的镜面反射光的强度比较

Cook 和 Torrance【Cook82】在两个方向上扩展了这个模型。首先,它们考虑了如何将入射光的能量分成环境光、漫反射光和镜面反射光。其次,他们考虑到在实际材料中折射率随入射光波长的变化。这个变化也将引起菲涅耳系数随波长的相应变化,结果就导致镜面反射光随不同的入射角而改变颜色。这种颜色随着入射角度的变化,或称为"颜色偏移",在真实世界中也能被观察到。

如果上述每种现象都有一个表达式恰当表示,那么组合这些等式就可以得到关于此模型的一个通用表达式:

$$I_r = I_{ar}k_aF(0,\eta_r) + I_{sr}d\omega k_dF(0,\eta_r) \times lambert + I_{sr}k_sd\omega\frac{F(\phi,\eta_r)DG}{(\mathbf{m}\cdot\mathbf{v})} \quad (12.32)$$

该式只给出了红色光的强度;对于绿光和蓝光,除了将 η_r 分别换成 η_g、η_b 外,其他形式是一样的。环境光和光源的红色光强度分别为 I_{ar} 和 I_{sr},而环境光、漫反射光和镜面反射光这 3 个成分的贡献也可以分辨出来。I_{ar} 通常占 I_{sr} 的一小部分(如果存在多于一个光源,而且每个光源都有类似的漫反射光和镜面反射光,就是把所有光源的贡献加到一块)。

当垂直入射时,环境光和漫反射光的反射系数取决于菲涅耳系数 $F(0,\eta_r)$,这是 Cook 和 Torrance 做的简单近似。当 ϕ 值在 0 附近变动时,$F(\phi,\eta_r)$ 只有微小的变化,因此只有在 $\phi=0$ 附近这个近似才是合理的。注意到对于不同的波长(它们对应不同的颜色",这个系数是不同的:绿光和蓝光分别由 $F(0,\eta_g)$ 和 $F(0,\eta_b)$ 决定,因为不同颜色光的折射率不同,所以这两个值也不相同。

漫反射项中包含了我们所熟悉的 *Lambert* 项(参见式(12.24)),以及其他两个因子。$d\omega$ 项是光源相对于交点处的立体角(参见图 12.34)。为了简便起见,我们认为这一项对于场景中每一个点都是一个常数,而且通常令它为一个很小的常数,比如 0.0001(从地球观察太阳时,这个角的球面度为 0.000068)。因子 k_d(在镜面反射项中与之相对应的是 k_s)用来表示入射光是如何被分成漫反射光和镜面反射光的。这两个因子体现了材料的特性,且它们的和为 1,$k_d+k_s=1$。镜面反射项就是等式(12.31)中的 *spec* 项,其中菲涅耳系数对波长存在着明显的依赖关系。

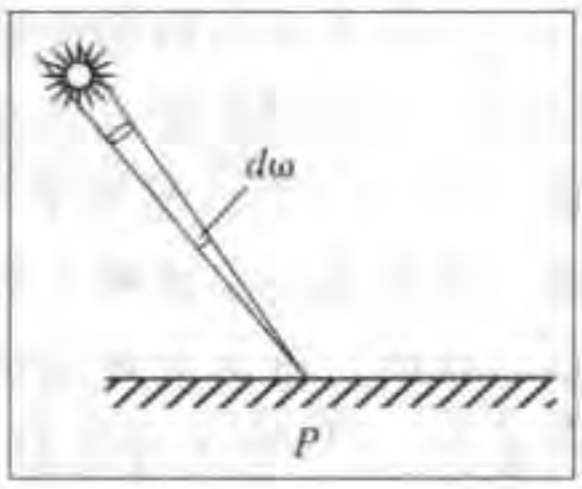


图 12.34 光源所对的立体角 $d\omega$

为了使用这些等式,我们必须要知道不同波长的光在材料中的折射率。通常得不到这个信息,我们得到的只是不同材料的"垂直反射率"的测量值,即不同波长光对应的 $F(0,\eta)$ 值。Cook 和 Torrance 根据下面的理论推导出折射率:如果我们知道 $\phi=0$ 时等式(4.28)中的 $F(\phi,\eta)$ 值,则 $c=1$,$g=\eta$(检验这一结果)。于是整个表达式可以化简为

$$F_0 = \frac{(\eta-1)^2}{(\eta+1)^2}$$

其中 F_0 表示垂直反射率 $F(0,\eta)$。由这个等式可以很容易地推导出 η 的表达式:

$$\eta = \frac{1+\sqrt{F_0}}{1-\sqrt{F_0}} \quad (12.33)$$

因此,给定 F_0 就能求出相应的折射率。图 12.35 给出了 4 种金属的 F_0 测量值【Touloukian】。

	红光的 F_0	绿光的 F_0	蓝光的 F_0
金	0.989	0.876	0.399
银	0.95	0.93	0.88
铜	0.755	0.49	0.095
铁	0.53	0.505	0.480

图 12.35 不同波长上测得的垂直反射率

Cook-Torrance 模型在计算上比 Phong 模型复杂得多，但是为了得到这种模型所提供的更逼真的效果，很多人宁可接受这些复杂的计算。如果要效果进一步逼真，可以采用更多波长的光来渲染场景。虽然只用红、绿、蓝 3 种颜色或多或少都可以与 CRT 显示器的颜色相匹配，但是如果要使计算机产生的图像达到终极的逼真度，那么它们始终受到显示设备的制约。

练习

练习 12.7.2 镜面反射对应的角是多少？

请证明图 12.30(b)中角 δ 必须满足 $\delta=(\theta-\phi)/2$ 才能使位于角 ϕ 的光源发出的光反射到达位于角 θ 处的观察者。

练习 12.7.3 微平面模型的光束方向图

由于入射光和反射光位于同一个平面内，因此等式(12.31)中采用的 G、D、F 等项都可以用入射角 θ 和观察角 ϕ 表示出来。

a. 证明等式(12.28)中的 G 可以表示为

$$G(\phi,\theta)=\min\left[1,\frac{\cos\left(\frac{\theta-\phi}{2}\right)\cos(\phi)}{\cos\left(\frac{\theta+\phi}{2}\right)},\frac{\cos\left(\frac{\theta-\phi}{2}\right)\cos(\theta)}{\cos\left(\frac{\theta+\phi}{2}\right)}\right]$$

b. 证明等式(12.31)中的 $spec$ 可以表示为

$$spec=\frac{F(\phi,\eta)D\left(\frac{\theta-\phi}{2}\right)G(\phi,\theta)}{\cos(\theta)}$$

c. 试画出当 ϕ、粗糙度 m 和折射率取不同值时，$spec$ 随 θ 的变化曲线图。

12.8 添加表面纹理

正如第 8 章所述，通过在各种表面上绘制纹理，可以使计算机生成的图像更加生动逼真。图 12.36 展示了一个带有几种纹理的场景的光线跟踪结果。

第 8 章中涉及的表面都是多边形，每个面用 OpenGL 渲染。对于每个面 F，它的每个顶点对应一组纹理坐标，OpenGL 用纹理图像内对应点绘制面片内每个像素。

我们想知道如何把纹理和光线跟踪过程结合起来。通常使用以下两类纹理。

(1) 图像纹理：将二维图像贴入物体的每个表面。

(2) 体纹理：物体是由某种材料块雕刻得到的。材质本身是有纹理的。光线跟踪显示出物体表面每个点的纹理颜色。

我们先考虑体纹理，因为它处理起来比较简单。我们研究在案例 8.5 中用 OpenGL 绘制多边形表面时，讨论了一种应用体纹理的方法，但是这个方法不是很自然。所幸的是，此方法可以很自然地合并到光线跟踪的框架内。

图 12.36 物体表面带有不同纹理的一个场景

12.8.1 体纹理

体纹理有时也称为三维纹理，它由 Perlin 和 Peachey[Perlin85, Peachey85]同时首先提出。我们把物体看作由纹理材料例如大理石或木头雕刻得到。纹理用函数 $texture(x, y, z)$表示，它在空间内每个点产生一个颜色值(r, g, b)。想象这种纹理是随着坐标变化的颜色或彩墨，我们在不同点(x, y, z)可以(用 X 射线视觉)观察到不同颜色。在此空间内定义一个具有某种形状的物体，将形状外所有材料裁剪掉之后，物体表面上的点(x, y, z)就会显现出来，并具有特定的纹理。

我们先详细解释实例 8.5 中提到的几个有趣的例子，然后介绍怎么在光线跟踪器中加入这类纹理。接下来，我们再介绍一些更丰富的材质，如木纹和大理石。

例 12.8.1 填满整个空间的三维棋盘

想象一个三维棋盘，由红色和黑色立方体在整个空间内交替堆积构成，将其中一个小方块的一个顶点放置在点(0, 0, 0)处，并调整其大小使其对角顶点位于点 $S=(S.x, S.y, S.z)$处。对于其他小方块也采用同样大小(宽为 $S.x$，高为 $S.y$，等等)，并将它们一个挨一个地排列在整个三维空间中。对于这样一个棋盘纹理，很容易写出其表达式：将 $x/S.x$、$y/S.y$ 和 $z/S.z$ 的整数部分相加，再将得到的和关于 2 取模：

$$\text{jump}(x,y,z) = ((\text{int})(x/S.x) + (\text{int})(y/S.y) + (\text{int})(z/S.z))\%2 \quad (12.34)$$

然后，若 $jump()=0$，则令 $texture(x, y, z)$返回黑色；若 $jump()=1$，则返回红色。

图 12.37 展示了由这种体纹理材料构成的一个通用球体和一个通用立方体。同时显示的是有类似纹理的一组相互贯穿的平面和一个逐渐变小的锥形柱面。纹理颜色用于设置表面上的漫反射系数，因此材料的颜色就是纹理的颜色。漫反射成分随着位置(相对于光源)的变化而变化，Phong 镜面反射成分就是光源的颜色。注意到此球体和立方体显然是由实体“小方块”构成的。对比一下，如果将一个二维棋盘图像粘贴到这些物体上，会是什么样？

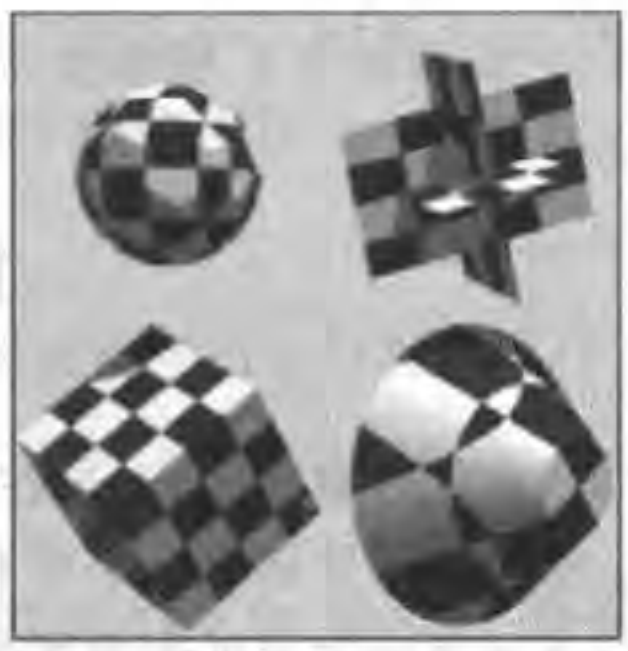


图 12.37 对于一些具有棋盘体纹理的物体的光线跟踪

例 12.8.2　一堆颜色渐变的立方体

也可以将一组同样的立方体堆积起来。在这些立方体内部移动时，可以看到平滑变化且覆盖整个光谱的颜色。立方体所有 8 个角都是黑色，而中心点(0.5，0.5，0.5)为白色，从而在中心和角落之间颜色平滑地变化。为了使红色成分沿着 x 方向上升为 1 再降回 0，可以取 $red=1-|2x-1|$。对于绿色成分和蓝色成分，可以分别得到沿 y 方向和 z 方向的类似表达式。把无限多个这样的立方体在空间中堆积在一起，利用 x、y、z 的小数部分(fraction)，最后可以得到

$$\begin{aligned}\text{texture}(x,y,z) = (&1-|2\text{fract}(x)-1|,\\ &1-|2\text{fract}(y)-1|,\quad 1-|2\text{fract}(z)-1|)\end{aligned} \tag{12.35}$$

其中 fract(x)为 1−(int)x 即 x 的小数部分。想象一下，一个由这种材料制成的半径为 100 的球体会是什么样？

对体纹理对象进行光线跟踪

一旦得到函数 texture()，那么基本上不需要其他任何信息就可以将体纹理添加到光线跟踪器中。纹理可以通过下述几种方法改变从表面某点发出的光线：

1. 光线可以设置成 texture()本身，看起来就像物体本身发出这种光。
2. 纹理可以调整环境光和漫反射系数(回忆第 8 章)，使得：

$$I = \text{texture}(x,y,z)(I_a\rho_a + I_d\rho_d lambert) + I_s\rho_s phong \tag{12.36}$$

其中 texture(x，y，z)在交点(x，y，z)计算。这是纹理最普遍的用途：根据 texture()的波动，物体表面不同点处的固有颜色看起来变得更暗或更亮。这里，高光反射的颜色和光源相同，不受纹理影响。这使得纹理物体表现出发光特性，仿佛它是由颜料制成的。

等式(12.36)中用到的交点(x，y，z)既可以是在通用坐标系下，也可以是在世界坐标系下。不过一般都将它置于通用坐标系中，在这种情况下，物体在旋转或移动到场景中最终位置的过程中，会一直“携带”着它的纹理。在动画中，物体从一帧旋转或移动到另一帧，而纹理却始终牢牢地附在这个物体表面。

另一方面，如果(x，y，z)位于世界坐标系下，则纹理在空间中是固定的。此时当物体在动画中旋转或移动时，纹理将会“扫过”物体，使物体在每个新位置看上去都像是由新材料雕刻出来一样。这个过程将产生非常有趣的视觉效果。

丰富多变的体纹理很容易创建，其中一些还能很真实地模拟实际材质，两个经典的例子就是木纹和大理石。

木纹纹理

不同颜色的同心圆柱形成了木材上的纹理，切开原木时就可以看到相应的圆环。随着点到中心轴的距离发生变化，函数值在两个数值之间来回变动。这种效果可以用以下模函数模拟：

$$\text{ring}(r) = ((\text{int})r)\%2$$

其中，此圆环的中心轴为 z 轴，半径为 $r=\sqrt{x^2+y^2}$。随着 r 由 0 开始增大，rings%2 的值在 0 和 1 之间来回变动。利用函数

$$\text{simple_wood}(x,y,z) = D + A \cdot rings(r/M)$$

可以使纹理在两个预设值 D 和 $D+A$ 之间来回变动。这里仍有 $r=\sqrt{x^2+y^2}$,这就产生了一系列厚度为 M、以 z 轴为中心的同心圆环。

如果对这些圆环进行摆动、扭曲或者旋转[Watt92],就会得到更有趣的结果。为了摆动这些圆环,我们添加了一个分量,它随着(关于 z 轴的)方位角 θ 的变化而变化:

$$\text{rings}\left(\frac{r}{M} + K\sin\left(\frac{\theta}{N}\right)\right)$$

这样一来,即使固定半径不变,这些圆环也会随着 θ 的改变而发生变化,此时围绕 z 轴观察,圆环会里外摇摆 N 次。注意我们在原函数内嵌套了一些别的函数来添加更多效果:*rings*()的参数不只是 r,而是 r/M 与一个正弦函数的和,而这个正弦函数本身的参数是一个角度关于 N 的商。Perlin[Perlin85]充分利用了这种复合函数,创造出许多有趣的视觉效果。利用下面的等式可以进一步在摆动的木纹纹理上增加一种"螺旋效果":

$$\text{rings}\left(\frac{r}{M} + K\sin\left(\frac{\theta}{N} + Bz\right)\right) \tag{12.37}$$

这样就使得正弦函数的相位随着 z 变化,在 z 变化时有效地旋转上述摆动。如果我们想倾斜木纹,使它以 z 轴之外的轴为中心,那么在计算 r 和 θ 之前就要先对它进行一次旋转。例如,我们取 r 为 $\sqrt{x'^2+y'^2}$,其中 $(x',y',z')=T(x,y,z)$ 对应于旋转变换 $T()$。

图 12.38 展示了一些从木纹材料"雕刻"出来的物体,这些木纹材料就是按上述方法定义的。

三维噪音和大理石纹理

像大理石这样的材料,它的纹理非常混乱,如图 12.39 所示。深色材料代表的湍流(turbulence)穿过大理石,并随机地产生螺旋和斑点,使得大理石看上去像是由一些急速搅拌的熔融材料形成的。

图 12.38 从木头雕刻出来的物体

图 12.39 从大理石雕刻出来的物体

我们可以模拟这种湍流。先构造一个"噪音"函数,让它在空间内每个点 (x, y, z) 处产生明显的随机值,然后,可以用一种容易控制的方法搅动这个"噪音场"来形成湍流现象。

噪音场本身很容易编程实现。想象空间每个整数位置(即在点 $(x, y, z)=(i, j, k)$ 处,其中 (i, j, k) 是任意整数 i、j、k 的组合)随机地给定一个值。这种点排列称为整数格

(interger lattice)。图 12.40(a)显示了一个二维的情况，图中对二维整数格中每个点都用 0～1 之间的噪音值标记。例如，点(1，2)的噪音值是 0.653，而点(3，1)是 0.129。图 12.40(b)显示了一个三维整数格。其中每个整数点都对应某个固定的噪音值，例如在(2，2，1)处对应的是 0.7341。

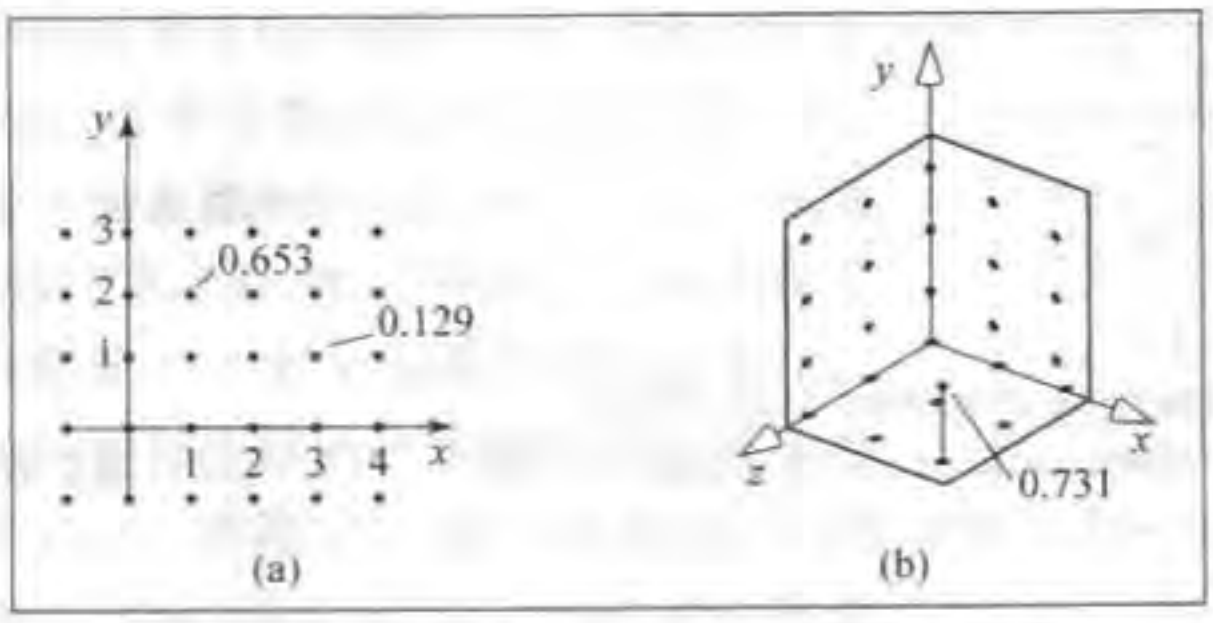


图 12.40　在每个三维整数格点定义噪音值

所有噪音值可以保存在一个巨大的数组 float N[10000][10000][1000]中，但是这样占用的内存之大令人无法接受。而如果只在每次需要时产生一个噪音值，问题就简单多了。为此需要构造一个函数，例如 float latticeNoise(int i, int j, int k)，当给定整数 i、j、k 来指定整数格位置时，这个函数就返回一个明显的随机值。这个函数必须足够高效并且是完全可重复的，也就是说对于给定的(i，j，k)，该函数返回的噪音值总是一样的。在 Texturing and Modeling【Ebert98】这篇优秀的论文中，几位作者介绍了几种适当的方法。这里重点介绍其中一种，这种方法由 Perlin 创立，并由 Peachey 进行了改进。

产生可重复的随机数

主要技巧是在初始化阶段建立一个固定的数组，例如 noiseTable[]，用来存放伪随机噪音值。研究发现，长度为 256 的数组已经足够大，因此就采用这个长度。

接下来，主函数 latticeNoise(i, j, k)只要用可重复的方法索引 noiseTable[]即可。为了确保当 i、j、k 变化时噪音值的变化没有规律，索引函数将(i，j，k)组合有效地散列编码为介于 0～255 之间的数。这只要用另一个数组 index[]就可以很容易地实现，该数组包含 0～255 中所有整数的一个随机排列。Peachey 建议定义下面两个宏：

```
#define PERM(x) index[(x) & 255]
#define INDEX(ix, iy, iz) PERM( (ix)+PERM( (iy)+PERM(iz)))
```

宏 PERM 取 x 的整数部分，再对它和数值 255 执行一个“按位与”操作，只保留 x 的低 8 位，再将此低 8 位值散列编码为 0～255 中某个数。因此，PERM 的值就是 index 数组中的一个数。宏 INDEX 利用 PERM 访问 index 数组 3 次，每次分别根据 ix、iy、iz 选择 index[]中的元素。注意到这个操作是可重复且高效的，期间还有大量的置乱操作。于是，函数 latticeNoise()可以简化为：

```
float latticeNoise(int i, int j, int k)
{
    return noiseTable[ INDEX(i, j, k)];
}
```

开发 Noise 类

如果把刚才提到的所有操作都封装到一个类 Noise 中,将会非常方便,这个类将用来生成大理石及其他类噪音纹理。图 12.41 给出了这个类的声明和定义(也可参见本书网页)。

```
class Noise{
public:
    Noise()                                          //构造函数
    {
        int i;
        index=new unsigned char[256];
        for(i=0; i<256; i++) index[i]=i;             //用索引填充数组
        for(i=0; i<256; i++)                         //置乱
        {
            int which=rand()%256;                    //选择数组中随机位置
            unsigned char tmp=index[which];          //将它们互相交换
            index[which]=index[i];
            index[i]=tmp;
        }
        noiseTable=new float[256];
        for(i=0; i<256; i++) noiseTable[i]=rand()/32767.99;
    }    //构造函数结束
    float noise(float x, float y, float z);
    float noise(float scale, Point3&p);
    float turbulence(float x, float y, float z);
    float marble(float x, float y, float z);
    float marble(float strength, Point3& p);
private:
    float * noiseTable;                              //噪音值数组
    unsigned char * index;                           //伪随机索引
    float mySpline(float x);                         //用于生成大理石纹理
    float latticeNoise(int i, int j, int k)
    {    //返回整数格上的噪音值
        #define PERM(x) index[(x) & 255]
        #define INDEX(ix, iy, iz) PERM( (ix)+PERM( (iy)+PERM(iz)))
        return noiseTable[INDEX(i, j, k)];
    }
};
```

图 12.41 生成可重复的噪音值

这个类中最有趣的方法是 marble(),它返回一个 0~1 之间的亮度值,这个亮度值的大小视位置而定,可以用来模拟大理石上的明暗波纹。稍后研究这个算法的细节。这个类可以用来生成一种带绿色的大理石纹理,只要在光线跟踪的开始阶段通过命令

```
Noise n;                    //构造和创建一个噪音对象
```

构造一个噪音对象,然后再用 n.Marble(x, y, z)获得每个点(x, y, z)处的 texture(x, y, z)值即可。

方法 marble()使用了 noise()和 turbulence()(随后将对它们做简短介绍)以及辅助函数 latticeNoise()。这个类的构造函数创建并填充数组 noiseTable[]和 index[]。利用 C 标准库函数 rand()生成随机值并将其写入 noiseTable[],调整这些值的大小使之介于 0.0 和 1.0 之间。先按顺序将 0～255 中的数赋给数组 index[],然后再将该数组中每个元素依次和另外一个随机选中的元素交换,这样一来,元素的顺序就被"打乱"了。

函数 latticeNoise()在整数格点上产生随机值,我们希望能利用它来构造一个函数 noise(x, y, z),可以在整格点之间(也就是空间任意点处)生成随机值。此外我们还希望这个噪音函数随着 x、y 和 z 的变化而平缓变化。

在这些整数格之间进行简单的线性插值可以得到令人满意的结果[①]。图 12.42 显示了二维下的插值,三维的情况与之类似。这里,我们希望利用 4 个相邻的整数格点处的噪音值,来计算点(x, y)=(0.6, 1.4)处的噪音。首先沿 y=1 和 y=2 对 x 进行插值,得到以下值[②]

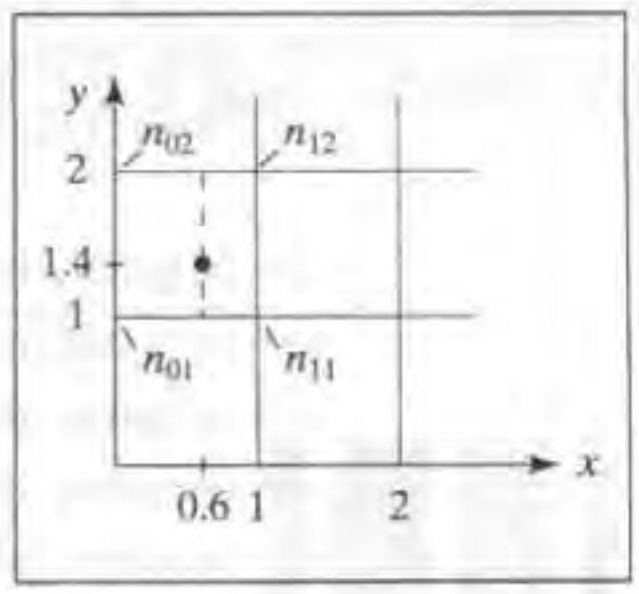


图 12.42 二维下噪音值的线性插值

$$n(0.6,1) = \mathrm{lerp}(0.6, n_{01}, n_{11})$$

$$n(0.6,2) = \mathrm{lerp}(0.6, n_{02}, n_{12})$$

然后对这些值关于 y 插值,得到

$$n(0.6,1.4) = \mathrm{lerp}(0.4, n(0.6,1), n(0.6,2))$$

图 12.43 给出了在三维中实现 noise()函数功能的一种方法。它利用一个额外的参数 scale 来对三维点(x, y, z)进行缩放;在产生湍流时,这个方法是很有用的。首先,将经过缩放后的点在 x、y 和 z 方向上各偏移 1000,使所有分量都为正(因为格点值是随机的,所以这一偏移不会改变所产生的噪音的统计性质)。然后,在这个点的周围 8 个整数格点处生成噪音值。最后,再用 7 个 lerp 函数插值得到噪音值。

```
float Noise::noise(float scale, Point3& p)
{     //对格点噪音进行线性插值
    #define lerp(f, A, B)A+f*(B-A)
    float d[2][2][2];
    Point3 pp;
    pp.x=p.x*scale+10000;          //偏移量不应为负值
    pp.y=p.y*scale+10000;
    pp.z=p.z*scale+10000;
    long ix=(long)pp.x;
    long iy=(long)pp.y;
    long iz=(long)pp.z;
    float tx, ty, tz, x0, x1, x2, x3, y0, y1;
    tx=pp.x-ix;                    //分量
```

图 12.43 在任意点 p 处生成噪音的函数

① Perlin 和 peachey 认为三次插值得到的结果更真实。

② 回忆第 5 章介绍的 lerp()函数:lerp(f, A, B)=A+(B−A)f,f 为小数,表示一个介于 A 和 B 之间的值。

```
    ty=pp.y-iy;
    tz=pp.z-iz;
    float mtx=1.0-tx, mty=1.0-ty, mtz=1.0-tz;

    for(int k=0; k<=1; k++)     //获取8个格点处的噪音值
    for(int j=0; j<=1; j++)
    for(int i=0; i<=1; i++)
        d[k][j][i]=latticeNoise(ix+i, iy+j, iz+k);

    x0=lerp(tx, d[0][0][0], d[0][0][1]);
    x1=lerp(tx, d[0][1][0], d[0][1][1]);
    x2=lerp(tx, d[1][0][0], d[1][0][1]);
    x3=lerp(tx, d[1][1][0], d[1][1][1]);
    y0=lerp(ty, x0, x1);
    y1=lerp(ty, x2, x3);
    return lerp(tz, y0, y1);
}
```

图 12.43(续)

图 12.44(a)显示了函数 noise(20, x, y, 0)对应的图,黑色代表 0.0,白色代表 1.0。图中 x 和 y 的值都是从 −1 变到 1。由于随机数生成的特殊过程,因此噪音场显现出了某些结构,但还不是十分清晰。

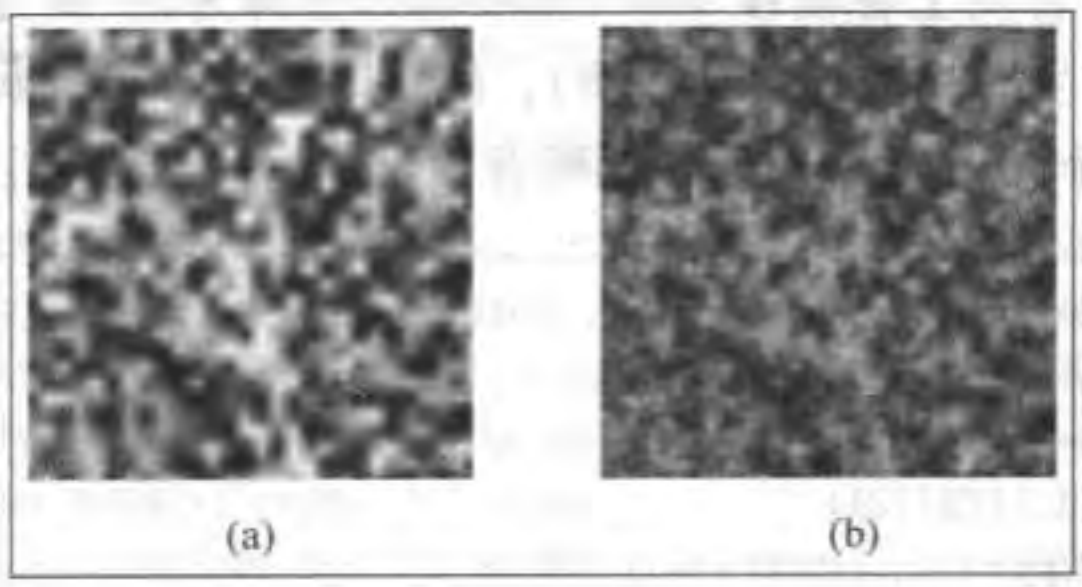


图 12.44

(a) noise()的样图;(b) turb()的样图

扰动

Perlin【Perlin85,Ebert98】设计了一种方法用来产生比前述结果更加有趣的噪音。这种方法的思想是将几个噪音分量叠加在一起。这些分量包括:随着空间中位置的移动函数值发生缓慢波动的分量、波动速度为前述速度 2 倍的分量、波动速度为前述速度 4 倍的分量等。变化速度越快的分量,它的强度越小。函数

$$\text{turb}(s,x,y,z) = \frac{1}{2}\text{noise}(s,x,y,z) + \frac{1}{4}\text{noise}(2s,x,y,z) + \frac{1}{8}\text{noise}(4s,x,y,z) \tag{12.38}$$

中添加了 3 个这样的分量，其中每个分量的强度依次是前一个的一半，而变化速度则依次是前一个的两倍。参数 s 的作用和在 noise()中一样，仍然用于调整距离。图 12.45 提出了一个在二维中观察 turb()的波动情况的方法。想象用 noise(1, x, y, 0)覆盖 xy 平面，对于每个点 $P=(x,y)$，turb(1, x, y, 0)将平面上(x, y)、($2x$, $2y$)和($4x$, $4y$)三个点上的噪音值相加。在 P 点的临近点 P' 处，noise(1,x',y',0)和 noise(1,x,y,0)的值非常接近，但 noise(2,x',y',0)与 noise(2,x,y,0)的值之间差距就比较大，而 noise(4,x',y',0)和 noise(4,x,y,0)差距就更大了。第一个噪音分量表现出来的特征只有第二个分量的一半，第三个分量的四分之一。

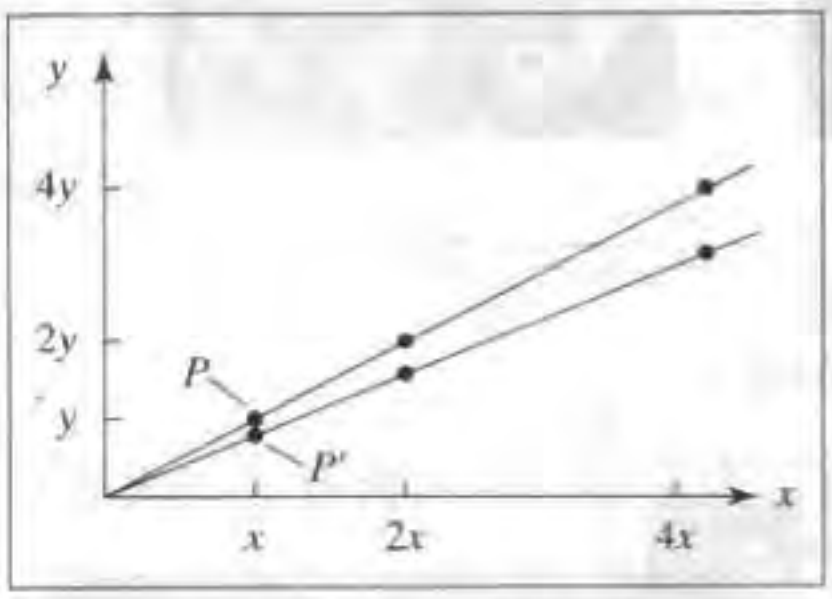


图 12.45　turb()函数

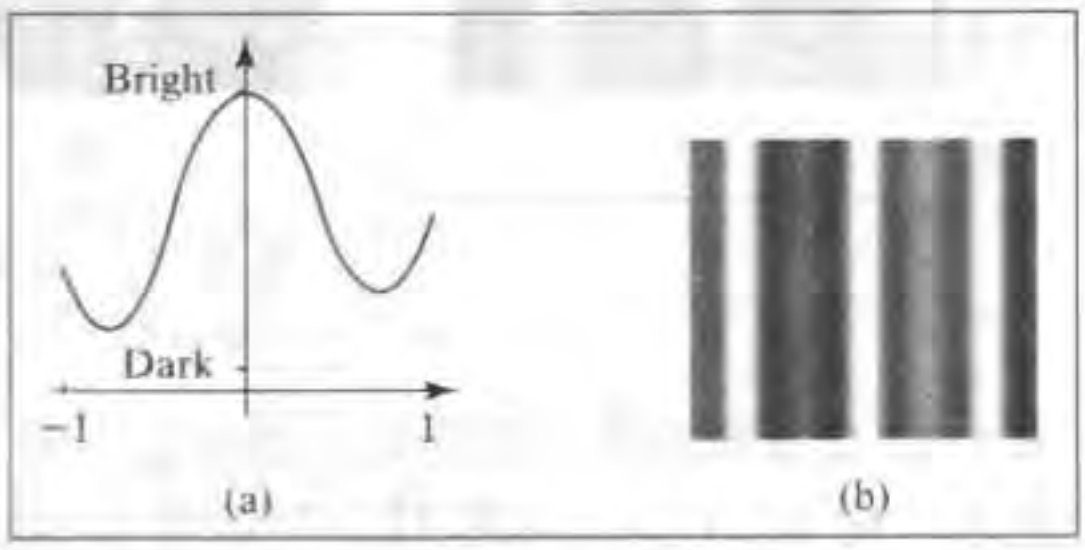


图 12.46　构造大理石纹理

(a) 样条曲线；(b) 未经扰动的大理石纹理

在任何细节尺度上，每个分量的大小都是前一分量的 1/2，而它的频率(波动速度)则是前一分量的两倍。这就形成了附录 5 中介绍的**自相似性**，实际上，在一定条件下，turb()可以近似得到所谓的“1/f 噪音”，各种自然图形中都会表现出这种噪音。图 12.44(b)显示了 $M=3$ 时由图 12.46(a)的 noise()场产生的 turb()函数的图，其中明显有更多细节，而波动则更平缓，更像云雾。类似 turb()这样的函数所产生的扰动可以用来改变外形或纹理的某些性质，使其显得更有真实感，这在后面对大理石纹理的讨论中可以看到。

大理石纹理

大理石材料呈现出明暗相间的脉络，这些脉络有一定规律，但同时也呈现出杂乱而不规则性。根据 Watt【Watt92】的观点，通过给脉络一个平滑的波动(例如在 z 方向上)，再用 turb()函数去扰动它，就可以构造出一种类似大理石的三维纹理。我们先考虑一种在 x 和 y 方向上都是常数，只在 z 方向上发生渐变的纹理：

```
marble(x, y, z)=undulate(sin(z))
```

其中 undulate()是图 12.46(a)所示的样条形函数【Watt92】，当参数从 −1 变到 1 时，这个函数将在某个暗值和某个明值之间变化(本节后面的练习将讨论此样条函数)。把 sin(z)作为 undulate()的参数，就产生了一个 z 方向的周期性“波纹”，这个“波纹”在样条曲线上来回移动，每个周期移动一次，于是就产生了图 12.46(b)所示的亮度波动。由于大理石中颜色的纵向脉络过于规则(这是很自然的)，因此就要调整 sin()的参数使它带有某种波动：

$$\text{marble}(x,y,z) = \text{undulate}(\sin(z + A \cdot \text{turb}(s,x,y,z)))$$

sin()函数的相位在大理石中的不同位置有不同偏移，这样就产生了更逼真的脉络。参

数 s 使不同点处的扰动变得更快或更慢;而参数 A 则改变扰动总量的大小。Watt 证明如果用简单的直线代替 undulate(),所产生的效果就会很差。

图 12.47 显示了一个立方体表面上的大理石纹理。相应的函数是:

$$g = \text{undulate}(\sin(2\pi z + A \times \text{turb}(5, x, y, z)))$$

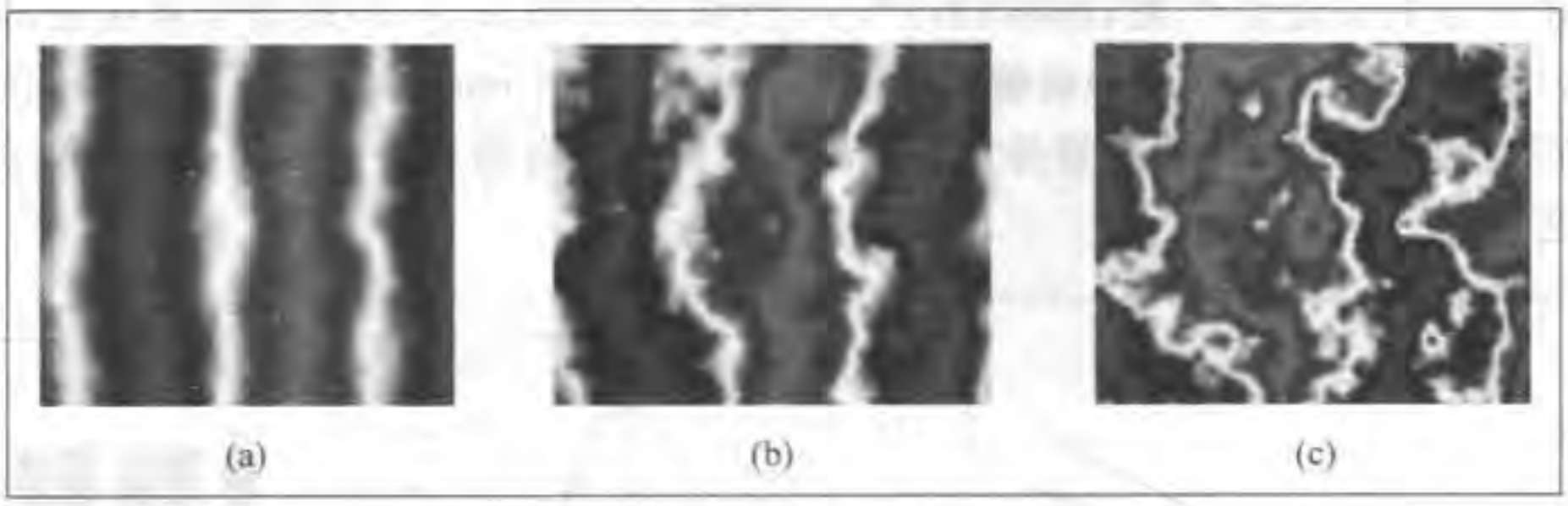


图 12.47 大理石纹理

(a) A=1; (b) A=3; (c) A=6

其中,z 自右向左从 0 变到 1,y 指向上。图 12.47(a)中振幅 $A=1$,因此表现出来的扰动非常小。图 12.47(b)和图 12.47(c)中,A 的值分别是 3 和 6,此时扰动也相应变大了。

可以将 marble()的值作为等式(12.36)的反射系数,调整不同光线命中物体表面不同点后返回的光的总量。最直接的方法是,扩展 marble(),使其分别返回全色光线跟踪器的红、绿、蓝分量。

图 12.48 展示了一个包含大量大理石物体的场景内的光线跟踪,效果相当令人满意。

图 12.48 由大理石物体组成的一个场景

12.8.2 在表面上粘贴图像

第 8 章分析了如何利用 OpenGL 将图像粘贴到多边形表面上。现在我们介绍如何利用光线跟踪器将图像粘贴到任意曲面上。实现此操作的过程很简单,结果也非常好,但是执行时间要比使用 OpenGL 长。因为除了其他额外负荷,每个像素都要单独计算,无法利用扫描行的连贯性。

和第 8 章一样,假设已定义一个二维纹理函数 texture(u, v),其中 u 和 v 分别从 0 变到 1,该函数在每个点(u, v)都产生一个亮度值或颜色。函数 texture(u, v)可以是一种过程纹理(如棋盘或 Mandelbrot 集),也可以是保存在像素图中的图像纹理。假设像素图是一个存储像素值的 $N \times M$ 数组,记为 txtr[][]。那么给定 u 和 v 在 0~1 之间的值后,就可以用下述语句在 txtr 中检索相应的像素:

```
txtr[(int)(u/N)][(int)(v/M)]
```

在一般物体上直接粘贴纹理,比在经过场景坐标变换后的物体上粘贴纹理要简单得多。设计者将纹理坐标和一般物体的位置坐标关联起来,使得当把物体变换到场景中时,纹理也

正确地(并以适当的纵横比)显示在变换后的物体上。

用纹理覆盖表面

我们需要一种方法,把一般物体表面上的点(x, y, z)和纹理坐标(u, v)关联起来。显然对于不同的物体形状,需要使用不同的映射。

例 12.8.3　正方形和平面的纹理

通用平面即 xy 平面,通用正方形就位于此平面内。因此 texture(u, v)的图像平面和通用正方形或平面之间存在一个很自然的关联。如图 12.49 所示,设计者在平面上选择一个窗口($left$, top, $right$, $bottom$),当光线与此窗口相交时,可以很容易计算出应使用纹理中哪个点(u, v)。

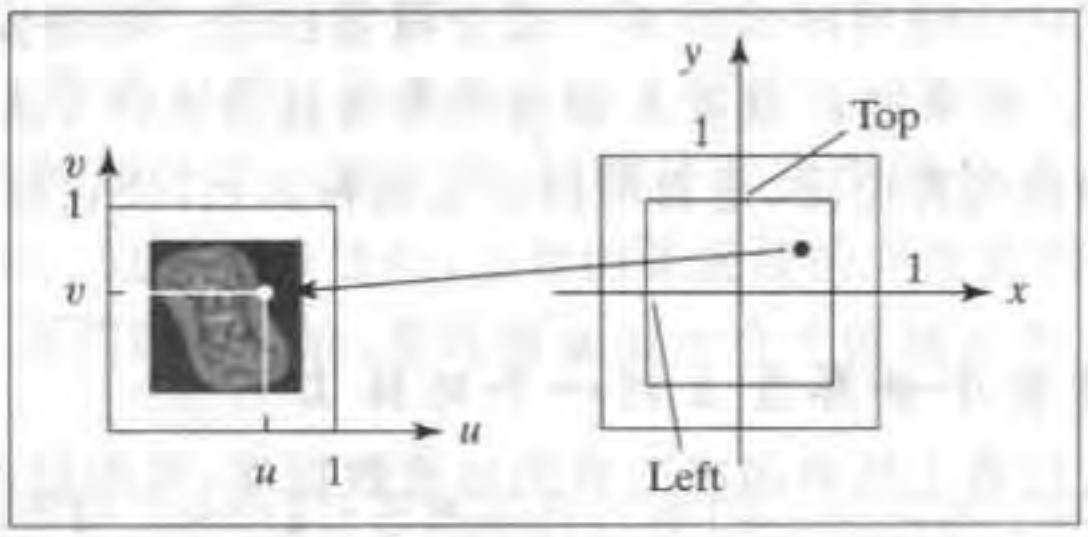


图 12.49　将纹理映射到正方形或平面上

具体地说,给定通用坐标系下的交点(x, y, 0),则相应的纹理坐标为

$$u = \frac{x - left}{right - left}, \quad v = \frac{y - bottom}{top - bottom} \tag{12.39}$$

设计者还必须确定如何处理落在窗口之外的交点。当这种情况出现时,一个常用的方法就是,取为某个预先指定的纹理颜色。

例 12.8.4　柱面上的纹理

在通用圆柱面上覆盖纹理同样简单。回忆 8.5.5 节中关于如何把纹理覆盖到圆柱形曲面上的讨论。通用锥形柱体如图 12.50 所示,在它的表面上有一个指定的"窗口",窗口的方位角从 a_1 扩展到 a_2,在 z 轴方向从 z_1 延伸到 z_2。当一条光线在(x, y, z)处与柱体相交时,容易计算出方位角为,然后即可用下式计算纹理坐标(u, v):

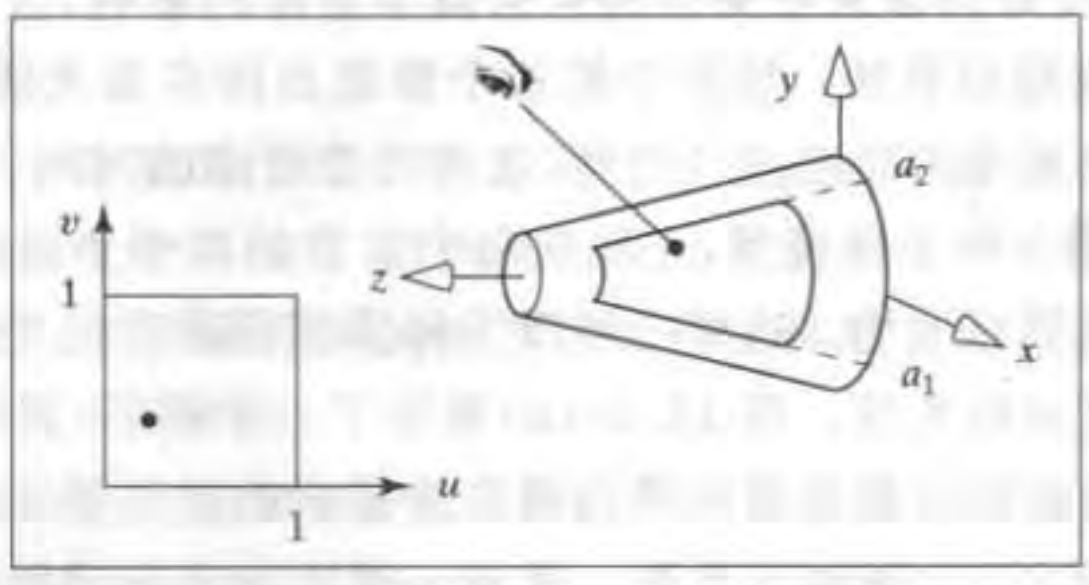


图 12.50　在柱体上覆盖纹理

$$u = \frac{\theta - a_1}{a_2 - a_1}, \quad v = \frac{z - z_1}{z_2 - z_1} \tag{12.40}$$

若交点位于指定窗口之外,设计者须选择用于该处的纹理颜色。注意,如果锥体缩小很快,则映射的纹理图像将发生扭曲。接下来的练习将解决如何将纹理粘贴到柱体的顶部或底部,另外还提到了在球体或圆锥体上覆盖纹理的问题。

练习

练习 12.8.1 将纹理覆盖到柱体的顶部或底部

试详细说明如何将纹理覆盖到一个柱体的顶部或底部。将位图的一部分粘贴上去,使之完全覆盖顶部或底部。

练习 12.8.2 在球面上覆盖纹理

试详细说明如何在一个通用球体的某个部分覆盖纹理。纹理窗口的方位角从 a_1 扩展到 a_2,纬度从 l_1 变到 l_2。如果把 l_1 或者 l_2 设置为非常接近 $\pm 90°$,试分析位图发生扭曲的程度如何。如果用纹理函数代替位图,是否可以定义某种函数,使它即使在接近球的两极也不会发生扭曲?

练习 12.8.3 将整个世界覆盖到一个球体上

假设你有一个关于世界上所有国家边界的地理数据库,其中包含了大量的折线,它们的端点用(经度,纬度)表示。利用此数据库是否可以将世界地图准确地覆盖到一个通用球体上?如果可以,试说明如何实现。

练习 12.8.4 在圆锥上覆盖纹理

试详细说明如何在一个通用圆锥体的侧面的某个部分覆盖纹理。对窗口参数的自然解释是什么?讨论位图在各种情况下会发生什么样的扭曲。说明如何将纹理映射到圆锥的底部。

12.9 光线跟踪算法中的反走样技术

光线跟踪算法是一个点采样的过程——沿着每条单独的光线将场景离散开来。因此,光线跟踪的图像经常出现走样问题,这并不奇怪。

前面学过,可以通过对场景采样更多的点来减少走样的影响,这个通常称为**超采样**。这个技术也可以用于光线跟踪算法:场景中的每个像素点用多条光线来跟踪,将每条光线返回值的平均作为当前像素点的亮度值。当然,这将花费更多的时间。

图 12.51(a)给出了一种采样模型,光线从每个像素的四个角通过。像素的最终颜色取为它四个角的光线颜色值的平均。这样一种反走样算法容易实现并且时间花费较少。超采样可以为每个像素取更多的光线。图 12.51(b)展示了一个例子,其中每个像素采样九条光线。最后将这九条光线返回的颜色值的平均值作为像素的颜色值。

还有几个更有效的方法也被提了出来。Whitted【Whitted80】提出了一种自适应的处理方法,在更需要反走样处理的区域即图像颜色值改变迅速的区域插入更多的光线。

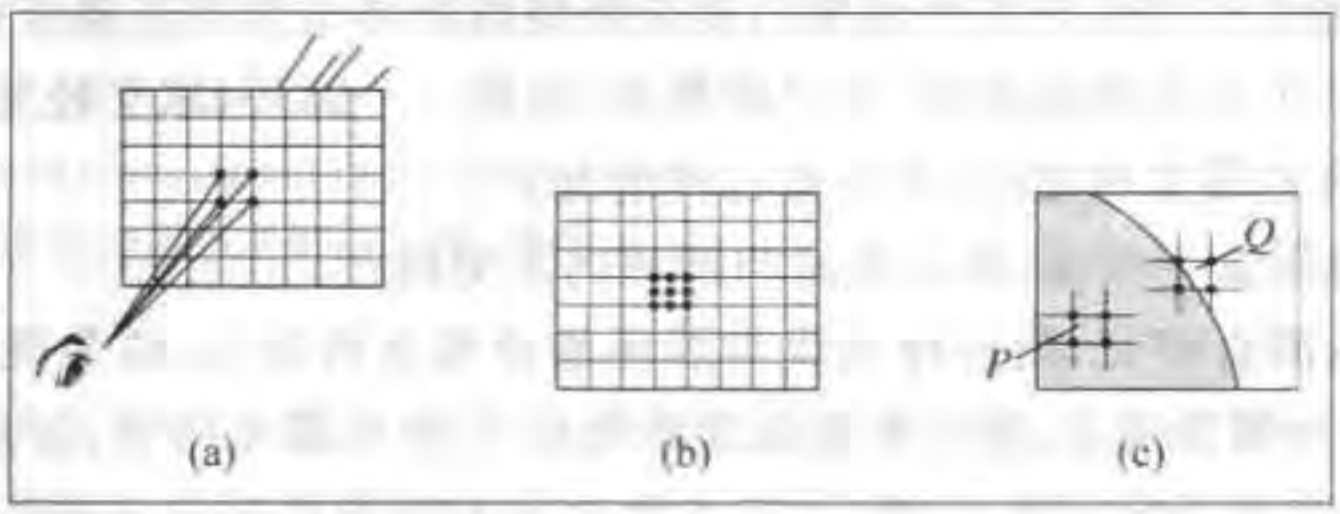


图 12.51　反走样中的超采样技术

在 Whitted 的方法中，从像素的四个角发射四条光线，然后对四条光线的亮度进行平均，这个平均值将与四个单独的亮度值进行比较。如果有一个角上的光线亮度值与平均值相差很大，那么这个像素将被一分为四，接着对亮度相差大的这四分之一块的四个角再采样光线。参见图 12.51(c)。像素 P 的四个角的光线返回了几乎相同的值，这是因为在这个区域，场景的变化并不大。像素 Q 的一条光线返回了与其他三条相差很大的亮度值。因此三条新的光线将从 Q 左下的四分之一块射出，同样计算这个四分之一块的四条光线的平均值，并与每条光线做比较。这样一直迭代，直到四条光线的亮度值都很接近或者迭代到了用户指定的次数为止。在迭代结束之后，最终像素值由这个区域所有光线值的一个加权平均来决定。

另外一种技术是基于【Cook84】分布式采样技术的。它采用了随机采样的形式，这个在前面第 9 章的纹理反走样里面提到过。对每个像素做随机的光线采样，然后对这些光线返回的亮度值做平均。例如，将像素细分成规则的 4 乘 4 的网格。但并不是对这些规则网格点进行采样，而是对抖动的网格点进行光线采样。通过对图像加入噪音的度量来抖动采样点，这些加入的噪音比走样的误差对人眼的视觉影响要小得多。采用抖动采样点的小网格采样比不采用它的采样效果好很多。

图 12.52　台球绘制中的抖动效果

图 12.52 是用随机采样的光线跟踪技术绘制一个台球的场景。特别注意图中的走样现象，特别是在球的边缘。

12.10　包围盒技术

光线跟踪算法是一个重复迭代的过程，它对大量的光线一遍又一遍地进行同样的操作。每条光线要对场景中的每个物体相交，这样就涉及到大量求交的运算。当考虑到阴影、反射和折射时，情况变得更糟。这时就需要能减少待处理物体数量的技术。包围盒技术的运用极大地提高了光线跟踪算法的速度。

一个物体的包围盒是能把这个物体包裹在内的一种形状。它通过快速排除与被包围物体不相交的光线来提高光线跟踪的速度。如果一条光线与某个包围盒不相交，那么它必然与此包围盒内的物体不相交。如果包围盒的形状是简单形状，那么判别这样一个简单形状

与一条光线是否相交比判别与它所包围的复杂外型的物体是否相交要节省很多时间。

图 12.53 是一个长方体包围盒(求交较简单)包围一个圆环(求交较复杂)的例子。当判断一条光线是否与包围盒相交时,有下面三种情况:

(1) 光线与包围盒不相交,那么光线与圆环的求交跳过。

(2) 光线与包围盒相交,则进行光线与圆环是否相交的测试,结果光线与圆环不相交。(这种情况也被称为错误警报;前一半测试预示光线可能与圆环相交,但是事实上它们并不相交)

(3) 光线与包围盒相交,进行光线与圆环是否相交的测试,光线确实与圆环相交。

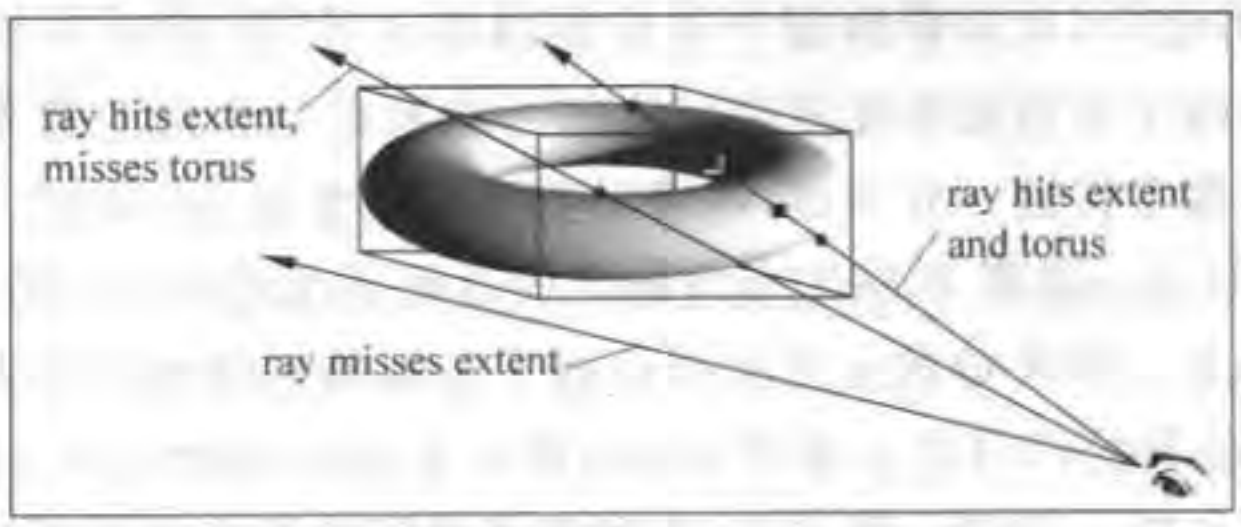


图 12.53 圆环和它的长方体包围盒

如果对物体进行求交测试比对包围盒进行求交测试花费大得多,用包围盒技术省去对物体求交要比维护这样一个求交列表花费少很多。

补充:包围盒技术对时间节省的一点分析

假设测试一条光线与包围盒是否相交花费 T 个时间单位,而一条光线与圆环的求交测试花费 mT 个时间单位。假设绘制这个场景需要测试 N 条光线,其中有比率 f 部分是与包围盒相交的。绘制需要计算 N 次光线与包围盒的交,花费 NT,fN 次光线与圆环的交,花费 $fNmT$。因此总的时间花费是 $NT(1+fm)$。另一方面,如果没有使用包围盒技术,所有的 N 条光线都要跟圆环做求交测试,总的花费则是 mNT。于是得到如下比率:

$$\text{speed ratio} = \frac{\text{cos t without extents}}{\text{cos t with extents}} = \frac{m}{1+f\cdot m} \tag{12.41}$$

例如,如果 $m=20$,$f=1/40$(在一个典型的场景中,每个物体只占整个场景的一个很小部分)。这个比率大概是 13,因此如果禁止包围盒求交,那么总的时间将延长至使用包围盒技术的 13 倍之多。

从上面的分析注意到,f 应该尽可能的小。也就是说,每个包围盒应该越小越好,这样就会有更少的光线与其相交。这跟人们的直觉很相似,包围盒应该尽量紧地包裹住原始的物体。

在光线跟踪算法的哪个部分插入包围盒求交测试比较合适呢? Arvo 和 Kirk【Arvo89】对所有的可能做了个精彩的综述,其中包括一个包围盒包裹一组物体的层次包围盒技术。这里集中考察改进基本的 hit()函数的方法,这样每个物体都将处理到。回忆前面图 12.4 中提到的基本程序 getFirstHit(),当前光线必须对场景中所有物体求交,对物体使用 getFirstHit()方法:

```
for(each object, obj)
{
    if(!obj->hit(ray, inter))continue;
      比较本次求交与当前最好的求交,等等
}
```

应该把包围盒求交测试放到 hit()里面来加速每次单独的求交过程。如果可以在整个求交计算之前快速判断光线与当前物体不会相交,那么整个光线跟踪算法的速度将大幅提高。将包围盒求交测试放到 hit()里面的一个好处是可以调整求交算法适应特殊的几何形状。

有很多方法可以在光线跟踪算法中创建和应用包围盒技术。一些很容易实现,另一些需要更多的程序量但有很好的效果。这里考察少数几个简单的方法,并说明它们是如何符合光线跟踪算法的逻辑思维的。

12.10.1　长方体和球体包围盒

包围盒技术中最常用的两种形状是长方体和球体:

- 球体包围盒:一个完全包裹物体的球体;由表达式(C,r)指定,球心为 C,半径为 r。
- 长方体包围盒:边与坐标轴平行的规则平行六面体。有六个参数指定:$(left,top,right,bottom,front,back)$。

判断一条光线和这些形状是否相交的算法速度是很快。对于圆球,求交有一个对应的二次等式,应该判断其表达式是否为正(参见图 12.15)。而对于边与坐标轴平行的长方体,要判断光线与六个平面是否相交(或者更少,如果足够幸运),然后判断区间(t_{in},t_{out})是否为零(参见图 12.23)。与每个平面的求交是相当快的,因为不需要点积运算。

进行包围盒测试的坐标空间可分为两类,世界坐标和通用坐标。通过坐标变换可以将每个物体放置到场景中。为求光线与物体的交,首先将光线通过坐标逆变换转换到通用坐标中,再求通用坐标中的光线和物体的交。但在这之前,先在世界坐标中对物体进行包围盒测试。如果光线与包围盒不相交,光线与物体的求交就不需要进行,这样就节省了坐标变换的花费。通常也可以在通用坐标空间对物体进行包围盒求交测试,如果光线与包围盒不相交,则可以避免光线与物体的求交运算。

世界坐标中的包围盒可能并不是很适合当前的物体。图 12.54 展示了场景中进行光线跟踪绘制的一个圆锥体。简单起见,图 12.54(b)用二维显示了圆锥体被长方体和圆球体包裹的情形。如果圆锥体细而长,圆球包围盒就会不适合,利用它来进行求交测试会出现很多

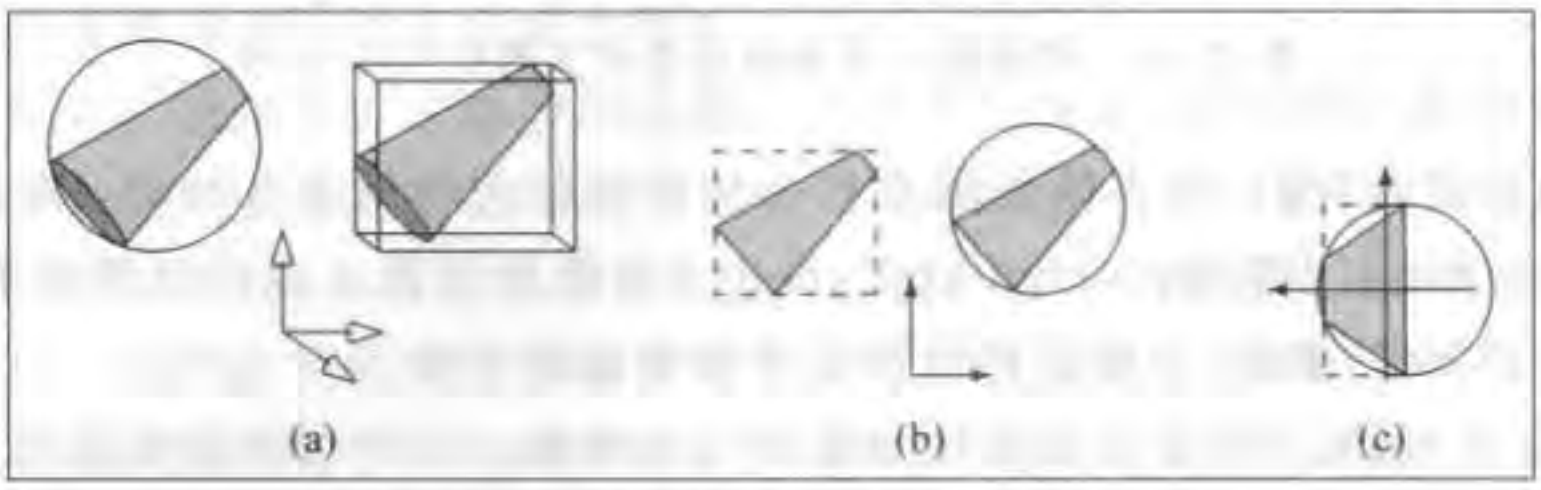


图 12.54　世界坐标和一般坐标中的包围盒

的错误警报(光线与包围盒相交但不与物体相交)。而长方体包围盒却能有效地包裹这样细且长的圆锥体,除非圆锥体旋转之后与坐标轴不平行。后面的联系请你对不同情况的错误警报做一个估计。

通用坐标中的包围盒通常会更适合物体。图12.54(c)展示了通用坐标的圆锥体与它的圆球包围盒和长方体包围盒。圆球包围盒跟物体很适合,长方体包围盒更紧致地包裹了物体。

包围盒求交测试是在世界坐标中进行还是在通用坐标中进行是有一个平衡的。在世界坐标中求交速度比较快,因为它不需要对光线进行坐标逆变换,但是通常包围盒对物体的包裹不够紧致。在通用坐标中求交需要计算通用坐标中的光线,但它的包围盒通常更适合物体。很难说前者与后者哪个更好。同样圆球包围盒和长方体包围盒的好坏也是无法判断。人们通常是在样本场景中对一个或多个方法进行时间测试,然后决定在他们的光线跟踪算法中采用何种方法组合。对一个拥有100个随机放置的圆锥体的场景进行测试的结果是,在世界坐标空间中使用圆球体包围盒会节省27%的光线跟踪时间,使用其他任何组合的时间花费都会比这个高。但是后面会看到,当物体的形状越来越多的时候,结果会很不一样。

图12.55是将包围盒求交测试集成到圆锥体情形的hit()里的伪代码。这里的包围盒是圆球体,并且分别在世界坐标系和通用坐标系中进行测试。在实际的算法中,只需要用到一种测试。函数rayHitsSphereExtent()用来判断光线与圆球包围盒是否相交,如果不相交,程序从这里提前退出hit()函数。这个函数首先被用来与世界坐标中的圆球包围盒worldSphereExtent()求交。如果光线与其相交,对光线做一个坐标变换。函数被第二次调用,用来判断光线是否与通用坐标中的圆球包围盒相交。同样,如果不相交,则退出求交函数,光线与圆锥体的求交将被跳过。

```
bool TaperedCylinder::hit(Ray &r, Intersection &inter)
{
    if(!rayHitsSphereExtent(r, worldSphereExtent)) return false;

    Ray genRay;                          //一般坐标中的光线
    xfrmRay(genRay, invTransf, r);       //花费高的

    if(!rayHitsSphereExtent(genRay, genSphereExtent))
return false;
…光线与包围盒相交;必须做花费很高的光线与景物的求交测试…
}
```

图12.55 圆锥体的圆球包围盒求交测试hit()函数

若先前经验告诉我们,长方体包围盒将会比圆球体包围盒更加有效,那么上面的函数rayHitsSphereExtent()将用rayHitsBoxExtent()函数来代替。也可以将两种包围盒的测试函数都放入到hit()里面,虽然这样会影响求交测试的速度。

这样,为hit()函数选择测试的方法就显得尤为重要。有些选择是明显的。对于一个球体的物体来说,没有必要对其进行通用坐标系下的球体包围盒求交测试(为什么?),类似地,

没有必要对长方体物体进行通用坐标系下的长方体包围盒求交测试。平面没有包围盒，因此函数 Plane::hit()不包含函数过程。为正方形物体建立包围盒是有意义的，而直接对这样的物体求交也很简单以至于用包围盒测试很少或者不会带来效率上的提高。另一方面，对于一个网格物体，它由大量的平面组成，对它直接求交花费很大，所以包围盒求交测试可以带来效率上的巨大提高。

如何实现包围盒求交测试

不同几何形状的物体的包围盒是如何计算的？每个包围盒求交测试是如何执行的？自然的方法是将包围盒的信息储存到它所对应的物体里，因此在 GeoObj()类里加入一些域的信息(参见本书的相关网页)。前面提到的四种包围盒求交方法经常会被用到，可以为每种方法加入一些域信息：

```
SphereInfo genSphereExtent, worldSphereExtent;
Cuboid genBoxExtent, worldBoxExten;
```

类 SphereInfo(参见本书的相关网页)记录了球体的两个信息：center 记录球体包围盒的球心位置，radSq 记录了半径的平方。可以只记录球体的半径信息，但在求交的时候只会用到半径的平方。类 Cuboid 包含了 6 个信息：left、top、right、bottom、front、back。长方体包围盒沿 x 轴从 left 延伸到 right，沿 y 轴从 bottom 延伸到 top，沿 z 轴从 front 延伸到 back。

在光线跟踪之前，物体列表建立之后，列表中的每个物体都储存了对应的包围盒信息。当光线跟踪进行时，一些绑定在 hit( )函数内部的函数 rayHitsSphereExtent( )和 rayHitsBoxExtent()将被调用，如图 12.55 所示。光线与球体包围盒的求交测试很容易实现(见练习)。

考察对应的二次等式的判别式是否为正。计算判别式需要做十一次乘法，因此这个测试也不是完全不花费时间的。

rayHitsBoxExtent()是 Cube 类中 hit()函数里的一个自适应函数，它使用了类 Cuboid 中的长方体包围盒结构。这个函数调整了程序运算的最快速度。

构造圆球体和长方体包围盒

每个给定的物体都需要计算球体和长方体包围盒。通用坐标系下每个物体的包围盒只需要计算一次，但世界坐标系下的包围盒每次都需要重新计算，因为每个物体的不同实例都有不同的仿射变换。仿射变换使得包围盒的计算更加复杂。怎样为一个经过复杂的伸缩和旋转变换的圆锥体构造圆球体包围盒呢？

一个特殊并且直接的方法是使用**点束**。点束是一些点，它们的凸包包裹着物体。空间中固定点集的凸包的直观的想像是：在这些点外面放置一个气球，然后让这个气球向这些点慢慢萎缩，最后得到的多面体便是它对应的凸包。对于一给定点束，很容易可以得到包裹它的一个球体。当然这样的球体也包裹着对应的物体。经过变换的物体可以通过这样得到它的凸包：对点束里的每个点做同样的变换，然后用这些变换过的新点求出变换过的凸包。因为仿射变换保持物体的相对位置不变，所以如果物体 A 在包围盒 B 之内，那么经过变换

的物体 $T(A)$ 一定在经过变换之后的凸包 $T(B)$ 之内。

因此需要对每种形状建立点束。长方体和平面的点束的建立很简单：直接采用它们的顶点。同样，网格也很简单：将其顶点作为点束即可。

对球体和圆锥体建立对应的点束要复杂一些，因为它们由一系列的对称关系所定义，而不是由很多点来定义的。一个合理的做法是求出这样的形状的一个紧紧包裹住它的多面体，然后用这个多面体的顶点来作为其点束。图 12.56(a)里一个圆锥体被一个底面为六边形的棱柱所包裹。为了紧紧包围圆锥的底面圆，棱柱的上下底面六边形的半径必需分别为圆锥上下底面圆半径的 4/3。注意不同的圆锥体实例有不同的六边形棱柱来包裹，这样求解点束的方法也只适用于圆锥体和圆柱体。图 12.56(b)是一个二十面体包围一个圆球的示意，它的 12 个顶点被用来作为圆球的点束。这个二十面体的顶点必须放在离球心大约 1.26 倍半径的地方，这样二十面体可以比较好地包裹圆球体。

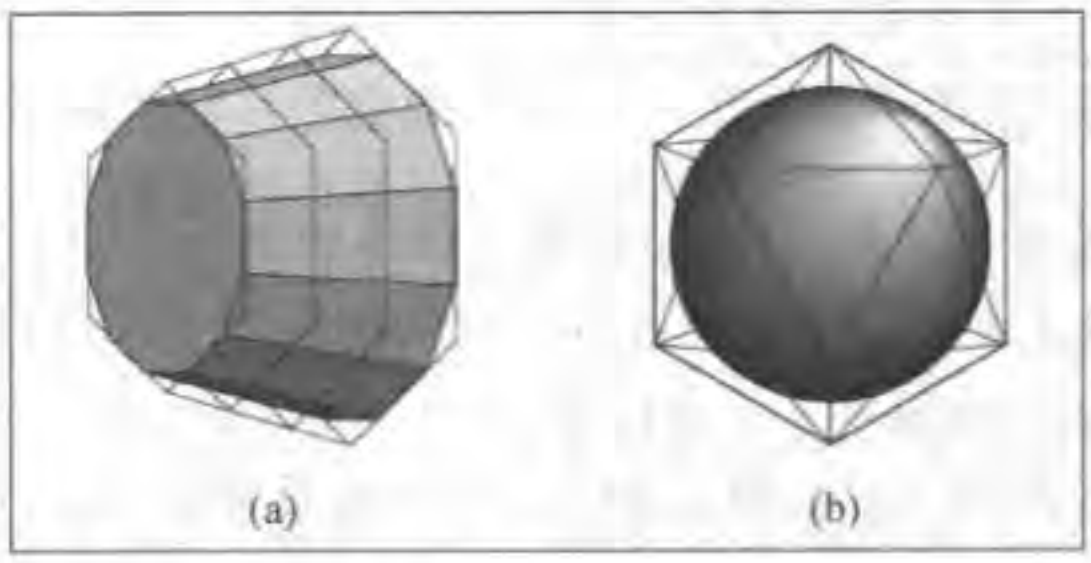


图 12.56 圆锥体和圆球体的点束定义

每种形状的点束可以储存在一个简单的数据结构 PointCluster 中，它记录点束中点的个数，并且用一个数组 pt[]记录点的坐标值。在光线跟踪算法之前的预处理中，每种形状都调用一个对应函数 makeExtentPoints(PointCluster& clust)。

一旦物体的点束建立成功，为这个物体创建长方体和圆球体的包围盒就很方便了。长方体包围盒的三个方向上的边的两个端点分别采用点束里面的点在对应方向上的极大值和极小值，因此可以通过一个单循环来求得。可以通过编写一个程序 void makeBoxExtent (Pointcluster& clust, Cuboid& cub)来分析点束，生成一个 Cuboid 类型数据结构并将其储存到对应物体中。

构建圆球体包围盒稍微复杂一些。点束的中心作为圆球的中心，因为这是一个很好的中心点。计算点束的中心，将点束中的点在三个方向上的分量分别求和，然后除以点的个数。而圆球体包围盒的半径选择为点束中的点到点束中心的最大距离。这样，函数 void makeSphereExtent (Pointcluster& clust, SphereInfo& sph)就很容易编写。

最后，构建世界坐标下的长方体包围盒和圆球体包围盒是很简单的。只需将通用坐标系下点束中的每个点都坐标变换到世界坐标系下，然后用上面同样的方法即可求得。

练习

练习 12.10.1 光线与圆球体包围盒的求交测试

(a) 求证包裹圆柱体(半径为 1)的最小圆球包围盒的球心是(0,0,0.5)，半径为 $\sqrt{1.25}$。

(b) 求证这样的圆球体包围盒可以用隐式 $F(x,y,z)=x^2+y^2+\left(z-\frac{1}{2}\right)^2-1.25$ 来表示。

(c) 证明求解等式 $F(S+\mathbf{c}t)=0$ 可以转换为求解 $At^2+2Bt+C=0$，其中 $A=|\mathbf{c}|^2$，$B=c\cdot S$，$C=|S|^2-1.25$，这里定义 $S'=(S_x,S_y,S_z-0.5)$。

练习 12.10.2　立方体包围盒测试的实现

请在函数 Cube::hit()里面加入函数 bool rayHitsBoxExtent (Ray& ray, Cuboid& cub)用来测试给定的光线是否与用 cub 描述的包围盒相交。这在不改变逻辑的基础上简化了函数 hit()。考察立方体每个面的 numer 和 denom 值是如何依赖于 cub 里面的数据的。例如，令 numer=cub. top-ray. start. y；denom=ray. dir. y。因为立方体的每个面都可能与光线相交，候选的区间(t_{in}，t_{out})不断地被更新，如果此区间长度变为零，函数退出并返回值 true。

练习 12.10.3　圆球体的点束

考察图 5.28 中的二十面体，这个二十面体的顶点要缩放多少倍才可以正好包裹那个圆球体。(缩放因子 $f=1.071$ 够不够?)

练习 12.10.4　创建圆球体和长方体包围盒的程序

编写从给定点束创建圆球体和长方体包围盒的程序：void makeSphereExtent (Pointcluster& clust, SphereInfo& sph)和 void makeBoxExtent (Pointcluster& clust, Cuboid& cub) 。

12.10.2　投影包围盒

光线跟踪中还有另外一种包围盒技术，它使得光线跟踪算法的速度得到很大的提高。与在三维空间中的长方体和圆球体不同的是这种包围盒是屏幕上的一个矩形区域。

物体的**投影包围盒**指的是屏幕上一块包裹物体投影的矩形区域，如图 12.57(a)所示。图 12.57(b)展示了投影包围盒的更多细节。投影包围盒记录四个坐标信息{*left*，*top*，*right*，*bottom*}。

在光线跟踪算法中使用投影包围盒十分简单。每条光线在一个特定的屏幕行列值上穿过，记为(r，c)。如果(r，c)在物体投影包围盒的外面，光线肯定不与物体相交。例如在图 12.57(b)中，如果 c 小于 244 或大于 296，或者 r 小于 237 或大于 319，那么这条光线与物体不相交。

因为投影包围盒与物体的相机和人眼的位置密切相关，它只适用于**眼光线**即从眼睛发出的光线。后面的章节将会介绍如何为阴影、反射光和折射光生成光线，因为这些光线从场景中任意点发出，没有屏幕可以被这些光线投影。但对眼光线，这个技术可以运行得很好，并且极大地提高光线跟踪算法的效率。

与长方体和圆球体包围盒一样，在预处理阶段为每个物体求投影包围盒，并且把它与对应的物体储存在一块。这样需要在 GeomObj 里面加入一个域 IntRect screenExtent 结构。还需要扩展类 Ray 是的光线知道它在哪里与屏幕相交。这里再为其提供一个信息：**递归级**

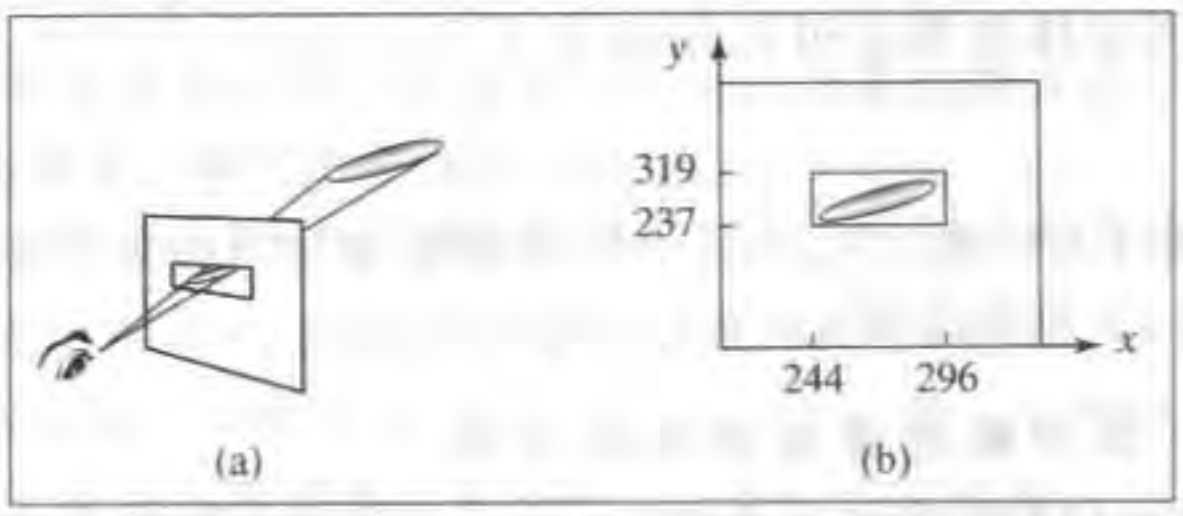


图 12.57 景物的投影包围盒

别。后面将知道,这在反射和折射中占有很大一部分。这里简单用来判断光线是否是眼光线。眼光线的级别是 0。

在长方体和圆球体包围盒测试之前,光线跟踪算法的函数 hit()已开始执行一个新的测试。图 12.58 展示了在类 TaperedCylinder 中这个测试放置的位置。同样的测试被用于除平面外的不同物体。与图 12.55 比较。这个测试检测光线是否为眼光线,如果是,接着检测光线与屏幕的交点是否在与物体储存在一块的投影包围盒之内。如果光线与屏幕的交点在投影包围盒之外,光线与物体不相交,函数 hit()返回 false。这个测试速度很快。

```
bool TaperedCylinder::hit(Ray &r, Intersection &inter)
{
    if(r.recurseLevel==0 &&
r.col<projExtnt.left ||
r.col>projExtnt.right ||
r.row<projExtnt.bottom ||
r.row>projExtnt.top)return false;     //与投影包围盒不相交
    if(!rayHitsSphereExtent(r, worldSphereExtent)) return false;
    求出逆变换,求出一般坐标下的光线
    if(!rayHitsSphereExtent(genRay, genSphereExtent))
return false;
…光线与包围盒相交;必须做花费很高的光线与景物的求交测试…
}
```

图 12.58 带有圆锥体 hit()方法的圆球体包围盒测试

计算投影包围盒

必须在预处理阶段为场景中所有的物体计算投影包围盒。给定物体的点束,相应的投影包围盒很容易计算。通用坐标系下的点束通过坐标变换转换到世界坐标下,和前面的世界坐标下的圆球体、长方体包围盒一样。图 12.59 是变换后的圆锥体的 12 个点的点束。

每个点都投影到相机所对的平面上。给定场景中的一个点 p,可以通过点 p 的坐标和相机的位置信息计算出它的投影 p'。这里记录了投影点 p'的行值 r 和列值 c。

图 12.60 给出了光线跟踪场景中每个物体的投影包围盒。为了调试方便,一些程序员喜欢在光线跟踪算法开始之前求出投影包围盒。这些长方形在最终像素值绘制之后将会消失。

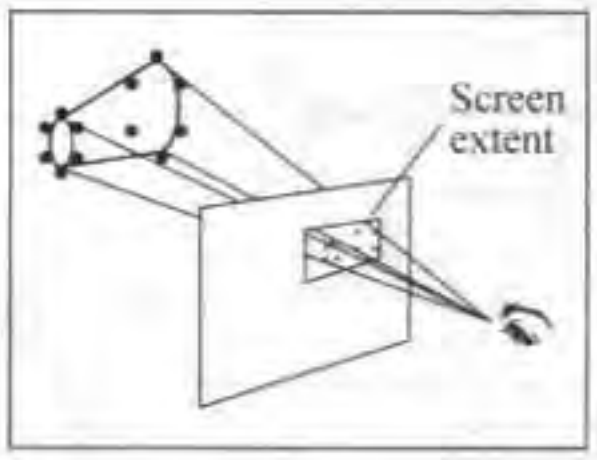


图 12.59　建立投影包围盒

图 12.60　光线跟踪场景中的可见投影包围盒

12.10.3　其他光线跟踪加速方法：二叉空间剖分树技术

我们知道，由于场景中有大量的光线和大量的物体，光线跟踪算法是十分耗时的。很明显，这么多光线与物体中，每条光线和每个物体都必须进行求交测试，这是耗时的关键所在。包围盒技术的运用一定程度上提高了算法的效率，但是对二级光线的帮助并不大。

因此需要其他的方法来加速光线跟踪。加速光线跟踪的一个很好的方法是二叉空间剖分树(BSP)方法【Arvo88】。它为光线跟踪构建了一个多层次的数据结构，通过快速决定哪些物体将不与光线相交排除了大量的求交运算。BSP 树是一种可以快速创建和遍历的数据结构。BSP 树通过某种顺序访问树的每个节点并且做当前节点与光线的求交测试来遍历。在每个节点，如果物体与光线相交，只需通过一个粗糙的求交测试即可判断。

构建 BSP 树

为了图示方便，这里我们通过二维例子介绍 BSP 树。因为树的组织与它所包含的物体的维数无关，所以把 BSP 树从二维推广到三维是很直接的。

图 12.61 展示了平面中的三个多边形和一个处在特定点的眼睛位置。问题是：从光线原点，可以看到什么？如果在三维中想象，一个观察者站在三块高出底面很多的大石柱之前。大石柱的横截面如图 12.61 所示。大石柱的哪些部分是可见的，哪些部分是不可见的？画出这个场景的草图。给每个边赋一个值，标出它的朝外的法向量。我们希望将每条边都插入到一棵树里面，并且满足平面被不重叠的多边形区域所分割开。

为这些边建立二叉树，先取一条边，记为#1。将此边的两端延长，边#1 将平面(多边形所在的)分成两部分，外部(边的朝外的法向量所指的半平面)和内部。我们将边#1 放置到二叉树的根部，如图 12.62(a)所示。接着将其他边按一定顺序都插入进去。考虑边#2，将其与边#1 比较。如果边#2 在边#1 的外部，那么将其放置在边#1 的右边树枝，标记成 outsideOnes。否则，将其放置到左边树枝，标记为 insideOnes。当每条边都插入进二叉树后，通过判断每个节点的走向，从上到下的路径就确定了。新边如果在当前节点的外部，则沿着节点往标记 outsideOnes 的子节点走，否则选择标记 insideOnes 的子节点，直到此新边成为二叉树的一片新叶子为止。

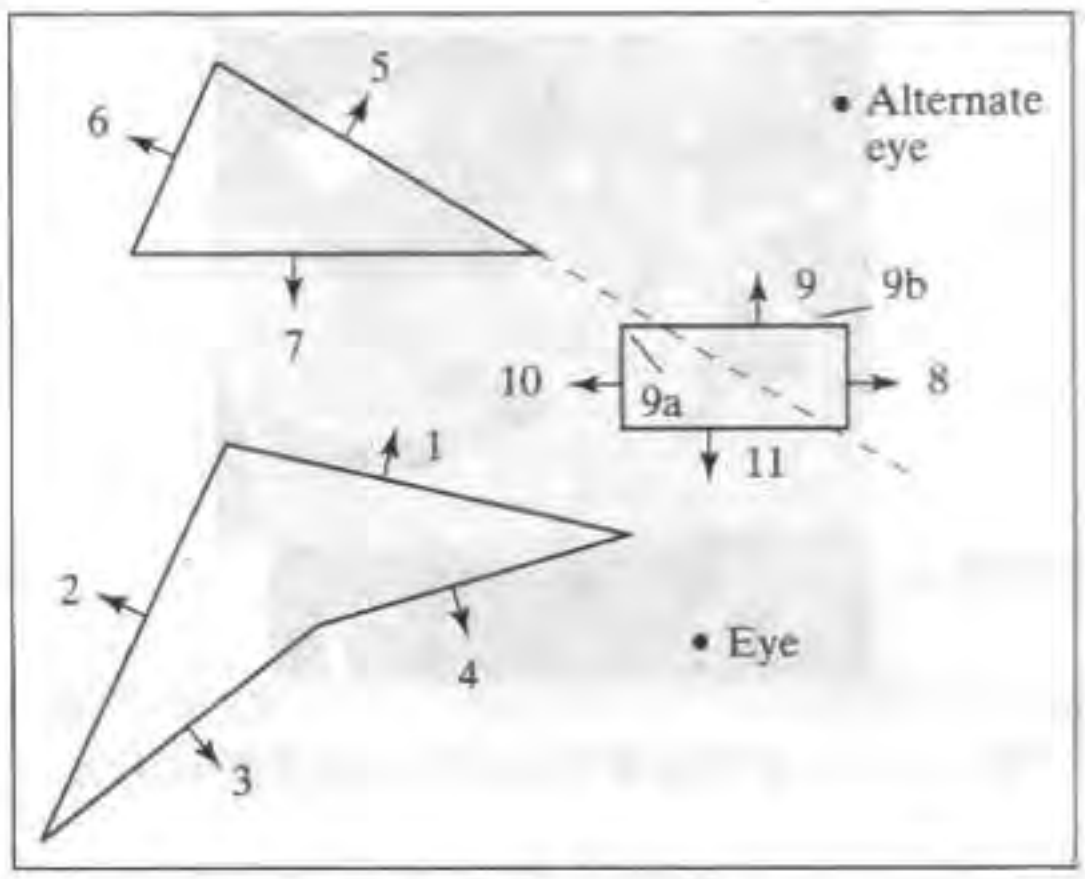


图 12.61 一些将被插入到一棵树结构的二维边

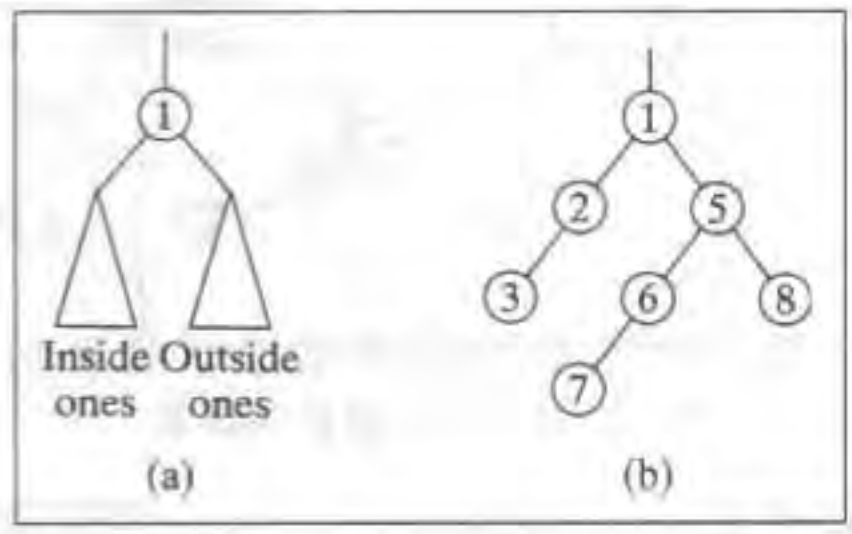


图 12.62 构建 BSP 树

图 12.62(b)给出了二叉树插入八个节点#1，…，#8 之后的情形。边#2 在边#1 的内部半平面，因此它被放置在 insideOnes 树枝。边#3 同时处在边#1 和边#2 的内部，因此它沿着图示方向往下走。边#4 没有在图中画出来，它应该处在图中的那个位置呢？

当我们插入边#9 时，会碰到新的情况。它处在边#1 的外部，但是在边#5 的两侧。这样，它被边#5 **分割**成两段，#9a 和#9b，如图 12.61 所示，每段可以继续往下走。边#9a 处在边#5 的内部，因此它往左边走，并且与边#6 和边#7 做测试判断；边#9b 处在边#5 的外部，边#8 的外部。边#10 和边#11 也按照同样的规则往下走，它们不需要分割成两段。(最终的二叉树是怎样的结构？)

BSP 树用特殊的方法将边列了出来，这样对每个边可以很快速地(通过它在二叉树中的位置)指出它所处的空间位置(例如，边#6 在边#1 的外部和边#5 的内部)。

而边插入的顺序对最终二叉树的形状有着极大的影响。一些顺序的选择方法可以使得最终的二叉树的形状宽并且“浅”(从树的根部到叶子只有很少数量的节点)，另外一些使得最终形状窄并且“深”。浅的树处理起来效率更高。BSP 树的最差的情况是从上到下只有一条单一的链。

三维场景中的 BSP 树

这个方法很直接地就可以推广到一个由三维多边形网格组成的场景中。考虑图 12.63(a)所示的多面体。为它的每个面都标记了数字，并指出了每个面相应的朝外的法向量。图 12.63(b)给出了同一多面体的另一个角度视图。如果按顺序将面#1，#2，…，#8 插入到二叉树，将得到一棵图 12.63(c)所示的树。(一个面按二维的方法往下走：如果它在当前节点面的外部半空间，那么它将往树枝 outsideOnes 方向走，继续直到最后。)面#7 是里面唯一需要分割的：它被面#3 分割成#7a 和#7b，如图中虚线所示。(有没有这样的插入顺序，使得最终没有面被分割的？)

这里我们看到了 BSP 树是如何完美地绘制一幅场景的。

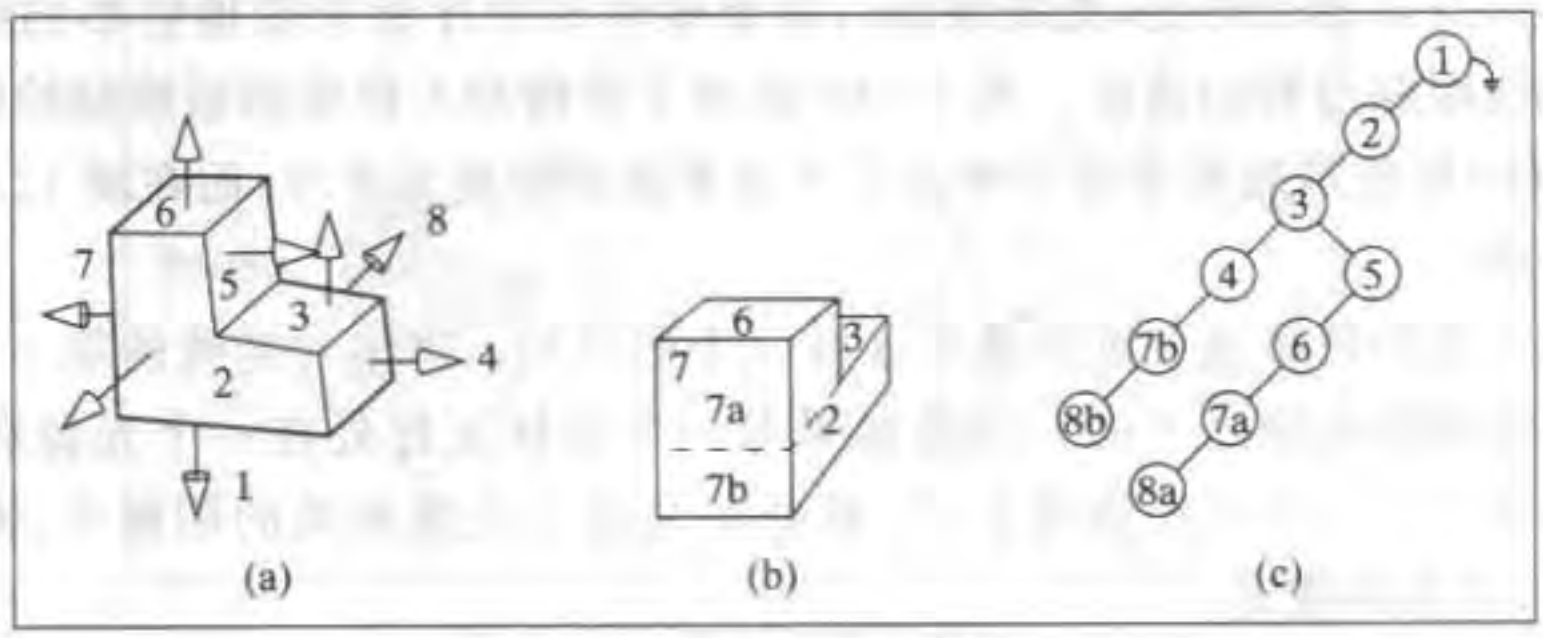


图 12.63　构建 BSP 树

横穿二叉树(用于光线跟踪)

首先应该决定哪个物体最先被光线碰撞。我们知道光线与物体相交时,同一层的节点中总是左边的节点先与光线相交。因此为了找到最小的时间,可以简单地从树根开始跟踪,每次都移动到下一层的左边子节点。最后移动到达图 12.63 所示的标记为 8b 的叶子,这片叶子就是最靠近眼睛的物体。

图 12.63(a)中三维场景的树(图 12.63(c))通过这种方法横穿,可以将此多面体正确地绘制出来。图 12.64(a)中的箭头表示访问图 12.63(c)中的 BSP 树节点的顺序,对应的是图 12.63(a)中多面体的特定朝向。图 12.64(b)给出了光线碰撞物体的升序排列时间的次序。下划线所指的面是不用绘制的面。

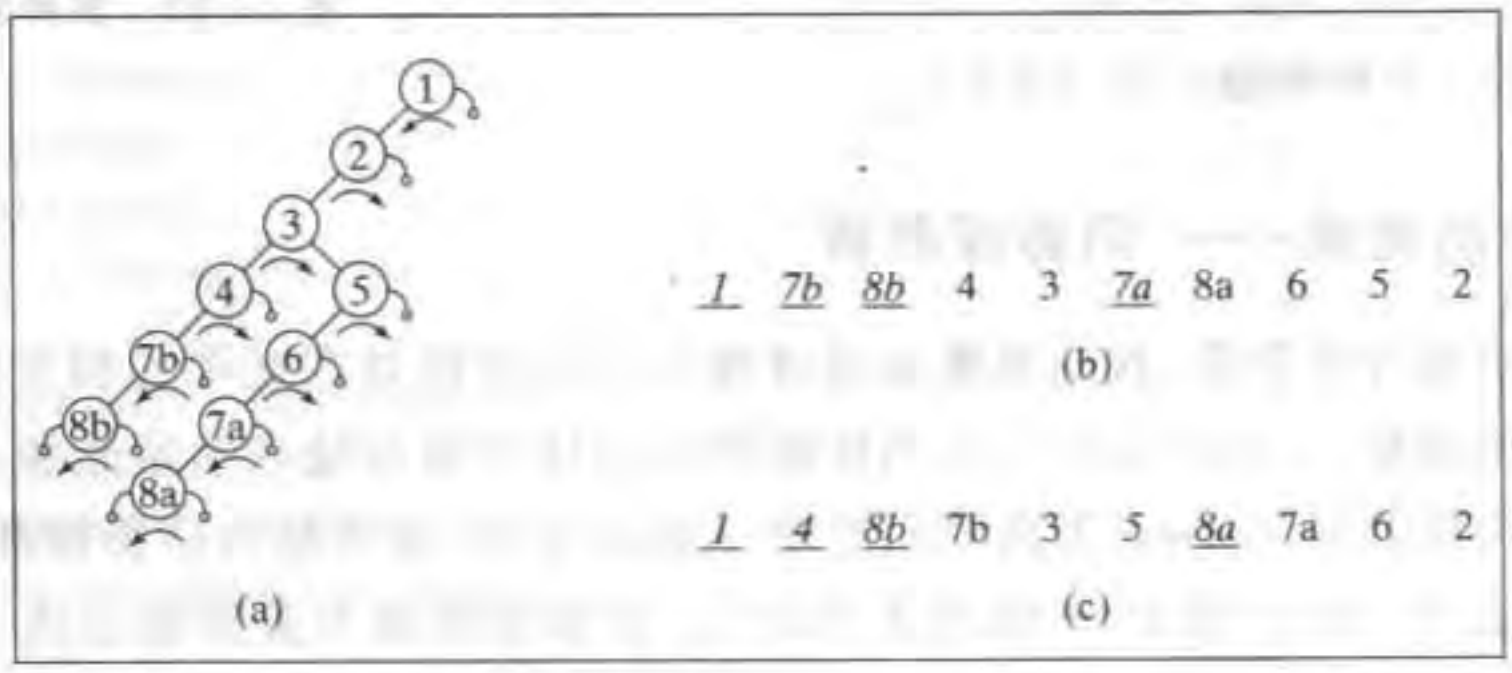


图 12.64　给定眼睛位置,面的绘制次序

有关光线跟踪算法中的 BSP 树技术的详细介绍,请参考"Graphics Gems III"【Kirk92】中的"Use of Residency Masks and Object Space Partitioning to Eliminate Ray-Object Intersection Calculations"和"Ray Tracing with BSP Trees"两篇文章。

12.11　添加阴影

为物体添加阴影使计算机绘制出来的图像更加真实。在第 8 章中我们了解到 OpenGL 提供了一些绘制阴影的基本机制,但是这些工具使用起来很不方便,而且只能在一些特定情

形下产生阴影。与其相反的是,光线跟踪只需要很少的程序量来获得阴影的绘制。不幸的是它降低了光线跟踪过程的速度。图 12.65 给出了使用和不使用阴影绘制的同一场景。注意在图 12.65(b)里很难判断多面体球离它下方平面的距离是多少,而在图 12.65(a)我们一眼就可以看出来。

到现在为止我们计算光亮度是基于这样一个假设的:物体与光线的第一碰撞点 P_h 总是沐浴在各种光源的光照之下的。但是如果有一个物体正好处在一个光源和 P_h 之间,上面的假设就不成立了。在这种情况下 P_h 就处在与这个光源相关的阴影中,漫反射和镜面反射都将消失,只有环境光。

图 12.66 给出了多种阴影的情形。从点 P 可以看到光源 L_1,因此点 P 不在光源 L_1 相关的阴影里面。但 P 处在与光源 L_2 相关的立方体的阴影里。此外,光源 L_3 被物体自身所阻挡,这时 P 处在自己所在物体的阴影中,这被称为**自身阴影**。对这样的场景结构,P 可以直接看到的唯一的光源是 L_1。

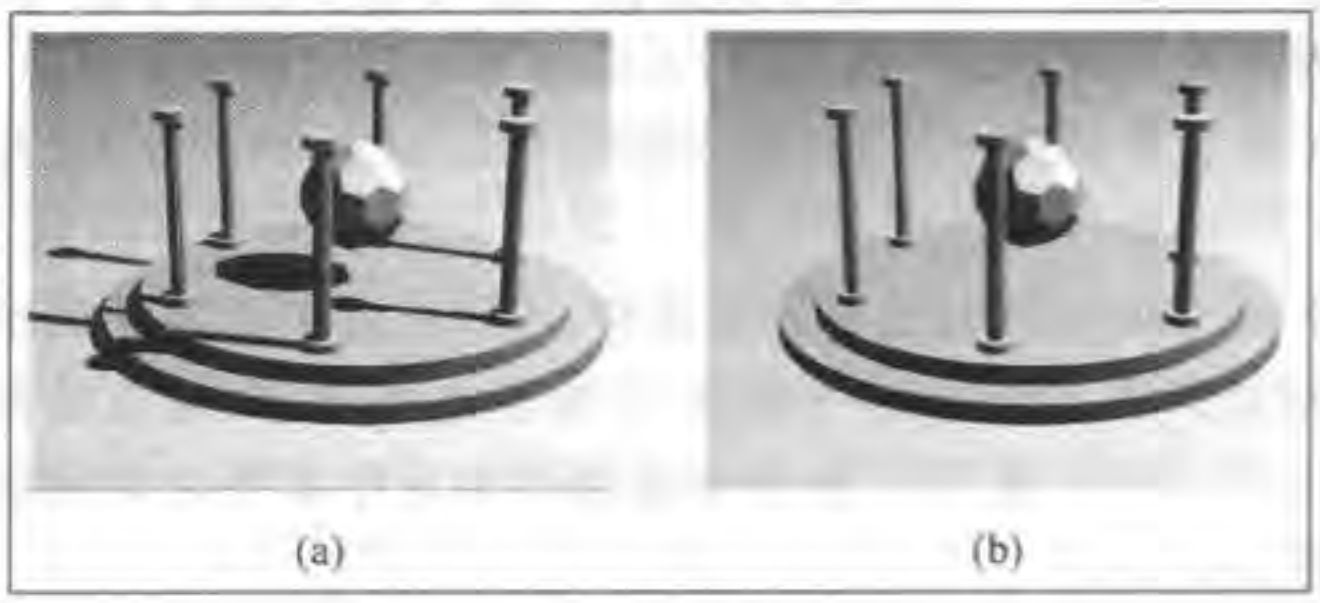


图 12.65

(a) 有阴影的渲染;(b) 无阴影的渲染

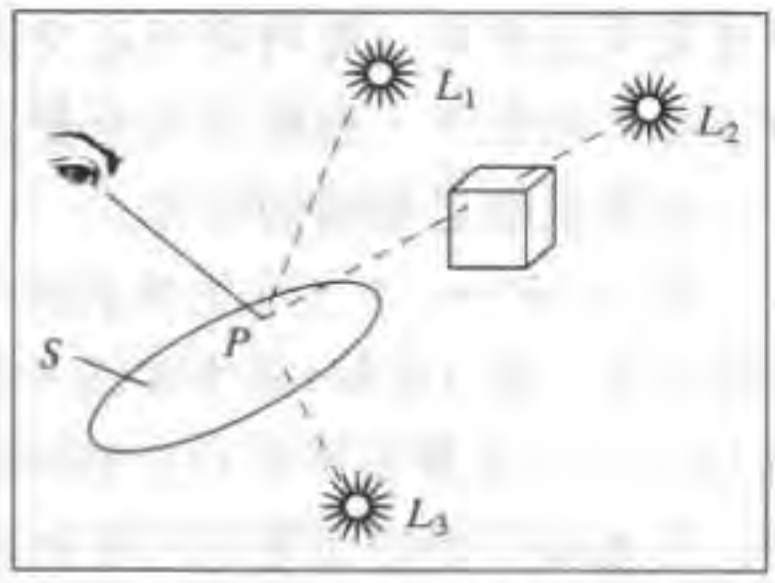


图 12.66 多种阴影情形

使用额外的光线——"阴影探测器"

为了能准确地计算阴影,我们需要知道碰撞点什么时候处在光源的相关阴影中。因此我们需要这样的函数,*isInShadow*(),当任何物体的任何部分处在给定光源与碰撞点之间时,返回 *true*,否则返回 *false*。为此我们使用一条新光线,通常称为**阴影探测器**,它在参数 $t=0$ 时从碰撞点 P_h 发出,在 $t=1$ 时到达光源 L。阴影探测器有参数表达式 $P_h+(L-P_h)t$。为了考察它是否碰撞到任何物体,所有的物体都将被扫描一遍,每个物体都与此光线执行一次求交测试。如果在 $t=0$ 和 $t=1$ 之间有物体相交,isInShadow()返回 true。这看上去要求比较高,但使用光线去探测场景的光线跟踪基本机制使得这变得简单。

直接简单地使用探测器会隐含一个严重的问题,物体自身的阴影。如果阴影探测器从 P_h 出发,那么探测光线与物体总是在位置 $t=0$ 处相交!于是函数 isInShadow()总是返回 true 值,这显然是错误的。一些人试图通过严格限制交点的参数 $t>0$ 这一条件来解决这个问题。但是,如图 12.67(a)所示,如果场景中光源与眼睛分别处在一片长方形或平面的两侧,这个真正的相交情况就会被误判。从 P 出发指向 L_1 的探测器正确地报告了没有交点,但是从 P 指向 L_2 的探测器也将给出没有交点的报告,然而点 P 在长方形自身的阴影之下。为什么这种异常不会在其他形状例如球体和锥体上出现?

一个比较可靠的策略是调整函数 isInShadow()里面的阴影探测器,如图 12.67(b)所

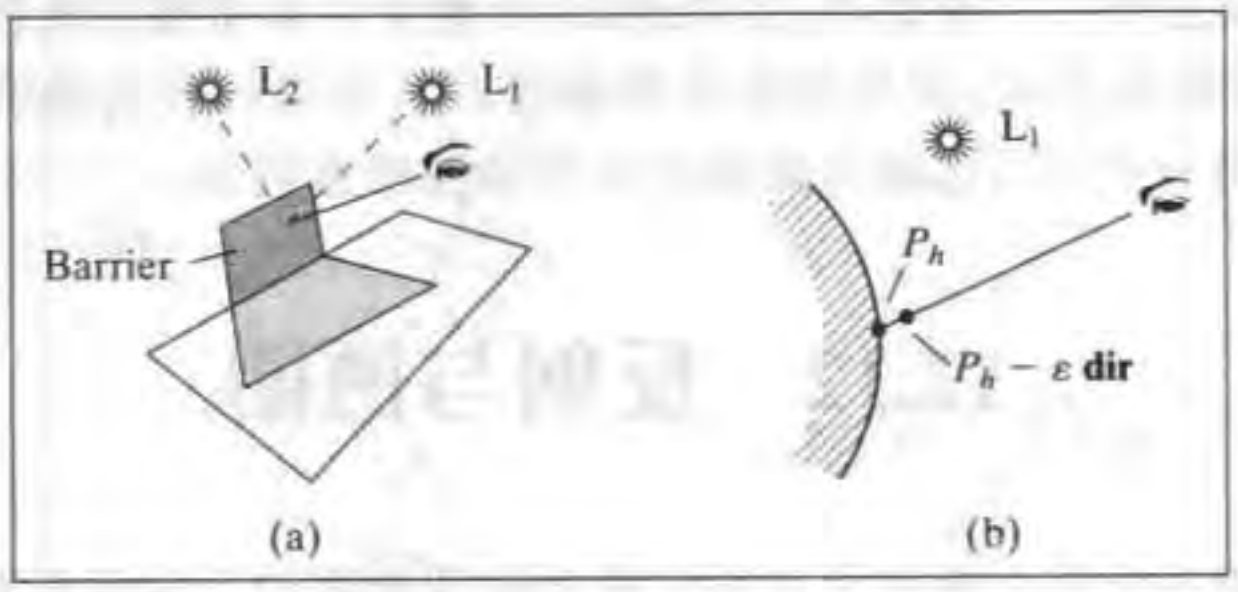


图 12.67　阴影探测器策略

示。让阴影探测器的开始点朝人眼方向移动一个微小的距离。如果光线的方向为 dir，碰撞点为 P_h，阴影探测器被移动到 $P_h-\varepsilon$dir，其中 ε 是一个很小的正数。这将开始点轻微移动到与光线碰撞的物体的靠眼睛侧。在阴影探测器中使用这样的开始点，在物体位置 t=0 时没有交点，问题解决了。

这种方法通过下面的方式嵌入到 Scene::shade(ray)函数中。当函数 getFirstHit()返回最佳的碰撞记录时，碰撞点与此点的法向量都确定了。光线的环境光颜色也确定了下来。接着一条探测光线从开始点 $P_h-\varepsilon dir$ 处发出。它的 recurseLevel 被设置为 1，这样投影包围盒将在不同的函数 hit()中禁止掉。

接着通过表达式 L. pos-feeler. start 为每个光源 L 计算探测方向，并且调用函数 isInShadow (feeler)来检查探测器是否碰撞到物体。如果发生碰撞，当前光源的漫反射和镜面反射的光线亮度计算将跳过。下面的伪代码是阴影相关的程序部分：

```
feeler.start=hitPoint-ε ray.dir;
feeler.recurselevel=1;
color=ambient part;
    for(each light source, L)
    {
        Feeler.dir=L.pos-hitPoint;
        If(isInShadow(feeler)) continue;
        color.add(diffuse light);
        color.add(specular light);
}
```

下面的代码是 isInShadow()函数的一种可能的实现。它简单地从物体列表里查看碰撞，如果有碰撞，则返回 false。如果没有碰撞发现，则返回 true。

```
bool Scene::isInShadow(Ray& f)
{
    for(GeomObj * p=obj; p; p=p->next)
        if(p->hit(f) return true;
    return false;
}
```

它对每个不同类型的物体使用一个简化版本的 hit()函数，它不需要构建碰撞列表。这个版本的 hit()函数与前面我们使用的有三点不同：

(1) 它只接受时间从 0 到 1 之间的碰撞，因为在光源另外一侧的物体不会产生阴影。

(2) 如果检测到这样一个碰撞,它立即返回,不需要计算碰撞的任何数据信息。

(3) 不能使用投影包围盒,因为阴影探测器可以从场景的任何地方发出。因此它必须为每种物体类型使用一些专门选取的球体或长方体包围盒组合。

12.12 反射与透明

光线跟踪算法的一个优点是它可以很简单地处理光线的反射和折射。这使得我们可以构建异常真实的场景,包括镜子、玻璃金鱼缸、透镜等。在到达人眼之前,光线可以在多个光亮的曲面上进行反射,或者同时进行反射与折射。每个这样的处理都需要构建额外的跟踪光线。

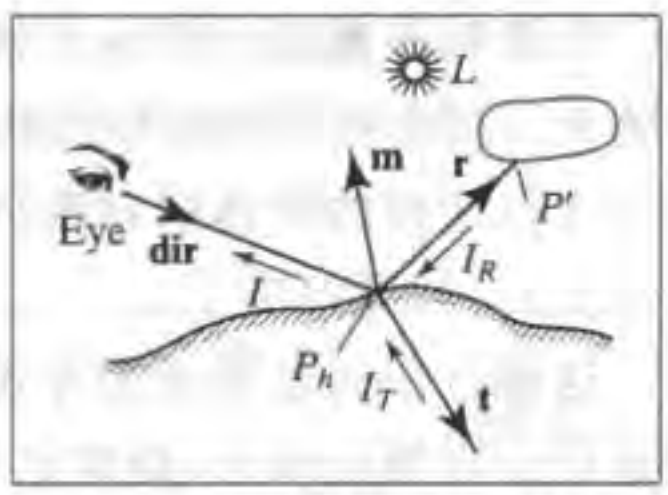


图 12.68 反射光与折射光

图 12.68 是一条从眼睛发出方向为 **dir** 的光线,与一曲面在 P_h 点相交。图示了二维情况的主要因素,这是可以接受的,因为反射与折射的属性使得所有的向量都在同一个平面上。所有的等式都可以推广到三维。当曲面为镜面或透明面或者两者皆有时,到达人眼的光线 I 可能由五个部分组成:

$$I = I_{amb} + I_{diff} + I_{spec} + I_{refl} + I_{tran} \qquad (12.42)$$

前面三项是熟悉的环境光、漫反射光和镜面反射光。漫反射和镜面反射部分由环境中 P_h 点可见的光源所产生的。I_{refl} 是反射光,由从方向 $-\mathbf{dir}$ 入射到点 P_h 的光线 I_R 产生的。方向是由入射角与反射角相等决定的,因此 **r** 由下面的等式给出:

$$\mathbf{r} = \mathbf{dir} - 2(\mathbf{dir} \cdot \mathbf{m})\mathbf{m} \qquad (12.43)$$

(其中我们假设 P_h 点法向量 **m** 已经归一化)。

相似地,I_{tran} 是光线传输的分量,它从光源 I_T 出发,沿着方向 $-t$ 到达点 P_h 其中一部分传输到透明材料内部。然后沿着方向 $-\mathbf{dir}$ 继续传输。折射的方向 **t**,取决于多个因素,有关细节我们下一节再讨论。

因为 I 是多种光的贡献之和,I_R 和 I_T 分别由其自己的五个分量组成:环境光、漫反射光等。如图 12.68 所示,光线 I_R 从 P' 出发到达 P_h 点,并在 P_h 点被人眼看见。为了计算 I_R,我们在点 P_h 构建沿方向 r 的第二条光线,找出与其碰撞的第一个物体,然后按等式(12.42)的重复计算光线分量。同样,I_T 的计算是通过构建一条 r 方向上的光线,求出与其碰撞的第一个物体,然后计算那里的光的贡献,诸如此类。

图 12.69 给出了对一条光线有所贡献的分量的数量是如何增加的。最终到达人眼的光线记为 I;它是很多个分量的组合,由很多个点的反射光和折射光以及很多个环境光、漫反射光和镜面反射分量组成。

如图 12.69(a)所示,I 由三个分量组成,反射分量 R_1、折射出来的分量 T_1 和局部分量 L_1(没有标记出来)。局部分量是 P_h 处的环境光、漫反射与镜面反射之和。局部分量的值取决于实际的光源;它们并不需要通过构建另外的光线来计算。环境光是模拟其他曲面上的漫反射与镜面反射对当前曲面的影响。R_1 是 R_3、T_3 和局部分量 L_3 之和。T_3 是 R_4、T_4 和 L_4 之和。每个贡献分量是另外三项之和,可能会出现无穷项。这样我们建议使用迭代的方法计算光亮度。

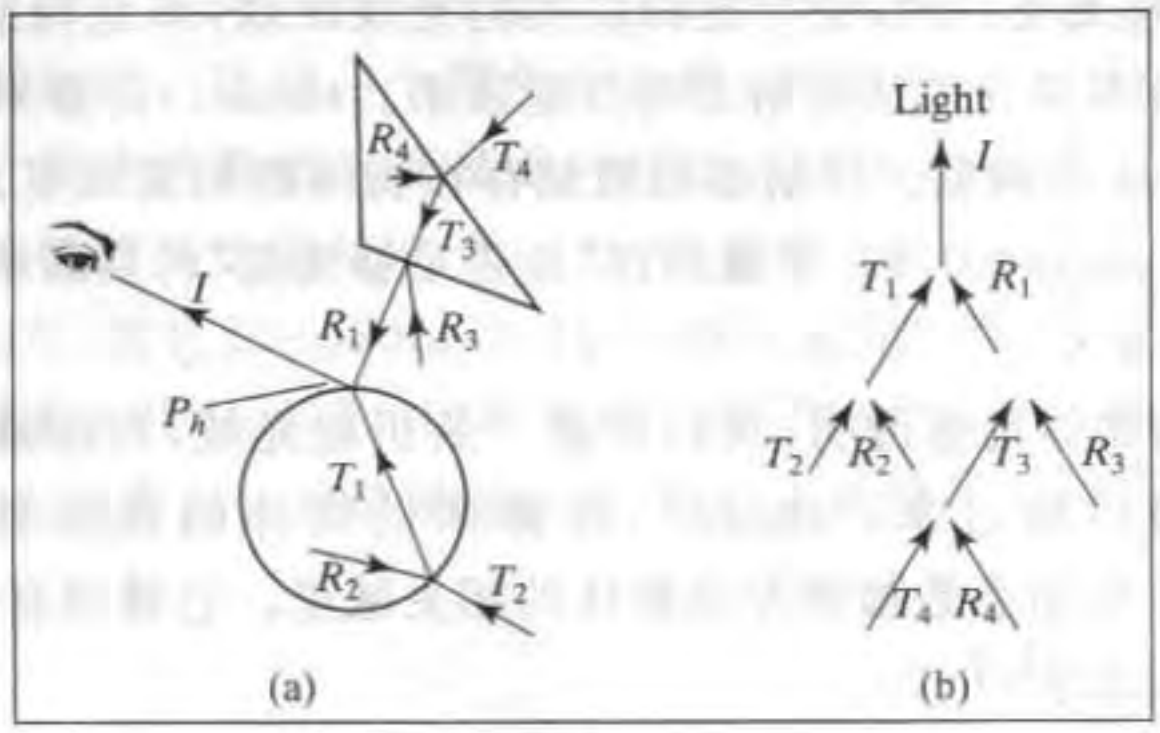


图 12.69　光线树

图 12.69(b)画出了由多个分量组成一棵树，树的左枝是折射分量，右枝是反射分量【Whitted80】。对每个节点，局部分量都必须添加上去，但是简单起见，这里没有画出来。

为了结合这几种视觉效果，函数 Scene∷shade()将被改成可以迭代地调用自己。图 12.70 给出了 shade()函数的结构框架，强调了在图 12.12 的 shade()函数中增加的部分。在正确的条件下，函数 shade()将调用自身两次来累计反射与折射光线的贡献。

```
Color3 Scene::shade(Ray& r)
Int recurseLevel=0;                          //设置迭代等级为 0
{
    计算第一次碰撞,构建碰撞信息 h
    Shape * myObj=(Shape * )h.hitObject; //指向碰撞景物的指针
    Color3 color.set(the emissive component);
    color.add(环境光分量);
    计算在碰撞点的归一化后的法向量 m
    for(每个光源)
        添加漫反射和镜面反射分量
    //下面添加反射与折射分量
    if(r.recurseLevel==maxRecursionLevel)
        return color;                        //不需要再迭代
    if(hit object is shiny enough)           //添加任何反射光
    {
        计算反射方向,构建反射光线 refl
        refl.recurseLevel=r.recurseLevel+1;
        color.add(shininess * shade(refl));
    }
    if(hit object is transparent enough)
    {
        计算折射方向,构建折射光线 trans
        trans.recurseLevel=r. recurseLevel+1;
        color.add(transparency * shade(trans));
    }
    return color;
}
```

图 12.70　函数 shade()的迭代结构

如果碰撞物体足够光亮,可以进行更深层次的光线跟踪,那么构造一条反射光线,运用函数 shade()来计算有多少光线从反射方向反射回来。shade()计算来的物体的光线亮度将通过反射参数 shininess 来调整。反射参数被储存在物体的相关域里。用户在 SDL 文件里指定参数,比如说 shininess=0.8。它被用在"是否足够光亮"的判断中,例如 if(shininess>0.6)。

同样,如果碰撞的物体足够透明,我们构造一条折射光线,利用函数 shade()来计算有多少光线从折射方向传输过来。shade()计算来的物体的光线亮度将通过折射参数 transparency 来调整。折射参数被储存在物体的相关域里。它被用在"是否足够透明"的判断中,例如 if(transparency>0.5)。

因为光线有可能一直不停地构建新的反射和折射光线,所以在迭代的深度上必须加一些限制。考虑这样一个场景,有四面镜子被适当放置以至于一条光线可以永远在它们之间反射而不离开。这样会导致无止尽的迭代,这应该得到控制。为了解决这个问题,我们控制每条光线的迭代深度;在阴影计算中提到的 recurseLevel 结构是每条光线储存结构的一部分。从人眼发出的光线的 recurseLevel 为 0。每当一条新的反射或折射光线被构建,这个值就自动增加。如果光线的迭代层次达到一个限制值 maxRecurseLevel(储存在 Scene 结构的一个域中,在 SDL 文件中被指定为 maxRecursionDepth),就不再产生反射与折射光线。

12.12.1 光线的折射

当一条光线遇到一个透明的物体,一部分光线穿透物体,如图 12.71 所示。如果光线在介质 1 和介质 2 中的传播速度不一样,它的方向将从 **dir** 变成 **t**。向量 **t** 仍处在 **dir** 和法向量 **m** 所在的平面内。如果光线的入射角为 θ_1,由 **Snell 定律**可以得到折射角为 θ_2,见下式【Halliday70,Rogeers92】:

$$\frac{\sin(\theta_2)}{c_2}=\frac{\sin(\theta_1)}{c_1} \tag{12.44}$$

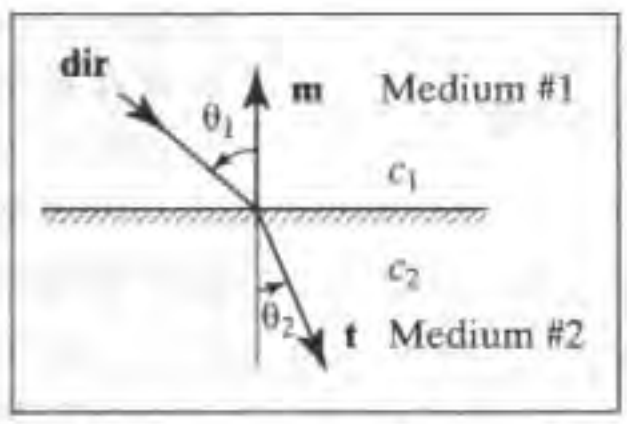


图 12.71 光线在透明介质中的折射

其中 c_1 是光线在介质 1 中的传播速度,c_2 是光线在介质 2 中的传播速度,如图所示。只有比率 c_2/c_1 才是重要的。通常称其为介质 2 关于介质 1 的**折射率**。注意若 θ_1 等于 0,θ_2 也为 0;若光线垂直入射到物体表面,折射光线也垂直表面。

例 12.12.1 求折射角

假设介质 2 为某种玻璃,光在其中的传播速度为真空中的 54%。再假设当前入射光的入射角是 60°。求折射角为多少?

解:

先求出等式(12.44)中的 $\sin(\theta_2)$,由 $c_2/c_1=0.54$,$\sin(\theta_1)=0.866$ 可以求得 $\sin(\theta_2)=(0.54)(0.866)=0.4676$,因此 $\theta_2=27.88°$。光线从传输快的介质进入传输速度慢的介质,传播方向会向表面法向量靠近。

图 12.72 给出了光在不同介质中传播的速度与其在真空(或空气)中速度的比值。这是一个简化的表格,因此不同波长的光在同一介质中传播的速度是不一样的。例如,光在溶化的石英中传播速度比值从 0.680(红光,波长 $\lambda=400\text{nm}$)到 0.685(绿光,波长 $\lambda=520\text{nm}$),到 0.687(蓝光,波长 $\lambda=680\text{nm}$)不等【Halliday97】。这种不同引起了一个常见的现象,当一束白光穿过一块玻璃棱镜时,被分离成像彩虹一样的光谱。

空气:99.97%
玻璃:52.2%～59%
水:75.19%
30%的盐水:72.46%
丙酮和乙醛:73.5%
氯化钠:64.93%
苯:55.5%
蓝宝石:56.5%
钻石:41.33%

图 12.72　不同介质中光的传播速度

临界角

例 12.12.1 中指出光线从速度快的介质进入速度慢的介质($c_2/c_1<1$),传播方向会朝法方向偏移。从等式 12.44 可以知道,因为 $\sin(\theta_2)$,等于 $c_2/c_1\sin(\theta_1)$,小于 $\sin(\theta_1)$,因此 θ_2 一定小于 θ_1。

反过来也成立,当光线进入一个传播速度相对快的介质:光线会偏离介质表面的法方向。当下标 1 和 2 互换的时候,Snell 定律是完全适用的。图 12.73(a)是光线从传播快的介质到慢的,图 12.73(b)是从传播速度慢的介质进入快的。这里两个介质中光线传播的角度是一样的,不同的只是代表它们的符号。图 12.73(c)和 12.73(d)给出了相似的情形,但是大角接近 90°。这时小角接近临界角。当小角(与传播速度慢的介质相关的)变得足够大时,它迫使大角接近 90°。更大的角度是不可能的,因此没有光线传输入相邻的介质中。这被称为全反射。考虑到图 d 的情形,光线从传播慢的介质进入传播快的介质。因为 $\sin(\theta_1)$ 等于 $c_1/c_2\sin(\theta_2)$,当 $\theta_2=90°$或 $\theta_1=\sin^{-1}(c_1/c_2)$时,角 θ_1 取得最大值。例如,当光线从水里传播到空气时,我们有 $c_1/c_2=0.7519$,因此临界角为 48.75°。

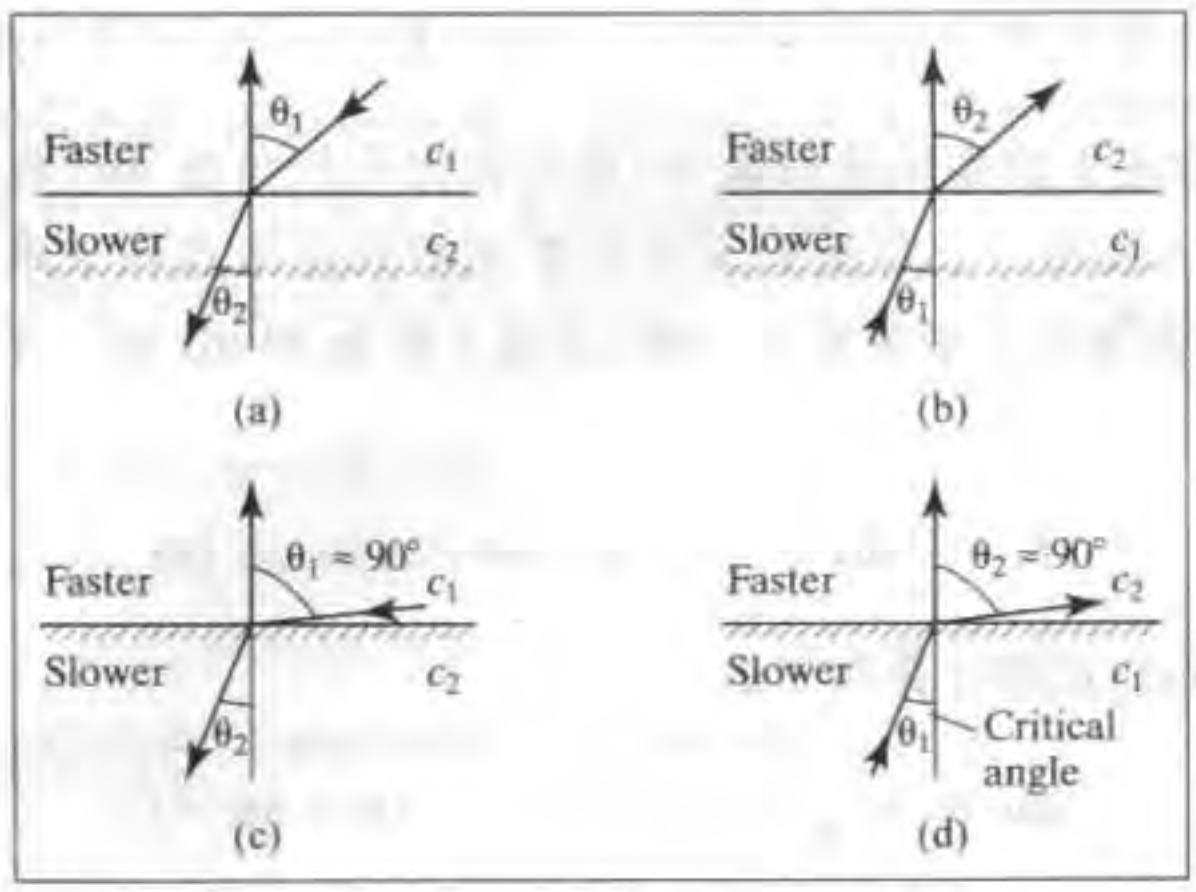


图 12.73　光线折射的对称性与临界角

临界角在一个有趣的情况下出现。当你处在池塘的水下,抬头向上看,你可以看到水面上的地平线以上的整个世界。当你以 48°角仰视的时候,你就可以看见池塘的边岸。另一方面,如果你在一条船上向下看池塘,你可以看到的角度范围是多少?有没有一个限制视角的圆锥体,在其之外你看不到任何东西?(参见本节最后的练习。)

例 12.12.2 彩虹的形状是怎样的?

光线在水中的传播速度随着波长的变化而变化,这就是彩虹的起因。图 12.74(a)所示是悬在空中的球体水滴。一条太阳光从左边入射,在水滴内部有个轻微的折射,在离开水滴之前又经历一个全反射。入射光线与出射光线的夹角大概为 42°。红色光线比蓝色光线折射角度小,因此出射的角度也有所不同。这样红色光线从成千上万个水滴中以同一方向反射出来,蓝色光线却以不同的方向出射。

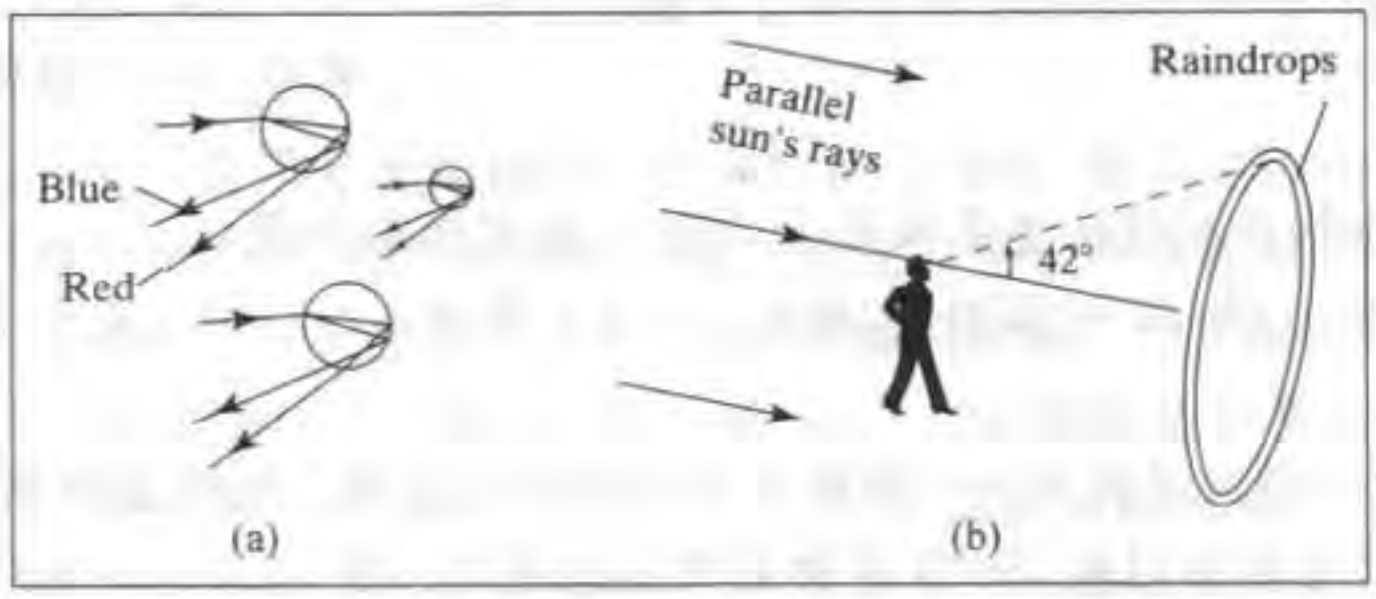


图 12.74 彩虹的起因

在 1637 年笛卡尔利用一个简单的光线跟踪模型来解释彩虹的形状。图 12.74(b)为一个背朝太阳的观察者,朝着由小水滴组成的气雾看去。当观察者的视角偏离太阳光 42°以上,她将看到一些光环。水滴沿着一个特定的角度反射一种特定颜色的光线,当角度有微小的变动时,反射回去的光线颜色也有所不同。因此,彩虹是一系列光线圆环的组合,每个观察者有她自己的彩虹。

计算折射角 t

在光线跟踪中,给定了曲面的法向量 **m**,和入射光线的角度 **dir**,我们必须计算折射角 **t**(见图 12.71)。我们应该在一个与坐标无关的系统中只使用点积来求得,为了可以适用任何 **m** 和 **dir** 方向。就像在练习中提到的一样,向量 **t** 是 **m** 和 **dir** 的一个线性组合(假设它们都被归一化):

$$\mathbf{t} = \frac{c_2}{c_1}\mathbf{dir} + \left(\frac{c_2}{c_1}(\mathbf{m} \cdot \mathbf{dir}) - \cos\theta_2\right)\mathbf{m} \tag{12.45}$$

其中 $\cos\theta_2$ 由 Snell 定律计算得来:

$$\cos\theta_2 = \sqrt{1 - \left(\frac{c_2}{c_1}\right)^2 (1 - (\mathbf{m} \cdot \mathbf{dir})^2)} \tag{12.46}$$

检查 **t** 在这两种情况下是否有正确的值:(a)两个介质中光传播速度一样;(b)入射角为时 0°。

如果等式 12.46 中平方根下的值为负数,就会出现全反射现象,这时,求解 **t** 的等式没有意义。这种情况在临界角时发生。练习中要求我们编写求解 **t** 的函数 transmitDirection()。

因为求解 **t** 只用到点积,它也适用于二维的情形。参见案例 12.8 中有趣的二维折射。

编写光线跟踪程序时,最简单的透明模型是不考虑光线波长对折射角的影响。在这种假

设下，同一条光线被用来跟踪红、绿、蓝的颜色分量。否则的话，每种不同颜色的光线需要不同的光线来跟踪，因为她们有不同的折射方向。这样计算的花费巨大，而且也仅仅是对光线的一个近似，因为精确地模拟折射需要考虑大量的颜色的光线，而不是这样三种主要颜色的光线。

练习

练习 12.12.1　折射的效果

图 12.75(a)给出了在高度 H 的人眼从空气往水里看的场景。一条与人眼水平距离相隔 L 的鱼在水下深度为 D。假设光在空气中传播速度是在水中的两倍，在什么地方人眼可以看见鱼？

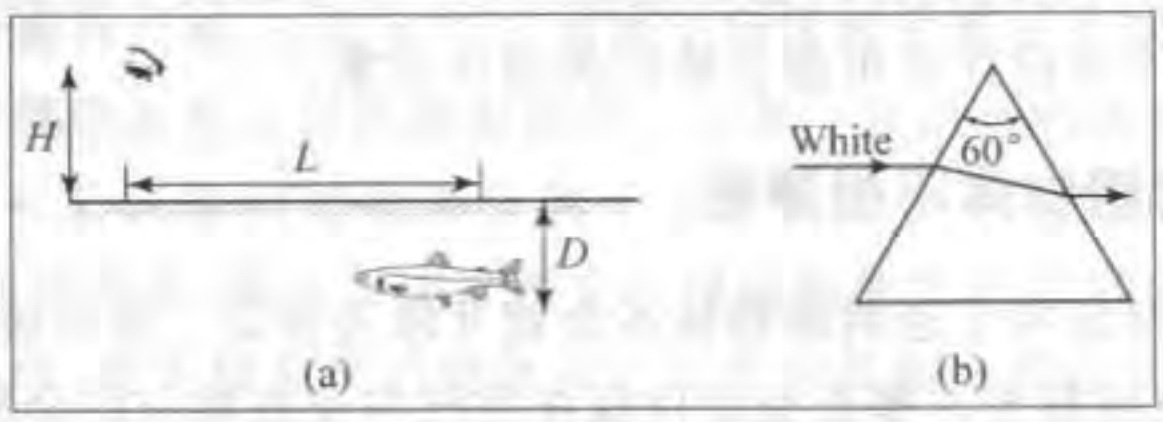


图 12.75　一些折射的图示

练习 12.12.2　画出彩虹

图 12.75(b)给出了一束穿过内角为 60°的棱镜的例子。棱镜是由石英制成的。计算并画出红、绿、蓝三种光穿过棱镜并分离的路径。若光线从一块两边平行的石英厚板穿过，存在光线分离之后的彩虹么？

练习 12.12.3　计算折射光线的方向

图 12.76 给出了类似【Heckbert89】中的一个简单的计算折射方向 **t** 的方法。单位向量 **dir** 和 **m** 唯一确定了一个平面，折射光与反射光也在这个平面之内。因此所有的向量都是单位向量，它们的坐标分量都是与 **m** 垂直或者平行，因此它们的长度可以用角 θ_1 和 θ_2 的正弦与余弦来表示。

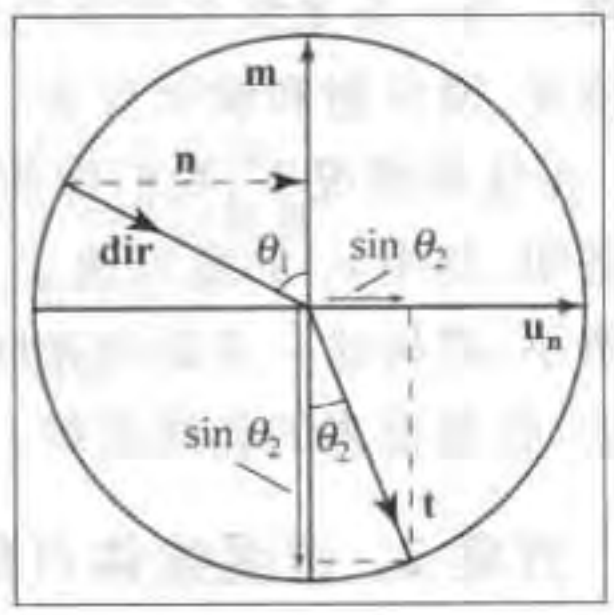


图 12.76　计算折射光线的方向

a. 证明：图中记为 **n** 的向量可表示为：

$$\mathbf{n} = \cos(\theta_1)\mathbf{m} + \mathbf{dir}$$

其长度为 $\sin(\theta_1)$，因此归一化之后的 $\mathbf{u}_n$ 等于 $\mathbf{n}/\sin(\theta_1)$。并用点积证明 **n** 处在 **m** 与 **dir** 所在的平面内，并且与 **m** 垂直。

b. 证明：

$$\mathbf{t} = \sin(\theta_2)\mathbf{u}_n - \cos(\theta_2)\mathbf{m} = \frac{c_2}{c_1}\mathbf{dir} + \left(\frac{c_2}{c_1}\cos(\theta_1) - \cos(\theta_2)\right)\mathbf{m}$$

并证明它与等式(12.45)等价。

c. 用 Snell 定律证明 $\cos(\theta_2) = \sqrt{1 - \left(\frac{c_2}{c_1}\right)^2 \sin^2(\theta_1)}$ 和等式(12.46)。

练习 12.12.4 计算折射方向的程序

给定单位法向量 **m** 和方向 **dir** 以及光线在两个介质中的传播速度,编写计算方向 **t** 的程序 Vector3 transmitDirection (Vector3 m, Vector3 dir, float c1, float c2)。

12.12.2 在函数 shade()中处理折射

当场景中包含透明的物体时,我们必须跟踪光线穿过的介质内部,因此我们可以计算下一次碰撞(光线穿出一种介质或者光线进入另外一种介质)时 c_2/c_1 的值。这可以很简单地做到:为光线添加一个域来储存一个指向它进入的物体的指针。

函数 shade()是如何处理物体内部的光线的呢?答案取决于绘制场景的人有多少自由度。我们可以想像若干个设计者可能会赞同的设计方案。

方案 1:两个透明物体不相渗透

首先假设设计者保证两个透明的物体不会相互渗透相交。没有玻璃大理石放置在另一个里面,没有水存在玻璃杯里。那么光线仅仅在空气中或在单一的介质中传播。

(a) 假设当前 shade()中的光线在所有物体之外,然后碰撞到一个足够透明的物体 A。利用等式(12.45)计算折射光线的折射角 **t**,这里空气中 $c_1=1$,而 c_2 取决于物体 A 的属性。一条包含指向 A 的指针的新光线被构建,它的 recurseLevel 适时地增加。然后 shade()递归地被调用,最终返回一个颜色值。这个颜色值被 A 的透明属性调整过并被添加到累计叠加的颜色里。

(b) 假设,另一方面,当前光线在某物体 A 内部并且碰撞到另外一张曲面。这张曲面一定是 A 的面,光线从 A 出射到空气中。程序可以检查光线是否碰撞的是它所在的物体。因为光线在物体内部,碰撞点的法向量的符号必须逆反一下;我们需要它指向光线传播的物体内部。当一条光线在物体内部时,通常认为它不在光源的照射之下,因此不去计算它的局部环境光、漫反射和镜面反射。A 的内表面被认为足够光亮,可以将一部分光线反射回物体 A。在这种情况下,我们像前面一样构建并计算反射光线。对于折射光线来说,A 显然足够的透明,其中 c_1 的值取决于 A 的属性,c_2 的值被设置成空气的对应值 1。如果入射角比临界角小,将构建一条新的光线(将其指针设置成 NULL,因为它不与任何物体相关)并且跟踪它,收集更多的光线贡献。

方案 2:透明物体可能相互渗透

如果我们允许把场景中透明的物体放置在一起并互相渗透,情况将会变得复杂一些。在图 12.77 的场景内,一个玻璃立方体嵌入了若干个物体:一个石英大理石球体,一个空气填充的圆柱体,一填满水的立方体和一块特别嵌入的玻璃锥体。我们将考察如何为第 12.13 节中提到的这样的**布尔物体**建立模型。光在每个物体里传播速度都不一样。图 12.77(b)所示光线在一系列相交的透明物体中的传播的情形。光线频繁地进出不同物体。用一个列表记录光线进出的物体。沿着光线的路径,物体按照一个复杂的次序在列表中增加或删除。这虽然是一个人为制造的场景,但是它值得用来检验函数 shade()是否可以正确地处理这类问题。光线跟踪算法可以添加这种额外的能力也可以不加。

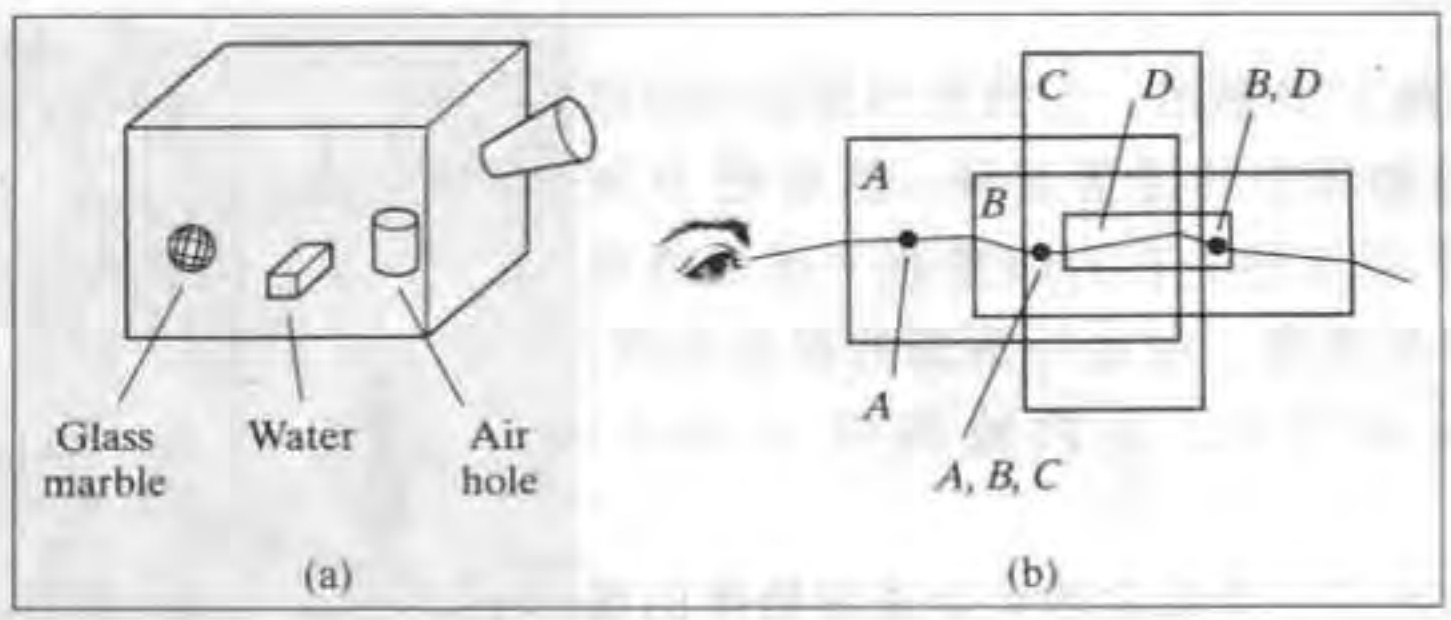


图 12.77　若干个互相渗透的透明景物

同时在两个透明物体内部，空间中的点被两个物体拥有是什么意思呢？在建模的层面，你必须决定想渗透的部分有什么样的材料属性。如果一块绿色的大理石完全包含于一块蓝色的大理石内部，直观上认为相交的部分属于蓝色大理石，给其着上蓝色。但是如果像图 12.77(b)所示的部分相交，那么要决定相交区域的颜色就不是那么明显了。设计者必须给每个物体一个优先级，优先级最高的物体决定每个与其相交的区域的颜色。

一种处理方法是给每个物体都增加一个域来储存优先级信息，并在 Ray 结构中增加一个指向此物体的指针，于是形成了一个指针列表。在任何时刻，光线处于某个物体集合当中，这时的列表实时记录了这个集合。如果列表为空，光线则处于所有物体之外。因此我们必须扩充 Ray 类型，如图 12.78 所示。列表被储存成一个数组而不是一些相互连接的列表是为了在添加物体的时候方便一次复制整个物体。

```
class Ray{
public:
    Point3 start;
    Vector3 dir;
    int recurseLevel;
    int row, col;                          //用于投影包围合
    int numInside;                         //列表中的景物数量
    GeomObj * inside[8];                   //景物指针数组
    Ray(){recurseLevel=numInside=0;}       //构造函数
    …其他方法…
}
```

图 12.78　Ray 类型的扩充

怎样在函数 shade()运用这个列表呢？在任何时刻当前光线都处在一些物体的内部，这被记录在列表中。当它继续碰撞到某个物体的表面时，记为 B，我们对其是入射还是出射 B 相应地采取不同的策略，这可以依靠 isEntering 来决定。

(a) 假设光线入射 B。如果 B 没有足够透明，就不构建折射光线，但如果 B 足够光亮，需要构建反射光线。如果 B 足够透明，使用当前列表里优先级最高的物体的光线传播速度 c_1。如果 B 的优先级别更高，那么 c_2 使用光在其中的传播速度；否则让 c_2 等于 c_1。添加 B 到物体列表。构建一条新的折射光线，将当前的列表(包含 B)复制过去。递归地调用 shade

()函数。

(b) 假设光线从 B 射出。c_1 我们用光在列表里面优先级最高的物体中的传播速度。然后把 B 从列表中删除。而 c_2 使用光在列表里剩下优先级最高的物体中传播的速度。构造一条新的折射光线,将当前的列表复制过去。递归地调用 shade()函数。

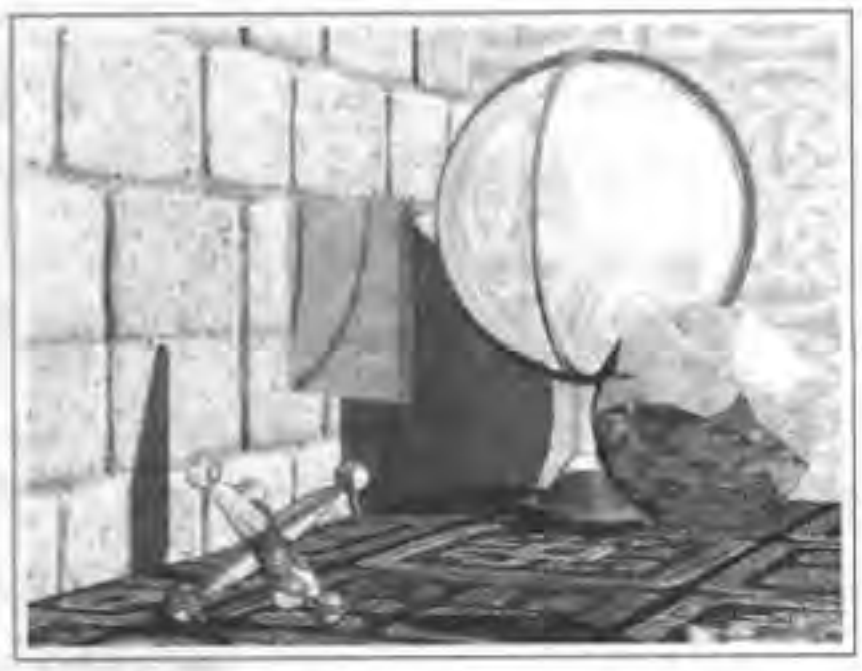


图 12.79 有折射和没有折射的场景

图 12.79 给出了一个包含若干个透明物体的场景。球体和架子是透明的,地板和墙是不透明的。当光线在透明物体中折射时,扭曲现象就会出现。

练习

练习 12.12.5 另外一种设计方案:如果两个透明物体相互渗透,一个包含另外一个

假设建模者规定不允许两个透明物体只有一部分相交:既不是分离,也不是一个包含另一个。用集合的术语来说,它们的交集既不为空,也不是它们中的任何一个。讨论逻辑上 shade()应该如何实现。特别地,说明处理列表过程变得简单:不需要优先级,列表可以被视为堆。讨论实现的细节。

练习 12.12.6 光线从有颜色的玻璃中穿过

讨论光线从桔黄色玻璃球穿过时,折射效果如何实现的。

12.13 复合体:物体上的布尔操作

到目前为止,我们只能光线跟踪有限的几种形状:变换过的球体、平面、锥体等。如果有一个方法可以将这些简单形状组合成一个复杂的物体,并开发适合这种情况的光线跟踪算法,那么将有更多更有趣的场景可以用光线跟踪算法绘制(图 12.80)。

构造实体几何(CSG)就是这样一种方法。任意的复杂形状被定义为一系列简单形状上的布尔**操作**的集合【Ballard82, Mortenson85】。象透镜、透明鱼缸和包含洞的物体之类的这些物体,可以很容易地通过目前已接触过的简单物体的组合来得到。这样的物体被称为**复合体**、**布尔体**或 **CSG** 体。将光线跟踪算法推广到这类物体上很自然:这是光线跟踪算法强大的一个地方,它与 CSG 模型结合的非常好。

我们看例子中的三种布尔操作:**并集**、**交集**和**差集**。图 12.81 给出了两个有球体构造出来的复合体。图 12.81(a)是有两个球体通过**交集**构造的透镜。一个点落在透镜内部,当且仅当它同时落在两个球体内部。球体 S_1 和 S_2 相交,符号上我们通常用 L 记成等式(12.47)。

$$L = S_1 \cap S_2 \tag{12.47}$$

图 12.80 复杂布尔物体的光线跟踪

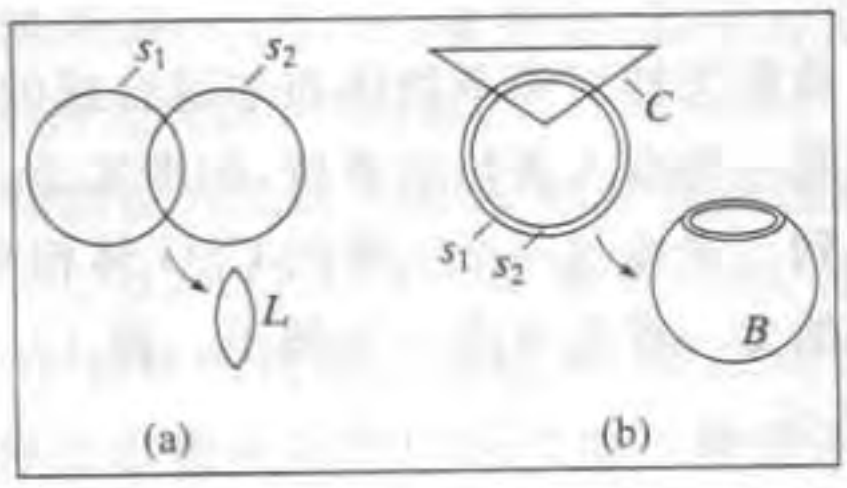


图 12.81 透镜和鱼缸

图 12.81(b)给出了由差集操作得到的一个鱼缸。一个点在集合 A 与 B 的**差集**中，记为 $A-B$，如果它在 A 中不在 B 中。差集相类似于移除物体，切割或雕刻。这个鱼缸由下面的等式(12.48)定义。

$$B=(S_1-S_2)-C \tag{12.48}$$

移除球体 S_1 内部的球体 S_2 里面所有点，可以得到一个内空的球壳。然后将其顶端移除锥体 C 里面的点得到一个开口。

一个点在两个集合 A 和 B 的**并集**中，记为 $A\cup B$，如果它属于集合 A 或集合 B 或两个集合。求两个物体的交集类似于将它们粘合在一起。图 12.82 给出了一个火箭模型，它由两个圆锥体和两个圆柱体通过并集操作得到，数学上表达为等式(12.49)。

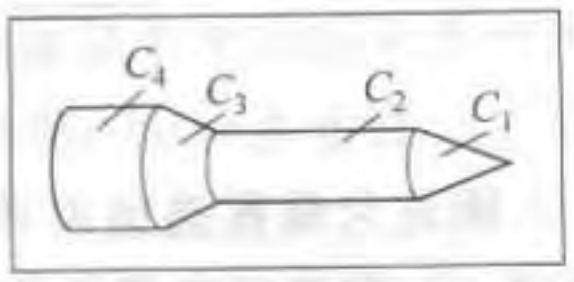


图 12.82 四个简单体的并集

$$R=C_1\cup C_2\cup C_3\cup C_4 \tag{12.49}$$

锥体 C_1 接在柱体 C_2 上。锥体 C_3 一部分嵌入到 C_2 里，另一边拼接在 C_4 上。

练习

练习 12.13.1 分解复合体

求图 12.83 中物体的等式表达式，运用球体、锥体、柱体和规则平行六面体的集合操作来表示。注意同一个物体的表达式可能不一样。

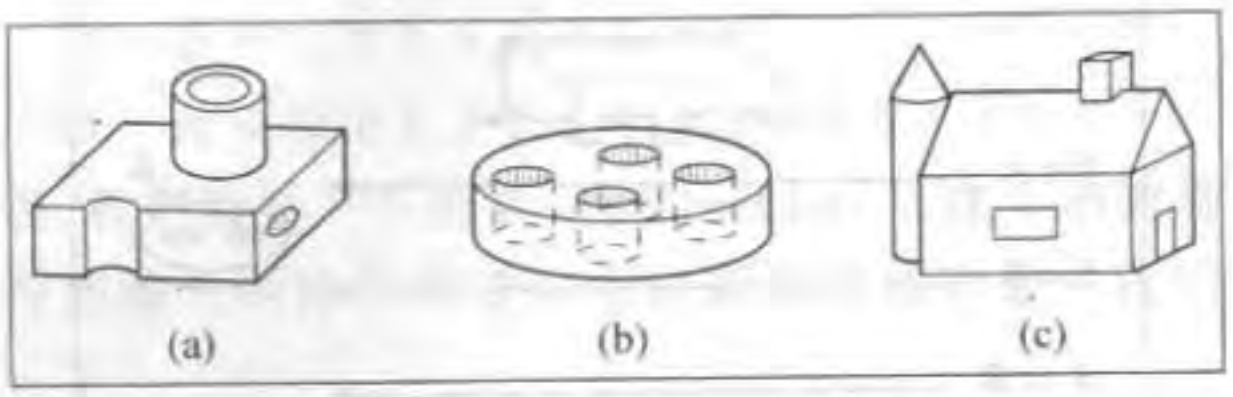


图 12.83 几个由简单形状组成的物体

练习 12.13.2 组合复合体

考虑保龄球类似的物体，球面上有三个便于抓牢的指洞。将其表达成一个球体和几个柱体的 CSG 组合。

12.13.1 CSG 体的光线跟踪

我们怎样对简单物体布尔组合成的物体使用光线跟踪技术呢？第一眼看上去可能会认为这是一件令人害怕的事情，但事实上，归功于光线跟踪过程的本质，这件事情可以很容易地办到。光线与一个三维的 CSG 体相交被简化成一个一维数字列表与一个 CSG 体的一维部分的交。首先考虑一个例子。图 12.84 给出了一条光线在标出的时间入射和出射球体 S_1 和 S_2。

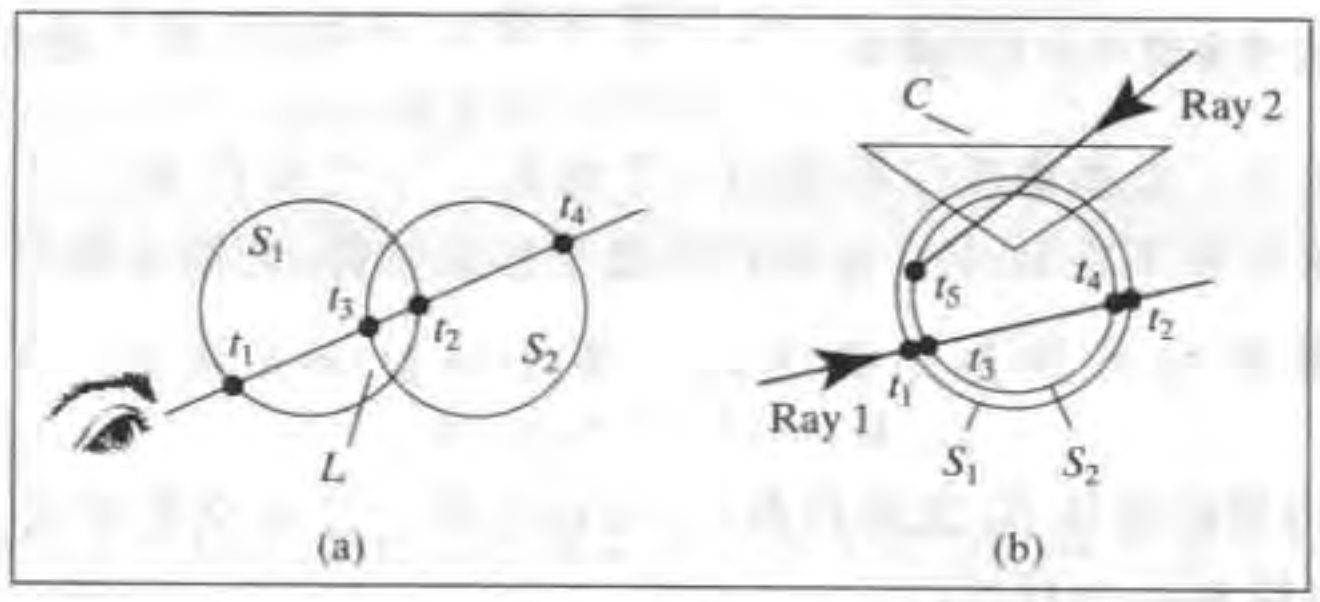


图 12.84 透镜与鱼缸的光线跟踪

因此光线在透镜 L 内部的时间从 t_3 到 t_2，碰撞的时间是 t_3。如果透镜是不透明的，我们将运用熟悉的渲染规则来决定透镜上碰撞点的颜色。如果它是镜面或者是透明的，我们将用合适的角度构造新的光线用于更深的跟踪。

图 12.84(b)中的鱼缸有着类似的情形。一方面，光线 1 首先与鱼缸在 t_1 相碰撞，最先相交的是 S_1 而不是 S_1 和 S_1。另一方面，光线 2 首先与鱼缸在时刻相碰撞。同样最先相交的是 S_2 而不是其他球体和锥体。最先碰撞的是组成鱼缸的分支物体，但不是鱼缸本身。

我们将这些想法组织成一个可以光线跟踪任何复合体的算法【Roth82】。考虑两个物体，A 和 B，还有一条光线。构建一个光线入射和出射 A 的时间列表，并按升序排列。因为物体是固体的，入射和出射时间轮流出现。为 B 构建一个类似的进出时间列表。图 12.85 给出了两个这样的列表。

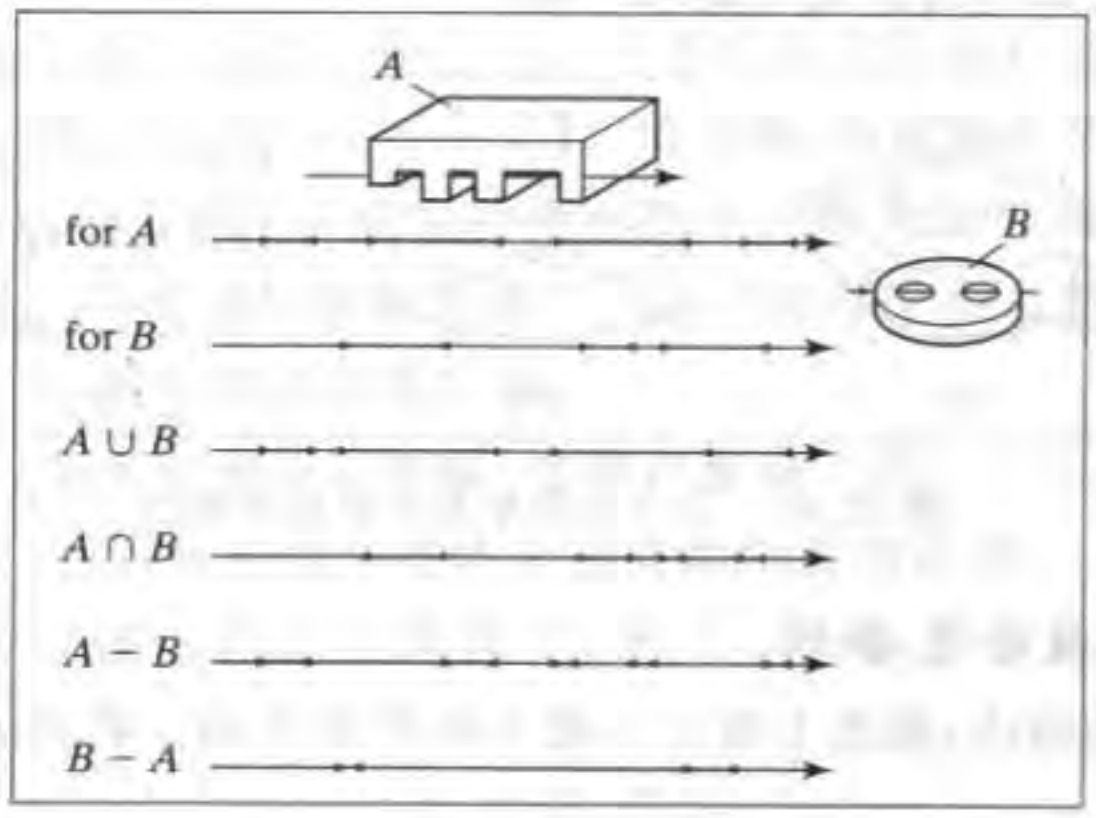


图 12.85 光线与两个物体的 t—值列表

入射的时间和下一个出射时间之间的区间用实线表示；否则用虚线表示。在实线区间当中，光线在物体的内部。如果在某个 t—值集合里光线处在物体的内部，我们则称其为**内部集合**。一个内部集合由一个包含入射与出射时间的列表(t_1，t_2，…)来指定，其中 t_1 是入射时间，t_2 是一个出射时间，以此类推。有时候称这个为光线的 t—列表。

给定一条光线与两个物体 A 和 B 的内部集合，我们想为光线与由 A 和 B 组成的复合体找到内部集合。我们可以构造的四个新物体为 $A \cup B$，$A \cap B$，$A - B$ 和 $B - A$。考虑 $A \cup B$。光线在物体并集内部当且仅当它处于至少一个物体的内部。因此 $A \cup B$ 的内部集合可以简单地定义为原来单个内部集合的并集。同样的思想可以推广到其他三种物体上。$A \cap B$ 的内部集合是原来单个内部集合的交集；$A - B$ 的内部集合定义为 A 的内部集合与 B 的内部集合的差集，同样地定义 $B - A$ 的内部集合(只需交换 A 与 B 的位置)。总的来说，如果我们记 $T(A)$和 $T(B)$为两个物体 A 和 B 的内部集合，如等式(12.50)所示，

$$T(A\ op\ B) = T(A)\ op\ T(B) \tag{12.50}$$

其中 op 为布尔操作 $\cup$，$\cap$ 或 $-$ 之一，那么复合体 $A\ op\ B$ 的内部集合即为 $T(A)\ op\ T(B)$。总之，如上所示，对物体进行布尔操作等价于与列表中的区间做同样的布尔操作。

虽然我们最终仅仅对内部集合的第一项(第一次碰撞)感兴趣，但我们必须在整个构建过程中为内部集合维持整个列表，因为最终的首次碰撞时间可能会被某个中间的列表所改写。值得庆幸的是，我们已经为光线跟踪中构建内部集合做好了准备：hit()方法在一个以时间排列的数组 inter.hit[]里为每种形状记录了光线的所有碰撞行为。从这个方法看来，我们只用到列表里的第一次碰撞时间。尽管如此，当我们开始光线跟踪布尔物体时，我们还需要利用整个列表。

例 12.13.1　构造内部集合

考虑下列两个内部集合的列表：

A_list：1.2 1.5 2.1 2.5 3.1 3.8

B_list：0.6 1.1 1.8 2.6 3.4 4.0

我们将运用上面的规则构造四个新的内部列表(查看它们的准确性)：

$A \cup B$ = 0.6 1.1 1.2 1.5 1.8 2.6 3.1 4.0

$A \cap B$ = 2.1 2.5 3.4 3.8

$A - B$ = 1.2 1.5 3.1 3.4

$B - A$ = 0.6 1.1 1.8 2.1 2.5 2.6 3.8 4.0

为了构造新的内部列表，我们需要一个函数 combineList()，这个函数将一个布尔操作和两个 t—列表作为参数，按照输入的操作构造相应的新的内部列表。我们将在后面讨论这个函数是如何工作的。

复合体的光线跟踪过程概括起来分下面几步，光线跟踪各个分支物体，为每个分支构建内部集合，最后将这些内部集合复合。复合后的列表中最先碰撞的时间就是光线最先碰撞到复合体的时间。接着运用普通的渲染技术来绘制，包括产生第二条光线，如果曲面为光亮面或透明面。

练习

练习 12.13.3 寻找向部列表

假设给定一条光线,物体 C 和 D 有如下 t 一列表:

C_list: 0.8 1.7 2.2 2.9 4.7 5.55

D_list: 1.2 1.5 2.1 2.5 3.1 3.8 4.7 8.3

找出 C 和 D 的四种可能布尔复合体的 t 一列表。

12.13.2 布尔物体的数据结构

在一个程序里我们如何表达类似 $((A\cap B)-C)-D$ 的布尔复合体?因为复合体总是由两个(可能也是复合体)物体复合而成,$Obj_1\ op\ Obj_2$,因此二叉树结构能很好的表示此类物体。图 12.86 给出了与物体 $(((B_1\cap B_2)\cup C_2)\cup((S_1-S_2)-B_3))-C_1)$ 相关的树的结构。每个内部的节点(用圆圈表示)代表一种操作,每片叶子(有矩形表示)代表一种原始的形状。一棵树由若干子树组成,这些子树也由原始形状或子树组成(这个过程无限进行下去)。当一个节点包含差集操作时,这被理解为这个节点构造了(lefttsubtree)-(rightsubtree),与(rightsubtree)-(lefttsubtree)相反。

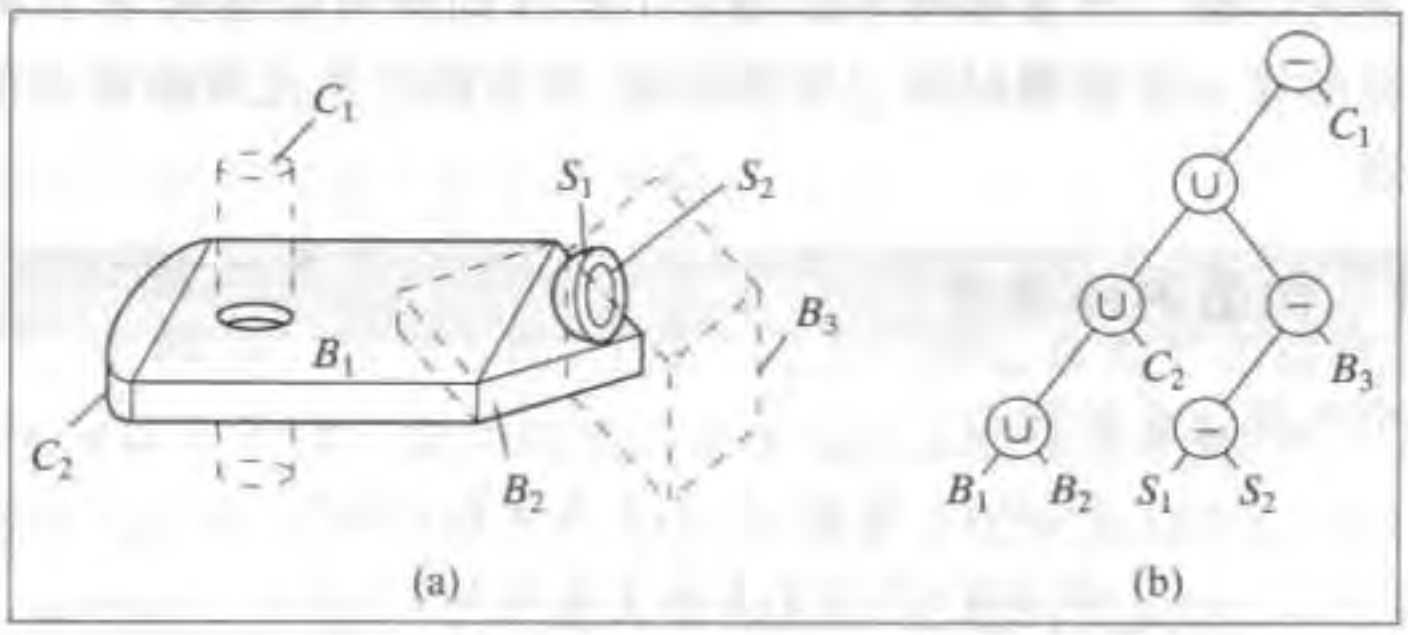


图 12.86 复合体和它的 CSG 树

复合体的表达式可以被重新排列而不改变最终表达的物体形状。每次重新排列将产生不同的树结构。因此一个给定的复合体不是只有一棵唯一的树来表示。

到目前为止,所有的物体类型例如 Sphere 和 Cube 都是由类 Shape 而来,而这个类又是从基本类 GeomObj 而来(见附录 3 的继承关系)。从 GeomObj 得到类 Boolean 看起来是很自然的事情,这样物体列表(一系列 GeomObj 的列表)就可以记录一个布尔体。一个布尔体必须可以记录一棵二叉树,因此我们给它两个指针 left 和 right 指向它的子树。这些指针必须指向 GeomObj 类型,因为一些情况下指针会指向集合图形,另一些情况下会指向布尔树。因此类 Boolean 的结构应该是这样的。

```
class Boolean : public GeomObj{
    Public:
      GeomObj *left, *right;                          //指向子分支的指针
      Boolean(){left=right=NULL;}                     //构造函数
      virtual bool hit(Ray &r, Intersection &inter);
```

```
    …其他方法…
};
```

GeomObj 类型拥有球体和长方体包围盒，布尔体也是，这个很重要，因为对布尔体光线跟踪的花费是很大的，任何加速的技术都是受欢迎的。我们应该为布尔体设计一个特殊的可以判断光线与布尔体相交的 hit()函数，使用多态性来简化代码。实际上，因为 hit()必须对并集、交集和差集做不同的操作，这样将会产生适用于这三种布尔体的不同类：UnionBool、IntersectionBool 和 DifferenceBool。UnionBool 的定义简单的表示如下。

```
class UnionBool: public Boolean{
    public:
      UnionBool(){Boolean();}        //构造函数
      virtual bool hit(Ray &r, Intersection &inter);
      …其他方法…
};
```

其他类的定义类似。每种类型将包含一个适用于不同物体性质的 hit()函数。

布尔体没有自己的仿射变换；只有 Shape 类型有。这样使得一棵布尔树只有在叶子的部分拥有变换。这种情况既带来了好处，也带来了坏处，事实上一些建模系统和光线跟踪算法有不一样的操作。为了理解这里的差别，考虑图 12.87(a)，它给出了一棵布尔体的树，它的每个内部节点和叶子都有自己的变换。每个变换可以影响它的子树中的所有节点。我们将在练习中讨论，这棵树在几何上与图 12.87(b)的树等价，即它们表示的形状是一样的。这些变换将一直被过滤到叶子节点，然后组合成一个新的变换。

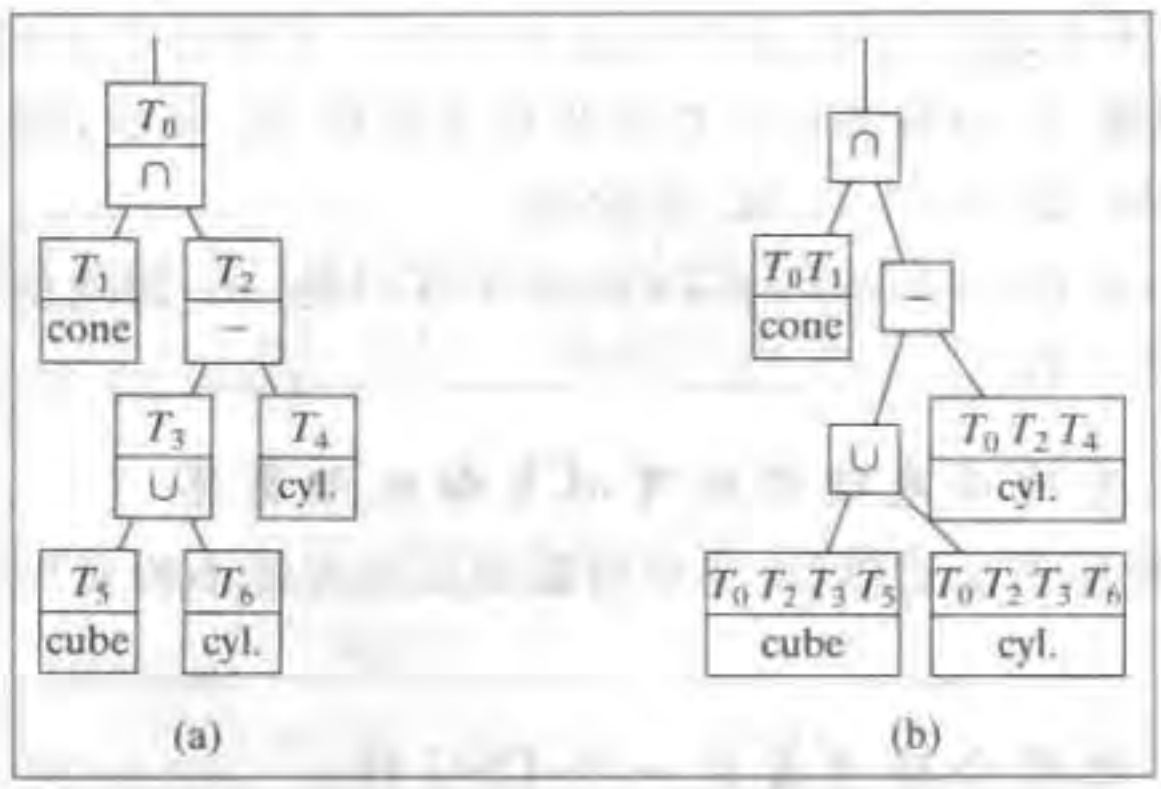


图 12.87　复合体包含变换的内部节点

保持节点中的显式变换的一个好处是，在 CSG 体被构建之后，设计者可以单独使用树中的一个变换以便于调整形状。这也为寻找布尔体的更紧致的包围盒提供了可能。一个重要的弱点是光线跟踪的速度，因为每一条光线必须在树的每个节点，不仅仅是叶子部分，逆变换一次。

在 SDL 文件里指定布尔体的方法是使用下面一种关键词：union、intersection 或 difference，然后指定左边的几何体，再指定右边的几何体(它们自身也可以是布尔体)。例如，一个如图 12.84(b)的鱼缸，由$(S_1-S_2)-C$给出，可以由如下的 SDL 代码指定。

```
!make a fish bowl
rotate -90 1 0 0
difference
    difference
        sphere !outer sphere
        push scale 0.9 0.9 0.9 sphere pop !inner sphere
    push translate 0 0 1.3 rotate 180 1 0 0 cone pop
```

第一个变换将整个鱼缸围绕 x 轴做了一次旋转。然后在左边物体(它本身是两个球体的差集)和锥体之间做差集。因为给予 SDL 的变换将在后面与当前变换做叠加,而当前的变换在形状被构建的同时就放置在了其中,所以变换将自动地往树的下面传播。

练习

练习 12.13.4 重新排列布尔体表达式

因为写出一个复合体的表达式有机动性,可以有不止一棵树来表达它。例如∪和∪等价,因此 $A\cup B=B\cup A$。判断下列等式的正误,并证明。

a. $(A-B)-C=A-(B\cup C)$

b. $(A\cup B)-(A\cup C)=A\cup(B-C)$

c. $(A\cup B)\cap(A\cup C)=A\cap(B\cup C)$

练习 12.13.5 变换是如何在布尔树中向下传播的

考虑图 12.84(a)中的树,每个内部节点都有一种变换。

a. 判断并证明这棵树所表示的几何形状与图 12.84(b)所表达的是否一样。证明过程依赖仿射变换么?

b. 如果图中的变换 T_i 由在各向同性的坐标中矩阵 M_i 表示,图 12.84(b)中立方体的矩阵组合与 $M_0M_2M_3M_5$ 或 $M_5M_3M_2M_0$ 等价吗?

提示:证明 $T(A\ op\ B)=T(A)\ op\ T(B)$其中 $T()$是一一对应的变换,A 和 B 为点集,op 代表∪,∩或−中的一种。

练习 12.13.6 为什么光线跟踪可以自动处理并集

解释为什么光线跟踪两个物体(两物体可能相交也可能不相交)之后的并集等价于光线跟踪它们的并集。

练习 12.13.7 把整个场景看成一个 CSG 体

讨论将整个场景看成一个复合体的好处与不好之处,与使用一个简单复合体列表做比较。

12.13.3 布尔体与光线的交

我们需要开发一个适用于各种布尔体的 hit()方法。它必须可以构建光线与左右子树的内部集合,并且可以恰当地将它们组合起来。

图 12.88 给出了交集操作下的 hit()函数的一个框架结构。包围盒测试判断是否可以提前退出函数(将在下节讨论)。然后左边子树调用合适的 hit()函数,除非光线与此子树不相交,不然构建碰撞列表 lftInter。如果它们不相交,函数 hit()返回 false,这是提前退出,因

为光线要与两棵子树的交集相交必须与它们同时相交。如果相交，我们再构建一个右子树的碰撞列表 rtInter，最后用交集操作将两个碰撞列表组合起来。

```
bool IntersectionBool::hit(Ray &r, Intersection &inter)
{
    Intersection lftInter, rtInter;
    if(光线与包围盒不相交) return false;
    if((!left->hit(r, lftInter)) || (!right->hit(r,
rtInter)))
        return false;                //提前退出
    构建复合列表：将其放置到 inter 内
    return(inter.numHits>0);     //
}
```

图 12.88　交集操作下的 hit()函数的一个框架结构

其代码与类 UnionBool 和 DifferenceBool 的相似。

构建复合 t—列表

构建复合 t—列表是复杂的但是符合逻辑的。考虑两个碰撞列表的例子，L 与 R（分别代表左和右），如图 12.89 所示。为了简单起见，我们记左边的列表为 L[]而不是 leftInter.theHit[]，右边列表相类似。而且，hitObject 和 surface 省略。L 里记录了 8 个碰撞，R 记录了 5 个碰撞。每个列表包含排列过的碰撞时间，结构中第 0 个元素记录了当光线入射或出射物体时，第一次碰撞时间是否为正。因为物体是固体，所以在第 0 个元素之后入射与出射的值交替出现。

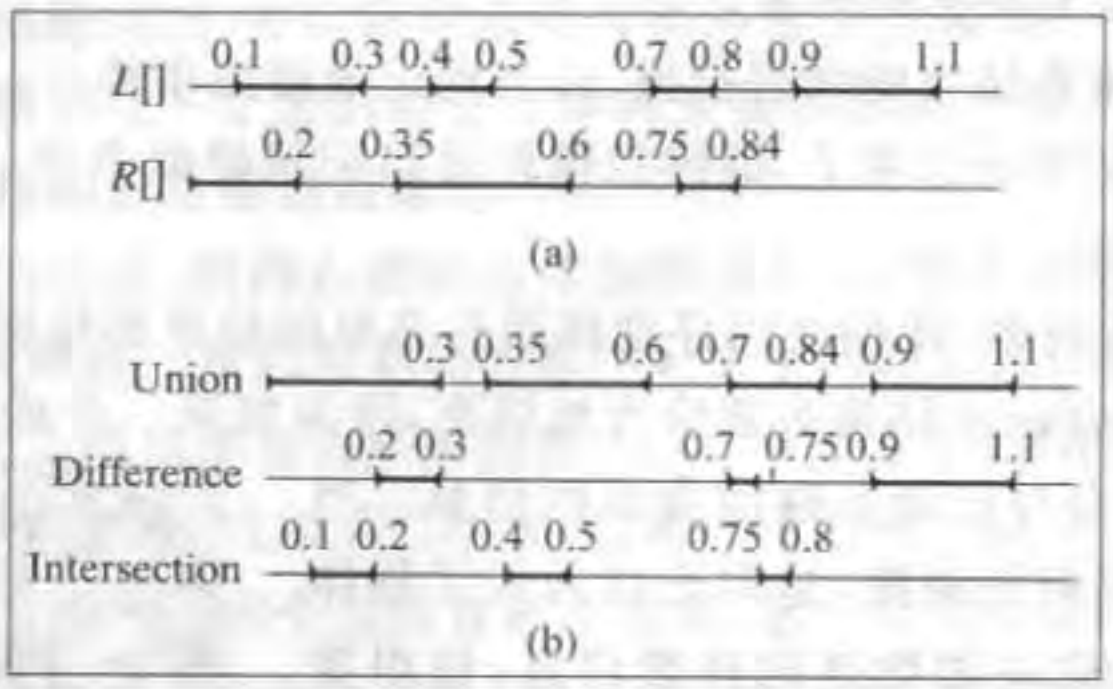


图 12.89　相交区间的例子

复合列表的构建是通过扫描这两个列表来实现的，在每一步中考察两个列表后面的哪一项拥有更小的碰撞时间，并在这个过程中保持光线在物体内部还是外部的跟踪。这些信息将通过不同布尔操作所对应的规则来组合。图 12.89(b)给出了三种不同的布尔组合所对应的复合 t—列表。

考虑一个特定的构建复合列表的例子。假设在差集的操作中，光线在碰撞下一个面之前是处在左边物体的内部和右边物体的外部，因此它处在它们的差集当中。检查每个列表

中的下一个碰撞时间;假设右边列表的下个碰撞时间更小。这意味着在下次碰撞之后,光线将进入右边物体,因此它现在应该在它们差集的外部。在这种情况下,光线的状态从在差集外部改变为在差集的内部。

当光线伴随下列变量从子树中通过时,我们保持跟踪它的状态(无论其现在处在物体的内部还是外部)。

```
bool lftInside;          //值为真,如果光线在左边物体内部;
bool rtInside;           //值为真,如果光线在右边物体内部;
bool combInside;         //值为真,如果光线在复合体内部;
```

这些是光线在碰撞之间的状态——即在下次碰撞之前的状态。lftInside 的初始值依照光线第一次碰撞时是射入还是射出左边物体——即依照 L[0].isEntering 的值。具体的,如果首次碰撞之后光线射入物体(在碰撞之前光线在物体外部),lftInside 的值即为 false;否则 lftInside 的值为 true。因此我们可以将 lftInside 初始化为! L[0].isEntering,rtInside 也类似。然后 combInside 被定义为 lftInside 和 rtInside 的一个逻辑上的组合;如果操作为并集,combInside 为真当且仅当 lftInside 或 rtInside 或两者都为真。这可以很简单地运用逻辑运算操作来实现:combInside=lftInside || rtInside。类似地,如果是交集操作,我们使用 combInside=lftInside && rtInside,如果是差集操作,我们用等式 combInside=lftInside && ! rtInside。

此算法过程不断地扫描列表 L[]和 R[],在每一步寻找较小的碰撞时间,用它来更新 lftInside,rtInside 和 combInside。如果 combInside 的值在某点处改变了,那么最后一次碰撞事件将被添加到列表 C[]中。

当列表 L[]或 R[]已经到达尽头(当它的最后一个碰撞时间被扫描到),其他没有结束的列表有可能不用再考虑:

- 交集时:忽略没有结束的列表;
- 并集时:继续没有结束的列表扫描,适当地记录碰撞事件;
- 差集时:如果左列表结束了,继续扫描并适当记录碰撞事件;如果右列表结束,忽略没有结束的列表。

注意在复合列表的时候,我们必须仔细将所有必要的信息都传递给每个新的 hit[]。特别地,isEntering 和 hitObject 结构必须恰当地填充,因为如果一个碰撞称为第一次碰撞,那么我们将从 hitObject 自身记录下碰撞曲面的信息。当右边物体发生了一个出射的碰撞,这时光线射入了物体自身,我们必须清楚指向左边物体的碰撞信息,使得我们透过洞看过去的是左边的物体。

当物体是透明的而光线在布尔体里面时(例如当光线穿过一个马提尼酒杯),我们也必须确保 CSG 物体的逻辑函数 hit()被正确执行。对于上面不同的物体,即使光线刚刚进入物体,正确的碰撞信息也会被放置在碰撞列表中。

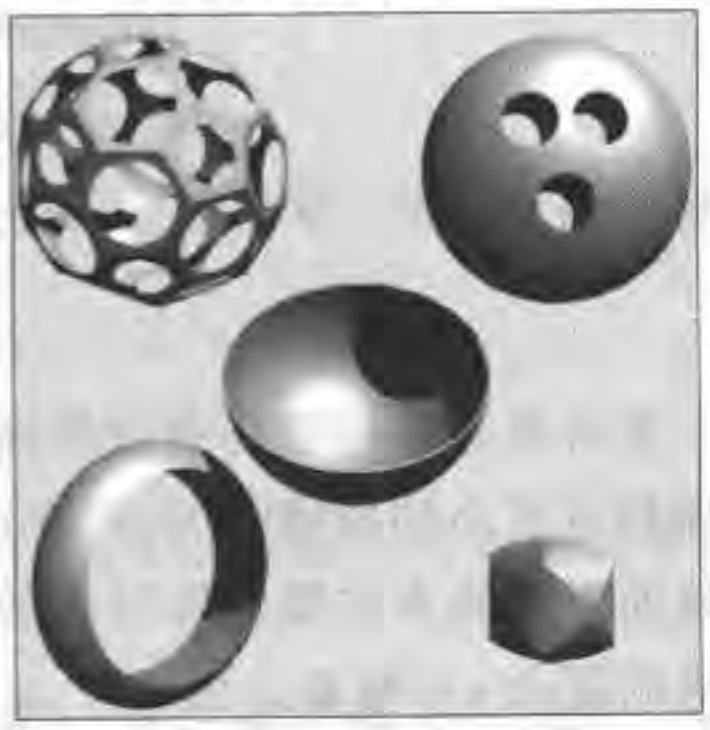


图 12.90 光线跟踪复合物体的例子

图 12.90 给出了用这些技术来光线跟踪多个复合

物体的例子。试着确定给定的布尔物体是由哪些物体构成的。

练习

练习 12.13.8　估计算法结果

利用图 12.88 中的复合列表的算法估计例 12.13.1 中几个例子的运算结果。

练习 12.13.9　编写函数 hit()

为类 UnionBool 和 IntersectionBool 实现对应的 hit()函数。每个函数必须很好地运行在指定的布尔操作上。

12.13.4　为 CSG 物体构建和使用包围盒

还剩下一个问题，那就是为 CSG 物体构建投影、球体、长方体包围盒。在预处理阶段，我们扫描 CSG 物体对应的树，为它的每个节点构建包围盒并且储存在节点自身当中。在随后的光线跟踪中，光线先同遇到的包围盒做求交测试，如果我们可以确定光线与物体不相交，函数就可以提早退出。这样做可以节省复合列表的花费。

在图 12.88 中提到的，包围盒测试总是放置在物体类的 hit()函数当中的。如果光线与当前节点的包围盒相碰撞，像通常一样我们认为它与左右子树相交。

构建长方体包围盒

一个 CSG 节点的合理的长方体包围盒的定义是怎样的呢？这里，当然，指的是世界坐标下的长方体包围盒，因为一个布尔体没有一般的视图。假设一个节点代表物体 $A\ op\ B$，这里 A 和 B 是形状，op 是一种布尔操作。物体的长方体包围盒必须包裹整个物体，使得如果光线与包围盒没有碰撞，我们可以肯定光线与物体也没有碰撞。物体的形状可能很复杂（例如，一个旋转过的圆环与一个歪斜的立方体的交集），因此在没有经过复杂的处理，要自动找到物体最紧致的包围盒是相当困难的。

我们考虑最简单的情况，利用 L 的长方体包围盒 $E(L)$ 和 R 的长方体包围盒 $E(R)$ 来构建 $L\ op\ R$ 的长方体包围盒。我们为不同的操作定义不同的长方体包围盒。

(a) **并集**：定义长方体包围盒为同时包裹 $E(L)$ 和 $E(R)$ 的调整过的长方体。这个等价于 $E(E(L)\cup E(R))$。（问题：这跟 $E(L\cup R)$ 总是一样的吗？这个比分别对 $E(L)$ 和 $E(R)$ 求交更有效吗？如果光线与这两个包围盒都不相交，它一定与 $L\cup R$ 不相交吗？）

(b) **交集**：定义长方体包围盒为 $E(L)$ 和 $E(R)$ 的交集。两个调整过的长方体的交集总是一个调整过的长方体，并且很容易计算它的（*left*，*top*，*right*，*bottom*，*front*，*back*）分量。（问题：如果光线与 $L\cap R$ 的包围盒不相交，它一定与 $L\cap R$ 不相交吗？）如果 L 和 R 的包围盒是紧的，那么这个交集的包围盒也被认为是紧的。

(c) **差集**：简单定义长方体包围盒为 $E(L)$。这是很保守的做法，因为 $E(L-R)$ 有可能比 $E(L)-E(R)$ 小。这样 $E(L)$ 可能会出现很大一块空间，但是做一个更精确的分析会花费很多。

长方体包围盒将递归地求得，因为一棵布尔树的内部节点的长方体包围盒将从它的子

节点的包围盒中求得。如果是简单形状,它的包围盒将不需要递归地构建,我们前面提到过:在世界坐标中求物体的点云,然后构建一个包裹它的调整过方向的长方体包围盒。每种形状类型拥有不同的makeBoxExtent()函数。每个函数都构建一个长方体包围盒并且保存在物体中,以备其他物体的调用。例如,类UnionBool可能用下面的代码。

```
Cuboid Unionbool::makeBoxExtent()
{
        Cuboid lft=left->makeBoxExtent();
        Cuboid rt=right->makeBoxExtent();
        Cuboid tmp;
        tmp.left=min(lft.left, rt.left);          //构建并集
        tmp.top=max(lft.top, rt.top);
        //等等,其他四个分量的定义
        worldBoxExtent=tmp;                       //将其保存到物体中
        return tmp;
}
```

这个处理过程为物体列表中每个物体都调用makeBoxExtent()函数。当一个布尔体调用这个方法的时候,它像每个中间节点一样,构建并在物体中保存长方体包围盒。

构建球体包围盒

世界坐标下的球体包围盒的构建与长方体包围盒的构建类似。像前面讨论的一样,为每个形状求世界坐标下的点云,球体包围盒是最适合这个点云的球体。对于布尔物体,我们必须判断如何构建两个物体的并集、交集和差集的球体包围盒。

差集很简单;用左边物体自身的球体包围盒。它不会非常紧,但是它是适用的。交集可以同样使用,因为为物体交集求真正的球体包围盒相当困难。并集情况可以先将两个物体的点云组合,然后求复合点云的球体包围盒。

构建投影包围盒

为了减少光线跟踪的花费,运用投影包围盒是非常必要的。一个CSG体的每个节点的投影包围盒的构建与其长方体包围盒的构建是一样的。一个节点的投影包围盒由其左右子树的投影包围盒组成。用P(object)来表示object的投影包围盒,按照上面的规则我们有如下表示:

$$P(L \cup R) = P(P(L) \cup P(R))$$
$$P(L \cap R) = P(L) \cap P(R)$$
$$P(L - R) = P(L)$$

像前面一样,每个类有其自身的makeBoxExtent()函数(只用于光线跟踪)。

12.14 光线跟踪与光线投射

光线投射方法始于1992年ID软件公司的游戏Wolfenstein 3D。在Wolfenstein 3D中玩家被放置进一个像迷宫一样的环境中,在这里玩家可以在房间里移动与魔鬼和敌人战斗。

虽然光线跟踪是一种可以产生适用与各种场景的高质量的图像的算法，但它非常费时，因此没有办法适用于实时绘制，例如游戏和实时模拟。而光线投射是一个非常快的技术，它只能处理受到严格几何约束的场景并且产生的图像是斑驳的、不真实的。

图 12.91　使用光线投射算法的游戏 Wolfenstein 3D 的截图

图 12.91 给出了一些 Wolfenstein 3D 的游戏截图，展示了那些可以运用光线投射算法的场景类型。注意斑驳的、粗糙的图像质量和纹理。近些年随着图形硬件的更新，光线跟踪算法的限制变得不重要，光线投射算法基本上被抛弃了。F. Permadi 的网站上有对光线投射算法的详细介绍。

本章小结

光线跟踪算法提供了一个简单的均匀的绘制华丽的真实图像的方法。场景中大量相关光线被跟踪，分析每一条光线最终返回观察者眼睛的光亮度。这个机制为绘制多种视觉效果提供了可能。光线跟踪不但可以模拟环境光、漫反射光和镜面反射光，还可以模拟相关的阴影、镜面反射和透明物体的投射。光线跟踪还为二维和三维场景提供了一种自然的纹理映射方法。

光线跟踪算法的核心是求光线与物体是否相交的程序。为了绘制高质量的图像，巨大数量的光线被构建，大部分光线还要产生若干条二次光线，这样使得大量的时间用于求交测试。因为每条光线与场景中每个物体都要求交，光线跟踪的时间花费与场景中的物体数量成线性关系。包围盒技术简化了这一过程，它可以提前判断大量光线与物体是否相交，这样可以避免大量的无谓的求交计算，于是提高了光线跟踪的速度。

光线跟踪算法自动解决隐藏面的绘制问题，因为一条光线与各种物体的碰撞时间是它们与观察者距离的一个直接的度量。因此拥有最小碰撞时间的物体是最近的一个，它不会被其他物体所遮挡。相同的规则可以用来直接计算阴影。从光线与物体的碰撞点发出一个阴影探测器，它指向光源，每个物体的任何碰撞都做测试。如果一个物体处在碰撞点与光源之间，那么这个点就在与这个光源相关的阴影中。因此，检测这样的阴影相当于检测物体与阴影探测器的交点是否处在 0 和 1 之间。

一些形状容易做光线跟踪，因为求光线与其的交点只需要求解一个线性或二次等式。这一类的典型是平面、球体、圆柱体和圆锥体。场景设计者从普通视图开始，然后用仿射变换将它们变换到一个设计需要的大小、朝向和位置。这样可以将圆球体变成椭球体，将立方体变成平行六面体，等等，这样就扩充了物体的种类。任意凸多面体的光线跟踪也很简单，因为它的内部几乎可以看成一系列半平面的交集。

光线是一类容易处理的，有很好性质的物体，当它经过仿射变换，得到的还是一条光线。因此为了检测哪些物体与它相交，从一个给定的起始点开始产生一些新的光线是不难的。这提供了一个直接的方法来累加光滑曲面的反射和透明物体的折射。实现这样的技术也是很简单的。

为光线跟踪添加跟踪 CSG 体的方法增加了需要渲染的场景的种类。十分复杂的物体

可以被构造和渲染。光线跟踪一个 CSG 体的内在逻辑是很直观的,并且容易实现。

今天,光线跟踪是最强大的绘制计算机合成的图像的方法之一。虽然巨大数量的计算使得它的运行速度很慢,但是它所产生的图像的真实度是其他方法所没有办法达到的。

案例分析

案例 12.1 放射型光线跟踪器

(难度级别:Ⅱ)编写并执行一个应用程序,实现对 SDL 中所描述场景的光线跟踪。该光线跟踪器只对自身发光的物体进行适当处理。不要求其计算环境光、漫反射光和镜面反射光分量。该程序应该至少能够跟踪球体、圆锥体、立方体、正方形和平面。SDL 场景文件中包含的其他形状可以忽略。光线跟踪开始之前利用 Scene::drawOpenGL()方法绘制出预览图。

利用键盘按键来控制各项功能。如 12.5 中讨论的,按键 1、2、4、8 用于设定像素块的尺寸,使得场景在不同分辨率下可以切换;其他按键则用于调整照相机的位置和方向。

还要构建一个 SDL 的场景,其中包含了几个插孔和八角体,并适当调整位置以便光线跟踪。每个插孔有三个互相垂直的长条,这里的长条是指一个小长球体,她的两端各有一个小球。八角体由一些小球组成,它们位于一个八面体的 6 个顶点上,而顶点之间的 12 条边则由小长球体组成。

案例 12.2 一种新型光线跟踪器

(难度级别:Ⅱ——假设实例分析 12.1 中的光线跟踪器已经存在)对其补充改进,是的它能够正确处理环境光、漫反射光和 Phong 镜面高光。它不需要处理反射、折射和阴影效果。它可以正确地跟踪所有的通用形状以及凸多边形网格。和上面一样,也要在光线跟踪之前绘制出场景的预览图。

为了实现进一步控制,请编写代码,使用户可以利用鼠标指定显示器上的矩形区域,然后在对指定的区域进行最大分辨率的光线跟踪(块尺寸=1)。这方便了那些需要快速光线跟踪,并且只需要在一个指定范围内进行最大分辨率光线跟踪的用户。

案例 12.3 在光线跟踪中实现阴影效果

(难度级别:Ⅰ——假设实例分析 12.2 中的光线跟踪器已经存在)请对其添加能显示阴影效果的功能。

案例 12.4 用包围盒加速光线跟踪算法

(难度级别:Ⅱ——假设实例分析 12.3 中的光线跟踪器已经存在)请在光线跟踪中加入球体包围盒、长方体包围盒和投影包围盒,并比较使用每种包围盒所起到加速计算的作用。允许用户使用一个按键来控制包围盒的生效和失效,从而确定每种光线跟踪花费的时间。

案例 12.5　对三维纹理的光线跟踪

(难度级别：Ⅱ——假设实例分析 12.2 中的光线跟踪器已经存在)改进光线跟踪，使其可以支持一些三维纹理显示，包括棋盘格、木头纹理和大理石纹理。用户使用不同的按键来选择纹理，然后再利用所选择的纹理来渲染场景中的所有物体。

确定一种方法，用于扩展 SDL 和方法 Scene::read()，使得 SDL 文件中的参数能够被指定并与随后定义的物体相关联。使用参数来选择物体的纹理，并指定纹理定义中的一些数值参数。例如，SDL 里：

```
parameter 3 12.6 -91
cube
```

可能关联到有关立方体的三个参数，所以将纹理 #3 关联到此物体。

案例 12.6　反走样

(难度级别：Ⅱ——假设实例分析 12.2 中的光线跟踪器已经存在)扩展实例分析 12.2 中的光线跟踪器，使它能够实现反走样处理。对于每条主光线，它将发射 N 条方向略微不同的光线，并对这些光线的返回值取平均。请取不同的 N 值，并在光线方向上添加不同数量的抖动，分别进行实验。

案例 12.7　对其他基本形状的光线跟踪

(难度级别：Ⅲ——假设实例分析 12.2 中的光线跟踪器已经存在)请研究一下 Jim Kajiya 关于对形如棱柱、不规则山脉以及旋转面等复杂形状的光线跟踪的文章。要求至少可以实现对分形山的光线跟踪，并使用关键字 mountain 以及一些用于指定山脉的属性的参数来扩展 SDL。

案例 12.8　一个用于研究折射的二维光线跟踪器

(难度级别：Ⅱ)因为在等式(12.45)中关于折射方向 t 的等式只用到点积，所以它同时适用于二维和三维的情况。编写并执行一个程序，产生类似于图 12.63 所示的图片。当用户给定折射率和入射角时，该程序绘制出入射和折射光线。

案例 12.9　反射光和折射光

(难度级别：Ⅲ——假设实例分析 12.2 中的光线跟踪器已经存在)扩展此光线跟踪器，使其能够正确处理光亮面的反射光线和透明物体的折射光。采用不同的折射率对不同的透明形状进行实验，从而研究折射现象的本质属性。

案例 12.10　光线跟踪物体的布尔组合

(难度级别：Ⅲ——假设实例分析 12.2 中的光线跟踪器已经存在)扩展实例分析 12.2 中的光线跟踪器，使其能够正确跟踪任意复杂的布尔体。创建一个包含此类物体的有趣的场景，包括：

- 内部有个圆柱形孔的物体;
- 一个球面棱镜 $A\cap(B-C)$(由两个球体相交而成);
- 由 $A\cap(B-C)$确定的物体,其中 A、B 和 C 均表示形状。

适当调整它们的位置,使得能够透过其他一些对象中的孔来看到某个物体,并且透过透镜看到某些物体。

进一步阅读

由 Andrew Glassner 编著的"An Introduction to Ray Tracing"【1989】,提供了大量优秀的论文,这些文章研究了用于光线跟踪的基本算法以及加速光线跟踪算法的一些方法。"Graphic Gems III"【Kirk92】提供了一些有用的有关光线跟踪的文章。Watt 和 Watt【Watt92】对光线跟踪以及相关的各种方法进行了更深入和广泛的研究。因特网上(http://www.irtc.org/)的光线跟踪网站搜集了一些光线跟踪的图像,并提供了有关光线跟踪的想法和动机。

附录 1

图形工具：怎样获取和安装 OpenGL

本书中所讨论的大部分代码是基于以 OpenGL 为应用程序接口(API)的。本附录将介绍在使用 OpenGL 构建自己应用程序时，如何获取所需的 OpenGL 工具。

A1.1 获取和安装 OpenGL

对于目前常用的任何一种计算机平台，获得必要的 OpenGL 软件都是直接且免费的。我们将针对各种主要的平台，讨论如何获取该软件。此外，因特网上提供了大量关于 OpenGL 的补充信息。

OpenGL 信息库

http：//www.opengl.org/上提供了大量关于 OpenGL 的一般信息以及软件下载的起始网址。从 http：//opengl.org/documentation/上也可以获得包括一些在线手册在内的许多信息。本书的网站(见前言)上提供了关于 OpenGL 的附加信息和连接。OpenGL Architecture Review Board (ARB)出版的 OpenGL Programming Guide(经常被称作"红宝书")是一本很优秀的关于使用 OpenGL 的必读书目。该书介绍了 OpenGL 软件的获取和安装，目前已经有第 5 版。

读者的需求

无论使用什么系统，你首先得有一个优秀的 C/C++ 编译器并且安装适当的 OpenGL 头文件和库。正如本书所述，为了使用 OpenGL，必须有以下三个库及相关文件：

- OpenGL(基本的 API 工具)；
- GLU(OpenGL 实用程序库)；
- GLUT(OpenGL 实用工具包，一个用于处理视窗系统操作的视窗工具包)。

一般来说，每个库都对应几个相关的文件：一个头文件(.h)、一个库文件(.lib)和在某些系统中的动态链接库文件(.dll)。

添加头文件：把 gl.h、glu.h 和 glut.h 这三个文件放在你编译器的 include 目录的 gl 子目录(你自己新建的)中。你需要在每个应用程序中写下面的包含语句：

```
#include <gl/Gl.h>
#include <gl/Glu.h>
```

```
#include <gl/Glut.h>
```

连接库文件：每个应用程序都将成为项目的一部分被编译和连接。除了自己写的(.c 或.cpp)文件之外，还应该在你的项目中添加适当的 OpenGL 库(.lib)文件以便连接器能够找到它们。

下面我们分别讨论对各种主要类型系统的要求。

Microsoft Windows 95/98/NT

大部分 Windows 机器上已经安装了 OpenGL。在写这本书的时候，Windows XP 机器上都自带了 OpenGL。

Windows 系统中使用 OpenGL 的一个合适的环境是微软的 Visual C++ 6.0 或更高版本。(如果你把应用程序声明为"win32 控制台应用程序"，就没有必要建立一个图形用户界面：键盘输入和文本输出分别通过不同的控制台窗口进行)。

单击 http://www.opengl.org/documentation/spec.html 中"Download OpenGL Specs"来获取基于 Windows 系统的 OpenGL 及其实用程序库的最新信息[①]。这些库也可以通过微软 FTP 站点提供的自解压文件 ftp://ftp.microsoft.com/softlib/mslfiles/opengl95.exe 来获得。其中包括 gl.h、glu.h、glue2.lib、opengl32.lib、glu32.lib 以及 opengl32.dll(不要使用老的.aux 文件)。

glut 库可以通过 http://www.opengl.org/resources/libraries/glut.html 获得。其中包含了 glut.h、glut32.lib 和 glut32.dll(不要使用 glut.lib 和 glut.dll)。下载并解压这些文件之后，将.lib 文件放在合适的目录下，以便将来把它们添加到你的每个工程中。

把.dll 文件放在 Windows/System 目录下(如果它们还未被放在这里)。加入下面的 include 语句将指定文件包括在你的工程中。

```
#include <windows.h>
#include <gl/Gl.h>
#include <gl/Glut.h>
#include <gl/Glut.h>
```

如果你将.h 文件放在了不同于上述的文件夹下，例如"myglut"，包括的指令就应该是 #include <myglut/glut.h>。

Macintosh

Macintosh 下的 OpenGL 主要来源是 http://www.apple.com/macosx/features/graphicsand media/。OS 9 和 X 用户应当注意 OpenGL 库和 GLUT 已经自动安装了！Opengl.org 也有关于如何在更早版本的 Macintosh 平台上运行 OpenGL 的信息。每一个 iMac、iBook、PowerBook、PowerMac G4 和 PowerMac G5 都有内建的 OpenGL 硬件加速。

---

① 翻译这本书时该网页已经无效。可以通过访问 http://www.opengl.org/documentation/specs/来获得相应信息。

UNIX 和 X Windows 系统

每一个 UNIX 工作站都支持 OpenGL。GLX 提供了 OpenGL 和 X Windows 系统之间的连接。所有的 OpenGL 实现都需要用 X。资源和下载信息可以在 http：//www. sgi. com/products/ software/opensouce/glx/上面找到。GLX 把 OpenGL 扩展到了 Linux、FreeBSD 和其他使用流行的 X 系统 XFree86 实现的操作系统。

Mesa3D

Mesa 是一个可以免费获得的类似于 OpenGL 的库。目前，最好的下载地址是 http：//www. mesa3d. org/#Systems。上面附带了说明及每一个可下载版本。另外一个选择是 Windows 下的 Cygwin。Cygwin 是 Windows 下运行的 Linux。Cygwin 允许应用程序员连接到 Windows 的 dll。Cygwin 比 Mesa 运行快，而且允许访问可能在本地 Linux 视频驱动上不能得到的视频硬件。

附录 2

计算机图形学的数学基础

本附录总结并归纳了全书所涉及的各种数学知识。有时还会提供结论的简要推导，不过这些资料主要是为了便于参考。

A2.1 一些关于矩阵及其运算的主要定义

本附录只是回顾了一些矩阵的基本概念和矩阵运算的方法。更加全面的分析可以参考其他书籍(例如，【Birkhoff76】，【Faux79】)。

矩阵是一个由元素组成的矩形数组，这里的元素通常是数字。一个 m 行、n 列的矩阵称为 $m\times n$ 矩阵。例如

$$A=\begin{bmatrix} 3 & 2 & -5 \\ -1 & 8 & 0 \\ 6 & 3 & 9 \\ 1 & 21 & 2 \end{bmatrix} \tag{A2.1}$$

是一个 4×3 的整数矩阵，而 $B=[1.34,-6.275,0.0,81.6]$ 是一个 1×4 的矩阵，也称为四元组或者向量。一般来说，一个 $1\times n$ 的矩阵就是一个行向量，而一个 $n\times 1$ 的矩阵是一个列向量。

按惯例，矩阵中的单个元素用小写字母符号表示并用下标来区分。矩阵 B 的第 ij 个元素表示为 b_{ij}。这代表了第 i 行第 j 个元素。同理，对于等式(A2.1)中的矩阵，$a_{32}=3$。如果一个矩阵行数和列数相等，就称之为方阵。在图形学中，我们经常使用 2×2、3×3 和 4×4 的矩阵。零矩阵和单位矩阵是两个常用的矩阵。零矩阵中的所有元素为零。除了主对角线($i=j$ 对应的 a_{ij})上值为 1 的元素外，单位阵中的其他元素也都是零。因此，3×3 的单位阵如下：

$$I=\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A2.1.1 矩阵运算

数字矩阵 B 可以被一个数 s 缩放。B 中的每个元素和 s 相乘。结果矩阵用 sB 表示。以等式(A2.1)中的矩阵 A 为例，我们可以得到

$$6A = \begin{bmatrix} 18 & 12 & -30 \\ -6 & 48 & 0 \\ 6 & 126 & 12 \end{bmatrix}$$

两个具有相同行数和列数的矩阵 C 和 D，称之为具有相同形状。它们可以相加。相加的和 $E=C+D$ 的第 ij 个元素是 C 和 D 中相应元素的和：$e_{ij}=c_{ij}+d_{ij}$。因此

$$\begin{bmatrix} 3 & 2 & -5 \\ -1 & 8 & 0 \\ 6 & 3 & 9 \\ 1 & 21 & 2 \end{bmatrix} + \begin{bmatrix} 0 & 5 & -1 \\ 9 & 8 & -3 \\ 2 & 6 & 18 \\ 4 & 2 & 7 \end{bmatrix} = \begin{bmatrix} 3 & 7 & -6 \\ 8 & 16 & -3 \\ 8 & 9 & 27 \\ 5 & 23 & 9 \end{bmatrix}$$

因为矩阵可以被缩放和相加，所以定义（具有相同形状的）矩阵的线性组合是很有意义的，例如 $2A-4B$。从这些定义可以直接得出关于具有相同形状的三个矩阵 A、B 和 C 的下列结论。

$$A+B=B+A$$
$$A+(B+C)=(A+B)+C$$
$$(f+g)(A+B)=fA+fB+gA+gB$$

通过交换矩阵 M 的行和列就可以构成 M 的转置，记做 M^T。M^T 的第 ij 个元素是 M 的第 ji 个元素。因此等式（A2.1）中矩阵 A 的转置为

$$A^T = \begin{bmatrix} 3 & -1 & 6 & 1 \\ 2 & 8 & 3 & 21 \\ -5 & 0 & 9 & 2 \end{bmatrix} \tag{A2.2}$$

一个行向量的转置是一个列向量。例如，

$$(3,2,-5)^T = \begin{bmatrix} 3 \\ 2 \\ -5 \end{bmatrix}$$

如果一个矩阵和它自己的转置相等，就称该矩阵为对称矩阵。只有方阵才可能是对称矩阵。因此，一个 $n\times n$ 的矩阵 M，如果满足 $m_{ij}=m_{ji}$ 对任意 1 到 n 之间的 i 和 j 成立，它就是对称矩阵。

A2.1.2 两个矩阵相乘

第 5 章开始讨论的变换涉及到了一个向量和一个矩阵相乘以及两个矩阵的乘积。其中，前者是后者的一个特例。

只有当两个矩阵 A 和 B 匹配时，才可以定义他们的积 AB。也就是说第一个矩阵 A 的列数要等于第二个矩阵 B 的列数。因此，如果 A 是 3×5 的，B 是 5×2 的，就可以定义 AB，但不能定义 BA。

A 和 B 乘积 $C=AB$ 中的每一项仅是 A 中某行与 B 中某列的点积。具体来说，积的第 ij 个元素 c_{ij} 是 A 的第 i 行和 B 的第 j 列的点积。因此一个 $n\times m$ 的矩阵和 $m\times r$ 的矩阵的乘积是一个 $n\times r$ 的矩阵。例如，我们可以得到

$$\begin{pmatrix} 2 & 0 & 6 & -3 \\ 8 & 1 & -4 & 0 \\ 0 & 5 & 7 & 1 \end{pmatrix} \begin{pmatrix} 6 & 2 \\ -1 & 1 \\ 3 & 1 \\ -5 & 8 \end{pmatrix} = \begin{pmatrix} 45 & -14 \\ 35 & 13 \\ 11 & 20 \end{pmatrix}$$

例如,由于$(2,0,6,-3)\cdot(2,1,1,8)=-14$,所以 $c_{12}=-14$。一个很相似的例子是行矩阵和列矩阵的乘积,如

$$(4,5)\begin{pmatrix} 2 \\ 3 \end{pmatrix} = 23$$

这个结果是因为$(4,5)$和$(2,3)^T$ 的点积是 23。一个 1×1 的矩阵和另一个 1×1 的矩阵的乘积自然等于简单的将两个数字相乘。

两个方阵乘积的例程已经在本书的网站上公开了。由此可以很容易地扩展到任意两个相互匹配的矩阵的乘积。

我们列举了矩阵乘法的一些有用的性质。假设矩阵 A、B 和 C 相互匹配。那么

$$(AB)C = A(BC)$$
$$A(B+C) = AB + AC$$
$$(A+B)C = AC + BC$$
$$(AB)^T = B^T A^T$$

且

$$A(sB) = sAB$$

其中 s 是一个数。当计算两个矩阵 A 和 B 的乘积的时候,矩阵顺序的不同会导致不同的结果。对表达式 AB 我们称之为"A 左乘 B"或者"A 被 B 右乘"。如果 A 和 B 是相同大小的方阵,在两种顺序中它们都匹配,因此 AB 和 BA 都可以定义,但是两种乘积可能包含了不同的元素。如果 $AB=BA$ 对于两个矩阵成立,就称他们**可交换**(两个对称矩阵总是可交换的吗?)。

向量和矩阵相乘

矩阵相乘的一个特例是其中的一个矩阵是行向量或列向量。在图形学中,经常见到一个列向量 w 被左乘以一个矩阵 M,形式为 Mw。例如,令

$$w = \begin{pmatrix} 2 \\ 5 \\ -3 \end{pmatrix} = (2,5,-3)^T$$

且

$$M = \begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix}$$

那么 $\mathbf{w}$ 和 M 相匹配,可以得到

$$M\mathbf{w} = \begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix} \begin{pmatrix} 2 \\ 5 \\ -3 \end{pmatrix} = \begin{pmatrix} -14 \\ 33 \\ 4 \end{pmatrix}$$

通过上述相同的方法，$M\mathbf{w}$ 的每个元素都是 M 的行和 $\mathbf{w}$ 的点积。同样可以对一个矩阵左乘以一个行向量 $\mathbf{v}$，如

$$\mathbf{v}M = (3,-1,7)\begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix} = (-2,34,71)$$

A2.1.3　矩阵分块

把矩阵分成元素块并给每块命名有时会很便利。例如，

$$M = \begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix} = \left(\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right)$$

其中各元素块被定义为

$$M_1 = \begin{pmatrix} 2 & 0 \\ 8 & 1 \end{pmatrix},\quad M_2 = \begin{pmatrix} 6 \\ -4 \end{pmatrix},\quad M_3 = (3,2)$$

和 M_4，由单个元素 7 组成。这称为把 M 分块为以上 4 块。需要注意的是，如果一个块在另一个的上方，他们必须具有相同的列数。与此相似，两个块并排时，它们必须具有相同的行数。两个按同样方式进行分块的矩阵（对应块具有相同的形状）可以通过分别相加对应块来计算。对分块矩阵进行转置，可以先对每块分别转置，然后在将块之间的排列进行转置。因此，

$$\left(\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right)^T = \left(\begin{array}{c|c} M_1^T & M_3^T \\ \hline M_2^T & M_4^T \end{array}\right)$$

同样，矩阵乘法也可以通过将它们的子矩阵按通常的方式相乘得到，只要子矩阵相互匹配：

$$\left(\begin{array}{c|c} M_1 & M_2 \\ \hline M_3 & M_4 \end{array}\right)\left(\begin{array}{c|c} M_5 & M_6 \\ \hline M_7 & M_8 \end{array}\right) = \left(\begin{array}{c|c} M_1M_5 + M_2M_7 & M_1M_6 + M_2M_8 \\ \hline M_3M_5 + M_4M_7 & M_3M_6 + M_4M_8 \end{array}\right)$$

A2.1.4　矩阵的行列式

每个方阵 M 都有一个与之关联的数字，这个数字被称为**行列式**，记做 $|M|$。行列式描述了特定几何形状的体积，并提供了说明线性变换对物体面积和体积产生的影响的信息。对于 2×2 的矩阵，其行列式很简单，就是两个积的差：

$$|M| = \begin{vmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{vmatrix} = m_{11}m_{22} - m_{12}m_{21}$$

如果 M 是一个 3×3 的矩阵，它的行列式具有如下形式

$$|M| = \begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{vmatrix} = m_{11}\begin{vmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{vmatrix} - m_{11}\begin{vmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{vmatrix} + m_{13}\begin{vmatrix} m_{21} & m_{21} \\ m_{31} & m_{32} \end{vmatrix}$$

因此，

$$\begin{vmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{vmatrix} = 294$$

注意到这里的$|M|$是三项之和：$m_{11}M_{11}-m_{12}M_{12}+m_{13}M_{13}$。所以它可以表示成点积的形式：$|M|=(m_{11},-m_{12},m_{13})\cdot(M_{11},M_{12},M_{13})$。$M_{ij}$项是什么？$M_{ij}$被称为矩阵$M$关于元素$m_{ij}$的余子式。由于在求解矩阵逆的过程中会得到余子式，所以可以很方便的定义它。

定义：方阵的每个元素m_{ij}都有一个与之对应的余子式M_{ij}。相应的余子式是把矩阵M中的第i行和第j列去掉所得的矩阵的行列式与$(-1)^{i+j}$的乘积。

注意，当在一行或一列上移动时，$(-1)^{i+j}$的值在1和-1之间交替变化。可以用一个棋盘来形象的表示1和-1在矩阵上的分布。计算任意一个$n\times n$的矩阵M的行列式$|M|$的一般规则如下：

取M中的任意一行，计算该行中每个元素对应的余子式，然后将该行与由余子式构成的n元组进行点积。同样，也可以选择M中的一列并执行相同的规则。(这个规则对于2×2的矩阵成立吗?)

下面是有关行列式的一些有用的性质：

- $|M|=|M^{T}|Z$。
- 如果M中的两行(或两列)相同，那么$|M|=0$。
- 如果M和B都是方阵，那么$|MB|=|M||B|$。
- 如果B是由M中的两行(或列)构成的，那么$|B|=-|M|$。
- 如果B是由M通过把M中的一行(或列)乘以一个常数k得到的，那么$|B|=k|M|$。
- 如果B是由M通过把M中的一行(或列)加上令一行(或列)构成的，那么$|B|=|M|$。

A2.1.5 矩阵的逆

如果$|M|$不等于0，就称$n\times n$的矩阵M是非奇异的。在这种情况下，M有逆矩阵，记为M^{-1}，具有如下性质

$$MM^{-1}=M^{-1}M=I$$

其中I是$n\times n$的单位矩阵。同样，方阵乘积的逆是

$$(AB)^{-1}=B^{-1}A^{-1}$$

用M的余子式来表示M^{-1}的元素是很简单的。记A为M的逆。那么A的第ij个元素

$$a_{ij}=\frac{M_{ji}}{|M|}$$

也就是说我们计算m_{ji}项的余子式，再除以整个矩阵的行列式。仔细的注意这里的下标：计算a_{ij}的时候用m_{ji}的余子式。计算M行列式的一个等价步骤如下：

(1) 建立一个由余子式$c_{ij}=M_{ij}$构成的中间矩阵。

(2) 用C的任何一行和M中对应行的点积求出$|M|$。

(3) 对 C 进行转置得到 C^T。

(4) 将 C^T 中的所有元素用 $|M|^{-1}$[①] 进行缩放得到 M^{-1}。

示例

求 $M=\begin{pmatrix} 2 & 0 & 6 \\ 8 & 1 & -4 \\ 0 & 5 & 7 \end{pmatrix}$ 的逆矩阵

解：

建立以 M 的余子式为元素构成的矩阵 C：

$$\begin{pmatrix} 27 & -56 & 40 \\ 30 & 14 & -10 \\ -6 & 56 & 2 \end{pmatrix}$$

然后用 $(2,0,6)\cdot(27,-56,40)=294$ 算出 $|M|$。然后对 C 转置，并用 $1/|M|$ 缩放所有的元素得到

$$M^{-1}=\frac{1}{294}\begin{pmatrix} 27 & 30 & -6 \\ -56 & 14 & 56 \\ 40 & -10 & 2 \end{pmatrix}$$

计算乘积 $M^{-1}M$ 和 MM^{-1} 看是否等于单位矩阵来验证这个结果。

正交矩阵

对于有些变换，例如旋转（见第 5 章），相关的矩阵都有一个很容易求解的逆矩阵。如果简单的将一个矩阵转置就可以产生它的逆矩阵，也就是 $M^T=M^{-1}$，就称该矩阵是一个正交矩阵。因此，$MM^T=I$。

如果 M 是正交的，$MM^T=I$ 说明 M 的每一行都是具有单位长度的向量，而且各行相互正交。对 M 的各列上述结论依然成立。（为什么？）例如，如果 M 是 3×3 的，以如下方式对其分块：

$$M=\begin{pmatrix} \mathbf{a} \\ \mathbf{b} \\ \mathbf{c} \end{pmatrix}$$

那么，三元组 $\mathbf{a}$、$\mathbf{b}$ 和 $\mathbf{c}$ 都是单位长度并且 $\mathbf{a}\cdot\mathbf{b}=\mathbf{a}\cdot\mathbf{c}=\mathbf{b}\cdot\mathbf{c}=0$。

A2.2 向量及其运算的一些性质

A2.2.1 向量的正交、正交点积

正交向量和正交点积只适用于二维向量。正如第 4 章所看到的，向量的正交和正交点积提供了求解两个方程两个未知数的有用的工具。

---

① 原书中为 $|M|^T$，应该是一处小错误。

向量的正交

令向量 $\mathbf{a}=(ax,ay)$。那么,$\mathbf{a}$ 沿逆时针方向旋转得到垂线,或"正交向量",记为 $\mathbf{a}^{\perp}$。定义为 $\mathbf{a}^{\perp}=(-ay,ax)$,满足如下性质:

a. 向量 $\mathbf{a}$ 和 $\mathbf{a}^{\perp}$ 具有相同的长度:$|\mathbf{a}|=|\mathbf{a}^{\perp}|$。

b. 线性性质:$(\mathbf{a}+\mathbf{b})^{\perp}=|\mathbf{a}^{\perp}|+|\mathbf{b}^{\perp}|$ 且 $(A\mathbf{a})^{\perp}=A\mathbf{a}^{\perp}$ 对于任意标量 A 成立。

c. 向量正交两次就得到原向量的负向量:$\mathbf{a}^{\perp\perp}=(\mathbf{a}^{\perp})^{\perp}=-\mathbf{a}$。

正交点积 $\mathbf{a}^{\perp}\cdot\mathbf{b}$

a. 正交点积 $\mathbf{a}^{\perp}\cdot b=a_xb_y-a_yb_x$ 对于任意的二维向量 $\mathbf{b}=(\mathbf{b}_x,\mathbf{b}_y)$[①]成立。

b. $\mathbf{a}^{\perp}\cdot\mathbf{a}=0$。($\mathbf{a}^{\perp}$ 垂直于 $\mathbf{a}$)

c. $|\mathbf{a}^{\perp}|^2=|\mathbf{a}|^2$。($\mathbf{a}^{\perp}$ 和 $\mathbf{a}$ 具有相同的长度)

d. $\mathbf{a}^{\perp}\cdot\mathbf{b}=-\mathbf{b}^{\perp}\cdot\mathbf{a}$。(正交点积是反对称的)

e. $\mathbf{a}^{\perp}\cdot\mathbf{b}$ 可以被写成行列式形式

$$\mathbf{a}^{\perp}\cdot\mathbf{b}=\begin{vmatrix} a_x & a_y \\ b_x & b_y \end{vmatrix}$$

f. 如果 $\mathbf{a}+\mathbf{b}+\mathbf{c}=\mathbf{0}$,那么 $\mathbf{a}^{\perp}\cdot\mathbf{b}=\mathbf{b}^{\perp}\cdot\mathbf{c}=\mathbf{c}^{\perp}\cdot\mathbf{a}$。

g. 当且仅当存在 $\mathbf{a}$ 到 $\mathbf{b}$ 的 CCW 旋转时 $\mathbf{a}^{\perp}\cdot\mathbf{b}>0$。

h. 如果 $\mathbf{b}$ 与 $\mathbf{a}$ 平行或者反平行,则 $\mathbf{a}^{\perp}\cdot\mathbf{b}=0$。

i. $|\mathbf{a}^{\perp}\cdot\mathbf{b}|$ 是向量 $\mathbf{a}$ 和 $\mathbf{b}$ 确定的平行四边形的面积。

A2.3 球坐标和方向余弦

本节回顾球坐标的概念并总结如何在球坐标和笛卡儿坐标之间进行相互转换。图 A2.1 图示了在球坐标中是如何定义点 U 的。R 是 U 到原点的径向距离,ϕ 是 U 到 xz 平面的夹角,称为点 U 的纬度。θ 是 U 的方位角,xy 平面与经过 U 和 y 轴的平面所成的角。ϕ 的范围是 $-\pi/2\leqslant\varphi<\pi/2$,$\theta$ 的范围是 $0\leqslant\theta<2\pi$。

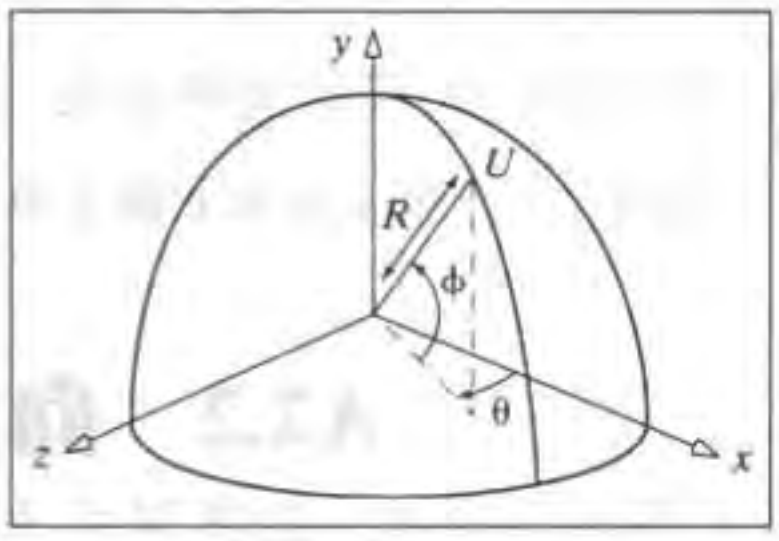


图 A2.1 球坐标的定义

使用简单的三角函数,可以直接算出这些量和 U 的笛卡儿坐标(u_x,u_y,u_z)之间的关系。方程为:

$$u_x=R\cos(\phi)\cos(\phi)$$
$$u_y=R\sin(\phi) \qquad (A2.3)$$

和

$$u_z=R\cos(\phi)\sin(\phi)$$

① 原书为 $\mathbf{b}=(\mathbf{b}_x,\mathbf{b}_y)=a_x\mathbf{b}_y-a_y\mathbf{b}_x$,应该是一处小错误。

也可以反过来用 u_x, u_y, u_z 来表示(R, ϕ, θ)：

$$R = \sqrt{u_x^2 + u_y^2 + u_z^2}$$

$$\phi = \sin^{-1}\left(\frac{u_y}{R}\right)$$

$$\theta = \arctan(u_z, u_x) \tag{A2.4}$$

函数 arctan (,)是两个变量形式的反正切函数，定义如下

$$\arctan(y, x) = \begin{cases} \tan^{-1}(y/x) & \text{如果 } x > 0 \\ \pi + \tan^{-1}(y/x) & \text{如果 } x < 0 \\ \pi/2 & \text{如果 } x = 0 \text{ 且 } y > 0 \\ -\pi/2 & \text{如果 } x = 0 \text{ 且 } y < 0 \end{cases} \tag{A2.5}$$

这个函数可以区分 x、y 都是正数和 x、y 都是负数的情况，不像通常的形式$\tan^{-1}(x/y)$，后者只能得到$-\pi/2$ 到 $\pi/2$ 之间的角。注意 x 是零而 y 不是零的情况，这将导致可怕的除数为零的情况。这种情况被小心地避免了。

例 2.1

假设点 U 和原点的距离为 2，在 xz 平面上偏 60°，并且沿 x 轴反方向。因此 U 在 xy 平面内。所以 U 的球坐标表示为(2,60°,180°)。用等式(A2.3)可以计算 U 的笛卡儿坐标，可得 $U=(-1,1.732,0)$。

方向余弦

上面例子中，点 U 的方向用两个角的形式给出：方位角和纬度。方向通常可以用另一个有用的方法——方向余弦来表示。过原点的直线的方向余弦是其分别与 x、y 和 z 轴三个夹角的余弦。回顾一下，两个单位向量夹角的余弦是由它们的点积给出的。用给定的点 U，我们可以得到位置向量(u_x, u_y, u_z)。从前述的讨论中可以看到这个向量的长度是 R，所以它必须被标准化到具有单位长度的向量 $\mathbf{m}=(u_x/R, u_y/R, u_z/R)$。所以它和 x 轴夹角的余弦可以通过点积 $\mathbf{m} \cdot i = u_x/R$ 给出，后者仅是 $\mathbf{m}$ 的第一个分量。类似的，$\mathbf{m}$ 的第二和第三个分量分别是方向余弦的第二和第三项。记与 x、y 和 z 轴构成的角度分别为 α、β 和 γ，就可以得出从 $\mathbf{0}$ 到 U 这条线的三个方向余弦：

$$\cos(\alpha) = \frac{u_x}{R}$$

$$\cos(\beta) = \frac{u_y}{R}$$

且

$$\cos(\gamma) = \frac{u_z}{R}$$

注意，这 3 个方向余弦是相关的，它们的平方和总是 1。

计算向量 $\mathbf{n}$ 的方向余弦，其中

1. $\mathbf{n}=(1,1,1)$。
2. $\mathbf{n}=(2,3,4)$。

附录 3

有用的类、例程以及 SDL

本附录定义了一些数据类型和算法，可用于开发图形学应用程序。基本的类型是通过各种类描述的，其中有些有完整的定义，另外的一些则只声明了大量的方法而没有给出定义。这有待读者进一步充实。有些类只给出了框架形式，以便说明在实际应用程序中那些可以被进一步改进。大部分的类采用简单的封装方法：为了简明和避免定义大量的读取和访问函数，大部分的数据域被声明为公有而非私有。另外一些优秀的类和与应用程序资料可以在“Graphics Gems”系列中找到，其在线存储库是 http：//tog. acm. org/GraphicsGems/。

本附录所定义的类收集在各种头文件(.h)和源文件(.cpp)中，它们在本书的网站上也能找到。(见前言。)由于某些代码片段可能被改进或者随时间调整，请参考对应的网站。对应的网站上也有一些程序员可能会觉得有用的附加代码。

本书中所描述的类的集合

1. 2D 图形类。这些类提供了对绘制 2D 图的支持，包括 IntPoint、Point2、Polyline、IntRect、Vector2 和 Canvas。

2. RGB 像素图。该类提供了对创建和绘制像素图的支持，包括读入一个图像以及存储为 BMP 格式。

3. SDL。这些类提供了对处理和绘制 3D 场景的支持，包括 SDL 中描述的场景。支持的类包括 Point3、Vector3、Color3、Light、Affine4、AffineStack、Material、GeomObj、Boolean、UnionBool、IntersectionBool、DifferenceBool、Shape、Cube、Sphere、TaperedCylinder、Square、Plane、Face、Mesh、Torus、Teapot 和 Scene。

4. 噪声。这些类中包括对固体纹理创建 3D 噪音和扰动的 Noise 类。

5. 光线跟踪类。包括 PointCluster、SphereInfo、Cuboid、Ray、HitInfo 和 Intersection。

A3.1 2D 图形类

A3.1.1 基本 2D 图形类

```
//graphics2d.h                                  //支持 2D 图形的类的集合
#ifndef _GRAPHICS2D
#define _GRAPHICS2D
#include <string>
```

```
#include <iostream>
#include <fstream>
#include <strstream>
using namespace std;
#include <windows.h>                    //如果使用 xWindows 或其他平台则需要改变这里
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>

//@@@@@@@@@@@@IntPoint class @@@@@@@@@@@@@@@@@
class IntPoint{                         //整型坐标的 2D 点
public:
int x,y;
void set(int dx, int dy){x=dx; y=dy;}
void set(IntPoint& p){ x=p.x; y=p.y;}
IntPoint(int xx, int yy){x=xx; y=yy;}
IntPoint(){ x=y=0;}
};
//@@@@@@@@@@@@@@Point2 class @@@@@@@@@@@@@@@@@
class Point2{                           //实数坐标的 2D 点
public:
float x,y;
void set(float dx, float dy){x=dx; y=dy;}
void set(Point2& p){ x=p.x; y=p.y;}
Point2(float xx, float yy){x=xx; y=yy;}
Point2(){x=y=0;}
};
//<<<<<<<<<<<<<<<<<<<<<<<<PolyLine >>>>>>>>>>>>>>>>>>>>>>>>>>
class PolyLine{                         //一个多边形是一个数①加上点的数组
public:
int num;
Point2 pt[80];                          //在某些情况下可能需要更大的数组
PolyLine(){num=0;}
};
//@@@@@@@@@@@@@@@IntRect class @@@@@@@@@@@@@@@@
class IntRect{                          //边界值为整数的矩形
public:
  int left, top, right, bott;
  IntRect(){left=top=right=bott=0;}
  IntRect(int l, int t, int r, int b)
```

① 原书中是“num”，此处译为“数”，指的是类中的代表多边形中有效点个数的整数变量 num。

```
  {left=l; top=t; right=r; bott=b;}
  void set(int l, int t, int r, int b)
  {left=l; top=t; right=r; bott=b;}
  void set(IntRect& r)
  {left=r.left; top=r.top; right=r.right; bott=r.bott;}
};
//@@@@@@@@@@@@@@Vector2 class @@@@@@@@@@@@@@@@
class Vector2{
public:
float x,y;
void set(float dx, float dy){ x=dx; y=dy; }
void set(Vector2& v){ x=v.x; y=v.y;}
void setDiff(Point2& a, Point2& b)        //设为 a-b 的差
{x=a.x-b.x; y=a.y-b.y;}
void normalize()                          //调整该向量为单位长度
{double sizeSq=x * x+y * y;
if(sizeSq<0.0000001)
{
cerr <<"\nnormalize() sees vector (0,0)!";
return;                                   //对零向量不做任何操作
}
float scaleFactor=1.0/(float)sqrt(sizeSq);
x * =scaleFactor; y * =scaleFactor;
}
Vector2(float xx, float yy){x=xx; y=yy; }
Vector2(Vector2& v){x=v.x; y=v.y; }
Vector2(){x=y=0;}                         //默认构造函数
float dot(Vector2 b)                      //返回和 b 点积的结果
{return x * b.x+y * b.y;}
void perp()                               //perp this vector
{float tmp=x; x=-y; y=tmp;}
float perpDot(Vector2& v)                 //返回和 v 正交点积的结果
{return x * v.x-y * v.y;}
};
//<<<<<<<<<<<<<<<<<<<<<<<Canvas class >>>>>>>>>>>>
//全局 Canvas 对象(见第三章描述)知道如何在世界坐标系中画线①
class Canvas {
private:
Point2 CP;                                //在世界中的当前位置
float CD;                                 //以角度表示的当前方向
public:
float windowAspect;
```

① 原书中此类有错误："class Canvas {"这一块误写为"// Beginning of graphics2d. h"且原书中漏掉了"class Canvas"，根据代码这里不应该是 graphics2d. h 的起始。

```
Canvas(int width, int height, char* title);
void setWindow(float l, float r, float b, float t);
void setViewport(int l, int r, int b, int t);
float getWindowAspect(void);
void lineTo(float x, float y);
void moveTo(float x, float y){CP.x=x; CP.y=y;}
void forward(float dist, int vis);
void initCT()                          //初始化 CT(模型试图矩阵)
{
glMatrixMode(GL_MODELVIEW); glLoadIdentity();
}
void rotate2D(double angle)
{
glMatrixMode(GL_MODELVIEW); glRotated(angle, 0.0, 0.0, 1.0);
  }
void translate2D(double dx, double dy)
{
glMatrixMode(GL_MODELVIEW); glTranslated(dx, dy, 0.0);
}
void scale2D(double sx, double sy)
{
glMatrixMode(GL_MODELVIEW); glScaled(sx, sy, 1.0);
}
void pushCT(void)
{
glMatrixMode(GL_MODELVIEW); glPushMatrix();
}
void popCT(void)
{
glMatrixMode(GL_MODELVIEW); glPopMatrix();
}
void ngon(int n, float cx, float cy, float radius);
};①

//graphics2d.cpp----使用#include "graphics2d.h"
//graphics2d.h 中定义的方法
#include "graphics2d.h"
Canvas::Canvas(int width, int height, char* title)
{
char* list;                            //glutInit 的虚列表
int numArgs=1;                         //glutInit 的虚赋值
glutInit(&numArgs, &list);
```

① 原书此处有好几行的缺失，根据和第二版的对比，对此处做了较大的修正。由于修改点太多，所以不一一赘述，有兴趣的读者可以参考原版。

```
glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize(width, height);
glutInitWindowPosition(100, 100);
glutCreateWindow(title);
CP.x=CP.y=0.0;
windowAspect=1.0;
}
void Canvas::setWindow(float l, float r, float b, float t)
{
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D((GLdouble)l, (GLdouble)r, (GLdouble)b, (GLdouble)t);
if(t==b) return;
windowAspect=(r-l)/(t-b);
}
void Canvas::setViewport(int l, int r, int b, int t)
{
glViewport((GLint)l, (GLint)b, (GLint)(r-l), (GLint)(t-b));
}
void Canvas::lineTo(float x, float y)
{
glBegin(GL_LINES);
glVertex2f((GLfloat)CP.x, (GLfloat)CP.y);
CP.x=x; CP.y=y;
glVertex2f((GLfloat)CP.x, (GLfloat)CP.y);
glEnd(); glFlush();
}
void Canvas::forward(float dist, int vis)
{
#define RadPerDeg 0.017453393                    //每度的弧度数
float x=CP.x+dist*cos(RadPerDeg*CD);
float y=CP.y+dist*sin(RadPerDeg*CD);
if(vis) lineTo(x, y);
else moveTo(x, y);
CP.x=x; CP.y=y;
}
void Canvas::ngon(int n, float cx, float cy, float radius)
{
#define RadPerDeg 0.017453393                    //每度的弧度数
if (n<3) return;                                 //错误的边数
double angle=0, angleInc=2*3.14159265/n;         //角度增加
moveTo(cx+radius, cy);
for(int k=1; k<=n; k++)
{
Angle+=angleInc;
```

```
lineTo(radius * cos(angle)+cx, radius * sin(angle)+cy);
}
}
//graphics2d.cpp 结束
```

RGBPixmap 类

```
//RGBpixmap.h: 支持使用 RGB 像素映射的类
#ifndef _RGBPIXMAP
#define _RGBPIXMAP
#include <fstream>
//需要定义 IntPoint 和 IntRect 类
typedef unsigned char uchar;
class mRGB{                                    //RGB 这个名字已经被 Windows 使用了
public: uchar r,g,b;
mRGB(){r=g=b=0;}
mRGB(mRGB& p){r=p.r; g=p.g; b=p.b;}
mRGB(uchar rr, uchar gg, uchar bb){r=rr; g=gg; b=bb;}
void set(uchar rr, uchar gg, uchar bb){r=rr; g=gg; b=bb;}
};
//$$$$$$$$$$$$$$$$$$RGBPixmap 类 $$$$$$$$$$$$$$$$
class RGBpixmap{
private:
mRGB * pixel;                              //像素数组
public:
int nRows, nCols;                          //像素映射的维数
RGBpixmap() {nRows=nCols=0; pixel=0;}
RGBpixmap(int rows, int cols)              //构造函数
{
nRows=rows;
nCols=cols;
pixel=new mRGB[rows * cols];
}
int readBMPFile(string fname);             //将 BMP 文件读入该像素映射
void freeIt()                              //从该像素映射中恢复内存
{
delete []pixel; nRows=nCols=0;
}
//<<<<<<<<<<<<<<<<<<<<复制 >>>>>>>>>>>>>>>>>>>>
void copy(IntPoint from, IntPoint to, int x, int y, int width, int height)
{ //把显示器区域的一部分复制回显示器
if(nRows ==0 || nCols ==0) return;
glCopyPixels(x, y, width, height,GL_COLOR);
}
//<<<<<<<<<<<<<<<<<<<<绘制 >>>>>>>>>>>>>>>>>>>>
```

```
void draw()
{ //在当前光栅位置绘制像素映射
if(nRows == 0 || nCols == 0) return;
//通知 OpenGL: 不要在内存中对像素进行 4 字节边界对齐
glPixelStorei(GL_UNPACK_ALIGNMENT,1);
glDrawPixels(nCols, nRows,GL_RGB, GL_UNSIGNED_BYTE,pixel);
}
//<<<<<<<<<<<<<<<<<<读取 >>>>>>>>>>>>>>>>>>
int read(int x, int y, int wid, int ht)
{                                          //读取像素矩形到像素映射
nRows=ht;
nCols=wid;
pixel=new mRGB[nRows * nCols]; if(!pixel) return -1;
//通知 OpenGL: 不要在内存中对像素进行 4 字节边界对齐
glPixelStorei(GL_PACK_ALIGNMENT,1);
glReadPixels(x, y, nCols, nRows, GL_RGB,GL_UNSIGNED_BYTE,pixel);
return 0;
}
//<<<<<<<<<<<<<<<<<从 IntRect 中读取 >>>>>>>>>>>>>>>>>
int read(IntRect r)
{ //读入像素矩形到像素映射表中
nRows=r.top-r.bott;
nCols=r.right-r.left;
pixel=new mRGB[nRows * nCols]; if(!pixel) return -1;
//通知 OpenGL: 不要在内存中对像素进行 4 字节边界对齐
glPixelStorei(GL_PACK_ALIGNMENT,1);
glReadPixels(r.left,r.bott, nCols, nRows, GL_RGB, GL_UNSIGNED_BYTE, pixel);
return 0;
}
//<<<<<<<<<<<<<<<setPixel >>>>>>>>>>>>>>
void setPixel(int x, int y, mRGB color)
{
if(x>=0 && x <nCols && y >=0 && y<nRows)
pixel[nCols * y+x]=color;
}
//<<<<<<<<<<<<<<<<<getPixel >>>>>>>>>>>
mRGB getPixel(int x, int y)
{
mRGB bad(255,255,255);
assert(x >=0 && x<nCols);
assert(y >=0 && y<nRows);
return pixel[nCols * y+x];
}
};                                         //RGBpixmap 类结束
#endif
```

```
//RGBpixmap.cpp-读取 BMP 文件的例程
#include "RGBpixmap.h"
typedef unsigned short ushort;
typedef unsigned long ulong;
fstream inf;                          //为了方便,在这个文件中是全局的
//<<<<<<<<<<<<<<<<<<<<<<getShort >>>>>>>>>>>>>>>>>>>>>
ushort getShort()                     //辅助函数
{ //使用低位在前存储方式整型的 BMP 格式
  //获取一个低位在前存贮方式的 2 字节整数
  char ic;
  ushort ip;
  inf.get(ic); ip=ic;                 //第一个字节是低位
  inf.get(ic);  ip |= ((ushort)ic <<8); //或在高字节①
  return ip;
  }
//<<<<<<<<<<<<<<<<<<<<<<getLong >>>>>>>>>>>>>>>>>>>>>
ulong getLong()                       //辅助函数
{                                     //使用低位在前存储方式整型的 BMP 格式
                                      //获取一个低位在前存贮方式的 4 字节整数
  ulong ip=0;
  char ic=0;
  unsigned char uc=ic;
  inf.get(ic); uc=ic; ip=uc;
  inf.get(ic); uc=ic; ip |= ((ulong)uc <<8);
  inf.get(ic); uc=ic; ip |= ((ulong)uc <<16);
  inf.get(ic); uc=ic; ip |= ((ulong)uc <<24);
  return ip;
  }
//<<<<<<<<<<<<<<<<<<<<RGBPixmap:: readBmpFile>>>>>>>>>>>>>>
int RGBpixmap:: readBMPFile(string fname)
{ //从未压缩的 BMP 文件中读入 mRGB 图像到内存中
//失败返回 0,成功返回 1
inf.open(fname.c_str(), ios::in|ios::binary);   //读二进制字符串
if(!inf){ cout <<" can't open file: " <<fname <<endl; return 0;}
int k, row, col, numPadBytes, nBytesInRow;
//读入文件头信息
char ch1, ch2;
inf.get(ch1); inf.get(ch2);           //类型: 总是'BM'
ulong fileSize=getLong();
ushort reserved1=getShort();          //总是 0
ushort reserved2=getShort();          //总是 0
ulong offBits =getLong();             //到图像数据的偏移量-不可靠
ulong headerSize=     getLong();      //总是 40
```

① 原文是"or in high order byte",此段代码是把高字节赋值给 ip 的高位。

```
ulong numCols =getLong();                    //图像中的列数
ulong numRows=getLong();                     //图像中的行数
ushort planes=     getShort();               //总是 1
ushort bitsPerPixel=   getShort();           //8 或 24; 这里允许是 24
ulong compression=   getLong();              //对未压缩的图片必须是 0
ulong imageSize=getLong();                   //图像的总字节数
ulong xPels=   getLong();                    //总是 0
ulong yPels=   getLong();                    //总是 0
ulong numLUTentries=getLong();               //对 8 位是 256,否则为 0
ulong impColors=getLong();                   //总是 0
if(bitsPerPixel !=24)
{ //错误-必须是 24 位的未压缩图像
cout <<"not a 24 bit/pixelimage, or is compressed! \n";
inf.close(); return 0;
}
//给每行加入若干字节以保证总字节数是 4 的倍数
//对 3* numCols 上舍入到 4 的倍数
nBytesInRow= ((3* numCols+3)/4) * 4;
numPadBytes=nBytesInRow-3* numCols;          //需要这么多
nRows=numRows;                               //设置类的数据成员
nCols=numCols;
pixel=new mRGB[nRows * nCols];               //为数组申请空间
if(!pixel) return 0;                         //内存溢出!
long count=0;
char dum;
for(row=0; row<nRows; row++)                 //读入像素值
{
for(col=0; col<nCols; col++)
{
char r,g,b;
inf.get(b); inf.get(g); inf.get(r);          //读入字节
pixel[count].r=r;                            //设置颜色
pixel[count].g=g;
pixel[count++].b=b;
}
   for(k=0; k<numPadBytes ; k++)             //跳过行尾的扩充字节
inf >>dum;
}
inf.close(); return 1;                       //成功
}
```

A3.1.2 场景和支持类

```
//SDL.h
```

```
//简单支持类的定义
#fndef _SDL
#define _SDL
#include <string>
#include <iostream>
#include <fstream>
#include <strstream>
using namespace std;
#include <windows.h>
#include <assert.h>
#include <math.h>
#include <gl/Gl.h>
#include <gl/Glu.h>
#include <gl/glut.h>
//如果要在场景中加入像素映射域,应该包括 RGBpixmap
#include "RGBpixmap.h"
//@@@@@@@@@@@@@@@@Point3类@@@@@@@@@@@@@@@@
class Point3{
public:
float x,y,z;
void set(float dx, float dy, float dz){x=dx; y=dy; z=dz;}
void set(Point3& p){x=p.x; y=p.y; z=p.z;}
Point3(float xx, float yy, float zz){x=xx; y=yy; z=zz;}
Point3(){x=y=z=0;}
void build4tuple(float v[])
{//用当前颜色建立 4元组:考虑齐次性 v[3]=1
v[0]=x; v[1]=y; v[2]=z; v[3]=1.0f;
}
};
//@@@@@@@@@@@@@@@@@@Vector3类 @@@@@@@@@@@@@@@@@
class Vector3{
public:
float x,y,z;
void set(float dx, float dy, float dz){ x=dx; y=dy; z=dz;}
void set(Vector3& v){ x=v.x; y=v.y; z=v.z;}
void flip(){x=-x; y=-y; z=-z;}          //翻转该向量
void setDiff(Point3& a, Point3& b)      //设为 a-b
{ x=a.x-b.x; y=a.y-b.y; z=a.z-b.z;}
void normalize();                       //调整该向量到单位长度
Vector3(float xx, float yy, float zz){x=xx; y=yy; z=zz;}
Vector3(Vector3& v){x=v.x; y=v.y; z=v.z;}
Vector3(){x=y=z=0;}                     //默认构造函数
Vector3 cross(Vector3 b);               //返回和 b 叉乘的结果
float dot(Vector3 b);                   //返回和 b 点积的结果
};
```

```
//@@@@@@@@@@@@@@@@@@Color3类 @@@@@@@@@@@@@@@@@
class Color3 {                                  //存储红、绿、蓝三元组
public:
float red, green, blue;
Color3(){red=green=blue=0;}
Color3(float r, float g, float b){red=r; green=g; blue=b;}
Color3(Color3& c){red=c.red; green=c.green; blue=c.blue;}
void set(float r, float g, float b){red=r; green=g; blue=b;}
void set(Color3& c)
      {red=c.red; green=c.green; blue=c.blue;}
void add(float r, float g, float b)
      {red+=r; green+=g; blue+=b;}
void add(Color3& src, Color3& refl);
void add(Color3& colr);
void build4tuple(float v[]);
};
//@@@@@@@@@@@@@@@@@@light类 @@@@@@@@@@@@@@@@@@
class Light{                                    //用于光源颜色位置的连接表
public:
Point3 pos;
Color3 color;
Light * next;
void setPosition(Point3 p){pos.set(p);}
void setColor(Color3 c){color.set(c);}
Light(){next=NULL;}
};
//@@@@@@@@@@@@@@@@@@Affine4类@@@@@@@@@@@@@@@@@
class Affine4 {                                 //管理齐次仿射变换
//包括逆变换
//和保存他们的栈
//被 Scence 类用来读取 SDL 文件
public:
float m[16];                                    //保存 4×4 的矩阵
Affine4();
void setIdentityMatrix();
void set(Affine4 a);
void preMult(Affine4 n);
void postMult(Affine4 n);
};                                              //end of Affine4 class
//@@@@@@@@@@AffineNode类 @@@@@@@@@@@
class AffineNode{
//被 Scence 类用来读取 SDL 文件
public:
Affine4 * affn;
Affine4 * invAffn;
```

```
AffineNode * next;
AffineNode()
{
next=NULL;
affn=new Affine4;                    //新申请一个 affine 并赋初值为单位矩阵
invAffn=new Affine4;                 //对逆变换也做相同的事情
}
~AffineNode()                        //析构函数
{
delete affn;
delete invAffn;
}
};
//@@@@@@@@@@@@@@@@AffineStack 类 @@@@@@@@@@@@
class AffineStack{
//被 Scence 类用来读取 SDL 文件
public:
AffineNode * tos;
AffineStack()                        //默认构造函数,把单位阵放在栈顶
{
tos=new AffineNode;                  //包含单位阵的节点
tos->next=NULL;
}
void dup();
void setIdentity();                  //设置栈顶元素为单位矩阵
void popAndDrop();
void releaseAffines();               //弹出并删除所有剩余元素
void rotate(float angle, Vector3 u);
void scale(float sx, float sy, float sz);
void translate(Vector3 d);
};                                   //AffineStack 类结束
//@@@@@@@@@@@@@@@@Material 类 @@@@@@@@@@@@@@
class Material{
public:
Color3 ambient, diffuse, specular, emissive;
int numParams;                       //用于纹理
float params[10];                    //用于纹理
int textureType;                     //0 代表无,负数代表固体,正数代表图片
float specularExponent, reflectivity, transparency, speedOfLight;
float specularFraction, surfaceRoughness;
void setDefault();
void set(Material& m);
};                                   //Material 类结束
//@@@@@@@@@@@@@@@@@GeomObj 类 @@@@@@@@@@@@@@@@@
class GeomObj{
```

```
public:
//IntRect scrnExtnt;
GeomObj * next;
GeomObj(): next(NULL){}
virtual void loadStuff(){}
virtual void drawOpenGL(){}
virtual void tellMaterialsGL(){}
};
//@@@@@@@@@@@@@@@@@@Boolean @@@@@@@@@@@@@@@@@@@
class Boolean: public GeomObj{
public:
GeomObj * left, * right;
Boolean():left(NULL),right(NULL){}
virtual void drawOpenGL()
{ //只绘制它的孩子
if(left)left->drawOpenGL();
if(right)right->drawOpenGL();
}
};
//@@@@@@@@@@@@@@@@@@UnionBool @@@@@@@@@@@@@@@@@@
class UnionBool: public Boolean{
public:
UnionBool(){Boolean();}                    //构造函数
};
//@@@@@@@@@@@@@@@@@@IntersectionBool @@@@@@@@@@@@@@@@@
class IntersectionBool: public Boolean{
public:
IntersectionBool(){Boolean();}
};
//@@@@@@@@@@@@@@@@@@DifferenceBool @@@@@@@@@@@@@@@@@@
class DifferenceBool: public Boolean{
public:
DifferenceBool(){Boolean();}
};
//@@@@@@@@@@@@@@@@@@Shape @@@@@@@@@@@@@@@@@@@
class Shape: public GeomObj{
public:
Material mtrl;
Affine4 transf,invTransf;
//virtual Color3 texture(HitInfo& h, int whichTexture);
Shape(){mtrl.textureType=0; mtrl.numParams=0;}
void setMaterial(Material& mt){mtrl.set(mt);}
void tellMaterialsGL();
};                                         //结束: Shape类
//@$@$@$@$@$@$@$@$@Cube类 $@$@$@$@$@$@$@$@$@$@$@
```

```
class Cube: public Shape{
public:
Cube(){}
void drawOpenGL()
{
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);                    //调用仿射
glEnable(GL_NORMALIZE);
if(doEdges) glutWireCube(2.0);
if(doFaces) glutSolidCube(2.0);             //一个以-1到1为顶点的立方体
glPopMatrix();
}
};
//@$@$@$@$@$@$@$@$Sphere类 @$@$@$@$@$@$@$@$@$@$@$@$@
class Sphere: public Shape{
public:
void drawOpenGL()
{
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
glutSolidSphere(1.0,20,20);
glPopMatrix();
}
Sphere() { }
};
//@$@$@$@$@$@$@$@$@$@TaperedCylinder类 @$@$@$@$@$@$@$@$@$@$
class TaperedCylinder: public Shape{
public:
float smallRadius;
TaperedCylinder(){}
void drawOpenGL();
};
//@$@$@$@$@$@$@$@$@$@Square类 @$@$@$@$@$@$@$@$@$@$
class Square: public Shape{
public:
Square(){}
void drawOpenGL() {                         //尚未实现};
};
//@$@$@$@$@$@$@$@$@$@Plane类 @$@$@$@$@$@$@$@$@$@$
class Plane: public Shape{
public:
Plane() {}
void drawOpenGL() {                         //尚未实现};
};
//##################class VertexID #################
```

```
//用于定义一个 Mesh
class VertexID{public: int vertIndex, normIndex;};
//##################class FACE ###############
//用于定义一个 Mesh
class Face{
public:
int nVerts;
VertexID * vert;                    //顶点数组和法线索引
Face(){ nVerts=0; vert=NULL;}
~Face(){delete[] vert; nVerts=0;}
};
//@$@$@$@$@$@$@$@$@Mesh 类 @$@$@$@$@$@$@$@$@$
class Mesh: public Shape{
private:
int numVerts, numNorms, numFaces;
Point3 * pt;                        //点数组
Vector3 * norm;                     //法线数组
Face * face;                        //面数组
int lastVertUsed;
int lastNormUsed;
int lastFaceUsed;
public:
void readMesh(string fname);
void writeMesh(char* fname);
void printMesh();
void drawMesh();
void drawEdges();
void freeMesh();
int isEmpty();
void makeEmpty();
Mesh();
virtual void drawOpenGL();
Mesh(string fname);
Vector3 newell4(int indx[]);
string meshFileName;                //保存这个 Mesh 的文件名
};                                  //Mesh 类结束
//@$@$@$@$@$@$@$@$@Torus class @$@$@$@$@$@$@$@$@$
class Torus: public Shape{
public:
void drawOpenGL(){
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
//if(doFaces) glutSolidTorus(0.2,1.0,10,12);
glPopMatrix();
}
```

```
};
//@$@$@$@$@$@$@$@$@$@Teapot class @$@$@$@$@$@$@$@$@$@$
class Teapot: public Shape{
public:
void drawOpenGL(){ tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
glutSolidTeapot(1.0); glPopMatrix();}
};
//@@@@@@@@@@@@@@DefUnit & DefUnitStack类 @@@@@@@@@@@@@@
//被 Scence 类用来读取 SDL 文件
class DefUnit {
//由 Steve Morin 开发
public:
string name, stuff;
DefUnit(string n, string s) {stuff=s;name=n;}
};
class DefUnitStack {
public:
DefUnitStack() { stack=NULL;}
void push(string n, string s);
void print();
int search(string s);
string contents(string s);
void release();
private:
struct D4S {
DefUnit * current;
struct D4S * next;
} d4s;
D4S * stack;
};                                    //DefUnitStack 类结束
//++++++++++++++TokenType ++++++++++++++
enum mTokenType {IDENT, LIGHT, ROTATE, TRANSLATE, SCALE, PUSH, POP, IDENTITYAFFINE,
GLOBALAMBIENT, BACKGROUND, MINREFLECTIVITY, MINTRANSPARENCY, MAXRECURSIONDEPTH, CUBE,
SPHERE, TORUS, PLANE, SQUARE, CYLINDER, CONE, TAPEREDCYLINDER, TETRAHEDRON, OCTAHEDRON,
DODECAHEDRON, ICOSAHEDRON, BUCKYBALL, TEAPOT, DIAMOND, UNION, INTERSECTION, DIFFERENCEa,
MAKEPIXMAP, MESH, DEFAULTMATERIALS, AMBIENT, DIFFUSE, SPECULAR, SPECULARFRACTION,
SURFACEROUGHNESS, EMISSIVE, SPECULAREXPONENT, SPEEDOFLIGHT, TRANSPARENCY, REFLECTIVITY,
PARAMETERS, TEXTURE, LFTCURLY, RGHTCURLY, DEF, USE, T_NULL, F_EOF, UNKNOWN };
//@@@@@@@@@@@@@@@@Scene 类 @@@@@@@@@@@@@@@@@@
class Scene{
public:
Light * light;                        //光源列表附在这里
GeomObj * obj;                        //物体列表附在这里
Color3 background, ambient;
```

```
int maxRecursionDepth;
//要有下面的纹理数据域就必须#include RGBpixmap
RGBpixmap pixmap[8];                    //附带像素映射的列表
float minReflectivity, minTransparency;
bool isInShadow(Ray& f);                //用于光线跟踪：留给读者实现
Scene():light(NULL),obj(NULL),tail(NULL)     //默认构造函数
{
currMtrl.setDefault();
background.set(0,0,0.6f);
ambient.set(0.1f,0.1f,0.1f);
minReflectivity=0.5;
minTransparency=0.5;
maxRecursionDepth=3;
}
Scene(string fname) {Scene(); read(fname); }
void freeScene();
void makeLightsOpenGL() {/* 尚未实现 */}
void drawSceneOpenGL();
bool read(string fname);
GeomObj * getObject();
private:
//用于读取场景的私有成员
int line;
int nextline;
ifstream  * file_in;
strstream * f_in;
strstream temp_fin;
DefUnitStack * def_stack;
GeomObj * tail;                         //物体列表尾指针
AffineStack affStk;                     //仿射栈
Material currMtrl;
string nexttoken(void);
float getFloat();
bool isidentifier(string keyword);
void cleanUp();
TokenType whichtoken(string keyword);
};                                      //Scene.h 结束
#endif
//SDL.cpp
//SDL.h 中类的支持代码
#include "SDL.h"
//Vector3 的方法
Vector3 Vector3:: cross(Vector3 b)      //返回和 b 叉乘的结果
{
   Vector3 c(y*b.z-z*b.y, z*b.x-x*b.z, x*b.y-y*b.x);
```

```
  return c;
}
float Vector3::dot(Vector3 b)                    //返回和 b 点乘的结果
{return x*b.x+y*b.y+z*b.z;}
void Vector3::normalize()                        //把该向量调整到单位长度
{
double sizeSq=x*x+y*y+z*z;
if(sizeSq<0.0000001)
{
cerr<<"\nnormalize() sees vector (0,0,0)!";
return;                                          //对零向量不做任何操作
}
float scaleFactor=1.0/(float)sqrt(sizeSq);
x*=scaleFactor; y*=scaleFactor; z*=scaleFactor;
}
//Color3 类的方法
void Color3::add(Color3& src, Color3& refl)
{ //加上原颜色和反射系数的乘积
red+=src.red*refl.red;
green+=src.green*refl.green;
blue+=src.blue*refl.blue;
}
void Color3::add(Color3& colr)
{ //把 colr 加到当前颜色
red+=colr.red ; green+=colr.green; blue+=colr.blue;
}
void Color3::build4tuple(float v[])
{//用当前颜色建立 4 元组：考虑齐次性 v[3]=1
v[0]=red; v[1]=green; v[2]=blue; v[3]=1.0f;
}
//Affine4 类的方法
Affine4::Affine4(){                              //建立单位变换
m[0]=m[5]=m[10]=m[15]=1.0;
m[1]=m[2]=m[3]=m[4]=0.0;
m[6]=m[7]=m[8]=m[9]=0.0;
m[11]=m[12]=m[13]=m[14]=0.0;
}
void Affine4::setIdentityMatrix(){               //建立单位变换
m[0]=m[5]=m[10]=m[15]=1.0;
m[1]=m[2]=m[3]=m[4]=0.0;
m[6]=m[7]=m[8]=m[9]=0.0;
m[11]=m[12]=m[13]=m[14]=0.0;
}
void Affine4::set(Affine4 a)                     //把当前矩阵设置为 a
{
```

```
for(int i=0; i<16; i++)
m[i]=a.m[i];
}
//<<<<<<<<<<<<preMult >>>>>>>>>>>>
void preMult(Affine4 n){                    //当前矩阵左乘以 n①
float sum;
Affine4 tmp;
tmp.set(* this);                            //tmp 副本
for(int c=0; c<4; c++)                      //form this=tmp * n
for(int r=0; r <4 ; r++)
{
sum=0;
for(int k=0; k<4; k++)
sum +=tmp.m[4 * k+r] *  n.m[4 * c+k];
m[4 * c+r]=sum;
}//end of for loops
}
//<<<<<<<<<<<<postMult >>>>>>>>>>>>
void postMult(Affine4 n){                   //当前矩阵右乘以 n
float sum;
Affine4 tmp;
tmp.set(* this);                            //tmp 副本
for(int c=0; c<4; c++)                      //form this=tmp * n
for(int r=0; r <4 ; r++)
{
sum=0;
for(int k=0; k<4; k++)
sum +=tmp.m[4 * k+r] *  n.m[4 * c+k];
m[4 * c+r]=sum;
}                                           //for 循环结束
}
//AffineStack 类的方法
void AffineStack::dup()
{
AffineNode *  tmp=new AffineNode;
tmp->affn=new Affine4(* (tos->affn));
tmp->invAffn=new Affine4(* (tos->invAffn));
tmp->next=tos;
tos=tmp;
}
void AffineStack::setIdentity()             //设置栈顶元素为单位阵
{
assert(tos !=NULL);
```

① 原书中此处为"postmultiplies this with n",根据后面代码可以看出此处应该是"premultiplies this with n"

```
tos->affn->setIdentityMatrix();
tos->invAffn->setIdentityMatrix();
}
void AffineStack::popAndDrop()
{
if(tos==NULL) return;                    //什么也不做
AffineNode * tmp=tos;
tos=tos->next;
delete tmp;                              //会调用析构函数来删除矩阵
}
void AffineStack::releaseAffines()
{                                        //弹出并释放所有剩余元素
while(tos) popAndDrop();
}
void AffineStack::rotate(float angle, Vector3 u)
{
Affine4 rm;                              //建立单位矩阵
Affine4 invRm;
u.normalize();                           //把旋转轴变为单位长度
float ang=angle*3.14159265/180;          //弧度变角度
float c=cos(ang), s=sin(ang);
float mc=1.0-c;
                                         //填充 3×3 的左上三角阵
rm.m[0]=c+mc*u.x*u.x;
rm.m[1]=mc*u.x*u.y+s*u.z;
rm.m[2]=mc*u.x*u.z-s*u.y;
rm.m[4]=mc*u.y*u.x-s*u.z;
rm.m[5]=c+mc*u.y*u.y;
rm.m[6]=mc*u.y*u.z+s*u.x;
rm.m[8]=mc*u.z*u.x+s*u.y;
rm.m[9]=mc*u.z*u.y-s*u.x;
rm.m[10]=c+mc*u.z*u.z;
//对于逆矩阵也一样,只是 s 的符号改变了
invRm.m[0]=c+mc*u.x*u.x;
invRm.m[1]=mc*u.x*u.y-s*u.z;
invRm.m[2]=mc*u.x*u.z+s*u.y;
invRm.m[4]=mc*u.y*u.x+s*u.z;
invRm.m[5]=c+mc*u.y*u.y;
invRm.m[6]=mc*u.y*u.z-s*u.x;
invRm.m[8]=mc*u.z*u.x-s*u.y;
invRm.m[9]=mc*u.z*u.y+s*u.x;
invRm.m[10]=c+mc*u.z*u.z;
tos->affn->postMult(rm);
tos->invAffn->preMult(invRm);
}
```

```
void AffineStack::scale(float sx, float sy, float sz)
{                                          //用比例因子右乘栈顶元素
#define sEps 0.00001
Affine4 scl;                               //建立一个单位阵
Affine4 invScl;
scl.m[0] =sx;
scl.m[5] =sy;
scl.m[10]=sz;                              //调整为一个缩放矩阵
if(fabs(sx)<sEps || fabs(sy)<sEps || fabs(sz)<
{
cerr <<"degenerate scaling transformation!\n"; );
}
invScl.m[0] =1/sx; invScl.m[5] =1/sy; m[10]=1/sz;
tos->affn->postMult(scl);                  //
tos->invAffn->preMult(invScl);
}
void AffineStack::translate(Vector3 d)
{
Affine4 tr;                                //建立一个单位阵
Affine4 invTr;
tr.m[12]=d.x; tr.m[13]=d.y;tr.m[14]=d.z;
invTr.m[12]=-d.x;invTr.m[13]=-d.y; [14]=-d.z;
tos->affn->postMult(tr);
tos->invAffn->preMult(invTr);
}
//Material 类的方法
void Material::setDefault(){
textureType=0;                             //无用
numParams=0;
reflectivity=transparency=0.0;
speedOfLight=specularExponent=1.0;
specularFraction=0.0;
surfaceRoughness=1.0;
ambient.set(0.1f,0.1f,0.1f);
diffuse.set(0.8f,0.8f,0.8f);
specular.set(0,0,0);
emissive.set(0,0,0);
}
void Material::set(Material& m)
{
textureType=m.textureType;
numParams=m.numParams;
for(int i=0; i<numParams; i++) params[i]=m.params[i];
transparency=m.transparency;
speedOfLight=m.speedOfLight;
```

```
reflectivity=m.reflectivity;
specularExponent=m.specularExponent;
specularFraction=m.specularFraction;
surfaceRoughness=m.surfaceRoughness;
ambient.set(m.ambient);
diffuse.set(m.diffuse);
specular.set(m.specular);
emissive.set(m.emissive);
}
//Shape 类的方法
void Shape::tellMaterialsGL()
{
float amb[4],diff[4],spec[4], emiss[4];
float zero[]={0,0,0,1};
mtrl.ambient.build4tuple(amb);          //填充数组
mtrl.diffuse.build4tuple(diff);
mtrl.specular.build4tuple(spec);
mtrl.emissive.build4tuple(emiss);
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_AMBIENT,amb);
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_DIFFUSE,diff);
if(specularOn)
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_SPECULAR,spec);
else
glMaterialfv(GL_FRONT,GL_SPECULAR,zero);
glMaterialfv(GL_FRONT/* _AND_BACK* /,GL_EMISSION,emiss);
glMaterialf (GL_FRONT/* _AND_BACK* /,GL_SHININESS,mtrl.specularExponent);
}
//Mesh 类的方法
Mesh ::Mesh(){
numVerts=numFaces=numNorms=0;
pt=NULL; norm=NULL; face=NULL;
lastVertUsed=lastNormUsed=lastFaceUsed=-1;
}
void Mesh::freeMesh()
{                                       //释放被该 mesh 占用的内存
delete [] pt;                           //释放整个顶点列表
delete [] norm;
for(int f=0; f<numFaces; f++)
delete[] face[f].vert;                  //删除面的 vert[]数组
delete [] face;
}
int Mesh::isEmpty()
{
return (numVerts==0) || (numFaces==0) || (numNorms==0);
}
```

```
void Mesh::makeEmpty()
{
numVerts=numFaces=numNorms=0;
}
void Mesh::drawOpenGL()
{
tellMaterialsGL(); glPushMatrix();
glMultMatrixf(transf.m);
if(doFaces) drawMesh();
if(doEdges) drawEdges();
glPopMatrix();
}
Mesh::Mesh(string fname){                    //读入该文件,建立 Mesh
numVerts=numFaces=numNorms=0;
pt=NULL; norm =NULL; face=NULL;
lastVertUsed=lastNormUsed=lastFaceUsed=-1;
readMesh(fname);
}
Vector3 Mesh::newell4(int indx[])
{ /* 返回规范化的法线,使其和顶点 pt[indx[0]],…,pt[indx[3]]相对。例如: indx[]中包含四个
指向顶点列表的索引,这些顶点用于计算 Newell。 */
Vector3 m;
for(int i=0; i<4 ; i++)
{
int next=(i==3) ? 0: i+1;                    //哪一个是下一个索引?
int f=indx[i], n=indx[next];                 //表示这一对的名字
m.x += (pt[f].y-pt[n].y) * (pt[f].z+pt[n].z);
m.y += (pt[f].z-pt[n].z) * (pt[f].x+pt[n].x);
m.z += (pt[f].x-pt[n].x) * (pt[f].y+pt[n].y);
}
m.normalize();
return m;
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<readMesh >>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void Mesh:: readMesh(string fname)
{
  fstream inStream;
inStream.open(fname.c_str(), ios::in);  //打开操作需要一个类 C 的字符串
if(inStream.fail() || inStream.eof())
  {
cout <<"can't open file or eof: " <<fname <<endl;
makeEmpty();return;
  }
inStream >>numVerts >>numNorms >>numFaces;
//建立顶点、法线和面的数组
```

```
pt=new Point3[numVerts]; assert(pt !=NULL);
norm=new Vector3[numNorms]; assert(norm !=NULL);
face=new Face[numFaces]; assert(face !=NULL);
for(int i=0; i<numVerts; i++)                    //读入顶点
inStream >>pt[i].x >>pt[i].y >>pt[i].z;
for(int ii=0; ii<numNorms; ii++)                 //读入法线
inStream >>norm[ii].x >>norm[ii].y >>norm[ii].z;
for(int f=0; f<numFaces; f++)                    //读入面数据
{
inStream >>face[f].nVerts;
int n=face[f].nVerts;
face[f].vert=new VertexID[n]; assert(face[f].vert !=NULL);
for(int k=0; k<n; k++)                           //读入表示该面的顶点索引
inStream >>face[f].vert[k].vertIndex;
for(int kk=0; kk<n; kk++)                        //读入表示该面的法线索引
inStream >>face[f].vert[kk].normIndex;
}
inStream.close();
}                                                //readMesh 结束
//<<<<<<<<<<<<<<<<<<<<<<<<drawMesh >>>>>>>>>>>>>>>>>>>>>>
void Mesh :: drawMesh()
{                                                //用 OpenGL 绘制该网格中的每个面：绘制每个多边形
if(isEmpty()) return;                            //网格为空
for(int f=0; f<numFaces; f++)
{
int n=face[f].nVerts;
glBegin(GL_POLYGON);
for(int v=0; v<n; v++)
{
int in=face[f].vert[v].normIndex;
assert(in >=0 && in<numNorms);
glNormal3f(norm[in].x, norm[in].y, norm[in].z);
int iv=face[f].vert[v].vertIndex; assert(iv >=0 && iv<numVerts);
glVertex3f(pt[iv].x, pt[iv].y, pt[iv].z);
}
glEnd();
}
glFlush();
}
//<<<<<<<<<<<<<<<<<<<<<<<<<<write mesh>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
void Mesh:: writeMesh(char * fname)
{                                                //把网格对象写入一个新的具有第 6 章格式的文件当中
if(numVerts ==0 || numNorms ==0 || numFaces ==0) return;     //空
fstream outStream(fname, ios :: out);    //打开输出流
if(outStream.fail()) {cout <<"can't make new file: " <<fname <<endl; return;}
```

```
outStream <<numVerts <<" " <<numNorms <<" " <<numFaces <<"\n";
//写顶点和顶点法线列表
for(int i=0; i<numVerts; i++)
outStream <<pt[i].x <<" " <<pt[i].y <<" " <<pt[i].z <<"\n";
for(int ii=0; ii<numNorms; ii++)
outStream<<norm[ii].x <<" " <<norm[ii].y <<" " <<norm[ii].z <<"\n";
//写面数据
for(int f=0; f<numFaces; f++)
{
int n=face[f].nVerts;
outStream <<n <<"\n";
for(int v=0; v<n; v++)                    //写该面上的顶点索引
outStream <<face[f].vert[v].vertIndex <<" ";outStream <<"\n";
for(int k=0; k<n; k++)                    //写该面的法线索引
outStream <<face[f].vert[k].normIndex <<" "; outStream <<"\n";
}
outStream.close();
}
//Scene 类的方法
//<<<<<<<<<methods >>>>>>>>>>>
string Scene::nexttoken(void)             //##########nexttoken()
{
char c;
string token;
int lastchar=1;
if (!f_in) {return(token); }
if (f_in->eof()) {return(token);}
while (f_in->get(c))
{
if (f_in->eof()) {
return(token);
}
switch (c) {
case '\n': nextline +=1;
case ' ' :
case '\t':
case '\a':
case '\b':
case '\v':
case '\f':
case '\r': {
if ( lastchar ==0 ) {return(token);}break; }
case '{': {
token=c; return(token); break;}
case '}': {
```

```
token=c;
return(token);
break; }
case '!': {
while ( c != '\n' && f_in->get(c)) {
}
nextline++; break;}
default: {
token=token+c;
lastchar=0;

if ((f_in->peek() == '{') ||
(f_in->peek() == '}') ) {
if ( lastchar == 0 ) {
return(token);
} else {
f_in->get(c);
token=c;
return(token);
}
}
line=nextline;
}
}
}
return(" ");
}
//<<<<<<<<getFloat >>>>>>>>>>>>
float Scene::getFloat()                    //############getFloat()
{
strstream tmp;
float number;
string str=nexttoken();
tmp <<str;
if(!(tmp >>number))
{
cerr <<"Line " <<line <<": error getting float" <<endl;
exit(-1);
}
else
{
char t;
if ( (tmp >>t ) )
{
cerr <<"Line " <<line <<": bum chars in number" <<endl;
```

```
exit(-1);
}
}
return number;
}
//<<<<<<<<<isidentifier >>>>>>>>>>>
bool Scene::isidentifier(string keyword) {//########isidentifier
string temp=keyword;
if (!isalpha(temp[0])) return(false);
for (int count=1; count<temp.length(); count++) {
if ((!isalnum(temp[count]))&& (temp[count]!='.')) return(false);
}
return(true);
}
//<<<<<<<<<cleanUp >>>>>>>>>>>
void Scene::cleanUp()                    //#########cleanUp
{                                        //分析文件后释放材料
affStk.releaseAffines();                 //删除仿射栈
def_stack->release();
delete def_stack;                        //释放 DefUnitStack 内存
}
//<<<<<<<<<freeScene >>>>>>>>>>>
void Scene::freeScene()
{                                        //释放物体和光列表
GeomObj * p=obj;
while(p)
{
GeomObj * q=p;
p=p->next;
delete q;
}
Light * q=light;
while(q)
{
Light * r=q;
q=q->next;
delete r;
}
}
//<<<<<<<<<<<<<<whichtoken >>>>>>>>>>>>>>
TokenType Scene::whichtoken(string keyword)
{
string temp=keyword;
if ( temp =="light" )            return (LIGHT);
if ( temp =="rotate" )           return (ROTATE);
```

```
if ( temp =="translate" )          return (TRANSLATE);
if ( temp =="scale")               return (SCALE);
if ( temp =="push")            return (PUSH);
if ( temp =="pop"  return (POP);
if ( temp =="identityAffine")     return (IDENTITYAFFINE);
if ( temp =="cube")               return (CUBE);
if ( temp =="sphere")             return (SPHERE);
if ( temp =="torus")              return (TORUS);
if ( temp =="plane")              return (PLANE);
if ( temp =="square")             return (SQUARE);
if ( temp =="cylinder")           return (CYLINDER);
if ( temp =="taperedCylinder")    return (TAPEREDCYLINDER);
if ( temp =="cone")               return (CONE);
if ( temp =="tetrahedron")        return (TETRAHEDRON);
if ( temp =="octahedron")         return (OCTAHEDRON);
if ( temp =="dodecahedron")       return (DODECAHEDRON);
if ( temp =="icosahedron")        return (ICOSAHEDRON);
if ( temp =="buckyball")          return (BUCKYBALL);
if ( temp =="diamond")            return (DIAMOND);
if ( temp =="teapot")   return (TEAPOT);
if ( temp =="union")              return (UNION);
if ( temp =="intersection")       return (INTERSECTION);
if ( temp =="difference")         return (DIFFERENCEa);
if ( temp =="mesh")               return (MESH);
if ( temp =="makePixmap")         return (MAKEPIXMAP);
if ( temp =="defaultMaterials")   return (DEFAULTMATERIALS);
if ( temp =="ambient")            return (AMBIENT);
if ( temp =="diffuse")            return (DIFFUSE);
if ( temp =="specular")           return (SPECULAR);
if ( temp =="specularFraction")   return (SPECULARFRACTION);
if ( temp =="surfaceRoughness")   return (SURFACEROUGHNESS);
if ( temp =="emissive")           return (EMISSIVE);
if ( temp =="specularExponent")   return (SPECULAREXPONENT);
if ( temp =="speedOfLight")       return (SPEEDOFLIGHT);
if ( temp =="transparency")       return (TRANSPARENCY);
if ( temp =="reflectivity")       return (REFLECTIVITY);
if ( temp =="parameters")         return (PARAMETERS);
if ( temp =="texture")  return (TEXTURE);
if ( temp =="globalAmbient")  return (GLOBALAMBIENT);
if ( temp =="minReflectivity")  return (MINREFLECTIVITY);
if ( temp =="minTransparency")  return (MINTRANSPARENCY);
if ( temp =="maxRecursionDepth") return (MAXRECURSIONDEPTH);
if ( temp =="background")         return (BACKGROUND);
if ( temp =="{")                  return (LFTCURLY);
if ( temp =="}")                  return (RGHTCURLY);
```

```
if ( temp =="def")                    return (DEF);
if ( temp =="use")                    return (USE);
if ( temp =="  " )                    return (T_NULL);
if ( isidentifier(temp) )             return (IDENT);
cout <<temp <<":" <<temp.length() <<endl;
return(UNKNOWN);
}                                          //whichtoken 结束
//<<<<<<<<<<<<<<drawSceneOpenGL>>>>>>>>>>>>>>>>>>>>
void Scene::drawSceneOpenGL()
{//绘制物体列表中的每一个物体
for(GeomObj * p=obj; p; p=p->next)
p->drawOpenGL();                           //绘制它
}
//<<<<<<<<<<<<<<<<<Scene :: read>>>>>>>>>>>>>>>>>>
bool Scene:: read(string fname)            //如果成功返回 true,否则返回 false
{
file_in=new ifstream(fname.c_str());
if(!(* file_in))
{
cout <<"I can't find or open file: " <<fname <<endl;
return false;
}
f_in=new strstream();
line=nextline=1;
def_stack=new DefUnitStack();
char ch;
freeScene();                               //删除所有以前的场景

//初始化所有用于读的数据
obj=tail=NULL;
light=NULL;
affStk.tos=new AffineNode;
affStk.tos->next=NULL;

while (file_in->get(ch)) {* f_in <<ch;}  //读整个文件
while(1)                                   //读文件,收集物体直到文件结束或者发生错误
{
GeomObj * shp=getObject();                 //获取下一个形状
if(!shp) break;                            //没有物体:错误或者文件结束
shp->next=NULL;                            //为了安全
if(obj ==NULL){ obj=tail=shp;}             //目前是空列表
else{tail->next=shp; tail=shp;}            //给队列中加入新物体
}
file_in->close();
cleanUp();                                 //删除临时列表等
```

```
return true;
}                                          //read()结束
//<<<<<<<<<<<<<<<Scene :: getObject >>>>>>>>>>>>>>>>
GeomObj * Scene :: getObject()
{                                          //从 f_in 流中读入标记,(Scene 类的数据成员),
//建立光源,获取材质,进行变换
//直到找到新的物体
//如果发生任何错误或文件技术则返回 NULL
string s;
GeomObj * newShape;
TokenType typ;
while ((typ=(whichtoken( s=nexttoken() ))) !=T_NULL)
{
if(typ ==UNION || typ ==INTERSECTION || typ ==DIFFERENCEa)
{
switch(typ)
{
case UNION:newShape=new UnionBool();break;
case INTERSECTION:newShape=new IntersectionBool();break;
case DIFFERENCEa:newShape=new DifferenceBool();break;
}                                          //小 switch 结束
GeomObj * p=newShape;
p=getObject();                             //获取左孩子
if(!p) return NULL;                        //错误! 应该总是得到一个物体
((Boolean * )newShape)->left =p;           //把它链接起来
p=getObject();//获取右孩子
if(!p) return NULL;
((Boolean * )newShape)->right=p;           //把它链接起来
return newShape;
}//if(typ ==UNION etc….结束
switch(typ)
{
case LIGHT: {
Point3 p;
Color3 c;
p.x=getFloat(); p.y=getFloat();p.z=getFloat();
c.red=getFloat(); c.green=getFloat();c.blue=getFloat();
Light * l=new Light;
l->setPosition(p);
l->setColor(c);
l->next=light;                             //把它放入列表
light=l; break;}
case ROTATE: {
float angle;
Vector3 u;
```

```
angle=getFloat(); u.x=getFloat();
u.y=getFloat(); u.z=getFloat();
affStk.rotate(angle,u);break;}
case TRANSLATE: {
Vector3 d;
d.x=getFloat(); d.y=getFloat(); d.z=getFloat();
affStk.translate(d);break;}
case SCALE: {
float sx, sy, sz;
sx=getFloat(); sy=getFloat(); sz=getFloat();
affStk.scale(sx, sy, sz);break;}
case PUSH: affStk.dup(); break;
case POP:  affStk.popAndDrop(); break;
case IDENTITYAFFINE: affStk.setIdentity();break;
case AMBIENT: {
float dr, dg, db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.ambient.set(dr,dg,db); break;}
case DIFFUSE: {
float dr,dg,db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.diffuse.set(dr,dg,db); break;}
case SPECULAR:{
float dr,dg,db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.specular.set(dr,dg,db); break;}
case EMISSIVE: {
float dr,dg,db;
dr=getFloat(); dg=getFloat(); db=getFloat();
currMtrl.emissive.set(dr,dg,db); break;}
case PARAMETERS: {                          //获取 numParams 参数列表
currMtrl.numParams=(int)getFloat();
for(int i=0; i<currMtrl.numParams; i++)
currMtrl.params[i]=getFloat();
break;}
case SPECULARFRACTION: currMtrl.specularFraction=getFloat(); break;
case SURFACEROUGHNESS: currMtrl.surfaceRoughness=getFloat(); break;
case TEXTURE: {                             //获取类型,0 表示无
currMtrl.textureType=getFloat();}
break;
case DEFAULTMATERIALS: currMtrl.setDefault();break;
case SPEEDOFLIGHT: currMtrl.speedOfLight=getFloat(); break;
case SPECULAREXPONENT: currMtrl.specularExponent=getFloat(); break;
case TRANSPARENCY:currMtrl.transparency=getFloat(); break;
case REFLECTIVITY: currMtrl.reflectivity=getFloat(); break;
```

```
case GLOBALAMBIENT:
ambient.red=getFloat(); ambient.green=getFloat();
ambient.blue=getFloat(); break;
case BACKGROUND:
background.red=getFloat();
background.green=getFloat();
background.blue=getFloat();break;
case MINREFLECTIVITY: minReflectivity=getFloat(); break;
case MINTRANSPARENCY:minTransparency=getFloat(); break;
case MAXRECURSIONDEPTH: maxRecursionDepth=getFloat(); break;
case MAKEPIXMAP: {                          //为像素映射获取 BMP 文件名
  int which=getFloat();                     //像素映射数组中该像素映射的索引
  if(which<0 || which >7){cout <<"\nbad index of RGBpixmap! \n";}
  string fname=nexttoken();                 //获取网格文件名
  cout <<"I got fname=" <<fname <<endl;
  if(!pixmap[which].readBMPFile(fname))
  {//读入 BMP 文件到该像素映射
cout <<" \ncan't read that RGBpixmap file! \n";
return NULL;
  }
  break;}//case: MAKEPIXMAP 结束
case T_NULL: break;                         //空标记代表文件结束
case DEF: {
string name, temp, lb, rb;
int l=line;
string inp;
name=nexttoken();
if ( whichtoken(name) !=IDENT ) {
cout <<"Error:  Identifier expected." <<endl;
return NULL;
}
if ( def_stack->search(name) ) {
cout <<line <<": " <<name;
cout <<": attempt to redefine. " <<endl;
return NULL;
}
lb=nexttoken();
if ( whichtoken(lb) !=LFTCURLY ) {
cout <<"Error: { expected." <<endl;
return NULL;
}
while ( whichtoken( temp=nexttoken()) !=RGHTCURLY ) {
cout <<temp <<endl;
inp=inp+temp+" ";
if (!f_in) {
```

```
cout <<"Error: end of file detected." <<endl;
return NULL;
}
}
//字符串内容压栈
def_stack->push(name, inp);
break;}                                    //end of case: DEF
case USE: {
string name;
name=nexttoken();
if ( whichtoken(name) !=IDENT ) {
cout <<line <<": " <<name;
cout <<": identifier expected.";
return NULL;
}
if (!def_stack->search(name) ) {
cout <<line <<": " <<name;
cout <<": not defined.";
return NULL;
}
cout <<def_stack->contents(name) <<endl;
strstream * temp_fin=new strstream;
* temp_fin <<def_stack->contents(name) <<" ";
* temp_fin <<f_in->rdbuf();
delete (f_in);
f_in=temp_fin;
break; }                                   //case: USE 结束
default:  {                                //Shapes 的内 switch
switch(typ)
{
case CUBE:newShape=new Cube;break;
case SPHERE:newShape=new Sphere;break;
case TETRAHEDRON:newShape=new Mesh("tetra.3vn");break;
case TORUS:newShape=new Torus;break;
case PLANE:newShape=new Plane;break;
case SQUARE:newShape=new Square;break;
case TAPEREDCYLINDER:newShape=new TaperedCylinder;
((TaperedCylinder * )newShape)->smallRadius=getFloat(); break;
case CONE:newShape=new TaperedCylinder;
((TaperedCylinder * )newShape)->smallRadius=0; break;
case CYLINDER:newShape=new TaperedCylinder;
((TaperedCylinder * )newShape)->smallRadius=1; break;
case OCTAHEDRON:newShape=new Mesh("octa.3vn");break;
case DODECAHEDRON:newShape=new Mesh("dodeca.3vn"); break;
case ICOSAHEDRON:newShape=new Mesh("icosa.3vn"); break;
```

```
case BUCKYBALL:newShape=new Mesh("bucky.3vn");break;
case DIAMOND:newShape=new Mesh("diamond.3vn"); break;
case TEAPOT:newShape=new Teapot; break;
case MESH: {                                    //获取该网格的文件名(带扩展名)
string fname=nexttoken();                       //获取网格文件名
newShape=new Mesh(fname); break;
}//case: MESH结束
   default: {
cerr <<"Line " <<nextline <<": unknown keyword " <<s <<endl;
return NULL;
}
}                                               //内 switch结束
//对所有形状的常规操作
((Shape * )newShape)->mtrl.set(currMtrl);
//加载变换及其逆变换
((Shape * )newShape)->transf.set(* (affStk.tos->affn));
((Shape * )newShape)->invTransf.set(* (affStk.tos->invAffn));
return newShape;
}//default: block结束
    }                                           //外 switch结束
  }                                             //while结束
  return NULL;
}                                               //getObject结束
//<<<<<<<<<<<<<<<<<<<<<<<<<DefUnitStack方法>>>>>>>>>>>>>>>>>>>>>>>>>>
void DefUnitStack::push(string n, string s) {
D4S * temp_d4s=new D4S;
temp_d4s->current=new DefUnit(n, s);
temp_d4s->next=stack;
stack=temp_d4s;
}
void DefUnitStack::print() {
D4S * temp=stack;
string t;
while (temp) {
cout <<temp->current->name <<":" ;
cout <<temp->current->stuff <<endl;
temp=temp->next;
}
}
int DefUnitStack::search(string s) {
D4S * temp=stack;
while (temp) {
if ( temp->current->name ==s ) {
return(1);
}
```

```
temp=temp->next;
}
return(0);
}
string DefUnitStack::contents(string s) {
D4S * temp=stack;
while (temp) {
if (temp->current->name ==s ) {
return(temp->current->stuff);
}
temp=temp->next;
}
return(NULL);
}
void DefUnitStack::release()
{
while(stack)
{
D4S *  tmp=stack;                          //获取它
//cerr <<"releasing def_stack item: "<<tmp->current->name<<endl;
stack=stack->next;                         //增加指针
delete tmp->current;                       //释放两个字符串
delete tmp;                                //释放节点
}
stack=NULL;
}
//SDL.cpp 结束
```

噪音类

```
//Noise.h
//用来产生基于按 Peachey/Perlin 方式噪声框架的伪随机噪声场的 Noise 类
#include <assert.h>

class Noise{
public:
Noise()                                    //构造噪声对象
{
int i;
index=new unsigned char[256]; assert(index);
for(i=0; i<256; i++) index[i]=i;           //用索引填充数组
for(i=0; i<256; i++)                       //shuffle it①
{
```

① 指像洗牌一样随机的产生噪声。

```
int which=rand() %256;                    //在数组中选择随机的位置
unsigned char tmp=index[which];           //交换它们
index[which]=index[i];
index[i]=tmp;
}
noiseTable=new float[256]; assert(noiseTable);
for(i=0; i<256; i++) noiseTable[i]=rand()/32767.99;
}                                         //构造函数结束
float noise(float scale, Point3& p)
{                                         //线性插值的栅格噪声
#define lerp(f, A, B) A+f* (B-A)
float d[2][2][2];
Point3 pp;
pp.x=p.x* scale+10000;                    //偏移量避免负值
pp.y=p.y* scale+10000;
pp.z=p.z* scale+10000;
long ix=(long)pp.x; long iy=(long)pp.y; long iz=(long)pp.z;
float tx,ty,tz, x0,x1,x2,x3, y0,y1;
tx=pp.x-ix; ty=pp.y-iy; tz=pp.z-iz;  //小数部分
float mtx=1.0-tx, mty=1.0-ty, mtz=1.0-tz;

for(int k=0; k<=1; k++)                   //获取 8 个栅格点的噪声
for(int j=0; j<=1; j++)
for(int i=0; i<=1; i++)
d[k][j][i]=latticeNoise(ix+i, iy+j,iz+k);

x0=lerp(tx, d[0][0][0],d[0][0][1]);
x1=lerp(tx, d[0][1][0],d[0][1][1]);
x2=lerp(tx, d[1][0][0],d[1][0][1]);
x3=lerp(tx, d[1][1][0],d[1][1][1]);
y0=lerp(ty, x0, x1);
y1=lerp(ty, x2, x3);
return lerp(tz, y0, y1);
}
float turbulence(float s, Point3& p)
{
float val=noise(s, p)/2 +
noise(s* 2, p)/4 +
noise(s* 4, p)/8 +
noise(s* 8, p)/16;
return val;
}
float marble(float strength,Point3& p)
{
float turbul=turbulence(10, p);
```

```
float val=sin(6*p.z+strength*turbul);
return mySpline(val);
}
float
private:
float* noiseTable;                    //噪声值数组
unsigned char* index;                 //伪随机索引
float mySpline(float x)               //用于大理石
{
if(x<-0.4) return 0.15+2.857*SQR(x+0.75);
else if(x<0.4) return 0.95-2.8125*SQR(x);
else return 0.26+2.666*SQR(x-0.7);
}
float latticeNoise(int i, int j, int k)
{                                     //返回整数格点的 PR 噪声值
#define PERM(x) index[(x) & 255]
#define INDEX(ix, iy, iz) PERM( (ix)+PERM((iy)+PERM(iz)) )
return noiseTable[INDEX(i,j,k)];
}
};                                    //噪声类结束
```

A3.1.3 一些用于光线跟踪的类

```
//@@@@@@@@@@@@@@@@PointCluster 类 @@@@@@@@@@@@@@@@
class PointCluster{
public:                               //保存一个形状轮廓点的数组
int num;
Point3* pt;
PointCluster() {num=0; pt=NULL;}
PointCluster(int n)                   //建立 n 个点的串
{
pt=new Point3[n]; assert(pt);
num=n;
  }
};
//@@@@@@@@@@@@@@@@SphereInfo @@@@@@@@@@@@@@@@
class SphereInfo{                     //保存球的中心和半径
public:
Point3 center;
float radSq;
void set(float x, float y, float z, float rsq)
{
center.set(x,y,z);
radSq=rsq;
```

```
}
};
//@@@@@@@@@@@@@@@@@@@@Cuboid @@@@@@@@@@@@@@@@@@@
class Cuboid{                                   //保存立方体的 6 个边界值
public:
float left, top, right, bott, front, back;
void set(float l, float t, float r, float b, float f, float bk)
{
left=l; top=t; right=r; bott=b; front=f; back=bk;
}
void set(Cuboid& c)
{
left=c.left;top=c.top; right=c.right; bott=c.bott;
front=c.front; back=c.back;
}
};

//@@@@@@@@@@@@@@@@@@@Ray @@@@@@@@@@@@@@@@@@@
class Ray{
public:
Point3 start;
Vector3 dir;
int recurseLevel;
int row, col;                                   //用于表示屏幕范围
int numInside;                                  //列表中物体的数目
GeomObj * inside[10];                           //物体指针数组
Ray(){start.set(0,0,0); dir.set(0,0,0); numInside=0;}
Ray(Point3 origin);                             //构造函数：设置光线的起点
Ray(Point3& origin, Vector3& direction)
{ start.set(origin); dir.set(direction); numInside=0;}
void setStart(Point3& p){start.set(p);}
void setDir(float x, float y, float z)
{dir.x=x; dir.y=y; dir.z=z;}
void setRayDirection(Light * L);                //用于阴影触角
void setRayDirection(Vector3& dir);             //用于产生的光线
int isInShadow();
void makeGenericRay(GeomObj * p, Ray& gr);
};
//@@@@@@@@@@@@@@@@@@HitInfo @@@@@@@@@@@@@@@@@@
class HitInfo {                                 //一个表面的每次命中的数据
  public:
double hitTime;                                 //命中时间
GeomObj * hitObject;                            //命中的物体
int surface;                                    //那个面被命中
int isEntering;                                 //光线进入物体了吗？
```

```
Point3 hitPoint;                        //命中点
Vector3 hitNormal;                      //命中点处的法线
HitInfo()
{
hitObject=NULL; hitTime=-1000; surface=0; isEntering=0;
}
void set(HitInfo& h)
{
hitTime=h.hitTime; hitObject=h.hitObject;
surface=h.surface; isEntering=h.isEntering;
hitPoint.set(h.hitPoint); hitNormal.set(h.hitNormal);
}
};

//@@@@@@@@@@@@@@@@Intersection @@@@@@@@@@@@@@@@
class Intersection{                     //保存命中列表
public:
#define maxNumHits 8
int numHits;                            //命中次数
HitInfo hit[maxNumHits];                //命中列表;
Intersection(){numHits=0;}              //默认构造函数
void set(Intersection& intr)
{                                       //拷贝相交信息
numHits=intr.numHits;
for(int i=0; i<maxNumHits; i++)
hit[i].set(intr.hit[i]);
}
};
```

A3.1.4 SDL：场景描述语言

场景描述语言是一种用清晰通俗的方式描述几何物体和形状的非常简单的语言。Scene 类中的 read()方法可以处理这种语言。我们首先简单介绍一下 Scene 类的基础，然后描述 SDL 并讨论其用法。

要读取一个用 SDL 写的场景文件(例如 mySence. dat)，我们首先得定义一个全局的场景对象(例如 scn)，然后对它调用 read()方法。将某个待处理的 SDL 文件名给该方法：

```
Scene scn;                      // 建立一个场景对象
scn.read("myScene.dat");        //读 SDL 文件,建立场景
```

读入并解释 myScene. dat 文件，并建立物体列表。同时创建光源列表。这些列表可以分别通过 scn. obj 和 scn. light 域得到。第 5 章的 drawOpenGL()方法在用 OpenGL 程序渲染这些场景的时候用到这些列表。第 12 章介绍的 shade()也可以用这些列表通过光线跟踪渲染场景。

A3.1.5 Scene 类

Scene 类中的一个对象有多个域来描述场景的特性。主要的元素是一个在场景中的几何形状的列表。obj 域是一个指向列表中第一个元素的指针。为了绘制列表中的所有物体，我们只需简单遍历整个列表，并告知每个物体绘制自己。

```
for(GeomObj * p=scn.obj; p!=NULL; p=p->next)
p->drawOpenGL();
```

列表中的物体属于 Shape 中的一种或者另一种类型(球，立方体，十二面体，等等)，而且每一种 Shape 类型都知道如何绘制它自己。这里用到了多态：所有的 Shape 类型都继承自基本类型 GeomObj(几何物体"geometric object"的缩写)，所以任何 Shape 类型都能放在 GeomObj 指针的列表中。

```
Light * light;                  //光源列表
GeomObj * obj;                  //物体列表
Color3 background;              //背景颜色
Color3 ambient;                 //全局环境色
```

另外还有 3 个域用于光线跟踪(见第 12 章)：maxRecursionDepth、minShinyness 和 minTransparency。

A3.2 SDL 语法

SDL 区分大小写，但形式灵活：多个泛空格符(空格、Tab、换行、换页等)等于一个空格。注释以"!"开头持续到行尾。SDL 中的关键字用来指定不同的仿射变换、几何物体、光源和场景属性(如背景颜色)。

创建几何对象

简单的说明物体的类型就可以创建物体并将其放入物体列表中。例如，cube 添加一个立方体对象到物体列表中，而 sphere 添加一个球体对象。

其他几何对象包括(参考 SDL.h 获得完全列表) torus、plane、square、cylinder、cone、tetrahedron、dodecahedron、icosahedron、buckyball、diamond 和 teapot。当这些对象类型中的任何一个在文件中被指定的时候，相应的物体就被添加到物体列表(尾部)了。

有些几何物体需要一个或多个参数。参数是浮点数或者文件名。参数被直接放在物体名后面，例如：

```
taperedCylinder .312 !生成小端半径为 0.312 的圆台
mesh pawn.3vn !生成网格
```

第一个例子创建一个用参数定义其确切形状的圆台对象；第二个例子创建了一个顶点和面列表在文件 pawn.3vn 中描述的网格对象(本书的网站上有许多的 3vn 格式样例文件，

包括十二面体、菱形和 buckyball 等样例)。

管理仿射变换

每个物体被创建的时候都会存储一个仿射变换。(该变换的逆也同时被存储;逆变换在光线跟踪时有用。)和物体一起创建的那个变换是当前变换(CT),既当时生效的变换。SDL 文件中的各种关键字都修改 CT。例如:

```
identityAffine
```

将恒等变换(由一个 4×4 的单位针给出)放在 CT 中。这是当 read()开始解析 SDL 文件时的初始变换。

SDL 使用词语 scale、rotate 和 translate 以及每个词后面跟的一个合适的参数来改变 CT。这同 OpenGL 使用 glScalef()、glRatef()和 glTranslatef()改变模型视图或投影矩阵是相似的。

特别地,每个词用相应的变换右乘以 CT 并把乘积放回 CT,如:

```
CT=CT * Trans
```

这里的 Trans 是一个表示新变换的 4×4 矩阵。

三个动词如下:

```
scale <sx><sy><sz>
rotate <ang><ux><uy><uz>
translate <dx><dy><dz>
```

scale 有 3 个分别表示 x、y 和 z 方向上缩放比例的浮点型参数。rotate 有 4 个参数:旋转角度(以度为单位)和旋转轴在 x、y 及 z 方向上的分量。(从 u 向原点望去,ang 的正值产生关于 u 轴的 CCW 旋转。)translate 有 3 个参数:变换所用向量的 x、y 和 z 分量。

例如,命令

```
scale 2 1.3 -5.33
translate 4 -5 6
rotate 45 0 1 0
```

分别产生如下矩阵 $Sc=\begin{bmatrix}2&0&0&0\\0&1.3&0&0\\0&0&-5.33&0\\0&0&0&1\end{bmatrix}$, $Tr\begin{bmatrix}1&0&0&4\\0&1&0&-5\\0&0&1&6\\0&0&0&1\end{bmatrix}$ 和 $Rot\begin{bmatrix}0.707&0&0.707&0\\0&1&0&0\\-0.707&0&0.707&0\\0&0&0&1\end{bmatrix}$,每个命令把其对应的矩阵右乘以 CT 并把结果放回 CT。下面有一个完整的例子。

仿射变换栈

CT 实际上是矩阵栈的栈顶矩阵。利用字 push 和 pop 来操作该栈:

- push 创建 CT 的一个副本并将其压入栈中(使栈顶的两个元素相同)。
- pop 从栈中弹出 CT 并丢弃它。刚好在栈顶元素下面的矩阵变成 CT。如果栈顶元素下面没有元素,pop 不做任何操作。

管理材料属性

在创建每个物体的同时创建的材料属性记录,称为当前材料(CM)。记录包含如下域:

- 四个颜色域,每个域都包含(r,g,b)三元组:ambient、diffuse、specular 和 emissive。
- 六个标量域(用于光线跟踪):specularExponent、specular Fraction、surfaceRoughness、speedOfLight、transparency 和 reflectivity。
- 描述物体纹理性质的三个域:textureType 指明哪种纹理被用于物体,numParams 为物体定义的参数数目,params[]是一个有 10 个变量值的数组。textureType 编码为 0 代表空,-1 代表棋盘,-2 代表木材,-3 代表大理石。代码 makePixmap 2 filename.bmp 读入一个 BMP 文件并把它赋给纹理 2。2 这个 textureType 后面跟着一个图像文件名,它建立了该图的一个像素映射。

对 CM 的改变可以通过下面的 SDL 字控制,每个后面跟了一个或多个浮点数:

```
ambient <r><g><b>
diffuse <r><g><b>
specular <r><g><b>
emissive <r><g><b>
specularExponent <value>
specularFraction <value>
surfaceRoughness <value>
speedOfLight <value>
transparency <value>
reflectivity <value>
textureType <value>
parameters <value><value><value>p
```

关键字 parameters 后面跟着一个指定参数个数的数字和参数值列表。例如,要把 4.5、6 和-12 放进随后定义的物体的 params[]数组中,可以用

```
parameters 3 5 4.5 6 -12
```

CM 中最初包含了如下默认值:

```
ambient=<0.1><0.1><0.1>
diffuse=<0.8><0.8><0.8>
specular=<0><0><0>
emissive=<0><0><0>
specularExponent 1
specularFraction 0
surfaceRoughness 1.0
speedOfLight=1
transparency=0
```

```
reflectivity=0
textureType 0
```

关键字 defaultMaterials 可以被用来恢复 CM 为这些默认值。以下是和光源、全局属性、布尔对象和像素映射有关的关键字：

光源

```
light <x><y><z><r><g><b>!在<x, y, z>处放置一个颜色为<r, g, b>的光源
```

指定全局场景属性

```
globalAmbient <r><g><b>!指定全局泛光颜色为<r, g, b>
minReflectivity <value>
minTransparency <value>
maxRecursionDepth <value>
background <r><g><b>
```

下面是一个简单的 SDL 文件：

```
!myScene1.dat -f.s.hill
!有几个简单的发光物体
globalAmbient .4 .2 .3
light 0 10 0 1 1 1 !在(0,10,0)处的白光
background 0 0 .5
ambient .2 .2 .2
diffuse .8 .7 .6
emissive .8 0 0 !物体放出红色
cube !在原点处放置一个普通的立方体
emissive 0 1 0
!在 (2,0,0)处放置一个发光的椭圆
push translate 2 0 0 rotate 45 0 0 1 scale .5 .5 2 sphere pop
push translate -2 0 0 cone pop !也 (放置)一个圆锥在 (-2,0,0)
```

布尔对象

SDL 也支持指定布尔对象，使用的关键字为：

```
union
intersection
difference
```

这些词中的每个都可以创建一个指定类型的布尔对象并将它放在对象列表中。在 SDL 文件中，这样的每个对象都必须跟有两个对象(其中的每个可能是几何或者布尔对象)，它们被放在最初创建的对象的左孩子和右孩子的位置。下面的 SDL 代码描述了一个包含两个布尔对象的场景。第一个对象是由一个四面体和巴克球进行并运算然后再和一个立方体交得到的。

```
light 1 2 3 0.2 0.3 0.4
intersection
cube !交叉点的左孩子
union !交叉点的右孩子
diffuse .2 .5 .7 tetrahedronn !并节点的左孩子
rotate 180 3 4 5 buckyball !并节点的右孩子
difference !另一个布尔对象
tetrahedron intersection !差节点的左孩子
plane union !交叉节点的右孩子
dodecahedron torus !并节点的左孩子
```

动词

```
makePixmap <value><fname>
```

定义了一个用于纹理绘制的像素映射。第一个参数是用来识别像素图的整型。第二个是包含图像纹理的 BMP 文件名。例如，如下的 SDL 代码：

```
makePixmap 5 clouds.bmp
makePixmap 5 stone.bmp
texture 5
parameters 4 0 1 1 0
push scale 3 4 2 cube pop
texture 2
parameters 3 .2 .8 2
sphere
```

创建了两个纹理。然后它把 5 号(跟着 4 个参数)纹理和一个缩放了的立方体关联，2 号(跟着 3 个参数)和一个球关联。

A3.3 SDL 中的宏

为了方便，关键字 def 允许你把许多 SDL 命令组合到一个宏中，并给其一个名字。组成宏的 SDL 命令被括号包起来。例如：

```
def red{ambient 1 0 0 diffuse 1 0 0}
```

把名为 red 的宏和上述语句联系在一起。如果在 SDL 文件后面的部分用到了命令

```
use red
```

那么语句 ambient 1 0 0 diffuse 1 0 0 就会被放在该处，就像在文件的这个地方键入一样。宏可以保存键入语句并允许定义被重用。例如，一个"L"形栈可以用如下代码定义：

```
def Lstack
{
push cube translate 2 0 0 cube
```

```
translate -2 2 0 cube
translate 0 2 0 cube pop
}
```

以后这些宏可以被用来为场景放入多个不同的 L 形,实现这个功能的代码如下:

```
use Lstack
push translate 3 2 4 scale .5 .5 .5 use Lstack pop
push translate -3 -4 -4 scale .5 .5 .5 use Lstack pop
```

A3.4 扩展 SDL

可以直接给 SDL 添加关键字。我们通过两个例子予以说明如何实现这个功能。

示例 A3.4.1 给场景添加一个属性

如果你想给 Scene 类添加新的域 fogThickness,来描述场景中雾的薄厚。为了控制放入该域中的值,需要在 SDL 语言中添加关键词,例如 fogginess。SDL 文件中的语句 fogginess 0.5 就可以把 fogThickness 的值变为 0.5。为了完成以上内容,需要进行如下变动。

sdl.h 中的改动

1. 给 Scene 类添加 float fogThickness 域。
2. 将 FOGTHICKNESS 添加到 TokenType 枚举表中的任何位置。

sdl.cpp 中的改动

1. 在函数 whichtoken 中加入一行: if (temp == "fogginess") return (FOGTHICKNESS)。

2. 在函数 getObject 中添加一行: case FOGTHICKNESS: fogThickness = getFloat (); break。

当然,如何用这个新的域的问题得由程序员自己去研究了。例如,在开发光线跟踪器时,程序员可能给 Scene: : Shade( )中添加一些代码,如: if(fogThickness > 0) do something。

示例 A3.4.2 给场景添加一个属性

如果你想在可能出现在场景的物体集合中加入扇形饼。这个扇形饼将是位于 xy 平面一个薄圆盘的一部分。扇形由 0 度角开始(沿着 x 轴),连续逆时针(从(0, 0, 1)向原点看)旋转到以角度来度量的角 sweep。因此,如果 sweep 是 180,扇形将是一个处于正 y 轴空间的半圆,而如果 sweep 是 360 度,该扇形将是一个完整的圆。扩展 SDL 语言,使得它能够识别带有一个和 sweep 角相关的参数的 pieSlice 关键字,如

```
pieSlice 90
```

文件 sdl.h 中的改动

1. 定义个含有保存 sweep 角域的类 PieSlice，通过下述代码实现：

```
class PieSlice : public Shape {
public:
float sweep;
etc.
}
```

2. 为该类编写适当的方法，如 drawOpenGL()。
3. 在 TokenType 枚举列表中的任意位置加入 PIESLICE 项。

文件 sdl.cpp 中的改动

1. 在函数 whichtoken()中添加一行

```
if (temp==="pieSlice") return (PIESLICE);
```

2. 在函数 getObject()中添加如下行

```
case PIESLICE;
newShape=new PieSlice;
((PieSlice*) newShape)->angle=getFloat(); break;
```

附录 4

分形和曼德布洛特集合

A4.1 引　　言

计算机特别擅长做重复性工作：它们可以毫无抱怨地不断重复做一件事情。另外，现代计算机在计算方面有很高的精确度，从而产生一些算法，能够有效地把客观物体放大到很大程度，可以更近距离的观察。

在这个附录中，我们从很多方面更进一步扩展，发掘计算机图形学的潜力，展现计算机图形学可能遇到的一些问题。针对无穷的范围，检测三种方法：无穷小，放大到更深层的细节或者可以给一个图像添加更细微的细节；无穷大，检测的图案可以通过某些方法重新产生出更大的图案；无穷多次，研究一个进程理论上无限重复时会发生什么。

通过前面的章节，我们可以看出计算机图形学如何产生自然界中不存在或者曾经存在的事物的图片。我们现在研究的这些内容确实是这样的，而计算机图形学提供了很强大的工具去研究这些内容。这里更进一步，我们突破任何计算机产生图片的固有限制：存在一个有限的解法，有限的大小，可以在有限的时间内完成。因此，我们所作的图片仅仅是被研究生物的近似：图片的观察者把它作为基本的真实物体的样子的一种暗示。

A4.2 分形和自相似

我们想找到一些方法，使得我们接近无穷——更精细的无穷小——在某种组织方式下。这些方法是某种特征的递归，现代计算机能够很有效地做好这一点。递归通常会使得很复杂的集合问题变得很简单。在其他事情中，它使得一个分解或者精细的轮廓更加细小，达到理论上的无限。递归算法构造的形状不仅美妙迷人，在科学和工程中也有很大的应用。

在这里我们描述的很多曲线和图片，都有一个尤其重要的属性：它们是自相似的。直观上理解，在任何尺度下，他们表现形式都是相同的：不管把曲线的图片放大多少倍，它仍然具有相同层次的细节。一些曲线是**严格自相似**的，由此可以得出，如果一个区域被放大很多倍后，放大后的图和原图仍然是很像的(除了可能的旋转或者移动)。剩下的是统计意义下**自相似**的，曲线中的摆动和无规则在平均意义上是一样的，不管图片被放大多少倍。

自然界提供了一些模仿统计意义下自相似的例子。很典型的例子就是海岸线。从卫星上看，由于海湾、水湾和半岛等，海岸线有某种意义上险峻的层次。当一个人飞近一点看的时候，很多细节就显现出来了。一个海湾呈现某种险峻性是以前所看不到的。进一步放大，海滩上单个的大石头和波动，以类似险峻性的形式呈现在眼前。再进一步放大，小石头和小

鹅卵石看起来产生了同样的险峻性。继续这个过程，直到通过显微镜看到沙子的单个的颗粒。其他自然现象，也同样呈现了自相似的特性，比如树的枝条，海绵的表面，人行道上的裂缝，动物的血管系统等。云也具有大概的自相似，并且提供了更有趣的例子。当乘飞机飞行的时候，很难判断一个云有多大：它是小而近呢，还是大而远呢？

在20世纪70年代，耶鲁大学的本瑟姆·曼德布洛特(Benoit Mandelbrot)（那个时候他在IBM研究中心）一起广泛地研究自相似的性质(比如文章【Mandelbrot83】)。他把自相似曲线的各种形式称为不规则的图形。线是一维的，平面是二维的，但是有些自然现象介于两者之间。比如，定义一条曲线在一个有限的矩形内无限长：他们的维数介于1和2之间。

曼德布洛特和其他人的工作，关于自然界中不规则事物，在数学上和计算机图形学上产生了大量的调查研究结果，在世界许多研究中心同样很令人兴奋。

A4.3 曼德布洛特集合

图形学提供了很强大的工具来研究一些很有趣的集合。一些人认为曼德布洛特集合是数学上最复杂的对象。基于简单的几个很令人惊讶的定义，它的结构居然这么丰富，借助计算机图形学来表示的时候能够得到漂亮到令人吃惊的图片。

曼德布洛特集合是从分析的一个分支**迭代理论**中发展起来的。迭代理论研究的是无止境的反复调用某个函数时会产生什么现象。相关集合Julia set的很多关键的结果，在本世纪早期被Gaston Julia(1893～1978)和Pierre Fatou(1978～1929)发现(当然，并没有借助计算机)。他们的想法潜伏了很多年，直到他们再次被想起，并且被本瑟姆·曼德布洛特在20世纪70年代的时候扩充。作为他研究的一部分，他用计算机图形学做非常重要的实验，依靠很多线条激发了一些猜想，使得能够进一步分析和发现。

在这些发现的背后，有很多很有意思的故事，(见【Mandelbrot83，Peitgen88】)，同时附带了一些漂亮的图片，产生的方法我们下面会讲到。

A4.3.1 曼德布洛特集合和迭代函数系统

图A4.1给出了曼德布洛特集合的一个直观的例子。彩色图片可以在Plate37得到。它的内部是黑色的，看起来是由心形线组成，旁边还有很多瘤状圆。事实上，它的边界十分复杂，在边界的某个地方特写，不断放大，就可以看出它的复杂性。

理论上，可以永远重复地放大——边界是无限复杂的，事实上它就是个不规则曲线！图像上的每一个点都是渐变的，或者根据在重复函数系统上的实验结果给它标记颜色。关心的重复函数系统如图A4.2。它使用的是一个相当简单的函数，如等式A4.1，其中c是常数。

$$f(z) = z^2 + c \tag{A4.1}$$

这个系统产生的输出是输入的平方与c的和。我们认为这个过程从初始值0开始，因此这个系统产生了一个值的序列或者轨迹(见第2章)，如等式。(仔细检查)这个轨迹的只决定于一个因素：c的值。

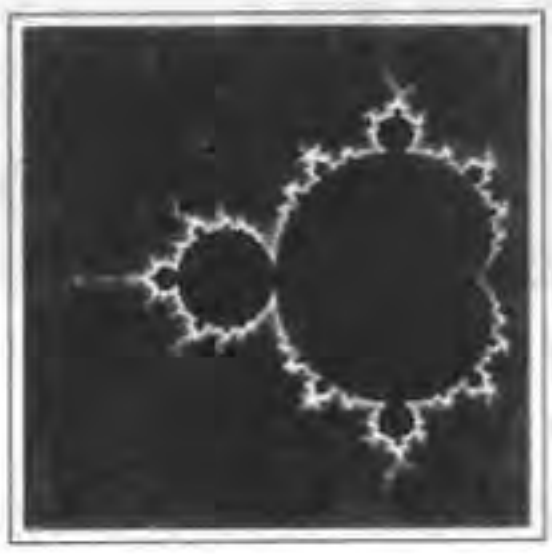


图 A4.1 曼德布洛特集合

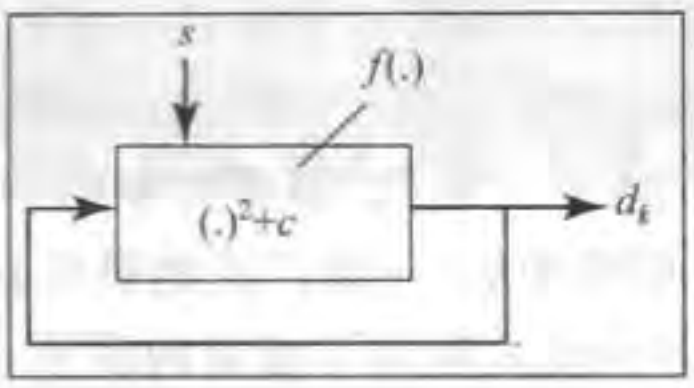


图 A4.2 曼德布洛特集合的迭代函数习题

$$
\begin{aligned}
d_1 &= c \\
d_2 &= (c)^2 + c \\
d_3 &= ((c)^2 + c)^2 + c \\
d_4 &= (((c)^2 + c)^2 + c
\end{aligned}
\tag{A4.2}
$$

注意到给定一个 c,经过一次迭代 0 的轨迹变成了 c 的轨迹。不管是从 0 开始,还是从 c 开始,我们得到的是同样的轨迹。c 的轨迹就是曼德布洛特集合主要研究的对象。最基本的问题是:给定一个 c 的值,当 k 不断变大的时候,轨迹上的点 d_k 如何表现?尤其是,当这个轨迹仍然是有限的时候(轨迹上所有的点和 0 的距离都是有限的)或者它会剧增(接近无限)吗?下面我们定义更准确一点,轨迹上仍然有限的点属于其相应的曼德布洛特集合,而剧增的点不属于这个集合。

例 A4.1 c 取不同值的时候,0 的轨迹

这里我们考虑 c 取不同值的时候的轨迹:

令 $c=-1$。这个轨迹是 0,0,0,0,0,0,0,……,无限重复。然而,这个轨迹是有限的,并且基于 $c=-1$ 的值属于曼德布洛特集合。

令 $c=1$。这个轨迹是 0,1,2,5,26,677,458330,……剧增,因此基于 $c=1$ 的值不属于曼德布洛特集合。

令 $c=-1.3$,这个轨迹是 0,-1.3,……,大约经过 50 步,可以得到一个循环的序列:-1.148665,0.019430,-1.299622,0.389018,不断重复。

我们尝试着把这些结果画出来,看看 c 取值在哪个范围的时候可以得到有限的轨迹。图 A4.3 给出了第一步尝试。它说明了选择其他 c 的值时候得到结果。比如,我们从 0 开始增加 c,存在有限轨迹的最大值是什么?

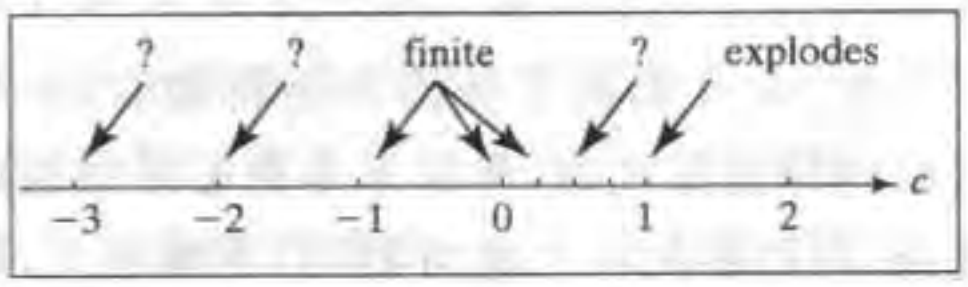


图 A4.3 对于 c 的每个值,0 的轨迹如何

这个系统已经展示了有趣的动态过程。当 c 取复数的时候,这个问题变得更加复杂。复数算术常使用在应用函数中。从算术到复数平面是曼德布洛特集合在理论上很重要的贡献。曼德布洛特集合在复数平面上存在。回想一下,图形学里我们用一个图表来表示复数:一个复数 $z=x+yi$,表示成在点(x,y)处。

图 A4.2 所示的系统能够很好的表现复数,就和实数一样。现在 c 的值是个复数,每次迭代,我们要把上次的结果平方,然后加上 c。回想平方复数 $z=x+yi$,可以得到一个新的复数,如等式(A4.3)。

$$(x+yi)^2=(x^2-y^2)+(2xy)i \tag{A4.3}$$

这个系统的固定点的注释

研究曼德布洛特集合时，检测系统 $f(.)=(.)^2+c$ 的不动点是很有用的。轨迹的行为很大程度上决定于这些不动点，这是因为，那些复数 z 影射到自己的时候 $z^2+c=z$。可以把这个式子转化成二次方程 $z^2-z+c=0$。系统的不动点就是该方程的两个根，表示为：

$$p_+,p_-=\frac{1}{2}\pm\sqrt{\frac{1}{4}-c} \tag{A4.4}$$

注意，我们处理的是复数情况，因此求 $1/4-c$ 的平方根没有问题，即使它是负数或者复数。附录 2 中给出了如何编程计算平方根。它们是系统的两个不动点。对于先前的例子，我们可以得到两个不动点 $p_+=1.2499323-0.33667i$，$p_-=-0.249227+0.33667i$。如果一个轨迹到达了某个不动点 p，它就会一直停留在那里。两个不动点位置上是对称的，它们距离 $1/2+0i$ 的距离是相同的。

可以更深层次地理解不动点，认为它们是引力中心(attracting)或者挡板(repelling)。严格地讲，如果一个轨迹离某个不动点 p 很近的话，轨迹上的下一个点必然满足下面的条件：

- 如果 p 是引力中心，则距离 p 更近。
- 如果 p 是挡板，则距离 p 更远。

如果一个轨迹距离某个引力中心不动点近的话，它会被"吸"进去；而对一个挡板不动点来说，会越来越远。很难表现出一个不动点是引力中心，只有它和源点的距离在 1/2 以内时，这个时候这些点在以源点为中心以 1/2 为半径的圆内。

图 A4.4 显示了不同 c 取值时的轨迹的例子，给曼德布洛特集合添加粗略的解释，使看起来更明白些。每个轨迹从各自的 c 开始，表示为一个点(我们可以从 0 开始画每条轨迹，但是这样的话，会导致整个图片被画满)。

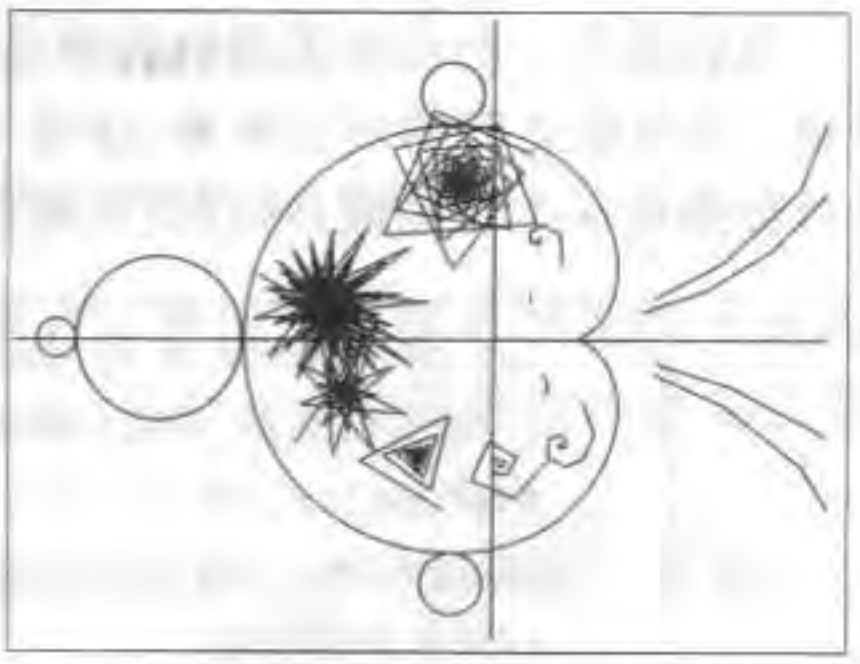


图 A4.4　不同 c 取值时的轨迹的例子

认真注意，每个点表示了是用在系统 $f(.)=(.)^2+c$ 中的相关 c 的值是重复的，和该轨迹起始点相同。

如果 c 选择的是不在曼德布洛特集合(称为集合 M)的元素轨迹的结果就会剧增。两个剧增的轨迹在途中表现出来了。如果选择的 c 恰好在 m 的边界上的话，该轨迹会变化很强烈，最后接近无穷。这些轨迹可能会使得图片混乱，因此没有画出来。

如果 c 的值在 M 内部，相应的曲线可以做很多事情。对于某些 c，它可能会很快到达一个不动点或者螺旋式的到达(当 c 比较接近 M 的边界时，这个速度非常慢)。另一方面，没有一个不动点是引力中心的话，这个轨迹一点都不会收缩到某个固定的值。取而代之的是，它会进入一个环形的序列(通常是周期性的)，在某个挡板不动点的周围。举个例子，c 取值的轨迹在 M 的顶部或者底部的小圆圈中的，周期是 3。

A4.3.2　如何判断点 c 是否属于曼德布洛特集合

现在转向应用，用于两个目的：

1. 更深入地研究曼德布洛特集合的特性。

2. 创造曼德布洛特集合炫目的全彩色图片。

给出 M 中的一个复数 c。路线是根据式检测轨迹上数字 d_k 的大小。当 k 增加的时候，$|d_k|$ 的值可能会剧增(c 不属于集合 M)或者不会(c 属于集合 M)。来自复数分析的一个理论说明，如果 $|d_k|$ 在某个时刻超过 2，那么轨迹在某个点的时候一定会剧增。使得 $|d_k|$ 大于 2 的重复的次数称为是该轨迹的 **dwell**，轨迹在某个区域停留的时间可能会上升。

但是如果 c 属于集合 M，轨迹必然存在一个有限的 **dwell**，如果没有无限重复我们就不知道这一点。我们最多可以设定重复最多次数的上限 Num。一个很典型的值就是 $Num=100$。如果重复了 Num 次后，$|d_k|$ 还没有超过 2，我们就认为它不可能超过了，认为 c 属于 M。可以得出刚好在 M 边界外的 c 的取值的那些轨迹存在一个很大的 $dwell$，如果这些 $dwell$ 超过了 Num，我们就会错误地把他们判定为在 M 内部。(属于 M 集合 c 的取值曾经被错误地解释过吗?)基于很小 Num 图画可以表现出曼德布洛特集合是相当巨大的。

图 A4.5 给出了如果压缩这些计算的程序 dwell()。对于给定的 $c=c_x+c_yi$，它的返回值是当 $|d_k|$ 大于 2 所需要的重复的次数，或者如果经过 Num 次重复没有超过 2，就返回 Num。为了简单，它为函数 dwell()定义了参数的数据类型。

每一次重复，和当前 d_k 相关的数对 (d_x,d_y)，先被平方然后加到数对 (c_x,c_y) 中得到一个 d 的值。$|d_k|^2$ 的值保存在 fsq 中，并且和 4 进行比较，这个和，用 $|d_k|$ 和 2 比较是等价的，而且还不用每次都求平方根。函数 dwell()在描绘曼德布洛特集合中起很重要的作用。

A4.3.3 描绘曼德布洛特集合

我们想在一个光栅图形设备中显示 M。为了实现这个，必须建立起显示器中的每个像素和 c 的取值之间的对应关系，还有 c 的值和其响应的 dwell。根据 dwell 是否有限或者它是否会到达某个有限值，给它设定颜色。

```
int dwell(double cx,double cy)
{//返回真正的 dwell 和 Num 中较小的一个
#define Num 100                    //想得到更好的图片就要增加该值
double tmp, dx=cx,dy=cy,fsq=cx * cx+cy * cy;
//开始计算轨迹
for(int count=0;count<=Num && fsq<=4;count++)
{
tmp=dx;                            //保存旧的实部
dx=dx * dx-dy * dy+cx;             //新的实部
dy=2.0 * tmp * dy+cy;              //新的虚部
fsq=dx * dx+dy * dy;               //所用的迭代次数
}
return count;
}
```

图 A4.5 估算 dwell

曼德布洛特集合的这个简单的图片，把在 M 内部的点设定为黑色(dwell 达到最大值)，在 M 外部的点设定为白色(dwell 比最大值小)。如果颜色和 M 外部的点联系起来的话，图片会更加吸引眼球。这些点的 dwell 比最大值小。根据 dwell 的大小不同，我们可以分别设定不同的颜色。dwell 小的值代表"hot"点，这些点的轨迹很快就剧增了，因此它们被标记为鲜艳的红色或者黄色。dwell 大的值代表轨迹"逃逸"的比较缓慢，这些点应该被标记为绿色或者蓝色。图 A4.6 使用五种颜色显示了一个可能的颜色设定的方式。使用的颜色越多，图片会更加漂亮。这些颜色的选择也是很仔细的，以达到最终的图片具有最好的感觉。想了解更多颜色的版本请参看相关站点。

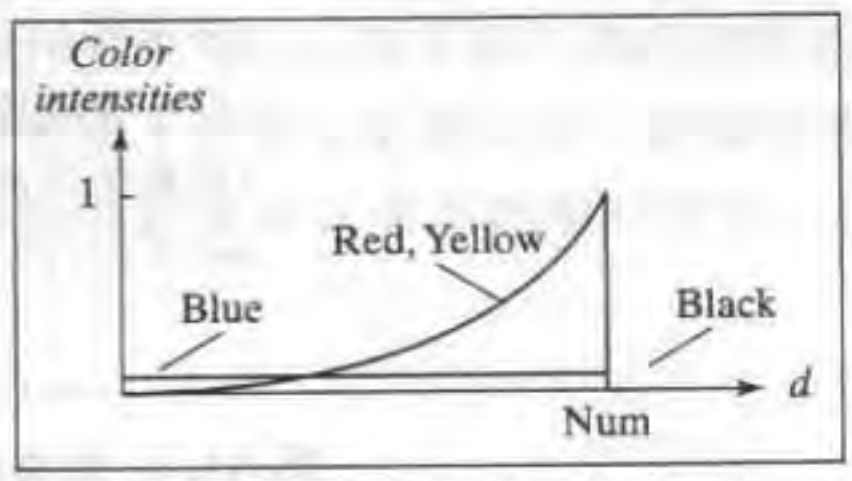


图 A4.6　根据轨迹上的 dwell 来赋予颜色值

还有一个遗留的问题是对于一个特殊的 c 的取值，如何与某个像素联系。图 4.7 给出了一个简单的方法。使用者指定期望的图片在屏幕上的大小，使用数据如下：

行的数目(如 rows=80)

列的数目(如 cols=120)

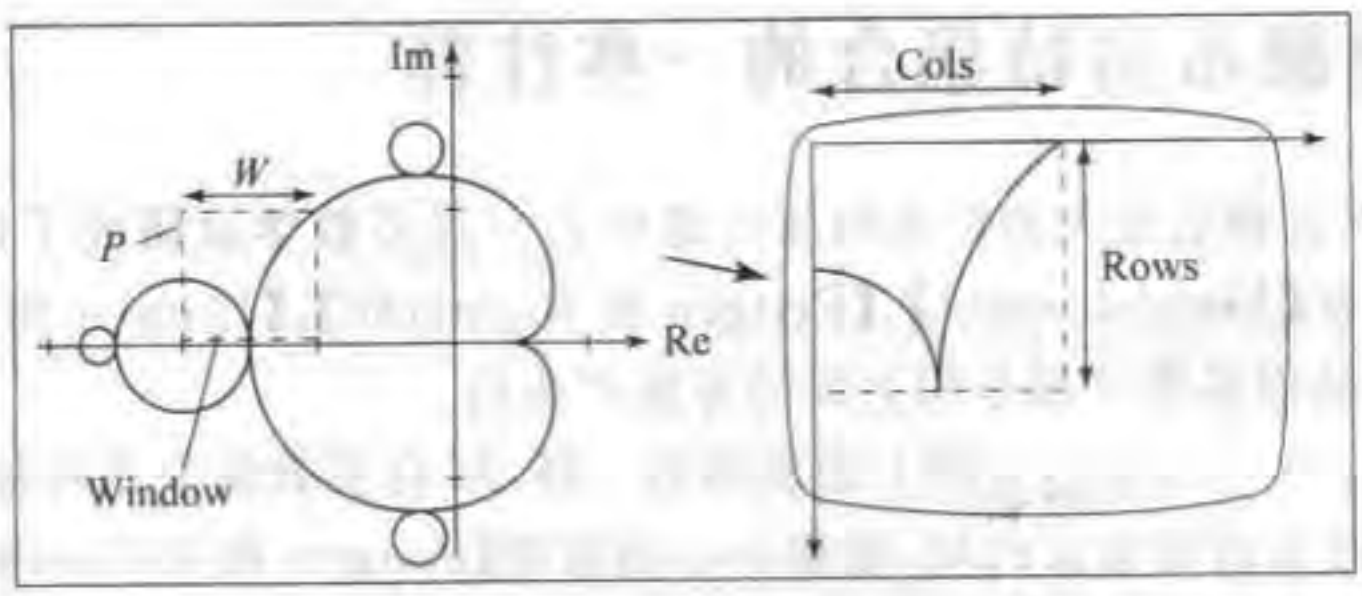


图 A4.7　建立 M 是你干的一个窗口以及点和像素之间的联系

这个决定了图像的纵横比：$R=rows/cols$。使用者还要选择要显示的复数平面的端口：与原图像有相同纵横比的矩形区域。要做到这一点，用户只需要简单地指定左上角 P 以及宽度 w(如：$P=-1+0.5i$ 或者 $P=(-1,5)$，$W=0.5$)。

计算机完成剩下的事情。矩形的高度被设定为需要的纵横比需要的高度。和图像中建议的方式一样，我们在左上角显示图像。

现在第 i,j 个像素的中心对对应的复数值 $c=c_x+c_yi$ 是多少？结合梯度，可以用等式(A4.5)表示。

$$c_{ij}=\left(P_x+\frac{i+\frac{1}{2}}{cols}W,\quad P_y-\frac{j+\frac{1}{2}}{rows}W\right) \tag{A4.5}$$

其中 $i=0,\cdots,cols-1$，$j=0,\cdots,rows-1$。

曼德布洛特集合所选择的区域被逐像素的画出来。对于每个像素，相关的值 c 被传给 dwell()，然后根据计算出来的 dwell 可以选择合适的颜色。该像素也就被设定为这个颜色。图 A4.8 给出了该算法的伪代码。

```
for(j=0;j<rows;j++)
for(i=0;i<cols;i++)
{
<根据式 A4.5 找到响应的 c 的值>
<估计该轨迹的 dwell>
<根据估计出来的 dwell 找到相应的颜色>
setPixel(j,i,color);
}
```

图 A4.8 绘制曼德布洛特集合一个区域的伪代码

一个实际的例子是研究曼德布洛特集合的特写镜头(close-up views),数字需要以高精度方式被记录和操作。需要双精度(或更高精度)的算术操作。同时,当与集合的边界比较近的时候,像上面讨论的一样,应该使用大一点的 Num。因此,当放大 M 的区域的某个边界时,每一副图片的计算次数会增加。但是,适当大小的图片,都能够在合理的时间内被微型计算机构造出来,这个时间是值得期待的。

A4.3.4 曼德布洛特集合的一些注释

曼德布洛特集合和它强大的扩展的属性是什么?很多数学家研究了这个问题,也得到很多有趣的事实(见【Mandelbrot83】、【Peitgen 和 Richter86】、【Peitgen 和 Saupe88】有很多观点)。这本书包括很多图片都是用上面的方法产生的。

1. M 中的点 $c=-1$,$c=-2$ 同上面提到的一样,M 在实数轴上是对称的。
2. 最大的曼德布洛特集合的块,被很小心的检测【Peitgen 和 Richter86】:
   - M 的块中心的边界是个心形线,用参数表示如下:

     $$c.x = 0.25 + 0.5(1 - \cos(t))\cos(t)$$
     $$c.y = 0.5(1 - \cos(t))\sin(t)$$

     对不动点来说,心形线内部的轨迹是引力中心,从 $z^2+c=z$ 的的两个根可以看出。
   - 心形线左边的圆半径为 1/4,中心在 $c=-1$,在圆内的轨迹以 2 为周期。
   - 它左边小一点的圆,半径为 0.0607,中心在 $c=-1.3107$,圆内的轨迹以 4 为周期。
   - 在心形线上面和下面的圆半径都为 0.0954,中心是 $c=-0.1226\pm0.7449i$。在圆内的点多周期是 3。
3. 集合的边界是看起来最有意思的地方,而且事实上他是个不规则曲线。对某个区域不断进行放大,使用越来越小的窗口时,一些新的细节就会出现。图 A4.9 表现了以 $c=0.7469+0.1073i$ 为中心的区域的 3 种连续的放大效果。

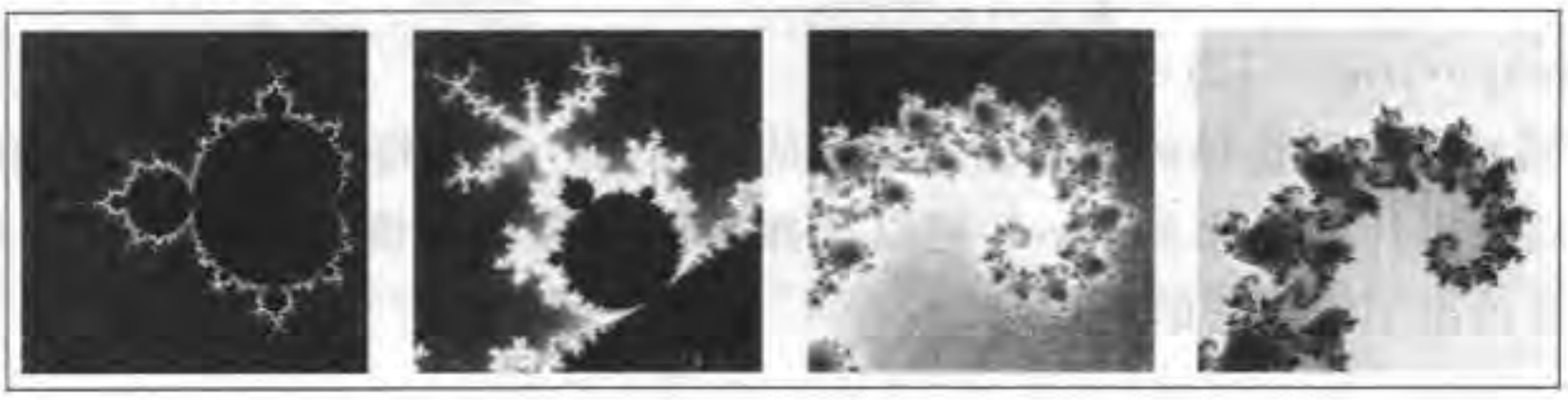


图 A4.9 在一个区域内缩放

序列中最后一幅图片使用的窗口，左下角是（－0.74758，0.10671），右上角是（－0.74624，0.10779）。每次放大时候，都可以看见一些新的细节。比如，一个缩放图中的一个单个的黑点，在下一次放大的时候变成了一个整个的新的瘤状的图像（“……一个小跳蚤拥有更小的跳蚤为它的战利品……；因此是个无穷的过程”）。最后，没有两个微小的缩影是一样的。另外一个被康奈尔大学的 John H. Hubbard 证明了的惊奇的事实是：曼德布洛特集合是连通的【Dewdney88】：在平面上即使是很微小的看起来毫无关系的瘤状结构，在父集合中总是存在很细小的点的序列把它们联系起来。

附录 5

相对性和海龟绘图

概述

在我们的工具包中加入一两个工具，将会获得明显的优势，能使一些画图任务变得更加容易。一般来说，固定一个点称为当前点(currPos)，而描述其他点时采用和当前点的相对位置，这样比较方便。因此，我们在写函数时，函数的参数表示点的变化：程序员指定绘制点必须沿着每个坐标轴走多远才能到达下一个目标点。

A5.1 编写 moveRel()和 lineRel()函数

下面，介绍两个新的程序：moveRel()和 lineRel()。函数 moveRel()较简单：将当前顶点移动(dx,dy)。函数 lineRel(float dx,float dy)实现同样的功能，但是它首先画一条从当前顶点到新的顶点的直线，然后再更新当前顶点。图 A5.1 展示了这两个函数。

```
moveRel(float dx,float dy){
currPos.set( currPos.x+dx , currPos.y+dy);
}
lineRel(float dx ,float dy) {
lineTo(currPos.x+dx, currPos.y+dy);
}
```

图 A5.1 函数 moveRel()和 lineRel()

例 A5.1 数组标记

在绘制的时候，不同形状的标记可以被放在许多点上以增加强调。图 A5.2 表明使用五角星(如图 A5.19 所描述)标记以突出线形图中的数据点。

因为同样的图要在许多不同的点绘制，简单地用 drawMarker ()会很方便，并且在 curPos 处绘制它。这样，图 A5.2 中的线形图能够使用下面的伪代码得到：

```
moveTo(first data point);
drawMarker();                    //在这里画一个标记
for(each remaining data point)
```

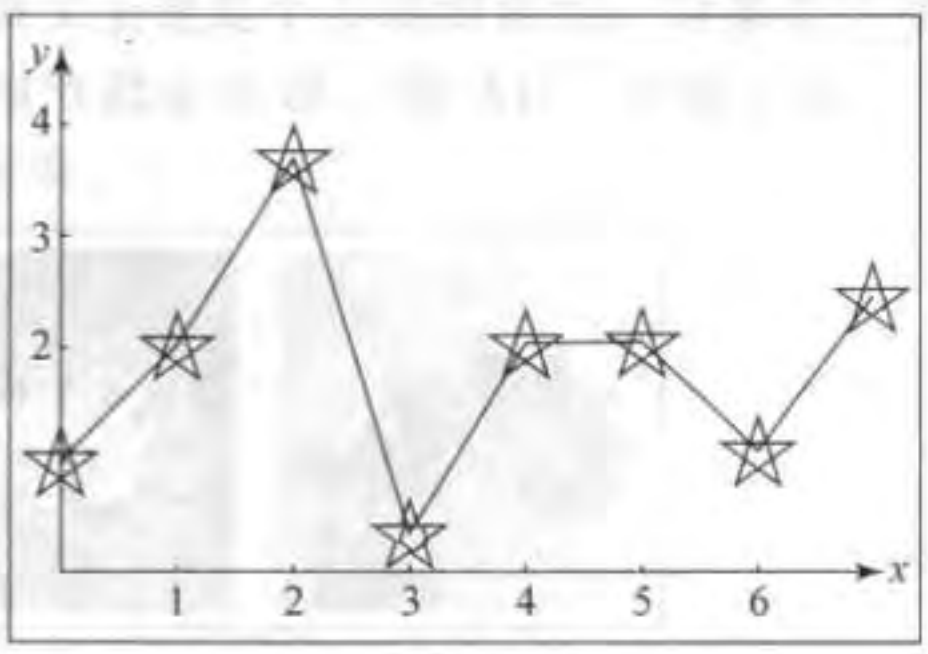


图 A5.2 为强调点放置五角星

```
{
lintTo(the next point);    //画出下一条线段
drawMarker();              //在 CP 画标记
}
```

图 A5.3 表示了一个箭头形状的标记，它是用图 A5.4 中的程序画的。这个箭头的最顶端定点就是 currPos，这样就能确定其位置。为了方便，函数使用 4 个参数 f、h、t 和 w，函数 arrow()只调用了 lineRel()，而且没用绝对坐标。应该注意的是，虽然 currPos 随着画图的进行而改变，绘图结束后 currPos 回到了它的初始位置。因此，这个函数没有产生副作用（在绘制操作自身范围之内）。

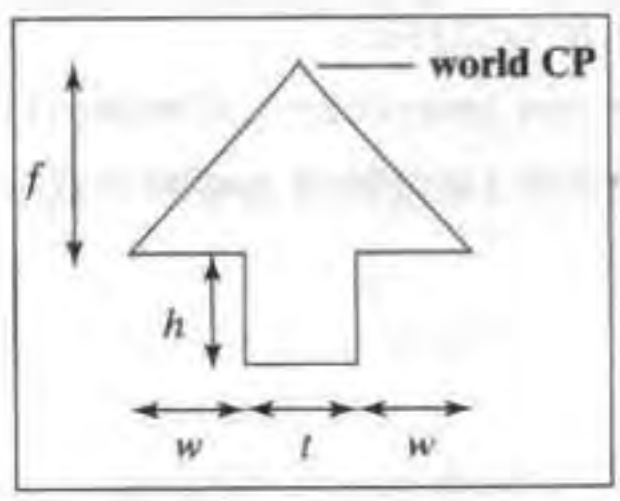


图 5.3 一个箭头模型

```
void arrow(float f, float h, float t, float w)
{
lineRel(-w -t/2, -f);      //往左下的边
lineRel(w, 0);             //往右
lineRel(0, -h);            //往下
lineRel(t, 0);             //王右
lineRel(0, h);             //往上
lineRel(w, 0);             //往右
lineRel(-w -t/2, f);       //往左上，回到最初顶点
}
```

图 A5.4 使用相对画图画一个箭头

A5.2 海龟绘图法

我们增加的另一个工具非常的方便。想法是一个海龟在纸上移动，在它自己的身后留下了一个看似线段的足迹。上述的海龟类似于绘图仪中的一支铅笔。它不仅用 currPos 保存了当前所处的位置，而且保存了当前的朝向。这是海龟绘图法（turtle graphics）的一种形式，是图形学中的一种自然的编程方法[①]。海龟被放在 currPos，朝向一个被称为当前方向的特定方向。currDir 表示从正 x 轴开始逆时针旋转的角度数。

添加函数功能来控制海龟是很简单的。首先添加三个函数：

(1) turnTo(double angle)。这个函数将 turtle 转向给定的角度，其实现如下：

```
turnTo(double angle){
currDir=angle;          //转向一个确定的角度
}
```

(2) turn(double angle)。这个函数将 turtle 逆时针旋转 angle 度，其实现如下：

```
turn(double angle){
```

① MIT 的 Seymour Papert 曾介绍过，是 LOGO 语言的一部分，可以教孩子们编程。参见[Abel81]。

```
currDir+=angle;
}
```

我们使用负的角度表示一个往右的旋转。注意,旋转是一个相对方向变化,我们不指定方向,只是指定方向的一个改变量。这个简单的函数能够对用海龟绘制复杂图形提供强大的功能。

(3) forward(float dist, int isVisible)。forward()稍微复杂一些:海龟从当前点 currPos 沿着当前方向 currDir 划一条长为 dist 的线段。注意,currDir 就像是一个静态变量一样,和背景颜色、画图颜色等变量类似。如果 isVisible 非 0,画一条可见线段,否则不画任何东西。

图 A5.5 表明,海龟在 x 方向移动 dist * cos(pai * currDir/180),在 y 方向移动 dist * sin(pai * currDir/180),所以 forward()的实现如图 A5.6 所示。

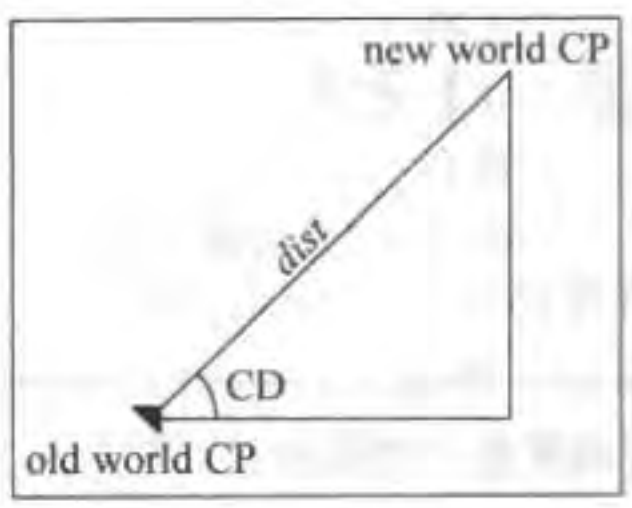


图 A5.5 forward()的功能

```
forward(float dist, bool visible){
const double radPerDeg=0.017453393f;
double x=currPos.x+(dist*cos(currDir*radPerDeg));
double y=currPos.y+(dist*sin(currDir*radPerDeg));
if(visible)
    lineTo(x, y);
else
    moveTo(x, y);
}
```

图 A5.6 forward()的实现

正如下面将要看到的,海龟绘图法让利用简单图形绘制复杂图形的工作变得很简单。

例 A5.2 画一个单位正方形

为了遍历一个单位正方形,只需画四条边,其中相邻两条边之间转动角度为 90°(如图 A5.7 所示),其实现如下面的伪代码。如果想让海龟结束的方向和开始的一致,则在最后加入 turn(90)语句:

```
forward(1,1);        //画一条边
trun(90);            //转第一个角度
forward(1,1);        //画下一条边
turn(90);            //转第二个角度
forward(1,1);        //画下一条边
turn(90);            //转第三个角度
forward(1,1);        //画最后一条边
turn(90);            //恢复到最初的角度
```

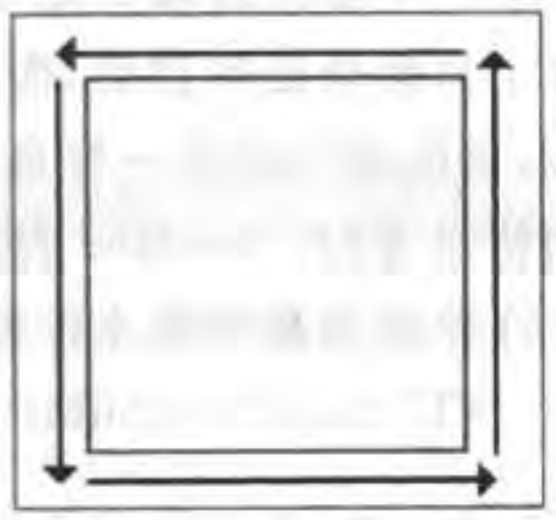


图 A5.7 画一个单位正方形

例 A5.3 建立一个以钩子为基本图案的图形

图 A5.8(a)中由三段组成的基本图案钩子能使用下面的代码绘制:

```
forward(3 * L, 1);     //L是长边的
turn(90);
forward(L, 1);
turn(90);
forward(L,1);
turn(90);
```

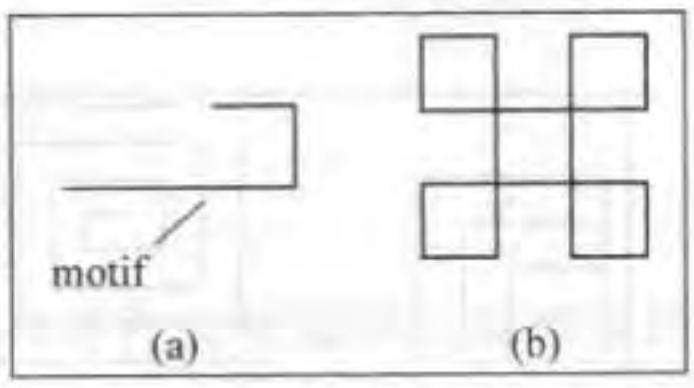


图 A5.8 通过几个海龟的运动来建立一个图形

假设一个函数 hook()包含了上面的代码，则图 5.8(b)中的图形能够用 4 个重复的 hook()画出来。该图能够通过改变初始的 currPos 和 currDir 进行平移和旋转。

例 A5.4 多线螺旋

大量被称为多线螺旋的图形能够很容易通过海龟绘图法产生。一个多线螺旋是一个每条连续线段都比它在某个方向相邻的前一条要大(或小)一个确定的值的多线。其框架类似于：

```
for(<一定数量的循环> )
{
    forward(length,1);        //沿当前方向画一条线
    trun(angle);              //转动 angle 角度
    length+=increment;        //增大线的长度
}
```

每画一条线，线的长度和方向逐步增加。如果增量为 0，则图形既不增长也不收缩。图 A5.9 展示了一些多线螺旋。

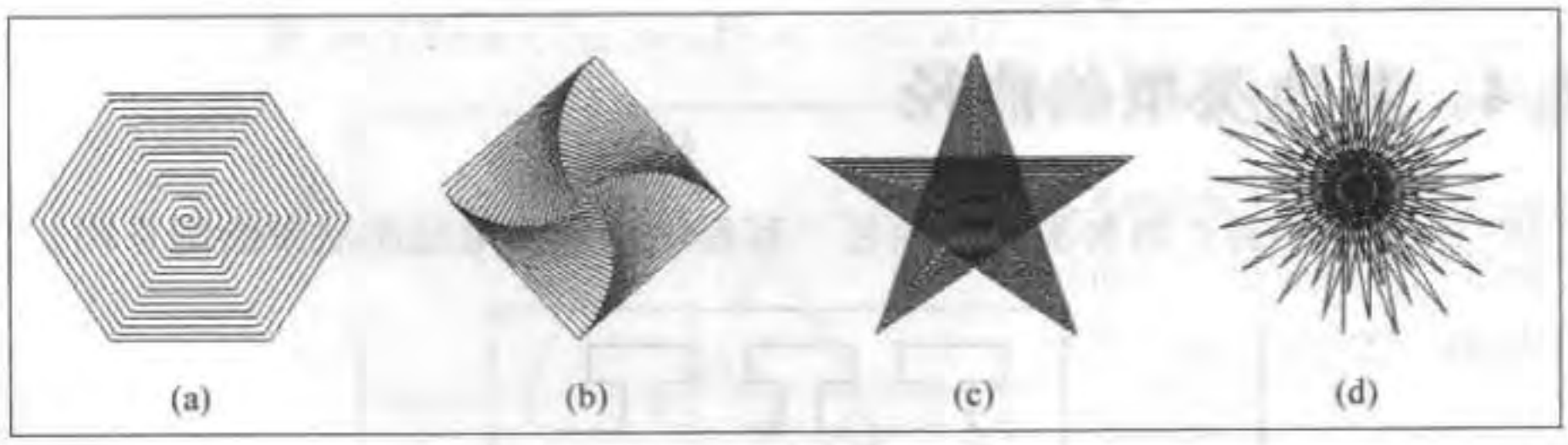


图 A5.9 多线螺旋的一些例子，角度为：
(a) 60；(b) 89.5；(c) −144；(d) 170

练习

练习 A5.2.1 绘制海龟图形

给出用海龟移动来绘制图 A5.10 所示的三个图形的程序。

练习 A5.2.2 绘制一个有名的标记

写一段程序让一个海龟来绘制图 A5.11 所示标记的轮廓(不需要填充多边形)。

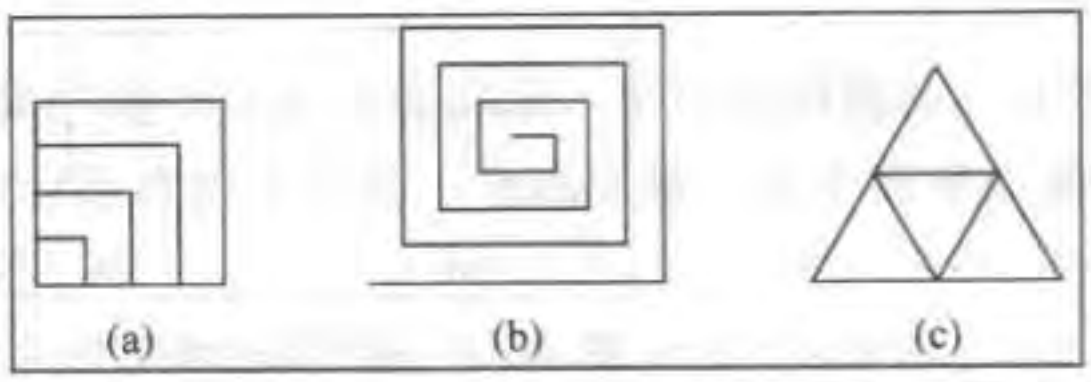


图 A5.10 另外一些简单的海龟图形

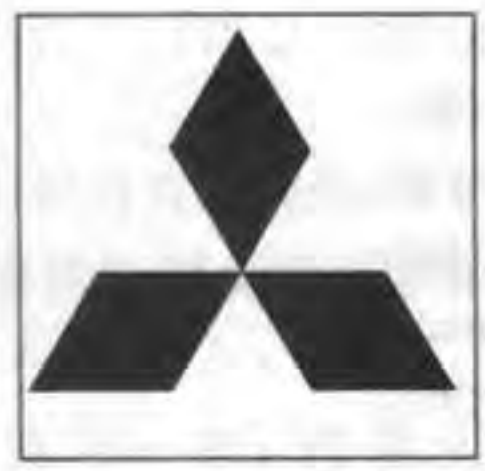


图 A5.11 一个有名的标记

A5.2.3 绘制曲径

曲径是类似于图 A5.12(a)所示的一种模式,经常是由连续的线绕某种路径形成。曲径常见于希腊的花瓶、中国盘子或者各个国家的地砖。这个曲径的基本图案如图 A5.12(b)所示。每个基本图案绘制之后,海龟转向以便准备绘制下一个基本图案。

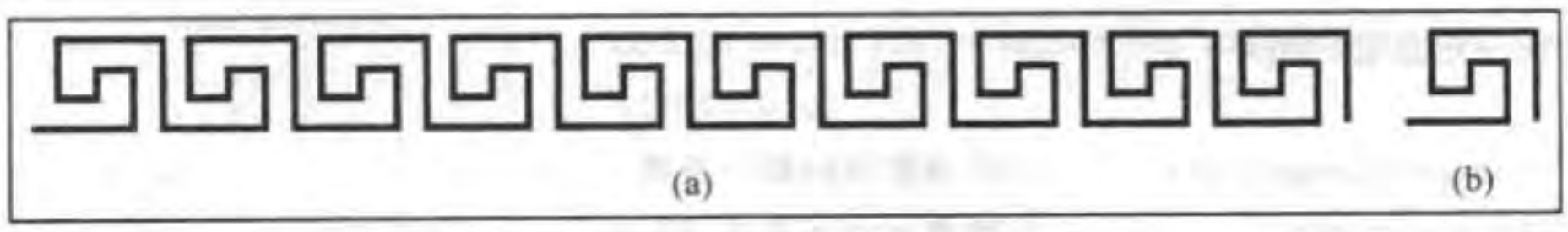


图 A5.12 一个曲径的例子

写一个程序来绘制这个基本图案,再写一个程序来绘制这个曲径(如果手头上有图形软件包可以支持控制线的宽度来画细线的话,曲径就很有吸引力)。正如后面的练习所述,可以设计出各种各样更为复杂的曲径。

A5.2.4 其他类型的曲径

图 A5.13 显示了另外两种类型的曲径。写程序利用海龟绘图法来绘制它们。

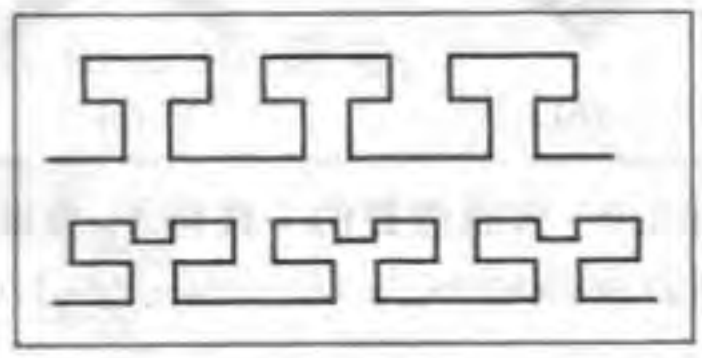


图 A5.13 额外的曲径图

A5.3 基于正多边形的图

正多边形构成了大量计算机图形学中常见的重要类型的形状,需要高效的方法来绘制它们。本节将介绍怎样实现上述功能并介绍怎样创建由正多边形演变的图形。

A5.3.1　正多边形

首先回顾一下正多边形的定义。

定义：如果一个多边形是简单多边形，它的所有边长度相等并且相邻的边有相等的夹角，它是正多边形。

正如第 1 章所述，如果一个多边形的任意两边不相互交叉（更精确的说，只有邻接的边可以接触，而且只能在它们的公共端点接触），则该多边形是简单的。我们给具有 n 条边的多边形起名为 n-gon。常见的例子有 3-gon（等边三角形）、4-gon（正方形）、5-gon（正五边形）和 8-gon（正八边形）等。图 A5.14 演示了各种例子。如果一个 n－gon 的边数很大，这个多边形看起来就像圆。事实上这就是后续将要介绍的绘制圆的一种方法。

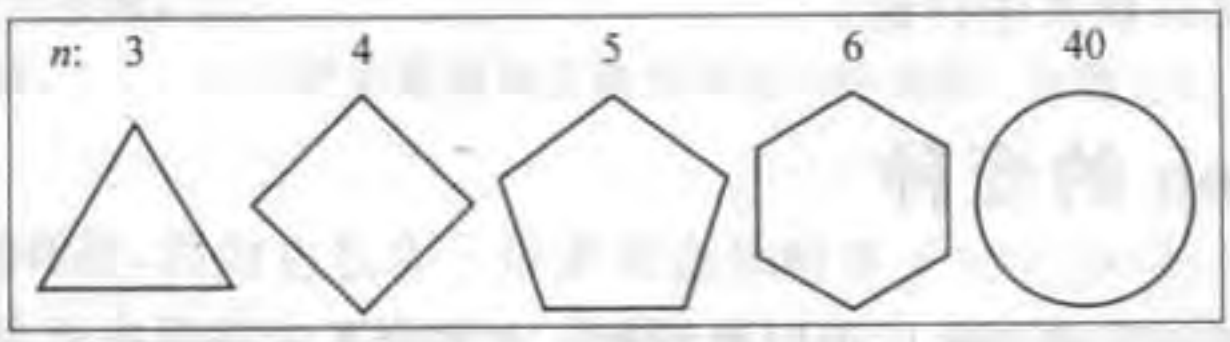


图 A5.14　n-gon 的例子

n-gon 的顶点在一个圆上，就是所谓的 n-gon 的母圆，这些顶点的位置很容易计算。图 A5.15 演示了一个正六边形的实例，其中每个顶点都均匀间隔 60°分布在圆上。半径为 R 的母圆中心在原点，第一个顶点 P_0 被放在正 x 轴上。其他顶点按如下方法分布，$P_i = R\cos(i-a), R\sin(i-a))$，$i=1,\cdots,5$，其中 a 是 $2\pi/6$ 弧度。类似地，普通 n-gon 的顶点位于

$$P_i = (R\cos(2\pi i/n), R\sin(2\pi i/n)), i = 0,\cdots,n-1$$

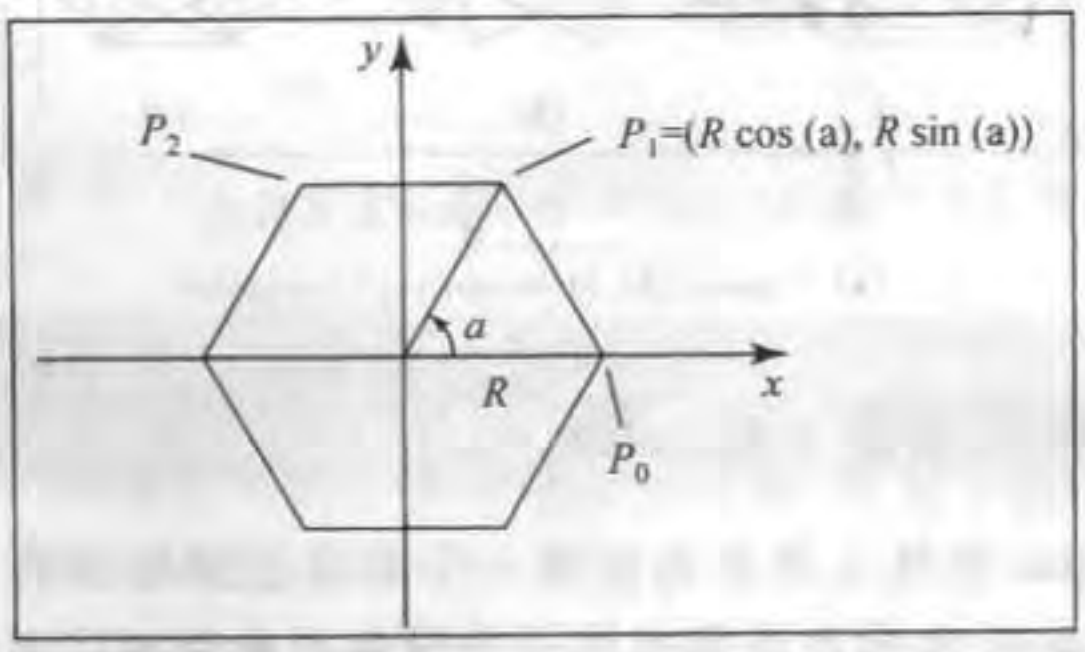


图 A5.15　计算 6-gon 的顶点

修改这个 n-gon 很简单。要想把它的中心调到 (cx, cy)，需要给 x 和 y 坐标分别加上 cx 和 cy。要想用比例因子 S 来缩放它，只需用 S 乘以 R。要旋转角度 A，只需给 sin ()和 cos ()的参数加上 A。实现几何变换更一般的方法在第 6 章中讨论了。正如在书中多次看到的那样，绘制 n-gon 的实现很直接。

例 A5.5 一个海龟-驱动[1]的 n-gon

用海龟绘图法绘制 n—gon 也很简单。图 A5.16 显示了怎样去绘制一个正六边形。海龟的最初位置和方向用小三角来指示。海龟简单地前行 6 次,每次移动之前逆时针转 60°。

```
for (i=0; i<6; i++)
{
    forward(L, 1);
    turn(60);
}
```

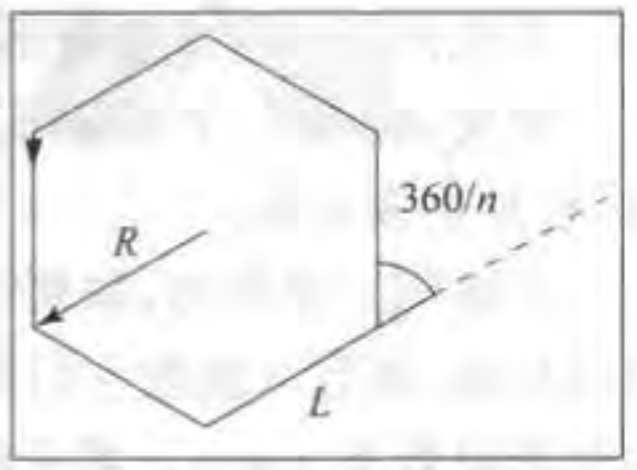


图 A5.16 绘制六边形

一个顶点位于初始 curPos,该操作保持 currPos 和 currDir 不变。绘制一般的 n—gon 及其几个变种的方法在联系中讨论。

A5.3.2 n-gon 的变种

基于 n-gon 顶点的有趣变种也可以被绘制。如图 A5.17 所示,n—gon 顶点可以被以多种方式连接以形成多种图案。图 A5.17(a)显示了通过连接邻接的顶点来绘制一个标准的 n-gon,而图 A5.17(b)则显示了通过连接每隔一个的点来形成 stellation(或星形图)。图 A5.17(c)显示了通过连接每个顶点和其他所有顶点形成 rosette。然后讨论 rosette。其他图案将在练习中描述。

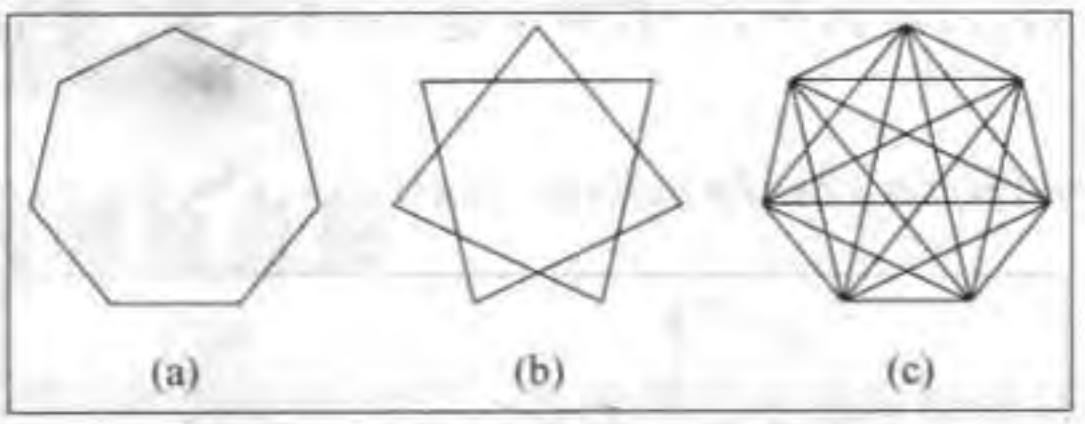


图 A5.17 一个 7-gon 及其后代

(a) 7-gon; (b) stellation; (c) 7-rosette

例 A5.6 Rosette 和黄金 5-Rosette

rosette 是一个 n-gon 中每个顶点和相隔一个的点连接形成的。图 A5.18 显示了 5-、11-,和 17-rosettes。rosette 经常被计算机图形学设备用来做测试图案。它有秩序的形状很容易揭示任何失真,而且设备的分辨率也可以通过观察遇到顶点时线条间的拥挤和模糊程度来确定。

Rosette 很容易被绘制:简单地将每个顶点和其他顶点相连。伪代码形式类似于

```
void Rosette(int N, float radius)
{
```

[1] Turtle-Driven 指利用上文所述的海龟绘图法。

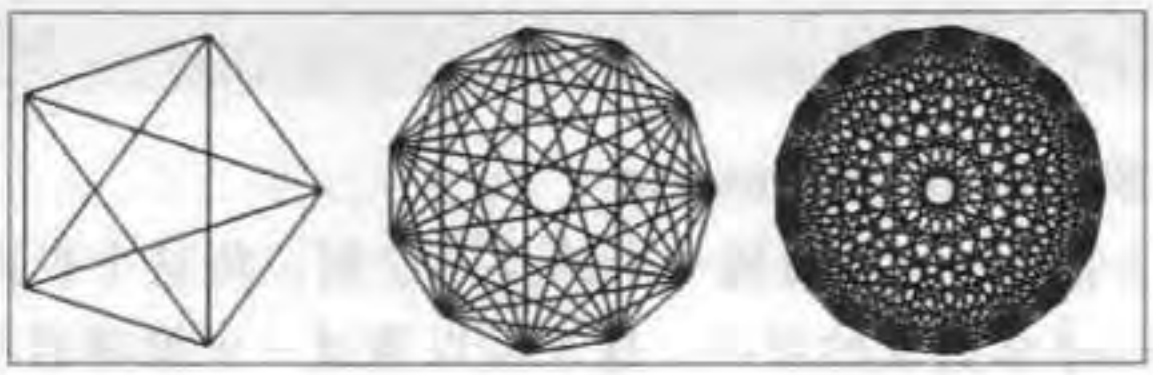


图 A5.18 5、11 和 17-rosettes

```
Point2 pt[< 针对最大的 rosette 的足够大的数> ];
< 生成图 3.43 中类似的顶点 pt[0], … pt[N-1]>
for(int i=0; i<N; i++)
for(int j= i+1; j<N; j++)
{
<forward(正确的距离, 1)>
turn(正确的角度);     // 正确的距离和正确的角度一样对每一步都相同
}
```

5-rosette 特别有趣，因为它包含了很多黄金分割率 ϕ 的实例（回顾第 2 章）。图 A5.19(a)显示了一个由外五边形和内五角形组成的 5-rosette。希腊人看到了这张图里的神秘意义。它的分隔有有趣的意义：每段都比下一个更小的长 ϕ 倍（见练习）。同样，由于五角星的边形成了内五边形，五边形的无穷回归是可能的，如图 A5.19(b)所示。

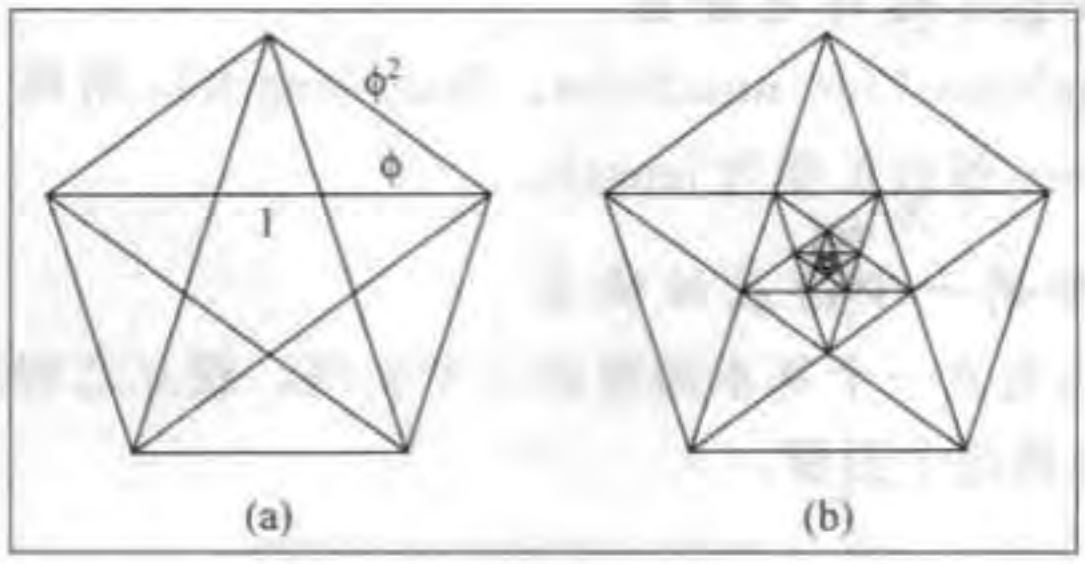


图 A5.19 5-rosette 和无限回归——五边形和五角星

例 A5.7 基于两个共心 n-gon 的图形

图 A5.20 显示了一些建立于两个同心母圆的形状，对某个因子 f 外半径 R 和内半径 fR。每个图用了一个半径在内外半径间相互交替的 n-gon 的一个变种。图 A5.20(a)和图 A5.20(b)两部分演示了基于 6-gon 和 10-gon 的熟悉的公司商标。图 A5.20(c)基于 14-gon，图 A5.20(d)显示的表现了内圆。

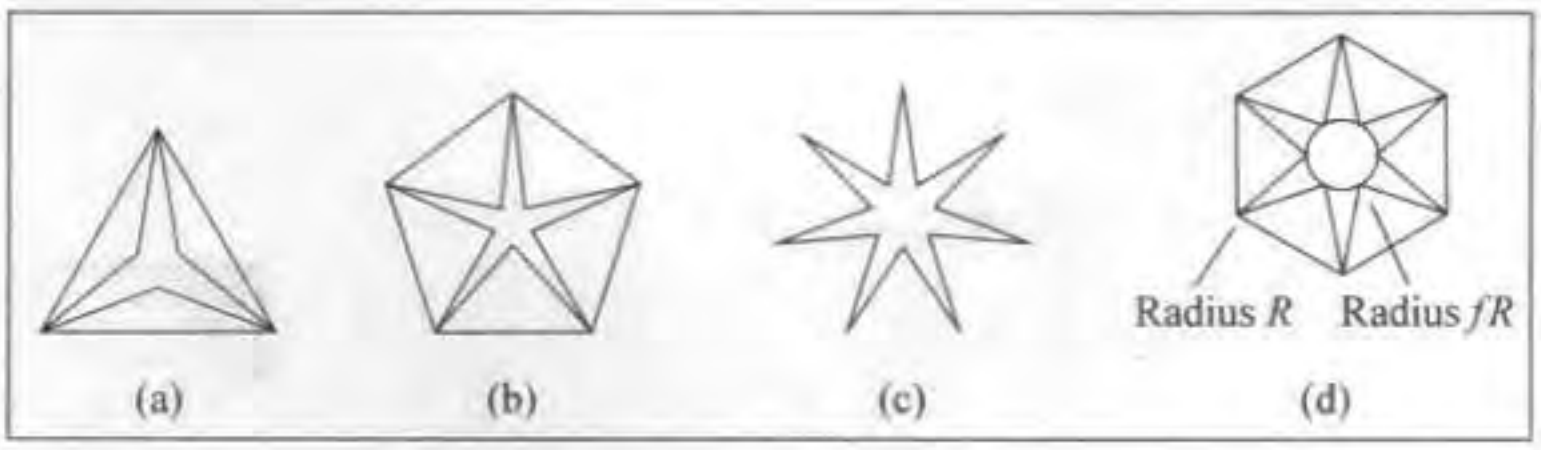


图 A5.20 熟悉的 n-grams 和 n-gons

练习

练习 A5.3.1 Stellation 和 rosette

五角星通过遍历 5-gon 并连接每隔一个的点来绘制。将这个扩展到任意的奇数 n-gon 并开发绘制所谓 stellated 多边形的程序。这个可以通过一个简单的初始化 moveTo()后面仅跟 lineTo()来完成吗(也就是说,不能抬笔)? n 是偶数又如何?

练习 A5.3.2 N-rosette 有多少条边?

证明基于 N-gon 和 N-rosette 的 rosette 有 $N(N-1)/2$ 条边。这和 N 个人坐成一圈每个人都和其他所有人一起敲玻璃的时候一个人所能听到的敲击声数目一样。

练习 A5.3.3 星形的几何形状

证明 5-rosette 上的每段长度都和下一个更小的满足黄金分隔率。解决这个问题的一个办法是证明五角星的角是具有内角弧度为 $\pi/5$ 的黄金角。证明 $2*\cos(\pi/5)=\phi$ 且 $2*\cos(2\pi/5)=1/\phi$。另外一个方法是仅用五角星中的两组相似角及 ϕ 所满足的关系 $\phi^3=2\phi+1$。

练习 A5.3.4 用相关的移动和绘制来画星

写一个程序,只用相关的移动和绘制来画中心在 currPos 的五角星。

练习 A5.3.5 n-gon 的海龟绘图

写一个程序 turtleNgon(int numSides, float length),用海龟绘图法来画一个有 numSides 个边的 n-gon,每边长度为 length。

练习 A5.3.6 绘制一个潜在的标志

图 A5.21 所示标志包含一个基本图案的三个实例。相互之间旋转了一定的角度。写一个程序用海龟绘图法画这个图形。

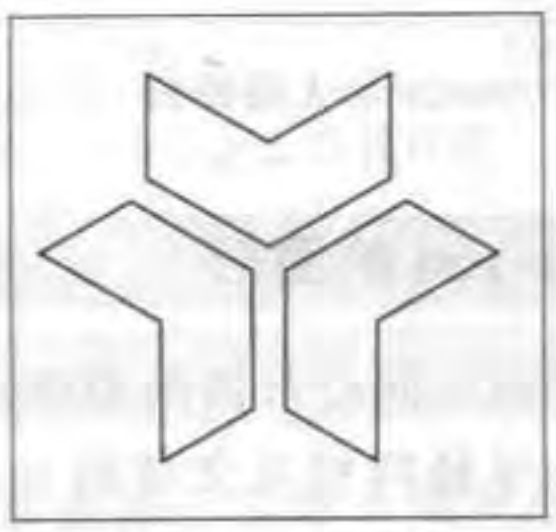


图 A5.21 一个标志的设计

COMPUTER GRAPHICS USING OpenGL

计算机图形学

（OpenGL版）（第3版）

计算机图形学是利用计算机研究图形的表示、生成、处理和显示的一门重要的计算机学科分支，而且是计算机科学中最活跃的分支之一。近年来，随着计算机及互联网技术的迅速发展，计算机图形学正越来越深入我们的生活，它在工业建模、视频处理、游戏制作、影视特技、生物信息和医药医疗等各行各业都有着及其重要的作用。可以说，计算机图形学的应用无所不在，且日益广泛；已经成为计算机科学技术与其他应用学科之间沟通的桥梁，成为许多计算机从业人员的必备素质之一。

本书是一本国外很有影响的教材，为许多国外著名大学所采用。本书通过最能代表技术发展状况的示例综合介绍了计算机图形学方面的原理和技术，本书对每个概念都进行了详细介绍，阐述了其背后的数学原理，并给出了用OpenGL实现的代码以及实现结果展示。新版本还为读者提供了计算机图形学领域的最前沿信息。本书是计算机图形学课程的很好教材，也是计算机图形学专业人员的很好参考书。

本书特点

* 本书配套的网站提供了书中使用到的代码、彩色插图、完整程序以及与本书密切相关的参考素材。
* 以C++语言作为程序设计语言。
* 深入介绍3D图形学及其数学知识，使学生可以创建真实的3D图形。
* 每章后面都给出了实际的案例研究。
* 清晰地介绍了计算机图形学的原理和技术。
* 大量真实而逼真的示例。
* 大量实践练习（每章大概有30多道）。

PEARSON
Prentice Hall

http://www.pearsoned.com

ISBN 978-7-302-18697-7

9 787302 186977 >

定价：99.00元