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）

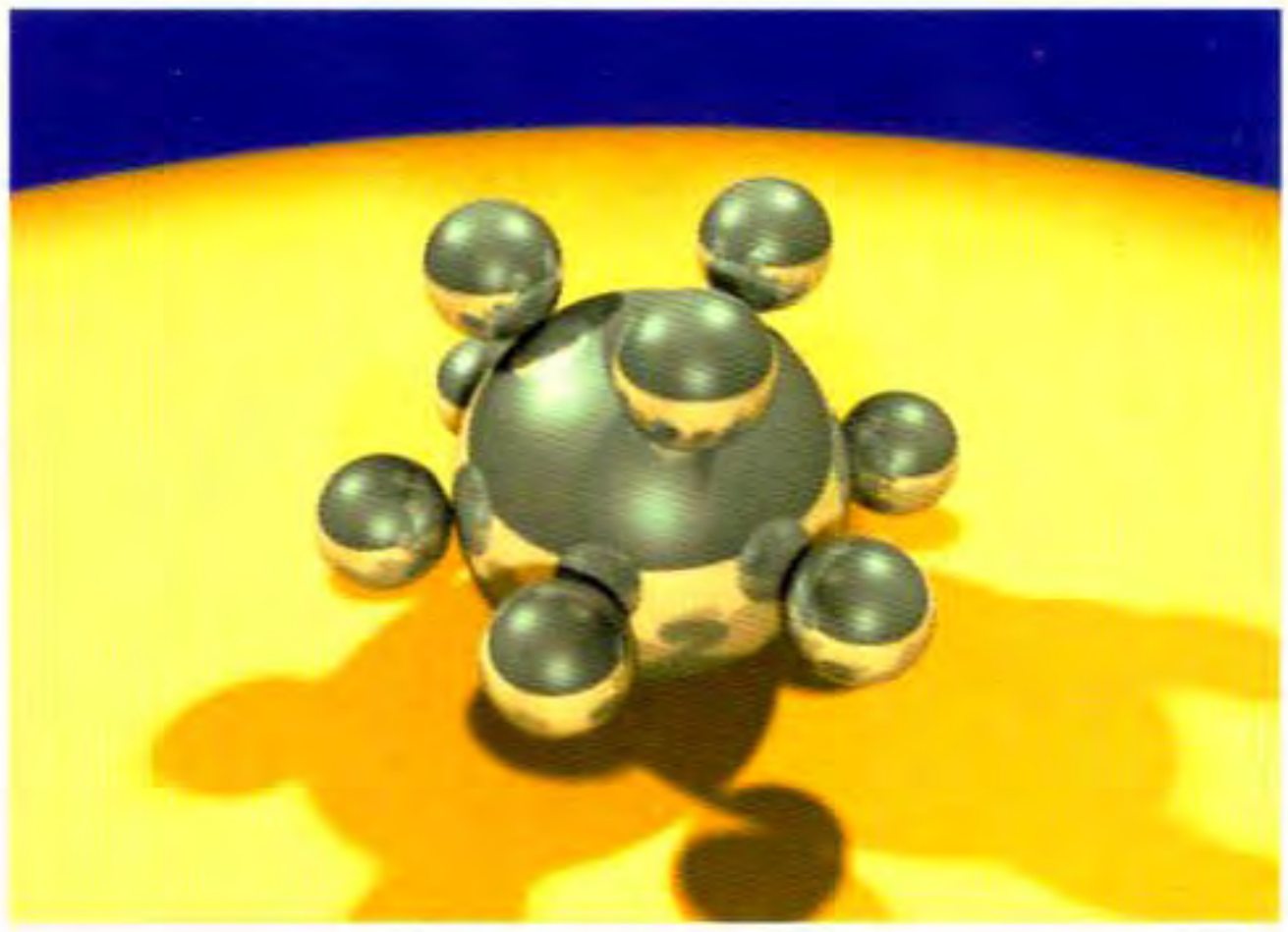
Francis S Hill, Jr.
Stephen M Kelley 著

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）

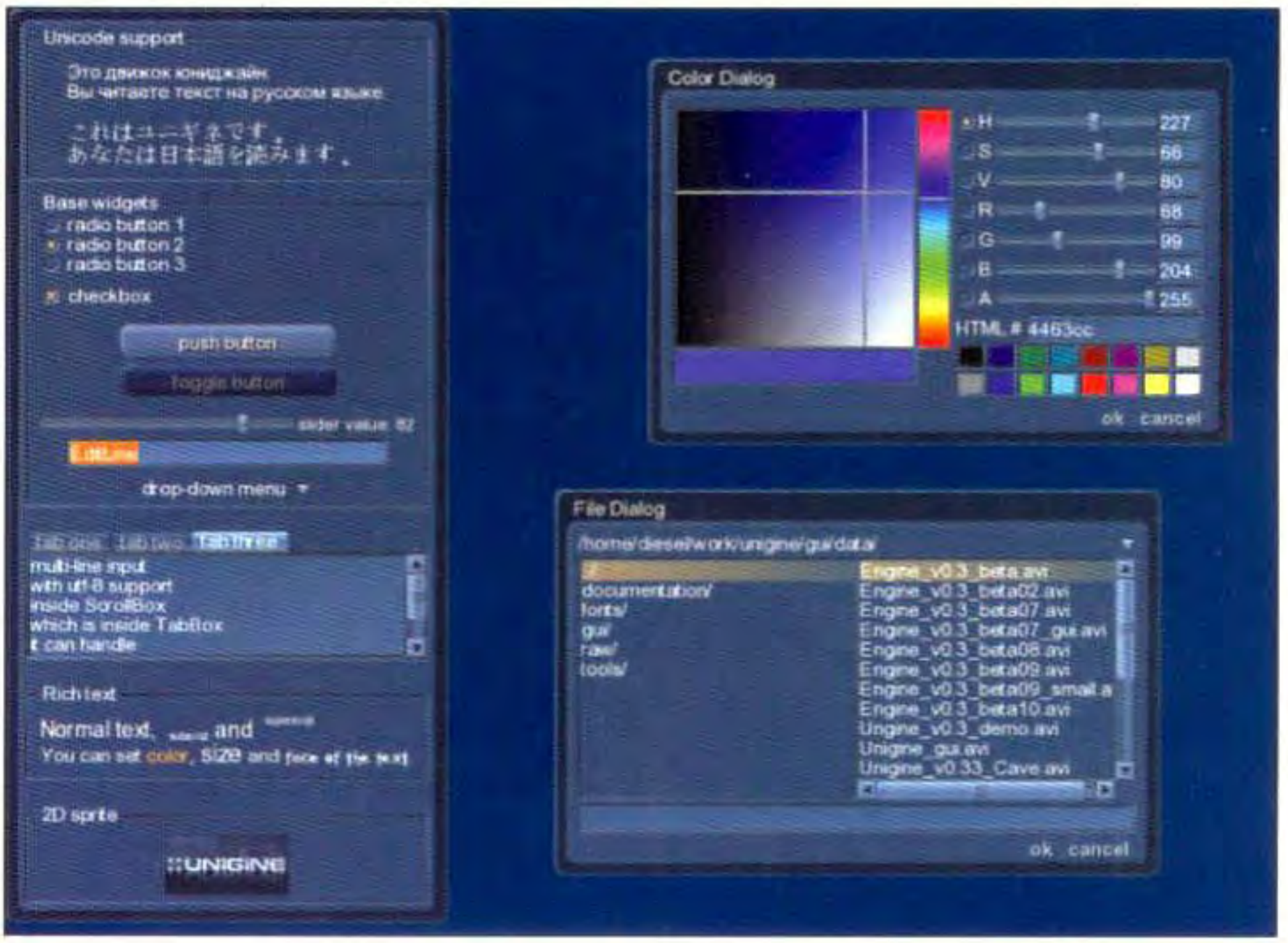
彩图 2：一个 3D 计算机游戏的截屏
（版权归 *High Moon Studios*，*Inc.* 所有）

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）

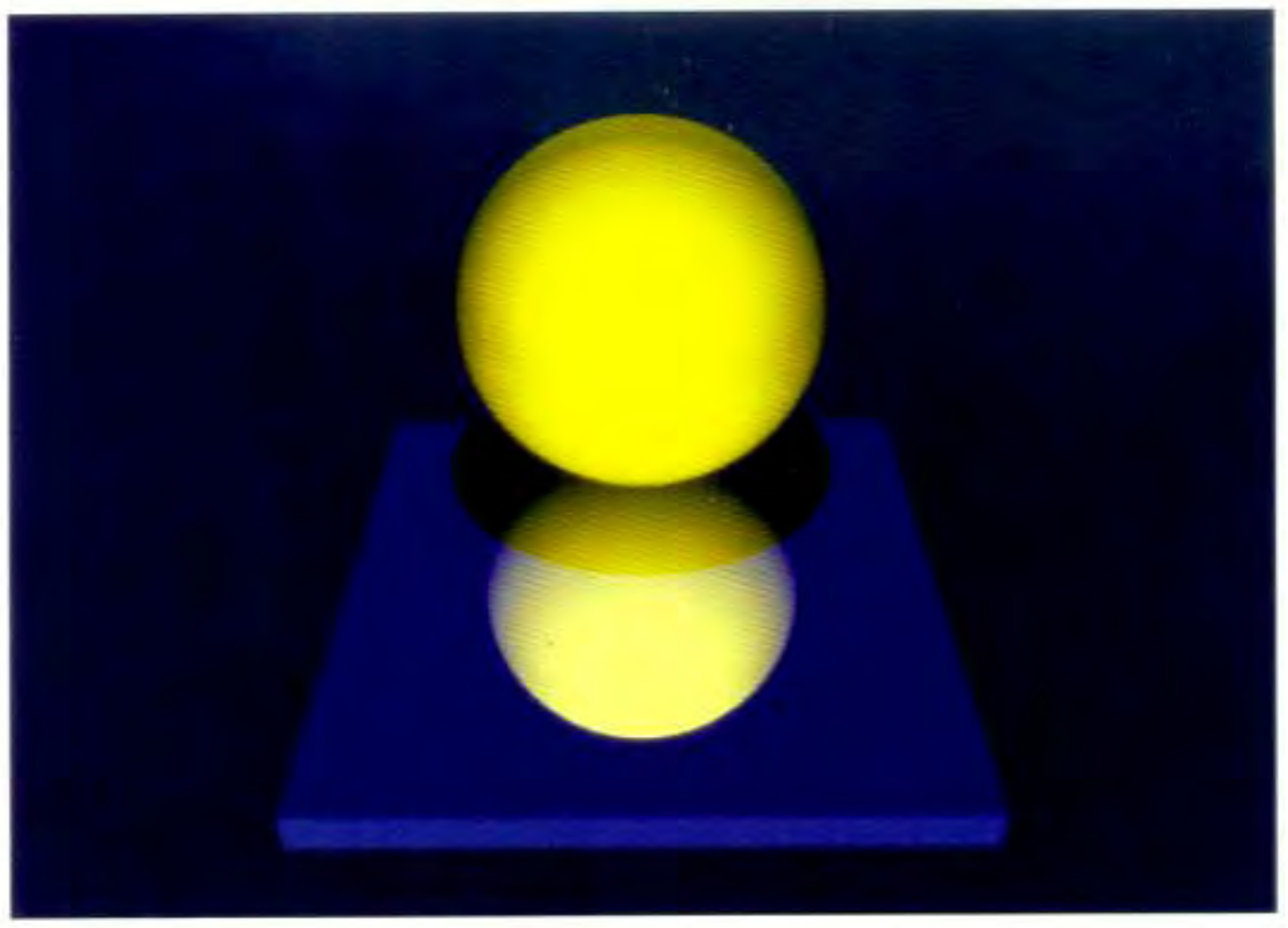
彩图 5：ATI 的视频卡

（版权归 *ATI* 所有）

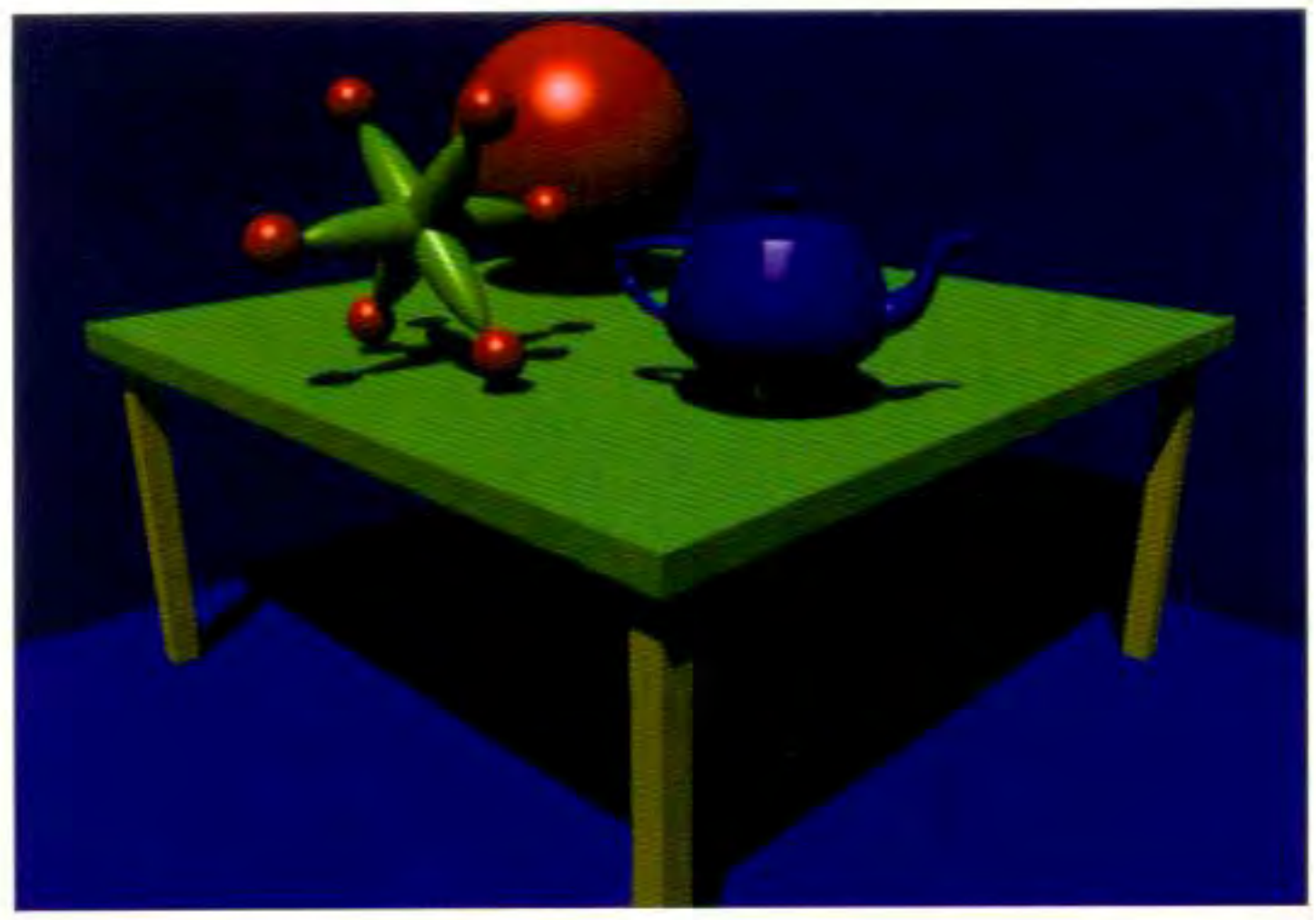


彩图 6：一个复杂的 OpenGL 菜单系统

（版权归 *Unigine* 所有）



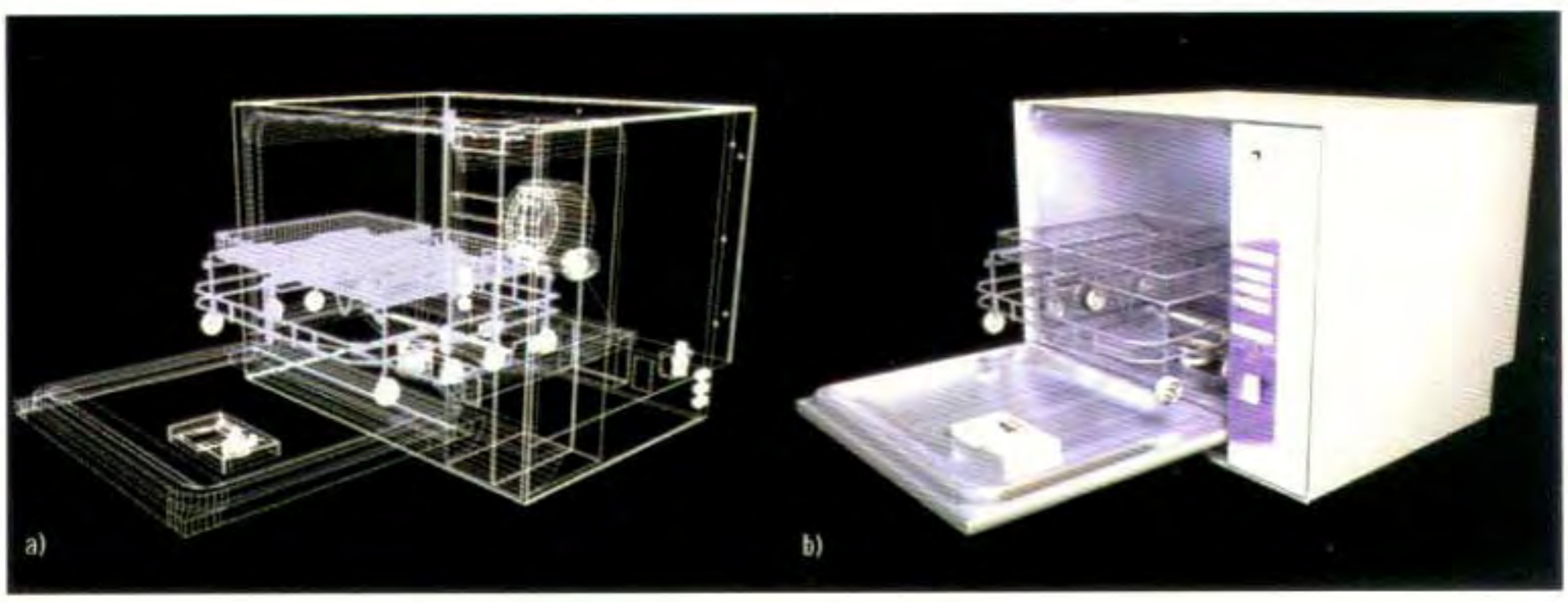
彩图 7：一个从有光泽表面反射的球
（版权归 *Sven Maerivoet* 所有）



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）

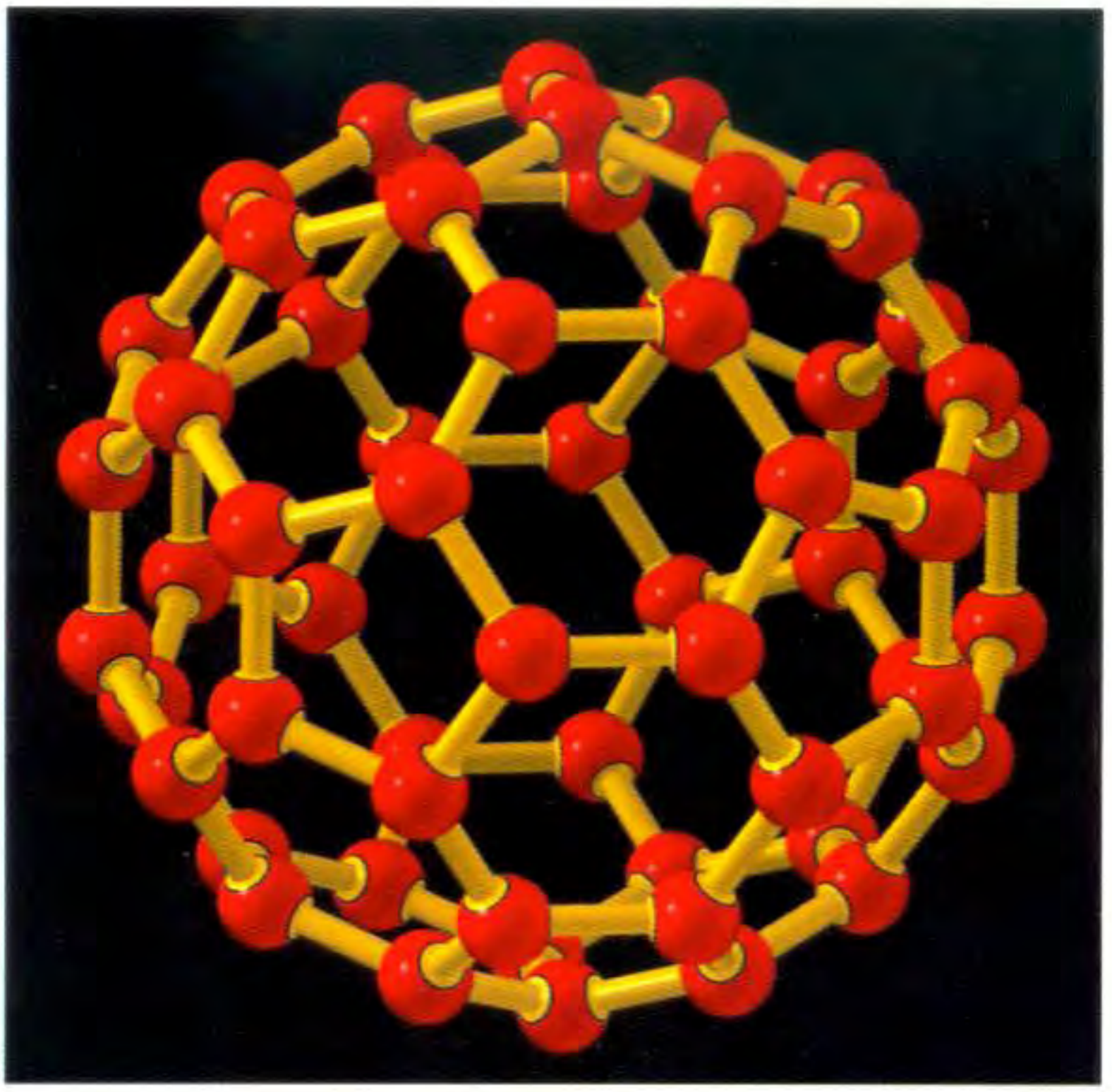
彩图 9：一个喷泉动画

（版权归 *Phillipp Crocoll* 所有），在线程序可用。



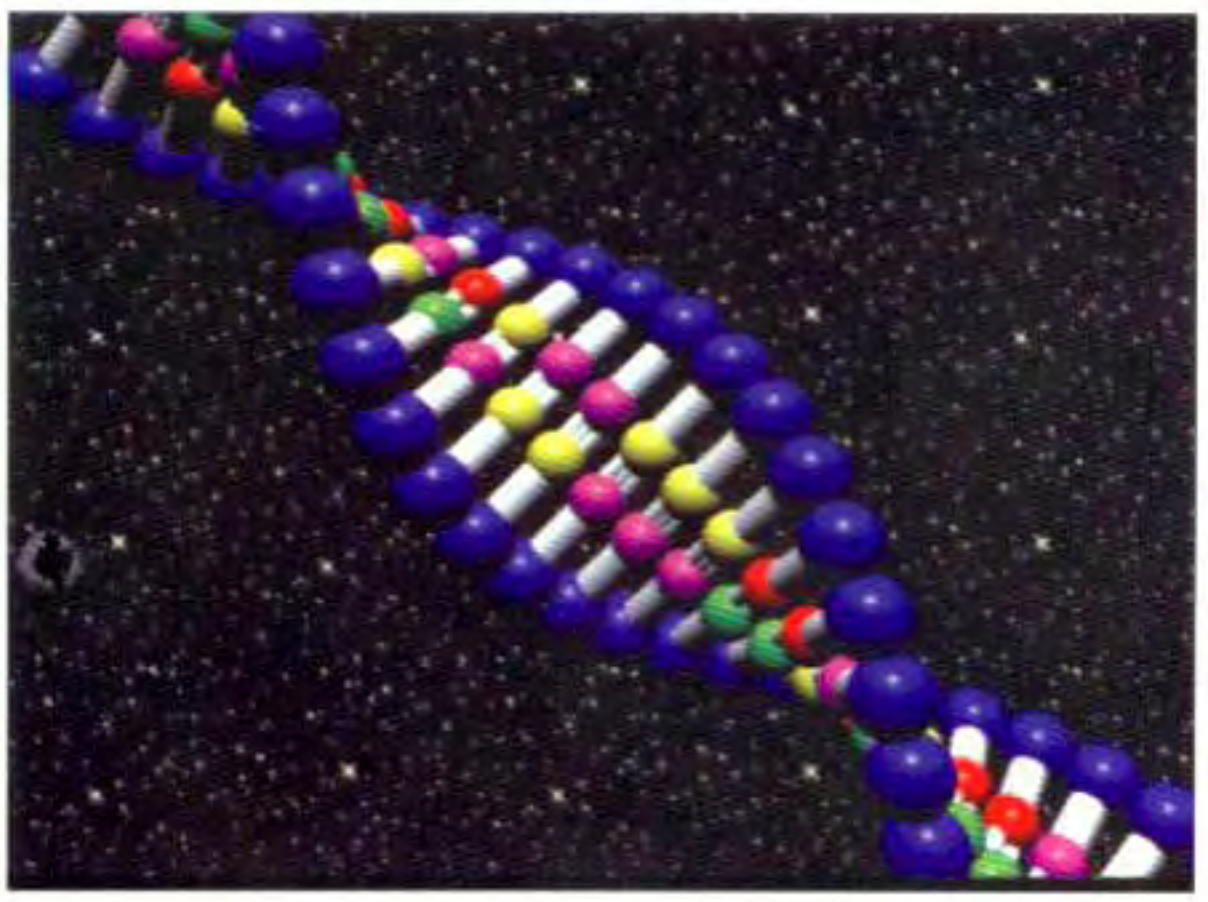
彩图 10：a）家具的线框模型；b）网格模型的渲染结果

（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

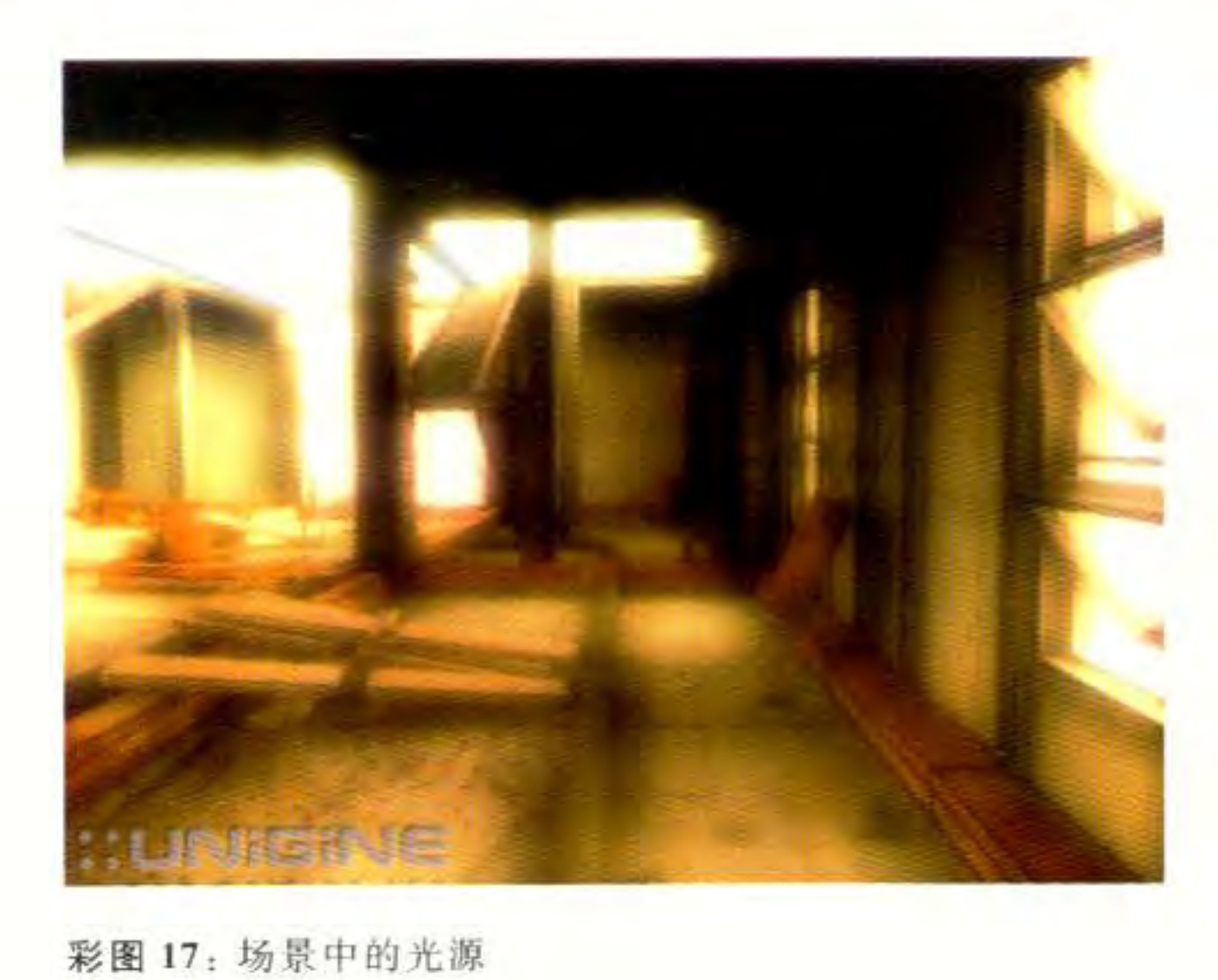
彩图 14：地形模拟图像（采用网格模型的山脉和粒子系统的云彩）
（版权归 *Rob Hall* 所有）

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）

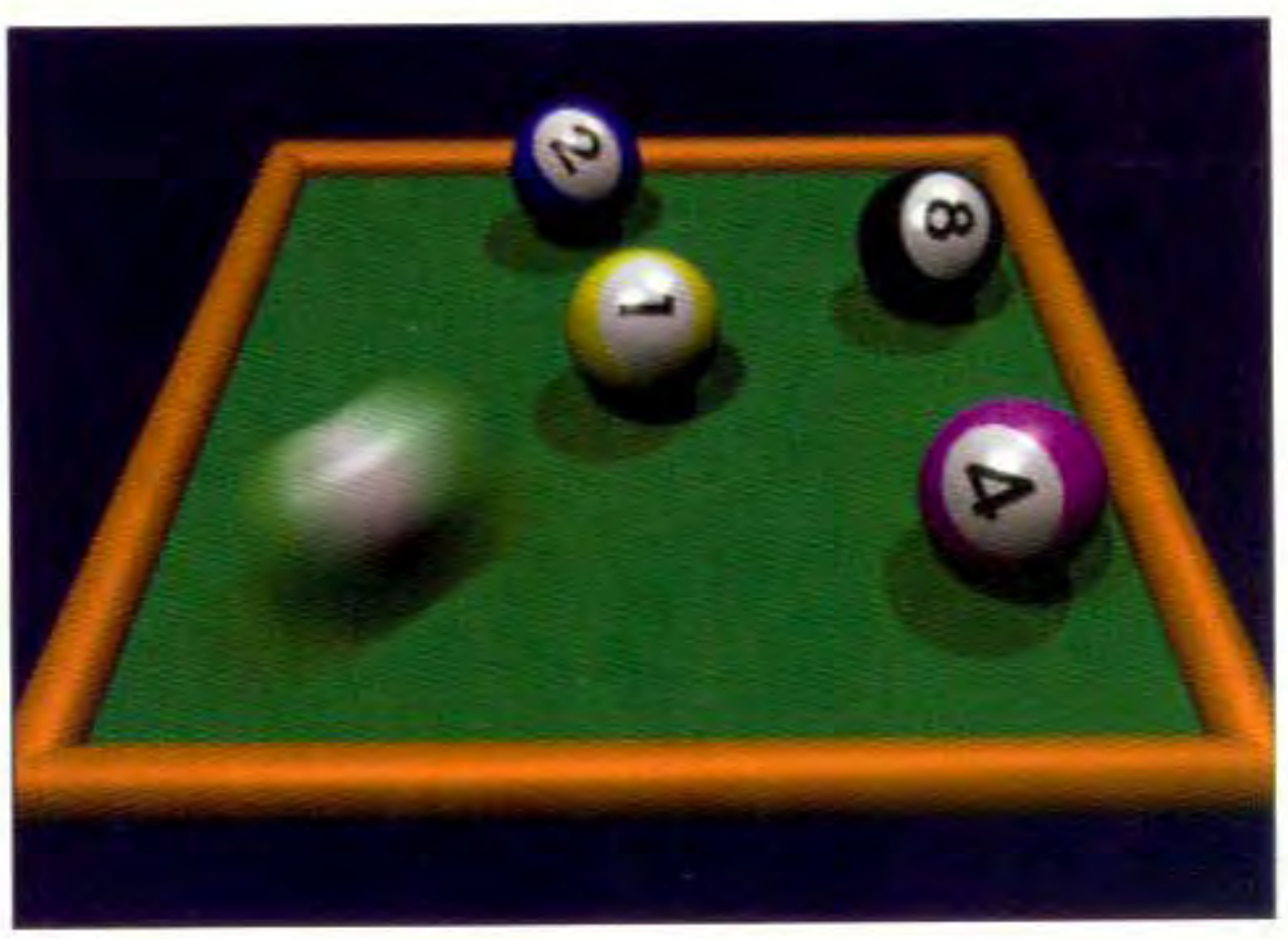


彩图 17：场景中的光源

（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

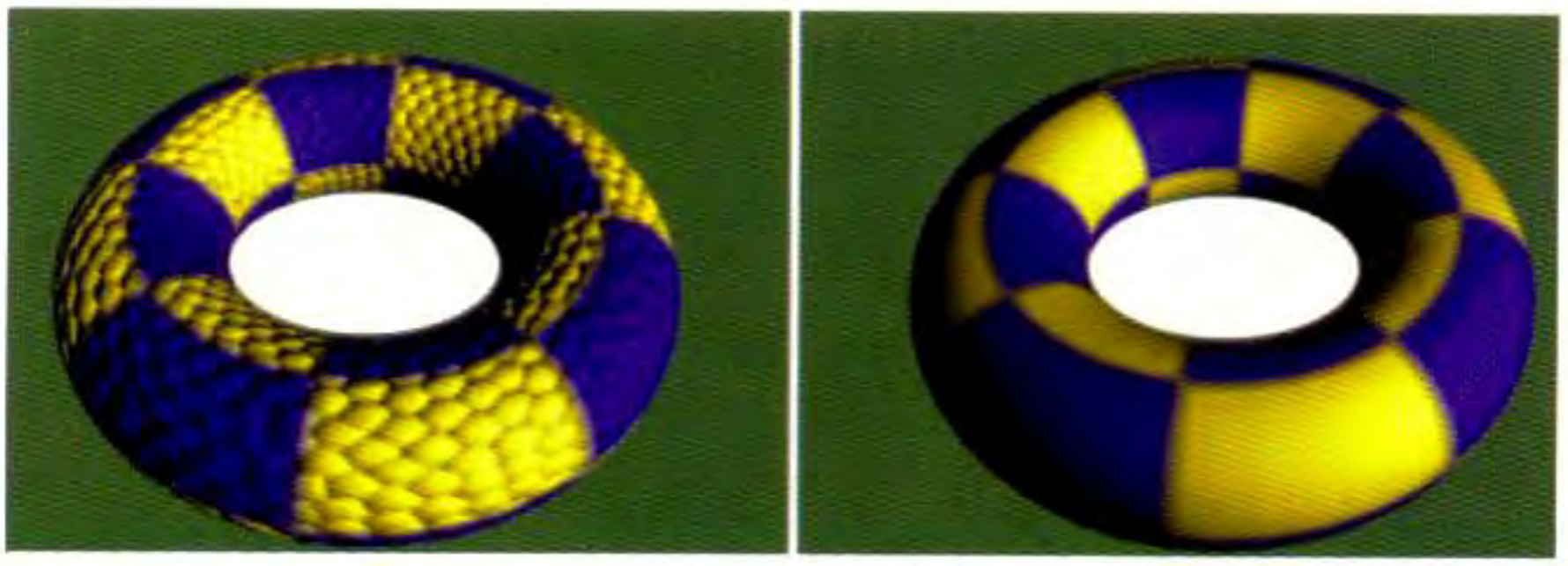
（版权归 *Sven Maerivoet* 所有）

彩图 20：铬环境图

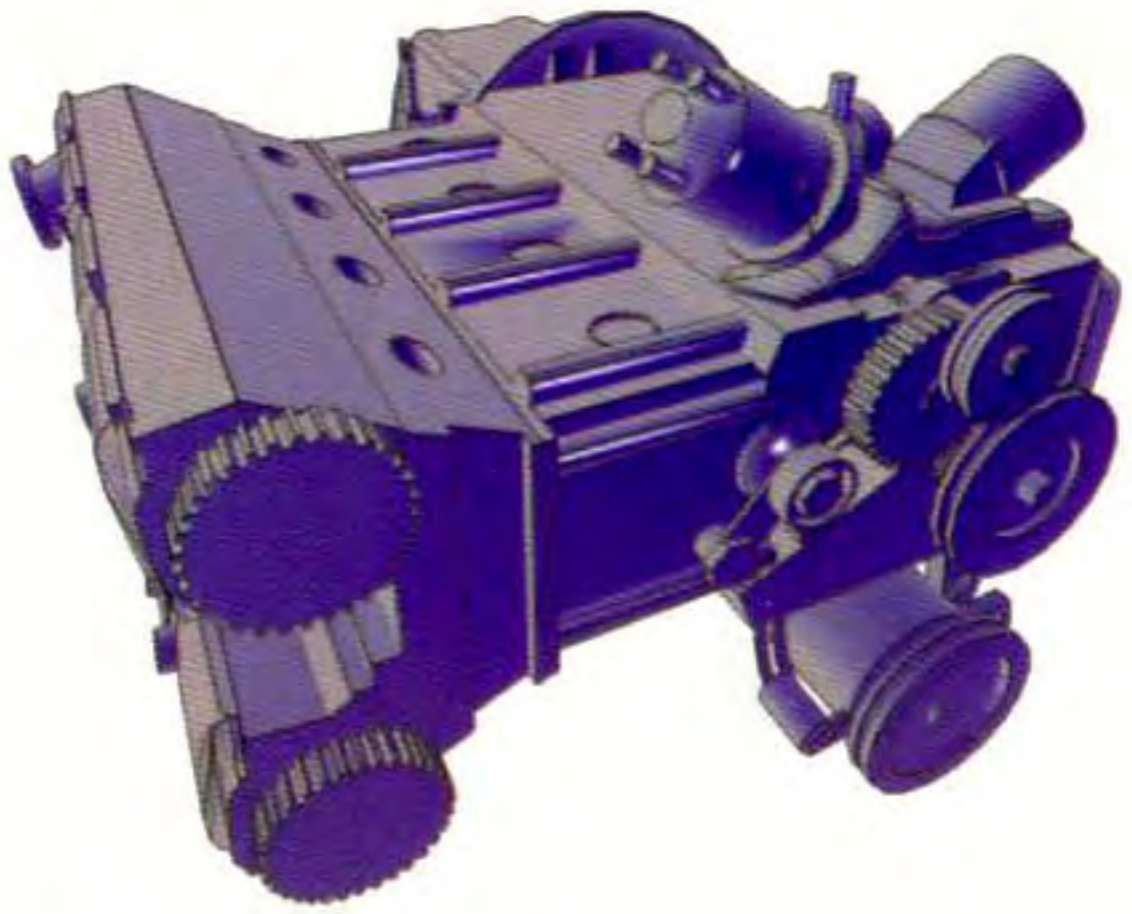
（版权归 *Eric Bpissard* 所有）

彩图 21：铬映射的例子

（版权归 *L. Latta* 和 *P. Debevec* 所有）

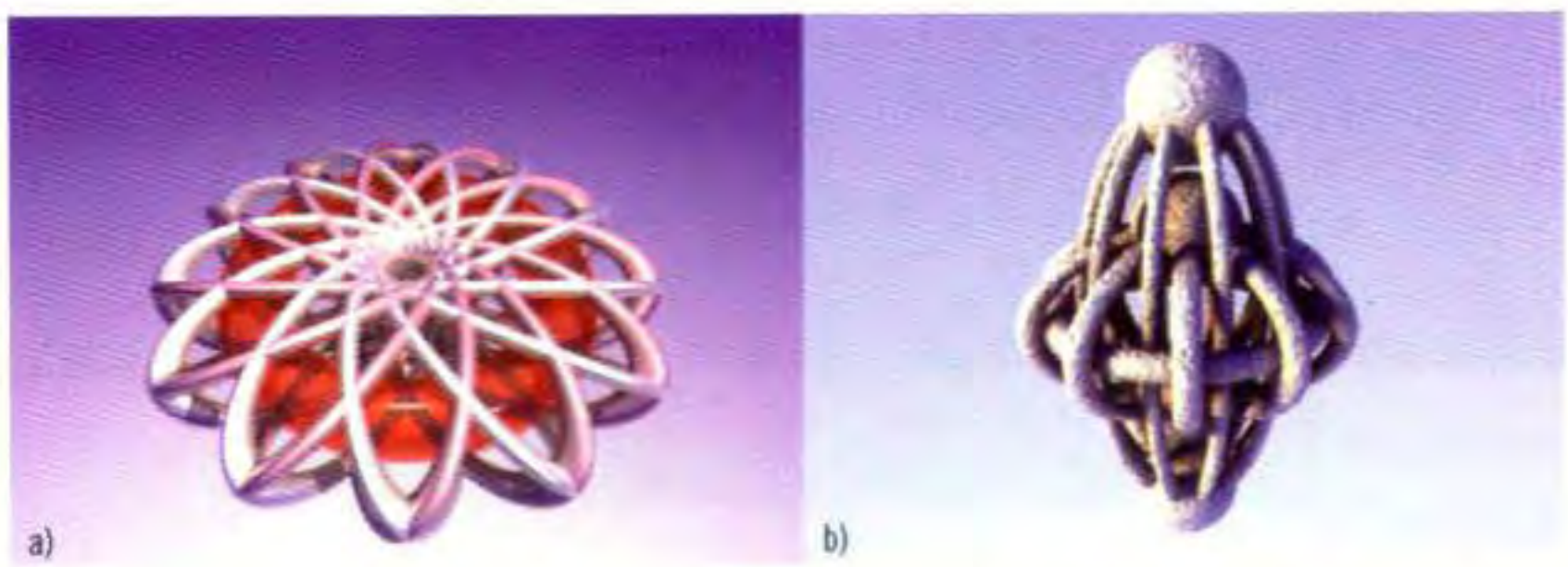


彩图 22：凹凸贴图的例子（左边有凹凸贴图，右边没有）



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）

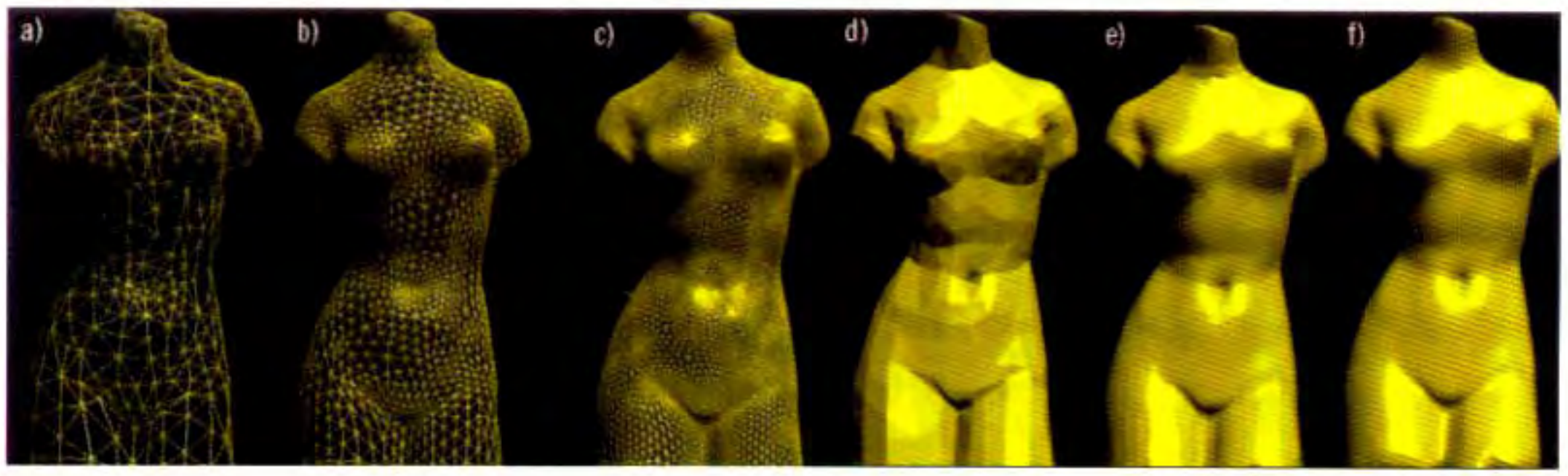
彩图 24：非真实感绘制
（版权归 *Kontantinos Sakelllis* 所有）



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）

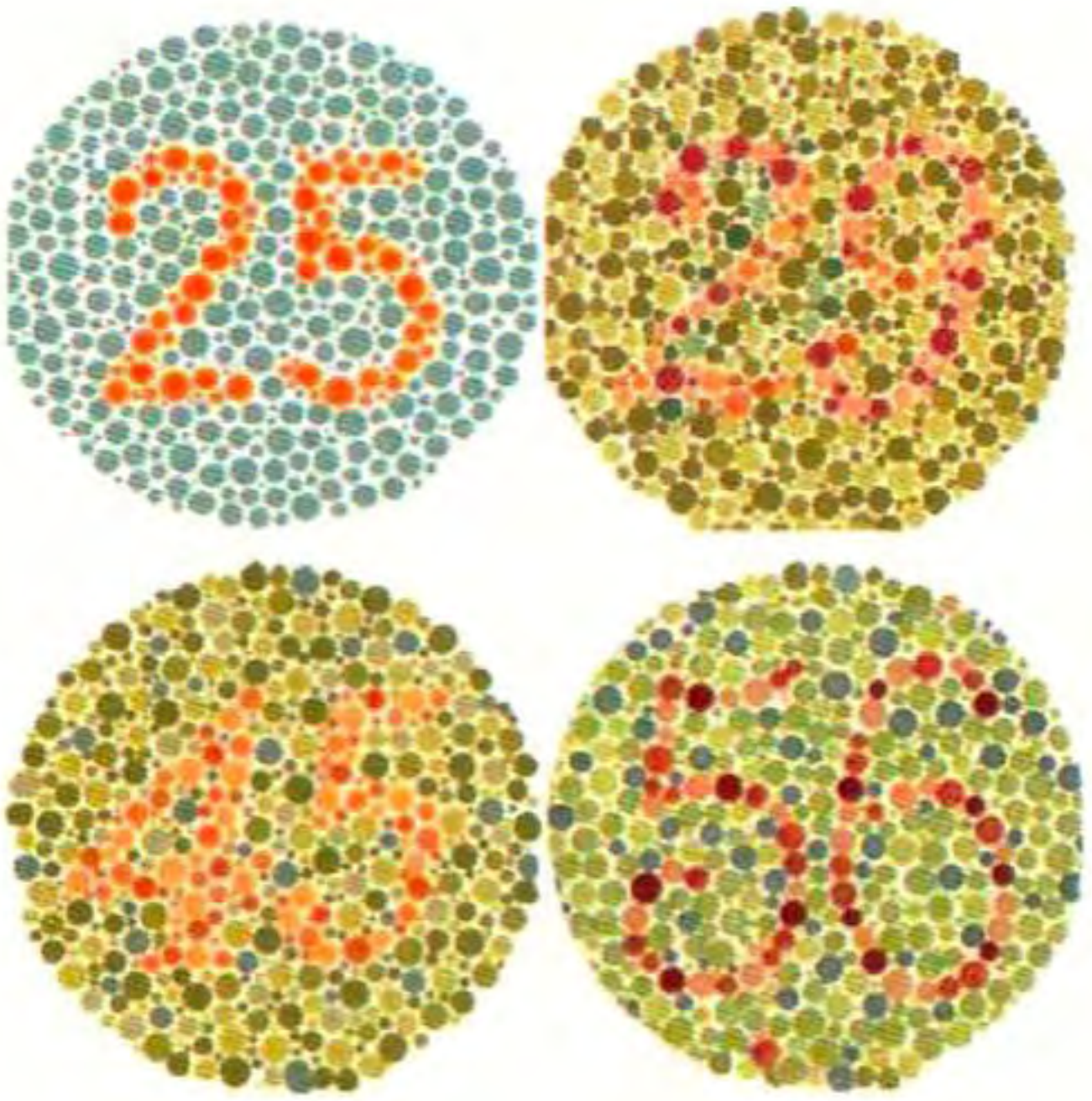
彩图 26：被随机弄乱了的球体模型

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



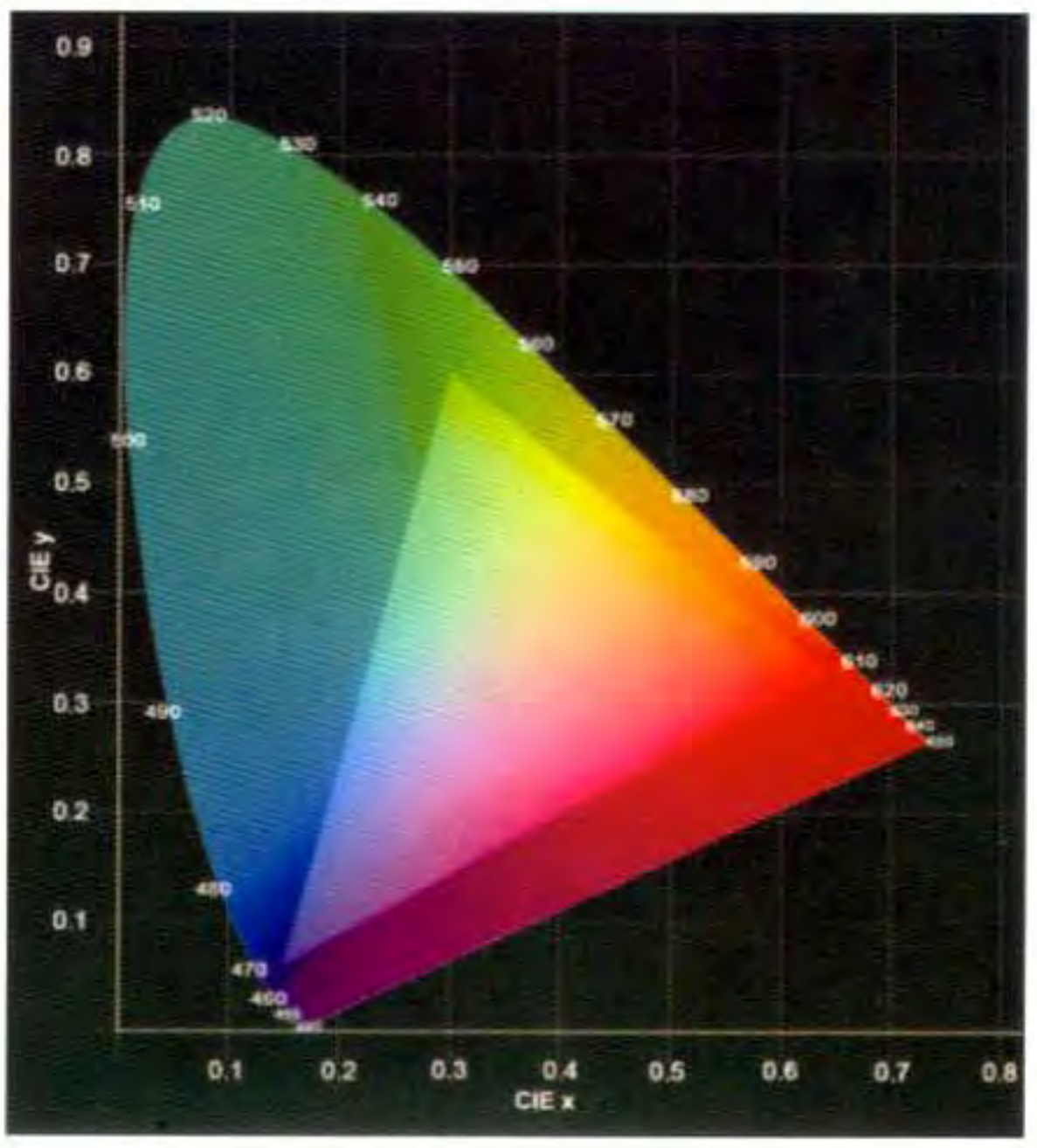
彩图 27：网格模型的连续细分。a)～c)为线框视图；d)～f)为明暗渲染视图

（版权归 *Jian Zhang* 所有）



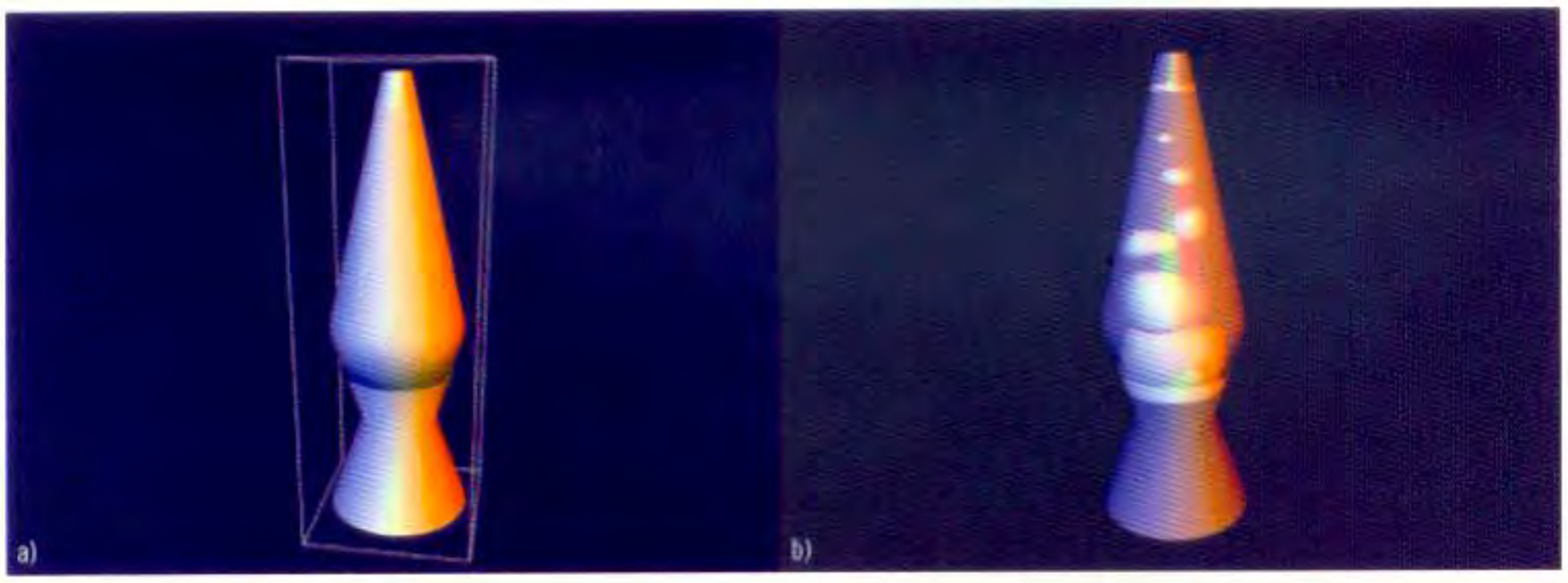
彩图 28：色音测试图

（版权归 *Terrance L. Waggoner* 所有）

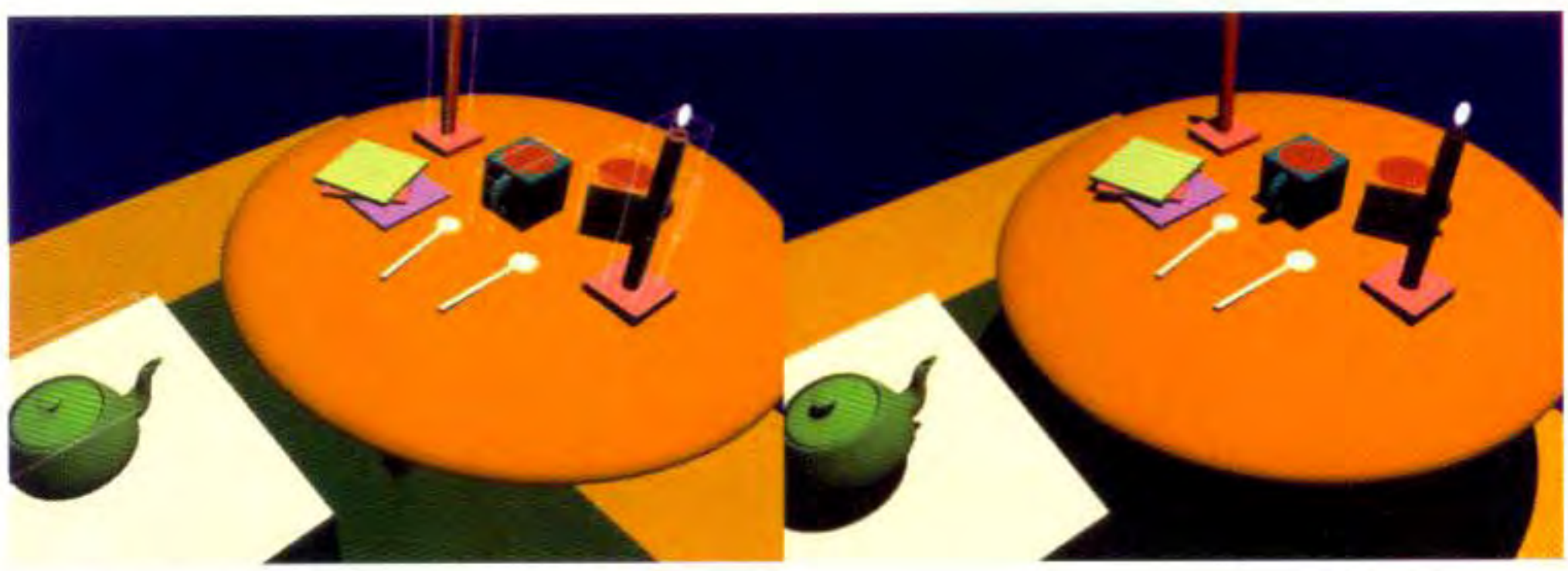


彩图 29：CIE 色度图

（版权归 *TechMind* 所有）

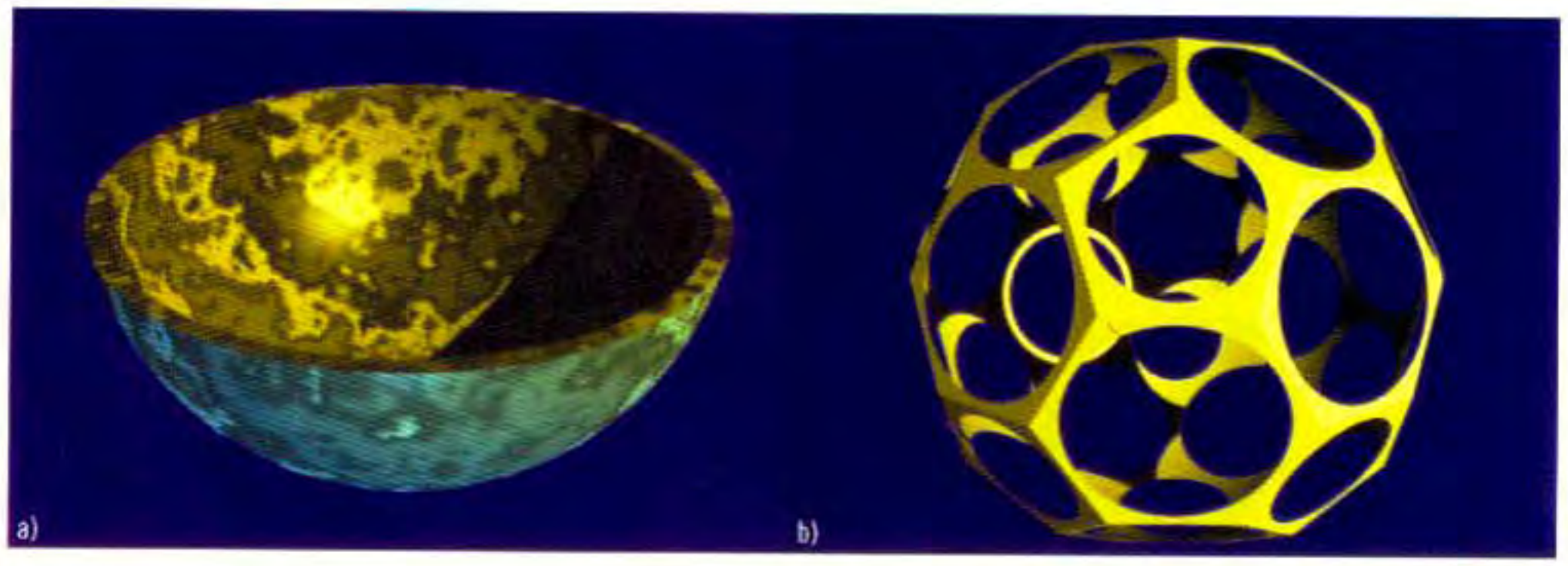


彩图 30：a）带包围盒的蜡灯网格模型（OpenGL 绘制）；b）左图模型的光线跟踪绘制结果（版权归 *Sven Maerivoet* 所有）



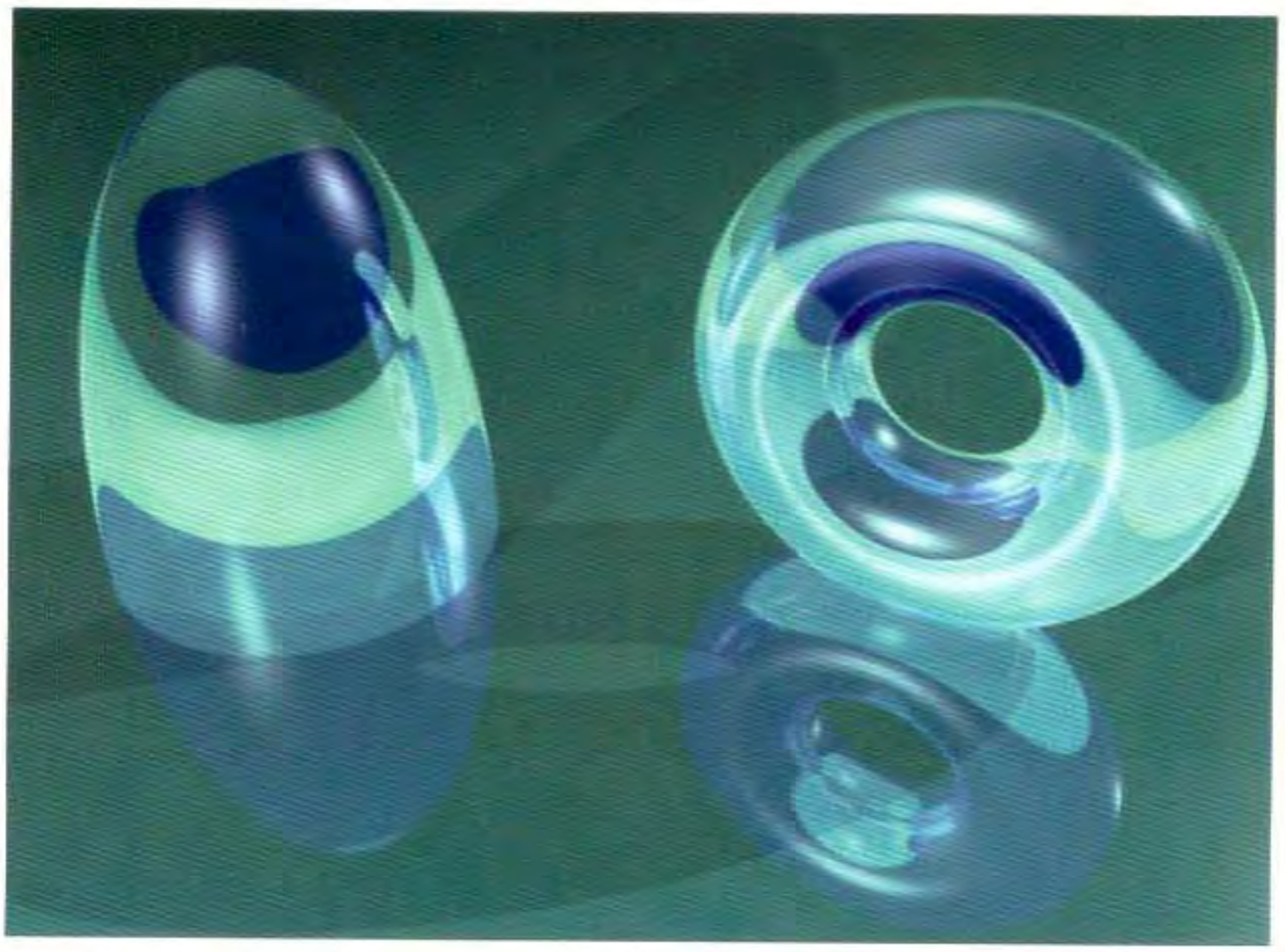
彩图 31：光线跟踪绘制以展示阴影的场景（版权归 *Sven Maerivoet* 所有）

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）



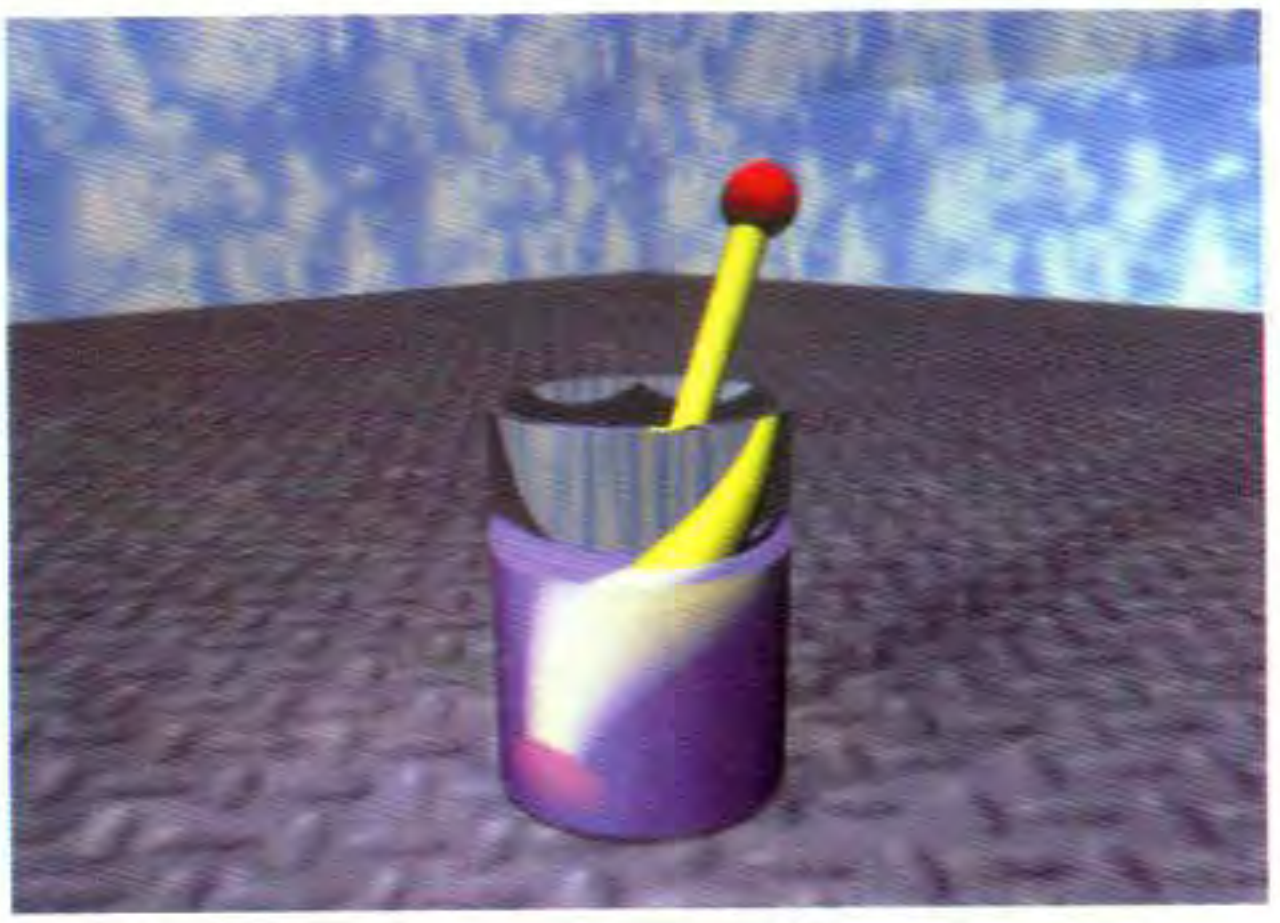
彩图 33：a）光线跟踪绘制的 CSG 物体；b）光线跟踪绘制的 CSG 物体，该物体为由巴克球截去的球体

（版权归 *Sven Maerivoet* 所有）



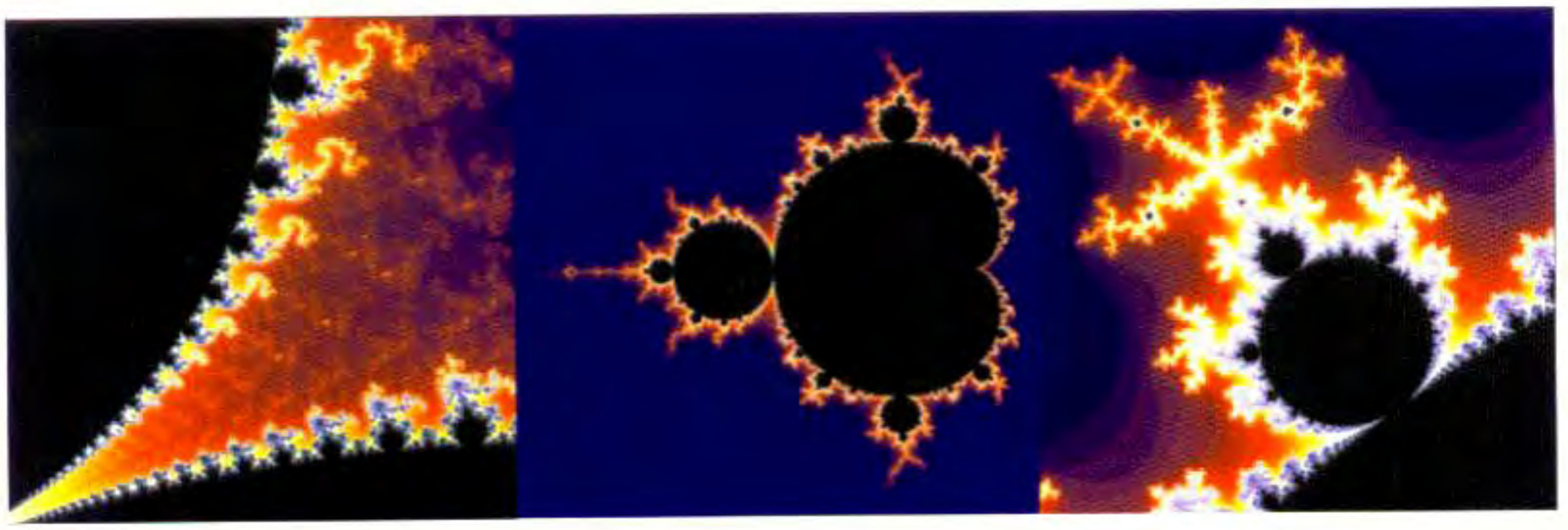
彩图 34：带反射、折射和阴影效果的两个玻璃物体

（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。

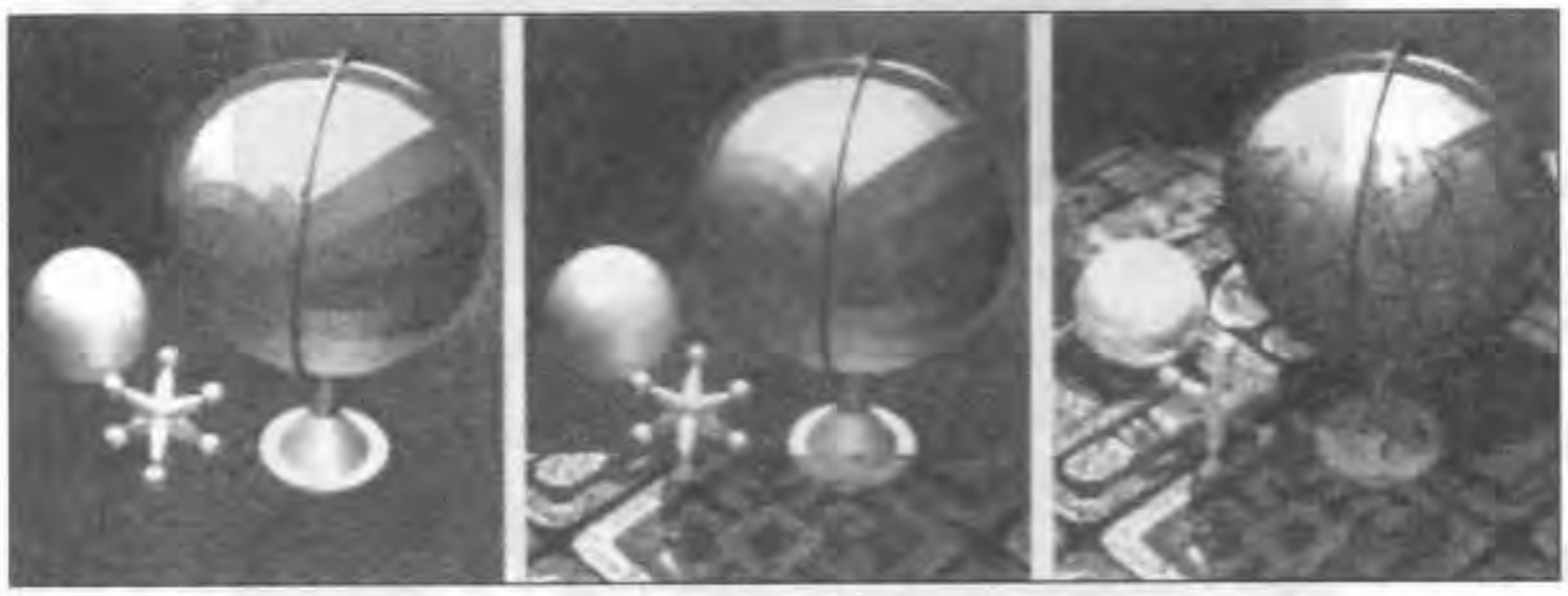


图 1.6 用于教学演示的幻灯片

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。

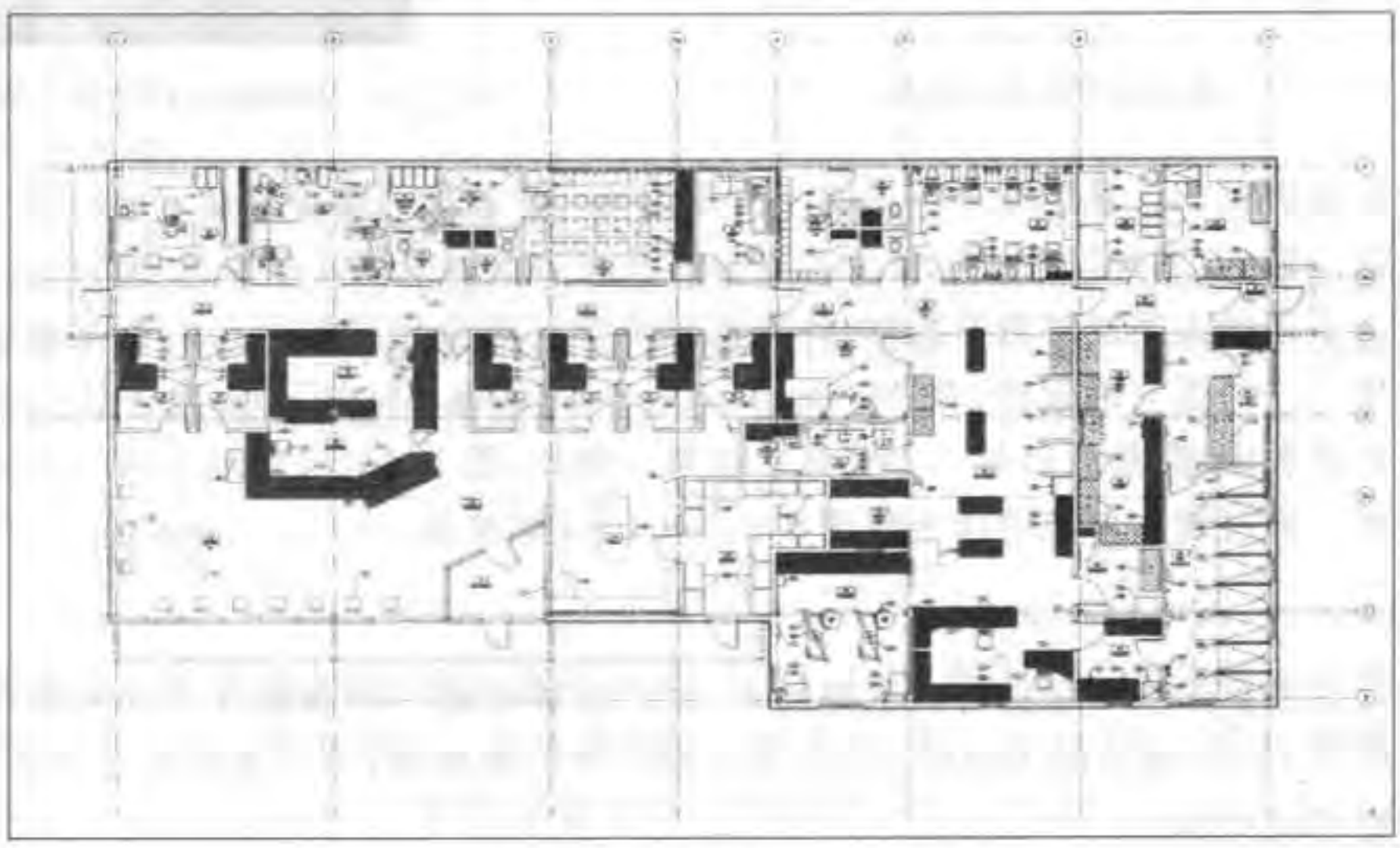


图 1.10　一个医院的平面图

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。

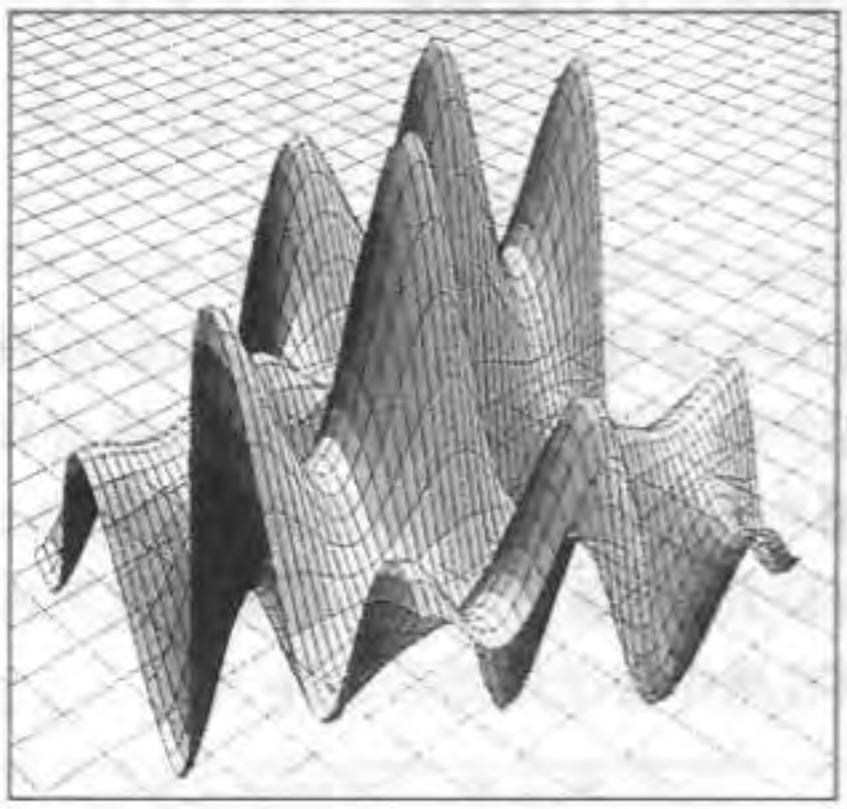


图 1.11 复杂科学数据的显示

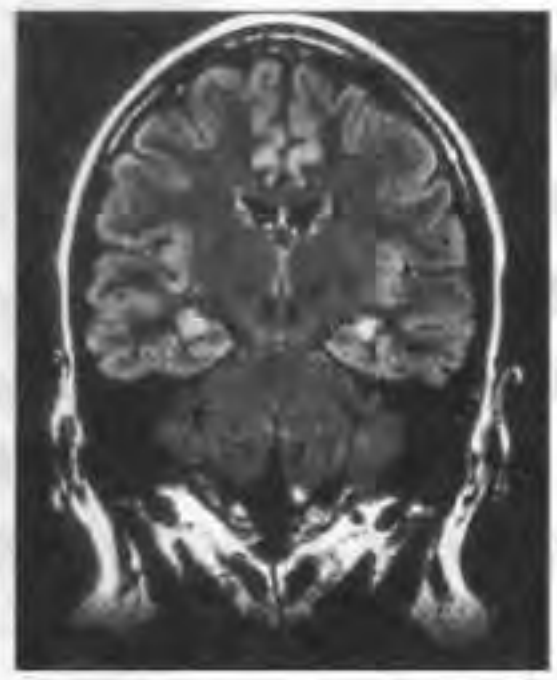


图 1.12 利用体可视化的人脑视图

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。

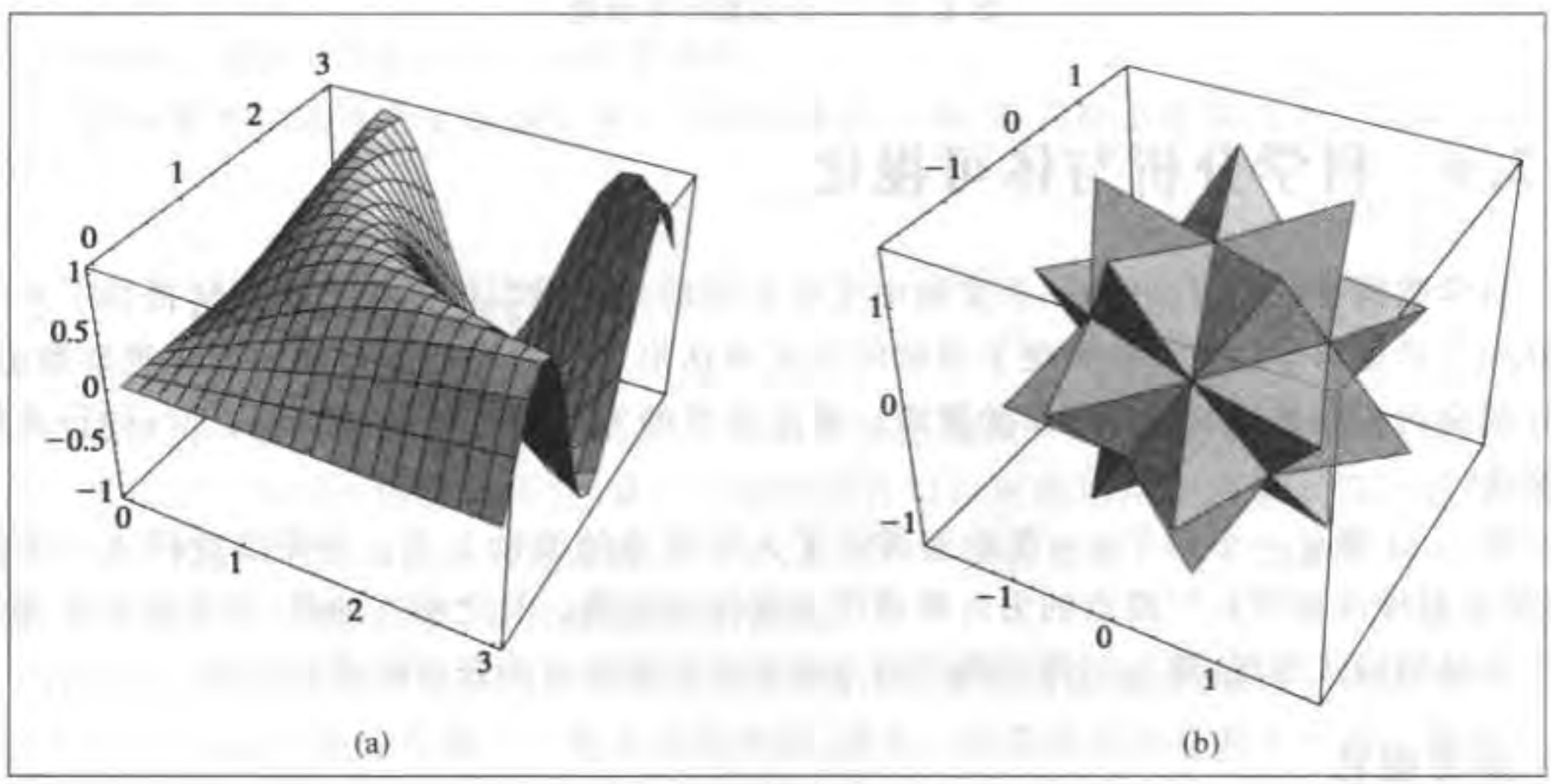


图 1.13 Mathematica 的显示

(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。

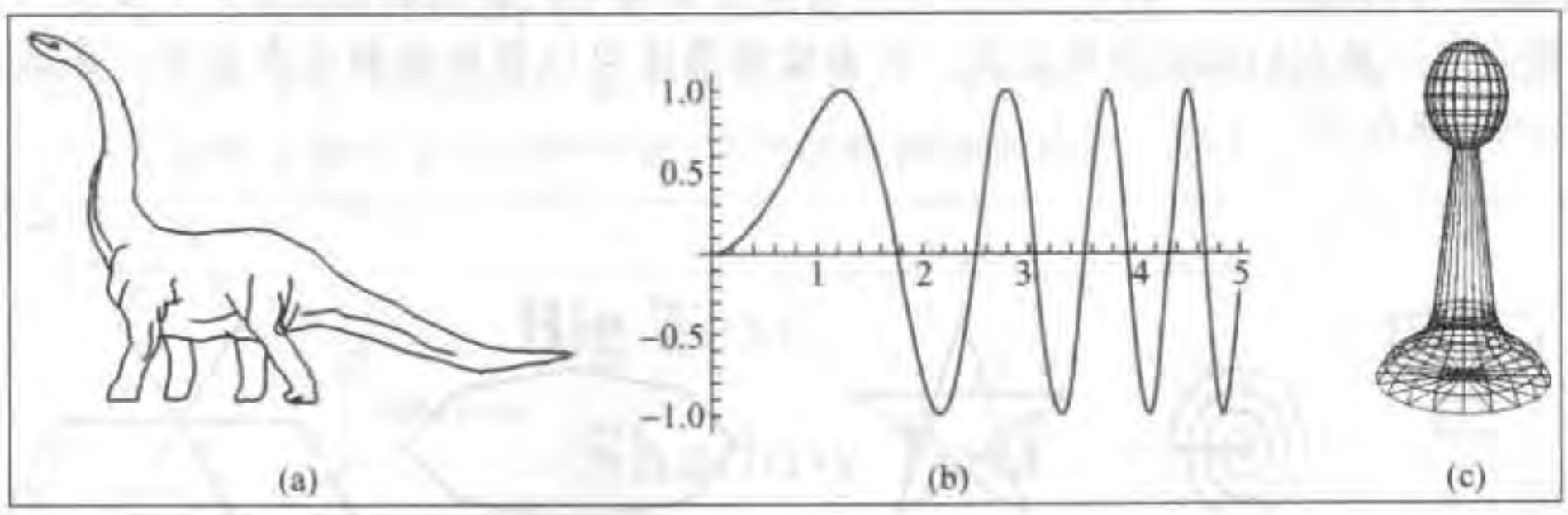


图　1.14

(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。

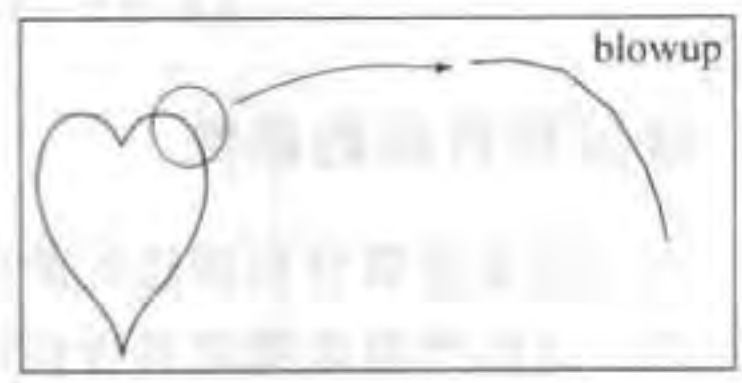


图 1.15　一个由直线段构成的曲线

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);
```

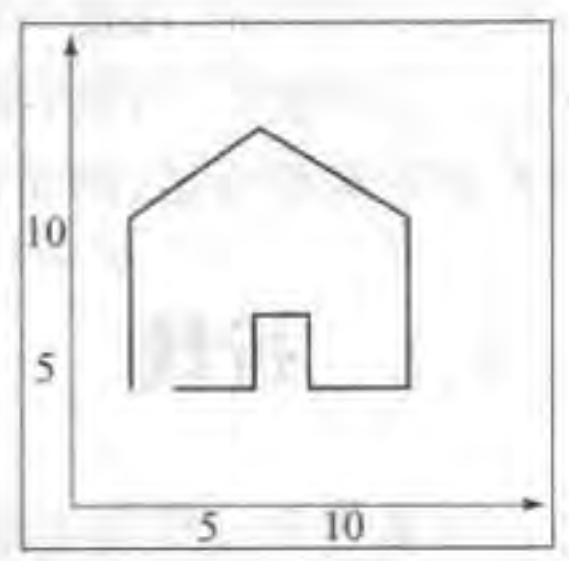


图 1.16 一个折线的例子

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。

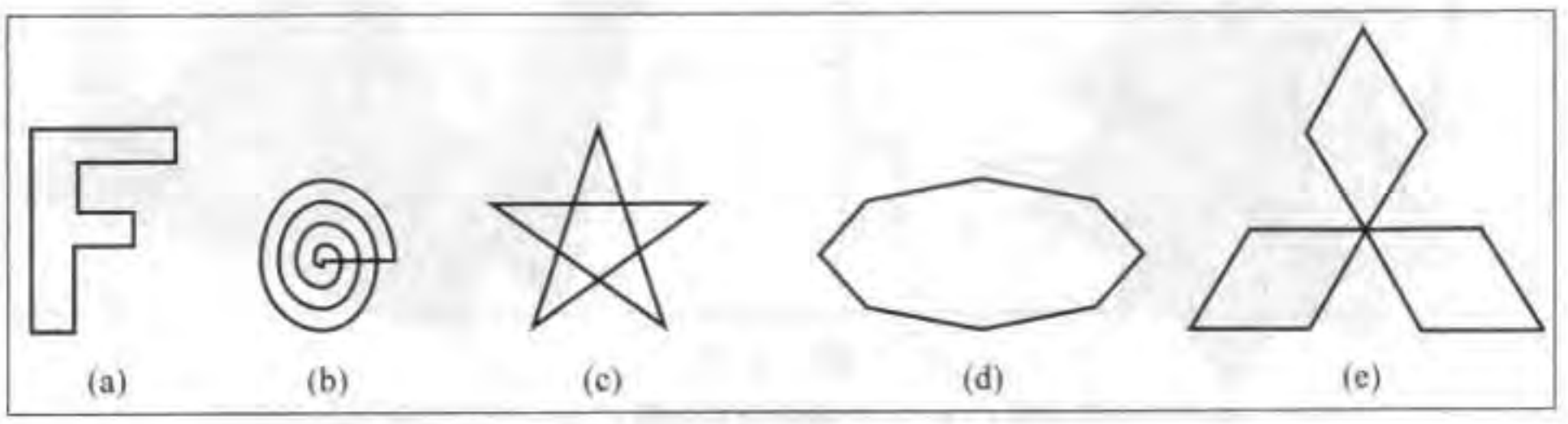


图 1.17 多边形的例子

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。

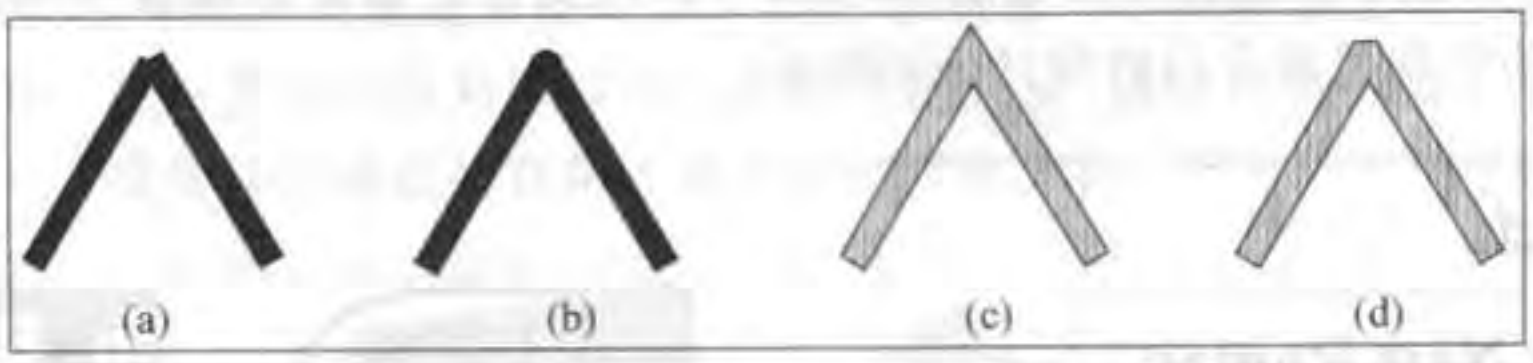


图 1.19　在一条折线中两条粗线段的连接方式

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。

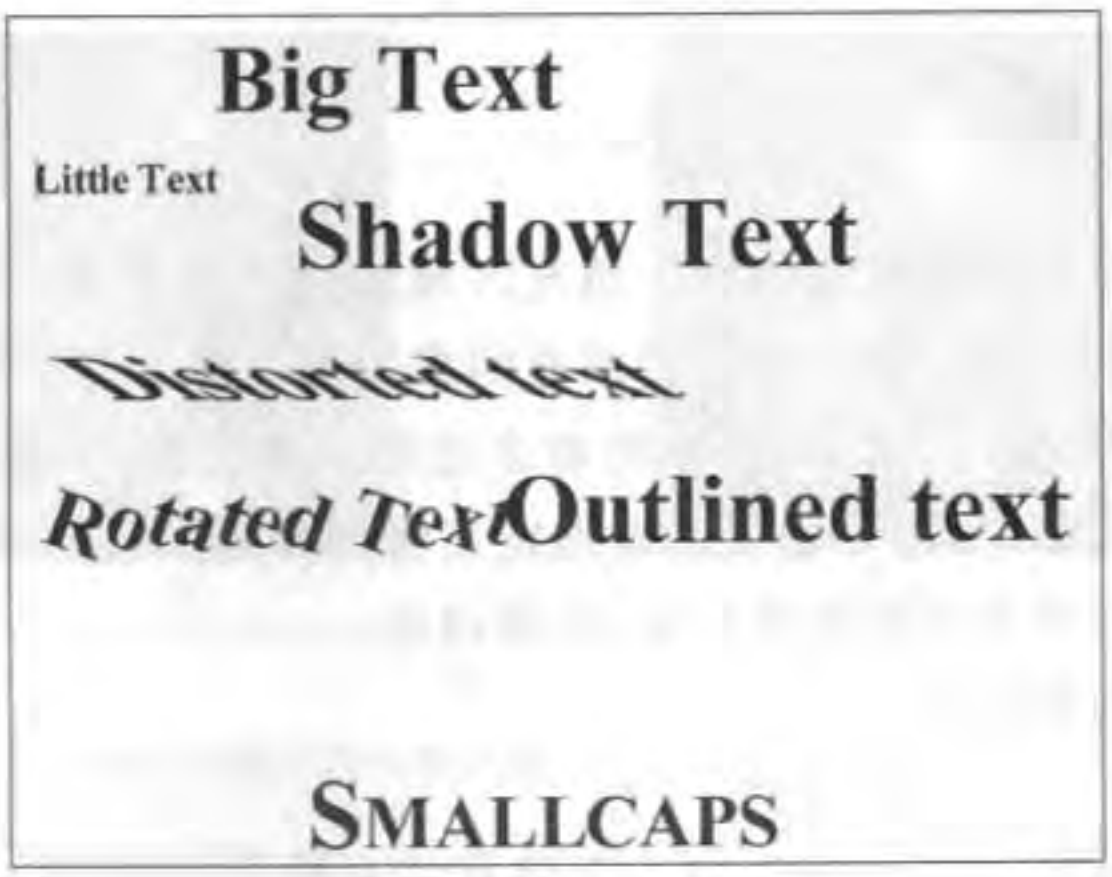


图 1.20　图形模式绘制的文本

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。

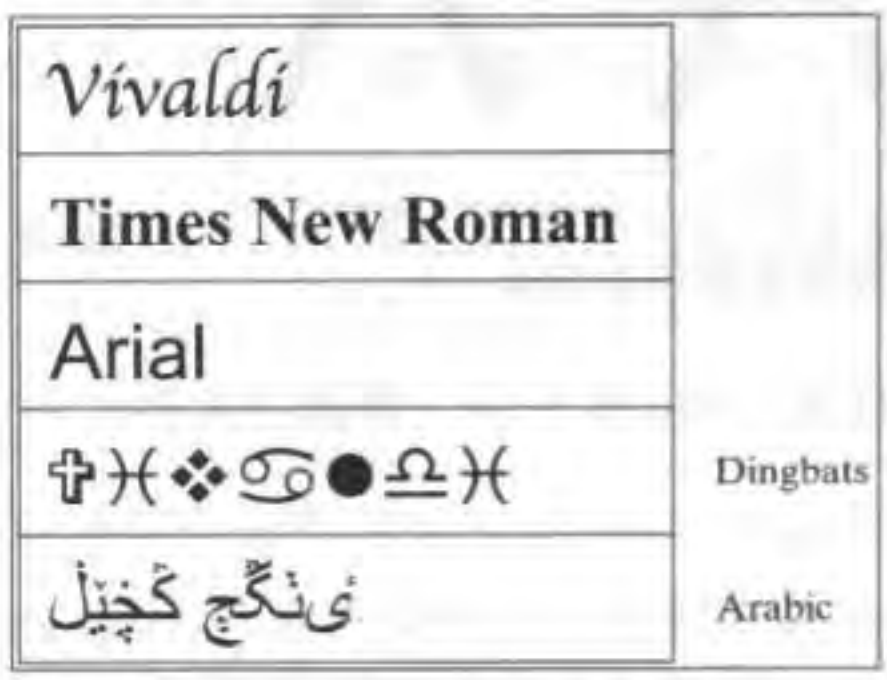


图 1.21 一些字体的例子

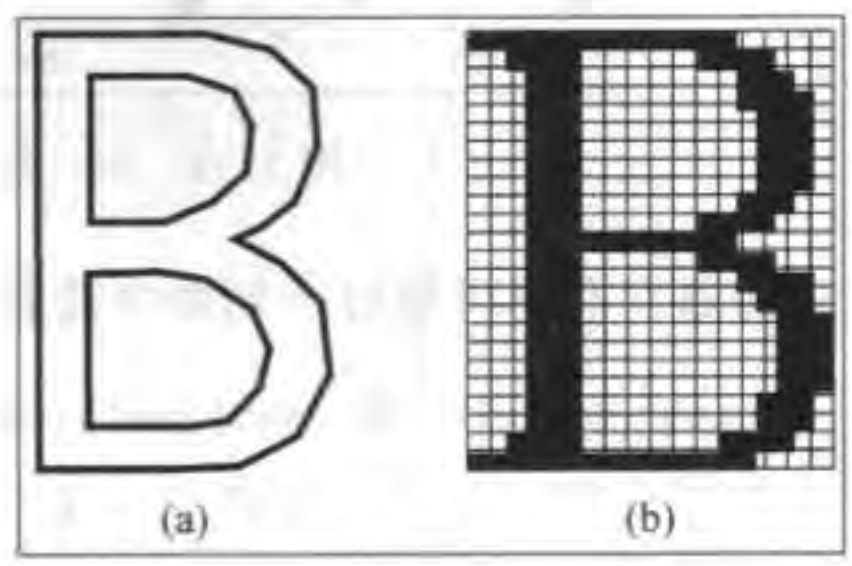


图 1.22 字符形状通过(a)折线和通过(b)点阵模式来表示

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

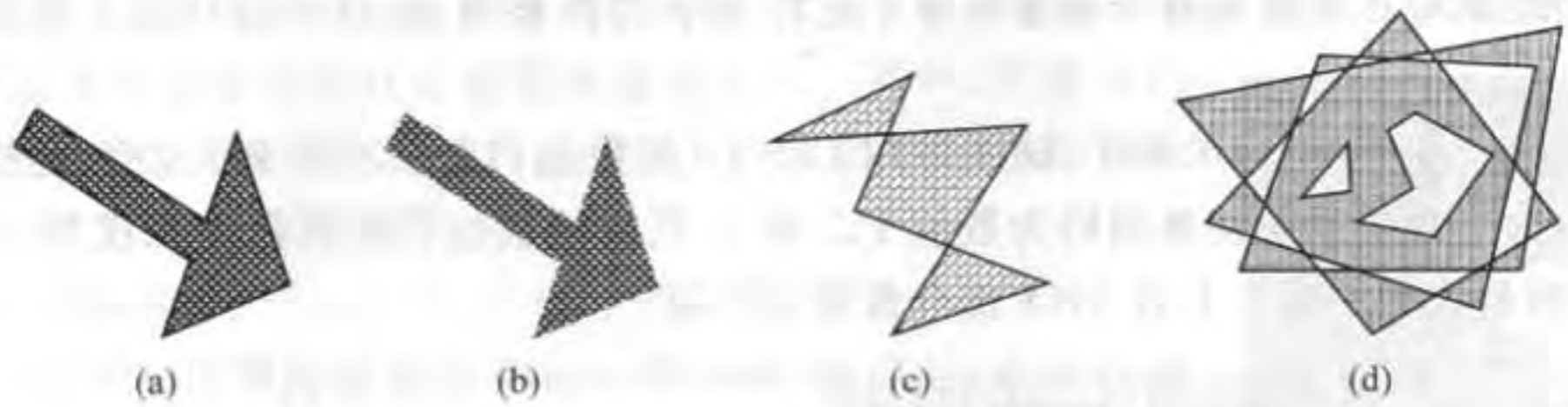


图 1.23　填充的多边形

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。

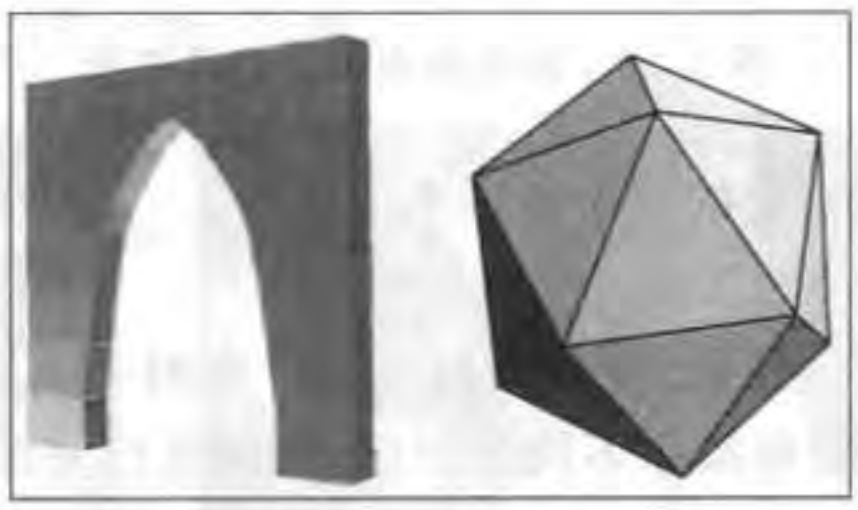


图 1.24　三维物体的多边形(面)填充以显示正确的着色

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。

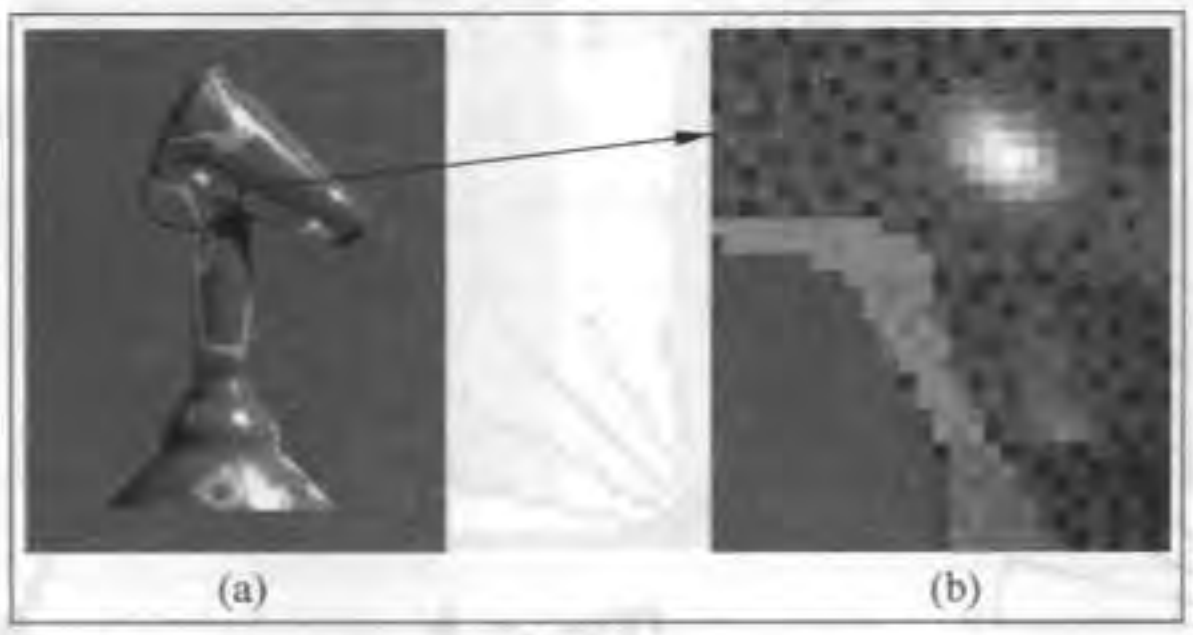


图　1.25

(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。

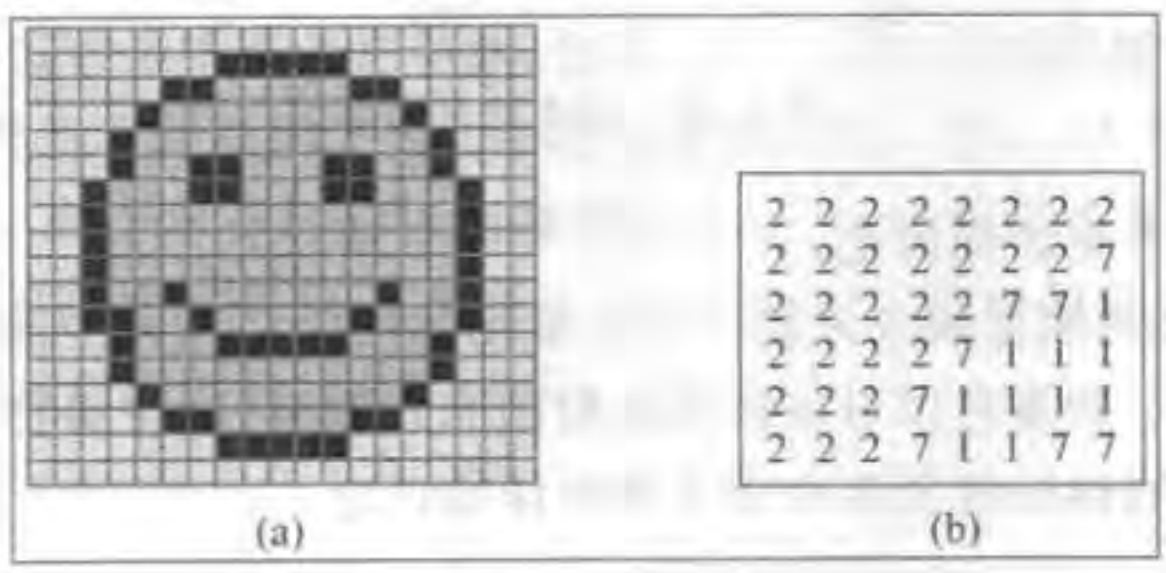


图 1.26 用位图表示的简单图形

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

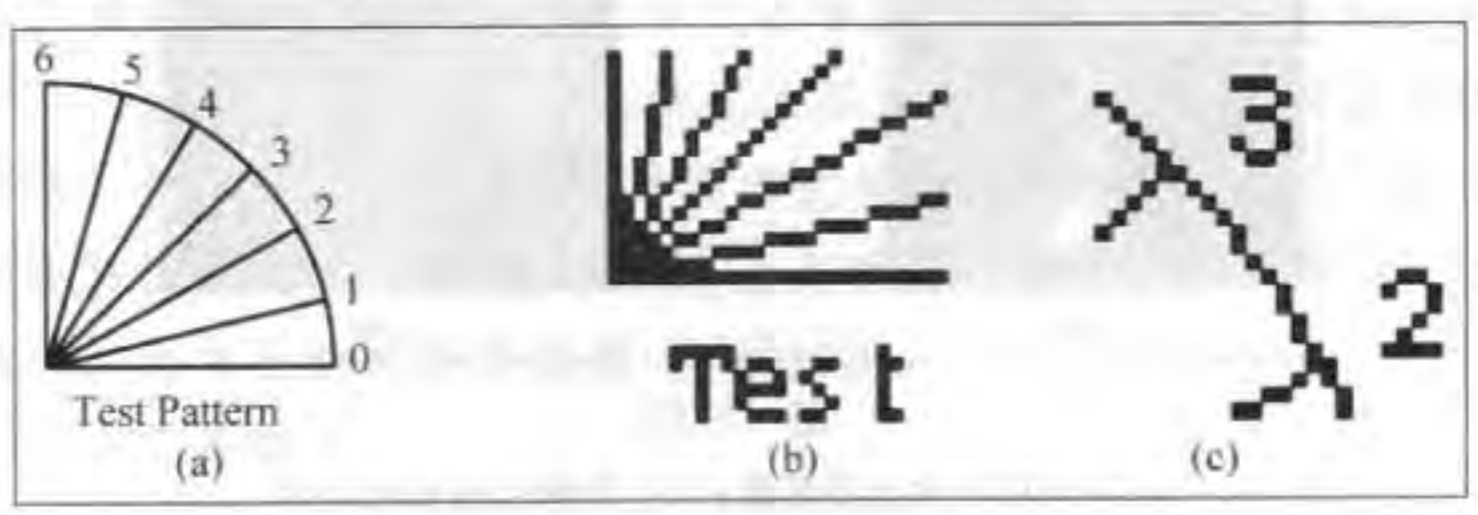


图 1.27

(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。

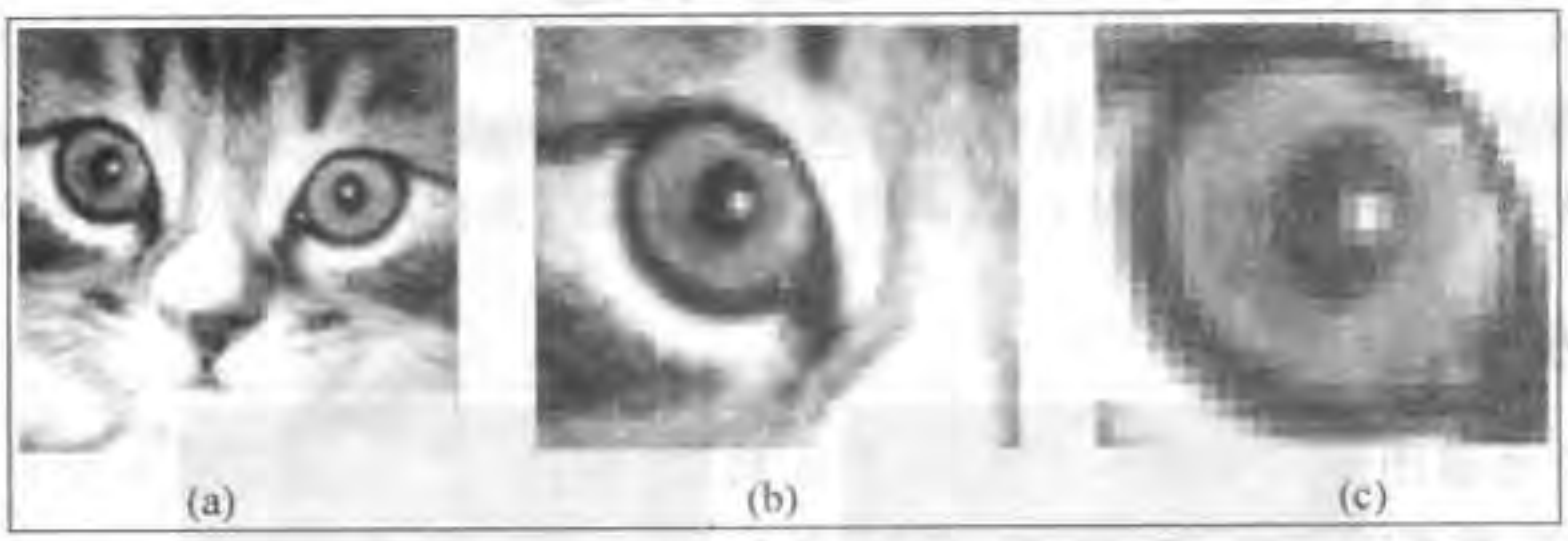


图 1.29　小猫图像的 3 幅连续放大图

(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。

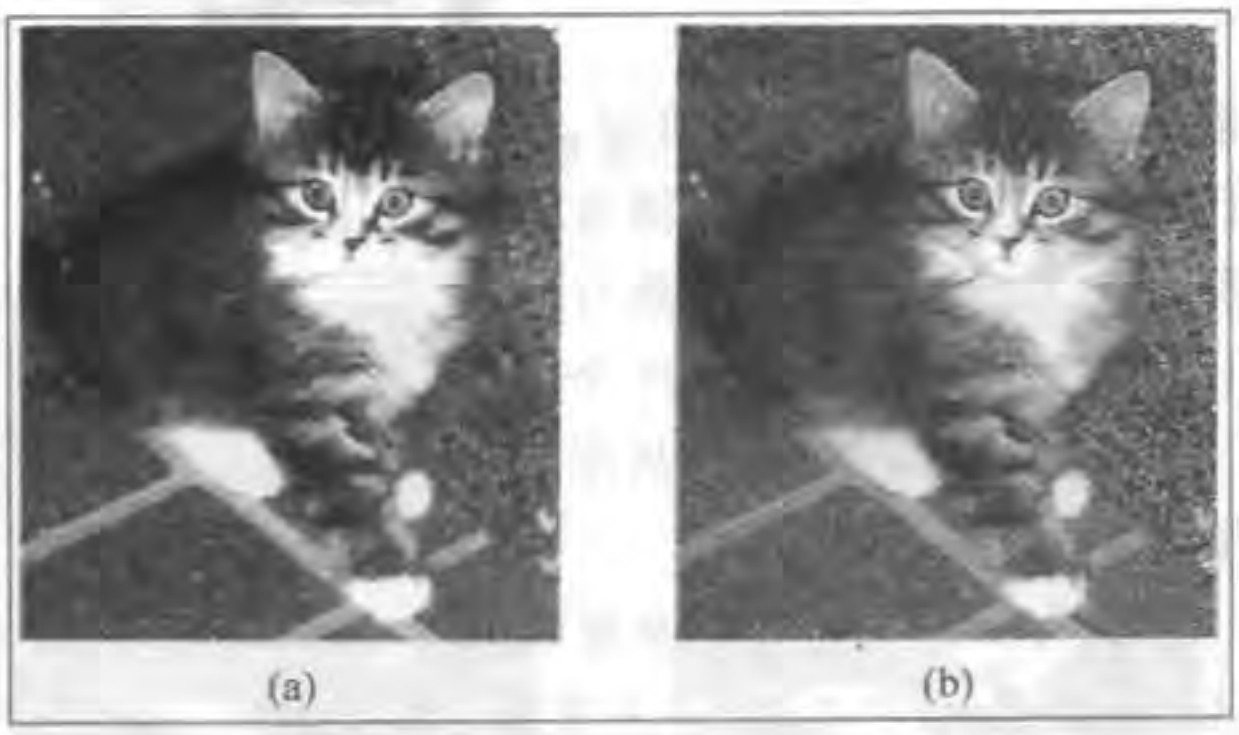


图 1.30　图像增强的例子

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。

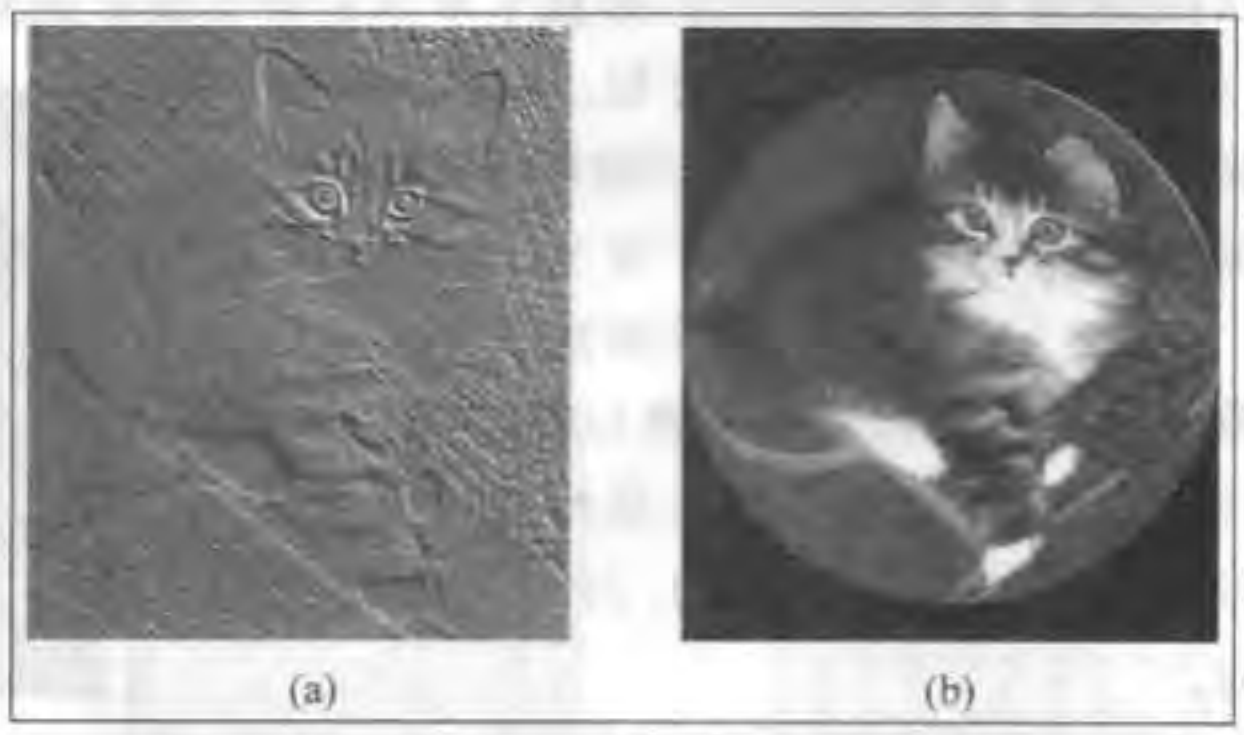


图 1.31
(a) 浮雕图像;(b) 几何变形的图像

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。

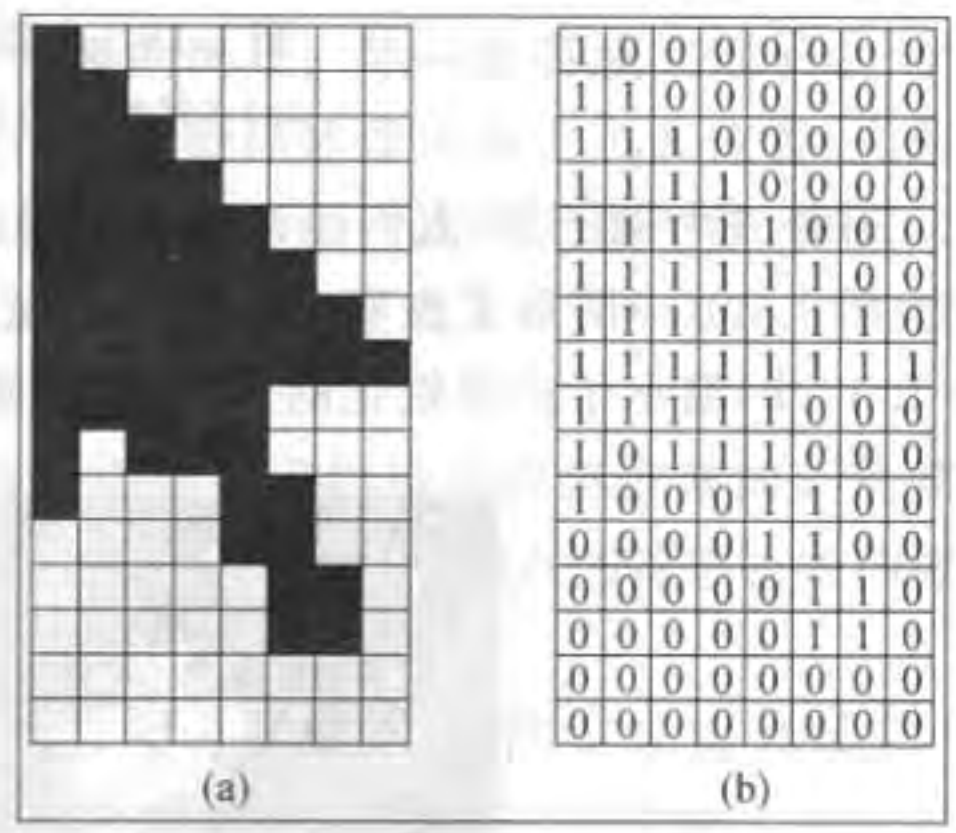


图 1.32
(a) 一个光标的二值图;(b) 它的位图

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。

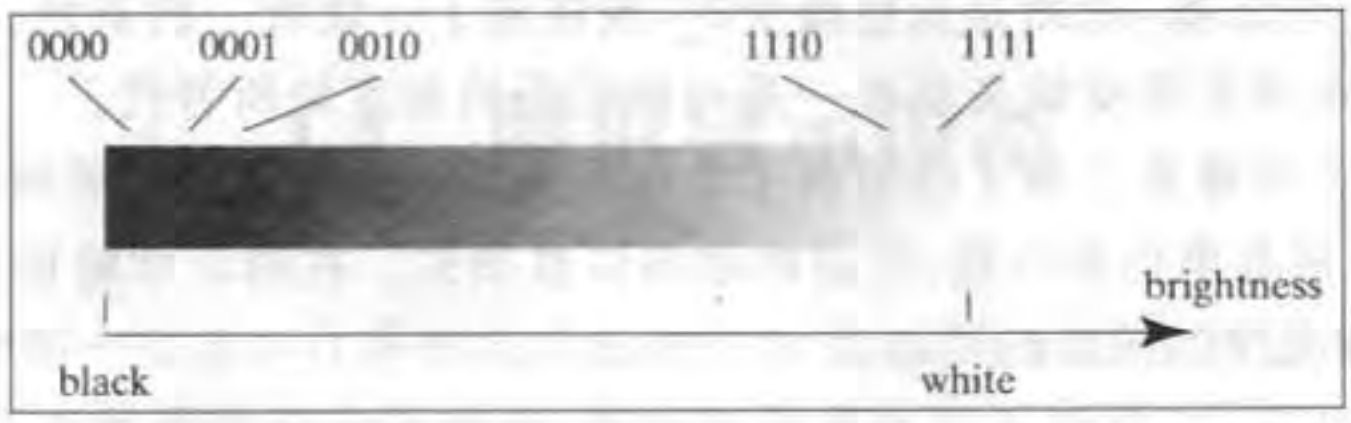


图 1.33　16 级灰度

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）。

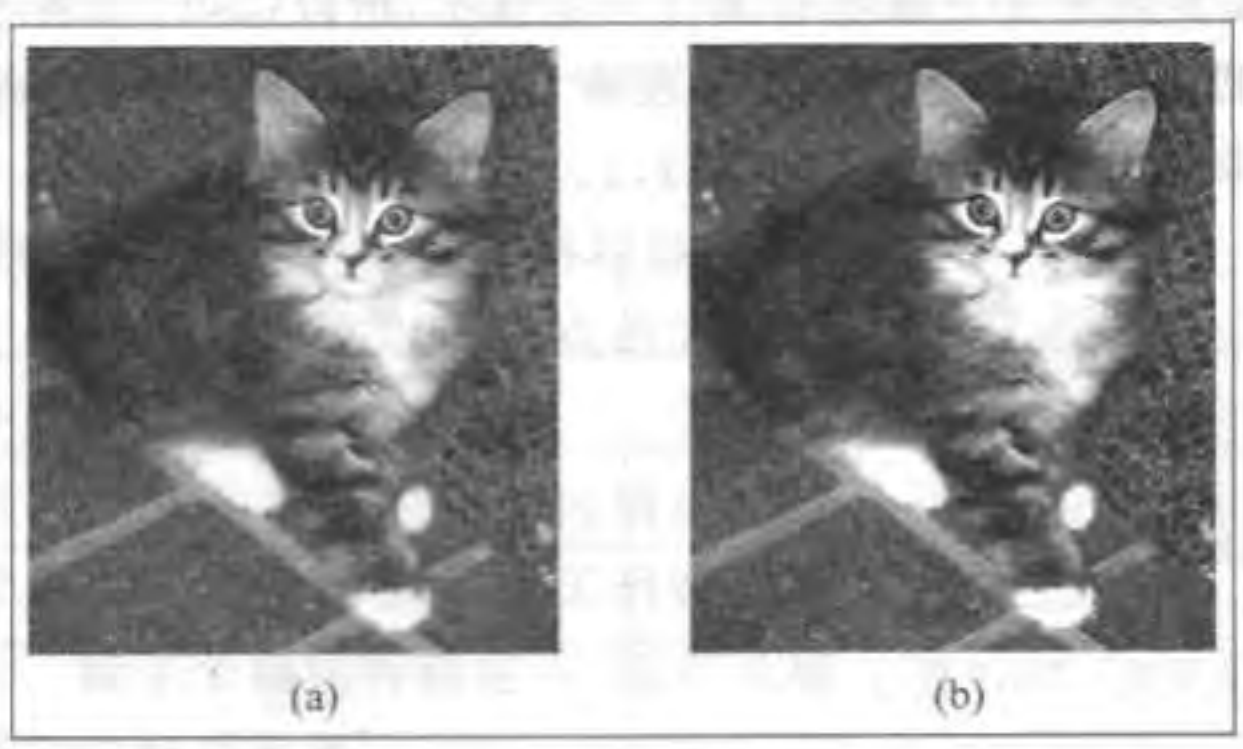


图 1.34　图像减为每像素 6 比特和每像素 3 比特

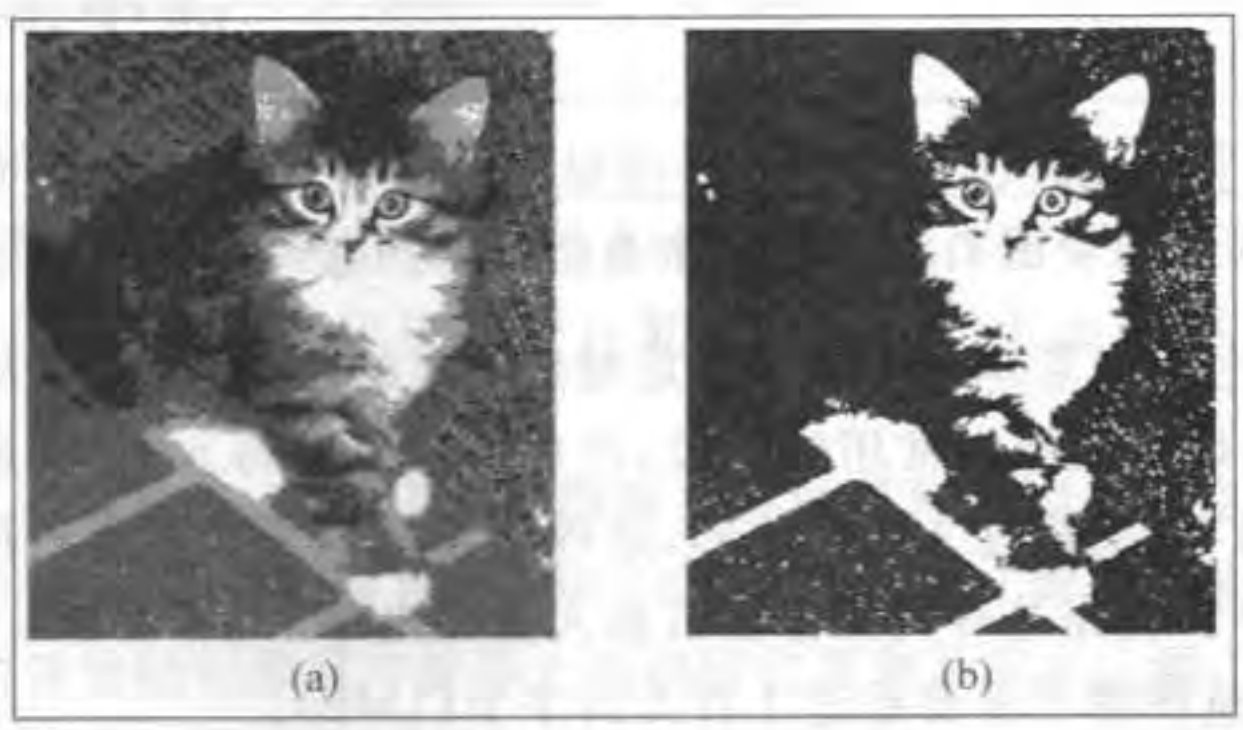


图 1.35　图像减为(a)每像素 2 比特和(b)每像素 1 比特

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。

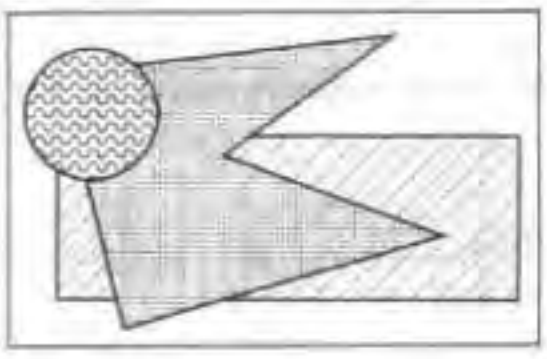


图 1.37　交叉阴影线模拟填充区域

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)。

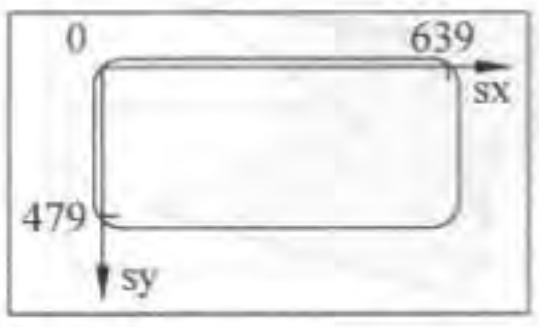


图 1.39 光栅显示器的内置坐标系

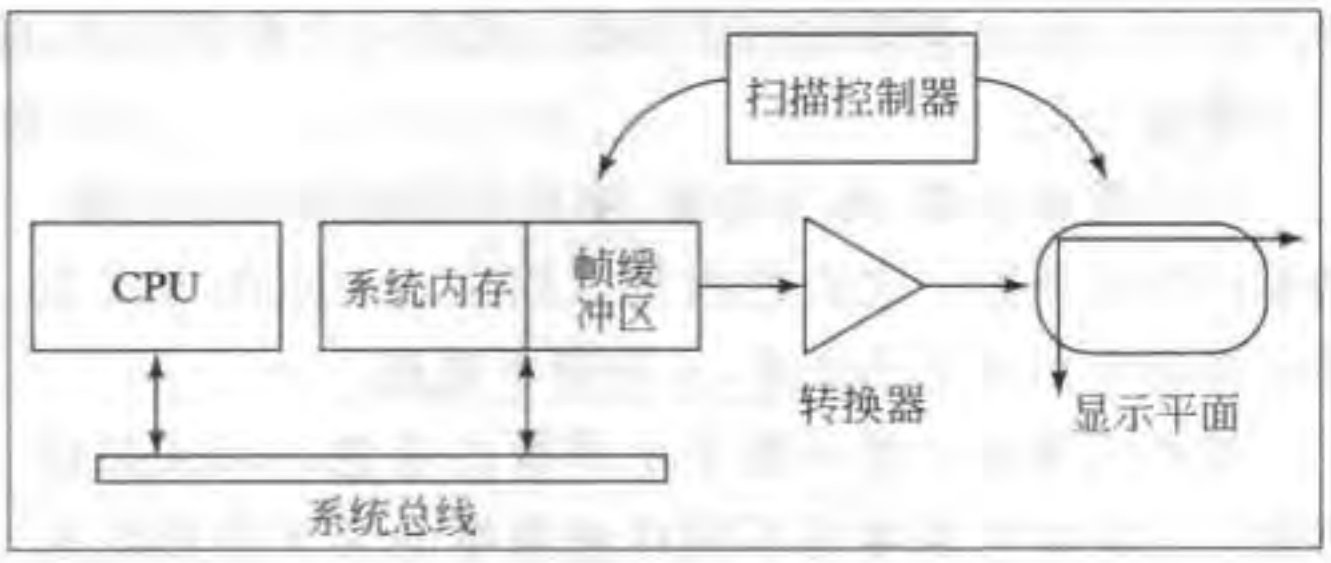


图 1.40 使用光栅显示器的计算机流程图

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)。

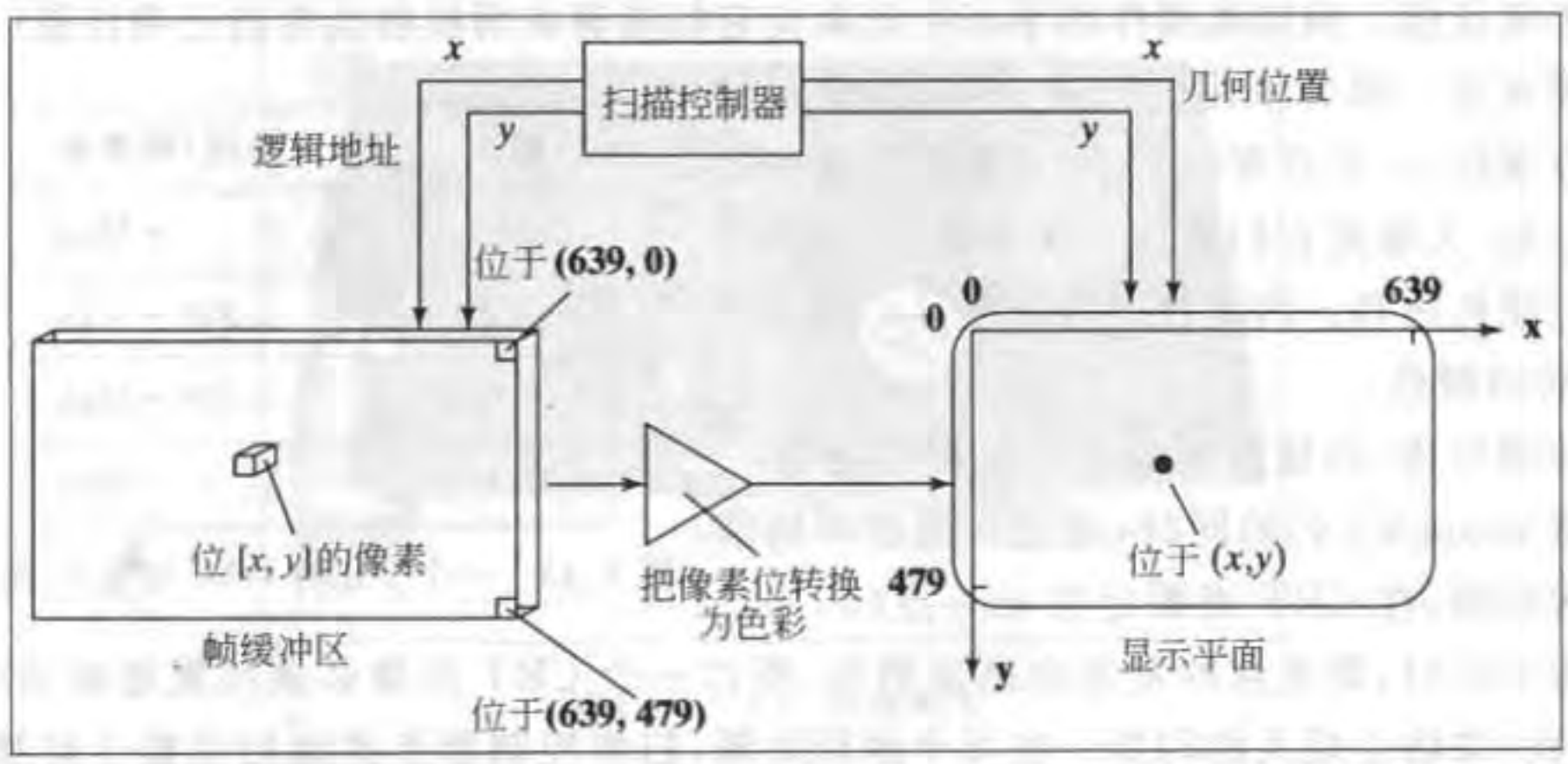


图 1.41　从帧缓冲区中扫描一个图像到显示器屏幕

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。

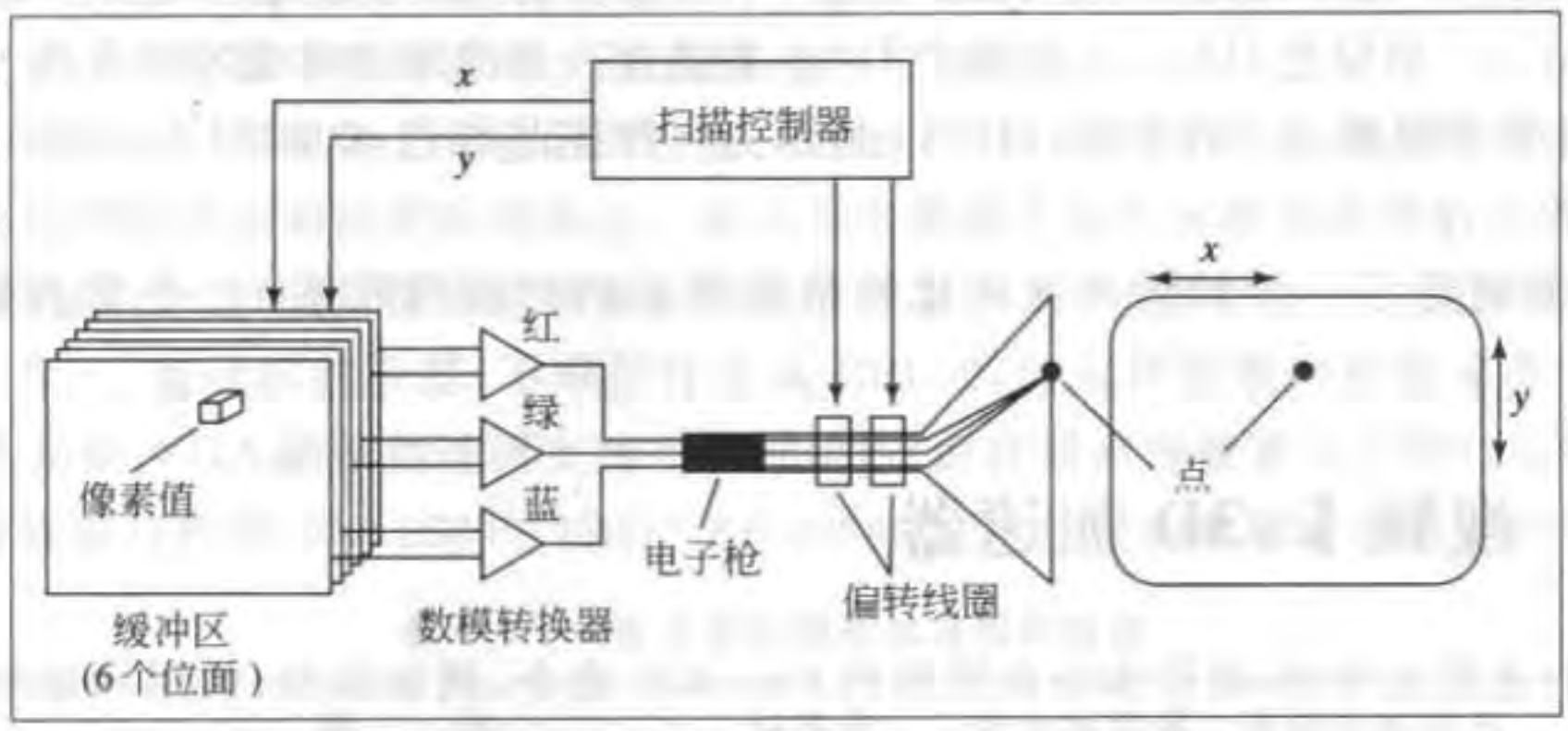


图 1.42　彩色视频显示系统的操作

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

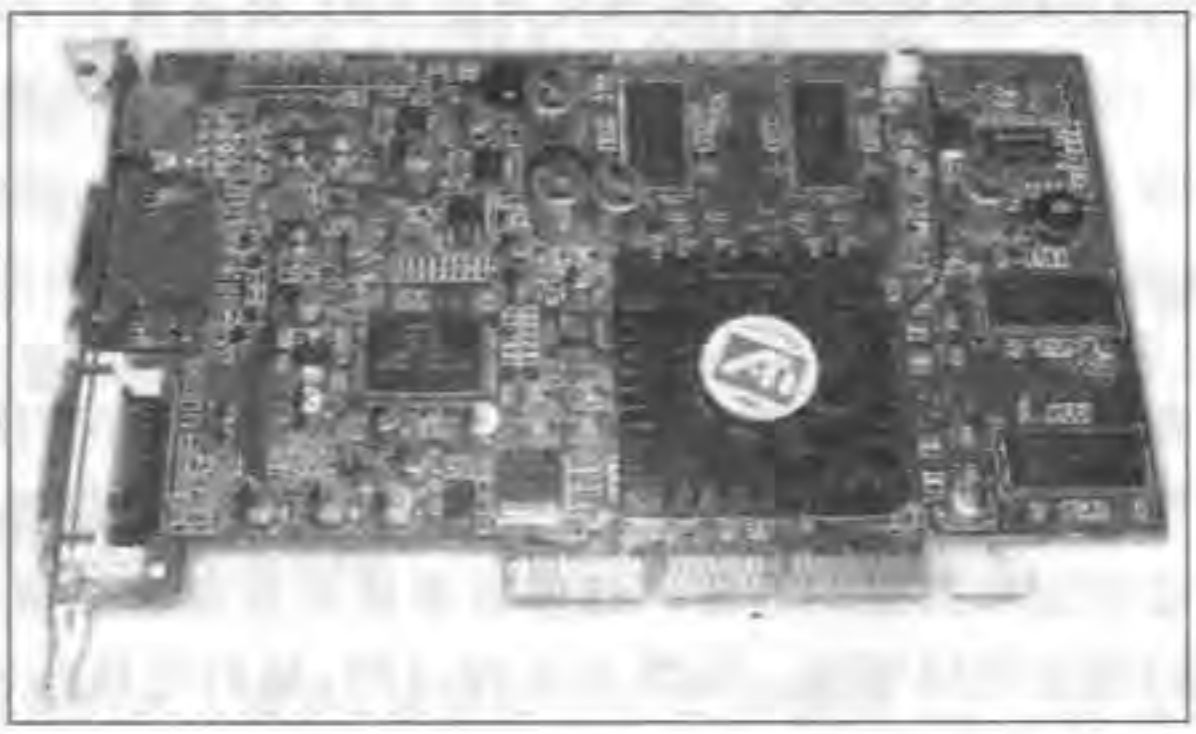


图 1.44　ATI 的视频卡

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。

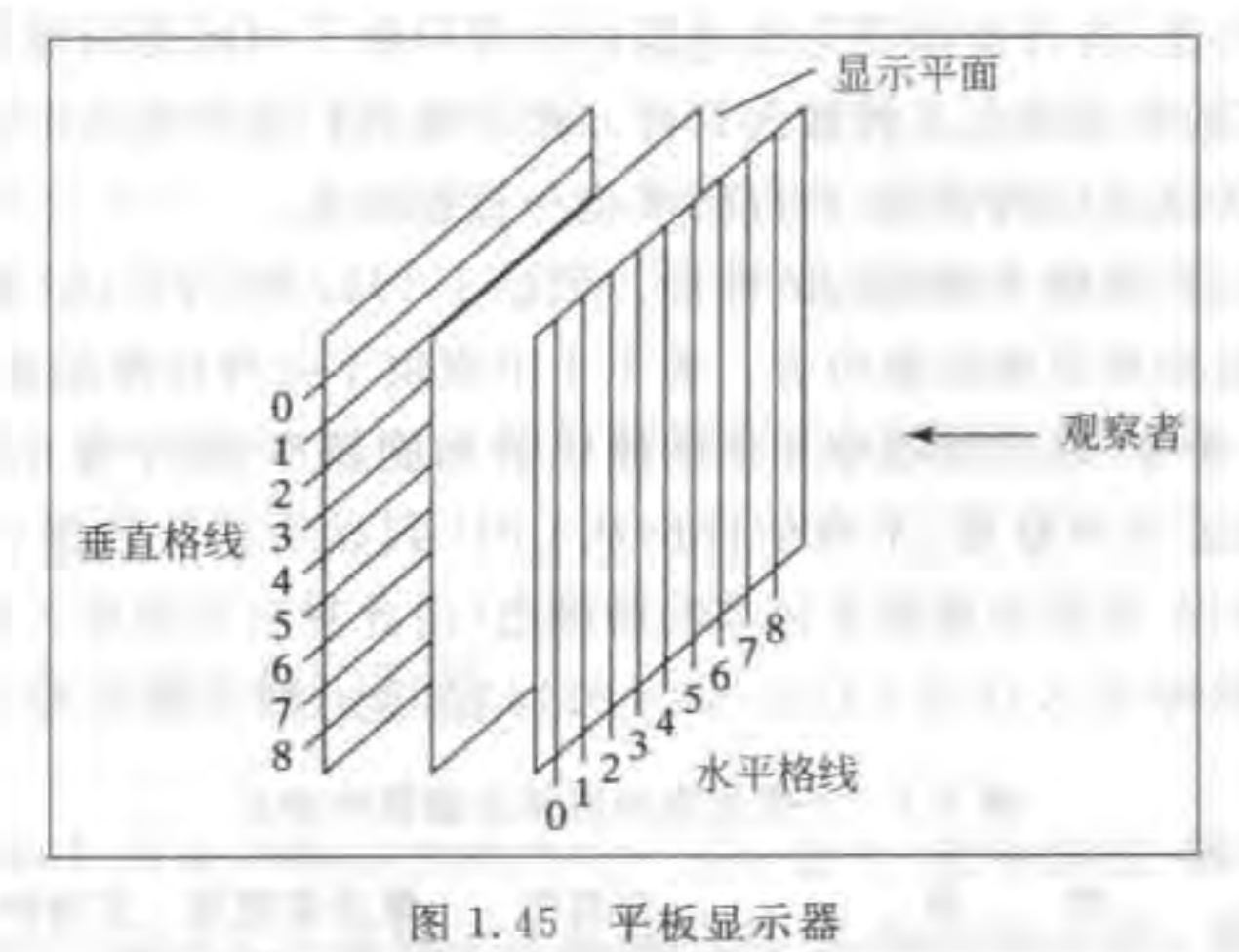


图 1.45 平板显示器

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。

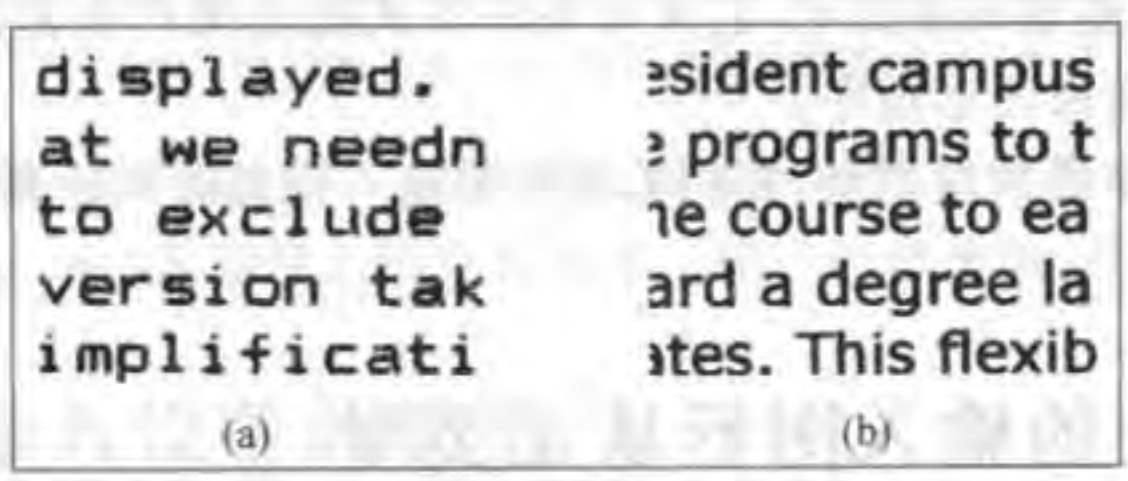


图 1.46　放大的点阵打印机和激光打印机图像

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。

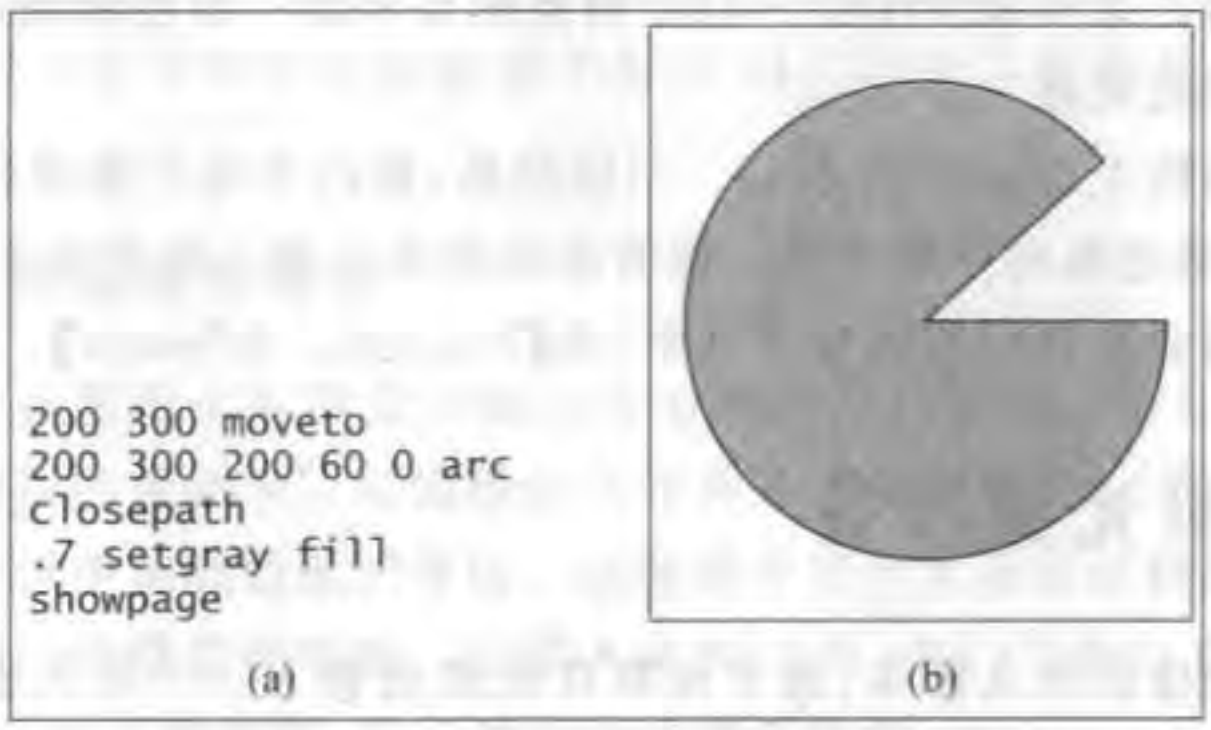


图 1.47 页面描述语言和用它生成的图像

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。

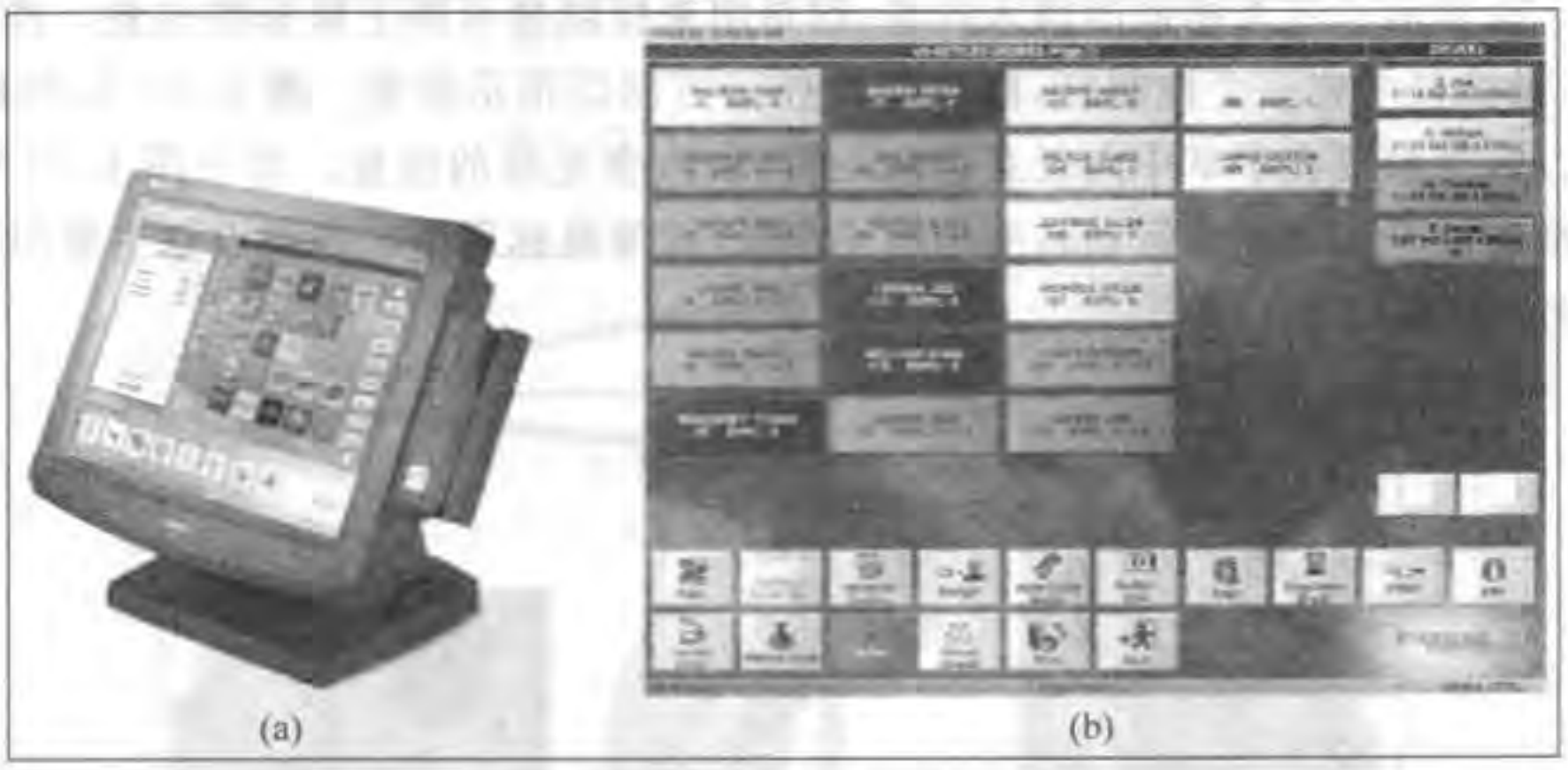


图　1.48

(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。

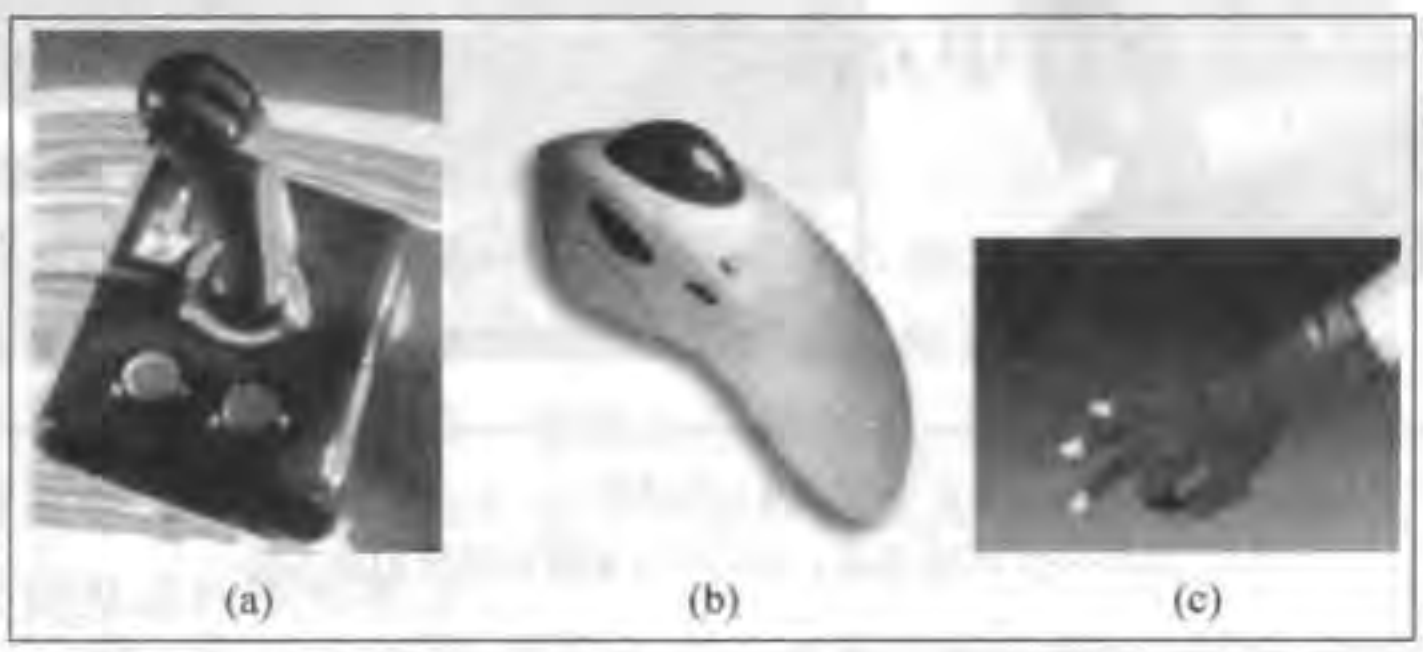


图 1.50

(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)。

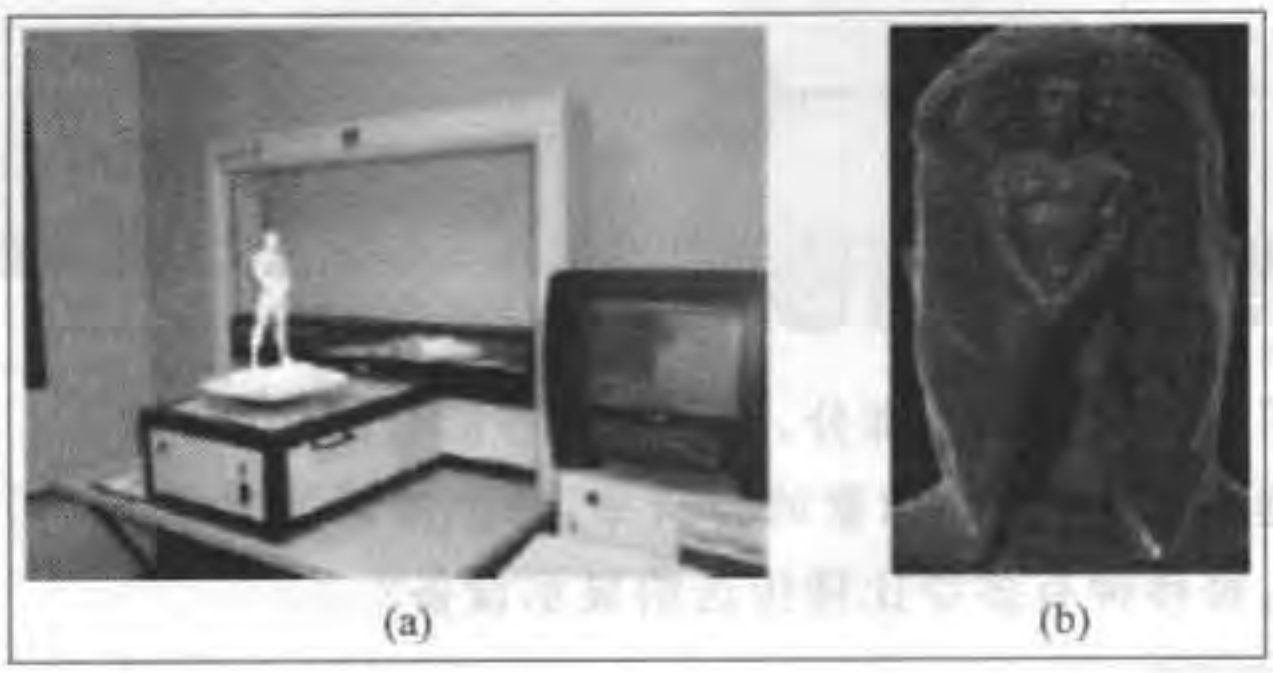


图 1.51　数字化三维模型

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。

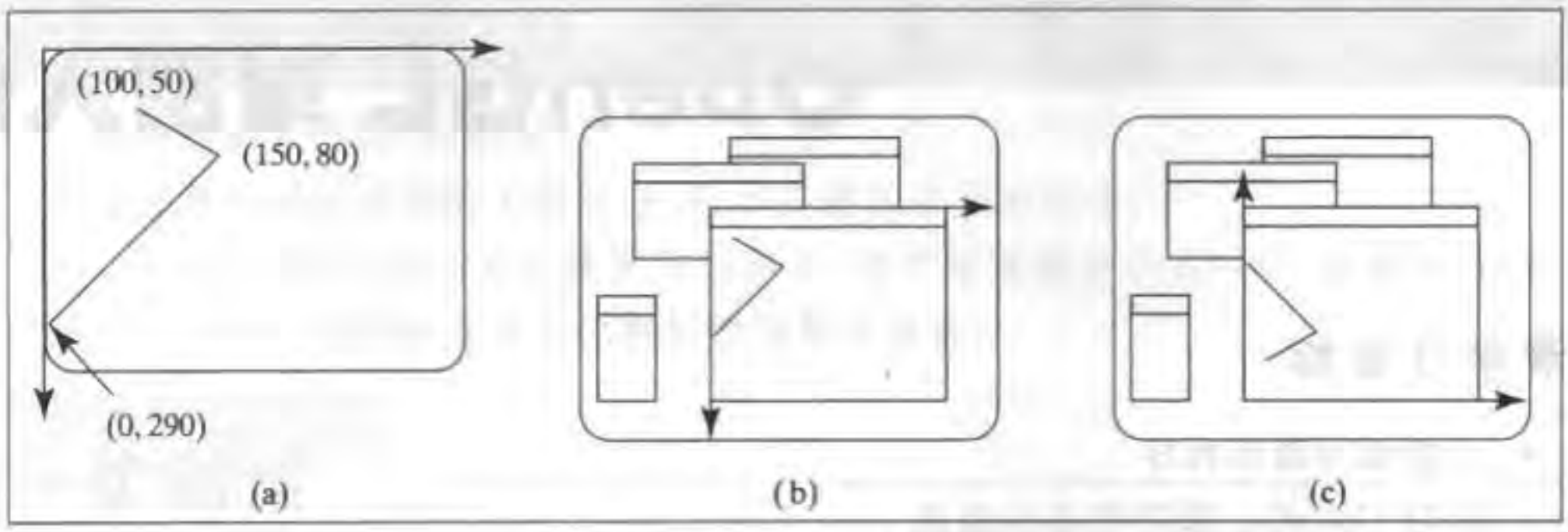


图 2.1 一些常见的显示布局

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
```

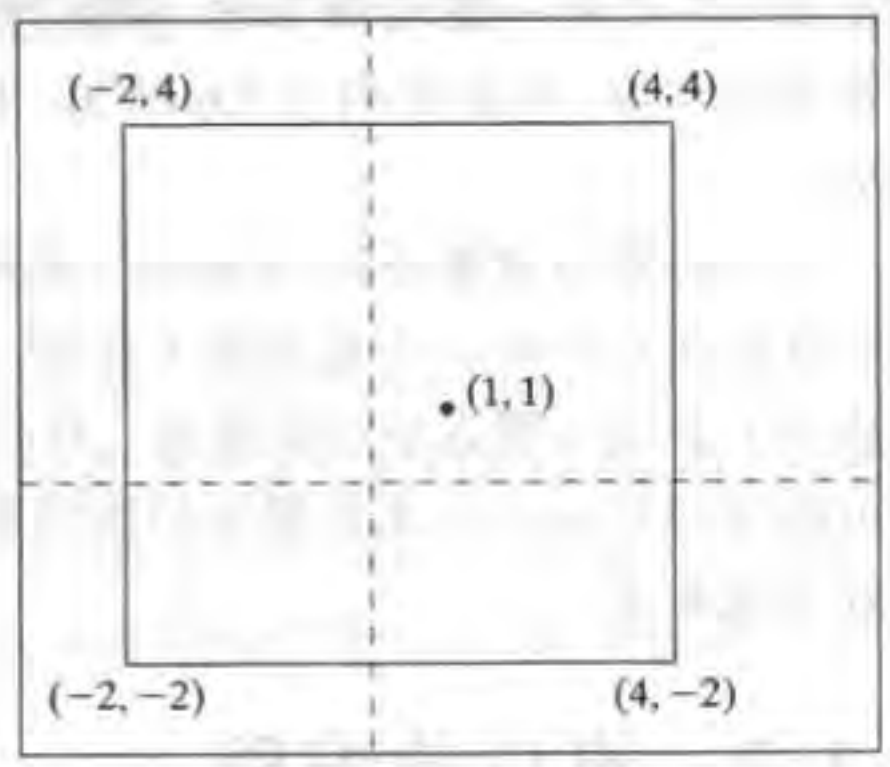


图 2.2　一个用 moveTo 和 lineTo 命令画的正方形

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。

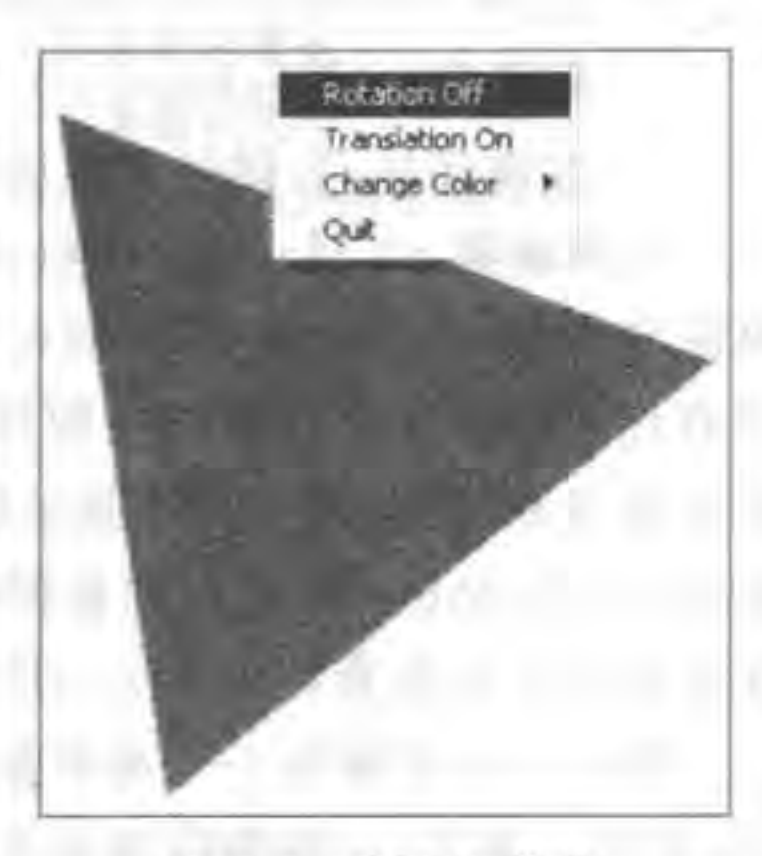


图 2.3　GLUT 菜单

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。

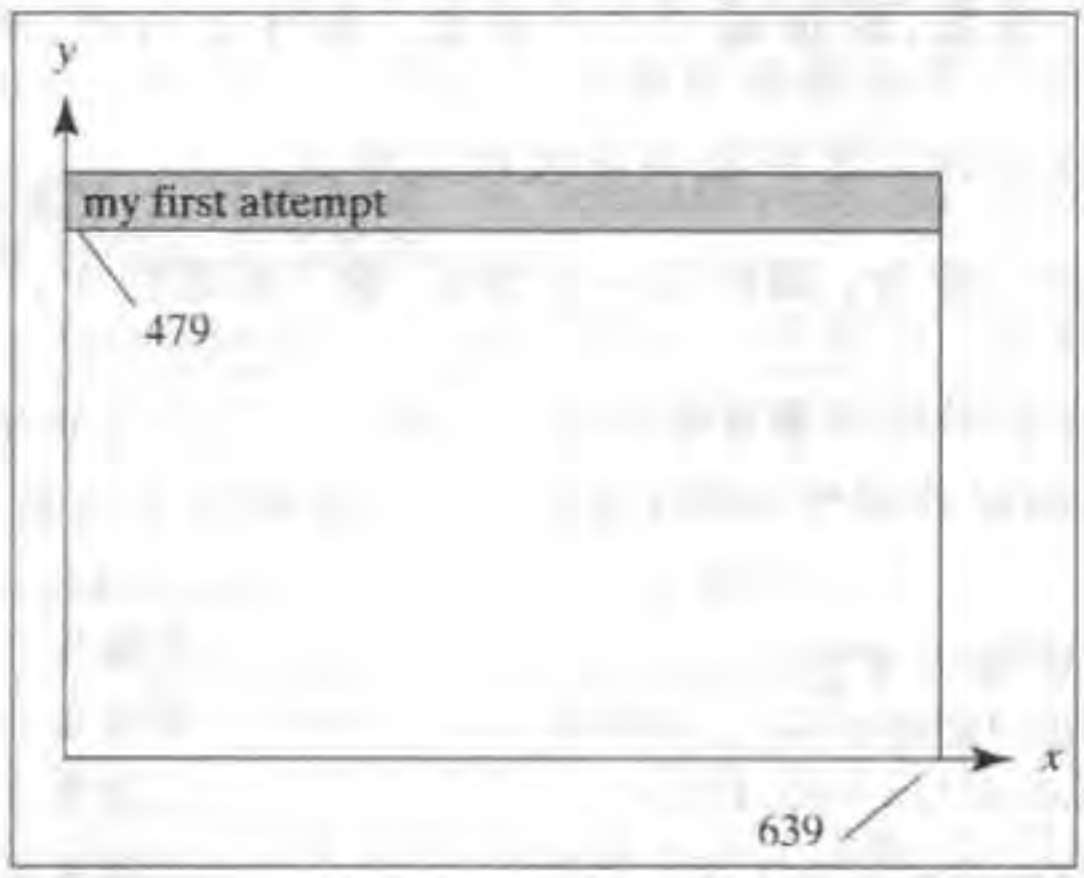


图 2.6 初始绘图坐标系

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。

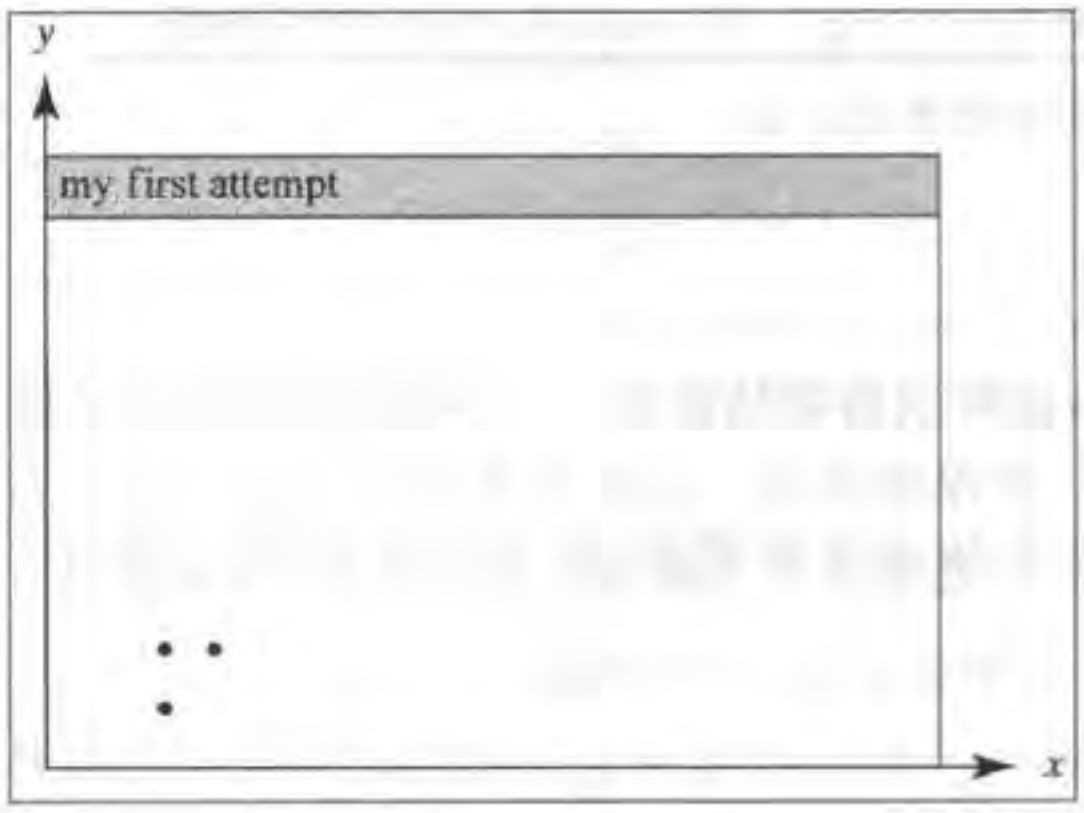


图 2.7　画 3 个点

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

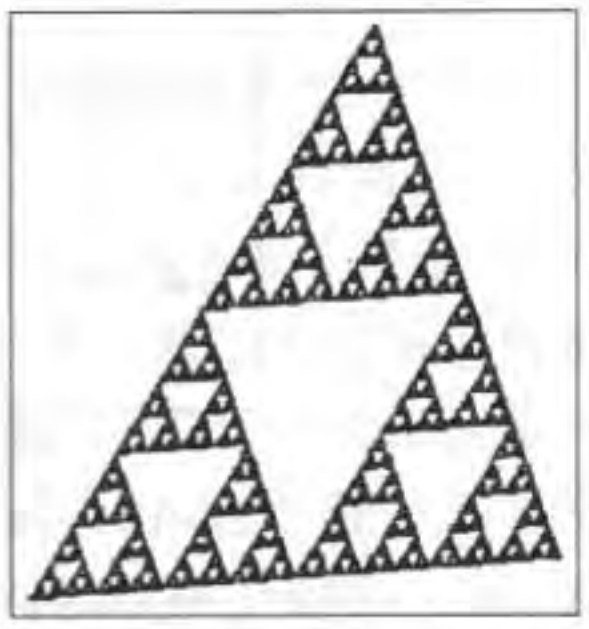


图 2.13　赛平斯基垫片

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。

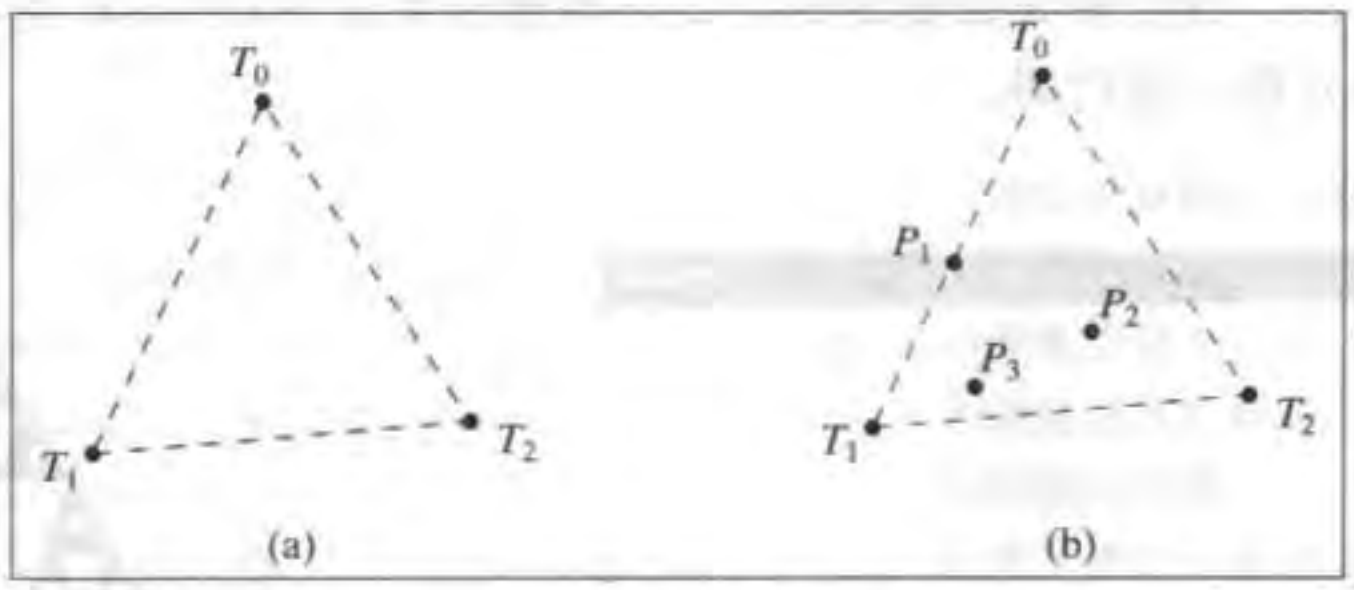


图 2.14 构建赛平斯基垫片

```
//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。

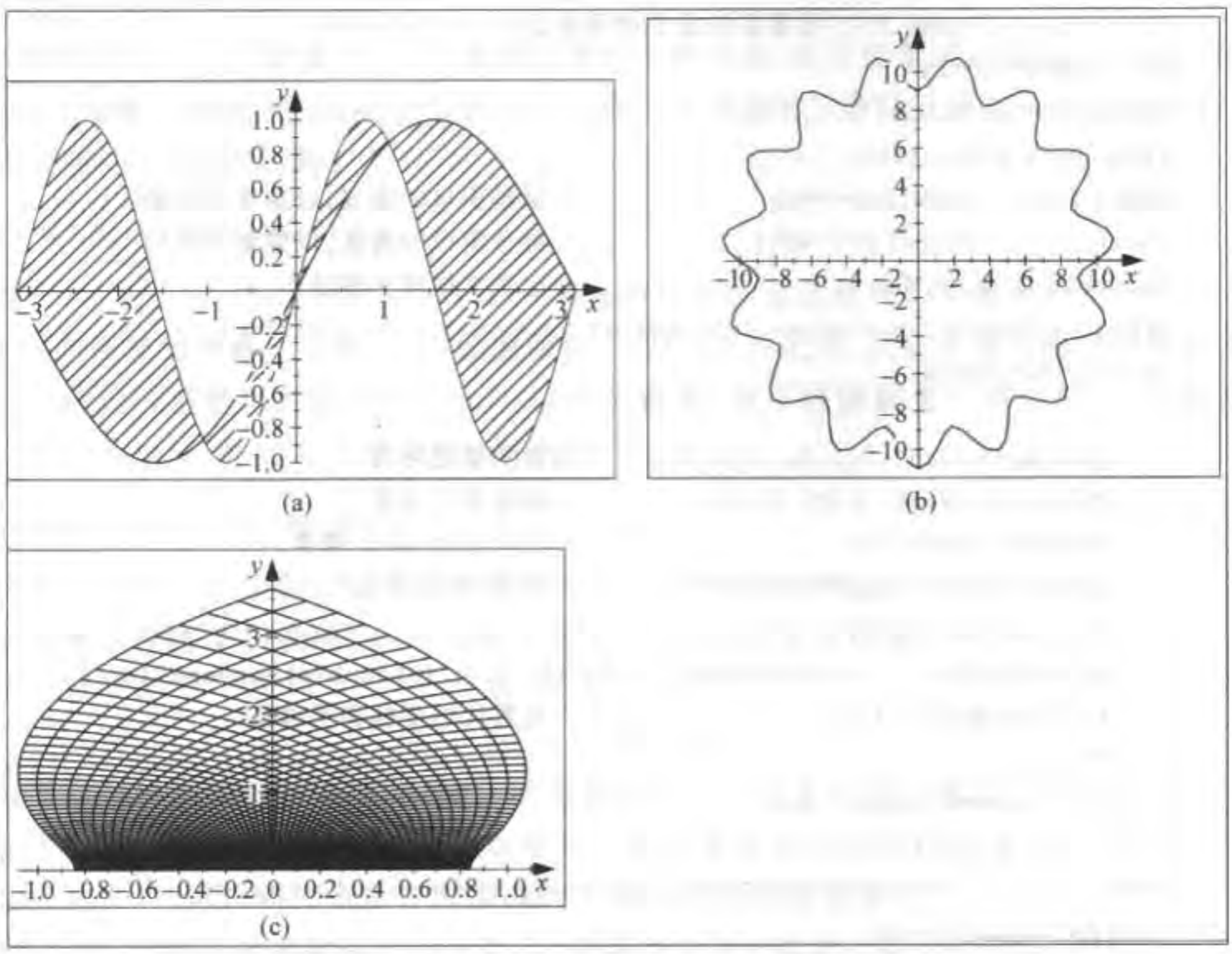
图 2.18(a)显示了两个重叠的函数,一个是 $f(x)=\sin(x)$,x 从 $-\pi$ 到 π。第二个函数是 $f(x)=\sin(2x)$,x 的变化范围与前者相同。

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();
```

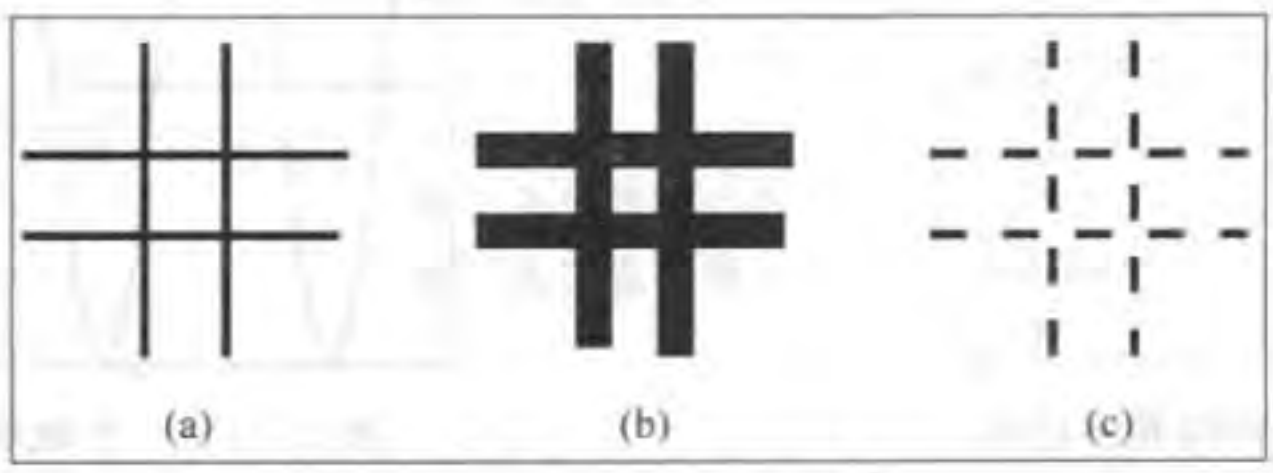


图 2.19　4 条线构成的简单图形

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。

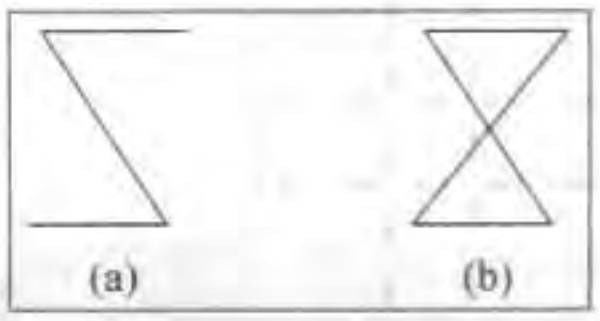


图 2.20 折线和多边形

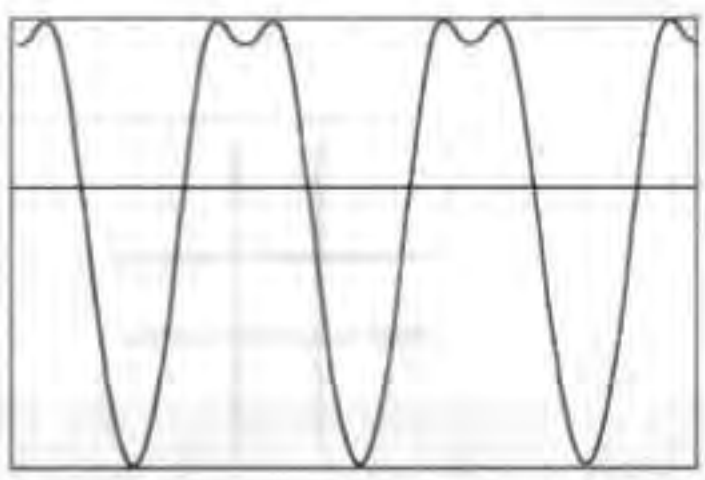


图 2.21 一个数学公式图

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。

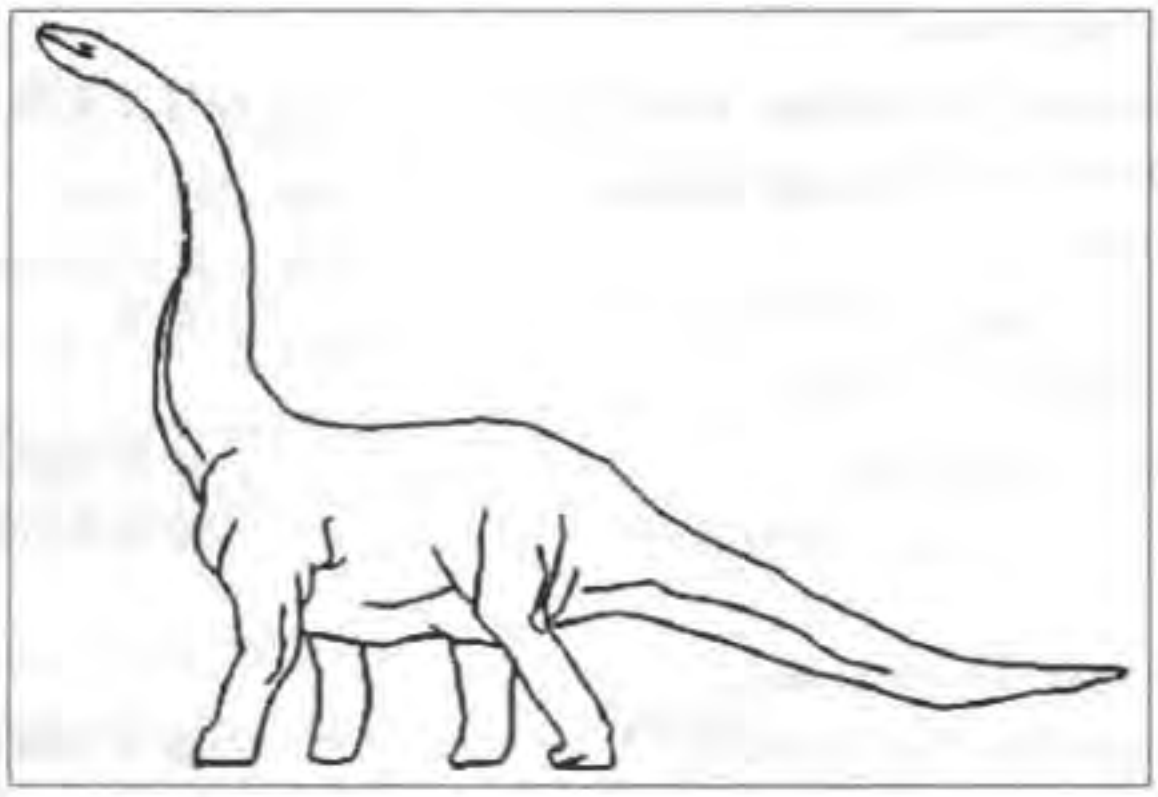


图 2.23　绘制存储在文件中的折线

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。

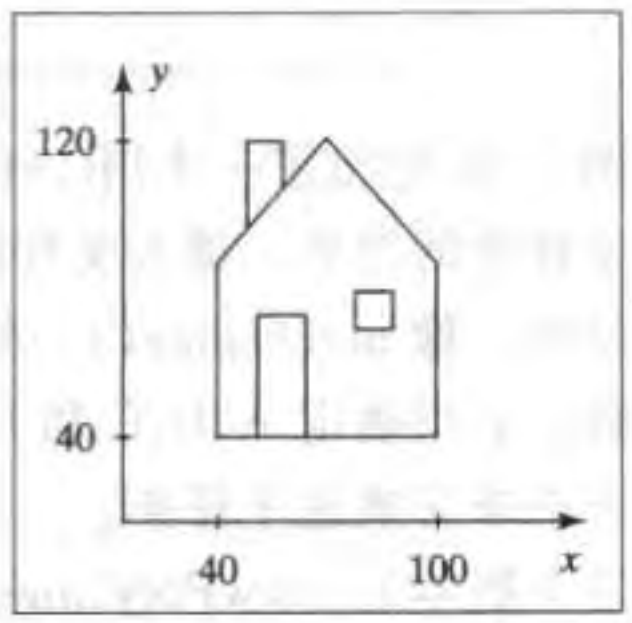


图 2.25 一栋房子

```
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

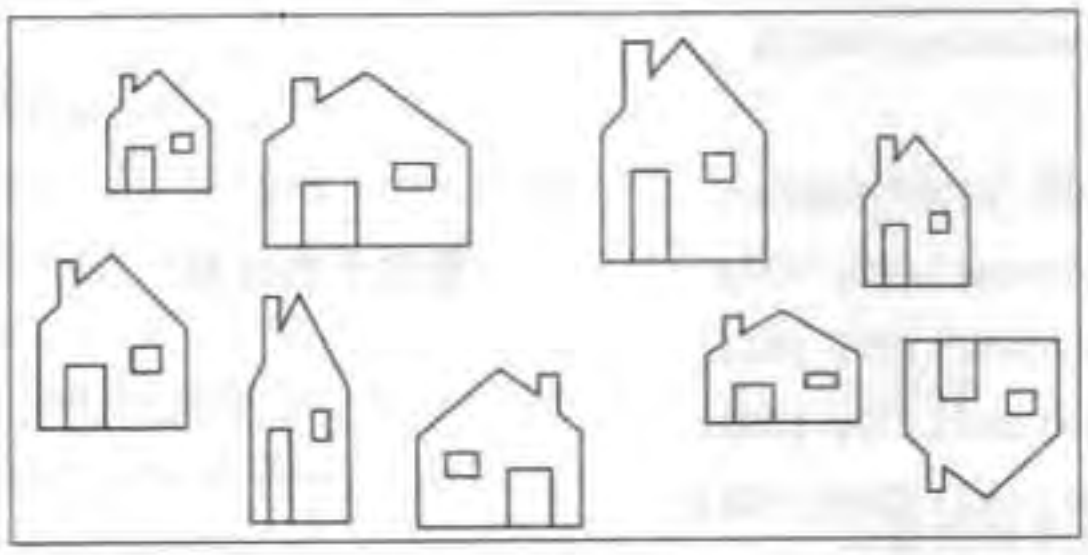


图 2.28 用 parameterizedHouse()画的村庄

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。

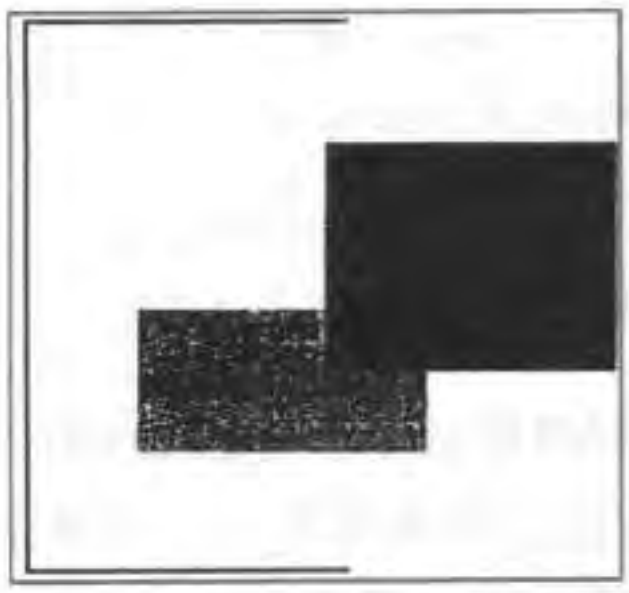


图 2.32 两个颜色填充的边校正矩形

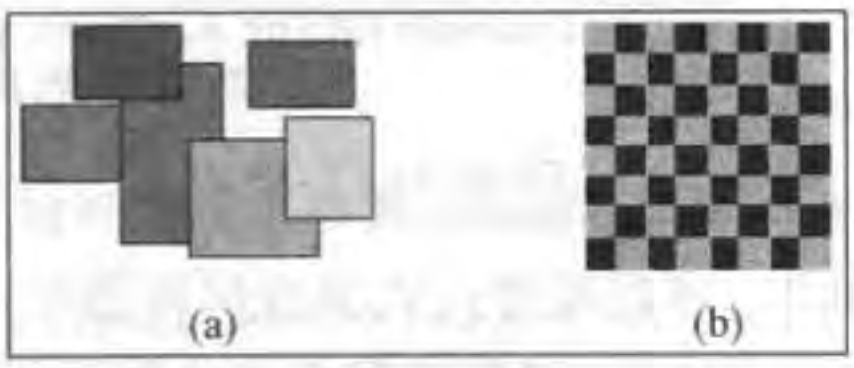


图 2.33

(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。

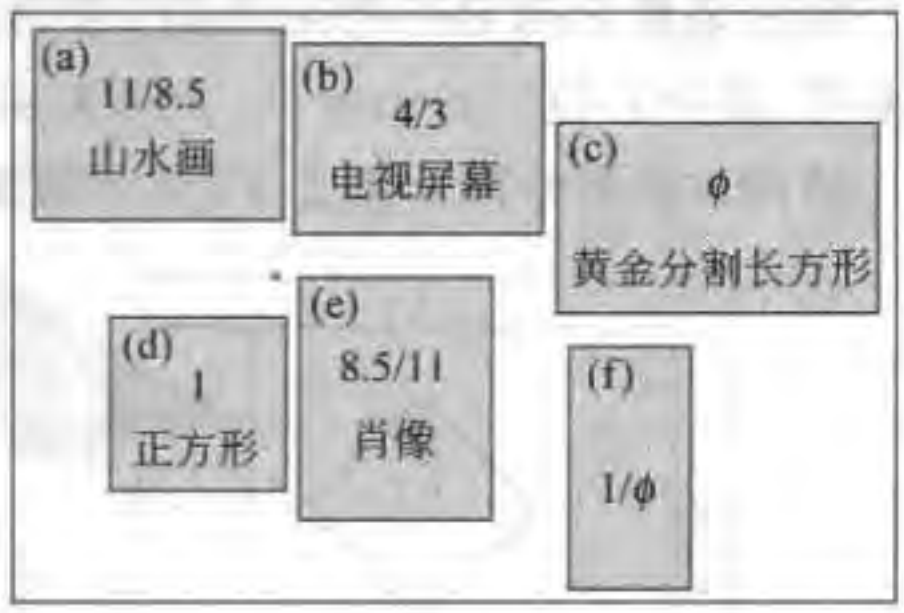


图 2.34 边校正矩形长宽比的例子

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。

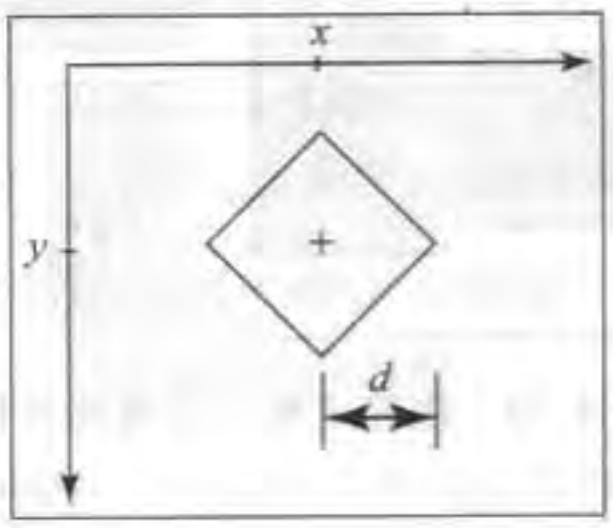


图 2.35 一个简单菱形

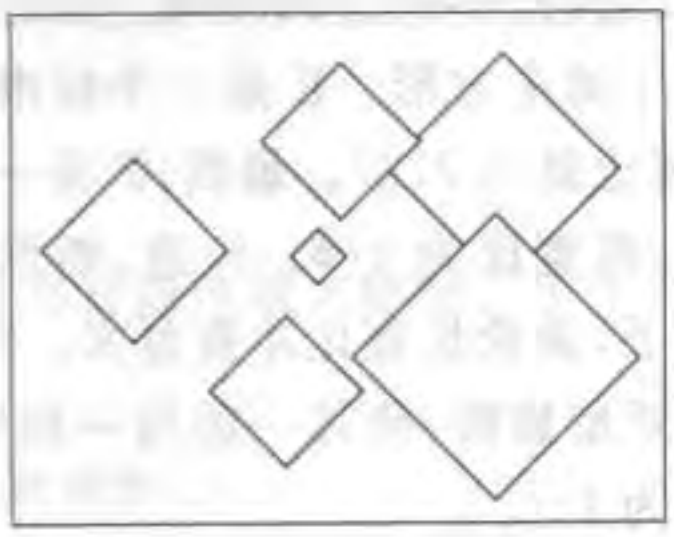


图 2.36 一些菱形

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)。

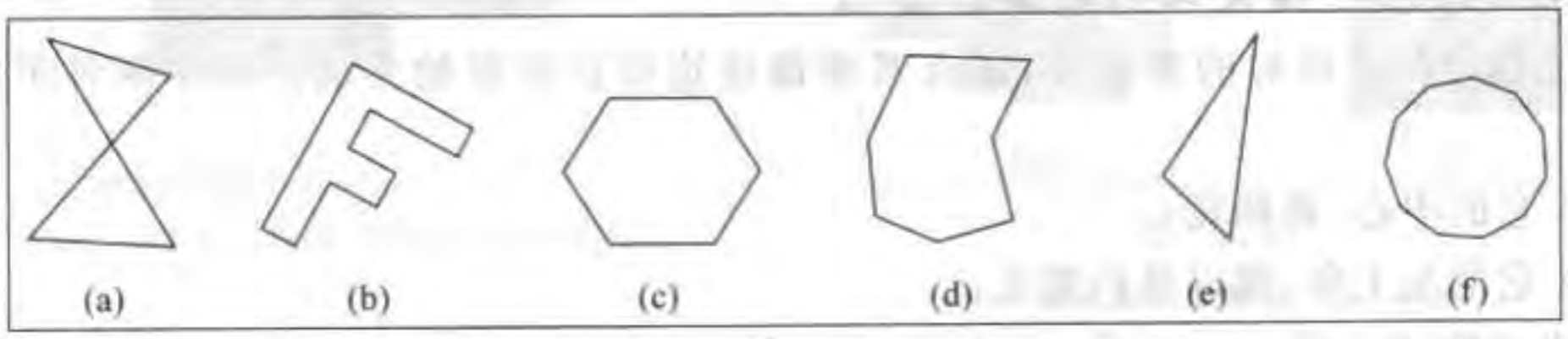


图 2.37 凸多边形和非凸多边形

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。

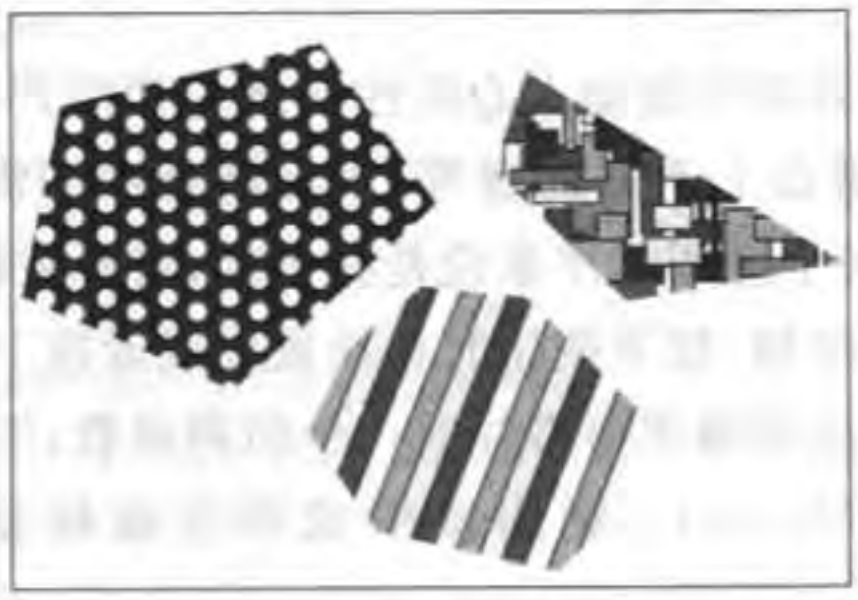


图 2.38 几个填充的凸多边形

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）。

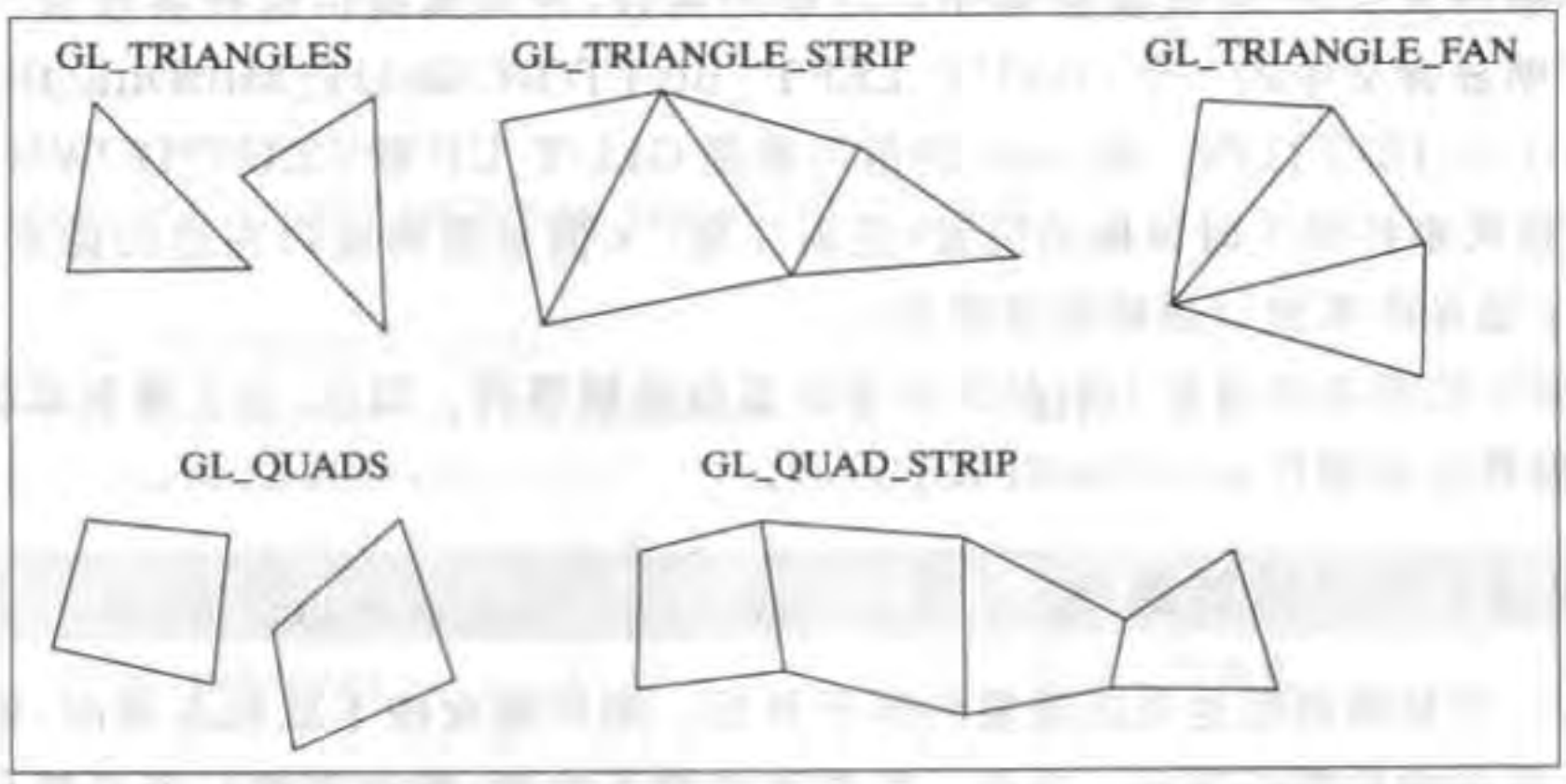


图 2.39 其他几何图形元素

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。

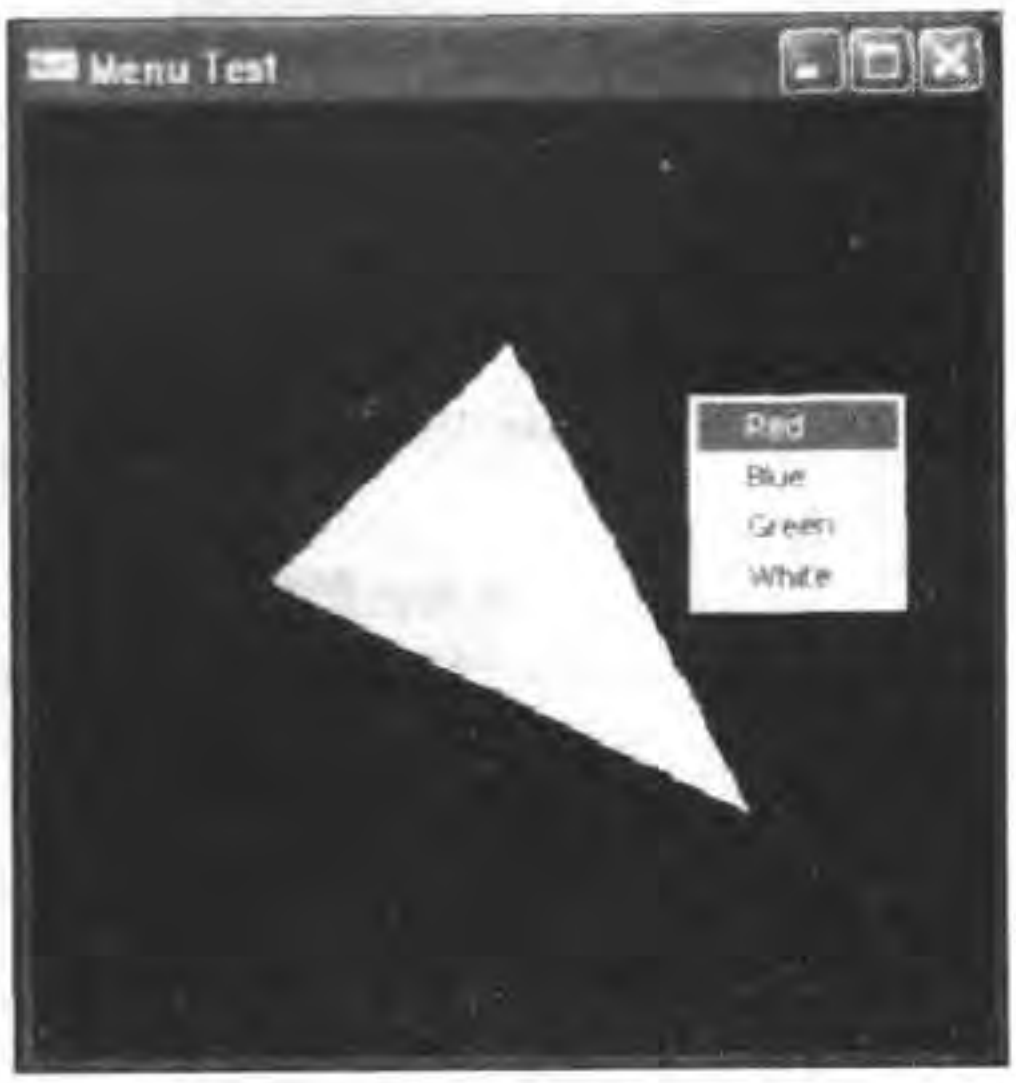


图 2.43 一个激活的 GLUT 菜单

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。

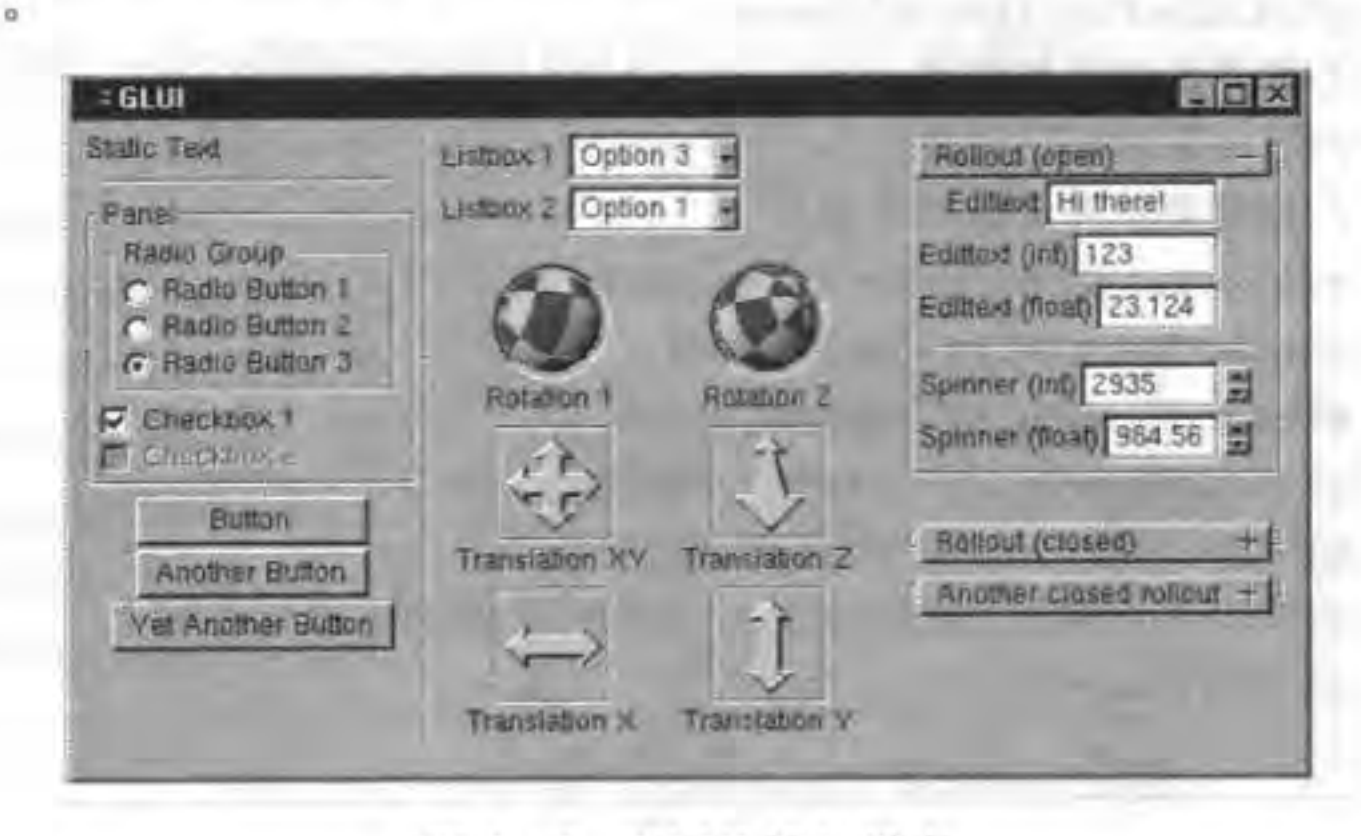
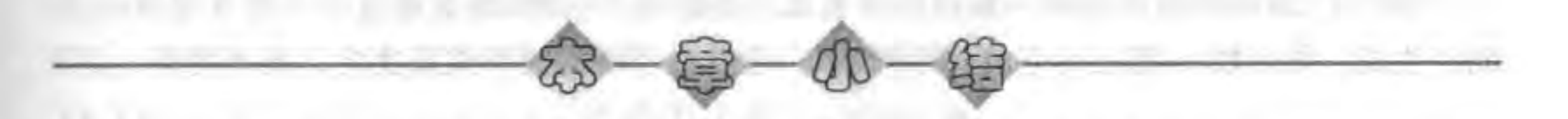


图 2.45 GLUI 接口样本

- 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？

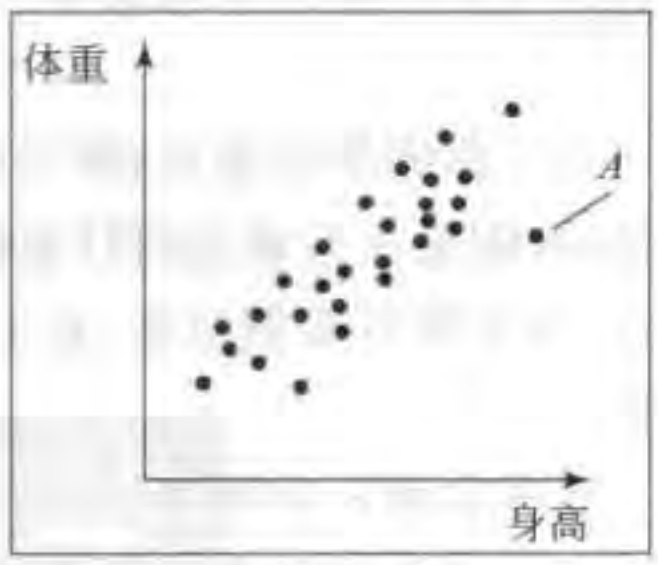


图 2.46 人的身高体重散点图

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。

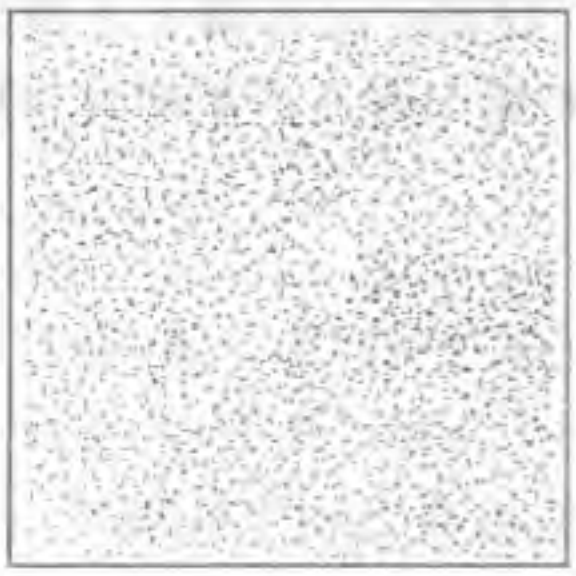


图 2.47　500 个随机点的点云

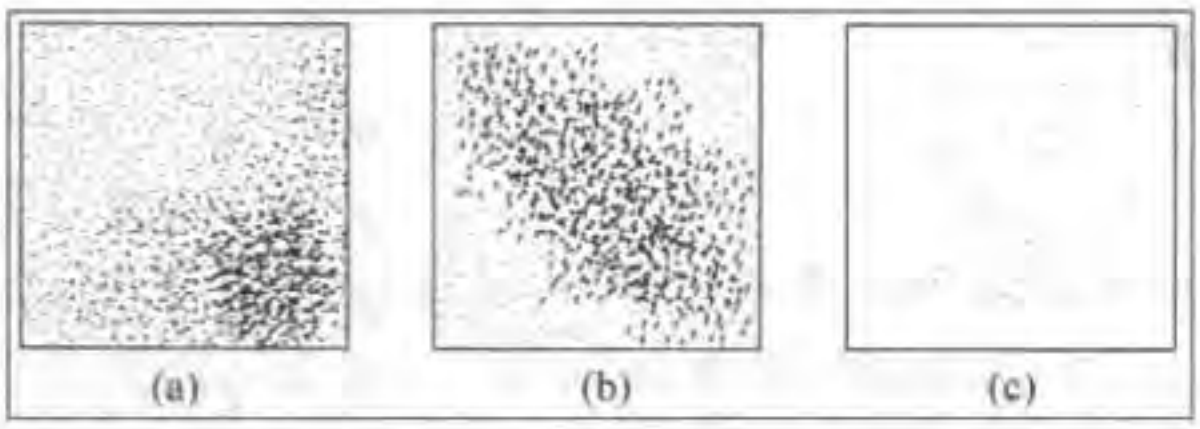


图 2.48　低级随机数生成器产生的点图

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。

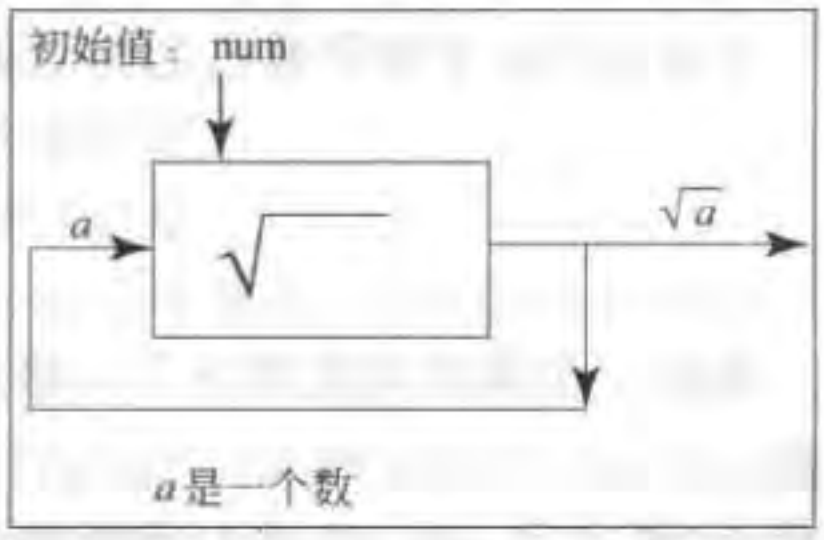


图 2.49　重复取平方根

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。

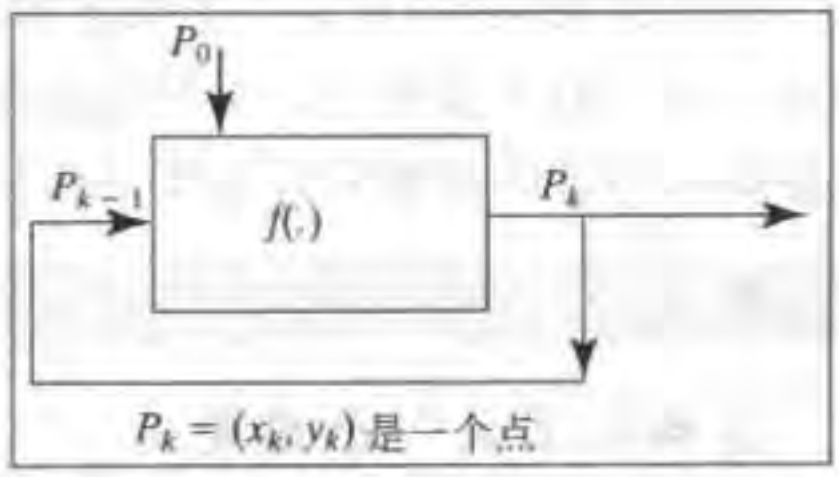


图 2.50　迭代函数点序列生成器

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)。

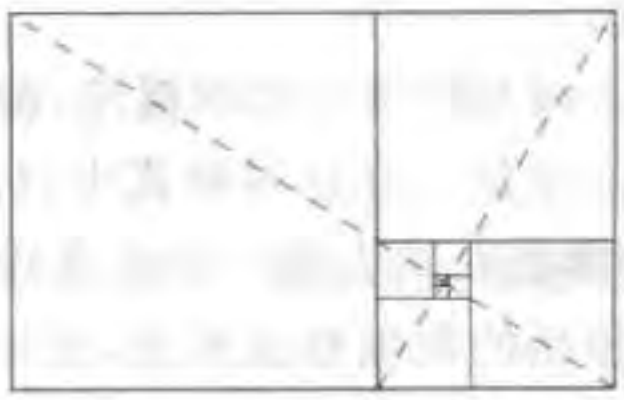


图 2.52　黄金矩形的无限回归

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。

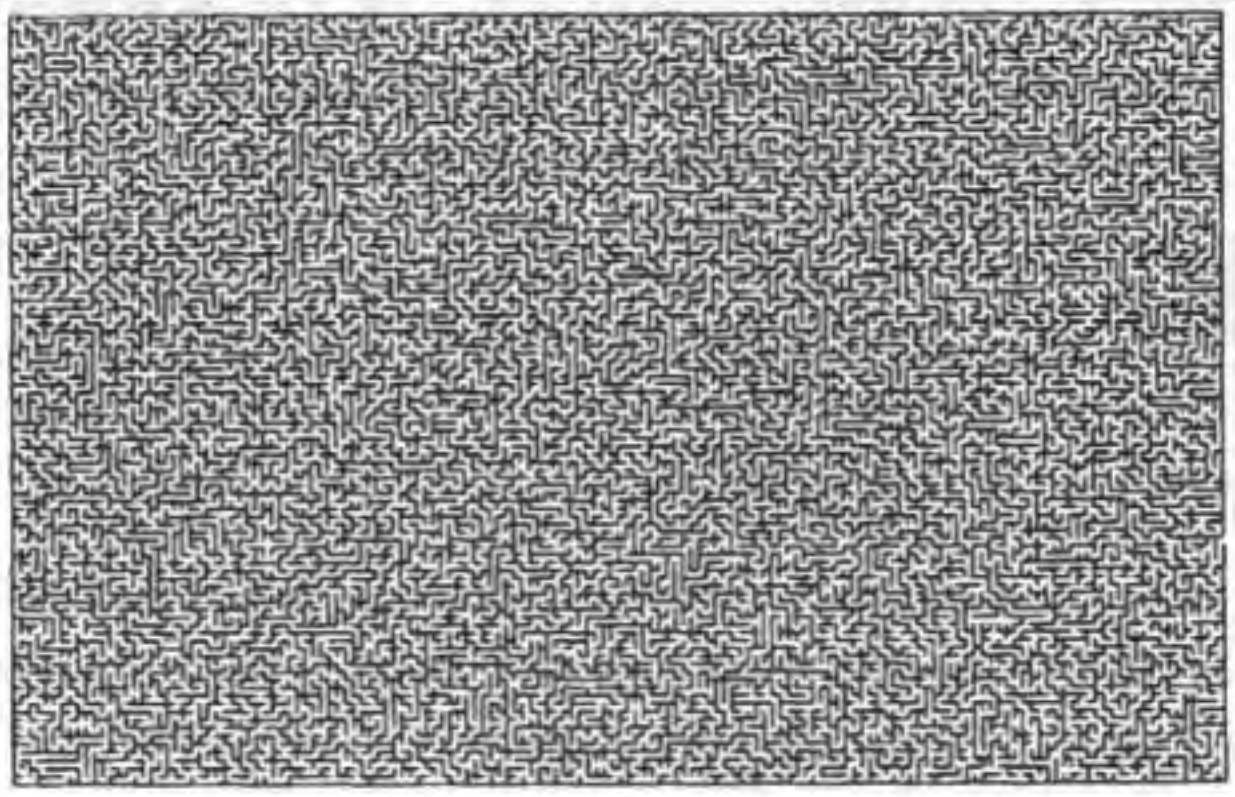


图 2.53　迷宫

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

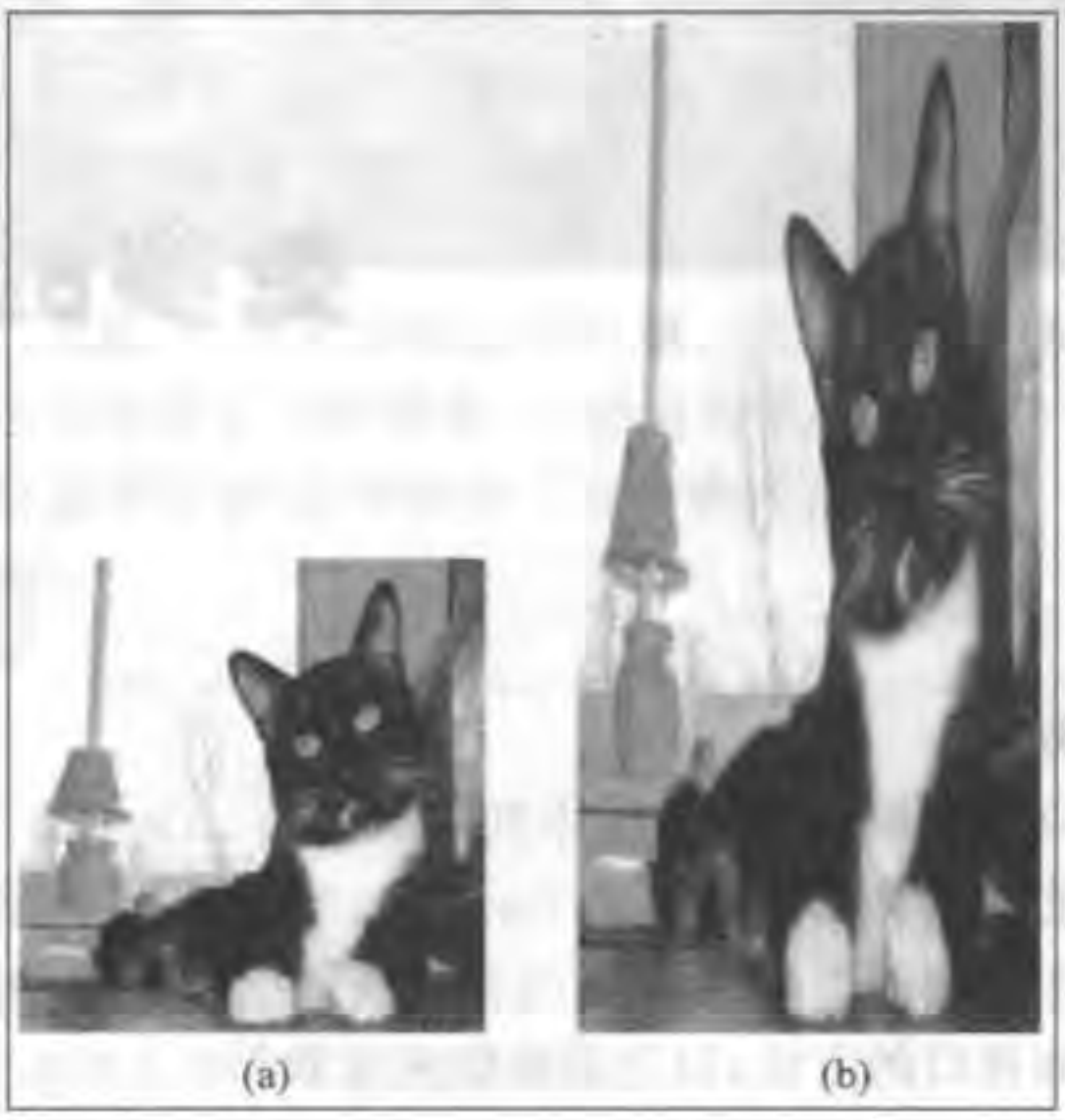


图 3.1

(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。

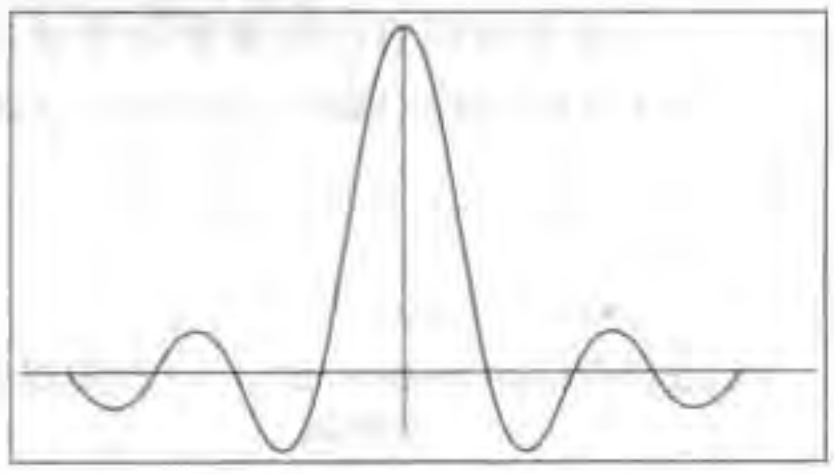


图 3.2　sinc 函数，图 3.3 程序执行结果的截图

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）

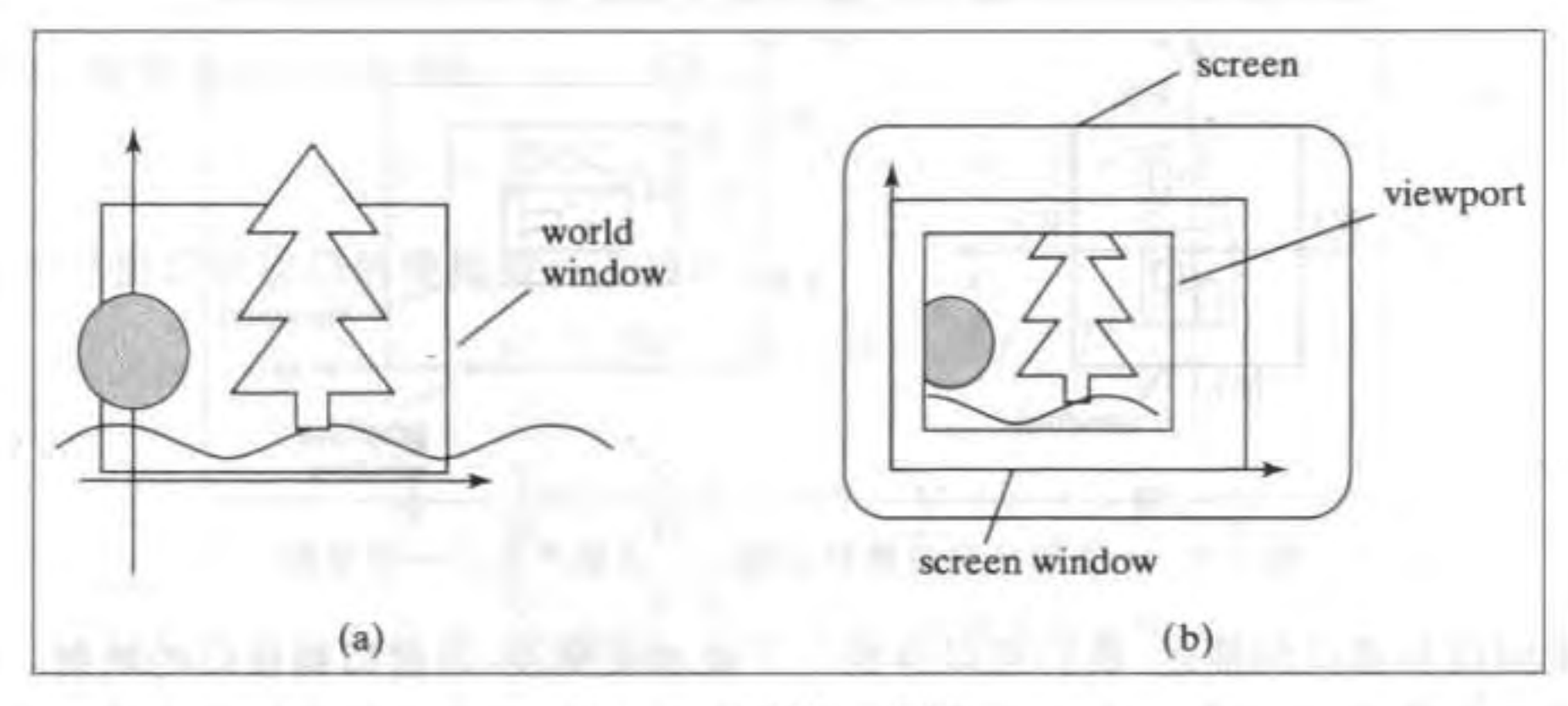


图 3.4　世界窗口和视口

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]。

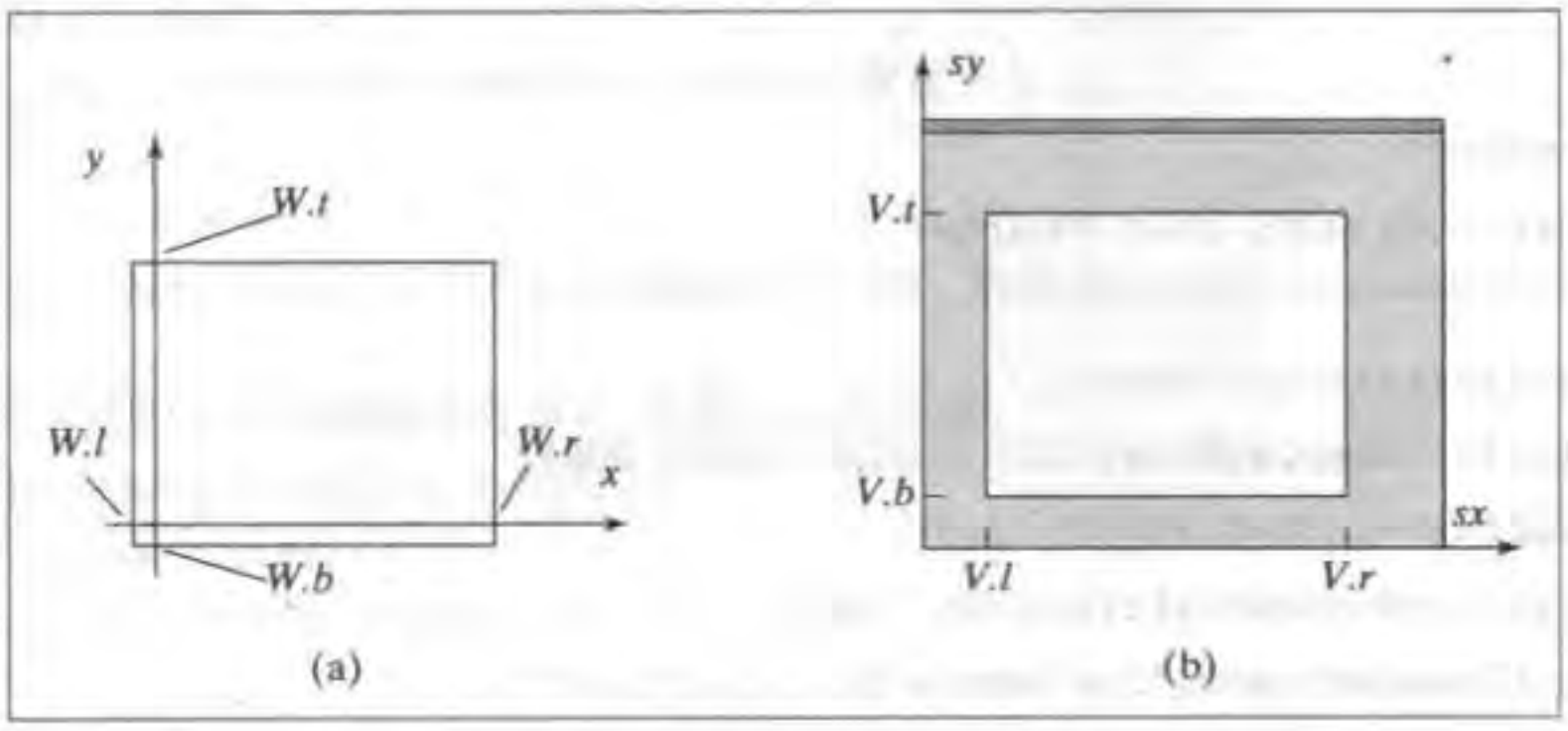


图3.5 指定(a)窗口和(b)视口

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。

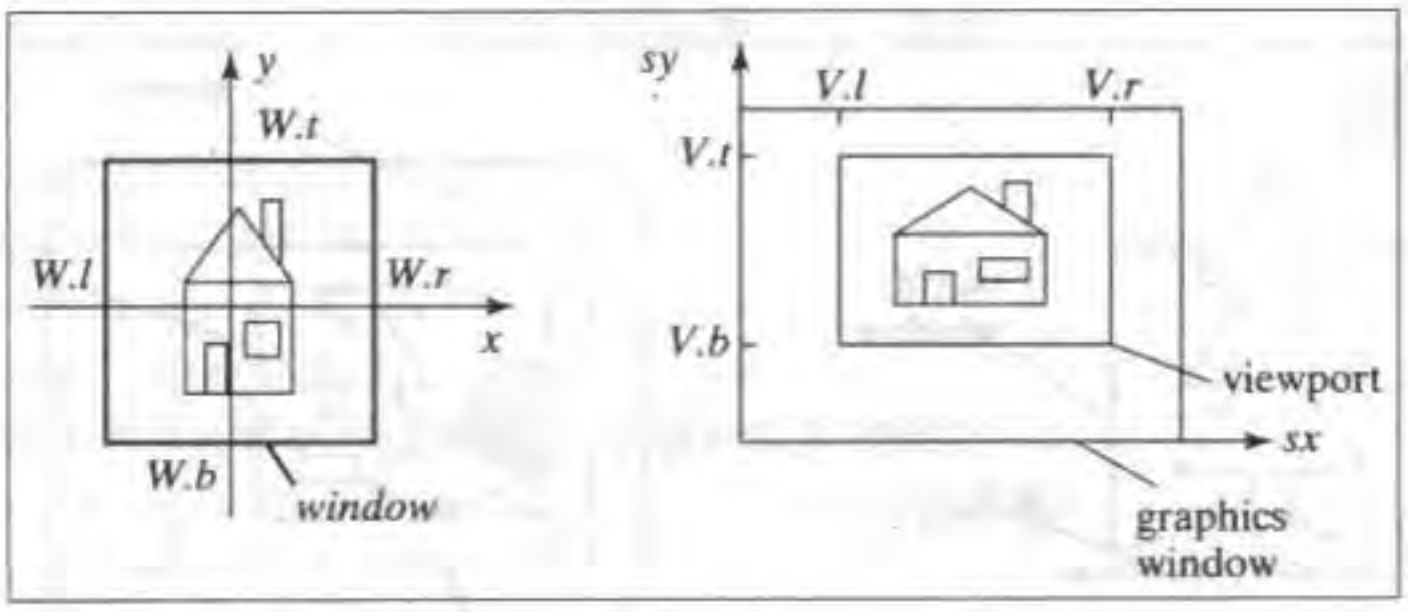


图3.6 一张图从窗口映射到视口，这里产生了一些变形

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。

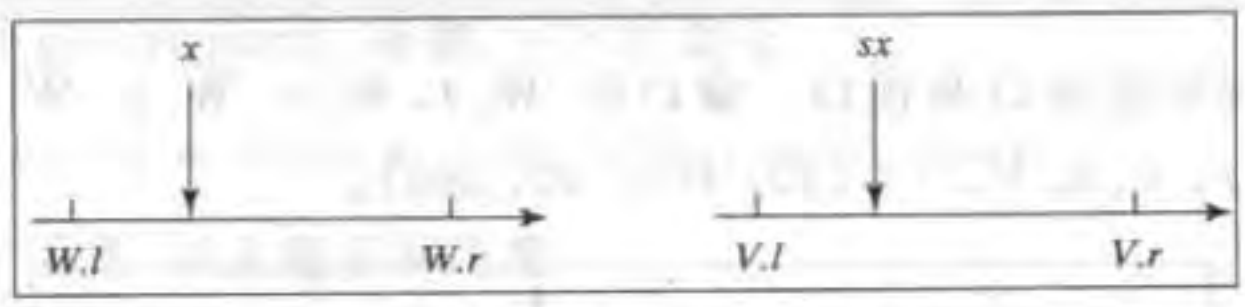


图 3.7　x 到 sx 映射的保持比例的性质

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)。

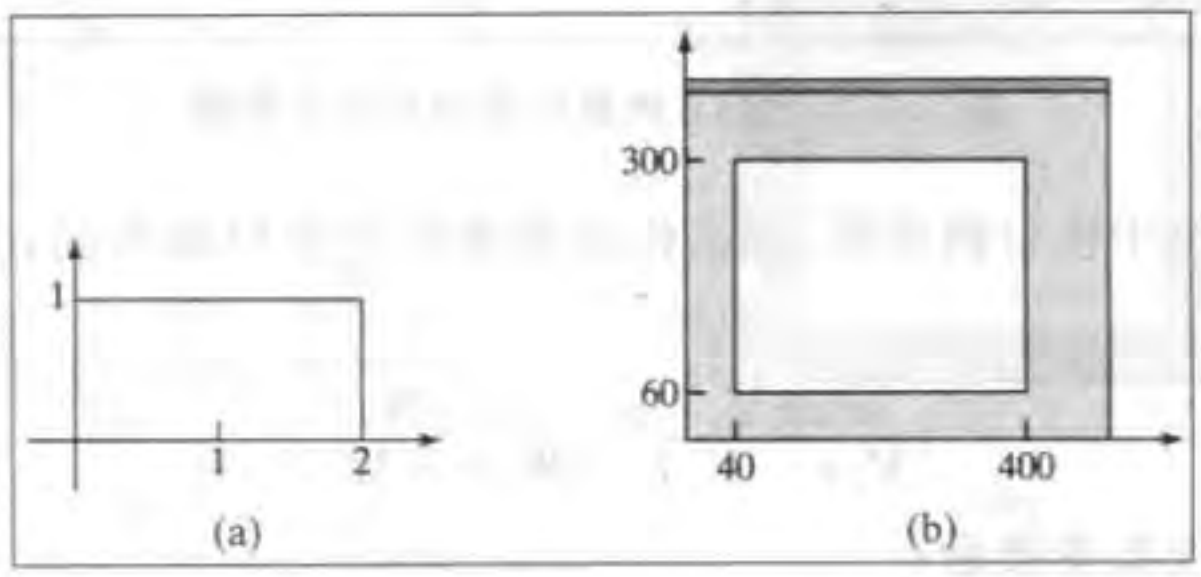


图 3.8 一个窗口和视口的例子

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。

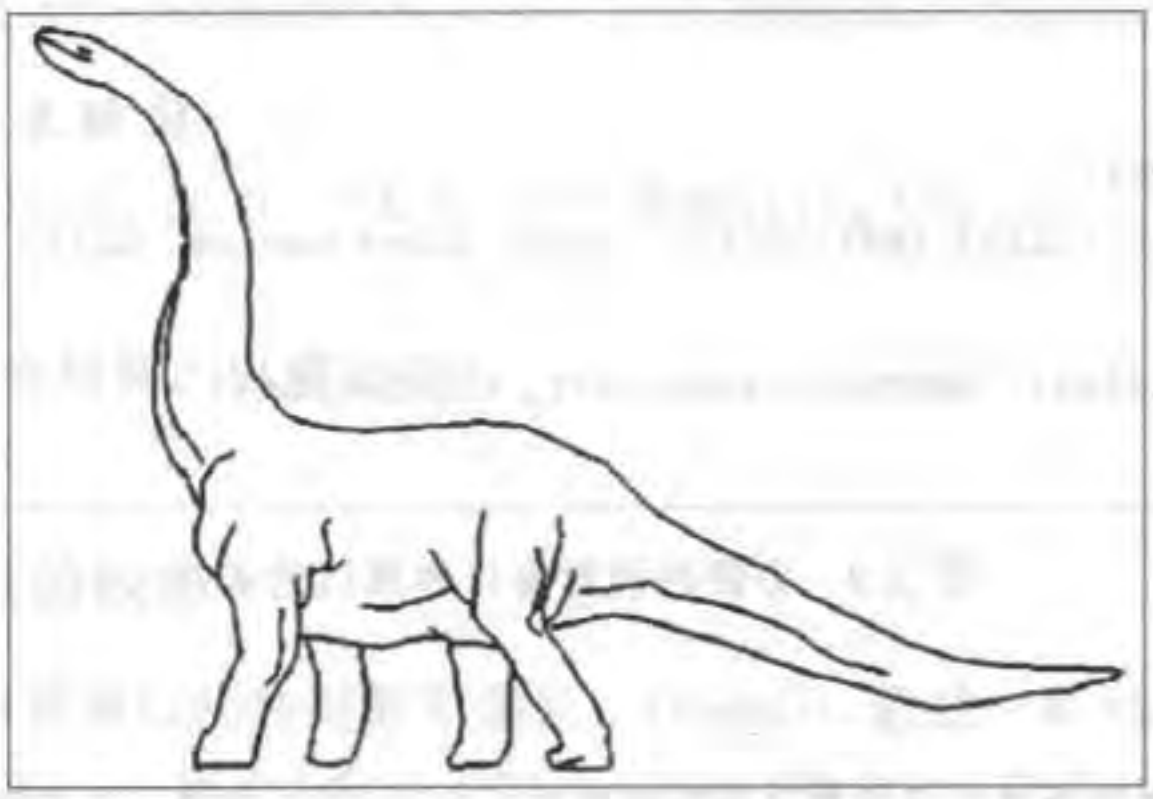


图 3.10 世界窗口中的恐龙

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
```

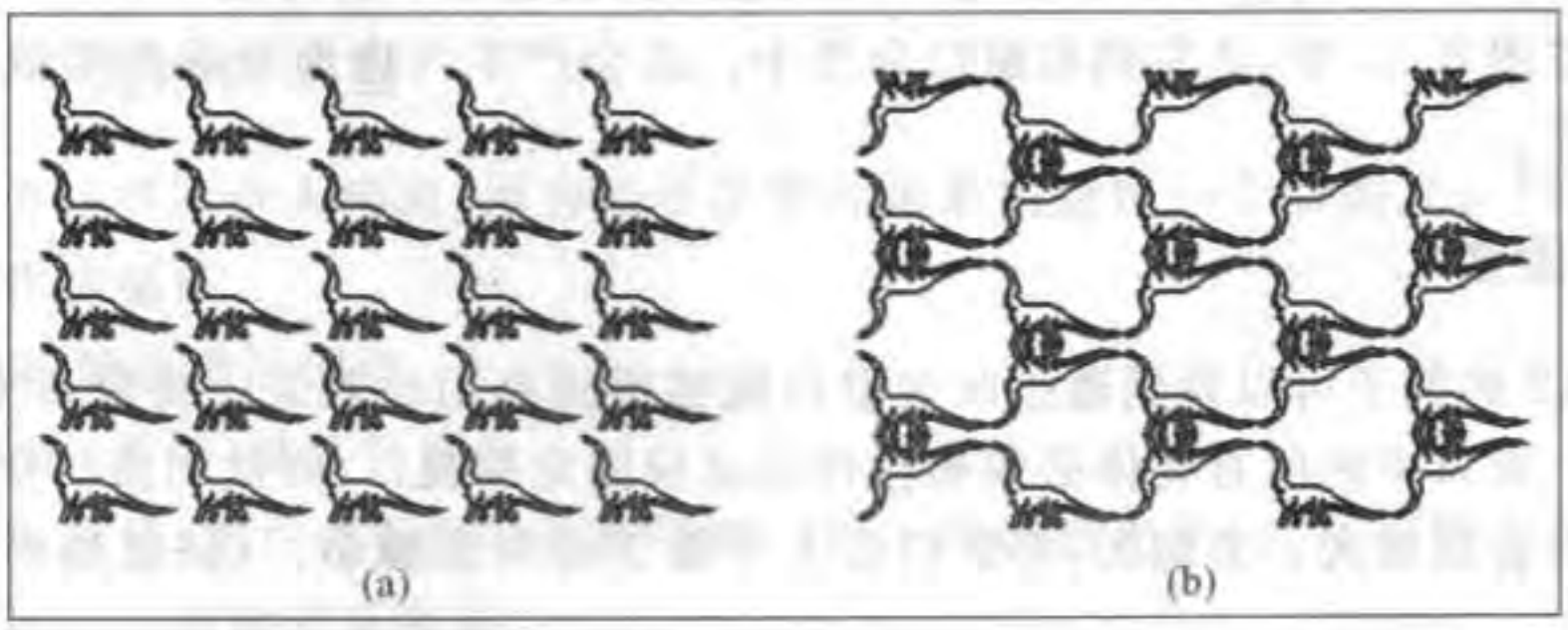


图 3.11　用恐龙的副本平铺屏幕

```
            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
```

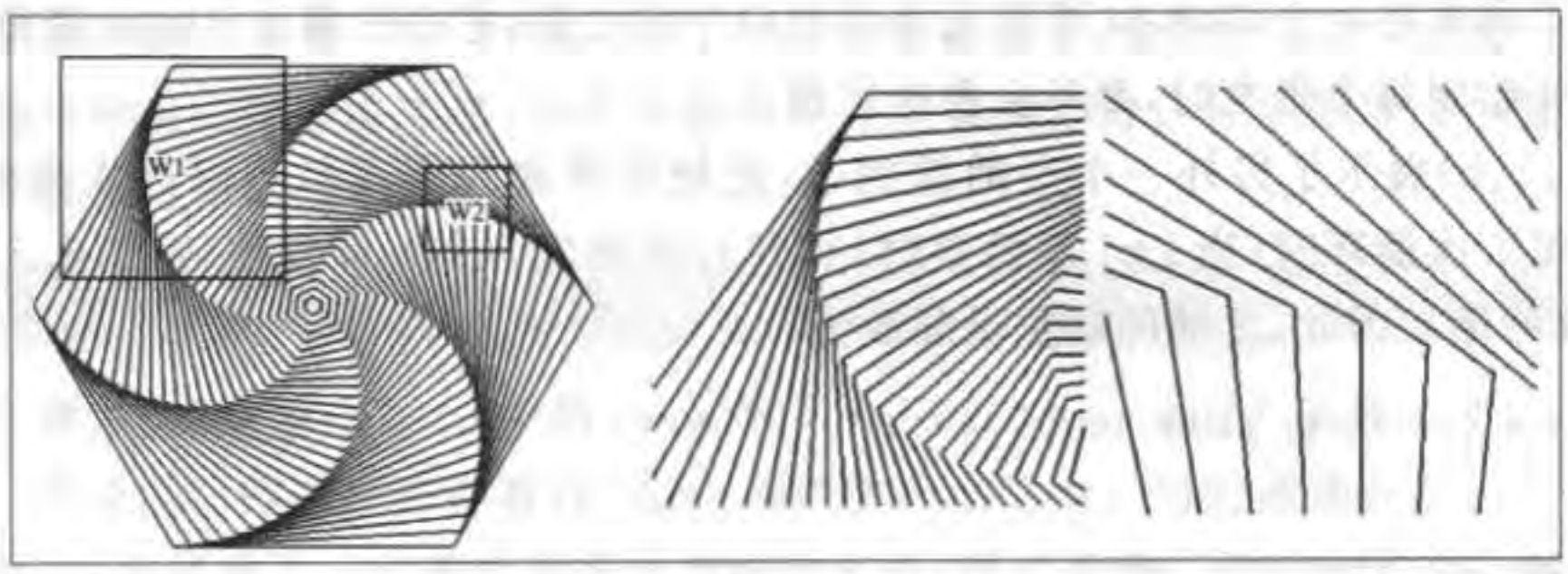


图 3.12 用窗口裁剪图像的一部分

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。

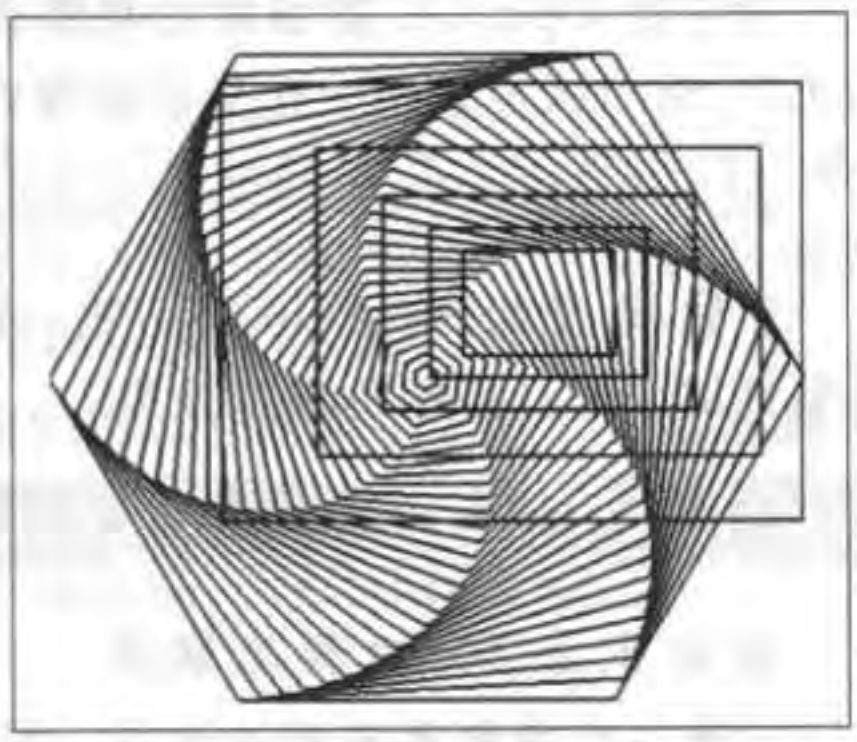


图 3.13　六边形漩涡的放大

```
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
```

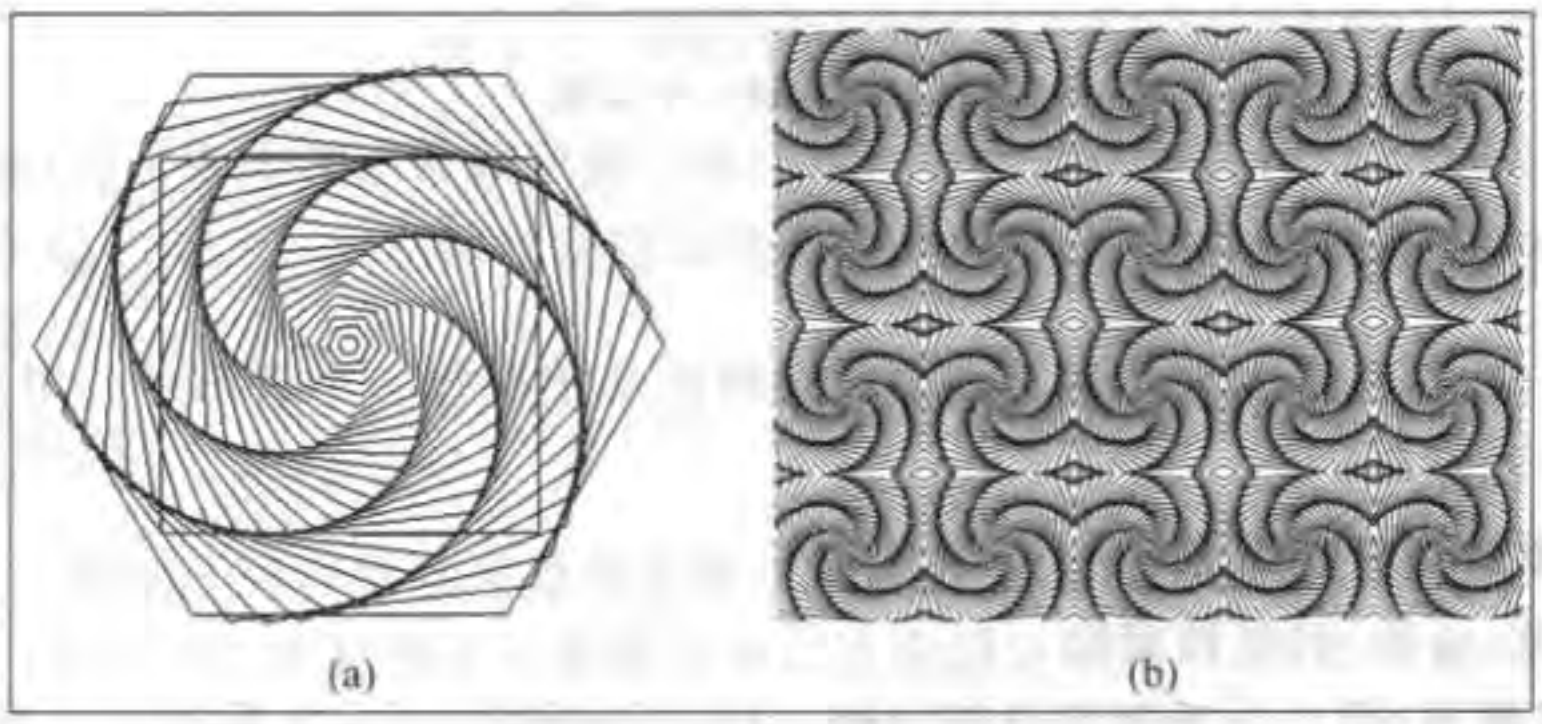


图 3.15

(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。

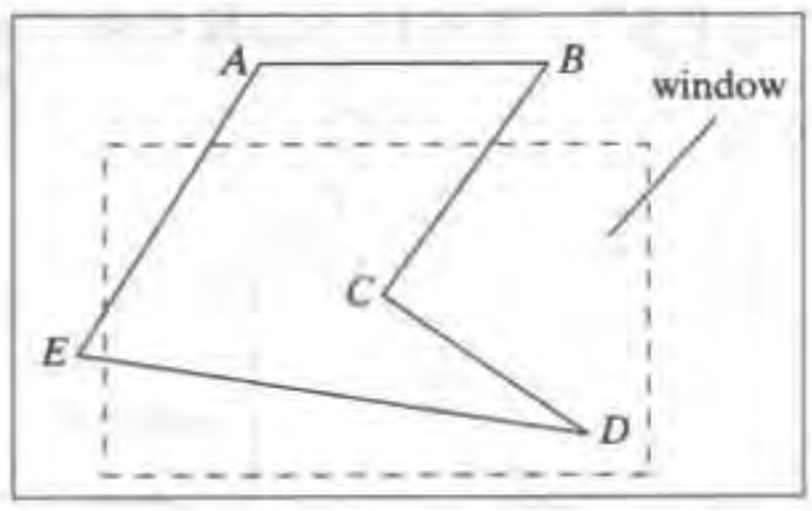


图 3.16　用窗口边界裁剪线

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。

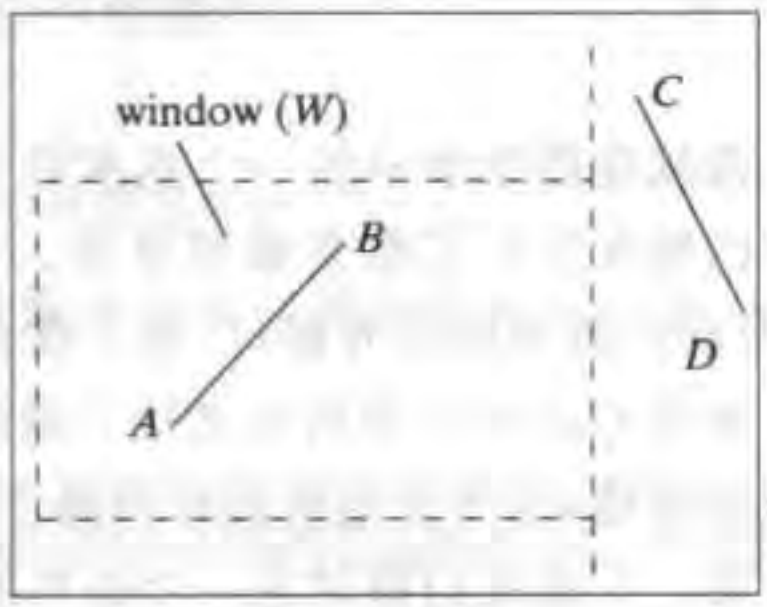


图 3.17 线段的平凡接受和平凡拒绝

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"。

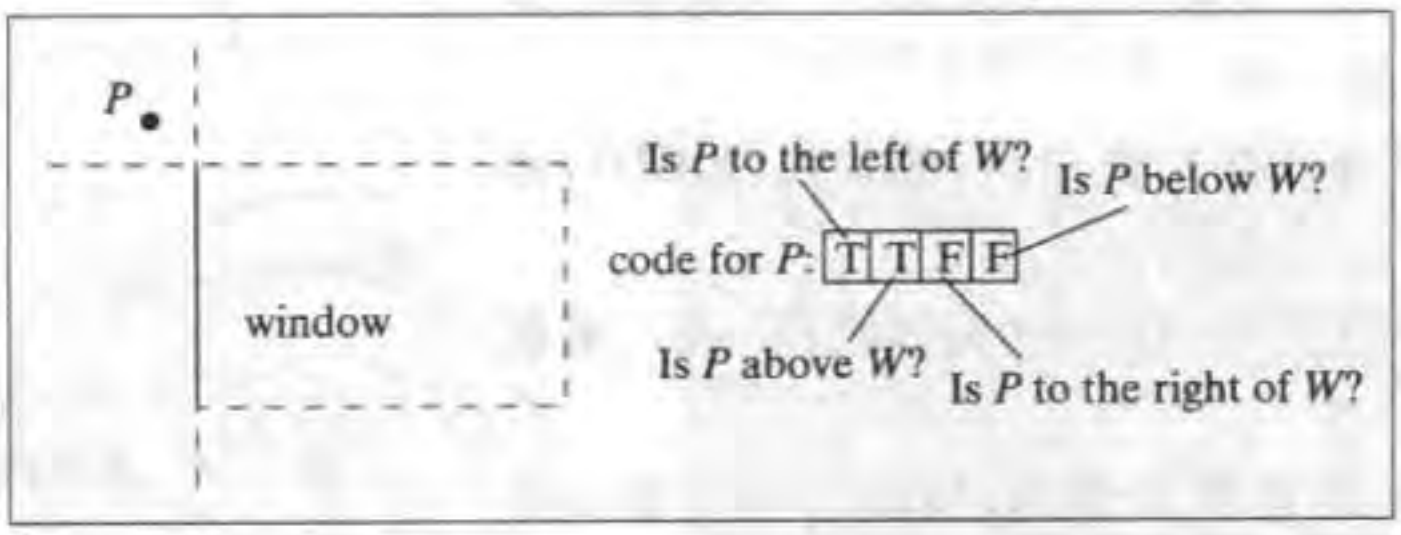


图 3.18 根据 P 在窗口的位置,对 P 做编码

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。

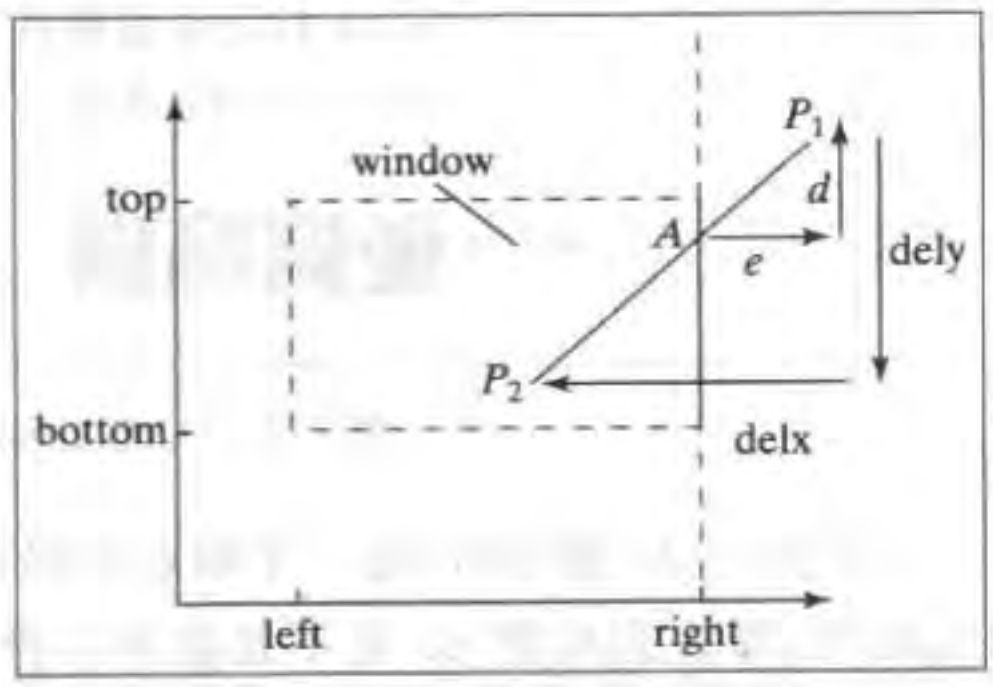


图 3.20　在边界处裁剪线段

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。

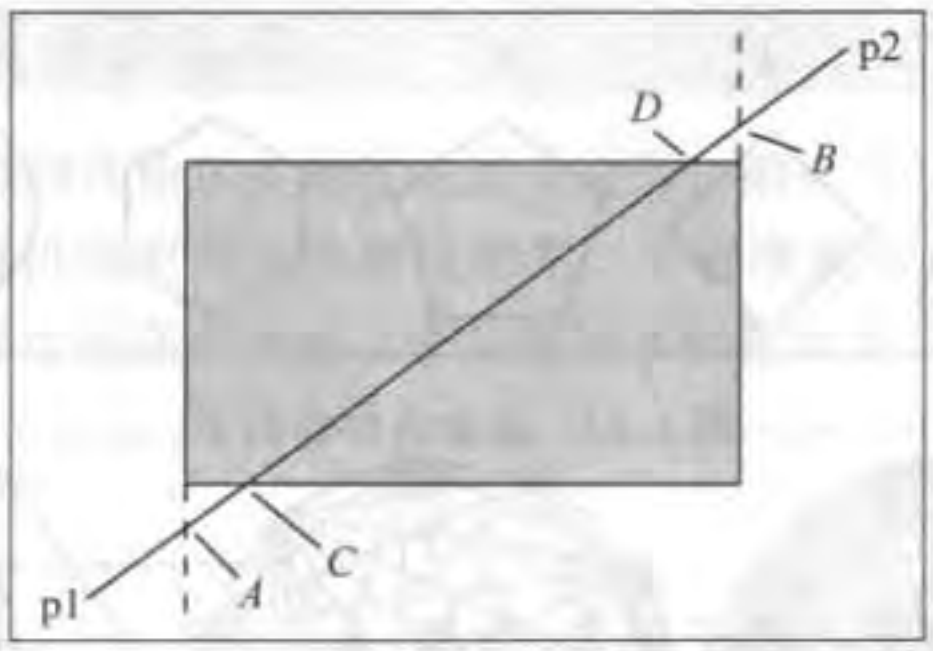


图 3.22　需要四次裁减的线段

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

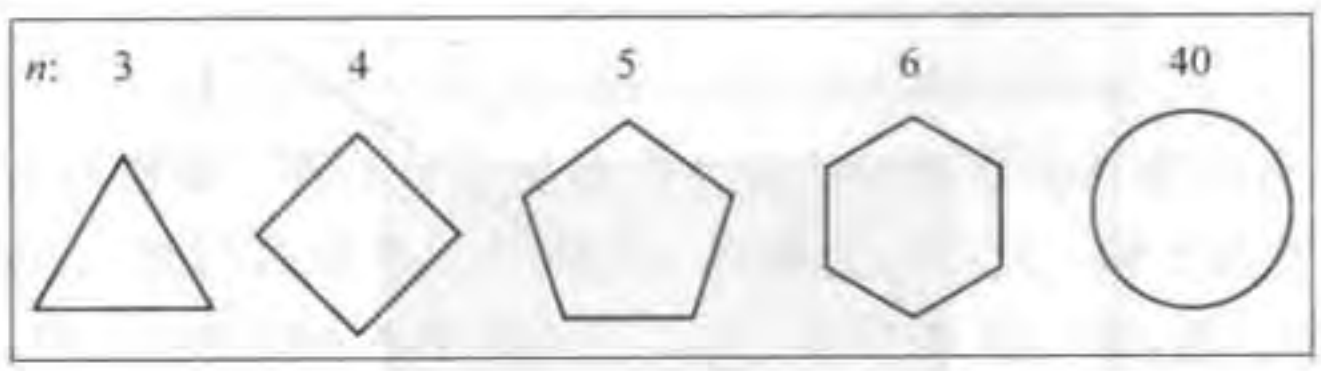


图 3.23 正多边形的例子

顶点,它们等距离的排放。

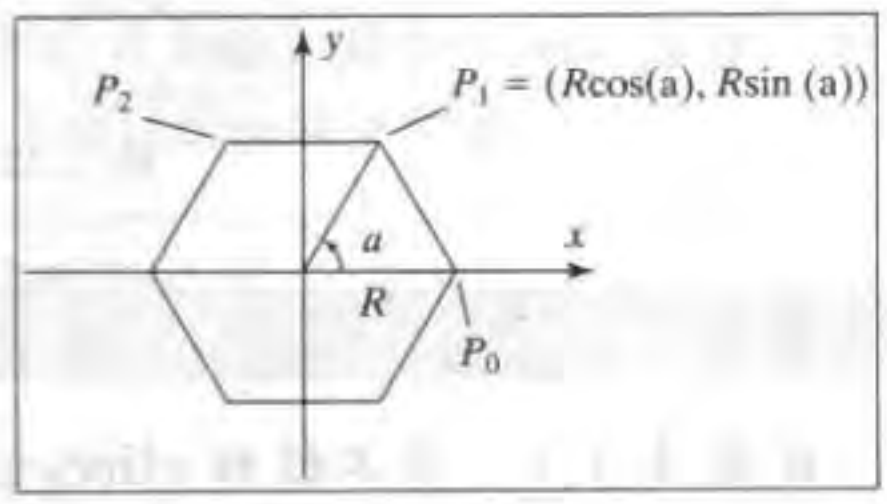


图 3.24 正 6 边形的顶点

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。

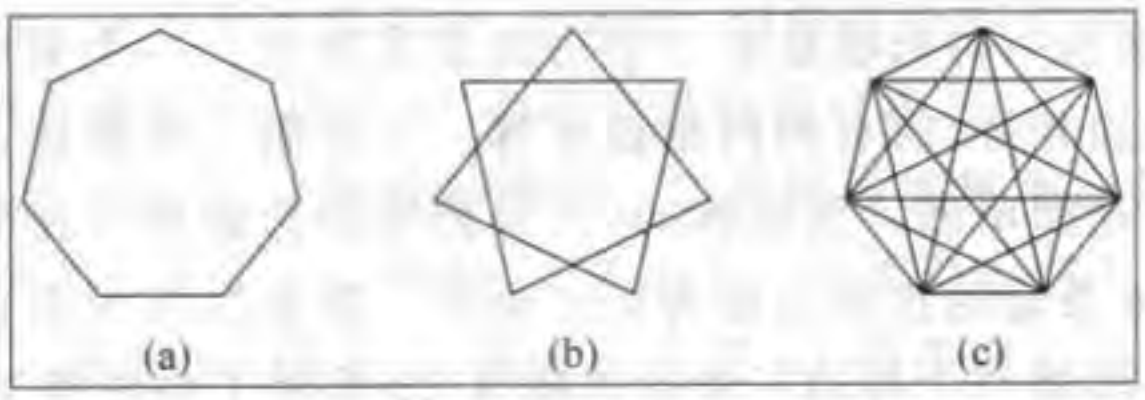


图 3.25

(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。

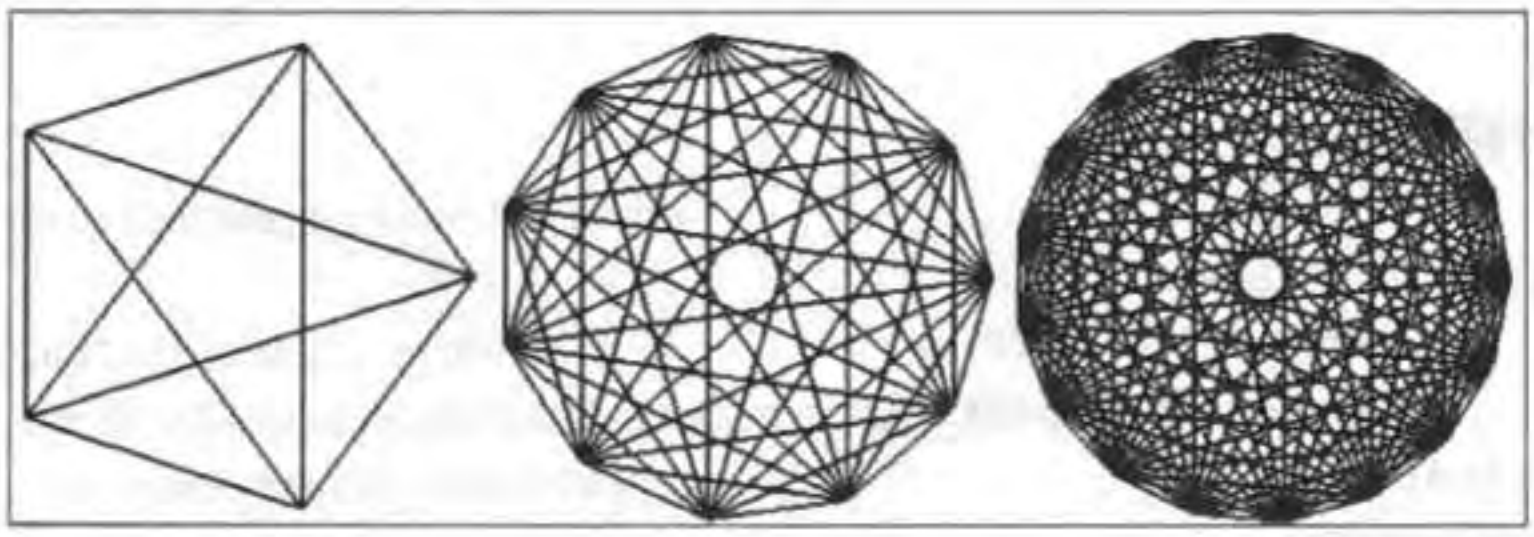


图 3.26　5、11 和 17 花环

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。

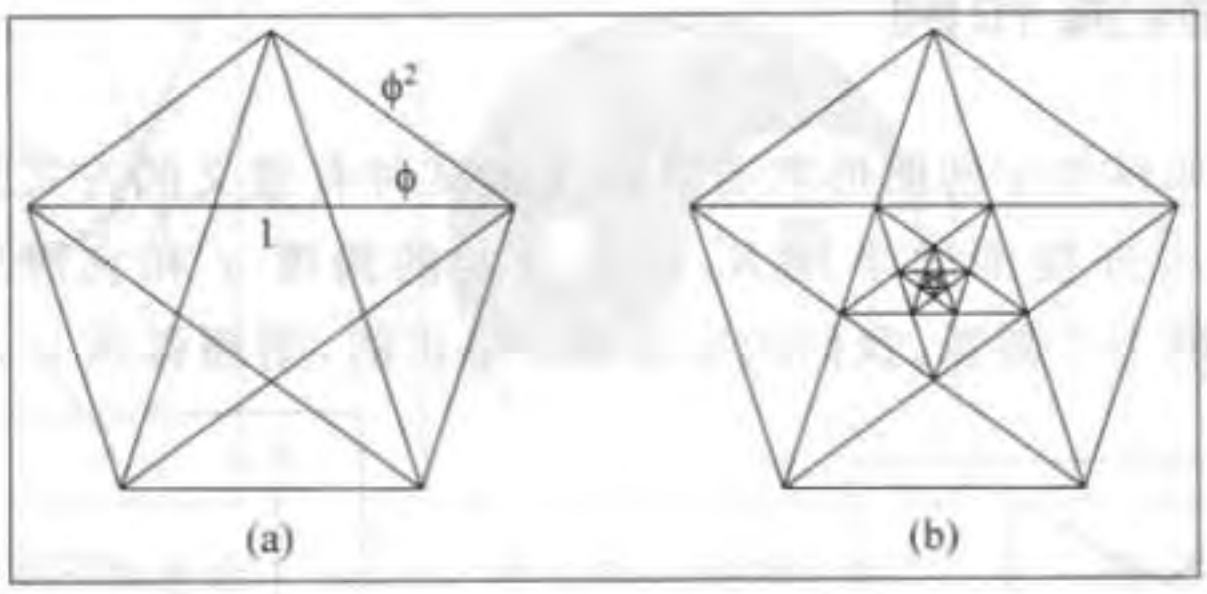


图 3.28　5 花环和它的无穷嵌套——五边形和五角星

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。

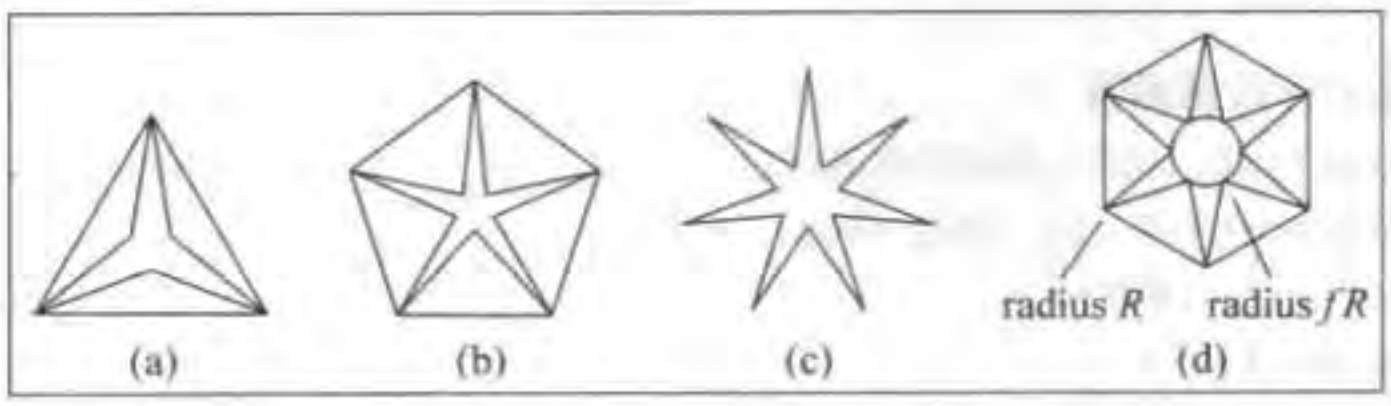


图 3.29 基于正 n 边形的有趣的图形角星

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

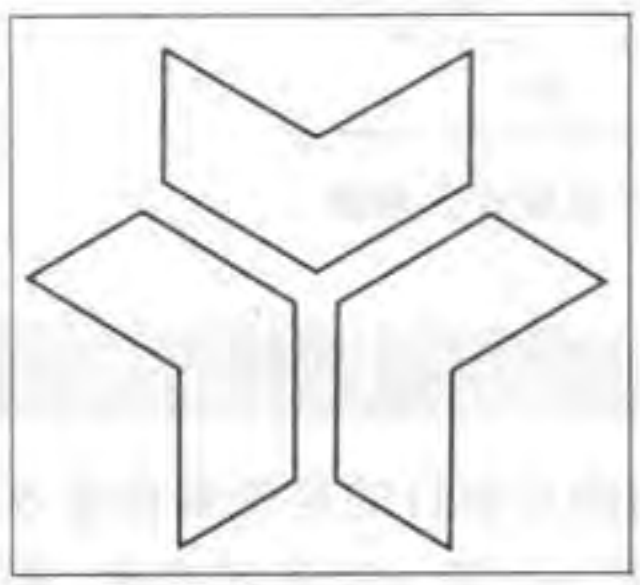


图 3.30 Massachusetts 大学的标志

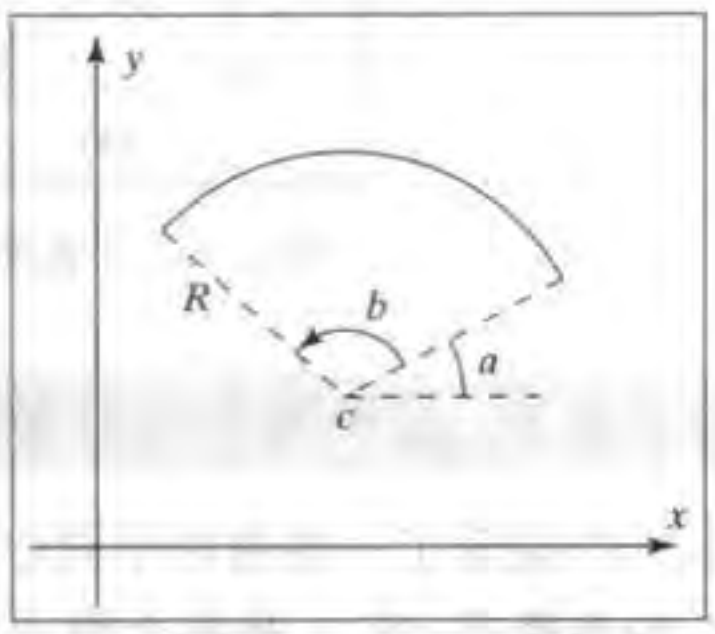


图 3.31 圆弧的定义

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。

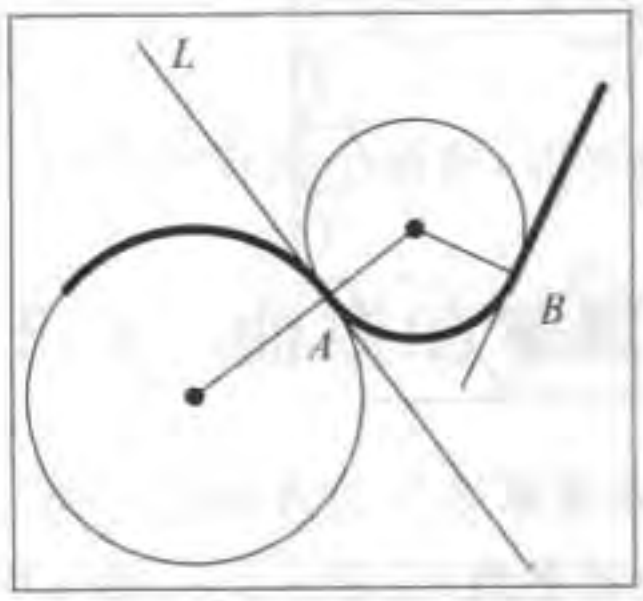


图 3.32　用切线将两个圆融合

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。

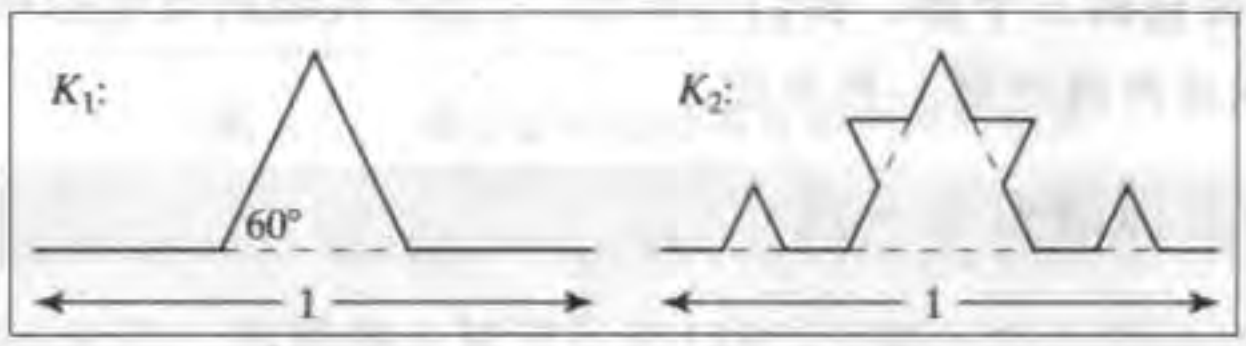


图 3.35 Koch 曲线的两代

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。

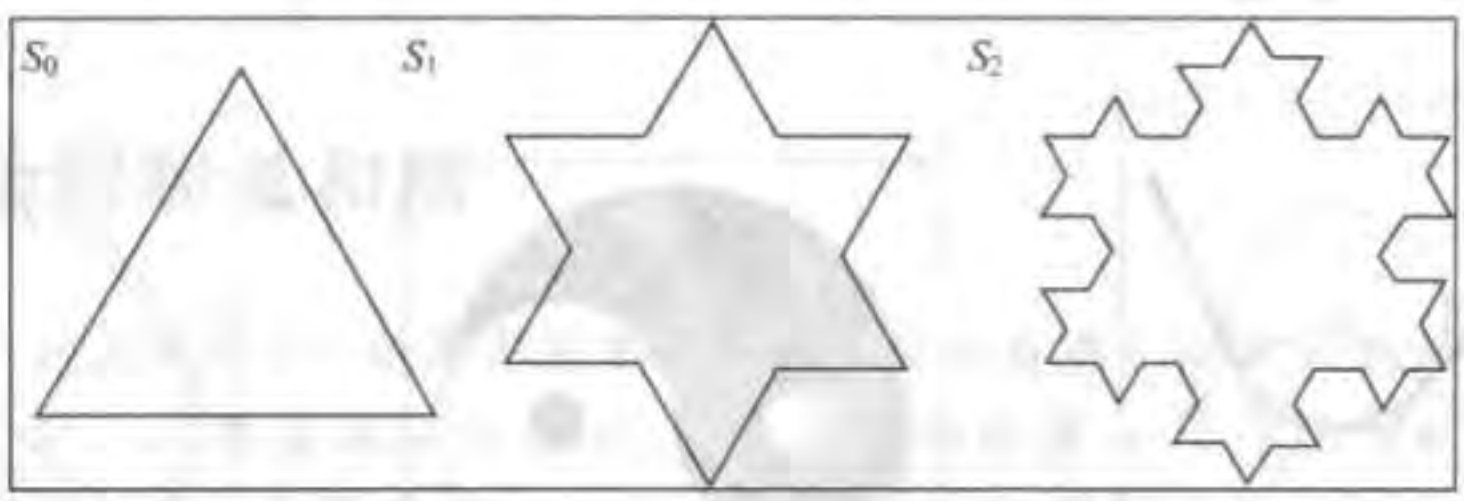


图 3.36 Koch 雪花的前几代

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 $y(.)$。

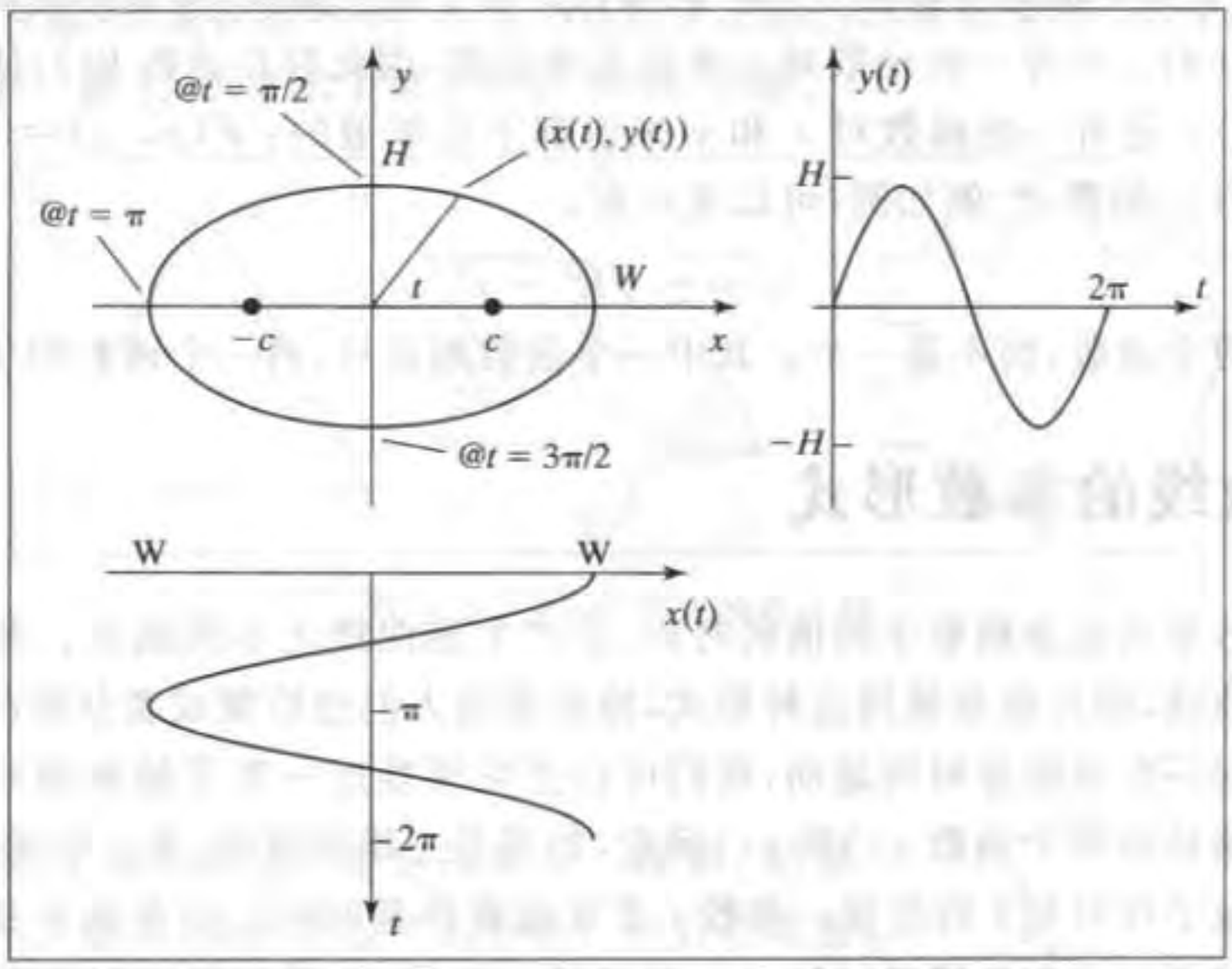


图 3.39 用参数方式表示椭圆

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。

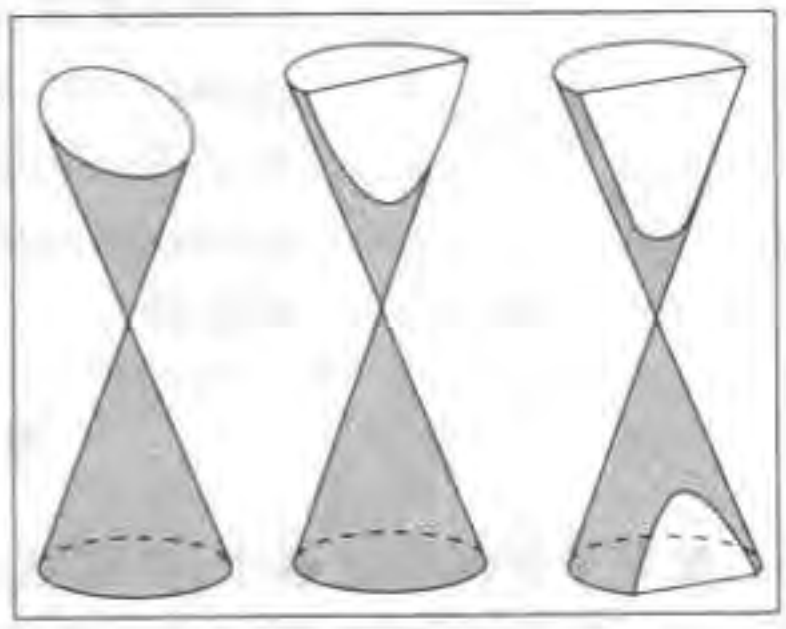


图 3.40　经典的圆锥曲线

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。

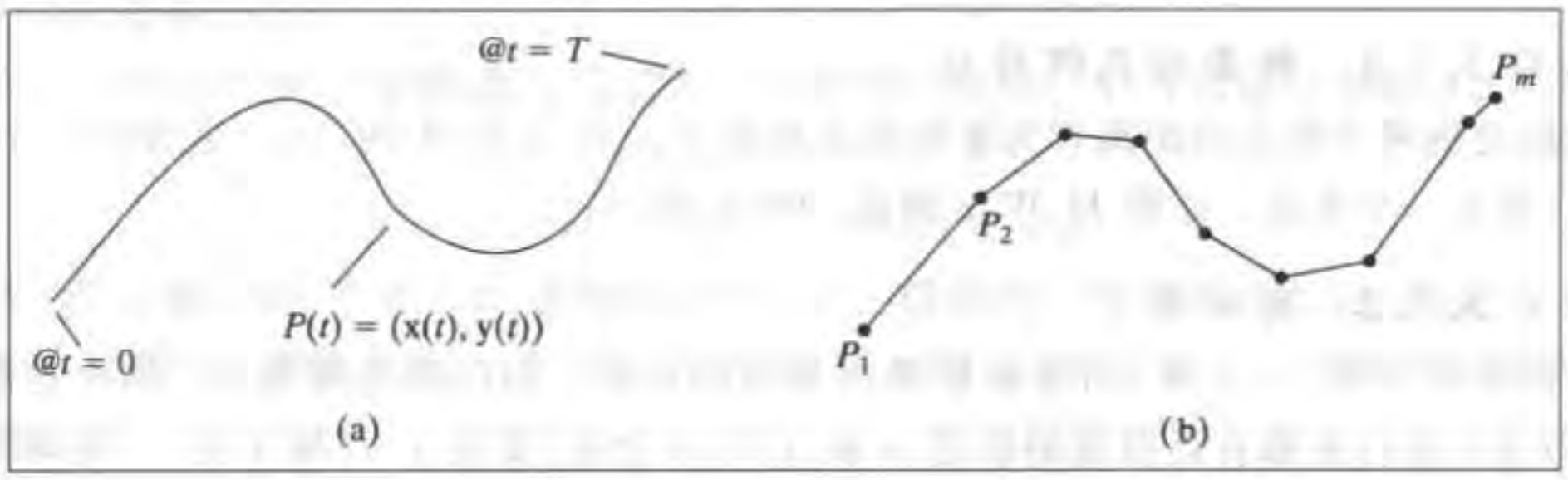


图 3.41 用折线近似曲线

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

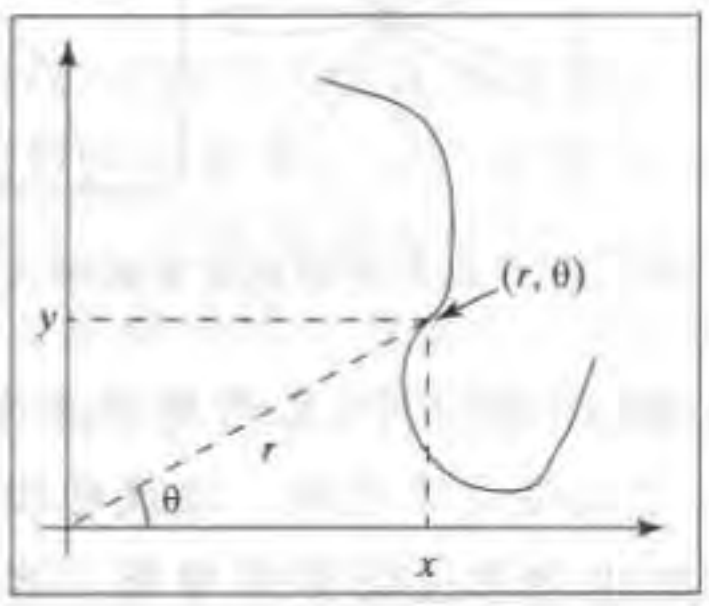


图 3.44　极坐标轴

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 $A2\pi$)。

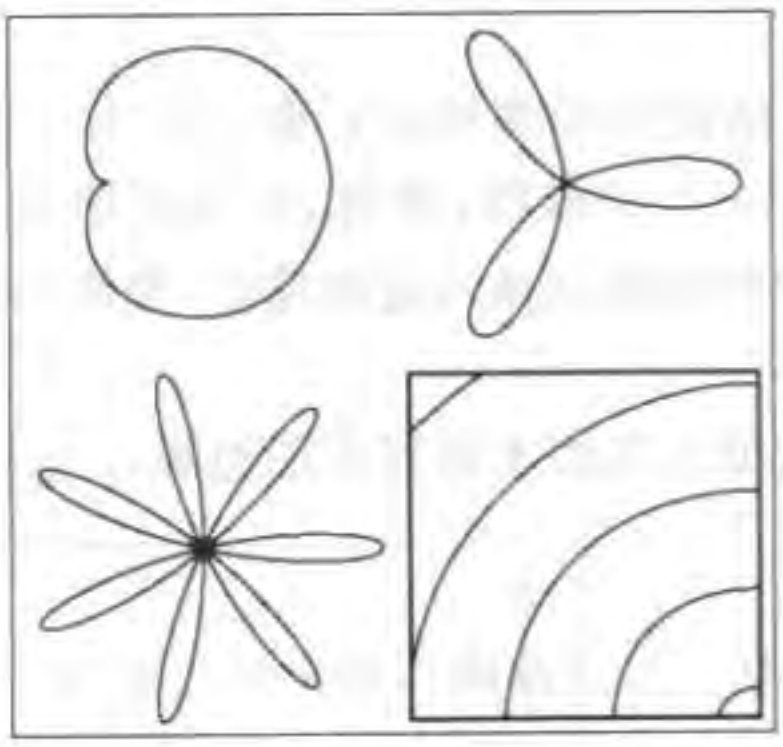


图 3.45 具有简单的极坐标形式的曲线的例子

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]。

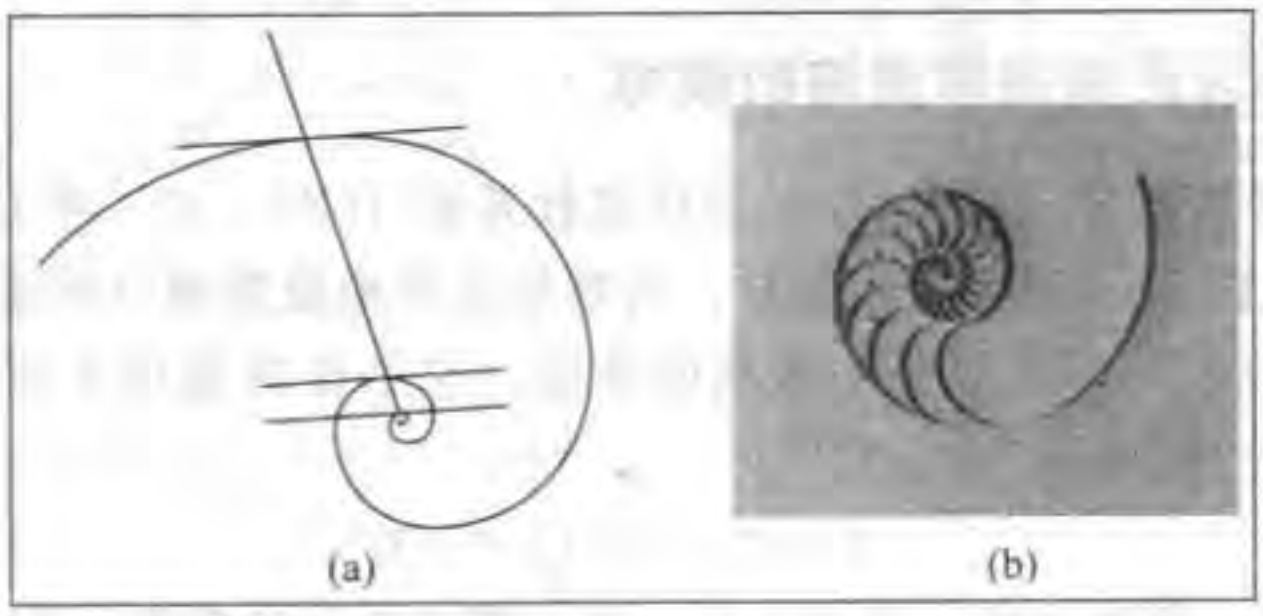


图　3.47

（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?)

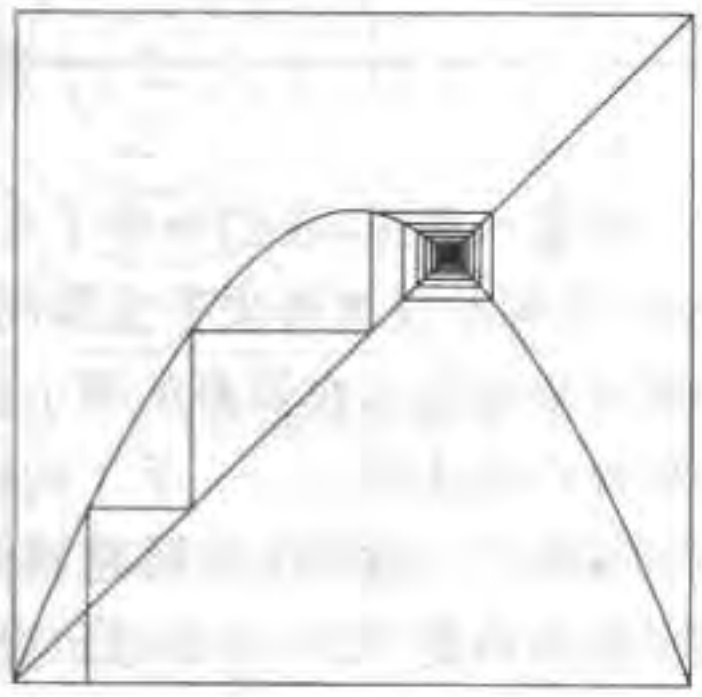


图 3.48 $\lambda=0.7$ 时的逻辑图

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

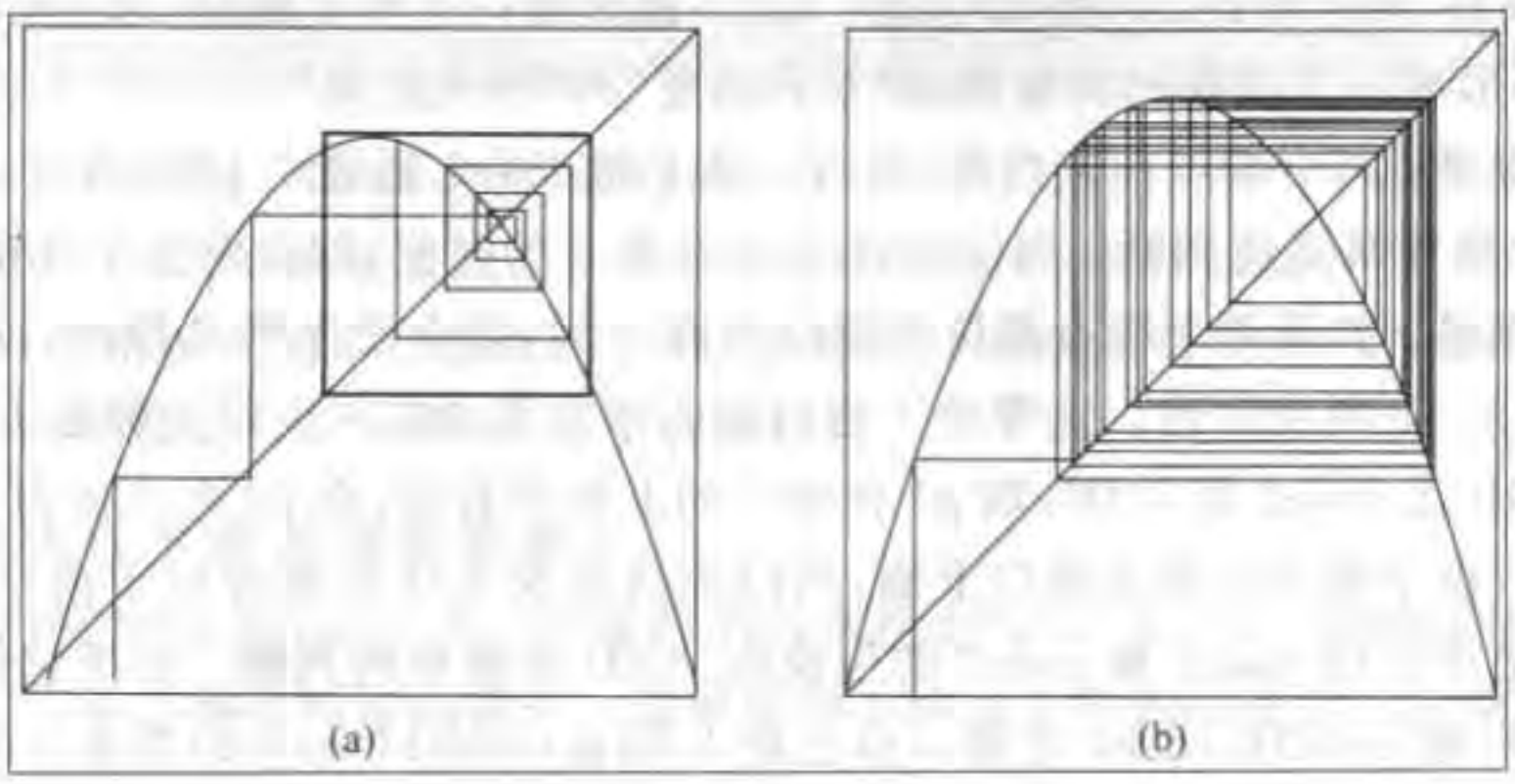


图 3.49

(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？)

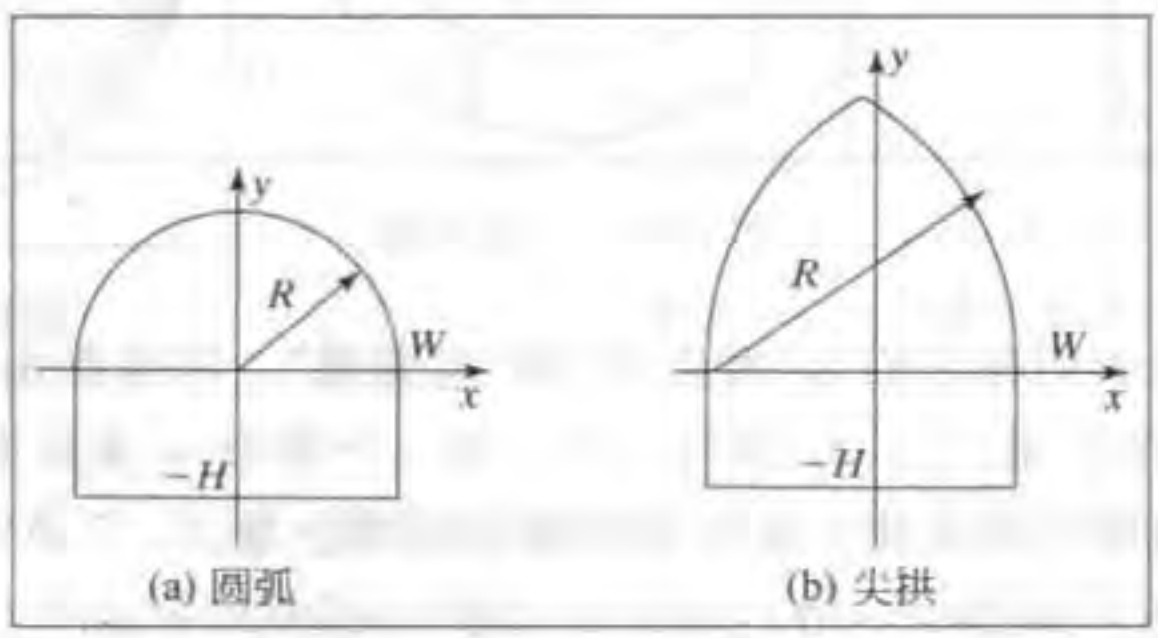


图 3.52　两种基本的弧形

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。

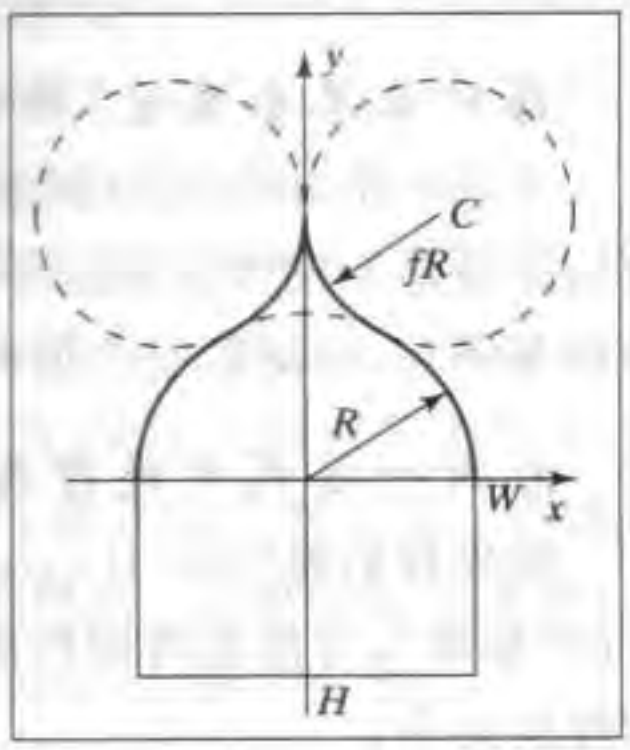


图 3.53 葱形穹顶

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。

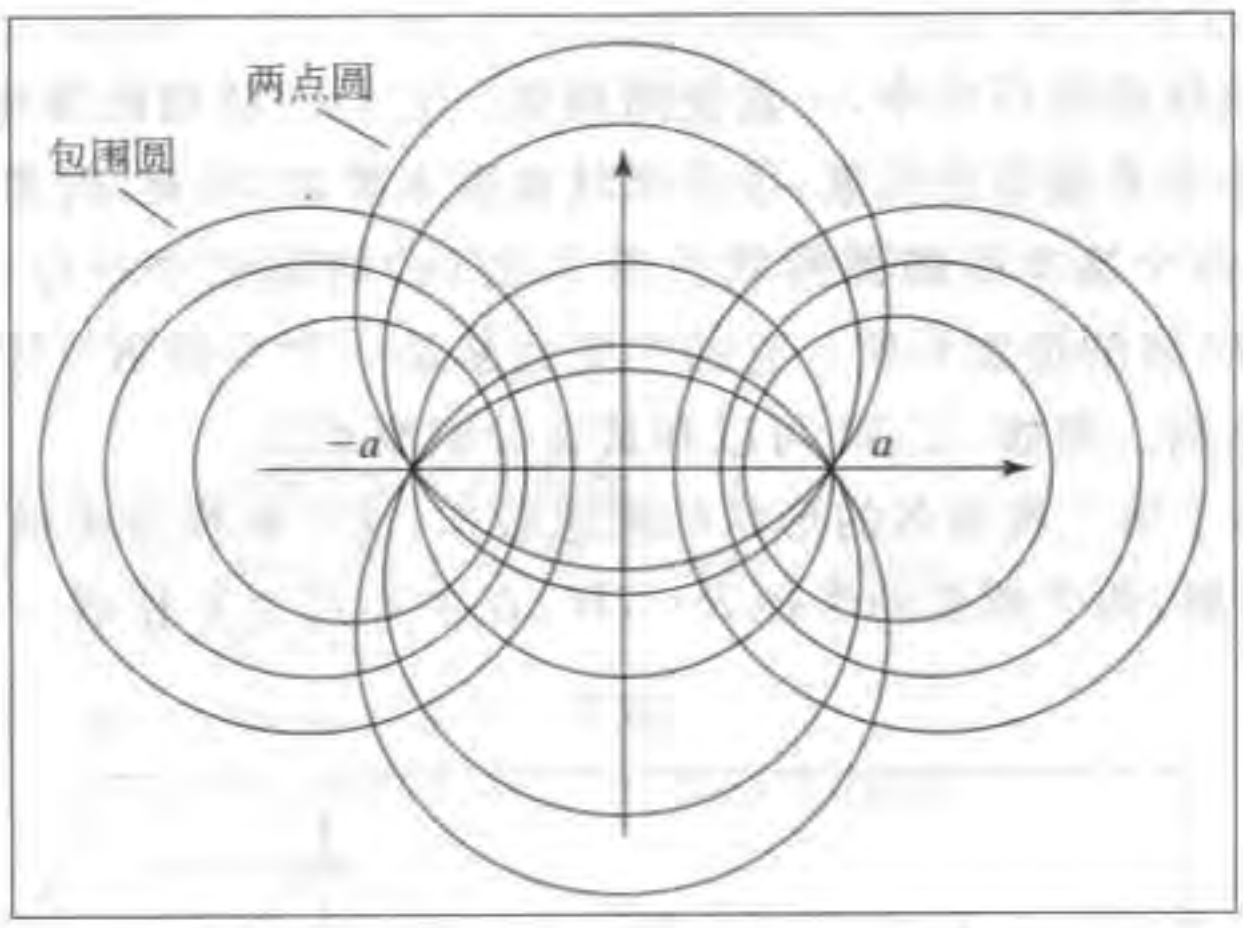


图 3.54 正交的圆

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。

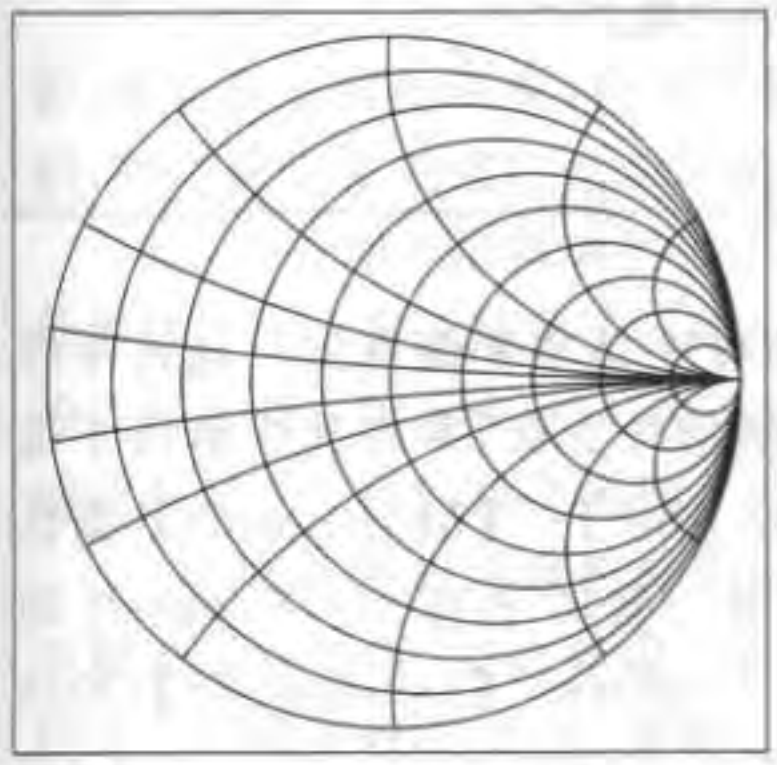


图 3.55　Smith 图表

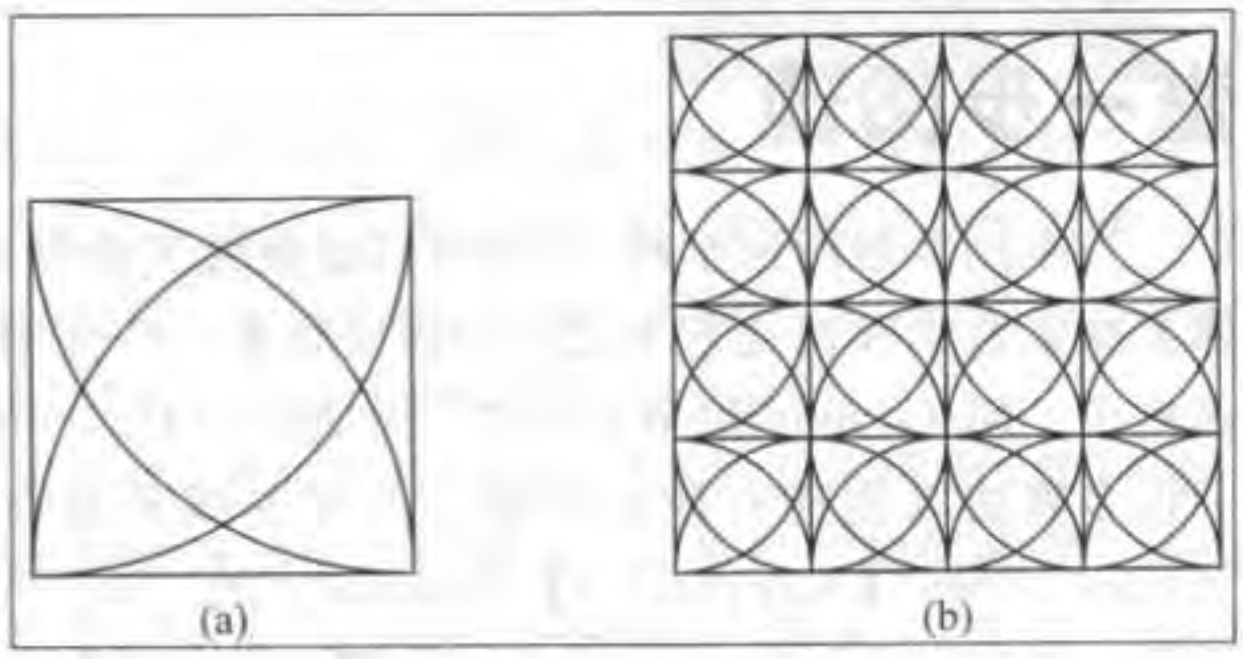


图 3.56　一个图元和用它平铺后的结果

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。

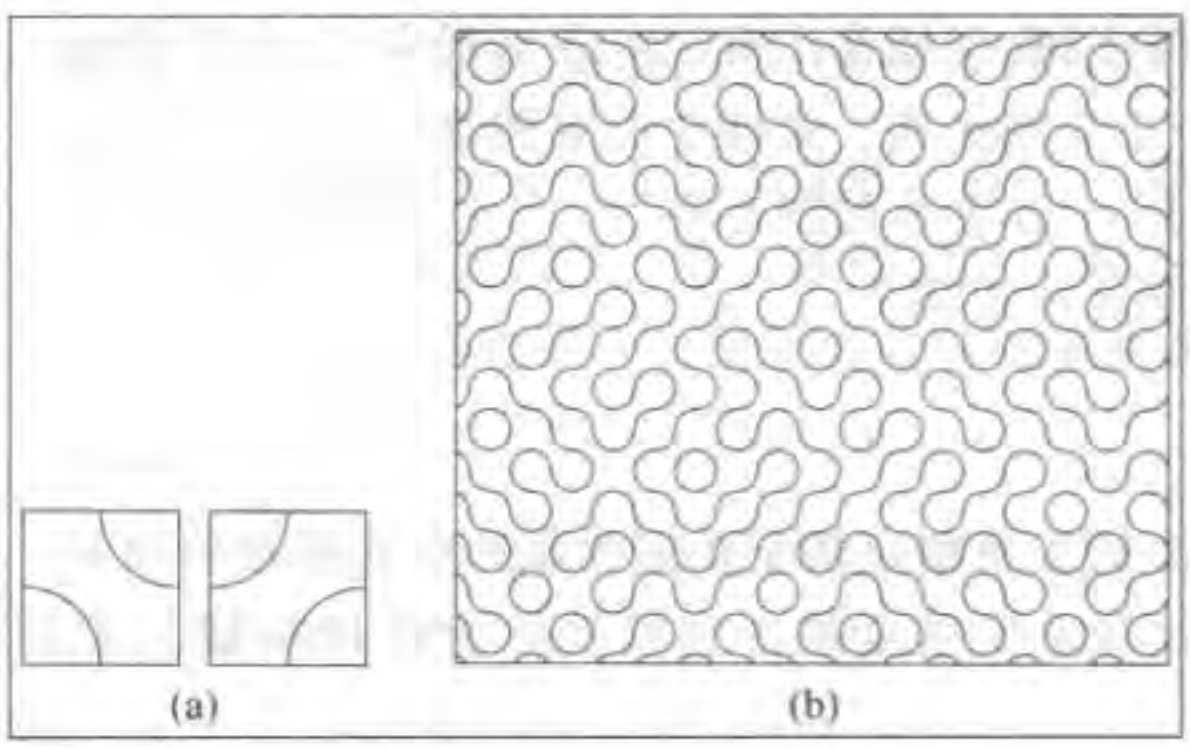


图 3.57 Truchet 平铺

(a) 两个瓦片;(b) 一个 Truchet 图案结果

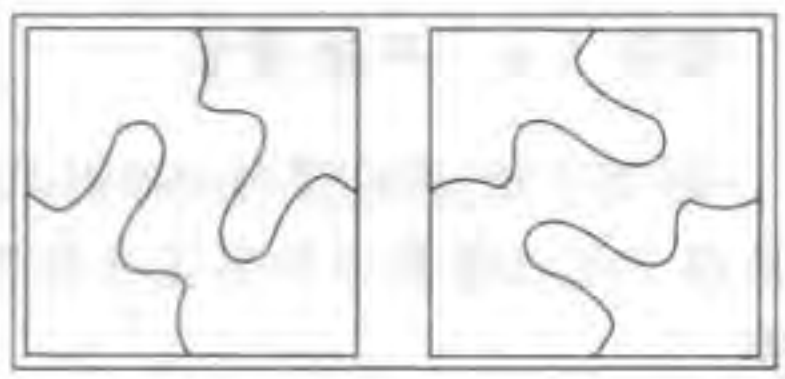


图 3.58 Truchet 平铺的扩展

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。

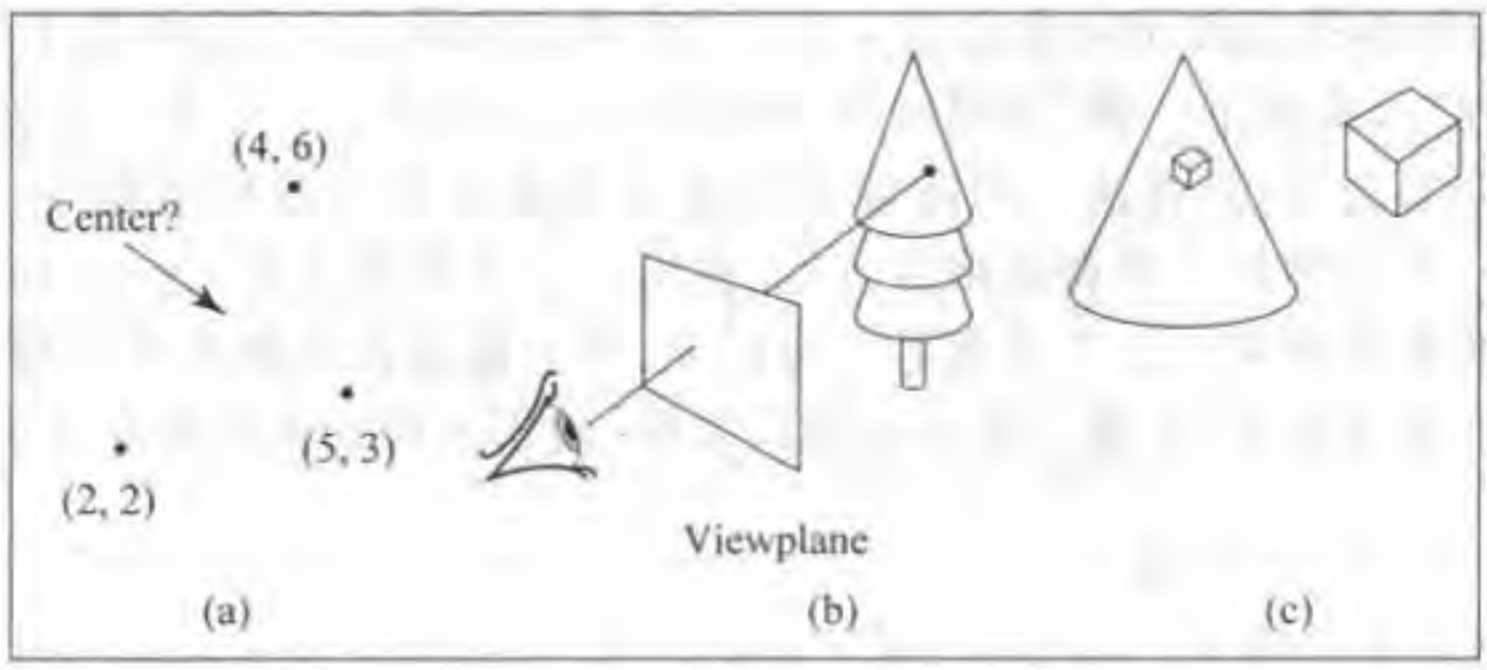


图 4.1 三个非常适合用向量解决的几何问题

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。

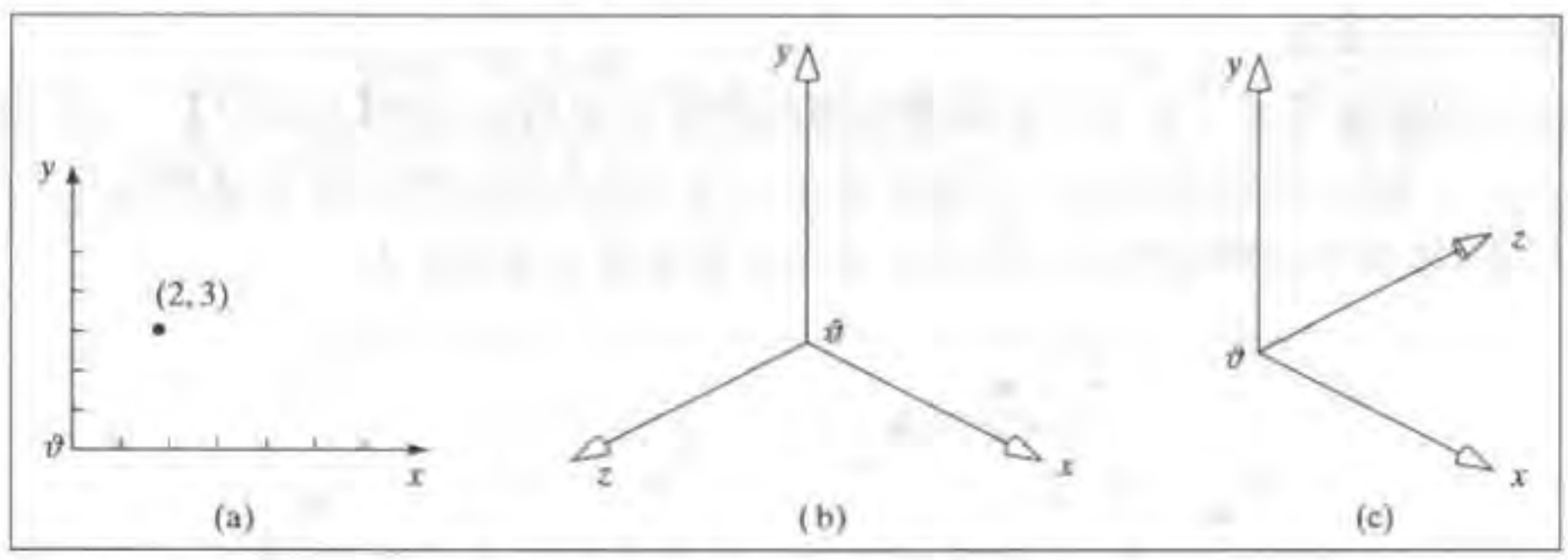


图 4.2　常用的二维和三维坐标系

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!

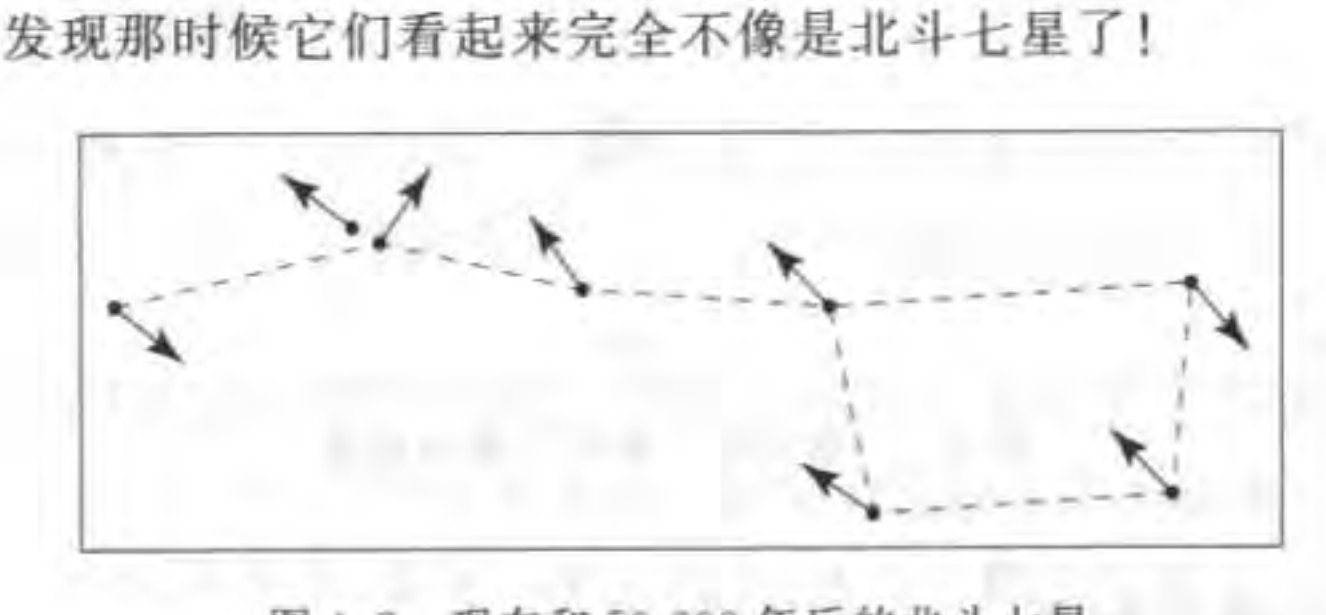


图 4.3 现在和 50 000 年后的北斗七星

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。

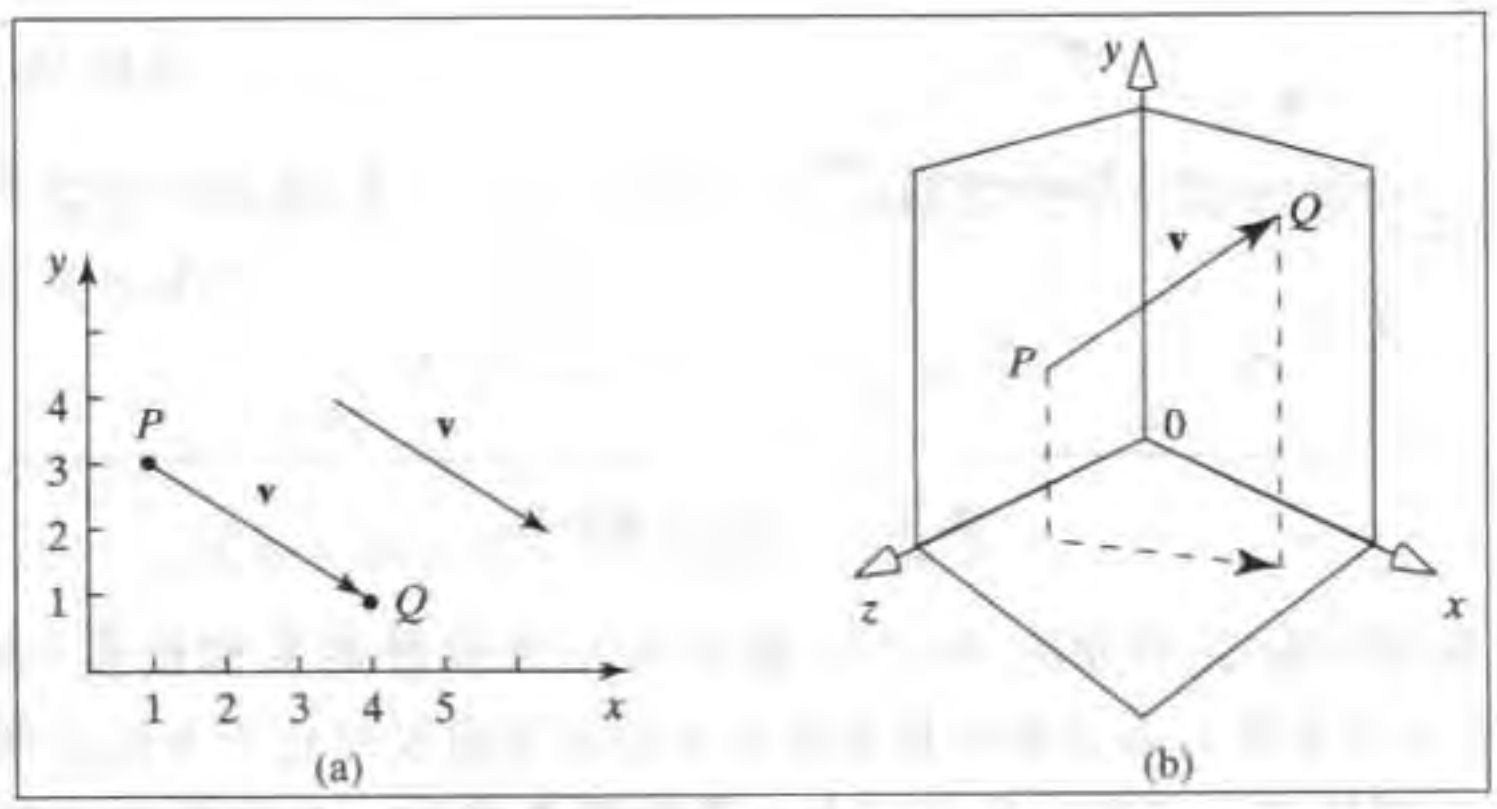


图 4.4　表示位移的向量

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。

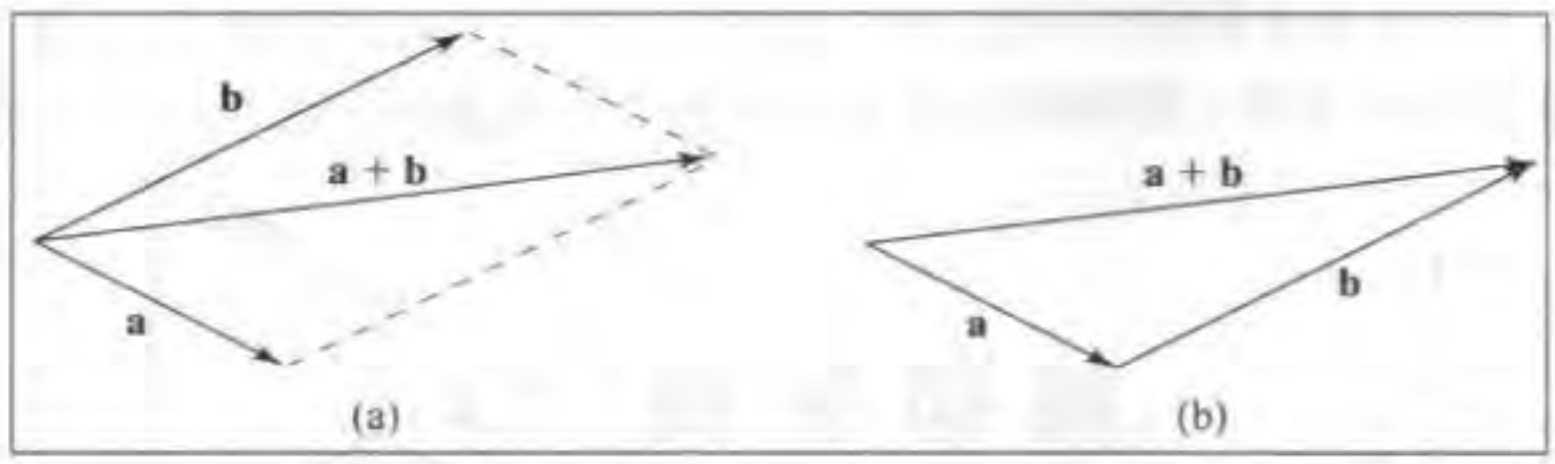


图 4.5 两个向量的和

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。

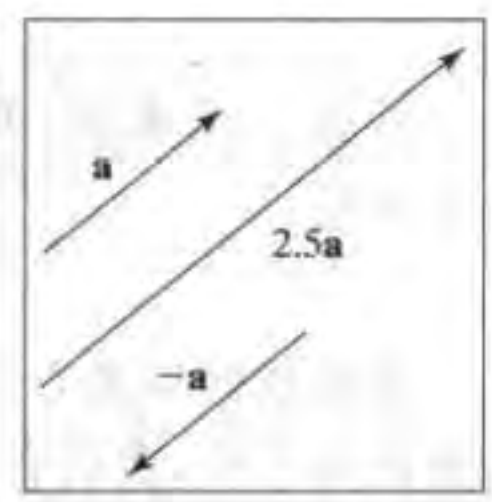


图 4.6 向量的数乘

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**。

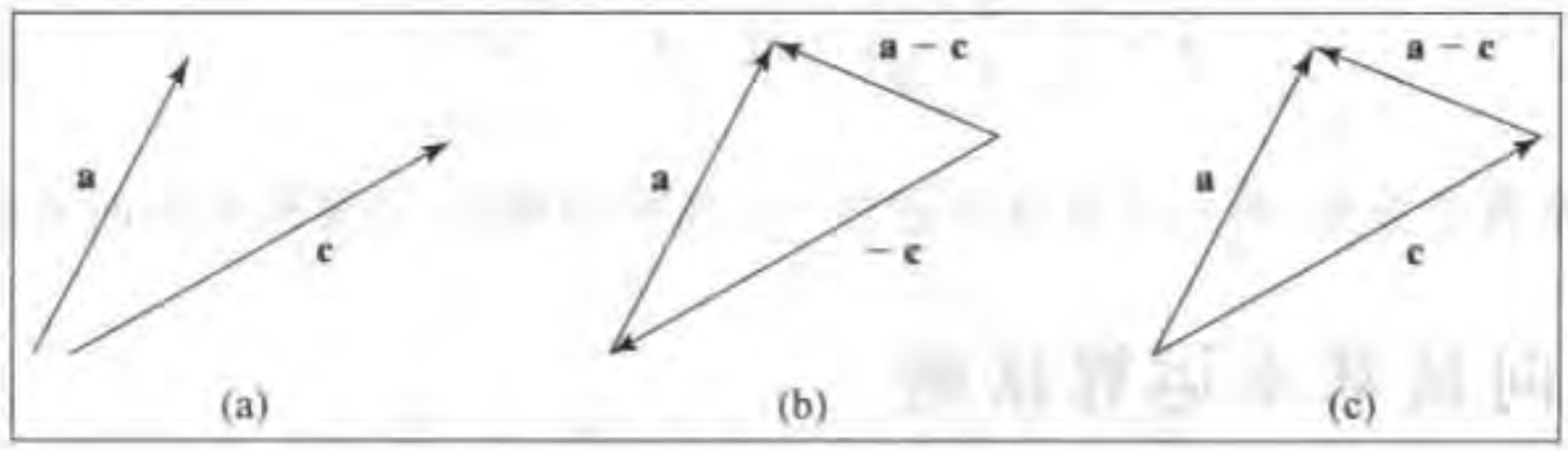


图 4.7 向量相减

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：

$$\mathbf{b}=(|\mathbf{b}|\cos\phi_b,\ |\mathbf{b}|\sin\phi_b)$$

$$\mathbf{c}=(|\mathbf{c}|\cos\phi_c,\ |\mathbf{c}|\sin\phi_c)$$

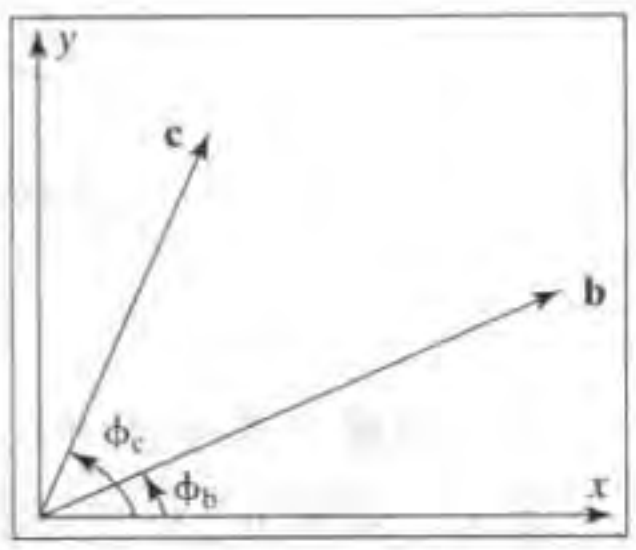


图 4.8　求两个向量的夹角

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。

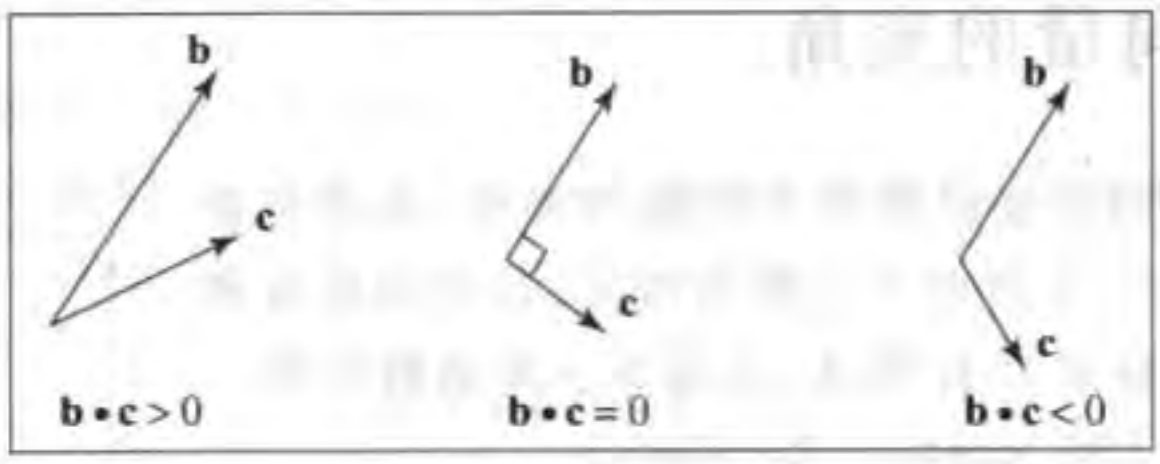


图 4.9 点积的符号

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**。

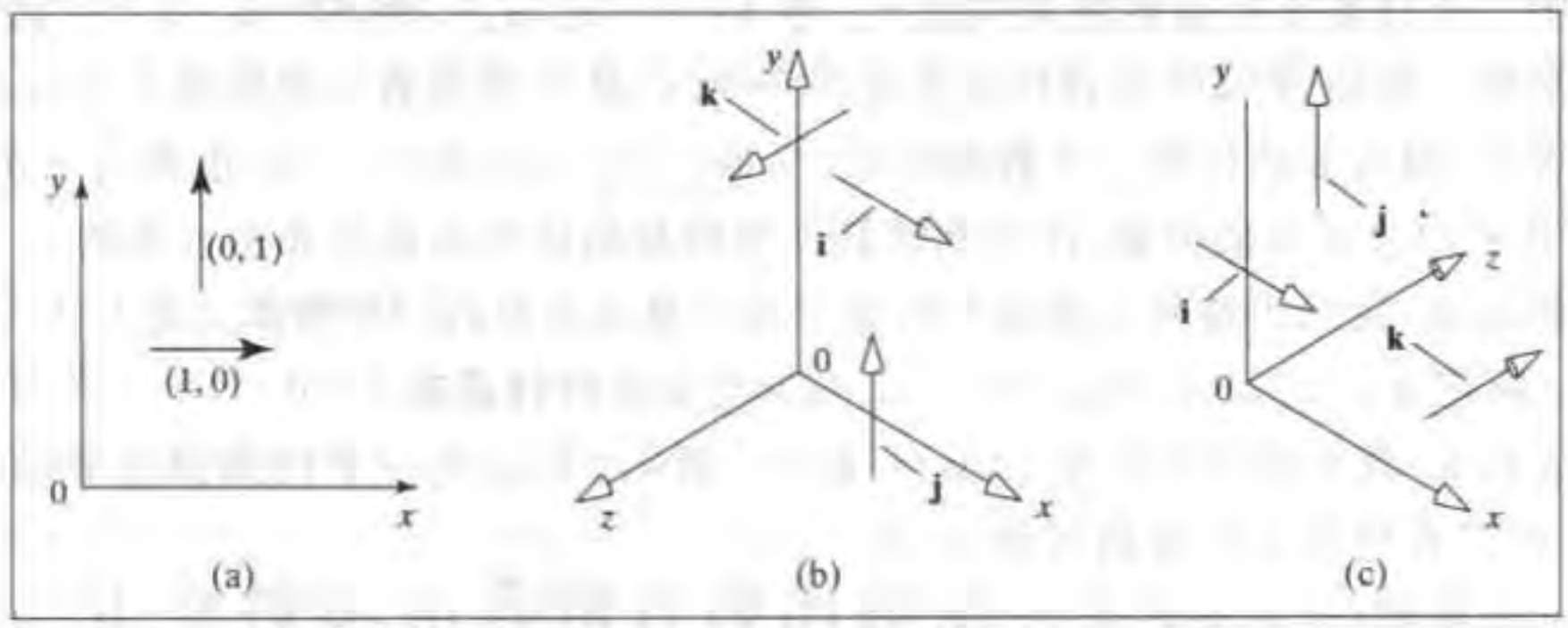


图 4.10　单位标准向量

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)。

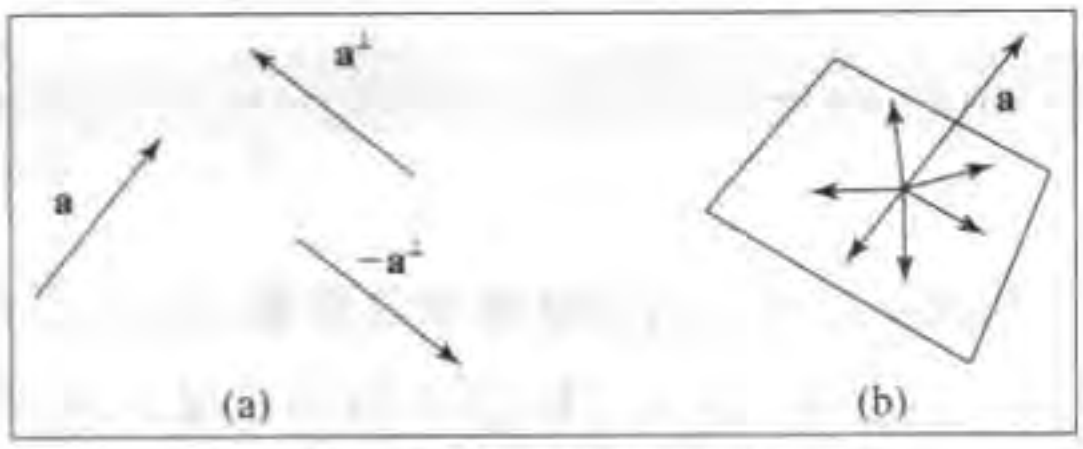


图 4.11 正交与向量 $\mathbf{a}$ 的向量 $\mathbf{a}^\perp$

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？

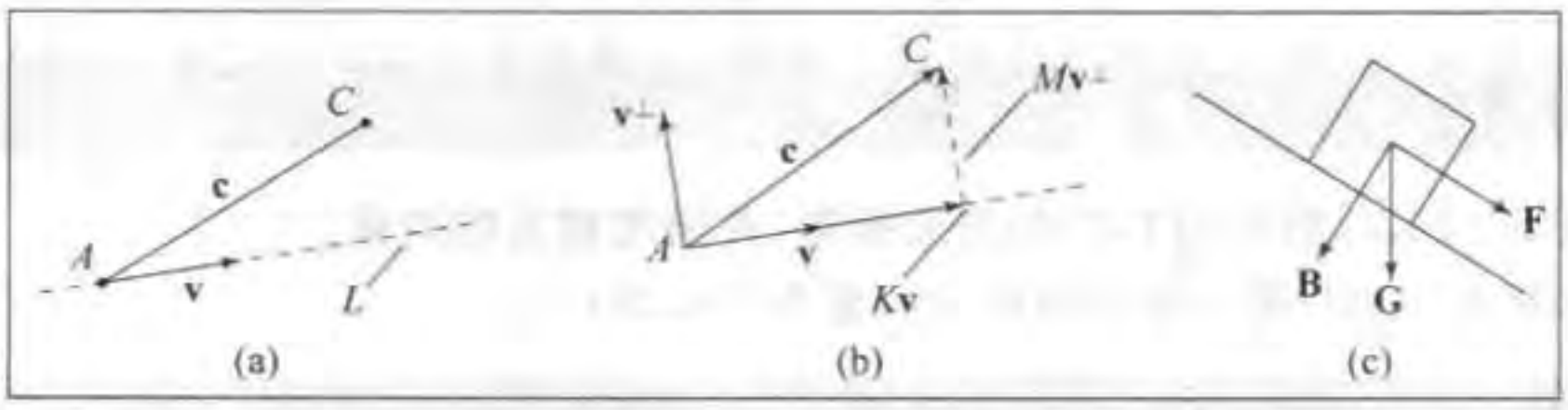


图 4.12　把一个向量分解为两个正交的向量

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)。

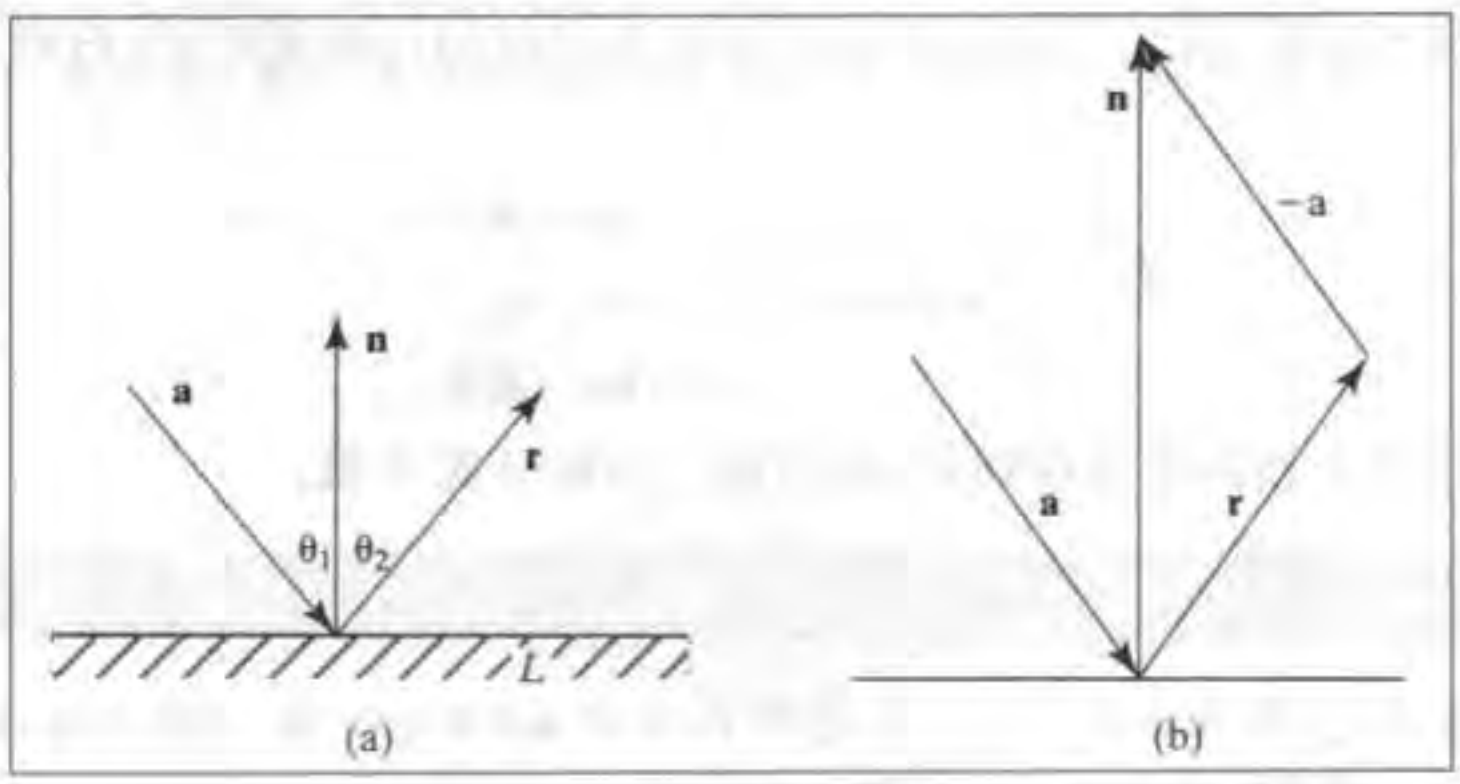


图 4.13　光线被平面反射的过程

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：

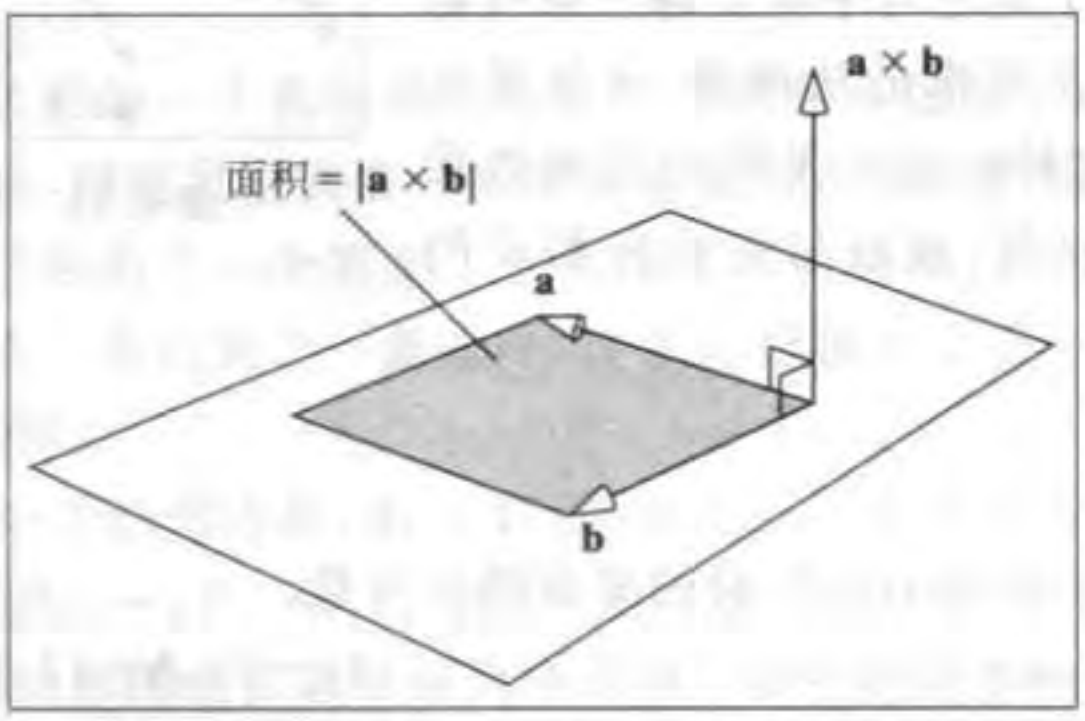


图 4.14　叉积的几何解释

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。

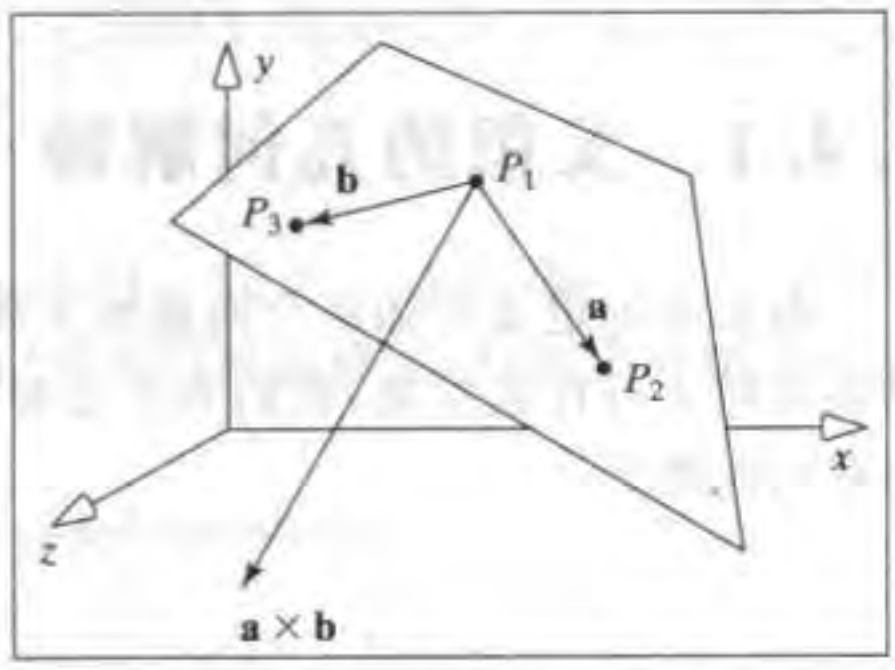


图 4.15 三点确定一个平面

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$(0,1,0)$。

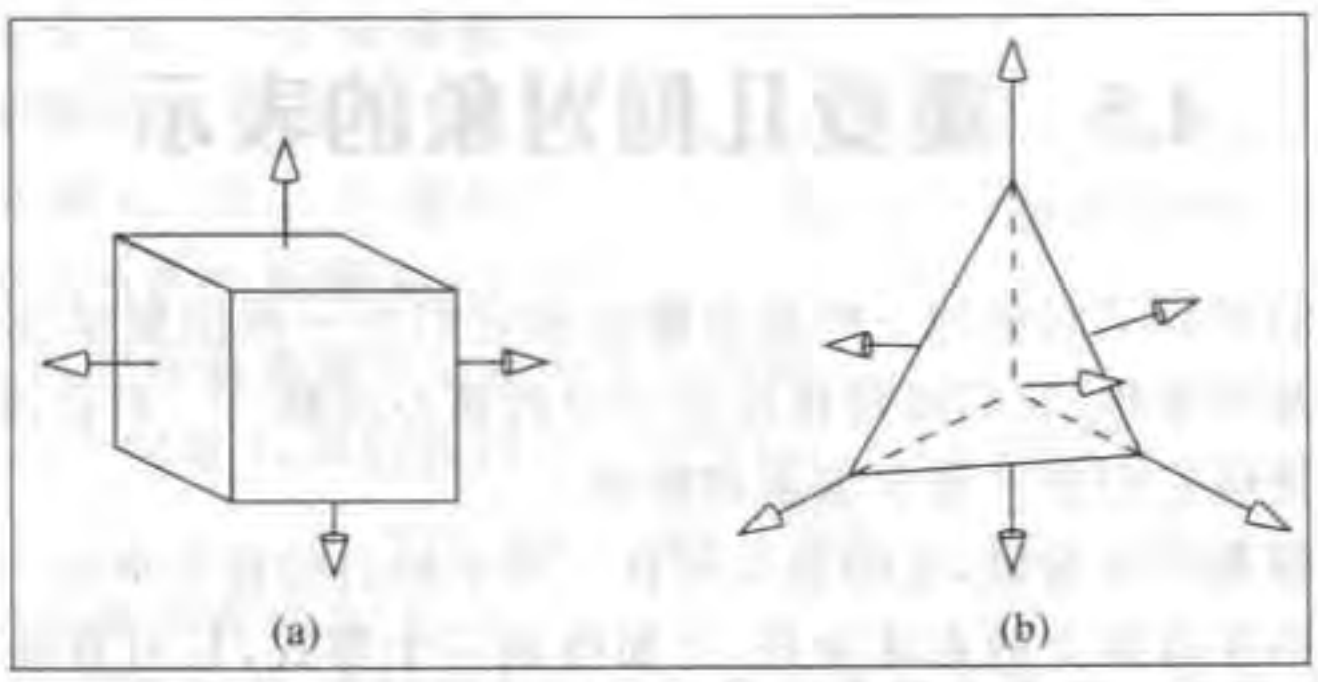


图 4.16　计算每个面的法向量

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:

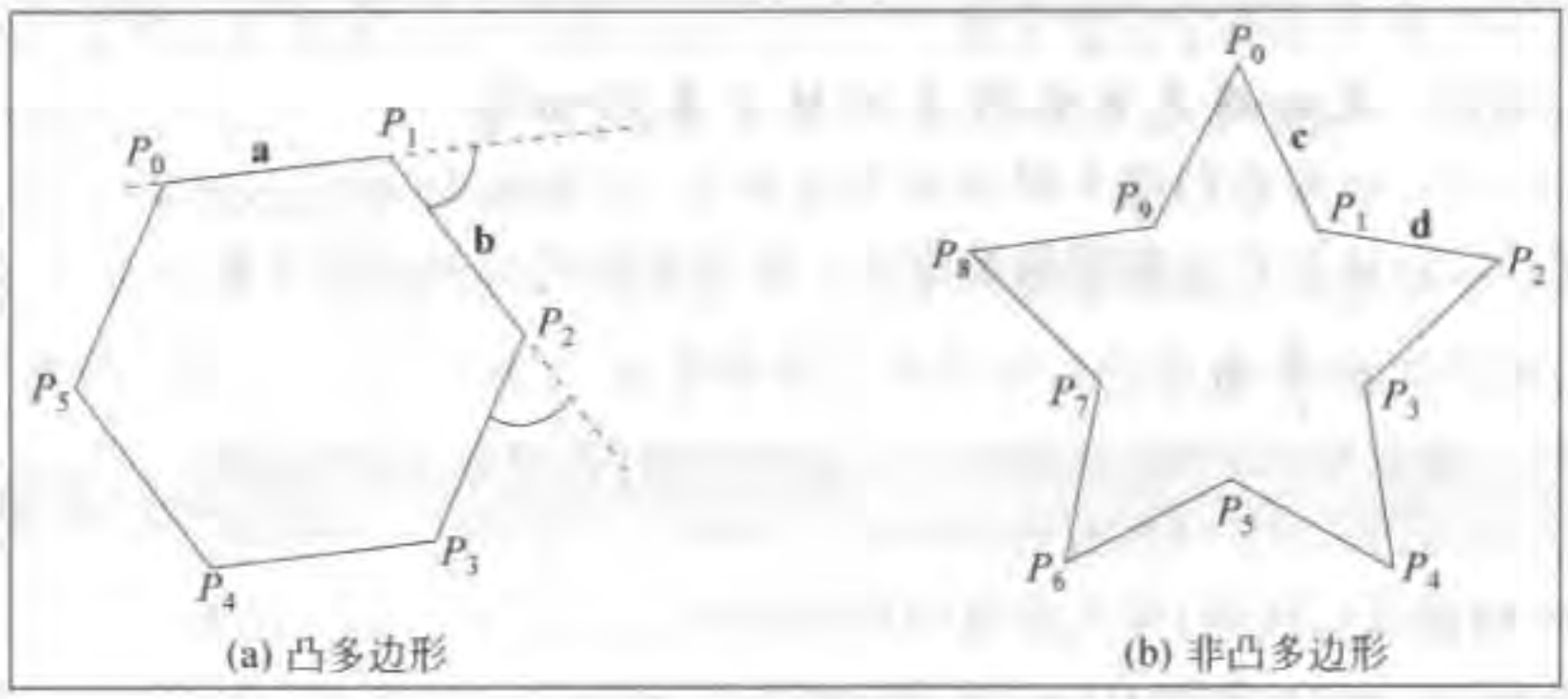


图 4.17 判断多边形的凸性

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。

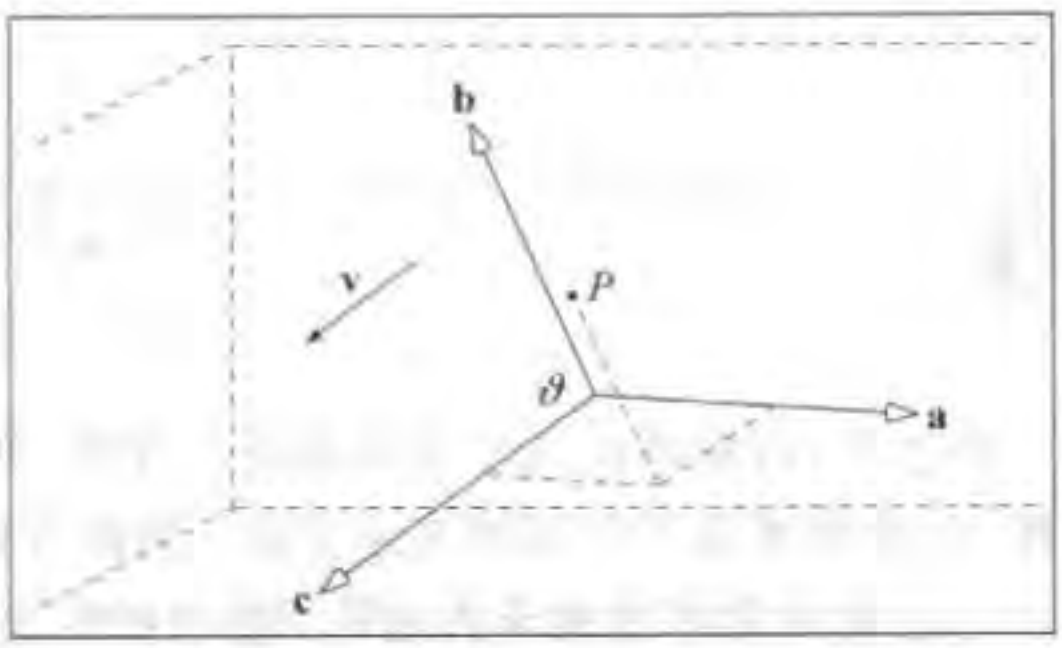


图 4.18　“世界”中的一个坐标框架

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。

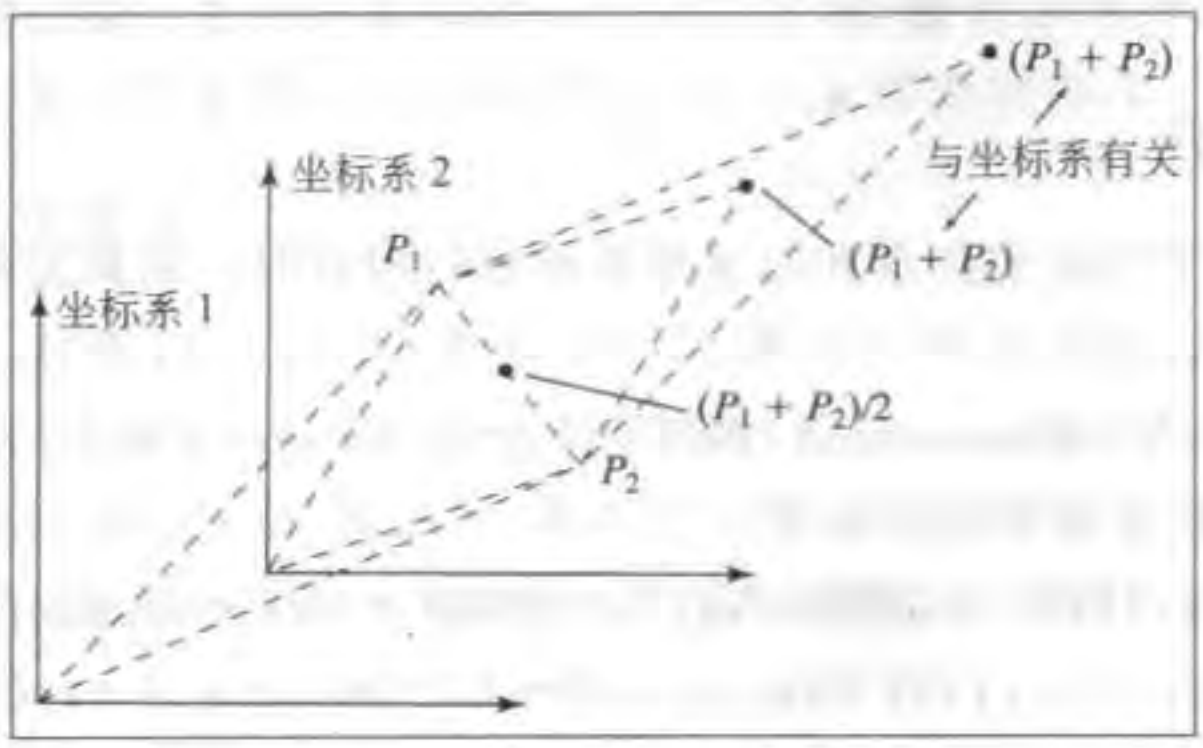


图 4.19 点的相加是不合理的

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])。

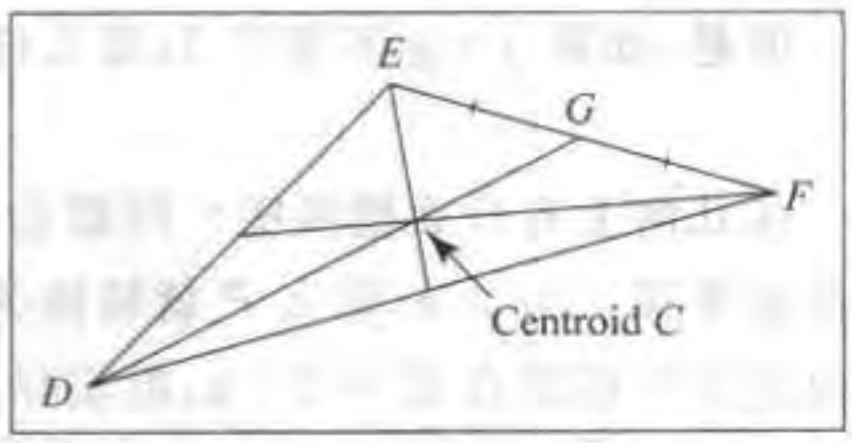


图 4.20 通过仿射组合求三角形的质心

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。

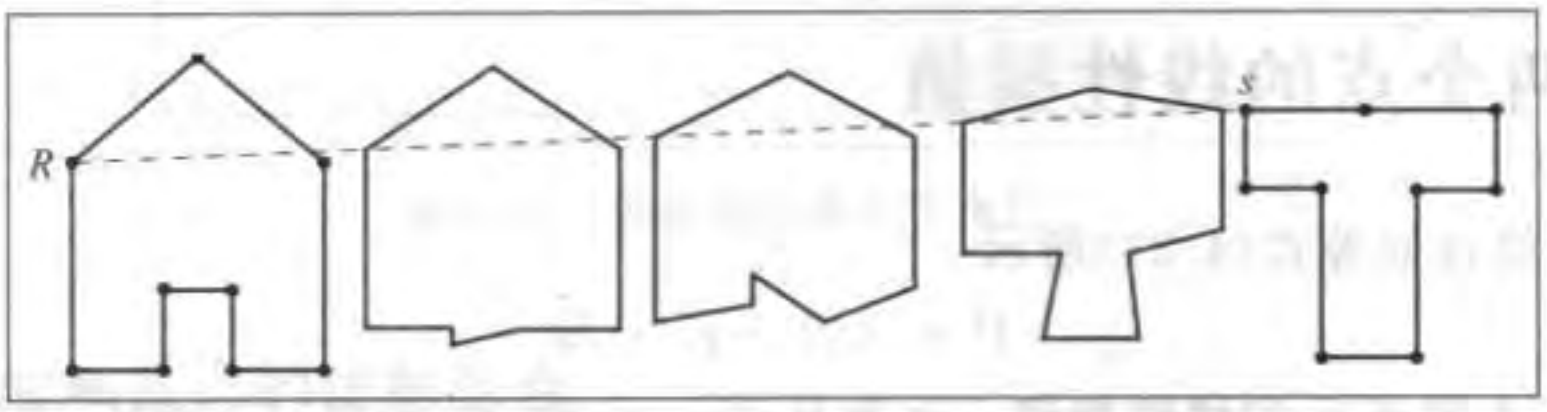


图 4.22 把房子渐变成 T 字型

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。

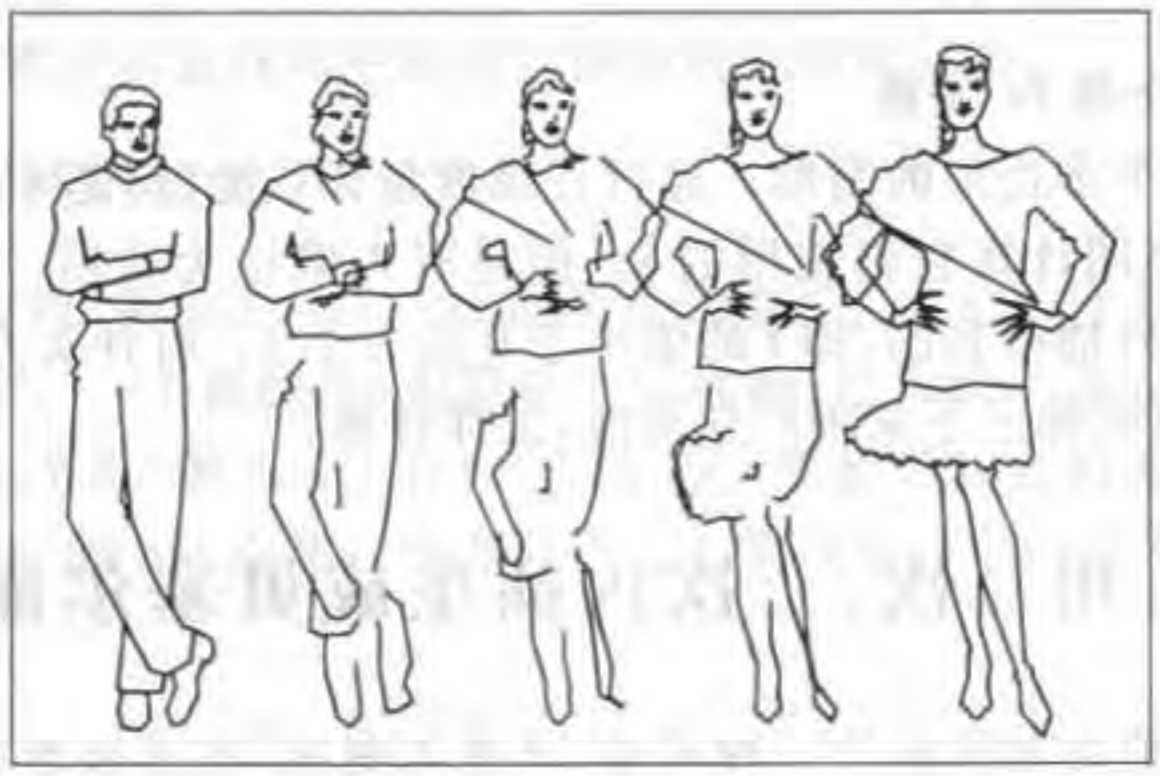


图 4.24　从男人变为女人

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。

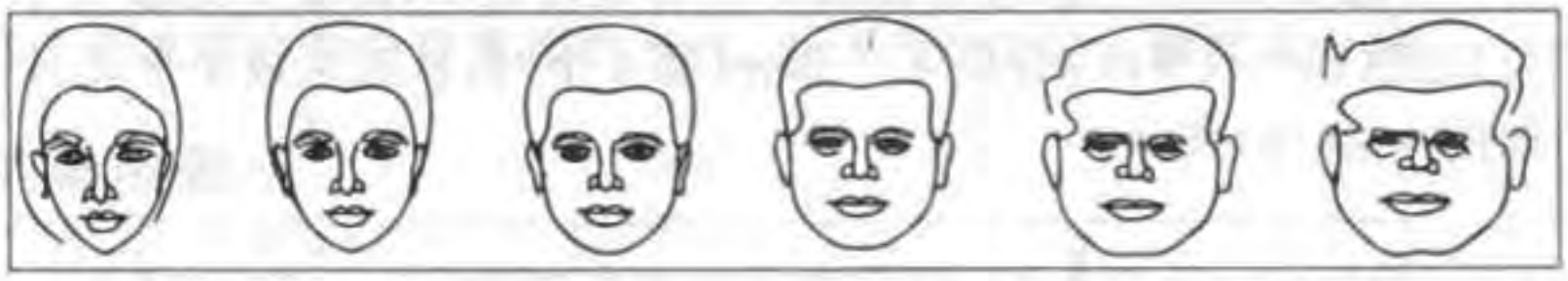


图 4.25　脸部讽刺漫画：内插和外插

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。

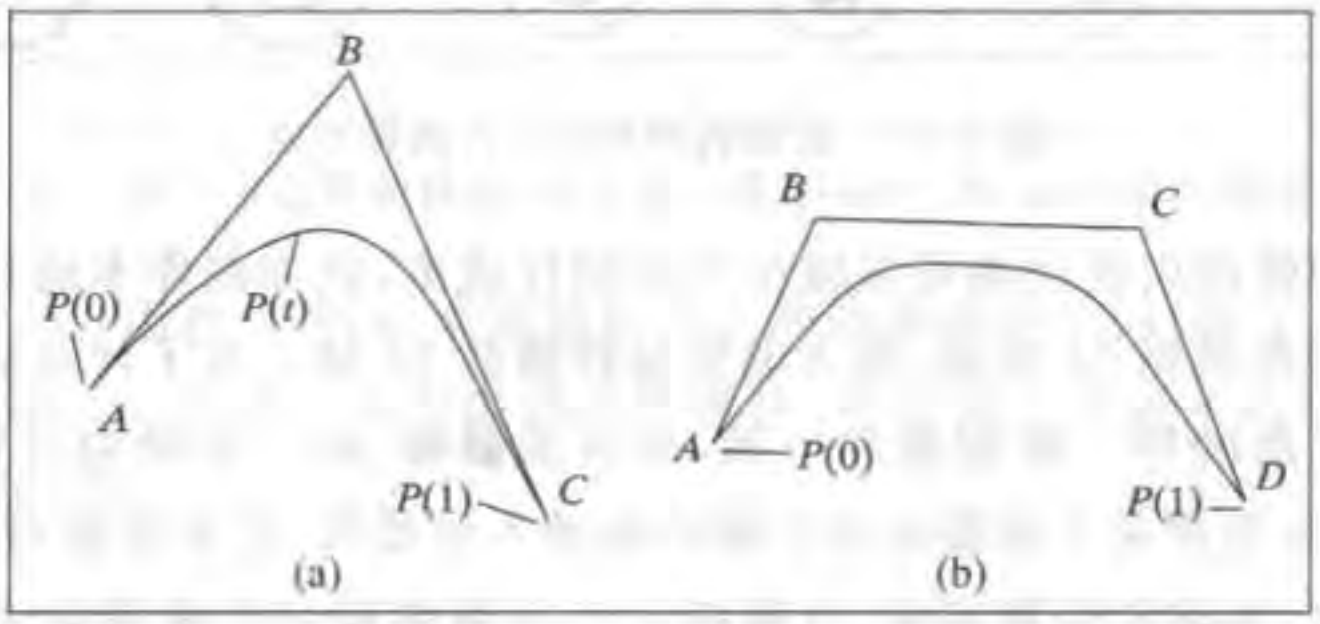


图 4.26 贝塞尔曲线是内插的结果

(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))。

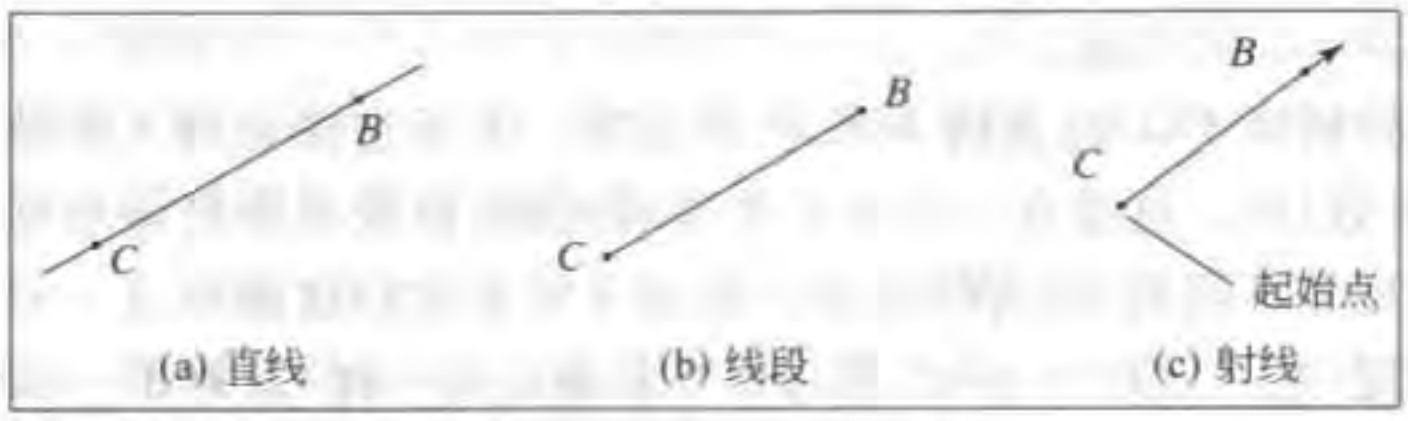


图 4.27　直线、线段和射线

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。

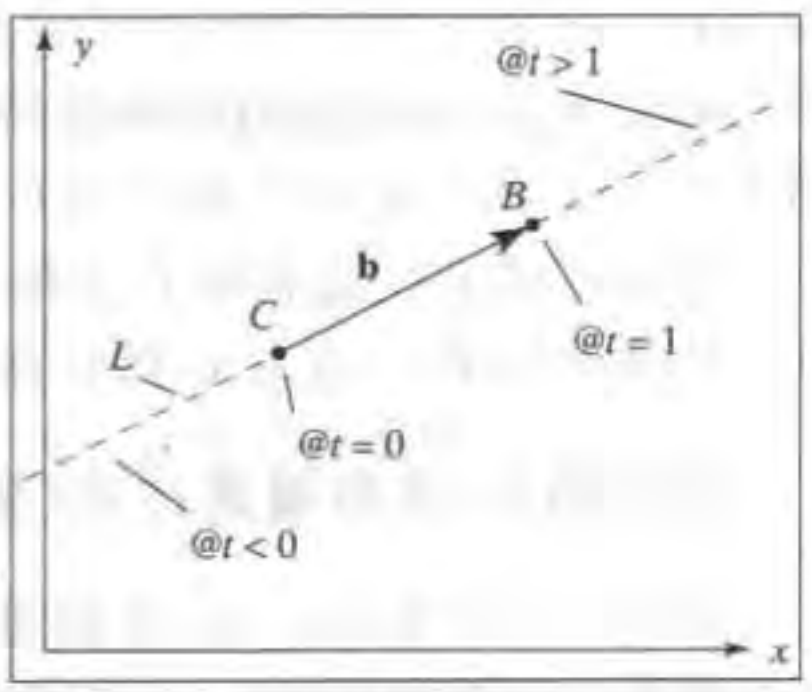


图 4.28　直线的参数表示 $L(t)$

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。

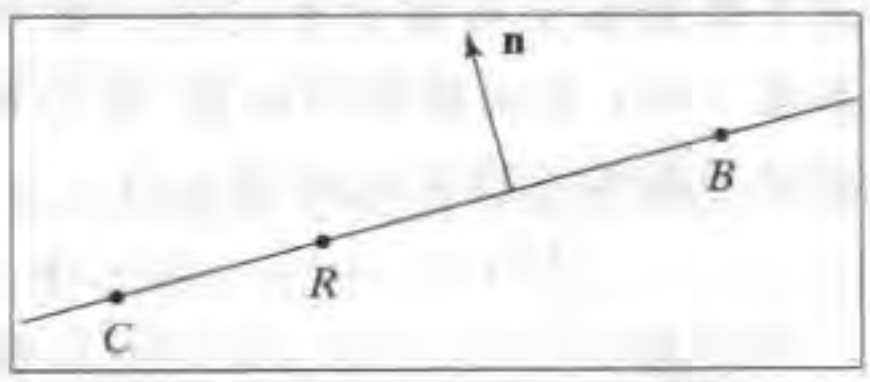


图 4.29　求二维直线的点-法向量表达式

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 $\mathbf{b}=B-C$。

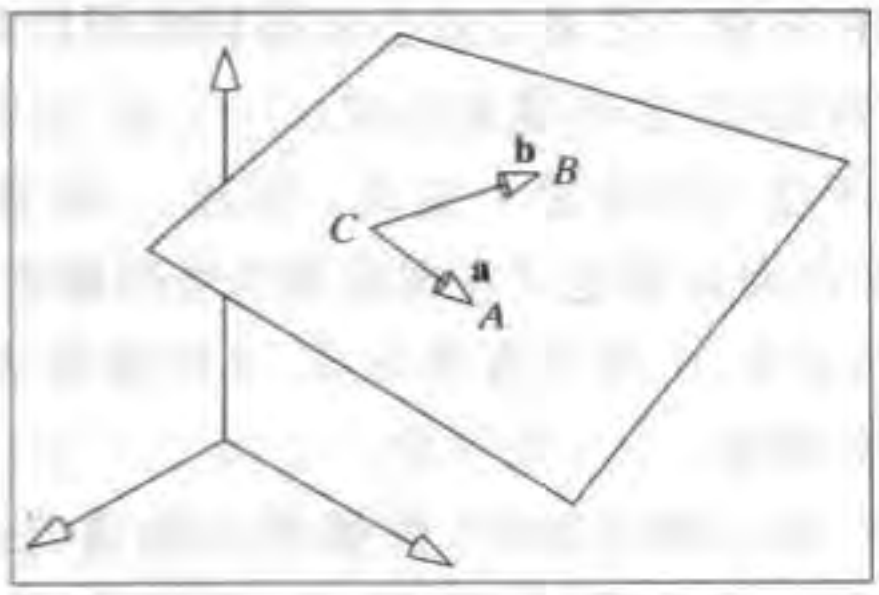


图 4.30 使用参数表示定义平面

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。

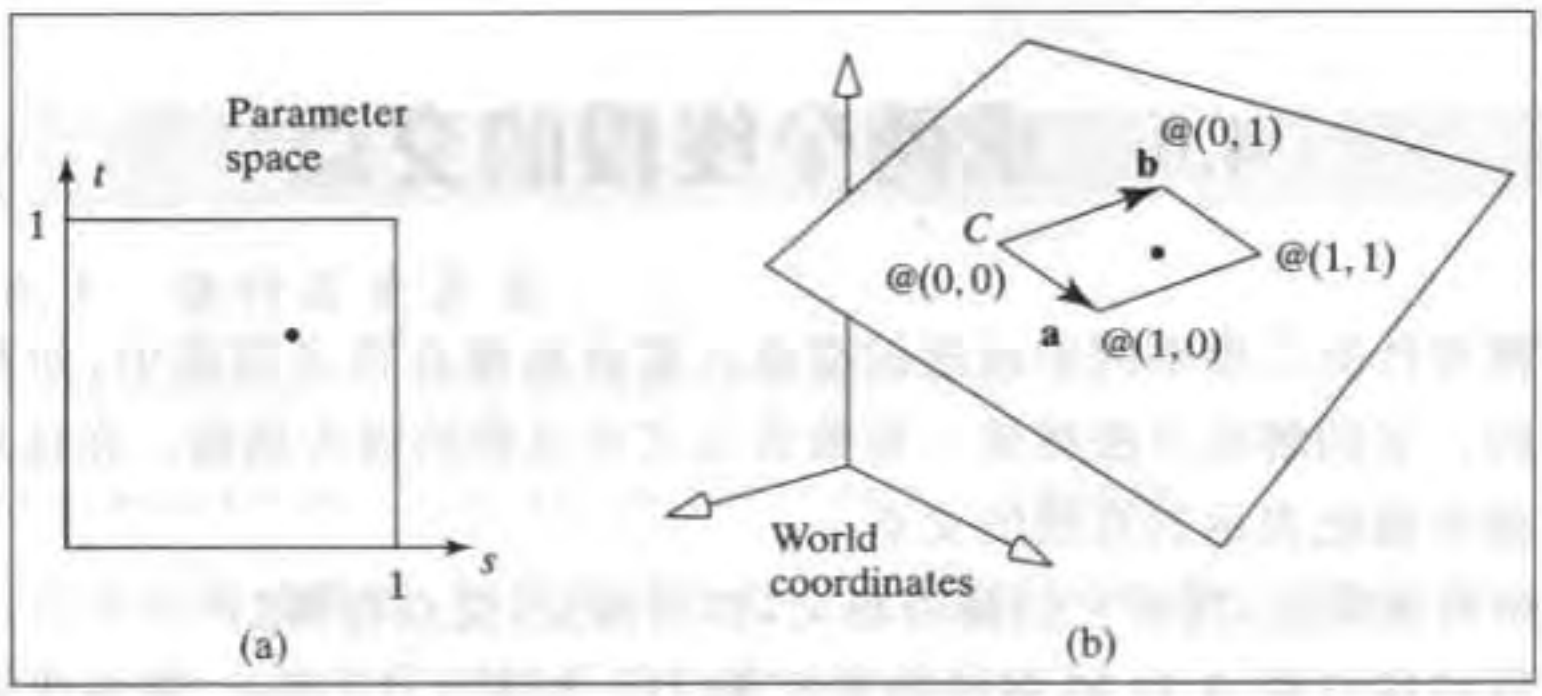


图 4.31 定义平面面片的两个空间

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。

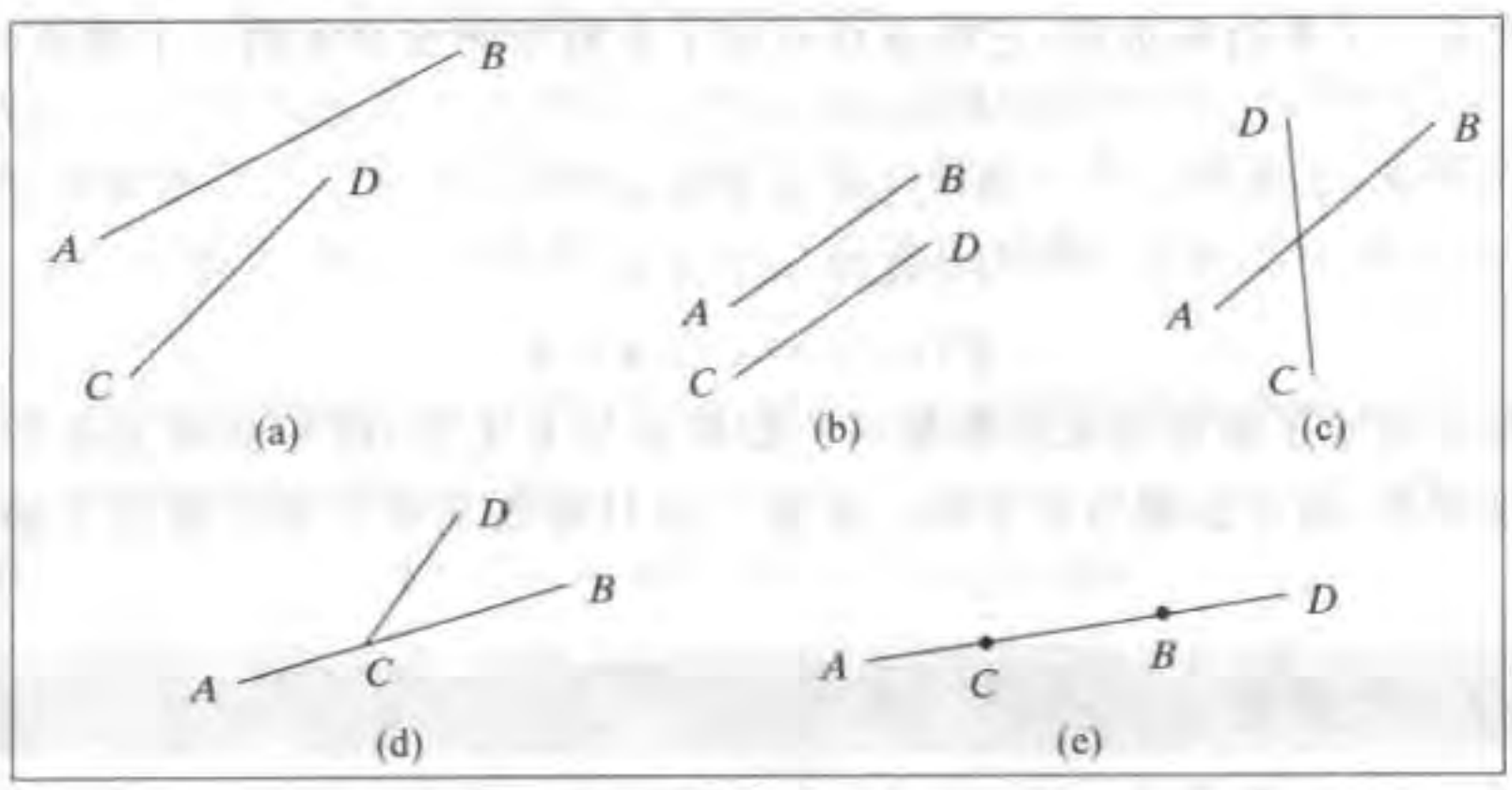


图 4.32 两条直线的多种位置关系

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。

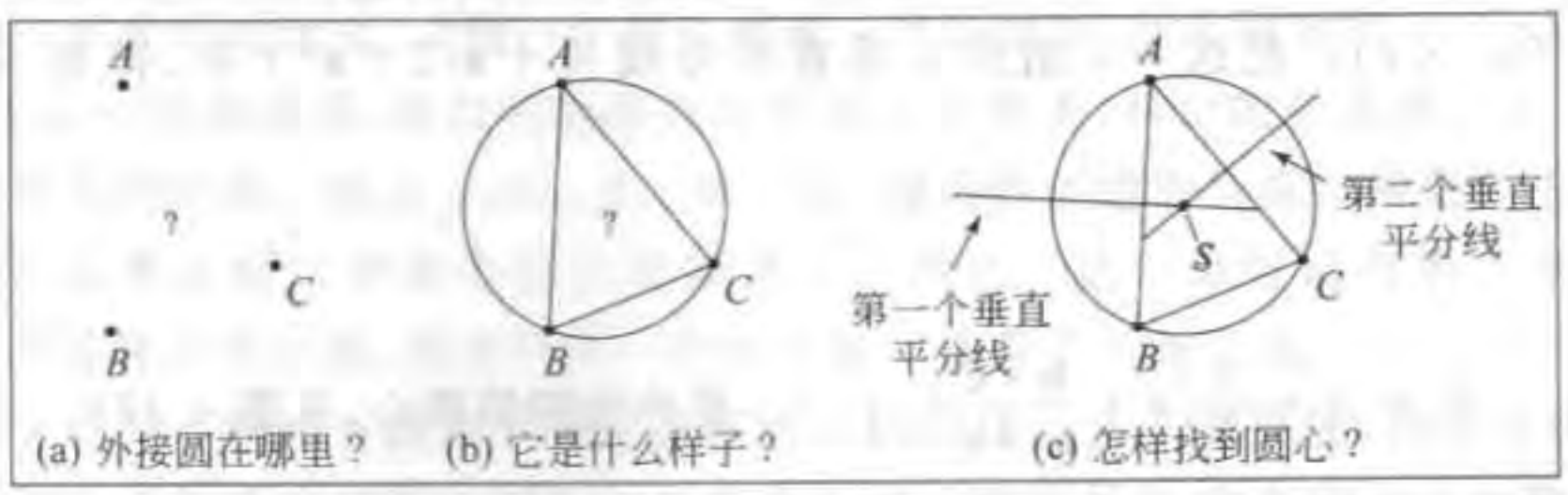
练习 4.6.3　线段求交

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。

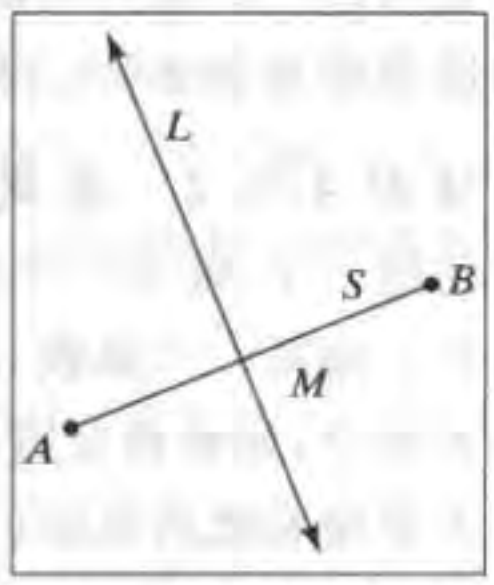


图 4.34 线段的垂直平分线

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。

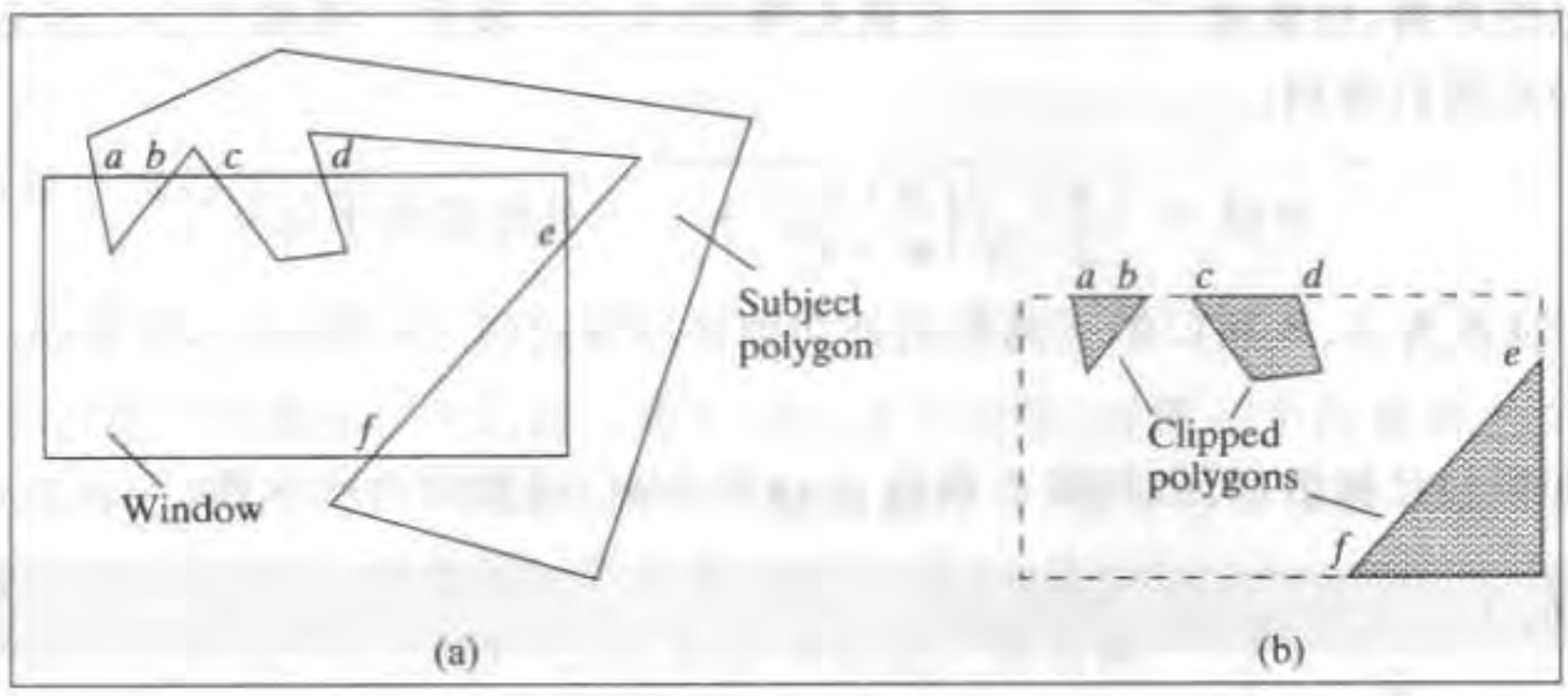


图 4.35 用窗口裁剪多边形

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。

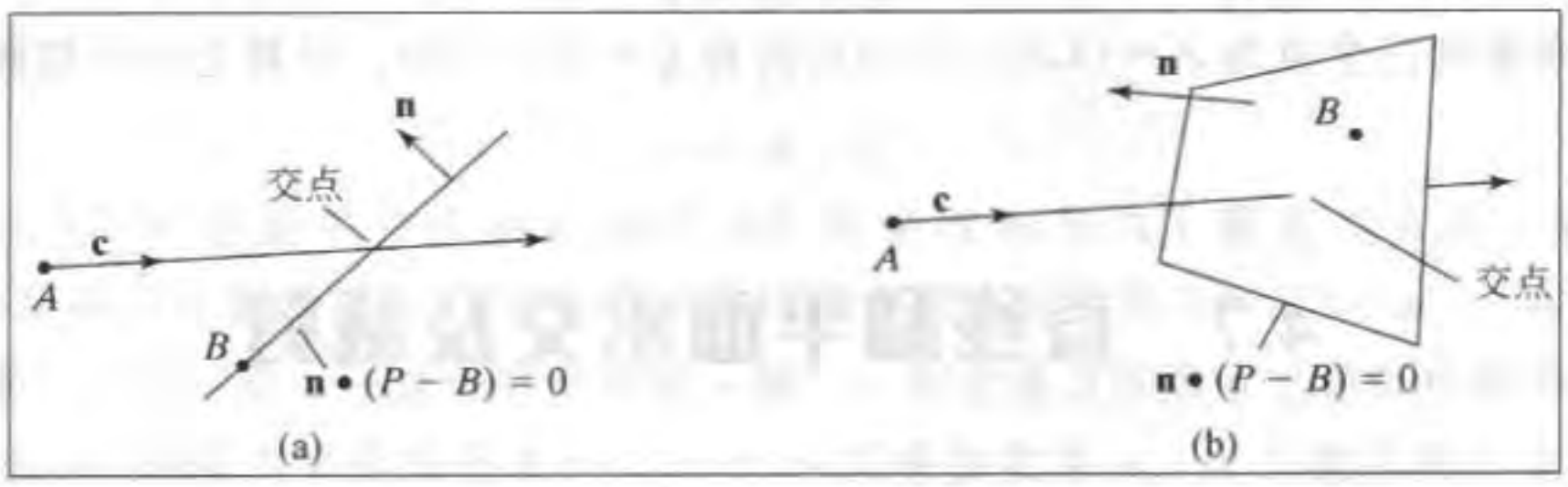


图 4.36 射线与直线或平面的交点在哪里?

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:

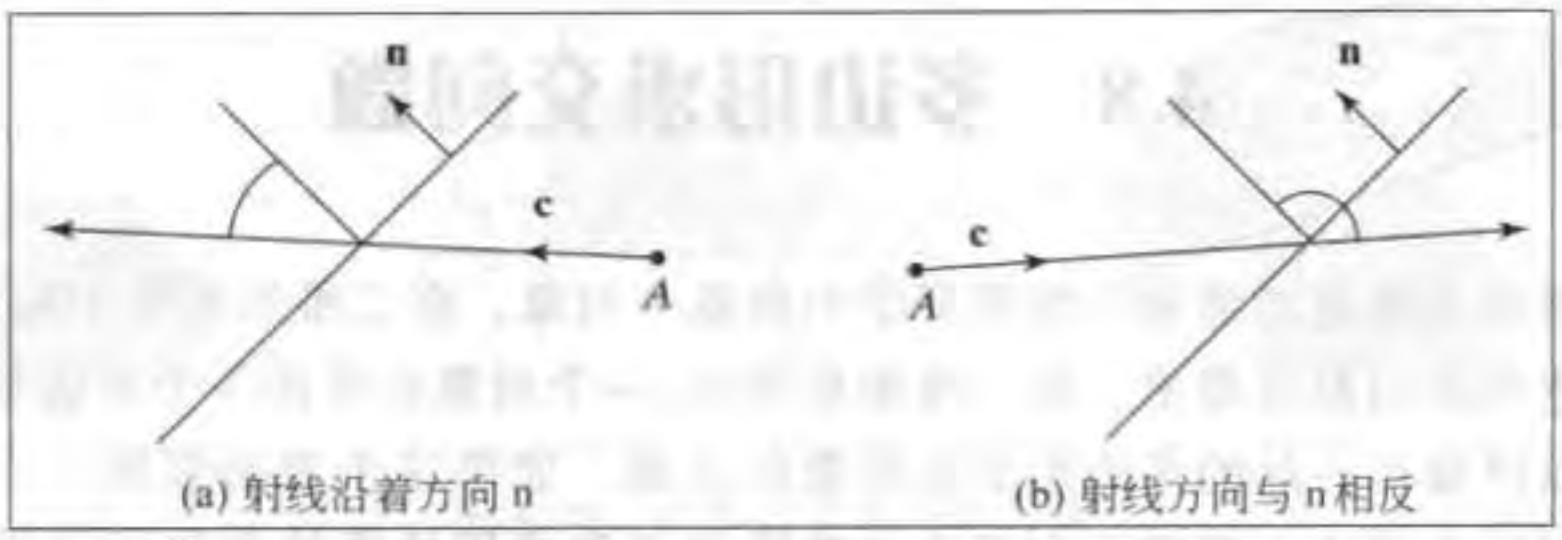
$$t_{hit}=\frac{\mathbf{n}\cdot(B-A)}{\mathbf{n}\cdot\mathbf{c}}\quad(\text{击中时间})\tag{4.50}$$

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:

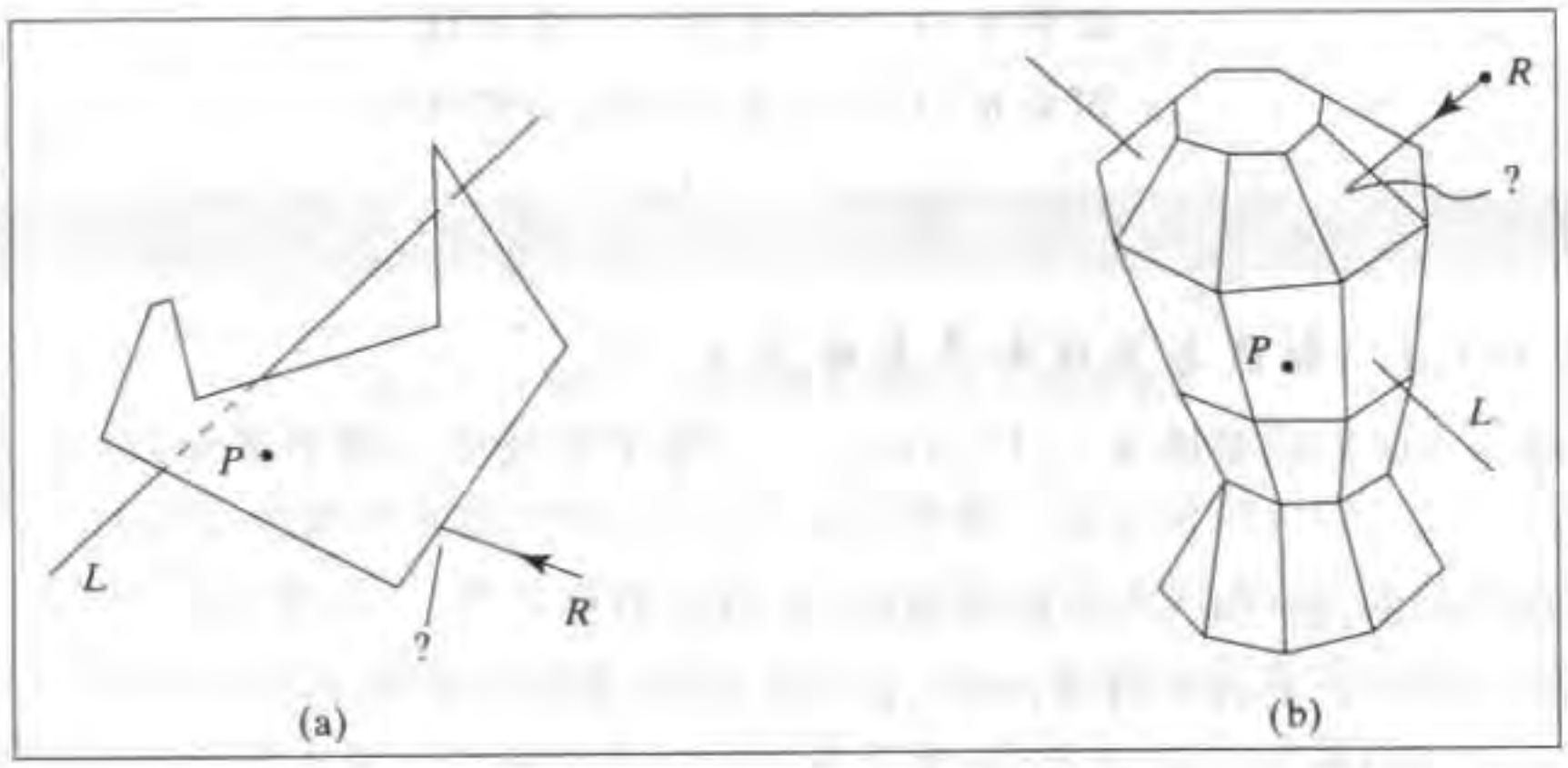


图 4.38 直线和多边形类对象的相交问题

(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。

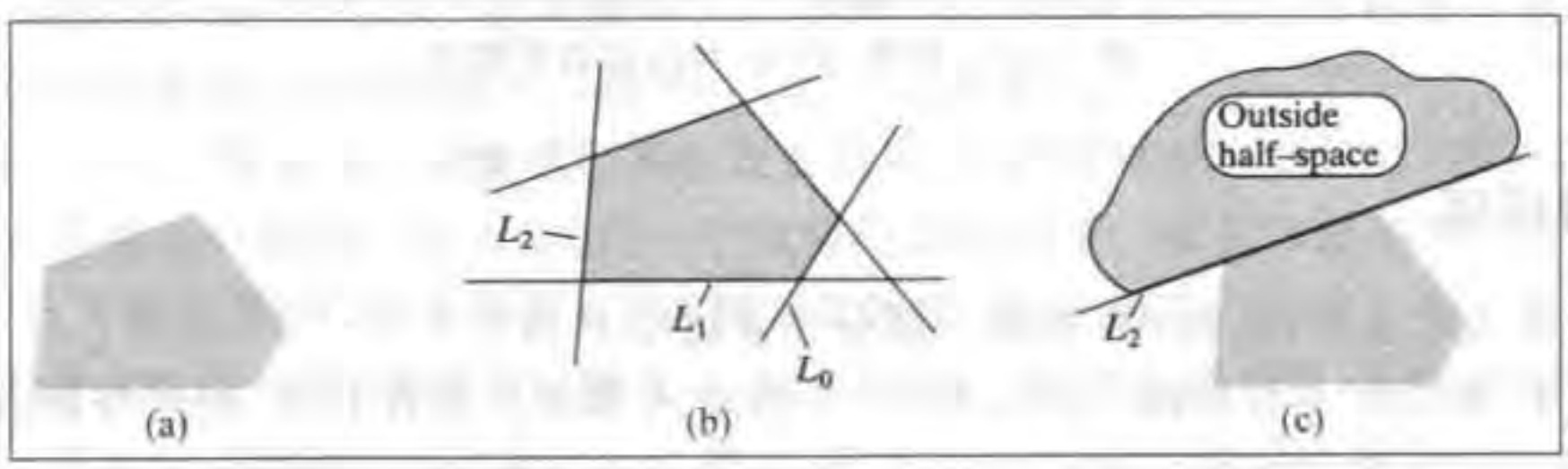


图 4.39　凸多边形和凸多面体

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：

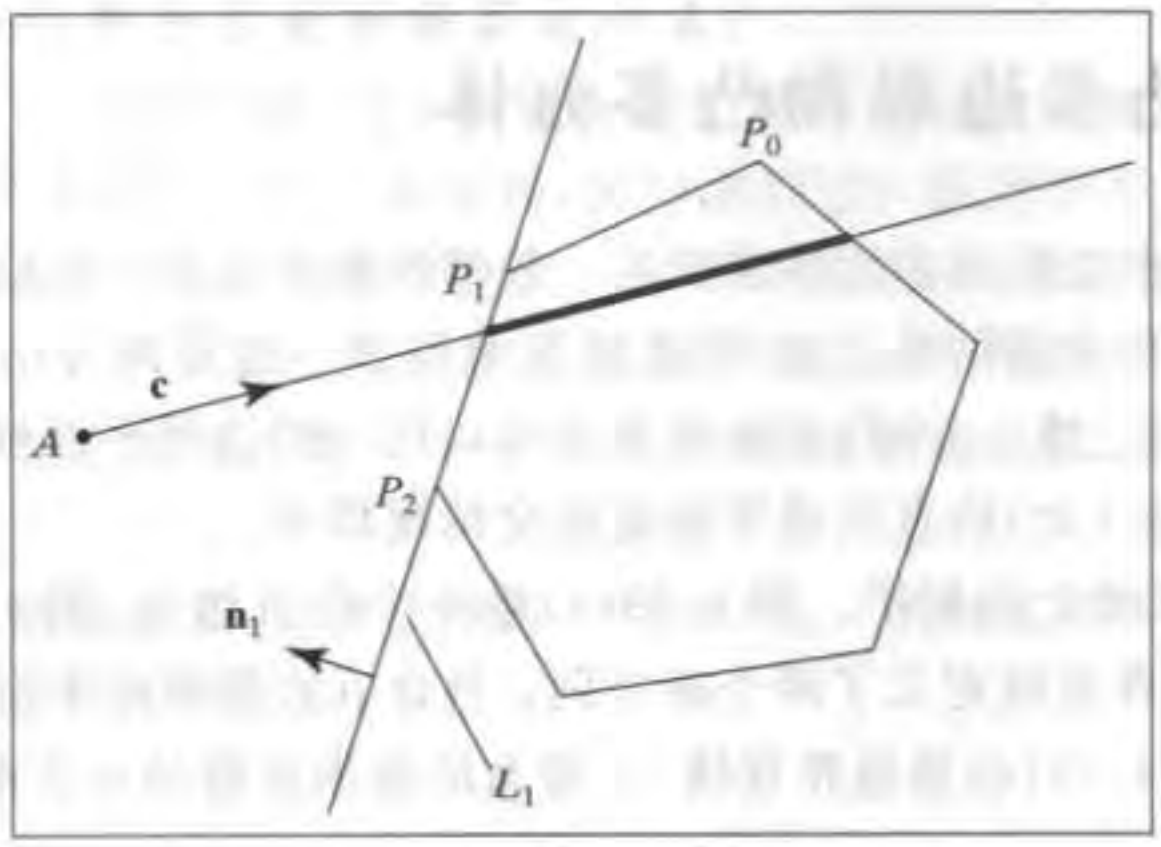


图 4.40 射线 $A+\mathbf{c}t$ 与凸多边形相交

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。

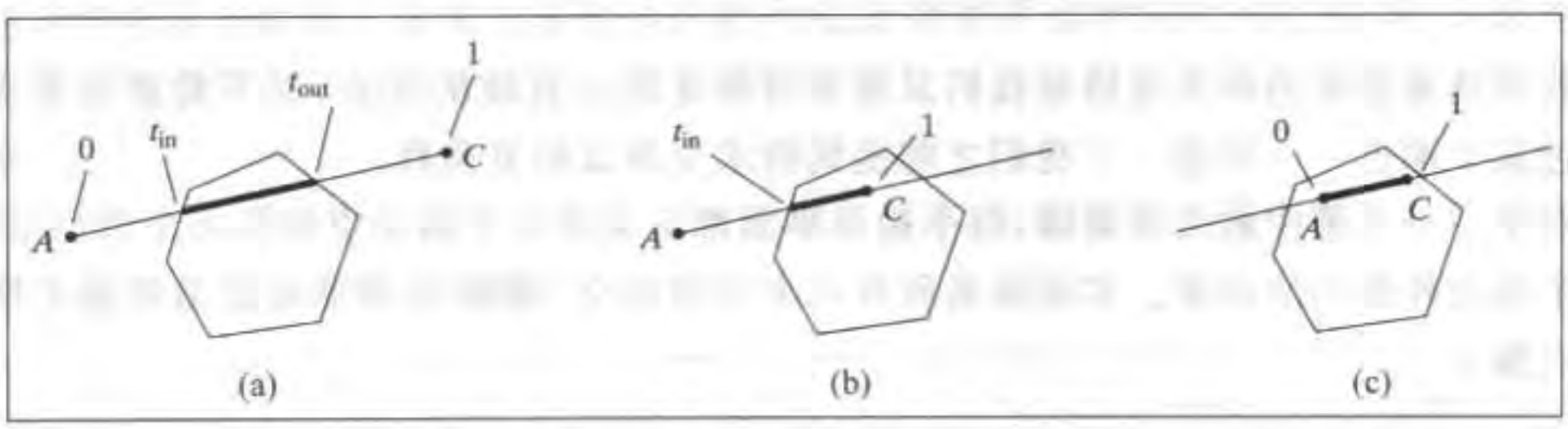


图 4.41 一条线段被一个多边形 P 裁剪

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}。

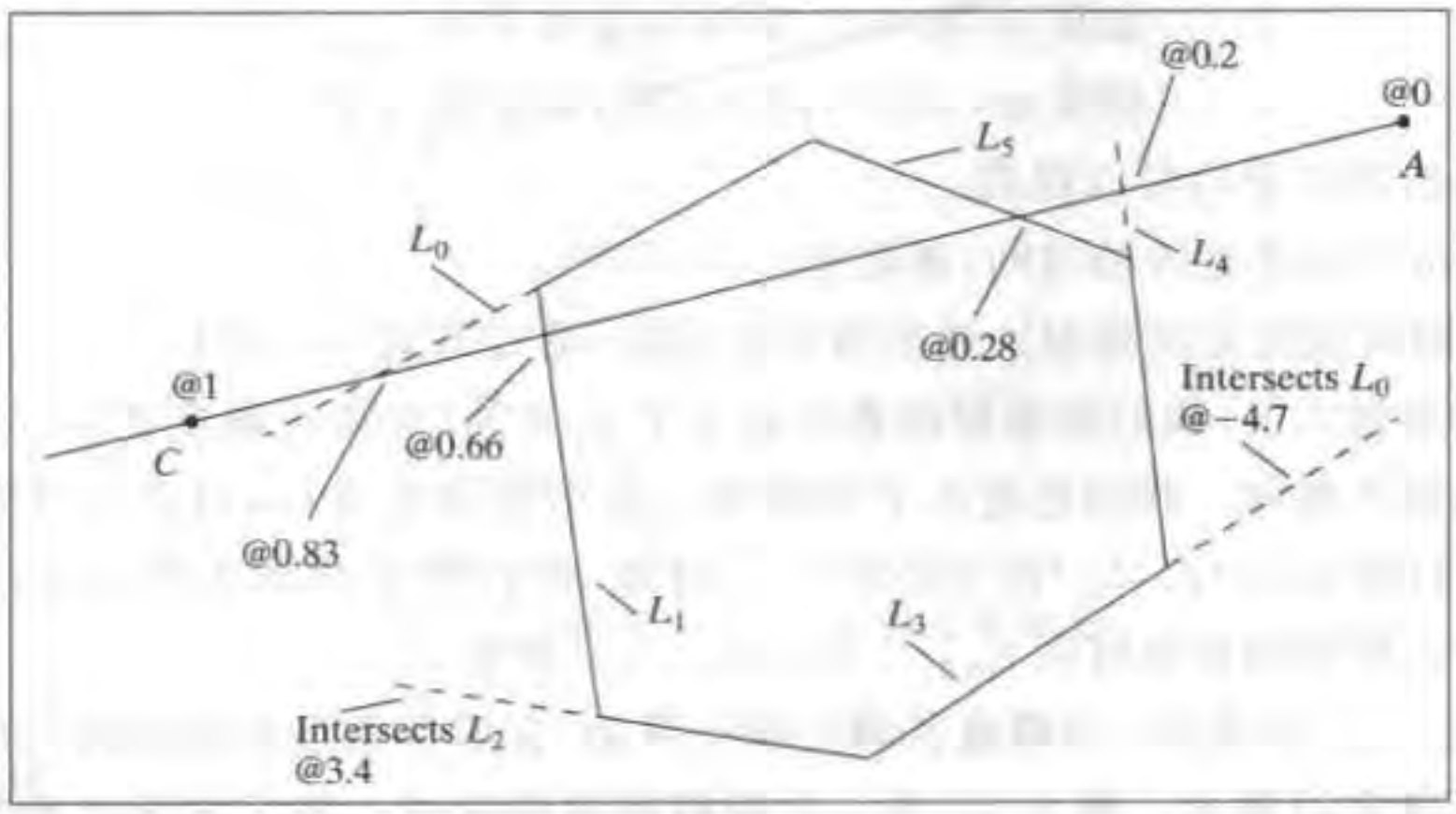


图 4.43 检测一条射线在凸多边形内的部分

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。

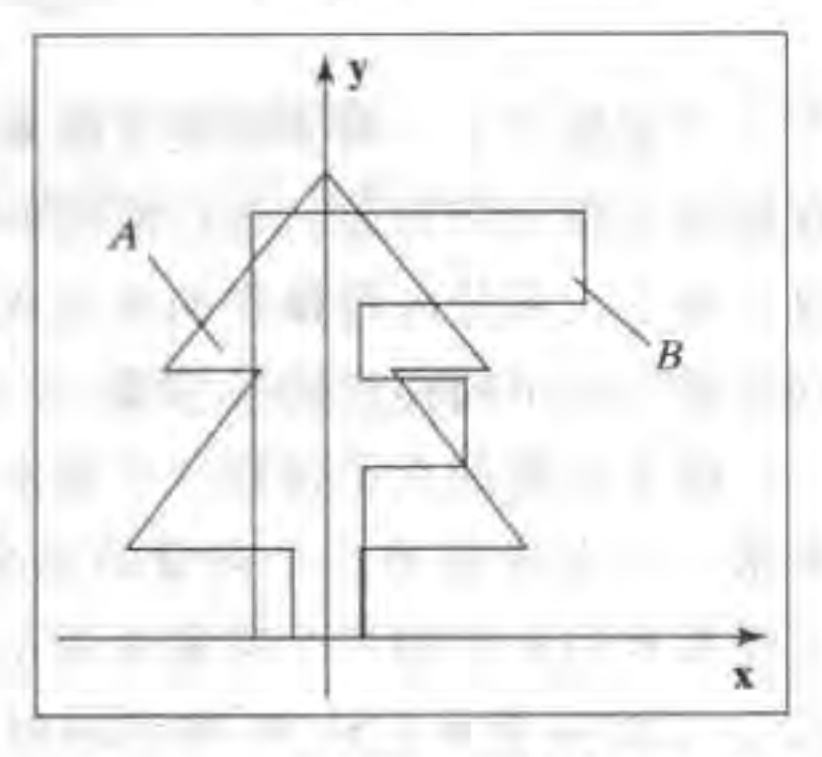


图 4.48 两个多边形的内插动画

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。

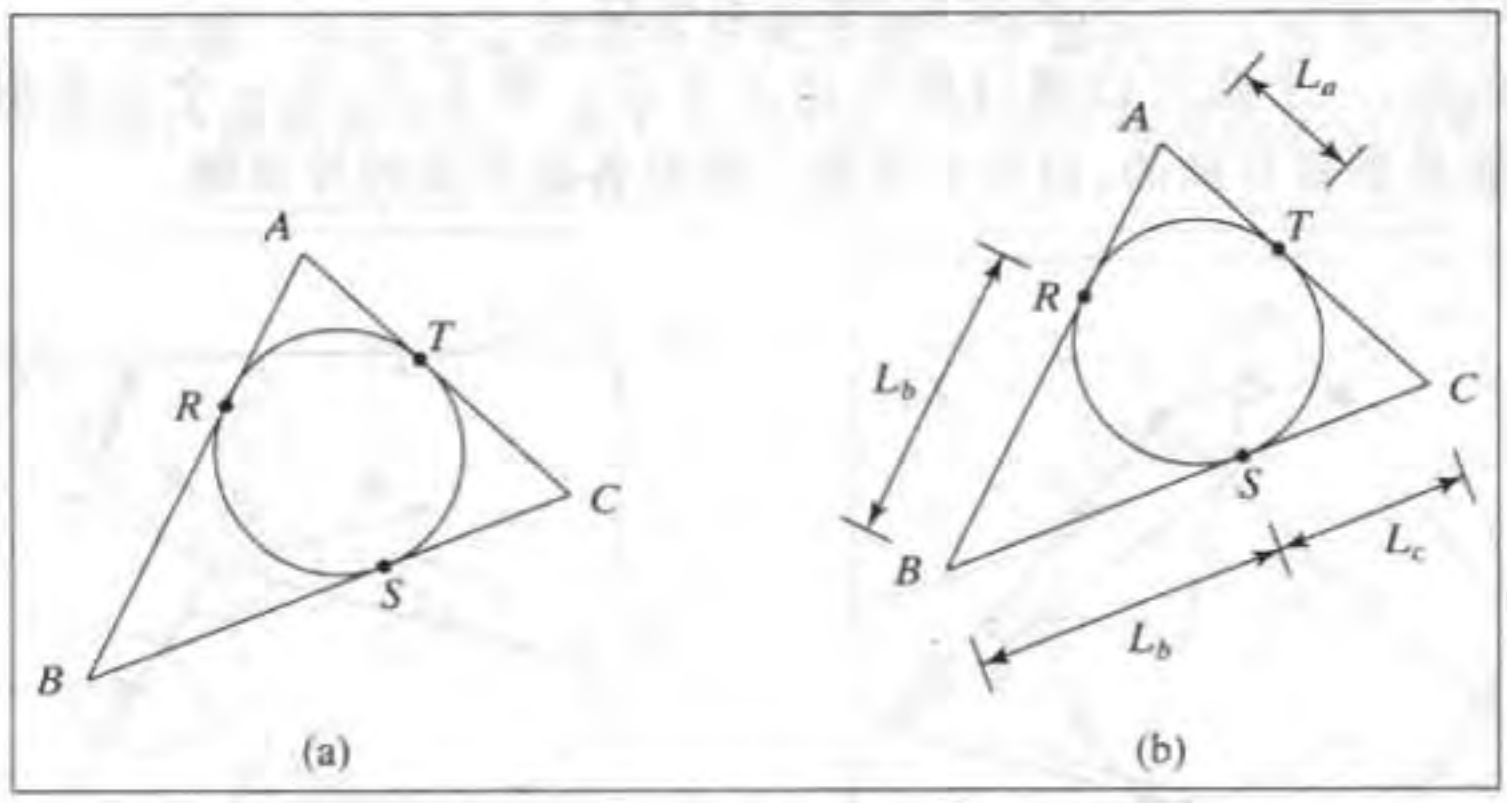


图 4.49　三角形 ABC 的内切圆是三角形 RST 的外切圆

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。

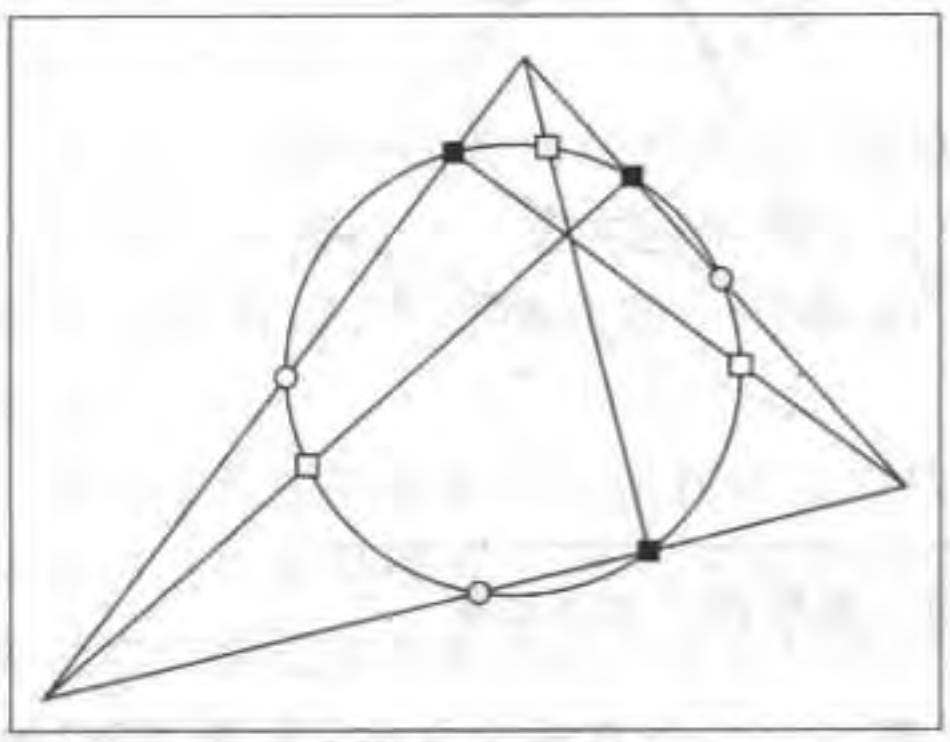


图 4.50 九点圆

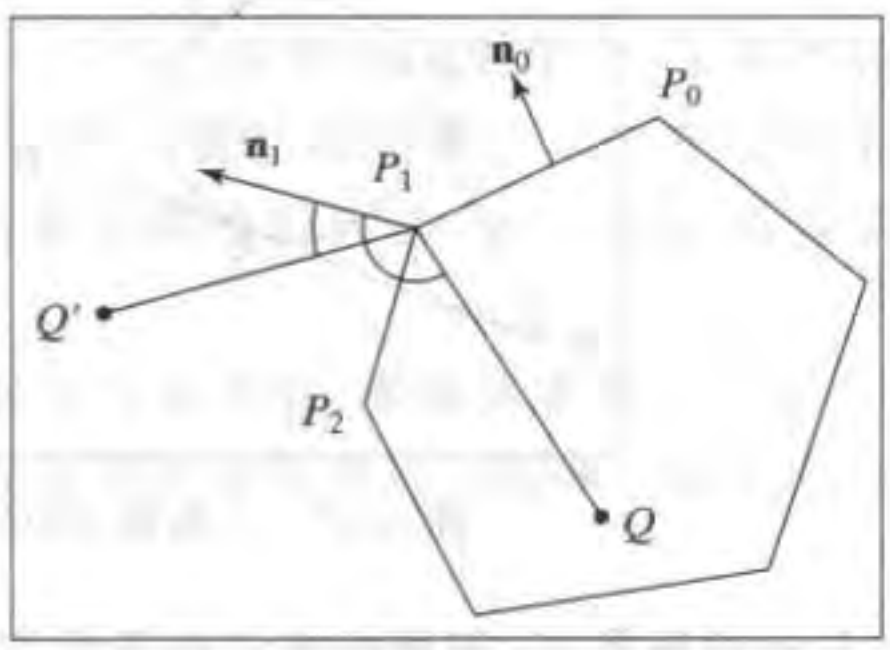


图 4.51 点 P 是否在多边形 Q 内

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。

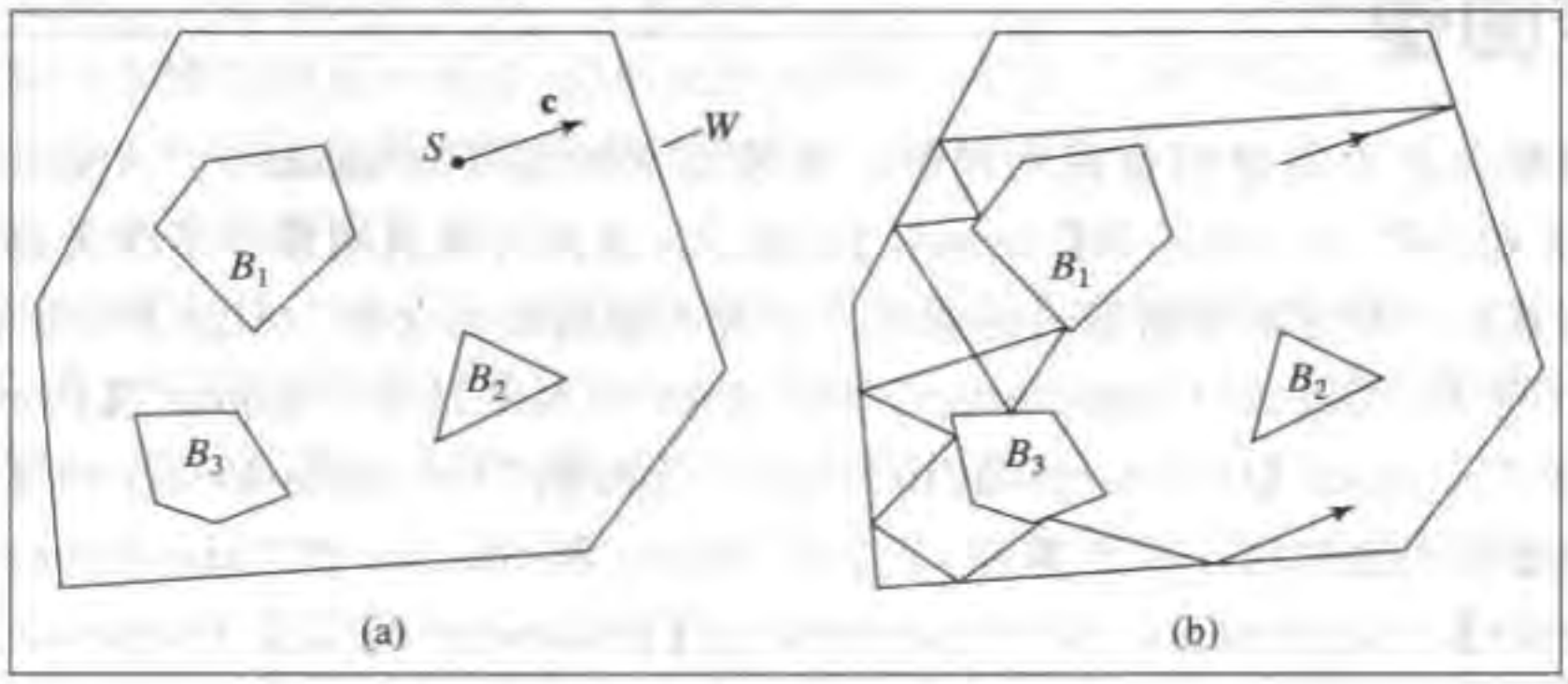


图 4.52　二维光线跟踪实验

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。

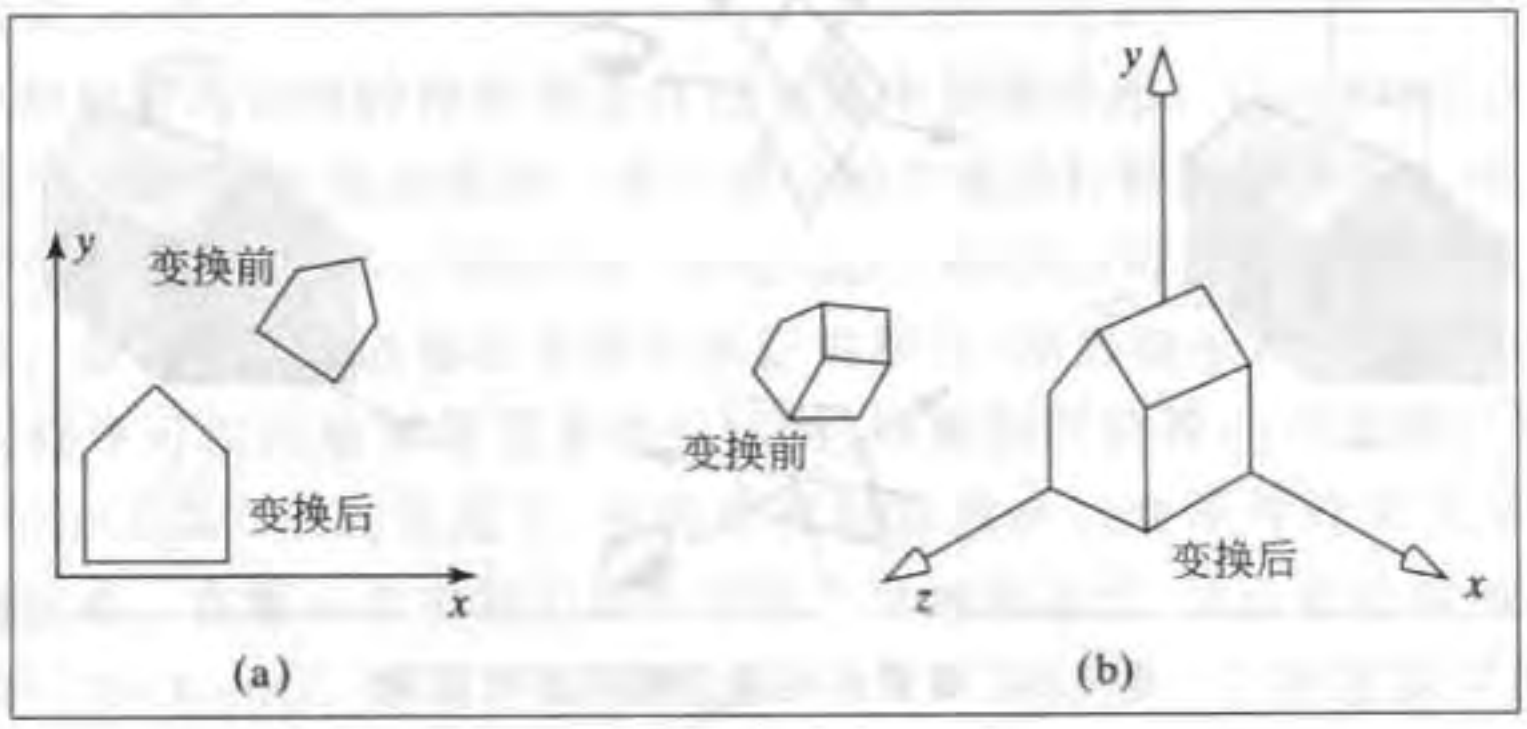


图 5.1　物体变换前和变换后的图像

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。

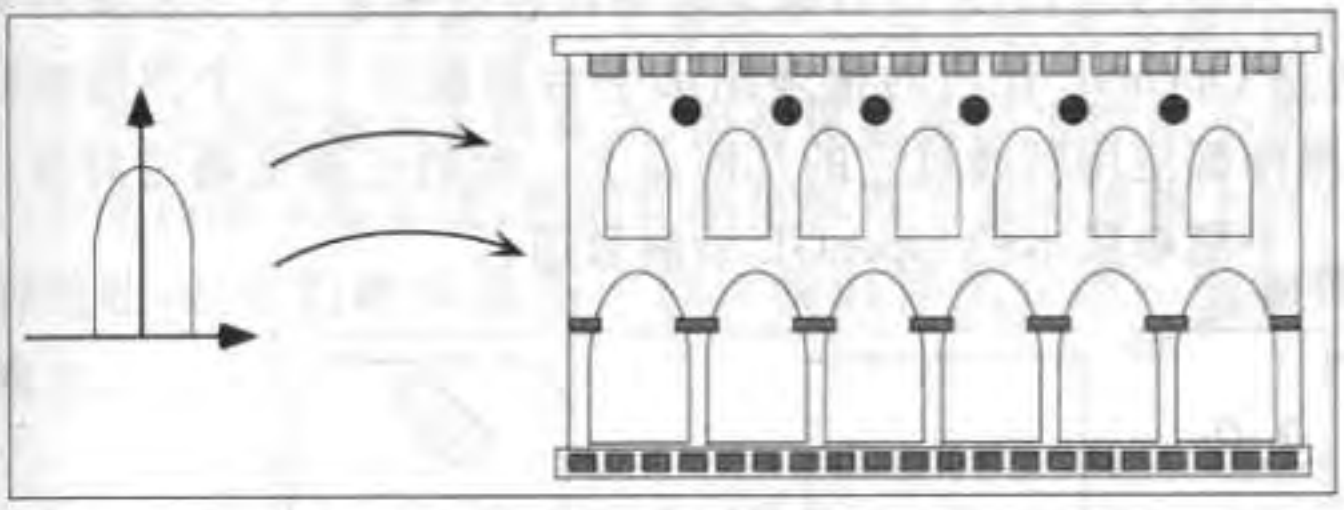


图 5.2　一幅简单实例重复组成的图像

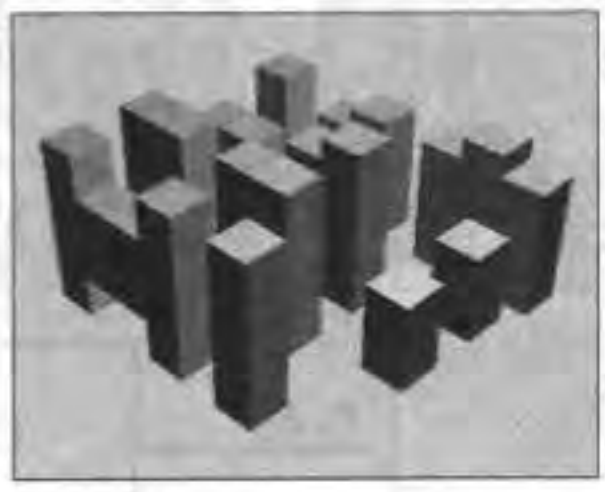


图 5.3　由基本图元构成的三维场景

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)。

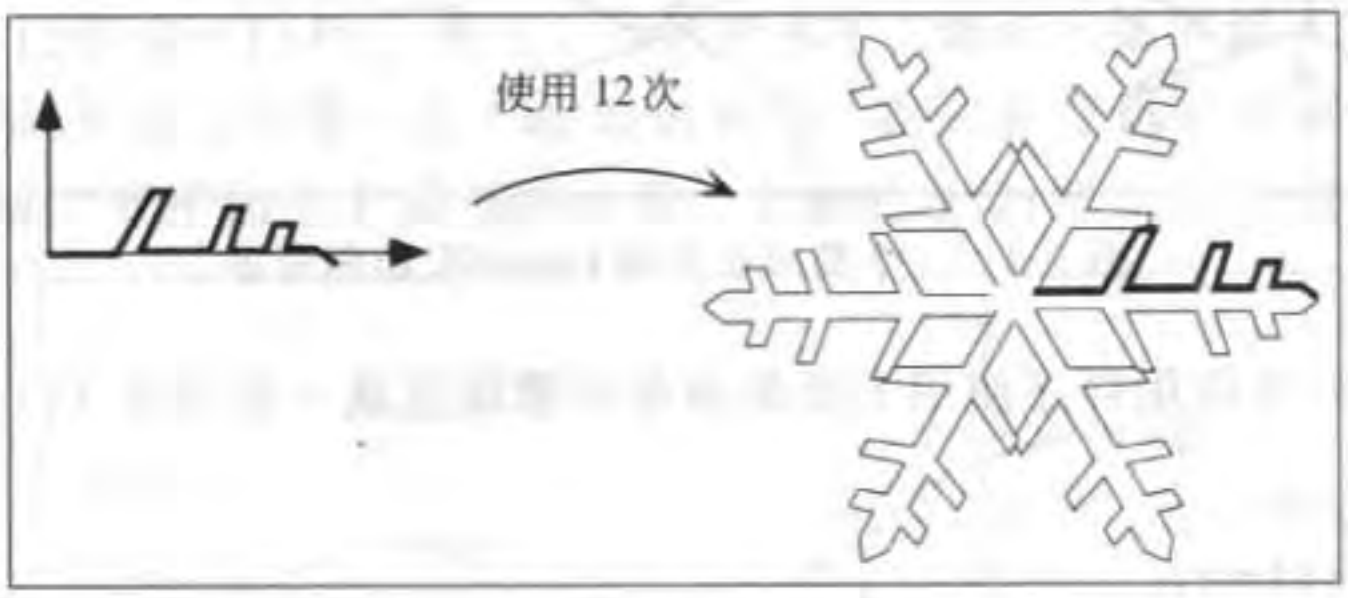


图 5.4　一个“基本图案”和它构成的图形

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。

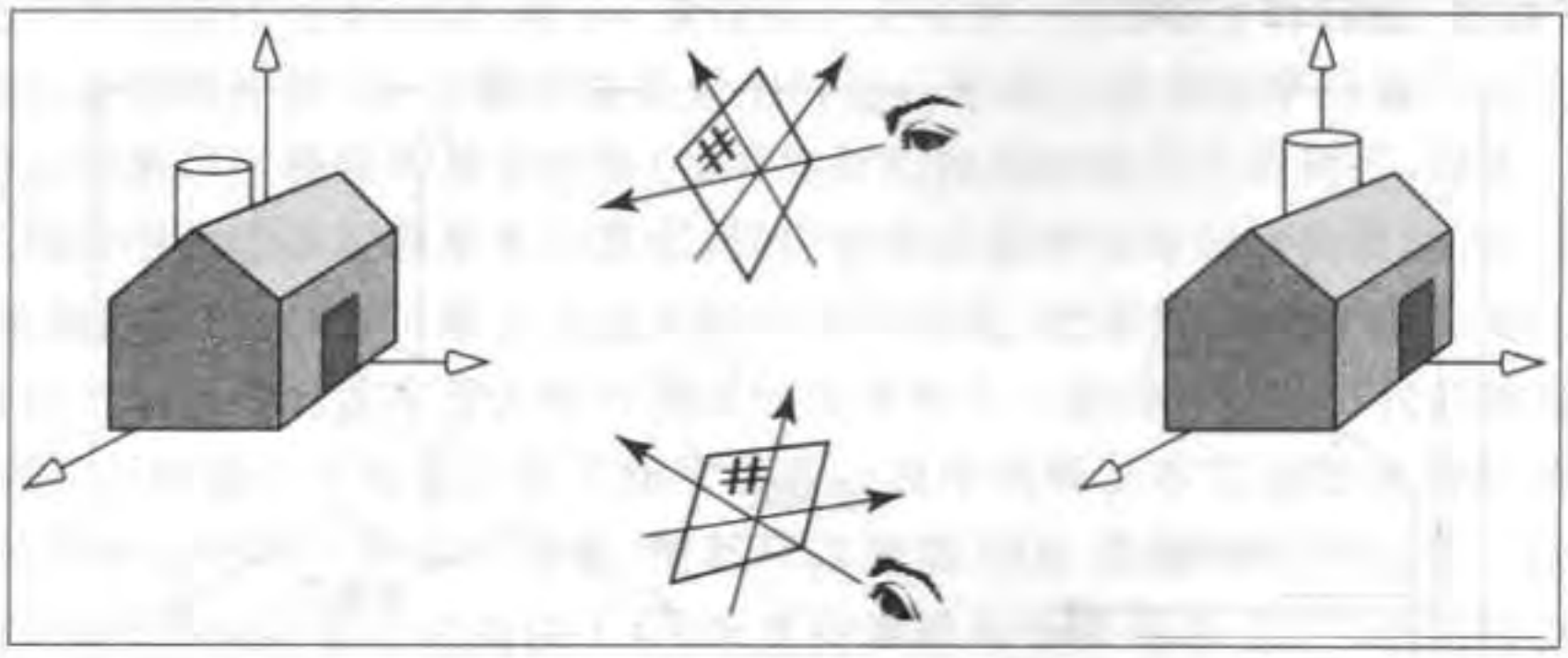


图 5.5 场景从不同角度观察的效果

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。

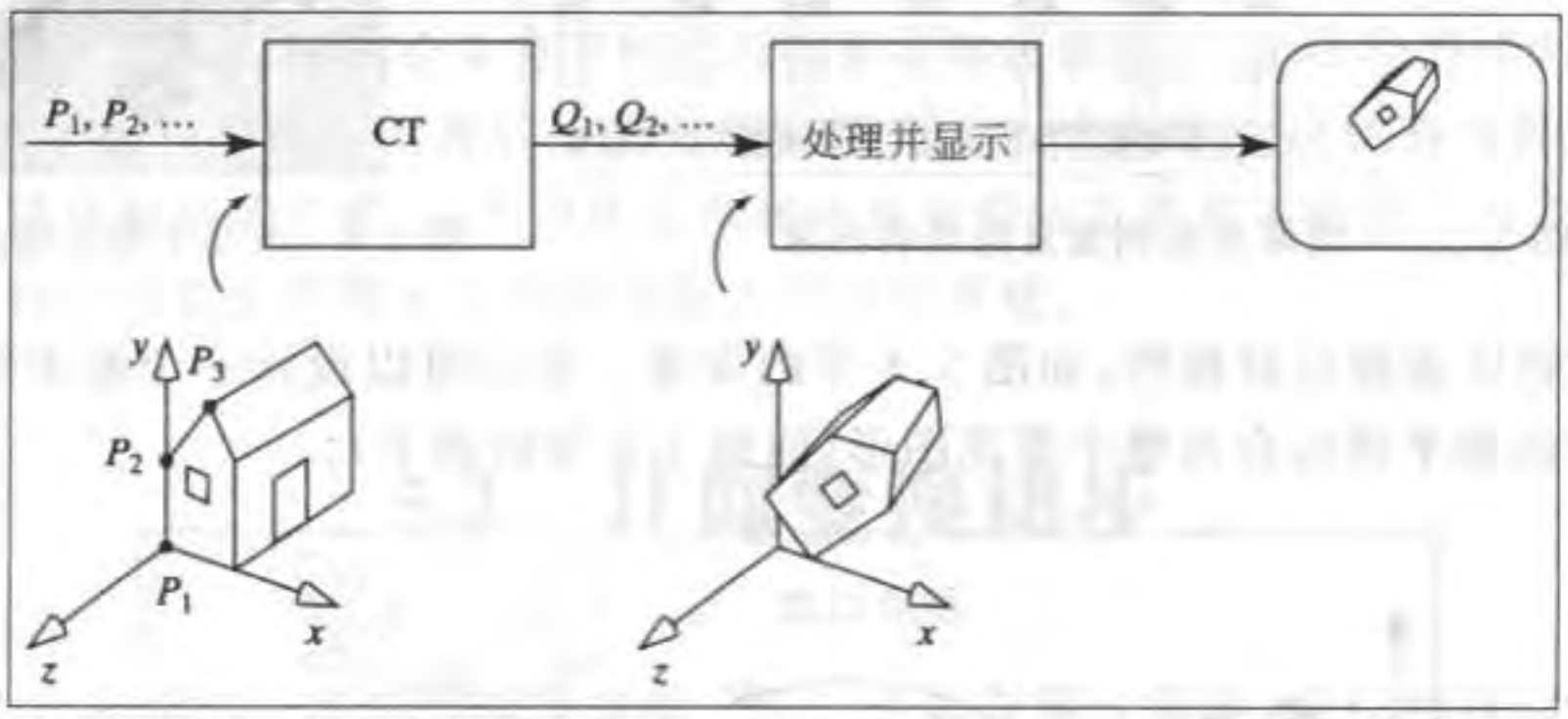


图 5.6 一个简单版本的 OpenGL 绘制管道

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

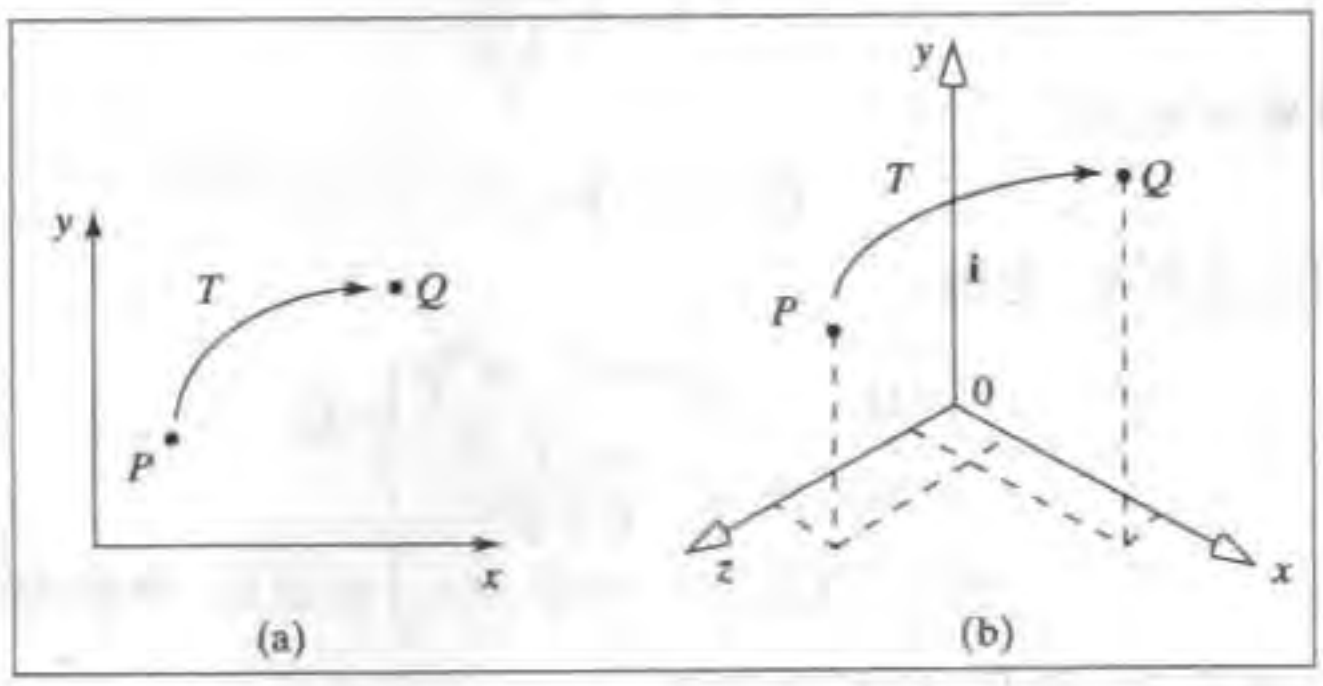


图 5.7 一个点映射到新的位置

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。

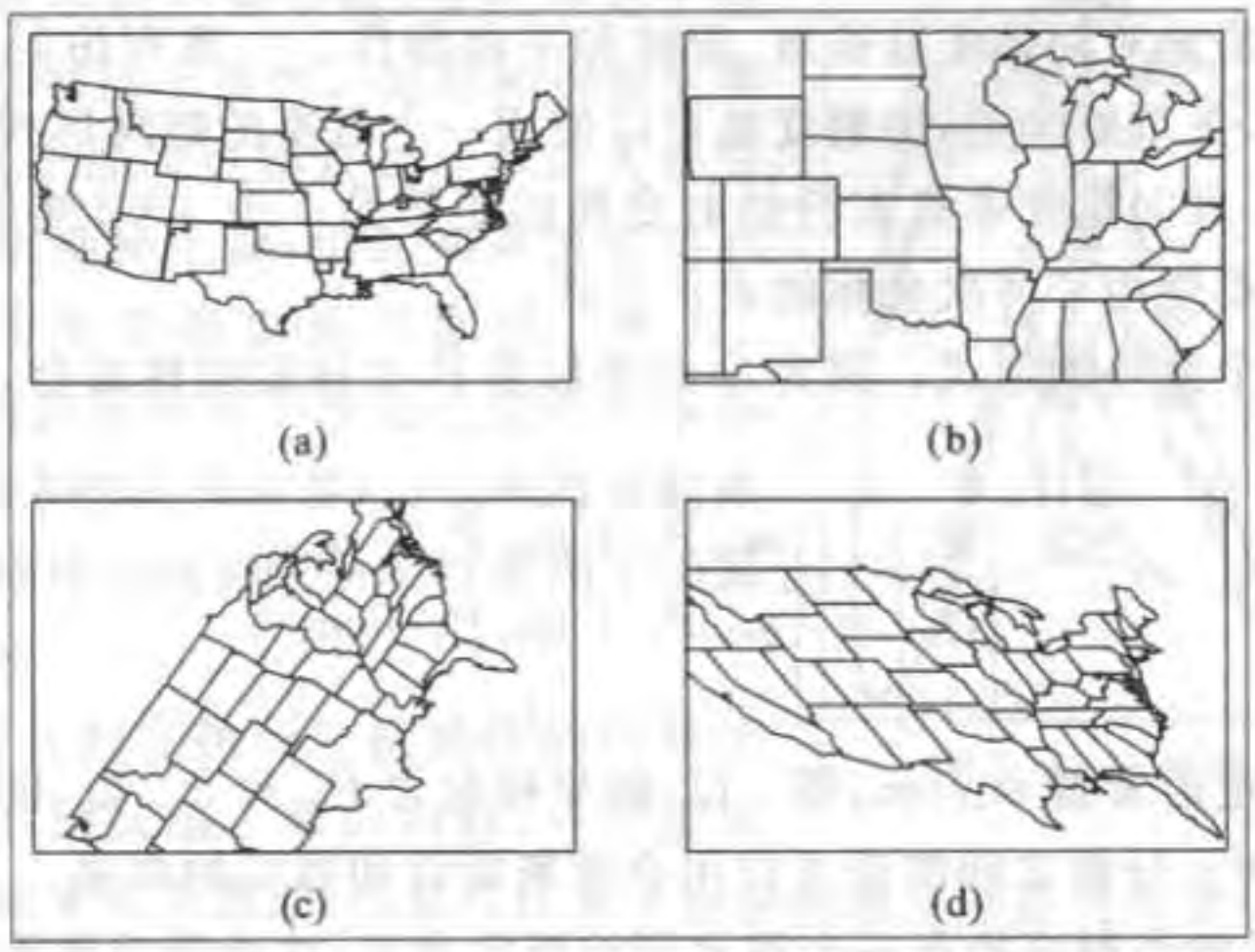


图 5.9 一幅地图的变换操作

(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 $\theta=60°$。

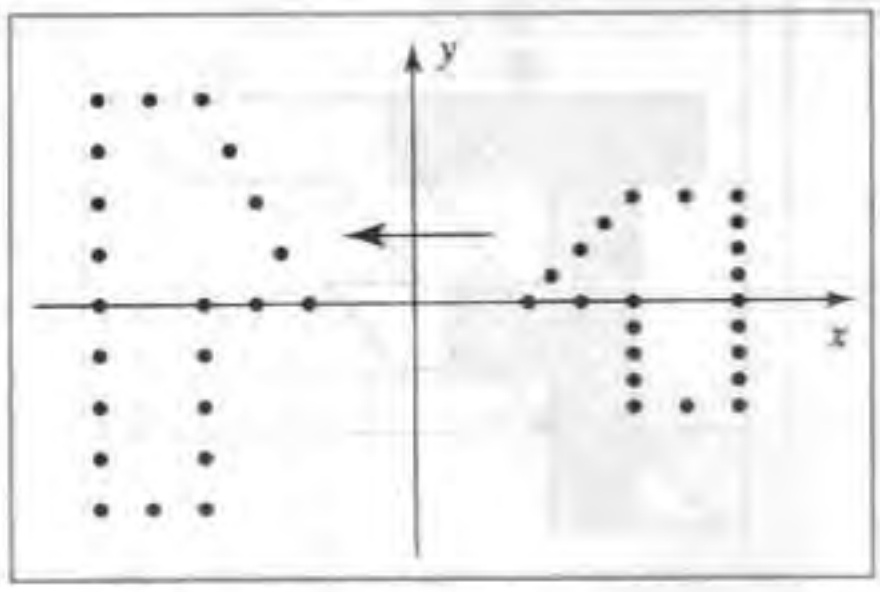


图 5.10　一个缩放和反射的组合

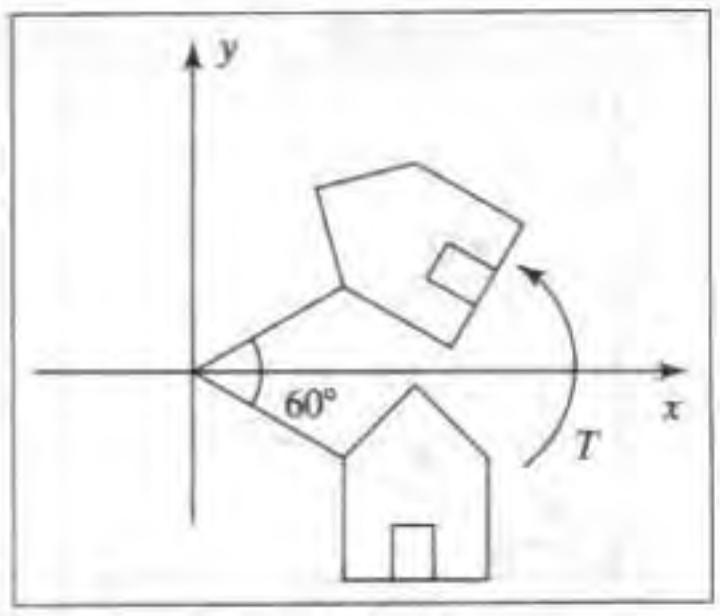


图 5.11　将一个点集旋转 60°

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

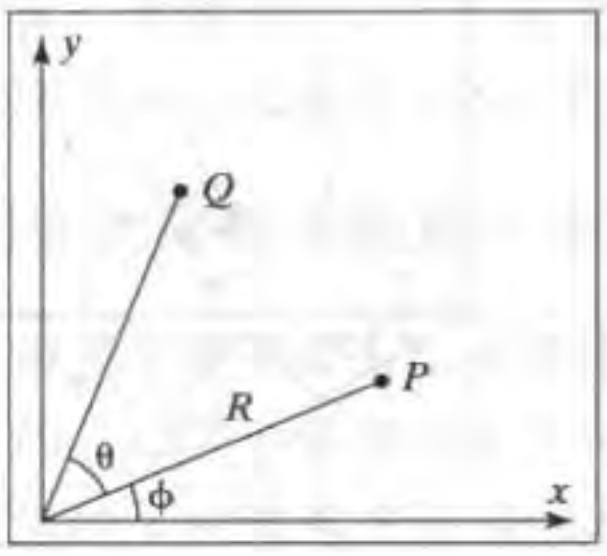


图 5.12 旋转映射的推导

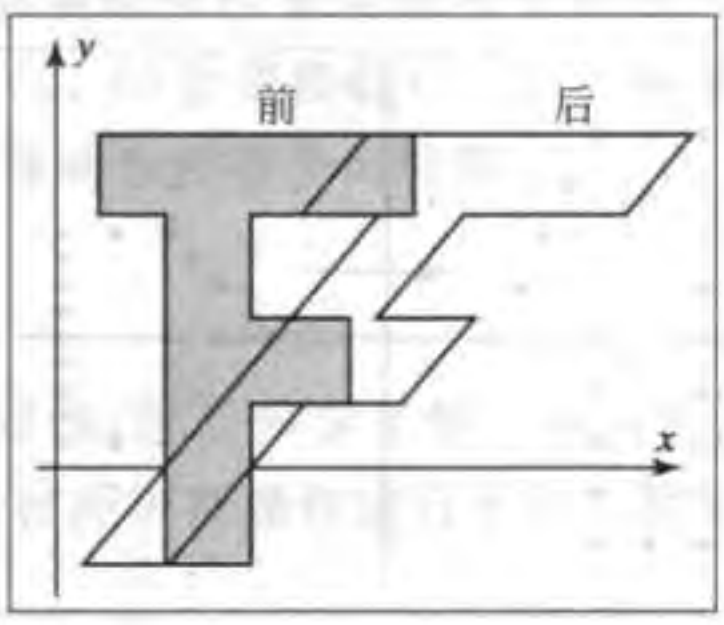


图 5.13 一个剪切的例子

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)。

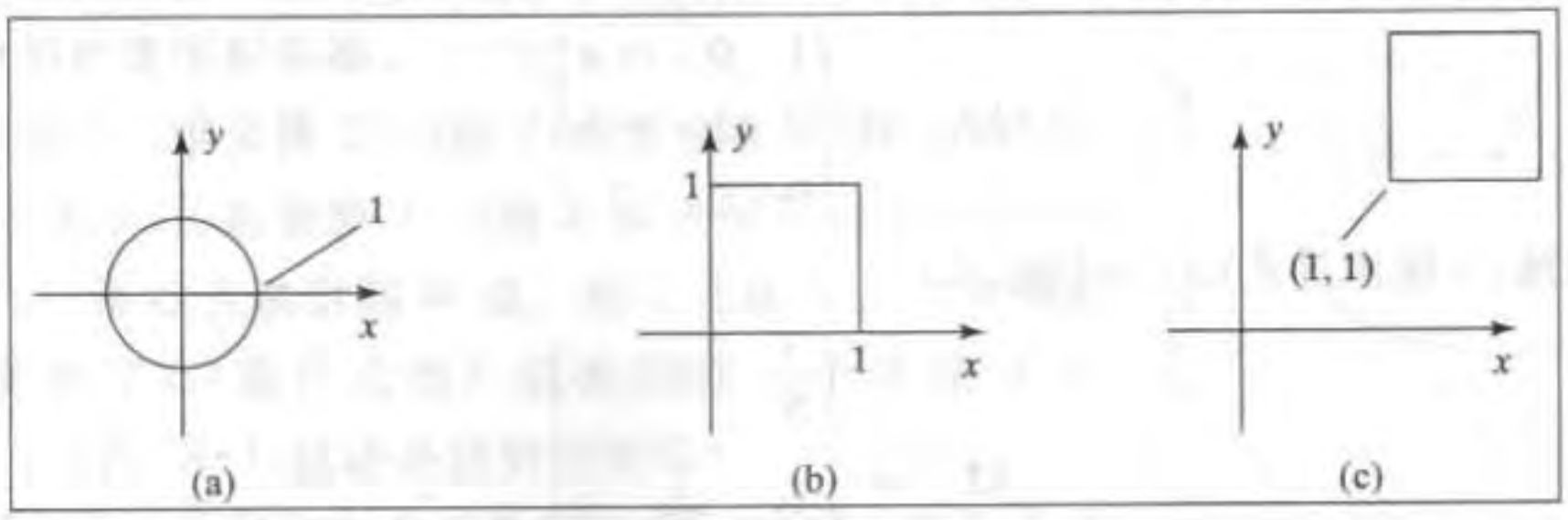


图 5.14 被缩放的物体

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？

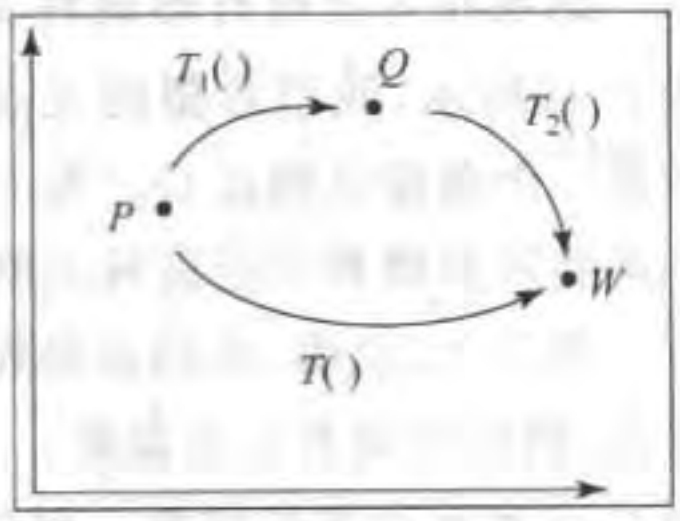


图 5.15 两个变换的组合

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。

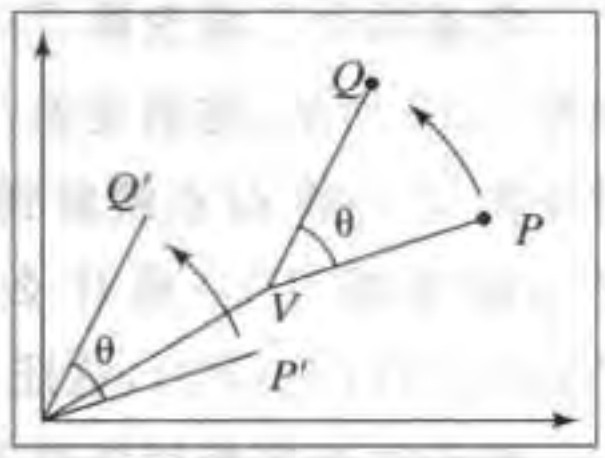


图 5.16 绕一个点旋转

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

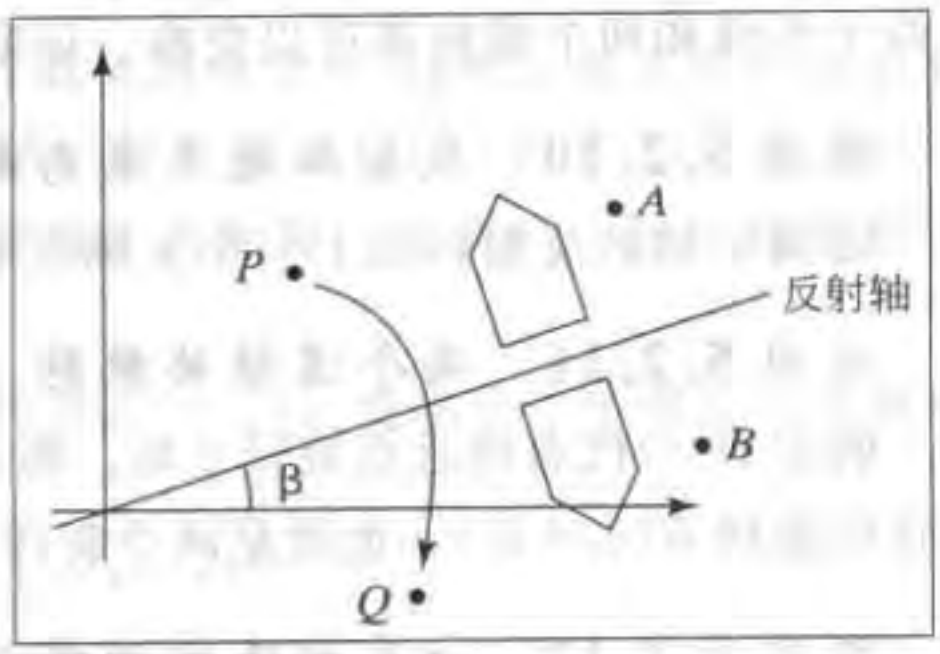


图 5.17 轴反射

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。

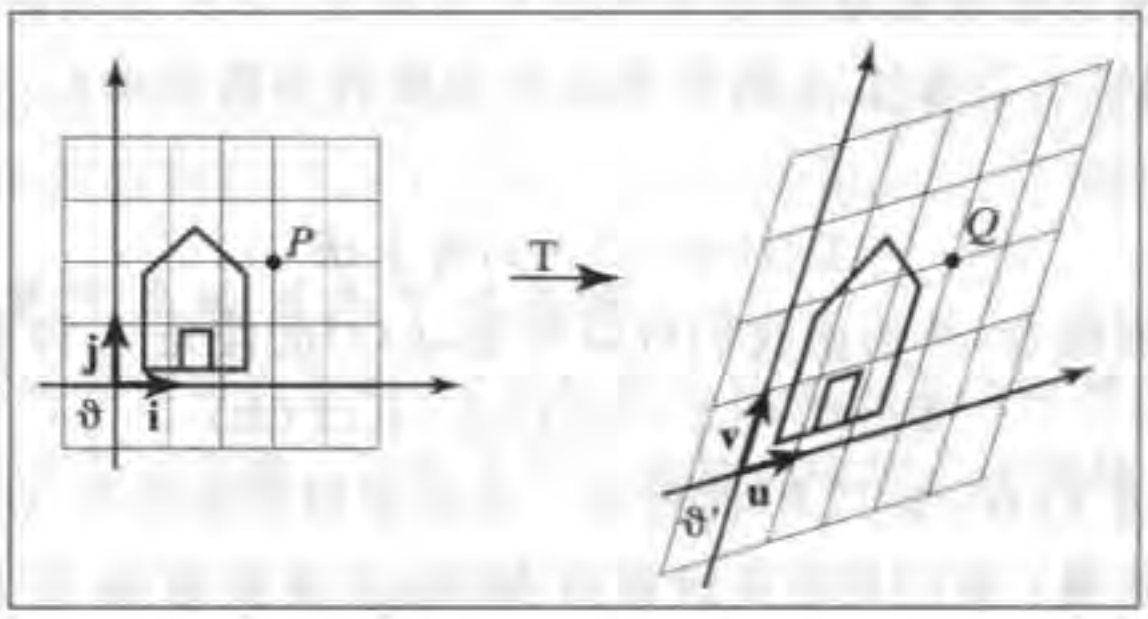


图 5.18 网格变换

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。

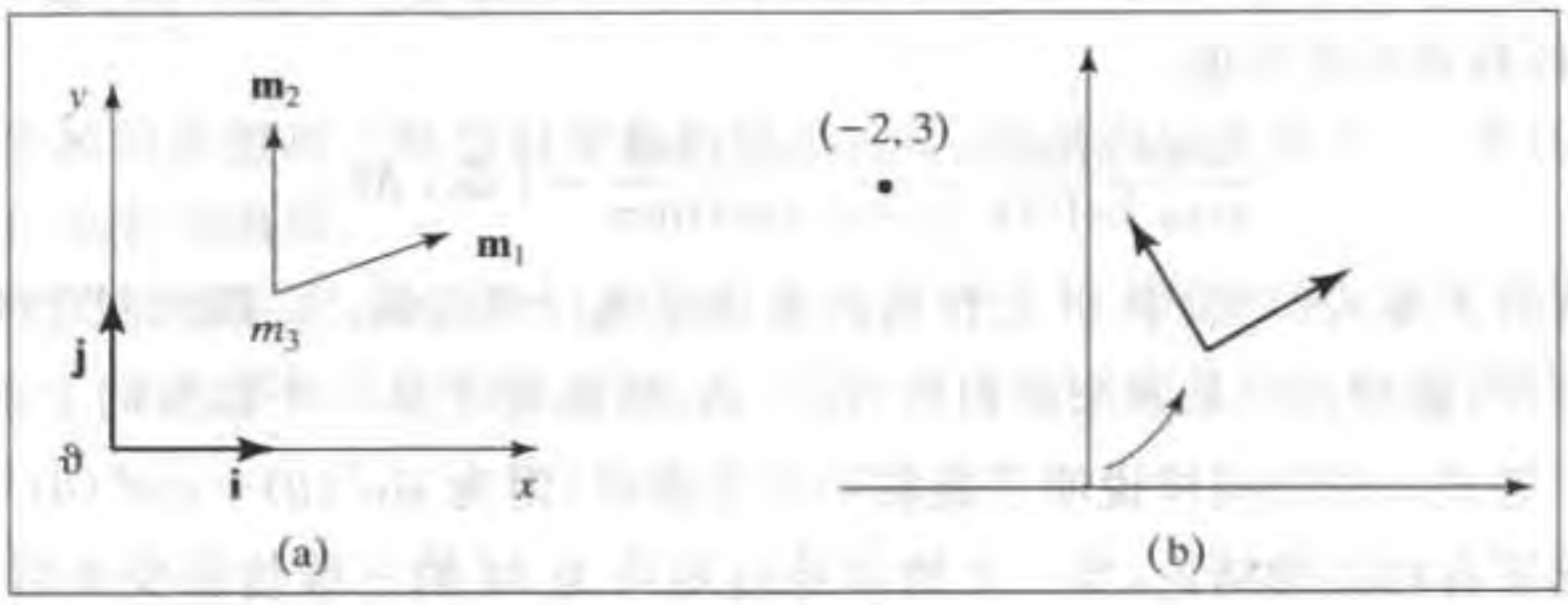


图 5.19　坐标变换

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。

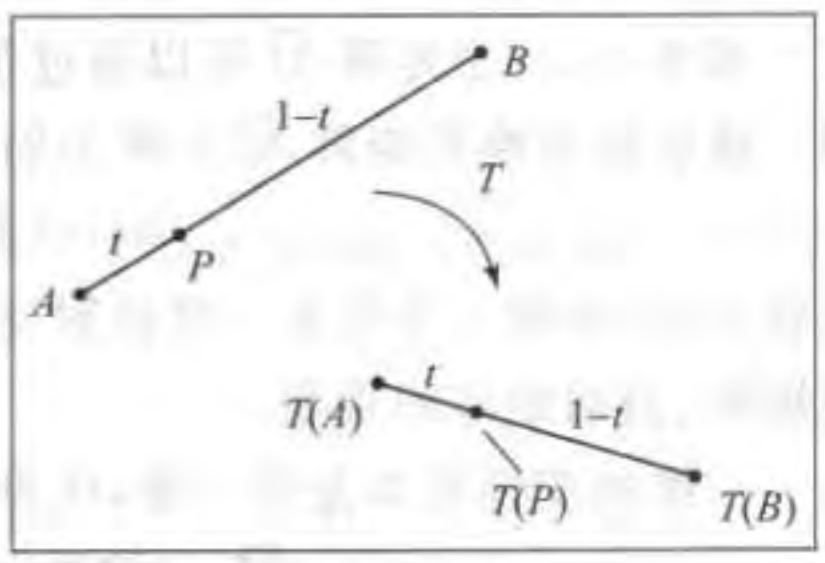


图 5.20　等比例变换

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。

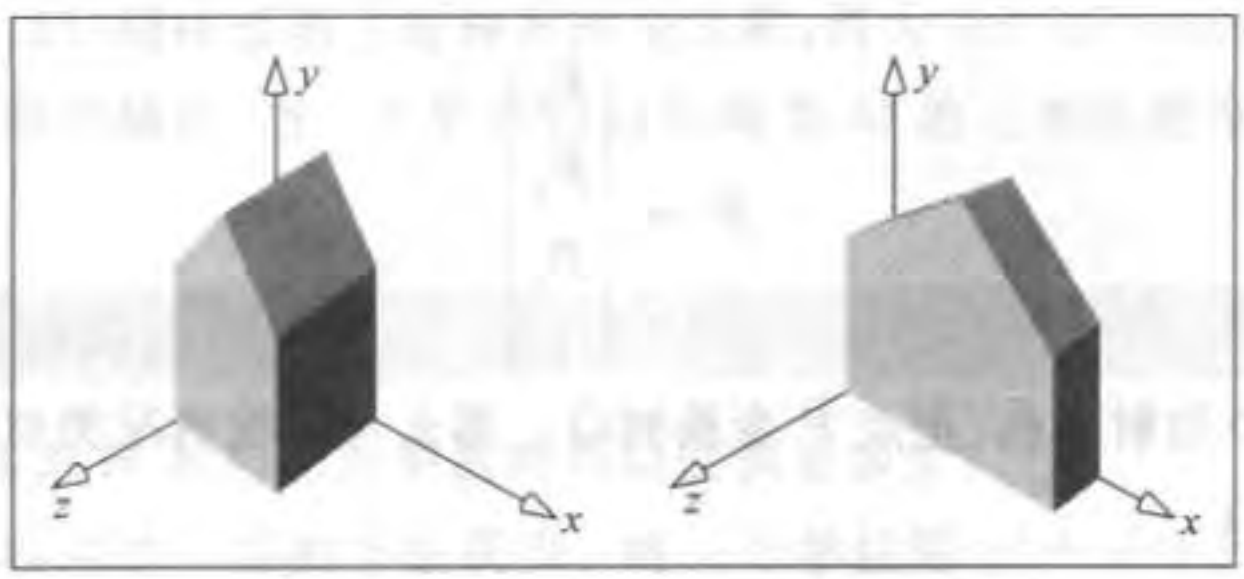


图 5.21 缩放

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。

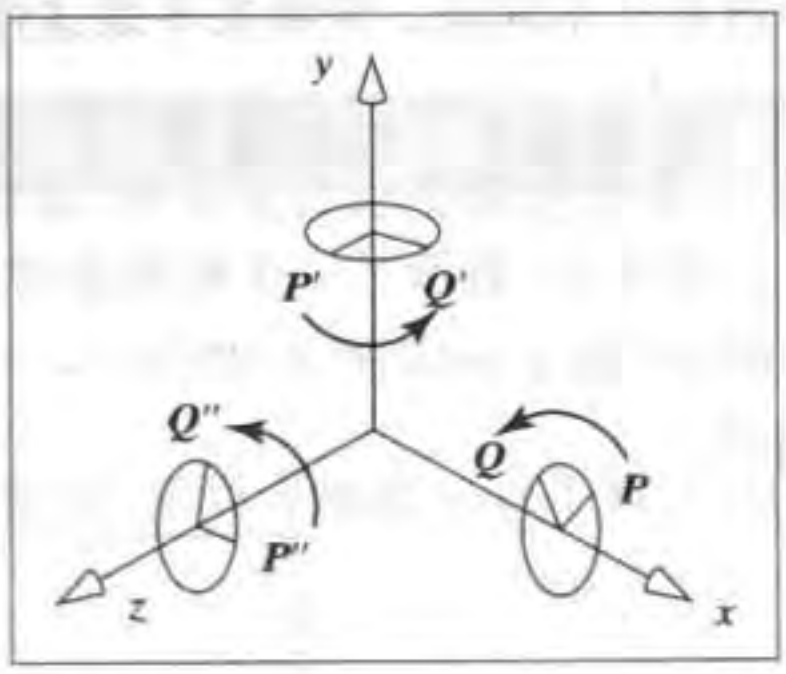


图 5.22　围绕 3 个轴的位置旋转

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。

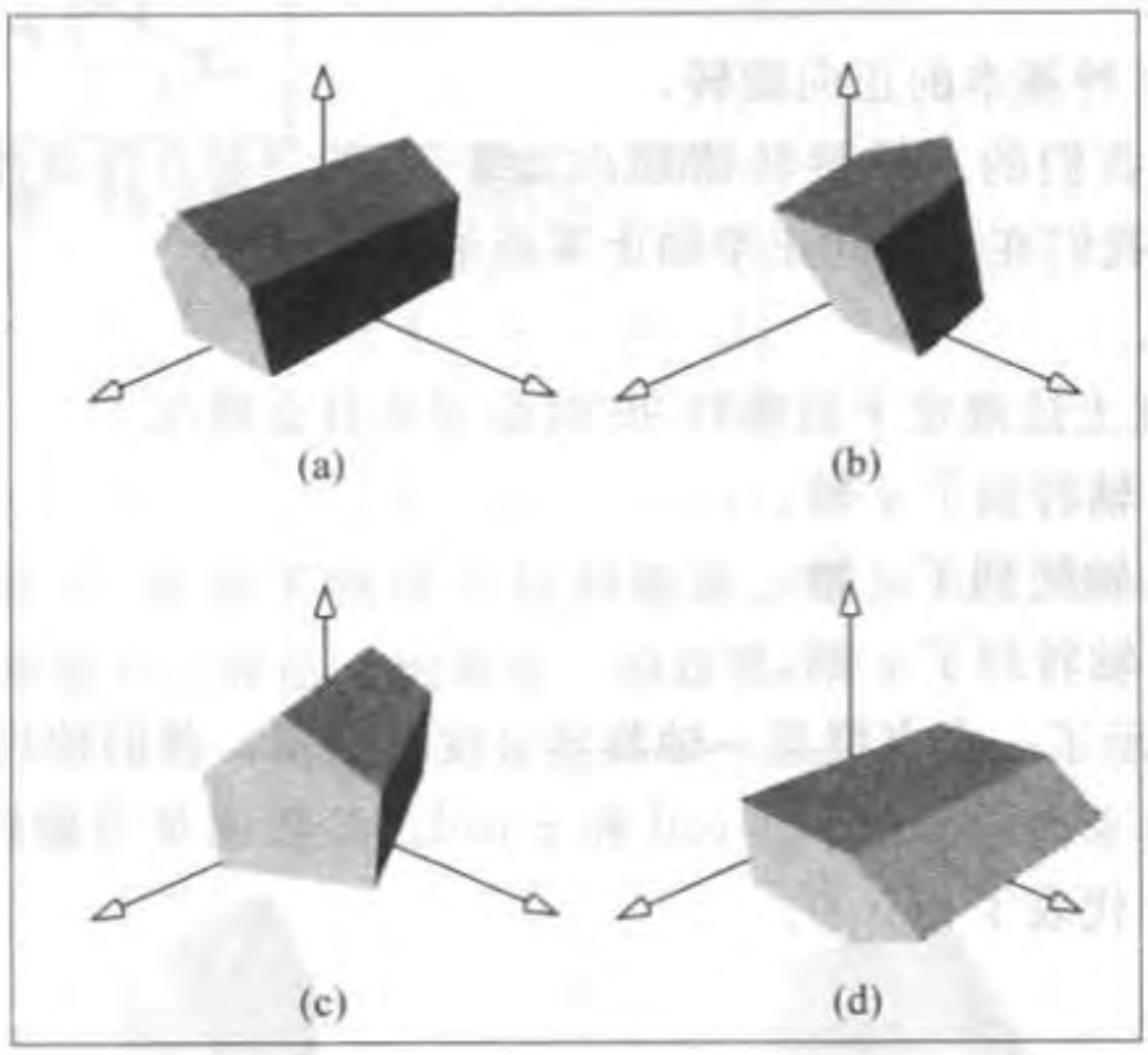


图 5.23 房子旋转

(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。

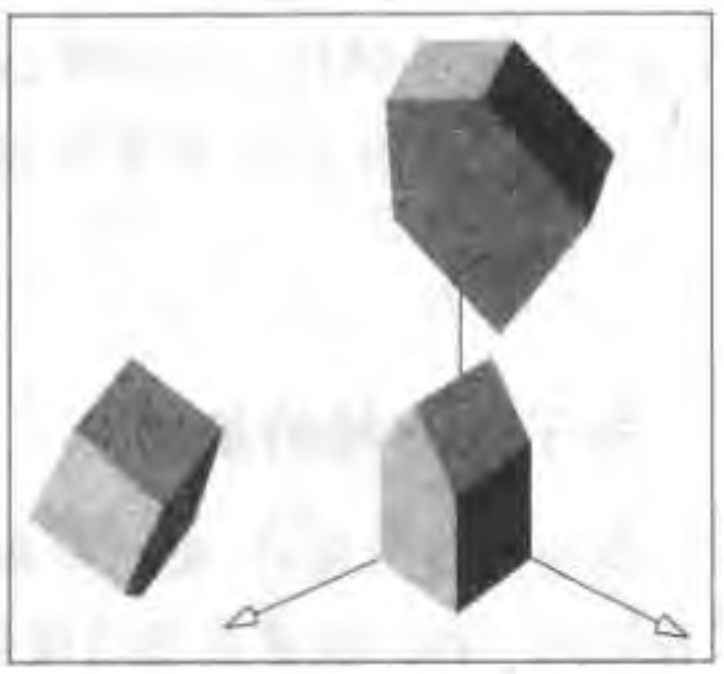


图 5.24　三维仿射变换组合

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。

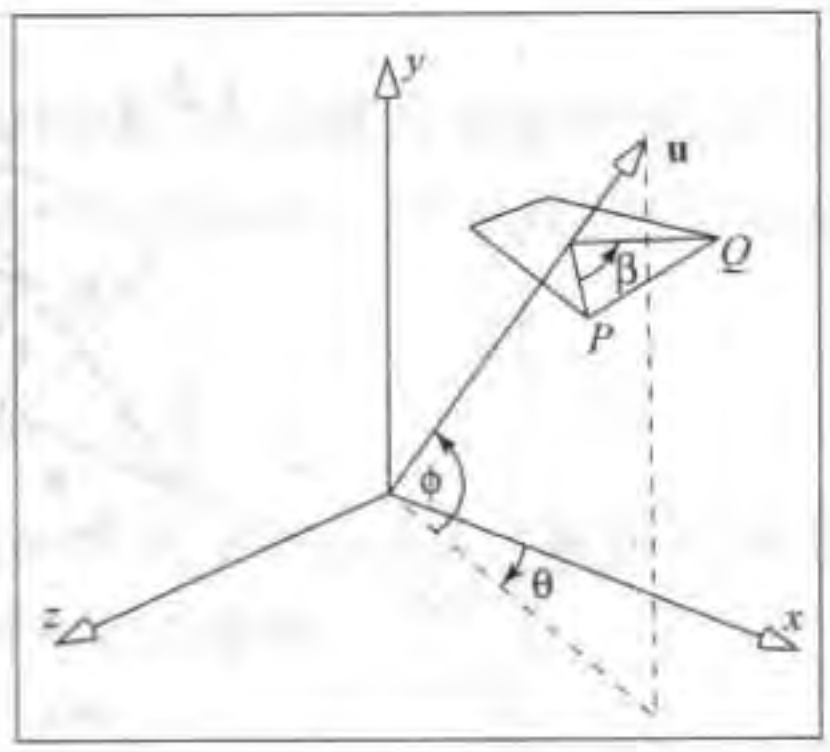


图 5.25 绕某轴旋转一定角度

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

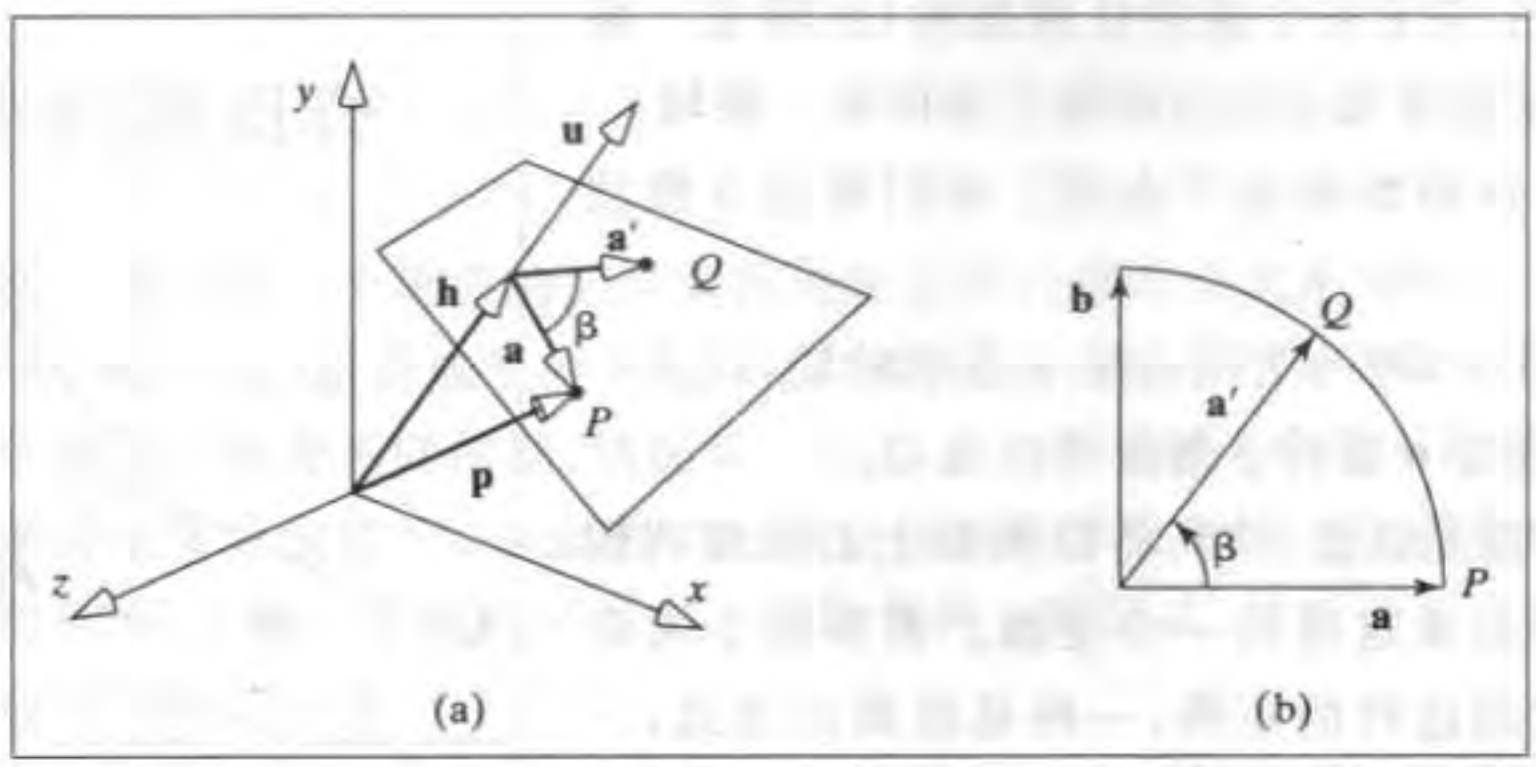


图 5.26 P 旋转到 Q

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。

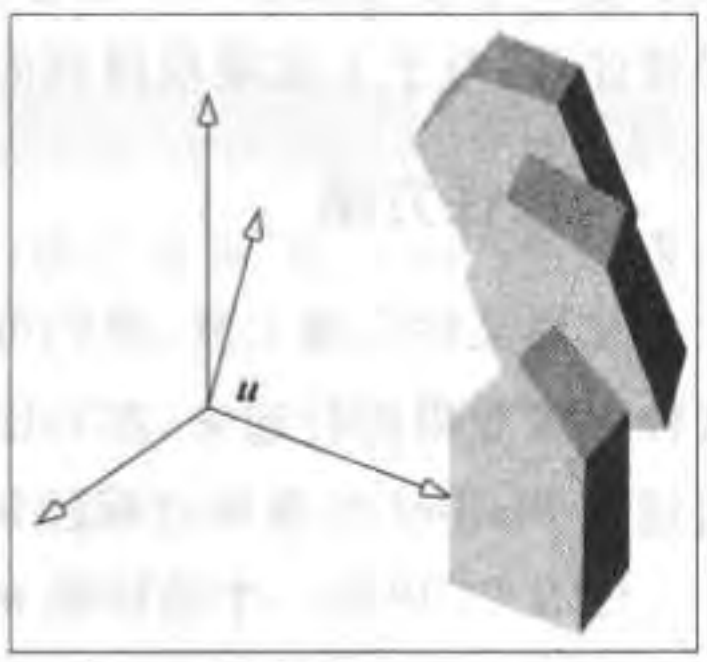


图 5.27 房子绕轴 **u** 的基本变换

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 $\mathbf{j}' = T(\mathbf{j})$。

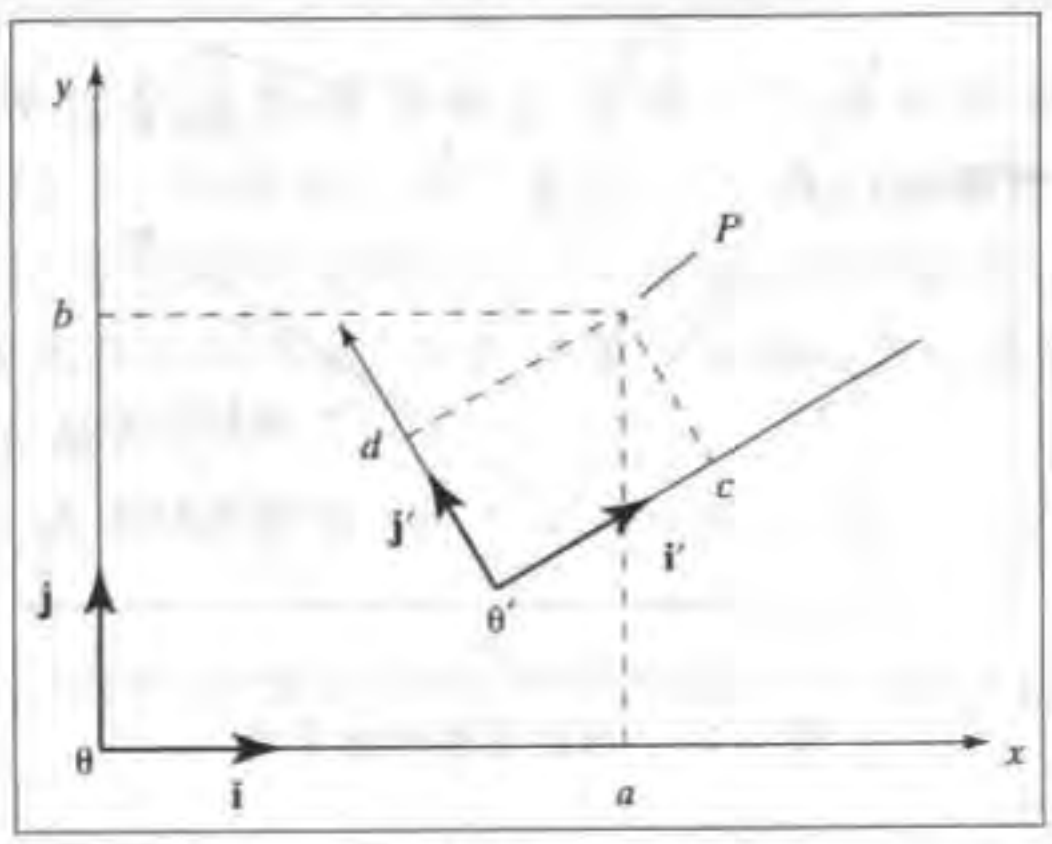


图 5.28　坐标变换

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。

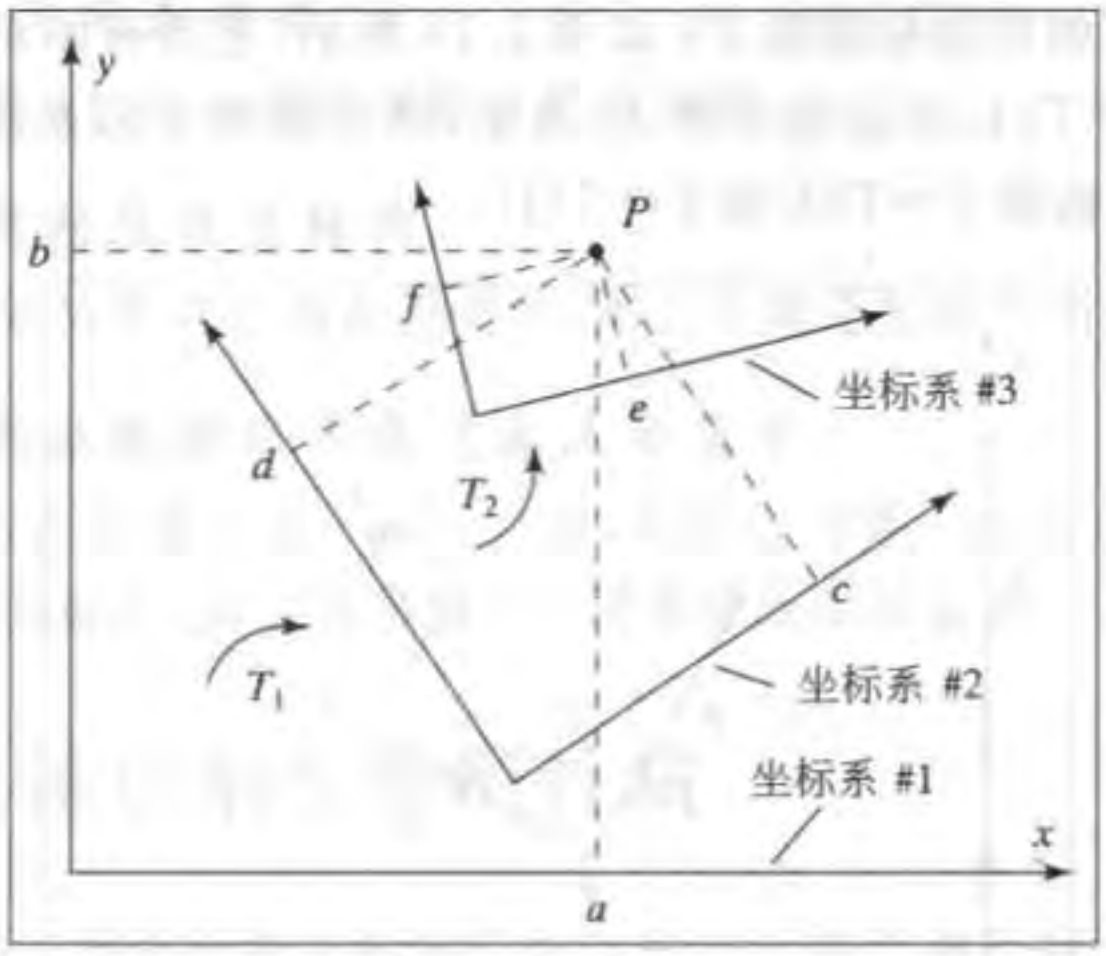


图 5.29 两次变换坐标系统

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'。

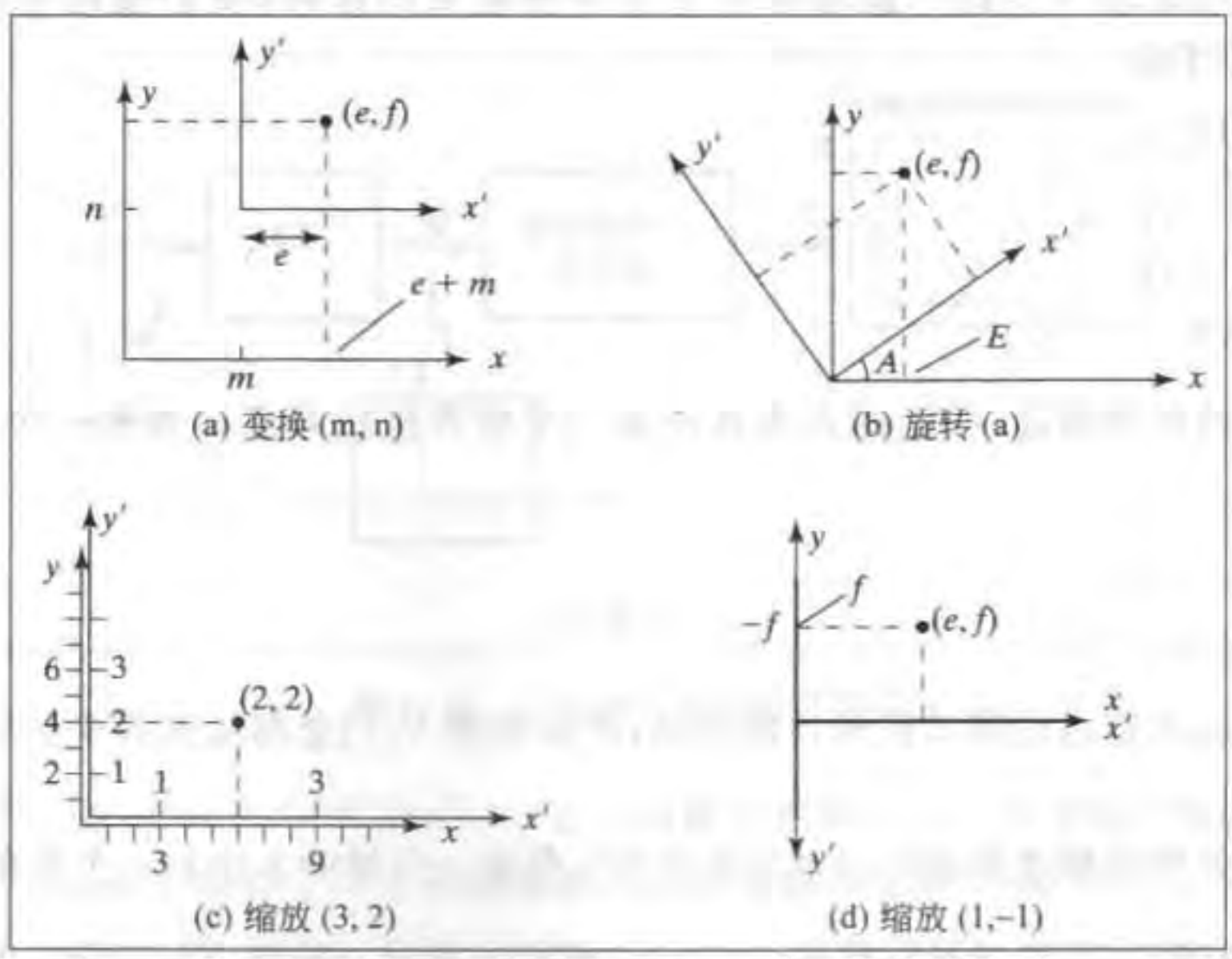


图 5.30 四个基本坐标系变换的效果

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。

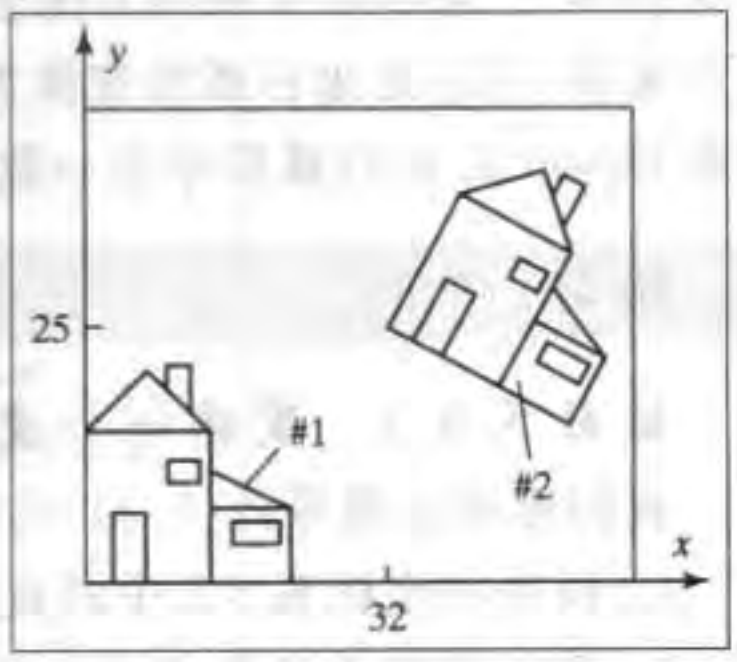


图 5.31 房子的旋转与变换

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)。

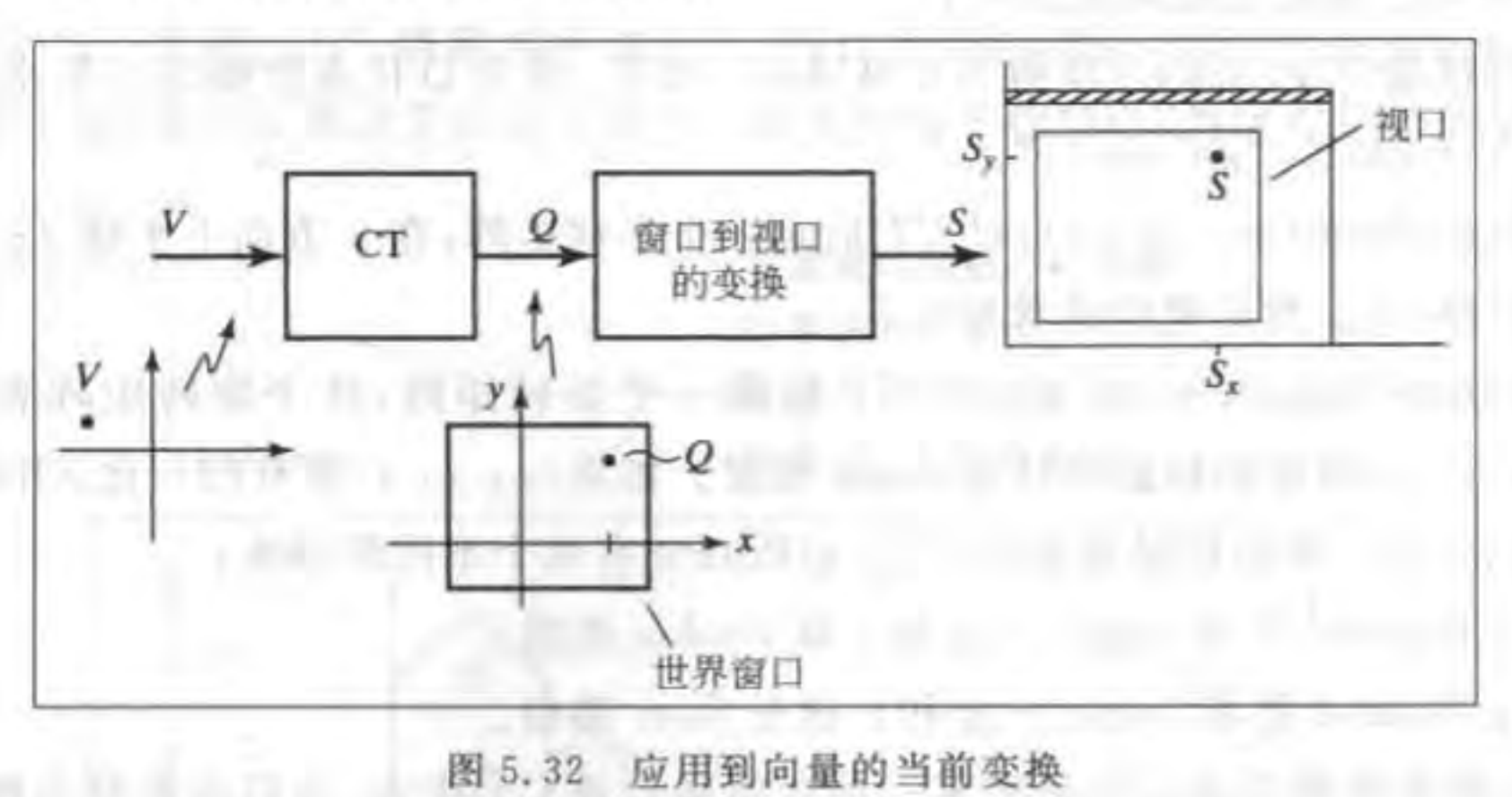


图 5.32　应用到向量的当前变换

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 $S_z=1$。

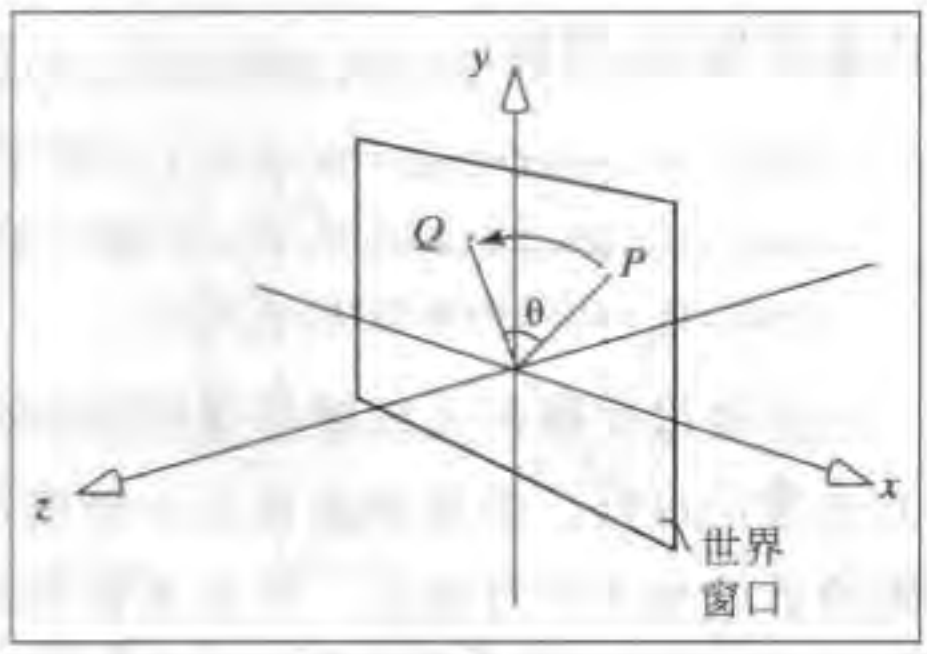


图 5.33 在 xy 平面上的二维绘图

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
```

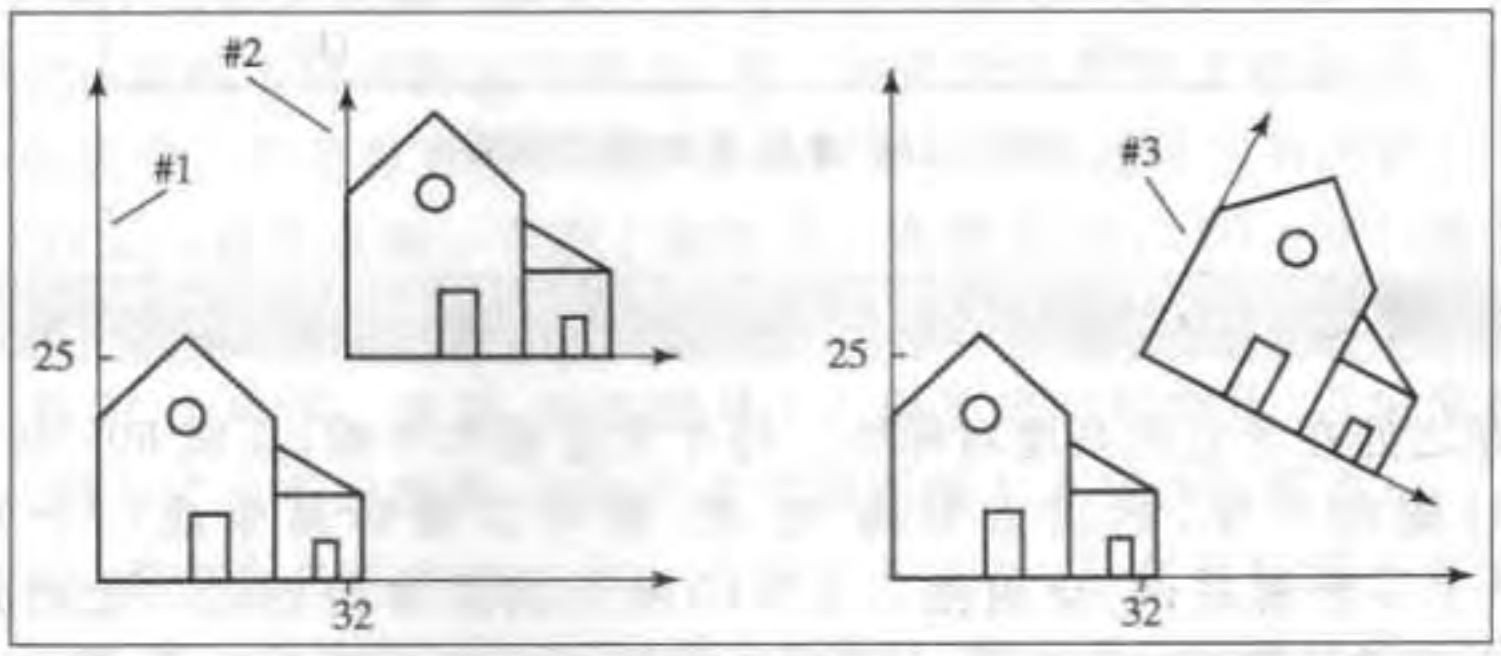


图 5.35 从坐标系的变换角度考虑

```
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。

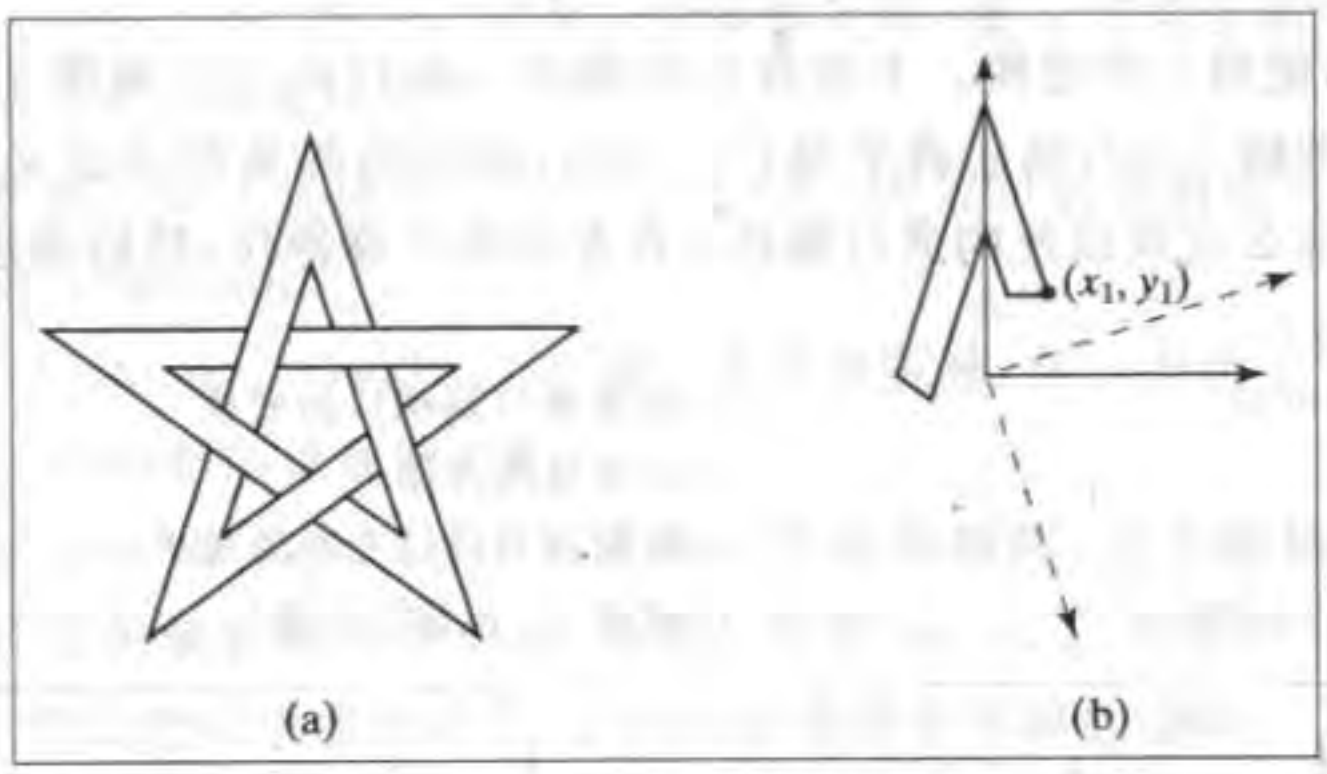


图 5.36 坐标系的连续旋转

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)。

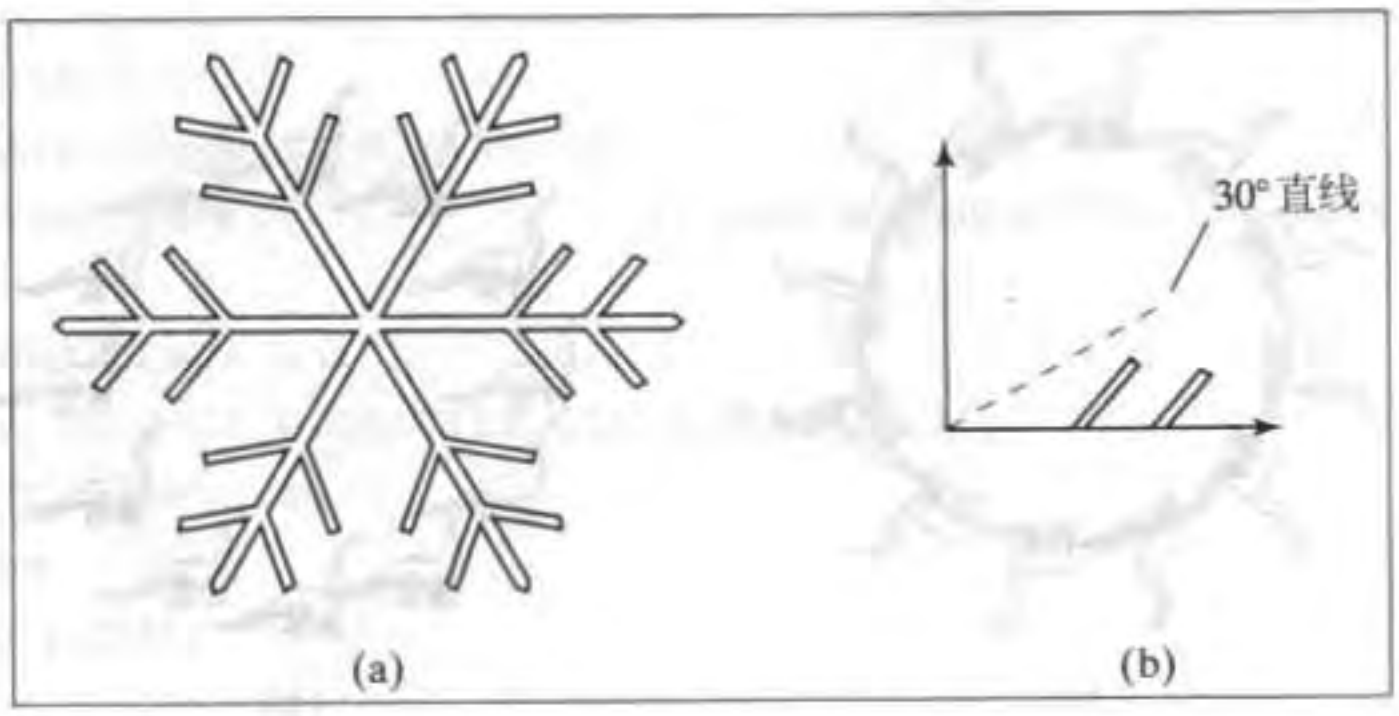


图 5.37 设计一片雪花

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°。

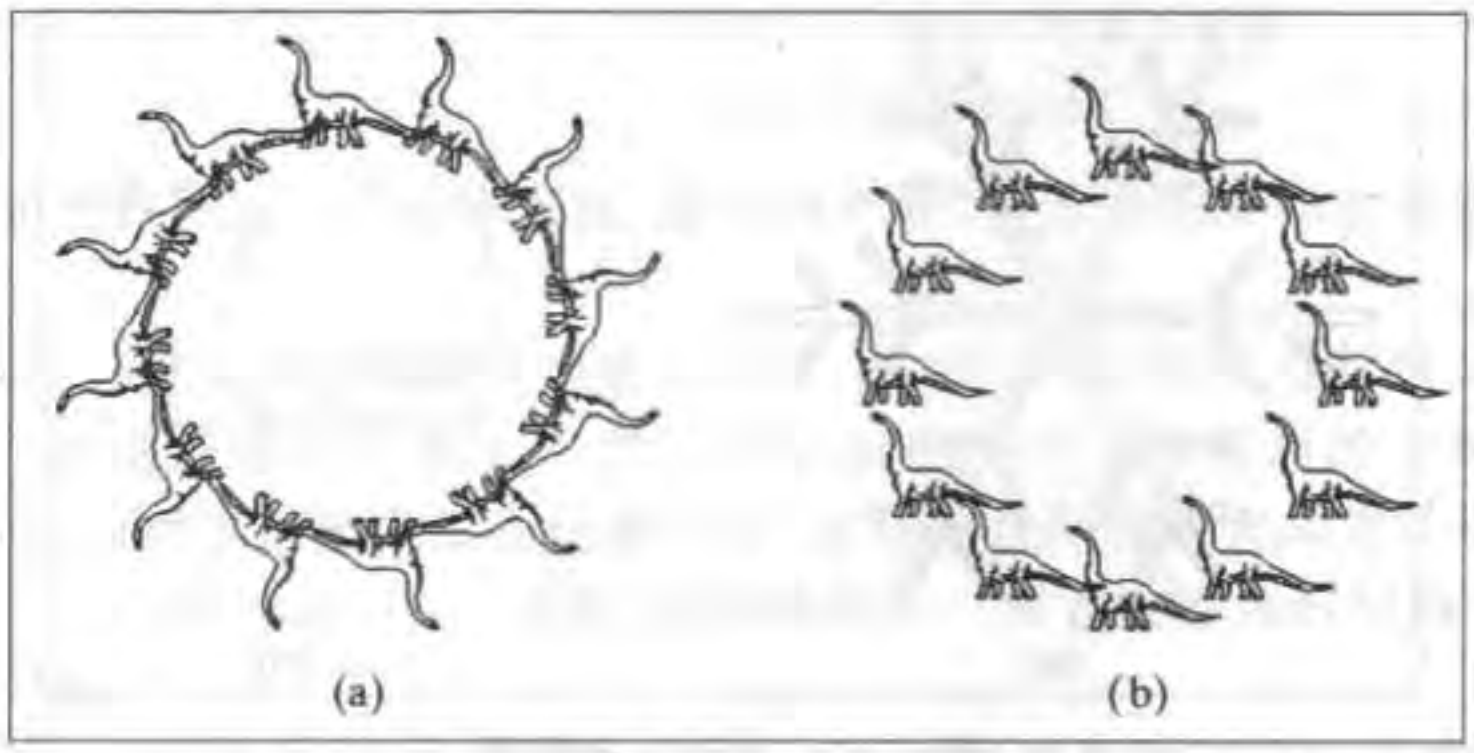


图 5.38 两个由恐龙组成的图案

(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。

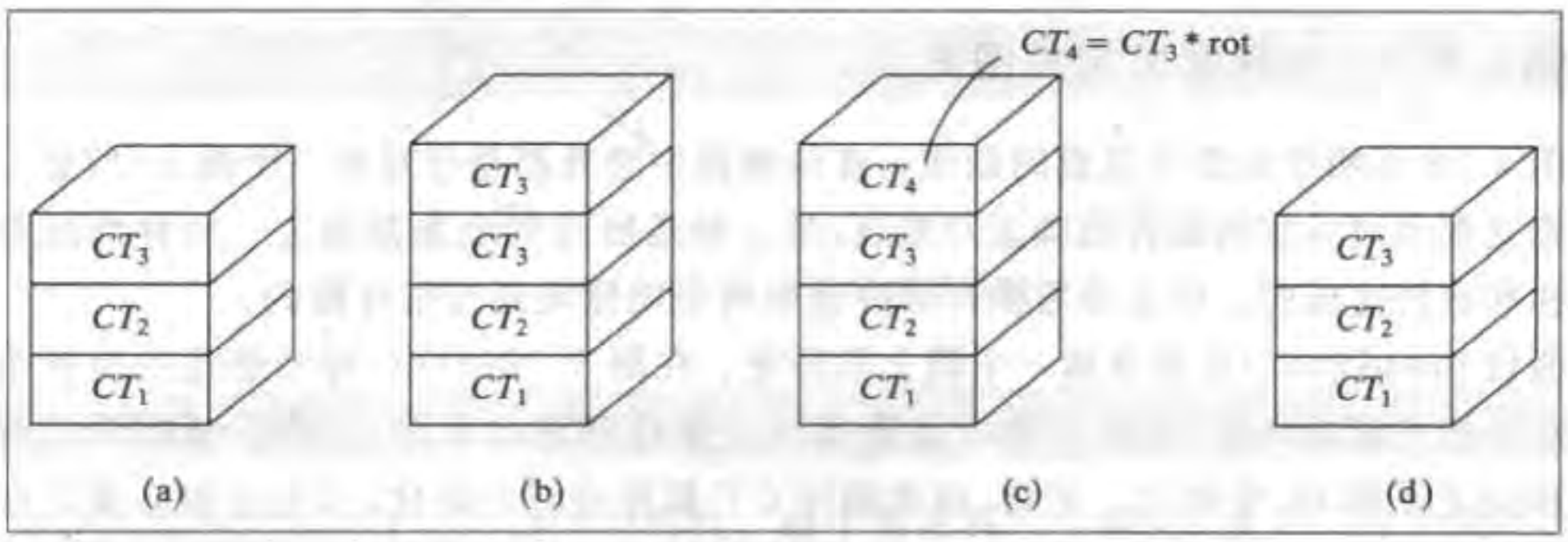


图 5.39 CT 栈

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。

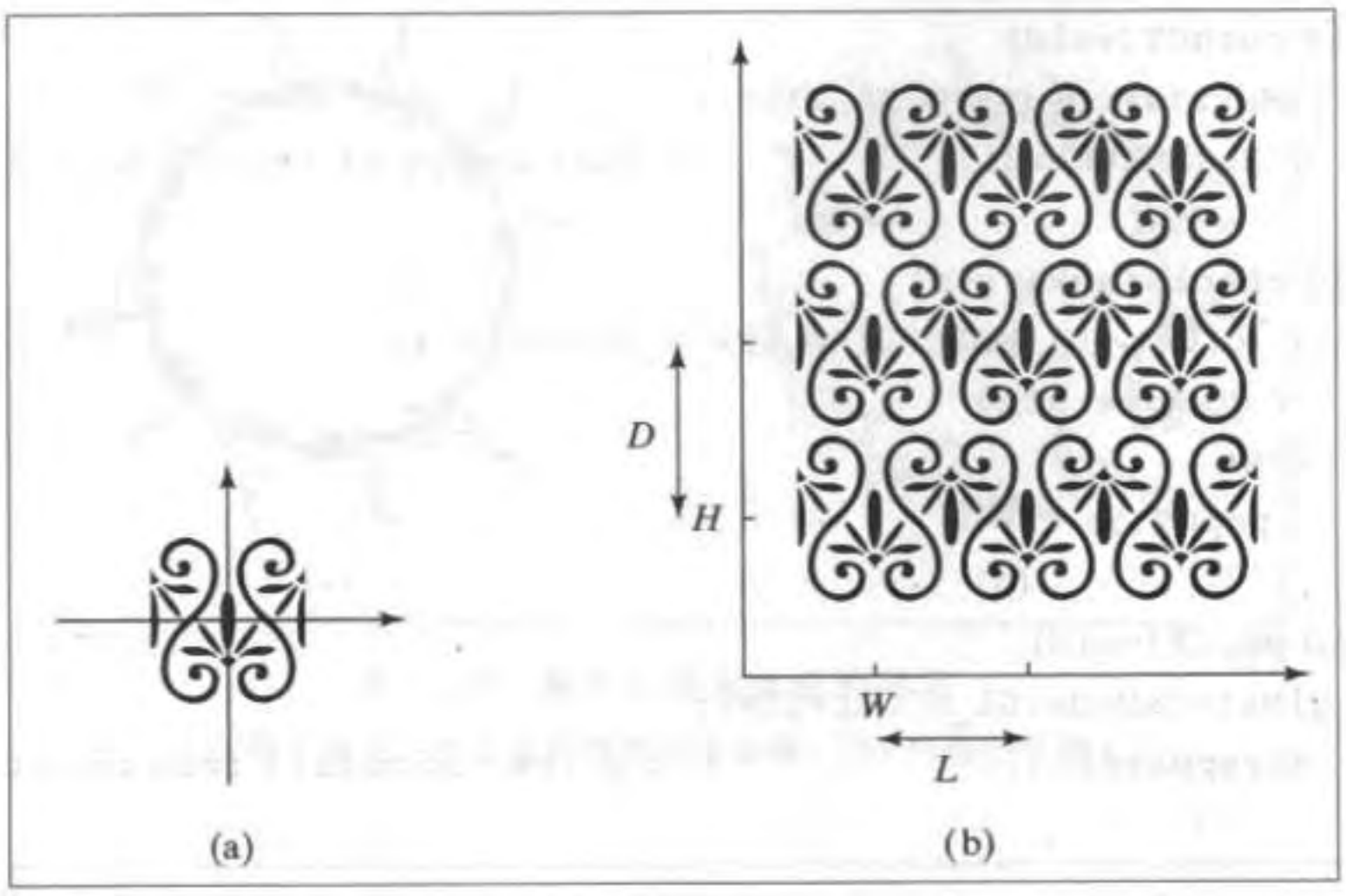


图 5.41 用基本图案制作拼贴

```
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

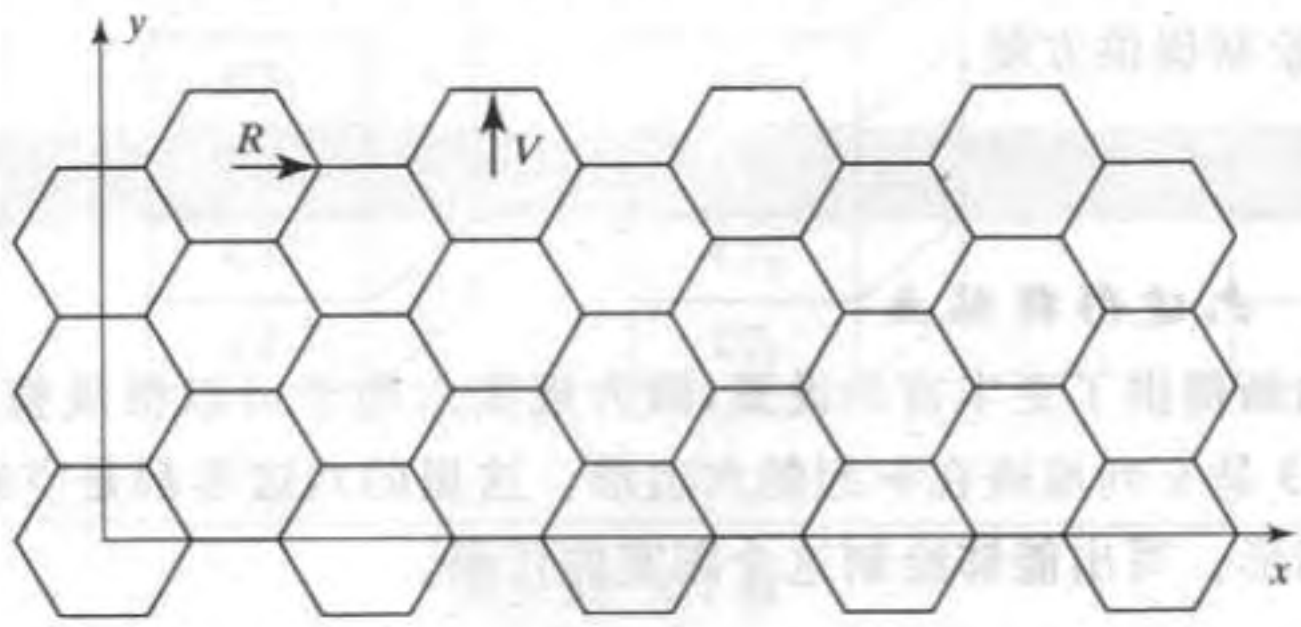


图 5.43 六边形的拼贴画

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

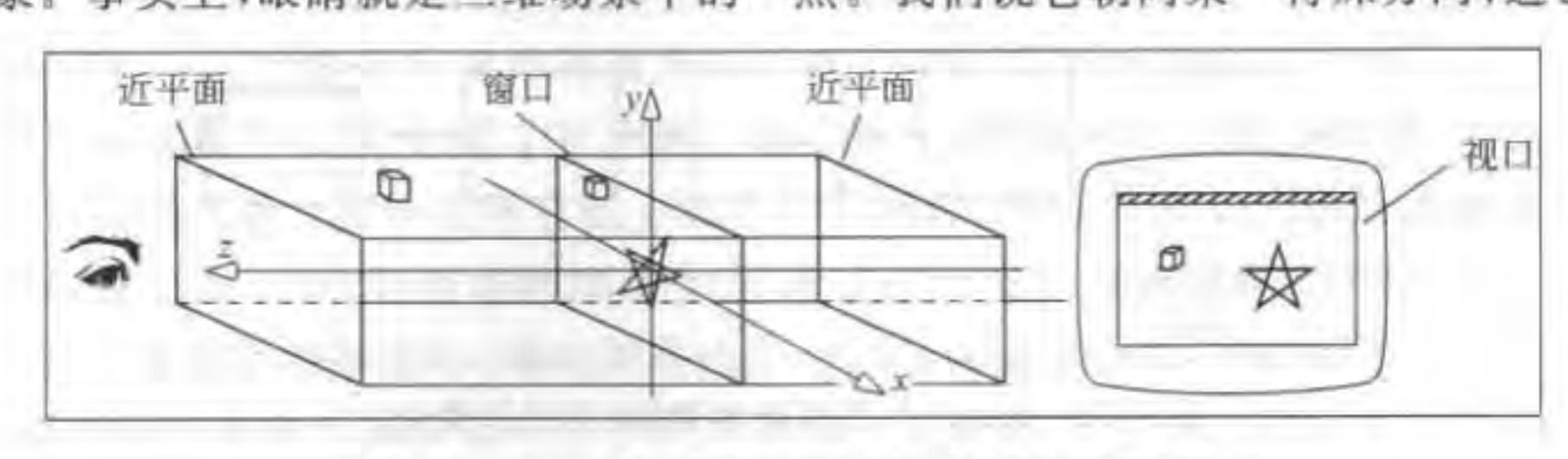


图 5.44　OpenGL 中用于二维绘图的简单观察过程

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。

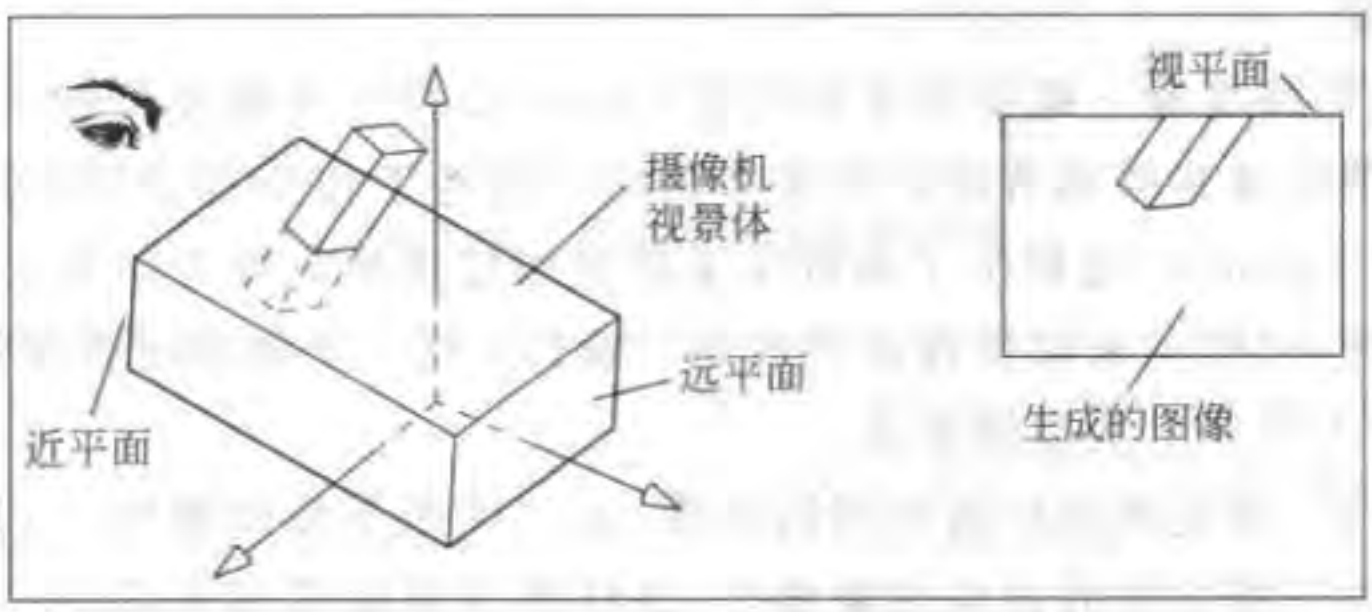


图 5.45 生成某个场景的平行视图的摄像机

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

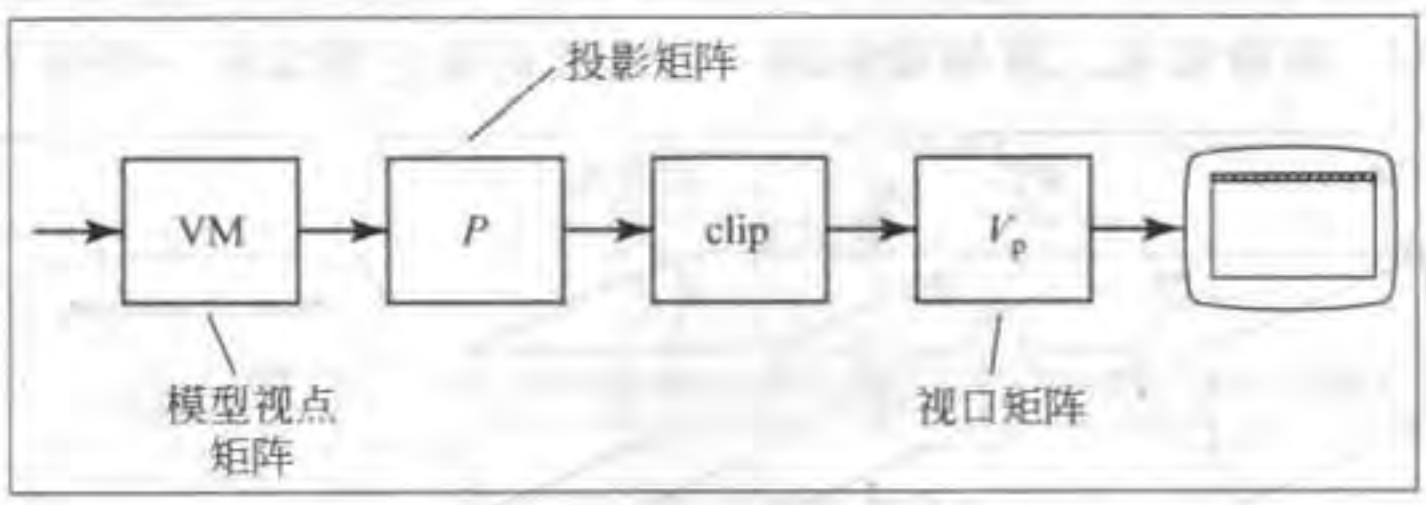


图 5.46 OpenGL 图形绘制管道(已稍作简化)

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。

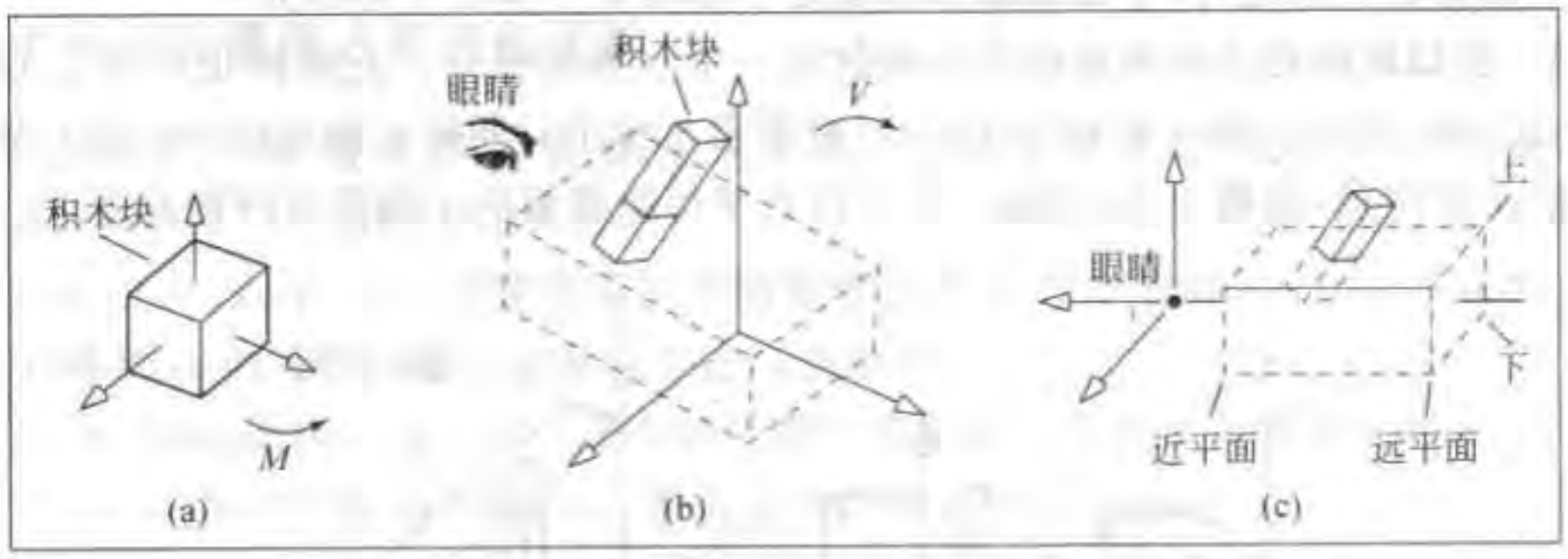


图 5.47 图形绘制管道中模型视点矩阵的效果

(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）。

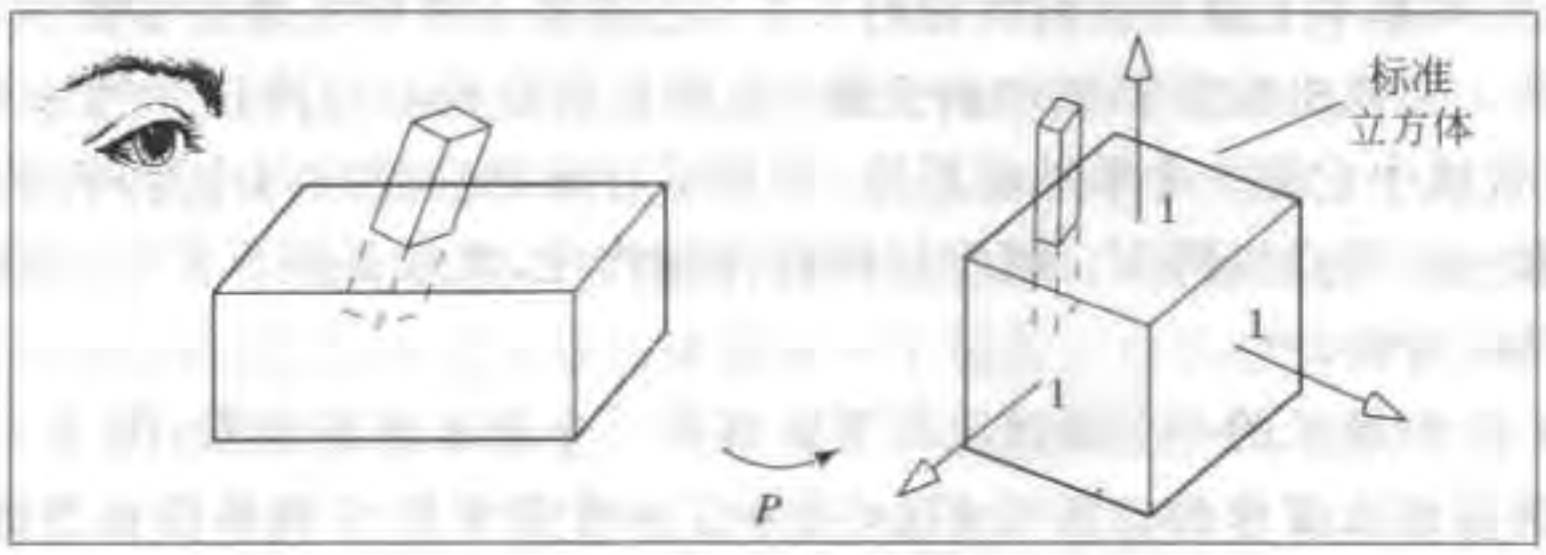


图 5.48 投影矩阵的效果

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。

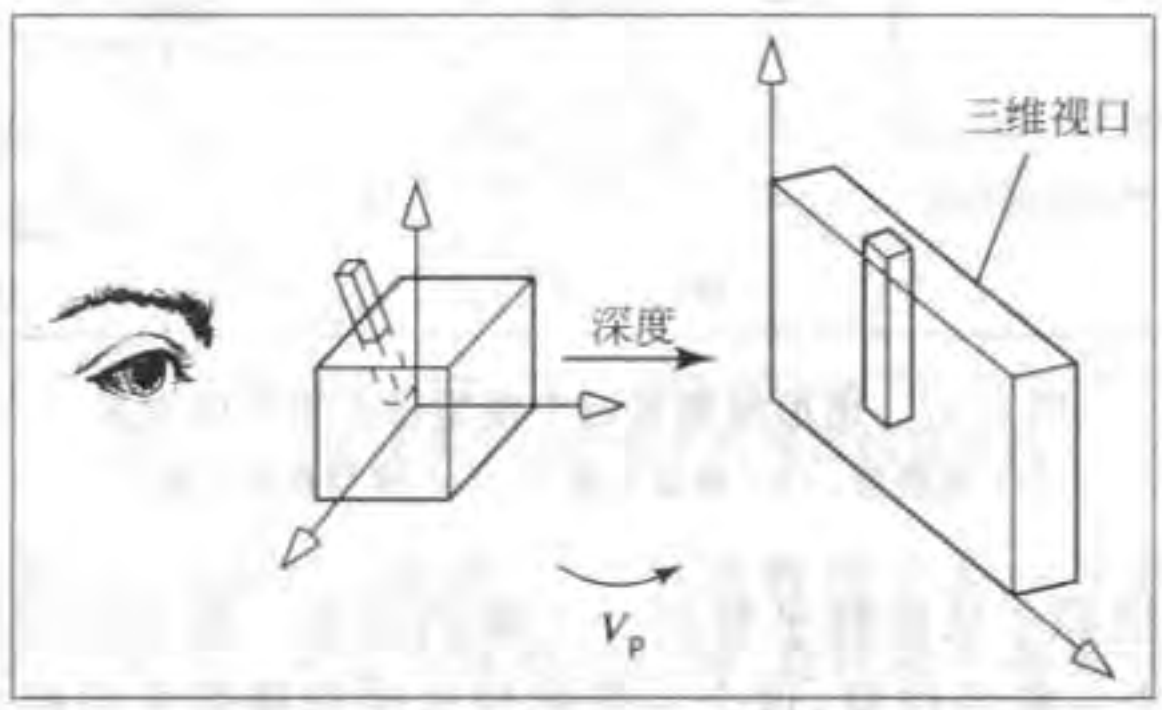


图 5.49 视口矩阵的效果

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))。

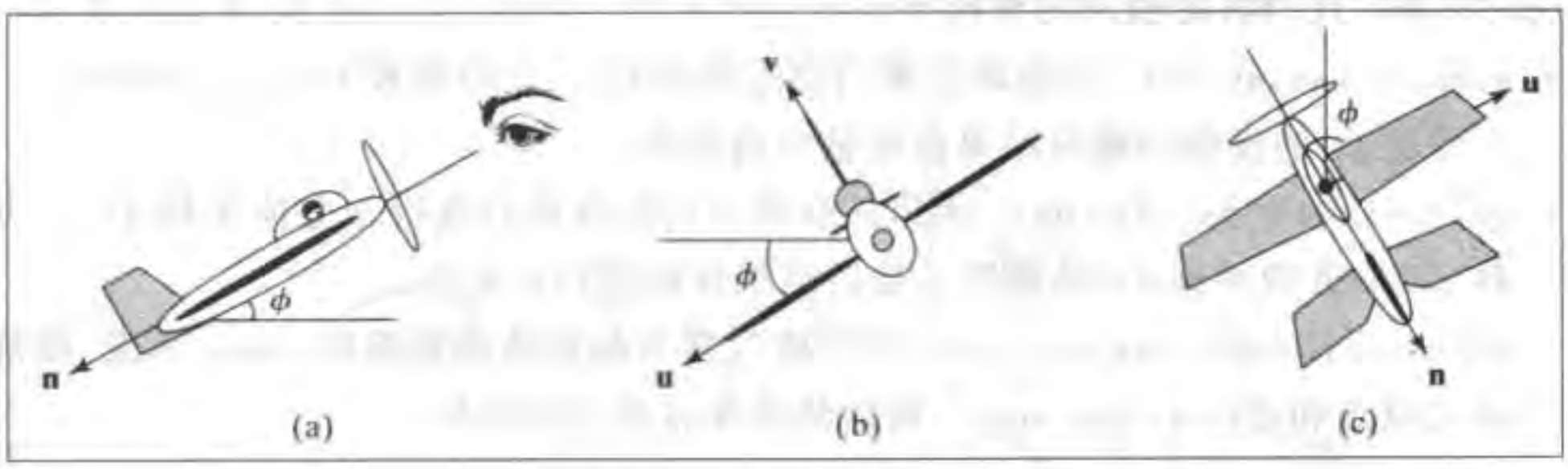


图 5.51 飞机旋转的描述

(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。

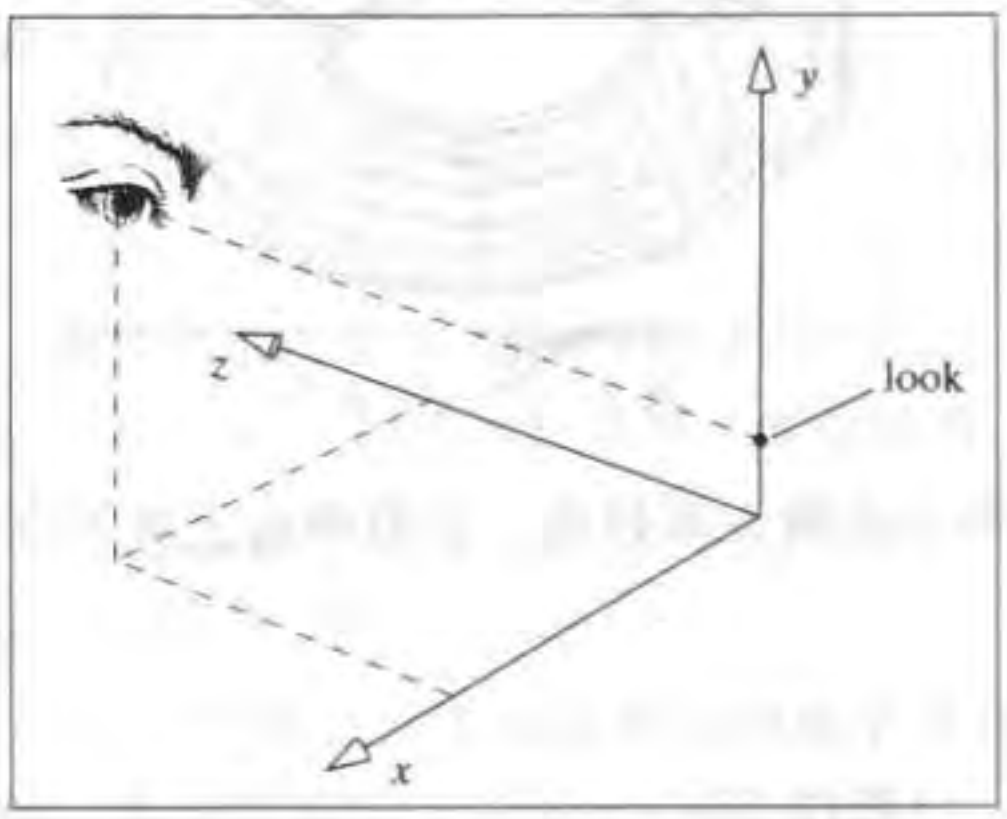


图 5.52 用 gluLookAt 设置一个摄像机

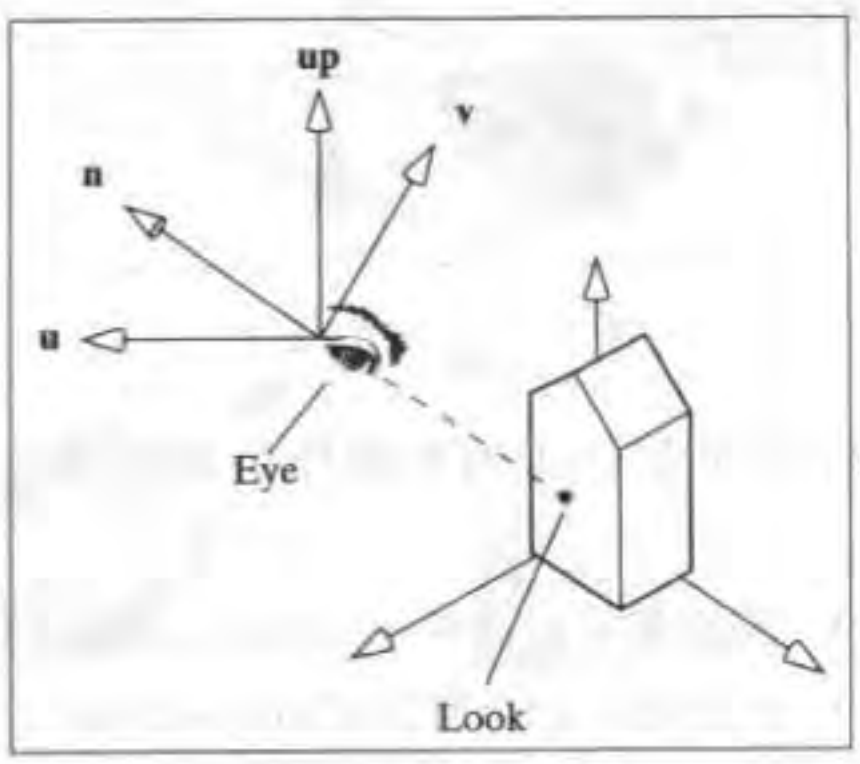


图 5.53 在摄像机坐标中转换世界坐标系统

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。

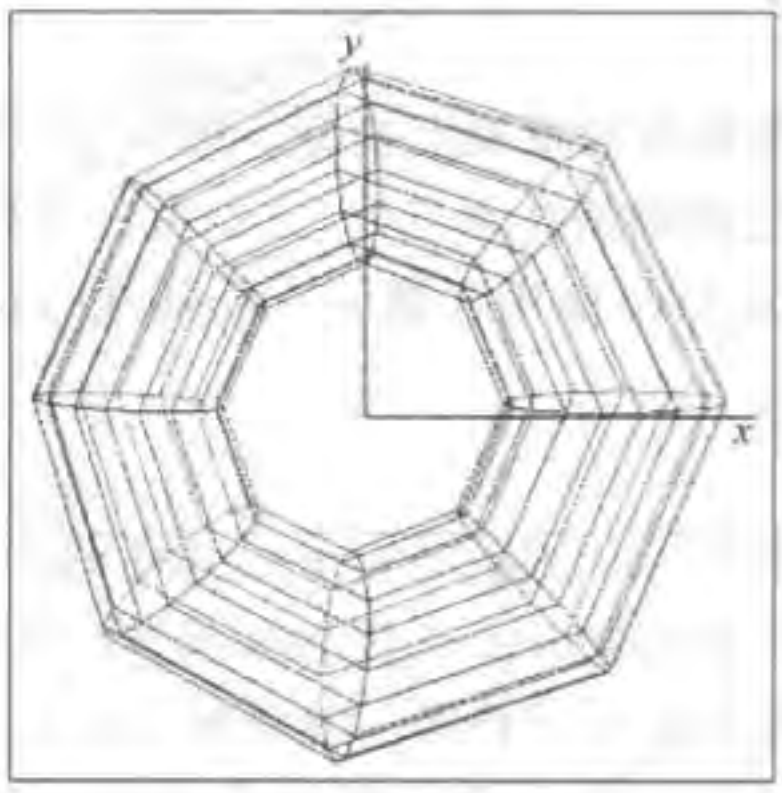


图 5.55 一个花环

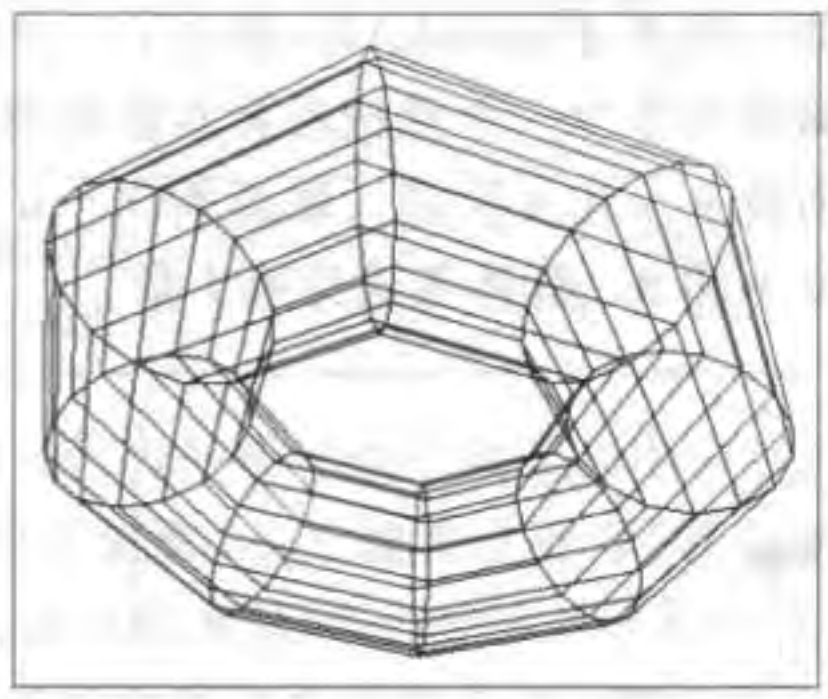


图 5.56 一个花环

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);
```

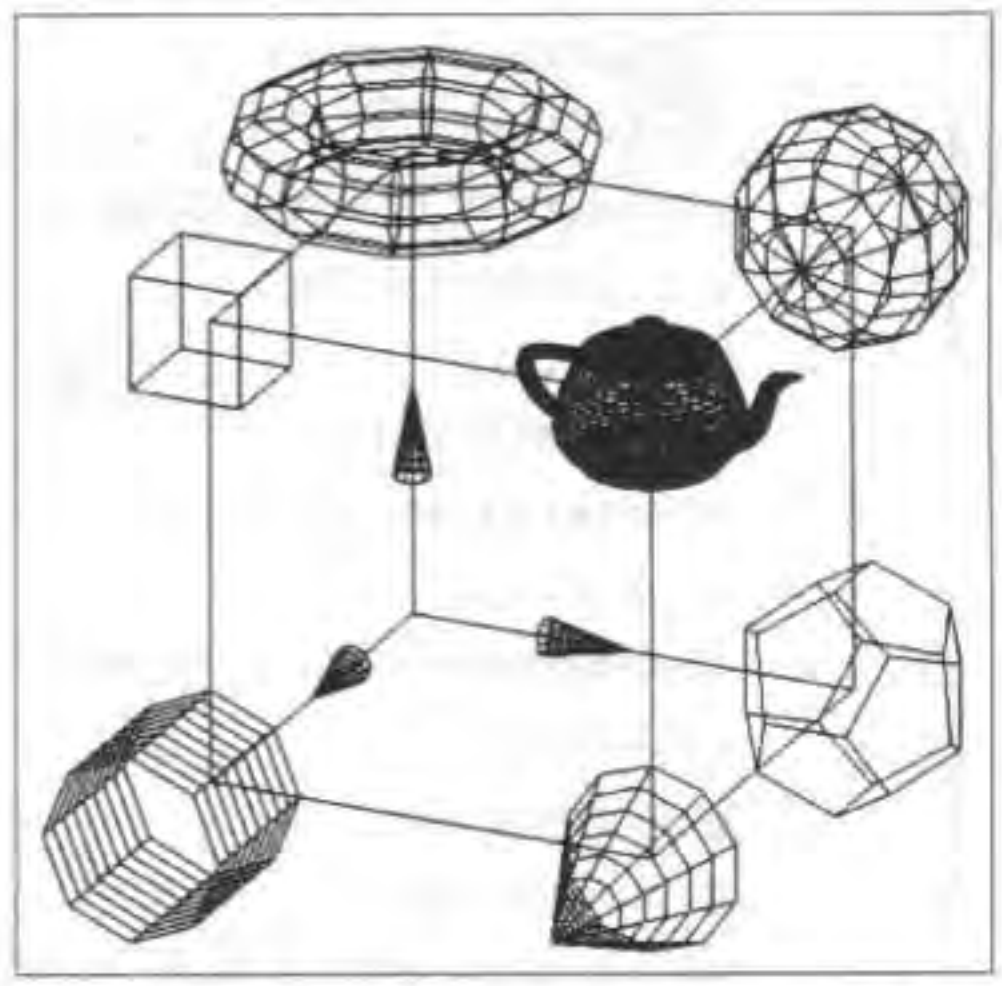


图 5.57 几个物体位于一个单位立方体的角

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。

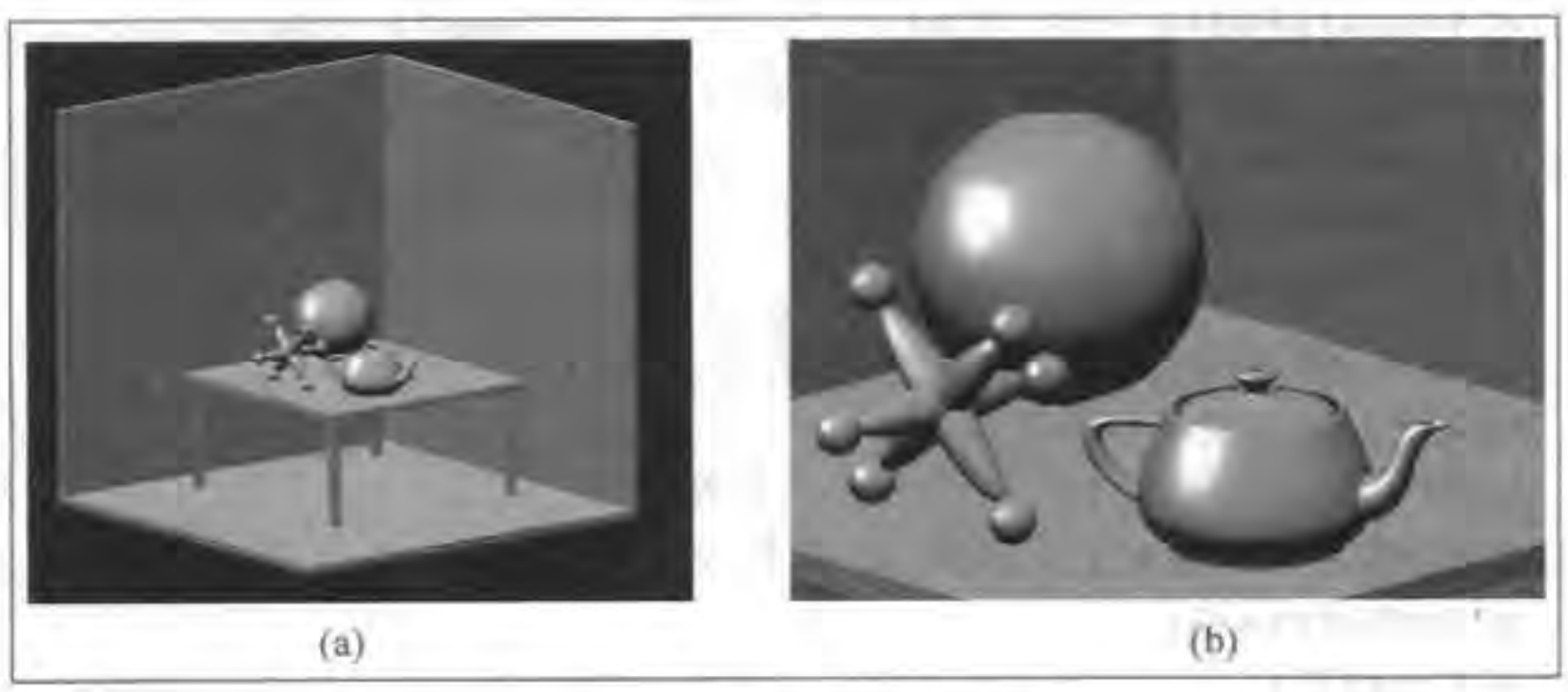


图 5.59 一个简单的三维场景

(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。

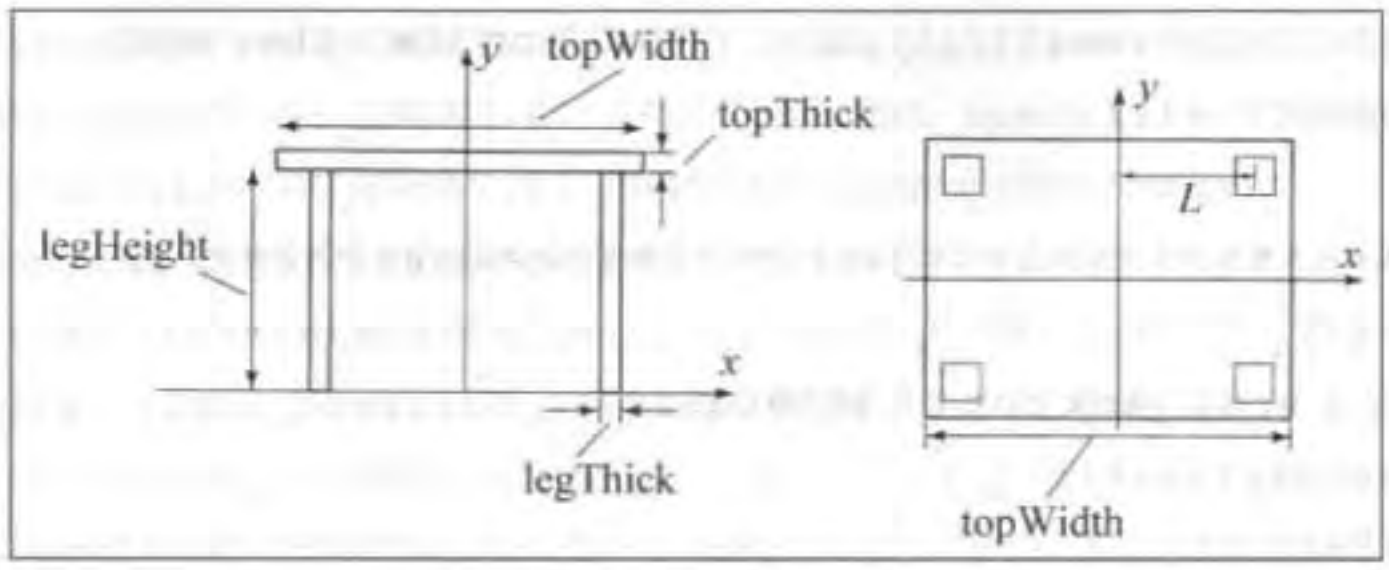


图 5.60 桌子的设计

```
#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
```

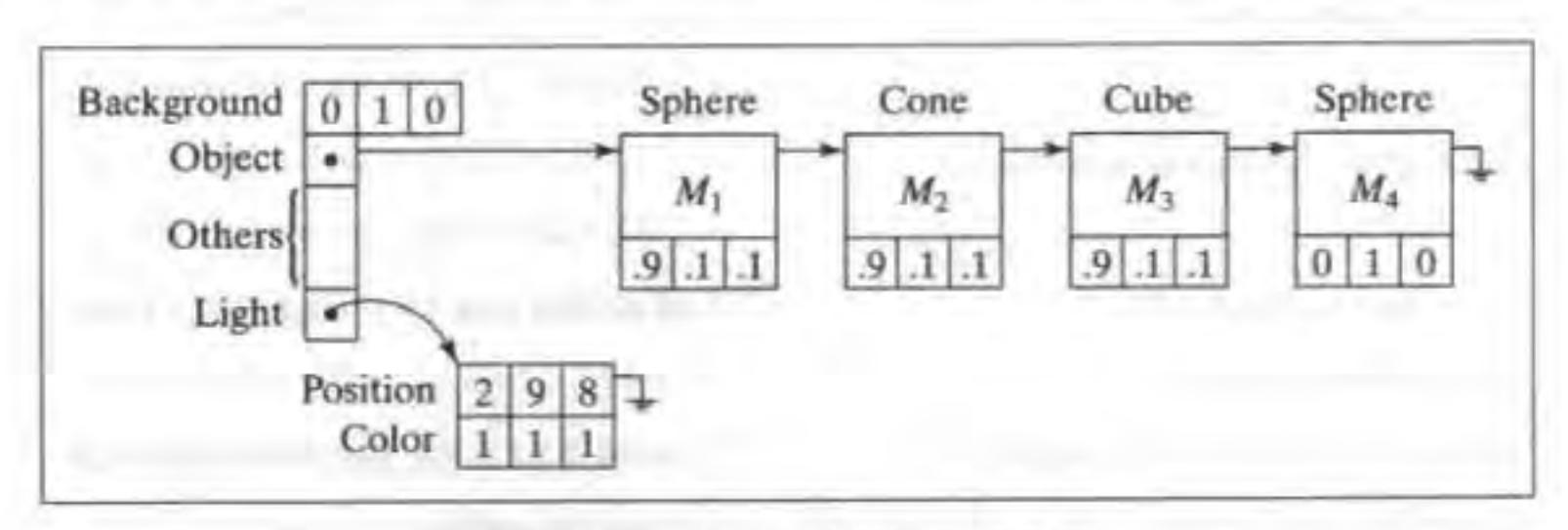


图 5.62 Scene 类的一个对象

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。

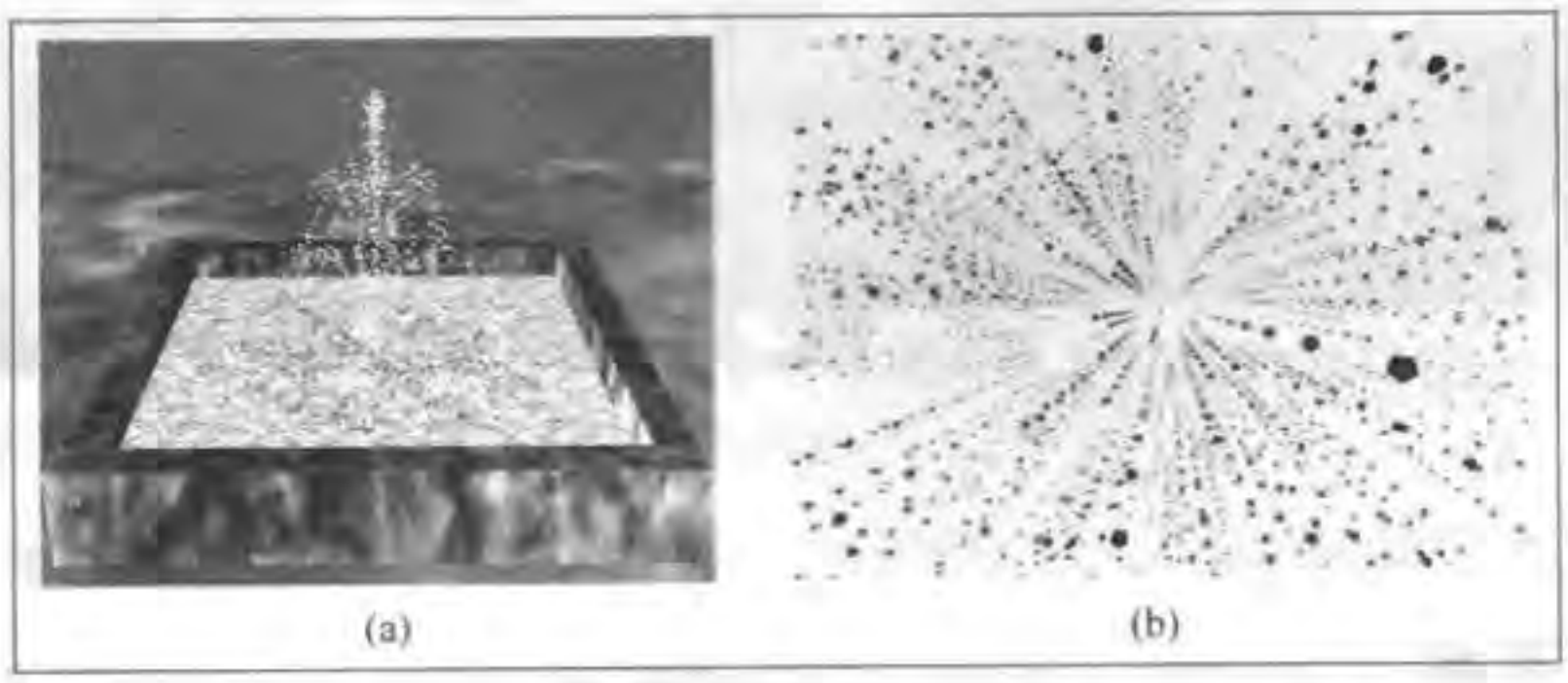


图 6.1

(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。

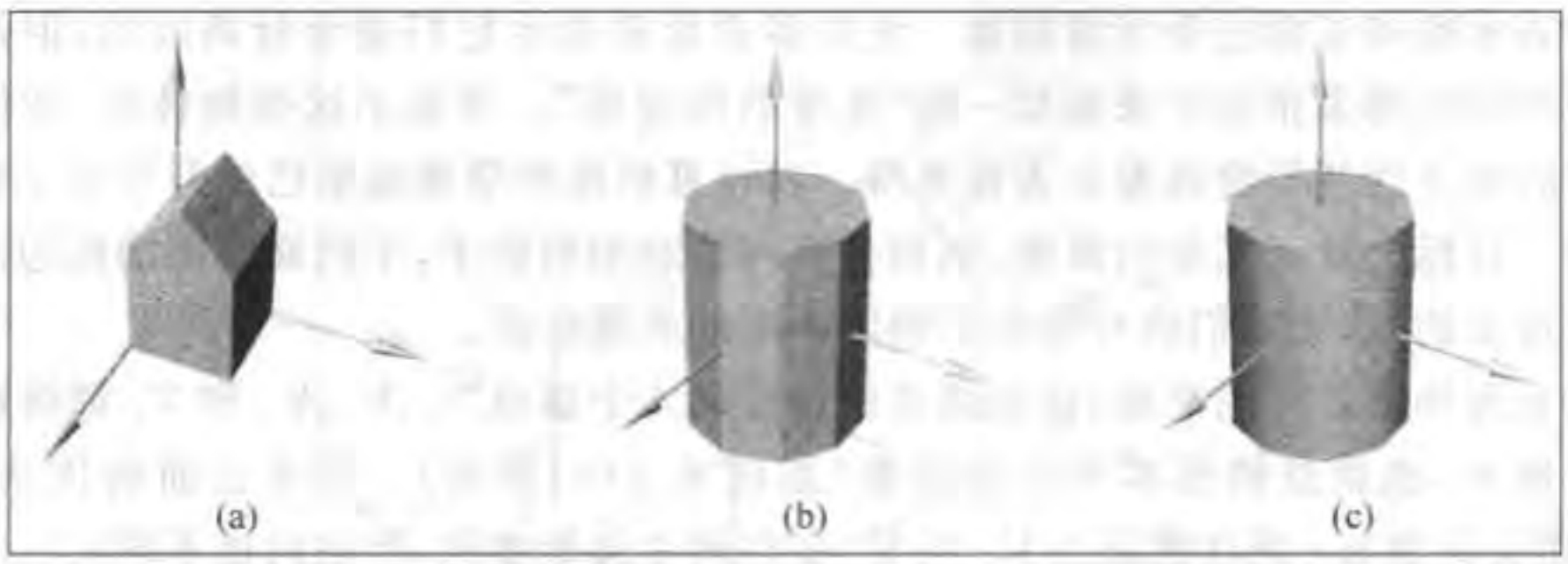


图 6.2　用 Mesh 表示的各种形体

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。

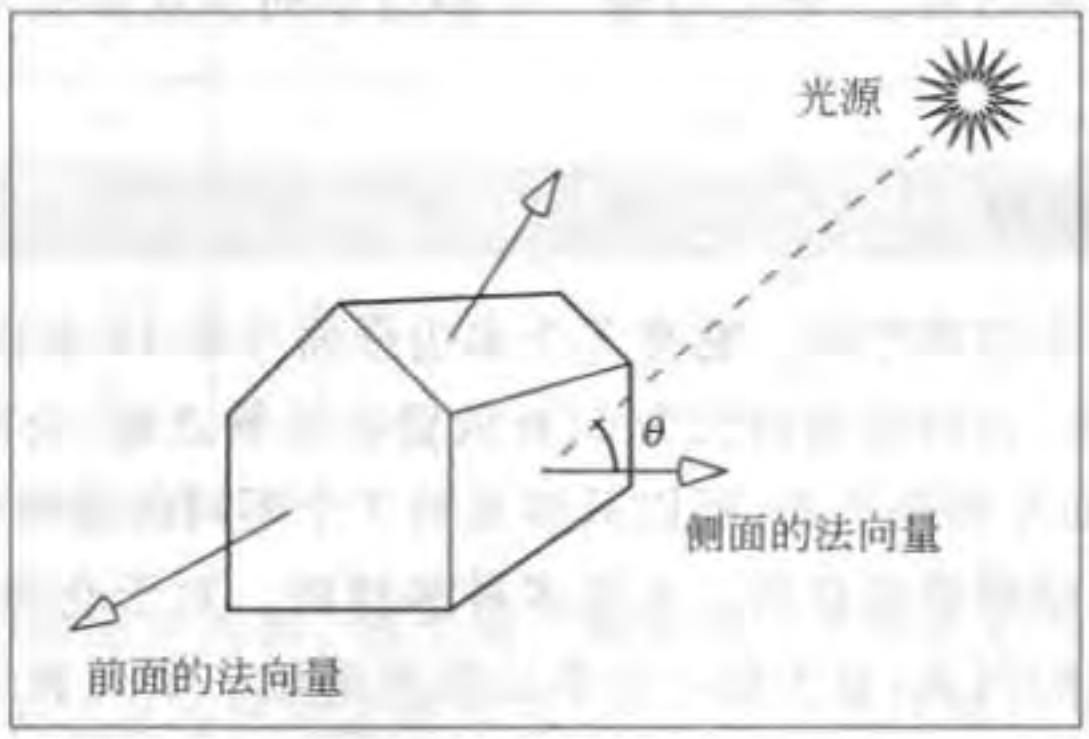


图 6.3　法线方向决定了面的明暗程度

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。

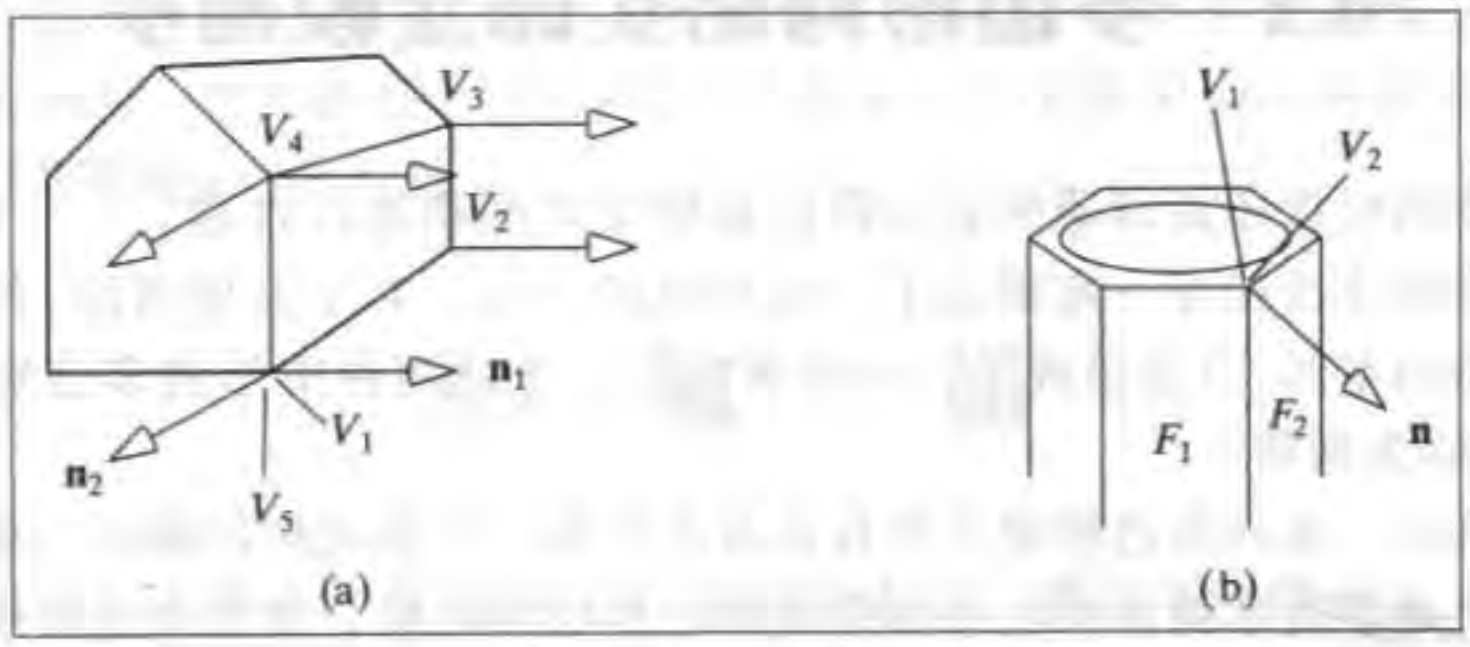


图 6.4 在面的每个顶点指定一个法线

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。

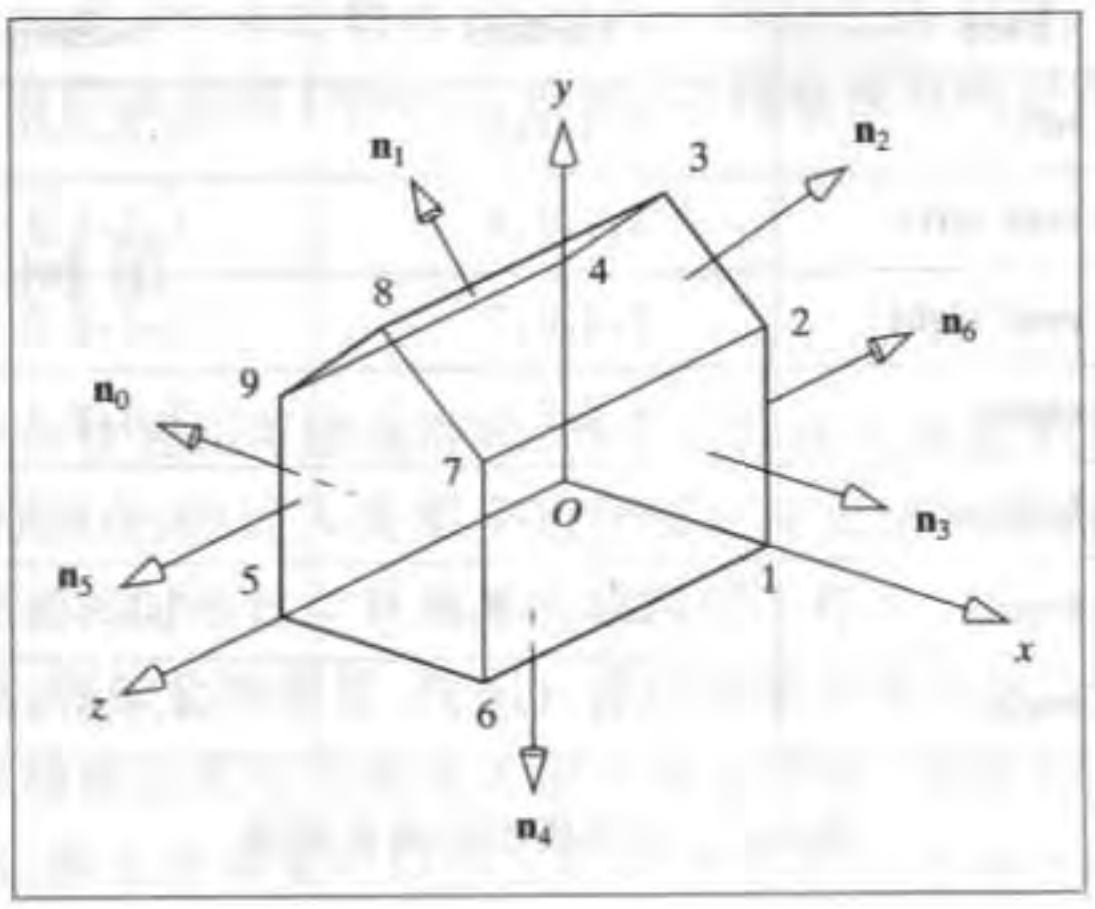


图 6.5　仓房模型

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?

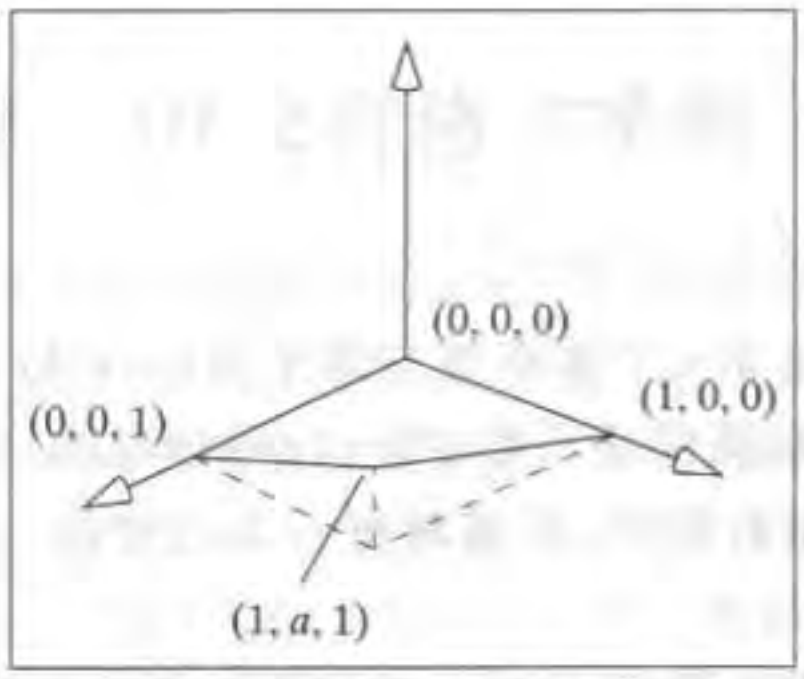


图 6.10 非平面多边形

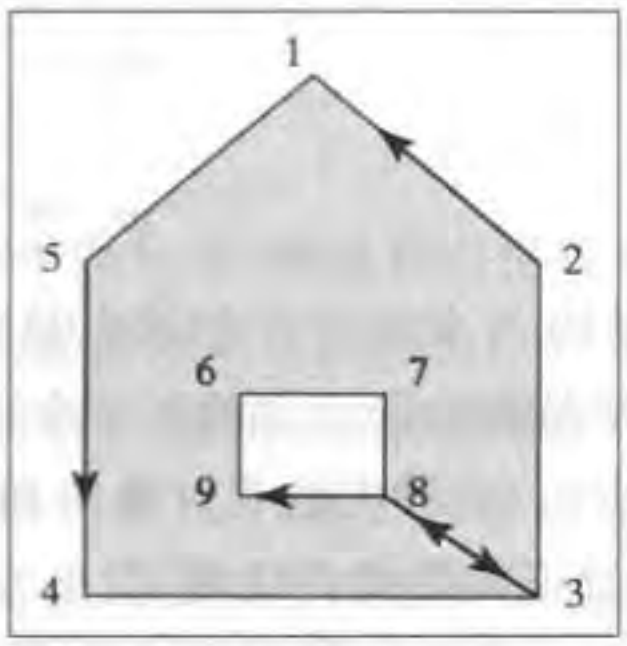


图 6.11 带洞的面

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。

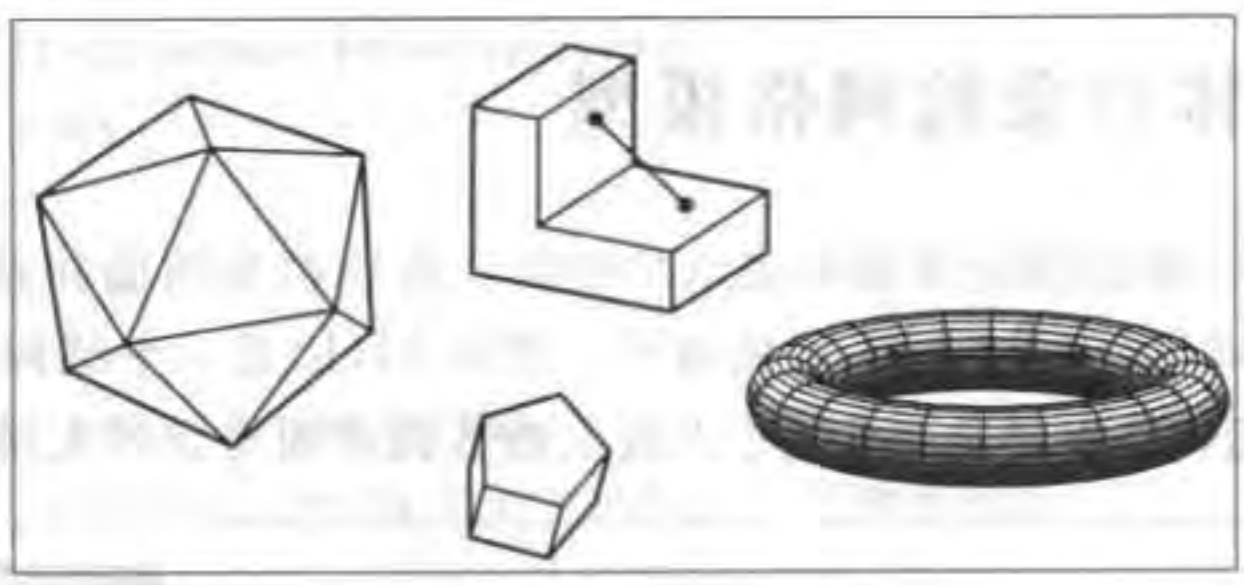


图 6.12　凸的以及非凸的 3D 物体

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。

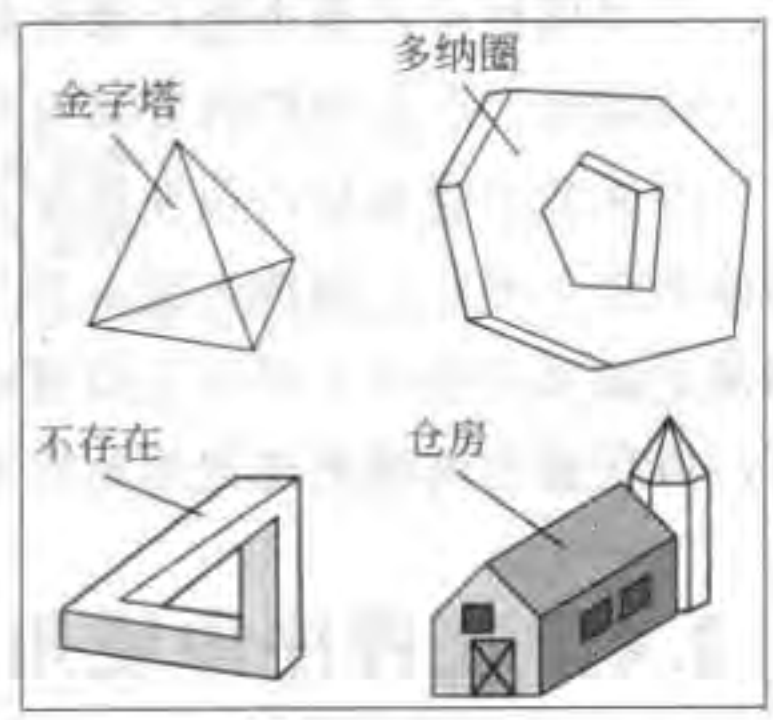


图 6.13　用网格描述的实体

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。

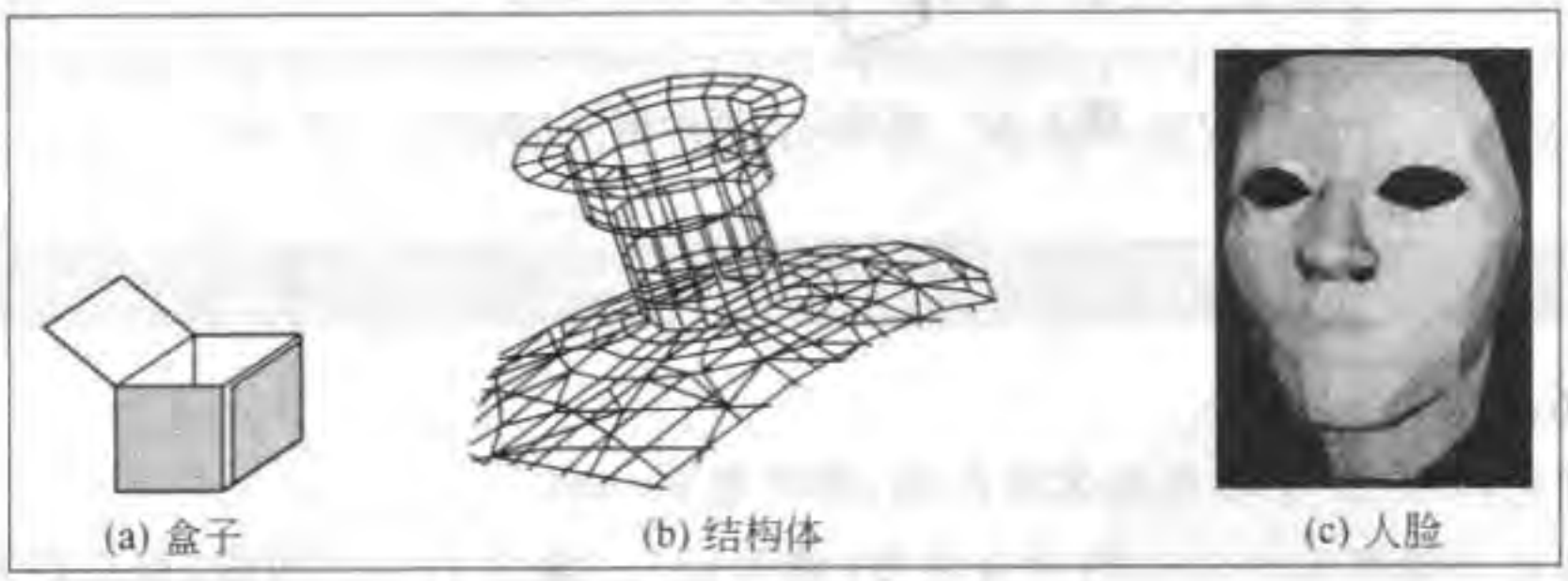


图 6.14 用网格描述的曲面

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)。

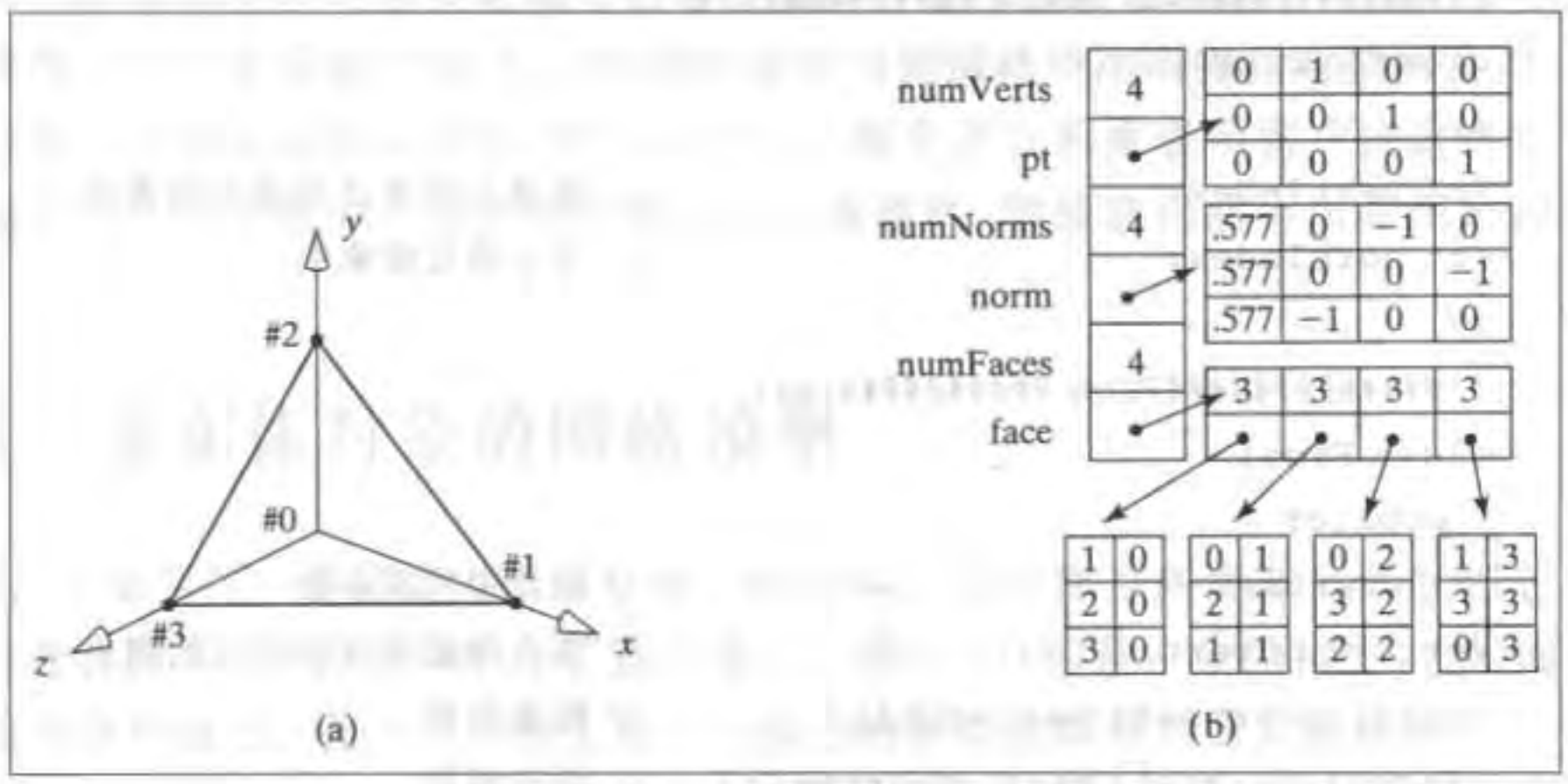


图 6.16 四面体的数据

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。

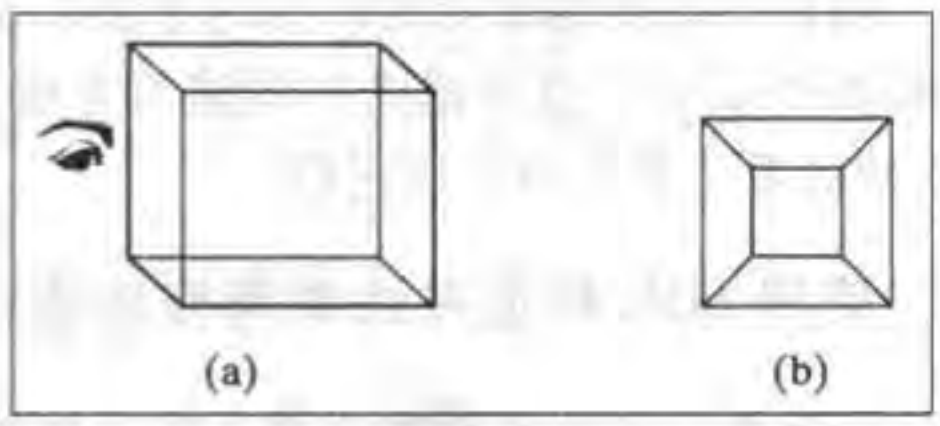


图 6.18 立方体的 Schlegel 图

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?)。

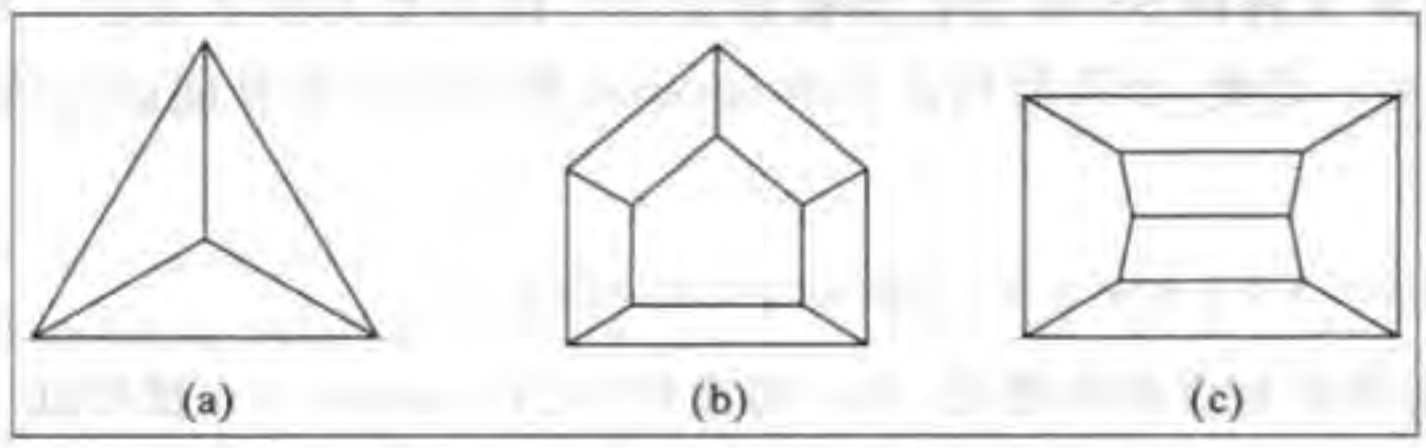


图 6.19 金字塔和仓房的 Schlegel 图

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。

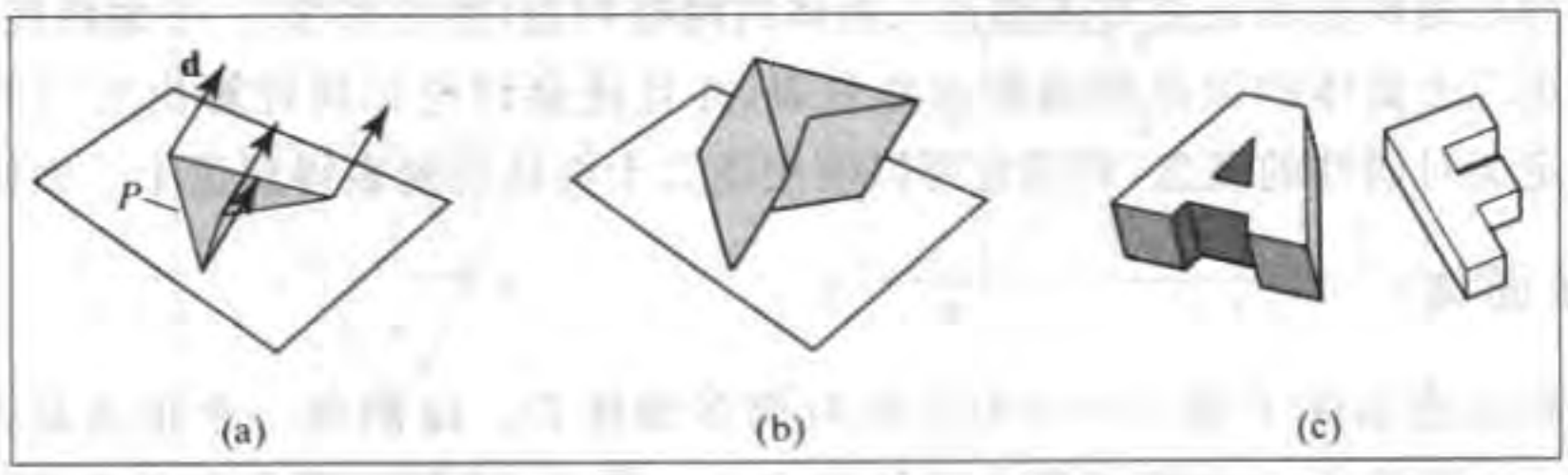


图 6.20　使用拉伸操作得到棱柱

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。

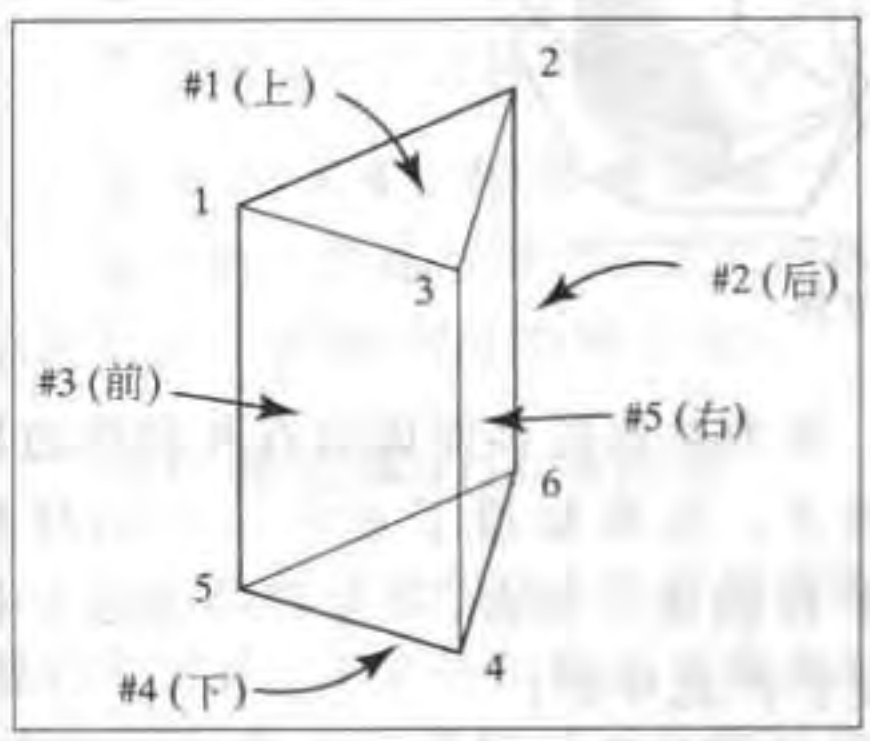


图 6.21　多面体物体范例

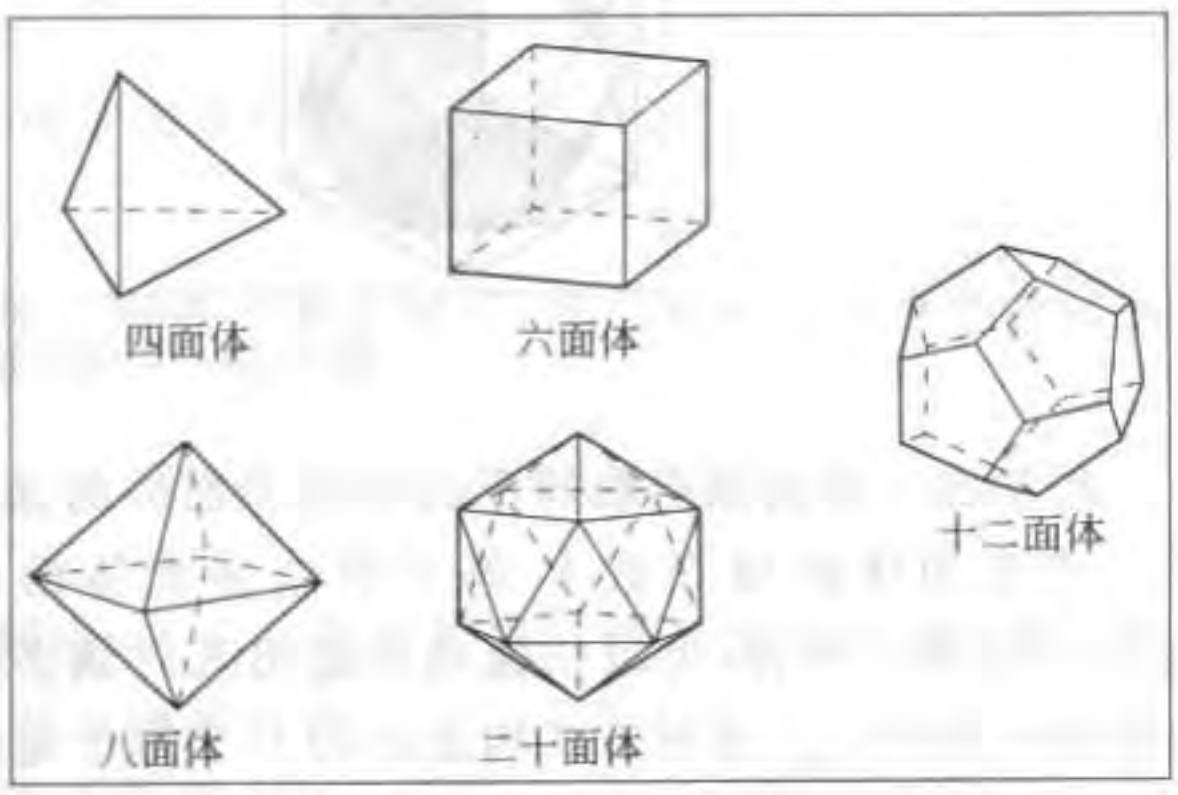


图 6.22　五种柏拉图立体

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。

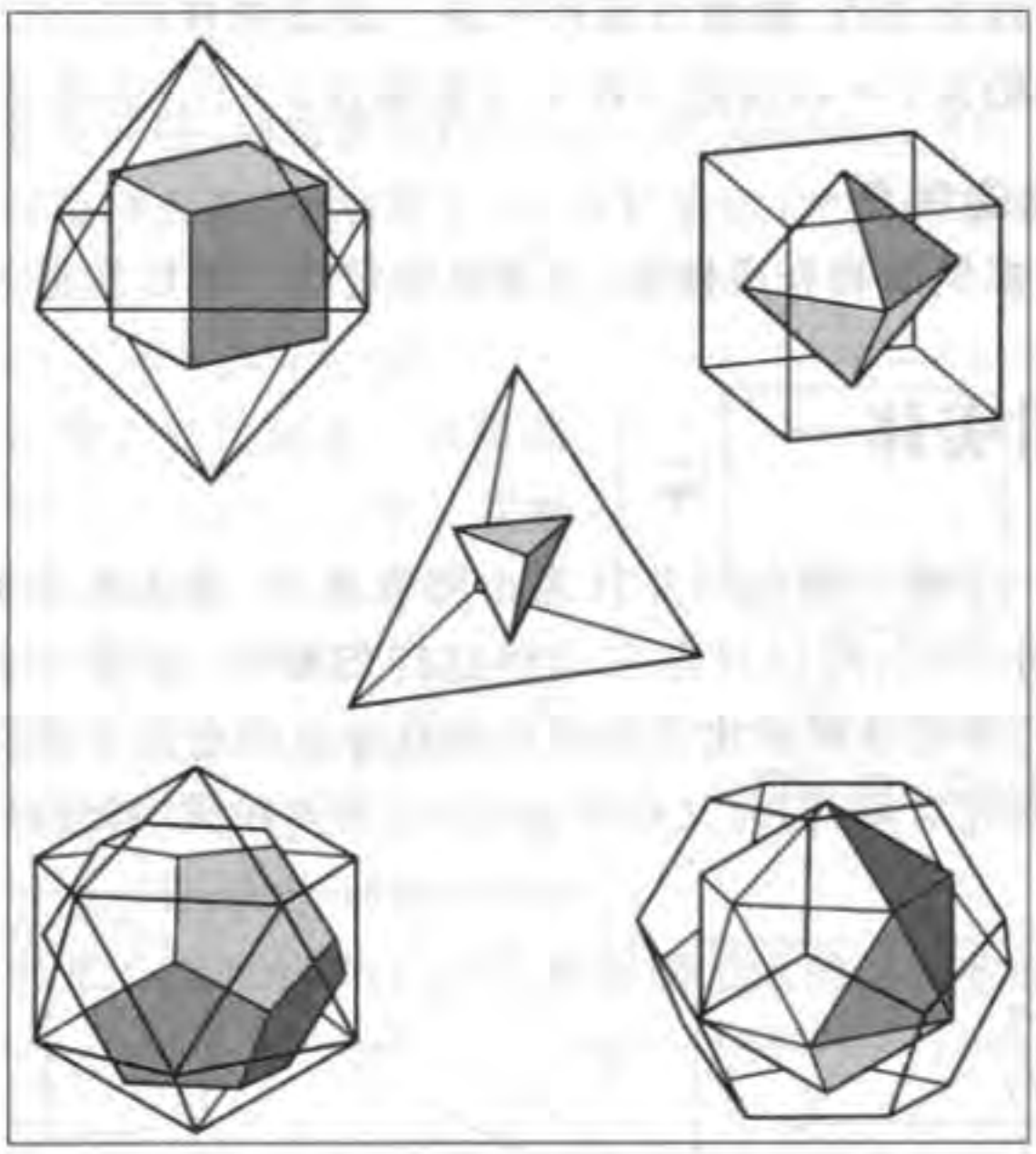


图 6.23 对偶柏拉图立体

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。

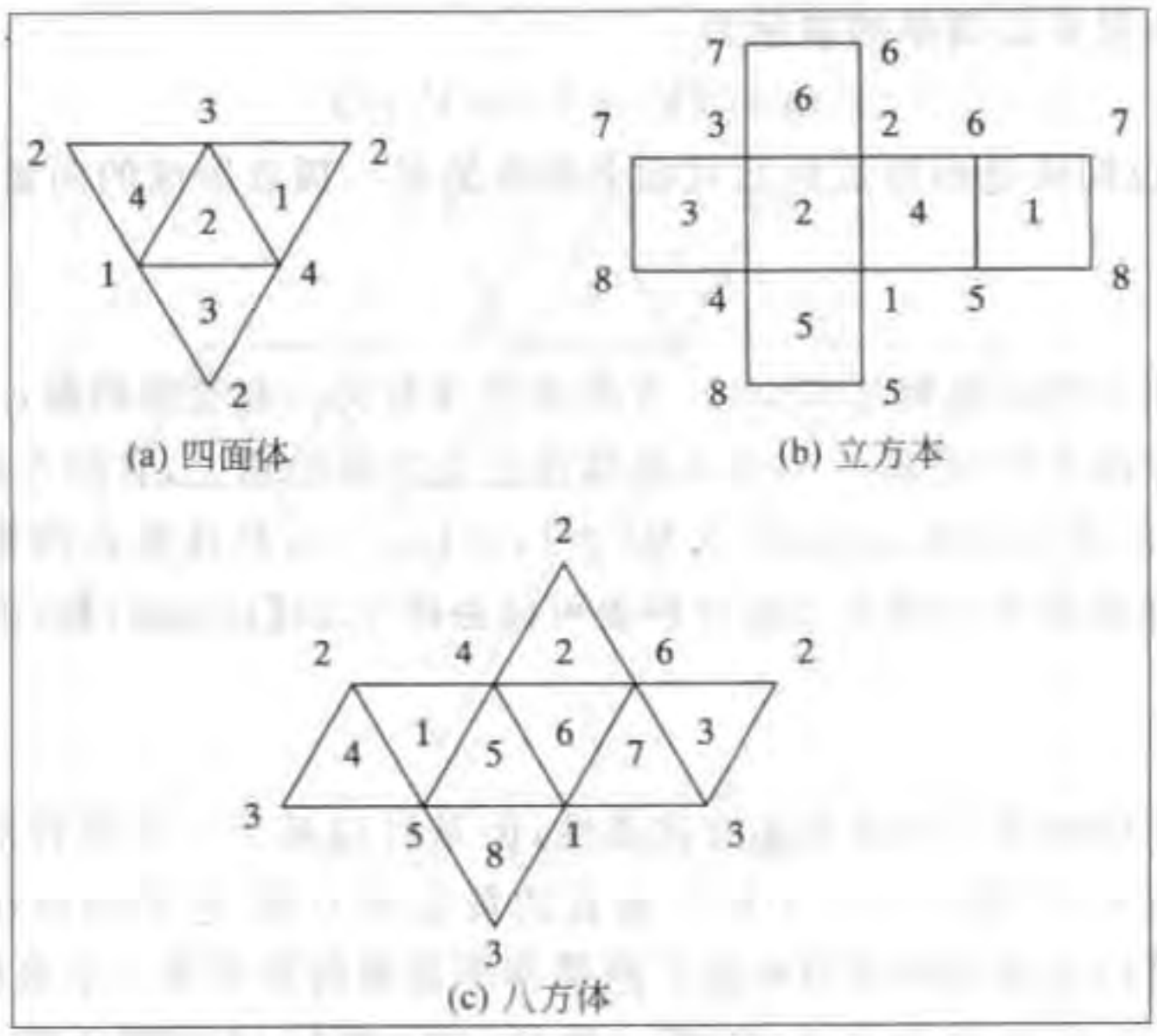


图 6.24　四面体、立方体和八面体的模型

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

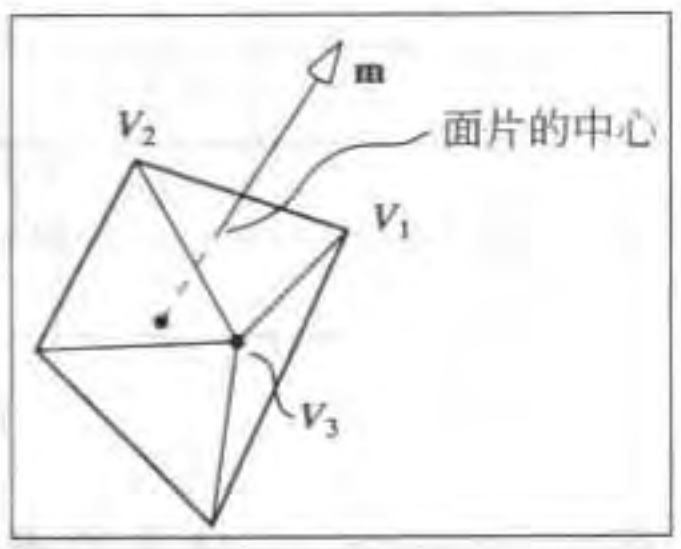


图 6.25　利用对称性计算面片的法线

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

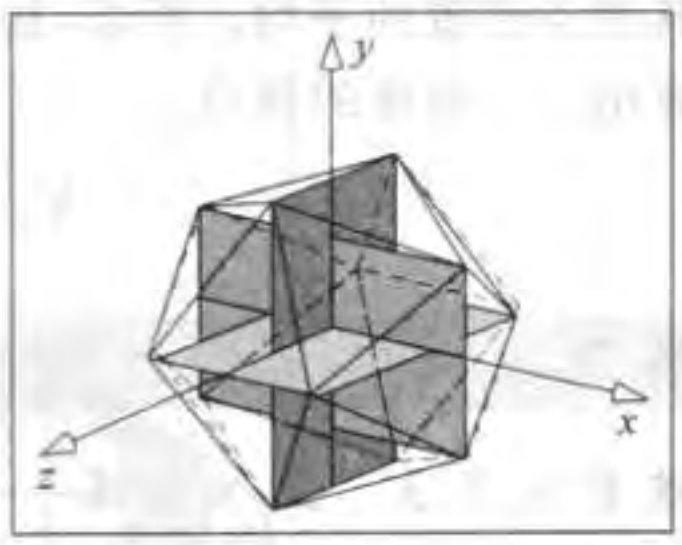


图 6.27　用黄金长方形定义正二十面体

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?)。

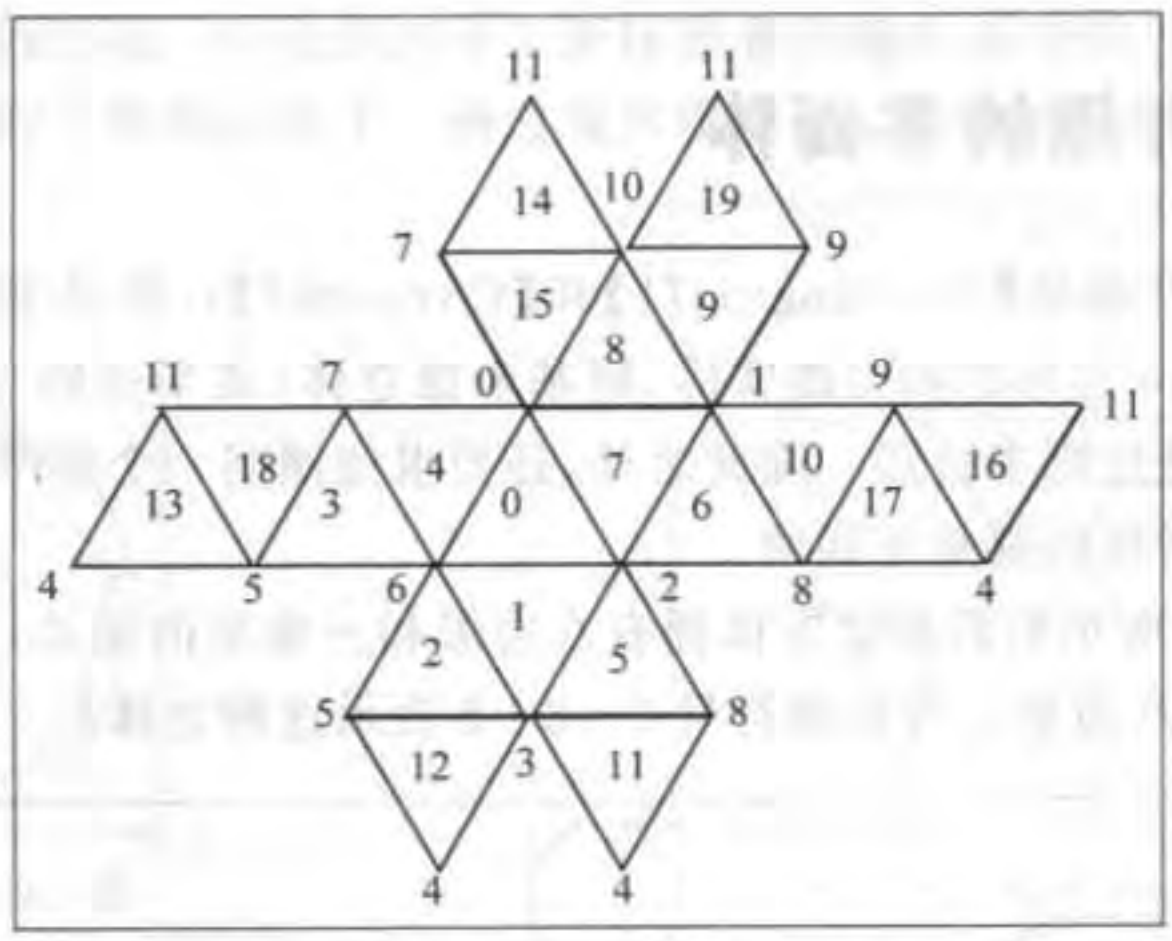


图 6.29　正二十面体的模型

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。

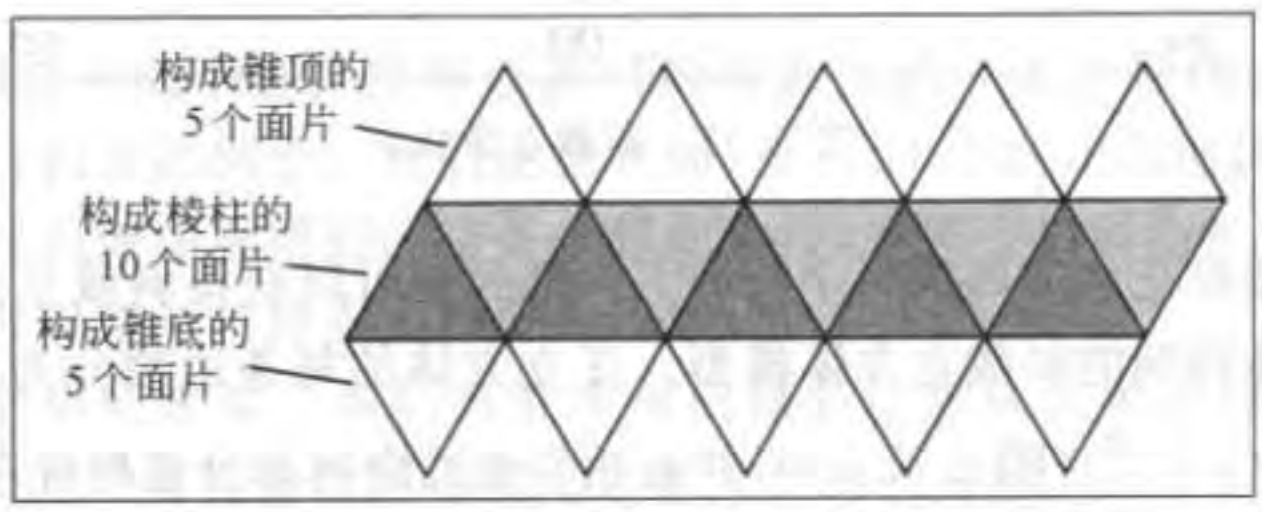


图 6.30　正二十面体是带有上下盖的棱柱

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。

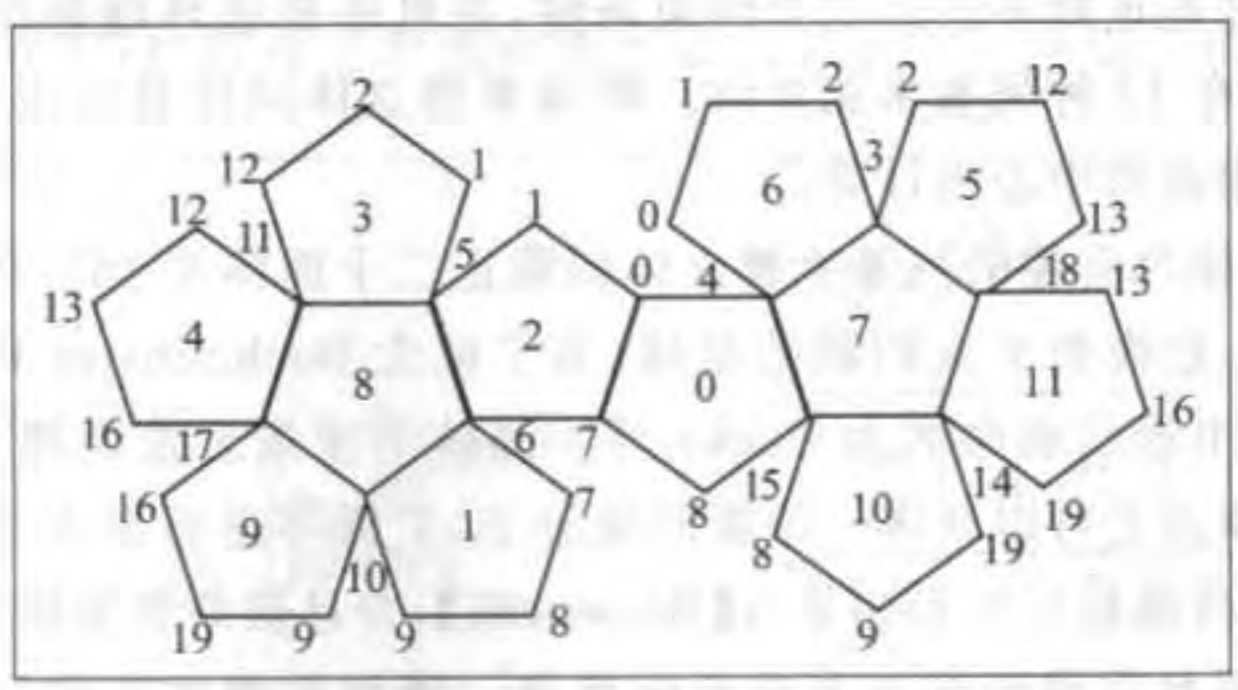


图 6.31　正十二面体的模型

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。

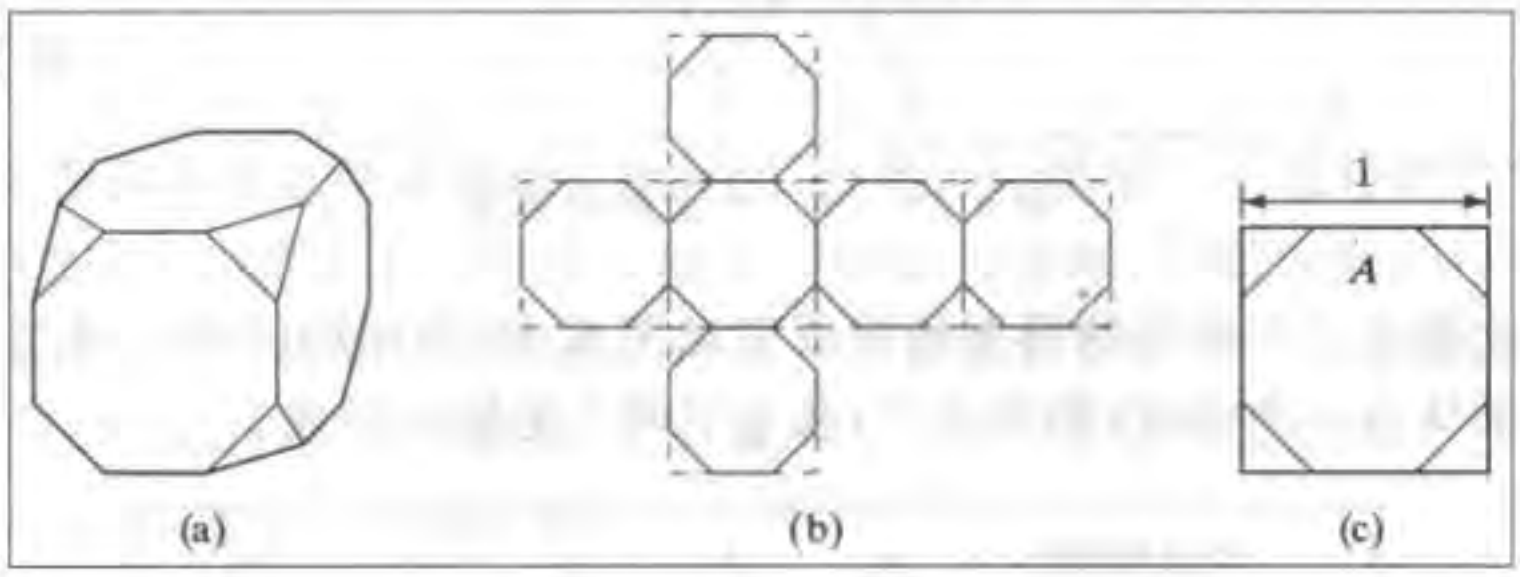


图 6.32 斜截立方体

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。

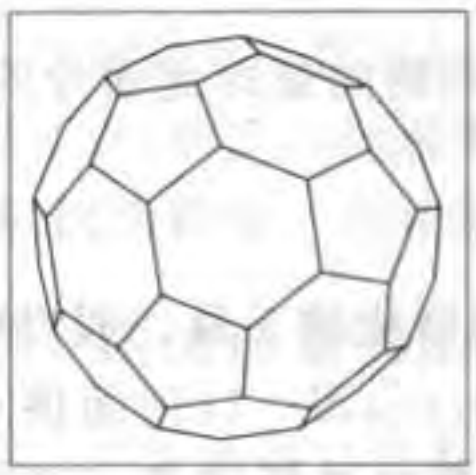


图 6.33 Bucky 球

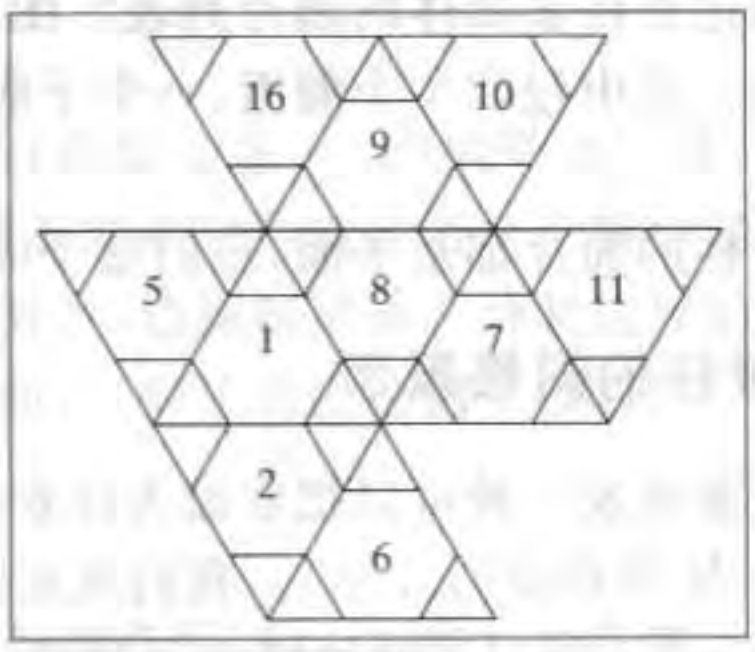


图 6.34 构造 Bucky 球

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

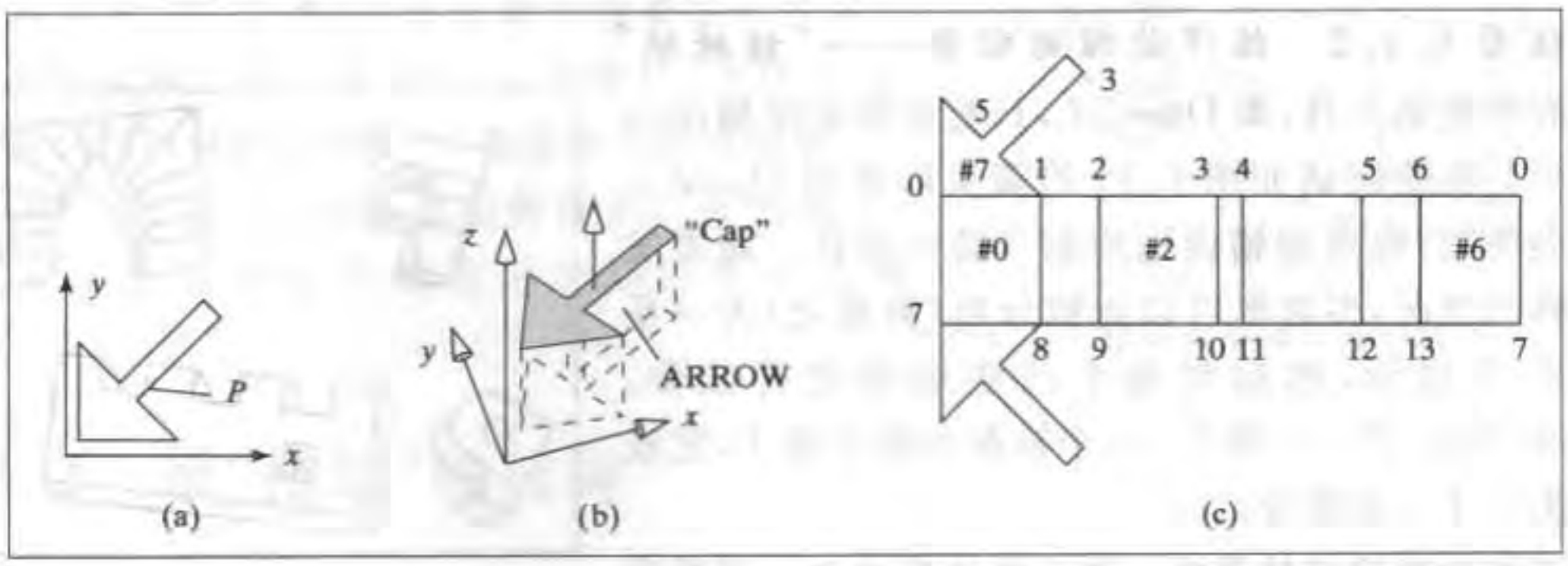


图 6.35 棱柱范例

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。

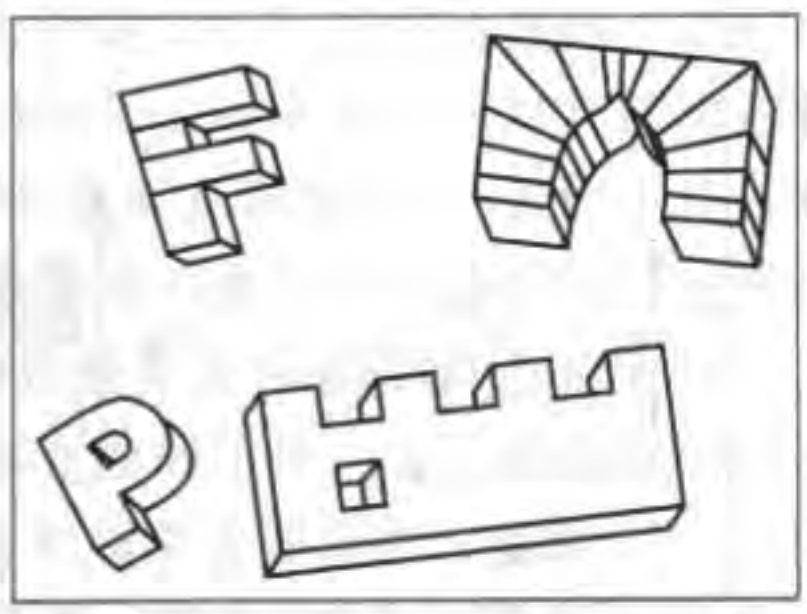


图 6.36 基于凸棱柱集合的拉伸体

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：

$$\text{四边形条带} = \{p_0, p_1, p_2, \cdots, p_{M-1}\} \tag{6.7}$$

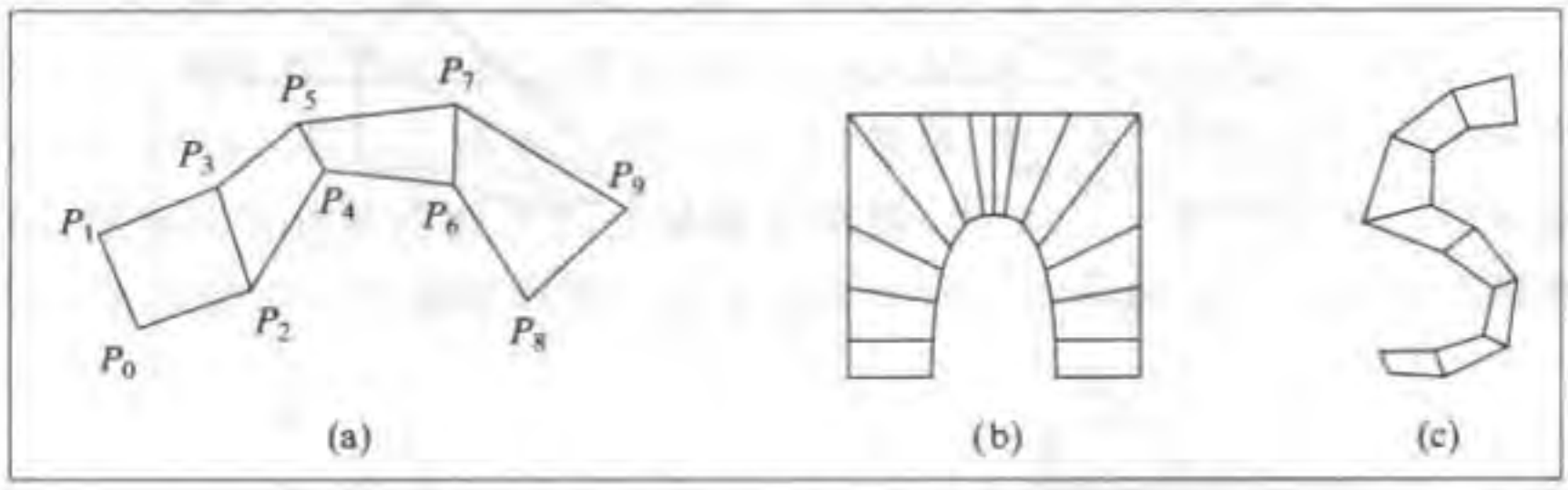


图 6.37　四边形条带与用四边形条带构造的棱柱

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。

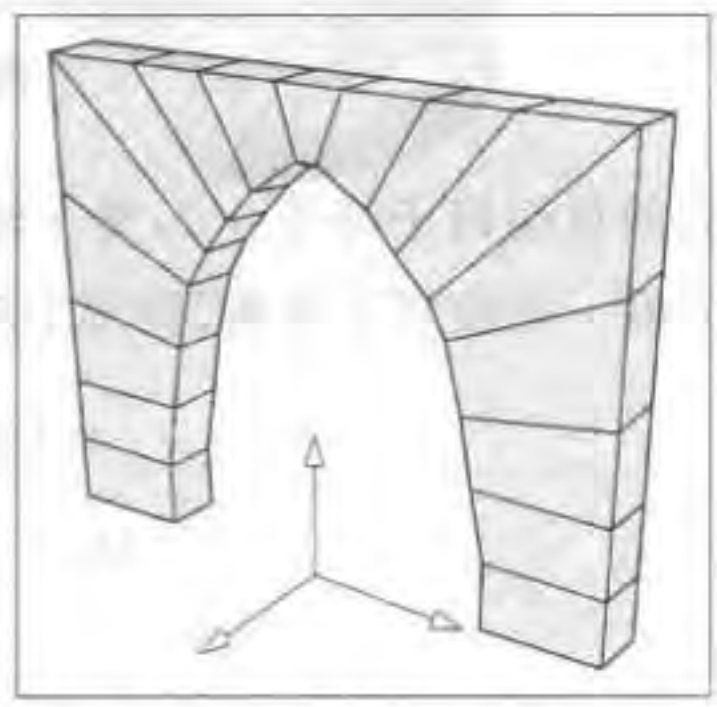


图 6.38　把四边形条带进行拉伸——拱门

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

$$M = \begin{bmatrix} 0.7 & 0 & 0 & 0 \\ 0 & 0.7 & 0 & 0 \\ 0 & 0 & 1 & H \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

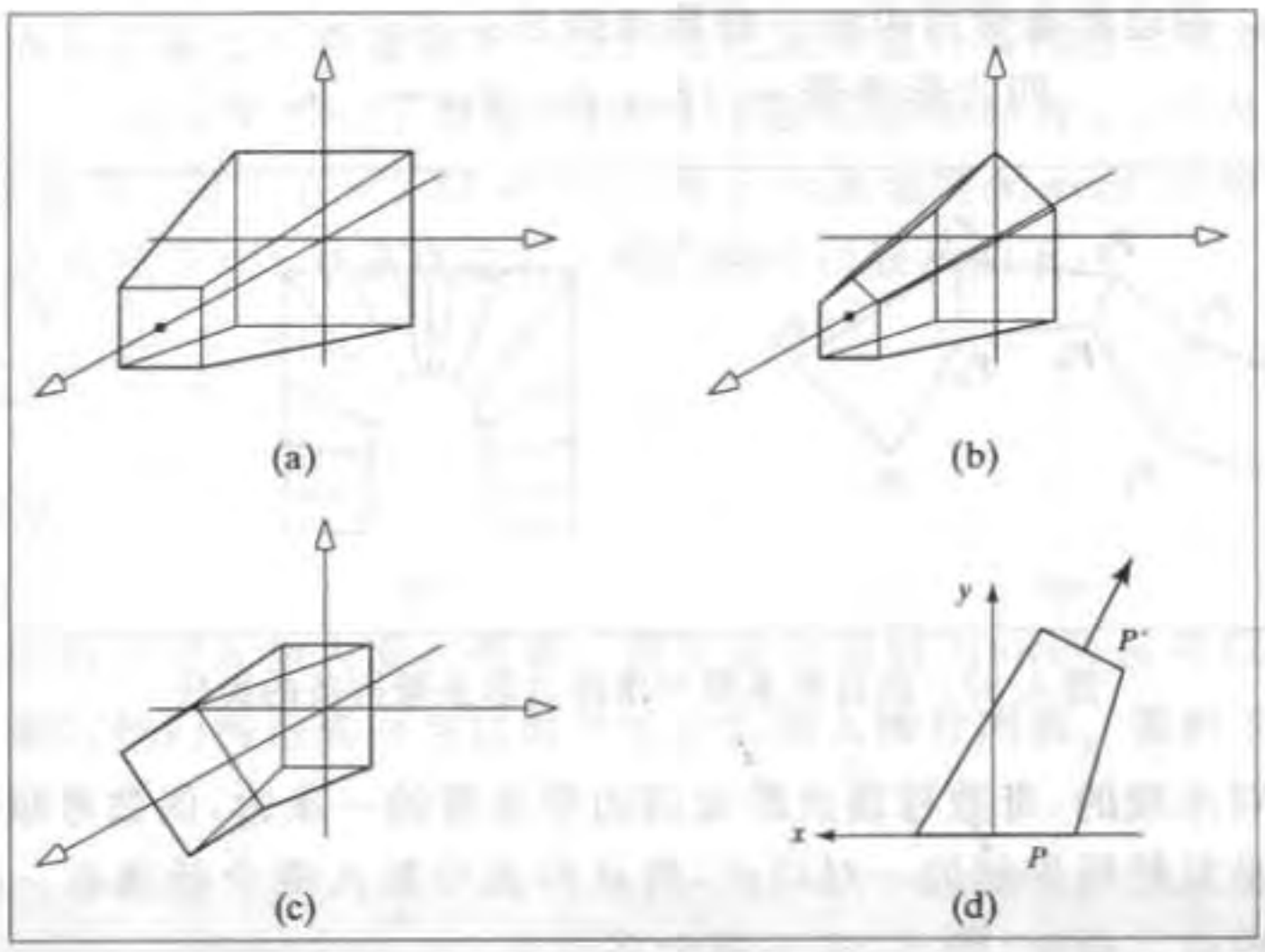


图 6.39 锥体和扭转的棱柱

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。

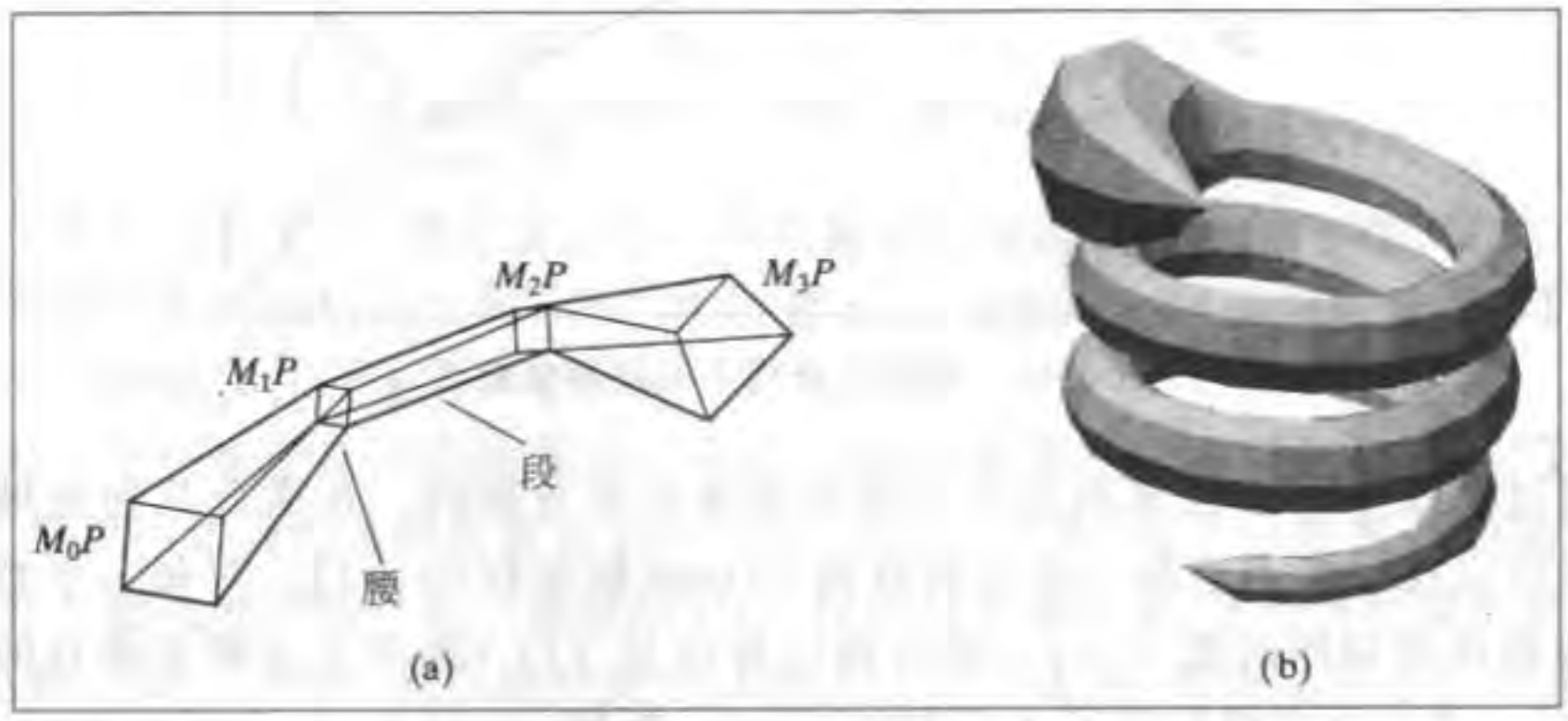


图　6.40

(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。

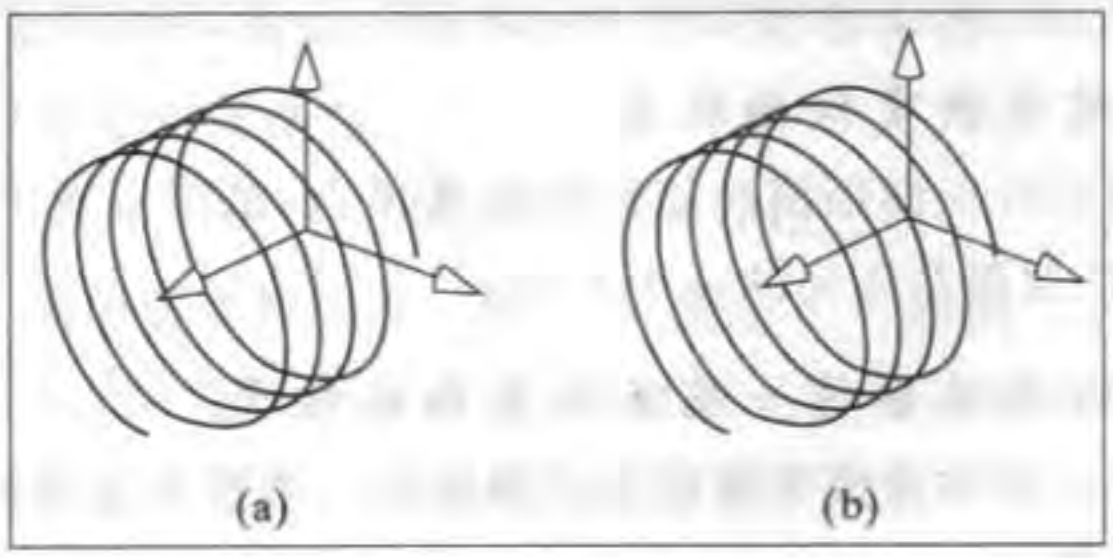


图 6.41 一对螺旋线

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。

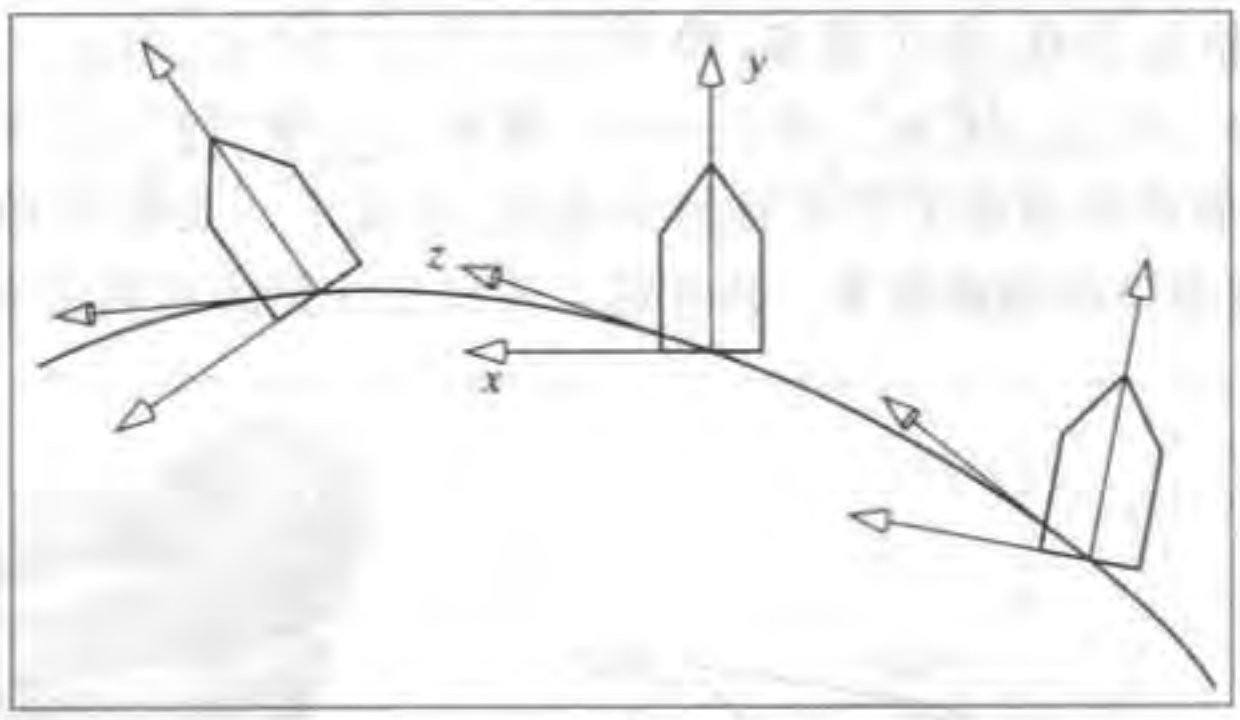


图 6.42 构建在脊线上的局部坐标系

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。

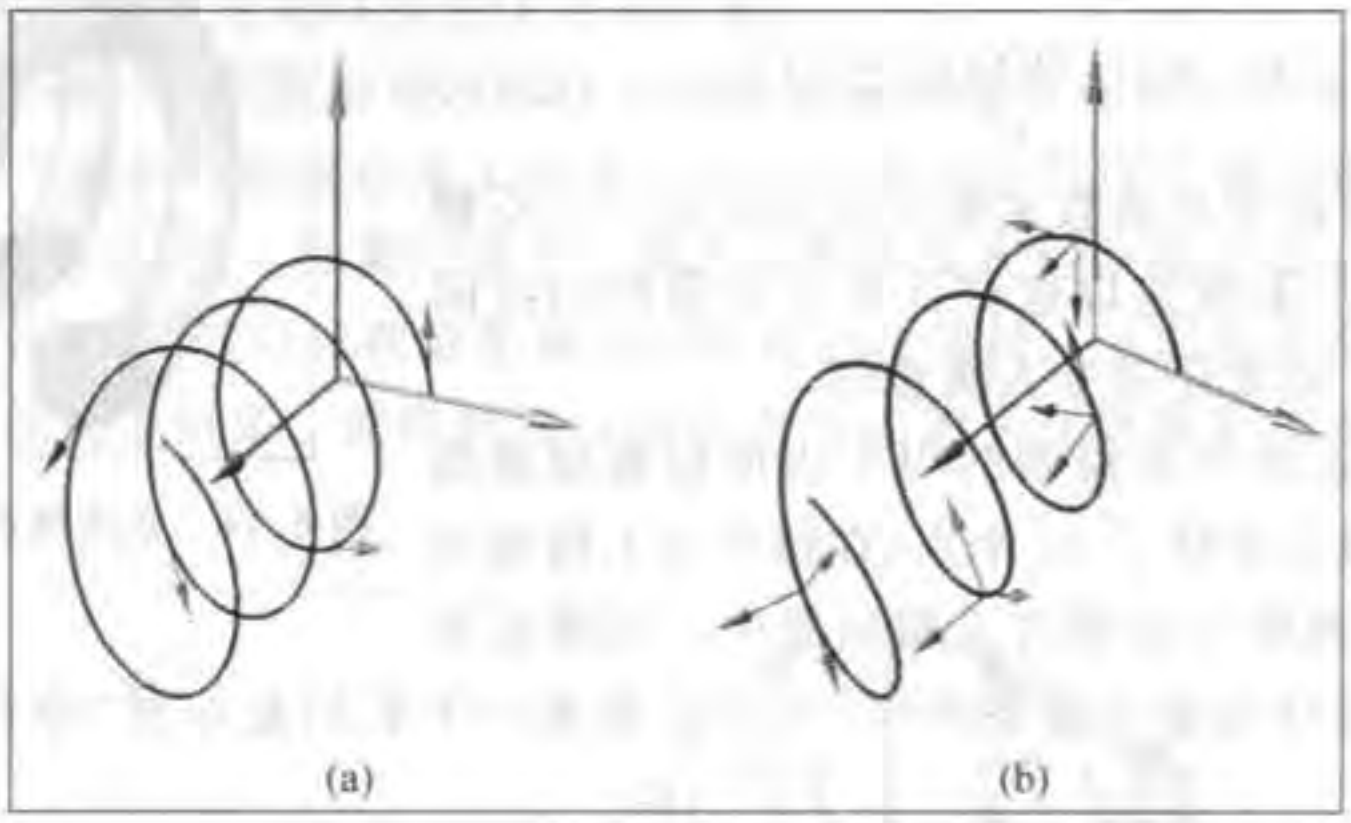


图　6.43

(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

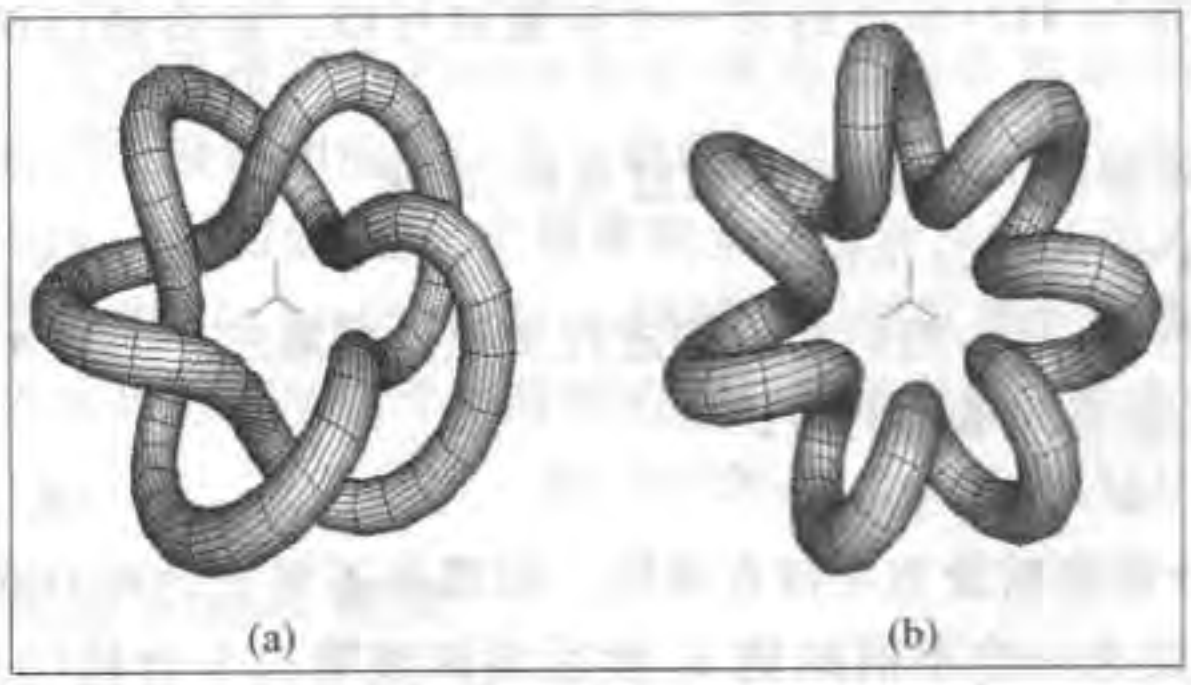


图 6.45 基于圆螺旋脊线的管状体

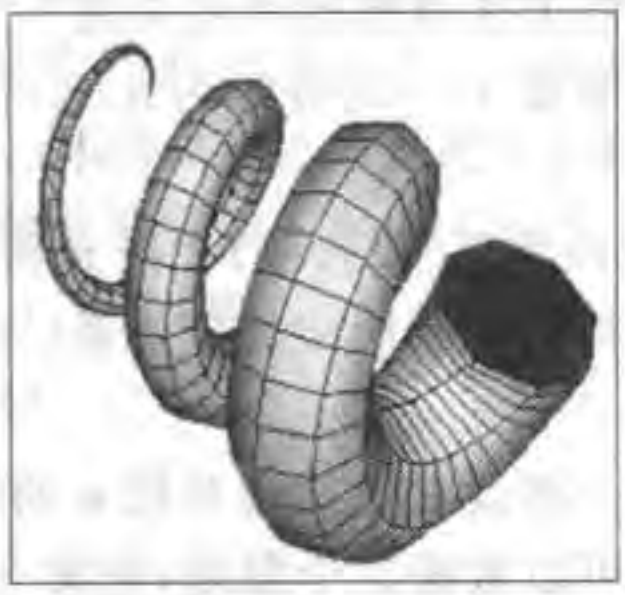


图 6.46 海螺形状

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。

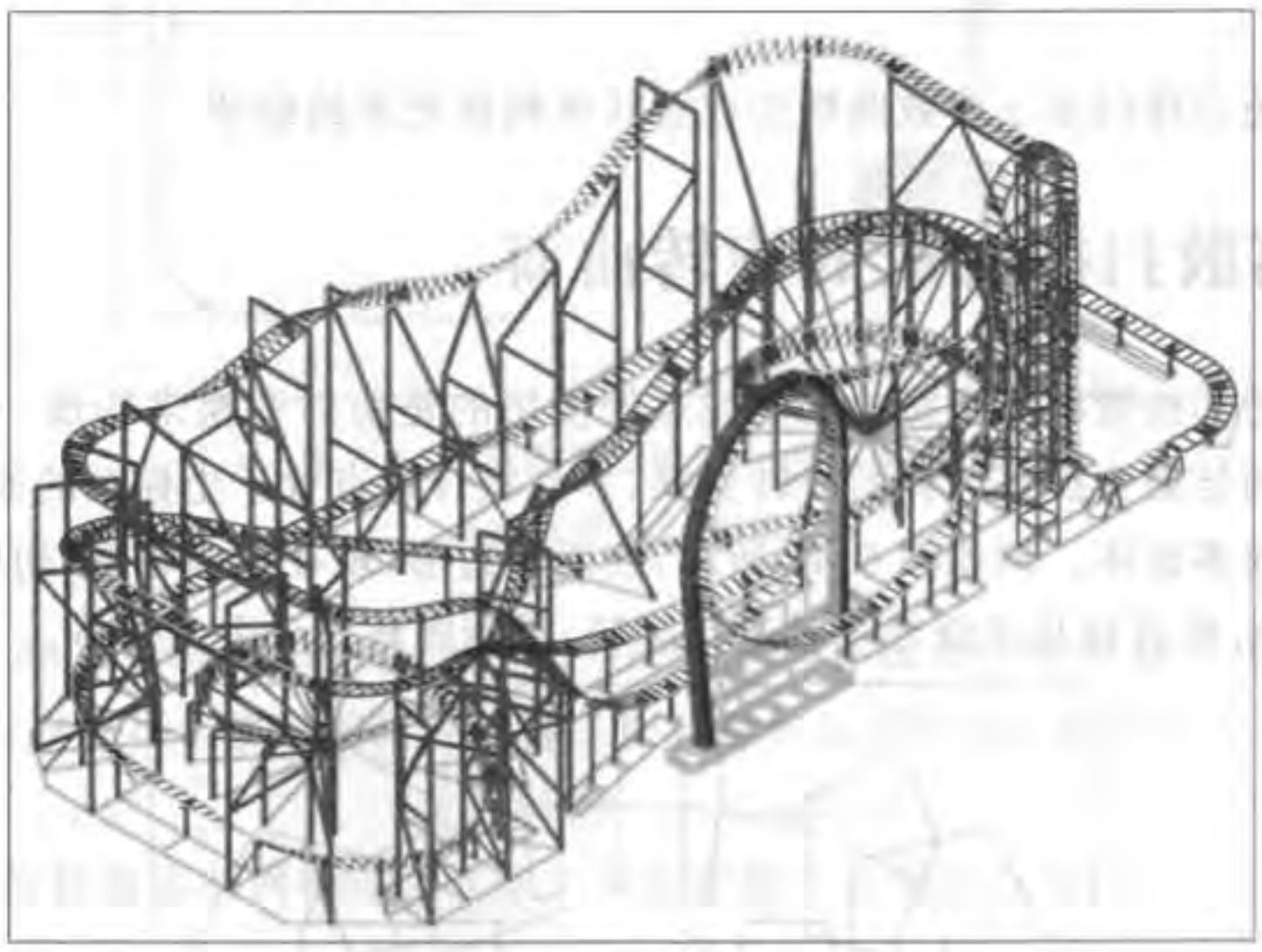


图 6.47　过山车模型

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 $M=2, N=3, \phi=0$。

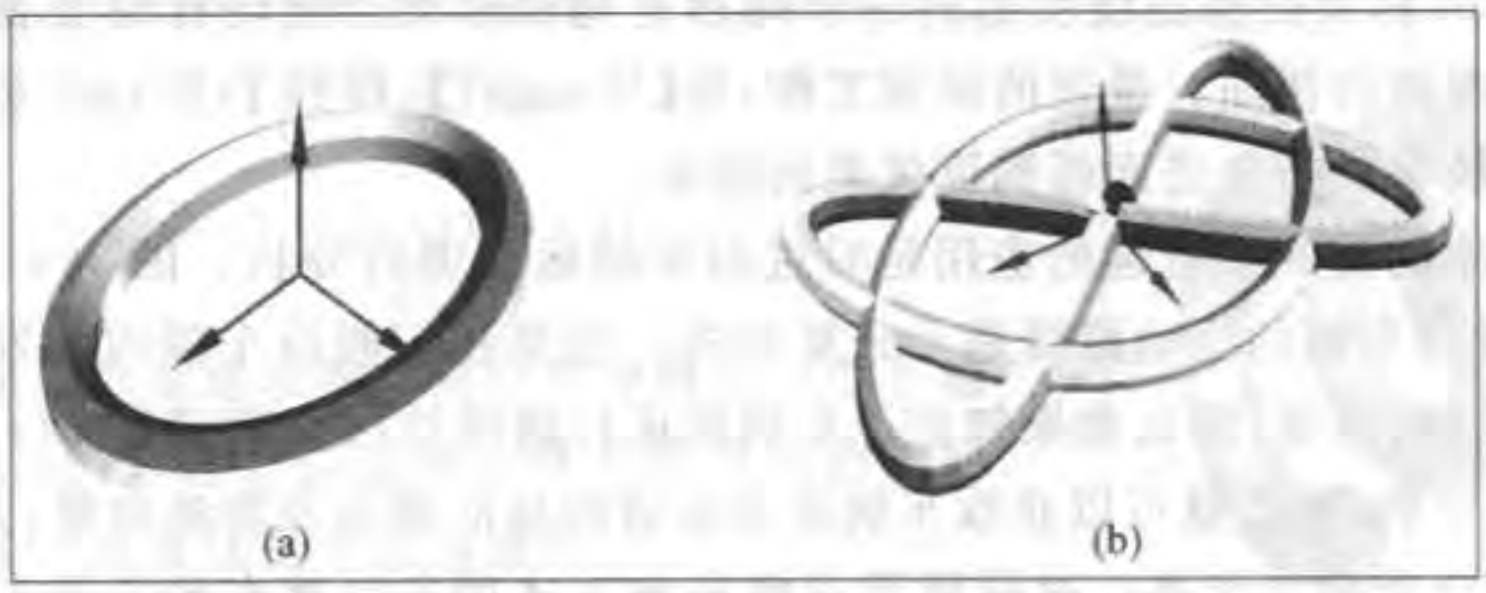


图 6.48

(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。

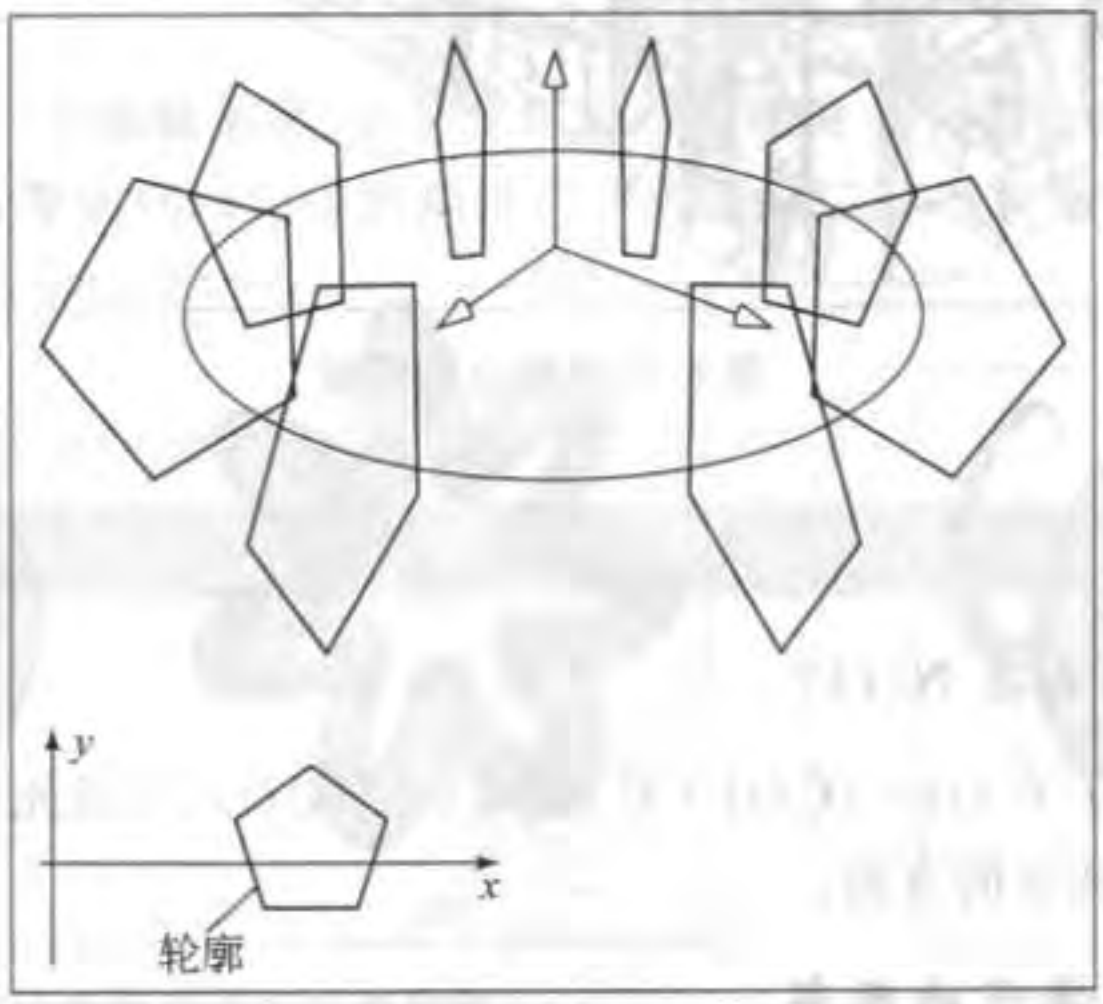


图 6.49 分步骤的旋转扫掠

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

$$\widetilde{M}_i = \begin{bmatrix} \cos(\theta_i) & 0 & \sin(\theta_i) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta_i) & 0 & \cos(\theta_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

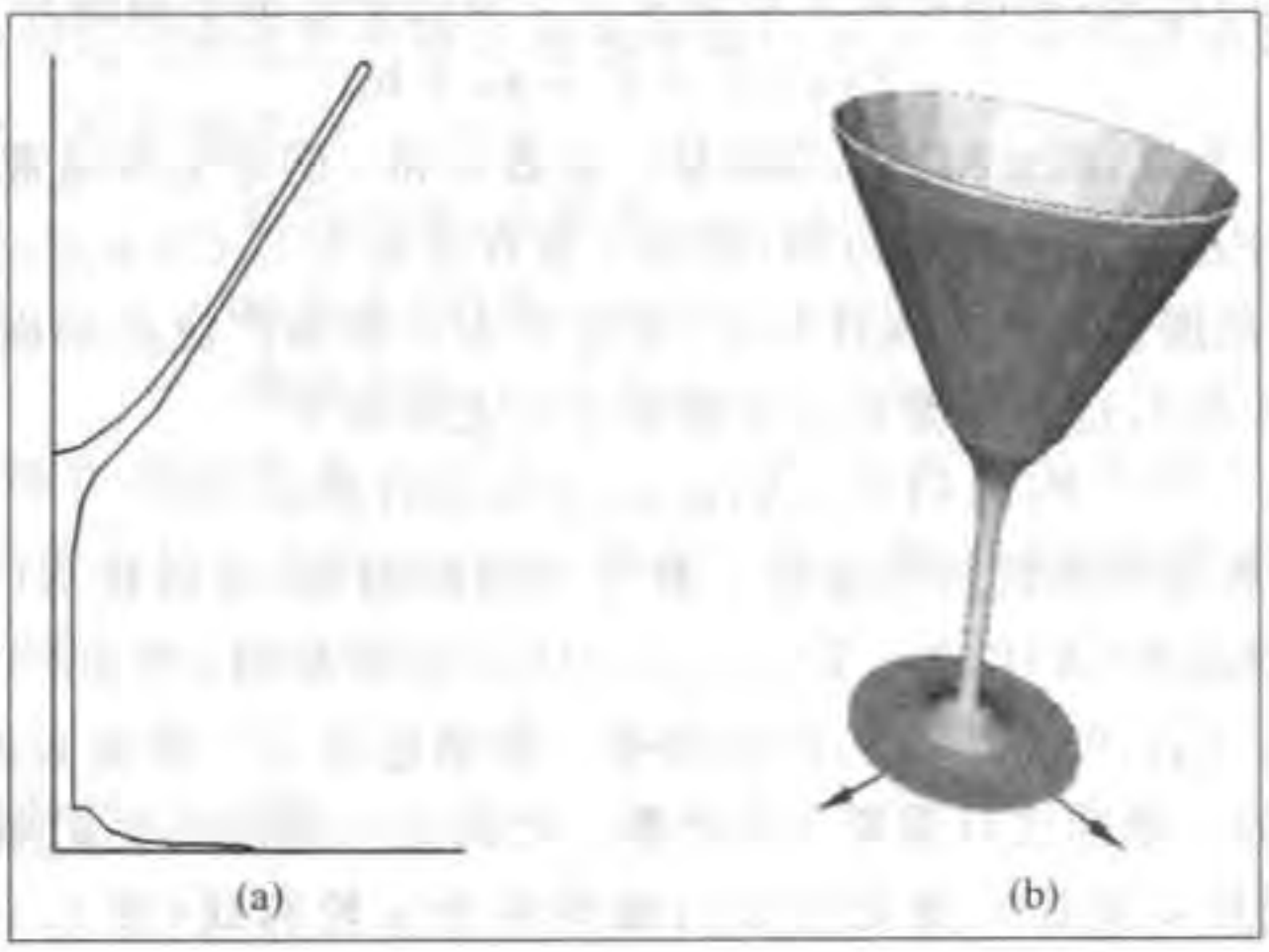


图 6.50　离散扫掠近似生成马蒂尼酒杯

(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。

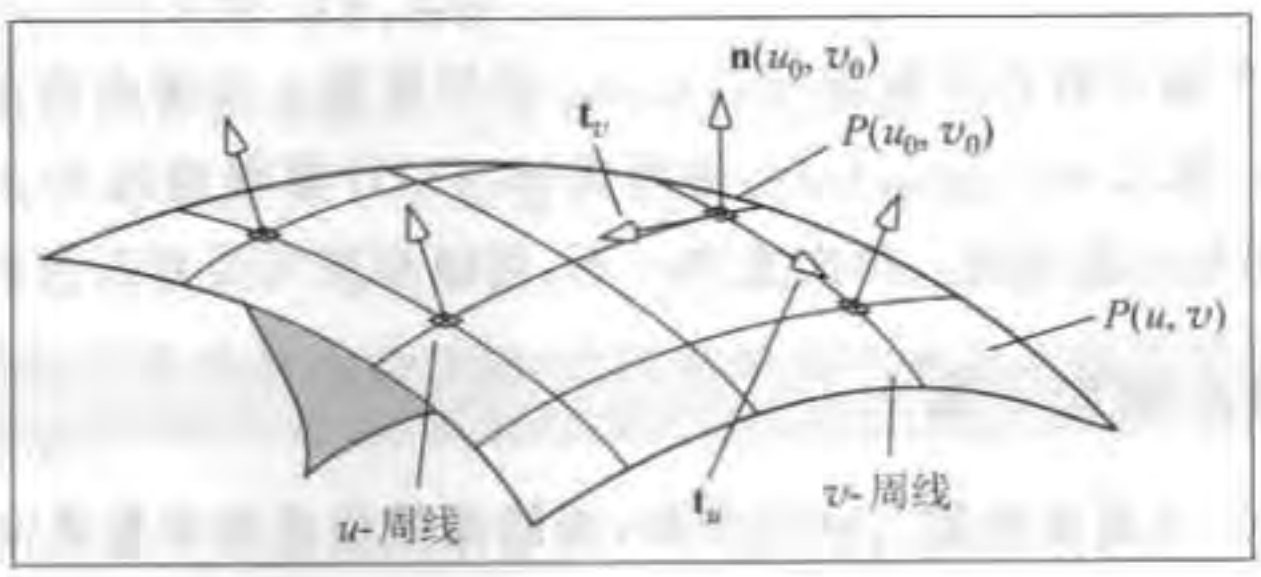


图 6.51　曲面的法向量

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:

$$F(x,y,z)=x^2+y^2+z^2-1 \tag{6.26}$$

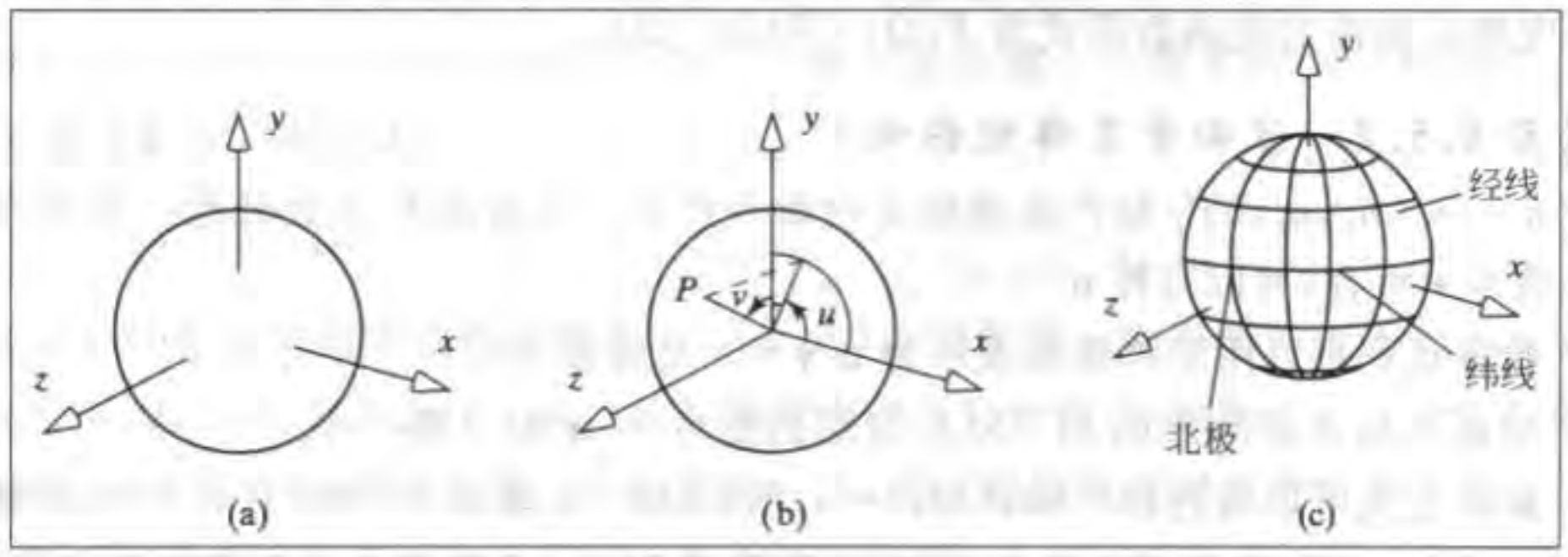


图 6.52

(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。

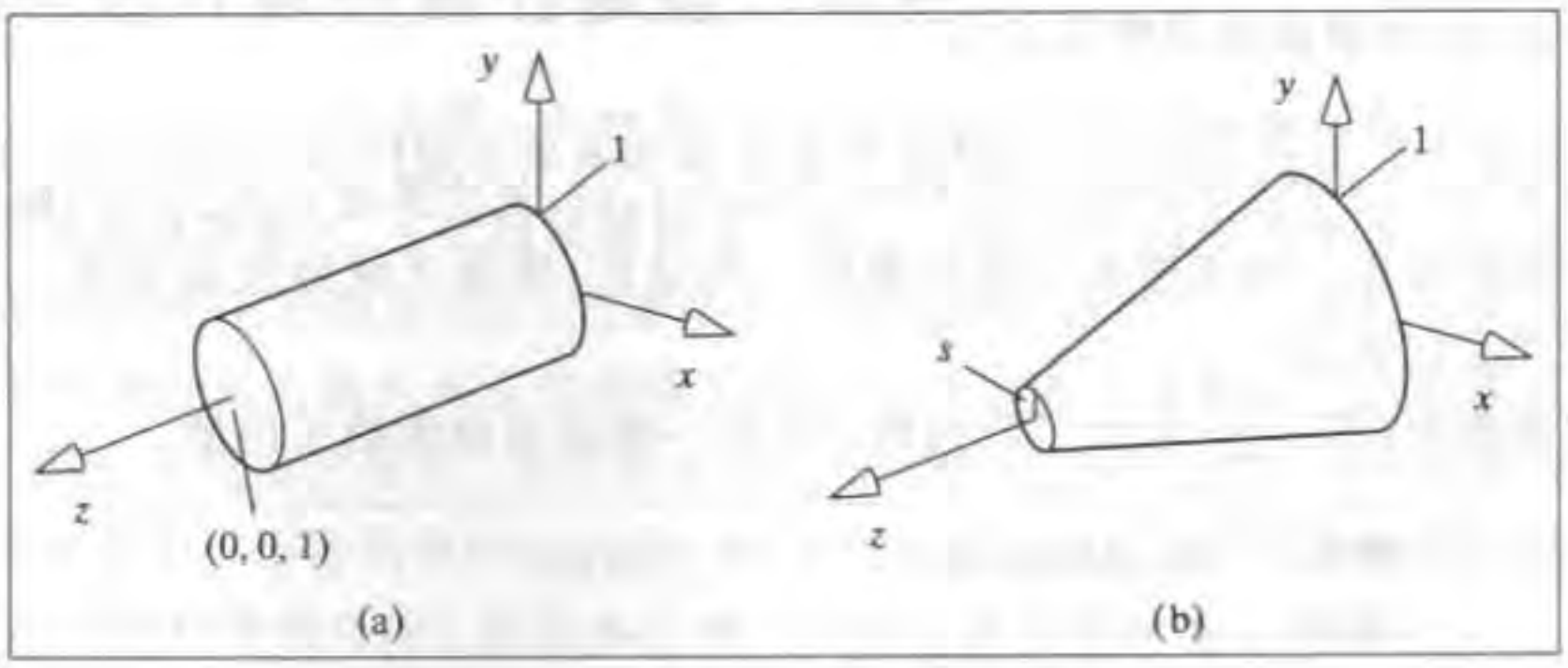


图 6.53　通用柱面和一端逐渐变小的柱面

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?)。

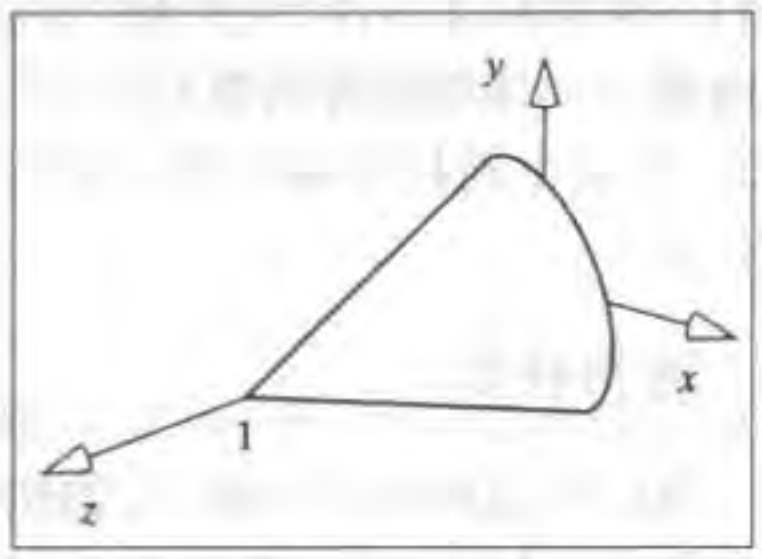


图 6.54 通用锥面

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。

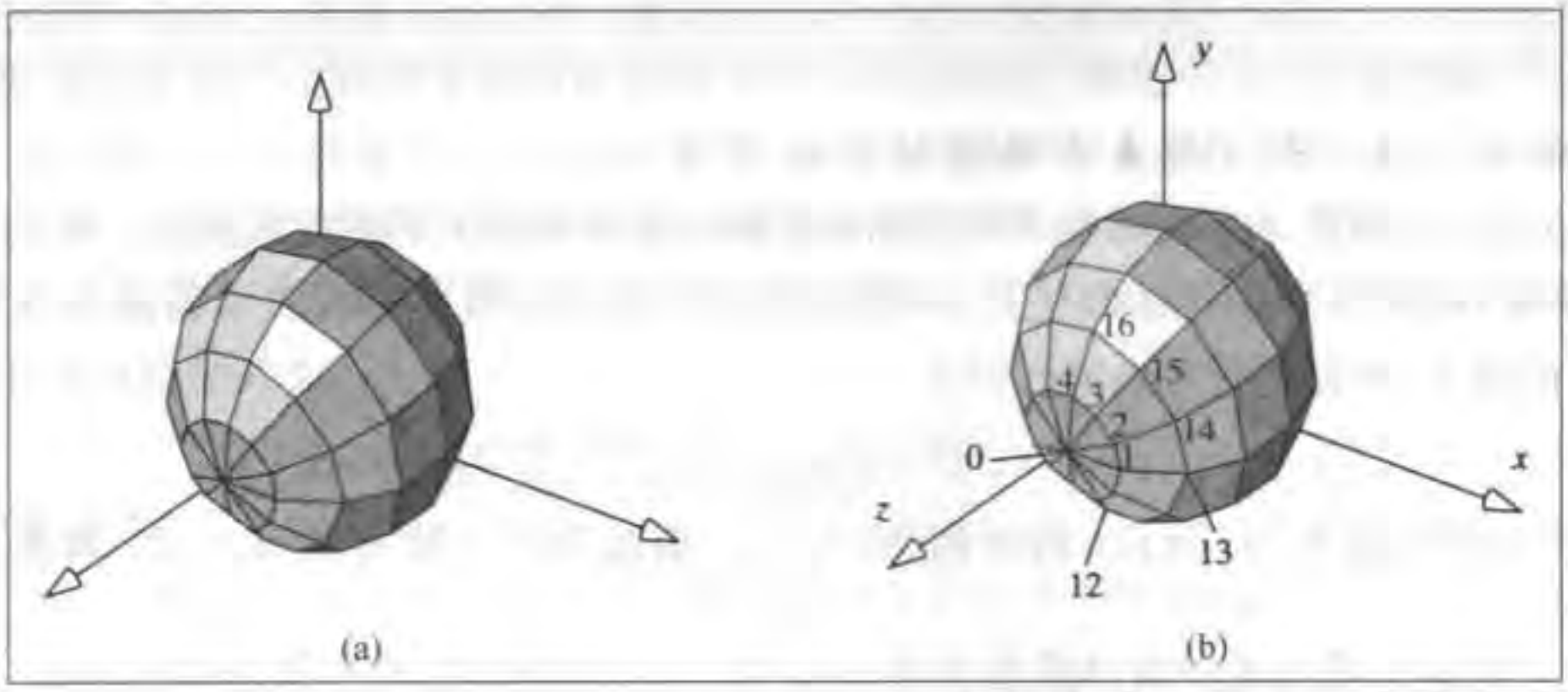


图 6.56

(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?))。

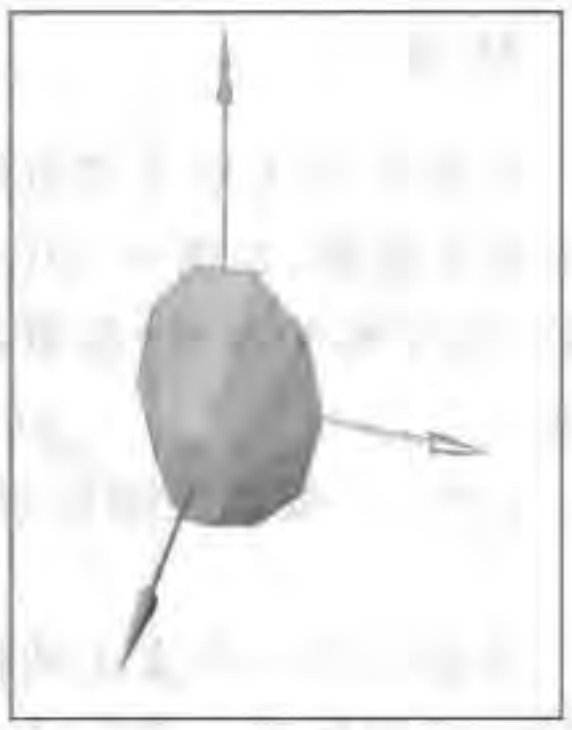


图 6.57　用网格近似逼近一端逐渐变小的柱面

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

$$P(u,v)=(1-v)P_0+vP_1(u)\quad\{\text{一般的锥面}\}\tag{6.33}$$

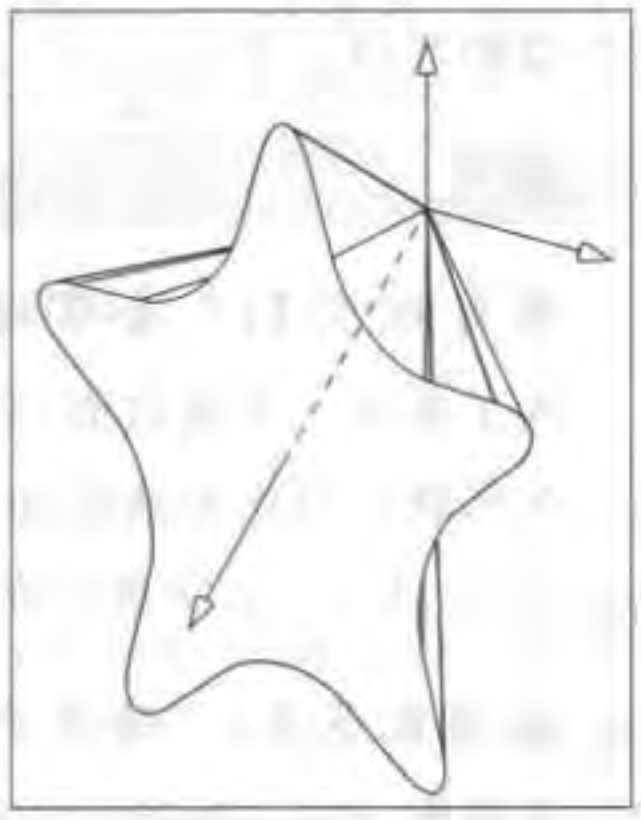


图 6.58　锥面

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。

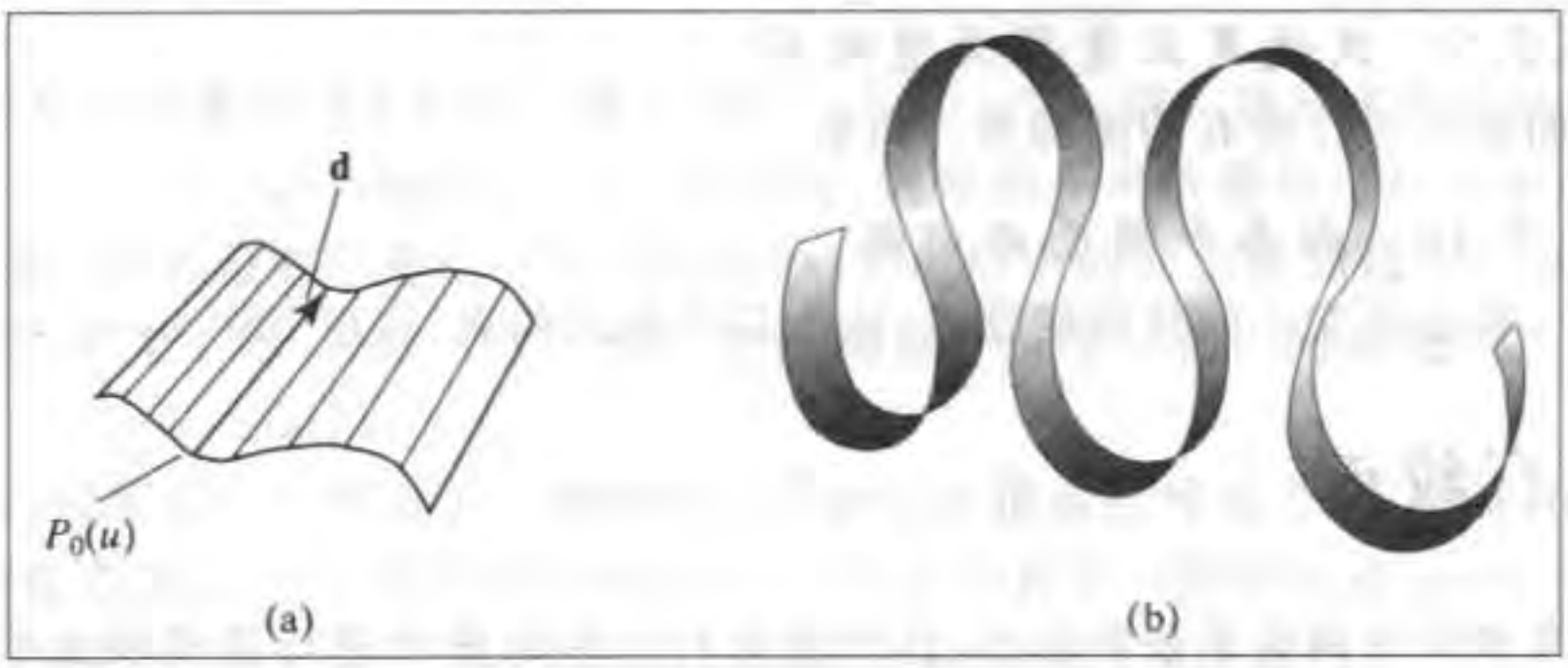


图 6.59
(a) 柱面;(b) 糖果彩带柱面

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

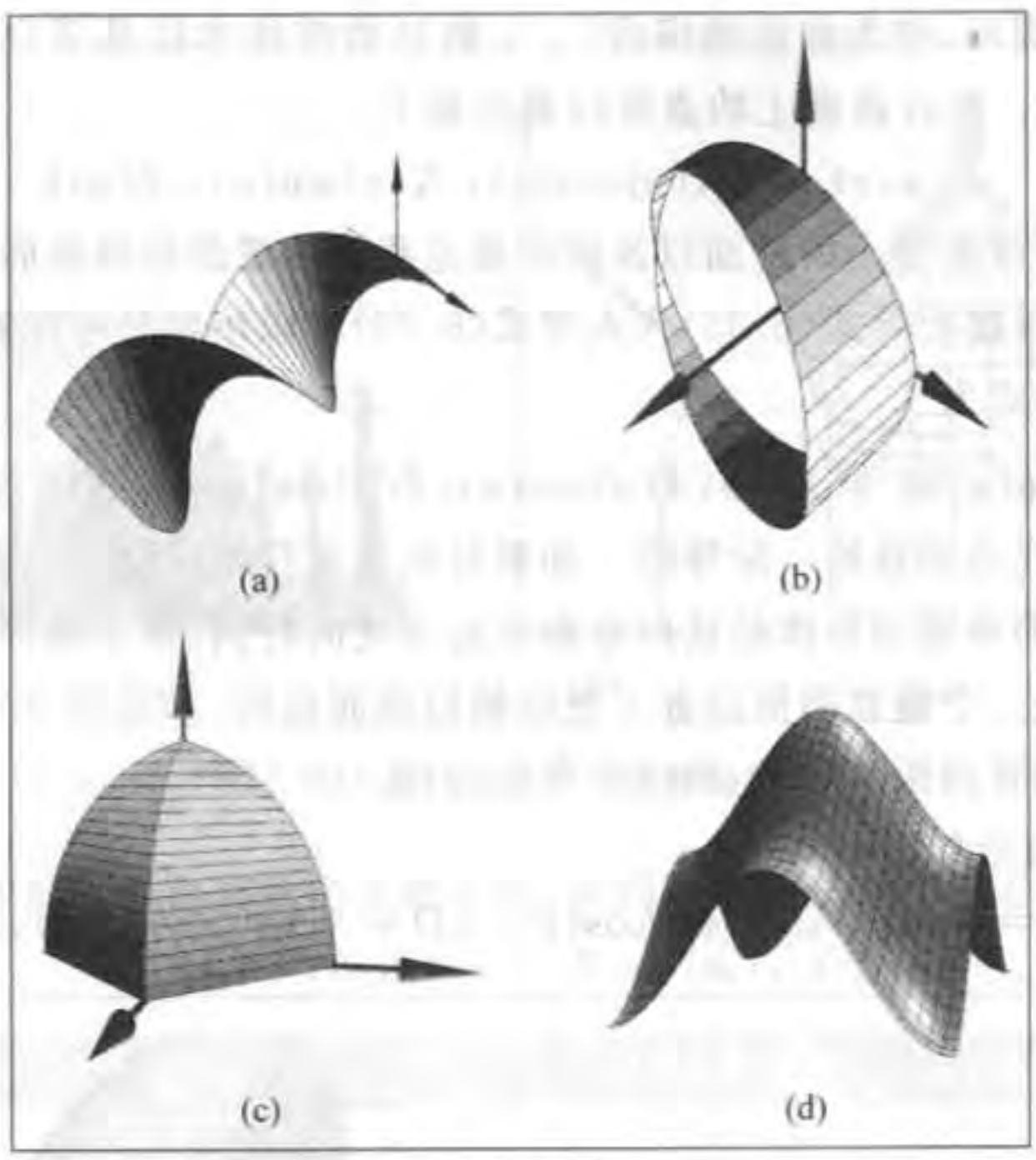


图　6.60

(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:

$$P(u,v) = ((D+A\cos(v))\cos(u),\ (D+A\cos(v))\sin(u),A\sin(v)) \tag{6.37}$$

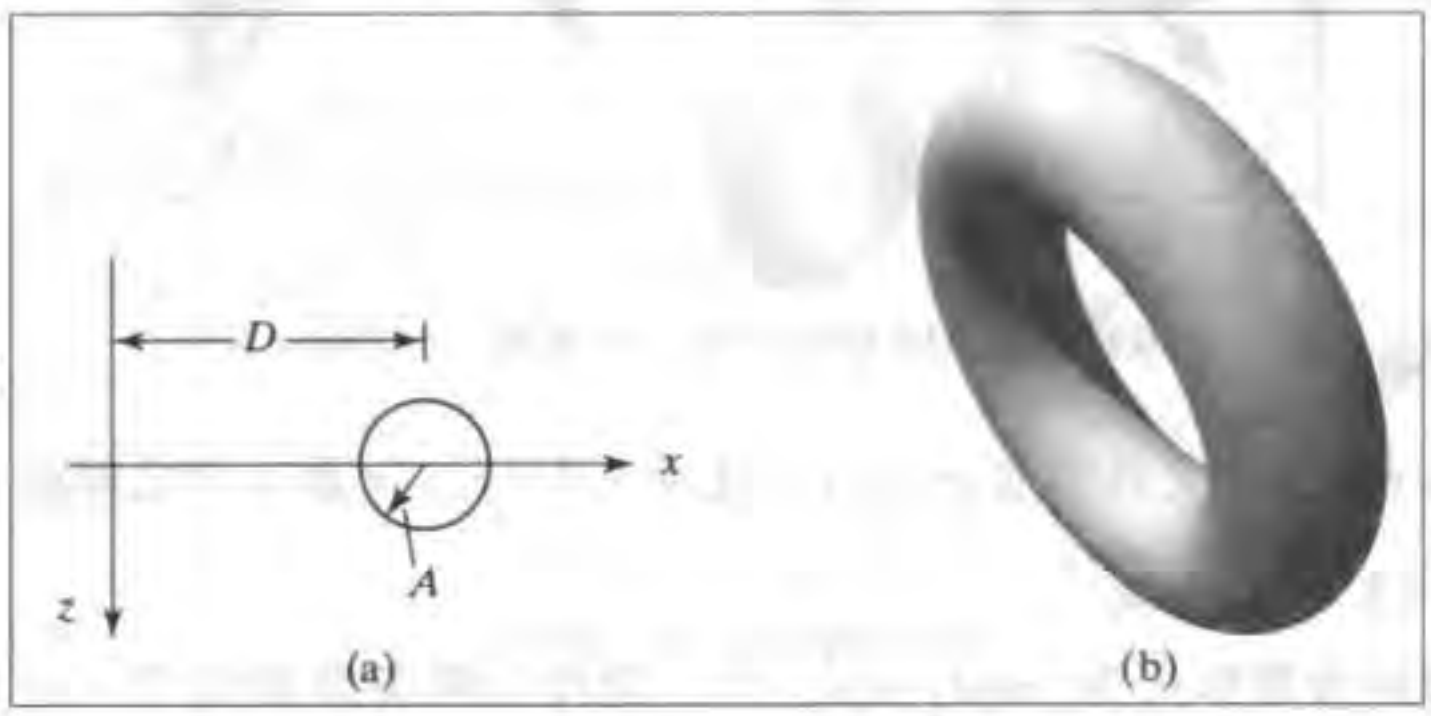


图 6.61 环面

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。

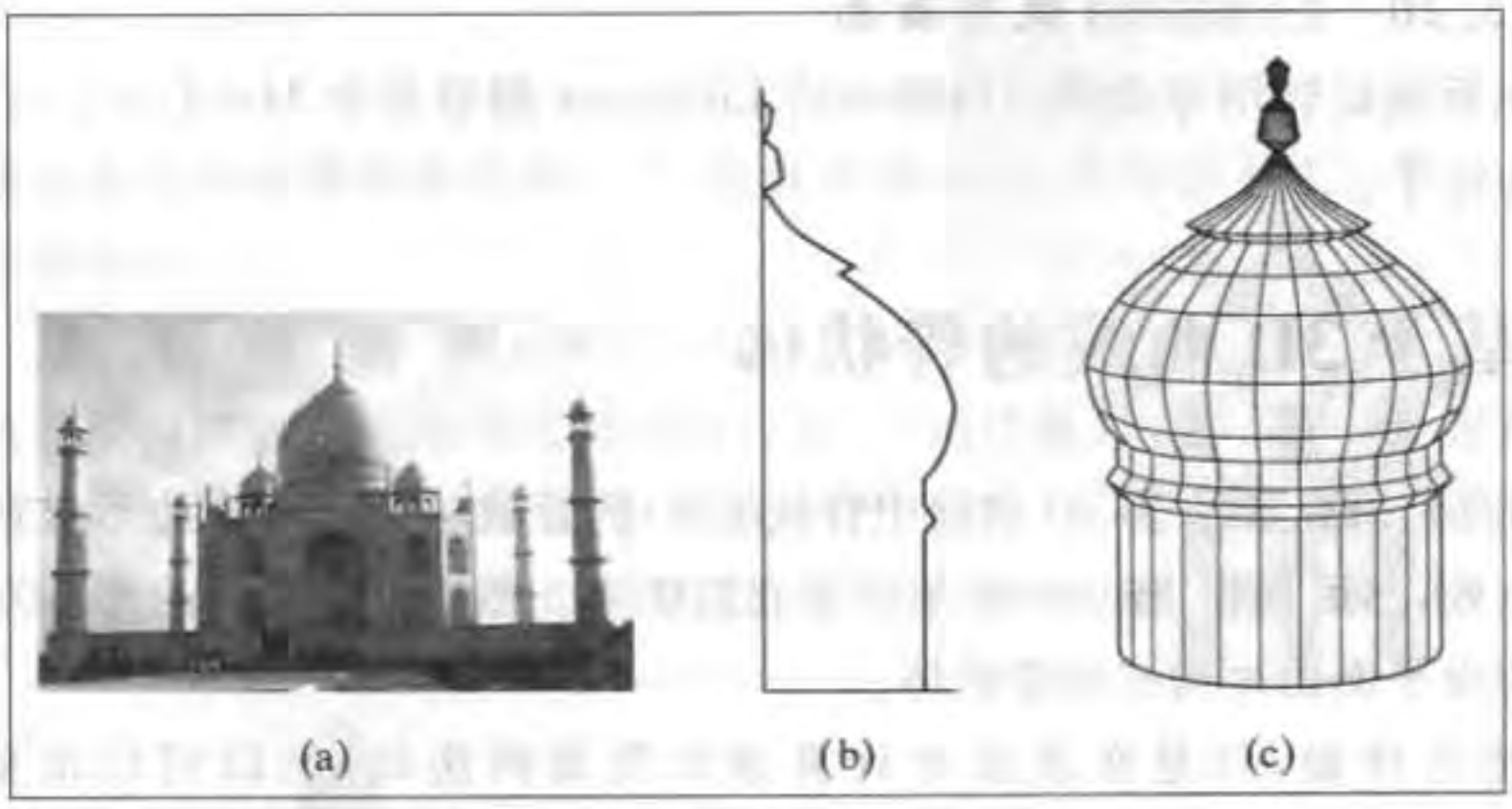


图 6.62　旋转曲面——泰姬陵穹顶

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。

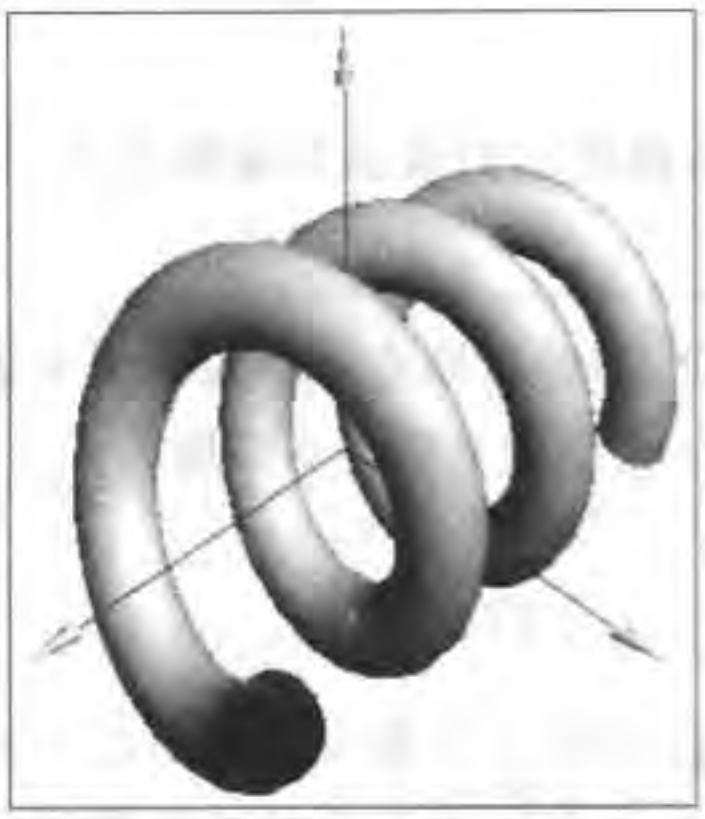


图 6.63 空间中的螺旋管状体

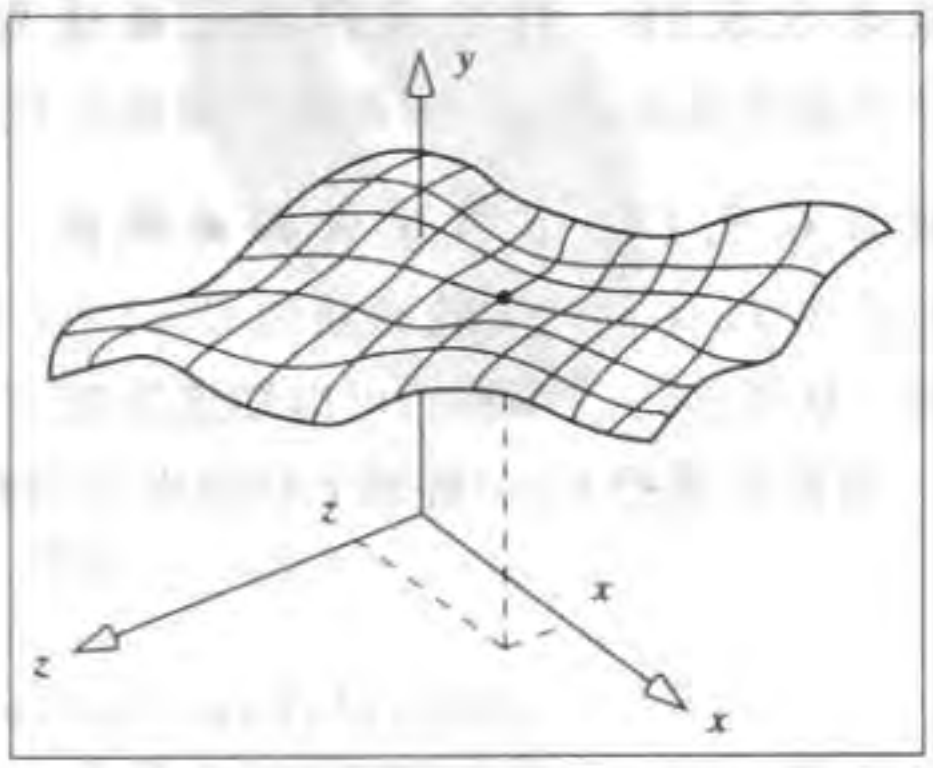


图 6.64 xz 坐标平面上的单值高度场

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。

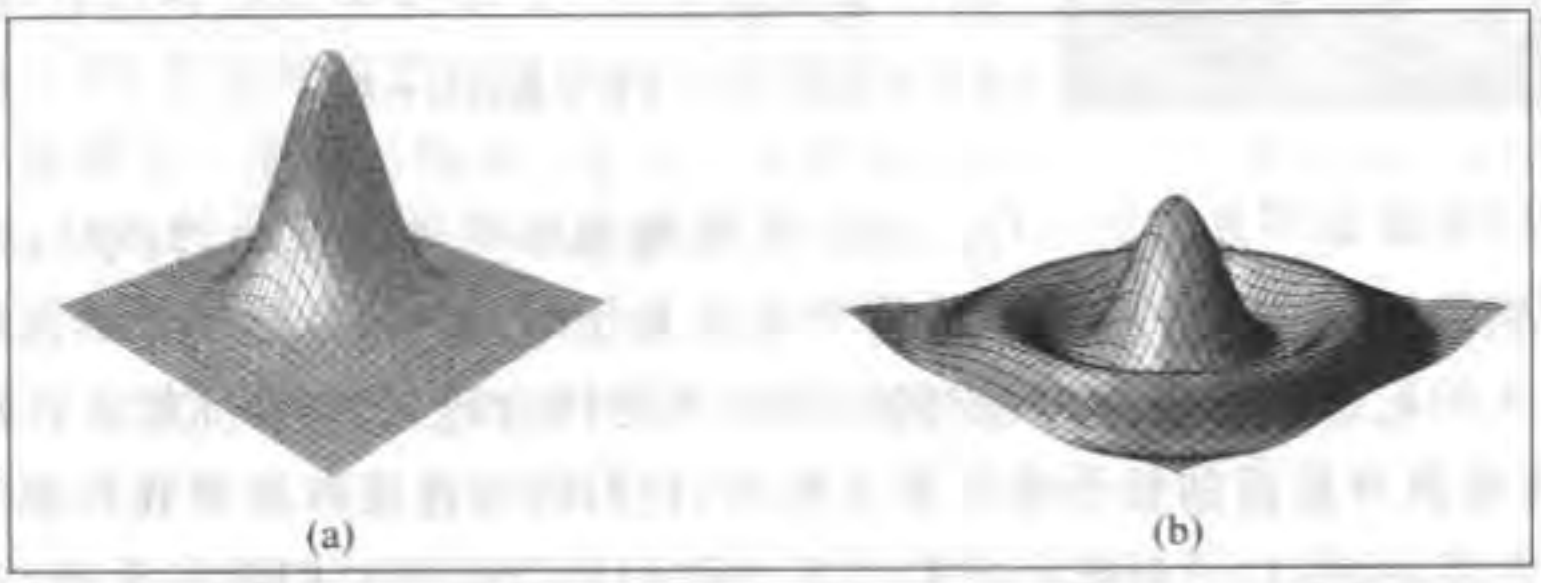


图 6.65　两种高度场

(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($f=ma$)。

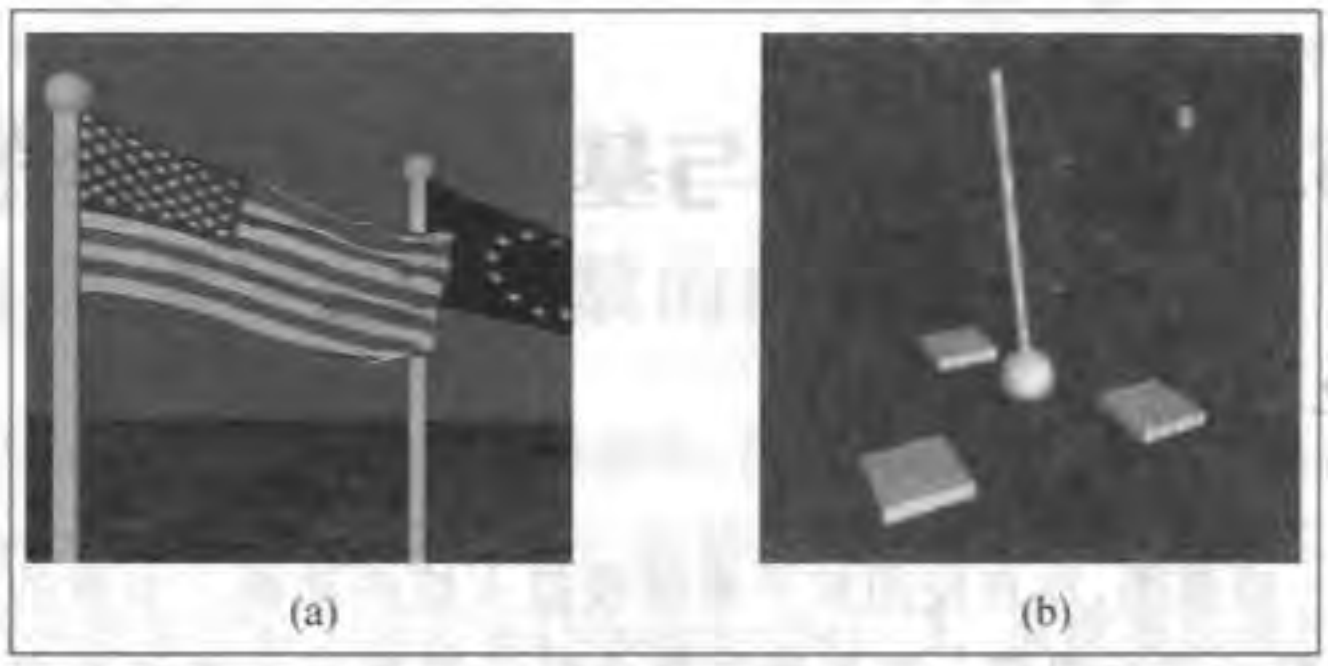


图 6.67

(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。

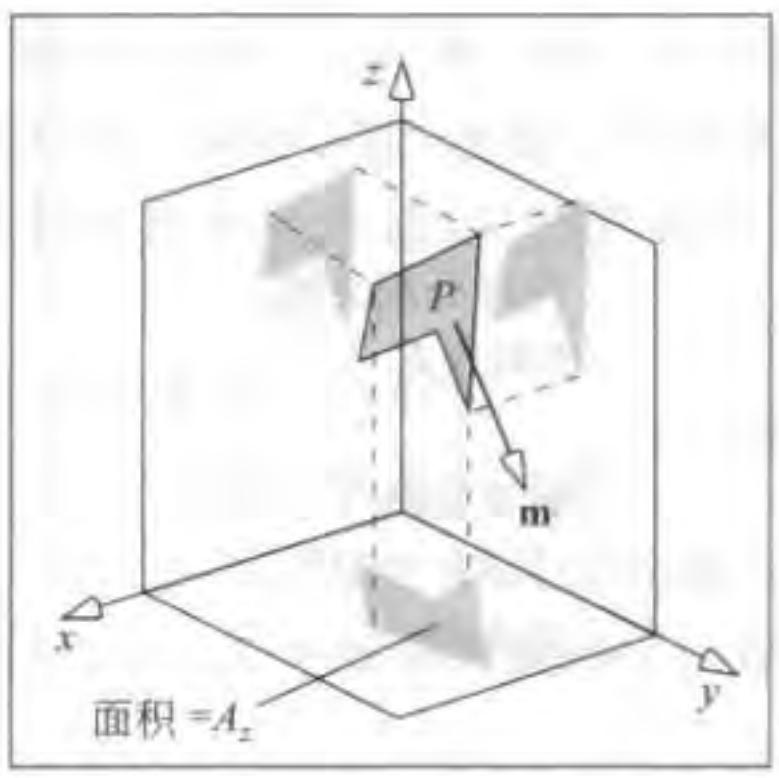


图 6.70 使用投影面积计算法向量

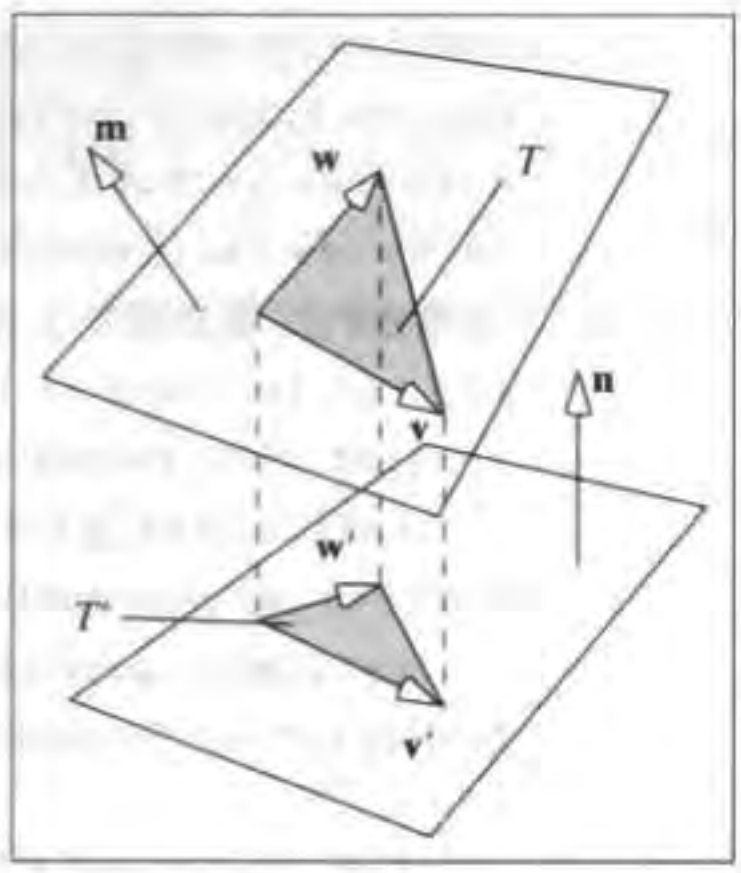


图 6.71 正交投影对面积的影响

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。

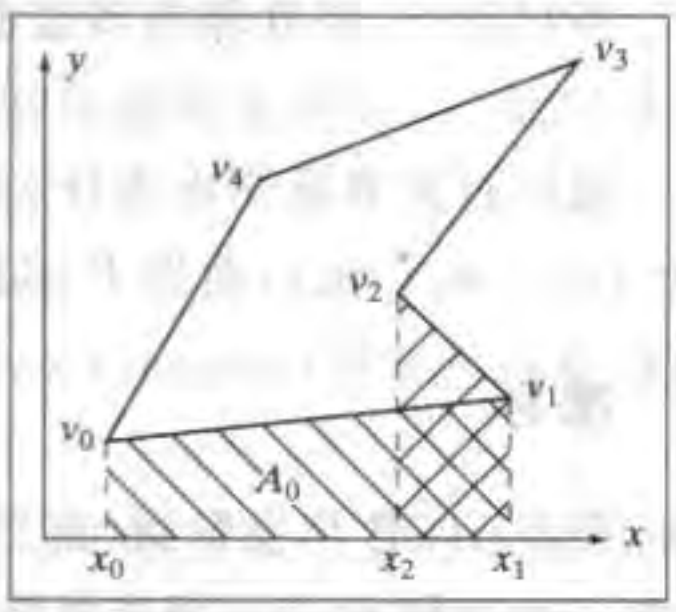


图 6.72 计算多边形的面积

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。

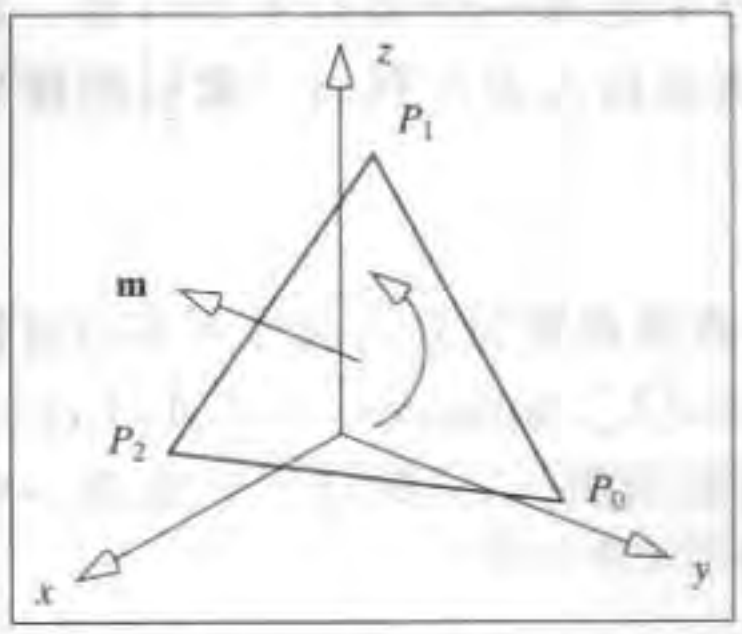


图 6.73　Newell 法计算的法线方向

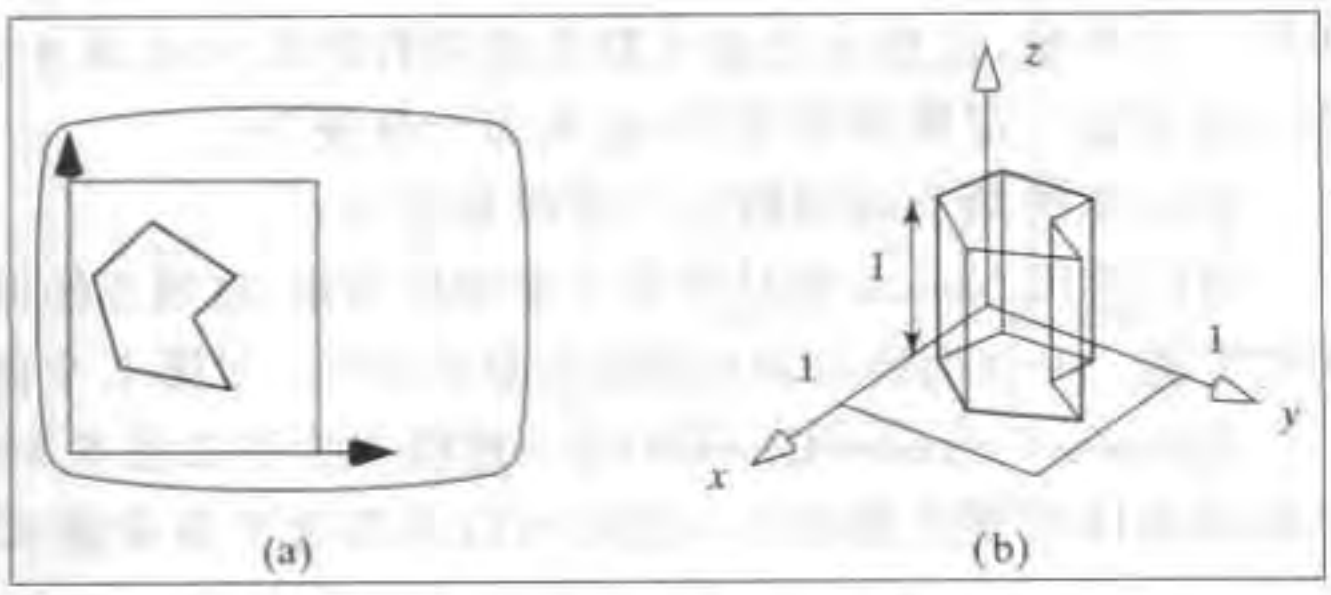


图 6.74　设计并生成一个棱柱网格

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。

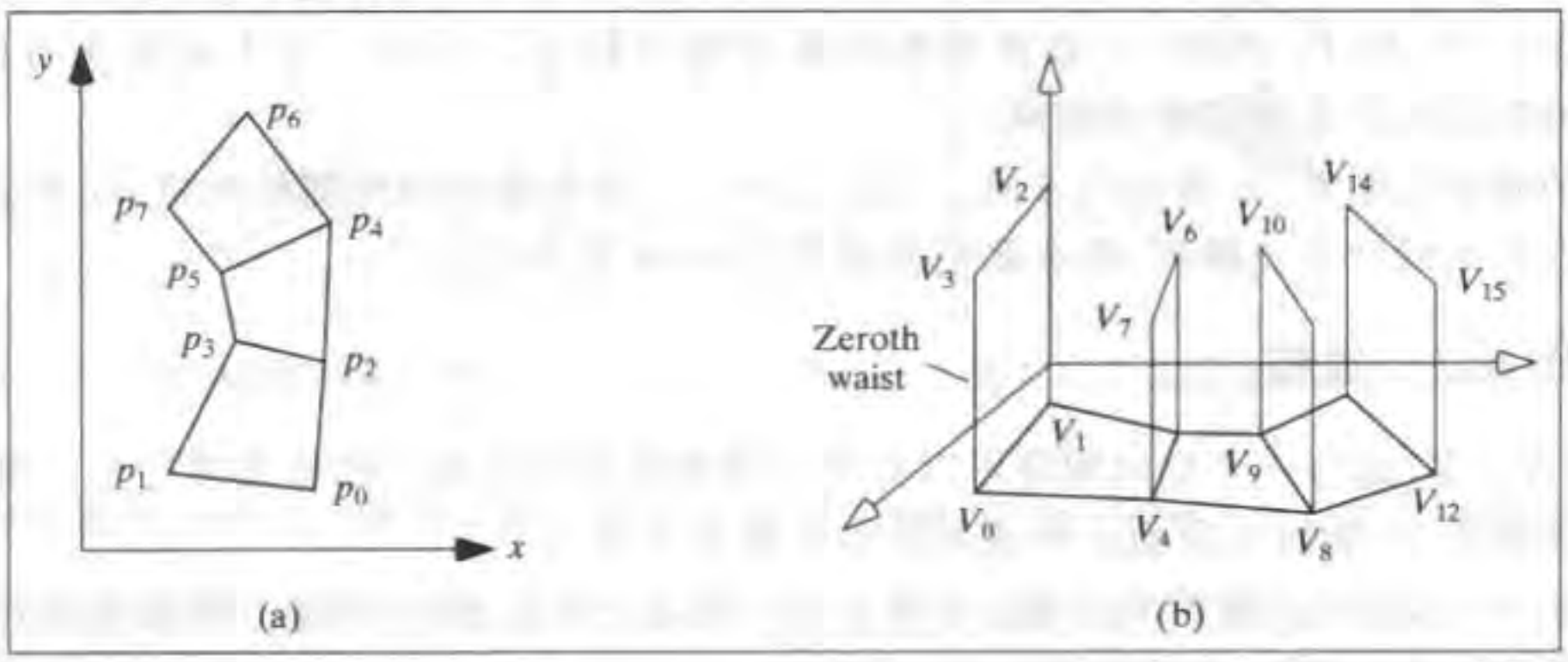


图 6.75 设计并生成一个棱柱网格

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。

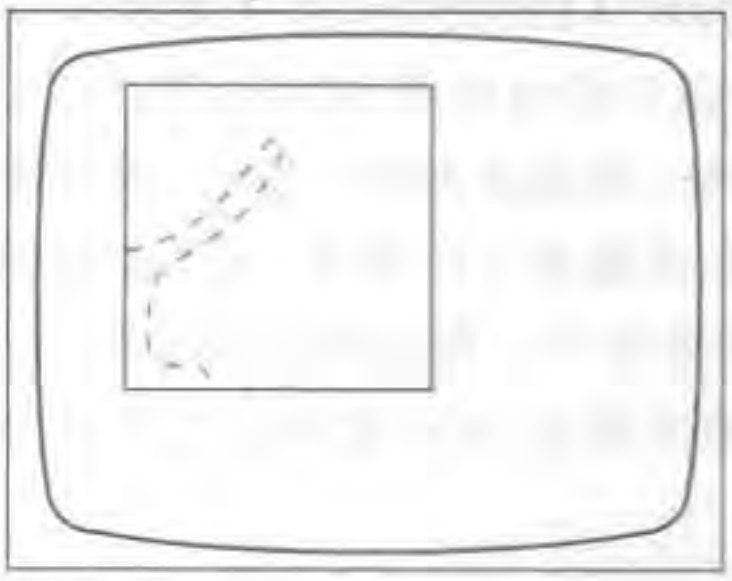


图 6.76　设计旋转曲面的轮廓

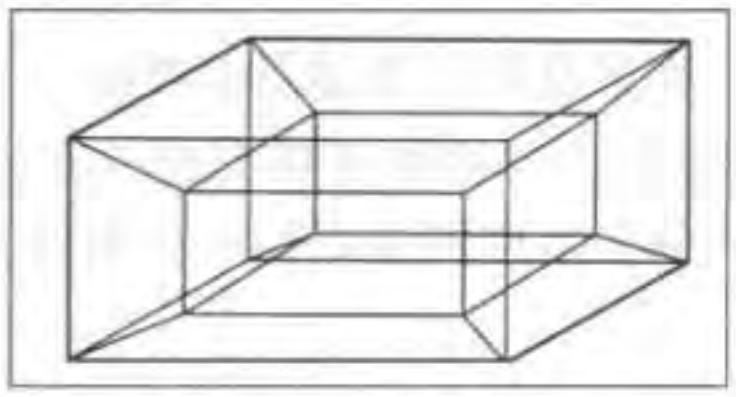


图 6.77　由二义性的物体

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。

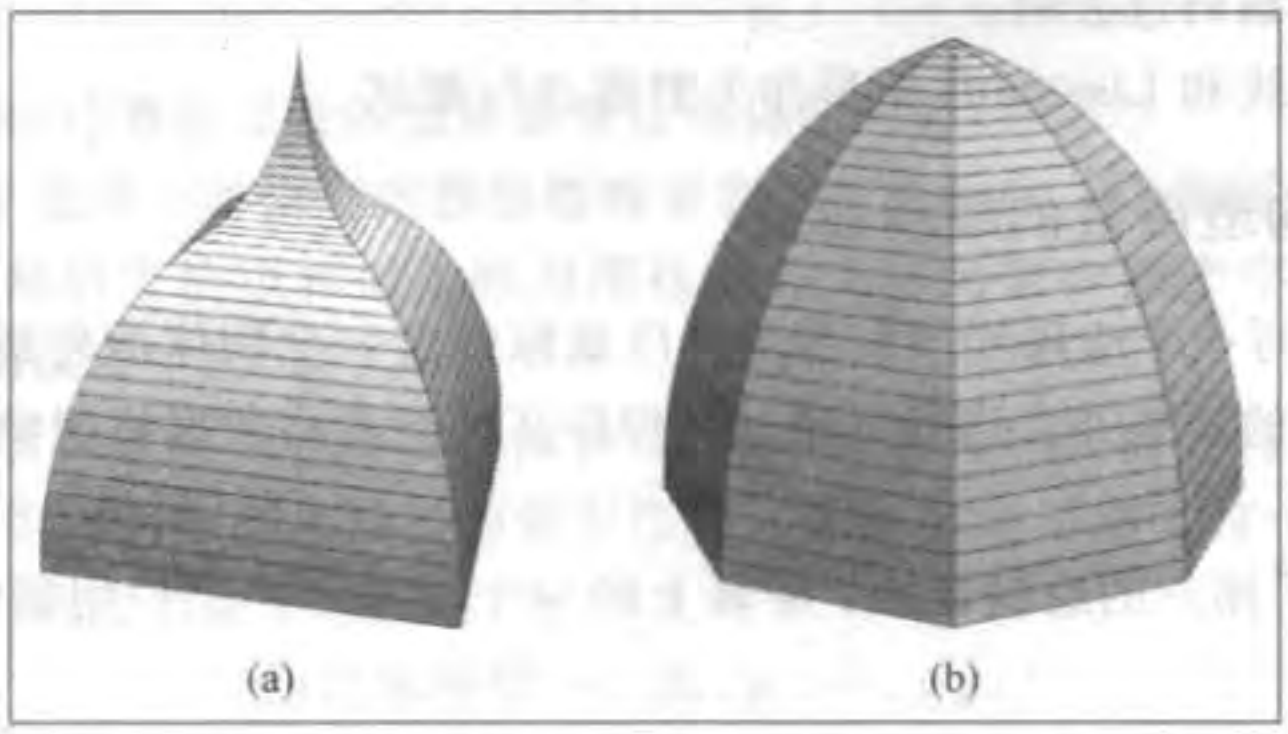


图 6.78 穹顶范例

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)。

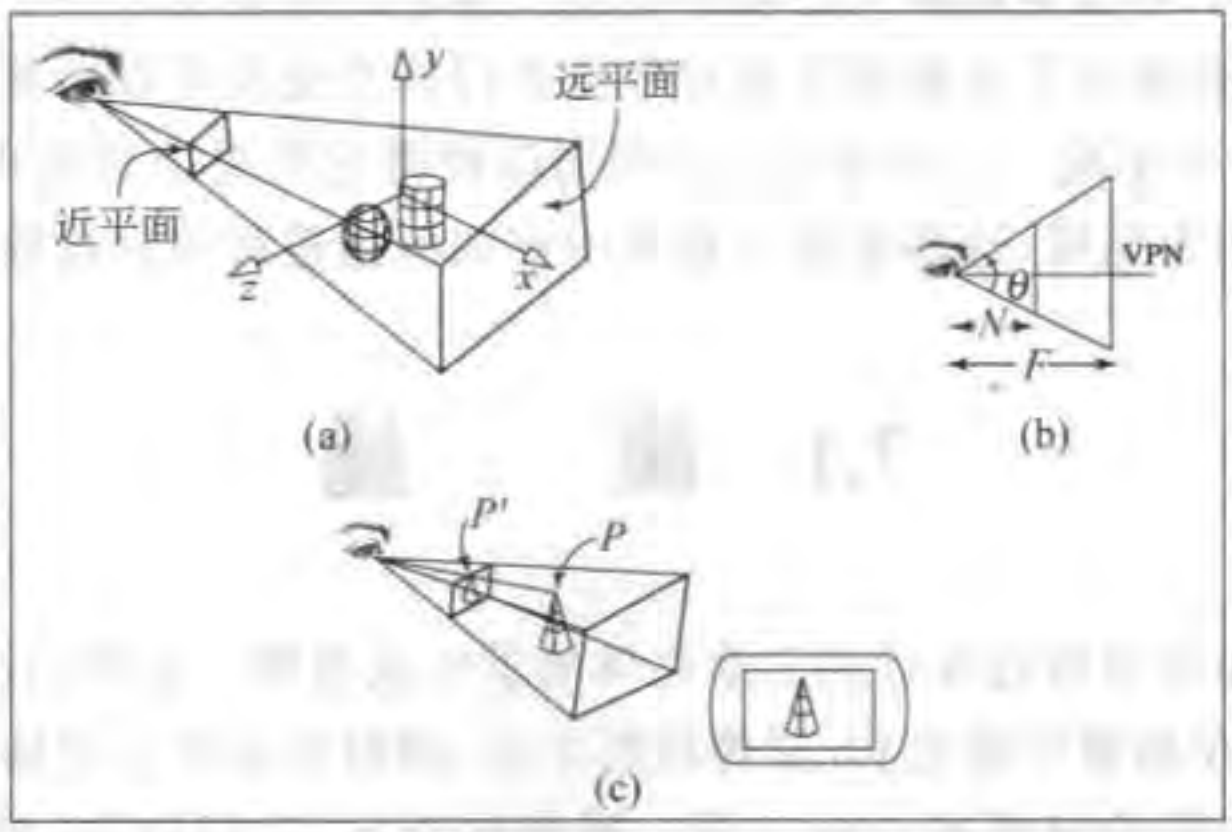


图 7.1 构造场景透视观察的摄像机

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)。

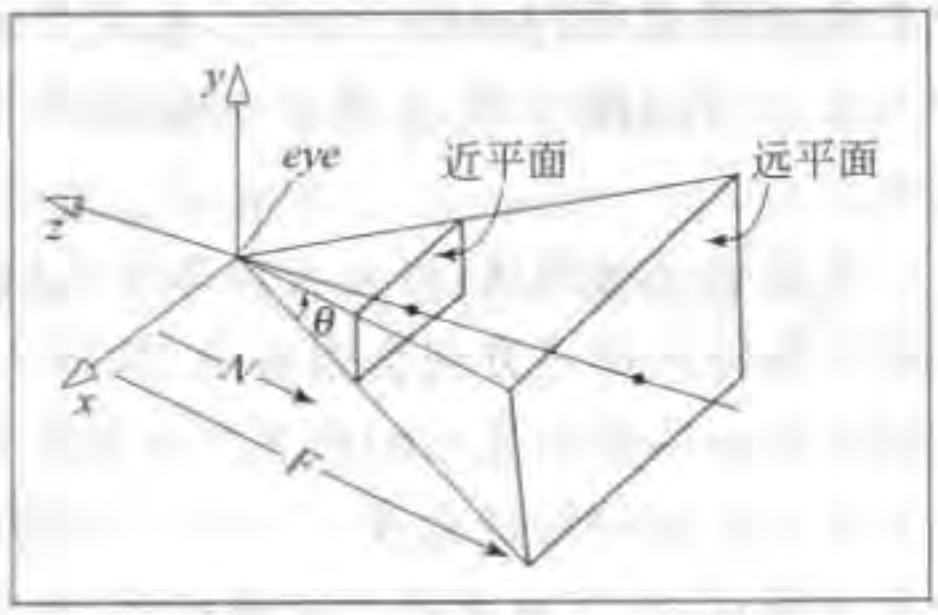


图 7.2 摄像机的默认位置

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。

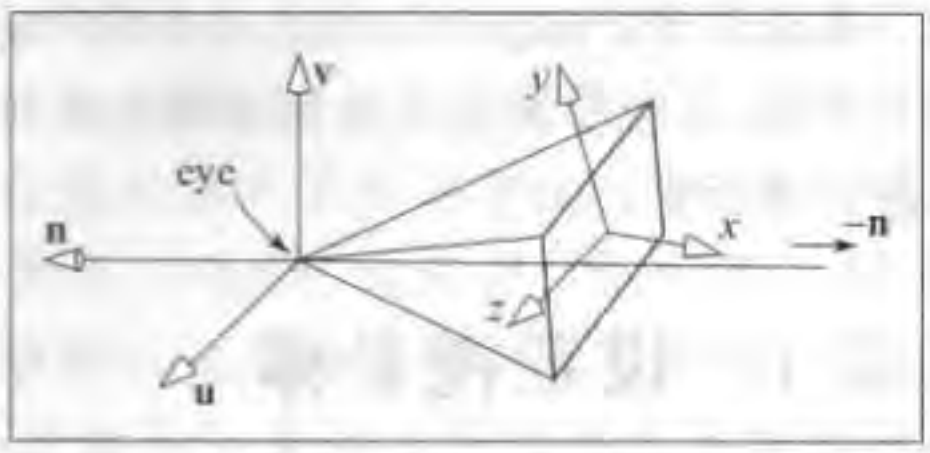


图 7.3 绑定摄像机到某个坐标系中

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。

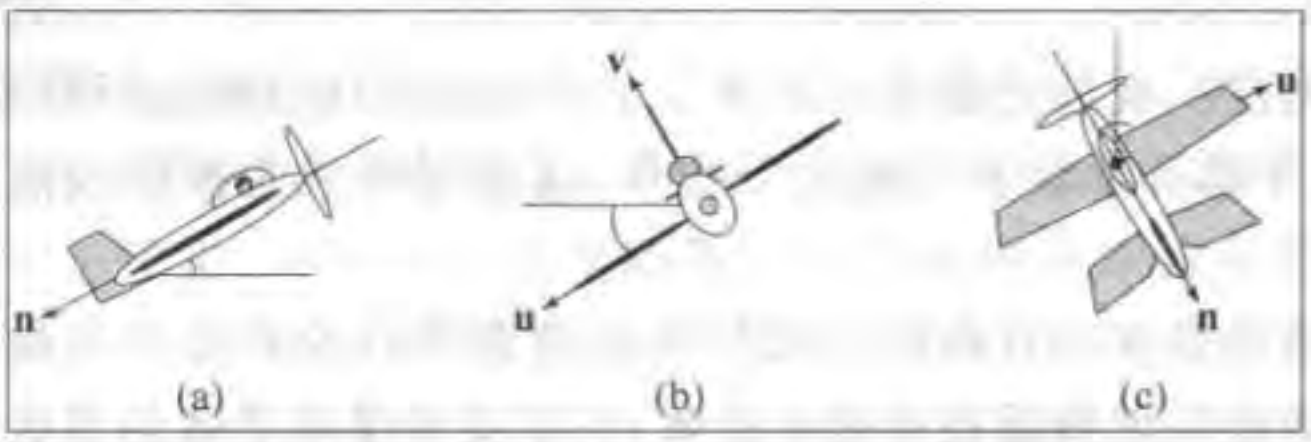


图 7.4 一架飞机相对于世界坐标系的朝向图示

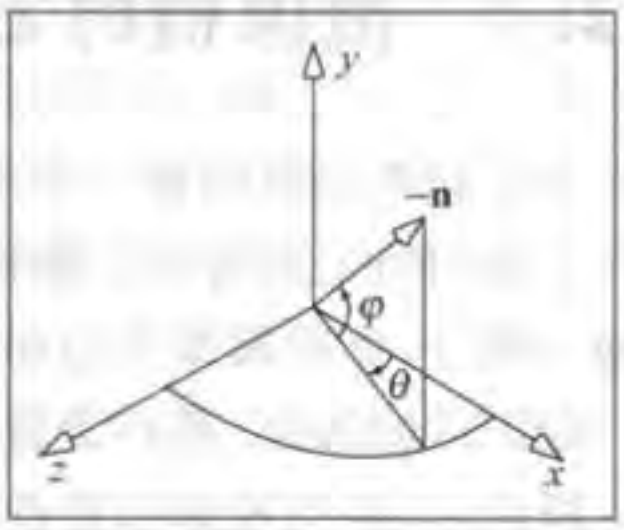


图 7.5 飞机的朝向和倾斜度

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

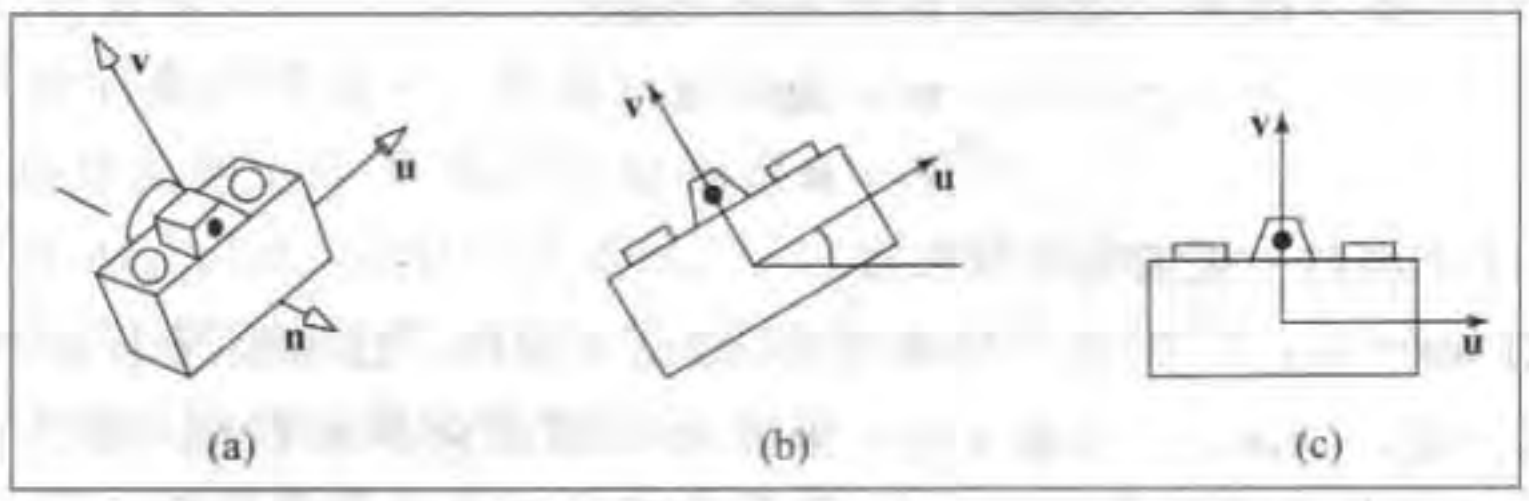


图 7.6 不同的摄像机姿态

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)。

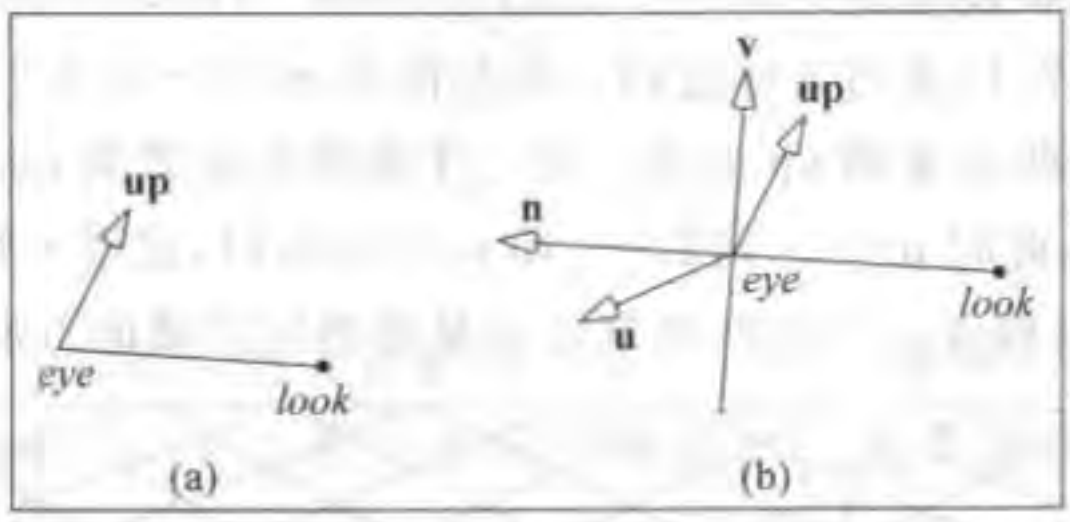


图 7.7 构建向量 **u**、**v** 和 **n**

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)。

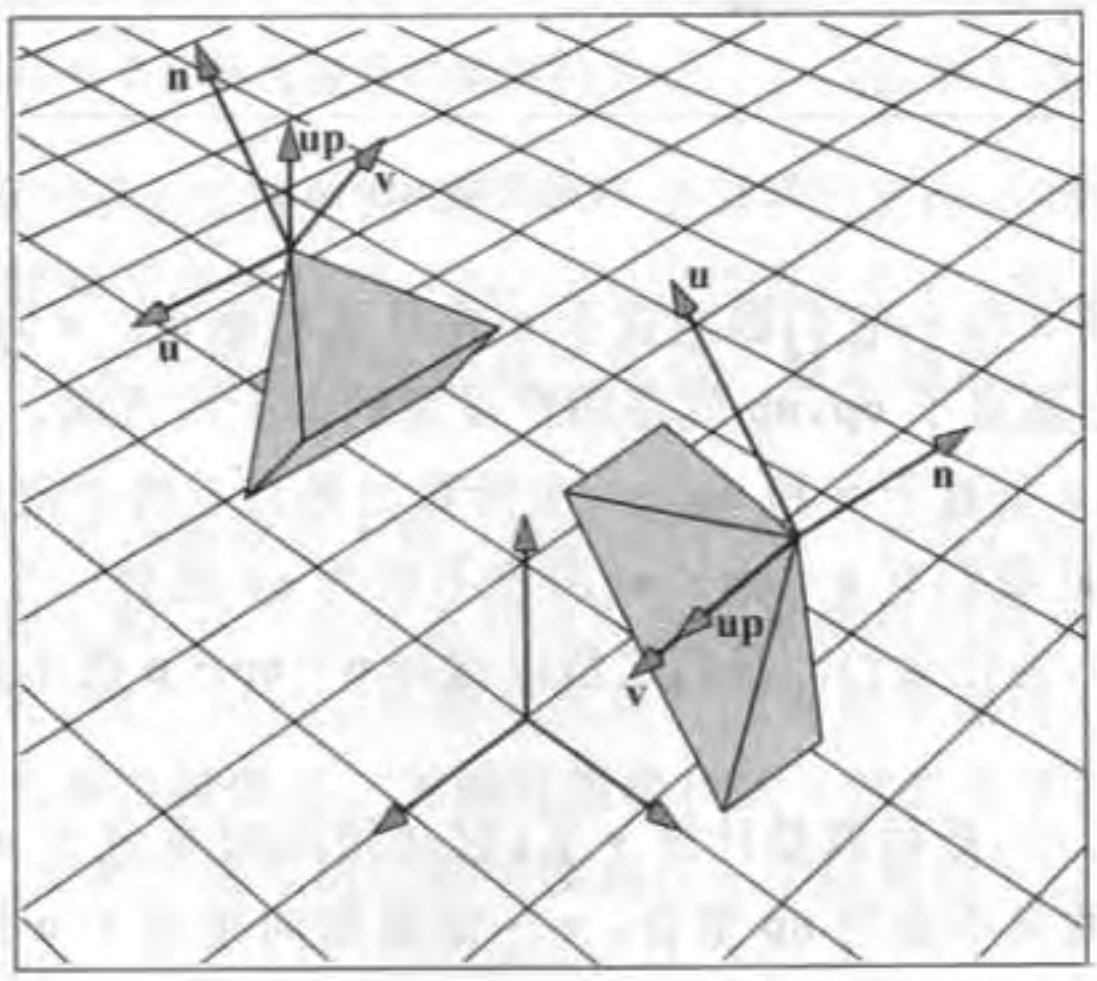


图 7.8 两个摄像机设置的图示

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：

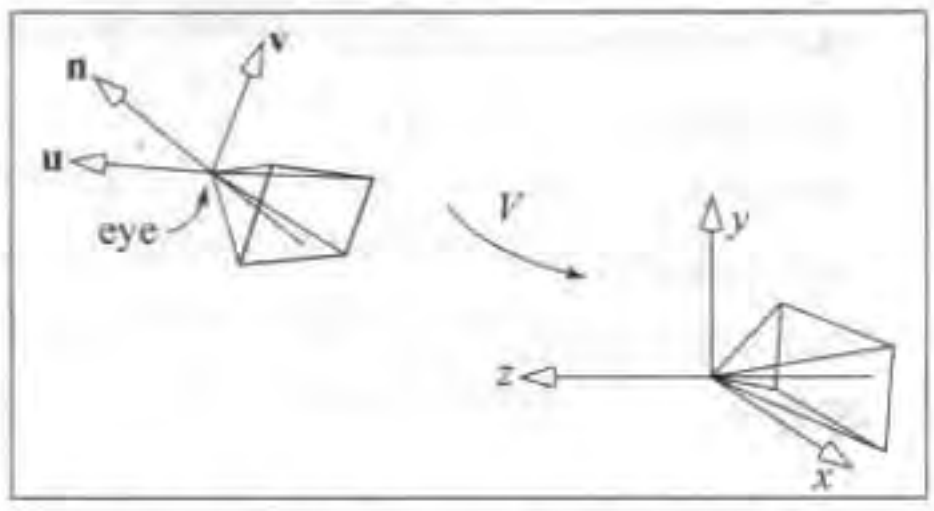


图 7.9 gluLookAt()函数设置的变换

$$V=\begin{bmatrix} u_x & u_y & u_z & d_x \\ v_x & v_y & v_z & d_y \\ n_x & n_y & n_z & d_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{7.2}$$

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 $\mathbf{v}'$。

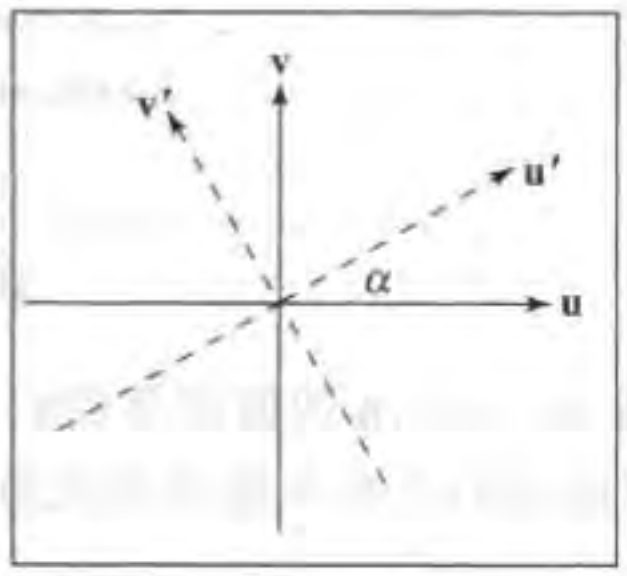


图 7.12 摄像机翻转

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。

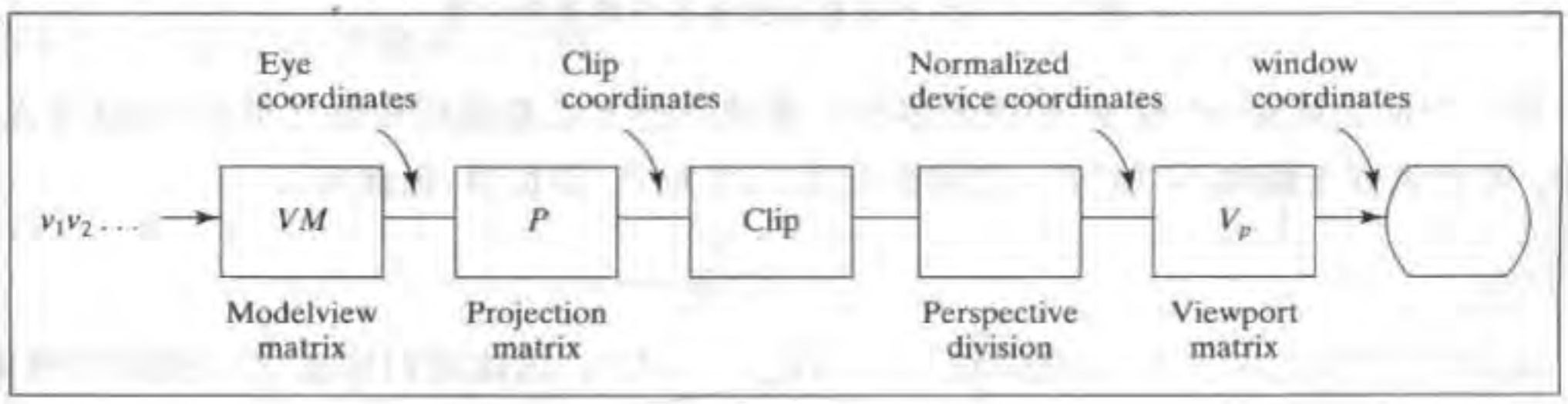


图 7.14　图形绘制管道

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。

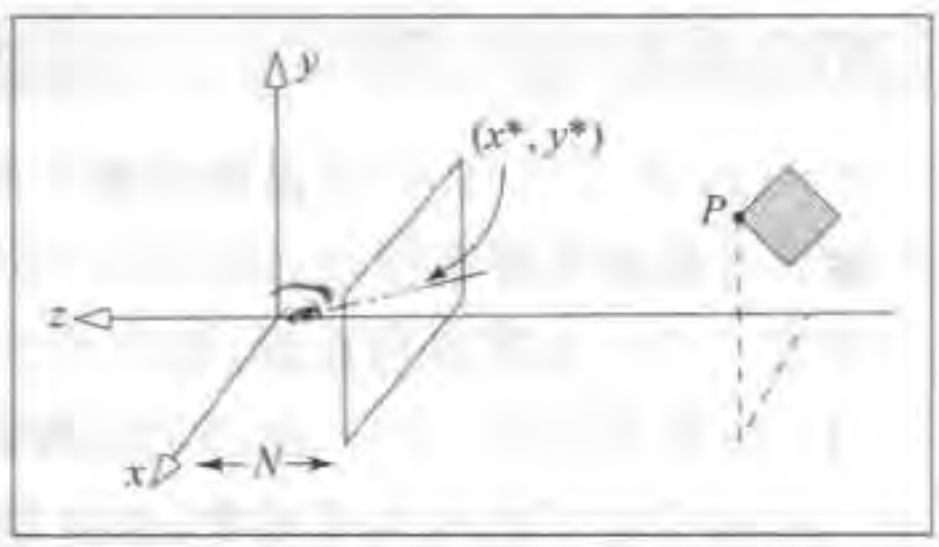


图 7.15　在视点坐标系中描述的一个顶点的透视投影

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。

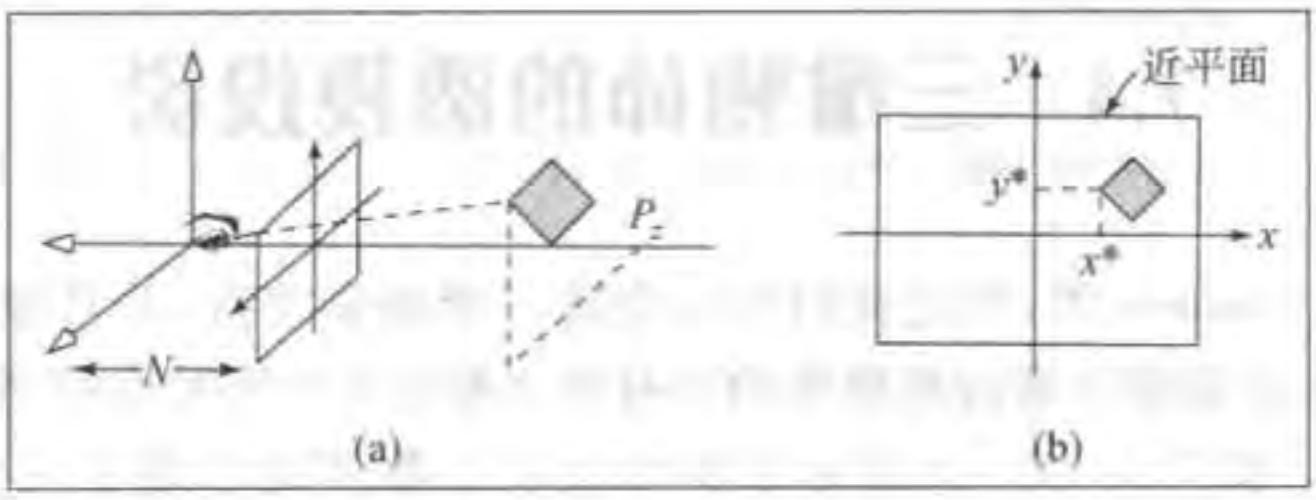


图 7.16 点 P 在视点坐标系中的投影位置

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。

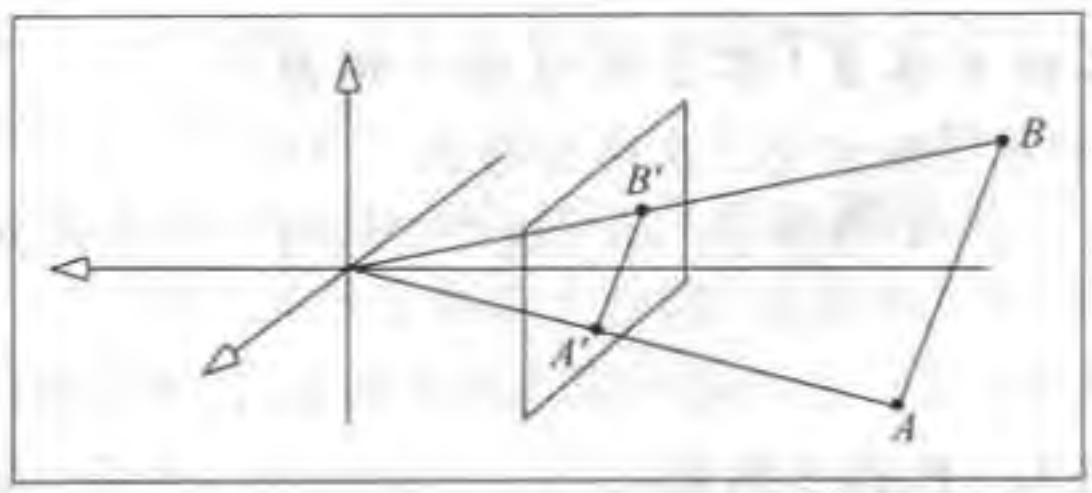


图 7.17　直线投影为直线的证明图示

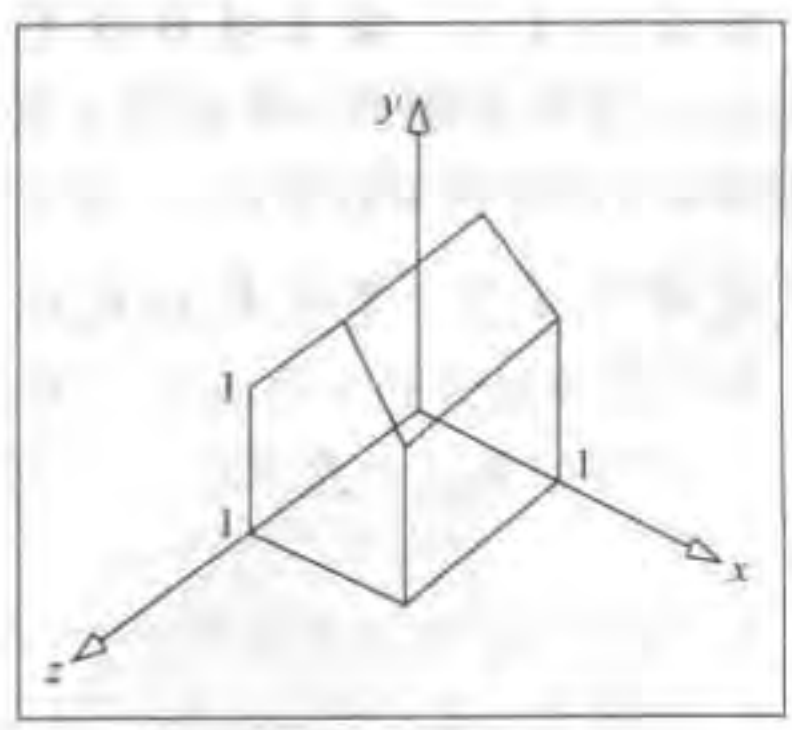


图 7.18　仓房的图示

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。

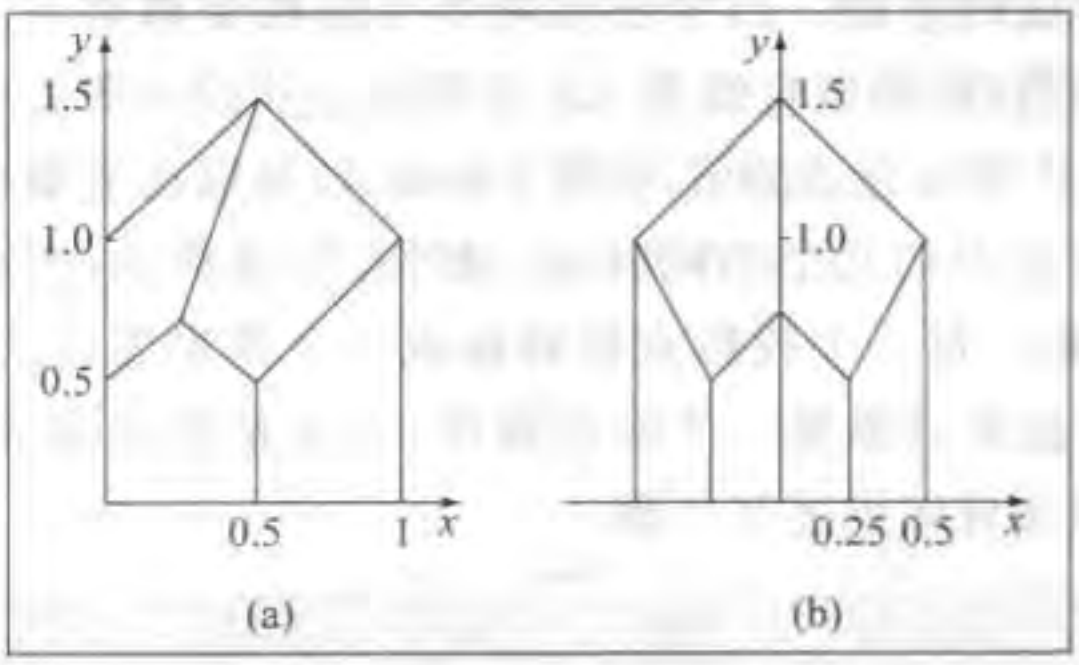


图 7.19 在视图#1 和视图#2 下的仓房投影

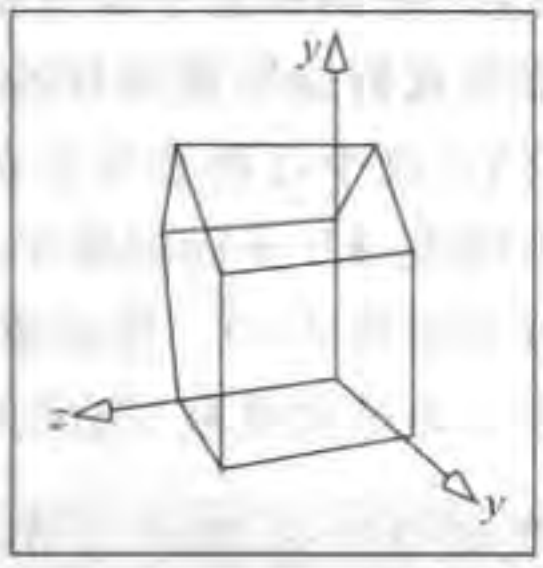


图 7.20 第三个视角下的仓房图像

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。

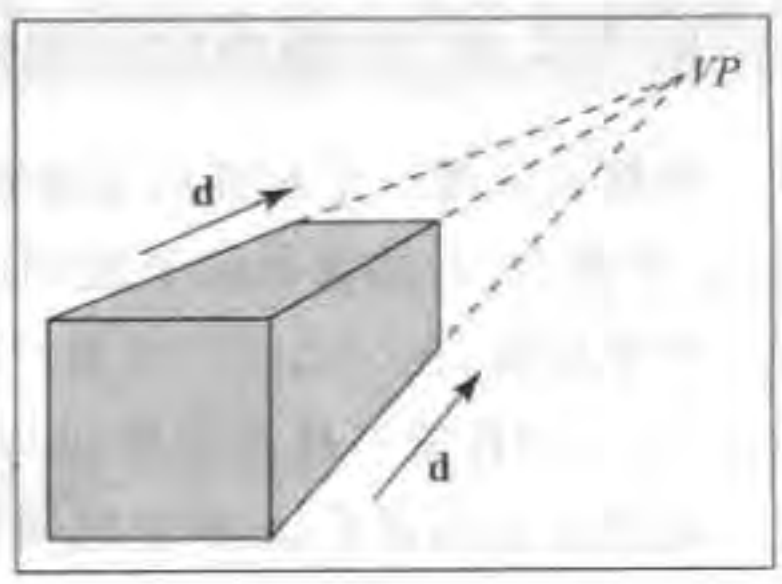


图 7.21　平行直线的灭点

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