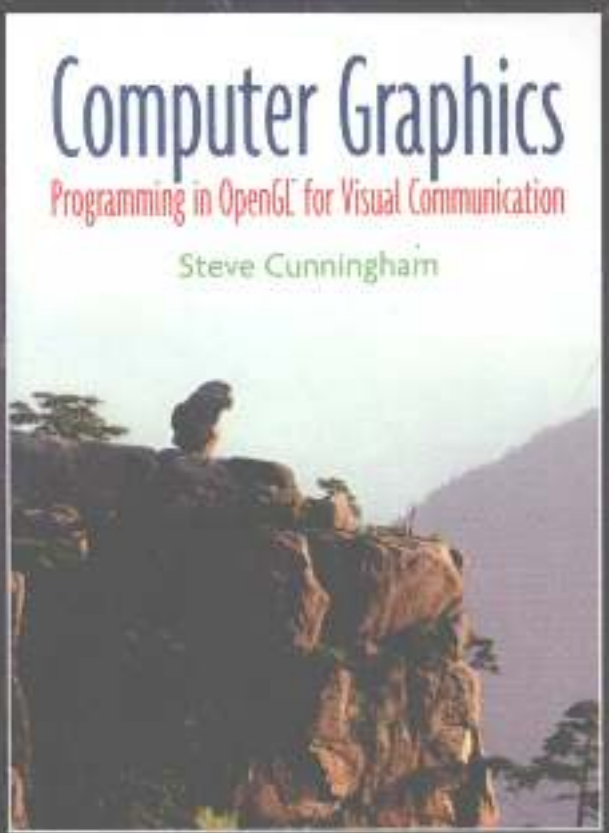
计 算 机 科 学 丛 书

计算机图形学

(美) Steve Cunningham 著 石教英 潘志庚 等译



Computer Graphics

Programming in OpenGL for Visual Communication

机械工业出版社
China Machine Press

计算机图形学 面向视觉交流的OpenGL编程技术

投影变换建模流水线绘制明暗处理，循序渐进
坐标视图颜色场景图光照视觉交流，由浅入深

本书主要介绍计算机图形学原理而不讨论实现这些原理的算法和数学细节，重点在于讲述如何采用图形API OpenGL的编程技术来解决实际问题。作者以描述性和面向过程的方式阐述了计算机图形学中的重要主题，使得计算机科学及相关专业的学生在学习阶段的早期便能接触并理解这些主题；同时使用OpenGL来说明计算机图形学的基本概念，使学生可以绕过图形学算法和数学细节，快速生成有意义的可交互且动态的三维图形，创建有效的视觉交流。

本书注重计算机图形学精髓的理解和图形编程技术的掌握，非常适合作为高等院校计算机及相关专业计算机图形学课程的教材，同时也适合作为具有熟练编程经验的其他专业学生和专业技术人员学习图形学及图形编程的自学教材。

主要特点

- 强调利用计算机图形进行有效的交流，特别是在科学领域。
- 广泛采用场景图组织图形程序。
- 首次在硬拷贝一章中介绍了三维硬拷贝（或称为快速原型生成）技术，在其他导论性教材中均没有该内容。
- 代码示例遍及全书，既包含伪代码，也包含全部OpenGL程序列表。
- 包含大量组织新颖独特的问题和练习。
  - 每章的学生问题划分为四部分：思考题、练习题、实验题和大型作业。
  - 这些问题帮助学生更深入地思考问题，进行编程练习，尝试新的思路和方法，以及开发大型的具有挑战性的项目。

作者简介

Steve Cunningham 美国著名的图形学专家，对计算机图形学理论和OpenGL编程均有很深的造诣。现任加州州立大学斯坦尼斯洛斯分校计算机系的Gemperle杰出教授，曾担任过ACM SIGGRAPH学会的主席和EURO GRAPHICS学会教育委员会的主任，长期活跃在计算机图形学的教育前沿，多次组织计算机图形学和可视化教学研讨会。



www.PearsonEd.com

PEARSON Education

投稿热线：(010) 88379604
购书热线：(010) 68995259, 68995264
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

ISBN 978-7-111-24102-7

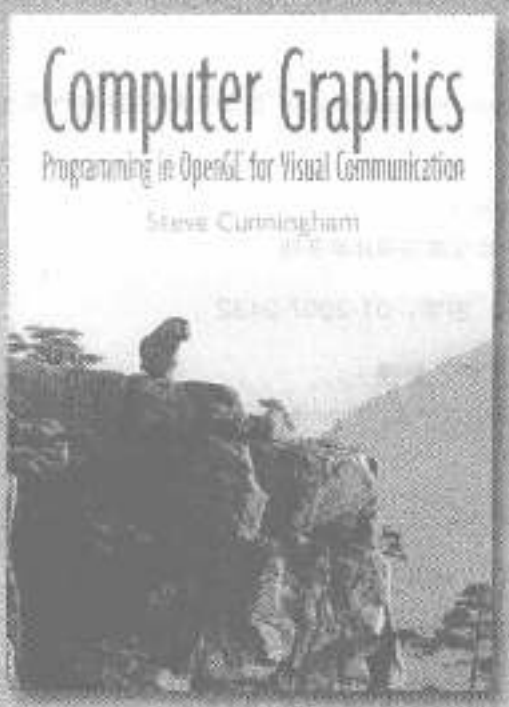
9 787111 241027

定价：48.00 元

计 算 机 科 学 丛 书

计算机图形学

(美) Steve Cunningham 著 石教英 潘志庚 等译



Computer Graphics

Programming in OpenGL for Visual Communication

机械工业出版社
China Machine Press

本书与大多数传统的计算机图形学教材不同，它仅简要介绍交互式计算机图形学方面的基本知识，主要侧重于介绍计算机图形学在数学及其他科学领域的应用，解决实际问题。本书按照计算机图形学的传统顺序介绍视觉交流、社图变换和投影处理、建模、绘制、光照、着色处理，以及OpenGL API如何实现基本概念和技术，使学生理解并学会使用图形API实现图形操作，为观察者创造有效的图像。

本书可作为高等院校计算机图形学的基础教材，对软件开发人员解决实际问题也有很高的参考价值。

Simplified Chinese edition copyright © 2008 by Pearson Education Asia Limited and China Machine Press.

Original English language title: Computer Graphics: Programming in OpenGL for Visual Communication (ISBN-13: 978-0-13-145254-1, ISBN-10: 0-13-145254-1) by Steve Cunningham, Copyright © 2007.

All rights reserved.

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

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。

本书法律顾问 北京市展达律师事务所

本书版权登记号：图字：01-2007-2432

图书在版编目（CIP）数据

计算机图形学/（美）坎宁安（Cunningham, S.）著；石教英，潘志庚等译. —北京：机械工业出版社，2008.6

ISBN 978-7-111-24102-7

Ⅰ.计… Ⅱ.①坎… ②石… ③潘… Ⅲ.计算机图形学 Ⅳ.TP 391.41

中国版本图书馆CIP数据核字（2008）第070208号

机械工业出版社（北京市西城区百万庄大街22号 邮政编码 100037）

责任编辑：王玉

三河市明辉印装有限公司印刷·新华书店北京发行所发行

2008年6月第1版第1次印刷

184mm×260mm · 23.75印张（含1印张彩插）

标准书号：ISBN 978-7-111-24102-7

定价：48.00元

凡购本书，如有倒页、脱页、缺页，由本社发行部调换

本社购书热线：（010）68326294

出版者的话

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

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

机械工业出版社华章分社较早意识到“出版要为教育服务”。自1998年开始，华章分社就将工作重点放在了遴选、移译国外优秀教材上。经过多年的不懈努力，我们与Pearson，McGraw-Hill，Elsevier，MIT，John Wiley & Sons，Cengage等世界著名出版公司建立了良好的合作关系，从他们现有的数百种教材中甄选出Andrew S. Tanenbaum，Bjarne Stroustrup，Brain W. Kernighan，Dennis Ritchie，Jim Gray，Afred V. Aho，John E. Hopcroft，Jeffrey D. Ullman，Abraham Silberschatz，William Stallings，Donald E. Knuth，John L. Hennessy，Larry L. Peterson等大师名家的一批经典作品，以“计算机科学丛书”为总称出版，供读者学习、研究及珍藏。大理石纹理的封面，也正体现了这套丛书的品位和格调。

“计算机科学丛书”的出版工作得到了国内外学者的鼎力襄助，国内的专家不仅提供了中肯的选题指导，还不辞劳苦地担任了翻译和审校的工作，而原书的作者也相当关注其作品在中国的传播，有的还专程为其书的中译本作序。迄今，“计算机科学丛书”已经出版了近两百个品种，这些书籍在读者中树立了良好的口碑，并被许多高校采用为正式教材和参考书籍，其影印版“经典原版书库”作为姊妹篇也被越来越多实施双语教学的学校所采用。

权威的作者、经典的教材、一流的译者、严格的审校、精细的编辑，这些因素使我们的图书有了质量的保证。随着计算机科学与技术专业学科建设的不断完善和教材改革的逐渐深化，教育界对国外计算机教材的需求和应用都将步入一个新的阶段，我们的目标是尽善尽美，而反馈的意见正是我们达到这一终极目标的重要帮助。华章分社欢迎老师和读者对我们的工作提出建议或给予指正，我们的联系方法如下：

华章网站：www.hzbook.com

电子邮件：hzedu@hzbook.com

联系电话：(010) 68995264

联系地址：北京市西城区百万庄南街1号

邮政编码：100037

华章教育

华章科技图书出版中心

译者序

我们应机械工业出版社编辑的邀请，组织翻译本书。这本书的作者Steve Cunningham先生是美国著名的图形学专家，曾经担任过ACM SIGGRAPH学会的主席和EUROGRAPHICS学会教育委员会的主任，并曾多次组织计算机图形学和可视化教学研讨会。

Steve Cunningham先生也是我们在美国的老朋友，他多次应邀访问中国，为我们做学术报告，对中美两国在计算机图形学方面的学术交流做出了重要贡献。另外，他对中国的美丽风景也是情有独钟，仅在2006年，他就二次来我国访问旅游。Steve Cunningham先生是美国加州大学Stanislaus分校计算机系资深教授，长期从事计算机图形学教学和研究工作，他对计算机图形学理论和OpenGL编程均有很深的造诣。他愿意花时间把自己的经验写出来和大家分享是一件非常好的事情。

图形学理论经过这么多年的发展，图形学技术本身也发生了很大的变化，可以用下面两句话来概括图形学的发展：

绘点绘线绘面绘体，描绘五彩世界

求好求快求新求美，追求永无止境

OpenGL作为一个性能优越的图形应用程序设计接口（API），适用于广泛的计算机环境，它已成为目前的三维图形开发标准，是从事三维图形开发工作的技术人员所必须掌握的开发工具。OpenGL有十分广泛的应用领域，如军事、电视广播、CAD/CAM/CAE、娱乐、艺术造型、医疗影像、虚拟现实等。本书的内容及特点可以用下面的对联给以概括：

投影变换建模绘制流水线着色处理，循序渐进

坐标视图颜色光照场景图视觉交流，由浅入深

就像本书作者在前言中所叙述的，本书非常适合于作为图形学的教材来用，也适合学习图形学及图形编程的自学者阅读。本书系统地介绍了交互式计算机图形学的基础知识和OpenGL图形接口，并给出一些例子帮助理解OpenGL提供的功能，每章附有思考题、练习题、实验题和大型作业，供读者检查自己掌握本书内容的程度。因此，本书是一本优秀的计算机图形学教材。

我们给本书读者提出的希望是：**认认真真，夯实图形基础；踏踏实实，掌握编程技巧。**希望读者通过本书的学习，能理解图形学的精髓、学会图形编程技术、找到一份好的工作。诚能如此，我们也就很欣慰了。

最后，要感谢参加本书翻译工作的张明敏副教授和各位研究生，他们各自负责翻译的部分是：前言和第0章（陈薇薇）、第1章和第6章（潘卫敏）、第2章和第3章（程杰）、第4章和第9章（朱杰杰）、第5章和第8章（张明敏）、第7章和第11章（王田和王总辉）、第10章和第15章（彭浩宇）、第12章和第13章（路华）、第14章和附录（张亚萍）。本书由石教英教授负责审校前言、第0章、第7章、第10～15章和附录，由潘志庚研究员负责审校第1～6章、第8章和第9章。另外，我们也要感谢机械工业出版社的编辑的组织和指导工作。

由于时间仓促，本书的翻译难免有疏漏和不当之处，敬请读者指正，我们会在后续的版本中更正。

感谢Mike Bailey提供优秀的计算机图形学教学示例和本书的许多写作思路。

封面照片是中国安徽省黄山著名风景——“猴子观海”。这张照片由本书作者摄于2006年6月的一次日出前。日出时刻是公认的观看黄山云海的最佳时间。下面这张黑白照片是“猴子观海”景点的铭牌。



作者以封面照片献给浙江大学石教英教授，一位中国计算机图形学界长期以来的领军人物。照片上的那只石猴子看起来很像一名学生正在沉思他的课程作业（当然是计算机图形学作业！）。

石教英、潘志庚
2007年12月于浙江大学

前　言

计算机技术对世界产生了重大影响，而计算机图形学正是其最令人振奋的成就之一。图形学渗透到我们生活的方方面面，从可以利用电子表格轻松地创建能够看到数据的图表，到可以通过图形学提供各种动画，特殊效果来增强娱乐性，再到借助图形更直观地展示和理解科学原理等。这些重要作用源于计算机系统中图形硬件与软件的不断进步。正是这些改进和提高，计算机图形学已成为不再需要昂贵的计算机和帧缓存，也不再要求程序员掌握生成图像所需要的所有数学和算法知识的高端技术领域。计算机图形学已经演化成一门让图形程序员关注视觉效果、创造与用户有高度交互性场景的学科。我们认为，初级计算机图形学课程的重点在于指导学生如何利用计算机图形学知识建立与用户的有效交流，例如交互和动画技术。图形学相关的算法与数学方面的更多技术细节将在图形学的高级课程中介绍。

什么是计算机图形学

计算机图形学研究的是应用计算机产生图像的所有工作，不管图像是静态的还是动态的，可交互的还是固定的，用于电影的、视频的、屏幕显示的还是打印输出的。这种特质使得计算机图形学有着非常广泛的应用领域，包括创意、商业或科学领域的许多应用。图形学广泛的应用范围促使我们为所有这些不同的领域开发出各种有用的生成图像和操纵图像的工具，而大量图形工具与应用实例也意味着我们可以借此了解和学习图形学的许多知识。

目前大部分计算机图形学方面的书籍可分为两类。第一类是传统的图形学教科书，强调建模、绘制、视图变换方面的算法与技术。这些都是很重要的概念，但是过分强调图像生成过程容易忽略图像内容。第二类着眼于图像生成的各种应用，特别是商业与娱乐行业的应用。这类课程受限于应用领域的需求和局限性，无法拓展到应用中未提到的部分。在学习应用实例时会提到一些基本概念，但重点还是学习应用本身。

本书将算法与应用结合起来。我们不过分强调计算机图形学领域内的算法与技术细节，也不会专注于图像生成的应用，而是将计算机图形学视为对图像内容中的几何、外观和表示等属性编程，并将编程结果展示在图形输出与交互设备上，从而生成合成图像的一门学科。这种强调通过编程方法生成图像的方式意味着我们必须掌握一些基本概念，因为我们要学会用计算机可理解的方式表示图形与交互的概念。这种强调通过编程生成图像的方法可以让学生掌握整个图像生成的过程，因而对学生既有机遇又有挑战。

掌握图像生成过程并非最终目标，由于图像会向其观察者传达某种信息，因此图像本身也很重要。我们要考虑图像的视觉交流效果。因此，图形学专业人员所做的图像生成工作包括理解图像表达的内容、开发组成图像的几何对象表示方法，将这些对象在几何空间组织起来以符合图像所需的关系，定义和表示场景中的每个对象，指定观察场景的方法和指定将看到的场景显示在图形设备上的方法。编程方法有很多种，但当前实践应用中通常采用能支持必要建模的与能完成通过编程定义的场景绘制工作的图形API。目前有很多的图形API，但OpenGL可能是目前应用最广泛的图形API，它提供了一个很好的学习图形技术的平台。

除了建模、视图变换、场景外观属性表示之外，图形学专业人员还有另外两个重要任务。

由于静态图像携带的信息量不如动态图像多，因此，在场景中加入运动信息（即定义图像动画）很重要。由于观察者希望或需要改变图像的内容、观察方式或图像的计算方法，因此，为用户设计与场景的交互方式也是十分重要的。图形API也支持这些额外的任务。

什么是图形API

API即应用编程接口（Application Programming Interface），是可供程序员开发应用程序的工具集。API的工具面向应用领域的具体任务，可以让程序员使用该领域的概念开发应用程序，而无需了解计算机系统的细节。API的作用在于屏蔽了任意一个计算机系统的细节，可以让程序员开发的应用程序运行在许多系统上。因此，图形API就是允许程序员开发包含交互式计算机图形操作的应用而不需要关注图形操作细节或任务系统细节（比如窗口处理和交互）的工具集。

本书除了包含交互式计算机图形学的基本知识外，还将介绍OpenGL图形API，并给出一些例子帮助理解OpenGL提供的功能，帮助大家学习将图形学编程与自己的工作相结合的方法。与大多数API一样，OpenGL除了提供本书中提到的入门内容之外，还支持很多高级操作。如果你需要进一步了解，请参考资源网站http://www.opengl.org/。

为什么需要计算机图形学

计算机图形学包含很多方面，因此，需要计算机图形学有很多原因。计算机图形学的一些最突出的应用是为许多用途创建图像，例如科学用途（将科学成果通过可视化的方式展示和解释给公众），娱乐用途（电影、视频游戏和特效），创意或艺术创作（艺术品、交互式影像装置），商业用途（广告、通信、产品设计），或日常交流（天气预报动画展示、图形资讯）等。本书提到的方法都是这些应用的基础，虽然有些应用可能涉及一些简单API编程无法达到的图像特殊效果或真实感效果。

在上文提到的所有应用领域，甚至更多领域中，计算机图形学在问题求解方面扮演非常重要的角色。问题求解是人们日常生活经常遇到的过程，因而，计算机图形学几乎在所有的领域都扮演着重要的角色，如图1所示。该图描述了问题求解过程：

- 提出问题
- 通过建模表达问题，使问题更抽象地表达
- 提出用几何模型表示问题的方法
- 由几何模型生成图像，将问题可视化
- 根据生成的图像理解问题或模型，从中思考解决方法

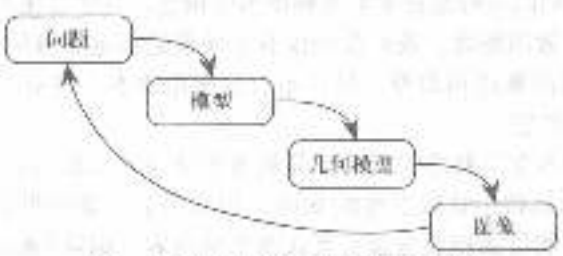


图1 图形学在问题求解过程中的作用

表示问题的图像可由多种方法生成。一种经典的简单方法是采用草图来表示讨论问题的

图像，和同事作非正式地交流。（有个民间笑话称，餐馆非常不希望见到一群科学家或数学家来吃饭，因为他们会在桌布上画草图！）图像也可采用计算机图形学来生成，特别是当需要与大量观众分享想法的时候很有用。如果根据问题生成模型，这些图像可以是动画或者能作交互式显示，这比用简单草图更能将问题表达清楚。然后，问题解决者或者观众可以利用图像进一步了解模型，从而进一步了解问题，使问题不断细化，创建更为复杂的模型，这个过程可以重复进行。

这个过程是第9章讨论图形学解决科学领域问题的基础，可应用到更广泛的领域。计算机图形学将大脑的视觉功能和智慧有效地应用到问题求解过程中，给我们提供了思考世界的强有力的工具。用俄勒冈州立大学的Mike Bailey先生的话来说，如同工具板手增加了我们的手部力量，计算机图形学给我们提供了“脑板手”来增强我们思考的能力。

本书概况

本书作为计算机图形学初级教材，要求学生具有较好的编程背景（大致相当于学生已经修完了一年的程序设计课程）。由于C语言是OpenGL应用中最常用的语言，因此本书的例子采用C语言编程。但本课程也可采用C++语言和其他面向对象语言。与传统图形学教材不同，本书不要求读者事先具备计算机图形学方面的知识，也不要求具备大量的数学基础和高级计算机科学知识。因为我们注重图形学编程而非算法和技巧，所以，本书不会涉及很多数据结构和其他算法技巧。与传统图形学教材不同，本书可作为计算机系学生早期阶段的计算机图形学教材，也可作为具有熟练编程经验的学生的自学教材。特别是本书可作为社区大学等的[⊖]计算机图形学教材。

本书中的许多例子来自科学领域，其中有一章介绍计算机图形学在科学和数学领域的不同应用。因此，本书可作为计算科学或需要与其他科学领域交叉的计算机科学课程教材，特别适合旨在为理科学生提供计算科学或科学可视化背景知识的课程教材。我们也曾想过在本书题目中加入可视化这个词，但后来考虑到这个词适合用在主要介绍科学而图形学为辅助内容的教材中，并不适合作为本书的题目。因为我们的观点刚好相反，认为科学可视化是计算机图形学的一种应用。

本书按照计算机图形学的传统顺序：投影处理、视图变换、建模、绘制、光照、着色处理等学科内容来组织。这些要素都纳入场景图之中，场景图可以很好地帮助程序员组织场景。我们还强调了产生图像的图形处理流水线。除了基本的图像处理技术，本书还着重介绍了如何采用计算机图形学来解决实际问题，及如何更有效地将结果展示给观察者的方法。本书介绍了计算机图形学中一系列的基本概念和技术，并说明OpenGL API如何提供实现这些概念和技术的图形学工具。我们的目标是使学生理解图形学概念，并学会使用图形API来实现图形学操作并为观察者创造有效的图像。我们深知仅有本课程是远远不够的，因此，我们会在介绍实现技术时提到涉及到的算法和原理，但并不会像理论课本一样对其深入展开。如有必要，将由你的导师为你深入讲解。

我们尽量将本书的各章节顺序安排为计算机图形学的入门教材应采用的顺序，有时候一章中提到的内容有可能依赖于该章之前的知识。如果在某一章中明显提到其他章节的内容，我们会做相应的说明。假设课程要包含交互式图形学内容，则第7章就是介绍交互技术（第7章数学章节可列为参考资料）。第10章图形流水线也与交互技术有关。除此之外，其他章节都

⊖ 社区大学类似于我国的民办高等职业技术学院。——译者注

是相对独立的，读者可按兴趣选读。

本书重点在于介绍采用图形API的编程技术，主要使用OpenGL API计算机编程来说明计算机图形学的基本概念。每一章讲述图形学的一个主题，并讨论其在OpenGL中的处理方式。除了用OpenGL作为例子讲述API的用法之外，有时也使用其他图形API。我们对图形学的基础算法和技术进行简要介绍，使大家能够理解图形学编程中的问题。这一点与大多数计算机图形学课本不同，在那些课本里面通常会花很多笔墨讲述算法和技术的细节。对于希望更深入了解这些知识的读者，我们建议选修高级计算机图形学课程。我们相信，基于API的图形学编程经验能够帮助读者更好地理解这些算法和技术的重要性，并可以在实际工作中更熟练地应用这些算法，没有相关算法背景知识会感到困难。

本书包含其他初级教科书中没有的几个特点。这些特点很好地满足了当前计算机图形学编程实践的需要。本书着重讨论三维图形技术，几乎完全不讨论二维技术。而传统图形学课程通常从二维开始再到三维，因为从二维开始掌握算法和技术相对比较容易，但直接从三维概念开始更为便利，因为二维图形学可视为三维图形学的特例，例如在X-Y平面上作三维建模。

建模是计算机图形学领域的基础，可用多种方法在图形显示器上为物体建模。由于OpenGL和大多数图形API支持多边形建模，本书也从标准的多边形建模开始。书中所讨论的建模着重讲述将场景图作为创建图形场景的基础工具。场景图的概念使学生可以设计变换、几何与复杂图元的外观属性，即使图形API本身并不直接支持场景图，也能方便地用清晰的代码来实现。这一点对层次式建模尤为重要，同时提供了统一的建模方法及许多有用的应用，使得读者能更多地关注场景，以管理运动和动画。

本书的一个重要特点在于强调采用计算机图形学创建有效的视觉交流。这个思想的提出是基于下列认识：计算机图形学在深入理解复杂问题并与他人交流方面的关键作用已被科学家群体和公众理解和认识。而这一点通常为计算机图形学教材所忽略。当然，我们希望大部分教师能在课程中提到这点。由于在图形建模、视图变换、颜色和交互中都应考虑有效交流的问题，因此本书的许多章节都会提到这个问题。相信关于这个主题的系统讨论对学生在以后的职业生涯中更有效地应用计算机图形学十分有用，无论他们从事计算机技术领域，还是从事计算机图形学的特殊应用领域的工作。

本书还强调创建交互显示的技术。大多数计算机图形学教科书都会提到交互和交互图形的创建。这一直以来是比较难于实现的领域，因为它涉及编写或应用一些特殊设备的驱动程序的技术。但随着OpenGL和其他图形API的日益普及，这项技术变得越来越普通了。由于我们非常关注图像与观众的有效交流，因而有必要了解与图像交互交流信息的重要作用。本文对交互的讨论包括事件驱动编程、OpenGL中使用的事件和回调函数，也包括交互在创建有效交流中的作用。我们在具体任务中观察交互的作用，而非仅学习交互技术，因此，应该将交互技术与本书所有内容结合起来学习。

可能你会注意到本书所举的例子比大多数计算机图形学教材的例子相对简单，这是有意如此设计的。我们认为既然本书要学习与OpenGL难度相当的图形生成方法，那么所举的例子也应该是这个难度。希望这些“自制的”例子比较有用，也希望读者能创建更好的例子。

本书介绍图形学原理而不讨论实现这些原理的算法和数学细节，这与大多数计算机图形学教科书有所不同。对希望更深入了解这门课程的读者，我们建议您再选读另一门图形学高级课程。我们相信，有了基于API图形学编程背景，学生将对计算机图形学算法与技术有更深入的理解，也有助于学生更熟练地应用图形算法和技术。

小结

读完本书后，相信您将具有以下方面的技能：

- 理解图形系统中图形信息的表示方法，使用图形系统编码生成图像
- 理解在程序中怎样组织图形信息，应用图形API生成图像
- 理解怎样使用图形系统中的事件信息生成交互图形显示
- 理解生成能与观察者有效交流的图像的相关知识
- 掌握用OpenGL API生成有效图像的技巧

我们希望读者将学到的原理和技巧应用到计算机图形学起重要作用的领域，并理解其在这些领域中的作用。本书主要将这些原理和技巧应用在科学领域（它们在很多其他领域也有广泛的应用）。如果有人采用本书为非科学或工程学科的学生开设课程，本书作者听到这一消息时将会十分高兴。

致谢

本书的写作最早得到美国国家科学基金会课题的支持，课题编号为DUE-9950121。本书提到的观点、发现、研究成果、结论与推荐意见均是作者本人提出的，与美国国家科学基金会无关。作者同时非常感谢加州州立大学Stanislaus分校（California State University Stanislaus）给我学术假期的支持，感谢圣地亚哥超级计算机中心（SDSC）的支持，特别是Mike Bailey（现在俄勒冈州立大学任教）在学术假期接待了我，使本书写作计划得以启动，也感谢他提供了大量可视化和科学方面的示例。圣地亚哥州立大学的Kris Stewart，沃福德学院的Angela Shiflet，SDSC的Rozeanne Steckler和Kim Baldridge，CSU Stanislaus的Ian Littlewood等学者为图形学在科学领域章节中提供了许多重要帮助。Grinnell学院的Sam Rebelsky提供了宝贵意见。Botswana大学的Sampson Asare，Swaziland大学的Petros Mashwama，São Paolo大学的Marcelo Zuffo允许作者使用研讨会文件的原稿。作者的几位在CSU Stanislaus和圣地亚哥州立大学的学生提供了重要例子并审读本书的有关章节。尤其是Ken Brown为本书原稿提出了许多有用的图形和概念。同时，还感谢我的学生Mike Dibley，Ben Endington，Jordan Maynard和Virginia Muncy为提供本书中的例子所做出的贡献。

感谢本书前期的审稿人。他们在许多方面，从基本概念到具体建议到校正错误提供了帮助，本书的成功完成源于我的学生、同事以及审稿人的共同努力。由于作者本人的学识有限，本书中难免有疏漏和不当之处，敬请读者不吝赐教。

Steve Cunningham
Coralville, Iowa

TP391.41/2133
2008

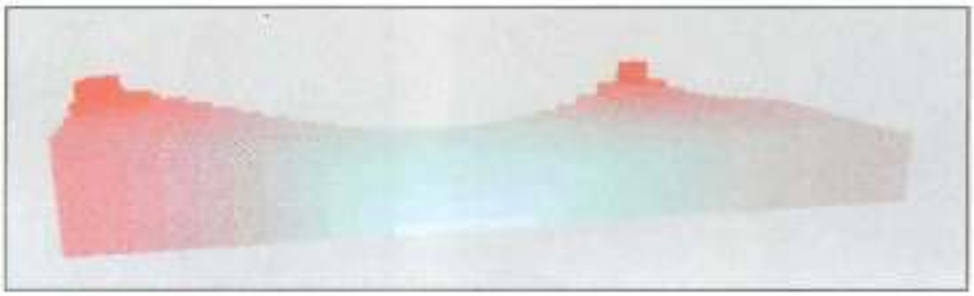


图0-7和图9-2 金属长条的热量分布

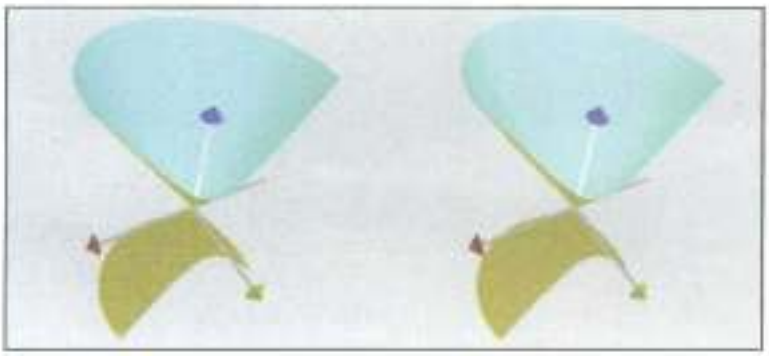


图1-15 两只眼睛分别看到的立体图像

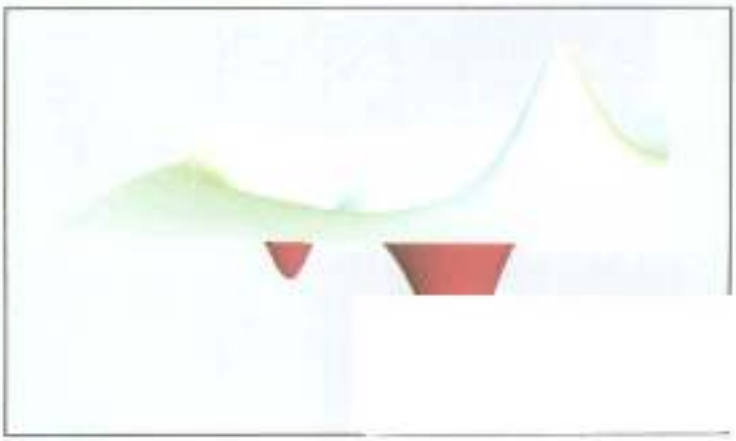
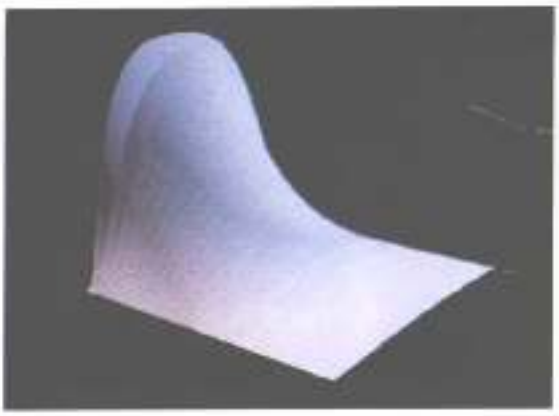


图2-18 用三个光源显示等线形体曲面的图像



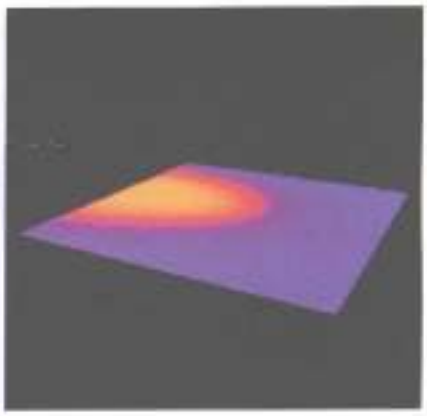


图2-20 t和1/3解决方案中温度随着时间改变



图2-29 场景图描述的一个场景



图2-33 效果与图2-29相同的场景，但是视点紧接在立方机里面





图5-1 从RGB立方体的白顶点（左）和黑顶点（右）看的特征



图5-2 蓝色与蓝色间的六种颜色序列

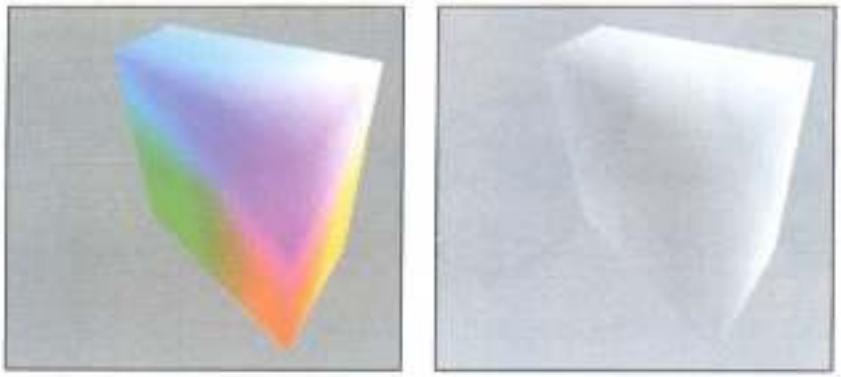


图5-3 以彩色（左）和灰度图（右）形式表现的RGB立方体的等亮度图



图5-4 HSV彩色模型图

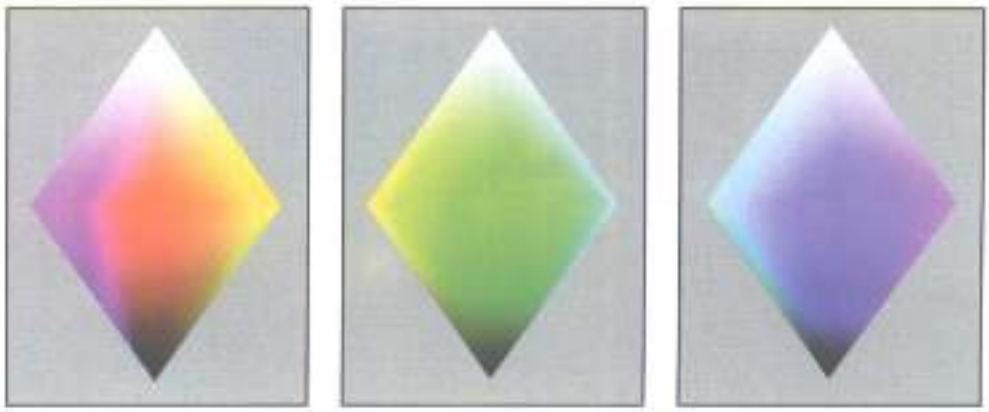


图5-5 HLS双圆锥模型，从三原色红（左）、绿（中）和蓝（右）边覆盖120度的圆锥表面

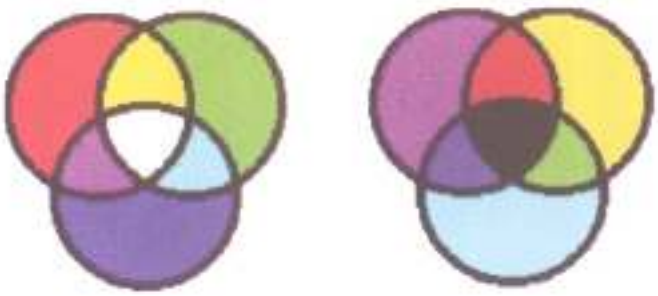


图5-6 发射色（左）和吸收色（右）

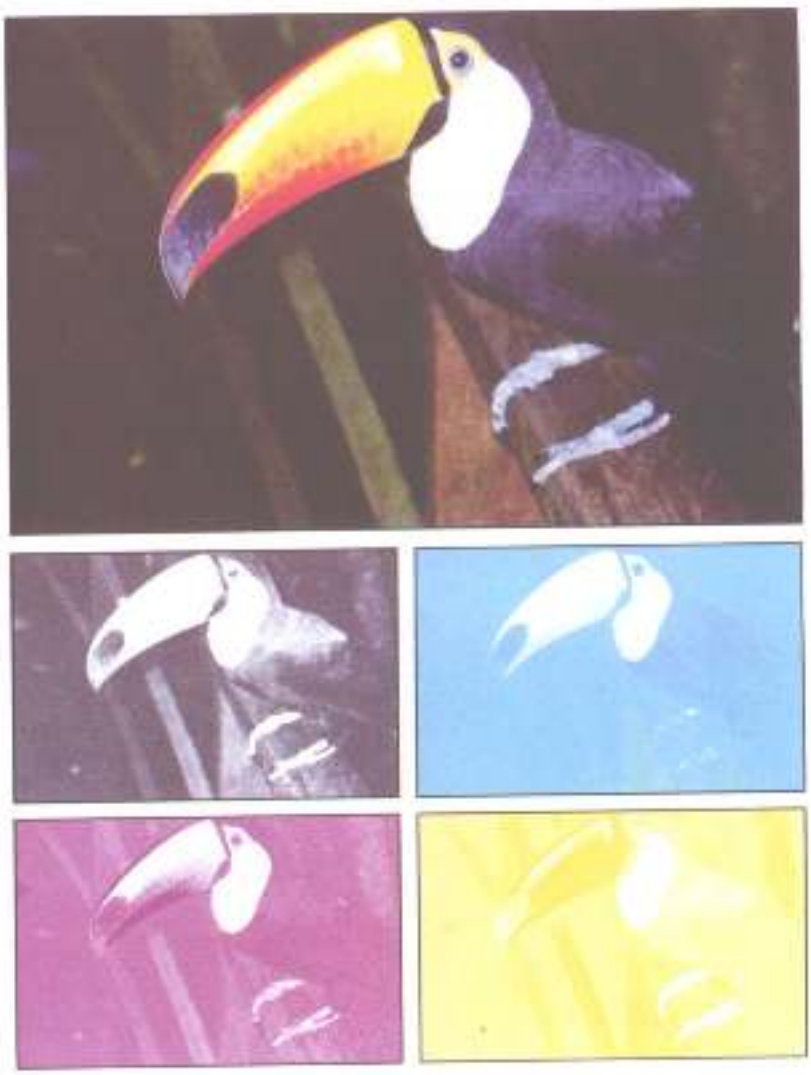


图5-7 印刷时的分色图片

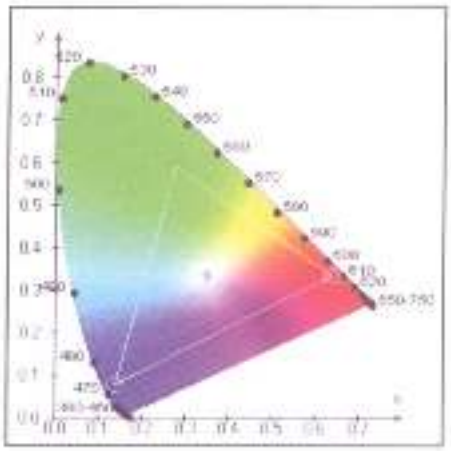


图5-9 CIE颜色空间



图5-11和图7-7 有一个强化色的图像

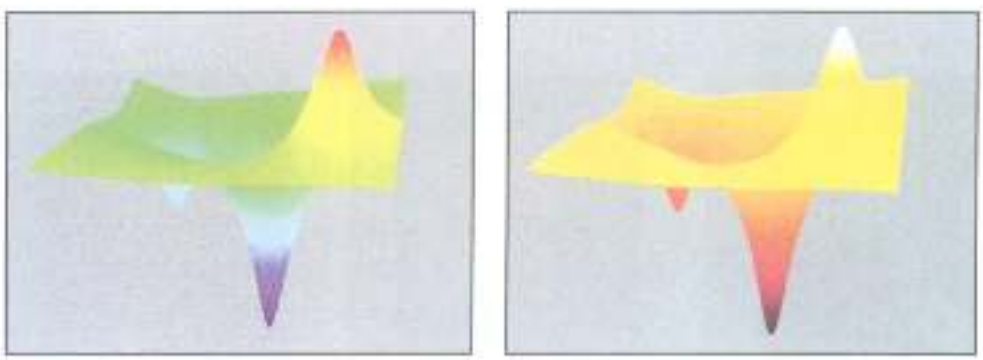


图5-14 静电势绘表面模型，用“彩虹”颜色渐变加强特定值（左）和统一亮度分布图（右）

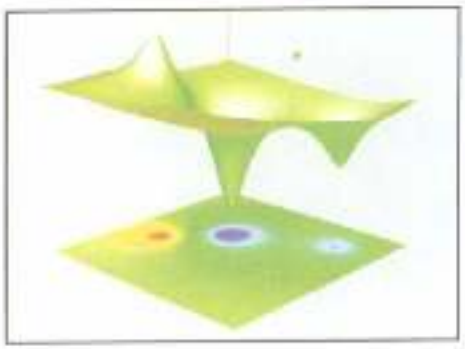


图5-15和图9-5　带光环表面的伪彩色表面

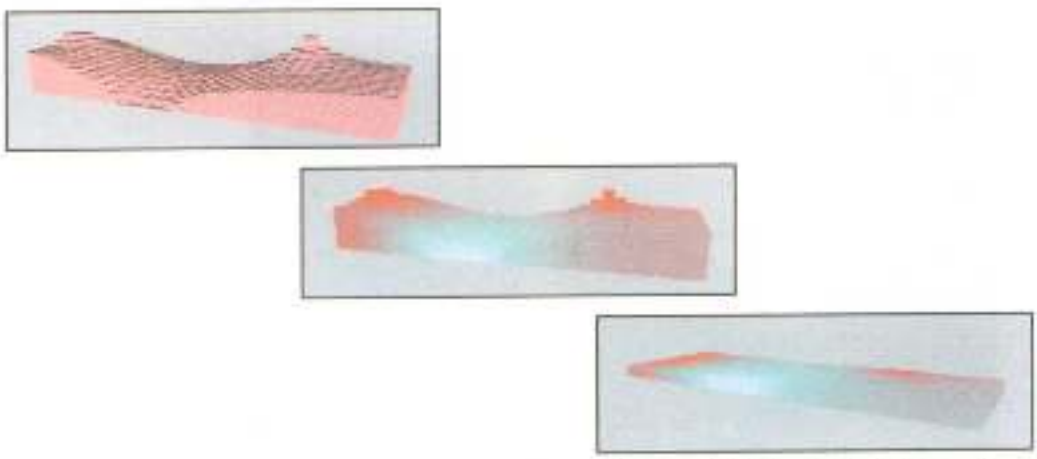


图5-16　长条上的深度这同一信息的二种编码方式：几何编码（左上），颜色编码（右下），二者兼用（中）



图5-19　坐标平面部分透明（左），同一坐标平面完全透明，但有相同的α值（中），同一坐标平面带经调节的α值（右）



图5-20 平面一分为四后的半透明效果，从后向前绘制



图6-5 在太阳和地球上放置不同光源的场景

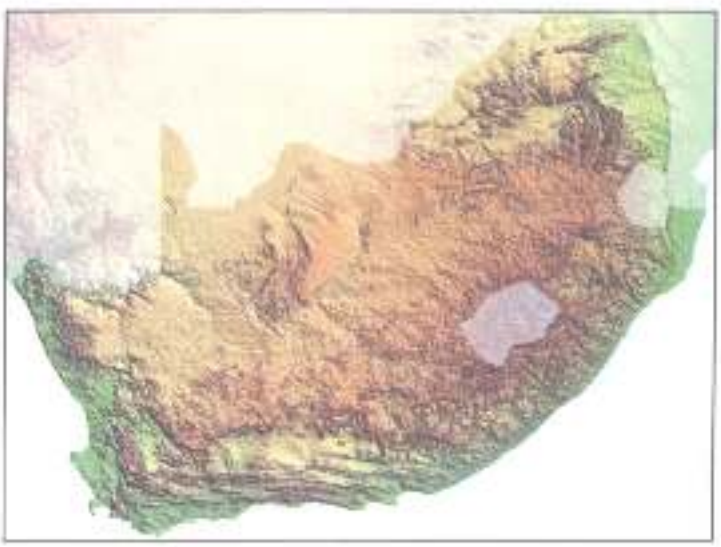


图6-6 经过光照处理的南非伪色彩地图



图6-12 用辐射度方法渲染的场景



图6-13 用光子映射模型渲染的场景



图6-14 用二种不同颜色的光滑规整的白色立方体



图8-10 使用多纹理

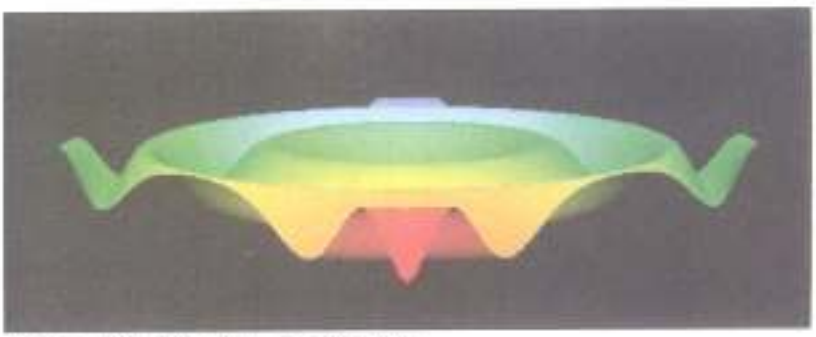


图8-13 代数曲面的色度－深度彩色图



练习题8.4









练习题8.5



图9-15　洞穴的漫画图片（包含漫画参考卡和激光扫描点）



图9-17　洞穴壁画（马和油灯）

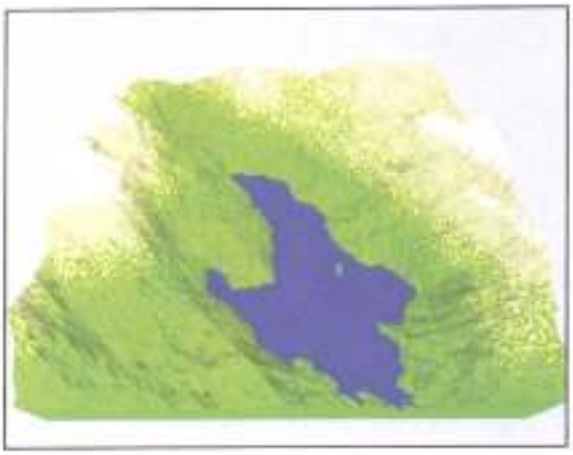


图9-18 采用着色处理和水的透明度显示“伯分形”地形图

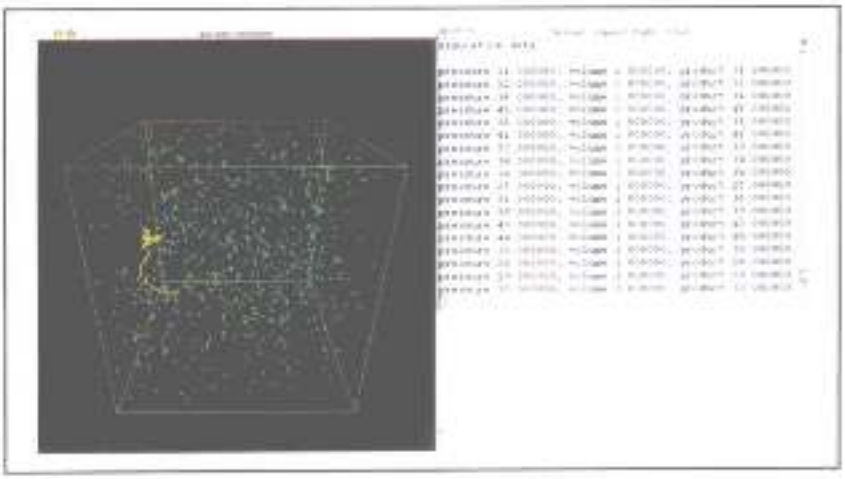


图9-19 把气体显示为固定空间中的分子，包括仿真打印结果

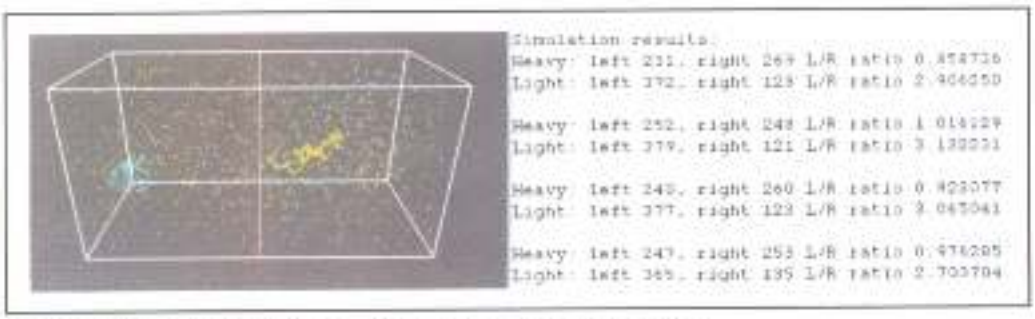


图9-20 直接选定物体的扩散仿真的显示（包括模拟中的数据输出）

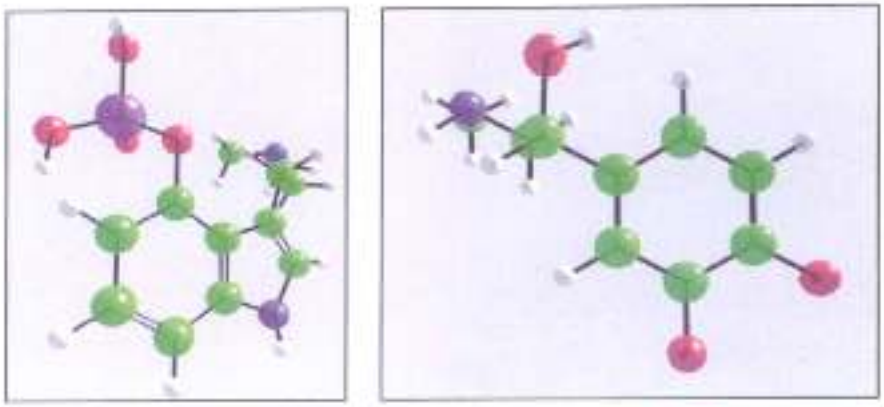


图9-21 显示psilocybin.mol（左）和adrenaline.pdb（右）

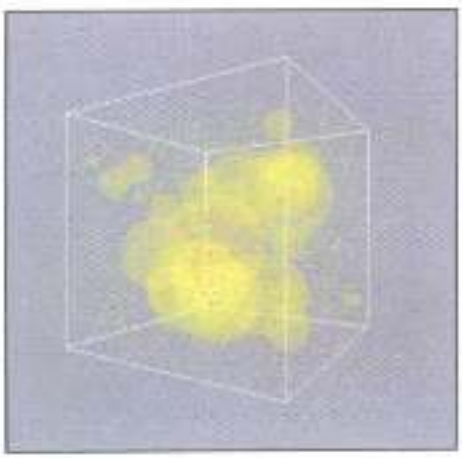


图9-23 复杂体的蒙特卡罗估计

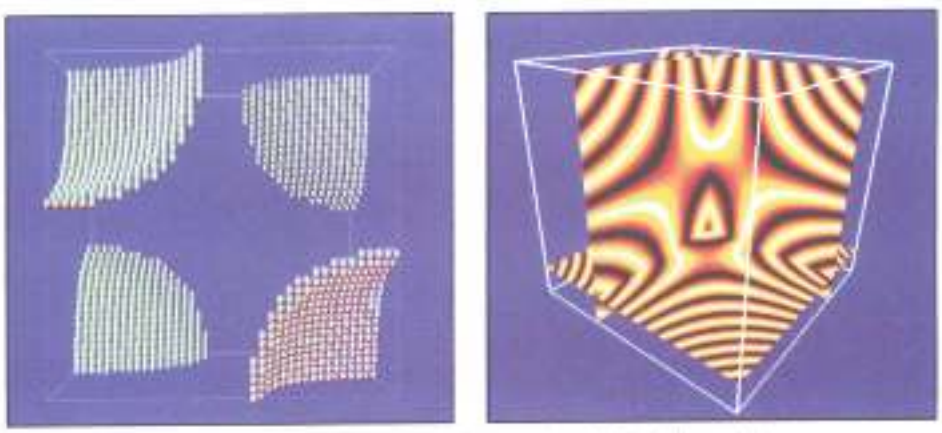


图9-24 用球（左）和函数值的横截面（右）定位表面的隐式曲面近似

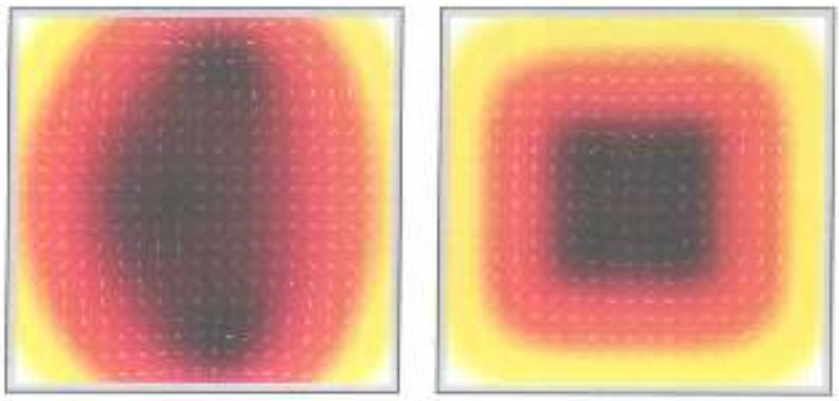


图9-25 同一主题的两种可视化：基于一维复变量的复函数（左）与微分方程的方向向量（右）

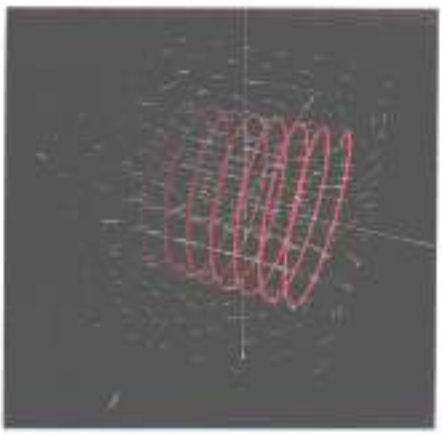


图9-26 通电导线周围的磁场

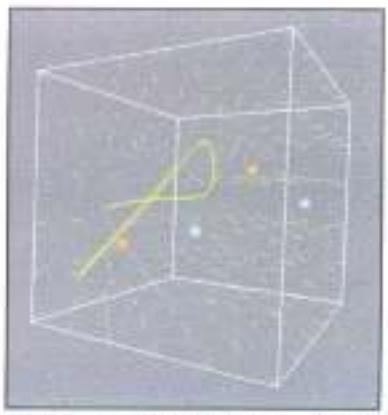


图9-27 以三维向量场模型三维库仑定律

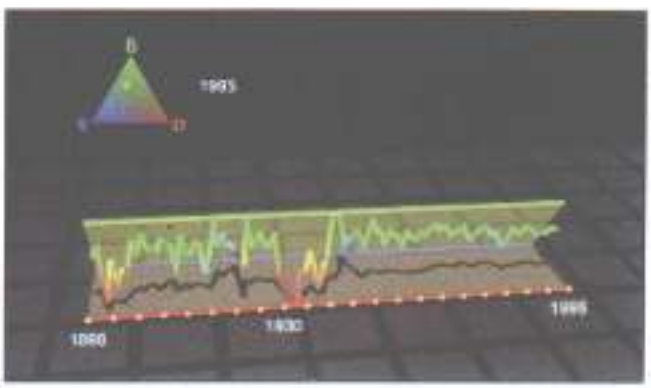


图9-28 经济数据展示



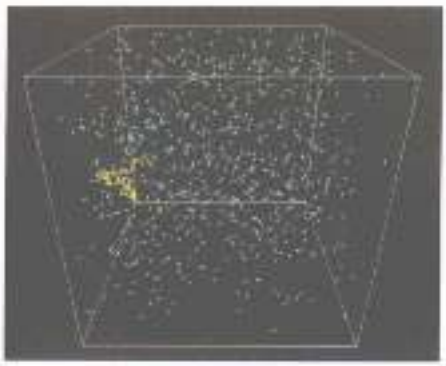


图11-7 两种运动物体的轨迹

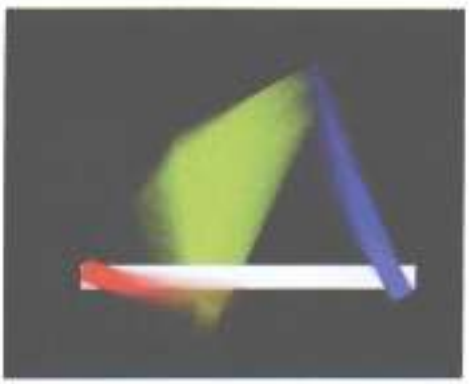


图11-8 一个运动的机械装置，其中一个部件固定，其他部分带有运动模糊效果



图12-2 雾化的立方体（其中一个面具有纹理贴图）

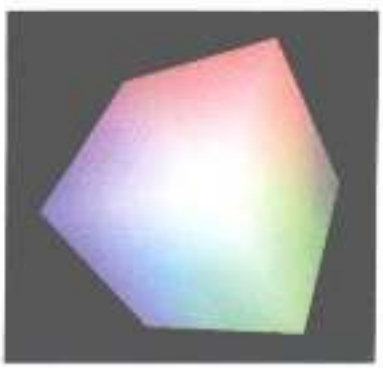


图12-7 使用顶点数组和法向数组绘制的立方体



图14-4 用POVRay光线跟踪程序绘制的图像



图14-5 客户单云模型的体可视化



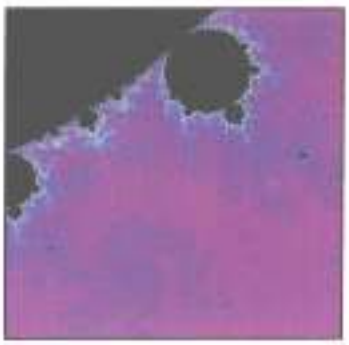


图14-12 完整的mandelbrot集图（左）和细节（右）



图14-13 针对一个固定c值的茹利亚集图

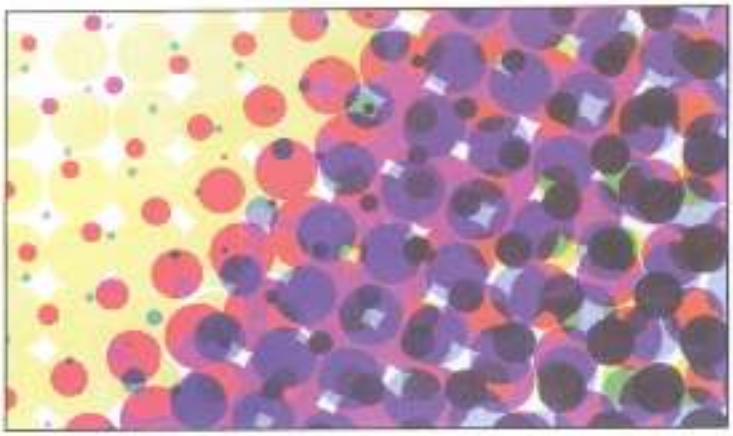


图15-1 放大的彩色图像中的C、M、Y和K色网板

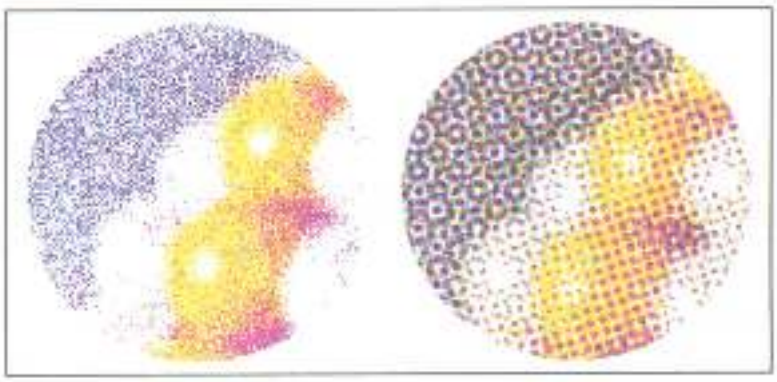


图15-2 随机筛选网板（左）和固定角度网板（右）的比较



图15-4 色度－灰度图像显示



图15-6 彩色立体图的例子，当通过红/蓝或红/绿眼镜观察彩色图时，可以看到三维彩色图像

目 录

出版者寄语
译者序
前言

第0章 导论……1
0.1 视觉交流与计算机图形学……1
0.2 视觉交流的基本概念……2
0.2.1 使用合适的信息表示方式……2
0.2.2 图像应突出重点……2
0.2.3 使用合适的信息展示级别……2
0.2.4 采用合适的信息格式……3
0.2.5 注意图像显示的准确性……3
0.2.6 理解并尊重观众的文化背景……3
0.2.7 使交互成为用户熟悉的高效操作……4
0.3 三维几何和几何流水线……5
0.3.1 场景与视图……5
0.3.2 三维模型坐标系……5
0.3.3 三维世界坐标系……5
0.3.4 三维眼坐标系……6
0.3.5 投影……6
0.3.6 裁剪……7
0.3.7 选择透视投影或正交投影……7
0.3.8 三维眼坐标……8
0.3.9 二维屏幕坐标……8
0.4 外观属性……8
0.4.1 颜色……9
0.4.2 纹理……9
0.4.3 深度缓存……9
0.5 观察过程……9
0.6 图形卡……10
0.7 一个简单的OpenGL程序……10
0.7.1 OpenGL程序main()函数结构……15
0.7.2 模型空间……15
0.7.3 模型变换……15
0.7.4 三维世界空间……16
0.7.5 视图变换……16
0.7.6 三维眼空间……16
0.7.7 投影操作……16
0.7.8 三维眼空间……17
0.7.9 二维屏幕空间……17
0.7.10 科学问题编程……17
0.7.11 外观属性……17
0.7.12 从另一角度分析程序……17
0.8 OpenGL扩展……18
0.9 小结……19
0.10 本章的OpenGL术语表……19
0.11 思考题……20
0.12 练习题……21
0.13 实验题……21
第1章 视图变换和投影……22
1.1 简介……22
1.2 视图变换的基本模型……24
1.3 定义……24
1.3.1 建立视图环境……25
1.3.2 定义投影……25
1.3.3 视域体……26
1.3.4 正交投影……27
1.3.5 透视投影……27
1.3.6 透视投影的计算……28
1.3.7 视域体裁剪……29
1.3.8 定义窗口和视口……30
1.4 管理视图的其他方面……32
1.4.1 隐藏面……32
1.4.2 双缓存……33
1.5 立体视图……33
1.6 视图变换与视觉交流……34
1.7 在OpenGL中实现视图变换和投影……34
1.7.1 定义窗口和视口……35
1.7.2 改变窗口的形状……35
1.7.3 设置视图变换的环境……36

1.7.4 定义透视投影 ……37
1.7.5 定义正交投影 ……37
1.7.6 隐藏面的处理 ……37
1.7.7 设置双缓存 ……38
1.8 实现立体视图 ……38
1.9 小结 ……39
1.10 本章的OpenGL术语表 ……39
1.11 思考题 ……40
1.12 练习题 ……40
1.13 实验题 ……41
第2章 变换导课 ……43
2.1 简单几何建模 ……44
2.2 定义 ……44
2.3 例子 ……46
2.3.1 单点和多点 ……46
2.3.2 线段 ……46
2.3.3 线段序列 ……46
2.3.4 三角形 ……47
2.3.5 三角形序列 ……47
2.3.6 四边形 ……47
2.3.7 四边形序列 ……48
2.3.8 通用多边形 ……49
2.3.9 多面体 ……50
2.3.10 走样和反走样 ……50
2.3.11 法线 ……50
2.3.12 裁剪 ……51
2.3.13 建模的数据结构 ……52
2.3.14 曲面的建模 ……53
2.3.15 其他的图形对象建 ……54
2.3.16 建模行为 ……55
2.3.17 建议 ……55
2.4 变换和建模 ……55
2.5 定义 ……56
2.5.1 变换 ……56
2.5.2 复合变换 ……58
2.5.3 使用变换栈 ……59
2.5.4 编译几何体 ……60
2.6 一个例子 ……60
2.7 建议 ……62
2.8 建模软件交流 ……62
2.9 认识形体的含义 ……62
2.10 维度 ……63
2.11 更高维度 ……65
2.12 图例和标签 ……66
2.13 精确度 ……67
2.14 场景图和建模语 ……67
2.15 场景图的概要 ……68
2.15.1 场景图中的裁剪 ……69
2.15.2 用场景图建模的例子 ……69
2.16 视图变换 ……71
2.17 场景图和深度测试 ……73
2.18 用建模图写代码 ……73
2.18.1 两个场景图的代码实例 ……75
2.18.2 使用标准的对象生成
更加复杂的场景 ……77
2.19 小结 ……77
2.20 思考题 ……77
2.21 练习题 ……78
2.22 实验题 ……79
2.23 大型作业 ……80
第3章 在OpenGL中实现建模 ……81
3.1 指定几何体的OpenGL模型 ……81
3.1.1 点和多点模型 ……82
3.1.2 直线段 ……82
3.1.3 线段序列 ……83
3.1.4 封闭线段 ……83
3.1.5 三角形 ……83
3.1.6 三角形序列 ……83
3.1.7 四边形 ……84
3.1.8 四边形条带 ……85
3.1.9 普通多边形 ……85
3.1.10 顶点数组 ……86
3.1.11 反走样 ……86
3.1.12 将在很多例子中使用的立方体 ……86
3.1.13 定义裁剪平面 ……87
3.2 OpenGL工具中的附加对象 ……88
3.2.1 GLU二次曲面对象 ……88
3.2.2 GLU圆柱体 ……88
3.2.3 GLU圆盘 ……88
3.2.4 GLU球体 ……89

3.2.5 GLUT对象……89
3.2.6 例子……89
3.3 OpenGL中的变换……90
3.4 图例和标签……92
3.5 变换的代码实例……93
3.5.1 简单变换……93
3.5.2 变换线……94
3.5.3 旋转视点变换……95
3.5.4 生成显示列表……96
3.6 到视点的距离……97
3.7 小结……97
3.8 本章的OpenGL术语表……98
3.9 思考题……100
3.10 练习题……100
3.11 实验题……101
3.12 大型作业……102
第4章 建模的数学基础……103
4.1 坐标系……103
4.2 四象限和八象限……104
4.3 点、直线和直线段……104
4.4 直线段、射线、参数化曲线和曲面……105
4.5 点到直线的距离……105
4.6 向量……105
4.7 向量点积和叉积……106
4.8 反射向量……107
4.9 变换……108
4.10 平面和半空间……109
4.11 点到平面的距离……110
4.12 多边形和凸面……110
4.13 多面体……111
4.14 极坐标、柱面坐标和球面坐标……111
4.15 降维投影……112
4.16 高维空间……114
4.17 小结……114
4.18 思考题……114
4.19 练习题……114
4.20 实验题……114
第5章 颜色及其混合……116
5.1 简介……116
5.2 原理……117
5.2.1 设置几何物体的颜色……117
5.2.2 RGB立方体……117
5.2.3 亮度和色调……118
5.2.4 其他颜色模型……119
5.2.5 颜色深度……120
5.2.6 色调……121
5.2.7 颜色混合与α通道……121
5.2.8 使用混合达到透明效果……122
5.2.9 索引颜色……122
5.3 颜色和视觉交流……123
5.3.1 强调色……123
5.3.2 背景色……125
5.3.3 自然色……124
5.3.4 伪彩色和颜色渐变……124
5.3.5 创建颜色渐变……124
5.3.6 颜色渐变的使用……125
5.3.7 比较形状和颜色编码……126
5.3.8 颜色的文化背景……126
5.4 例子……127
5.5 OpenGL中的颜色……128
5.5.1 颜色定义……128
5.5.2 使用混合……128
5.6 代码实例……129
5.6.1 带有全色谱的模型……129
5.6.2 HSV圆锥……129
5.6.3 HLS双圆锥……130
5.6.4 带半透明面的对象……131
5.6.5 索引颜色……131
5.6.6 OpenGL中的颜色渐变……132
5.7 小结……132
5.8 本章的OpenGL术语表……132
5.9 思考题……132
5.10 练习题……133
5.11 实验题……134
5.12 大型作业……134
第6章 光照处理和着色处理……135
6.1 光照处理……135
6.1.1 环境光、漫反射光和镜面反射光……136
6.1.2 表面法向……138
6.2 材质……139

6.3 光源属性 ……139
6.3.1 光源颜色 ……140
6.3.2 位置光 ……140
6.3.3 聚光灯 ……140
6.3.4 光线衰减 ……140
6.3.5 方向光 ……140
6.4 放置与移动光源 ……141
6.5 用光照实现特效 ……141
6.6 场景图中的光源 ……141
6.7 着色处理 ……141
6.8 在程序交流中考虑着色处理 ……142
6.9 定义 ……142
6.10 Flat着色处理和平滑着色处理的例子…143
6.11 计算每个顶点的法向 ……144
6.11.1 平均多边形法向 ……144
6.11.2 法向的解析计算 ……144
6.12 其他着色处理模型 ……145
6.13 各向异性着色处理 ……146
6.14 全局光照 ……146
6.14.1 辐射度方法 ……147
6.14.2 光子映射 ……147
6.15 局部光照和OpenGL ……148
6.15.1 指定和定义光源 ……148
6.15.2 选择性地使用光源 ……150
6.15.3 定义材质 ……150
6.15.4 使用GLU二次曲面物体 ……151
6.15.5 例子：把三原色光源应用于白色表面 ……151
6.15.6 示例代码 ……151
6.15.7 着色处理的例子 ……152
6.16 建议 ……154
6.17 小结 ……154
6.18 本章的OpenGL术语表 ……154
6.19 思考题 ……155
6.20 练习题 ……155
6.21 实验题 ……156
6.22 大型作业 ……157
第7章 事件和交互式编程 ……158
7.1 定义 ……158
7.2 事件的例子 ……159
7.3 交互的方式和方法 ……160
7.4 对象选择 ……161
7.5 交互和视觉交流 ……161
7.6 事件和场景图 ……162
7.7 建议 ……162
7.8 OpenGL中的事件 ……163
7.9 回调函数的注册 ……163
7.10 实现细节 ……165
7.11 代码实例 ……167
7.11.1 空闲事件回调函数 ……168
7.11.2 定时器事件回调函数 ……168
7.11.3 键盘回调函数 ……169
7.11.4 菜单回调函数 ……170
7.11.5 鼠标移动的鼠标回调函数 ……171
7.11.6 对象拾取的鼠标回调函数 ……171
7.12 拾取的实现细节 ……173
7.12.1 定义 ……173
7.12.2 拾取操作的实现方法 ……174
7.12.3 拾取矩阵 ……176
7.12.4 使用后颜色缓存做拾取 ……176
7.12.5 一个选择操作的例子 ……177
7.12.6 拾取小结 ……179
7.13 MUI工具 ……179
7.13.1 引言 ……179
7.13.2 应用MUI的功能 ……180
7.13.3 MUI用户界面对象 ……181
7.13.4 一个例子 ……183
7.14 在Windows系统中安装MUI ……185
7.15 建议 ……185
7.16 小结 ……185
7.17 本章的OpenGL术语表 ……186
7.18 思考题 ……187
7.19 练习题 ……188
7.20 实验题 ……188
7.21 大型作业 ……189
第8章 纹理映射 ……190
8.1 简介 ……190
8.2 定义 ……191
8.2.1 1D纹理图 ……191
8.2.2 2D纹理图 ……191

8.2.3 3D纹理图……192
8.2.4 纹理坐标与空间坐标的对应关系…192
8.2.5 对象颜色与纹理图颜色的关系……192
8.2.6 纹理图的其他含义……192
8.2.7 场景图中的纹理映射……193
8.3 创建纹理图……193
8.3.1 从图像创建纹理图……193
8.3.2 人工生成纹理图……194
8.3.3 噪声函数生成纹理图……194
8.4 纹理图中的插值操作……195
8.5 纹理映射和布告板技术……196
8.6 纹理图中包含多个纹理……196
8.7 纹理反走样……196
8.8 MIP映射……197
8.9 多纹理……197
8.10 OpenGL中的纹理映射……198
8.10.1 顶点与纹理点相关……198
8.10.2 从屏幕获取纹理……199
8.10.3 纹理环境……199
8.10.4 纹理参数……200
8.10.5 获取及定义纹理图……201
8.10.6 纹理坐标控制……202
8.10.7 纹理插值……202
8.10.8 纹理映射和GLU四边形……203
8.10.9 多纹理……203
8.11 例子……203
8.11.1 使用Chromadepth过程……204
8.11.2 使用2D纹理图在表面中加入信息…204
8.11.3 环境纹理图……204
8.12 建议……205
8.13 代码实例……205
8.13.1 1D颜色渐变……205
8.13.2 2D纹理例子……206
8.13.3 环境纹理图……207
8.13.4 使用多纹理……207
8.14 小结……208
8.15 本章的OpenGL术语表……208
8.16 思考题……210
8.17 练习题……210
8.18 实验题……211
8.19 大型作业……212
第9章 图形在科学计算领域中的应用…213
9.1 简介……213
9.2 例子……215
9.3 扩散……215
9.3.1 长条材料中的温度……215
9.3.2 疾病的传播……217
9.4 函数作图和应用……218
9.5 参数曲线与曲面……219
9.6 极限处理结果的图形对象……222
9.7 标量场……223
9.8 物体和行为仿真……224
9.8.1 气体定律和扩散原理……225
9.8.2 分子显示……226
9.8.3 科学仪器……227
9.8.4 蒙特卡罗建模过程……227
9.9 四维作图……228
9.9.1 体数据……228
9.9.2 向量场……229
9.10 高维作图……230
9.11 数据驱动图形……231
9.12 代码实例……232
9.12.1 扩散……232
9.12.2 函数作图……233
9.12.3 参数曲线与曲面……234
9.12.4 极限处理……235
9.12.5 标量场……235
9.12.6 物体及行为的表示……235
9.12.7 分子显示……236
9.12.8 蒙特卡罗建模……237
9.12.9 四维作图……237
9.12.10 高维作图……238
9.13 小结……239
9.14 思考题……239
9.15 练习题……239
9.16 实验题……240
9.17 大型作业……240
第10章 绘制与绘制流水线……242
10.1 引言……242
10.2 流水线……242

10.3 光栅化处理 ……244
10.4 OpenGL的绘制流水线 ……248
10.4.1 绘制流水线中的纹理映射 ……249
10.4.2 逐片段操作 ……249
10.4.3 OpenGL与可编程着色器 ……250
10.4.4 图形卡绘制流水线实现的实例 ……251
10.5 图形卡的部分三维视图变换操作 ……251
10.6 小结 ……252
10.7 本章的OpenGL术语表 ……252
10.8 思考题 ……252
10.9 练习题 ……253
10.10 实验题 ……253
第11章 动力学和动画 ……254
11.1 一个例子 ……255
11.2 动画的分类 ……256
11.2.1 过程动画 ……256
11.2.2 场景图中的动画 ……256
11.2.3 插值动画 ……257
11.2.4 基于帧的动画 ……258
11.2.5 一个插值例子 ……259
11.3 动画中的一些问题 ……260
11.3.1 帧速率 ……260
11.3.2 时间走样 ……260
11.3.3 动画制作 ……261
11.4 动画和视觉交流 ……261
11.5 在静止帧中表示运动信息 ……262
11.5.1 运动轨迹法 ……262
11.5.2 运动模糊法 ……263
11.6 一些有趣的观看动画的设备 ……263
11.7 建议 ……265
11.8 OpenGL的动画例子 ……265
11.8.1 在模型中移动物体 ……265
11.8.2 控制动画的时间 ……266
11.8.3 移动模型的部件 ……266
11.8.4 移动视点或模型的观察标架 ……267
11.8.5 场景的纹理插值 ……268
11.8.6 改变模型的特征 ……268
11.8.7 生成轨迹 ……269
11.8.8 使用累积缓存 ……270
11.8.9 创建数字视频 ……271
11.9 用OpenGL制作动画时应注意的一些要点 ……272
11.10 建议 ……272
11.11 本章的OpenGL术语表 ……272
11.12 思考题 ……273
11.13 练习题 ……273
11.14 实验题 ……273
11.15 大型作业 ……274
第12章 高级主题与技术 ……276
12.1 定义 ……276
12.2 技术 ……277
12.3 建模技术 ……277
12.3.1 减少可见多边形数量 ……277
12.3.2 巧妙运用纹理 ……278
12.3.3 减少光照计算 ……278
12.3.4 细节层次 ……278
12.3.5 雾化 ……280
12.3.6 开始距离和结束距离 ……280
12.3.7 雾化模式 ……280
12.3.8 雾密度 ……281
12.3.9 雾色 ……281
12.4 绘制技术 ……282
12.4.1 不使用硬件 ……282
12.4.2 使用硬件 ……282
12.4.3 多边形剔除 ……282
12.4.4 避免深度比较 ……283
12.4.5 从前到后绘制 ……284
12.4.6 二元空间划分 ……284
12.4.7 系统加速技术 ……285
12.5 碰撞检测 ……286
12.6 小结 ……287
12.7 本章的OpenGL术语表 ……287
12.8 思考题 ……288
12.9 练习题 ……288
12.10 实验题 ……289
12.11 大型作业 ……289
第13章 插值与样条建模 ……290
13.1 引言 ……290
13.1.1 插值 ……290
13.1.2 另一种Bézier样条的基本概念 ……293

13.1.3 另一种Bézier样条计算方法……293
13.1.4 扩展插值到更多控制点……293
13.2 样条曲面……295
13.2.1 扩展曲面片为曲面……295
13.2.2 生成曲面片法向……296
13.2.3 生成曲面片纹理坐标……296
13.2.4 另一种曲面片计算方法……296
13.3 其他类型的插值函数……297
13.4 OpenGL中的插值……297
13.4.1 使用求值器自动生成法向和纹理…298
13.4.2 其他技巧……299
13.5 定义……299
13.6 示例……300
13.6.1 样条曲线……300
13.6.2 样条曲面……301
13.7 小结……303
13.8 本章的OpenGL术语表……303
13.9 思考题……304
13.10 练习题……304
13.11 实验题……305
13.12 大型作业……305
第14章 非多边形图形技术……306
14.1 定义……306
14.2 光线投射……306
14.3 光线跟踪……308
14.4 体绘制……309
14.5 迭代函数系统……310
14.5.1 压缩映射……310
14.5.2 生成函数……311
14.6 芒德布罗集和茹利亚集……312
14.7 OpenGL支持的逐像素操作……313
14.8 小结……314
14.9 思考题……314
14.10 练习题……314
14.11 实验题……315
14.12 大型作业……315
第15章 硬拷贝……316
15.1 定义……316
15.2 选择输出媒介……316
15.2.1 数字图像……316
15.2.2 印刷……317
15.2.3 胶片……318
15.2.4 三维图像技术……319
15.2.5 三维对象成型技术……320
15.2.6 STL文件……321
15.2.7 视频……322
15.2.8 数字视频……323
15.3 支持硬拷贝的OpenGL技术……323
15.3.1 捕获输出窗口内容到文件……323
15.3.2 用OpenGL生成立体图……324
15.4 小结……325
15.5 本章的OpenGL术语表……325
15.6 思考题……325
15.7 实验题……325
参考文献和资源……327
附录……330
索引……335

第0章 导 论

本章介绍计算机图形学的基本概念，使读者能把握本书的内容框架。本章重点介绍三个关键领域，以使使读者了解本书相关内容的背景知识。

第一个关键领域是图形学在视觉交流中所起的作用。我们认为交流是学习和应用计算机图形学的最主要目的，因此，本书很多节讨论的内容都与如何对视觉交流进行有效支持有关。事实上，在后面关于科学领域的计算机图形学章节中（第9章），主题就是在科学领域生成有效交流的图像。开始学习计算机图形学时，我们提出一些基本的交流原则，这些原则是在生成计算机图形显示时必须时刻铭记的。

第二个关键领域讨论由三维几何流水线管理的三维几何变换和由绘制流水线管理的计算机图形物体外观属性。几何流水线显示了要生成图像所必须指定的关键信息，和要表示图像时图形系统应完成的计算过程。我们先介绍外观属性表示的几种方法，绘制流水线将在后面（第10章）再介绍。

第三个关键领域是OpenGL图形API在图形学程序中的使用方法。OpenGL API是本书采用的主要API。本章将介绍OpenGL的通用程序结构，并给出描述一个特定问题并生成带动画的图像的一个完整的程序实例。在这个例子中，你将看到如何在程序中定义几何流水线的信息和外观属性信息。在本章的练习中，你将有机会对程序作不同的改变，并观察改变后的不同效果。

0.1 视觉交流与计算机图形学

计算机图形学在与专家、专业团体、公众的信息交流方面已经取得了杰出的贡献，这与其在娱乐领域的应用不同（在娱乐方面，计算机图形学已很受重视），因为这里所指的信息交流是为了帮助人们深入理解复杂的问题。本书主要关注科学领域的信息交流，话题包括宇宙论，展示宇宙的基本结构；考古学和人类学，展示早期人类群体的组成和文化；生物学和化学，展示静电力和分子结构如何组成分子键；数学，理解高阶不稳定微分方程的特征；以及气象学，研究比如洋流温度或臭氧层厚度对气候的影响。

虽然视觉交流及相关的视觉词汇早已为艺术家、设计师和电影导演们所熟知，但其在科学领域的用途却是在1987年关于科学计算可视化[ViSC]报告中才被重点提出。该报告提到计算机图形学在帮助人脑从图像理解事物本质的特殊能力中的重要作用。报告引用了Richard Hamming在1962年的经典论断："计算的目的是洞察事物的本质，而不是获得数字"。这一论断在今天计算能生成揭示复杂问题更深层本质的图像的时代具有很强的实用性，因为图像比单纯数字具有更强的洞察力。如果我们把Hamming的论断借用于计算机图形学的话，那么，应用计算机图形学的目的是获得信息，而不是图像本身。

图像生成过程，尤其是使用高性能计算机和有效的图形API生成吸引人的、有趣的图像，是一个相对直接的、步骤清晰的过程，您将在本书的学习过程中体会到这点。计算机图形学的难点在于理解你所面对的问题，并提出描述问题的信息表示方法，从而创建出与观众交流的精确的图像。这一节将讲述这个任务，并讨论一些能启发思考的原理和例子。一开始就讨论这个问题的目的是希望提醒读者，图形学是用来与他人交流的，当通过计算生成图像时不要忘记交流这个目的。这一节中提到的一些技术在本书后面部分会有详细介绍，但它们并不

复杂。因此，在学会怎样使用这些技术之前应该对它们的用处有所了解。

本书在介绍一些技术时，考虑它们对视觉交流所起的作用，但这仅是各个主题的简单介绍而已。高水平的交流者总是不断寻求与特定观众交流特定信息的新方法，并为观众创造新的视觉词汇。我们并不会将本书当成视觉交流的完整教材，而是希望您可以多思考图像所表达的内容，以及怎样将内容传达给观众。

视觉交流还包括设计交互方式和方法，通过交互控制提供所需的视觉效果。运动、选择、
2 图形流程控制等交互方式，都会在呈现给用户的图形中表现出来，因此在后面的章节（第7章与第11章）中，将讨论设计用户与程序有效、方便地交互的方式。同样，我们的目的不是教您成为设计交互技术的专家，而是帮您更好地理解与使用交互技术。

0.2 视觉交流的基本概念

与观众交流时有几点需要重视，因此，在讨论细节之前先将它们提出来，并讨论与实现相关的一些问题。

0.2.1 使用合适的信息表示方式

使用合适的信息表示方法可以让观众理解图像要表达的主要意思。每种信息都有许多表示方法。有时使用颜色，有时可使用几何形状，有时使用符号图像或合成图像，而有时可使用自然图像。有时需要表示事物之间的关系而非事物本身，有时可以使用二维空间，或者使用三维空间，第三维代表的是影响力；而有时需要使用三维空间，第三维表示的是关键部分。了解是什么吸引观众注意力的最好方法是观察他们观察事物的习惯并询问哪一部分会引起他们注意。这包括给他们很多的选择题进行测试，其中有些选项对观众来说是新的，但是，别想当然地认为你能知道观众的选择，因为你和观众的思考方式是不一样的，而且想从图像中获得信息的也不是你，而是观众。

0.2.2 图像应突出重点

图像应该集中表达希望观众理解你想表达的信息。通过舍弃无关的或易分散注意力的内容使图像表达的信息更集中。不要生成仅是为了好看的图像；不要生成有歧义的图像。比如，对图像着色时需要考虑选择平面着色（每个显示的多边形只有一种颜色）还是平滑着色（每个多边形中颜色是平滑渐变的）。当采用几何图形表示实验样本数据时，可以用平面着色表示数据本身的精度，因为平滑着色表示的精度高于数据本身。另一方面，考虑用图表表示理论信息，若表示的是连续数据，则可以采用平滑着色。原则是不要为生成更吸引人的图像而歪曲事实。

0.2.3 使用合适的信息展示级别

一个非常有用的原则可以帮助你决定到底投入多少精力来优化图像，这就是根据图像展示信息级别高低来优化图像。图像展示信息的级别有三种：个人级（图像是给自己看的），同事级（图像是给同事或合作者看的）和专业级（图像是给你想吸引的观众们看的）。当自己要理解一些事情而作图时，可采用非常简单的图像，因为自己知道它们要表示什么；图像只需
3 要包括最重要的部分，而不需要精化。如果图像是与同事交流工作用的，而这些同事知道工作的大概情况但没有很深的理解，此时可以采用质量高一点的图像或使用图例帮助表达思想，但并不需要花很大力气精化图像。但是，当图像向公众演示，比如科学论文或项目申请书的

插图，应该将图越精化越好。明确了这些原则，您会发现有时工作可以非常简单，使用单图即可，有时要精化一点，稍微思考一下看待事物的方式，可能需要使用简单的动画或交互功能，从而使观众理解你的想法；而有时需要连续的动画和高分辨率图像，努力在最短的时间内达到最大震撼力。

0.2.4 采用合适的信息格式

信息（或数据）可分为区间数、序数、标称数。区间数是与有物理意义的数字相关的，比如速度、重量、数量，包括各种情况下的测量数据，并能用实数表示。这些实数可以用于生成图形。序数能与其他类似数据比较，但本身不一定有意义。比如，一个人的教育水平能与另外一个人比较，但更多的是说一个人比另一人的教育水平高（或低）。序数可由尺寸（更大，更小）或色彩映射表中的颜色表示（更亮、更暗；更红、更蓝）。色彩映射表将在颜色章（第5章）中介绍。注意，在色彩映射表中，可以根据颜色图例区分一种颜色与另一种颜色相比更高或更低，但不能得到颜色的精确值。因此，对序数和区间数来说，色彩映射表对前者更有用，因为当色彩映射表用于区间数时，它无法得到确切的值。标称数描述没有顺序或数字意义的事物。例如，头发的颜色可描述为“红色”，“棕色”，“金黄色”或“灰色”。标称数可用形状、图案或不同颜色表示。

0.2.5 注意图像显示的准确性

如果数据是零散的，那么就显示这些零散数据；不要采用几何渐变或颜色渐变等技术来显示它们，因为这些技术表示数据点之间的信息是连续渐变的，从而使人们对零散数据点产生错误解释。必须认识到，简单但更精确的表示比花样百出的表示方法更容易让用户理解。当使用简单的数值方法，比如微分方程，为行为建模或表示几何体运动时，需要在图例或标题中指明，而不要暗示表示的是更精确的图像。因为你要表达的模型并不精确。通常情况下，要努力消除演示图像的歧义。



图0-1 带两个变量的函数的三种视图：区域人工着色（左），构造人工着色曲面（中），构造自然着色曲面（右）

为了说明上述观点，采用函数曲面显示作为例子。这个例子将在第9章中多次出现。我们经常会碰到二维空间定义的实函数，可以采用一定方法将函数转换成可视图像。现在考虑函数$f(x,y)$定义的二维曲面，将其可视化，同时思考怎样将$f(x,y)$的值在二维领域内用颜色表示出来。通过观察图0-1的三个部分，可以发现有许多方式来表示该函数。理解本书的两个目的：其一在于告诉你如何创建并显示这样的图像，其二在于当你采用其中一种方法或其他方法与观众交流时，帮助你理解它们。

0.2.6 理解并尊重观众的文化背景

采用图像与观众交流时，观众只能通过自己的知识背景来理解图像。知识背景与文化有关，包括专业文化（工程、医学、高能物理、出版、教育、管理），社会文化（小城镇、大城市、农村），地域文化（北美、西欧、中国、日本），民族文化（美国原著、定居美国的墨西哥人、祖鲁族），或宗教文化（佛教、伊斯兰教、天主教、新教）。对不同文化背景的观众，颜色可能有不同的含义。有些符号或图像所表示的意思甚至与作者期望表达的大相径庭。必须确保图像正确表达希望传达给观众的意思，而不会由于未理解观众的文化背景而传达了错误信息。

0.2.7 使交互成为用户熟悉的高效操作

如果观众习惯使用特定的控制操作，那么就模拟这类控制操作，并使模拟的控制动作与用户习惯相吻合。你实现的控制操作可能与很多人正在使用的控制操作相类似，因此考察这些现有的交互实例很重要。有些控制操作可以嵌入控制面板中，供用户调用。有些控制操作则可以直接用来操纵图像。

我们先介绍控制面板的几种交互操作。如果想从几个不同的选项中选出其中一个，可采用单选按钮系列。单选按钮系列中每个选项都带有一个按钮，每个按钮显示选项是否被选中。选中任意一个按钮的同时，其他按钮的选择被取消。如果用户希望选择零个或多个非互斥的选项，则可用普通按钮表示每个选项，每个按钮可独立选择。如果想从连续数值范围中为参数选择一个值，则可用滑块或调谐钮来选择值（当值发生变化时，应该将变化后的值显示出来）。如果想输入文本（例如颜色名字，比如采用X Window颜色命名系统中所用颜色名字，或程序文件
5 名），可用文本框输入。所有这些控制操作的例子将在交互章（第7章）中详细介绍。

如果希望把控制操作应用于场景直接操纵场景对象，则需要实现另外一类交互操作。这类交互操作把场景理解为特定空间，通常是包含物理对象模型的三维空间，并认为对象模型的行为可在空间中定义。这样，可以在空间中点击鼠标选择（点击）物理对象，并识别鼠标点击选中的对象。还可以通过在场景中点击鼠标（选中场景中的对象），并保持点击状态（按下按钮）的同时，沿水平或竖直方向拖动鼠标，实现物体绕纬度方向和经度方向的旋转操作。如果把上述鼠标拖动操作解释为场景的放大和缩小，而不是二维旋转操作，则可以实现场景的放大和缩小。最后要说明的是，上述鼠标操作实现的交互控制都可以用键盘来代替实现。键盘拥有的大量按键为交互操作提供更大的自由度。另外，键盘具有由击键成单词，或由关键字符组成专门控制操作等功能为定义对象行为提供语义增强的功能。例如，可定义一组特殊符号来表达对象的行为，类似于文本编辑中的光标控制符或者老式游戏中的运动控制符。

总之，应把交互操作看成另外一种语言。不同用户具有不同的文化背景，对应众多不同的交互操作类型，从而为创建高效的交互程序奠定了基础。

书中在论述视觉交流和计算机图形学时提出了一些观点。我们认为当你开始学习计算机图形学时，需要理解这些观点，以便于理解后续章节。这些观点列举如下：

- 形状是准确表达观点的有用工具，需要小心使用。
- 颜色是创建有效图像的重要因素。比其他因素更为重要的是，颜色可以向观众展示图像的重要部分，传达需要表达的确切信息。
- 与观众交流信息时，选择自然还是人工的形状和颜色十分重要。
- 把颜色与运动结合在一起可表达高达五维的信息，但必须仔细考虑如何将不同信息纳入每个维度表示出来。
- 为了正确理解图像表达的含义，应向观众提供正确的视图和相关的背景知识。
- 现代图形API和计算机使创建动态图像变得很容易，因此可以利用动作使图像生动丰富。
- 不仅可创建动态图像，还可让观众与图像交互，让他们自己探索问题。
- 运动与交互是表示模型行为的方法，必须将其视为交流的重要组成部分。
- 观众总是运用自己的文化背景来理解图像，因此在设计图像时必须对观众的文化背景有足够了解。

当您生成图像时请牢记这些交流要点。它们不仅适用于生成简单图像，有些还适用于计
6 算机图形学范围之外的工作。其中许多要点需要相当的经验才能将其运用自如。只有理解了它们的重要作用，你才有可能创造有效的图像。

0.3 三维几何和几何流水线

计算机图形基本上是三维的，至少在本书中如此。因此图形系统必须处理三维几何数据。计算机图形系统通过创建三维几何流水线来处理这个问题。三维几何流水线是将三维点转换成二维点的一系列过程，其中三维点是三维几何的基本组成单元。这样，就将三维对象转换成二维对象。这个过程适用于应用程序开发者把模型对象转换成适用于显示硬件的对象。这里将列出几何流水线的步骤，帮助大家理解其操作过程。几何流水线各个步骤的细节将在第1章关于视图与投影中详细介绍。几何流水线如图0-2所示，本章将讨论开始几个步骤，更多详细内容将在下面几章中给出。



图0-2 几何流水线的步骤与映射

0.3.1 场景与视图

场景与视图是反复用于图形学的两个基本概念。场景是在图像中显示的三维空间对象组合，包括空间中所有的几何对象与光照。视图是用于创建图像的信息集，是包括场景空间、场景坐标系、坐标系中带观察方向的视点、将空间中可视部分映射到二维可视平面的投影操作。场景定义空间中的对象，而视图定义了空间中哪些部分可见及观众的观察方式。事实上，几何流水线包含了场景和视图的定义及创建，这些正是本节的主题。

0.3.2 三维模型坐标系

为了创建图像，必须定义表示图像各部分的几何数据。创建并定义几何体的过程称为建模，将在介绍建模原理和OpenGL建模中描述（分别是第2章和第3章）。通常情况下，建模过程是先将每个对象在其有意义的坐标系（即三维模型坐标系）中定义好，然后采用一系列模型变换将对象变换到统一空间的坐标系（即三维世界坐标系）中，在该空间中所有的对象均可见。 7
可以这样理解，最初的模型是图形对象的模板，通过模型变换将对象在世界空间中调整到合适的大小、方向和位置。常常通过定义模板对象来生成多个其他对象。模板对象可以是其他对象的一部分，或者重新调整大小、方向、位置后在场景中使用。模型变换将模板对象改变为可用的真实对象。这些模型变换可随时修改或在交互中修改，从而使模型得到所需的效果。

0.3.3 三维世界坐标系

场景包括三维坐标系中的一系列图形对象，所有对象都使用该三维坐标系。这个坐标系称为世界坐标系，将场景的所有组成部分放置到这个单一共享的世界中，通过显示设备展示给用户时，场景是一致的。注意，世界坐标系可表示真实世界中模型的真实坐标。与真实场景一样，该坐标系与其中的场景是客观存在的，与观察者无关。为了创建场景的图像，下一步就是添加观察者。

首先考虑这样一个场景，使用三维模型坐标为场景创建一个模型，并将其变换到三维世界坐标系。场景展示一片有许多常青树的森林。建模过程中，首先在模型坐标系中用树干（圆柱体）和树冠（圆锥体）构造一棵树，树干长1个单元，位于原点；树冠长3个单元，位于

树干顶部，所以，整棵树位于原点，高度为四个单元。但是森林应该有很多树，因此，我们希望在X-Y平面上x方向从－4到4，y方向从0到8的长方形区域内画100棵树。通过模型变换将树从模型坐标系变换到世界坐标空间中任意一点，在世界坐标空间中生成一片森林。图0-3展示了这片森林。



图0-3 一片森林

0.3.4 三维眼坐标系

创建好三维世界之后，希望能够从任意位置进行观察。不过计算机图形学的观察模型通常有指定的默认观察方向和（或）视点。例如，将视点的默认位置指定为原点，向－Z（有时是+Z）方向看，Y轴向上。视图变换时选择好视点位置和三维世界空间中的所有对象，重新调整坐标系，将视点调整到适当的位置，往适当的方向观察。视点与其他对象的相对位置没有改变；场景中的所有对象仅是在三维空间挪了个点而已。将视图变换应用到世界坐标系中，就得到三维眼空标系。视图变换有多种方式，由图形API决定。通常是指定视点的位置，视线朝向，视图“向上”的方向。这些信息足以让图形系统计算视图变换。

8

0.3.5 投影

接下来，在映射到图形显示设备之前，三维眼坐标必须转化成二维坐标。几何流水线下一步实现的操作称为投影。投影将视域体映射成二维长方形，称为视平面，即在眼坐标空间中平行X-Y平面并包含2D眼坐标系。在讨论投影之前，先想像一下图像的最终显示效果。想像眼睛在场景中的某一点，朝某个方向看，想像当通过一个长方形窗口看场景时能看到什么。你将不会看到整个场景，而只看到场景视域体之内的空间。

计算机图形学通常用两种投影，并为标准图形API所支持。一种将模型中的所有点通过与Z轴平行的方向投影到X-Y平面，将模型中的所有点从眼空间映射到视平面。与目视方向平行的线上的所有点映射到视平面上同一个点，相当于每一点保持x、y坐标而忽略了z坐标。这称为平行投影，可在x、y坐标系中得到精确值。平行投影中最常见的是正交投影，投影线与Z轴平行。图形API一般提供正交投影，也可能提供其他平行投影方法。工程图纸通常提供两个平行投影视图，一个如上所示，另一个将世界空间旋转90°，从而使z坐标可以精确表示。

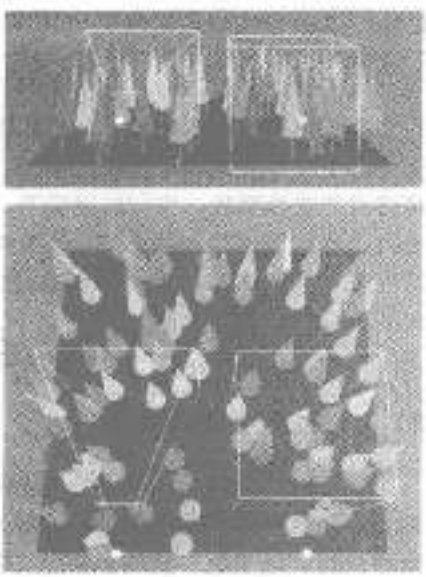


图0-4 透视视域体和正交视域体的正视图（上）和俯视图（下），两个视图表示整片森林

第二种常用投影将眼睛视为一个点，场景中的每个点沿着从眼睛到该点的视线映射到视平面，这是透视画法的经典方法。这种投影称为透视投影。平行投影中，无论对象与视平面距离多远，投影到视平面的大小都是一样的，而在透视投影中，对象距离眼睛越远，画得越小。透视投影看起来更真实，而平行投影更易于排列或测量空间物体。投影分为平行投影与透视投影，视域体也分为正交视域体和透视视域体。图0-4展示了透视和正交投影视域体。左边是透视投影的视域体，呈金字塔型，前面被截断。右边是正交投影的视域体，呈长方形。两种情况下，眼睛的位置都用白色

9

球体表示。两种投影的视平面都是视域体的前面，背平面是视域体的后面。透视投影的截金字塔型的视域体称为视截体。

视域体描述空间的可视区域，但真正的视图是显示在视平面的二维眼空间上的几何体。图0-5显示了图0-4中透视投影和正交投影及它们的视域体中的场景。将这些视图与前面图中的视域体的场景部分进行比较，可以发现它们在整个场景中的对应关系。图中每部分都显示一些特征。左图中，左边树有部分被视域体前面截除。右图中，由于地平面与视域体平行，因此看不到地平面，也看不到平面边界。由于是正交投影，所以所有树的高度都相同；由于很多被看得见的空间遮挡，所以只能看到场景后面一小部分树。

10



图0-5 图0-4视域体表示的透视投影场景（左）和正交投影场景（右）

0.3.6 裁剪

与人眼或照相机不能看到整个三维世界一样，图像也不能展示整个三维世界。三维视域体是场景中的可视部分，由下节定义的投影所决定。视域体通过视图的左、右、上、下、前、后边界来指定。理解了这些内容，就可以通过指定对象在视域体之内部分与之外部分来裁剪对象。舍弃视域体之外的所有对象，对视域体之内的所有对象进行投影。透视投影中，有些树（或树的有些部分）超出可视空间的左、右、上或下面部分。我们不需要考虑这些树。但图形系统必须将这些树变为不可见。图0-6是从另外的视点观察森林的一个例子。除了有些是场景边上或场景上面以及下面的对象不可见外，有些场景中的对象也是在可视空间之外的。因为它们距离视点太近或太远。注意，图中左边树的前面部分被裁剪了，已把它们处理成不可见了，因为其距离观察者的眼睛太近。场景中还有些树在图中所示的树后面，由于它们离视点太远，也被裁剪了。

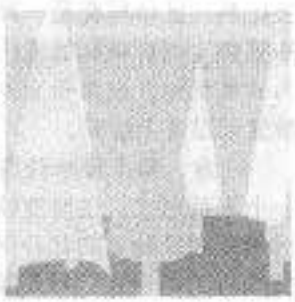


图0-6 场景裁剪

11

视域体裁剪操作与视图、三维眼坐标系在同一个空间中，在使用投影变换将场景投影到二维眼坐标系之前进行。裁剪操作不仅保证视图仅包含可视对象，还有利于提高图形系统的性能，因为它使后续的显示过程不必处理那部分不可见的对象。

0.3.7 选择透视投影或正交投影

计算机图形学的初学者经常遇到的一个问题是对图像选择透视投影还是正交投影。两者各有优劣，这里列出一些关于选择两种方法的快速指南。

正交投影适用于以下情况：

- 需要确认场景中的物体是否排列整齐或大小是否相同，
- 需要测量场景中的物体，
- 需要确认线条是否平行。

透视投影适用于以下情况：

- 使图像看起来比较真实，
- 能在场景中移动并具有与人眼类似的观察效果。

- 不需要测量或排列图像中的对象。

对这两种投影方法有了相关经验，或知道观众想要的效果之后，就很容易选择更合适的投影方法了。

0.3.8 二维眼坐标

投影操作将三维真实坐标空间映射成二维真实空间坐标，使观察者看到原始场景几何体的二维视图。二维眼坐标系中的一个点相当于三维眼空间的一整条线。因此在投影时深度信息丢失了，观察者很难知道场景中的深度信息，特别是采用平行投影时，无法从对象的相对大小来判断深度信息。对透视投影而言，可以从那些原来大小相同而距离不同的对象投影到二维平面后对象的大小来判断对象在空间的位置。如果进一步采用隐藏面剔除技术来显示场景，则还可以通过近处对象遮挡远处对象的原理帮助判断对象在空间中的位置，这对正交投影也同样适用。

12

0.3.9 二维屏幕坐标

几何流水线的最后一步是在二维眼坐标空间调整对象的坐标，使其适合二维显示设备的坐标系。显示屏幕是一种数字设备，因此二维眼空间坐标中的实数需要转化成代表屏幕坐标的整数。可对坐标值先作比例映射再取整。这个操作称为窗口到视口映射，新的坐标空间称为屏幕坐标，或显示坐标。这个步骤完成后，整个场景是用整数型屏幕坐标表示的，可在二维显示设备上绘制。

注意，整个流水线将几何体用对象顶点表示，通过多个不同变换从一个表示形式变为另一个表示形式。这些变换都保证不同表示形式的场景的几何体顶点的一致性，同时计算机图形学还假设场景的拓扑保持不变。举个例子，若三维模型空间中两个点通过一根线相连，则转换后的两点在二维屏幕空间中还是用一根线相连的。因此，原始模型中的几何对象（点、线、多边形或程序员自定义的对象）以及边与顶点的关系一直保持到屏幕空间，并在屏幕空间中绘制。

0.4 外观属性

除了几何属性，计算机图形学还可定义对象的外观属性，因此可以让对象看起来很自然，或用颜色表示特殊意思。

到目前为止我们仅讨论了模型顶点的坐标，还需生成顶点的许多其他属性才能创建场景图像，我们在定义顶点时设置这些属性，顶点通过几何流水线时这些属性仍然保持不变。有些属性涉及一些目前尚未提到的概念，包括：

- 顶点深度值，定义为视点参考方向上视点到该顶点的距离。
- 顶点颜色。
- 顶点法向量。
- 顶点材质。
- 顶点纹理坐标。

这些属性用来定义图像中对象的外观特性，图形系统根据这些属性，可以消除隐藏面，在顶点转换到二维屏幕坐标后，为屏幕上的每个顶点计算颜色。

13

外观属性由绘制流水线操作处理，绘制流水线操作在几何属性映射到屏幕空间之后再执行，前面提到的几何图元分成许多小块，再转换成窗口中的像素。外观属性信息与顶点相关，

当处理顶点信息时，也处理外观属性信息，从而生成每个像素的颜色。深度信息缓存等操作也在此时进行，为场景生成可见的表面。因此，外观属性信息处理操作在几何信息处理操作之后，外观属性相关的介绍也在大部分几何章节之后。这里我们介绍一些外观属性相关的内容，稍后再作展开。

0.4.1 颜色

颜色可由程序员直接定义，如果场景由光照与材质定义，则颜色也可由光照模型计算得到。大部分图像API支持RGBA颜色系统：颜色由三原色（红、绿、蓝）和α通道来定义。α通道用于绘图时物体与背景色的混合。这些系统可绘制大量颜色，一般是16 000 000种颜色数量级。这些在后续关于颜色与光照的章节中作详细介绍（分别是第5章和第6章）。

0.4.2 纹理

纹理映射是为场景添加视觉效果的最有用的方法之一。纹理映射可以将图像，比如照片中的视觉内容贴到场景对象上。通过纹理映射，可使对象表面达到照片的效果，或其他可使图像更有趣更有真实感的效果。这是很有用的功能。

0.4.3 深度缓存

创建场景时，只要画出与视点最近的对象，这些对象后面的对象会被遮挡，成为不可见。在绘制阶段，只要保存并比较视点与已绘制像素的距离和待绘制像素的距离，就可实现这一目标。如果待绘制像素到视点距离小于已绘制像素，则用新像素的颜色覆盖原来的颜色，反之，保持原来的像素颜色。这就是深度缓存操作。深度缓存计算简单，基本上所有图形学系统都支持。

0.5 观察过程

我们来看图形系统处理场景直到最后向用户显示过程中所完成的全部几何操作过程。图0-2中，若忽略裁剪和窗口到视口映射过程，我们可以看到，从定义基本几何模型开始，应用了模型变换、视图变换、最后在屏幕坐标系应用投影变换。可用函数组合来表示这个顺序

```
projection(viewing(modeling(geometry)))
```

或将函数组合写成乘法形式，若应用结合律，

```
projection*(viewing*(modeling(geometry)))=(projection*viewing*modeling)(geometry)
```

离几何模型较近的操作将先执行，因此，在定义几何模型之前，应首先定义投影操作，然后是视图变换，最后是模型变换。这一顺序与采用透视投影还是平行投影无关。这个顺序是本书稍后讨论的采用场景图组织场景的关键要素。

不同的实现，相同的结果

到现在为止，我们还未讨论顶点在几何流水线中传输过程的细节。有许多方法能够实现这个传输过程，每一种实现都能产生正确的结果。因此，若发现你的计算机图形系统的图形流水线与本文不同时不必惊讶，基本原理与主要操作步骤还是类似的。

举个例子，OpenGL将模型变换与视图变换结合成一个变换，称为模型视图变换（modelview transformation）。因此，在OpenGL中，模型变换与视图变换会有所不同。另外，

图形硬件系统通常在裁剪操作之前，还会执行窗口到规格化坐标变换操作，对特定坐标系统进行优化。这种情况下，其他步骤不变，只有最后一步将变为规格化坐标到视口的映射。

大多数情况下，我们并不考虑每个步骤是如何执行的，我们需要在建模阶段正确表示几何模型，确保几何模型外观属性信息定义正确，指定合适的视点，正确定义窗口与投影。图形系统就会执行几何流水线与处理外观属性信息。图形学技术的更多细节将留在高级图形学课程中介绍。

0.6 图形卡

早期计算机图形学与其他计算过程一样采用相同的通用处理器和内存。现在，几乎所有交互式图形系统都采用图形卡来加速图像生成操作，这些图形卡采用专用处理器与内存。特别是高端、高性能的图形卡，称为图形加速器，这些图形卡采用专用处理器来加速几何数据处理，还采用专用内存来存储待计算的图像（有时也称颜色缓存）和图形计算所需的其他数据（深度缓存，刻影缓存，纹理内存等）。这些图形卡的计算速度很快，因为是专为图形计算而设计的。图形API的作用之一就是将调用的图形API函数映射到图形卡功能上，以达到加速目的。

图形卡与图形API的发展相互促进。图形卡的设计要支持API所需的计算类型，而API的发展需要考虑利用图形卡日益提高的性能和灵活性。图形API的发展趋势是沿着支持图形卡的
15 新功能方向进行的。

0.7 一个简单的OpenGL程序

本书OpenGL的用例程序都比较相似。每个例子都使用了GLUT（Graphics Library Utility Toolkit）工具包。通常OpenGL系统都附带GLUT工具包。如果你的OpenGL版本不带GLUT，可以在网上获取GLUT源代码。参考http://www.opengl.org/resources/libraries/glut.html下载页面。需要下载源代码、编译并安装。同样地，在事件处理章节中，我们使用MUI（micro user interface）工具包，虽然某些OpenGL发行版并不附带MUI。在网上或通过上述链接可找到GLUT和用户接口功能的其他资料。

与其他功能强大的API一样，OpenGL十分复杂，提供许多编程方法来解决图形问题。我们的例子中仅采用了OpenGL对交互程序支持良好的部分功能，因为我们认为计算机图形学应该和用户交互技术一起学习。若使用高度真实感图形，图像生成将花费很长时间，用户无法与图形实现交互操作。

使用OpenGL生成交互图像的典型程序结构是怎样的呢？本书所有例子均采用C语言结构化程序设计方式。由于OpenGL操作对面向对象编程支持不是很好，因此，本书不采用C++。OpenGL维护了大量无法用图形类包装的状态数据，而面向对象编程通常使用对象来维护状态。许多函数，比如事件回调函数，不能处理参数，必须采用全局变量才能进行交互。因此，通常通过全局变量来创建应用环境，使用全局变量而非参数在函数内外传递信息。

从下面的代码中可以看到，main()函数一般由建立OpenGL系统的操作组成。通常有两种实现方法：第一，通过GLUT创建并设置用于显示的系统窗口；第二，通过定义回调函数，建立事件处理系统。回调函数是事件发生时使用的，并初始化模型与显示环境。然后，main()调用主事件循环操作，在主事件循环中驱动所有操作，这将由事件驱动一章（第7章）作详细介绍。

display()函数十分重要，它用于创建所定义的模型显示。在第2章讨论场景图时，display()的任务是遍历场景图，执行场景图定义的操作，即定义几何模型，设置外观属性和调用各类变换。

GLUT的事件驱动方法将在事件章节中讨论。GLUT完全通过事件来操作，对程序需要处理的每个事件，都需要在main()函数中定义相应的回调函数。回调函数是当相关事件发生时，系统事件处理程序调用的函数。主事件循环启动后，改变窗口（Reshape）事件生成窗口，显示（display）事件调用自身的回调函数在窗口中绘出初始图像，如果其他事件也定义了相应的回调函数，则当事件发生时，启动相应回调函数。回调函数允许用户拖动窗口或改变窗口大小，当用户对窗口进行操作时被调用。空闲（idle）回调函数在系统空闲时间（当系统不生成图像或响应其他事件时）重新计算图像，并在终端上显示改变的图像。 16

以下是按照这种格式所写的一份完整源代码，并用它介绍上述函数及回调函数的细节。注意，这里reshape回调函数设置系统的窗口参数，包括大小、形状、窗口位置，并定义视图的投影方法。进入主事件循环及窗口事件（比如调整大小或缩放）发生时，调用该函数。当reshape结束时，调用redisplay函数，在redisplay中调用display回调函数，用于设置视图并定义场景几何模型。这些操作完成后，OpenGL操作结束，图形系统转到计算机，查看是否有其他图形相关事件，如果有其他图形相关事件，则程序需要定义回调函数来进行管理。如果没有，则产生idle事件，调用idle回调函数。这时可能改变一些几何参数，然后又一次调用redisplay函数。

```
#include <GL/glut.h>      // windows系统，由于其他系统的其他头文件
    // 其他需要的头文件
//  所需要的全局数据块和类型定义

//  函数声明
    void doMyInit(void);
    void display(void);
    void reshape(int,int);
    void idle(void);
    // 其他函数声明
//  初始化函数
    void doMyInit(void) {
        set up basic OpenGL parameters and environment
        set up projection transformation (ortho or perspective)
    }
//  改变窗口回调函数
    void reshape(int w, int h) {
        set up projection transformation with new window
            dimensions w and h
        post redisplay
    }
//  显示回调函数
    void display(void) {
        set up viewing transformation as in later chapters
        define the geometry, transformations, appearance you need
        post redisplay
    }
//  空闲回调函数
    void idle(void) {
        update anything that changes between steps of the program
        post redisplay
    }
//  其他需要的图形和应用函数                                          17

//  主函数－建立系统，将控制权交给事件处理程序
    void main(int argc, char** argv) {
//      通过GLUT初始化系统和自己的初始化工作
        glutInit(&argc,argv);
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
        glutInitWindowSize(windW,windH);
        glutInitWindowPosition(topLeftX,topLeftY);
        glutCreateWindow("A Sample Program");
        doMyInit();
```

```
    //      定义事件显调函数
            glutDisplayFunc(display);
            glutReshapeFunc(reshape);
            glutIdleFunc(idle);
    //      进入不显示循环
            glutMainLoop();
    }
```

现在，我们了解了OpenGL程序的基本结构，下面我们来看一个完整的程序，并介绍其表示几何流水线的方法，描述OpenGL的使用细节。该程序用于简单模拟均匀金属长条的温度分布，在后续关于科学问题求解的章节（第9章）中详细描述。这里我们只研究程序结构，而非程序功能。程序输出图像如图0-7所示。代码放在图之后将代码分为几段，可以更好地理解每段代码在图形操作中的作用，然后在代码之后讨论每段程序。

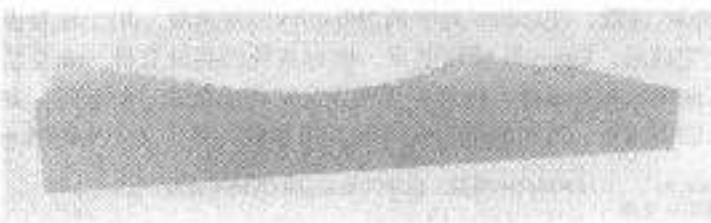


图0-7 金属长条的热量分布。参见彩图

在下面的代码中，每段代表一个特定的OpenGL函数，可以是初始化，建模，视图变换和回调函数。我们将每部分分别标出，以便于查找及重点关注。OpenGL函数本身比较简单，但整个
18 程序看起来有些复杂。不过，当你学习了本书前几章之后，会发现这份代码其实十分简单。

```
// 示例 具有设定热点和冷点的均匀金属长条的温度变化

// 系统和变量的声明及初始化
#include <GL/glut.h>      // 对于Windows，其他系统可能有变化
                          // 这也包含gl.h和glu.h
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#define ROWS 10     //   金属长条的大小为ROWSxCOLS（去单位）
#define COLS 30

#define AMBIENT 25.0;     //周围温度，摄氏度
#define HOT 50.0          // 热源单元的温度
#define COLD 0.0          // 冷槽单元的温度
#define NHOTS 4           // 热单元的个数
#define NCOLDS 5          // 冷单元的个数

GLfloat angle = 0.0;
GLfloat temps[ROWS][COLS], back[ROWS+2][COLS+2];
GLfloat theta = 0.0, vp = 30.0;

//  设置金属长条上固定热点和冷点的位置
int hotspots[NHOTS][2] =
    { {ROWS/2,0},{ROWS/2-1,0},{ROWS/2-2,0},{0,3*COLS/4} };
int coldspots[NCOLDS][2] =
    { {ROWS-1,COLS/3}, {ROWS-1,1+COLS/3}, {ROWS-1,2+COLS/3},
      {ROWS-1,3+COLS/3}, {ROWS-1,4+COLS/3} };
int myWin;

void myinit(void) {
    int i,j;

    glEnable(GL_DEPTH_TEST);
    glClearColor(0.0, 0.0, 0.0, 1.0);

//  设置单元的初始温度
```

```
    for (i=0; i<ROWS; i++) {
        for (j=0; j < COLS; j++) {
            temps[i][j] = AMBIENT;
        }
    }
    for (i=0; i<NHOTS; i++)
        temps[hotspots[i][0]][hotspots[i][1]]=HOT;
    for (i=0; i<NCOLDS; i++)
        temps[coldspots[i][0]][coldspots[i][1]]=COLD;
}
```

19

```
//  在模型坐标系的第一个八分象限内创建单位立方体
void cube(void) {

  typedef GLfloat point[3];

  point v[8] = {
      {0.0, 0.0, 0.0}, {0.0, 0.0, 1.0},
      {0.0, 1.0, 0.0}, {0.0, 1.0, 1.0},
      {1.0, 0.0, 0.0}, {1.0, 0.0, 1.0},
      {1.0, 1.0, 0.0}, {1.0, 1.0, 1.0} };

  glBegin(GL_QUAD_STRIP);
      glVertex3fv(v[4]);
      glVertex3fv(v[5]);
      glVertex3fv(v[0]);
      glVertex3fv(v[1]);
      glVertex3fv(v[2]);
      glVertex3fv(v[3]);
      glVertex3fv(v[6]);
      glVertex3fv(v[7]);
  glEnd();

      glBegin(GL_QUAD_STRIP);
      glVertex3fv(v[1]);
      glVertex3fv(v[3]);
      glVertex3fv(v[5]);
      glVertex3fv(v[7]);
      glVertex3fv(v[4]);
      glVertex3fv(v[6]);
      glVertex3fv(v[0]);
      glVertex3fv(v[2]);
  glEnd();
  }

void display(void) {
    #define SCALE 10.0
    int i,j;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  //  这部分定义模型变换
      glMatrixMode(GL_MODELVIEW);
      glLoadIdentity();
      // 视点  观察中心  向上方向
      gluLookAt(vp, vp/2., vp/4., 0.0, 0.0, 0.0, 0.0, 0.0, 1.0);

      // 为整个场景设置旋转操作
      glPushMatrix();
      glRotate(angle, 0., 0., 1.);

    // 画分图长条
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            setColor(temps[i][j]);
//  这是用于显示中各项的模型变换

    glPushMatrix();
    glTranslatef((float)i - (float)ROWS/2.0,
                 (float)j - (float)COLS/2.0,0.0);
    // 0.1冷, 4.0热
    glScalef(1.0, 1.0, 0.1+1.9*temps[i][j]/HOT);
    cube();
    glPopMatrix();
```

20

```
            }
        }

    // 通过前面的操作和变换来在屏幕上绘制
    glPopMatrix();
    glutSwapBuffers();

    }
}

void reshape(int w,int h) {
// 定义投影变换

    glViewport(0,0,(GLsizei)w,(GLsizei)h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, (float)w/(float)h, 2.0, 300.0);
    glutPostRedisplay();

}

void setColor(float t) {
// 颜色基于HOT=red(1,0,0)和COLD=blue(0,0,1)
// 假设在任何时候COLD<=HOT
    float r, g, b;
    r = (t-COLD)/(HOT - COLD); g = 0.0; b = 1.0 - r;

    glColor3f(r, g, b);

}

void animate(void) {
// 当系统空闲时，这个函数被调用，它调用
// iterationStep()去改变数据，因此下一幅图像被改变
21
    iterationStep();
    glutPostRedisplay();

}

void iterationStep(void) {
int i, j, m, n;
    float filter[3][3] = {{ 0.    , 0.125, 0.    },
                          { 0.125 , 0.5,   0.125 },
                          { 0.    , 0.125, 0.    } };

    // 增加材质的温度
    for (i=0; i<ROWS; i++)// 备份 temps 直到重新创建它
        for (j=0; j<COLS; j++)
            back[i+1][j+1] = temps[i][j]; // 将边界放置到back中

    // 用来自原始temps[][]中的邻近值填充边界
    for (i=1; i<ROWS+2; i++) {
        back[i][0]=back[i][1];
        back[i][COLS+1]=back[i][COLS];
        }
    for (j=0; j<COLS+2; j++) {
        back[0][j] = back[1][j];
        back[ROWS+1][j]=back[ROWS][j];
        }
    for (i=0; i<ROWS; i++)// 根据back值扩散
        for (j=0; j<COLS; j++) {
            temps[i][j]=0.0;
            for (m=-1; m<=1; m++)
                for (n=-1; n<=1; n++)
                    temps[i][j]+=back[i+1+m][j+1+n]*filter[m+1][n+1];
        }
    for (i=0; i<NHOTS; i++) {
        temps[hotspots[i][0]][hotspots[i][1]]=HOT;
        }
    for (i=0; i<NCOLDS; i++) {
        temps[coldspots[i][0]][coldspots[i][1]]=COLD;
        }
```

```
    // 更新旋转角
    angle += 1.0;}

int main(int argc, char** argv) {

// 初始化GLUT系统和定义窗口
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(500,500);
    glutInitWindowPosition(50,50);
    myWin = glutCreateWindow("Temperature in Bar");

    myinit();

// 定义事件回调函数并进入主事件循环
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(animate);
    glutMainLoop(); /*  进入事件循环  */
```

22

0.7.1 OpenGL程序main()函数结构

OpenGL应用程序中的main()函数结构可能与你之前所熟悉的程序结构不大相同。这里，main()函数有一些关键操作：设置显示模式，定义用于显示的窗口，及程序所需的初始化操作。还有一些你可能并不熟悉的操作：如定义一些事件回调函数，在事件发生时供系统调用。最后，在主事件循环函数中，将控制权转移给计算机事件系统。

设置显示模式时，同时告知系统程序所需的特性。在本例中，

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
```

告诉系统使用双缓存模式，RGB颜色模型，还有深度测试。有些属性在真正使用之前需要激活，比如深度测试需要在myInit()函数中用

```
glEnable(GL_DEPTH_TEST)
```

来激活。深度测试的细节和OpenGL对深度测试的处理见第1章。

设置窗口（或多窗口，OpenGL允许打开并激活多个窗口）要通过一系列GLUT函数调用来完成。GLUT函数设定窗口位置，定义窗口大小，并为窗口命名。在程序运行过程中，可以使用窗口系统的标准技术重新改变活动窗口。这是由GLUT reshape()函数使用底层窗口系统来实现的。

0.7.2 模型空间

代码中的函数cube()定义一个单位立方体，立方体每个面与坐标轴平行，一个顶点位于原点，一个顶点位于(1,1,1)。立方体通过如下方式创建：首先定义表示立方体的八个顶点的数组，然后用glBegin() ... glEnd()构造两条四边形条带，从而画出组成立方体的六个面。在OpenGL建模那章（第3章）中详细介绍。这里，立方体使用自身坐标系。这与要定义的热量传递模拟的坐标空间可能有关，也可能无关。

23

0.7.3 模型变换

模型变换操作出现在display()函数或调用display()的函数中，定义世界空间中几何模型应实施的基本变换操作。我们的例子中，基本几何模型是单位立方体，Z方向尺寸（而非X或Y方向尺寸）可变，再实施X与Y方向平移变换（而非Z方向），把长方体单元放置到正确的位置上。变换操作的顺序，它们的定义方法和代码中出现的glPushMatrix()/glPopMatrix()操作将在第3章介绍。第3章主要与OpenGL建模相关。现在你只需知道这个模型变换用来创建

长方形对象，其高度代表温度。

0.7.4 三维世界空间

这个程序的三维世界空间是通过模型变换后安置图形对象的空间。从变换中可得到该空间的信息：平移变换后的立方体的x坐标位于－ROWS/2与ROWS/2之间，y坐标位于－COLS/2与COLS/2之间。ROWS与COLS分别为30和10，因此x坐标在－15到15之间，y坐标在－5到5之间。由于z坐标不变，因此z坐标最小值是0，最大值不超过4。所以整个长方体位于x坐标从－15到15，y坐标从－5到5，z坐标从0到4的区域。（这个估算并不完全正确，不过精度够用。欢迎你找出其中的小错误。）

0.7.5 视图变换

视图变换在display()函数开头部分定义，包括设置模型观察矩阵为单位矩阵（不改变世界坐标的变换矩阵），然后指定视图参数。视图变换通过调用OpenGL的glutLookAt()完成：

```
glutLookAt(ex, ey, ez, lx, ly, lz, ux, uy, uz);
```

参数(ex, ey, ez)表示视点坐标，(lx, ly, lz)表示视线的方向，(ux, uy, uz)定义视线"向上"的矢量方向。这些在第1章中讨论。

0.7.6 三维眼空间

这个程序中没有特别表示出三维眼空间，因为它只是生成图像的中间步骤，不过，我们将视图的中心设置在原点，原点也是图像的中心。视点设置为中心的右上方，向原点望去。因此，在视图变换后，对象看起来有些向上倾斜，并退在一边。这是场景在三维眼空间的表示，最终将其投影到视平面。

0.7.7 投影操作

本例中，投影操作在reshape()函数中定义。投影也可在其他地方定义，不过在reshape()中定义可以很好地将投影操作与视图变换操作分离开来。

在OpenGL中很容易定义投影操作。正交投影可以用以下函数定义：

24

```
glOrtho(left, right, bottom, top, near, far);
```

left与right分别是正交视域体的左侧面与右侧面的x坐标，bottom和top分别是下底面和上底面的y坐标，near和far分别是前面与背面的z坐标。透视投影可以用以下函数定义：

```
gluPerspective(fovy, aspect, near, far);
```

第一个参数是视场角，第二个参数aspect是窗口高宽比[⊖]，参数near和far与上面相同。在透视投影中，眼睛位于原点，所以不需指定视域体其他四个剪裁平面。这四个剪裁平面由视场和高宽比决定。视场指定了视图的宽度，高宽比指定了视图的高度与宽度比值。因此，高宽比为0.5意味着视图的宽度是高度的两倍。

重绘窗口时，改变后的窗口宽度与高度可从reshape事件中得到，将投影的宽高比（宽度与高度之比）设为与窗口相同。例子代码中也是这么做的，重绘得到的窗口没有变形。如果采用

⊖ aspect通常译作"宽高比"，用来描述显示器屏幕宽度与高度之比，如计算机监视器屏幕的宽高比为4:3，高清晰电视屏幕的宽度比为16:9。OpenGL把aspect定义为宽高比。但本书多处将aspect定义为高度与宽度之比，故译作"高宽比"。但本书也有将aspect定义为宽度与高度之比的地方，仍译作宽高比。如下一段就是如此定义的。因此，本书中"宽高比"与"高宽比"通用，其含义是清楚的。——译者注

固定的高宽比，而窗口本身形状发生改变时，原来的场景在新窗口显示扭曲，使用户产生错觉。

0.7.8 二维眼空间

三维世界投影到视平面后称为二维眼空间，对应于视域体的前平面。二维眼空间的实际尺寸由API决定。OpenGL将眼空间单位化，因此，每个坐标值范围为－1到1。

0.7.9 二维屏幕空间

例子中的系统初始化时，窗口大小为500×500像素，上顶点位于（50, 50），或者说位于距离屏幕左上角向下50像素，向右50像素。窗口的屏幕空间即指屏幕的这个区域。不过窗口的坐标系与窗口本身的位置无关，三维眼空间中的点（0.0.0）在屏幕空间中的坐标为(249,249)。屏幕空间是离散整数坐标系，每个坐标代表不同的像素，坐标值从0开始计算。

0.7.10 科学问题编程

程序的大部分用于几何建模，设置视图，处理控制动画的事件，这种情况很正常，图形学需要处理大量用于生成图像的操作。不过程序中还包括金属长条中热量传递的科学问题。这部分由iterationStep()函数和filter[][]数据结构来处理，filter中的关键一点是数组的所有元素非负，且总和为1，所以，它保持能量守恒，这种扩散模型将在有关科学应用的章节中介绍（第9章）。这里仅指出本例中涉及的科学问题。如果要使用不同的热能传递模型或描述不同的扩散问题，则可将这部分代码用相应的代码替代。

0.7.11 外观属性

程序中，物体的外观属性由函数setColor()定义，由display()函数调用。由于建模也定义在display()函数中，所以外观属性其实是建模的一部分——同时对几何对象和外观属性进行建模。使用温度的值来计算每个单元的颜色，使用函数OpenGL glColor3f()计算颜色的红，绿，蓝成分。这是为对象外观定义颜色的最简单的方法，不过很有效。 25

0.7.12 从另一角度分析程序

从另一角度分析该程序，即从功能上划分，而非从几何流水线属性划分。下面对此作简单介绍。

myinit()的任务是建立程序运行环境。这里计算定义几何模型数组的值，定义特定颜色，或其他一次性操作。最后，该函数还设置初始投影操作类型。

display()的任务是完成所有图像生成操作所需的任务，这可能需要对大量数据进行处理，但函数本身没有参数。图形学问题的数据需要用到全局变量进行管理。全局数据可以看作程序员创建的环境，一些函数负责处理数据，而图形函数应用数据来定义显示参数。大多数情况下，全局数据只有在良好文档的情况下才改变，因此使用全局数据的方法是可行的（程序应该有很好的文档）。当然，有些函数可能生成或接受控制参数，由自己决定这些参数是作为全局数据还是局部数据来处理。但即使如此，一般还是应该将数据作全局声明，因为可能有许多函数用到这些数据。OpenGL维护自己的运行环境，称为系统状态。自己定义的函数也可改变系统状态。

reshape()的任务是处理用户对图形显示的控制。函数有两个参数，分别是窗口在屏幕空间（或像素）的宽度与高度。当用户调整窗口大小时，这两个参数用于重置屏幕尺寸并调整

场景的投影参数。GLUT与系统窗口管理器联动，使得窗口可以灵活移动或改变大小，而无需程序员直接管理任何与系统有关的操作。系统独立性是使用GLUT的重要原因！

animate()的任务是处理空闲（idle）事件——无任何事件发生的事件。这个函数定义了无用户交互时程序所作的工作，也是程序实现动画的一种方法。只有在深入介绍事件概念以后才能理解animate()函数的细节。animate()可以在改变全局环境之后，调用glutPostRedisplay()操作重新显示改变的结果。这个操作向系统发送"redisplay"事件，告诉系统下一步调用display函数。

若无其他事件发生，则程序的执行顺序如图0-8所示。注意，main()不调用display()，而调用事件处理函数glutMainLoop()。glutMainLoop()永远不会终止，等待事件进入系统事件队列，然后调用响应
26 的事件处理回调函数。由于此时窗口状态尚未设置，因此，由系统设置display事件，当glutMainLoop开始启动时display函数就立即被调回。若无其他事件行为，则程序将不断调用idle()函数，使图像随时间不断改变，也就是产生动画效果。

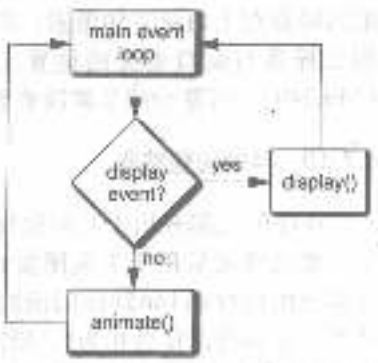


图0-8 idle和display事件循环

现在，我们对animate()函数作介绍。该程序展示金属棒中热量传输过程，该过程由热量方程来定义。程序中把从一处到另一处的热量传递看作扩散过程。每个单元的热量值设为原单元值的加权平均值，其中加权值由filter数组指定。在每个时钟周期或程序空闲时的每个时钟周期，调用扩散过程来计算新的温度，更新显示的旋转角度，函数最后调用glutPostRedisplay()，将调用display()函数按新的显示角度绘出新的温度分布图像。

考虑画一张图来表示函数之前的调用关系，这对理解简单程序中函数的执行顺序十分有用。程序是事件驱动的，所以回调函数animate()、display()、reshape()都不是程序直接调用的，函数调用者/被调用者的关系图如图0-9所示。

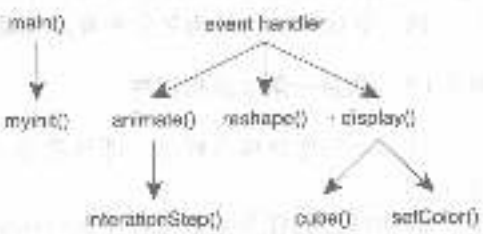


图0-9 例子程序中函数的调用者/被调用者关系图

与大多数OpenGL程序类似，本程序中的函数也只能由事件回调函数或初始化函数init()调用。init()仅在main()中调用一次，所有的事件回调函数都由事件处理程序调用。对大多数OpenGL程序，函数调用关系大致如下：回调函数可调用多个函数。但除了回调函数外，其他函数只能在程序初始化或事件回调函数中调用。不过，函数既可在初始化中调用，也可在回调函数中调用，或被其他函数调用，因此调用者/被调用者的关系图事实上应该是图结构，而非如图中所示的树结构。

现在，我们已经有了几何流水线的概念，也理解了程序的结构，可以进入后续章节，学习如何定义视图和投影环境，如何定义基本几何模型，如何在定义好的环境中使用display()
27 函数生成图像。

0.8 OpenGL扩展

本章以及全书都着重于介绍OpenGL图形API、计算机图形学和OpenGL的基本特点。我们不介绍系统的高级特性，只考虑每部分的基础应用。但是，不论是OpenGL的原始版还是特定图形的版本都能支持很复杂的图形功能。当你具备相当的图形技术之后，会发现本书采用的

"vanilla" OpenGL版也有功能上的局限性。

OpenGL的高级特性包括存储或控制场景信息的特殊操作。这些特殊操作包括多边形拼接建模，NURBS曲面建模，以及用户自定义并使用的变换方法；剪裁测试和更普遍的模版缓存和模版测试，能获得绘制细节的反馈模式渲染，以及客户/服务器所需的设施。OpenGL 2.0也包括顶点和片段的着色语言的说明，使用户能够编写专用着色器以用于特定场景。然而，本书是一本OpenGL的通用教材，要了解OpenGL的扩展功能请参阅更多资料，作为标准OpenGL的补充。用户可以通过前面给出的网址了解更多OpenGL的扩展功能。

0.9 小结

在这章中，我们讨论了几何流水线，并对流水线的各个步骤进行了具体的说明。指出每一步做些什么，对最终图像有何贡献。我们还说明了外观属性如何适应流水线（尽管外观属性是在绘制流水线中实施的），以及这些操作如何在一个完整的OpenGL程序中实现的过程。我们可以修改这个程序，使它成为其他图形学程序的基础。实际上，这个举例程序是一个有用的工具，在这章中我们没有遇到任何其他程序，不过通过这个样例程序，很快就能开始编写其他更复杂的程序。

0.10 本章的OpenGL术语表

本章中我们使用了一些OpenGL的函数和定义，本节中我们对这些函数和定义进行回顾。后续章节中我们也会有相似的术语表，包含新加入的OpenGL函数。 28

类型

GLfloat：和系统无关的浮点数定义

OpenGL函数

glBegin(xxx)：指定由顶点函数定义的几何模型的类型

glClear(parms)：清除由参数定义的窗口数据

glClearColor(r, g, b, a)：将图形窗口的的颜色设为背景颜色

glColor3f(r, g, b)：为后续顶点调用设置RGB值

glEnable(parm)：激活参数定义的性能

glEnd()：几何模型定义区域的结束标记，和glBegin(...)配对使用

glLoadIdentity()：将单位矩阵写入由glMatrixMode指定的矩阵中

glMatrixMode(parm)：指定后续操作使用到的系统矩阵

glPopMatrix()：在glMatrixMode指定的当前矩阵栈中，将栈顶的矩阵弹出栈

glPushMatrix()：复制当前矩阵栈中栈顶的矩阵，用于后续栈操作；当栈顶矩阵被glPopMatrix弹出后，该矩阵的值将恢复为最近调用的glPushMatrix栈顶矩阵值

glRotate(angle, x, y, z)：旋转几何模型，旋转轴的参数为（x,y,z），旋转角度为angle

glScalef(dx,dy,dz)：将顶点坐标乘以指定值，对几何模型进行缩放

glTranslatef(tx,ty,tz)：顶点坐标加指定值，平移几何模型

glVertex3fv(array)：根据三维矩阵设置几何模型顶点

glViewport(x,y,width,height)：使用整数窗口坐标，指定绘制图形的视口尺寸

GLU函数

gluLookAt(eyepoint, viewpoint, up)：通过定义视点位置，视点观察位置和观测向上方向设置观察环境参数

gluPerspective(fieldOfView, aspect, near, far)：基于观察环境参数，给定定义视域体的四个参数以定义透视投影

GLUT函数

glutCreateWindow(title)：创建图形窗口，并给出窗口名

glutDisplayFunc(function)：为显示事件指定回调函数

glutIdleFunc(function)：为空闲事件指定回调函数

glutInit(parms)：根据main()函数的部分参数初始化GLUT系统

29 glutInitDisplayMode(parms)：根据传入的符号参数设置系统显示模式

glutInitWindowPosition(x,y)：设置窗口左上角顶点屏幕坐标

glutInitWindowSize(x,y)：在屏幕坐标系中设置窗口的宽度与高度

glutMainLoop()：进入GLUT事件处理循环

glutPostRedisplay()：设置重绘事件，触发再次显示事件

glutReshapeFunc(function)：为改变窗口事件指定回调函数

glutSwapBuffers()：后台颜色缓存中的内容交换到前台颜色缓存中用于显示

参数

GL_COLOR_BUFFER_BIT：与glClear一起使用，表明清空颜色缓存

GL_DEPTH_BUFFER_BIT：与glClear一起使用，表明清空深度缓存

GL_DEPTH_TEST：指定使用深度测试

GL_MODELVIEW：指定使用模视矩阵

GL_QUAD_STRIP：指定所用顶点是连续有序的四边形条带的顶点

GLUT_DEPTH：指定窗口使用深度缓存（从而可进行深度测试）

GLUT_DOUBLE：指定窗口使用后台缓存（从而可使用双缓存）

GLUT_RGB：指定窗口使用RGB（或RGBA）颜色模型

0.11 思考题

1. 除了基于API的程序，还有其他的图形处理方法，比如不同的建模绘图和其他终端用户工具。列举基于API的图形处理方法和使用图形处理工具（比如Photoshop或者商业绘图软件）的不同之处。本章的样例程序可以说明基于API的图形处理是如何工作的，尽管这只是一个很简单的程序。更多更复杂的程序将在后续章节中介绍。
2. 跟踪样例程序中3D几何模型在几何流水线中的演变过程。从在模型空间中定义单位立方体顶点开始，应用系列变换将该顶点放置到世界空间中，再应用视图变换将该顶点放置到3D眼空间中，然后应用投影操作将该顶点放置到2D视空间中。不用任何数学工具，论述该顶点坐标随着上述变换过程所经历的变化。
3. 生活中充满视觉交流，经常要使用专门的术语。写一篇短文，运用专门术语论述一类视觉交流的例子。一个例子是在财务数据报表中的可视化术语。另一个例子是天气预报中的可视化术语。短文应包括一些例子，并分析形状、颜色、几何关系、行为等要素在这些领域中的应用。
4. 视觉交流也包含科学交流，在这种情况下使用术语更专业。参阅*Science*，*Scientific American*，或者

其他高档次的科学杂志，仿照上一个问题写一篇有关科学交流问题的文章。

0.12 练习题

1. （画图）视觉交流比计算机图形学范围更广，掌握其中一个方面的技能就能举一反三。在导师的指导下，在本地环境中选取一个视觉[30]，下面出这个视图，并和其他人交流关键的部分。在画完后，和其他人讨论你想通过这幅画传达什么信息，传达了多少信息。指出全画中和其他人交流最成功的部分，并讨论如何用计算机图形学技术创建这个部分。
2. 将本章中的样例程序编译并执行，熟悉图形学编程的编译器。拖动并改变窗口大小，熟悉reshape()函数。改变窗口的形状（使之变窄而不变短，或者使之变短而不变窄），观察窗口和图像的反应。

0.13 实验题

1. 利用本章的样例程序，实施下列实验：
   a. 改变窗口的大小和左上角坐标[main()函数]。
   b. 改变金属棒热点和冷点的位置[myinit()函数]。
   c. 改变filter()函数，以改变例子中热传导的方式。这样会使热量在不同方向上传播，或者使热量转移到中心单元的所有的八个邻接单元，而不是四个单元。
   d. 改变setColor函数，以改变金属棒颜色的计算方式。每个颜色值都是0~1之间的实数。
   e. 改变角度的增量，以改变图像旋转的速度[函数animate()]。
   f. 改变金属棒边缘的处理方式，即改变原来采用的通过重复边缘温度值来传递热量的方式，能够应用另一条边缘的温度值实现温度能有效地从一条边缘转移到另一个边缘的效果，如同金属棒是圆环面的[iterationStep()函数]。
   g. 改变金属棒的投影方式得到新的视图，从透视变换视图改变为正交投影视图[reshape()函数]（你需要认真参考第3章中关于正交投影和视图的知识）。

   尽量多地进行上述实验，在前面练习代码中添加并改变合适的代码，观察因此带来的图像和程序的变化。总结不同变数在生成最终图像时所起的作用。
2. 改变filter数组的值，从而改变金属棒中热量扩散的模型[函数iterationStep()]。filter中所有的值非负，其和为1，这样就能保证能量守恒。不过你可以改变这些值，使能量只在一个方向上或沿着一条线传递。这样很像纤维模型，能量只能在纤维中传递，而不是纤维之间。不过你需要改变热点和冷点来反映这种特性。[31]
3. 继续讨论reshape()函数。读懂这个函数的源代码并思考怎样才能让这个函数作出不同的响应。当前版本使用窗口尺寸w和h来定义透视投影，以保证原图像高宽比能够保存，但是太窄的窗口可能会切掉图像的一部分。在窗口变窄时可能需要改变投影角度。改写reshape()函数的源代码，使窗口的行为跟着改变。
4. 本书提供了一些样例程序，更多的OpenGL程序可以在因特网上找到。找一些这样的程序，通过创建本章介绍过的函数回调来证明（或否定）书中的论断：函数不是在程序初始化时操作，就是在事件回调时操作。这项工作对运用OpenGL开发图形程序的方法有何帮助？程序中用户定义的函数在函数回调图中操作最频繁的地方在哪里？[32]

第1章 视图变换和投影

本章详细介绍几何流水线的两个过程——视图变换和投影，对它们的基本模型和操作进行描述。视图变换涉及整个场景的内容，所以在场景上下文中讨论定义视图所需的关键信息。投影包括透视投影和正交(或者平行)投影，本章讨论定义它们的关键信息。本章假定读者对二维和三维解析几何有所了解，并熟悉简单的线性映射。如果对这方面存有疑问，请参看第4章的相关部分。

流水线是计算机图形学的一个基本特征，本章在对它进行讨论后，介绍如何使用OpenGL创建视图变换和投影、如何使用OpenGL定义视图变换和透视、正交投影函数、如何在程序中使用它们以及它们如何响应窗口操作。

本章还介绍一些和几何流水线中基本步骤相关的内容：包括投影过程中的裁剪、显示图像的屏幕窗口的定义、包含实际图像的视口的定义。另外还介绍了双缓冲(在另一个不可见的窗口中创建图像，把它和可见窗口进行交换)和管理隐藏面的方法，最后介绍如何利用按左右眼视点计算得到的两幅图像创建立体视图的方法，这两幅图像保存在相邻的视口中，通过一些合适的视觉技术可以混合在一起。

本章还简单介绍视图在创建高效视觉交流方面所起的重要作用，虽然这很简单，但是在设计图像的时候必须考虑这些问题。

[33] 在阅读完本章后，读者应该掌握针对场景选择适当的视图和投影、对它们进行定义并利用OpenGL编写代码的能力。同时也应该理解双缓冲的作用和三维图形学中隐藏面的概念，并能在图形编程中使用。

1.1 简介

我们强调二维计算机图形学的重要性，因为只有理解了图形的二维处理方法才能掌握计算机图形学的原理，而二维图形学只是三维图形学的一个特例。然而，几乎所有可用的图像观察技术都是二维的(例如显示器、打印机、视频和电影)，甚至眼睛中起成像作用的视网膜表现的也是二维环境。所以，为了用图像表现场景，必须建立二维场景的二维表现形式。正如在前面看到的，首先为场景建立一系列的模型，并把模型放到场景中，这样在世界空间中就有了一系列的物体。然后，定义观看场景的方式和视图在屏幕上显示的方式。

在前面定义几何流水线的时候，设定了一个场景，首先从设定三维坐标系统这一步开始，进而在三维世界中定义完整的场景，在进入这一步之前，首先要完成模型的建立和变换两项工作，这两步将在下面两章讨论。图1-1描述了略去建模步骤的处理过程。



图1-1 建立场景图像的几何流水线

首先看一个真实世界的例子，并分析如何表示真实世界。作者喜爱的美国约塞米蒂(Yos-

emite）国家公园，是三维空间一个非常好的例子它有一组由巨石，大树和流水组成的场景，从多个不同的视角可看到不同景观。图1-2显示的是分别从山谷中从下往上和从冰河点上从上往下看到的半圆顶巨石，这是公园中一个非常经典的场景。

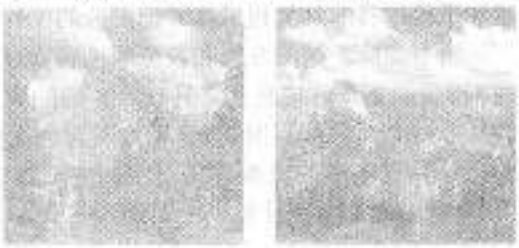


图1-2 分别从约塞米蒂山谷（左）和冰河点（右）看到的半圆顶巨石的风格景观

从照片中可以看出场景的组成部分。首先，视图取决于所站立的位置。如果站在山谷的底部，看到的是巨石的前部，如图1-2左图所示。如果站在约塞米蒂山谷的边缘，将看到巨石的右侧面。所以，视图取决于观察位置，谓之视点。其次，视图同样也取决观察的目标点，谓之视图参考点。图1-2左图是直接观察半圆顶巨石，而右图则是观察圆顶巨石后面的一个点。目标点不仅影响看到圆顶巨石的视图，而且影响看到圆顶巨石周围的区域。从山谷中看视图的右边，可以看到山谷南面的山壁，从冰河点上看视图的右边，可以看到流向Merced河的Vernal和Nevada瀑布，在右边更远处，还可以看到公园南部高高的齿状山脊。最后一点可能并不明显，因为我们头脑处理的是图像内容，而看到的视图取决于场景中的向上方向的定义，而不管站立时头是直立还是倾斜的。可能把视图想象成是由相机获取的而不是你所看到的景象更容易理解，假如把相机倾斜45度角得到的照片和水平时或垂直时得到的照片肯定是不一样的。

视图同样取决于观看的视野，或者说是看大场景还是空场景。在图1-2中，左图看到的只是半圆顶巨石的视图，而右图看到的则是包括圆顶巨石在内的全景图。当两张照片都是方形的时候，可以把左边的照片想像成纵向布局的照片，而右边的则可以看成是横向布局的照片。这是受图像的高宽比影响的结果。在所有的讨论中，场景世界是一样的，但是决定所看到图像的因素有眼睛的位置、观察的目标点、观察视图的向上方向、观看的宽度、视图的高宽比。在计算机图形学中，必须指定这些参数来才能正确地定义图像。

一旦确定了视图，必须把它转化为可以在二维成像设备上显示的图像。这就像用数码相机记录图像：视图中的每个点（图像中的每个像素）必须指定一个颜色。在数码相机中，光线通过镜头，把该点的颜色记录在感光器件上，但是在计算机图形学中，必须精确计算二维屏幕空间中每个点的颜色。把三维场景变换到二维空间需要定义一系列步骤：场景中哪一部分需能，哪一部分是落在视点看到的范围内，以及场景如何转化到二维视图空间中。解决最后一个问题的最好的办法是比较两种不同镜头的工作方式：一个是标准的镜头，它从相机前面的锥体中聚集光线；另一个是高级的摄影镜头，它从一个很小的圆柱体中聚集光线，这些光线到达感光器的时候基本上是平行的。

这个观察模型和计算机图形系统非常相似，并且它按照前一章介绍的几何流水线进行操作。先从本地坐标系中建立一系列的对象模型开始，然后通过坐标变换把这些对象放到世界坐标系中，在前一章的温度例子中可以看到这样的例子。在第2章和第3章将作详细介绍。最终目标是在三维世界空间中建立完整的模型，视图操作的基本工作是在世界空间中定义一个观察方式，让观察者可以观察到模型空间中允许看到的物体。定义观察方式包括把视点放在三维世界空间中，并且创建眼坐标系统，从而把三维世界空间转换到这个三维眼坐标空间，接着运用投影原理在三维眼坐标空间中定义一个二维平面，用来显示场景，实际上定义了一种映射关系，把那个平面看作为观察平面。通常可以把这个平面看成是一个屏幕，也可以是一张纸、一个视频帧或者其他空间。

有时候“切除”场景的一部分是非常有用的，这样就可以看到场景中隐藏在某些物体后

34

35

面的东西，本章简单讨论了裁剪平面——一种实现裁剪的技术，在以后章节作更详细的介绍。这和系统进行的裁剪视域体外面部分的工作相关。

模型变换、视图变换和投影等机制通过图形API来管理，所以，图形编程的任务就是给这些API提供正确的信息，并且按照正确的顺序调用这些API函数。接下来讨论视图变换和投影的基本概念，并且利用OpenGL处理这些过程。但是，现在还不必关注变换的细节，可以简单地把计算机图形的变换看成对空间中的点进行操作的函数，且这些函数保证几何模型不变。第4章将讨论如何在计算机中观察这些函数以及它们的工作原理。

1.2 视图变换的基本模型

假如在眼前放置一个长方形取景框，透过它观看世界，这可以帮助建立视图处理的物理模型。取景框可以移动，眼睛可以在任何位置，看任何方向。这相当于定义了视点和视图参考点，取景框的形状和视线的方向决定了图像的高宽比和向上方向。一旦在世界中确定了观看位置，把取景框放在眼前，就可以建立透视投影。通过把取景框拉近或远离，改变投影的范围宽度。最后，假如在这个取景框纸板上放一些很小的方形透明材料，并根据透过这些小方形看到的颜色填充，就建立了一幅可以随身携带的图像。

考虑图1-3左图所示的例子。图中的世界坐标系是按照常规定义的。在图中有一个（简单的）模型和视点。小白球表示视点，小黑球表示视图参考点，而小浅灰球则表示视图向上方向上的点。黑球和灰球都和视点相连。这样，一旦模型在视图中显示，就可以想像它是如何被观看的。为了方便起见，右图显示了从左图视点看到的模型，图中省去了坐标轴。本章最后的练习题将对此作进一步探究。

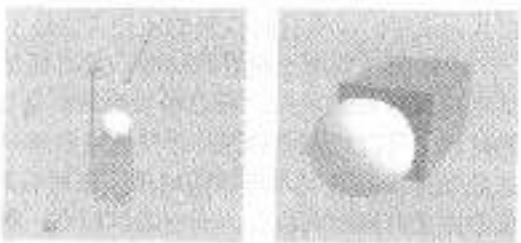


图1-3 在世界坐标系中建立的视图（左）和实际视图（右）

在世界空间里确定视点的位置就相当于定义了眼坐标系，由此定义了把视图中所有物体从世界坐标系转化到眼坐标系的变换。这是一个直观的数学变换，通过建立一个从世界坐标系到眼坐标系的基本变换矩阵并把它应用到场景中的所有物体来实现。这个变换是把视点放在原点、看向Z轴方向，同时Y轴是向上方向。这个过程称为*视图变换*，它把几何物体从世界坐标系变换到眼坐标系，同时保持模型中的几何关系不变。一旦视点放在标准的位置上，所有的几何模型都经过视图变换，系统在流水线的下一步就可以非常方便地把这些几何模型投影到观察平面上。

在第2章中讨论建模，包括对模型位置、方向、大小进行从局部坐标到世界坐标的变换。视点也可以通过这种建模来定义，即从眼睛放在标准位置开始，然后把它变换到想要的位置。

一旦组织好视图变换的信息，就可以开始着手确定场景投影到屏幕的方式。图形系统提供定义投影的方法。一旦定义了投影，就着手计算把场景映射到显示空间上。本章紧接着会讨论这个问题。

1.3 定义

在考虑如何观看场景的时候，有些因素是必须要考虑的，这些因素和具体使用的API无关。本章随后会讲述它们在OpenGL中是如何处理的，这些因素如下：

- 定义模型的观看方式，包括视点的位置、视图的方向和范围。这些定义了视图变换。

- 定义三维空间到二维空间的投影，因为三维空间必须在二维的显示设备（例如屏幕上才能被看见。对于不同的投影有不同的实现方式。
- 定义观看图像的视图设备区域，称为图形窗口，这不应该和屏幕上的窗口混淆，尽管它们可能指同一个空间。
- 定义视图在窗口中的显示位置，这定义了窗口中的视口和窗口到视口的映射，该映射把二维眼空间变换到屏幕空间。

这四个因素分别称为建立视图环境、定义投影、定义窗口和定义视口。下面将按照这个顺序对它们分别进行讨论。

1.3.1 建立视图环境

场景是通过在世界空间中建立一系列的原始模型，并对它们进行变换来建立的。正如第2章所述，为了把视点放到标准位置，我们把世界空间通过视图变换变到另一个三维空间。在这个三维空间中，定义三个关键元素：视点的位置、眼睛看的方向和眼睛的垂直方向。定义这三个元素后，对场景中的模型进行视图变换，按照定义的环境建立可见的场景的视图。

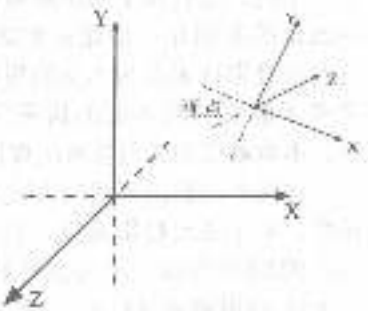


图1-4 标准OpenGL视图模型

图形API提供了定义视图的工具，根据计算来变换整个场景。例如，OpenGL在右手坐标系统里建模，然后把整个场景中的几何模型（包括所有的光源、方向，见第6章）变换到以视点为原点的坐标系中。该变换是在左手坐标系中，取以视点看向Z轴的负方向。如图1-4所示，视点放置在普通建模空间中，眼坐标系是与视点相关的另一个左手坐标系。

定义视图所需的信息包括：

- 视点位置的（x, y, z）坐标；
- 视点面向的方向或者正对的目标点的坐标，该方向将成为眼坐标系的Z方向；
- 在世界空间中视点的“向上”方向，该方向将成为眼坐标系的Y方向。

一般图形API都提供函数来设置视点和观察方向。

38

视图变换操作针对定义在世界空间中的物体，并且把眼睛变换到标准模型位置，这就得到了在前一章讨论的眼空间。视图变换的关键是旋转世界空间，使得向上方向和Y轴方向一致，视线方向和负Z轴方向一致。然后移动世界空间，把视点位置移动到原点，最后缩放世界空间，使得目标点或目标向量的值是（0.0, 1）。这些操作是把视点从标准位置移动到在API函数中定义的点的逆模型变换操作。这些对第2章的建模非常重要。所以，在利用OpenGL API定义视图环境的时候会作更深入地讨论。

1.3.2 定义投影

视图变换定义了三维眼空间，但这个空间还不能在标准设备上显示。所以，必须把场景映射到和显示设备对应的二维空间，例如计算机显示器、录像屏幕或者纸上。这种把三维空间中的物体映射到二维空间的技术称为投影操作，这些操作都是基于一些简单的原理定义的。

在现实世界中（或者用摄像机）看到东西，是光线经过凸透镜汇聚到达视网膜上（或者电影胶片上的CCD单元）产生的。这就是把三维空间投影到二维空间的过程。图1-5所示是一个非常好的透视例子。光线经过眼睛（或摄像机）的凸透镜，远处的平行光在地平线上汇聚，所以对于观察者来说，相对远的物体比近处的小。实际上，物体汇聚的方式取决于投影观看的花

视宽度。这种通过把物体投影到观察平面上或汇聚到点上的投影称为透视投影。

另外，假如让图像中的物体和实际场景中的物体一样大小。例如，在工程绘图中通过图像进行精确测量，那么正交投影可以达到这个目的，它把场景中所有的物体通过平行光线投影到观察平面上。在正交投影中，不管物体离眼睛有多远，它都和原来的物体大小相同。

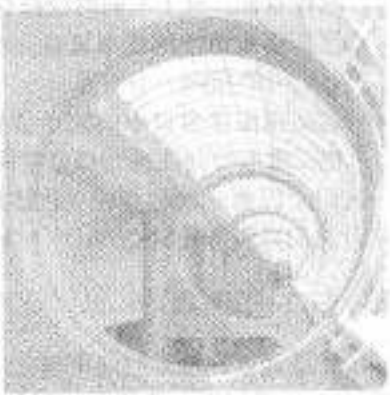


图1-5 一幅具有非常强烈透视感的建筑物照片

图1-6显示从同一个视点看到的两幅房子的图像和它的模型坐标系。为了能看到房子的隐藏部分，它的前面部分是半透明的。左图所示的是透视投影，可以看到房子前面和后面部分的大小不一样，并且房子的边和顶显得小了很多，就像它们离观察者后退了。右图是正交投影，可以看到房子前面和后面部分是同等大小的，而且房子的轮廓和顶边都是平行的。这两幅图像的差别并不是那么明显，但显然很容易发现。这两种投影技术各有不同的用处，通过对比这两个结果来了解它们在不同的情况下的工作方式也很有帮助。本章最后的练习会再次探讨这个问题。

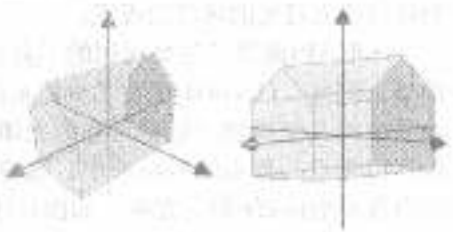


图1-6 一个简单房子模型的透视投影图像（左）和正交投影图像（右）

这两个投影以不同的方式对三维空间中的点进行操作。对于正交投影来说，只考虑所有三维眼空间中的点的X和Y坐标，所有点映射到XY平面上。如果点(x,y,z)映射到点(x',y')，那么x' = x且y' = y。每一个三维眼空间中的点都是和Z轴平行的直线在观察平面上的投影，所以又称正交投影为平行投影。

对于透视投影来说，每一个点都映射到三维眼空间Z = 1的平面上，它是这个点和原点的连线与Z = 1平面的交点。三维眼空间中的每个点表示该点与三维眼空间的原点生成的直线。如果点(x,y,z)映射到点(x',y')，则通过相似三角形有x' = x/z，且y' = y/z，透视投影的相关数学知识将在本章稍后讨论。图1-7显示了得出这些等式的相似三角形。

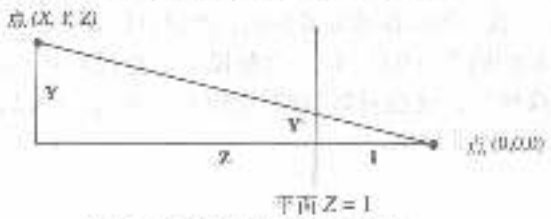


图1-7 计算透视投影的示意图

正如前一章所述，投影变换可以把场景映射到二维眼空间上。当点被变换的时候，它的z值应该保留，以方便以后进行深度测试或者透视校正纹理的计算。在一些API中，z值转化为整数，并且它的符号也改变了，所以离视点远的点（离三维眼空间远的点）z值更大，这和左手坐标系一致，使得系统在深度测试中使用正整数。

1.3.3 视域体

投影通常需要考虑视域体，即空间中经过投影后可以看到的区域。事实上，不管用什么

投影方法，在长方形显示设备上观看到的图像已经限定了场景的上下和左右边界，这与观看空间的顶面、底面和左右侧面是对应的。观看的对象通常不包括离视点太近或太远的物体，这就相当于限定了观看区域的前后边界，分别用ZNEAR和ZFAR来表示前后边界，注意，0< ZNEAR≤ZFAR，

视域体是指包含所有三维空间中经过投影后可以看到的物体的空间。图1-8分别显示了透视投影和正交投影的视域体，图中白球代表视点。右图的矩形体是正交变换的视域体，左图的棱锥台体是透视变换的视域体。它们表示了三维眼空间中映射到二维眼空间区域上的视域体，包括了图1-4所示左手坐标系中的Z轴。视点在原点，即标准位置，其Z轴所指的方向和普通的坐标系相反。两个视域体都在$X-Y$平面后面，而视点在$X-Y$平面上。

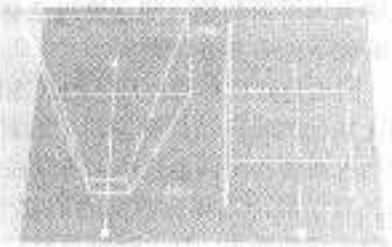


图1-8 透视投影（左）和正交投影（右）的视域体

透视投影的视域体假设观看空间是关于Z轴对称的，所以它只能指定在空间中特定的位置，但平行投影的视域体则不同，它可以指定在任何位置。所以，可以在空间任何部分建立平行投影，也可以随意移动平行投影视域体来观看模型的任何部分。这似乎让平行投影看起来很有用，但实际上不是的，因为可以通过简单的变换把想看的区域变换到标准位置。

如前一章所述，视域体是非常重要的，因为只有视域体中的物体才能显示。任何视域体外面的物体都被去除——可以理解为在投影操作中不可见，所以也不会被图形系统处理。任何物体如果部分在视域体中部分在外面，都将进行裁剪，只有视域体内的部分才能被看到。观看空间的边界投影为可视矩形的边，而视域体的远近平面则限制了投影中可见的最近和最远空间。这保证了图像中只出现想要显示的部分，防止出现眼睛后面或者过分远的物体。

1.3.4 正交投影

要定义正交投影，必须指定视域体的上下、左右和远近平面。每个平面都按照下面的等式定义：

coordinate（坐标）=value（值）

每个平面用一个实数定义。例如，下面六个等式定义了一个在每个坐标轴上包含两个单位的平行投影的视域体。

$x=-1;\quad x=1;$

$y=-1;\ y=1;$

$z=0.1;\ z=2.1;$

改变每一个值都会改变视域体在每个平面的大小。所以，假如要增加视域体左边的空间，就减少左边平面的值，要缩小看到的空间，就增大平面的值。对于右边来说，情况正好相反：减少值导致空间变小，增大值导致空间变大。

1.3.5 透视投影

透视投影比平行投影复杂。透视投影以Z轴为中心，它的大小取决于观察的范围(水平方向上，通常用角度衡量)和高宽比（垂直方向）。高宽比是指观察空间的宽度与高度之比，所以高宽比为1表示高度和宽度相等，高宽比为0.75表示比例是3:4，这是美国标准电视图像比例。对很多图像来说，如果投影的高宽比和窗口的高宽比是相等的，那么所看到的图像就准确地

反映了模型。观察的空间决定可以在窗口中看到多少模型，如图1-9所示。观察的范围大小就
43 像用广角镜头和长焦镜头的差别。

透视投影的另外参数是近平面和远平面。这跟正交投影是一样的。如果减少近平面的值（即向视点移动），那么靠近视点的物体就不会被切除。同样，如果增大远平面的值（即远离视点移动），那么远离视点的物体也不会被切除。

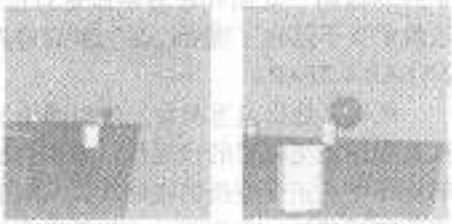


图1-9 从同一个视点，以不同的观察范围：宽（左）和窄（右），观察同一个简单场景的不同视图。两个方向的目标点都是红球

因为透视投影的视域体的远端比近端大，所以离得比较远的物体投影到窗口上的时候缩小也比较多，它们占图像的比例也较小。所以在屏幕上看到的也越小。对于两个物体，假如它们位于和三维眼空间XY平面平行的面上，那么透视投影对它们的影响是一样的。如果一个比另一个离视点远，那么远的看起来更小些。透视投影的结果是使远离的物体看起来更小，它是文艺复兴时期艺术界的一个重要发展。把它应用在表示一组排列整齐的物体的距离很有效，例如高速公路，一排电话线杆或者一排建筑。

透视画法包括一点透视、两点透视和三点透视三种。这个分类假设场景是在标准三维或者三维坐标系中分布的，它基于如下事实：假如两个物体位于同一个平面上，而这个平面和观察平面相交，那么离观察平面远的物体看起来小一些。

最简单的场景布局是两个坐标轴方向和观察平面平行，另一个和它相交。物体只有沿着那个坐标轴移动，才能改变它的投影大小，这就是一点透视。这一组平行线在观察平面上相交的点称为灭点。当然，视域体的远裁剪平面并不会让视线延伸到无穷远，但从图1-10左图可以看到它趋向于灭点。



图1-10 图1-6中房子模型的一点、两点和三点透视图

再稍微复杂一点，一个坐标轴和观察平面平行，
44 另外两个和它相交，那么物体在这两个坐标轴上的任意一个移动都会改变大小，这就是两点透视，如图1-10中间所示，虽然可能在图中看不到，但它有两个灭点。很显然，如果场景中有多组平行线，那么它就有多个灭点。正如在果园中的果树或者基地中呈规则网格分布的墓碑。作为第三种透视，如果坐标轴都和观察平面相交，就产生了第三个灭点。这种透视称为三点透视，如图1-10右图所示。

1.3.6 透视投影的计算

透视投影的计算非常直观，虽然图形系统会进行这些计算，但如果能了解这些过程还是非常有帮助的。二维透视计算与视域体透视计算的步骤基本相同，如图1-11所示，它和图1-7一样，从图中相似三角形可以得到$Y/Y' = Z$或者$Y' = Y/Z$。同理可以得到$X' = X/Z$。所以，透视投影就是在三维眼空间中把X和Y都除以Z，它的投影矩阵如下：

$$\begin{bmatrix} 1/Z & 0 & 0 \\ 0 & 1/Z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

这个3×3矩阵定义了从三维空间到三维空间的变换，它改变了X和Y的值，但保持Z的值不变。假如要严格地投影到二维空间，可以把最右一列消去。



图1-11 计算透视投影的示意图

这个矩阵表示从三维空间到三维空间的变换，称为透视变换。因为矩阵元素的分母中含有变量，所以这个变换不是线性映射。当对对象的某些属性进行插值时必须进行透视校正，这是非常重要的。如果X'和Y'是经过变换过的值，并且原先的Z值已知，可以重新计算得到原先的$X = X'*Z$和$Y = Y'*Z$。所以，在投影的计算过程中应该保留深度值，我们将在隐藏面和纹理映射中再次讨论这个问题。透视变换应用于透视投影，只有X'和Y'作为输出。在实际的操作中，可以先进行透视视域体的裁剪，然后再进行透视变换；或者先对整个场景进行透视变换，再进行平行投影，然后在变换后的空间上进行简单的裁剪。

45

1.3.7 视域体裁剪

在场景进行显示之前，要把在视域体外面的图像部分进行裁剪或者移除。对正交投影进行裁剪是很简单的，因为正交投影的包围平面通过单标常量来定义：$X = Xleft$，$X = Xright$，$Y = Ybottom$，$Y = Ytop$，$Z = Znear$，$Z = Zfar$。对一条直线相对于任意平面进行裁剪，只要检查这条直线是否在平面上，在平面外还是穿过平面。如果它在平面上，就保留，如果它在平面外，那么它被丢弃。如果它穿过平面，那么该直线视域体外那部分被丢弃。

但是，对透视投影的视域体进行裁剪却要难很多，因为它要求对倾斜的包围平面进行裁剪。可以用巧妙的办法来避免：裁剪前先进行透视变换，通过这个变换，把透视视域体的倾斜面变换成与Z轴平行，就跟正交视域体一样，然后就可以像正交投影一样进行裁剪计算。

裁剪作为投影工作的一部分通常在透视投影之后进行，这样所有的工作都是在每个坐标轴上定义的常量平面包围的空间上进行。裁剪通常针对线段，因为大部分工作都是基于线段或者由多条线段包围的区域。多边形也要进行裁剪，可以通过对包围多边形的线段进行裁剪来实现。

裁剪从线段的端点开始，它简单地检测顶点是否在矩形视域体内，把该点的坐标值和视域体边界平面的值进行比较。如果线段的两个端点都在该空间内，那么这条线段就在空间内；如果一个端点在视域体外面，那么该线段至少有一个点在空间外面，所以要替换线段参数等式的边界值并求出交点的参数。这个参数定义一个新的顶点，用该顶点替换前一个顶点，由此得到一条比原先短的线段。继续依此计算，直到线段两个端点都在空间内或者没有线段。画出最后剩下的线段，图1-12左图显示了二维空间上的例子。

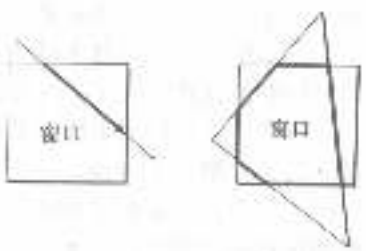


图1-12 线段和多边形裁剪

如果是对多边形进行裁剪，那么每次对多边形的一条边相对视域体的一个面进行。如果一条边从里面穿过视域体到外面，那么应该计算边和面的交点作为该线段的新顶点，并保存它，当下一条线段进入视域体的时候，同样计算它的交点。这样，先前保存的顶点和该顶点就定义了一条新线段，把该线段添加给这个多边形。图1-12右图显示了该算法在二维空

46 间的计算结果，图中粗线表示原来的线段，粗线表示裁剪后的线段。

最后，如果是对多面体进行裁剪，那么每次对多面体的一个面相对视域体的一个面进行。但裁剪完毕后，这个多面体可能会变得不完整，假如多面体有一个角在视域体外面，那么裁剪后会留下空洞，这样，严格来说该多面体就不再是多面体了。假如要填补空洞，需要保存先前和视域体边界的交点，把它们重新组成新的多变形作为多面体的边界的一部分。

一个简单的方法是对每个顶点和包围平面进行测试，但这个方法很慢。现在已经有了一些更高效的方法，其中最简单的是Cohen-Sutherlang方法[FO]，它计算每个顶点的外部码，并由外部码决定应该对线段进行什么样的操作。简单来说，外部码是由true/false值组成的六元组（如果喜欢的话也可以是0/1），该六元组的每一位值表示点相对于视域体的一个面是否应该被裁剪，如果点相对于那个平面是可裁剪的，那么该线段的外部码就是true，或者1，如果点是可见的，它是false，或者0，例如，如果视域体的边界是$-2\leqslant X\leqslant 2$，$-2\leqslant Y\leqslant 2$，且$3\leqslant Z\leqslant -1$，那么点$(-3,3,-2)$对应的六元组外部码是(1,0,0,1,0,0)。

要对线段进行裁剪，需要计算线段两个端点的外部码，并进行逻辑测试。如果两个外部码的逻辑或OR是0，那么整条线段被保留；如果两个外部码的逻辑与AND不全为0，那么整条线段都被裁剪，如果不能通过这些检测，那么每一个外部码中为1的位就表示线段穿过该平面，就不需要再比较，计算该交点的坐标值和新外部码，把它替换原先的端点，再重复这个过程。

伴随着高速图形硬件设备的发展，传统的裁剪操作对程序员来说没那么重要了，因为裁
47 剪操作被移到了图形卡中，它专门为顶点处理进行优化，速度无疑会更快（对程序员来说更简单）。除非能减少很多几何形状，建议避免向流水线中发送大量数据，把这项工作留给具体的硬件来做。

1.3.8 定义窗口和视口

投影后显示的场景在二维眼空间中，所有物体的坐标都是实数值。但显示空间是离散的，所以，创建图像的下一步是把几何体从二维眼坐标系中转化为整数值的离散坐标系。这需要区分离散的屏幕点和替换实数值的几何点，并引入采样问题，这需要非常谨慎地处理，一般图形API会处理这些事情。实际显示图像的显示空间依赖于定义的窗口和视口。

对于图形系统来说，窗口是观察空间中的一个矩形区域，在这个区域中程序进行具体的绘制，窗口的定义和显示设备的坐标系有关，而且它有自己的内部坐标系，尽管用来绘制的窗口可能占用桌面窗口的空间，但它与桌面显示窗口系统中的窗口是不一样的。我们在书中将非常小心地使用窗口来表示显示图形的区域。图形API提供图形窗口和管理设备窗口的接口。图形窗口中的空间称为屏幕空间，使用在几何流水线中描述的二维屏幕坐标。屏幕空间中使用的最小显示单位称为像素，是图像单元的简称。这个坐标系是相对于窗口定义的，并不是整个显示空间，所以并不随着显示窗口在屏幕上的移动而改变。为了一致起见，我们将在像素坐标中考虑显示空间，因为它们都是和图像相关的。

回忆一下几何流水线的最后一次变换，它是实现从二维眼坐标系到二维屏幕坐标系的转换。为了理解这个变换，需要理解点在这两个对应矩形区域中的关系。这两个矩形区域是不同的，一个在二维眼空间中，另一个在二维屏幕空间中。处理方法同样可以应用在两个矩形空间中对应点的处理，如屏幕空间中的光标位置对应的二维眼空间中、世界空间中和纹理空间中的点。

图1-13显示带有边界和点的二维窗口和视口。假设矩形的左下角坐标值是最小的。在左
48 边的矩形中，X的最小值是$XMIN$，X的最大值是$XMAX$，Y的最小值是$YMIN$，Y的最大值是$YMAX$。在右边的矩形中，X的最小值是L，X的最大值是R，Y的最小值是B，Y的最大值是T。

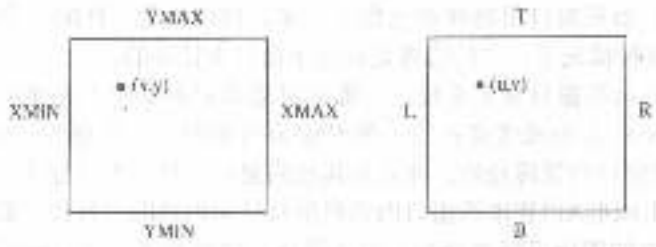


图1-13 与二维窗口（左）中相对应的视口（右）中的点

按照图中的名称，窗口的宽度*WW*和高度*WH*分别表示为：

$$WW = XMAX - XMIN \quad 和 \quad WH = YMAX - YMIN$$

视口的宽度*VW*和高度*VH*分别表示为：

$$VW = R - L \quad 和 \quad VH = T - B$$

由矩形的对比关系可以得到如下两个等式：

$$(x - XMIN)/WW = (u - L)/VW$$

和

$$(y - YMIN)/WH = (v - B)/VH$$

通过这个两个等式，给定任意一对值可以解出对应的另外一对值，如给定*x*，*y*可以解出*u*，*v*为：

$$u = L + (x - XMIN)*VW/WW$$
$$v = B + (y - YMIN)*VH/WH$$

同样，给出（*u*，*v*）的值也可以解出（*x*，*y*）的值。作为一个如何使用这些计算公式的例子，如果（*u*，*v*）坐标值表示在屏幕空间上鼠标单击的点，那么计算得到的（*x*，*y*）表示事件在二维眼空间中的位置。这个计算假设所有屏幕空间上的坐标比例都是实数，实际上是对计算的实数值坐标进行了归整操作，因为它们来自离散的屏幕空间坐标，所以，实际上是对二维眼空间坐标进行了归整操作。

在对图形显示窗口坐标系的讨论中，并没有指定它是如何组织的。图形API会对窗口坐标使用某种约定。窗口可能把它的原点或者（0, 0）放在左上角或者左下角。在前面的讨论中，按照通常的数学习惯把原点定在左下角，但图形设备可能把原点放在左上角，因为它对应内存的最低地址。如果API把原点定在左上角，可以对变量做一些改变：*Y'* = *YMAX*−*Y*，把*Y'*替换*Y*代入原来的等式进行计算。

很多图形系统在从二维眼空间到二维屏幕空间的转化过程中穿插了中间步骤，先把二维眼空间坐标转化到归一化设备坐标（NDC），它是二维空间的实数值，每个方向的取值介于0.0和1.0之间，然后转化为二维屏幕空间。把二维眼坐标映射到NDC，和把NDC映射到二维屏幕坐标都是线性映射。使用NDC可以很方便地处理窗口大小的改变，并且通常由图形设备来处理。大部分的图形API不把这作为单独的一步，所以，本书并不对此作更进一步的讨论。 49

可以把图像显示在窗口的子矩形区域而不是整个窗口中，这个子区域称为视口，视口是图形窗口的一个子矩形区域，它可以限制图像的显示边界。对于任意窗口或视口，它的宽度和高度比称为*高宽比*。一个窗口可以有多个视口，视口甚至可以相交，每个视口都显示自己的图像。把图像映射到视口的计算和前面讨论的计算一样，只不过此时的边界是视口边界，而不是窗口边界。大部分图形系统默认的视口是整个图形窗口。视口通常和它所占的窗口定

义的条件是一样的，如果窗口用物理单元指定，那么视口也是一样的。然而，视口也可能相对于窗口定义，在这种情况下，它的边界是相对于窗口来计算的。

如果图形窗口显示在窗口桌面系统上，那么可能希望和桌面上其他窗口一样对它进行操作：移动它、改变它的大小或者点击它，使它成为当前窗口。管理窗口是一件困难的工作，但图形API可以提供窗口的管理功能，使它和其他的窗口一样。当对包含图形窗口的桌面窗口进行操作的时候，由图形API管理的窗口内容将维持显示内容的一致性。假如改变窗口或视口的高宽比，那么视口的图像可能会变形，因为程序是按照原来的视口画的。这可以通过使程序改变投影来对窗口的调整作出响应。一个程序可以同时管理多个不同的窗口，并对每个窗口进行绘制。每个窗口都会指定一个独一无二的句柄，用来标识那个窗口得到绘制的命令。

1.4 管理视图的其他方面

定义了观看模型的基本特性，一些其他因素也会影响图像的创建和显示。接下来的几章将讨论这些情况。本章只对隐藏面和双缓冲技术进行讨论，因为使用它们可以更加高效地创建图像。

1.4.1 隐藏面

世界上大多数东西是不透明的，所以，我们只能看到最近的物体。然而，这对计算机产生图像来说却是个挑战，因为图形系统是按照人们给它的顺序来绘制物体的。为了创建一幅"显示最近的物体"的图像，需要使用看到场景一些合适的工具。

大部分的图形系统可以使用场景中几何的深度信息来决定哪些物体是离得最近的，并且只绘制物体的前面部分。这个技术称为深度缓冲，如果深度信息是基于场景中的z坐标，那么也称为z缓冲。在深度缓冲中每个像素有一个值，所以它的尺寸和窗口是一样大的，并且它的颜色值是和已经测试的点中离得最近的颜色是一样的。深度值是通过对场景中物体的z值或者眼坐标中视点到场景中点的距离计算得到的，它的值是经过视图变换后的值。每个顶点的z值在投影变换中被保留。

50

当在几何流水线中处理多边形的时候，要保存深度值。所以，当渲染多边形的时候（这将在第10章讨论），需要对每个顶点的深度值进行插值来得到每个像素的深度值。当要画一个新点的时候，把该点的深度值和当前对应像素存储的深度值进行比较，如果新点比当前存储的值更接近，那么用新点的颜色值代替当前像素的颜色值，并记录新点的深度值。否则，直接忽略该点。这个操作可以在硬件中由图形卡完成，也可以在软件中通过简单的数据结构来实现。

在一些图形API的处理过程中，还有一些细节需要了解。首先是对计算机来说，整数的比较比浮点数比较速度更快，所以深度值一般保存为无符号整型，并且分布在视域体的近平面和远平面之间，0表示近平面，最大正整数值表示远平面。当浮点深度值转换为整数的时候，会引起奇异现象（z-fighting）。它会导致离视点相同距离的物体深度值出现不一致。当近平面和远平面相隔比较大的时候，整数转换尤其是问题，因为这个时候整数深度值比相隔近的时候更粗糙。解决这个问题的办法是使近平面和远平面在包含显示物体的空间里尽可能得接近。这使得每一个整数深度单位表示更小的实数，使得两个实数表示同一个深度的可能性更小。另外一个办法是避免在建模的时候使两个物体共面，所以，原本在墙上的物体实际上应该稍微靠墙的前面。

另一个细节是如果直接把z值从三维眼空间转化为深度值，值的分布空间并不是线性的。如果近距离和远距离的差别很大，那么，在z插值过程中的非线性会引起z缓冲的分布更偏向

于近距离的物体。这样，远处的物体就更容易产生z-fighting。一种称为*w-buffer*的缓存用1/z来代替z来存储。因为1/z使插值更线性，这对远距离的物体效果会更好。

也可以使用其他技术来保证只显示场景中确实要显示的部分。例如，先计算模型中每个物体的深度值（和眼睛的距离），并对这些物体进行排序，然后就可以按照从远到近的顺序绘制物体。即先画远处的再画近处的。这种方法可以使近处的物体覆盖远处的物体，使得场景中只显示可见的物体。这种经典方法称为画家算法，它模仿画家使用不透明的颜色创建图像。这种方法只在一些有限的图形系统中广泛使用，有时候它确实比深度缓存有优势，也更快速，因为它不需要对每个像素进行深度比较，并且深度缓存包含本身不能解决的问题，如第6章讨论的模型透明混合。因为画家算法需要知道三维眼空间中每个物体的深度值，所以，如果图像中包含重叠的部分，移动物部分或者移动的视点，它就不那么容易了。这些情况需要一些更成熟的建模技术，例如空间划分，将在第4章讨论。从眼空间中获得深度值将在第2章和场景图一起讨论。 51

1.4.2 双缓存

缓存（例如深度缓存）是用来存储计算结果的一块内存，在图形屏幕上用来存储像素值。如果只用一个缓存，那么它就是颜色缓存。当产生图像的时候，需要逐个像素地写入缓存。因为缓存会自动连续地显示到屏幕上，所以，清除缓存并写入新的图像就可以让观看者看到结果。

大部分图形API允许使用两个图像缓存来保存计算的结果，它们分别称为前缓存和后缓存。因为创建一幅图像需要一些时间，而显示一帧图像的速度却很快，所以只使用一个缓存是不够的，除非只需要创建一幅图像。大部分时候是把图形画到后缓存中，而不是前缓存中。当图像创建完成，把两个缓存进行交换，这样后缓存（包换新建的图像）变成了前缓存，观看者就可以看见新图像。当图形是以这种方式完成的时候，我们称为使用双缓存。这种方法对动画是必需的，因为观看者需要看到的是已经完成绘制的图像序列。这种方法也在其他图形中频繁使用，因为向观看者显示完成的图像会让他们感到更满意。当图像创建完成的时候，一定要进行交换，否则观看者永远看不到结果。

1.5 立体视图

立体视图可以让我们看到一些视图变换的处理过程。它不应该是创建图像的首要目标，因为它需要视图变换的基本经验。我们讨论双目立体视图——要求眼睛能在电脑屏幕前或者打印的图像前集中于一点，当对图像聚焦的时候，它能产生真实三维效果。我们将在第10章和第15章中讨论其他创建立体视图的技术。

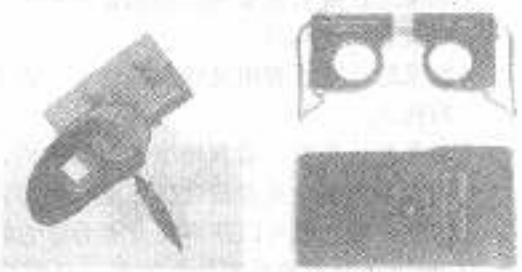


图1-14 两种立体投影仪，老式的（左）和现代的（右，用来查看航空图像）

立体视图从代表左右眼的两个视点计算得到同一个模型的两幅图像，并把它们呈现给观看者，这样观看者左右眼就可以分别看到每幅图像。在经过大脑的视觉处理后把它们混合成一幅图像。以前的立体感投影仪和立体幻灯片播放器等观看系统都是通过同时打印两幅照片来实现。图1-14显示了两种立体投影仪，它们比较复杂，不过可以建一个简化的版本来帮助人们观看立体图像。如果已经拥有或者能够借到一个早期的立体感投影仪，通过使用现代的技术为这个早期的观看设备创建图像是非常有趣的。第11章将会讨论另一个早期用于动画图像的观看技术。

这两幅图像可以分别显示在屏幕上同一个窗口的两个视口中，并以此来创建立体视图，如图1-15所示。为了达到这个目的，需要区分两个视点。这两个视点位于与观看向上方向垂直的平面上，并相隔一定的距离。最简单的方法是把一个坐标轴定义成向上方向（可以是z轴），取另一个相垂直的轴（可以是x轴），并把所有的视点与该轴对齐。我们之所以要这样定义，是因为当观看包含两个变量的等式定义表面的时候，更习惯于把x，y看成自变量，把z看成因变量。第9章将会对此进一步讨论，并详细介绍表面处理。

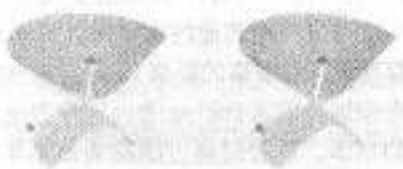


图1-15 两只眼睛分别看到的立体图像，参见彩图

52 然后，需要定义两个视点的距离，根据观看者眼睛的距离来定义，把每个视点从中心点移动该距离的一半。这使得每只眼睛更容易集中在各自的图像上，并使大脑的聚焦系统能够创建混合的立体图像。同样，保持整个显示区域的足够小也是非常重要的，因为这样可以使两幅图像的中心距离不至于比观看者眼睛的距离更大，可以使目光更好地关注于各自的图像。

相当多的人因为生理限制，不具备进行这种立体视图所需的聚焦能力，一些因为有聚焦问题而不能把眼睛聚集到一点来创建合并的图像；还有一些不能在屏幕附近看到聚焦发生的点。所以，如果不能正确得到立体图像对的空间，或者视点没有对齐，又或者两边的刷新率
53 不一致……都会导致立体视图不能很好地工作。即使立体视图是正常工作的，也有可能有人可以看到聚焦后的图像，有些却不能。

1.6 视图变换与视觉交流

为场景选择合适的视图在创建高效的视觉交流中是非常关键的。向观众展示信息的时候，必须把他们的视线集中到该让他们看到的。有很多方法可以实现这样的目的，这里仅讨论其中的一些方法。

- 如果想让观看者看到整个内容中的某些细节，那么首先创建一幅包含整个内容的大图像，然后对图像进行缩放操作来查看细节。这个效果是预先确定的，或者是用户可以控制的。
- 如果想让观看者看到图像的特别部分作为移动场景的一部分是如何工作的，那么应该把那部分固定在观看者面前，并移动其他部分来实现移动场景。
- 如果想让观看者全方位看到整个模型，那么应该绕着模型移动眼睛，这可以通过用户控制或者视点移动。
- 如果想让观看者跟随特殊的路径，或者通过模型移动物体，那么应该在模型中建立移动的视点。
- 如果想让观看者看到模型的内部结构，那么应该建立穿过模型的裁剪平面，让观看者看到内部细节，或者改变颜色混合的方式，使得模型结构中前面部分看起来部分或者完全透明，那样就可以使得观看者看穿它们。

不管想让观看者看到什么效果，必须非常谨慎地设计图像，并且注意观看者如何看到图像，这样可以避免让他们看到不该看的部分。

1.7 在OpenGL中实现视图变换和投影

以下OpenGL代码片段包含了本节讨论的大部分内容。它可能来自一个单独的函数，也可能来自多个函数的组合。在前一章OpenGL程序例子的结构中，我们建议视图变换和投影操作

应该分开，第一部分放在display()函数的顶部，然后在init()和reshape()函数的结尾部分放第二部分。

```
// 定义场景的投影
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);
// 定义场景的模型变换环境
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    // 视点，观察方向，向上方向
    gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

54

可以看到代码段分为两部分，而且这两部分非常相似。首先使用函数glMatrixMode()来选择操作模式（投影或模型视图），然后在定义具体的操作前先调用glLoadIdentity()函数，该函数对紧接的操作初始化（设置操作的单位矩阵），最后调用具体的函数来设置投影或视图变换操作所需的信息，而不与过去的操作数据冲突。

1.7.1 定义窗口和视口

在前一章的例子中，通过函数来定义窗口，这些函数对窗口的大小和位置进行初始化，然后创建窗口。为了使得API能够工作在多个平台上，特意对程序员隐藏了管理窗口的细节。在OpenGL中，创建窗口最简单的方法是使用GLUT工具包，它定义了很多OpenGL中依赖于系统的部分。这些函数通常在main()中调用。

```
glutInitWindowSize(width,height);
glutInitWindowPosition(topLeftX,topLeftY);
thisWindow = glutCreateWindow("Your window name here");
```

glutCreateWindow函数返回的是整数值thisWindow，它是窗口的句柄，可以通过它把窗口设置为活动窗口，然后在它上面进行绘制。这可以通过glutSetWindow函数来实现：

```
glutSetWindow(thisWindow);
```

它把句柄为thisWindow的窗口设置为当前窗口。如果想要获得当前窗口，可以通过glutGetWindow()函数来实现，它返回当前窗口的句柄。在任何情况下，在进入事件循环前，没有窗口是激活的，如前一章所述。

视口可以通过glViewport函数来定义，它指定了窗口中用来显示的左下角坐标和右上角屏幕坐标。如果定义的视口比图像窗口小，那么可以在程序的初始函数中调用这个函数：

```
glViewport(VPLowerLX, VPLowerLY, VPWidthX, VPHeightY);
```

在接下去立体视图的例子中，可以看到创建属于同一窗口的用于显示两幅独立图像的视口。

1.7.2 改变窗口的形状

当窗口被创建或者移动了位置，或者改变了大小的时候，称窗口改变了形状。这些操作可以很方便地用OpenGL处理，因为当窗口改变形状的时候，计算机会产生一个事件和事件回调。第7章将会详细讨论事件和事件回调函数，但改变窗口形状的回调函数是通过glutReshaperFunc(reshape)函数来注册的，它的参数声明为reshape(GLint w, GLint h)，当改变窗口形状事件发生的时候，不管是否要新产生图形，都会调用回调函数。

55

当改变窗口形状的时候需要定义投影、定义视图变换的环境、更新窗口中视口的定义，或者是一些和display()函数有关的动作。改变形状回调函数得到改变后窗口的大小，可以使

用它控制图像在窗口中的显示。例如，如果使用透视投影，定义投影的第二个参数是高宽比，可以用回调函数中得到的大小来设置它，如：

```
gluPerspective(60.0,(GLsizei)w/(GLsizei)h,1.0,30.0);
```

这使得投影能够适合新的窗口形状并保留原先场景的属性。另外，如果想以固定的高宽比来显示场景，那么以想要的高宽比定义视口。例如，如果想定义一个正方形视口，那么只要取宽度和高度中的最小值，在窗口的中心定义视口，然后在视口中进行所有的绘制。

任意视口都可以定义在reshape()回调函数里。因为这样可以在改变窗口大小的时候重新定义，或者在显示函数中定义，因为这样可以使用改变后窗口的尺寸。视口应该相对于窗口的大小和尺寸来设计，使用reshape函数的参数。例如，窗口以整数值(width, height)来定义尺寸，如果视口定义在窗口的右半部分，如立体成对例子，那么它的坐标是(width/2, 0, width/2, height)。那么窗口的高宽比是width/(2*height)。如果窗口改变了大小，那么应该调整视口的宽度，它不应该超过新窗口宽度的一半，这样才能保证占有窗口的一半，或者，它是窗口的高度乘以高宽比，来保证图像不失真。

1.7.3 设置视图变换的环境

要设置视图变换的环境，必须用GL_MODELVIEW矩阵来定义单位矩阵，然后指定两个点和一个向量来定义视图变换的环境。两个点是视点和观察的中心（所看方向上的点），向量是向上方向向量——投影为图像的垂直方向。唯一的限制是视点和观察中心不能是同一个点，并且向上方向向量与连接视点和观看方向的点不平行。它的示例代码如下所示：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
//   视点  观察中心  向上
gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
```

56

gluLookAt()函数可以被reshape()函数调用，也可以放置在display()函数中。视图变换的函数参数可以通过变量传递。通常，我们更倾向于在display操作的开始就包含gluLookAt操作。随着程序的运行，将使修改视图更简单，而且可以使用交互技术来改变视图。

gluLookAt(...)函数定义了把视点从默认位置和方向进行变换，这在前面已经讨论过。这些操作的结果跟直接以如下形式调用gluLookAt()函数的结果是一样的

```
gluLookAt(0., 0., 0., 0., 0., -1., 0., 1., 0.)
```

如果通过把默认的位置变换到想要的位置来放置视点，那么就定义了一系列针对视点的变换。图形API支持的变换操作将在第2章讨论，定义视点的操作如下：

1. 绕Z轴旋转，使得向上方向在观察平面上的投影和Y轴对齐；
2. 进行缩放，使得观察中心位于沿着Z轴负方向上一定距离的点；
3. 移动变换，使得观察中心位于原点；
4. 分别绕X和Y轴旋转，把视点放置在相对观察中心的正确位置；
5. 移动变换，把观察中心移到正确的位置。

正如在下一章讨论变换中看到的，按照这种顺序是非常重要的。

为了使场景符合所定义的视图要求，需要通过视图变换来调整整个场景，使得视点和方向在标准位置。视图变换通过把放置在原点的视点变换到所要的位置的逆变换来实现。函数有这样一种属性，两个函数乘积的逆是两个函数逆的交换乘积，对任意函数f和g，有$(f \cdot g)^{-1} = f^{-1} \cdot g^{-1}$。把上述的五个操作进行逆操作，并按照相反的顺序来应用。因为这些操作必须作用于场景中所有的几何体上，所以它必须在所有的几何都定义完毕后才能指定，所以，

gluLookAt(...)函数应该是首先出现在display()函数中的其中之一，它的变换操作步骤如下：

1. 把观察中心变换到原点；
2. 绕X，Y轴旋转，使得视点位于Z轴负方向；
3. 把视点变换到原点；
4. 进行缩放，把观察中心变换到（0.0, 0.0, −1）；
5. 绕Z轴旋转，使得向上方向和Y轴对齐。

在第2章中，当需要把视点（眼睛放在场景中物体的相对位置）作为建模的一部分来控制的时候，我们需要了解这些步骤。 57

1.7.4 定义透视投影

透视投影首先要指定工作在GL_PROJECTION矩阵下，然后设置单位矩阵，然后指定定义透视变换的各种属性。按照如下顺序：

1. 观看的范围（以弧度表示的角，表示观看范围的宽度）；
2. 高宽比；
3. zNear的值（从观看者到近裁剪平面的距离）；
4. zFar的值（从观看者到远裁剪平面的距离）。

这个过程很简单，设置多次后，就会发现这很自然。例如，下面的代码

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluPerspective(60.0,1.0,1.0,30.0);
```

这定义了一个透视视图，60度角的观看范围，三维眼空间中相等的宽度和高度，距离眼睛1个单元的近裁剪平面和距离眼睛30个单位的远裁剪平面。

通过设置不同的观看范围角得到的图像效果就不一样。如果减少观看范围角的值，那么更能产生远焦距镜头效果。如果增加角的值，那么更能产生广角镜头效果。

透视投影也可以通过glFrustum()函数来定义，它的参数包括定义可视物体视域体的六个参数，形式如下：

```
glFrustum(left, right, bottom, top, near, far);
```

实际使用中通常更偏向于gluPerspective()函数，所以，不打算进一步讨论glFrustum()函数的使用，把它留给有兴趣的读者。

1.7.5 定义正交投影

除了函数参数以外，正交投影的定义跟透视投影很相似。定义正交投影的视域体，只要定义如图1-3所示的边界，OpenGL会处理其他的工作。

```
glOrtho(xLow, xHigh, yLow, yHigh, zNear, zFar);
```

观看空间也是左手坐标系，所以，zNear和zFar分别指近平面和远平面与XY平面的距离，根据OpenGL中定义观看环境的方式，这些距离都是在视线方向上。zNear和zFar表示距离视点的整数值，这与考虑透视投影时的情景是一样的。

1.7.6 隐藏面的处理

在前一章中介绍了一个OpenGL的实用程序，并且在main()中通过使用GLUT的glutInitDisplayMode()函数定义显示的属性。如果在这个函数参数中包括GLUT_DEPTH，那么就可 58

以处理隐藏面：

```
glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
```

这里必须启用深度测试。启用是OpenGL的标准特性：许多系统的功能需要通过调用glEnable函数启用后才能用：

```
glEnable(GL_DEPTH_TEST);
```

接着，就开始使用深度测试，不需要对它进行干涉。当距离近的物体在后来显示，那么它将覆盖以前的物体；否则就不进行绘制。

深度测试启用后，绘制过程就会自动进行。为了更加高效地使用它，应该掌握使用它的原则。OpenGL使用整数值进行深度测试，而不是浮点数，在深度测试中引入了间隔尺度。当每一个像素被深度测试的时候，它的z值转化为无符号整数，表示占系统最大无符号整数值的比例。该比例为

```
(z-zfront)/(zback-zfront)
```

与在投影中指定的一样，zfront表示近裁剪平面的深度，zback表示远裁剪平面的深度。OpenGL的深度测试具有间隔尺度，可以避免Z-fighting。默认的测试是对于每个点，如果它的z值比0大，但比当前存储在深度缓存中对应位置的值小，那么就把这个点的值记录下来，通过glDepthfunc(value)函数来更改，value是个符号常量，本书只使用默认的深度测试，如果想进一步了解，可以参考OpenGL的指南。

如果想关闭深度测试，可以使用glDisable()函数，就跟使用glEnable()函数一样。留意在立体视图的示例代码中，启用和禁用裁剪平面的时候调用的enable和disable函数。

1.7.7 设置双缓存

双缓存是个标准工具，可以通过在glutInitDisplayMode()函数中使用GLUT_DOUBLE参数来设置双缓存。设置双缓存表示在绘制的时候将同时使用前后两个缓存。前缓存的内容用来显示，后缓存用来临时保存处理的结果，当完成绘制的时候，通过在display()函数中调用glutSwapBuffers()函数来把前后缓存进行交换，使得后缓存的内容被显示。双缓存的另一个好处是通过一些技术来检测后缓存中的内容，而不用把它和前缓存进行交换。所以，后缓存中所做的工作不一定能看见。

1.8 实现立体视图

本节讨论双目立体视图的实现。我们将产生一个模型的两个视图，正如它们是从两只眼睛分别看到的。把它们显示在一个窗口的两个视口中。这两幅图像通过对模型进行整体变换，观察者通过把两只眼睛聚焦在各自的图像上，然后对它们进行混合来获得一幅立体图像。

59

实现立体视图是非常简单的。首先，创建一个窗口，该窗口的宽度是高度的两倍，而它的宽度是两只眼睛距离的两倍。在分别占据窗口左右半部分的视口中两次显示模型。每一个显示都是一样的，除了左右两部分中分别代表左右眼睛的视点位置。可以通过窗口初始化函数来创建包括这两个视口的窗口：

```
#define W 600
#define H 300
width = W; height = H;
glutInitWindowSize(width, height);
```

宽度的初始值是高度的两倍。两次观看设置在x方向上距离原点ep长度，以z轴为向上方向看向原点，然后设置两只眼睛在y方向上相差一定距离。最后在display()函数中定义左、

右视口：

```
左边视口
glViewport(0, 0, width/2, height);
    ...
//  视点，观察中心，向上方向
    gluLookAt(ep, -offset, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    ... 实际绘图的代码
    ...
//  右边视口
    glViewport(width/2, 0, width/2, height);
    ...
//  视点，观察中心，向上方向
    gluLookAt(ep, offset, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    ... 同样是处理绘图的代码
    ...
```

这个例子可以很好地响应reshape(width,height)操作，因为它使用窗口的尺寸来设置视口的大小。如果窗口改变后它的高宽比不是2:1，那么它很容易产生变形的问题。我们把窗口高宽比改变后如何创建正方形视口的工作留给读者来实现。

1.9 小结

本章主要讨论了视图变换和投影，它们是计算机图形学的基础并且在图形编程中是必不可少的，视图变换的决定因素是视点、观察参考点和向上方向，透视投影的决定因素是观察的宽度和高度（即观察的角度和高宽比）以及远近裁剪平面，正交投影的决定因素是观察空间的宽度和高度以及远近裁剪平面。

视图变换和投影操作可以用相关的OpenGL函数来实现，与它们一起可以用OpenGL函数来实现的还有窗口和视口管理、双缓存、深度测试和一些通常的裁剪操作。

利用这些概念和操作，可以写一些图形程序对已经建模完成的场景进行操作，并实现诸如立体视图的技术。在接下来的几章，将会介绍一些通用的建模技术作为对本章的扩展，使得读者能够写出更加通用、功能更强大的图形程序。

1.10 本章的OpenGL术语表

本章新介绍了一些OpenGL函数，包括一些新的GLU和GLUT函数和系统参数。这里列出它们并做简单的解释。如果读者想更加详细地了解，那么建议您看OpenGL的文档。

OpenGL函数

glDepthFunc(parm)：通过符号参数指定计算函数，该函数决定顶点是否替换当前绘制缓冲中的顶点。通常使用本章讨论的深度测试函数是GL_LESS。

glDisable(parm)：跟前一章看到的glEnable()函数一样，通过符号参数表示禁用OpenGL的某些功能。

glFrustum(left, right, bottom, top, near, far)：通过参数左和右、上和下、前和后裁剪平面来指定透视投影的视图平截头体。

glGetFloatv(parm, *params)：通过符号名称指定参数要返回什么值，并指定保存该值的变量（用索引）。返回的值可能是标量，也可能是数组，这依赖于符号名称。

glLoadMatrixf(array)：把由array参数指定的矩阵（必须包含16个元素）复制到当前的激活矩阵——可以是投影矩阵或者模型视图矩阵，这依赖于用glMatrixMode()函数最后设置的模式。

glOrtho(left, right, bottom, top, zNear, zFar)：通过定义六个裁剪平面定义正交投影的视域体。

glViewport(lowX, lowY, width, height)：通过指定左下角位置、宽度和高度在图形窗口中定义

60

视口，所有这些屏幕坐标都是整数。

GLUT函数

glutGetWindow()：返回当前活动的窗口值。

61 glutSetWindow(winName)：设置当前活动窗口值（通常为标识名）

参数

GL_DEPTH_TEST：由glEnable()函数使用，它用来启用深度测试的符号名称

GL_MODELVIEW_MATRIX：保存当前模型和视图变换值的矩阵

GL_PROJECTION：由glMatrixMode()函数使用，它用来指定当前激活的矩阵是投影矩阵的符号名称

GL_PROJECTION_MATRIX：保存当前投影变换值的矩阵

1.11 思考题

这些问题用于加深对图形环境中视图变换和投影问题的理解。它们有助于看到本章介绍的定义视图，应用投影以及生成主题的效果。

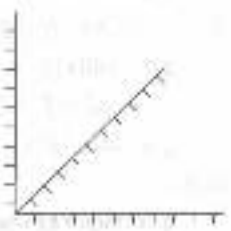
1. 寻找一个合适的环境，检查环境的视差依赖于视点和视图方向的方式。注意一下当视点移动的时候，物体是如何在其他物体的前后变化的；当移动视图方向的时候，物体是如何从一边进入视图，又从另一边出去的。如果透过纸或者纸板看，可能更有帮助。
2. 受眼睛的工作方式限制，我们并不能看到场景的正交视图。但是，如果把眼睛朝向固定的方向，然后在场景中在固定方向移动，那么眼前的视差就近似于正交视图。针对一个熟悉的环境，尝试一下这种方法，看看能不能在每个点建立所看到场景的缩略图，然后把这些缩略图综合进一幅图像里。
3. 考虑用画家算法来建立场景的视图，按照距离眼睛由远到近的顺序记录物体。接着移到场景中的另一个位置，想像一下按照在另一个视点记录的物体顺序来绘制物体，是不是有些物体顺序混乱了？哪些距离远的物体把近的物体覆盖了？为了看到场景的这种改变，应该把视点移到哪里？对于画家算法所需的计算，可以得出什么结论？
4. 想像一下有个平面穿过场景的中间，导致平面另一边的物体都将不进行绘制。如果让这个平面穿过某些物体，导致物体部分可见部分不可见，环境的视图看起来像什么？如果把平面两边的可见性交换一下，会发生什么？
5. 讨论选择一个合适的视点来展现场景，让观看者得到最好的信息时存在的一些问题。考虑一些诸如关键信息是否可见，场景中的关系是否正确表示，或者一些关键问题是否很难由视点区别开来等问题。

62 还要考虑有助于决定图像应该是静态的，或者应该是根据视点在场景中移动提供不同的视图等问题。

1.12 练习题

在这些练习中，需要进行一些计算，这些计算包含在建立场景的视图和定义从场景到屏幕的映射中。

1. 考虑用45度角的观看范围、1.0的高宽比、距离视域体的近平面1.0、远平面为20.0来定义一个标准的透视视图变换。对于近平面上的点$P=(x,y,1)$，每到棱台体中的线段参数方程，这条线段上所有的点都投影到点P。提示：这条线段经过原点和点P，这两点定义了包含这条线段的直线。使用视域体的近平面和远平面为参数方程确定线段的端点。



2. 在纸上建立如图所示的X-Y-Z网格。按照惯例，X轴向右，Y轴向上，Z轴指向纸面。

a. 在问题1的场景环境中，定义网格的单位长度，把环境中的物体放在有标准坐标的网格中。在这个例子中，把所有的物体都放入非负坐标的空间中（二维笛卡儿坐标系的第一象限），这样对坐标的处理更方便。

b. 在同一个空间中定义视点的位置和方向，观察在那个视图定义下可见的物体，这只原来的空间中看可视结果是否准确。如果不是，检查一下引起不准确的原因。

3. 在前面数字化建模的环境中，把视点放在（X,Z）平面的中间（从左到右，从前到后的中间），并且视点沿着该平面看向原点。计算坐标空间中每个点相对于眼坐标系的位置，并设法得出一个通用的过程进行计算。

4. 图1-3所示的模型是通过如下简单的代码段定义的(省略了一些OpenGL的细节)，这可能在display()函数中可以找到：

```
glPushMatrix();
   glTranslatef(1., 1., 1.);
   glScalef(1., .5, .5);
   cube();
glPopMatrix();
glPushMatrix();
   glTranslatef(-.5, 1., 1.);
   glScalef(.5, .5, .5);
   Sphere(1.); // parameter is radius
glPopMatrix();
```

请读者尝试下能否得出这个建模代码的功能。定义模型的多个视图，并且在实际有用的程序实现前，想像一下在每个视图中模型是怎样的。本书配套的光盘中包含该图的实现代码，可以对它进行修改，实现各种想法。 63

5. 编写一个程序，在视口中显示一个简单的模型，该视口占据图形窗口一半的空间。当空闲回调函数不断改变窗口中视口的位置的时候，把它绘制在视口上使它移动。注意，当移动视口的时候，要保证整个视口都在窗口里面。

1.13 实验题

1. 在前一章，我们看到了展示长类上热传递源程程序的完整代码，然后在练习题中还讨论了当窗口改变时程序的行为。在那个程序的reshape()函数中进行投影操作，建立其他显示方式：创建正交投影，创建总是可以使图像适合窗口的透视投影。

2. 在本章可以看到对深度测试使用glEnable(...)函数，还可以看到深度测试在创建图像时的效果，即离视点近的物体遮挡了离视点远的物体。在本实验中，用glDisable (GL_DEPTH_TEST)函数禁用深度测试，对场景进行绘制，这个场景应该跟使用深度测试时候的场景是一样的。从多个不同点观看场景，然后得出关于同样的场景，为什么不同的视点会得出不同的图像的结论。

在接下去的两个实验中，我们将会用简单的线段绘制图1-5所示的房子模型，当然我们鼓励读者用更有趣的模型来替代。以下给出创建以原点为中心的房子函数代码，这个函数在下面给出，以方便读者着手，它可以在display()函数中调用。绘制模式GL_LINE_STRIP将在第3章描述，它绘制顺序连接的线段，这些线段从第一个顶点开始，然后依次到达每个顶点，直到遇到了glEnd()函数。

```
void drawHouse(void) {
  point3 myHouse[10]= { {  1.0,  1.0,  2.0 }, { -1.0,  1.0,  2.0 },
                        {  0.0,  2.0,  2.0 }, {  1.0,  1.0,  2.0 },
                        {  1.0,  1.0,  2.0 }, { -1.0, -1.0, -2.0 },
                        { -1.0,  1.0, -2.0 }, {  0.0,  2.0, -2.0 },
                        {  1.0,  1.0, -2.0 }, {  1.0,  1.0,  2.0 } };
int i;
glBegin(GL_LINE_STRIP);
  for (i=0; i<5; i++)
    glVertex3fv(myHouse[i]);
  glVertex3fv(myHouse[0]);
```

```
glEnd();
glBegin(GL_LINE_STRIP);
  for (i=0; i<5; i++)
    glVertex3fv(myHouse[i+5]);
  glVertex3fv(myHouse[5]);
glEnd();
for (i=0; i<5; i++) {
  glBegin(GL_LINE_STRIP);
    glVertex3fv(myHouse[i]);
    glVertex3fv(myHouse[i+5]);
  glEnd();
}
```

64

3. 编写一个使用这个函数绘制房子或者其他场景的程序，当视点绕着场景移动的时候，程序会对视图产生什么影响（假设总是看向原点（0,0,0））？同时为这个场景定义透视和正交投影，比较使用不同的投影创建的图像的差别。
4. 在上述投影中，使视点固定，把视图参考点绕着场景改变，使每次看的方向是不同的，然后绘制房子。注意视图参考点绕着场景移动产生的效果。
5. 对上述同样的程序，固定视点，用透视视图的其他参数进行实验：远近视图平面，视图的高宽比以及投影视图的范围。对这几个参数中的每一个，需要注意在以后创建更加有用的图像的时候，能够对它们进行控制。

在接下去的两个实验中，将要考虑本章描述过的投影和视图变换矩阵。关于这些矩阵的更详细描述，参看第4章关于矩阵与变换的内容。

在OpenGL中，通用的glGet*v(...)查询函数返回很多不同的系统参数值。这可以得到本章讨论的一些变换矩阵的值。尤其是，我们可以得到对任何投影和视图定义的投影变换和视图变换矩阵的值。在接下去的两个问题中，我们来研究这个可能性。重新得到的变换将以4×4矩阵表示。所以，需要写个函数来显示4×4矩阵，这样才能清楚地看到矩阵中的元素。实际的变换值是位于矩阵前三行和前三列的3×3子矩阵中。

6. 为了得到投影变换的值，可以使用如下函数：

```
glGetFloatv(GL_PROJECTION_MATRIX, v)
```

v是长度为16的浮点数组，可以通过如下方式定义：

```
GLfloat v[4][4];
```

为了查看投影的矩阵，不管是透视投影还是正交投影，在定义投影后任何时候，只要插入上面的函数调用，打印返回的矩阵就可以了。如果是正交投影，应该可以通过矩阵的元素来辨别投影的参数；如果是透视投影，就比较难了，我们应该从本章简单讨论过的投影矩阵开始。在本实验中，对这个过程返回的投影定义的矩阵进行操作，改变它的值，然后通过下面的代码用新矩阵重置投影变换：

```
glMatrixMode(GL_PROJECTION);
glLoadMatrixf(v);
```

65

这会导致重新定义投影变换，它的变换矩阵是v。我们可以观察到原来的投影和新的投影的差别。

7. 为了得到视图变换的值，可以先得到OpenGL模型视图矩阵的值，它是视图变换和模型变换的乘积。所以，如果定义了视图后还没有定义任何模型变换（即还没有应用任何缩放、旋转或者平移操作），可以通过如下函数来得到：

```
glGetFloatv(GL_MODELVIEW_MATRIX, v)
```

v和上面的定义一样。这个矩阵比较复杂。如果只是对默认视图设置一些简单的参数（例如只改变视点和向上向量），那么应该和前一个实验一样，可以从视图定义的每个部分区分出视图变换矩阵的各个组成。对这个过程返回的视图变换定义的矩阵进行操作，改变它的值，然后调用上述过程用新矩阵重置模型视图矩阵，不过用GL_MODELVIEW_MATRIX参数代替GL_PROJECTION_MATRIX，然后观察原来的视图和新视图的差别。

66

第2章 建模原理

建模是图形流水线的第一步，这是一个讲述用图形工具创建基本几何元素，并用这些几何元素构建场景的过程。这一章对于读者理解如何使用标准的基于多边形的方法来建立模型是至关重要的，读者可以学会从简单对象的建模到相当复杂并且具有层次结构的对象的建模方法。很多图形API都基于图形的多边形表示方法，然而，在计算机图形学中还有其他的一些建模方法，其中有的方法涉及到更加复杂的建模技术（它们不在本章讨论）。如何用广泛的光线跟踪技术将在第14章中讨论。

本章分成四个部分，分别介绍图像生成过程中的四个不同建模阶段。首先介绍在世界坐标系内直接建立简单几何模型：在世界坐标系内直接定义每个顶点的坐标。这种方法非常直接，但是对于复杂物体的建模要花费很多时间，因此我们也讨论从不同种类的建模工具中导入模型的方法。

第二部分描述建模的下一个步骤：在物体自己的模型空间中的标准位置生成对象，使用模型变换方法把物体以其大小、其方向和其位置放在世界坐标系中。这个步骤可以让用户生成简单模型的集合，并加以使用，通过标准模型变换，用户可以在自己的场景中生成通用的模型组件。这些变换对于在场景中的移动也是非常关键的，因为用随着时间变化的参数来移动部分场景，比如物体、光源、视点等操作都是非常典型的。这可以让用户扩展建模来定义具有时间变化概念的动画。

以上两个部分覆盖了建模的概念，但是还没有给出使用图形API实现建模的细节信息。在第3章中，读者可以学到如何使用OpenGL实现上述概念，同时也可以看到一些例子。

第三部分讨论在生成有效的视觉交流过程中建模方法的作用。它给出了用户可能会用到的不同类型的建模方法的例子，强调交流科技观念，关注不同形状与大小的物体的建模，并且给出了一些例子。另外，还讨论使用标签和图例作为建模对象向读者传达图像中的文字以及其他的信息。

最后的一部分将介绍对于组织复杂图像十分重要的工具——场景图。场景图提供了一种统一的方法来定义所有的物体以及在场景中所有的变换方法，同时指定了它们之间的相互联系和表达的关系。借助于场景图用代码来实现建模所需要做的工作。这个概念让建模过程变得非常直接。场景图对于层次式的建模而言格外有价值，通过不同物体的组合来设计一个对象。层次式的物体可以让用户模拟出实际物理的组装，并且开发出模型结构，比如物理机器。场景图也可以让用户建立独立的组件，以及相对于其他组件移动的结构，这种结构用前面定义的基础法则来实现将是非常困难的。

学习本章，需要用户理解简单三维几何，了解如何在三维空间里定义几何点，同时需要有足够的编程经验来写出代码调用API函数完成指定任务。另外，用简单数据结构（比如堆栈等）设计程序以及在三维空间内组织物体的能力也很重要。读者学完这一章以后，可以组织一个基于简单模型的场景几何体，用模型变换的方法把这些几何体组合起来。与此同时，用户还能够用场景图的方法生成复杂且具有层次结构的场景，并且能够用图形基元的方式表达一个场景图。

这一章中包含了一些代码的片断，可以帮助读者了解概念是如何通过代码实现的。这些

代码片断使用C的语法，看起来非常类似OpenGL，但实际上却不是。在第3章中才讲述OpenGL的建模函数，并用OpenGL实现本章的所有概念。

2.1 简单几何建模

计算机图形学研究几何体在计算机中的定义、处理与显示。本书讨论的几何体是简单的三维体，所以用户必须要用合适的API工具来设计它。很多图形API是基于多边形的，这表示用户只能使用一些简单的图形学基元，比如点、线段、还有多边形来构建场景。

编程工作首先是确定对象的建模空间，即坐标系由原点和x，y，z三个方向构成。定义物体即指定点或顶点的坐标值，可通过键入、或者计算机辅助设计系统、或者数学函数定义的方法来实现。第9章给出了一些例子。在坐标系中定义一个物体要用到模型坐标，模型坐标是
68 通过常量定义或者用从算法中计算出的结果来指定每个点。如下代码所示：

```
vertex(x1, y1, z1);
vertex(x2, y2, z2);
...
vertex(xN, yN, zN);
```

因为每一个物体可以在它的坐标系中设计，因此，场景的不同部分可以定义在不同的建模空间中。为了把所有的物体集成到一个单独完整的3D世界空间内，就必须使用模型变换的方法。模型变换就像本书中描述的所有变换方式一样，在保持基本的几何关系的基础上变换物体的函数。这些函数通常包括旋转、平移和缩放等，在图形系统中可以直接获得。这些都是计算机图形学中的基础变换，旋转操作绕着一个固定的轴旋转一定的角度，平移操作对于每一个点的坐标移动增加固定的数值，缩放操作将每个点的坐标乘以一个固定的值。所有的变换都可以表达成矩阵。因此，在图形API中对矩阵的讨论几乎意味着涉及变换的操作。

模型变换通过实施一系列简单、标准的变换操作来完成。每一个变换都作用在它所见的几何体上，遵从函数的结合律，元代码表示如下：

```
transformOne(...);
transformTwo(...);
transformThree(...);
geometry(...);
```

transformThree作用在最初的几何体上，transformTwo函数作用在上述变换的结果上，同样transformOne函数作用在二次变换结果上。用t1,t2和t3依次表示以上三个变换函数，对于函数的组合，应用结合律：

```
t1(t2 (t3 (geometry))) = (t1 * t2 * t3) * (geometry)
```

以上表明，在变换乘积的应用中，首先是作用最右边的变换。这个规律对于在场景中操作物体时进行整体的理解是非常重要的。

如果场景中的各个物体随着时间的变化做不同的变换，就可以生成动画。比如，在刚体动画中，各帧中物体的模型转换进行不同的平移变换产生整体移动效果。

2.2 定义

当我们讨论建模的时候，需要介绍一些常用的术语。我们把建模视为定义场景中物体的过程，建立这个场景就是为了生成图像。同时，对于图像一系列的商用程序提供建模的方法，有多种高层次的工具对场景进行建模。然而，程序员（特别是初学者）会用定义基本几何体
69 的方法来建立自己的模型，这样可以控制建模的整个过程。

我们建模用的是简单的有x，y，z三个标准坐标的欧几里得三维空间。图2-1给出一个点、

一条线段、一个三角形、一个多边形以及一个多面体。这些都是计算机图形学中的基础元素，在本书中我们将经常用到。在这个空间中，一个点就是三维空间中的一个简单位置，有时候称作顶点。它的坐标是由实数组成的三元组(vx,vy,vz)。点在屏幕中通过点亮那个位置的一个像素而得到显示。需要绘制点的时候，用户就要指定这个点的坐标，通常是在三维空间内。图形系统的API就会计算在屏幕中这个点的坐标位置，然后驱动系统点亮那个屏幕像素。一个点通常在屏幕中表达成一个正方形，而不是一个圆点，如图2-1所示。



图2-1 一个点、一条线段、一个三角形、一个多边形和一个多面体

一条线段由两个端点组成，因此绘制一条线段首先需要定义两个点（或者称为顶点）。同样，这些点都要定义在三维空间内，图形API计算它们在屏幕中的位置，然后通过计算两点之间可以最好表示该线段的像素，点亮它们来绘制中间的部分。

多边形是一个空间区域，位于一个平面内并且被线段所包围。它由点的序列组成（称为多边形的顶点）的线段作为边界。一般假定多边形位于一个平面内，但是，如果顶点是三维空间的点，也许上述假定就不对了。因为三角形也是最简单的（同时也是最有用的）多边形，极大多数多边形的绘制实际上是通过绘制三角形来实现的。当需要绘制多边形的时候，用户首先需要指定想绘制的多边形，同时要确定顶点的坐标序列。图形系统就会计算在这个多边形内部的点的像素位置，然后驱动硬件去点亮那些屏幕像素。

多面体是由多边形所包围的三维空间区域，它的边界称为多面体的面。多面体通过指定面的序列来定义，每个面都是一个多边形。在三维空间内定义面，如果超过了3个顶点就不能保证是处于一个平面的。多面体通常需要定义三角形的面。三角形可以保证处于同一个平面内，因为三个点可以决定一个平面。在第6章介绍光照和阴影的时候，我们会看到访问多边形的每一个面的过程中，顶点的顺序是非常重要的。

在生成一幅图像之前，用户必须通过一些建模过程生成图像中的各个物体。图形编程之初最为困难的、或者最消耗时间的就是生成图像中的模型，难点之一是设计物体对象的本身，这个工作可能需要用户通过手工操作勾勒出图像的轮廓，以便能够正确地定义出顶点的坐标。也可以通过其他技术（比如分析计算）来定义顶点坐标。另外一个难点就是用一种合适的数据结构来输入点的数据，同时需要写代码将这些数据解释成点、线段、或者实型中的多边形。但是直到用户得到了那些点和它们之间的关系正确的时候为止，否则，用户不能获得正确的图像。

图形API也提供由简单对象组成较为大型复杂的物体，如离散的对象集合（比如点、线条带、四边形或者三角形）或者是点的连接对象集合（比如线段、四边形条带、三角形条带、或者三角扇形）。这里用到了称为几何压缩的概念，即定义一个几何物体所需的顶点数量比通常情况要少。下面展示了建立模型的技术指令表。

首先我们需要介绍另外一种方法来指定模型顶点。把三维空间视常嵌入在四维空间中，由于四维空间非常难理解，让我们先来考虑二维空间嵌入在三维空间中的简单情形，特别地，每一个二维空间中的点 (x, y) 与三维空间中的直线 $\{(xw, yw, w) \mid w$ 不等于零$\}$都存在着一种明确的对应关系。这条直线与$Z = 1$的平面相交的点是 $(x, y, 1)$，这是在三维空间中的二维齐次平面中的一个元素，如图2-2所示。如果点 (x, y) 等价于二维空间中的齐次点 $(x, y, 1)$，可以获得所有二维空间内的3×3的变换矩阵。同样的对应关系也存在于点 (x, y, z) 和 $(x, y, z, 1)$ 中，其中把三维空间中的点嵌入到四维空间中，令其第四个坐标为1，这个对应关系就是三维图形中的4×4的变换矩阵。

如果认为四维空间拥有X、Y、Z和W四个分量，那么，三维空间等价于四维空间内W等于1的仿射子空间。因此，点（x，y，z）等价于四维空间中的点（x，y，z，1）。相反，四维空间中的点（x，y，z，w）等价于三维空间中的点（x/w，y/w，z/w），w不等于零。这种把顶点表示成四维空间中（w非零）的形式就是齐次坐标表达法。对于一个齐次表达式，通过对各个坐标除以w来计算它的三维等价坐标的过程，称为齐次化。齐次化在三维空间内变换的时候是非常有用的，在第4章中可以看到。把变换视做矩阵的时候，是对点的齐次坐标操作的，因此，变换矩阵通常是4×4的形式。

71

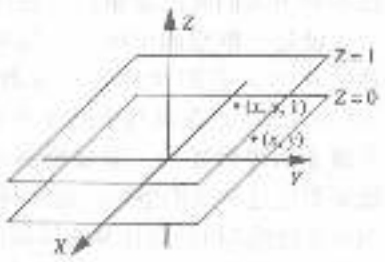


图2-2 在三维空间中的2D仿射平面

并非所有四维空间中的点都可以等价地表示成三维空间中的点，比如点（x，y，z，0）就没有三维空间中的对应点，因为它不能齐次化。但是它可以等价于由<x，y，z>定义的方向向量。可以认为是在特定方向上的“无穷远的点”，这在讨论方向光源代替位置光源的光照问题的时候是非常有用的，但在这里对图形学的讨论中不会经常使用齐次坐标。

2.3 例子

本节描述绝大多数图形API直接支持的简单对象。从非常简单的对象开始，逐步过渡到处理复杂的对象，用户会发现，在工作中既需要处理简单的，也需要处理复杂的对象。对于每一个简单对象，我们会描述如何定义对象，在后续的例子中，我们会生成一些点的集合，并且展示一些函数，这些函数可以绘制我们定义的物体对象。

2.3.1 单点和多点

要绘制一个点，首先定义点的坐标，然后把坐标传送到图形API函数中以绘制点。函数可以处理一个点或者几个点，因此，如果要处理一个点，只要提供点的坐标，如果我们要绘制许多点，要提供许多点的坐标。点的绘制非常快，因此，如果一个模型需要绘制数以万计的点是十分可行的。在一台没有什么图形硬件加速的普通机器上，有5万个点的模型可以在不到1秒内重新绘制。

2.3.2 线段

要绘制线段，将两端的顶点输入到图形API函数中。这个函数也让用户指定多条线段，然后绘制它们。对于每条线段，用户给出线段的端点，因此，需要指定两倍于所需线段数量的顶点数量。

然而，图形API处理线段的时候隐藏了一个重要的概念。直线是拥有实坐标的连续对象，但是在屏幕空间中显示的是整数屏幕坐标。这就是，一方面在模型空间和视点空间，另一方面在屏幕空间两者之间的区别。当关注3D空间内的几何体的时候，会忽略从视点空间到屏幕空间的转换细节，用户应该意识到这种转化的算法是以计算机图形学为基础的，而且用高层形式思考的能力也建立于这种基础之上。

2.3.3 线段序列

连接起来的线段——“头尾”相连的线段形成一个长的线段集合（如图2-3所示）。通常称为线段序列与封闭线段，用户使用的图形API应该会提供函数绘制它们。顶点列表定义了线

段，先使用两个顶点定义第一条线段，然后定义一个点产生一段新的线段，线段序列和封闭 72
线段的区别在于前者的最后一个顶点和第一个顶点是不连接的，呈开放的状态；后者包含一个额外的直线段从而形成一个闭合的环。线段序列绘制的线段数量比顶点列表中的顶点数目少1，而封闭线段绘制的直线段数目等于顶点列表中的顶点数目。这里可以使用几何压缩技术，用两个端点定义一条线段的方法被简化了，对于第一条线段之后的每一条线段，只有一个顶点需要额外定义。为了定义有N段的线段序列，用户只需要指定$N+1$个顶点就可以了，而不需要指定$2N$个。



图2-3 线段序列和封闭线段

2.3.4 三角形

为了绘制一个或者更多的不连续的三角形，图形API会有简单的三角形绘制函数提供。使用这个函数，每三个顶点可以定义一个独立三角形，因此，由顶点列表定义的三角形数目是顶点列表中顶点数目的三分之一。这些看似粗糙的三角形可能是多边形中最简单的形式，却是最重要的角色。无论用户如何使用三角形，无论什么样的点来组成顶点，它始终是凸的，并且位于一个平面内。另外，任何多边形，无论凸与否，都可以表示成三角形的集合。正因为此，大多数基于多边形的建模可以退化成三角形的建模，几乎所有的图形工具都知道如何管理三角形定义的物体对象。因此，处理好这些简单的多边形需要学会如何用三角形组织多边形和多面体。

2.3.5 三角形序列

三角形是绝大多数基于多边形图形的基础，用三角形定义大物体的边界或表面是非常普遍的。图形学API针对三角形序列提供两种不同的几何压缩技术：三角条带和三角扇形。这些技术在用三角形定义大图形对象时是非常有用的，如图2-4所示。图2-4中显示的几何基元，从外表上看好像是在2D空间内绘制的，实际上不是这样的，要让它们看起来是在3D空间内的，需要使用阴影技术，这个我们目前还没有涉及。因此，即使看起来是平的，我们也建议读者在三维空间内思考它们。



图2-4 三角条带和三角扇形

73

上述这两种图形API支持的技术用不同的方式解释顶点列表。为了生成三角条带，顶点列表中最开始的三个顶点形成第一个三角形，然后每加一个顶点形成一个新的三角形，三角形的另两个来源于紧靠着它的前两个顶点。如果要生成三角扇形的话，顶点列表中的前三个顶点形成第一个三角形，然后每加一个顶点都形成一个新的三角形，其中三角形的另外两个顶点来自于前面相邻的一个顶点和列表中的首顶点。因此，在每个例子中的三角形数目都比顶点列表中的顶点数目少2，所以，这是一种非常高效的定义三角形的方式。这两种方式对于多边形的顺序而言，表现相当不同，对于三角扇形，所有三角形的方向是相同的（顺时针或者逆时针），而对于三角条带，交替的三角形的方向是相反的。因此，当使用光照模型的时候，编程人员需要特别注意。

2.3.6 四边形

凸四边形经常称为四方形，是为了与一般的四边形相区别，因为一般的四边形不需要是凸的，凸的和非凸的四边形的例子如图2-5所示。图形API的函数绘制四方形可能会让用户绘制多个四方形，因为每个四方形需要顶点列表中的四个顶点，开始的四个顶点定义第一个四

方形。接下来的四个顶点定义第二个，依此下去，顶点列表中的顶点数目是四方形数目的4倍，顶点序列就是四方形中的顶点，顺序和用户绕着四方形的周长转一圈一样。在这一章后面的例子中，用6个四方形定义一个立方体。

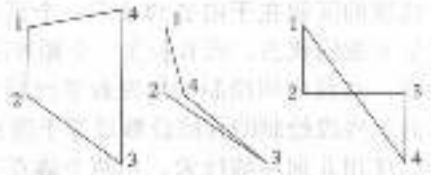


图2-5 凸的四方形（左边）和非凸的（中间的和右边的）四边形

在四边形中，指定顶点的序列是非常重要的。图2-5的凸四方形中，顶点的序列绕着四方形以逆时针顺序绕了一圈，这是标准的顺序，因为这样能支持正确的光照计算，用户会在第6章中看到这一点。更普遍的是，如果多边形是多面体的一个面，从多面体的外面看过去，顶点的顺序应该是逆时针的。

2.3.7 四边形序列

很多大的物体可以通过一些连接的四方形来定义，因此极大多数的图形API都有函数允许用户定义特殊的四方形序列，称为*四方形条带*。顶点列表中的顶点作为共边的四方形序列的顶点。最开始的四个顶点定义第一个四方形，然后这四个顶点中的后面2个顶点加上接下来的2个顶点组成下一个四方形，依此类推。顶点的顺序如图2-6所示。请读者仔细关注顶点的顺序，每一对点有相同的顺序，而不是期望的绕着四方形的序列，因此，序列3-4与期望早相反的方向，该相同序列在每一个额外顶点构成的另一对中也会继续下去。当用户实现四方形条带指令的时候，这个差异是非常重要的。当用三角形的形式来思考的时候这一点是很有用的，因为四方形条带处理顶点的时候就好像它确实是三角条带——顶点1/2/3的后面跟着2/3/4，再后面是3/4/5等。

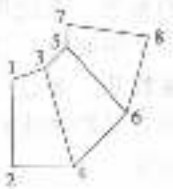


图2-6 在四方条带中的点序列

74

作为使用四方形条带和三角扇形的例子，我们来生成一个单位球体的模型。球体是我们熟悉的对象，一般有预先内置的球函数，然而看看如何使用基础工具建立熟悉的对象是很有帮助的。也有用户需要用到球体，而使用预先内置的球体函数又比较麻烦的时候。在“知识库”中有这样的一个例子是相当有用的。

在第4章中我们会描述在建模中球体坐标的使用方法，首先，用球体坐标对球体建模，然后转化到笛卡儿坐标中来实际绘制。建立球体模型首先沿着赤道分成N段，再沿着主子午线分成$N/2$段。在每一段中，角度的分割$theta = 360/N$度，假设球体的半径是1个单位，这样便于后面进行的变换操作。假定我们可以用球坐标的函数scvetex(...)来定义顶点，这样基本的代码结构为：

```
//  用三角扇形形成两个极的顶
doTriangleFan()   // 北极
    set scvertex at (1, 0, 90)
    for i = 0 to N
        set scvertex at (1, 360/i, 90-180/N)

endTriangleFan()
doTriangleFan()      // 南极
    set scvertex at (1, 0,  -90)
    for i = 0 to N
        set scvertex at (1, 360/i, -90+180/N)

endTriangleFan()
//  用四方条带形成球体
for j = -90+180/N to 90 - 180/2N
    // 绕着球体完整一个带子用一个四方条带
    // 在给定的纬度
```

```
doQuadStrip()
    for i = 0 to 360
        set scvertex at (1, i, j)
        set scvertex at (1, i, j+180/N)
    set scvertex at (1, i+360/N, j)
    set scvertex at (1, i+360/N, j+180/N)
endQuadStrip()
```

75

由于要创建模型的是球体，定义的四方条带是平面的，因此不需要把每个四方形分割成两个三角形来获得平面。注意此处设置三角扇形和四方条带时点的顺序。

一个分割较粗的线框球体的例子见图2-7所示。连同中间的三角扇形形成的极点顶以及右边展示的绕着球体的四方条带。请仔细观看四方条带环绕着球体的方式。



图2-7 一个线框球体（左图）、球体上的部分三角扇形（中图）以及球体中的四方条带（右图）

2.3.8 通用多边形

一些图像需要包含更多种类的多边形，虽然可以通过手工建立三角形和/或四边形来生成，然而把它们作为多边形来定义会简单许多。很多API只能处理凸多边形（凸多边形中任何两点连线上全部的点都位于其内部）。这是因为很多API中多边形是通过三角扇形的方式实现的，当多边形不是凸的时候，该方式无效。

图形API允许用户通过指定顶点的方式来定义一个多边形，顶点列表中的顶点以序列顺序作为多边形的顶点输入。凸多边形中的一个有趣的性质就是，如果取多边形中两个相邻的顶点，把其余的顶点以环绕多边形的顺序列出来的话，这样的顶点顺序就是定义三角扇形时的顶点序列。事实上，这可以作为凸多边形的定义，也给出了以三角扇形实现凸多边形的方法，就像对于四方条带和三角条带一样。（用户可以用这个方法处理非凸多边形的一些顶点，比如图2-5或图2-8中间的多边形，但是不能保证对于其中任意的两个邻接顶点该规律都成立。）我们来看图2-5中的凸与非凸的多面体，图2-8展示了多边形之间的不同之处。图2-8中间的物体是非凸的，因为顶点3和5之间的线段没有包含在图形内部，同样道理，最右边的物体也不是凸多边形。

76

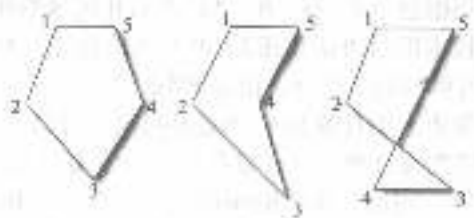


图2-8 凸的（左边）和非凸的（中间和右边）多边形

由于多边形是一个平面图形，因此它有两个面，一个是前向面，还有一个是后向面。这些通过多边形的平面法向量定义，我们很快会讨论法线的问题，我们希望法线一点指向多面体的外侧，或者一直指向一个由两个变量组成的函数所定义的表面上方。多边形在多面体外侧的面称为多边形的前向面，如果在没有多面体的情况下，（凸多边形）前向面的顶点是按逆时针顺序排列的，这个区别在图形学的计算中是很重要的。另外一种识别凸多边形前向面的方法是，取多边形内部的一个点，将它与多边形中的第一个顶点连起来，对于前向面而言，这条直线与该点连接其他顶点所得直线组成的线段之间的夹角随着用户取的顶点索引的增大而增大。

2.3.9 多面体

在图2-1中，我们注意到多面体是建模中的基本对象，特别是在计算机图形学中。通过指定所有组成外部边界的多边形来定义多面体。通常，绝大多数的图形API都把多边形的定义标准留给用户去做，当学习这个主题的时候，读者会发现多面体确实比较难定义。但是，凭借着用户的经验，他能够开发出一些多面体的集合，这些多面体是他所熟悉的，可以很轻松自如地使用。

虽然图形API一般不提供多面体集合供用户直接使用，然而大多数API有一些基本的多面体，它们是很有用的。这些取决于API本身，在第3章中就包含了用OpenGL和它的标准工具所描述的多面体。

2.3.10 走样和反走样

当用户生成一个点、一条线或者一个多边形的时候，系统在2D屏幕空间内定义像素点来表达这个几何体。该过程在第4章中有比较详细的描述，在2D屏幕空间内用整数定义像素，但是在模型空间内的几何体却是用实数定义的，在模型空间中的连续直线用屏幕空间中的离散像素集合来表达就是走样的一个例子。通常，通过采样获得数值时会产生走样，走样的概念在计算机图形学和其他学科中都有。

选择像素的方法是有或者无：通过计算出一个像素在几何体内部，这时用几何体定义的颜色表示，或者在几何体外部，这时保留它原来的颜色不变。由于屏幕空间相对是比较粗糙度的，这种有或者无的方法会在几何体和背景中间的空间留下锯齿状的边界。这种走样的表现见图2-9中左边的图像。

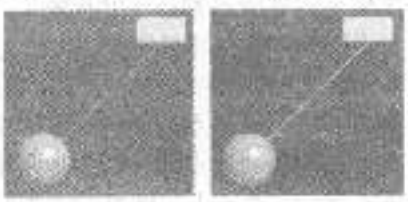


图2-9 走样的直线（左边）和反走样的直线（右边）

77

有很多技术可减少走样带来的负面效果，所有这样的技术称为反走样，它们的工作原理都是通过识别几何体的边界，对于独立的像素点采用一种合适的方法，仅部分覆盖一个像素。每一种反走样的技术都需要计算覆盖率，然后根据几何体中像素的覆盖率来点亮该像素。由于背景会发生改变，这种变化的亮度可以通过控制像素颜色的混合值来管理，使用颜色（R，G，B，A）表示几何体的颜色，A是几何对象中像素的覆盖比例。应用了反走样技术的结果请见图2-9中右边的图像。有些图形API中带有反走样技术，更多关于颜色和混合方面的细节信息，请见第5章。

在一幅非常高质量的图像中，要计算出一个像素的颜色可以对这个像素进行超采样。假定有比目前实际存在高很多的图像分辨率，计算在每一个像素中有多少"子像素点"，子像素点的比例作为新颜色的覆盖值，与原始像素的颜色进行混合。一种更为简单的技术利用基于多边形的线性几何建模的优势，精确计算2D视点空间的直线是如何与每一个像素相交的，获得像素的覆盖比例。这是相当常见的API计算方法，但是在不同的API之间以及在同一个API的不同实现中，具体方法有所差异，用户应该参考图形API手册获得更多的细节信息。

在图2-9中，反走样用在对直线段形状的平滑处理上，它也可以用于在平滑多边形边界上绘制点。该技术还可用于在已经绘制好的几何体上方绘制新的几何体。

2.3.11 法线

当定义一个对象的几何体时，用户既需要定义顶点的几何坐标，也需要定义该点垂直于该对象的方向，这对于生成具有阴影的图像是非常重要的，用户通过定义该物体的法线方向

来指定垂直方向，法线一般比较容易得到，用户可以通过对邻接的几条边做叉积运算获得平面的法线，这将在第4章中描述。对于很多图形API中可以得到的对象，法线放在对象的定义中。对于数学公式定义的对象，可以用微积分得到法线。 78

球体可以描述成四边形的序列，通过计算可获得法线。对于球体，其上面给定点的法线与在该点的半径向量是同向的。对于中心在原点的单位球，球面上一个点的半径向量与这个点的坐标分量相同。因此，如果用户知道了这个点的坐标，也就知道了在该点上的法线方向，如图2-10所示。



图2-10 在球体上一个四边平面中顶点的法线方向

为了在建模的定义中加入法线的信息，用户可以简单地使用图形API函数来设置几何基元的法线。如用户期待从图形学API中得到的那样，能够获得如下从球体的例子中引用的代码。在此采用球体表示法线和顶点，参见如下球体的代码段：

```
for (float j=-90.;j<=90.-180./N; j+=latstep)     // 纬度
    doQuadStrip()
    //  在任意纬度上，从经普球体每一条都有一个四方条带
        for (float i=0.; i<=360.; i+=longstep)     // 经度
            set scnormal to (i., j, z)
            set scvertex at (i., j, z)
            set scvertex at (i., j+180./M)
            set scvertex at (i., i+360/N, j)
            set scvertex at (i., i+360/N, j+180./M)
    endQuadStrip()
```

由于我们操作一个单位球体，该法线是单位长的。然而，用这种方法定义的法线可能不是单位长度的。因此，在使用法线之前，要对它进行归一化（使它的长度为1）。我们在这里不包含计算过程，因为图形API会自动对法线进行归一化，如果用每个四边形的中心代替顶点来求解法线，用户会得到更为精确的结果。其原理是一样的，但是用户需要通过对四边形的每个顶点求取平均值来找到它的中心点坐标。 79

2.3.12 裁剪

裁剪定义模型空间中的一个平面，并且绘制图形中位于该平面一侧或者在平面上的所有模型元素，但是模型中在平面另一侧的物体则不显示。裁剪定义了用户不想显示的部分场景。就像我们已经看到的，投影操作自动包含了裁剪，因为它们必须忽略在可视体的左边、右边、上边、下边、前面以及后面的物体。对于图像来说，包围视域体的每一个平面对于投影来说都是一个裁剪平面。平面通过线性方程来定义：

$$Ax + By + Cz + D = 0$$

因此，通过像第4章中四元实数组（A, B, C, D）就可以表达一个平面。图形API会让用户定义图像中其余裁剪平面，同时允许用户通过给出平面方程的四个系数来定义平面。使用裁剪的一个原因是用户想看到物体内部的情况，而不是看物体的表面，用户可以在透过该物体定义裁剪平面，达到只显示该平面一侧物体的目的。

裁剪是用图形API处理的，但用户需要知道它是如何做的。通常用多边形的方法来建立图形对象，因此通过判断顶点关于裁剪平面的位置来完成裁剪。有了上面的裁剪平面方程，用户可以通过判断表达式$Ax + By + Cz + D$的代数符号知道一个点（x, y, z）位于平面的那一侧。如果对于一个直线段的两个端点的判断都是负的，那么，整条线段都在裁剪平面"错误的"一面，因此需要丢弃。如果两个端点的表达式都是正的，说明整条线段都在"正确的"一面，

将会全部保留。如果一个端点的表达式是正的，而另一个端点的表达式是负的，那么，需要找到直线上满足$Ax + By + Cz + D = 0$的点，绘制的时候需要保留的是正号端点到该点之间的部分，而丢弃其余的那一段。如果直线段是通过线性参数方程定义的，那么该过程就是一个一元变量的线性方程，求解非常容易。对于直线段的裁剪可以扩展到多边形的范围，这在第1章中曾经提到过。

裁剪在模型中引入了不同种类的几何体——它们定义了可以看见的部分以及不能看见的部分。在裁剪之前，应用于模型的任何变换也应用于裁剪平面；任何在裁剪平面之后应用的变换则不被定义。

2.3.13 建模的数据结构

组成模型的对象可能有大量的顶点，因此，很自然就想问如何存储那些顶点，以及它们的属性。最自然的方法是通过表的结构，每个表通过顶点－顶点的方式来存储信息。用户可以使用任意的数据结构来实现这些表，只要满足该数据结构并允许简单的（通常是顺序的）方式对表中的元素进行存取。用户选择的表对于图形学来说是不重要的，因此，在这里将会限制只使用数组阵列。而用户在自己的工作中则可以自由选择使用。

作为例子，当用户定义多面体的时候，有很多方法来组织可以描述的信息。一种最简单的方式是三角形列表——这是一种三角形组成的数组阵列，每一个三角形都是三个顶点组成的数组。一个声明如下：

80
```
float triangles[N][3][3];
```

在这里每一项是三个二元组，即三个三角形。绘制一个对象就是简单地从列表中读入一个数组项，用三个顶点绘制一个三角形。在第15章将讨论用STL图形文件格式实现表的例子。

一种更为高效的，虽然稍复杂的方法就是生成三个列表。第一个列表是顶点列表，它仅包含对象中所有顶点的顶点数组阵列。如果该对象是一个多边形或者包含多边形。第二个列表是边的表，它包含多边形中每条边的一项。边的表中的每一项是有序的实数对，它们是顶点列表中点的索引，为了绘制从点i到点j之间的直线，用户用vertex[i]和vertex[j]来表示。如果对象是一个多面体，第三个列表是面的列表，包括多面体中每个面的信息。每一个面用组成面的边的索引列表来定义，其顺序则是面的方向。接下来，用户就可以通过边的非直接索引来绘制面，而边则可以通过顶点的非直接索引来绘制。要绘制对象，用户通过循环访问面表来绘制每一个面；对于每一个面，循环访问每一个边表来决定每一条边；对于每一条边得到两个顶点，这样就可以确定实际的几何体了。

让我们考虑经典的立方体例子，它的中心在原点，边长为2。我们定义顶点阵列，边阵列，还有立方体的面阵列，同时略述如何组织立方体来进行绘制。我们会在这一章的后面回到这个例子，同时在本书的其他部分也会讨论到这个例子。

我们从这个立方体的数据和数据类型开始，它的顶点是由三个点组成的阵列，边则是点列表中点的索引对，而面则是边表中的面索引的四元组。每一个面都有一个法向，但是这些也作为三个点的阵列给出，用C语言表示的代码如下：

```
typedef float point3[3];
typedef int edge[2];
typedef int face[4];      // 每一个立方体中的面有4条边

point3 vertices[8] = {{-1.0, -1.0, -1.0},
                      {-1.0, -1.0,  1.0},
                      {-1.0,  1.0, -1.0},
                      { 1.0,  1.0,  1.0},
                      { 1.0, -1.0, -1.0},
```

```
                        { 1.0, -1.0,  1.0},
                        { 1.0,  1.0, -1.0},
                        { 1.0,  1.0,  1.0} };
point3 normals[6] =  {{ 0.0, 0.0, 1.0},   // 每一个面一个法向量
                        { 1.0, 0.0, 0.0},
                        { 0.0, 0.0,-1.0},
                        {-1.0, 0.0, 0.0},
                        { 0.0,-1.0, 0.0},
                        { 0.0, 1.0, 0.0} };

edge edges[24]    =  {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                        { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                        { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                        { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                        { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                        { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};

     face cube[6] = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                     { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                     {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

81

在边列表中，每条边实际上出现了两次，每个方向出现一次。从立方体的外面看过去边序列呈逆时针方向。通过面的表确定组成立方体的点，然后将它们发送到通用的vertex(...)和normal（...）函数中，以绘制立方体。在以下伪代码中，假定多边形中的边没有自动闭合，因此，定义面必须在面的开始和结束都列出顶点。如果用户的API有自动闭合边，那么可以忽略第一个vertex（）的调用。

```
void cube(void) {
    for faces 1 to 6
        start face
            normal(normals[i]);
            vertex(vertices[edges[cube[face][0]][0]);
            for each edge in the face
                vertex(vertices[edges[cube[face][edge]][1]);
        end face
}
```

对于每一个面中的法线增加一个简单的法向列表结构，它支持平面阴影，或者由单一颜色组成的面阴影。在很多应用中，用户可能要使用平滑阴影，在平滑阴影中，颜色在多边形的每一个面之间平滑地混合起来，因此，每个顶点需要一个独立的法线。定义顶点就指定法线，在顶点列表后加入法向列表会让用户的工作非常轻松。例如，对于以上的代码，没有对于每一个面的法线，但是，可以通过以下的一对操作来取代对每个顶点的操作：

```
normal(normals[edges[cube[face][0]][0]);
vertex(vertices[edges[cube[face][0]][0]);
```

在第4章中读者会看见一些计算多边形顶点法向的方法和技巧。

2.3.14 曲面的建模

曲面建模是一种非常有用的建模。曲面是三维空间中点的集合，它是在二维空间内函数或者过程的图像。当使用一个函数的时候，我们考虑函数曲面。在自然科学中，也用它们来观察连续函数的行为。

可以通过在二维空间内生成点$P_{ij}=(x_i, y_j)$或者在三维空间内计算点$(u,v,w)_{ij}$来绘制曲面。对于四个点P_{ij}，$P_{(i+1)j}$，$P_{i(j+1)}$，$P_{(i+1)(j+1)}$，用户可以选择两个三角形来组成它们定义的多面体，确定三维空间中对应于这些网格的点，然后，就可以绘制三维空间中它们所定义的两个三角形。这个过程讨论起来比实现要困难很多。在第9章中我们展示了其中的一些细节，在那里我们会看到图形学在科学中应用的例子。

82

作为其中的一个特例，我们来看由两个变量函数定义的曲面。如果函数的表现相当好，

计算机图形学可以直接进行操作。两元变量函数的图像通常表达成一个曲面。对于每一个在定义域内的点(x, y)计算$f(x, y)$的值，而三元组$(x, y, f(x, y))$就是函数的图像。然后就像上面看到的，我们不需要绘制图上的每一个点，可以选择在定义域内有规律分布的点，计算点的函数值，连接这些定义点形成曲面中的一个部分。如果一次取出曲面中的四个顶点，就可以用两个三角形来组成这个连接着的曲面片，任何三角形都是共面的，用户可以选择固有的颜色或者通过光照和阴影所计算出的颜色来显示这个三角形，这取决于用户希望得到的结果，该过程的描述请见图2-11，它由比较粗的网格所组成。用户可以看见表面的基础是定义域内长方形的集合，在实际的曲面内，两个三角形定义了一个长方形。



图2-11 映射定义域长方形的集合到曲面三角形

2.3.15 其他的图形对象源

有趣的和复杂的图形对象是很难生成的，由于它们需要用户花费很多的工作量去测量或者计算每一个顶点的具体坐标。人们开发了一些更自动化的设备，包括3D扫描仪和激光的范围搜索设备，但是在大多数大学的课堂中，这些设备是很难获得的。那么，怎样才能得到令我们感兴趣的对象呢？这里介绍四种方法。

第一种得到模型的方法是购买。找3D模型的商业提供者，比如人体结构的模型可从医学和法医那里可以得到。这个方法虽然昂贵，但是确实省去了专家级的开发和建模过程。如果用户感兴趣，一个优秀的商业源（如本书的发行商）是http://www.digimation.com/ModelBankCollection/，里面有一些免费的例子可以观看。

第二种获得模型的方法是模型共享，如果用户有朋友在图形研究领域的话，可以问他们要些模型。比如，蛋白质数据银行（http://www.wwpdb.org）提供了很多结构模型，都是免费的。如果用户想要各种各样不同种类的模型，那么看看网站http://avalon.viewpoint.com，里面有很多公共领域的模型，是慷慨人士的捐赠。用户需要编写或者寻找到文件阅读器，把一种模型格式中的数据读入到用户程序的数据结构中。用这种方法得到的模型可有很多不同的数据格式，因此，用户可能需要编写或寻找将这些格式转换为可用格式的函数。

83

第三种得到模型的方法是用合适的数字化设备对物体进行数字化，但是它们的精确度和价格是成正比的。如果要数字化特定的物体，用户可以比较不同种类设备的性价比。通过软件工具来捕获点，并存储为标准格式的几何体，这样的方法可能与用户使用的图形API或者数据结构不兼容，像上面一样，需要做格式的转换。

第四种得到模型的方法是用户依靠自己的想象力来生成模型。许多免费或商业图形产品允许用户自己制作高质量的交互3D模型。同样会遇到文件格式的问题，但一个优秀的建模系统可以用几种不同的格式保存模型，用户可以任选其一应用图形API。用户也可以通过生成分析来生成有趣的模型，使用数学方法来生成顶点。用户对模型的质量和形式有最终的控制权，因此这个方法的效果很好。

如果用户从建模工具、数字化仪器或者其他地方获得模型，就会发现它们使用了不同的数据格式。有时候看起来每一个图形工具都有自己独特的数据格式。一些文件或者建模工具会用很多格式打开模型，允许用户用不同的格式保存，扮演一个格式转换器的角色。用户可能需要理解模型的文件格式，并且写出自己的函数来读入这些格式，然后生成内部的数据，应用到自己的模型中。我们需要花一些功夫来写滤波器用来转换格式；有时某种特殊格式转换比直接购买模型花费还要大。当然这取决于用户自己的选择。图形文件格式大全[MUR]是优秀的文件格式源，我们推荐用户参考那本书来了解特殊格式的细节信息。

2.3.16 建模行为

几何体是传统计算机图形学的基础，传统的图形学主要关注一次生成一个图像。我们需要介绍模型的动态行为，以方便与我们的观众做交流。模型的行为指的是一个模型如何随着时间变化而改变，或者如何对内部力作反应，或者对于外部输入的刺激作反应。如太阳系这个模型。一幅单一的图像不能告诉我们行星的行为，因此没有足够的信息来生成太阳、行星和月亮的几何模型，即使我们使用纹理映射让每一个行星和月亮看起来很真实。太阳系中更多的是运动而不是位置。

可以用两种方法来定义行星的运动。一种方法是用方程组表示每个行星和月亮随着时间变化的位置，该方程在开始有一个已知的初始位置。随着时间变化生成新的位置，然后绘制这一系列运动的图像从而显示该运动。这个方法是十分精确的，但取决于运动方程组本身的好坏。捕获行星运动的椭圆轨道的方程组以及包含行星和月亮之间相互作用的方程组是非常复杂的。

另一方面，可以从位置、速度的初始集合开始，应用物理学的定律知道，每个物体的加速度是由于每一个其他物体的引力效用引起的。这就是太阳系工作的基本方式，毕竟，如果根据它们的加速度得到的新速度来更新行星和月亮的位置，我们可以得到一个模型，这个模型不仅包括了以上方程组的行为，而且还有一些特殊效应，比如海王星的摇摆暗示了冥王星在20世纪30年代的发现。该方法以复杂差分方程来描述模型为代价，需要很多数值计算的知识进行编程，即使应用了最好的数值解法，还会有数值误差出现。该方法可能超过了绝大多数刚开始学习图形学的学生（还有相当数量的导师）的经验。

然而，无论建模过程如何，处理的问题都类似：随着时间的变化这些模型是如何变化的，或者是如何响应外界的不同输入，以及如何在程序中捕获模型的行为，以便图形系统可以精确地表现模型的行为？这些问题的答案可能已经超越了计算机图形学的范畴，因为它们来源于具体模型的研究领域，但是，计算机图形学需要提供工具对用户已经识别的模型行为进行建模。

2.3.17 建议

建模是图形中最耗时间的部分，但是除非用户非常仔细地完成了建模的过程，否则，不能生成有用并且很有趣的图像。练习建模编程的最好办法就是生成模型的一个简单版本。一旦当用户对编程结果满意的时候，他就可以替换掉简单的模型——这个简单模型只有一些多边形而已——用他表达的模型来替代它。

2.4 变换和建模

这一节需要一些数学背景知识。用户需要理解一般函数和复合函数的概念，以及理解3D几何及在三维空间内的物体运动，同时也要对堆栈数据结构有一般的了解。

在图形系统中，变换是生成图形的一个关键点。在世界坐标系内根据物体实际坐标进行建模是非常困难的事情，如果要在场景中用动画或者由用户移动物体，就更加困难。模型变换让用户能够使用任何空间定义每一个物体，并且可以按照自己的想法在世界坐标系内放置它，移动它。模型变换也允许用户在场景中放置光源和视点，并且按照需要移动它们。

在计算机图形中有多种变换形式：投影变换、视角变换，以及模型变换，用户图形API都支持它们。投影变换指三维空间中的场景映射到二维的屏幕空间内，在用户定义透视或者正交投影的时候定义。视角变换让用户能够从空间中的任何位置观察场景，当用户定义视区环境的时候定义（我们在前几章中已经讨论过）。模型变换用于在场景中放置物体对象，用户定义这

些物体的大小、方向以及位置。总之，这些变换组成图形流水线，这在第0章中已经讨论过。

有三个基本的模型变换：旋转、平移以及缩放，是用户建立更为复杂模型时候所应用的基础工具。在这一章的后面部分会介绍如何使用场景图来定义和维护复杂模型中各个部分之间的关系。

模型变换的好处是组合变换，以获得对模型完全的控制。简单的变换被结合到一个复合的模型变换中。这些复合变换可以储存起来，在以后恢复。图形API还有一个好处是它支持复合变换，而不需要程序员进行繁重的工作，这一点在第3章中的OpenGL建模操作中，用户能够看到。在这一节里，我们会看见用复合变换构建的建模实例。

最后，通过简单的建模和变换可以生成更为复杂的图形对象，但是这些对象的显示是需要耗费时间的。图形API可以保存预先编译好的对象，这样可以比简单定义的对象执行更快，这些已经编译好的对象通常易于生成和使用。

2.5 定义

本节描述几何变换的基础概念以及计算机图形学中使用的基础变换。接着，我们会描述基础变换如何独立针对用户场景的通用图形对象。

2.5.1 变换

变换是作用在n维空间内的函数。在计算机图形中使用的线性变换保留了很多几何上的关系，因此，我们把变换看作是输入几何体、输出新的几何体的函数。几何体是一切计算机图形系统能够处理的对象——投影、视角、光源、法向或者显示的对象。我们已经讨论过投影和视角了，因此，在这一节里，我们谈谈作为建模工具的变换。我们前面已经讨论过三种变换：旋转、平移和缩放。下面，我们将分别看每一种变换，然后整体来看如何用这些变换生成想要显示的场景。

第一个变换的例子就是生成和移动一个椭球，因为椭球体在一个轴上看要长一些，因此，观察它绕着短轴旋转是比较容易的，当然，观察平移也很容易。对球体进行缩放就可以得到椭球。首先来讨论缩放，展示如何用它来生成椭球，然后旋转，展示它如何绕着短轴旋转，接着是平移，以显示它可以移动到我们所希望的地方。最后，看变换是如何一起工作以生成旋转并移动球，把球像我们看见的那样被踢出去的。这个球用非常简单的光照和阴影作处理，就像我们将在第6章中描述的那样。

86

对每一个顶点的每个坐标乘以一个固定的数值完成对物体的缩放。缩放变换一次需要分别作用于物体的各个维度。图形API中的缩放函数把三个实数作为参数，对应于三个坐标。如果有一个点坐标是（x，y，z），三个方向上的缩放因子为Sx、Sy、Sz，应用缩放函数就把这个点变换到了（x*Sx，y*Sy，z*Sz）的位置。如果在原点放置一个简单的球体，并且在一个方向上缩放2.0倍（在我们的例子中是z坐标或者垂直向上），再向上平移2.0，这样它的底部就与地面平齐了，使用这样的函数：

```
translate(0.0, 0.0, 2.0);
scale(1.0, 2.0, 1.0);
sphere(1.0);
```

得到的椭球如图2-12中原始球体的右边所示。注意，这个缩放的操作对空间中的所有物体都起作用，如果恰巧在离原点外还有一个单位球体，缩放操作就会将球体移动到远离原点的地方，同时将它的所有坐标都乘以这个缩放因子。这个例子表明，对一个定义在原点的物体应用缩放操作只会改变它的尺寸大小。标准的建模方法将物体定义在原点，再应用缩放操作

到最接近于实际的几何体，因为那样是生成已知尺寸物体的最佳方式。

旋转操作将物体中的每一个顶点绕着一条直线旋转。定义旋转操作需要指定旋转量（用角度或者弧度）以及所需要围绕的轴。图形API函数会把角度和围绕的轴作为参数输入，原点和一个实数（即轴方向向量的坐标）确定的另一点构成一条旋转轴。橄榄球旋转函数的例子如下：

```
rotate(angle,0.0,1.0,0.0);
```

因为向量（0.0,1.0,0.0）指定了y轴，如果只改变中心位于坐标原点的物体的方向，旋转变换是非常有用的。在缩放之后，立即应用旋转是一种标准的建模方法。这样可以给出物体正确的大小以及在场景中正确的方向。



图2-12 一个没有缩放的球体（左边）和一个在y方向上缩放2.0倍，并且向上平移1.0个单位形成橄榄球（右边）

87

平移通过对每个物体的各个坐标增加一个固定的数值来改变物体顶点的坐标，它把物体中所有部分都移动一个相同的距离。平移操作需要三个参数，分别表示三个坐标的增加值。图形API平移函数如下所示：

```
translate(tx, ty, tz);
```

平移操作统一地对待物体中的各个部分，一般在缩放或旋转变换的后面使用，是建模的一种标准方法。平移可以将物体按正确的大小、方向放置在正确的位置。

最后，将所有变换放在一起，生成橄榄球在空间内移动的一系列图像，平移和旋转同时进行，如图2-13所示。首先用缩放来定义橄榄球，然后通过平移放置到地面上。接着通过缓慢的增加角度来旋转球体，在球体飞行的过程中分几次计算旋转和平移变换，用标准重力计算Tx，Ty，Tz对球体作平移变换，代码如下所示：

```
translate(tx, ty, tz)
rotate(angle, x-axis)
drawBall()
```



图2-13 通过变换来实现橄榄球在空间移动的一系列图像

88

上文中，drawBall()函数是这样定义的。它基于通用的drawSphere()函数，该函数绘制一个圆心在原点半径是1的球体：

```
scale(1., 2., 1.)
drawSphere()
```

球体从左向右运动，同时沿逆时针的方向缓慢地旋转，球体运动的位置可以用一条抛物线来描述。这样可以讨论球飞行时候的重力作用效果。下一节描述复合变换，其中的各个变换操作的顺序是非常关键的。

计算机图形学中的变换都有其简单的逆变换，或者是对原始变换的结果做撤销操作的变换。旋转Θ变换的逆变换就是绕着同一条轴线旋转−Θ角度，缩放（Sx，Sy，Sz）变换的逆变换就是缩放（$1/Sx$，$1/Sy$，$1/Sz$）；平移（Tx，Ty，Tz）的逆变换就是平移（$-Tx$，$-Ty$，$-Tz$），复合起来就是，如果S和T是两个变换，ST变换的逆变换就是S和T的逆变换，再用反向顺序结合起来，即：$(ST)^{-1}=T^{-1}S^{-1}$。

变换是从3D空间映射到3D空间的数学操作，有标准的数学表达方式，可以像处理实数阵列一样。用户不需了解详细细节，但从事高级图形学操作的人却必须了解。在第4章中讨论复合变换的标准矩阵操作方式。

2.5.2 复合变换

场景建模会用到超过1个的简单变换。这就是所谓的复合变换方法。举例来说，如果想用高度A，宽度B，深度C生成一个长方体的盒子，它的中心在(C_1, C_2, C_3)，并且相对于Z轴有一个角度，首先生成一个单位立方体，中心在原点。然后，应用如下的操作序列：

- 首先，对立方体用正确的尺寸进行缩放，使它具有长方体的边长A、B和C。
- 第二步，绕着$(0, 0, 1)$方向的直线旋转α角度。
- 第三步，将立方体平移到位置C_1、C_2和C_3。

这就是标准的建模方法，盒子的代码形式表达如下：

```
translate(...);
rotate(...);
scale(...);
cube();
```

对于变换，标准序列的操作顺序是很关键的。举例来说，如果先旋转，再用不同的缩放因子对每一个维度作缩放的话，会得到变形的盒子，因为盒子的边缘没有和坐标轴一致，因
89 此不同的缩放因子会改变x-y的比例关系。如果首先平移然后才旋转，那么旋转会把盒子移动到完全不同的位置。正因为如此，合理的操作顺序可以提供一个可预测的结果，即首先应用缩放变换，然后是旋转变换，最后是平移变换。

变换的顺序是非常重要的，已经超过了上面介绍的平移和旋转的例子的范围。一般来说，变换是不可交换顺序的，不可交换是指$f * g \neq g * f$（也就是说$f(g(x)) \neq g(f(x))$）。除非用户在其他课程中已经有不可交换操作的经验，比如线性代数，否则这对用户来说是一个新概念。我们来看上面所描述的操作：如果取点$(1, 1, 0)$，然后绕着Z轴应用90度的旋转操作，可以得到点$(-1, 1, 0)$，然后，应用平移$(2, 0, 0)$，又得到了$(1, 1, 0)$。但是，如果，先应用平移$(2, 0, 0)$，则先得到$(3, 1, 0)$，接着应用旋转，得到点$(-1, 3, 0)$，这样就不是原始点$(1, 1, 0)$了。以下是两种情况的伪代码：

```
(1)  rotate(90, 0, 0, 1)
     translate(2, 0, 0)
     setVertex(1, 1, 0)

(2)  translate(2, 0, 0)
     rotate(90, 0, 0, 1)
     setVertex(1, 1, 0)
```

产生不同的结果，因此，旋转和平移操作是不能交换的。在这一章后面有一个练习，读者可以尝试着用不同的顶点和不同的变换来看结果如何。

不可交换的性质并不局限于不同种类的变换。绕着不同轴的旋转操作的不同顺序也能产生不同的图象。下面用不同的顺序进行旋转，一个绕着Y轴，另一个绕着Z轴

```
(1)  rotate(60, 0, 0, 1)
     rotate(90, 0, 1, 0)
     scale(3, 1, .5)
     cube()

(2)  rotate(90, 0, 1, 0)
     rotate(60, 0, 0, 1)
     scale(3, 1, .5)
     cube()
```

90 两者的结果也是不同的，如图2-14所示。



图2-14 相同旋转操作的不同顺序的结果

为了计算方便用矩阵实现变换。我们用4元实数的齐次坐标来表示点，变换表达成一个4×4矩阵，将4元组的空间映射到同一个4元组的空间（我们在第4章中数学建模中会讨论原因，使用3×3的矩阵来表示建模转换的过程是不够的）。虽然本书不会经常用这种表达形式，图形API却一直是这样使用的，这样可以帮助我们解释变换的操作过程。举例来说，变换是不可交换的，因为矩阵乘法是不可交换的。虽然对于大多数API而言，用户不需要精通变换矩阵操作，但是到API之外去操纵变换时需要注意这一点。

为了保存当前的模型变换，或者恢复到已经保存的变换状态，用户可以使用变换栈。栈是一个数据结构，有如下性质：当我们从结构中取出数据的时候，取出的都是最近存放的。将东西放入栈称为压入栈，从栈内取出东西称为弹出栈。最后压入栈的东西保存在栈的顶部。变换栈里保存了变换的系列，在栈顶部的变换是当前的活跃变换。使用旋转、平移或者缩放时，将当前的活跃变换（在栈的顶部）和新的变换相乘。通过复制一个栈顶元素并且把它放入栈来保存当前的变换，保存的变换矩阵就放在当前栈顶的下面。通过弹出栈，将当前变换移出栈顶的方法来恢复前面的变换。本章的后面将详细讨论栈的使用方法。在后面（第4章），我们会看到变换表达成一个4×4的实数组成的矩阵，这也等同于16个实数组成的数组阵列，因此，我们可以把变换栈看成是数组的栈。

模型变换需要考虑如何方便计算，数学符号的使用可以有很多方法。有三种方法可以用于模型变换。第一种方法是简单地用“后指定—先应用”来定义变换的序列。第二种思考的方法是应用变换，让距离几何体最近的变换最先得到应用。第三种方法是通过建立复合函数。复合函数是由几个单独的函数相乘得到的，我们在此通过原先的函数右乘新变换来组成新函数。标准的操作序列如下：

```
translate(...);
rotate(...);
scale(...);
geometry();
```

可以通过代数的操作序列得到

```
translate * rotate * scale * geometry
```

或者作为复合函数的乘积，如下：

```
translate(rotate(scale(geometry())))
```

91

仔一看，这个序列似乎与上面标准操作序列的顺序相反。但缩放函数在代码中是最接近几何体的（函数geometry()）。这样，对于缩放函数正是一个正确的位置，因为变换有“后定义—先应用”的性质，图2-15生成了一个细长的、具有长方体形状的条，它向上45度角放置在定义的平面之上。我们从（左边的）简单立方体开始处理，将立方体做缩放变换，接着做旋转变换，最后用平移变换得到（右边的）图形。

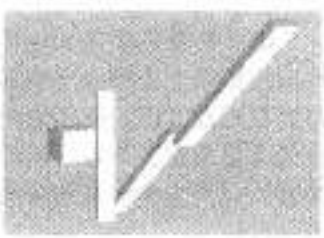


图2-15 立方体变换的序列图

如果P是一个投影变换，V是一个视口变换，T0，T1，…，Tlast是对场景进行建模的变换，它们的先后顺序就是出现在代码中的次序（也就是T1最先，Tlast是最后的），操作序列就是：

```
P→V→T0→T1→T2→……→Tn→Tn+1→……→Tlast→……→geometry
```

在前一章的例子代码中，我们看到投影变换定义在reshape()函数中，视口变换定义在init()函数的开始或者display()函数的开始。这两个变换在其他模型变换的前面。Tlast实际上是最先应用的，V和P是最后应用的。代码首先定义P，然后定义V，接着依次定义T0，T1，…，Tlast，最后定义几何体。现在请回到第1章中的实际代码例子，在那里追踪一下它的变换顺序，正是由于它对建立复杂的、具有层次关系的模型而言非常重要，因此用户需要较好地理解这个序列的顺序。

2.5.3 使用变换栈

在场景的定义中，可能需要定义一些标准的片段，然后把它们用特定的方式组装起来，最后使用合成的片段来生成用户场景中的部分对象。该过程可以通过不断重复来生成越来越

92

复杂的对象，整个过程通常在表达建模的函数中被编放。为了生成复杂的模型，首先通过函数来生成其中独立的部分，接着把它们都组装成一个整体，最后，就可以看到由各个不同的部分所组成的一幅完整的图像。

如上所述，在场景中定义几何对象一般都有某种变换存在，即使在只有投影和视口变换的情况下也同样如此。当用户开始把复合对象的各个简单部分放置起来的时候，他们会用一些变换来放置各个部分，但是当用户定义下一个部分的时候，可能需要撤销一些变换，开始一个新部分时，保存变换状态，然后，返回该变换状态，丢弃在标记以后增加的任何变换，再开始下一个部分。请注意，用户总是在我们上面描述的列表的末尾增加或者丢弃变换，因此这个操作很像堆栈。我们可以定义变换的堆栈，然后用下面的方式来管理：

- 定义变换，以右乘方式加入变换。
- 保存变换的状态，首先复制当前变换，将副本压入堆栈，然后对堆栈顶部的元素应用所有接下来的变换操作。为了要回到开始的变换，弹出堆栈的顶部，这样就得到了初始的变换。于是，我们就可以重新开始工作了。

正因为所有应用的变换都在堆栈的顶部，因此，当弹出堆栈的时候，我们就回到了初始的上下文环境中。

做撞球的例子定义了游戏场地以及目标的位置。模型只放置一次，那么，如何放置多次呢？对于场景中每一个球，向变换堆栈里压入一个变换的副本来保存当前的模型变换，接着对模型应用平移，旋转以及缩放操作，然后绘制球体，最后弹出变换的堆栈。因此，每一个球体的实例都是用它自己单独的变换放置到场景中去的，这些变换对于绘制球体以外的工作是没有效果的。

设计一个有很多几何体的场景，定义那些变换需要花费很多时间。在下一节中我们会介绍一些场景图的概念，这种设计工具可以帮助用户高效地生成复杂和动态的模型。

2.5.4 编译几何体

独立模型并将它变换到世界空间内有很大的工作量。首先在模型空间内计算顶点的坐标，然后应用模型变换以得到最终顶点的坐标。这些操作都将送入到视图和投影处理过程中，以得到在视点空间内的顶点坐标，最后还需要变换到屏幕空间内得到显示。如果模型非常复杂且经常要用到，而每一次绘制的时候它都需要重新计算的话，这样场景的显示速度就变得非常慢。应用变换涉及矩阵的乘法，因此，对于每一个变换和每一个顶点最多可以用16位浮点操作数。

93 为了在显示图像的时候能节约时间，很多图形API允许用户用某种方式在模型中“编译”几何体，使它们能够很快显示。这些编译了的几何体就是显示列表，它被发送到绘制流水线中。这将在第10章中详细介绍。当编译了的模型显示的时候，顶点和变换都不需要重新计算，只要把保存的计算结果发送到图形系统中就可以了。被编译的几何体要仔细选择，因为它们在显示之间是不能变化的。如果需要改变，就要重新编译。一旦用户找到了能编译的部分，就可以编译，并用编译好的版本使显示速度更快。我们在第3章中会讨论OpenGL是如何在显示列表中编译几何体的。如果用户使用其他的图形API，请仔细参考它的文档手册。

2.6 一个例子

让我们看看，如何从简单建模里生成有用的图形对象。首先设想一个3D的箭头，用于指出3D场景中的事物。目的是要得到如图2-16这样的箭头，它指向下方，与Y轴一致。当需要箭

头的时候就可以非常容易地重用，只要根据想要的大小和方向进行缩放和旋转，最后再平移到需要的地方即可。

制作这个箭头从两个简单且有用的简单形体开始（这些是GLU和GLUT中内建的函数，很有用，将在第3章中详细描述）。这些简单形体在设计时都在标准的位置，并且拥有标准的大小。这些模板在设计上并不需要方便，但需要看上去易于理解与使用。

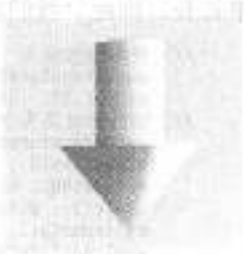


图2-16 在标准位置上的3D箭头

第一个简单形体是一个圆柱体，这是个标准的模板，可以通过变换拥有任意的大小和任意的方向。该圆柱体模板的中心线平行于*X*轴，一端的圆心在坐标原点，还有一端在*X*轴的正方向，半径是1，且长度也是1。圆柱体的横截面是由*NSIDES*条边的规则多边形所组成的。这是对一个实际圆柱体的合理近似，易于缩放。该模板请见图2-17中右侧的形体。第二个简单形体是一个圆锥体，其中心线在*Y*轴，其顶点在原点，基座上的半径是1，高度也是1，基座已经填充，如同圆柱体一样，该模板也易于根据要求缩放和改变方向。对圆锥体的基座也用*NSIDES*条边的多边形来构造。该圆锥体模板请见图2-17中的左图。



图2-17 组成箭头的模板：圆锥体（左侧）和圆柱体（右侧）

94

圆柱体的模板函数cylinder()的概略代码如下：

```
angle = 0.;
anglestep = TwoPI/(float)NSIDES;
for (i = 0; i < NSIDES; i++) {
    nextangle = angle - anglestep;
    beginQuad();
        vertex(0., cos(angle), sin(angle));
        vertex(1., cos(angle), sin(angle));
        vertex(1., cos(nextangle), sin(nextangle));
        vertex(0., cos(nextangle), sin(nextangle));
    endQuad();
    angle = nextangle;
}
```

圆锥体模板函数cone()的概略代码如下：

```
angle = 0.;
anglestep = TwoPI/(float)NSIDES;
beginTriangleFan();
    vertex(0., 0., 0.);
    for (i = 0; i < NSIDES; i++) {
        angle -= anglestep;
        vertex(cos(angle), 1., sin(angle));
}
endTriangleFan();
angle = 0.;
beginPolygon();
  for (i = 0; i < NSIDES; i++) {
    angle += anglestep;
    vertex(cos(angle), 1., sin(angle));
}
endPolygon();
```

95

请注意，即使图上已经用白色和红色分别表示，我们至今还没有设置圆柱体或者圆锥体的颜色，也没有定义像用户看到的图中的任何阴影处理。这些都是外观上的问题，我们将在第6章中讨论，这些图的外观就是为了让读者能更好地看清楚它们的形状和它们之间的关系。

现在已经建立了两个形体的模板，下面可以生成模板3D箭头函数arrow3D()了。将圆柱体的长度变成原来的2倍，其半径变成原来的一半，方向沿着*Y*轴，从原点向上平移一个单位。

将变换以后的圆柱体在原点和圆锥体相结合，从而形成箭头的形状。其结果就是该箭头有3个单位长。因此，我们需要均匀地把它以0.33的比例缩放成长度为1。

```
// 整体缩放箭头
pushTransformStack();
scale(.33, .33, .33);     // 1/3原始大小
// 缩放箭头中的圆柱体部分改变方向
pushTransformStack();
translate(0., 1., 0.);        // 放置在需要的地方
rotate(90., 0., 0., 1.);      // 绕着z轴转90度
scale(2., 0.5, 0.5);          // 正确大小的圆柱体
cylinder();
popTransformStack();
// 使用定义的箭体而不做改变
cone();
popTransformStack();
```

2.7 建议

变换的顺序很重要。当观察一幅明显有错误的图像时，要说出引起它的错误到底是什么，比如变换的顺序不对，是比较困难的。用户需要提高“视觉调试”的能力——即观察一幅图像，发现它是不正确的，然后指出是什么问题造成这样的情况。然而，一般情况下用户不能说出一幅图像是错误的，除非他知道正确的图像是什么样的。因此，用户必须首先知道他看到的应该是什么。比如一个简单的例子，如果用户正在制作一张科学图像，他必须足够了解该门科学，这样，他做出的图才有意义。

2.8 建模视觉交流

形体是任何图像中的基础部分。所有的图像都是从建模中得到的，而建模是基于生成有形体的几何对象的。动态的行为也是非常重要的，因为静态的图像不能给予我们关于动态和交互性质的良好感觉，视觉交流从图像中的形体开始，并且具有一定的行为特征。这里主要关注形体，在第7章中讨论事件以及在第11章中讨论动画的时候，会更深入地介绍动态的行为。 96

生成图像时，可以得到形体的类别和基本形体的排列种类。用户可以使用简单的形体，在交流中强调基本的简洁。由简单形体生成一幅简洁的图像，并且很容易表达用户想要表达的东西。如果要用更为复杂的形体进行思想的沟通，那么就要找到预先建立好的形体，或者通过基本的变换和API支持的几何体来设计并创建形体。

当然，不能任意使用形体。有时，用户的图像会描述物理的对象，要生成形体来表达对象，这样做或者通过对象的精确信息，或者通过可认知但很简单的方式来表达对象实现。当形体在理论上或者数值上是显示平滑改变的时候，这些形体是平滑的；当只有离散的数据并且不知道如何平滑地显示它们时，形体是比较粗糙的。有时候，用户的图像会处理非物理对象并且要生成抽象的形体，来给予观察者一些概念上的认识。这些形体通过它们的位置或者大小、形状、颜色以及运动来表达其中的意义。

请对可能用作符号的形体的文化内涵多加小心，如果用一个简单的形体来显示一个数据点的位置，可以使用一个圆、一个方形、一个十字叉、一个五角星或者一个六角星。而后面三个形体都有着与文化的关联，可能是合适的也可能是不合适的。因此，在使用之前请仔细考虑。总的来说，请对形体中蕴含的文化内涵多加在意，一个选择对于他本人而言可能是没有关系的，但是对于其他人而言可能有很大的影响。

2.9 认识形体的含义

一幅图像只有以观察者理解的方式来表现才能传达正确的信息。这是非常明显的，但是

制作这个图像的人往往容易假定观察者拥有与制作者同样的视觉词汇，从而能够理解制作者想要表达的含义。事实上，我们每个人都有着丰富的视觉词汇。举例来说，我们知道很多符号的形状，并且已经适应了阅读简单的直线图。数学和自然科学课程强调了能够阅读以及生成不同的图和图表的能力。事实上，这些形体的意义就是一幅图像中文化内容的一部分。

下面讨论形体表达信息的方式，静电学中的库仑定律认为，平面中任何点的电势是该点从每一个点电荷中获得的电势差的总和，而从每一个点电荷中获得的电势差则是该点电荷的电量除以该点和点电荷之间的距离。用公式来表达点 (x, y) 处的电势有：

$$P(x, y) = \Sigma \frac{Q_i}{\sqrt{(x-x_i)^2 + (y-y_i)^2}}$$

上式中，每一个Q_i放置在点 (x_i, y_i)，该公式在第9章中还要详细地讨论。在这一章中讨论这个问题的一些表达方法，以此作为视觉表达的例子来说明。 97

现在我们来看库仑定律用在有三个固定点电荷的长方形曲面上的情况，三个点电荷中一个是正电荷，两个是负电荷，我们可以计算出在曲面上各个点的电势。这是一个在实数2D平面内的二变量函数，可以将它的函数图像表示成在三维空间内的曲面。图2-18用非常传统的方式来表达它的3D曲面，请注意此处特别强调了曲面的形状。如果强调的是曲面本身，这样可能是表达图的一种很好的方式，因为光照可以很好地显示这个形体。这可以表达成一个平滑的曲面，因为理论告诉我们，电势在整个区域内除了在电荷点本身（其分母为0）的位置以外都是连续的。对于这个曲面，如果我们认为它是有界的，那么唯一不正确的地方就在于那几个点的位置上，因为静电荷所在的位置曲面是不连续的。事实上，图像在这些点上趋向于无穷数值，因为在那几个点电荷所在的位置上的除数是0。该图显示了曲面中的两个极小值以及一个极大值，分别对应两个负电荷和一个正电荷的位置。在这个图中使用的视点让我们看到了三个点，并且给予我们对形体的良好表达，如果要得到对曲面最好的理解，视点的使用是非常关键的，因为另一个视点可能会隐藏极小值或者两个极小值之间的区域细节信息。

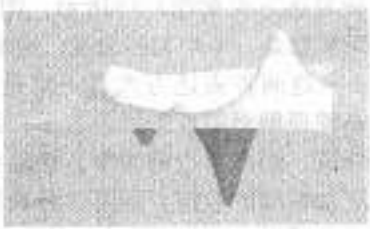


图2-18 用三个光源来显示传统形体曲面的图像。参见彩图

这种方法有一个潜在的问题，这取决于观察者的经验。观察者必须熟悉函数曲面的表达方式，因为在这个问题中没有实际的曲面；对于每一个2D空间中的点 (x, y)，点上的静电力是通过图像中的高度来表达的。图像中的山峰代表的静电力是正的，并且很大，低谷处则表示了静电力也很大，但是方向是负的，然而山峰和低谷正如我们看见的，都是不准确的，因为包含了不连续的区域，这个情况观察者需要理解。最后，由于观察者必须从曲面中看抽象的数值，因此，对于新手或者在没有其他相关讨论的情形下，这不是一个很好的表达方式。我们将在第9章中深入讨论这个问题。

如何用不同的方法表达这个概念呢？由于我们讨论的是在一个平面上的东西，因此可能会通过颜色代表电势的方式，用二维图像显示它的数值大小，也可以使用三维视角，但是可以用颜色斜率的方法替代自然颜色来实现，就像我们在温度渐变例子中做的那样。如何用图形的方式来表达一个概念，这个问题涉及图像中最为重要的部分，我们在本书中也会反复考虑。 98

2.10 维度

计算机图形学分为二维和三维，因为要在二维或者三维空间内生成图像。实际上，可以

使用超过3个维度，理解这一点能帮助我们挑选一些选择项，让图像给人以更加深刻的印象。

使用三个维度是显然的：因为普通空间就有三个坐标方向。随着时间变化生成动画，就有了对第4个维度的控制：时间。使用颜色来表达一个数值，就将颜色视为第5个维度。在这种方法中用到的颜色经常称作伪颜色。下面看一个特殊例子：表达一个物体对象的温度。对颜色表达的概念更加完整的讨论将会在第5章中进行。

我们从几个方面来看这个问题。首先从简单的一维概念开始，考虑沿着金属线的温度。在这个例子中，物理对象只有一个维度（长度），另外一个维度则是各个点上的温度。虽然不存在“温度”这个维度，但可以用数字或者颜色来代表温度。如果使用数字，就可以制作一张图，图的水平轴表示长度，竖直轴表示温度。这在数学和自然科学的教科书中是很熟悉的图。如果使用颜色，可以用一条直线来表达在各个点上的不同颜色，虽然这是一种很不同寻常的表达方式，但看起来却像实际的热力线。这两种表达方式如图2-19所示。图中的金属线初始时候都是一个均匀的温度，后来在两端变冷了。

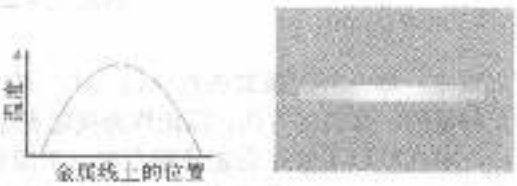


图2-19 在一条金属线上温度的两种表达方式：曲线（左图）和颜色（右图）

这两种表达方式有一点是相同的：观察者必须理解对温度的抽象表示，虽然抽象的方式是不一样的。在左图中，把温度作为第二个坐标值。而在右边的图中，把温度看作颜色。第一种方式更加量化，从图中，观察者可以得到更加精确的数值，而第二种表达方式能给人更加深刻的印象，同时让观察者对热和冷有更直
99 接的感受。用户必须自己决定使用哪一种表达方式。

如果要考虑在金属线上的温度是如何随着时间变换的，那就要加入第三个维度——时间了。下面有两种不同的方式：一种是作为第二个数值维度。第二种方法则是随着时间形成动画。因为原始的图像有两种选择，因此实际上，我们就有了4种可能的答案：

1. 在时间的任何点，温度用数值曲线，用数值方法扩展作为第三个维度以形成三维曲面；在这个曲面上，平行于初始曲线的切片都是在任何时间点上的温度，而垂直于初始曲线的切片则是一个固定点的温度随着时间的变化关系。

2. 对于任何点的温度用数值曲线，通过动画方式扩展使得曲线随着时间变化。

3. 对于任何点的温度用彩色直线，用数值扩展成第三个维度以产生一个有彩色的平面。在这个平面内，平行于初始曲线的切片是任何时间的温度，而垂直于初始曲线的是随着时间变化的某一点的温度。

4. 对于任何点的温度用彩色的直线，通过动画的方式使颜色随着时间改变。

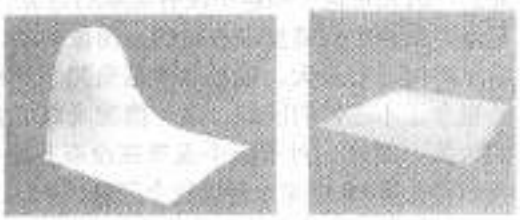


图2-20 1和3号解决方案中温度随着时间改变，请参见彩图

图2-20显示了两种表达方式增加时间作为第3维度。通过在侧着场景的右方向上增加一个水平轴来显示第3维。遗憾的是印刷的书本是不能显示动画的，因此通过动画来包含时间的方法在这里没有展现出来。

下面考虑3D空间中的温度，比如一间屋子中的温度。第4维度的选择是很有限的，因为时间不能在空间不同的点以不同的工作方式进行。
100 因此，用颜色作为温度的表达，这就是第4个维度了。同时，我们不得不用一些更高维的方法来观察这个场景（比如，2D的切片来显示颜色，或者等温面等）。为了看温度是如何随着时间改变的，我们一定要将显示动起来。

2.11 更高维度

这种建模（包括颜色的使用）当用户和2D空间内的函数一起工作时是很不错的。这里2D的点可以作为定义域，同时函数值可以提供第三个维度，由点的高度或者颜色值表示。然而，在3D空间中有可能函数不是单值的，在2D空间内用函数产生2D信息，在3D空间内随时间变换的过程，超过了用户在3D空间内说明信息的能力。在那些情况下，用户就必须自己找到其他的方法来描述自己的信息。这里我们只是提供一些包含颜色的例子，用户可以参考第5章来获得颜色的细节。

最简单的高维问题就是考虑一个过程或者函数，它能够在3D空间内操作，并且有一个简单的实数值。这是一个在空间点的位置上生成一个值的过程，比如温度或者压力。这里有两个问题。第一个问题是，“这个函数在空间中有哪些点有给定的数值？”这引出了空间中所谓的等值面的概念。找到体数据或者在有3个变量的函数中寻找等值面有复杂的算法。图2-21中左图显示一个针对等值面问题的简单方法，该方法将整个空间分割为很多小的立方体单元，函数在每个单元的每一个顶点上计算数值，如果一个单元中有一些顶点的函数值大于目标的数值，还有一些顶点的函数值小于目标数值，那么一个连续函数一定在这个单元内有某个点的函数值与目标数值相等，在该单元中绘制一个球体，第二种方法需要得到在3D空间内的一个2D子集中函数的数值，典型的就是一个平面。可以将一个剖平面放入3D空间中，检测在那个平面中的函数数值，然后用颜色在空间显示的平面中绘制出那些数值。图2-21右图显示了一个剖平面，该平面针对函数$f(x,y,z) = x+y+z$，在所有三个x，y，z空间分量上呈现双曲线形状，伪色彩就是上面所描述的统一光照亮度梯度。

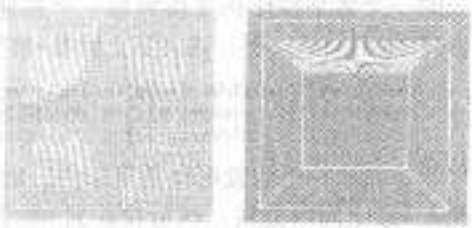


图2-21 一个相当简单的具有三个变量的函数中的等值面（左），沿着空间中的一个2D平面观察的3D空间中的函数值（右）

101

另一个问题考虑二维定义域，值域也是二维，如何显示这4个维度的信息。这个问题的两个例子是：在长方形实数空间中基于向量的函数，或者是单一复变量的复变函数。图2-22表达了上面的两种例子：两个二元变量的一阶微分方程组（左）以及一个复变量的复变函数（右）。对于定义域是标准的二维空间的矩形域，采用这样的方法，考虑每一个数值都是由长度和方向所组成的向量，将整个值域编码成两个部分。用颜色表示一个向量或者复数作为颜色，而向量或者复数的方向则是用固定长度的方向向量来表示。不同的概念可以有很接近的表达方式的事实正是由于基于向量数值的结果具有相似性，并且强调了表达经验的重要性。

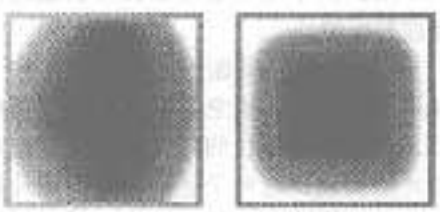


图2-22 两个可视化范围：一个复变量的函数（左），一个微分方程（右）

图2-22中表示的是基本的2D图像，其中函数的定义域由显示窗口表示，函数的值域由定义域内的颜色以及向量的方向表示。如果值域的维度超过了2，可视化结果是类似的，这类问题的解决方案是把向量用包含了更多信息的向量来代替。那样一个对象就称为字形，这是一个相对比较复杂的绘图符号，它可以用不同的方法进行变化，以同时显示几个变量的值。图2-23显示了一个例子，其模型来源于Donna Cox的工作[ELL]，通过三角形显示温度（颜色梯度来源于第5章的介绍），压力（长度），方向（对象的方向）以及在不同的时间注入模板（高

度）的方法来生成。这些对象放置在指标可以测量的地方。字形是抽象的，需要用户仔细地设计，这是由于它们用一种很复杂的方式包含了形状以及颜色的
102 信息。然而，字形对于传送大量的信息却是非常高效的，特别是当整个可视化的过程是动态的，并且要用动画的形式来表达。

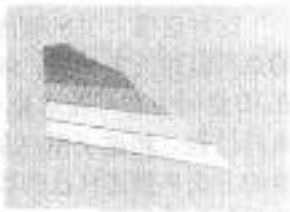


图2-23 在应用程序中可能用到的一个字形

当然，还有其他的方法可以处理更高维度的问题。其中有一种扩展投影的概念。我们知道在标准的3D图形学中从三维的视点空间可投影到二维的视图空间，用户也可以考虑从4维或者更高维度投影到三维空间中，其中的方法是很类似的。图2-24就是这样的一个例子，这是一个超立方体（四维立方体）的图像，它的每一个顶点有坐标（x，y，z，w），并且每一个分量都是1或者－1。在这个特殊的例子中，每一个在4维空间中的顶点都通过把第四维坐标的一个常量按比例变换到每一个其他的坐标中的方法来完成投影到3维空间的过程，这实际上就是4维空间正交投影到3维空间：

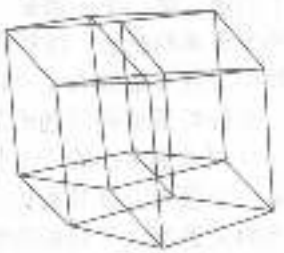


图2-24 投影到3维空间的超立方体

```
103 newVertex[0] = oldVertex[0]-0.3*oldVertex[3];
    newVertex[1] = oldVertex[1]-0.3*oldVertex[3];
    newVertex[2] = oldVertex[2]-0.3*oldVertex[3];
```

当然，其他种类的投影也是可以的。如图2-24所示，4维立方体首先在4维空间内旋转，然后投影到3维空间内。该例子的代码在本书的资源里可以找到。

还有很多关于维度的含义，也有很多不同的方法来表达以及展示不同的维度。如果用户有一个可以包含超过3维数据的简单模型，请在设计场景的时候仔细地看看选项。

2.12 图例和标签

图例和标签是文字或者简单的图形，它们在一幅图像中的作用就是传达图像和模型的信息给观察者。它们是用户生成模型的一部分，下面给出一个例子，该例子在第9章中有更详细的介绍。例子模型是传染性疾病通过传播的扩散过程，里面包含了图例以及标签来帮助观察者理解图像。

图例是很小的图形，包含文字和一些图像来帮助观察者理解场景的含义。它帮助观众理解用户图像中的信息，同时，在帮助观众理解展示的内容时，应该提供图例，如果使用伪色彩，就要显示颜色的标度来帮助观察者转化颜色信息。这样可以让人们理解颜色和问题内容之间的关系，这对于区别漂亮的图片和真实的信息而言是非常重要的一个部分。生成没有标度或者图例的图像容易引起可视误导。

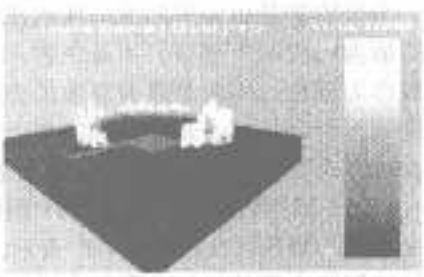


图2-25 一幅带有标签和图例的图像，提供理解信息

单一幅图像只能展示出它所要表达信息的一部分思想。标签是显示在屏幕上的文字信息。图2-25在主视口中显示了一幅带标签的图像（注解中说明此图像是关于疾病传播的）以及在主显示右侧的一个单独视口中的图例（注解说明了什么颜色表示什么含义，以及如何将颜色转变成对应的数字）。该标签将图像放置到适用的上
104 下文中，图例显示不同的颜色，告知在一个区域内感染了疾病的人数。这样，帮助观察者理解在主图中升降条仿真的结果，显示了疾病从一个单独

的初始点是如何传播的。

标签的另一个非常有用的形式就是放置在场景中布告板上的文字。布告板是在空间的简单2D长方形，它总是朝着观察者的方向，里面包含了一些纹理映射方面的信息。布告板在第8章的纹理映射中将给予介绍。它可以在场景中产生浮动文字的效果。如果它和直线或者描绘从布告板到特殊对象的箭头相结合，这是一种将某些物体表现成动画或者场景中的交互的有效方法。

2.13 精确度

当用户用图像表示模型时必须考虑表达的精确度。精确度是建模的一个部分，对于其他人理解图像也是非常重要的。用户用图像展示要表达的信息，让观众清楚、准确地理解信息。数据不能超出可以提供信息的范围之外。表现漂亮的图像是为了尽可能精确地表达数据或者理论。当然，制作吸引人的以及精确的图像是有用的，特别是当用户生成针对公共展示图像的时候。

精确度的关键是理解数据或者理论告诉了用户什么，而什么信息是没有提供的。用户可以制作演示来补充强调理论或者数据所没有提供的信息。对于数据的表达而言，该问题是相当直接的。如果数据是在规则的网格中组织的，那么使用基于多边形的网格来展示数据是非常直接的方法。如果用户的数据排列得不那么规则，那么对于大多数图形API而言可能就不得不生成多边形的数据表示形式了。如果用户不知道取得的数据是平滑变化的，那么就需要用离散数值表达，并且不能使用平滑插值或者平滑着色处理。如果数据在空间离散分布，那么最好的方法就是在采样的空间中显示数据，该采样空间在样本点上使用不同的尺寸或者不同的颜色来显示那些位置上点的数值。

另一方面，当用户把理论上的概念显示出来的时候，他可能会在一些点上计算出需要显示信息的精确数值。但是在这些点之间的动态过渡行为却不能得到精确的求解，这是很常见的，比如理论上可能会产生微分方程组没有闭合形式解的情况（即解不是精确的表达式）。这里用户就必须相当仔细地注意这些操作的数值解，并且要保证提供足够的精确度，特别地，要保证当用户的求解点从已知的数值上偏离一些的时候，求出的解不能与精确解之间出现分叉。用户拥有较多的关于数值计算的知识是很有用的，这样就可以从理论的表达中得到解的精确数值表达形式。 105

2.14 场景图和建模图

在这一章里，我们已经把建模定义成在场景中组织几何体的行为。现代图形API对于绘制图像可以提供大量的帮助，但是对于建模的支持通常较少，程序员在开始计算机图形学的工作时可能会遇到建模的困难。用变换的方法组织一个场景，特别是当该场景是层次结构，同时其中一些组成部分是变化的，就会比较复杂，需要为创建一个成功的场景特别仔细地进行组织。特别是当视点就是变化物体中的一部分的时候，情况就变得更加复杂了，如果视点相对于其他移动物体定义的时候，复杂程度更高。层次式的建模常常通过树或者类似树的结构来组织模型的各个分量，我们在后面会发现该方法是非常有用的。

近期的图形系统，比如Java3D和VRML 2，用已经形式化了的场景图作为场景建模和场景绘制组织的有力工具。通过理解和对场景图结构的适应，我们可以组织一棵树来解决层次模型的设计和实现的问题。这样的工具既能够管理模型中几何体的建模，又能够管理这些模型的行为以及它们各个分量的动画和交互控制。本节介绍场景图的概念和结构，以及针对一个

比较简单的建模图采用场景图的方法来设计场景。同时展示建模图是如何用三个关键变换来生成一个场景：即投影变换、视图变换以及模型变换。它的结构是通用的，在定义场景的时候可以管理所有基本规则，并且针对图形API转换成合适的代码。其中的术语基于Java3D的场景图，同时应该帮助所有使用系统的人理解场景图工作的方法。

2.15 场景图的概要

Java3D的API开发的场景图有很多部分，非常复杂以至于难以理解，我们将它抽象化，并用开发代码实现建模。图2-25中Java3D场景图的概要可以帮助我们讨论图结构建模的通用方法，它可以用在计算机图形学初级阶段。请注意该图的部分结构已被简化。

一个虚拟的宇宙有一个或者多个场景，这些是宇宙中放置场景图的位置。每一个场景图都有两个分支：一个是内容分支，它包含了形状、光源，以及其他内容；另一个是视图分支，它包含了视图的信息。这种区分是很灵活的，我们使用的是标准的建立框架方法。

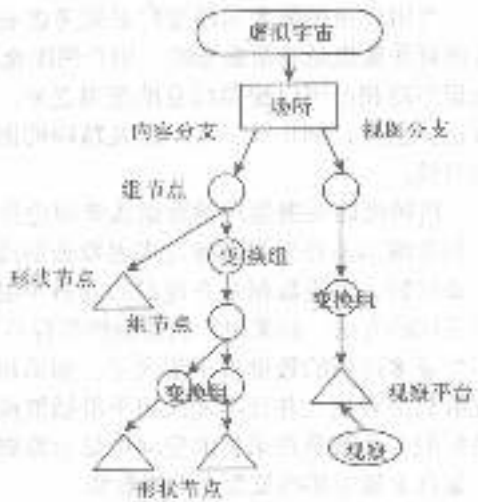


图2-26 在Java3D中定义的场景图的结构图

场景图中的内容分支是节点的集合，包括了组节点、变换组以及形状节点，如图2-26中左边的分支所示。组节点是任意多孩子的群组结构，除了简单的组织孩子节点的功能以外，组节点还可以包括一个开关，用以选择在场景中出现的孩子节点。变换组是模型变换的集合，将影响到在它下面的所有几何体。变换将应用于任何变换组的孩子节点，其中遵循“接近”几何体（定义在形状节点中，是图中的叶子节点）的变换首先应用的规则。一般来说，变换组在场景图中的表达方式与通用的组节点是相同的，即便它们的函数很不一样。形状节点包括单独图形单元的几何和外观数据。几何数据包括标准的3D坐标、法线以及纹理坐标，包括了点、线、三角形和四边形，以及三角条带、三角扇形和四边形条带。外观数据包括颜色、着色处理以及纹理信息。光照和视点作为特殊的几何体包括在内容分支中，它有位置、方向以及其他参数。场景图也包含共享组或者多个图形分支的组。共享组是形体的组，以非直接的方式包括在图中，并且对共享组的任何改动都会影响到组里的所有对象。这样就允许场景图包括基于模板建模的各个类型，这些类型在图形应用中经常出现。场景图的符号是这样的：圆圈代表组节点，椭圆形代表变换或者变换组，三角形则代表形状节点。

106

场景图的视图分支包含显示设备的规范，针对设备的合适投影显示在图2-26的右侧分支中，它同时规定了用户的位置以及在场景中的方向，并且包含了可以使用的不同显示设备的抽象。它支持同一个场景在不同设备上的显示，包括复杂的虚拟现实显示设备。部分对设备的支持来源于对该设备采用合适的投影变换。这比简单建模方法要适用的多。我们把视点看作场景中几何体的一部分，因此，可以通过将视点包括在内容分支中来设置视图，同时放置视点的变换信息生成视图分支中的视图变换。这一点很重要，在工作中，它能制作出视图分支来简单表达投影操作。

绘制场景时，除了使用场景图建模以外，Java3D也用它来组织处理过程。由于场景图是自底向上处理的，因此内容分支最先被处理，紧接着是视图变换，然后是投影变换。然而，

Java3D不能保证处理节点分支的序列顺序，因此可以通过选择处理的顺序来优化过程以提高效率，或者通过网络和多处理器系统来分布计算。因此，Java3D的程序员必须十分仔细地处理形状节点。在本书中，我们将只用场景图来开发建模的代码，不要求系统来处理场景图，开发一个简单场景图的剖析器留作一个项目，让用户来完成。 107

2.15.1 场景图中的裁剪

裁剪是图形学中较为特殊的操作，因为它定义了可见性，而不是几何体本身。不同的图形API可能会用不同的方法处理裁剪，但是用户可以在场景图中的任何地方设置裁剪，并且在坐标点的定义之后的任何地方应用裁剪。假定用户在遍历图的时候，"没有设置"裁剪，而返回到最初设置它的位置。

在标准的场景图中没有特别的符号来设置以及取消对裁剪的设置。用户可以定义自己的符号或者在图中用非正式的注解表明"这里我们设置了一个裁剪平面"。当然，如果用户要正式地定义场景图的数据结构以及通过代码操作它，那么就需要比较正式地处理它，这样图的剖析器才能完整正确地执行。这是本章后面中项目1的主题。

2.15.2 用场景图建模的例子

我们将针对两个例子场景开发场景图来显示该过程是如何工作的。首先，我们展示了完整的场景，然后分析它们是如何生成的，并且看看场景图是如何给我们提供其他展示场景的方法的。

第一个简单的例子重点强调了变换。图2-27制作一个简单的兔子头模型。该场景包含一个大的椭球型头、两个小的球形眼睛，还有两个中等大小的椭球形的耳朵。我们用椭球体（实际上是一个经过了缩放的球体，就像我们以前看到的）作为基本的部分，然后用模型变换把这个对象按不同方向和不同颜色放到不同的位置。 108



图2-27 兔子头

兔子头的建模图如图2-28所示。这里包含了所有组成部分（眼睛、耳朵等主要的部分）形成一个单元所需要的变换。基本几何体是球体。请注意，针对左耳和右耳的变换包括了旋转操作，这些可以通过使用单参数角度旋转变换产生兔子耳朵的前后摆动效果。

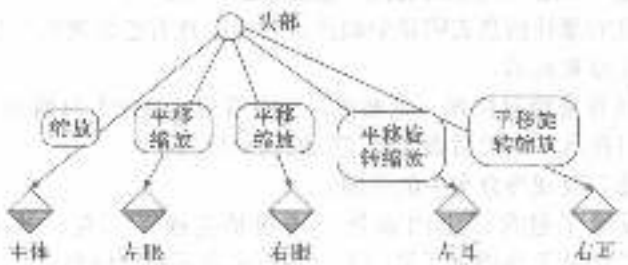


图2-28 兔子耳朵的建模图

此图中的变换特别是针对耳朵的变换可以通过参数来生成动画。这将在下文中讨论，考虑使用建模图来编写场景的代码。

图2-29中所示的场景表示一架直升机在地形图上飞行，场景是从一个固定视点观察的。该场景包含两个基本的物体：一架直升机以及一个地形的曲面。直升机由机身和两个螺旋桨组成，地形的曲面是通过多边形的集合来建模的（数据来源于俄勒冈州Crater火山湖的3D地 109

图）。该场景有一些层次性。因为直升机是由一些更小的部分组成的，场景图可以帮助我们识别这种层次性。因此，我们可以在绘制场景时，与场景图一起工作。另外，场景中包含一个光源和一个视点，这两者都在固定的位置。对该场景进行建模的第一个工作已经完成：识别出场景中的各个部分，将它们组织成一个对象的层次集合，然后将这个对象集合放置到视图背景中。下一步，我们将识别出地形中各个部分之间的关系，这样就可以生成树来表示该场景。在这个例子中，这种关系存在于地面和直升机的各个部分之间。最后，我们必须将所有这些信息放置到图像中。



图2-29 场景图描述的一个场景，参见彩图

场景的初始分析请见图2-29。另外，沿视线方向和内容分支进行组织。图2-30显示该图结构的初始草图。该图的内容分支包含了建模过程中各个成分的组织结构。该图的视图分支对应投影和视图。指定视点的位置和方向来完成视图变换以及使用的投影变换。

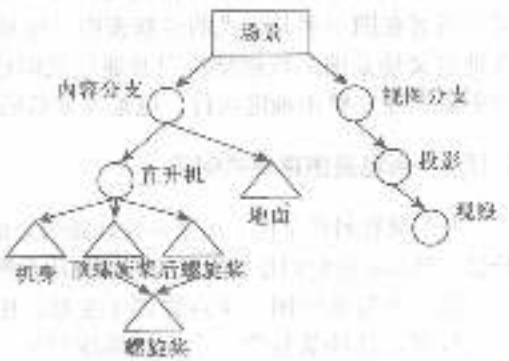


图2-30 组织简单场景建模的场景图

最初的结构与我们已经看到的简单OpenGL视图和建模的方法相兼容。在这里视图用API函数通过设置视点来实现，建模则由简单的基元和变换构建。然而，该方法只把我们带到那么远，这是因为它没有将眼睛集成到场景图中。如果将视点嵌入场景中，并且随着其他内容而移动，那么计算视图函数的参数是比较困难的，这将在绘制场景组成部分的另一个例子中讨论。

我们已经开始定义场景图，但几乎没有完成。图2-30只定义了部分场景，并没有完成所有的工作，而仅描述了那些部分与哪些其他部分之间的联系。为了能将它扩展为一个更加完整的场景图，我们必须增加如下内容：

- 在组节点中描述项目之间关系的变换，需要单独应用到每一个分支中。
- 每一个形状节点的属性信息表明诸如颜色、纹理以及着色处理的信息，通过图2-31中那些节点的阴影部分来表示。
- 光照和眼睛的位置是绝对位置（就如图2-30中所用，如图2-31所示），或者是相对于模型中其他分量（在这一章的后面会描述）的相对位置。
- 投影的技术规范以及视图分支中的视图。

110

场景图的扩展版除了包含以上内容外，还包括变换、外观、视点以及光照，如图2-31所示。我们已经给形状节点增加了第二个三角形来表示外观信息。

图中的内容分支处理建模过程，非常像场景图中的内容分支。图2-30中包括了图的几何节点，也包括了外观的信息。显式的变换节点以正确的大小、位置以及方向来放置几何体，组节点组装成内容进入逻辑群组。其中也包括光照和视点，并在固定位置上进行显示。在一些模型中，光源或者视点可能会和一个组相联系，这样就不用单独放置了；但这样会产生一些有趣的情况，我们在后面的一个例子中会介绍。这个例子给出了形状节点的几何体，比如螺旋桨或者单独的部件为共享。这可以实现，比如通过在函数中定义共享形状节点的几何体来实现。使用时从每一个螺旋桨或者树节点来调用此函数。

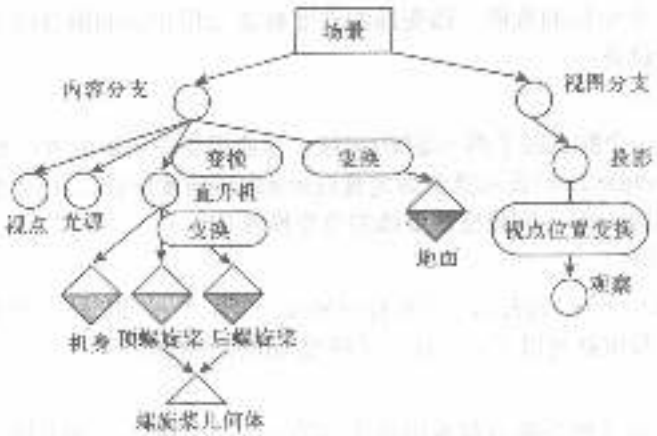


图2-31 包含变换以及外观属性的一个更加完整的图

图中的视图分支和场景图中的视图分支是类似的，但是在处理上则要简单很多，只需要投影和视图分量就可以了。投影包括场景中的投影（正交投影或者透视投影）定义以及窗口和视口的定义。视图分量包括视图变换所需要的信息，由于视点放置在内容分支中，这是变换集合的一个简单副本，同时场景中的视点位置也在内容分支中表示。

形状节点的属性包括颜色、光照、着色处理、纹理映射以及其他性质，我们会在第5、6和19章中介绍。形状节点的每一个顶点不仅有顶点坐标，还有法向分量、纹理坐标以及其他信息。这里只关注形状节点的几何体，后面各章会介绍其他属性，诸如外观等，请注意，外观内容可能是高质量图像中最重要的部分了。

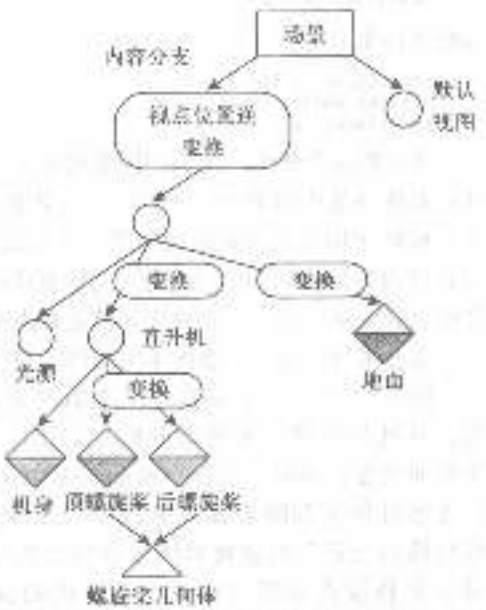


图2-32 将视图变换集成到内容分支以后的场景图

用户能通过变换的集合在场景中放置视点，也可以将视图变换定义成变换的逆。以放置视点，并且简单地使用场景中的默认视图。可以在内容分支的顶部计算并放置视图变换。将图2-31中的场景图调整成图2-32中所示的样子，这样它就可以接受任何视点位置。这个位置可以是固定的也可以是移动的，同时它可以相对于整个场景或者相对于场景中的任意部分。当我们讨论如何管理场景中的动态部分的视点时，这将是个关键点。

2.16 视图变换

在没有指定视点的场景图上，默认视点放置在原点，朝向z的负方向，并且y坐标轴向上。如果变换改变了视点的位置，可以通过逆变换，将视点恢复到默认位置上。该逆过程在变换序列中放置视点的位置，然后以逆向的顺序反向放置原始的变换，比如$T_1T_2T_3\cdots T_n$是原始的变换序列，该变换的逆变换就是$T_n^{-1}\cdots T_3^{-1}T_2^{-1}T_1^{-1}$，这里$T^{-1}$是$T$的逆变换。

缩放、旋转以及平移等基本变换的逆变换是容易求解的。缩放变换scale(Sx,Sy,Sz)表示用

三个缩放因子乘以三个坐标的数值，该变换的逆变换表示用相同的缩放因子来除每一个坐标值，因此它的逆变换就是：

```
scale(1/Sx,1/Sy,1/Sz).
```

当然，只有当每一个缩放因子都不是0的时候，才能很快地求解缩放函数的逆变换。

旋转变换rotate(angle,line)表示绕着固定直线line旋转角度angle，该变换的逆变换表示绕着该直线反向旋转角度angle，因此旋转变换的逆变换就是：

```
rotate(-angle,line).
```

平移变换translate(Tx,Ty,Tz)表示三个坐标分别加上了三个平移数值，逆变换只需要在每一个坐标上减去这三个数值就可以了，因此，平移变换的逆变换是：

```
translate(-Tx,-Ty,-Tz).
```

根据操作顺序，将这些变换合起来组成上面的复合逆变换。举例来说，操作集合的逆变换（用我们以前写过的伪代码）是这样的：

```
translate(Tx, Ty, Tz)
rotate(angle, line)
scale(Sx, Sy, Sz)
```

其逆操作集合是

```
scale(1/Sx, 1/Sy, 1/Sz)
rotate(-angle, line)
translate(-Tx, -Ty, -Tz)
```

现在将这个概念应用到视图变换中去。从树形结构中得到视点变换是最直接的方法。由于把视点看作场景中内容分量中的一员，因此可以把视点放置在相对于场景分量中的任何位置。可以沿着从内容分支的根节点到视点之间的路径来获得视点的变换序列，该变换序列就是视点的变换，可以通过求该序列的逆变换来生成视图变换。



图2-33 这是与图2-29相同的场景，但是，这里的视点在直升机的后面，请参见彩图

图2-33展示了定义直升机之后马上定义视点，并且从该视点观察直升机。看图2-29中视图如何改变。另外，图2-34展示了图2-31中的场景图是如何实现随着视点的改变而改变的。视点变换由把直升机放置到场景中的变换所组成，
113 接下来将视点放置到相对于直升机的位置上。相对于直升机的视点变换必须在定义直升机的空间内构建。视图变换是视点位置变换的逆变换，在这里相对于直升机放置视点的逆变换，接下去是将直升机放置到场景中的逆变换。

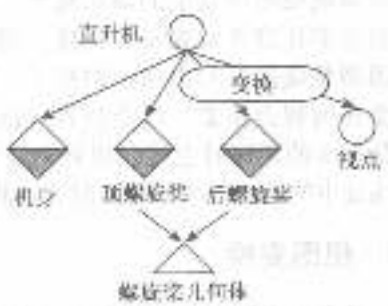


图2-34 这个图是对图2-30的场景图所作的修改，实现图2-33中的视图

在这个场景图中，可以用变换的集合$T_1, T_2, T_3, T_4 \cdots T_i T_j T_k$将直升机放置到场景中，并且用变换$T_a, T_b \cdots T_z$来将视点相对于直升机而放置。对于图2-31中结构的实现，首先用标准的视图显示代码，接下去是$T_z^{-1} \cdots T_b^{-1} T_a^{-1}$，再接下去是$T_k^{-1} T_j^{-1} T_i^{-1} \cdots$
114 $T_4^{-1} T_3^{-1} T_2^{-1} T_1^{-1}$，最后绘制图2-33中的标准场景。

视点位置的变换意味着引起视点变换的集合必须进行相应的修改，但是在定义场景图之前的变换的逆变换机制仍旧起作用；因此只有实际上被逆转的变换才会改变。这就是前面介

绍的场景图是如何帮助用户组织观察过程的。用户可以交互，比如说，用菜单切换选择视点在一个固定的点上还是在一个跟随直升机的点上；然后实现视点位置逆变换的代码实现合适的变换，这将取决于菜单上的选择。

由于视图变换在模型变换之前，从图2-32中可以看到，视点的逆变换必须在内容分支得到分析之后才能应用，并且它的操作过程也放置在代码中。这意味着显示操作必须以视点放置变换的逆变换作为开始，即将视点移动到内容分支的顶部，可以将视点路径的逆变换放置在每一个形体节点的变换集合的前面。

产生一幅图像的场景图是不唯一的，因为有很多方法可以组织一个场景，同时有很多方法组织如何实现场景图中所指定的操作。第1张场景图生成以后，用户会考虑是否有其他方法来组织场景图，让程序更高效，或提供对场景更清晰地描述。记住，场景图是一个设计工具，对于任何问题，都存在着很多方式来实现一个设计。

我们需要从场景图中的三种主要变换方式中提取信息来生成代码，从而实现我们的建模。投影变换是一种直接的方式，它是由在视图分支中的投影信息构建的。这很容易由图形API中的工具来管理。当用户分析视点放置变换的时候，视图变换是很容易在视图中通过变换的信息来生成的，这一点我们在前面已经提到过，不仅要取这一点是很直接的，更重要的是，从视点放置位置变换的逆变换中生成该信息。最后当绘制各个分量的时候，针对不同分量的模型变换，通过在内容分支中的不同变换来构建它们。

理解场景图可以描述一个动态的几何体是非常重要的，场景图中的变换参数可以是变量而非常量，事件的回调操作可以通过用户交互或者计算数值来控制。这让我们在场景图中捕获了几何体以及它们的行为。变换的使用在下一节继续讨论。这样一张单独的图就可以描述一个动态的场景，甚至能改变场景的各个视图，并且能够设计在场景中的用户输入。图中的一些部分可以视为带有外部控制器，这些控制器是事件回调的函数。

图中所有基础几何信息、变换以及行为，都称为场景中的建模图。建模图本质上是一个没有视图分支的场景图，但是在顶层组织有视图的信息作为视点放置变换的逆变换，它会成为场景编码的基础，我们会在本章的剩余部分对它进行叙述。 115

2.17 场景图和深度测试

前面创建的图像几乎都用到了图形API中提供的隐藏面功能。就如在第1章中所描述的，这需要用到深度缓存或者Z缓存，对于隐藏面的深度比较是对绘制场景中的每一个像素进行的。然而，有些时候用户想避免深度测试来控制场景中分量的序列，在第5章中我们介绍一个例子，从后往前的绘制序列通过混合操作来模拟半透明的效果。用户需要知道场景中的每一个片断的深度，或者每一个片断距离视点的距离。如果该场景是完全静态的，那么这样做是很简单的；但是，当片断是可以移动的或者视点是可以移动的时候，它将变得不那么容易了。

解决该问题需要做一些额外的工作。在更新变换以及绘制场景的选择之后，在实际绘制之前，用称为投影的工具，该工具可以在极大多数的图形API中找到。当视图和投影变换映射到3D的视点空间内的时候，它会计算在模型空间内任何点的坐标。点的深度是投影的Z坐标数值。使用投影操作以替换绘制操作来处理整个场景，得到场景中每一个片段的深度数值，接着使用该深度数值来决定各个部分的绘制顺序。场景图可以保证投影变换的正确性，同时确保获得正确的深度数值。

2.18 用建模图写代码

由于建模图被视为一个学习的工具而非一个生产工具，当介绍这个概念的时候，我们会

避免使用介绍过的术语项范围来给它定义：

- 形状节点包含了两个分量
  几何内容
  属性内容
- 变换节点
- 组节点
- 投影节点
- 光照节点
- 视点节点

我们不希望场景是通过自动解析建模图生成的，尽管能够这样做，而是用户用图组织结构和模型中的关系编写代码，来实现简单的或者具有层次关系的建模。

一旦用户组织了在建模图中的所有模型分量之后，需要编写代码来实现该模型。这很简单，可以有一个重入代码允许将图结构变成程序代码。在这个规则集中，首先假定应用的变换是申明的反向顺序，就像在OpenGL中的那样。这应该与用户在程序课程中处理树的经验相
116 一致，因为用户此时已经讨论过一棵表达树，该树是以叶子优先顺序（后序）遍历的。同时它也和Java3D和OpenGL的规范相一致，在那些API中，距离几何体最“接近”的变换（在场景图中嵌套最深的）是最先被应用的。

非正式的重写方针是这样的，包括了针对视图分支和内容分支的重写规则：

- 在视图分支中，节点仅涉及窗口、视口、投影以及视图变换。窗口、视口以及投影是由API中的简单函数来处理的，并且应该在显示函数或者重定位函数的顶端。
- 视图变换是由内容分支中的视点变换来构建的，通过逆变换将视点放置在内容分支的顶端。该序列应该是显示函数中的下一步。
- 建模图中的内容分支通常在显示函数中得到维护。但是它的一些部分可能通过其他在显示中的函数调用来处理，这取决于场景的设计。定义数个对象的几何体的函数可能由一个或者更多的形状节点来使用。而基本的建模可能通过事件的回调而受参数集合的影响，其中包括视点、光源或者在视图中显示的对象的选择。
- 组节点将一些元素组合到一个单独的对象中。每一个单独的对象来自于组节点中不同的分支。在编写包含一个变换组的分支代码之前，用户需要保存模型变换的状态，该过程需要在用户遍历图的时候，并在需要返回的分支之前完成，由于每一个变换元素的简单特征，如果需要就撤销视图变换是很直接的办法。这可以通过一个变换栈来处理，它通过压入栈的操作来保存当前的变换，并且通过弹出栈来恢复该变换。
- 变换节点包含用常规方法实现的平移、旋转以及缩放操作，同时也包含部分动画的变换或者用户控制的变换。从建模图中获得信息来编写程序代码的时候，用户需要用出现在树中的相同序列来写变换。因为变换在设计中具有自底向上的特性，这与变换的最后定义－最先使用的顺序相对应。
- 形状节点涉及几何体及其属性，并且外观属性必须在几何体定义之前设置，因为绘制几何体时，用到外观属性。

属性节点可以包含纹理、颜色、混合或者材质说明等外观属性，用于控制几何体的绘制
117 以及在场景中的出现。

几何节点包含顶点信息、法向信息以及几何结构信息，比如条带或者扇形组织。

- 极大多数在内容分支中的节点可以被交互或者其他事件驱动的活动所影响。这可以通过用参数定义的内容来实现，事件的回调可修改这些参数。参数可以控制位置（通过

对旋转或者平移的参数化），方向（通过旋转参数化）、尺寸大小（缩放参数化）、外观（外观细节的参数化），甚至是内容（组节点中对切换进行参数化）。

我们有一些来源于以下各节的建模图来编写图形代码的例子，请注意寻找规则。

2.18.1 两个场景图的代码实例

针对图2-27中简单兔子头的例子，我们重复了图2-35中的建模图，显示在图2-28中。该图包含了所有组成部分（眼睛、耳朵以及主要部分）形成一个单元所需要的变换。这些部分的基础几何体是球体。请注意兔子的左边和右边的耳朵变换包含了旋转，设计兔子耳朵旋转角度作为参数，让耳朵能够前后摆动。

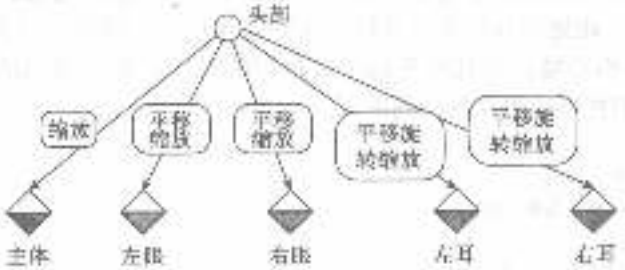


图2-35 兔子头的建模图

为了从建模图编写兔子头的代码，我们对模型变换栈应用以下的操作序列：

1. 将模型变换压入栈。
2. 应用该变换生成头部，并且定义头部：缩放，绘制球体。
3. 将模型变换弹出栈。
4. 将模型变换压入栈。
5. 应用变换放置头部中左眼，并且定义左眼：平移，缩放，绘制球体。
6. 将模型变换弹出栈。
7. 将模型变换压入栈。
8. 应用变换放置头部中右眼，并且定义右眼：平移，缩放，绘制球体。
9. 将模型变换弹出栈。
10. 将模型变换压入栈。
11. 应用该变换放置头部中的左耳，并且定义左耳：平移，旋转，缩放，绘制球体。
12. 将模型变换弹出栈。
13. 将模型变换压入栈。
14. 应用该变换放置头部中的右耳，并且定义右耳：平移，旋转，缩放，绘制球体。
15. 将模型变换弹出栈。

118

仔细地跟踪操作序列，观察兔子的头部是如何绘制的。如果想把兔子的头部放置到身体上，就需要把整个操作的集合作为一个单独的函数rabbitHead()，同时在压入和弹出堆栈中调用该函数，另外编写代码来放置头部，然后在调用函数之前移动它。这就是层次建模的基本思想——由其他对象来生成对象，最后在最底层将模型简化为简单的几何体。对于建模图而言，最底层就是在形状节点中树的叶子。

变换栈对于场景图是很重要的。它可以由图形API提供，也可以由用户自己写代码；即使由API提供，对于变换栈的深度也会有限制，这对于一些项目而言是不合适的，用户需要生成

自己的堆栈。我们会在第3章中讨论用OpenGL的API实现这些变换。

在图2-33直升机的例子中，我们会用上面介绍过的重写规则来遍历树，以编写程序代码，请见下面展示的代码主干。省略大部分细节，比如视点放置变换的倒置，模型变换的参数以及各对象的外观细节，但我们用缩进来展示压入和弹出模型变换堆栈，这样读者可以很容易看到在压入和弹出栈之间的操作过程。

该例子中加入动画是很简单的。可以在定义平面内增加额外的旋转来实现螺旋桨的动画，该操作需要紧接在缩放的后面，同时在变换之前将螺旋桨放置在直升机上，同时在每一次空闲事件回调执行的时候通过更新额外旋转的角度来实现上述动作。这些操作都体现在代码中。直升机的行为本身可以通过更新变换参数来实现位置的动画效果，同样道理，在空闲回调函数中进行更新。如果位置的变换参数是通过用户交互事件的回调来更新的，那么直升机的行为也可以由用户控制。此处没有包含直升机行为的程序代码，用户可以自己设计。因此，将该图表达成一个动态的环境，并且从开始生成模型的时候就包含动态的过程。图2-33和图
119 2-34中的建模在下面的程序代码片断中列出。

```
display()
    设置视口以及前景的投影
    将模型视图矩阵初始化为单位阵
    定义视图变换
        反转设置视野位置的变换
    通过gluLookAt的默认值来设置视点
    定义光源的位置                          //注意使用绝对坐标
    压入变换栈                              //地面
        平移
        旋转
        缩放
        定义地面的外观（纹理）
        绘制地面
    弹出变换栈
    压入变换栈                              //直升机
        平移
        旋转
        缩放
        压入变换栈                          //顶部的螺旋桨
            平移
            旋转                            //放置
            旋转                            //移动
            缩放
            定义顶部螺旋桨的外观
            绘制顶部螺旋桨
        弹出变换栈
        压入变换栈                          //后面的螺旋桨
            平移
            旋转                            //放置
            旋转                            //移动
            缩放
            定义后面螺旋桨的外观
            绘制后面的螺旋桨
        弹出变换栈
        //假设机身没有变换
```

定义机身的外观
绘制机身
弹出变换栈
前后缓存交换

在该场景中的其他变形中，用户可以通过修改从一个固定点到相对于地面的位置（将光源视为包含地面的分支组的一部分）或者相对于直升机的（将光源视为包含直升机分支组的一部分）光源位置的程序代码来实现。类似地，视点可以放置在相对于场景中的另一个部分中，或者放置在通过用户交互控制的变换中，在事件的回调函数中可以设置变换的参数。

需要提醒读者，应该在每一个形状节点中包含外观内容。在图形API中，很多外观参数都保存了状态，即一个形体的参数集合一直保留到被重置为新形状之前。用户可以将场景设计成共享的外观，可以连续使用，以提高绘制场景的效率，但这是一个专业化的组织形式，会和一些API产生矛盾，比如Java3D。因此，对于每一个形状体，重置外观是很重要的，在场景中的后面部分展现的时候，如果有用户不想要的对象，那么上面的操作就可以避免偶然地保留其外观的情况发生。 120

2.18.2 使用标准的对象生成更加复杂的场景

构建兔子头的方法实际上是一个大得多的例子中的一部分——使用一个标准对象的集合来定义一个大的对象。在一个需要兔子场景的程序中，我们可以用函数rabbitHead()来生成兔子头，代码在前面介绍过，应用变换将兔子头放置在兔子身体的合适位置。兔子本身可能是一个大场景中的一部分，用户可以用这种方式来生成想要的复杂场景。

2.19 小结

这一章中讨论了基于多边形建模的所有概念，这在很多的图形API中都有涉及。如何在模型空间内（即针对物体的一个3D空间）以图形基元的形式比如点、直线段、三角形、四边形以及多边形来定义一个对象；如何应用缩放、平移和旋转等模型变换将物体放置到世界空间内，并应用视图和投影操作以及用场景图在一个场景中组织层次化对象，使用户可以很简单地编写场景的程序代码。同时描述了如何改变变换操作，将运动加入到场景中。现在，我们已经做好了一切准备使用OpenGL这个图形API来实现这些概念，用户可以进行图形编程了，关于编程的内容，我们将在下一章进行介绍。

2.20 思考题

1. 我们知道可以用三角形对多面体进行建模，但是，为什么要用三角扇形来给球体的两极建模，而用四方条带给球体的剩下部分建模呢？你可以想象出一个物体，用任何四边形都不能对它进行建模吗？
2. 将自己放置到一个熟悉的环境中，想像该环境简化了，只是由方形、圆柱体以及其他基本的形状组成。进一步想象，场景中只有一扇门，房间里所有的东西都只能从那扇门里面进入。请写出变换的序列，将环境中的所有物体各就各位。现在想像每一个基本的形状都从一个标准的形状通过变换而来：一个单位立方体，一个直径是1同时高度也是1的圆柱体等，写出从标准的形状变换到每一个需要的物体的变换序列。最后，如果门只接受基本的对象，请将这两个过程放到一起，写出完整的变换集合来生成对象，并且将对象放置到空间内。
3. 一般而言变换是不可交换的，但是，对于变换A和B可交换，即$A*B = B*A$。举个例子来说，如果两个变换是同种基本类型的（旋转、缩放或者平移），这时它们是可以交换的；如果A和B中有一个是单位阵，它们是可交换的。你是否可以找到其他变换的例子，它们是可交换的或者验证说法任何“混合组”的变换是不能交换的。 121

4. 思考周围的环境，识别一些过程，它们需要超过3个维度来显示。一个例子可以是在大学校园中或者在你所在区域内的不同位置和时间上，风的方向和大小（一个2D环境中有2D的数据，并且伴随着1D的时间）。对于每一个所识别过程，请写出一种方法对它进行建模，让它能够以某种方式进行显示；如果发现不能对整个过程进行显示，请描述你如何对该过程的一部分进行选择，以让它能够显示。
5. 在高维度视图的讨论中，我们展示了一个复变量z的复函数f的表达方式，同时也展示了用复变量$f(z)$的极坐标(r, θ)表达的函数值，其中方向θ作为向量，大小r作为颜色。请讨论这种表达方式的效率以及这些可视化手段是否展示了函数的特性。
6. 采用思考题2中描述的环境，写出一个场景图来描述整个场景，需要用在上一个思考题中所指定的基础形状以及变换。同时，请将视点放置在场景图中，开始的视点位置是站在门口面对室内的标准视野。现在请想象在空间中的桌子上，有一幅芭蕾舞旋转的画，请指出在场景图中可以处理该移动对象的变换方式。

2.21 练习题

1. 请计算以下简单规则多面体的顶点坐标：立方体、四面体和八面体。对于八面体和四面体，尝试使用球面坐标，并且将它们转化成矩形坐标；请参考第4章的建模来获得更多的细节信息。
2. 请验证对于任何的x、y、z以及w，点$(x/w, y/w, z/w, 1)$是(x, y, z, w)和$(0, 0, 0, 0)$组成的直线段和超平面$(a, b, c, 1)$的交点，a、b、c是任意值的交点。证明这意味着在4D空间内的完整直线可以用3D空间中用齐次坐标的一个单点来表示。
3. 请写出如何用六个四边形来定义一个立方体。请写出通过用两个四方形条带来定义一个立方体的方法，请改进上述结果。能够只用一个四方形条带来写一个立方体吗？
4. 请问能用三角形的集合来写出任意的多边形，无论多边形是凸的还是非凸的吗？更进一步，试问能用三角扇形来表达任意的凸多边形吗？在三角扇形中选择哪个顶点作为第一个顶点有关系吗？你能够想出一种方式来验证OpenGL是否用三角扇形的方式生成多边形吗？（提示：如果要用多边形来绘制一个非凸的多边形，请观察OpenGL是如何处理的。）
5. 一种对多边形的边进行反走样的方法是考虑一个像素中有多少部分被多边形所覆盖，然后基于该比例数值给该像素设置一个颜色。如果在2D空间内定义一个多边形，该多边形中有一边没有和坐标轴平行，那么反走样的工作量比较大。请找到一个在2D空间中与多边形的那条边相交的单位正方形，并且计算多边形位于该单位正方形内部的比例。如果该正方形代表一个像素，请指出该像素的颜色中有多少比例应该来自于多边形，又有多少比例来自于背景色。 122
6. 在图2-10中的球体上面有一个四边形，其法线的程序代码是不精确的，因为它使用了顶点的法线来代替四边形中央的法线方向。请问应该如何正确地计算该法线，使它是面的法线而不是一个顶点的法线？
7. 请设计一个简单的非对称的物体，这样可以区别上面的任意两个位置。对该物体应用简单的旋转、平移以及缩放，然后对该物体的形状变化进行描述。接着定义两个变化T_1和T_2，然后用$T_1 * T_2$以及$T_2 * T_1$顺序分别对物体进行变换，看看结果是否相同？注意，T_1和T_2是不同的变换，如果T_1和T_2是旋转，那么它们需要绕着不同的直线旋转。请问，如果T_1是缩放变换，那么，如果它对各个坐标进行不同的缩放，对结果有影响吗？
8. 请考虑由一些不同形状的对象所生成的一个场景来表达建筑物，并且在一个简单的网格中可以表示街道的地图。勾画出对此物的建模看起来像什么，包括用标签来定义它们，作为建筑物表达位置的地图，还有一个图例显示图中每一个形状体作为一个特别的建筑物。
9. 虽然不同的分支可以共享相同的形状对象，场景图和树的结构还是比较类似。作为树来说，它们可以非常简单地被遍历。请说明如何遍历一个场景图来保持先后面再前面的绘制顺序，这里的前提是已经知道场景图中各个物体的深度信息。
10. 请给兔子头增加一个嘴巴和一个舌头，然后对场景图进行修改，让兔子能伸出它的舌头并且在外面

晃动。

11. 设想一个酒会或者一个庆祝会定义一个场景图。该对象有一个圆形的盘作为地面，一个圆锥体作为桌顶。还有很多柱子将地面和桌顶连接起来。另外还有一圈动物在一个算上，而这个圆正好在地面上直径的外面。动物们都在与它们最接近的边上的点平行于圆上的切线方向。当酒会进行的时候，这些动物会上上下下以周期的方式运动。可以假定每一个动物都是一个基元而不要尝试着对它们建模，但是，需要仔细地定义所有的变换来建立这个酒会，并且正确地放置这些动物。

2.22 实验题

1. 从avalon.viewpoint.com网站上获得一些模型，使用文本编辑器来检查该模型文件。看文件中的几何体是如何存放的，这样就可以阅读该模型文件，并且将模型数据读入到程序中。一般而言，数据是作为三角形的序列或者其他可用的图形基元的形式存放的。
2. 写出思考题3中熟悉空间中的场景图的程序代码，包括管理视点逆变换的代码。现在请指出一条简单的视点路径，通过参数化一些变换来放置视点以生成该路径，建立该场景的动画，这样就好像是从一个移动的视点看过去的那样。如同我们在第1章中看到的问题，调用函数

```
glGetFloatv(GL_MODELVIEW_MATRIX,v)
```

123

可以存取16个模型视图矩阵中的实数值，并且将它们恢复到一个数组中。该数组定义为：

```
GLfloat v[4][4];
```

如果我们让视图变换保持默认的状态，每次应用一次模型变换，我们可以使用这种方法来得到不同的建模矩阵的数值。在下面的每一个思考题中，请确认模型变换是模型视图堆栈中唯一的元素。这可以通过在调用模型变换函数之前将模型视图矩阵设置成单位阵来达到这个目的。它会在很大程度上帮助读者写出一个函数，很好地显示一个4×4的数组阵列，这样就可以容易地看到它的各个元素了。在这一节中所生成的矩阵需要和第4章中介绍的缩放、旋转以及平移的矩阵作比较。

就像我们在第1章视图变换和投影变换中所做的那样，可以显式地操作模型变换。如果对于简单的变换得到了模型视图的矩阵，那就可以改变矩阵中合适的数值。将该模型视图矩阵设置成修改后的矩阵。可以用修改以后的矩阵重新绘制这幅图，并且与原始的效果作比较。再修改矩阵来看图像的效果，而不要局限于数值上的效果。请在这里使用默认的视图数值，这样它们就不会影响到OpenGL中的模型视图矩阵的模型变换。

3. 从简单的缩放开始，比如用glScalef(α，β，γ)这个函数的设置，得到模型视图矩阵的数值。应该能够将缩放的数值看成就是矩阵中的对角线上的各个数值。尝试着用不同的缩放因子，先得到，再用好看的格式打印出这个矩阵。
4. 像上面一样来做一个旋转变换，比如用glRotatef(α,x,y,z)函数来设置，这里x,y,z的值需要设置成能够经过原点的轴转α角度来描述该旋转。比如对于x轴而言，设置x = 1以及y = z = 0，调用一定的格式打印出该矩阵，并且通过三角函数中的角度，确定矩阵中的各个分量。提示：可以用一些简单的角度，比如30°、45°、60°等。
5. 像上面一样来做一个平移变换，比如用glTranslatef(α，β、γ)函数来设置，得到模型视图矩阵的数值。确定矩阵中列的平移数值，请尝试用不同的平移数值来看矩阵是如何变换的。
6. 前面已经看到了每一个单独的模型矩阵，现在请将它们结合起来，通过与结果矩阵作比较，来看如何建立复合变换。特别地，取出两个在上面已经检查过了的简单变换，将它们组合起来，看复合变换的矩阵结果是不是两个原始矩阵的乘积。提示：需要考虑矩阵乘积的顺序。
7. 复合变换是不可以交换的，矩阵的乘法是不可以交换这个事实也验证了我们的观点。然而，我们可以通过组合两个变换得到的结果矩阵来更加直接地验证这个观点，这可以通过交换矩阵相乘的顺序来验

证。这两个矩阵在大多数的情况下不应该相同，请读者注意。如果碰巧得到了两个相同的矩阵，请检
124 查所使用的变换是不是太简单了，如果是的话，请将它们变得再复杂一些，然后再试试。

2.23 大型作业

1. （场景图分析器）用图（或树）的数据结构定义一个场景图，其中的节点应有适当的属性或变换声明。请写一段遍历场景树的程序，它能自动将这些声明排列成适当的序列，并且将场景用图形API实表示。使用本章的伪代码，能看出该如何使用变换参数来生成动画吗？当视点不在标准位置时，又如何生成
125 绕视点位置变换的声明？

第3章　在OpenGL中实现建模

本章讨论使用OpenGL图形API实现上一章所讨论的建模功能。与上一章的内容是互补的，这一章的内容包括指定几何体的函数，为模型空间中的几何体指定顶点的函数，为这些顶点指定法线的函数，以及将几何对象从模型空间变换到世界坐标系的函数。还包括实现多边形的函数，以及上一章中所描述的几何体压缩。最后讨论OpenGL中预生成的几何模型以及GLUT，它们可以帮助读者更加方便地创建自己的场景。当读者学习完本章，应该能够使用OpenGL创建上一章中所看到的场景，不过本章并不涉及能够使用户的场景显得更加有趣的外观信息。这些将在后面有关光照、着色处理和纹理映射的章节中介绍。

为了理解本章的内容如何应用于OpenGL API的图形学程序，回忆在第1章中讨论的OpenGL视图模型以及在第0章中概述的一个基于OpenGL程序的例子。一般而言，我们希望在本章中出现的函数能在显示事件回调中用到，虽然这样做并不是必须的。

3.1　指定几何体的OpenGL模型

为了给程序定义一个模型，可以使用单个函数为OpenGL指定几何体。这个函数采用顶点列表的方式定义几何体，它的参数说明该几何体是如何被解释的：

```
glBegin(mode);
// 顶点列表：在mode所指定的绘制模式下创建基层对象的顶点数据
// 外观信息比如颜色，法线和纹理坐标也可以在这里指定
glEnd();
```


这种glBegin(mode)-顶点列表-glEnd()的方式使用不同的mode值来说明顶点列表创建图像的方式。绘制模式和顶点列表的解释将在后面介绍。因为程序员通常需要在一个场景中使用许多不同的几何对象，所以可能会反复多次地使用这种方式绘制不同模式的图形。全书中都可以看到应用这种方式的例子。

在OpenGL中，顶点数据是通过一系列以glVertex*(...)命名的函数来定义的，这些函数将顶点坐标值送入OpenGL流水线，转换成图像信息。glVertex*(...)代表的是一系列的仅顶点数据类型不同的函数，读者可以用任何标准数值类型指定顶点数据，这些函数会让系统响应不同的需要。

- 如果指定的顶点数据是3个实数或者浮点数（我们将使用变量名x、y和z，当然常量也是允许的），那么可以使用函数glVertex3f(x,y,z)。这里，数字3代表顶点在三维空间中，字符f代表参数是浮点数。其他的数据类型或维数也是可以的。
- 如果将坐标值数据定义在一个数组中，可以将数据声明成glFloat x[3]这样的形式，并使用glVertex3fv(x)来指定顶点。为函数名增加字母v就是为了指明数据保存在数组当中。

这些函数的其他版本可以指定在二维空间中点的坐标（glVertex2*）；在三维空间中可以用整型（glVertex3i）、双精度浮点（glVertex3d）或短整型（glVertex3s）；还可以用四维点（glVertex4*）。四维版使用的是上一章中介绍的齐次坐标，在本章中可以看到这些版本的代码示例。

在OpenGL中，glBegin(mode)和glEnd()之间可以调用自己的函数来指定顶点列表的顶点。

这样就允许读者在glBegin(mode)和glEnd()函数之间而不是它们之外作任何需要的计算以得到顶点坐标。例如，可以把很多不同类型的循环包含进来，计算顶点序列；也可以使用逻辑判断语句来决定生成哪些顶点。由函数glVertex*(...)定义的顶点将以指定的绘制模式添加到顶点列表中。

还有很多其他的信息可以放在glBegin(mode)和glEnd()这一对函数中间。我们将会在第6章中看到加入顶点法向量的重要性。在纹理映射的章节中看到纹理坐标的重要性。所以，虽然结构简单，但它并不是只能指定顶点。这里可用的OpenGL操作包括glVertex、glColor、glNormal、glTexCoord、glEvalCoord、glEvalPoint、glMaterial、glCallList以及glCallLists。当然这并不是全部的。

3.1.1 点和多点模型

127 画点是通过在glBegin函数中将模式设置为GL_POINTS，glBegin(GL_POINTS)和glEnd()之间的顶点数据作为所画点的坐标值。如果只需要画一个点，在glBegin和glEnd之间给出一个顶点即可；如果需要画很多点，就给出更多点。点的大小一般是一个像素，如果想让每个点看上去更明显一些，可以通过函数glPointSize(float size)来设置每个点的像素数，其中size是任意非负实数，默认值为1.0。

下面的代码片断生成由一系列点排列成的螺旋线。这段代码使用普通编程方式来定义几何体，说明当我们可以通过计算来获得点的坐标时，不需要手算。下面的代码是通过计算坐标值并在glBegin和glEnd间的for循环中调用glVertex*()函数来指定点的坐标。这些函数计算了绕z轴螺旋的点，其中x和y坐标通过简单的三角函数获得。代码的运行结果如图3-1中的左图所示。完整代码可以通过网络下载，它还包括通过键盘控制来旋转螺旋线的功能。用户可以试着根据自己了解的数学函数来画其他类似的曲线。这些特定的参数曲线将在第4章中讨论。

```
#define PI 3.14159
#define N 100.0
void pointSet(void) {
    int i;
    float step, zstep;

    step = 2.0*PI/N;
    zstep = 2.0/N;
    glPointSize(2.0);
    glBegin(GL_POINTS);
        for (i=0; i<(int)(3*N); i++)
        glVertex3f(2.0*sin(step*i),2.0*cos(step*i),-1.+zstep*i);
    glEnd();
}
```



128 图3-1 由上面的代码生成的三维空间中的螺旋线——点组成的螺旋线（左）和线段组成的螺旋线（右）

3.1.2 直线段

对glBegin/glEnd使用GL_LINES模式可以画出直线段。对每一条要画的直线段，定义这两

个端点。这样，glBegin(GL_LINES)和glEnd之间的顶点列表中的每一对顶点定义了一条直线段。通常所画直线段的宽度是一个像素，这可以通过调用glLineWidth(width)函数来改变，其中width为任意非负实数。如果指定了反走样线条，不同宽度的直线段的效果可能会不一样。下面的代码片断指定了四条简单的平行直线段：

```
glBegin(GL_LINES);
    glVertex3f(0., 0., 0.); glVertex3f(5., 5., 5.); // 第一条直线段
    glVertex3f(1., 0., 0.); glVertex3f(6., 5., 5.); // 第二条
    glVertex3f(0., 1., 0.); glVertex3f(5., 6., 5.); // 第三条
    glVertex3f(0., 0., 1.); glVertex3f(5., 5., 6.); // 第四条
glEnd();
```

3.1.3 线段序列

相连接的线在OpenGL中称作线段序列，使用模式GL_LINE_STRIP来指定。顶点列表定义了之前讨论过的线段序列。所以，如果定义了N个顶点，将得到$N-1$条直线段。无论对于线段还是线段序列，都可以设置线宽来加粗（或减细）一根线条。较大的直线宽度比较小的直线宽度提供更加显眼的效果。线宽同样通过函数glLineWidth(float width)来指定，width的默认值为1.0，但任何非负宽度都可以使用。

作为一个线段序列的例子，我们可以用上面的点螺旋线代码来生成一条参数曲线，只需要将GL_POINTS换成GL_LINE_STRIP即可改变绘制模式。这两条曲线如图3-1所示。在点生成的螺旋线中，步长的累进次数是100，而在直线段的例子中减少到20。从这样的简化中可以看出使用单独的线段比使用更小量化单位效果好。但用户可以看前一个例子的源代码，并用步长的累进次数或者参数的曲线方程来做试验。

3.1.4 封闭线段

封闭线段和线段序列类似，不同的是将连接第一个顶点和最后一个顶点生成一个封闭的环。封闭线段是通过GL_LINE_LOOP来指定的。

3.1.5 三角形

使用GL_TRIANGLES模式作为glBegin/glEnd的参数，可以画出不相连的三角形。它的处理在前一章中讨论过。它生成很多三角形，每三个顶点对应一个三角形的三个顶点。

3.1.6 三角形序列

OpenGL具有两种标准的针对三角形序列的几何体压缩技术：三角形条带和三角扇形。它们分别用GL_TRIANGLE_STRIP和GL_TRIANGLE_FAN指定glBegin/glEnd模式。这些在之前已经讨论过。

129

由于两三角形序列有两种不同的模式，我们下面看两个例子。第一个是三角扇形，用来定义这样的对象，它的顶点是从一个中心点放射出来的。它的一个典型例子是一个球体的底面或顶面，三角扇可以生成一个南极或北极的圆锥。第二个例子是三角形条带，经常用来定义曲面，因为大部分的曲面都有弯曲，这样可以避免表面点构成的矩形都形成平面。在这种情况下，三角形条带比四边形条带对创建曲面更加有利。

我们的三角扇形的例子定义了一个圆锥，它的顶点在点(0.0,1.0,0.0)，底面在X-Z平面上，半径为0.5。这个圆锥的顶点朝向是y轴正方向，并将y轴作为轴。图3-2显示了这个曲面，并给出了第6章中介绍的光照和Flat着色处理的效果，尽管下面的代码并没有反映这部分内容，当

使用圆锥的时候，可以通过模型变换很方便地按照自己的需要定义它的大小、朝向和位置。

```
#define numStrips 20
glBegin(GL_TRIANGLE_FAN);
    glVertex3f(0., 1.0, 0.);     // 圆锥顶点
    for (i=0; i < numStrips; i++) {
        angle = 2. * (float)i * PI / (float)numStrips;
        glVertex3f(0.5*cos(angle), 0.0, 0.5*sin(angle));
        // 下面是计算法线的代码
    }
glEnd();
```

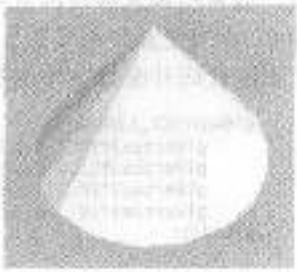


图3-2 使用三角扇形生成的圆锥

如图3-3所示的三角形条带的例子是一个定义在参数方程网格上的曲面，其中参数为：

$$y=\frac{x^2+2z^2}{e^{(x^2+z^2+1)}}$$

130

函数的定义域在X-Z平面上，值是每个顶点的Y坐标值。在X-Z平面上的网格点(x, z)通过简单的函数XX(i)和ZZ(j)给出，函数的值保留在一个数组中，vertices[i][j]给出在网格点(XX(i), ZZ(j))的函数值。下面的代码是从本书的在线资源中的完整代码中摘出的。XSIZE和YSIZE的典型值在100和250之间。



图3-3 用三角形条带创建的表面，其中一条单独的三角条带用青色突出出来

```
for (i=0; i<SIZE; i++)
    for (j=0; j<ZSIZE; j++) {
    x = XX(i);
    z = ZZ(j);
    vertices[i][j] = (x^x+2.0*z*z)/exp(x^x+2.0*z*z-t);
}
```

表面渲染可以按照嵌套循环的方式进行，其中每次循环迭代画一个三角条带，这个三角形条带在X方向的一个单位内沿Z方向伸展。这段代码如下所示，图3-3给出了绘制结果。这里省略了计算法线的代码，这个例子以及法线的计算将在关于着色处理的章节中详细讨论。这一类曲面会在第9章中进一步详细介绍。

```
for (i=0; i<XSIZE-1; i++)
    for (j=0; j<ZSIZE-1; j++) {
        glBegin(GL_TRIANGLE_STRIP);
            glVertex3f(XX(i),vertices[i] [j],ZZ(j));
            glVertex3f(XX(i+1),vertices[i+1] [j],ZZ(j));
            glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
            glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
        glEnd();
    }
```

131

该例子显示了一个由三个不同颜色的光源所照亮的白色表面，该技术将在第6章中描述。该表面的例子也在下面讨论的四边形中再次提到。请注意，点的序列和在接下来的四边形例子中会有一些细微的差别，这是由于四边形指定方式造成的。在该例中，有两个三角形来替换一个简单的四边形。同时，如果用户用四边形条带重新加工该例，而不用简单的多边形来显示数学曲面，那么，在此提到的改变以及用扩展的三角形条带来建立曲面会变得更为简单。

3.1.7 四边形

为了生成一组或者更多不同的四边形，用户可以用glBegin/glEnd函数的GL_QUADS模式，如上所述，每一个四边形有四个顶点。一个基于四边形的对象是上面的函数曲面。对于四边形，有类似下面的曲面代码：

```
for (i=0; i<XSIZE-1; i++)
    for (j=0; j<ZSIZE-1; j++) {
        // 四边形序列，点 (i,j) ,(i+1,j),(i+1,j+1),(i,j+1)
        glBegin(GL_QUADS);
            glVertex3f(XX(i),vertices[i][j],ZZ(j));
            glVertex3f(XX(i+1),vertices[i+1][j],ZZ(j));
            glVertex3f(XX(i+1),vertices[i+1][j+1],ZZ(j+1));
            glVertex3f(XX(i),vertices[i][j+1],ZZ(j+1));
        glEnd();
    }
```

3.1.8 四边形条带

为了创建四边形条带，glBegin/glEnd的绘制模式应该设为GL_QUAD_STRIP，正如我们之前讨论的，顶点的顺序与GL_QUADS模式不同，因为四边形条带的执行与三角形条带相同。所以，定义几何体时要注意，否则可能会产生不正常的显示。

在创建空心杆的应用中，可以使用四边形条带创建一个截面为正方形的又长又窄的管子，如图3-4。这里定义的四边形条带创建的管子是沿Z轴伸展，Z轴恰好穿过其截面中心。给定的尺寸使得这个管子是一个单位管——每一维的长度都是一个单位，它实际上是一个带两个不同末端开口的立方体。这些维度使得它很容易通过缩放来实现其他的应用。

```
#define RAD 0.5
#define LEN 1.0
glBegin(GL_QUAD_STRIP);
    glVertex3f( RAD, RAD, LEN ); // 第一个面的起点
    glVertex3f( RAD, RAD, 0.0 );
    glVertex3f(-RAD, RAD, LEN );
    glVertex3f(-RAD, RAD, 0.0 );
    glVertex3f(-RAD,-RAD, LEN ); // 第二个面的起点
    glVertex3f(-RAD,-RAD, 0.0 );
    glVertex3f( RAD,-RAD, LEN ); // 第三个面的起点
    glVertex3f( RAD,-RAD, 0.0 );
    glVertex3f( RAD, RAD, LEN ); // 第四个面的起点
    glVertex3f( RAD, RAD, 0.0 );
glEnd();
```

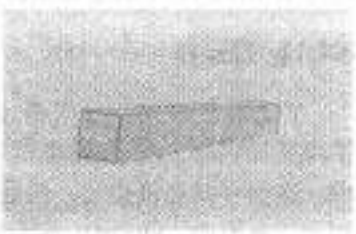


图3-4 用四边形条带生成的管子 132

3.1.9 普通多边形

在glBegin/glEnd中设置GL_POLYGON绘制模式可以让用户显示一个凸多边形。顶点列表中的顶点按照顺序被认为是多边形的顶点（从上往下看为逆时针，并且必须是凸多边形）。使用该操作不能显示多个多边形，因为这个函数假定它接受的所有顶点都属于同一个多边形。

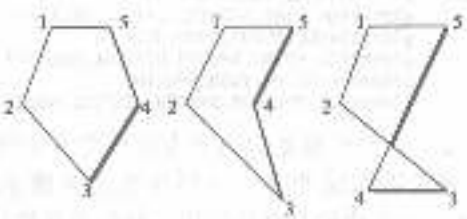


图3-5 凸多边形（左）和非凸多边形（中和右）

如果用户给出的点来自一个非凸多边形会怎样呢？（凸多边形和非凸多边形如图3-5所示，在前一章中我们也看到过这样的图形。）正如前面看到的，凸多边形可以用三角扇形表示，所以OpenGL试图用三角扇形来画凸多边形。这会导致如果要画的不是凸多边形那么将会画出非常奇怪的图形。如果一定要画非凸多边形，那么用户需要重组几何体，使得多边形由一组凸多边形组成。 135

最简单的凸多边形就是规则N边形——所有的边长相同，内角也相同。它的顶点可以很方便地通过三角函数来确定（以$N=7$为例）。

```
#define PI 3.14159
#define N 7
glBegin(GL_POLYGON);
    for (i=0; i<=N; i++)
        glVertex3f(2.0*sin((float)(i*360)/(float)N),
```

```
            2.0*cos((float)(i*360)/(float)N),0.0);
glEnd();
```

该多边形位于X-Y平面内，因为所有的Z值都是零。该多边形的颜色也是默认的（白色），因为没有为它指定颜色。这是一个“规范”对象的例子——该对象通常并不直接拿来使用，而是作为一个模板，能够通过变换来创建其他对象。规则多边形的一个有趣应用是创建规则多面体——所有面都是规则N边形的封闭实体。这些多面体的创建可以采用下面的方法：首先写一个简单的N边形函数，然后通过变换将这些多边形放在三维空间中的适当位置，组成多面体。

3.1.10 顶点数组

上面定义几何体的方法有点低效，因为它需要大量的函数调用，每个顶点都需要图形硬件的单独处理。单独的顶点处理除了来自glVertex*()函数，还来自其他函数比如glNormal*()和glTexCoord*()，它们同一些附加信息有关，比如法向量和纹理坐标。OpenGL允许使用数组，称作顶点数组，可以保存顶点、法线、纹理或其他信息。这些可以让用户指定函数调用的次数，因为它仅需要一个函数来处理整个数组。现代的图形加速器有一个专门的体系结构用来处理数组，很有价值。必须使用glEnable()函数来启用顶点数组，使用的方法也与单独顶点函数不同。因为使用顶点数组的主要原因是增加程序的效率，所以，这些内容将在第12章中讨论。

3.1.11 反走样

正如我们在前一章所看到的，使用反走样绘制的几何体比使用要么全有要么全无的像素模式更加平滑。OpenGL提供一些反走样的功能，让用户能够启用点平滑、线平滑和多边形平滑，这些可以直接指定，但是它们需要和颜色混合一起使用，而且和绘制几何体的顺序有关，如上一章所述，OpenGL基于几何体覆盖像素的比例计算覆盖因子，根据比例来混合边。颜色混合和绘制顺序将在第5章中介绍。

134 在使用OpenGL的反走样功能时，要利用glEnable()函数来选择不同的点、线或多边形平滑方式。每一个OpenGL的实现都为平滑定义一个默认的行为，这样就可能需要使用glHint(...)函数来定义自己的选择，从而覆盖默认的行为。适当的enable/hint对如下：

```
glEnable(GL_LINE_SMOOTH);
glHint(GL_LINE_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POINT_SMOOTH);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);
glEnable(GL_POLYGON_SMOOTH);
glHint(GL_POLYGON_SMOOTH_HINT,GL_NICEST);
```

一个更加复杂的多边形平滑方法涉及整个图像的反走样，通过加入轻微的偏移将场景绘制到累积缓冲区，这样每次边界像素的选择是不同的。这是一个很耗时的过程，并且是OpenGL比较高级的应用，我们将在第11章中讨论累积缓冲区和运动模糊。

3.1.12 将在很多例子中使用的立方体

因为立方体可以由六个四边形组成，所以尝试用一个四边形条带来生成立方体是很诱人的，然而，仅用一个四边形条带来生成立方体是不可能的，用户最多只能用四边形条带为立方体的表面生成四个四边形。用户可以创建两个四边形条带来组成立方体（想像棒球是如何缝在一起的），但是在这里我们使用由立方体的八个顶点指定的六个四边形。下面我们重复声明前一章中所介绍过的立方体的顶点、法线、边和面：

```
typedef float point3[3];
typedef int  edge[2];
```

```
typedef int  face[4];   // 立方体的每个面都有四条边
point3 vertices[8]   = {{-1.0, -1.0, -1.0},
                        {-1.0, -1.0,  1.0},
                        {-1.0,  1.0, -1.0},
                        {-1.0,  1.0,  1.0},
                        { 1.0, -1.0, -1.0},
                        { 1.0, -1.0,  1.0},
                        { 1.0,  1.0, -1.0},
                        { 1.0,  1.0,  1.0} };
point3 normals[6]    = {{ 0.0, 0.0, 1.0},
                        {-1.0, 0.0, 0.0},
                        { 0.0, 0.0,-1.0},
                        { 1.0, 0.0, 0.0},
                        { 0.0,-1.0, 0.0},
                        { 0.0, 1.0, 0.0} };
 edge edges[24]      =  {{ 0, 1 }, { 1, 3 }, { 3, 2 }, { 2, 0 },
                        { 0, 4 }, { 1, 5 }, { 3, 7 }, { 2, 6 },
                        { 4, 5 }, { 5, 7 }, { 7, 6 }, { 6, 4 },
                        { 1, 0 }, { 3, 1 }, { 2, 3 }, { 0, 2 },
                        { 4, 0 }, { 5, 1 }, { 7, 3 }, { 6, 2 },
                        { 5, 4 }, { 7, 5 }, { 6, 7 }, { 4, 6 }};
face cube[6]         = {{  0,  1,  2,  3 }, {  5,  9, 18, 13 },
                        { 14,  6, 10, 19 }, {  7, 11, 16, 15 },
                        {  4,  8, 17, 12 }, { 22, 21, 20, 23 }};
```

135

画立方体的过程是通过遍历面表和为立方体确定实际的顶点来进行的。我们扩展了早先给出的OpenGL代码。每一个面通过glBegin/glEnd函数对中的一次循环来定义。我们为每一个面都指定了法向量。由于GL_QUADS画图模式将每四个顶点作为四边形的顶点，所以不需要通过两次指定第一个点来让四边形封闭。

```
void cube(void) {
    int face, edge;
    glBegin(GL_QUADS);
        for (face = 0; face<6; face++) {
            glNormal3fv(normals[face]);
            for (edge = 0; edge<4; edge++)
                glVertex3fv(vertices[edges[cube[face][edge]][0]]);
        }
    glEnd();
}
```

该立方体如图3-6所示，展示了将每个不同颜色的面按顺序分别加到场景中的六个步骤，分别是红－绿－蓝－青－洋红－黄，这样可以看出绘制的过程。这是一个相当简洁的定义立方体的方法，只需要很少的代码。然而，还有其他定义立方体的方法。因为立方体是有六个四边形的规则多面体，因此可以定义一个标准的正方形，然后使用坐标变换由这个正方形来创建立方体的各个表面。这种方法的实现留给学生作为课后练习。

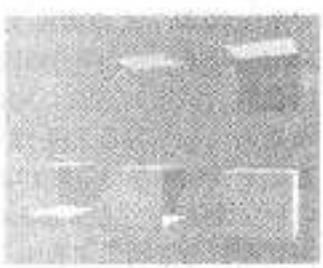


图3-6 立方体由各个四边形生成的过程

136

3.1.13 定义裁剪平面

除了OpenGL在投影时执行的裁剪，OpenGL允许用户自己定义至少六个裁剪平面来实现上一章讨论的裁剪问题，分别用GL_CLIP_PLANE0到GL_CLIP_PLANE5来命名。裁剪平面通过函数glClipPlane(plane, equation)定义，其中plane是预定义的裁剪平面，equation是一个由四个GLfloat类型的数组成的向量。一旦裁剪平面被定义，就可通过glEnable(GL_CLIP_PLANEn)或glDisable(GL_CLIP_PLANEn)来启用和禁用。当裁剪平面开启时会对任何建模基元进行操作。当它被禁用时，裁剪平面是不会被执行的。裁剪平面可以在需要的时候启用或禁用，并在场景中生效。下面将给出一段定义一个裁剪平面（通过指定它的方程的系数，必须用数组的方式）的示例代码，要显示的被裁剪的几何体要写在启用和禁用函数之间。

```
GLfloat myClipPlane[] = { 1.0, 1.0, 0.0, -1.0 };
glClipPlane(GL_CLIP_PLANE0, myClipPlane);
glEnable(GL_CLIP_PLANE0);
...
glDisable(GL_CLIP_PLANE0);
```

3.2 OpenGL工具中的附加对象

使用多边形建模使用户能够写出很多标准的合理的图形系统都应该包含的图形元素。OpenGL包括OpenGL实用库GLU，它有很多有用的函数，同时大多数的OpenGL版本都包括OpenGL实用工具GLUT。GLUT包括系统特有的函数，比如窗口管理函数。这些函数是跨平台的标准方式。GLU和GLUT包括很多已经生成好的图形元素可供调用。这些工具包提供的对象用很多的参数进行定义，比如对象在每一维上的旋转。很多具体细节是针对特定对象的，在使用时需要详细描述。

3.2.1 GLU二次曲面对象

GLU提供很多二次曲面对象，或者用二次曲面方程（含有三个变量并且不超过2次的多项式方程）定义的对象。这包括球体（gluSphere），圆柱体（gluCylinder）和圆盘（gluDisk）。每一个GLU体素声明成一个GLU二次曲面，并通过下面的函数进行分配：

```
GLUquadric* gluNewQuadric(void)
```

每一个二次曲面对象都是一个绕Z轴旋转而成的表面，它是由绕Z轴的分段（称为切片）和沿Z轴的分段（称为栈）构成。切片和堆栈确定了精度，或者说是模型的平滑度。图3-7给出了一个典型的预生成的二次曲面对象的例子和一个GLUT线框球体。该球体用少量的切片和线迹模而成，这样用户可以看见该定义的基础。使用GLU和GLUT对象的完整例子将在本节的结尾处给出。

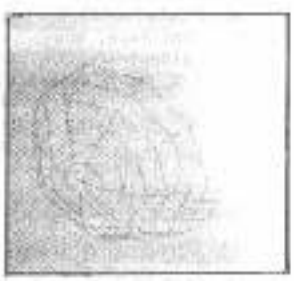


图3-7 有8个切片和12个栈的GLUT线框球体

GLU二次曲面非常有用，因为采用变换能够创建很多常见的
137 对象。GLU二次曲面也支持很多OpenGL渲染功能。比如可以用gluQuadricDrawStyle()函数设置绘图风格，这样对象可以是填充的、线框的、侧影的或由点构成的。也可以使用gluQuadricNormals()函数来为光照模型和平滑着色处理设置法向量。这样用户就可以选择是否使用法线，用Flat着色处理还是平滑着色处理。最后，用gluQuadricTexture()函数可以指定是否在二次曲面上应用纹理图来增加视觉效果。具体细节可以参看第6章和第8章。下面我们将通过列出函数原型来描述每一种GLU体素。更多的细节可以参看OpenGL手册中介绍GLU的部分。

3.2.2 GLU圆柱体

```
void gluCylinder(GLUquadric* quad, GLdouble base, GLdouble top, GLdouble
height, GLint slices, GLint stacks)
```

*quad*就是先前使用gluNewQuadric创建的二次曲面对象，*base*是圆柱体在$z = 0$处的半径，是圆柱的底，*top*是$z =$ height处的半径，*height*是圆柱体的高度。

3.2.3 GLU圆盘

GLU圆盘同其他GLU体素不同，因为它只有两维，完全在*X-Y*平面里。这样就不需要定义

栈了，第二个参数换成了loops，它定义的是圆盘同心圆的数量。

```
void gluDisk(GLUquadric* quad, GLdouble inner, GLdouble outer, GLint slices, GLint loops)
```

*quad*就是先前使用gluNewQuadric创建的二次曲面对象，*inner*是圆盘的内半径（可以为0），*outer*是圆盘的外半径。 138

3.2.4 GLU球体

```
void gluSphere(GLUquadric* quad, GLdouble radius, GLint slices, GLint stacks)
```

*quad*就是先前使用gluNewQuadric创建的二次曲面对象，*radius*是球体半径。

3.2.5 GLUT对象

GLUT提供的模型是几何实体。它们通常情况下用途并不广泛，因为它们的形状是固定的，不能用来进行通用的建模。除了茶壶模型以外，其他对象上不能进行纹理贴图。

GLUT模型包括：

- 圆锥（glutSolidCone/glutWireCone）。
- 立方体（glutSolidCube/glutWireCube）。
- 十二面体（glutSolidDodecahedron/glutWireDodecahedron），有十二个面的规则多面体。
- 二十面体（glutSolidIcosahedron/glutWireIcosahedron），有二十个面的规则多面体。
- 八面体（glutSolidOctahedron/glutWireOctahedron），有八个面的规则多面体。
- 球体（glutSolidSphere/glutWireSphere）。
- 茶壶（glutSolidTeapot/glutWireTeapot），犹他茶壶（计算机图形学的图标），有时候称为“茶壶多面体”。
- 四面体（glutSolidTetrahedron/glutWireTetrahedron）有四个面的规则多面体。
- 圆环（glutSolidTorus/glutWireTorus）。

上述这些模型都有标准的位置和朝向，典型的情况就是中心位于原点并且在标准体积内。如果是轴对称的物体，它是沿z轴排列。同GLU体素一样，GLUT的圆锥体、球体和圆环允许指定建模的粒度，但是其他的不行，因为它们有固定的几何形状。GLUT的这些函数中的solid并没有那么严格，它们实际上并不是实体，而是相邻的多边形组成的。“Solid”仅表示形体是被填充的，相比线框模式只是提供一个线框视图。如果切开“实体”，用户会发现它们实际上是空心的。

如果用户的OpenGL已经有GLUT，可以查看GLUT手册中关于这些实体以及其他重要的GLUT功能的详细信息。如果还没有，可以从OpenGL的站点（http://www.opengl.org/resources/ libraries/glut）下载不同系统下的GLUT代码，并安装它，就可以在用户的系统中使用它了。

图3-8显示了从GLU和GLUT中选出的对象集，从这幅图可以看出使用这些工具能够创建的几何项目。从左上开始顺时针移动，我们看到gluCylinder、gluDisk、glutSolidCone、glutSolidIcosahedron、glutSolidTorus和glutSolidTeapot。用户应该考虑怎样通过坐标变换用这些基本的对象创建其他形体。



图3-8 书中所述的一些GLU和GLUT对象

139

3.2.6 例子

这个例子是一个display()函数的简单应用，显示图3-8中所示的对象。这些对象使用了切

片和堆栈的粒度并把它们设置为常数，但是在示例代码中可以很方便地更改这些数值。代码舍弃了用在该例子中的图形上的颜色和着色处理规范，我们直到第6章才会看到它们。

```
// 全局变量
...
    GLUquadric *Q;
void myinit(void) {
    ...
}
void display(void)
{
    int i;
    ...
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
for (i = 0; i<6; i++) {
    glPushMatrix();
    glTranslatef(positions[i][0],positions[i][1],positions[i][2]);
    switch(i) {
    case 0: {Q=gluNewQuadric();
140  gluCylinder(Q, 2., 1., 1., 20, 1); break; }
case 1: {Q=gluNewQuadric();
         gluDisk(Q, 0.5, 1., 20, 10); break; };
case 2: {
         glutSolidCone(1., 1., 20, 10); break; };
case 3: {
         glPushMatrix();
         glScalef(2., 2., 2.);
         glutSolidIcosahedron();
         glPopMatrix();
         break; };
case 4: {glutSolidTorus(0.5,1.,20,20); break; };
case 5: {
         glPushMatrix();
         glRotatef(90.,1.,0.,0.);
         glutSolidTeapot(1.);
         glPopMatrix();
         break; };
}
glPopMatrix();
}
glutSwapBuffers();
}
```

3.3 OpenGL中的变换

在生成图像的时候，OpenGL只用到了两个活跃的变换：投影变换和模视变换。投影变换是由用户定义的投影生成的，模视变换则是从用户定义的视图变换以及所有在程序中应用到的模型变换中得到的。我们已经讨论了投影和视图，因此这里我们会关注在建模中用到的变换。

这里有三种基本的模型变换：旋转、平移以及缩放。在OpenGL中，这些变换相应地通过函数集合glRotate、glTranslate以及glScale实现。就像我们已经看到的其他OpenGL函数集合那样，这些函数也有着不同的版本，但其中的变化只在于他们采用的参数类型有所不同。

最常见的glRotate函数是

```
glRotatef(angle, x, y, z)
```

这里angle是旋转的角度，x、y和z指定了一个向量的坐标，所有这些参数都是浮点（f）类型的。旋转函数glRotated的操作方式与glRotatef完全相同，只是所有的参数都是双精度的浮点数（d）。参数中的向量定义了固定的旋转轴，旋转遵循右手法则，因此从向量（x, y, z)的方向上看过去，旋转将会是逆时针的。最简单的旋转是绕着三个坐标轴进行的，因此glRotatef(angle,1.0, 0.0, 0.0）会沿着X轴旋转模型空间。该函数可以用在投影或者模视变换中，这取决

于glMatrixMode函数中的值。当用户在投影模式中，允许他定义一个旋转的投影，或者当用户在模－视模式中，允许他在模型空间内旋转对象。用户可以使用glPushMatrix以及glPopMatrix两个函数来保存和恢复未经旋转的坐标系。 141

最常用的glTranslate函数是

```
glTranslatef(Tx,Ty,Tz)
```

其中*Tx*、*Ty*和*Tz*是浮点类型（f）的平移向量中的各个坐标。平移函数glTranslated的操作也是相同的，只是参数换成了双精度的浮点类型（d）。就像glRotate一样，该函数可以用在投影或者模视变换中。这取决于glMatrixMode的数值。因此，用户如果在投影的模式下可以定义平移的投影，而如果在模视的模式下则可以定义模型空间内的平移对象。用户可以使用glPushMatrix以及glPopMatrix函数来保存以及恢复未经过平移变换的坐标系。

最常用的glScale函数是

```
glScalef(Sx,Sy,Sz)
```

其中*Sx*、*Sy*和*Sz*是浮点类型（f）的缩放向量中的各个坐标。平移函数glScaled的操作是类似的，只是其参数是双精度的浮点类型（d）。该函数可以用在投影或者模视变换中，这取决于glMatrixMode的值。因此用户如果在投影的模式下可以定义缩放的投影，如果在模视的模式下可以定义模型空间内的缩放对象。用户可以接着使用glPushMatrix以及glPopMatrix函数来保存以及恢复未经过缩放变换的坐标系。由于缩放操作以非均匀的方式改变了几何体，因此缩放变换可能会改变一个对象的法线方向。如果在模视变换的模式下，模型缩放因子大于1.0，并且启用了光照，那么就要执行法线的自动归一化操作。请参考第6章来获得相关的细节信息。

OpenGL有一些工具可以和变换一起工作。3D计算机图形学中的变换通过一个4×4的阵列来表达，里面存储了16个实数。用户可以用以下函数来保存当前的模视矩阵：

```
glGetFloatv (GL_MODELVIEW_MATRIX,trans)
```

其中trans是数组阵列GLfloat trans[16]。用户不能直接恢复模视矩阵，然而，如果用户的变换模式通过glMatrixMode(GL_MODELVIEW)设置成了模视矩阵，用户就可以用函数glMultMatrix(myMatrix)将myMatrix与当前的模视矩阵相乘，然后将结果以16个元素的数组形式保存起来，用户可以清理该矩阵，并且用以下的保存过的矩阵来和它相乘：

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrix(myMatrix);
```

用户可以用类似的方法来操作OpenGL中的投影矩阵。操作的类型需要用户对变换的矩阵表达和操纵方式比较熟悉，然而OpenGL提供了足够多的变换工具，因此，用这种方式来处理变换是非常少的。

就像我们在上面所看到的那样，用户可以在图形场景中使用很多变换来定义一个对象。当针对全部的模型来思考完整的变换顺序的时候，我们不仅要考虑模型变换，也要考虑投影和视图变换。如果在代码中以这样的顺序使用变换T_0，T_1，…，T_{last}，那么我们定义对象的完整变换序列是： 142

$$P \rightarrow V \rightarrow T_0 \rightarrow T_1 \rightarrow \cdots \rightarrow T_n \rightarrow T_{n+1} \rightarrow \cdots \rightarrow T_{last}$$

上面*P*是投影变换，*V*是视图变换，而T_0，T_1，…，T_{last}都是在程序中指定的对场景进行建模的变换，并且T_0是第一个，T_{last}是最后一个且最靠近几何体的定义。该效果就是将这个复合序列

$$P(VT_0(T_1(\cdots(T_{last})\cdots)))(\text{vertex})$$

应用到顶点中来指定几何体。投影变换可能定义在reshape()函数中，而视图变换则可能定义

在init()、reshape()函数中，或者在display()函数的开头。在任何情况中，视图定义在建模过程的开始。用户需要非常好地理解该序列，因为对于用户理解如何建立复杂的、具有层次的模型而言，该序列是非常关键的。

3.4 图例和标签

在这一个短节中，我们来描述OpenGL是如何处理文本的。我们也已经展示过如何在一个单独的视口中处理图例，这也可能是最简单的处理图例的方式。该代码用于生成图2-25中的图像，虽然在例子中用到的一对函数没有在这里被包含进去。

在图例和标签中的文本是用于工函数生成的。该函数集成了一些工具来显示文本。函数doRasterString(…)显示GLUT 中的glutBitmapCharacter()函数定义的位图字符，显示的位置通过glRasterPos()函数来确定。在上述例子中，我们选择了一个24号的新罗马位图字体，而其他大小和风格的字体被GLUT的每一版支持，请检查系统来获得其他的选项。

```
void doRasterString(float x, float y, float z, char *s) {
    char c;

    glRasterPos3f(x,y,z);
    for (; (c = *s) != '\0'; s++)
        glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_24, c);
}
```

接下来的代码可以用来生成例子中的图例，这是非常直接的，如下文所示。其中设置了文本的颜色，同时也关闭了光照计算，这是为了能控制图例的展示效果。这些都在第6章中讨论。请注意，用C语言写的sprintf函数需要一个字符数组作为它的目标，而不是用字符指针。该代码包含了一个函数使用平滑着色处理来构建颜色的过渡，这在颜色一章中还会涉及到。它可以是display()回调函数的一部分，在那里将会重新绘制。

```
// 生成自身的视口内绘制图例
    glViewport((int)(5.*(float)winwide/7.),0,
               (int)(2.*(float)winwide/7.),winheight);
glClear(GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT);
...
// 设置视口的参数
    glPushMatrix();
    glEnable(GL_SMOOTH);
    glColor3f(1.,1.,1.);
    doRasterString(0.1, 4.8, 0., "Number Infected");
    sprintf(s,"%5.0f",MAXINFECT/MULTIPLIER);
    doRasterString(0.,4.4,0.,s);
// 颜色表示热量场梯度，其中0.3和0.89是最佳选取
    glBegin(GL_QUADS);

        glColor3f(0.,0.,0.);
        glVertex3f(0.7, 0.1, 0.);
        glVertex3f(1.7, 0.1, 0.);
        colorRamp(0.3, &r, &g, &b);
        glColor3f(r,g,b);
        glVertex3f(1.7, 1.36, 0.);
        glVertex3f(0.7, 1.36, 0.);
        glVertex3f(0.7, 1.36, 0.);
        glVertex3f(1.7, 1.36, 0.);
        colorRamp(0.89, &r, &g, &b);
        glColor3f(r,g,b);
        glVertex3f(1.7, 4.105, 0.);
        glVertex3f(0.7, 4.105, 0.);
        glVertex3f(0.7, 4.105, 0.);
        glVertex3f(1.7, 4.105, 0.);
        glColor3f(1.,1.,1.);
        glVertex3f(1.7, 4.6, 0.);
        glVertex3f(0.7, 4.6, 0.);
```

143

```
    glEnd();
    sprintf(s,"%5.0f",0.0);
    doRasterString(.1,.1,0.,s);
    glPopMatrix();
    glDisable(GL_SMOOTH);
// 现在回调主窗口要显示实际的模型
```

标签的实现与此非常类似，且更加容易一些，这是由于用户不需要包含图形。用户简单地生成文本，加入到图像中去，无论什么地方，只要是用户设计标志的地方都可以用，并且将该文本用简单的光栅定位和文本输出写到屏幕中。

3.5 变换的代码实例

3.5.1 简单变换

图3-9显示了三个简单的作用在一个给定正方形上的变换效果，该正方形最初放置于距离X轴外面几个单位的地方。该正方形的左边被旋转了，在中心处进行了缩放（请注意该缩放影响到了和原点之间的距离，同时也影响到了正方形的大小），而正方形的右边被平移了。在完全的例子代码中，上述这些内容都在相同的窗口中不同的视口中进行了显示。

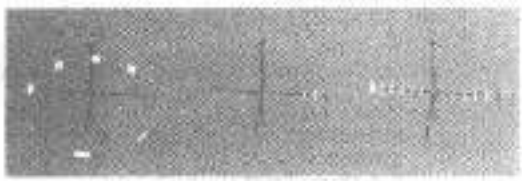


图3-9 将三个简单变换操作的正方形——旋转（左），缩放（中）和平移（右） 144

所有的代码例子用到了以下的简单正方形的定义。

```
void square(void)
{
   typedef GLfloat point [3];
   point v[8] = {{12.0, -1.0, -1.0},
                 {12.0,  1.0,  1.0},
                 {12.0,  1.0, -1.0} };

   glBegin(GL_QUADS);
     glVertex3fv(v[0]);
     glVertex3fv(v[1]);
     glVertex3fv(v[2]);
     glVertex3fv(v[3]);
   glEnd();
}
```

为了显示简单的旋转例子，用户可以使用以下的显示函数，该函数或者下面任意的其他display()函数，都可以放置在本书开始时介绍的通用函数模板中。

```
void display(void)
{ int i;
  float theta = 0.0;

  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  axes(10.0);
  for (i=0; i<8; i++) {
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
    glRotatef(theta, 0.0, 0.0, 1.0);
    if (i==0) glColor3f(1.0, 0.0, 0.0);
    square();

    theta += 45.0;
    glPopMatrix();
  }
  glutSwapBuffers();
}
```

145

为了实现简单的平移，用户可以用以下的显示函数：

```
void display(void)
{ int i;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  axes(10.0);
  for (i=0; i<=12; i++) {
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix();
    glTranslatef(-2.0*(float)i, 0.0, 0.0);
    if (i==0) glColor3f(1.0, 0.0, 0.0);
    square();
    glPopMatrix();
  }
  glutSwapBuffers();
}
```

为了实现简单的缩放，用户可以使用下面的显示函数：

```
void display(void)
{ int i;
  float s;
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  axes(10.0);
  for (i=0; i<6; i++) {
     glColor3f(1.0, 1.0, 1.0);
     glPushMatrix();
     s = (6.0-(float)i)/6.0;
     glScalef(s, s, s);
     if (i==0) glColor3f(1.0, 0.0, 0.0);
     square();
     glPopMatrix();
  }
  glutSwapBuffers();
}
```

3.5.2 变换栈

在OpenGL中操作变换栈的函数是glPushMatrix()和glPopMatrix()。从技术上来说，它们都应用在当前矩阵模式的变换栈中，该矩阵模式则是通过函数glMatrixMode来设置的。其参数可以是GL_PROJECTION或者GL_MODELVIEW。我们通常很少想到使用投影的栈（并且投影的栈中只保存两个变换），因此，我们几乎总是使用模视栈。在前一章中介绍过兔子头的例子（请读者参考图3-10），在那个例子里面就用到了下面的显示函数。该代码通过使用格式的缩排而使栈操作的可读性更强；而这正是打算强调的结果，在实践中值得推荐。我们已经对每个部分定义了比较简单的显示属性（也就是简单的颜色），然而，我们也可以使用更加复杂的属性集合，这样可以让各个部分看起来更加生动有趣。我们也可以用到更加复杂的对象来超越一个简单的gluSphere，这样各个部分从几何上看起来更加有趣。



图3-10 兔子头

```
void display(void)
{
//  缩排的层次显示变换栈的层次
//  该例子的每部是一个单位的gluSphere，
//  其他所有形物体通过显式的变换来完成
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
        // 对头部建模
        glColor3f(0.4, 0.4, 0.4);     //    将灰色的头部
        glScalef(3.0, 1.0, 1.0);
        myQuad = gluNewQuadric();
        gluSphere(myQuad, 1.0, 10, 10);
```

```
        glPopMatrix();
        glPushMatrix();
            // 对左眼建模
            glColor3f(0.0, 0.0, 0.0);    //    黑色的眼睛
            glTranslatef(1.0, -0.7, 0.7);
            glScalef(0.2, 0.2, 0.2);
            myQuad = gluNewQuadric();
            gluSphere(myQuad, 1.0, 10, 10);
        glPopMatrix();
        glPushMatrix();
            // 对右眼建模
            glTranslatef(1.0, 0.7, 0.7);
            glScalef(0.2, 0.2, 0.2);
            myQuad = gluNewQuadric();
            gluSphere(myQuad, 1.0, 10, 10);
        glPopMatrix();
        glPushMatrix();
            // 对左耳建模
            glColor3f(1.0, 0.6, 0.6);    //    粉红色的耳朵
            glTranslatef(-1.0, -1.0, 1.0);
            glRotatef(-45.0, 1.0, 0.0, 0.0);
            glScalef(0.5, 2.0, 0.5);
            myQuad = gluNewQuadric();
            gluSphere(myQuad, 1.0, 10, 10);
        glPopMatrix();
        glPushMatrix();
            // 对右耳建模
            glColor3f(1.0, 0.6, 0.6);    //    粉红色的耳朵
            glTranslatef(-1.0, 1.0, 1.0);
            glRotatef(45.0, 1.0, 0.0, 0.0);
            glScalef(0.5, 2.0, 0.5);
            myQuad = gluNewQuadric();
            gluSphere(myQuad, 1.0, 10, 10);
        glPopMatrix();
        glutSwapBuffers();
}
```

147

在OpenGL中，模视矩阵栈的深度至少是32，但是，这对于处理一些复杂的模型而言是不够的。OpenGL将变换存储成一个GLfloat类型的4×4矩阵，实际上的形式则是一个有16个元素的简单数组阵列。如果用户超过了那个深度，或者用户想要用不同的方式来操作栈，那么用户就需要生成自己的结构来保存这些数组。这样用户就可以用想要的方式来管理这些变换。为了处理模视变换，用户可以自己使用函数来保存并且设置它。用户也可以用函数来获得变换的当前数值：

```
glGetFloatv(GL_MODELVIEW_MATRIX,myTran);
```

(这里我们已经声明了GLfloat myTran[16])。如果用户正在模视的状态下，他就可以用函数

```
glLoadIdentity();
glMultMatrixf(myTran);
```

148

将矩阵myTran的数值设置成当前的模视矩阵。

3.5.3 逆转视点变换

在前一章中，我们谈论了设置一个视图并且证明我们可以通过使用标准的视图来得到相同的效果，以及得到这个变换和在模型空间内观察应用逆转变换两者之间的关系。它的实现在OpenGL中是很直接的，我们写了一个小程序，在程序中视点不是固定在一个位置上，而是跟随场景中的一个运动对象。在该例中，当球体飞行的时候，视点跟随着一个距离它4个单位距离的红色球体在一些几何体上方绕着圆周飞行。几何体是一个青色的平面，在它的上面以距离中心点等距的形式放置了一些圆柱体，同时还有一些坐标轴。其中Y轴是向上的，并且青

色的平面是X-Z方向的。从该简单模型中得到的一张快照如图3-11所示。同时我们也提供了显示函数的代码。

在这个模型中，球体是通过：

绕着Y轴旋转*theta*角度

沿着X轴平移5单位，沿着Y轴平移0.75单位

来放置的，同时视点是通过相对于球体：

在Z方向平移4单位

来放置的。

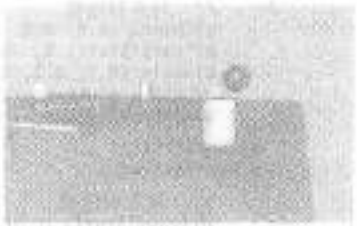


图3-11 视点跟随着一个在平面上越过一些圆柱体飞行的球体

当球体在X轴上，举例来说，视点在Z方向上－4单位的地方。因此，显示函数的开始是默认的视图，接下去则是以反向顺序表达的这些变换的逆变换，也就是

在Z方向上平移－4单位

在X方向上平移－5单位，在Y方向上平移－0.75单位

149 绕着Y轴旋转-*theta*角度

这些都是圆柱体放置以及视点放置的逆变换，并且应用到整个世界空间中以获得视图变换的效果。正是由于我们用反转视点位置来观察的方法，用户在代码中就看不到显式的视图规范。但是会在运行程序的时候看到正确的视图。

```
void display(void)
{
    ...
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();

//  定义视点的位置，视点相对于它所跟随的球体
//  gluLookAt(0.,0.,0.,0.,-1.,0.,1.,0.);这
//  是默认的并不需要的逆转变换来放置视点
    glTranslatef(0.,0.,-4.);
    glTranslatef(-5.,-0.75,0.);
    glRotatef(-theta,0.,1.,0.);

//  绘制我们跟随的球体
    glPushMatrix();   // 保存当前的模型变换
    glRotatef(theta, 0., 1., 0.);
    glTranslatef(5., 0.75, 0.);
    glColor3f(1., 0., 0.);
    myQuad = gluNewQuadric();
    gluSphere(myQuad, .25, 20, 20);
    glPopMatrix();
// 绘制所有球体飞越的几何体
    ...
    glutSwapBuffers();
}
```

3.5.4 生成显示列表

在上一章中，我们讨论了编译几何体的想法，这样可以更加高效地执行显示操作。在OpenGL中，图形对象可以编译进入显示列表中，显示列表包含了对象最后的几何体，它正等待显示。显示列表捕获了用户建模图中的一个分支，该图中包含了用户想要的显示内容，下面我们给出用户可以在显示列表里面包含什么内容的原则。

显示列表在OpenGL中的生成是相对比较容易的。首先选择一个无符号整数（通常是一个小的整数常量，比如1，2，…）来代表用户列表的名字。在用户定义列表的几何体之前，调

用函数glNewList(i)。将所有用户需要的几何体写入列表中，包括几何体、变换以及外观，然后在结束的时候，调用函数glEndList()就可以了。所有在新建列表和结束列表的函数之间的 150
内容将会在用户调用glCallList(i)的时候执行，其中将一个合法的列表名字作为参数，同时只有OpenGL系统中的绘制部分的实际指令集合才会被保存起来。当显示列表执行的时候，那些指令只是简单地发送到绘制系统中去，任何需要产生这些指令的操作都不会被包含，因为它们的工作已经被捕获到显示列表中去了。

显示列表只要定义一次就可以多次使用，因此，用户在函数中不能生成经常要用的列表，比如display()。通常都在init()函数中生成它们（请见下文），或者在从init()中调用的函数。下面给出了一些代码的例子，其中大多数的内容省略了，只保留了显示列表的操作。

```
GLint displayListIndex 1;
void Build_lists(void) {
   glNewList(displayListIndex, GL_COMPILE);
     glBegin(GL_TRIANGLE_STRIP);
       glNormal3fv(...); glVertex3fv(...);
       ...
     glEnd();
   glEndList();
}
static void Init(void) {
   ...
   Build_lists();
   ...
}
void Display(void) {
   ...
   glCallList(displayListIndex);
   ...
}
```

上面的显示列表是用GL_COMPILE模式生成的，该模式让函数直到列表被调用的时候才执行（对象不被显示）。如果用户想让列表在生成的时候就显示出来，只需要将该模式设置成GL_COMPILE_AND_EXCUTE就可以了。

OpenGL中的显示列表是用非零的无符号整数来命名的（技术上来说是GLint类型的数值），同时OpenGL中有很多工具来管理名字数值。在这里假定用户不需要很多的显示列表，用户可以自己管理为数不多的列表名字，但是，如果用户想在工程中用到大量的显示列表，请参考函数glGenLists, glIsList以及glDeleteLists来帮助管理列表。

3.6 到视点的距离

和我们在第1章中看到的以及后面在第5章中将会看到的那样，当我们从眼睛的位置出发对物体进行排序的时候，知道视点和场景中的对象之间的距离是非常重要的，如果用户使用深度排序来绘制场景，这一点就变得尤为重要，我们会在第12章中讨论。OpenGL中有函数可以告诉用户这个距离。如函数及其参数

```
glGetFloatv(GL_CURRENT_RASTER_DISTANCE)
```

151

返回视点和当前光栅位置之间的距离。请参考第12章中的相关内容来获得更详细的信息。

3.7 小结

在这一章内，我们介绍了OpenGL是如何让用户定义在前一章中展示的建模中的几何体以及变换的。我们讨论了OpenGL如何实现点、线段、三角形、四边形以及多边形这些基础形体，还有直线条带、三角形条带、三角扇形以及四方条带几何体的压缩技术。我们也看到了很多通过GLU和GLUT可以获得的几何对象，以及它们在生成场景中使用的方法。这一章主要关注几何体，然而我们也将会在第6章和第8

章中看到GLU和GLUT的基元能够很容易和OpenGL的表现函数工具一起工作。

我们也介绍了OpenGL如何给予用户缩放，平移以及旋转变换，以及OpenGL如何管理模视变换的栈，这样可以让用户容易地实现场景图。最后，OpenGL让用户可以将几何体编译进显示列表中，以获得比立即模式更高的效率。

在这里，用户应该对基于多边形的图形绘制流水线的每一步骤都有很好的理解，因此应该能够写出完整的图形程序。本书的后面包对样条的曲面插值进行了介绍。接下来的章节的关注焦点将会是图形对象中的外观。

3.8 本章的OpenGL术语表

这一章中基本上用OpenGL的函数来进行建模，因此，本章的术语表是非常丰富的。由于一些函数有扩展的参数列表，因此，我们将省略这些参数，以备查询。我们只列出在本章中新出现的那些函数，这些函数需要在本章中实际出现过而不是仅被提到过，同时我们要提醒读者，在前几章中的OpenGL函数不会在这里重复罗列。

OpenGL函数

glCallList(int):用参数指定要编译显示列表执行的索引

glClipPlane(int, vector):定义一个裁剪平面，它的数字序号是第一个参数，它的平面方程是通过第二个参数中的4维向量给定的

glEndList():结束显示列表的生成

glGetFloatv(parm, array):得到由第一个参数指定的系统矩阵的数值，并且将该数值存储到第二个参数指定的数组中

glHint(parm,parm):根据第二个参数的数值，告知系统由第一个参数指定的进程已经执行完毕

glLineWidth(float):用参数的数值，以像素为单位指定直线的宽度

glMultMatrix(array):用数组与当前的（投影或者模视）矩阵相乘

glNewList(int, parm):开始向显示列表保存命令，显示列表的索引由第一个参数指定，可以只是保存或者可以既保存又执行，这取决于第二个参数

152

glNormal*(...):对OpenGL的几何体指定一个顶点的法线方向，该法线可以是2维的或者3维的，可以是浮点类型的或者是整形的，可以用标量或者向量的形式给出

glPointSize(float): 用参数的数值，以像素为单位指定点的大小

glRasterPos3f(x,y,z):根据x，y和z，设置系统当前的光栅位置，这是设置光栅位置的函数族中的一个

glVertex*(...):对OpenGL的几何体指定顶点的坐标，顶点可以是2维的或者是3维的，坐标可以是浮点型的或者整数型的，坐标可以作为标量也可以作为向量给出

GLU函数

gluCylinder(...):第一个参数是指定一个存在的GLU二次曲面对象，其余的参数指定圆柱体的细节信息，定义圆柱的几何体同时进行绘制

gluDisk(...):第一个参数是指定一个存在的GLU二次曲面对象，其余的参数指定圆盘的细节信息，定义圆盘的几何体同时进行绘制

gluNewQuadric():生成并且返回一个指向新的GLU二次曲面对象的指针

gluQuadricDrawStyle():带一个符号参数，指定GLU二次曲面对象的绘制风格

gluQuadricNormals():当定义一个GLU二次曲面对象的时候，指定是否生成法线以及生成何种法线

gluQuadricTexture(): 当定义一个GLU二次曲面对象的时候，指定是否生成纹理坐标

gluSphere():第一个参数是指定一个存在的GLU二次曲面对象，其余的参数指定球体的细节信息，定义球体的几何体同时进行绘制

GLUT函数

对于这里包含的很大多数GLUT对象而言，都可以提供实体的以及线框模式的版本。我们将会列出实体版（包含单词“Solid”），然而用户可以通过将单词“Solid”替换成单词“Wire”来得到线框版本的相应函数。

glutBitmapCharacter(font, char):用已经命名的位图字体绘制一个位图化的字符

glutSolidCone(...):参数指定了一个圆锥体的细节，定义了圆锥体的几何体并且进行绘制

glutSolidDodecahedron():定义了一个12面体并且进行绘制

glutSolidIcosahedron():定义了一个20面体并且进行绘制

glutSolidOctahedron():定义了一个8面体并且进行绘制

glutSolidSphere(...):参数指定了一个球体的细节，定义了球体并且进行绘制

glutSolidTeapot(size):定义了一个给定大小的茶壶并且进行绘制，与此同时也产生了茶壶的法线和纹理坐标 153

glutSolidTetrahedron():定义了一个4面体并且进行绘制

glutSolidTorus(...):参数指定了一个圆环面的细节信息，定义了圆环面并且进行绘制

符号参数

GL_CLIP_PLANEi:是glEnable()的参数，指定一个特定的裁剪平面

GL_COMPILE:是glNewList()函数的第二个参数，指定函数生成列表但是不执行

GL_COMPILE_AND_EXECUTE: 是glNewList()函数的第二个参数，指定函数生成列表并且当它们进入列表的时候给予执行

GL_DONT_CARE:是glHint()函数的第二个参数，指定系统可以用默认的技术

GL_FASTEST: 是glHint()函数的第二个参数，指定使用最快的技术（通常意味着反走样不被处理）

GL_LINE_LOOP:是glBegin()函数的参数，指定将下面的顶点定义处理成直线循环的顶点

GL_LINES: 是glBegin()函数的参数，指定将下面的顶点定义处理成单独的直线顶点

GL_LINE_SMOOTH: 是glEnable()函数的参数，指定要使用直线的平滑处理

GL_LINE_SMOOTH_HINT: 作为glHint()函数的第一个参数，指定接下来的提示将用作直线的平滑处理

GL_LINE_STRIP: 是glBegin()函数的参数，指定将下面的顶点定义处理成直线条带的顶点

GL_NICEST: 是glHint()函数的第二个参数，指定应用产生最高质量结果的技术（通常意味着将进行反走样处理）

GL_POINT_SMOOTH: 是glEnable()函数的参数，指定使用点的平滑处理

GL_POINT_SMOOTH_HINT: 作为glHint()函数的第一个参数，指定接下来的提示将用作点的平滑处理

GL_POINTS: 是glBegin()函数的参数，指定将下面的顶点定义处理成单独点的顶点

GL_POLYGON: 是glBegin()函数的参数，指定将下面的顶点定义处理成（凸）多边形的顶点

GL_POLYGON_SMOOTH: 是glEnable()函数的参数，指定要使用多边形的平滑处理

GL_POLYGON_SMOOTH_HINT: 作为glHint()函数的第一个参数，指定接下来的提示将用作多边

形的平滑处理

154 GL_QUADS: 是glBegin()函数的参数，指定将下面的顶点定义处理成单独的四边形的顶点

GL_QUAD_STRIP: 是glBegin()函数的参数，指定将下面的顶点定义处理成单独的四边形条带的顶点

GL_TRIANGLE_FAN: 是glBegin()函数的参数，指定将下面的顶点定义处理成三角扇形的顶点

GL_TRIANGLES: 是glBegin()函数的参数，指定将下面的顶点定义处理成三角形的顶点

GL_TRIANGLE_STRIP: 是glBegin()函数的参数，指定将下面的顶点定义处理成三角形条带的顶点

3.9 思考题

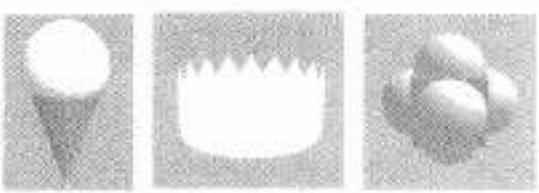
1. OpenGL的基元中包含四方形也包含三角形，那么，为什么包含四方形的基元是必要的呢？是否存在能用四方形绘制但却不能用三角形绘制的东西？
2. GLU对象的使用是现成的，然而它们可以非常简单地从OpenGL的四方形或者三角形来生成。请描述对于尽量多的对象，你如何来做？至少包含gluSphere, gluCylinder, gluDisk, glutSolidCone以及glutCube。
3. GLU的对象和一些GLUT的对象都使用了参数，称为切片，栈或者循环，它们定义对象的颗粒度。对于用这些参数定义的对象而言，你可以想出其他的方法来定义对象的颗粒度吗？对这些参数使用小的数值有什么优势？对于这些参数，如果使用大的数值又有什么优势吗？是否存在一些GLU或者GLUT的对象没有用到切片和栈？这又是为什么？
4. OpenGL中的缩放以及平移变换参数的数值是在模型空间中的、世界空间中的、还是在3D视点空间中的？请对你的回答给出理由。
5. OpenGL中旋转变换的角度是通过绕着一条固定的直线，以度为单位给出的。那么绕其旋转的那条直线是定义在什么空间的？在很多应用场合中，角度是以弧度来表达的，而不是度。那么，如何把弧度转变为度？
6. 在第0章中的热量传播例子中，请打印出源代码并且将它用到的OpenGL函数用高亮显示。对于每一个OpenGL函数，请解释它们都作了些什么工作并且说出它们为什么要出现在代码中的那个位置上。
7. 请思考在热量传播例子中display()函数里面的模型，比较生成模型以及显示模型所需要的操作数量，包含所有的变换，以及在显示列表中使用的glVertex(...)函数调用的数目。请据此得出显示列表和简单建模的相对效率的关系结论。
8. 事实上，对于在热量传播例子中的display()函数中的整个模型而言，不能使用显示列表，因为例子中的模型在每一个idle()回调的时候都会发生改变。为什么对于热量传播问题中尝试使用显示列表是没有意义的？
9. 在前一章中关于酒会场景的例子中，请将视点放置在酒会场景中的一个物体之上，并且从前面的练习中改变场景图来包含该视点的放置。接下来请写出场景图来旋转该视点并且将它放置于一个标准的位置上。
10. 我们已经说过模视矩阵栈至少有32层的深度，并且在深入场景图中获得场景对象以及遇到变换组的时候，就需要完成栈压入操作。如果由于建模图的性质让用户超过了模视矩阵栈深度的时候，请描
155 述上面两个事实之间的关系。

3.10 练习题

1. 在前一章中作为例子用到的3D箭头中，我们使用了通用的建模，而不是专用的函数。请使用GLU和GLUT建模工具来实现一个3D箭头，把它作为一个工具函数，在任何的场景中都可以使用它。
2. 在前面我们说过可以用两个四方条带形成一个立方体，以替换6个独立的四边形制作立方体的方法。

请使用我们在那个讨论中对于顶点以及边的声明，重新用两个四方条带来写出立方体。比较在两个立方体的实现中用glVertex*f...]函数设置的顶点数量，并且给出两者在数量上的比。可以用该数值作为几何压缩中的测量尺度吗？为什么？提示：所有关于生成图形对象的，因此练习中接下来的序列包含了制作一些用户可以用的通用图形对象。

3. 定义一个“单位卡通哑铃”，它是一个在x轴上的单圆柱体，其两个端点分别在1.0和−1.0，两个球形的端点是中等尺寸的，每一个都位于圆柱体底面的中心。称其为卡通哑铃是因为早期儿童的卡通涉及举重者的时候，经常会使用该形状。
4. 我们用不同尺寸的实体盘片硅码来制作更具真实感的砝码集合，用标准的砝码（5kg，10kg，20kg等）来定义标准盘片的集合，其中盘片的重量作为参数来决定砝码的厚度和/或半径，假定砝码的重量正比于其体积。请定义一个函数，用标准重量的组合生成一个给定重量的哑铃，将每个盘片都放在支撑观的合适位置上（请注意我们并没有要求生成一个在中间有一个孔的真实盘片）。
5. 让我们用一个圆柱形的孔来生成一个圆柱体的对象，其中圆柱形孔和圆柱体对象的中心轴是重合的。请定义一个单位长度的对象，其中它的内部以及外部的圆柱体都有给定的半径，由盘片将管子的两端套起来，请证明能够用该对象来生成在前一个练习中的那种更具真实感的哑铃。
6. 迄今为止，我们定义的所有基于圆柱的对象都有标准的方向，但是我们需要由任意开始点和结束点的圆柱体，这样就可以给圆柱体任意的方向。请思考一个圆柱体，它的一端在原点，另一端在点$P=(x, y, z)$上，P点和原点的距离是一个单位。请写出一个函数，它能将一个单位圆柱体进行旋转，其中一端在原点上，而另一端则在P点（提示：需要使用两个旋转，应该考虑反正切函数）。
7. 在已经解决上一个练习中圆柱体的方向问题的基础上，请写出一个函数来生成基于条带以连接点p_0，p_1，p_2，…，p_n，其中管子的半径是r，在端点上有相同大小的球，这样可以在一个管子到另一个管子之间用平滑的过渡来连接各个管子。请使用这个物体来生成一个柔软的绳子，作为练习题13中的卡通哑铃之间的连接绳。请证明该结构在两者之间有一个弯曲的条棒可以在中间支撑该结构。
8. 通过定义大量的顶点并且将它们用直线条带连接起来，生成在X-Y平面内的曲线。接着通过绕着Y轴旋转该曲线来生成普通的旋转曲面。请在曲面模型上，制作一些读者知道的对象，比如一个葡萄酒杯或者一个棒球帽，以及诸如此类轴对称的物体等。是否能够使用变换将它推广成绕着任何直线旋转一条曲线？如何做？ 156
9. 在前一章中的一个练习题让我们设计一个建筑物的位置图，里面包含了标签和图例。现在请实现那个地图，包含编写代码来生成标签以及图例。
10. （课堂项目）课堂中的每一个学生介绍自己感兴趣看到的一个简单模型给老师。老师可以选择其中的一些模型作为课堂项目或者课内讲解的例子。下图中显示了那些模型中的一些例子。它们来源于本书一位作者的授课班级。



3.11 实验题

1. 在2D空间内（比如X-Y平面）设计一个非凸多边形。尝试着用三角扇形绘制它。从单独的顶点出发，能够找到精确绘制该多边形的三角扇形吗？能够找到一个不能精确绘制多边形的三角扇形吗？另外，是不是能够证明，如果用从多边形中任意一点出发的三角扇形都可以精确绘制多边形，那么该多边形

就必须是凸的吗？

3.12 大型作业

1. 实现思考题9中讨论的酒会场景，使用一些标准的对象作为旋转木马，使用一个从外面获得的模型或者制作一个非常简单的“马”。请生成酒会的两个视图，其中一个是从外面看，而另一个则是从马上的视点观察。
2. （小房子）设计并生成一个简单的房子，有外墙和内墙、门、窗，在房间里有天花板，以及一个尖尖的屋顶，对不同的墙使用不同的颜色，这样从任何拐角看过去都有区别。在绕着房子的地方以及房子的内部设置一些不同的视点，显示从每一个视点看过去的房子呈现出什么样子？
3. （场景图分析器）实现在前一章中设计好的场景图分析器项目，每一个变换以及几何节点包含
157 OpenGL的函数名以及需要的参数来实现变换和几何体。该分析器应该能够对场景号display()函数。

第4章　建模的数学基础

立体解析几何是基于API的计算机图形编程技术的数学基础。在物理学导论、多元微积分、面向几何的线性代数等课程中对这些知识都进行了一定的介绍。本章将概述本书用到的基本数学知识，以帮助读者更好地理解本书内容。这些数学知识中的一部分是进行计算机图形编程所必需的，另外一部分则是为了知识的完整性，并且可以用于一些特殊的应用。掌握这些几何工具非常有用，例如，可以在使用光照的场景中计算三角形的法向。

本章先介绍三维直角坐标系以及点、直线、直线段在该坐标系下的参数表示，接着讨论向量和向量的计算方法，主要包括：向量内积、向量叉积以及各自的几何含义。接下来将三维空间用多个向量表示，并用矩阵来表示物体在空间中的变换（缩放、平移和旋转）。最后介绍关于平面的知识，包括多边形和凸形。此外，利用极坐标和柱面坐标有时能更好地描述建模过程，因此本章也对该部分知识进行介绍。

本章内容需要读者了解矩阵乘、矩阵转置等操作，并能习惯在3D空间中思考点、直线、多边形和多面体等问题。

4.1　坐标系

实线在欧氏空间中用两个点表示。如果一点为0.0（为与编程语言保持一致，本章用十进制实数表示），称为原点，另一点为1.0，称为单位点。从0.0到1.0是直线正方向，从1.0到0.0是直线反方向，称为负方向，直线方向和正负数相一致。 158

两垂直直线的交点称为它们的公有原点。在每条直线上选取离原点等距的两点作为单位点，它们就是测量的基准距离。原点和单位点构成二维直角坐标系，通常称为笛卡儿坐标系。原点到右边单位点的向量称为X单位方向，原点到上边单位点的向量称为Y单位方向，分别用$i = <1, 0>$和$j = <0, 1>$表示。该坐标系下的点可以用实数有序对(x, y)表示，也可以用从原点到该点的向量$<X, Y>$表示。如果用单位方向向量的话，则为：$Xi + Yj$。

二维直角坐标系下，任意两条不平行的直线必交于一点。在交点处，两直线共形成4个角，其中的锐角称为两直线的夹角。如果两条直线段起始于同一点，那么该角就是两直线段的夹角。用三角函数理论对夹角进行计算，它也是极坐标、球面坐标以及向量点积、叉积的基础。

三维直角坐标系建立在互相垂直且交于同一点的三条直线上，交点是公有原点，单位点是原点到三个方向上等距的点。该坐标系下的点都可以用三元有序组(x, y, z)表示。三条直线上从原点到单位点确定了三个单位方向向量，表示为$i = <1, 0, 0>$，$j = <0, 1, 0>$，$k = <0, 0, 1>$，分别对应X、Y、Z轴，称为三维空间的标准基。在这组基下，任意点(x, y, z)可以表示成$Xi + Yj + Zk$，如图4-1所示。

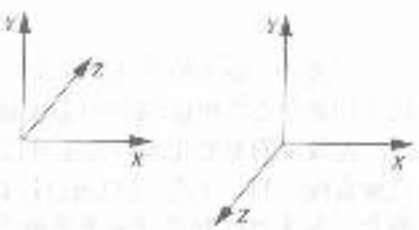


图4-1　左手坐标系（左）和右手坐标系（右） 159

在本书第1章的视域空间知识中出现了左手和右手坐标系的概念。这是由于三维直角坐标系中的第三条坐标轴是另外两条轴的叉积，它的朝向并不确定，因此出现左、右手坐标系，它们的判定方法如下：先将除大拇指外的4个手指指向第

一条坐标轴的朝向，然后弯曲四个手指从第一条坐标轴指向第2条，大拇指则指向垂直于这两条轴的第三条轴的方向。如果该过程用右手表示，则是右手坐标系；如果用左手表示，则是左手坐标系。试着用左右手表示图4-1所示的坐标系（X朝右，Y朝上）。

由于右手坐标系的自然性(电磁理论中的动量以及它产生的磁场都和右手坐标系相匹配)，许多计算机图形系统用它来表示。OpenGL建模也采用右手坐标系。

另外一些应用则采用左手坐标系。如RenderMan着色语言将X-Y平面作为空间的正面，并定义远离该平面的方向是Z轴方向。当点向空间的背面运动时，其Z值变大。这是左手坐标系。

4.2 四象限和八象限

在二维直角坐标系中习惯将其划分为4个象限。以点(x, y)为例，在第一象限，x, y都为正；在第二象限x为负，y为正；在第三象限x, y都为负；在第四象限中x为正，y为负。

三维空间可划分成8个象限，各个象限的名称叫法不一。但x, y, z都为正的象限统称为第一象限。人眼观察（摄像机）位置通常位于该象限。

4.3 点、直线和直线段

在一维空间中，任何实数都可以用直线上的点表示，它具有以下性质：

- 点到原点的距离是实数值乘以原点到单位点的距离
- 实数值正负号表示该点的方向

空间上的两点确定一条直线，如果令空间两点坐标分别为原点$P_0 = (X_0, Y_0, Z_0)$和单位点$P_1 = (X_1, Y_1, Z_1)$，那么，直线段P_0P_1上的所有点都可以用P_0和$P_1 - P_0$向量的一部分来表示。如果该向量进行了规格化，或者把它的长度变成1，那么该向量也可称为方向向量。任意直线段上的点$P = (X, Y, Z)$可以表示为：

160

$$P = P_0 + t(P_1 - P_0) = (1-t)P_0 + tP_1 \tag{4-1}$$

其中$t \in (0, 1)$。对于每个坐标分量的的参数方程可以表示为：

$$\begin{cases} X = X_0 + t(X_1 - X_0) = (1-t)X_0 + tX_1 \\ Y = Y_0 + t(Y_1 - Y_0) = (1-t)Y_0 + tY_1 \\ Z = Z_0 + t(Z_1 - Z_0) = (1-t)Z_0 + tZ_1 \end{cases} \tag{4-2}$$

因此，任何直线段都是单参数方程，所以，用线性参数方程来表示：

$$\begin{cases} x = a + bt \\ y = c + dt \\ z = e + ft \end{cases} \tag{4-3}$$

公式(4-3)表示的直线段称为参数化直线段。直线段上的点根据参数t确定。图4-2所示直线段上的各个点是根据t从0到1每次递增0.25求得。

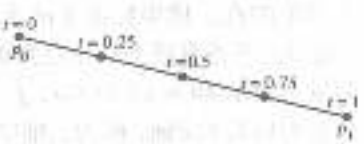


图4-2 参数化直线段以及根据参数值确定的点

直线段的参数化表示方法可以计算直线段的相交。假如已知直线上的一个点，可以通过计算确定那个点的参数t的值。例如，如果直线和平面或者其他几何体相交于点Q，通过向量计算$Q = P_0 + t(P_1 - P_0)$可以得到确定交点的t的值。根据具体情况，这个计算可能需要解一个或三个方程。直线和平面的相交的概念通常是几何计算的基础。

作为这种计算的例子，我们以两端点确定直线方程为例来说明该问题。假设点$P_0 = (3.0,$

4.0, 5.0)和点$P_1 = (5.0, -1.5, 4.0)$，那么，$P_1 - P_0 = (2.0, -5.5, -1.0)$。直线方程组为：

$$\begin{cases} x = 3.0 + 2.0t \\ y = 4.0 - 5.5t \\ z = 5.0 - t \end{cases}$$

用x, y, z表示的平面方程为：

$$Ax + By + Cz + D = 0 \tag{4-4}$$

如果平面方程是$6.0x - 2.0y + 1.5z - 4.0 = 0.0$，那么直线和该平面的交点是： [161]

$$6.0(3.0 + 2.0t) - 2.0(4.0 - 5.5t) + 1.5(5.0 - t) - 4.0 = 0.0$$

根据上面的等式求出$t = 27/43$后，我们就得到交点坐标(183/43, −41/86, 188/43)。

4.4 直线段、射线、参数化曲线和曲面

同样，当我们讨论通过0.0确定的唯一原点和1.0确定的单位点的直线的时候，任何直线段（它们的点位于直线段上两点之间）都对应于0和1之间的一个值。如果这两个点是P_0和P_1，t取0到1之间的任何值，则可以通过$P = (1-t)P_0 + tP_1$来表示它们之间的任何点。这是两点之间直线段的参数化形式。如果改变t的限制，可以得到其他线型。如果对t的取值不做限制，那么就得到了直线。如果对t只取正值，就得到以P_0为起点，并且没有终点的线，称为射线。

参数化直线段是通过从区间[0,1]到三维空间的函数决定的一系列连续点的特例。对于定义在[0,1]上的连续函数$x(t), y(t), z(t)$，点$\{x(t), y(t), z(t)\}$的集合是空间上的参数化曲线。这种曲线有非常大的用处，可以显示空间中移动点的位置，也可以用来计算在观看场景的时候沿着曲线的位置，或者用来描述平面曲线上二元变量的函数的表现特征。

参数化曲面是参数化曲线的二维形式。从二维的定义域$\{(u, v)|u, v \in [0,1]\}$开始，利用拥有两个自变量的三个函数$x(u, v), y(u, v), z(u, v)$来定义，点$\{x(u, v), y(u, v), z(u, v)\}$的集合形成了连续的曲面，参数化曲面也有十分重要的应用价值，我们将在第9章中详细介绍。

4.5 点到直线的距离

作为参数化直线方程的具体应用，我们计算空间中点到直线的距离。如果点是$P_0 = (u, v, w)$，直线方程为：

$$\begin{cases} x = a + bt \\ y = c + dt \\ z = e + ft \end{cases}$$

直线上的点$P = (x, y, z)$到P_0的距离平方为：

$$(a + bt - u)^2 + (c + dt - v)^2 + (e + ft - w)^2$$

它是关于t的二次方程，根据距离最小时，t的导数为0，可以得出：

$$2b(a + bt - u) + 2d(c + dt - v) + 2f(e + ft - w) = 0$$

上式是关于t的线性方程，只有唯一解。求出该解，将其代入直线方程就可以得到点P。 [162]

4.6 向量

空间向量和空间点都用二元实数<a, b, c>表示。这也可以用来表示空间里一点到另一点的运

动，向量的长度是所有分量平方和的根，记成$\|<a, b, c>\|=\sqrt{a^2+b^2+c^2}$。单位向量长度为1，由于单位向量表示了向量的方向，因此在许多建模和计算中都用到。如果$V=<a, b, c>$是长度为L的向量，归一化V向量就是将它的每个元素都除以它的长度得到单位向量：$<a/L, b/L, c/L>$。

两个向量之间的夹角是过原点的向量方向的两条直线段的夹角。如果向量是单位向量，那么夹角的cos值等于向量点积，在下一节讨论向量点积。

4.7 向量点积和叉积

向量点积和叉积是向量运算的两个重要操作。向量点积又称标量积，其结果是一个数值。假设两向量分别为$A=<X_1, Y_1, Z_1>$，$B=<X_2, Y_2, Z_2>$，A和B的点积就是：$A \cdot B = X_1*X_2 + Y_1*Y_2 + Z_1*Z_2$。向量点积的一个简单应用就是向量$A$的长度是其自身点积$A \cdot A$的二次方根。

向量点积的几何含义可以用下式说明：

$$U \cdot V = \|U\|*\|V\|*\cos(\theta) \tag{4-5}$$

其中$\|V\|$表示向量的长度，θ是两向量夹角。上式表示是将向量U投影到V后的长度，这也是向量点积几何意义的代数表示形式。如果两向量平行，那么它们的点积就是长度积，点积的正负号表明两向量是否同向。如果两向量垂直，那么它们的点积为零。不管向量的方向如何，因为锐角的余弦值是正的，所以如果两向量夹角是锐角，那么点积为正，反之如果是钝角，那么点积为负。如果已知两向量点积值和长度，则可以求出它们的夹角。

建立多边形的时候进行顶点计算是非常普遍的，所以需要了解可以通过$\cos^{-1}((U \cdot V)/(\|U\|*\|V\|))$计算任意顶点的角大小，其中$U$，$V$是顶点上的两个边向量。这个值在进行顶点法
163 向计算的时候非常有用，并且可以保存在多边形数据结构中。

图4-3说明了点积和向量夹角cos值之间的关系。任意两个向量点积的几何求解方法就是构造一直角三角形。斜边是向量的长度，因此，在V方向的向量长度是$\|U\|\cos\theta$，U投影到V的投影边长度是$U \cdot V/\|V\|$。如果V是单位向量，那么点积值就是投影向量的长度值。

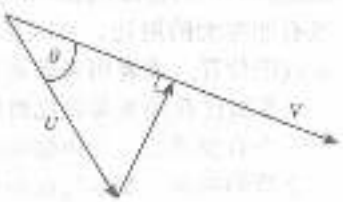


图4-3 向量U投影到向量V上

向量叉积又称为矢量积，用方阵的行列式求解，2×2方阵的叉积就是行列式值，而高阶方阵的叉积经递归求解。如2×2方阵，有

$$\det\begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc \tag{4-6}$$

3×3方阵，有

$$\det\begin{vmatrix} a & b & c \\ d & e & f \\ g & h & k \end{vmatrix} = a*\det\begin{vmatrix} e & f \\ h & k \end{vmatrix} - b*\det\begin{vmatrix} d & f \\ g & k \end{vmatrix} + c*\det\begin{vmatrix} d & e \\ g & h \end{vmatrix} \tag{4-7}$$

注意递归求解时各项正负号的使用。实际上不需要求大于3×3的高阶方阵，它们的计算方法也不太难。

向量叉积的几何含义是：两向量叉积结果也是一个向量。它垂直于前两个向量，长度是前两个向量长度积乘以它们夹角的sin值。如果两向量平行，那么叉积为零；如果两向量垂直，叉积的长度就是两条原向量的长度积。如果两个向量都是单位向量，叉积就是它们夹角的sin值。
164 如果用3×3矩阵表示向量叉积，那么第一行是坐标轴$<i, j, k>$，后两行分别是两向量的坐标值：

$$<a,\ b,\ c>\times<u,\ v,\ w>=\det\begin{vmatrix} i & j & k \\ a & b & c \\ u & v & w \end{vmatrix}=i\det\begin{vmatrix} b & c \\ v & w \end{vmatrix}-j\det\begin{vmatrix} a & c \\ u & w \end{vmatrix}+k\det\begin{vmatrix} a & b \\ u & v \end{vmatrix} \tag{4-8}$$

$$=<bw-cv,\ cu-aw,\ av-bu>$$

根据行列式变换交换法则，邻近两行或两列必须异号，我们知道$U\times V=-V\times U$，我们以$u=<3.0,\ 4.0,\ 5.0>$和$v=<5.0,\ -1.5,\ 4.0>$为例，求得$|u|=7.071$，$|v|=6.576$，因此$u\cdot v=15.0-6.0+20.0=29.0$

u, v之间的夹角的cos值是0.624。u, v的叉积可求解如下：

$$u\times v=\begin{vmatrix} i & j & k \\ 3 & 4 & 5 \\ 5 & -1.5 & 4 \end{vmatrix}=i\begin{vmatrix} 4 & 5 \\ -1.5 & 4 \end{vmatrix}-j\begin{vmatrix} 3 & 5 \\ 5 & 4 \end{vmatrix}+k\begin{vmatrix} 3 & 4 \\ 5 & -1.5 \end{vmatrix}=23.5i+13.0j-24.5k$$

叉积的结果应该和u, v都垂直，这可通过看它与u, v的点积值是否为0来判断。

向量叉积符合右手法则。如果4个手指方向是某一向量方向，手指卷曲方向是另一向量方向，那么大拇指指向就是叉积方向。这表明向量叉积的次序不能颠倒，而且向量叉积不能累加。举一个简单的例子，对上面提到的i, j, k，可以得到$i\times j=k$, $j\times i=-k$。如图4-4所示，三个向量方向在第一象限均可见，顺时针叉积是负值，逆时针叉积是正值。



图4-4 帮助识别向量方向的方法

向量叉积经常用于求解垂直于给定两向量的第三个向量。比如，根据多边形的两条边可以确定多边形的法向。如图4-5中的三角形，顶点A, B, C按逆时针排序形成三角形绘制时的正面。三角形的法向由向量$P=C-B$, $Q=A-C$的叉积得到。三角形法向在绘制时非常有用，比如法向和三角形内任意一点可以确定三角形所在的平面方程；当进行光照处理时，它们也需要归一化法向。

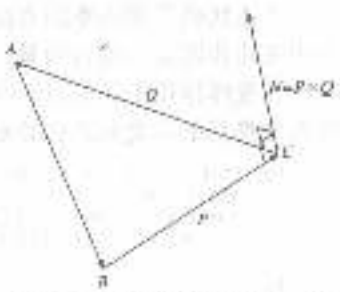


图4-5 用两条边的叉积表示的三角形法向

165

4.8 反射向量

反射向量在计算机图形应用中非常重要，它指的是某一方向向量在某物体表面的反射方向。在第6章光照模型中描述的镜面反射光（类似镜子的反射光）就是反射向量一个很好的例子。它在某点处的亮度与视点到该点的观察方向以及光线的反射反向相关。运动物体与另一物体表面发生碰撞后的反弹是反射向量的另一个例子。理想情况下，运动物体反弹后的速度与反弹前相同。为计算反弹方向，我们必须知道如图4-6所示的在碰撞点处的法向量。

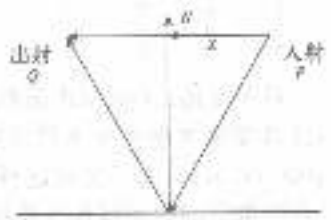


图4-6 入射向量P，出射向量Q，法向量N

166

图中，N是垂直于碰撞面的单位向量，P是入射向量。为计算反射向量Q，令N^*是Q投影到N的向量。根据对称性可知，N^*也是P投影到N的向量。因此有：

$$N^*=-(N\cdot P)N$$

从而得到$X=P+N^*=P-(N\cdot P)N$。从图中还可以得知$Q+P=2X$。因此，$Q=2(P-(N\cdot P)N)-P$，最终得到$Q=P-2(N\cdot P)N$。

4.9 变换

在前两章中我们对3维空间中的变换用函数操作表示，比较抽象。本章具体讨论这些函数操作怎样用矩阵表示，以及变换中缩放、旋转和平移矩阵的实现。

三维点(x, y, z)用齐次坐标表示是$(x, y, z, 1)$。平移变换时，变换矩阵T是在四维空间上的线性函数，可以用4×4矩阵表示如下：

$$T=\begin{pmatrix} t_{00} & t_{01} & t_{02} & t_{03} \\ t_{10} & t_{11} & t_{12} & t_{13} \\ t_{20} & t_{21} & t_{22} & t_{23} \\ t_{30} & t_{31} & t_{32} & t_{33} \end{pmatrix}$$

按次序的变换组合可以通过两个变换矩阵相乘实现。如果变换矩阵为S和T，分别用实数数组S和T表示，那么变换的组合是$C = S*T$。矩阵乘法的代码表示如下：

```
for (int i = 0; i < 4; i++)
   for (int j = 0; j < 4; j++) {
      C[i][j] = 0.;
      for (int k = 0; k < 4; k++)
         C[i][j] += S[i][k]*T[k][j];
   }
```

上述代码是矩阵乘的直接表示法。从几何角度解释，矩阵可以理解为向量的集合。矩阵相乘中左边矩阵是一组行向量，右边矩阵是一组列向量。因此，矩阵乘法是两组向量的的点积。

当变换作用于向量时，向量作为列向量右乘变换矩阵。如果变换矩阵是T[][]，向量是V_1，那么变换后的向量V_2的计算如下：

```
for (int i = 0; i < 4; i++) {
    V2[i] = 0.;
    for (int k = 0; k < 4; k++) {
        V2[i] += T[i][k]*V1[k];
    }
}
```

167

OpenGL中的变换操作可以用矩阵操作表示。对于缩放，如glScalef(sx,sy,sz)函数，则可以用下面的矩阵来表示：

```
sx  0   0   0
0   sy  0   0
0   0   sz  0
0   0   0   1
```

对于变换，OpenGL函数glTranslatef(tx,ty,tz)实际上需要4×4的满秩矩阵，在四维空间中的线性变换才能实现平移变换，三维空间中的线性变换只能把原点变换到(0,0,0)，不能变换到(tx, ty, tz)。为了实现这样的变换，我们必须用齐次坐标，并在四维空间中用线性变换将$(0,0,0,1)$变换到$(tx, ty, tz, 1)$。因此可以用满秩4×4矩阵来表示

```
1   0   0   tx
0   1   0   ty
0   0   1   tz
0   0   0   1
```

旋转变换比较复杂，以二维空间中绕原点旋转为例（如图4-7），旋转角度为θ，点$U = (1, 0)$变换到$(\cos\theta, \sin\theta)$，点$V = (0, 1)$变换到$(-\sin\theta, \cos\theta)$。二维空间中的所有点都可以用$xU + yV$线性表示。因此，$U$, V在旋转变换下的矩阵为：

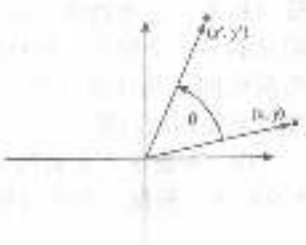


图4-7 二维空间中绕原点旋转θ角的示意图

$$R=\begin{bmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{bmatrix}$$

同样适用于其他点的旋转变换。用合适的U, V值右乘旋转矩阵，可以得到一个绕原点旋转θ的旋转变换矩阵。 168

OpenGL定义了一个绕直线旋转的函数glRotatef(angle,x,y,z)。其中*angle*是旋转的角度（度），<x, y, z>是直线方向。绕Z轴旋转可以表示为glRotatef(angle,0.,0.,1.)，由于绕Z轴旋转固定在XY平面，因此旋转矩阵是：

```
cos(angle) -sin(angle)    0       0
sin(angle)  cos(angle)    0       0
0              0          1       0
0              0          0       1
```

而绕X轴旋转表示为glRotatef(angle,1.,0.,0.)是固定在YZ平面，因此旋转矩阵是：

```
1          0           0          0
0     cos(angle) -sin(angle)      0
0     sin(angle)  cos(angle)      0
0          0           0          1
```

当绕Y轴旋转时，由于X轴与Z轴的叉积与Y轴反向，因此旋转角为负，这反映在sin函数的正负号上，所以绕Y轴旋转表示为glRotatef(angle,0.,1.,0.)，旋转矩阵是：

```
cos(angle)     0     sin(angle)     0
    0          1          0         0
sin(angle)     0     cos(angle)     0
    0          0          0         1
```

通过原点绕任意直线旋转的矩阵形式更为复杂，OpenGL的手册中已经给出了解释，本章不做详细讨论。

4.10 平面和半空间

在参数空间中，一条直线可以用一个参数确定，因此直线是一维的，而一个平面要用二个参数确定。因此平面是二维的。如果平面上两条不平行的直线交于点P，那么该平面上的点与点P之间的偏移可以用直线的方向向量与点P表示。虽然一条直线定义了平面的一维，但是一般不用两条直线定义平面，而是用不共线的三点来定义。这三点组成的两组点对实际上也定义了平面上的两条直线。

平面方程可以根据两条直线向量的叉积N = <A, B, C>来求。N垂直于两条直线，所以它也垂直于两直线所在平面。因此，平面的另一种定义如下：所有过固定点且垂直于<A, B, C>的直线。如果固定点坐标是<U, V, W>，平面上的点是<x, y, z>，我们得到直线方向向量<$x-U$, $y-V$, $z-W$>。如果用点积来表示<A, B, C>与该直线垂直： 169

$$\langle A, B, C\rangle \cdot \langle x-U, y-V, z-W\rangle = 0$$

将其展开可得到：

$$A(x-U)+B(y-V)+C(z-W)=Ax+By+Cz+(-AU-BV-CW)=0$$

上式给出了平面的参数化方程：

$$Ax+By+Cz+D=0$$

其中D的取值依赖于平面的固定点坐标。实际上，上述平面方程的系数<A, B, C>就是平面的法向量。必须注意，该方程中只有两个独立的变量，再次说明了平面是二维的。

我们知道一个平面将3维空间划分成两个半空间。那么空间中的点到底属于哪个半空间呢？这是二维空间点的归属问题。为了求解问题，先来看一下二维空间。任何直线把平面分

成两部分，如果直线方程是$ax + by + c = 0$，根据$f(x, y, z) = ax + by + c$的正负值可以确定点(x, y)是位于直线上方还是下方。同样，根据平面方程$f(x, y, z) = Ax + By + Cz + D$的正负值，我们可以确定空间点$(x, y, z)$位于平面的哪个半空间。实际上，当$f(x, y, z) = 0$时，空间点位于平面上。当$f(x, y, z) > 0$时，空间点位于平面的正半空间。当$f(x, y, z) < 0$时，空间点位于平面的负半空间。OpenGL使用$A$，$B$，$C$，$D$坐标来表示平面，并且在进行平面裁剪的时候利用半空间的方法和上述的函数正负号来决定是否显示。

例如，令三维空间不共线三点坐标分别为$A = (1.0, 2.0, 3.0)$，$B = (2.0, 1.0, -1.0)$，$C = (-1.0, 2.0, 1.0)$，这三点定义了一个平面。它的平面方程计算如下，不同的向量为：

$$A-B = <-1.0, 1.0, 4.0>,\ B-C = <3.0, -1.0, -2.0>$$

它们的叉积$(B-C) \times (A-B)$由2×2行列式得到，$<-2.0, -10.0, 2.0>$是平面法向。因此，平面方程是：

$$-2X-10Y + 2Z + D = 0$$

将点B坐标代入，常数D为-12.0，可得：

$$-2X-10Y + 2Z-12 = 0$$

170 满足$f(x, y, z) = -2x-10y + 2z - 12 > 0$的点位于平面的正半空间，反之则位于平面的负半空间。

4.11 点到平面的距离

点到平面的距离的求解方法在碰撞检测与其他应用中十分重要。给定平面方程$Ax + By + Cz + D = 0$，平面法向量$N = <A, B, C>$，单位法向量$n = <a, b, c>$。任意点$P = <u, v, w>$。为求点P到平面的距离，在平面上选取点$Q = (d, e, f)$（如图4-8），那么点P到平面的距离实际上就是向量$P-Q$在单位法向量n上的投影的长度$|(p-Q) \cdot n|$。这给了我们一种非常简便的方法来计算距离，尤其是我们可以任意选取Q点。

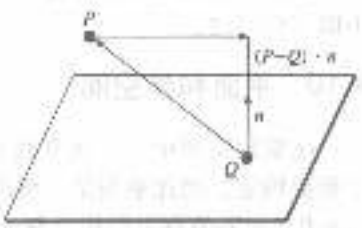


图4-8 点到平面的距离计算示意图

4.12 多边形和凸面

许多简单的图形系统（包括OpenGL）建立在多边形和多面体的建模和绘制基础上。多边形是一个由一系列线段围成的平面区域，这些线段首尾相连，最后一条线段的终点和第一条线段的起点重叠。由于多面体是由一组多边形在三维空间中形成的有界空间，因此它可以用多边形原理解释。我们重点讨论多边形的建模问题。

多边形建模是解决许多计算机图形学基本问题的关键。如用多边形插值实现第6章中的着色处理，第8章中的纹理映射，第10章中的几何模型绘制等。插值时为了保证多边形的属性完整与正确，多边形必须是凸的。凸多边形可以认为是没有缺口的多边形，也可以认为是多边形内任意两点（不管是多边形内部还是边缘）确定的直线段完全位于多边形内。

根据多边形划分的区域，可以简单定义多边形的外部或内部。对于凸多边形的内外部比较
171 容易确定，而一般多边形是非凸的也需要定义内部。对于凸多边形，如果有一点位于多边形内部，那么任何从该点出发的射线和该多边形只相交于一点（多边形的顶点是个例外，我们规定顶点只属于一条边）。如果一个点位于多边形内部，那么当它和多边形相交时，交点必定是0或2个。对于一般多边形，如果有一点位于多边形内部，那么任何从该点出发的射线和多边形相

交于奇数个点。如果点位于多边形外部，当点和多边形相交时，交点必定是偶数个。图4-9给出了3个实例。点A和点E是外点，点B，点C和点D是内点。注意点E，根据我们的定义，由于它的交点个数是偶数，因此是外点。但是实际上它是内点，所以我们的法则也不一定完全适用。

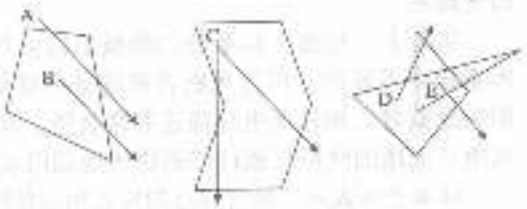


图4-9 多边形的内外点（左：凸多边形，中和右：非凸多边形）

用点的线性组合也可以定义凸面，我们首先来定义凸和。它是点$P_0, P_1, \ldots, P_n$的和$\sum c_i P_i$，系数c_i非负，且$\sum c_i = 1$。直线段的参数方程$(1-t)P_0 + tP_1$就是一个凸和的例子。如果多边形是凸的，由于所有直线段位于多边形内，因此多边形顶点的凸和也位于多边形内。反之也成立。这就给出了凸多边形生成的一种方法：凸多边形是从一组点集中选取任意子集并且包含凸和。该多边形称为顶点的凸包，是点集的最小凸多边形。其他用几何方法生成的凸包也非常有意思。

由于多边形的整个内部区域可以用所有顶点的凸和表示，因此顶点的深度信息，颜色平滑属性等都可以用顶点的凸和表示。由此可见，几何物体的凸面在计算机图形系统中非常重要。

许多图形系统（包括OpenGL）只支持凸多边形的绘制。对于非凸多边形，一般先将其细分成三角形或凸多边形，再进行绘制。OpenGL对于非凸多边形细分的方法比较复杂，可以参考相关文献，这里不进行讨论。 172

4.13 多面体

多面体是在三维空间中由多边形围成的体。操作多面体时，多边形就是它的边界。在场景图中，多面体是一个组结点，它的子元素是多边形。许多图形API不支持直接绘制多面体。在OpenGL中，也只有几个简单的多面体（球，圆环面等）模型。

凸多面体上任意两点组成的线段完全位于体内。多面体一般用多边形定义，因此在碰撞检测时我们也讨论多边形相交问题，不讨论多面体之间的相交。

4.14 极坐标、柱面坐标和球面坐标

我们已讨论过二维与三维直角坐标系。其他坐标系在某些应用中也十分有用，本节将讨论的坐标系基于角度而不是距离。但是图形API不能直接处理非直角坐标系，因此在最后，要将它们转换成直角坐标系。

二维平面上的点(X, Y)标记为从原点到该点的直线段。该点也可以用直线段与X轴正向之间的夹角θ和该点到原点的距离R来表示：

$$X = R^*\cos(\theta)$$
$$Y = R^*\sin(\theta)$$

其中，R是点到原点的距离，或者可以逆向表示成：

$$R = \mathrm{sqrt}(X^2 + Y^2)$$
$$\theta = \arctan(Y/X)$$

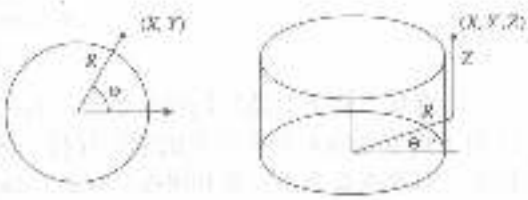


图4-10 极坐标系（左）和柱面坐标系（右）

其中，$\theta \in [0, 2\pi]$，它所在的象限根据X, Y的正负号确定。用(R, θ)来表示点的坐标方法称为极坐标系。如图4-10左图所示。 173

另一种代替三维空间直角坐标系的是柱面坐标系。它在二维极坐标系的基础上线性增加了一维，还有XY平面与过该点和Z轴平面的夹角以及点的Z值。柱面坐标系的点用(R, θ, Z)表

示。其中，R, θ与极坐标描述一致，Z是三维直角坐标中的值。图4-10右图表示了三维空间柱面坐标系。

实际上，柱面坐标系是二维极坐标系在三维空间中的扩展。由于它比较复杂，所以在图形系统中不常用。但当观察者在场景中移动时，为了保证平面物体在垂直方向向上，却始终朝向观察者，用柱面坐标描述非常合适。第8章中讨论的布告板技术就是柱面坐标应用的一个实例。而球面坐标在通过控制物体运动以实现角度或距离的平滑过渡时非常有效。

球面坐标表示三维空间点的位置和用经纬度表示地球上某点的位置相似。点的经度表示从赤道到该点的角度，从南90°到北90°，或－90°到90°。点的纬度表示从"子午线"到该点的角度，从0°到360°。地球上的任何一点都可用经纬度唯一确定。空间任何一点可以看成是相对于地球的位置；其经纬度是该点与地球中心连线到地球表面上该点的经纬度，其距离是该点到地球中心的距离。如果将地球看成是单位球，球面坐标系也基于相同的原理：从把地球看成以原点为中心的单位球开始，它有极轴和本初子午线。对于空间上的点P，从原点出发的射线指向P，并与球体相交，由此确定纬度Φ（从赤道平面上向北或向南的角度）和经度Θ（从本初子午线绕和赤道平面垂直的球体直径的旋转角度），以及从原点到P的距离R，那么空间点P的球面坐标就是(R, Θ, Φ)。

174 把球面坐标系转换为三维直角坐标系比较直观。图4-11说明了两者转换的关系。由于Z轴是垂直轴，因此转换的方程如下：

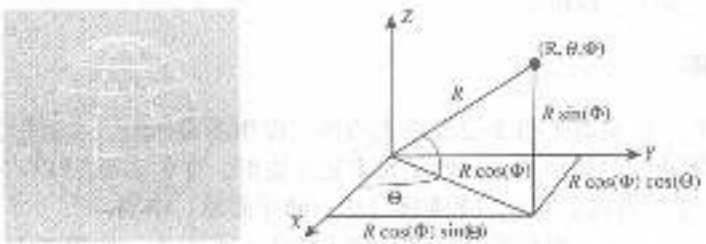


图4-11 球面坐标（左）转换成直角坐标（右）

$$x = R\cos(\Phi)\cos(\theta)$$
$$y = R\cos(\Phi)\sin(\theta)$$
$$z = R\sin(\Phi)$$

将直角坐标转换为球面坐标也很简单。参照图4-11，可以看到R是矩形的对角线，而角可以用基于边的三角函数来确定，可以得到如下等式：

$$R = \mathrm{sqrt}(x^2 + y^2 + z^2)$$
$$\Phi = \arcsin(z / R)$$
$$\theta = \arctan(y / x)$$

注意反三角函数是经度的主要值，而纬度取值是在0°到360°，因此，sin函数值和cos函数值的正负号必须和角度所在的象限保持一致。图4-11描述的球体包括了经线和纬线、直角坐标系，以及在直角坐标系和球面坐标系转换的方法。

4.15 碰撞检测

本节将讨论在物理空间中两物体在运动或交互时发生碰撞的逻辑关系。实际上，碰撞检测可以看成是一种特殊的建模方法，它包括若干步骤。我们将介绍几种碰撞检测的方法。

对于一个场景，我们必须清楚哪些物体可能发生碰撞，碰撞的类型又可能是哪几种，碰

撞检测过程包含许多步骤，而最好的两种加速方法是尽可能避免检测，或者是必须检测时用尽可能最简单的方法进行检测。

碰撞检测时，我们必须知道碰撞时具体的三维点坐标，虽然这些坐标可以在物理空间或在眼空间中获取，但这不符合高层图形API应用的原则。图形程序应尽可能避免对底层API的访问。许多图形API都提供了获取三维点坐标的API函数。在OpenGL中就用glGetFloatv(GL_MODELVIEW_MATRIX)函数来获取当前任何视点下的模视矩阵。它返回包含16个实元素的列数组，代表对当前点进行变换的4×4矩阵。我们将它乘以原始的顶点坐标，就得到眼空间中该顶点的坐标。

为简化碰撞检测过程，通常用是否可能发生碰撞来避免实际不发生碰撞的情况，对于不可能发生碰撞的物体直接丢弃。用真实物体的近似简化模型，如包围球、包围盒等包围物体进行碰撞检测，计算相对容易。包围球是以物体为中心的球，物体完全被包含在该球内。包围球用球心和半径表示。包围盒是一个六面体，每组对称面分别与一条坐标轴平行，物体完全包含在该六面体中。判断两个包围球是否发生碰撞的方法是检测它们的距离是否小于直径。而判断两个包围盒是否发生碰撞的方法是在每个轴向上计算包围盒边缘的距离。由于眼空间中物体可能发生变换，因此要求包围球或包围盒也定义在物理空间中，以免判断出错。

碰撞检测时先根据包围球或包围盒进行检测。如果可能发生碰撞，则进行更进一步检测。此时必须用物体本身，假设物体由多边形网格组成，而最小的多边形是三角形。因此最底层的碰撞检测是判断两个三角形是否相交。除非已知两个物体之间哪些三角形距离最近，否则需要测试所有可能的三角形对，所以，我们从如何快速排除三角形开始。

为了简化三角形相交测试，我们先计算一物体中的三角形与另一物体的包围球或包围盒的距离。以包围球为例，可以判断三角形三个顶点坐标到球心的距离是否大于三角形的最长边。如果已知三角形的重心，就可以用重心到球中心的距离是否大于半径来判断是否相交。三角形重心是三边垂线的交点，它也是三角形外接圆的圆心，如图4-12所示。

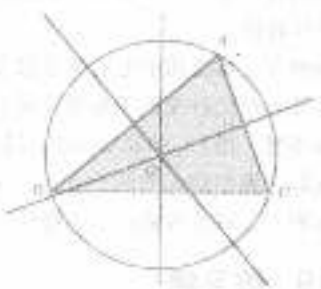


图4-12 三角形外接圆心与外接圆

如果三角形与物体包围球或包围盒相交，那么必须判断该三角形是否与物体上的所有三角形相交，此时，我们先判断该三角形是否与物体上另一三角形所在的平面相交。假定平面方程是：

$$f(x, y, z) = Ax + By + Cz + D = 0$$

如果将一个三角形三个顶点的坐标值代入平面方程得到的值的正负号一致，则它和该平面不相交。否则，我们判断该三角形的所有边是否与平面相交并计算出边与三角形所在平面的交点坐标。假设直线段的参数方程是$Q_0 + t(Q_1 - Q_0)$，则该直线段与平面的交点坐标是关于t的参数。如果$t < 0$或$t > 1$，那么直线段与平面不相交。否则就要判断交点是否位于另一个三角形内，如图4-13所示。

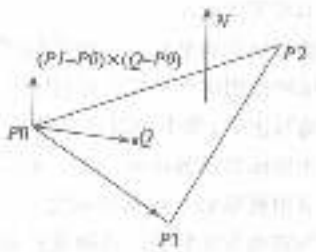


图4-13 点在三角形内

假设三角形顶点按逆时针排序，任何位于三角形内的点都在三角形边的左边。该条件可以用边向量和点到边顶点向量的叉积来表示。如果每个顶点的叉积的方向都一致，那么该点位于三角形内。实际上该点必须满足以下所有条件：

$$N \cdot ((P1-P0) \times (Q-P0)) > 0$$

$$N \cdot ((P2-P1)\times(Q-P1)) > 0$$
$$N \cdot ((P0-P2)\times(Q-P2)) > 0$$

其中N是平面法向。

4.16 高维空间

客观世界是三维空间，因此我们的感知与体验均受限于三维空间。但是，图形系统所显示的信息不应局限于三维空间。由于高维空间中的信息不能直接显示，因此必须通过其他方法进行处理，比如投影或数据混合等技术。此外，在三维信息中增加额外的非空间信息来表示高维信息也是常用的方法。在第9章中的可视交流与科学可视化及其应用中，我们将具体讨
177 论高维信息表达的方法。

4.17 小结

本章论述了计算机图形编程的三维解析几何知识。这些知识在几何模型的放置、定义法向量或场景中物体的运动属性以及它们之间关系时非常有用。它描述了图形程序所要表达的数学或几何行为，使得程序员能灵活而有效地应用。

4.18 思考题

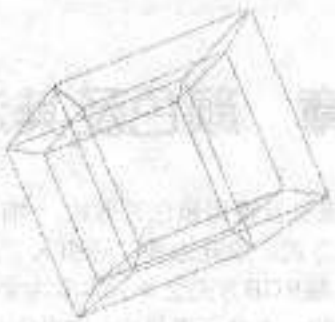
1. 假设在三维空间中定义了2种直角坐标系，试说明如何将几何模型的缩放、旋转和平移等操作在两者之间转换。
2. 请问在三维空间中也能实现思考题1要完成的功能吗？请给出具体的分析（提示：考虑左右手坐标系）。另外，在反射变换下也能实现两种坐标系的变换吗？反射指将某一坐标值的正负号反号，而其他坐标值不变。用3×3矩阵表示的反射具有什么样的形式？
3. 选取三维空间中的两个点，写出连接它们的直线段参数方程。请问怎样做变动才能改变该方程为射线方程？怎样改变成直线方程？

4.19 练习题

1. 给出几对向量，用本章知识求出它们的点积与叉积。在计算中请验证平行向量、正交向量与已知夹角的向量之间的点积与叉积，并通过计算验证点积与叉积中的三角关系。
2. 写出两次绕不同轴旋转、或1次旋转1次平移的变换矩阵。试用矩阵乘实现这些变换，并验证变换组合不可调换次序。
3. 请写出直角坐标系、极坐标系、柱面坐标系之间的转换方程，把它们做成通用的工具，以便在将来建模时使用这些坐标系时可以使用。
4. 请写出第2章中用四边条带与三角扇形在球面坐标系中逼近球体的程序，并实现将球面坐标转换为直角坐标的函数sovertex()，最后用glVertex返回坐标值。
5. 请用极坐标与柱面坐标建几个几何模型，并通过三角形、四边形或多边形来绘制，最后用练习题3中
178 的转换方法将它们转换成标准的OpenGL中的模型。

4.20 实验题

1. 假如有一个物体它的维度高于三维，那么有很多种方法可以把它投影到二维空间上，然后对它进行观察。例如，对于四维的“立方体”（它在四维空间上有16个顶点，每个顶点的分量是1或者0），如果把它投影到三维空间，可以得到下图所示的图像。



对每个顶点的坐标尝试用多种方法把四维物体投影到三维空间。然后运用每种方法创建那个四维立方体的视图。

2. 编写函数，测试点是否在三角形内，并利用这个函数实现给定直线和给定三角形是否相交算法，这些算法的实现都在三维空间上。最后把这个算法扩展成测试两个三角形是否相交。 179

第5章　颜色及其混合

本章介绍在计算机图形学中定义及使用颜色的方式，描述颜色主要使用RGB模式，这是大多数数字图形图像使用的颜色方式。除此之外，还引入了颜色亮度方式，这主要是便于弥补在颜色视觉理解方面的不足。除RGB方式之外，本章主要讨论HLS和HSV模式。在有些时候这两种颜色模式也是十分有用的，本章还提供它们与RGB模式之间的转换。RGBA是RGB模式的扩充，它可以用于颜色混合并模拟透明度。本章还引入了颜色浮雕这种作为基于颜色的3D显示技术。最后，给出了OpenGL中的颜色技术，用户可以根据颜色编程技术实现精彩的图像显示效果。

5.1　简介

颜色是计算机图形学中的基本概念。必须合理地定义计算机图形中的颜色，使得它们不仅能够大致反映真实世界中的颜色，并且能方便地被应用程序处理。图像中的物体获得颜色的方式有两种：直接为物体设置颜色，或者定义物体的材质属性，通过光照模型获得颜色。本章只讨论前一种方式，后一种方式在第6章中讨论。

人眼和感官系统对光线的感知能力比标准计算机图形学的处理方式复杂得多。人眼视网膜中有两种细胞与光线有关。一种细胞称为杆状细胞，它对亮度（而非颜色）敏感。杆状细胞在视网膜中心之外的区域密度很高，它是低亮度时的主要视觉来源，但它们对清晰度不敏感。因为它们不是分别对应于不同的神经元的。另一种细胞称为锥状细胞，锥状细胞分为三种，分别对应于不同的光线波长，分别感应于红色、绿色和蓝色，这就是色彩学中三刺激元的来源。这些细胞围绕着视网膜中心凹点汇聚在一起。每个细胞有自己的神经元，可以很好地区分颜色细节。计算机图形学中使用纯红色、纯绿色和纯蓝色来仿真眼睛中的三种锥状细胞，模拟真实世界中的物体外观属性。

仅用RGB来表示真实世界就过分简单了，一个物体可以反射整个颜色谱，而不仅是纯RGB波长，因此，需要用更复杂的模式来处理整个光谱。我们的颜色模型是基于采样值的，比如在以实数值表示的几何模型中，几何图形的整数坐标就是一种采样。因此，正如几何模型一样，颜色值对走样问题非常敏感。本书实现的颜色模型只是真实世界颜色问题的简化方法。

造成人类视觉系统色彩走样的一个原因来自于视觉神经元的特定行为。虽然前面提到的单个神经元对应一个细胞，但事实上人类的视觉感知来自于整个神经网络。神经网行为是一种抑制行为，对一个神经网区域的强烈刺激会抑制其附近区域的刺激。通常把这称为侧抑制。这是许多视觉模型（参见图5-19）以及马赫效应（参见图5-9）的根源。读者可以不关心此类问题的细节，但是在设计图像时最好关注到这个事实。

定义颜色有多种方法，但所有这些方法基本上都是基于人眼视网膜三种锥状细胞的刺激原理的，因此，计算机图形学的所有颜色模型也使用此三刺激元基本理论。计算机图形学一般使用RGB彩色模型，它与计算机监视器的物理特性相匹配，即电子束发射出红色、绿色、蓝色光线构成一种模式。还有其他定义颜色的方法，但在本章中只做简单介绍。对于不同模型之间的转换信息，请参见相关的参考文献，特别是[Foley *et al.*]。

因为计算机监视器使用三类阴极射线管，每类射线管可以根据电子束的强度发出不同亮度的光线，所以一般方法是通过三种原色及不同亮度级别来定义一种颜色。颜色亮度级别是原色最大亮度能量的比例，所以，RGB颜色可以表示成三元组（r，g，b），每个成分表示该种颜色中含原色的数量，按次序分别表示红色、绿色和蓝色的含量。在本书中，彩色成分表示为最大亮度的比例。每一基元的比例值都在0.0与1.0之间（包含0.0与1.0），这是基元颜色的饱和度。每一成分值越大，则该颜色的亮度越大。黑色可表示成（0.0，0.0，0.0），白色表示成（1.0，1.0，1.0），三原色分别表示成红色（1.0，0.0，0.0），绿色（0.0，1.0，0.0）和蓝色（0.0，0.0，1.0）。此时，该原色对应的成分达到最高亮度，而其他成分则为最低亮度，其他颜色是三种原色的混合。 181

有些图形API（特别是早期的版本）可能会使用基于整数的颜色表示方式。这时，每个颜色成分用一个整数表示，该整数的大小范围随系统的不同而不同，例如，一般情况下每个成分有8位，整数的范围在0到255之间。但是，目前的图形API更多的是使用实数，因为这种方法不依赖于设备，目前的图形卡一般使用16位或32位浮点数。这两种系统在彩色处理方面有细微的区别，这主要由API来处理，用户不必关注这些区别。生成彩色的过程事实上非常复杂，要由监视器或其他显示设备来处理。这些彩色表示方法和硬件处理方法对API程序员而言是透明的，而且它们可以用于许多平台。

除了光线的三个彩色成分，现代图形系统还加入了第四个成分，称为α通道，用于表示材质的透明度。透明度也用0.0与1.0之间的实数值来表示，0.0表示完全透明，1.0表示完全不透明。使用透明度可以在显示物体时也显示背景的部分内容，或者在显示前面的物体时也透出后面物体的部分内容。这称为α混合，α混合可以在新绘制的物体上混合部分原来保存在Z缓冲区里的内容。如果要生成的图像具有多级透明度，则必须十分关注所生成的物体序列，以从远到近的次序画图，获取背景物体彩色的渗透度。具体细节参见本章后面的部分内容。

5.2 原理

5.2.1 设置几何物体的颜色

使用颜色的基本原理非常简单：当一种颜色设置好之后，随后的所有几何元都被赋予这种颜色，直到颜色被修改。这意味着，如果场景中不同部分使用不同的颜色，则用户需要对这些不同的部分设置不同的颜色。做到这一点并不困难，但在具体实现时需要十分仔细。

在场景图中，颜色是外观节点属性，是与几何节点同时定义的，而且是外观属性中首先定义的。写场景图代码时，必须在处理几何信息之前先处理外观属性，这样才能获得正确的外观信息。

用户可以通过任何方式将颜色放入外观节点中。在下面的几节中我们讨论三种不同的彩色模型，用户原则上可以使用任意一种彩色模型，因为大多数API都支持RGB模式，所以不管使用哪一种彩色模式，都需要在生成外观节点之前将其他彩色模式转换成RGB模式。在下面的几节中我们还会提供两种颜色模式转换的程序代码。

5.2.2 RGB立方体

RGB模型是颜色空间的一种几何表示。颜色空间上所有的点(r, g, b)，每个成分都在0与1之间。因为颜色三元组类似于3D空间上的点，所以将颜色空间单位立方体表示成以r，g，b 182
为坐标的空间顶点的坐标。这种定义方式非常简单自然，大多数计算机图形学程序员非常容易理解。

RGB立方体如图5-1所示，从图5-1的黑色和白色两个顶点看，可以看到所有颜色都在立方体的表面上。白色顶点附近的三个顶点是洲蓝、桃红和黄色，黑色顶点附近的三个顶点是红色、绿色和蓝色，这是RGB颜色模式的一个关键特征，当原色数值增加时颜色变亮。当然，立方体内部的所有点也都有对应的颜色，例如，RGB立方体中央对角线方向，从(0，0，0)到(1，1，1)对应的颜色，其三原色成分值相等，即为灰度色，一般在表示彩色图像时用来提供中性背景色。

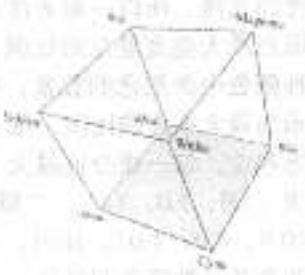


图5-1 RGB立方体表示，参见彩图

下面来看一个例子，说明在RGB系统中如何定义颜色。沿着RGB立方体的边线，从黄色(1，1，0)到蓝色(0，0，1)，看看颜色有什么变化，即颜色 $(a, a, 1-a)$，a的取值从0到1。从图5-2我们可以看到六个颜色，从左至右，a的值分别为0，0.2，0.4，0.6，0.8和1。我们注意到，颜色从黄色（即红色和绿色混合）变来，蓝色变暗（参见彩图），这些颜色显示在灰度值为0.5的背景上。中间没有空隙，特别要注意两个颜色之间的边，这是两个几乎相等的相邻颜色之间的视觉效果，称为马赫带，将在本章的后面讨论。

183



图5-2 黄色与蓝色间的六种颜色序列，参见彩图

5.2.3 亮度和色弱

颜色的亮度是指在不考虑颜色纯度的情况下颜色的亮度或强度。这个概念对发射光体系的屏幕有特别的意义，因为在这种情况下，亮度事实上对应于从屏幕发射出来的光线电子数。对RGB图像来说，亮度计算很容易。对于三原色来说，绿色是最亮的，因此它对颜色亮度的贡献最大，红色其次，蓝色亮度最弱。事实上真正的亮度随着系统的不同而不同，甚至因显示设备的不同而不同，因为颜色亮度的差异就是激发电子束发射的电压的差异和荧光屏对电子束做出反应的方式的差异。对于标准的基于TV的显示设备，我们可以使用如下相对精确的亮度公式

$$\text{亮度} = 0.30^{*}\text{红色} + 0.59^{*}\text{绿色} + 0.11^{*}\text{蓝色}$$

因此，绿色:红色:蓝色的亮度比值大致为6:3:1。更精确的红绿蓝比例分别为0.299:0.587:0.114。对于HDTV标准，该参数值还有一些细微的差别，绿色成分还要多一些，红绿蓝三者的比值分别为0.2125:0.7154:0.0721。

为了表示同一亮度下RGB颜色的效果，构建一个平面$0.30R + 0.59G + 0.11B = r$，随着$r$值的变化，该平面切割颜色立方体，如图5-3所示（更好的效果参见彩图）。但在灰度图中该平面上颜色是不一致的，将彩色转换成灰度时绿色比公式所表示的更亮一些。有关亮度的细节还有许多，用户在自己的系统中实现时还可获得更好的效果。



图5-3 以灰度图形式表现的RGB立方体的等亮度图，参见彩图

对于颜色，用户必须注意，某些颜色或颜色组合很难区分，大约8%至10%的高加索男士

是色弱人士，非高加索男士大约为4%，对于女性，则为0.5%。不管使用什么显示设备，这些人都不能区别颜色。但是，即使他们不能区分色度上的差别，但大多数人都可以区分亮度。如果用户系统是面向高加索男性的，就要注意图像中的颜色必须用不同的亮度来表示，而不仅仅是使用不同的色度。

亮度也是很重要的概念，因为对于图像的解释有一部分是专门针对亮度的，用户必须正确使用颜色的亮度值。例如，在本章的后面部分讨论颜色渐变时也用到了亮度信息进行颜色渐变，用颜色中的亮度值表现某种意义下的某个数值。

184

5.2.4 其他颜色模型

有时RGB模式使用起来并不方便。当用户要获取一个特定的颜色时，很少有人能想到它的红绿蓝色的比例。其他方法可以更自然地说明一种颜色，由于RGB模式与颜色的生成方式不太匹配，因此需要有更多的方法进行颜色建模。

两个比较直接的颜色模型是HSV模型（Hue-Saturation-Value）和HLS（Hue-Lightness-Saturation）模型。这两模型用色度（针对红色、桃红、蓝色、浅蓝、绿色或黄色标准色的偏移量）、亮度（颜色的着色度）和饱和度（颜色的纯度）来描述颜色。通过这种模型，"较暗的桔红色"可以用以下方式进行量化描述：色度值在黄色偏红色方向，亮度值较小，饱和度较高。

首先来看HSV模型。就像RGB颜色空间有立方体模型表示方式一样，HSV颜色空间也有一个几何模型：顶为圆平面的圆锥形（如图5-4所示），更详细的细节参见彩图。沿着圆周用角度表示色度，红色在0度，绿色在120度，蓝色在240度。从垂直轴到外边缘的距离表示饱和度或某个颜色的原色的量，数值从中心的0（无饱和度，只有灰度）到边缘的1（全饱和度亮色）。垂直轴表示亮度，从底端的0（黑色）到顶端的1（白色）。因此，HSV颜色可以用圆锥内或圆锥上的点（三元组）表示，(40.0，1.0，0.7）表示"较暗的桔红色"。本章的末尾部分将给出显示该几何内容的代码。

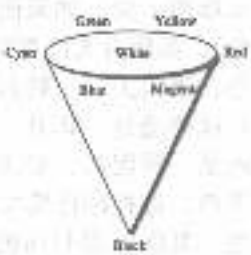


图5-4 HSV颜色模型。参见彩图

185

HSV模型空间需要仔细理解。其顶面表示almost色较亮的颜色，因为当颜色变得较亮时颜色的饱和度会下降。模型在底面收敛为一点，因为在颜色较黑处颜色的区别较小。在这个模型中，灰色的饱和度为0，并且在圆锥的中心垂直线上。其色度是无意义的，但必须包含在里面。

在HLS彩色模型（如图5-5所示，参见彩图）中，其几何模型与HSV模型很相似，但其顶面也收敛为另一圆锥的顶点。色度与饱和度的意义与HSV模型的意义相同。但用明度来替代亮度，最亮的颜色其明度为0.5。使用双圆锥方式的原因是当颜色越亮，其色度和饱和度将减小，就像颜色越暗时那样。HLS模型很接近于绘图学中的术语色彩和色调，色调最亮处在明度为0.5处，当明度增加时，色彩变浅；明度减小时，色调变深。与上面提到的HSV模型一样，圆锥中心线为0饱和度的灰度，这时的色度是无意义的。

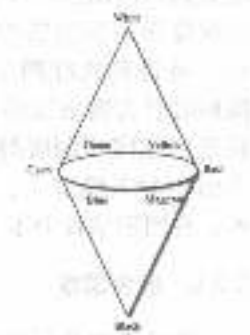


图5-5 HLS双圆锥颜色模型。参见彩图

186

从HLS双圆锥模型的顶端和底端看，与HSV单圆锥模型相似，但从侧面看，则有很大区别。图5-5分别从红、绿、蓝三原色侧面显示HLS双圆锥模型。注意，因为每一面图像大致要覆盖120度的圆锥表面，所以相邻的图像边必须匹配。从HLS模型的顶部和底部看，它的含义与HSV圆锥相似。图5-5所展示的图像可能与用户想到的不完全一样，本章的后面部分将给出代码例子，可以进行交互式修改。

从一个颜色模型转换到另一个颜色模型的函数很简单，本章的后面部分会给出从HSV转换到RGB的函数，以及从HLS转换到RGB的函数。参考文献[FO]中给出了所有的转换函数。

上面给出的颜色模型都是基于类似于计算机监视器或光线发射到人眼的设备的。这称为发射色，即当屏幕的阴极射线管发射电子时，将亮度加到不同的波长上。目前大多数程序的显示设备都是屏幕，因此，这是计算机图形学中提到的主要方式，但不是唯一的方式。例如，在印刷品中，用户看到的颜色是光线透过墨水从纸上反射的，这称为吸收色，即从纸反射出的光线减去某种颜色而给出的。这是一种完全不同的处理方式，必须做减法。图5-6给出它的工作原理，细节可参见彩图。RGB相加可生成CMY，最终生成白色，因此发射色有时称为加色，而CMY相减生成RGB，最终生成黑色，因此，吸收色有时称为减色。

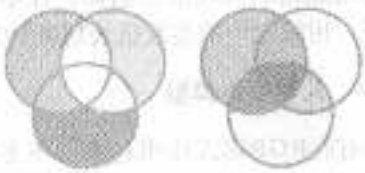


图5-6 发射色（左）和吸收色（右）。参见彩图

墨水或胶片使用吸收色处理，某些颜色可以透射出来，其他颜色则被滤掉。吸收色的基元是湖蓝（白色穿过蓝色和绿色形成的颜色）、桃红（穿过蓝色和红色的颜色）、黄色（穿过红色和绿色的颜色）。从原理上说，如果使用三种墨水（湖蓝、桃红、黄色），则所有光线都穿透不出去，因此，会显示出黑色。事实上，一般的材料都不是那么完美的，总有光线能透过，因此，用这三种颜色不能构成黑色，而是一种泥灰。要成为真正的黑色，还要加上部分黑色。这种颜色模型称为CMYK（湖蓝－桃红－黄色－黑色），是打印机或其他吸收色处理机制的基础。图5-7显示不同颜色分层构成完整颜色图像的过程，具体细节见彩图（因为受颜色的限制，在正文中只能看到黑白图像）。图5-7（具体参见彩图，正文只能是灰度图）顶部的图是全彩色图像，包含黑色、湖蓝、黄色和桃红四个分层图像（从左上角图开始，按顺时针方向依次向下的四个图像），这四个图像是彩色图像印刷制版时必须提供的。这里就不再详细介绍CMYK模型了，因为它只是用于打印和类似技术，在图形编程中不太常用。在第15章介绍图形硬件时再详细描述。

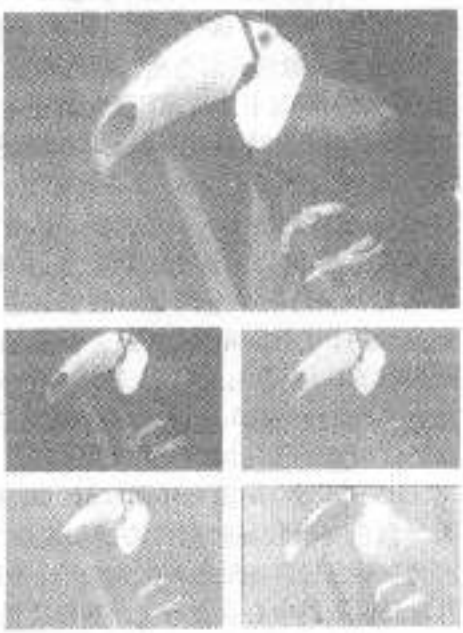


图5-7 印刷时的分层图片。参见彩图

5.2.5 颜色深度

颜色模型是与设备无关的，用实数表示，因此，理论上可以显示无穷多种颜色，但是事实上，没有一种设备可以显示无穷多种颜色，图形设备根据存储量的大小来使用颜色的个数。

计算机图形学的基本模型是基于屏幕显示的，称为真彩色，对于屏幕上每一像素，其颜色信息都存于图像缓冲区中。每一像素的颜色存储位数称为设备的颜色深度，一般情况下，R、G、B三原色分别使用8位，因此，每一种颜色有24位。但这不是统一的，有些系统使用少于或多于24位来存储颜色，并且三原色的存储位数也不总是一样的。现代图形系统经常使用32位彩色或48位彩色，甚至还有使用128位颜色的。但是，有些图像格式不支持太高位数的颜色，例如，在标准GIF格式[MUR]中只说明了8位索引色。

正如前面提到的，因为只能用三原色来表示真实世界的所有颜色，所以会产生颜色走样。但是，当系统只有有限的颜色深度时，表示真实世界的三原色也只有有限的颜色深度，此时情况会更糟。计算获得的精确颜色可能不能在屏幕上显示，只能取最到受系统颜色深度限制的附近颜色上。如果系统包括的颜色有限，则这种情况就更明显。这会导致一种严重的后果，称为马赫带（如图5-8所示）。当大区域的实体颜色中出现细微的变动时，会出现马赫带现象，在跨多边形边界处分段线性强度发生变化时也会出现马赫带现象，因此在有限颜色深度下马赫带现象很常见。人眼可以很容易地看到颜色分界线，由此可作为绘制算法质量的评判标准，这种边界上细微的差别会严重破坏平滑图像的质量。仔细观察图像的马赫带现象，将几何反走样技术用到颜色上，就可以使马赫带现象不明显。图5-8是一个用减小的颜色深度绘制的图像，其中包含马赫带，在图像前面的紫色部位可以很明显地看到马赫带。



图5-8 马赫带图像

189

5.2.6 色谱

颜色的限制不仅在于能显示的颜色数有限，而且在于显示设备技术的限制，不管使用哪种显示技术，比如屏幕的阴极射线管，纸上的墨水或平板显示器的LCD单元，所有显示技术都只能使用有限种颜色。对于所有颜色技术也如此（如打印机、视频、彩色胶片）。设备的颜色范围称为色谱，设备色谱限制了表示某些图像的能力。对不同设备色谱的讨论超过了计算机图形学的基本知识的范畴，但是理解这一点是很重要的。图5-9（最好参见彩图）显示最常用的颜色参考空间(1931 CIE空间)，图中给出了三个特殊颜色R、G、B点的值，以及监视器色谱的三角形区域。（注意，这只是真实颜色空间的近似，因为它本身也是用有限的颜色打印在设备上的。）

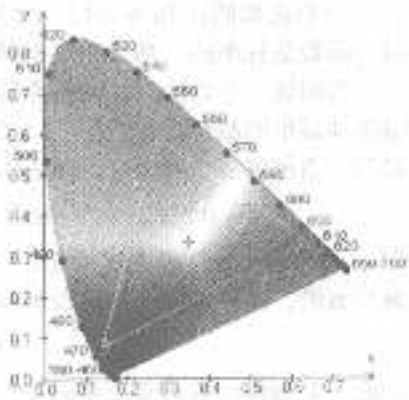


图5-9 CIE颜色空间，参见彩图

5.2.7 颜色混合与α通道

在大多数图形API中，颜色表示不仅使用RGB三原色，还可以包含混合级别（有时认为是透明级别）。颜色可以表示成四元组(r, g, b, a)，包含混合的颜色模型称为RGBA模型。颜色的混合级别a称为α值，取值范围为0.0～1.0，常常表示不透明度而非透明度，也就是说，如果使用标准类型的混合函数，且α值是1.0，则颜色完全不透明，如果α值为0.0，则颜色是全透明的。引入α通道可以对图像进行组合[POR1]，把一个图像覆盖到另一个图像上，而被覆盖图像的一部分可以显示出来。因此，“透明”功能（有时）可以通过混合操作完成。混合操作将要绘制的像素颜色与当前颜色缓冲器中的像素颜色做比较，根据所选择混合函数，将对象颜色与当前缓冲器中的颜色相混合产生一种颜色。常用的混合函数是将这两种颜色取平均。这种混合技术与生成图像的次序有关，参见本章后面的例子，如果生成图像的次序不同，则结果差别很大。

190

混合色和透明色的区别很大。对于透明色，一般考虑彩色玻璃，这种材质体现的是吸收色而非发射色，因为只有某些波长能够通过，而其他颜色则被吸收了。但这与α值的含义不

同，混合色是对发射RGB值的平均，不同于透明度的吸收模型。在创建图像某种效果时，了解这种区别很重要。关于混合的另一个问题是，对RGB空间中颜色的平均可能达不到预想的颜色，RGB颜色模型可能是感官颜色混合中最差的模型，但是，在大多数图形API中也没有其他选择。

5.2.8 使用混合达到透明效果

要达到透明效果可以使用颜色混合功能。首先，透明是指能够看到透明色后面的物体。这只是最简单的一步，如果要使用混合达到透明效果，最重要的是要让隐藏在其他物体后面的物体显示出来。因此，要先画不透明的物体（α成分为1.0的物体），然后再画透明的物体，在画有混合色的物体时关掉深度检测，画完之后再打开深度检测。

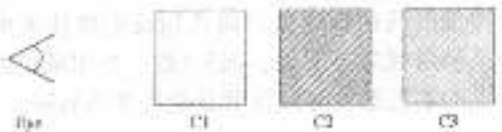


图5-10 画物体的顺序

然而，这么做还不够，事实上只是关掉深度检测来获得透明效果有可能造成图像次序混乱。假设要画如图5-10所示的三个物体。假设物体1、2、3的颜色分别为C_1、C_2、C_3，显示次序分别为1、2、3（靠近眼睛的物体为1），背景为白色，每种颜色的α值为0.5，假设不使用深度缓冲器，混合函数是标准的。那么如何在屏幕上显示这些物体的颜色呢？

191

当画第一个物体时，帧缓冲器中保存颜色C_1，缓冲器中没有其他颜色。画第二个物体时，帧缓冲器中的颜色是$0.5*C_1+0.5*C_2$，因为前景C_2的α值为0.5，背景色C_1的权重是$0.5=1-0.5$。最后，当画第三个对象时，颜色C_3在其他颜色上，为：

$$0.5*C_3+0.5*(0.5*C_1+0.5*C_2)，或者\ 0.5*C_3+0.25*C_2+0.25*C_1$$

即，最后画的物体颜色被加强，它比其他物体的颜色要深。如图5-19右图所示，红色方形是最后画的。另一方面，如果物体3在物体2之前画，物体2在物体1之前画，则最终颜色为：

$$0.5*C_1+0.25*C_2+0.25*C_3$$

因此，是画物体的时间次序而非空间位置决定了最终的显示颜色。

这就是混合颜色与透明颜色的区别，对于透明色模型，物体叠放的次序是无关的，每个物体都减去光线中的某个颜色成分，与物体叠放的次序是无关的。为此，创建由多个透明色物体构成的颜色是比较复杂的。

图5-10所示这里给出的问题及其结果，最后画的物体不一定是最靠近眼睛的。这种带混合的透明模式一般是从后到前画图，对于事实上的半透明效果，确实离眼睛近的物体对最终颜色的贡献比离眼睛远的物体大。如果以由远到近的次序画图，则用颜色混合来创建透明色比较好，参见本章的后面部分讲述OpenGL的程序例子。

5.2.9 索引颜色

在有些系统中，图像缓冲器不够大，不足以为每个像素提供三字节。这在20世纪90年代后期的系统中很常见。这些系统也是图形API支持的。这些系统使用索引色，帧缓冲器中为每个像素保存一个整数值，该值是RGB颜色数组（称为颜色表）中的索引值。一般情况下，该整数值是无符号8位字节，系统中可用的是256种颜色，用户在使用时，首先必须定义颜色表。

当场景中的颜色很多时（例如真实感对象）使用索引色就比较困难。场景中使用的颜色模型必须是RGB，必须仔细研究RGB场景中使用的颜色，看哪256种是场景中使用最多的或最接近于场景中使用的颜色。从许多传统的图形文章中都可以找到这种算法，通过这些算法分

析得到颜色表。

除了使用颜色表项而增加的计算量之外，带颜色索引的系统对颜色走样问题也比较敏感。马赫带就是一种颜色走样，科学计算中使用伪彩色造成的颜色差异也是一种颜色走样。 192

5.3 颜色和视觉交流

与计算机图形进行有效的视觉交流时颜色是最重要的工具之一，它使图像更丰富，更吸引人。许多工具可以表现信息给用户看，还可以表示图像之外的其他信息。但是，如果使用不正确，颜色也会破坏图像，所以必须仔细使用。使用颜色的目的是使观察者更清楚地了解图像的意义，即使图像本身是非自然的。本节将描述许多使用颜色的方法，以便于用户理解颜色的意义，以及使用颜色进行有效的视觉交流。

使用颜色必须仔细考虑颜色代表的含义。在表示人工图像与图像所描述的含义时，颜色是很关键的因素。当然还有一些其他因素。最重要的策略之一是表示与图像相关的某个特征值。例如，对于某种太空图像（例如星系太空），颜色值可表示太空中的某些物质（如某些天体发出的气体，颜色值可表示气体的速度或温度）。如果图像是机翼，则颜色可表示机翼上每一点的气压。颜色也可以本身没什么意义，但用于支持其他表示（例如前面提到的颜色深度显示）。但是不要在未理解颜色表示的意义之前使用颜色。在下面几节中将讨论使用颜色的几种方式。

5.3.1 强调色

图像中包含想要展示给用户的所有信息，包括模型的结构和其他细节。但是设计图像时，有可能要让用户的注意力集中到某些关键特征上。

将用户的注意力吸引到图像的某个特征上有许多方法。其中一个有效的技术是为该特征使用强烈的对比色，这种颜色比较鲜明，在场景的其他颜色中比较突出。使用这类强调色要做两件事：用哑色（muted color）设计场景，亮度可以从中突出，选择与哑色的对比色作为强调色，这样颜色比较突出。

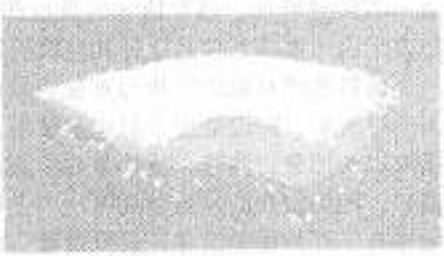


图5-11 有一个强化色的图像。参见彩图

图5-11是由一系列控制点定义的自由曲面，其中一个控制点高亮显示。背景、上下表面以及标准控制点都使用低饱和度的颜色，但是，特别标出的强化色控制点用红色，在灰度图中显示为黑色。正确的显示参见彩图。

5.3.2 背景色

除了高亮显示一些对象外，图像中还包含许多其他信息，如高亮显示背景色。背景色应该是不太引起用户注目的颜色。一般情况下，好的背景色一般较暗或者是中性色，但黑色并不是很好的选择，因为任何较暗的颜色都会淹没在黑色中。白色是较好的背景色，因为白色是中性色，但是与黑色一样，当对象较淡时也较难从白背景中显现出来。

表示图像时使用一种颜色的背景并不是一种好方法。职业的摄影师和照相师都不使用单色背景。他们使用中度强化色作为背景，可以使用户注意到主题。在背景的中央使用较亮的聚光，或者在背景中使用较亮的闪光灯（通常从左下角射到右上角），使用户的注意力吸引到 193
关键部分。图形API中没有直接提供这种背景，但是可以将纹理图贴在模型后面产生这种背景。来自于照片与视频的这种思想也可以用于计算机图形学中，参见图5-12。



194 图5-12 图像强化显示（例如偏振光束），使用聚光灯背景

5.3.3 自然色

对于真实对象的图像，关键是以逼真的形式显示出来。用户可以使用API构模工具，光照及阴影工具来创建相应的彩色图像。用户还可使用纹理映射产生真实感颜色显示。构模在第2章和第3章介绍，光照和阴影在第6章介绍，纹理映射在第8章介绍，因此，这里就不一一详述了。但是，图形API在真实感颜色方面还有一些限制，因为大多数情况下，只是使用自然色的近似值。这是一个非常有前景的研究课题。

5.3.4 伪彩色和颜色渐变

如果把颜色当作对象的一个单独特征，则颜色可用于承载图像的额外信息。颜色的一个重要用途是表示要显示的对象的特征，该特征可以是温度、速度、距离，甚至是可以用数值表示的任意特征。该数值先转换成颜色，然后再与对象显示时同时显示出来。我们在本书中把表示除图像颜色信息之外其他值的颜色称为伪彩色。

伪彩色将对象的形状与对象的颜色区分开来。如图5-13所示，左边是房子的正常显示，右边显示该房子发出的热量。注意，发热图片中强化色在于房子的窗户，因为窗户的密封性远远比不上墙。本例中的图像来自于红外图像而非计算机图像，但其本质是显示特征，同时也给出伪彩色图像的形状——该图片在物理温度特征的伪彩色使用方面是一个很好的例子。



图5-13 正常房子图像（左），红外图像（右）[MCC]

用颜色表示数值可以通过颜色渐变来实现，颜色渐变是一维颜色序列，0.0与1.0之间的数值都可以用颜色来表示。颜色渐变将颜色与数值相关

195 联，因此必须仔细选择颜色才能帮助用户理解用颜色表示的数值。颜色渐变中使用的颜色必须是要显示领域中比较熟悉的。颜色渐变可使颜色平滑变化，也可使颜色有较明显的边界。对于特定的应用，如何用不同的渐变实验创建颜色与数值的关系是一门艺术。

5.3.5 创建颜色渐变

创建颜色渐变非常简单，并且独立于图形API。下面的代码例子说明两个颜色渐变是如何创建的。假设数值变化到某一范围，返回数组有三个值，分别代表与该值有关的RGB颜色。该程序代码假设API使用RGB颜色模型。第一个颜色渐变提供彩虹序列的颜色，分别为红－橙－黄－绿－蓝－紫。第二个颜色渐变的颜色只有亮度的变化，从0.0均匀地变化到1.0。单亮度颜色渐变从黑色变化到红、到黄、再到白色。本章的练习题中给出了其他单亮度颜色渐变序列。必须注意，这些函数的区别只在于全局实数变量myColor[3]的不同。本章的后面部分还

会给出一些颜色渐变的例子。

```
//  彩虹颜色渐变
void calcRainbow(float yval)
{  if (yval < 0.2)                                  // 从紫到蓝的渐变
      {myColor[0]=0.5*(1.0-yval/0.2);myColor[1]=0.0;
       myColor[2]=0.5+(0.5*yval/0.2);return;}
   if ((yval >= 0.2) && (yval < 0.40))  // 从蓝到浅蓝色渐变
      {myColor[0]=0.0;myColor[1]=(yval-0.2)*5.0;myColor[2]=1.0;return;}
   if ((yval >= 0.40) && (yval < 0.6))  // 从浅蓝到绿的渐变
      {myColor[0]=0.0;myColor[1]=1.0;myColor[2]=(0.6-yval)*5.0;return;}
   if ((yval >= 0.6) && (yval < 0.8))           // 从绿到黄的渐变
      {myColor[0]=(yval-0.6)*5.0;myColor[1]=1.0;myColor[2]=0.0;return;}
   if (yval >= 0.8)                             // 从黄到红的渐变
      {myColor[0]=1.0;myColor[1]=(1.0-yval)*5.0;myColor[2]=0.0;}
   return;
}
// 基于照明颜色渐变
void calcLuminance(float yval)
{  if (yval < 0.30)
      {myColor[0]=yval/0.3;myColor[1]=0.0;myColor[2]=0.0;return;}
   if ((yval>=0.30) && (yval < 0.89))
      {myColor[0]=1.0;myColor[1]=(yval-0.3)/0.59;myColor[2]=0.0;return;}
   if (yval>=0.89)
      {myColor[0]=1.0;myColor[1]=1.0;myColor[2]=(yval-0.89)/0.11;}
   return;
}
```

196

图5-13中的红外图像中的颜色渐变与其他的有所不同。在原始彩色图像中，数值从0到13.9（摄氏度），颜色从黑色（最小值）到深蓝到桃红、到红到黄、再到白色。对颜色作划分，0～6表示从黑色到蓝色的渐变；6～9，表示蓝色减少，红色增加，然后，9～12，表示红色减少，黄色增加，最后，12～13.9，表示黄色减少，蓝色增加。这里数值较小时用蓝色，是根据习惯，较冷的物体用蓝色，温度逐渐增加时红色逐渐增加，最后通过加入绿色，红色逐渐变成黄色，然后通过加入蓝色，黄色逐渐变成白色。因为金属温度增加时，颜色从红色变成黄色，最后变成白色（即“白热化”）。因此，红外图像颜色渐变使用与习惯相似的温度表示方法。

5.3.6 颜色渐变的使用

颜色渐变是一种空间，其值是颜色而非数值。它创建了0与1之间的数值与颜色之间的关系。使用颜色是为了使数值可见，因此，可以通过颜色渐变将某一数值范围的对象显示出来。颜色渐变可以作为绝对颜色，也可以作为材质颜色，结合光照进行显示。一旦颜色被置为数值，则可用任意颜色模型处理它。



图5-14 静电势能表面模型，用“彩虹”颜色渐变加强极值（左）和统一亮度分布（右），参见彩图

在前面讨论几何形体时，给出了形状或颜色如何压缩信息的例子。该例子使用简单的蓝色红色颜色变化，而非显式的颜色渐变。根据前面提到的Coulomb法则结合颜色和形状，长方形上的静电能使用颜色渐变，如图5-14所示（参见彩图）。左边图像显示彩虹颜色渐变，右边图像显示统一亮度颜色渐变。仔细观察图像，看是否能从颜色获得图像中每个点的势能的数值。在实际应用中，不能简单地显示彩色表面，必须在图像中同时显示关键点的数值。

197

值得提出的是，这些例子并不是使用颜色渐变的唯一方法。与其将几何表面和颜色渐变当作不同表示空间，不如将它们当作同一空间链接的不同显示。图5-15显示光照表面和2D伪彩色平面的结合。这里用到了彩虹颜色渐变。

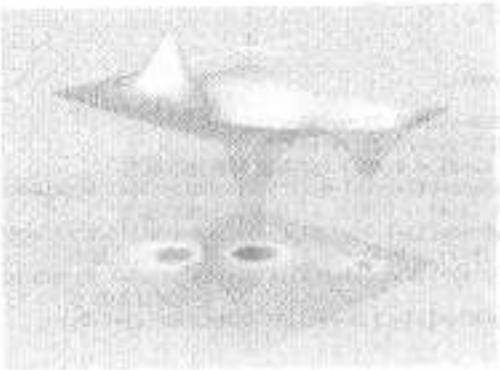


图5-15 带光和表面的伪彩色表面显示。参见彩图

5.3.7 比较形状和颜色编码

前面讨论过形状与颜色结合表现模型的问题。图5-16给出三种不同的表示温度的方式，用于第1章中热辐射的例子。图中给出三种表示温度的方式：使用几何和颜色编码（中），只使用几何编码（左上），只使用颜色编码（右下）。使用较宽的条带和较红的颜色表示较高的温度，反之，用较暗的条带和较蓝的颜色表示较低的温度。一个值表示一种信息，用不同的值以示区别。条带的高度也可表示信息的改变，这有利于某些不能理解颜色信息的用户。只用颜色编码表示温度对初学者有利。颜色与数值相结合则更有利于理解。编码信息的方式符合用户的习惯，这样更有助于理解。

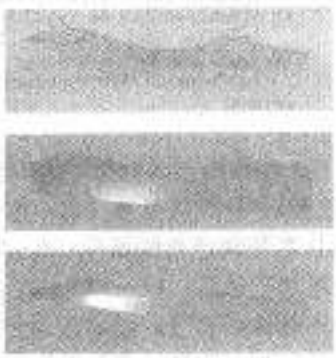


图5-16 同一信息（条带上的温度）的三种编码方式：只用几何值编码（左上），只用颜色编码（右下），二者都用（中）。参见彩图

5.3.8 颜色的文化背景

让用户理解所创建的图像，需要考虑其不同的文化背景。这是一个复杂的问题，涉及许
198 多不同的方面：职业背景、社会背景、地域背景及其他，如果用户必须学习理解一幅图像（例如，图像中的某些特征代表某个用户不理解的意义），则该图像的颜色应用是失败的。图像作者必须用易于人们理解的方式表达意思。

程序员必须了解用户的视觉认知，理解用户所处的文化背景中颜色信号的含义、图形表达的意义。例如，我们比较了解日本传统的空旷开放的设计思想，但是，现在日本文化更趋向于拥挤的、旗帜飘飘的日式网站，类似于日本报纸与杂志。当面向日本用户时，必须选择这两种方式中的一种来布局图像。

但是，大多数工作都是面向专业领域而非文化领域的，必须理解图像在物理学、化学、生物学或者工程方面所表达的概念，而非道德或宗教文化方面的概念。因此，必须了解物理学家、化学家、生物学家或工程师使用和理解图像的方式。在各领域使用颜色的方法有一些参考文献，如[THO]和[BR95]等。图5-17给出了在其他领域颜色所表达的意义。

那么，如何了解特定文化的视觉理解框架呢？可以通过阅读杂志、报纸或者专业出版物，从该类文化源获取一些图像常识，向熟悉该文化（比如道德、宗教或科学领域）的专业人士咨询。
199 同样，也可以开发标准结构、颜色机制或文化信息集，并通过专家检验。最终向用户提供合

适的图像集、标准结构、信息和颜色集。

颜色	过程控制工程师	财务管理人员	保健专家
蓝色	冷水	企业可靠	死亡
青绿	蒸汽	关和稳定	缺氧
绿色	正常安全	盈利	极感染
黄色	注意	重要	黄疸
红色	危险	亏损	传染
紫色	热辐射	富有	特注意

表5-17 专业领域的颜色使用

值得注意的是，颜色本身没什么意义，只是通过场景上下文才表现出来。场景上下文敏感的颜色会产生一些惊奇的效果。颜色在场景上下文中的描述参见参考文献[BR95]。图5-18给出了一个颜色与场景上下文关系的简单例子。该图中有一系列黑的格子，中间用灰度线隔开，在每个交点处画一亮白圆。从这张图中可以发现，这些点单独看是白色的，但合起来看却像是灰色的。这是为什么？

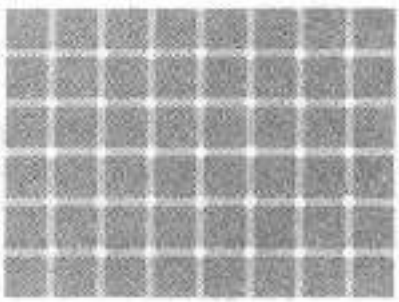


图5-18 神秘的黑色小球

200

5.4 例子

例：带半透明面的对象

绘制透明塑料面的一个标准平面时，可以透过每个坐标平面看到其他平面。本例给出的模型是用RGB基元色绘三个正方形，每个正方形都放在坐标平面上，中心在原点，α值为0.5，因此，可以透过一个正方形看到其他正方形。本节构模透明几何体，观察不同任选项下的效果。

图5-19的左图显示带深度测试的颜色显示。“透视”依赖于绘制物体的次序。在深度测试可行的情况下，如果靠近视点的物体是透明的，则远离视点的物体就不必与以前绘制的靠近视点的物体进行颜色混合。尽管第一个坐标平面是半透明的，但绘制时则体现出它对其他平面的完全不透明性，可以透过第二个坐标平面看到第一个坐标平面，但是对第三个平面则是完全不透明的。首先绘制蓝色的平面，它只对背景透明（该平面比原有的颜色暗一点，因为黑背景透出来了）。绿色平面被第二个绘制出来，只有蓝色平面或背景可透出来。第三次绘制红色平面，背景和其他两个平面都可以透出来。本例中，用户可以用键盘对平面进行旋转，在旋转中的任意位置正方形透明属性都是相同的，因为绘制次序不变。



图5-19 左图显示半透明坐标平面，中间的图显示坐标平面完全透明，但有相同的α值，右图显示有调节α值的同样的坐标平面。参见彩图

图5-19的中图关闭深度测试，其结果更像透明平面，但透明显示有点混乱，因为最后画的红色平面看上去像是画在最上面，因为它的颜色最亮。该图的结果表明OpenGL混合不等于透明，我们很难从该图中获得不同平面间的关系。除了可以使用颜色混合之外，该图的程序代码与上面图例完全一样。

201

在最后一幅图中，将三个正方形的α值取为不同的权重值，第一种颜色（蓝色）的α值为1.0，第二个颜色（绿色）的α值为0.5，红色的α值取为0.33。其结果如图5-19右图所示。各个区域的颜色权重如下：

- 三种颜色共享的区域都用α值0.33
- 蓝色和绿色共享的区域都用α值0.5
- 红色和绿色共享的区域都用α值0.33
- 红色和蓝色共享的区域中红色用α值0.33，蓝色用α值0.67
- 只有一种颜色的区域用它们各自初始的α值

原始的α值给出的是一个不透明的蓝色，半透明的绿色，比较透明的红色。修改后的颜色可以大致表示真实物体的透明属性，特别注意三色覆盖的灰色部分，仍有一些颜色没有表现出来。为了达到更好的效果，必须仔细分析绘制过程，即使如此也不能完全表示真实物体的透明属性。



图5-20 平面一分为四后的半透明效果，从后向前绘制。参见彩图

再考虑一下这个例子中的深度排序。三个平面相交，每个平面必须分成四个平面才能保证不相互交叠。如果没有了交叠，就可能使用与视点的距离进行排序。按从后到前的次序画图，颜色混合可以产生较好的透明效果。图5-20给出了绘制效果。这种调整技术并不都是有效的，因为有时剖分表面中的物体比较困难，但是对于这个例子是有效的。

对于深度优先绘制的另一个问题是：如果绘制的场景可以旋转，则有些部分不再最靠近于视点。这时，需要用到由图形API的特点标定物体与视点的距离。达到这个效果有三种方法。

- 函数与参数glGet(GL_CURRENT_RASTER_DISTANCE)返回视点与当前光栅位置的距离。
- 函数gluProject()与参数（原始模型顶点坐标），获得窗口中顶点的z值作为与物体的距离。
- 使用第4章提到的空间剖分技术进行物体排序。

以上三种方法都可以获得一个子物体到眼睛的距离，可以在绘图之前进行排序，在绘制
202 图像的过程中必须随时调整物体的次序。

正如在图中看到的，虽然每个平面的α值都是0.5，绿色和蓝色之间的亮度差仍然很明显，因为绿色平面在前面时，与蓝色（或红色）平面在前面看上去不同。不同颜色的亮度差在前面已讨论过了。

5.5 OpenGL中的颜色

OpenGL使用RGB和RGBA颜色类型，分量值都为实数。这些颜色与前面提到的RGB模型很类似，只需要修改小部分内容。后面将讨论OpenGL中的颜色混合，并给出一些使用颜色效果的程序例子。

5.5.1 颜色定义

OpenGL中用glColor*(...)函数说明颜色。该函数中有一些参数，比如大小、数据类型、数据以标量还是向量形式出现。具体函数例如：

```
glColor3f(r, g, b)   三个实数标量颜色参数，或
glColor4fv(V)        一向量V的四个实数颜色参数
```

可以以标量或向量形式定义RGB或RGBA颜色，在本书的其他部分可以看到许多颜色定义的例子。

5.5.2 使用混合

使用RGBA颜色模型时，用户必须说明是否要颜色混合，用户还必须指定颜色缓冲区中的

颜色与新对象颜色的混合方式，这可用以下两个简单函数调用实现：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

第一个是OpenGL一般函数，可用于许多地方，让用户选择绘制流水线中更有效的方式。第二个函数说明每个像素已有颜色与新对象颜色的混合方式，在模拟透明度时混合是很常见的。如果对象的α值是0.7，则70%的颜色取自新对象，30%的颜色取自该像素原有的颜色。

OpenGL混合函数有许多任选项，混合函数的说明格式如下： 203

```
glBlendFunc(src, dest)
```

源（src）和目标（dest）可以取许多值，OpenGL手册中有详细说明。

5.6 代码实例

5.6.1 带有全色谱的模型

本程序例画出RGB立方体的边，使用平移和变比创建一些小立方体并形成多条边。在这个例子中，立方体函数的参数有位元等，并通过位置设置颜色。

```
#define NUMSTEPS 20
#define SIZE 20.0
typedef GLfloat color[4];

void cube(float r, float g, float b) {
    color cubeclr;
    ...
    cubeclr[0]=r; cubeclr[1]=g; cubeclr[2]=b; cubeclr[3]=1.0;
    glColor4fv(cubecolor);
    glBegin(GL_QUADS);
        ...
    glEnd();
}

void ribboncube() {
    ...
    for (i=0; i<=NUMSTEPS; i++) {      // 红色部分之
        glPushMatrix();
        glScalef(scale,scale,scale);
        glTranslatef( SIZE+(float)i*2.0*scale*SIZE,SIZE,SIZE);
        cube((float)i/(float)NUMSTEPS,1.0,1.0);
        glPopMatrix();
    }
}
```

通过变换堆栈技术将当前模型变换压入变换栈中，应用平移和变比画立方体，然后弹出变换栈，恢复以前的模型变换。

5.6.2 HSV圆锥

程序显示颜色模型所使用的函数有两种。第一种是从HSV颜色转换成RGB颜色（参见参考文献[FO1]），该方法基于HSV圆锥与RGB立方体的几何关系，将立方体对角方向当作锥的中心轴。第二个函数以一般在HSV中定义的颜色绘制圆锥，并转换成RGB，同时在着色被执行颜色平滑处理。每一顶点的颜色在坐标之前给出，并产生如图5-4所示的平滑显示效果（参见彩页），有关平滑处理的详细程序参见第6章。 204

```
void
convertHSV2RGB(float h,float s,float v,float *r,float *g,float *b)
{
// 转换例子      foley et.al., fig. 13.34, p. 593
float f, p, q, t;
int k;
```

```
if (s == 0.0) {   // 无色情况
    *r = *g = *b = v;
}
else {   // 彩色情况
        if (h == 360.0) h=0.0;
        h = h/60.0;
        k = (int)h;
        f = h - (float)k;
        p = v * (1.0 - s);
        q = v * (1.0 - (s * f));
        t = v * (1.0 - (s * (1.0 - f)));
        switch (k) {
            case 0: *r = v; *g = t; *b = p; break;
            case 1: *r = q; *g = v; *b = p; break;
            case 2: *r = p; *g = v; *b = t; break;
            case 3: *r = p; *g = q; *b = v; break;
            case 4: *r = t; *g = p; *b = v; break;
            case 5: *r = v; *g = p; *b = q; break;
        }
    }
}
void HSV(void)
{
    #define NSTEPS 36
    #define steps (float)NSTEPS
    #define TWOPI (2.*M_PI)

    int i;
    float r, g, b;

    glBegin(GL_TRIANGLE_FAN);    // HSV圆锥
        glColor3f(0.0, 0.0, 0.0);
        glVertex3f(0.0, 0.0, -2.0);
        for (i=0; i<=NSTEPS; i++) {
            convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
            glColor3f(r, g, b);
            glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                2.0*sin(TWOPI*(float)i/steps),2.0);
        }
    glEnd();
    glBegin(GL_TRIANGLE_FAN);    // HSV圆锥的顶面
        glColor3f(1.0, 1.0, 1.0);
        glVertex3f(0.0, 0.0, 2.0);
        for (i=0; i<=NSTEPS; i++) {
            convert(360.0*(float)i/steps, 1.0, 1.0, &r, &g, &b);
            glColor3f(r, g, b);
            glVertex3f(2.0*cos(TWOPI*(float)i/steps),
                2.0*sin(TWOPI*(float)i/steps),2.0);
        }
    glEnd();
}
```

205

5.6.3 HLS双圆锥

显示双圆锥程序与HSV模型显示非常相似，这里就不列出了。模型转换的源代码也取自Foley的著作，这里给出的代码产生如图5-5所示的效果（参见彩图）。

```
void
convertHLS2RGB(float h,float l,float s,float *r,float *g,float *b)
{
// 转换来自      Foley et.al., Figure 13.37, page 596
        float m1, m2;

        if (l <= 0.5) m2 = l*(1.0+s);
        else     m2 = l + s - l*s;
        m1 = 2.0*l - m2;
        if (s == 0.0) {   // 单色情况
            *r = *g = *b = l;
        }
        else {   // 彩色情况
            *r = value(m1, m2, h+120.0);
            *g = value(m1, m2, h);
```

```
            *b = value(m1, m2, h-120.0);
        }
    }
    float value(float n1, float n2, float hue) {
    // HLS到RGB转换的帮助函数
        if (hue > 360.0) hue -= 360.0;
        if (hue < 0.0) hue += 360.0;
        if (hue < 60.0) return(n1 + (n2 - n1)*hue/60.0);
        if (hue < 180.0) return(n2);
        if (hue < 240.0) return(n1 + (n2 - n1)*(240.0 - hue)/60.0);
        return(n1);
    }
```

5.6.4 带半透明面的对象

图5-19的绘制工作很简单，但有几点需要注意。这三个正方形的颜色由以下代码给出：

```
GLfloat     color0[]={1.0, 0.0, 0.0, 0.5}, // R
            color1[]={0.0, 1.0, 0.0, 0.5}, // G
            color2[]={0.0, 0.0, 1.0, 0.5}; // B
```

206

这是全饱和红色、绿色和蓝色，α值为0.5。因此，每个正方形用50%的背景色与50%的正方形前景色绘制出来。左图给出颜色混合的效果。

每个面的几何信息由顶点数组给出，每个顶点是一个实数数组：

```
typedef GLfloat point3[3];
point3 plane0[4]= {{-1.0, 0.0, -1.0}, // X-Z 平面
                   {-1.0, 0.0, 1.0},
                   { 1.0, 0.0, 1.0},
                   { 1.0, 0.0, -1.0} };
```

从上面的例子可以看到，每部分的颜色都是分别回的。事实上同种颜色的部分并不需要回多次。在颜色修改之前，画的内容都用同一种颜色。

```
glColor4fv(color0);     // 红色
glBegin(GL_QUADS);      // X-Z 平面
    glVertex3fv(plane0[0]);
    glVertex3fv(plane0[1]);
    glVertex3fv(plane0[2]);
    glVertex3fv(plane0[3]);
glEnd();
```

为了将它扩展成图5-20所示的从后往前的绘制方式，首先需要将三个正方形剖分成4部分，每个部分不相交。然后安排这十二个部分的绘制次序，使离眼睛最远的部分先画。对于静态图，这样做很简单。但是，对于对象或视点运动的动态图，必须做一些实时的深度计算。在OpenGL中使用以下函数

```
GLint gluProject(objX,objY,objZ,model,proj,view,winX,winY,winZ)
```

其中，objX、objY和objZ是模型空间的顶点坐标，类型为GLdouble，model和proj是const GLdouble*变量，表示当前模型和投影矩阵（由glGetDoublev调用返回），view是Glint*变量，表示当前视点（由函数glGetIntegerv调用返回），winX、winY和winZ是GLdouble *变量，返回投影到3D视点空间后的顶点坐标。

有了这些信息就可以确定场景中每一成分的深度，并根据深度从后到前的绘制序列。如果数据是结构数组，则根据这些序列可以从后到前绘制场景中的对象。

5.6.5 索引颜色

除了本章讨论过的RGB和RGBA颜色模式，OpenGL还可处理索引色，但在这里不详细介绍了，因为索引色原理很简单，且较难生成高质量的图像。如果用户的系统只支持索引色，建议参考OpenGL手册以获到更多的信息。

207

5.6.6 OpenGL中的颜色渐变

因为颜色渐变可用颜色表示一个数值，首先计算出与数值对应的颜色渐变，再将该颜色的位置绘顶点或对象。使用本章前面部分提到的颜色渐变的概念，首先用calcRamp(float)计算全局变量myColor[]的RGB成分，然后设置材料颜色或纯色给渐变函数myColor。以下代码为一角形网格置纯色，该颜色由一角形三个顶点的平均高度来决定。

```
for (i=0; i<XSIZE-1; i++)
    for (j=0; j<YSIZE-1; j++){
        // 构造形的第一个三角形
        glBegin(GL_POLYGON);
            zavg = (height[i][j]+height[i+1][j]+height[i+1][j+1])/3.0;
            calcRamp((zavg-ZMIN)/ZRANGE);
            glColor3f(myColor[0],myColor[1],myColor[2]);
            // 给出三角形的坐标
        glEnd();
        // 构造形的第二个三角形
        glBegin(GL_POLYGON);
            zavg = (height[i][j]+height[i][j+1]+height[i+1][j+1])/3.0;
            calcRamp((zavg-ZMIN)/ZRANGE);
            glColor3f(myColor[0],myColor[1],myColor[2]);
            //给出三角形的坐标
        glEnd();
    }
```

5.7 小结

本章是关于颜色的一般讨论，包括RGB、RGBA、HLS和HSV颜色模式。同时还模拟带α通道的颜色混合函数的透明度，并且提出一种用颜色生成3D图像的方法。最后，我们给出OpenGL中颜色说明和实现的方式。这样就可以在图像中简单地使用颜色产生比几何形状更有趣的效果。

5.8 本章的OpenGL术语表

与其他图形API一样，OpenGL也有定义颜色的函数。这里给出了一些函数便于读者参考。这里给出的比较少，大多数在实照编程处说明，按图索骥，前面讲过的这里就不一一赘述了。

OpenGL函数

glColor*()：系统用于定义颜色的函数族，系统用该函数定义当前颜色画图，直到使用该函数改变颜色。颜色可以有三个分量或四个分量，可声明为标量或数组。

2018

glBlendFunc(const, const)：说明源(第一个形参)和目标(第二个形参)是否带变比因子来确定颜色混合。

常量

GL_BLEND：说明是否与颜色混合，由函数glEnable()和glDisable()使用。

GL_ONE_MINUS_SRC_ALPHA：函数glBlendFunc()使用的变比因子，说明颜色是否乘上$1-\alpha_{source}$。

GL_SRC_ALPHA：函数glBlendFunc()使用的变比因子，说明颜色是否乘上α_{source}。

5.9 思考题

1. 对于真实场景的数字图片或扫描图像，观察它们与其灰度版本的区别（例如，在单色打印机上打印的照片，单色显示器上显示的图像，用Photoshop单色处理后的图片）。在彩色图像中最重要的是哪些部分？在单色图像上呢？

2. 用较好的放大设备（如放大镜）观察标准颜色印刷，检查每个CMYK颜色点，看每种颜色点是以什么图案与式给出的？如何使它看上去像真实的颜色？
3. 计算机系统可以显示不同深度的颜色（常见的是256色、4096色和16.7M色）。在屏幕上显示一幅图像，修改颜色深度，看看它有什么变化。对自然图像和人工图片都做这个工作，特别注意是否会看到马赛克。
4. 本章讨论了多种颜色渐变方式。有些颜色渐变具有特别的科学含义，比如用颜色渐变值表示气温的变化。虽然气温数据是平滑的但颜色渐变值是离散的，可以将颜色值显示到图例上，在某个气温上显示特别的颜色值，可以使用户引起注意。
5. 在上面的思考题中提出了颜色渐变的科学含义，其实颜色渐变还有其他含义。注意日常使用颜色渐变的地方，例如地图上的高度线，气象预报上的温度线，新闻报导上的警戒线。请读者举例说明其他使用颜色渐变的例子。
6. 我们已经讨论过使用带光照的表面来表示某个函数对一个区域的影响，以及颜色渐变如何决定表面的颜色。请读者比较伪彩色表面中每个顶点的颜色和光照表面的信息，看看两者有什么区别。让一个表面同时具有伪彩色和光照信息是否有必要？读者对实现两者的方法有什么考虑？ 209
7. 从一幅伪彩色图像的颜色中确定一个数值容易吗？如果一幅灰度图像具有颜色渐变和数值条例来表示两者之间的关系，读者可通过某个颜色值与数值匹配吗？怎么做可使两者间的匹配关系变得很简单？

5.10 练习题

1. 绘出一些正方形，其亮度相同，颜色不同。在单色显示器上并列显示时，看看是不是还能区分它们？为什么？
2. 在OpenGL中创建RGB颜色立方体，表面都是四边形，颜色在顶点处混合。从不同点观察该RGB立方体，看不同面的效果。对顶点进行变比修改整个RGB立方体，使每个顶点的最大颜色值为C，C值在0与1之间，C值的大小由读者自己决定。找出其他观察RGB立方体子空间的方法。
3. 许多计算机或应用程序都有“颜色拾取器”，允许用户选择一种RGB，HSV或HLS颜色。请读者选择一种或二种特定颜色，使用颜色拾取器，找到它在RGB，HSV和HLS中的颜色坐标。然后手工执行本章中的颜色变换函数，验证HSV，HLS颜色变换到RGB颜色的方法。
4. 使用前一练习题中提到的颜色拾取器，对某种颜色找到其RGB值，比较该颜色（例如物理对象的颜色）与颜色拾取器给出的颜色的异同。先给出初始颜色估计，然后一点一点调整RGB成分，以达到最好的颜色匹配。
5. 回到本章的开始部分如图0-7所示的温度分布例子，该图像包含温度的颜色渐变。关于这个例子，创建另一种颜色渐变，替换简单的红—蓝温度颜色渐变。检查读者用的颜色渐变是否比原来的好。
6. 创建两个非常接近的表面，它们的交角很小，但是颜色差别很大。看它们在相交处是否有z值冲突。接着对于两个共面的表面，只有很小的平移，看其z值的冲突情况。从这个例子中读者仔细体会颜色对于分辨几何体有什么帮助。
7. （教室项目）Delphi方法可用于估算数值问题。比如，多人给出不同的答案，则最终答案取平均值。对于练习4，教室内的每个人都估算一个对象的RGB颜色，对它取平均，则它与练习4给出的答案有多接近？ 210
8. （画家算法）对于只包含一些多边形的图像，不使用含隐藏面的深度缓冲，而使用画家算法：从远到近画多边形。请读者创建一个多边形序列，根据与摄像的距离，从远到近排序（使用OpenGL函数gluProject(...)），并按照实大小的次序画多边形。
9. 前面提到只包含亮度的颜色渐变。下面要求从黑色—红色—黄色—白色进行颜色渐变。请比较两种颜色渐变的区别和感受。

5.11 实验题

1. （颜色混合）用不同物体的颜色和不同α值，在不同对象上绘制2到3个半面对象，验证α值在模拟相等颜色对最后图像的作用，同时修改不同对象的排放次序，看α值必须如何修改才能保持相等的颜色仿真效果。
2. 本章中我们提到HTDV的亮度公式与一般监视器的亮度公式是不一样的。对于这两种不同的公式做练习题1的工作，比较它们的区别，看看在读者的系统中哪种公式效果较好。
3. 取已有的图像，通过以下方式获得单色图像：将颜色缓冲器中的数据读入RGB数组，计算亮度值，并用亮度值替换R、G和B的每个成分。将新图像读入颜色缓冲器并显示，效果如何？它与原始图像在单色显示器上的效果有什么不同？与Photoshop上转换成的单色图像又有什么不同？
4. 用前文讨论过的黑－红－黄－白颜色渐变创建一个图像，实现其伪彩色，然后转换成其他颜色渐变，讨论本章不同于练习题9的其他情况，看看效果如何？你会发现改变颜色渐变会影响对整幅图像的感觉。
5. 对于如表6-7所示的热传导程序，使用简介部分提到的代码，修改温度与颜色转换函数setColor()，使其使用不同的颜色渐变，检查不同的颜色渐变对温度棒上的颜色表示有什么变化。
6. 通过修改图像中的颜色来模仿色弱。例如，仿真红－绿色盲（这是最常见的色弱之一），修改所有的颜色，使红色值和绿色值都等于原来值的平均。（也可以通过修改颜色渐变计算达到这个目的，还可
211 以修改光线和对象的颜色，在光照表面上完成这个工作。）

5.12 大型作业

1. （小房子）对于前面提到的小房子，修改墙壁颜色，使其包含α值，同时使用颜色混合，使用户在外面就可以看到屋子的结构。
2. （场景图剖析器）修改第2章开始提出的场景图剖析器，让用户定义对象几何节点的颜色属性，并保存该属性。由于颜色是由系统维护的（直到它被修改），修改颜色的代码之前读取颜色，看看它的
212 效果如何。

第6章　光照处理和着色处理

本章介绍比颜色更进一步的、用来创建更具有吸引力和更加真实的图像技术。它主要是关于产生增强图像效果的两个方面：一个是基于简单模型的光照处理，主要是光线和表面的交互；另一个是基于简单模型的着色处理，主要是物体表面的颜色变化。光照处理模型主要基于光线的三个组成：间接光照、直接光照以及反射光照，介绍光线的概念和物体材质属性。着色处理模型主要是通过对多边形顶点的颜色进行平均来得到整个多边形内部的颜色。采用这些技术，可以使图像比单纯地采用颜色看起来更加真实。

为了更好地掌握本章的内容，应该在前一章的基础上理解颜色，并仔细观察周围世界中光和颜色的工作方式，以及理解多边形表示方式和对整个多边形进行平均的概念。

6.1　光照处理

通常在两种情况下可以看见物体：一种是物体本身固有的颜色。在计算机图形学中，这种情况相当于在RGB颜色空间里定义了一种颜色，然后告诉图形系统用那种颜色来绘制物体。这种方法很简单，并且很容易创建图像，因为只需要在定义和绘制几何体的时候设置颜色。事实上，当物体本身不具有颜色但需要通过颜色来表现一些信息或属性的时候，正如在前一章描述的，这是一个比较合适的方法。

然而，在现实世界中，并不是简单地看见有色物体，它还依赖于光照条件，所以必须考虑光在表现场景中的作用，因此，另一种看见物体的方式是周围的光线与物体进行物理上的交互作用后到达眼睛。光线会发射出能量，到达物体后，经过物体的反射，再到达我们的眼睛，它包含了光线的颜色以及物体的各种物理属性。在计算机图形学中，只考虑场景中光线的光照处理称为局部光照模型。通常，这种模型是指Phong光照模型，它把光照分为三部分并且被大多数的图形API支持，编程者只需要在场景中定义光源和物体的材质，就可以得到场景的颜色。本章主要讲述这种方法在光照模式下如何看见场景，计算机图形学中如何对模型做简化处理并计算得到场景中反映光线和材质关系的图像。

首先探究在局部光照模型中能否独立自然光照的模型。一个简单有效且被大部分API使用的方法是把光照分为三个基本组成：环境光、漫反射光和镜面反射光，它们的定义如下：

环境光：因为空间中的全局照明，所以场景中存在光线，它不依赖于任何具体光源，是场景中经过反射后光线的分布模型。

漫反射光：光线直接来自光源，到达物体后，与物体进行交互作用后直接到达观看者。根据物体的材质，物体反射的是光线波长的一部分，并由此产生物体的颜色。

镜面反射光：光线直接来自光源，到达物体后，不与物体进行交互直接到达观看者，因为不与物体进行交互作用，所以看到的颜色不是物体的而是光源的。

这三个组成部分的计算依赖于光源的属性，物体的材质属性以及光线和物体的几何关系。它们对整个光照计算都有作用，图形系统通过RGB分量来进行计算。三个组成部分的和是所看到物体的颜色。

通过定义光源的位置，颜色以及其他属性，计算机图形学可以建立光照模型。不同的图形API的光源属性可能有差别，但它们总是包括位置、颜色以及其他非必需的属性，这依赖于

具体的系统，光源可能是聚光灯，它有方向和上限的宽度，如果光源距离很远，那么就是平行光，由它发出的所有光线都是平行的，并且没有位置属性。光线的能量可能随着距离的增加以不同方式衰减。可以通过多种方式对光源的属性进行改变，来得到所要的场景效果。

正如为光源建立模型一样，也需要为物体的材质建立模型，以此来反映它们对光线的作用方式。每个物体都有属性，这些属性用来表现物体如何对光线的每个组成部分做出反应。环境光属性定义了物体对环境光部分呈现的颜色；漫反射光属性定义了对漫反射光部分呈现
214 的颜色；镜面反射光属性定义了对镜面反射光部分呈现的颜色。这个简单的光照处理模型假设物体对环境光部分和漫反射光部分的作用是一样的，并且对于镜面反射光部分，物体只是简单地呈现光线的颜色。如果计算是为了显示有光照的图像，那么光线和物体材质的属性会被分别对待。

所以，为了在场景中进行光照计算，必须定义一个或更多的光源，并指定它们的三个组成部分的颜色，同时，为每个物体定义材质属性。这跟简单地使用颜色有很大不同，它考虑更周到，修改起来也不费，不同的图形API有不同的方式来指定光源和材质，我们将在光照处理实现中讨论在OpenGL中是如何进行的。

6.1.1 环境光、漫反射光和镜面反射光

环境光没有源头，它只是简单地存在于场景中。场景中某些部分并不直接被光源照明，但它却存在亮度。全局环境光是因为场景中物体之间相互反射环境光引起的。单个光源对一个物体的环境光计算公式为$A = L_a * C_a$，C_a取决于物体的材质，L_a取决于场景中的环境光。L_a和C_a都是RGB三元组，并不是常量，计算产生的结果也是RGB值。如果有多个光源，那么总的环境光计算公式是$A = (L_0 + \sum L_a) * C_a$，$L_0$是全局环境光值，$\sum L_a$是指场景中所有光源的求和。所有的环境光都是近似值，可以一起计算。例如，如果强调直射光，那么应该把环境光系数设置相对小些（产生黑夜中或者阴暗小屋中光照的效果）。另外，如果想看清场景中的每个物体，并且光照强度看起来比较均匀，那么应该把环境光系数设置相对大些。如果想强调物体的形状，那么应该把环境光系数设置相对小些，因为这样着色处理会使表面产生变化，本节稍后会讲到。

漫反射光直接来自光源，被物体的表面反射，反射光根据物体材质的不同，波长也不同。OpenGL和其他API使用的漫反射光模型是基于每单元亮度或者光能量的思想。光线沿着它传播的方向在每个单元上都发射着一定的能量，当单位光照射到表面上的时候，表面的光强度与光线照射的表面大小成比例。在图6-1中，光所接触到的单位面积或者与光线方向垂直的单
215 位面积，大小是$1/\cos(\Theta)$。如果在光线方向上每单元的光能量是L_0，那么表面上每单元的光能量是$L_0 \cos(\Theta)$。当入射角减少的时候，单位光所照的表面减小。当光线和平面平行的时候，光强变为0。因为不可能存在“负光强”，所以，当余弦值是负的时候，就用0代替它，表示当表面背对光线时，漫反射光不起作用。

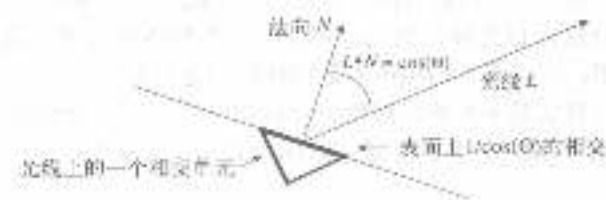


图6-1 漫反射光照

既然已经知道了单位表面上漫反射光能量的大小，那么它是怎么被眼睛看到的？漫反射光在物体表面的反射符合朗伯定理：

一个给定方向上反射光能量的大小与该方向和表面法向的夹角余弦成比例。

如果表面上每单元的漫反射光大小是D，从表面到眼睛的单位向量是E，那么从单位表面反射到眼睛的能量与$D^*\cos(\Theta) = D^*(E \cdot N)$成比例。眼睛看到的表面并不是单位表面，它看到的大小是$\cos(\Theta) = E \cdot N$，并且随着到眼睛向量角增大而减小。所以，感觉到的表面光强度大小与两个方面成比例关系，一个是接收到的光能量大小，另一个是眼睛看到的表面面积大小，而且这个接收到的光能量大小始终是D，不管眼睛放在哪里。这也意味着在漫反射光的计算中，视点的位置是没有关系的，这也就是为什么当我们绕着物体移动的时候，看到的颜色是不变的。

从以上讨论可以得出，漫反射光照的强度计算如下：

$$D = L_D{}^*C_D{}^*\cos(\Theta) = L_D{}^*C_D{}^*(L \cdot N)$$

L_D是每个光源的漫反射光分量，C_D是材质的环境光属性，$L \cdot N$这一项表明表面法向在漫反射光计算中的作用。如果有多个光源，那么漫反射光总计$D = \Sigma L_D{}^*C_D{}^*(L \cdot N)$，求和是对整个场景中的光源（每个光源的光向量$L$是不同的）进行的。在这个计算中，用点积代替了余弦，假设法向N和光向量L是单位向量。

直射光源与被照明的物体进行交互作用后，产生我们所看到的物体的颜色。物体并不反射所有的光线，相反，它吸收一定波长（或颜色）的光线并把其余的反射出去。我们看到的物体颜色，是它反射的光线，而不是吸收的光线，这种效果在我们定义材质的漫射属性的时候指定。

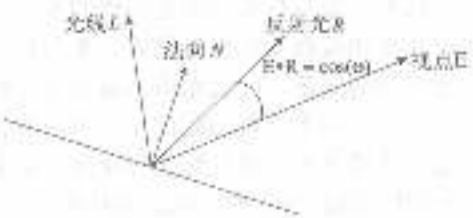


图6-2 镜面反射光

镜面反射光是个表面现象，它在发亮的表面产生非常明亮的区域，镜面反射光依赖于表面的平滑度和电磁属性，所以平滑的金属物体能更好地反射光线。镜面反射光的能量并不按照朗伯定理反射，它不与材质进行交互，而是直接反射，所以它的入射角和反射角相等，如图6-2所示。镜面反射光在它离开物体的时候有一些细微的“扩散”，所以镜面反射光的标准模型中允许定义物体的光泽度属性来模拟扩散。它通过指定光泽系数来模拟，随着这个值的增大，产生的镜面反射光越小，越明亮，使得材质看起来更明亮，如图6-3所示的三幅连续图片。在图4-6中，反射向量的计算公式是$R = 2(N \cdot L)N - L$，这些符号如图6-2所标注。 216

物体的镜面反射光计算公式是：

$$S = L_S{}^*C_S{}^*\cos^M(\Theta) = L_S{}^*C_S{}^*(E \cdot R)^M$$

L_S是光源的镜面反射光分量，C_S是物体的镜面反射光系数，M是光泽系数。同样，视线向量E和光向量L必须是单位向量。镜面反射光依赖于视线和反射光的夹角，它在物体表面以镜面方式反射，并且它随着这个角度的余弦值呈指数级衰减。镜面反射光描述了表面反光的程度，所以它的值越大，对象的反光程度越大。光泽度是个相对的概念，即使不反光的材质也可能有不规则的表面产生非常类似镜面反射的效果。随着光泽系数的增加，镜面反射光区域变得更小而且更集中——球体看起来更加光泽明亮。所以，光泽系数是材质的一个属性。这可以很好地建立物体的光泽度模型，因为相对大的M（例如，M接近或大于30），如果Θ的值很小，函数$\cos^M(\Theta)$的值接近于1。随着角度的增加，衰减值很快减小，随着指数值的增大，减小的速度也变快。



217 图6-3 分别用20、50和80（左、中、右）作为镜面反射光系数得到的镜面反射光效果

镜面反射光的计算对每个光源和每个物体分别进行。这个计算基本上依赖于物体到光源的方向和物体到眼睛的方向，所以，镜面反射光会随着物体、光源或者视点的移动而改变。

镜面反射光与物体的交互方式与漫反射光完全不同。假设镜面反射光在反射时并不被物体吸收，所以，镜面反射光的颜色与光源的颜色是一致的。只需要定义材质的镜面反射光颜色为白色，也可以为材质指定镜面反射光颜色，但为了更真实，这个颜色最好和物体的漫反射光颜色一样。

因为在漫反射光与镜面反射光的计算中都需要用到表面法向，所以，需要重温如何得到表面法向。一种方法是通过解析计算。对于球体，任意一点的法向是从球心到该点的连线，所以只需要知道球心和该点的值就可以计算法向。如果表面是由有两个自变量的连续函数表示的，那么，只要计算函数在某个点上的方向导数，并把导数进行叉乘，因为导数定义了表面的切平面，所以，法向和切平面垂直。如果不能进行解析计算，那么只能从多边形的坐标计算，这在第4章讨论数学基础的时候已经进行了描述。只需要把多边形两条边所指方向的向量进行叉乘。

知道了如何计算这三种光线分量的值，接下来来看光照计算中的常量。环境光的常量是环境光分量与材质环境光属性的乘积，分别对红色、绿色和蓝色光分量进行计算。同理，漫反射光和镜面反射光也是对应光与材质分量的乘积。所以，白色光与任何颜色的材质产生的是材质的颜色，红色光或者黄色光与红色的材质产生的是红色，但红色光与蓝色材质产生的是黑色，因为没有与蓝色材质对应的蓝色光，也没有与红色光对应的红色材质。每一点最终光强是环境光、漫反射光与镜面反射光的和，每一项都是由RGB分量计算得到。如果有任意一项的值超过1，那么它就被限制为1。

如果有多个光源，那么它们的效果是叠加的——环境光强是整个场景的全局环境光与每个单独光源环境光的和，漫反射光强是场景中所有光源漫反射光分量的和，镜面反射光强度是场景中所有光源镜面反射光分量的和。跟上面一样，如果这些求和的结果超过了1，那么它就被限制在1。

本章稍后将讨论着色处理，但现在所讨论的光照计算都是针对模型上一个点的。可以对多边形上的一个点或者每个顶点进行光照计算。如果只对多边形的一个点进行光照计算，那么这个多边形只能得到一种颜色，这就是*flat*着色处理。如果对每个顶点进行光照计算，那么就可以得到平滑着色处理，它可以使产生的图像看起来更真实。如果一个顶点是多个多边形的公共点，假如要计算这个顶点的法向，这个法向对这几个多边形都适用，那么可以通过解析计算的方法或者在这几个多边形的基础上，对它们的法向进行平均来得到该顶点的法向。每个顶点的颜色都将用于计算多边形中其他点的颜色。

这些光照计算方法都不处理阴影，因为阴影依赖于到达表面的光线，这与光线从表面反
218 射出去的方式有很大区别。阴影处理是非常困难的，在大部分的图形API中需要特别的编程处理。第8章将讨论基于纹理映射的简单阴影处理方法。

6.1.2 表面法向

在漫反射光和镜面反射光的计算中需要表面法向，通常可以在形状节点中的几何说明中

包含法向量。计算法向的过程包括对物体进行分析，可以精确计算通过解析方法定义的物体的法向。如果不能进行解析计算，可以用多边形的两条边进行叉乘，然而，只指定法向是不够的，这个法向还必须是单位法向，或者说法向长度是1（通常称为归一化向量）。对向量进行缩放需要计算，如果在定义几何体的时候这样做是不够的。因为对几何体的缩放或者其他变换会改变法向的长度。如果图形API说明所有的法向在使用前都是单位法向，那么这将是很有帮助的。

两个向量的叉乘产生另一个向量，这个向量和原来的两个向量都垂直。对于多边形，对一条边的两个顶点做减法，可以得到与原先边平行的向量。如果对两条相邻的边进行这种运算，就可以得到在这个多边形平面上的两个向量。所以，它们的叉乘得到的向量和这两条边都垂直。这个向量也和这个多边形垂直，由此就得到了多边形的法向。

正如在第2章所看到的，每个多边形都是两面的，称为前向面和背面。我们希望法向总是指向多面体的外面，法向指向的那一面称为前向面，这个差别在光照处理计算以及其他图形计算中非常重要。可以对多边形的前向面和背面分别定义材质。

6.2 材质

光照处理包括在场景中指定光源和物体与光线相关的属性。为了在场景中使用光照，需要指定以下两方面：光源的属性和物体的材质属性。本节将讨论材质的指定。实现光照处理包括把光源和材质整合在一起，正如本章最后的例子描述的那样。

场景中每一个物体都对反射光线有作用，这个光线决定它显示时的颜色。当我们讨论光源的三个组成成分时，同时引入了材质的四个属性：材质对环境光的反射率C_a，材质对漫反射光的反射率C_d，材质对镜面反射光的反射率C_s，镜面反射光的光泽系数M。三个反射率都包括颜色，它们通过RGB分量来指定，而光泽系数则是点乘$R \cdot E$的指数标量，这四个属性是由物体材质指定的，每个物体都有这些属性，这样系统才能进行光照计算。有些图形API可能把这些作为独立的一部分来定义，并被认为是形状节点中外观信息的一部分。API可能有其他材质特性，例如分别指定前向面和背面的材质属性，OpenGL就是这方面的例子，但不同的API，可能不同。

在光照处理的讨论中，假设物体只具有反射性，但物体也有可能有发射性。即它自身会 219
发光，发射的光只增加到物体本身的光强中，并不增加到场景中，这使得可以定义表示场景中诸如实际光源的现场。发射性可以通过定义材质具有发光属性来确定，最终光照计算在计算物体颜色的时候，会把发射光强累加到其他光强计算的结果上。

在第2章讨论场景图的时候，也讨论了形状节点的外观。在那里，我们注意到颜色和着色变化属于表现外观的因素，但材质同样是外观属性。当为每一个形状节点分别定义材质的时候（与定义颜色和着色变化一样），使用图形API提供的工具来为一组形状同时定义材质将会更加方便。然而，这取决于API如何在几何定义之间保留数据。这提醒我们可以在整个场景中采用变化的着色处理，并且允许以同样的方式在绘制的时候改变颜色或者材质。

6.3 光源属性

所有的光线都来自光源，所以，光源是建模工作中非常重要的部分。第2章介绍了如何在场景图中包含光源。每个光源都应该有位置属性，这是直接被场景图支持的，还需要其他方面的属性，本节将对此做论述。

图形API一般允许为光源定义一系列的属性。通常，包括它的位置或方向、它的颜色、它

随着距离增加的衰减方式以及它是全方位光还是聚光灯。我们将对这些属性进行描述，但并不深入，实际上位置和颜色属性是非常重要的，至于其他的属性，如果要实现一些特殊的效果，那么也是非常有用的。OpenGL中关于光源的细节将在本章的OpenGL部分讨论，本章最后的例子将展示如何使用位置和颜色属性。

6.3.1 光源颜色

光源颜色的值通常是一个以RGB颜色模型来表示的值。因为光照处理模型包括环境光、漫反射光和镜面反射光，一些图形API允许对这三个分量分别设置。可以参看API来得到使用的颜色和光照处理模型。

6.3.2 位置光

如果要使用定位在场景中的光源，应该给光源指定一个具体的位置。为了定义光源的位置，只要使用标准的模型坐标并指定正确的位置，正如在OpenGL部一节中所描述的。

6.3.3 聚光灯

位置光是朝着所有的方向发光的。假如要一种只朝固定方向发光的光源，那么只需要把光源定义成聚光灯就可以了。聚光灯不只包括位置属性，还有其他诸如方向、光锥角和角衰减系数。如图6-4所示。方向是个三维
220 向量，它与光源方向平行。光锥角是介于0.0与90.0之间的值，是聚光灯张角的一半，由它决定光线聚集还是分散（光锥角越小，光线越集中）。角衰减系数决定了聚光灯主轴与边之间光强度的衰减程度。



图6-4 聚光灯方向和光锥角

更详细一点，如果一个聚光灯的位置是P，角衰减系数是d，方向是D，并且光锥角是Θ，那么从聚光灯出发，沿着V方向上任意一点Q，如果点积$(Q-P)\cdot D$的绝对值比$\cos(\Theta)$大，那么该点的光能量是0，否则，它的光能量要乘以$((Q-P)\cdot D)^d$。

6.3.4 光线衰减

根据物理学原理，到达单位表面的光能量跟光源与表面的距离平方成反比。这称为光线衰减，计算机图形学中有多种方式来建立这种模型。一个比较精确的模型是让光线随着距光源的距离d的平方减弱，把它乘以k/d^2，图形系统会相应地减少光强度。但人类的感知系统更倾向于对数形式，而不是线性的，所以这种计算并不真实。因此，需要一种更慢的衰减。一般图形API会在光线衰减模型上提供一些选择。

6.3.5 方向光

如果光源在场景中有确定的位置，那么光照计算模型把光源到该点的方向作为光线方向。假如需要一种类似太阳的光源，场景中所有点的光线方向都是一样的，就像是光源在无穷远。如果光源位置坐标的第四个坐标分量是0，这在实际的坐标值中是不正确的，那么，该光源就是方向光源，坐标的前三个分量就是它的方向。所有的光照计算都以这个方向来计算。支持方向光的图形API通常有其方式来指定光源是方向光源(而不是位置光源)，并指定接收光线的
221 方向。

6.4 放置与移动光源

光源是场景中非常重要的元素，因为它可以表现物体的形状与轮廓。当光源的位置确定的时候，所有的变换都会影响光源。以下对场景图的内容概述将提醒我们模型中光源的问题：

- 如果光源在场景中的位置是确定的，它的几何体位于场景图的最上层。这样的光源与观看位置和其他模型都是无关的。
- 如果光源的位置是相对于视点确定的，那么它的几何体将被视图变换修改，但不会被随后的模型变换影响。如果采用视图变换是视点变换的逆变换这一约定，那么可以把光源指定在场景图中的视图分支来实现光源的这种位置。
- 如果光源的位置是相对于物体确定的，那么把它的几何体指定在定义物体的组节点分支上。任何影响物体的操作都将作用于对应的组节点上，并影响光源。
- 如果光源是在场景中到处移动的，那么它是场景图的独立节点内容，当节点确定的时候，它的几何体也确定了。

这种通常的概念就是位置光源的几何体是建模过程的另一部分，并且可以与管理其他物体一样管理它。光源的其他外观属性可以在任何允许的地方设置，也可以作为光源定义的一部分。

6.5 用光照实现特效

如果只考虑一个简单的场景，那么很自然的想法是为它建立一组光源来显示场景中所有的物体。但有时候需要为场景的不同部分使用不同的光照，那样可以通过突出物体的形状、亮度或者其他方面来强调它与场景中其他物体的不同。为了使用这种光照效果，需要在场景中定义一些光源，并根据显示场景的不同部分来打开和关闭某些光源。大部分场景绘制的时候使用一组标准的光源来建立整体概念，但为了绘制突出的物体，要先打开关键的光源，绘制后在再把光源关闭。通过这种方式也可以得到其他效果，例如要给多个物体一次一个特写，可以建立动画，对它们轮流突出显示。

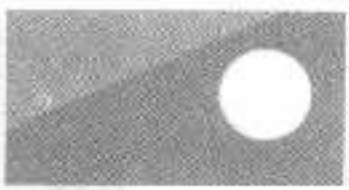


图6-5 在太阳和地球上放置不同光源的场景。参见彩图

图6-5展示了突出显示的简单例子。这是地球绕着太阳转的过程中产生春秋分的演示动画，地球被放置在太阳中心的光源照亮。太阳被放置在视点位置的光源照亮，这样就得到了想象中的自己发光球体。这个光照处理显示了地球是如何被真实地照亮的，包括强调太阳的直射中心是在北半球还是在南半球（图中描述的是夏季在南半球），并且太阳也被形象地表示，而且不需要通过刻意地去追求真实性。这个例子的代码包含在本书的附加资源中。 222

6.6 场景图中的光源

光源是作为场景一部分的图形物体，与其他图形物体一样放在场景图中。每个光源可以用形状节点来表示，包括几何位置以及方向，还包括外观数据：颜色、类型及其他参数。然而，启用和禁用光源功能并不属于光源形状节点的外观属性。因为每个物体都可以决定光源是否可以影响它。如果精心选择一个可以影响所有物体的光源，那么是为观看者高亮所有物体。可以把打开光源的列表作为表示场景中可见几何物体的形状节点的外观属性的一部分。

6.7 着色处理

着色处理是计算场景中各个元素颜色的过程，它基于物体表面的光照效果。着色处理过

程基于光线的物理特性，最细致的着色处理计算可以包括深入光线行为的细节，例如光线在拥有不同表面细节的各种材质上的发散。在这方面已经有了大量的研究，真正的真实感渲染必须考虑表面的众多细节。

大部分图形API没有能力处理这些细节化的计算。通常初级的API，如OpenGL只支持简单的局部光照模型。这在前面看到过，只有两种简单的多边形着色处理模型：Flat着色处理和平滑着色处理。Flat着色处理为每个多边形上单种颜色，平滑着色处理通过平均顶点的颜色来给多边形上多种颜色。两种模型都可以使用，但通常平滑着色处理更令人满意并且真实。除非有很好的理由使用Flat着色处理，例如，为了更加准确地表示数据或者不连续的概念，不然就应该使用平滑着色处理。我们将简单讨论其他更成熟的着色处理，虽然它们不被初级API支持。

6.8 在视觉交流中考虑着色处理

如果要绘制一幅表现真实物体并使用自然色彩的图像，那么应该尽可能地使图像看起来
223 真实。绘制实际的物体可以通过让物体看起来更真实来实现——虽然有时为了突出显示特殊属性而使用特殊的光照或者不真实的色彩。如果在场景里显示的物体并不表示实际的物体，尤其是使用一些合成的颜色来表现某个属性，那么应该考虑清楚是否应该在图像中使用光照处理和着色处理。

如果使用合成色，那么对场景使用光照处理会使它的形状突出，而且经过着色处理的图像颜色并不能精确表示数据值。这种情况可以通过为环境光和漫反射光使用同样的颜色来最小化(尤其是白色的时候)，并使用来自渐变色的材质颜色，这种渐变色的亮度是相对稳定的，但色度会稍微变化，所以在这种光照处理模式下的着色处理只改变亮度，而不改变颜色。图6-6显示了经过着色处理的南非地貌图，不同的颜色表示海拔高度，但图中的着色处理结果给人感觉光照是在天空的东部，陆地的北面——实际太阳是在早上照射在陆地上的。如果合成颜色的图像模型没有表现几何形状的物理意义，那么就没有理由使用光照处理和着色处理了。

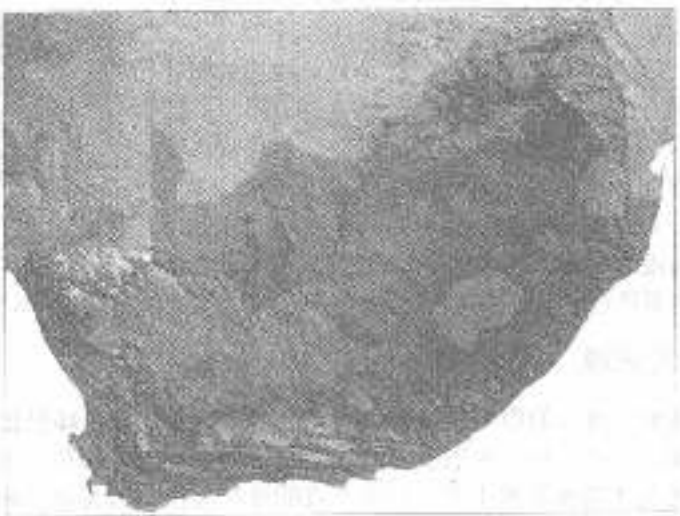


图6-6 经过光照处理的南非伪色彩地表，参见彩图

6.9 定义

多边形的Flat着色处理用一种颜色表示一个多边形，它假设多边形上所有的点所进行的光照

处理都是一样的。Flat的意思是整个多边形的颜色是平坦的（不变化），或者说多边形着色的时候是平坦的，所以照亮时它的颜色不变。如果给多边形单纯地设置一种颜色而且不进行光照处理也可以得到这种效果，或是使用光照处理和材质模型，但显示的时候对整个多边形使用同一个法向向量（这个多边形是平坦的）。同一个法向向量使整个多边形的光照处理计算都是一样的。 224

对多边形进行平滑着色处理使得整个多边形内部的像素颜色平滑变化。因为图形系统在三角形内部通过对各顶点线性插值计算颜色，所以，要求多边形的每个顶点的颜色不同。插值是在投影之后确定了顶点位置才进行的，所以这个线性计算很容易在图形卡中实现。每个顶点的颜色可以通过为每个顶点赋予不同的颜色来实现，也可以通过对每个顶点进行光照计算来实现。为了分别计算每个顶点的颜色，必须为多边形的每个顶点定义法向向量，使得光照模型可以为每个顶点产生不同的颜色。

每种图形API都以同样的方式支持Flat着色处理，但对于平滑着色处理，可能不同的图形API有不同的实现方法，所以需要了解特定的API是如何进行处理的。最简单的平滑着色处理可以通过先计算每个顶点的颜色，然后对整个平面进行平滑的颜色插值来实现。这种方式的Gouraud着色处理模型将在下一节介绍。如果多边形是三角形，那么三角形中的每个点都是顶点的凸组合，所以也可以对它使用同样的顶点颜色的凸组合。随着计算机图形学越来越成熟，可以看到图形API中更多复杂的多边形着色处理，所以确定多边形中像素点的颜色将变得更加具有灵活性。

6.10 Flat着色处理和平滑着色处理的例子

在本书中我们看到了很多多边形的例子，但并没有仔细区分它们是进行Flat着色处理还是平滑着色处理。如图6-7显示的两幅图像，对于用同一个逼近函数定义的曲面，分别采用Flat着色处理（左）和平滑着色处理（右）。经过平滑着色处理的图像看起来更加平滑，但有些区域的三角形变化仍然非常快，并且相邻三角形之间的颜色插值在视觉上变化也很快。平滑着色处理能达到比较好的结果（通常比Flat着色处理优秀），但并不是最完美的。 225

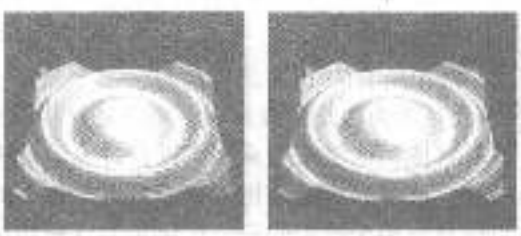


图6-7 对同一表面分别采用Flat着色处理（左）和平滑着色处理（右）的效果

在平滑着色处理中，每个顶点都有通过解析方法计算得到的法向，这将在下一节讨论。所以，光照计算模型为每个顶点计算不同的颜色，然后插值计算多边形中每个像素的颜色，这些颜色在多边形内部平滑地变化，在多边形上产生平滑的渐变颜色。这种插值称为Gouraud着色处理。它的计算速度非常快，因为它只依赖于多边形顶点的颜色，但它可能在多边形内部丢失光照效果。很显然，在显示沿着多边形边沿的点颜色的时候，与理想的平滑着色处理相比，它更容易受影响，如图6-7右图所示。其他类型的插值并没有这种问题，但它们可能不被图形API支持。Phong着色处理就是这方面的一个例子，我们将在下一章讨论。

有个非常有趣的实验可以帮助理解着色处理表面的属性，那就是考虑平滑着色处理和显示网格分辨率的关系。理论上，可以通过对非常精细的网格使用Flat着色处理来达到和对相对粗糙的网格使用平滑着色处理相同的结果。定义特殊的网格大小和Flat着色处理，通过寻找更小的网格，使它产生的图像和对原来的网格使用平滑着色处理产生的图像接近。图6-8是这方面的一个例子。图中的表面上仍然有一些Flat着色处理的小平面，但它避免了大部分使用粗糙平滑着色处理的快速变化的表面方向带来的问题，并且在很多方面比图6-7中用平滑着色处理

的多边形效果要好。增加网格的大小会使程序比原来的平滑着色处理变快，或者变慢，这依赖于图形流水线中的多边形插值效率。这是一个对计算机图形学非常有用的实验方法的例子：如果有多种不同的方法来近似一种效果，那么应该尝试每一种方法，并且在特别的程序中观察在效果和速度方面哪个最好。

226

降低网格的粒度可以一直继续到每个像素对应一个多边形的水平。在那个水平上，不是对整个多边形进行着色处理，而是每个多边形都有独立的法向向量，这就是Phong着色处理。

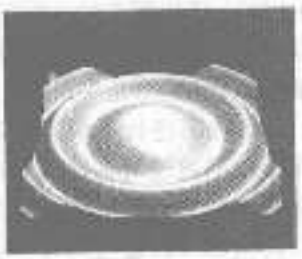


图6-8 在每个方向上以三倍于前面图像的粒度进行细分，然后使用Flat着色处理得到的图像

6.11 计算每个顶点的法向

图6-7所示的两幅图像设计的真正差别在于用Flat着色处理的图像对一个多边形只使用一个法向，而用平滑着色处理的图像对多边形每个顶点都使用独立的法向向量。因为这是个数学方程表示的曲面，可以通过解析的方法计算法向来确定顶点的实际法向。对每个顶点计算法向比对每个多边形计算法向的工作量大，这就是平滑着色处理的代价。这里讨论的计算法向的方法可以是包含该顶点的所有多边形法向的加权平均，也可以是解析计算的结果。

6.11.1 平均多边形法向

如果一个顶点被模型中的多个多边形共享，那么可以通过计算所有的与该顶点相交的多边形法向N的加权平均来得到该顶点的向量。它的计算公式如下：

$$N = (\Sigma a_i N_i) / (\Sigma a_i)$$

对于所有的i，多边形P_i包含那个顶点，每个多边形P_i的法向是N_i，并且对该顶点的权重是a_i。（因为这是向量求和，所以是对法向的x、y、z分量分别进行计算。）每个权重可以根据多边形对该顶点法向的重要性进行计算。如果所有的权重是相等的，就把所有的顶点法向进行平均。通常的办法是把多边形在这个点上的角度作为权重。角度a_i的大小是多边形P_i中两条与该顶点相交的边的归一化向量的点积的值的反余弦函数值，因为点积是两条边夹角的余弦值。这些多边形的角可以通过顶点信息得到并使用。

6.11.2 法向的解析计算

在图6-7的例子中，通过解析计算每个顶点的法向N是可能的，因为这个表面是通过闭合方程定义的：

$$f(x, y) = 0.3*\cos(x*x + y*y + t)$$

我们可以计算每个顶点的解析方向导数，在x方向和y方向分别是：

$$fx(x, y) = \partial f / \partial x = -0.6*x*\sin(x*x + y*y + t)$$
$$fy(x, y) = \partial f / \partial y = \quad 0.6*y*\sin(x*x + y*y + t)$$

227

用它们来计算这些方向的切线向量，它们的叉乘可以得到顶点的法向。对于函数f，它的偏导为fx和fy，那么曲面在x和y方向的切向量分别是<1.0, fx >和<0.1, fy >。通过第4章讨论的方法计算叉积，可以看到表面上该点的法向向量为<−fx, −fy, 1>。可以通过计算或者启用GL_NORMALIZE来对向量进行归一化，完成表面法向的计算。本章最后的示例代码对此进行了展示。

也可以从其他模型中得到准确的法向：以前看到过球的法向是球的半径向量。所以，简单的几何模型可能有非常准确的法向。通常，如果模型允许通过解析或者几何计算法向，那么会更加准确而且可以产生比插值更好的效果。

6.12 其他着色处理模型

我们不能假设初级的API例如OpenGL提供的平滑着色处理模型能够准确地表现平滑表面。因为它假设多边形的表面是均一变化的，所以在计算整个多边形颜色的时候包含每个顶点的信息，并且依赖于RGB颜色空间的线性特征。这其实是不正确的。和许多计算机图形系统的特征一样，它也只是近似真实，其实有更好的方法来实现平滑表面。例如，Phong着色处理模型，为每个顶点设立一个法向并且通过对法向进行插值而不是颜色插值来计算多边形中每个像素的颜色。插值法向比插值颜色更复杂，因为在屏幕空间上均匀分布的像素并不是来自均匀分布在三维眼空间或者三维模型空间中的点，在透视投影中还要除以眼空间中点的Z坐标值。这使得法向插值比颜色插值更加复杂（速度也更慢），而且不为一系列的图形API支持。然而，Phong着色处理模型假设整个多边形都是真正的平滑表面，可以在多边形内部产生镜面反射光，而且沿着多边形的边也非常平滑。Gouraud和Phong着色处理的细节在其他图形学书中都有讨论。读者可以把它们作为在很多计算机图形处理中使用的插值的极好的例子。

图6-9显示了分别用Flat着色处理、平滑着色处理和Phong着色处理过的三幅图像在视觉上 228
的差别。从图中可以注意到，经过Flat着色处理的球有明显的小面片，经过平滑着色处理过的球有一些面片边并且镜面反射光反射的分布也很不平滑，但经过Phong着色处理的球，面片边和镜面反射光反射比较平滑。



图6-9 分别用平面（左），平滑（中），Phong（右）着色处理球体得到的结果

Phong着色处理模型是基于对整个多边形法向进行连续的改变，还有一种着色处理模型是基于对整个多边形的法向进行控制，就像纹理映射，下面将对此进行讨论。这种模型可以产生随着表面变化的效果。我们可以独立能够修改多边形内部法向的映射，使得着色处理模型能够产生凹凸表面的效果。这称为凹凸贴图，跟Phong着色处理一样，每个像素的法向都是单独定义的。每个像素的法向是通过组合下面两个法向得到：一个是Phong着色处理的法向，另一个是通过颜色梯度得到的凹凸贴图的法向。每个像素点的颜色通过使用该像素点的法向用标准的光照模型计算得到。图6-10显示了凹凸贴图效果的一个例子。注意凹凸贴图只是一幅2D图像，每个点的高度通过灰度等级来定义，称为高度场。法向通过计算场中颜色的改变来得到。

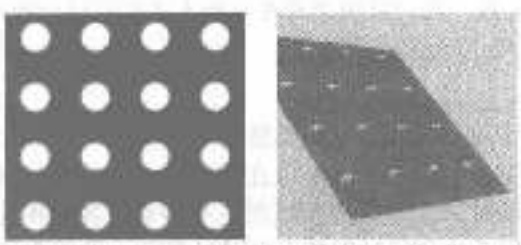


图6-10 作为高度场定义的凹凸贴图（左）和应用到具有镜面反射光的表面的结果（右）

通过灰度图表示高度可以看成是对表面应用灰度渐变色。在地图上，对区域的这种表示称为数字高度图。通过结合高空照相与数字高度图，可以非常详细地表示这个3D区域。第9章将给出这方面的一个例子。

6.13 各向异性着色处理

到目前为止的着色处理模型都是基于前一章介绍的简单光照处理模型，假设光线是均匀
229 地从表面法向反射（半方性光照）。然而，有一些材质的光照参数根据光线和眼睛围绕法向的夹角而变化。这些材质包括用毛刷刷过的金属，或者CD上没贴标签的那个表面。对这些材质的着色处理称为各向异性着色处理，因为不同方向的反射是不一样的。光照计算中的那些角包括漫反射中的法向角度和镜面反射光反射中反射光线的角度，被一个更加复杂，称为双向反射分布函数（BRDF）代替，用ρ表示。它依赖于眼睛的纬度角Θ和经度角Φ，以及光照射的点位置：$\rho(\Theta_r, \Phi_r, \Theta_i, \Theta_i)$。BRDF也考虑对不同波长（或不同颜色）的光线采取不同的行为。这种材质的光照计算比通常的各向同性的更复杂，而且已经超出了图形API的范围，但可以在一些专业的图形工具中发现这种着色处理的方法。图6-11显示了对图6-9的球进行各向异性着色处理的结果。这种着色处理可以通过修改表面法向来模拟近似的BRDF，当达到把每个像素当作一个多边形来处理的时候，就是真正的各向异性着色处理。



图6-11 用各向异性着色处理后的球体效果图

顶点和像素着色器

着色处理一个非常重要的进步是着色处理语言的开发，使得程序员可以更加精细地定义着色处理。这些语言提供了一种可编程的方式来访问可编程图形卡的功能，允许对顶点和逐个像素进行操作。这样可以开发各向异性着色、运动模糊、凹凸贴图、智能纹理和其他一些奇特的效果。现在开发的着色器语言可以作为图形API的一部分，供程序员使用。在第10章我们将看到它是如何被整合进渲染流水线的结构的。OpenGL的最新版（OpenGL 2.0）包括着色处理语言。但是对它进行深入描述已经超出了本书的范围。

OpenGL着色处理语言GLSL是一种高级的面向过程的语言，和C语言有些相似，但添加了
230 一些反映可编程图形特性的特有的数据和操作。它代替了图形卡的某些函数。我们将在第10章中讨论。它的原理很简单，把顶点数据发送给图形卡，把它们转化为像素位置，通过多边形扫描线（称为片段）来设置每个像素的颜色。GLSL包括顶点着色器（用来处理坐标、法向、纹理坐标或者颜色）和片段着色器（用来提供更真实的就如各种非照相真实感的效果）。它们都允许把纹理内存另作它用，而且极大地增加了可以在图形卡中进行的工作。这对高级图形系统开发是个非常重要的工具，我们鼓励读者在开始本书的阅读后使用它们。

6.14 全局光照

前面讨论了计算机图形学中的局部光照模型，还有另一种光照处理方法可以产生更加真实的图像。在周围的世界中，光线并不只来自直射光源，也并不只有单个环境光值。光线被场景中的每个物体和表面反射，这些间接光源在整个场景中各不相同。解决这种问题的光照处理称为全局光照模型，因为光照是针对整个场景计算的，并不和视点相关，也不是根据视

点对每个多边形进行计算，正如本章之前所做的。全局光照过程并不包括着色处理模型，任何光线能量到达表面产生的着色处理结果都传递给另外的表面。

全局光照的另一个好处是一旦计算了场景中每个点的光照，就可以非常快速地显示场景。这是一个很好的非对称图形处理的例子。通过一次性进行大量处理来建立模型的光照处理，一旦计算完成，以后从不同视点多次显示模型就不需要太多处理。全局光照有很多有趣、且非常有用的应用，但它现在还不被初级的图形API支持，尽管在以后可能会得到支持。我们将介绍两个广泛使用的全局光照模型。

6.14.1 辐射度方法

经典的全局光照模型是辐射度方法，它基于在封闭空间内辐射能量的传递。假设所有的表面都能辐射能量，通过一系列步骤计算辐射度，最后光照趋向稳定。首先，光源发射光能量，所有其他的源头都不发射能量。计算到达每个表面的能量并存储在表面上。在以后的步骤中发射出来。在随后的步骤中，每个表面都根据它接收到的能量和材质属性发射能量。重复上一步直到每个像素点两次计算的能量差异很小为止。当显示场景的时候，每一点都根据它辐射的能量上色。图6-12显示的是用辐射度方法计算的图像。



图6-12 用辐射度方法渲染的场景，参见彩图 231

实际上，场景必须被分割为很多个小区域，每个小区域都是很好的漫反射器（如图6-1显示的反射光线）。场景中每个物体都被细分成这样的区域，所以每个区域的光照大致相等。如何以这种方式细分场景是定义辐射度方法的一个难点。每个区域都从其他区域接收能量并且发射能量给其他区域，辐射度方法的主要计算工作是计算能量的传递。这可以通过为每个区域建立一个半球面（或者近似的，如半立方体），并计算其他区域在这个球面上的投影。因为是计算漫反射，一个区域接收的能量和该区域的投影面积成比例，也和该区域发射的能量有关。当能量稳定后，能量被转化为光照，接着就是渲染场景。

6.14.2 光子映射

另一个全局光照模型称为光子映射。它使用和光线追踪相似的技术来建立场景中光能量的模型（见第14章）。光子映射的基本步骤包括从场景中的每个光源产生发射大量的随机方向的直射光线。每根射线表示来自光源的特殊颜色（依赖于光源的属性）的光子路径。当光线和物体相交的时候，物体记录该颜色光子的到达情况，根据接收到的光子数来衡量光照强度。但是，全局光照的前提是物体不但接收光照，也发射光照，每个光子到达物体后又从该物体发射出去，这取决于组成物体的材质表现的概率计算。如果光子被发射，它的方向是随机的，它的颜色取决于初始光源和材质。这个过程递归继续，直到达到了概率计算或者给定的计算步骤数而结束。实际上，发射和跟踪的光子数可能非常小。 232

当所有的光子都被发射并且场景中所有的物体都累计了和光子相交的信息，继续处理场景来决定每个物体的光照值。和大部分全局光照模型一样，光照计算只需进行一次，除非有物体在场景中移动了。一旦光照强度已知，场景可以通过光线投射算法（如果场景中没有反射或者折射材质）、光线跟踪算法（如果有反射或者折射材质）、或者其他技术进行渲染来展现给观看者。因为光照计算不需要每次渲染都重新进行，所以，在诸如进行场景漫游显示的时

候可以得到很好的帧率。光子映射的完整讨论可以参看Jensen[JEN]。图6-13显示通过这种方式渲染的结果。



图6-13 用光子映射模型渲染的场景，参见彩图

6.15 局部光照和OpenGL

与全局光照相比，它考虑每个表面反射的能量，局部光照假设光能量只来自已经定义的光源。这也是OpenGL的处理方式，任何一点的光照只考虑环境光、漫反射光和镜面反射光这三种光照组成，这在前面已经讨论过。本节将讨论这在OpenGL中是如何进行的。

OpenGL图形API支持大部分先前讨论的局部光照功能。此处我们列出OpenGL支持的光照和材质处理的主要函数。一些OpenGL函数例如glLightf (light, pname, set_of_value) 使用独立的实数（或者整数）参数，另一些函数例如glLightfv (light, pname, vector_values) 则使用向量作为参数。如果可以选择，那么可以任意选择一种与设计和代码相匹配的形式。

233

按照OpenGL通常的做法，需要使用特殊的名字来指定某些值。在标准的OpenGL中，光源必须指定为从GL_LIGHT0－GL_LIGHT7（有些实现可能允许更多的光源，但八个光源是OpenGL的标准）。定义属性的参数也是光源外观属性的一部分。参数pname必须是以下可供使用的参数之一：

```
GL_AMBIENT,
GL_DIFFUSE,
GL_SPECULAR,
GL_POSITION,
GL_SPOT_DIRECTION,
GL_SPOT_EXPONENT,
GL_SPOT_CUTOFF,
GL_CONSTANT_ATTENUATION,
GL_LINEAR_ATTENUATION, or
GL_QUADRATIC_ATTENUATION
```

本节我们将讨论使用这些参数的OpenGL光源的属性。

6.15.1 指定和定义光源

当设计场景和光照的时候，可以通过使用函数glLightModel(...)来设置光照的一些基本属性来定义光照模型。这个函数最重要的功能是定义场景进行单面还是双面光照处理，它通过如下函数定义：

```
glLightModel[f|i](GL_LIGHT_MODEL_TWO_SIDE, value);
```

[f|i]表示使用f还是i，分别表示参数value是实数还是整数。如果数值参数的值（实数或者整数）是0，那么使用单面光照处理，也就是说只有材质的前向面进行光照计算；如果value非零，那么前向面和背面都进行光照计算。

该函数的另外一个功能是在假设观看方向和Z轴平行或者指向视点的条件下，选择是否进行镜面反射光计算。通过如下函数进行：

```
glLightModel[f|i](GL_LIGHT_MODEL_LOCAL_VIEWER, value);
```

value的值为0表示观看方向和Z轴平行，非0表示观看方向指向视点，这就是我们在本章前面讨论的镜面反射光计算的情况。默认的值是0。

最后，可以通过这个函数来设置全局环境光。对来自场景中每个光源的环境光部分进行补充，可以定义独立于任何特殊光源的全局环境光。通过如下形式定义：

```
glLightModelf(GL_LIGHT_MODEL_AMBIENT, r, g, b, a)
```

234

r,g,b,a也可以是等价的向量形式，这个光源值被添加到全局环境光的计算当中。

OpenGL允许在场景中定义多达八个光源，它们的符号名称为GL_LIGHT0...GL_LIGHT7，要使它们变得可用，需使用glLight*(...)函数定义属性来创建光源。以下列出对光源LIGHT0的位置和颜色进行的定义，其他光源的位置和颜色的定义可以以此类推：

```
glLightfv(GL_LIGHT0, GL_POSITION, light_pos0); // 光源 0
glLightfv(GL_LIGHT0, GL_AMBIENT, amb_color0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diff_col0);
glLightfv(GL_LIGHT0, GL_SPECULAR, spec_col0);
```

此处我们定义了光源位置，并且为镜面反射光、漫反射光和环境光指定了颜色，这些值可以通过如下语句定义：

```
GLfloat light_pos0 = { .... .... .... };
GLfloat diff_col0 = { .... .... .... };
```

原则上，这两个向量都是四维的，其中位置向量的第四维是齐次坐标，光源颜色向量的第四维是α值。对位置光来说，齐次坐标的值是1.0；对方向光来说，它是0.0。我们为颜色指定了α值，一般它的默认值是1.0，我们建议对光源颜色的α也使用这个值，只要在定义光源的时候指定RGB光源分量就可以了。

正如在本章前面注意到的，为了使光照计算成功，需要为物体表面定义法向。因为光照计算需要通过点积来计算余弦值，所以必须保证法向向量是归一化的，在显示函数中指定任何几何体前，通过函数glEnable(GL_NORMALIZE)开启自动归一化，可以保证这一点。这项工作可以在初始化函数中进行。

任何光源在场景中可用之前，必须开启场景的光照功能而且每个使用的光源也要开启。这在OpenGL中是个非常简单的过程，首先，通过调用如下函数表明在场景中使用光照：

```
glEnable(GL_LIGHTING);//使用光照模型
```

然后通过调用启用函数使每个光源可用，下面通过启用三个光源的例子来说明：

```
glEnable(GL_LIGHT0);//使用LIGHT0
glEnable(GL_LIGHT1);//使用LIGHT1
glEnable(GL_LIGHT2);//使用LIGHT2
```

光源也可以通过glDisable(...)函数来禁用，可以用来决定光源什么时候可用，什么时候不可用。光源的这种功能可以用来强调或者突出场景的某一部分，这可以通过为需要突出的物体使用独立的光源来实现，也可以在动画中使用它，或者在设计例如允许用户交互选择光源的显示时使用它。

235

光源的其他属性在OpenGL的设置也非常简单。如果想设置一个聚光灯，那么需要像设置标准位置属性一样，设置聚光灯的方向、光锥角和角衰减系数。这些属性的设置通过如下方式使用glLightf*(...)函数来实现：

```
glLightf(light, GL_SPOT_DIRECTION, 1.0, -1.0, -1.0);
glLightf(light, GL_SPOT_CUTOFF, 30.0);
glLightf(light, GL_SPOT_EXPONENT, 2.0);
```

如果没有指定聚光灯的光锥角和角衰减系数，那么它们分别是180°（这也意味着该光源并不是聚光灯）和0。如果设置了光锥角，那么该参数是以度数表示的光锥角大小，其值并被限整到0°和90°之间。

实际上，OpenGL并没有为光线衰减建立模型，但可以通过某种方式设置它，使它变得有用。光线衰减有三个属性：常量、一次和二次衰减，每个属性的值可以分别用常量

GL_CONSTANT_ATTENUATION、GL_LINEAR_ATTENUATION和GL_QUADRATIC_ATTENUATION表示。如果这三个衰减系数分别是A_C、A_L和A_Q，光源到表面的距离是D，那么光强值要乘以衰减因子：

$$A = 1/(A_C + A_L * D + A_Q * D^2)$$

其中D是光源位置与顶点的距离，A_C、A_L和A_Q的默认值（常数，一次，二次衰减项）分别是是1.0、0.0和0.0。实际的衰减常量值可以通过以下函数来设置：

```
glLightf(GL_* ATTENUATION, value)
```

通配符*可以用CONSTANT、LINEAR或QUADRATIC来代替。

方向光的指定可以通过把位置向量的第四个分量设为0来实现。光线的方向可以通过前三个分量来确定，并且由模型视图矩阵进行变换。方向光不可以有衰减属性，但它的其他属性和别的光源一样；它的方向用于漫反射光和镜面反射光的计算，但并不计算距离。通常按照如下方式定义方向光：

```
glLightf(light, GL_POSITION, 10.0, 10.0, 10.0, 0.);
```

6.15.2 选择性地使用光源

为了选择性地使用光源，必须定义两个或更多的光源，并且可以启用或禁用任何一组光源。在图6-5显示的春秋分例子中，使用了两个光源，在display()函数中可以看到如下使用GL_LIGHT1显示太阳和使用GL_LIGHT0显示地球和赤道面的代码。

```
// 太阳
    glEnable(GL_LIGHT1);
    glDisable(GL_LIGHT0);
...
// 地球
    glDisable(GL_LIGHT1);
    glEnable(GL_LIGHT0);
...
// 赤道面
...
```

236

6.15.3 定义材质

为了使OpenGL能够建立光线和物体的交互模型，物体必须定义表处理环境光、漫反射光和镜面反射光的方式。那意味着必须定义物体在环境光中的颜色以及在漫反射光中的颜色。我们不需要定义镜面反射光的颜色，因为镜面反射光是用光线的颜色代替物体的颜色。但是必须定义材质处理镜面反射光的方式。它实际上表示物体的光泽程度以及光泽的颜色。

OpenGL利用多边形的双面性，允许为材质指定照亮多边形的前向面，还是照亮多边形的背面（前面讨论过的前向面和背面），或者是两个面都照亮。在指定材质时使用参数GL_FRONT、GL_BACK或者GL_FRONT_AND_BACK实现。如果使用双面光照处理，那么必须为前向面和背面同时指定材质属性。通过用参数GL_FRONT_AND_BACK代替分别使用GL_FRONT和GL_BACK，为前向面和背面定义相同的材质属性。这允许为物体的前向面和背面使用不同的颜色，并且当物体不封闭的时候，可以明确哪一面将被看见。

为了定义物体的材质属性，使用一系列glMaterial*(...)函数。它们的通用形式如下：

```
glMaterial[if][v](face, pname, value)
```

可以分别或者以向量的形式（[V]）使用整数或实数参数值（[if]）。参数face是符号名，必须是GL_FRONT、GL_BACK或者GL_FRONT_AND_BACK三个之一，pname是个符号常

量，必须是GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS或者 GL_AMBIENT_AND_DIFFUSE之一。最后，参数value的值可以是单个数、一组数或者一个向量。用来设置依赖于参数名字的符号参数值。以下是设置这些值的简单例子：

```
GLfloat shininess[]={ 50.0 };
GLfloat white[] = { 1.0, 1.0, 1.0, 1.0};
glMaterialfv(GL_FRONT, GL_AMBIENT, white);
glMaterialfv(GL_FRONT, GL_DIFFUSE, white);
glMaterialfv(GL_FRONT, GL_SPECULAR, white);
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);
```

以上代码给材质赋予了非常中性的颜色，使它可以体现光源的颜色。

大部分的参数和值都在先前讨论光照模型时介绍了。这里的GL_AMBIENT_AND_DIFFUSE参数并没有多大的价值。因为材质在环境光和漫反射光中有相同的属性是很普遍的。在这两种 237
情况下，光能量被材质吸收，然后带着物体自身的颜色再辐射出去。该参数允许为它们定义相同的属性。

6.15.4 使用GLU二次曲面物体

正如我们第一次看到GLU二次曲面物体时讨论的一样，OpenGL可以为这些物体产生自动法向向量。通过如下函数实现：

```
gluQuadricNormals(GLUquadric* quad, GLenum normal)
```

它允许把法向normal设置为GLU_FLAT或者GLU_SMOOTH，这依赖于为物体使用的着色处理模型。

6.15.5 例子：把三原色光源应用于白色表面

一些光照情况很容易看到。如果用白光照射有色表面，那么看到的是表面的颜色，因为白光包含所有的光成分，而表面拥有的颜色是它们中间被反射的。同样，如果用有色光照射白色表面，那么看到的是光的颜色，因为只有那个颜色是可见的。如果用有色光照射有色表面，那么这种情况有点复杂，因为表面只反射到达它的颜色。所以，如果用（纯）红色光照射（纯）绿色表面，那么将得不到任何反射，表面是黑色的。在现实世界中并不能看到这种情况，因为现实世界中并不能看到纯色光，但是在人工的场景中却很容易发生这种情况。

让我们来看有色光照射在白色表面上的效果。白色表面会反射任何它接收到的光，如果它接收到红色光，那么它只反射红色。如果在一个拥有三种颜色（红、绿、蓝）光源的场景中放置一个简单形状（立方体），可以看到它反射这些颜色。在接下去的例子中，光源从三个不同的方向照射在白色立方体上。这个立方体和前面的立方体有非常大的差别，这个立方体不但包含各个面的顶点，还包含每个面的法向。如果对立方体进行旋转，使得每个面都受到多于一个光源的照射，那么可以看到光源照射在各种不同的面上，可以对它们各种反射属性进行实验。从这个实验可以看到单个光源的效果，也可以看到两个或三个光源照射在表面上的效果。读者可以对光源进行移动，并且重编译代码来实现其他光源效果。

6.15.6 示例代码

在初始化函数中定义光源颜色和位置：

```
GLfloat light_pos0[]={0.0, 10.0, 2.0, 1.0};//y轴向上
GLfloat light_col0[]={1.0, 0.0, 0.0, 1.0};//红色光源
GLfloat amb_color0[]={0.3, 0.0, 0.0, 1.0};
```

```
GLfloat light_pos1[]={5.0, -5.0, 2.0, 1.0};//右下
GLfloat light_col1[]={0.0, 1.0, 0.0, 1.0};//绿色光源
GLfloat amb_color1[]={0.0, 0.3, 0.0, 1.0};
GLfloat light_pos2[]={-5.0, 5.0, 2.0, 1.0};//左下
GLfloat light_col2[]={0.0, 0.0, 1.0, 1.0};//蓝色光源
GLfloat amb_color2[]={0.0, 0.0, 0.3, 1.0};
```

238

在初始化函数中定义光源属性和光照模型：

```
glLightfv(GL_LIGHT0, GL_POSITION, light_pos0);  //light 0
glLightfv(GL_LIGHT0, GL_AMBIENT, amb_color0);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_col0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_col0);
glLightfv(GL_LIGHT1, GL_POSITION, light_pos1);  //light 1
glLightfv(GL_LIGHT1, GL_AMBIENT, amb_color1);
glLightfv(GL_LIGHT1, GL_SPECULAR, light_col1);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light_col1);
glLightfv(GL_LIGHT2, GL_POSITION, light_pos2);  //light 2
glLightfv(GL_LIGHT2, GL_AMBIENT, amb_color2);
glLightfv(GL_LIGHT2, GL_SPECULAR, light_col2);
glLightfv(GL_LIGHT2, GL_DIFFUSE, light_col2);
glLightModelv(GL_LIGHT_MODEL_TWO_SIDE, &i);     //双面
```

在初始化函数中启用光源：

```
glEnable(GL_LIGHTING);//使用光照模型
glEnable(GL_LIGHT0);//使用光源LIGHT0
glEnable(GL_LIGHT1);//...LIGHT1
glEnable(GL_LIGHT2);//...LIGHT2
```

在绘制表面的函数中定义材质颜色，必须定义物体材质的环境光和漫反射光部分，如以下代码所示。光泽度必须是长度为1的数组(指向数的指针)。光泽度越高，在物体上产生的镜面反射光更聚集而且更小。此例子中并没有为背面指定材质的属性，因为物体是封闭的，所以，所有的背面材质都是不可见的。

```
GLfloat shininess[] =       { 50.0 };
GLfloat white[] =           { 1.0, 1.0, 1.0, 1.0 };
GLfloat mat_specular[] =    { 0.8, 0.8, 0.8, 1.0 };
glMaterialfv(GL_FRONT, GL_AMBIENT, white);
glMaterialfv(GL_FRONT, GL_DIFFUSE, white);
glMaterialfv(GL_FRONT, GL_SHININESS, shininess);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
```

对如图6-14所示的立方体进行旋转，使得一个角指向观看者。很显然，红色光在上面，绿色光在下面并延伸到观看者视点的右边，蓝色光在下面并延伸到观看者视点的左边。我们鼓励读者编写这个代码来感受这些效果。

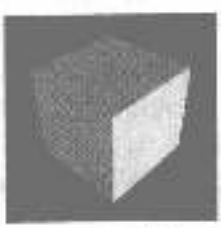


图6-14 用三种不同颜色的光源观看的白色立方体。参见彩图

6.15.7 着色处理的例子

使用OpenGL的着色处理必须选择着色处理模型并且为每个顶点设置颜色，可以通过显式调用glColor*(...)函数，

239

也可以调用glNormal*(...)函数为每个顶点设置法向，然后使用光照处理。OpenGL默认的着色处理是平滑着色处理，除非为模型指定了法向，否则看不到平滑着色处理的视觉效果。OpenGL允许使用glShadeModel函数来选择着色处理模型，并且该符号常量的值只能是

GL_SMOOTH或者GL_FLAT。可以在任何时候使用glShadeModel函数在Flat和平滑着色处理间切换。

如果使用Flat着色处理，那么对于每个三角形，它的顶点记为P[0],P[1],P[2]，计算它的法向并使用它。在这个例子中，可以看到如下的代码：

```
glBegin(GL_POLYGON);
// 计算三角形法向Norm
    calcTriangleNorm(p[0],P[1],P[2],Norm);
    glNormal3fv(Norm);
    glVertex3fv(P[0]);
    glVertex3fv(P[1]);
    glVertex3fv(P[2]);
glEnd();
```

以上代码可以为每个三角形计算一个法向。假设已经定义了函数calcTriangleNorm(...)，可以通过第4章讨论的方法，例如叉积来实现。

然而，我们希望在例子中使用平滑着色处理。在下面的示例代码中，我们考虑通过函数$f(x, y) = 0.3(x^2 + y^2 + t)$定义的表面，该函数只有一个变量$t$。通过定义解析法向向量在每个顶点上设置平滑着色处理。开始，通过在init()函数中调用如下函数来自动保证所有的法向是单位长度，这样可以避免自己进行计算。

```
glEnable(GL_NORMALIZE);//变换后使法向向量单位化
```

我们通过使用表面的解析属性为每个顶点生成法向。可以通过计算函数的偏导数$\partial f/\partial x$和$\partial f/\partial y$来得到每个顶点的切向量。

```
#define f(x,y) 0.3*cos(x*x+y*y+t)          //原始函数
#define fx(x,y) -0.6*x*sin(x*x+y*y+t)      //x的导数
#define fy(x,y) -0.6*y*sin(x*x+y*y+t)      //y的导数
```

240

在显示函数中，首先通过在定义域中计算网格点的函数XX(i)和YY(j)来计算x和y的值。用它们计算切向量<1,0, fx>和<0,1, fy>。然后为表面上每个三角形进行内联叉积，如以下代码所示。我们必须很仔细地按照（X偏导）x（Y偏导）这种顺序来计算三角形表面向量，只有这样才能根据叉积的右手规则产生正确的法向方向。我们本来可以使用解析的形式进行叉积，这样可以避免进行叉积计算，但实际上后者的使用更普遍，而且可以在没有有效的解析形式时使用。

```
glBegin(GL_POLYGON);
    x=XX(i);
    y=YY(j);
    vec1[0]=1.0;
    vec1[1]=0.0;
    vec1[2]=fx(x,y);          //X-Z平面上的偏导数
    vec2[0]=0.0;
    vec2[1]=1.0;
    vec2[2]=fy(x,y);          //Y-Z平面上的偏导数
    Normal[0]=vec1[1]*vec2[2]-vec1[2]*vec2[1];
    Normal[1]=vec1[2]*vec2[0]-vec1[0]*vec2[2];
    Normal[2]=vec1[0]*vec2[1]-vec1[1]*vec2[0];
    glNormal3fv(Normal);
    glVertex3f(XX(i), YY(j), vertices[i][j]);
    ...//为三角形的其他顶点重复两次相同的代码
glEnd();
```

这可能比先建立向量vec1和vec2，然后调用工具函数计算叉积更高效。这个例子产生如图6-7所示的平滑着色处理效果。

6.16 建议

OpenGL光照模型缺乏一些非常重要的功能，这些功能可以让场景达到真正想要的真实效果。其中非常重要的一个是阴影。OpenGL有创建阴影的技术，但它们需要特殊的处理。另一个是“热”色彩，它是指发射出的特定颜色比它们从光线中接收到的更多，正如在很多书中描述的一样，没有办法纠正这个问题，因为任何计算机屏幕发光粉的色域宽度都是有限的。最后，OpenGL不允许方向（各向异性）反射，如果需要独立材质模型例如刷过的金属铝，可以通过编写特殊的计算机程序。不要认为OpenGL的光照模型是处理颜色最正确的方式，它可以处理一些通常情况，要达到一些非常奇特的效果，需要特殊处理。

6.17 小结

本章讲述了如何利用Phong光照模型处理顶点颜色，利用Gouraud着色处理模型进行线性平均颜色，
241 建立在整个多边形上平滑变化的颜色，并利用它们来创建图像。这对于处理图像外观部分非常有意义，允许为观看者产生更加有趣且含有更多信息的图像。至此，外观部分的简单扩展只剩下纹理映射了，我们将在第8章讨论。

6.18 本章的OpenGL术语表

OpenGL有一些函数用来指定光照处理和着色处理，此处列出它们以供参考。本章新介绍的OpenGL函数不多，但它们使用的参数很多，按照惯例，我们并不包含以前介绍过的函数或者常量。

OpenGL函数

glLight*(light, pname, value[s]):为指定的光源设置命名参数的值的函数集，函数名反映value值是整型还是浮点型。

glLightModel*(pname, value[s]):为将要使用的光照模型设置参数值的函数集，函数名反映了value值是整型，浮点型还是向量，并且value的个数依赖于设置的参数。

glMaterial*(face, pname, value):为光照模型指定材质属性的函数集，函数名反映了value值是整型，浮点型还是向量，并且value的个数依赖于设置的参数。

glShadeModel(pname):根据所用的参数，选择Flat着色处理或者选择平滑着色处理。

gluQuadricNormals(quad, pname):根据所用的参数，为GLU物体选用Flat着色处理或者选用平滑着色处理。

字面常量

GL_AMBIENT:定义光源的环境光部分或者材质的环境光反射系数的参数

GL_CONSTANT_ATTENUATION:指定光源具有常数衰减系数的参数

GL_DIFFUSE:定义光源的漫反射部分或者材质的漫反射系数的参数

GL_FLAT:在glShadeModel()函数中选择Flat着色处理的参数

GL_LIGHT*:标识启用、禁用或定义特殊光源的参数

GL_LIGHT_MODEL_LOCAL_VIEWER:定义镜面反射光是相对观察者的眼睛位置还是相对z轴计算的参数

GL_LIGHT_MODEL_TWO_SIDE:为glLightModel()函数指定选择单面还是双面光照模型的参数

GL_LINEAR_ATTENUATION: 指定光源具有线性衰减系数的参数

GL_POSITION:在四维齐次坐标中指定光源位置的参数

GL_QUADRATIC_ATTENUATION：指定光源具有二次衰减系数的参数 242
GL_SMOOTH：在glShadeModel()函数中选择平滑着色处理的参数
GL_SPECULAR：定义光源的镜面反射光部分或者材质的镜面反射光反射系数的参数
GL_SPOT_DIRECTION：表示值是用来定义聚光灯方向的参数
GL_SPOT_EXPONENT：表示值是用来定义来自聚光灯的光线的分布强度的参数
GL_SPOT_CUTOFF：表示值是用来定义聚光灯最大发散角的参数
GLU_FLAT：为GLU物体选择Flat着色处理的参数
GLU_SMOOTH：为GLU物体选择平滑着色处理的参数

6.19 思考题

这些问题覆盖了在周围环境中看到的光照处理和着色处理问题。这些有助于在OpenGL简单的局部光照模型中看到所用的不同光照，也有助于了解模型的一些局限性。

1. 在环境中，区分物体只显示环境光，只显示漫反射光以及只显示镜面反射光的例子。注意这些物体和直接光源的关系，得出物体与光源关系的结论：只显示环境光，同时显示环境光和漫反射光，以及只显示镜面反射光。观察镜面反射光，看它拥有的是物体的颜色还是光源的颜色。
2. 在环境中，区分物体高、中、低三个级别的镜面反射光。材质的什么属性使得这些物体展现出镜面反射光？
3. 在环境中，找出位置光源和方向光源的例子，讨论观察方式是如何影响场景中这两种光源的选择的。
4. 在环境中，选择一些由不同材质组成的物体，分别区分环境光、漫反射和镜面反射光属性，尝试用OpenGL的材质函数分别进行定义。
5. 在环境中，找出环境光照在不同的位置强度不同的例子。是什么因素导致了这样的结果？对于局部光照模型的精确度和全局光照模型进行比较时，说明了什么现象？

6.20 练习题

这些问题要求对建模和光照处理中一些非常重要的东西进行计算，然后从这些计算中得出结论。在本章的基础上，这些计算应该是非常直观的。

1. 给定一个三角形，它的顶点序列是$V_0 = (0,0,0)$，$V_1 = (5,0,5)$和$V_2 = (0,5,5)$。计算每个边的向量，然后对边向量进行叉积得到三角形的单位法向，也许利用三维技术对三角形进行绘制可以帮助了解。这个方法同样对下面的练习4和练习5有帮助。 243
2. 对由包含两个自变量的函数$f(x, y) = e^{xy}\sin(ax)\cos(by)$决定的曲面，分别计算它在$x$和$y$方向上的偏导数。对于点$P = (1,2,3)$，计算两个方向上的切线，然后利用叉积计算曲面上点$P$的单位法向。
3. 对一些特殊的物体，它们的表面法向计算非常简单，请说明如何方便地计算平面、球面和圆柱面的法向。
4. 回到练习1，把顶点V_0和V_2进行互换，然后再次进行计算。观察新计算的单位法向和原来单位法向的关系。
5. 在练习1中，多边形的朝向决定了表面法向的方向。它影响漫反射和镜面反射光公式中点积的符号。对练习1中的向量，假设有个光源位于点(5,5,5)，分别用这两个法向计算光源的漫反射和镜面反射光分量。哪个朝向有意义？对几何体的定义采用不同的顶点顺序意味着什么？
6. 在漫反射光的计算公式中，假设光源的能量是1个单位。利用漫反射光公式计算不同角度从法向反射的能量，用标准的角度±30°，±45°和±60°。同样，计算单位面积表面以不同角度投影到平面上的面积。然后，计算能量和投影面面积的比率，这应该是个常量。利用这些计算结果以及本章讨论的漫反射光照计算，描述一下为什么会这样。

7. 在镜面反射光公式的基础上，找到角Θ，使得镜面反射光的能量是原光照能量的50%。写下这个公式。看看这个公式如何随着镜面反射光系数（光泽系数）N变化的。
8. 对聚光灯的角衰减效果，定义一个聚光灯，它的光锥角是45度，光衰减系数是常量，然后用角衰减系数1、2和4以5度的步长计算从0度到45度的光能量，对于聚光灯的中心和边上的能量，这提供了什么信息？
9. 对光线的衰减，考虑OpenGL中的衰减公式：$A = 1/(A_C + A_L * D + A_Q * D^2)$。定义一些点，分别距离光源1、2、3、4和5个单位，把光源的各个系数都设为1，然后分别计算各个点的光照能量。计算下面三种情况下的衰减系数A_i：
   (a)只有常数衰减，$A_L = A_Q = 0$。
   (b)只有线性衰减，$A_C = A_Q = 0$。
   (c)只有二次衰减，$A_C = A_L = 0$。
   当观察用位置光源的时候，对于线性和二次衰减的效果，可以得出什么结论？
10. 对于第2章思考题中定义的旋转木马的场景图，如果光源是下面这几种情况，那么怎样放置在场景图中？
    (a) 光源在旋转木马的场景中间，
    (b) 在旋转木马底座的外圆边上，
244 (c) 旋转木马场景中马的头顶上。
11. 判别：对有光照的模型，假如平滑着色处理的模型对每个顶点都使用面向量，那么对于多边形，Flat着色处理和平滑着色处理是一样的。为什么你的答案是正确的？
12. 通过为每个顶点指定颜色定义一个三角形，对一个顶点的颜色随时间进行改变，产生平滑着色处理的动画效果。观察颜色在整个三角形表面变化的方式。通过选择一些颜色再做这个实验，看看变化方式是否对于所有的颜色都是一样的，或者对某些颜色表现的更加强烈。
13. 如果三角形很小（位于演染三角形内部的像素很少），那么对于使用Flat或者平滑着色处理有关吗？请说明理由。

6.21 实验题

1. 在讨论创建经过光照和着色处理的图像时，我们对启用一些属性或性质进行了讨论。为了真正理解它们的作用，编写一个使用光照和着色处理的程序，系统地禁用（不是启用）每个属性或性质，看看将发生什么。例如，禁用GL_NORMALIZE功能，看看没有单位法向将会发生什么。记录下这些结果，以便以后没有正确启用这个功能的时候可以认识到这个问题。
2. 我们曾讨论了一个原则，那就是如果用很小的多边形，那么Flat着色处理和平滑着色处理都将接近Phong着色处理。建立一个基于2D定义域的网格基的表面。尝试不同的网格粒度以及不同的表面着色处理的质量。估计每个像素一个多边形网格的近似验证了还是反驳了这个原则。
3. 考虑第3章遇到的gluCylinder函数。我们的目标是建立一个着色处理后的圆柱体，它的颜色从一端渐变到另一端。为了实现它，我们用一系列的四边形表示圆柱体。四边形的顶点在底面的圆周上。给圆柱体指定底面颜色，并为图像指定平滑着色处理。对每个四边形，为每个底面上的顶点指定适当的颜色，看圆柱体的着色属性是否跟所期望的四边形平滑着色处理一样。

   利用球面坐标或者其他在第2章或第3章的建模技术定义一个拼凑的半球面。自己设计这个半球面，以便可以从各个方面控制图形的分辨率，就像GLU和GLUT模型允许定义面和块。这个半球面将是实验题4～8的基础。
4. 对上述问题，尝试定义与现实世界相匹配的材质属性，并在半球面的图像中使用这些材质。看看它和实际的物体外观有多接近。如果这个近似不是很好，能不能发现原因并对材质定义进行改进？

5. 因为是对半球面进行处理，所以，很容易得到每个顶点的解析法向，然而，每个面片的法向和每个顶点的法向是不一样的，看看怎样才能得到半球面上每个面片的法向。 245
6. 用常规代码显示多边形模型，并用相对粗糙的定义来表示半球面，比较Flat着色处理（用面法向）和平滑着色处理（用顶点法向）的效果，比较用粗糙的平滑着色处理和用更精细粒度的Flat着色处理的效果。
7. 对于显示Flat着色处理，为每个面选择一个顶点法向来代替面法向，要选择相对于面的同一个顶点，比较各个方式下得到的视图，为什么比起面法向视图，顶点法向视图比较不准确？这种差别重要吗？
8. 因为显示代码允许对半球面进行旋转，并允许看到球面的内部和外部，所以可以考虑不同的光照模型：

```
GL_LIGHT_MODEL_TWO_SIDE
GL_LIGHT_MODEL_LOCAL_VIEWER
```

并为材质定义不同的表面参数：

```
GL_FRONT
GL_BACK
GL_FRONT_AND_BACK
```

尝试所有或者这些选项的大部分，对每个选项，注意图像产生的不同效果。

6.22 大型作业

1. （小房子）在第3章的项目部分建了所简单的房子，在房子的里面和外面增加一些光源，把涂了色的墙改成用合适的材质描述，看看先前的房子经过光照和着色处理后的样子。
2. （场景图分析器）在第3章的场景图分析器中添加光源信息，以便每个光源都可以通过适当的变换来放置，同时用光源节点存储每个光源的属性，以便分析器产生合适的光照信息。
3. （场景图分析器）在第3章的场景图分析器中添加外观信息。分析器要产生的代码应该包括适当的与前面的几何函数一样的材质定义函数，着色处理信息使几何体的每个面片都可以独立地应用Flat或者平滑着色处理。对场景图进行分析的时候，看看能不能对当前的材质定义进行跟踪，这样可以避免新的材质定义，因为当新的定义被读入的时候，它们是多余的。 246

第7章　事件和交互式编程

图形编程所包含的内容远比创建图像和动画更丰富。我们越来越认识到创建具有以下特点的应用程序的价值所在，这些应用程序能让用户与展示图形进行交互，允许用户控制图像的生成方法，或者通过与过程图形对象的交互来控制过程本身。这些可交互的图像在娱乐、教育、工程以及自然科学等各个领域具有广泛的应用价值。交互式计算机图形应用为我们提供了与图像进行交互的能力，这对于该领域的成功是至关重要的。

应用程序中与用户交互的部分称为用户界面，用户界面本身也是一门学科，它专门研究怎样设计和评价交互式应用程序的界面。但在本章，我们讨论的重点不是用户界面，而是图形交互技术。很多用户界面都采用图形方式来为用户提供信息，进行图形交互，再根据结果来控制程序的执行流程。我们把这种图形交互技术看作图形学的一部分。很多界面工具包都能和OpenGL一起使用，它们都有各自的优势。如果用户要开发一个真正的交互式应用程序，那么应该仔细挑选工具包，然后弄懂它。当然，有些工具包很复杂，要花一些精力去学习。

本书力求简洁，我们会选择一些容易找到并简单易用的界面工具包来支持本章的编程。在本章的结尾，将介绍针对OpenGL的界面工具包MUI（Micro User Interface），它使得用户能很容易在OpenGL程序中添加基本的界面。我们相信读者知道用户界面对用户正确理解图像具有重要作用。显然，对用户界面的系统介绍已经超出本章的范畴。我们的观点是计算机的交互界面应该由经过人机工程学、界面设计和评估等方面训练的专业人士来设计，而不应该由计算机科学工作者来设计。但是，计算机科学工作者可以根据设计方案来具体实施。本章将介绍实施图形交互的技术。

交互式计算机图形程序设计充分利用了现代计算机系统的事件处理机制，因此，必须了解什么是事件，怎样使用它们来写交互式图形程序。事件的概念相当抽象，事件的种类较多，因此，在逐步展开介绍这个概念时，我们会探究其中的细节。但是，现在的图形API处理事件的机制非常简洁，你会发现一旦熟悉了事件处理机制，写基于事件驱动的图形程序其实并不困难。一些基础的图形API本身不包括事件处理机制，所以，为了使用它，需要对这些API进行扩展。

在学完本章以后，读者将会了解一个标准的图形API提供的交互功能，也能够用合适的事件驱动工具编写交互式图形程序。

7.1　定义

事件是计算机系统控制状态的转换。事件可以从多个源头产生，当系统响应这个状态转移时，能引发一些计算机行为。我们把事件看作用来设计交互式程序的一个抽象的概念，它为计算机系统提供具体明确的数据。事件记录是某种系统行为的正式记录，这种行为通常来自一个计算机输入设备，比如鼠标或键盘。这个记录的数据结构包含用来区分不同事件的标记以及与该事件相关联的数据。这个数据结构对于程序员来说是不能直接访问的，但它的值会通过系统函数返回给系统和应用程序。比如说，一个键盘事件记录包括被按下键的标识和键被按下时光标的位置；如果鼠标键被按下，鼠标事件记录包括被按下鼠标键的标识，以及在事件发生时，光标指针相对屏幕的位置。

事件记录在事件队列中保存，由操作系统来管理。事件队列保存事件发生的顺序，并作为这些事件的处理进程的资源。当一个事件发生时，它对应的事件记录加入相应的事件队列，称为事件记入队列。随着事件到达队列的前端，当进程请求事件记录时，操作系统会把记录分发给与之对应的进程，如图7-1中所示。一般来说，和屏幕位置相关事件会分发给涉及这个位置的程序。而发生在程序窗口外面的事件就不会分发到该程序。



图7-1 事件被发送到事件队列(左)，操作系统找到事件并交给相应的进程进行处理(右) 248

和大部分交互式程序一样，使用事件控制的程序通常通过函数来管理控制，这个函数叫做事件处理程序。事件处理程序可以通过多种方式访问事件队列，大部分API通过回调函数来处理事件：把一个回调函数和事件关联起来的过程叫做为这个事件注册该回调函数。当系统向程序传递一个事件记录时，程序会分辨事件的类别。如果某个回调函数已经注册给这个事件，程序控制权会交给这个回调函数。这种交互式程序包括初始化函数、动作函数、回调函数和一个主事件循环。这个主事件循环会调用事件处理程序来获得事件，找出处理相应事件的回调函数，然后把程序控制权交给该回调函数。当函数完成操作时，将控制权再交回给事件处理程序。

这种主事件循环的作用是直接的，程序把执行控制权交给事件处理程序，这样程序的控制权能够有效地被用户掌握。从这里开始，用户将通过已经创建的回调函数来响应事件。在本章，我们会看到很多通过这种方式实现的例子。

回调函数是当程序识别某个特定事件时执行的函数。这种识别是指事件处理程序将事件从事件队列中取出，程序曾经关注过这个事件。在程序中使用某个事件的关键就是通过注册事件对这个事件表示关注，并标识出当该事件发生时执行哪个函数。

7.2 事件的例子

事件通常可以根据引发该事件的动作的种类来分类。一种比较容易想到的分类方法就是根据传统的计算机设备来划分。即：

按键事件。比如*keyDown, keyUp, keyStillDown,*…按键事件记录了键盘的一个键被按下以及与该键对应的值。注意，有两种按键事件：一种是使用常规的键盘，另一种是使用“特殊键”，比如功能键或光标控制键。用不同的事件处理程序注册给不同类型的按键事件。需要特别注意，当使用特殊键时，不同计算机会有不同的特殊键，即使键相同也可能会有不同的布局。

菜单事件。比如从弹出式或者下拉式菜单或者子菜单中选择一项。这类事件取决于用户对菜单的定义和赋给菜单设备的值。

鼠标事件。比如*leftButtonDown, leftButtonUp, leftButtonStillDown,*…注意到不同“种类”的鼠标会有不同数量的按键，所以，对于某些鼠标来说，有些事件会用不同的方法处理，比如双击或shift+点击。

软件事件。这类事件是程序本身发送的，目的是让程序接下来执行一个特殊的操作，比如redisplay重显示事件。 249

系统事件，比如idle空闲事件和timer定时器事件。这类事件是由系统分别根据当前的事件队列状态或系统时钟产生的。

窗口事件，比如移动窗口或改变窗口大小。这类事件是由窗口的基本操作引起的。

这些事件非常具体，其中很多事件并没有包含在图形学常用的API和API扩展中。但是，当我们深入开发系统程序时，需要用到这些事件。

7.3 交互的方式和方法

交互可以有多种方式，但在计算机图形学中，交互通常是一类可视的操作，可以理解为一种通过视觉交流的交互。在本章后面部分将讨论交互视式觉交流方面的内容。这里我们重点讨论交互与各种计算机硬件设备和软件产生的事件之间的关系。

用户在使用应用程序时，总是把注意力集中在工作内容上，而不是设计和开发这个应用程序的过程。最好的应用程序就是能让用户感觉它是无形的，用户通常希望和应用程序及相应数据之间能够自然地、舒适地进行交流，让他们很容易做自己需要做的事。界面设计师的工作就是设计让人感觉很自然的界面，且界面不会妨碍用户的工作。界面设计和计算机图形学是不同的学科领域，但是界面设计有助于我们了解交互的方式和方法。

我们已经介绍了基于常用的计算机设备键盘或鼠标的交互。人们对这两种设备有着截然不同的操作方式。作为交互设备，键盘是一个离散的输入设备，根据敲击不同的键产生不同的操作。这些操作让用户做出抽象的选择，包括选择动作和选择对象。在简单的文字游戏中，控制光标漫游的键盘输入就是选择动作的一个例子。鼠标键也可以作为选择设备来使用，尽管它们的主要作用是选择和控制屏幕上的图形对象。由于没有规定鼠标上应该有多少个键的标准（Macintosh鼠标一般只有一个键，Windows鼠标一般有两个键，Unix鼠标一般有三个键），这样对于程序员编写包含鼠标交互的可移植程序是一个挑战。键盘和鼠标的键都是离散输入设备，只能提供有限个定义明确的计算机操作。

鼠标本身还有另一个用处，它可以提供连续的输入，能用来控制屏幕上的连续运动。它可以控制一个选择好的对象，把它移动到指定的位置，或者控制显示属性（比如平移或旋转）或者模型属性（比如某个点的温度）。鼠标事件也包含了鼠标键中某个或某些键被用户按下的信息。当你要为应用程序设计交互时，应该确定用户使用离散选择还是连续控制，然后根据用户期望交互方式和任务的需求，用键盘或者鼠标来具体实现这些交互。鼠标的类型有好多种，如图7-2所示，有时也会用到更多按键的鼠标。

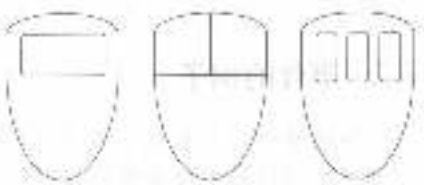


图7-2 各种不同的计算机鼠标，单键的、双键的和三键的鼠标

还有一些设备在计算机上尚未普及，但将来会很常见。比如，六自由度（6DF）操纵杆，它可以代替鼠标。鼠标运动只有4个自由度，沿X和Y轴，作正向和负向移动。鼠标设备能让我们控制显示在屏幕上的对象的运动，比如根据我们的视线方向，控制对象作或上或下、或左或右运动，但是它不能控制作向前或者向后运动。为了实现前后移动的控制，还需要两个自由度，这是鼠标无法做到的。然而，我们有其他的设备可以让用户同时实现向左向右、向上向下和向前向后的移动，比如图7-3中的6DF SpaceBall，这些设备可以用来控制虚拟环境，也适用于图形API和图形应用程序。



图7-3 六自由度设备SpaceBall 5000[TM]

7.4 对象选择

通过前面的内容我们看到，通过菜单、键盘敲击和鼠标功能等交互方式，图形API能将用户输入集成到应用程序中，这些输入操作先为选择的对象指定应完成的动作，然后操纵图像。如果需要在场景里识别某个对象，并赋予其一动作，就不能简单地靠鼠标点击来识别它，必须通过程序解析这个点击来完成对象选择操作。在这一节，我们会告诉你怎样选择场景中的一个对象。这样可以让用户用一种更直观的方式与场景交互，而不是和菜单选择或者键盘按下等事件交互。这些事件更抽象，而且和用户看到的信息不直接相关。通过对象选择操作，我们可以直观地做我们所熟悉的图形用户界面一样的操作。在图形用户界面中，用户可以选择一个图形对象，进而对它进行操作。概念上来讲，选择操作让用户用鼠标引导光标来识别对象，然后当光标仍在对象上时通过鼠标点击选择该对象。程序必须能识别所选择的对象，这样才能对该对象进行用户指定的操作。

为了理解选择操作是怎样实现的，我们从鼠标点击开始介绍。当一个鼠标事件产生时，从事件回调函数得到4个信息：被按下鼠标键值，该键的状态，以及事件发生时屏幕上鼠标位置的整数坐标。为了识别通过鼠标点击所指定的对象，我们把窗口坐标转化为三维眼坐标，然后作投影反向，回到三维眼空间。这样的话，一个二维眼坐标空间的点在三维眼坐标空间变成了一条线，如图7-4所示。这样我们的问题就变成了如何辨识和这个线段相交的所有对象，并区分这些对象中哪一个是被用户选中的。

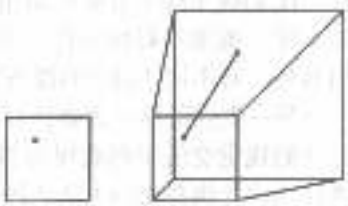


图7-4 三维眼空间中的一个点和与该点对应的视域体中的一条线段

我们需要进行几何对象的碰撞检测逻辑计算。对于场景中每一个对象，计算线段是否和该对象相交。当完成所有的计算，就可以分辨在选择点下面有哪些对象。每一个相交点在三维眼坐标中，我们可以选择离眼睛最近的对象，这也是用户在鼠标点击位置所直接看到的对象，也可以根据逻辑需要选择其他的相交点。然而，这种方法的计算量非常大，而且还需要回到眼空间中进行计算，这给具体实现带来了困难。

另一种识别线段上的对象的方法就是采用相反的逻辑。前一种方法的思路主要集中在怎样选择点下面的对象，我们发现所有和这条线相交的对象都会在绘制时用到选择的像素。所以，我们可以跟踪这个像素，并保存每一个用到这个像素的对象的信息，这样就可以识别光标位置下的所有对象。因为绘制过程记录了深度值，我们可以得到视域体中对象的深度信息，利用深度信息来识别对象。值得注意的是，系统可能保存了跟该点相关的其他信息，所以，我们可以考虑采用其他信息来识别对象。

对于前面介绍的两个技术，我们不知道某个特定图形程序应用了哪一种技术，也不知道某个特定的图形API采用了哪一种技术。但是，OpenGL采用的是第二种技术，在本章的后面部分，我们将介绍它是怎样实现的。

7.5 交互和视觉交流

这一章里，我们介绍如何设计和实现交互式图形应用程序，帮助用户理解显示图像的含义。这种交互就是用户和程序之间的一种交流，通过这种交流用户可以对图像施加影响，因此需要很好的视觉交流。这种交互也让用户可以通过操纵图像来理解图像要表达的信息。一个交互式应用程序需要考虑用户和程序之间是如何通过交互来进行交流的，这样用户才能最大限度地利用该交互方式。

一类常见的交互式应用是从不同的视点来观察整个场景或者观察某个对象。它能让用户在场景中漫游，以及放大或者缩小场景。在场景中漫游的另一种实现方式是在世界坐标空间旋转场景本身，而保持视点方向不变。无论采用哪种方法，都需要完成相对某个固定点的旋转操作。要么固定视点，要么固定场景，同时控制眼睛移向这个固定点或者远离这个固定点。这种旋转操作的结果改变了视点的经纬度位置，相当于沿垂直方向（改变纬度）和水平方向（改变经度）移动鼠标时可以看到的结果。运用鼠标可以很自然地控制旋转操作，当鼠标键按下时，鼠标的垂直移动转化为视点的纬度方向移动，鼠标的水平移动转化为视点的经度方向移动，鼠标的这种使用方式在应用程序中很常见，也是你熟悉的使用方式。应用鼠标可以实现的另一种控制就是场景的缩放，这是一种鼠标的一维运动就可以实现的操作。通过按下另一个鼠标键，用鼠标的水平移动来完成这种操作，尽管这样会使用户感觉操作容易混淆，同样的一个动作按下不同的键却会有不同的操作含义。实现场景缩放的另一种方法就是使用键盘，比如按下和键来控制用户在场景中作向前或者向后移动。这种方法在英语环境中很容易实现，也能很好地工作；如果用户使用另一种语言，效果就可能不会很好。当然，你可以用另外一对不同于和的键来完成这些操作，同样会有很好的效果。

另一类应用程序需要针对图像的某一部分进行部分图像操作，例如选择，并作相关的操纵。这样的视觉交流问题表现为创建选择操作，选择图像的一部分，指示该部分图像已经被选中，
253 并向用户提供自然的方法去操纵它。一种指示对象被选中的方法就是当它指向可以选择的对象时改变光标的形状。在图形API支持下，我们可以使用被动鼠标移动模式和设置新的光标形状来实现。当可选择的对象在视图中总在同一个位置时，这种方法最有用；如果鼠标形状不能改变，那么可被选择和不可被选择的对象的显示方法应该有所不同，或者至少有标签或者图例表示哪个是可选择的。实际的选择操作可能是一个鼠标点击，对象的操纵需要根据具体应用的需要来选择。我们前面介绍了用高亮颜色来表示对象被选中，我们也介绍了一维和二维操纵的一些方法。菜单可以帮助用户完成更复杂的操纵，建议专家级用户不妨考虑采用菜单快捷键。

7.6 事件和场景图

当我们设计程序的交互操作时，通过场景图来设计事件处理是非常有效的。可以将事件和交互一并纳入场景图中，来管理所有的交互操作。场景图有四种节点：组节点、变换节点、几何节点和外观节点。交互可以对大部分节点产生作用：

- 我们可以改变变换矩阵实现在场景中移动对象或者改变视点等交互功能。
- 我们还可以改变外观属性来指出哪个对象被选中，或添加或删除对象，以及给场景一个全新的外观。
- 我们还可以改变几何节点来为对象选择合适的几何信息，或者用另一个对象或者对象的集合来替换一个对象。

所以，有很多途径将事件和交互与场景图中的节点联系起来。

事件影响场景图的一个重要途径是，在选择一个特定的对象或者对象集合时，赋于它特定的处理方式。包括是否运动（变换），或外观，或表示处理。这种操作需要触发特定对象去执行特定操作，这样就需要在各种场景图节点中配置触发器，这些触发器由选择操作来激活。这些超出了场景图原始的建模功能，但是，我们可以通过在节点前或在节点数据结构中设置一个布尔选项来实现上述功能。

7.7 建议

本节讨论通过事件处理来实现交互的机制，但是它不包括为交互式应用程序创建自然的

用户控制方式的内容。如何设计一个用户界面涉及到很多深入和细节的问题，这里没有讨论它们。用户界面领域的大量文献将帮助你在这个领域起步，但是一个专业的应用程序需要一个专业的界面、一个由这个领域的专业人士设计、测试以及改进的用户界面。

下面的几个例子会尽力介绍不是太复杂的用户控制机制，但它们设计的重点是在事件和回调函数，而不是用户使用的方法。当你写自己的交互程序时，应该更多地思考怎样让用户感受他们的任务，而不是怎样使自己最容易编程的方法。

7.8 OpenGL中的事件

OpenGL API中一般通过Graphics Library Utility Toolkit GLUT（或者类似的扩展）来实现 254
对事件和窗口的处理。GLUT定义了许多事件，并提供给程序员一系列的回调函数与之相对应。在有GLUT扩展的OpenGL中，主事件循环显式地调用函数glutMainLoop()，把它作为主程序的最后一个语句。

7.9 回调函数的注册

下面我们列举了一些OpenGL的事件，对于每一个事件给出相应回调函数的注册函数。在后几节中我们用一些示例代码说明注册和使用这些事件来实现某些功能。

事件	回调函数的注册函数
idle	glutIdleFunc(functionname)需要一个回调函数作为参数。该回调函数具有形式void functionname(void)。这个函数是一个事件处理程序，决定每一个空闲周期做什么。通常，这个函数的结束是调用glutPostRedisplay()（在下面介绍这个函数）。这个函数定义程序在没有其他事件处理时做什么操作，它通常用来驱动实时的动画。
display	glutDisplayFunc(functionname)需要一个回调函数作为参数。该回调函数具有形式void functionname(void)。这个函数也是一个事件处理程序。无论显示事件何时收到，它会产生一个新的显示命令。要注意，这个显示函数将被事件处理程序调用，无论这个显示事件何时到达。这个事件是由glutPostRedisplay()函数发送的，当窗口被打开、移动或者调整时发送。
reshape	glutReshapeFunc(functionname)需要一个回调函数作为参数，该回调函数具有形式void fucntionname(int, int)。这个函数管理任何观域设置的改变，以适应新调整的窗口和新的投影定义。reshape回调函数的参数是窗口改变以后的宽度和高度。
keybourd	glutKeyboardFunc(functionname)需要一个回调函数作为参数，该回调函数具有形式void functionname(unsigned char, int, int)。这个带参数的函数是处理键盘被按下的事件处理程序。接收哪些字符被按下以及当前的光标位置(int x, int y)。与所有包括屏幕位置的回调函数一样，该函数的屏幕位置会转化为相对窗口的坐标。同样，这个函数的结束是调用glutPostRedisplay()来重新显示 255 其个特定键盘事件引起改变的场景。
special	glutSpecialFunc(functionname)需要一个回调函数作为参数，该回调函数具有形式void functionname(int key, int x, int y)。这

个事件产生于那些“特殊键”被按下时，这些键包括功能键、方向键和一些其他键。第一个参数是被按下的键值，第二和第三个参数是窗口整型坐标，它记录了当特殊键被按下时光标的位置。通常的做法是用一个特殊的符号名字来命名特殊键，这些将在后面的键盘回调函数例子中具体讨论。特殊键和普通键的回调函数之间的唯一不同，仅在于事件源于不同的键。

menu

int glutCreateMenu(funcitonname)需要一个回调函数作为参数，该回调函数具有形式void functionname(int)。传递给这个函数的是一个整型参数，它记录了当菜单打开并有选项被选中后，所选中的菜单项对应的编号。在后面的例子中，我们会继续介绍菜单项是怎样和这些整数进行关联的。

glutCreateMenu()函数会返回给菜单一个确定的ID。在后续操作中可以根据该ID改变菜单选项。这些操作将在后面介绍怎样操纵菜单时讨论。glutCreateMenu()函数会根据鼠标键被按下的事件创建一个菜单，这个事件由glutAttachMenu(event)函数指定，这个函数将当前菜单和特定的事件联系起来。glutAddMenuEntry(string, int)函数设定了菜单中每一个选项，然后给每个选项定义一个返回值，也就是当用户选择某个带有字符串标签的菜单项时，这一返回值作为参数返回给菜单回调函数。菜单项的设置要放在菜单被关联之前，如下面的代码所示：

```
glutAddMenuEntry("text", VALUE);
...
glutAttachMenu(GLUT_RIGHT_BUTTON)
```

glutAttachMenu()函数表示创建菜单过程的完成。

和菜单一样，也可以创建子菜单，选中菜单的菜单项可以显示菜单，即级联子菜单。子菜单可以通过两种方式创建，我们介绍用glutAddSubMenu(string, int)函数加入子菜单的方法，这里的字符串代表显示在该菜单中的文字，整数代表级联子菜单的主菜单项的标识ID。当主菜单中该字符串选项被选中时，子菜单就会弹出显示。在GLUT函数里，只能在菜单项最后一项添加子菜单，因此，添加了菜单就会结束整个主菜单的设置。在本章的后面部分，我们将介绍如何在一个菜单中添加更多的子菜单。

256

mouse

glutMouseFunc(functionname)需要一个回调函数作为参数，回调函数具有形式void funcitonname(int button, int state, int mouseX, int mouseY)。这里button代表哪一个键被按下（每一个鼠标键代表一位整数，那么一个三键鼠标的按键状态可以表示1～7的任何数），state表示鼠标的状态（用有含义符号标记的值表示鼠标的状态，比如GLUT_DOWN），无论按下和放开按键都会产生事件，还有整型值xPos和yPos来记录当事件发生时光标在窗口中的坐标。

如果鼠标键定义为用来触发一个菜单，则鼠标事件将不会使用该回调函数。

mouse active motion

glutMotionFunc(functionname)需要一个回调函数作为参数，该回

调函数具有形式void functionname(int, int)，这两个整型参数代表了该事件发生时光标相对窗口的坐标。这个事件是鼠标在一个或多个鼠标键被按下的状态下移动时产生。

mouse passive motion　glutPassiveMotionFunc(functionname)需要一个回调函数作为参数，回调函数具有形式void functionname(int, int)。这两个整型参数代表了事件发生时光标相对窗口的坐标，这个事件是鼠标在鼠标键放开（即没被按下）状态下移动时产生。 257

timer　glutTimerFunc(msec, funtionname, value)需要一个整型参数(msec)来表示经过多少毫秒回调函数被触发，一个回调函数参数，回调函数具有形式void functionname(int)，还需要一个整型参数(value)作为调用该回调函数时的传入参数。

在上面的任何一种情况下，回调函数名可以是NULL函数。这样可以为代码创建模板注册系统支持的所有事件，也可以方便地把NULL函数赋给任意一个要忽略的事件。

除了常见的设备事件之外，还有一些软件事件，比如display事件（通过调用glutPostRedisplay()来创建）、idle事件和timer事件。还有一些在大学实验室不经常见到的设备事件，比如图7-3中的SpaceBall、绘图板——一种计算机辅助设计中常见的设备，在很多应用中仍然用到。如果要了解更多的关于这些设备的处理方法，请查阅GLUT手册。

7.10 实现细节

对于大部分OpenGL回调函数来说，事件回调函数中参数的含义可以很容易从字面含义来理解。它们大部分都是标准字符或者整数，比如窗口尺寸或者光标位置。但是，对于特殊键事件的回调函数，就需要特殊的有含义的字符来标记它们，其中大部分都容易理解，但有些比较难懂。完整的特殊键和参数对应表格如下：

F1到F12的功能键：	GLUT_KEY_F1 到 GLUT_KEY_F12
方向键：	GLUT_KEY_LEFT, GLUT_KEY_UP,
	GLUT_KEY_RIGHT, GLUT_KEY_DOWN
其他特殊键：	GLUT_KEY_PAGE_UP (Page up)
	GLUT_KEY_PAGE_DOWN (Page down)
	GLUT_KEY_HOME (Home)
	GLUT_KEY_END (End)
	GLUT_KEY_INSERT (Insert)

在使用这些特殊键时，要用这些有含义的符号名来处理这些键值，这些键值返回给特殊键回调函数。

timer事件与其他任何事件都不一样，它是一个“注册一次作用一次”的事件。当注册了一个定时器事件回调函数，事件在系统时钟走到预先定义好的时间时就会激活，如果不重新注册这个回调函数，定时器事件不会再次发生。可以对不同激活时间的事件分别注册不同的回调函数。这样，当要驱动一系列固定时间间隔的事件时，需要在每次定时器事件发生时重新注册这个事件。我们会在后面给出具体定时器代码例子时详细讨论。 258

创建和操纵菜单

菜单是交互式编程的关键部分，GLUT和计算机的窗口系统结合在一起为程序提供一个设

备无关的菜单实现方法。这些菜单会遵循窗口系统的约定，对Windows和Unix操作系统，菜单是弹出式的，而对于Macintosh系统，既可以是弹出式的，也可以是附加在屏幕上方的菜单条上。这种处理方式避免了复杂的系统层编程。在讨论如何创建菜单时，可以参考与图7-5所示菜单对应的代码片段。

要创建一个菜单需要调用glutCreateMenu(...)函数，把菜单对应的菜单事件回调函数名传给它。这个回调函数会得到菜单项的值，这个值通过glutAddMenuEntry(name, value)函数和每一个菜单项的名字联系起来。创建菜单的函数会返回一个整数，这个整数就是菜单名，代表这个菜单。当你创建好一个菜单，可以通过glutSetMenu(int)函数来激活菜单，也可以通过glutAddMenuEntry(...)函数来添加菜单项。这些函数需要一个字符串和一个整数作为参数，菜单显示这个字符串，当该字符串被选中时返回前面设定的整数。最后，当你添加完所有的菜单项时，就可以把菜单绑定到特定的鼠标键上，这样，点击该键会调出这个菜单。你现在已经创建好一个菜单，可以给它写回调函数了，这个回调函数的输入是从菜单选项返回的值，完成你要做的操作。

菜单是一个复杂的资源，它需要更多的处理能力，而不是对程序的动态信息的简单响应。在OpenGL中，菜单可以被激活，也可以取消激活；可以被创建，也可以被销毁；菜单项可以被添加，删除或修改。这类菜单操作的基本工具可以在GLUT软件包中找到，在本节中我们将介绍这些操作。

当定义一个菜单时，调用glutCreateMenu()函数返回一个整型值，这个值叫做菜单编号。当创建一个菜单时，这个菜单就是当前的活动菜单。如果创建一个以上的菜单，需要根据菜单号来激活特定的菜单，以便做进一步操作。要查看任何时候当前活动的菜单编号，可以用这个函数：

```
int glutGetMenu(void)
```

它返回一个菜单编号。如果要操作另一个菜单，就需要改变当前活动菜单，可以用新的菜单编号作为下面这个函数的参数：

```
void glutSetMenu(int menu)
```

它把菜单编号传递给活动菜单。这样我们前面介绍的操作就在此基础上实现。要注意，主菜单和子菜单都有相应的菜单编号，所以追踪这些菜单号很重要。

菜单可以是动态的，它随着程序的运行而改变。可以通过下面的函数改变任何菜单项的显示字符串及其返回值：

259

```
void glutChangeToMenuEntry(int entry, char * name, int value)
```

这里的name是要显示的新字符串，value是新的值。这个值是在事件处理程序处理该菜单项被选中时返回给系统的整型值。将要改变的菜单是活动菜单，可以按照前面介绍的方法对活动菜单进行设置。

当用glutAddSubMenu()函数对主菜单只添加一个子菜单时，你可以用下面的函数在后面添加新的子菜单：

```
void glutChangeToSubMenu(int entry, char * name, int menu)
```

这里的entry是当前的菜单编号（第一个菜单项的编号为1），把这项变成一个子菜单的触发器，name是显示在那个位置的新字符串，menu是给新的子菜单的编号。这样就可以在菜单的任何位置添加你想添加的子菜单。

菜单也是可以销毁的。GLUT函数

```
void glutDestroyMenu(int menu)
```

将错误传递给该函数的整数参数所对应的菜单。

下面这段代码在菜单中添加菜单项，结果如图7-5所示。这个菜单也包含一个子菜单，子菜单相关的代码也在这段代码中。

```
mymenu = glutCreateMenu(submenu);
mainmenu = glutCreateMenu(menu);

glutSetMenu(mainmenu);
glutAddMenuEntry("FullScreen Mode",1);
glutAddMenuEntry("Windowed Mode",2);
glutAddMenuEntry("----------                                    ----------",0);
glutAddMenuEntry("Toggle Lighting",3);
glutAddMenuEntry("Toggle Smoothing",4);
glutAddMenuEntry("Toggle Wireframe",5);
glutAddMenuEntry("Toggle Axes",6);
glutAddMenuEntry("Toggle Texture",7);
glutAddMenuEntry("Toggle Environment Mapping",8);
glutAddMenuEntry("----------                                    ----------",0);
glutAttachMenu(GLUT_RIGHT_BUTTON);
glutAddSubMenu("Function",mymenu);
glutSetMenu(mymenu);
glutAddMenuEntry("z = [empty set]/0",0);
glutAddMenuEntry("z = 0/1",1);
glutAddMenuEntry("z = x*y/2",2);
glutAddMenuEntry("z = sin[x-y]/3",3);
glutAddMenuEntry("z = sin[y*cos[x*y] div sin[x*y]]/4",4);
glutAddMenuEntry("z = cos[sqrt(x*x + y*y)]/5",5);
glutAddMenuEntry("z = [-cos[sqrt(x*x + y*y)]]/6",6);
glutAddMenuEntry("z = x*x-y*y/7",7);
glutAddMenuEntry("z = 1 div -[x*x+y*y]/8",8);
glutAddMenuEntry("z = sin[sqrt[x*x + y*y]] div x*y/9",9);
```

260

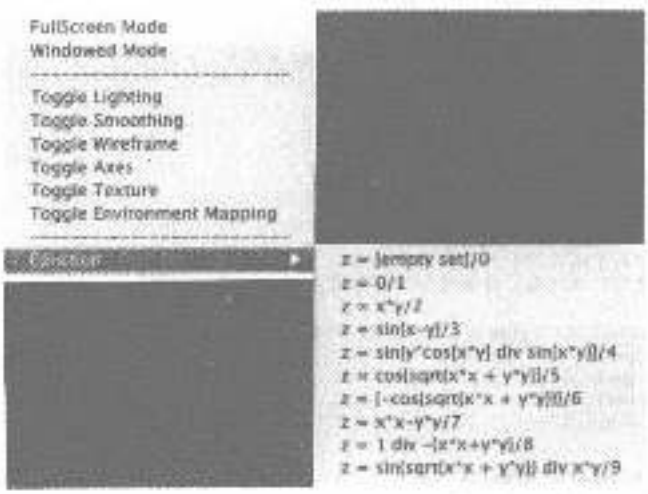


图7-5 带有子菜单的菜单

除非你需要在程序运行时改变菜单，否则上面这段代码可以直接应用。当你需要改变菜单时，你会发现GLUT软件包的功能已经足够完成任务。

7.11 代码实例

这节中我们将介绍四个简单的例子。第一个例子是一个简单的动画，它利用空闲事件让一个立方体作绕圈运动，时而在半径之内，时而在半径外，时而上，时而下。用户无法控制它的运动，当你编译并运行这段程序时，看你能不能想象出该立方体运动空间的体积。除了用空闲事件来驱动外，这个例子也包括了用定时器事件驱动同样操作的解决方法，可以有趣

地看到由于采用事件的不同，带来代码的不同。

第二个例子是用键盘回调函数使用户通过键盘的几个键来控制立方体的上下、左右和前后移动。这里用的是主键盘区的键，而不是特殊键，比如数字小键盘键或方向键。我们不采用数字小键盘的原因是某些键盘不带数字小键盘，我们不用方向键是因为我们需要控制六个方向，而不仅仅是四个方向。

第三个例子是用鼠标回调函数实现一个弹出式菜单，同时允许对菜单项进行选择以设置立方体的颜色。这个例子演示一个很简单的操作，但代码详细给出了前面所介绍的菜单实现
261 细节和一些补充。

最后，第四个例子用带有对象选择功能的鼠标回调函数，选择所显示的两个立方体中的一个，进而改变它的颜色。同样这也不是一个很复杂的动作，它使用了本章后面讨论的完整的选择缓存的处理过程。现在，我们建议先把重点放在事件和回调函数的概念上，读完本章后面介绍选择操作内容以后，再回过头来理解这个例子。

7.11.1 空闲事件回调函数

在这个例子中，假设函数cube()可以画一个简单的立方体，边长是2.0，中心在原点(0, 0, 0)。随着时间的推进，通过改变坐标来移动这个立方体，这样，让idle事件回调函数来设置立方体新的坐标，然后发送一个redisplay事件。显示函数将根据idle回调函数中设置的坐标画出新的立方体。大部分的实现代码已经被省去，我们用下面的代码片段来说明回调函数的注册、显示函数和空闲事件回调函数之间的关系。

```
#define deltatime 0.05
GLfloat cubex = 0.0,cubey = 0.0, cubez = 0.0, time = 0.0;

void display(void) {
    glPushMatrix();
    glTranslatef(cubex, cubey, cubez);
    cube();
    glPopMatrix();
}
void animate(void) {
// 立方体位置设置由基于事件的行为建模来设置
// 使时间的常量与时间相乘，并观察行为的变化

    time += deltatime; if (time > 2.0*M_PI) time -= 2*0*M_PI;
    cubex = sin(time);
    cubey = cos(time);
    cubez = cos(time);
    glutPostRedisplay();
}
void main(int argc, char** argv) {
// 在以下函数之前是标准的GLUT初始化
    ...
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutIdleFunc(animate);

    myInit();
    glutMainLoop();
}
```

7.11.2 定时器事件回调函数

定时器回调函数可以按照设置的时间表来驱动程序的动作。这个回调函数可以在程序的
262 任何一步进行注册，定时器的计时从它被注册的那一刻开始。可以灵活地运用这个功能，比如，可以通过回调函数的注册来设置回调函数的参数以便控制它自身完成的功能。在我们的

例子中，我们用定时器回调函数来替代空闲回调函数管理立方体移动的动画过程。下面的代码用定时器回调函数实现上述用空闲回调函数实现的功能，并允许控制动画过程的节奏。设定合适的延迟时间可以使动画在快速的系统上不至于太快。要注意，回调函数注册下一个定时器事件，这条注册代码应该被在定时器回调函数的最前面，函数中的建模时间正好用作时间延迟，这样就可以更好地控制帧与帧之间的时间间隔。

```
#define frameDelay 33
#define dTime 0.05

void timer(int i) {
    aTime += dTime; if (aTime > 2.0*PI) aTime -= 2.0*PI;
    cubex = sin(2.0*aTime);
    cubey = cos(3.0*aTime);
    cubez = cos(aTime);
    glutTimerFunc(frameDelay, timer, 1);
    glutPostRedisplay();
}

int main(int argc, char** argv) {
    ...
    glutTimerFunc(frameDelay, timer, 1);
    ...
}
```

7.11.3 键盘回调函数

我们还是从熟悉的cube()函数开始，这次我们让用户通过简单的键盘按键来控制立方体的上下、左右和前后移动。我们在键盘上定义了两个虚拟小键区：

```
Q W         I O
  A S         J K
    Z X       N M
```

小键区的最上面一行控制上下移动，中间行控制左右移动，最下面的行控制前后移动。举个例子，对于第一行来说，如果用户按下了Q或者I键，立方体向上移动；按下W或O，它向下移动。大写和小写字母输入都可以。其他行以相类似的方式工作。读者当然可以针对自己程序的需要来设定不同的小键区或者不同的控制模式。

我们再次省略了大部分的实现代码，但显示函数和上一个例子的工作方式类似：事件处理程序设置全局的位置变量，显示函数根据用户的选择对模型做相应的平移。注意在这个例子里，平移操作是沿立方体面的方向进行，而不是相对窗口的方向。

```
GLfloat cubex = 0.0;
GLfloat cubey = 0.0;
GLfloat cubez = 0.0;
GLfloat time  = 0.0;
void display( void ) {
    glPushMatrix();
    glTranslatef(cubex, cubey, cubez);
    cube();
    glPopMatrix();
}
void keyboard(unsigned char key, int x, int y) {
    ch = ' ';
    switch(key)
    {
        case 'q' : case 'Q' :
        case 'i' : case 'I' :
            ch = key; cubey -= 0.1; break;
        case 'w' : case 'W' :
        case 'o' : case 'O' :
            ch = key; cubey += 0.1; break;
        case 'a' : case 'A' :
        case 'j' : case 'J' :
            ch = key; cubex -= 0.1; break;
```

```
        case 'S' : case 'S' :
        case 'K' : case 'K' :
            ch = key; cubex += 0.1; break;
        case 'Z' : case 'Z' :
        case 'N' : case 'N' :
            ch = key; cubez -= 0.1; break;
        case 'X' : case 'X' :
        case 'M' : case 'M' :
            ch = key; cuber += 0.1; break;
    }
    glutPostRedisplay();
}
void main(int argc, char** argv) {
/*          标准GLUT初始化                */
    glutDisplayFunc(display);
    glutKeyboardFunc(keyboard);

    myinit();
    glutMainLoop();
}
```

类似的函数glutSpecialFunc(...)可以用来读取键盘上特殊键的输入，我们前面讨论特殊键事件时已作过介绍。

7.11.4 菜单回调函数

在前面讨论菜单时，我们仅用一些代码片段，而不是一个完整的例子程序来介绍所有的想法。在这个例子中，我们同样从cube()函数开始，但这一次，我们不需要让立方体做运动。我们只是定义一个菜单能选择立方体的颜色。当选择了颜色后，新颜色将应用到立方体上。这个例子只用一个静态的菜单。因此，glutCreateMenu(...)函数返回的值将被main()函数忽略。

264

```
#define RED 1
#define GREEN 2
#define BLUE 3
#define WHITE 4
#define YELLOW 5

void cube(void) {
    ...
    GLfloat color[4];

//  根据菜单选择设置颜色
    switch(colorName) {
        case RED:
            color[0] = 1.0; color[1] = 0.0;
            color[2] = 0.0; color[3] = 1.0; break;
        case GREEN:
            ...; break;
        case BLUE:
            ...; break;
        case WHITE:
            ...; break;
        case YELLOW:
            ...; break;
    }
// 绘制立方体
    ...
}
void display(void) {
  cube();
}
void options_menu(int input) {
    colorName = input;
    glutPostRedisplay();
}
void main(int argc, char** argv) {
    ...
```

```
    glutCreateMenu(options_menu);          // 创建选项菜单
    glutAddMenuEntry("Red", RED);          // 1 增加菜单项
    glutAddMenuEntry("Green", GREEN);      // 2
    glutAddMenuEntry("Blue", BLUE);        // 3
    glutAddMenuEntry("White", WHITE);      // 4
    glutAddMenuEntry("Yellow", YELLOW);    // 5
    glutAttachMenu(GLUT_RIGHT_BUTTON, "Colors");

    myinit();
    glutMainLoop();
}
```

265

7.11.5 鼠标移动的鼠标回调函数

在这个例子中，我们将用回调函数来处理鼠标的点击和鼠标移动事件。在使用鼠标键按下作为控制鼠标移动位置时都要用到这些事件。这些代码可以在图形程序中找到。比如用户需要按住鼠标键并在窗口中移动光标，程序需要根据鼠标的移动在窗口中移动和旋转场景作为响应。这个过程是这样完成的：先通过鼠标键回调函数mouse(...)得到鼠标键被第一次按下时的位置信息，然后从鼠标移动回调函数motion(...)中得到更新的位置信息。鼠标移动回调函数不断地跟踪当前鼠标位置和起始鼠标点击位置之间的距离，然后用这个距离通过计算全局变量来设置场景的旋转，以便能正确显示该操作。在计算距离时要格外小心，以得到正确的正负符号，这样鼠标移动才能符合你想要的场景运动。下面的代码段使用整型坐标spinX和spinY来控制旋转，除此之外这个坐标还可以用作许多用途。应用程序代码本身没有在下面给出。

```
float spinX, spinY;
int curX, curY, myX, myY;

void mouse(int button, int state, int mouseX, int mouseY) {
    curX = mouseX;
    curY = mouseY;
}

void motion(int xPos, int yPos) {
    spinX = (GLfloat)(curX - xPos);
    spinY = (GLfloat)(curX - yPos);
    myX = curX;
    myY = curY;
    glutPostRedisplay();
}

int main(int argc, char** argv) {
    ...
    glutMouseFunc(mouse);
    glutMotionFunc(motion);

    myinit();
    glutMainLoop();
}
```

7.11.6 对象拾取的鼠标回调函数

这个例子比前面几个例子都要复杂，因为对象选择中的鼠标事件的使用包括了一些复杂的步骤。我们从一段简单的代码开始，同样用我们熟悉的cube()函数来创建两个立方体，然后用鼠标选择其中的一个。当我们选择一个立方体时，两个立方体会交换颜色。

在这个示例代码中，我们从下面的函数开始介绍：一个完整的Mouse(...)回调函数，render(...)函数（该函数将两个立方体注册到对象名称列表），还有DoSelect(...)函数。它在GL_SELECT模式下管理场景的绘制，以及在鼠标事件发生时分辨哪一个（些）对象被鼠标所在的位置选中。这里我们没有给出注册这些鼠标回调函数的代码，因为在前面的例子中已经介绍过。

266

在这个例子之后我们介绍拾取的概念，并对这个例子进行扩展。一个更通用的DoSelect()

函数实现将在稍后给出。

```
void Mouse(int button, int state, int mouseX, int mouseY) {
    if (state == GLUT_DOWN) { /*       本例每个物体被选择       */
        hit = DoSelect((GLint) mouseX, (GLint) mouseY);
    }
    glutPostRedisplay();
}
...
void render(GLenum mode) {
//  即使在GL_SELECT模式下也总是绘制两个立方体，但是只有当一个物
//  体名称列表中被标识出来
//  GL_SELECT模式下绘制时才是可选择的
    if (mode == GL_SELECT)
        glLoadName(0);
    glPushMatrix();
    glTranslatef(1.0, 1.0, -2.0);
    cube(cubeColor2);
    glPopMatrix();
    if (mode == GL_SELECT)
        glLoadName(1);
    glPushMatrix();
        glTranslatef(-1.0, -2.0, 1.0);
    cube(cubeColor1);
    glPopMatrix();
    glFlush();
    glutSwapBuffers();
}
...
GLuint DoSelect(GLint x, GLint y) {
    GLint hits, temp;

    glSelectBuffer(MAXHITS, selectBuf);
    glRenderMode(GL_SELECT);
    glInitNames();
    glPushName(0);

//  设置视图模型
    glPushMatrix();
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
//  根据选择的
//  x和y值以及视口信息设置用于帮助拾取物体的矩阵

    gluPickMatrix(x, windH - y, 4, 4, vp);
    glClearColor(0.0, 0.0, 1.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    gluPerspective(60.0,1.0,1.0,30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
267 //  视点位置，观察中心和向上方向
    gluLookAt(7.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

    render(GL_SELECT); // 绘制用于拾取的场景

    glPopMatrix();
//  查询命中记录点数量，并重置绘制模式
    hits = glRenderMode(GL_RENDER);
//  重新初始模型为GL_MODELVIEW模式
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0,1.0,1.0,30.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(10.0, 10.0, 10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
//  如果存在的话，返回选择物体的标识
    if (hits <= 0) {
        return -1;
    }
//  根据选择结果对系统进行变更
    ...
    return selectBuf[3];
}
```

7.12 拾取的实现细节

OpenGL有多种通过鼠标事件选择识别对象的方法，在本章我们介绍其中的两种方法。第一种方法采用不改变颜色缓存的绘制模式，并跟踪覆盖选中像素或者选中像素周围小区域的所有对象，称为标准的选择方法。在第二种方法中，用互不相同的合成颜色标注场景中可以被选择的对象，通过检查选择点在颜色缓存中的像素颜色来识别离该点最近的对象。我们先来介绍第一个标准的选择方法。

标准的选择方法用“不可见的”画图方法，它其实并不在颜色缓存中写入任何数据，但它会记录所有绘制该选择点的对象。这种方法引入了画图法的*Render*（绘制）模式和*Selection*（选择）模式的概念。用标准方法生成图像时，用GL_RENDER模式画场景。这也是默认的画图模式。鼠标事件之后执行的鼠标事件回调函数中，将绘制模式变成GL_SELECT模式，并重画场景中每一个带有唯一名称的感兴趣单元。当场景在GL_SELECT模式重画时，实际上没有任何信息写到颜色缓存中，但是那些被重画的像素会被分辨出来。如果任何有名称的对象在重画时涵盖了鼠标选择的像素，它们的名称就被放到叫做选择缓存的数据结构中，实际上它是一个无符号整数栈，由名称来保存。选择缓存用命名的层次结构保存了所有命中的对象。当在这个模式下场景绘制完毕时，就生成了一张命中记录表，表中每一项对应一个对象的名称，该对象重画时涵盖了鼠标点击点。在系统切换回GL_RENDER模式时，这些命中记录数会返回给系统。这些命中记录的数据结构将在下一节中介绍。如果有命中，可以处理这张表来具体分辨哪些对象被命中，包括计算从视点开始到命中点有多少距离，也可以对这个信息做你需要做的任何事情。 268

这种“感兴趣单元”的概念比直接的选择更复杂，感兴趣单元可以包括一个对象、一组对象甚至一个具有层次结构的对象组。从接下来的一些例子程序可以看怎样对这些感兴趣单元命名。解决具体问题时尽可能地发挥你的创造力，你会惊奇地发现这种选择方法的功能非常强大。

7.12.1 定义

关于对象选择第一个需要理解的概念就是*名称栈*，这是一个选择模式绘制时保存每点的活动对象名称的数据结构。在下面的例子中对名称栈载入、压入或弹出名称，这样名称栈保存绘制时在某点所有可被选择的对象的名称。我们可以为名称栈中可选择对象创建一个层次结构，这样我们就有了一个强大的工具来辨识对象或对象组，进而操作它们。当在选择模式下绘制时，每次绘制到选择点时，名称栈中所有名称和其他选择信息都会存入选择缓存中。

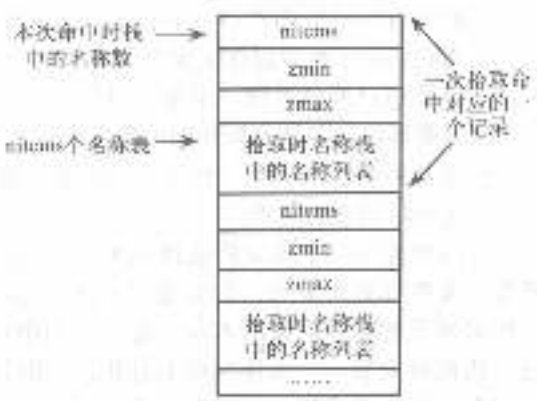


图7-6 选择缓存的数据结构

我们要介绍的下一个概念是存放选择的对象数据的选择缓存的数据结构。它是一个无符号的整型数组（GLuint selectBuf[SIZE]），它保存鼠标点击的命中记录表。命中记录是一个变长的整型数组，它包括了如图7-6所示的内容。每条记录都存放了在选择模式绘制中选择点被使用过一次的信息。这些信息保存在一个命中记录中，包括名称栈中的名称条目数量、到绘制对象的最近（zmin）和最远 269

(zmax)的距离，以及该次选择名称栈中所有名称的列表。这些距离是整型的，因为它们是从OpenGL的深度缓存中读取的，回想一下，采用整型是因为在比较距离时效率更高。这些距离是和投影环境相关的，最近的点有最小的非负值，因为这个环境把视点设在原点，对象离视点越远距离越大。

在一个典型的选择缓存处理过程中，查看每一个命中记录，找到的zmin最小值，因为这是离视点最近的命中。然后从中取出名称来完成你要做的工作。这是一个典型的对变长列表的处理操作。如果命中记录从索引N开始，索引N所对应的值代表名称栈中有K个名称，那么栈中的第一个名称就在索引$I = N + 3$的位置。这样，第一个名称从索引I开始，可以读出K个名称，对于每个名称可以对它所对应的场景做你要做的事情。举例来说，可以对每个用这个名称的对象设置某一类标记，这样随后的事件（动画，鼠标移动，键盘敲击等等）都会对这个对象起作用；你也可以改变模型变换参数，使被选择的对象产生运动。场景图可以帮助你确定要做的动作的实现方法。这种程序写起来并不困难，但在并发时需要小心，因为每个命中记录是可变长度的。

用选择缓存做选择操作时有一个关键点必须要知道：只需在选择模式下绘制场景，不需要做其他任何产生选择缓存的事，事情就是这样简单，系统会为你做剩下的事情：当绘制到被选择像素或拾取区域时，系统把活动名称栈中的所有名称放到选择缓存中。只要简单地把系统设置为选择模式，生成场景，调用glRenderMode(...)函数把系统恢复回绘制模式时，就从该函数的返回值中得到命中数，然后处理选择缓存中的这些命中信息。

OpenGL有两种用选择缓存拾取对象的方法，这两种方法都需要在选择模式下生成部分场景或者生成全部场景，我们会在下一节讨论这个问题。可以用选择缓存来识别是否有对象和鼠标点击的像素相交，或者可以用拾取矩阵设置一个附加的投影，剔除当前投影视域体外的所有对象。第一个方法可能是比较简单，因为它并没有对基本的绘制做任何改变；但第二个方法会更快，因为它用剔除技术来避免绘制离该像素不是很近的对象。我们会从简单的方法开始讨论，在后面部分介绍使用拾取矩阵的方法。

7.12.2 拾取操作的实现方法

在前面讨论的选择缓存实现概要中，似乎在选择模式下和在绘制模式下绘图没有什么不同，但实际上并非如此。我们有很多种方法可以让选择模式下绘图工作比绘制模式更快、更有效。这些方法包括：

270

- 如果有些对象用户将它们设置为不可以选择的，我们不需要在选择模式绘制它们。因为它们对于拾取操作来说是不可见的。
- 如果用户希望选择一个复杂对象，没必要在选择模式下把这个对象完整地画出来，而只需要设计跟该对象相似的简单对象，把它绘制出来即可。
- 读者甚至可以产生一些无形的控制，即允许用户拾取只在选择模式画出来而在绘制模式不画出来的对象。

可以想像用一个独立的选择场景图作为可替换的模型在选择模式下绘制。我们称为选择模型。发挥你的想象力，你会发现可以用选择操作来做很多事情。事实上，在应用这些技术时你必须了解具体对象的不同之处，比如用户要选择一个线框对象，在选择模式绘制时，把这个线框对象用一个实体对象来替代，否则将拾取不到这一线框对象，因为用户会将线框空间视作对象的组成部分，而OpenGL不会。另一个重要的用途就是选择文字，因为在OpenGL里不能选择光栅化以后的字符。如果在选择模式画任何光栅化的字符，无论你在哪里点击（不管点中还是没有点中），OpenGL都会认为字符被选中。如果要让一个光栅化后的词可以被

选择，就要在包围这个词所有字符的空间上创建一个长方形，用这个长方形来替代字符在选择模式下绘制。

那么，怎样确定哪些对象是可选择的，或者可成为感兴趣的选择对象呢？不是所有在选择模式下绘制的对象自动都是可选择的，必须指定哪些或哪组对象可以选择。感兴趣的选择对象是赋予选择名称的对象。选择名称对我们来说是一个新的概念，但OpenGL有一套完整的处理名称的机制。一个名称简单来说就是一个整数，在建模过程中可用它来识别一个点。由#define语句给这些整数一些符号含义的名称，名称在名称栈中管理。可以在名称栈中装入一个名称，即用glLoadName(int)函数随时更换栈顶的名称。也可以用glPushName(int)函数将一个新的名称压入栈顶。每一个名称栈中的名称都是活动的。当一个新的名称压入，原有和新加入的名称都是活动的。最后，当要让一个名称处于非活动状态，可以用glPopName(int)函数把它从名称栈中弹出。当一个选择发生时，名称栈中的所有名称都将放在命中记录中，通过嵌套名称，可以创建一个包含可被选择对象的层次结构。

举一个例子，假设要处理汽车，可以让用户选择汽车的某些部件。允许用户在不同的层次上选择，比如，可以选择整辆汽车，或者汽车的车身，或者只是选择一个轮胎。在下面的示例程序中，将针对一辆汽车（"Jaguar"）或者汽车的不同部件（"车身"，"轮胎"等等）创建一个选择的层次结构。在这种情况下，名称JAGUAR，BODY，FRONT_LEFT_TIRE，FRONT_RIGHT_TIRE就是相对于整数的有符号含义名称，如前面介绍的那样。这些整数定义在程序的其他地方。假设当这段代码在运行时名称栈是空的。

```
glPushName(DUMMY); // 将一个虚构名称放入列表
glLoadName(JAGUAR);
glPushName(BODY);
    glCallList(JagBodyList);
glPopName();
glPushName(FRONT_LEFT_TIRE);
    glPushMatrix();
    glTranslatef(...);
    glCallList(TireList);
    glPopMatrix();
glPopName();
glPushName(FRONT_RIGHT_TIRE);
    glPushMatrix();
    glTranslatef(...);
    glCallList(TireList);
    glPopMatrix();
glPopName();
```

271

当选择操作发生时，选择缓存中保存着所有对象，这些对象的显示像素中包含该选择像素，例如包括整辆汽车也包括了基础部件。如果要选择汽车的右前方轮胎，nitems为3，命中记录将包括三个名称：FRONT_RIGHT_TIRE，BODY和JAGUAR，你的程序会知道你选择了包含这3个对象的层次结构，你可以继续（或者允许用户挑选）某种选择方式或其他的选择逻辑。

在组织可选择对象时使用名称栈，与建模中的变换矩阵栈的使用非常相似。对象名称可以加入到场景图的信息中，当名称被下调加入到场景图的某个分支上，这个名称就会压入到名称栈上，当名称上传从分支中恢复，这个名称将从名称栈中弹出。这种机制把名称栈的创建和操作纳入建模过程中，使得处理过程更具系统性。

这里有名称栈的几个问题需要注意。第一个就是名称栈初始化时是空的，这样，不能简单地将一个名称装入栈中，这会产生一个错误。正确的做法是，必须在栈中压入某个名称，这样才能在装入一个名称并替换它。在这个例子中，在装载第一个真实名称之前，可以向名称栈压入一个dummy名称。第二点，在glBegin(mode)-glEnd()包含的程序区间，不能装载新的名称，所以，对象使用任何几何压缩操作时，所有的压缩必须作用在有单一名称的对象上。

第三个要注意的问题就是，装载一个名称只能是替换名称栈栈顶的名称。如果使用一个层次结构，并要从名称栈中移出整个层次结构，就必须不断地弹出直到只剩一个名称，然后，装载一个新的名称替换它。其实在实践中使用名称栈时，这些问题都会变得很简单。

7.12.3 拾取矩阵

用拾取矩阵做选择，在逻辑上和用选择像素做选择是一样的，但我们将它们分开介绍，是因为它使用不同的步骤，定义所谓"邻近"的概念，并用这一概念来讨论分辨邻近选择点的对象的方法。

在拾取过程中，你可以在鼠标点击位置周围定义一个非常小的窗口，然后辨别在该邻接区域内绘制的所有对象。该分辨结果返回到标准的选择缓存中，处理的方法也和前面讨论的
272 方法一样。由于创建了一个新的窗口，系统会把窗口外的所有对象剔除掉，使选择绘制事件非常高效。这个过程通过gluPickMatrix(...)函数设定的变换来完成，这个变换将在投影变换之后实施（其实是在投影变换之前定义的，请回忆前面介绍的变换的识别顺序和使用顺序之间的关系）。完整的函数调用是：

```
gluPickMatrix(GLdouble x, GLdouble y, GLdouble width, GLdouble height, GLint viewport[4])
```

这里x和y是鼠标选择点的坐标，也是拾取区域的中心，width和height是拾取区域用像素表示的尺寸，有时称为拾取容差，viewport是一个四个整数组成的向量，由下面的函数调用返回：

```
glGetIntegerv(GL_VIEWPORT, GLint * viewport)
```

这个拾取矩阵的功能是识别以鼠标点击点为中心的小区域，并选择在这个区域绘制的所有对象。拾取过程返回一个标准的选择缓存，然后处理选择缓存的信息，按前面介绍的方法来识别拾取的对象。

下面的代码段实现了该拾取方法，这段代码对应于前面代码中的doSelect(...)语句，标记为"设置标准的视图模型"和"标准的透视视图"。

```
#define PICK_TOL ...          // 拾取容差
int viewport[4];              // 存放视口数值
    ...
dx = glutGet(GLUT_WINDOW_WIDTH);
dy = glutGet(GLUT_WINDOW_HEIGHT);
    ...
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if(RenderMode == GL_SELECT) {
    glGetIntegerv(GL_VIEWPORT, viewport);
    gluPickMatrix((double)Xmouse, (double)(dy - Ymouse),
    PICK_TOL, PICK_TOL, viewport);
}
... call glOrtho(), glFrustum(), or gluPerspective() here
```

这里采用的名称与名称栈和前面介绍的完全一样。

7.12.4 使用后颜色缓存做拾取

另一种避免完全依靠选择缓存做拾取的方法就是采用双缓存绘制技术。在这种方法中，当要做选择操作时，用绘制模式绘制场景，将绘制结果用独特的方式写入后颜色缓存中，只画可以被选择的对象，并给每一个对象唯一的颜色来区分它们。如前面介绍的那样，可以使用另一种表达方式来表示被选择的对象，即用一个代理对象或替代对象来表示被选择对象。
273 当鼠标事件发生时，鼠标回调函数得到选择点的像素位置，检查后缓存中该像素位置的颜色。该颜色所对应的对象就是你拾取到的对象。如果绘制时同时进行深度测试，就会得到用户当

前看到的那个对象。在后缓存得到了想要的信息之后，不把它和前缓存做交换，而是让下一个正常绘制结果写入后缓存中来代替原来的人工色对象。

这种方法实现机制看起来很直观，当后缓存画满了人工色的图像，通过函数glReadBuffer(GL_BACK)来读取后缓存的内容（尽管后缓存是双缓存模式下默认的读取缓存）。然后用glReadPixels(...)函数读选择位置处1×1的像素颜色（也就是选择像素点的颜色），先前为每个对象定义的人工颜色帮助你识别是哪一个对象被选择。这是很直观的实现技术，但当需要从大量对象中选择时，就需要考虑一个比较好的颜色集来进行分辨。

7.12.5 一个选择操作的例子

选择过程会在下面的程序中说明，这个程序是学生Ben Eadington写的交互式形状选择程序，它设置一组可选择的控制点集来绘制Bézier样条曲面。当一个控制点被选择时，该点就被高亮显示，然后可以用键盘事件回调函数移动该点，曲面形状根据新的控制点集进行调整。程序运行的一个截图样本如图7-7所示，其中有一个控制点被选择（在前面，用一个红色的立方体显示而不用默认的绿色显示，参见彩图）。



图7-7 由可控制的点集所确定的曲面，其中有一个点被选择，参见彩图

这个选择作业的代码段在下面给出，完整的程序在本书的附带光盘中。所有的数据声明和求值函数的部分都省略了，还有一些函数的标准部分也省略了，只有重要函数的一些关键部分在下面讨论。我们会解释代码中的一些关键点，帮助你理解选择过程是怎样实现的，这些关键点分散在代码或函数中。

前几行代码是全局的选择缓存声明，选择缓存将保存最多200个值。对于这个具体问题来说，这个数目已经足够大了，因为这里没有层次结构的模型，同时不会安排有太多的控制点。实际使用的大小可能不会超过每个选择控制点4个GLints，最多不会超过10个点，这样看来选择缓存可以设置为只存放40到50个数据。其他问题可以通过类似的方法进行分析。

274

```
// 全局变量初始化代码
#define MAXHITS 200 // 命中记录中GLuints类数量
// 选择过程中使用的数据结构
GLuint selectBuf[MAXHITS];
```

下一要点的代码是鼠标回调函数，这就是要捕获鼠标键按下的事件，然后调用下面介绍的DoSelect函数，来处理鼠标选择。当点击事件处理完成（包括光标位置没有点击操作的情况），会触发一个redisplay事件，然后控制权会交给主处理进程。变量hit是一个全局变量，通过数组映射函数，用来辨识被选择控制点的索引。

```
//  用于选择的鼠标回调函数
void Mouse(int button, int state, int mouseX, int mouseY) {
    if (state == GLUT_DOWN) { // 重新绘制物体做选择
    hit = DoSelect((GLint) mouseX, (GLint) mouseY);
    }
    glutPostRedisplay(); /*    重绘显示    */
}
```

控制点可以在GL_RENDER或者GL_SELECT两种模式下绘制，这样drawpoints()函数必须处理这两种情况。唯一的不同就是必须对每一个控制点装入名称，如果这些控制点中的任何一个被前面的选择命中，它就会被识别出来，用红色绘制而不是绿色，值得注意的是，名称是按照绘制的顺序按次序装入，逻辑语句（hit == i*16+j%16）是数组映射函数，它将名称映射到一个16×16的控制点矩阵。但实际上在这个函数里没有指明被另一次鼠标点击选中的或

名未选中的控制点，这些将在后面的DoSelect()函数中处理。

```
void drawpoints(GLenum mode) {
    int i, j;
    int name=0;
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, green);
    // 遍历控制点数组
    for(i=0; i<GRIDSIZE; i++)
    for(j=0; j<GRIDSIZE; j++) {
        if (mode == GL_SELECT) {
            glLoadName(name); // 赋予每个点一个名称
            name++;           // 增加名称数量
        }
        glPushMatrix();
        ... 将点以正确比例置于正确的位置
        if(hit==i*16+j%16) { // 选择的点，需要用红色绘制
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
            red);
            glutSolidCube(0.25);
            glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,
            green);
        }
        else glutSolidCube(0.25);
        glPopMatrix();
    }
}
```

绘制模型时唯一要考虑的就是，在两个绘制模式下到底需要绘制和不需要绘制哪些东西。曲面只有在GL_RENDER模式才绘制，因为曲面上没有点可以选择。在GL_SELECT模式下唯一需要绘制的就是控制点。

```
void render(GLenum mode) {
... 进行适当的变换
if (mode == GL_RENDER) { // 如果模式为GL_SELECT，不要绘制曲面
    surface(ctrlpts);
    ... 其他一些无关的操作
}
drawpoints(mode); // 总是绘制控制点
... 按照需要做出变换，并合理退出
}
```

这个最后的函数是整个问题的核心。显示环境（投影和视图变换）的设置，glRenderMode函数把绘制模式设置成GL_SELECT模式，在这个模式下绘制图像。当再次调用glRenderMode函数并将绘制模式设置回GL_RENDER模式时，返回命中数，根据下一次绘制的需要再次设置显示环境，然后扫描选择缓存来找到zmin的最小值的对象的名称作为被选中的对象。然后返回这个数值，这样drawpoints函数会判断哪个控制点需要绘成红色，其他函数也知道应调整哪个控制点。

```
GLint DoSelect(GLint x, GLint y) {
    int i;
    GLint hits, temphit;
    GLuint zval;
    glSelectBuffer(MAXHITS, selectBuf);
    glRenderMode(GL_SELECT);
    glInitNames();
    glPushName(0);
    // 设置视图模型
    ... 标准透视和以及视点变换设置
    render(GL_SELECT); // 绘制用于选择的场景
    // 查询命中记录数量并重置绘制模式
    hits = glRenderMode(GL_RENDER);
    // 重置视图模型
    ... 标准透视和以及视图变换设置
    // 如果存在，返回选择的物体名称
    if (hits <= 0) return -1;
    else {
```

```
            zval = selectBuf[1];
            temphit = selectBuf[3];
            for (i = 1; i < hits; i++) { // 对于每个命中
                if (selectBuf[4*i+1] < zval) {
                    zval = selectBuf[4*i+1];
                    temphit = selectBuf[4*i+3];
                }
            }
        }
        return temphit;
    }
```

7.12.6 拾取小结

我们对前面讨论的标准拾取过程和例子代码做一个小结，可以得出以下三点：

- 定义一个无符号整数数组作为选择缓存。
- 设计鼠标事件回调函数来调用一个函数做下面的工作：

将绘制模式设置成GL_SELECT模式，画出图像的可选择部分，在此之前要装载好名称，这样选择时就可以辨别出相应对象。

当这步绘制完成时，返回选择缓存以便后续处理，然后将绘制模式设置回GL_RENDER模式。

- 在GL_SELECT模式下绘制对象时，管理好名称栈，这样你就可以在名称栈上得到正确的结果，根据它找到要拾取的对象。

这些步骤都很直观并容易理解，只要稍微注意和做好计划，你就可以容易地实现它。

7.13 MUI工具

图形API一般具有一些基本的交互能力，但很少提供一个完善的工具包供创建完整的用户界面。这些工具要包括交互图形对象，比如按钮、滑动条、调谐钮等等。没有工具包的支持，就需要自己写函数提供这些工具来实现交互式程序控制。这是一个非常繁琐的工作，需要花大量的时间。

正因为这些API不提供界面工具包，很多人就为图形API开发自己的用户界面工具包，其中有一些得到了广泛的使用。这些功能为程序添加用户界面，需要包含合适的头文件和对合适的库文件进行链接。在这一节中，我们介绍一个在OpenGL里常用的非常简单的界面工具，你可以获得使用这些界面工具的编程经验。就像我们在本章开始时提到的那样，我们不指望这些会让你成为一个高效的用户界面设计师，但会帮助你理解怎样用标准的工具包来为程序实现用户界面。 277

为了理解本节内容，并能使用图形API的交互工具包，必须理解基于事件驱动的编程机制。懂得通过OpenGL的GLUT工具包使用一些简单的事件和回调函数。还可以回顾一些标准的应用程序的界面功能，看看它们怎样使用按钮、滑动条、文本框和其他控件。

7.13.1 引言

我们常常可以找到很多用户界面工具包，但我们不能很方便地用它们在OpenGL上编程，甚至结合GLUT工具包也不行。其中，MUI（发音为"mooey"）工具提供的工具包能很好地和OpenGL的GLUT扩展一起工作。通过MUI，可以创建和使用滑动条、按钮、文本框和其他的工具。它们比标准GLUT提供的界面更直观、更易用。当然，也可以尝试写属于自己的工具包，但你可能更希望把重点放在解决手头的问题上，而不是写一个用户界面。所以MUI工具可以为你提供帮助。

MUI有着X-Motif界面的外观和使用感受，因此你不要期望你的程序外观会像Windows或者Macintosh上的程序。你只需更多地注意程序自身要实现的功能，然后利用MUI工具实现这些功能。这些工具的可视表现形式叫做控件，就像X窗口系统界面一样。这个概念将贯穿整个讨论过程。

这一节是根据Steve Baker的“MUI简明用户指南”[BAK]编写的，它们共同的特点就是：以少量例子和一些中等实验工作为基础进行介绍。它只是一个指南，并不是一个用户手册，尽管我们希望它对这个有用的工具的文档有所贡献。

7.13.2 应用MUI的功能

在使用MUI的任何功能之前，必须用muiInit()函数初始化MUI系统，如在本章后面的代码程序中所示，这个调用是在main()函数中执行的。

MUI控件由UI列表[⊖]来管理。需要用muiNewUIList(int)函数来创建一个UI列表，用一个整数参数作为这个列表的名称，然后用muiAddToUIList(listid, object)函数向这个列表添加想要的控件，其中listid是创建列表时给定的那个整数名称。可以创建多个列表，然后选择其中一个是活动的，让你的用户界面具有上下文敏感特性。但是，UI列表是静态的，而不是动态的，因为一旦UI表创建好，就不能删除这个表中的项目，也不能删除整个表。

MUI的每一个功能都可以设成可见的或不可见的，活动的或者是非活动的，可用的或者不可用的，可以根据程序特定的上下文来设定用户界面，这给你的程序增加了一些的灵活性。实现这些功能的函数是：

```
void muiSetVisible(muiObject *obj, int state);
void muiSetActive(muiObject *obj, int state);
void muiSetEnable(muiObject *obj, int state);
int muiGetVisible(muiObject *obj);
int muiGetActive(muiObject *obj);
int muiGetEnable(muiObject *obj);
```

278

图7-8给出了MUI的大部分功能，文本标签、水平和垂直滑动条、常规按钮、单选按钮和文本框。有些文本已经输入到了文本框中。这个例子给你一个标准MUI控件的外观概念。由于MUI源代码是公开的，如果需要，可以自定义控件，尽管这已经超出了这节讨论的范围。界面布局可以通过调整得到MUI对象的大小，可以通过下面的函数得到：

```
void muiGetObjectSize(muiObject *obj, int *xmin, int *ymin, int *xmax, int * ymax);
```

MUI对象回调函数是可选的（举个例子，可能不需要为一个固定文本的字符串注册一个回调函数，但是一个活动的部件，比如按钮，就需要回调函数）。为了注册回调函数，在对象创建时需要对它命名，然后用下面的函数把这个对象和回调函数连接起来：

```
void muiSetCallback(muiObject * obj, callbackFn)
```

其中，回调函数须按照下面的结构声明：

```
void callbackFn(muiObject *obj, enum muiReturnValue)
```

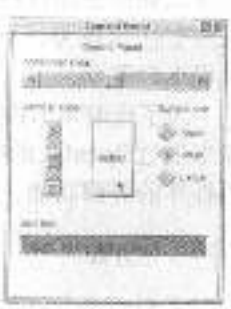


图7-8 在单窗口中MUI的功能元素集

请注意这个回调函数不需要与对象一一对应，在下面的例子中我们定义了一个回调函数，它可以注册给三个不同的滑动条部件，另一个回调函数可以注册给三个不同的单选按钮，因为每个对象要求回调函数响应的动作是相同的，当我们需要知道是哪一个对象在处理这个事件时，这个对象名称

⊖ UIList User Interface list用户界面列表。——译者注

由回调函数的第一个参数给出。 279

如果要使用回调函数返回的值，muiReturnValue的定义是这样的：

```
enum muiReturnValue {
    MUI_NO_ACTION,
    MUI_SLIDER_MOVE,
    MUI_SLIDER_RETURN,
    MUI_SLIDER_SCROLLDOWN,
    MUI_SLIDER_SCROLLUP,
    MUI_SLIDER_THUMB,
    MUI_BUTTON_PRESS,
    MUI_TEXTBOX_RETURN,
    MUI_TEXTLIST_RETURN,
    MUI_TEXTLIST_RETURN_CONFIRM
};
```

这里你可以显式地看出这些值的定义。在下面的例子中，对于按钮来说，“按钮按下”是唯一与之相关的返回值，而对于滑动条来说，我们需要得到滑动条的返回值，而不是直接处理MUI的动作。

7.13.3 MUI用户界面对象

MUI功能包括下拉式菜单、按钮、单选按钮、文本标签、文本框、水平和垂直滑动条。我们将介绍每种控件是如何工作的，并给出一些示例代码演示如何使用它们。

在使用MUI时要做的最主要的事情就是，由MUI从GLUT接管事件处理功能。所以在同一个窗口里不要混淆MUI与GLUT的事件处理机制。这就意味着必须为MUI控件和显示主程序分别创建独立的窗口。也许你会觉得这种方法很笨。但是在设计应用程序时你必须做出平衡：为了利用优秀的MUI功能是否要创建有别于传统应用程序界面的新型用户界面？这全取决于你。在你做出决定之前，你需要了解每一个MUI部件可以做些什么。

菜单栏

MUI菜单栏本质上是绑定到MUI对象的GLUT菜单。把这个对象加入到UI列表中。假设定义了一组GLUT菜单，名字叫做myMenus[...]，可以通过下面的函数创建一个新的下拉菜单。然后用下面的函数来向下拉式菜单中添加新的菜单：

```
muiObject *muiNewPulldown();
muiAddPulldownEntry(muiObject *obj,char *title,int glut_menu,
                    int is_help);
```

后面这个函数的一个例子就是：

```
myMenubar = muiNewPulldown();
muiAddPulldownEntry(myMenubar, "File", myMenu, 0);
```

这里，向myMenu中添加了一个GLUT菜单myMenu，这个GLUT菜单的值为0。在菜单栏中最后一个菜单is_help的值是1，因为在传统设计中，帮助菜单都是在菜单栏最右侧的菜单。 280

根据Baker的指南，在GLUT窗口被移动或者改变大小时，下拉式菜单显然会出问题。读者在使用MUI功能时，需要谨慎地处理与窗口相关的操作。

按钮

按钮用一个矩形区域来表示。在鼠标点击它时会设定某一个变量的值，或者完成某个特定的操作。当光标进入这个矩形区域时，按钮要高亮显示，表示它是可以点击的。按钮可以通过下面的函数来创建：

```
muiNewButton(int xmin, int xmax, int ymin, int ymax)
```

它返回一个指向muiObject*的指针。函数的参数定义了按钮的矩形区，用窗口（像素）坐标来表示，窗口左下方的像素坐标是(0, 0)。通常来说，MUI窗口的布局都会遵循这样的坐标

定义。

单选按钮

单选按钮和标准的按钮比较相似，但它们只有两种固定尺寸（标准尺寸和小尺寸）。这些按钮可以设计为复选方式，也就是同时可以有多个按钮被按下（允许用户选择选项集的子集），或者可以设计为单选方式，按钮连接在一起，当其中一个被按下时，其他按钮就弹起（允许用户选择选项集的一项）。和标准按钮一样，在光标经过它们时会高亮显示。

可以通过下面的函数来创建单选按钮：

```
muiObject *muiNewRadioButton(int xmin, int ymin)
muiObject *muiNewTinyRadioButton(int xmin, int ymin)
```

其中xmin和ymin代表按钮的左下角的窗口坐标。按钮可以通过下面的函数连接起来：

```
void muiLinkButtons(button1, button2)
```

其中button1和button2是两个按钮对象的名字，要连接更多的按钮，可以不断地连接有一个按钮重复的按钮对，参见后面示例代码。调用下面的函数可以清除按钮组中的全部按钮，其中的参数可以是按钮组中任意一个按钮：

```
void muiClearRadio(muiObject * button)
```

文本框

文本框是一种允许用户将文本信息输入给程序的工具，输入的文本可以根据程序需要任意使用。文本框也有局限，比如说，不能键入超过文本框长度的字符串。然而，文本框允许用户在键入文本时使用退格键或删除键来改正错误。文本框通过下面的函数创建：

```
muiObject * muiNewTextbox(xmin, xmax, ymin)
```

这里参数是窗口坐标，有很多函数用来设定文本框中的字符串：

281

```
muiSetTBString(obj, string)
```

清除文本框字符串的函数如下：

```
muiClearTBString(obj)
```

得到文本框的字符串的函数如下：

```
char * muiGetTBString(muiObject * obj)
```

水平滑动条

滑动条是返回滑动块位置的控件。这个位置用0到1之间的单精度浮点数表示，必须把这个值换算成应用程序需要范围内的值。水平滑动条通过下面函数创建：

```
muiNewHSlider(int xmin, int ymin, int xmax, int scenter, int shalf)
```

其中xmin和ymin是滑动条左下角的屏幕坐标，xmax是滑动条右边的屏幕坐标，scenter是滑动条中心杆的屏幕坐标，shalf是中心杆自身大小的一半。在滑动条的回调函数中，函数muiGetHSVal(muiObject * obj)返回的是一个浮点数，送往应用程序。相反的操作——设置滑动条的值——可以用下面的函数：

```
muiSetHSValue(muiObject * obj, float value)
```

垂直滑动条

垂直滑动条和水平滑动条有着类似的功能，但是它们在窗口中竖直放置，而不是水平放置。它们有和水平滑动条几乎一样的使用函数：

```
muiNewVSlider(int xmin,int ymin,int ymax,int scenter,int shalf)
muiGetVSValue(muiObject *obj, float value)
muiSetVSValue(muiObject *obj, float value)
```

文本标签

文本标签就是MUI控制窗口中的一条文字。程序可以通过显示在窗口上的文字和用户进行交流。标签上可以显示固定或者变长字符串。下面的函数用来设置文本标签的固定字符串：

```
muiNewLabel(int xmin, int ymin, string)
```

其中xmin和ymin代表文本标签显示位置的左下角坐标。要定义一个变长字符串的标签，需要给字符串一个muiObject名字，通过下面的函数

```
muiObject * muiNewLabel(int xmin, int ymin, string)
```

使该名字和标签绑定在一起，然后用函数muiChangeLabel(muiObject *，string)来改变显示在标签上的字符串。

文本标签可以用来识别含有控件的窗口，也可以在按钮上放一个名字，在单选按钮旁边放几个名字，还可以在滑动条上放相应名字。总之，文本标签告诉用户窗口中每一个控件的含义，并且可以在这些含义改变时，改变文本标签上的文字。 282

7.13.4 一个例子

我们通过一个简单的应用程序，来说明如何用MUI工具来创建控件。这是一个选择颜色的应用程序，根据用户的需要，用三个（R/G/B）或者四个（R/G/B/A）滑动条来操作。这类程序通常都会在一个足够大的区域中显示所选择的颜色，才能让用户不受旁边其他颜色的干扰，去感知选择到的颜色。这个程序和前面不同，不仅显示颜色，而且显示RGB立方体三个固定颜色分量平面，然后在立方体上选择一种颜色，用该颜色画一个（带光照的）球体。

这个应用程序的设计是基于第9章的一个例子，它给出了一个带三个变量的实数函数对应的三个正交平面的位置。我们用MUI滑动条来控制三个平面的位置。在窗口中再添加单选按钮，用户可以用它定义位于三平面相交处的球的大小。

从这个程序中选出来的代码包括muiObjects的声明，滑动条和按钮的回调函数，以及主程序中定义MUI部件的代码，把它们和回调函数关联起来的代码，以及把它们添加到MUI列表中以便识别的代码。这里的关键是MUI回调函数，就像我们前面讨论的GLUT回调函数一样，需要输入参数，以及通过改变全局变量来实现大部分功能，这些全局变量在程序的建模和绘制操作中还要用到。读者可以在本书的其他资源中找到这个例子的完整程序。

```
//  muiobjects和窗口识别符的选择声明
muiObject *Rslider, *Gslider, *Bslider;
muiObject *Rlabel, *Glabel, *Blabel;
muiObject *noSphereB, *smallSphereB, *largeSphereB;
int muiWin, glWin;
//  按钮和滑动条的回调函数
void readButton(muiObject *obj, enum muiReturnValue rv) {
    if (obj == noSphereB)
        sphereControl = 0;
    if (obj == smallSphereB)
        sphereControl = 1;
    if (obj == largeSphereB)
        sphereControl = 2;
    glutSetWindow(glWin);
    glutPostRedisplay();
}
void readSliders(muiObject *obj, enum muiReturnValue rv) {
    char rs[32], gs[32], bs[32];
    glutPostRedisplay();
    rr = muiGetHSVal(Rslider);
    gg = muiGetHSVal(Gslider);
    bb = muiGetHSVal(Bslider);
    sprintf(rs,"%6.2f",rr);
```

283
```
        guiChangeLabel(Rlabel, rs);
        sprintf(gs,"%6.2f",gg);
        guiChangeLabel(Glabel, gs);
        sprintf(bs,"%6.2f",bb);
        guiChangeLabel(Blabel, bs);

        DX = -4.0 + rr*8.0;
        DY = -4.0 + gg*8.0;
        DZ = -4.0 + bb*8.0;

        glutSetWindow(glWin);
        glutPostRedisplay();
    }
    void main(int argc, char** argv){
        char rs[32], gs[32], bs[32];
    //  创建OpenGL窗口及其回调函数
        glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
        glutInitWindowSize(270,350);
        glutInitWindowPosition(600,70);
        mainWin = glutCreateWindow("Control Panel");
        glutSetWindow(mainWin);
        guiInit();
        guiNewUIList(1);
        guiSetActiveUIList(1);
    //  定义颜色控制滑动条
        guiNewLabel(90, 330, "Color controls");

        guiNewLabel(5, 310, "Red");
        sprintf(rs,"%6.2f",rr);
        Rlabel = guiNewLabel(35, 310, rs);
        Rslider = guiNewSlider(5, 280, 265, 130, 10);
        guiSetCallback(Rslider, readSliders);

        guiNewLabel(5, 255, "Green");
        sprintf(gs,"%6.2f",gg);
        Glabel = guiNewLabel(35, 255, gs);
        Gslider = guiNewSlider(5, 225, 265, 130, 10);
        guiSetCallback(Gslider, readSliders);

        guiNewLabel(5, 200, "Blue");
        sprintf(bs,"%6.2f",bb);
        Blabel = guiNewLabel(35, 200, bs);
        Bslider = guiNewSlider(5, 175, 265, 130, 10);
        guiSetCallback(Bslider, readSliders);
    //  定义单选按钮
        guiNewLabel(100, 150, "Sphere size");
        noSphereB  = guiNewRadioButton(10, 110);
        smallSphereB = guiNewRadioButton(100, 110);
        largeSphereB = guiNewRadioButton(190, 110);
        guiLinkButtons(noSphereB, smallSphereB);
        guiLinkButtons(smallSphereB, largeSphereB);
284     guiLoadButton(noSphereB,    "None");
        guiLoadButton(smallSphereB, "Small");
        guiLoadButton(largeSphereB, "Large");
        guiSetCallback(noSphereB,   readButton);
        guiSetCallback(smallSphereB,readButton);
        guiSetCallback(largeSphereB,readButton);
        guiClearRadio(noSphereB);
    //  添加滑动条和单选按钮至UI列表
        guiAddToUIList(1, Rslider);
        guiAddToUIList(1, Gslider);
        guiAddToUIList(1, Bslider);
        guiAddToUIList(1, noSphereB);
        guiAddToUIList(1, smallSphereB);
        guiAddToUIList(1, largeSphereB);
    //  创建显示窗口及其回调函数
        ...
    }
```

这个应用程序的显示窗口和用户界面如图7-9所示。水平滑动条用来控制颜色的R、G和B
285 三个分量，具体的数据在滑动条的上方给出，对应于R、G和B值的三个平面显示在RGB立方体中，在三个平面的相交处有一个球体，这个球体的颜色是控件选择的颜色，它的大小由单

选按钮选择。RGB立方体可以用前面介绍的键盘控件来旋转，使用户可以将球体的颜色和周围三个平面上的颜色进行比较。这里我们也碰到了活动窗口问题。当用户想旋转立方体时需要激活显示窗口，想操纵颜色控件时需要激活控件窗口。

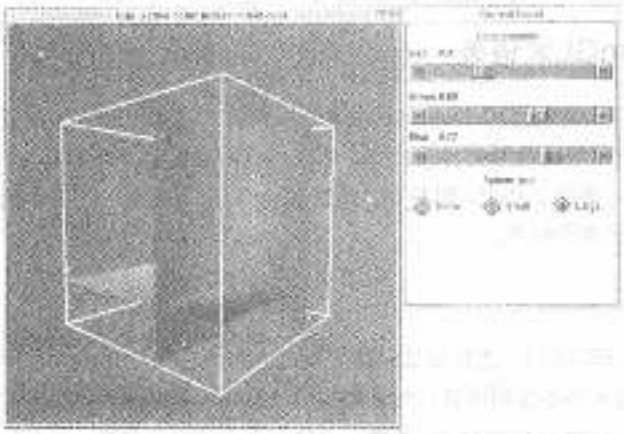


图7-9 颜色选择器，两个窗口分别是显示窗口和控件窗口

7.14 在Windows系统中安装MUI

MUI 通常和GLUT 一起发布，如果系统中已经安装了GLUT，也许也已安装好了MUI。如果还没有安装GLUT，可以下载并解压缩GLUT的发行包，得到一些头文件（在include/mui目录下）以及一些库文件：如Unix下使用的libmui.a和Windows下使用的mui.lib，把这些文件安装到常用位置上。比如，针对Windows的Visual Studio，把mui.lib安装到下面的目录：

```
<drive>:\Program Files\Microsoft Visual Studio\VC98\Lib\
```

把头文件也放到常用位置。对于Windows的Visual Studio，放到下面的目录：

```
<drive>:\Program Files\Microsoft Visual Studio\VC98\include\
```

然后把mui.lib加到工程文件中，就可以使用MUI了。

7.15 建议

MUI控制窗口有一些行为超出了程序员的控制，为了避免出现意外的错误，就必须注意这些问题。要注意的主要问题是，当光标经过窗口的MUI的控件区域时，将产生一系列事件（以及和它们相关的redisplay命令）。如果应用程序不能很好地处理这些redisplay命令所造成的变化，程序会出现一些想不到的变化，甚至不再产生任何事件。所以，当使用MUI时，应该特别注意程序处理redisplay时的结构，确保在显示命令之前，清除会引起显示变化的全局变量。

7.16 小结

这一章中，我们列举了图形系统中用到的一些标准事件，以及这些事件用到的回调函数。还给出了结合这些事件编程的例子。本章还介绍了一个用户界面工具包，包括它的价值以及在图形API中能找到这种交互工具包的例子。这些工具可以在图形程序中引入很多交互功能，使用户能运用这些交互功能与图像进行交互操作。

如果MUI限制了你的工作，可以试试GLUI用户界面工具包。可以从以下地址获得它的C++源代码及其文档：

286 http://glui.sourceforge.net/

如网站介绍的那样，GLUI是基于GLUT的C++用户界面程序库，为OpenGL程序提供按钮、选择框、单选按钮以及微调等控件。它不依赖于窗口系统，依靠GLUT来处理系统相关的问题，比如窗口和鼠标的管理。它不要求在显示窗口中有单独的用户界面窗口，这样可以实现更强大的功能。

7.17 本章的OpenGL术语表

在本章中通过GLUT工具包介绍了OpenGL的事件驱动机制。下面列举了介绍过的一些GLUT的函数和常量，可能有些GLUT函数功能没有详细介绍，所以，我们建议读者查阅GLUT的文档以免有所疏漏。

本章还提到MUI用户界面工具包，但并不包括这部分内容。这里的内容跟其他的参考文献一样详细，只要参阅本章即可了解必要的细节。

OpenGL函数

glRenderMode(GL_SELECT)：选择场景绘制的模式，在GL_RENDER模式下将场景绘制结果写入颜色缓存，GL_SELECT模式并不绘制场景，而是把相关信息写入选择缓存中。

glSelectBuffer(value, buffer)：创建一个给定大小的缓存数组，保存GL_SELECT模式下得到的信息。

GLUT函数

glutAddMenuEntry(string, value)：在当前菜单的底部添加一个菜单项，显示string字符串，如果该菜单项被选中返回value值。

glutAddSubMenu(string, value)：在当前菜单的底部添加一个子菜单触发器，显示string字符串，然后将标识符是value的菜单作为子菜单。

glutAttachMenu(event)：将当前菜单和鼠标键选择事件关联起来。

glutAttachMenuName(event, string)：在Macintosh（Macintosh操作系统的惯例是从菜单栏产生下拉菜单，而不是弹出窗口）中，将菜单和菜单栏中的菜单项关联起来，该菜单项显示string字符串，通过已命名event的事件来识别菜单项的选择，这个已命名event事件必须是一个鼠标键事件。

void glutChangeToMenuEntry(index, string, value)：将当前菜单中index所指示的项换做新菜单项，新菜单项显示字符串为string，被选择时返回value值。

void glutChangeToSubMenu(index, string, menu)：将当前菜单中index所指示的项换做新的子菜单触发器，该子菜单索引为menu，显示字符串为string。

287 int glutCreateMenu(functionname)：为某个菜单项被选中时产生的事件设置回调函数。

void glutDestroyMenu(value)：销毁索引值value传递给函数的菜单。

glutDisplayFunc(functionname)：把display事件的回调函数设置为以functionname命名的函数。

int glutGetMenu(void)：返回活动菜单的索引编号。

glutIdleFunc(functionname)：把idle事件的回调函数设置为以functionname命名的函数。

glutKeyboardFunc(functionname)：把keypress事件的回调函数设置为以functionname命名的函数。

glutMotionFunc(functionname)：把鼠标移动事件（鼠标在图形窗口中移动，同时有鼠标键被按下）的回调函数设置为以functionname命名的函数。

glutMouseFunc(functionname)：把mouse事件的回调函数设置为以functionname命名的函数。

glutPassiveMotionFunc(functionname)：把鼠标被动移动事件（鼠标在图形窗口中移动，但是没有鼠标键被按下）的回调函数设置为以functionname命名的函数。

glutReshapeFunc(functionname)：把reshape事件的回调函数设置为以functionname命名的函数。

glutSetMenu(value)：将索引编号为value的菜单设置为活动菜单

glutSpecialFunc(functionname)：把special按键事件的回调函数设置为以functionname命名的函数。

glutTimerFunc(msec, functionname, value)：把timer事件的回调函数设置为以functionname命名的函数，设置定时调用的时间为msec毫秒，以及传递给回调函数的值value。

OpenGL参数

GL_RENDER：在glRenderMode()函数中设置的参数，系统将绘制结果写入颜色缓存中。

GL_SELECT：在glRenderMode()函数中设置的参数，系统将识别所有由选择像素点绘制的图元，并将它们写入选择缓存中。

GLUT参数

GLUT_KEY_F*：功能键F1～F12的符号名

GLUT_KEY_LEFT：左光标控制键的符号名

GLUT_KEY_UP：上光标控制键的符号名

GLUT_KEY_RIGHT：右光标控制键的符号名

GLUT_KEY_DOWN：下光标控制键的符号名

GLUT_KEY_END：END特殊键的符号名

GLUT_KEY_HOME：HOME特殊键的符号名

GLUT_KEY_INSERT：INSERT特殊键的符号名

GLUT_KEY_PAGE_UP：PAGE UP特殊键的符号名

GLUT_KEY_PAGE_DOWN：PAGE DOWN特殊键的符号名

GLUT_LEFT_BUTTON：三键鼠标上左键的符号名

GLUT_MIDDLE_BUTTON：三键鼠标上中键的符号名 288

GLUT_RIGHT_BUTTON：三键鼠标上右键的符号名

7.18 思考题

1. 在本章中我们遇到了4DF和6DF控制（DF是degree of freedom的缩写，即自由度）。但也有2DF控制，在讨论MUI的滑动条时使用了2DF控制。请详细地描述怎样用三个滑动条（或者其他2DF控制）来实现6DF控制。它是不是一个很好的创建6DF控制的方法？为什么？你可以用鼠标移动来控制4个自由度，而用键盘控制另外2个自由度吗？同样，这种6DF控制是一个好方法吗？
2. 当用鼠标移动来控制旋转时，很重要的一点就是鼠标控制的方向要和生成图像移动的方向一致。可以采用两种移动方法——在固定的场景中移动视点，或者保持视点不动而移动场景中的对象。如果用鼠标的x和y坐标的正方向代表向右以及向上的移动，那么你怎样实现前面提出的两种移动？
3. 想像在空间中布置一组对象，例如第2章中创建的旋转木马模型。假设让其中的一部分对象可以被选择（比如旋转木马中的柱子，而不是里面的动物）。请描述你将怎样定义选择过程以实现上述功能。
4. 有时，有些图形对象由一些很难被选择的部件组成。比如点或者是直线段，你会怎样设计一个“选择场景”，用替代品表示这些难被选择的对象，使这些对象更容易选择？你会选择什么样的对象来替代点或者直线段？
5. 请从效率和调取不同大小对象的方便程度等方面讨论选择和拾取两种操作的不同。两种操作各有什么优点和缺点？

6. 选择和拾取过程都要求分析选择缓存中的数据，来分辨哪些对象离视点最近，或者最远，或者与视点的其他关系）。为什么使用后缓存的方法不需要额外的工作就能找到距视点最近的对象？有没有方法用后缓存选择其他的对象？
7. 设计交互操作是一个很复杂的问题。学科HCI（human-computer interaction 人机交互）专门研究这方面的内容。但是，我们不妨将交互理解为应用程序给用户的操作和应用领域需要的操作之间的关系，后者是用户在计算机环境之外，实际生活中熟悉的操作。对下面每一个交互技术，请找出一个使用该技术的计算机以外的工作领域。
   a) 用滑动条来控制程序中的参数
   b) 用旋钮来控制程序中的参数
   c) 用按键来选择一个选项
   d) 一组单选按钮从一组选项中选择一个
   e) 用鼠标点击来识别图像中的对象
   f) 用鼠标拖拽来移动图像或者图像中选中的对象
   g) 用菜单来选择选项列表中的一项

7.19 练习题

289
1. 在很多程序中，将视点在三维空间中移动，并把视角控制在一个合适的方向以实现场景漫游。请设计一种方法，用键盘上一块菱形键区来控制运动的向左、向右、向前以及向后，通过鼠标移动旋转视点的方向。所谓菱形区域，我们指的是S-E-D-X（左手边）或者是I-J-K-M（右手边）这样的按键组。
2. 请用第9章的例子或者自己的程序，实现通过动画表达动作的应用，设计idle或者timer回调函数来处理必要的参数变化来实现动画效果。
3. 请用第9章的例子或者自己的程序，实现用户从一系列选项中作选择的应用，创建一个菜单界面允许用户选择一个或者数个选项。
4. 请用第9章中的例子或者自己的程序，实现用户从一系列选项中作选择的应用，创建一个键盘界面允许用户选择一个或者数个选项。请注意采用键盘对用户来说是易于理解的。
5. 请用第9章中的例子或者自己的程序，实现用户从一系列选项中作选择的应用，创建一个MUI按钮界面允许用户选择一个或者数个选项。
6. 请用第9章中的例子或者自己的程序，实现允许用户选择特定的图形对象进行操作的应用，创建鼠标选择操作以便用户选择图形对象。
7. 请用第9章中的例子或者自己的程序，实现需要用户输入参数或其他类的应用，创建一个MUI界面允许用户通过文本输入窗口键入或者用滑动条输入。
8. 试用以下方法来研究命中记录的实质：通过修改包含选择操作的程序，即在程序中引入一段存储代码能在完成选择时将选择缓存的内容逐字节地存到一个文件当中，并检查命中记录的内容，可用文件dump工具，比如Unix's od来查看这个文件。然后查看选择缓存的字节数组中各个组成部分的内容，以及查看这些组成部分是怎样组织的。
9. 将多个对象组合成一组并用一个名称，重新做上题的练习，将对象组成一个层次结构再做上题的练习。在选择时，名称栈中的名称记录列表会更复杂，将它们拆解开来理解对象的组成结构和层次结构如何完成选择工作。

7.20 实验题

1. 用idle和timer两种回调函数分别创建一个非常简单的模型（比如代码中用到的立方体）的动画，比较

两种动画的帧速率以及帧速率的一致性。然后将简单模型换成一个更复杂的模型，再做同样的工作。如果你有不同速度的计算机，将代码复制到不同的机器上运行。观察机器速度会给两种方法带来多大影响。

2. （课堂作业）对一个程序界面作完整的评测和用户测试超过了本书的范围。为了评价程序控件的有效性，可以邀请一些朋友，比如同学，来使用这个程序，然后让他们作出评价。选择一个涉及用户交互的研究问题，课堂上的每个人都为这个问题设计和实现自己的交互方法。将全班同学的程序公开，每个同学都去运行其他同学的程序，然后对其中的交互方法做简短的评价。整理这些资料使能发现哪个程序是最好的，然后讨论为什么这个程序的交互比较有效。 290

3. 回想在第1章中提到的，计算视体中的直线段的参数方程，将三维世界中的点投影到屏幕上的一个点。在可视空间中放置一些简单的球体和多边形图元，视图再在前视图平面上选择一个点，计算该点到每一个图元连线的交点，试着找到一种方法来分辨哪一个图元是距视点最近的。

4. 考察选择缓存的结构，试创建几个场景，它们含有不同的对象和场景组织结构，在每一个场景中做选择操作，然后将缓存中的内容打印出来，并手工分析这些数据（分析将命中记录返回程序时的数据）。创建的场景组织结构应包括下面几种情况：选择不和其他对象重叠或不是层次结构的单一对象，选择重叠的两个对象，选择点在重叠位置，选择一个有层次结构的对象。手工分析可以帮助你理解在处理选择缓存时应如何编码。

5. 通过在选择模式下绘制对象，尝试用后缓存做拾取操作，其中不同对象用不同的颜色绘制，然后将视点设定在屏幕的给定点上，分辨哪一个对象离视点最近。

6. 下面来做一个应用不同类型交互的实验。创建一个平衡木，横梁的一端有未知重量的对象，为一端添加重量来平衡横梁。用不同的交互方法来添加重量，试用完成平衡任务需要的时间以及用户完成这项工作的容易程度两方面来评价哪种方法最有效。下面给出几种可用的交互技术：

   a) 用调节钮或者滑动条来调整重量以达到平衡。

   b) 用拾取技术选择标准砝码来调整平衡（可以考虑在物理实验中的一组砝码）。

   c) 用键盘每次增加或者减少单位重量以达到平衡。

7. （场景图）为了使场景图系统化地包括事件处理的功能，可以在场景图中引入“事件节点”，来记录事件对于场景主中某一部分的控制，比如变换操作，外观属性，甚至是几何信息。请创建这种新的节点，并针对一个交互式图形程序通过添加这些节点来修改场景图。向场景图中添加事件处理功能可能会有些困难，看看你能不能做得到。

7.21 大型作业

1. （小房子）创建一个交互式的小房子漫游程序。先按练习题1要求的漫游方式实现，然后根据用户选择的视点位置，显示房子的视图。

2. （场景图分析器）向场景图中添加一个名称节点，让它和变换节点在同一个位置。向分析器中添加名称栈的压入和弹出操作。 291

第8章 纹理映射

在本书提到的所有技术中，纹理映射能产生最逼真的图形显示效果。本章中我们介绍1D、2D和3D纹理的纹理映射并给出一些1D和2D纹理映射的效果图。本章还会介绍如何通过自然照片和人工图片制作纹理。最后，通过OpenGL图形API在图形环境中实现纹理映射功能，并给出一些程序代码。为便于理解本章的内容，读者还需了解三维空间中的多边形几何知识和从一个空间线性映射到另一空间的映射技术。

8.1 简介

在前面章节中读者已经了解了光线和着色处理如何产生颜色，并将该颜色填充到多边形中。纹理映射是另一种在多边形中产生颜色的方式，它可以将一幅图像映射到多边形上以产生精美的画面，或者将相应信息加入到图像中而不必计算额外的几何值。纹理映射是计算机图形学中一种非常有用的工具，它的内容非常广泛，这里我们不可能涵盖所有的内容，只描述一系列适合于当前图形API的功能，以便于读者在编程中有效地使用纹理映射。

纹理映射的基本思想是当图形几何值已经计算和显示完成之后，在图像中加入附加的可视内容。本章讨论的API几何体一般是基于多边形的，当用纹理映射计算多边形上的像素值时，
202 该像素的颜色值可以通过纹理图数组值计算而得。纹理图是一组颜色数组，这些颜色来自于图像，该图像是用户想加到场景中的某一对象上进行显示的。这里提到的纹理图可以是1D、2D或者3D的数组，本章中着重讨论1D和2D的数组。纹理映射就是将场景中对象的点与纹理图中的点对应起来，以便于使用简单的几何图产生非常逼真的视觉效果图像。

本章主要给出纹理映射的实现细节。一般假设纹理是一幅图像，因此生成对象时，其表面的颜色值是来自于纹理图的。纹理图的来源很多，可以是数字或扫描照片、数字艺术图片或人工生成的图形，这项工作可以在物体表面生成非常逼真的显示效果。还可以加入其他技术处理，用纹理图改变物体表面的色彩亮度或α值。

图像并不是纹理映射的唯一来源。纹理数组也可以是沿着物体表面一个像素一个像素地计算而得到。本书中提供了一些用纹理映射的简单例子。通过计算得到过程纹理。后面将详细讨论过程纹理的技术细节。

纹理映射涉及两个空间，一个是2D屏幕空间，即显示物体的空间；另一个是纹理空间，该空间放置要映射到物体上去的纹理图的信息。纹理数组上存放关于纹理空间点的离散信息，称为纹理元。当创建图像时，这两个空间必须合理地链接起来，在设计最终的图像时必须考虑这两者的关系。

为了协调纹理映射中两个空间的关系，以及创建要映射到物体上去的纹理信息，还必须向图形系统提供纹理映射要用到的特殊参数。可以将这些参数看作纹理映射的状态操作，不同的API函数有不同的绑定操作，必须设定的绑定有：

- 存放在纹理内存中的纹理名（通常是一个短整数）
- 纹理图的维数
- 纹理图的图形格式
- 纹理图的色彩表示（纹理图可包含的颜色数）

- 按第10章描述的方式生成得纹理的多边形时，纹理图和物体的绑定方式
- 如果纹理坐标超过了基本纹理空间，该纹理的处理方式
- 纹理应用于物体的各个部分时如果发生走样，该纹理的处理方式
- 纹理是否有边界

用户在利用图形API处理纹理映射时要充分考虑这些绑定。

创建纹理图有许多方法。对于1D纹理，可以利用线性颜色函数将各种颜色放置到线段上。这类似于第5章中描述的伪彩色映射图。对于2D纹理，原始图可以是扫描图像、数字照片、数字图画或者截屏图，也可以利用Photoshop等图像工具进行处理，以达到更精彩的效果。通过图形API，用户还可以获得帧缓冲区中的数据，将它读入文件或作为一个纹理图。2D纹理图是最为丰富的，也是最常见的纹理。对于3D纹理，可以通过为空间上的点赋以色彩来获得纹 293
理图，但这种方式实现较困难，原因是扫描或绘制三维物体的工具较少。然而，用户可以从一个3D模型中计算出3D纹理的值，各种医学扫描图都可提供3D数据，因此3D纹理也有相应的应用。

大多数图形API都可接受不同格式的纹理图。针对不同的应用，纹理图的颜色可以是一基元的到四基元的，如选择RGB、RGBA或纹理图中四基元中的一种。但是一般的方法是直接使用24位RGB颜色（每个像素RGB各8位），它直接来源于一个图像文件，不作任何压缩或用特殊文件格式，Photoshop称为“原始RGB”。

最后，纹理映射比直接将颜色赋予物体表面丰富得多。根据图形API的不同，纹理图还可以有一系列特征，如透明度、亮度等。在许多复杂的图形系统中，纹理还可以有α颜色值，以便产生诸如云的效果，或者具有法向量方向，以便产生凹凸映射的光照效果以及各向异性的反射效果。

8.2 定义

这里定义的纹理图都是1D、2D、3D的颜色数组。也可以把它们当作包含颜色信息的1维、2维、3维空间。当使用纹理图时，纹理图中的顶点可能与要填充的多边形的像素不正好相配，这时系统需要一种方法从纹理数组中选择颜色。因为颜色有可能是从纹理顶点之间取出（而非顶点本身的颜色），所以要将纹理图当作是一个空间而非一个数组。图形API对来自于纹理空间的颜色进行插值并赋给某一像素。插值技术包括直接取纹理数组中最接近的顶点的颜色，该点附近顶点颜色的平均等。但是，刚开始了解纹理时不必关注这个问题，在后面的参考文献及本章的后面讲解OpenGL API时再详述这个问题。

8.2.1 1D纹理图

1D纹理图是一维数组，可以应用到物体的任何方向。特别地，如果一个方向的数据复制多遍，由此扩展到另一维时，则它可当作2D纹理图使用。因此，可以使用1D纹理图来强化选择的方向（本章的后面将给出此类例子）。1D纹理随着视平面与物体的距离不同而发生改变。

8.2.2 2D纹理图

2D纹理是二维数组，可用于场景中的二维表面。模型表示将一幅图像“贴”到物体表面上，这是纹理中最自然、也最容易理解的方式。另一种解释方法是，图像原来是放在弹性面上的，将该弹性面的顶点钉到物体表面的顶点上。纹理映射使物体多边形上点与纹理空间上的点(即纹理数组上顶点坐标)相对应，该点也就有了相应的颜色。画多边形时，使用纹理空间 294

中对应点的颜色。

8.2.3 3D纹理图

3D纹理是与3D空间中物体对应的3D数组。[WO00]中给出了一种有效的可视3D纹理技术。在科学计算可视化的体绘制中3D纹理非常有用。(CAT扫描)数据或电子枪发出的电子束分布数据都可以当作纹理数据，也可以将切片数据当作纹理数据，由此来理解空间中的高维信息。

8.2.4 纹理坐标与空间坐标的对应关系

用户定义几何内容时，需要将几何空间点与纹理空间点一一对应，这类似于在建立带平滑阴影的光照时需要给出每个顶点的法向量。现在对于每个顶点要给出更多的信息：顶点的空间坐标、顶点的颜色或者计算颜色时要用到的法向量，以及与该顶点对应的纹理坐标。绘制流水线部分要用到顶点坐标以便确定像素的坐标，而颜色、法向量和纹理信息则用于确定物体内部像素的外观属性。

根据图形API的不同，顶点可以与纹理中纹元坐标相关联，也可以与纹理图中顶点坐标相关联。后者更有效，因为它与真实纹理图的大小无关。纹理图中的顶点坐标一般在0与1之间，代表纹理图中某一点的位置比例关系。基于多边形对象的几何体与纹理相关联看上去比较复杂，实际上是比较简单的（如图8-1所示）。注意，纹理必须与对象几何体一致，因此，要保证几何体中的相邻边在纹理图中必须匹配等诸如此类的问题。

295

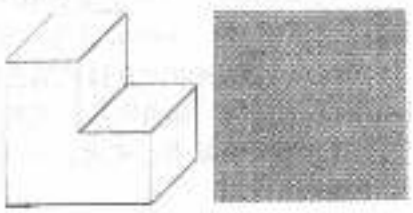


图8-1 对象与纹理相关联，对于左边的对象和右边的纹理，问题的关键是如何关联对象坐标与纹理坐标，使产生的效果完美

第一个问题是如何处理对象的边。对于表面是砖块的建筑物，某些角（通常是外角）在两个相邻的平面中都要显示出来，因此纹理映射要使用砖块中部的纹理坐标。这时，纹理显示可以将两块砖的中间切开，这时仿真每一条边都比较逼真。另一方面，还有一些角（通常是内角），两块砖之间的边缝有灰泥。如果仔细选择，可以选到这类纹理。最后，如果希望砖块沿水平边对齐，部分砖块沿垂直角对齐，也要仔细选择纹理。在以上的例子中，纹理中有20块垂直砖和水平砖，因此两块水平砖之间空隙为0.05纹理单元。刚好放下所有水平砖，此类计算在为任意几何体布局任意纹理时都需要。

8.2.5 对象颜色与纹理图颜色的关系

纹理映射中要用到图形对象和纹理图，该对象和纹理图都有自己的颜色属性。定义将纹理映射对象中的颜色时要考虑图形对象和纹理图中的颜色。

可能在大多数情况下，纹理映射用纹理图上的颜色替代源对象上的颜色，但图形API也会给出许多种组合两种颜色属性的方式。如果纹理图中带有α通道，则可以利用第5章中提到的颜色混合方式将纹理图中的颜色与图形对象中的颜色进行混合。当然也可以使用其他方式进行对象颜色与纹理颜色的混合，以获得其他效果。因此，用纹理图的颜色替代源对象的颜色并不是唯一方式，还有很多更有趣的效果。

8.2.6 纹理图的其他含义

纹理图不仅表示要映射到图形对象上去的图像，还可通过修改α值、亮度和多边形的像素

强度等方式来修改图形对象的外观属性，修改外观属性的细节随API的不同而不同。因此，用户可以有多种方法来修改图形对象的外观属性。对于多纹理情况更有效，此时定义一个图像使用了分层混合、亮度及一个或多个纹理。

8.2.7 场景图中的纹理映射

纹理是几何对象的外观属性，因此，应该是几何节点的外观属性。因此，几何节点本身必须定义与每个顶点相关的纹理坐标，并把纹理包含在几何节点中。

当创建节点的外观属性时，最直接的方法是考虑纹理图的所有细节（正如在OpenGL中使用的那样）。就像外观属性中的材质和阴影定义一样，纹理在定义几何节点之前定义。定义几何节点时，必须在其中包含每一顶点的纹理坐标，就像使用光照模型时必须在几何模型中加入每一顶点的法向量一样。不管是手工还是自动应用于场景图，表示外观属性的代码要在表示几何图形的代码之前出现，这个方法很简单但也会产生一些问题。 296

8.3 创建纹理图

纹理必须在装入纹理数组之前创建。我们可以通过读入一幅图像到纹理数组中来创建纹理，也可以通过计算过程来创建纹理。在本节中有这两种方式并说明创建纹理图的细节，在后面部分还会给出一些例子。

8.3.1 从图像创建纹理图

用图像作为纹理图非常普遍，特别是要让几何图带有自然的感觉时用得更多。诸如沙土、混凝土、砖块、草地、树木、冰块（这里只提到其中的一小部分）的自然纹理都来自于这些材质的扫描或数字照片。其他一些纹理（如火焰、烟雾）可以用数字绘画系统创建并保存在文件中。所有基于图像的纹理都用同一种方式处理：创建图像并存为相应的某种格式，该文件可被图形程序读入纹理数组中，并被API纹理处理程序使用。当然，这些都是2D纹理，因为纹理来自于二维图像。3D纹理图也可以由3D扫描输入（例如医学扫描），但这种情况比较少见。

使用图像文件作为纹理图的一个主要问题是图形文件格式非常多，参考文献[MUR]整本书都是分类讲解图像文件格式的。有些图像格式还含有压缩技术，使用时需要繁琐的计算。使用压缩图像格式需要RIP技术（光栅图像处理器）由文件创建像素数组，RIP技术本身也很复杂。但是，许多图形API实现也包含读取多种格式图像的程序接口。除非用户拥有这些程序接口，否则作者建议回避使用诸如JPEG、GIF、PICT，甚至BMP，而直接使用RGB序列。使用压缩图像文件格式的最简单的方式是用通用图像处理工具（例如Photoshop）打开图像，然后再存为原始RGB格式。



图8-2 本章中用到的作为纹理图的图像

这里我们用到的作为纹理图的示例图像是由作者摄制的一组非洲企鹅（如图8-2所示）。图形API可能对纹理图的大小有限制（例如，OpenGL标准要求图像的长宽（不含边界）都是2的幂），即使图像格式比较低级，格式中不包含大小信息，重新调用一次也很方便。虽然存在一些工具可以阅读不同格式的图像文件，但是为了方便起见，我们使用原始RGB格式图像，建议在使用原始RGB图像时，在文件名中加入大小信息，例如ivy_128×64.rgb，文件中就不必记录大小信息了。在本章的后面部分将详细描述用图像文件作为纹理图的程序例子。

8.3.2 人工生成纹理图

297 因为纹理图只是颜色、亮度、颜色深度和α值的数组，因此创建数组值可以通过计算而不必读取一个文件。通过计算生成纹理是一个非常有用的技术，计算量可大可小。这里将介绍一些创建计算纹理时的基本要求。

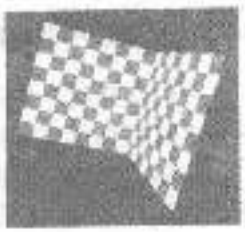


图8-3 作为纹理图的简单棋盘格方图

最简单的纹理是如图8-3所示的棋盘格方图。在创建64×64的纹
298 理数组时，如果(i/4+j/4)%2为0，数组下标变量tex[i][j]的颜色值为红色，如果(i/4+j/4)%2为1，数组下标变量tex[i][j]的颜色值为白色。

```
for (i = 0; i < 64; i++)
    for (j=0; j<64; j++) {
        if ((i/4+j/4)%2) tex[i][j] = red;
        else tex[i][j] = white;
    }
```

以上的程序中，red和white分别表示颜色向量的符号名。首先在纹理的左上格放置4×4的红格子，然后是白格子，然后再是红格子，这样就创建了传统的棋盘图案。这种纹理图常用于纹理映射图像调试，使用这种纹理图比较容易发现问题。

8.3.3 噪声函数生成纹理图

另一类比较有用的生成纹理的方法是使用噪声函数。噪声函数是单变量、双变量或三变量的单值函数，在统计意义上与旋转（在任何方向上随机）和平移无关（在任何位置随机）。在自变量一定变化范围内，函数的变化也受限。创建这类函数的方法很多，这里不一一阐述了，这里只提出一个相对简单的方法来定义噪声函数，并利用它生成纹理图。

除了噪声函数本身之外，有些简单纹理图也带有噪声函数的特点：与平移和旋转无关。如果对于纹理图的任一点赋予0与1之间的一个随机数，则纹理图相邻点之间就没有任何关系，这种纯随机纹理图在许多应用中是不合适的。

为了给出平滑的，但是与平移及旋转无关的随机纹理，可以使用第9章给出的过滤处理。过滤器将每一像素值用该像素附近像素点值的加权和替代，产生加权平均的像素值。首先产生随机纹理，然后用过滤器处理几遍，就可以获取平滑纹理，同时又具有与平移及旋转无关
299 的特性。这种方法很简单，2D处理结果如图8-4所示。图中所示的是灰度纹理图，对RGB三个成分分别处理则可以创建类似的彩色纹理图。使用3D过滤器则可以处理3D纹理，它是2D过滤器的扩展。

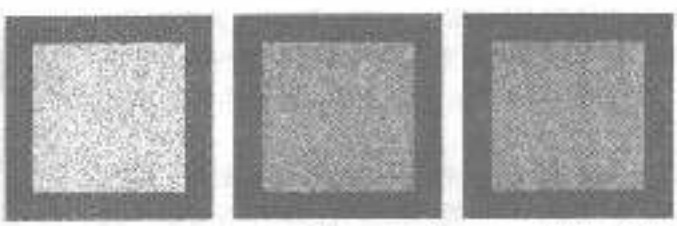


图8-4 随机2D纹理（左）和两遍过滤平滑后的纹理图（中）及五遍过滤平滑后的纹理图（右）

以上定义的随机纹理是使用噪声函数的例子，称为白噪声纹理，其随机值在0与1之间。然而还可以使用许多其他噪声，比较著名的是1/f噪声。该噪声函数是不同频率M白噪声函数f_M的线性组合，振幅f_M调整到$1/M$，如果它的频率为2^n（N为正值），则函数和的振幅为$\Sigma(1/N)$，和包含所有2的幂值，函数和的上限为1，因此0到1之间的数值都可能出现，使用该函数生成的纹理图在低频部分的数值较多，高频部分的数值（细节）较少，可用于模拟某些自然现象。

如图8-5所示，看一个1D的例子，噪声函数可看作是值在0与1之间的分段线性函数。噪声函数的频率是不同线段的个数。如果函数定义为0.0，0.5，1.0三点构成的线段，则其频率为2。如果函数定义为0.0，0.25，0.5，0.75，1.0五点构成的线段上，则其频率为4。显然取得频率为8，16或其他2的幂的函数是比较容易的。对于频率为2^N的函数f_N是在每点$x = i/2^N$上独机值在[0,1]之间的随机分段线性函数，这里i值从0到2^N。图中给出一个简单例子：分段线性函数的变量从0到16，它可用于创建1D纹理图，该图有16个像素宽。左边的一列函数表示不同的f_N函数，其中$N = 4$，8和16，中间的一列函数表示函数$f_N/2^N$，右边的函数是中间各函数的和函数。显然这是非常简单的函数例子，但是比较有用，特别是对于噪声纹理图。随着线段数的增加，可以产生更复杂的分段线性函数。



图8-5 不同频率的噪声函数（左列），除以频率后的函数（中列），创建一个噪声和函数（右列）

300

为了创建2D或3D噪声纹理，函数或纹理图必须是2D或3D空间，而不是1D空间上的。在2D情况下，在大小为$2^N \times 2^M$的正规网格下创建坐标为(x, y)的多边形，其z坐标为随机值，创建规则图形的方式如第9章描述。对于每一层的平面都插入一些点，然后再把所有层加起来获得的和函数值就是噪声函数。对于3D情况，则需要在3D规则网格上计算，该3D网格包含在八个顶点坐标已知的空间立方体区域内，计算出该空间区域内插值点的数值，这是2D方法的直接延伸，但是可视化表示比较复杂。2D纹理和3D纹理都可以通过不同频率的噪声函数，然后加权平均给出1/f噪声。

目前实际使用比这里描述的还要复杂。可以使用Peachy在参考文献[EB]中提出的梯度插值的方法，这也是一类在Renderman Shader系统中使用的噪声函数。这里就不一一详述了，本书的补充材料中给出了这类噪声函数的示例代码。图8-6是使用这类噪声产生的纹理图，下面描述一下这种方法在3D情况下的使用方法。



图8-6 1/f噪声函数产生的纹理图

生成噪声函数的一般处理方法是生成特定函数f_N不同频率的3D网格图。3D网格点的x,y,z分量与噪声函数中的定义域相对应，单位向量的三个随机分量表示该点的梯度值。假设每一网格点的高度为0，则梯度表示基本噪声函数的平滑度，具体的描述参见参考文献[EB]。

301

8.4 纹理图中的插值操作

为了显示带2D纹理映射的多边形，绘图流水线根据多边形顶点的纹理坐标对每一像素的纹理坐标进行插值。如果场景使用透视投影，且插值操作考虑透视变换，则在插值之前将每一像素的2D坐标反投影到原模型空间中去，就可以提供高质量的纹理映射。如果没做透视投影校正，则多边形边界上的纹理会产生奇怪的效果，如图8-7左边所示，棋盘纹理的四边形看上去像两个三角形，左下三角形的所有纹理线平行于左边和底边，右上三角形的纹理线平行于右边和上边，因此在三角形交界处产生了问题。对于图8-7的右边图，因为考虑了透视投影，棋盘则显示正确。该图表示透视校正插值对

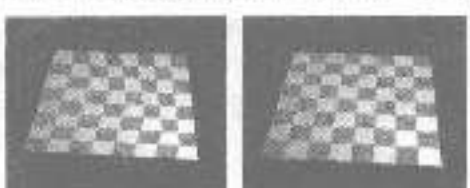


图8-7 纹理映射中不带透视校正（左）和带透视校正（右）的两个三角形定义的平面四边形区域

纹理映射的作用。对于1D和3D纹理映射也会产生同样的问题。正如在本章前面部分提到的那样，棋盘纹理对于纹理映射检测是很有用的。这里提到的技术只是第10章讲述的透视校正插值问题的一个简单特例。

8.5 纹理映射和布告板技术

第12章我们将引入布告板的概念，即三维空间的一个2D多边形，它不停地旋转，总是面对观察者，并且有纹理图映射其上，该多边形上显示的图像是出现在场景中的3D对象。这是纹理映射中的一个简单例子，多边形的颜色来自于纹理图，有些部分带有零α值表示透明。布
302 告板的几何原理在本章的后面讨论。

因为用于布告板的自然图像没有α通道，所以需要一些额外处理。首先，编辑一幅图像作背景图，然后根据颜色调整α值，即读入RGB文件生成RGBA数组。如果该像素不是背景色，则保存原RGB色，α值置为1.0（表示不透明度最大）。如果该像素为背景色，则置α值为0.0，表示在颜色混合时可以忽略这个像素的颜色。这类似于电视或电影编辑时的蓝屏技术。

使用布告板技术可以利用2D技术来显示场景中的3D对象。此时布告板和视点一般都是直接在场景中，而不是相对于其他几何体而体现在场景中，即视点和布告板都必须处于场景图的顶层，此时处理变换时比较简单。

布告板的建立比想像的简单。关键的问题是必须找出观察方向。这点不难，如果视点为(x_1, y_1, z_1)，视觉参考点为(x_2, y_2, z_2)，则观察方向为向量$d = <x_2-x_1, y_2-y_1, z_2-z_1>$，该向量最好能归一化，在球坐标下则表示为$(1, \theta, \varphi)$，$\theta$和$\varphi$的计算方法参见第4章。如果希望布告板的垂直分量保持竖直固定（如树木或标文），则将布告板沿垂直方向旋转θ度；如果希望布告板随时绕地面对观察方向，则两个方向分别旋转θ和φ度。

如果希望布告板或视点保持在场景中的水平面上，则让布告板面对视点就要困难一些，因为场景中还存在变换序列，这时要用到场景图，布告板造成矩形放入场景之后还有一些位置和方向的变换，必须对这些变换作逆变换(计算方法参见第3章)，使其恢复到场景图的根部，然后再加入一些变换使布告板面对视点，这是布告板绘制之前的最后的模型变换。

8.6 纹理图中包含多个纹理

许多图形API都可以在系统中保留多个纹理图，可在这些纹理图之间切换，以便于在场景中使用不同的纹理。但是，有时用到的纹理数超过能同时处理的纹理数时，例如，利用布告板技术创建标签时，场景有一系列不同的标签，此时可以创建一个纹理图用于所有的标签，在不同的纹理坐标时使用不同的标签。因此只需保留一个纹理图及多个标签内容。再举一个例子，可用一个大纹理图以及多个火焰图来创建闪烁的火焰图像。处理的方法是一次性装入整个纹理图，当要利用多个图像产生闪烁的火焰效果时，修改该区域的纹理坐标。

303 另一种情况是纹理图只需要使用图像的一部分。对于非矩形的照片，可将整个照片读入纹理内存，通过纹理坐标只选择感兴趣的部分显示。这可以绕过纹理只考虑矩形的问题。

8.7 纹理反走样

将纹理映射到一个几何体时，需要将几何体的顶点与纹理空间坐标相对应。该坐标可以是整数也可以不是（即纹理图的实际索引），几何体上的每一像素点都要作插值操作。几何体上的像素可以小于一个纹元（纹理单位），相邻像素之间的纹理坐标之差也可小于1，当然相邻像素之间的纹理坐标之差也可以大于几个纹元。此时会产生两种纹理走样：如果相对于对

象而言，纹理较粗糙，则产生纹元放大走样（由放大过滤器处理）。如果纹理相对于对象而言比较精细，则要在多个纹元中选一个颜色（由缩小处理器处理）。放大会造成锯齿状，即多个像素使用一个纹元。缩小会产生跳跃，即相邻像素没有使用相邻纹元。使用放大过滤器和缩小过滤器可以改善这些走样效果。

因为纹理可能走样，所以图形API常常包含纹理反走样机制。对于放大过滤器，相邻像素点可能都有同一纹元。使用最近点过滤器表示法取最近纹元点作为该像素的颜色。这种做法将多个像素设置为某个纹元点的颜色，因此会产生块状图像片。另一种方法是线性过滤，每个像素的颜色由它附近的纹元点做加权平均，权值由该点离像素的远近决定。当然，还有一些更复杂的反走样方法，但是图形API还要考虑合理的效率问题。OpenGL API中只能使用线性过滤器作反走样工具，其他API还可利用其他工具，有些研究系统还可利用自制的多种反走样工具。新一代的可编程图形卡中还有许多不同的反走样工具，具体参见参考文献[EB]。

8.8 MIP映射

当纹理映射中只包含一幅纹理图时，图形系统必须使用反走样技术从原始图中选择像素的颜色。如果多边形的像素空间大于纹理图，即并不是每个像素都能得到一个对应的纹元值，此时就需要用到诸如线性过滤等技术。当多边形的像素空间小于纹理空间时，则多边形上相邻的像素并不对应于纹理空间上相邻的纹元的颜色。当移动多边形时，颜色会发生跳变，产生未预期的效果。

此类问题的一个解决方法是，纹理图给出不同大小的层次，让系统选择最适合多边形大小的纹理图。MIP映射就是此类方法之一（MIP表示multum in parvo，即许多东西放一小盒子里）。这种方法可以提供多种分辨率的纹理图，控制每个层次的纹理版本。不同分辨率的纹理图都放在同一纹理存储区中，根据要显示的多边形的大小选择合适的纹理图。图8-8显示一组纹理图，其大小是2的幂次方，每一子图的数据读自其母图，大小为其母图的四分之一。使用MIP映射时所有尺寸的纹理图可以一直降到最小，维度变为1。



图8-8 MIP映射中用到的一组纹理

在OpenGL中MIP映射还可以通过图形API提供的函数计算获得纹理图，详细信息参见API文档。

MIP映射也可以视作层次细节处理（参见第12章），强调性能不如强调质量，它对提高纹理映射质量非常有好处。

8.9 多纹理

如图8-9所示，多纹理绘制技术是对单个表面使用两个或多个纹理。例如，对某一表面可使用一个木纹纹理和一个光照纹理。两个纹理结合使用可以产生有光照的木纹表面（如图8-10）。在游戏编程中这个用法非常普遍。另一例子是结合使用大气图、GIS（地理信息系统）定位信号、等高线来生成



图8-9 绘制流水线中的纹理映射，单个纹理（上）和多纹理（下）

305 一幅纹理图，产生真实感的地理、感兴趣的位置以及高度信息。

使用标准纹理映射机制也能达到与多纹理同样的结果。如果多个纹理具有同样的大小，纹理坐标也相同，则可以将多个纹理合并到同一纹理图中。具体的处理方法是，从不同的纹理数组中读取数据，用相应的操作将这些数据结合起来构成一个新纹理。但一般情况下并不这么简单，因为纹理图可能大小不一致，相对于要映射的对象来说其方向也不一致，因此，多纹理机制中的每一纹理都有自己的属性，在绘制处理时才做组合工作。



图8-10 使用多纹理，具体参见彩图

一般情况下，多纹理机制类似于使用多个不同的纹理。可从不同的图像源创建不同的纹
306 理图。这里需要指定：多纹理中有几个纹理，纹理如何组合及组合方式。定义几何图时需要定义纹理坐标，在OpenGL多纹理处理时这一步并不特别困难，将在本章的后面部分解释。

8.10 OpenGL中的纹理映射

在图像中使用纹理还有许多技术细节。OpenGL手册中包含所有这些细节，这里先讨论一些常见的问题，包括纹理环境、纹理参数，创建纹理数组，定义纹理图。本章还给出许多程序例子。

下面讨论与纹理有关的函数。这些函数虽然不多，但需要按照一定的次序使用。下面列出OpenGL中与纹理有关的基本函数以及有关说明，在后面有详细的使用介绍，在后面讨论例子时给出使用这些函数的次序。

glEnable(...)	允许纹理映射，glDisable(...)表示不允许纹理映射操作
glGenTextures(...)	生成用于纹理的一个或多个名称（整数）
glBindTexture(...)	把纹理名（在glGenTextures生成）与纹理对象绑定，如GL_TEXTURE_1D, GL_TEXTURE_2D，或GL_TEXTURE_3D
glTexEnv*(...)	定义纹理操作环境
glTexParameter*(...)	定义纹理反走样、反卷等操作参数
glTexImage*(...)	与定义纹理参数的信息绑定：如颜色坐标数，纹理数据的内部格式，纹理解释方式，图大小等等
glTexCoord*(...)	纹理坐标与几何体顶点相关
glTexGen*(...)	控制几何体顶点纹理坐标的自动生成
glDeleteTextures(...)	删除由glGenTextures生成的一个或多个纹理

这些函数分成四大类。前三个函数（glEnable,glGenTextures,glBindTextures）是OpenGL系统关于纹理映射的预处理函数。随后两个（glTexEnv,glTexParameter）定义系统绘制时纹理
307 与几何相关的属性。下一个函数glTexImage定义纹理数组，并且解释纹理数据如何装入纹理存储器中。下一个函数glTexCoord定义几何体上的相关纹理坐标。这些函数的使用次序在本章后面的纹理映射示例程序中给出。

8.10.1 顶点与纹理点相关

在OpenGL中如第3章描述的方式定义基本图元时，在glBegin(...)和glEnd()函数对之间使用glVertex*()和glNormal*()函数，也可以使用glTexCoord*()为每个顶点定义纹理坐标。当然，

这些函数根据坐标类型的不同也有一些变种，类似的模型如下：

```
glTexCoord1f(float)
glTexCoord2f(float, float)
glTexCoord3f(float, float, float)
glTexCoord1fv(float[1])
glTexCoord2fv(float[2])
glTexCoord3fv(float[3])
```

在调用glVertex()函数之前必须先说明纹理坐标。因为纹理坐标是作为顶点的状态出现的。

事实上纹理坐标是纹理点在纹理图中的位置，其坐标值在[0,1]之间。如果坐标值在此范围之外，则根据纹理属性wrap或clamp决定其取值。

8.10.2 从屏幕获取纹理

创建纹理的一种常见方法是：创建一幅图像，将颜色缓冲存为数组。该数组可作为纹理图使用。这种做法中的纹理图可以是各种类型的图像。许多图形API都支持这种处理方法。例如，在OpenGL中，利用glReadBuffer(mode)函数可取得颜色缓冲中的内容作为纹理使用，如果使用单缓冲器，则取当前缓冲器中的内容作为纹理；如果使用双缓冲器，则取备用缓冲器中的内容作为纹理。glReadPixels(...)函数将缓冲器中的内容（RGB或RGBA格式）复制至目标数组中。除此之外，该函数还可存储深度缓冲器中的一个颜色通道，或者亮度信息。这里就不一一详述，有关细节可以参见相关手册。

glReadPixels(...)函数的返回数组可以写入文件中以便于以后使用，可以直接由程序读入作为纹理数组使用。如果存为文件，则存为原始格式数据文件可能更有用，当然也可以加入一些其他信息存为其他格式，使其更有通用性。例如，如果在文件头加入图像的高度和宽度信息，则该文件类似于.ppm格式，可以让其他图像处理程序使用。此外，还可以加入脚本，将截获得到的图像数据流写入数字视频文件中，将图像转换成动画。参见第11章的详细介绍。 308

8.10.3 纹理环境

在图形API中使用纹理映射时，还需要定义纹理环境，说明如何使用纹理。OpenGL中应用于多边形的相应函数调用是：

```
glTexEnvf(GL_TEXTURE_ENV,GL_TEXTURE_ENV_MODE,*)
```

函数的最后一个参数表示纹理的处理方式，它可以在以下参数间选择：

```
GL_BLEND, GL_DECAL, GL_MODULATE, GL_REPLACE
```

在这里，我们用C、A、I、L分别表示颜色、α值、强度和亮度，f和t分别表示几何体像素值和纹理值。

如果纹理数据为*RGB*颜色，则以下参数表示：

GL_BLEND:	像素的颜色是$C_f(1-C_t)$，
GL_DECAL	像素的颜色是C_t，用纹理色简单地替代原来的像素色
GL_MODULATE	像素的颜色是C_f*C_t，用纹理色和几何像素色相乘来替代原来的像素色，
GL_REPLACE	颜色与GL_DECAL相同

如果纹理数据为*RGBA*颜色，则以下参数表示：

GL_BLEND:	像素的颜色是$C_f(1-C_t)$，α通道为A_f*A_t
GL_DECAL	像素的颜色是$(1-A_t)C_f+A_tC_t$，α通道为A_f，
GL_MODULATE	像素的颜色是C_f*C_t，α通道为A_f*A_t

GL_REPLACE	像素颜色为C_t，α通道为A_t
如果纹理数据为α通道值，则以下参数表示：	
GL_BLEND	像素的颜色是C_f，α通道为A_t
GL_DECAL	操作未定义
GL_MODULATE	像素的颜色是C_f，α通道为A_f*A_t
GL_REPLACE	像素颜色为C_t，α通道为A_t
如果纹理数据为亮度，则以下参数表示：	
GL_BLEND	像素的颜色是$C_f(1-L_t)$，α通道为A_f
GL_DECAL	操作未定义
GL_MODULATE	像素的颜色是C_f*L_t，α通道为A_f
GL_REPLACE	像素颜色为L_t，α通道为A_f
如果纹理数据为强度，则以下参数表示：	
GL_BLEND	像素的颜色是$C_f(1-I_t)$，α通道为$A_f(1-I_t)$
GL_DECAL	操作未定义
GL_MODULATE	像素的颜色是C_f*I_t，α通道为A_f*I_t
GL_REPLACE	像素颜色为I_t，α通道为I_t

309

8.10.4 纹理参数

纹理参数决定在几何体上如何放置纹理。OpenGL中纹理参数决定纹理如何反卷及过滤。纹理反卷由GL_TEXTURE_WRAP_*参数决定，说明在纹理坐标超出[0,1]范围时将如何处理。两个选项是纹理的重复和拉伸（如图8-11所示），在水平和垂直方向可以分别操作。重复纹理表示纹理坐标只关心坐标的小数部分，如果纹理坐标超过1就变复到0，因此可做到纹理在几何空间重复使用。拉伸纹理表示纹理坐标超过[0,1]的部分使用最接近0或1的值。使得超过[0,1]范围的纹理使用其边界值。可用函数glTexParameter*(...)决定使用重复纹理还是拉伸纹理。这两函数为：



图8-11 在垂直方向反卷纹理，在水平方向拉伸纹理的四边形

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
```

310

如果使用重复纹理，则要关注如何在几何体上表现纹理，即如何达到较好的效果。特别是纹理图的上下左右边界部分，使得边界对应条块能拼接起来。在某些画面背景上也使用条块拼接技术。方法之一是使用Photoshop工具，将图片平移到中间，并使几幅图两两相接。使用Photoshop工具还可对图片进行模糊处理，使得图片接合处的线条不明显。这种做法对图像的上下左右边界都合适，因此完成四方连续图像。

另一种重要的纹理参数用于处理反走样的像素过滤器。OpenGL中有缩小过滤器（图像中的每个像素对应多个纹理点）和放大过滤器（每个纹理点对应像素中的多个像素），用于控制对于纹理图不同的像素点着色。对于场景中的像素点，几乎不可能正好对应一个纹理数组中的索引。因此系统必须在纹理空间中做相应的处理，计算出几何空间上点的颜色。OpenGL中使用函数glTexParameter*(...)和参数GL_TEXTURE_*_FILTER进行处理。根据图像对应纹理图的大小选取缩小或放大过滤器。如果像素小于纹理元，则选GL_TEXTURE_MIN_FILTER，如果像素大于纹理元，则选GL_TEXTURE_MAG_FILTER。使用方法如下：

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
```

图8-12显示使用参数GL_NEAREST和GL_LINEAR后的不同之处。注意观察金鹏的放大图片。从图中可以看到，放大过滤器中使用GL_NEAREST参数显示的图片比使用GL_LINEAR的粗糙。如果过滤器中使用参数GL_NEAREST，则系统选取纹理空间最接近于该纹理坐标的纹理点值；如果过滤器中使用参数GL_LINEAR，则系统选取该纹理坐标附近四个最近纹理点值的平均。前者速度较快，但存在走样问题，后者较慢，但图像较平滑。如何使用依赖于用户对速度和质量的权衡。

311

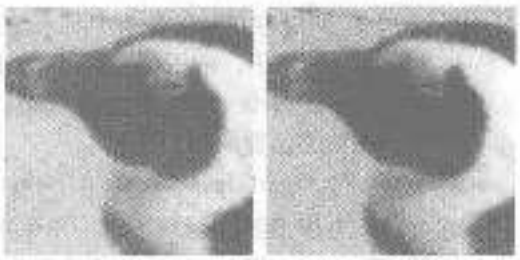


图8-12 选用GL_NEAREST（左）和GL_LINEAR（右）放大过滤器的金鹏头纹理图

8.10.5 获取及定义纹理图

获取及定义纹理图由函数glTexImage*D(...)来执行，这类函数较复杂，参数较多，这些函数可以处理1D，2D和3D纹理（函数中的星号表示维数），参数结构相同。

在应用glTexImage*D(...)函数之前必须先定义纹理数组并存储纹理数据。纹理数组中的无符号整数（GLuint）的维数必须与纹理维数相同。正如在讨论纹理内部格式那样，组织纹理数组的方法有多种：可以从文件中读入，也可以通过编程产生数据。本章的例子中给出这两种方法。

glTexImage*D(...)函数是处理纹理时参数最复杂的函数之一。这些参数包括：

- *target*，常为GL_TEXTURE_*D，其中*是1、2或3。可以使用1D纹理，但它超出了本章的讨论范围。本参数可用在多个地方，用于定义纹理图。
- *level*，表示层次细节号的整数。它支持多层次MIP映射。层次号为0表示没有MIP映射的图像。
- 纹理图的*internal format*，OpenGL支持多种格式，适应于不同的应用需求。OpenGL中的内部格式只是符号常量，可取不同的值，这里我们只列出常用的一些。大多数API都支持每颜色成分多个位数，但这里为简单起见，只介绍每个颜色成分8位的情况，其他的专用格式可以参见相应的手册：

```
GL_ALPHA8
GL_LUMINANCE8
GL_INTENSITY8
GL_RGB8
GL_RGBA8
```

- 纹理图的*dimensions*表示维数，类型为GLsizei，如果使用1D纹理图，则GLsizei参数表示宽度；如果使用2D纹理图，则参数值表示宽度和高度；如果使用3D纹理图，则参数表示宽度、高度和深度。其值在2^k + 2*(*border*)之内，*border*的值为0或1（由下一个参数说明）。
- *border*，其值为0（不含边界）或1（含边界）。 312
- *format*，纹理数组内的像素数据格式的符号常数，为如下值之一，当然还有其他取值，这里不一一例述了：

```
GL_ALPHA
GL_RGB
GL_RGBA
```

```
GL_INTENSITY
GL_LUMINANCE
```

这里的格式说明创建图像时如何使用纹理，这些参数已在前面关于纹理模式的影响和图像模式讨论的纹理格式中说明过了。

- *type*，说明保存在纹理数组中每个像素的数据类型的符号常数，通常比较简单，如本章后面部分的例子，常用的值为：GL_FLOAT和GL_UNSIGNED_BYTE。
- *pixels*，内存中像素数据（纹理数组）的存放地址。

因此，完整的函数调用如下：

```
glTexImage*D(target,level,internal format, dimensions,border,format,type,pixels)
```

在介绍立方体表面2D纹理的那一节中可以找到以上的完整函数调用。

函数glTexImage*D(...)只是介绍纹理数组的存储，没有提及图像的来源，如果图像来自于压缩文件，则必须先对图像进行解压缩，并存入*pixels*数组中。

用户可以用任何工具创建纹理图，具体创建的方法我们没有在这里介绍，这与用户的喜好有关，或与工具的可用性有关。这里关注的是使用纹理的方法。

8.10.6 纹理坐标控制

当对多边形中使用纹理时，可以用函数glTexCoord*()说明纹理坐标如何与几何体的顶点对应，正如在前面提到的那样，可以让OpenGL系统直接指定纹理坐标，可以用函数glTexGen*(...)指定纹理坐标，并说明纹理生成的细节。

glTexGen*(...)函数有三个参数。第一个是纹理坐标定义，可以是GL_S、GL_T、GL_R、GL_Q之一，S、T、R和Q是纹理的第一、二、三和齐次坐标。第二个参数是符号常数GL_TEXTURE_GEN_MODE、GL_OBJECT_PLANE、GL_EYE_PLANE之一。如果第二个参数是GL_TEXTURE_GEN_MODE，则第三个参数可以是符号常数GL_OBJECT_LINEAR、
313 GL_EYE_LINEAR、GL_SPHERE_MAP之一。如果第二个参数是GL_OBJECT_PLANE，则第三个参数是四值向量，定义对象线性纹理平面。如果第二个参数是GL_EYE_PLANE，则第三个参数是四值向量，定义含视点的平面。在后两种情况下，对象线性平面和视点线性平面都是参数平面。如果第二个参数是GL_TEXTURE_GEN_MODE，第三个参数是GL_SPHERE_MAP，则通过平面与纹理图的反射向量的近似来创建纹理。

有关纹理生成的应用包括色度深度纹理（ChromaDepth），这是一维的与视点有关的线性纹理，生成参数包括纹理的起点和终点。另一例子是自动轮廓生成，可以通过GL_OBJECT_LINEAR和GL_OBJECT_PLANE定义轮廓开始的基平面。因为轮廓一般都是从海平面（某一坐标为0）开始，所以该平面可作为基平面，很容易定义对象平面的系数。最后，GL_SPHERE_MAP纹理可以作为环境图。

8.10.7 纹理插值

正如在本章前面部分提到的那样，如果图像投影是透视投影，则绘制几何体的扫描线插值时需要考虑透视变换以获取较好的纹理质量。用以下的OpenGL函数可以控制插值质量：

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, hint)
```

这里，hint值可以是GL_DONT_CARE（系统默认）、GL_NICEST（透视调整以获得最好质量）或GL_FASTEST（不考虑透视调整以加快速度）。这些都在预处理步完成，可以假定默认为GL_FASTEST，大多数OpenGL实现都将考虑透视调整插值作为默认操作。

8.10.8 纹理映射和GLU四边形

正如在第3章关于OpenGL建模时提到的，GLU四边形对象有内嵌的纹理映射功能，这是建模中易于使用的特征之一。这时需要完成的任务有：将纹理装入系统并给纹理命名，定义四边形及其法向量和纹理，将纹理与要画的几何体绑定。下面给出这三步的简短代码例子，函数readTextureFile()需要用户提供，由用户与GLU函数来创建要画的四边形。

```
readTextureFile(...);
glBindTexture(GL_TEXTURE_2D, texture[1]);
glTexImage2D(GL_TEXTURE_2D,...);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

myQuadric = gluNewQuadric();
gluQuadricNormals(myQuadric, GL_SMOOTH);
gluQuadricTexture(myQuadric, GL_TRUE);
gluQuadricDrawStyle(myQuadric, GLU_FILL);
glPushMatrix();
    // 根据需要进行变型变换
    gluXXX(myQuadric, ...);
glPopMatrix();
```

314

8.10.9 多纹理

OpenGL函数glGenTextures(N,texNames)可用来定义多纹理。用户可以根据系统所需来定义纹理个数。对于每个纹理对象，可通过函数glTexImage*()和glTexParameteri()来定义纹理属性。用函数glBindTexture()和glTexEnvi()定义纹理单元，以整数形式给出可用的纹理名。绘制带纹理的几何对象时，先使用第一个纹理，然后对第一个纹理映射的结果使用第二个纹理，依此类推（参见图8-9）。

事实上使用纹理时，每一纹理的纹理坐标通过函数glMultiTexCoord*()与几何体的顶点对应。本章的最后部分将给出多纹理的代码例子。

8.11 例子

前面提到过，使用以下函数可以有多种方法来使用纹理：

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, mode)
```

第一种方法是使用掩膜技术（模式GL_DECAL），纹理内容作为不透明图像放置在几何体表面，除了纹理内容不显示其他内容。第二种方法是使用调制技术（模式GL_MODULATE），纹理内容显示在几何体表面，看上去像彩色塑料面。这种模式首先绘制白色表面，然后绘制调制纹理，由此可以显示带光照的着色表面。第三种方法是GL_BLEND模式，它将几何体的颜色与纹理图的颜色通过α值相混合，类似于颜色混合的作法。在下面的例子中，用1D调制纹理创建ChromaDepth图像，可以显示基表面的着色效果，用2D掩膜纹理创建映射立方体图像，因此立方体表面就都是纹理图的信息。在一幅图像中可以使用多个不同纹理，例如，可以在白色纯几何地形图模型上通过GL_MODULATE模式应用航空图2D纹理图，然后以GL_BLEND模式应用1D纹理图产生大多数透明效果，但部分特殊层面呈彩色效果，最终在地形图上产生高度线的效果。用户需要绘制的图像和开发新的技术。

下面讨论三个纹理图的应用例子。第一个例子使用1D纹理图定义模型中从某点开始的颜色变化。在本例中，距离表示与3D视图空间中视点的距离，用专用的滤镜表示颜色，该颜色表示场景的深度。第二个例子中，对几何体应用2D纹理，在平淡的场景中加入某些信息。第三个例子中，应用特别的2D纹理产生对象反射场景的效果。

315

8.11.1 使用Chromadepth过程

ChromaDepth过程使用1D纹理图来表示深度。如果用白光照射到白色物体上实现光照模型，则可以表示物体的着色处理。如果使用1D纹理实现颜色渐变，使视点处为红色，远离视点处为蓝色，显示结果如图8-13（详细情况参见彩页）。通过ChromaDepth玻璃管就可以看到三维图像，因为玻璃管对不同颜色有不同的衍射率，红色的衍射角大于蓝色，红色物体的跨度大于蓝色物体，而人眼就是通过跨度来区别远近的。如果跨度大，则人眼认为距离近，所以通过玻璃管，人眼认为红色物体比蓝色物体距离近。



图8-13 代数曲面的ChromaDepth彩色图，参见彩页

完成这个功能的程序代码参见后面关于1D颜色渐变部分，与第5章已经讲述过的伪彩色渐变极为类似的颜色渐变的问题，可以通过glTexImage1D()函数与颜色渐变结合，建立纹理环境和1D纹理所需参数，最后，通过glTexGen*()函数自动生成与视点线性相关的纹理应用于生成的表面，具体细节参见1D颜色渐变的例子。

8.11.2 使用2D纹理图在表面中加入信息

纹理图最常用的情况是创建相对简单的对象，将纹理图加到对象上产生较复杂的效果，特别是需要模拟真实世界的某个对象时，此时，需要将图像（例如真实世界的某个图像）映射到较简单的物体上。如图8-14所示，将企鹅图像映射到立方体上作为纹理图，此时的立方体比原来的纯几何图具有更多的视觉内容，并且将图像放到立方体的正方形表面上也十分简单。完整代码参见下面关于2D纹理程序代码部分。



图8-14 3D立方体，一面带企鹅纹理图

8.11.3 环境纹理图

环境纹理图是通过某一对象提供对真实世界的反射图像，可以提供非常有趣的效果，因
316 为对真实世界的反射效果可以提供非常重要的视觉信息。环境图可以是照片或人工图（即要反射的对象），调整纹理参数给出真实感效果。一个简单的例子是反射色彩表面，在图8-15中，图像是一幅香港的照片做成的纹理图，它是通过Photoshop球型滤波处理得到的照片。使用这种滤镜使环境图更有意思，因为环境图使用某点处的表面法向量来表示整幅图的纹理图。



图8-15 环境图原纹理（左），环境图放置在一个表面上（右）

本例的2D纹理是通过glTexGeni()函数自动生成的，表面纹理坐标通过每一点的法向向量
317 生成。除此之外，与其他2D纹理的处理方式相同。正如在1D线性纹理例子中提到的那样，本例在白色表面上使用光照，纹理以GL_MODULATE方式加入，保证形状信息中带有光照，纹理信息中带有环境图。

8.12 建议

纹理映射比这里提到的简单例子要复杂得多。可以使用1D纹理作表面的轮廓线，用颜色编码作高度值，产生第5章提到的视觉效果。使用2D纹理作凹凸表面（带亮度的纹理），产生多变云层的效果（使用带α的分形纹理），或带阴影的云图（在地形图上带亮度的纹理）。这类工作都非常有意义。

使用纹理映射时必须考虑几个问题：如果选择的纹理坐标不正确，则会产生意想不到的结果，因为对象的几何图与纹理图的几何不匹配。例如，如果纹理图的大小与映射空间的大小不匹配，则会无意中破坏纹理的比例。更严重的是，如果将长方形纹理图映射到非长方形区域时会产生纹理的非线性走样。想像一下将砖块纹理映射到非凸多边形或者锥面上的效果（如图8-16所示，绘制带砖图案的纹理锥）。另一问题是，如果将多边形间带有缝隙的两张图映射到相邻几何体时也会产生一些问题。很像贴墙纸时在角处没对齐，结果会破坏真实感。最后，如果纹理图的分辨率与几何体的分辨率差别很大，也会产生纹理走样问题。在本章中使用放大和缩小过滤器来解决这个问题。



图8-16 带砖图案的纹理锥台，显示不匹配的边界和不一致的大小问题

使用色度深度（ChromaDepth）玻璃管的1D纹理映射处理可以产生精彩的3D视觉效果。但是它不能把颜色作为编码和交流信息的方式。它只能用于通过形状运载信息的情况，事实上它在地理信息和工程图像以及分子模型领域非常有用。

8.13 代码实例

8.13.1 1D颜色渐变

在色度深度例子中使用1D纹理映射的程序代码如下所示。程序声明中建立颜色渐变，定义整数纹理名，创建纹理参数数组。

```
float b1, b2;
float texPars[4];
static GLuint texName;
float ramp[256][3];
```

Init()函数中有如下函数调用：定义纹理图、纹理环境和参数，启动纹理生成和应用。

```
makeRamp();
glPixelStorei(GL_UNPACK_ALIGNMENT, 1) ;
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE) ;
glTexParameterf(GL_TEXTURE_1D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(GL_TEXTURE_1D,GL_TEXTURE_MAG_FILTER,GL_LINEAR) ;
glTexParameterf(GL_TEXTURE_1D,GL_TEXTURE_MIN_FILTER,GL_LINEAR) ;
glTexImage1D(GL_TEXTURE_1D,0,3,256,0,GL_RGB,GL_FLOAT,ramp) ;
glEnable(GL_TEXTURE_GEN_S) ;
glEnable(GL_TEXTURE_1D) ;
```

makeRamp()函数创建全局数组ramp[]保存纹理图数据。这个渐变不使用RGB值，而使用HSV颜色模型的插值。色度是角度值（度），饱和度和亮度都是1。函数中的常量240来自于HSV模型的结构：红色是0度，蓝色是240度，绿色是120度。从红色到蓝色的全饱和颜色通过0～240度的插值而得，蓝色再到绿色也是通过插值得到。HSV转换到RGB的函数为hsv2rgb(...)（参见第5章）。

```
void makeRamp(void)
{
```

```
    int i;
    float h, s, v, r, g, b;
// 1D纹理图的色彩变
// 从0开始至240结束，共256步
    for (i=0; i<256; i++) {
        h = (float)i*240.0/255.0;
        s = 1.0; v = 1.0;
        hsv2rgb(h, s, v, &r, &g, &b);
        ramp[i][0] = r; ramp[i][1] = g; ramp[i][2] = b;
    }
}
```

最后，display()函数包含如下代码，ep是gluLookAt(...)函数中用到的视点参数。控制纹理坐标的生成，并使纹理与修数名texName绑定。注意texParms[]数组的值是基于视点的，采用1D纹理，图像的前向面绘制成红色，背面是蓝色，到视点的距离在D1和D2之间。

```
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_EYE_LINEAR);
D1 = ep + 1.0; D2 = ep + 10.0;
texParms[0] = texParms[3] = 0.0;
texParms[2] = -1.0/(D2-D1);
texParms[3] = -D1/(D2-D1);
glTexGenfv(GL_S, GL_EYE_PLANE, texParms);
glBindTexture(GL_TEXTURE_1D, texName);
```

texParms[]数组的值表示透视投影之后的3D视空间的结构，纹理映射之后还要进行深度调整，调整值x和y都为0，z和w有修正。

8.13.2 2D纹理例子

2D纹理映射的程序代码分为四步。首先，定义数据，建立内部纹理图（texImage）、用于纹理的纹理名（texName）。在init()函数中使用glEnable()允许使用2D纹理映射。第二步，将文件读入纹理数组中，第三步建立OpenGL函数，定义如何使用纹理图。第四步，绘制带纹理图的立方体表面。

```
#define TEX_WIDTH 512
#define TEX_HEIGHT 512
static GLubyte texImage[TEX_WIDTH][TEX_HEIGHT][3];
static GLuint texName[1];     // 参数是使用的纹理号
void init() {
    ...
    glEnable(GL_TEXTURE_2D); // 允许使用2D纹理图
    ...
}
```

```
void setTexture(void) // 将文件读入RGB8格式数组中
{   FILE * fd;
    GLubyte ch;
    int i,j,k;
    fd = fopen("penguin.512.512.rgb", "r");
    for (i=0; i<TEX_WIDTH; i++) {
        for (j=0; j<TEX_HEIGHT; j++) {
          for (k=0; k<3; k++) {
            fread(&ch, 1, 1, fd);
            texImage[i][j][k] = (GLubyte) ch;
          }
        }
      }
    fclose(fd);
}
```

```
// 最后一面可用纹理
    glEnable(GL_TEXTURE_2D);
    glGenTextures(1, texName); // 定义第6面的纹理
    glBindTexture(GL_TEXTURE_2D,texName[0]);
    glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL);
```

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB8,TEX_WIDTH,TEX_HEIGHT,
  0,GL_RGB,GL_UNSIGNED_BYTE,texImage);
```

```
glBegin(GL_QUADS);          // 第6个四边形：x面为负
glNormal3fv(normals[1]);    // 只有一个法向量用于Flat着色处理
  glTexCoord2f(0.0, 0.0); glVertex3fv(vertices[0]);
  glTexCoord2f(0.0, 1.0); glVertex3fv(vertices[1]);
  glTexCoord2f(1.0, 1.0); glVertex3fv(vertices[3]);
  glTexCoord2f(1.0, 0.0); glVertex3fv(vertices[2]);
glEnd();
glDeleteTextures(1, texName);
```

这是OpenGL中典型的纹理函数。首先定义数组，装入纹理数据，数据可来自文件或人工合成创建，然后通过以下序列：

- 允许纹理映射
- 为纹理名生成纹理
- 纹理与纹理类型（这里为GL_TEXTURE_2D）绑定
- 设置纹理环境
- 定义纹理参数
- 创建纹理图像

为纹理映射建立OpenGL环境。上面这个次序很重要但并不是唯一的，设置纹理环境和定义纹理参数也可以以其他次序出现，但最好先给出一个次序，然后用该次序进行工作。

8.13.3 环境纹理图

环境纹理图例子使用2D纹理图，对照片进行球面变形实现广角镜头模拟，建立环境纹理图的关键部分是纹理参数调整。这里还使用了glHint(...)函数定义透视计算和点平滑（当然这么做是有很大计算量的）。图8-15的结果表明付出这个代价是值得的。

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT,GL_NICEST);
glHint(GL_POINT_SMOOTH_HINT,GL_NICEST);

...

//  下面两行在SHIT纹理方向生成环境纹理图

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);
```

8.13.4 使用多纹理

当在OpenGL中引入多纹理时，建议最好给出使用的方法。这里就给出用两种纹理的例子。

定义textures[]数组的方法如下：

```
int textures[2]
```

在初始化函数中，进行以下纹理对象定义：

```
//  载入及绑定纹理
glGenTextures(2, &textures);

//  将第一个纹理数据载入暂时数据中
file.open("tex0.raw");
file.read(textureData, 256*256*3);
file.close();

//  建立第一个纹理
```

321

```
glBindTexture(GL_TEXTURE_2D, texture[0]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256,256, GL_RGB,
    GL_UNSIGNED_BYTE, textureData);
// 将第二个纹理数据放到数组中
file.open("tex1.raw");
file.read(textureData, 256*256*3);
file.close ();
// 建立第二个纹理
glBindTexture(GL_TEXTURE_2D, texture[1]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
    GL_NEAREST_MIPMAP_LINEAR);
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGBA, 256,256, GL_RGB,
    GL_UNSIGNED_BYTE, textureData);
```

在函数display()中，如下定义纹理元：

```
// 设置第一个纹理并绑定
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textures[0]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
// 设置第二个纹理并绑定
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, textures[1]);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
```

在display()或其他建立模型的函数中，可用以下代码将纹理坐标与顶点坐标结合起来：

```
glBegin(GL_TRIANGLE_STRIP);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 0.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 0.0);
    glVertex3f(-5.0, 5.0,0.0);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 0.0, 1.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 0.0, 1.0);
    glVertex3f(-5.0,5.0,0.0);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 0.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 0.0);
    glVertex3f(5.0, -5.0,0.0);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB, 1.0, 1.0);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB, 1.0, 1.0);
    glVertex3f(5.0,5.0,0.0);
glEnd();
```

8.14 小结

纹理映射是一种非常直接的过程，可以让用户用不同的方式在图像中加入大量的额外信息。通过本章的讲述，读者应该了解通过照片和人工图像创建纹理的方法，以及将纹理坐标与几何体坐标相关的方法。同时我们描述了在OpenGL图形API的相关用法。第10章将详述如何在创建图像时实现纹理映射的方法，最常用的是简单线性插值方法，有些还有一些透视调整纹理处理。

8.15 本章的OpenGL术语表

本章描述了许多OpenGL图形API，引入许多函数及参数，下面将一一详述，不过略去了一些很复杂的函数和参数。这个术语表的内容不是一个完整的手册，只是一个简明列表。

OpenGL函数

glBindTexture(...)：纹理名与纹理目标绑定

glDeleteTextures(...)：从活动纹理表中删除纹理

glGenTextures(...)：生成一系列纹理名

glHint(parm,value)：选择OpenGL函数的选项

glPixelStore*(parm,value)：说明像素在存储又是否压缩

glReadBuffer(mode)：定义读像素的颜色缓冲区

glReadPixels(...)：从颜色缓冲区读一块像素

glTexCoord*(...)：定义当前顶点的纹理坐标，(*表示纹理维数)，纹理坐标的数据类型，以及坐标以向量还是标量形式给出。

glTexEnv*(...)：说明纹理环境参数值

glTexGen*(...)：控制纹理坐标的生成值，选项*表示参数是否整型、浮点型、双精度型，以及参数以向量还是标量形式给出

glTexImage*D(...)：说明纹理图像，*表示维数为1、2或3，该函数有参数的个数

glTexParameter*(...)：定义目标纹理以及要定义的纹理属性并指定属性的值

GLU函数

gluBuild2DMipmaps(...)：建立一系列不同分辨率的预过滤的2D纹理集（mipmap） 324

gluQuadricTexture(quadric,value)：说明GLU四边形是否生成纹理坐标

OpenGL参数

GL_ALPHA：说明纹理数组取单个α值

GL_ALPHA8：说明纹理内部格式取8位整型α值

GL_BLEND：说明纹理值如何与像素混合

GL_BLUE：说明纹理数组取单个蓝色成分

GL_CLAMP：说明当纹理坐标超过边界时作纹理拉伸

GL_DECAL：说明纹理值如何应用到像素

GL_DONT_CARE：glHint()函数的参数，说明系统可以使用该函数处理的任何层次

GL_EYE_LINEAR：说明纹理生成时要考虑视坐标的参考平面

GL_EYE_PLANE：说明如何生成与视点平面有关的线性纹理，其后的参数表示视点平面参数

GL_FASTEST：glHint()函数的参数，说明系统进行最快速度的纹理映射处理

GL_GREEN：说明纹理数组中取单个绿色成分

GL_INTENSITY：说明纹理数组中取单个强度成分

GL_INTENSITY8：说明纹理数组中取单个8位强度成分

GL_LINEAR：说明纹理值为像素四邻域纹理值的平均

GL_LUMINANCE：说明纹理数组中取单个亮度成分

GL_LUMINANCE8：说明纹理数组中取单个8位整型亮度成分

GL_MODULATE：说明纹理值如何映射到几何像素上

GL_NEAREST：说明纹理值为最接近（距离$s+i$）该像素中心的纹理值

GL_NICEST：glHint()函数的参数，说明系统进行产生最佳质量的纹理映射处理

GL_OBJECT_LINEAR：纹理是通过世界坐标值产生的

325 GL_OBJECT_PLANE: 说明对象与纹理是线性映射关系，其后的参数定义纹理平面

GL_PERSPECTIVE_CORRECTION_HINT: 纹理映射的透视修正处理策略

GL_Q: 纹理图的各向同性维（第四维）

GL_R: 纹理图的第三维

GL_REPLACE: 纹理值替换到纹理映射的几何像素上

GL_RED: 说明纹理数组中取单个红色成分

GL_RGB: 说明纹理数组中取RGB三元组的成分

GL_RGB8: 说明纹理数组中取8位整型RGB成分

GL_RGBA: 说明纹理数组中取RGBA四元组的成分

GL_RGBA8: 说明纹理内部格式中取8位整形RGBA成分

GL_S: 纹理图的第一维

GL_SPHERE_MAP: 纹理映射为球面纹理图

GL_T: 纹理图的第二维

GL_TEXTURE_*D: 说明glTexImage*()函数的目标纹理

GL_TEXTURE_ENV: 说明glTexEnv*()函数必须的第一个参数

GL_TEXTURE_ENV_MODE: glTexEnv*()函数的第二个参数，说明生成纹理的模式(modulate, decal,blend,replace)

GL_TEXTURE_GEN_MODE: 后续参数定义的纹理生成的方式，其后的参数表示采用哪种方式

GL_TEXTURE_WRAP_*: 说明纹理坐标（*号表示）是拉伸还是重复的方式

GL_TEXTURE_MAG_FILTER: 如果要映射的空间小于或等于纹理空间，该参数说明是否使用纹理放大过滤器

GL_TEXTURE_MIN_FILTER: 如果要映射的空间大于纹理空间，该参数说明是否使用纹理缩小过滤器

GL_UNPACK_ALIGNMENT: glPixelStore*()函数的参数，说明内存中每一行像素是否使用对齐方式

GL_REPEAT: 说明纹理坐标超过边界时是否重复纹理

8.16 思考题

1. 有些纹理映射的OpenGL函数是有调用次序的，某些函数必须先调用，另一些必须随后调用，以某个序列出现，讨论一下为什么必须这么做。
2. 用GIF或JPEG文件图像作为纹理图必须做哪些工作？您的图形API还可以装入其他哪些格式文件作为纹理图吗？
3. 考虑一下使用多纹理可以产生哪些效果？从某些简单的效果开始，多纹理还可以产生受挤压的木料，
326 金属中的子弹洞，表面上的水珠等效果，想像一下还可产生哪些效果并且如何实现？如果你的图形API中有多纹理功能，创建并实现你的想法。

8.17 练习题

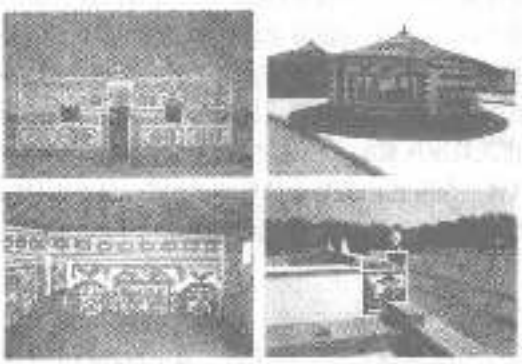
1. 由简单的来源创建纹理图。尽量多地采用以下可能性：(a)数字图片 (b)扫描图片 (c)截屏图像 (d)OpenGL程序的帧缓冲器内容，通过截获前帧保存的内容得到。将以上内容存为某种文件格式，并读入作为程序的纹理图。
2. 采用一些应用（如Photoshop）中的文本函数将一些文字写在图像的不同行上，再将该图像存为文件作纹理图。用一些点来分离不同的词或短语，写一个小的图形程序将这些文字映射到图像上。

3. 考虑2D空间中三角形的三个顶点坐标为V0 = (1,1)，V1 = (9,1)，V2 = (1,5)（逆时针走向）。取一点P = (3,2) = α*V0 + β*V1 + δ*V2，其中α + β + δ = 1。如果纹理大小为256 × 256，则三角形V0、V1和V2的纹理坐标分别为T0 = (50,20)，T1 = (180,70)和T2 = (30,80)。计算α、β和δ的值，以及P点的纹理坐标。如果这些顶点的纹理坐标不是整数值，讨论一下如何根据纹理空间的相近点来计算顶点P的颜色，并在OpenGL中通过过滤器选项实现。

4. 创建"芒草"纹理作为一个过程合成纹理的例子。在一个长方形内画许多平行线，每条线都有起点和终点，就像真正的芒草一样，每条线都从棕色画到晒杀色。这种纹理也可用于模仿树皮。图中给出了效果。可以通过指定坐标画线段，也可以随机指定坐标，但是线段要足够多，足以充满整个空间，多少数目的线段是足够多，这一点无法保证。 327



5. 除了扫描图片之外，用户还可以人工绘制纹理。这里画出了世界著名的南非Ndebele图像。观察一下图中给出的四种图案。你的任务是使用图像创建程序或相应的函数创建类似于这里的图案，并存为一种文件格式，用于纹理图（本章中可用JPEG格式）。



6. 创建与图8-3类似的人工纹理。选择2D整数空间中的随机点和随机颜色。对于纹理空间中的每一点，其颜色依赖于它的最近点。或者利用第9章提出的伪颜色，就像棋盘纹理一样使用该纹理，看看其效果。 328

7. 创建3D过滤器，3 × 3 × 3数组中都是非负数，总和为1.0，使其产生平滑随机纹理，类似于本章中关于2D纹理的情况。

8.18 实验题

1. 考虑纹理映射到表面的不同方法：GL_BLEND，GL_DECAL，GL_MODULATE和GL_REPLACE，创建一个带纹理图的场景，尝试一下不同纹理环境，并记录结果。

2. 考虑纹理过滤作用于图像的效果。创建一个简单场景（可以是单个多边形），其纹理图要么太大（每像素有多个纹元），要么太小（每像素纹元太少），对纹理使用不同的放大或缩小过滤器，讨论一下纹元与像素的比例关系，以及有效的过滤器。
3. 使用在色度深度处理中引入的1D纹理图概念，如第9章所示，可以表示高度值，还可将它扩展到轮廓线映射。
4. 采用一个绘制程序（如POVRay），试用它的纹理功能，目的是产生纹理的效果，特别是建立在噪声函数上的纹理。
5. 创建纹理图用于布告板技术，即创建自然图像并进行适当编辑，使用背景为单一图像前景的单种颜色。将编辑后的图像存为RGB未压缩文件，修改本章中的例子，读入原始RGB图像，并按本章的方法创建α通道，将本图像当作RGBA纹理，看看α混合后的效果。
6. 选择一种可以看到纹理细节的纹理图（如带有小正方形的棋盘纹理），将它映射到GLU四边形对象中。看看表面纹理有什么特别的地方。是否能找出几何表面上有特别意义的那些顶点。
7. 将第一个练习中创建的纹理用于图8-14中立方体的其他面上。
8. 使用人工纹理（如黑白棋盘纹理）来实验强度、亮度或混合纹理映射。结果应该能显示棋盘图案，但图案是通过多边形表面颜色显示出来的，棋盘本身是看不见的。
9. 输入一幅图像，输出“鱼眼”变换后的图像（如用工具软件Photoshop中的类似功能）。两幅图像都使用环境映射技术中的平滑技术，并讨论结果。
10. 使用交互技术来实验2D纹理映射到几何图上的情形。定义一个简单形状（如单个四边形）。选择纹理坐标，使用键盘或鼠标，将几何图在纹理图上移动（或移动几何图后面的纹理图）。使用交互技术对等地改变纹理坐标（每个纹理坐标分量相加相同的值），由此生成“纹理检查”工具。 329
11. 考虑图8-16的情况，将砖块纹理映射到锥体上。该图不是很成功，因为将砖块映射到锥体时不能保持形状一致。实验人工砖块纹理，看看是否能对砖块作变形，使它映射时在大小上一致。
12. 就像前一个练习中，创建一个简单的几何体，映射一个已知的纹理，使用交互技术进行纹理变形。例如，在一个顶点上仅移动纹理坐标，因此较多（或较少）纹理可以映射到几何体上。注意纹理变形方式，讨论图素改变的方式。用不同的反走样过滤器完成这些工作，并讨论过滤器影响纹理变形的效果。
13. 许多机构都有图像处理程序库，读者在网上也可找到。找一个带压缩格式的读图像文件的程序并显示它，将它用作读取纹理图的工具并创建图像数组。用该函数将压缩格式的文件读入作为纹理图。
14. 写一个函数可以读RGB格式的图像数组，再通过给出α值创建RGBA格式的数组。可以将其中的一个原色赋给α通道，也可以将三个原色中的最大值作为α通道。用它来修改简单的RGB图像数组，并创建RGBA数组，再将该RGBA数组作为纹理图，试验带α值的纹理映射。

8.19 大型作业

1. （小房子）以Ndebele房子为例创建纹理图，用于前一章中设计的小屋的外墙和内墙，看看在小屋中漫游的情形。试验后可以发现，漫游的刷新速率会降低，因为计算机重建纹理会占用大量的时间。
2. （场景图分析器）将纹理映射加入本章中提到的场景图分析器中。在display()函数中写入适当的 330
OpenGL代码。

第9章　图形在科学计算领域中的应用

本章介绍计算机图形学作为解决问题的工具在自然科学中的应用，集中描述了讨论已久的视觉交流问题。举例说明用各种图形学技术展示不同种类的信息，包括图形学的模型和图像。这些图像是通过编程得到的，同时用在看似简单但却很实际的科学问题上。本章的几个源代码示例包含在本书的附加材料中。因为这些技术的范围相当广泛，不容易按序介绍，所以，按照作者的思路进行。读完本章后，读者应该很好地掌握在自然科学中使用图形建模和仿真技术，懂得这些技术如何创建图像来表达各个自然科学领域中的各种数据。

要从本章学到知识，读者需要通过建模、视图变换和色彩来理解计算机图形学的基本概念并进行有效实践。这也需要作者具有足够的编程经验，运用本章介绍的多种程序设计技术方法产生图像。如果对这些问题有比较好的基础，那么在学习的过程中会事半功倍。

9.1　简介

在过去的二十年中，科学理论的不断发展和大量的科学数据不断涌现导致了图像在展示一些概念和实验上的广泛应用。这种展现方式称为科学可视化，现在，它是大多数科学工作的关键部分。科学可视化的重要之处不是用产生的图像描述科学原理或过程，而是创建图像所要解决的图形问题，高效地展示图像所需的视觉传达，以及让科学界理解图像的科技含义，这有助于学生学习科学知识，帮助公众更多地了解科学的发展；帮助投资者更好地对投资的科学研究项目进行决策，帮助科学工作者更全面地理解所研究事物的含义。在人类发展过程中，对图像的理解进化得最好，所以，诸如“我看到它了！”之类的一般表达绝不是偶然的。

不管在科学领域内还是在领域外，计算机图形学的作用是对问题提供一个更好的理解方式。我们用可视的图形术语表达问题，通过建立实际的图像或图像集把问题具体化，最后把图像作为对问题进行反映和深入理解的工具，由此导致对问题的更深入理解。如图9-1闭合环的描述，最初解决问题的任务由问题→图形的过程完成。我们必须了解自然科学，并找到一种方法对自然科学建立模型，通过图形的方式来表现模型。那个模型在计算机图形学中的具体化表示在问题→图形和图形→图像链接中，它用图形学建模来表示自然科学的模型，就像用图形表现的模型和用户与模型之间的交互一样。如果自然科学的模型和由图形及交互组成的视觉传达都能很好地完成，那么就给用户解决问题提供很好的指导性的作用，可以让用户更好地认识问题，就像在图像→观察过程中表示的，通过这种计算机图形学提供的方法，可以完成认知→问题的过程，也就是提高认识的过程。综合来说，通过结合分析和可视化过程，可以提供一个非常有用的方法从而可以描述任何一种问题。

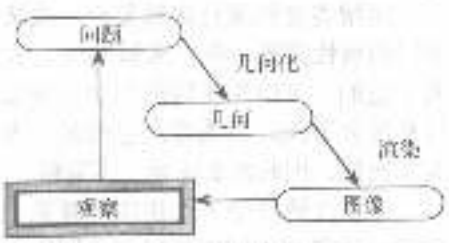


图9-1　图形问题求解循环

当考虑问题以及如何用图形和几何的方式来观看它的时候，会有一些基本问题需要回答。能用一些本身就有图像的自然物体（汽车、房子、机器、动物……）或者熟悉的表示（棒形、球形、金字塔形、平面……）来实现要求解的问题吗？如果是，就用这些自然的或者熟悉的

物体开始，看看如何用它们对问题进行描述。如果不是，就需要试着寻找或者发现描述这个
332 问题的方法，因为没有图形就不能很好地表现要解决的问题。正如伽利略所说。

> 在这部重要的书中，自学被描述成一直凝视着我们的万物。如果不能理解语言或者读懂字母，是不能理解这本书的。它是用数学语言描述的，它的基本文字是三角形、圆和其他的几何形状。没有它们，人类不可能读懂；没有它们，人类只能在一片黑暗中徘徊[SOV, P16.]。

我们把这里描述的所有过程称为图形化的问题解决。这个过程的关键是确定一个方法，这个方法可以用图形术语来描述问题，允许设计展示一部分问题的图像。表述问题并使问题更好理解的一般过程称为建立问题模型，建立自然科学问题模型的范围很广，包含在自然科学内部，经常使用应用数学如微分方程组来建立模型。然而，图形化问题解决的一部分是帮助理解模型建立过程，本章阐述一些已经应用于各种自然科学建模的图形建模方法。这些模型相对比较简单，因为我们考虑通过基本的图形工具就可以创造的图形模型。如果要建立并使用更复杂的和成熟的模型，那么应该去看科学可视化的文章来加深了解。

当我们在考虑图形建模的例子的时候，需要考虑不同类型的自然科学问题。每一个问题都要对它进行描述以及如何用图形进行建模，并在选择特殊的展示方式和适当的描述之间做出权衡，然后描述根据模型建立图像的方法。随后，本章会讨论为模型产生图像的计算机建模方法（不同于问题建模）的一些细节。这些有时会包括图形技术的讨论，有时又集中在一些程序设计问题上，我们的最终目的是描述问题的解决过程，并能够应用到项目和问题上。综合这个过程的例子来看，这可以应用图形化问题解决方法来达到目标。

数据和视觉交流

在讨论科学应用的例子之前，先说一下在处理科学信息的时候遇到的一些数据，因为必须仔细地使用适合数据的模型。不同种类的数据称为区间数、有序数和标称数。当用图形展示它们的时候，需要注意不同的方式。区间数用实数表示，有序数有自然的顺序，但是没有意义明确的数字表示，标称数是在不同种类中无序的数据。处理区间数有一定的困难，比如温度、力、花费或价格。我们对这些数据的数值特性很熟悉，并能用传统的方法表述出来，例如维度甚至颜色渐变。对于有序数，可以用多/少，大/小等概念来描述，举一个某人受教育的程度为例，可以分为小学、中学、高中、大学或研究生等水平。我们可以用位置，相对大小、递增亮度的颜色离散集合，或某个其他的方法来观察相对的大小。对于标称数，有一组不同的属性需要考虑，例如男/女，农村或者地方的州，或者是区域联盟。这种数据不能排序，
333 对于它们，可以用不同的形状或颜色，但是可能需要用图例来区分每个值的表示，因为没有自然区分方法，但需要注意的是，观众中可能存在着无意识的层次性，原有的展示方式暗示着层次性，因此需要注意一下策略。

许多自然科学工作往往用测量（就是区间）数，当然也存在与研究学习和理论关联的标称数，这包含不同的环境和不同的测试群体。对不同男女患者的药物测试就是这方面的一个例子，病人的种类就是标称数。然而，测试结果可能是区间数，显示方法可能是一对直线或者平面图形，每个对应一个区间数。对比这些图就可以对比标称组的值。这样，在本章就可以不用看任何标称数的处理方法。

许多科学研究或理论的特征是高维度，它们包括对一个点的一些采样，或者一些相关的变量。当处理高维的区间数以及数据的维数（或者函数定义域或值域的维数和）超过了三维，可能需要允许用户在观看数据时做一些选择。这在第2章已经讨论过了，但探测科学数据包括给用户提供在观察的高维度数据方面更多的控制。通过提供各种投影以及对数据移动的控

制来提供数据的探测。我们经常把二维屏幕认为是三维空间的投影，所以也需要把高维空间投影到三维空间上。随着智能三维沉浸式观察设备的提高，将来也许没有必要投影到三维空间上。

9.2 例子

正如此前注意到的，在本章中，描述了很多可以用在自然科学工作中的技术。在某种意义上，展示一系列的技术可能导致把焦点集中在图形技术上，而不是所要检查的问题上。所以，要在开始的时候先考虑问题，然后再寻找图形学或者可视化的表现。然而，我们的目的是在遇到问题的时候，提供一系列的指导，学会分析问题并找到一种合适的技术。这种技术只能通过实践得到。

我们的技术来源于多年来对展示的观察和对科学可视化方法的讨论。我们随后给出的例子并不属于展示中最成熟的图像。因为我们想把展示工作变得尽量简单，即基于简单的图形API，比如OpenGL就可以编写自己的程序，而不是成熟的科学可视化工具。如果掌握了这些简单的技术，就会有一个非常好的基础来理解更复杂的工具的工作方式，并且更好地利用它们。

9.3 扩散

扩散是一种广泛观察的过程，在这个过程中，呈现在空间中的某个点的属性随着时间从原来的点变迁到相邻的点上，在整个空间上蔓延。很多不同的过程会形成这种变迁，但是我们对它们并不感兴趣，只对变迁的事实感兴趣。当对扩散过程进行建模并用于计算时，通常把空间分成一些“点”的区域，可以是面积单元，也可以是体积单元，空间里每个网格点的 334
属性数据都假设有初始值。这个属性可能是单位体积水溶解盐的多少，也可以是单位体积物质的热量大小，当然也可以是单位区域的事件数目。建立这个过程的模型的前提是从给定点到相邻点转移的属性数目与两个点属性数量的差异成比例，可能是决定性地，也可能是随机地包含在过程中。这是一个非常通用的过程，可以应用到的问题非常广泛，在这一节中，我们看通过它建立的两个模型。

9.3.1 长条材料中的温度

我们来正式地看一下在第0章提到的例子。这个例子可以将理解的非正式的可视化与一个更科学的模型建立联系。首先来看一个某种材料的矩形长条，这根长条放在绝缘介质中，上面有一系列固定的温度连接点（可以看成是热源）。我们的目的是考察长条的热分布随时间变化的情况。假设长条具有常数的厚度，长条的材料在各个点处不随厚度而发生改变，因此可以将长条看成是一个2维的实体。长条可能是同质的（各向同性），也可能是异质的（各向异性）；长条的材料热传导性可能变化很大；长条上的连接点的活性可能不同，其温度随着时间而改变（但是连接点的温度是由外物决定的，与长条本身没有关系）。最基本的热分布可以用一个热力学方程 $\frac{\partial F}{\partial t} = k\frac{\partial^2 F}{\partial x^2}$ 表示，这是一个表示热传导的偏微分方程，其中k由长条的材料决定。如果材料是同质的，则k是一个常量；如果材料是异质的，则k是一个与长条位置有关的函数。后面这种情况的偏微分方程很复杂，在这个简单例子中不做考虑，但对于包括诸如绝缘子的不同材料的热力学模型是很有用的。

从方程看，在长条的某个位置，温度（随时间）的变化率与温度对空间的梯度成正比（即二阶导数）。也就是说，温度随时间的变化是与温度在空间上的传导有关的。如果温度随

空间的分布是恒定的，那么不管分布情况如何，温度变化率dF/dt都是0，即温度不随时间发生变化。温度在时间上的变化必然是由温度在空间上的分布引起的。我们的目标是在给定初始条件和边界条件的情况下，估算给定的任何时刻长条的近似热分布情况。

在为长条的热分布建立模型时要考虑3个主要因素：出于计算的目的，应该如何表示这个分布；如何定义长条的热属性；以及如何显示结果以展现长条上温度的变化情况。

对于第一方面，应该直接求解偏微分方程，或者将长条定义成网格模型，即把长条看成网格的2维数组，那么从一个网格到邻近网格的热传导过程就是与最初网格的温度的正比例关系。如果相邻的两个网格的温度一样，那么经过相同大小的热传递后两个网格仍然保持温度不变。

对于第二方面，应该将温度变化过程看成一个简单的散热过程，主要考虑网格之前的热流。标准情况下，一个网格和周围4个相邻的网格有相似的特性。如果一个网格保留的热能是
335 原来的α，那么传递给每个相邻的网格的热能是$(1-\alpha)/4$。从一个初始状态开始，根据条件更新每个网格的热能，用计算的值来替换假设为常量的值（如那些有固定温度连接点的地方），然后在更新之后显示每个网格中的值。可以把所有网格的α设为相同的常量，也可以为每个网格设置不同的α对异质的材料建模。在这个问题的实际解决过程中，要确定材料是同质的还是异质的，确定热连接点的位置以及热连接点的特性。如果将问题简化，可以假设材料是同质的，热连接点在固定的位置并且温度恒定，下面会讨论这种情况。我们也会考虑异质材料的情况，并且给出如何处理热连接点温度不恒定或位置不固定的情况。

对于第三方面，我们需要回顾第2章讨论过的视觉交流问题。在第2章我们看到，可以用不同的颜色或者高度来表示长条上不同位置的温度，我们也讨论了不同的显示方法给观察者的印象。我们将使用颜色和高度来表示在长条上的温度变化，因为这似乎是最强有力的视觉表现形式。图9-2给出了结果图，代码在本书的最前面一章已经给出。



图9-2 长条中温度的简单表示（有固定温度连接点），参见彩图

对于上面提到的最简单的情况，结果也很容易理解。长条的温度在靠近最热连接点的地方温度最高，在靠近最低连接点的地方温度最低。这里不显示温度和颜色的关系图例，这样可以让信息更充分；我们也没有显示连接点的位置和温度信息，尽管这会随着时间而改变。用户可以通过控制连接点的温度和位置来获得长条上热分布的不同结果，我们也建议读者尝试不同的可能性看看发生的不同情况。我们特别建议读者观察这样一个长条，长条的连接点集中在一端并且随着时间温度在热和冷之间变
336 化。读者应该可以发现热和冷的热波从连接点开始移动，而且可以解释这种现象的原因。

对于更加复杂的异质材料，假设关注的是长条的有的区域不会太热，至少不会迅速变得很热，不管热连接点注入多少热量。我们可以创造一个不同的地方具有不同热传导性的组合长条。假设热连接点在一端，在两端之间有一部分是绝缘介质，对于处在绝缘区的网格，将其热传导常量设置得比较小，使得热量传进和传出网格都比较慢。这种有绝缘区的长条相对没有绝缘区的长条，热传递要慢一些，并且可以看到热连接点附近的点升温也慢很多。我们可以为这种异质材料建立更一般的模型，为每个网格存储其材料的热传导属性，从而调整热传播的模型，使得在这类物体上热运动的方式更容易理解。

这些例子很好地给出了热分布的定性解释，但是这种解释的准确性如何，是否可以更加准确呢？我们可以将连续的过程离散化，从而用一个传播模型来估算热传递的物理性质。我们可以将模型得到的结果和测量到的实验结果进行比较，为了使估算结果更加准确，我们可以增加网格点的个数来减小离散化估计的误差，同时缩小时间的步长，相应调整模型参数以

更小步表示。这样处理可以使建立的热分布模型更容易理解，在很多情况下帮助我们选择更合适的热材料，从而改善为热敏感环境的设计。甚至可以通过建立包含绝缘材料部分的异质材料的模型，使模型更加复杂和成熟。

9.3.2 疾病的传播

作为基于传播模型的另外一个应用，我们来看一下由不同社区组成的某个区域的传染性疾病的传播过程。我们没有必要指明是什么疾病，因为很多疾病的传播过程都类似，也没有必要特别强调各个社区的行为。但是，我们需要建立一系列假设，使得对这个应用的建模结果更易于显示。

最基本的假设是，疾病传播的基本机制是：要发生疾病传播，必须有感染者和易感者（没有感染也不具备免疫功能）之间的接触。当发生接触时，易感者就有一定程度的可能性会感染上疾病。因为不可能为个人之间的每次接触建立模型，因此假设两个人群之间的接触机会与两个人群中的人口之间的乘积成正比。进一步假定社区之间的分布呈网格状，每个社区就是一个大的矩形区域，只有相邻社区的人才会有接触，这里的相邻是指社区处在同一行或同一列，并且位置接近，采用一个索引。

假设最初的时候，每个社区的所有人都是该疾病的易感人群，再假设这种疾病不是致命的。一旦感染，经治愈就对疾病产生免疫能力。假设感染者在经历单位时间后有β的概率可以康复（疾病周期管理）。如果易感者与感染者接触了n次，则易感者有$n\alpha$的概率感染疾病，而接触次数n在上一段中已经确定。

基于这样的假设，我们建立的模型包括一个pop[][3]数组来表示在二维网格中每个社区易感者、感染者和免疫者人数，每一行表示一个社区。对于模拟的每一步，我们可以通过对每个点上易受感染人的数量和相邻点的感染人数（包括点本身）进行乘积来计算该点上被感染的人和易感染的人的会面次数。从感染人群中减去计算得到的康复人数，对免疫的人数进行更新，然后从该点本身和相邻的节点中加上新感染的人数。最后，从易感染的人中减去新感染的人数。这些计算的实现是通过使用类扩散（diffussion-like）模型实现的，它对新的感染使用基于二维的过滤器m[3][3]，这个数组定义了和每个相邻的单元相遇的可能性：

```
infected[i][j] = a*pop[i][j]*(m[1][1]*pop[i][j] +
                 m[0][1]*pop[i-1][j] + m[2][1]*pop[i + 1][j] +
                 m[1][0]*pop[i][j-1] + m[1][2]*pop[i][j + 1])
```

必要的时候对cell[i][j]中的免疫和康复人数作适当的更新。

该模拟从一个初始值开始，这个值定义为每个单元的感染数。接着进行逐步计算，通过扩散模型可以得到每个单元中的感染人数是如何改变的。一旦计算了新的数量，就更新显示并重新绘制。图9-3显示基于这个模型建立的模拟动画的一帧。通过包含一个没有人口的区域，使该模拟引进了一个额外的特征，并改变了扩散的形状，但并没有改变感染会蔓延到整个区域的事实。该模拟的代码包含在本书的附件材料中。

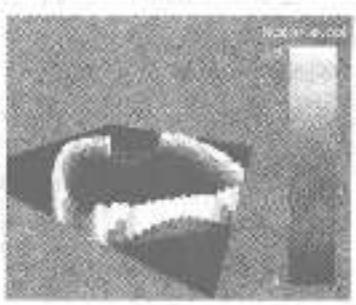


图9-3 表现隔离群体中行为的疾病传播模型

这个模型预言了社区中的每个人最终都会对该疾病免疫，所以该疾病经过一次后最终会从该地区中永远地消失。这显然不是合理的结论。如果对这个模型的弱点进行检查，我们就会发现导致这些不准确的原因。假设没有新生儿，因为新生儿并没有对疾病免疫，而

且除了相邻的往来外，没有人口流动，所以该地区的某些地方新病例是自然产生的。同样假设疾病本身的结构是不变的，这样先前免疫的人就不会易感染。所以这个模型是相当简单的，但它在某种程度上展示了传染病的属性。如果对该模型添加一些附加功能，就可以对它进行改进。

9.4 函数作图和应用

我们习惯于在数学、科学和工程课本中用曲线和曲面来对函数作图，通常把曲面和曲线认为是理解问题和寻求答案的有效工具。事实上，我们如此地习惯于用图形、曲线或者曲面来描述问题，以至于很难把曲线和曲面的标准绘制认为是需要讨论的问题。这里我们还是帮助读者了解这个方法，并让它们变得更加高效。

绘制单变量的实函数图像是非常简单的。函数f的图像定义为$(x, f(x))$的点集，x的值属于函数的定义域。绘制这类图像和在学校里学的是一样的：在定义域中取样一些值，计算域内每个值的函数值，然后依次连线。我们对这种初等数学的二维作图非常熟悉，所以不再进一步讨论，但需要确定是否能用程序绘制这样的图像。

绘制两个变量的实函数图像稍微复杂些，但我们在第2章已经讨论过了这方面的基础。“波纹”函数是以这种方式作图的简单表面的一个例子，它表现的是从中心向外扩散的一组同圆，如图9-4所示。在第6章的时候看到过这张图片，但那时关注图像的表现方式。之所以把这个函数称为“波纹”，是因为它的形状像是把一块石头丢在水中产生的水波。通过此前描述的方法进行计算，绘制的函数是$z = \cos(x^2 + y^2 + t)$。在整个方形的定义域$-5 \le x \le 5$和$-5 \le y \le 5$内，对x，y进行双重循环，并对每对点(x, y)计算z值。这些值可以存储在二维的z值数组里，也可以实时计算，然后在整个定义域进行循环并绘制组成表面的三角形。注意，函数的参数中包含了t（可以认为是时间）。这个参数是递增的，这样余弦函数的值也递增。每次图像重绘时在idle()函数里通过对t增加一个常量来使它线性地递增。这个图像就运动起来了。波浪从中心向外面连续地移动。这种参数化的编码给波浪随着时间的实际运动建立了模型，而且是一个基于时间的显示的很好的例子。通过使用大量的三角形、光照和材质技术，使得这个图像非常平滑，正如在第6章讨论的。



图9-4 函数表面显示的例子

339

这种类型的绘图代码非常直观。假设定义域是从X_0到X_1和Y_0到Y_1，并对每个方向计算N步，那么：

```
Xstep = (X1-X0)/N; Ystep = (Y1-Y0)/N;
    for (x = X0; x += Xstep; x < X1)
        for (y = Y0; y += Ystep; y < Y1) {
            xx = x - Xstep; yy = y + Ystep;
                glBegin(GL_TRIANGLE_STRIP);
                    glVertex3f(x,  y,  f(x,y));
                    glVertex3f(x,  yy,  f(x,yy));
                    glVertex3f(xx, y,  f(xx,y));
                    glVertex3f(xx, yy,  f(xx,yy))
                glEnd();
}
```

当然有很多方法可以使这个程序更加高效，但这个简单的代码可以计算并显示组成表面的基本三角形。

很多问题都可以用数学函数来理解，如果用图像来辅助的话，可以理解得更好。例如，对于平面上的静电电荷，要了解平面上点的静电电势分布，可以从把点(x, y)的标量静电电势

记为P并始，当点(x_i, y_i)的电量是Q_i时，根据库仑定律：

$$P(x, y) = \Sigma \frac{Q_i}{\sqrt{(x-x_i)^2 + (y-y_i)^2}}$$

对任何固定点固定电量的集合，这个等式定义了两个变量的函数，它可以按照此前描述的方法进行作图。这个函数很简单，如果只有这个等式很显然是不能说明静电电势的属性的。图9-5针对给定点的一个正电荷和二个负电荷的特定结构描述了在一个矩形内的静电电压。正如在第5章可视化交流中描述的颜色问题一样。从这幅图中我们可以清楚地看到这种电压犹如一些长钉立于一张弹性的薄片上方或下方，其具体的相对位置取决于电荷的正或负。通过在图像中引入度量尺度，可以从二维的伪彩色平面估计出某一点处电压的实际值。



图9-5 从单一平面上三点电荷生成的彩色表面（一点为正极，另两点为负极），分别以一个三维表面和一个伪彩色平面表示

对于一个给定的点和一组移动的电荷，它们的电量是固定的。如果要建立它的静电电势图，可以通过某种方法选择一个电荷点，通过鼠标或键盘在平面内移动它。同样，如果对于固定的点，但它的电势是可变的，也可以选择那个点并对它的电势进行改变。这两种技术都会让观察者看到某点电势的变化，通过少量的试验，就可以得到感兴趣点的电势，甚至还可以进行更多的试验，从而观察是否存在其他可以得到相同电势的更简单的方法。由此，图像提供了一种于目标位置上建立电压的交互工具。

函数曲面的另一个示例表现的行为差异是一种交互波现象。有两个波函数（也可以有更多，不过我们限制为两个以便于讨论）。任何一点处的全局位移都是这两个函数的和。我们需要考虑两类波：波列和源于给定点的波。波列是一组沿单一方向移动的并行波，可以用形如$f(x, y) = a\sin(bx + cy + d)$的函数进行描述，其中$a$是振幅，$b$和$c$分别决定了频率和方向，$d$是位移。在每次重绘图像的时候增加位移值，就把波的行为形象化。由此，两个波列的行为就可以通过两个这种方程的和函数给出。图9-6的左边显示了当一个较高频率的波列与一个幅度相同但频率较低的波列以大约120度角相交的效果。通常，将水波建模为不同角度、频率和幅度的多个波列之和。

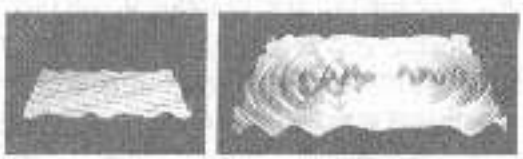


图9-6 两个以小角度相交的波列（左），以及与原点偏离3π/2的两个环形波（右）

第二类波函数由一个给定点处的波形给出，其行为与图9-5相似。这种波的函数的一般形式是从先前的“波纹”例子函数$f(x, y) = a\cos((x-x_0)^2 + (y-y_0)^2)$扩展而来。所以，每个这种波函数都是由初始点$(x_0, y_0)$、幅度$a$及频率$b$来定义。而对于分别有各自的初始点、幅度和频率的两个波则通过这两个函数的和给出。如果有二个（或更多）这种波函数叠加在一起，它们就会形成非常复杂的干扰和叠加模型，如图9-6右边所示，这两个波函数都与中心稍微偏离，而相同的幅度和频率相互叠加，从而形成复杂的表面。当然，这与向湖水中抛入一块石头得到波纹是一样的，也可以观察到波列和环形波的交互过程。

9.5 参数曲线与曲面

参数曲线由二维或三维空间中的线或线段的函数给出。这种函数可能是解析式（由一个或一组公式表示），也可以是数据值的插值。现在可能使用解析式曲线更容易一些，不过在第

8章会看到一些插值曲线的例子。

作为三维空间内解析式曲线的第一个例子，先来考虑用两个变量画圆，而第三个变量用来控制移动圆的绘制，看看我们能得到什么。第一个版本，使用参数来定义圆形的半径和它与给定平面之间的偏移，令这个参数为t，则有如下参数方程：

```
x = a*t*sin(c*t)+b
y = a*t*sin(c*t)+b
z = c^t
```

对于实数常量a、b和c，看到一条类似倒置的熏香的圆锥。当此曲线向上移动时，底部的圆形半径会有所增加，如图9-7左图所示。

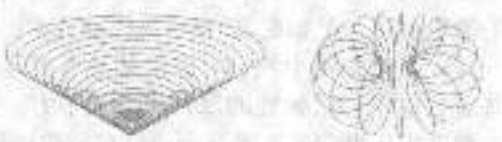


图9-7 饰形螺旋曲线（左）和环形螺旋曲线（右）

在此类例子的另一个版本中，圆的半径是个定值，以第三个变量作为绕中心线的角度沿着一个环形面移动。当缓慢增加该角度的值时，显现的效果就如一个末端交汇的线圈。使用参数方程

342

```
x = (a*sin(c*t)+b)*cos(t)
y = (a*sin(c*t)+b)*sin(t)
z = z*cos(c*t)
```

对于实数常量a、b和c，$(a*\sin(c*t)+b)$是圆的参数形式，方程的其他部分则控制着此圆沿着另一个圆的移动。因此，第一部分定义了一个螺旋线，而另一部分控制它在另一圆内的移动。图9-7右图所示螺旋形的相应参数值为$a = 2.0$，$b = 3.0$，$c = 18.0$。图中所示的螺旋形是当t在0到2π间取值时，绕着环形面移动的结果。以小步长取t值计算这个参数方程，得到一系列的点，将它们用直线相连，产生一条相当平滑的曲线。如图9-7所示，以偏离原点三个单位的点为圆心，以2为半径，此螺旋形绕过环形面18次（$c = 18$）。

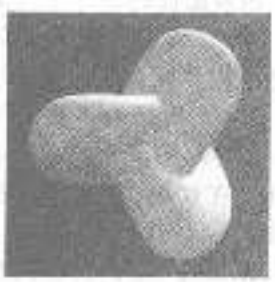


图9-8 示例中的凸雕表面示意图

相比之下，参数表面会稍微复杂一些。对于每个表面我们以多种不同方式将平面内的二维区域映射到三维空间当中。这种建模可能需要一定的布局，但可以得到一些有趣的效果。考虑一个凸雕表面，它的定义域为一个矩形域，参数s和t的变数均为$[0,\pi]$。如图9-8所示，相应的示例代码包含在本书附带的资源当中。

```
x := cos(t)*sin(s);
y := sin(t)*sin(s);
z := cos(s);
f := 1/2*((2*x^2-y^2-z^2)*z*y*z*(y^2-z^2)+z*x*(x^2-z^2)-x*y*(y^2-x^2));
g := sqrt(3)/2*((y^2-z^2) + z*x*(z^2-x^2) + x*y*(y^2-x^2));
h := (x+y+z)*((x+y+z)^3 + 4*(y-x)*(z-y)*(x-z));
plot3d([h/8,f,g], s=0..Pi, t=0..Pi)
```

343

通过从Maple代数系统中选取参数函数，可以为此示例编写代码。该系统可以从如下地址得到：http://www.geom.uiuc.edu/zoo/toptype/pplane/boy/。

该Maple代码转换后的代码中包含s和t从0到π的循环，对每个点(s, t)，分别计算$u = h/8$，$v = f$，$w = g$三个坐标。最终曲面是绘制两个三角形得到的，这两个三角形是对从定义域中的四边形的四个点进行划分得到的。

```
(u(s_i,t_j), v(s_i,t_j), w(s_i,t_j))
(u(s_i,t_j+1), v(s_i,t_j+1), w(s_i,t_j+1))
(u(s_i+1,t_j+1), v(s_i+1,t_j+1), w(s_i+1,t_j+1))
(u(s_i+1,t_j), v(s_i+1,t_j), w(s_i+1,t_j))
```

图9-9显示一个更复杂的参数表面的例子，称为(4,3)环形面。在这里，我们采用了平面长方形并把它划分成三个部分，将交叉部分折叠成为正三角形。然后将此三角形扭曲大约4/3周并将其围绕一个圆环进行拉伸，最后把三角形管的两端重新放到一起。最终的曲面是单面的（也可以跟踪所有表面面保证没有和任何的三角形边相交），操作起来也很有趣。在第15章讨论硬拷贝的时候可以看到用不同的三维硬拷贝技术构建的此类表面的图例。

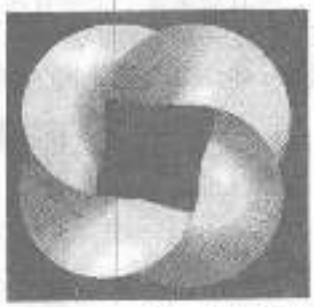
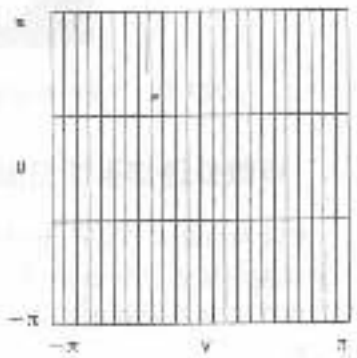


图9-9 一个二变量三维参数表面

对此表面进行更加仔细的观察，u和v的定义域是矩形域，分别为：$-2\pi \le u \le 2\pi$，$-2\pi \le v \le 2\pi$。此表面的方程为：

```
x(u,v) = (4+2cos(4u/3-v))cos(u)
y(u,v) = (4+2cos(4u/3-v))sin(u)
z(u,v) = 2+sin(4u/3+v)
```

这些方程看上去相当的熟悉，因为它与前面描述的螺旋管形的曲线非常相似。仅有的区别在于对螺旋管形的曲线我们仅针对t使用小值步长，而这里则是针对v使用小步长（绕着圆环100步），同时参数u只递增3次，在每步之间留有巨大的空间，使得交叉的曲面三角化。如图9-10的参数空间布局图所示，总的来说，在进一步定义更多细节的曲面时，用这样的方式给出的参数空间是非常有帮助的。特别在参数空间的布局上，能够看出u空间用于创建三角形，v空间用于创建围绕圆环面的细分步数。也许能推测出用四步或者其他数目的步数代替u空间的三步会产生什么结果（读者看到方形的交叉了吗？）。如果这样做，就必须注意调整方程中的常数4/3，这样才可以得到闭合表面。



344

345

图9-10 针对图9-9所示的曲面的基础参数空间，通过在v参数上减少步数得以简化

对于曲面的实际显示，从简单参数空间获得的又长又窄的三角形产生的着色处理效果非常奇怪。通过把每一个长矩形分解为一系列小的矩形以得到更加接近等边的三角形，就可以解决该问题了。

如果觉得这些参数化的曲面简单，那么可以把二维定义域映射到四维空间并观察产生什么来测试一下几何直觉。当然，四维空间可能不能直接显示，所以需要尝试利用各种投影把结果映射到三维空间。有一些非常经典的此类表面，例如克莱因瓶。一系列用于克莱因(Klein)瓶的参数化方程（只在三维空间）如下：

```
bx = 6*cos(u)*(1 + sin(u));
by = 16*sin(u);
rad = 4*(1 - cos(u)/2);
if (Pi < u <= 2*Pi) X = bx + rad*cos(v + Pi);
   else X = bx + rad*cos(u)*cos(v);
if (Pi < u <= 2*Pi) Y = by;
   else Y = by + rad*sin(u)*cos(v);
Z = rad*sin(v);
```

这些都是从数学函数中翻译过来的，图9-11左图是通过把图9-9的环的函数进行替换得到的，把定义域从[-π, π]变成[0,2π]。事实上，一旦有了参数化表面的代码，那么把它用于不同的曲面将非常容易。

克莱因瓶的实际构造比这里所示的复杂一点。克莱因瓶定义域在二维空间里是个矩形，跟上面所述的扭转圆环的定义域相似，但是这个函数对这个定义域有个难处理的地方：如图9-11中的右图所示，按如图所示的各侧标识，标记为b的两条边在柱面形成过程中必须匹配，

346

而标记为a的两条边必须是定向的。这个过程不能在三维空间中实现，但却能在四维空间中实现，而且结果像一个带有回进的柱体，而且这在三维投影中只能暗示。参数化方法是实现克莱因瓶的一种有效方法，它在三维投影下有一个特殊的性质，而且具有一些十分有趣的几何属性，如：四维空间中的克莱因瓶可以用2个三维默比乌斯带粘合在一起表示。

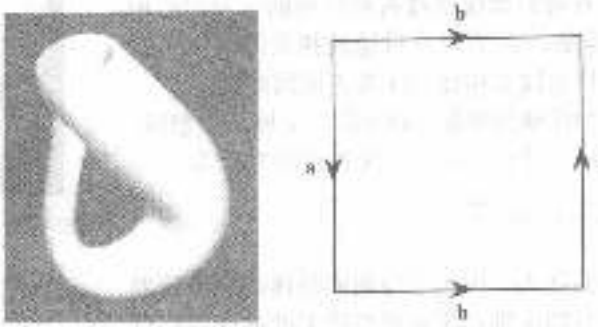


图9-11 一个四维参数曲面。克莱因瓶（左），定义它的参数区域的结构（右）

9.6 极限处理结果的图形对象

曲线和曲面并不是全由用先前讨论的封闭函数形成。非封闭函数形成的曲线或曲面具有许多有趣的属性，比如用极限处理绘制的模型。因为极限法是与分形和递归函数相关的，所以将在第14章中详细描述。本章先给出一个例子，例子中的函数定义了曲面绘制所需要的参数。

一些用极限处理绘制的模型十分有用。根据微积分知识可知，在某点可微的函数在该点处必连续；反之则不一定成立，比如不可微的连续函数。虽然微积分中介绍了许多这样的函数，但是我们很难看到一个直接、可视的效果。为此，我们绘制牛奶冻（blancmange）函数的曲面模型，它看起来非常像经常英国在假日供应的多块状牛奶冻的布丁。该曲面由递增的分段双向线性函数和定义，用递归实现。曲面上点的坐标是$(i/2^s, j/2^s, z)$，当i或j是偶数时$z = 0$，两者同时是奇数时$z = 1/2^s$。由于函数值在任意点处的和都小于$\Sigma_s 1/2^s$，因此，该函数是一个收敛的几何序列，而且分段序列的和也收敛，因此函数值比较好地覆盖在曲面上。但是，在曲面上任何点的邻域$(x, y, f(x, y))$内有许多关于i, j的偶数点$(i/2^s, j/2^s)$。在这些点处有尖角，所以函数不是处处可微的。图9-12是该函数的近似图形，我们可以通过增加该函数曲面上定义域中的点的数据来增加迭代次数。随着迭代次数的增加，计算量呈几何级增加。这使得对函数曲面的交互操作变得十分困难，因此要合适选取迭代次数。更多关于牛奶冻函数的数学介绍，请看[TAL]。

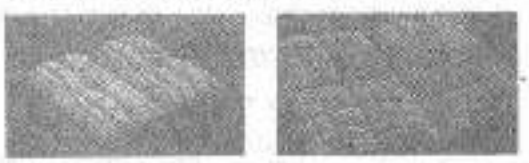


图9-12 牛奶冻表面（左）和迭代更多次得到的牛奶冻表面（右）

347

类似牛奶冻函数的曲面也可以用三角形来代替四边形。一个四边形用4个三角形表示，并将四边形中心与四边形高度成比例地上移。这样，中心点和四边形四个顶点组成了4个三角扇形，称为Takagi分形曲面。该类曲面在参考文献[PIE]中有更详细的描述。一种更通用的方法是用随机点创建各种拓扑结构的曲面。第14章将对分形做更详细的讨论。

9.7 标量场

标量场是定义在值域上的一元实函数，具有普遍性。如果值域是在一维实数空间上的某个区域，则标量场可以用通常的一元函数定义。二维标量场定义在二维值域上，是一个二元实数函数。二维标量场比先前讨论的封闭函数与迭代函数更为普遍。许多方法可以生成标量场，可以生成不连续的曲面，比如用扫描或者雷达测量所产生的数据生成的函数，就是标量场。它们给出的是一个非封闭的函数（就是不能用等式表示的函数）。由于二维标量场或二元实数函数通常以体数据的形式表示，因此我们将在体绘制章节中进行讨论。

数字高度图（DEM）是二维标量场的一个重要应用。数字高度图是二维灰度图（可以从USGS或者另外的资源，比如geogdata.csun.edu上获取）。在图像空间，每点的灰度值表示该点的高度值。事实上，很多彩色图像把高度用伪颜色表示。如果已知地图上最低和最高点的高度值，某点的高度值只要按照像素值就可确定。这样就形成了一个曲线网络来表征该区域的拓扑结果。另外，航拍图、雷达扫描图等也是二维标量场的应用（Google Earth是获取地理图像的理想工具）。图9-13左图给出一幅实际的数字高度图（第6章中描述的灰度高度图），右图是一幅航拍图片（圣地亚哥大学校园的一部分）。我们可以看到图9-13右图左下角显示的校园对应于图9-14的右下角。

348



图9-13 USGS数字高度图中的高度场（左）和航拍图像中的部分纹理图



图9-14 把高度场和纹理应用到地形可视化中

将高度数据点连接成多边形，并将纹理映射到多边形后，数字高度图具有一定的真实感。图9-14是圣地亚哥地区东部中心的纹理数字高度图。圣地亚哥州立大学校园可在图中的右下角附近看到。由于拍摄照片时的角度与我们观察纹理数字高度图时的角度不一样，所以只看到部分建筑。

通过采样点也可以得到其他变量的标量场。比如激光测距扫描得到某采样点的距离标量场。在地形图中，采样点转化成扫描空间中多边形上的顶点。作为该方法的一个例子，对Cap Blanc的Paleolithic洞穴艺术的研究就是同时采用对洞穴照相，如图9-15所示，和激光测距技术进行的。

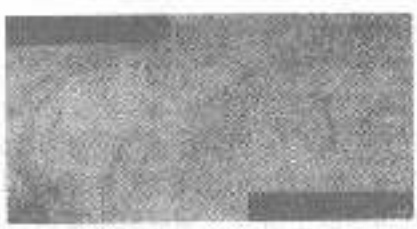


图9-15 洞穴处的照片（包含测量参考点和激光扫描点），参见彩图

洞穴的几何形状通过在洞穴里用激光扫描深度信息得到。扫描结果得到顶点与相对顶点几百个点的水平和垂直偏移距离的采样点。如图9-16所示，通过激光扫描（左）得到的偏移量与距离，可以计算洞穴的深度值。更进一步，可以生成网格（右），网格上每个点的位置和照片上每个点对应，经过纹理映射后，研究者们得到了一个几何和颜色都比较精确的洞穴墙壁模型。

349

现在，研究者们就可以利用洞穴几何模型来研究洞穴本身。比如，为获取洞穴在不同光照下的动画效果，可以改变绘制洞穴几何模型时的光照条件实现。在图9-17中，可以看到形

状像马的物体在一盏灯下面。首先，把灯显示在空间的底部中间。当这盏灯从左边向右边移动过程中，马也好象在运动。该过程演示了原始洞穴人在洞穴墙壁上雕刻的一系列马在活动的效果。

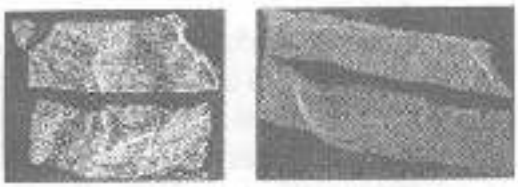


图9-16 原始激光扫描数据（左）和生成的响应几何网格（右）

人工合成曲面也是标量场的一个例子，其中的一个技术叫伪分形技术（fractal forgeries）。该技术是创建与分形有关的相同属性：它们是自相似的，从小区域相似开始直到大区域仍相似。伪分形场景基于多边形的类型，图9-18给出了这样一个例子。

350

351

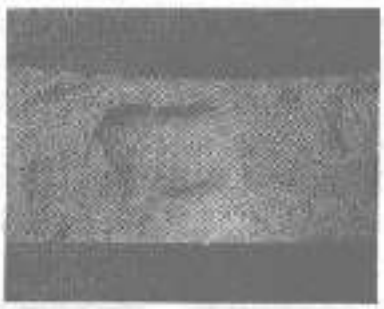


图9-17 洞穴墙壁上的画（模拟马和油灯）

实现这样一个场景比较简单，但要注意细节。曲面函数不是解析表达的，它所确定的曲面是在程序运行时对函数进行分段采样得到。由于该过程存在一定的随机性，所以每次绘制的结果都不一样。

从函数定义域中的某个网格出发，对整个网格进行细分。网格的数目必须是2^n+1。指数N必须足够大，以保证细分完成。但N太大会导致计算量过大。图9-18是一个257×257的网格，网格点索引值从0到256，每个网格点映射到定义域空间的某个值(x, y)。



图9-18 “伪分形法”生成的场景（有着色处理效果和透明的水），参见彩图

我们可以初始化一小组网格点值，它们的下标都是2的指数，如0，128，256等共9个网格点。为使初始值可以控制场景形状的值，我们选取每个点的初始值z。通过对该值取2的递增指数进行迭代，并让网格点周围的点的深度值取平均深度值z，而且网格点到周围点的偏移距离用随机数表示。如此迭代，直到定义域中所有的网格点都定义了z值。

我们对网格上的四边形用一对三角形来表示，并生成曲面。为实现光照，对每个网格点计算它的法向量。此外，还要给曲面上的每个三角形进行着色，并根据高度图，进行纹理映射。在图9-18中，对每个三角形给定一个随机高度值，进行着色后，得到一个弯曲的树线和雪线。我们也可以增加其他特征，比如水面，使场景更加有趣。为此，定义水面高度，并将它与曲面上所有三角形的高度进行比较。如果三角形高度比水面低，将该三角形表示成水，用蓝色表示，并设定它的α值以表示水的透明度。如果两者相同，将它表示成水面，用蓝色表示。绘制整个场景时，必须先绘制陆地然后绘制水面和水，我们就得到图9-18。

9.8 物体和行为仿真

将几何模型和其行为与真实物体进行比较，能帮助我们理解物体和现象，这个比较的过程称为仿真或可视化。

该问题涉及面广，我们不具体分析细节，只给出两个理想气体运动的例子来说明如何可视化分子运动，以及如何根据仿真数据来测试气体的预期运动。仿真结果也可以得到数值数

据与可视数据，这些帮助观察者对仿真结果的正确性进行评价。本节也对科学计算工具的仿真进行讨论，它有助于帮助我们理解各个操作，并解释仿真结果。

9.8.1 气体定律和扩散原理

理想气体在不同条件下的运动是化学与物理学课程的主要部分。在第一个仿真例子中，将多个分子（用点表示）封闭在空间中。每个分子的运动以随机的方向移动（随机产生向量的坐标，并通过单位化使它成为方向），并在该方向移动一定距离。该例子是气体在恒温下的运动仿真。仿真对分子和墙壁之间的碰撞进行检测，当碰撞发生时，分子按反方向运动，以模拟分子和墙壁之间的弹性关系。但是，不对分子之间的碰撞进行检测。空间分子运动结果图和仿真数据如图9-19所示。仿真时，显示每次分子与墙壁发生碰撞的压力与封闭空间大小的乘积。根据气体定律$pv = nRT$可知，理想气体的压强和体积的乘积是温度的常数倍，它等于另一常量乘以温度，而该常量是气体的摩尔数n和通用气体常数R的乘积。这里有统计的成分在里面。气体定律表明如果数量很大，那么随机性几乎消失。但在我们的简单模拟当中，随机性却很重要。在本例中，仿真结果也给出了单个分子的运动路径。 352

仅给出上述仿真结果，不能有效模拟气体运动定律，我们通过改变体积进行多次试验。正常情况下，每气体膨胀结束后，压强和体积的乘积是常量。为了进行统计，我们记录不同体积下的多个压强样本数据，并根据分析得出体积与压强的乘积是否是常量的结论。图9-19给出了程序在几个阶段跟踪某个分子路径的结果，它表现了分子随机运动的特性，不同试验的数据结果也在屏幕右边给出。该仿真程序的代码可以参考本书附带资源。读者可以改变代码，将试验结果输出到文件，并用于各种分析。 353

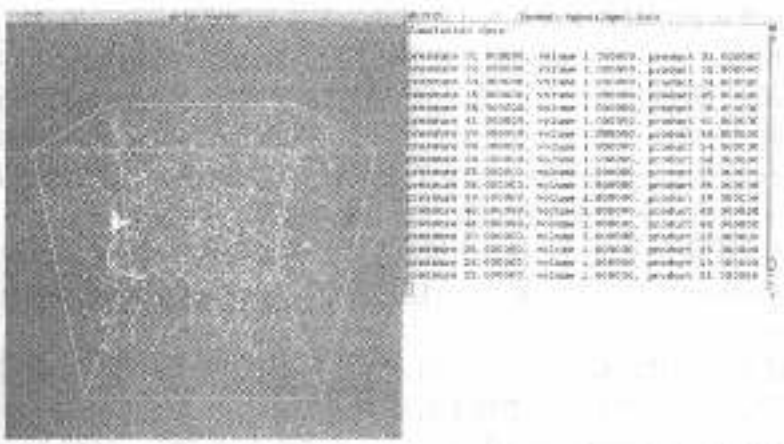


图9-19 把气体显示为固定空间的分子，包括仿真结果（左）和数值打印结果（右）。参见彩图

第二个仿真程序建立在第一个基础上。先将盒子用半透明隔膜隔开，如图9-20所示。分子在不同方向上与隔膜碰撞后，分子通过隔膜的概率也不一样，在气体多的一侧气体通过隔膜比较容易。而且不同类型的分子通过隔膜的概率也不一样。这导致隔膜将两种不同类型的分子分开，仿真程序用粒子系统模拟气体中的分子，用第一个仿真程序中的方法处理气体和墙壁、气体和隔膜的碰撞。仿真程序跟踪每种类型中的一个分子的运动路径。如果分子和隔膜发生碰撞则用随机函数确定分子是通过隔膜还是弹回。程序将所有分子的状态列成一张表显示在屏幕右端，程序同时计算隔膜两边分子的个数以及它们的比值并显示。初始仿真时所有分子在隔膜的一端，仿真过程中我们可以看到稳定状态下两边不同类型分子的比例。图9-20

给出了该程序运行时的一幅截图。该程序代码也包含在本书附带的资料中。



图9-20 直接透过薄膜的扩散仿真的例子（包括模拟中输出的数据），参见彩图

9.8.2 分子显示

有时，为了更好地理解一个问题，希望看到一些不可见的物体或者结构，但却缺少这样的函数或者过程来处理。这在化学领域中是由来已久的问题，在这个领域中，区分和显示分子结构的能力是化学中非常重要的组成部分。分子可视化推动了药品设计及其他方面的巨大
354 进步。然而，分子可视化是个相当大的课题，很多可视化技术非常复杂，需要深入理解分子层次的物理学。我们不希望在计算机图形学课程的开始进行这么复杂的工作，但可以演示一下这个过程的开始。

理解分子结构的传统方法（至少在作者的学生时代）是用分子弹簧球进行演示。在这个演示当中，每个原子用一个小球表示，小球的颜色表示原子的种类，每根骨骼用连接两个小球的弹簧表示。通过这个方法可以组装一些简单的分子，通过移动这个拼装的结构对它进行操作，可以从不同的角度观看它。我们可以非常容易地进行这些操作，并且可以进行一些扩展。

开始前，首先需要了解分子的基本几何形状。在很大程度上，分子的几何形状已经确定并且对大众公开了。其中一个主要的信息来源是蛋白质数据库，可以通过http://www.rcsb.org进行访问（它有一些镜像站点）。另一个是MDL信息系统（http://www.mdli.com）。分子描述信息以标准的格式存储在这些（甚至更多）站点中，可以对这些信息源进行访问，查找并下载想要检查的分子描述，用它们建立起分子的显示。这些描述通常是下列两种主要格式之一：.pdb（蛋白质数据基）格式或者.mol（CT文件）格式。本书的附录详述了这些格式的细节，本书的源代码资源里包含了一些从中读取基本几何形状的简单函数。查看Bristol大学的"Molecule of the Month"列表，可得到一些有趣的例子：http://www.bris.ac.uk/Depts/Chemistry/MOTM/motm.htm。

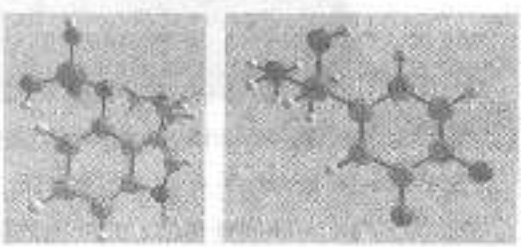


图9-21 显示psilocybin.mol（左）和adrenaline.pdb（右），参见彩图

从分子描述来显示非常直观，一旦对描述文件进行了解码，就可以得到分子中原子名称和位置的列表，还包括分子中的原子骨架。然后在指定的位置绘制原子并且绘制表示骨骼的连接。通常的做法是把原子画成球体，用不同的颜色和尺寸表示原子的类型；球体可以绘制成透明或部分透明，颜色和尺寸可以在文件读
355 取函数中提供。连接通常画成线段。图9-21显示两个简单分子的例子，它们来自与本书附带资源一起提供的分子阅读器和本书作者写的显示函数。原子的颜色包括在源代码资源中，可以对它们进行改变。从图中可以注意到原子是用相当小的透明度值绘制的，这样可以看见骨架和其他原子。同样注意来自.mol文件的例子，它显示已知的双骨骼结构（这些包括在.mol文

件中，但不包括在.pdb中）。正如在第7章中描述的，很容易和这些显示图进行各种交互。

在更进一步的工作中，使用包括额外信息的显示是很普遍的。例如，把分子显示成包围球体的平滑表面，用不同的着色表示表面上不同位置的静电力。这种显示通过显示表面形状和引导对接过程的力有助于演示分子对接是如何进行的。

9.8.3 科学仪器

除了可视化科学过程和结构，也可以设想可视化科学仪器是如何工作的。其中之一是气相色谱仪。当物质蒸发后，蒸发物经过试管的时候，可以用来衡量物质中不同种类的分子数目。当分子经过试管的时候，质量大的分子移动的速度比小的慢。当分子经过试管末端的探测器的时候，分子的数目用折线图记录下来。这个操作过程对不同的物质产生不同的轮廓，所以可以用来区分物质的组成。

为了理解气相色谱是如何工作的，一个叫Mike Dibley的学生写了一个模拟程序，演示一组分子从试管左端出发。这些分子有三种类型，用不同颜色的单个点表示。它们同时从试管左端出发，以不同的（带有细微的随机化）速度沿着试管移动，这依赖于分子的质量。当它们经过试管右端的时候，进行计数。任何时候到达末端的粒子数目被记录在折线图上。这个折线图显示在屏幕的底部，这样，这个演示就同时包含了三维元素的粒子试管和二维折线图。图9-22显示了这个模拟的三个阶段。在这个图中，左边的图像显示了第一组粒子（红色）开始离开试管时候的模拟，中间的图像显示的是第二组粒子（绿色）开始离开的时候，第三幅是最后一组粒子（黑色）开始离开的时候。从图中可以注意到慢的粒子有更多的时间对它们的运动进行随机处理，所以，到达试管末端的时候更分散，反映在折线图上的就是曲线更宽更低。



图9-22 气相色谱仿真的三个阶段

356

9.8.4 蒙特卡罗建模过程

至此，我们已经看过了一些基于分子随机运动的模拟，但在科学领域中还有更多的模拟例子。很多非常有趣的模拟是先通过产生大量随机事件来建立，然后观察系统在长时间中的行为。因为随机数在很长时间里和赌博联系在一起，并且蒙特卡罗（Monte Carlo）的casino（一种由二至四人玩的纸牌游戏）世界闻名，这种过程通常叫做蒙特卡罗过程。蒙特卡罗过程可以很简单，也可以非常复杂。我们已经看过了气体定律以及扩散模拟，都是这方面的例子，但还有很多其他蒙特卡罗模拟。

我们来看一个非常简单的例子：体积估计。考虑一个二维的区域，由非常复杂的曲线包围而成，因此很难衡量这个区域的面积。假如可以很快确定一个点是否落在区域内，通过包含该区域的面积中（通常是矩形）产生大量的随机点，那么蒙特卡罗估计就可以用来计算区域的面积。先统计产生点的总数，然后统计落在区域内的点数目并计算该比率，这个比率乘以已知区域的面积就是对该区域的面积估计。在三维空间中有非常相似的过程可以用来估计体积，如图9-23所示。在立方体中有10 000个点，这个立方体的每边是两个单位长，包含随机分布的（相交）球体。估计球体的体积。在图中，球体用较小的混

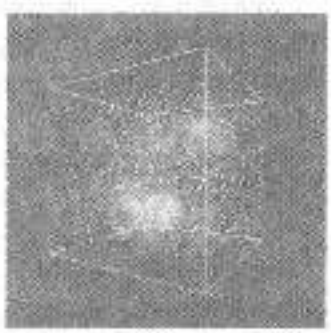


图9-23 复杂体的蒙特卡罗估计。彩见彩图

357

合值以黄颜色渲染，球体内部的点着上红色，外部的着上绿色。这可以得到点的相对比率的大致轮廓。蒙特卡罗过程并不能得到精确的解，但它给予了一个适当的估计，这对于向外行人（可能是法官或者陪审团）解释这个内容和过程是很有帮助的。通过程序展示（这个程序已经包含在本书的资源里），允许用户在任意方向上进行旋转来观看高亮的点在体积里是如何分布的。

在其他领域中可以找到更加复杂有趣的蒙特卡罗模拟。例如，在排队论和运输工程中，到达时间用一定的参数（标准差和概率分布）随机定义，服务和通过时间也是随机定义的，要研究的目标系统就是通过这些事件驱动。系统的属性就可以通过它对这些模拟的反应进行了解。系统的可视化可以帮助用户了解系统随着时间的变化最终是否达到平衡。例如，在交通模拟中，可以用各种颜色显示运输系统中的交通流，用高度关注的颜色来显示问题区域，用较低关注的颜色显示流畅的交通流。如果分别用红色和绿色表示，将是交通学习很好的途径。这些模拟也许超过了这个讨论的范围，但它们都可以用蒙特卡罗方法进行模拟。当然，统计模拟也可以做得更加复杂，因此，战争游戏、商业模拟和大规模的经济模拟都与蒙特卡罗模型有关。

9.9 四维作图

维度是计算机图形学中一个非常有趣的问题。我们现在说图形是三维的，因为我们生存的世界是三维的，所以对此感觉非常好。当然，我在讨论的是关于计算机图形API和标准的表示。但图形和可视化实际上并不仅是三维的，还有更多通常不需要考虑的维度。通常认为驾驶是二维过程，因为道路是以二维形式的，但在驾驶的过程中，还有速度向量、燃料等级、温度以及其他除了位置外的属性，这些在驾驶的过程中是必须不断平衡的。当开始运用计算机图形学来解决问题，就会发现用三维来定义和观察物体极大地限制了表达能力。正如刚才讨论的，在讨论建模和视觉交流中更高的维度时，必须为很多问题建立多于三维的模型。下面讨论这些技术的例子。

9.9.1 体数据

体数据是个一维数据，对应体积中每个点的实数值。可以对二维标量场符号进行扩展，把它作为三维空间中的标量场。把体数据看作来自有三个变量的实数值函数。因为实际上不能看见四维的空间，所以，将采取两种方法来显示这个场：在体积中寻找隐式曲面，或者把这些值显示在场的截面上。

我们考虑的隐式曲面是由点组成的表面，这些点的函数是常量值，这些也称为体积中函数的等值面。要找到它们是非常困难的，因为体数据是无结构的，要辨别出数据的结构来显示这些表面。有几种方法可以实现这个目的，最通用的是步进立方体处理。通过这个方法，
358 体积被划分成一些小的立方体，对每个立方体进行分析，看看表面是否穿过立方体。如果穿过，就对它进行进一步的分析，看看这个体穿过哪些边。这使得我们可以看到立方体上的表面是什么结构，接着对立方体中的表面进行显示。参看[WAZ]可以得到关于这个过程的更多细节。它所需的编程细节已经超出了我们讨论的范围，但图形本身只是直观地绘制多边形。

作为步进立方体的替代，我们将简单地区分这些立方体，它只包含一个点，这个点的值定义了表面，然后用包含表面的方式显示这些立方体。最简单的显示方式是在立方体中放置一个点亮的小球，如图9-24左图所示。这些适当数量的点亮球体的形状显示了表面的大概轮廓（虽然细节很少）。然后开始分析立方体中的场标量。如果立方体的数目（也是球的数目）

增加，显示将会变得更平滑，但这将降低计算速度，所以需要在速度和平滑度之间做个平衡。通过使用交互技术来改变定义表面的值，这种显示可以导致对空间的探查。通过清除覆盖体的一个区间值，可以看到标量场的所有形状，可以更好地理解产生场的问题。

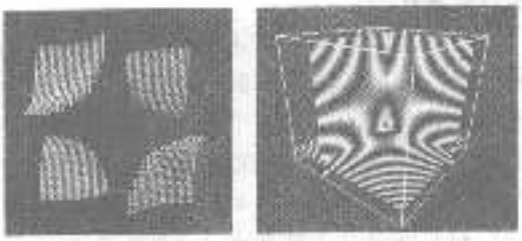


图9-24 使用球（左）和函数值的截面（右）来定位表面的隐式曲面近似。参见彩图

另一个理解空间中标量场的属性的方法是把空间进行切割，并把标量场看成是在切割平面或者横截面上的二维显示，如图9-24右图所示。这使得我们可以把这个函数看成是二维标量场的集合，可以在切割平面上利用网格和伪色彩来观看这个场。针对这个问题，这是个比隐式曲面更加精确的方法，与形状级别的值对比，同样可以理解整个空间值的关系，所以它很好地补充了隐式曲面过程。同样，可以对这个显示进行交互，让用户可以轻松地探测整个空间，从而对标量场的自然属性的整体概念有更好的理解。

359

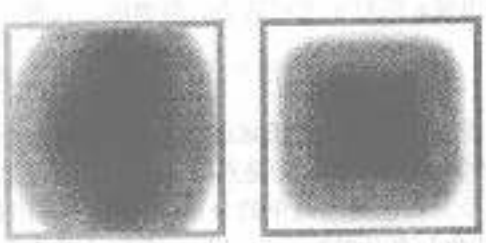
图9-24的两幅图像都来自交互式程序，允许用户通过增加或减少定义隐式曲面的值来清除空间，或者通过移动和坐标轴平行的三个切割平面来观看标量场的整个轮廓。这些交互探测对完全理解这些数据非常关键，并且非常容易实现。示例代码已经包含在本书的资源中，参看第7章的交互来得到更多的细节和例子。

图中的两幅图像实际上表示同一个函数：二元双曲线函数$f(x, y, z) = xyz$。隐式曲面描述的是子空间几何，$f(x, y, z)$函数值是常量，它是三维空间中八个象限中四个象限中的双曲线集合。这四个象限是变量的符号适合常量符号的象限，并且每个象限中的形状由常量决定。另一方面，利用横截面，找到一种方式使用颜色渐变来表示二维空间中的值，这个二维空间是切割三维空间中立方体的平面。我们已经使用了快速重复的颜色渐变来表示标量场的轮廓，希望能够从图中读懂实际值，那么在立方体中使用单一的颜色渐变可能更好。第5章中有关于这方面内容的详细描述。

9.9.2 向量场

我们可以把标量场从函数符号扩展到二维或三维空间中的向量值函数。这些函数称为向量场。它们在很多情况下出现，并不是所有的向量场都必须以向量表示的。任何函数的范围落在二维或三维空间内都可以把它的值看成向量，甚至可能不是最初定义函数的目的。例如，一个复变量的复函数有两维的定义域和两维的值域，它以复数变量为参数。因为复数是用实数对来表示的。我们来看一些函数产生向量场的例子。

前面已经提到过了复变量的复函数。下面考虑一下在[BRA]场景中描述的函数$w(z) = z^2 + 12z + 2$。通过实函数类推，可以推断它是个二次函数，对于等式，它应该有二个“根”——二个点使得$w(z) = 0$。在图9-25的左图中，利用另一种复数的表示方法$re^{i\theta}$，把r的大小显示成颜色（在第5章定义的统一亮度颜色渐变），把方向θ显示成方向向量来显示等式的图。图像中三个最黑的点对应所希望的三个根，可以看到整个空间中的向量场是平滑的。这表明复函数在



360

图9-25 对同一主题的两种可视化：基于一维复变量的复函数（左）与微分方程的方向向量（右），参见彩图

整个定义域是平滑的（并且是无限可微的）。通过这个显示，可以更好地理解函数和它的行为的自然属性。

另一类问题产生描述整个定义域内值的变化的微分方程组，可以使用同样的显示方式。例如，考虑液体流过平面，液体在平面上的每个点都有速度，平面是两维区域，每个点上的速度也是两维的。点是个位置，液体在那个点上的流动可以看作是位置的二维导数。求解这个微分方程就是为了确定描述液体流动的函数。在图9-25的右图中，可以看到一对微分方程 $\partial x/\partial y = y^2 - 1$ 和 $\partial y/\partial x = x^2 - 1$[BUC]对这种情况进行了描述，图像显示了流体的自然属性：在(±1, ±1)四个点上低速率（速度的大小），在其中符号相异的两个点产生涡流，在其中符号相同的两个点产生源/下沉流，如上所述，我们用颜色值表示流的速度，用向量表示方向。

9.10 高维作图

我们可以更进一步考虑三维空间的函数，它的函数值同样在三维空间里，这需要我们更好地去想象，因为实际上它是工作在六维的空间里，但这种问题在周围随处可见。标准的例子包括电磁和静电力、重力场以及液体流函数，尤其是在考虑液体流的时候把空气也像常规液体一样考虑在内。在这个问题中，因为考虑图形化描述的难度，所以选取相对简单的例子，避免一些非常繁琐的计算。不使用液体流的例子，作为替代，考虑使用电磁和静电力。

考虑电流在导线中流动时产生的电磁场。这些场在空间里每个点上都有向量值，可以用从三维空间到三维空间的函数表示，需要用六个维度来完整地表示这个函数。这个函数可以通过在规则的网格定义域上取样，并在每个点上显示场向量，如图9-26所示。这个例子选自Jordan Maynard教授给学生布置的项目，代码包含在本书的附加材料。这种向量场经常用于很多高维度的问题，但对于新手来说这比较难于理解。



图9-26 通电导线周围的磁场。参见彩图

另一个考察三维空间的值的方法是用一组向量来看函数对定义域中选择的一组点的结果。此前已经使用向量来可视化二维空间上的二维问题，所以读者应该熟悉这种技术。对于三维问题，要重新回到此前在本章以标量形式表示的静电力的库仑定律。这个定律表明两个粒子之间的静电力由 $F = kQq/r^2$ 给定，q 和 Q 是两个粒子上的电量，r 是两个粒子间的距离，k 是个适当的常量。这个力的方向沿着一个粒子指向另一个粒子。上面
361 计算的值是向量的长度。

如果在三维空间中看这个问题，首先从空间中任一点带有单位正电量 Q 的粒子开始，然后计算该粒子和空间中一组分别带有电量 q_i 的粒子之间的力，得到的公式是：

$$F(x, y, z) = \sum_i kq_iV_i / r_i(x, y, z)^2$$

对于三维空间中的任意点的力向量 $F(x, y, z)$，V_i 是任意点到 i 粒子的向量，$r_i(x, y, z)$ 是点 (x, y, z) 和 i 粒子的距离。图9-27显示了这个力在三维空间中的分布，它包括了一组带有+2（绿色）和−1（红色）电量的粒子，以及一系列由青色到橘红的向量。除了静态力向量，图中还显示了带有+1电量的粒子以0速率释放在空间中的轨迹。可以通过这种方式来跟踪一个或多个粒子，对于一组最初十分靠近的点进行跟踪

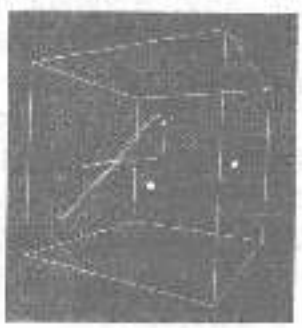


图9-27 以三维向量场模拟三维库仑定律，参见彩图

很有帮助，可以看到整个区域，而不是单个点的力的形状。

在高维的例子中，可以选择很多不同的显示种类。例如，上面看到在二维定义域上显示二维向量场的例子，它把大小和方向进行了分离。同样可以通过使用三维的定义域切割，在每个切面上显示每个向量的三维方向和一维大小，在三维定义域上查看三维向量。在这里读者有很多表现创新能力的机会。

统计是一个具有非常久远历史的多维数据场的例子。它很少有少于三维的变量，所以，要添加额外的信息到三维分布图和其他尝试看到更多信息的显示，允许用户通过使用选择变量或者组合变量来探测数据。这种显示的一个关键特征是观看者能够在空间里到处移动来观看分布图，从不同的角度来观看任何结构。强调交互式的图形有助于找到一种方式让观看者选择并操作数据和空间。 362

9.11 数据驱动图形

当基于数据而不是基于理论来讨论图形问题时，有一些新的问题需要考虑。数据不总是精确的，也不总是能够在一个数据集合中直接进行比较的（例如数据是在一天中不同的时间或不同的条件下获得的），也可能不是在它容易绘制的时候收集的，也有可能不能精确准确要表达的东西。对可视化而言，当创建图像去表达数据时，需要处理这些数据。然而，可视化的一个目的是理解有取舍的数据，并指出这些数据可能出错的地方。

在数据驱动显示中的数据可视表达不必太复杂。对于有限维的数据，数据分布图可以很好地可视化数据。在一个、两个或三个数据的数值变量确定的空间内，通过简单地画出数据点来表示几何对象，就可以有效地揭示模式的特征，而它的大小、形状、颜色和纹理则可以由数据的其他变量来确定。我们必须认识到：使数据的位置和表示能够向观察者传递数据的意义是很重要的。正如所讨论的，当考虑不同类型的数据表达时，一般数据可以用形状和纹理表示，有序数据可以用离散的颜色表示，间隔数据可以用位置和连续的颜色表示。 363

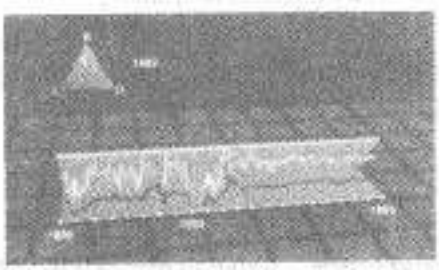


图9-28 经济数据展示，参见彩图

直线图是非常经典的画图方法，为大家所熟悉。但它通常以两维的形式出现，绘制时用一个独立变量和一个因变量。使用计算机图形学作为工具，这种图能表示更多的信息。图9-28画出的随时间变化（标有年份的红色轴）的经济状态（移动的线条）展示了这种图的效果。这幅图的作者Bernard Pailthorpe [PAI]这样描述：给定年份的经济状态根据它与三个特征（原型）的相似性画出来。这三个特征是：繁荣（绿色）；物价上涨（蓝色）和萧条（红色）。颜色帮助显示相对于三个轴的位置。阴影线表达经济相对于萧条和物价上涨两个轴的位置。注意在图的左上角的三角形，它表示每年的精确数据，作为颜色意义的图例。

如果已知数据来自两个独立变量和一个因变量，可以将它表示为曲面的形式。如果我们不知道数据是否来自一个连续的过程，只能用数据分布图来暗示一个曲面。如果数据是连续的，那么可以创建一个更加传统的曲面来表达数据。在两维空间中可能没有独立变量的规则分布，所以需要把定义域细分成三角形，创建Flat着色处理多边形曲面来表现真实的数据分布。另一方面，我们有理由知道数据可能来自对一个已知类别曲面的采样。所以，用数据去定义真实曲面的参数（如，使用最小二乘参数拟合），然后以点的方式用数据画出计算的曲面。这里又有机会让读者创造性地思考如何把这些信息传递给观看者。

正如在讨论可视化交流的时候注意到的，图像的真实性是非常重要的。创建包含或者符
364 合数据特征的平滑表面是非常简单的，也能够说明实际的数据是平滑变化的，但实际上，有时数据会包含跳跃或者其他不连续的人为因素，产生平滑的表面就意味着值是连续的，这不真实。简单性和真实性比用错误定义的图像吸引眼球更重要。

9.12 代码实例

在对建模和可视化技术进行讨论后，再来讨论编码实现的问题，因为对实现的讨论以及对本章中图表和例子的模型实现的示例代码的展示是很有帮助的。这些讨论和代码不仅关注图形本身，也关注建模和对建模过程的支持。当我们集中讨论图形相关的操作时，会使用一些通用的 API 函数，这些函数可以很容易在OpenGL中找到。然而，有时候，我们会使用一些严格的OpenGL函数来说明我们要解决的问题。

9.12.1 扩散

我们的扩散模型是本书开始时介绍的热传播模型。传播过程发生在一个网格上，所以第一个任务是定义一个二维矩形网格。这个网格需要记录网格上变量值的实数数组镜像。我们可以通过下面方法来声明实现：

```
#define LENGTH 50
#define WIDTH 30
float grid[LENGTH][WIDTH]
```

一旦定义了变量和初始化的网格，就可以考虑定义这些数值在材料中传播的代码了。在编码时，我们意识到需要利用网格上的数值来计算网格上新的数值。如果用新计算出的数值代替原来的数值，可能会产生错误。所以，需要利用一个网格镜像来保留我们正在生成的数值，这些数值可以这样定义：

```
float mirror[LENGTH][WIDTH]
```

通常，这个过程就像是计算加权和。实际的热传播依赖于两个相邻单元的温度差，我们将根据单元格的热量来计算整个热量。两个单元格之间的传播过程需要考虑温度差。网格中给定单元格的热量的计算方法如下：

$$newKept = \sum_{adjCells} heatAvail(cell)*weight$$

*heatKept*是单元格中剩余的热量，*heatAvail*是单元格中可以与相邻单元格共享的有效（非剩余）热量，*weight*依赖于使用的传播模型。能量守恒要求单元格中的剩余热量与有效热量之
365 和等于原始热量，不需要保留的热量假设被周围的四个单元格平均共享。我们假设对任何材料，单元格会保留一定比例的热量。权值是传播出去的与周围单元共享的热量的比例。如果我们认为热量是被周围单元格均匀共享的话，那么可以是0.25。于是任意单元格中的热量方程应该是：单元格中的热量等于单元格中的剩余热量加上从相邻单元格中传来的热量。这个值被计算出来并保存在镜像数组中，在计算完成后，把数组复制回网格数组。代码如下：

```
#define KEPT prop // 每个单元中保留的属性比例
#define SHARED .25*(1.-KEPT) // 与其他单元共享的比例
for i
    for j {
        mirror[i][j]  = KEPT*grid[i][j];
        mirror[i][j] += SHARED*grid[i+1][j];
        mirror[i][j] += SHARED*grid[i][j+1];
        mirror[i][j] += SHARED*grid[i-1][j];
        mirror[i][j] += SHARED*grid[i][j-1];
    }
```

```
for i
    for j
        grid[i][j] = mirror[i][j]
```

一个更加灵活的方法是定义一个3×3的滤波器(也可以是不同的大小)，或一个和为1的非零数组，乘以[i][j]周围的单元格grid[i][j]。这是热传递例子的代码方法，可以利用一个小的双重循环来计算单元格中的热量，如下：

```
float filter[3][3] = {{.05,.1,.05},{.1,.4,.1},{.05,.1,.05}};
mirror[m][n] = 0.0;
 for i
     for j
         mirror[m][n] += filter[i][j]*grid[m+i-1][n+j-1];
```

最后一行坐标的变化是因为下标为i和j的滤波器的值从0变化到2，而网格下标必须从$m-1$变化到$m+1$，从$n-1$变化到$n+1$。这对于很多计算来说都是个非常经典的过程，可以作为平时的编程小技巧。

当然，目前没有考虑边界单元格的传播过程。在边界单元格中，必须记住没有热量从边界单元格之外传播进来（同样，也会有更少的热量流失）。这将作为练习留给读者。最后，保持恒温的单元在传播之后需要恢复原来固定的数值。

以上我们假设材料是同质的，如果不是，就需要考虑材料不同的热力学系数。如果两个相邻单元格是异质材料，必须使用一个合适的热传播系数。系数需要是对称的（一个单元格与相邻单元格的系数相等），我们建议使用最小的传播系数。这个变化会应用到同质和异质的材料中，但不会影响同质的情况。这需要另外一个数组记录剩余热量的比例（0~1之间），声明如下： 366

```
float prop[LENGTH][WIDTH]
```

prop中的数值在计算过程中不是固定不变的，我们将采用原始点与新点比例的最小值。于是，对于第一个相邻点，我们可以定义如下的共享热量：

```
.25*(1-min(prop[i][j],prop[i+1][j])*grid[i-1][j];
```

不管在什么情况下，必须保证整个模型的热平衡。

当完成仿真的每一个步骤，都需要显示一下计算结果。我们可以利用立方体作为单元格，这个立方体在绘制的时候可以缩放、平移和着色。如果我们有一个单位立方体定义在第一象限，一个顶点位于原点，我们可以构建第一象限的网格，从原点开始，网格上的单元格的边长为L：

```
for i
  for j
    set color to colorRamp(grid[i][j]);
    set translation(i*L, j*L);
    set scaling(L, L, L*grid[i][j]);
    draw cube;
```

如果定义了一个合理的视点，设置了诸如深度缓存之类的变量，画出这些单元格就不成问题。也可以通过加入交互来增强显示，使用户可以更方便地观察。在模拟运行的时候，可以利用idle事件来更新图像，这样就可以让用户实时地观察到热扩散时温度的变化。

9.12.2 函数作图

此前简要介绍了作图的过程。它是在定义域空间中建立一个均匀网格，像上面利用网格来模拟热传播的过程，计算网格上每一个点的值。我们可以利用网格坐标和组成曲面的四边形和三角形的函数值顶点。下面的代码将详细说明这个过程。这是非常基础的操作，所以必须详细了解它。代码假设所有的安装和声明都已完成，使用元API来绘制三角形。代码如下：

```
// 假设函数calcValue(x,y)为定义域中的每个点计算一个浮点数；同时进一步假设
//
// 每个方向上的点是一样多的，并且把计算结果得到的值保存在二维的实数组中。
//
// 假设函数calcXValue和calcYValue计算定义域中每个网格点的x-和y-值，需要
//
// 注意的是我们在每个方向上比所要创建的矩形区域数额外多一个。
for (i=0; i<=NPTS; i++)
    for (j=0; j<=NPTS; j++) {
        x = calcXValue(i); // calculate i-th point in x-direction
        y = calcYValue(j); // calculate j-th point in y-direction
        values[i][j] = calcValue(x,y);
    }
// 利用计算得到的值，通过为网格中把每个矩形分成两个三角形来创建网格曲面。
//
// 我们从//上至下逆时针方向计算。
for (i=0; i<NPTS; i++)
    for (j=0; j<NPTS; j++) {
    // 计算矩形每个角的x，y坐标
        x0 = calcXValue(i);
        x1 = calcXValue(i+1);
        y0 = calcYValue(j);
        y1 = calcYValue(j+1);
        // 画第一个三角形
        beginTriangle();
            // 计算三角形的属性，如法向，
            // 此处省略
            setPoint(x0,y0,values[i][j]);
            setPoint(x1,y0,values[i+1][j]);
            setPoint(x1,y1,values[i+1][j+1]);
        endTriangle();
        beginTriangle();
            // 计算三角形的属性
            setPoint(x0,y0,values[i][j]);
            setPoint(x1,y1,values[i+1][j+1]);
            setPoint(x0,y1,values[i][j+1]);
        endTriangle();
        }
```

9.12.3 参数曲线与曲面

对于参数曲线，很容易对它的定义域进行划分，每个部分是是实轴上的一段区间[a,b]。通过定义一些点来计算参数函数的值。如果用参数方程$ta + (1-t)b$来定义区间，定义域包含了所有在[0,1]之间的t。把这个区间分成N个相等的区间，对于在0和N之间的值，第i个点可以简化成$f(ta + (1-t)b)$，$t = i/N$，假设函数f生成点而不是一个值。这条曲线就可以通过简单地连接点之间的线段绘制出来。这个过程在二维和三维空间中是一样的。如果定义域的划分不相等，点的计算过程将会不同。但是绘制方法是一样的，都是通过连接每个点的函数值。

用代码来实现它非常直观，假定曲线上一个点的坐标x，y，z能用参数函数$fx(t)$，$fy(t)$，$fz(t)$确定：

```
#define START 0.0
#define END 1.0   // 起始，终止区间可以是任意值
beginLines();
  x = fx(START); y = fy(START); z = fz(START);
  setPoint(x,y);
  for i from 1 to N, inclusive
    t = START*(N-i)/N + END*i/N;
    x = fx(t); y = fy(t); z = fz(t);
    setPoint(x,y,z);
endLines();
```

对于参数曲面，处理起来相对要复杂些，但也不尽然。曲线的定义域为u，v空间里的矩形，这里$a = u = b$，$c = v = d$，让曲面上每个区间大小一样。每个变量的间隔可能不一样，就像前面讨论圆环形的二角形曲面一样。然后计算矩形里的点(u_i, v_j)，它表示在u方向的第i个点和v方向的第j个点，借助参数表达式，可以从这些点计算得到曲面坐标。计算定义域里的网格单元

的四个顶点，曲面上的点可以由它得到，然后建立四边形（或一对三角形）。可以借助图形API画出这些三角形或者四边形网格。除了曲面上每个点的三个坐标需要由函数决定，而不是由单个坐标决定外，实现这一功能的代码和上面曲面绘制三角形网格的十分相似。

9.12.4 极限处理

极限处理和计算是一对矛盾，因为计算能力总是有限的，而极限处理实质上无限的，如果不能在有效的时间内使极限计算收敛，可以考虑计算一个近似解，然后把这个近似解呈现给观察者。

对于极限曲线或曲面，可以简单地把步骤划分得尽可能多，因为越多的步骤就越精确。但是这也意味着需要更多的计算时间和存储空间。一旦完成了有限的计算，就得到一个函数值（或者对于一个参数提供的若干函数），我们可以在图像中把这个结果表示出来。

9.12.5 标量场

除了可能有其他方法在定义域里确定每个点的标量值之外，一维或二维标量场实质上与函数图或曲面是一样的。因此，显示一维或二维标量场实际上已经包含在上面的讨论里面，而三维标量场包含在下面的四维作剖。

9.12.6 物体及行为的表示

有时候，和前面讨论的气相色谱分析例子一样，计算机图形学用来显示模拟科学研究中的对象和它的行为。在本章给出的都是相当简单的图形例子，我们把重点放在细节的显示。例如，显示气体分子行为的模拟时，从体积中选择一小部分粒子用来显示。因为这样做可以避免混淆，可以跟踪模拟过程中单个粒子的轨迹。通过给每个粒子坐标加上一个小随机数，可以产生粒子随机的效果。通过跟踪记录典型粒子的运动轨迹模拟它的行为特点。

这两个模拟中最有意思的是粒子碰击区域边界的反应。在气体定律的例子中，让粒子反弹进入体内；在半渗透薄膜的例子中，我们对薄膜膜做了同样的实验，只是用了一个随机数来决定粒子是反弹还是渗透进薄膜。从这个意义上来讲，这个模拟有些蒙特卡罗的意思。

当一个粒子正常地随机离开体时，我们能侦测它的状态。在那个例子中，通过累记器增加记数的方式记录击中，这很容易做到。假定粒子遵循“反弹”规则，借助全反射来计算每个粒子反弹的位置。这个反射非常直观，曾经在第4章介绍过。在阅读下面这段代码前，也许应该回顾一下全反弹几何的相关内容。假定包围区域的是到原点的距离为常数的墙壁，且每个点的坐标都依次记录在数组p[i][j]里面：

```
typedef GLfloat point3 [3];
point3 p [NPTS];
if (p[i][j] > bound)
     {p[i][j] = 2.0*bound - p[i][j]; bounce++;}
if (p[i][j] <-bound)
     {p[i][j] = 2.0*(-bound) - p[i][j]; bounce++;}
```

画出每个点的运动轨迹非常直观：用一个数组记录粒子运动的最后N个位置，每次显示前把数组里记录的粒子位置向后移一位，把粒子新的位置加在数组前面，把数组里点的位置用线段连接起来，就可以显示粒子新的运动轨迹，这样有助于显示单个点的运动行为，也有利于观察者更好理解仿真显示。

最后，我们收集各种统计数据（每个空间有多少粒子，在最后的时间有多少粒子击中了体壁等），然后在图形系统显示这些统计数据，或者把这些统计数据打印到终端或文件，如果

369

用事件来触发它，通常用键盘敲击的方式实现。

9.12.7 分子显示

像前面注意到的，显示分子的数组是通过读.pdb或者.mol格式的文件得到的。这些数组有
370 如下形式：

```
typedef struct atomdata {
    float x, y, z;
    char name[5];
    int colindex;
    } atomdata;
atomdata atoms[AMAX];
    typedef struct bonddata {
    int first, second, bondtype;
    } bonddata;
bonddata bonds[BMAX];
```

这里程序可以通过在下面描述的查找表中查找得到原子结构中的colindex域，这个索引接着用于查找匹配显示的合适原子的颜色和尺寸。

正如声明的一样，这个函数可以读取文件，并把数组中这些结构的结果存回去。接着对数组进行遍历，可以从查找表中得到保存诸如每个原子的尺寸、颜色的额外数据信息。第一步是通过名称查找原子，并返回在表中原子的索引。信息保存在数组中后，可以在遍历数组的时候创建图像。用索引到的尺寸和颜色把分子绘制出来。查找表的部分示例如下：第一个表用来匹配名字，其他的用来得到和原子相关的颜色和尺寸。

```
char atomNames[ATSIZE][4] = { // 通过名字得到 x标
    {"H "}, // 氢
    {"He "}, // 氦
    {"Li "}, // 锂
    {"Be "}, // 铍
    {"B "}, // 硼
    {"C "}, // 碳
    {"N "}, // 氮
    {"O "}, // 氧
    ...
    };
float atomColors[ATSIZE][4] = { // 任意颜色
    {1.0, 1.0, 1.0, 0.8}, // 氢
    {1.0, 1.0, 1.0, 0.8}, // 氦
    {1.0, 1.0, 1.0, 0.8}, // 锂
    {1.0, 1.0, 1.0, 0.8}, // 铍
    {1.0, 1.0, 1.0, 0.8}, // 硼
    {0.0, 1.0, 0.0, 0.8}, // 碳
    {0.0, 0.0, 1.0, 0.8}, // 氮
    {1.0, 0.0, 0.0, 0.8}, // 氧
    ...
371 };
float atomSizes[ATSIZE] = { // angstroms中的大小
    {0.37}, // 氢
    {0.50}, // 氦
    {1.52}, // 锂
    {1.11}, // 铍
    {0.88}, // 硼
    {0.77}, // 碳
    {0.70}, // 氮
    {0.66}, // 氧
    ...
    };
```

一旦得到了每个原子的位置和通过它的名称得到属性，那么绘制原子将是件非常简单的事，骨骼可以简单地在两个原子间用直线来绘制。骨骼定义了两个原子的索引。如果有双骨骼结构，那么就画两根直线，每根都稍微偏离原子中心。因为观看者希望能够从不同的角度观看分子的结构，需要允许进行任意的旋转，让观看者可以选择原子的可选表现，例如增加

或减少透明度，增加原子大小（布满空间的表示），或者其他一些可以让用户通过控制面板、菜单或键盘来选择的选项，这也会是很有帮助的。

9.12.8 蒙特卡罗建模

本节使用蒙特卡罗这个术语来泛指基于随机值的过程。在这种意义下，气体定律和半渗透薄膜模拟都是蒙特卡罗模型，这在此前已经注意到了。然而，有时候蒙特卡罗模拟用于表示事件是直接通过随机数设置的，体积估计就是这方面的一个例子，它的事件就是放置单个点。当计算点（p.x, p.y, p.z）是否落在点（sphere.x, sphere.y, sphere.z）的半径sphere.r内时会产生大量随机放置的点，并对一个或多个球体统计落在半径内的数目并不是件难事。

其他的蒙特卡罗模型可能稍微复杂些。例如著名的实验是估计π的值，称为Bouffon针实验。它需要在纸上画一些距离和针的长度一样的平行线，然后丢大量的针到纸上。穿过直线的针的比例大概是2/π。用计算机图形学模拟这个过程非常直观。先产生一个随机的点作为针的一个端点，再产生一个随机的角度（0到2π之间），沿着这个角度在距离第一个点一个单位的地方放上第二个点，比较两个端点的值，看“针”是否穿过“直线”。当然，可以实时绘制这些针，使得观看者可以看到整个过程。

9.12.9 四维作图

三维标量场是非常难以显示物体的，因为它包含一个三维的定义域和一个一维的值域，所以，我们实际上是工作在四维的层次上。我们已经看到过有两种方法可以表示这种信息，当然，还有很多其他方法。这两种方法的代码相当简单易懂。对于等高面方法，我们把体积分 372
成若干立方体，然后在立方体的8个顶点处分别计算它们的标量场函数。如果函数经过定义等高面的固定值，我们就会看到有些顶点的值大于固定值，但有些会小于该值，所以这个立方体就包含了平面的一部分，我们就在这个位置画一个球来表示。通过一个简单的小技巧来确定函数是否经过一个固定值：把这个值和所有顶点处的函数值相减，然后把所有差相乘。如果乘积符号是负值，那么就经过了那个固定值。所以，代码包含了三个嵌套的循环，如果测试的是正数，就画一个球，具体如下：

```
for (i=0; i<XSIZE; i++)
    for (j=0; j<YSIZE; j++)
        for (k=0; k<ZSIZE; k++) {
            x = XX(i); x1 = XX(i+1);
            y = YY(j); y1 = YY(j+1);
            z = ZZ(k); z1 = ZZ(k+1);
            p1 = f(x, y, z);  p2 = f(x, y,z1);
            p3 = f(x1, y,z1); p4 = f(x1, y, z);
            p5 = f(x,y1, z);  p6 = f(x,y1,z1);
            p7 = f(x1,y1,z1); p8 = f(x1,y1, z);
        if (((p1-C)*(p2-C)<0.0)      || ((p2-C)*(p3-C)<0.0) ||
            ((p3-C)*(p4-C)<0.0)      || ((p1-C)*(p4-C)<0.0) ||
            ((p1-C)*(p5-C)<0.0)      || ((p2-C)*(p6-C)<0.0) ||
            ((p3-C)*(p7-C)<0.0)      || ((p4-C)*(p8-C)<0.0) ||
            ((p5-C)*(p6-C)<0.0)      || ((p6-C)*(p7-C)<0.0) ||
            ((p7-C)*(p8-C)<0.0)      || ((p5-C)*(p8-C)<0.0))  {
                drawSphere (x, y, z, rad);
    }
}
```

对于裁剪平面显示，简单地在空间里定义一个平面，把该平面当作函数的定义域，然后按照二维标量场的方法进行迭代。也就是说，用二维网格把二维网格放到两个剩余变量上，计算网格上每一个矩形的中点函数值，然后用函数值确定的颜色，在3D空间中画出矩形网格。由于这和二维标量场非常相似，我们就没有列出代码。使用交互技术来改变坐标轴穿过裁剪

平面或者改变定义裁剪的轴的值都是非常直观的。定义其他平面来切割空间也是有可能的，虽然可能稍微复杂些，我们并没有对此进行尝试。

当我们在一个二维定义域中考虑二维向量场时，同样有四个维度，还要选择怎样对显示进行组织。显然不能直接显示域中每个点处的二维向量，因为这将导致无法找到一个单独的向量。然而，如果知道向量场是相对平滑的，就可以在定义域中指定的点显示结果向量，给观察者一幅带有各种长度和方向的向量图像。这样，就可以让观察者理解结果，同时一体化可能带有重叠（也可能不重叠，取决于向量场）的向量图像。这个方法不错，但必须小心使用。然而，该方法确实也有内在的缺陷，如果选择的点之间有间隔，就会丢失向量场的一些
373 重要特征，比如一个突然的异常，就会导致图像的不准确。

我们选择了一个稍微有点区别的方法，对结果向量的大小与方向分别显示。大小仅是个标量场，我们已经看到过使用一些诸如伪彩色渐变技术来进行显示。因为单独显示了向量的大小，我们就可以用单位向量来显示向量的方向，这样就能显示出定义域中方向的分布情况。把这两部分集合到一块，就能使显示变得既简单又能提供丰富的信息。然而需要指出的是，用户可能无法立即理解显示所表达的含义，因为颜色和方向是不相关的概念。这样就需要一些相关的用户培训。

当标量场绘制完后，向量就绘制在定义域网格上10×10的块中每个中点。我们把向量颜色回成青色，因为这样就可以和由黑到黄的标量场颜色形成对比。下面的这段代码假设已经计算出定义域中每一个网格矩形边界的x，y值，并且计算出了矩形中点的向量值。这段代码简单地描述了如何创建向量部分的显示。

```
if ((i%10==5) && (j%10==5)) { // 每10个单元的中间
x = 0.5*(XX[i]+XX[i-1]);
y = 0.5*(YY[j]+YY[j-1]);
len = 5.0 * sqrt(vector[0]*vector[0]+vector[1]*vector[1]);
glBegin(GL_LINES);
   glColor4f(0.0, 1.0, 1.0, 1.0);
   glVertex3f(x,y,EPSILON); //因此向量在曲面之上
   glVertex3f(x+vector[0]/len, y+vector[1]/len, EPSILON);
glEnd();
}
```

9.12.10 高维作图

当涉及任何高维图形的时候，必须非常仔细地保持图像清晰以及针对性。如果尝试包含过多信息的话就很会变得难以理解，当其他维上存在更多还不能显示的信息时，就需要确定哪些数据要展示出来以及如何展示它们，或者确定如何将这种选择交由观察者决定。

当在超过三维的定义域中讨论向量场时，就会碰到上面提到的怎样表达大量信息，以及投影带来的信息隐藏的问题。让观察者任意移动数据（换个说法就是，让观察者随意移动数据体）非常重要，这样能全面观察数据。还有一点也很重要，就是有选择地显示信息，这样观察者就能够观察到相关数据，而不是一次看到所有的信息。与在三维网格空间中显示每个点的所有向量不同，可以采用和上面的向量显示非常类似的技术，只显示空间中相对较少的向量。通过规定方式把它们放置在空间中，我们可以显示向量场的轮廓而不是整个场。相应
374 的代码也比较简单，代码如下：

```
set the color for the vectors
for i
   for j
      for k {
         calculate coordinates of the i,j,k-th point
         calculate vector from magnetic field function for point
```

```
    begin lines
      set the point
      set the offset from that point by the vector
    end lines
)
```

我们不再追求高维作图中的更多选择，因为方法比较多。但经常关注科学研究中高维绘制比较好的例子。比如阅读一些原始资料，如《科学》或者《科技美国》这两本杂志，它们以可视化技术闻名，不过只有部分内容是计算机生成的。正如我们在前面对视觉交流的讨论中看到的，可以通过使用颜色、形状以及其他方法来表现高维信息。这些工作有的适合显示标称数（如形状），有的适合于表达有序数（如颜色），但是若要作更深的研究，就需要更多的创造力。

9.13 小结

在这一章，我们看到了用于建模和表达科学问题的一些技术，包括表面绘制和对力场中粒子运动轨迹的跟踪等。它们提供了一系列不错的例子，从而扩展计算机图形学解决科学问题和通用问题的方法。这本书的最终目的是帮助读者培养使用图形学技术来解决问题的能力。相信读者在学完本章后，可以部分实现这个目标。

9.14 思考题

1. 在对科学现象建模的相关诸多问题中，有很多针对连续函数的操作，如求导或积分。然而，很多情况下，我们无法找到简单的等式来表达这些操作，而必须使用这些操作的离散版本。除非能使用复杂的数值计算技术，我们必须使用简单的微分方程组对这些操作建模，但是这会带来误差。请描述一些微分方程组可能带来的误差，同时给出一些减少误差的方法。
2. 在第5章，我们重点讨论了许多有关创建和使用颜色渐变来给出数值数据的可视化表示。在用于科学数据作图的颜色渐变中是否存在一些额外的问题？是否存在一些用颜色表达的科学现象可以帮助选择颜色渐变？
3. 找一期《科学》或者《科技美国》仔细地阅读一遍，找出包含高质量图像的文章。从中挑出至少一篇，写一篇简短的论文，主要是关于为创建图像用到的建模、图形绘制以及视觉交流，然后用现在所学到的工具和图形API来创建其中一幅图像的近似版本。 375

9.15 练习题

1. 在一个科学问题中，找一个包含两个连续变量的函数例子，创建该函数的曲面表达式。
2. 根据一个马鞍表面，找到一个参数曲面表达式，然后为该曲面创建图像。
3. 找一个空间受力情况的例子，如重力或静电力。创建一个物体受到这些力时，在空间运动轨迹的表示。如果没有较好的积分器，可以在一个很小的时间段内使用简单的分段线性轨迹。
4. 找一个高度场的图像，从图像中根据灰度值翻译出所表示的高度，然后根据这些高度值创建表面。
5. 在热传导例子或疾病传播模型描述中的扩散模型在许多科学领域都很常见，例如，有一个针对某地区的人口增长模型，在这个地区的人通过高速公路向外迁移。找一个基于扩散的问题，对它进行建模，指出通过扩散增长的模型。
6. 对热扩散程序，把每个单元认为是个点，建立棒中温度表示问题的曲面。（可以为每个单元索引建立x和y值，令z为单元的温度，让顶点的颜色表示单元的颜色）。为三角形或四边形使用平滑着色处理。这样得到的曲面图像表示单元的温度的效果好了还是坏了？
7. 使用相对简单的标准分子描述文件（.pdb或.mol文件），以及本书资源中的一些函数，读入分子数据，

创建分子的表示，允许用户通过简单的控制来进行操作。

8. 从科学问题中找出一个包含3个连续自变量的函数实例，然后对函数创建体描述。
9. 按照下面的方式实现Bouffon针实验：首先生成一定数量的单位长的线段，把它们画在一个窗口中，该窗口中包含一系列平行线，两两相隔一个单位长。统计和直线相交的线段数目。同时计算它们在所有针中所占的比例。这个比率是不是对$2/\pi$很好的估计，如何解释这个结果？当产生更多的针时，是不是估计得更好？

9.16 实验题

1. 在热传播的例子中，在固定点上温度是常数。修改这个例子，使这些点的温度随时间变化，能否创建这样一个温度变化模型，使得热量以波期波的形式在空间中传播。
2. 修改本章的热传递模型，使得传播过程更加有趣。首先，让传播非对称，方法是改变传播过滤器，使热量直接传向邻近的方格的同时，也传向对角相邻的方格。然后改变传递过滤器，使得热量在某一个对角线上传播更容易（这可以给一些材料建模，如纤维，热量沿纤维传播时比在纤维之间传播更快）。
376 运行程序，看看在这种情况下的热传递情况。
3. 在（4,3）圆环的讨论中，我们建议用相似的方法创建其他一些有趣的表面。其中比较有趣的表面是默比乌斯带，这是一个矩形平面，它的首尾两端反向连接。使用（4,3）圆环的模板，创建一个默比乌斯带。
4. 在讨论体数据时，给出了一个隐式曲面和横截面来理解体信息的本质，能否找到另一个方法来表达这种信息，这个方法是不是能够加深对体的理解？
5. 在函数绘制这一节中，我们讨论一个处处平滑的特殊函数(拥有任意阶的连续导数)。画出拥有某种断点的不连续函数的图像，并且检查一下所产生的曲面，能不能找到一种方法可以处理这种使曲面看起来也不连续的断点？
6. 科学问题的一个很好的来源就是对物体作用的力。其建模方法如下：首先为物体选择一个初始位置和速度，然后使用经典方程$f = m^*a$来计算任一时刻的加速度，然后利用加速度更新速度，速度再更新位置。以这种方式处理n-$body$问题，即确定一个系统中n个物体的行为，物体两两之间都有万有引力的吸引，服从经典引力公式。看看得到的结果是否符合现实，或者因为是基于微分方程组而不是差分方程组让人难以接受。

9.17 大型作业

1. 模拟绝缘球体表面上移动带电粒子的行为，找出系统最低的能量状态。假设所有粒子有相同的电荷，这就使实际电荷大小与系统无关。可以使用你所喜欢的任何方法，但是下面的$O(N^2)$的算法可以作为着手点：
   a. 在单位球体表面上随机放置N个点。
   b. 使用半透明球或线框球在球体表面画出各点。如果点的数量很多，可以不画出球体。
   c. 对于每一个点，
   - 计算每一个点到其他各点的向量
   - 按照向量长度平方的倒数（库仑定律），为这些向量设置权值
   - 对加权向量求和，得到该点的力向量

   d. 将所有力向量除以长度最大的那个向量的大小，使系统归一化
   e. 取出每个点的力向量，减去它的径向分量（单位径向向量点乘力向量），得到与球面相切的力向量，通过一个很小的常数（可以尝试0.05）缩放这些向量，使得更加容易看到这些点的运动。
   f. 对于每一个点，把得到的向量加到点的位置上，然后用点到球心的距离除以新点的位置坐标，使得

点存回到球体表面。

重复b-f，直到所有点都收敛到期望的精度，在收敛处，所有的“力”都是径向朝外的，所以，这些点都不动。这个算法总会使点收敛到一个最优的解决方案上，并且独立于坐标系。所有的点绕球心同步的旋转都是正确的。 377

如果要对这个问题进行实验，允许点有可变的电势（但是符号相同），可以在c步的时候引入第二个点来实现它。

2. 使用“伪分形”方法来创建地形，利用已掌握的工具。例如，使用很小的扰动创建一个地形，然后用一张农场的航空帖图纹理映射到上面。或者采用剧烈的扰动，然后使用亚利桑那大峡谷的颜色为其建模。这个例子可以让读者很好地发挥自己的创造力。

3. （检查一下所选择的科学问题）找一个感兴趣的科学问题，并对该问题提出一些可以解释或理解的疑问。为该问题创建一个可验证的模型，详细地描述这个问题，并说明为什么模型对它是一个很好的表达，利用OpenGL或另一个图形API编程实现这个模型，创建一幅图像来传达问题的信息，详细说明为什么该图能够传达信息，为什么它对问题能提供一个很好的理解。 378

第10章　绘制与绘制流水线

在本书前几章介绍的几何和外观属性基础上，本章阐述如何实现基于多边形的图形系统的图像绘制方法。本书起始部分介绍了几何流水线，几何流水线产生的几何图元由绘制流水线作进一步处理，并生成图形程序所描述的最终图像。绘制流水线需要计算由几何流水线产生的每个顶点的外观属性，将几何对象分解为对应输出光栅扫描线的片段，还包括颜色、深度和纹理映射等几项计算，最后将片段信息写入颜色缓存之中。我们将介绍一些将线段离散化为每条扫描线上的点的机制，这些点构成片段的端点。还将介绍扫描线属性值的线性插值和透视修正两种插值方法。为了更好地理解本章讨论的内容，必须掌握图元和几何流水线，才能理解绘制流水线是如何处理图元的细节。本章不直接涉及图形编程技术，但是学习完本章后，应理解图形系统生成图像的处理过程，这有助于编制更有效率的图形程序。

10.1　引言

前面各章讨论了几何流水线，描述了图形API将几何对象从3D模型坐标系转换为2D屏幕坐标系的操作细节。有了这些2D屏幕坐标之后，仍然需要进行一系列操作，并将最终图像绘制显示在屏幕或者其他输出设备上。这些操作随使用的图形系统不同而不同，但一般说来，这些操作具有流水线结构，因为这条流水线将几何流水线的输出绘制生成图像，所以称为绘制流水线。

379

必须指出，我们描述的绘制流水线仅适用于基于多边形的图形系统。这些系统使用多边形描述场景，通过处理每一个多边形来绘制场景。并非每一个图形系统都采用这种方式。如光线跟踪系统会为显示系统的每个像素生成一条或一组射线，并计算出这些射线与场景中最近物体的交集，然后根据这些物体的光学特性来计算出像素点的外观属性，这些问题留待第14章介绍逐像素操作时再深入讨论。这里讨论的绘制过程只是对多边形中每个像素点的外观属性进行计算，因而光线跟踪系统也就不采用本章描述的绘制流水线。

本章重点描述基于多边形图形系统的绘制流水线细节，解释在绘制图像过程中必要的几种操作，最后重点介绍绘制流水线的OpenGL实现。

10.2　流水线

开始绘制一个真实场景的时候，只有很少的一些已知信息，如场景各部分的基础结构信息（如三角形、四边形、多边形、位图、纹理图、光源或裁剪面等）；每个顶点的2D屏幕坐标定义了每个顶点的几何信息外，还有诸如像素的深度、点的颜色、法向量、纹理名称和纹理坐标等附加信息。还有描述场景的基本信息，如是否采用深度缓存、平滑着色处理、照明模型、雾化效果设定等。绘制的任务就是综合所有这些已知信息，其中某些信息还会因场景中对象的不同而有所差别，生成这些信息所描述的最终图像。

整个过程可以分为几个阶段。其一，对场景中的每个多边形，根据其顶点得到多边形在光栅显示设备中相应扫描线的端点，多边形内部的像素点信息可以根据这些端点插值得到；其二，插值过程中根据颜色、光源或纹理数据确定扫描线上每个像素点的颜色值；其三，每个像素点的数据又进一步应用于深度测试、裁剪、雾化效果和颜色混合等过程。总而言之，

这些过程提供了根据特定的场景属性来生成高质量图像的所有计算。

我们已经探讨过几何流水线各阶段的计算实现，如图10-1前半部分所示。这些变换操作将模型顶点从三维模型空间转换到二维屏幕空间。从屏幕空间开始进入绘制流水线阶段，如图10-1后半部分所示。在此阶段，屏幕空间的顶点（加上伴随的其他信息）被转换为最终在显示器屏幕或其他显示设备上的像素。除了二维屏幕点的X和Y坐标之外，顶点数据结构还保留顶点其他信息，此像素点在原始模型空间的深度值（用来进行精确的多边形插值），根据简单颜色定义或者光照计算得到的颜色信息（用来确定多边形的颜色或者进行颜色插值），顶点纹理坐标（用来进行纹理映射），其他需要的信息视具体图形API而定。如顶点在世界坐标系下的法向量，这在使用Phong光照模型时需要用到。

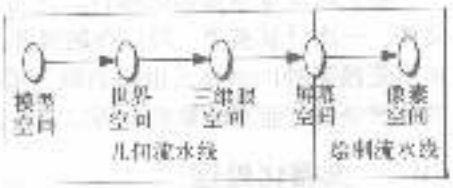


图10-1 从几何流水线到绘制流水线

380

绘制流水线从接收已变换过的模型数据（二维屏幕顶点及其该顶点定义的数据）开始。一个屏幕顶点不能构成完整的多边形，绘制流水线需要得到多边形所有顶点的信息。当一个多边形所有顶点的信息都准备好之后，就可以开始绘制该多边形。绘制过程包括确定多边形内部各像素点的特性，以及将可见的像素点写入图形输出缓存等步骤。

假定所使用的图形硬件是基于扫描线的，即图形硬件按扫描线顺序一次一行地生成图像，如图10-2所示。一条扫描线是显示设备上具有相同y值的像素点的集合，是显示设备上的一条水平行。在同一条扫描线上处于同一个多边形内部的像素点的集合称为一个片段（fragment）。绘制一个多边形需要确定组成多边形的所有片段，以及每个片段上所有像素点的特性。对于凸多边形，每条扫描线只与多边形相交一次而产生一个片段，而非凸多边形在一条扫描线上有可能产生多个片段。这是大多数图形API只能处理凸多边形的原因，处理非凸多边形时都需要将它们打散为多个凸多边形才能进一步处理。如在OpenGL中，所有用GL_POLYGON声明的多边形都默认为凸多边形来处理。如果实际上不是凸多边形，则将产生相当奇特的图像。



图10-2 凸多边形上的扫描线（左），非凸多边形上的扫描线（右）

一旦得到几何流水线产生的多边形各个顶点在屏幕空间的坐标值，绘制流水线的第一步工作就是将顶点插值得到多边形每条边上的点。这些点是扫描线段的端点。有了它们才能进一步处理这些扫描线段并写入帧缓存。必须将每条扫描线段与多边形边的交点的坐标进行插值，以计算多边形内将被投影到屏幕对应扫描线的原始点的坐标。同样的插值操作也应用于顶点的其他属性，如深度值和纹理坐标。插值计算可以采用线性方法或者透视校正方法，采用何种插值方法将会影响插值顶点的深度、纹理坐标及其他数据的计算。第8章曾经提及这点，下一节介绍光栅化时将作深入讨论。

381

计算出扫描线端点坐标以及它的各项数据后，就可以生成端点间扫描线上的各像素点，填充此片段。端点间的数据必须再一次进行插值处理，也可能再次应用透视校正。现在已经得到每个像素的真实颜色或者纹理坐标。但是，并非所有的像素都会写入输出缓存，因为可能存在深度测试和对场景的裁剪，所以每个像素在写入前必须作几项测试。如果应用了深度测试，那么只有深度值小于深度缓存中对应该像素的深度值时才能写入输出缓存，同时用新的深度值更新深度缓存。如果存在其他裁剪平面，像素点的原始坐标将被重新计算并与裁剪平面比较，根据比较结果将像素点写入缓存或者丢弃。如果像素点颜色带有α通道并且此像素

点可见，则在写入前需要对该像素点颜色和图像缓存中的颜色进行混合。如果存在雾化操作，那么像素写入前将会依据像素深度和雾化参数进行雾化计算。这一系列逐像素点的操作需要时间来执行，所以大多数操作可以根据需要来打开或关闭。

除了这些显示像素的操作，还可能需要产生像素纹理信息的操作。纹理图可能来自一个文件，一次计算或者一段保存的屏幕内存，但它们都必须转换为API需要的内部格式。纹理图通常是按某种内部格式组织的颜色值的阵列，阵列的索引就是模型的纹理坐标。由于像素点的纹理坐标可能并非整数，所以需要进行一些计算，以从纹理图中得到适用的像素颜色。

10.3 光栅化处理

光栅化处理在绘制流水线中担任重要角色。多边形在进入光栅化处理时以一组屏幕空间中的顶点形式出现，几何顶点转换为扫描线片段，以备后续更深入的逐片段操作之用。这个转换过程称作多边形扫描转换。通过扫描转换，多边形用可以被系统显示的一组像素来表示。扫描转换在OpenGL内部进行，所以并非必须掌握它，但是了解它的相关细节可以帮助理解计算机图形学的一些基础概念。本节阐述光栅化处理的细节。

首先回忆一下几何顶点进入绘制流水线时附带的信息：其中有顶点的二维屏幕坐标，这是将顶点从三维眼坐标向二维眼坐标投影，并映射到屏幕坐标的计算得到的；还有每个顶点在三维眼坐标中的z值，z值无需显示，所以并不需要转换为屏幕坐标，但需要它进行某些计
382 算。z值可以转换为更方便的表达形式，如用整数0表示视域体的前平面位置，用OpenGL支持的最大整数表示视域体的后平面位置。几何顶点信息还包括顶点颜色，通常是RGB三元组，可以由模型本身提供，或者根据光照模型计算得到。几何顶点信息还可能包括顶点的法向量（使用平滑着色处理或者其他着色处理计算时要用到法向量）和纹理坐标。可见每个顶点都附带了大量的上述信息而非简单的屏幕坐标。

光栅化处理过程必须根据多边形顶点几何信息所定义的线段进行扫描转换，插值线段端点以得到片段，并获得每一片段在多边形中的像素，以确定所处理多边形的完整像素集。扫描转换首先对多边形的边进行操作以得到代表边的所有像素，当所有边操作完成后，就得到了界定此多边形的所有像素。对凸多边形而言，任意扫描线都只与两条边相交，所以可以将像素点组织为一组像素点对的形式，一条扫描线对应一对像素点，每一对像素点也确定了一条片段，即这对像素点之间的全部像素构成这一片段。

当前存在许多对线段进行光栅化的算法，这里从也许是最简单的一种算法数字微分分析法（Digital Differential Analyzer, DDA）开始介绍。此算法通常使用直线方程和近似舍入方式根据线段两端点的屏幕空间坐标求得线段在每条扫描线上的像素点，由于像素点的坐标是整数而线段是连续的，所以必须看到任何光栅化都只能生成走样的近似线段而非精确线段。DDA算法通过在扫描线上取最接近真实值的点作为最佳近似的像素点。

开始之前，先假定线段的端点为(X_1, Y_1)和(X_2, Y_2)，标记$\Delta X = X_2 - X_1$，$\Delta Y = Y_2 - Y_1$，线段的斜率$m = \Delta Y/\Delta X$。为方便起见，假定ΔX和ΔY都是正值。如果不是，可以调整代码中的代数符号。这在稍后讨论。同时还注意到线段可以作任意平移变换，因为计算出线上的所有像素后，可以通过平移线段上的每个像素点来平移变换整条线段，所以，可以假定线段位于第一象限。

接着分析在$\Delta Y > \Delta X$或者$\Delta Y < \Delta X$两种情况下，线段上像素的分布状态差异。如果$\Delta Y > \Delta X$，那么线段在每条扫描线上只有一个像素；如果$\Delta Y < \Delta X$，则线段在屏幕空间里的任意垂直线上只有一个像素。这里只讨论$\Delta Y > \Delta X$情形下的算法，此时X可以看作Y的函数，另一情形下的算法在数学上可以通过交换X和Y来实现。

对Y_1和Y_2间的每条扫描线，DDA算法希望求出一个像素点，它最接近线段上的真实点，用它来表示线段上的真实点。先从X作为Y的函数直线方程式开始：

$$X = X_1 + ((Y - Y_1)/(Y_2 - Y_1)) * (X_2 - X_1)$$

表达式$(X_2 - X_1)/(Y_2 - Y_1)$表示直线的斜率$\Delta X/\Delta Y$，用它代替更常用的$\Delta Y/\Delta X$，因为这里需要计算在Y轴上两条扫描线间X的变化幅度。根据方程式对扫描线Y（整数），计算出X（浮点数），取最接近X的整数作为X的最终值，用X和Y确定的像素点来表示线上的真实点。用伪代码描述的算法如下：

```
Input:  屏幕上的两个点 (X1, Y1) 和 (X2, Y2)，其中Y2>Y1并且
        (Y2-Y1)/(X2-X1) > 1
Output: 屏幕空间中代表两点之间线段的像素集合

for (int Y = Y1; Y < Y2; Y++) {
    // 下面的讨论不包含
    float P = (Y-Y1)/(Y2-Y1);
    float X0 = X1 + P*(X2-X1);
    int X = round(X0);
    setpixel (X,Y);
}
```

另外介绍一种对线段进行扫描转换的Bresenham算法，它以一种误差值处理作为选择像素的基准。下面的讨论只考虑简单情形，即插值线段的斜率不超过1.0，并取左端点为$(0, 0)$位置像素点。满足上述条件的线段位于平面中第一八分之一象限。对这样的线段，屏幕坐标系下对应每个X值只需确定一个像素，寻找新像素的问题变为选择紧邻前一像素点（具有相同的Y值）的像素或者选择比前一像素点的Y值大一个单位的像素作为新像素。这是Bresenham算法要解决的问题。

Bresenham算法的输入是两个顶点(X_0, Y_0)和(X_1, Y_1)。假定$X_0 < X_1$且$Y_0 < Y_1$，记号$DX = (X_1 - X_0)$和$DY = (Y_1 - Y_0)$表示两个方向上的距离，算法的任务是找到一个简单方法来确定：对任意X值，其对应的Y值是与$X-1$处的Y值相等还是$X-1$处的Y值加1。

首先从线段最左下角的顶点X_0开始，哪一个将是$X_0 + 1$处的像素？图10-3左图把这个问题归结为：真实直线上$X = X_0 + 1$处的Y值是否大于$Y_0 + 0.5$？也就是看(DY/DX)是否大于1/2，或者说是否$2*DY > DX$。这给出了初始的判断条件$P = 2*DY - DX$，和判断逻辑：如果$P > 0$则Y加1，反之则不增加。

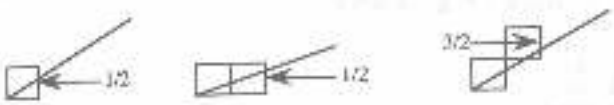


图10-3 从一个像素确定下一个像素的判断标准

找到直线的第一个新顶点后，再来看如何确定第二个顶点。如果第一个顶点没有改变Y值，则如图10-3中图所示的情形。此情形下，第二个顶点的确定条件变为是否满足$2*(DY/DX) > 1/2$，也可表示为$4*DY > DX$或$4*DY - DX > 0$。引入前面提到的标记P，条件变为$P + 2*DY > 0$，另引入标记$C_1 = 2*DY$，则一般的操作规则为：如$P < 0$，则将判断条件P改为$P = P + C_1$。

如果第一个新顶点的Y改变了，如图10-3右图所示的情形。此时判断第二个顶点的条件为是否$2*(DY/DX) - 1 > 1/2$，不等式等价变换为$2*DY > 3*DX/2$，或$4*DY - 3*DX > 0$，引入P之后变为$P + 2*(DY - DX) > 0$。同样，令$C_2 = 2*(DY - DX)$，得到第二个一般操作规则：如果$P > 0$，则判断条件P改为$P = P + C_2$。

算法整个处理过程如下：定义判断条件的初始值，作出判断并确定下一个像素，再根据最近一次的判断更新判断条件，如此循环反复就可以从找到第一个像素开始直到最后一个像

素，找出线段上全部像素。将算法推广到普遍情形并无困难。所以，该算法能插值任意方向和斜率的直线。上面对Bresenham算法的讨论可以实现如下：

```
Bresline(x1, y1, x2, y2)
int x1, y1, x2, y2;
{   int dx, dy, bx, by, xsign, ysign, p, const1, const2;
    int sign;

    bx = x1;
    by = y1;
    dx = (x2 - x1);
    dy = (y2 - y1);
    if (dy == 0)  /* 如水平线 */
    { xsign = dx / abs(dx);
        setpixel(bx, by, COLOR);
        while(bx != x2)
            [ bx += xsign;
              setpixel(bx, by, COLOR);
            ]
        }
else if (dx == 0) /* 如垂直线 */
{ ysign = dy / abs(dy);
    setpixel(bx, by, COLOR);
    while(by != y2)
    [ by += ysign;
      setpixel(bx, by, COLOR);
    }
}
else                    /* 使用Bresenham 算法 */
{ xsign = dx / abs(dx);
    ysign = dy / abs(dy);
    dx = abs(dx);
    dy = abs(dy);
    setpixel(bx, by, COLOR); /* 设置直线的初始点 */
    if(dx < dy)     /* 直线垂直程度大于水平程度 */
    { p = 2 * dx - dy;
      const1 = 2 * dx;
      const2 = 2 * (dx - dy);
      while (by != y2)
         { by = by + ysign;
           if(p < 0) p = p + const1;
           else
              { p = p + const2;
                bx = bx + xsign;
              }
              setpixel(bx, by, COLOR);
         }
    }
    else            /* 直线水平程度大于垂直程度 */
    { p = 2 * dy - dx;
        const2 = 2 * (dy - dx);
        const1 = 2 * dy;
        while (bx != x2)
        [ bx = bx + xsign;
          if (p < 0) p = p + const1;
          else
          { p = p + const2;
            by = by + ysign;
          }
          setpixel(bx, by, COLOR);
        }
      ]
    }
]
```

385

此算法适用于与深度无关的属性的插值计算，为属性设定一个步长值，每产生一个新像素就将此属性值增加一个步长值。Bresenham算法也适用于与深度有关联的属性。插值此类属性时，需要先求出像素的近似深度，再用这一近似深度进一步做透视校正的纹理插值。

这些算法的工作过程如图10-4所示，左图在像素光栅阵列中显示了一条线段的两个端点，假定每个像素都取其左下角的坐标值。这也是通常的惯例。右图显示了扫描转换过程如何在

两端点之间的扫描线上确定像素。请注意，对X值的向上舍入将得到实际线段右边的像素，这使得由像素构成的两端点间的线段显得较为自然。

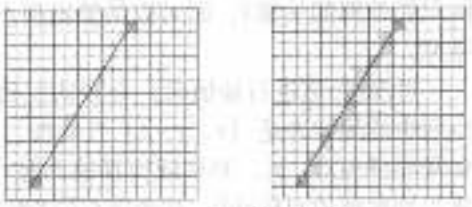


图10-4 边线的扫描转换

当扫描转换一个完整的多边形时，首先对构成多边形的所有边进行扫描转换。边的像素不会马上写入帧缓存而是存入一个像素阵列，以备进一步用边的像素生成多边形的片段。比较理想的是采用二维数组，用一个索引表示此像素属于哪一条扫描线，用另一个索引指向此扫描线上的两个像素（回忆一下，对于凸多边形，任意扫描线与之相交于零个或者两个点）。将扫描转换的每个像素写入合适的数组，就可以通过像素的X值在二维数组中找到对应的扫描线。每个这样的二维数组表示多边形的一条片段——多边形内部有同样扫描线值的线段。

386

处理这些扫描线片段前需要理解一些细节，因为前述的定义中包括一些不明确点。例如，前面没有提到如何得到多边形的最高或最低顶点处的“片段”。这两个位置的片段会两次包含同一个像素点。两条边线上的共享顶点也有类似问题。前面也没有提到如何处理多边形与其他多边形的公共边问题，公共边应该只属于其中一个多边形的一部分而不属于两个多边形。如果两个多边形都考虑此公共边，则最终图像将会受到多边形绘制顺序的影响，这容易产生问题。为解决这些问题，这里介绍一些生成片段的约定俗成的方法。第一，只生成多边形底部水平边的片段，而不生成顶部水平边的片段。这可以很容易地通过任意非水平边的线段扫描转换时剔除最顶部的像素来做到。第二，只考虑多边形左侧的边而忽略右侧的边。要做到这点，只需在定义每条扫描线的片段时作如下约定即可：扫描线片段将包含“从（且包含）左侧像素直到（但不包含）右侧像素”。最后，将所有扫描线作为片段处理。所以，无需处理多边形的水平边。

依据上述算法和惯例，可以处理任意凸多边形，并生成一组帧缓存中表示该多边形的片段。然而，当插值顶点时，每个像素的其他属性也必须进行插值，包括颜色、深度和纹理坐标等。其中一些属性如颜色，与像素深度无关，但另外一些属性如纹理坐标和深度值本身，则与深度值有关联。任何与深度无关的属性可以简单地沿着边线进行线性插值得到每个片段端点上的属性值，然后再根据片段端点上的属性值沿着片段进行插值得到片段内部各像素点的属性值。这个插值过程与DDA或Bresenham算法插值几何数据的过程完全一致。但对于深度相关的属性，属性与显示平面上像素点之间没有线性关系，通常用的线性插值可能失败，如图10-5所示（摘自第8章）。

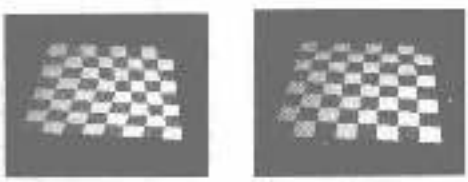
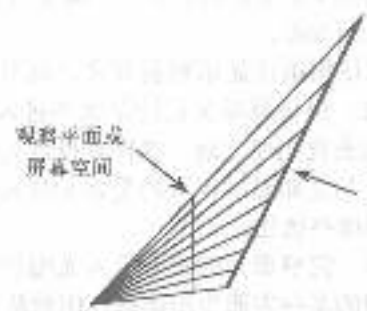


图10-5 不带透视校正的纹理映射的矩形纹理（左）和带有透视校正的纹理映射的矩形纹理（右）



387

图10-6 与线性像素序列对应的原始边上的点分布

当在屏幕空间对像素坐标进行线性插值时，线段上的每一点与其对应像素之间并非是线性分布关系，如图10-6所示，真实线段顶部点的间隔距离远大于底部点的间隔距离。所以必须采用透视校正来重构三维眼坐标系中的真实点。回忆在第2章中介绍的透视投影操作是通过将顶点原始坐标除以顶点的z值来计算顶点在二维眼空间坐标的方法，所以需要

真实深度值z来计算顶点的原始坐标。一旦计算或者估算出原始深度由于这样的像素坐标（实际只能近似的估算），则可以估算出顶点原始坐标，再利用简单的几何原理估算出真实的纹理坐标。

对深度z值进行插值时，必须认识到是在对经过透视变换后的点进行插值。如果在三维眼空间中的原始点是（x，y，z）且其在二维眼空间中是（X，Y），透视变换为$X = x/z$和$Y = y/z$。
388 如果要插值x_1，x_2，则实际要插值的是x_1/z_1和x_2/z_2。为此，首先必须插值$1/z_1$和$1/z_2$以估算$1/z$，再得到被插值点的z值。单考虑点的X坐标时有$x = (1-t)*x_1 + t*x_2$，相应的z值为$(1-t)/z_1 + t/z_2 = ((1-t)*z_2 + t*z_1)/(z_1*z_2)$。利用这个估算出的z值可插值得到x、y值，通过它们的相乘可重构出三维眼坐标系中的原始点。

10.4 OpenGL的绘制流水线

OpenGL系统具有如图10-7系统结构所描述的处理过程。系统通过程序中的OpenGL函数从CPU获得输入，系统的输出为帧缓存中的像素。输入信息中顶点的几何信息、几何变换信息进入到求值器，纹理信息通过像素操作后进入到纹理内存。这些操作的很多细节通过glEnable函数中的参数设置来控制，并作为状态保存在系统中。这里将概要介绍系统操作的不同阶段，以便使用者理解如何将几何描述转变成帧缓存中的图像。这里将采用OpenGL描述进行介绍。

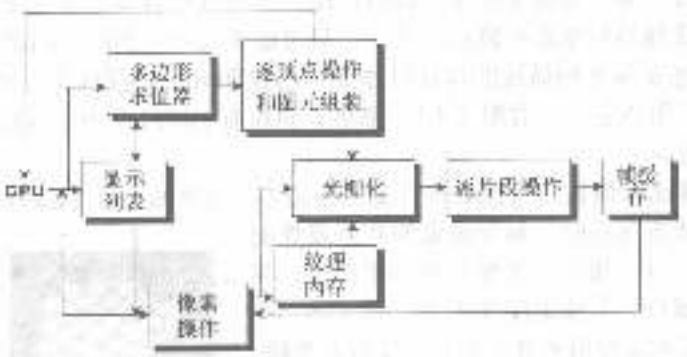


图10-7 OpenGL系统模型

首先介绍简单多边形的立即模式操作。从CPU传来用户定义的模型空间顶点几何数据被送入多项式求值器，进行完整的几何变换和裁剪操作，本质上讲这就是几何流水线的任务。接着，输出的三维顶点信息进入到逐顶点操作，同时应用光照模型计算每个顶点的颜色数据。这一步的结果是变换后的顶点（附带经过此过程而保留下来的顶点的其他信息），为图元装配
389 和光栅化而准备。

如果使用编译显示列表方式，而不采用立即模式，则顶点和几何变换数据将发送到多边形求值器，但计算结束后的结果不进入光栅化的顶点操作，而改为进入显示列表内存备用。当显示列表被处理之后，操作直接进入绘制处理阶段，如同已经通过了立即模式的逐顶点操作一样。与立即模式不同的是显示列表可以省略掉几何变换的计算开销，且数据进入显示列表时可以做些优化。

此刻，完整图元的顶点进入光栅化阶段，进行前述的插值和扫描线处理。在这个阶段需要对处理的某些方面作出选择，例如是否进行第8章中提到的透视校正插值。当每个像素的可见性和颜色算出后，根据用户设置计算出每个像素的颜色或者纹理数据，得到的扫描线数据即可准备进行逐片段操作。

读者可能没有注意到图10-7中从逐顶点操作到CPU之间的反馈连接线，而它十分重要。这个反馈机制支持第7章中讨论的拾取和选择操作。逐像素操作和CPU之间的反馈联系可以让系统知道某个给定像素与生成的图元对象有关，并通知选择缓存以及返回给应用程序。

除了顶点操作，OpenGL还包括其他操作，如在处理样条和基于一组控制顶点的求值器时，可以使用多项式求值器。这些求值器可以运用在几何数据或许多其他图形元素中。求值器产生的多项式被处理后得到的结果可供系统使用。

10.4.1 绘制流水线中的纹理映射

纹理映射涉及纹理内存和绘制系统的其他部分。一个纹理图可以从文件中读入，或者对帧缓存或其他来源的数据应用像素操作而得。纹理图是光栅化过程所需纹理数据的来源。纹理图的内容传输到纹理内存以备纹理映射。在图10-7中从帧缓存回到像素操作的箭头表明帧缓存中的信息可以被取出并写入帧缓存的其他部分；从帧缓存到纹理内存的箭头提示了帧缓存中的信息甚至可以用作纹理图本身，详细过程如图10-8所示。

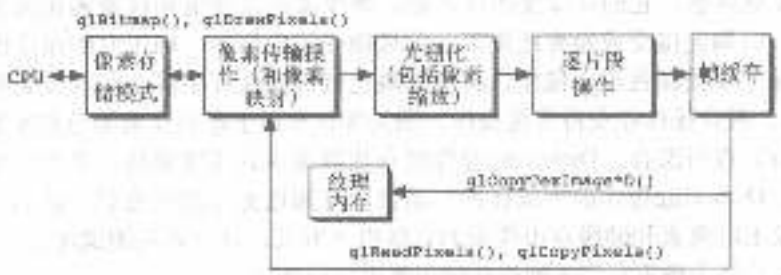


图10-8 纹理图处理

由图可见，纹理内存的内容可以来自CPU，从文件读入之后纹理内容在CPU转化为阵列格式。因为OpenGL并不知道文件格式，所以必须将普通图形文件格式（参见[MUR1]）得到的数据解码为OpenGL支持的格式。同时如第8章讨论纹理映射时提到，也可以用计算的结果作为纹理阵列的内容。甚至纹理内存可以复制帧缓存中的数据，可使用glCopyTexImage*D(...)函数或者其他像素级操作。这种方法可以生成非常有趣的纹理，即使用户使用的OpenGL版本不支持多纹理映射。

一个像素的纹理坐标正好匹配纹理点索引的情形非常罕见。像素可用实数型纹理坐标代替整数型纹理坐标。这时像素的纹理数据可能需要通过计算才能得到，包括计算求得最近的纹理点或者计算相邻点数据线性组合作为纹理数据，如同第8章讨论纹理映射时所描述。 390

10.4.2 逐片段操作

OpenGL的主要功能表现在光栅化进程中片段的处理能力，或处理一组扫描线数据的能力。片段操作包括图10-9所示的子流水线操作。其中某些操作属于OpenGL高级功能范畴，这里不作深入介绍。这些操作的大多数需要先启动（例如用glEnable(GL_SCISSOR_TEST)，GLEnable(GL_STENCIL_TEST)和其他类似操作来启动），还有一些操作需要用户的图形系统具备特殊功能。如果读者希望了解本书介绍不够详细的内容，可以参考OpenGL用户手册或其他高级教程。

第一个片段操作是剪裁测试，定义为glScissor(...)，用来增加一次对矩形包围盒的裁剪。第二个操作是测试像素的α值来生成纹理掩膜，定义为glAlphaFunc(...)。下一个操作是Stencil

测试，它类似于α测试，但它采用Stencil缓存中的值来生成掩膜。Stencil操作根据你画的Stencil掩膜通过普通OpenGL操作来完成。在画片段时可运用Stencil掩膜选择删除片段中的某
391 一个像素。Stencil测试就是把Stencil缓存中的值与一参考值进行比较，片段中的每一个像素可以根据Stencil测试结果选择保持不变或用新设置的值来代替。Stencil测试的关键函数有glStencilFunc(...)，用来设定测试函数，glStencilMask(...)控制对Stencil缓存的写入，glStencilOp(...)确定Stencil测试的执行方式。

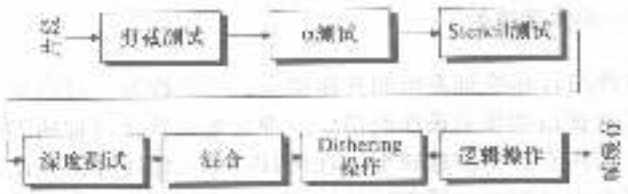


图10-9 片段处理的细节

下一组操作更熟悉，它们从深度测试开始，将像素的深度值和该像素在深度缓存中的深度值进行比较，以确定接受或拒弃此像素。如果接受了此像素，则用它的深度值来更新深度缓存中的值。接下来是颜色混合操作，将像素颜色值与帧缓存中混合函数设置的颜色按像素的α值进行混合。混合操作亦支持雾化操作，因为雾化本质上是对像素颜色和雾色（依据像素深度值计算而得）进行混合。Dithering操作组合周围像素的不同颜色，并加以平均生成所需的颜色，因此，Dithering操作能生成比图形系统固有颜色更丰富的色彩。最后，逻辑操作允许用户确定片段上的像素和帧缓存中像素的具体组合方式。这一系列测试决定片段是否可见。若可见，则确定片段上像素在写入帧缓存时还要进行的处理。

10.4.3 OpenGL与可编程着色器

本节简要介绍可编程顶点操作和可编程片段操作以及着色语言的概念，使读者了解这些思想的背景。最新的OpenGL版本或普遍接受的扩展库都支持这些可编程操作。

在标准的OpenGL中，当顶点进入绘制流水线时，除了坐标外还有大量已知信息，如颜色（不管是由光照模型确定还是直接设置），或许还有纹理坐标。除了上述附加信息，没有理由表明顶点不能附加更多的信息，例如可以增加位移向量、多达八个的多纹理坐标、以及一些特殊的变换操作等，甚至可以储存程序的地址，这些程序可以用来计算形状、颜色、用面向光照的法向量代替普通几何法向量来计算各向异性的着色处理以及凹凸纹理图等。图形卡开始拥有强大的逐顶点编程能力，如每个顶点带有16个或更多的四维实数向量来装载附加数据，虽然每个图形卡基于其特定的体系结构具有完全不同的指令集。

除了可以在每个顶点附带一段程序外，逐片段操作中除执行上述片段操作外，还可以增加一些其他处理技术。一些图形卡的可编程操作模仿纹理组合操作的概念，可以对片段作更多的操作处理。增加的可编程功能使得绘制流水线可以理解为可编程绘制流水线，带有三个可编程阶段：群组处理、顶点处理和片段处理。其中两个阶段与图10-7描述的OpenGL绘制流水线类似，而群组处理是一组对顶点集合的操作，而非单个顶点的操作以提高效率。这条可编程流水线如图10-10所示。

这为OpenGL高级（或扩展）版或其他图形API提供了一个新的开发方式，即为每个顶点提供一段程序来计算上面提到的顶点属性。为了兼容最广范围的硬件，此类程序设计语言应独立于特定图形卡，而由图形API提供编译方式或解释执行方式为图形卡生成所需的操作。了

解图形API的高级编程方式能做些什么将是非常有意义的事情。 392

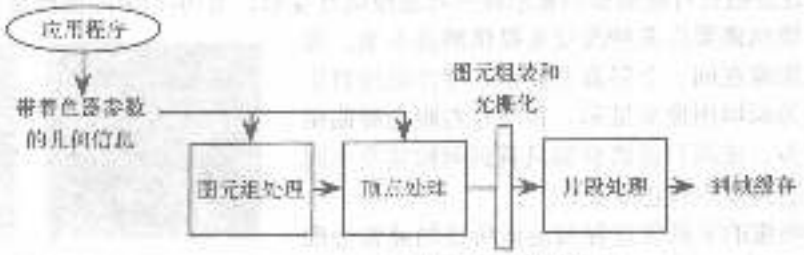


图10-10 带三个可编程阶段的可编程流水线

10.4.4 图形卡绘制流水线实现的实例

图10-7到图10-9描述的系统具有通用性，展示了实现OpenGL处理过程所需的各项操作。实际上，实现系统的方式有很多种，图10-11显示了一块与OpenGL兼容的图形卡上的实现图解，它很典型，也很简单。

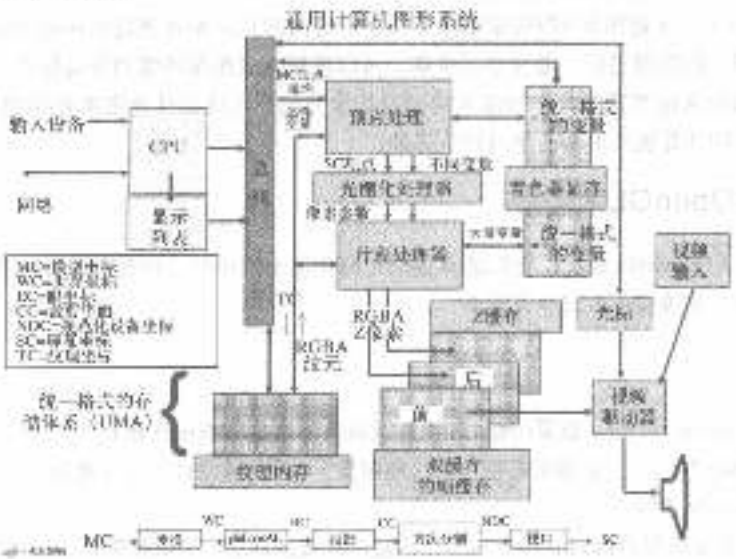


图10-11 典型图形卡的OpenGL实现 393

流水线处理器执行几何处理，从原始三维模型空间的顶点产生二维屏幕像素。纹理内存是相对独立的，用来保存经CPU解码后的纹理图。光栅化处理器处理光栅化和逐片段操作；Z缓存和双缓存的帧缓存保存这些操作的输入和输出数据，光标单独处理，因其需要独立于帧缓存内容自由移动。视频驱动器转换帧缓存内容和其他输入信号（光标、视频）来驱动监视器等显示设备。这种API功能和硬件功能的一一对应关系是OpenGL成为现代图形应用的重要部分的原因，它为市场提供了良好的性价比。

10.5 图形卡的部分三维视图变换操作

除了前面介绍的生成三维视图的技术外，还有一类技术使图像可以被某些特殊硬件选取和显示，如StereoGraphics公司的CrystalEyes眼镜。这类技术从帧缓存中获取数据并交替地为左右眼提供图像，使观众将两幅图像当作同一个场景的两个视图。有很多不同方法可以让左

眼图像和右眼图像被特殊显示硬件选取，包括左右排列图像、上下排列图像和交错排列图像等方式。这些组合可能需要由显示硬件对图像做些变形。图10-12中的图像显示了相应的变形。视频流可能也需要作某种改变来提供解决方案。如果左右眼图像在同一个屏幕上显示，硬件必须将信号流分离为两幅图像来显示，并与左右眼交替偏振隐藏相同步，使两只眼睛分别只看到两幅完全不同
394 的图像，达到自然的立体效果。

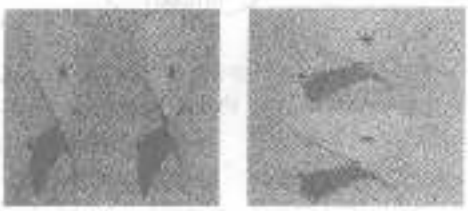


图10-12 左右排列图像（左）和上下排列图像（右）

这些图像的光栅化过程与前面所述的通常处理有所不同。两幅中的每幅图像发送到不同的显示缓存并复制每一个像素。对左右排列的图像而言，像素水平复制到缓存，达到中心线后交换缓存；对上下排列的图像，像素垂直复制，达到中心线后交换缓存。通过系统维护，由一个源图像得到两个单独图像，生成交替显示硬件所需的双重图像。

10.6 小结

本章介绍了应用大多数图形API将原始图形程序中给出的几何和外观信息在输出颜色缓存中生成图像（一幅像素光栅）的处理过程。通过学习本章，可以理解生成图像所需的各类操作，从而理解为何某些类型图像的生成比其他更费时，也为深入理解第12章介绍的生成高性能图形程序的某些技术做准备，为以后进行更深入的计算机图形学的学习打下基础。

10.7 本章的OpenGL术语表

本章介绍了一些OpenGL函数，它们是多边形光栅化流水线中的一些独特操作。对简单程序而言它们的作用有限，但在高级操作中却非常重要。

OpenGL函数

glAlphaFunc(parm, value)：设置α测试的函数（从符号选项列表中选择）

glCopyTexImage*D(...)：复制像素到一个纹理图像，图像维数由*（1，2或3）给定，由一组参数定义格式、尺寸和像素位置等

glScissor()：定义图形窗口内的一个矩形，当裁剪测试启动后，只有矩形内部的像素能被修改

glStencilFunc()：设置Stencil测试的函数和参考值

glStencilMask()：定义一个位掩膜平面来控制对Stencil平面上特定位的写操作

glStencilOp()：定义Stencil测试通过或失败后的操作

OpenGL参数

GL_SCISSOR_TEST：glEnable()的参数，指示应用裁剪测试

GL_STENCIL_TEST：glEnable()的参数，指示应用Stencil测试

10.8 思考题

395 1. 应用平滑着色处理模型和顶点颜色各不相同的唯一小三角形图像。用计算机屏幕抓取工具抓取显示在屏幕上的图像，用Photoshop之类可以放大图像的工具打开抓取的图像。放大此图像并观察此三角形，鉴别出三角形上的扫描线，并注意穿过扫描线构成三角形的片段。请问从三角形顶部向底部移动时片

段将如何变化?

2. 手工生成一小段片段（例如10个像素长），每个像素带有深度和颜色（包括α值）。手工生成一条带自己深度和颜色信息的扫描线。使用深度和颜色缓存。手工完成将片段置入扫描线上的操作。说明如何使用深度缓存来决定片段上的像素是否可用。如果可用，说明如何使用颜色缓存和像素的α值确定扫描线上像素的颜色。可以选用任意的混合函数。
3. 说明为何对凸多边形而言，每条扫描线与之相交为唯一片段，但反之不正确：因为存在一类多边形，每条扫描线与之相交为单一片段，但它不是凸多边形。

10.9 练习题

1. 生成一个由正方形网格构成的小屏幕，每个方格可以填充任意颜色——一种"胖像素"屏幕。改写Bresenham算法来插值一个深度无关的顶点属性（例如颜色），用插值颜色值绘制方格颜色来体会这一操作是如何工作的。尝试能否插值本章提到的深度或者带透视校正的纹理坐标。

10.10 实验题

1. 在讨论片段处理时提到需用不同操作支持不同类型的图形处理。设计一个实验，定义一些简单几何物体，用不同的技术绘制它们，比较绘制时间的长短。因为现在的图形系统都非常快，需要绘制成千甚至上万个简单物体以便测量差别。实验中请使用以下几种三角形：采用平面着色处理绘制的三角形，采用平滑着色处理绘制的三角形和纹理映射的三角形。

396

第11章　动力学和动画

本章将介绍怎样创建动画图像。所谓动画图像是指图像不受用户或者观察者干预，可随时间的推进连续播放。本章讨论的话题包括理解事物运动必须遵循的物理定律及其与观察者交流信息的方式。我们站在一个比较广的视角来看动画的制作，而不仅仅拘泥于怎样让图像产生运动效果的技术问题。我们还会通过一些例子进一步介绍和讨论动画技术，这些例子仅初步向我们展示了动画制作过程。读者需要理解具体问题的运动规律，才能制作出逼真的动画效果。

计算机动画的内容十分丰富，有大量相关的图书和课程，我们不打算在这本计算机图形学初级课程中对动画作深入介绍。当然，学习以及熟练运用计算机动画工具包十分重要，而本书的重点主要放在制作相对简单的动画，通过它们来介绍本书中创建的模型和图像，最后我们还会介绍一些科学领域的例子。

动画是一个连续播放的图像序列，序列中的每一幅图像叫做一帧，播放的速度要足够快，以使观察者看来事物在帧与帧之间的运动是平滑的。动画主要分为两类，一类叫做实时动画，即通过程序的运行在屏幕上展现每一帧图像，另一类叫做录制动画。这种动画技术是先将每一帧图像绘制好，保存在一个可以播放的文件格式中（也可以是单独制作的电影或者视频，我们将在第15章详细介绍）。这两种动画技术都需要对模型、光照计算以及随时间变化的视点做详细的规划。为了能够高效地实时生成动画，实时动画技术采用相对简单的模型和绘制算法来达到比较高的屏幕刷新率，但是录制动画趋向于采用更复杂的场景模型和更细致的绘制方法。由于采用了简单的模型和绘制方法，实时动画可能没有录制动画来得真实，而且由于图像生成速度跟不上，会使帧速率下降，从而达不到很好的实时效果。尽管这样，实时动画是由运行的程序产生动画效果，如果允许用户和动画物体做交互操作，用户就会通过实时动画技术达到身临其境的使用效果。随着计算机速度以及图形硬件性能的不断提高，实时动画技术得到了越来越多的应用，几年前录制动画中才能完成的效果，现在已经可以很好地在实时环境中实现。

本章在实时动画和录制动画之间不会厚此薄彼，将会介绍它们的共性方法。我们认为制作动画的真正目的是实现视觉交流，在针对视觉交流的动画领域中有大量的专业词汇和技术。本章将对这些问题做一个浏览，而不是对每一个问题做深入介绍。但我们建议读者花点时间看一些成功的动画片，尝试找出哪些元素使它们成功，我们还建议读者从搞明白要实现的某个想法开始，努力去实现它。

不管要实现什么样的动画，关键是要生成随时间变化的场景，以及针对这种变化绘制一段连续的图像序列。由于动画和场景设计紧密结合在一起，需要读者掌握场景随时间推进产生变化的概念。

本章和其他章节不太一样，实际上没有图例可以真正阐明本章的内容，我们讨论的是关于运动的问题，而印刷品还不能有效地再现运动属性，我们决定仅收录动画的代码，而不是生成的动画视频本身，因为本书是关于怎样创建图形以及实现可视交流的教材，而不是简单的为了让读者欣赏生成的图像。读者可以运行我们提供的代码，在自己的计算机系统上观看最后生成的效果。要注意，对于大部分的例子程序，计算机系统速度会对动画的速度有一定

的影响，使动画中事物的运动速度可能和期望的速度快慢不一样。

要理解本章的大部分内容，读者必须理解怎样通过参数来定义场景的视图，包括尺寸、形状、位置、朝向、外观属性以及其他的方方面面。可能最好的方法就是通过场景图来定义视图，读者要理解怎样逐帧地改变这些参数，来控制随时改变视图。通常采用基于时间的事件来生成新的动画帧，比如采用idle事件或timer事件，或者直接用系统时钟，在生成新的一帧图像后更新视图的参数，然后再生成下一帧图像。

11.1 一个例子

当我们在第0章介绍图形学编程的时候，用了一个热量在棒内流动的例子。我们用动画技术来旋转这根金属棒，让用户能从各个角度观察，展示棒中温度随时间的变化，从而使用户理解热量的流动（这种热的流动是不可见的）。这是一个自然科学领域中动画的应用例子，还有很多动画应用的例子。特别是可以看见物体在空间中运动的动画，为此在本章一开始我们通过介绍一个非常有用的建模技术粒子系统，来说明怎样应用粒子系统来生成动画。

粒子系统是一些点或者对象的集合，比如可用非常小的球来代替这些点，那么每一个球代表了某一种物体或者过程。比如把每一个点视为一个小的液滴，这样就可以用粒子系统来 398
为液体建模，然后引入力使这些粒子运动起来进而模拟液体的流动，喷泉就是一个简单的液体粒子系统的例子。另一个常见的例子就是焰火，最基本的方法就是为每一个粒子定义位置和初速度，然后采用物理定律来计算给定环境中粒子的加速度，对于每一个时间段，计算出新的位置和速度，并显示所有活动粒子。活动粒子是按照程序的物理规律运动并显示出来的粒子；非活动粒子是准备激活为活动粒子并绘制的粒子，或者是粒子的物理运动已经结束，不需要继续绘制和显示的粒子。对每个活动粒子的计算可以采用合适的技术，但有一些标准的算法可供选择，如物理学的标准运动方程法（如果封闭解是已知的），差分方程法（新的速度是初始位置上的速度加上一个加速度，新的位置根据这个速度计算出相对原位置移动的距离）或者用数值积分法计算新位置和速度。

我们用粒子系统对瀑布建模作为例子，如图11-1所示。假设水沿一个固定宽度的河道以恒定速度流下，直到一个陡坡。然后水从这个位置以重力加速度洒落，直到撞到一个障碍物，水开始飞溅，然后继续下落。不用看代码我们就知道这个模型有四个关键步骤：粒子在河道中的运动，粒子下降运动，粒子撞击障碍物以及反弹，还有粒子的继续下落。通过一些简化（包括只显示粒子，用球体来模拟粒子，只在三个光源的环境中绘制粒子），得到了这段动画中的一帧。



图11-1 粒子系统动画中的一帧

先建立一个大的粒子数组，每一个时间段从这个大的粒子集合中选择生成一些新的粒子，它们沿近似水平的虚拟河道以小范围内随机的速度流下（图中没有显示出这个河道），在河道的尽头粒子开始下落直到撞到障碍物。在撞击障碍物的时候，根据与障碍物的反弹计算出新的速度，然后继续下落直到碰到下一级。当这些粒子完成了运动周期，便消失，返回到可用的粒子集合中。 399

粒子运动可以用不同的方法建模。一种方法就是用普通物理定律来计算粒子的水平运动：

$$x = x_0 + v_x \Delta t$$

这里忽略摩擦和重力的影响，水平速度v_x是恒定的。对垂直运动分量，在粒子离开河道的时候，可以用下面的公式来计算重力下的垂直运动：

$$y = y_0 + v_y \Delta t + 0.5g \Delta t^2$$

这里y_0是河道的高度，v_y是初始的y分量速度，或者就是0，因为我们假设河道是水平的，水平速度v_x和前面一样保持不变。当粒子撞击固定位置障碍物的时候，将生成新的y_0（现在是障碍物的高度）和v_{y0}。粒子撞击障碍物的速度按照前一点和当前点的y分量插值来计算。然后将速度反向得到粒子反弹后的速度，再对x和y分量各加上一个随机扰动实现飞溅效果。然后粒子按照新的反弹速度继续下落，直到一个“谷底”y位置，粒子完成运动周期，并消失，返回给粒子集合。

我们实现图11-1动画的方法如下：保存位置和速度数据。粒子在河道上运动的时候垂直加速度分量为零，当粒子离开河道的时候以标准的32m/s²的重力加速度下落。这个加速度按时间等比例地加入微分方程的垂直速度。然后计算粒子的位移再为下一次绘制更新位置。在没有和障碍物撞击之前，按照前面的方法计算垂直方向的速度。在一个7维数组中存放每一个粒子的数据，包括位置和速度，还有粒子是否处于活动状态的逻辑值。在每一帧开始，可以从非活动粒子中选择某些粒子放到粒子系统中运动。当粒子下落到运动终止位置的时候再设置回非活动状态。每个粒子的创建、流动、下落、撞击、再下落，最后消失的过程不断重复。我们没有在本章中详细列出这段代码，读者可以查阅随书附带的资料。

11.2 动画的分类

动画是生成图像序列以及呈现图像序列的过程，使观察者能感知到平滑流畅的运动序列。运动序列有助于阐明事物之间的关系，能表现装配零部件的过程，有助于设计演示的内容，以及使用户可以从不同观点观察整个设计场景。

设计动画序列有很多方法可供选择，设计动画序列受限制的不是方法，是你的想像力。一些常用的技术非常简单易用。我们将给出一些例子或参考前面介绍过的例子来简要地介绍这些技术。要注意，我们介绍的技术和例子比游戏和大部分的商业娱乐动画简单得多，本章
400 的重点在于介绍动画的概念。

11.2.1 过程动画

我们经常讨论用参数来定义模型，从通过参数控制场景模型的角度入手研究动画。程序中这些变量（可以操作的参数）控制了物体的位置、光源的位置和属性、物体的形状及其相互关系、颜色、纹理坐标或者其他重要的属性。程序可以随着时间的推进改变这些参数的值，观察者随着参数的改变感受到运动的图像。这就使你注意模型的这些重要特征，并关注运用这些重要特征与观察者进行交流。这种动画是由计算过程驱动的，称为过程动画，可以通过基于时间的计算，显式地控制这些参数。应用这种方法可以很容易地通过少数几个参数控制，实现简单模型的动画（这里“少数”的含义是由计算机系统和具体动画序列的难易程度来决定的）。

本书前几章介绍的科学应用中的几个动画例子都属于过程动画技术。它们都是根据某一科学原理计算随时间改变的对象位置或其他属性，并按其变化显示整个图像序列。本章开始时介绍的瀑布的例子也属于过程动画范畴。这种直接通过计算得到动画序列每一帧的重要参数的方法是过程动画的特征，过程动画技术也可用来生成复杂的动画序列，只要你能通过计算获得全部动画参数。

11.2.2 场景图中的动画

对场景建模以及设计动画时，请回想场景图描述场景的四个部分，每一部分都有自己随

时间变化的方法。大多数变化都涉及参数，也是过程控制的参数，虽然这些参数可由用户输入，或者可由定时事件或特定事件驱动的触发器来改变场景。下面给出场景图的组成部分及其变化方法：

- 场景的几何模型：可以用参数来定义场景的几何模型。比如在第9章中介绍的参数曲面方程$z = \cos(x^2 + y^2 + t)$，t可以代表时间，随着t的改变，场景的几何将会发生变化。
- 场景的变换：可以通过参数来定义场景中物体的旋转、平移和缩放，比如热传导的例子中，棒的旋转就可以通过参数来控制，也可以通过参数控制物体的位置和朝向。
- 场景中物体的外观属性，比如颜色或者纹理：可以根据需要来改变颜色、纹理或者其他外观属性。比如，曲面可以有一个α颜色通道$(1-r)$，使t为0时刻物体完全不透明，t为1时刻物体完全透明。随着参数的改变，曲面从完全不透明变为完全透明，这样让用户可以透过曲面不同的透明度看到里面的物体。
- 场景的视图：可以根据参数来改变视点位置、视线方向、向上方向，或者其他观察参数。随着时间控制这些参数的改变可以用不同的方法观察场景，或根据需要观察场景的不同部分。

这些都是很浅显易懂的建模应用程序，读者学习到本章应该具有一定的建模经验，可以很容易地设计随时间变化的模型。 401

11.2.3 插值动画

如果脱离计算参数的思路，从用参数定义模型的角度考虑动画问题，可以得出一种“参数动画”，它通过控制模型的参数集来定义整个场景，比如，把模型定义为一个向量$P = <a, b, c, \ldots, n>$。用$P_M = <a_M, b_M, c_M, \ldots, n_M>$来表示某特定帧M的参数向量。在计算某一动画序列的一段连续帧的时候，比如从第K帧到第L帧，我们必须计算在下面两点之间的所有参数向量：

$$P_K = <a_K, b_K, c_K, \ldots, n_K> \text{和} P_L = <a_L, b_L, c_L, \ldots, n_L>$$

这两帧的参数向量值可以按建模或者交流的要求选择合适的插值方法求得。这个插值方法可以是线性的，也可以是非线性的。本章稍后会有介绍。这种生成动画的方法叫做插值动画。使用这种方法时，必须定义两个模型，然后通过插值将第一个模型的几何信息转换成第二个模型的几何信息。本章稍后面会有一个例子阐述这种方法，但这种方法在科研领域的应用并没有过程动画那么普及。

插值动画的一个应用就是变形，也就是从一个物体（一张脸、一个动物或者一辆汽车）开始，最后变形为另一个物体，目的就是强调一个物体到另一个物体的变化。为了在两个图像中间生成一系列的变形图像，需要在起始和终止图像上插入一系列的关键点，这样，问题就变成了采用插值技术使一幅图像中的关键点移动到另一幅图像的关键点，在两幅图像之间生成一系列“过渡图像”，然后将插值出的关键点与过渡图像的纹理一一映射。这种从一个图像到另一个图像的变化非常复杂，因为两个图像上的每一个关键点都有与之对应的纹理坐标点，需要格外注意选择合适的起始和终止纹理。举例来说，如果将一张脸变形成另一张脸，人脸的关键特征比如眼睛、鼻子和嘴的轮廓必须恰当地标注并一一对应。如果对汽车进行变形，比如车灯、车轮、挡风玻璃和尾灯等关键特征也必须恰当地标注并一一对应。这需要对一幅或者两幅纹理作扭曲处理，才能使几何信息和纹理信息相对应，具体方法在[WOL]中有介绍。变形是从一幅图像到另一幅图像的转换过程，也可以看作是一种动画，这是一种非常专业的操作，这里不再作进一步讨论。

一般来说，插值可以是简单的线性插值，也可以是复杂的插值。第13章将完整地介绍插

值技术，读者可以采用那里介绍的任何一种插值技术。第一种插值的方法就是对参数做线性插值，假设插值的第一帧和最后一帧为K和L，两个关键帧中间一共有$C = L - K$帧，对于K和L之间的整数i和参数p，有$p_i = (ip_K + (C - i)p_L)/C$，如果令$t = i/C$，那么参数$p_i = (tp_K + (1 - t)p_L)$。这是一个熟悉的线性插值公式。这种计算方法非常简单，通过对参数的平滑控制，可以转化为对关键帧之间动画的平滑控制。

但是，要做到帧与帧连续运动，比上面简单的线性插值复杂得多。实际上，不仅要实现两个特定帧之间的平滑运动，而且需要该帧之前的运动能和该帧之后的运动平滑混合过渡。这里介绍的线性插值方法并不能实现这种平滑效果，在特定帧处可能出现动作的急剧变化。
402 这样，就需要采用一种更一般的插值方法，叫做动作的渐进和渐出。一种实现渐进渐出的方法就是在第一帧开始的时候运动得比较慢，然后中间运动得比较快，在插值的最后阶段再回到慢速运动，在到达结束帧的时候停止参数的变化（这样也停止了图像上的运动）。在图11-2中，可以比较左面的简单线性插值和右面的逐渐启动/逐渐减速插值。右图显示了一个用$s(t) = 0.5(1 - \cos(t\pi))$表示的正弦曲线，这里$t$表示插值的单位参数，也可以理解为$t$是时间或者是帧数，对照前面$p_i = (tp_K + (1 - t)p_L)$的线性插值公式，可以采用$s(t)$代替$t$，得到非线性插值公式$p_i = (s(t)p_K + (1 - s(t))p_L)$，这里$t$和前面一样为均匀间隔。这种改变在编程上没有很大的变化，但是实际效果就相差很明显。令$s(t) = t$，即线性插值。

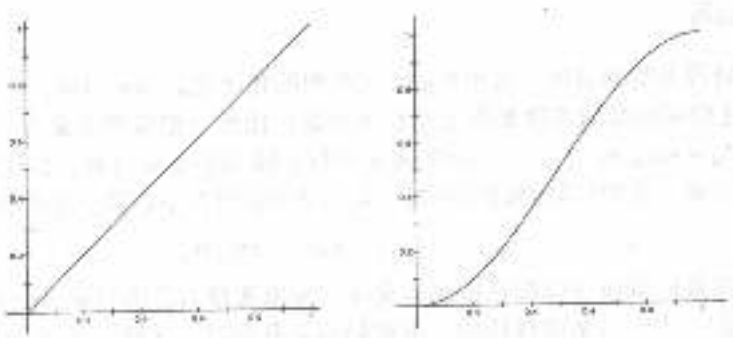


图11-2 两种插值曲线：线性曲线（左）和正弦曲线（右）

实际上，这种动作的渐进渐出并不足以实现逼真的动画。为了强调物体的运动状态，需要回溯到物体启动之前的运动状态，以及考虑物体到达目的地后直到停止前的运动状态。可以通过改变插值函数来得到上面的效果。在起点位置刚开始插值时插值曲线稍稍向负方向过渡一点，然后在到达终点前插值曲线稍稍超过1一点。虽然起点和终点的位置以及插值的整体趋势没有变化，这种插值产生的运动的确使动画效果更有美术感。然而，我们无法明确告诉你何时应该采用这一技术，只有当这项技术能起作用时，才能采用它。只有当你试用了大量线性插值和渐进插值实例后，才能理解这项技术的适用场合。

对特定参数采用正弦曲线插值可以达到缓慢运动的效果，这里仍然有一个问题，那就是参数只能控制下一关键帧之前的动作或效果，然而过了这个关键帧就会采用下一个完全不同的动作。换句话说，运动在跨越特定关键帧的时候并不是非常平滑。为了实现关键帧前后的平滑运动，要学习使用更复杂的插值方法，在第13章会详细介绍。就像我们用插入控制点来生成曲线，
403 我们也可以采用Catmull-Rom样条来插值出一系列的点，生成平滑的插值曲线来逼近原始的点。

11.2.4 基于帧的动画

可能最简单的实现动画的方法就是通过简单的参数来控制整个场景的变化，然后每次更

新参数，生成一帧新的动画。可以用时间作为参数，将动画看成随时间变化的模型。这可能是一个比较自然的方法来解决科学问题，时间在建模过程中起到了关键的作用。有相当多的科学计算是和单位时间内的变化相关的。这种建模方式需要做积分来表示这种变化。如果你清楚生成场景所需的时间，甚至可以根据这一生成时间调整帧与帧之间的时间，这样观察者就能以一个比较恒定的速度实时地观察变化的场景。

另一个参数就是帧数，也就是计算生成的特定图像在一段动画序列中的序号。如果你的动画是记录在数字或者模拟的硬拷贝媒介上，再以规定的速度播放，这样就把帧数转换成时间参数。参数的不同名字代表了不同的意义，这是因为你并不在意要多长时间生成一帧，而是关心这一帧在动画序列中的位置。

引入帧的概念，可以通过创建一系列关键帧来设计一段动画。关键帧就是在特定时间显示在屏幕上的一组特定图像。通过这种方式生成的动画称为关键帧动画。当创建一段关键帧动画时，需要指定哪些帧作为关键帧，然后生成其他的非关键帧图像，使动画能在关键帧之间流畅的播放。关键帧是由帧数来指定。

在卡通动画中，关键帧通常需要完整的设计和绘制的图像，然后通过一种处理过程来生成其他帧的图像，这种处理过程叫做“渐变”。在关键帧之间生成中间帧。在传统的动画中，动画师对关键帧之间的每帧动画都重新绘制出运动的部分。但是现在我们用计算机来生成动画图像，就必须用模型取代手工的绘制。如果采用关键帧方法，需要定义各种各样的图像参数，然后根据这些参数创建中间帧。如果直接从关键帧中插值出参数，那么关键帧动画就和前面介绍过的插值动画一样了。

11.2.5 一个插值例子

我们的重点在于简单图像的动画，而不是商业娱乐动画，因此，用插值来做简单的小动画。举个例子，假设有一对模型代表相似的物体，用相似的定义方法来对它们建模，包括几何信息和纹理图，然后平滑地将其中的一个转换成另一个。这里例子中的物体就是有不同几何信息和不同表面纹理的两面墙。我们先从两幅纹理图的插值开始，进行几何信息的插值。最后我们会在习题中让读者将这两个技术结合到一起。

为了完成在第一和第二个场景的动画插值，首先在场景中对第一和第二两个墙面的纹理图进行插值，并为插值场景创建新的纹理图。通过简单的像素颜色的线性混合生成插值纹理图，对数组中的每一个像素中的颜色都用线性插值来得到新的插值颜色。尽管我们需要时间来生成每一个插值图，还需要重新做纹理映射，然而这种方法非常简单直观。用这种简单方法插值出中间纹理图，如图11-3所示。

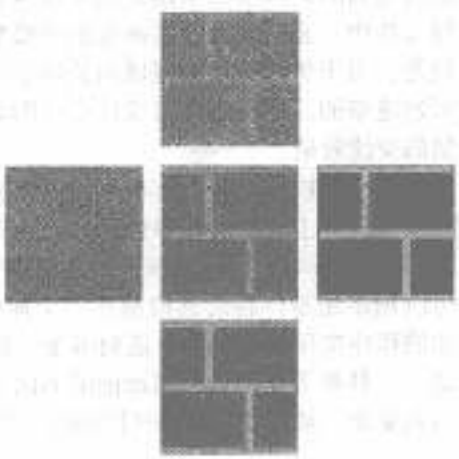


图11-3 砖纹理（左）、转墙纹理（右），以及根据它们生成的含有右图25%比例（上）、50%比例（中）、75%比例（下）的三个插值纹理

现在要考虑墙的几何信息，以及怎样对它们进行插值。第一个墙面是一个简单的矩形，第二个墙面是一个向上凸起的弯曲表面。在这个例子中，这两面墙都是简单的表面，不考虑墙面的厚度。第一个墙面的几何信息由一系列垂直的矩形表示，每一个矩形宽度都是一个纹理单位，高度为数个纹理单位。这样便于编程，也容易确定纹理坐标。第二面墙采用类似的方法用垂直的四边形建模。

404

405

每个四边形宽度都是一个纹理单位。由于墙面是向上凸起，确定上边沿的纹理坐标不是非常方便，如果继续用纹理单位作为度量基准，就可以用纹理单位的几分之几来表示凸起的高度，并作为上边沿的纹理点。对于这两堵墙，将模型分割成垂直的部分，结合前面的纹理渐变进行绘制。原始墙面和插值墙面的几何信息在下面的图11-4中给出。



图11-4 将要插值的两个墙面（左和右）以及各占50%比例插值的结果（中）

为了将前面两部分工作连接起来，假设矩形的墙面采用图11-3中左侧的瓷砖纹理图，曲面墙面采用图11-3右侧的砖墙纹理图。为了进一步简化任务，假设一张纹理充满各自的整个墙面。我们将这部分内容在习题2中留给读者来完成纹理和几何信息的组合插值。

11.3 动画中的一些问题

当创建动画的时候，需要考虑一些与静态（甚至交互的）表示不一样的问题，包括帧速率以及时间走样等问题。

11.3.1 帧速率

实时生成动画的关键就是帧速率，即在动画序列中生成新图像的速度。尽管生成速度越慢，图像绘制结果越好，但通常需要每秒24－30帧的速度才能使动画看起来流畅。就像我们前面提到的，高度细节模型动画可以通过预计算方法将动画绘制好保存在数字或者模拟的视频文件中。直接实时绘制高度细节模型动画的帧速率比这种方法低很多。这里还有一个不同就是：由于实时环境中视域内的场景在不停地变化，帧速率对于实时生成图像来说不可能是恒定速率的。这些问题对设计自己的动画提出了挑战，从而需要考虑动画是否能够实现所期望的交流效果。

动画的帧速率的数值在不同速度的机器上相差很大。程序（以及它的动画输出）在更快更新的机器上就要运行得快一些。当使用idle事件来生成动画的时候，上述情况一定会发生。而采用timer事件生成动画会得到比较平稳的帧速率，但仍然不可预测。要实现最佳的效果，
406 可以用系统事件将动画控制在一个理想帧速率而不至于生成太快。如果要支持不同的系统，你的程序实现起来可能会遇到困难，我们将在后面OpenGL的讨论中介绍基于GLUT的解决方法。一种基于预计算生成动画的方法可以确保制作自定义动画视频硬拷贝时，不管每一帧有多么复杂，帧速率是精确可控制的。读者可以参考第15章的内容详细了解硬拷贝技术。

11.3.2 时间走样

在制作动画的时候，需要生成一系列的图像来展示场景在不同时刻的状态。当顺序地观察这一系列的图像时，可能会看到一些奇怪的，并不是你所期望的现象。有些现象是图形系统造成的，比如，显示一个非常小的物体时，它显示的大小随时间不断地改变，物体占据的像素数一会儿多，一会儿少。这是一个屏幕走样问题，可以用部分覆盖像素的反走样技术来消除这个问题。但有些问题是制作动画过程固有的，不能完全消除。因此，必须对播放的图

像序列会出现其他现象，导致动画效果背离你的本意的可能性有清醒的认识。

我们看一个例子。假设有一个如图11-5的物体（图中叶片的夹角大概是45度），随着时间的推进旋转。如果缓慢地旋转它，我们的眼睛会自然地跟着每一个叶片的运动，因为叶片在下一帧里的位置往往出现在离上一帧位置不远的地方。如果你希望叶片做顺时针旋转，就要让叶片每帧旋转的角度要小于22.5度，或者说是叶片夹角的一半。当旋转角度远比这个角度小的时候，动画效果就会更好。但是，如果旋转的速度快一些，比如每帧旋转40度，这样下一帧一叶片的位置仅和上一帧的该叶片顺时针下一个位置相差5度（承接这句话确信你理解其中的意思）。这样，你的眼睛将每一个叶片和前一帧图像中的下一个叶片联系起来，这样，叶片似乎是在作逆时针旋转！在电影里或者用闪光灯照射正在旋转的物体可能出现类似的事与愿违的效果。我们要认识到出现此类效果的可能性，并学会在程序中控制这种效果。本书附带的资料中包括了这段动画的代码，开始时动画中叶片运动得很慢，每按下一个键，运动都会加速，随着不断地增加每帧叶片转动的角度，你会看到前面描述的那种随着角度增加，视觉上叶片做反向旋转的效果。

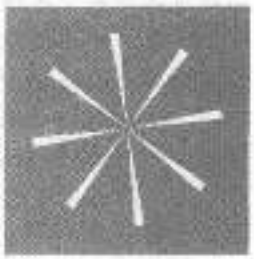


图11-5 旋转的物体 407

这类问题是由时间走样造成的，或者说是在时间域中对时间序列作离散采样造成的，我们可以把它和几何走样做个类比，几何走样是在空间域对图像作离散采样造成的，我们看到几何走样是怎样将一条平直的直线段变成了一条阶梯状线段。时间走样则会造成车轮叶片运动反向或者运动冻结的现象，或者在夜间用闪光灯从下方照射飞机螺旋桨叶片时也会出现和车轮转动类似的走样效果。

时间走样现象是现实世界中可以看到的现象，所以在需要的时候要能够表现这种走样效果。如果你不想让自己的动画产生走样效果，可以采用时间反走样技术。一种方法就是对场景做运动模糊，后面我们会介绍它。其他方法包括用小的时间片来避免走样（在前面的例子中，小的时间片就是减少相邻帧叶片的旋转角度）或者采用没有明显时间走样特征的模型。

11.3.3 动画制作

控制动画的制作过程就好比导演执导电影的过程。电影积累了非常丰富的技术来表现各种不同的信息，这些电影手法对我们开发科学研究中的动画很有帮助。如果读者想深入学习这方面的内容，可以参考职业动画师的技术（比如可以参考[POC]）。动画方面的参考书会告诉你很多很多的方法，帮助你改进动画的设计和制作。开始时，你可能只保留一个固定的视点和几个固定的光源使你的动画制作变得简单些，然后增加一些模型的控制功能，使部分模型可以随时间运动。然而，随着导演采用运动摄像、悬杆式摄像机和手持移动摄像机等各种新技术，电影领域发现了用运动摄像机在运动场景中拍摄的重大价值。你也会发现把运动视点和运动对象结合起来会产生最佳效果，尝试各种不同的组合来找到适合你的动画的最佳"拍摄"方式。

11.4 动画和视觉交流

现代图形API支持运动的模拟能力，是一个强大的交流工具，不管动作是通过动画产生的或者是交互产生的，它可以让你讲述一段随时间改变的故事，并使每一个观众都有独特的体会。在自然科学或其他研究领域经常要考虑动态情形，这非常适合用一个随时间变化的动画来演示。有些现象不依靠动作是发现不了的，比如在星空中观测某个星体，只有星体的不同

408 运动方式才能将其和其他星体区分开来。动作本身也包含某种信息。一个典型的例子就是芝加哥伊利诺伊大学电子可视化实验室的Dan Sandin创建的将两种运动显示在一个屏幕上，然后通过区分某个区域内部和外部的运动进而识别这个区域。这里不可能给出运动的动画，在本书附带的资料中读者可以找到例子的源代码，这个代码也是本章后面一些习题的基础。在图11-6中我们看到动画的一帧（左），但是没有运动，你不可能分辨区域的边界。我们用虚线给出了边界在哪里，这些虚线点并没有在原始动画中画出（右）。

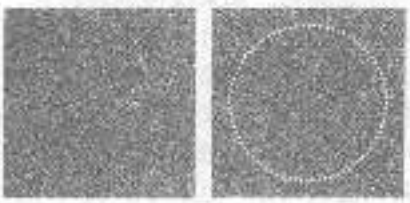


图11-6 随机噪声的正方形区域（左）其中包含一个通过运动来定义的区域。这个区域用虚线圆给出（右）

创建动画时，应该考虑清楚哪些物体是运动的，运动的节奏怎样。有时需要让场景中的大部分物体保持静止，只让部分物体运动，使动画突出运动物体引起的场景中对象关系的变化。有时需要让所有的物体都参与运动，这样就可以对场景的运动有一个整体评价。如果通过动画来表达这种运动，建模时就需要一个时间参数，这样就可以简单地通过更新时间来更新模型，然后对它重新绘制。这种情况下，需要控制模型中所有对象都按照场景统一的时间节奏运动。如果通过交互引入运动，这就更复杂些，因为要允许观察者移动场景中指定的部分物体，但你需要对模型创建一个一致的行为模型。

现代计算机技术本身使创作动画成了一个很有意思的挑战。使用更快的系统和不断增强的图形硬件使绘制场景的时间不断减少，动画帧速率越来越快。我们不希望让这种计算机飞速发展的趋势有所减速，或者停止。同时，我们也在冒创建实时联机动画的风险，因为在一些更新的系统上动画的速率会快得出奇，以至于使人无法理解。你可能希望把动画帧速率控制在一个指定的速率。这可以通过系统敏感的工具来达到稳定的帧速率。这样的功能可以通过系统函数
409 pause(N)（或者类似系统函数或者API）来实现。系统函数pause能让你的程序进入空闲状态达N毫秒。在本章后面我们会给出一个例子。当然，如果运动是由人机交互方式引入并定义的，就不存在这个问题，因为人的动作总是系统中最慢的，观察者能够控制自己的帧速率。

有时对某些特定的观众，比如面对公众或者面对投资人，你希望生成非常高质量的动画，这时需要考虑演示动画其他方面的因素。其中需要考虑的因素之一就是声音。你不可能看到公众场合播放的动画是没有声音的动画。现在的图形API还不支持声音，但我们希望一些辅助的API将增加对声音的支持功能，这样就可以在动画中加入声音。这种声音可以是一种录制的画外音，或者是强调动画动作的音响效果，或者是音乐声道，或者是三者的混合。如果采用视频硬拷贝来存放要演示的动画，可以在制作视频硬拷贝时就加入声音。如果仅提供动画的联机版本，可以在以后再考虑加入声音。

11.5 在静止帧中表示运动信息

当你向观众传达几何对象的运动信息时，可能需要用动画来表达。但是，当观众不仅想看到运动部分，而且希望看到它们是如何运动的，就需要在当前帧中保留一些它们在前面数帧中位置的信息。有两种方法可以实现运动效果，即运动轨迹法和运动模糊法。

11.5.1 运动轨迹法

给出运动轨迹的一个常用的方法就是在显示物体的同时，给出物体以前运动位置的某种痕迹。可以通过生成一组线条或类似的几何图元来显示这种轨迹。轨迹应该有一个长度限制

(除非你真的需要给出物体运动的整个历史轨迹，这是另一种完全不同的可视化信息)以及可以用降低的α值来给出物体前面数帧的位置。图11-7中给出两个运动轨迹的例子。左图用一串连接的圆柱来代表圆柱体运动的轨迹，圆柱的颜色和物体的颜色一致，采用降低的α值来表示不同位置，而右图给出的是通过直线段表示粒子随机运动的前后位置。 410

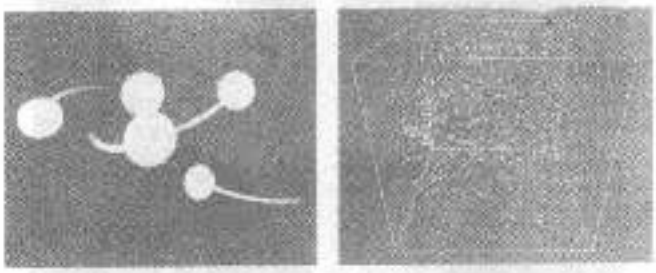


图11-7 两种运动物体的轨迹。参见彩图

11.5.2 运动模糊法

在体育运动或者高速动作摄影的场合，我们更习惯于看到图像中运动物体是模糊的。这就是运动模糊现象，用这种方法来表示运动时，就是对运动的物体生成模糊效果的图像，而对静止不动的物体生成边界清晰的图像。运动得越快，图像越模糊。在图11-8中我们看到一个平板(绿色)上有两个连接的支杆(红色和蓝色)，两个支杆中间用一个固定的横杆连接(白色)。当其中的一个支杆移动时，另一个支杆和平板会跟着移动，而横杆保持不动。图中给出了这个运动模糊的图像。

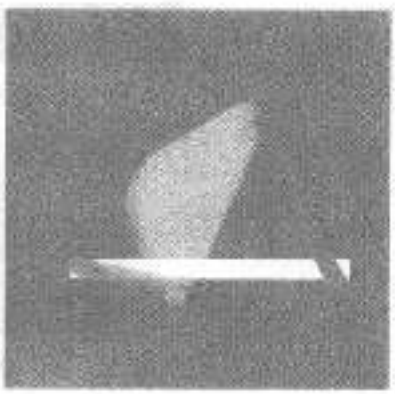


图11-8 一个运动的机械装置，其中一个部件固定，其他的部分带有运动模糊效果，参见彩图

图11-8中显示的运动模糊效果有几种不同的实现方法，标准方法是把存储在累积缓存中不同时刻生成的图像合成在一起，形成一帧合成图像。这种合成技术让我们可以同时看到场景不同时刻的图像。这样，运动物体会出现在不同位置上，因此它们看起来有些模糊。静止的物体仍然在同一个位置保持不动，它们的边界就非常清晰。很多图形API都提供累积缓存工具，在本章后面的OpenGL部分会详细介绍如何使用这些API来实现这种图像合成技术。 411

11.6 一些有趣的观看动画的设备

通过连续播放静止图像可以观察到运动的图像，从我们意识这一点开始，出现了各种各样的设备来帮助我们观察到运动的效果。这些设备都采用事先绘制好的不同题材的图像(通常是照片或者手绘)，而且需要手工操作。我们可以在博物馆和古玩店中惊奇地发现这些历史上观看动画的设备，正如看到我们在第1章中介绍立体感投影仪那样。这些设备还包括西洋镜和翻动型动画书。西洋镜是一个鼓桶状的装置，在桶的边沿有数个可以观察的小槽以及有一条连续的图像带，如图11-9所示。西洋镜也可以在桶的底部绘制一圈连续的图像。

西洋镜的原理是：眼睛透过观察槽只能看到一定宽度的图像，当一个完整的图像出现在这个区域时，眼睛会抓住这幅图像。当图像继续旋转的时候，眼睛会抓到下一个图像，如此

连续下去。由于眼睛注视图像的序列，就会将他们混合成一个运动图像。一条西洋镜的图像带在图11-10中给出，其中几幅细节图像在图11-11中给出。西洋镜的机构以及图像的大小和数量是决定能否将分离的连续图像混合成动画效果的重要因素。

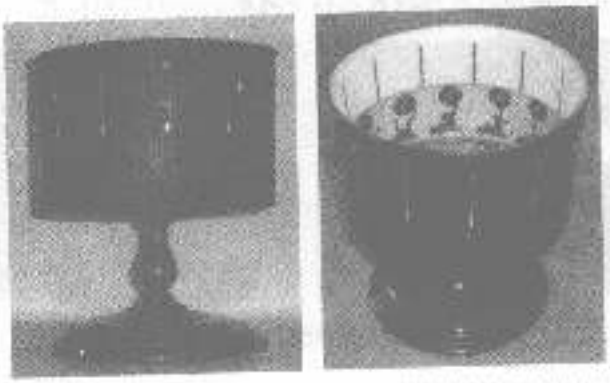


图11-9 一个古董西洋镜的两张照片，整个设备的构造（左）以及观察槽和观察图像之间的关系（右）



412 图11-10 一条绘有非常简单的连续图像的古董西洋镜纸条



图11-11 纸条中的三幅图像，展示相邻图像的处理

制造一个西洋镜就更复杂。图11-9中的西洋镜有13个观察槽，每一个观察槽有3/16英寸宽，整个金属鼓桶的内圆周有36英寸长，同样，图像带也有36英寸长，上面有13幅分离的图像，每一幅图像不到3英寸宽还要包括两幅图像之间的空白区域。整个鼓桶通过一个简单的中心轴由手动旋转，然后观察者透过观察槽观看动画效果，每一个时刻观察者只能看到条带上的一幅图象。随着鼓以一个合适的速度旋转，图像在观察者的视野中混合，形成动画。如果动画本身是循环的，效果会更好，这样条带上动画的结束刚好接上这个动画的开始，当然我们没有必要刻意这样做，实际上大多数手绘的动画，比如图11-10以及图11-11所示的动画序列都是在条带的一端开始，在另一端结束。

原始的西洋镜鼓桶的观察槽是在金属鼓侧表面挖出的槽。但是我们发现有些西洋镜是从鼓顶部直接向下开槽。这样的开槽方法看起来很简单，但它的鼓体不如侧边挖槽结实，如果鼓桶是金属制造的，锋利的槽角很危险。如果这样构造西洋镜，我们建议在鼓的上部用某种条带来保护手指，以免被鼓边和槽角刮伤。

你可能采用不同数量的图像在不同半径的鼓桶上制作西洋镜，但是需要考虑观察者看到的图像大小以及鼓的直径等因素。如果在西洋镜中放置更多的图像，就要将鼓桶做得更大，但是，这样的话观察者透过槽就会看到更多的图像。我们看过很多西洋镜，它们的图像条带

都只有12、13或者14个图像，所以，我们建议读者在图像条带中采用这些数目的图像。当然你也可以尝试更多数目的图像。制作不同的西洋镜鼓桶是一个不小的工作量！

翻动型动画书的构造更简单，它不像西洋镜那样复杂，也没有那么多物理限制。翻动型动画书是由一叠纸张构成的，每一张纸上有一个图像，然后装订成书。制作一本翻动型动画书非常简单，只需要把序列图像打印到纸页上，一页一幅，在一边留出足够大的装订空白（通常是左边或者上边）。然后将纸页按观察的顺序整理好，最后在空白处将纸页本装订起来（比如用一个或数个钉书钉）。观看动画的时候，就可以一只手拿住装订位置的边沿，另一只手以能够瞥见图像的速度翻动书的每一页。如果翻页速度控制合适，我们就可以观察到动画效果了。翻动型动画书上的动画可以有非常多的图像，不像西洋镜图像数目有限制。采用较重的纸张，翻动型动画书做成25页到100页是可行的。制作翻动型动画书的最大技巧就在于要用稍微硬一点的纸张来制作，这样每一页图像就可以在手指翻页的时候能有一定的停留，同样也要考虑纸张的大小要合适。翻动型动画书在动画发展早期占有重要地位。一些早期的动画机器就是握在观察者手中的翻动型动画书，只不过是用观察者身旁的旋转盘来翻页而已。

413

11.7 建议

设计动画和设计单一场景截然不同，它需要更多的经验来达到满意的效果。在专业动画中用到的技术就是故事板——整个动画的规划书，里面记录什么时候做什么事情，以及每一个拍摄镜头要告诉观众什么样的信息。

11.8 OpenGL的动画例子

下面介绍一些动画的例子，包括一些适用代码。这些例子应用了前面介绍的一些技术。在例子代码中可以看到，动画常常是在idle事件的回调函数中，通过变化模型的参数来控制。这些例子展示了用回调函数来控制模型、场景以及显示的方方面面。通过扩展这些例子和不断的实验，学会掌握哪些东西是可以控制的，以及怎样控制它们，使动画成为一个流畅的交流工具。

11.8.1 在模型中移动物体

动画和动作息息相关，一种实现动画的方法就是移动模型中的物体，比方说，可以采用一种数学的方法来定义立方体的位置，控制它在空间中移动。在这个非常简单的例子中，立方体从初始位置原点平移到新的位置cubex，cubey，cubez。这是一组以时间aTime为参数的三角函数，控制移动的位置，时间参数以固定的deltaTime来不断更新，位置是由计算得出的，所以这是一个过程动画。最后的效果就是立方体在各个方向上以不同的方式运动，这样的运动看起来很随机，但是实际上它并不是随机运动。这种运动控制可以通过下面的idle事件回调函数animate()来实现：

```
void animate(void)
{
    #define deltaTime 0.05
//  通过某个时间行为的继续方法定义立方体的位置
    aTime += deltaTime; if (aTime > 2.0*PI) aTime -= 2.0*PI;
    cubex = sin(2.0*aTime);
    cubey = cos(3.0*aTime);
    cubez = cos(aTime);
    glutPostRedisplay();
}
```

414

这个函数设置了三个位置变量cubex、cubey和cubez，将这三个值作为display()函数的参数做平移变换，改变立方体在空间的位置。也可以用同样的方法改变其他的参数，比如方向、大小或者模型的其他属性，实现其他变换。

11.8.2 控制动画的时间

迄今为止，在我们看到的大部分动画程序中，都采用idle事件来调用函数（一般命名为animate()，读者可以用它在本书附带的代码中搜索）来更新模型以及发送重显示指令，但是每一次生成完整的一帧后都会发出idle事件，所以用户的计算机速度越快，或者程序生成每一帧速度越快，下一帧的生成就会越早。这样，如果你的动画中有些帧很简单，有些帧很复杂，那么idle事件生成每帧的时间会很不均匀，当你在更快（或者更慢）的机器上运行程序的时候，动画会以完全不同的速度播放，观察者会对最后的效果感到疑惑，甚至厌烦。所以管理好动画的时间非常重要。

一种解决的方法就是用timer事件而不是idle事件。就像我们在第7章看到的那样，只有从回调函数注册开始，时间前进到设定的毫秒数，timer事件才会被触发。但是采用这样的方法系统代价太高，程序需要连续的重新注册回调函数（在前面的例子中，timer回调函数每执行一次，就要注册一次）它并不能让你很好地控制时间，但它确实比用idle事件控制时间好得多。

还有另外一种控制帧速率的方法，就是根据直接获取的系统时钟来控制。GLUT中有函数glutGet(state)，可以返回当前系统状态变量值。如果用GLUT_ELAPSED_TIME作为参数就会得到当前的系统时钟。所以，glutGet(GLUT_ELAPSED_TIME)就会得到从glutInit()开始或者从第一次glutGet(GLUT_ELAPSED_TIME)开始过去了多少毫秒数。这样当生成一帧（调用glutPostRedisplay()）的时候，就可以通过调用这个函数来得到时间。当结束下一帧计算并准备生成时检查这个时间。如果已经过去足够的时间，就可以生成redisplay命令；如果过去的时间还不够长，需要继续等待（可采用旋转等待或者调用系统sleep事件）直到过去了足够的时间，然后再产生redisplay命令。这样不管你的程序运行在多么快的机器上，它保证了你的动画不至于跑得过快。

11.8.3 移动模型的部件

就像我们可以移动整个物体一样，可以移动层次结构模型的单个部件。正如在建模时可以用变量来改变部件的相对位置、相对角度，或者相对大小，最后在合适的回调函数中对这些值做改动。甚至可以做更复杂的变化，比如改变物体的颜色或者透明度来呈现给观察者特定的效果。在第2章讨论层次建模时，我们设计了一个兔子头的场景图，如图11-12所示。这个场景图包括变换节点，其中某些节点中有一个参数*t*控制兔子的耳朵。在下面的代码中我们不断地增加参数*t*的值，用这个参数来设置旋转角度，达到摆动兔子耳朵的动画效果。我们注意到场景图应该包括把耳朵安放到头上的变换，这里没有给出这些变换的细节，这段代码包含了这一功能。

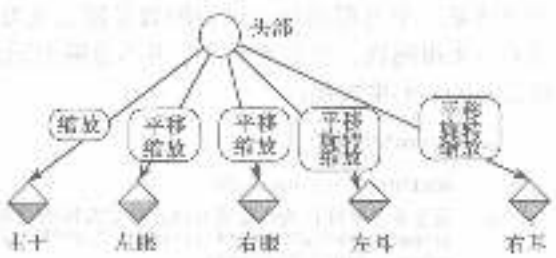


图11-12 兔子头的场景图表示

兔子一只耳朵的定义代码在下面给出。这个过程使用了很多变换，但是只有两个变换是与旋转耳朵有关的，其中通过参数*wiggle*来改变帧与帧之间耳朵的位置，其他变换都是固

定不变的，即不会随着图像改变而变换。

```
glPushMatrix();
    //左耳建模
    glColor3f(1.0, 0.6, 0.6);           // 粉红色耳朵
    glRotatef(-10.0*wiggle, 0.0, 0.0, 1.0);
    glTranslatef(-1.0, -1.0, 1.0);
    glRotatef(-45.0, 1.0, 0.0, 0.0);
    glTranslatef( 0.5, 0.0, 0.0);       // 开始
    glRotatef(-10.0*wiggle, 0.0, 0.0, 1.0);
    glTranslatef(-0.5, 0.0, 0.0);       // 结束
    glScalef(0.5, 2.0, 0.5);
    myQuad = gluNewQuadric();
    gluSphere(myQuad, 1.0, 10, 10);
glPopMatrix();
```

下面给出的idle事件回调函数animate()告诉我们怎样操纵*wiggle*参数，来不断改变耳朵旋转角度。

```
void animate(void)
{
#define twopi 6.28318

    t += 0.1;
    if (t > twopi) t -= twopi;
    wiggle = cos(t);
    glutPostRedisplay();
}
```

11.8.4 移动视点或模型的观察标架

另一种动画技术就是在场景中定义可控的视点移动，使观察者能看到模型所有部件，以 416
及能从特定的位置对特定的部件进行观察。这种运动可以完全由脚本控制，或者由用户控制，尽管后者实现起来会更困难。与这个技术相关的第一个例子中，视点从立方体的前方移动到后方，然而视点方向始终朝向立方体的中心。另一种更复杂（也更有趣）的视点运动路径是用求值函数根据设定的控制点算出。视点的z轴坐标在animate()函数定义的两个端点之间移动，然后gluLookAt(...)函数会用到这个视点的位置，采用样条插值移动视点位置问题将作为本章后面的实验内容。

```
void display(void)
{
//  将视点作为变量，移动视点……
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(ep.x, ep.y, ep.z, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
...
}

void animate(void)
{
    GLfloat numsteps = 100.0, direction[3] = {0.0, 0.0, -20.0};

    if (ep.z < -10.0) whichway = 1.0;
    if (ep.z > 10.0) whichway = -1.0;
    ep.z += whichway*direction[2]/numsteps;
    glutPostRedisplay();
}
```

更复杂的例子就是用13章中介绍的插值技术来确定视点在场景中的移动。你可以定义数个点来指定视点经过的位置，然后在中间添加额外的控制点来指定视点将经过这些控制点。这些额外的点的选取将允许视点在移动路径一部分向另一部分衔接的时候平滑的过渡，参阅第13章。为了实现这种效果，Catmull-Rom样条曲线是一个很好的插值技术，因为它定义了一条经过控制点的比较合适的曲线。关于样条曲线，我们将在第13章中具体介绍。

为了让视点在空间内移动，需要通过插值参数t求得$x(t)$、$y(t)$、$z(t)$函数的值来决定视点的位置。你需要确定观察方向，这可以通过曲线上的前后两点来求得，因为这是你前进方向的很好的近似。你还需要确定向上方向，通常我们会采用固定的向上方向，也就是你所在场景逻辑的向上方向。图11-13中给出了一个简单的空间模型（可能是四个高塔的建筑），图中还给出了控制点和根据它们得出的视点移动曲线。读者可以在本章结束的练习4中完成视点的移动过程。

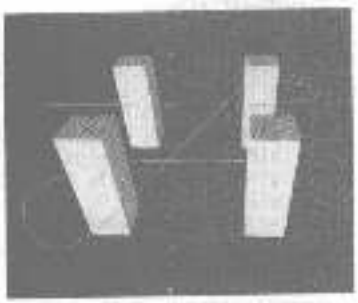


图11-13 跟踪穿越空间的路径，图中显示了控制点的位置

视点在空间中漫游，不仅需要控制视点的位置，还要控制整个视域环境。在OpenGL术语中就是
417 gluLookAt(...)函数的整个参数表。在创建合适的移动视点时，不仅要考虑视点本身，还要考虑观察参考点和向上方向向量。进一步说，当你在空间中移动，会发现有时你靠近物体，有时远离物体。这就意味着你需要使用第12章介绍的细节层次（LOD）技术，来控制这些物体呈现给观察者的图像，同时还能使帧速率尽可能高。使用细节层次技术需要大量的工作，但你可以从非常容易的模型起步。

另一个控制运动的方法就是给定一个初始位置和一个在特定时间生效的速度向量的集合。每一个速度都是一个三维的向量，这样就可以将速度视为点，然后用样条曲线插值这些点，生成一个速度向量集合。从初始位置开始运动以后，不断地将曲线定义的这一系列速度向量作用到物体上。

11.8.5 场景的纹理插值

我们在前面讨论图11-3的纹理插值时，只是简单地对两个纹理基本数组作颜色的线性组合生成插值纹理。下面的代码段实现了这种方法，其中假设tex1和tex2是用任意方式生成的纹理，texImage是在纹理映射时使用的同样大小的纹理数组。

```
for (i = 0; i < TEX_WIDTH; i++)
    for (j = 0; j < TEX_HEIGHT; j++)
        for (k = 0; k < 3; k++)
            texImage[i][j][k] = alpha*tex1[i][j][k]+(1.-alpha)
                                *tex2[i][j][k];
```

对动画的每一帧来说，需要破坏旧的插值纹理，生成新的插值纹理，并将新的纹理作为
418 活动纹理。当你这样做的时候，随着帧与帧之间几何信息变化纹理作自然的过渡。

11.8.6 改变模型的特征

模型有很多特征，在显示时，可以随时间变化来表达你的想法。这些可改变的特征包括颜色、光照属性、透明度、裁剪平面、变化、纹理映射、模型的粒度等。几乎所有可用变量定义而不是用常数定义的属性都可随着模型改变而改变。

作为这项技术的一个例子，我们改变分子模型中某种原子的尺寸和透明度来显示分子的结构，如图11-14所示。图像的变化由参数t驱动。参数t在idle回调函数中改变。图像中某原子的尺寸和透明度参数随着参数t的变化作正弦变化。这一效果在视觉上突出了该原子，使用户理解该原子在分子中的位置。这只是这一复杂方法的一个简单应用，你可以选择其他属性的动画效果来突出模型的某一部件。

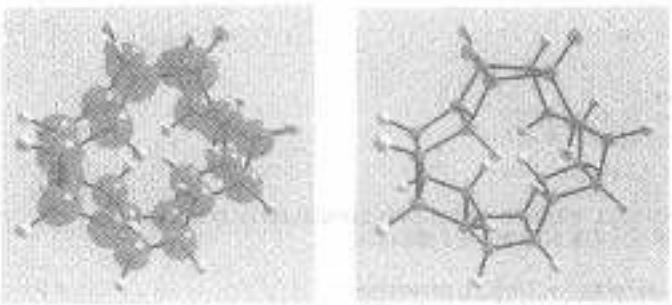


图11-14 含碳原子的分子的膨胀形态（左）和收缩形态（右）

实现这个效果的代码在下面给出，请参考第9章的分子观察模型例子。下面的animate()函数只改变sizeMult和alphaAdd两个参数的值。这两个参数用来设定分子颜色的α值和表示原子的gluSphere的尺寸。

```
void molecule(void)
{
...
    j = atoms[i].colindex;      // 原子i的颜色索引
    for (k=0; k < 4; k++)
    {   // copy atomColors[j], adjust alpha by alphaMult
        myColor[k] = atomColors[j][k];
    }
    if (j==CARBON) myColor[3] += alphaAdd;
    glMaterialfv(..., myColor);
    glPushMatrix();
        glTranslatef(...);
        if (j==CARBON)
            gluSphere(atomSphere,sizeMult*ANGTOAU(atomSizes[j]),GRAIN,
            GRAIN);
        else
            gluSphere(atomSphere,ANGTOAU(atomSizes[j]),GRAIN,GRAIN);
    glPopMatrix();
...
}
...
void animate(void)
{
    t += 0.1; if (t > 2.0*M_PI) t -= 2.0*M_PI;
    sizeMult = (1.0+0.5*sin(t));
    alphaAdd = 0.2*cos(t);
    glutPostRedisplay();
}
```

419

11.8.7 生成轨迹

图11-7所示的一种生成物体轨迹的方法就是生成一系列圆柱体，把物体以前数个位置连接起来，然后随着出现时间增长而淡出。下面的代码实现了这一功能，采用了一个全局变量tails，它把物体最后几个位置的坐标保存在数组list中，列表中的元素记录了物体前面的位置和方向，以及颜色和每个轨迹片段的长度。变量valid是在轨迹初始化时，在所有的轨迹片段尚未生成之前使用。

```
typedef struct { // 存放用于每个的单个圆柱体的属性
    point4 color;
    point3 position;
    point3 direction;
    float length;
    int valid;
} tailstruct;
```

```
void draw_tail()
{
    int j;
    float angle;
    point3 rot_vect;
    point3 origin={0.0,0.0,0.0};
    point3 y_point={0.0,1.0,0.0};

    for(j=0; j < T_LENGTH; j++)
    if(tails.list[j].valid) {
        glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,tails.list[j].color);
        // 计算圆柱体的旋转角度，使它朝向右边

        angle = asin(tails.list[j].direction[1]
            /sqrt(tails.list[j].direction[0]*tails.list[j].direction[0]
            +tails.list[j].direction[1]*tails.list[j].direction[1]
            +tails.list[j].direction[2]*tails.list[j].direction[2]));
        angle = angle*180/PI-90;
        // 计算垂直于方向向量和y轴的向量，该向量作为旋转轴
        //
        normal(tails.list[j].direction, origin, y_point, rot_vect);
        glPushMatrix();
        // 移动圆柱体段到正确位置，进行旋转，并设置其长度
        glTranslatef(tails.list[j].position[0],
            tails.list[j].position[1], tails.list[j].position[2]);
        glRotatef(angle, rot_vect[0], rot_vect[1], rot_vect[2]);
        glScalef(1.0, tails.list[j].length, 1.0);
        // 用包含12个片段的圆柱体绘制尾部分段
        cylinder(radius/30., 12);
        glPopMatrix();
    }
}
```

图11-7的另一个例子演示了一个随机运动的粒子在一定时间段内的轨迹，在这里我们采用了一个类似但是更简单的过程来表示这个运动轨迹，因为我们不希望对每段轨迹做特别处理。相反，我们保留了几个前面的位置，直接用折线段将这些位置连接起来，并用对比度很强的颜色显示出来。

11.8.8 使用累积缓存

累积缓存是OpenGL提供的各种缓存中的一个，是用于绘制的缓存，也是生成图11-8效果的主要工具。这个缓存中保存了RGBA颜色的浮点值，同时它和帧缓存逐像素对应。累积缓存中保存的值在[-1.0, 1.0]之间，如果缓存操作的结果超出了这个范围，这个结果将作为未定义的结果来处理（也就是结果随不同的系统而作不同处理，最后的值是不确定的）。这样，就需要注意定义你的操作，不要超过这个范围。使用这个缓存的基本目的，就是将一些显示操作要产生的不同权重的结果累积起来。它还有很多使用的方法，这些应用已经超出了本章的范围。如果读者对累积缓存的高级应用方法感兴趣，可以查阅OpenGL高级技术的手册和文献。

就像其他缓存一样，累积缓存也需要在OpenGL系统初始化的时候通过下面的函数来选择：

```
glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE|GLUT_ACCUM|GLUT_DEPTH);
```

累积缓存通过glAccum(mode, value)函数使用，其中的mode参数使用下面的某个符号名字定义的常量，value参数用某个浮点数作为它的值。可用的mode包括：

GL_ACCUM　　得到当前读缓存的RGBA值（单缓存情况下默认是从FRONT缓存中读，双缓存情况下是从BACK缓存中读，这样就不需要选择读哪个缓存），将这些值从整型转化为浮点型值，将值乘上value参数，然后将其累加到累积缓存。如果缓存有n位深度，整型向浮点型转化时将读缓存的值除以2^n-1。

GL_LOAD　　这个操作和GL_ACCUM模式很相似，它也会从读缓存读取数据，转

	化为浮点数，然后乘上value值，但它是用结果值替换掉累积缓存中的值，而不是与之相加。
GL_ADD	将value值与累积缓存中每像素点的R、G、B、A分量值相加，然后保存到该像素在累积缓存中的原始位置。
GL_MULT	将value值与累积缓存中每像素点的R、G、B、A分量值相乘，然后保存到该像素在累积缓存中的原始位置。
GL_RETURN	将累积缓存里面的像素值乘以value并将其按比例转换成读缓存的模型值以后保存到指定的读缓存。如果缓存有n位深度，那么转换的比例就是将值乘以2^n-1，这样就将值域控制在$[0, 2^n-1]$。

你可能并不需要用这些操作来展示运动轨迹。比如说要累加10个位置的图像，可以重画这个场景10次，然后把重画的10帧图像按权重2^{-i}累加到累积缓存中，i表示场景号，这里场景1代表了时间上最近的位置，场景10代表了时间上最远的位置。这里利用了10个权重之和

$$\sum_{i=1}^{10} 2^{-i}$$

非常趋近于1.0这个事实。这样，我们保证累积后的值一定在1.0之内。这项技术同样适用于静止物体，因为不同权重的物体图像的多重复制生成的图像几乎与原来单帧图像完全一样（如果第$i=10$步用2^{-5}权重，那么就不是“非常趋近于”1.0，而结果就是1.0）。实现这一技术的代码在下面给出：

```
// 假设函数的时间参数为t，将一系列时间参数存放在数组
// drawObjects(t) 中，用于绘制物体。这个例子中调用时
// 抖动为手times[10] that动设置，其他方式可以是随
// 机设置，当然随机设置不会产生本例的效果
//
//
//
    drawObjects(times[9]);
    glAccum(GL_LOAD, 0.5)
    for (i = 9; i > 0; i--) {
        glAccum(GL_MULT, 0.5);
        drawObjects(times[i-1]);
        glAccum(GL_ACCUM, 0.5);
    }
    glAccum(GL_RETURN, 1.0);
```

422

times[]数组在idle()函数中更新，每一次调用display()函数展示的是下一步动作以后的物体顺序。

这里有几件事件要提醒：在画时间上最远的图像前将该图像装入累积缓存而不是清掉缓存，这样做会节约一些时间；按时间从最远到最近的次序绘制图像；在绘制下一次图像前对累积缓存乘以0.5，新图像也乘以0.5以后再累加到累积缓存。这是一个自动不断地缩小时间上更远图像的权重的方法。

当然还可以使用其他的技术。例如，可以取一帧不管用什么方法生成的图像，将它乘以0.5权重后再存入到累积缓存中，绘制新的场景，再乘以0.5权重后加入到累积缓存中，最后用权重1.0返回场景。这可能比前面的方法快一点，而且不会和前面的方法相差太多。但这种方法不可能产生各种形式的抖动效果。抖动效果是一个有用的技术。

11.8.9 创建数字视频

创建数字化的动画相对容易，特别是把它与数字电影标准的复杂度相比较的话。本书是讲图形学的，不是讲数字视频格式，所以，可以用简单的工具抓取动画来生成数字视频，数

字视频也是从设计模型开始，包括设计随机变化的行为模型，然后通过计算机程序实现这个模型。实时动画由程序生成一系列帧的图像，然后按序播放呈现出动画效果，通过idle回调函数或者基于时钟的redisplay对动画的时间进行控制。生成数字视频的时候使用同样的顺序列，不过不是将它们显示出来（或者说不是生成的时候就立即显示它们），而是用OpenGL工具将每一帧保存在一个数组中，比如用下面的函数

```
glReadPixels(0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE, the_view)
```

其中将颜色缓存读到一个名为the_view的数组中。然后将每一个数组写到文件中，用动画工具将这些文件顺序连接起来。可能要通过编程使这些文件名除了序列号码不一样以外，还有一致的名字，这样，动画工具也能方便地处理这些文件。这些动画工具根据选择的数字视频文件格式（QuickTime，MPEG等）将连续的图像文件组合成动画，这样可以通过互联网发布，让大家共享你的动画。动画工具可能会给你一些选项，比如选择回放动画的帧速率。

数字视频（或者其他视频格式）的一个很大的优点在于，在动画中可以采用复杂的模型，而动画的回放速度不会因模型复杂度增加而下降。同时，动画的用户也不需要有编程背景，来编译运行你的原始程序。当需要对大量观众演示动画的时候，数字视频是很好的演示媒体。

11.9 用OpenGL制作动画时应注意的一些要点

处理移动视点比简单地生成单幅图像需要了解更多的OpenGL知识。视图变换是模型变换
423 的一部分。如果希望用参数控制视角，就要在合适的位置设置这些参数。在viewcube.c这个例子的display()函数中，你会注意到这里将模型变换矩阵设置为单位矩阵，然后调用gluLookAt()函数，变换结果保存为将来用。然后再调用旋转变换。这一变换过程让视图变换不是改变模型中物体的位置，而是设置好动画的观察方向。

最后，当在动画中使用纹理图的时候要格外注意产生纹理图走样现象。在动画中纹理图走样可能产生非常奇怪的视觉现象。在作动画时，应该采用一些纹理反走样的技术。

11.10 建议

看一下计算机图形学的视频作品，对应用这些工具可以做出什么样的动画有一个全面的了解。一般来说，你不会做高端的娱乐动画，而是关心富含信息的动画作品，即表达科学或技术工作的动画。请记住，当你观看电视或者商业视频动画时，你所看到的是专业级动画，或者说是一些能感动观众的动画。这样的作品往往要求非常详细的设计，需要非常复杂的考虑，以及高端的图形系统和工具。在计算机图形学初级课程中你将要做的是一些个人级或者同事级的动画，这些动画只是表达自己的想法，即自己用的，或者与你的同事、朋友交流用的。我们的经验是这些动画可能会非常有价值，已经有一些科技界的朋友要求复制一些学生做的动画作品，因为这些作品能有效地表达一些科学原理。因此不必和你看到的高端视频作品质量做比较，关键在于找到你的作品要交流的思想，你的作品才会有价值。

11.11 本章的OpenGL术语表

本章我们没有介绍太多新的OpenGL或者GLUT函数，我们介绍的函数不如前面介绍的通用，但是它们很有趣，同时也提供了非常有用的功能来实现动画技术。

OpenGL和GLUT函数

glAccum(param, value)：根据参数指定的模式使用累积缓存

int glutGet(state)：根据参数得到OpenGL的状态值

OpenGL和GLUT的参数

GL_ACCUM：指定函数glAccum()得到当前选择的读缓存中的RGBA分量，将它们乘以函数中的参数value，将结果加到累积缓存中。

GL_ADD：对累积缓存中的每一个元素都加上同一个值value。

GL_LOAD：和GL_ACCUM有同样的用法，只是读缓存数据经过比例放缩后装入到累积缓存。 424

GL_MULT：对累积缓存的每一个元素乘以指定的value值。

GL_RETURN：累积缓存中的值传递给选定的颜色缓存

GLUT_ELAPSED_TIME：glutGet()函数的参数，用来设定从调用glutInit函数（或者从第一次调用glutGet(GLUT_ELAPSED_TIME)函数）到该函数之间程序运行经过了毫秒数。

11.12 思考题

1. 一些老的计算机游戏和计算机动画存在一个问题是，游戏和动画运行的速度取决于处理器的速度以及图形卡的性能，也和系统其他属性相关。请讨论如何或者能否用timer事件将动画图像控制在一致的速度。采用glutGet(...)方法做同样的讨论。这样的技术可以让慢速的动画动得快一些吗？可以让快速的动画动得更慢一些吗？请对你的答案进行分析和论证。
2. 请讨论针对力控制的运动系统创建过程动画的方法：读者可以参考第9章的例子，或者用单个物体运动作例子，比如斜坡，或者物体从斜坡滑下，或者抛到空中的物体，或者用两个空间运动的物体在万有引力作用下绕共通旋转作例子。

11.13 练习题

1. 请用粒子系统设计一个焰火燃烧的模型，然后用动画显示你设计的焰火。可以简单地把它设计为单次爆炸使粒子朝各个方向散开，粒子燃烧产生固定的颜色，并保持固定的时间。也可以设计粒子在不同时刻有不同的颜色，或者两次爆炸，或者朝一个固定方向的爆炸等各种不同的变化（为了达到真实的焰火效果变化是没有止境的）。你还可以绘制焰火粒子的运动轨迹。
2. 用插值形状和插值纹理的思想生成一段小动画，从一个带纹理的形状开始变化到另一个带纹理的形状结束。如果你不仔细地处理中间的过程，插值的中间动画看起来会很别扭，所以，你需要在动画的每一步让形状和纹理的变化都是合理的。
3. 用随时间变化的一组参数生成一段动画，观察这组参数对物理系统的控制效果。比如生成一张表示带电粒子产生的二维空间中静电力的曲面（可以引用第9章介绍的库仑定律）。然后让一个粒子的电量随时间改变，观看表示空间力形状的曲面的变化。令粒子不仅改变其电量强弱，而且改变其正负极性。
4. 请按照本章介绍的移动视点方法，创建在空间中移动观察场景的操作。定义一个简单的空间场景。空间中可以安排你想要的物体，并在空间中设置一些控制点作为视点位置。创建一个有到下房子的简单场景，因为本章项目作业中包含了一道在房子中漫游的题目。请用三次基函数插值这些控制点生成一条路径，可以用Catmull-Rom样条，也可以用其他方法。写一段idle事件回调函数代码来计算路径上的 425
点，并用这些点作为显示该空间场景的视点位置和视线方向。

11.14 实验题

1. 运行并实验对应书上的例子程序，看看你还可以实现怎样的效果。你能让叶片看起来固定不动吗？什

么情况下会出现这种现象？你能否用叶轮模拟类似汽车加速的运动？在匀加速或者减速的时候你会看到什么样的叶片运动？你能将看到的叶轮运动和电影中的轮子运动联系起来吗（尤其是老电影）？

2. 用下面两个函数：$f_1(x) = a_1x^3 + b_1x^2 + c_1x + d_1$和$f_2(x) = a_2x^3 + b_2x^2 + c_2x + d_2$定义两个三次曲面。用它们实验从一个曲面插值过渡到另一个曲面的过程。将这一插值过程分为100步，先将曲面参数线性插值，生成插值的中间曲面，并用本书前面几章介绍的方法显示出来。然后再用平滑插值方法重复这一过程。你能否感觉到两种方法生成的动画有质的不同？
3. 将两个场景之间的插值扩展到在几个场景之间进行插值，可以为一组三次曲面定义对应的参数集$\{a_i\}$、$\{b_i\}$、$\{c_i\}$和$\{d_i\}$。然后建立相邻两组参数之间的平滑插值，从第一个场景开始逐个进行插值直到最后的场景结束。
4. 实现创建运动路径的概念：先定义一个起始点和一组速度。将速度进行样条插值生成一系列作用于路径的速度。请问用这种方法可以生成什么样的路径？
5. 为了搞清楚在观察静态图像中看不见的东西时运动所起的作用，这个实验要求重新实现芝加哥伊利诺伊大学电子可视化实验室Dan Sandin的经典工作，即图11-6所示的工作。创建带两个视口的窗口。一个视口是整个窗口，另一个视口位于水平和垂直窗口中央，其大小为窗口的1/3。在每一视口中创建一组随机的点集，它们的密度相同，即小视口中点的数目为大视口中点数的1/3。令大视口中的点缓慢地向右运动，小视口中的点缓慢地向左运动，运动的启动与终止由一按键控制。因此你可以方便地控制运动的启动和终止。观察静态图像，你能说出这里存在两个点群的事实吗？观察运动图像你又能说些什么？结论是什么？
6. 扩展上一个项目实验，令区域内外粒子做不同运动来看清楚区域形状。可以采用与图11-6所示随机噪声背景不同的背景，还可以采用与图中所示图形区域不同的区域形状，也可以采用与原始实验不同的方法来判断像素位于区域内还是区域外，原始实验中粒子运动采用滑动方式，可以采用不同的粒子运动方式。另外一些有趣的选项包括：采用扫描图像作背景，采用双色照片生成的复杂区域，由其中一色来定义区域。从这个实验中你能否就运动对人眼感知区域形状所起的作用做一评论。
426
7. 取一幅你生成的包含运动的图像，用累积缓存技术使图像中运动产生模糊效果。尝试不同权重把图像存入累积缓存。比如采用均匀权重，或者采用被图像“拍摄”时刻为中心分布的权重。请问哪一种权重的效果最好？
8. （场景图）在第7章中，有一个实验是将一个事件节点添加到场景图中，统计事件驱动引起的模型和视图环境的改变。请对idle事件做同样的工作，这样就能管理时间驱动的变化。
9. 我们前面看到应用idle事件生成的动画会产生不可预测的速度。在第7章中，我们将timer()回调函数作为控制动画速度的一种方法，但在本章中我们介绍了glutGet(GLUT_ELAPSED_TIME)函数，以及怎样用它来有效控制帧速率。请用idle、timer、glutGet()技术控制时间和动画，给出你的结果。

11.15 大型作业

1. 用前面交互或者动画项目作业中生成的图像，制作一本翻动型动画书来传达这些作业的思想。尝试不同数量、不同大小、不同硬度的纸张，这样可以方便地翻动这本书。翻动型动画书中表示的效果屏幕上显示的效果一样吗？你能发现显示器的某些特征能让屏幕显示效果更好些，还是翻动型动画书的效果更好？
2. 制作一个西洋镜作为课堂作业。课堂上创建一系列短小的描述科学问题的动画，你可以用idle回调函数生成动画中的图像来作为西洋镜的图像，或者你也可以专门生成少量的循环显示的图像。当然，如果班上有人拥有西洋镜，就不必自己制作了。
3. 应用你在前面作业中生成的图像制作一段数字视频。可以将每一帧图像，或者一段间隔相等的图像保

存到数字图像文件中，再把每一个文件转换成视频编辑器可以识别的文件格式，最后将不同的文件整合成一段数字视频文件。

4. （小房子）在第7章讨论事件的时候，有一个作业就是对房子实现漫游，现在让这个漫游自动化，在房子里选择若干特定的点作为观察点，然后用样条曲线或者其他技术将它们连接起来，可以得出这些观察点之间的视点位置，这样摄影机才能自动沿着路径对房子进行漫游。

5. （场景图解析器）实现将事件节点添加到场景图的实验，用它来指定和生成模型关于时间的变化，以及对场景主基于时间的观察。 427

第12章　高性能图形技术

本章我们将介绍一些计算技术和图形技术，这些技术可以加快图形应用程序的运行速度。具体而言，借助于这些技术，可以提高动画的帧速率，也可以加速图形应用程序对用户的交互响应。但是本章并不打算深入介绍每一项技术，因为在游戏开发领域中存在许多技术，用于加速游戏程序的图形处理，并且图形硬件加速器和图形API的发展速度比任何一本书的出版速度都快。对于本章介绍的各种技术，有些与系统相关，有些涉及建模技巧，有些需要利用图形API高级甚至非常复杂的功能，还有一些将跳过图形API以便使用其他类型的图形处理功能。对此，我们建议读者参考本书中与游戏相关的参考文献，以及游戏开发组织提供的高级游戏开发资料。

高性能图形研究的最终目标是达到实时图形处理，即按照被模拟系统同样的速度运行。该研究领域包含沉浸式虚拟现实中的图形处理技术。在沉浸式虚拟环境中，用户需要实时感知被模拟的虚拟世界。由此可见，高性能图形处理不仅是图形学领域的挑战，也是仿真、可视化以及虚拟现实等领域的挑战。因此，这些研究领域中的相关文献也具有参考价值。

为了掌握本章介绍的各项技术，读者需要对计算机图形学的基本知识具有比较深入的理解，并且熟练掌握OpenGL图形API。正是基于这些深入的理解和对细节技术的掌握，使得我们能利用一些非常规的方法快速地生成高质量的图像。

12.1　定义

无论绘制单个场景还是动画，我们总是希望能尽快看到绘制结果，因此图像的生成速度是非常重要的。等待图像缓慢地生成不但令人厌烦，甚至会影响图像的感知效果。大多数图
428 形应用都需要面对这个问题，尤其是计算机游戏。因此，本章所讨论的技术多以游戏为背景。但是，这些技术对于其他图形应用也十分重要，因为图形处理性能始终是图形计算中的一个关键问题。

制作高质量的计算机游戏需要涉及很多方面，包括构思情节、创建角色、维护游戏情节数据库，以及许多加速游戏玩家操作的编程技巧。由于游戏的图形界面将整个游戏呈现给玩家，具有最大的计算开销，因此成为影响游戏性能的关键因素之一。这个问题不同于前面介绍过的所有计算机图形学问题。前面章节以图像质量为中心，尽可能地保证绘制性能。而在本章中我们将重心反过来，以程序性能为中心，而尽可能地保证图像质量。这个研究重心的转变使得本章介绍的处理方法和技术与前面的章节非常不同。

实际上，高性能图形处理并非计算机图形学研究的新方向。早在二十多年前，计算机领域就将“实时图形处理”纳入其研究内容。不过最初的研究主要集中在飞行模拟器、重要安全系统过程的实时监控、实时仿真等。这些研究往往借助于当时最快的计算机来完成。在虚拟现实领域中也开展了一些实时图形处理研究，其中主要研究如何将可交互的虚拟世界实时地呈现于用户面前。教育和培训应用中也使用过一些实时图形处理技术，但这些应用本质上可视为仿真。就目前发展而言，游戏是实时图形处理研究的主要推动力。高性能图形应用迫切需求各种图形处理技术，这些应用需求与广泛使用的具有不同软硬件配置的个人计算机密切相关，因此有必要在一本图形学教程中介绍一些重要的实时处理技术。

12.2 技术

高性能计算机图形技术，尤其是应用于游戏中的技术，通常都利用了以下几个简单原则：

- 尽可能利用硬件加速，但并非所有用户都拥有并且愿意使用硬件加速器。
- 对不需要显示的部分进行判断。
  - *利用精选技术，要求计算量小，能从整个需要显示的集合中精选一些物体或物体的一部分。
- 利用纹理映射功能。
  - *通过纹理映射创建物体，而不必完整地绘制物体。
  - *使用多纹理和具有高低不同分辨率的纹理。
- 尽可能利用各种支持快速显示的技术。
  - *顶点队列
  - *显示列表
  - *细节层次
- 避免不必要的光照计算。
  - *启动物体附近的光源来绘制物体。
  - *适当使用雾化，以隐藏质量不高的绘制结果。
- 采用近似碰撞检测，避免执行精确碰撞计算。

429

上述列表包含了建模技术和绘制技术，在后面分别进行讨论，便于读者理解和掌握。本章将尽可能介绍各种加速技术，帮助读者提高图形处理性能。但是加速技术不仅限于本书所介绍的，有些技术依赖于特定系统，或者超出了本书的范畴。如果读者发现图形处理性能受限，并且需要加速，那么首先需要分析应用程序，找出运行时间消耗在哪里。根据性能分析，再选择恰当的加速技术，从而获得最大的性能改进。

与其他图形应用不同，游戏开发中还有一些独特的问题，如碰撞检测。由于碰撞检测要求运算简便，并且存在一些方法，能有效将其计算过程简化，因此本章将对其进行简单介绍。

12.3 建模技术

建模技术主要涉及两类，一是减少模型的多边形数量，从而简化场景，减少绘制工作量；二是增加模型特征，从而简化其绘制方法，提高绘制速度。虽然这类简化程度有限，但当图形处理系统处于性能极限时，一定程度的简化将会非常有用。

12.3.1 减少可见多边形数量

在进行场景的总体布局时，确保无论从哪个视点观察，整个场景中只有有限数量的多边形可见。一般游戏中总是会构造许多墙，也是基于此原因。许多大的多边形通常使用纹理映射来显示其细节，在任何视点下可见的多边形数量不多。而视觉感知是通过在不同场景之间快速移动变得丰富的。因此，当玩家从一个场景移动到另一个场景时，总能看见不同的事物。虽然场景简单，但只要保持不断变化，就能使游戏具有新鲜感。

其他技术是根据视点的位置预先计算出玩家可以看见的物体。举个简单的例子，当玩家从一个场景移到另一个场景时，原来场景中的物体就变得不可见，因此可以忽略处理其中所有的多边形。这种预计算方法通常需要维护一个可见多边形列表，该列表随着视点位置和视线方向的变化而更新。这种方法采用的是典型的以空间换速度的方法。

12.3.2 巧妙运用纹理

纹理不但可以使简单的场景看起来更复杂，而且可以增强虚拟物体的真实感。本章在利用纹理映射功能时，将以另外一种方式来处理这些图形操作，即前面所介绍的以难以察觉的方式降低精度，从而提高图形处理效率。有些技术在第8章中已经介绍过，但由于它们对于提高图形处理性能非常重要，因此我们仍将其包含在本章中。

一种技术是布告板技术，用纹理映射方法创建复杂物体，并将表示复杂物体的纹理映射到二维物体（一块矩形板）上进行显示。复杂物体的纹理可用快照方法获得，也可用图形学方法生成。利用纹理映射中的α通道，可将纹理映射到一个朝向视点的矩形板上，从而在该矩形板上显示出树木、建筑或车辆的图像，α通道可以使快照中除了所要表示的物体可见外，其他物体均不可见。如果重复上述处理过程多次，则可以构建诸如森林、城市、停车场之类的复杂物体，而不必要通过大量计算来创建和绘制这些复杂的物体。为了使每个布告板都朝向视点方向，
430 需要进行以下操作：一是计算视点和布告板的位置，可以在场景图中合成作用于它们的平移变换而得到；二是以布告板为坐标中心计算视点的柱面或球面坐标，根据视点相对于布告栏的经度和纬度，旋转布告板使之朝向视点。需要注意，布告板有两类不同的观察方式，如果布告板表示具有固定基平面的物体，如树木、建筑、人物等类似物体，则一般按其固定垂直轴进行旋转；如果布告板表示的物体没有固定基平面，如标示牌、雪花等，则通常需要将其绕两个轴进行旋转，使其朝向视点。在第8章中已经介绍过如何计算旋转，在此省略。

另外一种技术是通过金字塔纹理贴图（mipmaps）使用不同分辨率的纹理。其中mipmaps为具有多个层次的不同分辨率的纹理贴图集。从具有最高分辨率（也是最大尺寸）的纹理贴图开始，可以自动创建出一系列的低分辨率的纹理贴图。在OpenGL中，由于纹理贴图每一维的大小都必须是2的幂次方，因此新生成的低分辨率纹理贴图一般是原纹理贴图的二分之一、四分之一等。使用这些不同分辨率的纹理贴图，还可以避免使用单一高分辨率纹理贴图所带来的走样现象。

最后一种技术是使用分层纹理映射，以达到预期的视觉效果。这也就是第8章介绍的多纹理映射技术，即按照一定顺序将多个纹理贴图应用到同一多边形上。例如，首先应用颜色纹理贴图构造一面砖墙，然后对部分区域应用亮度纹理贴图以增加其亮度，这样不需进行任何光照计算，就可以模拟出光线穿越窗口照射在墙上，或者火炬照亮墙面等效果。

12.3.3 减少光照计算

在OpenGL中，可以在场景放置8个（或更多）光源，每个新增的光源都会增加场景的绘制时间，请回忆一下光照的计算过程，为了计算每个多边形或顶点的光照，需要计算场景中每个光源的环境分量、漫反射分量和高光分量，并把它们加起来。可是，如果光源是具有衰减性质的位置光源，距离该顶点很远，那么它对顶点的光亮度贡献将非常小。因此可以关闭位置较远的光源，从而减少光照的计算时间。上述方法的基本原理与本章前面介绍的相同，即减少计算开销，从而为图形处理节约时间。这种方法在绘制动画时存在一个问题，就是快速的关闭光源会使得动画产生明显的亮度闪烁现象，因此在实际应用时需要做折衷处理。

12.3.4 细节层次

细节层次技术（也称为LOD）可以根据场景中用户视点的变化而对绘制物体的细节进行调整，内容主要包括以下几个方面。构建图元的多种表示，并根据图元到视点的距离选择其中一种表示进行绘制，当物体距离视点非常远，使得其显示非常小时，可以选择不绘制该物

体，而当物体距离视点不是很近，可选择其粗糙或模糊表示进行绘制。通过细节层次技术，当物体接近视点时就显示其所有细节，而物体远离视点时，就显示其简化表示。从而不但可以节省绘制时间，还可以控制物体的绘制方式，即绘制或不绘制，粗糙绘制或精细绘制。

除了mipmaps技术外，OpenGL并不直接支持细节层次技术，因而也没有给出细节层次的相关定义。但是，随着几何物体和虚拟场景变得日益复杂，实时绘制这些复杂几何数据也变 431
得更加重要，细节层次技术成为了图形系统中一个十分重要的问题。即使利用速度更快的计算机系统，其最终绘制性能也是有限的，因此使用高效的场景绘制技术总是必要的。

细节层次技术的一个重要思路是，无论物体距离视点远近，其绘制结果图像都应该是尽可能相似的外观。具体而言，当物体距离视点越远，就可以绘制越少的细节，或者使用较粗糙的逼近表示。当然，如果图元的绘制结果小于一个或几个像素，就可以选择不绘制该图元。但是如何减少远距离物体的细节，以及如何增加近距离物体的细节，目前通常采用启发式的方法，而自动网格简化的相关研究正不断对此进行改进。

细节层次技术比雾化技术更难于解释，因为它要求从图元的多个模型表示中选出一个进行绘制。对此，一个标准的方法就是选出图元上的一点（*ObjX*，*ObjY*，*ObjZ*），用该点确定视点到图元的距离。OpenGL提供一些函数用于实现该功能，类似的示例代码如下所示：

```
glRasterPos3f(ObjX, ObjY, ObjZ);
glGetFloatv(GL_CURRENT_RASTER_DISTANCE, &dist);
if (farDist(dist)) { ... // 远距离图元定义
}
        else { ... // 近距离图元定义
}
```

根据以上代码，当物体远离视点时（具体距离由函数float farDist（float）确定），就可以选择绘制某种模型表示；而当物体靠近视点时，就可以选择绘制另一种模型表示。当物体具有两个以上的建模表示时，仍然可以使用距离函数

```
glGetFloatv(GL_CURRENT_RASTER_DISTANCE, &dist)
```

返回物体到视点的距离，并根据该距离来修改图元的几何建模语句。

下面通过一个示例来解释细节层次技术的概念，该示例根据距离的远近显示不同分辨率的GLU球体。请回忆第3章介绍的一种建模技术，即使用函数

```
void gluSphere(GLUquadricObj *qobj, GLdouble radius, GLint slices,
               GLint stacks);
```

可定义一个位于原点并具有指定半径的GLU球体。整型参数slices和stacks用于指定球体的精细程度，其值越小，球体越粗糙，所含多边形越少，绘制速度越快；相反，球体越平滑，绘制速度越慢。对于此类问题，细节层次技术使用的方法就是，定义在什么距离改变球体的分辨率，以及将球体分辨率改变到多少，即将slices和stacks的数值调整到多少。当然最理想的处理方法就是，首先分析球体每个多边形需要显示的像素个数，然后选择与之匹配的slices和stacks数值。

这里所采用的建模方法是创建mySphere（）函数，输入参数为球体中心和球体半径。球体的深度通过以下步骤确定，判断出球心位置，然后查询得到其到视点的距离，通过简单的 432
运算得到slices和stacks的值，并将其传入gluSphere（...）函数中，从而用于选择恰当的多边形精细度。其核心代码如下所示。图12-1显示了不同细节层次球体的绘制结果。

```
    myQuad=gluNewQuadric();
    glRasterPos3fv(origin);
//  howFar = 球心到视点的距离
    glGetFloatv(GL_CURRENT_RASTER_DISTANCE, &howFar);
    resolution = (GLint) (200.0/howFar);
    slices = stacks = resolution;
    gluSphere(myQuad , radius , slices , stacks);
```

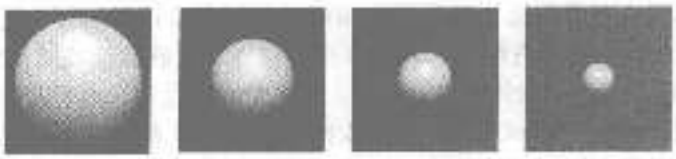


图12.1 球体的细节层次，从高细节层次（左）到低细节层次（右）

将LOD技术应用于动画或动态场景时，需要注意避免以下两个现象，一是物体的突然出现或突然消失（例如被远裁剪平面裁剪掉或没剪掉），二是物体外观属性的突然变化。因为这些现象会引起用户感知的中断，从而降低相关操作的可信度。对此，为了避免物体从空间某个位置突然跳出，可以在场景远处创建雾化区域，让物体经过雾化而逐渐显示出来。

12.3.5 雾化

雾化技术通过减少模型的可见性来隐藏细节，从而使得在场景中可以采用简化模型。但是这种折衷方法并不一定能提高绘制性能，因为计算雾化效果的时间，有可能比从绘制简化模型中节省的时间还要多。在这一节介绍雾化技术，并不是出于对效率的考虑，而主要因为它与细节层次技术的内在原理很相似。

在使用雾化技术时，OpenGL将图像绘制到颜色缓存，而最终的显示图像是将其与雾色混合得到的。具体的混合方式由深度缓存中的深度值控制。在控制雾化效果时，可以指定颜色混合的开始距离和结束距离，以及雾色在这两个距离之间区域中的变化方式。启动雾化效果后，比开始距离更近的物体，其颜色将不会发生改变；而在开始距离和结束距离之间的物体，其颜色将随着距离的增加而褪色，并逐渐过渡到雾色；对于远于结束距离的物体，其颜色将被雾色替代，最终的雾色将由雾密度参数确定。上述方法可以提供场景的深度提示，在某些场合中非常有用。

为了在OpenGL使用和控制雾化，下面将介绍一些基本概念。这些概念所对应的具体控制
433 可以通过glFog*（param，value）函数实现。与其他系统参数设置类似，所有大写的参数表示具体的参数值。在下面的讨论中，假设颜色采用RGB或RGBA模式。

12.3.6 开始距离和结束距离

雾化的开始距离和结束距离分别由GL_FOG_START和GL_FOG_END参数指定。在这两个距离之间，将增加雾化效果，而在开始距离之前，没有雾化效果；在结束距离之后，雾化效果恒定不变，等于雾色。需要注意，这两个参数值采用通常的坐标惯例，即观察中心在坐标原点，视点位置位于坐标轴负方向。一般情况下，雾化开始距离设置为0，而结束距离设置为1。

12.3.7 雾化模式

OpenGL中提供三种内置雾化模式：线性、指数和指数平方。这三种雾化模式按照不同的方式计算雾化因子ff，从而改变图元颜色和雾色的混合结果，具体计算方式如下：

- GL_LINEAR：$ff = density * z'$其中$z' = (end - z) / (end - start)$并且$z$位于$start$和$end$之间
- GL_EXP：$ff = \exp(-density * z')$对于z'同上
- GL_EXP2：$ff = \exp(-density * z')^2$对于$z'$同上

在计算得到雾化因子后，将其截断到[0, 1]范围之间，对于上述三种雾化模式，计算最终显示颜色Cd的方式都相同，即使用雾化因子ff对图元颜色Ce和雾色Cf进行插值，插值公式如下所示：

$$Cd = ff * Ce + (1 - ff) * Cf$$

12.3.8 雾密度

可以将雾密度看做是雾化对图元颜色的最大衰减程度，当然该最大衰减程度也依赖于所使用的雾化模式。雾密度越大，物体消失在雾化中的速度越快，雾化效果看起来也更浓密。请注意，雾密度的数值一定介于0到1之间。

12.3.9 雾色

虽然一般认为雾是灰色的，但是在图形系统中雾可以呈现出各种颜色。雾色可以使用glFog*()函数来设置，给定的颜色参数可以是一个四元组向量参数或四个单独的参数值，其数值类型可以是整型或浮点型。glFog*()的各版本与glColor*()和glMuterial*()函数类似，具体细节请参考OpenGL手册。由于雾化只会改变图元颜色，而不会影响背景颜色，因此将雾色设置为与背景颜色相同的效果比较好，如图12-2所示。



图12-2 雾化的立方体（其中一个面具有纹理图），参见彩图

下面简单介绍雾化技术的其他两个设置选项。第一，除了RGB或RGBA模式外，在索引颜色模式下，也可以使用雾化效果，此时，雾化因子将对颜色的索引进行插值，而不是对颜色本身进行插值（在前面讨论颜色模型时并没有涉及索引颜色模式，但在一些较早的图形系统中只能使用索引颜色模式）。第二，雾化效果具有提示机制，可以使用glHint(...)函数，并给定参数GL_FOG_HINT和提示级别，从而加速雾化的绘制速度。

下面将展示如何使用雾化技术。所有的雾化效果可以在初始化函数中通过下列参数来定义，包括雾化模式、雾色、雾密度、雾化开始距离和结束距离。但实际的雾化效果要在绘制时才会显示出来。最终的绘制图像将根据指定的雾化效果，将图元颜色与雾色混合得到。混合的方法在第10章中已经介绍过，在此省略。下段代码仅给出了与雾化相关的函数。

```
void myinit(void)
{
    ...
    static GLfloat fogColor[4] = {0.5,0.5,0.5,1.0}; // 50% 灰色
    ...
//  定义雾化参数
    glFogf(GL_FOG_MODE, GL_EXP);          // 按指数方式增加雾化
    glFogfv(GL_FOG_COLOR, fogColor);      // 设置雾色
    glFogf(GL_FOG_START, 0.0);            // 标准开始距离
    glFogf(GL_FOG_END, 1.0);              // 标准结束距离
    glFogf(GL_FOG_DENSITY, 0.50);         // 设置雾密度
    ...
    glEnable(GL_FOG);                     // 启动雾化
    ...
}
```

图12-2显示了常用的带纹理贴图的立方体在雾化空间中的效果，其中，立方体有三种不同的侧面（红色、黄色和带纹理贴图的），而雾密度仅设置为0.15。请读者尝试改变其中的雾化模式、雾色、雾密度、雾化开始距离和结束距离等参数，并观察这些参数对雾化效果和最终图像的影响。

雾化技术的吸引人之处就在于它可以使物体变得朦胧，而不像一般图形技术绘制出的物体那样鲜艳夺目。纹理映射技术和平滑着色处理技术也具有类似的优点，使用纹理映射避免物体看起来像平滑的塑料制品，采用平滑着色处理避免物体看起来表面粗糙。但是，使用这

434

些技术都需要额外编写程序，并且会增加绘制时间。因此，在确定其对视觉交流有益的情况
435 下，可以使用这些技术，否则将偏离图形处理的真实意义。

12.4 绘制技术

同样是为了提高绘制性能，但建模技术和绘制技术的研究思路却不同。建模技术主要研究如何减少需要处理的绘制量，而绘制技术将主要研究如何让绘制过程变得更高效。有些绘制技术在进入图形流水线之前先对几何数据进行处理，而其他的将对几何数据进行调整，使得图形流水线能更有效地进行绘制。

12.4.1 不使用硬件

在图形处理流水线中，有些处理步骤可以使用硬件执行，从而提高绘制性能，这些正是图形硬件加速器的长处。在具有图形加速器的系统上使用OpenGL时，图形处理系统一般都会自动使用硬件提供的功能。但是在某些情况下，这种方法并不能获取最好的绘制性能，虽然这看起来有些矛盾。例如，有些用户的图形加速器并没有所需要的加速功能。此外，即使使用硬件加速，也需要耗费一定的处理时间。更根本的原因是，利用其他技术来避免硬件加速器对应的处理过程，将比使用一般方法然后结合硬件加速的绘制速度更快。

举个例子。目前大多数图形加速器都支持深度缓存，在使用深度缓存进行深度比较时，需要对绘制的每个像素进行读取、比较和写入操作。如果图像加速器的速度足够快，那么这些读取、比较和写入操作的执行速度也很快。但是，如果能避免这些操作，显然比对其进行优化更快。目前已经有些技术提出，它们通过运算的计算，就可以剔除整个多边形，甚至完全避免深度测试。

12.4.2 使用硬件

正如有时不需要利用图形加速器提供的硬件功能一样，有时候的确需要充分利用图形加速器，并基于图形加速器来编写程序。一些通用技巧将在本章后面介绍，包括几何压缩、显示列表和顶点数组。一般而言，将更多的绘制任务加载到图形加速器上，从而充分利用其片上处理器，图像的绘制速度就越快。随着图形API开始支持诸如顶点或片元处理这样的功能，图像的绘制结果也变得越来越精致，同时可以保证一定的绘制速度。

12.4.3 多边形剔除

一种减少绘制工作量的方法就是，用多面体（或多面体集合）构造物体，然后识别出多面体中背向视点的多边形。假设多面体均为不透明的，那么任何背向视点的面都不可见。如果绘制这些背向视点的多边形，再依赖深度缓存进行隐藏面消除，但实际上由于所有像素都不可见，它们对最终的结果图像将毫无贡献。可见更有效的方法就是识别出这些不可见多边形，并且不绘制它们。

测试多边形面向视点或背向视点的方法很简单。由于多边形的法向量指向其表面的正向（或外向），如果该法向量指向视点，则该多边形为正面；如果该法向量背离视点，则该多边形为背面。其中，正面有可能可见，但对于表面
436 封闭的物体，其背面一定不可见。如图12-3中所示，其中N表示多边形的法向量，E表示指向视点方向的向量。对于正

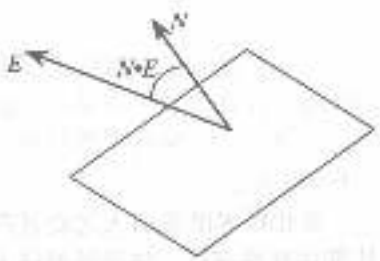


图12-3 正面多边形测试

面，其N与E的夹角为锐角，且点积$N \cdot E$为正值。对于背面，即将E改为反方向，则N与E的夹角为钝角，且点积$N \cdot E$为负值。根据上述分析，可见性测试可以简化为判断点积$N \cdot E$的代数符号。上述方法通常称为背面剔除，简而言之，就是对多边形进行测试，并且不绘制背向视点的多边形。

背面剔除的程序实现很简单，一般可以在调用图形API函数之前执行，但许多图形API也直接支持背面剔除，在OpenGL中，启动GL_CULL_FACE操作就可以使用背面剔除。其中，函数

```
void glFrontFace(GLenum mode)
```

用于指定正面的定义方式。参数mode可选择GL_CCW（逆时针方向）或GL_CW（顺时针方向），具体方式根据从视点观察正面时顶点的走向而确定。函数

```
void glCullFace(GLenum mode)
```

用于指定将被剔除的表面类型，参数mode可以选择GL_FRONT、GL_BACK或GL_FRONT_AND_BACK。在启动背面剔除功能后，满足glFrontFace和glCullFace函数所定义的多边形将被剔除掉，从而不再绘制。

另外一种剔除方法在视域体中执行，它将多面体或多边形的每个顶点与视域体的包围平面进行比较，如果所有顶点均位于同一包围平面的外侧，那么该多面体或多边形在视域体中不可见，不必绘制。上述比较操作应该在视图变换之后，实际绘制之前执行，这样才能使用正确的视域体的边界平面进行比较。前面已经介绍过，视域体定义为一个矩形金字塔，其顶为坐标原点，并且朝Z轴的负方向延伸。因此，实际的比较计算可以表示为如下方式：

$y > T*Z/ZNEAR$ 或 $y < B*Z/ZNEAR$

$x > R*Z/ZNEAR$ 或 $x < L*Z/ZNEAR$

$z > ZNEAR$ 或 $z < ZFAR$

其中T、B、L、R分别表示近裁剪平面$Z = ZNEAR$的上、下、左、右四个坐标，具体位置如图12-4中所示。

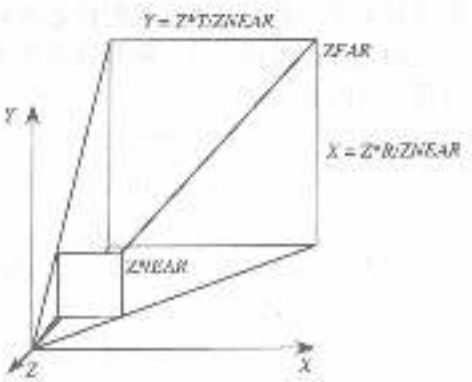


图12-4 与视域体包围平面进行比较

12.4.4 避免深度比较

画家算法是一种最流行的图形处理技术，其模拟光线离开物体射入人眼的过程，按照深度对物体进行排序，然后按从后到前顺序绘制物体。在Z缓冲区出现之前，这种算法非常流行。虽然目前大多数图形系统都具有Z缓冲功能，但是使用Z缓冲区进行深度比较还是需要一定的计算开销。因此，若能避免深度比较，将能进一步提高程序的绘制性能。

如果模型为静态的，不存在连锁多边形，而且在一个视点观察，那么画家算法实现起来很简单。因为在这种情况下，可以很容易地给出“前”“后”位置的定义，以及判断任意两个多边形的前后关系。然而上述假设条件不是交互式图形应用的设计原则，尤其是对于游戏。因为对于交互式图形应用，物体和视点总是在运动，而这些运动使得物体的前后关系也在发生改变。若要使用画家算法，必须要计算从任何方向到视点的距离，而这个计算量将会非常大。

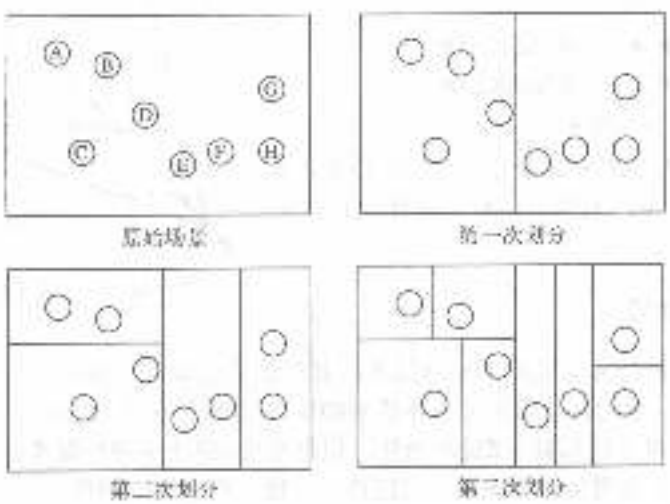
对此，一种可能的解决方法就是按某种方式定义场景，无论从哪个视点位置观察场景，场景中物体之间的深度关系都已经确定了，或者能通过某种方法将这些深度关系计算出来。二元空间划分就是解决该问题的常用方法。

12.4.5 从前到后绘制

有时，对一些好方法进行逆向思维，结果仍然是不错的方法。例如对于画家算法，在完成排序后，按照从前到后的顺序绘制物体，这种方式仍然有用。此时，虽然仍然需要进行测试，但是只需在写入像素前测试该像素是否被写入过（而不必进行读出深度信息再进行比较）。
438 有些复杂多边形，其每个像素的计算开销都很大，例如带有复杂的纹理图的多边形。对这些多边形，一种不希望看到的结果就是，在经过复杂的计算之后，该像素又被覆盖和重写。对此，如果从前到后的绘制物体，并使用Z缓冲区消除不需要绘制的像素，就可以只对确需要绘制的像素进行实际复杂的计算。BSP树（二元空间划分树）技术可以用于选择最近的物体，并首先绘制这些物体。也可以通过其他技术或预先设计场景来确定最近的物体。

12.4.6 二元空间划分

还有其他方法可以避免深度比较。例如，二元空间划分不但可用于判断物体的可见性，还可以用于确定在给定视点下物体的前后顺序。二元空间划分使用场景空间中的平面，将场景划分为不同的凸子区域，从而使得各子区域的前后关系可以很容易地计算。场景的划分过程通常是递归的：首先找到一个平面，要求该平面与场景中的物体不相交，并且该平面两侧的物体数量大致相等；然后将该平面两侧的子空间看做单独的场景，按照这种方式继续划分。通常尽可能选择简单的平面，诸如与坐标平面平行的平面，如果建模系统创建模型不支持这种平行平面，也可以选择其他任意类型的平面。如果无法在两个物体之间确定一个平面，二元空间划分技术将失效，除非通过其他更复杂的建模技术。图12-5以二维空间为例说明二元空间划分的基本原理。



439 图12-5 位于划分空间中的物体

上述划分过程可以根据一棵二元空间划分树（BSP树）来绘制图像。这棵树通常随着空间的划分而逐渐建立，树的中间节点为划分平面，叶节点为实际绘制对象，对于每个中间节点，将存储对应划分平面的平面方程，该节点的每个分支将存储一个标志，用于标记该分支位于这个划分平面的正面或负面，即坐标代入这个划分平面方程后为正值还是负值。图12-6给出了BSP树的一个示例，其中叶节点用其所包含物体的字母来表示。利用这类BSP树，可以很容易地计算出哪一侧距离视点更近。任意给定一个视点，即使运动视点同样适用，对每个中间

节点，判断其哪个分支距离视点更近，将更近的分支调整为右分支，而更远的分支调整为左分支。图12-6经过调整后，视点将位于该BSP树右下方叶节点的外侧。实际绘制时只需从左到右遍历该BSP树的叶节点，逐个绘制叶节点中的物体。

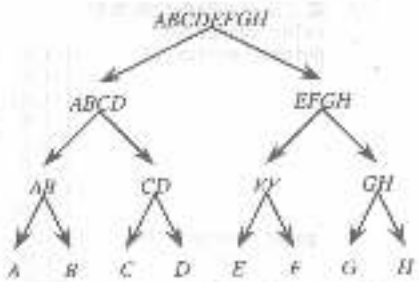


图12-6 二元空间划分树（BSP树）

在判断中间节点的哪侧距离视点更近时，只需利用视点位置和中间节点对应的平面方程即可，具体方法是，将视点位置和物体坐标分别代入中间节点的平面方程，若得到数值的符号相同，则物体与视点位于该平面的同侧，物体所在的分支距离视点也更近。当视点移动时，只有当视点跨越某个划分平面时，才需对BSP树进行调整，并且通常只需进行部分调整，因为其中某些分支的前后关系并不发生改变。

如果场景存在运动物体，那么需要考虑它们与其他物体，以及与BSP树的位置关系。一种常用的方法是，只让运动物体出现在其他物体之前。对此，先用BSP树绘制场景，然后再绘制运动物体。如果运动物体可能处于其他物体之间，则需要将其加入到BSP树中，并根据其运动调整其所属的节点分支，其计算量与确定视点在BSP树中的位置一样，类似地，运动物体也只有在跨越一个划分平面时才会改变其所处的区域。 440

12.4.7 系统加速技术

几何压缩是图形API中最简单的一类系统加速技术，如三角形条带、三角扇形和四边形条带都是常用的几何压缩技术。即使已经完成了可见性剔除和细节层次选择操作，得到了需要绘制的多边形列表，仍然可以使用这些压缩技术来绘制几何图元，减少传输到图形处理系统的顶点数量，从而进一步提高绘制性能。

将几何数据编译到显示列表中是OpenGL中的另外一种系统加速技术，在第3章中介绍过，可以将许多图形操作组合到显示列表中，其执行速度比原始操作更快。这是因为在显示列表创建时所需的计算就已经完成了，而在运行时只需将计算结果发送到最后的显示输出阶段。如果将图像块预先组织到显示列表中，也能提高其绘制速度。但是由于几何数据一旦编译到显示列表中，就不能对其进行修改，因此显示列表中不能包含多边形剔除，或者改变显示列表中图元的顺序。

采用顶点数组和法向数组（以及颜色数组和纹理数组）是一种更为通用的系统加速技术，其速度比几何压缩技术快，但比显示列表技术慢。利用这种技术，所有的几何数据，如顶点、颜色、法向和纹理等，都只需存储一次，而使用时只需提供索引即可。在使用这种技术时，首先需要定义数组，然后将几何数据填充到数组中，最后启动需要使用的数组。在启动数组后，就可以调用glArrayElement(int)函数访问给定顶点索引的几何数据。此外，也可以使用glDrawElements(...)和glDrawRangeElements(...)函数访问顶点数组中的一组数据，从而提高访问效率。

为了说明顶点数组的使用方法，下面将修改前面示例中的标准立方体函数，示例代码如下所示，示例中使用了两个数组，分别用于存储顶点几何数据和顶点法向数据。顶点几何数据与原来的标准立方体一样，而顶点法向数据为顶点几何数据的副本，即每个顶点的法向量为对象中心到该顶点位置的向量，因此该立方体的法向实际与球体的法向相同。定义两个数组后，需要使用glEnableClientState(...)函数启动顶点和法向数组，并使用glVertexPointer()和glNormalPointer()函数将定义的数组与数组操作方式进行绑定，最后就可以使用glDrawElements(...)函数进行绘制。图12-7给出了该代码的绘制结果。

```
void cube(float r, float g, float b)
{
//  定义顶点和颜色数据类型
    color cubecolor;
    point3 vertices[8]=   {{-1.0, -1.0, -1.0},
                           {-1.0, -1.0,  1.0},
                           {-1.0,  1.0, -1.0},
                           {-1.0,  1.0,  1.0},
                           { 1.0, -1.0, -1.0},
                           { 1.0, -1.0,  1.0},
                           { 1.0,  1.0, -1.0},
                           { 1.0,  1.0,  1.0} };
441
    point3 normals[8]=    {{-1.0, -1.0, -1.0},
                           {-1.0, -1.0,  1.0},
                           {-1.0,  1.0, -1.0},
                           {-1.0,  1.0,  1.0},
                           { 1.0, -1.0, -1.0},
                           { 1.0, -1.0,  1.0},
                           { 1.0,  1.0, -1.0},
                           { 1.0,  1.0,  1.0} };
GLubyte face1[4] = {1, 5, 7, 3};
GLubyte face2[4] = {7, 6, 2, 3};
GLubyte face3[4] = {2, 6, 4, 0};
GLubyte face4[4] = {5, 4, 6, 7};
GLubyte face5[4] = {4, 5, 1, 0};
GLubyte face6[4] = {0, 1, 3, 2};
... // 材质及其他属性定义省略
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glNormalPointer(3, 0, normals);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, face1);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, face2);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, face3);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, face4);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, face5);
glDrawElements(GL_QUADS, 4, GL_UNSIGNED_BYTE, face6);
442
}
```

从中可见，绘制代码比原来的cube()函数压缩了很多。现在只需调用一个函数就可将顶点和法向数组发送到图形系统，绘制立方体每个面也只需要调用一个函数。而原来的标准立方体函数需要对每个顶点分别调用3次顶点和3次法向赋值函数，总共需要48次函数调用。这种方式不但可以节省函数调用以及将数据发送到图形处理系统的开销，而且可以使图形加速器可以利用其向量处理能力达到更高的绘制性能。



图12-7 使用顶点数组和法向数组绘制的立方体。参见彩图

本书并没有对这些OpenGL函数的语法细节做详细的介绍，否则本书将会变成OpenGL参考手册。但是值得注意的是，glVertexPointer()和glNormalPointer()函数具有相似的参数。此外，还有其他不同的绘制方式，前面已经介绍过。有关这些技术的细节，请读者参考OpenGL的手册。

12.5 碰撞检测

在第4章中介绍过碰撞检测技术。对由多边形组成的物体，物体间的碰撞检测可以归结为多边形之间的碰撞检测，并可进一步归结为三角形之间的碰撞检测。第4章中只对碰撞检测的基本知识进行了介绍，本章将着重介绍如何提高碰撞检测的性能。提高碰撞检测和性能最有效的途径就是，采用计算量不大的近似处理，而不是进行精确的计算。例如可以用物体的包

围体近似表示原物体，并用包围体进行碰撞检测，其碰撞检测结果则可以认为是原物体的碰撞结果。

在所有包围体类型中，包围球是最简单的一种形式。对此，碰撞后的运动方向可以从包围球的位置近似计算得到，也可以用离包围球上碰撞点最近的物体表面上的点到该碰撞点的向量作为运动方向。如果物体运动很快，不需要进行精确的碰撞行为计算，就可以使用包围球碰撞来代替物体间的碰撞，不需细化计算，观察者很难察觉其差别。

提高碰撞检测性能的第一步就是，判断那些碰撞情形是不可能的，从而避免不必要的计算开销。例如建立物体的包围体，首先判断包围体是否相交。如果包围体相交，才用实际物体进行相交测试，而这通常是对构成物体表面的三角形进行相交测试。如果将多边形分解为三角形，则三角形的相交测试又可进一步分解为边与三角形的相交测试。第4章中介绍过如何判断边与三角形的相交，即将边扩展表示为参数化直线，用该直线与多边形平面求交。如果满足以下一系列判断准则，则这条边与该多边形相交：

- 直线与平面的交点对应的参数值位于0和1之间。
- 直线与平面的交点位于三角形的外接圆中。
- 直线与平面的交点位于三角形中。

443

图12-8给出了上述比较判断过程的示意图。

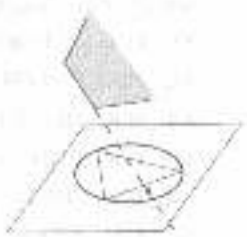


图12-8 碰撞检测计算

如果对运动的多面体进行碰撞检测，希望得到多边形碰撞的精确时刻，则需要对相交进行更多计算和处理。对此，需要在前一步（相交发生前）和当前步（相交已经发生）之间的时间段进行计算。可以对时间使用二分的方法来确定，判断相交发生在该时间段的前半段还是后半段，并且可以继续二分方法，直到对相交时间获得满意的估计。此外，如果物体在前一步和当前步的位置及速度为已知，则可以对相交时间进行精确的解析求解，这种方式还可以重新计算出物体反弹后的位置以及其他碰撞反应。

12.6 小结

本章介绍了一些简单直观的加速技术，相对于其他章节介绍的标准技术，采用这些技术的图形应用程序具有更快的图像绘制速度。虽然了解这些技术只是研究高性能图形的一个开始，但是它们完全可以让读者了解该问题的大致内部。恰当的运用这些技术有助于编写更简单的图形应用程序，从而获得更高的帧速率，并支持更快的交互响应速率。

12.7 本章的OpenGL术语表

本章介绍了顶点数组、雾化和绘制多边形序列等几个专用操作。对于普通图形应用程序也许没什么用，但对于包含大量预定义几何信息，或需要使用雾化以提高图像质量的应用程序，将会非常有用。下面列出最新的OpenGL术语。

OpenGL函数

444

glArrayElement（value）：指定绘制数组中的哪个顶点

glCullFace（param）：指定剔除多边形的正面或背面

glDrawElements（...）：使用数组数据绘制几何图元

glDrawRangeElements（...）：glDrawElements()的限定形式，只用位在指定范围内的元素

glEnableClientState（...）：启动客户端的功能，诸如顶点数组或法向数组

glFog*（param，value）：指定雾化参数及其具体的数值

glFrontFace（param）：定义多边形正面和背面的绕向，默认值为GL_CCW，即逆时针方向

glGetFloatv（...）：返回特定系统参数的数值

glNormalPointer（...）：定义法向数组

glVertexPointer（...）：定义顶点数组

OpenGL参数

GL_BACK：用于glCullFace()，指定剔除背面

GL_CURRENT_RASTER_DISTANCE：用于glGetFloat*()，返回视点到当前光栅位置的距离

GL_EXP：指定雾化模式为指数模式

GL_EXP2：指定雾化模式为指数平方模式

GL_FOG：glEnable()的参数，表示绘制雾化开始

GL_FOG_COLOR：glFog*()的参数，设置雾色

GL_FOG_DENSITY：glFog*()的参数，设置雾密度

GL_FOG_END：glFog*()的参数，设置雾化方程的结束距离

GL_FOG_HINT：glHint()的参数，设置绘制雾化时应用给定的提示。

GL_FOG_MODE：指定雾化模式，可选参数为线性、指数和指数平方

GL_FOG_START：glFog*()的参数，设置雾化方程的开始距离

GL_FRONT：用于glCullFace()，指定剔除正面

GL_FRONT_AND_BACK：用于glCullFace()，指定剔除正面和背面，即所有面，也就是不会绘制多边形，而只绘制线和点

GL_LINEAR：指定雾化模式为线性模式

12.8 思考题

1. 为什么将背面剔除看做一种重性能图形技术，而不是一种简化深度测试的方法？请比较下面两种方式所需的操作数量，一是进行背面测试，二是将图元发送到图形流水线并进行像素深度测试。

445 2. 使用细节层次技术，物体的某些细节被隐藏，直到观察者能看清楚为止。物体能否被看清楚通常用其尺寸来定义，因此当物体在图像平面上的大小大于一定数量的像素时，它将突然变为可见。请讨论这种方法可能引起的变化

a. 如果物体背景为纯中性色彩（如灰、黑或白色——译者注），

b. 如果物体背景为带自然场景的纹理贴图墙面，

c. 如果物体的颜色最初配合较低α值显示，并且在靠近视点时，α值不断增大。

3. 雾化将场景中的物体隐藏，直到它距离视点足够近，即小于雾化的最大距离，因此可以将其归属为LOD技术。使用该技术，超过最大距离时，物体将不可见（不绘制），直到小于最大距离才进行绘制。请讨论这种方法是否能较好地避免物体突然出现。

4. 使用场景图设计一个场景，在地模空间中不同位置放置多个几何图元，使得在某些视点图元会相互覆盖。手动创建场景的BSP树，并编写程序，在关闭深度测试的情况下，按BSP树中的图元顺序绘制场景图，其绘制结果应该和启动深度测试时的绘制结果一样，即前面的图元不会被后面的图元覆盖。

12.9 练习题

1. 创建同一几何物体的两个或多个模型，使得它们看起来相似，但一个包含大量多边形，而另一个只包

含少量多边形，创建一个场景，并将该物体放置于该场景中。当物体远离视点时，绘制具有少量多边形的模型，反之则绘制具有大量多边形的模型。两个模型之间的显示切换是否平滑到你会在实际应用中采用这个模型？为什么可用？为什么不可用？

2. 与思考题4类似，通过编程实现BSP树的自动创建，根据该BSP树实现和控制画家算法来绘制场景。修改代码，测试BSP树，在绘制过程中设置图元的顺序，从而可以根据视点变化，自动调整BSP树。

12.10 实验题

1. 以下面两种方式构建一个场景：一是使用传统的顶点、法向和纹理坐标的多边形，二是将相同数据存储在顶点数组（分别为GL_VERTEX_ARRAY、GL_NORMAL_ARRAY、GL_TEXTURE_COORD_ARRAY）中。以这两种方式绘制场景，测试其运行时间的差别。为了便于比较测试结果，通常需要创建规模较大的场景，因此可以使用算法自动生成几何数据，请参考第14章中介绍的“伪分形”地形场景方法。
2. 构建一个场景并将其组织成多边形的列表，利用本章介绍的OpenGL工具自动测试每个多边形到视点的距离，按照该距离对多边形排序，并且关闭深度测试，按排序结果绘制多边形，然后允许深度测试，并按任意顺序绘制多边形列表，比较两种方法的绘制结果，哪种方法速度更快？哪种方法质量更好？ 446
在测试到视点的距离时，增加背面剔除，去除列表中背向视点的多边形，再次比较其结果，是否有差别？差别是否明显？
3. 使用GLUT和GLUT中的物体创建一个简单场景，并使用空间划分组织场景，从而实现从前到后绘制场景，与前面创建多边形列表的方式进行比较，这种方法的难度有多大？

12.11 大型作业

1. 实现一个台球模型，使用碰撞检测处理台球之间的相互碰撞，以及台球与球台栏板之间的碰撞。实现台球与栏板之间的能量转换物理学原理，碰撞后方向的变化，以及台球受到摩擦力影响随时间而减速，不考虑碰撞受旋转的影响。注意，球体的碰撞检测非常容易实现。 447

第13章 插值与样条建模

本章介绍一种新的图形建模技术，可以用少量控制点来定义几何形状，创建复杂的曲线和曲面。这种技术使用一组特殊插值函数的线性组合对控制点进行插值，从而得到完整的曲线和曲面。该技术简单直观，使用灵活（掌握该技术只需编写少量程序或掌握某种图形API）。这种插值技术已在动画一章中提到过（第11章），并用于漫游程序创建平滑的视点轨迹。这种技术是一种最基本的交互式建模技术，其最主要的应用为通过交互式的操作控制点来创建曲线和曲面。

除了建模功能外，这种插值技术还可以根据插值函数参数，方便地为插值的几何形状生成法向和纹理图。在创建复杂曲线和曲面的同时，可以生成完整光照效果和纹理图。在本书中，这类插值技术是创建复杂几何形状最强大的工具。

为了理解本章的内容，读者需要具有一定的参数化函数知识，尤其是双参数函数，以及基于三角形条带之类的简单几何建模知识。

13.1 引言

第4章解释了如何将直线段看做两个端点的线性插值，但没有给出准确的表示方法。本章
448 将介绍点的其他插值技术，包括样条曲线和样条曲面。我们将详细介绍Catmull-Rom样条和Bézier样条。虽然这些样条技术很简单，但本章仅限于讨论一维样条曲线。将一维扩展到二维用于曲面建模稍微复杂一些，我们将借助OpenGL图形API中的求值器函数介绍其主要思想，样条以及其他插值曲线和曲面技术研究内容十分丰富，但本书不进行深入的探讨。

样条技术应用广泛，可用于创建平滑曲线来逼近一维域中的点（一维插值），或创建平滑曲面来逼近二维域中的点（二维插值）。样条的插值功能主要用于几何模型，但本章后半部分也将介绍样条的其他用途。图形API如OpenGL，一般提供用一组给定的点创建样条插值的功能，这些点称为控制点。

通常，一条完整的样条曲线或一张完整的样条曲面可看做场景图中独立的几何元素，它们定义在单独的建模空间中，具有一组独立外观参数。尽管它们比较复杂，但通常表示为一个单独的形状节点，对应于场景图中的一个叶节点。

13.1.1 插值

第4章介绍过直线段的参数化形式，该形式将点P_0和P_1之间直线段上任意位置的点与参数t建立一一映射，其中参数t位于单位直线段[0,1]之间。实际上这是通过构造一直线段来对两个点进行插值。该插值线段可以表示为直线段的参数化形式：

$$(1-t)P_0+tP_1，\text{其中}t\text{位于}[0,1]\text{之间}$$

该表达式应用价值不大，但具有重要的启发意义。它启发了插值两个点的通用方式，即通过构造如下函数来计算新的插值点：

$$f_0(t)P_0+f_1(t)P_1$$

其中f_0和f_1为具有一定关系的设定函数。需要考虑点与插值函数的关系，并验证相应的插值结果。通常，插值函数具有以下特殊性质：

- 函数定义在单位区间上，使得参数t位于标准定义域。
- 在定义区间上，函数值为非负值。
- 对任意参数值t，所有函数值的和一般为1，即插值点为原始点的凸点和。

插值函数在参数定义区间的0、1端点处一般都有值，使得在$t=0$时，插值结果为P_0，在$t=1$时，插值结果为P_1。

对于最开始介绍的插值线段，其插值函数分别为$f_0(t)=(1-t)$和$f_1(t)=t$，具有上述性质，显然，$f_0(0)=1$，$f_1(0)=0$，因此，当$t=0$时插值结果为P_0；$f_0(1)=0$，$f_1(1)=1$，因此，当$t=1$时插值结果为P_1。可见插值点始于P_0，止于P_1。对参数定义区间上的任意t值，有

$$f_0(t)+f_1(t)=(1-t)+t=1$$

449

在参数区间[0,1]上函数f_0和f_1的取值都为非负。由于插值函数为参数t的线性函数，最终的插值点组成了一条直线段。

作为两点间直线段插值的推广，将插值应用于对给定的一组控制点进行插值，按序求出一组插值点来逼近控制点间的空间。给定控制点可以为三个点、四个点甚至更多。在以后的讨论中，假设所有点都位于三维空间，因此得到的插值曲线（以及插值曲面）均为三维，如果读者希望使用二维插值，忽略三维坐标中的一个分量即可。

插值三个点P_0、P_1和P_2比插值两个点更为有趣，可以提出更多不同的插值方法。例如，将三个点放置在圆上，因此有时也把圆看做为一种插值。将这种思路扩展到参数化直线的插值，可以得到以下形式的二次插值：

$(1-t)^2P_0+2t(1-t)P_1+t^2P_2$，其中$t$位于[0,1]之间。

其中使用了三个插值函数f_0、f_1、f_2，分别表示为：

$$f_0(t)=(1-t)^2$$
$$f_1(t)=2t(1-t)$$
$$f_2(t)=t^2$$

至此，可以看到这些插值函数十分重要。在以后的讨论中，这些函数称为插值基函数，而点P_0、P_1和P_2称为插值控制点（在样条曲线研究领域中，称为结点，并且把端点称为插值连接点）。上述三个插值函数与前面介绍的线性基函数具有类似的性质，即$f_0(0)=1$，$f_1(0)=0$，$f_2(0)=0$，以及$f_0(1)=0$，$f_1(1)=0$，$f_2(1)=1$，结果为一个平滑的二次插值函数，当$t=0$时，插值结果为P_0，当$t=1$时插值结果为P_2。当t在0和1之间任意取值时，该插值函数为三个点的线性组合，图13-1给出了三个点和对应的插值曲线。



450

图13-1 三个点的二次插值曲线

有意思的是，前面介绍的两种插值基函数都具有相同出处，对于线性插值，函数$f_0(t)$和$f_1(t)$分别为下式的两个展开项，

$$((1-t)+t)^1$$

对于二次插值，函数$f_0(t)$、$f_1(t)$和$f_2(t)$分别为下式的三个展开项，

$$((1-t)+t)^2$$

对于这两种插值，给定t值，所有基函数加起来都等于1。若将每个插值点前的系数看做权重，则这两种插值的所有插值权重之和为1。

根据线性插值和二次插值，可以扩展到更一般的插值方法，N次插值多项式即对以下多项式的所有项进行插值

$$((1-t)+t)^3$$

需要插值的几何数据作为多项式每一项的系数。根据这种方式可以构造三次插值($N=3$)，插值四个点P_0、P_1、P_2和P_3的插值函数如下所示：

$(1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3$，其中$t$位于[0,1]之间

图13-2显示了四个点的插值曲线形状（为了便于比较曲线形状，前三个点与前面二次样条中的三个点一样）。实际上，该曲线就是插值四个控制点的标准Bézier样条函数的表示，其中的四个多项式

$$f_0(t) = (1-t)^3$$
$$f_1(t) = 3t(1-t)^2$$
$$f_2(t) = 3t^2(1-t)$$
$$f_3(t) = t^3$$



图13-2 基于伯恩斯坦基函数的插值四个点的Bézier样条曲线

称为样条曲线的三次伯恩斯坦（*Bernstein*）基（其代码和本章中的其他图表都包含在本书附带的资料中）。

451

该插值Bézier样条曲线经过四个控制点中第一个和最后一个控制点（P_0和P_3），但不经过中间两个控制点（P_1和P_2）。这是因为其基函数在$t=0$和$t=1$时具有与二次样条相同的性质，即$f_0(0)=1$，$f_1(0)=0$，$f_2(0)=0$，$f_3(0)=0$，以及$f_0(1)=0$，$f_1(1)=0$，$f_2(1)=0$，$f_3(1)=1$。如果计算Bézier样条函数关于t的导数，可以得到$t=0$时导数为$3(P_1-P_0)$，在$t=1$时导数为$3(P_3-P_2)$。因此，当曲线经过第一个控制点时，其运动方向与第一个控制点到第二个控制点的方向相同；当曲线经过最后一个控制点时，其运动方向与第三个控制点到第四个控制点的方向相同。曲线形状取决于中间两个控制点，它们确定了曲线的起始和终止方向。此外，基函数作为权重也会影响曲线形状，并且曲线的平滑性取决于基函数的平滑性。

以上定义插值基函数的方式并非唯一，还有其他方式可以定义适当的插值函数。一般而言，插值曲线不一定必须经过给定的点，但这些点将以某种方式影响和决定曲线性质。如果的确需要曲线经过控制点，可以采用其他样条表达式。Catmull-Rom三次样条具有以下形式：

$f_0(t)P_0 + f_1(t)P_1 + f_2(t)P_2 + f_3(t)P_3$，其中$t$位于0，1之间

其基函数为

$$f_0(t) = (-t^3 + 2t^2 - t)/2$$
$$f_1(t) = (3t^3 - 5t^2 + 2)/2$$
$$f_2(t) = (-3t^3 + 4t^2 + t)/2$$
$$f_3(t) = (t^3 - t^2)/2$$



图13-3 使用Catmull-Rom三次样条插值四个点

该插值曲线与Bézier曲线具有不同性质，它只经过第二个和第二个控制点，如图13-3所示。该性质差异主要来自基函数的差异，$f_0(0)=0$，$f_1(0)=1$，$f_2(0)=0$，$f_3(0)=0$，以及$f_0(1)=0$，$f_1(1)=0$，$f_2(1)=1$，$f_3(1)=0$。从中可见在两端点处，该曲线只插值P_1和P_2，并经过它们，而与P_0和P_3无关，也不经过这两个点。后面将扩展该样条曲线到更多控制点上，并实现对每一对控制点的插值。因此，当要求插值曲线经过所有控制点时，Catmull-Rom样条曲线就非常有用。

452

虽然可以使用任意阶的多项式，但我们不讨论二次以上的插值样条曲线。在简单样条中，

三次样条使用最广泛，是非常有用的建模工具。目前给出的示例都为二维曲线，在后面的示例中将看到，如果插值的控制点位于三维空间，将得到一条三维曲线，即从直线投到三维空间的函数变换。本章后面部分将讨论这些三维曲线。

13.1.2 另一种Bézier样条的基本概念

表达样条曲线的方式有多种，相关理论非常丰富。除了用伯恩斯坦基，Bézier样条还可以以另外一种方式阐述，即用四个控制点描述一段曲线。这种方式可以从四个点精确地勾勒出Bézier曲线的大致形状。

Bézier样条曲线按如下方式连接控制点的端点：从第一控制点出发，朝第二个控制点的方向前进，并沿第三个控制点的方向，进入到第四个控制点。实际上，曲线在第一个控制点处的斜率等于连接第一个和第二个控制点的直线斜率，曲线在第四个控制点处的斜率就等于连接第三个和第四个控制点的直线斜率。样条函数在两端点处的速度分别等于位于第一个和第二个控制点之间的向量，以及位于第三个和第四个控制点之间的向量的三分之一。以上事实可用于确定三次曲线等式中的四个系数，具体计算细节留作本章练习。

13.1.3 另一种Bézier样条计算方法

以基函数和控制点表示的Bézier样条曲线形式如下所示：

$$(1-t)^3P_0 + 3t(1-t)^2P_1 + 3t^2(1-t)P_2 + t^3P_3$$

该形式不是我们所熟悉的多项式形式，若将其分解合并，可得到如下形式：

$$(-P_0 + 3P_1 - 3P_2 + P_3)t^3 + (3P_0 - 6P_1 + 3P_2)t^2 + (-3P_0 + 3P_1)t + P_0$$

由于每个控制点为三维，该多项式实际由三个联立方程组组成，分别对应x，y和z坐标。相比基函数方式，这种方式的编程容易实现。

基于该多项式表示方式，可以进一步利用矩阵性质表示样条函数，该表达式可写成

$$\begin{bmatrix} P_0 & P_1 & P_2 & P_3 \end{bmatrix} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} t^3 \\ t^2 \\ t^1 \\ t^0 \end{bmatrix}$$

453

该矩阵操作的维数分别为（1×4）－（4×4）－（4×1），最终得到一个点。将基函数转换为矩阵表达十分有用。实际上，对于已知基函数的任何三次样条函数，都可以按此方式得到类似的样条矩阵表示形式。

13.1.4 扩展插值到更多控制点

前面我们只讨论了对少数控制点的插值及其结果，这些插值方法很容易扩展到更多控制点，但具体扩展方式依赖于所使用的插值类型。

Bézier曲线经过第一个和最后一个控制点，但不经过中间两个控制点。如果首先使用前四个控制点，再使用后面紧接的三个控制点（前面四个控制点的最后一个加上其后的三个控制点），按照这种方式，直到最后一个控制点，那么得到的插值曲线将是连续的，并且每隔三个控制点就经过一个控制点（第一个，第四个，第七个，等）。但该曲线会在与控制点连接处突然改变方向，如图13-4所示，其中显示了由多个控制点生成的不平滑曲线。

为了使整个曲线平滑变化，需要增加更多控制点，这些新增控制点使曲线进入一组控制点的最后一个控制点的方向，与曲线离开下一组控制点的第一个控制点的方向相同。具体而言，在每一对索引为$2N$和$2N+1$的控制点之间都将新增一个控制点，其中$N \geqslant 1$，并且不包括最后一对控制点。新增的控制点可以定义为这些点对的中点，即表示为$Q_N = (P_{2N} + P_{2N+1})/2$。通过增加新的控制点可以得到新的控制点序列，与原始的控制点序列关系如下所示：



图13-4 没有平滑过渡的两段曲线

原始序列：P_0 P_1 P_2 P_3 P_4 P_5 P_6 P_7

454 新的序列：P_0 P_1 P_2 Q_0 P_3 P_4 Q_1 P_5 P_6 P_7

其中Q表示新增控制点，取值为其前后两个控制点的平均。插值计算使用的控制点序列为：$P_0-P_1-P_2-Q_0$、$Q_0-P_3-P_4-Q_1$和$Q_1-P_5-P_6-P_7$。请注意，Bézier曲线的控制点必须为偶数，只有当控制点数大于六个，才需要扩展原始控制点，扩展后，曲线第一段和最后一段有三个原始控制点，其余曲线段只有两个原始控制点。图13-5给出了具有这种连接关系的曲线示例，其中的曲线不仅处处连续，而且处处可微。

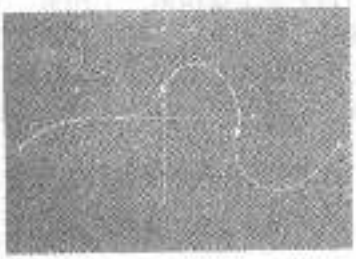


图13-5 通过增加中间控制点（以大点表示）扩展Bézier曲线

对于三次Catmull-Rom样条，其插值曲线只连接控制点P_1和P_2，因此具有不同的扩展方式。由于插值P_0-P_3的曲线段不经过起始控制点P_0和结束控制点P_3，因此，如何开始曲线具有一定困难。对此，我们从三部分来考虑整个插值曲线，即开始曲线段、中间曲线段和结束曲线段。

开始曲线段的构造很简单：重复第一个控制点两次。从而第一组控制点包括P_0、P_0、P_1和P_2。由于P_0和P_1为中间两个控制点，所以第一段插值曲线经过这两个控制点。结束曲线段的构造类似，即将最后一个控制点重复两次。则最后一组控制点为P_1、P_2、P_3和P_3，并且曲线段经过中间两个控制点P_2和P_3。如果将这种扩展方法应用到前面介绍的四个控制点的示例中，将得到三段插值曲线段，结果如图13-6左图所示。



图13-6 通过重复端点（左）和沿原始点新增控制点（右）扩展Catmull-Rom曲线

如果控制点更多，希望扩展插值曲线使其经过所有控制点，则可以使用控制点的“滑动集”方法，即从P_0、P_1、P_2和P_3开始，逐点向后移动，新控制点组包含前一个曲线段的后三个控制点，并增加一个控制点。按照这种方式，第二组控制点为P_1、P_2、P_3和P_4，接着是P_2、P_3、

455 P_4和P_5，等等。这种滑动集方法实现简单（可以使用包含四个点的数组，每移动一次使索引值减1，即P[1]变成P[0]，P[2]变成P[1]，P[3]变成P[2]，新增控制点存放到P[3]）。对于点序列P_0-P_5，每段曲线（包括头尾两曲线段）的控制点组分别是$P_0-P_0-P_1-P_2$、$P_0-P_1-P_2-$

P_3，$P_1-P_2-P_3-P_4$，$P_2-P_3-P_4-P_5$，$P_3-P_4-P_5-P_6$，$P_4-P_5-P_6-P_7$，$P_5-P_6-P_7-P_8$，$P_5-P_7-P_8-P_8$。按这种方式构造的插值曲线如图13-6右图所示，其控制点集合与图13-5中的扩展Bézier样条曲线相同。

13.2 样条曲面

前面介绍的样条曲线技术很容易扩展，用于生成插值曲面。样条曲面具有与样条曲线类似的性质：由一组控制点指定，可移动控制点按照预期方式改变曲面形状，只要方法恰当，曲面一般是平滑的，可以表示各种形状。实际上，样条建模通常是指样条曲面建模，而不是样条曲线建模。本章首先介绍样条曲线，这有助于理解样条曲面。

假设函数$f_0(t)$，$f_1(t)$，$f_2(t)$，$f_3(t)$为三次样条的基函数，使用两个参数u和参数v，其中$0\leq u$，$v\leq 1$，结合16个控制点P_{ij}，其中i，j取值0到3，从而得到一个双变量函数：

$$f(u,v)=\sum_{i=0}^{3}\sum_{j=0}^{3}f_i(u)f_j(v)P_{ij}$$

其中函数值对16个控制点i，j取和。根据该函数，可以按任意方式改变参数u和v，从而绘制该曲面，其方式与第9章中的绘制双变量函数的方式类似。示例代码如下所示，其中插值函数F0，F1，F2和F3为前面介绍过的Bézier曲线的三次伯恩斯坦基函数。 456

```
float u, u1, v, v1, ustep, vstep, x, y, z;
float cp[4][4][3];    // 三维控制点的4×4数组

for (u=0.; u<1.; u+=ustep)
    for (v=0.; v<1.; v+=vstep) {
        u1 = u+ustep; v1 = v+vstep;
        beginQuad();
            vertex3(eval(u,v,0),eval(u,v,1),eval(u,v,2));
            vertex3(eval(u,v1,0),eval(u,v1,1),eval(u,v1,2));
            vertex3(eval(u1,v1,0),eval(u1,v1,1),eval(u1,v1,2));
            vertex3(eval(u1,v,0),eval(u1,v,1),eval(u1,v,2));
        endQuad();
    }

float eval(float u, float v, int i)
{
    float result = 0.;
    result += F0(u)*F0(v)*cp[0][0][i]+F1(u)*F0(v)*cp[1][0][i];
    result += F2(u)*F0(v)*cp[2][0][i]+F3(u)*F0(v)*cp[2][0][i];
    result += F0(u)*F1(v)*cp[0][1][i]+F1(u)*F1(v)*cp[1][1][i];
    result += F2(u)*F1(v)*cp[2][1][i]+F3(u)*F1(v)*cp[3][1][i];
    result += F0(u)*F2(v)*cp[0][2][i]+F1(u)*F2(v)*cp[1][2][i];
    result += F2(u)*F2(v)*cp[2][2][i]+F3(u)*F2(v)*cp[3][2][i];
    result += F0(u)*F3(v)*cp[0][3][i]+F1(u)*F3(v)*cp[1][3][i];
    result += F2(u)*F3(v)*cp[2][3][i]+F3(u)*F3(v)*cp[3][3][i];
    return result;
}
```

函数eval（...）计算给定参数u和v的插值点的第i个坐标分量。而第一段代码访问参数空间，生成并绘制四边形。没有指定四边形的外观属性，但可以使用前一章介绍的颜色、光照或着色处理等操作进行处理。本章后面将给出三维样条曲面的示例。

13.2.1 扩展曲面片为曲面

本章前面介绍了将Bézier样条曲线从四个控制点扩展到任意偶数个控制点仍然保持曲线平滑，需要增加一些控制点，新增控制点是原始控制点之间直线段的对分点。扩展Bézier样条曲面也存在类似的情形。为了将曲面从单个曲面片扩展到任意二维控制点的数组（数组每一维都具有偶数个点），并且使曲面平滑，同样需要新增控制点。这些新控制点将增加到与曲线中类似的位置：首先在第3个控制点后增加一个控制点，然后在每两个控制点后增加一个控制点，

直到新控制点后面只存在3个原始控制点为止。并且在两个方向同时增加控制点，从而得到新
457 控制点的行和列。对于扩展曲面，需要处理一种新情形：即在控制点的新行和新列相交的角点处必须新增一个控制点，并且该新增控制点必须取与该角点相邻的原始4个控制点的平均值。通过新增控制点，可以对每个4×4的控制点组生成一个曲面片，并且使最终的曲面在所有方向都可微，在进行光照处理时，这个性质对于解析生成曲面片法向十分重要。

13.2.2 生成曲面片法向

由于任何曲面片都可以表示为基函数的双线性组合，因此对于曲面片上任意一点，都存在解析表达式。尤其对于绘制曲面片时所需的网格上的顶点，可以根据该曲面的解析函数计算其两个方向的导数。它们的简化公式如下所示，其中f_i和f_j为单变量函数：

$$\partial f(u,v)/\partial u = \sum_{i=0}^{3}\sum_{j=0}^{3} df_i(u)/du * f_j(v) * P_{ij}$$

$$\partial f(u,v)/\partial v = \sum_{i=0}^{3}\sum_{j=0}^{3} f_i(u) * df_j(v)/dv * P_{ij}$$

这些导数是相对于参数空间坐标系，而不是相对于世界空间坐标系。根据这些导数，可以求出切线和切平面，可以计算曲面上任意一点$f(u, v)$的两个切向量，通过计算这两个向量的叉积并将其归一化，可以得到曲面在该顶点的单位法向。通过这种方式，可以把法向增加到顶点列表中，从而可以对曲面应用完整的光照模型。

13.2.3 生成曲面片纹理坐标

在生成样条曲面的曲面片时，需要在二维单位参数空间上生成顶点网格。由于该网格本身定义在二维空间上，因此很容易将其关联到其他二维空间网格，如二维纹理坐标空间。与上述向顶点列表增加法向类似，也可以将纹理坐标增加到曲面片的顶点列表中，在此解释一种最简单的情况，对于由函数$f(u, v)$确定的每个顶点，该函数是从二维（u，v）空间到三维空间的映射，将该顶点的参数u，v映射到其对应的纹理坐标U，V，从而得到一个纹理映射函数，该函数将纹理坐标（U，V）与曲面上的点一一关联。当然，如果希望纹理能很好地映射到曲面上，那么相邻控制区域的曲面片也必须使用相邻的纹理区域。

13.2.4 另一种曲面片计算方法

前面介绍过如何用矩阵方式表示三次样条曲线，实际上双三次样条曲面也可以用矩阵方式表示。基于前面的讨论，其扩展方式非常简单，假设M表示对应基函数的4×4的矩阵：

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 6 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

458

G（代表几何）表示4×4的控制点P_{ij}的数组，$E(w)$（代表指数）表示参数w的指数的4×1的数组：

$$G = \begin{bmatrix} P_{00} & P_{01} & P_{02} & P_{03} \\ P_{10} & P_{11} & P_{12} & P_{13} \\ P_{20} & P_{21} & P_{22} & P_{23} \\ P_{30} & P_{31} & P_{32} & P_{33} \end{bmatrix} \quad \text{and} \quad E(w) = \begin{bmatrix} w^3 \\ w^2 \\ w^1 \\ w^0 \end{bmatrix}$$

则整个曲面片可表示为：

$$E(u)^T M^T G M E(v)$$

其中$E(u)^T$和M^T的上标代表矩阵转置。这种基于矩阵的方式能紧凑而有效地表示曲面，其编程实现也很容易。

13.3 其他类型的插值函数

Bézier样条和Catmull-Rom样条使用简单并且应用广泛。还有一些其他类型的样条可用于专业的曲线和曲面建模，例如，*B*样条函数比Bézier样条更通用（具体而言，Bézier样条是B样条的一种特例），具有更多有用的性质，但是B样条的计算比Bézier样条复杂，而且OpenGL仅只支持其特例NURBS。

*有理函数*是比多项式函数更通用的插值函数，可定义为两个多项式的商。若两个多项式均可表示为参数形式，则该有理参数函数可用于表达B样条曲线无法表达的形状，诸如标准的圆以及其他二次函数。非均匀有理B样条简称NURBS，是有理函数的一种，其混合函数为分子和分母均为B样条的有理函数。非均匀有理B样条功能强大，广泛应用于专业设计领域，但是本书将不对其进行介绍。GLU工具集支持这种复杂的技术，只有深入理解了NURBS，才能灵活运用这种技术。如果读者对曲线和曲面设计十分感兴趣，这是值得继续研究的方向。

13.4 OpenGL中的插值

在OpenGL中，求值器函数提供样条功能。输入一组控制点到求值器，能够生成一组插值该控制点的另一组点。因此，只需通过设置控制点和求值器，就可以得到精细的曲线和曲面，从而完成曲线和曲面建模。

OpenGL中有两类求值器：一维求值器和二维求值器。一维求值器可用于插值点以生成单参数信息（即只有一个自由度的曲线或其他数据类型，如一维纹理和一维几何曲线）。二维求值器可用于插值二维点数组以生成双参数信息（即具有两个自由度的曲面或其他数据类型，如二维纹理和几何曲面）。OpenGL的两类求值器实现了前面讨论过的Bézier三次样条。这两种求值器使用都很方便，能够为最终显示的曲线和曲面设置细节程度。 459

从图13-7到图13-9给出了使用OpenGL求值器定义的几何形状的示例。图13-7显示了由一维求值器定义的空间曲线的两个不同视图，其中显示了30个控制点，也显示了用于平滑曲线新增的控制点。图13-8显示了用二维求值器根据4×4的控制点组定义的单个曲面片。图13-9显示了由16×16的控制点组定义的曲面，新增的控制点没有显示。13.6节中给出了生成这些图的部分代码。

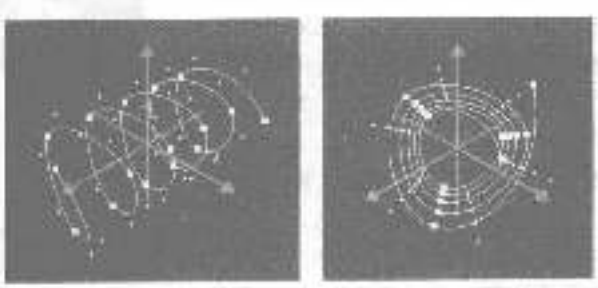


图13-7 使用一维求值器定义的样条曲线。视点为x = y = z（左），旋转以显示控制点和曲线形状的关系（右）。小的控制点为原始控制点，大的控制点为新增控制点，新增方式与前面介绍的扩展Bézier曲线方法相同

图13-8中的样条曲面的α值为0.7，因此可以看见位于曲面片后的控制点和其他部分。在这些例子中，请观察控制点和实际曲面的关系。其中只有四个角点与实际曲面相交，所有其他控制点均偏离曲面，但它们却能影响曲面片的形状。此外，整个曲面片均位于所有控制点的凸包内。前面介绍的Bézier样条也具有该性质。曲面片上的高光有助于根据光源观察曲面形状。对于图13-9中的大曲面，曲面在不同控制点组之间平滑扩展。

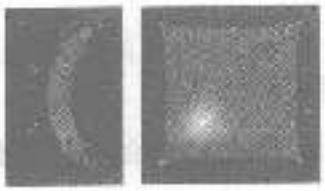


图13-8 使用二维求值器和四个控制点定义的一块样条曲面片

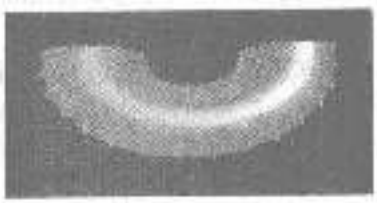


图13-9 使用二维求值器和16×16的控制点组定义的样条曲面。用原始控制点对插值点进行扩展

上述示例完整地使用了求值器的曲线和曲面生成功能，即根据控制点生成整个曲线或曲面。但有时只需生成曲线或曲面上对应一个或多个给定参数的单个点。例如，对于已经构建好的模型，在“浏览”或“漫游”该模型时，要沿着由给定的控制点定义的曲线来定位视点位置。为了沿着曲线上的视点生成对应的图像，就需要根据曲线上对应的参数值来计算视点位置，即根据参数对曲线进行求值以计算每个视点的位置，得到相应的几何坐标，从而赋予gluLookAt（...）函数。

460

13.4.1 使用求值器自动生成法向和纹理

图13-8和图13-9中显示的图像包含完整的标准光照模型以及高光。实际上，并没有使用显式的顶点定义，也没有使用显式的法向定义，而是通过二维求值器自动生成顶点坐标以及顶点法向，读者可以参考本章后面将给出的示例代码。其中的关键部分主要包括以下几点：

- 指定16个控制点的数组。
- 启动GL_MAP2_VERTEX_3，表明用二维映射生成三维空间中的点。
- 启动GL_AUTO_NORMAL，表明法向由求值器解析生成。
- 使用glMap2f(...)指定二维映射参数，指定使用以上控制点数组。
- 使用glMapGrid2f(...)指定如何使用二维定义域空间来生成用于绘制的顶点网格。
- 使用glEvalMesh2(...)执行求值计算并显示曲面，如果只要计算曲面上对应参数值（u，v）的点，可使用glEvalCoord2f(u，v)得到该点的坐标。

461

实际上，OpenGL有很多类似的函数，用于在维度不同的空间执行类似的操作，请读者参考OpenGL手册查找相关细节。

除了能自动生成法向外，OpenGL求值器还具有为曲面生成其他相关信息的功能，如为求值器曲面自动生成纹理坐标。举个例子，启动GL_MAP2_TEXTURE_COORD_2，并使用glMap2f(...)函数，给定该函数第一个参数为GL_MAP2_TEXTURE_COORD_2，其他参数与生成顶点坐标中一样，然后使用glEvalMesh2(...)函数可得到二维曲面片上纹理的s和t坐标。上述过程看起来有点复杂，但分析完一段代码之后将变得容易，相关代码请参考本章后面给出的样条曲面示例。此外，从一维或二维网格生成一维到四维纹理由许多不同之处，具体细节请参考OpenGL手册。图13-10中的图像是从图13-8中的图像扩展得到的，将曲面颜色改为白色，并使用自



图13-10 使用自动纹理坐标生成功能得到的图13-8中的曲面片的纹理图

动生成的纹理坐标将纹理进行混合。这两个图中的自动法向和纹理坐标生成的代码将在本章后面给出。

除了上述功能外，还可以使用glMap2f(...)函数和GL_MAP2_NORMAL参数为曲面自动生成法向，或者使用glMap2f(...)函数和GL_MAP2_COLOR_4参数为曲面自动生成颜色。

13.4.2 其他技巧

除了建模以外，样条技术还可用于生成平滑过渡的颜色、法向或纹理坐标，或者用于插值得到其他数据类型。但是不同于生成曲线和曲面，OpenGL中没有内置函数用于自动应用这 462
些插值点。因此，这些参数化函数需要单独管理。为此，需要给 (u, v) 参数空间中的每个点定义一个数值，然后使用glEvalCoord1f (u) 或glEvalCoord2f (u, v) 函数从求值器得到实际插值点，接着就可以像处理其他已经定义的点那样使用这些插值点。根据生成图像的具体要求，这些点可以用于表示颜色、法向或纹理坐标。

具体而言，假设一个曲面由两个参数定义，每个参数位于[0，1]之间，为了定义曲面上的法向以达到期望的光照效果，可以首先定义一组控制点逼近所需的法向（一个法向为一个三维向量，即一个点）。然后定义一个求值器用于创建这些控制点对应的新曲面，并建立从原始曲面参数到新曲面参数的映射。为了得到参数坐标 (u, v) 对应点的法向，只需要在新曲面上找到对应的参数坐标 (u', v')，然后通过函数 $f(u, v)$ 得到原始曲面上的点坐标 (x, y, z)，再通过函数glEvalCoord2f (u', v') 得到新曲面上的点坐标 (r, s, t)，最后可以在以下函数调用中应用这些顶点的坐标数据，绘制生成图像：

```
glNormal3f (r, s, t); glVertex3f(x, y, z);
```

样条函数还可以应用于动画，用于得到视点轨迹的平滑曲线，在动画一章（第11章）中已经简单介绍过。由于视点在运动，为了给出完整的视图变换，还需要处理其他问题。其中向上向量很容易处理，对于简单的动画，一般保持该向量不变即可。观察中心稍微复杂些，为了保持动画运动的真实感，观察中心也需要做相应变化。本书建议的方法是在样条曲线上取三个点，即前一个点、当前点和下一个点，用前一个点和下一个点来定义观察方向，而观察中心则为距离当前点固定距离的一个点，按观察方向观察。这种方式可以产生比较合理的运动和观察配置。在动画中，样条还可用于平滑移动模型的不同部分以产生逼真的运动。

13.5 定义

正如图13-7和图13-8中所示，OpenGL求值器作用于4个控制点（一维）或4×4的控制点组（二维），并且只经过这些控制点中的端（角）点，而不经过其他控制点。在这些端（角）点处，曲线的切线平行于从该点到其邻接控制点之间的直线段，如图13-11所示，其速度由该点到其邻接控制点之间的距离决定。

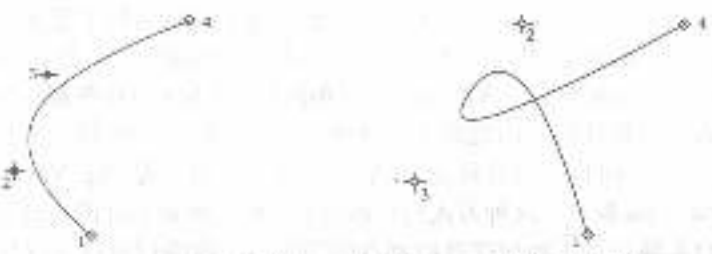


图13-11 两条样条曲线显示曲线通过不同的控制点排布

在扩展样条曲线时，为了控制形状，需要对控制点进行调整，使得对应控制点到其邻接
463 控制点的方向和距离都一样。这可以通过新增控制点到恰当的原控制点之间来实现。具体步骤已经介绍过。从而使得曲线从第一个端（角）点平滑移动到第一个新增控制点，到第二个新增控制点，再到第三个新增控制点，按照这种方式直到最后一个端（角）点。

该构造过程及其内在关系如图13-7中大的新增控制点所示。除了第一个和最后一个控制点外，在每两个原始控制点后就新增一个控制点。新增控制点将其两个端点之间的插值线段对分，曲线仅与新增控制点相交，而不与原控制点相交，两个端点除外。在编写交互式应用程序，使用户能方便地操纵控制点来调整曲线形状时，只需允许用户操纵原始控制点即可，因为新增控制点用来定义曲线，只是平滑曲面的一种具体实现方法，不必让用户操纵。在用户改变控制点后，重新计算曲面信息并对其重新绘制。

类似地，对二维求值器的控制网格可以增加控制点，用于创建更丰富的曲面片，实现曲面片之间的过渡。其方式也要遵循曲面片的边之间的线段等距离和同方向的原则，这样才能使曲面从一个曲面片平滑过渡到另一个曲面片。13.6节给出了其中的关键代码，根据具体曲面片的位置应进行不同的处理。具体细节请参考后面的示例代码。

那么，样条曲线和样条曲面如何产生？三次样条曲线由参数变量u的三次多项式决定，其公式如下所示，参数u取值位于0到1之间。

$$\sum_{i=0}^{3} a_i u^i$$

四个系数a_i可以通过曲线的四个约束条件确定。而这4个约束条件可以分别从4个控制点得到，这些控制点用于确定一段三次样条曲线。前面介绍过如何从4个基本多项式和控制点确定的系数来计算对应的四个数值。OpenGL中的一维求值器可根据Bézier曲线定义计算这四个
464 系数，并计算该多项式的值，从而生成曲线上的点或曲线本身。

双三次样条曲面由参数变量u和v的双三次多项式决定，其公式如下所示，其中参数u和v取值在0到1之间。该公式中有16个系数a_{ij}需要计算，可以从16个控制点得到，这些控制点定义了一块双三次样条曲面片。

$$\sum_{i=0}^{3}\sum_{j=0}^{3} a_{ij} u^i v^j$$

在OpenGL中，可将控制点输入二维求值器，根据伯恩斯坦基函数确定这16个系数，并计算对应多项式的值，从而生成曲面模型。

13.6 示例

13.6.1 样条曲线

本节将给出样条曲线生成程序的具体细节，包括定义控制点供求值器使用，根据目标数据类型启动求值器，定义曲线上的全部控制点，遍历控制点构建每4个点为一组的控制点序列，最后调用求值器绘制实际曲线。下面给出图13-2和图13-3中Bézier样条曲线的生成代码，主要列出了建立控制点的关键代码，而忽略了一些细节。完整的示例代码包含在本书附带的资料中。代码用glEvalCoord1f(...)函数返回曲线上的点，而不是用glVertex*(...)函数配合glBegin(...)和glEnd(...)函数对。这种方式与后面将介绍的二维曲面片的自动生成方式不同。

这段代码中的关键点在于如何管理控制点的四元组。前面讨论过，仅使用原始控制点无法得到平滑曲线，因此需要新增控制点对原始控制点进行插值，使曲线段之间的过渡更为连

续和平滑。

```
glEnable(GL_MAP1_VERTEX_3)
#define LAST_STEP (CURVE_SIZE/2)-1
#define NPTS 30
void makeCurve(void)
{   ...
    for (i=0; i<CURVE_SIZE; i++) {
        ctrlpts[i][0]= RAD*cos(INITANGLE + i*STEPANGLE);
        ctrlpts[i][1]= RAD*sin(INITANGLE + i*STEPANGLE);
        ctrlpts[i][2]= -4.0 + i * 0.25;
    }
}
void curve(void) {
    int step, i, j;
    makeCurve(); // 计算整个曲线的控制点
    // 从ctrlpts逐段计算点到segpts，以定义每个曲线段
    // 第一段和最后一段不在中间段
    for (step = 0; step < LAST_STEP; step++) {
        if (step==0) { // 第一种情况
            for (j=0; j<3; j++) {
                segpts[0][j]=ctrlpts[0][j];
                segpts[1][j]=ctrlpts[1][j];
                segpts[2][j]=ctrlpts[2][j];
                segpts[3][j]=(ctrlpts[2][j]+ctrlpts[3][j])/2.0;
            }
        }
        else if (step==LAST_STEP-1) { // 最后一种情况
            for (j=0; j<3; j++) {
                segpts[0][j]=(ctrlpts[CURVE_SIZE-4][j]
                            +ctrlpts[CURVE_SIZE-3][j])/2.0;
                segpts[1][j]=ctrlpts[CURVE_SIZE-3][j];
                segpts[2][j]=ctrlpts[CURVE_SIZE-2][j];
                segpts[3][j]=ctrlpts[CURVE_SIZE-1][j];
            }
        }
        else for (j=0; j<3; j++) { // 一般情况
            segpts[0][j]=(ctrlpts[2*step][j]+ctrlpts[2*step+1][j])/2.0;
            segpts[1][j]=ctrlpts[2*step+1][j];
            segpts[2][j]=ctrlpts[2*step+2][j];
            segpts[3][j]=(ctrlpts[2*step+2][j]
                        +ctrlpts[2*step+3][j])/2.0;
        }
        // 定义求值器
        glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &segpts[0][0]);
        glBegin(GL_LINE_STRIP);
            for (i=0; i<=NPTS; i++)
                glEvalCoord1f((GLfloat)i/(GLfloat)NPTS);
        glEnd ();
        ...
    }
}
```

上面的代码中使用glEvalCoord1f(...)函数从OpenGL的求值器得到曲线上的点。还有一种方法更加简便，就是使用glMapGrid1f(...)函数生成曲线上均匀间隔的点，然后使用glEvalMesh1(...)函数直接生成曲线。以这种方式生成样条曲面的代码见下面的示例。

13.6.2 样条曲面

本书附带的资料中有两个样条曲面代码示例。第一个示例显示了如何绘制简单曲面片（基于4×4的控制点网格的曲面），第二个示例显示了使用更多控制点绘制大的曲面。下面列出第一个示例中的部分代码，生成由曲面片的4×4点数组给定的曲面，其绘制结果如图13-8所示。在代码中，首先初始化4×4的点数组，启动自动法向生成（该功能可用是因为使用了glEvalMesh(...)函数），确定求值器目标，执行求值器操作。其中曲面片的控制点是有意设置为这些简单数据的，以便于观察对应的结果。请注意，曲面片的点通常以参数化方式工作，

面不是索引方式。后面的通用曲面生成代码中将会显示。这段代码中同样使用了

```
glEnable(...)和
glMapGrid2f(GL_MAP2_TEXTURE_COORD_2,...)
```

来自动生成纹理坐标，结果如图13-10所示。但具体的纹理映射代码没有列出。请注意，glMapGrid2f(...)函数的第3个参数指定纹理坐标生成为4.0，其含义是将纹理坐标映射到4个网络点。

```
point3 patch[4][4] =
        {{{-2.,-2.,0.},{-2.,-1.,1.},{-2.,1.,1.},{-2.,2.,0.}},
         {{-1.,-2.,1.},{-1.,-1.,2.},{-1.,1.,2.},{-1.,2.,1.}},
         {{1.,-2.,1.},{1.,-1.,2.},{1.,1.,2.},{1.,2.,1.}},
         {{2.,-2.,0.},{2.,-1.,1.},{2.,1.,1.},{2.,2.,0.}}};

void myinit(void) {  ...
    glEnable(GL_AUTO_NORMAL);
    glEnable(GL_MAP2_TEXTURE_COORD_2);
    glEnable(GL_MAP2_VERTEX_3);
}
void doPatch(void) {
//绘制由4×4的控制点确定义的曲面片
#define NUM 20    //
    glMaterialfv(...);  //需要定义材质
    glMap2f(GL_MAP2_VERTEX_3,0.0,1.0,3,4,0.0,1.0,12,4,&patch[0][0][0]);
    glMap2f(GL_MAP2_TEXTURE_COORD_2,0.0,4.0,3,4,0.0,4.0,12,4,
        &patch[0][0][0]);
    glMapGrid2f(NUM, 0.0, 1.0, NUM, 0.0, 1.0);
    glEvalMesh2(GL_FILL, 0, NUM, 0, NUM);
}
```

与前面一个示例一样，也可以在glMap2f(...)函数后，配合glBegin(GL_QUADS)函数和glEnd()函数，并使用两重循环，每重循环中调用glEvalCoord2f(...)函数生成顶点，来绘制实际的曲面片。但是这种方式将不能自动生成法向和纹理坐标。读者可以权衡这两种操作方式的利弊，选择满足实际要求的方式来绘制图像。

使用二维求值器可以将单个曲面片扩展到完整曲面，其方式与使用一维求值器创建扩展曲线类似。这种方式首先需要定义一组控制点，定义并启动适当的二维求值器，从控制点生成曲面片，并绘制每个曲面片。具体细节在下面的示例代码中列出。

下面这段示例代码包含两部分。第一部分为分步生成二维控制点的函数，这种生成方式
467 与前面曲面片示例中以及第7章示例中手工定义控制点的方式不同。分步生成控制点对于分步生成曲面是一个非常有用的工具。第二部分为从控制点生成曲面片的部分代码，其中列出了如何在原始控制点之间新增中间点。由于新增中间点都为曲面片的边界点，所以新增中间点在曲面片点数组中的索引位置为0或3，内部点全为原始控制点。实际绘制曲面片由函数doPatch（...）完成。其具体代码与前面示例类似，因此在此省略。

```
point3 ctrlpts[GRIDSIZE][GRIDSIZE];

void genPoints(void)
{
#define PI 3.14159
#define R1 6.0
#define R2 3.0
    int i, j;
    GLfloat alpha, beta, step;

    alpha = -PI;
    step = PI/(GLfloat)(GRIDSIZE-1);
    for (i=0; i<GRIDSIZE; i++) {
        beta = -PI;
        for (j=0; j<GRIDSIZE; j++) {
            ctrlpts[i][j][0] = (R1 + R2*cos(beta))*cos(alpha);
            ctrlpts[i][j][1] = (R1 + R2*cos(beta))*sin(alpha);
            ctrlpts[i][j][2] = R2*sin(beta);
            beta += step;
```

```
            }
            alpha += step;
        }
    }
    void surface(point3 ctrlpts[GRIDSIZE][GRIDSIZE])
    {...
        ...{ //  一般情况（曲面片内部）
            for(i=1; i<3; i++)
                for(j=1; j<3; j++)
                    for(k=0; k<3; k++)
                        patch[i][j][k]=ctrlpts[2*xstep+i][2*ystep+j][k];
            for(i=1; i<3; i++)
                for(k=0; k<3; k++) {
                    patch[i][0][k]=(ctrlpts[2*xstep+i][2*ystep][k]
                        +ctrlpts[2*xstep+i][2*ystep+1][k])/2.0;
                    patch[i][3][k]=(ctrlpts[2*xstep+i][2*ystep+2][k]
                        +ctrlpts[2*xstep+i][2*ystep+3][k])/2.0;
                    patch[0][i][k]=(ctrlpts[2*xstep][2*ystep+i][k]
                        +ctrlpts[2*xstep+1][2*ystep+i][k])/2.0;
                    patch[3][i][k]=(ctrlpts[2*xstep+2][2*ystep+i][k]
                        +ctrlpts[2*xstep+3][2*ystep+i][k])/2.0;
                }
            for(k=0; k<3; k++) {
                patch[0][0][k]=(ctrlpts[2*xstep][2*ystep][k]
                                +ctrlpts[2*xstep+1][2*ystep][k]
                                +ctrlpts[2*xstep][2*ystep+1][k]
                                +ctrlpts[2*xstep+1][2*ystep+1][k])/4.0;
                patch[3][0][k]=(ctrlpts[2*xstep+2][2*ystep][k]
                                +ctrlpts[2*xstep+3][2*ystep][k]
                                +ctrlpts[2*xstep+2][2*ystep+1][k]
                                +ctrlpts[2*xstep+3][2*ystep+1][k])/4.0;
                patch[0][3][k]=(ctrlpts[2*xstep][2*ystep+2][k]
                                +ctrlpts[2*xstep+1][2*ystep+2][k]
                                +ctrlpts[2*xstep][2*ystep+3][k]
                                +ctrlpts[2*xstep+1][2*ystep+3][k])/4.0;
                patch[3][3][k]=(ctrlpts[2*xstep+2][2*ystep+2][k]
                                +ctrlpts[2*xstep+3][2*ystep+2][k]
                                +ctrlpts[2*xstep+2][2*ystep+3][k]
                                +ctrlpts[2*xstep+3][2*ystep+3][k])/4.0;
            }
        }
    ...
    }
```

13.7 小结

本章介绍了如何利用各种类型的基函数对几何点进行插值。介绍了功能强大的工具，使用少数控制点就可以创建复杂的曲线和曲面。这种新的几何建模技术扩展了前面章节介绍的简单图元的创建方式，可用于创建复杂的几何模型。此外，这种插值技术同样可用于增加曲面的着色处理和纹理映射等功能。

13.8 本章的OpenGL术语表

本章介绍了OpenGL中的样条功能、相关函数及其与样条相关的参数。由于这些函数通常带有许多参数，用于定义曲线和曲面，本节将不具体列出。相关参数请参考OpenGL手册。前面章节介绍过的函数及其参数也不再列出。

OpenGL函数

glEvalCoord*(...)：计算一维或二维映射，按照单精度或双精度浮点坐标

glEvalCoord*1(...)：计算由glMap*1（）函数定义的一维或二维映射

glEvalMesh*(...)：计算点或直线的一维或二维网格

glMap*1(...)：基于多个参数定义一维或二维求值器

469 glMapGrid*(...)：定义一维或二维网格，按照单精度或双精度浮点类型

OpenGL参数

GL_AUTO_NORMAL：属性开关，使glMap2{}函数在GL_MAP2_VERTEX_*方式下自动生成法向

GL_MAP*_COLOR_4：glMap*{}的参数，指定求值器的控制点分别为颜色的RGBA分量，*表示求值器用于定义曲线还是曲面

GL_MAP*_NORMAL：glMap*{}的参数，指定求值器的控制点分别为法向的分量，*表示求值器用于定义曲线还是曲面

GL_MAP*_TEXTURE_COORD*：glMap*{}的参数，指定求值器的控制点分别为纹理坐标的分量，*表示求值器用于定义曲线还是曲面，第2个*表示纹理的维数

GL_MAP*_VERTEX_*：glMap*{}的参数，指定求值器类型，第1个*可以为1或2，分别表示求值器用于定义曲线还是曲面，第2个*可以为3或4，表示控制点维数

13.9 思考题

1. 本章介绍的每种类型的基函数的取值范围都为[0，1]，基函数的取值都为非负，并且基函数的和为1。用Catmull-Rom样条的基函数验证上述性质。此外，对于每类基函数，除了一个基函数外，其他所有基函数在端点处的取值均为0，而只有该基函数在该端点处取值为1。是否具有上述性质的函数集合都可以作为插值基函数？其插值结果与使用标准基函数的插值结果是否相同？为什么？
2. 给出几对函数，使它们具有前一个练习题中基函数的性质，并用它们对两个点进行插值。这些函数可以不是线性的，例如，$f_1(t) = 1 - t^2$和$f_2(t) = t^2$。请思考是否还有其他类型的函数。除了直线外，插值得到的点还能具有其他形状吗？如何解释这种情况？
3. 给出一组函数对3个或4个点进行插值，要求这组函数不同于本章介绍过的基函数，但具有与这些基函数类似的性质。给出最终插值函数的封闭解析表达式，使用一种方法绘制其曲线，该曲线与本章介绍的标准插值曲线是否相似，有何不同？你能否创建不同于本章介绍过的插值曲线？
4. 根据本章对Bézier样条曲线及其在顶点P_1、P_4，P_2和P_3处性质的介绍，运用简单微积分推导一般参数方程$f(t) = at^3 + bt^2 + ct + d$的导数，并计算该导数$f(t)$在$t = 0$ (P_0)和$t = 1$ (P_3)处的值。

a. 曲线在P_0处的斜率与从P_0到P_1的直线段斜率一样

b. 曲线在P_3处的斜率与从P_2到P_3的直线段斜率一样

c. $f(t)$在P_0处的导数为从P_0到P_1的向量的三分之一

d. $f(t)$在P_3处的导数为从P_2到P_3的向量的三分之一

470 根据上述已知条件推导一段Bézier样条的方程组。

13.10 练习题

1. 选择一个物体，其形状是你感兴趣的，试着设计一组点，将其作为控制点时，对应的样条曲线或曲面与该形状相同或类似。使用本章的示例中介绍的求值器方法用控制点绘制该曲线或曲面，验证结果是否正确。
2. 给定6个或更多控制点用于创建三次曲线，保证控制点数目为偶数。手工新增控制点，将该三次曲线扩展到所有控制点，并保证其分段平滑。使用大于4×4的控制点二维数组，对曲面的控制点完成上述相同的练习。
3. 使用Bézier三次样条矩阵形式的推导过程，构建Catmull-Rom三次样条的矩阵形式。

13.11 实验题

1. 在本章示例代码中构造样条曲线的控制点，然后调用OpenGL的求值器函数创建实际的曲线或曲面片。本章前面也讨论过另外一种方式，即对给定的1个或2个参数，手工计算曲线或曲面上的点。编写程序实现这种计算方式，并用其生成插值曲线或曲面上的点。
2. 使用前一个实验中的控制点，为OpenGL求值器建立控制点数组。使用求值器生成这些控制点确定的曲线或曲面。比较手工编写程序和使用求值器来生成这些曲线或曲面所需的编程工作量。

13.12 大型作业

1. （小房子）在前面的小房子建模中，屋顶可能使用非常简单的形状。根据本章介绍的样条曲面技术，对屋顶设计进行更新，构造出外观更精致的房子。将原来的屋顶函数注释掉，并将新设计的屋顶添加到房子场景中。
2. （场景图分析器）如何在场景图的几何节点中表示求值器的几何数据？分析场景图时如何为求值器生成对应的代码？如果使用校对机方式，而不是求值器方式，如何处理上述两种情况？ 471

第14章　非多边形图形技术

在前面的几章我们主要介绍面向多边形的图形API，并学习了如何使用这种工具来生成需要的图像。然而，计算机图形学并不是只有这一种描述和生成图像的方式。

另一种计算机图形技术是通过处理几何模型来独立地决定场景中各个像素的颜色，我们称为逐像素图形技术，它包括几种不同的绘制方法。光线投射是其中较为简单的一种，通过由视点出发经过虚拟屏幕中任意像素的光线，计算其与场景中的几何模型的第一个交点来决定对应像素的颜色。而光线跟踪则是一种更为复杂、精准的方法。类似于光线投射，一开始光线也是从视点出发，经过屏幕中的每一个像素，当然也可以为每个像素生成多根光线。当光线与场景中的几何物体相交时，它可能会被反射或折射，从而产生间接光线。这些光线将综合决定屏幕中对应像素的颜色。另外，还有一些利用像素来记录计算值的方法。这在迭代函数或分形研究中可以看到。由于这种图形应用的领域十分广泛，因此我们相信它在初级图形学课程中会很有价值。

这一章介绍几种生成计算机图像的逐像素操作实例。当然，在这些操作上我们不会涉及太深，但会对它们做一些介绍，以便读者能了解它们的用途并决定是否需要阅读其他资料作深入研究。为了更好地学习本章，读者应该对第4章中介绍的用代数表示的几何操作有所了解。

14.1　定义

在计算机图形学中，光线跟踪和光线投射这两个词语的定义并不是很明确。我们在这里
472 将光线投射定义为通过由视点出发经过屏幕上像素的一根光线与场景中的几何模型的交点来计算该像素颜色的图像生成过程。此过程不考虑阴影、反射或者折射等属性的计算。而将光线跟踪定义为光线投射的扩展，去掉了以上的限制。如果需要，可以为一个像素投射多根光线，可以使用任何一种阴影、折射或反射计算方法。接下来将对这两种处理过程进行简要介绍，读者可以参考Glassner的教程[GL]或者开源的光线跟踪程序POVRay来获得更多的经验。

14.2　光线投射

在第1章我们为透视变换定义的标准视图变换设置和视域体中，视域体的前截面可看作实际显示屏幕的等同体。我们可以通过一个窗口到视口的逆操作将实际的屏幕映射到这个空间，实质上是通过创建一个屏幕到视平面的操作，从而在视域体的前截面创建了一个虚拟的屏幕。

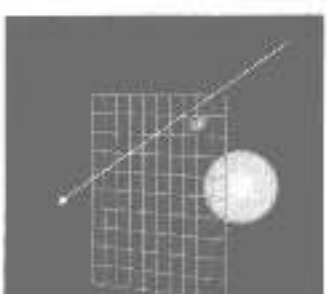


图14-1　三维空间中视点和虚拟屏幕的相对位置关系

为此，我们采用视图和投影变换中的透视视域体的想法，像第4章所描述的那样来设置视图变换，将视点设置在原点，视域体的前截面位于$z = -1$的平面上。正如OpenGL中用视场角α和高宽比ρ来定义透视投影一样，我们能确定视域体前截面右上角的坐标$x = \tan(\alpha)$，$y = \rho^* x$，这个以Z轴为中心，宽高分别为$2x$和$2y$的平面是我们将在其上“绘制”的虚拟屏幕。将这个虚拟的屏幕划分为步长为$2^* x/N$的像素点，其中N为屏幕水平方向的像素个数。图14-1显示了一个世界坐标系中的虚拟屏幕。

以及视点和一根采样光线。为了满足设置的高宽比，应该在屏幕的垂直方向也使用相同的步长来划分屏幕。在OpenGL透视投影中最后的两个参数是前后裁剪平面。可以使用这些参数来限制需要作光线与几何模型相交计算的范围，即位于前后裁剪平面之间的几何模型才需作相交计算。而位于前后裁剪平面之外的几何模型则无需作相交计算。 473

光线投射的关键是生成一根从视点出发经过虚拟屏幕上任意像素点的光线。这条光线可用参数值为正的参数线段来表达。由于每个像素实际上对应的是虚拟网格上由相邻水平线和垂直线构成的一个小方块，所以可以选择使用像素中的一点来构造光线。通常来说，选择哪一点都无所谓，我们可以选择四边形的中心。这样光线将会从原点出发经过点(x,y,1)，对应的参数方程为(xt,yt,t)。

一旦知道了光线的参数方程，就能够确定光线与模型是否相交。如果没有交点，那么简单地用背景颜色来填充像素。如果有一个以上的交点的话，选择距视点最近的交点，用它的颜色来填充像素。光线与物体的求交计算非常耗时，是光线投射中的关键问题。它其实是我们在第4章讨论过的碰撞检测的特殊情况。最简单的求交计算是计算与球是否相交的情况，可以使用点到直线的距离公式来衡量光线离球心的远近，如果距离小于球的半径则说明存在相交，那么用一个简单的二次方程就可以求得交点。

像素的颜色大部分取决于我们正在观察的模型，如果没有使用光照模型，那么每个物体都有自己的颜色，像素的颜色将由经过对应虚拟像素的光线和物体的交点决定。如果使用了光照模型，而系统又不提供局部光照模型的话，那么我们需要自己实现它。这意味着在光线与物体相交的点上，必须自己实现在第6章中提到的环境光、漫反射光和镜面反射光的计算。这涉及计算局部光照的四个向量：法向量、观察向量、反射向量和光源向量。观察向量很容易确定，它就是我们使用的光线；法向量可以通过解析计算或者是几何计算获得，就像曾经为OpenGL的glNormal*(...)函数求取法向量那样计算；反射向量则如我们在数学建模中讨论的那样计算；而光源向量则直接由交点到场景中的光源生成。有关光照计算已经在第6章进行了详细讨论，这里不再作深入介绍。

由于能按自己的方式生成光照效果，这样我们就能使用比OpenGL提供的还要复杂的着色处理模型。这些着色处理模型通常采用标准的计算环境光和漫反射光的算法，但可以通过改变表面的光线反射方式来改变镜面高光在物体上的表现方式。在第6章我们介绍过这些称为各向异性的着色处理技术，通过双向反射分布函数（BRDF）来生成不同的反射向量。这种函数采用光源向量的x,y,z分量来决定光线将被反射的方向，并用这个反射向量来计算标准的镜面反射。详细的计算过程比这里介绍的更为复杂，读者可以通过查阅一些高级的参考资料获得更多的信息。

在颜色预计算场合，光线投射方法也可以结合期望的技术用来观察模型。当光线和模型相交时，该交点的颜色将被赋给图像中对应的像素。这种技术或者是一种更为复杂的光线跟踪技术都能用来计算全局光照模型，如辐射度模型或者光子映射模型。光照处理用来计算模型中每个表面的光强，也可得到其他的视觉特性如纹理映射等。对于由虚拟屏幕中的像素定义的光线，与模型的交点的颜色可以从模型中读取，或者通过模型中的相关信息计算得到，然后赋给对应的像素，即使是应用简单的光线投射方法生成的图像，也会具有模型传神的细节。 474

由于光线投射是每个像素只有一根采样光线，屏幕坐标与真实世界中实际值的精细程度相比又很粗糙，因此，光线投射存在走样问题。在由OpenGL生成的图像中可以看到这种走样现象，我们也注意到OpenGL有一些反走样处理能力，这些走样问题是光线投射固有的，不能在图像的生成过程中解决。但可以通过后期的图像处理来进行平滑。这可能会丢失细节，所以必须考虑是否值得进行平滑。最基本的平滑原理是在帧缓存中生成图像，然后把帧缓存中

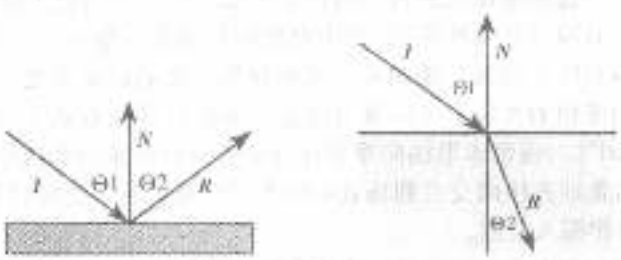
的图像保存在一个颜色数组中，接着采用滤波技术对这个数组进行平滑处理，消除走样。当然也可以采用这章中稍后会介绍的一个像素多根光线的超采样技术，可不考虑反射和折射，用所有采样点的颜色平均值作为像素的颜色。

14.3 光线跟踪

正如所定义的那样，光线跟踪与光线投射之间的不同是光线跟踪包括了以下一种或多种技术，如反射、折射、阴影和一个像素多根光线的超采样。光线跟踪是一种目前讨论较为广泛的通用技术，经典的参考文献有Andrew Glassner的教程[GL]。

我们先介绍光线跟踪是如何处理阴影的。当在包含一个光源的场景里计算光照模型时，从交点到光源能生成光源向量。然而，只有光源确实照到模型表面时这根光线才能参与漫反射光和镜面反射光计算，否则该交点位于光源的阴影处。这样，只需要从交点向光源投射一根光线，看它是否和场景中的其他物体相交。如果相交，则说明该交点不能被光源照射到，因此光源对它也不会产生漫反射或者镜面反射作用，该交点位于光源的阴影处。如果没有和任何物体相交，则说明光源对该交点有漫反射和镜面反射作用。这要求从场景中的任意点向任意方向发出射线，进行额外的光线－物体求交计算，因此光线跟踪很费时。

光线跟踪通常应用于光亮或者透明表面的图像生成，因为它能很好地处理物体表面的反射光和穿透物体表面的折射光。图14-2显示了光线在遇到物体时发生的反射和折射现象。



475 图14-2 光线与物体相交时产生的反射光线（左）和折射光线（右）

反射相对容易处理些，因为在第4章介绍过如何计算反射向量，即，如果入射向量为P，法向量为N，那么反射向量为$R = P-2(N \cdot P)N$。当光线穿过物体表面继续在另一边传播时，称这种过程为折射。光线会继续在同一平面上传播，但它的方向则由于其在媒介内外的传播速度不同而发生了改变。材料的折射系数η用来衡量这种传播速度的不同。Snell定律描述了折射光方向的计算公式：如果Θ_1和Θ_2分别表示入射光和出射光与法向量的夹角，η_1和η_2分别为媒介外部和内部的折射系数，那么从公式$\sin(\Theta_1)/\sin(\Theta_2) = \eta_1/\eta_2$，可以计算出出射光向量。

一旦有了反射向量和折射向量，那么在光线与物体的交点处会产生新的光线，一根是反射光线，另一根是折射光线，它们都位于由入射光线和表面法向量决定的平面内。这两根光线作为初始光线以同样的方式迭代下去，最终每根光线将返回一个颜色值。这两个值会按照光线被反射、折射或者就是简单地被漫反射回来的比例与物体本身的颜色进行组合。

图14-3显示光线发生反射和折射递归生成的光线树，其中R为反射光，T为折射光，L为入射光。当然，我们不能无限制地产生反射光或者折射光，必须定义一种停止机制，这可以通过跟踪光线的衰减来实现，每次只有部分光线被反射或折射，当只剩下很小一部分的光时可以停止迭代。另一种更简单的方法是定义一个迭代次数作为阈值，一旦到达该阈值，就停止迭代。从任意点出发的光线与物体的求交计算导致的光线迭代使大家都知道光线跟踪是一类速度很慢的算法。在计算反射时，可以使用简单的反射方式，也可使用BRDF方式来处理图像

中的各向异性反射。

476

由于在光线投射方法中一个像素只发出一根光线，无法避免走样问题，因此只能通过后期处理，对每个像素应用过滤函数来减少走样现象。然而这种方法会使最终的图像分辨率降低，这是我们想避免的。在光线跟踪方法中，可以通过为每个像素生成多根光线来决定像素的颜色，这样能更好地表现像素对场景的贡献。可以通过把像素分成规则的子像素，为每一子像素生成一根光线来系统地生成所有的光线，或者通过随机选择像素中的一些点，让光线穿过它们，生成这些光线。当然在这两种方法之间存在着一些不同，读者可以通过阅读参考文献[GL]和[SHI]来获得更为详细的信息。在获得这些光线的颜色值之后，需要根据这些颜色值来重新计算像素的颜色，可以简单地取平均，或者是根据光线距像素中心的远近来设置颜色权重，通常距离中心较近的光线具有较高的颜色权重，而距离中心较远的则权重较小。

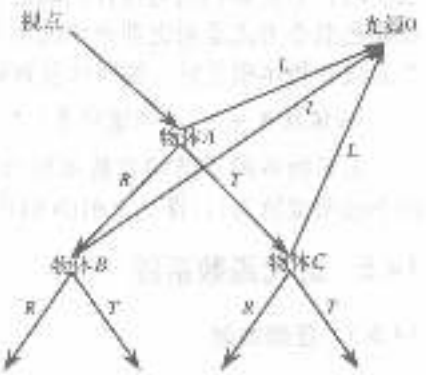


图14-3 光线树



图14-4 由POVRay光线跟踪程序绘制的图像，参见彩图

简单光线跟踪程序的编写并不十分困难，通常在一些计算机图形学课程中，学生的设计项目之一就是编写光线跟踪程序。另外，现在也有许多光线跟踪程序可以无偿获得，其中最著名的就是POVRay程序(http://www.povray.org)，它适用于大多数的计算机系统。事实上，POVRay也可作为一种图形API，因为它有自己的模型表达方式以及绘制控制方法，可以用它的语言来描述场景，生成非常漂亮的图像。另一个可以获得的光线跟踪程序是斯坦福大学的Rayshade(http://graphics.stanford.edu/~cek/rayshade/)，它适用于多种平台。这两个光线跟踪程序的源码是公开的，读者可以深入了解这两个系统的实现细节。图14-4是POVRay光线跟踪程序绘制系统提供的测试基准场景的结果图像，注意看场景中有许多反射和阴影效果。

477

14.4 体绘制

光线投射思想的一个重要应用是绘制体数据。在第9章曾介绍过，三个变量的函数可看作是一个体，即该函数将三维空间中的点与值相关联。这个函数可以解析表达，即由公式或过程来定义；或者是由数据推导得到，还可根据实验过程得到。函数的值可能表示实数、颜色、密度或者它们的任意组合。

在第9章中，我们曾介绍过一些简单的技术，用于分析定义在体上的实值函数，以显示体数据的等值面。另一种更好的方法是向体中投射一组光线，然后求出它们与等值面的交点。已知交点后，就可以决定交点所处的体素，并通过考察等值面如何与体素边界相交来求出法向量，接下来就可以运用通常的光照模型进行绘制，得到的图像能够更好地表现体中的等值面。

然而，光线投射技术除了能够应用于简单的等值面研究之外，还能应用于更多的研究领域。我们能够用它来创建非常复杂且具有丰富环境特征的体的模型，将它和任何适用于模型的技术相结合来生成体的图像，而不仅用它来处理简单的体实数函数和生成等值面图像。如

478 图14-5所示，来自哈勃太空望远镜的数据用来创建猎户星云的模型。模型根据星云的太空特性定义了体中每一个点的颜色和透明度。当每条光线投射进入太空时，它会累积所遇物体的颜色，沿着路径对颜色进行混合，该颜色混合方式是对之前介绍的用于简单颜色混合操作的推广。当光线和物体相交时，按照从前到后的顺序来计算光线的颜色：

物体颜色 + (1 − 不透明度) * (来自余下光线的颜色)

而新物体的不透明度被加到当前不透明度值上，直到累积的不透明度值为1，详见文献[NAD]。



图14-5 猎户星云模型的体可视化（参见彩图）

14.5 迭代函数系统

14.5.1 压缩映射

迭代函数系统（IFS）的概念包括多种操作，我们只讨论其中的两种。一种是压缩映射操作。压缩映射是一种将封闭有限的区域映射到一个更小的封闭有限区域的函数。任何压缩映射函数都有一个特性，如果应用足够多次的话，初始区域就会被映射到一个任意小的区域。假设从点$q = q_0$开始，采用一组压缩映射函数$\{f_i\}$，并定义$q_i = f_i(q_{i-1})$。每一个映射函数f_i被应用的概率为p_i，那么将存在一组称为IFS吸引子的明确定义的点，对于任意点q及足够大的i，点q_i会任意地接近吸引子中的一点。迭代函数系统图像可以通过递归绘制压缩映射定义的整个区域来生成，但迭代函数系统图像通常通过生成大量随机点，并对每个点重复应用压缩映射函数来生成。在多次迭代后，点被显示，吸引子的形状被呈现。

Sierpinski垫圈是一种由压缩映射生成的有趣物体。虽然它能通过多种方式生成，但利用压缩映射来生成的方法是用四个函数将一个四面体等概率地映射成四个四面体，每个四面体的高是初始四面体高的一半并占据初始四面体的一个角。采用向量表达方式，线性压缩映射函数可表示为$f_i(p) = (p + p_i)/2$，$\{p_i\}$为四面体的四个顶点。无限制地重复这个过程，直到四面体的体积接近于零，并且了四面体沿边占据空间位置。四个压缩映射函数中的每个都会将四面体中的每个点移动至距对应顶点一半距离的位置。这很容易计算。如果允许随机地选择映射函数就有以下的处理过程：随机选择一个顶点，将点移动至距顶点一半距离远的位置。多次重复这个过程，点的有限位置将会落在Sierpinski垫圈上。这个过程的更多细节可以参看[PIE]的2D情况，3D则是一种直接的扩展。在图14-6中可以看到用红色显示的四面体的四个顶点和一些青色点，这些青色点是由50 000个任意点应用这个过程计算得到的。



图14-6 Sierpinski 垫圈

另一种我们感兴趣的压缩映射IFS是用来模拟蕨类植物叶片的二维映射。由于这是一种涉及呈现IFS吸引点的二维映射，所以我们可以用本章的逐像素操作来绘制这一效果。以下的线性函数及其概率定义了蕨类植物叶片的映射：

$$f_0(x, y) = (0, 0.16y) \qquad p = 0.01$$

$$f_1(x, y) = (0.85x + 0.04y, -0.04x + 0.85y + 1.6) \qquad p = 0.85$$

$$f_2(x, y) = (0.20x - 26y, 0.23x + 0.22y + 1.6) \qquad p = 0.07$$

479 $$f_3(x, y) = (-0.15x + 0.28y, 0.26x + 0.24y + 0.44) \qquad p = 0.07$$

压缩映射将正方形区域分成四个子区域，如图14-7中用不同颜色标注的区域，经过多次

迭代后结果图像如14-8所示。

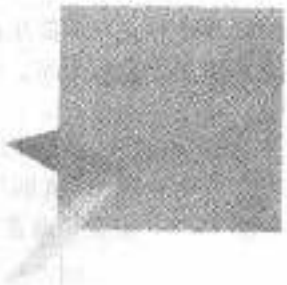


图14-7 蕨类叶片的四个压缩区域（分别用黑、青、橙和灰色表示）



图14-8 用IFS生成的蕨类植物的叶片

14.5.2 生成函数

另一种迭代函数系统（IFS）是生成函数。通过将递归的生成函数应用于每一个几何图元来递归地定义几何结构。第9章的3D牛奶冻函数的2D版本可以通过这种方式定义，即从一条 480
线段开始，然后用图14-9中的一对线段来替代每一条线段，而线段对的中心偏移原线段长度的四分之一。产生的极限函数的“处处连续”特性是因为最终的函数是一致连续函数收敛序列的极限。“处处相似”的特性是因为无论选择的一对值是多么接近，多次迭代之后它们之间总会有一根线段，在下次迭代时将会有一个尖角出现在中间。



图14-9 2D牛奶冻函数的生成过程

本章的逐像素操作并没有介绍这种函数，但它可通过整本书中介绍的多项式地模技术来实现。在这里介绍它是因为它和压缩映射操作的相似性。

接下来我们来看另一种称为“龙曲线”的示例，它用两条线段替代原有的线段来获得，就像牛奶冻曲线一样。但又不同于牛奶冻曲线，在用两条线段替代原有线段时，“龙曲线”是通过交替将两条线段放置到直线的两边来实现的，这种生成操作如图14-10所示，图中的实线段替代了虚线段部分。



图14-10 龙曲线的生成过程 481

结果曲线可以连续迭代所期望的次数。图14-11左图是第十次迭代停止时的龙曲线。龙曲线一个迷人的特性是这种曲线围绕一个给定点填充空间的能力。如图14-11右图所示，四条不同颜色的曲线相交在一点。

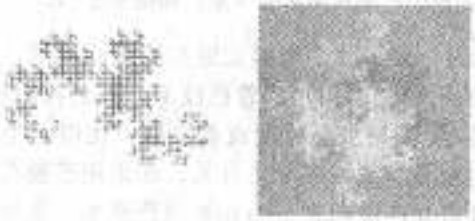


图14-11 一条简单的龙曲线（左）和围绕一点的四条龙曲线（右）

14.6 芒德布罗集和茹利亚集

分形包括多种处理过程，我们主要介绍两种与众不同的处理过程。它们能生成具有动态系统特征的图像。动态系统中许多有趣的问题能够通过计算机图形学进行探索。这方面工作的参考资料包括[PIE]和很多论述分形的科普读物。

假设一系列二次复变函数$\{f_c(z)\}$定义为$f_c(z) = z^2 + c$，$f_{c-1}(z) = f_c(z)^2 + c$，我们感兴趣的是这个序列随着$n$值的增加收敛还是发散。有一种直接的检验方法：随着$k$值的增加，如果$|f_c(z)|$有界，则函数集收敛，否则发散。注意，复数由一对实数(a, b)定义，通常记为$z = a + bi$，这里$i^2 = -1$。所以有以下关系式：

$$(a + bi)^2 = (a^2 - b^2) + 2abi$$
$$(a + bi) + (c + di) = (a+c) + (b+d)i$$
$$|a + bi| = \text{sqrt}(a^2+b^2)$$

如果将不同的复数c应用于函数序列$\{f_c(z)\}$，且总是从初始值$z = 0$开始，就能研究参数空间$\{c\}$的行为。芒德布罗（Mandelbrot）集是一个能让具有这个初始值的函数序列收敛的复数c
482 的集合。如果对于任何k值，$|f_c(z)| > 2$，那么序列将会发散。所以，只需简单地检查对于k的取值直到一个相当大的数如500时，$|f_c(z)|$是否小于2。如果在终止序列之前，发现某个k值使$|f_c(z)| > 2$，那么对于该复数c，就返回这样的第一个k值；如果没有，那么就返回0。芒德布罗集就是由返回0的复数c组成的集合。

为了用图显示这种情况，用2D点(a, b)来表示一个复数$(a + bi)$，根据之前为该复数记录的值，用一个整数颜色来涂染这个点。也可以采用其他的方法进行标识：每个点定义一个复数，然后运用上面提到的方法来处理这个复数，从而决定点的颜色。这样可以得到一个2D区域，并在该区域上创建一个与我们在示例代码中描述的窗口尺寸匹配的网格，运用上面提到的方法处理每一个网格点，然后像之前那样涂染对应的像素。图14-12显示了一幅芒德布罗集的图像，而右边的细节图像则显示了在一个更小区域内收敛的过程。图中的整个芒德布罗集对应一个a的取值在$[-1.5,0.5]$，b在$[-1,1]$的复数$(a + bi)$集，而细节区域的取值为$a \in [0.30, 0.32]$，$b \in [0.48, 0.50]$。

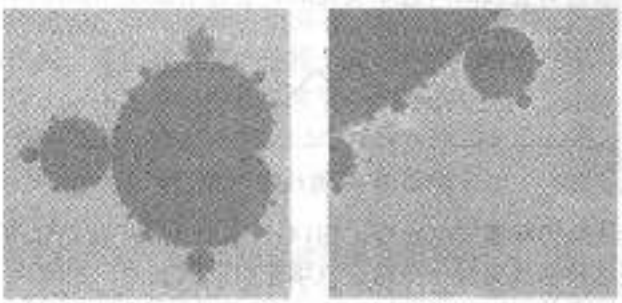


图14-12 完整的芒德布罗集图（左）和细节图（右），参见彩图

对于这一系列函数$\{f_c(z)\}$，如果改变取固定值和变量的规则，即选择一个固定值c，对于不同的z值计算序列函数，使用和之前相同的着色技术，那么我们可能都会问一个非常相似的问题，即这个序列是否随着n值的增加而收敛或者发散。使得这个序列收敛的复数z的集合称为茹利亚（Julia）集。茹利亚集与芒德布罗集有关，如果用芒德布罗集里的任何一个复数c来
483 生成茹利亚集时，茹利亚集将会连接起来，而如果用芒德布罗集外的一个复数c来生成茹利亚

集时，茹利亚集将是完全断开的。用芒德布罗集中非常接近边缘的复数能够生成非常有趣和与众不同的茹利亚集。图14-13显示由固定点（-0.74543, 11301）计算得到的特殊茹利亚集。当然也可以像显示芒德布罗集那样选择其中的一小部分进行显示，给出更为详细和迷人的图像。茹利亚集和芒德布罗集之间的关系将会在本章末的几个实验中进行研究。

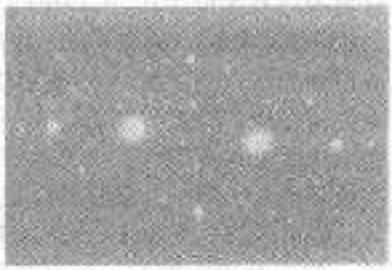


图14-13 针对某一固定c值的茹利亚集，参见彩插

14.7 OpenGL支持的逐像素操作

OpenGL提供两种方法显示逐像素操作的图像。第一种是用GL_POINTS绘制模式，颜色由点决定。这涉及如下的一个循环操作：

```
glBegin(GL_POINTS)
    for row = 0 to N-1
        for column = 0 to M-1
            calculate point (x,y) for pixel (M,N)
            calculate color for point (x,y); return color
            glColor(color)
            glVertex2^(x,y)
glEnd()
```

这种操作需要定义一个宽为M个像素，高为N个像素的窗口，并设置一个与该窗口尺寸匹配的2D正交投影，这很容易实现，在main()中可以用

```
glutInitWindowSize(M,N)
```

在init()中可以用

```
gluOrtho2D(0. , (float)M, 0. , (float)N)
```

484

这种操作不支持窗口尺寸的改变，因为它直接与初始窗口尺寸联系在一起，除非动态地计算屏幕空间的大小，然后重新定义显示空间的尺寸。图14-12就是通过这种操作生成的，下面是芒德布罗集内部计算的详细代码。

```
xstep = (XMAX - XMIN)/(float)(WS-1); //WS－若干像素表示该窗口尺寸
ystep = (YMAX - YMIN)/(float)(WS-1);
glBegin(GL_POINTS);
for (i = 0; i < WS; i++) {
    x = XMIN + (float)i * xstep;
    for (j = 0; j < WS; j++) {
        y = YMIN + (float)j * ystep;
        test = testConvergence(x,y);
        // ITERMAX－迭代步最大次数
        // colorRamp函数已在第6章中介绍过颜色渐变
        colorRamp((float)test/(float)ITERMAX);
        glColor3fv(myColor);
        glVertex2f((float)i,(float)j);
    }
}
glEnd();
```

有两种其他的方法可以用来显示这种计算结果。一种方法是创建有足够多点的网格区域达到期望的细节要求，然后像在逐像素操作过程中那样，对区域内的每个点设置一个颜色值，这样就可以创建一组多边形，顶点由网格确定。可以像处理函数曲面那样将函数表示为一组多边形，如果使用平滑着色处理算法，就可以得到一幅比逐像素操作还平滑的图像，尽管必须注意不要因平滑着色处理而掩盖了实际的断裂。由于在收敛与发散区域的边界上会出现不可信细节，因此这是一种相当差的表现芒德布罗集和茹利亚集的方法。

另一种显示结果的方法是采用定义颜色的技术来给区域内的每个点赋予一个高度值，用

这些高度值创建一个3D曲面。当然，这并不是一种逐像素操作，但这种方法能够生成有趣的分形表达和类似于图14-14中的东西。这里利用复杂动态系统问题的结论，当k为零时，将高度设为1；当k不为零时，高度则为1－1/k。图14-14是用这种方法重画图14-12的结果，图中把芒德布罗集显示为一个高原，它的边界随着动态系统的发散而下落，边界的阶梯特征是因为发散迭代的次数是整数，因此，曲面的高度是不连续的。

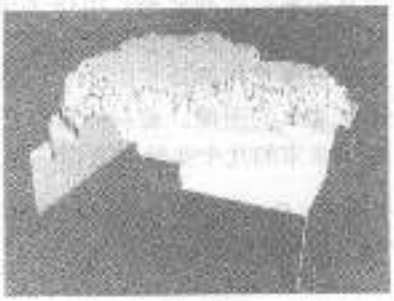


图14-14 用简单曲面表示芒德布罗集及其邻近区域

极限处理过程如Sierpinski吸引子包含一些非通用操作，每一种操作都与一种极限处理过程一一对应。对于Sierpinski吸引子，处理过程决定了每一个点的位置，我们
485 只需要在它们出现的地方画出它们就行了，而每一步的更新则可以利用在第7章介绍过的idle()回调函数来实现。这一回调函数需要包括一些能改变点位置的简单操作，然后再调用重显示操作把它显示到屏幕上。下面给出了这种操作的一段代码，其中vertices是四面体的顶点数组。

```
float points[3][N], vertices[3][4];
// 在display函数中有如下语句
beginPoints();
    for i = 0 to N
        setPoint(points[0][i], points[1][i], points[2][i]);
endPoints();
// 在idle()函数中有如下语句
for i = 0 to N {
    j = (int)random()%4;
    for k = 0 to 2 {
        points[0][k] = (points[0][k] + vertices[0][j])/2.0;
        points[1][k] = (points[1][k] + vertices[1][j])/2.0;
        points[2][k] = (points[2][k] + vertices[2][j])/2.0;
    }
}
发送一个重显示事件
```

14.8 小结

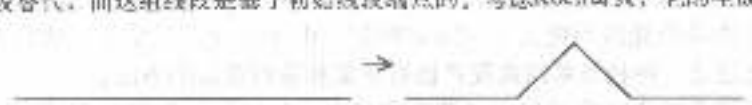
本章主要介绍了光线投射、光线跟踪、基于光线投射的体绘制、分形和迭代函数系统。它们都能通过逐像素操作来实现。这种操作每次计算一个像素来生成图像，而不是计算基于多边形的几何体。这可
486 以从另一角度来研究能够生成有趣的、有用的图象的图形学。但这种逐像素操作需要不同的建模和观察方法。如果使用这些方法，我们需要更深入地学习，但这是值得的。

14.9 思考题

1. 从http://www.povray.org上下载POVRay系统，并阅读它的使用指南。该系统的几何图元是什么？它所使用的图元与光线简单相交的几何物体（如球或平面）之间有关系吗？

14.10 练习题

1. 考虑其他一些生成函数，看看使用它们能生成什么样的图像。试创建如下的生成函数：对任何一条线段都用一组线段替代，而这组线段是基于初始线段端点的。考虑Koch曲线，它的生成函数如下图所示



将这个生成函数应用于等边三角形，结果图像是一种称为Koch雪花的图形。

14.11 实验题

1. 阅读POVRay系统提供的一些示例模型，了解它们是如何组织的。用这些模型来运行系统，看结果图像是怎样的。然后通过改变图元、纹理图、光照或其他属性来修改模型，再看看这些改变是如何影响图像的。
2. 通过创建两个显示窗口来考察芒德布罗集和茹利亚集之间的关系。其中一个窗口显示芒德布罗集并包含一个鼠标回调函数，该回调函数根据鼠标点击位置来返回复数c。另一个窗口则根据该复数来创建茹利亚集并显示它。让这个程序完全实现交互，以便在任何有鼠标点击芒德布罗集的时候能够生成一个新的茹利亚集。

14.12 大型作业

1. 使用实验题2的思想，沿着穿过芒德布罗集的一条线段移动点c，生成不同的茹利亚集，然后创建这些集的动画。对每一个c值，绘制茹利亚集，把对应的颜色缓存存到数组里，再写到文件中，接着就像在第4章描述的那样，把这些文件组成一部电影。一旦完成以上操作，可试着在包含芒德布罗曲线的空间里描述一条曲线并沿着这条曲线移动点c。如果沿着芒德布罗集的边界移动点c，这会是一个有助于理解茹利亚集本质的研究问题。 487
2. 任意选取一种芒德布罗集图或茹利亚集图，编写一个交互程序，并用它画出本章所描述的初始图像，并允许用户在所选择的图中选择两个点，作为矩形的两个对角。然后在与用户所选择的矩形具有相同高宽比的新区域内重画选中的部分。让用户一直操作下去，对图像进行深入研究。这里的关键是将在屏幕空间选择的点转换到图像空间的点。 488

第15章 硬 拷 贝

我们经过艰苦努力分析清楚了问题，开发出相当好的模型和漂亮的图像以及动画，对问题给出了解决方案。但这些图像和动画无法与广大观众分享，只能运行于某台特定计算机并展现在屏幕上。现在需要将这些工作展示给其他人，在将这些图像转移到其他媒介时不损失图像的质量。又或者这些工作不必与他人分享，但想保留一份记录存档以备将来使用。

本章讨论创建工作记录时可能面临的问题，对计算机图形学中不同类型的工作需要采用不同的技术，包括数字图像和印刷的图像、电影胶片、视频，以及生成计算机3D模型图像的对象原型技术。学完本章内容后，读者将掌握几种创建硬拷贝的技术，能够为自己的工作以及具体应用选择合适的硬拷贝技术。为了更好地理解本章内容，读者应当理解颜色和视觉交流的本质，知道如何使图像变得生动的因素，并了解人们应用计算机图形学结果的常用方式。

15.1 定义

计算机图形学中的硬拷贝指的是将图形计算结果输出到固定的媒介上的技术，使图形结果与产生它的计算环境在物理上脱离关系，这样可以无需原始的计算环境也能将图形结果呈现到观众面前。硬拷贝的方法有很多种，基本原理是只要能携带图像的任何类型媒介都可以作为硬拷贝的候选者，硬拷贝可使用物理媒介（纸张、雕塑），也可以使用数字媒体（图像、视频）。每种媒介都有其固有的性能和复制图像的要求。本章将介绍几种最通用的硬拷贝媒介和高效率使用它们的方法。

制作硬拷贝就是生成一份可以发送到某种输出设备上的数字记录。这些设备可能隶属于计算机，如打印机或者胶片记录器，或者是通过网络交换数据的设备，也可能是硬盘或CD-ROM。某些硬拷贝媒介可以直接用数码输出，但有些包含额外的制作过程。所以，对图形硬拷贝的讨论包括如何组织数据的描述，以便于使用这些外部制作过程。

15.2 选择输出媒介

在计算机屏幕上生成有效的图像是一回事，而在其他媒介上有效生成呈现给观众的图像可能是完全不同的另一回事。作为发展图形视觉交流的一部分，创建图形硬拷贝时必须了解选择什么样的媒介将图像传达给观众，以及怎样应用这些媒介。打印、在线内容、视频[⊖]、数字视频或者用快速成型工具生成的物理对象等都有各不相同的特性，影响着硬拷贝呈现的效果。为了更好地了解各类媒介的特性，应尝试采用不同媒介，研究不同媒介的特性和观众的使用方式，才能学会为不同工作选用不同硬拷贝媒介。

15.2.1 数字图像

如果只是生成静态图像，或者从交互式程序截取单一的图像，则可以将它们用一些通用的图形文件格式写入文件。前面章节中提到将颜色缓存中的内容保存到内部颜色阵列，然后把这个阵列转换成标准图像文件格式，如GIF、TIFF、JPEG、PNG或其他格式。用简单的表

⊖ 本章用“视频”一词专指“模拟视频”技术。——译者注

示RGB格式的无符号字符存入每个字节的方式将阵列写入文件，可以得到卓越RGB文件。当不想开发自己的工具将图像转换为标准格式，或者找不到合适的文件格式转换库时，可以使用原始RGB文件，Photoshop等工具可以打开它并保存为任意标准格式的图像文件，Photoshop在这里成了文件格式转换工具。本书中的许多图例就是使用这种方式编辑的，作者力荐这种转换方法。

如果读者不熟悉这些文件格式，请看这里准备的简要介绍。GIF是Graphics Image Format的缩写，它采用8位索引颜色无损压缩保存图像。但GIF文件格式使用了Lempel-Ziv-Welch (LZW)文件压缩，这是一项受专利保护的技术。如果使用它来创建商业软件时必须先取得许可。TIFF是标记图像文件格式（Tagged Image File Format），是一种非常通用的格式。它可以存储任意颜色深度的图像。TIFF不带文件压缩，因此，TIFF文件通常非常大，但它没有损失图像质量，所以成为良好的存档格式（特别适用于廉价的硬盘和CD-R/DVD-R）。JPEG文件格式通常采用基于离散余弦变换的有损压缩，但在选择使用高质量压缩的情况下也可以达到无损压缩的效果。JPEG对自然图像表现非常好，但在处理包含许多线条和尖锐边缘的图像时较差。因为离散余弦变换以64×64的像素块为操作单位，块的边缘信息可能被打散了。特别 490
是JPEG对包含文字的图像处理效果非常差。它的名字来自它的开发者：Joint Photographic Experts Group。PNG是便携式网络图形的缩写（Portable Network Graphics），这个格式的出现是为了代替GIF成为广泛使用的网络图像格式。它的一个主要特点是没有使用任何受法律保护的数据压缩算法。PNG包含了GIF的功能但增加了全颜色的支持，包括α值和16位灰度支持等功能。这些格式和更多图形文件格式的详细介绍见[MUR]。

15.2.2 印刷

创建印刷硬拷贝的方法之一是应用计算机系统上的标准彩色打印机，如第5章所述。因为这些打印机是在纸张上喷印色彩，通常是CMYK设备。打印机驱动程序会自动处理从RGB到CMYK的转换。在进入打印机前一般需要做些色彩校正工作，这与打印机使用的墨水有关。但专业的图像转换程序如Photoshop能够根据其内含的打印机配置文件调整图像，这对生成完美的图像非常有帮助。下面列出了一些与色彩直接输出相关的技术：

- 喷墨，彩色墨水微滴喷射到纸张上，要处理的问题有墨水被纸张吸收，以及墨迹扩散和过湿问题。
- 蜡染变换，使用彩色蜡棒，蜡棒熔化后在纸张上印上一层薄蜡膜。
- 染料渗透，使用渗透了饱和染料的薄片把彩色传递到纸张上。

打印机所使用纸张的质量也是影响打印质量的重要因素。便宜纸张通常多孔易渗墨，喷墨时容易扩散。相纸质量的纸张使用效果好得多。

这些设备都是基于像素的，有多个分辨率等级，但通常都高于计算机屏幕分辨率。所有这些技术也适用于胶片投影仪。

这里所指的印刷也包括标准丝刷支持的工艺。这类标准印刷技术复制彩色图像的工艺相当复杂。由于印刷品是一种透射色或减色媒介，在开始准备印刷材料前需要将RGB颜色的图像转换为CMYK颜色图像。印刷过程中需要为印刷作品制作印板，包括制作彩色插图5-7中所示的分色印版。在广泛使用的彩色复制技术里，分色板由*C*、*M*、*Y*和*K*单色光栅网板制成。这些单色网板按不同角度覆盖图像。得到的四张分色图写入照相底片，用于制作印板。这些分色印板使每种颜色墨水注到纸上时与其他颜色的干扰达到最小。一个放大的网板如图15-1所示，这里网板被放到如此之大，可以看到对应于*C*、*M*、*Y*、*K*颜色网板的角度。请注意，在网板中，点的位置是固定的，而大小有变化。这种技术有时称为"AM颜色"，取无线电技术中

的“调幅”（amplitude modulation）之意，（必须非常靠近彩色印刷品中的图像观察时，才能看到标准分色印板上的圆形印记。）为颜色敏感的图像生产分色板可说是一种艺术形式，通常由印刷工业中的分色专家负责处理。如果印刷品对彩色质量要求非常高，强烈建议作高质量的彩色校样，还建议采用低于原图像的分辨率，因为印板制作和印刷工艺不能在纸张上提供非常高的分辨率。

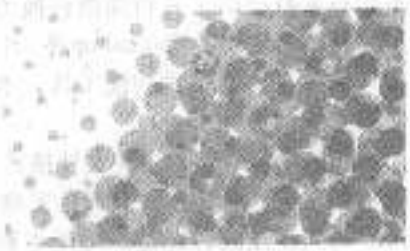


图15-1 放大的彩色图像中的C、M、Y和K色网板。参见彩图

在众多的生产彩色印刷图像的技术中还包括一种随机筛选技术。这种技术根据算法改变彩色点大小和位置来控制颜色的量和位置，产生比上述标准网板技术更高的分辨率和更佳的色彩饱和度。它有时称作“FM颜色”，取自无线电技术中的调频概念，因为颜色点趋向同样大小但其位置或者频率有变化。这个技术早在20世纪90年代在某些印刷厂开始出现，后来广泛应用于印刷工业。这一技术适用于有大量细节的图像，而标准固定角度网板技术无法表达这些细节。然而，它不如标准网板技术那样得到普遍应用，因此很难找到它的分色印板样本。图15-2比较了随机筛选技术和标准网板技术的图例，随机筛选技术中点的尺寸和位置是可变的。

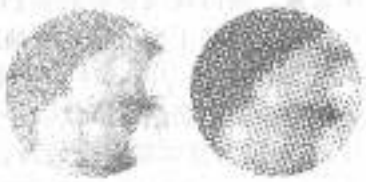


图15-2 随机筛选网板（左）和固定角度网板（右）的比较。参见彩图

15.2.3 胶片

有时需要给观众展现最高质量的图像——具有最饱和色彩和最高分辨率的图像。有时需要在不依靠计算机投影技术的情况下展现图像。这两种情形都可以考虑采用数字胶片记录器制作的标准摄影图像。这些设备使用高质量的黑白监视器、色轮和照相机来生成图像，并可采用任意类型的胶片（通常是用幻灯片、柯达彩色胶片和Ektachrome彩色胶片或类似产品）。由于胶片色彩的差异，请使用胶片记录器认可的胶片。

胶片记录器的结构如图15-3所示。黑白监视器分别生成每种颜色的黑白图像，通过色轮把黑白图像转换为彩色图像并记录在胶片上。因为黑白监视器无需阴罩来分辨不同颜色荧光点，也因为监视器可设计为长颈状，小屏幕对电子波进行极为精密的控制，所以它能提供非常高的分辨率；8K线解析度是相当好的标准，有的胶片记录器可以达到32K线解析度。所以，胶片记录器可以生成显示屏上无法达到的高分辨率的图像。

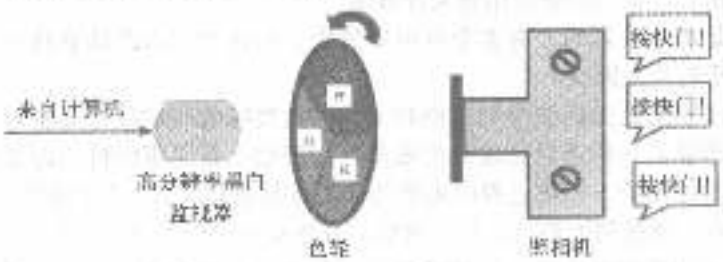


图15-3 数字胶片记录器示意图

胶片的问题比印刷少，因为可以直接处理图像而无需考虑分色片，通常使用RGB色彩模型。幻灯片由光线投射透过其自身而产生图像，如同幻灯片本身是类似于显示屏的发光媒介。

唯一要考虑的问题是采用相机提供的分辨率，还是在分辨率不够的情况下采用胶片记录器进行插值处理。

15.2.4 三维图像技术

第1章提到的立体视图技术是创建由不同视点生成的两幅视图，观察者汇聚这两幅图像来产生立体效果。这里介绍另外一些途径使眼睛看到的两幅图像能混合成单个三维视图。本节介绍能把信息存储在单幅图像中的三种技术。使得大多数人能用眼睛从图像中分辨出两幅独立的图像，并混合为一个三维图像。所有这些技术都需要使用独特的眼镜从单幅图像中分离出不同图像。 493

第一项技术要求特殊工艺来显示图像。用两台带正交偏振滤光镜的投影仪叠加显示两幅图像，佩戴一副带同样正交偏振滤光镜的观察者能分别看到这两幅图像。这是Geowall科技（http://geowall.geo.lsa.umich.edu/）使用的技术，由Geowall协会分享这项技术。Geowall系统可由现成的硬件和软件组装，除了细心的校正外无需更多工作。更多信息可访问Geowall网站。



图15-4 一幅色度－深度图像显示图例。参见彩图

第二项三维视图技术是基于对图像中的颜色进行有趣的操纵。在第8章讨论纹理图时描述了一些一维纹理图技术，其中有一种技术根据离眼睛的距离来确定颜色的纹理图，图像中越靠近眼睛的点颜色越红，离得越远越蓝。图15-4显示了一个例子，通过一副色度－深度眼镜观察这种图像时，这些颜色使得图像能自动聚合，所以大多数人可以看到图像的空间效果。色度－深度图像是普通的彩色图像，可以用前面介绍过的方式储存或者印刷。如果使用明亮的色彩和高质量打印机，这些图像将更加生动。

另外一个产生全彩色三维图像的有趣技术是使用红/蓝眼镜，这早在20世纪50年代的三维电影或三维漫画书中就使用过。这些图像称为立体图。它同时为左右眼生成两幅图像，图像混合时左眼图像使用红色信息，右眼图像使用蓝和绿色信息，得到一幅完整图像，如图15-5所示。混合后的图像看起来与图15-6所示图像非常相似，但通过红/蓝（或红/绿）眼镜将左眼配红色滤光镜就可以看到三维图像，且颜色与源图像相同。这种方法的效果直接，且容易实现，本章末尾将介绍如何用OpenGL实现这一技术。如上所述，立体图可以使用标准文件格式或者用印刷来保存，但JPEG也许不是合适的文件格式，因为它模糊了红色通道和蓝/绿通道间的内容。一些商业立体图的例子可以在http://www.studio3D.com/pages/anaglyph.html上找到。 494

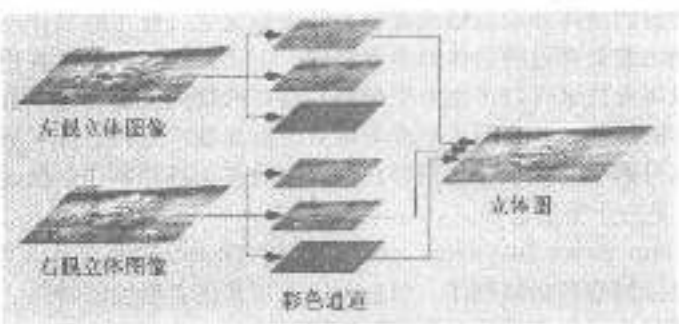


图15-5 由两幅图像混合生成立体图



图15-6 彩色立体图的例子。当彩色图像通过红/蓝或红/绿眼镜观察
495 时，就可以看到三维彩色图像。参见彩图

15.2.5 三维对象成型技术

有时候仅生成对象的图像是不够的，还可能需要用手指来触摸对象以理解它的形状，可能需要把两个对象握在一起看它们是否相配，或观察对象的形状来确定如何制作它。从计算机模型生成三维对象的技术称作三维打印或三维成型，它们通过特殊设备来制作。这样得到的模型可以当作后期制造对象的原型，或者图形模型和图像的实体模型。图15-7－15-10展示了几种通过三维成型技术（见图中的说明）生成的（3，4）绕环的照片。（3，4）绕环已经在讨论科学图形的章节中介绍过。生产这些硬拷贝的公司的联系方式在本章结尾处给出。当然还存在更早的三维硬拷贝技术，包括控制数控机床切割刀具的轨迹等类似技术，但这些内容超出了三维成型的范畴。

存在很多种生成原型对象的技术，但大多数技术创建的是层状实体模型。实体模型每层水平切割面的边界曲线形状由系统根据对象表面形状算出，再控制相关技术生成。图15-7－15-10展示了当前产生实体模型的几项技术。

Cubic Technologies公司（http://www.cubictechnologies.com）的叠层对象制造（LOM）技术先用背面带粘合剂的单纸片一层一层叠起来，然后用激光切割出每层的外形轮廓。纸片落在对象外面的部分被标记出来并可用简单工具（小心地）清除。叠层对象制造技术不能用来构造具有非常狭窄开口的对象，因为无法在对象内部清除多余的废料。LOM制作的对象的尖锐边缘容易损坏，尤其是顶层和底层部位更易损坏，但一般说来还是非常坚固的。图15-7显示了用LOM技术制作的绕环，表面的矩形网格由废料刻痕形成，波纹图样是对象中每层纸片烧制后露出的边缘所形成。图中的光泽表面由物体上的漆所形成。上漆是为了保护物体在处理、运输和其他使用过程中免遭损坏。

Z-Corp公司（http://www.zcorp.com）的三维打印机使用淀粉铺设薄层，在需要保留的部分落上液体胶合剂（在最近的版本中，胶合剂可以带有不同颜色）。这样制作的物体非常易碎，但可以使用渗透性好的液体如液态蜡或者强力胶来稳定它。使用蜡制作出来的物体可能仍然有点脆弱，但用强力胶处理过的物体则非常坚固。由于原始对象中没有用胶合剂处理的部分是普通粉末，所以用此技术可以产生中空带狭窄开口的物体。图15-8中是使用一台Z-Corp公司三维打印机制作的绕环，其表面不平滑的特质是由其制作原料是粉末所致。最新的Z-Corp打印机能制作更高分辨率的对象。使用彩色胶合剂的打印机能制作彩色表面的物体。在形状之外增加了色彩，是一个重要进步。

3D Systems（http://www.3dsystems.com/）公司的ThermoJet系统可以为对象的每一层注入不同材料。图中显示的绕环由蜡制作，但也可以选用其他更坚固的材料。此类部件必须有一
496 个支撑结构。支撑结构可以在设计对象的时候同时设计，也可以由ThermoJet系统自动提供。

对象的强度与所选用的材料有关。由于依赖支撑结构，所以难以制造中空小开口的对象。同样因为支撑结构，对象完成前需要清除底部支撑结构，并将光滑除部位的表面。图15-9显示了ThermaJet系统生成的绕环，可见对象中暗表面的轻微光泽。颜色与外观属性决定于所使用的材料。

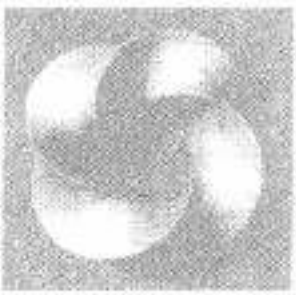


图15-7 LOM系统生成的绕环



图15-8 Z-Corp系统生成的绕环

3D Systems的立体平板印刷（stereolithography）系统用液态聚合物构建每一层，通过激光扫描每一层使需被保留的部分硬化。与ThermaJet系统一样，它也需要一个非常坚固的支撑结构，因为聚合材料被激光处理时有轻微的收缩现象。对象制造完成后必须清除支撑结构，并对表面做些抛光工作。液态聚合物可以通过开口排出，所以，此技术也适合制作中空物体等非凸对象。液态聚合物在成型之后非常坚固，所以，此技术产生的对象非常稳固。图15-10展示了立体光敏系统制造的绕环。



图15-9 3D Systems的ThermaJet系统生成的绕环



图15-10 3D Systems的立体光敏系统生成的绕环

三维成型技术有一个共同的问题，就是它们所生产对象的耐用性。某些技术能生产出非常耐用的对象；平板印刷系统使用的聚合物材料非常坚固，采用液态强力胶稳定的基于粉末的对象甚至可以承受一个人的重量。其他技术，如用蜡沉积或用蜡来稳定的粉末制作的对象，则相当脆弱。所有这些技术在制造小的或尖锐的部件时，或多或少都会有破损的危险。选择什么样的技术与所制作对象需求的强度有关。

497
|
498

15.2.6 STL文件

所有这些三维成型技术都需要使用STL（平板印刷）文件格式，从中读取数据来控制操作。这是个非常简单的文件格式，很容易由图形程序生成。（3，4）绕环所使用的STL文件有2 459 957个字节，文件的首尾两部分在下面列出。文件由面（facets）组成，每个面都有可选的法向量和面的顶点列表。如果使用显式坐标定义顶点，可以将内容直接写入STL文件，而不用调用图形输出函数。唯一需要注意的是对齐两个三角形上的顶点，就像边界三角形上的

顶点一样，必须具有完全相同的值，大多数三维成型系统都需要特别注意轻微的裂口。最好的方法是让边界上顶点使用保存的值，而不要通过计算来得到，因为计算可能产生微小的舍入误差。STL文件格式的详细介绍见附录C。

```
solid
    facet normal -0.055466 0.024069 0.000000
        outer loop
            vertex -5.000010 -0.000013 -1.732045
            vertex -5.069491 -0.160129  1.688424
            vertex -5.000009 -0.000013  1.385635
        endloop
    endfacet
    facet normal -0.055277 0.019631 0.002301
        outer loop
            vertex -5.069491 -0.160129 -1.688424
            vertex -5.000009 -0.000013  1.385635
            vertex -5.034917 -0.159669 -1.342321
        endloop
    endfacet
    ...
    facet normal -0.055466 -0.024069 0.000000
        outer loop
            vertex -5.000009 0.000014 1.385635
            vertex -5.069491 0.160130 1.688424
            vertex -5.000010 0.000014 1.732045
        endloop
    endfacet
endsolid
```

15.2.7 视频

视频对表现生动的计算机图形学成果而言是非常重要的媒介，只有它能展现运动，传递许多重要信息。同时，视频也是所能得到的最受限制的媒介之一，至少从20世纪的早期视频诞生起，直到进入21世纪这段时间一直如此，这里重点讨论NTSC视频，但与PAL或SECAM视频都有相似之处。如果读者所在地区采用PAL或SECAM制式，则需要仔细核对这里的内容能在多大程度上实际应用。

在处理视频时有些重要问题要考虑。第一是分辨率，NTSC视频的分辨率甚至比最小的计算机分辨率还要低。NTSC标准建议隔行扫描的525条水平扫描线，其中480条可见，所以分辨率一般为640×480。当然，许多电视机都有调整功能，所以无需担心出现空白边缘。隔行扫描意味着每1/30秒只显示一半水平线，所以必须避免只有单个像素宽的水平线，以防止闪烁。许多电视机亮色能力很差，必须避免只有单个像素宽的垂直线，以避免渗色。如果在设计时以上面提到的分辨率的一半来规划工作，则可以得到最佳的效果。

视频的第二个问题是色域。与由RGB构成的色彩不同，NTSC电视的色彩标准更多考虑适应有限的传播带宽和兼容黑白电视机（NTSC标准制定于20世纪30年代末，远早于彩色电视或现代电子学和其他科技的出现）。NTSC色彩标准是YIQ三组元模型，但这三个组元完全针对视频领域而定义。Y组元表示亮度，它占据信号带宽的大部分。I组元表示橙到蓝成分，所占带宽略高于Y组元带宽的1/3。Q组元表示紫到绿成分，所占带宽略高于I组元的1/3。视频中最佳的颜色看起来总是欠饱和，这是受到技术标准的制约。下表列出了视频图像中各颜色组元更精确的带宽和水平分辨率数据。

成分	带宽	分辨率/扫描线
Y	4.0 Mhz	267
I	1.5 Mhz	96
Q	0.6 Mhz	35

为了使图像得到尽可能好的水平分辨率，必须使跨越扫描线的图元具有不同的亮度，以及更多地关注橙—蓝成分而非紫—绿成分。如果要确切知道颜色在YIQ模型下如何改变，下面的转换矩阵可帮助理解从图像颜色到视频颜色的转换规律：

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

当然，视频的问题远不止这些。除了NTSC（或者PAL、SECAON）制式还存在许多数字视频格式，如QuickTime、AVI或MPEG，这些都是计算机上的显示格式。数字视频采用RGB颜色，所以在实际播放到电视屏幕之前没有NTSC那么多问题。实际上，MPEG II是DVD的视频标准，它提供了转换到NTSC的另一种可选方式。

从长远看，电视必然发展到完全兼容计算机的数字格式。高清晰电视（HDTV）标准将直接支持RGB颜色、高分辨率和非隔行扫描图像，所以大家都很乐意看到这里介绍的内容将被淘汰。但在目前，这些问题仍然是每个计算机图形学者把图像转换成视频时必须面对、必须解决的问题。

15.2.8 数字视频

利用合适的工具将动画制作为视频是非常容易的。在第11章讨论动画时介绍了这个过程，如果使用它作为硬拷贝技术，则需要考虑更多问题。数字视频可能是系统相关的，因为某些视频文件格式只能在某个平台上运行。为了使尽可能多的观众应用这种硬拷贝，建议采用平台无关的格式。MPEG格式有多个级别，每个级别比上个级别具有更高的压缩率。许多数字创作工具可以在数字视频上增加声道，从一个动画序列转换到另外一个序列，给电影增加字幕或其他文本信息。当视频开发完毕后，可以写入CD-R或者DVD-R媒介与他人共享，或者放到网上供人下载或者作为流媒体播放。在线视频是一种扩展的技术，这里不做进一步讨论。 501

15.3 支持硬拷贝的OpenGL技术

15.3.1 捕获输出窗口内容到文件

OpenGL带有捕获颜色缓存的工具，利用这些工具可以随时将屏幕显示内容“导出”。关键的OpenGL函数是glReadBuffer(BUFNAME)——指定读取的缓存，和glReadPixels(...)，它的参数指定了读取缓存和写入阵列的方式，一旦缓存写入阵列，就可以直接通过文件的相关技术把阵列写入文件。如果所使用的OpenGL实现或其他工具包带有相关功能函数，则可以将阵列保存为标准格式（例如JPG），这非常有用。

一个简单的实现上述操作并将阵列保存为原始RGB文件的函数如下。注意最外层循环的顺序，它对行的扫描按照从上到下的顺序，而缓存的下标从左下角而非左上角开始。在这一点上也许各种OpenGL的实现不会有不同，但清楚这点会更有帮助。

```
#define BUF_WIDTH 512
#define BUF_HEIGHT 512
static GLubyte bufImage[WIDTH][HEIGHT][3];

// 函数将前屏幕缓冲区内容读入一个数组中，将数组名称维数信递给函数。得到
// 的文件包含原始的RGB数据，任何兼容该文件格式的应用程序（如Photoshop）
//
// 均可以打开并操作该文件。

void saveWindow(char *outfile, int BUF_WIDTH, int BUF_HEIGHT)
{
```

```
    FILE * fd;
    GLubyte ch;
    int i,j,k;

    fd = fopen(outfile, "w");
    glReadBuffer(GL_FRONT);              // 设置读取缓冲区
    glReadPixels(0, 0, WIDTH, HEIGHT, GL_RGB, GL_UNSIGNED_BYTE,
                 bufImage);
    for (i=WIDTH; i>0; i--)              // 对于每一行
      {
        for (j=0; j<HEIGHT; j++) // 对于每一列
        {
            for (k=0; k<3; k++) // 读取像素的RGB分量
              {
                  ch = bufImage[i][j][k];
                  fwrite(&ch, 1, 1, fd);
              }
        }
      }
    fclose(fd);
}
```

502

15.3.2 用OpenGL生成立体图

生成立体图的技术更多地使用了OpenGL的高级颜色特性。前面关于视图的图1-15显示了立体视图，它需要左眼图像和右眼图像，两幅图像都使用RGB颜色。可以从这两幅独立的图像中提取颜色信息，生成一幅单独的人眼可辨的三维图像。首先生成场景的左眼图像和右眼图像，保存到独立的颜色阵列中。然后逐个读取两幅图像中的每个像素进行组装：从左眼图像的像素中读取红色信息，从右眼图象中读取蓝色和绿色信息。然后混合为此像素点的颜色。最后显示混合的图像，并通过红/蓝或红/绿眼镜观看三维效果。

为得到组装图像所用的颜色阵列，需要使用捕获颜色缓存到文件的一些技术。首先，在后缓存中生成左眼图像并保存到阵列，如平常一样生成或载入图像，但不调用glutSwapBuffers()，这时图像就保留在后缓存中。用函数glReadBuffer(GL_BACK)指定读取后缓存，然后使用函数glReadPixels(0,0,width,height,GL_RGB,GL_UNSIGNED_BYTE,left_view)将后缓存中内容读取到left_view阵列中。

假定读取整个窗口（左下角为(0,0)），窗口宽width个像素，高height个像素，使用RGB模式。保存数据的阵列left_view能容纳3*width*height个无符号字节（GLUByte类型）。阵列参数需要转换为（GLvoid*）类型传入。也可以使用别的像素数据类型，但GL_UNSIGNED_BYTE看起来最简单。如果使用扫描输入的图像，可以用相同类型的数据读入阵列。

存储完左眼图像后，对右眼图像做同样的工作，也是输出到后缓存，然后存储到right_view阵列。现在内存中有两个RGB颜色值的阵列。建立第三个同样数据类型的merge_view阵列，循环遍历像素，从left_view阵列中读取红色值，right_view阵列中读取绿色和蓝色值，复制到merge_view阵列中。

现在有了如图15-6所示的混合颜色阵列，使用glDrawPixels(width,height,GL_RGB,GL_UNSIGNED_BYTE,merge_view)将阵列写入到后缓存中。接着交换缓存来显示图像，也可以将立体图像写入前面描述过的文件。

图15-6显示的是由带灰度的左右视图构造的样本图像，这意味着图像的任意部分都带有红绿蓝颜色信息，都能在合成的立体图像中辨识出来。如果图像中有个区域的红色通道或者绿蓝通道为零（很可能是人工合成图像而非扫描图像），则立体图在这个区域不能给眼睛任何信息，从而导致丧失立体效果。

15.4 小结

本章介绍了几种硬拷贝技术：电子图像、印刷、胶片、视频和三维成型。它们都是保存计算机图形成果的可选方案，必须认真考虑成果面向的观众对象和创建硬拷贝的原因，再选择合适方案展现成果或作为档案保存。随着时间的推移，这里给出的例子需要补充更多新技术，因为生成能永久保留工作成果的记录是永恒不变的主题，必须清醒地认识到这一点，并关注新工艺和新技术。当新技术来临、旧技术被淘汰时，用旧技术保存的图像需要改用新技术，以便继续保存。

本章的要点在于让读者明白如何选择合适的媒介，以及如何依据媒介来设计图像或可视化工作。在设计中更多地依靠实验，因为很多硬拷贝媒介要通过实验才能掌握，而不是依靠文字知识。

15.5 本章的OpenGL术语表

本章有九个新的OpenGL术语，都与在指定缓存中读写像素有关。

OpenGL函数

glDrawPixels(...)：将一块像素写入到缓存。需要大量参数指定像素写入的方式

glReadBuffer(...)：指定一个颜色缓存作为像素读取的源

glReadPixels(...)：从缓存中读取一块像素到数组中，需要大量参数指定像素读取和存储的方式

15.6 思考题

1. 找一张前面项目完成的图像（最好带有大量细节的纹理图，或者有一些尖锐线条），并用屏幕抓取工具或其他可以精确保存屏幕图像的工具保存这张图像。用Photoshop打开这幅图像，或用其他可以打开和保存不同图像格式的图像处理程序打开这幅图像。再把这幅图像用下列格式GIF、TIFF和JPEG保存起来。比较这几种格式文件的大小和图像质量。
2. 用寸镜（一种珠宝店使用的放大镜）或其他放大镜观察几种不同类型的彩色硬拷贝图像。研究这些图像的构成。观察照相底片或传统相机（不是数字相机或胶片记录器）的照片、高质量艺术书籍和杂志的彩页、更多普通杂志的彩页、报纸彩图和黑白图。评价它们的线条分辨率和颜色分布。你能在这些印刷品中看出色点的图案吗？它们是AM色还是FM色？如果使用AM色，你能确定分色片的角度吗？
3. 用数字胶片记录器生成一张图像的幻灯片，把这张幻灯片的图像与照相机拍摄的幻灯片图像以及屏幕显示图像进行比较。幻灯片上的图像是否比屏幕上的图像具有更高的分辨率？幻灯片图像是否带有人工痕迹例如可见到扫描线，而照相机幻灯片却没有？
4. 访问本章列出的提供一些快速成型工具的公司网站，评价这些工具在耐久性、细节精良度和为对象使用支撑结构的困难程度等方面的性能指标。
5. 许多城市都有许多公司在工作中使用一些快速成型工具。看你的导师能否为你安排参观这些公司和考察制作的原型对象。观察这些原型并按前面所列的三个方面作出评价。考察这些公司如何使用这些原型对象。

15.7 实验题

1. 用任意方式生成一张数字图像，最好是用屏幕截取方法获得一幅有明暗着色或者使用纹理贴图的图像，颜色变化越多越好。用全功能图象处理程序如Photoshop打开此图像，毫无疑问它是RGB图像。现在将它转换为CMYK格式，并打印出图像的四个颜色的分色图。它们是四幅灰度图像。如果使用正确的墨水混合在一起可以形成一张彩色印刷图像。检查这四幅分色图，判断如何用CMYK墨水混合出RGB

三原色。如果你有HLS或HSV图像的经验，你能鉴别出黑色（K）分色片与颜色光照的关系吗？

2. 回到当地用三维成型技术制作原型对象的公司的话题，看能否找到一个公司能为学生制作的模型做一个原型对象，准备好模型和它的STL文件，在公司生产线有空闲时间时带到公司做好准备工作。

3. （立体图）如果可以找到红/蓝或红/青眼镜，再找到一对可用的立体图像。立体图像可以是一对计算得到的图像或一对数码照片。一个好的选择是用stereopticon（一种老的立体观察方式）制作的幻灯片－在两个视点得到的两幅灰度图像。扫描这对图像，通过混合两幅图像颜色通道的方式得到一幅立体图，用本章介绍的方法，通过眼镜观察立体图包含的颜色和深度信息。

505

参考文献和资源

[AMES] Ames, Andrea L., David R. Nadeau, and John L. Moreland, *VRML 2.0 Sourcebook*. Wiley, 1997.

[AN02] Angel, Ed, *Interactive Computer Graphics with OpenGL*, third edition. Addison-Wesley, 2002.

[BAK] Baker, Steve, A Brief MUI User Guide, distributed with the MUI release.

[BAN] Banchoff, Tom, et al., "Student-Generated Software for Differential Geometry," in Zimmermann and Cunningham, eds., *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991, pp. 165–171.

[BRA] Braden, Bart, "A Vector Field Approach in Complex Analysis," in Zimmermann and Cunningham, eds., *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991, pp. 191–196.

[BR95] Brown, Judith R., et al., *Visualization: Using Computer Graphics to Explore Data and Present Information*. Wiley, 1995.

[BR99] Robson Brown, K. A., A. Chalmers, T. Saigol, C. Green, and F. d'Errico, "An automated laser scan survey of the Upper Palaeolithic rock shelter of Cap Blanc." *Journal of Archeological Science*, 1999.

[BUC] Buchanan, J. L., et al., "Geometric Interpretation of Solutions in Differential Equations," in Zimmermann and Cunningham, eds., *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991, pp. 139–147.

[CO93] Cohen, Michael, and John Wallace, *Radiosity and Realistic Image Synthesis*. Morgan Kaufmann, 1993.

[CU90] Cunningham, Steve, "3D Viewing and Rotation Using Orthonormal Bases," in Glassner, ed., *Graphics Gems*, Academic Press, 1990, 516–521.

[CU92] Cunningham, S., and R. J. Hubbold, eds., *Interactive Learning Through Visualization*. Springer-Verlag, 1992.

[CU01] Cunningham, Steve, and Michael J. Bailey, "Lessons from Scene Graphs: Using Scene Graphs to Teach Heirarchical Modeling," *Computers & Graphics* 25 (2001), pp. 703–711.

[DE] Devaney, Robert L., and Linda Keene, eds., "Chaos and Fractals, The Mathematics Behind the Computer Graphics," Proceedings of Symposia in Applied Mathematics, vol. 39, American Mathematical Society, 1988.

[DU] Durett, H. John, ed., *Color and the Computer*. Academic Press, 1987.

[EB] Ebert, David, et al., *Texturing and Modeling*, third edition. Morgan Kaufmann, 2003.

[ELL] Ellson, Rich, et al., "Plastic Injection Molding," in http://archive.ncsa.uiuc.edu/SCMS/Metascience/Articles/MS_Plastic-Injection-Molding-Ellson.html

[FO] Foley, James D., et al., *Computer Graphics Principles and Practice*, second edition. Addison-Wesley, 1990.

[GG1] Glassner, Andrew S., ed., *Graphics Gems*. Morgan Kaufmann, 1990.

[GG2] Arvo, James, *Graphics Gems II*. Academic Press, 1991.

[GG3] Kirk, David, *Graphics Gems III*. Academic Press, 1992.

[GG4] Heckbert, Paul S., *Graphics Gems IV*. Academic Press, 1994.

[GG5] Paeth, Alan, *Graphics Gems V*. Academic Press, 1995.

[GL] Glassner, Andrew S., ed., *An Introduction to Ray Tracing*. Academic Press, 1989.

[GR] Green, Phil, and Lindsay MacDonald, *Colour Engineering: Achieving Device Independent Colour*. Wiley, 2002.

[HA] Hall, Roy, *Illumination and Color in Computer Generated Imagery*. Springer-Verlag, 1988.

[HE] Hearn, Donald, and M. Pauline Baker, *Computer Graphics*, second edition, C version. Prentice-Hall, 1997.

[HIL] Hill, F. S., Jr., *Computer Graphics Using OpenGL*, second edition. Prentice-Hall, 2001.

[JEN] Jensen, Henrik Wann, *Realistic Image Synthesis Using Photon Mapping*. A K Peters, 2001.

[JO] Joy, Kenneth I., et al., eds., *Tutorial: Computer Graphics: Image Synthesis*. IEEE Computer Society Press, 1988.

[LA] Landau, Rubin H., and Manuel J. Páez, *Computational Physics: Problem Solving with Computers*. Wiley, 1997.

[LE] Levkowitz, Haim, *Color Theory and Modeling for Computer Graphics, Visualization, and Multimedia Applications*. Kluwer Academic, 1997.

[LI] Lischinski, Dani, "Combining Hierarchical Radiosity and Discontinuity Meshing," SIGGRAPH 94 Course Notes, Course 28.

[LU] Luebke, David, et al., *Level of Detail for 3D Graphics*. Morgan Kaufmann, 2004.

[MAC] MacDonald, Lindsay W., and M. Ronnier Luo, eds., *Colour Image Science: Exploiting Digital Media*. Wiley, 2002.

[MCC] McCullen, Dave, "Using Infrared for Residential Energy Surveys," InfraMation 2004 Proceedings.

[MUR] Murray, James D., and William vanRyper, *Encyclopedia of Graphics File Formats*, second edition. O'Reilly & Associates, 1996.

[NAD] Nadeau, D. R., J. D. Genetti, S. Napear, B. Pailthorpe, C. Emmart, E. Wesselak, and D. Davidson, "Visualizing Stars and Emission Nebulas," *Computer Graphics Forum*, March 2001.

[PAI] Pailthorpe, Bernard, and Richard Carson, "Economics Gets a New Look at the History of the U.S. Economy," Gather/Scatter online, 1997; http://www.sdsc.edu/GatherScatter/GSspring97/pailthorpe.html

[PER] Perlin, Ken, "An Image Synthesizer," *Computer Graphics* 19(3), Proceedings of SIGGRAPH 85, July 1985, 287–296.

[PIE] Pietgen, Heinz-Otto, and Dietmar Saupe, eds., *The Science of Fractal Images*. Springer-Verlag, 1988.

[POC] Pocock, Lyon, and Judson Rosebush, *The Computer Animator's Technical Handbook*. Morgan Kaufmann, 2002.

[POR] Porter, Thomas and Tom Duff, "Compositing Digital Images," *Computer Graphics* 18(4), SIGGRAPH 84, July 1984.

[ROG] Rogers, David F., and J. Alan Adams, *Mathematical Elements for Computer Graphics*, second edition. McGraw-Hill, 1990.

[ROS] Rost, Randi, *The OpenGL Shading Language*, second edition. Addison-Wesley, 2006.

[SHI] Shirley, Peter, and R. Keith Morley, *Realistic Ray Tracing*, second edition. A K Peters, 2003.

[SHR] Shreiner, Dave, ed., *OpenGL Reference Manual*, third edition. Addison-Wesley, 2000.

[SIL] Sillion, François, and Claude Puech, *Radiosity and Global Illumination*. Morgan Kaufmann, 1994.

[SOV] Sova, Dawel, *Galileo's Daughter*. Walker & Co., 1999.

[SOW1] Sowrizal, Henry, Kevin Rushforth, and Michael Deering, *The Java3D 3D API Specification*. Addison-Wesley, 1995.

[SOW2] Sowizral, Henry A., and David R. Nadeau, *Introduction to Programming with Java 3D*, SIGGRAPH 99 Course Notes, Course 40.

[SPE] Spence, Robert, *Information Visualization*. Addison-Wesley/ACM Press Books, 2001.

[SVR] The SIGGRAPH Video Review (SVR), an excellent source of animations for anyone wanting to see how images can communicate scientific and cultural information through computer graphics, as well as how computer graphics can be used for other purposes. See `http://www.siggraph.org/SVR/` for information.

[TAL] Tall, David, "Intuition and Rigour: The Role of Visualization in the Calculus," in Zimmermann and

Cunningham, eds., *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991, pp. 105–119.

[THO] Thorell, R. G., and W. J. Smith, *Using Computer Color Effectively: An Illustrated Reference*. Prentice-Hall, 1990.

[UP] Upstill, Steve, *The RenderMan Companion*. Addison-Wesley, 1990.

[ViSC] McCormick, Bruce H., Thomas A. DeFanti, and Maxine D. Brown, eds., *Visualization in Scientific Computing*, *Computer Graphics* 21(6), November 1987.

[VDB] van den Bergen, Gino, *Collision Detection in Interactive 3D Environments*. Morgan Kaufmann, 2003.

[vSEG] von Seggern, David, *CRC Standard Curves and Surfaces*. CRC Press, 1993.

[WAT] Watt, Alan, and Fabio Policarpo, *3D Games: Real-time Rendering and Software Technology*. Addison-Wesley/ACM SIGGRAPH Series, 2001.

[WA2] Watt, Alan, and Mark Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*. Addison-Wesley, 1992.

[WOL] Wolberg, George, *Digital Image Warping*. IEEE Computer Society Press, 1990.

[WO98] Wolfe, Rosalee, ed., *Seminal Graphics: Pioneering Efforts That Shaped the Field*. ACM SIGGRAPH, 1998.

[WO00] Wolfe, R. J., *3D Graphics: A Visual Approach*. Oxford University Press, 2000.

[WOO] Woo, Mason, et al., *OpenGL Programming Guide*, third edition (version 1.2). Addison-Wesley, 1999.

[WY] Wyszecki, G., and W. S. Styles, *Color Science*, second ed. Wiley, 1982.

[ZIM] Zimmermann, Walter, and Steve Cunningham, *Visualization in Teaching and Learning Mathematics*, MAA Notes Number 19, Mathematical Association of America, 1991.

附　录

A　PDB文件格式

国家蛋白质数据库文件（Protein Data Bank, PDB）格式来自于世界蛋白质数据库（wwPDB, http://www.wwpdb.org/）。这个蛋白质数据库十分复杂，拥有的信息量远远超出学生作业的需要。我们将从有关这种文件格式的参考文献中提取一些用于简单分子显示的信息。从化学学科的观点出发，应该鼓励学生看更长的文件描述，以便了解在创建一个完整分子记录时需要记录多少信息。

在PDB文件中有两种至关重要的记录：原子位置记录和联结描述记录。它们详细说明了分子中的原子以及它们之间的联结关系。通过阅读这些记录，我们能够将所需的信息填充到内部数据结构中，用于创建分子显示。这里给出的有关原子位置（ATOM）和联结描述（CONECT）记录的信息来自于参考文献。还有另一种用关键字HETATM描述原子的记录，这里没有介绍，大家可以通过查阅PDB格式指南获得相关信息。指南可以在RCSB PDB（http://www.rcsb.org）中找到。本附录是根据PDB格式版本（1992标准版）编写的，该标准版的URL是http://www.rcsb.org/pdb/file_formats/pdb/pdbguide2.2/PDB_format_1992.pdf。更新、更复杂的标准扩展了这个版本。

ATOM记录

ATOM记录以埃为单位描述了标准残基的原子坐标。同时，它们也描述了每个原子的占有率和温度因子。元素符号会出现在ATOM记录中。

记录格式

列	数据类型	字段名称	定义描述
1-6	Record name	"ATOM "	
7-11	Integer	serial	原子序列号
13-16	Atom	name	原子名
17	Character	altLoc	交替位点标识
18-20	Residue name	resName	残基名
22	Character	chainID	链标识符
23-26	Integer	resSeq	残基序列号
27	AChar	iCode	残基的插入代码
31-38	Real(8.3)	x	X坐标
39-46	Real(8.3)	y	Y坐标
47-54	Real(8.3)	z	Z坐标
55-60	Real(6.2)	occupancy	空间大小
61-66	Real(6.2)	tempFactor	温度因子
73-76	LString(4)	segID	段标识符，左对齐
77-78	LString(2)	element	元素符号，右对齐
79-80	LString(2)	charge	原子的电荷

由于原子除了标准名称之外还有其他的名称，因此“原子名”字段变得很复杂。在PDB文件示例中，我们一直避免使用与周期表中标准原子名不同的名称。这意味着我们不能使用来自化学数据库的所有PDB文件。如果化学程序要求使用一种特殊的分子作为示例，而该分子的数据文件却使用其他的原子名称格式，那么我们需要修改这些示例的readPDBfile()函数。

示例

```
         1         2         3         4         5         6         7         8
12345678901234567890123456789012345678901234567890123456789012345678901234567890
ATOM      1  C   1      -2.053     2.955     3.329  1.00  0.00
ATOM      2  C   1      -1.206     3.293     2.266  1.00  0.00
ATOM      3  C   1      -0.945     2.371     1.249  1.00  0.00
ATOM      4  C   1      -1.540     1.127     1.395  1.00  0.00
ATOM      5  C   1      -2.680     1.705     3.426  1.00  0.00
ATOM      6  C   1      -2.381     0.773     2.433  1.00  0.00
ATOM      7  O   1      -3.560     1.422     4.419  1.00  0.00
ATOM      8  O   1      -2.963    -0.435     2.208  1.00  0.00
ATOM      9  C   1      -1.455    -0.012     0.432  1.00  0.00
ATOM     10  C   1      -1.293     0.575    -0.967  1.00  0.00
ATOM     11  C   1      -0.022     1.456    -0.953  1.00  0.00
ATOM     12  C   1      -0.156     2.668     0.002  1.00  0.00
ATOM     13  C   1      -2.790    -0.688     0.814  1.00  0.00
ATOM     14  C   1      -4.014    -0.102     0.081  1.00  0.00
ATOM     15  C   1      -2.532     1.317    -1.376  1.00  0.00
ATOM     16  C   1      -3.744     1.008    -0.897  1.00  0.00
ATOM     17  O   1      -4.929     0.387     1.031  1.00  0.00
ATOM     18  C   1      -0.232    -0.877     0.763  1.00  0.00
ATOM     19  C   1       1.068    -0.077     0.599  1.00  0.00
ATOM     20  N   1       1.127     0.599    -0.684  1.00  0.00
ATOM     21  C   1       2.414     1.228    -0.914  1.00  0.00
ATOM     22  H   1       2.664     1.980    -0.132  1.00  0.00
ATOM     23  H   1       3.214     0.453    -0.915  1.00  0.00
ATOM     24  H   1       2.440     1.715    -1.915  1.00  0.00
ATOM     25  H   1      -0.719     3.474    -0.525  1.00  0.00
ATOM     26  H   1       0.827     3.106     0.281  1.00  0.00
ATOM     27  H   1      -2.264     3.702     4.086  1.00  0.00
ATOM     28  H   1      -0.781     4.288     2.207  1.00  0.00
ATOM     29  H   1      -0.301    -1.274     1.804  1.00  0.00
ATOM     30  H   1      -0.718    -1.756     0.076  1.00  0.00
ATOM     31  H   1      -4.617     1.581    -1.255  1.00  0.00
ATOM     32  H   1      -2.429     2.128    -2.117  1.00  0.00
ATOM     33  H   1      -4.464     1.058     1.509  1.00  0.00
ATOM     34  H   1      -2.749    -1.794     0.681  1.00  0.00
ATOM     35  H   1       1.170     0.665     1.425  1.00  0.00
ATOM     36  H   1       1.928    -0.783     0.687  1.00  0.00
ATOM     37  H   1      -3.640     2.223     4.961  1.00  0.00
ATOM     38  H   1       0.111     1.848    -1.991  1.00  0.00
ATOM     39  H   1      -1.166     0.251    -1.707  1.00  0.00
ATOM     40  H   1      -4.560    -0.908    -0.462  1.00  0.00
```

Conect记录

CONECT记录描述已给出坐标的原子间的连接关系。这种连接关系是以该记录的原子序列号的形式表现的。

记录格式

列	数据类型	字段名称	定义描述
1-6	Record name	"CONECT"	
7-11	Integer	serial	原子序列号
12-16	Integer	serial	原子序列号
17-21	Integer	serial	原子序列号

（续）

列	数据类型	字段名称	定义描述
22-26	Integer	serial	原子序列号
27-31	Integer	serial	原子序列号
32-36	Integer	serial	氢键连接的原子序列号
37-41	Integer	serial	氢键连接的原子序列号
42-46	Integer	serial	盐桥连接的原子序列号
47-51	Integer	serial	氢键连接的原子序列号
52-56	Integer	serial	氢键连接的原子序列号
57-61	Integer	serial	盐桥连接的原子序列号

示例

```
         1         2         3         4         5         6         7
1234567890123456789012345678901234567890123456789012345678901234567890
CONECT    1179     746    1184    1195    1203
CONECT    1179    1211    1222
CONECT    1021     544    1017    1020    1022    1211    1222    1311
```

在本附录开始时我们就提到PDB文件是很复杂的，大多数示例都十分庞大。图A-1中的文件是其中最简单的一种，描述的是肾上腺素分子。它是adrenaline.pdb所附带的材料之一。

```
HEADER  NONAME 08-Apr-99                                                NONE 1
TITLE                                                                   NONE 2
AUTHOR  Frank Oellien                                                   NONE 3
REVDAT   1  08-Apr-99    0                                              NONE 4
ATOM     1  C           0  -0.017   1.378   0.010  0.00  0.00      C+0
ATOM     2  C           0   0.002  -0.004   0.002  0.00  0.00      C+0
ATOM     3  C           0   1.211  -0.680  -0.013  0.00  0.00      C+0
ATOM     4  C           0   2.405   0.035  -0.021  0.00  0.00      C+0
ATOM     5  C           0   2.379   1.420   0.013  0.00  0.00      C+0
ATOM     6  C           0   1.169   2.089   0.002  0.00  0.00      C+0
ATOM     7  O           0   3.594  -0.625  -0.035  0.00  0.00      O+0
ATOM     8  O           0   1.232  -2.040  -0.020  0.00  0.00      O+0
ATOM     9  C           0  -1.333   2.112   0.020  0.00  0.00      C+0
ATOM    10  O           0  -1.177   3.360   0.700  0.00  0.00      O+0
ATOM    11  C           0  -1.785   2.368  -1.419  0.00  0.00      C+0
ATOM    12  N           0  -3.068   3.084  -1.409  0.00  0.00      N+0
ATOM    13  C           0  -3.443   3.297  -2.813  0.00  0.00      C+0
ATOM    14  H           0  -0.926  -0.557   0.008  0.00  0.00      H+0
ATOM    15  H           0   3.304   1.978  -0.019  0.00  0.00      H+0
ATOM    16  H           0   1.150   3.169   0.008  0.00  0.00      H+0
ATOM    17  H           0   3.830  -0.755  -0.964  0.00  0.00      H+0
ATOM    18  H           0   1.227  -2.315  -0.947  0.00  0.00      H+0
ATOM    19  H           0  -2.081   1.509   0.534  0.00  0.00      H+0
ATOM    20  H           0  -0.508   3.861   0.214  0.00  0.00      H+0
ATOM    21  H           0  -1.037   2.972  -1.933  0.00  0.00      H+0
ATOM    22  H           0  -1.904   1.417  -1.938  0.00  0.00      H+0
ATOM    23  H           0   3.750   2.451  -1.020  0.00  0.00      H+0
ATOM    24  H           0  -3.541   2.334  -3.314  0.00  0.00      H+0
ATOM    25  H           0  -4.394   3.828  -2.859  0.00  0.00      H+0
ATOM    26  H           0  -2.674   3.886  -3.309  0.00  0.00      H+0
CONECT    1    2    6    9    0                                  NONE 31
CONECT    2    1    3   14    0                                  NONE 32
CONECT    3    2    4    8    0                                  NONE 33
CONECT    4    3    5    7    0                                  NONE 34
CONECT    5    4    6   15    0                                  NONE 35
CONECT    6    5    1   16    0                                  NONE 36
CONECT    7    4   17    0    0                                  NONE 37
CONECT    8    3   18    0    0                                  NONE 38
CONECT    9    1   10   11   19                                  NONE 39
CONECT   10    9   20    0    0                                  NONE 40
CONECT   11    9   12   21   22                                  NONE 41
CONECT   12   11   13   23    0                                  NONE 42
CONECT   13   12   24   25   26                                  NONE 43
END                                                             NONE 44
```

图A-1 一个简单分子的PDB文件格式

B CT文件格式

CT文件的结构很简单，可以分成以下几个部分：文件头、计数行、原子块、键块和其他信息。文件的前面三行是文件头，它包括分子的名称（第1行）；用户名、程序名、日期和其他信息（第2行）；注释（第3行）。文件的下一行是计数行，开始的两个数值分别代表分子的数目和键的数目。接下来的几行是原子块，用来描述分子中各个原子的特性：每一行中包括原子的*X,Y,Z*坐标和化学符号。再接下去的几行是键块，同样用来描述分子中各个键的特性：每一行中包括构成键的两种原子的序号（从1开始）以及该键是否为单键、双键或者三键等标志。这些行之后是有关分子的额外描述信息，在我们的讨论中将不会用到。图B-1给出的是采用简单CT文件格式描述的分子文件示例。所使用的文件格式的完整描述日期标注为2005年6月，可以从Elsevier MDL(http://www.mdl.com)提供的CT文件格式文档中找到。

很明显，文件中有许多化学家感兴趣的信息，而事实上这是一个极为简单的文件示例，但对我们的项目作业而言，我们感兴趣的只是分子的几何结构，因此在读文件时可跳过文件中的其他信息。

```
L-Alanine (13C)
CSMACCS-II01691115362D 1 0.00366 0.00000 0

 6 5 0 0 1 0 3 V2000
-0.6622   0.5342  0.0000 C   0 0 2 0 0 0
 0.6220  -0.3000  0.0000 C   0 0 0 0 0 0
-0.7207   2.0817  0.0000 C   1 0 0 0 0 0
-1.8622  -0.3695  0.0000 N   0 3 0 0 0 0
 0.6220  -1.8037  0.0000 O   0 0 0 0 0 0
 1.9464   0.4244  0.0000 O   0 5 0 0 0 0
1 2 1 0 0 0
1 3 1 1 0 0
1 4 1 0 0 0
2 5 2 0 0 0
2 6 1 0 0 0
M CHG 2 4 1 6 -1
M ISO 1 3 13
M END
```

图B-1 一个简单分子的CT文件格式

C STL文件格式

STL（有时称为StL）是一种由加利福尼亚的Valencia 3D系统公司开发的用于描述3D硬拷贝系统信息的文件格式。STL的名称来源于立体成型。一种3D硬拷贝技术，在第15章中提到这种格式也被其他硬拷贝技术所采用。这里的信息改编自俄勒冈州3D硬拷贝中心的远程制造设施文档（C3H）。

.stl或立体成型格式是一种用于制造业的ASC II或二进制文件格式。它是一个三角形曲面列表，这些三角形曲面表示计算机生成的实体模型。这是一种标准输入，适用于在第15章硬拷贝介绍的大多数快速原型机器。二进制文件格式是最紧凑的，但我们只介绍ASC II文件格式，因为这种格式更容易理解，并且作为学生项目作业的输出也更容易生成。

ASC II.stl文件必须始于小写字母的关键字solid，终止于endsolid关键字。在这些关键字之间的是用于定义实体模型的面的三角形列表。每个三角形描述定义了一个实体曲面外向的法向量，三角形的三个顶点都使用*x,y,z*分量表示。这些值使用笛卡儿坐标且为浮点数。三角形的值应该都为正数，并且包含于模型体内。可以构建的最大模型体会随着机器的不同而不同，

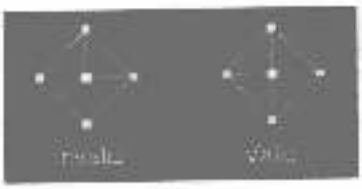
但典型尺寸是x为0～14英寸，y为0～10英寸，z为0～12英寸。应该考虑通过缩放或旋转模型以优化构建时间、强度和碎片清除等性能指标。法向量是一个基于原点长度为1的单位向量，如果没有法向量，那么大多数软件都会使用右手定律来生成它们。如果没有给出法向量信息，那么可以使用法线，而且法向量的三个值应该设为0.0。接下来给出STL文件中的一个三角形的ASCII描述示例。

```
solid
...
facet normal 0.00 0.00 1.00
    outer loop
        vertex 2.00 2.00 0.00
        vertex -1.00 1.00 0.00
        vertex 0.00 -1.00 0.00
    endloop
endfacet
...
endsolid
```

当用计算机程序生成三角形坐标时，一些本该具有相同坐标的点会因为累积舍入误差而稍微不同。例如，如果通过每增加一个圆周角计算圆上的一个点，那么绕圆一周很可能会得到一个与起始点稍微不同的终止点。这会在对象定义中留下间隙，从而导致按文件制造的产品出现问题。文件检查软件会注意到点之间的不同，然后提示我们对象不是封闭的，但通常它会自动修复对象中的小间隙，因此我们需要确认确实不希望出现间隙的要求。

顶点到顶点规则

在STL文件中最普遍的错误是和顶点到顶点规则不兼容。STL规范要求所有相邻的三角形有两个相同的顶点，如图C-1所示，左边的图像显示顶部三角形包含了四个顶点，且外层的顶点没有和其他三角形共享。而对于下面的两个三角形，每一个都包含一个顶点和第四个无效的顶点，为了使STL规范满足顶点到顶点规则，顶部三角形必须细分成如右图所示。



图C-1 无效顶点（左），相邻三角形没有共享中心顶点，有效顶点（右）

索　引

索引中的页码为英文版书页码，书中页边标出原书页码。

按键事件（Keypress events）
按钮（Buttons），11-12，80
　MUI按钮（MUI），281
　单选按钮（radio），5,281
凹凸纹理贴图（Bump map），229
八象限（Octant(s)），160
半空间（Half-spaces）
　负半空间（negative），170-171
　正半空间（positive），170-171
包围对象（Bounding objects），175-176
颜色饱和度（Saturation, of a color），185
背景色（Background colors），193-194
背面剔除（Backface culling），437
变换（Transformation(s)）
　组合变换（composite），89-90
　视点变换（eyepoint），149-150
　模型变换（modeling），7,8,24,69,85-96
　模型视图变换（modelview），15,141
　节点变换（nodes），117
　在OpenGL中的变换（in OpenGL），141-151
　透视变换（perspective），45-46
　投影变换（projection），141
　简单变换（simple），144-146
　栈变换（stacks），91,92-93,146-148
　视图变换（viewing），9,24,37-39,113-115
变换组（Group,transform），106
变形（Morphing），402
标称数（Nominal data），4,333
标量场（Scalar field(s)）
　一维标量场（1D），348,369
　二维标量场（2D），348-350,359-360,369
　三维标量场（3D），348
标签（Labels），104-105,143-144
布告板技术（Billboard(s)），174
裁剪（Clipping），11-12,80
　裁剪平面（planes），137
　场景图中的裁剪（in scene graphs），108
　视域体裁剪（on the view volume），46-48
菜单（Menus）
　活动菜单（active），259
参数（Parameters）
参数曲面（Parametric surfaces），343-347,369
参数曲线（Parametric curves），162,342-343,368-369
参数线段（Parametric line segment），161
侧抑制（Lateral inhibition），181
插值（Interpolation）
场景（Locales），106
场景图（Scene graph(s)）
　场景图动画（animation in the），401
　场景图编码（coding and），118-121
　纹理映射（texture mapping and），296-297
窗口（Window）
窗口到视口映射（Window-to-viewport mapping），13
蛋白质数据库（Protein Data Bank），84,355
等值面（Isosurfaces），101,358
点（Point(s)），160-162
　控制点（control），449450,454-456,463-465
　绘制点（drawing），72,127-128
　齐次点（homogeneous），71
迭代函数系统（Iterated function systems）
　收缩映射（contraction mappings），479-480
　生成函数（generating functions），480-482
叠层对象制造技术（Laminated Object Manufacturing(LOM)technique），496
顶点（Vertex(vertices)），70,134
顶点列表（Vertex list），81
顶点数组（Vertex arrays），134
动画（Animation）
　翻动型动画书（flip book），412-414

基于帧的动画（frame-based），397-398,404
帧速率（frame rates），406-407
插值（interpolation），402-406,418
关键帧（keyframe），404
运动模糊（motion blurring），411
运动轨迹（motion traces），410-411,420-421
运动物体（objects,moving），411-415
同事级（peer-level），424
个人级（personal-level），424
专业级（presentation-level），424
过程动画（procedural），401
实时动画（real-time），397-398
场景图动画（in the scene graph），401
时间走样（temporal aliasing），407-408
时间控制（time, controlling），415
视觉交流（visual communication and），408-410
西洋镜（zoetrope），412-414
动画的控制时间（Time in animation,controlling），415
动力学（Dynamics.*See* Animation）
多边形（Polygon(s)）
凸多边形（convex），76,171-172
多边形剔除（culling），436-438
多边形的绘制（drawing），76-77,133-134
多边形着色处理（shading of a），218,224-229
多层贴图（Multitexturing），305-307,315
多面体（Polyhedron(polyhedra)），70,171
凸多面体（convex），173
多面体绘制（drawing），77
多面体的面（faces of a），70
二次函数对象（Quadric objects,GLU.*See* GLU quardic objects）
二维蒙特卡罗仿真（2D Monte Carlo simulations）
二维屏幕坐标（2D screen coordinates），13
二维投影（2D projections）
二维纹理图（2D texture maps），293-294
二维向量场（2D vector fields），260-261
二元空间划分（Binary space partitioning），439
发射光（Emissive light），219-220
发射色（Emissive colors），187
法向（Normals）
曲面片的法向（patch），458
曲面的法向（surface），78-79,218,219
反走样（Antialiasing），134-135
方向光源（Directional lights），221,236
方向向量（Direction vector），160
仿射平面（Affine plane），71
仿真（Simulations）
非多边形（逐像素）图形学（Nonpolygon(per-pixel)graphics）
迭代函数系统（iterated function systems），479-482
茹利亚集（Julia sets），483-485
芒德布罗集（Mandelbrot sets），482-485
光线投射（ray casting），472-475,478-479
光线跟踪（ray tracing），472-473,475-477
体绘制（volume rendering），478-479
非均匀有理B样条（NURBS(non-uniform rational B-splines)），459
分形（Fractals）
伪分形（forgeries），351-352
分子显示（Molecular display）
封闭线段（Line loops），72-73
辐射度（Radiosity），231-232
杆状细胞（Rods(cells in the retina)），180-181
感兴趣单元（Item of interest），268
高度场（Height field），229
高宽比（Aspect ratio），35,50
高性能图形技术（Graphics techniques, highperformance）
碰撞检测（collision detection），443-444
建模（modeling），430-435
原理（principles of），429-430
绘制（rendering），436-443
各向异性着色处理（Anisotropic shading），229-230,474
关键帧（Key frames），404
关键帧动画（Keyframe animation），404
关节（Joints），450
观察参考点（View reference point），34
光的明度（Lightness,of a color），185
光（源）和光照（Light(s) and lighting）
环境光（ambient），214,215,218
光线衰减（attenuation and），221,236

光的颜色 (color of), 220
漫反射光 (diffuse), 214,215-216,218
直接光 (direct), 216
方向光 (directional), 221,236
发射光 (emissive), 219-220
全局光 (global), 231-233
局部光 (local), 214,233-241
位置光 (positional), 220
光的属性 (properties of), 220-221
光辐射度 (radiosity), 231-232
镜面反射光 (specular), 214,216-218
聚光灯 (spotlights), 220-221
表面法向量 (surface normals), 218,219
光滑度 (Smoothness,degree of(granularity)), 137
光栅图像处理器 (Raster image processor(RIP)), 297
光栅化处理 (Rasterization process)
光线跟踪 (Ray tracing)
光线投射 (Ray casting)
光泽度 (Shininess), 216-218
光泽度系数 (Shininess coefficient), 216-217
光子映射 (Photon mapping), 232-233
归一化设备坐标 (Normalized device coordinates (NDC)), 49
归一化向量 (Normalized vectors), 163,219
轨迹球 (Trackballs), 251
函数 (Functions)
基函数 (basis), 450
回调函数注册和函数列表 (callback registering and list of), 255-258
生成函数 (generating), 480-482
有理函数 (rational), 459
参数曲面 (Function surface), 82
函数作图 (Function graphing)
滑动条 (Sliders)
水平滑动条 (horizontal), 282
垂直滑动条 (vertical), 282
画家算法 (Painter'algorithm), 51,438
环境 (Environment)
环境纹理图 (maps), 316-318
环境光 (Ambient light)
缓存 (Buffer(s))
累积缓存 (accumulation), 41,421-423
后缓存 (back), 52,273-274
深度缓存 (depth), 14,50
双缓存 (double), 52,59
前缓存 (front), 52
选择缓存 (selection), 268-270,272
w缓存 (w-), 51
z缓存 (z-), 50,438
回调函数 (Callback), 16,249
绘制 (Rendering)
光栅化处理 (rasterization process), 382-389
体绘制 (volume), 478-479
绘制模式 (Render mode), 268
基于帧的动画 (Frame-based animation)
关键帧 (Key frames), 404
帧间 (tweening), 404
极限处理 (Limit processes), 347-348,369
几何建模 (Geometric modeling)
几何数据 (Geometry)
几何压缩 (compression), 71
几何节点 (nodes), 117
三维几何处理流水线 (3D pipeline), 7-13
几何压缩 (Compression,geometry), 71
记录 (Record(s))
事件记录 (event), 248
命中记录 (hit), 269-270
计算机鼠标 (Mice, computer), 250
加强色 (Emphasis colors), 193
加色系 (Additive colors), 187
建模 (Modeling), 67-68,70-71
建模图 (modeling graphs), 106,115-121
场景图 (scene graphs), 106-108,110-121
样条建模 (spline), 448-469
曲面建模 (of surfaces), 82-83
建模空间 (Modeling space)
建模图 (Modeling graphs), 106
交互 (交互式编程) (Interaction(s)(interactive programming)), 247-248
操纵杆 (joysticks), 251
键盘 (keyboards), 250
鼠标 (mice), 250
MUI (MUI), 277-286

拾取（picking），252-253,268-277
轨迹球（trackballs），251
视觉交流和交互式编程（visual communication and），253-254
节点（Nodes）
节点属性（attribute），117
几何节点（geometry），117
组节点（group），106,117
形状节点（shape），106-107,117
变换节点（transformation），117
结点（Knots），450
局部光照处理（Local lighting）
科学计算可视化（Visualization in Scientific Computing），2
科学可视化（Scientific visualization），331-332
控件（Widgets），278
控制点（Control points），449,450,463-465
库仑定律（Coulomb's law），97-98,340
立方体（Cube(s)）
立体成型（Stereolithography），394-395
立体视图（Stereo viewing）
双目立体视图（binocular），52-54
立体图（Anaglyphs），494-495
粒度（Granularity(degree of smoothness)），137
粒子系统（Particle system(s)），398
两点透视（Two-point perspective），45
亮度（Luminance）
点度，纹理图（Intensity, texture maps and），310
列表（List(s)）
显示列表（display），94,150-151
边表（edge），81
面表（face），81
三角形表（triangle），80-81
顶点列表（vertex），81
滤波操作（Filter(s)）
线性滤波（linear），304
放大滤波（magnification），304
缩小滤波（minification），304
最近点滤波（nearest），304
马赫带（Mach banding），189-190,192
漫反射（Diffusion）
漫反射光（Diffuse light）
蒙特卡罗仿真（Monte Carlo simulations）
灭点（Vanishing point），44
明度（Illumination. *See* Light(s) and lighting）
RGBA模型（RGBA color model）
色彩分离（separations），491-492
颜色图谱（shape and），198
减色系（subtractive），187
透射（transmissive），187
模型变换（Modeling transformations）
模型库（Model Bank Collection），83
默比乌斯带（Möbius bands），347
碰撞检测（Collision detection）
片段（Fragment(s)）
平面（Plane(s)）
仿射平面（affine），71
割平面（cutting），101-102
平移（Translations），69,88,141
屏幕空间（Screen space），48
屏幕坐标（显示坐标）（Screen coordinates(display coordinates)），13,25
普通四边形（Quadrilateral,general），74
气体定律和扩散原理（Gas laws and diffusion principles），353-354
气相色谱仪（Gas chromatograph），356
切片（Slices），137
求值器（Evaluators）
一维求值器（1D），459-460
二维求值器（2D），459-462
球（Sphere(s)），
包围球（bounding），175-176
球面坐标（Spherical coordinates），174-175
区间数（Interval data），4,333
曲面（Surface(s)）
扩展曲面片到曲面（extending a patch to a），457-458
函数曲面（function），82
等值面（iso-），358
曲面建模（modeling of），82-83
参数曲面（parametric），343-347,369
样条曲面（spline），448,456-459,466-469
曲面片（Patch）
曲线（Curves）

参数曲线（parametric），162,342-343,368-369
样条曲线（spline），448-456,465-466
全局光照（Global lighting），231-233
软件事件（Software events），249
三点透视（Three-point perspective），45
三角扇形（Triangle fans），73-74,129-130
三角形（Triangle(s)）
包围三角形（bounding），176
三角形重心（circumcenter of a），176
三角形外接圆（circumcircle of a），176
三角形绘制（drawing），73,129
三角形序列（sequence of），73-74，129-132
三角形列表（Triangle list），80-81
三角形条带（Triangle strips），73-74,129-132
三维成型（打印）（3D prototyping(printing)），496,498-499
三维打印（成型）（Printing(prototyping),3D），496,498-499
三维几何流水线（3D geometry pipeline）
裁剪（clipping），11-12
投影变换（projections），9-11
场景与视图（scene and view），7
三维几何流水线步骤（stages），7
三维眼坐标（3D eye coordinates），8-9
三维模型坐标（3D model coordinates），7-8
三维世界坐标（3D world coordinates），8
二维眼坐标（2D eye coordinates），12-13
二维屏幕坐标（2D screen coordinates），13
三维模型坐标（3D model coordinates），7-8
三维世界坐标（World coordinates,3D），8,24
三维系统（3D Systems）
立体平板印刷系统（stereolithography system），496
扫描变换（Scan conversion），382
扫描线（Scanline），381
色调（Tones,color），186
色度（Hue），185
色度－深度图像（ChromanDepth images），315-316,319,494
色谱（Range(gamut),of color），190
色域（Gamut(range),of color），190
色泽（Tints），186
深度（Depth）
深度缓存（buffering），14,50
颜色深度（color），189-190
消除深度比较（comparisons,avoiding），438
深度测试（testing），116
时间反走样（Temporal antialiasing），408
时间走样（Temporal aliasing），407-408
拾取（选择对象）（Picking(selecting objects)），252-253
拾取矩阵（Pick matrix），272-273
拾取容差值（Tolerance,pick），273
事件（Events），247
显示事件（display），16,255
事件处理程序（handlers），249
空闲事件（idle），16,26,255,262,414
按键事件（keypress），263-264
菜单事件（menu），249,255-256,264-265
鼠标事件（mouse），249,257-258,266-268
事件队列（queue），248
事件记录（record），348
重显示事件（redisplay），262
改变窗口事件（reshape），16,255
场景图和事件（scene graph and），254
软件事件（software），249
特殊事件（special），256
系统事件（system），250
事件术语（terminology），248-249
定时器事件（timer），258-259,262-263
窗口事件（window），250
视点（Eyepoint）
视截体（Frustum），10
视觉交流（Visual communication(s)）
视觉交流建模（modeling for），96-105
科学可视化（scientific visualization），331-332
视口（Viewport），38
视平面（View plane），9
视图变换（Viewing transformation），9,37
视图变换过程（Viewing process），14-15,33
双缓存（double buffering），52
隐藏面（hidden surfaces），50-51,58-59
立体视图变换过程（stereo），52-54,59-60
视图分支（View branch），107,110,112,117

视野宽度（Breadth of field），35
视域体（View volume），41,42
　视域体裁剪（clipping on the），46-48
　三维视域体（3D），11
收缩映射（Contraction mappings），479-480
鼠标事件（Mouse events）
属性节点（Attribute nodes），117
数据（Data）
　区间数（interval），4,333
　标称数（nominal），4,333
　有序数（ordinal），4,333
　体数据（volume），358-360
数字微分分析法（DDA(Digital Differential Analyzer)algorithm），383-384
衰减（Attenuation），221,236
双向反射分布函数（Bidirectional reflection distribution function(BRDF)），230,474
四边形条带（Quad strips），74-76
四面体（Tetrahedron,GLUT），139
四维作图（4D graphing）
　向量场的四维作图（of vector fields），360-361
　体数据的四维作图（of volume data），358-360
随机筛选（Stochastic screening），492
缩放（Scaling），69,87,141
索引颜色（Indexed color），192,207
体绘制（Volume rendering），478-479
体数据四维作图（Volume data,4D graphing of），358-360
体素（Voxels），358-359
贴图和映射（Maps and mapping）
　凹凸（bump），229
　收缩映射（contraction），479-480
　数字高程（digital elevation），229,348
　光子映射（photon），232-233
投影（Projection(s)），33-34
　组立投影（organizing），37
　正交投影（orthographic），9,12,39-41,43,58
　平行投影（parallel），9,41
　透视投影（perspective9-10,12,39,41,43-48,58）
　三维投影到四维空间（3D,into 4D space），347
　投影变换（transformations），141
　二维投影到四维空间（2D,into 4D space），346-347
　二维投影到三维空间（2D,into 3D space），334,342-347,360
　向量投影（of vectors），164
　投影视域体（view volumes of），41-42
透明色（Transparent colors），191-192
　半透明面（partially transparent faces），201-203,206-207
透射色（Transmissive colors），187
凸包（Convex hull），172
凸多边形（Convex polygons），76,171-172
凸多面体（Convex polyhedra），173
凸四边形（Convex quadrilaterals. *See* Quads）
图（Graphs）
　建模图（modeling），106,115-121
　场景图（scene106-109,111-121,223,296-297,401）
图例（Legends），104,143-144
图形卡，图形加速卡（Graphics cards(accelerators)）
拓扑（Topology），13
外部码（Outcodes），47
外观（Appearance）
位置光源（Positional lights），220
文本框（Text box(es)），5
　MUI文本框（MUI），281-282
文件格式（File formats）
纹理（Textures），14
纹理空间（Texture space），293
纹理拉伸（Clamping,texture maps and），310
纹理图（Texture map(s)）
　纹理拉伸（clamping and），310
　创建纹理图（creating），297-301
　纹理映射环境（environment,defining the texture），309-310
　环境纹理图（environment maps），316-318
　MIP技术（MIP），304-305
　多纹理（multitexturing），305-307,315
　绘制纹理（rendering and），390
　合成纹理图（synthetic），297-299
纹元（Texels），293
雾化（Fog）
　雾色（color of），434-435

雾密度（density of），434
雾化模式（modes of），434
雾化开始距离和结束距离（starting and ending），434
西洋镜（Zoetrope），412-414
细节层次（Level of detail(LOD)），431-433
显示列表（Display list(s)），94,150-151
显示坐标（屏幕坐标）（Display coordinates (screen coordinates)），13,25
线段序列（Line strips），72-73
相交三角形（Intersecting triangles），176-177
向量（Vector(s)）
两向量夹角（angle between two），163
两向量叉积（向量积）（cross (vector)product of two），164-166
向量方向（direction），160
两向量点积（标量积）（dot(scalar)product of two），163-164
向量长度（length of a），165
归一化向量（normalized），163,219
向量投影（projection of），164
反射向量（reflection），166-167
向量变换（transformations of），167-169
单位向量（unit），163
向量场（Vector fields）
向上方向（Up direction），35
象限（Quadrants），160
像素（Pixels(picture element[s])）
像素走样（aliasing and），77-78
像素反走样（antialiasing and），78
像素着色器（shaders and），230-231
亚像素（sub-），78
行为建模（Behavior, modeling），84-85
形状、颜色（Shape,color and），198
形状节点（Shape node(s)），117
旋转（Rotation(al)），87-88,141
旋转向量（vectors and），168-169
选择对象（Selecting objects.*See* Picking）
选择缓存（Selection buffer）
选择名字（Selection name），271
选择模式（Selection mode），268
选择模型（Selection model），271
颜色（Color(s)），14
加色表（additive），187
颜色走样（aliasing of），77-78,189,192
背景色（background），193-194
CMYK模型（CMYK(cyan-magenta-yellow-black) model），187-189,491
颜色深度（depth of），189-190
真彩色（direct），189
发射色（emissive），187
颜色加强（emphasis），193
颜色域（gamut(range) of），190
颜色混合（Color blending）
颜色渐变（Color ramps），4,195,208
RGB颜色模型（RGB(red,green,blue) color model）
颜色值（Value of a color），185
眼坐标（Eye coordinates）
三维眼坐标（3D），8-9,24
二维眼坐标（2D），12-13,25
样条（Splines）
样条建模（Spline modeling）
样条曲线、插值（Spline curves,interpolation and），448-454
一点透视（One-point perspective），44
一维纹理图（1D texture maps），293
阴影（Shadows），218-219
隐藏面（Hidden surfaces），50-51
印刷（Print），491-492
硬拷贝（Hardcopy），
数字图像（digital images），490-491
胶片（film），493
印刷（print），491-492
STL文件（STL file），499-500
三维图像技术（3D image techniques），493-495
三维成型（印刷）（3D prototyping(printing)），496,498-499
视频（video），423,500-501
用户界面（User interface），247
游戏手柄（Joysticks），251
有理函数（Rational functions），459
有序数（Ordinal data），4,333
原始RGB文件（RGB file,raw），490
运动（Motion）

运动模糊 (blurring), 411
运动轨迹 (traces), 410-411,420-421
噪声 (Noise)
1/f噪声 (1/f), 300
白噪声 (white), 300
噪声函数 (Noise function(s))
栈 (Stack(s)), 137
名称栈 (name), 269,271
变换栈 (transformation), 91,92-93,146-148
折射 (Refraction), 476
帧速率 (Frame rates), 406-407
直射光 (Direct light), 216
直线 (Lines), 160-162
直线段 (Line segment(s))
参数化直线段 (parametric), 161
直线段序列 (sequence of), 72-73
逐像素图形学 (Per-pixel graphics.See Nonpolygon graphics)
主事件循环 (Main event loop), 249
注册 (Registering), 249
锥状细胞 (Cones(cells in the retina)), 180-181
着色处理 (Shading)
彩色锥着色处理 (of a cone), 204
像素着色器 (pixel shaders), 230-231
多边形着色处理 (of a polygon), 218,224-229
平滑着色处理 (smooth), 204,218,225-227
字形 (Glyph), 102
自然色 (naturalistic color), 195
伪彩色 (pseudo-), 99,192,195
走样 (Aliasing), 77-78,189,192
时间走样 (temporal), 407-408
组合变换 (Composite transformation), 89-90
组节点 (Group node), 106,117
作图 (Graphing)
数据驱动作图 (data-driven), 363-365
四维作图 (4D), 358-361,372-374
函数作图 (function), 339-342,367-368
高维作图 (higher-dimensional), 361-363, 374-375
向量场作图 (of vector fiels), 360-361
体数据作图 (of volume data), 358-360
三维作图 (3D graphing), 361-363
坐标 (Coordinates)
直角坐标 (Cartesian(rectangular)), 159,173,174-175
柱面坐标 (cylindrical), 174
显示 (屏幕) 坐标 (display(screen)), 13,25
眼坐标 (eye), 8-9,12-13,24,25
模型坐标 (model), 7-8,68-69
极坐标 (polar), 173
球面坐标 (spherical), 174-175
曲面片的纹理坐标 (texture,for a patch), 458
世界坐标 (world), 8,24
齐次坐标 (Homogeneous coordinates), 71-72
坐标系 (Coordinate systems), 158-160

α blending (α混合), 182,190-191
α channel (α通道), 182
AM color (AM (调幅) 颜色), 491
Bernstein basis (伯恩斯坦基), 451,457,465
Bézier splines (Bézier样条), 448-449,451-453,457,459,464,465
Blancmange function (Blancmange函数), 247
Boy's Surface (凸曲面), 343-344
Bresenham algorithm (Bresenham算法), 384-386
B-spline (B样条), 459
Catmull-Rom splines (Catmull-Rom样条), 403,448-449
CMYK(cyan-magenta-yellow-black) color model (CMYK颜色模型), 187-189
Cohen-Sutherland clipping (Cohen-Sutherland裁剪算法), 47
CrystalEyes glasses (CrystalEyes 立体眼镜), 394-395
CT file format (CT文件格式), 514-515
Flat shading (Flat着色处理)
FM color (FM (调频) 颜色), 492
GLU quadric objects (GLU 二次曲面对象), 238
Google Earth (Google地球), 248
Gouraud shading (Gouraud着色处理), 226
HLS(hue-lightness-saturation) color model (HLS (色度－亮度－饱和度) 颜色模型), 185-187

double cone（双圆锥），206
HSV(hue-saturation-value)color model（HSV（色度－饱和度－亮度值）颜色模型），185-187
cone（圆锥），204-206
Java3D（Java 3D语言），106-108,116-117,121
Julia sets（茹利亚集），483-485
Klein bottle（克莱因瓶），346-347
Lambert's law（朗伯定理），216
LZW(Lempel-Ziv-Welch)file compression（LZW文件压缩），490
Mandelbrot sets（芒德布罗集），482-485
Maple algebra system（Maple代数系统），343-344
Marching cubes（Marching cube算法），358-359
Mathematica（Mathematica函数），346
MDL Information Systems(Elsevier MDL)（MDL信息系统（Elsevier出版社的MDL信息系统）），355,514
MIP texture maps（MIP纹理贴图），304-305
Menu bars, MUI（MUI菜单条），280-281
MUI(Micro User Interface)（MUI（微用户接口））
buttons（按钮），281
installing（安装），286
menu bars（菜单条），280-281
sliders,horizontal（水平滑动条），282
sliders,vertical（垂直滑动条），282
text boxes（文本框），281-282
text labels（文本标签），282-283
MUI functions（MUI函数）
OpenGL,modeling in.*See* Modeling in OpenGL（OpenGL中的建模）
OpenGL extensions（OpenGL扩展），28
OpenGL functions（OpenGL函数）
OpenGL Utility Library(GLU)（OpenGL实用库），137
OpenGL Utility Toolkit(GLUT)（OpenGL实用工具），137
PDB(Protein Data Bank) file format（PDB（蛋白质数据库）文件格式），509
ATOM records（ATOM记录），509-511
CONECT records（CONECT记录），511-513
Phong illumination model（Phong光照模型），214
Phong shading（Phong着色处理），228-229,381
POVRay（POVRay光线跟踪程序），477
Rayshade（Rayshade光线跟踪软件），477
RGB(red,green,blue)color model（RGB颜色模型）
RGBA color model（RGBA颜色模型）
Sierpinski gasket/attractor（Sierpinski垫/吸引子），479,485
Snell's law（Snell定理），476
STL file format（STL文件格式）
Takagi fractal surface（Takagi分形曲面），348
ThermaJet system（ThermaJet系统（喷热系统）），496,498
w-buffer（w缓存），51
z-buffers（z缓存），50,438
z-fighting（z混淆），51