OpenSceneGraph
快速入门指导

对跨平台开源
场景图形 API 的
初步介绍

[美] Paul Martz 著

王 锐　钱学雷 译

本书谨献给

每一个场景图形开发技术的初学者。

目录

译者序

OpenSceneGraph 图形系统是一个基于工业标准 OpenGL 的软件接口，它让程序员能够更加快速、便捷地创建高性能、跨平台的交互式图形程序。本书是 OSG 开发小组推荐的 OpenSceneGraphx.x 版入门级指南。

本书首先介绍了场景图形的概念，OSG 的历史和开源组织、它的能力、如何获取和正确安装 OSG，以及一些简单示例程序的运行；然后深入探讨了一些 OSG 的内部管理机制和实用技术，包括内存管理、场景图形结构、OSG 的状态属性和模式控制、较复杂的场景图形系统、图形节点的概念和特性、I/O 接口、以及文字添加等功能的具体介绍；最后重点探讨了如何将 OSG 集成到用户程序中去的各种关键技术，包括场景的渲染、视角的改变、图像节点的选取以及在系统运行时动态地修改场景图形数据的技术。

本书要求读者有一定的 C++语言基础和数学知识，适合所有对 OSG 编程感兴趣的读者阅读。

自 Sutherland 在 1965 年提出“Ultimate Display”并设计实现了世界上第一个交互式图形系统后，计算机图形学及人-机交互技术取得了难以想象的进步。在过去 20 年当中，随着计算机图形加速技术的快速发展，由计算机实时传输、处理、可视化亿级比特数据、并为终端用户提供三维交互式场景已经成为现实。而以虚拟现实为代表的显示技术和图形信息管理技术也取得了很大进步。目前，全球有数以千计的公司的业务涉及或正在使用三维交互式图形系统，而这些软件在显示、模拟、仿真、计算机辅助设计、科学数据可视化及分析领域的应用随处可见。这些应用已成为科研开发、工业生产中的重要工具。

从系统开发人员的角度看，相比工业标准 OpenGL 或其它的图形库，OpenSceneGraph 的优点显而易见。除了开源和平台无关性以外，它封装并提供了数量众多的提升程序运行时性能的算法、针对包括分页数据库在内的几乎所有主流数据格式的直接数据接口、以及对脚本语言系统 Python 和 Tcl 的支持，特别的，支持脚本语言系统的意义不仅限于用户可以使用除 C++语言以外的工具进行

图形系统的开发，事实上，对弱类型计算机语言的支持将突破现有交互式图形系统在人-机交互性能方面的最终限制。

为了将这本书尽快带给国内读者，尤其是当 OSG 开发小组在 SIGGRAPH 2007-OSG BOF 上宣布将本书的翻译工作列为 OSG 开发项目之一后，本书译者带着紧迫感，在原书作者的帮助下，经过一个多月的努力，终于完成了本书的翻译工作。在此要特别感谢 Robert Osfield、Don Bruns，以及 OSG 开发小组主要成员之一、本书作者 Paul Martz，没有他们和其它小组成员的共同努力，OSG 不可能成为当今应用最为广泛的图形系统开发库，当然，也就不可能有这本小书的问世。

译者简介

王锐：清华大学制造工程研究所研究员。2006 年毕业于清华大学精密仪器与机械学系。现主要从事数字控制技术、虚拟现实与虚拟产品等领域的开发与研究工作。本人在虚拟设计论坛 WWW.VRDEV.NET 的注册用户名为 array，主要联系方式：电子邮箱 wangray84@gmail.com，MSN 为 wangray84@hotmail.com。

钱学雷：清华大学制造工程研究所博士后。主要从事数字控制技术、CAD/CAM/CAPP 集成技术、虚拟现实与虚拟产品开发、数控系统计算机仿真等领域的科研工作。

前言

本书是一本对于 OpenSceneGraph（OSG）的简明介绍。OSG 是一个跨平台的开源场景图形程序开发接口（API）。本书特别地针对 OSG 1.3 的版本。OSG 在 3D 应用程序的层级中扮演着重要的角色。它作为中间件（middleware）为应用软件提供了各种高级渲染特性，IO，以及空间结构组织函数；而更低层次的 OpenGL 硬件抽象层（HAL）实现了底层硬件显示的驱动。

一直以来，OSG 都是以源代码作为可阅读的文档资料。OSG 的发布版本中包含了一些示例程序，用于介绍各种不同渲染效果的实现，以及 OSG 与终端用户软件的集成方法。有相当部分的开发者可以通过这些示例程序，以及使用调试工具深入了解 OSG 的核心，成长为熟练的 OSG API 程序员。

尽管根据以往的经验，源代码足可起到编程文档的作用，但是它仍然不能替代正式格式的编程文档。图形和表格是编程手册中常见的易于理解的教学工具，但是它们在源代码中几乎无法体现。正因为 OSG 的迅速发展和愈发复杂的体系，对于它的新用户来说，在缺乏参考文档的情况下学习 OSG 所需的时间也就越长，这是我们所不愿看到的。事实上，在这本书面世以前，由于编程文档的匮乏，一部分开发者已经对 OSG 的成熟性和稳定性产生了疑问，怀疑它是否能胜任专业级应用程序的开发工作。

2006 年中期，Don Burns 和 Robert Osfield 认识到了编写 OSG 书籍的重要性。Don 的客户之一，计算机图形系统开发公司（CGSD），要求编写一定量的 OSG 图书和文档。Don 将这一文档开发工作转交给 Paul Martz，而 Robert 则建议第一本 OSG 的书籍应当是免费且通俗易懂的。因此，《OpenSceneGraph 快速入门指导》诞生了。这本书是一个简短的编程指南，它介绍了基本和核心的 OSG API 函数。它同时也是一系列计划出版的 OSG 书籍的第一部，OSG 的文档将随着这个系列的发行而逐渐完善。《OpenSceneGraph 快速入门指导》的编写目的如下。

- 快速且高效地向新的 OSG 开发者介绍 OSG 的基础知识。
- 使开发者对于 OSG 的发行和源代码结构有更加深入的认识。
- 介绍常用的 OSG API 函数及其用途。

- 指导读者学习 OSG 的源代码及相关文档。

基于开放源代码的理念，《OSG 快速入门指导》一书的 PDF 版本将不必支付任何费用即可获得。但是，读者也可以通过购买全彩薄页印刷版本的方式，向 OSG 的开发团体进行捐助。用户可以访问 Lulu.com 网站并搜索 OpenSceneGraph，通过网络订购的方式来获取本书。

http://www.lulu.com

本书装订版本的收入将用于文档的修订工作，以保证本书版本是最新的。

无论用户是免费下载，还是购买了本书的装订版本，用户的反馈意见都将对本书的修订工作产生极大的帮助。请您将相关的修改意见发布在 OSG 用户邮件列表中。关于用户邮件列表的相关信息，请参见“附录：从这里开始”。

关于本书的最新修订版本，请登陆《OpenSceneGraph 快速入门指导》网站：

http://www.openscenegraph.org/osgwiki/pmwiki.php/
Documentation/QuickStartGuide

上面的网址包括了本书最新版本的获取方式，书中示例程序的源代码，以及其他一些相关的出版信息。

适宜读者群

本书的内容不多，而缩短其篇幅并不是一件容易的事。本书将主要介绍实用 OSG 的函数及其运用，因此，本书的适宜阅读群体也就限制为特定的人群。

本书专为准备开始学习 OSG，并使用 OSG 进行程序开发的开发者所编写。本书所涉及的工具可能包括特定行业的应用软件，本书同时还提供有关虚拟现实和仿真方面的介绍，这也正是 OSG 的强大之处所在。

OSG 是一个 C++ API 库，因此本书假定读者具备相当的 C++开发经验。特别地，本书的读者应当对于 C++的设计特性较为熟悉，例如公有和私有成员，虚函数，内存分配，类继承，以及构造和析构函数等。OSG 对于标准模板库（STL）的运用十分广泛，因此读者应当对 STL 容器，特别是列表（list），向量组（vector）

和映射（map），有较深的了解。如果读者对于设计模式（design patterns）也有一定的认识，这对于 OSG 的学习大有裨益，不过并不是必要的。

读者应当熟练掌握和运用数据结构的相关知识，例如树结构和链表。

在自己的程序中使用 OSG 之前，读者首先要对 3D 图形学有一定的了解。对于本书而言，读者需要对标准的跨平台底层 3D 图形库 OpenGL 较为熟悉。读者需要理解不同坐标空间的概念，并熟练应用笛卡尔三维坐标系来指定几何数据。读者还需要了解纹理贴图的本质，即向几何体指定图形，不过并不需要对底层图形硬件的实现有很深入的认识。

读者最好还具备一定的线性代数知识。了解用向量表达 3D 位置的方法，以及渲染中图形系统按矩阵变换向量的过程。读者需要了解矩阵连乘来表现几何变换的有关知识。

阅读建议

如果你对上述的方方面面有某些疑惑和不理解的话，阅读下面列出的书籍应当对你的学习大有帮助。

- 《OpenGL® Programming Guide》，第五版，由 OpenGL ARB，Dave Shreiner，Mason Woo，Jackie Neider 和 Tom Davis 合著（Addison-Wesley）[ARB05]。
- 《Geometric Tools for Computer Graphics》，Philip Schneider，David H. Eberly 著（Morgan Kaufmann）。
- 《Real-Time Rendering》，第二版，Tomas Akenine-Moller，Eric Haines 著（AK Peters）。
- 《Computer Graphics》，Principles and Practice 出版，第二版，James D. Foley，Andries van Dam，Steven K. Feiner 及 John F. Hughes 著（Addison-Wesley）。
- 《The C++ Programming Language》，第三版，Bjarne Stroustrup 著（Addison-Wesley）。

本书的组织

《OpenSceneGraph 快速入门指导》由三个主要章节和附录组成。第一章“场景图形与 OpenSceneGraph 概述”，将简要地介绍 OSG 组织的历史，获取和安装 OSG 的方法，以及如何使用 OSG 发行版本中的示例和工具程序。本章的内容还包括对于场景图形概念的介绍，以及 OSG 及其开发团队的介绍。

在第二章“建立一个场景图形”中，你将学习如何使用 OSG 的数据结构来保存和渲染几何体。本章还将介绍一些核心的 OSG 板块，例如引用指针，场景图形节点，几何体类，渲染状态（纹理贴图及光照）等。本章还包括了对 osgText 场景图形工具，向场景快速添加文字的方法，以及通过文件 I/O 保存场景图形数据和图像的方法的介绍。通过这一章的学习，你将熟练掌握使用 OSG 建立场景图形并显示各种几何体的技术。

在第三章“在用户程序中使用 OpenSceneGraph”中，你将学习渲染，视口的放置和确立，以及动态修改场景图形和动画化的方法。

最后，在附录“从这里开始”中，你将了解获取更多关于 OSG 的信息的方法。欢迎你加入 OSG 的开发团队。

关于作者

Paul Martz 是 Skew Matrix Software LLC 的现任主席。该集团主要承担软件定制开发，文档化，以及开发培训的服务。Paul 自从 1987 年以来就一直参与 3D 图形软件的开发工作，并编写了 OpenGL® Distilled 一书[Martz06]。他同时还是打击乐和音乐教育的爱好者，并喜爱一种扑克牌游戏。

致谢

如果没有 OpenSceneGraph，就不会有本书的问世。因此我们将首先感谢 Don Burns 和 Robert Osfield，OSG 的创始人和开发的领导者。同时还要感谢所有的 OSG 开发团队，超过 1600 名优秀的开发者，正在为这个开源软件的标准化时刻贡献着自己的力量。

感谢 Robert Osfield 以及他的 OpenSceneGraph 专业支持团队。我们的第一本 OpenSceneGraph 书籍能够成为一本免费的快速指南，正是出于他的建议。同时感谢 CGSD 的 Roy Latham 以及安第斯（Andes）计算机公司的 Don Burns，他们为本书的第一版提供了资金上的援助。

再次感谢 Robert Osfield，以及 Leandro Motta Barros，他们共同编写了 OSG 的部分开发文档。这无疑成为了本书编写的基石。感谢 Ben Discoe 和虚拟地形开发计划（Virtual Terrain Project）。本书的源代码中使用了他们的数据库中提供的植物图形。

OSG 开发团队中的许多开发者为本书提供了技术支持，以及其它的一些帮助。感谢 Sohaib Athar，Ellery Chan，Edgar Ellis，Andreas Goebel，Chris “Xenon” Hanson，Farshid Lashkari，Gordon Tomlinson，以及 John

Wojnaroski。他们都为本书的出版作出了不同的贡献。

最后，我要感谢 Deedre Martz，她对本书进行了专业的审校工作，并帮助我改正了很多不好的个人写作习惯。

1、场景图形与 OpenSceneGraph 概述

最初的这一章将为你介绍场景图形的概念。你将了解到 OSG 的历史和组织，了解它的能力，并学习如何获取和安装 OSG，以及一些简单例子的运行。通过对这一章的学习，你将逐渐熟悉场景图形和 OSG。本章将不会介绍 OSG 程序的编程细节，你可以在第二章和第三章的学习中获得更多的知识。本章仅仅是一个概述而已。

1.1 OpenSceneGraph 的历史

早在 1997 年，Don Burns 便作为软件设计顾问受雇于 Silicon Graphics（SGI），他在业余时间还喜好于滑翔运动。正因为对计算机图形和滑翔机同样的热衷，以及对尖端渲染设备的了解，他使用 Performer 场景图形（SGI 专有）系统，设计了一套基于 SGI Onyx 的滑翔仿真软件。

由于受到其他滑翔爱好者的鼓励，Don 开始尝试使用 Linux 上的 Mesa3D 和 3dfx 的 Voodoo 设备，以开发基于更多硬件平台的仿真软件。当这套软件开始支持 OpenGL 的时候，场景图形的概念还未能应用于 Linux。为了填补这一空缺，Don 开始编写一套简单的类似于 Performer 的场景图形系统，名为“SG”。SG 的开发强调朴素且易用，它满足了当时人们对于场景图形系统的需求，也使得 Don 的滑翔仿真软件能够运行于低成本的 Linux 系统。

到了 1998 年，Don 在滑翔爱好者的邮件组中遇到了 Robert Osfield。那时 Robert 在 Midland Valley Exploration 工作，那是一个来自苏格兰格拉斯哥的油气公司。Robert 同样对计算机图形学和可视化技术有着浓厚的兴趣。两人开始合作对仿真软件进行改善。Robert 倡导开源，并提议将 SG 作为独立的开源场景图形

项目继续开发，并由自己担任项目主导。项目的名称改为 OpenSceneGraph，当时共有九人加入了 OSG 的用户邮件列表。

2000 年底，Brede Johansen 对 OpenSceneGraph 做出了第一份贡献——添加了 OSG 的 OpenFlight 模块。当时他在挪威孔斯贝格的 Kongsberg Maritime 船舶仿真公司工作。该公司后来设计了基于 OSG 的 SeaView R5 视觉系统。

同样在 2000 年，Robert 离开了原来的工作单位，作为 OpenSceneGraph 的专业服务商开始全职进行 OSG 的开发工作。那段时期，他设计并实现了今天 OSG 所使用的许多核心功能，并且是在完全没有客户和薪酬的情况下。

Don 到了 Keyhole 数字地图公司（现在是 Google 的 Google Earth 部门），随后于 2001 年辞职。他也组建了自己的公司——Andes ComputerEngineering，位于加利福尼亚州的圣何塞。公司成立后继续进行 OSG 的开发工作。第一届 OpenSceneGraph“同好”会在 SIGGRAPH 2001 举行，只有 12 人参加。听众中包括了来自 Magic Earth 的代表，其目的是寻求一个开源的场景图形库来支持油气相关软件的开发。他们和 Don 和 Robert 讨论了开发的事宜，并成为了 OSG 的第一个收费用户。

每一年，OSG 同好会的参与者都会增加，OSG 用户邮件列表的成员也以明显的速度飞快增长（如图 1-1）。在本书付印之时，OSG 已经拥有超过 1600 名用户了。

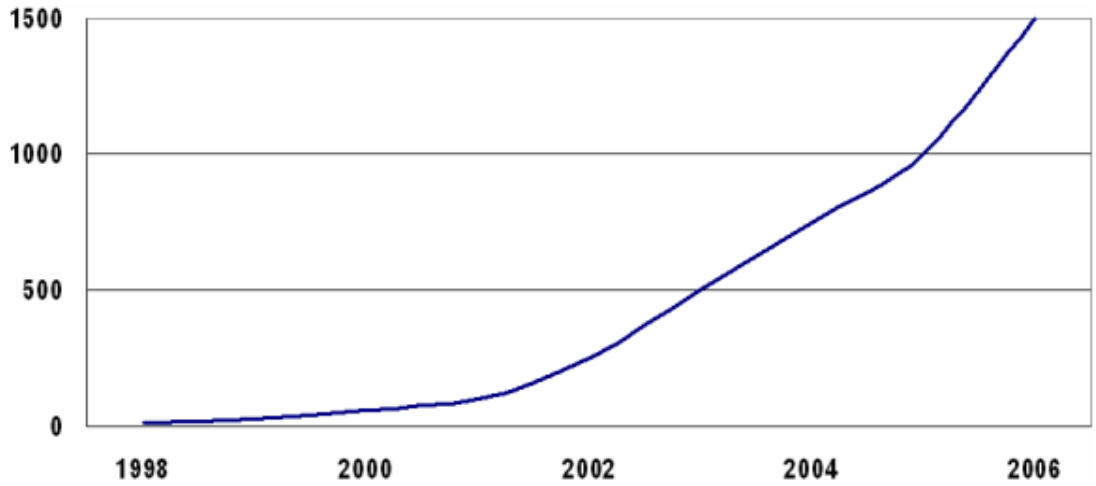


图 1-1 OSG 用户邮件组的增加

OSG 用户邮件组的增加速度令人瞩目。

OSG 的模块和附加库的开发也在不断加速。2003 年，OSG 的通用库 Producer

（原名为 OSGMP）诞生，它为 Magic Earth 提供了多通道渲染的能力。2004 年，又增加了大型数据库分页，地形系统，以及阴影的支持。2006 年，OpenFlight 插件得到了全面的完善，同时还增加了 osgViewer 库，这是一个用于管理和渲染场景视图的集成库。

如今，一部分高性能的软件已经使用了 OSG 来渲染复杂的 2D 和 3D 场景。虽然大部分基于 OSG 的软件更适用于可视化设计和工业仿真，但是在使用 3D 图形的每个领域，都已经出现了 OSG 的身影。这其中包括了地理信息系统（GIS），计算机辅助设计（CAD），建模和数字内容创作（DCC），数据库开发，虚拟现实，动画，游戏和娱乐业。

1.2 OSG 的安装

上一节已经介绍了 OSG 的起源和历史。本节将介绍如何获得和安装 OSG，运行 OSG 提供的例子，并开发自己的 OSG 程序。OSG 维基网站 OSGWiki 已经提供了各种格式的文件包和机制，提供 OSG 的下载。

运行时文件：使用 OSG 运行时文件包，安装必要的链接库文件，以运行 OSG 示例和程序。

OSG 源代码：OSG 的开发者应当获取一份 OSG 的源代码。OSG 提供了多种获取完整源代码的方式。你可以下载一份 OSG 稳定版本的压缩文档，下载每日更新的 tar 文件包，或者使用版本控制系统（SVN）获取最新的源代码。

OpenThreads：OSG 的核心库依赖于 OpenThreads，以获得多线程的支持。你必须建立 OpenThreads 的开发环境以编译 OSG 的源代码和基于 OSG 的应用程序。

第三方的支持：从源代码编译 OSG 时，某些可选的组件可能需要其他软件包的支持，例如 libTIFF，libPNG 等。如果你的系统还没有安装诸如此类的第三方软件和库，这些可选组件的编译可能会失败。

示例数据：包括一些 2D 图形，3D 模型和其他数据文件。

下面的章节将介绍获取和安装 OSG 运行时文件的方法。虽然 OSG 可以在各种平台上运行，但是这里只涉及 Apple Mac OS X，Fedora Linux 和 Microsoft Windows 平台的部分。关于其它平台上 OSG 的获取方法，请登陆 OSG 维基网站 OSGWiki。

如果安装用的可执行文件无法在你的系统平台上使用，或者你打算建立自己的 OSG 开发环境，你可以选择编译 OSG 的源代码。关于 OSG 源代码，第三方支持软件，OpenThreads 和其他示例数据的获取方法，请登陆 OSG 维基网站 OSGWiki。

1.2.1 硬件需求

如今的 OSG 已经可以在多种硬件平台和操作系统上运行，并且能够在大部分计算机系统上正常使用。

处理器：OSG 可以在大部分的 CPU 上编译通过。OSG 具备线程安全性，并且可以有效利用多处理器和双核结构的特性。OSG 可以在 32 位或者 64 位处理器上运行通过。

图形：你的计算机系统需要配置一块 AGP 或者 PCI 总线的图形显示卡。OSG 可以在大部分用于建模，仿真和游戏的专业级或大众级图形设备上运行。可以运行 OSG 的图形设备必须高效地支持 OpenGL，因此你应当从设备商处获得最新的 OpenGL 设备驱动程序。OSG 对显卡 RAM 的需求因用户的使用而异，但是 256MB 应当足够了。OSG 可以在多管（multi-pipe）显示系统上运行，并且可以利用多显卡来提升渲染速度。

RAM：最小的系统 RAM 内存需求是由显示数据的数量和类型决定的。推荐配置为 1GB，大型数据集的开发可能需要更多的内存支持。

磁盘：和 RAM 一样，磁盘空间的需求大小由数据量决定。对于任何程序来说，更高速和更大容量的磁盘无疑可以减少数据读取的时间。

1.2.2 Apple Mac OS X

应用于苹果 Mac OS X 系统的 OSG 是以.dmg 的格式存储的，其中包括了运行时文件和完整的开发环境，可以从 OSGWiki 上获得。

安装 OSG 的步骤如下：

登陆 OSG 维基网站 OSGWiki，选择“Downloads”；

下载“OSG Universal Binaries for OSG”，这是一个.dmg 文件；

下载结束之后，加载该.dmg 文件；

将“.dmg Frameworks folder”的内容拖动到“/System/Frameworks”；

在“/Library/Application”中，建立名为 OpenSceneGraph 的文件夹；

将.dmg 文件中的各种插件拖动至新的文件夹中。

1.2.3 Fedora Linux

OSG 可以在多种版本的 Linux 系统上运行。大部分的 Linux 环境均提供软件包安装界面，搜索并安装新的软件。例如，在 Ubuntu Linux 中，运行“Synaptic Package Installer”并搜索 OpenSceneGraph，找到并选择 OSG 运行时文件和开发环境的软件包，安装即可。

应用于 Fedora Core 4 的最新版本的 OSG 文件，可以登陆 OSG 维基网站 OSGWiki 下载。选择“Downloads”，在“Binaries”下选择“Fedora Core 4”的链接。

1.2.4 Microsoft Windows

应用于 Microsoft Windows 的 OSG 运行时文件可以从 OSG 维基网站 OSGWiki 下载，它采用 InstallShield 安装包的形式。其安装步骤如下：

登陆 OSG 维基网站 OSGWiki，选择“Downloads”；

下载“OSG Win32 Binaries”，它是一个.exe 文件；

下载完成后，双击.exe 文件并按照提示执行安装。

缺省的安装方式将修改注册表中的系统环境变量。要使这些修改生效，需要注销用户或者重新启动系统。

1.2.5 检查 OSG 的安装

OSG 安装完成之后，需要检查安装的正确性。其步骤如下：

打开系统的命令行窗口；

输入命令，

osgversion

此命令执行了 osgversion 程序，输出为 OSG 的版本号，如下所示，

OpenSceneGraph Library 2.0

这一步确认系统已经找到 OSG 的可执行文件（系统 PATH 变量已经正确设置），并输出当前所用 OSG 的版本，同时保证 OSG 基本可以使用。

要检查 OSG 在本地系统上的渲染能力，执行以下的命令，

osglogo

输出的结果类似图 1-2。

osglogo 通过动态更新场景，旋转地球标志。它也支持鼠标接口，用户可以通过鼠标左键旋转并观察图形标志。

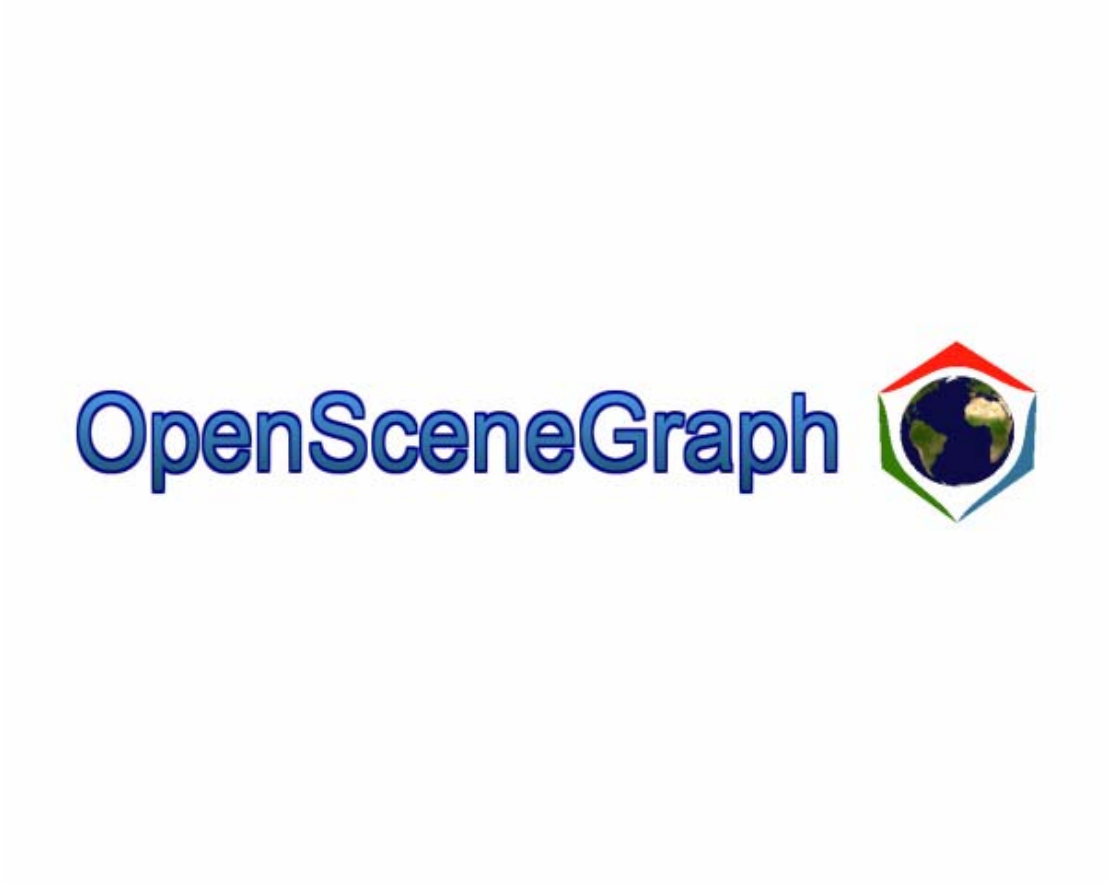


图 1-2 OSG 的标志
这是执行 osglogo 程序后显示的结果。

1.3 运行 osgviewer

在上一节，你已经尝试运行过 osgversion 和 osglogo。它们可以用于检验 OSG 的安装是否正确，但是其本身依然功能有限。本节将介绍 osgviewer 的运行，它是 OSG 的一个强大且灵活的模型浏览工具。下面的命令将读取一个简单的奶牛模型并且加以显示：

osgviewer cow.osg

运行结果如图 1-3 所示。

图 1-3 osgviewer 的输出

本图显示了命令 osgviewer cow.osg 的运行结果。程序 osgviewer 可以显示多种图片和模型格式的文件。

1.3.1 获取帮助

在运行 osgviewer 程序时，按下“H”键（小写 h）可以显示按键及其对应功能的帮助列表。按键“1”到“5”可以切换不同的摄像机控制模式，即，鼠标运动对摄像机位置的操纵方式。当前的摄像机选择的是“1”键对应的轨迹球模式，这也是程序的缺省模式。使用键盘命令控制显示模式的更多方法将在下一节阐述。

按下 Esc 键可以退出 osgviewer 程序，这在帮助列表中也有说明。现在按下 Esc，osgviewer 将退出并返回到命令行提示符下。

输入如下的命令，可以看到详细的 osgviewer 命令行选项列表：

osgviewer –help

osgviewer 将显示所有可用的命令行选项。以下对其中部分常用的选项作了介绍。

- --clear-color：此选项允许用户设置清屏颜色，也就是背景颜色。例如，输入如下的命令，osgviewer 将使用白色作为背景：

osgviewer --clear-color 1.0,1.0,1.0,1.0 cow.osg

- --samples：此选项将启用硬件多重采样（hardware multisampling），也就是我们常说的“全屏幕抗失真”（full screen antialiasing）。其中包括一个数字型的参数，代表每个像素的采样数值。例如，输入如下的命令可以启用 16 子像素的多重采样：

osgviewer --samples 16 cow.osg

- --image：此选项将使 osgviewer 读取一幅单独的图片，并将其作为一个四边形几何体的材质显示。输入如下的命令：

osgviewer --image osg256.png

除了上述的命令行参数和键盘命令以外，你还可以使用一些环境变量来控制 osgviewer。要阅读详细的 osgviewer 帮助文档，在命令行方式下输入以下命令：

osgviewer --help-all

下面的章节将对 osgviewer 的应用作进一步介绍。

1.3.2 显示模式

osgviewer 中有相当一部分按键用于指定不同的显示模式，以控制读入模型的外观。以下列出了部分常用的控制命令。

- 多边形模式（Polygon mode）：反复按下“W”键（小写 w）可以在线框模式，点模式和填充多边形渲染模式之间切换。

- 贴图映射（Texture mapping）：按下“T”键（小写 t）可以切换显示或不显示模型的贴图。
- 光照（Lighting）：按下“L”（小写 l）键决定禁止或者允许光照。
- 背面拣除（Backface culling）：按下“B”（小写 b）键触发或禁止背面拣除。对于 cow,osg 中的模型而言，此选项不会改变其外观，但是它可能影响其它一些模型的外观和渲染性能。
- 全屏幕模式（Fullscreen mode）：按下“F”键（小写 f）切换全屏幕渲染和窗口渲染。

你可以花费一些时间来试验各种显示模式的组合效果。例如，要清楚地观察到一个模型的多边形结构，可以使用线框模式，同时禁止模型贴图和光照的显示。

1.3.3 环境变量

OSG 以及 osgviewer 程序提供了非常多的环境变量，不过你需要尽快了解和熟悉其中的两个。在使用 OSG 进行设计时，你会经常用到这两个环境变量。

文件搜索路径

环境变量 OSG_FILE_PATH 指定了 OSG 程序读取图形和模型文件时的搜索路径。如果你运行命令 osgviewer cow.osg，但是当前目录中并没有 cow.osg，OSG 将在 OSG_FILE_PATH 指定的路径中查找并读取这个文件。

安装 OSG 的运行时文件时，将自动设置 OSG_FILE_PATH 变量。你可以向这个变量中添加更多的目录。在 Windows 系统下，使用分号分隔各个目录，在其它系统下则使用冒号。如果环境变量为空或者没有设置，OSG 在读取图形和模型文件时只搜索当前的目录。

调试信息显示

OSG 可以将各式各样的调试信息输出到 std:cout。这在开发 OSG 程序时十

分有用，你可以借此观察 OSG 的执行的各种操作。环境变量 OSG_NOTIFY_LEVEL 用于控制 OSG 调试信息显示的数量。你可以将此变量设置为七个不同的信息量层级之一：ALWAYS（最简略），FATAL，WARN，NOTICE，INFO，DEBUG_INFO 以及 DEBUG_FP（最详细）。

一个典型的 OSG 开发环境可以设置 OSG_NOTIFY_LEVEL 为 NOTICE，如果要获取更多或者更少的输出信息，可以根据信息量的详细程度上下调整此变量。

1.3.4 统计信息显示

键盘的“S”键对于性能测试十分有用，它将适用 osgUtil 库的 Statistics 类来收集和显示渲染性能的信息。循环按下“S”键（小写 s）可以切换四种不同的显示方式：

1、帧速率：osgviewer 将显示每秒钟渲染的帧数（FPS）。

2、遍历时间：osgviewer 将显示每一次遍历更新（update），拣选（cull）和绘制（draw）操作总共使用的时间，包括绘制图 1-4 所示的图表所用的时间。

3、几何信息：osgviewer 将显示当前渲染的 osg::Drawable 对象数目，以及每一帧处理的所有顶点和几何体的数目。

4、无：osgviewer 将关闭统计信息的显示。

按下“S”键两次，程序将显示遍历时间的统计信息及图表信息，如图 1-4 所示。

图形的显示表现为一系列帧的渲染。特别地，渲染过程与显示器的刷新速率是同步的，这是为了避免诸如图像撕裂（image tearing）等问题的出现。在图 1-4 中，显示器的刷新率为 60Hz，因此每一帧将占用一秒的 1/60，即大约 16.67 毫秒。上面的显示图表说明了场景更新（update），拣选（cull）和绘制（draw）过程所花费的时间。这种反馈方式从本质上提供了分析性能问题并帮助解决程序渲染性能瓶颈的途径。

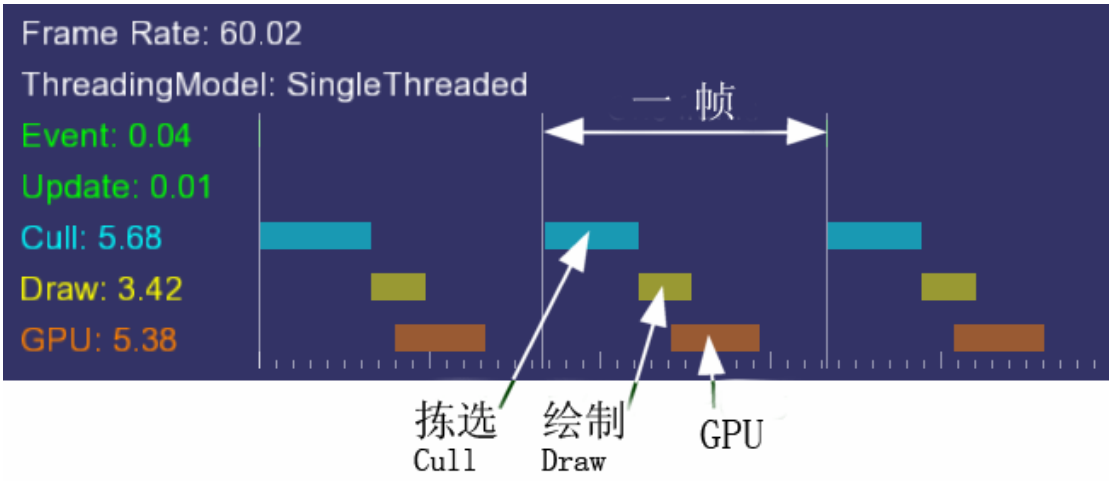


图 1-4 遍历时间的图表显示

上图显示了一个典型的遍历时间统计信息，程序的运行使用 60HZ 的显示器。更新（update）操作所用的时间为 0.04 毫秒，因为过于短暂，此结果在图表中没有显示出来。不过，拣选(cull)和绘制（draw）所用的时间分别为 2.68 毫秒和 3.05 毫秒，在图表中它们分别用青色和暗黄色的图块显示在一帧的范围里。图中输出的文字还表明，OpenGL 测量的结果，GPU 处理渲染指令所用的时间为 1.87 毫秒。

1.3.5 记录动画

开发者往往需要反复地进行测试以便有效地调整和检测程序渲染性能。为了简化性能调整的过程，osgviewer 允许用户轻松地记录摄像机的运动序列并且进行回放。这一序列被叫做动画路径（animation path）。

运行 osgviewer 时，按下“Z”键（小写 z）将立即开始记录动画路径。此时如果使用鼠标旋转和缩放模型，OSG 都将记录摄像机运动的信息。然后，按下“Shift + Z”键（大写 Z）将停止动画路径的记录，并立即开始回放。回放的过程中，你将会看到所有被记录下来的摄像机运动路径。

使用 Esc 键退出 osgviewer，在当前目录下可以得到一些新的文件。其中有一个名为 saved_animation.path 的文件，正如其名称所示，包含了记录下来的动画路径。用户按下“Z”键（小写 z）时，osgviewer 将信息写入此文件。你可以

使用如下的命令回放此动画路径的内容。

osgviewer –p saved_animation.path cow.osg

回放动画路径时，osgviewer 将序列所消耗的时间输出到 std::out。如果 osgviewer 并没有显示这些信息，按下 Esc 退出 osgviewer，设置环境变量 OSG_NOTIFY_LEVEL 为 INFO，并重新启动 osgviewer。

1.4 编译 OSG 程序

要生成基于 OSG 的应用程序，你需要建立一个包括头文件和链接库文件的 OSG 开发环境。OSG 运行时文件中包括了头文件和优化的链接库文件。如果想创建调试用的链接库，你需要下载并重新编译 OSG 和 OpenThreads 的源代码。OSG 和 OpenThreads 的源代码均可以在 OSG 维基网站的下载专区上取得。OSG 维基网站还包括了如何生成 OSG 的教学文档。

要正确编译基于 OSG 的程序，首先要设置正确的头文件路径，以便编译器找到必要的头文件。在编译器的包含文件搜索路径中添加以下的目录：

<parent>/OpenSceneGraph/include

<parent>/OpenThreads/include

将<parent>替换为 OpenSceneGraph 和 OpenThreads 的最高层安装目录。使用 Linux 系统的 gcc 进行编译时，可以添加-I 参数。在 Linux 系统上，<parent>通常会被替换为/usr/local/include，因此 gcc 的命令行参数应当包含如下的格式：

-I/use/local/include/OpenSceneGraph/include

-I/usr/local/include/OpenThreads/include

如果使用 Microsoft Visual Studio，可以打开 Project Properties 对话框的 C/C++选项卡，在 Additional Include Directories 中添加适当的路径。

同样的，你也需要向链接器说明 OSG 链接库的位置。在 Linux 操作系统下，OSG 链接库通常位于/usr/local/lib，因此链接程序不需要再使用-L 参数就可以找

到它们。

Microsoft Visual Studio 所需的链接库文件可以在如下的源代码目录中找到：

<parent>/OpenSceneGraph/lib/win32

<parent>/OpenThreads/lib/win32

将这两个目录添加到 Project Properties 对话框的 Linker 选项卡，Additional Library Directories 选项。

最后，选择应用程序将要链接的 OSG 链接库。正如 1.6.3 节“组件”中叙述的，OSG 是由多个不同的库组成的，每个库都提供了不同的功能模块。一个简单的基于 OSG 的程序往往需要使用 osgViewer，osgDB，osgUtil 和 osg 库，其 gcc 命令参数如下：

-losgViewer –losgDB –losgUtil –losg

使用 Microsoft Visual Studio 进行编译时，将库文件的名称添加到 Project Properties 对话框的 Linker 选项卡，Additional Dependencies 选项。在 Microsoft Windows 系统下，OSG 针对 Debug 和 Release 版本生成不同名字的链接库。对于 Release 版本的编译，添加如下的链接库名称：

osgViewer.lib osgDB.lib osgUtil.lib osg.lib

对于 Debug 版本的编译，在文件扩展名前添加“d”：

osgViewerd.lib osgDBd.lib osgUtild.lib osgd.lib

上面所添加的链接库只是一个例子，程序中实际添加的库取决于程序中用到的 OSG 功能模块。你的程序可能需要链接其它的库，例如 osgText，osgShadow，osgGA 等。在 Mac OS X 系统中，如果已经设定 Xcode 使用 OSG 框架，那么上述所有的工作均可以由操作系统自动进行处理。

如果你错误地设置了编译和链接的选项，你的程序在生成过程中可能会出现诸如“unable to open include file”，“unable to find library file”，“unresolved symbol”的错误。当你遇到这样的错误提示时，请仔细检查错误提示信息，并确定你已经正确设置了程序编译和链接的选项。

1.5 场景图形初步

前几章的内容主要是关于 OSG 的起源，安装和例子程序的运行。如果你一直在按照指导阅读本章，你应该已经使用 OSG 在屏幕上创建了一些有趣的图形了。本书的剩余部分将更深一步地对 OSG 进行探讨：本节从概念的层次介绍场景图形；第 1.6 节，“OpenSceneGraph 概述”，将对 OSG 的特性作更高层次的介绍；然后，第二章“建立场景图形”，以及第三章“在用户程序中使用 OpenSceneGraph”，将对 OSG 应用编程接口部分的内容作进一步阐述。

场景图形采用一种自顶向下的，分层的树状数据结构来组织空间数据集，以提升渲染的效率。图 1-5 描述了一个抽象的场景图形，其中包含了地形，奶牛和卡车的模型。

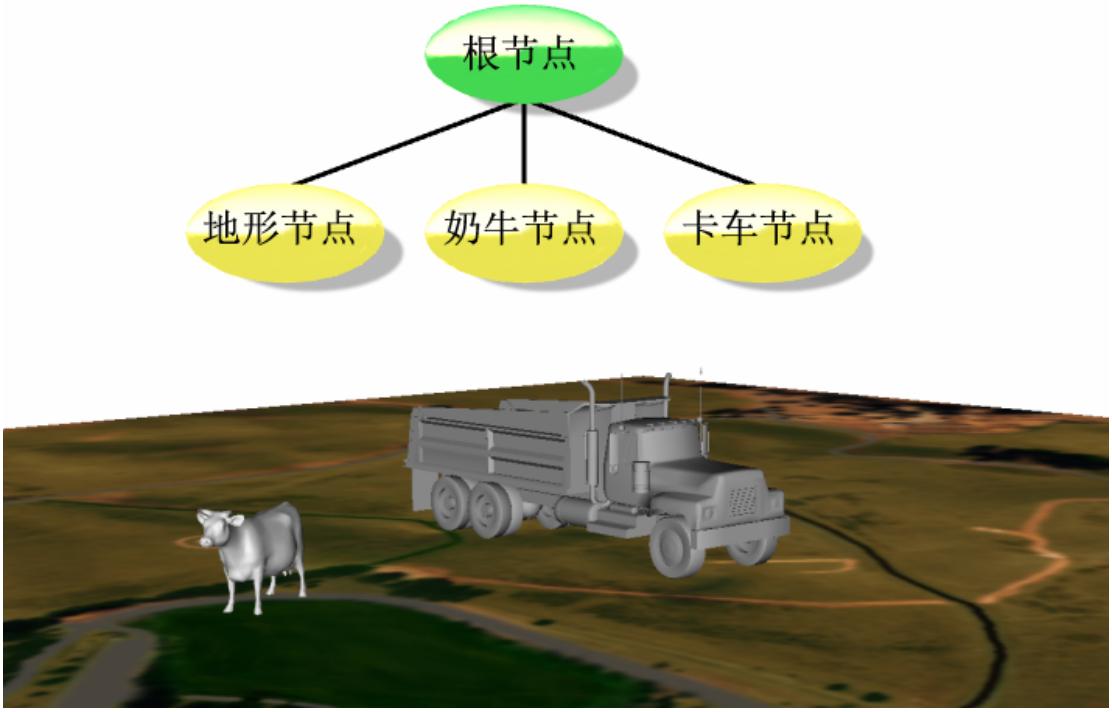


图 1-5 简单、抽象的场景图形

渲染一个包含了地形、奶牛和卡车的场景时，场景图形采用了一个根节点与三个子节点的形式。每个子节点都包含了用于绘制对象的几何信息。

场景图形树结构的顶部是一个根节点。从根节点向下延伸，各个组节点中均包含了几何信息和用于控制其外观的渲染状态信息。根节点和各个组节点都可以

有零个（有零个子成员的组节点事实上没有执行任何操作）或多个子成员。在场景图形的最底部，各个叶节点包含了构成场景中物体的实际几何信息。

OSG 程序使用组节点来组织和排列场景中的几何体。想象这样一个三维数据库：一间房间中摆放了一张桌子和两把一模一样的椅子。你可以采用多种方法来构建它的场景图形。图 1-6 表示一种可能的组织方式。根节点之下有四个分支组节点，分别为房间几何体，桌子几何体，以及两个椅子几何体。椅子的组节点为红色，以标识它们与子节点的转换关系。椅子的叶节点只有一个，因为两个椅子是同样的，其上级组节点将这把椅子转化到两个不同的空间位置以产生两把椅子的外观效果。桌子组节点只有一个子节点，即桌子叶节点。房间的叶节点包括了地板、墙壁和天花板的几何信息。

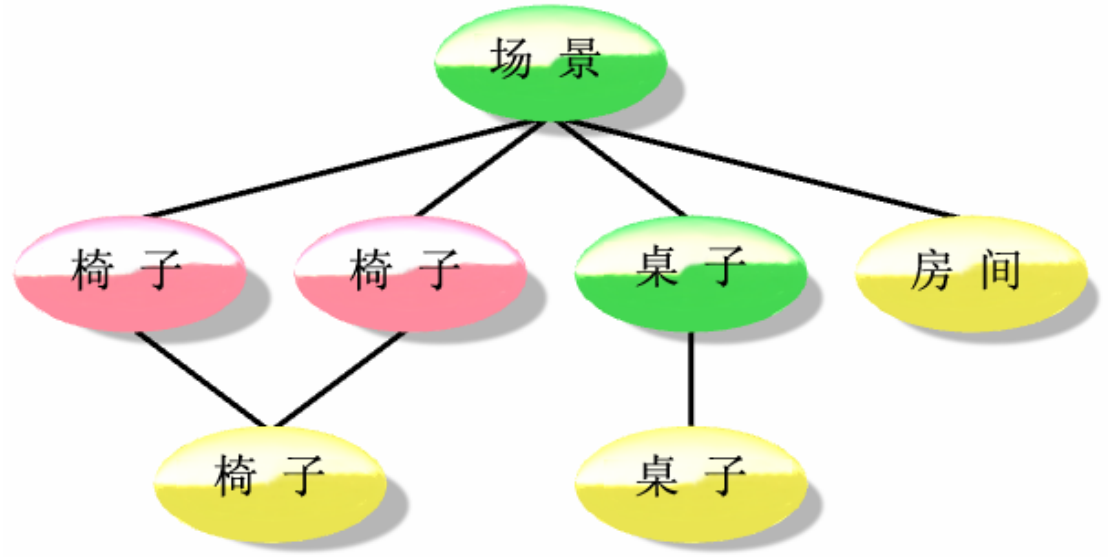


图 1-6 场景图形示例

组节点可能有多个子节点，程序可以有序地安排其几何和状态数据。在此例中，两个椅子组节点将其唯一的子节点变换到两个不同的空间位置上，产生了两把椅子的外观效果。

场景图形通常包括了多种类型的节点以执行各种各样的用户功能，例如，开关节点可以设置其子节点可用或不可用，细节层次（LOD）节点可以根据观察者的距离调用不同的子节点，变换节点可以改变子节点几何体的坐标变换状态。面向对象的场景图形使用继承的机制来提供这种多样性，所有的节点类都有一个共有的基类，同时各自派生出实现特定功能的方法。

大量定义的节点类型及其内含的空间组织结构能力，使得传统的底层渲染API 无法实现的数据存储特性得到了实现。OpenGL 和 Direct3D 主要致力于图形硬件特性的抽象实现。尽管图形设备可以暂时保存即将执行的几何和状态数据（例如显示列表和缓冲对象），但是底层 API 中对于上述数据的空间组织能力在本质上还是显得过于简单和弱小，往往难以适应大部分 3D 程序的开发与应用需求。

场景图形是一种中间件（middleware），这类开发软件构建于底层 API 函数之上，提供了高性能 3D 程序所需的空间数据组织能力及其它特性。图 1-7 表现了一个典型的 OSG 程序层次结构。

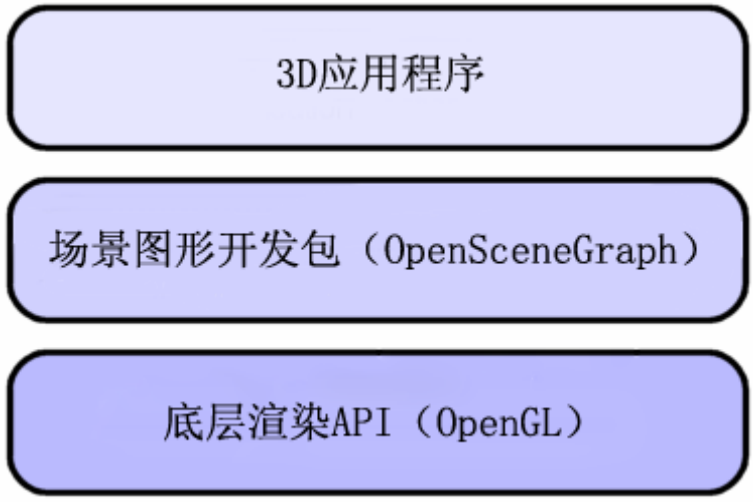


图 1-7 3D 程序层次结构

多数 3D 应用程序并不是直接通过底层渲染 API 显示人机交互界面，而是需要使用 OpenSceneGraph 等开发包中提供的额外的功能。

1.5.1 场景图形特性

场景图形除了提供底层渲染 API 中具备的几何信息和状态管理功能之外，还兼具如下的附加特性和功能：

- 空间结构：场景图形所采用的树状数据结构更直观，也更符合人们一般理解中的空间事物的排布和组织结构。

- 场景拣选：使用本地 CPU 的投影剔除（frustum culling）和隐藏面剔除（occlusion culling）来减少系统总体负担，其基本原理是，在最终渲染图像时忽略对不会显示的几何体的处理。
- 细节层次（LOD）：使用几何体包围盒计算观察者与物体的距离，使得用户可以更高效地渲染处于不同细节层次上的物体。并且，实时的，场景中进入指定观察距离的那部分对象将从磁盘中载入，而它们一旦超出这一距离时，将从内存中被移除。
- 透明：要实现透明或半透明几何体的正确和高效的渲染，需要首先渲染所有不透明的几何体，再渲染透明几何体。而且，透明几何体必须按照深度排序并按照“从后向前”的顺序渲染。场景图形一般都会提供上述这些操作。
- 状态改动最少化：为了最大限度地提升程序性能，应该避免冗余和不必要的状态改变。场景图形会按状态对几何体进行排序以最小化状态改动，OpenSceneGraph 的状态管理工具则负责消除冗余的状态改变。
- 文件 I/O：场景图形可以高效地读写磁盘上的 3D 数据集。在将数据读入内存之后，应用程序可以方便地通过内建的场景图形数据结构操控动态 3D 数据。场景图形也是一个高效的文件格式转换工具。
- 更多高性能函数：除了底层 API 提供的基础函数之外，场景图形库还提供了高效能的功能函数，例如全特性的文字支持，渲染特效的支持（例如粒子特效，阴影），渲染优化，3D 模型文件读写的支持，并支持对跨平台的输入、渲染及显示设备的访问。

几乎所有的 3D 程序都需要其中的部分特性。因此，直接使用底层 API 来构建程序的开发者不得不自行在程序中实现其中的某些性能，从而增加了研发的时间、人力及资金投入。使用现有的、支持上述特性的场景图形库，将有助于实现快速的程序开发工作。

1.5.2 场景图形渲染方式

一个的场景图形系统允许程序保存几何体并执行绘图遍历，此时所有保存于场景图形中的几何体以 OpenGL 指令的形式发送到硬件设备上。但是该执行机制无法实现前述的诸多高级特性。为了实现动态的几何体更新，拣选，排序和高效渲染，场景图形需要提供的不仅仅是简单的绘图遍历：事实上，有三种需要遍历的操作：

- 更新（update）：更新遍历（有时也称作程序遍历）允许程序修改场景图形，以实现动态场景。更新操作由程序或者场景图形中节点对应的回调函数完成。例如，在飞行模拟系统中，程序可以使用更新遍历来改变一个飞行器的位置，或者通过输入设备来实现与用户的交互。
- 拣选：在拣选遍历中，场景图形库检查场景里所有节点的包围体。如果一个叶节点在视口内，场景图形库将在最终的渲染列表中添加该节点的一个引用。此列表按照不透明体与透明体的方式排序，透明体还要按照深度再次排序。
- 绘制：在绘制遍历中（有时也称作渲染遍历），场景图形将遍历由拣选遍历过程生成的几何体列表，并调用底层 API，实现几何体的渲染。

图 1-8 表示上述三类遍历。

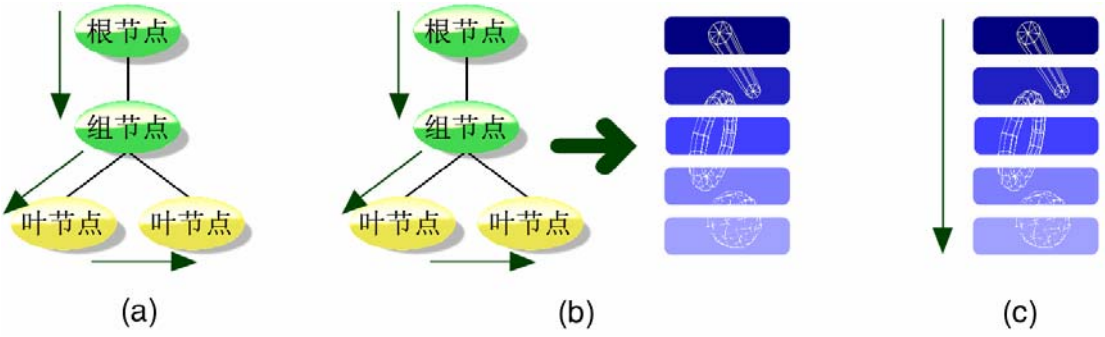


图 1-8 场景图形遍历

渲染一个场景图形需要三个遍历过程。(a) 更新遍历，修改几何体、渲染状态，或者节点参数，保证场景图形的更新对应当前帧；(b) 拣选遍历，检查可见性，将几何体和状态量置入新的结构（在 OSG 中称为渲染图形，render graph）之中。(c) 绘制遍历，遍历渲染图形并向图形硬件设备发出绘制指令。

特别地，这三种遍历操作在每一个渲染帧中只会执行一次。但是，有些渲染特例需要将同一个场景（不同或相同部分）在多个视口中进行同步显示，例如立体化渲染和多屏显系统的实现。在这种情况下，每一帧的更新遍历将只会执行一次，但拣选和绘制遍历在每个视口内均执行一次（对于简单的立体化渲染，每帧执行两次；对于多屏显系统，每块显卡每帧执行一次）。这样多处理器和多显卡的系统就可以实现并行场景图形处理。其中，拣选遍历必须为只读操作以允许多线程下的数据访问。

1.6 OpenSceneGraph 概览

OSG 包含了一系列的开源图形库，主要为图形图像应用程序的开发提供场景管理和图形渲染优化的功能。它使用可移植的 ANSI C++编写，并使用已成为工业标准的 OpenGL 底层渲染 API。因此，OSG 具备跨平台性，可以运行在 Windows，Mac OS X 和大多数类型的 UNIX 和 Linux 操作系统上。大部分的 OSG 操作可以独立于本地视窗系统。但是 OSG 也包含了针对某些视窗系统特有功能的支持代码，例如 PBuffers。

OSG 是公开源代码的，它的用户许可方式为修改过的 GNU 宽通用公共许可证（GNU Lesser General Public License，LGPL）。OSG 采用开源形式的共享方案具备了诸多益处：

- 提高品质：OSG 由 OSG community 的诸多成员反复进行检查、测试和改善。直接参与 OSG 1.2 的开发人员已经超过了 200 人。
- 提高程序质量：要编写高质量的程序，开发者需要十分了解自己所用的开发包。如果这个开发包不开放源代码，与它相关的开发信息就被封闭起来，用户只能借助开发商的文档和客户支持来获得开发信息。开放源代码使得程序员可以检查和调试所用开发包的源代码，充分了解代码内部信息。
- 减少费用：开源意味着免费，除了一开始购买软件所需的费用。

- 没有知识产权问题：对于开源且易于所有人阅读的代码而言，不存在侵犯软件专利的可能性。

1.6.1 设计和体系

OSG 的设计兼顾系统的可移植性和可扩展性。因此，OSG 适用于多种硬件平台，并可在多种不同的图形硬件上进行高效的、实时的渲染。OSG 具备灵活、可扩展的系统特性，使其能自适应不同时期的设计和应用需求。综上所述，OSG 已具备了满足各类客户的、不断增长的需求的能力。

为达到以上设计目标~~理念~~，OSG 采用了以下设计理念和工具进行系统的设计和构建：

- ANSI 标准 C++；
- C++标准模板库（STL）；
- 设计模式（Design patterns，Gamma95）。

这些工具使得程序员可以在自己喜好的平台上使用 OSG 进行开发，并~~且~~依据用户指定的平台对 OSG 进行配置。

1.6.2 命名习惯

以下列举了 OSG 源代码中的一些命名习惯。这些命名习惯可能不会被严格遵守，例如许多由第三方开发的 OSG 插件中就有违反命名习惯的情况。

- 命名空间：OSG 的域名空间使用小写字母开头，然后可以使用大写字母以避免混淆。例如，osg，osgSim，osgFX 等。
- 类：OSG 的类名以大写字母开头，如果类的名称是多个单词的组合，此后每个单词的首字母大写。例如，MatrixTransform，NodeVisitor，Optimizer。
- 类方法：OSG 类的方法名使用小写字母开头，如果方法的名称是多个单词的组合，此后每个单词的首字母大写。例如，addDrawable()，

getNumChildren()，setAttributeAndModes()。

- 类成员变量：类的成员变量命名与方法命名的方式相同。
- 模板：OSG 模板的命名用小写字母，多个单词之间使用下划线分隔。例如，ref_ptr<>，graph_array<>，observer_ptr<>。
- 静态量：静态变量和函数的名称使用 s_开头，此后的命名与类成员变量及函数的命名方法相同。例如，s_applicationUsage，s_ArrayNames()。
- 全局量：全局类的实例命名用 g_开头。例如，g_NotifyLevel，g_readerWriter_BMP_Proxy。

1.6.3 组件

OSG 运行时文件由一系列动态链接库（或共享对象）和可执行文件组成。这些链接库可分为以下五大类：

- OSG 核心库。它提供了基本的场景图形和渲染功能，以及 3D 图形程序所需的某些特定功能实现。
- NodeKits。它扩展了核心 OSG 场景图形节点类的功能，以提供高级节点类型和渲染特效。
- OSG 插件。其中包括了 2D 图像和 3D 模型文件的读写功能库。
- 互操作库。它使得 OSG 易于与其它开发环境集成，例如脚本语言 Python 和 Lua。
- 不断扩展中的程序和示例集。它提供了实用的功能函数和正确使用 OSG 的例子。

图 1-9 表示了 OSG 的体系结构图。下面的章节将更进一步地讨论 OSG 的各个功能模块。

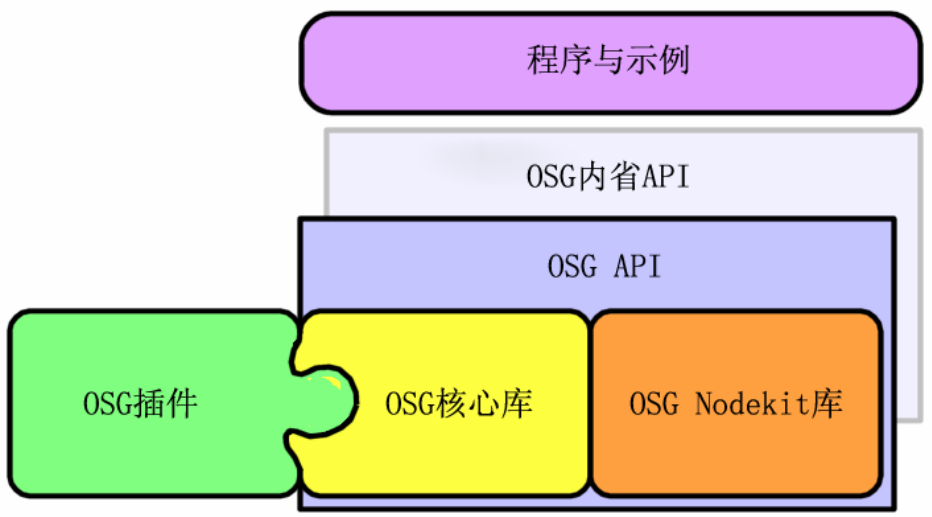


图 1-9 OSG 体系结构

OSG 核心库提供了应用程序和 NodeKits 所需的功能模块。而 OSG 核心库和 NodeKits 一同组成了 OSG 的 API。OSG 核心库中的 osgDB 则通过对 OSG 插件的管理，为用户提供了 2D 和 3D 文件 I/O 的接口。

OSG 核心库

OSG 核心库提供了用于场景图形操作的核心场景图形功能、类和方法；开发 3D 图形程序所需的某些特定功能函数和编程接口；以及 2D 和 3D 文件 I/O 的 OSG 插件入口。OSG 核心库包含了以下四个链接库：

- osg 库：osg 库包含了用于构建场景图形的场景图形节点类，用作向量和矩阵运算的类，几何体类，以及用于描述和管理渲染状态的类。osg 库中还包括 3D 图形程序所需的典型功能类，例如命令行参数解析，动画路径管理，以及错误和警告信息类。
- osgUtil 库：osg 工具库包括的类和函数，可以用于场景图形及其内容的操作，场景图形数据统计和优化，以及渲染器的创建。它还包括了几何操作的类，例如 Delaunay 三角面片化（Delaunay triangulation），三角面片条带化（triangle stripification），纹理坐标生成等。
- osgDB 库：此链接库包括了建立和渲染 3D 数据库的类与函数。其中包括用于 2D 和 3D 文件读写的 OSG 插件类的注册表，以及用于访问这些

插件的特定功能类。osgDB 数据库分页机（database pager）可以支持大型数据段的动态读入和卸载。

- osgViewer 库：这个库是 OSG 的 2.0 版本新增的，它包含了场景中视口及可视化内容的管理类。osgViewer 已将 OSG 集成到多种多样的视窗系统中。

OSG 的 2.0 版本还包括了用于改写界面事件的 osgGA 库。但不久以后 OSG 将会放弃独立的 osgGA 链接库，并将其所包含的功能集成到 osgViewer 中去，而不再作为一个独立的库存在。

在以下部分中，我们将更进一步地讨论这四个核心的链接库。

osg 链接库

命名空间：osg
头文件：<OSG_DIR>/include/osg
Windows 库文件：osg.dll，osg.lib
Linux 和 Mac OS X 库文件：libosg.lib

osg 库是 OpenSceneGraph 的核心部分。它定义了组成场景图形的核心节点，以及帮助用户进行场景图形管理和程序开发的一些附加类。下面将对其中的一些类作简要的叙述。本书的第二章将更为详细地介绍它们，并指导你如何在程序中使用这些类。

场景图形类

场景图形类用于辅助场景图形的构建。OSG 中所有的场景图形类都继承自 osg::Node。从概念上讲，根节点，组节点和叶节点是不同的节点类型。在 OSG 中，它们都源自于 osg::Node，但是特定的类会提供不同的场景图形功能。此外，OSG 中的根节点并不是特定的节点类型，它仅仅是一个没有父类的 osg::Node 类。

- Node：Node 类是场景图形中所有节点的基类。它包含了用于场景图形遍历、拣选、程序回调，以及状态管理的方法。

- Group：Group 类是所有可分支节点的基类。它是场景图形空间组织结构的关键类。
- Geode：Geode 类（即 Geometry Node）相当于 OSG 中的叶节点。它没有子节点，但是包含了 osg::Drawable 对象，而 osg::Drawable 对象中存放了将要被渲染的几何体。
- LOD：LOD 类根据观察点与图像子节点的距离选择显示子节点。通常使用它来创建场景中物体的多个显示层级。
- MatrixTransform：MatrixTransform 类包含了用于实施子节点几何体空间转换的矩阵，以实现场景对象的旋转、平移、缩放、倾斜、映射等操作。
- Switch：Switch 类用布尔掩板来允许或禁止子节点的运作。

以上内容并未涵盖所有的 OSG 节点类型。其它的节点类型还有很多，如 Sequence，PositionAttitudeTransform 等。你可以参考 osg 库的头文件来了解有关它们的信息。

几何体类

Geode 类是 OSG 的叶节点，它包含了渲染用的几何数据。使用以下列出的类可以实现 Geode 中几何数据的存储。

- Drawable：Drawable 类是用于存储几何数据信息的基类，Geode 维护了一个 Drawable 的列表。Drawable 是纯虚类，无法直接实例化。用户必须实例化其派生类，如 Geometry，或者 ShapeDrawable（允许用户程序绘制预定义的几何形状，如球体、圆锥体和长方体）。
- Geometry：Geometry 类与 PrimitiveSet 类相关联，实现了对 OpenGL 顶点数组功能的高级封装。Geometry 保存顶点数组的数据，纹理坐标，颜色，以及法线数组。
- PrimitiveSet：PrimitiveSet 类提供了 OpenGL 顶点数组绘图命令的高层次支持。用户可以从相关的 Geometry 类中取得保存的数据，再使用这个类来指定要绘制的几何体数据的类型。
- Vector 类（Vec2，Vec3 等）：OSG 提供了预定义好的二维，三维和四维

元素向量，支持 float 或者 double 类型。使用这些向量来指定顶点、颜色、法线和纹理坐标的信息。

- Array 类（Vec2Array，Vec3Array 等）：OSG 定义了一些常用的数组类型，如用于贴图纹理坐标的 Vec2Array。指定顶点数组数据时，程序首先将几何数据保存到这些数组中，然后传递至 Geometry 类对象。

也许上面的叙述有些混乱，不过可以总结为这样几条：Geode 类是场景图形的叶节点，其中保存了 Drawable 类；Geometry 类（仅 Drawable 类型）保存了顶点数组数据和及其顶点数组的渲染指令；数据的组成为向量数组。第二章“建立场景图形”将涵盖上述所有的内容并作详细讲解。

状态管理类

OSG 提供了一种机制，用以保存场景图形所需的 OpenGL 渲染状态。在拣选遍历中，同一状态的几何体将被组合集中到一起以使状态的改变呈最小化。在绘制遍历中，状态管理代码将记录当前状态的历史轨迹，以清除冗余的渲染状态变更。

和其它场景图形系统不同，OSG 允许状态与任何场景图形节点相关联，在一次遍历中，状态将呈现出某种继承关系的。

- 状态集合（StateSet）：OSG 在 StateSet 类中保存一组定义状态数据（模式和属性）。场景图形中的任何 osg::Node 都可以与一个 StateSet 相关联。
- 模式（Modes）：与 OpenGL 的函数 glEnable()和 glDisable()相似，模式用于打开或关闭 OpenGL 固定功能（fixed-function）的渲染管道，例如灯光，混合和雾效。方法 osg::StateSet::setMode()在 StateSet 中保存一个模式信息。
- 属性（Attributes）：应用程序使用属性来指定状态参数，例如混和函数，材质属性，雾颜色等。方法 osg::StateSet::setAttribute()在 StateSet 中保存属性信息。
- 纹理模式和属性：纹理模式和属性可应用在 OpenGL 多重纹理的某个指

定纹理单元上。应用程序必须在设定纹理模式和属性时提供纹理单元的信息，注意，和 OpenGL 不同，OSG 不存在缺省的纹理单元。StateSet 类的方法 setTextureMode()和 setTextureAttribute()用于设定状态参量以及纹理单元信息。

- 继承标志：OSG 提供了一些标志量，用于控制场景图形遍历中的状态值。缺省情况下，子节点中的状态集合将重载父节点的状态集合，但是也可以强制父节点的状态重载子节点的状态，或者指定子节点的状态受到保护而不会被其父节点重载。

上述状态系统已经被证明是非常灵活的。所有新添加到 OpenGL 规范中的状态数据，包括最新的 OpenGL 着色语言（Shading Language，[Rost06]），都可以顺利地嵌入到 OSG 状态系统中。

其它实用类

除上述类外，osg 链接库还包括了一些实用的类和工具。其中一些涉及到 OSG 的内存引用计数策略（reference-counted memory scheme），这种策略可以通过清理不再引用的内存以避免内存泄露。第二章“建立场景图形”将详细讲解内存引用计数的内容。

- Referenced：Referenced 类是所有场景图形节点和 OSG 的许多其它对象的基类。它实现了一个用于跟踪内存使用情况的引用计数（reference count）。如果某个继承自 Referenced 的对象，其引用计数的数值到达 0，那么系统将自动调用其析构函数并清理为此对象分配的内存。
- ref_ptr<>：模板类 ref_ptr<>为其模板内容定义了一个智能指针，模板内容必须继承自 Referenced 类（或提供一个与之相同的、能实现引用计数的接口）。当对象的地址分配给 ref_ptr<>时，对象的引用计数将自动增加。同样，清除或者删去 ref_ptr 时，对象的引用计数将自动减少。
- Object：纯虚类 Object 是 OSG 中一切需要 I/O 支持，拷贝和引用计数的对象的基类。所有的节点类，以及某些 OSG 对象均派生自 Object 类。
- Notify：osg 库提供了一系列控制调试，警告和错误输出的函数。用户可

以通过指定一个来自 NotifySeverity 枚举量的数值，设定输出的信息量。OSG 中的大部分代码模块执行时都会显示相关的信息。

osg 链接库还包括了一些本节没有提及的类。你可以参考 osg 库的源代码和头文件，以了解更多的类及其特性。

osgUtil 链接库

命名空间：osgUtil
头文件：<OSG_DIR>/include/osgUtil
Windows 库文件：osgUtil.dll，osgUtil.lib
Linux 和 Mac OS X 库文件：libosgUtil.lib

osgUtil 库集合了许多用于场景图形处理和几何体修改的工具。osgUtil 库最知名之处可能就是其中一系列支持更新、拣选和绘制遍历的类。在典型的 OSG 程序中，这些遍历由更高层次的支持类，例如 osgViewer::Viewer 来进行处理，用户不需要直接和它们进行交互。

交运算（Intersection）

一般来说，3D 程序需要为用户提供一些实现交互和选择的功能，比如图形对象的拾取。通过提供大量用于场景图形交运算的类，osgUtil 库可以高效地支持拾取操作。当用户程序从需要进行图形对象拾取的用户那里接收到事件输入时，可以使用以下的类，获得场景图形中被拾取部分的信息。

- Intersector：Intersector 是一个纯虚类，它定义了相交测试的接口。osgUtil 库从 Intersection 继承了多个类，适用于各种类型的几何体（线段，平面等）。执行相交测试时，应用程序将继承自 Intersector 的某个类实例化，传递给 IntersectionVisitor 的实例，并随后请求该实例返回数据以获取交运算的结果。
- IntersectionVisitor：IntersectionVisitor 类搜索场景图形中与指定几何体相交的节点。而最后相交测试的工作将在 Intersector 的继承类中完成。

- LineSegmentIntersector：LineSegmentIntersector 类继承自 Intersector 类，用于检测指定线段和场景图形之间的相交情况，并向程序提供查询相交测试结果的函数。
- PolytopeIntersector：与 LineSegmentIntersector 类似，该类用于检测由一系列平面构成的多面体的相交情况。当用户点击鼠标，希望拾取到鼠标位置附近的封闭多面体区域时，PolytopeIntersector 类尤其有用。
- PlaneIntersector：与 LineSegmentIntersector 类似，这个类用于检测由一系列平面构成的平面的相交情况。

优化

场景图形的数据结构在理论上有助于实现优化和数据统计工作。osgUtil 库包含的类可以遍历并修改场景图形，以实现渲染的优化和收集场景统计信息的目的。

- Optimizer：正如其名字所示，Optimizer 类用于优化场景图形。其属性使用一组枚举标志进行控制，每一个标志都表示一种特定的优化方式。例如，FLATTEN_STATIC_TRANSFORMS 使用非动态 Transform 节点来变换几何体，通过清除对 OpenGL 的 model-view 矩阵堆栈的修改，实现场景的渲染优化。
- Statistics 和 StatsVisitor：为能够高效地设计 3D 应用程序，开发者应当对将要渲染的对象有尽量多的了解。StatsVisitor 类返回一个场景图形中节点的总数和类型，而 Statistics 类返回渲染几何体的总数和类型。

几何体操作

许多 3D 程序都需要对读入的几何体进行修改，以获得所需的性能和渲染效果。osgUtil 库包含的类支持一些通用的几何形体运算。

- Simplifier：使用 Simplifier 类减少 Geometry 对象中几何体的数目，这有助于低细节层次的自动生成。
- Tessellator：OpenGL 不直接支持凹多边形和复杂多边形。Tessellator 类可根据一组顶点的列表，生成由前述顶点列表所描述的多边形，即一个

osg::PrimitiveSet。

- DelaunayTriangulator：正如其名称所示，这个类实现了 Delaunay 三角网格化运算，根据一组顶点的集合生成一系列的三角形。
- TriStripVisitor：一般来说，由于共享顶点的缘故，连续的条带图元（strip primitives）的渲染效率要高于独立的图元（individual primitives）。TriStripVisitor 类可遍历场景图形并将多边形图元转换成三角形和四边形条带。
- SmoothingVisitor：SmoothingVisitor 类可生成顶点法线，也就是所有共享此顶点的面的法线平均值。
- 纹理贴图生成：osgUtil 库包含了帮助建立反射贴图，中途向量（half-way vector）贴图，以及高光贴图的代码。此外，使用 TangentSpaceGenerator 类还可以逐个的建立各顶点的向量数组，帮助实现凹凸贴图。

osgUtil 库还包含了其它一些本节未提及的类。你可以参考 osgUtil 库的源代码和头文件，以了解更多的类及其特性。

osgDB 链接库

命名空间：osgDB
头文件：<OSG_DIR>/include/osgDB
Windows 库文件：osgDB.dll，osgDB.lib
Linux 和 Mac OS X 库文件：libosgDB.lib

osgDB 库允许用户程序加载、使用和写入 3D 数据库。它采用插件管理的架构，可以支持大量常见的 2D 图形和 3D 模型文件格式。osgDB 负责维护插件的信息注册表，并负责检查将要被载入的 OSG 插件接口的合法性。

OSG 可以支持自己的文件格式。.osg 文件是对场景图形的一种无格式 ASCII 码文本描述，而.osga 文件是一组.osg 文件的有序集合。osgDB 库包含了以上文件格式的支持代码。（另外，OSG 还支持一种二进制的.ive 格式。）

由于大型的 3D 地型数据库通常是多段数据块的组合体，。因此，应用程序

从文件中读取各部分数据库信息时，需要在不干扰当前渲染的前提下以后台线程的方式进行。osgDB::DatabasePager 提供了这样的功能。

osgViewer 链接库

命名空间：osgViewer
头文件：<OSG_DIR>/include/osgViewer
Windows 库文件：osgViewer.dll，osgViewer.lib
Linux 和 Mac OS X 库文件：libosgViewer.lib

osgViewer 库定义了一些视口类，因而可以将 OSG 集成到许多视窗设计工具中，包括 AGL/CGL，FLTK，Fox，MFC，Qt，SDL，Win32，WxWindows，以及 X11。这些视口类支持单窗口/单视口的程序，也支持使用多个视口和渲染器~~面~~的多线程程序。每个视口类都可以提供对摄像机运动，事件处理，以及 osgDB::DatabasePager 的支持。osgViewer 库包含了以下三个可能用到的视口类。

- SimpleViewer：SimpleViewer 类负责管理单一场景图形中的单一视口。使用 SimpleViewer 时，应用程序必须创建一个窗口并设置当前的图形上下文（graphics context）。
- Viewer：Viewer 类用于管理多个同步摄像机，他们将从多个方向渲染单一的视口。根据底层图形系统的能力，Viewer 可以创建一个或多个自己的窗口以及图形上下文，因此使用单一视口的程序也可以在单显示或者多显示的系统上运行。
- CompositeViewer：CompositeViewer 类支持同一场景的多个视口，也支持不同场景的多个摄像机。如果指定各个视口的渲染顺序，用户就可以将某一次渲染的结果传递给别的视口。CompositeViewer 可以用来创建抬头数字显示（HUD），预渲染纹理（prerender textures），也可以用于在单一视口中显示多个视图。

osgViewer 库还包括一些额外的类，用以支持显示统计，窗口提取和场景的

处理工作。

NodeKits

NodeKits 扩展了 Nodes，Drawables 和 StateAttributes 的概念，也可以看作是 OSG 内核中 osg 库的一种扩展。NodeKits 的意义远大于对 OSG 类的继承，事实上它还能够提供对.osg 的封装（一种支持对.osg 文件进行读写的 OSG 插件）。总之，NodeKit 由两部分组成：NodeKit 本身，以及针对.osg 的封装插件库。OSG 2.0 版本包含有六种 NodeKits。

- osgFX 库：此类 NodeKit 提供了额外的场景图形节点，以便于特效的渲染，例如异向光照（anisotropic lighting），凹凸贴图，卡通着色等。
- osgParticle 库：此类 NodeKit 提供了基于粒子的渲染特效，如爆炸、火焰、烟雾等。
- osgSim 库：此类 NodeKit 提供了仿真系统中以及渲染 OpenFlight 数据库所需的特殊渲染功能，例如地形高程图，光点节点，DOF 变换节点等。
- osgText 库：此类 NodeKit 提供了向场景中添加文字的得力工具，可以完全支持 TrueType 字体。
- osgTerrain 库：此类 NodeKit 提供了渲染高度场数据的能力。
- osgShadow 库：此类 NodeKit 提供了支持阴影渲染的框架结构。

如果要详细描述 OSG NodeKits 的所有功能，那将超出本书所许可的范围。2.6 节“NodeKits 与 osgText”将介绍 osgText 的基本用法，而你在学习第二章里 osgText 相关内容的同时，一定也会对其它 NodeKits 的进一步探索以及使用产生浓厚兴趣。

OSG 插件

OSG 的核心库提供了针对多种 2D 图形和 3D 模型文件格式的 I/O 支持。osgDB::Registry 可以自动管理插件链接库。只要提供的插件确实可用，Registry 就可以找到并使用它，应用程序只需调用相应的函数来读取和写入数据文件即可。

osg 库允许用户程序采用“节点到节点”（node-by-node）的方式直接建立场景图形。相反的，OSG 插件允许用户程序仅仅通过编写几行代码就能够从磁盘中调用整个场景图形，或者调用部分的场景图形，然后应用程序可以将其列入整个场景图形系统中去。

OSG 的 2.0 版本支持大量常用的 2D 图形文件格式，包括.bmp，.dds，.gif，.jpeg，.pic，.png，.rgb，.tga 和.tiff。OSG 还支持用于读取电影文件的 QuickTime 插件，并有专门的插件用于读取 FreeType 类型的字体。OSG 广泛支持各种 3D 模型文件格式，其中包括 3D Studio Max（.3ds），Alias Wavefront（.obj），Carbon Graphics’ Geo（.geo），Collada（.dae），ESRI Shapefile（.shp），OpenFlight（.flt），Quake（.md2）和 Terrex TerraPage（.txp）等常见格式。

除上述标准格式以外，OSG 还定义了自身的文件格式。其中，.osg 格式是场景图形的另一种 ASCII 文本描述格式，用户可以使用文本编辑器对其进行编辑和修改；而.ive 格式则是一种二进制格式，经过优化之后它更适合于迅速读取。

除 2D 图形和 3D 模型文件以外，OSG 插件还支持对压缩文件和文件集的 I/O 操作，OSG 目前支持的压缩文件格式有常见的.tgz 和.zip，以及 OSG 特有的.osga 格式。

此外，OSG 还包含了一组名为“PseudoLoader”的插件，以提供除简单文件读取之外更多的功能。

- 缩放、旋转和平移：此类 PseudoLoader 读取文件并在已读入场景图形根节点上添加一个 Transform 节点，并指定放缩、旋转和平移属性的值以配置 Transform。
- 图标：图标类 PseudoLoader 允许在已读入 3D 场景之上显示 HUD（抬头显示）样式的图片文件。

有关在用户程序中使用 OSG 插件的方法，第 2.5 节“文件 I/O”将提供更多信息。

互操作性

用户可以在任何支持 C++库链接的编程环境中使用 OSG。为了确保 OSG 可以在更多环境中运行，OSG 提供了一个语言无关的、可供运行时访问的接口。

osgIntrospection 库允许用户软件使用反射式和自省式的编程范式与 OSG 进行交互。应用程序或其它软件可以使用 osgIntrospection 库和方法迭代 OSG 的类型，枚举量和方法，并且无需了解 OSG 编译和链接时的具体过程，即可调用这些方法。

Smalltalk 和 Objective-C 等语言包括了内建的反射式和自省式支持，但使用 C++的软件开发人员通常无法运用这些特性，因为 C++并未保留必要的元数据（metadata）。为了弥补 C++的这一不足，OSG 提供了一系列自动生成的、从 OSG 源代码创建的封装库，用户程序不需要与这些 OSG 的封装库直接交互，它们将完全由 osgIntrospection 进行管理。

作为 osgIntrospection 及其封装的结果，许多语言如 Java，Tcl，Lua 和 Python，都可以与 OSG 进行交互。如果要详细了解 OSG 的语言互操作性及代码封装，请访问 OSG 维基网站[OSGWiki]，Community 页，并选择 LanguageWrappers。

程序与示例

OSG 发行版包含了五个常用的 OSG 工具程序，它们对于调试和其它基于 OSG 的软件开发均十分有益。

- osgarchive：这个程序用于向.osga 文件包中添加新的文件。也可以用这个程序实现包的分解和列表。
- osgconv：这个程序用于转换文件格式。尤其有用的是，它可以将任意文件格式转换为经过优化的.ive 格式。
- osgdem：这个程序用于将高程图等高度数据及图像数据转换为分页的地形数据库。
- osgversion：这个程序将当前 OSG 版本以及一些记录了 OSG 源代码改动情况和贡献者信息送入 std::cout。
- osgviewer：这是一个灵活而强大的 OSG 场景及模型浏览器。1.3 节“运行 osgviewer”详细说明了这个程序的实用方法。

OSG 发行版还包含了一些展现 API 功能的示例程序。例程源代码同时也展现了 OSG 程序在其开发过程中所应用的大量编程理念和实用技巧。

2、建立一个场景图形

本章将为你展示如何编写代码以建立一个 OSG 场景图形。其内容包括了最基础的场景图形的构建，以及 OSG 载入 3D 模型文件到场景中的能力。

第一部分将介绍内存管理。场景图形及其数据的储存会消耗大量的内存，而这一章节将讨论 OSG 的内存管理机制以及避免悬挂指针和内存泄露的方法。

最简单的场景图形由一个包含了几何信息（geometry）和状态（state）的叶节点组成。第 2.2 节“叶节点（Geode）和几何信息”将介绍几何体、法线以及颜色的设置方法。

接下来的部分中，你将学习如何通过改变 OSG 状态（state）属性和模式来控制几何体的外观。

现实中的应用程序需要的是比单一节点复杂得多的场景图形系统。这一章也将对 OSG 的节点家族作简要介绍。节点家族为场景图形提供各种各样的特性，它们已经封装在各个场景图形库中。

大多数应用程序需要从 3D 模型文件中读取几何体信息。本章也会简介 OSG 的文件读取接口。这一接口提供了对许多 3D 文件格式的支持。

最后，本章还包括了如何添加文字到应用程序的教程。OSG 在其节点工具（NodeKits）中封装了大量高级功能。本章将对其中的一个，osgText，详加讲解。

2.1 内存管理

在开始建立场景图形之前，你需要对 OSG 场景图形节点和数据所使用的内存管理机制有所了解。对这一概念的掌握将有助于编写清洁的代码，并避免出现悬挂指针和内存泄露。

前一章中我们已经对最简单的，从一个根节点引申的场景图形做了图示的讲解。在一个十分典型的实际应用中，程序保存一个指向根节点的指针，但是不保存场景图形中其它节点的指针。根节点将直接或者间接地“引用”（reference）

场景图形中所有的节点。图 2-1 描述了这一典型的场景。

当应用程序不再使用场景图形时，每个节点所使用的内存需要释放以避免内存泄露。如果要编写代码遍历整个场景图形，并依次删除所有的节点及其数据，那么这项工作将是巨大且容易出错的。

幸运的是，OSG 提供了一种自动的“废弃物”收集系统，它使用一种名为内存引用计数器（referencecounted memory）的方式工作。所有的 OSG 场景图形节点均采用引用计数（reference count）的方式，当引用计数值减为 0 时，此对象将被自动释放。其结果是，删除如图 2-1 所示的场景图形时，程序只需要简单地释放指向根节点的指针。这一动作将引发连锁的效果，将场景图形中的所有节点和数据逐一释放，如图 2-2 所示。

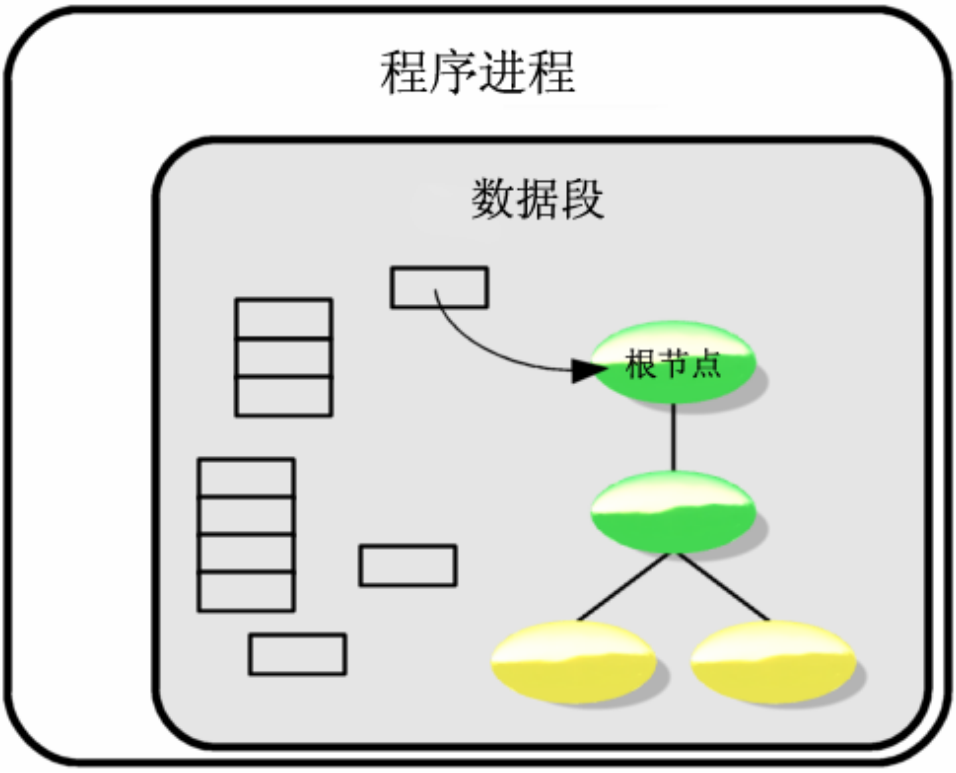


图 2-1 引用场景图形

一个典型的应用程序使用一个指针来保存根节点的地址，从而引用（reference）整个场景图形。应用程序并不保存场景图形中其它节点的指针。其它所有的节点，均通过根节点直接或者间接地被引用。

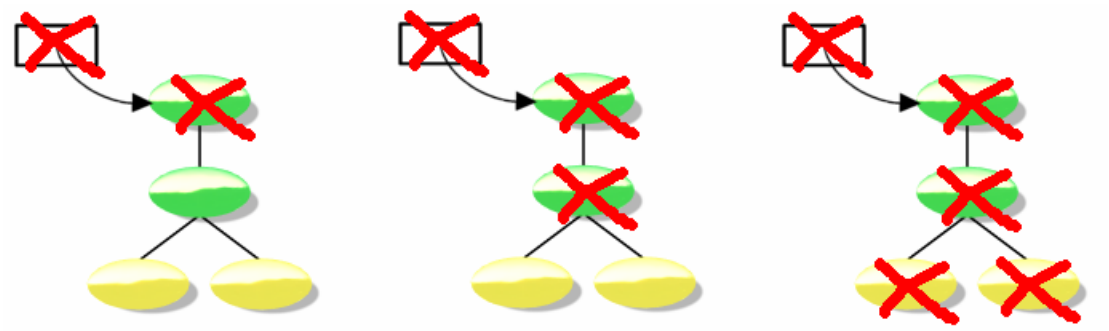


图 2-2 场景图形的连锁释放

OSG 的内存管理机制将在最后一个指向根节点的指针被释放时，释放整个场景图形。

OSG 的“废弃物”收集系统有以下两个组件：

- 通用的基类 osg::Referenced，所有的 OSG 节点和场景图形均继承自这一基类，它包含了一个整型的引用计数器。
- OSG 定义的智能指针模板类 ref_ptr<>。当代码中一个 Referenced 对象指针赋予类型变量 ref_ptr<>时，Referenced 类的引用计数器会自动加 1。

如果程序要保存一个继承自 Referenced 的对象的指针，那么将指针保存在 ref_ptr<>中要好过使用标准 C++指针变量。如果代码中可以始终遵循这一原则，那么当最后一个引用对象的 ref_ptr<>被弃用时，对象所占用的内存将自动释放。

当你创建任何继承自 Referenced 的场景图形节点或数据时，你的应用程序不可以直接释放其内存空间。除了极少数例外，几乎所有的 Referenced 派生类都声明了保护析构函数。这样可以保证，继承自 Referenced 的对象只能够通过减少引用计数器到 0 的方式来释放其内存。以下的文字将详细介绍 Referenced 类和 ref_ptr<>模板，并提供一些示例程序。

警告：

不要使用标准 C++指针变量来长时间地保存继承自 Referenced 类的对象指针。不过例外的是，如果指针最后还会保存在 ref_ptr<>中，那么你可以暂时地使用标准 C++指针变量。不过，使用 ref_ptr<>仍旧是最安全的途径。

2.1.1 Referenced 类

Referenced 类（命名空间：osg）实现了对内存区段的引用计数器功能。所有的 OSG 节点和场景图形数据，包括状态信息，顶点数组，法线，以及纹理坐标，均派生自 Referenced 类。因此，所有的 OSG 场景图形均可以进行内存引用计数。

Referenced 类包括了三个主要组成部分：

- 保护成员整型变量_refCount，用作引用计数，在构造时被初始化为 0。
- 公有函数 ref()和 unref()，用于实现_refCount 值的增加和减少。当_refCount 为 0 时，unref()将自动释放该对象所占用的内存。
- 作为保护成员存在的虚析构函数。堆栈的创建和显示的析构均会因为析构函数受保护而被禁止，而虚函数的特性将允许用户执行子类的析构函数。

总体上来说，用户的代码基本上不需要直接调用 ref()和 unref()函数，只要使用 ref_ptr<>进行处理即可。

2.1.2 ref_ptr<>模板类

ref_ptr<>（命名空间：osg）用于实现一个指向 Referenced 对象的智能指针，并对其引用计数器进行管理。当最后一个引用 Referenced 对象的 ref_ptr<>失去作用时，对象将确保被释放。ref_ptr<>简化了场景图形内存释放的工作，并保证当错误的调用堆栈展开时，对象也可以被正确释放。

ref_ptr<>模板类包括以下三个主要的组成部分：

- 一个私有指针_ptr，用于保存管理内存区域的地址。可以用 get()方法返回_ptr 的值。
- 为了使 ref_ptr<>可以像正常的 C++指针一样工作，重载或定义了一些方法，如 operator->()和 operator=()。
- valid()方法用于判断 ref_ptr<>是否为空，不为 NULL 时返回 TRUE。当程序将一个地址指定给 ref_ptr<>变量时，ref_ptr<>的重载函数 operator=()

将会假定此地址指向一个 Referenced 派生对象，并自动调用 Referenced::ref()，将引用计数值自动加一。

ref_ptr<>变量的引用计数值减少的情形有这样两种：ref_ptr<>被释放（在类的析构函数里执行减一），或者重新进行了赋值（在 operator=()里执行减一）。在以上两种情况中，ref_ptr<>都会通过调用 Referenced::unref()来执行减少引用计数值的操作。

2.1.3 内存管理示例

以下的代码中用到了 osg::Geode 和 osg::Group 类。Geode 类也就是 OSG 的叶节点，它包含了用于渲染的几何信息；请参阅 2.2 节“叶节点（Geode）和几何信息”以了解其详情。Group 节点可以有多个子节点；这一点可以参照 2.3 节“Group 节点”。这两个类均派生自 Referenced 类。

在标准的用例中，用户往往会创建一个节点并将其作为场景图形中另一个节点的子节点：

```
#include <Geode>
#include <Group>
#include <ref_ptr>
...
{
    // 创建新的 osg::Geode 对象。将其赋予 ref_ptr<>，
    // 同时将引用计数器加一。
    osg::ref_ptr<Geode> geode = new osg::Geode;

    // 假设 grp 是指向一个 osg::Group 节点的指针。
    // Group 也使用 ref_ptr<>指向其子节点，
    // 因此 addChild()将再次把引用计数器加一，此时其值为 2。
    grp->addChild( geode.get() );
```

```
}
// ref_ptr<>变量 geode 已经超过了其有效范围，
// 此时把引用计数器减一，其值为 1。
```

这个示例中，其实并没有使用 ref_ptr<>的必要，因为程序本身不需要长时间保存 geode 这个指针。事实上，在上述的简单例子中，ref_ptr<>仅仅是增加了变量构造过程中无用的开支。这里使用简单的 C++指针就已经足够了，因为父节点 osg::Group 内部的 ref_ptr<>已经可以负责管理新的 osg::Geode 所占用的内存了。

```
// 构建新的 osg::Geode 对象，不必增加其引用计数的值。
osg::Geode* geode = new osg::Geode;
// Group 内部的 ref_ptr<>将会把子节点 Geode 的引用计数值置为 1。
grp->addChild( geode );
```

当使用标准 C++指针指向 Referenced 对象时要特别注意，为了保证 OSG 的内存管理系统正常工作，Referenced 对象的地址必须赋予一个 ref_ptr<>变量。上述的代码中，这一赋值过程在 osg::Group::addChild()方法中实现。如果 Referenced 对象从未分配给一个 ref_ptr<>变量，那么这将会引发内存泄露：

```
{
    osg::Geode* geode = new osg::Geode;
}
// 这样将会引发内存泄露！
```

如前所述，你不能够显式地释放派生自 Referenced 的对象，也不能在堆栈中构建它。下面的代码将会产生错误：

```
osg::Geode* geode1 = new osg::Geode;
delete geode1;
```

```
// 此处将产生编译错误：析构函数为保护成员。
{
    osg::Geode geode2;
}
// 此处将产生编译错误：析构函数为保护成员
```

ref_ptr<>类型的变量只能够指向派生自 Referenced 的对象，或者与 Referenced 类有相同接口的对象：

```
// 因为 Geode 派生自 Referenced，因此这样声明正确
osg::ref_ptr<Geode> geode = new osg::Geode;

int i;
osg::ref_ptr<int> rpi = &i;
// 这样是错误的！int 类型并不是派生自 Referenced 类的，因此不支持相应的接口。
```

正如本章前面的部分所述，OSG 的内存管理特性使得整个场景图形树的连锁释放变得十分简单。当指向根节点的唯一的 ref_ptr<>被释放时，根节点的引用计数将减为 0，其析构函数自动释放根节点及指向子节点的 ref_ptr<>指针。因此下面的代码并不会引发内存泄露：

```
{
    // top 使得 Group 节点的引用计数置 1。
    osg::ref_ptr<Group> top = new osg::Group;
    // addChild()使得 Geode 节点的引用计数置 1。
    top->addChild( new osg::Geode );
}
// ref_ptr 变量 top 超过作用周期，Group 和 Geode 所占用的内存被释放。
```

总结：

- 将派生自 Referenced 的对象赋予 ref_ptr<>变量，这一动作将自动调用 Referenced::ref()并使引用计数加一。
- 如果将 ref_ptr<>变量指向其它的对象，或者将其释放，那么将调用 Referenced::unref()方法，使引用计数减一。当计数器的值为 0 时，unref()自动释放对象所占用的内存空间。
- 为新的 Referenced 对象开辟内存空间时，要尽量将其赋予 ref_ptr<>，以保证 OSG 的内存管理工作正确。

作为一本旨在“快速入门指导”的图书，介绍这么多内容未免显得罗嗦。但是这一概念又是十分重要，对于 OSG 内存管理的正确理解，对每个 OSG 开发者来说都是不可或缺的。

下一节将介绍一些由 Referenced 派生出来的常用类，其中涉及的程序段会大量使用 ref_ptr<>变量。当你阅读完本节之后，请牢记，OSG 会在内部为每个长期保存的指针分配 ref_ptr<>，例如前面的例子中，调用 osg::Group::addChild()时的那种情况。

2.2 叶节点（Geode）和几何信息

上一章对 OSG 的内存管理理论进行了介绍。如果你是第一次接触到引用内存计数的概念，不妨参看实际的 OSG 例子程序以加深自己的理解。本章将实现一个简单的 OSG 示例程序，其中用到了前述的内存管理技术，并对使用 OSG 几何相关类来建立场景图形的方法进行了阐述。以下的代码初看起来可能比较晦涩，毕竟你还没有足够熟悉这其中大部分的类。代码之后将对几何信息类作详细的讲解。

清单 2-1 建立简单的场景图形

以下是一个简单的例子程序清单，其详细内容可以在本书的附送代码中找到。createSceneGraph()函数指定了一个单一四边形图元的几何信息。这个四边

形的每个顶点颜色不同，整个几何图元的法线只有一个。

```
#include <osg/Geode>
#include <osg/Geometry>

osg::ref_ptr<osg::Node> createSceneGraph()
{
    // 创建一个用于保存几何信息的对象
    osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;

    // 创建四个顶点的数组
    osg::ref_ptr<osg::Vec3Array> v = new osg::Vec3Array;
    geom->setVertexArray( v.get() );
    v->push_back( osg::Vec3( -1.f, 0.f, -1.f ) );
    v->push_back( osg::Vec3( 1.f, 0.f, -1.f ) );
    v->push_back( osg::Vec3( 1.f, 0.f, 1.f ) );
    v->push_back( osg::Vec3( -1.f, 0.f, 1.f ) );

    // 创建四种颜色的数组
    osg::ref_ptr<osg::Vec4Array> c = new osg::Vec4Array;
    geom->setColorArray( c.get() );
    geom->setColorBinding( osg::Geometry::BIND_PER_VERTEX );
    c->push_back( osg::Vec4( 1.f, 0.f, 0.f, 1.f ) );
    c->push_back( osg::Vec4( 0.f, 1.f, 0.f, 1.f ) );
    c->push_back( osg::Vec4( 0.f, 0.f, 1.f, 1.f ) );
    c->push_back( osg::Vec4( 1.f, 1.f, 1.f, 1.f ) );

    // 为唯一的法线创建一个数组
    osg::ref_ptr<osg::Vec3Array> n = new osg::Vec3Array;
    geom->setNormalArray( n.get() );
```

```
    geom->setNormalBinding( osg::Geometry::BIND_OVERALL );
    n->push_back( osg::Vec3( 0.f, -1.f, 0.f ) );

    // 由保存的数据绘制四个顶点的多边形
    geom->addPrimitiveSet(
    new osg::DrawArrays( osg::PrimitiveSet::QUADS, 0, 4 ) );

    // 向 Geode 类添加几何体（Drawable）并返回 Geode
    osg::ref_ptr<osg::Geode> geode = new osg::Geode;
    geode->addDrawable( geom.get() );
    return geode.get();
}
```

清单 2-1 的代码中大量使用了前面介绍过的 ref_ptr<>模板类。清单代码中所有分配的内存均使用了引用计数的管理方法。用于创建场景图形的 createSceneGraph()函数的返回值也是一个 ref_ptr<>。严格说来，这些代码完全可以使用标准 C++指针来改写，因为函数最后的返回地址将会保存在 ref_ptr<>中。但是，在你的程序中坚持使用 ref_ptr<>是一个很好的习惯，因为它可以在异常事件产生或者中断函数并返回时自动释放内存。本书及其示例代码中均会使用 ref_ptr<>，以鼓励用户适应这种内存管理方式。

图 2-3 清单 2-1 中的场景图形

代码清单 2-1 创建了只有单一叶节点 Geode 的场景图形。

除了代码清单 2-1 中所示的场景图形创建过程，你还可能希望学习将其渲染成图形和动画的方法。不过本章的例子将仅使用 osgviewer 来观察场景图形，关于如何编写用户程序的场景观察代码的讲解，请参阅第三章“在用户程序中使用

OpenSceneGraph”。如果想要在 osgviewer 中观察场景图形，只需将其写入到磁盘文件即可。代码清单 2-2 实现了调用清单 2-1 中的函数并将场景图形作为.osg 文件写入磁盘的过程。当场景图形作为文件保存在磁盘上之后，用户就可以使用 osgviewer 来观察它了。

清单 2-2 写入场景图形到磁盘

这一段代码编写是这个简单例子程序的 main()函数入口。main()函数调用清单 2-1 中的 createSceneGraph()函数来创建场景图形，并将其写入到一个名为“Simple.osg”的文件中。

```
#include <osg/ref_ptr>
#include <osgDB/Registry>
#include <osgDB/WriteFile>
#include <osg/Notify>
#include <iostream>
using std::endl;

osg::ref_ptr<osg::Node> createSceneGraph();

int main( int, char** )
{
    osg::ref_ptr<osg::Node> root = createSceneGraph();
    if (!root.valid())
        osg::notify(osg::FATAL) << "Failed in createSceneGraph()."
                                << endl;

    bool result = osgDB::writeNodeFile( *(root.get()), "Simple.osg" );
    if ( !result )
        osg::notify(osg::FATAL) << "Failed in osgDB::writeNode()."
                                << endl;
```

}

在调用清单 2-1 所示的函数创建场景图形之后，清单 2-2 的代码将这个场景图形作为文件“Simple.osg”写入到磁盘中。文件格式.osg 是 OSG 特有的 ASCII 编码文件格式。作为 ASCII 文件，.osg 通常较大且载入较为缓慢，因此在产品级的代码中很少使用。不过，作为开发时的调试环境和快速演示，这种格式还是十分有用的。

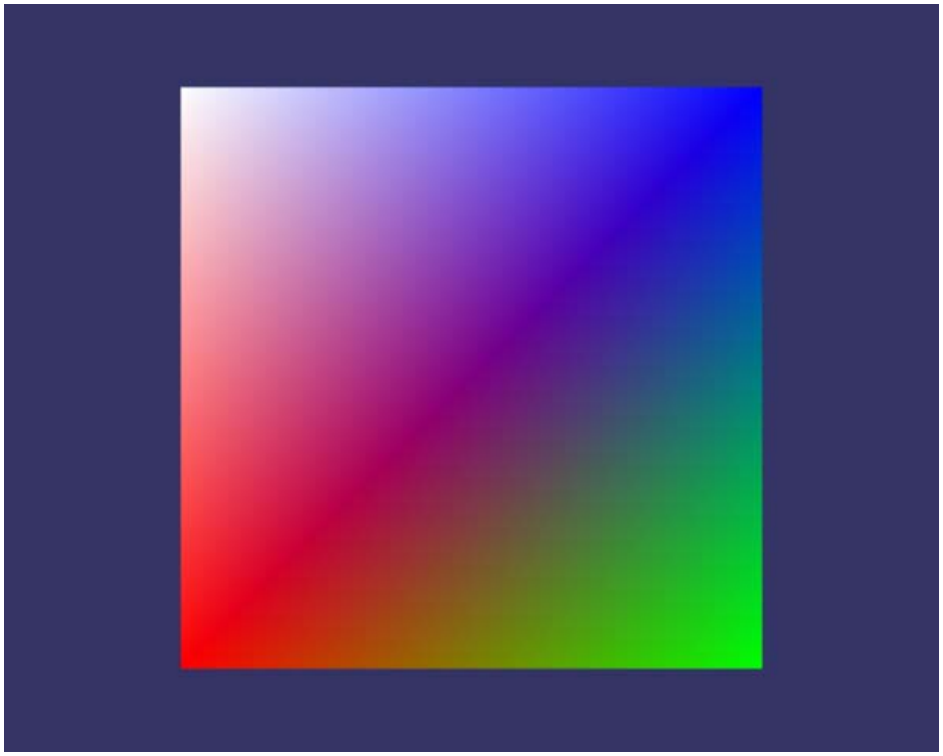


图 2-4 osgviewer 中显示的示例代码场景图形

本图中显示的是清单 2-1 创建的四边形图元，它已经被清单 2-2 的代码写入一个.osg 文件并显示在 osgviewer 中。

清单 1 和 2 所列出的代码均出自本书附带源码中的例子 Simple。如果你还没有从本书的网站上获取附带源码，请马上登陆获取并编译运行 Simple。运行该程序之后，你可以在当前目录下看到生成的文件 Simple.osg。如果你想观察其中的场景图形，请使用 osgviewer：

osgviewer Simple.osg

osgviewer 将会显示一幅类似于图 2-4 的图像。第一章已经对 osgviewer 和它的用户接口作了介绍。举例来说，用户可以使用鼠标左键旋转渲染的几何体，使用右键实现放大和缩小。

清单 2-1 中所示的代码大量使用了 OSG 的几何体相关类。下面的文字将会对这些类的应用提供一些高层次的讲解。

2.2.1 Geometry 类概述

清单 2-1 中的代码可能会引起一些混乱，不过实际上，它仅仅执行了三个步骤的操作。

1、创建了顶点、法线和颜色数据的数组。

2、将 osg::Geometry 对象实例化，并将上述数组添加到对象中。同时还添加了一个 osg::DrawArrays 对象以指定数据绘制的方式。

3、将 osg::Geode 场景图形节点实例化，并将几何信息添加到其中。

本节将对这里的每个步骤作一一讲解。

向量与数组类

OSG 定义了大量的类用于保存向量数据，如顶点、法线、颜色、纹理坐标等。osg::Vec3 是一个三维浮点数数组；可以用来保存顶点和法线数据；osg::Vec4 用于保存颜色数据；而 osg::Vec2 可以用于保存 2D 纹理坐标。除了简单的向量存储以外，这些类还提供了完整的有关长度计算、点乘和叉乘、向量加法、向量矩阵乘法的运算函数。

OSG 还定义了用于保存对象的模板数组类。数组模板的最常见用法是保存向量数据。事实上，正因为它十分常用，因此 OSG 提供了对向量数据数组的一系列类型定义：osg::Vec2Array，osg::Vec3Array 以及 osg::Vec4Array。

清单 2-1 使用 Vec3 为每个 XYZ 顶点坐标创建了独立的三元向量，然后将各个 Vec3 压入 Vec3Array。之后，采用基本相同的方式，用 Vec3Array 来保存 XYZ 法线向量数据。清单 2-1 中使用了 Vec4 和 Vec4Array 来保存颜色数据，因为颜色是一个四元数（红、绿、蓝、Alpha）。在后面的章节中，还会出现使用 Vec2

和 Vec2Array 来保存二元纹理坐标的例子。

所有的数组类型均继承自 std::vector，因此可以使用 push_back()方法添加新的元素，如清单 2-1 所示。作为 std::vector 的子类，数组类同样可以使用 resize()和 operator[]()方法。下面是一段使用 resize()和 operator[]()方法的例子。

```
osg::ref_ptr<osg::Vec3Array> v = new osg::Vec3Array;
geom->setVertexArray( v.get() );
v->resize( 4 );
(*v)[ 0 ] = osg::Vec3( -1.f, 0.f, -1.f );
(*v)[ 1 ] = osg::Vec3( 1.f, 0.f, -1.f );
(*v)[ 2 ] = osg::Vec3( 1.f, 0.f, 1.f );
(*v)[ 3 ] = osg::Vec3( -1.f, 0.f, 1.f );
```

Drawables 类

OSG 定义了 osg::Drawable 类，用于保存要渲染的数据。Drawable 是一个无法直接实例化的虚基类。OSG 核心库从 Drawable 派生出三个子类：osg::DrawPixels，封装了 glDrawPixels()的相关功能；osg::ShapeDrawable，提供了一些已经定义好的几何体的使用接口，如圆柱体和球；以及 osg::Geometry。例子代码中出现了 Geometry 类，它也是使用最为灵活，也最为广泛的一个子类。

如果你对于 OpenGL 的顶点数组已经比较了解，那么对你来说 Geometry 类也十分易于使用。Geometry 类为应用程序提供了指定顶点数据数组和数据解析渲染的接口。这一点与 OpenGL 用于指定顶点数组数据的指令入口类似（参见 glVertexPointer()和 glNormalPointer()的使用）。清单 2-1 的代码主要使用了以下一些 Geometry 类方法：

- setVertexArray()，setColorArray()，setNormalArray() - 这些方法与 OpenGL 的 glVertexPointer()，glColorPointer()与 glNormalPointer()类似。用户程序可以使用它们来指定顶点数组，颜色，以及法线数据。setVertexArray()和 setNormalArray()均使用一个 Vec3Array 指针作为输入参数，而 setColorArray()使用一个 Vec4Array 指针。

- setColorBinding()和 setNormalBinding() - 这些方法用于设置 Geometry 类中颜色和法线数据的绑定方式。其输入参数为 Geometry 类的枚举量。清单 2-1 中，颜色绑定方式为 osg::Geometry::BIND_PER_VERTEX，即每种颜色对应一个顶点。而法线的绑定方式为 osg::Geometry::BIND_OVERALL，即整个 Geometry 几何体对应唯一的一个法线数据。
- addPrimitiveSet() - 这个方法用于设置 Geometry 类数据渲染的方法。其参数为一个 osg::PrimitiveSet 指针。PrimitiveSet 是一个无法直接实例化的虚基类。一个 Geometry 类可以添加多个 PrimtiveSet 对象。

addPrimitiveSet()方法允许用户程序设定 OSG 如何绘制 Geometry 类保存的几何数据。清单 2-1 中指定了一个 osg::DrawArrays 对象。DrawArrays 继承自 PrimitiveSet。它相当于对 glDrawArrays()顶点数组绘图命令的封装。其它的 PrimitiveSet 派生类（DrawElementsUByte，DrawElementsUShort，DrawElementsUInt）则形同 OpenGL 的 glDrawElements()指令入口。OSG 还提供了 DrawArrayLengths 类，它并不等价于任何 OpenGL 的函数。从实现的功能上讲，它相当于多次调用 glDrawArrays()，且每次均使用不同的长度和索引范围。这其中最常用的 DrawArrays 类，其构造函数声明如下：

osg::DrawArrays::DrawArrays(GLenum mode, GLint first,
GLsizei count);

其中的参数 mode 可以选择十个 OpenGL 图元枚举类型之一，例如 GL_POINT，GL_LINES，或者 GL_TRIANGLE_STRIP。基类 PrimitiveSet 为之定义了相对应的枚举量，例如 osg::PrimitiveSet::POINTS 等，用户程序可以随意使用这些枚举变量的任何一种。

在 OSG 执行渲染时，参数 first 是顶点数组中的要使用的第一个元素，而 count 是将要使用的顶点元素的总数。例如，如果用户的顶点数组数据中包含有 6 个顶点，而用户准备用它们来需渲染连续的多个三角形，那么可以添加如下的 DrawArrays 图元到 Geometry 对象中：

geom->addPrimitiveSet(

```
new osg::DrawArrays( osg::PrimitiveSet::TRIANGLE_STRIP, 0, 6 )
);
```

向 Geometry 对象添加了顶点数据，颜色数据，法线数据，以及 DrawArrays 图元序列之后，清单 2-1 中的代码将完成最后一项对 Geometry 几何体的操作：将其关联到场景图形的一个节点上。下一节将对这一点详加阐述。

OSG 绘图的方式：

虽然 PrimitiveSet 子类提供了与 OpenGL 顶点数组特性几乎等价的功能，但是不要认为 PrimitiveSet 在内部机理上也是如此使用顶点数组的。根据渲染环境的不同，OSG 可能会使用顶点数组（附加或不附加缓冲对象），显示列表，甚至 glBegin()/glEnd()来渲染几何体。继承自 Drawable 类的对象（例如 Geometry）在缺省条件下将使用显示列表。你也可以调用 osg::Drawable::setUseDisplayList(false)来改变这一特性。

如果用户设置了 BIND_PER_PRIMITIVE 这种绑定方式，那么 OSG 将依赖 glBegin()/glEnd() 函数进行渲染。BIND_PER_PRIMITIVE 方式为每个独立的几何图元设置一种绑定属性（例如，为 GL_TRIANGLES 中的每个三角形）。

叶节点（Geode）

osg::Geode 类是 OSG 的叶节点，它用于保存几何信息以便渲染。清单 2-1 中建立了一个可能是最简单的场景图形，它只有一个单一的叶节点。在清单 2-1 的最后，createSceneGraph()函数以 ref_ptr<>的形式返回 Geode 的地址给一个 osg::Node。在 C++语言中这是合法的，因为 Geode 类派生自 Node 类。（在定义上，所有的 OSG 节点均派生自 Node 类。）

作为叶节点，osg::Geode 不会再有子节点，但是它可以包含几何体信息。Geode 这个名字是“geometry node”的简拼，意即包含几何信息的节点。应用程序渲染的任何几何信息都必须与 Geode 相关联。Geode 提供了 addDrawable()方

法来关联应用程序中的几何信息。

Geode::addDrawable()将一个 Drawable 指针作为传入的参数。如前一节所述，Drawable 类是一个派生了 Geometry 等很多子类的虚基类。请参阅清单 2-1 中的示例代码，它演示了使用 addDrawable()向 Geode 添加 Geometry 对象的方法。这一操作的代码部分在 createSceneGraph()函数的尾部，返回 Geode 指针之前。

2.3 组节点（Group）

OSG 的组节点，osg::Group，允许用户程序为其添加任意数量的子节点，子节点本身也可以继续分发子节点，如图 2-5 所示。Group 作为基类，派生出了许多实用的节点类，这其中包括本节将要介绍的 osg::Transform，osg::LOD 和 osg::Switch。

Group 类是由 Referenced 类派生的。在通常情况下，只有 Group 的父节点引用了这个 Group 对象，因此当场景图形的根节点被释放时，会引发连锁的内存释放动作，因此不会产生内存泄露。

Group 类是 OSG 编程的核心部分，因为它使得用户应用程序可以有效组织场景图形中的数据。Group 类的强大之处在于它管理子节点的接口。Group 还从基类 osg::Node 中继承了用于管理父节点的接口。本节将概述子接口和父接口的相关知识。在对它们进行介绍之后，本节还会对三个常用的继承自 Group 的类作逐一讲解，它们分别是 Transform（变换），LOD（细节层次）和 Switch（开关）节点。

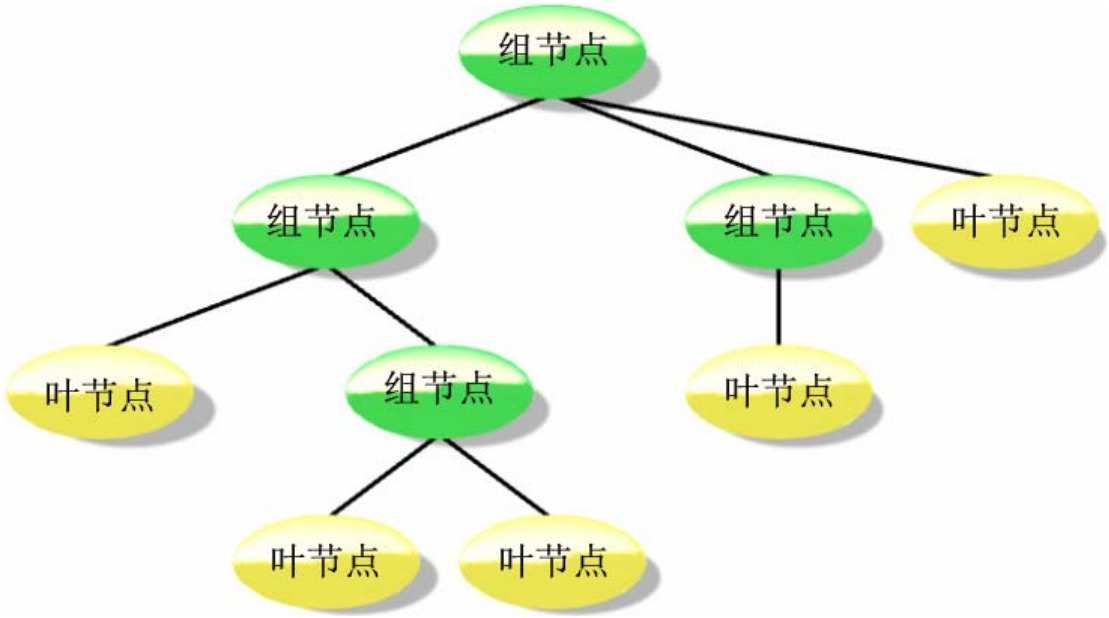


图 2-5 组节点

图中绿色的组节点可以有多个子节点，这些子节点同样可以是自己拥有子节点的组节点。

2.3.1 子接口

Group 类定义了针对子节点的接口，所有派生自 Group 的节点均会继承这一接口。大多数 OSG 的节点均派生自 Group（不过 Geode 是一个例外），因此通常来说，你可以假设绝大部分节点均支持子接口的特性。

Group 使用 std::vector< ref_ptr<Node> >来保存所有子节点的指针，这是一个使用 ref_ptr<>模板的 Node 类数组。正因为 Group 使用了数组，所以你可以根据索引来访问某个子节点。Group 同时还使用 ref_ptr<>来保证 OSG 的内存管理机制生效。

下面的代码段是 Group 类关于子接口的部分声明。所有的类均位于 osg 命名空间。

```
class Group : public Node {
    public:
        ...
```

```
        // 添加一个子节点。
        bool addChild( Node* child );
        // 删除一个子节点，
        // 如果输入节点不是子节点，那么放弃操作并返回 false。
        bool removeChild( Node* child );
        // 用新的子节点替换一个子节点。
        bool replaceChild( Node* origChild, Node* newChild );
        // 返回子节点的数目。
        unsigned int getNumChildren() const;
        // 如果指定节点是一个子节点，那么返回 true。
        bool containsNode( const Node* node ) const;
        ...
};
```

一个简单的场景图形可能会包含一个 Group 父节点和两个 Geode 子节点。你可以使用下面的代码创建这样的一个场景图形。

```
{
    osg::ref_ptr<osg::Group> group = new osg::Group;
    osg::ref_ptr<osg::Geode> geode0 = new osg::Geode;
    group->addChild( geode0.get() );
    osg::ref_ptr<osg::Geode> geode1 = new osg::Geode;
    group->addChild( geode1.get() );
}
```

注意，Group 使用了 ref_ptr<>指向其子节点。在这个例子中，geode0 和 geode1 均由 group 引用。即使 geode0 和 geode1 超过了有效范围，它们占用的内存也依然存在，而 group 被释放后，内存也随即释放。

2.3.2 父接口

Group 类从 Node 类继承了父节点管理的接口。同样继承自 Node 的 Geode 类当然也可使用这些接口。OSG 允许节点有多个父节点。

下面的代码段是 Node 类关于父接口的部分声明。所有的类均位于 osg 命名空间。

```
class Node : public Object {
    public:
        ...
        typedef std::vector<Group*> ParentList;
        // 返回父节点的列表。
        const ParentList& getParents() const;
        // 返回指定索引处父节点的指针。
        Group* getParent( unsigned int I );
        // 返回父节点的数目。
        unsigned int getNumParents() const;
        ...
};
```

Node 类继承自 osg::Object。Object 类是应用程序无法直接实例化的虚基类。Object 类提供了一系列接口用于保存和获取名称，指定保存数据是静态的还是动态可更改的，以及继承自 Referenced 的内存管理机制。第 3.2 节“动态更改”将更详细地介绍 Object 类的名称和动态数据接口。

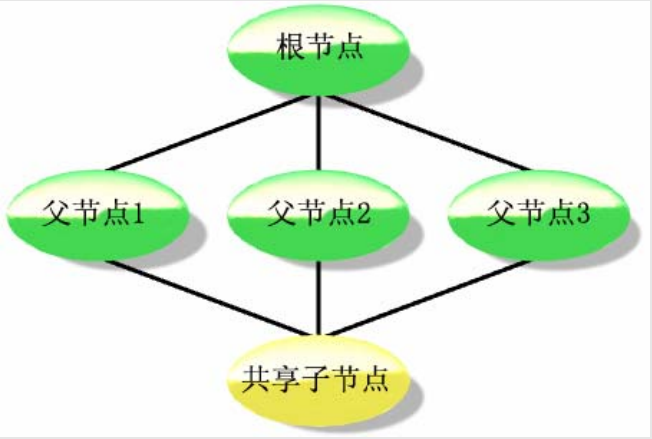
注意，osg::Node::ParentList 是一个保存标准 C++指针的 std::vector。虽然一个 Node 保存了所有父节点的地址，但是 Node 类并不需要使用 OSG 内存管理机制来引用其父节点。当父节点被释放时，父节点会自动从子节点的相应列表中删除自己。

通常情况下，一个节点有一个父节点（getNumParents()返回 0），要获取这个父节点的指针，可以调用 getParent(0)。

在构建和操作场景图形的过程中，你可能会经常运用到子接口和父接口。不过，继承自 Group 的类还提供了更为有用的附加功能。下面的章节将讨论其中的三个常用节点类型。

多个父节点：

如果用户把同一个节点作为多个节点的子节点，那么这个子节点就拥有多个父节点，如图表所示。用户可能希望以此来多次渲染同一幅子图，而不必重复创建这幅子图的拷贝。第 2.4.3 节“渲染状态设置示例”将演示如何以不同的变换方式和渲染状态来渲染同一个 Geode 的信息。



如果一个节点有多个父节点，OSG 将多次遍历这个节点。每个父节点都将自己的 ref_ptr<>指向这个子节点，因此这个子节点不会被释放，直到所有的父节点放弃引用它为止。

2.3.3 变换节点（Transform）

OSG 通过 osg::Transform 节点类家族来实现几何数据的变换。Transform 类继承自 Group 类，它可以有多个子节点。但是 Transform 类是一个无法由应用程序实例化的虚基类。用户应当使用 osg::MatrixTransform 或 osg::PositionAttitudeTransform 来替代它，这两者均继承自 Transform。它们提供了不同的变换接口。根据用户程序的需要，可以使用其中任意一个或者同时使用这两者。

Transform 对 OpenGL 的模型-视图（model-view）矩阵堆栈产生影响。多个紧密排列的 Transform 节点可以创建连续级联的变换，将多个矩阵压入矩阵堆栈的栈顶，这一点与 OpenGL 的矩阵操作命令相同（glRotatef()，glScalef()等）。

Transform 节点允许用户指定参考系。缺省情况下，参考系是相对的（osg::Transform::RELATIVE_RF），因此也就有了前述的级联特性。而 OSG 也允许用户指定绝对参考系，这与调用 OpenGL 函数 glLoadMatrixf()是等同的。

```
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
mt->setReferenceFrame( osg::Transform::ABSOLUTE_RF );
```

矩阵和 osg::Matrix 类：

osg::Matrix 类保存了一个 4×4 的矩阵，共包含了 16 个浮点数，并提供了相应的运算操作。Matrix 类不是由 Referenced 派生的，因此不能实现引用计数。

Matrix 类提供的接口与 OpenGL 标准以及大多数 OpenGL 书籍所述有些不同。Matrix 提供了与 C/C++二维列数组相一致的接口：

```
osg::Matrix m;
m( 0, 1 ) = 0.f; //  设置第二个元素（第 0 行第 1 列）
m( 1, 2 ) = 0.f; //  设置第七个元素（第 1 行第 2 列）
```

OpenGL 矩阵可以表示为一维数组，也就是其相关文档中所述的行数组：

$$\begin{bmatrix} m0 & m4 & m8 & m12 \\ m1 & m5 & m9 & m13 \\ m2 & m6 & m10 & m14 \\ m3 & m7 & m11 & m15 \end{bmatrix}$$

```
GLfloat m;
m[1] = 0.f; //  设置第二个元素
m[6] = 0.f; //  设置第七个元素
```

虽然有这样的区别，但 OSG 和 OpenGL 矩阵在内存中的存放并不相同——OSG 在向 OpenGL 送入矩阵参数时不用进行额外的转换。但是对开发者来说，还是有必要在使用各种数据之前记得转换 OSG 的矩阵。

Matrix 提供了完整的向量-矩阵乘法和矩阵级联的功能。要使 Vec3 变量 v 沿着新的原点矩阵 T，按照矩阵 R 执行旋转变换，只需要以下的代码：

```
osg::Matrix T;
T.makeTranslate( x, y, z );
osg::Matrix R;
R.makeRotate( angle, axis );
Vec3 vPrime = v * R * T;
```

Matrix 采用了左乘（premultiplication）操作，这一点与 OpenGL 文档中所述的 v' = TRv 正好相反。

上述 OpenGL 运算得到的结果是相同的，因为 OpenGL 的行数组矩阵采用右乘的方式，而 OSG 的列数组矩阵采用左乘的方式，这两者是等价的。

如果要对一个 Geode 使用上述变换，首先应该创建一个包含矩阵 T 的 MatrixTransform 节点，向其添加一个包含矩阵 R 的 MatrixTransform 子节点，然后再向这个旋转变换节点添加子节点 Geode，如图所示。这与下面的 OpenGL 语句是等价的：

```
glMatrixMode( GL_MODELVIEW );
glTranslatef( ... ); // 变换矩阵 T
glRotatef( ... ); // 旋转矩阵 R
...
glVertex3f( ... );
glVertex3f( ... );
...
```

总的说来，OSG 的列数组接口与 OpenGL 的行数组接口有所不同，但是它们在内部采用了等价的运算操作，因此它们的矩阵是 100%兼容的。

矩阵变换节点（MatrixTransform）

MatrixTransform 的内部使用了 osg::Matrix 对象。要创建一个执行变换的

MatrixTransform 节点，首先要创建变换矩阵 Matrix 并将其关联到 MatrixTransform。

```
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
osg::Matrix m;
m.setTranslate( x, y, z );
mt->setMatrix( m );
```

Matrix 不是 Referenced 的派生类。用户可以创建 Matrix 的局部变量。MatrixTransform::setMatrix()方法将 Matrix 的参数复制到 MatrixTransform 节点的 Matrix 成员变量中。

Matrix 类提供了通用变换的操作接口，例如移动、旋转和放缩。用户可以显式地设置矩阵：

```
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
osg::Matrix m;
// 设置矩阵的所有 16 个值：
m.set( 1.f, 0.f, 0.f, 0.f,
       0.f, 1.f, 0.f, 0.f,
       0.f, 0.f, 1.f, 0.f,
      10.f, 0.f, 0.f, 1.f ); // X 方向移动 10
mt->setMatrix( m );
```

本书源代码中的渲染状态实例使用了多个 MatrixTransform 节点来渲染不同位置的同一个 Geode，这些 MatrixTransform 节点各自有不同的渲染状态。第 2.4.3 节“渲染状态设置示例”将对此进行讨论。

位置属性变换节点（PositionAttitudeTransform）

使用 PositionAttitudeTransform 节点可以实现使用 Vec3 位置坐标和一个四元

数完成的变换操作。OSG 使用 osg::Quat 来保存四元数方位数据。Quat 不是从 Referenced 派生的，因此不能进行引用计数。

Quat 提供了丰富的设置接口。下面的代码包含了一些创建和设置四元数的函数。

```
// 创建一个四元数，实现围绕 axis 旋转 theta 弧度。
Float theta( M_PI * .5f );
osg::Vec3 axis( .707f, .707f, 0.f );
osg::Quat q0( theta, axis );

// 使用 yaw（绕 Z 轴）、pitch（绕 X 轴）、roll（绕 Y 轴）角度
// 创建一个四元数。
osg::Vec3 yawAxis( 0.f, 0.f, 1.f );
osg::Vec3 pitchAxis( 1.f, 0.f, 0.f );
osg::Vec3 rollAxis( 0.f, 1.f, 0.f );
osg::Quat q1( yawRad, yawAxis, pitchRad, pitchAxis, rollRad, rollAxis );

// 两个四元数级联。
Q0 *= q1;
```

你可以向一个 PositionAttitudeTransform 节点添加任意多个子节点，因为 PositionAttitudeTransform 继承了 Group 的子接口特性。与 MatrixTransform 类似，PositionAttitudeTransform 节点可以根据位置和属性参数执行子节点几何体的变换。

2.3.4 细节层次节点（LOD）

使用 osg::LOD 节点可以实现不同细节层次下物体的渲染。LOD 继承自 Group 类，因此也继承了子接口的特性。LOD 允许用户指定各个子节点的有效范围。

该范围包括了最大和最小值，缺省情况下其意义表示为距离。当某个子节点与观察者的距离在这个节点的有效范围之内时，LOD 将显示这个节点。LOD 的子节点可以按照任何顺序存放，且不必按照细节的数量或者距离排序。

图 2-6 所示为一个 LOD 节点及其三个子节点。第一个子节点是自己包含了子节点的 Group 节点。当观察点的距离符合第一个子节点的有效范围时，OSG 将遍历这个节点及其子节点。LOD 节点对于第二个和第三个子节点的逻辑与之相同。根据距离和有效范围的不同，OSG 可以不显示、显示任意一个，或者显示所有的 LOD 子节点。

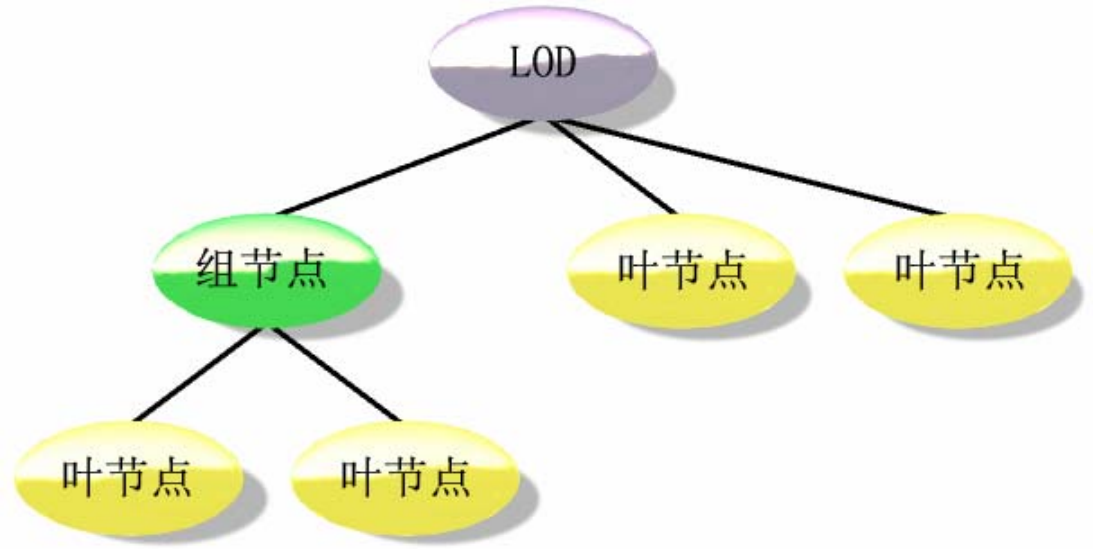


图 2-6 LOD 节点

本图所示为 LOD 节点及其三个子节点。每个子节点都有一个有效范围。当观察点的距离符合子节点的有效范围时，LOD 节点允许该子节点显示。

下面的代码添加了一个有效范围 0 到 1000 的 Geode 子节点。父节点 LOD 将在视点与这个子节点的距离小于 1000 个单位时显示它。

```
osg::ref_ptr<osg::Geode> geode;
...
osg::ref_ptr<osg::LOD> lod = new osg::LOD;
// 当距离在 0.f 与 1000.f 之间时，显示 geode。
```

```
lod->addChild( geode.get(), 0.f, 1000.f );
```

如果子节点的范围有交叠，LOD 将同时显示多个子节点。

```
osg::ref_ptr<osg::Geode> geode0, geode1;
...
osg::ref_ptr<osg::LOD> lod = new osg::LOD;
// 当距离在 0.f 与 1050.f 之间时，显示 geode0。
lod->addChild( geode0.get(), 0.f, 1050.f );
// 当距离在 950.f 与 2000.f 之间时，显示 geode1。
lod->addChild( geode1.get(), 950.f, 2000.f );
// 因此，当距离在 950.f 到 1050.f 之间时，将同时显示 geode0 和 geode1。
```

缺省条件下，LOD 计算视点到物体包络体中心点的距离。如果这样并不符合渲染要求，那么用户可以指定一个自定义的中心点。下面的代码将设置 LOD 节点使用用户定义的中心点。

```
osg::ref_ptr<osg::LOD> lod = new osg::LOD;
// 使用用户定义的中心点来计算距离。
lod->setCenterMode( osg::LOD::USER_DEFINED_CENTER );
// 设置自定义中心点的 X 坐标 10，Y 坐标 100。
Lod->setCenter( osg::Vec3( 10.f, 100.f, 0.f ) );
```

osg::LOD::DISTANCE_FROM_EYE_POINT or osg::LOD::PIXEL_SIZE_ON_SCREEN.

如果要重新设置距离计算的缺省特性为物体的包络球，可以调用 osg::LOD::setCenterMode(osg::LOD::USE_BOUNDING_SPHERE_CENTER)。

缺省条件下，LOD 使用距离的最大值和最小值来表示范围，但是用户也可以要求 LOD 使用像素大小来设置范围值，此时如果子节点在屏幕上的像素大小

符合其有效范围，LOD 将显示这个子节点。要设置 LOD 节点的有效范围模式，可以调用 osg::LOD::setRangeMode()，并设置参数为 PIXEL_SIZE_ON_SCREEN，或者 DISTANCE_FROM_EYE_POINT。

2.3.5 开关节点（Switch）

使用 osg::Switch 节点可以渲染或者跳过指定的子节点。Switch 类的典型用法有，根据当前渲染的负荷有选择地渲染子图形以实现渲染性能的均衡，或者在游戏的界面和层级之间有选择地切换。而在应用程序 Present3D 中，使用 Switch 节点实现了连续幻灯片的显示。

```
osg::ref_ptr<osg::Group> group0, group1;
...
// 创建一个 Switch 父节点和两个 Group 子节点。
osg::ref_ptr<osg::Switch> switch = new osg::Switch;
// 渲染第一个子节点。
switch->addChild( group0.get(), true );
// 暂时不渲染第二个子节点。
switch->addChild( group1.get(), false );
```

如果你的程序中没有指定子节点是否要渲染，那么 Switch 默认值为 true。

```
// 添加子节点，默认为显示这个节点。
switch->addChild( group0.get() );
```

你可以使用 Switch::setNewChildDefaultValue()来改变新加入节点的缺省值。

```
// 改变新加入节点的缺省值：
switch->setNewChildDefaultValue( false );
// 现在新加入的子节点将会按照缺省值关闭显示。
```

```
switch->addChild( group0.get() );
switch->addChild( group1.get() );
```

当你将一个子节点添加到 Switch 之后，你可以改变它的参数。使用 Switch::setChildValue()传入子节点和新的开关值。

```
// 添加子节点，开关值初始化为 true。
switch->addChild( group0.get(), true );
// 禁止 group0 的显示。
switch->setChildValue( group0.get(), false );
```

上面的代码段只是理想化的情况。如果要在程序运行时实现允许和禁止各种条件下 Switch 开关的子节点显示，你需要使用节点更新回调（osg::NodeCallback）或者节点访问（osg::NodeVisitor）来控制场景图形的节点和各帧渲染。本书将在第三章“在用户程序中使用 OpenSceneGraph”里讨论更新回调和 NodeVisitor 类的话题。

2.4 渲染状态（Rendering State）

在第 2.2 节“叶节点（Geode）和几何信息”的示例代码中，我们建立了一个场景图形，它生成的数据可以通过 osgviewer

程序来进行检验。当你在 osgviewer 中旋转矩形图形时，你可以立即发现这个矩形已经被视图中的一个光源照亮了。而 osgviewer 是通过配置 OSG 渲染状态来实现此光照的。

光照只是 OSG 支持的许多渲染状态特性之一。OSG 支持绝大部分的 OpenGL 固定功能管道（fixed function pipeline）渲染，例如 Alpha 检验，Blending 融合，剪切平面，颜色蒙板，面拣选（face culling），深度和模板检验，雾效，点和线的光栅化（rasterization），等等。OSG 的渲染状态也允许应用程序指定顶点着色

（vertex shader）和片段着色（fragment shader）。

用户的应用程序需要在 osg::StateSet 中设置渲染状态。用户可以将 StateSet 关联到场景图形中的任意一个节点（Node），或者关联到 Drawable 类。正如大部分开发者所知，OpenGL 程序的开发需要尽量使状态量的变化实现最小化，并避免冗余的状态设置；StateSet 对象能够自动实现这些优化过程。

在 OSG 遍历整个场景图形时，StateSet 类会对 OpenGL 的状态属性堆栈进行管理。因此，用户程序可以对不同的场景图形子树作不同的状态设置。在每个子树的遍历过程中，OSG 将会高效地执行保存和恢复渲染状态的操作。

用户需要尽量使关联到场景图形的 StateSet 最少化。StateSet 越少，内存的占用也越少，OSG 在一次场景图形遍历中所耗费的工作量也越少。StateSet 类派生自 Referenced，以实现更好的数据共享。也就是说，共享同一个 StateSet 的 Node 或 Drawable 类不需要额外的代码来清理其内存空间。

OSG 将渲染状态分成两个部分，渲染属性（attribute）和渲染模式（mode）。渲染属性也就是控制渲染特性的状态变量。例如，雾的颜色，或者 Blend 融合函数都是 OSG 的状态属性。OSG 中的渲染模式和 OpenGL 的状态特性几乎是一一对应的，这些特性在 OpenGL 中通过函数 glEnable()和 glDisable()进行控制。用户程序可以设置模式量以允许或禁止某个功能，例如纹理映射、灯光等。简单来说，渲染模式是指渲染的某个功能，而渲染属性是这个功能的控制变量和参数。

如果要设置渲染状态的值，用户程序需要执行以下几步操作：

- 为将要设置状态的 Node 或 Drawable 对象提供一个 StateSet 实例。
- 在 StateSet 实例中设置状态的渲染模式和渲染属性。

要从某个 Node 或 Drawable 对象中直接获得一个 StateSet 实例，使用下面的方法：

osg::StateSet* state = obj->getOrCreateStateSet();

在上面的程序段中，obj 是一个 Node 或 Drawable 类实例；getOrCreateStateSet()是这些类定义的方法。这个方法返回一个指向 StateSet 实例的指针，该实例属于 obj。如果 obj 还没有设置过与之关联的 StateSet，那么这个方法返回一个新指针并将其关联到 obj 上。

StateSet 继承自 Referenced 类。与 StateSet 关联的 Node 或 Drawable 类内部使用 ref_ptr<>来引用 StateSet 实例，因此，不是长时间引用 StateSet 的情况下，也可以使用标准 C++指针来定义 state。如果 state 变量是某个函数内的局部变量，且应用程序不会长时间引用这个 StateSet，那么上述代码的使用是完全正确的。

上面代码中的 state 变量是指向该 obj 对象的 StateSet 指针。当应用程序获得了一个 StateSet 的指针时，就可以设置属性和模式。以下的部分将对此详加叙述，并提供相应的例子。

2.4.1 渲染属性（Attribute）和渲染模式（Mode）

OSG 为每个状态属性定义了不同的类，以便应用程序采用。所有的属性类均继承自 osg::StateAttribute，StateAttribute 类是一个无法直接实例化的虚基类。

继承自 StateAttribute 的类有数十个之多，本书将对其中的一些属性类作简要阐述，并对灯光和纹理映射的属性作更详细的讲解。对所有属性类的详尽讲解不在本书的范畴之内。如果你希望作更深入的了解，可以阅读 OSG 开发环境下 include/osg 目录的头文件代码，并在其中搜索派生自 StateAttribute 的类即可。

OSG 将所有的属性和模式分为两大部分：纹理（texture）和非纹理（non-texture）。本节将主要探讨非纹理渲染状态的设置。纹理渲染状态的设置将在第 2.4.4 节“纹理映射”中讨论。OSG 之所以为纹理属性的设置提供不同的接口，主要是因为纹理属性需要特别为多重纹理设置纹理单元（texture unit）。

设置渲染属性（Attribute）

如果要设置一项属性，首先将要修改的属性类实例化。设置该类的数值，然后用 osg::StateSet::setAttribute()将其关联到 StateSet。下面的代码段用于实现面剔除（face culling）的属性：

```
// 获取变量 geom 的 StateSet 指针。
osg::StateSet* state = geom->getOrCreateStateSet();
// 创建并添加 CullFace 属性类。
```

```
osg::CullFace* cf = new osg::CullFace( osg::CullFace::BACK );
state->setAttribute( cf );
```

在上面的代码段中，geom 是一个 Geometry 几何类对象（当然也可以是任何其它派生自 Drawable 和 Node 的对象）。获取 geom 的 StateSet 指针后，代码创建了一个新的 osg::CullFace 对象，并将其关联到状态量。

CullFace 是一个属性类，派生自 StateAttribute 类。它的构造函数有一个参数：一个用于指定剔除表面是正面还是反面的枚举量，可选的值有：FRONT，BACK，FRONT_AND_BACK。这些枚举量在 CullFace 的头文件中定义，它们等价于 OpenGL 的枚举量 GL_FRONT，GL_BACK 和 GL_FRONT_AND_BACK。

如果你对 OpenGL 足够熟悉，那么你可以把上面的代码理解成调用 glCullFace(GL_BACK)。不过，请记住 OSG 是一个场景图形系统。当应用程序将 CullFace 属性关联到一个 StateSet 时，只是记录了用户的请求，而不是直接向 OpenGL 发送命令。在绘图遍历（draw traversal）中，OSG 将跟踪状态数据的变化，并在必要的时候发送命令 glCullFace()到 OpenGL。

与大多数 OSG 对象相同，StateAttribute 也是继承自 Referenced。当你将派生自 StateAttribute 的属性对象实例化并将其关联到 StateSet 时，StateSet 将对这个属性实施引用计数，不用担心它所占用的内存如何释放的问题。

在代码的编写符合规范的前提下，用户可以将 StateAttribute 临时赋予一个标准 C++指针，而 StateAttribute 关联到 StateSet 之后，StateSet 自带的 ref_ptr<>将接管并负责内存的管理。

设置渲染模式（Mode）

用户可以使用 osg::StateSet::setMode()允许或禁止某种模式。例如，下面的代码将打开雾效模式的许可：

```
// 获取一个 StateSet 实例。
osg::StateSet* state = geom->getOrCreateStateSet();
// 允许这个 StateSet 的雾效模式。
state->setMode( GL_FOG, osg::StateAttribute::ON );
```

setMode()的第一个输入参数可以是任何一个在 glEnable()或 glDisable()中合法的 OpenGL 枚举量 GLenum。第二个输入参数可以是 osg::StateAttribute::ON 或 osg::StateAttribute::OFF。事实上，这里用到了位屏蔽的技术，在 2.4.2 节“状态继承”中将继续予以讨论。

设置渲染属性和模式

OSG 提供了一个简单的可以同时设置属性和模式的单一函数接口。在许多情况下，属性和模式之间都存在显著的关系。例如，CullFace 属性的对应模式为 GL_CULL_FACE。如果要将某个属性关联到一个 StateSet，同时打开其对应模式的许可，那么可以使用 osg::StateSet::setAttributeAndModes()方法。下面的代码段将关联 Blend 融合检验的属性，同时许可颜色融合模式。

```
// 创建一个 BlendFunc 属性。
osg::BlendFunc* bf = new osg::BlendFunc();
// 关联 BlendFunc 并许可颜色融合模式
state->setAttributeAndMode( bf );
```

setAttributeAndModes()的第二个输入参数用于允许或禁止第一个参数中渲染属性对应的渲染模式。其缺省值为 ON。这样用户的应用程序只需用一个函数，就可以方便地指定某个渲染属性，并许可其对应的渲染模式。

2.4.2 状态继承

当你设置节点的渲染状态时，这个状态将被赋予当前的节点及其子节点。如果子节点对同一个渲染状态设置了不同的属性参数，那么新的子节点状态参数将会覆盖原有的。换句话说，缺省情况下子节点可以改变自身的某个状态参数，或者继承父节点的同一个状态。图 2-7 表现了这个概念的实现过程。

图 2-7 状态继承

在场景图形中，根节点许可了光照模式。它的第一个子节点禁止了光照，即覆盖了父节点传递下来的的光照状态。因此 OSG 将禁止光照以渲染第一个子节点。第二个子节点没有改变渲染状态。因此，OSG 将沿用父节点的渲染状态来渲染第二个子节点，此节点的光照模式被允许。

这种继承的特性在许多情况下都非常实用。但是有时候渲染可能需要更多特性。假设场景图形中有一个包含了实体多边形几何体的节点。如果要以线框模式来渲染场景图形，你的程序就需要覆盖这种多边形渲染模式状态，不论它出现在什么位置。

OSG 允许用户根据场景图形中任意位置的渲染属性和模式需求，而单独改变原有的状态继承特性。用户可以选择以下这几种枚举形式：

- osg::StateAttribute::OVERRIDE - 如果你将一个渲染属性和模式设置为 OVERRIDE，那么所有的子节点都将继承这一属性或模式，子节点对它们更改将会无效。
- osg::StateAttribute::PROTECTED - 这种形式可以视为 OVERRIDE 的一个例外。凡是设置为 PROTECTED 的渲染属性或模式，均不会受到父节点的影响。
- osg::StateAttribute::INHERIT - 这种模式强制子节点继承父节点的渲染状态。其效果是子节点的渲染状态被解除，而使用父节点的状态替代。

你可以对这些参数进行位或叠加的操作，然后再作为 setAttribute()，setMode()

和 setAttributeAndModes()的第二个参数输入。下面的代码段将强制使用线框模式渲染场景图形。

```
// 获取根节点的渲染状态 StateSet。
osg::StateSet* state = root->getOrCreateStateSet();
// 创建一个 PolygonMode 渲染属性。
osg::PolygonMode* pm = new osg::PolygonMode)
osg::PolygonMode::FRONT_AND_BACK, osg::PolygonMode::LINE );
// 强制使用线框渲染。
state->setAttributeAndModes( pm,
        osg::StateAttribute::ON | osg::StateAttribute::OVERRIDE );
```

使用 PROTECTED 参量可以保证父节点的渲染状态不会覆盖子节点的渲染状态。举例来说，你可能创建了一个发光的场景，其中包含有使用亮度照明的光源几何体。如果其父节点禁用了光照，那么光源几何体的渲染将会出错。这个时候，对光源几何体的 GL_LIGHTING 渲染状态使用 PROTECTED，就可以保证它依然可用。

2.4.3 渲染状态设置示例

第 2.3.2 节“父接口”描述了将同一节点添加为多个节点的子节点的方法。“矩阵变换节点（MatrixTransform）”这一节则介绍了使用 MatrixTransform 节点来变换几何体的方法。第 2.4 节“渲染状态”着重讲述了 OSG 的渲染状态。下面的示例代码将会融合以上所有的概念。

本节将会提供一个修改 OSG 渲染状态的简单例子。代码中创建了一个 Geode 节点，其中添加了包含两个四边形的 Drawable 类实例，这个节点有四个渲染状态各不相同的 MatrixTransform 父节点。图 2-8 显示了这一场景图形，代码清单 2-3 列出了用于创建节点的程序段，此程序出自本书附带源代码的相关例子。

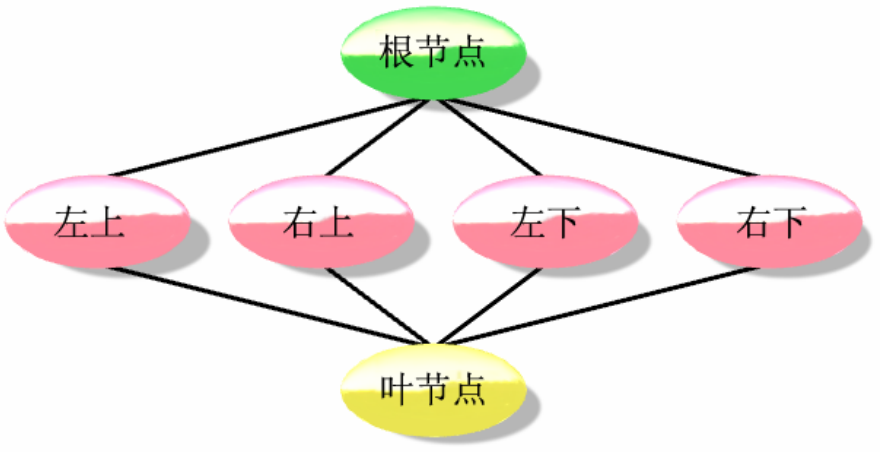


图 2-8 渲染状态示例程序的场景图形

这个例子将同一个 Geode 渲染了四次。它使用四个 MatrixTransform 节点来放置这个 Geode,每个 MatrixTransform 节点都有自己的 StateSet。

清单 2-3 修改渲染状态

附带的示例代码中已经向一个 Geode 添加了多个 Drawable 实例。这里为每个 Drawable 设置不同的渲染状态，并通过修改 Geode 的相关属性来禁止所有几何体的光照。

```
#include <osg/Geode>
#include <osg/Group>
#include <osg/MatrixTransform>
#include <osg/Geode>
#include <osg/Geometry>
#include <osg/StateSet>
#include <osg/StateAttribute>
#include <osg/ShadeModel>
#include <osg/CullFace>
#include <osg/PolygonMode>
#include <osg/LineWidth>
...
```

```
osg::ref_ptr<osg::Node> createSceneGraph()
{
    // 创建 Group 根节点。
    osg::ref_ptr<osg::Group> root = new osg::Group;
    {
        // 在根节点的 StateSet 中禁止光照。
        // 使用 PROTECTED 以保证这一修改不会被 osgviewer 覆盖。
        osg::StateSet* state = root->getOrCreateStateSet();
        state->setMode( GL_LIGHTING,
                        osg::StateAttribute::OFF |
                        osg::StateAttribute::PROTECTED );
    }

    // 创建 Geode 叶节点并关联 Drawable。
    osg::ref_ptr<osg::Geode> geode = new osg::Geode;
    geode->addDrawable( createDrawable().get() );
    osg::Matrix m;
    {
        // 左上角：使用缺省属性渲染几何体。
        osg::ref_ptr<osg::MatrixTransform> mt =
                new osg::MatrixTransform;
        m.makeTranslate( -2.f, 0.f, 2.f );
        mt->setMatrix( m );
        root->addChild( mt.get() );
        mt->addChild( geode.get() );
    }
    {
        // 右上角：设置着色模式为 FLAT（单色）。
        osg::ref_ptr<osg::MatrixTransform> mt =
                new osg::MatrixTransform;
```

```
    m.makeTranslate( 2.f, 0.f, 2.f );
    mt->setMatrix( m );
    root->addChild( mt.get() );
    mt->addChild( geode.get() );
    osg::StateSet* state = mt->getOrCreateStateSet();
    osg::ShadeModel* sm = new osg::ShadeModel();
    sm->setMode( osg::ShadeModel::FLAT );
    state->setAttribute( sm );
}
{
    // 左下角：开启背面剔除。
    osg::ref_ptr<osg::MatrixTransform> mt =
            new osg::MatrixTransform;
    m.makeTranslate( -2.f, 0.f, -2.f );
    mt->setMatrix( m );
    root->addChild( mt.get() );
    mt->addChild( geode.get() );
    osg::StateSet* state = mt->getOrCreateStateSet();
    osg::CullFace* cf = new osg::CullFace(); // 缺省值为 BACK
    state->setAttributeAndModes( cf );
}
{
    // 右下角：设置多边形填充模式为 LINE（线框）。
    osg::ref_ptr<osg::MatrixTransform> mt =
            new osg::MatrixTransform;
    m.makeTranslate( 2.f, 0.f, -2.f );
    mt->setMatrix( m );
    root->addChild( mt.get() );
    mt->addChild( geode.get() );
    osg::StateSet* state = mt->getOrCreateStateSet();
```

```
        osg::PolygonMode* pm = new osg::PolygonMode(
            osg::PolygonMode::FRONT_AND_BACK,
            osg::PolygonMode::LINE );
        state->setAttributeAndModes( pm );
        // 同时还设置线宽为 3。
        osg::LineWidth* lw = new osg::LineWidth( 3.f );
        state->setAttribute( lw );
    }
    return root.get();
}
```

渲染状态的示例代码建立了一个名为 root 的 Group 组节点，作为整个场景图形的根节点，并设置根节点的渲染状态 StateSet 为禁止光照。此处使用 PROTECTED 标识以避免受到 osgviewer 的影响。

此处的代码使用了一个名为 createDrawable()的函数来创建一个几何对象，该对象包含了两个四边形，其顶点皆采用了不同的颜色。清单 2-3 中并未包含 createDrawable()的内容。请下载附带代码以查看这个函数。也许你已经想到了，这个函数的内容与清单 2-1 有些类似。而本段代码中，将把返回的 Drawable 实例关联到新的 Geode 类实例 geode 上。

- 第一个 MatrixTransform 实例使用缺省的 StateSet，将 geode 移动到左上角。geode 继承了父类的所有渲染状态，在这里它使用的是缺省状态。
- 第二个 MatrixTransform 实例为 StateSet 设置了一个属性 ShadeModel，赋值为 FLAT，并将 geode 移动到右上角。这样系统将使用单一着色渲染的方式渲染四边形。其颜色取决于最后一个顶点的颜色。
- 第二个 MatrixTransform 实例为 StateSet 设置了一个属性 CullFace，并将 geode 移动到左下角。缺省情况下，通过调用构造函数，CullFace 会剔除多边形的背面（BACK）。使用 setAttributeAndModes(cf)可以关联 CullFace 并且打开 GL_CULL_FACE 模式的许可。（createDrawable()返回的两个四边形有着相反的顶点时针顺序，因此无论视点的位置如何，都

会有一个背面存在。）

- 最后一个 MatrixTransform 实例将 geode 移动到右下角，它的 StateSet 具有两个渲染属性，PolygonMode 和 LineWidth。此处的代码设置多边形填充模式为，正反面（FRONT_AND_BACK）皆为线框模式（LINE），且设置线宽为 3.0。

与前面的示例相同，本处的例子将场景图形写入了文件 State.osg。运行这个例子之后，可以使用下面的命令在 osgviewer 中显示图像。

osgviewer State.osg

图 2-9 所示为 osgviewer 中的场景内容。

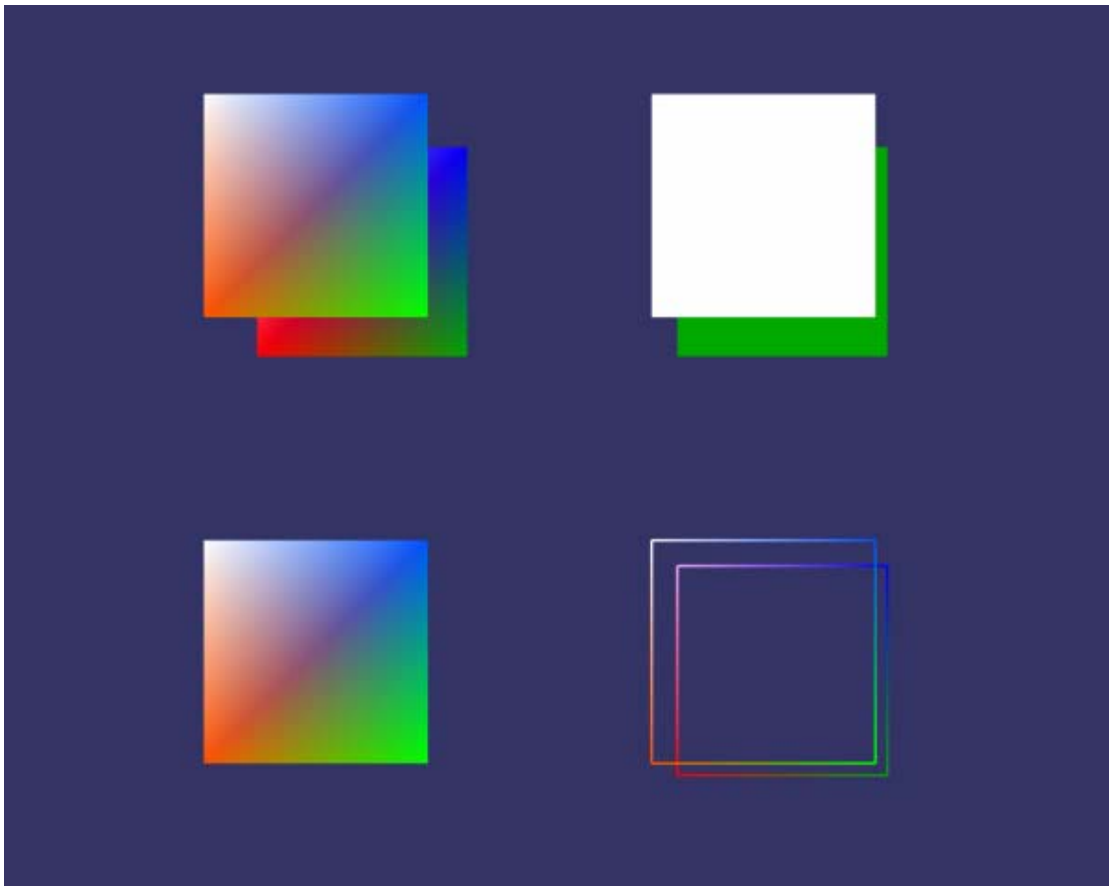


图 2-9 osgviewer 中显示的渲染状态示例场景图形

清单 2-3 中生成的场景图形的渲染演示了不同的渲染属性和模式的运用。左上：缺省状态。右上：着色模式设为 FLAT。左下：允许背面的面剔除。右下：设置多边形模式为 LINE 且线宽为 3.0。整个场景的光照被禁止。

2.4.4 纹理映射

OSG 全面支持 OpenGL 的纹理映射机制。为了在程序中实现基本的纹理映射功能，用户的代码需要遵循以下的步骤：

- 指定用户几何体的纹理坐标。
- 创建纹理属性对象并保存纹理图形数据。
- 为 StateSet 设置合适的纹理属性和模式。

本节将针对上面的每一步分别加以讨论。在本书的示例代码中，例子程序 TextureMapping 展示了基本的纹理映射技术。为了节省空间，本文将不再罗列代码。

纹理坐标

第 2.2 节“叶节点（Geode）和几何信息”介绍了 Geometry 对象中有关设置顶点，法线，颜色数据的接口。Geometry 类同时还允许用户程序指定一个或多个纹理坐标数据数组。当你指定了纹理坐标之后，还要指定相应的纹理单元，OSG 使用纹理单元的值来实现多重纹理。

多重纹理（Multitexture）：

OpenGL 的早期版本并不支持多重纹理。而加入多重纹理的特性之后，OpenGL 仍然支持非多重纹理的函数接口，以实现向下兼容。从本质上说，此时 OpenGL 将非多重纹理接口解释为，使用纹理单元 0 对应所有纹理数据。

与 OpenGL 不同，OSG 并不支持非多重纹理接口。因此，用户程序必须指定一个纹理单元，以对应纹理坐标数据和纹理状态。如果要使用单一纹理的话，只需要指定到纹理单元 0 即可。

下面的代码段创建了一个 osg::Vec2Array 数组，用于保存纹理坐标，同时将其关联到 Geometry 实例的纹理单元 0。如果要对单一的 Geometry 设置多个纹理，只需要将多个纹理坐标数组关联到 Geometry，并针对不同的数组指定不同的纹理单元即可。

```
// 创建一个 Geometry 几何体对象。
osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
// 创建一个 Vec2Array 对象以保存纹理单元 0 的纹理坐标，
// 并将其关联到 geom。
osg::ref_ptr<osg::Vec2Array> tc = new osg::Vec2Array;
geom->setTexCoordArray( 0, tc.get() );
tc->push_back( osg::Vec2( 0.f, 0.f ) );
tc->push_back( osg::Vec2( 1.f, 0.f ) );
tc->push_back( osg::Vec2( 1.f, 1.f ) );
tc->push_back( osg::Vec2( 0.f, 1.f ) );
```

osg::Geometry::setTexCoordArray()的第一个输入参数是纹理单元号，第二个参数是纹理坐标数组。用户不需要使用类同 osg::Geometry::setTexCoordBinding()的函数输入点来绑定纹理数据。纹理坐标总是绑定到每个顶点的。

读取图像

用户程序可以使用两个类来实现基本的 2D 纹理映射：osg::Texture2D 和 osg::Image。Texture2D 属于 StateAttribute 的派生类，用于管理 OpenGL 纹理对象，而 Image 用于管理图像像素数据。如果要使用 2D 图像文件作为纹理映射的图形，只要将文件名赋给 Image 对象并将 Image 关联到 Texture2D 即可。下面的代码段将文件 sun.tif 作为纹理映射的图形。

```
#include <osg/Texture2D>
#include <osg/Image>
...
osg::StateSet* state = node->getOrCreateStateSet();
// 读取纹理图像。
osg::ref_ptr<osg::Image> image = new osg::Image;
image->setFileName( "sun.tif" );
```

```
// 将图像关联到 Texture2D 对象
osg::ref_ptr<osg::Texture2D> tex = new osg::Texture2D;
tex->setImage( image.get() );
```

Texture2D 属性设置完毕后，可以将其关联到 StateSet。下一节“纹理状态”将对这一步骤作更详细的讲解。

大量使用纹理贴图的程序，往往需要实现更紧凑的内存管理。Image 类继承自 Referenced 类，而 Texture2D 内部保存了一个指向 Image 的 ref_ptr<>指针。在第一次渲染时，OSG 创建了用于保存图像数据的 OpenGL 纹理对象，其结果是产生了两个纹理图像的副本：一个是 Image 对象，另一个由 OpenGL 拥有。简单的单环境（single-context）场景渲染中，你可以通过设置 Texture2D 解除对 Image 的引用来降低内存损耗。如果当前引用 Image 对象的只有 Texture2D 对象，那么 OSG 将释放 Image 及其内存空间。下面的代码演示了设置 Texture2D 解除对 Image 引用的方法：

```
// 创建 OpenGL 纹理对象后，释放内部的 ref_ptr<Image>，
// 删除 Image 图像。
tex->setUnRefImageDataAfterApply( true );
```

缺省情况下，Texture2D 不会自动释放对 Image 的引用。在多环境（multi-context）场景渲染中，这是一种期望行为，前提是纹理对象并没有在各环境中共享。

纹理状态

用户程序可以使用纹理状态函数接口为每个纹理单元指定渲染状态。纹理状态函数接口与非纹理状态的接口十分类似。用户可以使用 osg::StateSet::setTextureAttribute() 将一个纹理属性关联到 StateSet 对象。setTextureAttribute() 的第一个参数是纹理单元，第二个参数是继承自 StateAttribute 类的一种纹理属性。合法的纹理属性类共有六种，其中包括五种纹理类型（osg::Texture1D，osg::Texture2D，osg::Texture3D，osg::TextureCubeMap

和 osg::TextureRectangle），另一个类用于纹理坐标的生成（osg::TexGen）。

下面的代码将根据给定的 Texture2D 属性对象 tex 和渲染状态 StateSet，将 tex 关联到渲染状态，并设置使用纹理单元 0：

```
// 创建一个 Texture2D 属性。
Osg::ref_ptr<osg::Texture2D> tex = new osg::Texture2D;
// ...
// 关联材质属性到材质单元 0。
state->setTextureAttribute( 0, tex.get() );
```

与此类似，用户可以调用 osg::StateSet::setTextureMode()来设置材质渲染模式。这个方法与 setMode()方法类似。用户可以使用 setTextureMode()来设置以下模式：GL_TEXTURE_1D，GL_TEXTURE_2D，GL_TEXTURE_3D，GL_TEXTURE_CUBE_MAP，GL_TEXTURE_RECTANGLE，GL_TEXTURE_GEN_Q，GL_TEXTURE_GEN_R，GL_TEXTURE_GEN_S，以及 GL_TEXTURE_GEN_T。

与 setTextureAttribute()相似，setTextureMode()的第一个参数表示纹理单元。下面的代码段将禁止纹理单元 1 的 2D 纹理映射。

```
state->setTextureMode( 1, GL_TEXTURE_2D, osg::StateAttribute::OFF );
```

当然，用户也可以使用 osg::StateSet::setTextureAttributesAndModes()来关联纹理渲染属性到 StateSet，同时允许相应的纹理模式。如果属性是一个 TexGen 对象，那么 setTextureAttributesAndModes()将设置相应的坐标生成模式 GL_TEXTURE_GEN_Q，GL_TEXTURE_GEN_R，GL_TEXTURE_GEN_S 和 GL_TEXTURE_GEN_T。对于其它纹理属性来说，这一模式是隐含的。例如，下面的代码中，由于第二个参数传入了一个 Texture2D 对象作为纹理属性，setTextureAttributesAndModes()将允许 GL_TEXTURE_2D 模式。

```
// 创建一个 Texture2D 属性对象。
osg::ref_ptr<osg::Texture2D> tex = new osg::Texture2D;
```

```
// ...
// 在纹理单元 0 上，关联 2D 纹理属性并许可 GL_TEXTURE_2D 模式。
state->setTextureAttributeAndModes( 0, tex );
```

setTextureAttributeAndModes()的第三个参数缺省值为 ON，即允许纹理渲染模式。与 setAttributeAndModes()类似，你可以对这个参数使用位或操作，以修改纹理属性的继承特性，包括 OVERRIDE，PROTECTED 和 INHERIT。你还可以通过修改 setTextureMode()和 setTextureAttribute()的第三个参数来指定这个继承标志。

2.4.5 光照

OSG 全面支持 OpenGL 的光照特性，包括材质属性（material property），光照属性（light property）和光照模型（lighting model）。与 OpenGL 相似，OSG 中的光源也是不可见的，而非渲染一个灯泡或其他自然形状。同样，光源会创建着色效果，但是并不创建阴影——osgShadow 可以用来创建阴影。

如果要在用户程序中使用光照，需要遵循下面的步骤：

- 指定几何体法线。
- 允许光照并设置光照状态。
- 指定光源属性并关联到场景图形。
- 指定表面材质属性。

本节将针对上面的每一步进行讲解。

法线

只有几何体数据中设有单位长度法线时，才可以实现正确的光照。第 2.2 节“叶节点（Geode）和几何信息”已经对法线数组的设置和绑定到 Geometry 对象作了介绍。

在大多数 3D API 中，法线数据必须单位化以得到正确的结果。注意缩放变换的动作会改变法线的长度。如果你的 Geometry 对象中已经包含了单位长度的

法线数组，但是光照的计算结果过于明亮或过于暗淡，那么这一现象可能是缩放变换造成的。最有效的解决方案是在 StateSet 中允许法线的重放缩模式。

```
osg::StateSet* state = geode->setOrCreateStateSet();
state->setMode( GL_RESCALE_NORMAL, osg::StateAttribute::ON );
```

与 OpenGL 中相同，这一特性可以保证法线在均匀放缩变换时仍然保持单位长度。如果场景中的放缩变换是非均匀的，那么你可以允许法线归一化模式，以保证法线为单位长度。

```
osg::StateSet* state = geode->setOrCreateStateSet();
state->setMode( GL_NORMALIZE, osg::StateAttribute::ON );
```

由于要进行法线的重新放缩，归一化模式往往会耗费大量的时间。编程时要尽量避免。

光照状态

要在 OSG 中获得光照效果，你需要允许光照并至少允许一个光源。程序 osgviewer 在缺省情况下就是这样做的，它在根节点的 StateSet 中已经设置了相应的模式。你可以在自己的程序中进行相同的设置。下面的代码段用于允许光照并为根节点的 StateSet 允许两个光源（GL_LIGHT0 和 GL_LIGHT1）。

```
osg::StateSet* state = root->getOrCreateStateSet();
state->setMode( GL_LIGHTING, osg::StateAttribute::ON );
state->setMode( GL_LIGHT0, osg::StateAttribute::ON );
state->setMode( GL_LIGHT1, osg::StateAttribute::ON );
```

下面的部分将叙述如何控制独立的光源属性，例如它的位置和颜色，以及 OpenGL 颜色跟踪材质的特性（包括便面材质颜色的设置）。

OSG 还提供了从 StateAttribute 派生的 osg::LightModel 属性，用以控制全局的环境颜色，局部视图，双面光照，以及分离镜面颜色（separate specular color）等 OpenGL 特性。

光源

要在场景中添加一个光源，可以创建一个 osg::Light 对象以定义光源参数。然后将 Light 添加到一个 osg::LightSource 节点中，并将 LightSource 节点添加到场景图形。LightSource 是一个包含了唯一的 Light 定义的高效的组节点。而由 Light 定义的光源将对整个场景产生影响。

OSG 支持最多八个光源，从 GL_LIGHT0 到 GL_LIGHT7，这与你的 OpenGL 版本也有关系。你可以使用前述的 setMode()方法来允许这些光源。如果要把一个 Light 对象与 OpenGL 的光源联系起来，可以使用设置光的位置数的方法。例如要把一个 Light 对象与 GL_LIGHT2 相关联，则设置位置数为 2：

```
// 创建一个 Light 对象来控制 GL_LIGHT2 的参数。
osg::ref_ptr<osg::Light> light = new osg::Light;
light->setLightNum( 2 );
```

缺省情况下光源的位置数为 0。

Light 类实现了 OpenGL 中 glLight()命令的大部分功能。用户程序可以使用其方法设置光的环境色，散射颜色，镜面反射颜色。用户可以创建点光，直线光或者锥光，也可以指定衰减参数，使得光的密度根据距离的不同逐渐削减。下面的代码创建了一个 Light 对象并设置了一些常用的参数。

```
// 创建一个白色的锥光光源。
osg::ref_ptr<osg::Light> light = new osg::Light;
light->setAmbient( osg::Vec4( .1f, .1f, .1f, 1.f ));
light->setDiffuse( osg::Vec4( .8f, .8f, .8f, 1.f ));
light->setSpecular( osg::Vec4( .8f, .8f, .8f, 1.f ));
```

```
light->setPosition( osg::Vec3( 0.f, 0.f, 0.f ));
light->setDirection( osg::Vec3( 1.f, 0.f, 0.f ));
light->setSpotCutoff(25.f );
```

位置状态：

当一个 OpenGL 程序使用 glLight()来设置光的位置时，OpenGL 根据当前的模型-视图（model-view）矩阵来变换位置。在 OSG 中，这一概念被称为位置状态（positional state）。在拣选（cull）遍历中，OSG 向一个位置状态容器中追加各种位置状态量，以保证在绘制（draw）遍历中所有的变换都是正确的。

警告：

许多新的 OSG 开发师会错误地以为 LightSource 的子节点会自动设置发光。事实上并非如此。OSG 根据场景图形中当前的渲染状态来实现光照，而非根据 LightSource 节点的层次关系。用户必须允许 GL_LIGHTING 和至少一个光源（例如，GL_LIGHT0），才能实现场景图形的照明。

用户可以把 LightSource 看作是包含了一个单一 Light 对象的叶节点。但是，用户也可以向 LightSource 节点添加子节点，因为 LightSource 继承自 Group 类。一般来说，用户程序可以将用于渲染灯光的自然形体的几何体关联为 LightSource 的子节点。

要添加 Light 对象到场景中，首先要创建一个 LightSource 节点，将 Light 添加到 LightSource 中，并将 LightSource 关联到场景图形中。灯光的位置可以由 LightSource 节点在场景图形中的位置决定。OSG 根据当前 LightSource 节点的变换状态来改变灯光的位置。

OSG 程序开发师往往将 LightSource 关联为 MatrixTransform 的子节点，以控制灯光的位置，如下面的代码所示：

```
// 创建灯光并设置其属性。
osg::ref_ptr<osg::Light> light = new osg::Light;
// ...
// 创建 MatrixTransform 节点，以决定灯光的位置。
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
osg::Matrix m;
m.makeTranslate( osg::Vec3( -3.f, 2.f, 5.f ) );
mt->setMatrix( m );

// 添加 Light 对象到 LightSource。
// 然后添加 LightSource 和 MatrixTransform 到场景图形。
osg::ref_ptr<osg::LightSource> ls = new osg::LightSource;
parent->addChild( mt.get() );
mt->addChild( ls.get() );
ls->setLight( light.get() );
```

OSG 在缺省情况下根据 LightSource 的当前变换位置来确定灯光的位置。你也可以通过设置 LightSource 的参考系来禁止这一特性。这与第 2.3.3 节“变换节点（Transform）”中所述的 Transform 节点参考系设置方法相同。下面的代码将使 OSG 忽略 LightSource 的当前变换，而是把灯光的位置作为一个绝对值来看待。

```
osg::ref_ptr<osg::LightSource> ls = new osg::LightSource;
ls->setReferenceFrame( osg::LightSource::ABSOLUTE_RF );
```

材质属性

状态属性类 osg::Material 封装了 OpenGL 的 glMaterial()和 glColorMaterial()指令的函数功能。要在用户应用程序中设置材质属性，首先要创建一个 Material 对象，设置颜色和其它参数，然后关联到场景图形的 StateSet 中。

Material 类允许用户设置全局、散射、镜面反射和放射材质的颜色，以及镜面和高光指数参数。Material 类定义了枚举量 FRONT，BACK 和 FRONT_AND_BACK，以便用户程序为几何体的正面和背面的设置材质属性。举例来说，下面的代码为一个几何图元的正面设置了散射颜色，镜面反射颜色和镜面指数的参数。

```
osg::StateSet* state = node->getOrCreateStateSet();
osg::ref_ptr<osg::Material> mat = new osg::Material;
mat->setDiffuse( osg::Material::FRONT,
osg::Vec4( .2f, .9f, .9f, 1.f ) );
mat->setSpecular( osg::Material::FRONT,
osg::Vec4( 1.f, 1.f, 1.f, 1.f ) );
mat->setShininess( osg::Material::FRONT, 96.f );
state->setAttribute( mat.get() );
```

与 OpenGL 类似，此处镜面指数的值必须在 1.0 到 128.0 至间，除非用户指定了拓宽这一范围的方法。

在进行很多 OpenGL 的操作时，直接设置材质属性可能会过于耗费资源。而一种名为颜色跟踪材质（color material）的特性，可以允许用户程序通过改变当前颜色的方法，自动改变某一特定的材质属性。在许多情形下，这一操作比直接修改材质属性的效率要更高，还能加强光照场景和无光照场景的联系，并满足应用程序对材质的需要。

要允许颜色跟踪材质的特性，需要调用 Material::setColorMode()方法。Material 类为之定义了枚举量 AMBIENT，DIFFUSE，SPECULAR，EMISSION，AMBIENT_AND_DIFFUSE 以及 OFF。缺省情况下，颜色跟踪模式被设置为 OFF，颜色跟踪材质被禁止。如果用户程序设置颜色跟踪模式为其它的值，那么 OSG 将为特定的材质属性开启颜色跟踪材质特性，此时主颜色的改变将会改变相应的材质属性。下面的代码段将允许颜色跟踪材质，此时几何体正面的环境材质和散射材质颜色将自动跟踪当前颜色的改变而改变。

```
osg::StateSet* state = node->getOrCreateStateSet();
osg::ref_ptr<osg::Material> mat = new osg::Material;
mat->setColorMode( osg::Material::AMBIENT_AND_DIFFUSE );
state->setAttribute( mat.get() );
```

注意，根据颜色跟踪模式的取值不同，Material 类会自动允许或禁止 GL_COLOR_MATERIAL。因此用户程序不需要调用 setAttributeAndModes()来允许或禁止相关的模式值。

光照示例

本书源代码附带的光照示例中，创建了两个光源，并使用七种不同的材质属性渲染几何体。节省空间起见，源代码在这里并没有列出。源代码将生成文件 Lighting.osg 并在其中写入场景图形。用户可以使用下面的命令显示场景图形：

```
osgviewer Lighting.osg
```

图 2-10 所示为 osgviewer 中显示的场景图形。

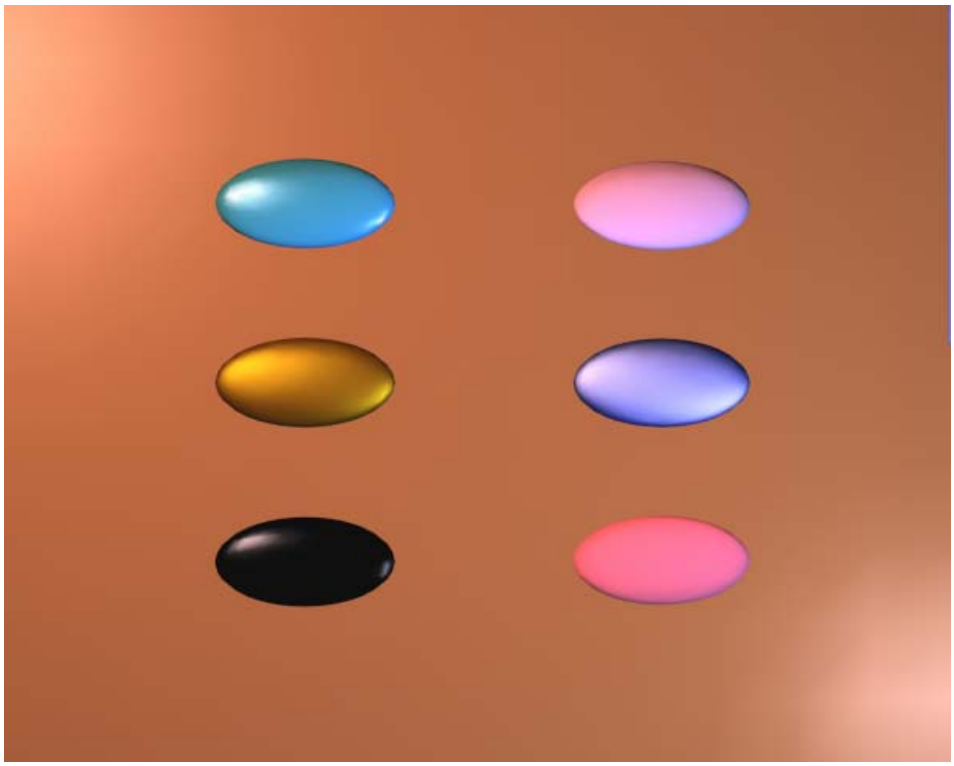


图 2-10 osgviewer 中显示的光照示例场景图形

这个例子渲染了六个菱形几何对象和一个背景平面。每个几何对象均有各自不同的材质设置。本场景中有两个光源负责照明。

2.5 文件 I/O

第二章所介绍的是创建含有几何体和渲染状态的场景功能图形的编程技术，大多数应用程序也确实可以手动编程来创建一些几何对象。但是，应用程序通常需要从文件中读取并显示大的，复杂的模型。这就需要使用一个函数来读取文件中的模型，并返回预设的场景图形信息。

osgDB 库提供了用户程序读取和写入 2D 图像和 3D 模型文件的函数接口。osgDB 库还负责管理 OSG 的插件系统，以实现对不同文件格式的支持。第 1.6.3 节“组件”已经对插件的概念作了简介，而图 1-9 所示为插件与整个 OSG 体系的嵌入方式。

本章所述的例子均使用 osgDB 来实现文件的 I/O。所有的例子均使用.osg 插

件将场景图形写入一个.osg 文件。光照示例中使用.osg 插件从名为 lozenge.osg 的文件中读取了子图形信息，而纹理映射示例中使用.png 插件读取纹理图像。但是，前面的文字并没有介绍文件 I/O 功能的用法。本节将更详细地阐述插件的使用，以便用户可以在自己的程序中高效地使用它们。本文将介绍读取和写入文件的函数接口，OSG 搜索文件的方式，以及 OSG 选择相应的插件以读取文件的方式。

2.5.1 接口

osgDB 提供了文件 I/O 的函数接口，从而将插件系统完全隐藏在用户程序之下。用户需要两个 osgDB 的头文件来定义这个接口：

#include <osgDB/ReadFile>

#include <osgDB/WriteFile>

用户程序如果要使用 osgDB 的文件 I/O 功能，可以在源代码中包含这两个头文件。在其中的 osgDB 命名空间里定义了一系列用于实现文件 I/O 的函数。

读取文件

使用函数 osgDB::readNodeFile()和 osgDB::readImageFile()来读取 3D 模型和 2D 图像文件。

osg::Node* osgDB::readNodeFile(const std::string& filename);

osg::Image* osgDB::readImageFile(const std::string& filename);

用户可以使用 readNodeFile()读取 3D 模型文件。OSG 将根据文件的扩展名来识别文件类型，并使用相应的插件转换文件到场景图形中。readNodeFile()返回一个指向场景图形节点的指针。同样，readImageFile()可以读取 2D 图像文件并返回指向 Image 对象的指针。文件名参数中可以包含绝对路径或者相对路径。如果用户指定了绝对路径，OSG 将到指定的位置去搜索文件。

如果文件名包含了相对路径（或者不包含路径），OSG 将使用 osgDB 的数据文件路径表来搜索文件。用户可以通过设置环境变量 OSG_FILE_PATH 来设置这个目录列表，具体内容可以参照 1.3.3 节“环境变量”。

要添加指定的数据目录到数据文件路径表，可以使用 osgDB::Registry::getDataFilePathList() 函数。osgDB::Registry 是一个单态类（singleton），因此要调用这个函数的话，需要使用单态类的实例。函数会返回一个 osgDB::FilePathList 的指针，也就是一个简单的 std::deque<std::string>。假如要从一个字符串 newpath 中添加目录到列表的话，可以使用下面的代码：

```
osgDB::Registry::instance()->getDataFilePathList().push_back( newpath );
```

如果因为某些原因，OSG 不能读取文件，那么这两个函数都会返回 NULL 指针。要查看文件读取失败的原因的话，可以设置 OSG_NOTIFY_LEVEL 环境变量为一个较高的值（例如 WARN），然后再次尝试读取文件，并检查程序控制台输出的警告或者错误信息。

写入文件

使用函数 osgDB::writeNodeFile()和 osgDB::writeImageFile()将数据写入到 3D 模型或 2D 图像文件中。

```
bool osgDB::writeNodeFile( const osg::Node& node,
                           const std::string& filename );
bool osgDB::writeImageFile( const osg::Image& image,
                            const std::string& filename );
```

如果因为某些原因 OSG 不能写入文件，上述函数将返回 false。用户可以将 OSG_NOTIFY_LEVEL 设置为 WARN，并再次运行代码以观察操作失败的输出信息。如果写入操作成功，那么函数将返回 true。

如果文件名参数中包含了绝对路径，writeNodeFile()和 writeImageFile()将尝试将文件写入绝对路径的位置。如果文件名包含了相对路径（或者不包含路径），那么上述函数将尝试将文件写入当前目录的相对路径位置。

OSG 的写入操作会不作任何警告地覆盖同名文件。要避免这一特性的话，用户程序需要自行检查文件是否存在并采取相应的措施。

2.5.2 插件的搜索和注册

OSG 插件也就是是一组动态链接库，其中实现了 osgDB 头文件 ReaderWriter 定义的的接口。为了保证 OSG 可以找到这些插件，插件所在目录必须在 Windows 的 PATH 环境变量或者 Linux 的 LD_LIBRARY_PATH 中列出。最终用户也可以在 OSG_LIBRARY_PATH 环境变量中指定新的插件搜索路径。

OSG 仅识别符合下面的命名格式的插件库。

- Apple——osgdb_<name>
- Linux——osgdb_<name>.so
- Windows——osgdb_<name>.dll

<name>通常指文件的扩展名。例如，读取 GIF 图片的的插件在 Linux 系统下名为 osgdb_gif.so。

对于开发者来说，使用文件扩展名来命名插件的方法并不一定总是适用，因为有的插件可以支持多种文件的扩展名。例如用于读取 RGB 图形的插件，事实上可以读取这些格式的文件：.sgi，.int，.inta，.bw，.rgba，.rgb。osgDB::Registry 的构造函数包括了特定的代码来实现这类插件。Registry 类维护了一个扩展名的化名映射表来对应不同的文件格式和支持此类格式的插件。

警告：

插件不一定同时支持读取和写入的操作。例如，OSG 目前的版本仅支持 3D Studio Max 插件中.3ds 文件的读取操作，而不支持写入.3ds 文件的操作。事实上，多数 OSG 插件仅支持文件的读入，而不支持输出。

要获取最新的文件格式支持信息，请登陆 OSG 维基网站[OSGWiki]。

OSG 不可能查找并加载所有的插件以获取它们支持的文件格式。这样在程序启动时将会是一笔很大的开销。因此，OSG 使用职责链（Chain of Responsibility）的设计模式[Gamma95]，以加载尽量少的插件。当用户程序尝试使用 osgDB 读取或写入文件时，OSG 将按照下面的步骤来查找合适的插件：

1、OSG 搜索已注册的插件列表，查找支持文件格式的插件。开始时已注册插件列表仅包含了 Registry 类构造函数中注册的插件。如果 OSG 找到了可以支持此文件格式的插件，并成功执行了 I/O 操作，那么它将返回相应的数据。

2、如果没有发现可以支持此格式的已注册插件，或者 I/O 操作失败，那么 OSG 将根据前述的文件命名规则创建插件文件的名称，并尝试读取相应的插件库。如果读取成功，OSG 将添加此插件到已注册插件列表中。

3、OSG 将重复执行步骤 1。如果文件 I/O 的操作再次失败，OSG 将返回失败信息。

总的来说，用户不必了解 OSG 内部如何实现文件 I/O 操作，就可以使插件顺利工作。反之，如果文件 I/O 操作失败的话，用户也可以根据给出的错误信息跟踪插件源代码中的相关内容。

2.6 NodeKit 与 osgText

OSG 提供了丰富的设计功能，不过，开发者还是经常需要从 OSG 的核心节点类中派生出自己的特定节点。而这些派生的功能往往并不属于 OSG 核心的部分，而是更适合作为辅助的模块库存在。所谓 NodeKit 就是指一种开发集成库，它扩展自 OSG 核心功能的特定节点类（Node），渲染属性类（StateAttribute）和绘图类（Drawable），并且使用 osg 包装类（wrapper）对这些新的功能类提供.osg 文件格式的支持。

第 1.6.3 节“组件”介绍了 NodeKit 的概念，并对 OSG 目前已有的 NodeKit 做了纵览。本节会举例说明如何使用常用的 NodeKit。本节将提供一个使用 osgText 的例子，用来在场景图形中显示纹理贴图的文字。

2.6.1 osgText 组件

osgText 库定义了一个命名空间，osgText。在这个命名空间中有一些十分实用的字体加载和文字渲染类。

osgText 库的核心组件是 osgText::Text 类。Text 继承自 Drawable，因此用户程序应当使用 addDrawable()方法把 Text 实例添加到 Geode 中（与添加 Geometry 实例的方法相同）。Text 可用于显示一个任意长度的字符串。因此，用户程序可以为每个将要显示的字符串创建一个相应的 Text 对象。

osgText 库的另一个核心组件是 osgText::Font 类。osgText 的函数可以根据字体文件的名称来创建 Font 对象。Font 类使用 FreeType 插件来读取字体文件。用户程序将 Font 对象和 Text 对象相关联时，Font 将创建一个用于绘制字符串图形的纹理贴图。在渲染时，Text 将使用与该图形相符的纹理坐标，为文本中的每一个字符绘制一个已添加纹理的四边形。osgText 库还定义了一个 String 类，以支持多字节字符（multibyte）和各类文字编码。

2.6.2 使用 osgText

Text 和 Font 对象的定义分别位于两个头文件中。下面的代码演示了在程序中包含它们的方法。

```
#include <osgText/Font>
#include <osgText/Text>
```

要在程序中使用 osgText，用户通常要遵循下面三个步骤：

1、如果要使用一种字体显示多行文字，只需要创建一个 Font 对象，然后在 Text 对象间共享即可。

2、为每一段要显示的字符串建立一个 Text 对象。指定对齐方式，文字方向，位置和大小参数。将步骤 1 中创建的 Font 对象关联到新的 Text 对象中。

3、使用 addDrawable()函数将 Text 对象添加到 Geode 节点。用户可以向一个

Geode 添加多个 Text 对象，或者根据自己的需要创建多个 Geode 节点。

将 Geode 节点作为场景图形的子节点加入。

下面的代码演示了使用 Courier New TrueType 字体文件 cour.ttf 创建一个 Font 对象的方法。

```
osg::ref_ptr<osgText::Font> font =
    osgText::readFontFile( "fonts/cour.ttf" );
```

函数 osgText::readFontFile()可以方便地使用 OSG 的 FreeType 插件来加载字体文件。它使用 osgDB 库来查找文件，如 2.5 节“文件 I/O”中所述，程序会搜索 OSG_FILE_PATH 中的路径来获取文件位置。但是，readFontFile()同时还会搜索不同平台上的字体文件目录列表。如果 readFontFile()函数不能找到指定的字体，或者指定的文件无效，那么它将返回 NULL。

下面的代码用于创建一个 Text 对象，关联字体，并设置要显示的文字。

```
osg::ref_ptr<osgText::Text> text = new osgText::Text;
text->setFont( font.get() );
text->setText( “Display this message.” );
```

Text::setText()可以使用 std::string 作为输入参数，不过它使用 osgText::String 来实现多字符编码（multibyte encodings）的支持。osgText::String 的一些非显式构造函数可以接收 std::string 或者字符串常量。在上面的代码中，调用 setText()时使用的字符串常量参数将被转换成 osgText::String，这是在运行时完成的。

如果使用 readFontFile()加载字体失败，且用户程序在运行的系统上无法找到任何可用的字体，那么不必调用 Text::setFont()。这样 Text 会使用缺省且保证可用的字体来代替。

Text 有数种方法来控制字的大小，外观，方向和位置。下面的部分将讨论如何使用这些参数。

坐标位置

Text 与 Geometry 相似，都可以在拣选（cull）和绘制（draw）遍历中变换自己的坐标位置。缺省情况下，对象坐标的位置在其原点。用户可以使用 Text::setPosition()方法改变这一数值，此方法的输入参数为一个 Vec3 变量。

```
// 在（10.0，0.0，1.0）绘制文字。
text->setPosition( osg::Vec3( 10.f, 0.f, 1.f ) );
```

坐标位置本身并不能完全决定文字最后出现的位置。Text 将变换的位置，方向和对齐方式这些参量结合到一起，来确定文字渲染的实际区域。有关文字方向和对齐方式的内容将在下面予以讨论。

文字方向

方向决定了渲染文字在 3D 空间中的朝向。用户可以使用 Text::setAxisAlignment() 方法来设置文字的位置，其输入参数为 Text::AxisAlignment 的枚举量。如果要创建一个一直朝向视口的广告牌形式的文字，可以使用 Text::SCREEN。

```
text->setAxisAlignment( osgText::Text::SCREEN );
```

或者，你也可以让文字沿着轴对齐的平面水平放置。缺省的文字方向为 Text::XY_PLANE，即，文字朝向 Z 轴水平放置在 XY 平面上。

```
text->setAxisAlignment( osgText::Text::XY_PLANE );
```

Text::AxisAlignment 的枚举类型总共有七种：Text::XY_PLANE（缺省），Text::XZ_PLANE，Text::YZ_PLANE 将文字面向某个轴，放置在指定的平面上；Text::REVERSED_XY_PLANE，Text::REVERSED_XZ_PLANE 和 Text::REVERSED_YZ_PLANE 与此类似，但是文字朝向指定轴的负向；Text::SCREEN 使文字总是朝向屏幕。

对齐方式

此处的对齐方式与字处理软件中文字对齐方式，或者电子表格中单元对齐的概念相似。它决定了渲染文字相对于其坐标位置（使用 setPosition()函数）的水平和竖直对齐方式，Text 定义了枚举类型 Text::AlignmentType，其中的枚举量可用于确定文字的水平和竖直对齐方式。缺省选项是 Text::LEFT_BASE_LINE，即沿着文字的左边界水平放置，沿着文字的底线竖直放置。图 2-11 演示了各种对齐方式下，文字相对于自身坐标位置的差异。

text->setAlignment(osgText::Text::CENTER_TOP);

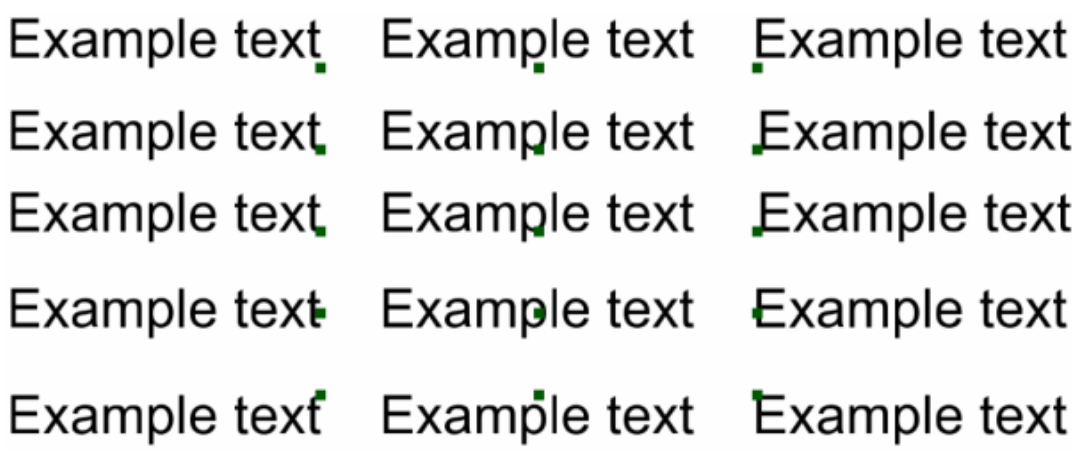


图 2-11 文字对齐方式

本图演示了十五种不同的 AlignmentType 枚举量的作用效果示例。在每个例子中，文字的坐标位置（使用 setPosition()函数）使用暗绿色的点表示。从左至右，从上到下依次为：RIGHT_BOTTOM，CENTER_BOTTOM，LEFT_BOTTOM，RIGHT_BOTTOM_BASE_LINE，CENTER_BOTTOM_BASE_LINE，LEFT_BOTTOM_BASE_LINE，RIGHT_BASE_LINE，CENTER_BASE_LINE，LEFT_BASE_LINE，RIGHT_CENTER，CENTER_CENTER，LEFT_CENTER，RIGHT_TOP，CENTER_TOP，及 LEFT_TOP。

文字尺寸

缺省的字符高度为 32 个物体坐标单元。而字符的宽度可变，由字体的性质决定。Text 服务根据 Font 对象中保存的信息，按照正确的比率渲染文字。

要改变缺省的文字高度，可以调用 Text::setCharacterSize()，下面的代码将字符的高度改变为一个物体坐标单元。

text->setCharacterSize(1.0f);

缺省情况下，Text 将 setCharacterSize()传入的参数视为物体的实际大小。不过 Text 也允许用户指定字符在屏幕上显示的大小，而不是物体的真实坐标。使用 Text::setCharacterSizeMode()方法可以指定使用屏幕坐标。

text->setCharacterSizeMode(osgText::Text::SCREEN_COORDS);

将字符高度的设定模式改为屏幕坐标后，Text 将根据视角，适当放缩文字几何体以保持它在屏幕上的恒定尺寸。注意，OSG 是在拣选（cull）遍历中调整文字的，它会根据最后一帧的状态进行自动地调整，因此会产生一个单帧延迟。在帧连贯性很好的程序中，这个延迟是可以忽略不计的。

分辨率

用户程序通常需要改变字体纹理贴图的图形分辨率，以避免文字出现模糊。缺省情况下，osgText 为每个图形分配了 32×32 个像素元。要改变这个数值的话，可以使用 Text::setFontResolution()方法。下面的代码增加了字体的分辨率，osgText 将因此为每个图形分配 128×128 个像素元。

text->setFontResolution(128, 128);

如果多个分辨率不同的 Text 文字对象共享一个 Font 对象，且不同的文本中包含了同样的字符，那么字体纹理贴图中将包含这些字符的多个冗余的纹理拷贝，它们的分辨率是各不不同的。

注意，字体分辨率的上升也会使得硬件资源的需求，譬如显卡纹理的内存上升。因此用户应当在文字的生成结果可以接受的前提下，选用最小的字体分辨率。

颜色

缺省情况下，Text 使用白色来绘制文字。用户可以使用 Text::setColor()方法

来改变这一缺省值。要设置颜色的话，可以向 setColor()指定一个 osg::Vec4 的 RGBA 颜色值。下面的代码将 Text 渲染文字的颜色改为蓝色。

```
// 设置文字颜色为蓝色。
text->setColor( osg::Vec4( 0.f, 0.f, 1.f, 1.f ) );
```

osgText 库包含的 Text 类和 Font 类还允许用户设置一切其它的参数，不过这已经超出了本书的讨论范围。用户可以阅读头文件 include/osgText/Text 及 include/osgText/Font 来获取更多信息。

2.6.3 文字示例代码

本书源代码附带的文字示例代码演示了相对一个单一几何体放置文字的方法。代码创建了一个简单的只包含 Geode 的场景图形。Geode 包含四个 Drawables，其中包括一个采用高氏着色（Gouraud-shaded）的 XZ 平面的四边形和三个 Text 对象。其中两个 Text 对象是面向屏幕的，分别标识了四边形的左上角和右上角。第三个 Text 对象放置在 XZ 平面上，四边形之下。

和本章的大多数例子一样，文字示例程序只是简单地创建场景图形并将其写入到.osg 文件中。要观察场景图形的话，可以使用下面的语句：

```
osgviewer Text.osg
```

图 2-12 所示为 osgviewer 中所示的场景图形。

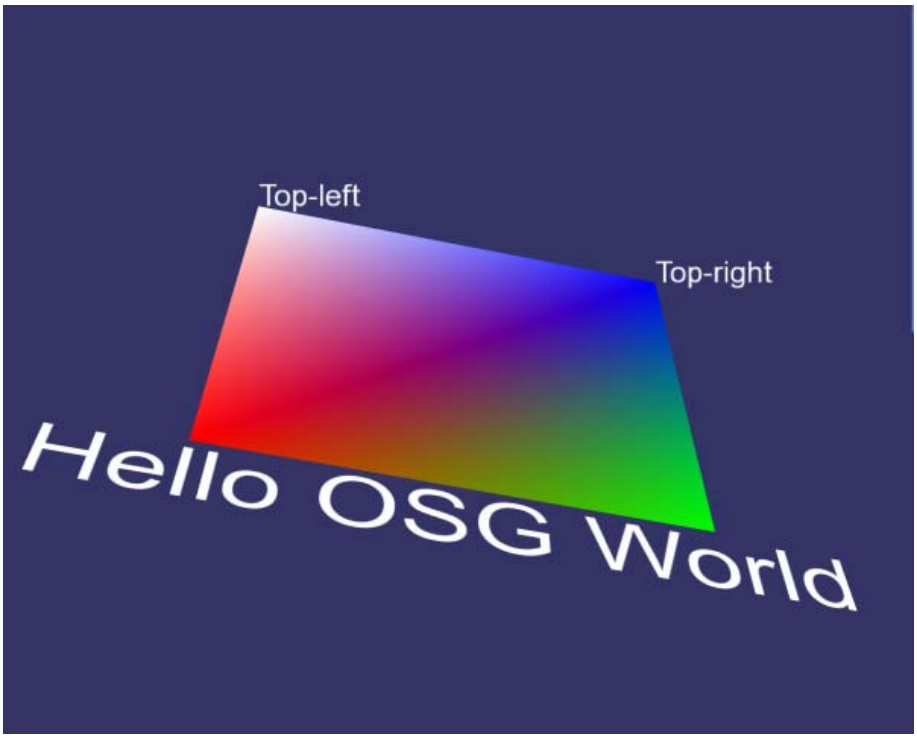


图 2-12 在 osgviewer 中显示的文字示例场景图形

文字示例代码创建了三个 Text 对象，其中使用了两种不同的字体分辨率和朝向（SCREEN，XZ_PLANE）。

2.6.4 .osg 文件格式

所有的 NodeKit 均包括两种格式的库。一是 NodeKit 的主要功能实现，比如一个继承了 Node，Drawable 或 StateAttribute 的新类。用户程序可以链接这个库以使用新的功能类。第二种库的形式是.osg 封装，即允许使用.osg 格式的文件读取和写入新类的数据信息。

.osg 文件格式是标准的 ASCII 文本，并不是一种可以实现高效存储和读取的数据结构。不过它是一种极好的调试工具。本章的所有示例程序都把其场景图形写入了.osg 文件。在渲染过程中这是没有必要的——实际应用的程序会自行创建场景图形并实现渲染和显示。例子程序中使用.osg 文件，更重要的是为了演示这种场景图形调试的技术。

进行程序开发时，如果遇到了不可预料的情况，用户可以将子场景或者整个场景图形作为.osg 文件输出。你可以使用文本编辑器打开.osg 文件来检查渲染出

错的根本原因。如果你认为某处可能是问题所在，那么也可以通过手动修改.osg 文件的方式来验证你的判断。

许多 OSG 开发者会使用 OSG 用户邮件列表来咨询自己遇到的渲染问题。如果能随之提供一份包含了出错的子场景信息的.osg 文件，那么将大大有助于其他邮件用户对这个问题进行诊断和修正。

清单 2-4 所示为 Text 示例程序的.osg 文件输出。其中包含了 osg 库所保存的 Geode 和 Geometry 类的信息，以及顶点和颜色数据。文件中同时还包括 osgText 库封装的 Text 对象和它的相关参数。

文件采用了缩进的格式以便于阅读。子节点和保存的数据均相对其父节点对象缩进两个空格，并使用花括号标识嵌套层次。

清单 2-4 Text 示例程序场景图形的.osg 文件

Text 示例程序创建了一个包含单一叶节点和四个 Drawable 对象（一个 Geometry 和三个 Text 对象）的场景图形。

```
Geode {
  DataVariance UNSPECIFIED
  nodeMask 0xffffffff
  cullingActive TRUE
  num_drawables 4
  Geometry {
    DataVariance UNSPECIFIED
    useDisplayList TRUE
    useVertexBufferObjects FALSE
    PrimitiveSets 1
    {
      DrawArrays QUADS 0 4
    }
    VertexArray Vec3Array 4
    {
```

```
            -1 0 -1
            1 0 -1
            1 0 1
            -1 0 1
        }
        NormalBinding OVERALL
        NormalArray Vec3Array 1
        {
            0 -1 0
        }
        ColorBinding PER_VERTEX
        ColorArray Vec4Array 4
        {
            1 0 0 1
            0 1 0 1
            0 0 1 1
            1 1 1 1
        }
    }
    osgText::Text {
        DataVariance UNSPECIFIED
        StateSet {
            UniqueID StateSet_0
            DataVariance UNSPECIFIED
            rendering_hint TRANSPARENT_BIN
            renderBinMode USE
            binNumber 10
            binName DepthSortedBin
        }
        supportsDisplayList FALSE
```

```
        useDisplayList FALSE
        useVertexBufferObjects FALSE
        font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
        fontResolution 32 32
        characterSize 0.15 1
        characterSizeMode OBJECT_COORDS
        alignment LEFT_BASE_LINE
        autoRotateToScreen TRUE
        layout LEFT_TO_RIGHT
        position 1 0 1
        color 1 1 1 1
        drawMode 1
        text "Top-right"
    }
    osgText::Text {
        DataVariance UNSPECIFIED
        Use StateSet_0
        supportsDisplayList FALSE
        useDisplayList FALSE
        useVertexBufferObjects FALSE
        font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
        fontResolution 32 32
        characterSize 0.15 1
        characterSizeMode OBJECT_COORDS
        alignment LEFT_BASE_LINE
        autoRotateToScreen TRUE
        layout LEFT_TO_RIGHT
        position -1 0 1
        color 1 1 1 1
        drawMode 1
```

```
            text "Top-left"
        }
        osgText::Text {
            DataVariance UNSPECIFIED
            Use StateSet_0
            supportsDisplayList FALSE
            useDisplayList FALSE
            useVertexBufferObjects FALSE
            font C:\OSGDev\OpenSceneGraph-Data\fonts\arial.ttf
            fontResolution 128 128
            characterSize 0.4 1
            characterSizeMode OBJECT_COORDS
            alignment CENTER_TOP
            rotation 0.707107 0 0 0.707107
            layout LEFT_TO_RIGHT
            position 0 0 -1.04
            color 1 1 1 1
            drawMode 1
            text "Hello OSG World"
        }
    }
```

在清单 2-4 的顶级是一个 Geode 叶节点。文件中的一些设置参数，如 NodeMask，CullingActive 等，已经超出了本书学习的范围，而参数 num_drawables 被设置为 4。四个 Drawable 对象位于同一层级。Geometry 中的第一个 Drawable 用于渲染四边形几何体。它包含了例子中指定的，Geometry 所需的所有参数。

三个 Text 对象在 Geometry 对象之后。前两个 Text 对象均含有一个设置为 TRUE 的 autoRotateToScreen 参数，因此它们将始终朝向屏幕。第三个 Text 对象包括一个旋转参数，它有四个 Quat 的值以保证放置文本于 XZ 平面内。Text 对

象还包含了其它一些常用的参数，例如颜色（设置为白色），位置值，字体文件名称等。

作为实验，你可以修改其中一个 Text 对象的颜色 RGBA 值，保存文件并使用 osgviewer 进行观察。例如，下面一行设置颜色参数为紫色。

color 0.6 0 1 1

这种修改可能显得并不足道，但是如果你正在调试一处灯光的错误，并且你认为灯光的散射颜色可能太暗了，那么不妨修改.osg 文件并加亮灯光的散射颜色。这是一种快速验证和调试程序的方法。

第一个 Text 对象还包含了一个 StateSet。StateSet 的参数意义已经超出了本书讨论的范围，但它们可以从本质上说明，OSG 将最后渲染 Text 对象，并且采用从后向前的顺序实现透明和图像混合的效果。（Text 在内部会自动允许渲染时图形混合的渲染模式。）另外两个 Text 对象并没有包含 StateSet 字样，因为 osgText 库可以在两个 Text 对象之间实现 StateSet 的共享，以节约内存。如果你注意观察其它两个 Text 对象，会发现其中包含这样一行：

Use StateSet_0

当 OSG 读取.osg 文件时，使用 Use 参数表明是数据共享。在这种情况下，OSG 将设置另外两个 Text 对象与第一个 Text 对象一起共享渲染状态 StateSet_0。

作为 OSG 开发者，用户应当对.osg 文件有所熟悉。因此不妨花一些时间通读本章示例程序中所列出的每个.osg 文件。也许你并不能理解所有的参数，但是至少可以理解这种文件结构，以及它匹配文件内容并创建场景图形的这种方法。

3、在用户程序中使用 OpenSceneGraph

一个真正的应用程序要完成的工作绝不仅仅是建立场景图形和将其写入到文件中去。本书最后一章将探讨将 OSG 集成到用户应用程序中去的各种技术。你可以在本章学到场景图形渲染，观察视角的改变，图形对象的选择操作，以及动态的修改场景图形数据的方法。

3.1 渲染

OSG 开放了所有的功能模块。因此用户程序完全可以使用最底层的 OSG 功能来执行渲染操作。假设有用户希望能够完全自主地控制场景图形的渲染，那么也可以按照下面的步骤编写应用程序的代码。

- 设计自己的视角管理代码，以改变 OpenGL 的模型-视图（model-view）矩阵。
- 创建用户窗口和 OpenGL 上下文，并将它们激活。如果有需要的话，用户也可以自行编写管理多窗口和多个设备上下文的代码。
- 如果用户程序需要使用分页数据库（paged database），那么可以使用 osgDB::DatabasePager 类。
- 可以将 osgUtil::UpdateVisitor，osgUtil::CullVisitor 和 osgUtil::RenderStage 对象实例化，以实现场景的更新，拣选和绘制遍历。如果用户希望获得更多的控制权，则可以自己编写类来实现以上遍历的特性。
- 编写主循环代码来处理操作系统系统返回的事件。并且调用自己的视角观察代码来更新模型-视图矩阵。
- 在渲染一帧之前先要调用 glClear()。渲染时要依次执行场景更新，拣选和绘制遍历，最后交换缓存数据。

- 如果用户程序或者运行该程序的软、硬件平台需要立体化渲染（stereo rendering）或者多通道渲染（multipipe rendering）功能的支持，则可以自行编写额外的代码。
- 最后，请依照平台无关性的要求来编写所有的代码，这样你的代码才可以在所有的目标平台上运行通过。

上述的步骤并非不可行，但是十分繁琐和浪费时间。并且，由于程序使用了底层的接口，当 OSG 的后继版本修改了这些底层函数之后，用户的程序将不再和 OSG 的最新版本兼容。

幸运的是，OSG 每时每刻都在进化，它不断的将一些更为简捷的渲染函数合并到现有版本中，为用户程序提供更多更强大的渲染方案。用户使用 OSG 编写程序时，往往可以利用以下这些工具和库来简化开发的过程。

- osgUtil::SceneView——这个类封装了更新，拣选和绘制遍历，但是并不启用 DatabasePager。有一部分应用程序会使用 SceneView 作为 OSG 渲染的主接口。
- Producer 和 osgProducer——Producer 是一个外部的摄像机库，可以支持多通道渲染。osgProducer 是一个集成了 Producer 和 OSG 的应用库。Producer 有相当多的用户，目前有一部分的 OSG 程序是基于 Producer 和 osgProducer 开发的。

OSG 2.0 版本的核心库添加了一个新的库成员——osgViewer 库。osgViewer 包含了一系列用于控制视口显示的相关的类，并封装了大量用户常用的功能函数，例如显示管理，事件响应，场景渲染等。这个库使用 osg::Camera 类来管理 OpenGL 的模型-视图矩阵。与 SceneView 类不同，osgViewer 库的视口类提供了对 DatabasePager 的全部支持。osgViewer 还可以针对同一个场景图形，提供并通过多个独立的视口显示该场景。

第 1.6.3 节“组件”对 osgViewer 的三个视口类 SimpleViewer，Viewer 和 CompositeViewer 作了一个大概的介绍。本章将针对如何在程序中使用 Viewer 类实现 OSG 渲染功能这一问题，继续加以讨论。

3.1.1 Viewer 类

Viewer 类的示例程序可以在本书的附带源代码中找到，它包含了在应用程序中渲染 OSG 图形所需的最简短代码。Viewer 将实例化一个 osgViewer::Viewer 对象，而后将一个场景图形关联到该对象，并进行渲染。源代码中只有三行是与这一工作相关的，工作效率很高，如清单 3-1 所示。

清单 3-1 Viewer 示例代码

本代码演示了在应用程序中渲染 OSG 图形所需的最小代码。

```
#include <osgViewer/Viewer>
#include <osgDB/ReadFile>
int main( int, char ** )
{
    osgViewer::Viewer viewer;
    viewer.setSceneData( osgDB::readNodeFile( "cow.osg" ) );
    return viewer.run();
}
```

此示例代码的运行结果类似于执行第 1.3 节中所述的 osgviewer 命令行指令。

```
osgviewer cow.osg
```

这一相似并不是巧合。究其原因可知，osgviewer 同样使用了 Viewer 类来实现场景的渲染。不过，osgviewer 为 Viewer 配置了许多附加的功能，正如你所期望的那样，程序也将会比清单 3-1 中的代码更加丰富。

改变视口

Viewer 类在内部创建了一个 osg::Camera 摄像机对象来管理 OSG 的模型-视图矩阵。用户可以通过下面两种方法来控制 Camera 对象。

- 将一个摄像机控制器对象关联到 Viewer 中。如果你的程序不需要这么

做，那么 Viewer::run()将自动创建一个 osgGA::TrackballManipulator 对象来控制摄像机的工作。osgGA 库定义了一些常用的控制器类。用户可以调用 Viewer::setCameraManipulator()来指定一个期望的控制器。

- 设置 Camera 对象的投影矩阵和观察矩阵为自定义的矩阵值。这样也可以保证用户程序能够完全控制视口的浏览动作。

如果你选择直接设置 Camera 对象的矩阵值，那么就不要使用 Viewer::run()函数，因为它不允许每帧都实时的改变视口的参数。你可以编写一个小的循环来代替这个函数，循环中可以反复更新视口并渲染单帧图像。清单 3-2 中的程序所示为这种方法的实现示例。

清单 3-2 直接控制视口

下面的代码段实现了直接控制 Viewer 中的 Camera 对象，从而改变每一帧的视口显示。

```
osgViewer::Viewer viewer;
viewer.setSceneData( osgDB::readNodeFile( "cow.osg" ) );
viewer.getCamera()->setProjectionMatrixAsPerspective( 40., 1., 1., 100. );

// 创建矩阵，指定到视点的距离。
osg::Matrix trans;
trans.makeTranslate( 0., 0., -12. );
// 旋转一定角度（弧度值）。
double angle( 0. );

while (!viewer.done())
{
    // 创建旋转矩阵。
    osg::Matrix rot;
    rot.makeRotate( angle, osg::Vec3( 1., 0., 0. ) );
    angle += 0.01;
```

```
    // 设置视口矩阵（旋转矩阵和平移矩阵连乘）。
    viewer.getCamera()->setViewMatrix( rot * trans );

    // 绘制下一帧
    viewer.frame();
}
```

清单 3-2 中的代码在渲染循环外只设置过一次 Camera 的投影矩阵。Camera 类提供了一些指定投影矩阵的方法，对于大多数 OpenGL 开发者而言，这些方法应当说都是似曾相识的。

```
void setProjectionMatrix( const osg::Matrix& matrix );
void setProjectionMatrixAsOrtho( double left, double right,
double bottom, double top, double zNear, double zFar );
void setProjectionMatrixAsOrtho2D( double left, double right,
double bottom, double top );
void setProjectionMatrixAsFrustum( double left, double right,
double bottom, double top, double zNear, double zFar );
void setProjectionMatrixAsPerspective( double fovy,
double aspectRatio, double zNear, double zFar );
```

Camera::setProjectionMatrix()方法的输入参数为 osg::Matrix 对象，这与下面的 OpenGL 命令是类似的：

```
glMatrixMode( GL_PROJECTION );
glLoadMatrixf( m );
```

Camera::setProjectionMatrixAsOrtho()方法同样可以创建一个投影矩阵。其计算方法基本等同于使用 OpenGL 命令 glOrtho()。

setProjectionMatrixAsOrtho2D()方法与 GLU 函数入口 gluOrtho2D()更为相

似。Camera 类同样提供了设置透视投影参数的方法，其用法与 glFrustum()和 gluPerspective()命令相似。

在渲染循环中，示例代码每帧都会更新 Camera 的视口矩阵以增加旋转角度值。Camera 类自然也为之提供了一些为 OpenGL 开发者所熟悉的设置函数。清单 3-2 的代码显式地使用 setViewMatrix()方法来设置视口矩阵，而 Camera 类还提供了与之相似 setViewMatrixAsLookat()方法，它的输入参数类同于 OpenGL 的 gluLookAt()函数。

设置清屏颜色

除设置视口外，Viewer 类的 Camera 对象还提供了其它一些实用的操作。用户程序可以使用 Camera 类来设置清屏颜色。下面的代码将设置清屏颜色为黑色。

```
viewer.getCamera()->setClearColor( osg::Vec4( 0., 0., 0., 1. ) );
```

缺省情况下，Camera 会清除深度和颜色缓存。如果要改变这一缺省特性，可以使用 Camera::setClearMask()方法，并传入适当的 OpenGL 缓存标志。

```
viewer.getCamera()->setClearMask(GL_COLOR_BUFFER_BIT |
    GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT );
```

上面的代码段将清除颜色、深度和模板缓存。

3.1.2 SimpleViewer 和 CompositeViewer

osgViewer 还提供了两个额外的视口类，不过这已经超出了本书的讨论范围。本节将在一个较高的层面上对这两个类作简单的阐述。

Viewer 类可以简化 OSG 程序的开发过程，它提供了大量扩展的功能函数。如果用户要将一个已有的程序移植到 OSG 环境中，则需要有一个可以与现有程序框架相配合的简单的视口类。SimpleViewer 类就是基于这一目的设计的。与 Viewer 类不同，SimpleViewer 类不会主动创建窗口或者设备上下文。相反，它需要依赖用户程序来创建窗口、上下文，并将其激活为当前设备。如果操作正确的

话，SimpleViewer::frame()将会在用户给定的窗口中进行渲染。

Viewer 类只能在一个场景中添加一个视口（不过似乎可以使用一组摄像机来支持多通道渲染），而 CompositeViewer 类可以支持一个或多个场景的多个视口显示，并允许用户程序指定其渲染顺序。

CompositeViewer 可以支持渲染到纹理（render-to-texture，RTT）的操作，即，允许用户程序将一个视口中渲染的图像作为另一个视口的纹理贴图。

关于 SimpleViewer 和 CompositeViewer 的最新信息，请登陆 OSG 维基网站[OSGWiKi]。

3.2 动态更改

OSG 允许用户动态的修改场景图形并因而改变每一帧的显示。这一特性是任何交互式的图形应用程序所必需的。用户可以更改几何数据，渲染状态参数，Switch 节点设置，以及任何场景图形的结构。

正如第一章所述，拣选（cull）遍历中关联了渲染图形中的几何数据和渲染状态信息，它们将在绘制（draw）遍历中进行处理。osgViewer 库支持多线程模式，每一个线程均独立地运行拣选及绘制遍历。出于性能优化的考虑，OSG 并没有为了线程的安全性增设内存锁，而是要求用户程序只可在拣选及绘制遍历的时域之外修改场景图形。

有几种方法可以确保用户的修改不会与拣选及绘制线程发生冲突。一个简单的方案是，在 Viewer::frame()的调用之外进行场景图形的修改，这需要在主渲染循环中添加额外的代码。但如果用户希望自己的程序更加整洁和规范的话，可以选择在更新遍历中进行场景的修改操作。

本节将介绍一些与场景图形动态更改相关的基本技术。

- 出于性能优化和线程安全性的考虑，用户需要通知 OSG，场景图形的哪些部分是可能要进行修改的。用户可以通过设置 Object 对象（Node，Drawable，StateSet 等）的数据变度（data variance）属性来完成这一工作。

- OSG 允许用户程序为 Node 和 Drawable 设置回调（callback）。OSG 将在特定遍历中执行这些回调。例如要在更新遍历中修改 Node 或者 Drawable 对象的值，就可以通过设置更新回调（update callback）来完成。
- 用户程序有时不能预知场景图形的哪一部分需要修改。这时需要搜索整个场景图形来查找特定的节点，或者由用户使用鼠标等输入设备来选择一个节点。

下面的章节将对此详细加以讲解。

3.2.1 数据变度

osgViewer 支持的多线程模型允许用户程序主循环不必等到绘制遍历结束就可以继续运行。这就是说 Viewer::frame()方法在绘制遍历仍未结束的时候就可以返回。换句话说，上一帧的绘制遍历可以与下一帧的更新遍历产生交叠。仔细考虑这一线程模型的实现的话，会发现它几乎很难避免与绘制遍历线程的冲突。不过，OSG 提供了 osg::Object::DataVariance()方法作为这一问题的解决方案。

程序崩溃的原因：

当用户开发了动态更改场景图形的代码时，可能会在修改场景图形时遇到程序崩溃或者段错误的情况。诸如此类的问题往往都是由于用户在拣选和绘制遍历过程中修改了场景图形的数据，而造成系统崩溃。

要设置一个 Object 对象的数据变量，可以调用 setDataVariance()并设置输入参数为 Object::DataVariance 枚举量。初始状态下，变度的值是 UNSPECIFIED。用户程序可以将数据变度更改为 STATIC 或者 DYNAMIC。OSG 将确保绘制遍历在所有的 DYNAMIC 节点和数据处理完成后才会返回。同样，由于绘制遍历在函数返回后仍然可以继续渲染场景图形，OSG 将确保此时只有 STATIC 数据可以继续进行图形渲染。如果用户的场景图形包含了很少的 DYNAMIC 数据，那么绘制遍历可以很快返回，保证用户程序可以继续执行其它的任务。

3.2.2 回调

OSG 允许用户设置 Node 和 Drawable 对象的回调类。Node 可以在 OSG 执行更新和拣选遍历时进行回调，而 Drawable 可以在拣选和绘制遍历时进行回调。本节将介绍在更新遍历中使用 osg::NodeCallback 动态修改 Node 节点的方法。OSG 的回调接口则基于一定的回调设计模式[Gamma95]。

如果要使用 NodeCallback，用户程序需要执行下面的步骤。

- 从 NodeCallback 继承一个新的类。
- 重载 NodeCallback::operator()方法。使用这个方法来实现场景图形的动态更改。
- 将用户从 NodeCallback 继承的类实例化，然后使用 Node::setUpdateCallback()方法关联到将要修改的 Node。

在每个更新遍历过程中，OSG 都会调用派生类中的 operator()方法，从而允许用户程序对 Node 进行修改。

OSG 向 operator()方法传递了两个参数。第一个是回调类所关联的 Node 的地址，也就是用户回调将在 operator()方法中进行动态更改的 Node 节点。第二个参数是 osg::NodeVisitor 对象的地址。下一节将讨论有关 NodeVisitor 类的问题，因此目前你可以忽略这一参数。

用户可以使用 Node::setUpdateCallback()方法将 NodeCallback 关联到 Node。setUpdateCallback()有一个输入参数，就是 NodeCallback 派生类的地址。下面的代码段演示了将 NodeCallback 关联到节点的方法。

```
class RotateCB : public osg::NodeCallback
{
    ...
};
...
node->setUpdateCallback( new RotateCB );
```

回调可以被多个节点共享。NodeCallback 类间接继承自 Referenced，而 Node

类内部维护了一个更新回调的 ref_ptr<>指针。当最后一个关联此回调的节点被删除时，NodeCallback 的引用计数将减到 0 并被自动释放。因此，在上面的代码中，用户程序没有也不需要维护 RotateCB 对象的指针。

本书的示例代码包括了一个回调的例子，它演示了更新回调的使用方法。这段代码将牛的模型与两个 MatrixTransform 节点关联。然后代码将从 NodeCallback 派生出一个类并关联到其中一个 MatrixTransform 对象。在更新遍历中，新的 NodeCallback 将动态改变矩阵的值，从而使其中一个牛的模型不断旋转。回调示例程序的输出结果请见图 3-1。

清单 3-3 所示的示例代码包括了三个主要部分。第一部分定义了一个名为 RotateCB 的类，它派生自 NodeCallback。第二部分是一个名为 createScene()的函数，用于创建场景图形。注意当函数中创建了第一个 MatrixTransform 对象 mtLeft 时，它将调用 mtLeft->setUpdateCallback(new RotateCB)来指定 mtLeft 的更新回调。如果用户将这一行变成注释并重新运行例子程序，可以发现牛的模型不再转动了。程序的最后一部分是 main()入口点，用于创建一个视口并进行渲染。

图 3-1 使用更新回调进行动态更改

这幅图所示是回调示例程序的输出结果。这段代码让左边的牛沿着垂直轴进行动态旋转，而对右边的牛没有作任何操作。

清单 3-3 回调示例源代码

这段例子代码演示了创建 NodeCallback 并在更新遍历中更新场景图形的过程。

```
#include <osgViewer/Viewer>
#include <osgGA/TrackballManipulator>
#include <osg/NodeCallback>
#include <osg/Camera>
#include <osg/Group>
#include <osg/MatrixTransform>
#include <osgDB/ReadFile>

// 从 NodeCallback 继承一个类，以修改 MatrixTransform 对象的矩阵。
class RotateCB : public osg::NodeCallback
{
public:
    RotateCB() : _angle( 0. ) {}
    virtual void operator()( osg::Node* node, osg::NodeVisitor* nv )
    {
        // 通常应该确认一下更新访问器（update visitor）是否存在，
        // 不过这个例子中没有必要这样做。
        osg::MatrixTransform* mtLeft =
            dynamic_cast<osg::MatrixTransform*>( node );
        osg::Matrix mR, mT;
        mT.makeTranslate( -6., 0., 0. );
        mR.makeRotate( _angle, osg::Vec3( 0., 0., 1. ) );
        mtLeft->setMatrix( mR * mT );

        // 下一次回调时角度就会增大。
        _angle += 0.01;
```

```
        // 指定继续传递参数，
        // 这样 OSG 可以接着执行其它带有回调的节点。
        traverse( node, nv );
    }
    protected:
        double _angle;
};

// 创建场景图形。
// 这是一个带有两个 MatrixTransform 子节点的 Group 根节点，
// 而 MatrixTransform 节点又同时都是一个 Geode 的父节点。
// Geode 叶节点从模型文件 cow.osg 中读取数据。
osg::ref_ptr<osg::Node> createScene()
{
    // 加载牛的模型。
    osg::Node* cow = osgDB::readNodeFile( "cow.osg" );
    // 设置数据变量为 STATIC，因为程序中不会修改它。
    cow->setDataVariance( osg::Object::STATIC );

    // 创建 MatrixTransform 来显示左边的牛。
    osg::ref_ptr<osg::MatrixTransform> mtLeft =
    new osg::MatrixTransform;
    mtLeft->setName( "Left Cow\nDYNAMIC" );

    // 设置数据变量为 DYNAMIC，
    // 告诉 OSG 这个节点将在更新遍历中被修改。
    mtLeft->setDataVariance( osg::Object::DYNAMIC );
```

```
    // 设置更新回调。
    mtLeft->setUpdateCallback( new RotateCB );
    osg::Matrix m;
    m.makeTranslate( -6.f, 0.f, 0.f );
    mtLeft->setMatrix( m );
    mtLeft->addChild( cow );

    // 创建 MatrixTransform 来显示右边的牛。
    osg::ref_ptr<osg::MatrixTransform> mtRight =
        new osg::MatrixTransform;
    mtRight->setName( "Right Cow\nSTATIC" );

    // 设置数据变量为 STATIC，因为程序中不会修改它。
    mtRight->setDataVariance( osg::Object::STATIC );
    m.makeTranslate( 6.f, 0.f, 0.f );
    mtRight->setMatrix( m );
    mtRight->addChild( cow );

    // 创建 Group 根节点。
    osg::ref_ptr<osg::Group> root = new osg::Group;
    root->setName( "Root Node" );

    // 设置数据变量为 STATIC，因为程序中不会修改它。
    root->setDataVariance( osg::Object::STATIC );
    root->addChild( mtLeft.get() );
    root->addChild( mtRight.get() );
    return root.get();
}

int main(int, char **)
```

```
{
    // 创建视口并设置场景数据为以上代码所创建的场景图形。
    osgViewer::Viewer viewer;
    viewer.setSceneData( createScene().get() );

    // 设置清屏颜色不同于以往的淡蓝色。
    viewer.getCamera()->setClearColor( osg::Vec4( 1., 1., 1., 1. ) );

    // 渲染循环。OSG 将在更新遍历中调用 RotateCB::operator()。
    viewer.run();
}
```

RotateCB::operator()中包含了对 traverse()的调用。这是一个 osg::NodeVisitor 类的方法。如果场景图形中有不止一个更新回调的话，这个方法将允许 OSG 访问其它的节点并执行其回调。下面的章节将更为详细地讨论 NodeVisitor 类。

回调示例代码创建的场景图形见图 3-2。Group 根节点有两个 MatrixTransform 子节点，从而将 Geode 中牛的模型变换到两个不同的位置。如图所示，其中一个 MatrixTransform 对象设置了数据变度为 DYNAMIC，而另一个由于不会被修改，因此使用了 STATIC 数据变度。左侧的 MatrixTransform 关联了更新回调，因此可以在更新遍历中动态地修改旋转矩阵的值。

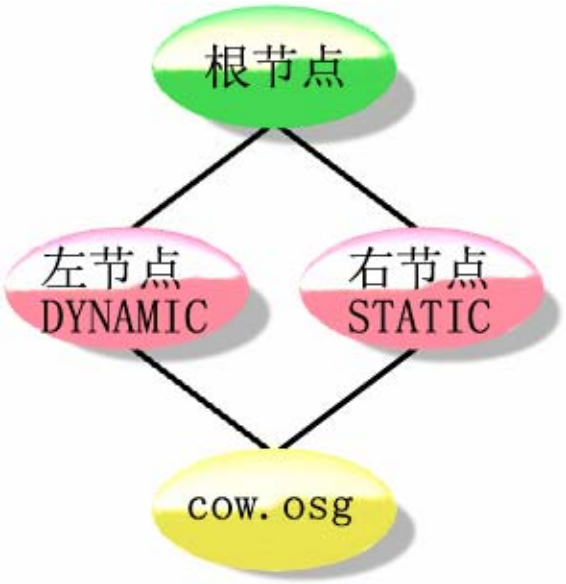


图 3-2 回调示例代码的场景图形

本图所示为回调示例代码的场景图形层次。注意两个 MatrixTransform 节点的数据变量并不相同。

正如示例代码中演示的那样，由于关联更新回调到一个已知节点的方式十分容易，因此动态修改节点的方法也比较简单。但是当用户程序要修改场景图形内部的某个节点，而且整个场景图形又是从外部模型文件读入的时候，或者用户要交互地将进行选择的时候，问题就显得比较复杂了，下面的章节将给出一些 OSG 在运行时进行节点识别的方法。

3.2.3 NodeVisitor 类

NodeVisitor 类是 OSG 对于访问器（visitor）设计思想[Gamma95]的具体实现。从本质上说，NodeVisitor 类遍历了一个场景图形并为每一个被访问节点调用特定的函数。这一简单的技术却是许多 OSG 操作的基类，包括 osgUtil::Optimizer，osgUtil 库的几何体处理类，以及文件输出类。OSG 使用 osgUtil::UpdateVisitor 类（继承自 NodeVisitor）来实现更新遍历。而在前面章节的代码中，UpdateVisitor 恰恰正是调用了 NodeCallback::operator()方法的 NodeVisitor 类。总之，NodeVisitor 在 OSG 的应用中无处不在。

NodeVisitor 是一个基类，用户程序无法直接将其实例化。但是用户程序可以使用 OSG 提供的任何 NodeVisitor 派生类，也可以自己编写继承自 NodeVisitor 的类的代码。NodeVisitor 类包含了一些经过重载的 apply()方法，其输入参数涵盖了大部分 OSG 的节点类型。当一个 NodeVisitor 对象遍历整个场景图形时，它将会为每个被访问的节点调用其相应的 apply()方法。

从文件读取了场景图形之后，用户程序通常会从被读取的场景图形中搜索所需的节点。举例来说，假设有一个机器人手臂的模型，它的每一个接合处都已经建立了一个作为关节的变换节点。那么在读取这个文件之后，用户程序可能会使用 NodeVisitor 对象来定位所有的 Transform 节点，以实现动画的效果。在这种情况下，用户程序可以自己定制新的 NodeVisitor 派生类，并重载 apply(osg::Transform&)方法。用户的 NodeVisitor 派生类的对象遍历整个场景图形的时候，每一个派生自 Transform 的节点都会作为被反复调用的 apply()方法的参数传递进来，用户程序也因此可以执行必要的操作来实现节点动画，例如，首先将这些节点的地址保存到列表中。

使用特定的名称来搜索节点也是一种简单而常用的操作。清单 3-4 中所示的代码实现了一个名为 FindNamedNode 的类。这个类的构造函数使用一个字符串作为输入参数，并保存与输入名称相符的节点的地址。

允许 NodeVisitor 遍历：
缺省情况下，NodeVisitor 基类禁止执行遍历。因此在你的派生类中，需要使用枚举量 NodeVisotor::TRAVERSE_ALL_CHILDREN 来初始化基类，以允许执行遍历。否则，OSG 将不会调用用户的 apply()方法。

清单 3-4 FindNamedNode 类的定义

此处代码所示为一个简单的 NodeVisitor 派生类的声明和定义，这个名为 FindNamedNode 类的功能是查找指定名称的节点。它是示例程序 FindNode 的一部分。

```
// 这个类从 NodeVisitor 派生，用于查找指定名称的节点。
class FindNamedNode : public osg::NodeVisitor
{
public:
    FindNamedNode( const std::string& name )
    : osg::NodeVisitor(        // 遍历所有子节点。
        osg::NodeVisitor::TRAVERSE_ALL_CHILDREN ),
    _name( name ) {}

    // 这个方法将调用场景功能图形中的每个节点，
    // 检查其名称是否符合输入的要求。
    // 如果符合的话，将保存节点的地址。
    virtual void apply( osg::Node& node )
    {
        if (node.getName() == _name)
            _node = &node;

        // 继续遍历场景图形剩余的部分。
        traverse( node );
    }

    osg::Node* getNode() { return _node.get(); }

    protected:
        std::string _name;
        osg::ref_ptr<osg::Node> _node;
};
```

如果要使用 NodeVisitor 来遍历整个场景图形，可以将 NodeVisitor 作为

Node::accept()的输入参数传递。你可以在任何一个节点上调用 accept()，NodeVisitor 将从那个节点开始遍历整个场景图形。如果要搜索整个场景图形的话，可以从根节点开始调用 accept()。

清单 3-4 所示的 FindNamedNode 类是一个名为 FindNode 的示例程序的一部分。FindNode 示例程序从磁盘中读取了一个场景图形文件，查找这个场景图形中指定名称所对应的节点，并在渲染之前修改这个节点的数据。FindNode 程序可以与 2.4.3 节提到的 State 示例程序协同工作，后者可以将场景图形输出到文件。当 FindNode 读取文件之后，它将查找 StateSet 已经被设定为单一着色的 MatrixTransform 节点，而后改变其渲染状态为平滑着色。

3.2.4 用户选择

大多数的 3D 程序都需要某种形式的用户选择功能，终端用户可以以此来交互地选择当前被显示的画面的某个一部分。用户选择中最简单的形式为，用户将鼠标移动到场景中特定的位置，并点击鼠标。程序内部将进行运算，以便将 2D 的鼠标 *XY* 坐标位置映射到正确的 3D 场景图形节点上，并保存节点地址以便将来使用。

从本质上说，OSG 程序通过两个步骤来实现用户选择。

- 接收鼠标事件。osgGA 库提供了允许程序接收鼠标事件的事件类，它具备平台无关的特性。
- 判断场景图形的哪个部分被鼠标光标覆盖。osgUtil 库提供了一种相交集（intersection）类，可以在鼠标 XY 坐标的周围创建包围盒，并判断包围盒与场景图形的相交情况。osgUtil 将按照由前至后的顺序返回与包围盒相交的节点列表。

本节将讨论如何实现这两个步骤的操作。

捕获鼠标事件

如 1.6.3 节“组件”所述，osgGA 库提供了平台无关性的 GUI 事件驱动支持。而本章的示例程序将使用 osgGA::TrackballManipulator 来实现视口矩阵的控制。

TrackballManipulator 将鼠标事件作为输入，并修改用于控制用户视口的 osg::Camera 视口矩阵。

TrackballManipulator 派生自 osgGA::GUIEventHandler 类。GUIEventHandler 是虚基类，无法被直接实例化。但是用户程序可以从 GUIEventHandler 派生自己的类，以实现各种基于 GUI 事件的操作。要实现鼠标控制的选择操作，可以从 GUIEventHandler 派生新的类并重载 GUIEventHandler::handle()方法，以接收鼠标事件。然后用户可以创建新类的实例并将其关联到应用程序的观察视口。

handle() 方法有两个输入参数，osgGA::GUIEventAdapter 和 osgGA::GUIActionAdapter，如下所示。

```
virtual bool GUIEventHandler::handle(
    const osgGA::GUIEventAdapter& ea,
    osgGA::GUIActionAdapter& aa );
```

用户实现的 handle()可以接收来自 GUIEventAdapter 的各种 GUI 事件，包括鼠标事件。GUIEventAdapter 类的头文件中定义了枚举类型 EventType，用户程序可以从中选择自己需要的 GUI 事件，例如鼠标事件。而使用 GUIEventAdapter::getEventType()方法可以得到当前的事件类型。

GUIActionAdapter 是用户程序返回给 GUI 系统的程序接口。当遇到鼠标选择的操作时，用户将用于选择事件的 GUIEventHandler 关联到视口类，则视口类就是一个 GUIActionAdapter。用户需要使用它来实现当前视口与场景之间的交互。

在渲染视口之前，用户往往会创建一个 GUIEventHandler 派生类的实例，并使用 Viewer::addEventHandler()方法将其关联到视口。正如这个函数的名称所示，一个视口可以有多个事件处理器，Viewer 类将事件处理对象添加到一个事件处理器列表中。运行时 Viewer 将调用每个 GUI 事件的 handle()函数，直到其中一个的 handle()函数返回 true 为止。

清单 3-5 的代码包含了一个名为 PickHandler 的类，它继承自 GUIEventHandler。其中 handle()方法实现了对鼠标事件类型 PUSH，MOVE 和

RELEASE 的支持。该类将用来记录鼠标在 PUSH 和 MOVE 下的 XY 坐标，当鼠标不再移动时，触发其 RELEASE 事件，既而触发选择目标对象的操作。如果用户选择成功的话，handle()返回 true。其它情况下它将返回 false，并允许其它事件处理器继续进行扫描。

清单 3-5 PickHandler 类

这里使用派生自 osgGA::GUIEventHandler 的子类来实现 OSG 中的用户选择功能。清单所示为 Picking 例子程序中的 PickHandler 类部分。这个类定义了两个方法，一个用于接收鼠标事件，另一个用于实现鼠标释放时用户选择的功能。

```
// PickHandler，用于实现用户选择的 GUIEventHandler 派生类。
class PickHandler : public osgGA::GUIEventHandler
{
public:
    PickHandler() : _mX( 0. ),_mY( 0. ) {}
    bool handle( const osgGA::GUIEventAdapter& ea,
                 osgGA::GUIActionAdapter& aa )
    {
        osgViewer::Viewer* viewer =
            dynamic_cast<osgViewer::Viewer*>( &aa );
        if (!viewer)
            return false;

        switch( ea.getEventType() )
        {
        case osgGA::GUIEventAdapter::PUSH:
        case osgGA::GUIEventAdapter::MOVE:
        {
```

```
            // 记录鼠标按下和移动时的位置信息。
            _mX = ea.getX();
            _mY = ea.getY();
            return false;
        }

        case osgGA::GUIEventAdapter::RELEASE:
        {
            // 如果鼠标在按下按键时没有移动，那么执行用户选择的处理，
            // 否则将由默认的鼠标控制器类进行处理。
            if (_mX == ea.getX() && _mY == ea.getY())
            {
                if (pick( ea.getXnormalized(), ea.getYnormalized(), viewer ))
                    return true;
            }
            return false;
        }

        default:
            return false;
        }
    }

protected:
    // 保存鼠标按下和移动时的 XY 坐标。
    float _mX,_mY;

    // 执行用户选择的操作。
    bool pick( const double x, const double y, osgViewer::Viewer* viewer )
    {
```

```
if (!viewer->getSceneData())       // 没有可以选择的。
    return false;

double w( .05 ), h( .05 );
osgUtil::PolytopeIntersector* picker =
    new osgUtil::PolytopeIntersector(
osgUtil::Intersector::PROJECTION, x-w, y-h, x+w, y+h );
osgUtil::IntersectionVisitor iv( picker );
viewer->getCamera()->accept( iv );

if (picker->containsIntersections())
{
    osg::NodePath& nodePath =
        picker->getFirstIntersection().nodePath;
    unsigned int idx = nodePath.size();
    while (idx--)
    {
        // 查找交集节点路径中的最后一个 MatrixTransform；
        // 它就是将要与回调相关联的选择结果。
        osg::MatrixTransform* mt =
            dynamic_cast<osg::MatrixTransform*>(
        nodePath[ idx ] );
        if (mt == NULL)
            continue;

        // 到了这里，
        // 说明已在节点路径中找到了所需的 MatrixTransform。
        if (_selectedNode.valid())
            // 清除原来的选择节点回调，以保证它停止运行。
            _selectedNode->setUpdateCallback( NULL );
```

```
                _selectedNode = mt;
                _selectedNode->setUpdateCallback( new RotateCB );
                break;
            }
            if (!_selectedNode.valid())
                osg::notify() << "Pick failed." << std::endl;
        }
        else if (_selectedNode.valid())
        {
            _selectedNode->setUpdateCallback( NULL );
            _selectedNode = NULL;
        }
        return _selectedNode.valid();
    }
};

int main( int argc, char **argv )
{
    // 创建场景的视口。
    osgViewer::Viewer viewer;
    viewer.setSceneData( createScene().get() );

    // 添加用户选择处理器。
    viewer.addEventHandler( new PickHandler );
    return viewer.run();
}
```

清单 3-5 中也包括了 Picking 示例程序的 main()函数，其中使用

Viewer::addEventHandler()方法将事件处理器关联到视口。

综上所述，要实现接收鼠标事件并实现用户选择的功能，需要经过以下几个步骤：

- 从 GUIEventHandler 继承新的类。重载 handle()方法。
- 在 handle()中，检查 GUIEventAdapter 参数传递的事件类型，并针对需要的事件类型执行相应的操作。方法返回 true 时将阻止其它事件处理器继续接收事件消息。
- 在渲染之前，创建事件处理器类的实例，并使用 addEventHandler()方法添加到视口中。OSG 将会把视口作为 GUIActionAdapter 参数传递给 handle()方法。

上述的技术并不局限于使用鼠标进行选择。用户程序可以尝试实现与 TrackballManipulator 类相似的接收鼠标事件的类。另外，也可以接收键盘事件并实现对按键的响应操作。

下面的部分将会介绍如何确定场景图形区域与鼠标选择的交互关系，并结束对于鼠标选择对象这一话题的讨论。

交集

你可以将通过点击鼠标的节点选择想象成是从鼠标(光标)位置向场景中发射了一条射线。被鼠标选中的场景部分将与射线有一个交集。如果场景是由线和点元素组成的，那么射线的交运算可能无法符合用户的实际选择，因为鼠标的位置几乎无法与这些图元产生精确的空间交集。此外，在典型的透视渲染中，射线交运算的精度将与观察者所处的距离成反比。

OSG 使用一种名为多胞体（polytope）的金字塔形包围盒代替射线，以克服上述的问题。这种金字塔形的顶峰位于视点，其中心轴直接穿过鼠标(光标)的位置。它距离视点的宽度是由视场和程序控制的宽度参数决定的。

OSG 使用场景图形的自顶向下的继承结构，从而避免了 OpenGL 普遍存在的“迟钝的”选择特性，而在本地 CPU 上进行高效的计算。osgUtil::IntersectionVisitor 类继承自 NodeVisitor，它可以检测每个定点的包围盒与交集包围盒的关系，并允许在某个子图形不可能存在有交集的子节点时，跳过

该子图形的遍历。

用户可以设置 IntersectionVisitor 类并使用几种不同的几何结构进行交集检测，例如平面和线段。其构造函数使用 osgUtil::Intersector 作为输入参数，Intersector 类定义了选择操作的几何体并执行实际的交集测试。Intersector 是一个纯虚基类，用户程序无法将其实例化。而 osgUtil 库从 Intersector 派生了一些代表不同几何结构的新类，例如 osgUtil::PolytopeIntersector，也就是前文所述的、较理想的鼠标点选判定模型。

有些程序需要拾取单独的顶点和多边形；而有些程序只需要简单地获取那些包含了被选节点的 Group 或 Transform 父节点。IntersectionVisitor 返回 osg::NodePath 对象来满足这些需求。NodePath 是一个 std::vector<osg::Node>向量，它表示沿着从根节点到叶节点的、呈现出一定排列层次的节点路线。如果用户程序需要获取中间的 Group 组节点，那么只需要从后向前搜索满足程序要求的节点即可。

综上所述，要实现 OSG 中的鼠标选择操作，需要按照如下的步骤编写代码：

- 创建并设置 PolytopeIntersector，其中的鼠标位置应当使用 GUIEventAdapter 中经过归一化的数据。
- 创建 IntersectionVisitor 对象，并将 PolytopeIntersector 作为其构造函数的输入参数。
- 由场景图形的根节点加载 IntersectionVisitor，一般来说是通过 Viewer 中的 Camera 对象，如下面的代码所示：

  // iv 即是一个 IntersectionVisitor 对象。

  viewer->getCamera()->accept(iv);
- 如果 PolytopeIntersector 的返回值包含了交集，那么可以获取返回的 NodePath 病搜索符合要求的节点。

如清单 3-5 所示，Picking 示例程序中的 PickHandler::pick()方法演示了上述步骤。Picking 示例程序创建了一个类似于回调示例程序的场景图形。不过，前者的场景图形在层次上前后连续使用了两个 MatrixTransform 节点，其中一个用于保存平移数据，另一个用于保存旋转数据。对于一次成功的对象选择，例子代

码将搜索 NodePath 并获取保存旋转数据的 MatrixTransform 节点。它将会把更新回调关联到这个节点之上，以实现子节点几何体的动态旋转。

当用户运行 Picking 示例程序时，它将显示两个牛的模型，这一点与回调示例程序相同。但是，当用户选择任意一个牛的模型时，模型将会响应操作并开始进行旋转。

附录：从这里开始

希望本书能够成为一本优秀的 OSG 入门读物。但是，作为一本快速入门指南，本书不可能涵盖所有的 OSG 资源。本节将介绍其它一些附加的资源信息，它们受到许多 OSG 开发者的推崇。

源代码

从开发者的角度来说，开源产品的主要优势在于源代码是全部开放的。当用户开发基于 OSG 应用软件时，通过深入 OSG 源代码并研究其内部机理，可以迅速而方便地解决用户遇到的开发问题。

如果你还没有做到这一步，不妨按照 1.2 节所述的步骤，下载完整的 OSG 源代码，并创建包括自己的调试信息的 OSG 执行库。第一次编译 OSG 可能是困难且耗费时间的，但是这势必使得以后的软件开发事半功倍。

OSG 的发布代码中同样包括大量的实用示例程序，它们演示了许多 OSG 特性的正确用法，其内容远超出本书所涵盖的部分。这些示例程序对于任何一个 OSG 开发者都具有无法衡量的价值。

仅仅使用 OSG 执行库来开发 OSG 应用程序是可行的。但是如果能灵活运用 OSG 的源代码，示例程序，以及调试版本的执行库，将大大加速软件开发的过程。

OSG 维基

OSG 维基网站[OSGWiki]包含了大量与 OSG 相关的有价值的内容，包括最新的 OSG 新闻，下载，编译和安装的技巧，OSG 成员提供的附加文档，示例程序，OSG 社区信息，OSG 社区所提供的 OSG 相关组件，支持信息等。

邮件列表

OSG 用户邮件列表将帮助你与其他 OSG 用户和开发者进行联系。如果你在编译 OSG 时，无法解决代码中的问题时，或者对于 OSG 内部机制的某些方面有

所疑问时，不妨发送邮件给其他的 OSG 用户，以获取大量有用的回复信息。要在 OSG 用户邮件列表中发布一条消息，请访问下面的地址。

http://www.openscenegraph.net/mailman/listinfo/osg-users

专业支持

由于 OSG 的成功发展，一些企业已经开始提供 OSG 的开发，顾问，培训和文档服务。不同的公司有不同的资源，费用和工作效率。有关这些服务的最新信息，请使用 OSG 邮件列表进行咨询，或者访问下面的地址。

http://www.openscenegraph.com/osgwiki/pmwiki.php/Support/Support

词汇表

.osg	这是一种基于 ASCII 的 OSG 自定义文件格式，用于保存所有的场景图形元素。
数据变量（Data variance）	这是一个 osg::Object 类的属性，用于指定程序是否要动态更改 Object 对象的数据。用户可以使用 Object::setDataVariance() 设置该属性，并传入 Object::DYNAMIC 或 Object::STATIC 参数。请参阅“Object 类”。
数据文件路径列表（Data file path list）	当用户程序尝试使用 osgDB 接口读取 2D 图形或者 3D 模型文件时，OSG 将搜索这个列表中所列的文件目录。
.OSG 封装（Dot OSG wrapper）	这是一个 OSG 插件库，用于实现 NodeKit 对.osg 文件的 IO 操作。
Drawable 类	osg::Drawable 类包含了将要进行渲染的几何数据。场景图形中的 Geode（参见“Geode 类”）类型的对象中往往包含了一系列的 Drawable。场景图形（参见“场景图形”）中包含了对 Drawable 的引用。
Geode 类	osg::Geode 类是 OSG 的叶节点。Geode 没有子节点，但是包含了一系列的 osg::Drawable 对象（参见“Drawable 类”），以及一个 osg::StateSet 对象（参见“StateSet 类”）。这个词是由“geometry”和“node”

	两个词组合而成。请参见“叶节点”，并参阅 Geode 头文件中的相应内容。
Group 类	osg::Group 类提供了对常见场景图形组节点概念的支持。它可以作为场景图形的组节点或者根节点。许多场景图形类都是从 osg::Group 派生的，以便实现对多重子节点的支持。参见“组节点”。
组节点（Group node）	组节点拥有子节点。并且组节点也拥有一个或多个父节点，根节点除外（参见“根节点”）。
叶节点（Leaf node）	此类场景图形节点没有子节点。在大部分场景图形中，叶节点中包含渲染数据，例如几何信息等。
LGPL 协议（Library GPL）	也就是通常所说的 GNU 宽通用公共许可证。它是 GNU 通用公共许可证的一个较宽松的版本，并且是 OSG 许可证的基础。
多管道渲染（Multipipe rendering）	这是一个并行的进程，可以将渲染的工作量扩展到多个显示卡或者系统上进行。在一个典型的多管道场景中，显示设备按照并排的排列方式或者保存在数组当中，每个图形卡负责渲染场景的一部分，并传递到一个显示设备中。
Node 类	所有 OSG 节点类的基类。请参阅 Node 头文件中 osg::Node 类相关的部分。
NodeKit	OSG NodeKit 是一个用于增强 OSG 核心库功能的模块，它可以向核心库添加新的场景图形节点类。

NodeVisitor 类	这个类用于遍历场景图形，并对遍历中遇到的每个节点执行用户操作（或者收据数据）。osg::NodeVisitor 类实现了访问器的设计思想[Gamma95]。
Object 类	这个纯虚类定义了一些基本属性和方法，可用于 Nodes，Drawables，StateAttributes，StateSets，以及其它 OSG 组件。
拾取（Picking）	用户与 3D 图形软件的常用交互方式。用户从渲染的图形中选择一个感兴趣的对象，这一过程通常通过指定鼠标光标掠过物体的位置，并且点击鼠标来完成。
插件（Plugin）	这个结构将允许符合标准接口的库或模块在运行时被自动加载。OSG 使用插件结构实现 2D 和 3D 数据的文件支持。符合 osgDB::ReaderWriter 中所定义的接口的链接库将被识别为 OSG 插件。用户程序通过 osgDB 库来实现对 OSG 插件的操作。
位置状态（Positional State）	这个渲染状态量包含了受当前变换矩阵影响的位置信息。例如，位置状态量中包括剪切平面和光源位置状态。
Pseudoloader	这个 OSG 插件提供了读取文件之外的一些附加功能。例如，变换 Pseudoloader 可以在读取文件的根节点之上添加一个 Transform 节点。

渲染图形（Render graph）	Drawable 及 StateSet 引用对象的集合。拣选遍历（cull）中将几何信息和渲染状态从渲染图形中传递给底层的图形硬件设备，以实现最后的显示工作。
渲染状态（Rendering state）	用于控制几何信息处理和渲染的内部变量。OSG 渲染状态由模式（布尔型变量，可选“允许”或者“禁止”，例如光照和雾效）和属性（配置渲染参量的变量，例如雾的颜色，图像混合方程等）组成。
根节点（Root node）	场景图形中所有节点的父节点。根据定义可知，根节点没有父节点。
智能指针（Smart pointer）	这个 C++类包括一个指针，并负责维护与其内存地址相关联的引用计数器。对于智能指针的一个实例，引用计数器将在构造函数中加一，在析构函数中减一。当引用计数达到零以后，相对应的内存空间将被释放。在 OSG 中，智能指针名为 ref_ptr<>。
StateSet 类	这个 OSG 对象用于保存渲染状态数据。它与 Node 和 Drawable 类相关联，可以共享以提高效率。在拣选遍历中，OSG 将按照 StateSet 的数据对 Drawable 对象进行排序。
条带化（Stripification）	这一过程将一系列隐含了共享顶点的独立三角形的集合转换成更高效且顶点明确共享的三角条带集合。

Viewer 类	这个 OSG 类负责管理场景中的一个或多个视口。Viewer 类也可以用于管理不同的渲染表面，例如窗口和帧缓存对象。Viewer 类同时还可以实现摄像机变换视口的控制，以及事件的处理。

参考书目

[ARB05] OpenGL ARB，Dave Shreiner，Mason Woo，Jackie Neider，Tom Davis：
《OpenGL® Programming Manual》，第五版，
Addison-Wesley，2005。

[Gamma95] Gamma，Erich，Richard Helm，Ralph Johnson，John Vlissides：
《Design Patterns: Elements of Reusable Object-Oriented Software》，
Addison-Wesley，1995。

[Martz06] Martz Paul：
《OpenGL® Distilled》，
Addison-Wesley，2006。

[OSGWiki] OpenSceneGraph 维基网站，
http://www.openscenegraph.org/

[Rost06] Rost Randi：
《OpenGL® Shading Language》，第二版，
Addison-Wesley，2006。