A Quick Introduction to

the Cross-Platform Open Source Scene Graph API

OSG 快速入门指南

Paul Martz 著

David.M 江湖浪子 译 V2.0

王锐 钱学雷 译 V1.3

2008 年 12 月 24 日

封面图像为 Mark Bryden 和依荷华州立大学 VRAC 研究所作品，尾封的两幅图像为 3DNature's NatureViewExpress 和 Visual Nature Studio with Scene Express 生成，尾封下部的图像为 Andes Computer Engineering 公司作品，特致谢意。

部分手稿的准备工作是由美国空军发起的，其联系方式为：FA8650-05-C-6537.

Paul Martz Skew Matrix Software LLC 284 W. Elm St. Louisville, CO 80027 USA

pmartz@skew-matrix.com

谨以此书献给 OSG 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 OpenSceneGraph2.0 版入门级指南。

本书首先介绍了场景图的概念，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、场景图结构、OSG 的状态属性和模式控制、较复杂的场景图系统、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，也就不可能有这本书的问世。

V1.2 版译者

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。

V2.0 版译者

David.M（江湖浪子），齐鲁人氏。毕业于 N.U.D.T.，控制科学与工程专业，系统仿真方向，现在

主要从事虚拟现实、仿真系统、模拟器材相关软硬件开发。此版本参阅了王锐的 1.2 版译本。限于译者水平，错误难免，欢迎指正。缘于开源精神，以期对同好有所裨益。

感谢 OSG 开发团队。希望有一天，人类的智慧可以共享；希望有一天，我们可以不用做许多重复劳动，可以站在巨人的肩上观察我们这个世界。选你所爱，爱你所选。

Mail: David_M@126.com，QQ：82280347

前　言

本书是一本关于 OpenSceneGraph（OSG）的简明教程。OSG 是一个跨平台的三维开源场景图程序开发接口（API）。特别要注意的是，本书基于 OSG V2.0 的版本。OSG 在 3D 应用程序开发的阶段划分中扮演着重要的角色。相对于低层次的 OpenGL 硬件抽象层（HAL）主要作用于底层硬件来说，它作为中间件（middleware）为应用软件提供了各种高级渲染特性、输入输入接口（IO）以及其他的空间结构管理功能。

多年以来，OSG 可阅读的文档资料仅仅是源代码。值得庆幸的是，OSG 的发布版本中也提供了一些示例程序，用于显示各种不同渲染效果的实现和 OSG 与终端用户软件的集成实现方法。相当部分的开发者通过这些示例程序，在编程、调试的过程中深入到 OSG 的核心，成长为熟练的 OSG 程序员。

尽管以往历史表明，源代码可以起到编程文档的作用，但是它仍然不能替代传统格式的编程教学文档。图形和表格是编程手册中最为常见且易于理解的教学手段，但是源代码几乎无法使用这些手段。随着 OSG 的迅速发展，其体系越来越复杂，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，这是我们不愿看到的情况。事实上，在这本书面世之前，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的开发工作。

缘于 OSG 社区其他人士的一再要求，Don Burns 和 Robert Osfield 开始撰写 OSG 图形库系统的文档。2006 年中期，Don 的一个客户，即计算机图形系统开发公司（CGSD），致电 Don 要求提供 OSG 图形库系统的文档，以方便程序开发。根据这个要求，Don 将这一文档编写工作交给 Paul Martz。Robert 建议第一本 OSG 的书籍应当免费，而且要通俗易懂。因此，《OSG 快速入门指南》诞生了。

这本书是一个简短的编程指南。它主要介绍了基本和核心的 OSG API 函数。它同时也是计划出版的 OSG 书籍系列的第一本，OSG 文档将随着该系列图书的编写逐渐变得完善起来。《OSG 快速入门指南》的编写目标如下：

- 向 OSG 入门开发者提供快速高效地掌握 OSG 的基础知识的途径；
- 使读者熟悉 OSG 发行版本和源代码结构；
- 介绍常用的 OSG API 函数及其用法；
- 为读者高效阅读 OSG 源代码提供相关文档支持。

基于开放源代码的理念，《OSG 快速入门指南》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平装印刷版本的方式，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购买此书。

http://www.lulu.com

本书印刷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。

无论您的图书是通过免费下载还是购买印刷版本，你的反馈意见对本书的修订来说，是及时而有帮助的。因此，请您将相关的修改意见发给 OSG 用户邮件列表中的人员。关于用户邮件列表的相关信息，请参见“附录：从这里开始”部分列出的信息。

关于本书的最新修订版本，请登陆 OpenSceneGraph 图书网站：

http://www.osgbooks.com/

上面的网址包括了获取本书最新版本的信息、书中示例程序代码下载以及其他一些相关的出版信

息。

➢ 适宜读者群

本书内容不多，相对 OSG 庞大的代码数量来说，而缩短其篇幅并是一件容易的事。由于本书的编写目标是对 OSG 的基本功能进行介绍，因此，本书的适宜阅读群体定位为 OSG 入门者。

本书专为准备开始学习 OSG、使用 OSG 的程序开发者所编写。本书所涉及的工具不包括一些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，例如结构树和链表。

在学习使用 OSG 编程之前，读者首先熟悉 3D 图形学。对于本书而言，读者需要熟悉 OpenGL 编程接口，而 OpenGL 是标准的跨平台底层 3D 图形 API 函数库。读者要求能够理解对不同类型的坐标空间，熟练应用笛卡尔三维坐标系来表示三维几何数据。读者需要了解一些纹理贴图的概念，不过并不需要对底层图形硬件的实现有很深入的认识。

同样地，读者最好还具备一定的线性代数知识，熟悉采用向量表示三维坐标的方法以及渲染过程中采用矩阵变换进行图形系统的相关操作，读者还需要了解矩阵相乘的有关知识。

➢ 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》由三章和附录组成。

第一章“**场景图与 OpenSceneGraph 概述**”，简要介绍了 OSG 发展的历史、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。本章还包括场景图概念的介绍、OSG 及其开发团队的介绍。

第二章为“**建立一个场景图**”。在本章中，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、场景图节点、几何体类、渲染状态（纹理贴

图及光照）等。本章还介绍了 osgText 场景图工具，该工具用于向场景快速添加文字的方法。同样地，本章还包括通过文件 I/O 保存场景图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、视点的定位与旋转、视点动画以及动态修改场景图的方法。

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，加入 OSG 的开发团队途径。

- **本书约定**

为方便阅读，本书采用了以下约定：

加粗：OSG 类名以及名称空间的名字，OSG 和 OpenGL API 函数，OSG 和 OpenGL 类型；

斜体：变量，参数名称，自变量，矩阵以及空间坐标（诸如 x，y，z 之类三维数据）；

等宽字体：代码列表，文字段落中的代码片段，枚举变量以及常量。

另外，对于网络地址 URL 采用等宽字体单独列出；采用*斜体*表示新名词术语。

- **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，喜爱一些常见的扑克牌游戏。

- **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 OSG 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。

1 场景图与OSG概述

本章主要阐述场景图（Scene Graph）的基本概念。通过本章的阅读，你可以了解到 OSG 的发展史和组织结构，对其能力有一个初步的了解，掌握如何获取和安装 OSG，学会利用一些简单的例子。本章将带你逐渐熟悉场景图和 OSG 的相关概念，但是本章将不包括编写 OSG 程序的具体细节，你可以在第二章和第三章的学习中获得此方面的知识。本章仅仅是一个概述，仅此而已。

1.1 OpenSceneGraph的历史

在 1997 年，DonBurns是SiliconGraphics（即SGI公司）公司的一名软件设计顾问。在业余时间，他喜爱滑翔运动。缘于他对计算机图形和滑翔机同样的热衷以及对高端渲染硬件的知识，他采用Performer[1]三维场景编程API，设计了一套基于SGIOnyx的滑翔仿真软件。

受其他滑翔爱好者的鼓励，使其开发的仿真软件可以在更廉价的硬件平台上运行，Don开始尝试使用Linux上的Mesa3D和 3dfx的Voodoo[2]设备。当这套软件提供了OpenGL支持的时候，场景图的概念还未能广泛应用于Linux。为了填补这一空缺，Don开始编写一个简单的、类似于Performer的场景图系统，命名为SG。SG的开发强调简单、易用。后来，SGI将performer移植到了Linux系统上，Don不再需要他的SG场景图系统，暂时停止了开发。

20 世纪 90 年代末期，Robert Osfield 在滑翔爱好者的邮件组中遇到了 Don。那时 Robert 为一个滑翔机制造商的设计顾问，对建模的倾斜稳定性非常有经验。两人开始为提高仿真器的性能开始共事。Robert 非常崇尚开源，并且建议 Don 将 SG 其做为一个单独的开源场景图工程继续开发。但由于 Don 满足于使用 performer，当时对继续开发 SG 的兴趣不大，Robert 就承担起了 SG 继续开发工作的主导地位。此时，工程的名称改为 OpenSceneGraph，有九人加入了 OSG 用户列表。

在 2000 年后期，Brede Johansen 为 OSG 做了第一个比较重大的贡献——他为 OSG 编写了 OpenFlight 插件模块。他那时供职于挪威康斯伯的 Kongsberg Maritime 船务仿真公司。他开发了基于 OSG 的 SeaViewR5 视觉仿真系统。

2001 年，Robert 开始将 OSG 开发做为一项全职工作，以 OSG 专业服务商的身份提供相关服务。那时，他设计并实现了今天 OSG 许多核心功能或特性。Glasgow 科技中心虚拟现实展示部门，Robert 的早期客户之一，一直致力于三维展示软件的开发，最终发展成了 Present3D 应用软件。

Don 跳槽到 Keyhole 科技公司，也就是今天 Google 的 GoogleEarth 部门，不过很快就在 2001 年辞职了。他也建立了自己的公司，名为 Andes Computer Engineering，位于加利福尼亚圣迭戈，主要是继续进行 OSG 开发。第一期 OSG 同好会在 SIGGRAPH2001 举行，那时只有区区 12 个人参加，其中听众中包括了来自 MagicEarth 的代表。他们的目标是寻求一个开源场景图系统来支持油气相关软件的开发。他们决定同时联系 Don 和 Robert，研究 OSG 的服务支持和后期发展，并成为了 OSG 的第一个收费用户。

---

[1] Performer，SGI 公司开发的场景图系统，运行于 Unix、Windows 等平台。MultiGen 公司（现为 Pregis）开发的 Vega 软件即基于 Performer 场景图系统。

[2] Voodoo 卡与 3dfx 公司：20 世纪 90 年代以前著名的 3D 显示加速卡制造商和 3D 标准提供者，后因业绩不佳被收购。

每一年，出席 OSG 同好会持续地增长。OSG 用户列表中的用户以显著的步速保持着较快增长，如图 1-1 所示。在本书付诸印刷时，订阅 OSG 用户列表的人数已经超过 1700 了。

OSG 功能和插件库同样保持着较快的增长。2003 年，OSG 通用库 Producer（最初称为 OSGMP）诞生，它是专门为 MagicEarth 开发的提供多管道渲染性能的模块；2004 年，大容量地形数据库调度、阴影功能被添加到 OSG 库中。2006 年，OSG 功能得到全面完善，重新改写了 OpenFlight 插件，与新添加的 osgViewer 一样，增强了场景的管理和视口渲染功能。

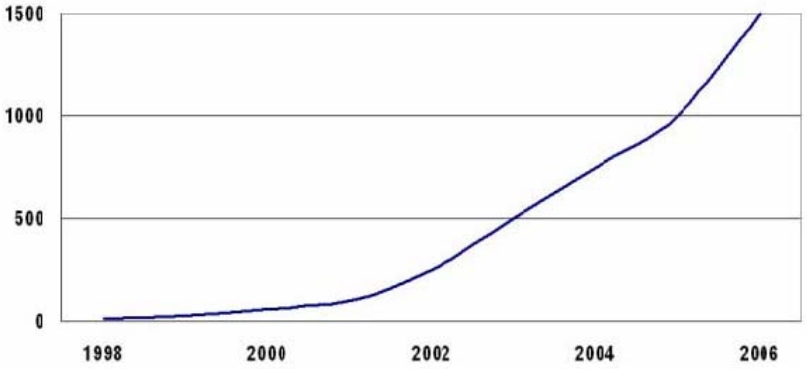


图 1-1 OSG 用户随时间变化得到了显著的增长

如今，不少高性能三维软件使用了 OSG 做为底层来管理渲染复杂的 2D 和 3D 场景，尽管大部分基于 OSG 的软件更适用于可视化设计和工业仿真。在使用 3D 图形的多个领域，已经出现了 OSG 的身影，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安装

前一节阐述了 OSG 起源。本节介绍如何获取和安装 OSG，以便读者学习运行 OSG 示例程序以及进行 OSG 应用程序的开发。OSG 一直处于不断的完善之中。OSG 社区持续修正 OSG 代码的漏洞，并不断添加 OSG 的新特性。本文档基于 OSG 2.0 版代码，阐述了以前版本中不存在的特性。更多信息请登录 OSG Wiki 网站[OSGWiki]查询。

OSG Wiki 网站提供了多种机制下载 OSG 安装版本以及 OSG 源代码的方法：

运行库文件：使用 OSG 运行库文件包、安装必要的链接库文件以运行 OSG 示例和程序。

OSG 源代码：OSG 的开发者应当持有 OSG 的源代码。OSG 提供了多种获取完整源代码的机制。你可以下载一份 OSG 稳定版本的压缩包，下载每日更新的 tar 文件包（一种文件压缩格式），或者使用版本控制系统（Subversion，SVN）签出获得最新的源代码。

第三方支持库：从源代码编译 OSG 时，某些可选的组件可能需要其他支持软件包的支持，例如 libTIFF，libPNG 等。如果你的系统还没有安装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。

下面的章节主要介绍了获取和安装 OSG 运行库文件的方法。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 OSG Wiki。

如果安装用的可执行文件无法在您的系统平台上使用，或者您计划建立自己的 OSG 开发环境，你可以选择编译 OSG 的源代码。关于 OSG 源代码、第三方支持软件和其他示例数据的获取方法，请登陆 OSG 维基网站 OSG Wiki。

1.2.1 硬件需求

OSG 可以在多种硬件平台和操作系统上运行，并且能够在当今主流配置的大部分计算机系统上正常使用。

处理器：OSG 可以在多数 CPU 上编译通过。OSG 具备线程安全性，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。

图形硬件：你的计算机系统需要配置一块 AGP 或者 PCI-E 总线的显示卡。OSG 可以在多数设计用于建模、仿真和游戏的专业级或大众级图形设备上运行。运行 OSG 的设备要求较好的 OpenGL 性能，使用时，应当尽可能地从设备提供商处获得最新的设备驱动程序。OSG 对显存的要求因用户而异，但是 **256MB** 是个比较合适的起点。OSG 可以在多管道（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：最小的系统内存需求是由显示数据的数量和类型决定的。1GB 是一个比较合适的入门级配置容量，对于更大型数据集的开发，可能需要更多的内存支持。

磁盘：和内存需求一样，磁盘空间的需求大小由数据量决定。对于任何程序来说，更快和更大容量的磁盘无疑可以减少数据读取时间。

1.2.2 苹果Mac OS X

运行于苹果 Mac OS X 系统的 OSG 是以.dmg 的格式存储的磁盘镜像文件，其中包括了运行库文件和完整的开发环境，可以从 OSGWiki 上获得。请查阅 OSG 苹果 QuickTime 文档[MacOSXTips]得到更多的细节。OSG 的安装步骤如下：

1.登陆 OSG 维基网站 OSGWiki，选择“Downloads”；

2.下载“OSG Universal Binaries for OSG V2.0”，它是一个.dmg 文件；

3.下载结束之后，加载该.dmg 文件；

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

5.在“/Library/Application”中，建立名为 OpenSceneGraph的新文件夹；将.dmg文件中的各种插件拖动至新的文件夹中（注意文件夹名称“PlugIns”的拼写，它是由大小写字母混合成的匈牙利表示法[3]拼写方式，某些系统需要保留此种样式）。

1.2.3 Fedora Linux

[3] 又称为骆驼式，因最早由一名匈牙利程序员使用而得名，该方式表达直观，便于记忆，在 C 编程中得到广泛的应用。

OSG 提供了对多种版本的 Linux 系统的支持，并提供了多种 Linux 环境的软件安装包供用户搜索、安装。例如，在 Ubuntu Linux 中，运行 Synaptic Package Installer 并搜索 OpenSceneGraph，很快就可找到并选择 OSG 运行库文件和开发环境的软件包，选择安装即可。

对适合 Fedora Core 4 的最新版本的 OSG 文件，可以登陆 OSG 维基网站 OSGWiki 选择下载。在 Binaries 栏目中选择“Fedora Core 4”的链接。

1.2.4 Microsoft Windows

适于 Microsoft Windows 的 OSG 运行库可以从 OSG 维基网站 OSGWiki 下载，它是一个采用 InstallShield 制做的安装包。其安装步骤如下：

1.登陆 OSG 维基网站 OSGWiki，选择“Downloads”；

2.下载“OSG Win32 Binaries”，它是一个.exe 文件；

3.下载完成后，双击.exe 文件并按照提示执行安装。

缺省的安装方式会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[4]。

1.2.5 检查OSG安装

OSG 安装完成之后，需要检查安装的正确性。其步骤如下：

1.打开系统的命令行窗口；

2.输入命令，

```
osgversion
```

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

OpenSceneGraph Library 2.0

这一步说明系统已经找到 OSG 的可执行文件（也就是说，系统 PATH 变量已经正确设置），并输出当前所安装的 OSG 版本号，同时最小程度地保证了 OSG 基本上是可以使用的。要检查 OSG 在本地系统上的渲染能力，执行以下的命令，

```
osglogo
```

输出的结果如图 1-2 所示。osglogo 通过动态更新场景，旋转标志中地球。同时，它也提供了对鼠标支持的接口，用户可以通过鼠标左键旋转并观察图形标志。

图 1-2 osglogo 命令的执行结果

[4] 此时实际上并不需要重新启动机器，只需要注销当前用户重新登陆，环境变量即可生效。

1.3 运行osgviewer

在上一节，通过运行 osgversion 和 osglogo 命令，检验了 OSG 的安装是正确的，但是 OSG 功能依然受到一定的限制。本节将介绍如何运行 osgviewer，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。

奶牛的模型是以 OSG 自己特有的.osg 格式存储的，osgViewer 支持许多类似于.osg 格式的模型文件，在本章的后续节中将对其一一列举。

如同 osgLogo 程序一样，osgViewer 允许用户与程序进行交互操作。osgViewer 默认支持类似于轨迹球（TrackBall）的运动模型用户接口。如果要旋转奶牛，你可以拖曳鼠标左键。当你释放鼠标键的时候，模型会继续保持转动。你也可以通过鼠标右键以拉近或拉远镜头的方式观察模型。点击空格键可以返回初始化时的状态。

图 1-3 osgViewer 打开 crow.osg 的运行结果

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 获得帮助

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，也就是鼠标对摄像机位置的操纵方式[5]。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。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。

[5]摄像机控制模式相当于 VegaPrime 中的运动模型，MotionModel。后文中均以运动模型来翻译，David 注。

--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
```

--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 采用了热键的形式进行不同显示模式的相互切换。下面列出了部分常用的控制命令。

- *多边形模式（Polygon mode）*：按下“W”键（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（t）可以切换显示或者关闭模型的贴图。
- *光照（Lighting）*：按下“L”（l）键决定取消或者启用光照。
- *背向面剔除（Backface culling）*：按下“B”（b）键启用或停止背向面的剔除。对于 cow.osg 模型而言，此选项基本不改变其外观，但是该命令可能影响其它一些模型的外观和渲染性能。
- *全屏模式（Fullscreen mode）*：“F”键（f）进行切换全屏幕模式和窗口渲染模式的切换。

读者可以花一点时间来测试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 程序支持很多环境变量，不过你至少需要非常熟悉其中的两个。在使用 OSG 进行设计时，你会经常用到这两个环境变量，即：

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 的安装包时，系统将自动设置 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“;”来分隔各个目录，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 只在当前目录中读取图形和模型文件。

调试信息显示

OSG 可以将大量的调试信息输出到 std:cout。这在开发 OSG 程序时十分有用，开发人员可以借此观察 OSG 正在执行何种操作。环境变量 OSG_NOTIFY_LEVEL 控制着 OSG 调试信息显示。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，使用者可以根据对调试信息的需要对此变量进行调整。

1.3.4 统计信息显示

特别地，键盘的“S”键用于程序的性能测试，它使用 osgViewer 类库的 Statistics 类来收集和显示与渲染性能有关的信息。循环按下“S”键（小写 s）在以下四种不同的显示方式之间切换：

1. *帧速率*（Frame rate）：osgviewer 将显示每秒钟渲染的帧数（FPS）。
2. *遍历时间*（Traversal time）：osgviewer 将显示每一次遍历更新（update）、剔除（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*（Geometry information）：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。一般来说，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。该图说明了场景更新（update）、剔除（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性能问题分析、解决程序渲染性能瓶颈的手段。

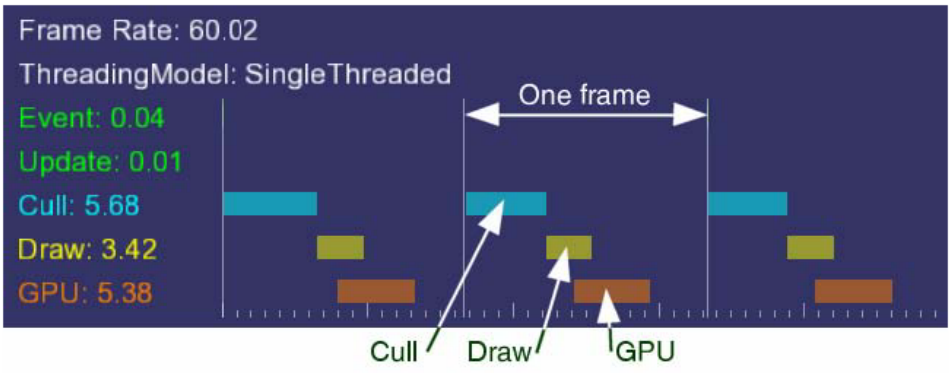


图 1-4 程序统计信息的显示

图 1-4 显示了一个典型的、使用了 60Hz 显示器运行 OSG 程序的遍历时间统计信息。事件和更新（update）操作所用的时间因为过于短暂，此结果没有在图中显示出来。不过剔除（cull）和绘制（draw）所用的时间分别为 5.68 毫秒和 3.42 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。图中最后一行显示了 GPU 每个周期计算使用了 5.38 毫秒，这是采用 OpenGL 测量的结果。

1.3.5 路径动画生成

开发者往往需要对程序进行反复的测试，以便有效地调整和检测程序渲染性能。为了简化性能调整，osgviewer允许用户轻松地记录摄像机运动的位置序列并且进行回放。这一序列被称为路径动画（animation path）[6]。

运行 osgviewer 时，按下小写“z”键将立即开始记录动画路径。此时可以使用鼠标旋转和缩放模型。然后，按下 “Shift +z”键将停止动画路径的记录，并立即开始回放。回放时可以看出，OSG 采用记录的摄像机运动路径来进行模型的显示，也就是当前视点运动所观察到场景的情况。

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，注销当前用户并重新启动 osgviewer 即可看到。

1.4 编译OSG应用程序

要生成基于 OSG 的应用程序，需要建立一个包括头文件和链接库文件的 OSG 开发环境。OSG 运行时文件中已经包括了头文件和优化的链接库文件。如果读者需要创建 Debug 版本的动态链接库，你需要下载并重新编译 OSG 的源代码。OSG 的源代码均可以在 OSG 维基网站的下载区[OSGWiki]上取得。OSG 维基网站还包括了如何生成 OSG 的示意文档。

要正确编译基于 OSG 的程序，需要告诉编译程序和链接程序如何找到 OSG 头文件和链接库的路径，还要告知连接程序 OSG 库文件的名称。对于不同的操作系统来说，其操作是不相同的。下面段落列出了一些常见操作系统下 OSG 的配置方法。

如果你没能正确配置好编译和连接程序，应用程序编译时就有可能会出错，出现诸如：“unable to open include file”，“unable to find library file”，“unresolved symbol”之类的错误。如果你遇到了该类错误，检查最近的出错信息，并校验是否已经正确对编译环境进行了配置。

1.4.1 苹果Mac OS X

OSG 的 Mac OS X 版本包括了一个 XcodeTemplates 的文件夹。该文件夹包括了使用 Xcode 开发环境编译 OSG 应用程序的模板，读者借助它可以快速构建一个 OSG 应用程序框架。读者安装该模板时，建议首先阅读一下该文件夹中 TemplateNotes.rtf 文件。更详细的介绍可以参阅 OSG 的 Apple QuickTime 文档。

模板安装完成后，打开 Xcode，点击 create a New Project。在 New Project 对话框中，从列表框中选择 OSG Application，模板会使用 Xcode 框架生成一个配置好了头文件和链接库环境的 OSG Xcode 工程，它同时还创建了一个 main.cpp 文件。该工程已经创建了最基本的 OSG 渲染环境，它已经可以正常

[6] 摄像机路径动画也称为视点运动动画，它实际上是视点位置变化而形成的场景变换动作，实体的运动是相对于人眼而言的，相对场景来说并没有产生运动，所以此处的动画只能称为路径动画或视点路径动画。

编译和运行。你现在所需要做的，就是向该工程中添加你需要的特性代码而已。

1.4.2 Fedora Linux

在小红帽 Linux 或其他的 Linux、Unix 环境中，编译 OSG 或 OSG 的应用程序需要使用 g++编译命令。在 g++的命令行模式中，通过-I 选项指定 OSG 的头文件所在的路径。一般来说，OSG 安装在/usr/local/include，所以 g++命令行环境下的 include 路径应当与此类似：

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

与 OS X 相似，在 Linux 下也需要指定 OSG 库文件的位置。默认 OSG 库文件安装在/usr/local/lib 路径下，连接程序可以自动找到。如果您的 OSG 安装路径指定了别的位置，可以使用 g++的-L 选项重新指定 OSG 库文件的路径。

最后，告知连接程序需要哪些 OSG 的库文件。正如 1.6.3 组件一节中描述的那样，OSG 由多个不同类型的库文件组成，每个库文件提供了不同的功能。如编译一个需要 osgViewer、osgDB、osgUtil 或 osg 库的应用程序，需要在 g++命令行模式中指定。

```
-losgViewer -losgDB -losgUtil -losg
```

上面列出的库文件仅是一个举例，具体你的程序需要哪些动态库，与具体使用了哪些库有关。你可能会需要其他库的支持，如 osgText，osgShadow 或 osgGA 库，有兴趣的读者可以自行尝试。

1.4.3 Microsoft Visual Studio

微软的 VisualStudio 开发环境允许创建多种类型的应用程序。创建 OSG 应用程序的最简单的方法是创建一个 Win32 的控制台程序。在 VisualStudio8 以前，许多编译器与 OSG 并不兼容。打开 Project Properties 对话框，检查动态类型信息启用（Enable Run-Time Type Info）是否为打开。同样地，设置运行库（Runtime Library）为 Multi-threaded DLL，对应的 DEBUG 版本应当是 Multi-threaded Debug DLL。在 Project Properties 对话框中为编译器设定 OSG 头文件的路径。OSG 默认的安装路径是 C:\Program Files，因此，在 C/C++选项卡的 Additional Include Directories 添加的路径应当如下：

```
C:\Program Files\OpenSceneGraph\include
```

同样地，还需要告诉连接程序 OSG 库文件的路径。在 Additional Library Directories 的属性中添加下列的内容：

```
C:\Program Files\OpenSceneGraph\lib
```

最后，添加需要使用的 lib 库文件。添加的位置在 Project Property 对话框的 Linker 选项卡的 Additional Dependencies 属性中添加。在 Windows 操作系统中，OSG 编译的发行版与调试版的库文件名称不同。在发行版本中，使用类似于 osgViewer、osgDB、osgUtil、osg 库这样的名称定义方式，添加时如下所示：

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

对调试版来说，其命名规则是在扩展名前添加‘d’来结尾，如下所示：

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

上面列出的库文件名称仅是为举例，读者的程序可能使用的库与此不同，有可能会用到如 osgText，osgShadow 或 osgGA 库，使用时自行添加即可。

1.5 场景图（Scene Graphs）介绍

前几节的内容主要介绍了关于 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。本书剩余部分将更深一步地对 OSG 进行探讨。本节从概念的层次对场景图进行介绍；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”、第三章“在用户程序中使用 OSG”，将对 OSG 应用编程接口部分的内容作进一步阐述。

为提高渲染效率，场景图采用类似于继承树的数据结构来管理空间数据。图 1-5 显示了包括地形、一头牛、一辆卡车的场景图概念的抽象。

场景图树以最高层的根节点作为起点。从根节点开始，组节点中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。在场景图的最底部，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同样的椅子。你可以采用多种方法来构建它的场景图。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，其上层的组节点将这把椅子变换到两个不同的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。

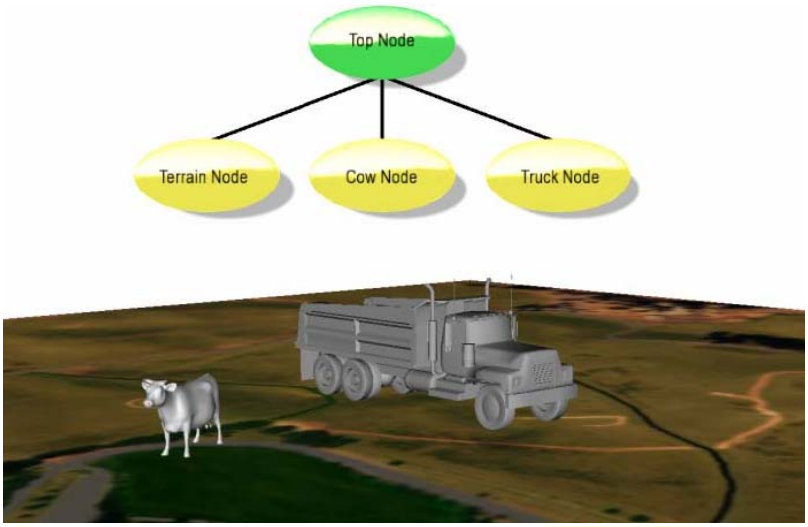


图 1-5 场景图示例

渲染一个包含了地形、奶牛和卡车的场景时，场景图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。

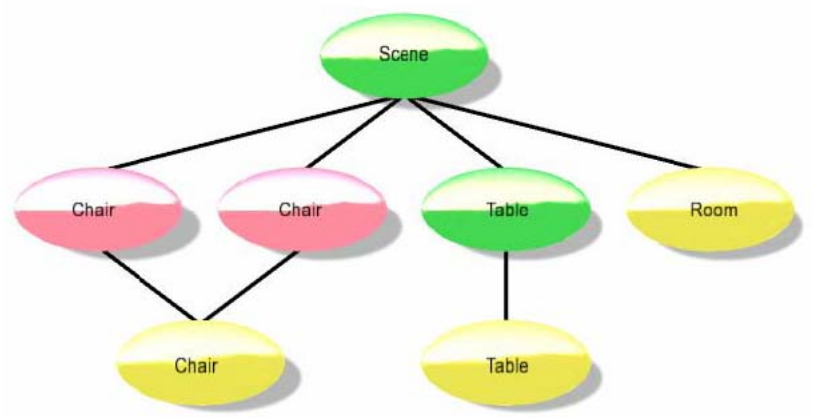


图 1-6 场景图的可能组织方式

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。

场景图通常包括了不同类型的节点以执行各种各样的用户功能，例如，开关节点(switch)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(Transform)可以改变子节点几何体的坐标变换状态。场景图是面向对象的，使用继承的机制来提供节点多样性，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。

OSG 通过定义多种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。

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

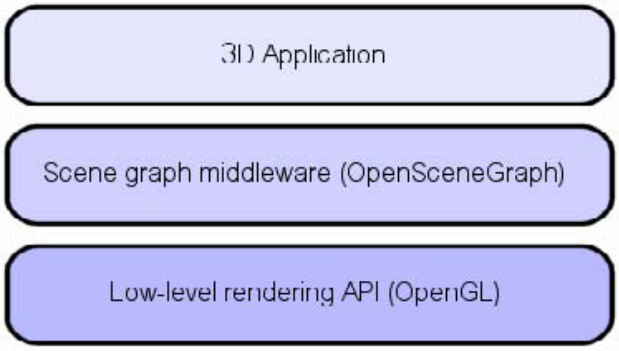


图 1-7 典型的 OSG 程序层次结构

一般来说，多数 3D 应用程序开发需要的并不是直接访问底层渲染 API 的接口，而是需要类似于 OSG 等中间函数库开发包中提供的高级功能。

1.5.1 场景图特性

场景图除了提供类似于底层渲染 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：

• 空间数据管理：场景图所采用的树状数据结构更直观，也更便于人们理解。

• 场景剔除：使用目标机器的 CPU 上进行投影剔除（frustum culling）和隐藏面剔除（occlusion culling）来减少系统总体负担，即在忽略最终渲染图像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。根据观者者的视点位置变化，场景数据会动态地从磁盘加载，而景物一旦超出视点的观察距离，又会自动地从内存中移除。

• 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。场景图一般都会提供上述这些操作。

• 状态改动最小化：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数据结构操控三维动态数据。场景图也是一种高效的文件格式转换的方式。

• 更多高效函数：除了底层 API 提供的基础功能之外，场景图库还提供了高效率的函数，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设备、渲染显示设备的访问。

几乎所有的 3D 程序都需要其中全部或部分特性。因此，直接使用底层 API 的开发者不得不自行编写实现其中的某些功能，从而增加了研发投入。使用现成的、支持上述特性的场景图 API 函数库，将有助于实现快速的程序开发工作。

1.5.2 场景图渲染方式

不管是多小的场景图系统，都具备允许程序保存几何体并执行绘图遍历，此时所有场景图中的所有几何体以 OpenGL 指令的形式发送到硬件设备上。但是该执行机制缺少前几节介绍的诸多高级特性。为实现动态的几何体更新、剔除、排序和高效渲染，场景图需要提供的不仅仅是简单的绘图遍历。一般来说，场景图至少应当支持以下三种需要遍历的操作：

• 更新（update）：更新遍历（有时也指作程序遍历）允许程序修改场景图以实现场景的动态更新。更新由程序或者场景图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。

• 剔除：在剔除遍历中，场景图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），场景图将遍历由剔除遍历过程生成的几何体列表，通过调用底层 API，实现几何体的渲染。

实际上，OSG 还包括第四种遍历——事件遍历，主要在每帧更新之前处理输入和其他事件。图 1-8 表示上述三类遍历。

特别地，这三类遍历通常要求每帧都需要执行一次。但是，许多情况下同时需要同一场景的多个视口，立体显示和多屏显示系统是两个典型的例子。在这些情况下，更新遍历每帧都需要执行一次，但是剔除和绘制遍历在每个视口的每帧中都需要执行一次。（也就是说，在简单的立体显示系统中，每帧至少要绘制两次；而多屏显示系统需要每视口每帧执行一次）。这就要求 OSG 必须提供对多处理器和多显示卡系统并行进行场景图处理的支持。为支持多线程访问，剔除线程必须为只读操作。

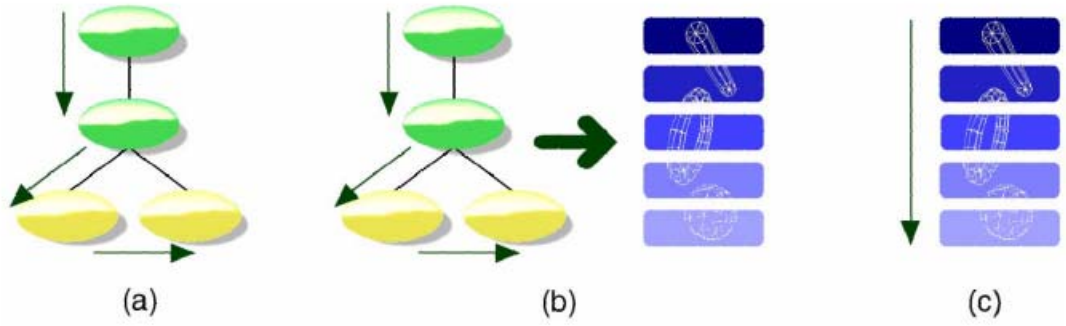


图 1-8 场景图遍历方式

1.6 OpenSceneGraph概述

OSG 是一套开源图形库，主要为应用程序提供场景管理和优化的三维图形渲染性能。OSG 以可移植方式编写。

场景图形的渲染通常分为三个阶段：

（a）更新遍历需要修改几何体、渲染状态以及节点参数，以确保当前帧的场景图保持最新；

（b）剔除遍历检查几何体的可视性，在一个新数据结构中定位几何体和对几何体状态的引用（OSG 内部称为**渲染图形**）；

（c）绘制遍历过程中遍历渲染图形，发出绘制命令在硬件上执行。

ANSI C++采用了工业标准的 OpenGL 底层 API，因此，OSG 可以在 Windows、Mac OS X 和多数 UNIX、Linux 操作系统等平台上运行。多数 OSG 程序独立于本地化 windows 操作系统。然而，OSG 也包含了某些特定操作系统相关的内容，如输入输出设备、窗口创建以及缓存管理等。

OSG 是公开源代码的，用户许可方式为修改过的 GNU 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：

- 高品质：OSG 由 OSG community 的诸多成员反复进行修订、测试和完善。直接参与 OSG 2.0 的开发人员已经超过了 260 人。
- 高质量：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源，相关运行信息就难以获得，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，充分掌握代码内部信息，提高软件开发效率。
- 低费用：开源意味着免费，除了最初购买开发软件所需的费用。

• 无产权纠纷：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。

通过订阅邮件列表可以方便地获得 OSG 支持，或者与专业的服务机构联系。更多信息，可以从“附录：从这里开始”找到。

1.6.1 设计与结构体系

OSG 设计时，注重了较高的可移植性和可扩展性，因此，OSG 适用于多种硬件平台，并可以在多种不同的图形硬件上进行高效的、实时的渲染。 显然，OSG 正是为高灵活性、可扩展性而设计，可以适应不同时期的设计和应用需求。可以看出， OSG 已具备了满足各类客户的、不断增长的需求的能力。为达到以上设计理念，OSG 采用了以下设计概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所需的平台上使用 OSG 进行开发，并且依据用户指定的平台对 OSG 进行移植。

1.6.2 名称约定

下面几行列举了 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：静态变量和静态函数的名称使用 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 OSG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。这些链接库可分为下面五类：

• OSG 核心库。它提供了基本的场景图和渲染功能，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。

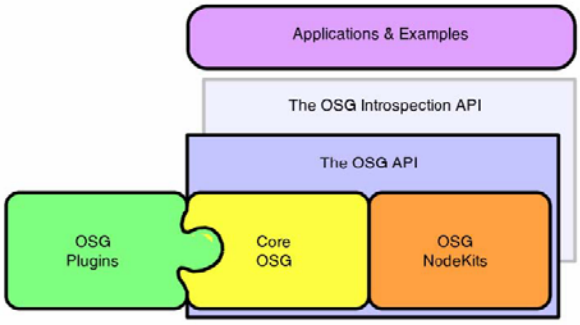


图 1-9 OSG 体系结构图

OSG 核心库提供了应用程序和 NodeKits 所需的功能模块，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 插件的接口，为用户提供了 2D 和 3D 文件 I/O 的接口。

1.6.3.1 OSG核心库

OSG 核心库提供了用于场景图操作的核心场景图功能、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。OSG 核心库主要包含了以下四个链接库：

- **osg 库**：osg 库包含了用于构建场景图的场景图节点类、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 库中还包括一些三维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 工具库包括可以用于场景图及其内容操作的类和函数，可以进行场景图数据统计和优化以及渲染器的创建。它还包括了几何体操作的类，例如 Delaunay 三角化（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、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 插件类的注册管理以及用于访问这些插件的特定功能类。osgDB 地形数据管理（database pager）可以支持大型地形数据的动态读取和卸载。
- **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 集成到多类别的视窗系统中。

OSG 的 2.0 版本还包括了用于处理界面事件的 osgGA 库。但不久以后 OSG 重新设计其 osgGA 模块，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。

a. Osg 通用库

osg 库是 OpenSceneGraph 的核心部分。它定义了场景图组成的核心节点，还有帮助用户进行场景图管理和应用程序开发的一些附加类。下面将对其中一些类进行简要的叙述。本书的第二章将对其更为详细地介绍，并指导如何在 OSG 程序开发中使用。

➢ Scene Graph 类

场景图(Scene graph)类用于辅助场景图的构建。OSG 中所有的场景图类都 osg::Node 派生。从概念上讲，根节点、组节点和叶节点是完全不同的节点类型。在 OSG 中，它们完全都派生于 osg::Node，只不过特定的类会提供不同的场景图功能。此外，OSG 中的根节点并不是特殊节点类型，它仅仅是一个没有父类的 osg::Node 类。

表 1-1 osg 链接库属性

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

• Node：Node 类是场景图中所有节点的基类。它包含了用于场景图遍历、剔除、程序回调以及状态管理的方法。

• Group：Group 类是所有具有子节点的基类。它是场景图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 类根据视点与场景图子节点的距离选择显示哪一个子节点。通常使用它来创建场景中物体的多个显示层级。

• MatrixTransform：MatrixTransform 类包含了用于实施当前节点及其子节点空间转换的矩阵，可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，可用来允许或禁止对场景图子节点的操作。

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。

➢ Geometry 类

Geode 类是 OSG 场景图的叶节点，它包含了实时渲染用的几何数据。使用下面列出的类可以实现 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，不能直接实例化。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 double 类型。OSG 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，需要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 类是场景图的叶节点，保存了 Drawable 类对象；Geometry 类（一种 Drawable 类型）保存了顶点数组数据和及其对应的顶点数组的渲染指令；数据组合成为向量数组。第二章“建立场景图”将涵盖上述所有的内容，并进行详细的讲解。

- **状态管理类**

OSG 提供了一种机制，用以保存场景图所需的 OpenGL 渲染状态。在剔除遍历中，处于同样状态的几何体将被组合，尽量使场景状态改变量最小化。在绘制遍历中，状态管理代码将记录当前状态变化情况，尽量清除冗余的渲染状态变更。

和其它场景图系统不同，OSG 允许渲染状态与任何场景图节点相关联，在遍历中，状态具有鲜明的层次继承关系。

• **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()类似，模式用于打开或关闭 OpenGL 特定功能（fixed-function）的渲染管道，例如灯光、混合和雾效。可能通过使用 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存储状态参数，例如混和函数、材质属性、雾颜色等。使用方法 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于 OpenGL 多重纹理的特定纹理单元。和 OpenGL 不同，OSG 不存在缺省的纹理单元，应用程序必须支持可设置纹理模式和属性的纹理单元。StateSet 类的方法 setTextureMode()和 setTextureAttribute()可以进行状态参量以及纹理单元信息的设定。

• **继承标志**：OSG 提供了一些标志量用于控制场景图遍历中的状态值。缺省情况下，子节点中的状态设置将重载父节点的状态集合。但是也可以强制父节点状态重载了子节点的状态，或指定保护子节点的状态而不重载其父节点的状态信息。

上述 OSG 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。2.4 节渲染状态非常详细地描述了状态相关的问题。

- **工具类和其他相关类**

除上述类之外，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），这种策略可以通过释放引用计数为零的内存以避免出现内存泄露。第二章“建立场景图”将详细讲解内存引用计数的内容。

• Referenced：Referenced 类是所有场景图节点和许多其它 OSG 对象的基类。它包括用于跟踪内存使用情况的引用计数（reference count）。如果某个从 Referenced 派生的对象，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释放对象分配的内存。

• 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：为纯虚类，Object 是 OSG 中一切需要 I/O 支持、拷贝和引用计数对象的基类。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。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 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。

b. osgUtil 库

很显然，osgUtil 库集成了许多用于场景图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、剔除和绘制遍历的类。在典型 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。

表 1-2 osgUtil 链接库属性

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

➢ 相交测试

一般来说，3D 应用程序需要为用户提供交互和选择的功能，比如三维场景对象拾取。通过提供相对高效进行场景相交运算类，osgUtil 库可以高效地进行拾取操作。

• Intersector：Intersector 是一个纯虚类，它定义了相交测试的接口。osgUtil 库从 Intersector 派生了多个类，每类适用于不同类型的几何体（线段、平面等）。执行相交测试时，应用程序将继承自 Intersector 的相应类实例化，传递给 IntersectionVisitor 的实例，随后请求该实例返回数据以获取相交运算的结果。

• IntersectionVisitor：IntersectionVisitor 类，主要用于检索场景图中与指定几何体相交的节点。实际的相交测试的工作将在从 Intersector 派生的子类中完成。

• LineSegmentIntersector：LineSegmentIntersector类继承自Intersector类，用于检测指定线段和场景数据之间的相交情况，并提供了相交测试结果的查询函数。[7]

• PolytopeIntersector：与 LineSegmentIntersector 类似，该类用于检测由一系列平面构成的多面体与场景图的相交情况。该类在用户希望拾取到鼠标位置附近封闭多面体区域时，PolytopeIntersector 类尤其有用。

• PlaneIntersector：与 LineSegmentIntersector 类似，这个类用于检测与一个由一系列平面构成的平面的相交情况。

➢ 优化 Optimization

从理论上看，场景图的数据结构在有助于实现优化和数据统计工作。osgUtil 库包含了一些特定的

[7] 该处存在疑惑。是究竟与场景图相交还是与其他的实体相交？

类，这些类通过遍历和修改场景图，可以方便地实现渲染流程的优化和场景统计信息收集的工作。

• Optimizer：正如其名字所示，Optimizer 类用于优化场景图，其动作通过一系列枚举标志控制。这些标志中，每一个标志都表示一种特定的优化方式。例如，FLATTEN_STATIC_TRANSFORMS 标志是指通过使用非动态 Transform 节点来变换几何体。具体来说，就是通过消除对 OpenGL 的 model-view 矩阵堆栈的修改，实现场景的渲染优化。

• Statistics 和 StatsVisitor 类：为高效地设计 3D 应用程序，开发者必须对将要渲染的对象有尽可能多的了解，这两个类就是为这个目标设计的。StatsVisitor 类可以获得场景图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。

➢ 几何体处理（Geometry Manipulation）

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 库设计了许多支持一些通用的几何形体运算的类。

• Simplifier： Simplifier 类可以减少 Geometry 对象中几何体的数目，这对于低细节的 LOD 模型自动生成有帮助。

• 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，根据一组顶点集合生成一系列三角形。

• 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 类可生成单个顶点的法线，也就是所有共享此顶点的面的法线的平均值。

• 纹理贴图生成：osgUtil 库包含了帮助建立反射贴图、半向量（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。

表 1-3 osgDB 链接库属性

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

c. osgDB 库

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

法性。

OSG 支持一种特有的文件格式，即.osg 格式。.osg 文件是对场景图的一种 ASCII 码文本描述，而.osga 文件是一组.osg 文件的集合（或组）。osgDB 库包含了以上文件格式的支持代码。同时，OSG 还支持一种二进制的.ive 格式。

由于大型的 3D 地型数据库通常是单块创建的地形文件的集成到一起的组合体。这种情况下，应用程序从文件中读取各部分数据库信息的过程，应当在不干扰当前渲染过程的前提下以后台线程的方式进行。osgDB::DatabasePager 提供了这样的功能。

d. osgViewer 库

osgViewer库定义了数个相关视口类，提供了将 OSG集成到许多窗体设计工具中的功能，包括AGL/CGL，FLTK，Fox，MFC，Qt，SDL，Win32，wxWindows以及 X11 等窗体设计系统。这些视口类支持单窗口/视口的程序，也支持使用多个视口和渲染通道[8](render surfaces)的多线程程序。每个视口类都可以提供对摄像机运动、事件处理以及对osgDB::DatabasePager的支持。osgViewer库包含了以下两个可能用到的视口类。

表 1-4 osgViewer 链接库属性

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

• **Viewer类**[9]：Viewer类可用于管理多个同步摄像机，用于同一个视景的多显示器渲染输出。与底层图形系统的能力相关，Viewer可以创建一个或多个同一个场景的显示通道和图形上下文，因此使用单一视口的程序也可以在单显示通道或者多显示通道的系统上运行。

• **CompositeViewer 类**：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。如果指定了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 可以用来创建平视显示器（HUD）、预渲染纹理（prerender textures），也可以用于在同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。

1.6.3.2 NodeKits

NodeKits 扩展了 Nodes、Drawables 和 StateAttributes 对象的概念，也可以看作是 OSG 内核中 osg 库的一个扩展。NodeKits 不仅仅是对 OSG 库的继承，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 包含有七种 NodeKits，即：

• **osgFX 库**：此类 NodeKit 提供了额外的场景图节点，以便于特效的渲染，例如异向光照（anisotropic

[8] Render surface，这里理解为通道是否可以？Window 指窗口，Viewer 指视口，在同一视口中可以渲染多个不同的显示位置，类似于 VP 的通道概念，这里暂且如此理解。

[9] 早期版本中还有 SimpleViewer 类，该类主要负责管理单一场景图中的单一视口。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）。2.0 版本不再支持？

lighting)、凹凸贴图、卡通着色等。

• **osgManipulator 库**：NodeKit 包括了多个用于在场景图中处理选择实体（对象）的类。

• **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 提供了仿真系统和 OpenFlight 数据库所需的空间渲染需求，例如地形高程查询类、探照灯效果（lightPoint）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 提供了支持阴影特效渲染的框架结构。

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 的进一步探索使用产生浓厚的兴趣。

1.6.3.3 OSG Plugins

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 可以自动管理插件库。只要插件确实可用，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。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，或者应用程序通过操作整个场景图结构只加载部分场景图，实现设计目标。

OSG 2.0 版支持大量常用的二维图形文件格式，包括.bmp，.dds，.gif，.jpeg，.pic，.png，.rgb，.tga 和.tiff。OSG 还支持读取视频文件的 QuickTime 插件以及读取 FreeType 类型的字体的插件。

更重要是，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 还定义了专用的文件格式。其中，.osg 格式是场景图的一种 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 格式则是一种经过优化的二进制格式，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。

OSG 插件集中还具有一个可以基于广域网进行文件读取的.net 插件。

最后，OSG 还包含了一组名为**“PseudoLoader”**的插件。该插件集提供了除简单文件读取之外更多的功能。

• **缩放、旋转和平移**： PseudoLoader 读取文件后，在已读入场景图根节点之上添加一个 Transform 节点，并指定放缩、旋转和平移属性的值对 Transform 节点进行了配置。

• logo 标志类：标志类的 PseudoLoader 允许在已读入 3D 场景之上加载一个类似于平视显示器样式的图片文件。

更多关于在程序中使用 OSG 插件的方法，第 2.5 节“文件 I/O”将提供介绍。

互操作性

用户可以在任何支持 C++链接库的编程环境中使用 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访问接口。

osgIntrospection 库允许软件使用**反射式和自省式（**reflection and introspection**）**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 类和类方法对 OSG 类型、枚举量和方法进行访问，而无需了解 OSG 编译和链接时的具体过程。

Smalltalk和Objective-C等语言包括了内建的反射式和自省式[10]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）[11]。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，应用程序不需要直接与这些OSG的封装库交互，它们将完全由osgIntrospection进行管理。

因为有了 osgIntrospection 及其封装，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，请访问 OSG 维基网站[OSGWiki]->Community 页，并选择 LanguageWrappers。

应用程序和示例

程序与示例 OSG 发行版包含了五个常用的 OSG 工具程序，它们对于调试和基于 OSG 的软件开发十分有帮助。

- **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压缩包的分解和内容列表查询。
- **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。（自从 OpenSceneGraph2.0 后，osgdem 已从 OSG 中移出，作为单独的一个部分 VirtualPlanetBuilder 来发展。随着 VirtualPlanetBuilder 的不断发展，最终有可能成为 OSG 处理地形的专用工具。）
- **osgversion：**这个程序将当前 OSG 版本以及一些记录了 OSG 源代码改动情况和作者信息送入 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。

---

[10] 反射式和自省式 reflection and introspection？未见到相关的参考文献，待查证

[11] MeteData: The data which defines the form of data stored in a database. Metadata can be thought of as “data about data.”定义存储在数据库中数据的形式的数据，可认为是关于数据的数据

2 建立一个场景图

本章将为你展示如何编写代码以建立一个 OSG 场景图应用程序。其内容包括了场景图构建的具体细节以及 OSG 从硬盘上加载三维模型文件并进行场景构建的机制。

第一部分为OSG内存管理。场景图及其相关数据消耗大量内存，本章将讨论OSG避免悬挂指针[12]（dangling pointers）和内存泄露的机制。

最简单的场景图程序由只包含了一个具有几何体（geometry）和状态（state）信息的节点组成。第 2.2 节“几何节点（Geode）与几何体信息”将介绍几何体、法向量以及颜色的设置方法。后续部分中，读者可以学习到如何通过改变 OSG 状态（state）属性和模式来控制几何体外观的方法。

众所周知，实际应用程序比单节点场景图系统要复杂得多。因此，本章也将对 OSG 的节点系列进行简单的介绍。不同的节点为场景图提供了各种各样的特性，并且已经封装在各个场景图运行库中。

多数应用程序需要从 3D 模型文件中读取几何体信息。本章也会对 OSG 的文件读取接口进行简介，通过该接口，可以实现对多种 3D 文件格式的支持。

最后，本章还包括了添加文字显示到应用程序的示例。OSG 在其节点工具（NodeKits）中封装了很多高级功能模块，本章将对其中的 osgText 模块详加讲解，以期达到抛砖引玉、举一反三的目标。

2.1 内存管理

在构建场景图应用程序之前，读者需要理解 OSG 节点和数据管理的内存管理机制。对这一概念的掌握，有助于编写清洁的代码，避免出现悬挂指针和内存泄露。

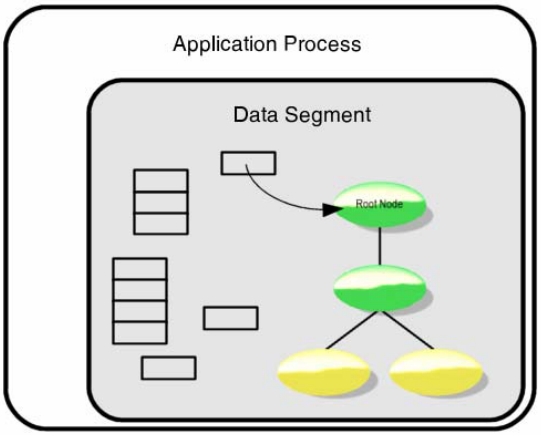


图 2-1 内存指针与引用计数

通常来说，OSG 应用程序一个保存了场景图根节点地址的指针，实现对整个场景图的引用（reference）。也就是说，应用程序并不保存场景图中其它节点的指针，所有其它节点均通过根节点直接或者间接地被引用。

前一章中我们已经对最简单的、继承于单个根节点的场景图进行了图示讲解。对典型的 OSG 使用模式来说，程序通常会保存一个指向根节点的指针，而不会保存场景图中其它节点的指针。通过根节点将直接或者间接地“引用”（reference），实现对场景图中所有节点的访问。图 2-1 描述了这一典型的使用模式。

[12] 悬挂指针?待查。

当程序不再使用场景图时，各场景图节点申请的内存需要及时释放以避免出现内存泄露。如果自行编写场景图遍历和依次删除所有节点及数据的代码，可以看出，该项工作极其乏味，而且容易出错。

幸运的是，OSG提供了一种自动的“内存回收机制[13]，它采用一种称为内存引用计数（reference counted memory）的方式工作。所有的OSG场景图节点均采用引用计数（reference count）的方式，当某节点引用计数值减为 0 时，此对象将被自动释放。因此，正如如图 2-1 所示，在需要删除场景图数据时，程序只需要简单地释放指向场景图根节点的指针。这一动作将引发连锁的后果，会将场景图中的所有节点和数据逐一释放，具体过程如图 2-2 所示。

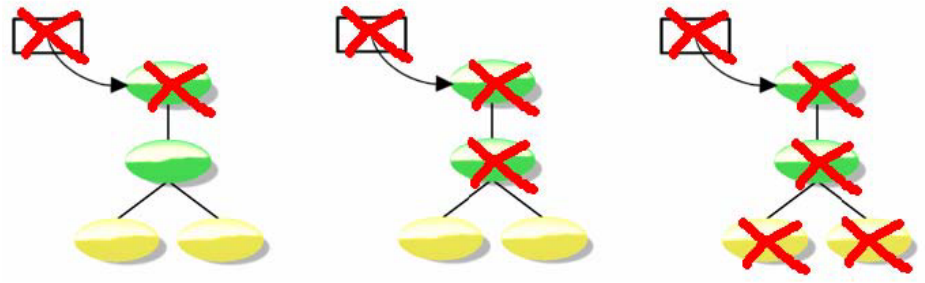


图 2-2 OSG 对象删除过程示意图

OSG 的内存管理系统通过删除指向场景图根节点的指针，删除场景图中的所有实体。

该内存回收机制由两个部分组成：

• OSG 节点和场景数据类都从同一基类 osg::Referenced 派生。该基类包括一个引用计数的实数和一些增加或删除引用计数的方法。

• OSG定义了一个称为osg::ref_ptr<>的**智能指针模板类**[14]，读者可以象使用普通C++指针一样使用该指针。读者可以使用**ref_ptr**<>指针保存OSG节点和场景图数据，保存在堆（heap）供访问。当代码一个Referenced对象地址赋予一个ref_ptr<>类型变量时，Referenced类的引用计数器会自动加 1。

如果程序代码中要保存一个继承于 Referenced 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。如果代码中始终遵循这一原则，那么当最后一个引用对象的 ref_ptr<>指针被弃用时，对象所使用内存将自动释放。

警告

绝对不要使用普通的 C++指针长时间存储从 **Referenced** 派生对象类的指针。有点例外的是，因为堆内存的地址是保存在 ref_ptr<>指针中，可以临时使用普通的 C++指针来保存对象地址。然而，使用 ref_ptr<>是最安全的方法。

ref_ptr<>使用了运算符重载技术，因此，其行为与一般的 C++指针变量极其相似。举例来说，ref_ptr<>在解除对指针地址的引用时，重载了 **operator** ->()运算符和 **operator** *()运算符。

当在应用程序中创建任何继承自 Referenced 的场景图节点或数据时，在程序不可以直接释放其内存。除了极少数例外，几乎所有的 Referenced 派生类都声明了保护类型的析构函数。通过该方式保证了所有继承自 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。以下文字将详

[13] automated garbage collection system ，Array 将其译为“废弃物收集系统”，似乎不太专业。

[14] 模板与一般的 C++应用不同，原非 C++标准的一部分，是通用编程技术兴起后加入 C++标准的。查阅 C++ Primer 第三版或设计模式等教程。

细介绍 Referenced 类和 ref_ptr<>模板，并为读者提供一些示例程序代码。

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功能。所有 OSG 节点和场景图数据，包括状态信息、顶点数组、法线以及纹理坐标等，均派生自 Referenced 类。因此，所有的 OSG 对象均使用了内存引用计数功能。

Referenced 类包括了三个主要组成部分：

• 包括一个保护类型的整型成员变量*_refCount*，该成员用于记录引用计数，在构造时初始化为 0。

• 包括公有类型的成员函数 ref()和 unref()，主要用于控制*_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。

• 析构函数为保护类型，并且为虚函数。因此，该函数在堆栈上的创建、显式的析构都会因为该函数是保护类型而被禁止，而该函数虚函数的特性使得应用程序可以执行子类重载的析构函数。

如同规则，用户代码不需要直接调用 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）实现了一个 Referenced 类型的智能指针，并对其引用计数进行管理。当最后一次引用 Referenced 对象的 ref_ptr<>指针为空时，该对象必定会被释放。ref_ptr<>简化了场景图内存的释放工作，并保证当出现异常的堆栈弹出调用时，对象可以被正确释放。

ref_ptr<>模板类包括以下三个[15]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，用 get()方法可以返回_ptr 的值。

• 模板类通过重载或定义了很多方法，使得可以像正常的 C++指针一样地来使用 ref_ptr<>，如 operator->()和 operator=()。

• 当 ref_ptr<>非空时，使用 valid()方法返回 TRUE。

当程序将一个地址指定给 ref_ptr<>变量时，ref_ptr<>的赋值重载函数 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<>都是通过调用 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 类包含了用于渲染的几何信息，只能是 OSG 的叶节点。对其更多信息请参阅 2.2 节“叶节点（Geode）和几何信息”。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。

下面例子中显示了一个 ref_ptr<>变量的声明、赋值以及检查是否有效的方法。

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

[15] 英文原文此处写为“四个”，但后续文中只写了三个。

```
#include <osg::ref_ptr>
  ...
osg::ref_ptr<osg::Geode> geode = new osg::Geode;
if (!geode.valid())
// ref_ptr<> is invalid. Throw exception or display error.
// ref_ptr<>无效，抛出异常或显示错误。
```

象其他模板一样，包括大括号中变量声明的那些，上面的示例创建了一个 ref_ptr<>变量用于保存 osg::Geode 地址。注意这里赋值方法与一般 C++指针变量的方法相同。

在标准的用法中，用户往往会创建一个节点，并将其作为场景图中其他节点的子节点：

```
#include <osg/Geode>
#include <osg/Group>
#include <osg/ref_ptr>
...
{
   // 创建 osg::Geode 对象；对其赋值使得引用计数增加到一；
   osg::ref_ptr<Geode> geode = new osg::Geode;
   // 假定‘grp’是一个指向 osg::Group 的指针。Group 节点使用 ref_ptr<>
   // 指向其子节点，因此，addChild()使得其引用计数增加到 2；
   grp->addChild( geode.get() );
}
// ‘geode’对象的 ref_ptr<>变量不在括号限定的范围之内,引用计数减至 1
// （本部分代码可能不全）
```

本例中其实并不是非使用 ref_ptr<>不可。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<>仅仅是增加了构造进程中的无用消耗。在此使用简单的 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 所占的内存了。

```
// Create a new osg::Geode object. Don’t increment its reference count.
osg::Geode* geode = new osg::Geode;
// The internal ref_ptr<> in Group increments the child Geode
// reference count to 1.
grp->addChild( geode );
```

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 对象的地址必须赋给一个 ref_ptr<>变量。上述的代码中，这一赋值过程在方法 osg::Group::addChild()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; } //不能这样做，会引起内存泄露！
```

如前所述，OSG 不允许显式地释放派生自 Referenced 的对象，也不能在堆栈中构建。下面的代码将会产生编译或连接错误：

```
osg::Geode* geode1 = new osg::Geode;
delete geode1; // Compile error: destructor is protected. 编译错误：析构函数为保护类型。
```

```
{ osg::Geode geode2; } // Compile error: destructor is protected. 编译错误：析构函数为保护类型
```

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：

```
// OK, because Geode derives from Referenced:
osg::ref_ptr<Geode> geode = new osg::Geode;
int i;
osg::ref_ptr<int> rpi = &i; // NOT okay! 'int' isn't derived
// from Referenced and doesn't support the Referenced interface.
```

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树的连锁释放变得十分简单。当指向根节点唯一的 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' increments the Group count to 1.
    osg::ref_ptr<Group> top = new osg::Group;
    // addChild() increments the Geode count to 1.
    top->addChild( new osg::Geode );
}
// The 'top' ref_ptr goes out of scope, deleting both the Group and Geode memory.
```

当使用函数返回的对象地址时要十分小心。如果进行了错误操作，返回值堆栈中还没有完成对地址保存， ref_ptr<>进行内存地址保存可能会导致越界。

```
// DON'T do this. It stores the address as the return value on
// the call stack, but when the grp ref_ptr<> goes out of
// scope, the reference count goes to zero and the memory is
// deleted. The calling function is left with a dangling pointer.
osg::Group* createGroup()
{
    // Allocate a new Group node.
    osg::ref_ptr<osg::Group> grp = new osg::Group;
    // Return the new Group's address.
    return *grp;
}
```

对于如何返回 Referenced 对象的地址这个比较棘手的问题，可以采用不同的方法来解决。本书示例代码给出的方法是返回一个保存了对象地址的 ref_ptr<>指针，如下所示：

```
osg::ref_ptr<osg::Group> createGroup()
{
    osg::ref_ptr<osg::Group> grp = new osg::Group;
    // Return the new Group's address. This stores the Group
    // address in a ref_ptr<> and places the ref_ptr<> on the
    // call stack as the return value.
     return grp.get();
```

```
}
```

总结：

• 将派生自 Referenced 的对象赋予 ref_ptr<>变量，将会自动调用 Referenced::ref()，并使得引用计数增加 1。

• 如果 ref_ptr<>变量指向其它对象或者被释放，程序将调用 Referenced::unref()方法，使引用计数减一。当计数器值为 0 时，unref()会自动释放对象所占用的内存空间。

• 当 Referenced 对象申请内存时，要尽量将其赋予 ref_ptr<>类型的指针，以保证 OSG 的内存管理系统正确工作。

作为一本快速入门指导教材，以上内容未免显得有些冗长。但是，以上概念是非常重要的，特别是对 OSG 内存管理模式的正确理解，对于每个 OSG 开发者都是不可或缺的。

下一节将介绍一些由 Referenced 派生出来的常见类，涉及的程序段会大量使用 ref_ptr<>类型的指针。在阅读本章时，特别要注意 OSG 内部为每个长期保存的指针使用 ref_ptr<>，就跟前面的例子中调用 osg::Group::addChild()时的情况相似。

2.2 Geodes节点与几何体

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内存引用计数的概念，不妨参看实际的 OSG 例子程序以增强理解。本章将使用前述的内存管理技术，实现一个简单的 OSG 示例程序，并对使用 OSG 几何体相关类构建场景图的方法进行介绍。这些代码开始看可能会比较晦涩，毕竟到现在为止，读者还没有熟悉这其中大部分的类。代码之后将对几何信息类做详细的讲解。

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都使用了引用计数的管理方法。createSceneGraph()函数创建场景图，其返回值也是一个 ref_ptr<>。严格来说，这些代码完全可以使用标准 C++指针来改写，因为函数将返回地址保存在 ref_ptr<>指针中。但是，在程序中使用 ref_ptr<>是一个很好的习惯，因为它可以在异常或中断返回时自动释放内存。本书及其示例代码中均会使用 ref_ptr<>，以鼓励用户养成良好的编程习惯。

清单 2-1 构建一个简单的场景图

下面是一个简单的例子程序，其详细内容可以在本书的附送代码中找到。createSceneGraph()函数指定了一个单四边形图元的几何信息。该四边形的各个顶点颜色不同，但整个几何图元的法线是一致的。

```
#include <osg/Geode> #include <osg/Geometry>
osg::ref_ptr<osg::Node> createSceneGraph()
{
    // Create an object to store geometry in.创建用于保存几何体数据的对象
    osg::ref_ptr<osg::Geometry> geom = new osg::Geometry;
    // Create an array of four vertices.  创建包括四个顶点数据的数组
    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 ) );
    // Create an array of four colors.  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 ) );
    // Create an array for the single normal.  创建同法线的包括四个顶点的数组
    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 ) );
    // Draw a four-vertex quad from the stored data.  根据前面的顶点数据绘制四边形
    geom->addPrimitiveSet( new osg::DrawArrays( osg::PrimitiveSet::QUADS, 0, 4 ) );
    // Add the Geometry (Drawable) to a Geode and return the Geode.
    //  向 Geode 节点添加几何体信息并返回该节点指针
    osg::ref_ptr<osg::Geode> geode = new osg::Geode;
    geode->addDrawable( geom.get() );
    return geode.get();
}
```

代码清单 2-1 创建了只有单一叶节点 Geode 的场景图。图 2-3 显示了只包括了一个节点的场景图结构。该只有一个节点的场景图只对于初学者比较有意义，因为实际的场景图要复杂得多。

注意清单 2-1 中在 y=0 平面上定义了四个顶点。跟 OpenGL 相似，OSG 并没有对程序使用的坐标系进行严格的限制。不过，值得注意的是，osgViewer 类使用的世界坐标系定义为：x 轴正向指向右，y 轴正向指向屏幕里面（前方），z 轴指向上方。这种坐标系定义方式在很多地形与 xy 平面平行的视景仿真系统中得到了广泛的应用。第三章“在程序中使用 OSG”中介绍了如何改变默认的坐标系。清单 2-1 列出的代码使用默认的角度渲染出了一个面向观察者的四边形。

图 2-3 创建只包括单一节点的场景图

除了清单 2-1 中所示的场景图创建过程之外，你还可能希望学习将其场景渲染成图像和动画的方法。不过本章的例子将仅使用了 osgviewer 来观察场景，对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 OSG”。如果想要在 osgviewer 中观察场景图，只需将其写到磁盘文

件即可。清单 2-2 列出了通过调用 2-1 中的函数，并将场景图以.osg 文件格式写入磁盘的代码。当场景图以文件形式保存在磁盘上时，用户就可以使用 osgviewer 来观察它的结构了。

清单 2-2 将场景图写入磁盘

列表显示了示例程序的主入口函数 main()。Main()函数通过调用列表 2-1 中给出的 createSceneGraph()函数创建场景图，并将场景图写入到一个名为 simple.osg 的磁盘文件中。

```
#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::writeNodeFile()." << endl;
}
```

在调用清单 2-1 给出的创建场景图函数之后，清单 2-2 的代码将场景图写入到磁盘文件“Simple.osg”中。文件格式.osg 是 OSG 特有的 ASCII 编码文本文件格式。作为 ASCII 文件，.osg 通常较大且载入速度较为缓慢，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。

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，可以登录网站获取代码，并编译运行 Simple 例程。运行该程序之后，你可以在工作目录下看到生成的文件 Simple.osg。如果你想研究具体的场景图结构，可以使用 osgviewer，命令如下：

```
osgviewer Simple.osg
```

osgviewer 将会显示一幅类似于图 2-4 的图像。对于 osgviewer，相信读者已经不陌生了。因为第一章已经对它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。

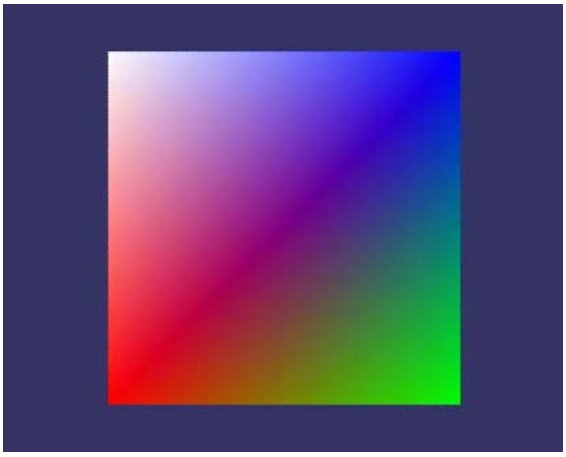


图 2-4 创建的简单场景图示例

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 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 几何体类概述

清单 2-1 中的代码可能看上去会有一些让人迷惑。实际上，它主要执行了三个步骤的工作：

1. 创建顶点、法线和颜色数据的数组。

2. 将 osg::Geometry 对象实例化，并将第一步创建的数组添加到对象中。与此同时，还添加了一个 osg::DrawArrays 对象以指定数据绘制的方式。

3. 将 osg::Geode 节点实例化，并将几何体信息添加为它的子节点。

本节将对这里的每个步骤逐步进行详细的讲解。

- **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、矩阵点乘和叉乘、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维坐标向量，然后将各个 Vec3 压入 Vec3Array。对于法线向量数据，OSG 也采用基本相同的方式，用 Vec3Array 来保存。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 透明度）。最后，本章还提供了使用 Vec2 和 Vec2Array 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，正如清单 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，主要用于保存需要渲染的数据。Drawable 是一个虚拟基类，无法直接实例化。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图形接口，如圆柱体和球；

• osg::Geometry。例子代码中的 Geometry 类是一个主要用于几何体保存与渲染的通用类。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 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 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 类的方法：

• 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。该函数使用 Geometry 类中定义的枚举值做出输入参数。在清单 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 几何体对应唯一一个法线数据。

• addPrimitiveSet():这个方法用于设置 Geometry 类数据如何进行渲染的方法。其参数为一个 osg::PrimitiveSet 指针。PrimitiveSet 是一个虚拟基类，无法直接进行实例化。同一个 Geometry 类可以添加多个 PrimtiveSet 对象。

addPrimitiveSet()方法允许应用程序指定 OSG 对 Geometry 对象中保存的几何体数据的绘制方式。清单 2-1 指定了一个 osg::DrawArrays 对象。DrawArrays 派生自 PrimitiveSet，可以将其理解为 glDrawArrays()顶点数据绘制命令的一个封装。其他的 PrimitiveSet 子类（如 DrawElementsUByte，DrawElementsUShort 以及 DrawElementsUInt）分别列举了 OpenGL 的 glDrawElements()函数入口。OSG 同样也提供了 DrawArrayLengths 类，openGL 中没有函数与其对应。从功能上看，它的执行结果跟使用不同的范围和长度反复执行 glDrawArrays()函数的最终结果类似。

OSG 如何绘制

虽然 primitiveSet 子类提供了与 OpenGL 顶点数组特性几乎相同的功能，但是不能认为 primitiveSet 在内部机制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，甚至是 glBegin()/glEnd()来渲染几何体，继承自 Drawable 类的对象（如 Geomtry）在缺省条件下使用显示列表。同样，应用程序也可以通过调用 osg::Drawable::setUseDisplayList(false)来改变这一特性。

如果用户设置了 BIND_PER_PRIMITIVE 绑定方式，OSG 将使用 glBegin()/glEnd()方式进行渲染，该方式为每个独立的几何图元设定一种绑定属性（如为 GL_TRIANGLES 的每个三角形设定属性）。

DrawArrays()函数最常用的构造函数形式如下：

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

mode 参数为 openGL 中十种基本类型之一，如 GL_POINTS、GL_LINES 或 GL_TRIANGLE_STRIP。在 PrimitiveSet 基类中定义了相同的枚举值，如 osg::PrimitiveSet::POINTS，应用程序中可以根据喜好灵活使用。

first 参数是 OSG 渲染时顶点数据数组中首个元素的索引，而 count 是所有使用元素的总数。举例来说，当顶点数据包括 6 个顶点时，如果你需要根据这些数据渲染一个三角带，可能需要使用 DrawArrays 函数对几何体进行如下操作：

```
geom->addPrimitiveSet( new osg::DrawArrays( osg::PrimitiveSet::TRIANGLE_STRIP, 0, 6 );
```

在添加顶点数据、颜色数据以及 DrawArrays 基本图元到一个几何体对象之后，列表 2-1 给出的代码对几何体进行了最后的操作——在场景图中将几何体数据关联一个节点上。下一节对该操作进行了讲解。

- **Geodes 节点类**

osg::Geode 类是 OSG 的叶节点，主要用于保存几何信息以便渲染。清单 2-1 中建立了一个最简单的场景图，它只有一个独立的叶节点。在清单 2-1 的最后，createSceneGraph()函数通过 ref_ptr<>返回 Geode 地址给 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 没有子节点，但是它可以包含几何体信息（点线面之类）。Geode 这个名字是“geometry node”的合并，意即包含几何信息的节点。应用程序所有渲染的几何信息都必须与 Geode 相关联。Geode 提供了 addDrawable()方法来关联几何体信息。

Geode::addDrawable()将一个 Drawable 指针作为传入参数。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 Group组节点

OSG 的组节点 osg::Group，允许应用程序添加任意数量的子节点，子节点本身也可以有自己的子

节点，如图2-5所示。Group节点是许多其他实用节点类的基类，其中包括本节将要介绍的osg::Transform，osg::LOD和osg::Switch等。

毫无疑问，Group类是由Referenced类派生的。通常情况下，只有Group的父节点引用了当前的Group对象，因此当场景图的根节点被释放时，会引发连锁的内存释放动作，不会产生内存泄露。

可以说，Group类是OSG的核心，它使得应用程序可以有效地组织场景图中的数据。Group类的强大之处在于它提供的子节点管理接口。Group还从其基类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的节点类进行逐一讲解，它们分别是Transform（变换）、LOD（细节层次）和Switch（开关）节点。

2.3.1 子节点接口

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（除Geode节点类之外），因此，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Node的ref_ptr<>变量数组。正因为Group使用了数组，所以应用程序可以根据索引来访问某个子节点。同时，Group类还通过使用ref_ptr<>来保证OSG的内存管理机制生效。

下面的代码段是Group类子接口的部分声明。所有类均位于osg命名空间。

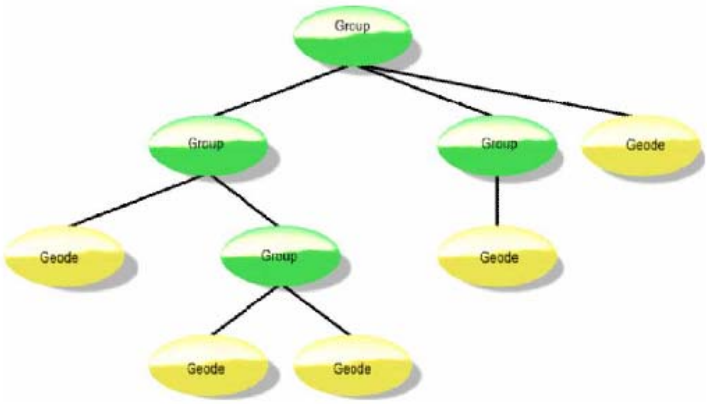


图 2-5 组节点与子节点

绿色标志的Group节点，可以有多个子节点。同样地，其子节点也可以拥有自己的子节点。

```
class Group : public Node
{
    public:
    ...
        // Add a child node.  添加子节点
        bool addChild( Node* child );
        // Remove a child node. If the node isn't a child, do nothing and return false.
        //  移除一个子节点。如果该节点非子节点，直接返回 false
        bool removeChild( Node* child );
        // Replace a child node with a new child node.用新的子节点替换当前子节点
```

```
    bool replaceChild( Node* origChild, Node* newChild );
    // Return the number of children.  返回子节点的数量
    unsigned int getNumChildren() const;
    // Return true if the specified node is a child node.如果所指定的节点为子节点，返回 true；
    bool containsNode( const Node* node ) const;
  ...
};
```

一个简单的场景图可能会包含一个 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。

```
{
  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 均被 group 节点所引用。因此，即便 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 父节点接口

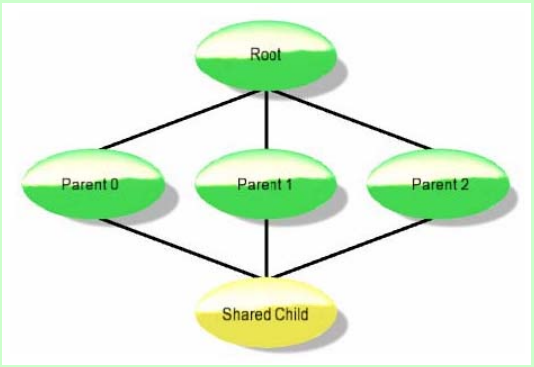
Group 类从 Node 类继承了父节点管理的接口。同样继承自 Node 的 Geode 类当然也可使用这些接口。OSG 允许节点有多个父节点。下面的代码段显示了 Node 类关于父接口的部分声明。所有的类均使用 osg 命名空间。

```
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 类是一个虚基类，应用程序无法直接实例化。Object 类提供了一系列接口用于保存和获取名称、指定要保存数据是静态的还是动态可更改的以及继承自 Referenced 类的内存管理机制。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。

多父节点(Multiple Parents)

当将同一个节点添加为多个节点的子节点时，该节点成为多父节点的子节点，如图所示。应用程序有可能需要对同一部分场景进行多次渲染，并不需要创建多个该场景的备份。2.4.3节的状态设置示例代码显示了如何在不同的变换和不同的渲染状态下渲染同一几何节点(Geode)。



当某个节点存在多个父节点时，OSG渲染时多次遍历该节点，每个父节点保存对该子节点的ref_ptr<>引用，因此，只有当所有父节点不再引用该节点时，它才会被释放。

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1），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的类还提供了许多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。

2.3.3 Transform节点

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类是一个无法直接实例化的虚基类。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。

Transform会影响到OpenGL的模型-视图（model-view）矩阵堆栈。多个连续排列的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 );
```

- MatrixTransform节点

MatrixTransform内部使用osg::Matrix矩阵对象（请看旁边栏）。为创建一个MatrixTransform节点以便进行转换，OSG创建变换矩阵并将矩阵赋给MatrixTransform。

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

有所不同的是，Martrix 并不是从 Referenced 派生的。读者可以通过在堆栈上创建本地的 Matrix 变量。MatrixTransform::setMatrix()方法可以将 Martrix 参数复制到 MartrixTransform 节点的 Martrix 成员变量。

Martrix 提供了一般的变换接口，如平移、旋转以及缩放。读者也可以显式地设计变换矩阵 Martrix，如下所示：

```
osg::ref_ptr<osg::MatrixTransform> mt = new osg::MatrixTransform;
osg::Matrix m;
//设定转换矩阵的 16 个元素
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 );
// OSG 根据转换矩阵的内容，平移 10 米
t->setMatrix( m );
```

读者可以查阅 2.4.3“状态设置示例”一节给出的例子。该例子中，使用多个 MatrixTransform 节点来渲染同一个 Geode 节点，该 Geode 节点在显示时被变换到了不同的位置，而且各个 MatrixTransform 节点的状态设置各不相同。

- PositionAttitudeTransform 节点

PositionAttitudeTransform 节点用途是：通过指定一个 Vec3 类型的位置参数以及一个四元数来进行特定的变换。OSG 提供了 osg::Quat 类用于存储四元数。

矩阵和 osg::Matrix 矩阵

Osg::Matrix 矩阵类存储一个由 16 个浮点数构成的矩阵，并允许对进行操作。该类不是由 Referenced 派生，不能采用引用计数。

矩阵类提供了与 OpenGL 规范或多数 OGL 教程描述类似的接口。显然，该接口由一个二维行优先（row-major）的 C++数组组成。

```
osg::Matrix m;
m( 0, 1 ) = 0.f; // Set the second element (row 0, column 1)
m( 1, 2 ) = 0.f; // Set the seventh element (row 1, column 2)
```

OpenGL 矩阵通常为一维数组表示。在 OpenGL 文档中通常采用列优先数组（column major）的方式显示，如下所示：

0m	4m	8m	12m
1m	5m	9m	13m
2m	6m	10m	14m
3m	7m	11m	15m

```
GLfloat m[16];
m[1] = 0.f; // Set the second element
m[6] = 0.f; // Set the seventh element
```

不管显示如何不同，实际上 OSG 和 OpenGL 矩阵在内存中是完全相同的，OSG 不需要在将其矩阵提交给 OpenGL 之前再进行耗时的转置（transpose）操作。但是，做为一个开发者，应当记得在访

问每个元素之前将 OSG 矩阵在头脑中进行一次转置操作。

Matrix 类为顶点数据的显示矩阵提供了一系列便于理解的接口。

Quat 类也不是从 Referenced 类派生的，同样没有采用引用计数技术。Quat 类提供了很多操作方便的设置接口。下面代码显示了创建和配置四元数矩阵的数个方法：

```
// Create a quaternion rotated theta radians around axis. 创建一个绕轴旋转 theta 弧度的四元数矩阵
float theta( M_PI * .5f );
osg::Vec3 axis( .707f, .707f, 0.f );
osg::Quat q0( theta, axis );
```

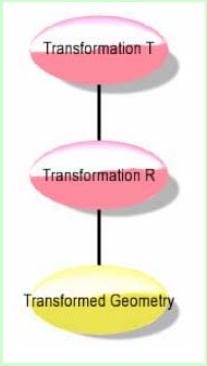
矩阵乘法与连乘（multiplication and matrix concatenation）

为使一个 3 维矩阵 v 绕新原点旋转角度 R，可以使用下列的代码:

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

Matrix 类采用了左乘（premultiplication-style）操作，与常见 OpenGl 文档中经常使用的左乘 v′ = T · R · v 结果恰好相反。

实际上，两者的运算结果是相同的。因为OpenGL运算时右乘的是一个列优先矩阵，而OSG运算时左乘的是一个行优先矩阵，因此两者是等价的[16]。



为对整个 Geode 节点进行变换，首先需要创建一个包含 T 的 MatrixTransform 节点，然后为其添加一个包含 R 的 MatrixTransform 子节点，最后向该旋转变换的 MatrixTransform 节点添加一个 Geode 节点，如图所示。整个过程与下列 OpenGL 代码执行的结果相等。

```
glMatrixMode( GL_MODELVIEW );
glTranslatef( ... );
// Translation T
glRotatef( ... );
// Rotation R ...
glVertex3f( ... );
glVertex3f( ... );
```

[16] This OpenGL notation produces the same result because OpenGL's postmultiplication notation with column-major matrices is equivalent to OSG's premultiplication-style operators with row-major matrices.

总体来看，OSG 描述其接口采用了行优先数组，与 OpenGL 文档给出的列优先数组有所不同，但其内部执行了等价的操作，其矩阵是完全兼容的。

```
// Create a quaternion using yaw/pitch/roll angles.使用姿态角创建一个四元数矩阵
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 );
// 假定 H、P、R 角度已经预先定义
osg::Quat q1( yawRad, yawAxis, pitchRad, pitchAxis, rollRad, rollAxis );
// Concatenate the two quaternions q0 *= q1; Configure a PositionAttitudeTransform by using its
// setPosition() and setAttitude() methods. (x, y, z, theta, and axis are externally defined and declared.)
// 连接两个四元数。通过 setPosition()和 setAttitude()方法设定一个 PositionAttitudeTransform
// 对于 x, y, z, theta, 以及 axis 都认为已经在外部定义过
osg::Vec3 pos( x, y, z );
osg::Quat att( theta, axis );
osg::ref_ptr<osg::PositionAttitudeTransform> pat = new osg::PositionAttitudeTransform;
pat->setPosition( pos );
pat->setAttitude( att );
```

使用时，可以根据应用程序需要，向一个 PositionAttitudeTransform 节点添加任意多个子节点，因为 PositionAttitudeTransform 类继承了 Group 类的子接口特性。与 MatrixTransform 节点的作用机制相似，PositionAttitudeTransform 节点可以根据位置和属性参数，对子节点的几何体进行变换。

2.3.4 LOD节点

使用osg::LOD节点[17]可以实现不同细节层次下物体的渲染。LOD类继承自Group类，因此也继承了Group类所有子接口特性。LOD允许用户指定各个子节点的作用范围。该范围包括了最大和最小值，缺省情况下，其意义表示为距离远近。当某个子节点与当前观察者的距离在这个节点所标示的有效范围之内时，LOD将显示这个节点。LOD的子节点可以按照任何顺序存放，且不必按照距离大小进行排序。

图 2-7 所示为一个 LOD 节点及其三个子节点。第一个子节点是一个包含了子节点的 Group 节点。在第一个子节点与当前视点之间的距离刚好处于这个子节点显示的有效范围时，OSG 将遍历这个节点及其子节点。第二个、第三个子节点的显示控制方法与第一个节点相同。根据节点到视点的距离，OSG 可以选择不显示、显示任意一个或者显示所有的 LOD 子节点。

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;
```

[17] LOD，Level of Detail，译为细节层次，根据不同的要求（一般是到当前视点的距离），以不同的粒度显示物体。

```
// 当视目距离符合要求时 0.f <= distance < 1000.f，显示几何体节点
lod->addChild( geode.get(), 0.f, 1000.f );
```

如果当前多个节点与当前视点的距离都在其设定显示的范围之内，LOD 节点可以同时对多个节点进行显示。

```
osg::ref_ptr<osg::Geode> geode0, geode1;
// Initializa the Geodes. 初始化 Geode 节点
...
osg::ref_ptr<osg::LOD> lod = new osg::LOD;
// Display geode0 when 0.f <= distance < 1050.f
lod->addChild( geode0.get(), 0.f, 1050.f );
// Display geode1 when 950.f <= distance < 2000.f
lod->addChild( geode1.get(), 950.f, 2000.f );
// Result: display geode0 and geode1 when 950.f <= distance < 1050.f
```

缺省情况下，LOD 节点计算当前视点到物体包围体中心点的距离。如果这样并不符合用户的渲染要求，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;
// Use a user-defined center for distance computation
lod->setCenterMode( osg::LOD::USER_DEFINED_CENTER );
// Specify the user-defined center x=10 y=100
Lod->setCenter( osg::Vec3( 10.f, 100.f, 0.f ) );
```

如果应用程序已经将缺省的计算中心点进行了修改，现在需要恢复计算中心点为物体的包围球体中心，那么此时可以调用 osg::LOD::setCenterMode(osg::LOD::USE_BOUNDING_SPHERE_CENTER)。

缺省条件下，LOD 使用最大距离值和最小距离值来表示范围，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。如果要设置 LOD 节点的有效范围模式，应用程序可以调用 osg::LOD::setRangeMode()，并将显示参数设置为 PIXEL_SIZE_ON_SCREEN，或者 DISTANCE_FROM_EYE_POINT。

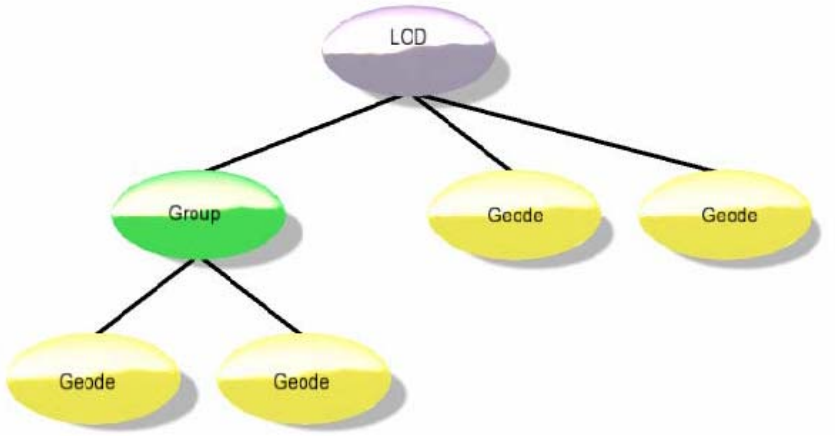


图 2-6 LOD 节点

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。

2.3.5 Switch节点

使用 osg::Switch 节点可以指定需要渲染或者跳过的子节点。Switch 类的典型用法有：根据当前渲染负载，执行自定义的渲染均衡策略，有选择地对需要渲染的场景图的一部分进行渲染；或者在游戏的界面和层级之间有选择地切换。

与 LOD 节点类似，Switch 节点的用户子接口也是从 Group 节点类派生。每个 switch 节点都关联了一组布尔数值。运行时，switch 节点根据其子节点对应的布尔值确定该节点是否参与渲染，布尔值为 true 则渲染，否则忽略。下面的代码生成一个具有两个子节点的 switch 节点。两个子节点中，一个是可见的，另一个不可见。

```
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 );
```

如果应用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() );
```

在程序中，可以使用 Switch::setNewChildDefaultValue()来改变所有新加入节点的缺省值。该设置会保持到重新设置为止。

```
// 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() );
```

当程序将一个子节点添加为 Switch 节点的子节点后，可以在程序中改变它的参数。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或禁止各种条件下 Switch 开关的子节点显示，你需要使用节点更新回调函数（osg::NodeCallback）或者节点访问函数（osg::NodeVisitor）来控制场景图的节点显示和对各帧进行渲染。本书将在第三章“在用户程序中使用 OSG”里讨论更新回调和 NodeVisitor 类的话题。

```
osg::ref_ptr<osg::Group> group0, group1;
...
// Create a Switch parent node and add two Group children:
osg::ref_ptr<osg::Switch> switch = new osg::Switch;
// Render the first child:
switch->addChild( group0.get(), true );
// Don't render the second child:
switch->addChild( group1.get(), false );
```

如果用户代码中没有对 switch 子节点进行属性设置，switch 节点会自动赋予子节点属性为 true。

```
// Add a child. By default, it's visible
switch->addChild( group0.get() );
```

应用程序可以通过调用 Switch::setNewChildDefaultValue()方法来更改新添加子节点的默认属性。

```
// Change the default for new children:
switch->setNewChildDefaultValue( false );
// These new children will be turned off by default:
switch->addChild( group0.get() );
switch->addChild( group1.get() );
```

在向 Switch 添加完子节点后，可以使用 Switch::setChildValue()方法为子节点设置新的属性值。

```
// Add a child, initially turned on:
switch->addChild( group0.get(), true );
// Disable group0:
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的各种可能情况下，完全实现对 Switch 子节点的显示控制，应用程序需要使用节点更新遍历回调（osg::NodeCallback）或者 NodeVisitor 节点访问器（osg::NodeVisitor），逐帧地控制场景图节点的数值。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

在第 2.2 节“叶节点（Geode）和几何信息”的示例代码中，我们构建了一个场景图应用程序，它生成的数据可以通过 osgviewer 程序来进行检查。当你在 osgviewer 中旋转矩形图形时，可以发现这个矩形已经被场景中的一个光源照亮了。osgviewer 通过配置 OSG 渲染状态来实现场景的光照效果。

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）渲染状态，例如Alpha检验、Blending融合、剪切平面、颜色蒙板、面剔除（faceculling）、深度和模板检验、雾效、点和线的光栅化（rasterization）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[18]（fragment shader）。

用户应用程序通过在 osg::StateSet 中设置渲染状态。用户可以将 StateSet 关联到场景图中的任意一个节点（Node），或者关联到 Drawable 类。正如很多 OpenGL 开发者所知，OpenGL 程序的开发需要尽

[18] Shader，阴影？着色？

量使状态量的变化实现最小化，避免冗余的状态设置；StateSet 对象可以自动实现这些过程的优化。

在 OSG 遍历整个场景图时，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场景图子树进行不同的状态设置。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。

应用程序应当尽量使关联到场景图的 StateSet 对象保持最少。StateSet 对象越少，占用的内存也就越少，OSG 一次场景图遍历中所需要完成的工作量就越少。为更好地实现 StateSet 对象数据的共享，StateSet 类派生自 Referenced 类。也就是说，通过共享同一个 StateSet 对象，Node 或 Drawable 类不需要额外的代码来对其内存空间进行清理。

OSG 将渲染状态分成两部分：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 的渲染模式和 OpenGL 的状态特性几乎是一一对应的；在 OpenGL 中，这些特性通过函数 glEnable()和 glDisable()进行控制。用户程序可以通过设置渲染模式，以允许或禁止某个功能，例如纹理映射、灯光等。简单来说，渲染模式是指渲染特性，而渲染属性是控制此功能的变量和参数。

如果要设置渲染状态的值，用户程序需要完成以下操作：

——为需要设置状态的 Node 或 Drawable 对象提供一个 StateSet 对象实例。

——为 StateSet 实例，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，使用下面的代码：

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

在上述代码段中，obj 是一个 Node 或 Drawable 类对象；getOrCreateStateSet()是两个类均定义的方法。该方法返回一个指向 obj 对象相关联的 StateSet 的指针。如果 obj 还没有设置过关联的 StateSet，那么这个方法可以返回一个新指针并将其关联到一个 obj 对象。

StateSet 从 Referenced 类派生。Node 或 Drawable 类内部使用 ref_ptr<>来引用 StateSet，因此，如非长时间引用 StateSet 的情况下，也可以使用标准 C++指针变量来保存 obj 的状态。如果 state 变量是某个函数内的局部变量，而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 变量是一个指向 obj 对象的 StateSet 指针。当应用程序获得了一个指向 StateSet 的指针时，就可以设置其属性和模式。后续章节提供了相应的例子，并对此进行详细的介绍。

2.4.1 属性与模式

OSG 为每个状态属性定义了不同的类，使用非常方便。所有属性类均继承自 osg::StateAttribute，StateAttribute 类是一个无法直接实例化的虚基类。

由于从 StateAttribute 派生的类有数十个，本书将对其中的一些属性类作简要阐述，主要对灯光和纹理映射的属性进行介绍。限于篇幅，本书不可能对所有属性类的进行详尽的讲解。如果读者希望进行更深入的了解，可以阅读 OSG 开发环境下 include/osg 目录的头文件代码，可以获得关于派生自 StateAttribute 的类更多信息。

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设置一个纹理单元（texture unit），以实现多重纹理的效果。

- **设定属性**

如果要设置一项属性，需要首先将要修改的属性类实例化。通过设置该类的数值，然后用osg::StateSet::setAttribute()将其关联到 StateSet。下面的代码段显示了如何实现面剔除（face culling）属性的过程：

```
// 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 );
```

上面的代码段中，geom 是一个 Geometry 几何体类对象（当然也可以是任何其它派生自 Drawable 和 Node 的对象）。在获取 geom 的一个 StateSet 指针后，代码创建了一个新的 osg::CullFace 对象，并将其设置为 state 变量的属性。

CullFace 是一个派生自 StateAttribute 类的属性类。它的构造函数只有一个参数，即指定需要剔除的平面是几何体正面还是反面，该参数是一个枚举变量，可供选择的值有：FRONT，BACK，FRONT_AND_BACK。这些枚举量在 CullFace 的头文件中定义，它们与 OpenGL 的枚举量 GL_FRONT，GL_BACK 和 GL_FRONT_AND_BACK 等价。

如果你对 OpenGL 足够熟悉，那么你可以把上面代码理解成对函数 glCullFace(GL_BACK)的调用。不过，请记住 OSG 是一个场景图系统。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。在绘制遍历（drawtraversal）中，OSG 将跟踪状态数据的变化，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。当应用程序将派生自 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。在上述的典型应用中，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 通过 ref_ptr<>来负责内存的管理。

- **设定渲染模式（Mode）**

用户通过使用 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：

```
// 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()中可以使用的 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 提供了一个方便的、可以同时进行属性和模式设置的单一函数接口。在多数情况下，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，可以使用 osg::StateSet::setAttributeAndModes()方法。下面的代码段将关联 Blend 融合属性，同时打开融合模式。

```
// 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渲染属性对应的渲染模式。其缺省值为 ON。这样用户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

在设置节点的渲染状态时，该状态被赋予当前节点及其子节点。然而，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参数。换句话说，缺省情况下子节点会继承父节点的状态，直到子节点改变其状态参数为止。图 2-7 表现了这个概念的实现过程。

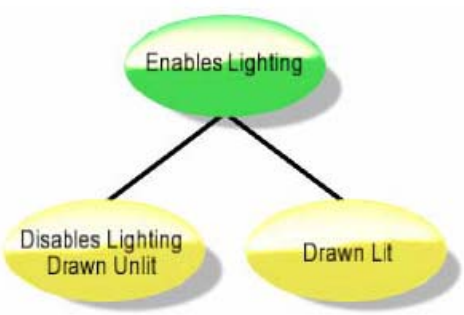


图 2-7 状态继承

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。

这种继承特性在多数情况下都非常实用。但是有时候渲染可能需要不同的行为特性。假设场景图中有一个包含了实体多边形几何体的节点。如果需要以线框模式来渲染场景图，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。

在场景图中，OSG 允许用户根据任何一个节点的渲染属性和模式需求，单独改变原有的状态继承特性。用户以下这几种列举形式都是有效的。

• osg::StateAttribute::OVERRIDE 如果将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，而不管是否对其状态进行了更新。

• osg::StateAttribute::PROTECTE 这种形式为 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操作，然后作为 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。

```
//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=newosg::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，你可能创建了一个包括光源的场景，光源几何体使用亮度（luminance）光源。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。此时，如果设置光源的 GL_LIGHTING 渲染状态为 PROTECTED，就可以保证该光源有效。

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子节点的方法。在“矩阵变换节点（MatrixTransform）”一节中则介绍了使用 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。下面示例代码将会融合以上所有的概念进行介绍。

本节将会提供一个修改 OSG 渲染状态的简单例子。本例创建了一个 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不同的 MatrixTransform 父节点。图 2-8 显示了这一场景图，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。

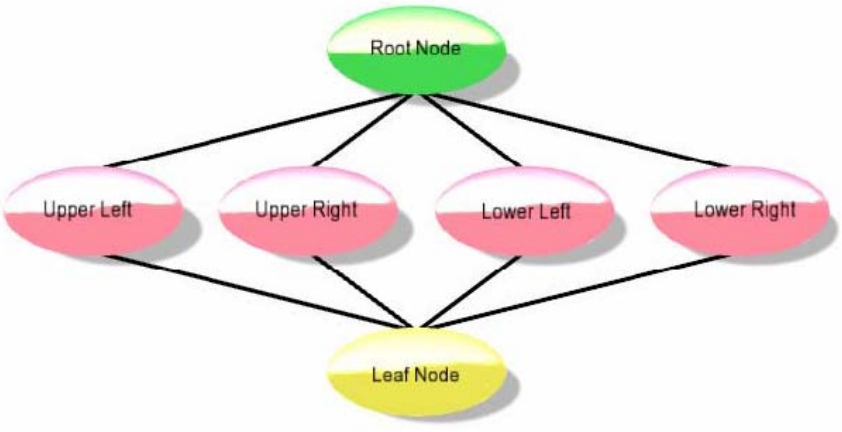


图 2-8 渲染状态示例程序的场景图

清单 2-3 状态修改

下面的程序段将多个 Drawable 对象添加到同一个 Geode 节点上去。代码对每个 Drawable 对象设置了不同的状态，并通过设置 Geode 对象的 StateSet 属性,禁止了所有实体的光照属性。

```
#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()
{
    // Create the root node Group.
    osg::ref_ptr<osg::Group> root = new osg::Group;
    {
      // Disable lighting in the root node's StateSet. Make
      // it PROTECTED to prevent osgviewer from enabling it.
      osg::StateSet* state = root->getOrCreateStateSet();
      state->setMode( GL_LIGHTING, osg::StateAttribute::OFF | osg::StateAttribute::PROTECTED );
    }
    // Create the leaf node Geode and attach the Drawable.
    osg::ref_ptr<osg::Geode> geode = new osg::Geode;
    geode->addDrawable( createDrawable().get() );
    osg::Matrix m;
    {
      // Upper-left: Render the drawable with default state.
      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() );
    }
    {
      // Upper-right Set shade model to FLAT.
      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 );
    }
    {
      // Lower-left: Enable back face culling.
      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();
      // Default: BACK
      state->setAttributeAndModes( cf );
    }
```

```
    {
      // Lower-right: Set polygon mode to LINE in
      // draw3's StateSet.
      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 );
      // Also set the line width to 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 实例关联到一个新 Geode 类型的实例 geode 上。

这个例子将同一个 Geode 渲染了四次。它创建了四个 MatrixTransform 节点，每个节点位于不同的空间位置，然后将 Geode 添加到各个 MatrixTransform 节点上做为其子节点。为对 Geode 节点的外观，每个 MatrixTransform 节点都有自己的设置 StateSet。

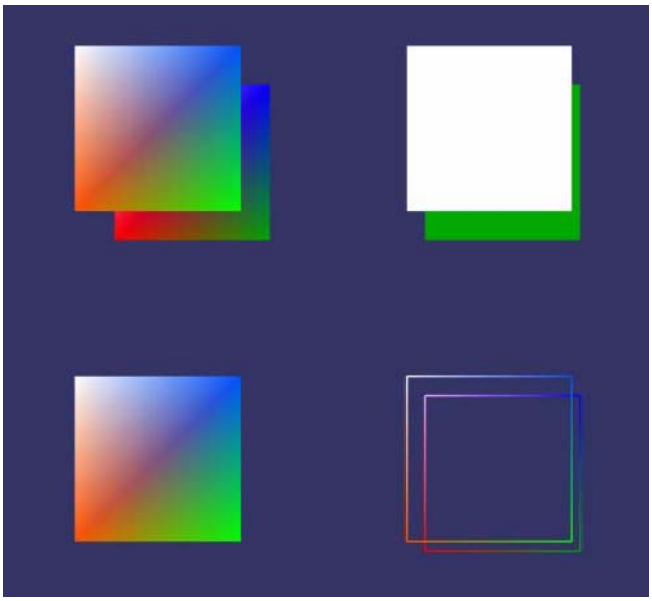


图 2-9 osgviewer 中显示的渲染状态示例场景图

清单 2-3 中生成的场景图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。

• 第一个 MatrixTransform 实例使用了原始的 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 ShadeModel 类型属性，赋值为 FLAT，并将 geode 移动到右上角。这样系统将使用平面着色(flat shading)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。

• 第三个 MatrixTransform 实例为 StateSet 设置了一个 CullFace 属性，并将 geode 移动到左下角。缺省情况下，通过调用构造函数，CullFace 会剔除多边形的背向面。通过使用 setAttributeAndModes(cf) 关联 CullFace 并且打开 GL_CULL_FACE 模式。（createDrawable()返回的两个四边形有着相反的顶点排列顺序，因此无论视点的位置如何，都会有一个背向面存在。）

• 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。此处的代码将所有正向面和背向面（FRONT_AND_BACK）都改为线框模式（LINE），并设置线宽为 3.0。

与前面的示例相同，本处的例子将场景图写入了文件 State.osg。运行这个例子之后，可以使用下面的命令在 osgviewer 中显示本例输出的结果：

```
Osgviewer State.osg
```

图 2-9 所示为 osgviewer 中的场景内容。

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：

• 指定几何体的纹理坐标。

• 创建纹理属性对象，并在其中保存纹理图形的数据。

• 在 StateSet 设置中，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 的早期版本不支持多重纹理，在加入对多重纹理支持的属性后，OpenGL 保留了对非多重纹理接口的支持，以实现对下兼容。从底层来看，此时 OpenGL 将非多重纹理接口解释为使用纹理单元 0 对应所有的纹理数据。

与 OpenGL 不同的是，OSG 不支持对非多重纹理接口。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。如果确实需要使用单一纹理，只需将所有纹理都指定到纹理单元 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 osg::Geometry::setTexCoordBinding()函数来绑定纹理数据。纹理坐标总是与各个顶点绑定的。

- **图像加载**

应用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 是 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() );
```

osgDB 库的 readImageFile()方法创建一个新的 Image 对象，然后加载图像文件并将其保存在 Image 对象之中，最后返回此对象。2.5 节文件输入输出将会对 readImageFile()函数进行更详细的介绍。

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。

一般来说，对于大量使用纹理贴图的程序，往往需要实现更严格高效的内存管理。Image 对象继承自 Referenced 类，而在 Texture2D 内部保存了一个指向 Image 的 ref_ptr<>指针。在第一次渲染时，OSG 创建了一个 OpenGL 纹理对象，用于保存图像数据。该操作的结果是产生了两个纹理图像的副本：一个由 Image 对象保存，而另一个则由 OpenGL 拥有。在简单的单通道（single-context）场景渲染模式中，应用程序可以通过设置 Texture2D 以解除对 Image 的引用来降低内存损耗。如果 Texture2D 对象是 Image 对象的唯一引用者，那么 OSG 将释放 Image 及其内存空间。下面的代码演示了如何设置 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 不会释放对 Image 的引用。在多通道（multi-context）的场景渲染模式中，如果纹理对象并没有在各通道中共享，那么应用程序会希望能够实现上述行为。

- **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接口十分类似。用户可以使用函数 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，第二个参数是从 StateAttribute 类派生的一种纹理属性。有效的纹理属性类总共有两大类六种，其中第一类包括了五种纹理类型（osg::Texture1D，osg::Texture2D，osg::Texture3D，osg::TextureCubeMap 和 osg::TextureRectangle），另一类属性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 和渲染状态对象 StateSet，将 tex 关联到渲染状态 State，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
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，setTextureAttributesAndModes()将设置坐标生成模式为 GL_TEXTURE_GEN_Q，GL_TEXTURE_GEN_R，GL_TEXTURE_GEN_S 以及 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。

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，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，可使用值包括 OVERRIDE，PROTECTED 和 INHERIT。程序还可以通过修改 setTextureMode()和 setTextureAttribute()的第三个参数来指定这个继承标志。

```
// 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 );
```

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，不是将其渲染成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，创建阴影时使用 osgShadow 类进行。

如果要在用户程序中使用光照，需要遵循下面的步骤：

- ✧ 指定几何体的法向量。
- ✧ 允许光照并设置光照状态。
- ✧ 指定光源属性并关联到场景图。
- ✧ 指定表面材质属性。

本节将针对上面的每一步进行讲解。

如果用户程序使用了第三章中描述的 osgViewer 类，注意 osgViewer 类中启用了光照，并且只用了单独一个光源。在用户应用程序中可以通过设置 GL_LIGHTING 模式或改变 GL_LIGHT0 参数对此进行重载。更多细节，请参阅光照状态一节。

➢ 法线

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

在多数 3D API 中，法线数据必须单位化才能得到正确的结果。注意缩放变换会改变法线的长度。如果应用程序中的 Geometry 对象中已经包含了单位化的法线数组，但光照计算的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 中启用对法线的缩放支持。

```
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缩放变换时仍然保持单位长度。如果场景中的缩放变换是非均匀的，那么你可以允许法线标准化模式，以保证法线为单位长度。

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

法线缩放变换后，对法线的标准化操作会耗费大量的时间，如非必要尽量避免。

- **光照状态**

为在 OSG 中获得光照效果，你需要启用光照并至少需要生成一个光源。程序 osgviewer 在缺省情况下就是这样做的，它在根节点的 StateSet 中设置了相应的模式。用户可以在自己程序中进行同样的设置操作。下面的代码段用于启用了光照效果，并为根节点的 StateSet 启用了两个光源（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（包括表面材质颜色的设置）。

OSG 还提供了从 StateAttribute 派生的 osg::LightModel 属性，用以控制全局的环境光（ambient）的颜色、局部视图、双面光照以及镜面反射的颜色（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 对象并定义光源的参数。将光源 Light 添加到一个 osg::LightSource 节点中，然后把 LightSource 节点添加到场景图结构中。LightSource 是一个包含了单个 Light 定义的叶节点，简洁而高效。由 Light 定义的光源将对整个场景产生影响。

SG 最多同时支持八个光源，编号从 GL_LIGHT0 到 GL_LIGHT7，而且这也与用户使用的 OpenGL 版本有关。使用多个光源会对顶点数有所限制的 OSG 应用程序影响很大，导致渲染效率降低。

警告

许多刚入门的 OSG 开发人员可能会错误地认为 LightSource 的子节点会自动设置为启用光照（lit）。事实并非如此！OSG 会根据场景图节点的当前渲染状态实现光照，而不是根据对 LightSource 节点的继承关系。OSG 应用程序必须启用 GL_LIGHTING 以及生成至少一个光源（如 GL_LIGTH0），才能实现光照效果。

实际应用中，可以把 LightSource 节点看做是包含了单一 Light 光源对象的叶节点。但是，用户也可以向 LightSource 节点添加子节点，因为 LightSource 继承自 Group 类。通常，为渲染光源的物理照明效果，应用程序可以将几何体关联到 LightSource 子节点。

OSG 允许应用程序使用比 OpenGL 实现更多的光源，但是该内容超出本书所要研究的范围，有兴趣的读者请查阅相关资料。

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，可以使用设置光源的序列号。例如要把一个 Light 对象与 GL_LIGHT2 相关联，则可以光源号为 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 );
```

注意，在缺省情况下光源序列号为 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(ambient)、散射光颜色（diffuse）、镜面反射光颜色（specular）。用户可以创建点光源、平行光源或者锥形光源（spot），也可以指定衰减参数，使得光强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 );
```

定位状态操作(Positional State)[19]

当 OpenGL 程序采用 glLight()函数设置光源位置时，OpenGL 会根据当前的模型-视图(model-View)矩阵变换光源位置。在 OSG 中，这一概念称为定位状态操作(Positional State)。在剔除遍历中，OSG 向一个定位状态容器中添加各种位置状态量，以保证在绘制遍历中所有的变换都是正确的。

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 关联到场景图中。灯光的位置可以由 LightSource 节点在场景图中的位置决定。OSG 根据当前 LightSource 节点的变换状态来改变灯光的位置。OSG 开发者通常将 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 );
```

[19] 此名称 Array 译本译为“位置状态”，似乎不能涵盖该词的所有含义。此处暂译为定位状态操作。

```
// 添加 Light 对象到 LightSource
// 然后添加 LightSource 和 MatrixTransform 到场景图
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。用户也可以通过设置 LightSource 的参考系（Reference Frame）来禁止这一特性，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，而是把灯光位置作为一个绝对值来看。

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

- **材质属性**

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

Material 类允许用户设置环境光、散射光、镜面反射和放射材质的颜色，另外还有镜面反射指数（specular exponent, or shinness 或高光）参数。Material 类定义了枚举量 FRONT、BACK 和 FRONT_AND_BACK，用户程序为几何体的正面和背面设置材质属性。例如，下面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反射指数。

```
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，此处镜面指数值必须在 1.0 到 128.0 至间，除非用户自行编写了新的实现，增大了范围。

采用 OpenGL 直接设置材质属性可能会比较浪费资源。而 OSG 提供的颜色跟踪材质（color material）特性为用户程序提供了一种高效的方法。应用程序只需要改变当前主要的颜色，就可以改变某一特定的材质属性。在多数情形下，这一操作比直接修改材质属性的效率更高，增强有光照场景和无光照场景的一致性，并满足应用程序对材质的需要。

要启用**颜色跟踪特效**，需要调用 Material::setColorMode()方法。Material 类定义了以下的枚举值：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特效，此时基本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特效，此时几何体前向面的环境材质和散射材质颜色将自动跟踪当前基本颜色的改变而改变。

```
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，源代码不再列出。源代码将生成一个场景文件 Lighting.osg，将场景图写入文件中。用户可以使用下面的命令显示场景图：

```
osgviewer Lighting.osg
```

图 2-10 所示为 osgviewer 中显示的场景图。

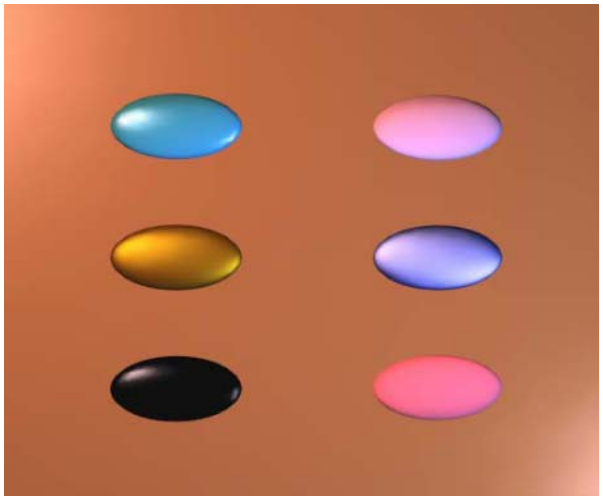


图 2-10 osgviewer 中显示的光照示例场景图

这个例子渲染了六个椭球体（lozenges，菱形，实际上为椭球体）对象和一个背景面。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 文件输入输出

前一节所介绍了创建几何体和渲染状态场景图的编程技术，其中多数示例程序通过手工编程，创建一些几何体对象。但是通常来说，应用程序是从磁盘文件中读取并显示大规模的、比较复杂的几何模型。这就需要提供与磁盘文件读写功能以加载模型，并返回预设的场景图信息。

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 文件格式插件系统，以实现对不同文件格式的支持。第 1.6.3 节“组件”已经对插件概念做了简介，而图 1-9 显示的是插件与整个 OSG 架构的嵌入方式。

本章所述例子使用 osgDB 来实现文件的输入输出。所有例子均使用 osg 读写插件将场景图写入磁

盘上一个.osg 文件之中。光照示例中，程序使用了 osg 格式插件从名为 lozenge.osg 的文件中读取子场景图信息，而纹理映射示例中使用 png 插件读取纹理图像。但是，前面章节中并没有对文件读写进行具体的介绍。为使用户能够合理高效地使用文件读写功能，本节将更详细地解释此类插件的使用。介绍的内容主要有：读写文件的接口、文件检索方式以及 OSG 通过插件读取文件的过程。

2.5.1 接口

osgDB 提供了文件 I/O 的函数接口，插件系统对应用程序完全透明的。用户只需要两个 osgDB 的头文件来定义这个接口：

```
#include <osgDB/ReadFile>
#include <osgDB/WriteFile>
```

应用程序如果要使用 osgDB 的文件 I/O 功能，直接可以在源代码中包含这两个头文件。在 osgDB 命名空间内，定义了一系列用于实现文件 I/O 的函数。

- **文件读取**

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 将根据文件扩展名来识别文件类型，并使用相应插件将文件转换为场景图。readNodeFile()返回一个指向场景图根节点的指针。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 filename 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 filename 包含了相对路径（或者不包含路径），OSG 将使用 osgDB 的数据文件路径列表（data file pathlist）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列表，可以使用 osgDB::Registry::getDataFilePathList()函数。osgDB::Registry 是一个单独类（singleton），因此要调用这个函数的话，需要访问该类的实例。函数会返回一个对 osgDB::FilePathList 的指针引用，也就是一个简单的 std::deque<std::string>。假如要在一个字符串 newpath 中保存目录列表的话，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。如果想要查看文件读取失败的原因，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，再次运行代码，观察输出信息，应该能够找到操作失败的原因。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对应的位置。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 在执行写入操作时，不对文件的存在性进行检查，因此，如果发现同名文件，将会覆盖该文件而不发出任何警告或提示。要避免这一状况出现，应用程序只能自行检查该文件是否已经存在，并采取相应的措施。

2.5.2 插件的搜索和注册

OSG 插件是一组动态链接库，其中实现了 osgDB 头文件 ReaderWriter 中定义的接口。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。

警告

文件操作插件不一定同时支持读取和写入操作。例如，OSG 目前版本仅支持 3D Studio Max 输出的 3ds 文件的读取操作，而不支持 3ds 文件的写入操作。事实上，多数 OSG 插件仅支持文件的读入，而不支持输出。

要获得最新的文件格式支持信息，请登陆 OSG 维基百科网站。

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扩展名的文件格式。例如用于读取 RGB 格式的二维图形插件，事实上可以读取很多格式，包括有.sgi、.int、.inta、.bw、.rgba、.rgb。osgDB::Registry 的构造函数包括了特殊的代码来实现这类插件。Registry 类维护了一个扩展名映射表，来实现对不同扩展名文件格式的支持。

OSG 并不是查找并加载所有的插件，实现对所有格式文件的支持。这在程序启动时将会是一笔很大的开销。因此，OSG 使用相关链（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。OSG 安装后，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。但是，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 节点与osgText类

尽管 OSG 提供了丰富的设计特性，但是在开发时，通常都需要开发人员自行从 OSG 的核心节点类中派生出特定的节点类。这些派生功能通常并不属于 OSG 核心库，更适合做为辅助模块库存在。

所谓 NodeKit 是指通过扩展特定节点类（Node）、渲染属性类（StateAttribute）、绘图类（Drawable）对象功能，以及通过 osg 封装类提供对.osg 格式支持的高级扩展库。

第 1.6.3 节“组件”介绍了 NodeKit 的概念，并从高层次上对 OSG 当前版本拥有 NodeKit 库做了介绍。本节将通过示例介绍 osgNodeKits 库的一个常用组件——osgText，该库主要用来在场景中显示带有纹理贴图的三维文字。

2.6.1 osgText组件库

osgText 库定义了一个名称 osgText 命名空间，并在这个命名空间中提供了一些非常实用的相关字体加载类和文字渲染类。

osgText 库的核心部分是 **osgText::Text** 类。Text 类继承自 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。一般来说，应用程序应当为每个需要显示的字符串创建一个 Text 对象。

osgText 库的另一个核心部分是 **osgText::Font** 类。osgText 函数可以根据字体文件名称来创建 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 类将创建一个包括需要渲染字体轮廓的纹理贴图。渲染时，Text 对象为字符串中的每个字符绘制一个附加纹理的四边形，通过匹配纹理坐标显示正确的文字形状。

osgText 库还定义了一个 String 类，以支持多字节字符（multibyte）和多国文字编码。

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个字符串，只需要创建一个 Font 对象，然后在 Text 对象间共享即可。

2.为整段显示的字符建立一个 Text 对象，并指定对齐方式、文字方向、位置和大小参数。将步骤 1 中创建的 Font 对象关联到该 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，或者根据需要创建多个 Geode 节点。将 Geode 节点作为场景图的子节点加入。

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.” );
```

尽管看上去 Text::setText()是使用 std::string 作为输入参数，实际上，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，setText()函数使用的字符串常量参数，在运行时实时地被转换成 osgText::String。

如果通过 readFontFile()加载字体失败，且程序在当前系统上无法找到任何可用字体，此时不能调用 Text::setFont()。这样 Text 会使用缺省可用的字体。

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。

- **位置坐标**

Text 与 Geometry 相似，都可以在剔除（ cull）和绘制（ draw）遍历中变换自己的坐标位置。缺省情况下，位置是在对象自身坐标原点（所在的位置）。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在视野中的位置。Text 通过将位置变换、正面方向和对齐方式等参数结合到一起，确定文字渲染的实际位置。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三维空间中的朝向。用户可以使用 Text::setAxisAlignment()方法来设置文字位置，其输入参数为 Text::AxisAlignment 类型的枚举值。如果要创建一个广告牌类型的文字，这种类型可以始终朝向当前视点位置，创建时可以使用 Text::SCREEN。

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

或者，你也可以让文字沿着坐标轴所在的平面放置。缺省的文字方向为 Text::XY_PLANE，即文字朝向 Z 轴正向，水平放置在 XY 平面上。

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

表 2-1 中列出了 Text::AxisAlignment 的枚举类型以及其对文字朝向的影响。

Text::AxisAlignment 的枚举类型总共有七种：

Text::XY_PLANE（缺省）为默认值。

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。

表 2-1 Text::AxisAlignment 枚举类型及效果

Table 2-1 Text Orientation AxisAlignment Enumerants	
`Text::XY_PLANE`	(Default.) Places text in the *xy* plane facing positive *z*.
`Text::XZ_PLANE`	Places text in the *xz* plane facing positive *y*.
`Text::YZ_PLANE`	Places text in the *yz* plane facing positive *x*.
`Text::REVERSED_XY_PLANE`	Places text in the *xy* plane facing negative *z*.
`Text::REVERSED_XZ_PLANE`	Places text in the *xz* plane facing negative *y*.
`Text::REVERSED_YZ_PLANE`	Places text in the *yz* plane facing negative *x*.
`Text::SCREEN`	Renders text that always faces the screen.

- **对齐方式**

此处对齐方式的概念与字处理软件中文字对齐方式或者电子表格中单元格对齐的概念相似。它决定了渲染文字相对于其坐标位置（使用 setPosition()函数）的水平和垂直对齐方式，Text 定义了一系列的枚举类型，称为 Text::AlignmentType。枚举值命名时，首字用于确定文字的水平方向，第二个字用来确定垂直方向。缺省选项是 Text::LEFT_BASE_LINE，即水平上是沿着文字的左边界对齐，垂直方向上沿着文字的底边对齐。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。

为改变文字对齐特性，可以使用 Text::setAlignment()函数。下面代码显示了水平居中顶端对齐的设置方法。

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

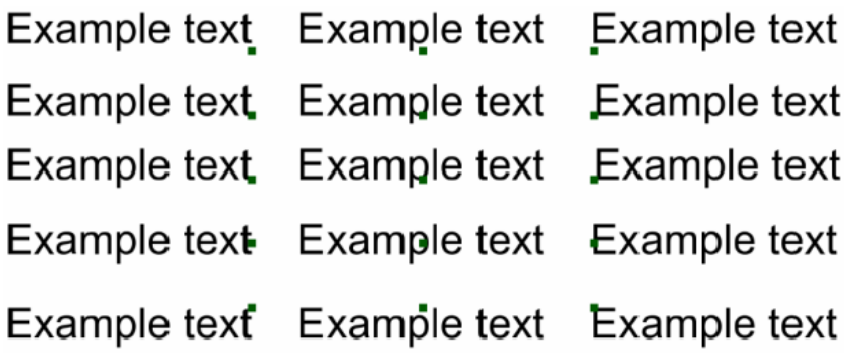


图 2-11 文字对齐方式

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	LEFT_TOP

- **文字大小**

文字尺寸缺省的字符高度为 32 个坐标单元。而字符宽度可变与字体有关。Text 根据 Font 对象中保存的信息，按照比率渲染文字。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坐标大小。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 );
```

将字符高度的显示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 是在剔除（ 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，因此会产生一个帧的延迟。在高帧率显示的程序中，该延迟可以忽略不计。

- **分辨率**

应用程序为避免文字出现模糊，经常需要改变字体纹理贴图的图形分辨率。默认情况下，osgText 为每幅贴图分配 32×32 个像素元。如果需要改变这个数值，可以使用 Text:: setFontResolution()方法。下面代码提高了字体分辨率，osgText 为每幅贴图分配 128×128 个像素元。

```
text->setFontResolution( 128, 128 );
```

如果多个分辨率不同 Text 文字对象共享一个 Font 对象，且不同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不同分辨率的多个纹理拷贝，以确保显示效果。

值得注意的是，字体分辨率的提高会使得系统对硬件资源的需求提高，如对显卡纹理内存需求的提高。因此，在显示结果可以接受的前提下，因此应用程序应当选用最小的字体分辨率。

- **颜色**

缺省情况下，Text 对象使用白色绘制文字。用户可以使用 Text::setColor()方法来改变这一缺省值。设置颜色时，可以向 setColor()指定一个 osg::Vec4 保存的 rgba 色彩值。下面代码将 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 库中 Text 类和 Font 类还允许用户设置一些其它参数，不过这已经超出了本书的讨论范围。更多信息可以通过追踪头文件 include/osgText/Text 及 include/osgText/Font 来获取，一定会有所斩获。

2.6.3 Text示例代码

本书源代码附带的文字示例代码演示了在一个几何体旁边放置文字的方法。代码创建了一个只包含单个 Geode 的简单场景图。Geode 包含四个 Drawables 对象，其中包括位于 XZ 平面上的一个采用高氏着色（Gouraud-shaded）的四边形和三个 Text 对象。其中，两个 Text 是面向屏幕的 Billboard 对象，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。

和本章的其他例子类似，文字示例程序只是简单地创建了场景图并将其写入.osg 文件中。要观察

运行结果，可以使用下面的命令：

```
osgviewer Text.osg
```

图 2-12 所示为 osgviewer 中所示的场景图。

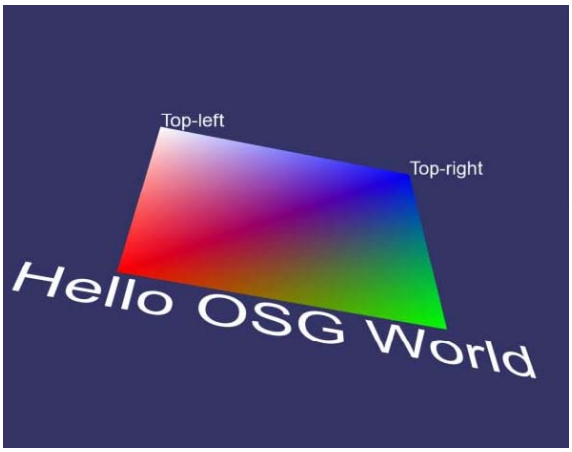


图 2-12 Text 例子运行结果

本例子采用两种不同的分辨率和不同的文字朝向(SCREEN and XZ_PLANE)创建了三个文本对象。

2.6.4 .osg文件格式

所有的 NodeKit 均包括两种格式的库。第一种实现是 NodeKit 的主要功能，如一个继承了 Node、Drawable 或 StateAttribute 的新类。应用程序必须链接该库以使用新的功能类。第二种库是对.osg 的封装类，即允许新类读取和写入.osg 格式的文件。

.osg 文件格式是标准的 ASCII 文本，并非为实现高效存储和读取数据而设计。它是一种极好的调试工具。本章的所有示例都把其场景图写入了.osg 文件。在渲染过程中这是没有必要的——实际应用程序会自行创建（或从文件中加载）场景图并实现渲染和显示。例子程序中使用.osg 文件，主要目标是为了演示场景图调试技术。

在进行程序开发时，如果遇到了不可预料的错误，可以尝试将子场景或者整个场景图作为.osg 文件输出。然后通过文本编辑器打开.osg 文件检查渲染出错的原因。如果认为某处可能是问题所在，直接采用手动修改.osg 文件，再运行 osgViewer 验证判断是否正确。

许多 OSG 开发者使用 OSG 用户邮件列表来咨询自己遇到的渲染问题。在邮件中提供一份包含出错子场景信息的.osg 文件，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 显示的是 Text 示例程序的输出的.osg 文件。其中包含了 osg 库所保存的 Geode 节点、Geometry 类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。

```
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 顶层节点是一个 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。后面是参数 num_drawables，其值被设置为 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 所需的所有参数和数据，包括 Geomtry 需要的一些额外的参数。

Geometry 对象之后是三个 Text 对象。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 对象包括一个参数 rotation，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参数，例如颜色（设置为白色）、位置值、字体文件名称等。

作为试验，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。在调试灯光时，如果灯光散射颜色太暗，此时可以通过修改.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 在最后对 Text 对象进行渲染，并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，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

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的这种方法。

3 运用OSG编程

真正的 OSG 应用程序要完成的工作绝不仅仅是场景图的构建和将场景图写入文件。本章将探讨将 OSG 集成到应用程序中的方法。通过对本章的学习，读者一方面可以学会场景显示、观察视角改变等基本操作的方法，另一方面，还可以掌握对场景实体的拾取以及动态修改场景数据的方法。

3.1 渲染

由于 OSG 开放了所有的功能细节，用户程序可以使用最底层的 OSG 功能来执行渲染操作。假设有读者希望能够完全自主地控制场景图的渲染，完全可以按照下面步骤编写应用程序代码：

- 设计视点管理代码，以动态改变 OpenGL 的模型-视图（model-view）矩阵。
- 创建一个窗口和 OpenGL 上下文，并将其激活。如果需要的话，编写管理多窗口和多个设备上下文的代码。
- 如果程序需要使用分页数据库（paged database），启用 osgDB::DatabasePager 类。
- 实例化 osgUtil::UpdateVisitor，osgUtil::CullVisitor 和 osgUtil::RenderStage 对象，由此实现场景的更新、剔除和绘制遍历。如果希望获得更多的控制权，可以自行编写类来实现以上遍历的特性。
- 编写主循环过程代码，处理操作系统系统返回的事件，通过调用自行编写代码来更新模型-视图矩阵。
- 在渲染前调用 glClear()初始化渲染上下文。依次执行场景更新、剔除和绘制遍历，最后交换缓存数据。
- 如果程序需要立体显示（stereo rendering）或者多通道显示（multipipe rendering）功能的支持，自行编写额外的代码。
- 遵循平台无关性的原则编写全部代码，以实现跨平台的目标。

上述步骤并非不可行，但十分繁琐和浪费时间，而且会带来一些问题。由于按上述步骤编写程序时使用了底层接口，当 OSG 的后继版本修改了这些底层函数之后，用户程序可能会与 OSG 的最新版本不兼容。

幸运的是，OSG 处于不断完善之中，一些更为简捷功能函数不断地合并到现有版本中，为用户提供编程的方便。在使用 OSG 编程时，通常会用到下列工具和类库：

- **osgUtil::SceneView**——这个类封装了更新、剔除和绘制遍历操作，但是不支持 DatabasePager 类。部分应用程序使用了 SceneView 作为 OSG 渲染的主接口。
- **Producer和osgProducer**——Producer是一个外部的摄像机视点管理[20]库，支持多通道显示。osgProducer应用库集成了Producer和OSG，而前者Producer有着相当壮大的用户群。目前，很多OSG程序是基于Producer和osgProducer开发。

OSG 2.0 版本添加了一个新成员——osgViewer库。osgViewer包含了一系列用于视口显示控制的相关类，封装了大量常用的功能函数，如显示管理、事件响应、场景渲染等。这个库使用osg::Camera类

[20] Camera library，这里主要指视点管理，因此译为视点管理库。

来管理 OpenGL的模型-视图矩阵。与SceneView类不同，osgViewer库的观察者类（Viewer）提供了对DatabasePager的全功能支持。osgViewer还提供了对同一个场景的多个独立视口显示功能的支持[21]。

第 1.6.3 节组件一节中，对 osgViewer 的两个观察者类 Viewer 和 CompositeViewer 作了一个大概的介绍。本章将对如何在程序中使用 Viewer 类实现 OSG 渲染功能问题继续加以介绍。

3.1.1 Viewer观察者类

Viewer 类的示例程序可以在本书附带源代码中找到，它包含了在应用程序中渲染 OSG 图形所需的最短代码。例中 Viewer 将实例化一个 osgViewer::Viewer 对象，尔后将场景图关联到该对象，并进行渲染显示。可以看出，源代码中简短高效，如清单 3-1 所示。

清单 3-1 Viewer 示例

下面列表显示了 OSG 程序的最小示意代码。

```
#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 节中中 osgviewer 命令行指令：

```
osgviewer cow.osg
```

这一相似并不是巧合。从原因来看，osgviewer 同样使用了 Viewer 类来实现场景的渲染。不过，osgviewer 为 Viewer 增加了许多附加功能。实际上 osgViewer 程序内容比清单 3-1 中的代码显示的要丰富得多。

- **视口更新**

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（视点）控制对象与 Viewer 关联。如果应用程序没有做这项工作，那么 Viewer::run()会创建一个 osgGA::TrackballManipulator 对象来控制摄像机（视点）。osgGA 库定义了数个常用的控制器类。编程时可以通过调用 Viewer::setCameraManipulator()来定义控制器。

• 设置 Camera 对象的投影矩阵和观察矩阵为自定义矩阵。这样也可以保证用户能够完全控制视口的动作。

如果程序需要直接设置 Camera 的显示矩阵，那就不能直接使用 Viewer::run()函数。该函数不允许实时地逐帧改变视口参数。相应地，用户可以编写一个小循环，在循环中反复更新视口并渲染单帧图像。

[21] View 与 Viewer：View 在 OSG 中类似于 VP 的 View，也就是执行显示的视口，该视口依附于其他窗体存在，与本窗体其他的视口共享消息循环，如对话框中的编辑框；Viewer 类似于 VP 中的观察者，也就是场景中一个独立的视点，或是 camera，也指程序中的指针，这里译为观察者类，观察器类。

清单 3-2 中程序列出了实现方法。

清单 3-2 直接控制视点

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

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

// Create a matrix to specify a distance from the viewpoint. 定义当前位置到视点距离的转换矩阵
osg::Matrix trans;
trans.makeTranslate( 0., 0., -12. );

// Rotation angle (in radians) 转旋角度（单位：弧度）
double angle( 0. );

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

    // Set the view matrix (the concatenation of the rotation and translation matrices).
    // 设置观察矩阵（旋转矩阵与变换矩阵之积）
    viewer.getCamera()->setViewMatrix( rot * trans );
    // Draw the next frame. 绘制下一帧
    viewer.frame();
}
```

第 2.2 节Geodes节点和几何体一节中，简要介绍了OSG的默认世界坐标系坐标轴方向。默认摄像机（camera，实际上这里指的是视点）的坐标系为世界坐标系，其基准方向是：X轴正向朝右，Z轴正向朝上，Y轴正向指向屏幕内部[22]。下面的文字描述了如何改变默认摄像机（camera）的显示矩阵。

清单 3-2 中的代码在渲染循环外设置过一次 Camera 的投影矩阵。Camera 类提供数个指定投影矩阵的方法，对于多数 OpenGL 开发者而言，下面这些方法应当都似曾相识。

```
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()方法以 osg::Matrix 对象为输入参数，这与下面的 OpenGL 命令是类似的：

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

[22] 也就是观察者的前方，整个坐标系为左手系。注意与 OpenGL 不同，与数学坐标系、VP 相同。

Camera::setProjectionMatrixAsOrtho()方法使用与 OpenGL 命令 glOrtho()同样的算法创建投影矩阵。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 的显示矩阵以更新旋转角度值。同样地，Camera 类也提供了一些为 OpenGL 开发者所熟悉的类似函数。清单 3-2 的代码显式地使用 setViewMatrix()方法来设置显示矩阵，而 Camera 类还提供了相似的 setViewMatrixAsLookat()方法，其输入参数类与 OpenGL 的 gluLookAt()函数也相似。

- **设置清屏色**

除视口设置外，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。下面的代码将清屏颜色设置为黑色：

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

缺省情况下，Camera 会清除深度缓存和颜色缓存。如果要改变这一缺省特性，可以使用 Camera::setClearMask()方法并传递适当的 OpenGL 缓存设置标志。

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

上面的代码段在帧循环每一帧开始时清除颜色、深度和模板缓存。

3.1.2 CompositeViewer类

osgViewer 还提供了额外的视景显示类，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次上对该类进行阐述。

如果说Viewer类只用单一显示视口来表现场景（多通道渲染可能需要使用一组摄像机，），CompositeViewer类则可以支持一个或多个场景的多通道显示，并且允许用户程序指定不同通道的渲染顺序。CompositeViewer还支持渲染到纹理（render-to-texture，RTT）操作，即允许用户程序将一个视口中渲染的图像作为后继显示通道的纹理贴图[23]。

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

3.2 SG动态修改

OSG 允许用户动态的修改场景图，从而产生显示上的变化。这一特性是任意交互式的图形应用程序所必须具备的。通过 OSG，用户可以更改几何数据、渲染状态参数、Switch 节点设置甚至于场景图的结构。

正如第一章所述，剔除遍历中在渲染图形中保存了几何数据的引用和渲染状态信息，供随后的绘制遍历处理。osgViewer 库支持多线程模式，也就是说，剔除与绘制遍历在多个线程中的一个或多个中进行。为了达到性能上的最优化，OSG 没有使用内存锁技术来确保线程的安全。相反地，OSG 要求**应用程序只能在剔除和绘制遍历之外对场景图进行修改**。

[23] 窗口、视口、通道的关系。窗口是指某操作系统下一个窗体，具有显示、消息循环等功能与属性。视口从本质上来说也是一个窗体，但是其不具备单独的消息循环、输入输出设备等。窗口可以包括一个或多个视口。通道是与显示设备有关的区域，它是屏幕上的一个矩形部分。因此，通道可以显示窗口或视口的全部或部分。David 根据理解注解。

这个问题的关键在于必须确保用户修改不会与剔除及绘制线程发生冲突，人们也找到了几种方法。一个简单的方案是：**在 Viewer::frame()的调用之外进行场景图的修改**，这需要在主渲染循环中添加额外的代码。如果读者希望寻求一个更加整洁规范的解决方案，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动态更改相关的技术。

• 出于性能优化和线程安全方面的考虑，用户需要通知 OSG 可能要修改场景图的哪些部分。用户可以通过设置 Object 对象（Node、Drawable、StateSet 等）的**数据变异属性**（data variance）来实现。

• OSG 允许用户程序为 Node 和 Drawable 设置回调函数（callback）。OSG 将在特定遍历中执行这些回调函数。如要在更新遍历中修改 Node 或者 Drawable 对象的值，通过设置更新回调（update callback）即可完成。

• 应用程序不可能始终可以预知需要修改场景图的哪一部分，这时可能会需要搜索整个场景图来查找特定的节点，或者由用户使用鼠标等输入机制来选择一个节点。

后续文字将对上述三种方法一一进行描述。

3.2.1 数据变异

osgViewer 支持的多线程模型允许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重叠。当仔细考虑该线程模型的实现时，看上去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。

在设置一个 Object 对象的数据变异属性时，可以用 Object::DataVariance 的枚举值为输入参数，调用函数 setDataVariance()。该属性的初始值是 **UNSPECIFIED**。应用程序可以将其更改为 **STATIC** 或者 **DYNAMIC** 中的一个。

程序崩溃的原因

当编写动态修改场景图的代码时，程序执行修改场景图的过程中可能会遇到程序崩溃或非法访问，这类崩溃多数是因为在场景图剔除或是绘制遍历阶段时修改场景图所致。

OSG 要求绘制遍历只有在所有属性为 **DYNAMIC** 的 Drawable 和 StateSet 都处理完成后才返回。由于绘制遍历在函数返回后仍然继续渲染场景图，但是此时**渲染图形**中只有属性为 **STATIC** 数据。如果用户的场景中包含了很少 **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。

如果应用程序试图修改非 **DYNAMIC** 数据，如 Drawable 或 StateSet 数据，可能会遇到因多个线程同时访问同一数据导致冲突。多数操作系统采用放弃对该程序处理的方式来解决此问题。因此，应当将需要修改的数据都设置成 **DYNAMIC** 属性。

渲染图形表只包括对 Drawable 或 StateSet 对象的引用。然而，如果应用程序需要修改场景图中的一个节点，如 Switch 节点的子节点切换操作，应当设置节点的数据变异属性为 **DYNAMIC**。该操作可以防止 osgUtil::Optimizer 改变程序中的场景图结构。

3.2.2 回调函数

OSG 允许应用程序设置 Node 和 Drawable 对象的回调函数。OSG 在更新和剔除遍历时执行 Node 对象的回调函数，而 Drawable 对象的回调函数则在剔除和绘制遍历时执行。本节将介绍在更新遍历中如何使用 osg::NodeCallback 动态修改 Node 节点的方法。OSG 的回调函数接口则基于回调函数设计模式[Gamma95]。

为使用 NodeCallback，应用程序必须进行下面的步骤：

- 从 NodeCallback 派生一个新的类。
- 重载 NodeCallback::operator()方法。使用这个方法来实现场景图的动态更改。
- 实例化从 NodeCallback 继承的类，然后使用 Node::setUpdateCallback()方法将其关联到将要修改的 Node 节点。

在每次更新遍历过程中，OSG 都会调用派生类中的 operator()方法，从而允许应用程序对 Node 进行修改。

OSG 向 operator()方法传递了两个参数。第一个参数是回调函数所关联的 Node 节点的地址，也就是用户回调函数在 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，目前你可以暂时忽略该参数。

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 );
```

多个节点可以共享回调函数。NodeCallback 类间接派生自 Referenced，而 Node 类内部维护了一个更新回调函数的 ref_ptr<>指针。当最后一个关联此回调函数的节点被删除时，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函数的例子，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 节点相关联。代码将从 NodeCallback 派生出一个类，并将其关联到其中一个 MatrixTransform 对象。在更新遍历中，新的 NodeCallback 动态改变了矩阵值，使其中一头牛的模型不断旋转。回调示例程序的输出结果请见图 3-1。

图 3-1 回调函数例程运行结果

这幅图显示了回调函数例程的输出结果。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 列出了示例代码，它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。注意当函数中创建了第一个 MatrixTransform 对象时（称为 mtLeft），它通过调用 mtLeft->setUpdateCallback(newRotateCB)来指定 mtLeft 的更新回调。如果用户将这一行注释掉，重新运行例子程序，可以发现牛的模型不再转动了。程序的最后部分是 main()函数入口，用于创建一个视口并进行渲染。

表 3-3 回调函数示例代码

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的过程。

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

// Derive a class from NodeCallback to manipulate a MatrixTransform object's matrix.
// 从 NodeCallback 派生新类用于处理 MatrixTransform 对象的矩阵
class RotateCB : public osg::NodeCallback
{
public:
    RotateCB() : _angle( 0. ) {}
    virtual void operator()( osg::Node* node, osg::NodeVisitor* nv )
    {
        // Normally, check to make sure we have an update visitor, not necessary in this simple example.
        // 通常，检查并确信需要一个更新访问器，对本例并非必要
        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 );

        // Increment the angle for the next from.
        _angle += 0.01;
        // Continue traversing so that OSG can process
        // any other nodes with callbacks.
```

```
        traverse( node, nv );
    }
protected:
    double _angle;
};
// Create the scene graph. This is a Group root node with two
// MatrixTransform children, which both parent a single
// Geode loaded from the cow.osg model file.
osg::ref_ptr<osg::Node> createScene()
{
    // Load the cow model.
    osg::Node* cow = osgDB::readNodeFile( "cow.osg" );
    // Data variance is STATIC because we won't modify it.
    cow->setDataVariance( osg::Object::STATIC );
    // Create a MatrixTransform to display the cow on the left.
    osg::ref_ptr<osg::MatrixTransform> mtLeft = new osg::MatrixTransform;
    mtLeft->setName( "Left Cow\nDYNAMIC" );
    // Set data variance to DYNAMIC to let OSG know that we
    // will modify this node during the update traversal.
    mtLeft->setDataVariance( osg::Object::DYNAMIC );
    // Set the update callback.
    mtLeft->setUpdateCallback( new RotateCB );
    osg::Matrix m; m.makeTranslate( -6.f, 0.f, 0.f );
    mtLeft->setMatrix( m );
    mtLeft->addChild( cow );
    // Create a MatrixTransform to display the cow on the right.
    osg::ref_ptr<osg::MatrixTransform> mtRight = new osg::MatrixTransform;
    mtRight->setName( "Right Cow\nSTATIC" );
    // Data variance is STATIC because we won't modify it.
    mtRight->setDataVariance( osg::Object::STATIC );
    m.makeTranslate( 6.f, 0.f, 0.f );
    mtRight->setMatrix( m );
    mtRight->addChild( cow );
    // Create the Group root node.
    osg::ref_ptr<osg::Group> root = new osg::Group;
    root->setName( "Root Node" );
    // Data variance is STATIC because we won't modify it.
    root->setDataVariance( osg::Object::STATIC );
    root->addChild( mtLeft.get() );
    root->addChild( mtRight.get() );
    return root.get();
}

int main(int, char **)
{
    // Create the viewer and set its scene data to our scene
    // graph created above.
    osgViewer::Viewer viewer;
    viewer.setSceneData( createScene().get() );
    // Set the clear color to something other than chalky blue.
    viewer.getCamera()->setClearColor( osg::Vec4( 1., 1., 1., 1. ) );
    // Loop and render. OSG calls RotateCB::operator()()
    // during the update traversal.
    viewer.run();
}
```

RotateCB::operator()中包含了对 traverse()的调用，它是 osg::NodeVisitor 类的方法。该回调函数允

许更新遍历（**osgUtil::UpdateVisitor**）在当前组节点的子节点之间进行切换。采用 **NodeCallback** 进行帧前遍历或帧后遍历，是 OSG 遍历回调申请的设计特性之一，与遍历回调函数代码的位置息息相关。忽略执行当前节点的回调函数，会阻止 OSG 执行子节点的回调函数。在后续章节中会对 **NodeVisitor** 进行更详细的介绍。

图 3-2 显示了回调示例创建的场景图。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 对象设置了数据变异属性为 **DYNAMIC**，而另一个由于不会修改，因此使用了 **STATIC** 数据变异属性。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。

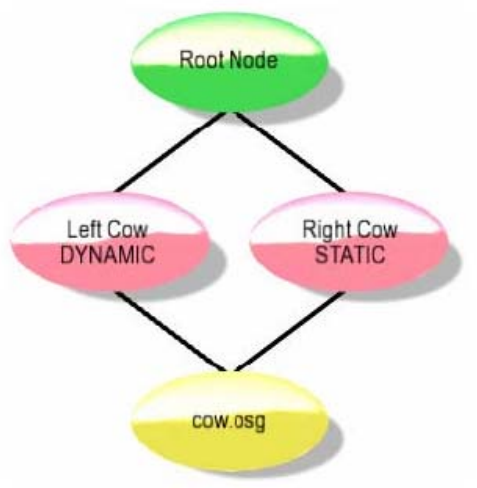


图 3-2 回调函数构建场景图的结构

本图所示为回调函数示例的场景结构图。注意两个 MatrixTransform 节点的数据变量并不相同。

象示例代码中演示的那样，关联更新回调到一个已知节点方式很容易，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内部的某个节点，或者用户要交互地进行选择的时候，问题就显得比较复杂了，后面一节将给出一些 OSG 运行时节点识别的方法。

3.2.3 NodeVisitors类

NodeVisitor类是OSG中访问器（迭代器）设计模式[Gamma95]的具体实现。从本质上说，NodeVisitor类遍历了场景图，访问每一个节点并调用其函数。该类使用技术简单有效，是许多OSG操作类的基类[24]，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。在前面几节的代码中，UpdateVisitor正是调用了NodeVisitor类的NodeCallback::operator()方法。总之，NodeVisitor在OSG的应用中无处不在。

NodeVisitor类是一个具有虚函数性质的基类，应用程序无法直接将其实例化。但是用户程序可以使用OSG提供的任何类型的NodeVisitor派生类，也可以自己编写继承自NodeVisitor类的代码。NodeVisitor类包含了适应于多数OSG节点类型的数个重载[25]的apply()方法。当NodeVisitor对象遍历场景

[24] 原文为 This simple technique exists as a base class for many OSG operations，比较难译。
[25] 关于重载（overload），重写(override)，派生与继承(derive)的关系。

图时，会根据访问的节点调用其相应的apply()方法。对于自定义的NodeVisitor 只需要重载需要处理节点类型的apply()方法。

程序读取文件并加载场景图之后，根据需要从场景图中搜索相应的节点。举例来说，假设有一个机器人手臂模型文件，其每一个接合处都建立了作为关节的变换节点。那么在读取该文件后，程序可能会使用 NodeVisitor 节点来定位所有的 Transform 节点，以实现动画效果。在这种情况下，用户程序可以定制 NodeVisitor 派生类并重载 apply(osg::Transform&)方法。当用户 NodeVisitor 派生类对象遍历整个场景图的时候，会执行各个派生 Transform 节点的 apply()方法，程序执行该节点的必要操作以支持动画，如在列表中保存节点的地址。

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 NodeVisitor::TRAVERSE_ALL_CHILDREN 来初始化基类，以允许遍历。否则，OSG 将不会调用用户的 apply()方法。

如果 NodeVisitor 对 apply()方法进行了多次重载，OSG 会调用该节点最合适的 apply()方法。例如，Group 节点从 Node 节点派生。程序对特定名称的节点搜索是一项简单而常用的操作。清单 3-4 中列出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地址。

NodeVisitor 重载了两种形式的 apply 函数，分别是 apply(Node&)和 apply(Group&)。NodeVisitor 遍历时如遇到 Group 节点或是从 Group 节点派生的节点，OSG 会调用 apply(Group&)。如果 OSG 遇到 Geode 节点，在本示例中，OSG 会调用 apply(Node&)，这是因为 Geode 从 Node 节点派生而不是从 Group 节点派生。

表 3-4 FindNamedNode 类定义

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。

```
// 从 NodeVisitor 类派生一个新类，根据节点名称查找节点。
class FindNamedNode : public osg::NodeVisitor
{
public:
    // Traverse all children.遍历所有子节点
    FindNamedNode( const std::string& name ) : osg::NodeVisitor(
    osg::NodeVisitor::TRAVERSE_ALL_CHILDREN ), _name( name ){}
    // This method gets called for every node in the scene
    // graph. Check each node to see if its name matches
    // our target. If so, save the node's address.
    // 下面方法调用场景图的每个节点，检查节点名称是否是要查找的节点，是则返回节点地址。
    virtual void apply( osg::Node& node )
    {
        if (node.getName() == _name)
        _node = &node;
        // Keep traversing the rest of the scene graph. 继续遍历其他节点
        traverse( node );
```

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

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

traverse()方法是 NodeVisitor 的一个成员函数。该函数与 3.2.2 回调函数一节中的 NodeCallback 同名成员函数不同又有些相似。遍历场景图时，NodeVisitor 使用以下的规则：

- NodeVisitor 属性配置成 **TRAVERSE_ALL_CHILDREN** 则遍历所有的子节点；
- 自定义的 NodeVisitor 类重载的一个或多个不同形态的 apply()方法，负责调用 NodeVisitor::traverse()函数来遍历节点的子节点。该需求允许自定义 NodeVisitor 进行帧前或帧后遍历，或者在需要的时候停止遍历。
- 由 NodeVisitor 类执行的回调函数的使用方法，与前几节描述的更新回调函数类似。NodeVisitor 遍历节点的子节点并不使用回调函数。NodeVisitor 也不通过子节点关联的回调函数来访问子节点。相反，如果应用程序需要进行遍历子节点时，只通过回调函数运算符()调用 NodeCallback:: traverse()即可。

如果要使用 NodeVisitor 遍历场景图，可以将 NodeVisitor 作为 Node::accept()的输入参数传递。可以在任何一个节点上调用 accept()，NodeVisitor 将从该节点开始遍历整个场景图。如果要搜索整个场景图的话，可以从根节点开始调用 accept()。

清单 3-4 所示 FindNamedNode 类是 FindNode 示例程序的一部分。FindNode 示例程序从磁盘中加载一个场景图文件，查找这个场景图中指定名称对应的节点，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 设置程序协同工作，而后者可以将场景图输出到文件。当 FindNode 读取文件之后，它将查找 StateSet 设定为平面着色（flat shading）的 MatrixTransform 节点，而后改变其渲染状态为平滑着色(smooth shading)。

3.2.4 拾取操作Picking

由于需要为用户提供场景选取功能，很多三维应用程序需要提供对拾取功能的支持。对拾取操作来说，最简单的形式是用户在场景显示的实体位置上定位鼠标并进行点击操作。从内部看，程序将鼠标在屏幕上的二维（x,y）位置映射到对应的三维场景图节点，并保存节点地址为后续应用做准备。

从本质上看，基于 OSG 的三维应用程序执行以下两步来进行拾取操作：

- 接收鼠标事件。osgGA 库提供事件类允许 OSG 应用程序以平台无关的方式接收鼠标事件。
- 检查鼠标箭头的当前位置对应了场景图的哪个节点。osgUtil 库提供了相交检测类，该类通过以当前鼠标位置（x,y）值创建检测体（Volume），检查它是否与场景图节点相交。osgUtil 返回按从前至后排列与场景图相交的节点列表。

本节描述了如何实现上面两个步骤的操作。

➢ **捕捉鼠标事件**

如同 1.6.3 组件一节中描述，osgGA 库提供了对平台无关的界面（GUI）事件的支持。本章示例使

用 osgGA::TrackballManipulator 操作显示矩阵，TrackballManipulator(轨迹球运动模型)将鼠标事件做为输入，并对视点的 osg::Camera 显示矩阵进行修改。

TrackballManipulator 运动模型从 osgGA::GUIEveentHandler 派生。osgGA::GUIEveentHandler 为一个抽象基类，不能直接实例化。相应地，应用程序需要自行从 GUIEveentHandler 派生新类执行基于 GUI 的事件。为了执行基于鼠标的拾取等相关操作，需要从 GUIEventHandler 派生新类并且重载 GUIEventHandler::handle()方法。实例化该类并与应用程序视点相关联。

方法 handle()方法有两个参数，即 osgGA::GUIEventAdapter 和 osgGA::GUIActionAdapter，如示例程序中显示：

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

在 GUIEventAdapter 中，应用程序重载的 handle()方法接收了包括鼠标事件在内的所有 GUI 事件。在 GUIEventAdapter 头文件中声明了 EventType 枚举变量，通过它可以方便地获取感兴趣的事件，如鼠标事件等。应用程序可以通过 GUIEventAdapter::getEventType()接口获得事件类型。

GUIActionAdapter 是应用程序返回 GUI 的接口。鼠标拾取时，程序的 GUIEventHandler 与 viewer 类关联。因此，通过与 viewer 相关 GUIActionAdapter，程序可以对当前视口的场景进行相交检测。

在渲染 viewer 之前，应用程序需要创建自定义 GUIEventHandler 类，然后采用 Viewer::addEventHandler()方法将其与 viewer 相关联。如其名字含义的那样，viewers 可以有多个事件处理函数，而且 viewer 可以将自定义事件处理函数添加到可能事件处理函数列表中。Viewers 调用为每个 GUI 事件的 handle()函数直到某个 handle()函数返回 true。

列表 3-5 代码包含了一个从继承于 GUIEventHandler 的名为 PickHandler 的派生类。该派生类的 handle()方法支持鼠标点击、移动和释放等鼠标事件。它记录了鼠标点击或移动事件的屏幕 x、y 坐标位置，如果鼠标暂停移动，此时释放鼠标事件将触发一个拾取事件。如果拾取成功，handle()函数返回 true，否则返回 false 以便于让其他的事件处理函数进行事件的处理。

需要注意的是，只有当鼠标不再移动且释放鼠标键才会执行拾取操作。这允许传递鼠标移动事件到其他事件处理函数，如 TrackballManipulator 运动模型。

表 3-5 PickHandler 类

为在 OSG 中实现拾取操作，程序需要使用从 osgGA:GUIEventHandler 派生的子类。列表显示了拾取示例程序中的 PickHandler 类。该类定义了两个成员函数，一个负责接收鼠标事件，另一个负责在鼠标释放时处理拾取操作。

```
// PickHandler – 实现拾取操作的 GUIEventHandler
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:
        {
          //鼠标点击和移动事件时记录
          _mX = ea.getX();
          _mY = ea.getY();
          return false;
        }
        case osgGA::GUIEventAdapter::RELEASE:
        {
          // 如果最后一次鼠标点击或移动事件后，鼠标没有再移动，则进行拾取操作
          // 否则 trackball 模型进行处理
          if (_mX == ea.getX() && _mY == ea.getY())
          {
            if (pick( ea.getXnormalized(), ea.getYnormalized(), viewer ))
            return true;
          }
          return false;
        }
        default:
          return false;
        }
    }
protected:
    // 保存鼠标按键或移动事件时鼠标(x,y)位置
    float _mX,_mY;
    // 执行拾取操作
    bool pick( const double x, const double y, osgViewer::Viewer* viewer )
    {
    if (!viewer->getSceneData())
    // 如果没有检测到相交，返回 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())
    {
      Const osg::NodePath& nodePath = picker->getFirstIntersection().nodePath;
      unsigned int idx = nodePath.size();
      while (idx--)
      {
          // Find the LAST MatrixTransform in the node path; this will be the MatrixTransform
          // to attach our callback to.
          // 在节点路径中寻找最后一个转置矩阵，关联回调函数转置矩阵
          osg::MatrixTransform* mt =dynamic_cast<osg::MatrixTransform*>( nodePath[ idx ] );
          if (mt == NULL) continue;
          // If we get here, we just found a
          // MatrixTransform in the nodePath.
          // 到达此位置时，已经找到转置矩阵
          if (_selectedNode.valid())
          // Clear the previous selected node's callback to make it stop spinning.
          // 清空前一选择节点的回调函数以停止其旋转
```

```
            _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 )
{
    // 建立场景图视点
    osgViewer::Viewer viewer;
    viewer.setSceneData( createScene().get() );
    // 添加拾取操作
    viewer.addEventHandler( new PickHandler );
    return viewer.run();
}
```

清单 3-5 中列出了 Picking 示例程序的 main()函数，演示了使用 Viewer::addEventHandler()方法关联事件处理器到 viewer 的方法。

综上所述，要实现接收鼠标事件并实现拾取功能，需要经过以下几个步骤：

- 从 GUIEventHandler 继承新类，重载 handle()方法。
- 在 handle()方法中，检查 GUIEventAdapter 参数传递的事件类型，选择需要的事件类型并执行相应的操作。方法返回 true 时将阻止其它事件处理函数接收事件。
- 在渲染之前，创建事件处理器的实例，并将 addEventHandler()方法添加到 viewer 中。OSG 将会把 viewer 作为 GUIActionAdapter 参数传递给 handle()方法。

上述技术并不仅限于使用鼠标进行拾取。应用程序可以尝试实现和 TrackballManipulator 类（轨迹球运动模型）相似方式进行鼠标事件接收处理类。另外，程序也可以接收键盘事件并实现对按键的响应操作。

下面部分将阐述如何确定鼠标点击位置与场景图的哪个部分相交，然后结束对鼠标拾取操作问题的讨论。

➢ 相交检测

鼠标点击拾取可以将想象成是从鼠标箭头位置向场景发出一条射线。被鼠标选中的部分就是场景与该射线相交的部分。如果场景是由点、线等元素组成，那么采用**射线相交测试法**可能无法达到用户要求，因为射线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，射线相交检测法的精度将与视点目标距离成反比。

为解决上述问题，OSG使用一种多面体（polytope）的金字塔形包围盒检测体[26]代替射线。视点位于金字塔检测体的顶点，金字塔检测体的中心轴通过鼠标箭头的位置。检测体外缘宽度与该位置到视点距离有关，并可根据需要调整其参数。

OSG 采用了场景图的自顶向下的继承结构，可以在本地 CPU 上进行高效的计算，从而避免了 OpenGL 拾取时普遍存在“迟钝”特点。osgUtil::IntersectionVisitor 类派生自 NodeVisitor，它不仅可以检测每个节点包围盒与金字塔包围盒检测体的关系，而且当某图形节点的子节点不可能与检测体有交集时，跳过对子场景图的遍历。

用户可以将 IntersectionVisitor 类配置为使用多种不同几何结构（检测体）进行交集检测，例如平面和线段。其构造函数采用 osgUtil::Intersector 作为输入参数，其原因主要在于 Intersector 类可以定义相交检测的几何体（检测体）并执行实际的相交测试。Intersector 是一个纯虚基类，用户程序不能直接将其实例化。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、较理想的鼠标拾取模型类。

实际中，有些程序需要拾取独立的顶点和多边形；而有些程序只简单地需要获取被选几何体的 Group 或 Transform 父节点。为满足这些不同的需求，IntersectionVisitor 返回了一个 osg::NodePath 对象。而 NodePath 是一个 std::vector<osg::Node>向量，它表示沿着从根节点到叶节点路径的向量。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，鼠标位置使用 GUIEventAdapter 存储的标准化数据。
- 创建 IntersectionVisitor 对象，并将 PolytopeIntersector 作为其构造函数的输入参数。
- 从场景图根节点开始启动 IntersectionVisitor，通常是通过 Viewer 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 检测有相交事件，根据返回的 NodePath 并以此搜索到符合要求的节点。

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。不过，前者的场景图在层次上使用了两个 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。对一次成功的选择操作，例子代码将搜索 NodePath 直到遇到 MatrixTransform 节点。该操作为该节点关联更新回调，并动态地旋转子节点几何体。

运行 Picking 示例程序时，程序运行结果是将显示两头牛的模型，这一点与回调示例程序相同。相应地，用户任意选择一头牛的模型时，程序会响应并开始进行旋转。

[26] 检测体，类似于 Volume 等单词，指用于检测与场景相交的图元、平面或几何体。David。

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许多 OSG 开发者的推崇的附加资源信息。

A.源代码

从开发者的角度来看，开源产品的主要优势在于开放源码。在用户开发基于 OSG 应用程序过程中，通过深入 OSG 源代码和对内部机制的研究，可以迅速解决遇到的问题。

如果读者还没有做到这一点，可以按照 1.2 节所述的步骤，下载完整的 OSG 源代码，并创建自己的 OSG 调试版运行库。首次编译 OSG 可能较为困难而且耗费时间较多，但必将为后续 OSG 学习打下良好的基础。

OSG 发行版代码中包括大量的实用示例程序，它们给出了许多 OSG 功能函数的正确用法，其内容远超出本书内容。这些示例程序对于任何一个 OSG 开发人员都具有较高的价值。

应当说，仅使用 OSG 执行库来开发应用程序，不存在任何问题。但是如果能灵活运用 OSG 源代码、示例程序以及调试版运行库，将大大加速软件开发进程。

B.OSG Wiki

OSG 维基网站[OSGWiki]包含了很多 OSG 相关的有价值内容，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。

http://www.openscenegraph.org/

C.用户邮件列表

OSG 用户邮件列表可以帮助读者与其他 OSG 用户和开发者进行联系。当用户编译 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

D.专业服务支持

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

E.词汇表

附表 1 词汇与注释

词汇名称	注　　释
.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”。
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（参见“场景图”）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。
组节点（Group node）	osg::Group 类提供了对常见场景图组节点概念的支持。它可以作为场景图的组节点或者根节点。许多场景图类都是从 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）	此类场景图节点没有子节点。在大部分场景图中，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节点类。
NodeVisitor 类	这个类用于遍历场景图，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	Drawable 及 StateSet 引用对象的集合。剔除遍历（cull）中将几何信息

词汇名称	注　　释
（Render graph）	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。在剔除遍历中，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。

F. 参考书目

类别	相 关 书 籍
[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。
[MacOSXTips]	OSG Apple QuickTime video documentation. http://www.openscenegraph.org/index.php?page=Tutorials.MacOSXTips
[Martz06]	Martz Paul：《OpenGL® Distilled》， Addison-Wesley，2006。
[OSGWiki]	OpenSceneGraph 维基网站， http://www.openscenegraph.org/
[OSGBooks]	http://www.osgbooks.com/
[Rost06]	Rost Randi：《OpenGL® Shading Language》，第二版， Addison-Wesley，2006。

G. 修订历史

14 June 2007 Improvements and clarifications regarding Apple Mac OS X. Changed book URL to

osgbooks.com. Some minor reformatting.

6 June 2007 Technical review and copyedit. Fixes many glaring errors and clarifies concepts and topics. Added the Index.

7 April 2007 Initial revision.