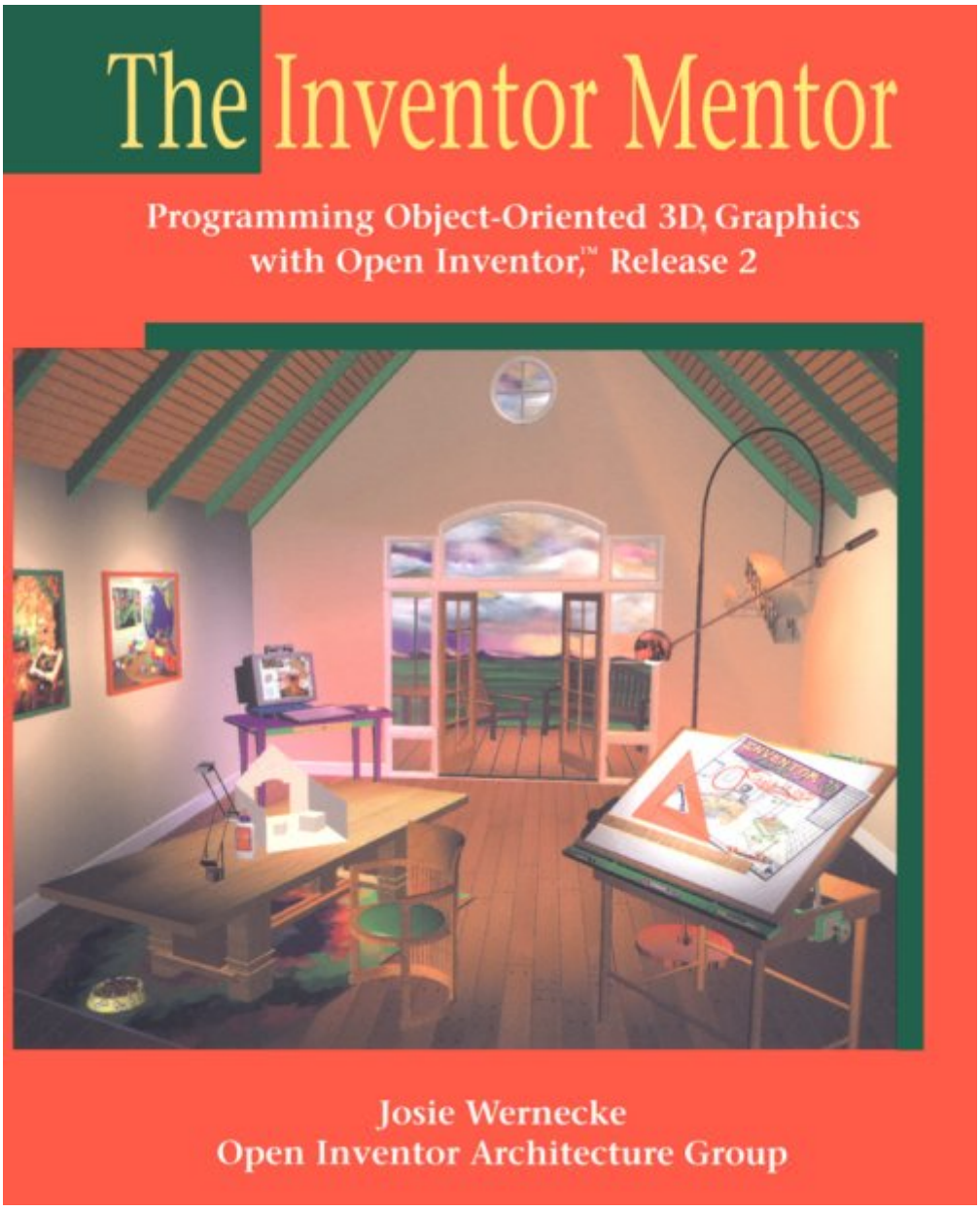


The Inventor Mentor

运用Open Inventor™ 进行面向对象的 3D图形开发，第二版

Josie Wernecke
Open Inventor 框架工作组
郝伟 译

目录

关于这本书

本书包括的内容
怎样使用本书
阅读本书前需要具备的知识
本书的约定用法
关键的场景图符号表
进一步阅读建议（参考文献）
致谢

彩图

第一章 综述

什么是 Open Inventor
只要对象,不用绘制 (Objects,not Drawings)
Open Inventor 与 OpenGL 的关系
Inventor 工具包
Inventor 组件库
Inventor 类树
扩展工具包

第二章 一个 Inventor 例子

“Hello, Cone”
命名约定
场景基本类型（Scene Basic Types）
Inventor 使用的坐标系统
包含文件

第三章 节点与组

场景数据库
图形场景
节点的类型
创建组节点
共享节点实例.
路径（Paths）
节点中的域(Fields)
重置(Override)标记 (高级内容)
引用和删除
节点类型
节点命名

第四章 照相机和灯光

使用灯光和照相机
照相机
灯光

第五章 形体，属性和绑定（Binding）
简单形体
复杂形体
属性节点
绑定节点
几何变换

第六章 文本
二维文本
三维文本

第七章 纹理
创建有纹理的对象
关键概念
SoTexture2 节点
将纹理映射到物体上

第八章 曲线与曲面
综述
关键概念
NURBS 曲线的例子
NURBS 曲面
进一步阅读建议

第九章 应用动作（Actions）
Inventor 动作
基本方法
应用动作
渲染
计算包围盒
累积计算几何变换矩阵
输出到文件
搜索节点
拾取
回调程序

第十章 处理事件和选择器
概述
事件处理的通用编程模型
Inventor 事件处理

节点如何处理事件：SoHandleEventAction
使用事件回调节点（*高级内容*）
直接向应用程序发送事件（*高级内容*）
选择器

第十一章 文件格式
输出图形场景
从文件中读入场景
文件语法格式
ASCII 和二进制版本
读取扩展的节点与引擎
从字符串读入场景

第十二章 传感器
传感器简介
数据传感器
其它的延迟队列传感器
定时队列传感器

第十三章 引擎
引擎简介
引擎的一般用途
引擎类型
域连接
引用计数
禁用连接
更新数据
全局域
动画引擎
闸门引擎
算法引擎
产生动画效果的节点

第十四章 节点工具包
为什么使用节点工具包
隐藏的子节点与 SoNodeKitPath
节点工具包类
节点工具包目录
缺省创建的部件
选择与设置部件数据
其它方法：getPart（）和 setPart（）
用于获取部件的宏
指定部件名称
创建指向部件的路径

使用部件列表
使用SoSeparatorKit创建多级运动场景
例子程序

第十五章 拖拽器和操作器
什么是拖拽器？
拖拽器的类型
操作器 VS 拖拽器
简单拖拽器
操作器
定制拖拽器（高级内容）

第十六章 Inventor 组件库
组件简介
Xt 工具函数
渲染区
Xt 组件
通用模式
观察器
使用 3D 剪贴板

第十七章在 Inventor 中使用 OpenGL
简介
OpenGL 状态变量与 Inventor
颜色索引模式
使用 SoCallback 节点
在一个 GLX 窗中使用渲染动作

附录 A：C 程序员面向对象编程入门
什么是面向对象的编程方法？
一个类的例子：球体
阅读建议

附录 B：C API 入门
命名 C 函数
缩写 C 函数名称
在 C 语言中创建和删除对象
重载 C++方法
调用函数
C 类和参考手册
一个用 C 语言编写的简单的 Open Inventor 程序

附录 C：错误处理
运行时刻类型检查

发送错误
处理错误
调试

关于这本书

The Inventor Mentor 这本书主要是面向图形编程人员和应用程序开发人员，讲解面向对象的 3D 开发包－Open Inventor。Open Inventor 是一个用于开发交互式 3D 程序的图形库。虽然它本身是使用 C++编写的，但 Open Inventor 也可以在 C 语言中使用。（**Although it is written in C++, Open Inventor also includes C bindings。bindings 的意思是联编。联编主要是指编译器如何将代码翻译成机器码的过程。在这里可以理解成 Open Inventor 允许在 C 语言环境中使用。译者注**）

为了简单起见，本书所带的例子代码都是使用 C++编写的。所有的 C++例子代码，包括等同的 C 例子代码，都可以从网络中下载得到。如果读者是一个 C++新手的话，请阅读附录 A“**C 程序员面向对象编程导论**”，它将帮助读者理解贯穿本书所使用的类、子类、和其它面向对象的概念。如果读者使用 C 语言来开发程序，请阅读附录 B“**C 语言 API 导论**”。

本书包括的内容

本书将讨论如何使用 Open Inventor 来开发应用程序。本书的姊妹书 *The Inventor Toolmaker* 是为高级程序员准备的，主要用于讨论怎样创建新的 Inventor 类和定制已经存在的类。

本书包括下列章节：

- 章节 1，**“综述”**,概述 Open Inventor 的基本概念和分类，讨论 Inventor 与 OpenGL，Inventor 与 X 窗口系统之间的关系。

- 章节 2，**“一个 Inventor 例子”**，首先创建一个包含有圆锥的简单程序。然后通过修改这个程序，向用户展示像“引擎”、“操作器”、和“组件”等这些重要的 Inventor 对象的用法。

- 章节 3，**“节点与组（Groups）”**，介绍场景图(scene graph)的概念，展示如何创建节点(nodes)，如何将节点增加到不同类型的“组”中。

- 章节 4，**“照相机和灯光”**，讨论用于观察场景的照相机节点和用于提供照明的灯光节点。

- 章节 5，**“形体，属性和绑定（Binding）”**，讨论如何创建简单的和复杂的物体，怎样使用包括材质（material），绘制风格（draw style）和光照模型等属性节点。同时也将解释如何将材质和表面法线绑定到形体节点上。

- 章节 6，**“文本”**，演示了 2D 和 3D 文本节点的用法。

- 章节 7，**“纹理”**，讨论如何将纹理“贴”到场景物体的表面上。

- 章节 8，**“曲线和曲面”**，解释如何使用 NURBS 曲线和曲面。

- 章节 9，**“应用动作（Actions）”**，描述怎样在 Inventor 的场景中使用动作。“动作”包括：OpenGL 渲染、拾取、计算包围盒、计算变换矩阵、写文件、在场景中查找某些特定类型的节点等。

- 章节 10，**“处理事件和选择器（Selection）”**，解释 Inventor 如何从窗口系统中接收事件。描述选择器节点如何管理选择列表和执行高亮选择。

- 章节 11，**“文件格式”**，阐述 Inventor 用于读入和写出的文件交换格式。讨论如何利用拷贝，粘贴等方式进行数据交换。

- 章节 12，**“传感器（Sensors）”**，描述 Inventor 的传感器节点如何监视某些类型的事件以及当这些事件发生时怎样调用用户提供的回调函数。

- 章节 13，**“引擎（Engines）”**，描述如何通过使用引擎节点来驱动场景中的物体。讨论如何创建场景中节点间的互相依赖关系。

- 章节 14，**“节点工具包（Node Kits）”**，介绍节点工具包。节点工具包可以很方便地创建一组有关联的 Inventor 节点。每个节点包都包括有一个用户想选择使用的节点目录（catalog）。

- 章节 15，**“拖拽器（Draggers）和操作器（Manipulators）”**，描述怎样使用拖拽器和操作器这两种特定的对象来响应用户事件。操作器是一种允许用户直接修改其自身域值（field Values）的节点。

- 章节 16，**“Inventor 组件库”**，演示如何使用 Inventor 的 Xt 组件。这个组件是一个内建的用于交互改变场景的用户接口。

- 章节 17，**“在 Inventor 中使用 OpenGL”**，讨论怎样在 Inventor 中使用 OpenGL 库。

书中有三个附录：

- 附录 A，**“C 程序员面向对象编程导论”** 阐述数据抽象和继承等面向对象编程的基本概念。
- 附录 B，**“C 语言 API 导论”**，解释 Open Inventor 的 C 和 C++接口的不同。
- 附录 C，**“错误处理”**，描述 Inventor 的错误处理机制。

怎样使用本书

指望读者能从头到尾地去读一本冗长的编程手册是不切实际的。在读完一些基础章节后，根据具体的需要和目标，读者可以略去或者跳过某些章节。这里有一些捷径可以帮助读者阅读本书。

如果想知道怎让创建节点并将它们增加进场景中，读者可以先阅读 1 至 5 章节。然后阅读第 9 章节“**应用动作**”和第 10 章节“**处理事件和选择（Selection）**”。

如果读者主要对如何将文件读入到 Inventor 的数据库中感兴趣的话，可以先阅读 1，2 章节，对 Inventor 有一个大概的了解后，直接跳到第 11 章节“**文件格式**”。

如果读者是一个有经验的 OpenGL 程序员的话，章节 1，2，10 和 17“**在 Inventor 中使用 OpenGL**”将是非常重要的起始章节。此外，为了对建立场景有一个基本的理解，读者应需要阅读 3 至 5 章节和第 9 章节。

章节 15，“**拖拽器和操作器**”和章节 16，“**Inventor 组件库**”描述 Inventor 关于用户接口关联方面的编程。每个类的在线帮助系统都对组件的用户接口进行了描述。

一旦理解了 1 至 5 章节所介绍的基础知识，读者就可以跳到第 13 章节“**引擎**”和章节 14“**节点工具包**”。引擎和其他普通节点一样，也是组成场景的基本元素。它可以驱动场景中的部分物体，并对这些物体添加额外的功能。如果读者想方便快捷地创建场景的话，“节点工具包”也许可以提供一些捷径。

阅读本书前需要具备的知识

本书假定读者熟悉基本的 3D 图形编程概念。例如，本书假定读者对下列术语：灯光、渲染、顶点、多边形、光源、拾取、矩阵、OpenGL、像素、表面法向向量等都有正确的理解。如果这些术语对读者来说是陌生的，建议读者先阅读在位于“**进一步阅读建议**”中的部分书籍。

另外，本书还假定读者熟悉面向对象编程的概念。“**进一步阅读建议**”和附录 A、B 在这方面都提供了很好的背景知识。

本书的约定用法

本书中所有的 Inventor 类、方法、和域的名字都使用黑体字体印刷：**SoNode**，**SoMaterial**，**getValue**（），**setValue**（），**ambientColor**，**Center**。带圆括号的表示方法，例子代码使用 Courier 字体。

提示

有些图标被印刷在页边的空白处，用于高亮显示不同种类的文字。提示带有自己的标题标记。相应的图标显示在右边。

高级内容

有些高级内容在初次阅读时，读者可以跳过这些内容。高级内容使用标题(*高级内容*)来

标记。这个标题可以放在文章的单段上、整个节上、或章节上。

关键的场景图符号表

图 1-1 显示了本书中出现的所有场景图标中所使用的符号图标

进一步阅读建议（参考文献）

对于计算机图形学入门而言，请阅读下列书籍：

- Foley, J.D., A. van Dam, S. Feiner, and J.F. Hughes, *Computer Graphics Principles and Practice,* 2e. Reading, Mass.: Addison-Wesley, 1990.
- Neider, Jackie, Tom Davis, Mason Woo, *OpenGL Programming Guide*. Reading, Mass.: Addison-Wesley, 1993.
- Newman, W., and R. Sproull, *Principles of Interactive Computer Graphics*, 2e. New York: McGraw-Hill, 1979.

对于 C++语言入门而言，请阅读下列书籍：

- Lippman, Stanley B., A C++ *Primer*, 2e. Reading, Mass.:Addison-Wesley, 1991.
- Shapiro, Jonathan, A C++ *Toolkit*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1991.

对于面向对象编程入门而言，请阅读下列书籍：

- Meyer, Bertrand, *Object-Oriented Software Construction*. London: Prentice Hall International, 1988.

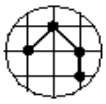
图 1-1 场景图符号

Transform

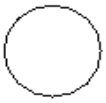
Light

Appearance

Manipulator

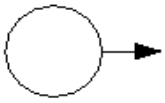

Metric/Topology

Node Kit


Property (Misc.)

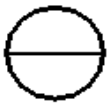
SoSelection

Shape


Callback

Group

Path


Separator

Render Area/Component

Engine

Subgraph

Switch

Field

Camera

realTime Global Field

致谢

作为一个学习并使用 Open Inventor 的研究者，在这里我要深深地感谢那些给予我极大帮助的原 Inventor 开发组成员，Rikk Carey, Gavin Bell, Alain Dumesny, Dave Immel, Paul Isaacs, Howard Look, David Mott, Paul Strauss, 和 Helga Thorvaldsdóttir。他们为这本书无私奉献了大量的精力和时间，丰富了本书的内容。即使在开发最终期限到来之际和残酷竞争的压力下，Inventor 开发组的成员们仍然一如既往地满足我的需求：回答所有的问题，评论无数的草稿，并且编写了本书的主要例子代码。Rikk Carey 为 Open Inventor 项目贡献了其非凡的领导才能。Paul Strauss，是 Inventor 的主要架构师之一。他审阅了绝大多数的草稿，并用红(绿)墨水对这些草稿进行了详细说明。他带有深刻见解的评论对我来说是无价之宝。另外包括 Ronen Barzel, Sam Chen, Beth Fryer, 和 Kevin Goldsmith.等人都对本书提出了见解独到的评论。

如果将所有的难题放在一起：屏幕拷贝，插图（Line Art），颜色搭配，文字，例子代码等等 — 所有这一切都需要花费巨大的人工。Rikk Carey 设计了封面图案，Catherine Madonia 使用 Rikk 的场景为每部分每章节的标题头设计了黑白插图。Paul Isaacs 展示了他的设计天赋 — 不是说他在恐怖的最终期限到来之前的工作能力 — 为彩图 1 至 21 设计了许多模型和场景。我也要非常感谢 Maria Mortati，他设计了彩图的章节并且说服我让图表（figure captions）更容易编辑。Catherine Madonia 帮助我设计了概念上的图案和例子程序的屏幕图像拷贝。太多的与卖主打交道的事情：例如，调整打印机、行程安排、生产印刷书籍等等这些工作都被 LauraCooper 和 Lorrie Williams 高效的处理完了。Kay Maitz，CheriBrownb,Dan Youndg，HowardLook 和 LorrieWilliams 绘制了插图。Hellga Thorvaldsdottir 在例子代码上花费了大量精力：编写许多的例子代码，并在最后对它们进行了修正和润色。

非常感谢普林斯顿大学交互计算机图形实验室的 Kirk Alexander，他提供的许多图片和修订令我们非常满意。为了使用户有产品使用前的最初体验（Out-of-Box Experience），SGI 工作站设计的一个应用程序展示了彩图 37 至 40 。它们是由 Silicon Graphics 的 LeoBlume，Mark Daly，Kevin Goldsmith, Howard Look, and Chee Yu 和 Colossal Pictures 的 Brad de Graf, Shari Glusker, Jill Huchital, Karen Hughes, Peter Oppenheimer, Kark Swain ,Drew Takahashi, and Phil Zucco 设计的。第 30 个彩图“艺术画廊漫游”是 Silicon Graphics 的 Gavin Bell, Mark Daly,Kevin Goldsmith,和 Linda Roy 的工作成果。在封面场景中的许多模型都是由 Acuris，Inc 公司创建的。

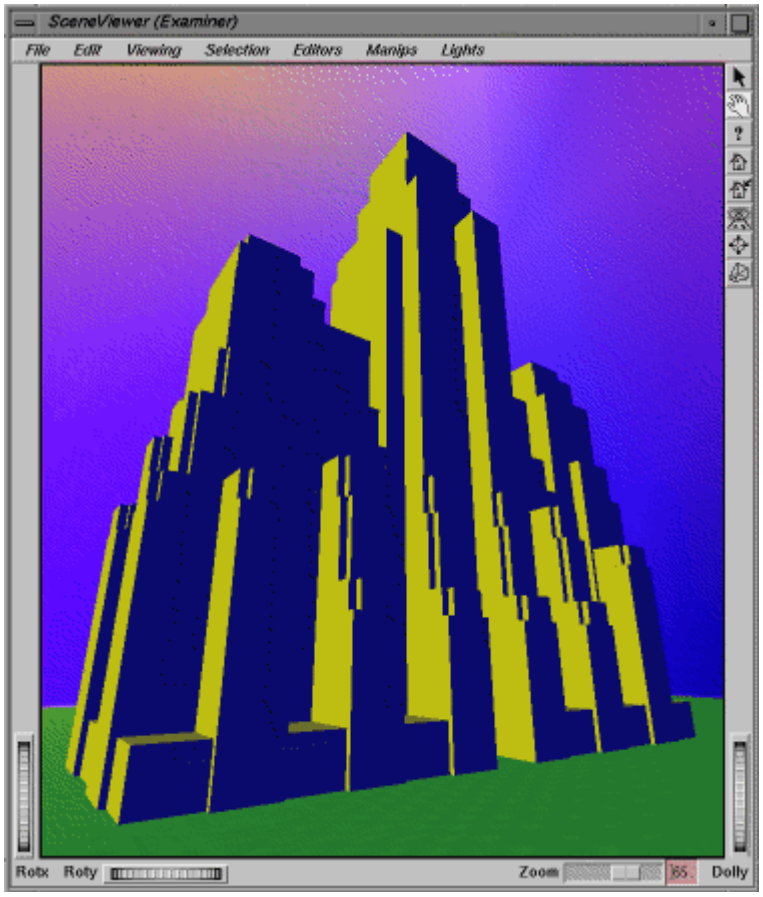
我还要感谢 Silicon Graphics 公司的 Visual Magic Division 项目经理 Jackie Neider 和我所在部门的成员－Patricia Creek,Arthur Evans,Liz Deeth,Beth Fryer,Jed Hartman, Ken Jones, John Stearns, Eleanor Bassler,和 Carol Geary。在项目的整个过程中他们给予我以坚强的支持和鼓励（包括向我分享高速打印机）。

同时，最重要的是我要感谢我的丈夫 Steve 和我的儿子 Jeff 和 Evan。感谢他们为我精心制作了丰盛的晚餐，以及在漫漫长夜中的促膝长谈。

彩图

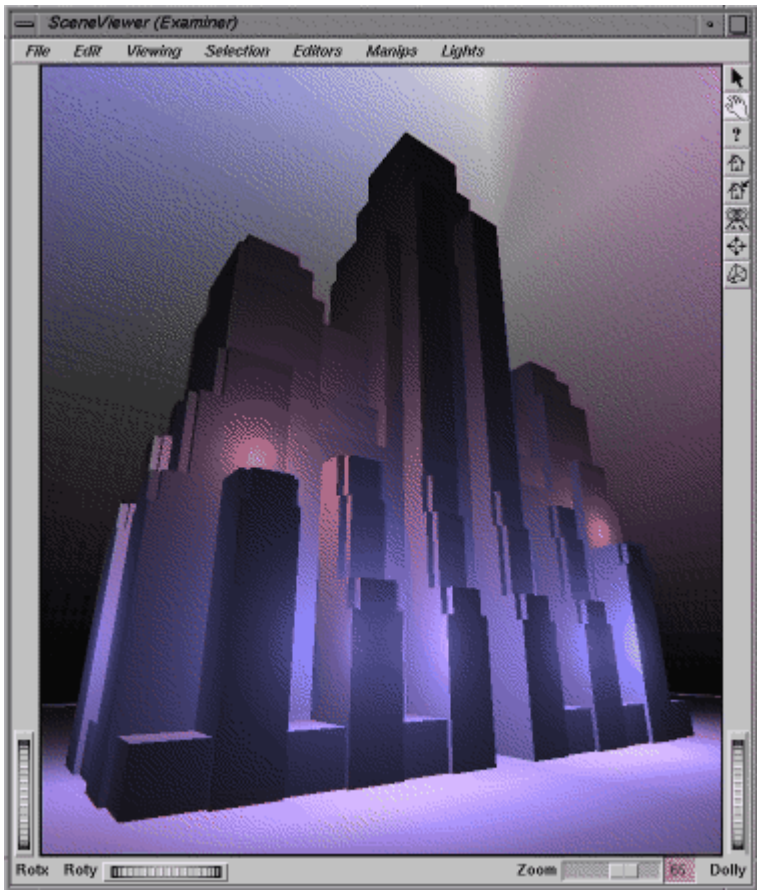
本节包含了本书中引用的所有彩图。

彩图 1



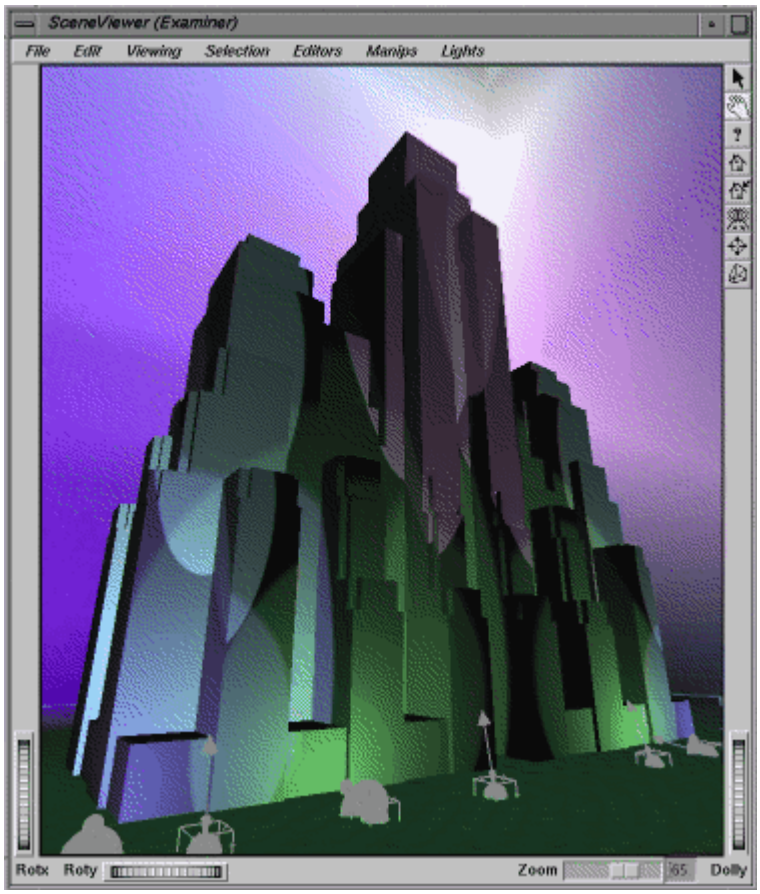
这个场景展示了射灯的效果。场景中包括了篮色和黄色的射灯；建筑本身是白色的(彩图 1 至 21 是由 Paul Isaacs 提供的)

彩图 2



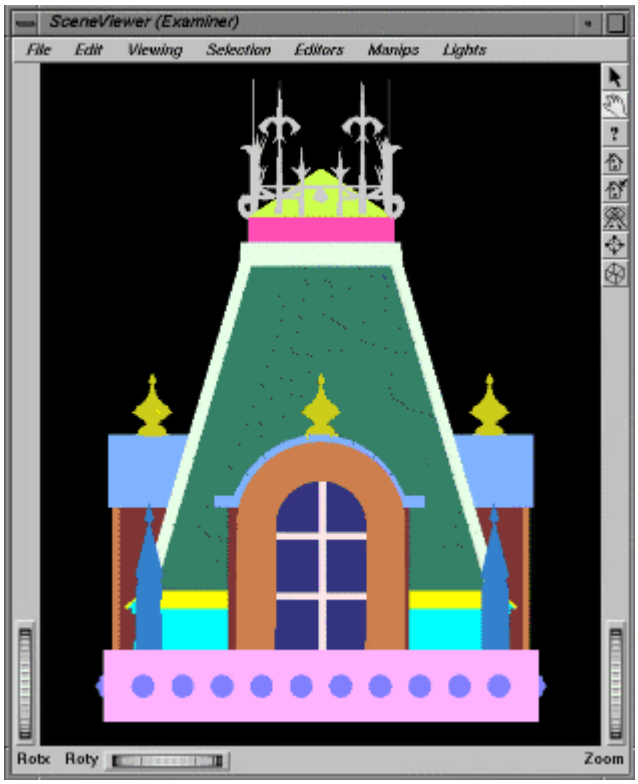
相同的场景，使用了紫色的点射灯。

彩图 3



相同的场景，使用聚光灯展示了舞台效果；图中灰色物体代表聚光灯的位置和方向。见第 4 章。

彩图 4

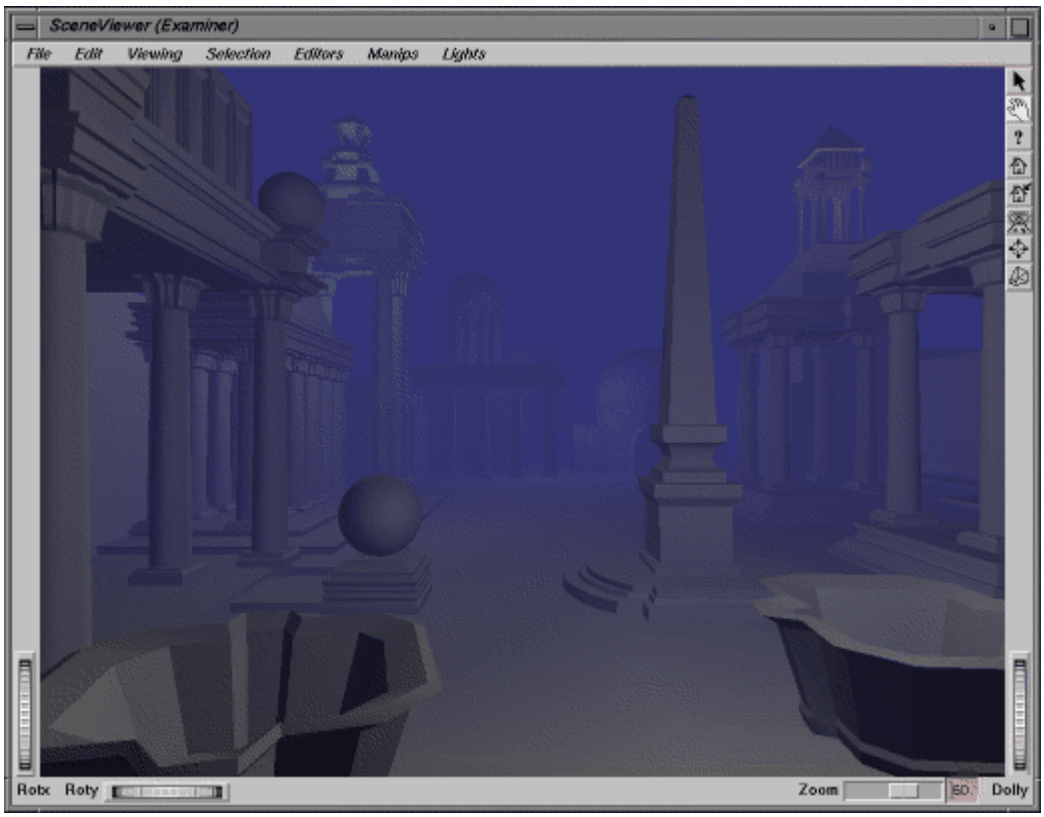


使用 BASE_COLOR 类型的光照模型来渲染场景。

彩图 5

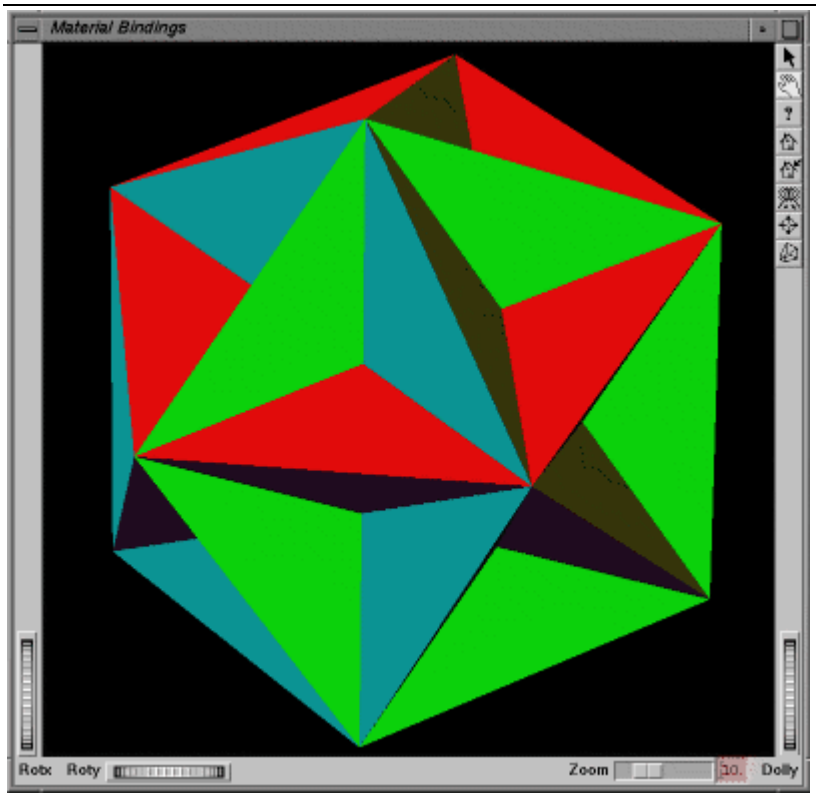
相同的场景，使用 PHONG 方式的光照模型渲染场景。见第 5 章。

彩图 6



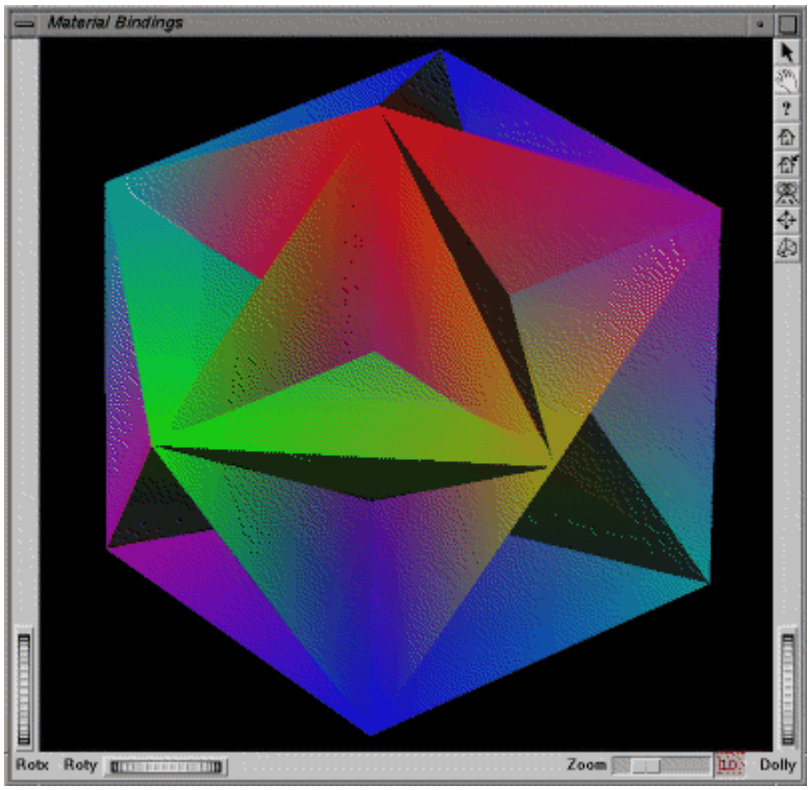
展示了 SoEnvironment 节点的效果。雾的类型是 FOG，颜色是淡紫色。见第 5 章。

彩图 7



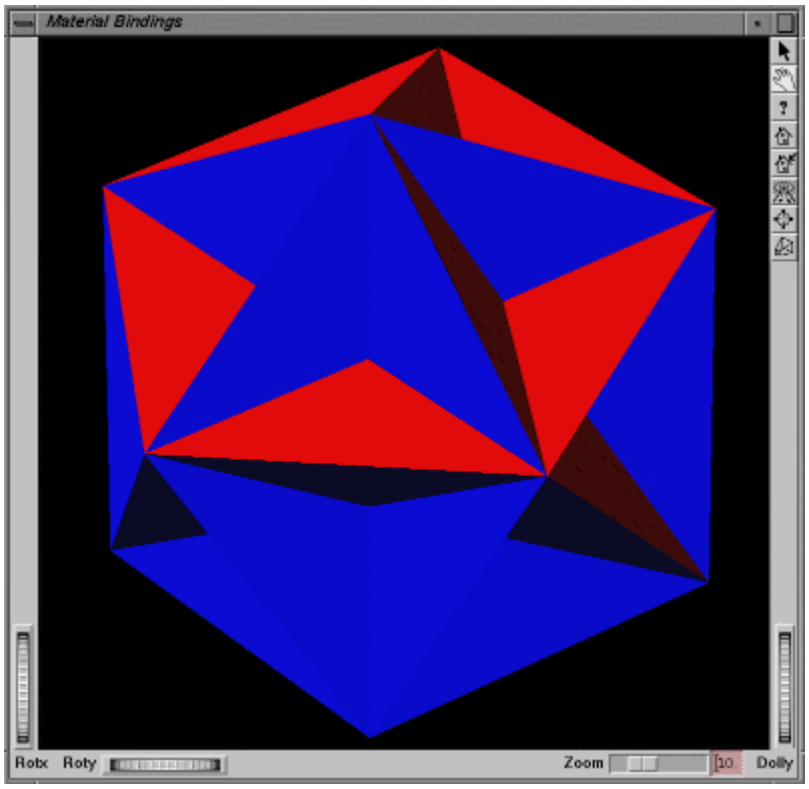
显示一个各面材质绑定(per-face material binding)的索引面集（Indexed Face Set）。

彩图 8



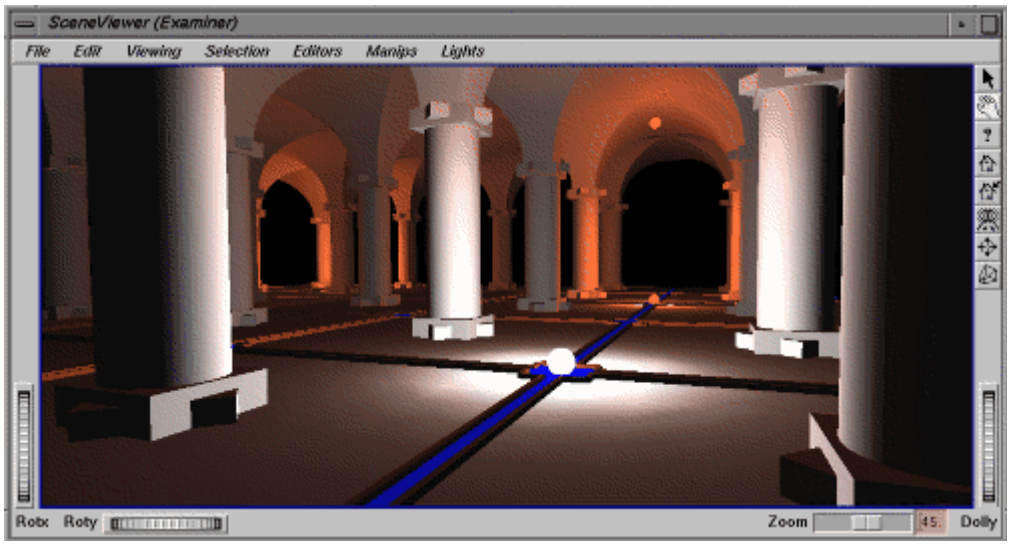
显示一个各顶点索引材质绑定(per-vertex-indexed material binding.)的索引面集。

彩图 9



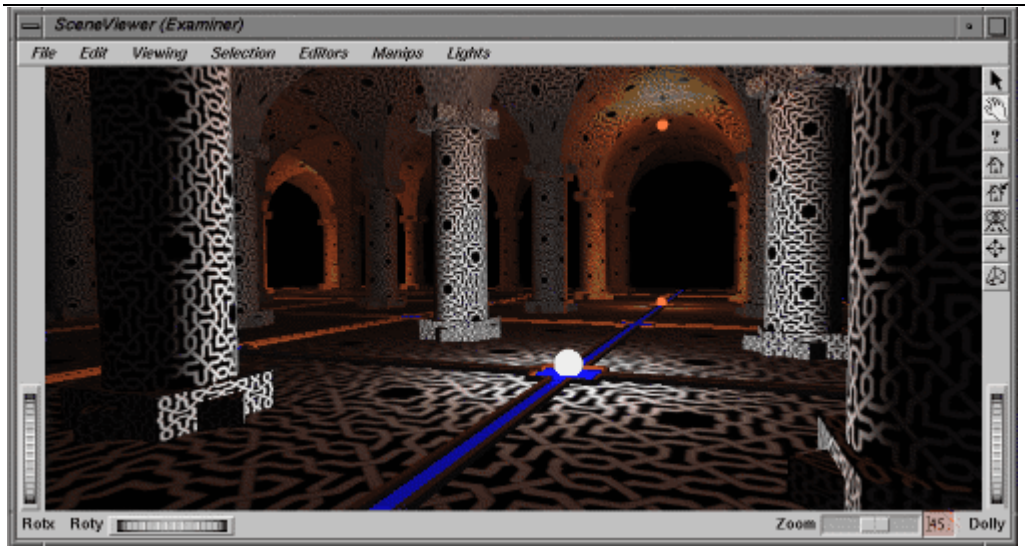
相同的面，但使用各面索引材质绑定(per-face-indexed material binding)。见第 5 章。

彩图 10



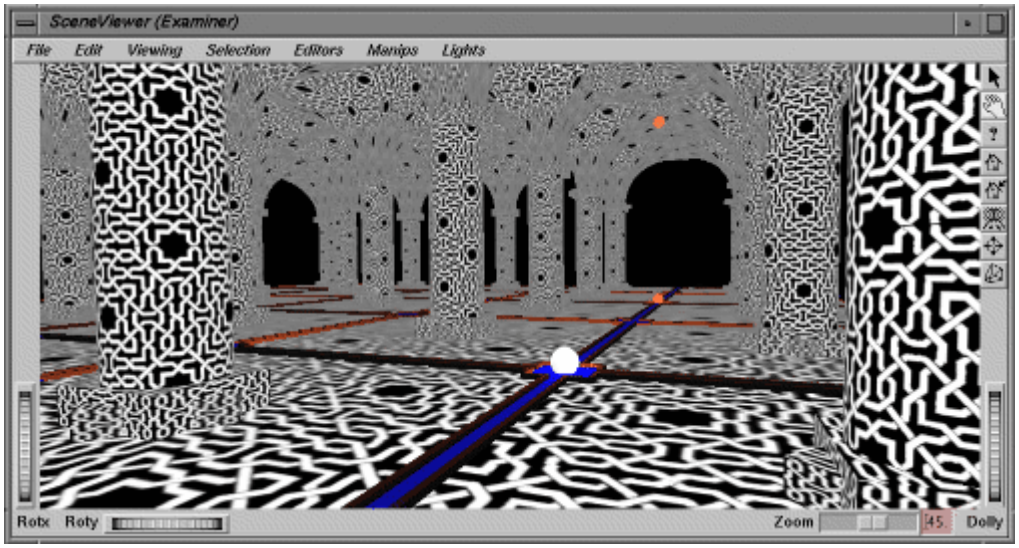
场景使用了无纹理的表面，前景有一个白色点光源，背景有一个橙色点光源。

彩图 11



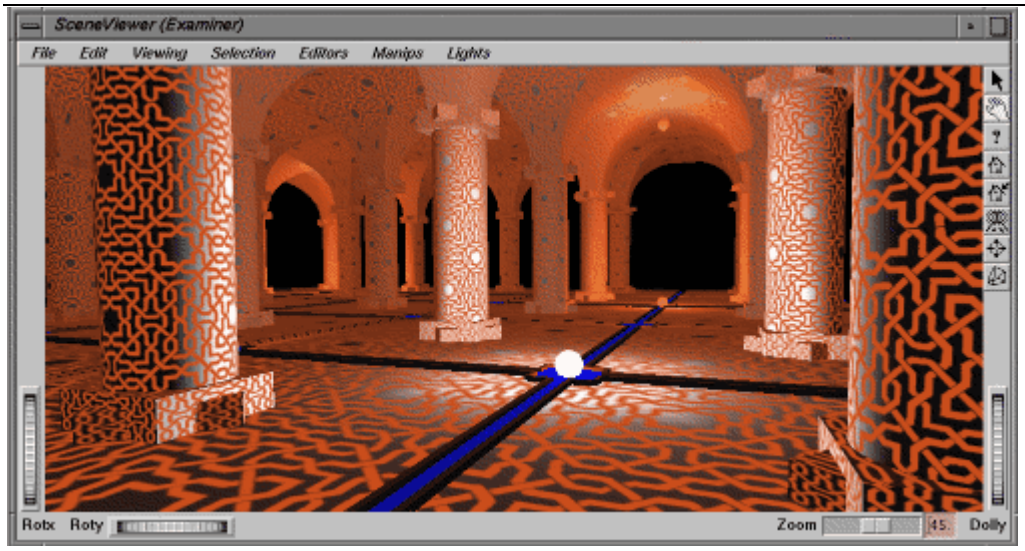
相同的场景，使用 MODULATE 纹理模型。

彩图 12



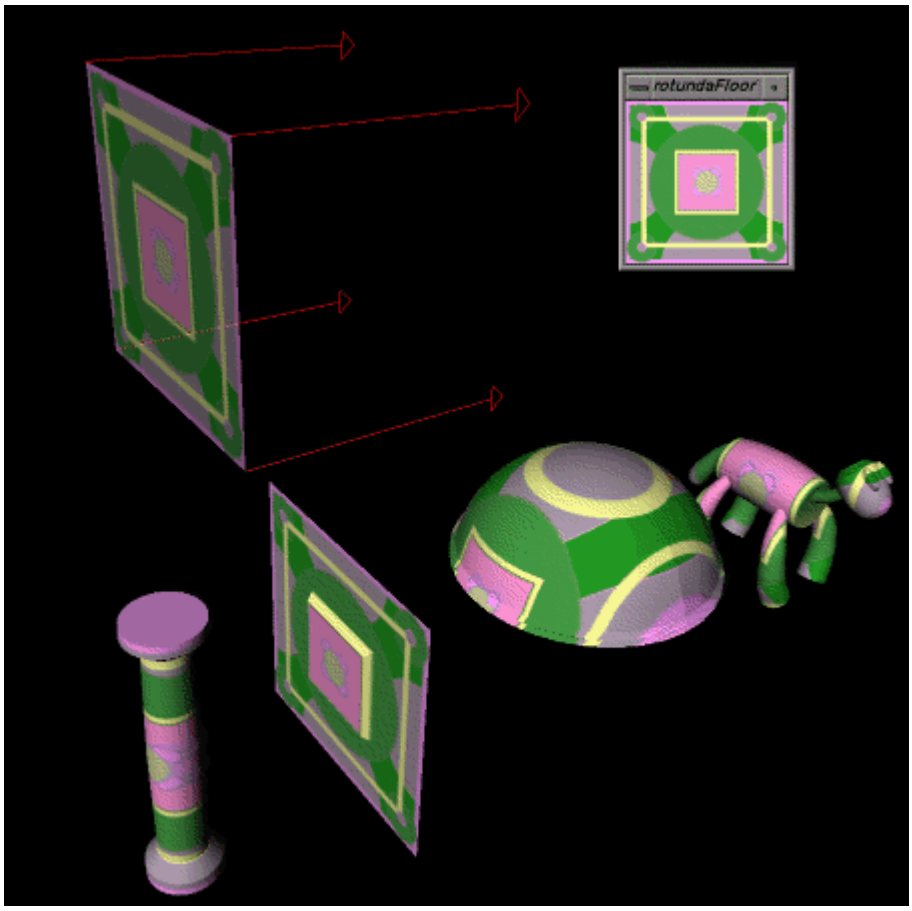
相同的场景，使用 DECAL 纹理模型。

彩图 13



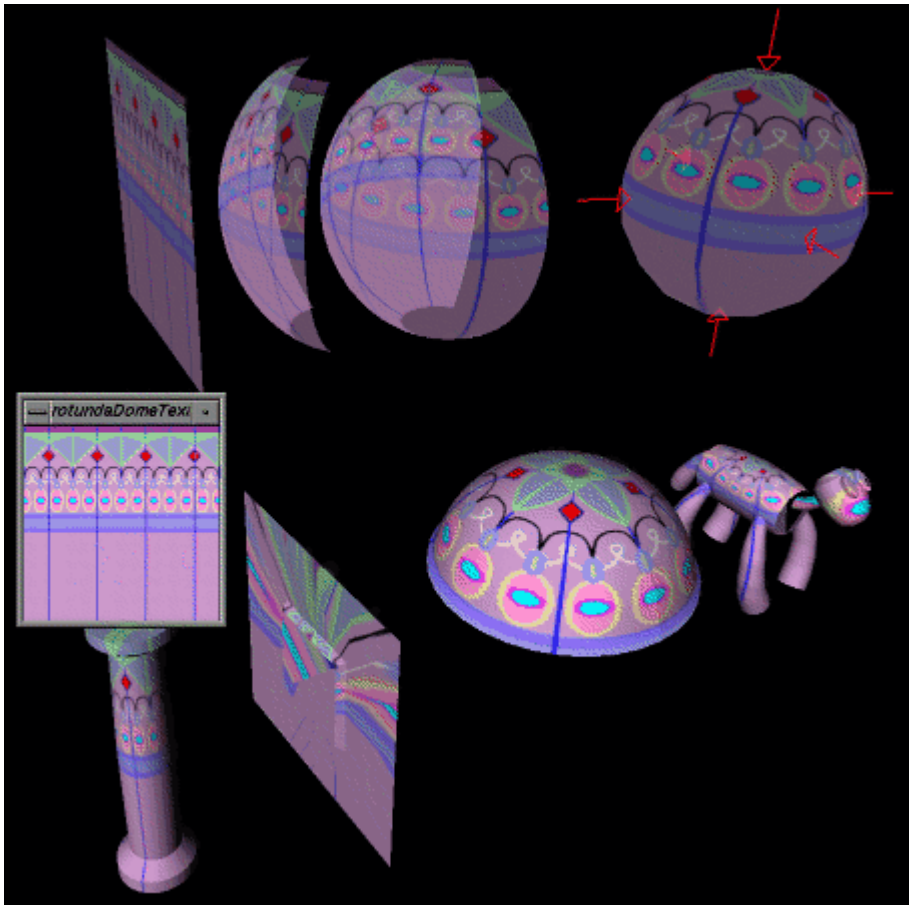
相同的场景，使用具有混金色的 BLEND 纹理模型，带有金壁辉煌的效果。见第 7 章

彩图 14



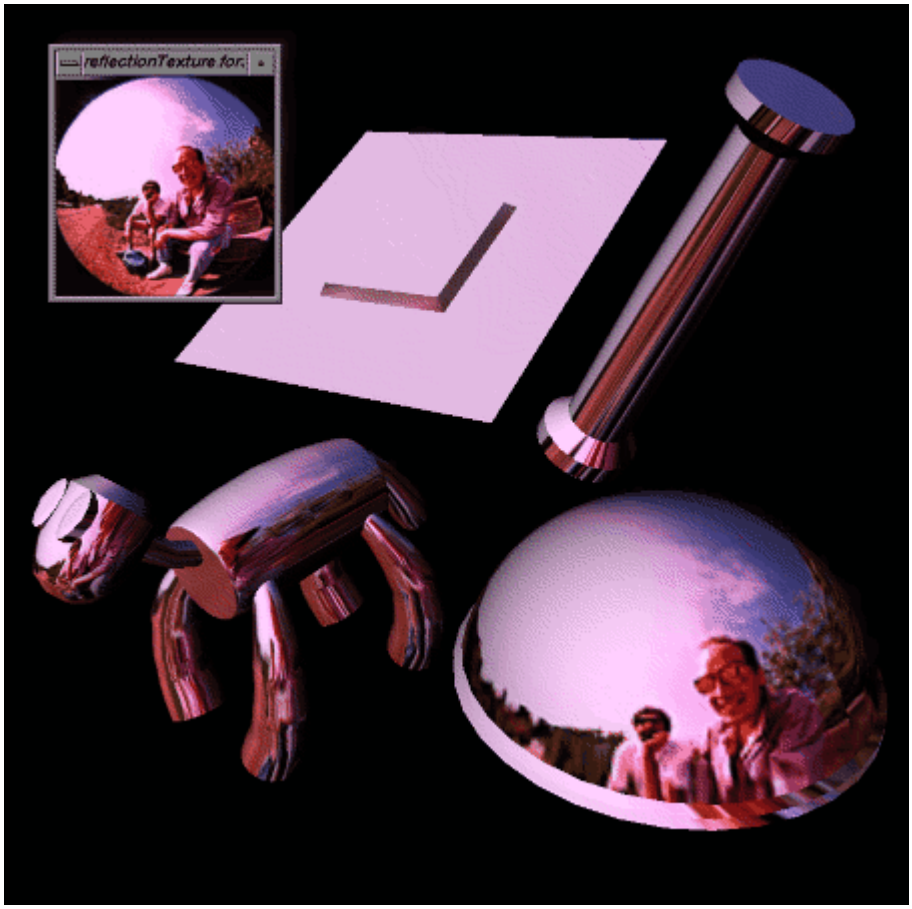
使用 SoTextureCoordinatePlane 进行纹理映射。左边(带有箭头)的方形纹理投影到 4 个不同的物体上。

彩图 15



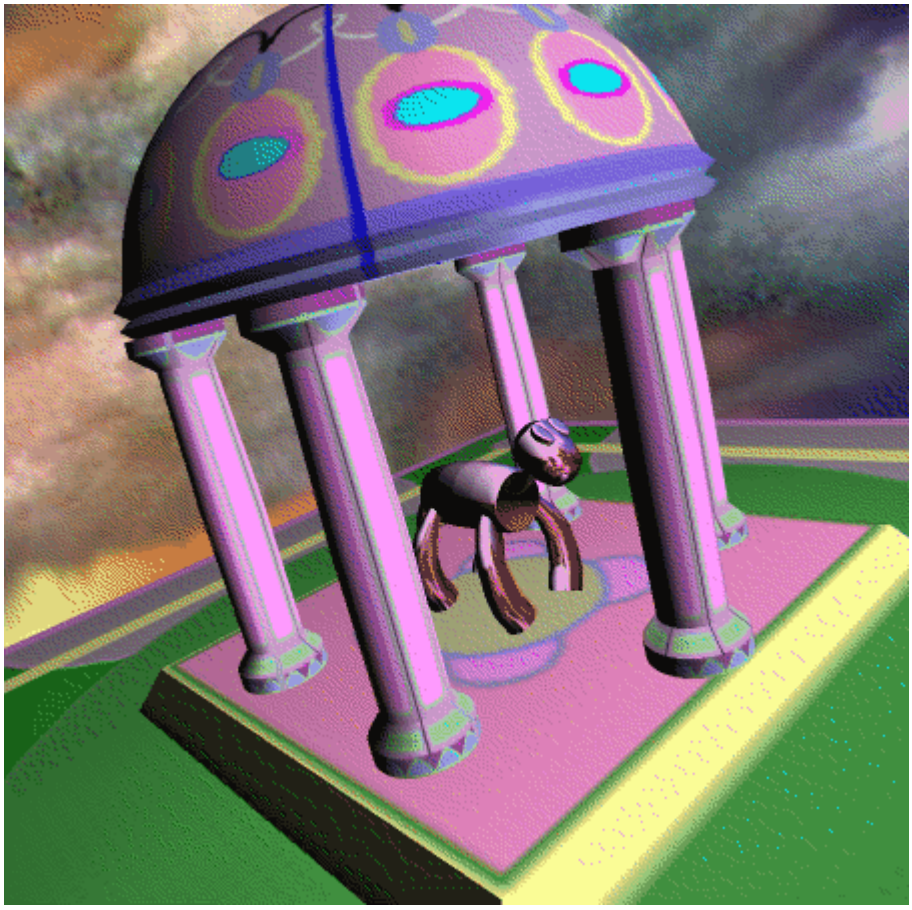
使用 SoTextureCoordinateSphere 进行纹理映射。右边(带有箭头)的球形纹理投影到 4 个不同的物体上。

彩图 16



使用 SoTextureCoordinateEnvironment 进行纹理映射。

彩图 17



场景同时使用多种纹理坐标函数节点。见第 7 章

彩图 18

场景使用 BLEND(混合)的透明方式。渲染的顺序是背景、球体、锥体、黑色和黄色建筑。

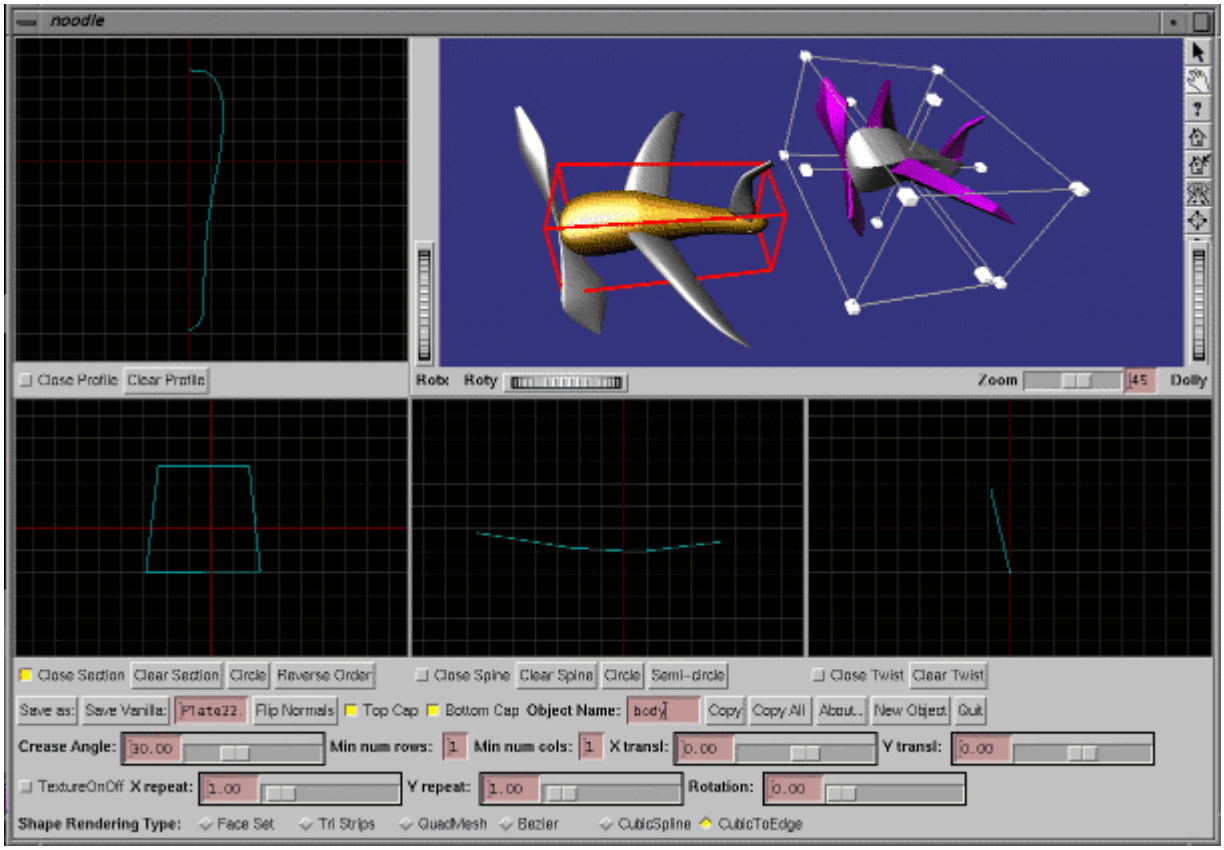
彩图 19

场景使用 DELAYED_BLEND 的透明方式。渲染的顺序是不透明物体(背景、黑色和黄色建筑)，然后透明物体(球体，锥体)。

彩图 20

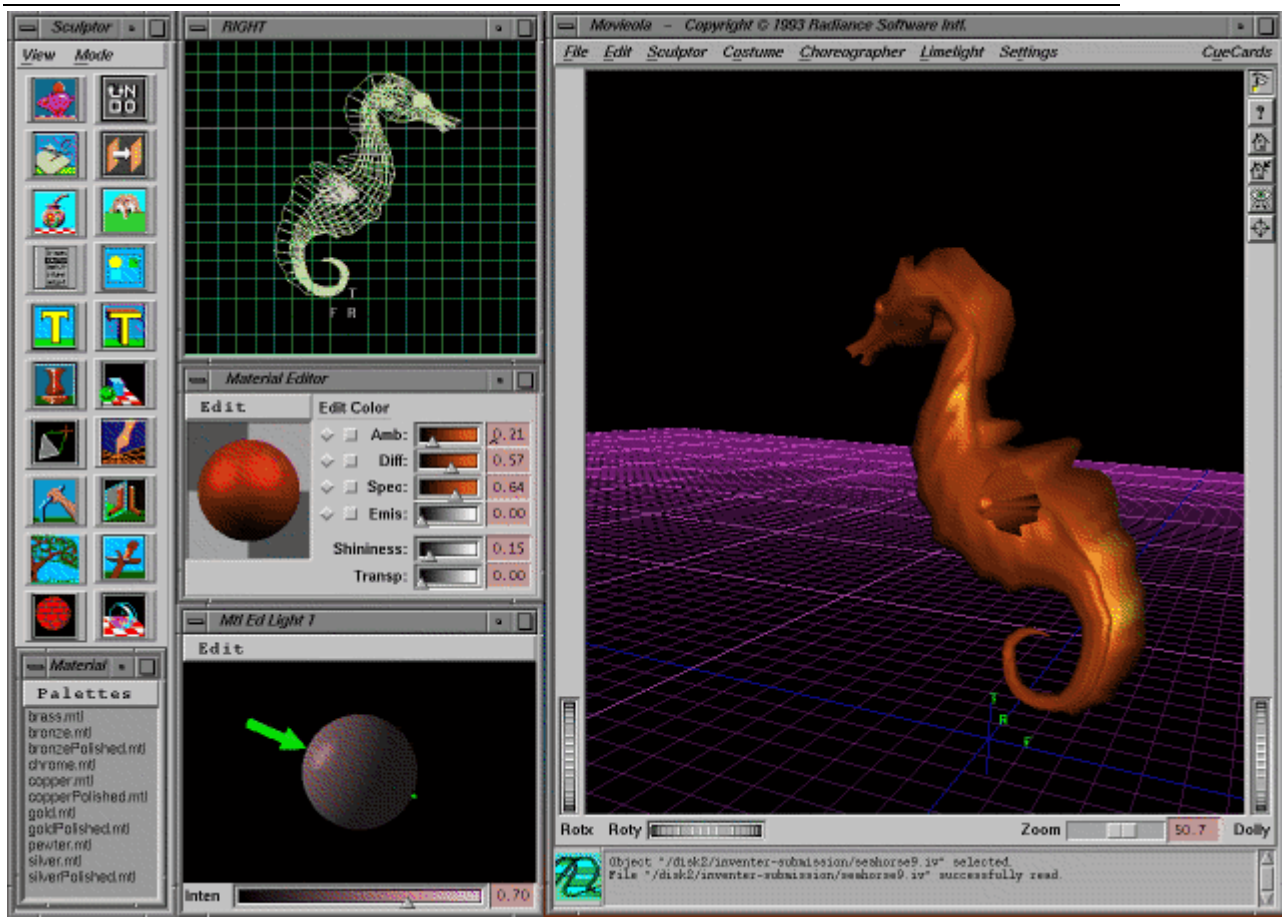
场景使用 SORTED_OBJECT_BLEND 的透明方式。渲染的顺序先是不透明物体(背景、黑色和黄色建筑)，然后是按照从后到前的顺序排列的透明物体(球体、锥体)。见第 9 章

彩图 21



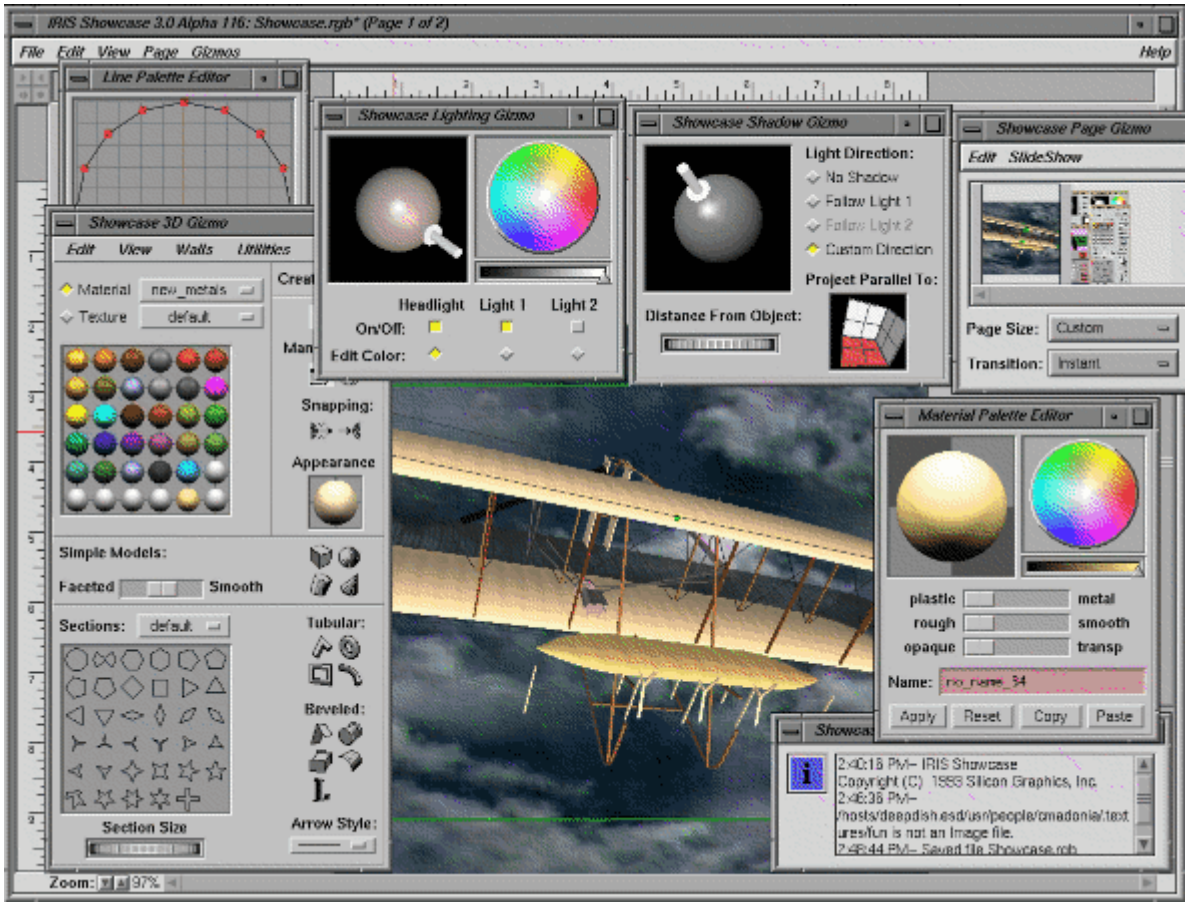
Noodle,是一个建模工具程序。用户可以对任意物体进行截面(cross-section)，样条化(spline)，剖面(profile)，和扭曲等操作。金色的飞机是使用 Inventor 的 NURBS 曲面制作的。紫色的飞机也是使用和金色的飞机同样的数据制作的。

彩图 22



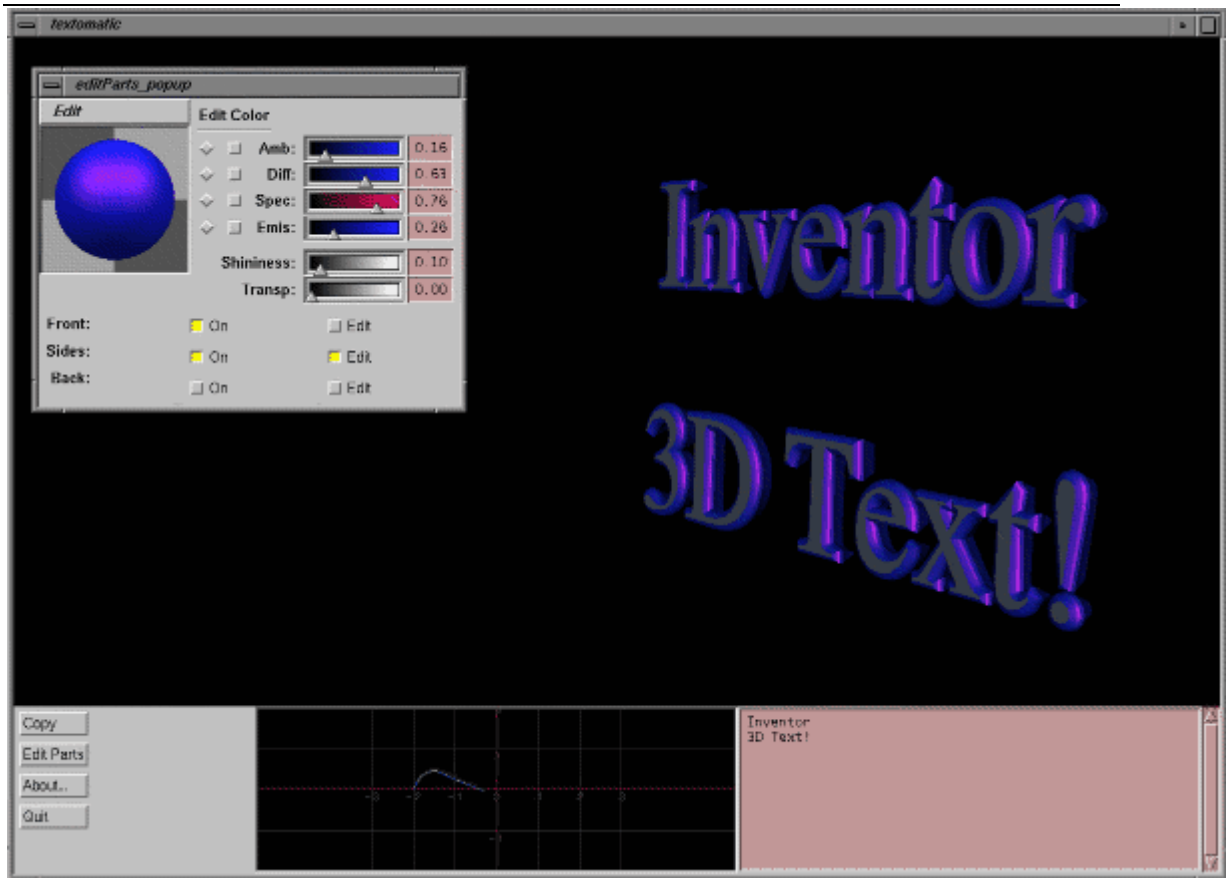
Movieola®，是一个建立在 Inventor 基础上的模型动画制作系统。这个应用程序大量采用了 Inventor 的操作器、组件、节点包(Node Kits)。这为它提供了创建复杂物体的直观工具.。(© 1993 Radiance Software International)

彩图 23



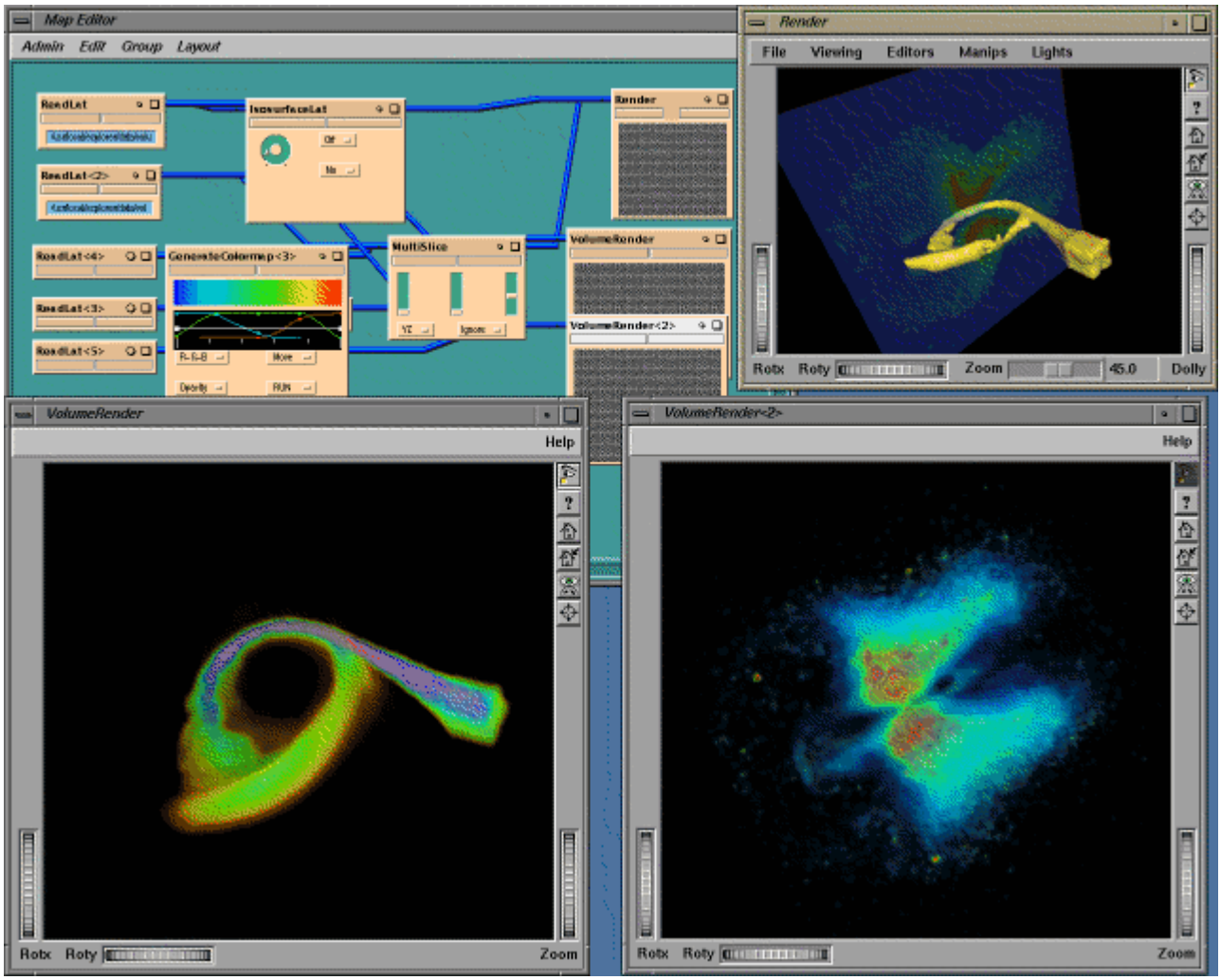
Showcase，这个程序用于创建多媒体平台。所有的 3D 支持，包括渲染、交互、编辑等都是由 Inventor 提供的。(Rikk Carey 提供此图)

彩图 24



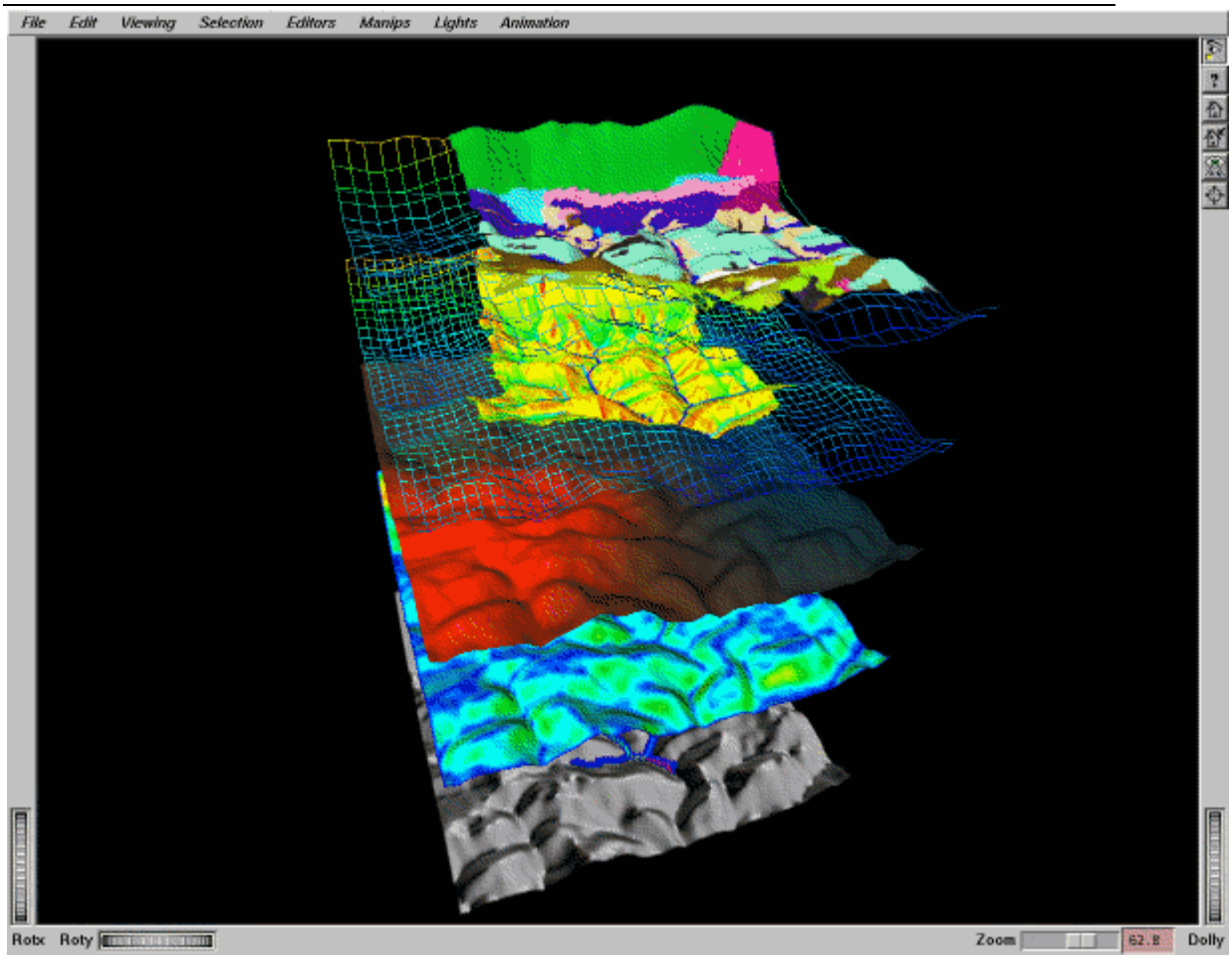
Textomatic，一个简单的 Inventor 应用程序，用于交互式地定义 3D 字体的轮廓和材质。(Catherine Madonia 提供此图)。

彩图 25



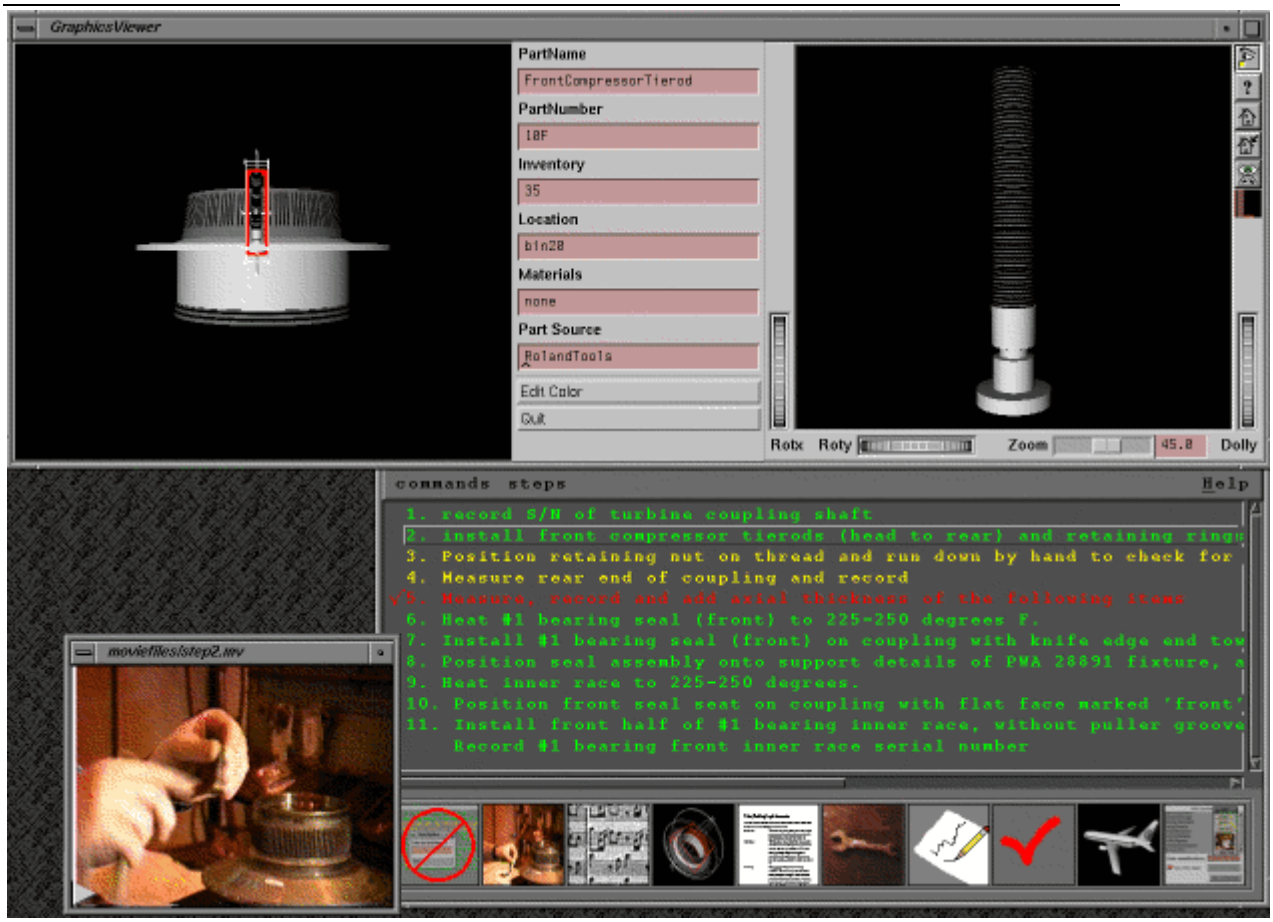
流体力学仿真程序。这个程序用于数据的可视化显示。Inventor 被用于 3D 渲染。 (Data courtesy of Drew Whitehouse and Gustav Meglicki, Australian National University Supercomputer Facility. Image by Roy Hashimoto)

彩图 26



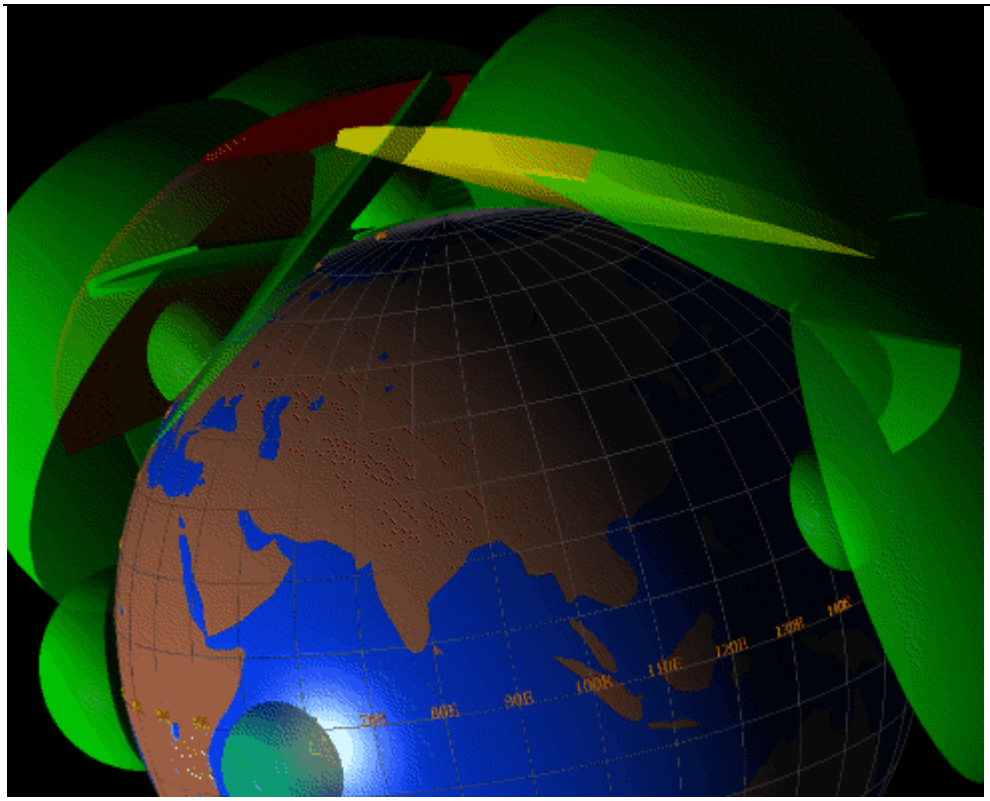
这五层地图显示了美国农业部(USDA)提供的在宾夕法尼亚州克里克地区的一个分水岭的地理情况。这些地图分别代表悬崖的方向、斜坡的方向、降雨量、土壤饱和程度、土壤种类等。 (Data courtesy of Dominique Thongs, Department of Civil Engineering; image courtesy of Interactive Computer Graphics Laboratory, Princeton University)

彩图 27



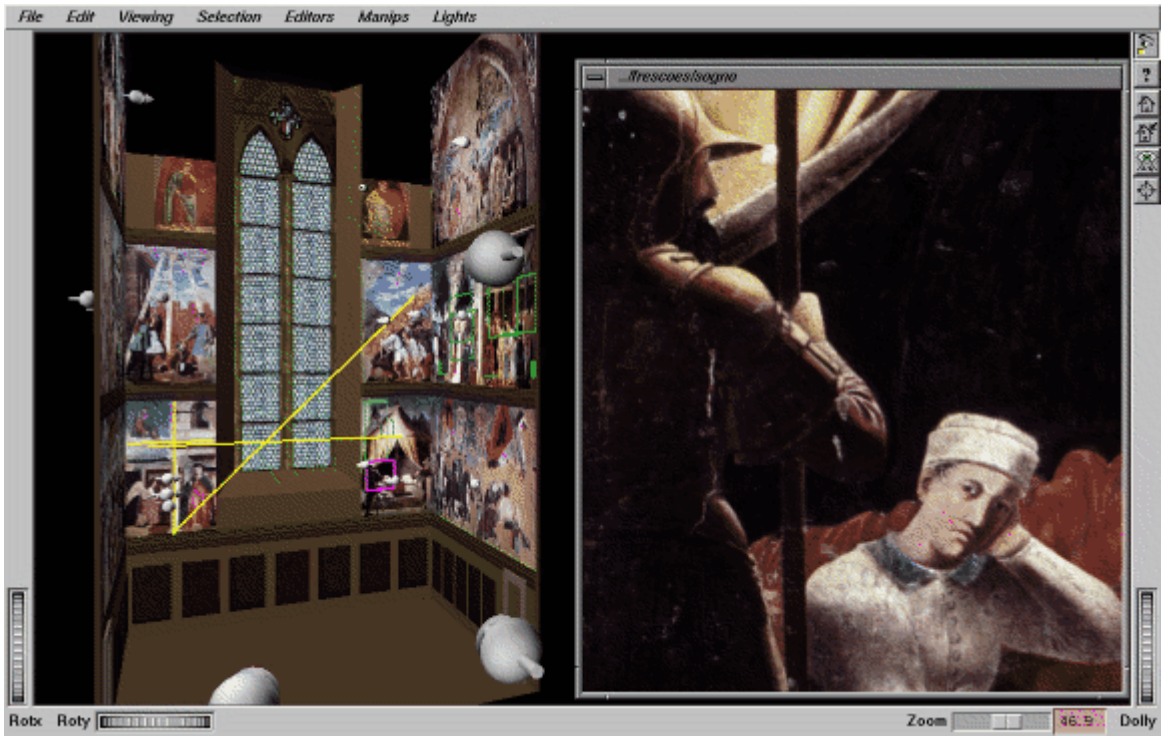
一个使用多媒体方式显示的维修手册和产品目录数据库。用户可以移动工件，可以在工件选择某些部分进行放大显示，可以看演示录像，听解释录音。 (Kevin Goldsmith 提供此图)

彩图 28



在 3D 图形中显示雷达覆盖范围和状态的应用程序。使用了两个新的几何体类：一个象圆屋顶那样的几何体代表可移动雷达天线。另外一个象馅饼那样的几何体代表了定相阵列雷达。(Decision-Science Applications, Inc 提供此图)

彩图 29



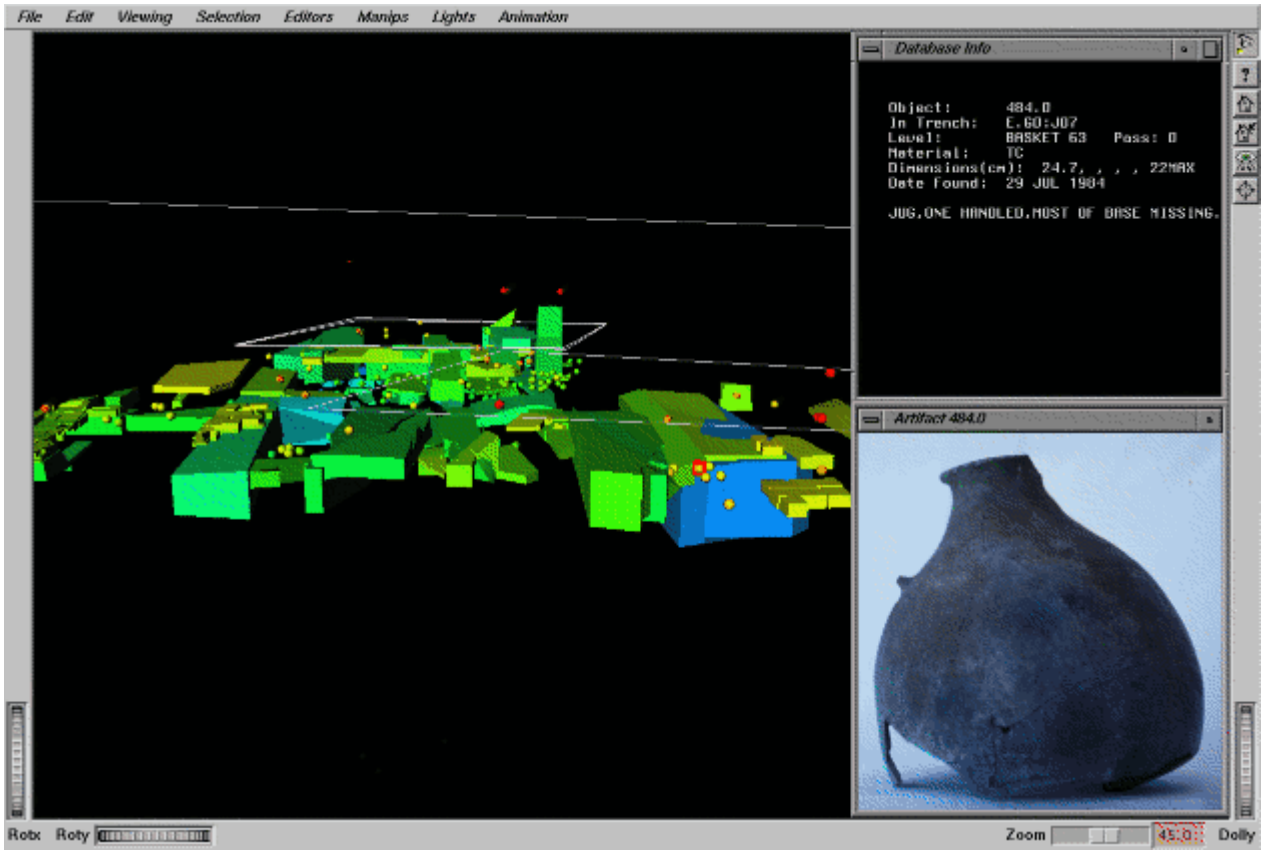
The Piero Project 是一个利用交互式 3D 计算机图形技术来讲授艺术史的应用程序。这张

图像向我们展示了在意大利小镇 Arezzo 的圣弗朗西斯卡大教堂的重建原貌。点击粉红色线框盒可以产生如右边显示的高精度图像。点击白色的球体可以允许用户以球体所在的位置处观察整个场景。使用这些白球也可以产生这个教堂的漫游动画录像. (Image courtesy of Interactive Computer Graphics Laboratory, Princeton University, and The Piero Project by Kirk Alexander, Kevin Perry, and Marilyn Aronberg Lavin.)

彩图 30

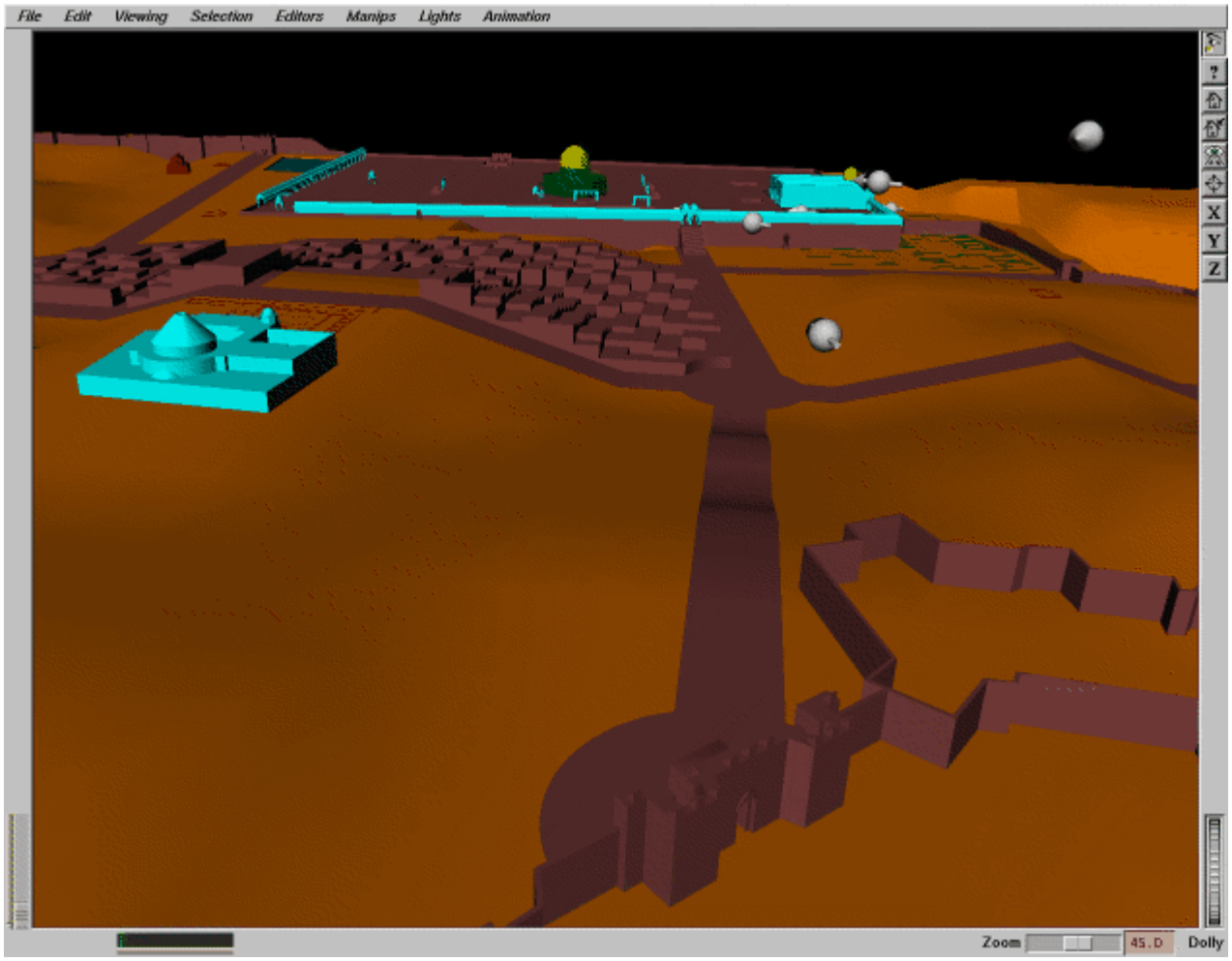
一个可以交互式地漫游艺术画廊的程序。(Gavin Bell and Kevin Goldsmith 提供此图)

彩图 31



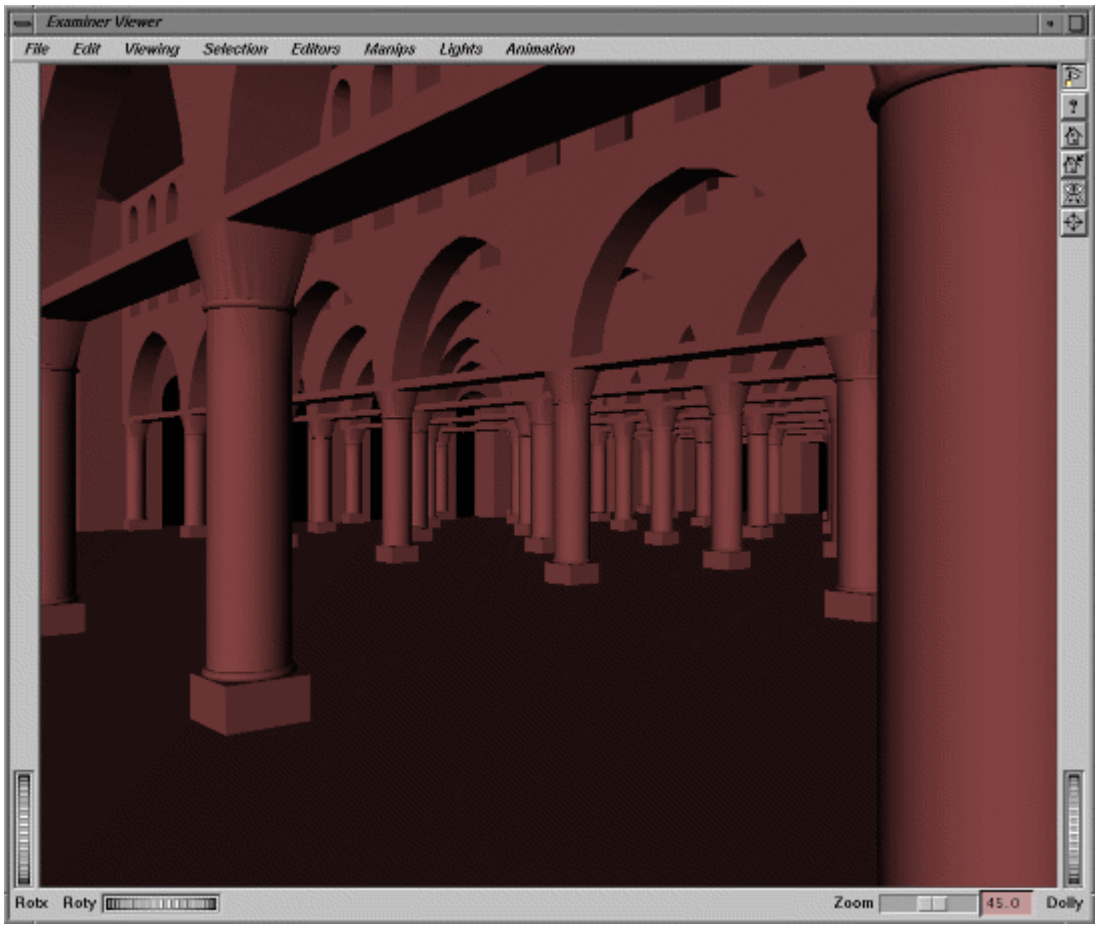
Trenchmaster 是一个用于考古学上的应用程序。在主窗口中包含物体代表着这在一个考古挖掘现场发现的史前文物。点击其中的一个物体将可以产生一个带有解释文字和文物图片的窗口。(Data courtesy of William Childs, Dept. of Art and Archaeology; image courtesy of Interactive Computer Graphics Laboratory, Princeton Univ.)

彩图 32



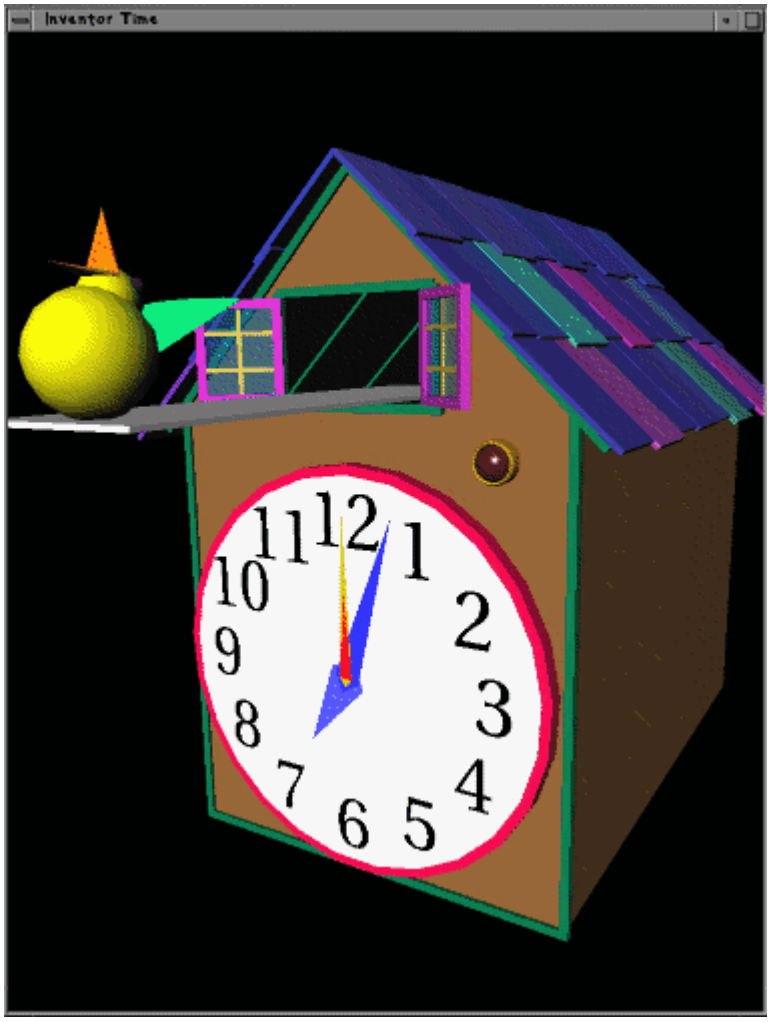
一个用于重现早期伊斯兰圣地耶路撒冷的程序。用户可以漫游城市街道并游览城市建筑。(Data for Plates 32 and 33 courtesy of Mohamed Alasad and Oleg Grabar, Institute for Advanced Study, Princeton, New Jersey.)

彩图 33



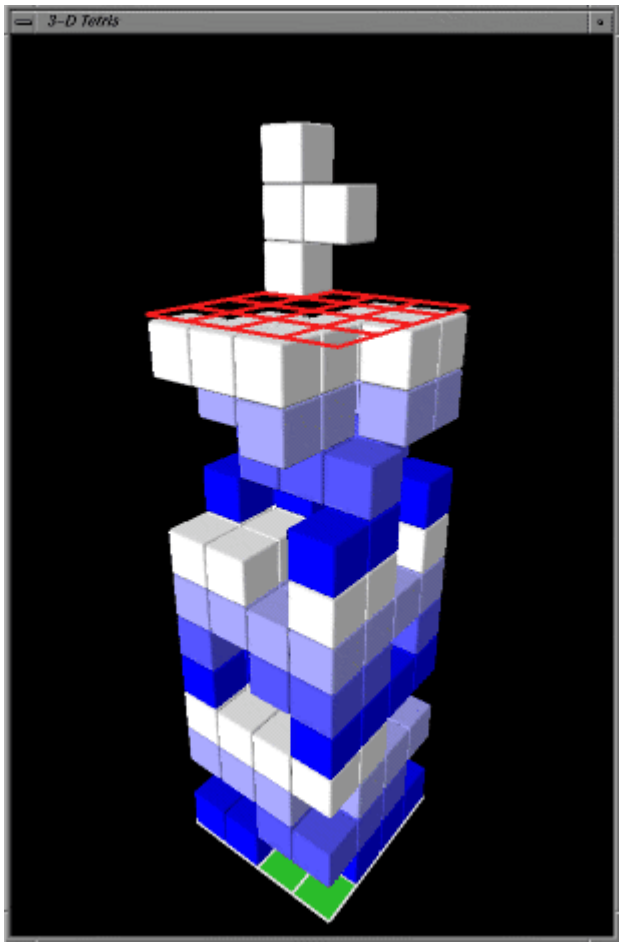
在内部观察清真寺的结构，其中的一个建筑曾在彩图 32 中出现过。(Images for Plates 32 and 33 courtesy of Interactive Computer Graphics Laboratory, Princeton Univ)

彩图 34



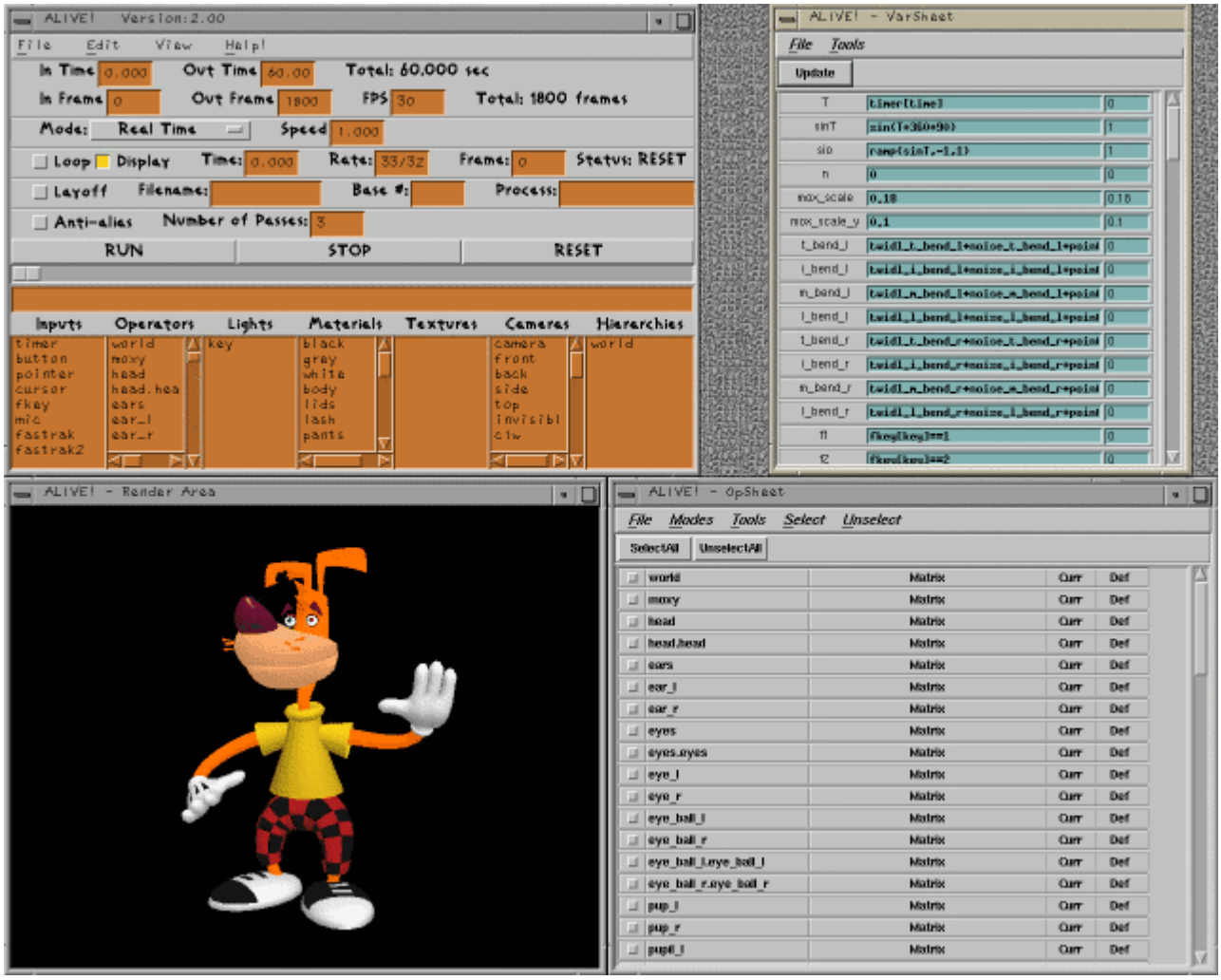
布谷鸟钟。使用一个与实时全局域(real-time global field)相连接的引擎对象来驱动时钟指针。(Rikk Carey 提供此图)

彩图 35



俄罗斯方块。使用时间传感器(timer sensor)来驱动落下的方块。(David Immel 提供此图)

彩图 36



Moxy Moto 是一个用 Alive 程序产生的计算机角色。角色的实时动画程序是基于 Inventor 之上的。(Moxy was created by (Colossal) Pictures and by the Cartoon Network. Alive is a product of deGraf/Associates.)

彩图 37

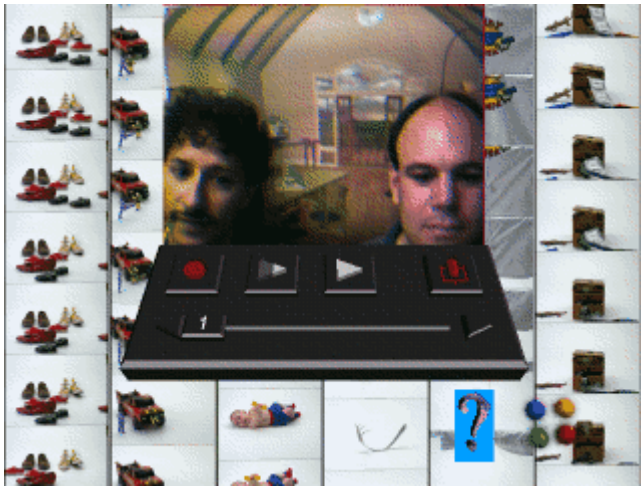
产品使用前的最初体验（Out-of-Box Experience）的场景。这是一个隐藏式的多媒体平

台。这个程序向 Inventor 的工具包中增加了一个新的多媒体节点。这个节点可以和 Inventor 的其它对象进行同步。如果点中了其中移动的一个圆球将可以给用户增加一个新的重玩机会。

彩图 38

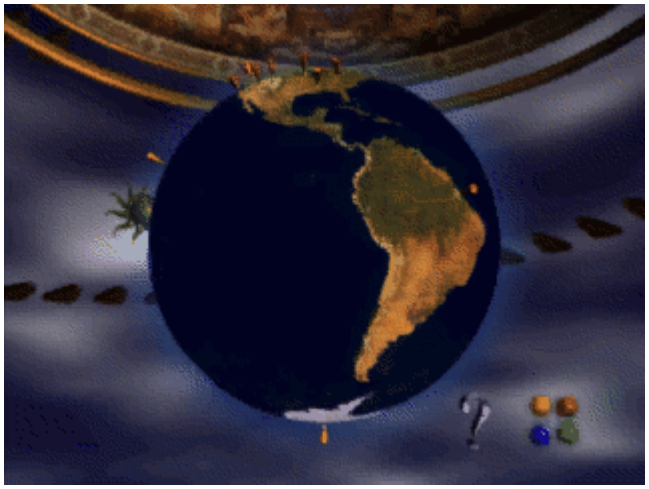
一个动画屋。每一个动画对象都被一个不可见的 Inventor 对象环绕着。当用户点击其中一个对象时，就会表演一段动画录像。

彩图 39



在动作停格室(Stop-Motion Room)里的一个实时录像窗口。控制面版和背景都是用 Inventor 对象建立的。

彩图 40



地球室(Earth Room),用户可以旋转地球。当用户点击球上一点时，将会演示一段这个地区的地理录像或者录音的片断。(Images by Kevin Goldsmith and (Colossal) Pictures)

第一章 综述

本章目标

在阅读完本章节后，我们将可以做下列事情：

- 认识组成 Open Inventor 工具包的几个关键组件。
- 理解 Open Inventor 与 OpenGL 的关系。
- 描述扩展 Open Inventor 的几种方法。

本章将概要地阐述组成 Open Inventor 的几个关键组件，并将简要描述如何使用 Open Inventor 来完成一些特殊的需求。本章还将解释 Inventor 与我们可能已经很熟悉的编程工具，例如 OpenGL 和 X Window System，之间的关系。本章提及大多数的主题都将在以后的章节中作详细解释。

什么是 Open Inventor ？

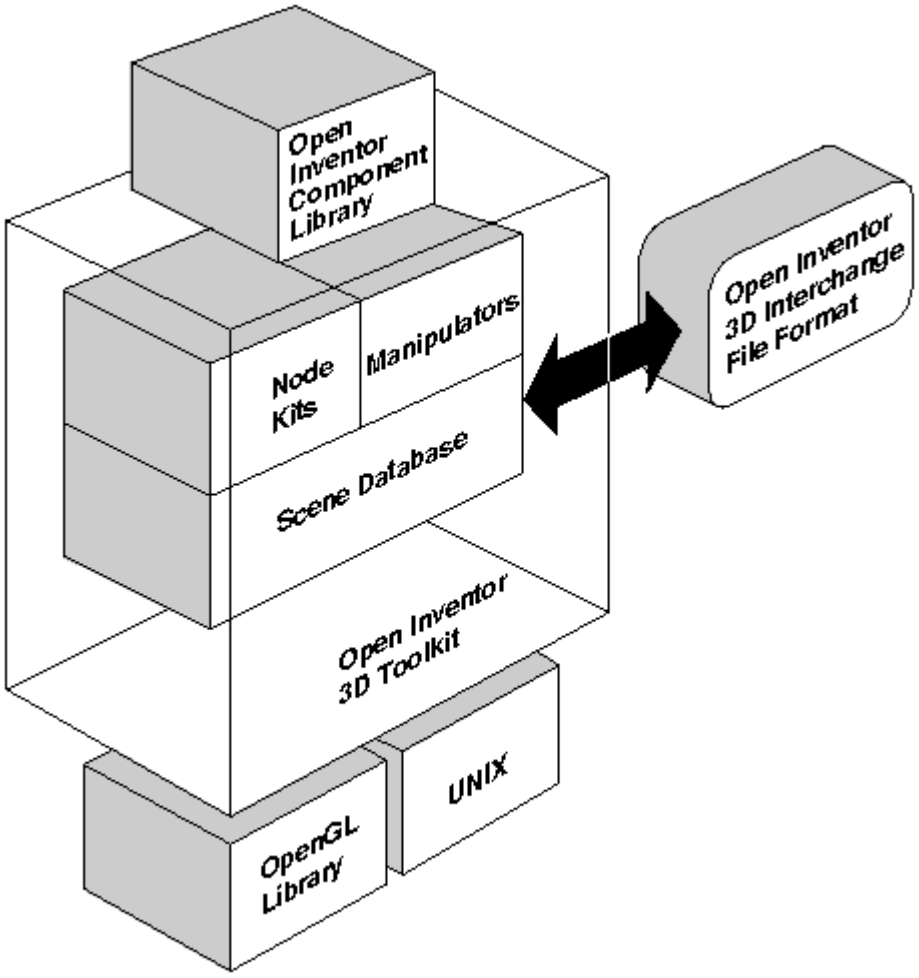
The Inventor Mentor 这本书主要是面向图形编程人员和应用程序开发人员，讲解面向对象的 3D 开发包－Open Inventor。Open Inventor 是一个用于开发交互式 3D 程序的图形库。虽然它本身是用 C++编写的，但 Open Inventor 也可以在 C 语言中使用。

Open Inventor 是由一系列的对象模块组成的，利用这些对象模块，开发人员可以花费最小的编程代价，开发出能充分利用强大的图形硬件特性的程序。Open Inventor 是一个建立在 OpenGL 基础上的对象库，开发人员可以任意使用、修改和扩展对象库。Inventor 对象包括：数据库图元（*database primitives*）、形体、属性、组、和引擎等对象；还有例如像手柄盒（Handle Box）和轨迹球等*操作器*、材质编辑器、方向灯光编辑器、examiner 观察器等组件。

Inventor 提供了一个完整且经济高效的面向对象系统。除了能简化应用程序开发之外，Inventor 还可以通过内建的 3D 交换文件格式，在应用程序之间进行数据交换。操作 3D 应用程序的用户可以通过剪切、粘贴场景中的 3D 对象，与桌面上的其它程序共享这些对象数据。

如图 1-1 所示，Inventor 是建立在 OpenGL 和 UNIX 共同基础上的。Inventor 提供了一个建立在 OpenGL 基础上的面向对象的应用程序策略(*application policy*)。同时 Inventor 还为 OpenGL 程序提供了一个编程模型和用户接口。

图 1-1 Inventor 体系结构



Inventor 具有操作系统无关性。构件库可以对特定的操作系统使用 Inventor 提供帮助（**构件库就是指图 1-1 中 Open Inventor Component Library 模块，这个模块向下屏蔽具体的操作系统细节，向上为用户提供统一的用户界面。例如针对 Microsoft Windows 操作系统，该模块提供了 SoWinXXX 等接口。对 Xt Windows 系统，提供了 SoXtXXX 等接口。译者注**）。本书将讨论 Inventor 工具包提供的一个构件库。这个构件库可以帮助用户在 Inventor 中使用 Xt 窗口系统的窗口和事件功能。在本书的姊妹书 *The Inventor Toolmaker* 中，详细介绍了如何将 Inventor 扩展到其它窗口操作系统上。

只要对象,不用绘制 (Objects,not Drawings)

Inventor 的注意力主要放在如何创建 3D 对象上。所有关于这些对象的信息－形状、大小、颜色、表面纹理、空间位置等－都将被保存在场景数据库中。这些信息可被用于多个方面。最主要的用途是在屏幕上显示或渲染(Render)一个 3D 物体。

对于很多其它的 3D 图形开发包来说，它们最终的目标可能仅仅是在屏幕上将 3D 场景显示出来。但是，如果用户想移动场景中的某个物体，或者想从一个稍微不同的视角观察某个物体，如果用户希望对场景内的物体或者背景色设置不同的颜色，如果一个化学家想重新

对两个分子进行排列，如果一个飞机设计师要重新设计飞机机翼的曲线轮廓。所有这一切，如果场景只是作为屏幕上的一幅图画而存在的话，那么开发人员可能就需要编写非常复杂的代码才能实现上述功能。另外，还需要增加上驱动场景中物体的代码。如果使用 Open Inventor，其编程模型已经针对这些变化需求内部做了相应的处理。修改场景中的物体，向场景增加物体，让物体之间可以相互作用，等等这些都将变成一个非常简单的过程。Inventor 之所以有这些能力，是因为这些修改需求是 Inventor 明确定义的接口的一部分，同时也是 Inventor 设计的基本预定目标。

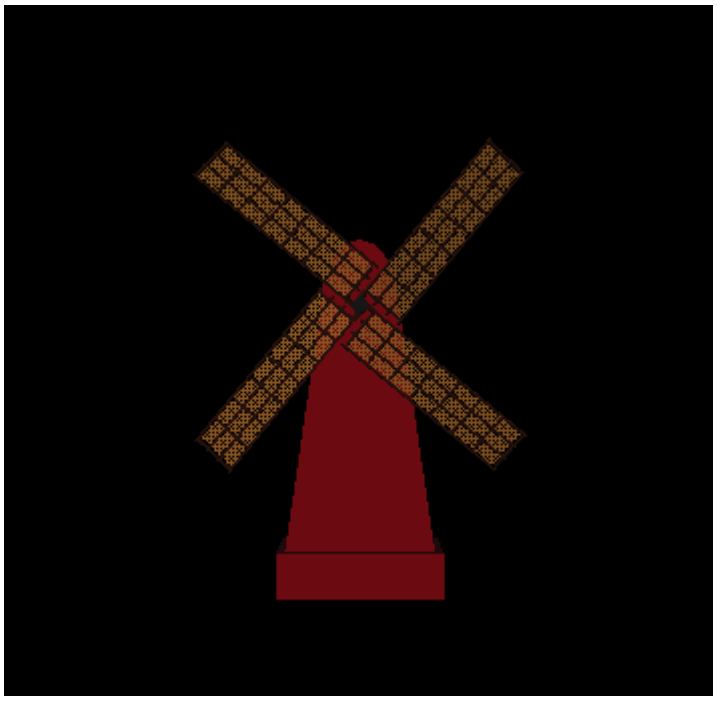
多种方式使用数据对象

因为 Inventor 数据库只保存物体自身 3D 空间内的信息，而不保存显示在屏幕上的 2D 像素数组，所以除了可以渲染对象外，还可以对物体执行其它的操作。场景中的物体可以被拾取、高亮显示、作为单独实体被操作、计算包围盒(Bounding-box)、打印和查找、从文件中读取对象数据或将对象写进文件。上述这些内建的功能为开发人员提供了一个灵活而又强大的开发平台。同时这种编程方式也很直观，因为它是建立在我们所生活世界中的物理和机械法则基础之上的。

动画

Inventor 的对象还可以将动作信息保存到场景数据库中（**即场景数据库中不仅包含描述物体属性的静态信息，还可以包含有描述物体动作的动态信息。例如下面的例子中，数据库里包含了如何控制风车旋转的信息，译者注**）。例子 1-1 是从一个 Inventor 文件里摘录出来的，它描述了一座扇叶按照一定速率旋转的风车。当 Inventor 读入此数据文件后，屏幕上将显示出风车图像，并且风车扇叶还在做旋转运动。注意：在 Inventor 程序中不需要添加任何代码就可以让扇叶旋转起来，因为这个旋转动作的描述信息是风车对象本身的一部分（**即风车对象中包括了扇叶是按照何种方式旋转的参数数据，译者注**）。图 1-2 显示了这个风车的图像.

图 1-2 风车



例 1-1 描述一个旋转的风车的文件

```
#Inventor V2.0 ascii

Separator {
   Separator {
      RotationXYZ {
         axis Z
         angle 0 =
            ElapsedTime { # Engine to rotate blades
               speed 0.4
            }
            . timeOut      # Engine output connected to
                           # angle of rotation
      }
      Transform {
         translation 0 0 0.5
      }
      Separator {# Shaft for blades
         Material {
            diffuseColor 0.05 0.05 0.05
         }
         Transform {
            rotation 1 0 0 1.5708
            scaleFactor 0.2 0.5 0.2
         }
```

```
        Cylinder {
        }
     }
     DEF Blade Separator {        # Blade geometry and properties
        Transform {      # Blade interior
           translation 0.45 2.9 0.2
           rotation 0 1 0 0.3
        }
        Separator {
           Transform {
              scaleFactor 0.6 2.5 0.02
           }
           Material {
              diffuseColor 0.5 0.3 0.1
              transparency 0.3
           }
           Cube {
           }
        }
        Separator {               # Blade frame
           # .... (Details omitted)
        }
    }
    Separator {          # Second blade
       RotationXYZ {
          axis Z
          angle 1.5708
       }
       USE Blade
     }
     Separator {         # Third blade
        RotationXYZ {
           axis Z
           angle 3.14159
       }
       USE Blade
     }
     Separator {         # Fourth blade
        RotationXYZ {
           axis Z
           angle -1.5708
        }
        USE Blade
```

```
      }
   }
   Separator {   # Windmill tower
        # ... (Details omitted)
   }
}
```

Open Inventor 与 OpenGL 的关系

如果读者熟悉 OpenGL 的话,可能会很想知道 OpenGL 与 Open Inventor 之间的关系是什么。本节将概要讲述这两个库之间的相互关系。第 17 章将提供另外详细的内容来阐述如何在程序中同时使用 Open Inventor 和 OpenGL。一 既使用 OpenGL 灵活高效的 3D 渲染功能,同时也可使用 Inventor 的高级对象和基础场景数据库。

Open Inventor 使用 OpenGL 作为渲染引擎。然而在 OpenGL 中，渲染命令是必须显式地写在代码中。而在 Inventor 中，渲染以及像拾取、读写文件、计算包围盒等这些操作都被封装到对象自身中了。

OpenGL 提供直接模式（immediate-mode）访问帧缓冲区(frame buffer)，它也可以用显示列表来记录绘图命令，并在需要时回放这个显示列表。

Open Inventor 不提供直接模式来访问帧缓冲区。正如在前面章节“**只要对象,不用绘制**”中所阐述的那样，Open Inventor 是建立在面向对象编程模型基础上的，这种模型可以在数据库中创建高级，可编辑的对象。每个对象都封装了一系列像渲染、拾取、数据查询搜索和包围盒计算等操作。在 Inventor 中，渲染帧缓冲区是发生在渲染操作被调用的时候。如果 Inventor 程序从来就没有发出这些命令(直接或间接的)，那么屏幕上将不会产生任何变化。

一个简单的类比可以帮助我们理解 Open Inventor 和 OpenGL 的差别。假设现在是 2020 年，我们有足够的时间、金钱、和所必须的技术来造一座自己的房子。我们可以从下列两种建造方式中选择一个，当然也可以同时结合使用这两种方式。

方式一，直接去建筑材料市场，分头购买所需的材料—钉子、木头、管线、铁丝、开关、混凝土等等。这种方式非常灵活，但是它需要我们有相关的知识和技能来决定那些材料是必需的。并且还需要知道如何使用这些材料来建造房子。

方式二，从 *Dream Home Catalog* 家装杂志中定购一些成品建筑构件。这本杂志是由一家公司在 10 年前出版发行的。这家公司借鉴了日本家庭建筑的概念，是 90 年代非常成功的成品办公建筑公司。这本杂志中提供了大量如墙结构单元(wall-frame units)、混凝土模板、滑轨、窗户、门等产品。

第一种方式—从未加工的原始材料开始－这类似于使用 OpenGL 写交互式图形应用程序。使用这种方式建造房屋，我们可以任意使用这些未加工的材料。但前提是我们需要非常

熟悉建造房屋的所有细节，同时在建造房屋不同部分时也需要不同的技能－管道工、电工、木工等等。

第二个方式—从杂志中选择成品构件—这很像使用 Inventor 开发应用程序。墙板已经预先铺设好了电线和连接管道。这个预制件可以和 Inventor 提供的内建事件模型相类比。另外关于所有材料的数目、尺寸和费用的详细数据都已经在我们下订单的时候自动计算好了。类似的，所有的操作（渲染、拾取、包围计算等等）都也内建于 Inventor 的物体对象中了。不需要附加代码（相当于建造房子的时候，进行附加的计算）就可以获得这些信息。因为提供这些产品的公司常年在建材市场大量购买建筑材料，所以他们可以准确地知道，怎样用最经济实惠的方式购买材料，确定材料数量。同样地，Open Inventor 也可以从对 OpenGL 的使用上获得更高的性能。

虽然杂志提供了一系列的成品构件，但我们仍然可以自由决定使用那些产品，以及该如何将它们组合起来。无论何时从家装杂志中购买成品构件，它们的尺寸都是标准的，都可以很方便地替换、修理、更新屋子里的其他部分。因为 Inventor 提供了一系列标准的用户界面组件，所以使用 Open Inventor 开发的应用程序可以向用户提供统一的用户界面体验。

如果杂志没有我们需要的部分产品，那么装修公司也允许用户自己设计定制，并直接从建材市场购买原材料。也许我们希望墙角是弯的而不是直的。Inventor 也允许定制对象（通过子类化，在 *The Inventor Toolmaker* 有阐述）。有了这些自由，就可以不受这些杂志的约束，同时也可以在适当的时候使用它们来节约时间和金钱。

如果想节约更多的时间，我们也可以从家装杂志中定购一套完整的房屋，这些房屋具有不同的风格：埃及风格的、田园风格的、维多利亚女王时代风格的、殖民地风格的。这些房屋类似于 Inventor 的节点工具包（*node kits*），节点工具包是常用对象的包装集。

当房子建完后，我们很难再用肉眼分辨出那些房子是用原始建筑材料建造的，那些使用预制件建造的。这些房子都有这上乘的质量，都使用了最好的材料，而且看起来都很坚固。都代表了非凡的创造力和与众不同的设计。

类似的说法也同样适合说明使用 OpenGL 和 Open Inventor 编写的应用程序，方法必须要适合于程序员的需要，并且两种方法可以随意组合使用，可以同时使用预建的 Inventor 对象和 OpenGL 指令。

Inventor 工具包

Inventor 提供了多种级别意义上的编程支持。在最终用户级别上，Inventor 为 3D 图形提供了统一的用户界面。在编程级别上，Inventor 工具包（见图 1-1）提供了下列工具，这些工具将在稍后的章节中做详细的解释：

- 3D 场景数据库，场景数据库中包含有形体、属性、组、引擎和传感器等节点对象。通常可以利用这些节点来创建一个分层的 3D 场景。
- 一组节点工具包，可用于方便地创建预制的 Inventor 节点组。

- 一组操作器，包括像手柄盒（handle box）和 轨迹球等的操作器节点。操作器节点是一种用户可以在场景数据库中直接进行操作的对象。
- Inventor Xt 系统组件库，包含渲染区（用于渲染的窗口）、材质编辑器、观察器（viewers）、和一些功能函数。它们通常用于执行一些高级的交互式任务。

本书将采用自下而上的方法来介绍 Open Inventor。下面首先介绍 3D 场景数据库。

场景数据库

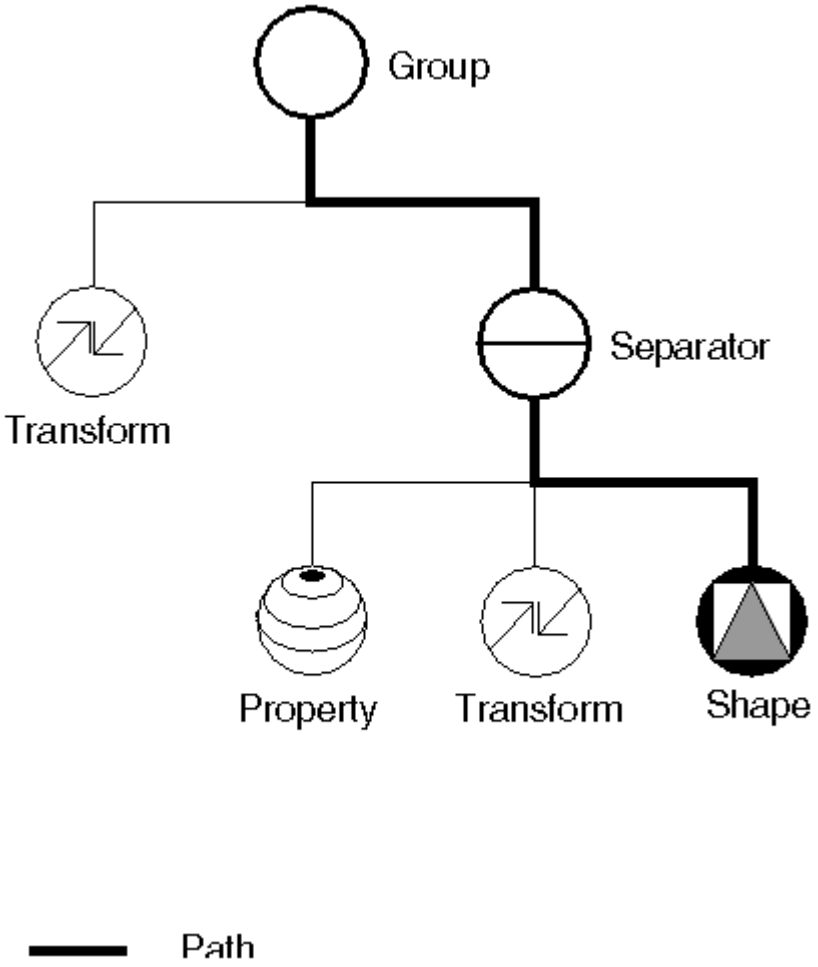
节点是一种用来创建 Inventor 3D 场景数据库的“积木块”（building block）。每个节点里都存有一些信息，这些信息可能是表面材质、形状描述、几何变换、灯光或者照相机。所有位于场景中的 3D 物体、属性、照相机和光源都可以看作是节点。

节点的有序集合被称为场景图（图 1-3 显示了一个简单的场景图，在“**关于这本书**”章节中，图 1-1 已经列出了本书所有场景图中使用的符号图标）。场景保存在 Inventor 的基础数据库中。Inventor 负责储存管理场景数据。Inventor 基础数据库可以包含多个场景。

在创建完一个场景后，可以对场景应用像渲染、拾取、查找、包围盒计算、写文件等操作或*动作*。

数据库图元（database primitives）的种类包括形体节点(例如球、立方体、圆柱体、网状曲面)、属性节点(例如，材质、灯光模型、纹理、环境)、和组节点（例如，隔离节点（separator）、level-of-detail（**LOD-Level of Detail 节点会随着摄影机的远近切换细节显示复杂度 译者注**）、开关节点（switch））。数据库图元中还特别包含了“引擎”和“传感器”。引擎是一种可以和场景中其他对象相连接的对象，通常它可以驱动场景中的某些物体，或者使场景中的某些物体和另外一些物体保持某种相互制约关系。（见 13 章 “引擎”）。传感器对象是一种可以探测出数据库中数据何时发生了改变的对象。如果它发现数据发生了变化，传感器会立即调用应用程序提供的回调函数。传感器可以响应定时请求（例如，每 n 秒做一次），或数据库中数据的变化（见 12 章 “传感器”）。

图 1-2 一个场景例子



节点工具包（Node Kits）

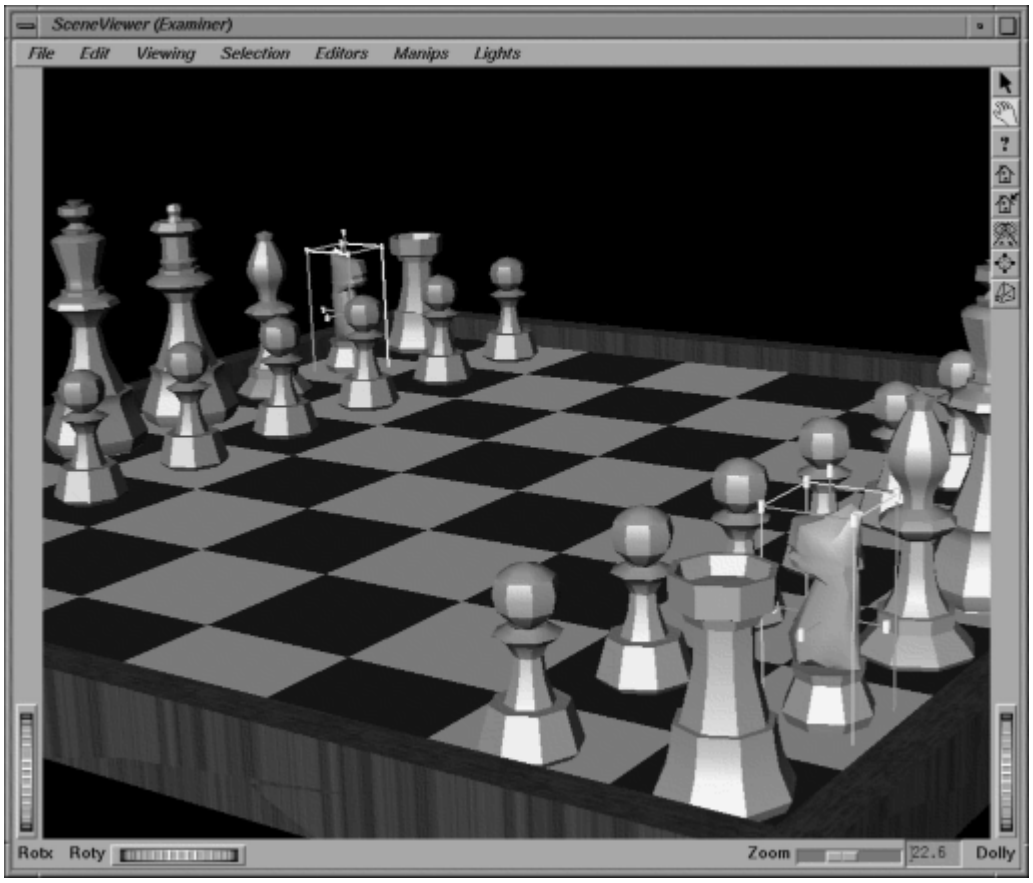
节点工具包的主要作用是有助于创建协调一致的场景数据。节点工具包是一些按照某种规则而组合在一起的节点集。一个与节点工具包相关联的模板（template）决定必要时将那些节点增加到工具包内，以及这些节点应该安排在那里。例如，SoShapeKit可以代表任意的Inventor形体对象。如果使用它，我们就不必分别单独创建和管理每个节点。缺省情况下，**SoShapeKit**的模板中包含有一个**SoCube**节点，在需要的时候，**SoShapeKit**还允许在正确的位置上插入材质、几何变换和其它属性节点。

节点工具包另外的一个作用，是可以定义应用程序自己的对象和语义（semantics）。举例来说，一个飞行模拟包软件中可能包含有多个代表飞机的对象。这些飞机对象都是由相同的场景数据结构组成的－例如，都有机身、机翼和起落架。这些飞机对象还包含有一些特有的方法－例如，**bankLeft()**（倾斜转弯）、**raiseLandingGear()**（升起起落架）等。对于使用这个模拟包的开发者而言，他可以按照相同的方式使用不同类型的飞机。即使不了解代表起落架子图形的详细数据，开发者仍可以将飞机升起来。因为模拟包中有一个通用的方法**raiseLandingGear()**。创造新对象和新方法需要通过子类化（subclassing）技术来扩展Open Inventor。这些技术将在*The Inventor Toolmaker* 书中作详细描述。我们强烈建议在程序中使用节点工具包，它将使程序变得简明和高效（order and policy）。

操作器

操作器是一种特殊类型的节点。它可以响应用户产生的事件，用户可以在屏幕上通过鼠标直接操作它。操作器通常都带有外观形状，而且都提供了将事件转换进数据库中的方法。手柄盒（*Handle Box*）就是一种操作器的例子，它是一个包围其它物体的包围盒，在它的顶角和边框上都有操作手柄。在图 1-3 中，手柄盒包住了国际象棋中的马，通过抓住操作手柄，并使用鼠标拖拽它，用户就可以改变手柄盒和其内部物体的大小和位置。操作器为应用程序提供了一种与 3D 物体间直接进行交互作用的简便方法.

图 1-3 手柄盒操作器



Inventor 组件库

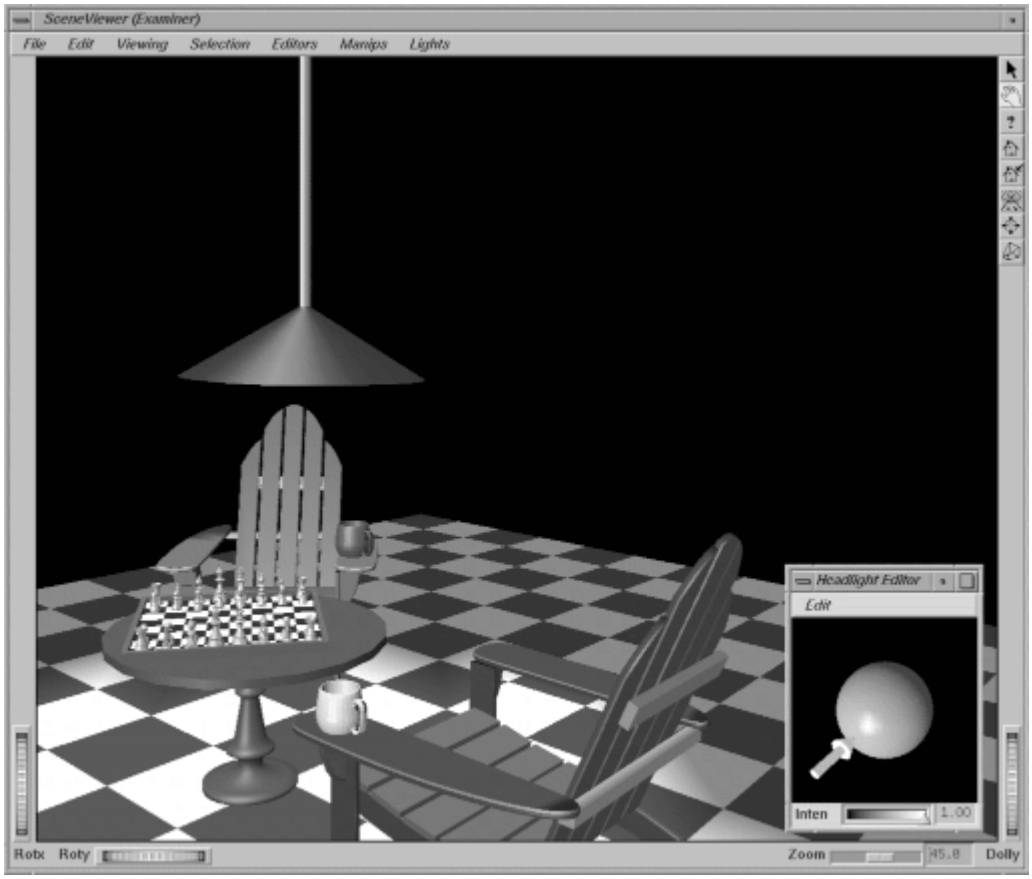
Inventor 组件库提供了对窗口系统的支持，并整合了 X 窗口系统。它包括下列特性：

- 渲染区(窗口)对象
- 消息主循环和初始化函数。
- 事件转换工具。
- 编辑器。
- 观察器。

渲染区收到一个 X 窗口系统的事件后，首先将事件转换成一个 Inventor 事件，然后将事件传递到像“操作器”这样的可以处理事件的“聪明”对象中。

Inventor 组件库还包括一组属于通用组件种类的观察器和编辑器。这些组件都是可重用模块，它们都包含有渲染区和用户接口。为了以不同的方式观察场景，它们通常用于编辑修改场景中的节点(材质、灯光、几何变换)。我们完全可以选择好一个 Inventor 组件在我们的程序中使用。这样做远胜于那些为了解决相同问题，而一遍又一遍地重复开发相同的功能。如果需要增加新的功能，我们可以写自己的组件，并将它增加到 Inventor 中(见 *The Inventor Toolmaker*)。组件库中包含有材质编辑器、灯光编辑器(见图 1-4)，fly 观察器（“飞”过场景）和 examiner 观察器(可以使用透视的方式观察一个物体)

图 1-4 一个组件例子：直射灯编辑器（右下角）



Inventor 类树

图 1-5 摘要的显示了Inventor的类树。基类在左边，派生类在右边。 **SoBase**是**SoFieldContainer**的基类，所有的节点类和引擎类都是由**SoFieldContainer**派生出来的。动作类都从**SoAciton**类派生出来。**SoXtComponent**是另外一个基类。 Xt渲染区、观察器和编辑器等都是从**SoXtComponent**派生出来的。位于类树右边的类继承基类的方法和域（fields）。

扩展工具包

Inventor 最重要的方面之一就是将新对象和新操作作为工具包的一个扩展的能力。

Inventor 所提供的扩展对象集的其中一个方法就是从已知的 Inventor 类派生出一个新类。具体可以阅读 *The Inventor Toolmaker* 书中创建新类的例子代码。

为 Inventor 增加新功能的另外一种方法是使用回调函数(*callback functions*)。 回调函数为定制新功能或不通过子类化的方式创建新节点提供了一种简单的机制。回调函数是一种用户提供的可以在特定条件下被调用的函数。Inventor 提供了下列的回调函数：

- **SoCallback**－在数据库中为所有的数据操作提供回调函数的通用节点(见第 17 章，“**在Inventor中使用OpenGL**”)。
- **SoCallbackAction**－在遍历数据库时，对每个节点调用一次回调函数(见第 9 章“**应用动作**”)。
- **SoEventCallback**－在接收到操作事件的时候调用用户自定义函数的节点 (见第 10 章“**处理事件和选择器**”)。
- **SoSelection**－选择回调节点 (见第 10 章“**处理事件和选择器**”)。
- **操作器**－为事件处理提供回调函数（见第 15 章“**拖拽器和操作器**”)。
- **SoXt** 组件－当发生一个变化时，调用它们自己的函数 (见第 16 章“**Inventor 组件库**”)。

图 1-5 Inventor 类树摘要 (1 / 3)

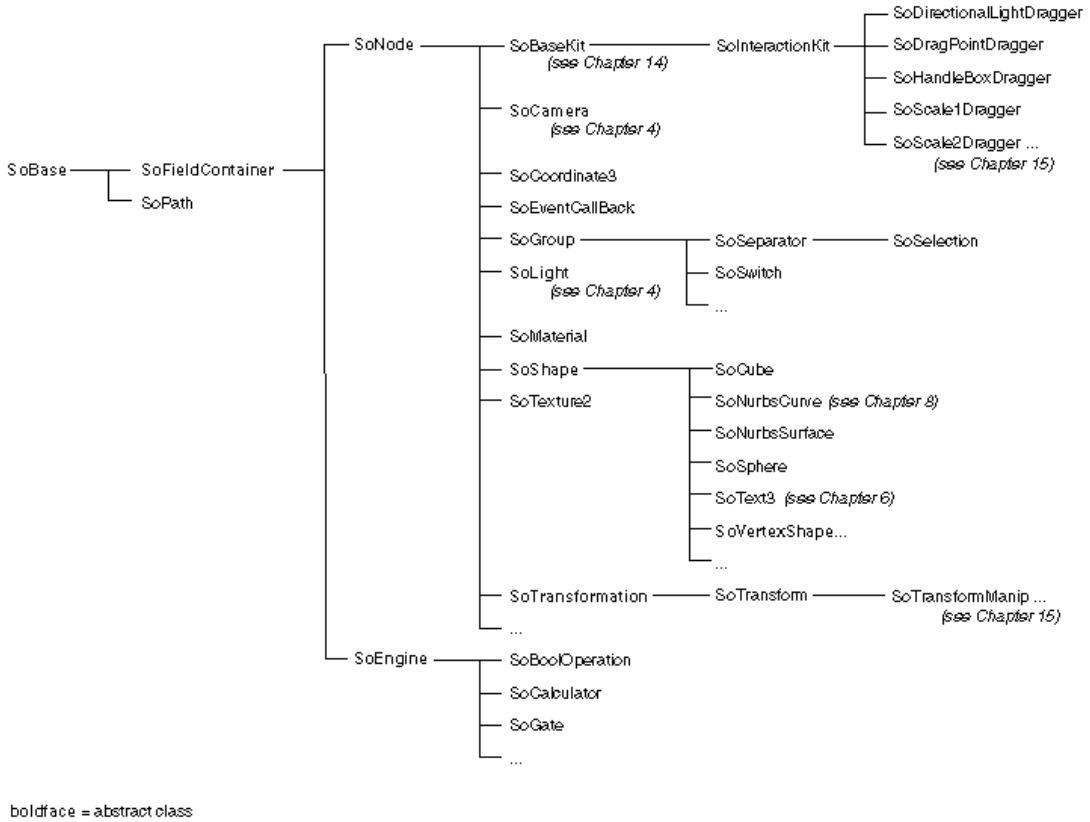


图 1-6 Inventor 类树摘要 (2 / 3)

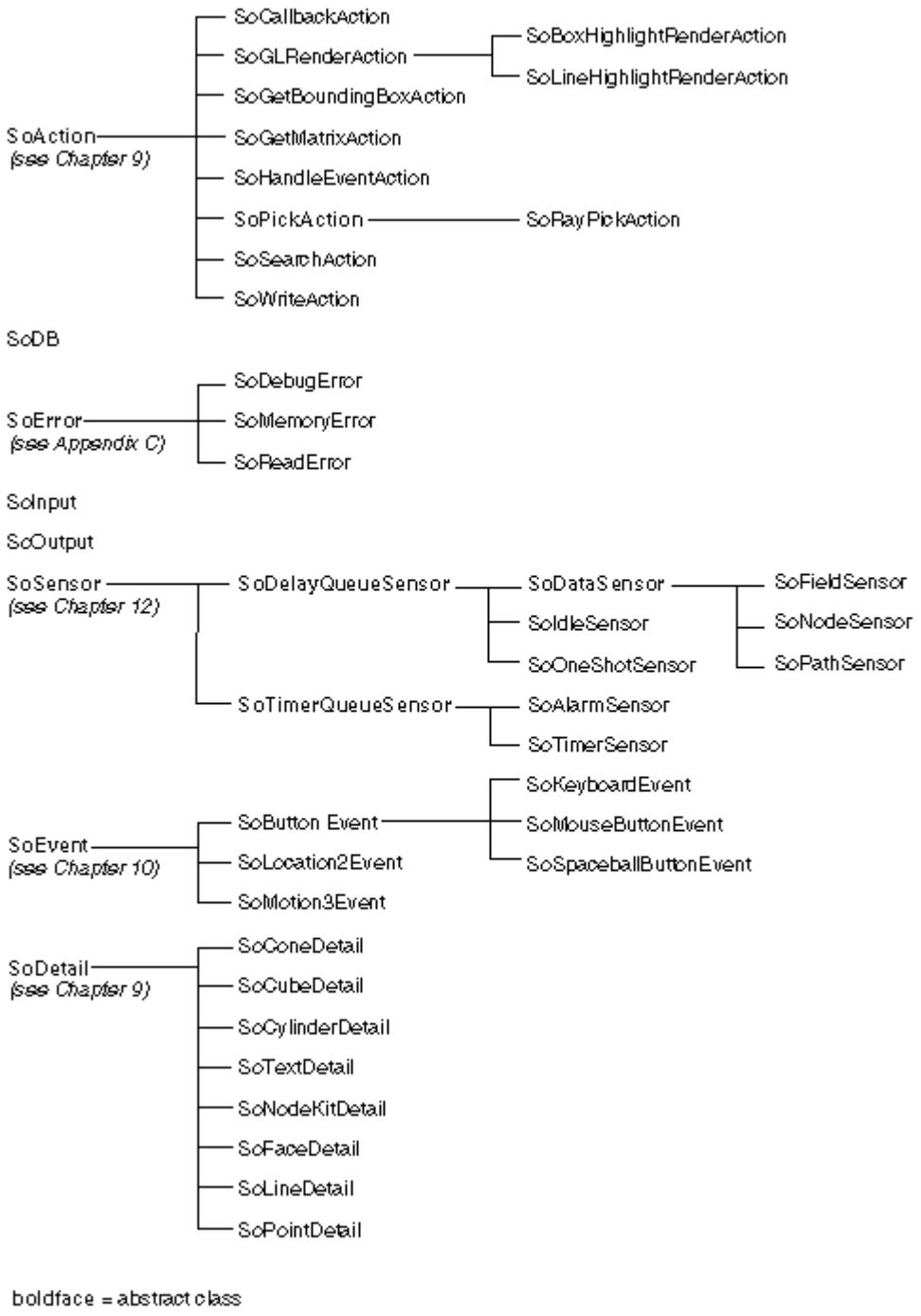
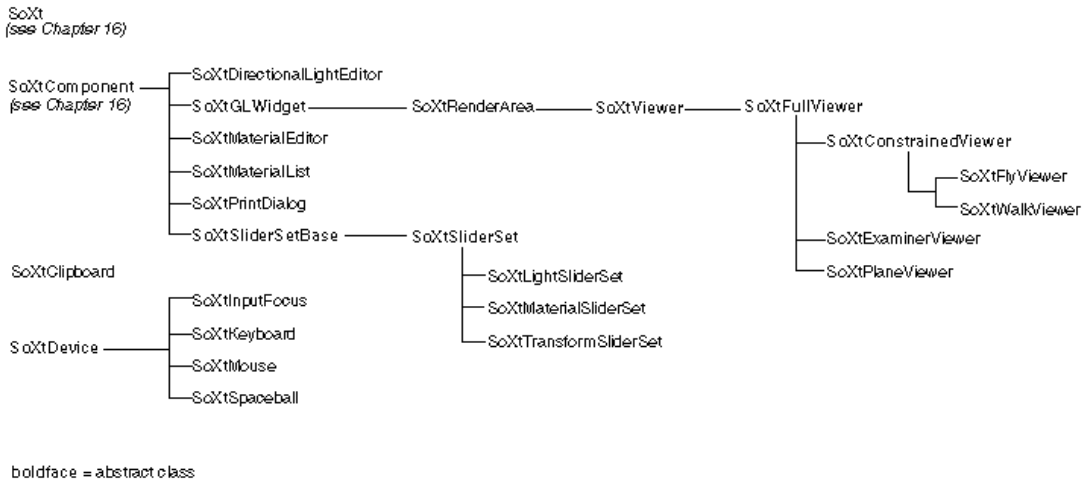


图 1-7 Inventor 类树摘要（3 / 3）


SoXt
(see Chapter 16)
SoXtComponent
(see Chapter 16)
SoXtDirectionalLightEditor
SoXtGLWidget
SoXtRenderArea
SoXtViewer
SoXtFullViewer
SoXtConstrainedViewer
SoXtFlyViewer
SoXtWalkViewer
SoXtExaminerViewer
SoXtPlaneViewer
SoXtMaterialEditor
SoXtMaterialList
SoXtPrintDialog
SoXtSliderSetBase
SoXtSliderSet
SoXtLightSliderSet
SoXtMaterialSliderSet
SoXtTransformSliderSet
SoXtClipboard
SoXtDevice
SoXtInputFocus
SoXtKeyboard
SoXtMouse
SoXtSpaceball
boldface = abstract class

第二章 一个 Inventor 例子

本章目标

在阅读完本章节后，我们将可以做以下事情：

- 理解 Inventor 程序的基本结构。
- 了解 Inventor 的使用约定。

本章将概述所有 Inventor 程序都带有的大约 5%的代码。本章将首先讨论一个显示红色圆锥体的简短程序。之后，逐步扩展这个程序，分别演示如何使用某些重要的 Inventor 对象：引擎、操作器和组件等。本章还将阐述 Inventor 命名约定和基本数据类型。

“Hello, Cone”

本章首先用一组例子程序作为开始，举例说明 Inventor 程序的基本结构形式。例 2-1 创建了一个红色的圆锥，然后在一个窗口中渲染显示。这个例子使用了一个 Inventor Xt 窗口，这是 Inventor 组件库的一部分。组件库带有窗口管理和事件处理的功能，并且还包含了一系列的 Inventor 组件(观察器和编辑器)。

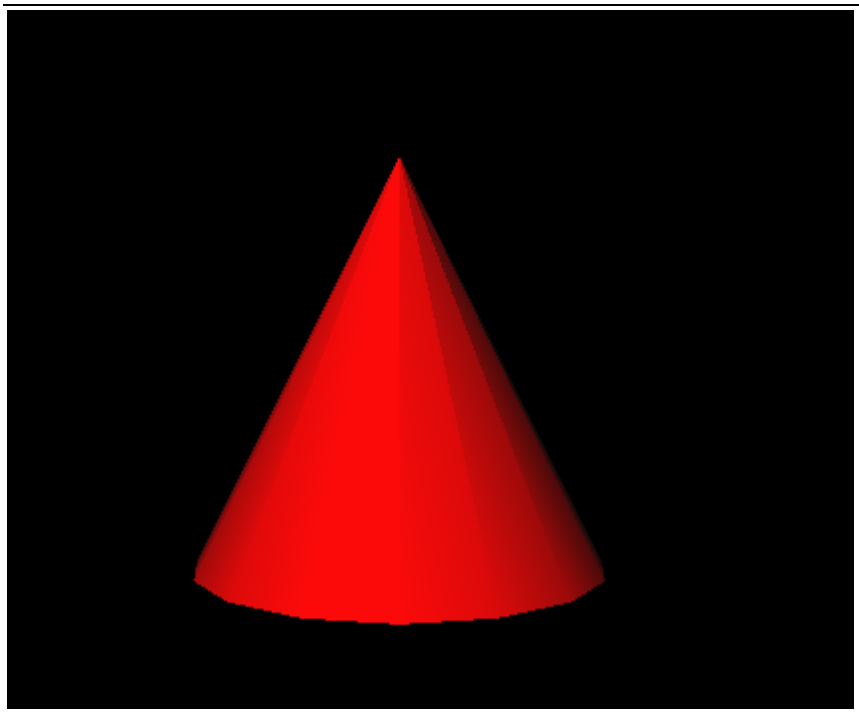
例 2-1 的代码创建了一个由照相机节点、灯光节点、材质节点和圆锥节点组成的简单场景。在以后的章节中，我们将继续探究如何创建节点、设置域值、构建数据库、应用动作等技术细节。本章的目的只是简单地让我们有一个使用体验，帮助我们开始编写第一个 Inventor 程序（The purpose of this chapter is simply to convey a feel for the tools Inventor offers and to help you get started writing an Inventor program）。

红色圆锥体

第一个例子程序将举例说明编写 Inventor 程序的基本步骤。

1. 首先创建一个用于场景渲染的窗口。本例子使用了**SoXtRenderArea**类。
2. 创建属性和形体节点，将它们合并到组节点中，最后建立场景。

图 2-1 显示了例 2-1 产生的圆锥图像。



例 2-1. 基本的“Hello Cone”程序

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

main(int , char **argv)
{
   // Initialize Inventor. This returns a main window to use.
   // If unsuccessful, exit.
   Widget myWindow = SoXt::init(argv[0]); // pass the app name
   if (myWindow == NULL) exit(1);

   // Make a scene containing a red cone
   SoSeparator *root = new SoSeparator;
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   SoMaterial *myMaterial = new SoMaterial;
   root->ref();
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // Red
   root->addChild(myMaterial);
```

```
   root->addChild(new SoCone);

   // Create a renderArea in which to see our scene graph.
   // The render area will appear within the main window.
   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

   // Make myCamera see everything.
   myCamera->viewAll(root, myRenderArea->getViewportRegion());

   // Put our scene in myRenderArea, change the title
   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Hello Cone");
   myRenderArea->show();

   SoXt::show(myWindow);  // Display main window
   SoXt::mainLoop();      // Main Inventor event loop
}
```

使用引擎旋转圆锥

例 2-2 举例说明了如何使用引擎来旋转圆锥。一个引擎节点与一个**SoRotationXYZ**节点的角度数据域相关联。当实时时钟（real-time clock）发生变化时，引擎会同时修改在rotationXYZ节点中的角度数据。这样将导致圆锥不断地循环旋转。在每次修改后，场景会自动地被渲染区重新渲染。连续的旋转就可以达到我们想要的旋转圆锥的效果。

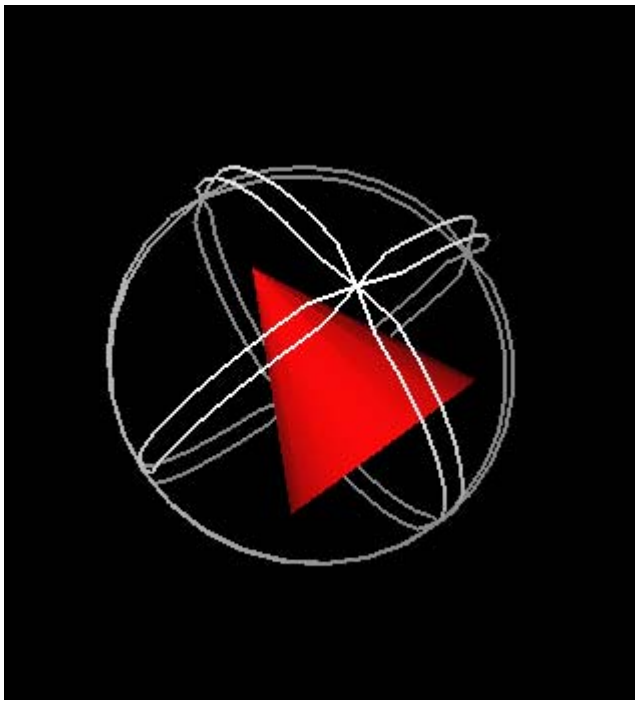
例 2-2. “Hello Cone”程序，使用引擎

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/engines/SoElapsedTime.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoRotationXYZ.h>
#include <Inventor/nodes/SoSeparator.h>

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);
```

```
    SoSeparator *root = new SoSeparator;
    root->ref();
    SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
    root->addChild(myCamera);
    root->addChild(new SoDirectionalLight);

    // This transformation is modified to rotate the cone
    SoRotationXYZ *myRotXYZ = new SoRotationXYZ;
    root->addChild(myRotXYZ);

    SoMaterial *myMaterial = new SoMaterial;
    myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);   // Red
    root->addChild(myMaterial);
    root->addChild(new SoCone);

    // An engine rotates the object. The output of myCounter
    // is the time in seconds since the program started.
    // Connect this output to the angle field of myRotXYZ
    myRotXYZ->axis = SoRotationXYZ::X;     // rotate about X axis
    SoElapsedTime *myCounter = new SoElapsedTime;
    myRotXYZ->angle.connectFrom(&myCounter->timeOut);

    SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
    myCamera->viewAll(root, myRenderArea->getViewportRegion());
    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle("Engine Spin");
    myRenderArea->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

增加上轨迹球操作器

接下来的两个例子将演示编辑场景中节点的另外两种方法。例 2-3 针对前面的第一个例子(见图2-2)增加上一个操作器(轨迹球)。轨迹球本身看上去像3个围绕在圆锥四周的圆环。当鼠标左键点击轨迹球后，轨迹球以不同的颜色高亮显示自己，以此表明它现在是处于激活状态。当轨迹球处于激活状态时，可以使用鼠标同时旋转轨迹球和位于它内部的物体(在这里是圆锥)。在这个例子中，轨迹球替换掉了例 2-2 中的**SoRotationXYZ** 节点。每次用户旋转轨迹球时，轨迹球的内部数据都会被修改，同时圆锥也会跟它一起旋转。因为渲染区内有一个传感器附着在场景中，所以当场景数据被修改后，场景将会自动被渲染，这样圆锥看上去就运动起来了。



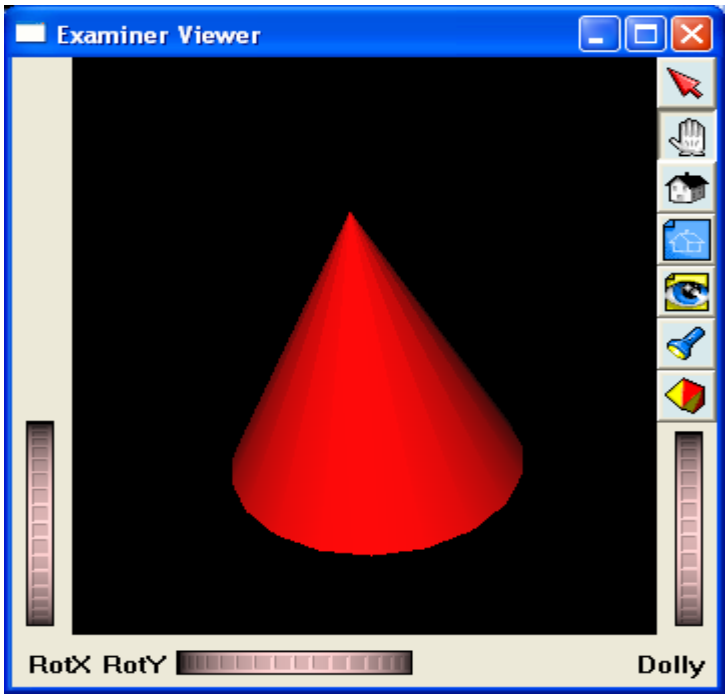
例 2-3 “Hello Cone” 使用轨迹球操作器

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/manips/SoTrackballManip.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   root->addChild(myCamera);               // child 0
   root->addChild(new SoDirectionalLight); // child 1
   root->addChild(new SoTrackballManip);        // child 2
```

```
    SoMaterial *myMaterial = new SoMaterial;
    myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);
    root->addChild(myMaterial);
    root->addChild(new SoCone);

    SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
    myCamera->viewAll(root, myRenderArea->getViewportRegion());
    myRenderArea->setSceneGraph(root);
    myRenderArea->setTitle("Trackball");
    myRenderArea->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

增加 Examiner 观察器

例 2-4 使用Examiner观察器组件替换掉第一个例子中使用的渲染区。Examiner观察器通过修改照相机的参数，允许我们从不同的方位来观察圆锥。它提供了一个用户接口，允许使用鼠标修改照相机在场景中的位置。（注意，这个例子的效果和例 2-3 中使用轨迹球的效果很相似。但是，这里是移动照相机，而不是移动圆锥本身（**这和在现实生活中观察物体是一样的，我们可以旋转物体本身观察它，也可以通过我们自己绕这个物体转动来观察它，这两种观察方式的结果都是一样的。译者注**））。这个程序不需要创建一个照相机，也不需要调用照相机的**viewAll()**函数，因为Examiner观察器内部已经自动替我们做好了这些工作。



例 2-4. “Hello Cone” 使用 Examiner 观察器

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

main(int , char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);
   root->addChild(myMaterial);
   root->addChild(new SoCone);

   // Set up viewer:
   SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Examiner Viewer");
   myViewer->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

命名约定

在Inventor中基本数据类型的名字都是以字母**Sb**开始的，例如：

- SbColor
- SbViewVolume

除了上面的基本数据以外，在Inventor中所有其他的类都是以字母**So**作为前缀（场景对象）。例如：

- SoCone
- SoPerspectiveCamera

- SoMaterial
- SoTransform

方法和变量使用小写字母作为开头字母。在类、方法、变量的名字中，每个单词都以大写字母开头。例如：

- getNormal()
- setSceneGraph()
- myCube

枚举类型数值都是用大写字母表示的。例如：

- FILLED
- PER_PART

场景基本类型（Scene Basic Types）

本节将讨论Inventor的**Sb** 基本数据类型。这些数据类型被很多的Inventor对象所使用。Inventor为不同类型的数据转换以及执行特定的 3D操作提供了很多方便实用的方法。

Inventor 定义了下列类型：

SbBool	布尔值 (TRUE or FALSE)
SbBox*nx*	2D 矩形或者 3D 的立方体数据。这些数据所表示的各个侧面都是平行于主轴的。可以通过设置对角线上的两点坐标来指定它们（包括 SbBox3f, SbBox2f, SbBox2s）；n 表示的是内部数值的个数（2 或 3 个），x 表示的是内部数值的类型（f 表示浮点类型，s 表示短整形）
SbColor	RGB (红/绿/蓝) 颜色值。这个类有转换函数可以将颜色值转换到其它的颜色空间中（**例如，YUV 颜色空间（色调，饱和度，亮度）。译者注**）
SbCylinder	圆柱类
SbLine	有向 3D 直线类
SbMatrix	4×4 矩阵类
SbName	字符串类。这些类被保存在一个特殊的表中（**hash 哈希表。译者注**），可以快速方便的进行查找。通常用于标识对象.
SbPList	通用指针(void *)列表类。
SbPlane	3D 平面类
SbRotation	代表一个绕任意轴进行旋转的 3D 旋转类
SbSphere	球类
SbString	“灵巧”的字符串类，对处理字符串提供了许多方便的方法。
SbTime	表示秒，毫秒的时间类；也可以使用 timeval 结构类型来表示
SbVec*nx*	2D 或 3D 向量类。用于表示点或者方向（包括 SbVec2f, SbVec3f, SbVec2s)；*n* 是在这个类中的数据个数(2, 3, or 4 个)，*x* 是数据类型 f 表示浮点类型，s 表示短整形)

SbViewportRegion	代表在显示窗口中的视口区域
SbViewVolume	视见体裁剪（view volume）类。 （或称“可视体积”。当使用平行投影时，view volume 就是一个长方体，当使用透视投影时，它就是一个近大远小的截锥方体，Inventor 使用这个类来判断场景中的物体是否是处于长方体或者截锥方体的范围内，如果物体已不在视见体范围之内，Inventor 就不会渲染这个物体，以提高渲染效率。具体说明可翻阅本书第四章“照相机和灯光”。译者注）

类方法

每种**Sb** 类都带有一些与其相关的实用操作。例如，可以对**SbVec3f**类型的变量进行取反、乘矩阵、单位化等操作。下面的代码将定义一个基于特定方向和单位长度的向量。

```
SbVec3f v(1.0, 2.0, 3.0); // declares and initializes the vector
v.normalize(); // normalizes the vector to unit length
```

类似的，**SbMatrix** 类也包含有一些实用的方法，例如，**multVecMatrix()** 这个方法将当前矩阵与一个行向量相乘，并返回计算结果。**rotate()** 方法可以按照给定的旋转量数据设置矩阵。*Open Inventor C++ Reference Manual* 书中对每个**Sb**类的可用方法都有完整的描述。

类型 VS 域

第三章“节点和组”有关于域（*fields*）的完整讨论。域是节点用来存储参数的内存结构。每个域都带有一个特定类型的变量。域通常是包含在Inventor的节点中。许多域都带有一个相对应的**Sb** 类型的变量。例如

- SoSFVec3f 域带有一个 SbVec3f 类型的变量
- SoSFRotation 域带有一个 SbRotation 类型的变量
- SoSFName 域带有一个 SbName 类型的变量

Inventor 使用的坐标系统

Inventor 对 3D 数据使用的是右手坐标系，从屏幕内指向外，表示 z 轴的正方向。所有的角度单位都是弧度。对象都是在自己的局部坐标系空间下进行描述的，既众所周知的“对象坐标系空间”（*object coordinate space*）。当场景中的所有物体都已经进行完坐标变换后，那么它们就都在“世界坐标系空间”下描述了（*world coordinate space*）。照相机和灯光节点处于世界坐标系空间下。

包含文件（Include Files）

Inventor为每个类都提供相应的包含头文件。在程序中使用Inventor类之前都需要首先包

含相应的头文件。例如，**SoSphere** and **SoTransform** 这两个节点类需要包含SoSphere.h 和 SoTransform.h两个头文件。绝大多数的Include头文件都位于一个子目录中－例如 *nodes/SoSphere.h* 和 *sensors/SoNodeSensor.h*。

另外，如需要使用 Inventor 组件库来编写交互式应用程序的话，还需要包含 SoXt.h 头文件。

如果使用 C 应用程序编程接口进行程序设计的话，需要使用 *Inventor_c* 目录来代替 *Inventor* 目录。

第三章 节点与组

本章目标

在阅读完本章节后，我们将可以做下列事情：

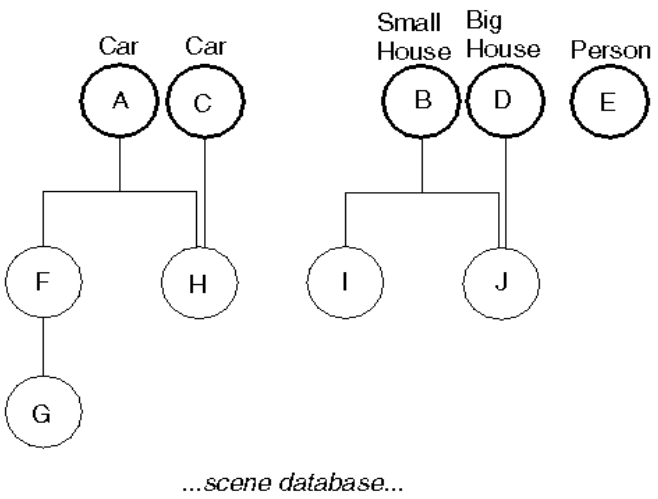
- 利用形体、属性、组等节点创建图形场景。
- 解释节点如何继承上层节点的属性数据。
- 理解隔离节点（separator）的重要性。
- 解释在场景中使用共享节点实例的优势。
- 定义“路径”术语，并且解释路径存在的必要性。
- 设置查询域值。
- 忽略节点中的某个特定域。
- 了解在 Open Inventor 中如何删除节点。
- 使用 Inventor 的运行时类型检查机制。

本章将举例说明怎样利用形体、属性、组等节点创建图形场景。解释遍历图形场景的通用规则（主要是 OpenGL 的渲染遍历）。最后还将介绍场景数据库操作（actions）和遍历状态的概念。

场景数据库

Inventor场景数据库是由代表一个或多个 3D场景的数据信息所组成的。如图 3-1 所示，数据库**SoDB**可以包含多个图形场景。每个场景都是由一组相关的 3D对象和属性构成的。例如，在图 3-1 中，场景可以表示为一辆汽车、一栋小房子、或者另外一辆汽车、一栋大房子和一个人。

图 3-1 Inventor场景数据库



场景数据库有两个基本操作（或方法）可供调用。首先需要初始化数据库：

```
SoDB::init()
```

这必须是在我们编写Inventor程序时的第一个Inventor函数调用。如果在我们开发的程序中使用了Inventor组件库的话，那么当调用**SoXt::init()**的时候，场景数据库将会自动被初始化(见第 16 章“Inventor组件库”)。如果没有使用组件库，但使用了有交互作用的节点（**例如操作器，译者注**）或节点工具包（Node Kits）时，必须首先调用**SoInteraction::init()**函数，这个函数会同时初始化数据库、交互作用模块和节点工具包。

其次，可以将一个 Inventor 文件读到场景数据库中，增加新的图形场景。

```
SoSeparator* readAll(SoInput *in)
or
SbBool read(SoInput *in, SoNode *&rootNode) const
```

或者

```
SbBool read(SoInput *in, SoPath *&path) const
```

使用第一种调用方法，Inventor 将从参数 *in* 所指定文件中读取场景数据，并返回一个指向 SoSeparator 节点的指针，这个 SoSeparator 节点将是所读文件中所有场景的根节点。使用第二种调用方法，Inventor 将从参数 *in* 所指定文件中读取场景数据，然后将文件中的场景根节点作为结果返回给 *rootNode* 参数。使用第三种调用方法，Inventor 将从参数 *in* 所指定文件中读取场景数据，然后将文件中的场景根节点作为结果返回给 *path* 参数。如果读取的过程中发生了错误，函数将返回 FALSE。(查看第 11 章“文件格式”，了解更多有关 **SoInput** 的内容)。

图形场景

图形场景是由一个或多个代表形体、属性、组等节点对象组成的。通过向组节点增加子节点，可以创建出一种分层场景(Hierarchical scenes)。分层场景是一个有向无环图(*directed acyclic graph)*。

注意：尽管场景负责组织和管理所有的 Inventor 节点，但是 Inventor 并没有强制规定场景数据库内部是如何组织的。例如，我们可以创建自己的节点，新节点可以保存在我们自己的数据结构中，而不一定非要保存在场景中。（*The Inventor Toolmaker* 书中列有更多有关扩展 Open Inventor 方面的内容）。

图 3-1 展示了一个包含有五个图形场景的简单数据库。图形场景中最顶层的节点称作根节点(节点 A 至节点 E)。请注意观察，节点 H 是如何连接到两个不同的父节点上的。这种情

况叫做共享实例(*shared instancing*)。还请注意，节点 E 没有和数据库中任何其它的节点相连接。通常这只是一种临时的状态，随着不断地增加场景内容，这个节点就应该会和其它的节点相连接了。

节点的类型

节点是组成图形场景的基本元素。它内部包含用于定义 3D 形体、属性、或组的数据和方法。当创建了一个节点，这个节点会被当作根节点自动地保存到场景数据库中（When a node is created, it is automatically inserted into the database as a root node。**这句话的意思有些让人费解，请读者自己甄别。译者注**）。通常的情况下，可以通过将节点与数据库中的其它节点相连接，构造出一种具有分层关系的场景。

节点可以分为三大基本类：

- 形体节点(*Shape nodes*)，代表 3D 几何模型。
- 属性节点(*Property nodes*)，表现对象的外观或其它场景特征。
- 组节点(*Group nodes*)，是一种将节点聚合包含进场景中的容器（containers）。

这些分类的划分并不是特别严格的，这样做不过是可以帮助我们更好地学习 Inventor 所包含的类。

创建节点

使用 C++ **new** 操作符来创建一个节点，例如：

```
SoSphere *headSphere = new SoSphere;
```

注意：不要创建节点数组。(具体原因见后面的“怎样删除节点”章节)。**(既不能这样程序，SoSphere *headSphere = new SoSphere[5]。译者注）**

注意：尽管是使用 new 操作符创建了新节点，但却不能使用 delete 操作符删除它们。在后面的“如何删除节点”章节中，将详细讨论如何在 Inventor 中删除节点，以及节点何时被删除的问题。理解“引用计数”的原理对于我们使用 Inventor 是至关重要的，因为我们必须知道在什么样的条件下，节点会被自动删除。

节点里有什么 ？

每个节点都是由一组叫做“域”的用于描述节点参数的数据元素组成的。例如，一个点光源节点(**SoPointLight**)包含了四个域：*intensity*(亮度)、*color*(颜色)、*location*(位置)和*on*(开关)。**intensity**里包含有一个从 0.0 (无光)到 1.0 (最亮)的变量值。**color**用于设置光源的红/绿/

蓝照明色。**location**用于指定光源的位置。**on**指定灯光是否处于开的状态。

Inventor定义了许多域的类型。每种域类型都有唯一的方法来访问其数据。在每个节点中，根据域的用途，每个域都被命名了相应的名称。例如：下面是一些节点和它们包含的域：

节点	域
SoCoordinate3	point
SoNormal	vector
SoMaterial	ambientColor diffuseColor specularColor emissiveColor shininess transparency
SoPerspectiveCamera	viewportMapping position orientation aspectRatio nearDistance farDistance focalDistance heightAngle

注意，包含有多个数值的域，像在**SoCoordinate3** 节点中的**point**，这些域也只有一个单一的名字。

当对节点执行动作时发生了什么? (高级内容)

每个节点都会执行自己的动作行为（Each node implements its own action behavior）。当需要对场景执行一个特定的动作时，我们必须首先要创建一个动作类的实例(例如，**SoGLRenderAction** 或 **SoGetBoundingBoxAction**)，然后再将这个动作应用到场景的根节点上。数据库对每个动作都管理着一个叫做遍历状态(*traversal state*)的数据结构，这个数据结构是在某个给定时刻内动作的元素与参数的集合（For each action, the database manages a *traversal state*, which is a collection of elements or parameters in the action at a given time）。通常，当执行一次动作时，动作将按照从上到下，从左到右的顺序遍历整个场景。在遍历的过程中，节点将根据它们对此次动作所做的特定行为来修改遍历状态。

本节将集中讨论OpenGL 渲染动作(*OpenGL rendering action*)，因为创建 3D场景的最主要动机之一就是为了观察和操作 3D对象。渲染遍历状态是由一组可以被给定的节点类修改的元素所组成的。当执行一个渲染动作时，将以指定的方式使用和解释每个元素（When a rendering action is applied, each element is used and interpreted in a specified manner）。下面列出了少数的几种在遍历状态中使用的元素：

- 当前几何变换。
- 当前的材质成份。
- 当前光照模型。
- 当前绘制风格。
- 当前字体。
- 当前坐标系。
- 当前法向向量
- 当前灯光。
- 当前观察方式（Current viewing specification）（**主要是定义平行投影，还是透视投影。译者注**）

举例来说，**SoMaterial**节点用于设置遍历状态中各种材质元素的当前值，**SoDrawStyle**节点用于设置遍历状态中绘制风格元素的当前值。还有像**SoSphere**这类的形体节点，对渲染遍历来说是尤其重要的。因为在渲染的时候，这类节点会使用遍历状态的当前参数，将它们自己形状绘制在屏幕上。

形体节点

形体节点用于表示一种 3D几何对象。它们比较特殊，因为它们代表的是一些受属性节点和组节点影响的实际物体。在执行渲染动作期间，它们实际上会将自己形状绘制在屏幕上。形体节点类包括：**SoSphere**、**SoIndexedFaceSet** 和 **SoText3** 等类。图 3-2 展示了部分形体节点类的类树图：

属性节点

属性节点代表的是像“表面材质”、“绘制风格”、或“几何变换”等这些用于展现场景外观和特有性质的节点。图 3-3 展示了包含属性节点的部分类树图。属性节点可以分成若干类，场景图一般使用三种不同的图标来表示它们：

- 变换类图标：表示执行几何变换的属性节点，像**SoTransform**、**SoRotation**、**SoScale**、**SoTranslation**、**SoRotationXYZ** 和**SoResetTransform**等节点。这些节点都是从**SoTransformation**派生出来的。
- 外观类图标：表示可以修改对象外观的属性节点，像**SoMaterial**、**SoBaseColor** 、**SoMaterialBinding**、**SoComplexity**、**SoDrawStyle**、**SoLightModel** 和**SoFont**。
- 度量类图标：示包含有坐标系、法线、或其它几何信息的节点，像**SoCoordinate3**、**SoCoordinate4**、**SoProfileCoordinate2**、**SoProfileCoordinate3**、**SoNormal** 和 **SoNormalBinding**。

通常，在遍历过程中，属性节点将自己的新值替换掉在遍历状态中相应元素的值。但是几何变换节点是一个例外，它们会和当前几何变换进行累积运算。

让我们以材质节点为例。材质节点用来表现物体的表面和颜色的属性。假设我们希望创建一种类似青铜的材质效果，首先要创建一个材质节点，然后设置适当的域值。

```
SoMaterial *bronze = new SoMaterial;

// set field values
bronze->ambientColor.setValue(.33,.22,  .27);
bronze->diffuseColor.setValue(.78, .57,  .11);
bronze->specularColor.setValue(.99, .94, .81);
bronze->shininess = .28;
```

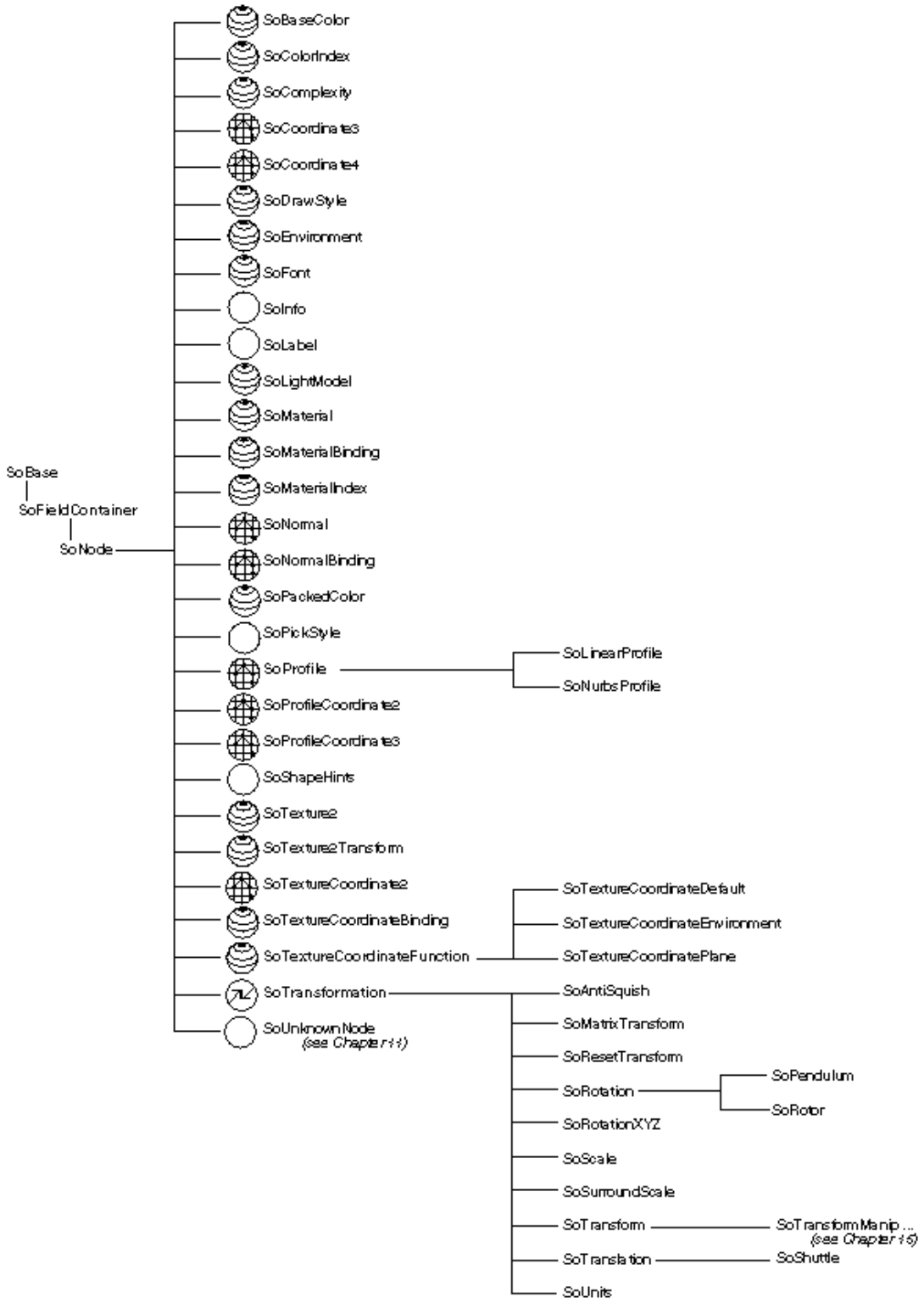
如果没有显式地为节点设置域值，Inventor 将会对这些域使用缺省值（具体请查阅 *Open Inventor C++ Reference Manual*）。例如，在上面的例子中，没有对材质节点的 transparency 域赋值，Inventor 将会对 transparency 使用其缺省值 0.0。

SoTransform节点可以产生几何变换，它包含有缩放域、旋转域、平移域。下面的代码定义了一个沿着y轴方向平移-1 单位的几何变换节点。

```
SoTransform *myXform = new SoTransform;

// set field value
myXform->translation.setValue(0.0,  -1.0,  0.0);
```

为了使这个变换产生效果，这个节点必须要插入到场景的某个适当位置上。(这里是插在需要平移的形体节点之前)。

图 3-3 属性节点类



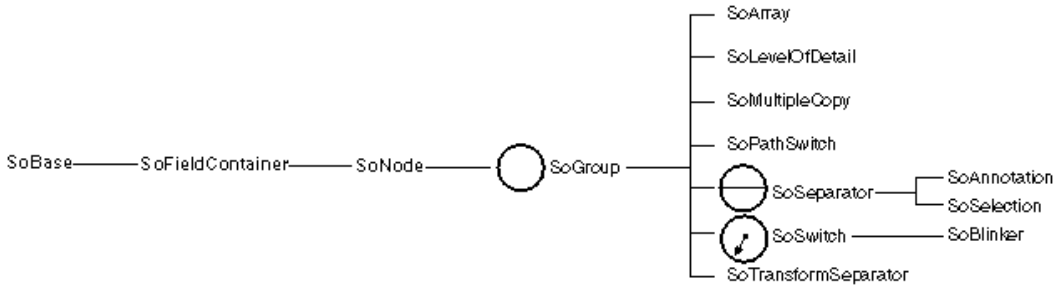
组

组节点是一个用于收集子节点对象的容器。组节点可以将属性、形体、和其它组节点收

集到场景中。图 3-4 展示了部分组节点类的类树图。Inventor 中有多种不同类型的组节点，每种组节点都有其特定的“分组”(grouping)特性。

当创建完一个组节点后，初始情况下它是没有子节点在其中的。**SoGroup**是所有组节点的基类。所有从它派生出来的节点类都有一个**addChild()**方法。

图 3-4 组节点类



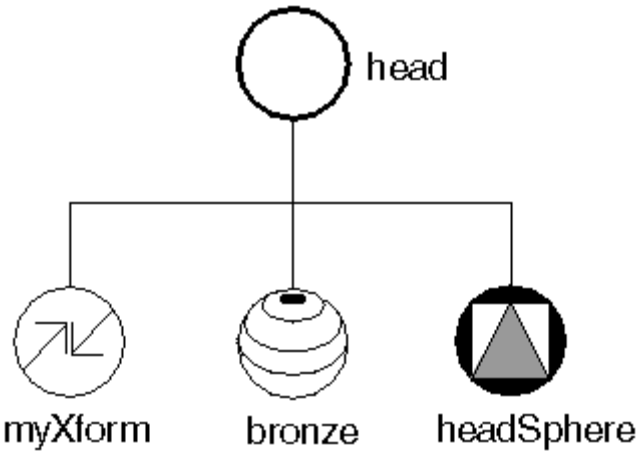
创建组节点

假设，如果想将前面创建的几何变换节点、材质节点和球体节点合并到机器人的“head”组节点中，首先要创建一个**SoGroup**节点，然后按照下列的步骤调用**addChild()** 方法，将其他子节点包含进组节点中。

```
SoGroup *head = new SoGroup;

head->addChild(myXform);
head->addChild(bronze);
head->addChild(headSphere);
```

图 3-5 显示了这个组的结构图。图 1-1 列出了在所有场景图中使用的图标。按照惯例，组节点中的第一个子节点显示在最左边，子节点是按照从左到右的顺序显示的。

图 3-5 一个简单的组



子节点的顺序

如前面的代码所示，**addChild()**方法会将某个特定节点增加到组节点内子节点列表的末端位置上。每个加到组中的子节点都分配有一个关联的索引值。组中第一个子节点索引值为 0，第二个子节点的索引值为 1，以此类似。

insertChild()方法

```
void  insertChild( SoNode *child,  int newChildIndex);
```

按照参数 *newChildIndex* 所指定的位置，将子节点插入到组中。例如：

```
SoDrawStyle *wireStyle;

wireStyle = new SoDrawStyle;
wireStyle->style = SoDrawStyle::LINES;
// Insert as child 1 (the node right after the first child,
// which is child 0.
body->insertChild(wireStyle, 1);
```

将一个框线(wireframe)绘制风格的节点作为第二个子节点插入到 body 组中。

组节点还有其他可调用的方法，例如查询在组中有多少个子节点，查找某个特定子节点的索引值，根据给定的索引值使用相应的子节点，以及移除子节点等方法。

子节点的顺序为什么是重要的 ？

每种节点类对于给定的数据库动作都有自己的响应方式。在本节的讨论中，我们假设只处理 GL 渲染动作 (简称渲染)。

- 如果当前渲染的节点是组节点，那么组节点将按照顺序对其每个子节点调用渲染动作，调用顺序通常都是按照子节点在场景中从左到右的顺序。
- 每个子节点依次执行它们自己的渲染方法，这些方法在某些方面会影响遍历状态（见第 9 章“应用动作”）。如果子节点是一个属性节点，那么它可能会修改诸如像散射光颜色、物体缩放比例、线宽度等当前遍历状态的元素数据。绝大多数的属性节点只是简单地（使用自己的数值）替换掉遍历状态中对应元素的数值(例如，青铜材质的节点将使用自己的数值替换掉遍历状态中当前材质的数值)。几何变换是一种例外情况，它们是彼此互相结合，累积产生合成变换。
- 如果当前渲染的节点是形体节点的话，那么形体节点将使用当前的遍历状态来绘制自己。

渲染时，Inventor 将以场景的根节点作为开始，按照从左到右，从上到下的顺序遍历整个场景。注意，在场景中的右边节点（下边节点）将继承由左边节点（上边节点）设置的遍

历状态。

图 3-6 展示了节点的继承状态。当渲染 *waterMolecule* 节点时，*waterMolecule* 节点将首先访问它的第一个子节点 *oxygen*。然后 *oxygen* 组节点将按照下面的顺序分别访问它自己的子节点：

1. 材质节点(*redPlastic*)将当前遍历状态中的材质元素修改成有红色光泽的材质。
2. 球体节点(*sphere1*)将使用当前的遍历状态渲染一个球体。一个有红色光泽的球体将被绘制在坐标原点的位置上。

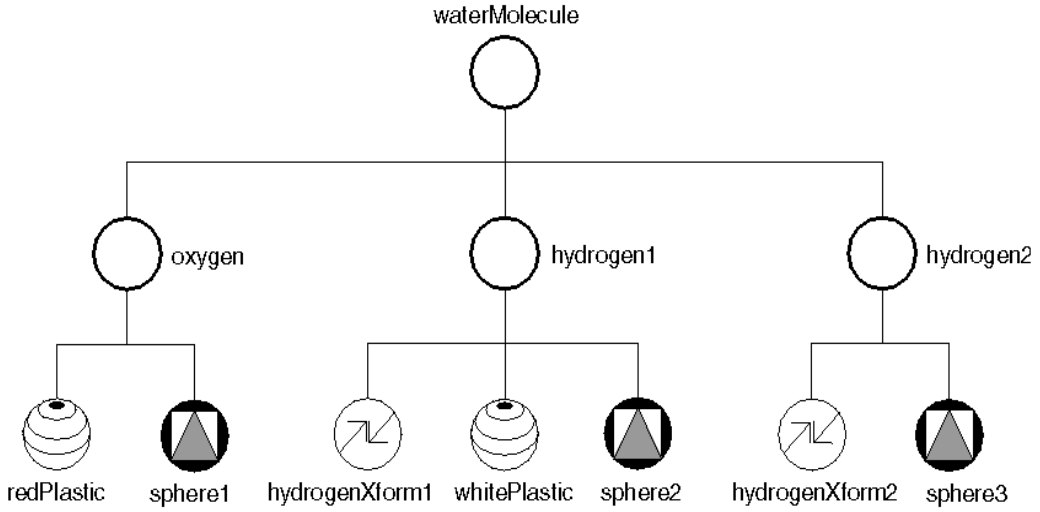
场景继续遍历右边的下一个组节点 *hydrogen1*，这个组节点同样按照从左到右的顺序依次访问它的每个子节点：

1. 几何变换节点(*hydrogenXform1*)修改了变换矩阵(也就是说，它在 x,y,z 轴上等比缩小了 75%)。同时它还为变换矩阵增加上了一个 0.0, -1.2, 0.0(分别在 x,y,z 轴方向)的平移变换。
2. 材质节点(*whitePlastic*)将当前遍历状态中的材质元素修改成有白色光泽的材质。
3. 球体节点(*sphere2*) 将使用修改过的遍历状态渲染另一个球体。这个球体是白色的。另外，因为在它的组节点中有**SoTransform**节点(*hydrogenXform1*)，所以*sphere2* 显示在一个新的位置上，并且它的大小也是按比例缩小的。

接下来，*hydrogen2* 组节点按照从左到右的顺序访问它的子节点：

1. 几何变换节点(*hydrogenXform2*)修改变换矩阵，在+x 轴和+y 轴方向进行了平移。
2. 球体节点(*sphere3*) 将使用修改过的遍历状态渲染第三个球体。这个球仍然是白色的，并且也被缩小了 0.75，这是因为它继承了在 *hydrogen1* 组节点中的属性。

图 3-6。 化合物组节点(Combining Groups)



例子 3-1 演示了创建这个分子节点的代码：

例 3-1 Molecule.c++

```
// Construct all parts
SoGroup *waterMolecule = new SoGroup;      // water molecule

SoGroup *oxygen = new SoGroup;             // oxygen atom
SoMaterial *redPlastic = new SoMaterial;
SoSphere *sphere1 = new SoSphere;

SoGroup *hydrogen1 = new SoGroup;          // hydrogen atoms
SoGroup *hydrogen2 = new SoGroup;
SoTransform *hydrogenXform1 = new SoTransform;
SoTransform *hydrogenXform2 = new SoTransform;
SoMaterial *whitePlastic = new SoMaterial;
SoSphere *sphere2 = new SoSphere;
SoSphere *sphere3 = new SoSphere;

// Set all field values for the oxygen atom
redPlastic->ambientColor.setValue(1.0,  0.0,  0.0);
redPlastic->diffuseColor.setValue(1.0,  0.0,  0.0);
redPlastic->specularColor.setValue(0.5,  0.5,  0.5);
redPlastic->shininess = 0.5;

// Set all field values for the hydrogen atoms
hydrogenXform1->scaleFactor.setValue(0.75,  0.75,  0.75);
hydrogenXform1->translation.setValue(0.0,  -1.2,  0.0);
hydrogenXform2->translation.setValue(1.1852,  1.3877,  0.0);
whitePlastic->ambientColor.setValue(1.0,  1.0,  1.0);
whitePlastic->diffuseColor.setValue(1.0,  1.0,  1.0);
whitePlastic->specularColor.setValue(0.5,  0.5,  0.5);
whitePlastic->shininess = 0.5;

// Create a hierarchy
waterMolecule->addChild(oxygen);
waterMolecule->addChild(hydrogen1);
waterMolecule->addChild(hydrogen2);

oxygen->addChild(redPlastic);
oxygen->addChild(sphere1);
hydrogen1->addChild(hydrogenXform1);
hydrogen1->addChild(whitePlastic);
hydrogen1->addChild(sphere2);
hydrogen2->addChild(hydrogenXform2);
hydrogen2->addChild(sphere3);
```

隔离节点（Separators）

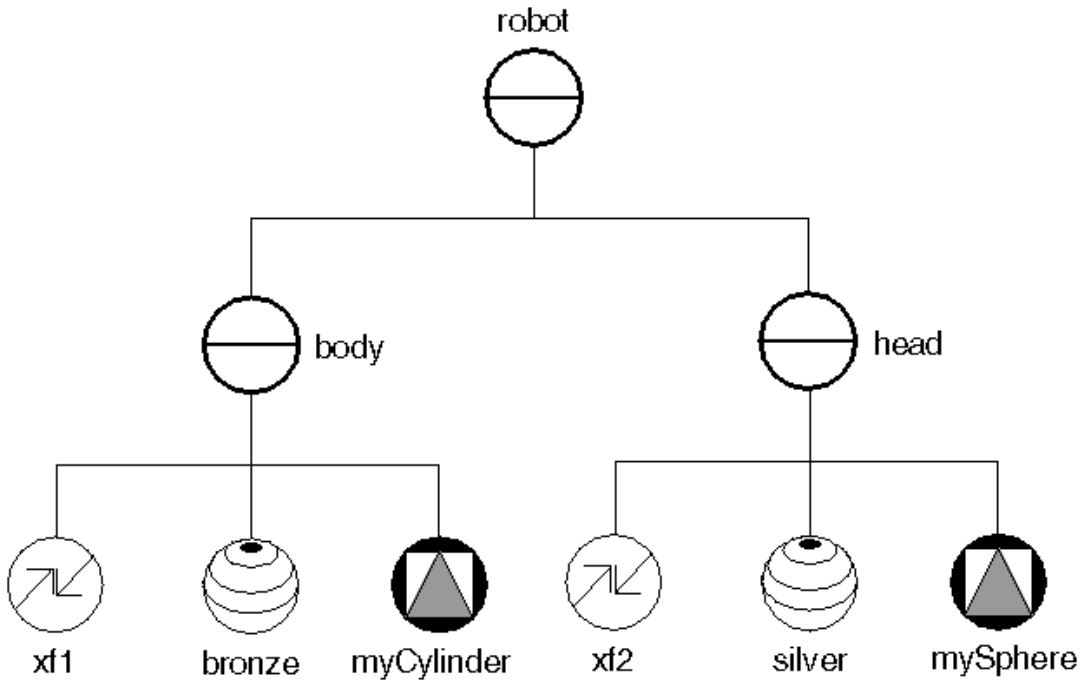
使用从**SoGroup**派生出来的子类**SoSeparator**节点，可以隔离其子节点所产生的影响。**SoSeparator**节点在遍历其子节点之前，首先会保存当前的遍历状态，当遍历完所有的子节点后，**SoSeparator**会恢复以前的遍历状态。因此，位于**SoSeparator**中的节点将不会对位于其之上或之右的任何节点产生影响。

例如，图 3-7 显示了一个机器人头部与身体的场景。*body* 组节点是一个隔离节点，它包含的 **SoTransform** 和 **SoMaterial** 两个节点影响了圆柱节点所要使用的遍历状态。当遍历完位于 *body* 组中的所有子节点后，隔离节点将恢复原来的遍历状态。这样，*head* 组就不会受到 *body* 组中子节点的影响。因为 *head* 组也是一个隔离节点，所以遍历状态在遍历开始的时候再次被保存起来，在遍历结束后被自动恢复。

隔离节点的使用代价非常低，对构建场景有很大的帮助。我们将会经常使用到它。

提示：在连续渲染的时候，如果希望重新使用遍历状态的话，那么场景的根节点应该是一个隔离节点。

图 3-7 隔离组



下面给出了创建机器人身体和头部的代码：

```
// create body parts
SoTransform *xf1 = new SoTransform;
xf1->translation.setValue(0.0,  3.0,  0.0);

SoMaterial *bronze = new SoMaterial;
bronze->ambientColor.setValue(.33,  .22,  .27);
bronze->diffuseColor.setValue(.78,  .57,  .11);
bronze->specularColor。setValue(.99,  .94,  .81);
bronze->shininess = .28;

SoCylinder *myCylinder = new SoCylinder;
myCylinder->radius = 2.5;
myCylinder->height = 6;

// construct body out of parts
SoSeparator *body = new SoSeparator;
body->addChild(xf1);
body->addChild(bronze);
body->addChild(myCylinder);

// create head parts
SoTransform *xf2 = new SoTransform;
xf2->translation.setValue(0,  7.5,  0);
xf2->scaleFactor.setValue(1.5,  1.5,  1.5);

SoMaterial *silver = new SoMaterial;
silver->ambientColor.setValue(.2,  .2,  .2);
silver->diffuseColor.setValue(.6,  .6,  .6);
silver->specularColor.setValue(.5,  .5,  .5);
silver->shininess = .5;

SoSphere *mySphere = new SoSphere;

// construct head out of parts
SoSeparator *head = new SoSeparator;
head->addChild(xf2);
head->addChild(silver);
head->addChild(mySphere);

// add head and body
SoSeparator *robot = new SoSeparator;
robot->addChild(body);
```

```
robot->addChild(head);
```

SoGroup 的其它子类

除了 **SoSeparator** 之外，**SoGroup** 还包括下列的子类：

- SoSwitch
- SoLevelOfDetail
- SoSelection (见第 10 章，“处理事件和选择器”)

在机器人的例子中，**SoSeparator** 节点将节点的影响隔离在一个特定的组中；因为我们不希望让“头部”节点去继承“身体”节点的几何变换和材质等属性。相反，在分子的例子中使用了 **SoGroup** 节点，它是累积了一组属性数据后，将累积后的状态应用到后面的节点上。

切换节点（SoSwitch）

切换节点很像 **SoGroup** 节点，除了遍历子节时，它只访问其中的某个子节点之外。它包含一个叫做 **whichChild** 域，这个域用来指定要遍历的子节点的索引值。例如，下面的代码指定了要遍历访问切换节点的 c 子节点。

```
SoSwitch *s = new SoSwitch;
s->addChild(a);  // this child has an index of 0
s->addChild(b);  // this child has an index of 1
s->addChild(c);  // this child has an index of 2
s->addChild(d);  // this child has an index of 3
s->whichChild = 2;
```

whichChild 缺省值是 SO_SWITCH_NONE，表示不遍历组中任何子节点。

可以通过使用 **SoSwitch** 节点切换若干不同的照相机节点，以此达到以不同的方式来观察场景的目的。也可以使用 **SoSwitch** 节点来创建一种简单原始的动画效果（rudimentary animation）。例如，我们可以通过循环操作一连串的组节点，使鸭子上下扇动翅膀。或者可以让机器人在屏幕上行走。**SoBlinker** 节点是从 **SoSwitch** 派生来的，它可以循环操作其子节点，并还提供了一些对动画显示非常有用的附加功能。(见 13 章“引擎”)

SoLevelOfDetail

SoLevelOfDetail 节点可以对相同的物体指定不同的细节变化程度。它的子节点是按照从高到低的细节程度进行安排的。投影到视口中的物体尺寸决定着那些子节点可以被真正渲染。这个节点对于那些需要快速渲染的应用程序是很有帮助的。**(当物体投影后的尺寸变得非常小时，可以认为人的肉眼很难分辨了，所以可以不必渲染它，进而可以提高渲染速度。译者注)**。

它含有一个域：

screenArea　　屏幕上的面积，用于和level-of-detail 组节点的包围盒进行比较。缺省

(SoMFFloat)　　值为 0.0。表示将只渲染组中的第一个子节点。

为了决定遍历渲染那个子节点，Inventor 首先要计算 SoLevelOfDetail 组中所有子节点的包围盒数据，然后将包围盒投影到视口上，接着再计算包围这个包围盒且和屏幕方向对齐的 2D 矩形的面积。最后将这个面积与保存在 **screenArea** 域中的面积进行比较。例如，图 3-8 给出了一个带有 3 个子节点的 SoLevelOfDetail 组节点。假设 **screenArea** 域保存有[400.0, 100.0]两个数据。如果 3D 包围盒投影面积是 390.0 平方点（square pixels）(即小余 400.0，但大于 100.0)，那么将只显示 *childB* 节点。如果投影面积是 450.0(既大于 400.0)，那么将只显示 *childA*。如果投影面积小余 100.0，那么将只显示 *childC*。

SoComplexity 节点(将在第 5 章“形体，属性和绑定”中讨论)也会对 SoLevelOfDetail 的子节点选择产生影响。如果复杂度是 0.0 或者是 BOUNDING_BOX 类型，那么位于 **SoLevelOfDetail** 中最后的那个子节点将始终会被访问到。如果复杂度是 1.0，则总是使用第一个子节点。如果复杂度是大于 0.0 而小余 0.5，那么包围矩形面积将适当的按比例缩小，以便使用较低的细节度来显示场景。如果复杂度是大于 0.5，则矩形面积将适当的按比例放大。如果复杂度等于 0.5，则 Inventor 将使用原来计算好的包围盒矩形面积。

图 3-9 演示了一个使用不同显示细节的对象模型。每组烛台模型是按照最高细节的模型位于左边，中等细节的位于中间，最差细节的位于右边的方式进行排列的。当烛台靠近照相机时(如图 3-9 中左边的第一组)，将使用最高显示细节的模型，这个模型在烛台的基座上使用了贴图纹理并且显示了灯芯。当模型远离照相机时，使用最差显示细节的模型，这是因为细节已经是不可见的了。当模型位于中间位置时(图 3-9 中中间组)，将使用具有中等显示细节的模型。

图 3-8 带有 Level-of-Detail 节点的场景图

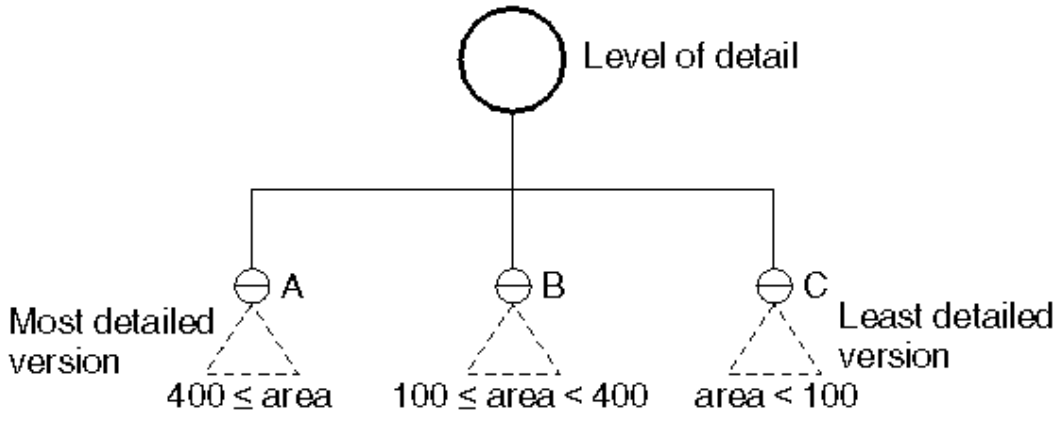
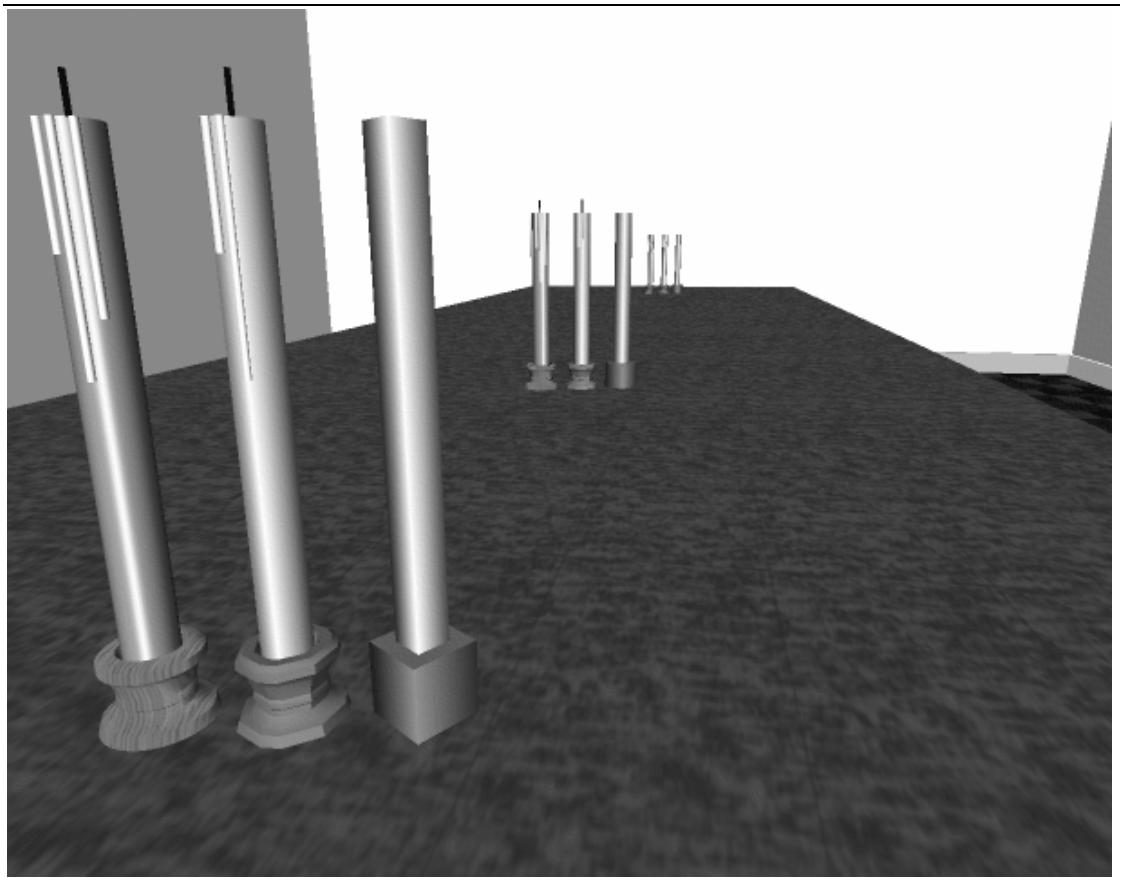


图 3-9 同一物体不同的显示细节度



共享节点实例

我们可以将任意一个节点增加到多个组节点中。例如，一个自行车模型可以使用同一个车轮组节点来代表前后两个车轮，只需要稍加修改这两个车轮的大小和位置即可。术语共享实例（*shared instancing*）就是那种一个单一节点有多个父节点的情况。

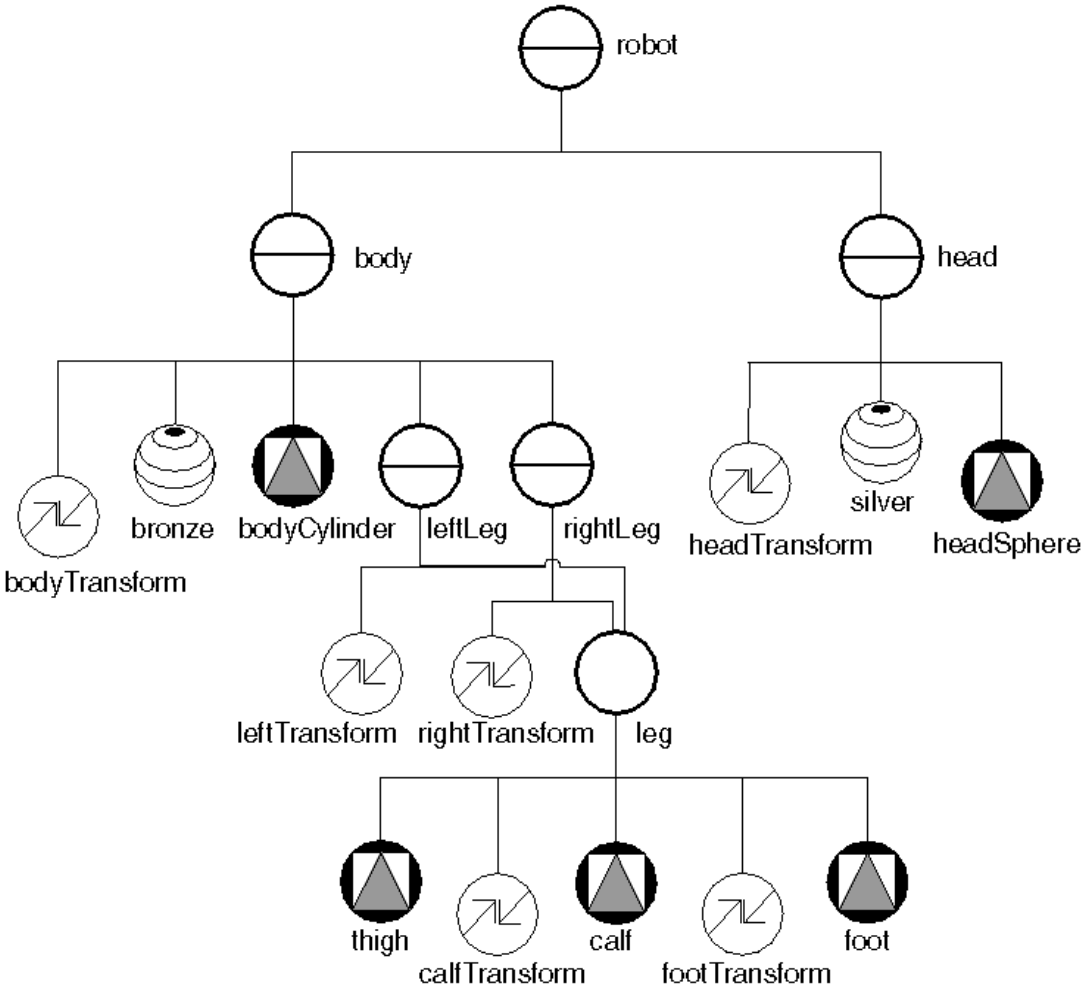
如图 3-10 所示，机器人的左右腿模型共享使用了 *leg* 组节点。*leg* 组包含有一个圆柱(大腿)，一个经过平移的圆柱(小腿)，和另外一个经过平移的立方体(脚)。左右腿组节点(即 *rightLeg* 和 *leftLeg*)中都包含有一个额外的 **SoTransform** 节点。这个节点将整个腿定位到机器人身体的正确位置上。

对于 *leg* 组内的任何修改都将影响到它的所有实例。例如，如果将 *foot* 节点的立方体高度放大两倍的话，那么左右脚的高度将都被放大两倍。

共享实例对于数据库和程序而言是非常经济实惠的，因为对象数据是重用的而非是复制了一份。如有可能的话，应该尽可能地重用节点(和组)以节约程序所需的时间和内存资源。

然而，注意不能在场景中创建一个循环节点。一个节点可以连接到多个父节点中，但不能作为自己或任何其后代节点的子节点。（Do not, however, create cycles within a given scene graph. A node can connect to multiple parents but should not be a child of itself or any of its descendants）

图 3-10　Leg 组的共享实例场景图



例 3-2 给出了上面描述的机器人的代码。渲染图显示在图 3-11 中。

例 3-2 Robot.c++

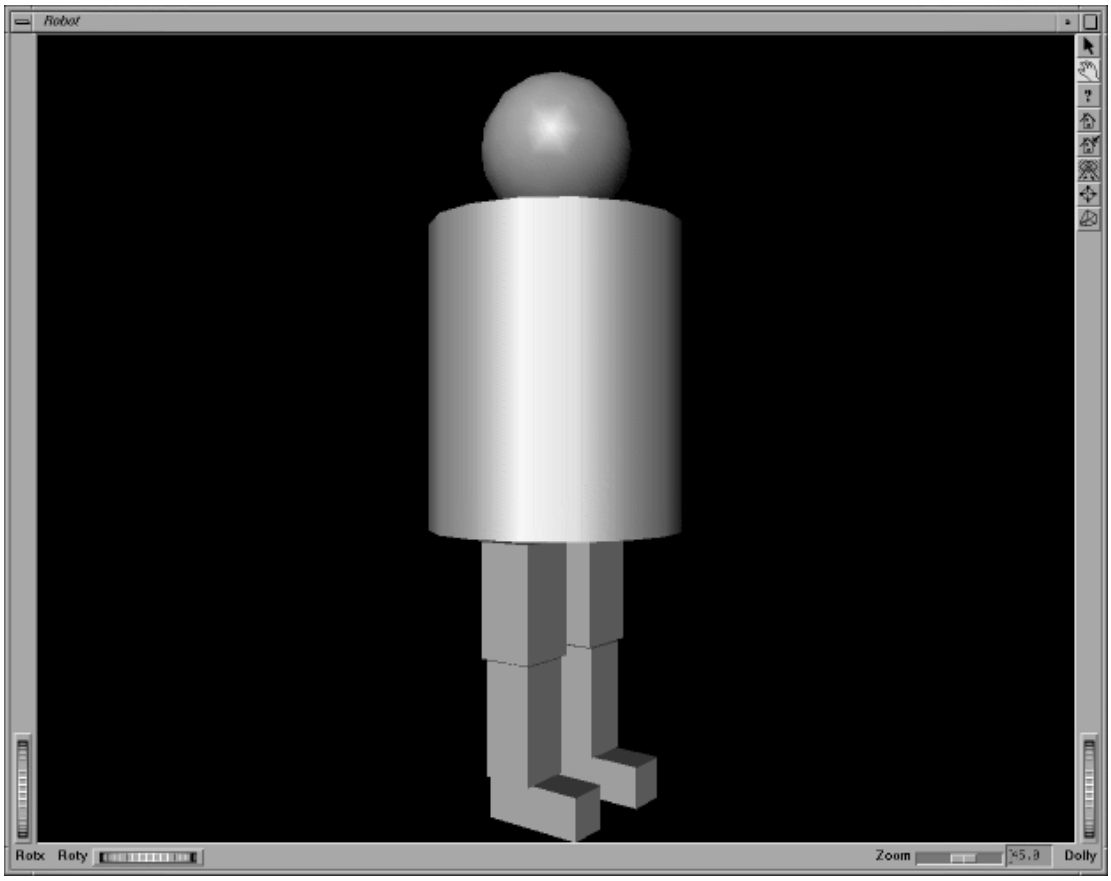
```
// Robot with legs

// Construct parts for legs (thigh,  calf and foot)
SoCube *thigh = new SoCube;
thigh->width = 1.2;
thigh->height = 2.2;
thigh->depth = 1.1;

SoTransform *calfTransform = new SoTransform;
calfTransform->translation.setValue(0,  -2.25,  0.0);

SoCube *calf = new SoCube;
calf->width = 1;
```

```
calf->height = 2.2;
calf->depth = 1;

SoTransform *footTransform = new SoTransform;
footTransform->translation.setValue(0,  -2,  .5);

SoCube *foot = new SoCube;
foot->width = 0.8;
foot->height = 0.8;
foot->depth = 2;

// Put leg parts together
SoGroup *leg = new SoGroup;
leg->addChild(thigh);
leg->addChild(calfTransform);
leg->addChild(calf);
leg->addChild(footTransform);
leg->addChild(foot);

SoTransform *leftTransform = new SoTransform;
leftTransform->translation = SbVec3f(1,  -4.25,  0);

// Left leg
SoSeparator *leftLeg = new SoSeparator;
leftLeg->addChild(leftTransform);
leftLeg->addChild(leg);

SoTransform *rightTransform = new SoTransform;
rightTransform->translation.setValue(-1,  -4.25,  0);

// Right leg
SoSeparator *rightLeg = new SoSeparator;
rightLeg->addChild(rightTransform);
rightLeg->addChild(leg);

// Parts for body
SoTransform *bodyTransform = new SoTransform;
bodyTransform->translation.setValue(0.0,  3.0,  0.0);

SoMaterial *bronze = new SoMaterial;
bronze->ambientColor.setValue(.33,  .22,  .27);
bronze->diffuseColor.setValue(.78,  .57,  .11);
bronze->specularColor.setValue(.99,  .94,  .81);
```

```
bronze->shininess = .28;

SoCylinder *bodyCylinder = new SoCylinder;
bodyCylinder->radius = 2.5;
bodyCylinder->height = 6;

// Construct body out of parts
SoSeparator *body = new SoSeparator;
body->addChild(bodyTransform);
body->addChild(bronze);
body->addChild(bodyCylinder);
body->addChild(leftLeg);
body->addChild(rightLeg);

// Head parts
SoTransform *headTransform = new SoTransform;
headTransform->translation.setValue(0,  7.5,  0);
headTransform->scaleFactor.setValue(1.5,  1.5,  1.5);

SoMaterial *silver = new SoMaterial;
silver->ambientColor.setValue(.2,  .2,  .2);
silver->diffuseColor.setValue(.6,  .6,  .6);
silver->specularColor.setValue(.5,  .5,  .5);
silver->shininess = .5;

SoSphere *headSphere = new SoSphere;

// Construct head
SoSeparator *head = new SoSeparator;
head->addChild(headTransform);
head->addChild(silver);
head->addChild(headSphere);

// Robot is just head and body
SoSeparator *robot = new SoSeparator;
robot->addChild(body);
robot->addChild(head);
```

提示：当要构建一个复杂的图形场景时，我们也许希望使用 Inventor 文件格式（见第 11 章“文件格式”）来定义场景，然后从文件或内存字符串中读取场景。这种方法更简单，且比使用编程方式更能减少发生错误的可能。

图 3-11 机器人的渲染图

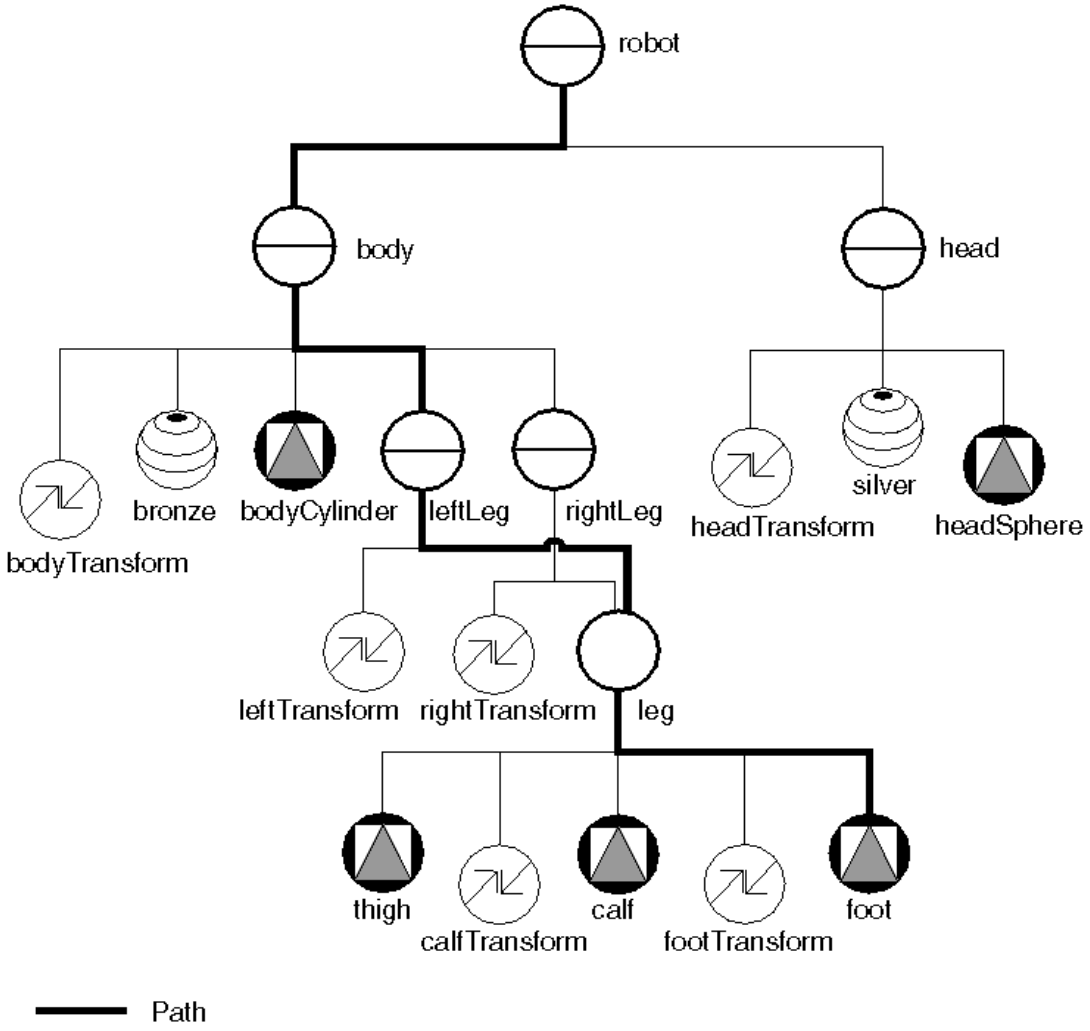


路径（Paths）

路径用于区分场景中某个特定对象。假设，我们想使用机器人的左脚。在图 3-10 中那个节点能代表左脚呢？我们不能简单地认为是 *foot* 节点，因为这个节点(*foot*)是被左腿和右腿同时使用的。答案是使用路径（或称为链）来表示左脚。路径起始于 *robot* 节点(根节点)，沿着图形向下直至通到 *foot* 节点。图 3-12 指出了表示左脚的路径。

路径包含有一个节点链的引用(references)。其中每个节点都是前一个节点的子节点。一个路径可以表示一个完整场景或子场景（一个场景的某个部分）。在本书的所有场景图表中，使用一条连接节点链的粗线段来表示路径。

图 3-12 表示左脚的路径



如何得到路径(Where Do Paths Come From) ?

可以通过拾取和搜索动作返回一条路径。我们也可以手工构造自己路径。(第 9 章 “应用动作”将详细讨论交互式拾取)。一个交互式应用程序的用户可以在屏幕的某个物体上点击鼠标，导致程序拾取了这个物体，然后就可以对这个物体执行一个操作－例如移动它、修改它的颜色、或者删除它。选择器节点管理着一个路径列表作为当前已经被选中的节点。

路径的用途

所有对节点可以操作的动作都同样可以被应用到路径上。这些动作包括：计算路径的包围盒、累加路径的几何变换矩阵、和将路径写入文件中。

我们怎样使用包含在路径中的信息依赖于我们编写的程序。我们可以使用整个路径，也可以使用部分路径。如果用户在机器人的左脚上点击了鼠标，那么用户是想选择整个机器人呢，还是想选择左腿，或者仅仅是想选择左脚呢？(也许用户第一次点击使，想选择整个机器人，接着再点击，将选择机器人的下级部分，例如左腿或者左脚)。

节点中的域 (Fields)

当创建完一个节点后，节点中的域就已经被设置上了预定义的数值。然后，我们就可以直接修改域中的数值了。设置域值的语法主要依赖于域的类型以及域是单值域还是多值域之分。下面的例子代码创建了一个绘制风格节点，并设置了其域值：

```
SoDrawStyle *d = new SoDrawStyle;
d->style.setValue(SoDrawStyle::LINES) ;
d->lineWidth.setValue(3) ;
d->linePattern.setValue(0xf0f0);
```

在上面的例子中，当前的绘制风格将变成非填充、虚线型、线宽是 3 个象素。如果没有显式地设置域值的话，Inventor 将对节点使用缺省值。下面列出的就是 **SoDrawStyle** 节点的缺省值：

域	缺省值
style	SoDrawStyle::FILLED
lineWidth	1
linePattern	0xffff (solid)
pointSize	1

下面的章节将讨论如何设置和读取不同类型的域值。第 13 章“引擎”，将讨论一种特殊类型的域—全局域(*global* fields)和开关域(*trigger* fields)的连接关系。

为什么使用域(高级内容)？

为什么 Inventor 的节点使用域而不是使用简单的成员变量？。本节将概要讨论一下域的运行机制。 *The Inventor Toolmaker* 将提供关于这个话题的更详细的背景知识。

首先，正如下面章节和 *Open Inventor C++ Reference Manual* 书中描述的那样，域为设置和查询数据提供了统一的方法。第二，域为 Inventor 探测场景数据库的变化提供一种机制。第三，可以将一个节点的域连接到另外一个节点的域上，在第 13 章“引擎”将做详细讨论。最后，域提供了向文件读写节点数据的统一的方法。

单值域对多值域

单值域的内部只有一个给定类型的变量值，在它们的类名称中都包含有 **SF** 字符。例如：

SoSFBool	包含一个 SbBool
SoSFFloat	包含一个单一的 float
SoSFRotation	包含一个 SbRotation
SoSFName	包含一个 SbName

SoSFColor　　包含一个单一的 SbColor

单值域通常用于那些不需要数组的节点，如线型、平移值、旋转值、照相机屏幕的高宽比等等。

多值域内含有一个数组。在类名称中都包含有 **MF** 字符—例如 **SoMFBool**、**SoMFFloat**、**SoMFVec3f** 和 **SoMFColor**。多值域通常用于保存坐标点和法向矢量这些数据。多值域也可以用于保存材质数据，这样我们就可以为不同的顶点设置不同的颜色。大多数的域同时都具有 **SF** 和 **MF** 形式。在 *Open Inventor C++ Reference Manual* 书中，对每个节点中的域都做了详细的描述。

单值域：设置和查询数据

在本章中，先前的例子演示了如何声明和创建节点。本节将通过另外的例子来演示如何设置和查询节点单值域的数据。(大多数的域都带有 **setValue()** 和 **getValue()**方法。而且也可以同时使用 = 操作符来设置数据)

Floats、Longs、和 Shorts

第一个例子将通过使用 **setValue()**方法来设置 **SoOrthographicCamera** 节点的 **height** 域值。这个域的类型是 **SoSFFloat**。

```
SoOrthographicCamera *cam = new SoOrthographicCamera;
cam->height.setValue(1);
```

或

```
cam->height = 1.; // = operator has been defined for this field
```

使用 getValue()方法可以查询这个域的数据。

```
float result = cam->height.getValue();
```

向量(Vectors)：

可以使用多种不同的格式来设置 **SoSFVec3f** 的域。每种格式都会定义一个 3D 向量：

- 通过一个向量(**SbVec3f**)来设置。
- 通过 3 个浮点数来设置（既可以是一个向量，也可以是三个单独的浮点数）。
- 通过一个 3 维浮点数数组来设置。

下面的例子将演示如何设置 **SoSFVec3f** 的域值。**SoTransform** 节点有一个叫做 **translation** 的域，这是一个 **SoSFVec3f** 类型的域，它包含一个 **SbVec3f** 类型的变量。变量

xform 是一个 **SoTransform** 节点的实例。

```
SoTransform *xform = new SoTransform;

//(1) Setting the field from a vector

        SbVec3f vector;
        Vector.setValue(2.5,  3.5,  0.0);
xform->translation.setValue(vector);
        // or:  xform->translation = vector;


//(2a) Setting the field from a vector of three floats

        xform->translation。setValue(SbVec3f(2.5,  3.5,  0.0));
        // or:  xform->translation = SbVec3f(2.5,  3.5,  0.0);


//(2b) Setting the field from three floats

        float x = 2.5,  y = 3.5,  z = 0.0;
xform->translation.setValue(x,  y,  z);


        //(3) Setting the field from an array of three floats

        float floatArray[3];
floatArray[0] = 2.5;
floatArray[1] = 3.5;
floatArray[2] = 0.0;
xform->translation.setValue(floatArray);
```

使用 **getValue()**方法可以查询此域值，下面的例子复制了一个向量，然后修改这个向量，最后在复制回去：

```
SbVec3f t = xform->translation.getValue();

t[0] += 1.0;
xform->translation.setValue(t);
// or:  xform->translation = t;
```

旋转

旋转域可以指定一个在 3D 空间中的旋转量。因为 **SbRotation** 代表着绕一个轴转一个角度的旋转量，所以我们可以通过指定轴和角度来设置这个域值：

```
SbRotation r;
SbVec3f axis(0.,  1.,  0.);
float angle = M_PI; //from math.h
r. setvalue(axis,  angle);
// or SbRotation r(SbVec3f(0.,  1.,  1.),  M_PI);
```

也可以定义一个从一个方向向量转到另一个方向向量的旋转。

```
SbRotation r(SbVec3f(0.0,  0.0,  1.0),  SbVec3f(0.0,  1.0,  0.0));
```

设置 **SoTransform** 节点中的 **rotation** 域值。

```
SoTransform *xform = new SoTransform;
xform ->rotation = r;
```

我们也可以使用 **setValue()**方法，通过向参数传入旋转轴和角度、一组四元数、或者两个向量来设置 **rotation** 域的数值。

=(赋值)操作符可以将一个域值赋值到另外一个相同类型的域值上。和使用向量一样，**getValue()**用于返回域值。

提示：如果想用一个旋转轴和角度数据来定义旋转的话，必须使用一个 **SbVec3f** 和一个浮点数作为输入参数。如果想用一个四元数来定义旋转的话，必须使用 4 个浮点数作为输入参数。

多值域：设置和查询数据

SoMaterial 节点包含有下列域：

域名	类
ambientColor	SoMFColor
diffuseColor	SoMFColor
specularColor	SoMFColor
emissiveColor	SoMFColor
shininess	SoMFFloat

transparency SoMFFloat

下面的例子将演示设置 **SoMaterial** 节点域值的多种不同方式。其中 **transparency** 域是 **SoMFFloat** 型的，所以它可以包含一个或多个浮点类型的数值。**diffuseColor** 域是 **SoMFColor** 型的，所以它可以包含一个或多个 **SbColor** 类型的数值。在 **SoMFFloat** 域中设置多个数值的语法如下：

```
nodeName->fieldname.setValues(starting index, number of values, pointer to array of values);
```

例如：

```
SoMaterial *mtl;
float vals[3];
vals[0] = 0.2;
vals[1] = 0.5;
vals[2] = 0.9;
mtl->transparency.setValues(0, 3, vals);
```

数组的空间在必要的时候可以被从新分配。数据将从数组中拷贝到域中。**(注意，这里提示了 Inventor 的一个重要规则，即 Inventor 是将应用程序中的数据保存在自己分配的内存中。这样做的优点是，保证了 Inventor 和应用程序的数据分离，应用程序的数据不会对 Inventor 产生影响。但是使用这种方式，程序可能会需要两份内存，因此会浪费内存空间和 CPU 时间。目前在最新的 Inventor 软件实现版本中，对域增加了一种新的赋值方式“引用赋值”，这和 C++中的引用赋值的概念是一样的，Inventor 在内部只保存应用程序数据的指针。译者注)**

下面的例子是使用 **Sb** 类型的变量来设置多值域：

```
SoMaterial *mtl;
SbVec3f vals[3];

vals[0].setValue(1.0, 0.0, 0.0);
vals[1].setValue(0.0, 1.0, 0.0);
vals[2].setValue(0.0, 0.0, 1.0);
mtl->diffuseColor.setValues(0, 3, vals);
```

如果只想在 **SoMFFloat** 域中设置一个数值的话，可以使用下列快捷方式：

```
nodeName->fieldname.setValue(value1);
```

例如：

```
mtl->transparency.setValue(.25);
//or mtl->transparency = .25;
```

快捷方式只设置一个数值，且将这个数值设置到内部数组中的第一位置上。注意，这样做丢弃掉原来数组中先前设置的所有数据(**即从数组的第二个位置到最后位置的所有数据将被删除掉。译者注**)，所以只有要对域设置一个数据的情况下，我们才应该使用这个方式。如果只想修改数组中的一个数据并且还要保护好其他数据的话，可以使用 **setValues()**或 **set1Value()** 方法。

我们可以按照下列方式，使用[]操作符来查询多值域中的特定数据：

```
f = myMtl->transparency[13]; // get 14th value of array
```

也可以使用循环语句来使用域中的所有数据：

```
for (i = 0; i < myMtl->transparency.getNum(); i++) {
      printf("transparency value %d is %g\n",  i,
               myMtl->transparency[i]);
}
```

将一个数据插到域的中间位置上：

```
float newValues[2];
newValues[0] = 0.1;
newValues[1] = 0.2;

// First,  make space; after this,  myMtl->transparency[10]
// and myMtl->transparency[11] will have arbitrary values:

myMtl->transparency.insertSpace(10,  2);

// Set the space created to the right values:

myMtl->transparency.setValues(10,  2,  newValues);
```

从域中删除数据：

```
// Delete myMtl->transparency[8] and myMtl->transparency[9];
// the values in myMtl->transparency[10] on up will be moved
// down to fill in the missing space,  and the transparency
// array will have two fewer values。
```

```
myMtl->transparency.deleteValues(8,  2);
```

忽略标记(Ignore Flag) (高级内容)

每个域都带有一个与之相关联的忽略标记。**setIgnored()**方法用来设置和复位忽略标记。当设置了域的忽略标记后，这个域的数据将被忽略掉。忽略标记允许只忽略节点中的某些域，而同时还可以使用其他的域。例如，可以忽略材质节点中的镜面反射颜色域，这样这个域值将继承前面材质节点中的镜面反射颜色域数据。

```
SoMaterial *bronze = new SoMaterial;

bronze->ambientColor.setValue(.33,  .22,  .27);
bronze->diffuseColor.setValue(.78,  .57,  .11);
bronze->specularColor.setIgnored(TRUE);
bronze->shininess = .28;
```

关闭忽略标记：

```
bronze->specularColor.setIgnored(FALSE);
```

如果某个域设置了忽略标记，方法 **isIgnored()**将返回 TRUE。

```
if (bronze->specularColor.isIgnored()) {
   printf("Yes,  specular is ignored\n");
}
```

有些域值是不能从前面的域中继承过来的，因此忽略标记对这些域没有作用。例如，形体节点、灯光节点、一些组节点、照相机节点、以及在 **SoEnvironment** 节点中的这些域都不能被继承。如果我们对这些域设置了忽略标记，Inventor 将会简单地使用这些域的缺省值。

重置(Override)标记 (高级内容)

每个域都有一个与之相关联的重置标记。重置标记是一种强大的面向对象机制，通常用于靠近顶端场景的节点。当设置重置标记之后，在场景中之后遇到的所有相同类型的节点都将被忽略掉，即使那些节点本身也设置了重置标记。例如，我们可以在图形场景的顶端插入一个属性为“线型”的 **SoDrawStyle** 节点，对这个节点设置上重置标记，这样不管以后相同的 **SoDrawStyle** 节点如何设置，我们可以保证整个场景中都使用框架线来绘制物体。**setOverride()**方法可以设置或复位重置标记。方法 **isOverride()**可以返回重置标记的状态。

例如：

```
// This function toggles the given draw-style node between
// overriding any other draw-style nodes below it in the scene
// graph,  and not having any effect at all on the scene graph。
//
void
toggleWireframe(SoDrawStyle *myDrawStyle)
{
   if (myDrawStyle->isOverride()) {
      myDrawStyle->style.setIgnored(TRUE);
      myDrawStyle->setOverride(FALSE);
   } else {
      myDrawStyle->style = SoDrawStyle: : LINES;
      myDrawStyle->style.setIgnored(FALSE);
      myDrawStyle->setOverride(TRUE);
   }
}
```

重置标记一般不会用在模型场景上，主要用在程序中想临时修改整个场景的地方。

注意：重置标记是不会被写进文件中的(见 11 章“文件格式”)

如果对一个域值不能被继承的域设置重置标记(例如，对一个半径为 7 的球上设置重置标记)，重置标记不会对场景中那个类型的节点产生任何影响。

引用和删除

虽然节点是按照 C++方式创建的，但是在删除节点时却是有别于 C++方式。下面将讨论节点将如何计算针对自己的引用，以及引用数何时增加与减少。还要概要描述节点解除引用(unreferencing)的正确步骤，解除引用将导致节点的删除。

引用计数

每个节点都在数据库中保存有一个引用计数。节点带有几种不同引用类型：

- 父－子 引用
- 路径－节点 引用

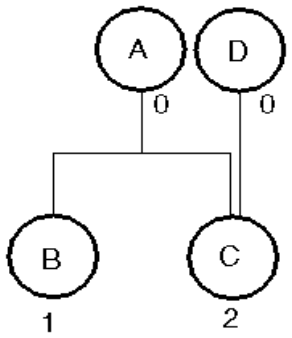
引擎也保存有引用计数(见第 13 章“引擎”)。当引擎的输出连接上一个域之后，其引用计数就加 1。我们也可以通过调用 **ref()** 或 **unref()** 函数来手工增加或减少引用计数。

图 3-13 给出了在一个小的子场景中节点的引用计数值。无论何时，只要我们对节点产生了一个引用，节点就会增加它的引用计数。

```
A->addChild(B)
```

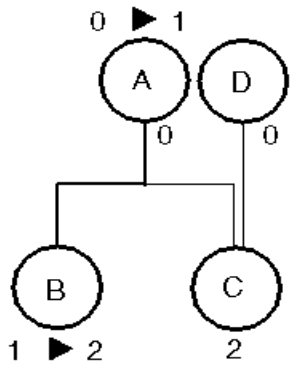
将 B 节点增加到 A 节点中的这个行为也会导致 B 节点的引用计数加 1。在图 3-13 中，节点 C 的引用计数是 2，因为它被加到了两个不同的父节点中。此时，A 和 D 节点的引用计数是 0。(因为它们没有被其他节点所使用。译者注)

图 3-13 引用计数



如图 3-14 所示，引用一个路径中的节点也会增加这个节点的引用计数。节点 A 的引用计数现在变成了 1，而节点 B 的引用计数变成为 2。

图 3-14 增加引用计数



提示： 一定要保证引用了图形场景中的根节点：root->ref()，因为根节点不能作为其他节点的子节点，所以根节点不能被其他的节点所引用。

如何删除节点

Inventor 使用引用计数的机制来删除节点和其子场景。为了理解节点是如何被删除的，我们首先需要知道节点的引用计数是如何被增加与减少的。

当删除节点的一个引用时，会减少节点的引用计数值。移去一个子节点时，也会减少子

节点的引用计数值。 当一个节点的引用计数值回到 0 时，这个节点就会从数据库中被删除掉。然而在下面的情况中，删除一个节点可能会引起一些问题。(见图 3-13)

问题 1： 如果从节点 A 中去掉节点 B，那么节点 B 的引用计数就变成了 0，这样 B 节点也就被删除了。但是，如果我们还想再使用节点 B 该怎么办呢?

问题 2： 怎样删除节点 A?，它的引用计数已经是 0 了。

问题 3： 如果对一个引用计数为 0 的节点(例如节点 A)应用了一次动作（action），动作创建了一个引用此节点的路径(**这时此节点的引用计数变成了 1。译者注**)，当动作结束后，路径被删除(**这时此节点的引用计数由 1 变成了 0。译者注**)，同时节点也被删除了。

解决上述问题的方案是：当想防止一个节点被删除的话，我们就显式地引用它：

```
B->ref();
```

引用一次节点就对其引用计数增加 1，这样就可以保证节点不会被意外地删除掉。当我们显式地引用了节点 B 之后，我们就可以安全地从节点 A 中去除掉节点 B，而不必担心删除节点 B 了。(解决了问题 1)

同样的，为了防止节点 A 被删除(问题 3)，我们也可以显式引用它：

```
A->ref();
```

如果想删除 A(问题 2)，我们可以对它解除引用，这将减少它的引用计数。因为只引用了一次节点 A，所以现在节点 A 就被删除了。(**在前面调用了 A->ref()，所以 A 的引用计数是 1，当调用 A->unref()后，A 的引用计数就变成了 0，最终导致了 A 被删除。译者注**)

```
A->unref();
```

当删除一个组节点时，组节点中首先要移走所有的子节点，这将导致所有子节点的引用计数都被减 1。 例如，在下面的图 3-15 中，如果调用了：

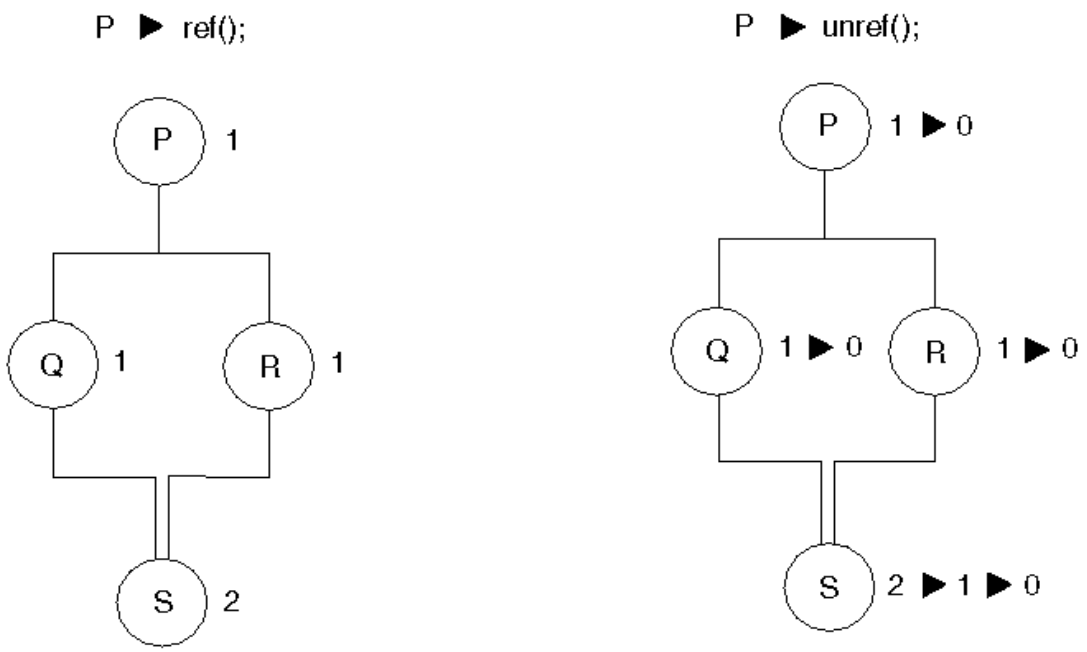
```
P->unref(); // reference count for P goes to 0
```

那么所有子节点的引用计数将按照下列方式减少：

1. Q 变成 0
2. S 变成 1
3. R 变成 0
4. S 变成 0

因为所有的引用计数都变成了 0，所以所有的节点就都被删除了。

图 3-15。 减少引用计数



提示：不要分配节点数组、路径数组、或引擎数组。因为当数组的某个节点的引用计数减为 0 时，Inventor 将试图释放掉这个节点在数组中所分配中内存。这将产生内存使用方面的问题。

当我们对一个节点应用动作时，这个动作将自动创建一个引用此节点的路径。当动作结束后，动作将会自动删除那个路径。因此相应地节点的引用计数也会减少。在这里，我们再次提醒读者，如果一个节点原来的引用计数为 0 的话，那么它将在动作结束后被删除。

提示：对一个已经被删除的节点进行“解除引用”操作，经常会引起随机内存错误的发生。如果发生了这样的错误，请首先检查程序，确保没有试图使用已经被删除的节点。Inventor 库的调试版本可以捕捉到许多通常的引用计数错误。

引用计数为 0 的节点

节点、路径、引擎这些对象都只能使用 **new** 来创建，绝对不能在“栈”中声明定义它们。这些对象只能是在它们的引用计数变成 0 的时候才可以被释放掉，而不能是在超出了它们的生存期范围的时候被删除。

一个新创建节点的引用计数为 0。这并不意味这它马上就会被删除，因为 Inventor 只有

在节点的引用计数***减少到 0(decremented to 0)***的时候，才会删除这个节点。有时候对于能够恢复节点的初始状态(即，引用计数减少到 0，但它依然存在)是非常重要的。例如：

```
// Create a sphere of a certain radius and returns its bounding
// box。 NOTE： BUGGY VERSION; provided for discussion only!

SoSphere *makeSphere(float radius,  SbBox3f &box)
{
   sphere = new SoSphere; // reference count of 0
   sphere->radius.setValue(radius);

   ba = new SoGetBoundingBoxAction;
   ba->apply(sphere);     // does a ref/unref
   box = ba->getBoundingBox();

   return  sphere;         // ERROR! returning node that
               // was deleted when ref count
               // went back to zero!
}
```

在上面的例子中，**SoGetBoundingBoxAction** 对球节点进行引用和解除引用的操作。当解除引用时，球节点的引用计数变成了 0，这样它就被删除了(**所以这个函数返回一个错误的指针，任何使用这个指针的代码都将引起内存异常，译者注**)。球节点在应用动作之前，首先要被引用一次。

如果我们想返回一个处于原始“新鲜”状态的球节点(引用计数为 0，但没有被删除)，可以使用 **unrefNoDelete()**这个方法。下面是使用 **unrefNoDelete()**的例子。

```
// Create a sphere of a certain radius and returns its bounding
// box。 NOTE： CORRECT VERSION

SoSphere *makeSphere(float radius,  SbBox3f &box)
{
   sphere = new SoSphere; // reference count of 0
   sphere->ref();// we want it to stay around
   sphere->radius.setValue(radius);

   ba = new SoGetBoundingBoxAction;
   ba->apply(sphere);     // does a ref/unref
   box = ba->getBoundingBox();

   sphere->unrefNoDelete();        // ref count goes to zero,
         // but sphere stays around
```

```
   return  sphere;        // returns sphere with ref
        // count of zero
}
```

引用和删除的总结

表 3-1 总结了当节点和引擎的引用计数增加和减少的时候所发生的事情。 注意将引擎连接到一个节点的域上时，并不会增加这个节点的引用计数。(引擎将在第 13 章中进行讨论)。

表 3-1 引用和删除

引用计数加 1	引用计数减 1
将一个节点作为子节点增加到另外的一个节点中，将增加这个子节点的引用计数	去除一个节点的子节点，这个子节点将减少引用计数
将节点增加进路径中	从路径中去除节点
对节点或路径应用一个动作后，将使所有被遍历过的节点增加引用计数。	当动作遍历完成后，所有被遍历的节点都将被解除引用
将一个节点增加进 SoNodeList 节点中	从 SoNodeList 节点中除去一个节点
将 SoSFNode 或 SoMFNode 数值指向某个节点，该节点的引用计数将增加。	修改 SoSFNode 或 SoMFNode 中的某个数值，使其指向另外不同的节点或者指向 NULL，或者删除这个数值，那么这个数值原来所指向的节点将减少引用计数。
将一个引擎的输出连接到一个节点的域上，将增加引擎的引用计数	从一个节点域上断开引擎的输出，引擎的引用计数将减少

节点类型

Inventor 通过 **SoType** 类提供了运行时类型检查(runtime type-checking)的功能。一个节点实例可以使用 **getTypeId()**方法得到它的 **SoType** 类。像节点、引擎、动作、细节（details）、和事件这些绝大多数的 Inventor 类都提供了运行时类型检查的功能。

SoType 类可以返回类型的父类(**getParent()**)，可以创建一个特定类型的实例(**createInstance()**)，也可以返回类型的名称(**getName()**)。例如下面的代码将可能返回诸如“Material”或“Group”这样的类型名称：

```
node->getTypeId().getName();
```

下面的两个声明都将返回一个 **SoMaterial** 节点的 **SoType**(第一个更有效率)

```
// (1)
SoMaterial::getClassTypeId();
// (2)
SoType::fromName("Material");
```

可以使用 = 操作符来确定一个实例是否是某种特定类型，如下：

```
if (myNode->getTypeId() == SoGroup::getClassTypeId())
      // Is this an SoGroup?
```

使用 **isOfType()**方法或者 **SoType：：derivedFrom()**方法可以判定一个实例是否和另一个特定类具有相同的类型，或者确定这个实例是否是从另外一个特定类派生出来的。

```
/ (1)
if (myNode->isOfType(SoGroup::getClassTypeId()))
   // Is this an SoGroup,  SoSeparator,  SoSwitch,  and so on

// (2)
if (myNode->getTypeId().isDerivedFrom(
      SoGroup::getClassTypeId()))
```

可以阅读第 9 章的 **SoSearchAction** 描述，这个类可以在图形场景中查找出某一特定类型的节点，或者是派生于某类的节点。

节点命名

我们可以为节点、路径、或引擎赋值一个名字，然后可以根据名字就能查找到这些对象。因为对象在读写文件的时候，其名字都是要保存的，因此名字也是一种标识对象的有效途径。基类 **SoBase** 提供了一个叫做 **setName()**的方法，这个方法允许为一个节点、路径、或引擎指定一个名字。同时基类也提供了 **getName()**方法用于返回对象的名字。

任何节点、路径、或引擎都只能有一个名字，但名字不必是唯一的(**Inventor 允许同一个名字赋值在多个节点、路径或引擎对象上。译者注**)。名字是 **SbName** 类型的。**SbName** 必须以大小写字母(A-Z)或者下划线字母(_)为开头，在 **SbName** 中的所有字符必须是数字 0-9、大小写字母 A-Z、或者下划线。一个对象的缺省名字是空字符串(“ ”)。

通过 **SoNode** 类的方法 **getByName()**，可以根据一个给定的名字找到一个或者多个节点(**SoPath** 和 **SoEngine** 也提供了类似的 **getByName()**方法)。“搜索动作”也能根据一个给定的名字找到一个或者多个节点。

下面是一个如何使用节点名字的实际例子。这是一个轨道车竞速游戏程序(Slot Car Racing)。程序允许用户设计自己的赛车。这个功能首先需要有一些简单的事先约定，如赛车有多大，那个方向是向上的，赛车上必备的节点或引擎是怎样命名的等等。例如，其中的一个约定就是，用于控制方向盘旋转的 **SoTransform** 节点必须命名为 *SteeringWheelRotation*。当程序将一个用户自己设计的赛车模型读入游戏场景后，程序就会搜索名字为的 *SteeringWheelRotation* 节点，找到后，程序就可以使用这个节点来驱动方向盘了。

在例子 3-3 中，将演示使用 **setName()**为几个节点命名，然后使用 **getByName()**来返回一个特定的节点。名字叫做 *MyCube* 的子节点将从名字叫做 *Root* 的父节点中去除掉。

例 3-3 节点命名

```
#include <Inventor/SoDB.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/sensors/SoNodeSensor.h>

void RemoveCube();  // Defined later…

main( int ,  char ** )
{
   SoDB::init();

   // Create some objects and give them names:
   SoSeparator *root = new SoSeparator;
   root->ref();
   root->setName("Root");

   SoCube *myCube = new SoCube;
   root->addChild(myCube);
   myCube->setName("MyCube");
   SoSphere *mySphere = new SoSphere;
   root->addChild(mySphere);
   mySphere->setName("MySphere");

   RemoveCube();
}

void
RemoveCube()
{
   // Remove the cube named 'MyCube' from the separator named
   // 'Root'。  In a real application,  isOfType() would probably
```

```
   // be used to make sure the nodes are of the correct type
   // before doing the cast。

   SoSeparator *myRoot;
   myRoot = (SoSeparator *)SoNode::getByName("Root");

   SoCube *myCube;
   myCube = (SoCube *)SoNode::getByName("MyCube");

   myRoot->removeChild(myCube);
}
```

第四章 照相机和灯光

本章目标

在阅读完本章节后，我们将可以做下列事情：

- 将不同类型的照相机增加到场景中。试验修改照相机的位置、方向、和视区映射（viewport mapping）。
- 将不同类型的灯光加入到场景中，试验修改灯光的类型、强度、和颜色。

从第 4 章到第 8 章, 我们将着重讨论几种不同类型的节点。之所以要首先讨论照相机和灯光节点，是因为如果没有它们的话，我们创建的对象将是不可见的。在接下来的其它章节中，我们还将学习更多其它类型的节点，包括形体、属性、绑定、文字、纹理、和 NURBS 曲线曲面等。我们可以根据自己的兴趣和需要，自由选择阅读那些章节。

使用灯光和照相机

在前面的章节中，我们介绍了组、属性、和形体等节点，并且演示了如何使用这些节点来创建场景。现在，我们将要学习可以影响到 3D 图形场景外观的两个节点类：灯光和照相机。在 Inventor 中，如同在现实世界里那样，灯光提供照明以便我们观察物体。如果在一个场景中没有包含任何灯光，并且当前的光照模型是缺省的 Phong lighting（**一种特定的光照计算公式，是 Bui Tuong Phong 于 1973 年发明的算法。译者注**），那么场景中的物体都将处于黑暗中并且变得不可见。就像在现实世界中有多种光照类型那样－电灯泡、太阳、舞台灯－Inventor 同样也提供了不同类型的灯光供我们在场景中使用。

照相机是我们观察场景的“眼睛”。Inventor 提供了一个和人类眼睛具有相同透视方式的照相机类。同时，Inventor 还另外提供另外一种用于产生场景 2D“快照”的照相机类，这种照相机使用另外一种类型的透视方式。本章将首先讨论照相机，并且假设在场景的最顶端处至少存在一个灯光节点。

提示：观察器组件（Viewer components）将自动创建自己的照相机和灯光。更多内容请阅读第 16 章

照相机

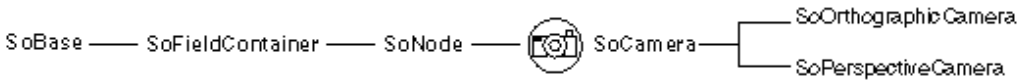
照相机节点可以对场景中所有位于它之后的节点“拍摄”一张照片。因为照相机必须位于我们想要观察的物体之前，所以，通常要将照相机放在靠近场景最顶端的位置上。一个场景在同一时刻只能有一个激活的照相机。当前几何坐标变换将会影响照相机的空间位置。

技巧：当使用多个照相机时，可以使用切换（switch）节点来激活其中的某个照相机。

SoCamera

所有照相机节点类都是从抽象基类 **SoCamera** 派生出来的。(见图 4-1)

图 4-1 照相机节点类



SoCamera 具有下列域：

viewportMapping (SoSFEnum)	用于处理当照相机镜头的横纵比与视口窗口的横纵比不同的情况下,如何协调它们之间的映射关系.(详见“将照相机镜头的横纵比映射变换到视口窗口的横纵比上”)
position (SoSFVec3f)	用于定位照相机的视点位置。当前几何变换可以修改这个位置。
orientation (SoSFRotation)	照相机的观察方向。这个域用来描述照相机是如何相对于缺省方向进行旋转的。缺省情况下，照相机的观察方向是从(0.0, 0.0, 1.0)的位置指向坐标系原点，照相机的上方向(up direction)是(0.0,1.0, 0.0)。 这个域连同当前几何变换可以指定照相机在世界坐标系下(又称全局坐标系)的观察方向。
aspectRatio (SoSFFloat)	照相机视口的宽高比.这个值必须大于 0 .在 *SoCamera.h* 中预定义了一些宽高比数值: SO_ASPECT_SQUARE (1/1) SO_ASPECT_VIDEO (4/3) SO_ASPECT_HDTV (16/9)
nearDistance (SoSFFloat)	照相机的视点到近剪裁面的距离
farDistance (SoSFFloat)	照相机的视点到远剪裁面的距离
focalDistance (SoSFFloat)	照相机的视点到焦点的距离(通常用于 examiner 观察器)

在本章节的后面部分，图 4-2 和图 4-3 演示了照相机位置、方向、远近剪裁面、宽高比之间的关系。

当进行渲染遍历时，如果遇到的是一个照相机节点，Inventor 将执行下列步骤：

1. 在执行渲染动作期间，首先在场景中定位照相机。(根据照相机的 position 和

orientation 域指定照相机的位置和方向。同时修改当前的几何坐标变换也可以影响照相机的位置和方向)。

2. 照相机根据远近剪裁平面、横纵比、高度和高度角(依赖于照相机的类型)创建一个取景裁剪体(*view volume*)。取景裁剪体通常也叫做视图截锥(*viewing frustum*)，它是一个用来包围所要观察物体的六面锥台体。所有在取景裁剪体之外物体都将被剪裁丢弃。(本节的最后，有图表将演示不同类型的照相机是如何创建取景裁剪体的)。
3. 下一步，将 3D 取景裁剪体中的物体压缩映射到一张 2D 照片上，这个过程类似于使用光学照相机对真实世界进行拍摄的过程。然后将 2D 照片简单地映射到屏幕的 2D 窗口中 (见“将照相机的横纵比映射到视口上”)。
4. 接下来，使用照相机创建出来的投影矩阵来渲染图形场景的其余部分。

我们可以使用**pointAt()**方法来修改照相机**orientation**的域值。这个方法可以将照相机的方向指向一个特定的目标点。如果可能的话,它将尽可能保持照相机的“上方向”平行于+Y轴方向。否则将保持照相机的“上方向”平行于+Z轴方向。

pointAt()方法的语法如下：

```
void   pointAt(const  SbVec3f  &targetPoint)
```

SoCamera另外的两个常用方法是**viewAll()**和**getViewVolume()**。**viewAll()**方法可以很容易地让照相机使用当前的方向来观察整个场景。这个方法需要提供被观察场景的根节点作为第一个参数(通常这个根节点要包含本照相机节点)，以及需要提供渲染动作所要使用的视口区域（viewport region）作为第二个参数。*slack*参数通常用于定位远近剪裁平面。*slack*等于1.0(缺省值)将使远近剪裁平面“最紧密地包围住”（tightest fit）整个场景。**viewAll()**的语法如下：

```
void viewAll(SoNode *sceneRoot, const SbViewportRegion vpRegion,
             float slack = 1.0)
```

viewAll()方法会修改照相机的**position**、**nearDistance**、**farDistance**域值。它不影响照相机的方向值。在“使用不同照相机观察场景”的章节中有如何使用**viewAll()**的例子。

getViewVolume()方法返回照相机的取景裁剪体，通常用于和拾取操作有关的功能。

SoCamera的子类:

SoCamera类包括两个子类，如图 4-1 所示：

- SoPerspectiveCamera
- SoOrthographicCamera

SoPerspectiveCamera

SoPerspectiveCamera照相机类可以模拟人眼的功能：远处的物体变小，近处的物体变大。如果想模拟物体是怎样显示在人类的眼中，使用透视投影照相机是最自然不过的事情了。

SoPerspectiveCamera节点除了有**SoCamera**类所定义的所有域外，还另外附带一个域：heightAngle（SoSFFloat）指定取景裁剪体的垂直高度角(弧度单位)

SoPerspectiveCamera节点所定义的取景裁剪体是一个如图 4-2 所示的截棱锥。高度角和横纵比按照下面的公式来决定宽度角：

```
widthAngle = heightAngle * aspectRatio
```

SoOrthographicCamera

相对于透视投影照相机，**SoOrthographicCamera** 类所代表的是平行投影(*parallel* projections)照相机。平行投影方式不会因为距离的原因而使图像发生变形。对于某些需要确保精度的设计工作，视觉的变形有可能对准确测量产生干扰。所以平行投影照相机对这类工作特别有用。

SoOrthographicCamera节点除了有**SoCamera**类所定义的所有域外，还另外有一个域：

height (SoSFFloat) 指定取景裁剪体的高度。

SoOrthographicCamera 所定义的取景裁剪体是一个如图 4-3 所示的长方体。高度和横纵比按照下面的公式来决定矩形的宽度:

```
width = height * aspectRatio
```

图 4-2 SoPerspectiveCamera 节点的取景裁剪体和观察投影

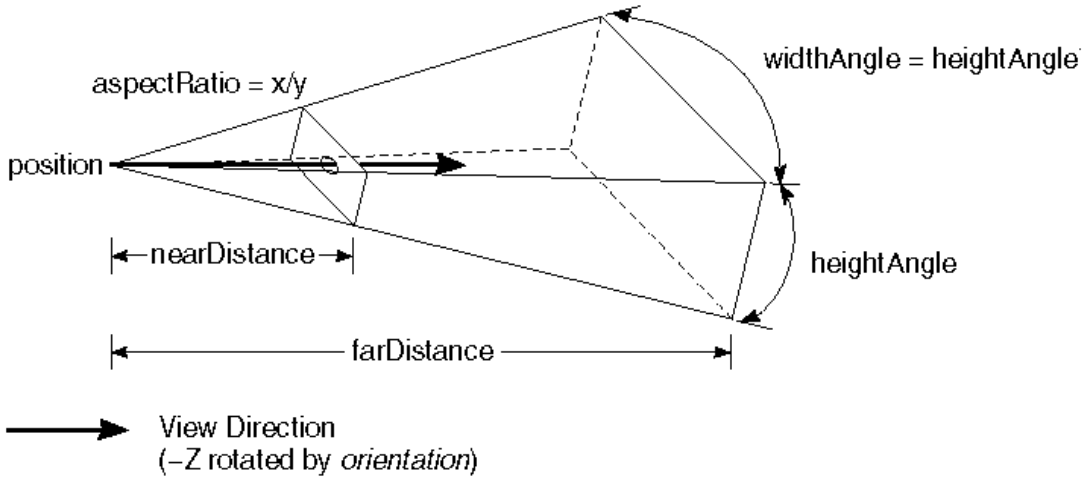
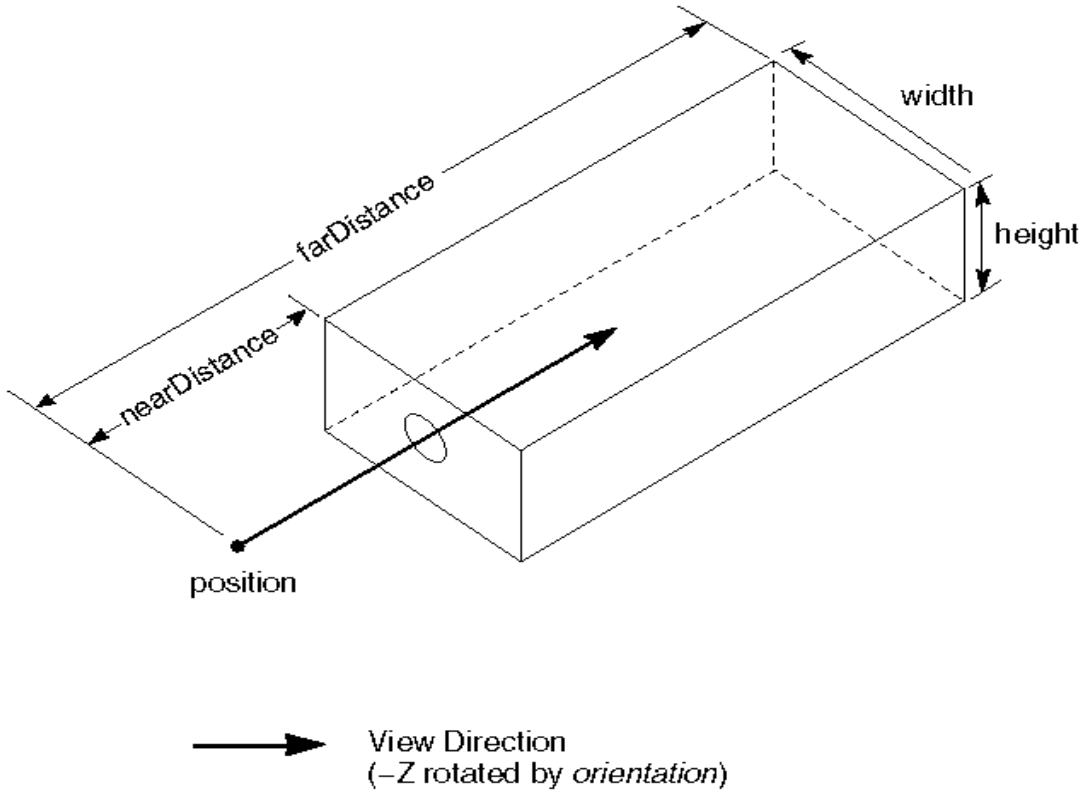


图 4-3 SoOrthographicCamera节点的取景裁剪体和观察投影



将照相机的横纵比映射到视口上

视口(*viewport*)是窗口用于显示被渲染场景的矩形区。缺省情况下，视口和窗口(**SoXtRenderArea)** 的尺寸大小相同。当构造**SoGLRenderAction** (见第 9 章)对象时，需要指定视口作为构造函数的其中的一个参数。

SoCamera的**viewportMapping**域是用来指定当照相机的横纵比和视口不同时，如何将照相机的投影映射到视口上。前三个选项是通过修改视口来匹配照相机的投影。这三个选项的优点是照相机的横纵比保持不变 (缺点是视口可能会有空白区(dead space)出现)。

- CROP_VIEWPORT_FILL_FRAME 调整视口以匹配照相机(见图 4-4)。使用最适当的横纵比率来绘制视口。同时在没有用的地方填充上灰色。
- CROP_VIEWPORT_LINE_FRAME 调整视口以匹配照相机。使用框线来绘制视口的边界。
- CROP_VIEWPORT_NO_FRAME 调整视口以匹配照相机。不绘制视口的边界.

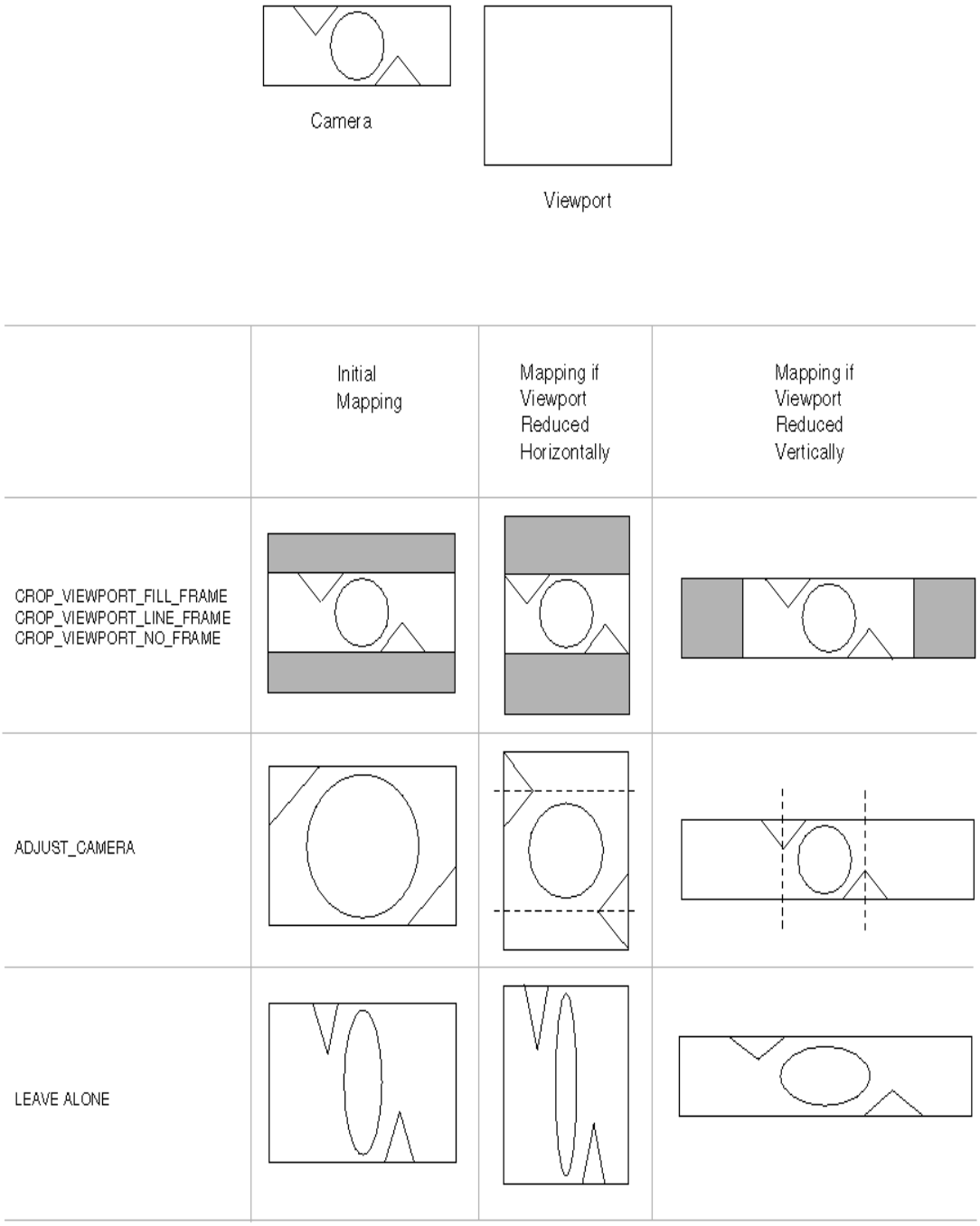
下面两个选项是调整照相机来匹配视口：

- ADJUST_CAMERA 调整照相机以匹配视口(见图 4-4)。照相机的投影图像是正常显示的，没有产生变形。(实际上是，保存在**aspectRatio**和**height**/**heightAngle**域中的数据是没有被修改的。如果视口映射需要的话，这些值只是临时地覆盖掉)。这个选项是缺省选项。
- LEAVE_ALONE 不修改任何数据。调整照相机的图像大小来匹配视口。这将有可

能产生一个变形的图像(见图 4-4)。

图 4-4 展示了不同类型的视口映射。在图 4-4 的例子中，照相机的横纵比是 3：1 而视口的横纵比是 1.5：1。最上面的照相机使用了 CROP_VIEWPORT_FILL_FRAME 视口映射选项。中间的照相机使用了 ADJUST_CAMERA。底下的照相机使用了 LEAVE_ALONE。图 4-4 还演示了映射三个过程（stages），最左列是初始视口映射。中间列是当视口被水平压缩后，是如何修改映射的。右边列是当视口被垂直压缩后，如何修改映射的。

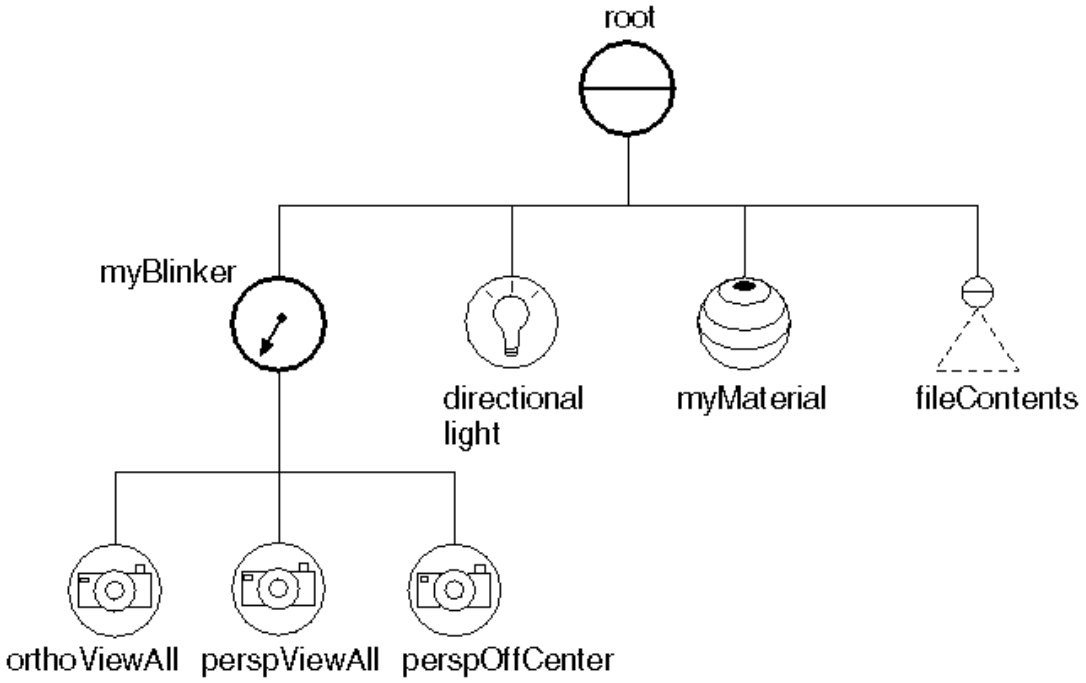
图 4-4 将照相机的横纵比映射到视口上



使用不同类型的照相机观察场景

例 4-1 演示了在不同的位置上使用一个平行投影照相机和两个透视投影照相机来观察场景的代码。例子使用了一个频闪节点（blinker node）(见 13 章描述)来切换这三个照相机节点。场景(一个公园长椅)是从一个文件中读取的。图 4-5 显示了例子中图形场景的结构。图 4-6 是这个例子产生的图像。

图 4-5 照相机例子的场景图



例 4-1 切换多个照相机

```
#include <Inventor/SbLinear.h>
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoBlinker.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoOrthographicCamera.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTransform.h>

main(int, char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
```

```
if (myWindow == NULL)
   exit(1);

SoSeparator *root = new SoSeparator;
root->ref();
```

图 **4-6** 照相机例子

```
// Create a blinker node and put it in the scene. A blinker
// switches between its children at timed intervals.
SoBlinker *myBlinker = new SoBlinker;
root->addChild(myBlinker);

// Create three cameras. Their positions will be set later.
// This is because the viewAll method depends on the size
// of the render area, which has not been created yet.
SoOrthographicCamera *orthoViewAll = new SoOrthographicCamera;
SoPerspectiveCamera *perspViewAll = new SoPerspectiveCamera;
SoPerspectiveCamera *perspOffCenter = new SoPerspectiveCamera;
myBlinker->addChild(orthoViewAll);
myBlinker->addChild(perspViewAll);
myBlinker->addChild(perspOffCenter);

// Create a light
root->addChild(new SoDirectionalLight);

// Read the object from a file and add to the scene
SoInput myInput;
if (! myInput.openFile("parkbench.iv"))
   return 1;
SoSeparator *fileContents = SoDB::readAll(&myInput);
```

```
   if (fileContents == NULL)
      return 1;

   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(0.8, 0.23, 0.03);
   root->addChild(myMaterial);
   root->addChild(fileContents);

   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);


   // Establish camera positions.
   // First do a viewAll() on all three cameras.
   // Then modify the position of the off-center camera.
   SbViewportRegion myRegion(myRenderArea->getSize());
   orthoViewAll->viewAll(root, myRegion);
   perspViewAll->viewAll(root, myRegion);
   perspOffCenter->viewAll(root, myRegion);
   SbVec3f initialPos;
   initialPos = perspOffCenter->position.getValue();
   float x, y, z;
   initialPos.getValue(x,y,z);
   perspOffCenter->position.setValue(x+x/2., y+y/2., z+z/4.);

   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Cameras");
   myRenderArea->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

当我们读完这个例子后，可以试着修改照相机的每个域值，看看当修改了照相机的位置、方向、横纵比、剪裁面位置和照相机高度(高度角)等参数后，屏幕图像发生了什么变化。试试使用**pointAt()**方法来修改照相机的方向。记住，在一个图形场景中，某一时刻只能有一个激活的照相机。而且这个激活的照相机必须是放在所要观察物体的前面。

灯光

当使用缺省光照模式时(Phong 模式)，在可以观察物体之前，场景中必须包含至少一个灯光节点。在执行渲染动作期间，如果遍历遇到了场景中的灯光节点，渲染将开启这个灯光节点(During a rendering action, traversing a light node in the scene graph turns that light on.)。场

景中灯光节点的位置可以决定两件事情：

- 灯光将照亮那些物体？ — 在图形场景中，灯光节点可以照亮跟随它之后任何物体。（灯光参数是遍历状态的一部分（见第 3 章中的描述）。使用**SoSeparator**节点可以将一个特定灯光节点所产生的效果与图形场景中其它部分隔离开来**(即灯光节点如果是包含在一个SoSeparator节点内,则这个灯光节点只会照亮SoSeparator节点内的物体。对于SoSeparator节点外的物体将不会产生影响。译者注)**）
- 灯光在 3D空间中位于何处? — 有些灯光节点(例如，**SoPointLight**)有一个**location**域。灯光的位置是受到当前几何坐标变换影响的。另外一些光源节点有一个**direction**域(例如，**SoDirectionalLight**)，同样，灯光的方向也是受到当前几何坐标变换影响的。

关于所有光源节点的另一个重要事实是：灯光效果是累积的。每当向图形场景中增加一个灯光节点时，场景就会变得亮一些。Open Inventor 能开启的最大灯光数依赖于系统当前 OpenGL 的具体实现。

(高级内容)

在某些情况下，我们也许想把场景中的灯光位置和它所照亮的物体分隔开。在例 4-2 所使用的**SoTransformSeparator**节点就可以只移动灯光的位置。使用传感器和引擎也是一种行之有效的影响灯光行为的方法。例如可以将一个传感器附着在一个球体上，当球体的位置发生改变时，传感器也可以同时修改灯光的位置。或者可以使用一个引擎来找到一个特定物体的路径，然后修改照亮这个物体的灯光的位置。(见“*Open Inventor C++ Reference Manual*”中**SoComputeBoundingBox)**。

SoLight

所有的灯光类都是从抽象基类**SoLight**派生出来的。**SoLight**是从**SoNode**派生出来的，它没有增加新的函数。**SoLight**有下列的域：

on (SoSFBool)	灯光是否开启
intensity (SoSFFloat)	灯光的亮度.数值的范围从 0.0(无光)到 1.0(最大亮度)
color (SoSFColor)	灯光的颜色

SoLight 的子类

SoLight类有三个子类，如图 4-7：

- SoPointLight
- SoDirectionalLight
- SoSpotLight

图 4-7 灯光节点类

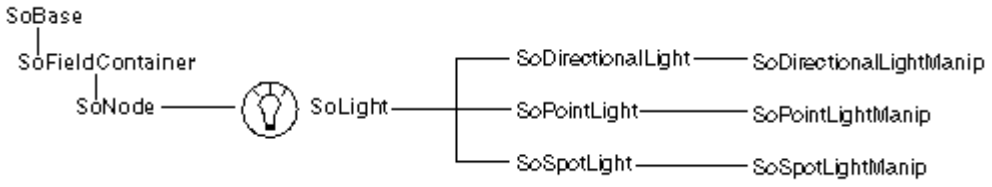


图 4-8 演示了不同类型的灯光效果。左边的图表示的是光线的方向。右边的图显示了在相同场景下不同灯光类型渲染的效果。彩图 2、彩图 3、彩图 4 也展示了不同类型的灯光效果。

提示：射灯(Directional lights)通常要比点光源(point lights)渲染的要快。它们俩同时又都比聚光灯(spotlights)渲染的要快。所以，如果想增加渲染速度的话，就要尽量使用少量、简单的灯光。

SoPointLight

SoPointLight灯光类，就像一颗星星那样，在给定的 3D空间位置上，均匀地向四周放射光线。**SoPointLight**节点有一个附加的域：

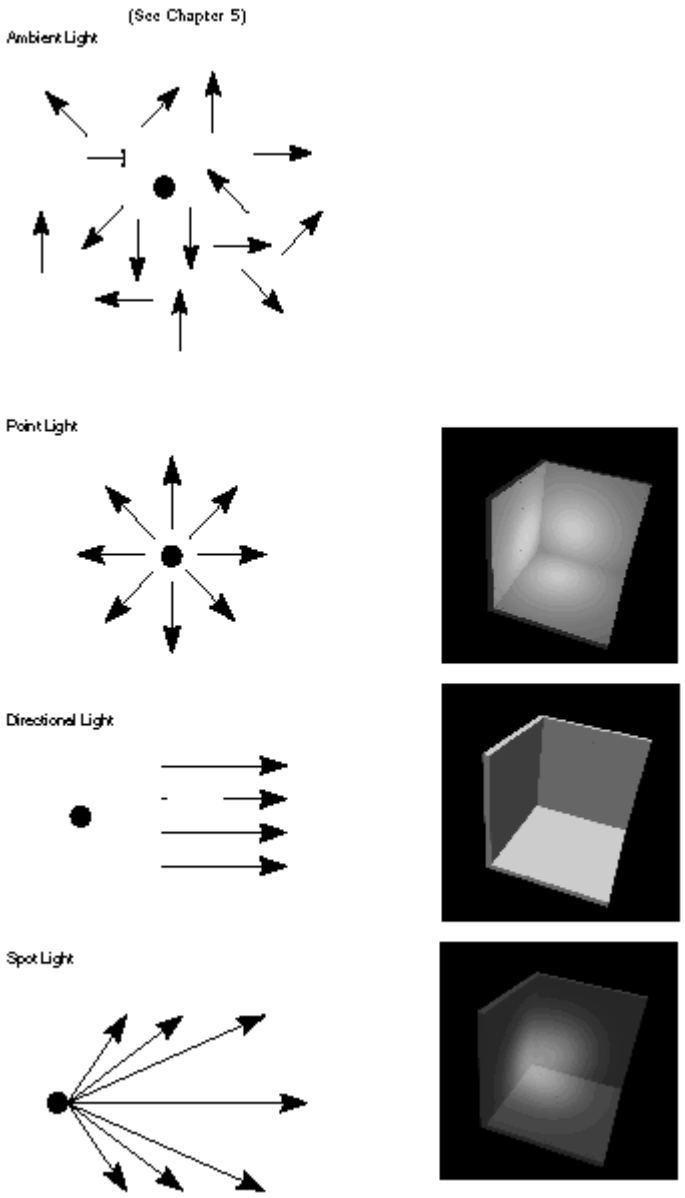
`location` (SoSFVec3f) 点光源的 3D空间位置(这个位置是受到当前几何变换影响的)

SoDirectionalLight

SoDirectionalLight灯光类只是均匀地按照一个方向放射光线。因为它的光照距离是无限的，所以它不需要指定 3D空间位置。**SoDirectionalLight**节点有一个附加的域：

`direction` (SoSFVec3f) 指定射灯发出光线的方向。(这个方向是受到当前几何变换影响的)。

图 4-8 灯光类型



提示：如果一个平面只是由一个多边形组成的(例如是一个大的矩形),并且这个多边形每个顶点的法向向量都相同,那么 Inventor 将不会显示出点光源的任何效果.这是因为灯光计算(使用 OpenGL)只是针对于每个顶点而言的。只有复杂的表面才能显示理想的效果.(这里所说的点光源的效果,就是指当点光源照射一个由多个多边形组成的复杂物体时,在物体的表面会有一处是最亮的,在这个最亮处的周围会逐渐暗下来.而如果点光源照射的是只是一个多边形的话,受到 OpenGL 的限制,在这个多边形上就不会出现上述的效果,多边形依照与光源位置的相对关系，要么整个都是明的，要么整个都是暗的，不会出现中间过渡的部分.译者注)

如图 4-8 所示, **SoDirectionalLight**灯光节点发出的光线都是平行的。平面上的所有点按照相同的方向反射这些光线。这样整个平面具有相同的光照强度。与此相反，从一个

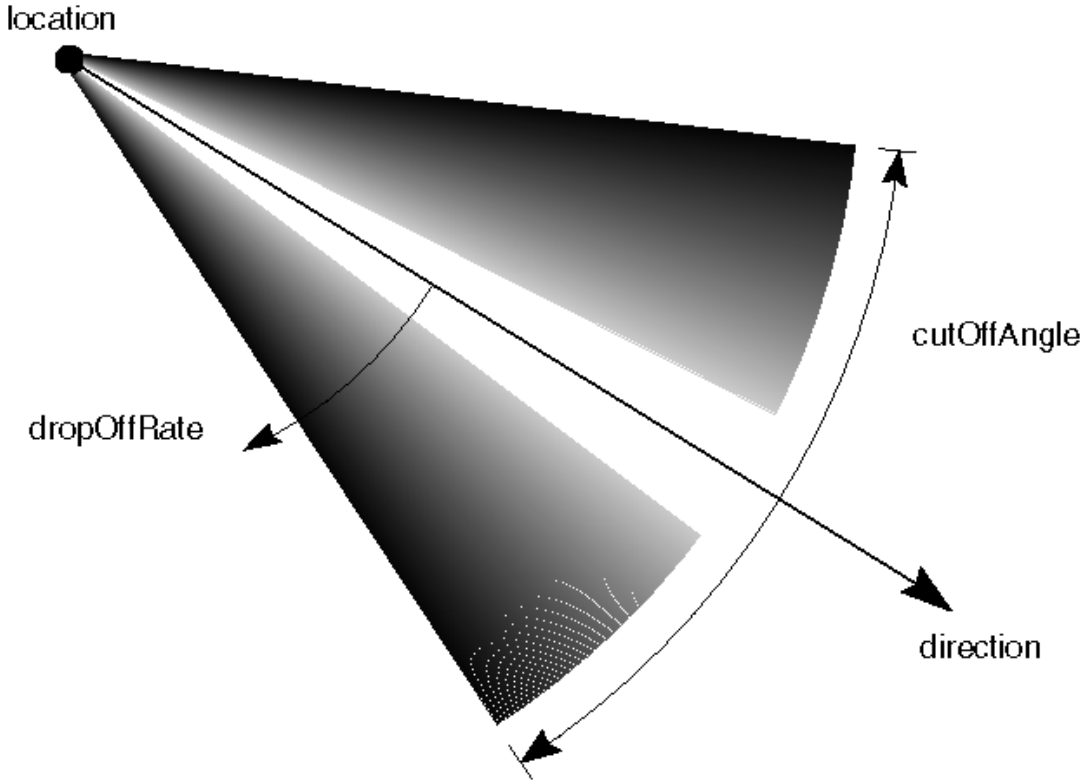
SoPointLight 光源发出的光线，照射到平面上时，它的光线强度是变化的。这是因为平面上不同点的法向方向和点光源的入射线之间的夹角角度不同。

SoSpotLight

SoSpotLight 灯光类所发出的光线是从空间中某一点开始,沿着一个主方向进行照射的。就像舞台上的聚光灯那样，它的灯光形状是一个以灯光所在位置为顶点的圆锥体光柱。**SoSpotLight** 节点有四个附加的域(见图 4-9)：

location(SoSFVec3f)	聚光灯的 3D位置(这个位置是受到当前几何变换影响的)
direction (SoSFVec3f)	照射的主方向
dropOffRate(SoSFFloat)	偏离照射主方向的光线强度衰减率(0.0 = 不改变强度. 1.0 = 最大衰减率).
cutOffAngle (SoSFFloat)	以弧度为单位的角度.所有在这个角度以外的光线强度都是 0.0. 这个角度是从圆锥的一个边到另外一个边来算的.

图 4-9 SoSpotLight 节点的域



使用多个灯光

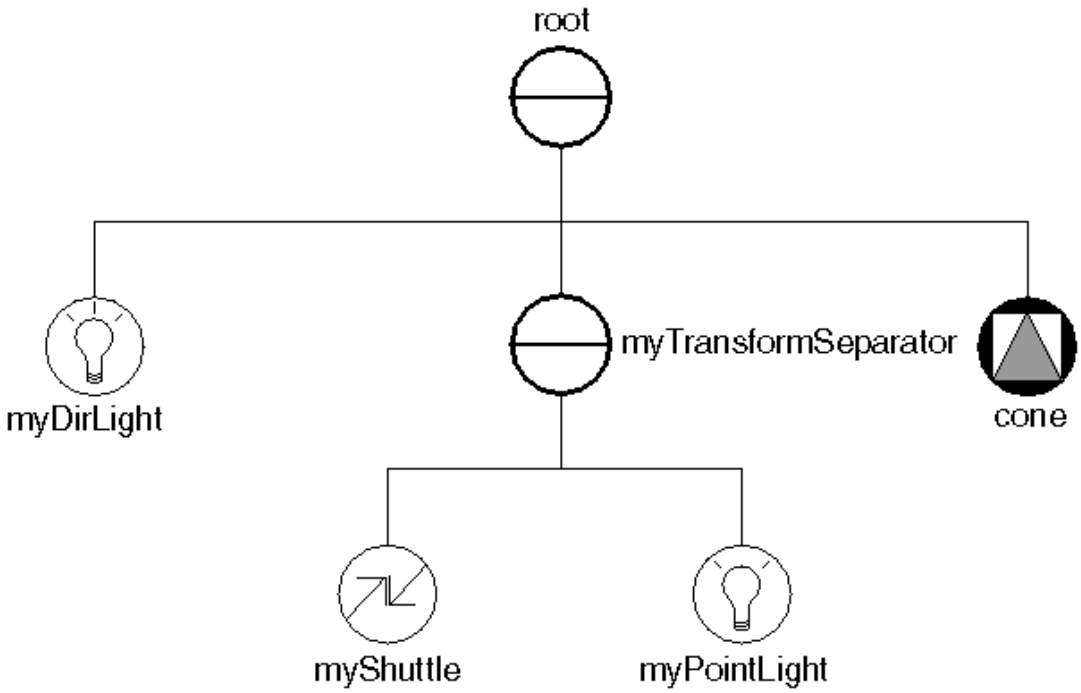
现在我们可以试验向场景中增加不同类型的灯光。例 4-2 中包含有两个光源：一个固定位置的红色射灯源和一个绿色的点光源。绿色的点光源被 **SoShuttle** 节点(见 13 章)所控制，前后来回往复运动。图 4-10 给出了这个例子的整个场景图。

例 4-2 使用不同类型的光源

```
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPointLight.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoShuttle.h>
#include <Inventor/nodes/SoTransformSeparator.h>

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL)
      exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   // Add a directional light
   SoDirectionalLight *myDirLight = new SoDirectionalLight;
   myDirLight->direction.setValue(0, -1, -1);
   myDirLight->color.setValue(1, 0, 0);
   root->addChild(myDirLight);

   // Put the shuttle and the light below a transform separator.
   // A transform separator pushes and pops the transformation
   // just like a separator node, but other aspects of the state
   // are not pushed and popped. So the shuttle's translation
   // will affect only the light. But the light will shine on
   // the rest of the scene.
   SoTransformSeparator *myTransformSeparator =
       new SoTransformSeparator;
   root->addChild(myTransformSeparator);

   // A shuttle node translates back and forth between the two
   // fields translation0 and translation1.
   // This moves the light.
   SoShuttle *myShuttle = new SoShuttle;
   myTransformSeparator->addChild(myShuttle);
```

```
    myShuttle->translation0.setValue(-2, -1, 3);
    myShuttle->translation1.setValue( 1,  2, -3);

    // Add the point light below the transformSeparator
    SoPointLight *myPointLight = new SoPointLight;
    myTransformSeparator->addChild(myPointLight);
    myPointLight->color.setValue(0, 1, 0);
```

图 4-10 灯光例子场景图




```
    root->addChild(new SoCone);

    SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Lights");
    myViewer->setHeadlight(FALSE);
    myViewer->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

(学习完本章的读者可能会知道，如果要创建一个场景，必须首先要创建好照相机和灯光这个两个节点，否则的话,我们将看不到任何后来产生的物体。但是读者在读一些 **Open Inventor** 的例子时会发现，有的例子中并没有创建照相机和灯光，却可以照常地看到场景中的物体。这是因为我们在创建场景时，一般会产生一个观察器对象(例如 **SoWinExaminerViewer**)。观察器对象有一个隐含的功能，它如果发现场景中没有照相机和灯光的话，它会自动为我们创建一个缺省的照相机和灯光，并把它们插放到整个场景的最顶部。这样即使我们没有创建照相机和灯光，我们也是可以看到物体的。当然，如果读者想增加一些必要的控制，例如提供正视图、俯视图、后视图等功能的话，就必须要创建自己的照相机了。译者注)

第五章 形体，属性和绑定（Binding）

本章目标

在阅读完本章节后，我们将可以做下列事情：

- 在场景中熟练使用多种类型的几何物体，包括那些利用顶点坐标和法线来表示的比较复杂的几何物体。
- 了解索引形体节点(indexed shapes)如何按照索引顺序来使用顶点坐标、材质、法向向量、和纹理数据。
- 体验不同颜色、亮度、和透明的效果。
- 用不同的绘制风格渲染场景中的不同部分。
- 使用不同类型的光照模型来渲染场景。
- 创建带有“雾”效果的场景。
- 利用形体提示（shape hints）、复杂度、和 Level of Detail 等节点提高渲染性能。
- 体验不同类型的材质和法向向量绑定。

为了便于讨论，我们将几何物体分成两类：简单形体和复杂形体。简单形体是“自包含”的节点，即节点包含自身所有的几何参数信息。相反，复杂形体则需要通过其它节点来为其指定顶点坐标和法向向量。本章还将讨论一些例如材质、绘制风格、光照模型等重要节点，举例说明关于几何坐标变换和材质、法向向量绑定等重要概念。

简单形体

所有形体节点都是从抽象基类**SoShape**派生出来的。Inventor提供了下列几种简单形体类：

- 立方体 (可以定义宽度、高度、深度)
- 圆锥体 (可以定义高度、底边半径)
- 球体 (可以定义半径)
- 圆柱体 (可以定义高度和半径)

图 5-1 绘制出了部分的形体类树图。

复杂形体

如图 5-2 所示，像三角条带集(triangle strip sets)和面集(face sets)等这样复杂形体，它们都至少需要一组顶点坐标数据。如果当前光照模型是 PHONG 的话，复杂形体还需要一组平面法向向量数据。顶点和法向数据是定义在场景中单独的节点内的，这样做的好处是其它节点可以共享这些数据。

复杂形体包括：

- 面集(Face set)，索引面集(indexed face set)
- 线集(Line set)，索引线集(indexed Line set)
- 三角条带集(triangle strip sets)，索引三角条带集(indexed triangle strip sets)
- 点集
- 四边形网格(Quad mesh)
- NURBS 曲线和曲面

图 5-1 形体节点类

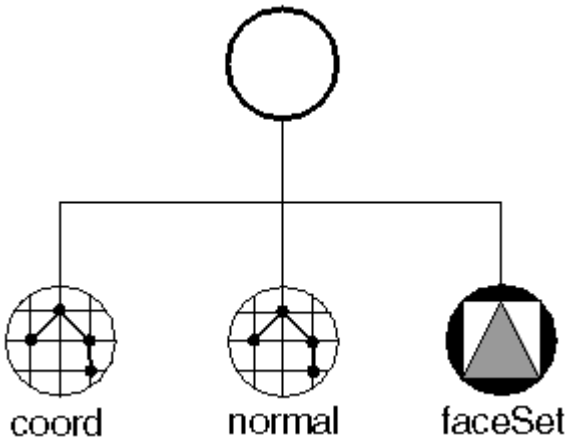
SoCoordinate3节点可以为某个点设置渲染状态中的当前坐标数据。它包含有一个**SoMFVec3f**类型的域(**point**)，例如：

```
SbVec3f                          verts[6];
SoCoordinate3                     *coord = new SoCoordinate3;
// ...Initialize vertices array ...
coord->point.setValues(0, 6, verts);
```

SoNormal节点可以为某个向量设置渲染状态中的当前平面法向数据。它包含有一个**SoMFVec3f**类型的域(vector)。

提示：Inventor也可以自动产生法线。在这种情况下，我们就不需要使用**SoNormal**节点了。详见“**自动产生法线**”章节

图 5-2 用于创建简单的索引面集的一些节点

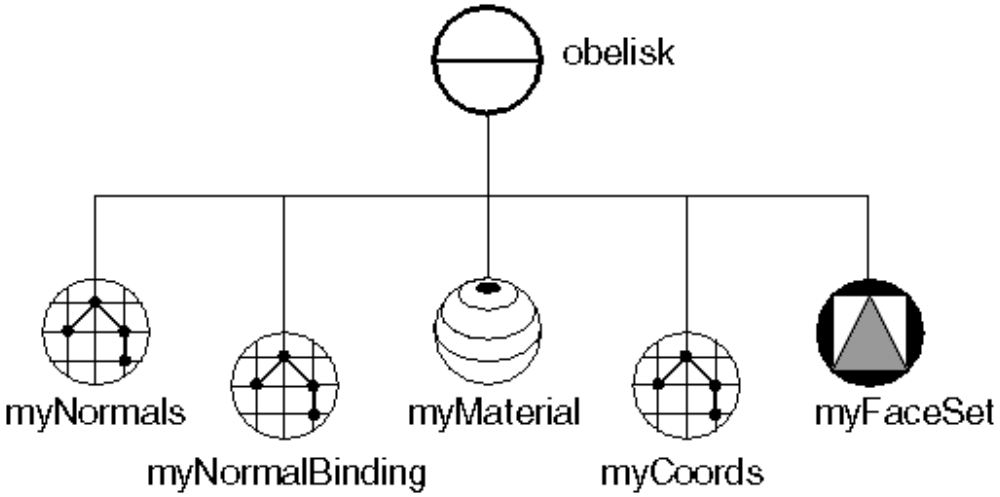


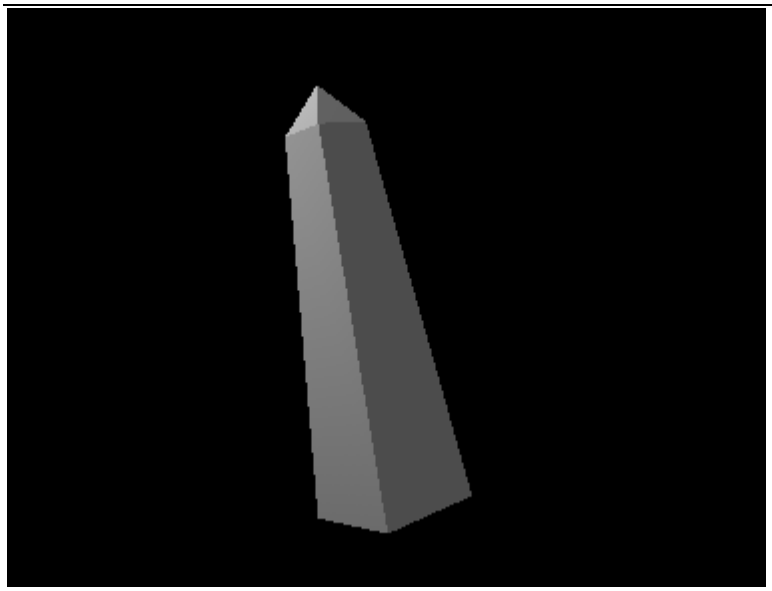
面集(Face Set)

SoFaceSet是用于表示多边形物体的节点，它以当前坐标、当前法线、当前材质、和当前纹理等节点为基础来构造多边形物体。它按照那些节点给定的顺序来使用节点中的数据。(如果希望按照不同的顺序使用坐标、法线、和材质的话，就需要使用**SoIndexedFaceSet**节点，我们将在下一节中做详细讨论)

例 5-1 利用一个由 8 个面组成的面集节点创建了一个方尖塔。图 5-3 展示了这个例子的场景图。现在我们先暂时不考虑法线绑定节点（normal binding），这个节点将在“绑定节点”章节中再做详细的讨论。

图 5-3 使用面集的场景例子





例 5-1 创建一个面集

```
//  Eight polygons. The first four are triangles
//  The second four are quadrilaterals for the sides.
static float vertices[28][3] =
{
   { 0, 30, 0}, {-2,27, 2}, { 2,27, 2},            //front tri
   { 0, 30, 0}, {-2,27,-2}, {-2,27, 2},            //left  tri
   { 0, 30, 0}, { 2,27,-2}, {-2,27,-2},            //rear  tri
   { 0, 30, 0}, { 2,27, 2}, { 2,27,-2},            //right tri
   {-2, 27, 2}, {-4,0, 4}, { 4,0, 4}, { 2,27, 2},  //front quad
   {-2, 27,-2}, {-4,0,-4}, {-4,0, 4}, {-2,27, 2},  //left  quad
   { 2, 27,-2}, { 4,0,-4}, {-4,0,-4}, {-2,27,-2},  //rear  quad
   { 2, 27, 2}, { 4,0, 4}, { 4,0,-4}, { 2,27,-2}   //right quad
};

// Number of vertices in each polygon:
static long numvertices[8] = {3, 3, 3, 3, 4, 4, 4, 4};

// Normals for each polygon:
static float norms[8][3] =
{
   {0, .555,  .832}, {-.832, .555, 0}, //front, left tris
   {0, .555, -.832}, { .832, .555, 0}, //rear, right tris
   {0, .0739,  .9973}, {-.9972, .0739, 0},//front, left quads
   {0, .0739, -.9973}, { .9972, .0739, 0},//rear, right quads
};

SoSeparator *
```

```
makeObeliskFaceSet()
{
   SoSeparator *obelisk = new SoSeparator();
   obelisk->ref();

   // Define the normals used:
   SoNormal *myNormals = new SoNormal;
   myNormals->vector.setValues(0, 8, norms);
   obelisk->addChild(myNormals);
   SoNormalBinding *myNormalBinding = new SoNormalBinding;
   myNormalBinding->value = SoNormalBinding::PER_FACE;
   obelisk->addChild(myNormalBinding);

   // Define material for obelisk
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(.4, .4, .4);
   obelisk->addChild(myMaterial);

   // Define coordinates for vertices
   SoCoordinate3 *myCoords = new SoCoordinate3;
   myCoords->point.setValues(0, 28, vertices);
   obelisk->addChild(myCoords);

   // Define the FaceSet
   SoFaceSet *myFaceSet = new SoFaceSet;
   myFaceSet->numVertices.setValues(0, 8, numvertices);
   obelisk->addChild(myFaceSet);

   obelisk->unrefNoDelete();
   return obelisk;
}
```

技巧：当要构建一个场景时，我们应该使用最少的节点来完成我们的目的。例如，要创建一个多面体，最好的方式是将多面体的所有坐标数据都放进一个**SoCoordinate**节点中，将所有面集的描述都放到一个**SoFaceSet**(或**SoIndexedFaceSet**)节点中，而不要使用多个节点来表示多个面。

索引面集(Indexed Face Set)

SoIndexedFaceSet也是用于表示多边形物体的节点，它同样是以当前坐标、当前法线、

当前材质和当前纹理等节点为基础来构造多边形物体。与**SoFaceSet**节点只能按照节点给定的顺序使用数据不同，**SoIndexedFaceSet**节点可以按照任意的顺序使用这些数据(当前坐标、法线、材质和纹理数据)。这个节点包含有四个用于指定数据顺序的索引域：

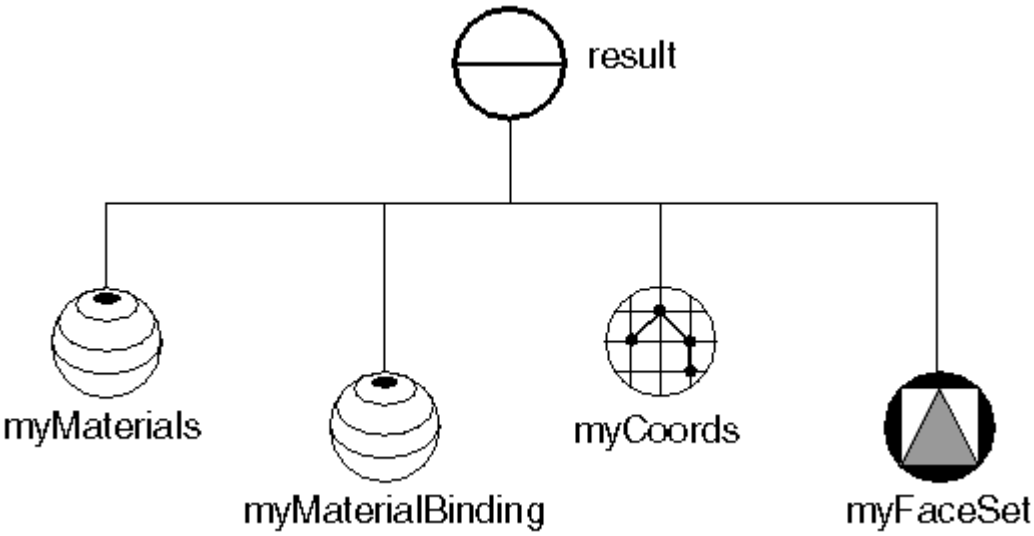
coordIndex (SoMFLong)	保存坐标数据列表中的索引值。按照这些索引值指定的坐标数据就可以构建起一组曲面。SO_END_FACE_INDEX (-1)代表一个面的结束和下一个面的开始。这个域永远都需要设置。
materialIndex (SoMFLong)	保存所用材质位于当前材质(组)中的索引值。这个域只有在 SoMaterialBinding 节点指定了索引材质绑定(indexed material binding)时才被使用。具体见“绑定节点”。
normalIndex (SoMFLong)	保存所用法线位于当前法线(组)中的索引值. 这个域只有在 SoNormalBinding 节点指定了索引法线绑定(不是每个顶点，就是每个面)时才被使用。具体见“绑定节点”。
textureCoordIndex (SoMFLong)	保存所用纹理坐标的索引值.(见第 7 章)

要保证包含在索引面集中的索引值是合法的，即这些索引值所指向的数据肯定能够在当前坐标点数组和当前法线数组中找到，否则的话，Inventor 将产生一个随即内存错误。

注意：如果我们使用**SoShapeHints**节点来提示Inventor，当前顶点是按照逆时针排列的，那么我们必须按照右手法则(*right-hand rule*)来指定顶点索引值。在使用右手法则时，如果将手指(除了拇指以外的四个手指)沿着平面顶点指定的方向弯曲手指，那么拇指所指向的方向就是几何法线方向。当然，我们也可以按照顺时针的顺序来给指定顶点的索引值，这时就应该使用左手法则来决定几何法线的方向。

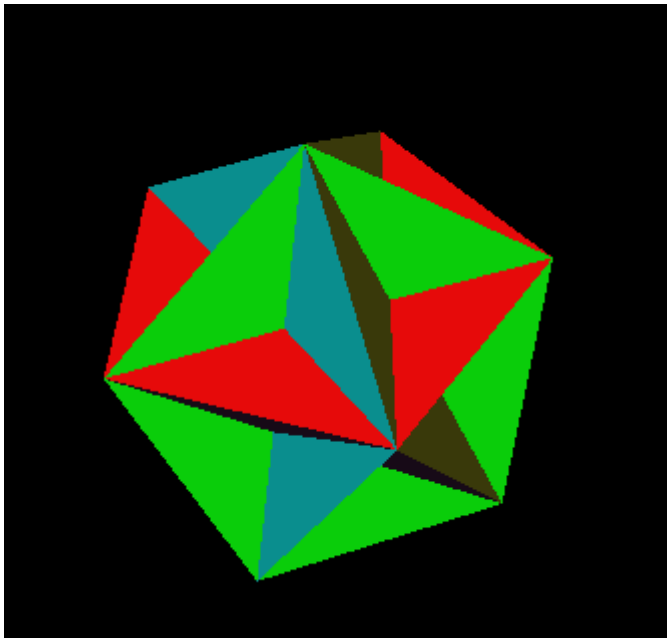
例子 5-2 将使用索引面集创建一个十二面星状体。这十二个相互交叉的平面都是一个五边形。图 5-4 给出了这个例子的场景图。

图 5-4 索引面集例子的场景图



例 5-2 产生一个索引面集

```
// Positions of all of the vertices:
//
static float vertexPositions[12][3] =
{
   { 0.0000,  1.2142,  0.7453},  // top

   { 0.0000,  1.2142, -0.7453},  // points surrounding top
   {-1.2142,  0.7453,  0.0000},
   {-0.7453,  0.0000,  1.2142},
   { 0.7453,  0.0000,  1.2142},
   { 1.2142,  0.7453,  0.0000},

   { 0.0000, -1.2142,  0.7453},  // points surrounding bottom
   {-1.2142, -0.7453,  0.0000},
   {-0.7453,  0.0000, -1.2142},
   { 0.7453,  0.0000, -1.2142},
   { 1.2142, -0.7453,  0.0000},

   { 0.0000, -1.2142, -0.7453}, // bottom
};
```



```
// Connectivity, information; 12 faces with 5 vertices each },
// (plus the end-of-face indicator for each face):

static long indices[72] =
{
   1,  2,  3,  4, 5, SO_END_FACE_INDEX, // top face

   0,  1,  8,  7, 3, SO_END_FACE_INDEX, // 5 faces about top
   0,  2,  7,  6, 4, SO_END_FACE_INDEX,
   0,  3,  6, 10, 5, SO_END_FACE_INDEX,
   0,  4, 10,  9, 1, SO_END_FACE_INDEX,
   0,  5,  9,  8, 2, SO_END_FACE_INDEX,

    9,  5, 4, 6, 11, SO_END_FACE_INDEX, // 5 faces about bottom
   10,  4, 3, 7, 11, SO_END_FACE_INDEX,
    6,  3, 2, 8, 11, SO_END_FACE_INDEX,
    7,  2, 1, 9, 11, SO_END_FACE_INDEX,
    8,  1, 5,10, 11, SO_END_FACE_INDEX,

    6,  7, 8, 9, 10, SO_END_FACE_INDEX, // bottom face
};

// Colors for the 12 faces
static float colors[12][3] =
{
```

```
   {1.0, .0, 0}, { .0,  .0, 1.0}, {0, .7,  .7}, { .0, 1.0,  0},
   { .7, .7, 0}, { .7,  .0,  .7}, {0, .0, 1.0}, { .7,  .0, .7},
   { .7, .7, 0}, { .0, 1.0,  .0}, {0, .7,  .7}, {1.0,  .0,  0}
};

// Routine to create a scene graph representing a dodecahedron
SoSeparator *
makeStellatedDodecahedron()
{
   SoSeparator *result = new SoSeparator;
   result->ref();

   // Define colors for the faces
   SoMaterial *myMaterials = new SoMaterial;
   myMaterials->diffuseColor.setValues(0, 12, colors);
   result->addChild(myMaterials);
   SoMaterialBinding *myMaterialBinding = new SoMaterialBinding;
   myMaterialBinding->value = SoMaterialBinding::PER_FACE;
   result->addChild(myMaterialBinding);

   // Define coordinates for vertices

   // Define coordinates for vertices
   SoCoordinate3 *myCoords = new SoCoordinate3;
   myCoords->point.setValues(0, 12, vertexPositions);
   result->addChild(myCoords);

   // Define the IndexedFaceSet, with indices into
   // the vertices:
   SoIndexedFaceSet *myFaceSet = new SoIndexedFaceSet;
   myFaceSet->coordIndex.setValues(0, 72, indices);
   result->addChild(myFaceSet);

   result->unrefNoDelete();
   return result;
}
```

三角条带集(triangle strip sets)

SoTriangleStripSet 节点使用当前坐标下的顶点数据来构造三角条带(triangle strips)。这是 Inventor 绘制多边形物体最快的方法之一。三角条带集以 **startIndex** 域所指定的索引值作为起点，按照顺序使用当前坐标数据。(如果没有指定索引值的话，则以第一个索引作为起始)

numVertices域用来指定每个三角条所使用的顶点数目，三角条带集可以按照下列方式声明：

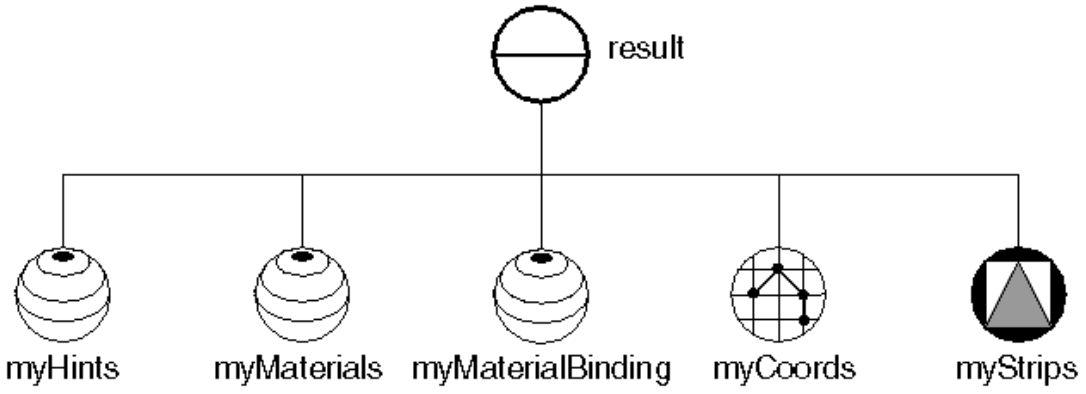
```
static long numVertices[2] =
{
   32, // flag
   8   // pole
};
SoTriangleStripSet *myStrips = new SoTriangleStripSet;
myStrips->numVertices.setValues(0, 2, numVertices);
```

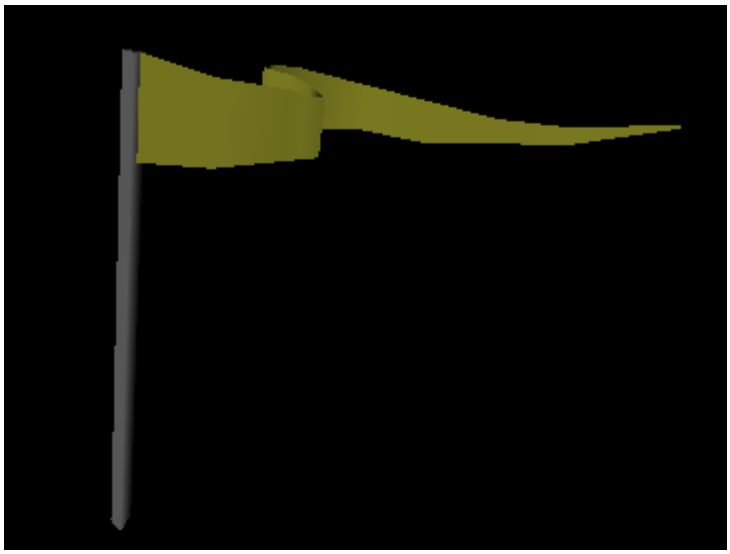
因为**numVertices**域包含有一个带有两个数据的数组，所以上面的代码将会创建两个三角条带。第一个三角条带(旗子)使用前 32 个坐标数据，第二个三角条带(旗杆)使用接下来的 8 个数据。第 0 个面决定顶点顺序 — 在这个例子中，顶点顺序是按照逆时针排列的。

提示：三角条带集和四边形网格(Quad mesh)通常比面集渲染速度要快.

例 5-3 将演示产生细长三角旗的代码。图 5-5 显示了这个例子的场景图。

图 5-5 三角条带集例子的场景图





例 5-3 创建一个三角条带集

```
// Positions of all of the vertices:

static float vertexPositions[40][3] =
{
   {  0,    12,     0 }, {    0,    15,     0},
   {2.1,  12.1,   -.2 }, { 2.1,  14.6,   -.2},
   {  4,  12.5,   -.7 }, {    4,  14.5,   -.7},
   {4.5,  12.6,   -.8 }, { 4.5,  14.4,   -.8},
   {  5,  12.7,    -1 }, {    5,  14.4,    -1},
   {4.5,  12.8,  -1.4 }, { 4.5,  14.6,  -1.4},
   {  4,  12.9,  -1.6 }, {    4,  14.8,  -1.6},
   {3.3,  12.9,  -1.8 }, { 3.3,  14.9,  -1.8},
   {  3,    13,  -2.0 }, {    3,  14.9,  -2.0},
   {3.3,  13.1,  -2.2 }, { 3.3,  15.0,  -2.2},
   {  4,  13.2,  -2.5 }, {    4,  15.0,  -2.5},
   {  6,  13.5,  -2.2 }, {    6,  14.8,  -2.2},
   {  8,  13.4,    -2 }, {    8,  14.6,    -2},
   { 10,  13.7,  -1.8 }, {   10,  14.4,  -1.8},
   { 12,    14,  -1.3 }, {   12,  14.5,  -1.3},
   { 15,  14.9,  -1.2 }, {   15,    15,  -1.2},

   {-.5,  15,     0 }, { -.5,  0,     0},    // the flagpole
   {  0,  15,   .5 }, {    0,  0,   .5},
   {  0,  15,  -.5 }, {    0,  0,  -.5},
   {-.5,  15,     0 }, { -.5,  0,     0}
};
```

```
// Number of vertices in each strip.
static long numVertices[2] =
{
   32, // flag
   8   // pole
};

// Colors for the 12 faces
static float colors[2][3] =
{
   { .5, .5,  1 }, // purple flag
   { .4, .4, .4 }, // grey flagpole
};

// Routine to create a scene graph representing a pennant.
SoSeparator *
makePennant()
{
   SoSeparator *result = new SoSeparator;
   result->ref();

   // A shape hints tells the ordering of polygons.
   // This ensures double-sided lighting.
   SoShapeHints *myHints = new SoShapeHints;
   myHints->vertexOrdering = SoShapeHints::COUNTERCLOCKWISE;
   result->addChild(myHints);

   // Define colors for the strips
   SoMaterial *myMaterials = new SoMaterial;
   myMaterials->diffuseColor.setValues(0, 2, colors);
   result->addChild(myMaterials);
   SoMaterialBinding *myMaterialBinding = new SoMaterialBinding;
   myMaterialBinding->value = SoMaterialBinding::PER_PART;
   result->addChild(myMaterialBinding);

   // Define coordinates for vertices
   SoCoordinate3 *myCoords = new SoCoordinate3;
   myCoords->point.setValues(0, 40, vertexPositions);
   result->addChild(myCoords);

   // Define the TriangleStripSet, made of two strips.
   SoTriangleStripSet *myStrips = new SoTriangleStripSet;
```

```
    myStrips->numVertices.setValues(0, 2, numVertices);
    result->addChild(myStrips);

    result->unrefNoDelete();
    return result;
}
```

四边形网格(Quad mesh)

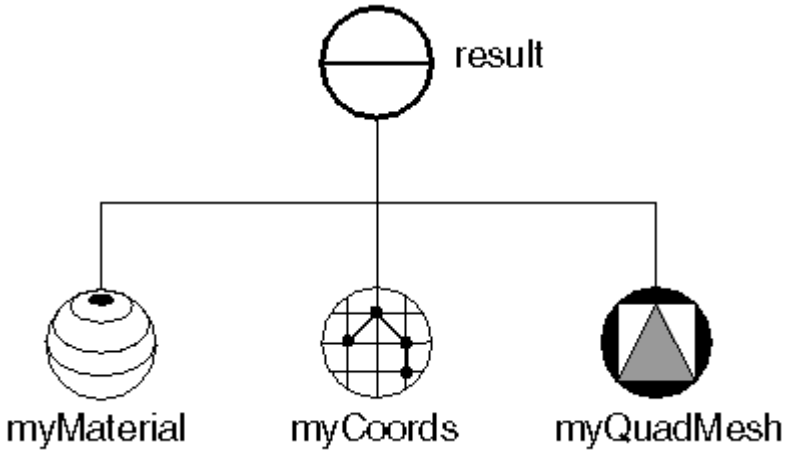
SoQuadMesh节点使用位于当前坐标下的顶点数据来构造四边形网格。四边形网格以**startIndex**域所指定的索引值作为起点，按照顺序使用当前坐标数据。(如果没有指定索引值的话，则以第一个索引作为起始)。

verticesPerColumn域和**verticesPerRow**域用来设置在网格中每行每列的顶点数。例 5-4 创建了一个四边形网格，代码如下：

```
SoQuadMesh *myQuadMesh = new SoQuadMesh;
myQuadMesh->verticesPerRow = 12;
myQuadMesh->verticesPerColumn = 5;
```

在这个四边形网格中，每行有 12 个顶点，每列有 5 个顶点。图 5-6 给出了这个例子的场景图。

图 5-6 四边形网格例子场景图



例 5-4 产生一个四边形网格

```
// Positions of all of the vertices:

static float vertexPositions[160][3] =
{  // 1st row
   {-13.0,  0.0, 1.5}, {-10.3, 13.7, 1.2}, { -7.6, 21.7, 1.0},
   { -5.0, 26.1, 0.8}, { -2.3, 28.2, 0.6}, { -0.3, 28.8, 0.5},
   {  0.3, 28.8, 0.5}, {  2.3, 28.2, 0.6}, {  5.0, 26.1, 0.8},
   {  7.6, 21.7, 1.0}, { 10.3, 13.7, 1.2}, { 13.0,  0.0, 1.5},
   // 2nd row
   {-10.0,  0.0, 1.5}, { -7.9, 13.2, 1.2}, { -5.8, 20.8, 1.0},
   { -3.8, 25.0, 0.8}, { -1.7, 27.1, 0.6}, { -0.2, 27.6, 0.5},
   {  0.2, 27.6, 0.5}, {  1.7, 27.1, 0.6}, {  3.8, 25.0, 0.8},
   {  5.8, 20.8, 1.0}, {  7.9, 13.2, 1.2}, { 10.0,  0.0, 1.5},
   // 3rd row
   {-10.0,  0.0,-1.5}, { -7.9, 13.2,-1.2}, { -5.8, 20.8,-1.0},
   { -3.8, 25.0,-0.8}, { -1.7, 27.1,-0.6}, { -0.2, 27.6,-0.5},
   {  0.2, 27.6,-0.5}, {  1.7, 27.1,-0.6}, {  3.8, 25.0,-0.8},
   {  5.8, 20.8,-1.0}, {  7.9, 13.2,-1.2}, { 10.0,  0.0,-1.5},
   // 4th row
   {-13.0,  0.0,-1.5}, {-10.3, 13.7,-1.2}, { -7.6, 21.7,-1.0},
   { -5.0, 26.1,-0.8}, { -2.3, 28.2,-0.6}, { -0.3, 28.8,-0.5},
   {  0.3, 28.8,-0.5}, {  2.3, 28.2,-0.6}, {  5.0, 26.1,-0.8},
   {  7.6, 21.7,-1.0}, { 10.3, 13.7,-1.2}, { 13.0,  0.0,-1.5},
   // 5th row
   {-13.0,  0.0, 1.5}, {-10.3, 13.7, 1.2}, { -7.6, 21.7, 1.0},
```

```
   { -5.0, 26.1, 0.8}, { -2.3, 28.2, 0.6}, { -0.3, 28.8, 0.5},
   {  0.3, 28.8, 0.5}, {  2.3, 28.2, 0.6}, {  5.0, 26.1, 0.8},
   {  7.6, 21.7, 1.0}, { 10.3, 13.7, 1.2}, { 13.0,  0.0, 1.5}
};

// Routine to create a scene graph representing an arch.
SoSeparator *
makeArch()
{
   SoSeparator *result = new SoSeparator;
   result->ref();

   // Define the material
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(.78, .57, .11);
   result->addChild(myMaterial);

   // Define coordinates for vertices
   SoCoordinate3 *myCoords = new SoCoordinate3;
   myCoords->point.setValues(0, 60, vertexPositions);
   result->addChild(myCoords);

   // Define the QuadMesh.
   SoQuadMesh *myQuadMesh = new SoQuadMesh;
   myQuadMesh->verticesPerRow = 12;

   myQuadMesh->verticesPerColumn = 5;
   result->addChild(myQuadMesh);

   result->unrefNoDelete();
   return result;
}
```

属性节点

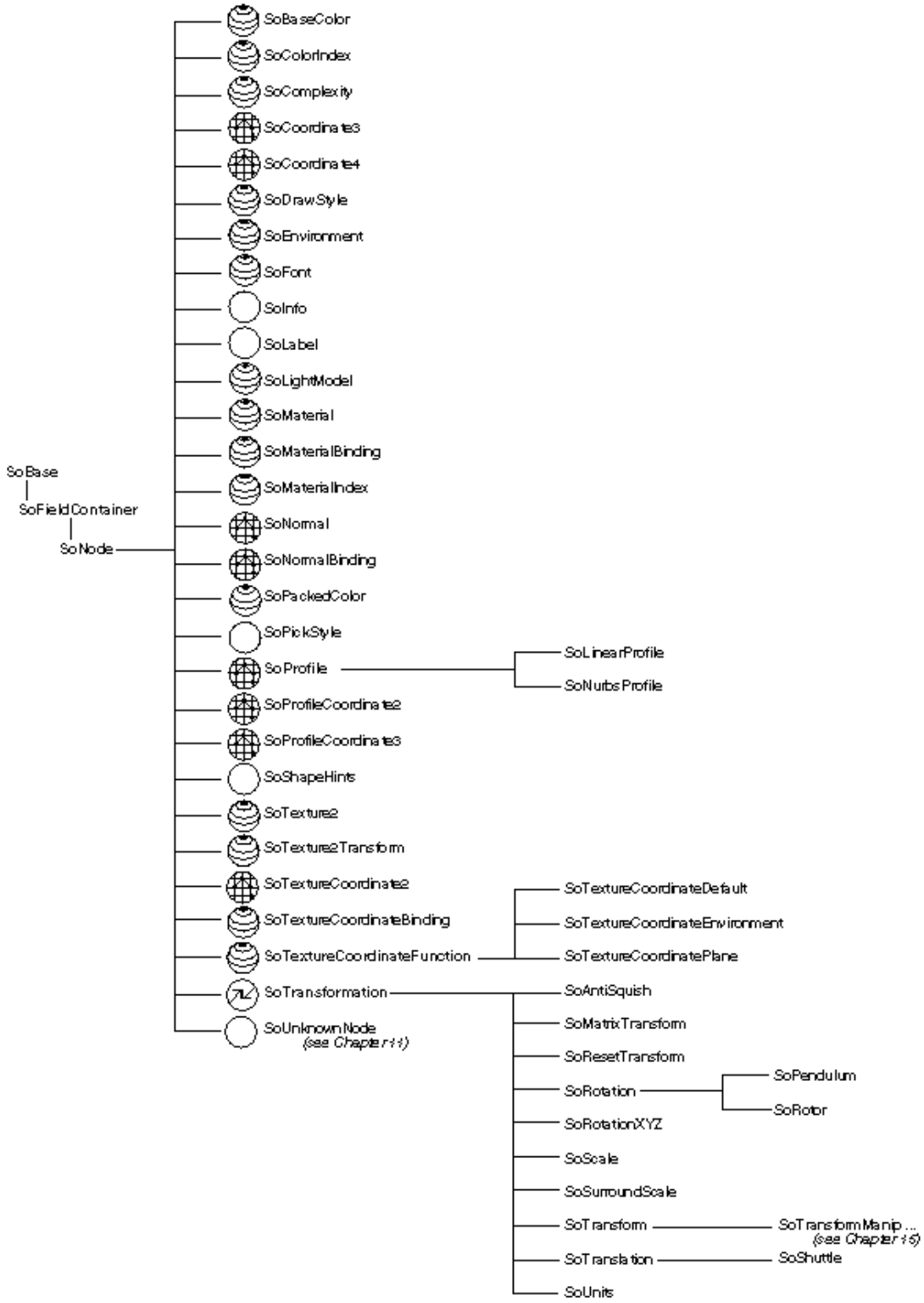
本节将讨论许多重要的属性节点，这些属性节点类都是从**SoNode**派生的：

- **SoMaterial** 设置当前材质的周围环境颜色(ambient color)、漫射光颜色(diffuse color)、镜面反射颜色(diffuse color)、辐射光颜色(emissive color)、亮度、和透明度等属性。
- **SoDrawStyle**在渲染过程中指定形体节点使用何种绘制风格。
- **SoLightModel** 在渲染过程中指定形体节点该如何计算光线效果。

- **SoEnvironment** 可以模拟一些例如烟雾、模糊、污渍(pollution)、烟尘等大气效果。还可以描述其它例如像环境光和光线变化等全局环境属性。
- **SoShapeHints** 提供一些有关几何顶点的附加信息，可以帮助Inventor优化一些渲染特性。
- **SoComplexity** 设置组成形体对象多边形的细分程度。也可以指定贴图纹理的复杂度和细节级别(level of detail)。
- **SoUnits** 在场景中为所有后面的形体节点定义度量单位。

如在本节后面所描述的那样，这些类将会影响到不同的渲染状态元素。图 5-7 给出了部分属性节点的类树图。

图 5-7 属性节点类



材质节点

SoMaterial节点包括下列域值：

ambientColor (SoMFColor) 环境光颜色。缺省值为[0.2, 0.2, 0.2]

diffuseColor (SoMFColor)	漫反射颜色。缺省值为 [0.8, 0.8, 0.8]。
specularColor (SoMFColor)	高光颜色，缺省值为[0.0, 0.0, 0.0]。
emissiveColor (SoMFColor)	物体自身发光的颜色。缺省值为[0.0, 0.0, 0.0]。
shininess (SoMFFloat)	物体表面发光度,范围从 0.0 到 1.0。 0.0 表示无光的漫反射面。1.0 表示高光的抛光面。缺省值为 0.2
transparency (SoMFFloat)	物体表面的透明度。范围从 0.0 到 1.0。 0.0 表示不透明表面。而 1.0 表示一个完全透明的表面。 缺省值为 0.0.

提示：在渲染动作中指定透明类型(见第 9 章)

下面是一个在**SoMaterial**节点中设置数据的例子：

```
SoMaterial *gold   = new SoMaterial;
//Set material values
gold->ambientColor.setValue(.3, .1, .1);
gold->diffuseColor.setValue(.8, .7, .2);
gold->specularColor.setValue(.4, .3, .1);
gold->shininess = .4;
```

因为金子是不透明的，所以我们使用缺省的透明度 0.0。

SoBaseColor也是从**SoNode**派生出来的。这个类只设置当前材质的漫反射颜色域(diffuse color field)，不会影响其它材质域。

提示：如果只是想改变物体的漫反射颜色, 可以使用**SoBaseColor**节点来代替**SoMaterial**节点。例如，要表现一个具有许多不同漫反射颜色的复杂地形时，可以首先使用一个**SoMaterial**节点来指定物体的环境光色、高光色、自身发光色，然后再使用一个**SoBaseColor**节点来定义那些带有漫反射颜色的数据。**SoBaseColor**节点类在光照模型是BASE_COLOR时也是非常有用的(见“光照模型节点”章节)。

(使用SoBaseColor节点的好处是可以节约内存。SoBaseColor要比SoMaterial占用的内存小很多。为了真实地表现地形特征，就必须为地形上的每个顶点都指定一个颜色，这就需要保存大量的颜色信息。SoBaseColor节点非常适合于这种情况。译者注)

绘制风格节点

SoDrawStyle节点包括下列域：

style 当前绘制风格，域值可以为：
(SoSFEnum) SoDrawStyle::FILLED 填充(缺省)
SoDrawStyle::LINES 非填充，轮廓线
SoDrawStyle::POINTS 点型
SoDrawStyle::INVISIBLE 隐藏

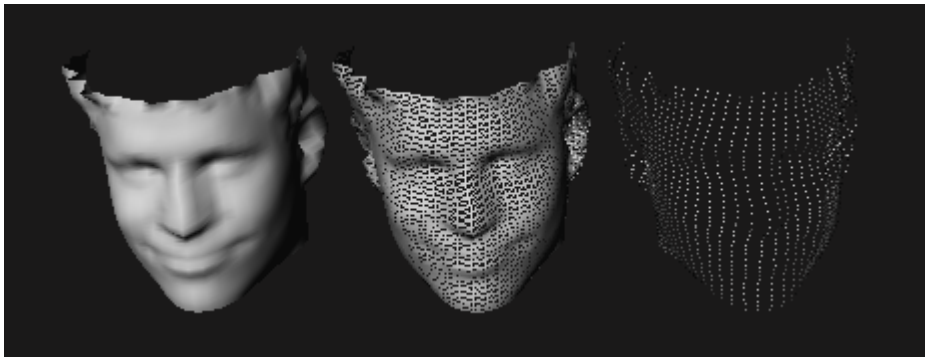
pointSize (只针对点型风格)
(SoSFFloat) 以打印机上点宽度(一英寸=72.2 打印机点)为单位，定义点的半径。缺省值为 0.0。0.0 表示让 Inventor 自己决定使用什么半径来最快地渲染场景，通常半径为 1.0 的点渲染速度最快。 如果这个数据不是 0.0，则 Inventor 要根据视口区域的每英寸的点数，按照所需数目的比例来调整点的大小，以此来保证这些点有固定的大小。

提示：LINES 和 POINTS 绘制风格在 BASE_COLOR 光照模型下会有最好的绘制效果.

lineWidth (只针对线型风格)
(SoSFFloat) 以打印机的点宽度(一英寸=72.2 打印机点)为单位，定义线的宽度。数据范围从 0.0 到 256.0。缺省数值为 0.0，表示使用最快的宽度数据来渲染场景。

linePattern (只针对线型风格)
(SoSFUShort) 当前点画线的样式。数据的范围从 0 (隐藏)到 0xffff(填充)。缺省值为 0xffff。

下面是同一模型使用不同的绘制风格，最终渲染出的效果。



光照模型节点

SoLightModel节点包括下列域值：

model (SoSFEnum) 设置当前的光照模型，将应用到场景中位于其后的物体节点上。光照模型指定在渲染过程中形体节点该如何计算光线效果。这个域值可以是：

SoLightModel::BASE_COLOR：

忽略光源。只使用当前材质的漫反射颜色和透明度.

SoLightModel::PHONG：(缺省的光照模型)

使用 OpenGL Phong 光照模型。这种模型要考虑场景中所有的光源，并且还要考虑物体表面方向与光线方向之间的关系。使用这种光照模型，场景中至少需要存在一个光源。(也许也会有物体辐射光和环境光等)

注意：在 Inventor 中,着色处理(shading)(例如对地面或平面)是由物体材质、光照模型和法线绑定这些因素共同决定的。在 Inventor 中没有明确地指定着色模型(shading model)。

彩图 5 和彩图 6 使用不同光照模型来显示相同的物体。（彩图 5 使用 BASE_COLOR，彩图 6 使用 PHONG）。

SoMaterial和**SoBaseColor**可以和任何类型的绘制风格和光照模型一起使用。然而在某些情况下，有些材质的属性也许会被忽略掉。例如，如果我们将**SoLightModel**的**model**域值设置成BASE_COLOR，那么渲染将只使用当前材质的漫反射颜色和透明度。如果我们只使用**SoBaseColor**设置了基本颜色，而同时又选择了Phong光照模型的话,这时Inventor将如何处理呢？在这种情况下,Inventor使用我们定义的基本颜色来设置当前材质的漫反射颜色，使用缺省或当前材质元素的数据来设置**SoMaterial**的其它域值。

注意：缺省情况下，Inventor的光照模型是PHONG。为了准确地渲染图像，我们需要指定法线和光源。但是，如果我们只是想看到一个有颜色的物体，而不想有什么其它的效果时，我们可以将光照模型指定为BASE_COLOR，并使用**SoBaseColor**指定漫反射颜色。

环境节点

使用**SoEnvironment**节点可以模拟一些例如烟雾、模糊、污渍(pollution)、烟尘等大气效果。一般来说，这些大气效果统称为术语“雾”（*fog*）。“雾”和“模糊”这两种效果之间的区别，简单说就是颜色与密度不同而已。

使用**SoEnvironment**节点可以设置环境光的颜色和亮度、点光源和射灯源的灯光衰减系数以及雾的类型、颜色、可视因子（visibility factor）等参数。彩图 7 演示了**SoEnvironment**节点所产生的效果。这张图像使用的是FOG类型的雾效果，雾的颜色是(0.2, 0.2, 0.46)。

SoEnvironment节点包含下列域：

ambientIntensity (SoSFFloat)	环境光亮度。这个域只用在 Phong 光照模型下。
ambientColor (SoSFColor)	环境光颜色。这个域只用在 Phong 光照模型下。
attenuation (SoSFVec3f)	定义光线如何随着距离的增加而逐步减弱。可以设置光线到物体表面之间的平方、线性和常量形式的衰减系数（向量中的 3 个内部变量按照顺序分对应平方、线性和常量衰减系数）。这个域只用在 Phong 光照模型下。
fogType (SoSFEnum)	雾的类型。这个域可以是下列数据： SoEnvironment::NONE 没有雾效果(缺省设置) SoEnvironment::HAZE 雾的不透明度随与照相机距离的增加而以线性方式增加 SoEnvironment::FOG 雾的不透明度随与照相机距离的增加而以指数方式增加 SoEnvironment::SMOKE 雾的不透明度随与照相机距离的增加而以指数平方方式增加
fogColor (SoSFColor)	雾的颜色。
fogVisibility (SoSFFloat)	指定一个与照相机位置之间的距离，场景中所有位于这个距离之外的物体将变得完全模糊。缺省值使（0.0）。如果数据是 0.0，则距离将被调整为照相机远裁剪面的距离。否则，将按照实际数值设定距离。

提示：如果要绘制一个有真实感的场景，应该在绘制雾化物体之前，首先将窗口背景颜色设置成雾的颜色。（具体见**SoXtRenderArea**::**setBackgroundColor()** 方法）。

形体提示节点（Shape-Hints）

缺省情况下，Inventor并不会对诸如：形体顶点是否按一定的顺序排列、形体表面多边形是闭合还是开放的、是凹的还是凸的等这些事情做任何的假设。但是，如果我们自己事先知道顶点都是按照一致的顺序排列的、形体都是闭合的、形体表面多边形都是凸的，那么我们可以使用**SoShapeHints**节点通知Inventor，这样可以优化一些渲染特性。

SoShapeHints节点带有 4 个域：

vertexOrdering (SoSFEnum)	对于所有从 SoVertexShape 派生出来的，基于顶点的形体节点来说，这个域指定了一个关于顶点顺序的提示。这个域描述了当从形体的外部观察时，物体的所有平面中的所有顶点的排列顺序。 域值可以是下面的数值： SoShapeHints::UNKNOWN_ORDERING 顶点顺序是未知的（缺省设置） SoShapeHints::CLOCKWISE 描述每个面的顶点数据是按照顺时针方向指定的。 SoShapeHints::COUNTERCLOCKWISE 描述每个面的顶点数据是按照逆时针方向指定的
shapeType (SoSFEnum)	SoShapeHints::UNKNOWN_SHAPE_TYPE 形体类型是未知的（缺省设置） SoShapeHints::SOLID 形体是一个封闭的实体（不是一个开放的平面）
faceType (SoSFEnum)	SoShapeHints::UNKNOWN_FACE_TYPE 面类型是未知的 SoShapeHints::CONVEX 形体的所有多边形面都是凸的（缺省设置）
creaseAngle (SoSFFloat)	用于自动计算法向向量。具体见“自动计算法向向量”。

如果shapeType域是SOLID，vertexOrdering域是CLOCKWISE 或COUNTERCLOCKWISE，那么Inventor将启用背面剔除（backface culling）功能，并关闭双面光（two-sided lighting）。如果shapeType域不是SOLID，vertexOrdering域是CLOCKWISE 或COUNTERCLOCKWISE，那么Inventor将关闭背面剔除功能，并且启用双面光。其它情况下，背面剔除和双面光都将被关闭。如果要使用**SoShapeHints**节点的话，一定要正确描述物体真实的情况，否则物体也许会发生渲染错误。

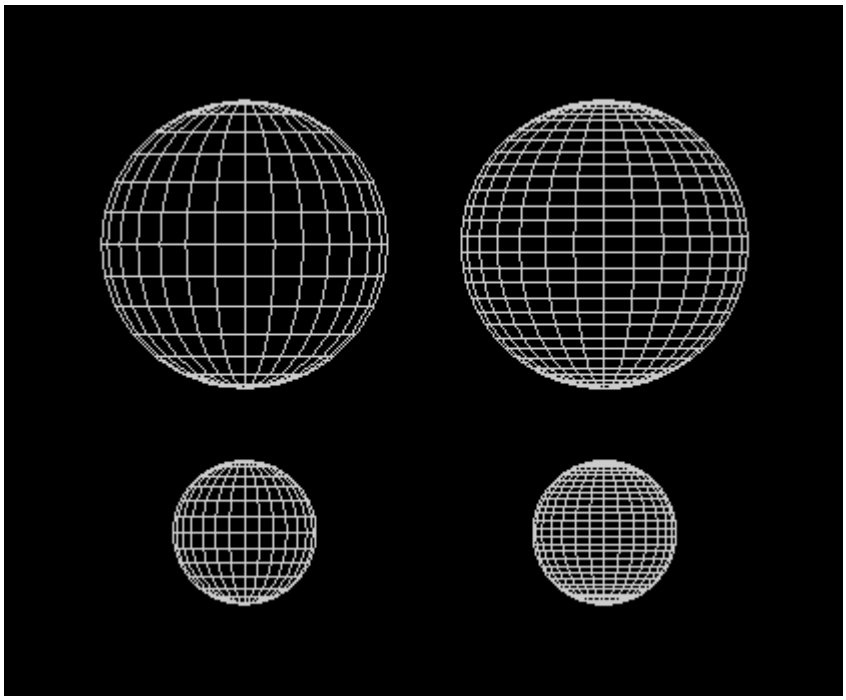
提示：通常情况下，我们使用形体提示节点提供的信息越多，渲染速度也就越快。唯一的例外是 shapeType 域不是 SOLID，vertexOrdering 域是 CLOCKWISE 或 COUNTERCLOCKWISE。在这种情况，因为自动打开了双面光，关闭了背面剔除，可能会导致渲染速度变慢。

复杂度节点

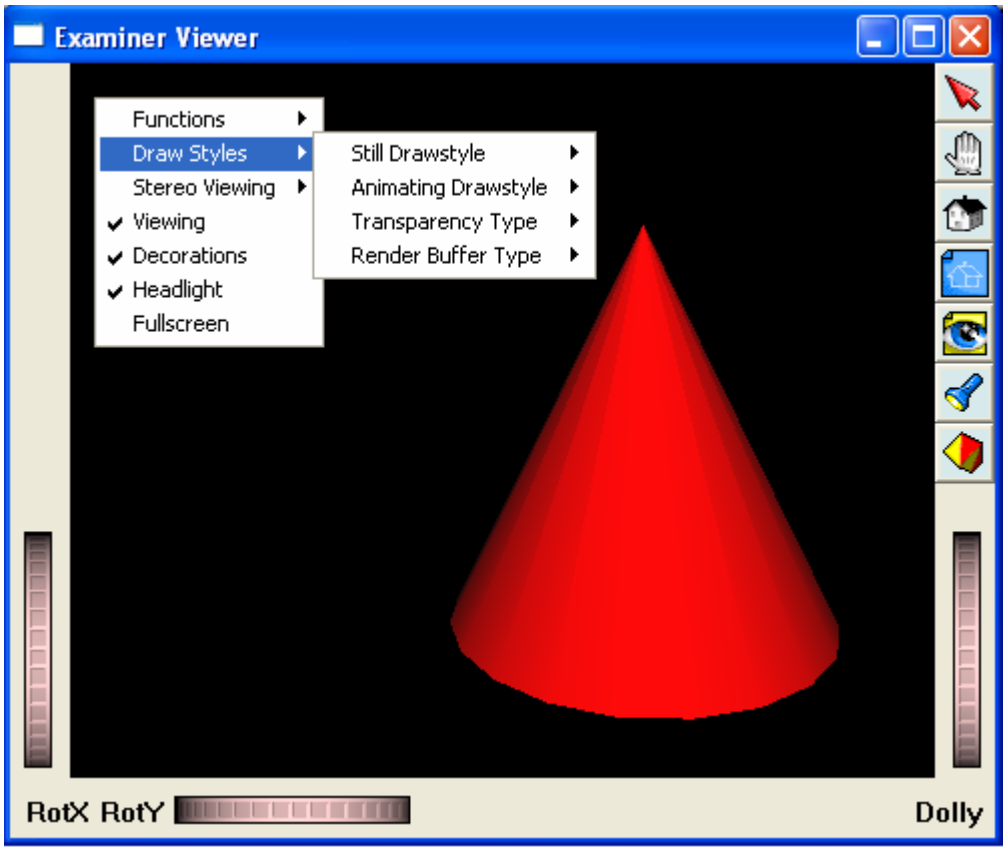
SoComplexity节点用于为场景中位于其后的形体节点指定多边形细分程度。节点包含有 3 个域：

type (SoSFEnum)	复杂度类型。可以是下列域值： SoComplexity::OBJECT_SPACE 基于物体自身的复杂度。不管物体位于屏幕何处，或者物体的那个部分更接近观察者（缺省设置）。 SoComplexity::SCREEN_SPACE 基于物体所占屏幕面积的复杂度。物体需要知道更多关于全屏幕的细节。小物体只需要更少的细节。这样靠近观察者的物体要比远离观察者的物体获得更多的显示细节。这种复杂度类型需要比其它的类型花费更多的计算时间。另外，当照相机移动后，这种复杂度类型将导致渲染缓存失效。（具体见第 9 章 渲染缓存）。 SoComplexity::BOUNDING_BOX 只在物体所在位置显示一个包围盒。这种类型用于不需要精确显示物体，而只需要显示速度的情况。这种类型使用当前绘制风格来渲染包围盒。
value (SoSFFloat)	指定希望的细分程度。0.0 是最小复杂度，1.0 是最大复杂度。缺省值是 0.5。
textureQuality (SoSFFloat)	指定物体的纹理贴图品质。应在渲染速度和纹理品质之间寻找平衡点。0.0 代表最大渲染速度（可能会完全关闭纹理贴图）。1.0 代表最好的纹理品质。缺省值为 0.5。

下面的“设置不同级别的复杂度（左边是OBJECT_SPACE；右边是SCREEN_SPACE)”例图演示了同一个物体使用不同级别的复杂度所产生的不同效果。左边的球体使用Object-Space复杂度类型，复杂度为 0.5。右边的球体使用Screen-Space复杂度类型，复杂度是 0.6。 在第 8 章NURBS中，有使用**SoComplexity**节点的例子。



提示：简单场景的渲染速度肯定要快于复杂场景的渲染速度。举例来说，使用少量的灯光，关闭纹理贴图或者降低纹理贴图的品质，选择一个像线框这样简单的绘制风格，设置一个低复杂度数值，这些都可以增加渲染速度。观察器的弹出菜允许我们为提高渲染速度而禁止某些特性。（这里的观察器是指 SoWinExaminerViewer，下图所示。译者注）



单位节点

Inventor允许我们在多种不同单位下定义数据。Inventor使用“米”作为缺省单位，但我们也可以使用**SoUnits**节点来指定一个不同的度量单位。单位节点的行为有些像比例节点（scale node），它使用指定的单位对其后的形体进行按比例缩放。如果已经定义了其它的单位，那么**SoUnits**将根据其它单位来调整比例缩放系数。单位节点调整了缩放比例后，以前定义的单位节点将不再有效。

SoUnits节点只有一个域：

units (SoSFEnum)	定义当前度量单位。在场景中所有位于这个单位之后的形体都将使用此单位来定义。下面是可能的数值： SoUnits::METERS SoUnits::CENTIMETERS SoUnits::MILLIMETERS

SoUnits::MICROMETERS
SoUnits::MICRONS
SoUnits::NANOMETERS
SoUnits::ANGSTROMS
SoUnits::KILOMETERS
SoUnits::FEET
SoUnits::INCHES
SoUnits::POINTS
SoUnits::YARDS
SoUnits::MILES
SoUnits::NAUTICAL_MILES

如果不想使用上面预定义的单位来渲染场景的话，可以使用**SoUnits**节点将当前单位设置成原来的“米”单位，然后再定义一个缩放节点来将“米”缩放到想要的单位上。

绑定节点

材质和法线可以通过不同的方法绑定到形体节点上。本节的开始部分主要关注材质绑定（*material binding*），即讨论定义在**SoMaterial**节点中的当前材质是如何被映射到形体节点上的。因为法线绑定类似于材质绑定，所以我们首先讨论材质绑定。（本章的例 5-1 将演示如何使用法线绑定节点）

SoMaterialBinding节点包含有一个用来描述怎样将材质绑定到形体上的域。域值可以使用下列数值：

SoMaterialBinding::DEFAULT	对每个形体使用“最好的”绑定方式。大多数形体将这种绑定方式解释为 OVERALL。
SoMaterialBinding::NONE	无材质。
SoMaterialBinding::OVERALL	对整个形体使用当前材质的第一个数值。
SoMaterialBinding::PER_PART	将材质绑定到形体的不同“部分”上。“部分”的定义依赖于不同的形体。对于面集、立方体来说，“部分”表示的是其中的一个面。对于线集来说，“部分”表示其中的一个线段。对于圆柱来说，“部分”表示的是侧面，顶面，底面。
SoMaterialBinding::PER_PART_INDEXED	根据索引值将材质绑定到形体的不同“部分”上。
SoMaterialBinding::PER_FACE	将材质绑定到形体的每个面上。 binds one material to each face in the shape.
SoMaterialBinding::PER_FACE_INDEXED	根据索引值将材质绑定到形体的每一个面上。（只对索引顶点形体（indexed vertex shapes））
SoMaterialBinding::PER_VERTEX	将材质绑定到形体的每一个顶点上。
SoMaterialBinding::PER_VERTEX_INDEXED	根据索引值将材质绑定到形体的每一个顶点上。

每种形体节点解释绑定类型可能稍微有些不同。例如，**SoSphere**球体节点没有“部分”、面、或索引，所以有些绑定类型（PER_PART, PER_FACE, PER_VERTEX）对球体而言没有意义。我们可以将材质绑定节点所定义的绑定类型看作是为形体关于绑定方面的一个“暗示”。如果我们指定的绑定类型对一个特定的形体没有意义（例如对圆柱指定PER_FACE绑定类型），那么形体将按照对自己可以产生“最好”的效果来解释这些信息。（因为圆柱没有“面”，所以在这种情况下，PER_FACE类型的绑定将被解释成OVERALL类型）。在*Open Inventor C++ Reference Manual*中，可以查找到不同形体是如何解释不同的绑定类型。

假设我们对圆柱设置了 PER_PART 绑定类型。圆柱有三个“部分”（侧面、顶面、底面）。如果当前材质包含了三个材质数据－例如，橙色、紫色、黄色－那么，这些材质数据将会被用在圆柱的三个“部分”上，将产生一个橙色侧面、紫色顶面、和黄色底面的圆柱体。如果当前材质包含的数据多于“部分”数的话，如我们所猜测的那样，Inventor 将简单地忽略掉多于的材质。（如果当前材质列表中包含有五个材质数据，则圆柱将忽略掉最后的两个材质数据。）

如果当前材质包含的数据要少于绑定所需的话，Inventor 按照需要循环使用当前数据。例如，如果我们对一个立方的节点指定 PER_FACE 绑定类型，而当前材质列表中只包含有三个数值（紫罗兰色、长春花红色、兰绿色），则结果如下：

Face 1 紫罗兰色

Face 2 长春花红色

Face 3 兰绿色

Face 4 紫罗兰色

Face 5 长春花红色

Face 6 兰绿色

索引绑定

到目前为止，我们都是按照顺序来使用当前材质中的数据。然而，如果我们对一个索引顶点形体设置了PER_FACE_INDEXED 或PER_VERTEX_INDEXED绑定类型，或者对带有“部分”的形体使用PER_PART_INDEXED绑定类型，那么我们就可以按照一个新的顺序来使用当前材质数据。当使用这些绑定类型的时候，Inventor将引用形体节点（例如**SoIndexedFaceSet**, **SoIndexedLineSet**）的材质索引域（materials-index field）。Inventor将会按照我们任意指定的顺序在材质列表中索引定位数据，而不是只以第一个材质为起始，顺序使用材质列表。

举一个例子，考虑使用**SoIndexedFaceSet**节点来显示一个四面体。当前材质列表（在**SoMaterial**节点中）包含有下列数值：

材质列表

0 桃色

1 黄褐色

2 白色

materialIndex域（在**SoIndexedFaceSet**节点中）包含有下列数值：

材质索引

1

1

0

2

如果我们指定了 PER_FACE 绑定类型（无索引的），Inventor 将忽略 **materialIndex** 域，且将按照顺序循环使用材质列表：

Face 1 桃色

Face 2 黄褐色

Face 3 白色

Face 4 桃色

另一方面，如果我们指定了 PER_FACE_INDEXED 绑定类型，那么 Inventor 将按照 **materialIndex** 域中的索引值，从材质列表中提取数据，如下表所示：

Face 1 黄褐色

Face 2 黄褐色

Face 3 桃色

Face 4 桃色

使用索引方式是非常经济的，因为我们可以只使用一个单一的材质子集就可以达到各种目标和目的。

顶点绑定

Inventor 提供了两种类型的顶点绑定：PER_VERTEX 和 PER_VERTEX_INDEXED。当使用非索引的材质顶点绑定类型时（即 PER_VERTEX 类型。译者注），Inventor 将简单地从材质列表中按照顺序选择材质，然后将选择的材质分别赋值到形体节点的每一个顶点上。之后 Inventor 将在顶点间进行材质插值运算，并将材质覆盖在形体表面。（It then interpolates the materials between the vertices and across the faces of the shape.）

细微差别（高级部分）

SoMaterial 节点包含有六个域，每个域都可以保存多个数据。然而，保存在这六个域中的数据个数也许会不相等。例如，也许在环境光、漫反射、高光、自发光这些域中有五个数据，但在亮度域中只有两个数据，在透明域中只有一个数据。在这种情况下，Inventor 将会选择循环使用，直到和那个拥有最大数值个数（五个）的域相等为止。在只有少数数据的域中，域的最后那个数据将被重复使用直到循环结束。

当指定了 PER_VERTEX 绑定类型时，数值-1 对于在 **SoIndexedFaceSet** 节点（或者任

何其它索引形体节点）中的 **materialIndex** 域或 **normalIndex** 域来说，表示可以使用坐标顶点索引数据来绑定材质和法线。SO_END_LINE_INDEX、SO_END_FACE_INDEX 和 SO_END_STRIP_INDEX 这些预定义的常量用于这种规定。这种规定可以节约时间和内存，并且可以确保索引值匹配。当我们使用了一个“特殊”的坐标索引值（像 SO_END_FACE_INDEX）时，相对应的材质索引将被跳过，以便保证索引数组匹配。(When PER_VERTEX binding is specified, a value of -1 (the default) for the materialIndex field or the normalIndex field in an SoIndexedFaceSet (or any other indexed shape node) indicates to use the coordinate indices for materials or normals. The defined constants SO_END_LINE_INDEX, SO_END_FACE_INDEX, and SO_END_STRIP_INDEX can be used for this specification. This saves time and space and ensures that the indices match up. When you use a “special” coordinate index (such as SO_END_FACE_INDEX), the corresponding material index is skipped over so that the arrays of indices match)

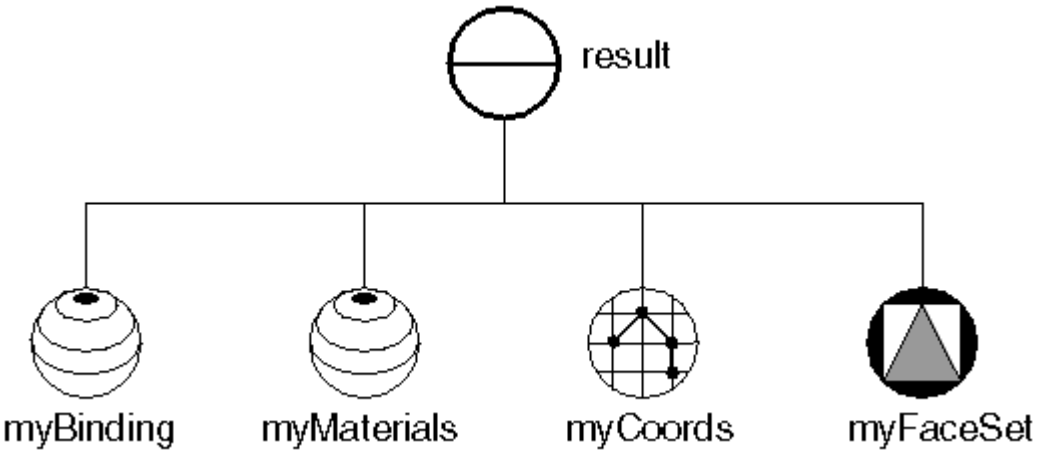
提示：为了获得更好的性能，我们应该使用 PER_FACE 或 PER_FACE_INDEXED 绑定类型，将一个材质节点和一个定了多个多边形的面集节点绑定在一起，而不应该使用 OVERALL 绑定类型，多个材质节点和多个面集节点绑定在一起。

使用材质绑定节点

例 5-5 将演示使用不同类型的材质绑定应用到例 5-2 所创建的十二面体上（在这里，我们忽略掉那些通用的代码）。图 5-8 展示了这个例子的场景图。当运行此例子的时候，可以输入一个数字来选择不同类型的材质绑定。

- 0 指定 PER_FACE (见彩图 8)
- 1 指定 PER_VERTEX_INDEXED (见彩图 9)
- 2 指定 PER_FACE_INDEXED (见彩图 10)

图 5-8 材质绑定例子的场景图



例 5-5 使用不同类型的材质绑定

```
// Which material to use to color the faces
// half red & half blue
static long materialIndices[12] = {
   0, 0, 0, 0, 0, 0,
   1, 1, 1, 1, 1, 1,
};

switch(whichBinding) {
  case 0:
    // Set up binding to use a different color for each face
    myBinding->value = SoMaterialBinding::PER_FACE;
    break;
  case 1:
    // Set up binding to use a different color at each
    // vertex, BUT, vertices shared between faces will
    // have the same color.
    myBinding->value = SoMaterialBinding::PER_VERTEX_INDEXED;
    break;
  case 2:
    myBinding->value = SoMaterialBinding::PER_FACE_INDEXED;
    myIndexedFaceSet->materialIndex.setValues(
             0, 12, materialIndices);
    break;
}
```

法线绑定

绑定法线具有与绑定材质几乎同样的语法形式。定义在**SoNormalBinding**节点中的法线绑定类型可以指定如何将当前法线应用到形体节点上。像**SoIndexedFaceSet**和**SoIndexedTriangleStripSet**这类的索引形体节点都有一个**normalIndex**域，这个域用来保存法线列表（在**SoNormal**节点中）索引值。如果指定的绑定类型不需要索引数据（例如，PER_VERTEX），那么就不再使用**normalIndex**域。

索引法线和索引材质主要不同点是索引法线不循环使用索引数据。如果使用索引法线，那么法线数必须和形体的面、顶点、或“部分”数精确匹配。如果法线数没有精确匹配的话，那么 Inventor 会自动产生缺省的法线（具体见下节的讨论）。我们必须要提供足够多的法线来绑定面、“部分”、或顶点。

自动计算法线

任何从**SoVertexShape**派生出来的形体节点都可以自动计算法线。因为自动计算法线的过程需要做大量的计算处理，所以我们建议读者启用自动缓存或者显式地启用渲染缓存，这

样中间计算的结果将被保存，并可以被重新使用（关于缓存的更多内容见第 9 章）。如果当渲染需要，并且

- 使用缺省类型的法线绑定
- 没有指定任何法线数据或者法线数与顶点数不同

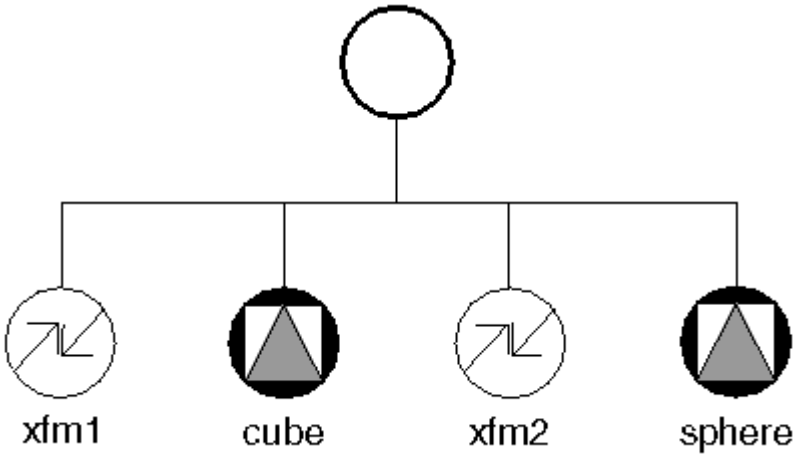
Inventor 将会自动生成法线。

当Inventor自动生成法线时，Inventor要查看**SoShapeHints**节点中的**creaseAngle**域值。折角（*crease angle*）定义为两个邻接平面的法线之间的角度。这个角度表示可以对两个邻接平面分离出法线的最大角度。例如，如果折角（*crease angle*）定义为 1 弧度，而两个邻接平面的法线之间的角度小于等于 1 弧度，那么这两个平面将共享相同的法线，这样做可以让两个平面之间共有的边界部分渐变的更加平滑。如果两个平面法线之间的角度大于 1 弧度，Inventor将对每个平面分别计算各自的法线，这样将产生一个折缝。如果我们想让物体显得更有棱角，可以将**creaseAngle**赋值为 0。如果我们想让物体显得更加平滑，可以将**creaseAngle**赋值为PI。

几何变换

不像其它属性节点那样，几何变换节点并不替换掉动作状态中的当前几何变换数据。相反，它们会对当前几何变换产生一个累加的效果。例如，在图 5-9 中，首先使用的是 *xfm1* 节点的变换，接着使用的是 *xfm2* 节点的变换。

图 5-9 几何变换节点的累加效果



立方体（cube）只受 *xfm1* 的几何变换影响。而球体（sphere）要同时受 *xfrm1* 和 *xfrm2* 这两个节点的几何变换影响。

SoTransform 节点

SoTransform 节点中包含有下列域：

translation (SoSFVec3f)　　在 *x, y, z* 轴平移。缺省值为[0.0 0.0 0.0]。

rotation (SoSFRotation)	根据一个轴和一个角度来定义旋转。缺省值为[0.0 0.0 1.0]， 0.0。
scaleFactor (SoSFVec3f)	在 *x, y, z* 轴上的缩放因子。缺省值为[1.0 1.0 1.0]。
scaleOrientation (SoSFRotation)	缩放方向的旋转定义。缺省值为[0.0 0.0 1.0], 0.0。
center (SoSFVec3f)	旋转和缩放的中心点位置。缺省值为[0.0 0.0 0.0]。

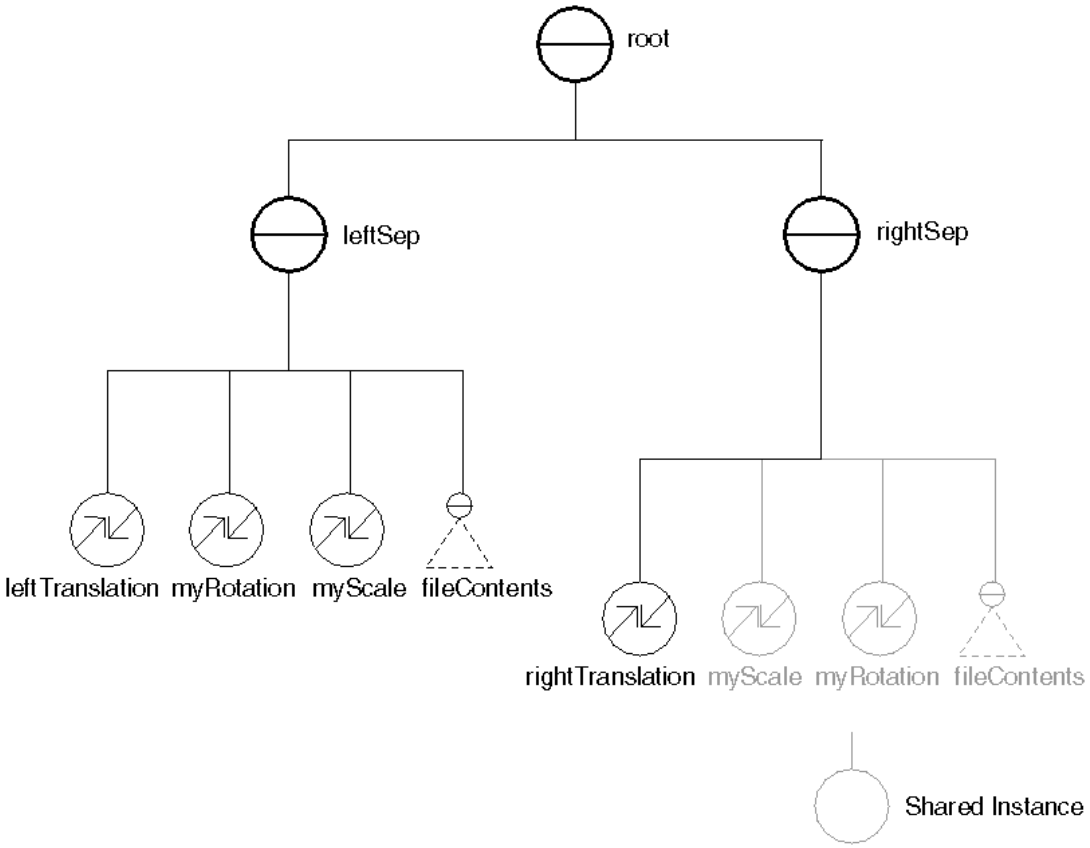
提示：如果我们只使用 SoTransform 节点中某一个域的话，可以使用相对应的“轻量级”版本。例如，如果只定义旋转，我们可以使用 **SoRotation** 和 **SoRotationXYZ** 这两个简化类。如果只定义平移，我们可以使用 SoTranslation。只定义缩放，我们可以使用 SoScale。

变换的顺序

在每个 SoTransform 节点中，最后那个域（center）会首先应用到形体对象上。使用域的顺序是中点（center），接着是缩放方向（scale orientaition）、缩放因子（scaling factor）、旋转（rotation），最后是平移（translation）。

图 5-10 和 例子“不同顺序的变换产生的效果” 将演示怎样因变换顺序的不同，而产生不同的结果。左边的教堂是先放大，再旋转，最后再平移。最接近物体的变换节点首先影响物体。因此我们需要从后往前阅读代码来理解变换的效果。右边的教堂是先旋转，然后再放大和平移的。例 5-6 给出了两类变换的代码。

Figure 5-10. 使用不同变换顺序的两组物体



Example 5-6. 修改变换顺序

```
#include <Inventor/Xt/SoXt.h>
```

```
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/SoDB.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoRotationXYZ.h>
#include <Inventor/nodes/SoScale.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTranslation.h>

main(int, char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   // Create two separators, for left and right objects.
   SoSeparator *leftSep = new SoSeparator;
   SoSeparator *rightSep = new SoSeparator;
   root->addChild(leftSep);
   root->addChild(rightSep);
   // Create the transformation nodes.
   SoTranslation *leftTranslation  = new SoTranslation;
   SoTranslation *rightTranslation = new SoTranslation;
   SoRotationXYZ *myRotation = new SoRotationXYZ;
   SoScale *myScale = new SoScale;

   // Fill in the values.
   leftTranslation->translation.setValue(-1.0, 0.0, 0.0);
   rightTranslation->translation.setValue(1.0, 0.0, 0.0);
   myRotation->angle = M_PI/2;   // 90 degrees
   myRotation->axis = SoRotationXYZ::X;
   myScale->scaleFactor.setValue(2., 1., 3.);

   // Add transforms to the scene.
   leftSep->addChild(leftTranslation);   // left graph
   leftSep->addChild(myRotation);        // then rotated
   leftSep->addChild(myScale);           // first scaled

   rightSep->addChild(rightTranslation); // right graph
   rightSep->addChild(myScale);          // then scaled
   rightSep->addChild(myRotation);       // first rotated
```

```
   // Read an object from file. (as in example 4.2.Lights)
   SoInput myInput;
   if (!myInput.openFile("temple.iv"))
      return (1);
   SoSeparator *fileContents = SoDB::readAll(&myInput);
   if (fileContents == NULL) return (1);

   // Add an instance of the object under each separator.
   leftSep->addChild(fileContents);
   rightSep->addChild(fileContents);

   // Construct a renderArea and display the scene.
   SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Transform Ordering");
   myViewer->viewAll();
   myViewer->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

第六章 文本

本章目标

在阅读完本章节后，我们将可以做下列事情：

- 向场景增加 2D 文本。
- 使用多种定制的外观（profiles）和字体，向场景增加 3D 文本。

本章将描述 2D 和 3D 文本的用法。Inventor 的 2D 文本技术为我们注释场景图像提供了一种简单、快捷的方法。为了具有更好的修饰性和灵活性，Inventor 可利用 3D 文本，为我们塑造 3D 字体外观提供了多种可能。本章将要介绍的关键概念包括：对齐（*justification*）、间距（*spacing*）、字体类型和大小、外观（*profiles*）。虽然也提及了 NURBS（非均匀有理 B 样条）曲线曲面的话题，但这个话题主要将在第 8 章中做全面的解释。

本章开始部分主要关注 2D 文本，同时介绍一些像对齐、间距、字体类型、和大小等 2D 和 3D 文本中通用的概念。本章的第二部分描述了 3D 文本的用法。3D 文本用法的主要概念是为文本定义一个横断面图（cross-sectional *profile*）。我们可以创建一个直线的、曲线的，或者二者相结合的轮廓图。

二维文本

SoText2节点定义了一个以 2D屏幕对齐（screen-aligned）方式显示的文本字符串。像其它的形体节点在渲染遍历到它们的时候，会将自身绘制到屏幕上一样。SoText2节点同样也会在渲染遍历到它的时候，以当前字体和颜色，将文本绘制到屏幕上。SoText2节点所使用的文本属性定义在SoFont节点中。这些属性包括字体类型和字体大小。二维文本不会因文本与照相机的距离发生改变而改变大小。

SoText2节点包含有下列域：

string (SoMFString)	被显示的字符串或字符串组。可以指定多个字符串
spacing (SoSFFloat)	两行文本间的行距。缺省值为 1.0。对于多行文本来说，某一行文本的顶端与下一行文本的顶端之间的垂直距离等于 spacing 乘以字体高度。
justification (SoSFEnum)	文本字符串如何与文本原点对齐。可以是左对齐（缺省）、右对齐、或中间对齐。

文本原点位于当前几何坐标系原点（0,0,0）处，并且当前几何变换可以影响文本的位置。文本显示的位置要根据指定的对齐方式，相对于文本原点而定。例如，如果指定右对齐，那么文本的右边将和文本原点对齐。

字体类型和大小

通过使用 SoFont 节点为场景中位于其后的文本节点（2D 和 3D）指定字体类型和大小。SoFont 节点包含有下列域：

name (SoSFName)	字体名称。具体名称请查看当前操作系统发行文档中列出的所有支持的字体类型。
size (SoSFFloat)	对于 SoText2 节点，此域表示以打印机点(一英寸= 72.2 打印机点)为单位的字体大小。对于 SoText3 节点，此域表示物体空间单位（object space units）内的字体大小。（缺省值为 10.0）。

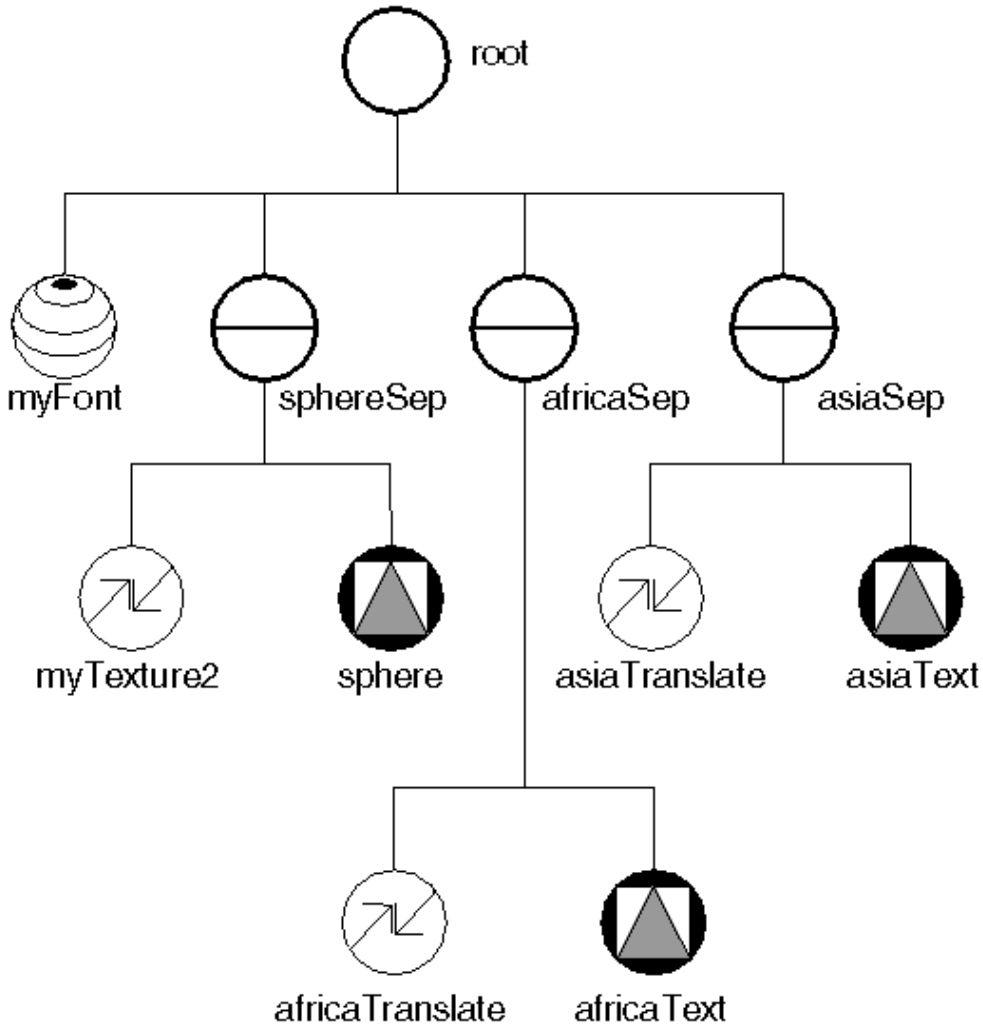
例如，可以定义一个 140 点大小，字体名称为 Courier bold italic 的字体。

```
SoFont *font = new SoFont;
font->name.setValue("Courier-BoldOblique");
font->size.setValue(140);
```

使用 2D 文本

例 6-1 渲染了一个地球，并使用 2D 文本标注非洲和亚洲大陆。使用 SoFont 节点定义了一个 24 点大小，名称为 Times Roman 字体作为当前字体。图 6-1 显示了这个例子的场景图。

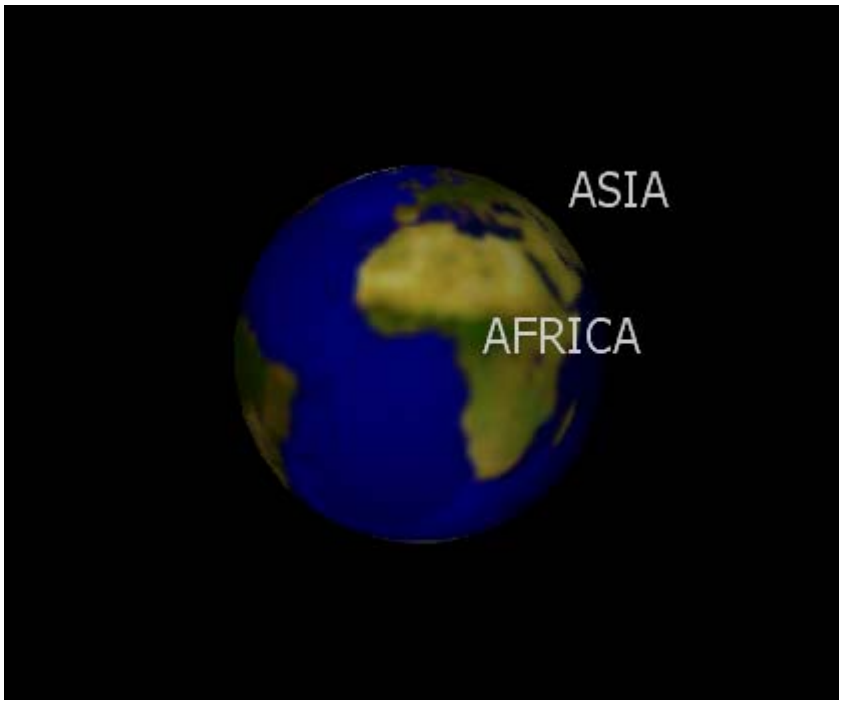
图 6-1 2D 文本例子



例 6-1．使用 2D 文本

```
#include <Inventor/nodes/SoFont.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoText2.h>
#include <Inventor/nodes/SoTexture2.h>
#include <Inventor/nodes/SoTranslation.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>

main(int argc, char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if(myWindow == NULL) exit(1);
```

```
SoGroup *root = new SoGroup;
root->ref();

// Choose a font.
SoFont *myFont = new SoFont;
myFont->name.setValue("Times-Roman");
myFont->size.setValue(24.0);
root->addChild(myFont);

// Add the globe, a sphere with a texture map.
// Put it within a separator.
SoSeparator *sphereSep = new SoSeparator;
SoTexture2  *myTexture2 = new SoTexture2;
root->addChild(sphereSep);
sphereSep->addChild(myTexture2);
sphereSep->addChild(new SoSphere);
myTexture2->filename = "globe.rgb";

// Add Text2 for AFRICA, translated to proper location.
SoSeparator *africaSep = new SoSeparator;
SoTranslation *africaTranslate = new SoTranslation;
SoText2 *africaText = new SoText2;
africaTranslate->translation.setValue(.25,.0,1.25);
africaText->string = "AFRICA";
root->addChild(africaSep);
africaSep->addChild(africaTranslate);
africaSep->addChild(africaText);
```



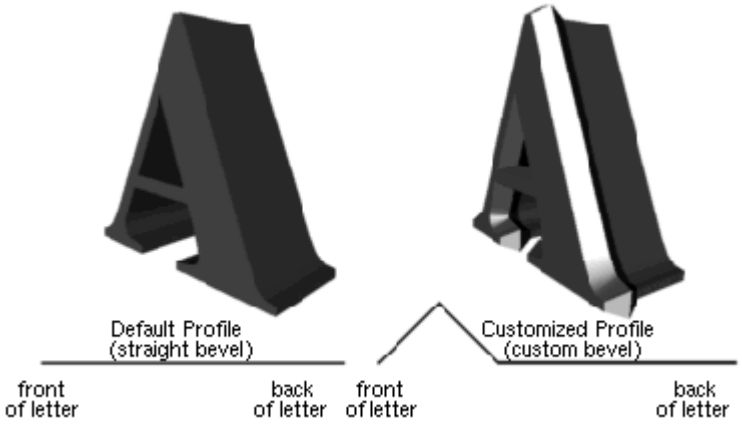

```
    // Add Text2 for ASIA, translated to proper location.
    SoSeparator *asiaSep = new SoSeparator;
    SoTranslation *asiaTranslate = new SoTranslation;
    SoText2 *asiaText = new SoText2;
    asiaTranslate->translation.setValue(.8,.8,0);
    asiaText->string = "ASIA";
    root->addChild(asiaSep);
    asiaSep->addChild(asiaTranslate);
    asiaSep->addChild(asiaText);

    SoXtExaminerViewer *myViewer =
             new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("2D Text");
    myViewer->setBackgroundColor(SbColor(0.35, 0.35, 0.35));
    myViewer->show();
    myViewer->viewAll();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

三维文本

与 2D 文本相反，3D 文本会随着与照相机的距离改变而改变大小，并且 3D 文本并不总是和屏幕对齐的。三维文本是有“深度”的。我们可以将组成 3D 字符的面按照直角方式拼接，也可以通过指定文本轮廓（text profile）使字符的边以斜角方式拼接。如图 6-2 中右面的图形，该图形显示了一个斜边字符“A”。

2D 文本的主要优势是显示速度比 3D 文本要快，而且因为 2D 文本总是和屏幕保持平行状态，所以它总是可读的。3D 文本的优势在于可缩放，且一般情况下会比 2D 文本显得更加美观。

图 6-2 为 3D 文本定制轮廓



SoText3节点包含有下列域：

string (SoMFString)	设置字符串或字符串组。我们可以指定多个字符串。
spacing (SoSFFloat)	文本之间的行距。缺省值为 1.0。对于多行文本来说，某一行文本的顶端与下一行文本的顶端之间的垂直距离等于 spacing 乘以字体高度。
justification (SoSFEnum)	文本字符串与文本原点间的对齐方式。可以是左对齐（缺省）、右对齐、或中间对齐。左对齐表示字符串中第一行第一个字符的正面左下角的位置在(0.0, 0.0, 0.0)处。接下来的每行起始于第一个字符以下的位置。右对齐表示字符串中最后结尾字符的右下角的位置在(0.0, 0.0, 0.0)处。接下来的每行终止于第一行的最后结尾字符以下的位置。中间对齐表示每行的中心位于(0.0, 0.0, 0.0)处。
parts (SoSFBitMask)	文本的可见“部分”（前面、侧面、背面、或全部）。缺省是前面。

3D 文本的“部分”

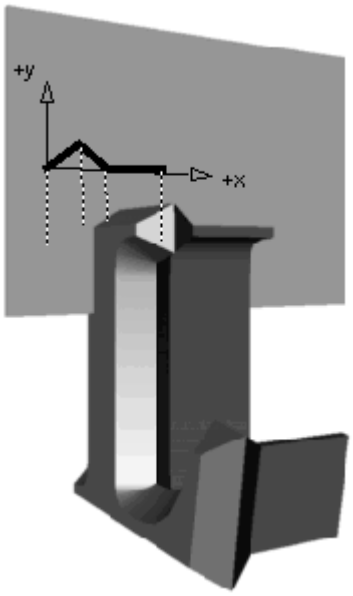
三维文本有三个“部分”:“前面”、“侧面”和“背面”。Inventor 使用当前材质渲染文本。如果材质绑定方式是 PER_PART，则“前面”使用第一个材质，“侧面”使用第二个材质，“背面”使用第三个材质。

提示：请注意，当我们启用 3D 文本的“侧面”和“背面”的时候，Inventor 需要绘制的多边形个数要比只显示文本“前面”的情况多 3 倍，所以这将会降低渲染性能。

轮廓

如图 6-3 所示，轮廓描述的是字符的横断面。轮廓绘制在自己的 2D 平面中。这个平面垂直于文本的正面（face of the text）。平面的原点位于字符的底边上。在这个坐标系下，大写字符是一个单位高。因此，轮廓的坐标范围必须在 0.0 到字体大小的 0.3 或 0.4 倍左右。

图 6-3 绘制文本轮廓的 2D 平面



直线轮廓

轮廓是通过使用当前轮廓坐标（*profile coordinates*）来构建的。如果轮廓是由一些直线段连接而成的，那么就需要使用**SoLinearProfile**节点来指定坐标是如何连接的。而轮廓坐标数据需要在**SoProfileCoordinate2**节点中指定。在场景中，**SoProfileCoordinate2**节点应该位于**SoLinearProfile**节点之前。（具体见例 6-3）

曲线轮廓

如果轮廓是曲线形式的，那么就需要**SoNurbsProfile**节点来指定如何使用坐标数据。如果对构建一个曲线轮廓感兴趣的话，我们需要首先阅读第 8 章节，学习使用NURBS曲线。坐标数据将根据曲线是无理的（nonrational）还是有理的（rational），分别保存在**SoProfileCoordinate2**节点或**SoProfileCoordinate3**节点中。（术语：“无理”和“有理”将在第 8 章做详细的解释）

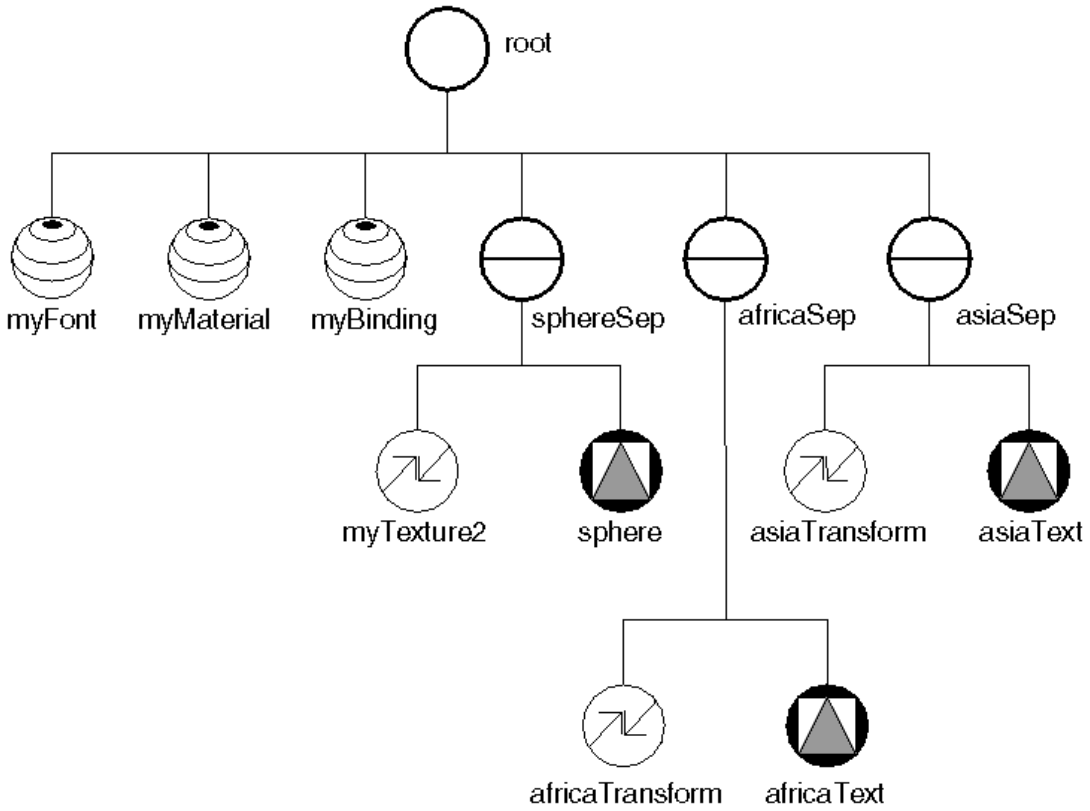
耦合轮廓（Linking Profiles）（高级内容）

如果文本轮廓是一个直线轮廓和曲线轮廓的结合体，那么我们可以将直线轮廓连接到曲线轮廓上。轮廓基类**SoProfile**包含有一个叫**linkage**的域，类**SoLinearProfile**和类**SoNurbsProfile**同时继承了这个域变量。这个域用来指定轮廓是否是START_FIRST（即文本的起始轮廓），是否是START_NEW（起始一个新轮廓，仅适用于NURBS拼接），或者是否是ADD_TO_CURRENT（将本轮廓添加到前一个轮廓中）。

3D 文本的简单用法

例 6-2 将演示 3D 文本的简单用法。例子渲染了一个地球，并使用 3D 文本标注非洲和亚洲大陆。使用 SoFont 节点定义了一个类型为 Times Roman 字体作为当前字体。图 6-4 是这个例子的场景图。

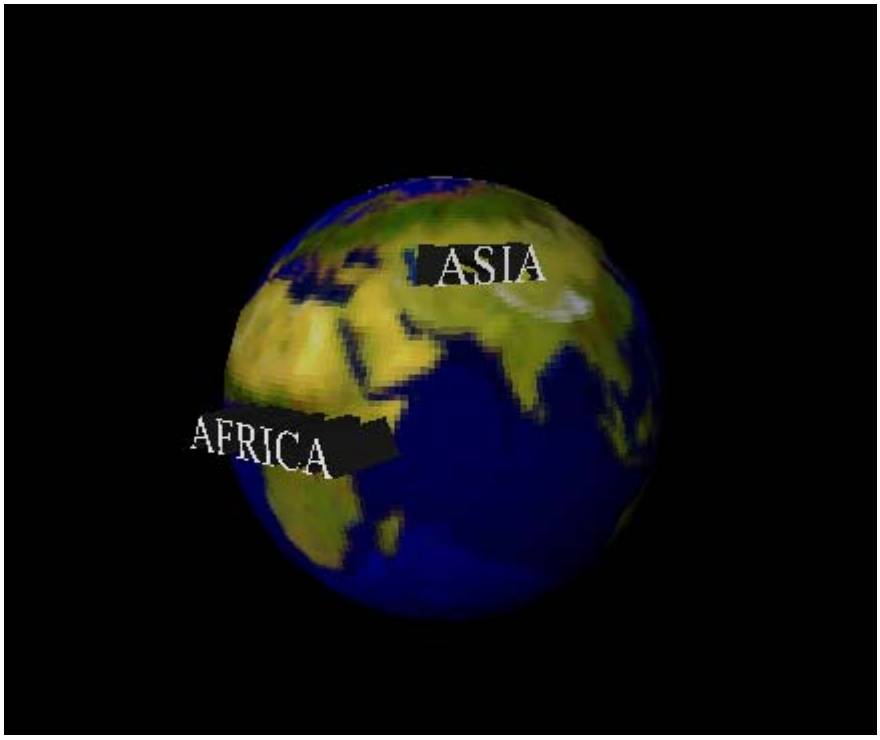
图 6-4 简单的 3D 文本例子场景图



例 6-2 使用 3D 文本

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoFont.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoMaterialBinding.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
```

```
#include <Inventor/nodes/SoText3.h>
#include <Inventor/nodes/SoTexture2.h>
#include <Inventor/nodes/SoTransform.h>


main(int, char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if(myWindow == NULL) exit(1);

   SoGroup *root = new SoGroup;
   root->ref();
```




```
   // Choose a font.
   SoFont *myFont = new SoFont;
   myFont->name.setValue("Times-Roman");
   myFont->size.setValue(.2);
   root->addChild(myFont);

   // We'll color the front of the text white, and the sides
   // dark grey. So use a materialBinding of PER_PART and
```

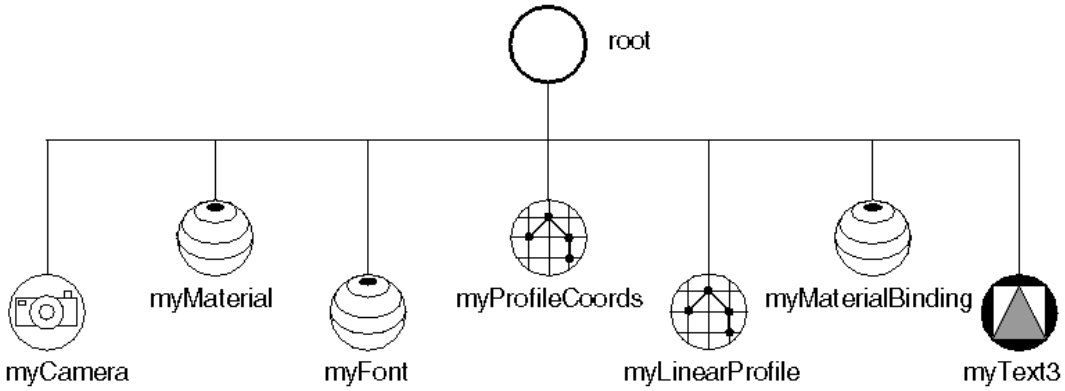
```
// two diffuseColor values in the material node.
SoMaterial        *myMaterial = new SoMaterial;
SoMaterialBinding *myBinding = new SoMaterialBinding;
myMaterial->diffuseColor.set1Value(0,SbColor(1,1,1));
myMaterial->diffuseColor.set1Value(1,SbColor(.1,.1,.1));
myBinding->value = SoMaterialBinding::PER_PART;
root->addChild(myMaterial);
root->addChild(myBinding);

// Create the globe.
SoSeparator *sphereSep = new SoSeparator;
SoTexture2  *myTexture2 = new SoTexture2;
root->addChild(sphereSep);
sphereSep->addChild(myTexture2);
sphereSep->addChild(new SoSphere);
myTexture2->filename = "globe.rgb";
// Add Text3 for AFRICA, transformed to proper location.
SoSeparator *africaSep = new SoSeparator;
SoTransform *africaTransform = new SoTransform;
SoText3 *africaText = new SoText3;
africaTransform->rotation.setValue(SbVec3f(0,1,0),.4);
africaTransform->translation.setValue(.25,.0,1.25);
africaText->parts = SoText3::ALL;
africaText->string = "AFRICA";
root->addChild(africaSep);
africaSep->addChild(africaTransform);
africaSep->addChild(africaText);

// Add Text3 for ASIA, transformed to proper location.
SoSeparator *asiaSep = new SoSeparator;
SoTransform *asiaTransform = new SoTransform;
SoText3 *asiaText = new SoText3;
asiaTransform->rotation.setValue(SbVec3f(0,1,0),1.5);
asiaTransform->translation.setValue(.8,.6,.5);
asiaText->parts = SoText3::ALL;
asiaText->string = "ASIA";
root->addChild(asiaSep);
asiaSep->addChild(asiaTransform);
asiaSep->addChild(asiaText);

SoXtExaminerViewer *myViewer =
         new SoXtExaminerViewer(myWindow);
myViewer->setSceneGraph(root);
```

```
    myViewer->setTitle("3D Text");
    myViewer->setBackgroundColor(SbColor(0.35, 0.35, 0.35));
    myViewer->show();
    myViewer->viewAll();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

3D 文本的高级用法（高级内容）

例 6-3 演示了 3D 文本另外一些有用的特征。例子为文本指定了一个使用 **SoProfileCoordinate2**和**SoLinearProfile**节点来定义的斜角横截面。文本使用两个不同的材质－一个用于文本的“前面”，一个用于文本的“侧面”和“背面”。字体节点使用Times Roman 字体。图 6-5 给出了例子的场景图。

图 6-5 3D 文本高级用法例子场景图



例 6-3 创建斜边 3D 文本

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoFont.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoLinearProfile.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoMaterialBinding.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoProfileCoordinate2.h>
#include <Inventor/nodes/SoText3.h>

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

```
{

   Widget myWindow = SoXt::init(argv[0]);
   if(myWindow == NULL) exit(1);

   SoGroup *root = new SoGroup;
   root->ref();

   // Set up camera.
   SoPerspectiveCamera     *myCamera = new SoPerspectiveCamera;
   myCamera->position.setValue(0, -(argc - 1) / 2, 10);
   myCamera->nearDistance.setValue(5.0);
   myCamera->farDistance.setValue(15.0);
   root->addChild(myCamera);
```

```
   // Let's make the front of the text white,
   // and the sides and back yellow.
   SoMaterial *myMaterial = new SoMaterial;
   SbColor colors[3];
   // diffuse
   colors[0].setValue(1, 1, 1);
```

```
colors[1].setValue(1, 1, 0);
colors[2].setValue(1, 1, 0);
myMaterial->diffuseColor.setValues(0, 3, colors);
// specular
colors[0].setValue(1, 1, 1);
colors[1].setValue(1, 1, 0);
colors[2].setValue(1, 1, 0);
myMaterial->specularColor.setValues(0, 3, colors);
myMaterial->shininess.setValue(.1);
root->addChild(myMaterial);

// Choose a font.
SoFont *myFont = new SoFont;
myFont->name.setValue("Times-Roman");
root->addChild(myFont);

// Specify a beveled cross-section for the text.
SoProfileCoordinate2 *myProfileCoords =
         new SoProfileCoordinate2;
SbVec2f coords[4];
coords[0].setValue(.00, .00);
coords[1].setValue(.25, .25);
coords[2].setValue(1.25, .25);
coords[3].setValue(1.50, .00);
myProfileCoords->point.setValues(0, 4, coords);
root->addChild(myProfileCoords);

SoLinearProfile *myLinearProfile = new SoLinearProfile;
long     index[4] ;
index[0] = 0;
index[1] = 1;
index[2] = 2;
index[3] = 3;
myLinearProfile->index.setValues(0, 4, index);
root->addChild(myLinearProfile);

// Set the material binding to PER_PART.
SoMaterialBinding *myMaterialBinding = new SoMaterialBinding;
myMaterialBinding->
         value.setValue(SoMaterialBinding::PER_PART);
root->addChild(myMaterialBinding);

// Add the text.
```

```
    SoText3 *myText3 = new SoText3;
    myText3->string.setValue("Beveled Text");
    myText3->justification.setValue(SoText3::CENTER);
    myText3->parts.setValue(SoText3::ALL);
    root->addChild(myText3);

    SoXtExaminerViewer *myViewer =
             new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Complex 3D Text");
    myViewer->show();
    myViewer->viewAll();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

第七章 纹理

本章目标

在阅读完本章节后，我们将可以做下列事情：

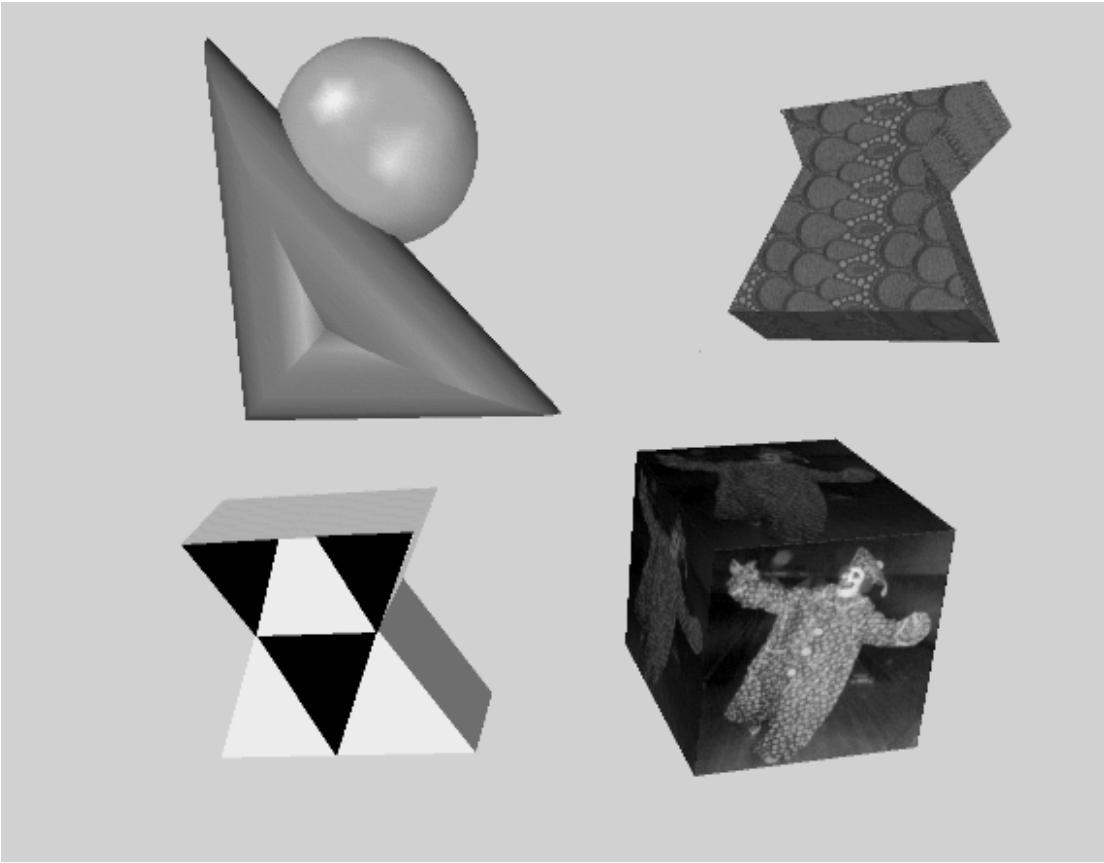
- 使用缺省纹理映射方式，将纹理“贴”到场景中的物体表面上。
- 通过直接指定纹理坐标，将纹理“贴”到场景中的物体表面上。
- 使用**SoTextureCoordinatePlane**和**SoTextureCoordinateEnvironment**这样的纹理坐标函数，将纹理映射到物体上。
- 在内存中创建一个纹理图，并将它映射到物体上。
- 通过重复循环使用纹理图形，将纹理包围住整个物体。
- 了解纹理如何影响物体的底色（underlying shaded color）。

本章将讨论如何使用纹理，使用纹理可以使场景更富有真实感和层次感。在 Inventor 中，我们可以创建一个 2D 的纹理贴图，然后将纹理图“贴”到 3D 物体表面上。依照定义，纹理的矩形片（The rectangular patch of texture）可以被延伸或压缩，以适应 3D 物体的需要。本章主要介绍的概念有：纹理贴图（*texture map*）、包装纹理（*wrapping textures*）、纹理模型（*texture model*）、纹理通道（*texture components*）、和环境映射（*environment mapping*）。

创建有纹理的对象

我们可以使用纹理创建出一个带有木纹的桌子、一个布满斑点的橘子、和一片草地。要想完成这些目标，我们必须首先要创建出木质纹理、橘皮纹理、和草地纹理，然后将它们应用到不同的物体上。图 7-1 对比演示了两类物体：右边的物体使用纹理映射，左边的物体不使用纹理映射。

图 7-1 纹理映射



什么是纹理贴图（*texture map*）

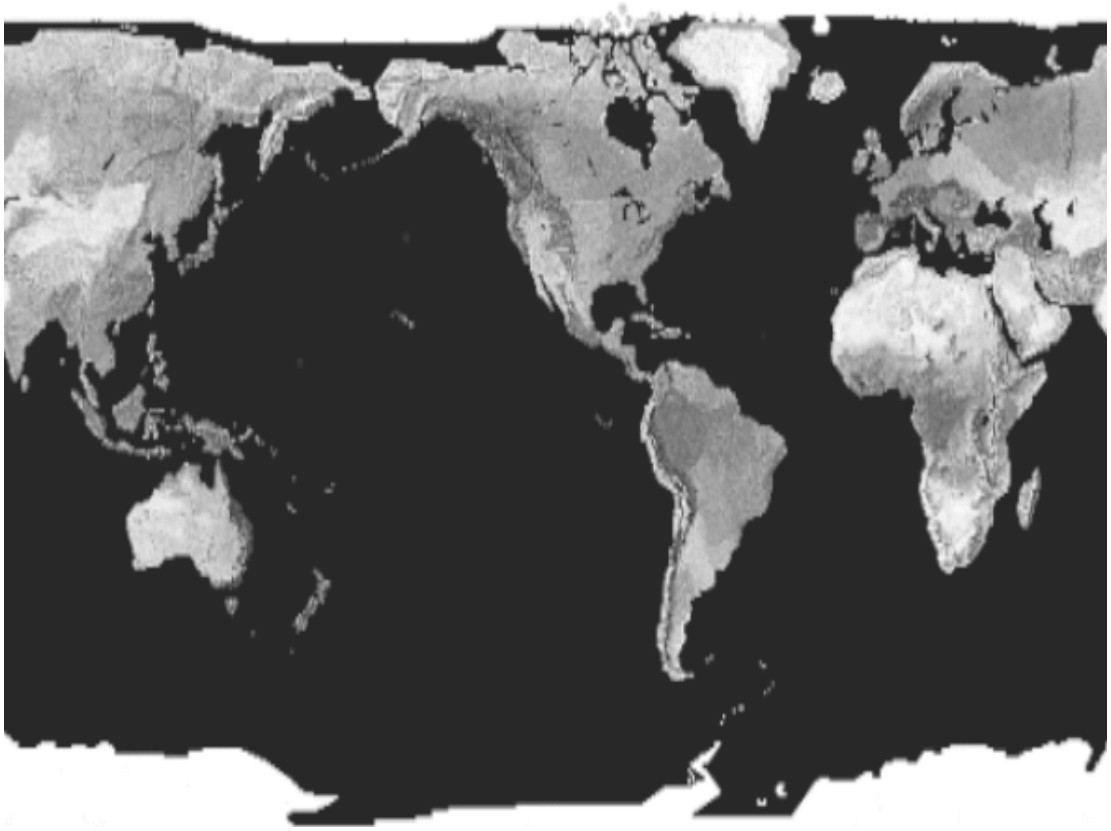
纹理贴图（*texture map*）是一个表示特定图案或纹理的 2D 像素信息数组。和 OpenGL 一样，Inventor 使用字符 s 表示水平纹理坐标，使用字符 t 表示垂直纹理坐标。如图 7-2 所示，纹理贴图是一个 1×1 大小的正方形，s 和 t 轴上的坐标范围是从 0.0 到 1.0。多边形的每个顶点都会分配一个纹理坐标（即可以由我们自己指定如何分配，也可以让 Inventor 自动来分配）。如果纹理中的像素不能和多边形中的像素精确匹配的话，Inventor 会利用一个过滤功能将纹理像素分配到物体上。我们即可以从文件，也可以从内存中读取纹理贴图。

用于纹理映射的节点

本节将讨论下列节点类：

SoTexture2	定义一个 2D 纹理贴图，为纹理映射设置关联参数。
SoTextureCoordinate2	为其后使用的顶点形体定义 2D 纹理坐标数据集。
SoTextureCoordinateBinding	定义当前纹理坐标如何绑定到其后的形体节点上。
SoTextureCoordinatePlane SoTextureCoordinateEnvironment	通过使用映射函数，将空间坐标映射到纹理坐标上。
SoTextureCoordinateDefault	关闭以前所有的纹理坐标函数，让其后的所有形体节点都使用它们自己的缺省纹理坐标。
SoTexture2Transform	定义一个纹理贴图的 2D 几何变换

图 7-2 纹理坐标



SoComplexity节点带有一个叫做**textureQuality**的域，这个域和纹理映射也有关系。域值的范围是从 0.0 到 1.0。0.0 表示要最快的渲染速度，1.0 表示要显示最好的纹理。（通常情况下，我们要在速度与纹理品质这两方面寻求平衡）。这个域的缺省值为 0.5。

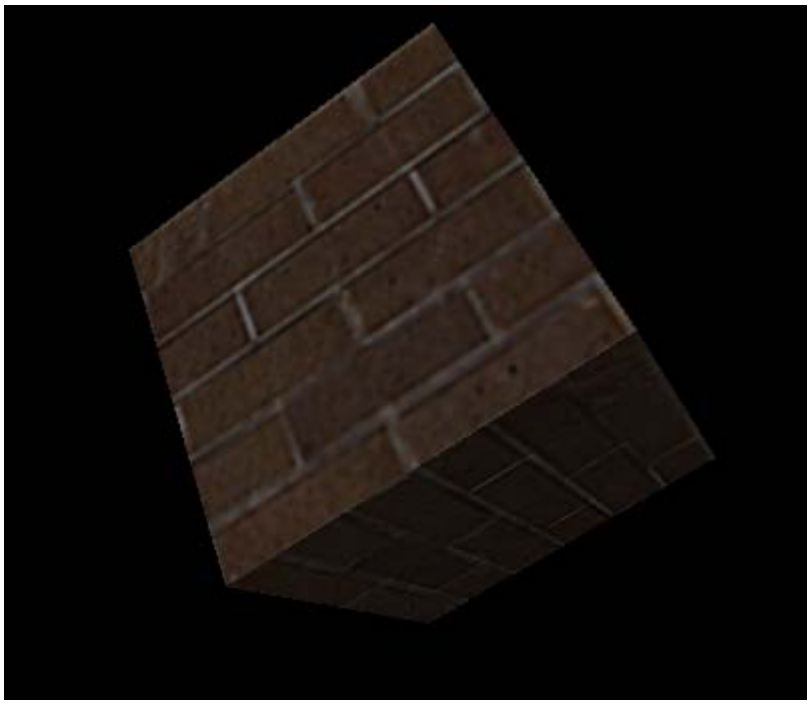
使用缺省纹理

尽管我们有多种方法可以影响纹理将如何“贴”到物体上，但使用纹理最简单的方法就是使用纹理的缺省值。如果我们想使用纹理，我们只需要定义一个**SoTexture2**节点（为纹理）和一个形体节点（目标物体）。例 7-1 举例说明了这种方法，例子显示了一个带有纹理的立方体。

例 7-1 使用缺省纹理值

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTexture2.h>

main(int , char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
```

```
    if(myWindow == NULL) exit(1);

    SoSeparator *root = new SoSeparator;
    root->ref();

    // Choose a texture
    SoTexture2 *rock = new SoTexture2;
    root->addChild(rock);
    rock->filename.setValue("brick.1.rgb");

    // Make a cube
    root->addChild(new SoCube);

    SoXtExaminerViewer *myViewer =
             new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Default Texture Coords");
    myViewer->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```



提示：如果有多个节点同时使用相同的纹理，那么应该让纹理节点位于所有节点都可以使用到它的位置上。如有可能，应该首先将要共享同一纹理的所有节点分成一组，然后再共享材质纹理。这样做的原因是因为切换纹理的代价是非常昂贵的。

关键概念

本节将讨论一些特殊的方法，这些方法可以改变纹理应用在物体上的映射方式，这些改变包括：

- 纹理如何包围物体。
- 纹理如何影响物体的底色。

另外，本节还将讨论如何将纹理图的像素点保存到内存中。

纹理包围物体

纹理坐标在每一维上都是位于从 0.0 到 1.0 之间的范围内(见“什么是纹理贴图”章节)。那么，如果有的多边形的纹理坐标范围是从 0.0 到 2.0 之间时，Inventor 将如何处理呢？。这种情况下，我们可有下列的选择：

- 即可以重复使用纹理，直到完全覆盖物体表面（或通过伸展纹理来覆盖物体）
- 也可以重复使用纹理贴图的最后一行像素，直至覆盖住剩余的表面（这种方式叫做纹理箝入（*clamping*））。

图 7-3 显示了这两种包围类型的示意图。左边的圆柱沿着高度方向和围绕圆周方向重复使用了两次纹理贴图。右边的圆柱通过把**wrapT**设置成CLAMP，在顶端进行了纹理箝入的操作。“SoTexture2 节点”章节中将详细讨论**wrapS**和**wrapT**域。

纹理如何影响底色（高级内容）

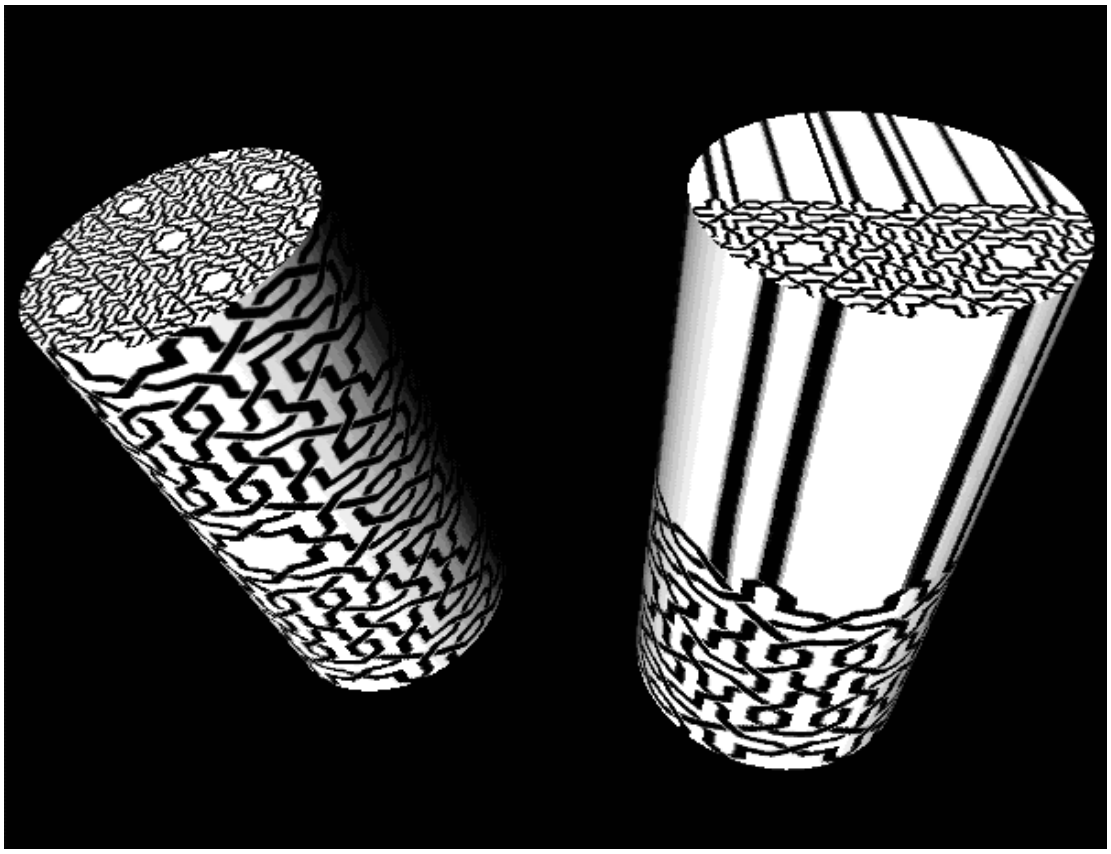
Inventor 有三种纹理模型可供我们使用（见“SoTexture2 节点”）。纹理贴图将根据不同的纹理模型，以不同的方式来影响多边形的底色。纹理模型有下列三种：

MODULATE 纹理颜色乘以底色（缺省模型）。如果纹理颜色带有 alpha 值，那么 alpha 值将用来调整物体的透明度。

DECAL 使用纹理颜色替换底色。如果纹理颜色带有 alpha 值，那么 alpha 值用来指定纹理的透明度，这样就允许物体的颜色透过纹理显示出来。

BLEND 将底色和一个常量颜色进行混合运算，结果作为纹理亮度。

图 7-3 纹理包围物体



MODULATE 模型可以被用于任何类型的纹理文件。BLEND 模型只能用于带有一个或两个颜色成分的纹理文件。DECAL 模型只能用于带有三个或四个颜色成分的纹理文件。具体见“**纹理通道**”。

提示：在物体带有高亮度材质的情况下，MODULATE 模型的显示效果会更好。因为纹理亮度是小于等于 1.0 的，它是要和物体底色相乘的（MODULATE works best on bright materials because the texture intensity, which is less than or equal to 1.0, is multiplied by the shaded color）。

彩图 11 至彩图 14 展示了每种纹理模型的例子。彩图 11 显示的是无纹理的场景。彩图 12 采用的是 MODULATE 纹理模型，建筑物的颜色是纹理与材质颜色的综合。彩图 13 采用的是 DECAL 纹理模型，建筑物的颜色完全受纹理贴图所决定。彩图 14 采用的是 BLEND 纹理模型，建筑物的颜色是材质颜色与混合色（金色）的混合。

提示：如果想在物体上显示一个亮绿色圆点花样的布料纹理，可以先产生一个带有白色圆点花样的黑白纹理贴图，然后使用混合色为绿色的 BLEND 纹理模型。

在OpenGL参考手册（*OpenGL Reference Manual*）中**glTexEnv()**函数给出了计算公式，这些公式用来计算被纹理化的物体最终的颜色。

提示：如果当前使用的纹理模型是MODULATE，我们可以将纹理贴图的四边设置成宽度为一个像素的白边，且将**wrapS**和**wrapT**设置成CLAMP，这样一来，在纹理贴图范围之外的地方将使用物体本身的颜色显示。

存储纹理图像（高级内容）

我们可以从文件或者内存中读取纹理贴图。Inventor 开发平台支持什么类型的图像文件格式，请查看 Inventor 发行文档。

我们可以将纹理贴图存成一个**SoSFImage**节点，然后在**SoTexture2**节点的**image**域上指定这个贴图。本节将详细讨论如何在内存中存储纹理贴图。纹理贴图（即可以保存在文件中，也可以保存在内存里）可以包含一到四个数据通道（components）。

纹理通道

纹理可以是下列四种类型：

- 单通道纹理（*One-component texture*）－ 只包含亮度数据。这类纹理经常被当作亮度图（*intensity map*）。例如，如果要渲染一幅山脉图像，我们可以根据一个单通道纹理的数据，变化带有常量颜色的多边形的亮度，这样可以让图像显得更加真实。
- 双通道纹理（*Two-component texture*）－ 包含亮度和 alpha(透明度)数据。例如，我们可以创建一个长满树叶的绿树，树叶是由从暗绿到亮绿变化的多边形组成。我们还可以变化树叶边缘的透明度，这样就可以让位于树叶边缘后面的物体也能显示出来。
- 三通道纹理（*Three-component texture*）－ 包含红色、绿色、蓝色数据。这种纹理是一种 RGB 类型的图像文件。通常用于显示照片或表现砖块、混凝土和布料的纹理。
- 四通道纹理（*Four-component texture*）－包含红色、绿色、蓝色和 alpha(透明度)数据。这种纹理类似于 RGB 三通道纹理，但它同时还包含透明度信息。我们可以利用前面讨论的双通道纹理所采用的技术，使用四通道纹理创造出一棵深秋时节华美的新英格兰枫树。

提示：单、双通道纹理通常要快于三、四通道纹理，因为它们只需要比较少的计算量。

将纹理存储到内存中

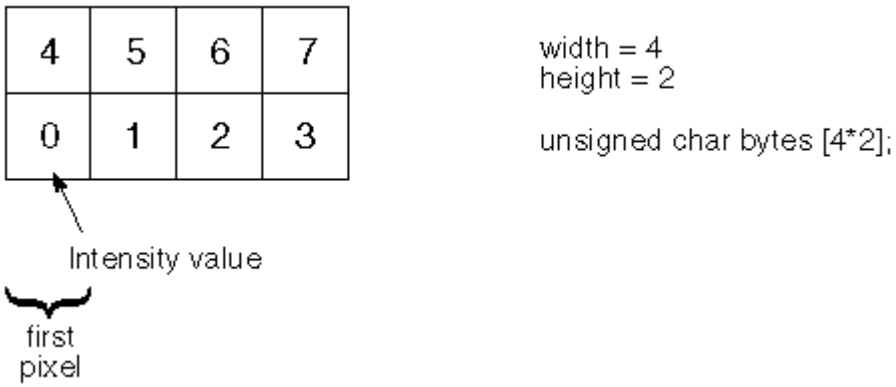
可以使用**setValue()**方法将数据赋值给**SoSFImage**。这个方法需要我们指定下列参数：纹理的大小（宽度 × 高度，以像素为单位）、纹理的通道数、每个像素通道数据（使用unsigned chars数组来表示，数据范围在 0 到 255 之间）。

对于单通道纹理来说，数组中每个字节保存一个像素的亮度值，如图 7-4 所示，第 0 号字节位于像素表中左下角，在每行数组里，字节编码是按照从左到右的顺序编码的。

例如，下面的代码将保存一个单通道纹理

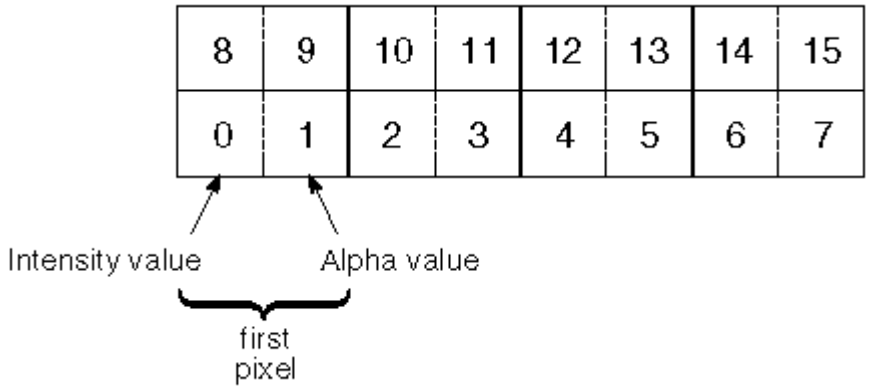
```
SoTexture2 *textureNode = new SoTexture2;
// A 3-by-2 array of black and white pixels; the array is
//upside-down here (the first pixel is the lower left corner)
unsigned char image [] = {
   255, 0,
   0, 255,
   255, 0
};
//Set the image field:
textureNode->image.setValue(SbVec2s(3,2), 1, image);
```

图 7-4 存储在内存中的单通道纹理格式



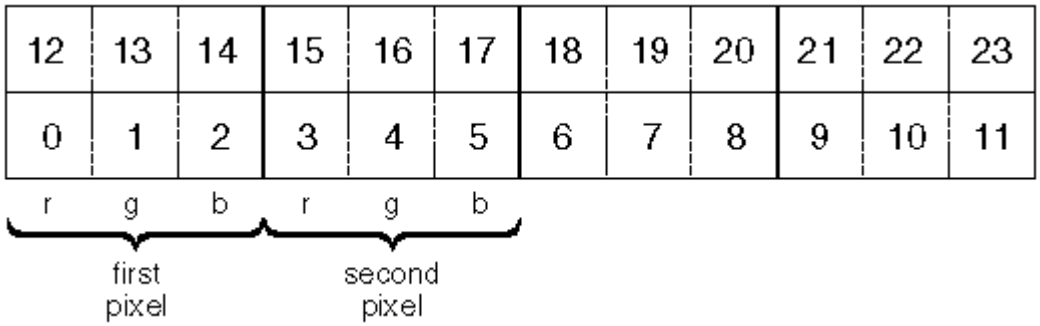
对于双通道纹理，数组中第 0 个字节是第一个像素的亮度值，第 1 个字节是第一个像素的 alpha(透明度)数据。第 2，3 个字节保存第二个像素的纹理信息，依次继续。（见图 7-5 所示）

图 7-5 存储在内存中的双通道纹理格式



三通道纹理要用三个字节保存一个像素的纹理信息。以第一个像素为例，第 0 个字节包含红色数据，第 1 个字节包含绿色数据，第 2 个字节包含蓝色数据（见图 7-6）。四通道纹理要用四个字节保存一个像素的纹理信息（红色、绿色、蓝色和 alpha 数据）。

图 7-6 存储在内存中的三通道纹理格式



SoTexture2 节点

SoTexture2节点用来为纹理贴图指定图像，定义纹理如何包围住物体，指明使用何种纹理模型。

SoTexture2节点包含的域：

filename (SoSFName) 指定当作纹理贴图的文件名称。请查看 Inventor 发行文档中支持何种类型的图形文件格式。我们既可以通过一个文件名称，也可以通过一个 image（见下个域）来指定纹理贴图。这两种方式是等效的。

如果指定了 filename 域，那么 Inventor 将从图形文件中读取数据，然后将图形文件中的像素信息设置在 image 域中。如果指定了 image 域，那么 filename 域将为“”（空字符串；缺省值）。通过这种方式，Inventor 就可以确认到底是那一个域用来指明纹理数据。

image (SoSFImage) 指定纹理贴图中的颜色和像素数

wrapS 指定纹理图像在 s（水平）和 t(垂直)方向包围物体的方式。可以是

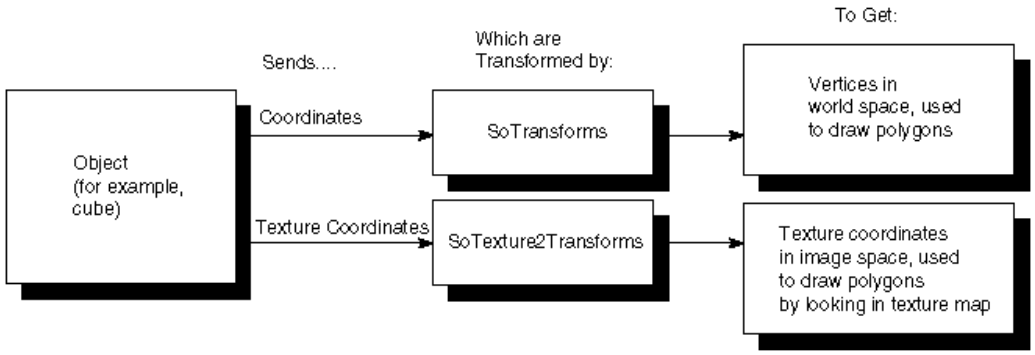
(SoSFEnum),wrapT (SoSFEnum)	下列数值： REPEAT 表示重复使用纹理图像直至填满整个物体（缺省方式） CLAMP 表示重复使用最后一行的像素
model (SoSFEnum)	指定所要使用的纹理模型。可以是下列数值： MODULATE 纹理颜色乘以物体底色（缺省方式） DECAL 使用纹理颜色代替物体底色 BLEND 将物体底色与一个指定的混合色进行混合(见 blendColor 域)
blendColor (SoSFColor)	当使用 BLEND 纹理模型的时候，这个域指定一个用来混合的颜色

SoComplexity节点中的**textureQuality**域可以控制纹理的过滤品质（quality of filtering）。域值为 0.0，表示完全禁止使用纹理。域值为 1.0，表示使用最高品质的纹理。缺省域值为 0.5。

纹理贴图的变换

我们可以在形体节点之前插入一个**SoTexture2Transform**节点，这样做可以对纹理贴图进行变换。**SoTexture2Transform**节点对纹理坐标变换具有累积效果。如图 7-7 所示，**SoTexture2Transform** 和 **SoTextureCoordinate2** 节点之间的关系类似于 **SoTransform** 和 **SoCoordinate**节点之间的关系。

图 7-7 SoTexture2Transform节点与纹理坐标的关系



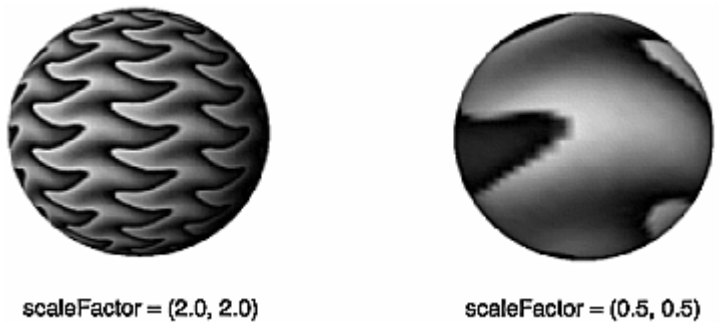
SoTexture2Transform节点包含有下列域：

translation (SoSFVec2f)	设置物体纹理坐标的平移变换
rotation (SoSFFloat)	设置物体纹理坐标的旋转变换。旋转角度以弧度为单位
scaleFactor (SoSFVec2f)	设置物体纹理坐标的缩放因子。缩放因子将乘以物体的 s 和 t 坐标。因此（2.0,2.0）的缩放因子将使纹理图像在物体上变小（见图 7-8 左边的图像）。（0.5,0.5）将使纹理图像变大（见图 7-8 右边的图像）。
center	设置旋转和缩放的中心。缺省值是（0.0,0.0），位于纹理贴图的左下角

(SoSFVec2f)　　位置。

在图 7-8 中，左边的球有一个（2.0,2.0）的纹理**scaleFactor**缩放因子，这样，纹理将在s和t轴方向被重复使用两次。右边的球有一个（0.5,0.5）的纹理**scaleFactor**缩放因子，所以在s和t轴方向只使用了一半的纹理贴图。

图 7-8 3D 纹理贴图的缩放因子所产生的不同效果



将纹理映射到物体上

我们可以选择三种技术将 2D 纹理空间映射到 3D 物体空间上：

1. 使用缺省纹理坐标。本节的后面部分将描述不同形状的物体是如何使用不同的方式将纹理映射到物体上。
2. 对于所有从**SoVertexShape**派生出来的物体来说，我们可以为它们显式地指定纹理坐标。我们可以首先创建一个**SoTextureCoordinate2**节点，然后在这个节点中为物体上的每个顶点指定对应的纹理坐标。
3. 使用纹理坐标函数（texture-coordinate functions）将纹理映射到物体上：
   SoTextureCoordinatePlane
   SoTextureCoordinateEnvironment

第一、三种技术是 Inventor 自动完成的，因此比较容易使用。第二种技术需要我们显式地给出纹理坐标，比较难用。但是，第二种技术允许我们对纹理做更多更直接的控制。下面的章节将详细讨论这些技术。

使用缺省纹理映射

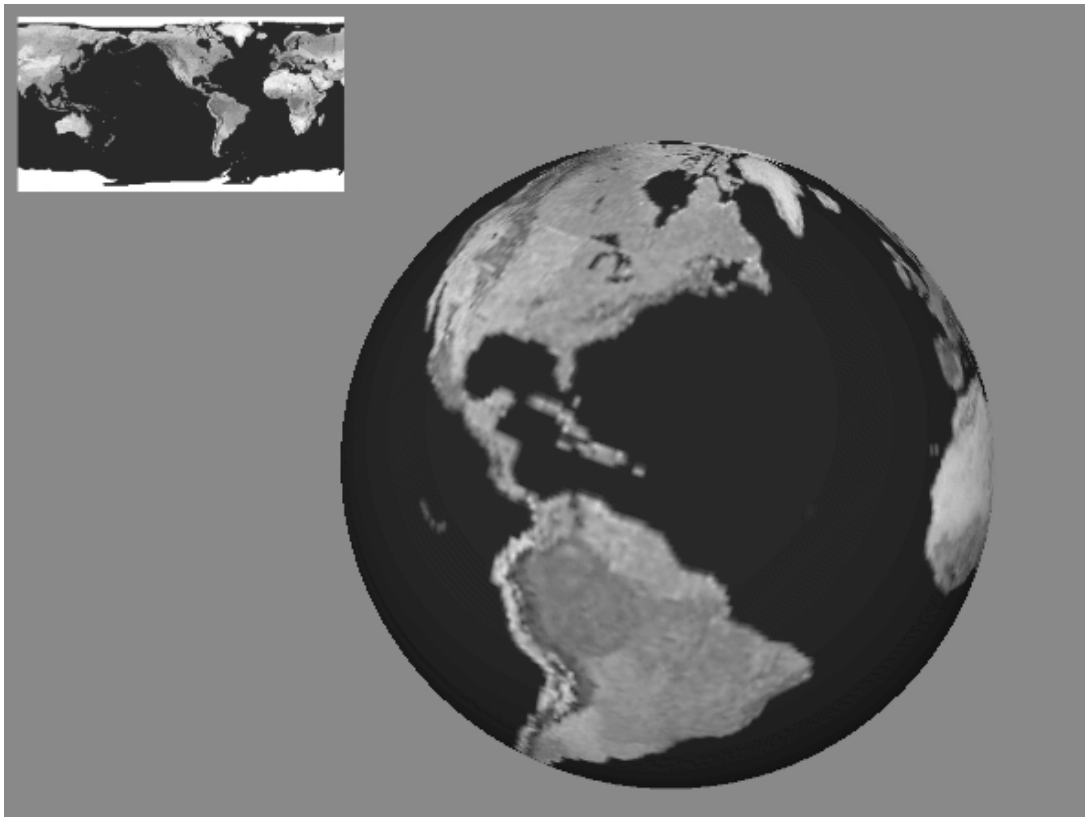
对于所有从**SoVertexShape**派生出来的形体而言，Inventor都使用相同的技术来为它们创建缺省的纹理坐标。首先，Inventor计算物体的包围盒。然后，将包围盒的最长边当作纹理的水平（s）轴，将第二长的边当作纹理的垂直（t）轴。s轴坐标值在 0.0 到 1.0 之间，对应于包围盒里最长边的一端到另一端。t值的范围在 0.0 和n之间，此处的n等于第二长边与第一长边的长度比值(这样做就可以避免纹理在沿着包围盒的最长边的方向上发生变形)。（**当然在另外的轴上有可能会发生纹理变形，这是不可避免的。Inventor这样做，最大程度上保证纹理不发生变形。译者注**）

对于所有不是从**SoVertexShape**派生出来的形体来说，不同的形体使用不同的方法产生缺省纹理坐标。这些形体包括SoCone, SoCube, SoCylinder, SoNurbsSurface, SoSphere, 和 SoText3 。下面的段落将详细讨论这些形体缺省的纹理映射方式。

SoSphere

假如，在场景中包含一个**SoTexture2**节点，紧随其后还有一个**SoSphere**节点，那么Inventor将使用缺省的纹理坐标将纹理映射到球体上。纹理将从球体的后面开始，按照逆时针的方向，包围覆盖住球体的整个表面（见图 7-9）。纹理围绕球体一周后，和自己相连接。如果纹理模型是非重复的，那么球体上有可能会出现一个连接的缝隙。

图 7-9 球体的缺省纹理映射

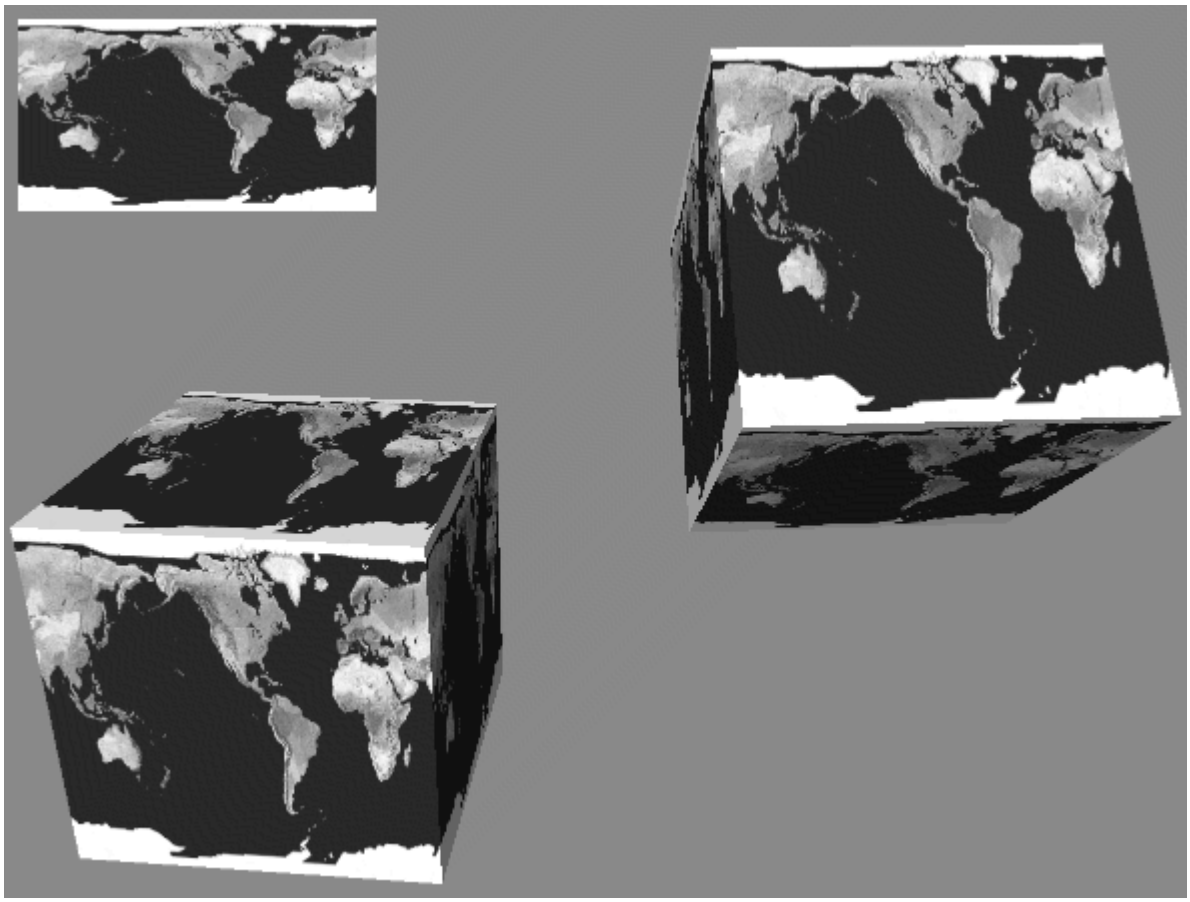


提示：提高简单形体的复杂度，可以改善物体纹理的外观。

SoCube

当一个纹理使用缺省的纹理坐标应用到**SoCube**节点上时，整个纹理将被“贴”到立方体的每个面上。在立方体的前，后，左，右平面上，纹理是正面朝上（right-side up）的。如果将方体面向我们倾斜，在立方体“上平面”上的纹理将以正面朝上方式显示。如果将立方体背离我们倾斜，在立方体“下平面”上的纹理将以正面朝上方式显示。（见图 7-10）。（这里的正面朝上（right-side up），是指纹理的t（垂直）轴方向应该朝向Inventor Y轴的正方向，译者注）

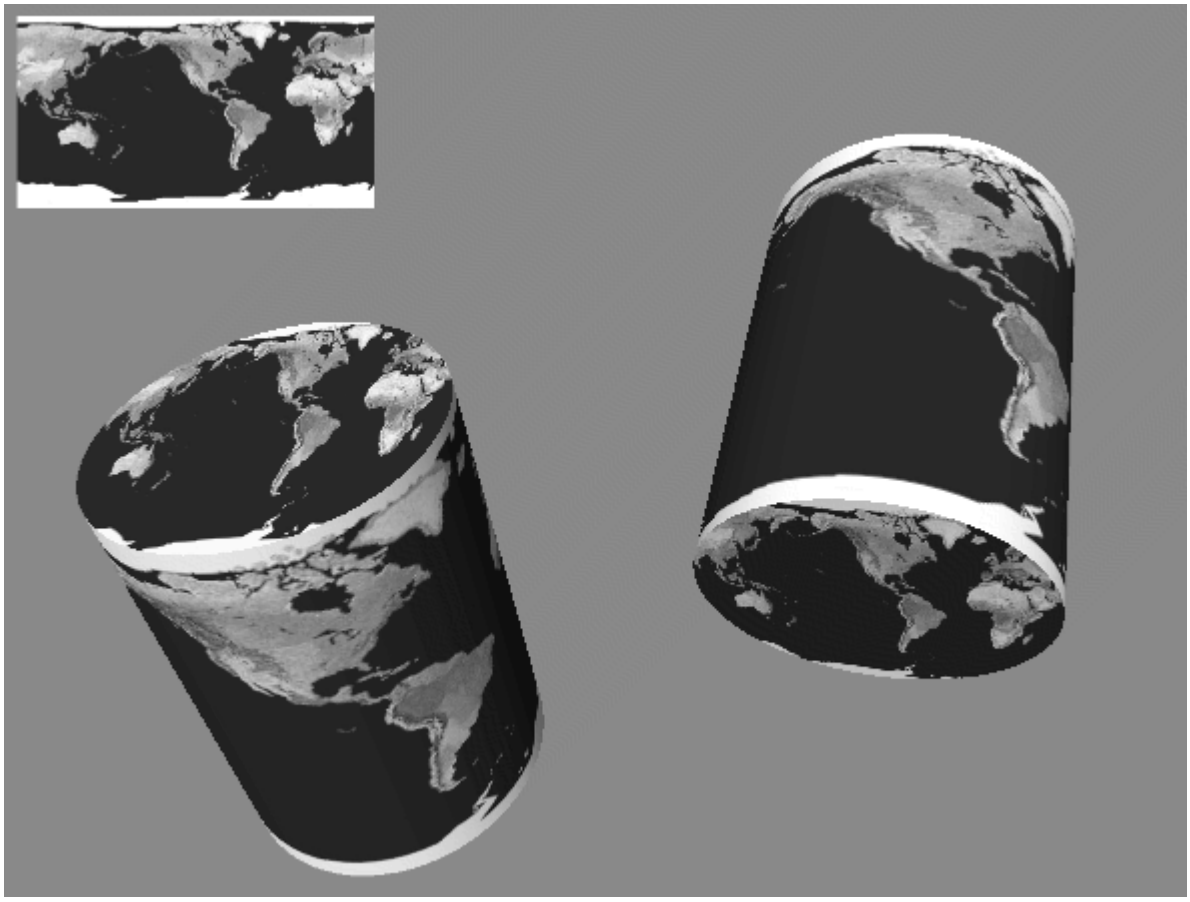
图 7-10 立方体的缺省纹理映射



SoCylinder

当一个纹理使用缺省的纹理坐标被应用到 **SoCylinder** 节点上时，纹理从-z 轴开始，沿着逆时针方向“贴”到圆柱体的侧面上。在纹理图像的中心位置截取一片圆形图像，然后将截取下来的图形“贴”到圆柱体的上、下底面上。当沿着 +z 轴方向观察圆柱体时，如果圆柱的顶端靠向我们，那么在圆柱体顶面上的纹理将以“正面朝上”的方式显示。如果圆柱的顶端远离我们，那么在圆柱体底面上的纹理将以“正面朝上”的方式显示。（见图 7-11）

图 7-11 圆柱体的缺省纹理映射

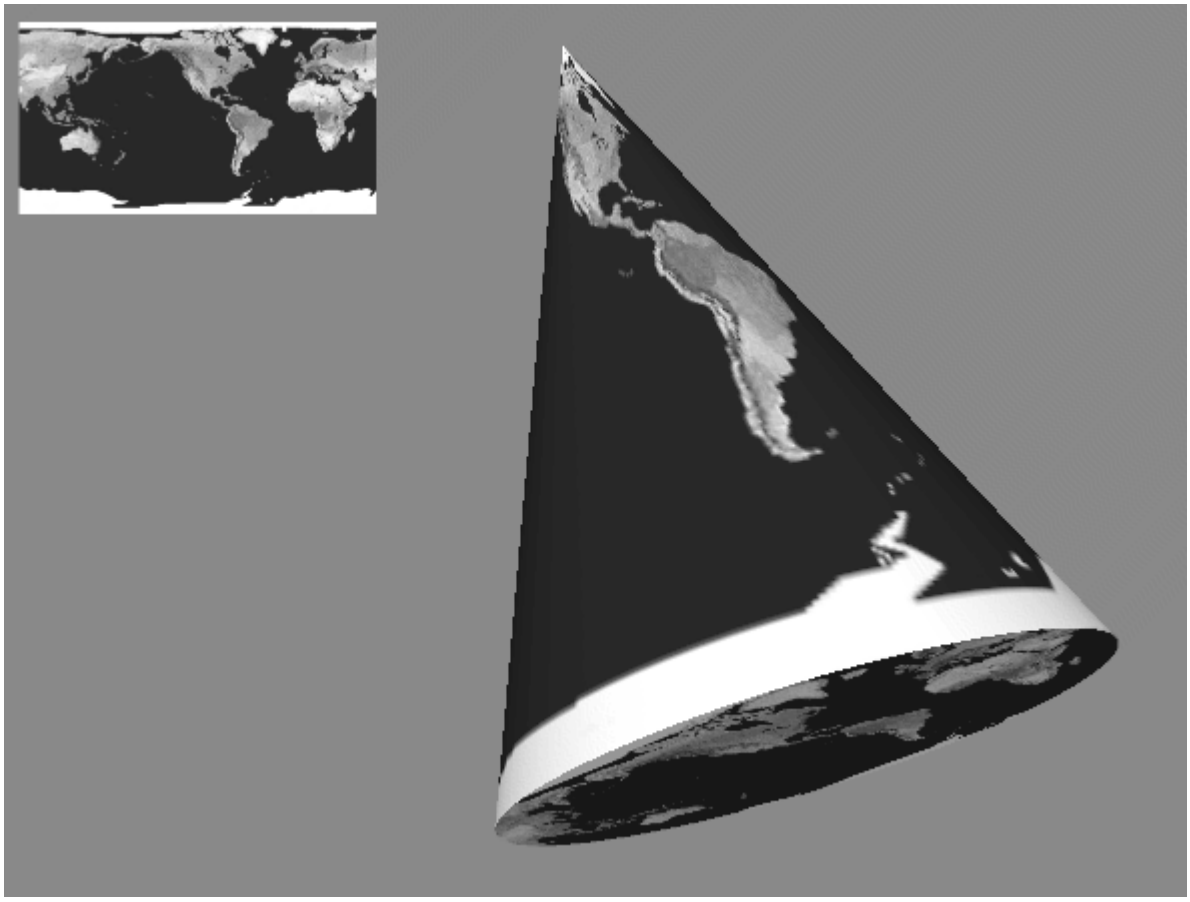


SoCone

当一个纹理使用缺省的纹理坐标被应用到 **SoCone** 节点上时，纹理从圆锥的后面开始，沿着逆时针方向“贴”到圆锥体的侧面上。纹理围绕圆锥体一周和自己相连接。如果纹理模型是非重复的，那么圆锥体上有可能会出现一个连接的缝隙。在纹理图像的中心位置截取一片圆形图像，然后按照和圆柱体一样的方式将截取下来的图形“贴”到圆锥体的下底面上。

提示：增加圆锥体的复杂度是尤为重要的，因为圆柱体顶端的纹理映射的效果好坏，会直接影响到整个圆锥体的映射效果。

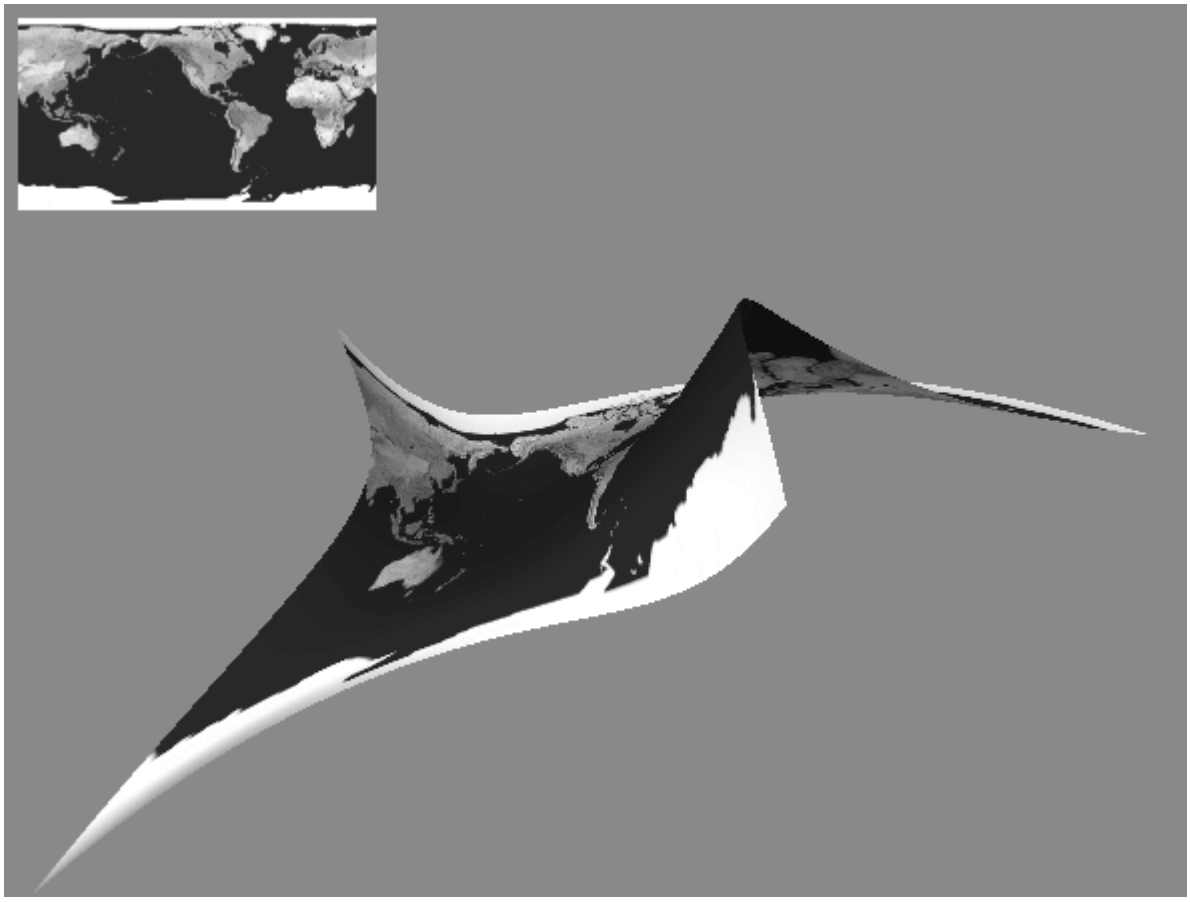
图 7-12 圆锥体的缺省纹理映射



SoNurbsSurface

当一个纹理使用缺省的纹理坐标被应用到 NURBS 曲面上时，纹理的边缘将被缩放，以匹配 NURBS 的片元（patch）。（见图 7-13）。一个曲面可以由多个片元组成，例如构成一个茶壶的曲面。如果 NURBS 曲面被剪裁了，那么纹理也同时被剪裁。

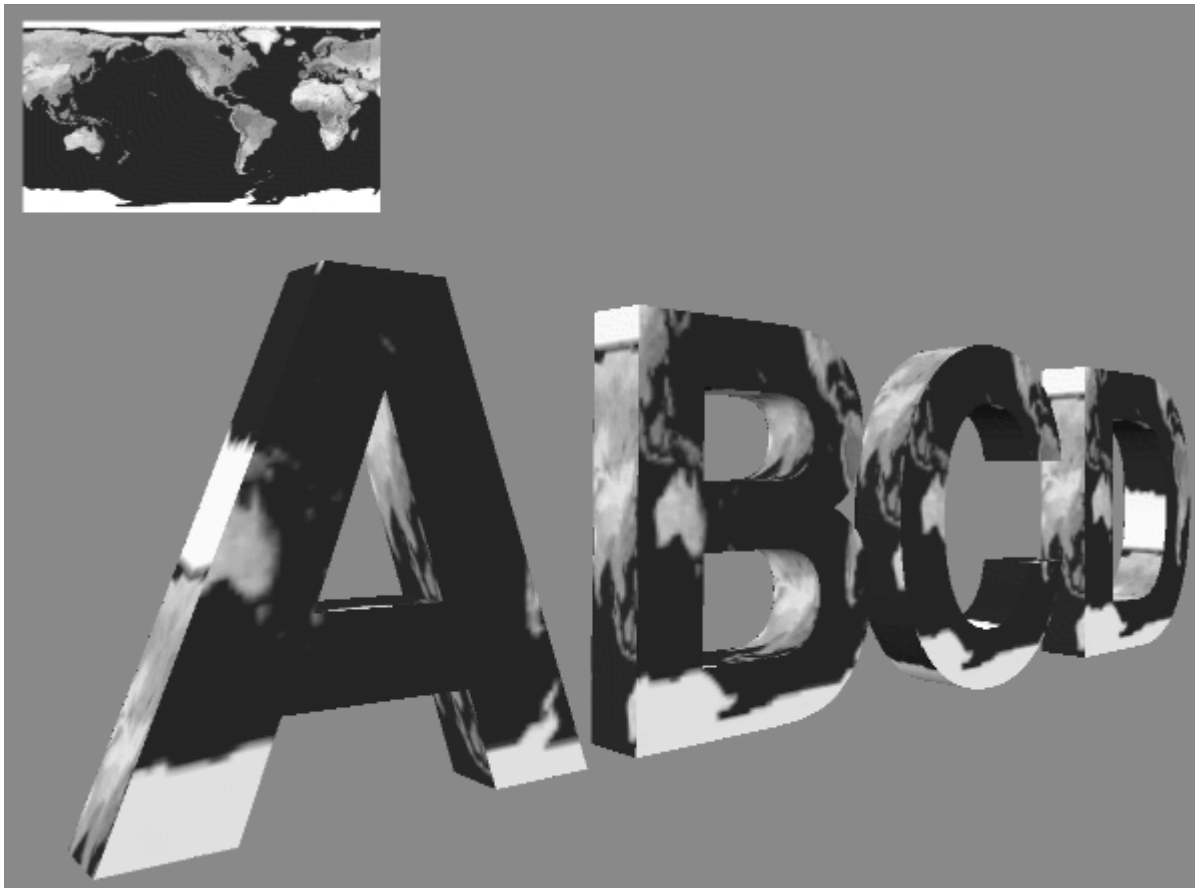
图 7-13 NURBS 曲面的缺省纹理映射



SoText3

当一个纹理使用缺省的纹理坐标被应用到**SoText3**的“前面”时，纹理坐标（0,0）将位于文本的原点。在纹理坐标s和t轴上，从 0.0 到 1.0 之间的距离等于字体的大小。当对文本的侧面也使用缺省纹理坐标时，以字符的背面作为纹理坐标（0，0）的开始，s轴纹理坐标沿着字符轮廓方向进行扩展，直到延伸到字符的前面。沿着轮廓方向字体大小的距离（*font-size* distance）等于纹理坐标 1.0 的距离。t轴坐标使用类似的方式，以顺时针方向包围字符的轮廓。（见图 7-14）

图 7−14 **SoText3 的缺省纹理映射**



显式指定纹理坐标（高级内容）

有时，我们希望显式地为物体上每个顶点指定纹理坐标。在这种情况下，我们可以先创建一个**SoTextureCoordinate2**节点，然后指定一个 2D纹理坐标数组，最后将数组中的数据应用到物体的每个顶点上。这样就达到了为物体指定纹理坐标的目的。

当使用这种技术时，我们必须首先给物体上的每个顶点都指定一个纹理坐标。纹理坐标是按照“成对”的方式给出的：先是 s 坐标，接着是 t 坐标。

例 7-2 将演示显式指定纹理坐标的效果。这个例子使用**SoTextureCoordinateBinding**节点来索引引用纹理坐标。

例 7-2 显式指定纹理坐标

```
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCoordinate3.h>
#include <Inventor/nodes/SoFaceSet.h>
#include <Inventor/nodes/SoNormal.h>
#include <Inventor/nodes/SoNormalBinding.h>
#include <Inventor/nodes/SoSeparator.h>
```

```
#include <Inventor/nodes/SoTexture2.h>
#include <Inventor/nodes/SoTextureCoordinate2.h>
#include <Inventor/nodes/SoTextureCoordinateBinding.h>

main(int , char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if(myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   // Choose a texture
   SoTexture2 *brick = new SoTexture2;
   root->addChild(brick);
   brick->filename.setValue("brick.1.rgb");

   // Define the square's spatial coordinates
   SoCoordinate3 *coord = new SoCoordinate3;
   root->addChild(coord);
   coord->point.set1Value(0, SbVec3f(-3, -3, 0));
   coord->point.set1Value(1, SbVec3f( 3, -3, 0));
   coord->point.set1Value(2, SbVec3f( 3,  3, 0));
   coord->point.set1Value(3, SbVec3f(-3,  3, 0));

   // Define the square's normal
   SoNormal *normal = new SoNormal;
   root->addChild(normal);
   normal->vector.set1Value(0, SbVec3f(0, 0, 1));

   // Define the square's texture coordinates
   SoTextureCoordinate2 *texCoord = new SoTextureCoordinate2;
   root->addChild(texCoord);
   texCoord->point.set1Value(0, SbVec2f(0, 0));
   texCoord->point.set1Value(1, SbVec2f(1, 0));
   texCoord->point.set1Value(2, SbVec2f(1, 1));
   texCoord->point.set1Value(3, SbVec2f(0, 1));

   // Define normal and texture coordinate bindings
   SoNormalBinding *nBind = new SoNormalBinding;
   SoTextureCoordinateBinding *tBind =
            new SoTextureCoordinateBinding;
   root->addChild(nBind);
```

```
    root->addChild(tBind);
    nBind->value.setValue(SoNormalBinding::OVERALL);
    tBind->value.setValue
             (SoTextureCoordinateBinding::PER_VERTEX);

    // Define a FaceSet
    SoFaceSet *myFaceSet = new SoFaceSet;
    root->addChild(myFaceSet);
    myFaceSet->numVertices.set1Value(0, 4);

    SoXtExaminerViewer *myViewer =
             new SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Texture Coordinates");
    myViewer->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

使用纹理坐标函数

将纹理坐标映射到物体上的第三种方法是使用纹理坐标函数（*texture-coordinate function*）。纹理坐标函数根据物体上每个顶点的位置为物体计算出纹理坐标。每种纹理坐标函数都使用不同的运算公式来计算纹理坐标，这些函数将在下面的章节中做详细的讨论。这

些函数允许我们以常规方式使用纹理映射，不需要我们显式地定义纹理坐标。纹理坐标函数将忽略掉**SoTextureCoordinate2**节点所定义的当前纹理坐标。

Inventor 包含有两种纹理坐标函数：

SoTextureCoordinatePlane	通过一个平面投影纹理贴图
SoTextureCoordinateEnvironment	指定物体表面看上去它们所处环境的反射图像（也称反射映射或环境映射）.

如果想让Inventor使用缺省的纹理坐标，可以使用**SoTextureCoordinateDefault**节点。(也就是说，使用**SoTextureCoordinateDefault**节点，可以“关闭”掉场景中前面定义的任何纹理坐标节点所产生的效果，并不一定非要使用separator节点才能达到隔离的效果)

SoTextureCoordinatePlane

SoTextureCoordinatePlane 可能是最常用的纹理坐标函数了，如彩图 15 所示，这个函数将纹理平面投影到物体表面上。我们可以定义 s 和 t 方向，这两个方向将用于为纹理定义一个平面。纹理坐标（s）是根据下面的公式定义产生的，其中 *coord* 是物体上的一个坐标值。

$$s = \left(\frac{1}{directionS}\right) * coord$$

SoTextureCoordinatePlane包含有下列域：

directionS (SoSFVec3f) s 坐标的投影方向（缺省值为 1.0,0.0,0.0）

directionT (SoSFVec3f) t 坐标的投影方向（缺省值为 0.0,1.0,0.0）

方向向量的长度等于纹理的重复间隔（repeat interval of the texture）（见例子 7-3）。

例 7-3 演示了**SoTextureCoordinatePlane**的用法。这个例子绘制了带有纹理的三个球体，每个球体上的纹理使用了不同的纹理重复周期，重复周期定义在**SoTextureCoordinatePlane**节点域中。

例 7-3 使用 **SoTextureCoordinatePlane**

```
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoTexture2.h>
#include <Inventor/nodes/SoTexture2Transform.h>
```

```
#include <Inventor/nodes/SoTextureCoordinatePlane.h>
#include <Inventor/nodes/SoTranslation.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>

main(int , char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if(myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   // Choose a texture.
   SoTexture2 *faceTexture = new SoTexture2;
   root->addChild(faceTexture);
   faceTexture->filename.setValue("sillyFace.rgb");
   // Make the diffuse color pure white
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(1,1,1);
   root->addChild(myMaterial);
```

```
// This texture2Transform centers the texture about (0,0,0)
SoTexture2Transform *myTexXf = new SoTexture2Transform;
myTexXf->translation.setValue(.5,.5);
root->addChild(myTexXf);

// Define a texture coordinate plane node.  This one will
// repeat with a frequency of two times per unit length.
// Add a sphere for it to affect.
SoTextureCoordinatePlane *texPlane1 = new
          SoTextureCoordinatePlane;
texPlane1->directionS.setValue(SbVec3f(2,0,0));
texPlane1->directionT.setValue(SbVec3f(0,2,0));
root->addChild(texPlane1);
root->addChild(new SoSphere);

// A translation node for spacing the three spheres.
SoTranslation *myTranslation = new SoTranslation;
myTranslation->translation.setValue(2.5,0,0);

// Create a second sphere with a repeat frequency of 1.
SoTextureCoordinatePlane *texPlane2 = new
         SoTextureCoordinatePlane;
texPlane2->directionS.setValue(SbVec3f(1,0,0));
texPlane2->directionT.setValue(SbVec3f(0,1,0));
root->addChild(myTranslation);
root->addChild(texPlane2);
root->addChild(new SoSphere);

// The third sphere has a repeat frequency of .5
SoTextureCoordinatePlane *texPlane3 = new
         SoTextureCoordinatePlane;
texPlane3->directionS.setValue(SbVec3f(.5,0,0));
texPlane3->directionT.setValue(SbVec3f(0,.5,0));
root->addChild(myTranslation);
root->addChild(texPlane3);
root->addChild(new SoSphere);

SoXtExaminerViewer *myViewer = new SoXtExaminerViewer(myWindow);
myViewer->setSceneGraph(root);
myViewer->setTitle("Texture Coordinate Plane");
myViewer->show();

SoXt::show(myWindow);
```

```
   SoXt::mainLoop();
}
```

SoTextureCoordinateEnvironment

SoTextureCoordinateEnvironment节点表示其后的物体应该反射它们周围的环境，就像表面锃亮的圣诞饰品可以反射其周围环境一样。为了产生最好的效果，所指定的纹理贴图应该是球面反射贴图。具体见*OpenGL Programming Guide*第 9 章中关于如何创建球面反射。

当使用**SoTextureCoordinateEnvironment**节点时，多边形的每个顶点都要进行一个计算，以此来决定从视点到顶点的向量所指向的地方那里应该被反射。这个反射点为多边形上的点定义了相应的纹理坐标。具体见彩图 17 和彩图 18。

因为环境映射是在 OpenGL 中实现的，所以环境映射只有在照相机没有和被反射的环境之间做相对运动时才是准确的。

第八章 曲线与曲面

本章目标

在阅读完本章节后，我们将可以做下列事情：

- 创建多种类型的曲线和曲面。
- NURBS 曲面的区域剪裁。
- 使用 NURBS 轮廓线为 3D 文本指定边缘斜面（beveled edges）。

（**高级内容**）

曲线与曲面提供了一种用于描述几何模型的数学方法，它代替了传统繁琐的制图术、金属样条、或泥塑模型等方法。设计师可以使用数学公式来表达像机翼、汽车外壳、机器工件上的曲面或者其它平滑的曲线和曲面。Inventor 使用一种特殊类型的参数多项式,称为 NURBS（非均匀有理 B 样条），来表达曲线和曲面。这将是本章所要讨论的最重要的内容。

综述

要想在 Inventor 程序中使用 NURBS 曲线和曲面，我们必须首先要对一些基本概念有所了解。本章将首先定义一些关键的概念，然后将讨论这些概念如何与 Inventor NURBS 类相对应。对于更多 NURBS 严格的数学描述，请看本章结尾的“**进一步阅读建议**”

NURBS 形体类

本节将描述下列类的用法：

SoNurbsCurve	描绘一条 NURBS 曲线（需要指定节点（knot）序列）
SoNurbsSurface	描绘一个 NURBS 曲面（需要指定节点（knot）序列）
SoNurbsProfile	使用 NURBS 曲线在 NURBS 曲面上剪裁一块区域
SoLinearProfile	使用连接线段在 NURBS 曲面上剪裁一块区域
SoProfileCoordinate2	为剪裁曲线定义 2D 坐标
SoProfileCoordinate3	为剪裁曲线定义有理 2D 坐标
SoCoordinate3	定义 NURBS 曲面或曲线的控制点
SoCoordinate4	定义 NURBS 曲面或曲线的有理控制点。

参数曲线（Parametric Curves）

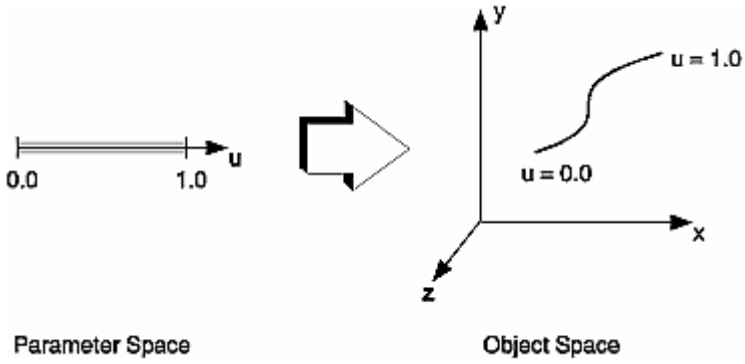
为了简单起见，本节将首先解释一个重要的 NURBS 概念，术语称“曲线（*curves*）”。曲线是位于在 3D 空间中的，就像一条螺旋线那样。一旦我们明白了如何定义 NURBS 曲线，那么定义 NURBS 曲面不过是定义曲线的简单扩展。

NURBS 曲线或曲面都是参数化的，也就是说，描述它们的方程是依赖于一些变量（或参数），这些变量（或参数）并不是几何模型的一部分。NURBS 曲线使用一个参数（术语称 *u*）来描述。下列的三个函数将这个单一参数映射到 *x-y-z* 空间中。

```
x = f(u)
y = g(u)
z = h(u)
```

利用参数空间内所有的 *u* 值，就可以求解方程，计算出位于物体空间内曲线上的对应点坐标（*x, y, z*）。图 8-1 给出了参数空间到物体空间的映射关系图。

图 8-1 映射参数曲线到到物体空间



关键概念

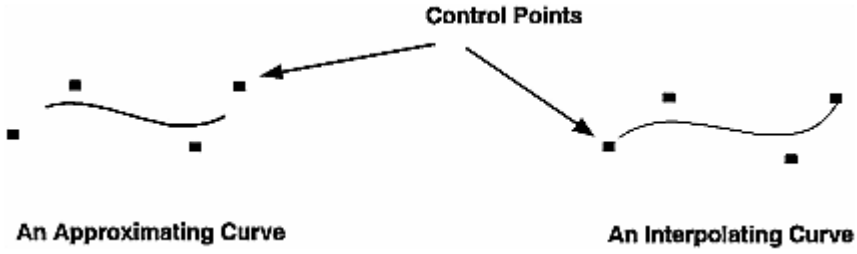
作为程序员，我们的任务只需要给定一些参数建立起参数方程即可（就是上节中讨论的 *f(), g(), h()*），不必自己去明确地定义参数方程。我们需要指定下面三个参数：

- 控制点 — 使用SoCoordinate3 或 SoCoordinate4 节点
- 节点（knots）序列 — 使用SoNurbsCurve 或 SoIndexedNurbsCurve节点
- 阶数（Order）— 由控制点个数和节点个数隐含定义。

本节将对这些参数做简要的描述，随同一起描述的是它们之间的关系，以及连续性的定义。更详细的讨论将在“基函数”小节中进行。

控制点是一些位于曲线物体空间内，可以影响曲线形状的点。曲线可以靠近控制点，如图 8-2 左边部分所示，也可以通过其中的一些控制点，如图 8-2 右边部分所示。控制点是一组数据点集，通过它们可以控制一条曲线的形状，或者通过一群网格点来描述一个例如像汽车车罩那样的曲面。在 Inventor 中，使用 **SoCoordinate3** 或 **SoCoordinate4** 节点指定控制点数据。

图 8-2 使用控制点来塑造曲线



节点（knots）序列是用来定义控制点将如何影响曲线形状。节点序列是一组简单的非减（nondecreasing）序列数。这些数决定了曲线是通过（和插值）控制点（插值曲线），还是逼近控制点（拟和曲线）。在 Inventor 中，使用 SoNurbsCurve 或 SoNurbsSurface（或 SoIndexedNurbsCurve, SoIndexedNurbsSurface）节点指定节点序列。

曲线的阶数（Order）决定了参数方程的形式。阶数等于参数方程中变量的最大次数加 1。例如，三次曲线的参数方程（次数=3，阶数=4）有下列形式：

```
x(u) = Axu3 + Bxu2 + Cxu + Dx
y(u) = Ayu3 + Byu2 + Cyu + Dy
z(u) = Azu3 + Bzu2 + Czu + Dz
```

类似地，二次曲线的参数方程（次数=2，阶数=3）有下列形式：

```
x(u) = Axu2 + Bxu + Cx
y(u) = Ayu2 + Byu + Cy
z(u) = Azu2 + Bzu + Cz
```

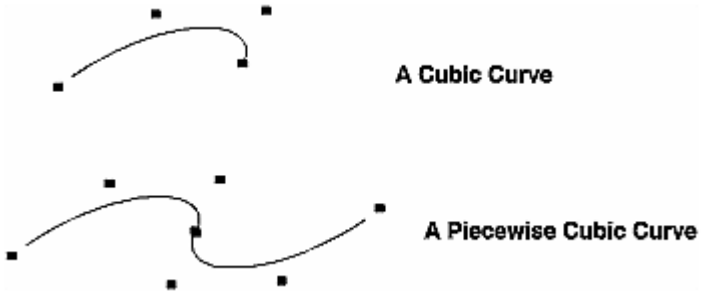
我们也可以将阶数看作是参数方程中系数的个数。至于曲线的阶数是如何影响到曲线的平滑度的，请看“曲线的连续度”章节。

在 Inventor 中，曲线的阶数不需要显式指定。因为阶数 ＝ 节点个数 － 控制点个数。

控制点和阶数

一条曲线的阶数决定了定义此曲线的控制点的最少个数。我们至少要有“阶数”个控制点才能定义一条曲线。（例如，定义一条 4 阶曲线，我们至少要有 4 个控制点才能定义出此曲线）。如果要定义一条带有超过阶数个控制点的曲线，我们可以通过将多条曲线段连接成一条分段曲线（*piecewise curve*）的方式来达到我们的目的（To make curves with more than order control points, you can join two or more curve segments into a *piecewise curve*）。（见图 8-3）

图 8-3 分段三次曲线



曲线的阶数还会影响着当控制点移动时曲线的形状。在 Inventor 中，一条 NURBS 曲线最高的阶数为 8。阶数过高会将扰动带到曲线中，当控制点移动的时候，曲线会发生不可预测的行为。三次曲线（阶数为 4）是最常用的曲线。对于大多数的几何建模软件来说，三次曲线即提供了足够的控制能力，同时还不会带有高阶曲线的那些缺点。

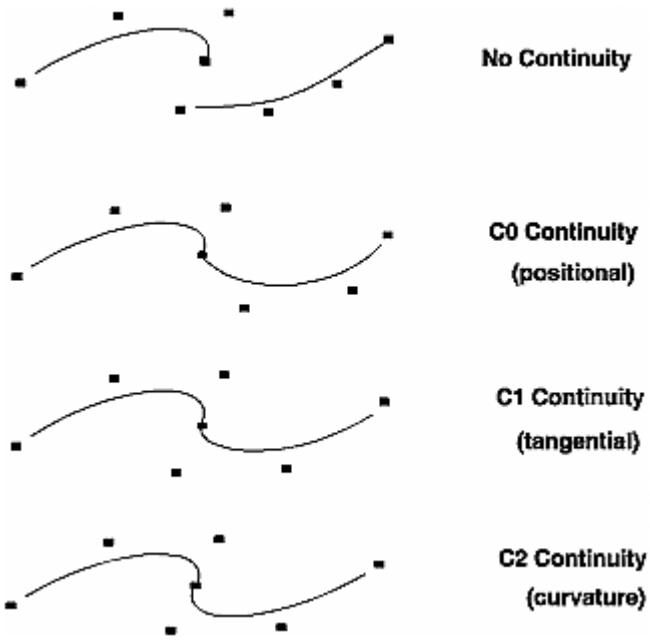
曲线的连续性

在分段曲线内，两条小曲线段相连接的地方称作断点（*breakpoint*）。在断点处的曲线连续性描述的是这些曲线段是如何在断点处连接的。图 8-4 给出了可能的四种连续性的示意图：

No continuity	两条曲线段不连接
C^0 *continuity*	两条曲线段端点相连接（即曲线只有位置连续）。连接点可能是一个尖点
C^1 *continuity*	两条曲线段在断点处有相同的切线（切线是指曲线在断点处斜率）。曲线连接平滑。C^1曲线同时满足位置连续。
C^2 *continuity*	两条曲线段在断点处有相同的曲率（曲率是指切线的变化率）。曲率连续也同时意味着切线和位置连续

曲线的阶数决定着曲线的最大可能的连续度。因此，如果希望曲线有更好的连续性，我们就必须要让曲线具有更高的阶数。最大连续度 ＝ 阶数 － 2。对于三次曲线来说，最大连续度为C^2(曲率连续)

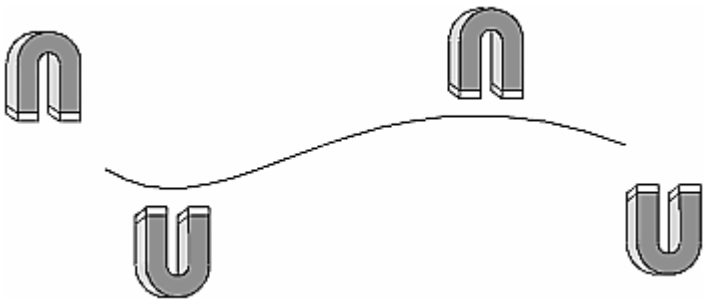
图 8-4 曲线的连续性



基函数

每个控制点都像一块吸引着曲线的磁铁（见图 8-5）。这些磁铁的吸力和影响的范围是通过一个特定的基函数来定义的。对于 BURBS 来说，这个函数就叫做 *B-spline* 基函数。（在“**进一步阅读建议**”中，提及了更多有关 *B-spline* 基函数的由来）。

图 8-5 控制点影响曲线形状

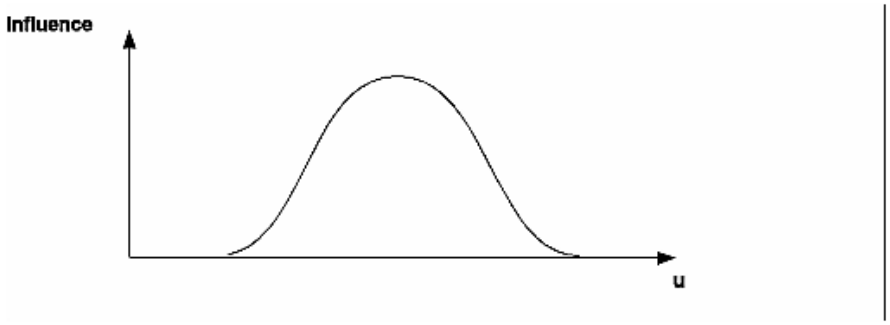


B-spline 基函数（图 8-6）是在参数（*u*）空间中描述曲线。对于每个 *u* 值：

每个控制点对曲线形状的贡献度 ＝ 控制点的位置 × 控制点的基函数

曲线的最终形状等于所有控制点对曲线形状的贡献值的总和。注意，通常一个控制点（相当于一个磁铁）可以影响到整个曲线，尽管离控制点越远的地方，控制点的影响力会变的越小。控制点的影响范围是由节点序列决定的。

图 8-6. B-Spline 基函数

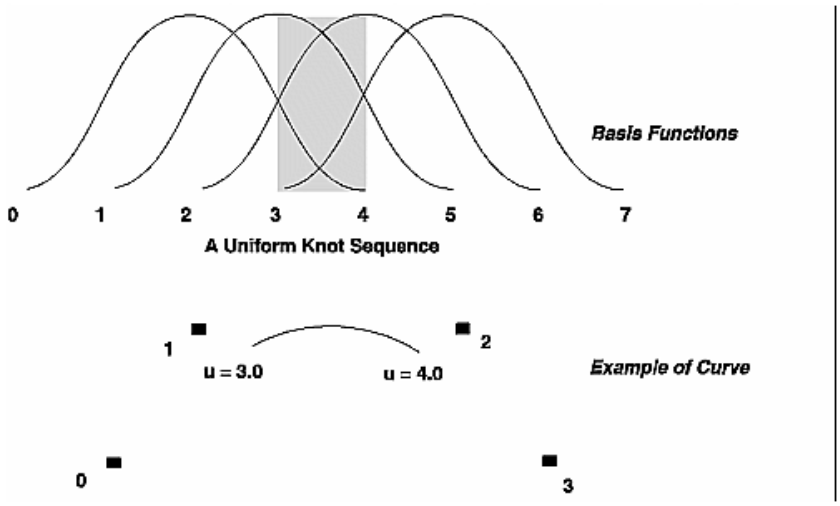


节点序列

基函数在参数空间内的分配是受节点序列（也称节点向量，或简称节点）所控制的。节点序列是一组非减（nondecreasing）序列数。每个节点都定义了基函数的起点和终点（Each knot defines the beginning and end of a basis function）。节点序列中的节点的个数必须是一个确切的值，即节点个数 ＝ 阶数 ＋ 控制点个数。曲线只定义在阶数个基函数相交迭的节点区间内。（见图 8-7 中，**曲线是四阶的，只有节点 3，4 之间是四个基函数的相交迭处，在其它节点区间内都没有四个基函数相交。所以曲线只定义在节点 3，4 之间。译者注**）。如果节点值都是唯一的（即没有重复的数值）并且节点值均匀分布（**即所有前后两个节点值的差都相等。译者注**），那么称这样的曲线为均匀 B 样条曲线（*uniform B-spline*）（见图 8-7）

在图 8-7 中，展示了一个均匀的节点序列。定义了四个控制点（在物体空间）。图中上面部分绘制了作用在每个控制点上的四个基函数。这些基函数在 u=3.0 到 u=4.0 之间相交迭。图中使用阴影区域表示。这幅图还说明了 NURBS 另外一个重要的关系：在曲线所定义的任意点处，所有基函数的相加之和等于 1（This figure also illustrates another important NURBS relationship: at any point where the curve is defined, the sum of all basis functions is equal to 1）。

图 8-7 均匀节点序列



重复节点（Knot Multiplicity）

节点值定义了分段的个数。一个基函数总是要跨越“阶数”个段。例如在图 8-7 中，以 0 为起点，4 为终点的基函数跨越了 4 个段。（节点 0 到节点 1、节点 1 到节点 2、节点 2 到节点 3、节点 3 到节点 4）。

在节点序列中重复的数值将增加节点的重复数（Duplicating values in the knot sequence increases that value's *multiplicity*），这将导致有多个基函数起始于那个重复的节点。同时也会降低曲线的连续度。在图 8-8 中，使用了两组相同的控制点，但不同的节点序列，分别绘制了上下两条曲线。注意：下面的曲线只有C^0连续度，上面的则有C^2连续度。重复节点个数与曲线连续度之间的关系可以使用下面的数学公式来表示：

$$C^{ORD-(M+1)}$$

这里，ORD 等于曲线的阶数，M 等于重复节点个数。

最大的重复数（可以重复一个节点的最大次数）等于阶数。表 8-1 列出了重复节点和对应的连续度数值。

图 8-8 重复节点

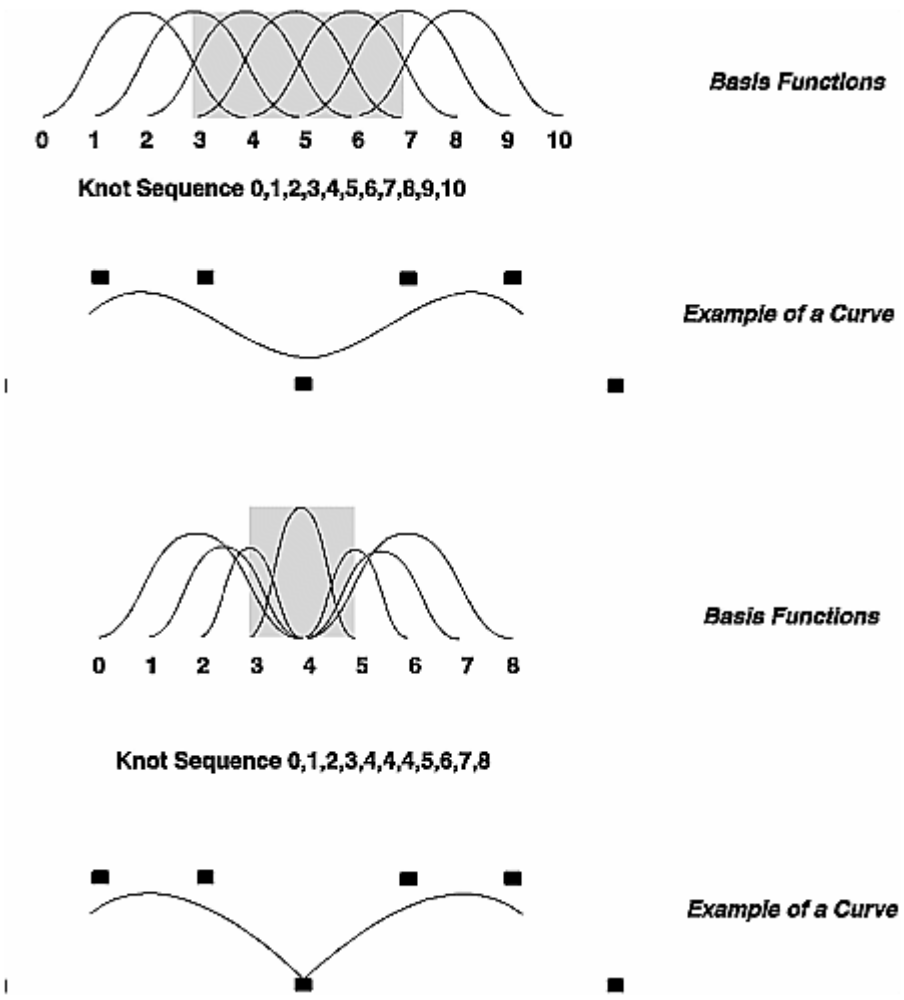


表 8-1 三次曲线的连续度与重复节点

重复节点数	连续状况	连续度
1	位置连续 切线连续 曲率连续	C^2
2	位置连续 切线连续	C^1
3	位置连续	C^0
4	不连续	不连续

常用的节点序列

下面的几种常用的节点序列对绝大多数应用程序来说是非常有用的。

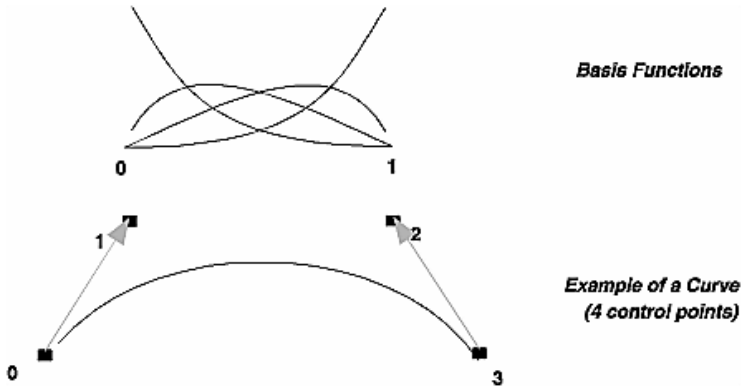
三次均匀 B 样条 *Uniform cubic B-spline*　　节点均匀分布，没有重复节点（例如 0, 1, 2, 3, 4, 5, 6, 7）

三次贝赛尔曲线 *Cubic Bezier curve*　　在起始和结尾处的重复数 = 4（例如 0, 0, 0, 0, 1, 1, 1, 1）

通过端点的三次均匀 B 样条 *Uniform cubic B-spline that passes through endpoints*　　在起始和结尾处的重复数 = 4；中间均匀分布（例如 0, 0, 0, 0, 1, 2, 3, 4, 5, 5, 5, 5）

贝赛尔曲线和三次均匀 B 样条的行为非常适合开发几何建模和 CAD 方面的应用程序。曲线经过了起始和终止控制点（见图 8-9）。第一个控制点与第二个控制点之间的连线就是第一个端点（endpoint）的切线。最后两个控制点之间的连线就是第二个端点的切线。

图 8-9 三次贝赛尔曲线



NURBS 参数间关系的总结

前面的章节中已经概要讨论了 NURBS 参数间的重要关系。下面总结这些参数关系：

- 阶数 ＝ 次数 ＋ 1；次数是指参数方程中变量的最大指数
- 必须至少有“阶数”个控制点才能定义一条曲线。
- 最大连续度 ＝ 阶数 － 2

- 节点个数 ＝ 阶数 ＋ 控制点个数
- 节点值必须是非减的。
- 最大的节点重复数 ＝ 阶数
- 连续度 ＝ 阶数－（节点重复数 ＋ 1）

例如，对于三次曲线来说，阶数等于 4，所以至少要有 4 个控制点才能定义三次曲线。最大连续度是C^2连续。在节点序列中最少需要有 8 个节点。节点允许重复的最大次数是 4。

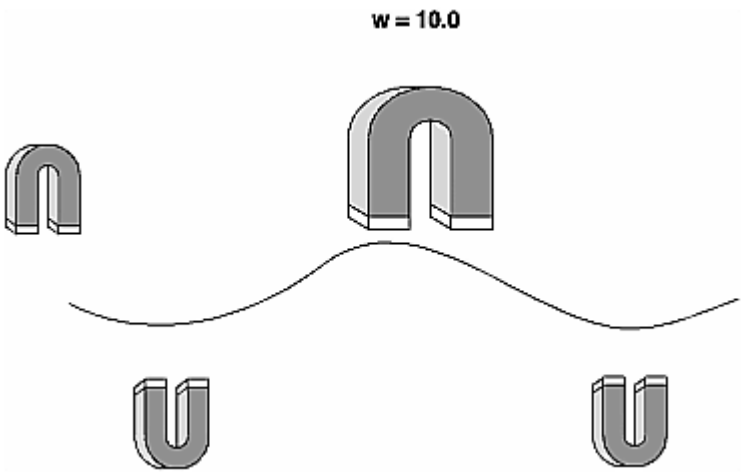
有理曲线

每个控制点都可以关联着一个权值，这个权值会对控制点的基函数的形状产生影响。如图 8-10 所示，这类似于使用不同大小的磁铁来吸引曲线那样。对于无理曲线来说，所有的控制点都带有一个相同的权值 1.0。而有理曲线却是每个控制点可以有不同的权值。如果控制点带有一个大于 1.0 的权值，那么这个控制点对曲线的影响将肯定大于带有 1.0 权值的控制点所产生的影响。

有理曲线的参数方程中带有分子和分母，称为比率。（分子是原来的参数方程。分母则是另外一个考虑了权值的参数方程） 我们建议权值一般要大于 0。可以使用**SoCoordinate4** 节点来指定*x, y, z,* and *w* (权值)值

有理曲线和曲面可以精确地表示圆锥截面、球体、和圆柱等几何体。更多内容见本章结尾处“**进一步阅读建议**”。

图 8-10 有理曲线



N-U-R-B-S 拼成 NURBS

如果我们已经深入地讨论了 NURBS，那么就应该能理解组成 NURBS 缩写词的这些字母的具体含义：

Non-Uniform 节点分布不需要均匀

Rational　　描述曲线的参数方程带有一个分母（就是说，它们可以是比率的）

B-Spline　　控制点影响的是 *B-spline* 基函数

NURBS 曲线的例子

本节将演示两条 NURBS 曲线的例子：一条是 B 样条曲线，另一条是通过控制端点的均匀 B 样条曲线。

B 样条曲线

例 8-1 产生并显示了一条B样条曲线。定义了 7 个控制点，节点向量中包含有 10 个节点。因为节点个数 ＝ 阶数 ＋ 控制点个数，所以这条曲线的阶数是 3。节点重复数是 2（一个节点被用了两次）。这条曲线具有C^0连续度。

图 8-11 给出了这个例子的场景图，图 8-22 显示了例子的结果。

图 8-11 B 样条曲线例子的场景图

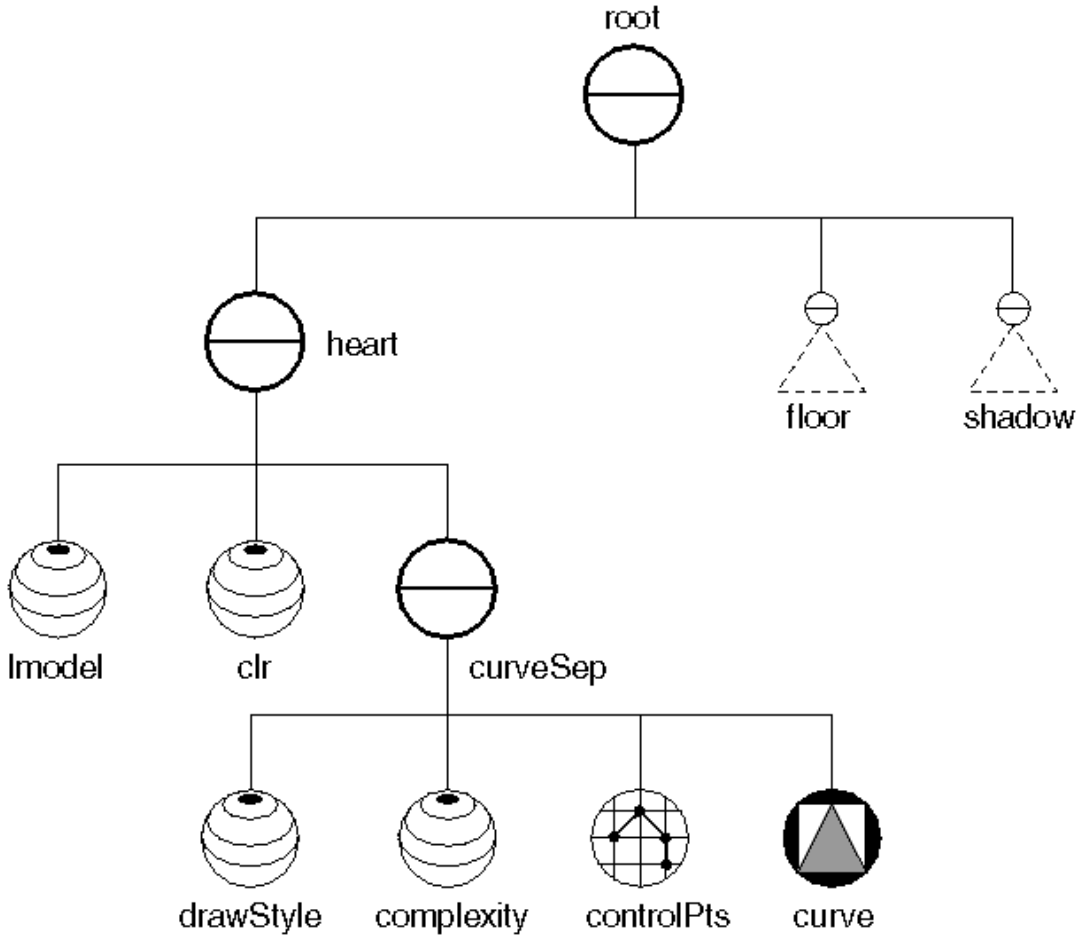
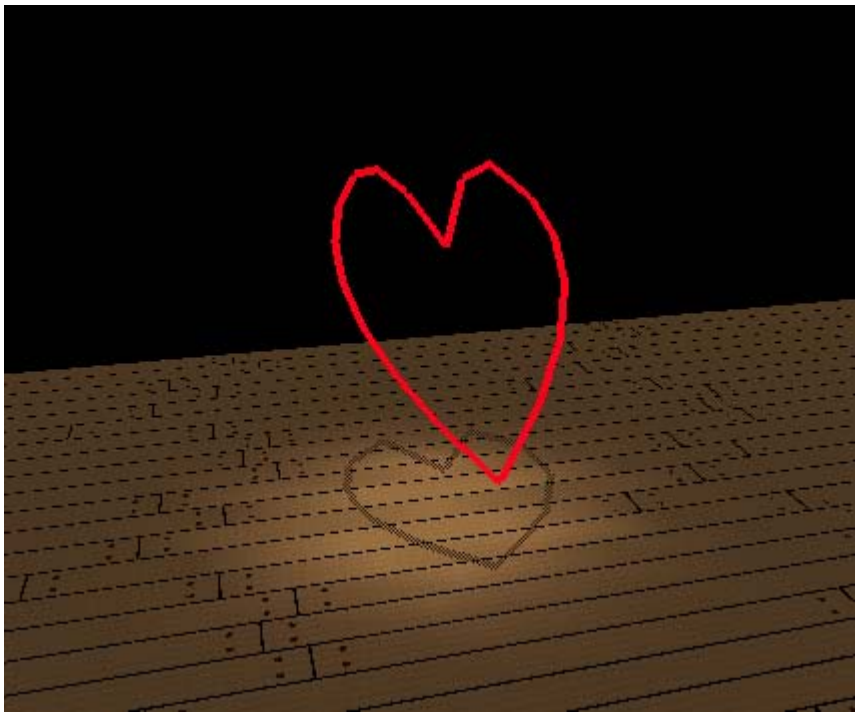


图 8-12 B 样条曲线



例 8-1 创建一条 B 样条曲线

```
// The control points for this curve
float pts[7][3] = {
   { 4.0, -6.0,  6.0},
   {-4.0,  1.0,  0.0},
   {-1.5,  5.0, -6.0},
   { 0.0,  2.0, -2.0},
   { 1.5,  5.0, -6.0},
   { 4.0,  1.0,  0.0},
   {-4.0, -6.0,  6.0}};

// The knot vector
float knots[10] = {1, 2, 3, 4, 5, 5, 6, 7, 8, 9};

// Create the nodes needed for the B-Spline curve.
SoSeparator *
makeCurve()
{
   SoSeparator *curveSep = new SoSeparator();
   curveSep->ref();
```

```
   // Set the draw style of the curve.
   SoDrawStyle *drawStyle  = new SoDrawStyle;
   drawStyle->lineWidth = 4;
   curveSep->addChild(drawStyle);

   // Define the NURBS curve including the control points
   // and a complexity.
   SoComplexity  *complexity = new SoComplexity;
   SoCoordinate3 *controlPts = new SoCoordinate3;
   SoNurbsCurve  *curve      = new SoNurbsCurve;
   complexity->value = 0.8;
   controlPts->point.setValues(0, 7, pts);
   curve->numControlPoints = 7;
   curve->knotVector.setValues(0, 10, knots);
   curveSep->addChild(complexity);
   curveSep->addChild(controlPts);
   curveSep->addChild(curve);

   curveSep->unrefNoDelete();
   return curveSep;
}
```

通过控制端点的均匀 B 样条曲线

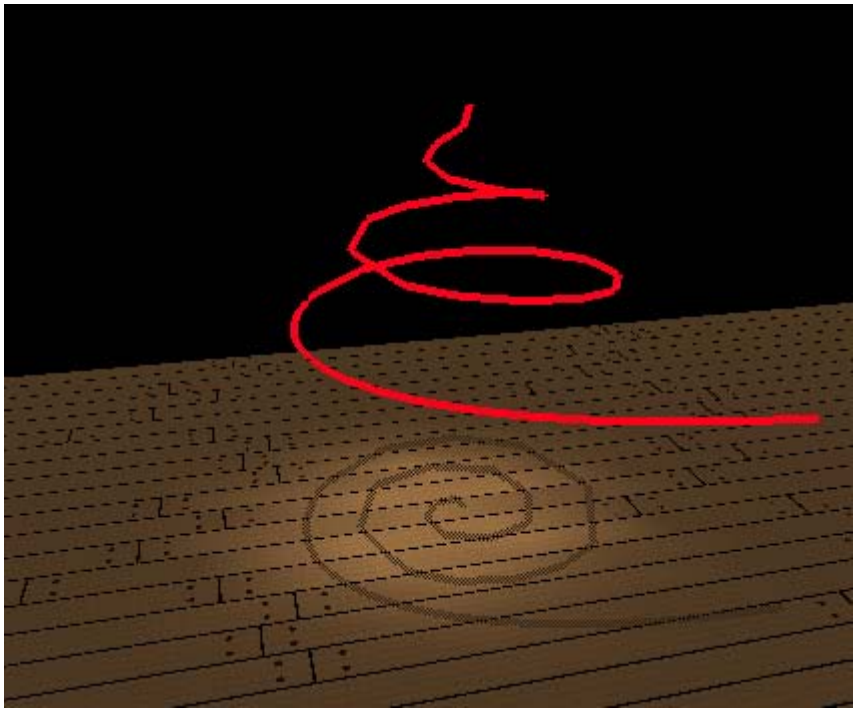
例 8-2 创建一个通过控制端点的均匀 B 样条曲线。节点序列在开始和结束处的重复数是 4，这样曲线将通过第一个和最后一个控制点。在中间，曲线这是均匀的。

这个例子的场景图和图 8-11 中显示的场景一样。

例 8-2 创建一条均匀 B 样条曲线

```
// The control points for this surface
float pts[16][3] = {
   {-4.5, -2.0,  8.0},
   {-2.0,  1.0,  8.0},
   { 2.0, -3.0,  6.0},
   { 5.0, -1.0,  8.0},
   {-3.0,  3.0,  4.0},
   { 0.0, -1.0,  4.0},
   { 1.0, -1.0,  4.0},
   { 3.0,  2.0,  4.0},
   {-5.0, -2.0, -2.0},
```

```
   {-2.0, -4.0, -2.0},
   { 2.0, -1.0, -2.0},
   { 5.0,  0.0, -2.0},
   {-4.5,  2.0, -6.0},
   {-2.0, -4.0, -5.0},
   { 2.0,  3.0, -5.0},
   { 4.5, -2.0, -6.0}};

// The knot vector
float knots[8] = {
   0, 0, 0, 0, 1, 1, 1, 1};
```



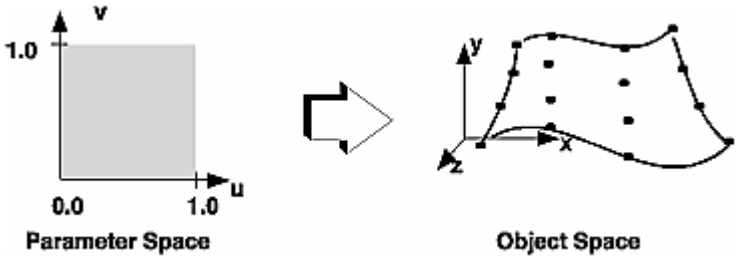
```
 // Create the nodes needed for the B-Spline curve.
SoSeparator *
makeCurve()
{
   SoSeparator *curveSep = new SoSeparator();
   curveSep->ref();

   // Set the draw style of the curve.
   SoDrawStyle *drawStyle  = new SoDrawStyle;
   drawStyle->lineWidth = 4;
```

```
   curveSep->addChild(drawStyle);

   // Define the NURBS curve including the control points
   // and a complexity.
   SoComplexity  *complexity = new SoComplexity;
   SoCoordinate3 *controlPts = new SoCoordinate3;
   SoNurbsCurve  *curve      = new SoNurbsCurve;
   complexity->value = 0.8;
   controlPts->point.setValues(0, 13, pts);
   curve->numControlPoints = 13;
   curve->knotVector.setValues(0, 17, knots);
   curveSep->addChild(complexity);
   curveSep->addChild(controlPts);
   curveSep->addChild(curve);

   curveSep->unrefNoDelete();
   return curveSep;
}
```

NURBS 曲面

曲面与曲线唯一的不同就是曲面有两个参数方向（*u* 和 *v*），而曲线只有一个（见图 8-12），而且曲面要同时对这两个参数指定阶数和节点向量。

图 8-12 曲面



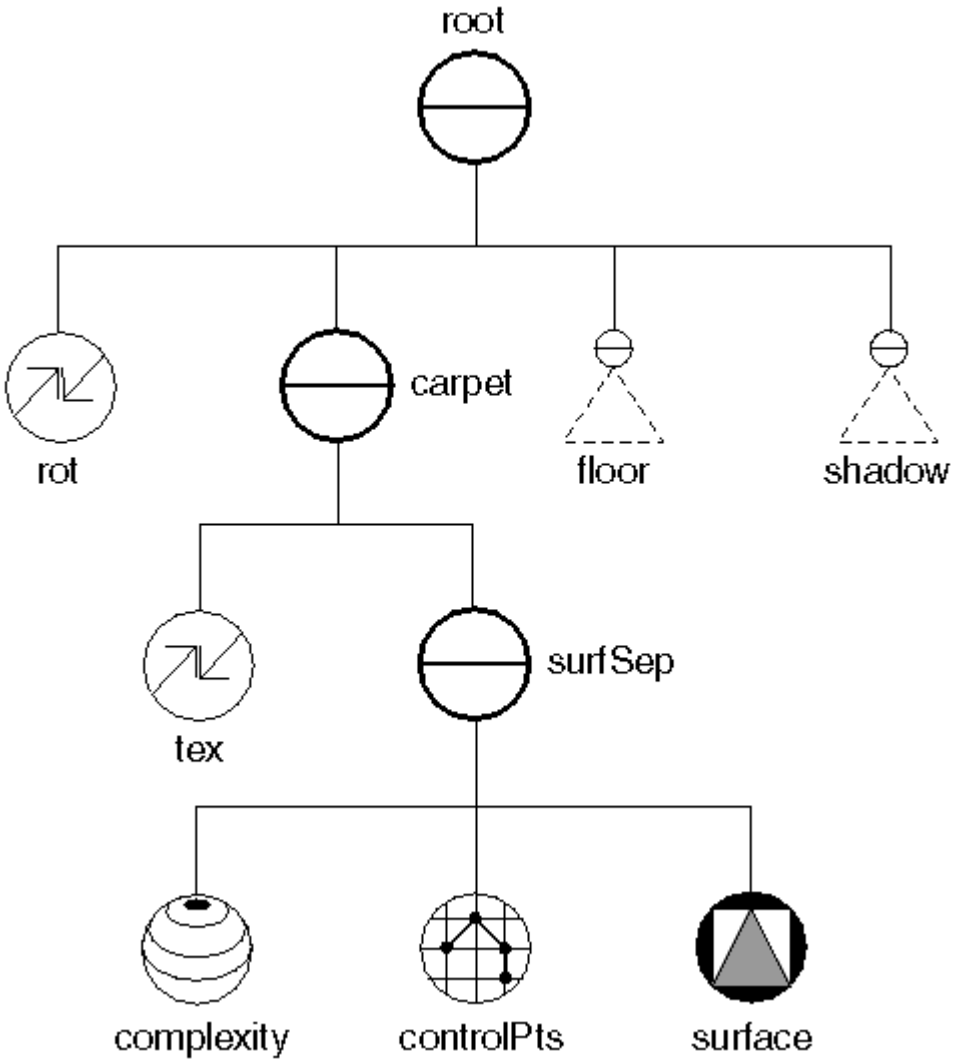
u 和 *v* 这两个参数维（parametric dimensions）将被映射到 3D 物体空间中。和曲线一样，控制点是定义在物体空间内。尽管 *u* 和 *v* 两个参数可以有不同的阶数和节点序列，但通常它们都是相同的。每个参数的阶数等于：阶数 ＝ 节点个数 － 控制点个数。

提示：将 NURBS 形体放在它们自己的 separator 节点内，这样做可以更容易缓存 NURBS 数据。

贝赛尔曲面

例 8-3 创建了一个平面贝赛尔曲面。节点向量定义了一个三次贝赛尔曲面（在开始和结尾处有 4 个重复数）。曲面的阶数是 4，有 16 个控制点，这些控制点分布在一个 4×4 的网格内。*u* 和 *v* 的节点向量长度都是 8。图 8-13 给出了这个例子的场景图。注意：在场景中，控制点（*controlPts*）必须放在 NURBS 节点（*surface*）的前面。

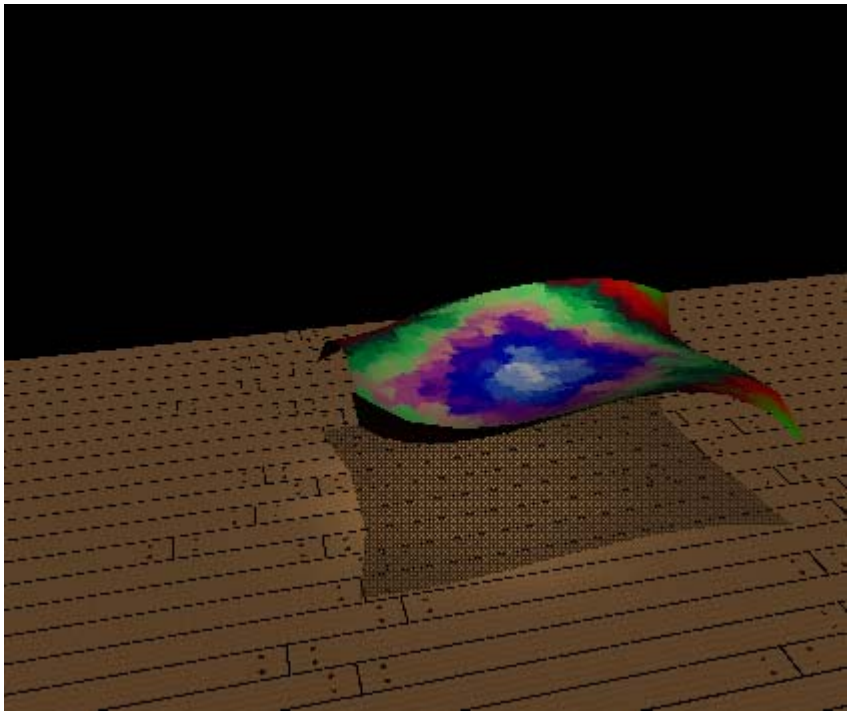
图 8-13 贝赛尔曲面场景图



例 8-3 贝赛尔曲面

```
// The control points for this surface
float pts[16][3] = {
   {-4.5, -2.0,  8.0},
   {-2.0,  1.0,  8.0},
   { 2.0, -3.0,  6.0},
   { 5.0, -1.0,  8.0},
```

```
   {-3.0,  3.0,  4.0},
   { 0.0, -1.0,  4.0},
   { 1.0, -1.0,  4.0},
   { 3.0,  2.0,  4.0},
   {-5.0, -2.0, -2.0},
   {-2.0, -4.0, -2.0},
   { 2.0, -1.0, -2.0},
   { 5.0,  0.0, -2.0},
   {-4.5,  2.0, -6.0},
   {-2.0, -4.0, -5.0},
   { 2.0,  3.0, -5.0},
   { 4.5, -2.0, -6.0}};

// The knot vector
float knots[8] = {
   0, 0, 0, 0, 1, 1, 1,
```



```
// Create the nodes needed for the Bezier surface.
SoSeparator *
makeSurface()
{
   SoSeparator *surfSep = new SoSeparator();
   surfSep->ref();
```

```
   // Define the Bezier surface including the control
   // points and a complexity.
   SoComplexity  *complexity = new SoComplexity;
   SoCoordinate3 *controlPts = new SoCoordinate3;
   SoNurbsSurface  *surface  = new SoNurbsSurface;
   complexity->value = 0.7;
   controlPts->point.setValues(0, 16, pts);
   surface->numUControlPoints = 4;
   surface->numVControlPoints = 4;
   surface->uKnotVector.setValues(0, 8, knots);
   surface->vKnotVector.setValues(0, 8, knots);
   surfSep->addChild(complexity);
   surfSep->addChild(controlPts);
   surfSep->addChild(surface);

   surfSep->unrefNoDelete();
   return surfSep;
}
```

提示：如果一个NURBS曲面正在变化中，在曲面节点的前面插入一个SoComplexity节点，并且将SoComplexity节点的type域设置为SCREEN_SPACE，这样做可以提高Inventor的性能。尤其是在NURBS曲面与我们的距离很远的情况下（If a NURBS surface is changing, inserting an **SoComplexity** node with SCREEN_SPACE specified as the **type** may improve performance, especially if the NURBS surfaces are far away）。

裁剪 NURBS 曲面

轮廓曲线（*Profile curves*）通常用来剪裁（切掉一块区域）NURBS 曲面。Inventor 不会渲染这些轮廓曲线。轮廓曲线只是简单地用于剪裁在场景中位于其后的 NURBS 曲面。就像几何变换节点那样，轮廓曲线也会被隔离节点入栈和出栈，并且作用效果也是累积的（Like transformations, profile curves are pushed and popped by separator groups,yet they accumulate with each other）。

轮廓曲线通常用于执行一个模版操作，就像使用剪刀从布料中剪掉一块小布料那样。轮廓曲线还可用于去掉 NURBS 曲面的某些尖角（sharp corners）。具体见例 6-3 3D 文本使用轮廓曲线。

裁剪 NURBS 曲面通常是比较高级的话题。如果是第一次接触 NURBS 的话，最好是先体验学习一下 NURBS 曲线和曲面，然后在学习裁剪 NURBS 曲面。

可以由线性轮廓曲线（**SoLinearProfile**）、NURBS曲线（**SoNurbsProfileCurve**）或二者组合来构造轮廓曲线。轮廓曲线使用**SoProfileCoordinate2**（无理轮廓曲线）或**SoProfileCoordinate3**（有理轮廓曲线）来保存坐标数据。最主要的要求是组合的轮廓曲线必须是封闭的，即起点和终点必须相同。另外，在轮廓曲线中不能出现曲线相交的情况。

提示：如果我们希望轮廓曲线是一条沿着曲面分布的直线，我们可以使用**SoNurbsProfileCurve**节点产生一条 2 阶的曲线（见例 8-4）。线性轮廓曲线也可以在物体空间内产生一条直线剪裁边，但它并不沿着曲面分布。很少使用**SoLinearProfile**来剪裁NURBS曲面。

轮廓曲线走向的定义是非常重要的。如果轮廓曲线按照顺时针方向定义，那么曲线内的区域是要被剪裁的区域，而曲线外的区域要保留。如果轮廓曲线按照逆时针方向定义，那么曲线内的区域要保留，而曲线外的区域则是要被剪裁的区域。轮廓曲线可以互相嵌套包围，但彼此之间不能相交。最外边的轮廓曲线必须按照逆时针方向定义（见例 8-4）。

轮廓曲线定义在参数空间内，但最终它将被映射到物体空间内。

例 8-4 对例 8-3 中创建的曲面增加上一条轮廓曲线。图 8-14 给出了这个例子的场景图。注意：在场景中，控制点（*controlPts*）必须放在 NURBS 节点（*surface*）的前面。类似的，定义轮廓曲线的点数据（*trimPts*）必须放在轮廓曲线节点（*nTrim1, nTrim2,* 和 *nTrim3*）之前。很显然，轮廓曲线节点必须放在要剪裁的 NURBS 曲面之前。

8-14 剪裁贝赛尔曲面的场景图

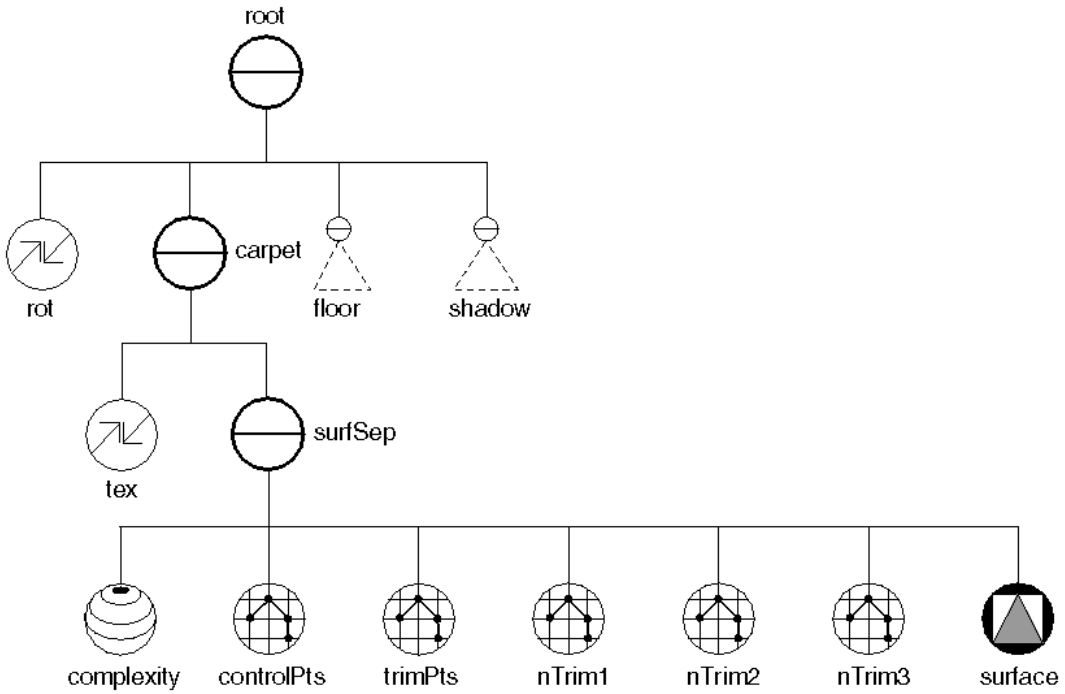
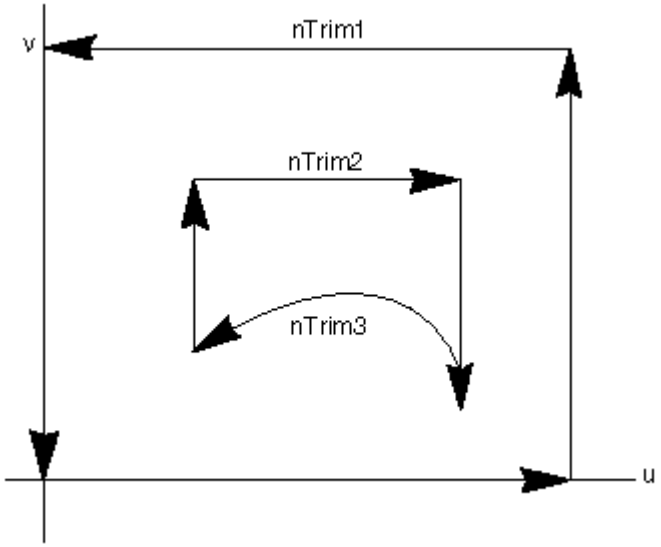


图 8-15 画出了例 8-4 所使用到的剪裁曲线，这条剪裁曲线是定义在参数空间内。例 8-4 使用了 3 条 NURBS 轮廓曲线。每条曲线都有自己的节点向量。第一条曲线，*nTrim1*，有 4 个段，5 个控制点（起点和终点相同）。这是一条经过端点且阶数为 2 的曲线。第二个轮廓曲线，*nTrim2*，也是一条线性曲线。这条曲线有 3 个段，也经过端点。第三条轮廓曲线，*nTrim3*，是一条三次曲线（阶数为 4）。在起点和终点处的重复数为 4，这样这条曲线就是一个经过端点的贝赛尔曲线了。

注意，这些剪裁曲线是相会嵌套包围的，并且最外边的曲线是按逆时针方向定义的。它们之间没有相交的地方。

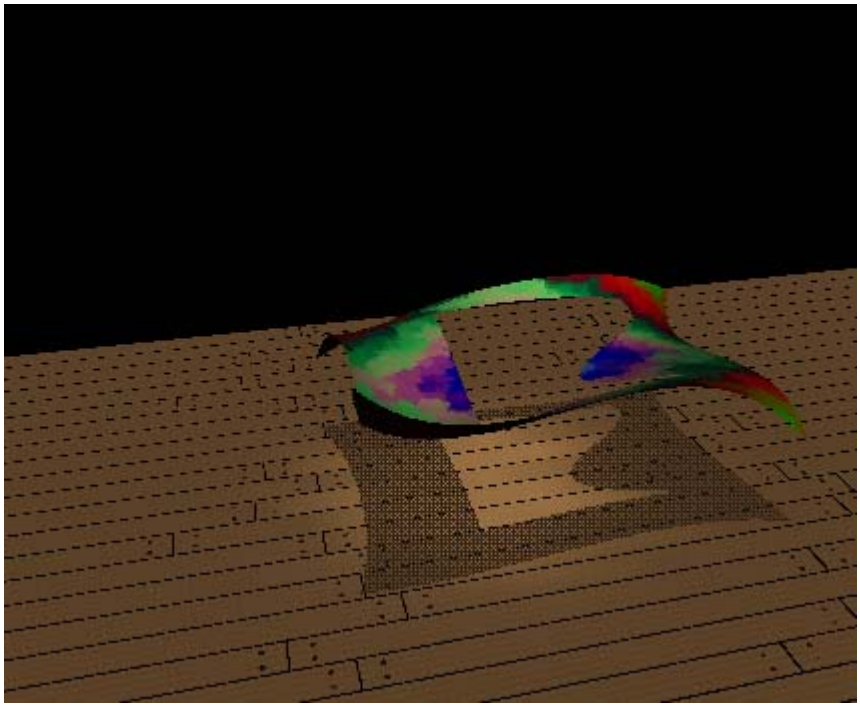
8-15 例 8-4 所使用的剪裁曲线



例 8-4 剪裁贝赛尔曲面

```
// The array of trim coordinates
float tpts[12][2] = {
   {0.0, 0.0},
   {1.0, 0.0},
   {1.0, 1.0},
   {0.0, 1.0},
   {0.2, 0.2},
   {0.2, 0.7},
   {0.9, 0.7},
   {0.9, 0.2},
   {0.7, 0.0},
   {0.4, 0.8}};

// The 16 coordinates defining the Bezier surface.
float pts[16][3] = {
```

```
    {-4.5,  -2.0,   8.0},
    {-2.0,   1.0,   8.0},
    { 2.0,  -3.0,   6.0},
```



```
    { 5.0,  -1.0,   8.0},
    {-3.0,   3.0,   4.0},
    { 0.0,  -1.0,   4.0},
    { 1.0,  -1.0,   4.0},
    { 3.0,   2.0,   4.0},
    {-5.0,  -2.0,  -2.0},
    {-2.0,  -4.0,  -2.0},
    { 2.0,  -1.0,  -2.0},
    { 5.0,   0.0,  -2.0},
    {-4.5,   2.0,  -6.0},
    {-2.0,  -4.0,  -5.0},
    { 2.0,   3.0,  -5.0},
    { 4.5,  -2.0,  -6.0}};

// The 3 knot vectors for the 3 trim curves.
float tknots1[7] = {0, 0, 1, 2, 3, 4, 4};
float tknots2[6] = {0, 0, 1, 2, 3, 3};
float tknots3[8] = {0, 0, 0, 0, 1, 1, 1, 1};
```

```
// The Bezier knot vector for the surface.
// This knot vector is used in both the U and
// V directions.
float knots[8] = {0, 0, 0, 0, 1, 1, 1, 1};

// Create the nodes needed for the Bezier patch
// and its trim curves.
SoSeparator *
makeSurface()
{
   SoSeparator *surfSep = new SoSeparator();
   surfSep->ref();

   // Define the Bezier surface including the control
   // points, trim curve, and a complexity.
   SoComplexity  *complexity     = new SoComplexity;
   SoCoordinate3 *controlPts     = new SoCoordinate3;
   SoNurbsSurface *surface       = new SoNurbsSurface;
   complexity->value = 0.7;
   controlPts->point.setValues(0, 16, pts);
   surface->numUControlPoints.setValue(4);
   surface->numVControlPoints.setValue(4);
   surface->uKnotVector.setValues(0, 8, knots);
   surface->vKnotVector.setValues(0, 8, knots);
   surfSep->addChild(complexity);
   surfSep->addChild(controlPts);

   SoProfileCoordinate2 *trimPts = new SoProfileCoordinate2;
   SoNurbsProfile *nTrim1        = new SoNurbsProfile;
   SoNurbsProfile *nTrim2        = new SoNurbsProfile;
   SoNurbsProfile *nTrim3        = new SoNurbsProfile;
   long trimInds[5];

   trimPts->point.setValues(0, 12, tpts);
   trimInds[0] = 0;
   trimInds[1] = 1;
   trimInds[2] = 2;
   trimInds[3] = 3;
   trimInds[4] = 0;
   nTrim1->index.setValues(0, 5, trimInds);
   nTrim1->knotVector.setValues(0, 7, tknots1);
   trimInds[0] = 4;
   trimInds[1] = 5;
```

```
    trimInds[2] = 6;
    trimInds[3] = 7;
    nTrim2->linkage.setValue(SoProfile::START_NEW);
    nTrim2->index.setValues(0, 4, trimInds);
    nTrim2->knotVector.setValues(0, 6, tknots2);
    trimInds[0] = 7;
    trimInds[1] = 8;
    trimInds[2] = 9;
    trimInds[3] = 4;
    nTrim3->linkage.setValue(SoProfile::ADD_TO_CURRENT);
    nTrim3->index.setValues(0, 4, trimInds);
    nTrim3->knotVector.setValues(0, 8, tknots3);

    surfSep->addChild(trimPts);
    surfSep->addChild(nTrim1);
    surfSep->addChild(nTrim2);
    surfSep->addChild(nTrim3);
    surfSep->addChild(surface);

    surfSep->unrefNoDelete();
    return surfSep;
}
```

进一步阅读建议

下列书籍可以提供有关 NURBS 曲线曲面更详细资料

Bartels, R., J. Beatty, and B. Barsky, An Introduction to Splines for Use in Computer Graphics and Geometric Modeling. Los Altos, Ca.: Morgan Kaufmann, 1987.

Farin, G., Curves and Surfaces for Computer Aided Geometric Design, 2e. San Diego, Ca.: Academic Press, Inc., 1990.

第九章 应用动作（Actions）

本章目标

在阅读完本章节后，我们将可以做下列事情：

- 绘制、渲染整个或部分场景。
- 打印输出场景。
- 利用场景的渲染图创建纹理贴图
- 计算物体的 3D 包围盒
- 计算物体累积的几何变换矩阵及其逆矩阵
- 将场景输出到文件中
- 搜索某些节点、搜索某类节点、或以名称搜索某些节点
- 拾取（pick）场景中的物体，获取物体的相关信息
- 编写回调函数，在 Inventor 遍历场景时，执行用户自定义的动作。
- 编写回调函数，获取 Inventor 形体产生的图元（primitives）信息（点、线、三角面）

本章将描述如何对 Inventor 场景使用动作。其实，在前面的章节中我们已经介绍过了 Inventor 最常用的动作－GL 渲染。这种动作会遍历整个场景，使用 OpenGL 库将场景绘制出来。本章将讨论使用动作的基本方法，以及给出像“拾取”、“计算包围盒”、“计算几何变换”、“输出场景”、“搜索场景”等这些动作相关的重要概念。

Inventor 动作

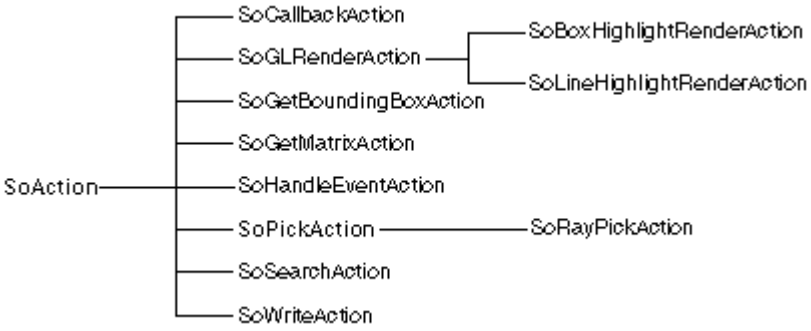
在前面的章节中，我们主要的精力放在如何使用组、属性、形体等这些节点来构建场景。一旦创建完场景，我们就可以针对这个场景使用*动作*。表 9-1 总结了我们使用 Inventor 动作的一些用法。

表 9-1 使用 Inventor 动作

完成此任务	可用这种动作
绘制、或渲染场景	SoGLRenderAction
计算物体的 3D 包围盒	SoGetBoundingBoxAction
计算当前累积的几何变换矩阵及其逆矩阵	SoGetMatrixAction
将场景输出到文件中	SoWriteAction
搜索某些节点、搜索某类节点、或以名称搜索某些节点	SoSearchAction
允许场景中的物体处理事件消息（见第 10 章）	SoHandleEventAction
沿着一个射线方向拾取场景中的物体	SoRayPickAction
遍历场景，累积遍历状态，然后利用回调函数，执行我们自定义的动作	SoCallbackAction

图 9-1 给出了部分动作类派生关系图

图 9-1 动作 类



基本方法

在场景中使用任何动作的时候，都应该按照下面的基本方法来使用：

1. 构造一个动作类的实例，初始化动作。我们可以在栈上构造动作实例（**所谓栈上构造，就是指函数内部直接定义一个动作类变量，而不像 Inventor 其它的对象类那样，必须使用 new 操作符动态来创建类实例。译者注**）。

```
SbViewportRegion region(300, 200);
SoGLRenderAction renderAction(region);
```

当然，也可以使用 new 操作符来动态创建一个动作实例

```
renderAction = new SoGLRenderAction(region);
```

如果我们使用 new 操作符创建动作，那么请不要忘记在使用完后要删除动作。

2. 为动作设置参数。例如，**SoGLRenderAction**类的构造函数需要指定视口区域和是否继承当前OpenGL设置这两个参数。如果我们指定：
SoGLRenderAction renderAction(region, TRUE);

则表示使用当前 OpenGL 中的线宽、材质等状态值。如果第二个参数为 FALSE（或忽略此参数）, Inventor 将使用自己的缺省设置。

3. 对节点、路径、或路径组使用动作。例如：
renderAction->apply(root);

4. 如果可用的话，获取动作的结果数据。有些动作带有一些附加的方法。例如

SoGetBoundingBoxAction类中有一个叫**getBoundingBox()**的方法，调用此方法可以返回动作计算的包围盒数据。另外一个叫做**getCenter()**的方法可以返回包围盒的中心位置。

应用动作

当对一个场景使用动作的时候，在场景中被遍历到的每个节点都将执行自己相对应的动作行为。有时，某个特定类型的节点对某种动作不做任何响应。例如，当**SoGetBoundingBoxAction**应用到**SoMaterial**节点上时，**SoMaterial**节点不做任何事情，因为**SoMaterial**只是一个属性节点，不是形体节点，所以它没有包围盒。也有时节点的动作行为很简单。例如，当动作应用到那些从**SoGroup**派生出来的节点上时，这些节点只是以特定的顺序对它们的子节点进行一些遍历操作而已。

当应用动作的时候，Inventor场景数据库管理着一个叫做“遍历状态”的数据结构（类似于OpenGL的渲染状态）。遍历状态是一个Inventor内部使用的类，用于在动作执行期间保存中间临时的状态数据（参数）。通常这种管理要求动作必须按照从上到下，从左到右的顺序来遍历场景。在遍历过程中，被遍历的节点将修改遍历状态中的元素数据。对于某些动作，例如像输出场景到文件（**SoWriteAction**），计算几何变换矩阵（**SoGetMatrixAction**），只需要维护很少，甚至不必维护遍历状态数据。在这种情况下，Inventor场景数据库就没必要再去跟踪管理场景中那些底层节点从顶层节点继承来的参数了（In these cases, the database does not need to keep track of all parameters inherited by nodes lower in the graph from the nodes above them.）。

在下面的章节中，我们将着重讨论的每一种类型的动作，讨论不同的节点是如何执行这些动作行为的。我们不必完全了解数据库是如何管理这些遍历状态的，我们只需要知道，当给定一个动作后，应该由那个节点来执行它，以及这些节点如何执行给定的动作。

动作可以在节点、路径、或路径组上使用。当动作应用到节点上时，动作将会遍历以此节点为根节点的子场景。当动作应用到路径上时，将遍历路径上的所有节点，如果有的话，也包括遍历位于路径上最后一个节点内的所有子节点。另外，还要遍历将会影响路径中节点的那些节点（通常，这些节点位于路径节点的左边和上边）（In addition, all nodes that affect the nodes in the path chain are also traversed (typically, these nodes are to the left and above the nodes in the path)）。应用动作到路径组和将动作应用到组中每个路径上很类似，除了对由两条或多条路径组成的子场景只遍历一次以外。

渲染

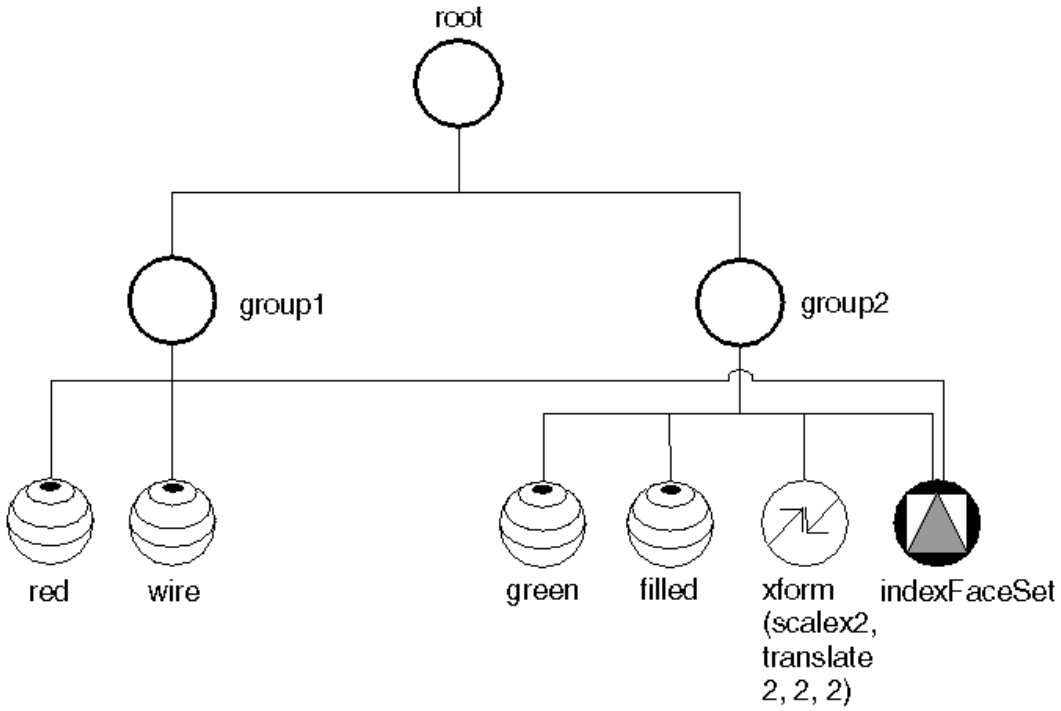
从第 3 章到第 8 章，我们已经解释过了不同的节点是如何执行**SoGLRenderAction**动作。这个动作可以将场景中的物体绘制在屏幕上。下面列出不同的节点执行**SoGLRenderAction**的行为：

- 如果节点是一个组节点，那么它将会按照特定的顺序访问自己的子节点。如果它是

一个**SoSeparator**节点，那么它在遍历其子节点之前要先保存当前的遍历状态，在遍历完后，还有恢复原来的遍历状态。

- 如果节点是一个属性节点，通常它会替换掉遍历状态中相对应的元素值（有些属性节点，例如**SoTransform**节点，会表现不同的行为）。例如：
  SoMaterial节点将替换当前材质数据。
  SoLightModel节点将替换当前光照模型。
  SoDrawStyle节点将替换当前绘制风格。
  SoCoordinate3节点将替换当前坐标数据。
- 如果节点是从**SoTransformation**类派生出来的，那么这个节点将修改当前几何变换矩阵（**注意，这里是修改，而不是替换。译者注**）。节点的变换矩阵将右乘（preconcatenated）当前的变换矩阵。
- 如果节点是一个形体节点，那么它将使用遍历状态中当前的元素值（例如，当前材质、当前纹理、当前灯光等），绘制出自己的形状。图 9-2 将显示一个位于两个不同组中的索引面集形体。当渲染 *group1* 时，使用遍历状态的当前元素值，结果是绘制了一个红色、框线风格的面集形体。因为接下来在 *group2* 中的节点修改了当前材质、绘制风格和变换矩阵，所以索引面集的下个实例显示的将是一个绿色，填充风格的面集形体。同时，绿色面集形体比红色面集形体大 2 倍，且位于一个新的位置上。

图 9-2 共享形体节点实例



设置透明品质

调用渲染动作的**setTransparencyType()**方法可以设置透明物体的渲染品质。Inventor 使用三种通用类型的透明渲染方式。*Screen-door transparency*类型使用填充图案（fill pattern）模拟透明效果。*Additive blending*类型是将透明物体的颜色和已在帧缓冲（frame buffer）里

的颜色相加作为透明结果颜色（***Additive blending*称为“饱和相加法”，就是将源像素和目标像素的RGB数值分别相加，如果某个RGB分量相加的结果超过了该分量的最大值，则结果取最大值。译者注**）。*Alpha blending*类型是按照“Alpha”因子的值来混合源像素和目标像素的颜色。根据真实度和渲染任务所需速度的要求，在通用的分类中还可以细分出3种*Additive blending*子类型和3种*Alpha blending*子类型。

关于 *Alpha blending* 的内容讨论请查看 OpenGL 编程手册（*OpenGL Programming Guide*）

透明品质级别

在 Inventor 中，可以指定下列的透明品质级别：

SCREEN_DOOR	使用 OpenGL 点画图案（stipple pattern）设置 screen-door 透明度。
ADD	使用饱和相加 OpenGL alpha 混合。
DELAYED_ADD	使用饱和相加法；先渲染不透明物体，最后渲染透明物体。
SORTED_OBJECT_ADD	使用饱和相加法。先绘制不透明物体，然后再绘制透明物体。根据透明物体与照相机的距离大小来排列透明物体的显示序列，按照从后到前的顺序显示透明物体。(这种类型的效果和 DELAYED_ADD 是一样的，因为加法是可交换的运算，与前后顺序无关）。
BLEND	使用 OpenGL alpha 混合。(见彩图 19)
DELAYED_BLEND	使用 OpenGL alpha 混合。先渲染不透明物体，最后渲染透明物体。（见彩图 20）。
SORTED_OBJECT_BLEND	使用 OpenGL alpha 混合。先绘制不透明物体，然后再绘制透明物体。根据透明物体与照相机的距离大小来排列透明物体的显示序列，按照从后到前的顺序显示透明物体。（见彩图 21）

权衡取舍

只有当透明物体要和已在帧缓冲中的物体进行混合时，才应该使用 ADD（或 BLEND）透明类型。这种透明类型是根据渲染物体的顺序进行透明计算的。

如果要确保透明物体最后被渲染，那么可以使用 DELAYED_ADD（或 DELAYED_BLEND）透明类型。例如，我们首先画一个透明的圆锥体，然后在圆锥的后面画一个不透明的圆柱体。如果使用 ADD 级别透明方式就不能看到透明效果。在这种情况下，我们必须使用 DELAYED_ADD（或 DELAYED_BLEND）透明类型。延迟级别的透明虽然比 ADD 或 BLEND 需要花费更多的计算时间，但却可以得到更真实的效果。

在渲染透明物体时，如果希望得到最佳真实的效果，我们须指定 SORTED_OBJECT_ADD(或 SORTED_OBJECT_BLEND）透明类型。这种透明类型可能需要最多的计算时间，但可以提供最佳的透明效果。这种类型总是在渲染完不透明物体后才开始渲染透明物体，并且根据物体与照相机的距离排列物体顺序，按照从后到前的顺序显示透明物体（见彩图 21）。

提示：像以面集方式显示的物体，它们内部的多边形是不能排序的，所以在这些物体中，有些多边形可能没有按照正确的透明顺序来绘制。如果物体是一个实体，那么使用一个带有恰当提示值（hints）的**SoShapeHints**节点，也许可以改善图像效果。

OpenGL 程序员请注意：如果使用延迟（delayed）或排序（sorted）透明级别，那么 Inventor 不会为透明物体更新 Z 缓冲，所以可以按照任意的顺序绘制透明物体（*Note to OpenGL programmers:* If you are using delayed or sorted transparency levels, Inventor does not update the *z* buffer for transparent objects so that they can be drawn in any order）。

如果使用**SoXtRenderArea**创建场景，我们可以调用**setTransparencyType()**方法为渲染透明物体指定透明品质级别。

反走样

SoGLRenderAction类还为反走样（*antialiasing*）提供了技术支持。反走样是一种用来消除或减少锯齿现象，使屏幕上的物体显示得更加光滑的技术。有两种反走样方法可供我们使用：

- 滤波方法：这是一种在处理时间方面相对“廉价”的方法。这种方法只适合对线和点进行反走样处理。
- 累积缓冲方法：这种方法比滤波方法需要更多的处理时间，但这种方法可以对整个屏幕图像做反走样处理，并且反走样的效果会更好。这种方法需要一个支持累积缓存的 OpenGL 窗口。

方法 1：滤波方法

使用**SoGLRenderAction::setSmoothing()**方法可以启动滤波反走样。**isSmoothing()**方法返回滤波标志的当前状态。这种形式的反走样技术只适合处理点和线图像。因为这种方法需要使用*Additive blending*或*Alpha blending*，所以当启动滤波反走样后，Inventor可能会在必要的时候修改透明类型。

方法 2：累积缓冲方法

正常情况下，Inventor每调用一次渲染动作就执行一次渲染流程（rendering pass）（**对于有些很复杂的图形效果，可能需要渲染多次，每次渲染一种效果，然后用GL_BLEND合成为最终效果。这种方法叫多道渲染（Multipass Rendering）。渲染一次就是一个pass，我们称为流程。译者注**）。我们可以调用**SoGLRenderAction::setNumPasses()**方法来增加每次渲染动作所执行的渲染流程次数。设置完后，Inventor会多次渲染场景，而且每次渲染场景时，会稍微移动一下照相机的位置，最后将多次渲染后的图像加起来取平均，作为渲染的最终结果。Inventor渲染场景的次数越多，反走样的效果也就越好。当然，增加渲染的次数的同时，必然会增加渲染场景的总时间。渲染流程次数的范围可以在 1 到 255（含 255）之间选择。如果设置次数为 1，则将禁止这种多道渲染反走样技术。

另外，如果我们将**SoGLRenderAction::setPassUpdate()**函数的参数设置为TRUE，那么在

每次渲染流程结束之后，Inventor会将累积缓冲区中的图形内容拷贝到当前激活的显示缓冲区中。这种设置虽然会降低渲染速度，但却可以允许我们看到在两次相邻的渲染流程之间，渲染图像发生了什么变化。**setPassUpdate()**函数的缺省参数是FALSE。

提示：可以使用**SoXtRenderArea::setAntialiasing()**方法来启动过滤反走样，并且可以为累积缓冲指定渲染流程次数。我们既可以使用此函数启用过滤反走样，也可以启用累积缓冲反走样，或者同时启用这两种反走样技术。

打印和离屏（Off-screen）渲染

SoOffscreenRenderer类可以打印Inventor的全部或部分场景图像。这个类将使用**SoGLRenderAction**动作将图像渲染到一个离屏内存缓冲区内。渲染缓冲区产生的图像即可以发送到PostScript打印机中（例 9-1），也可以当作一个纹理贴图来处理（例 9-2）。

位于缓冲区内的渲染图像可以是下列四种类型的图像格式：

LUMINANCE	单通道格式（灰度格式）
LUMINANCE_TRANSPARENCY	双通道格式（灰度外带 alpha 值）
RGB	三通道格式（真彩色）
RGB_TRANSPARENCY	四通道格式（真彩色外带 alpha 值）

在渲染图像之前，我们可以使用**SoOffscreenRenderer::setComponents()**方法来指定所产生的图像的通道数值。要打印黑白图像，须使用LUMINANCE参数。要打印彩色图像，须使用RGB参数。要产生一个带有透明内容的图像，须使用LUMINANCE_TRANSPARENCY或RGB_TRANSPARENCY参数。

提示：如果我们想将场景直接输出到打印机上，可以使用**SoXtPrintDialog**类，这是Xt库的一个组件。具体见*Open Inventor C++ Reference Manual*。

怎样为打印输出文件

要想将场景输出到Encapsulated PostScript (EPS)格式的文件中，我们必须首先使用离屏渲染器（off-screen renderer）来渲染场景，然后调用**writeToPostScript()**方法来产生PostScript输出，最后将这个输出写到指定的文件中。

例如，假设我们想打印一个 300×400 像素大小的屏幕区域。首先使用**SbViewportRegion**类中**setWindowSize**方法来指定要打印的视口（viewport）大小。然后按照上面的步骤输出屏幕图像。

```
SbViewportRegion vp;
vp.setWindowSize(SbVec2s(300, 400));

rootNode = getMyScene();

SoOffscreenRenderer renderer(vp);
renderer->render(rootNode);
renderer->writeToPostScript(stdout);
```

上段代码使用了缺省的每平方英寸所包含的像素数目（大约是 72）。我们可以使用**SbViewportRegion**类的**setPixelsPerInch()**方法来修改每平方英寸所包含的像素数目。通常，我们可以将打印机的分辨率设置为图像的分辨率。例如，对于一个打印分辨率为 300 DPI的打印机，我们可以按照下列代码设置渲染图像分辨率。

```
vp.setPixelsPerInch(300);
```

分辨率会影响到线宽、文字尺寸和点的大小。这些元素都是以像素为单位的。

如果我们想让打印的图片与屏幕上的图像大小相同。首先要使用**SbViewportRegion**类的**getViewportSizePixels()**方法得到视口区域的像素个数（在x和y轴方向）。然后使用**SoOffscreenRenderer**类的**getScreenPixelsPerInch()**方法得到以像素为单位的屏幕分辨率。如下面的代码段所示：

```
screenVp = renderArea->getViewportRegion();
SbVec2s screenSize = screenVp.getViewportSizePixels();
float screenPixelsPerInch =
        SoOffscreenRenderer::getScreenPixelsPerInch();
```

现在，我们将 x 和 y 方向的像素个数除以 *screenPixelsPerInch*，就可以计算出屏幕图像以英寸为单位的大小了。假设，我们要输出一个 300×400 像素大小的视口图像，视口的分辨率为 100 点每英寸，那么这个图像的大小就是 3×4 英寸。如果想让打印的图片与这个图像大小相同，我们应该按照下面的代码来设置参数：

```
vp.setWindowSize(SbVec2s(x_in_inches * printer_DPI,
        y_in_inches * printer_DPI));

vp.setPixelsPerInch(printer_DPI);
```

具体的OpenGL实现版本可能会限制视口的最大尺寸。使用**getMaximumResolution()**方法可以返回当前窗口系统所支持的视口最大分辨率。

例 9-1 将演示一个简单的函数，这个函数可以渲染场景，然后将渲染图像保存在一个

可被打印的文件中。

例 9-1 打印

```
SbBool
printToPostScript (SoNode *root, FILE *file,
   SoXtExaminerViewer *viewer, int printerDPI)
{
   // Calculate size of the image in inches which is equal to
   // the size of the viewport in pixels divided by the number
   // of pixels per inch of the screen device.  This size in
   // inches will be the size of the Postscript image that will
   // be generated.
   const SbViewportRegion &vp  = viewer->getViewportRegion();
   const SbVec2s &imagePixSize = vp.getViewportSizePixels();
   SbVec2f imageInches;
   float pixPerInch;

   pixPerInch = SoOffscreenRenderer::getScreenPixelsPerInch();
   imageInches.setValue((float)imagePixSize[0] / pixPerInch,
                        (float)imagePixSize[1] / pixPerInch);

   // The resolution to render the scene for the printer
   // is equal to the size of the image in inches times
   // the printer DPI;
   SbVec2s postScriptRes;
   postScriptRes.setValue((short)(imageInches[0])*printerDPI,
                          (short)(imageInches[1])*printerDPI);

   // Create a viewport to render the scene into.
   SbViewportRegion myViewport;
   myViewport.setWindowSize(postScriptRes);
   myViewport.setPixelsPerInch((float)printerDPI);

   // Render the scene
   SoOffscreenRenderer *myRenderer =
            new SoOffscreenRenderer(myViewport);
   if (!myRenderer->render(root)) {
      delete myRenderer;
      return FALSE;
   }

   // Generate PostScript and write it to the given file
   myRenderer->writeToPostScript(file);
```

```
   delete myRenderer;
   return TRUE;
}
```

创建纹理贴图

我们还可以将离屏渲染器产生的图像作为一个纹理贴图来使用。在这种情况下，首先使用**SoOffscreenRenderer::render()**方法渲染图像，然后使用**getBuffer()**方法得到图像缓冲区数据。

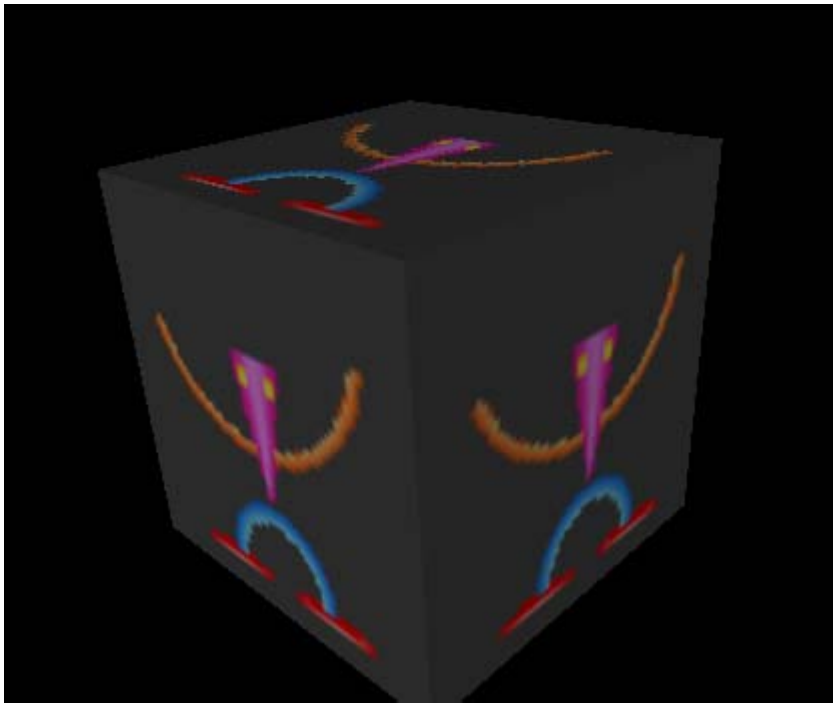
例 9-2 将演示使用渲染缓冲区来创建纹理贴图的通用步骤。

例 9-2 创建纹理贴图

```
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/SbViewportRegion.h>
#include <Inventor/misc/SoOffscreenRenderer.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoRotationXYZ.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTexture2.h>

SbBool
generateTextureMap (SoNode *root, SoTexture2 *texture,
   short textureWidth, short textureHeight)
{
   SbViewportRegion myViewport(textureWidth, textureHeight);

   // Render the scene
   SoOffscreenRenderer *myRenderer =
            new SoOffscreenRenderer(myViewport);
   myRenderer->setBackgroundColor(SbColor(0.3, 0.3, 0.3));
   if (!myRenderer->render(root)) {
      delete myRenderer;
      return FALSE;
   }
```

```
   // Generate the texture
   texture->image.setValue(SbVec2s(textureWidth, textureHeight),
            SoOffscreenRenderer::RGB, myRenderer->getBuffer());

   delete myRenderer;
   return TRUE;
}

main(int, char **argv)
{
   // Initialize Inventor and Xt
   Widget appWindow = SoXt::init(argv[0]);
   if (appWindow == NULL)
      exit(1);

   // Make a scene from reading in a file
   SoSeparator *texRoot = new SoSeparator;
   SoInput in;
   SoNode *result;

   texRoot->ref();
      in.openFile("jumpyMan.iv");
   SoDB::read(&in, result);

   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   SoRotationXYZ *rot = new SoRotationXYZ;
   rot->axis  = SoRotationXYZ::X;
   rot->angle = M_PI_2;
   myCamera->position.setValue(SbVec3f(-0.2, -0.2, 2.0));
   myCamera->scaleHeight(0.4);
   texRoot->addChild(myCamera);
   texRoot->addChild(new SoDirectionalLight);
   texRoot->addChild(rot);
   texRoot->addChild(result);

   // Generate the texture map
   SoTexture2 *texture = new SoTexture2;
   texture->ref();
   if (generateTextureMap(texRoot, texture, 64, 64))
      printf ("Successfully generated texture map\n");
   else
      printf ("Could not generate texture map\n");
   texRoot->unref();
```

```
    // Make a scene with a cube and apply the texture to it
    SoSeparator *root = new SoSeparator;
    root->ref();
    root->addChild(texture);
    root->addChild(new SoCube);

    // Initialize an Examiner Viewer
    SoXtExaminerViewer *viewer =
             new SoXtExaminerViewer(appWindow);
    viewer->setSceneGraph(root);
    viewer->setTitle("Offscreen Rendered Texture");
    viewer->show();

    SoXt::show(appWindow);
    SoXt::mainLoop();
}
```



缓存（Caching）

缓存用来保存一个操作的中间结果，这样以后在使用此操作时，可以避免重复计算。Inventor提供两种类型的缓存：渲染缓存和包围盒缓存（在“计算包围盒”章节中将详细描述**SoGetBoundingBoxAction**的用法）。我们可以将渲染和包围盒这两种动作的遍历结果保存在缓存中。例如，渲染缓存中含有一个OpenGL显示列表，这个显示列表保存着被渲染场景的遍历结果。如果场景没有发生改变，那么Inventor会在下次渲染的时候直接使用缓存中的内

容，而不必再次遍历场景。

SoSeparator节点中有两个用于缓存的域。域值可以是AUTO、ON或OFF。缺省值为AUTO。

renderCaching (SoSFEnum)	设置是否使用渲染缓存。域值 AUTO 表示当场景中隔离节点的内容没有发生变化时，将启用渲染缓存。域值 ON 表示无论缓存是否能提高效率，Inventor 都会试图建立一个渲染缓存。域值 OFF 表示不使用渲染缓存。
boundingBoxCaching (SoSFEnum)	设置是否使用包围盒缓存

SoSeparator类中有一个**setNumRenderCaches()**成员函数，这个成员函数可以设置每个**SoSeparator**节点同时能建立多少个渲染缓存。当然创建的缓存越多，所需要的内存也就越多。例如，如果一个场景在“框线”和“填充”这两种绘制模式下切换显示，那么我们可以将绘制模式设置在缓存之外，使用两个缓存分别保存两种绘制模式下渲染结果。（**这样做可以有效地提高切换场景的显示效率。译者注**）。这个成员函数只对调用它之后所创建的**SoSeparator**节点产生影响。如果在某些**SoSeparator**节点创建前，将渲染缓存区个数设置为 0，那么这些**SoSeparator**节点将关闭渲染缓存。渲染缓存缺省的个数为 2。

提示：如果渲染缓存是 AUTO，Inventor 将会为缓存渲染结果而进行多次渲染。Inventor 的缓存机制需要进行若干次比较渲染后，来确定场景内容是否发生了改变。如果没有发生改变，才可以缓存场景。（If render caching is AUTO, it will take several renderings for caching to take effect .caching mechanism requires several renderings for comparison to determine that nothing is changing and the scene can be cached.）

缓存的工作机制

缓存的处理过程起始于SoSeparator节点。具体的步骤如下：

1. Separator 组节点首先检查是否存在一个有效的缓存。
2. 如果存在有效的缓存，Separator 组节点将忽略其下的子场景，遍历将直接使用缓存内容作为 Separator 组节点的遍历结果。
3. 如果不存在有效的缓存，Separator 组节点将通过检查相关的域值来决定是否应该创建一个缓存。
4. 如果当前的缓存开关标志是 ON，Inventor 会强制打开一个缓存，然后遍历 Separator 组中的所有节点，并将遍历结果记录在缓存中，最后使用缓存内容作为遍历的结果。如果当前的缓存开关是 AUTO，Inventor 将使用一个特殊的状态集来确定是否有必要对当前 Separator 组节点创建缓存。

对于Separator组中的节点，有可能会继承位于它所在组之前的节点的属性值。例如，几乎所有的形体节点都有可能会使用到材质、坐标、纹理坐标、复杂度、法向向量以及绑定

等这些节点。如果这些节点的数据发生了改变，那么缓存内容也需要做修改。（注意，如果位于缓存外的纹理数据发生了改变，当前的缓存仍然有效。因为，形体节点不会对OpenGL发送纹理调用。只有当遍历到SoTexture2节点时，SoTexture2节点会直接将纹理数据发送给OpenGL）。

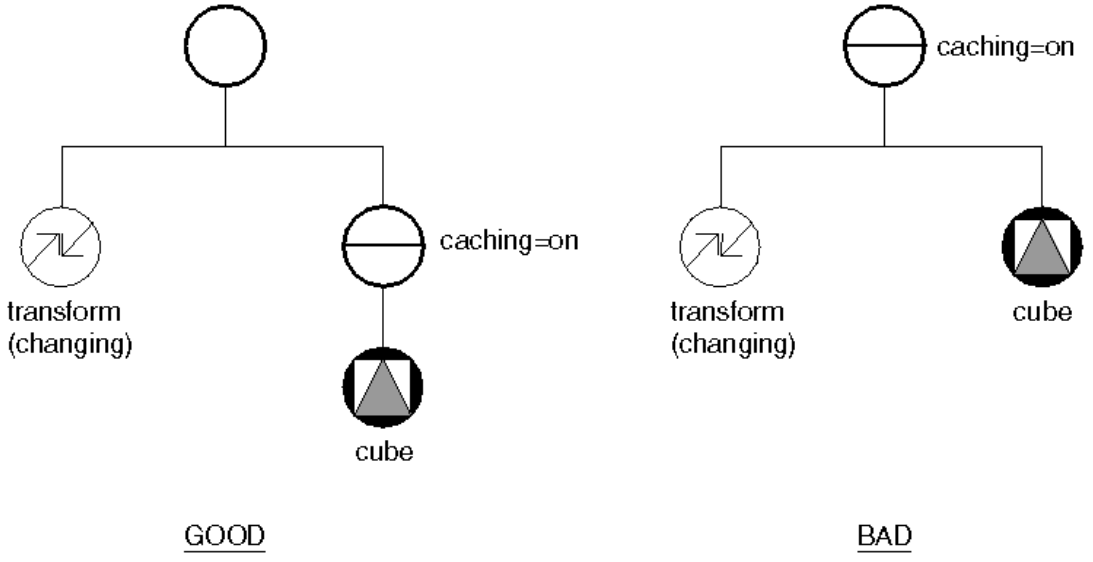
下面的这些修改也许会使当前的缓存失效。

- **SoText2**节点，修改字体和照相机的数据（因为这种文本总是与屏幕对齐的）
- **SoText3**节点，修改轮廓坐标数据或者轮廓的类型。

在确定当前缓存是否是有效这个问题上，Inventor 是非常谨慎的。（也就是说，即使那些继承下来的数据没有发生改变，缓存也可能会失效，这时 Inventor 将不得不重新建立缓存）

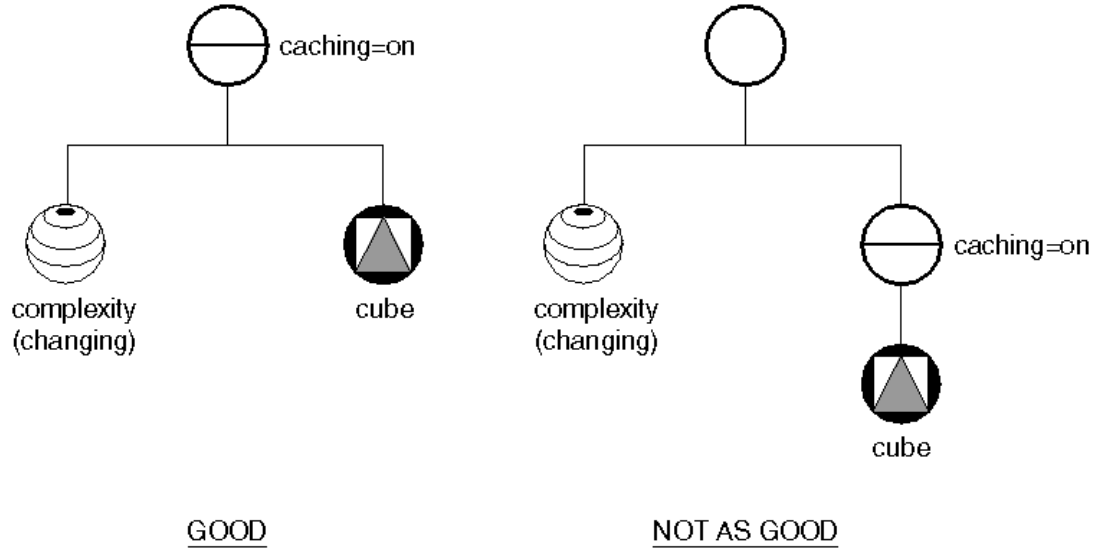
在图 9-3 所示的场景中，有一个频繁改变数据的几何变换节点和一个立方体节点。在这种情况下，如果开启立方体上面的 separator 节点的缓存开关，修改几何变换节点的数据将不会使立方体的缓存失效。（在图 9-3 的右面部分中，立方体节点上面没有一个单独的 separator 节点，而是立方体和几何变换节点同时位于一个 separator 节点下，如果打开这个 separator 节点的缓存开关的话，因为几何变换节点频繁地修改数据，导致立方体的缓存数据不断失效，所以这种情况下，缓存不会提高渲染的效率。译者注）

图 9-3 缓存形体



在图 9-4 所示的场景中，有一个频繁改变数据的复杂度节点和一个立方体节点。在这种情况下，我们应该将属性节点和形体节点放在同一个缓存里面，因为渲染形体节点总是要使用这个属性节点。（在图 9-4 的右面部分中，在立方体上面增加了一个单独的 separator 节点，并且打开了 separator 节点的缓存开关。在这种情况下，这个 separator 节点的缓存没有多大的效果。因为复杂度节点频繁地改变数据，导致立方体的渲染缓存不断失效。 因此图中左右两部分的缓存所产生的效果基本上是相同的，这两种缓存都不会大幅提高渲染效率。只不过右边的场景多了一个 separator 节点，所以在遍历的时候，右边的场景会因多遍历一个节点而导致效率稍微变差些。译者注）

图 9-4 一起缓存形体节点和一个不断变化的属性节点



权衡利弊

渲染缓存会消耗掉大量的内存。但渲染缓存也会对提高渲染速度起到至关重要的作用。如果使用AUTO方式的渲染缓存，那么Inventor将自己确定创建缓存是否有助节省渲染时间，提高渲染效率。

包围盒缓存相对来说是比较“经济”的。Inventor 利用包围盒缓存来提高“拾取”检测的速度。如果启用了包围盒缓存，当用户想点击拾取某个 separator 组所包含的子场景时，separator 节点会首先检查点击位置是否位于自身的包围盒范围内，如果点击位置不在包围盒内，separator 节点就会知道用户没有点击自己所包含的节点，所以就不必再遍历子场景了。

剔除部分场景

如果我们正在处理一个尺寸巨大的场景，而且我们还知道照相机将只是频繁地观察场景中的某些部分区域，这时我们就可以启用Inventor的渲染剔除功能。使用渲染剔除功能后，Inventor就可以不必浪费时间去渲染完全位于照相机视野之外的场景了。在**SoSeparator**节点中有两个有关渲染剔除的标志位数据：**renderCulling**和**pickCulling**。**renderCulling**的缺省值为AUTO，**pickCulling**的缺省值为ON。

下面将讨论渲染剔除的处理过程。（拾取剔除的处理过程和渲染剔除非常类似，而且拾取剔除相对来说是比较“经济”的。通常我们只需要简单设置为 ON 即可）。下面是一个有关渲染剔除处理过程的概要总结：

1. 当遍历到一个照相机节点时，照相机节点将会把自己所定义的一个位于全局空间（或称世界空间）下的可视体（view volume）放置在遍历状态中。

2. 当遍历到separator节点时，separator节点会查看它的**renderCulling**域值。如果域值为ON，那么separator节点将按照下列步骤剔除无用的场景区域：

- 在物体空间坐标系下，计算 separator 的包围盒数据。（这个数据可能已经位于缓存中）
- 将包围盒数据从物体空间坐标系下变换到世界空间坐标系下，然后让它与遍历状态中的可视体进行比较。
- 如果包围盒完全位于当前可视体范围之外，那么这个 separator 节点就不会在遍历它的子节点了。

因为步骤 2（计算包围盒并检测）还是相当花费时间的，所以缺省情况下，渲染剔除功能是关闭的。我们需要先对自己的场景做一个大概的评估，以此来决定渲染剔除是否真正有助于提高效率。例如，我们有一个场景，其中包含有围墙、复杂的电器设备、和连接管道等物体。尽管这个场景很复杂，但剔除功能并不会有助于提高显示效率。因为所有的物体都同时显示在照相机的视野中（**如果启用渲染剔除，反而会因为进行多余的剔除运算，导致效率降低。译者注**）。然而，对于那些物体非常分散地位于空间中的场景，例如一个太阳系场景，剔除功能还是非常有用的。

提示：为了更有效的使用剔除功能，应该首先按照空间位置来组织场景数据库。在 3D 空间内相互靠近的物体应该放在同一个 separator 节点下，彼此分隔很远的物体应该放在不同的 separator 节点下。例如在上面带有围墙的场景中，我们应该分别将每个墙面上的电器和管道都放置在一个 separator 节点下。

下面是一些使用渲染剔除的指导方针：

- 通常情况下，不要在一个必须使用渲染缓存的separator节点（即**SoSeparator**节点的**renderCaching**域显式地设置为ON）下增加一个使用剔除功能的separator节点。 可以在渲染缓存设置为OFF或AUTO的**SoSeparator**节点下增加一个剔除separator节点。

  提出这条执导方针的原因是因为剔除操作主要依靠场景中的照相机。如果一个 separator 节点做出了自己要被剔除的决定，那么它的任何缓存是否有效都将依靠于当前照相机。依靠照相机的缓存数据是经常会失效的，因为绝大多数的应用程序都需要频繁地修改照相机的参数。

  如果在一个 separator 节点中同时启用渲染剔除和渲染缓存这两个功能(或启用渲染剔除，渲染缓存设置为 AUTO)，还是可以提高显示效率的。
- 只对空间中相互分离的物体使用剔除功能。
- 只对具有大量多边形的物体使用剔除功能。如果物体剔除计算所花费的时间比绘制物体还多的话，那就没有必要再使用剔除功能了。

计算包围盒

包围盒动作用于计算包围物体的 3D包围盒数据。物体可以是某个节点下的子场景，也可以是一个路径所定义的物体。(The bounding-box action computes a 3D bounding box that encloses the shapes in a subgraph under a node or defined by a path.)。这个动作还可以计算出物体的中心点位置（见例 9-3）。**SoGetBoundingBoxAction**通常是对路径进行调用，这样做我们将可以获得物体在世界坐标系下的包围盒。包围盒动作返回一个**SbBox3f**数据变量，这个数据在世界坐标系下，沿着x，y，z轴平行方向定义了一个 3D立方体。

创建包围盒动作实例

下面是一个创建**SoGetBoundingBoxAction**实例的例子代码：

```
SbViewportRegion vpReg;
vpReg.setWindowSize(300, 200);
SoGetBoundingBoxAction bboxAction (vpReg);
```

SoGetBoundingBoxAction类的构造函数只需要一个视口区域参数，通常只有在计算以屏幕对齐（screen-aligned）或以屏幕像素为单位大小（screen-sized）的物体的包围盒时，例如**SoText2**，才需要使用这个参数。

（下面的翻译来自Coin3D 库的在线帮助文档，也许可以帮助读者更好地理解上面的内容。文档的URL为：http://doc.coin3d.org/Coin/classSoGetBoundingBoxAction.html ；译者注。

看上去**SoGetBoundingBoxAction**的构造函数没有必要为计算包围盒而传入一个视口区域参数，之所以这样做的一个原因是：有些形体节点在计算它们的包围盒时需要知道视口区域的大小，这些形体节点包括**SoText2**和**SoImage**等.

这些节点比较特殊的原因是：它们其实是 2D 图形，但它们却要显示在屏幕“表面”上，好像它们正在一个 3D 场景中。

为了准确地计算出显示在屏幕“表面”上的 2D 图形的包围盒，就需要将这些图形所占据的屏幕区域“向里投影”(de-project）到场景中特定深度上的 2D 矩形平面上。这种“向里投影”(de-project）操作需要知道视口尺寸信息。

还有，有些 3D 形体，例如像 SoNurbsSurface 节点，当场景中有一个 SoComplexity 节点，并且 SoComplexity 节点的 SoComplexity::value 设置为 SCREEN_SPACE 时，有可能会使 SoNurbsSurface 节点显示稍微发生变形。因而，这些节点也有必要知道视口区域参数，才能正确计算出它们的包围盒数据。

通常情况下，传入的视口区域大小和当前渲染区的大小相等。如果已经建立了一个 viewer 观察窗口，那么按照下面的代码很容易就可以得到视口区域参数：

```
const SbViewportRegion & vpreg = viewer->getViewportRegion();
```

如果程序在计算包围盒时还没有一个合法的 Viewer 或渲染区存在的话，其实可以不必太重视这个参数。相对于传入一个准确的视口区域参数来说，如果我们传入的参数是我们自己估计的数值，虽然会导致包围盒计算发生微小的误差，但通常都是很难被用户所能察觉到的）

应用包围盒动作

SoGetBoundingBoxAction节点可以应用在一个子场景的根节点上、一个路径上、或一个路径列表上。

获取结果

有三个函数可以获得**SoGetBoundingBoxAction**的计算结果：

getBoundingBox() 返回一个物体的包围盒。返回的数据类型为 SbBox3f

getCenter() 返回物体的中心点位置

getXfBoundingBox() 返回一个物体的包围盒。返回的数据类型为 SbXfBox3f

对于不同种类的物体，通过**getCenter()**所返回的中心点位置，其含意也都不相同。例如，**SoFaceSet**节点是使用其所有顶点坐标的平均值来当作中心点。组节点的中心点是使用这个组内所有物体的中心点位置的平均值来定义。

SbXfBox3f数据内部保存着一个物体原始的包围盒数据，以及一个用于将包围盒数据变换到世界坐标系下的几何变换矩阵。使用**SbXfBox3f**而不用**SbBox3f**的优势是，有时我们没有必要对包围盒进行变换。有时我们也许希望对包围盒做另外不同的变换。

例 9-3 给出了如何使用**SoGetBoundingBoxAction**来计算出物体中心点位置的代码。这个中心位置将来可以作为一个几何变换矩阵的旋转中心。

例 9-3 设置几何变换节点的中心域值

```
SbViewportRegion myViewport;
SoTransform *myTransform;

SoGetBoundingBoxAction bboxAction(myViewport);
bboxAction.apply(root);
myTransform->center = bboxAction.getCenter();
```

累积计算几何变换矩阵

对于任何从**SoTransformation**派生出来的节点或一个路径来说，使用**SoGetMatrixAction**可以返回这些节点或路径的当前变换矩阵。当我们将动作应用到一个**SoTransformation**节点上时，动作将返回这个节点的变换矩阵。如果我们将动作应用到一个路径上时，**SoGetMatrixAction**将会对这个路径所定义的子场景中所有的几何变换节点累积计算出一个变换矩阵。这个动作允许我们将一个坐标空间转换到另外的一个坐标空间内。（**也就是说这个动作可以产生一个从某个坐标系变换到另外一个坐标系的变换矩阵。译者注**）。当路径的头节点是整个场景的根节点时，所产生的变换矩阵表示的是从局部坐标系变换到世界坐标系下。

SoGetMatrixAction与其它动作的一个重要的区别是：当对场景中的节点或路径应用**SoGetMatrixAction**时，**SoGetMatrixAction**不会向下遍历这个节点。（**对于某个场景节点来说，使用SoGetMatrixAction不会去遍历这个场景中的子节点，而其它动作都是需要对场景中的子节点进行遍历操作。译者注**）。当动作应用到一个节点上时，它仅仅返回节点的当前变换矩阵（因此，**SoGetMatrixAction**只对几何变换节点感兴趣，而对于其它类型的节点，它只返回一个单位

矩阵）。 当动作应用到一个路径上时，它会收集路径中所有节点的几何变换信息，不过，只要搜索到了路径链中最后一个节点时，这个动作会立即停止遍历。

创建变换矩阵动作实例

SoGetMatrixAction类的构造函数不需要任何参数。

```
SoGetMatrixAction mtxAction;
```

应用动作

SoGetMatrixAction可以应用在节点或路径上。

获取结果

SoGetMatrixAction有两个方法用来返回计算结果

getMatrix() 返回节点或路径的累积变换矩阵，数据类型是 SbMatrix

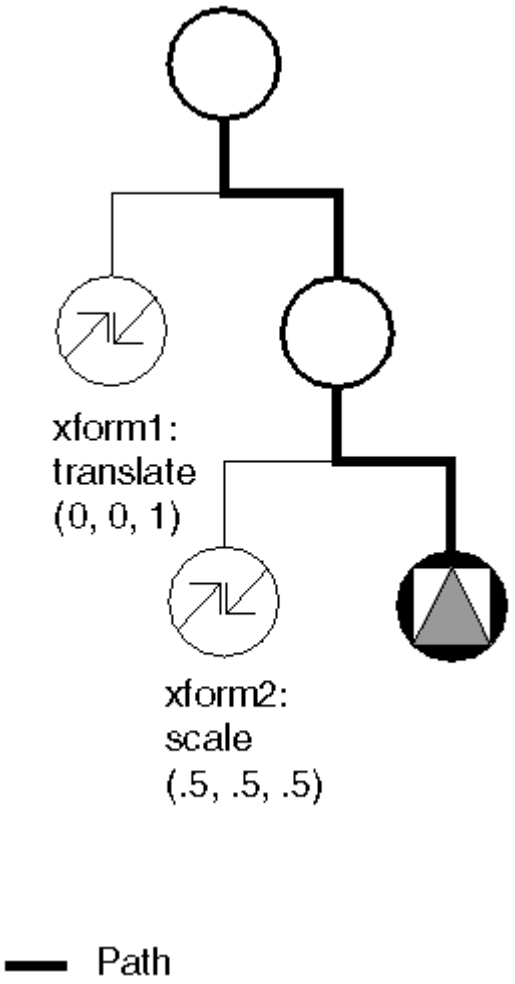
getInverse() 返回节点或路径的逆累积变换矩阵，数据类型是 SbMatrix

getInverse方法返回的变换矩阵可以将位于世界坐标系下的点映射到某个物体局部坐标系下。在*Open Inventor C++ Reference Manual*书中详细描述许多针对**SbMatrix**方便有效的函数。例如，可以使用**multVecMatrix()**函数将一个位置点变换到矩阵所定义的空间中。使用**multDirMatrix ()**函数将一个方向矢量变换到矩阵所定义的空间中。（在Inventor中，所有的向量都假设为行向量）。

提示：我们可以将一个位于A物体坐标系下的点转换到另外一个B物体坐标系下，首先对A物体应用**SoGetMatrixAction** ，通过使用**getMatrix()**函数所返回的变换矩阵，将点从A物体坐标系下变换到世界坐标系下。然后对B物体应用**SoGetMatrixAction** ，通过使用getInverse **()**函数所返回的变换矩阵，将点从世界坐标系下变换到B物体坐标系下。

作为一个例子，假设**SoGetMatrixAction**应用到图 9-5 中的路径上，*xform1* 节点包含有一个（0.0,0.0,1.0）的平移变换。*xform2* 节点包含有一个（0.5,0.5,0.5）的缩放变换。

图 9-5 将**SoGetMatrixAction**应用到路径上



每个新变换节点都被预乘（premultiplied）到当前变换矩阵中，在图 9-5 所示的情况下，矩阵将按照下面的顺序进行累加运算：

scale (xform2)

$$\begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

translate (xform1)

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

在这个例子中，**getMatrix()**返回如下的矩阵结果：

$$\begin{bmatrix} 0.5 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix}$$

可以使用**getTextureMatrix()**和**getTextureInverse()**方法，返回纹理的坐标变换数据。详细内容请查阅本书第 7 章和*Open Inventor C++ Reference Manual*书中的内容。

输出到文件

Inventor的场景内容可以按照ASCII或二进制格式写进一个文件中。**SoWriteAction**可用于将Inventor的场景数据输出到文件中。这个类的实例中包含有一个**SoOutput**实例，缺省情况下，**SoOutput**是以ASCII格式向**stdout**控制台输出场景数据。**getOutput()**方法返回一个指向**SoOutput**的指针。**SoOutput**包含的其它方法如下：

openFile()	打开并输出数据到一个指定的文件中，此文件不是**stdout**控制台文件
setFilePointer()	设置输出文件的指针
closeFile()	关闭被 openFile()打开的文件。当动作结束后，文件会自动被关闭。
setBinary()	如果参数为 TRUE，则按照二进制格式写文件；如果参数为 FALSE，则按照 ASCII 格式写文件（缺省方式）。
setBuffer()	将场景数据写入一个内存缓冲区中，而不是写入文件中。

例如，将场景以二进制格式写到一个文件中，这个文件由 *fp* 指针指定：

```
SoWriteAction myAction;
FILE *fp;

myAction.getOutput()->setBinary(TRUE);
myAction.getOutput()->setFilePointer(fp);
myAction.apply(root);
```

以 ASCII 格式输出到一个文件中：

```
SoWriteAction myAction;

myAction.getOutput()->openFile("myFile.iv");
myAction.getOutput()->setBinary(FALSE);
myAction.apply(root);
myAction.getOutput()->closeFile();
```

我们将在第 11 章详细介绍Inventor的文件格式。下面是**SoWriteAction**输出一个子场景的内容：

```
#Inventor V2.0 ascii
Separator {
           Separator {
                 Transform {
                                  scaleFactor 1 2 1
                 }
                 Material {
                                 ambientColor .2 .2 .2
                                 diffuseColor .6 .6 .6
                                 specularColor .5 .5 .5
                                  shininess .5
                 }
                 Cube{
       }
         }
}
```

搜索节点

SoSearchAction可以沿着场景路径，搜索某个特定的节点、某些类型的节点、或某个带有给定名称的节点。首先，我们需要初始化这个动作。然后指定要搜索的*节点*、*节点类型*、*或节点名称*（或这些参数的组合）。如果指定的是节点类型，那么我们还可以指定，搜索类型是需要和指定类型精确匹配，还是搜索类型也可以是指定类型的子类型。

指定搜索标准

首先，我们需要指定想搜索什么，是否需要找到所有匹配的结果，以及如何遍历场景。

搜索节点

我们可以使用**setNode()**方法来搜索一个特定的节点（由指针来指定）。例如，我们可以使用**setNode()**方法来搜索一个特定的光源节点，这样我们就可以关联并编辑这个光源（使用第 14 章中的光源编辑器关联并编辑光源。译者注）

搜索节点类型

有的时候，我们也许希望搜索某一种类型的节点，而不是只搜索某个特定节点。当要搜索一个节点类型的时候，我们需要选择，是只搜索这个特定类型的所有节点，还是同时允许搜索这个类型的派生类型的节点（缺省方式）。**setType()**的语法如下：

setType(SoType *t*, int derivedIsOk = TRUE);

搜索节点名称

使用**setName()**方法来指定要搜索的节点名称（见第 3 章中更多有关“命名”的内容）。

指定是否搜索所有匹配的节点

使用**setInterest()**方法来指定返回何种形式的路径：

FIRST 只返回第一次匹配时的节点路径（缺省）

LAST 只返回最后一次匹配时的节点路径

ALL 返回所有匹配的节点路径

指定遍历类型

使用**setSearchingAll()**方法可以指定：是按照正常遍历方式来搜索场景（沿着开关组节点和隔离节点的遍历顺序），还是搜索场景中的所有节点，不管开关组节点是如何设置的。缺省值是FALSE（即按照正常遍历顺序）。

应用动作

SoSearchAction和其它的动作有相同的使用方法。

获取结果

使用下面的方法可以得到搜索的结果：

getPath() 返回所找到的路径(如果 interest 是 FIRST 或 LAST)

getPaths() 返回所找到的所有路径列表(如果 interest 是 ALL)

*Open Inventor C++ Reference Manual*书中详细描述了**SoSearchAction**的所有可用的方法。

下面的例子将演示如何在场景中找到任何从**SoLight**派生出来的节点。如果没有找到的话，则代码将自己创建一个**SoDirectionalLight**节点。这个例子只搜索第一次匹配的节点，因为它调用了**setInterest(SoSearchAction::FIRST)**。

```
SoSearchAction mySearchAction;

// Look for first existing light derived from class SoLight
mySearchAction.setType(SoLight::getClassTypeId());
mySearchAction.setInterest(SoSearchAction::FIRST);

mySearchAction.apply(root);
if (mySearchAction.getPath() == NULL) { // No lights found

   // Add a default directional light to the scene
   SoDirectionalLight *myLight = new SoDirectionalLight;
   root->insertChild(myLight, 0);
}
```

拾取（Picking）

SoRayPickAction可以沿着一条从照相机开始，穿过可视体近平面上一点的射线，寻找所有位于这条射线路径上物体对象。通常都是以窗口空间内的像素坐标来定义这条射线。当我们使用这个动作时，**SoRayPickAction**将遍历所应用的场景，然后按照从近到远的顺序，返回射线穿过的所有物体的路径。拾取动作主要对几何体、几何变换和形体节点感兴趣。

技巧：**SoSelection**节点不必使用拾取动作来选择物体，它可以自动拾取物体。

SoHandleEvent 动作也会自动执行拾取操作。另外，**SoEventCallback**节点还允许我们注册一个回调函数，无论何时，只要在某个物体上发生了一个特定的事件（例如鼠标按下），都会自动调用那个回调函数。在第 10 章中有更多有关**SoSelection**、**SoHandleEvent**、和**SoEventCallback**的介绍。

拾取类型

缺省情况下，场景中的所有物体都是可以拾取的（即使是不可见物体和透明物体）。要想使拾取动作对某个物体或一组物体失效，可以先在场景中插入一个**SoPickStyle**节点，然后将这个节点的**style**域值设置为UNPICKABLE。这样，场景中**SoPickStyle**节点之后的所有物体都将不能被拾取，直到**SoPickStyle**节点被重新设置为SHAPE（拾取点的位置位于物体表面上）或BOUNDING_BOX（拾取点的位置位于物体的包围盒上）。BOUNDING_BOX拾取类型通常用于**SoText3**节点。像所有其它的属性节点一样，**SoSeparator**组节点也会保存和恢复当前的拾取类型。

创建拾取动作实例

SoRayPickAction的构造函数带有一个视口区域的参数。下面是创建**SoRayPickAction**实例的代码：

```
SbViewportRegion myViewport;
SoRayPickAction myPickAction(myViewport);
```

当遇到像**SoText2**这样的以屏幕对齐方式显示的物体时，视口区域参数用于计算这类物体的包围盒数据（**见计算包围盒章节中关于视口区域参数必要性的解释。译者注**）。

设置参数

在使用拾取动作之前，我们需要先设置下列的参数：

- 设置拾取射线的方向
- 对于射线所经过的所有物体来说，是将这些物体全部返回，还是只返回离照相机最近的物体。

拾取射线可以使用两种方式来定义：即可以用一个窗口点和一个射线半径来定义，也可

以在世界坐标空间内指定一个点和方向来定义。第一种定义方法更多地用于交互式应用程序，因为用户通常只对鼠标位置下的区域感兴趣。

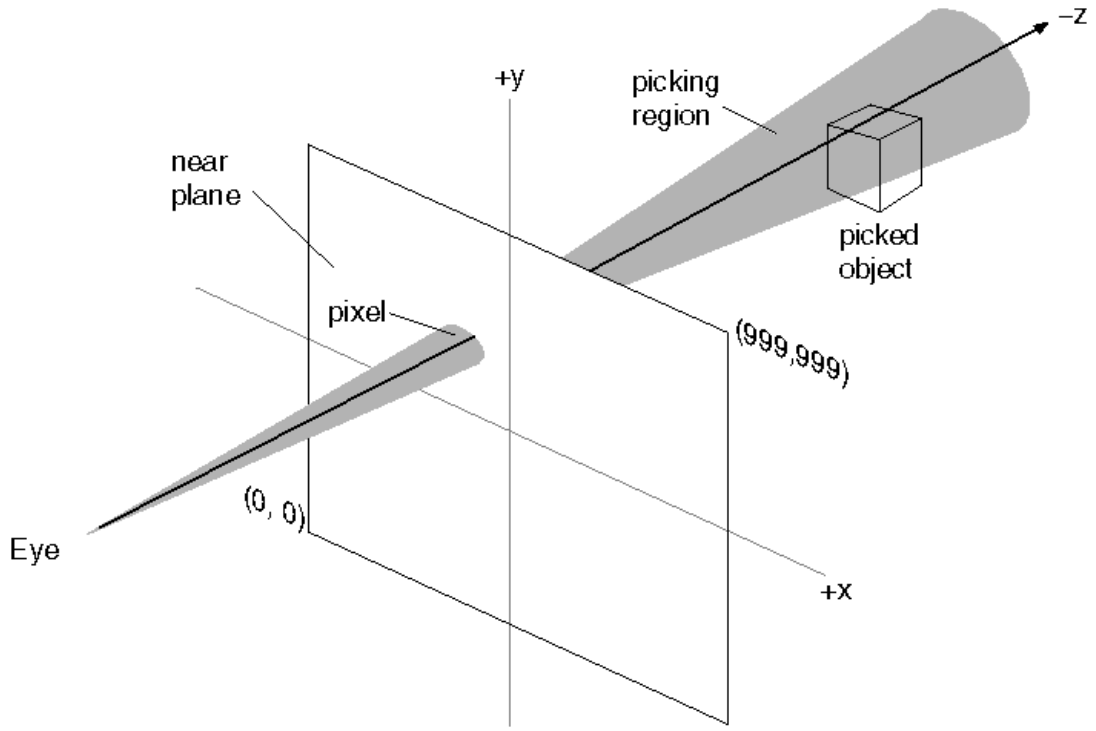
利用一个窗口点坐标设置拾取射线

在我们使用拾取动作之前，需要调用**setPoint()**和**setRadius()**方法来设置拾取射线的参数。拾取射线通常都是定义在视口坐标系下（*viewport coordinates*），视口窗口的左下角是（0,0）坐标，右上角是（vpWidth-1, vpHeight-1）坐标（见图 9-6）。在图 9-6 中，视口窗口的范围是 1000×1000。照相机的近平面映射到了拾取视口上。

为了更容易地拾取“点”和“线”，拾取射线可以扩张成一个圆锥体（针对投影透视照相机。见图 9-6），或扩展成一个圆柱体（针对平行投影照相机）。使用**setRadius()**方法可以设置圆锥体或圆柱体与照相机近平面相交处横截面的大小（缺省半径是 5 个象素）。被拾取的物体必须位于圆锥体（或圆柱体）范围之内：

- 对于“点”和“线”，如果它们的任何部分位于圆锥体之内，那么它们就可以被拾取。（一个以 LINES 绘制方式显示的球体仍然当作实体球进行选取）。
- 对于所有其它的物体，必须和拾取射线相交才能被拾取。

图 9-6 圆锥体表示投影透视照相机的拾取射线



在世界坐标空间内定义拾取射线

我们也可以在世界坐标空间内指定一条沿着拾取方向的射线来定义拾取射线。这条射线是由一个起始点，一个方向向量，和拾取物体的远近距离（**当物体不在最近和最远这两个数据所定义的范围内时，这个物体将不会被拾取到。译者注**）来定义的。这种定义方式不需要半径参数。例如：

```
SbViewportRegion viewport(400, 300);
SbVec2s cursorPosition(250, 125);

SoRayPickAction myPickAction(viewport);

myPickAction.setRay(SbVec3f(0.0, 0.0, 0.0),  // starting point
                    SbVec3f(0.0, 0.0, -1.0); // direction vector
```

上面的例子使用了 Inventor 缺省的远近拾取距离，缺省距离就是照相机的近平面和远平面的距离。

拾取最近的物体

使用**setPickAll()**方法可以指定拾取动作是返回所有被拾取的物体(按照从近到远的顺序排列)，还是只返回离照相机最近的那个物体。设置参数为TRUE，表示返回所有被拾取的物体；设置参数为FALSE（缺省值），表示返回最近的那个物体。

使用拾取动作

拾取动作可以应用在一个节点上、一个路径上、或一个路径列表上。下面的代码是将拾取动作应用到场景的根节点上：

```
pickAction->apply(rootNode);
```

获取结果

拾取动作的结果是保存在一个**SoPickedPoint**类（第一个被点击的物体）或一个**SoPickedPointList**类（所有被点击的物体）。调用**SoPickedPoint**的方法可以获得这些信息。

SoPickedPoint类代表着被拾取物体表面上的一个点。**SoPickedPoint**类中包含有下列信息：拾取射线与物体的交点，交点处的表面法向向量和纹理坐标，当前材质集的索引值，和一个指向被拾取物体的路径。使用下列方法可以得到这些信息：

getPoint()	返回拾取射线与物体的交点。交点定义在世界坐标空间内。
getNormal()	返回交点处的表面法向向量，向量定义在世界坐标空间内
getTextureCoords()	返回交点处的纹理坐标，坐标定义在纹理贴图的坐标空间内。
getMaterialIndex()	返回交点处当前材质集的索引值。如果交点处的材质是由两个顶点差值计算所产生的，那么将返回与交点最近的那个顶点的材质索引值。
getPath()	返回与射线相交物体的路径。

例如：

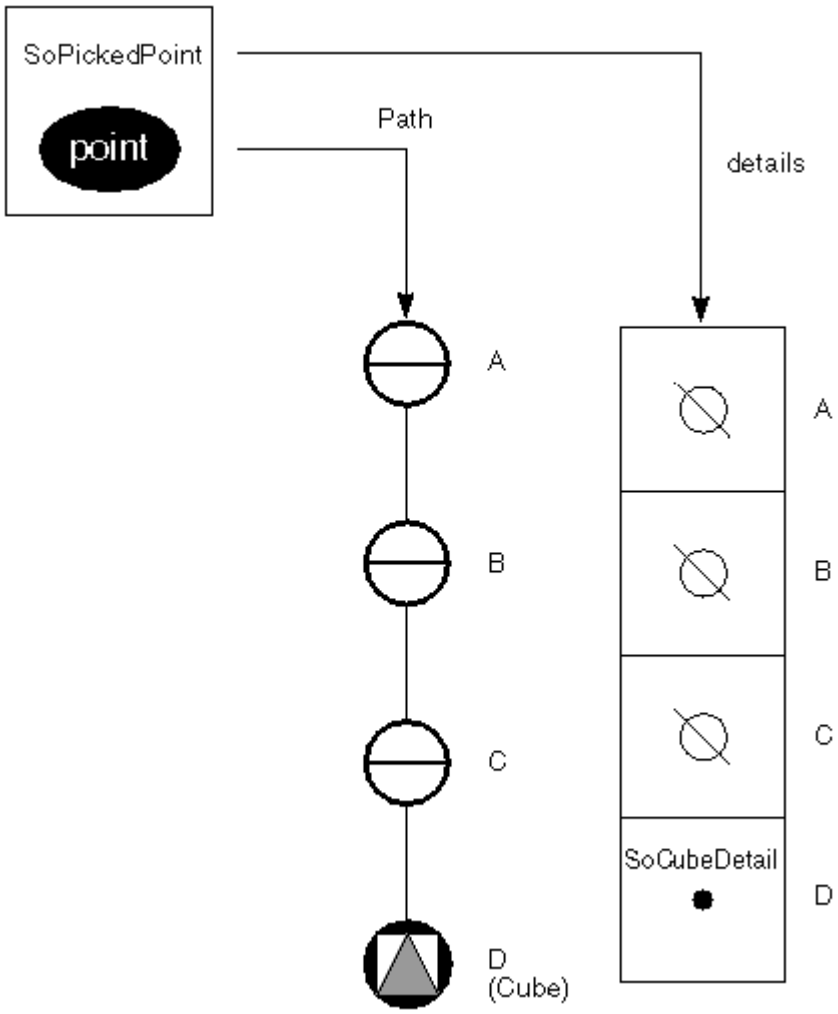
```
SoPath *pathToPickedObject;
```

```
const SoPickedPoint *myPickedPoint =
         myPickAction.getPickedPoint();
if (myPickedPoint != NULL)
   pathToPickedObject = myPickedPoint->getPath();
```

图 9-7 展示了一个由**SoRayPickAction**返回的路径图（通过调用**SoPickedPoint**的**getPath()**方法获得此路径）。这个路径包含着一个指向路径中每个节点的指针（This path contains a pointer to each node in the path to the picked object.）。使用**SoPickedPoint**的下列方法可以得到关于在某个特定节点物体的坐标空间内的拾取信息。我们可以传入一个感兴趣的节点指针作为参数，或直接使用缺省值参数（NULL）。如果使用缺省参数的话，表示要得到路径中结尾处节点的拾取信息。

getObjectPoint()	返回拾取射线与物体的交点。交点定义在物体坐标空间内。
getObjectNormal()	返回交点处的表面法向向量。定义在物体坐标空间内。
getObjectTextureCoords()	返回交点处的纹理坐标。定义在物体坐标空间内。

图 9-7 拾取点路径和细节列表



使用 SoDetail

在拾取路径中所包含的每个节点都可能会关联着一个**SoDetail**类信息。在**SoDetail**类中保存着有关拾取的附加信息。对于有些类节点来说，所关联的**SoDetail**为NULL。（SoDetail只针对形体节点。对于非形体节点，一般没有相关联的SoDetail，例如SoSeparator节点就没有相关联的SoDetail，所以在路径中就是NULL，在图 9-7 中，A、B、C节点都属于这种情况。译者注）。表 9-2 中给出了**SoDetail**子类所保存的信息。

图 9-8 细节类

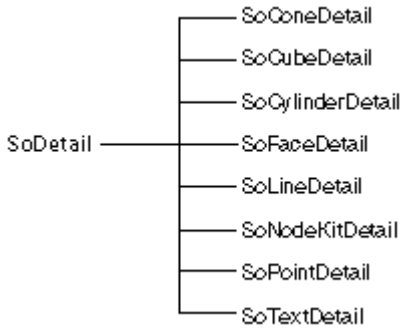


表 9-2 存储 **SoDetail** 的类

类名	细节类型	提供的信息
SoCone	SoConeDetail	包含圆锥体上那个部分被点击的信息
SoCube	SoCubeDetail	包含立方体那个面被点击的信息
SoCylinder	SoCylinderDetail	包含圆柱体上那个部分被点击的信息
SoText2, SoText3	SoTextDetail	包含点击到字符串中那个字符上、字符文本的那个部分、以及被点击字符的包围盒等信息。
SoFaceSet; all vertex-based shapes except lines, points, and NURBS	SoFaceDetail	指示点击到形体中那个面上。
SoLineSet, SoIndexedLineSet	SoLineDetail	指示点击到“线集”中那条线上
SoPointSet	SoPointDetail	指示点击到“点集”中那个点上

使用**SoPickedPoint**类的**getDetail()**方法可以得到在拾取路径中给定节点的细节信息。这个方法需要一个节点指针作为参数，其中指针所指向的必须是位于拾取路径中的节点。如果参数指针为NULL，或不指定参数，则这个函数将返回路径结尾处的节点细节信息。例如，先判断击到的物体是否是圆柱，如果是，则再判断是否点击到圆柱的顶面上。代码如下：

```
const SoDetail *pickDetail = myPickedPoint->getDetail();
if (pickDetail != NULL && pickDetail->getTypeId() ==
                        SoCylinderDetail::getClassTypeId()) {
   // Picked object is a cylinder
   SoCylinderDetail *cylDetail =
            (SoCylinderDetail *) pickDetail;

   // See if top of the cylinder was hit
   if (cylDetail->getPart() == SoCylinder::TOP) {
      printf("Top of cylinder was hit\n");
   }
}
```

下面的代码将演示，对于基于面的形体（face-based shape），我们如何通过使用**SoFaceDetail**类，找到与点击位置距离最近的顶点数据。**SoFaceDetail**类中包含着一个**SoPointDetails**数组。通过调用**SoPointDetail**类中**getCoordinateIndex()**函数，我们可以找到与点击位置距离最近的顶点坐标值。例子中寻找节点的代码留给读者自己完成（创建一个搜索动作，将搜索动作应用到拾取路径上，然后要求只查找路径中最后一个**SoCoordinate3**节点。读者还需要了解一些有关被搜索场景的结构信息，例如是否包含有重置标记（Override flags）或忽略标记（Ignore flags），因为这两种标记可能会影响到搜索的结果）。

```
//   This function finds the closest vertex to an intersection
//   point on a shape made of faces, passed in the
//   "pickedPoint" argument. It returns the SoCoordinate3 node
//   containing the vertex's coordinates in the "coordNode"
//   argument and the index of the vertex in that node in the
//   "closestIndex" argument. If the shape is not made of faces
//   or there were any other problems, this returns FALSE.

static SbBool
findClosestVertex(const SoPickedPoint *pickedPoint,
            SoCoordinate3 *&coordNode, int &closestIndex)
{
   const SoDetail *pickDetail = pickedPoint->getDetail();

   if (pickDetail != NULL && pickDetail->getTypeId() ==
                           SoFaceDetail::getClassTypeId()) {
      // Picked object is made of faces
      SoFaceDetail *faceDetail = (SoFaceDetail *) pickDetail;

      // Find the coordinate node that is used for the faces.
      // Assume that it's the last SoCoordinate3 node traversed
```

```
      // before the picked shape.
      SoSearchAction  mySearchAction;
      mySearchAction.setType(SoCoordinate3::getClassTypeId());
      mySearchAction.setInterest(SoSearchAction::LAST);
      mySearchAction.apply(pickedPoint->getPath());

      if (mySearchAction.getPath() != NULL) {  // We found one
         coordNode = (SoCoordinate3 *)
                     mySearchAction.getPath()->getTail();

         // Get the intersection point in the object space
         // of the picked shape
         SbVec3f objIntersect = pickedPoint->getObjectPoint();

         // See which of the points of the face is the closest
         // to the intersection point
         float minDistance = 1e12;
         closestIndex = -1;
         for (int i = 0; i < faceDetail->getNumPoints(); i++) {
            int pointIndex =
                  faceDetail->getPoint(i)->getCoordinateIndex();
            float curDistance = (coordNode->point[pointIndex] -
                  objIntersect).length();
            if (curDistance < minDistance) {
               closestIndex = pointIndex;
               minDistance = curDistance;
            }
         }

         if (closestIndex >= 0)
            return TRUE;
      }
   }

   return FALSE;
}
```

使用拾取动作

例 9-4 的代码将演示如何建立拾取动作，以及如何将拾取物体的路径输出到**stdout**控制台中。

```
SbBool
writePickedPath (SoNode *root,
   const SbViewportRegion &viewport,
   const SbVec2s &cursorPosition)
{
   SoRayPickAction myPickAction(viewport);

   // Set an 8-pixel wide region around the pixel
   myPickAction.setPoint(cursorPosition);
   myPickAction.setRadius(8.0);

   // Start a pick traversal
   myPickAction.apply(root);
   const SoPickedPoint *myPickedPoint =
            myPickAction.getPickedPoint();
   if (myPickedPoint == NULL)
      return FALSE;         // no object was picked

   // Write out the path to the picked object
   SoWriteAction myWriteAction;
   myWriteAction.apply(myPickedPoint->getPath());

   return TRUE;
}
```

回调程序（Calling Back to the Application）

SoCallbackAction允许我们自己遍历整个场景并累积遍历状态。它包含有一些方法，可以在遍历场景的过程中，只要遍历到一种指定类型的节点，就会回调应用程序提供的函数。对于每个当前遍历的节点，在回调函数中都可以访问整个Inventor遍历状态数据。因此，回调函数可以查询遍历状态中像当前坐标、当前向量、当前材质绑定等任意的元素数据。*Open Inventor C++ Reference Manual*书中详细讨论了**SoCallbackAction**所有的状态查询函数。

SoCallbackAction还允许我们注册一种只要遍历到某些形体节点，就会被调用的回调函数。用于绘制物体的图元信息（primitives）会被传递进回调函数中，以供应用程序分析使用。

这个动作提供了一种便捷的机制，使我们可以不必通过子类化的方法，就可以非常简单地向 Inventor 增加上自己的动作（在 *The Inventor Toolmaker* 书中，介绍了如何使用子类化的方法创建一种新的动作）。这种机制对需要为场景遍历增加新功能的 C 程序员来说是非常有用的。

创建回调动作实例

下面的代码是创建**SoCallbackAction**的例子代码：

```
SoCallbackAction cbAction;
```

注册回调函数

Inventor 提供了很多种设置节点回调函数的方法。每种方法都需要一个节点类型，一个用户回调函数的指针，一个用户自定义数据的指针作为参数。在遍历场景的过程中，如果遍历到了一个指定类型或派生类型的节点时，Inventor 就会调用用户注册的回调函数。

通用回调函数

下列的函数可以对任意类型的节点注册回调函数：

addPreCallback() 增加一个回调函数。在一个特定类型的节点被遍历前，将调用此回调函数。

addPostCallback() 增加一个回调函数。在一个特定类型的节点被遍历后，将调用此回调函数。

addPreTailCallback() 增加一个回调函数。仅在路径中最后那个节点被遍历之前，将调用此回调函数。

addPostTailCallback() 增加一个回调函数。仅在路径中最后那个节点被遍历之后，将调用此回调函数。

当遍历到一个SoSeparator节点时，在遍历其子节点之前，将调用由**addPreCallback()**注册的回调函数。当遍历完所有的子节点，并且还没有恢复原来的遍历状态之前，将调用由**addPostCallback()**注册的回调函数。一般我们只有在对路径使用回调动作的时候，才会使用**addPreTailCallback()**和**addPostTailCallback()**方法。

通用回调函数必须返回下列三种数值：

SoCallbackAction::CONTINUE 继续遍历场景

SoCallbackAction::PRUNE 不再遍历当前子场景，但继续遍历子场景上面和右面的其余场景。

SoCallbackAction::ABORT 停止遍历场景。将遍历状态恢复到根节点的状态。

产生图元（Primitive Generation）

下列回调函数只针对某种特定类型的*形体*节点。当设置完这些回调函数后，并且遍历到了一个形体节点时，Inventor会自动生成这个形体节点的图元数据，之后调用回调函数，将图元数据传递到回调函数中。例如，假设我们正在写一个自己的渲染器，我们希望在物体的三角面上都显示有花纹状的图案，这时我们可以使用**addTriangleCallback()**函数来实现这种功能。

addTriangleCallback()	对可以产生三角面的节点（如 SoFaceSet、SoNurbsSurface 节点）增加回调函数。
addLineSegmentCallback()	对可以产生线段的节点（如 SoLineSet、SoIndexedLineSet 节点）增加回调函数。（对于 SoFaceSet 或类似的节点，即使绘制风格为 LINES，这个函数也不会对它们生成线段图元）
addPointCallback()	对可以产生点的节点（如 SoPointSet 节点）增加回调函数。（对于 SoFaceSet 或 SoLineSet 节点，即使绘制风格为 POINTS，这个函数也不会对它们生成点图元信息）

三角面回调函数的定义形式如下：

```
void SoTriangleCB(void *userData,
                  SoCallbackAction *action,
                  const SoPrimitiveVertex *v1,
                  const SoPrimitiveVertex *v2,
                  const SoPrimitiveVertex *v3);
```

Inventor 会为组成形体的每个三角面调用一次三角面回调函数。举一个使用这种回调函数的一个例子，假设我们正在写一个光线跟踪的程序，我们希望对所有的多边形物体只按照一种类型的数据结构来处理。三角面回调函数可以注册到球体、圆锥、圆柱、NURBS 曲面、面集、和四边形网格物体上。

SoPrimitiveVertex类表示一个几何图元（三角面，线段，点）的顶点，通常都是由回调动作产生的。它包含有一个在物体坐标空间内的顶点坐标、法向向量、纹理坐标、材质索引、和一个指向**SoDetail**类的指针。**SoDetail**类中也许还包含这个顶点的其它附加的信息。

提示：在回调函数中，如果我们想知道三角面是以“点”风格显示，还是以“线型”风格显示的话，可以使用遍历状态中的绘制风格元素的数值来判断，例如：

```
if(SoDrawStyleElement::get(动作->getState())==
SoDrawStyleElement::LINES)
```

The Inventor Toolmaker 书中介绍了元素的更多内容。

应用动作

SoCallbackAction可以使用在一个节点上、一个路径上、或一个路径列表上。

使用产生图元数据的回调

例 9-5 将演示如何使用回调动作，将一个球体分解成一组三角面图元数据。

例 9-5 使用三角面回调函数

```
...
SoSphere *mySphere = new SoSphere;
mySphere->ref();
printSpheres(mySphere);
...
void
printSpheres(SoNode *root)
{
   SoCallbackAction myAction;

   myAction.addPreCallback(SoSphere::getClassTypeId(),
            printHeaderCallback, NULL);
   myAction.addTriangleCallback(SoSphere::getClassTypeId(),
            printTriangleCallback, NULL);

   myAction.apply(root);
}

SoCallbackAction::Response
printHeaderCallback(void *, SoCallbackAction *,
      const SoNode *node)
{
   printf("\n Sphere ");
   // Print the node name (if it exists) and address
   if (! !node->getName())
      printf("named \"%s\" ", node->getName());
   printf("at address %#x\n", node);

   return SoCallbackAction::CONTINUE;
}

void
printTriangleCallback(void *, SoCallbackAction *,
   const SoPrimitiveVertex *vertex1,
   const SoPrimitiveVertex *vertex2,
   const SoPrimitiveVertex *vertex3)
{
   printf("Triangle:\n");
   printVertex(vertex1);
   printVertex(vertex2);
   printVertex(vertex3);
}
```

```
void
printVertex(const SoPrimitiveVertex *vertex)
{
   const SbVec3f &point = vertex->getPoint();
   printf("\tCoords     = (%g, %g, %g)\n",
               point[0], point[1], point[2]);

   const SbVec3f &normal = vertex->getNormal();
   printf("\tNormal     = (%g, %g, %g)\n",
               normal[0], normal[1], normal[2]);
}
```

第十章 处理事件和选择器（Selection）

本章目标

在阅读完本章节后，我们将有能力做下列事情：

- 解释 Open Inventor 如何处理输入事件。
- 使用 Inventor 内建的选择策略，选择场景中的物体。
- 创建事件回调节点，执行自定义选择策略。
- 高亮选择（Highlight selected）场景中的物体。
- 编写选择回调函数，当选择器列表发生变化后，允许应用程序执行特定的操作。

本章将描述Open Inventor的事件模型。Inventor的事件模型为我们提供了一种简单的机制，通过这些机制，我们可以将像“按下键盘”和“移动鼠标”等这样的事件传递给场景数据库中的物体，以便这些物体可以处理这些事件。与窗口系统将事件传递到客户窗口中的做法类似，Inventor会将事件传递到数据库中能够处理它们的物体上。本章将要介绍的重要概念包括：Inventor的事件处理编程模型，以及**SoXtRenderArea**的用法，**SoXtRenderArea**是一个可以执行渲染操作和事件处理的窗口组件。本章还将详细讨论**SoHandleEventAction**类。同时也会对*事件回调函数*，*选择器节点*和*高亮选择*等这些概念进行探讨。

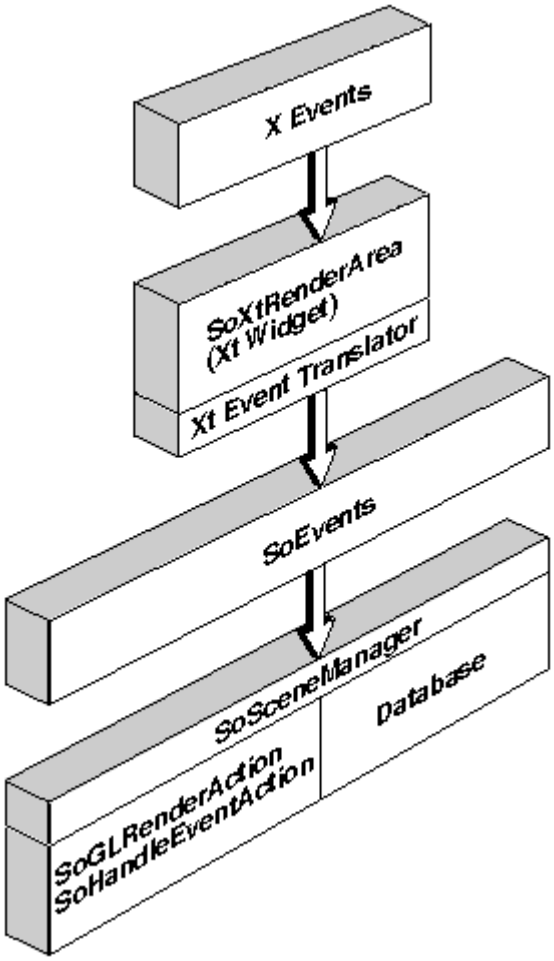
概述

当用户在手柄盒(handle-box)操作器上点击鼠标，然后将物体拖动到一个新位置上时，Inventor 是如何接收来自鼠标设备的输入，从而移动物体的呢？如果用户在一个不包含有任何物体的渲染区上点击鼠标会发生什么事情呢？Inventor 是如何跟踪多个被用户选中的物体呢？ 这些都是在我们开发交互式应用程序之前必须要回答的问题。

事件处理的通用编程模型

Inventor 包含有一个为场景数据库内建的事件模型。这个模型不依赖任何特定的窗口系统或软件包。当开发一个 Inventor 程序时，我们可以选择 X Window 编程模型，使用 X Window 系统提供的功能创建渲染窗口，向 Inventor 传递事件（**X Window 是一个基于 UNIX 环境下的视窗系统。译者注**）。Inventor 提供了将 X 事件转换到 Inventor 事件的转换类。图 10-1 展示了 X 事件是如何传递到渲染窗口中，然后又如何转换成可以被 Inventor 场景管理器处理的 Inventor 事件。因为 Inventor 独立于任何的窗口系统，所以我们可以选择其它不同的窗口系统，开发自己的事件转换类。（**Inventor 已经对 Microsoft Windows 窗口系统提供了支持类 SoWin，所以我们不必再针对 Microsoft Windows 开发事件转换类了。译者注**）

图 10-1 Inventor 的事件处理流程



使用 X Window 系统

Inventor 提供了一组使用 X Window 系统的工具包。工具包中包含有下列工具：

- 渲染区“窗口组件（widget）”
- 消息主循环和初始化函数。
- 事件转换工具

除了这些工具之外，Inventor 组件库还包含一组 Xt 组件。在这些组件中，包括了带有用户界面且可以直接修改场景数据库的观察器（viewers）和编辑器（editors）。

本章主要关注于 Inventor 独立于窗口系统方面的特性：

- 事件（派生自**SoEvent**类）
- 场景管理器
- 处理事件动作
- 事件回调函数
- 选择器节点

第 16 章将更详细地描述Inventor组件库的具体用法。如果想使用Inventor的观察器和编辑器。我们必须要使用Xt的渲染区（**SoXtRenderArea**）和X Window系统模型。如果不使用这些观察器和编辑器，我们可以选择另外的窗口系统，然后实现自己的渲染区、事件消息循环、以及事件转换。

渲染区

渲染区窗口组件提供了一个很方便的对象，通过它可以很容易地创建一个窗口，并将特定的窗口事件转换成通用类型的Inventor事件。通过使用X Window系统模型，我们可以创建一个**SoXtRenderArea**渲染区（图 10-1）。当特定的窗口事件传递进渲染区后，渲染区会自动将这些事件转换成通用的Inventor **SoEvent**事件类。

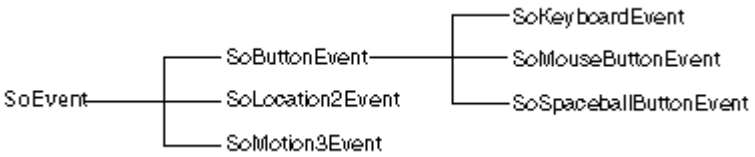
渲染区提供下列功能：

- 内建了场景数据库传感器（sensors）。当场景发生改变、渲染区窗口改变大小、或窗口由隐藏变成显示的时候，传感器会通知 Inventor 重新刷新窗口。
- 内建的事件处理功能
- 若干种属性设置功能。例如可以设置场景的透明类型和反走样程度。

Inventor 事件（SoEvent）

图 10-2 展示了**SoEvent**的类树图

图 10-2 事件类



每个**SoEvent**类实例中包含了下列信息：

- 类型标志（**SoType**）
- 事件发生的时间
- 事件发生时光标的位置
- 事件发生时辅助键（Shift、Control、Alt）的状态

SoEvent的派生类中包含有更多的信息。例如，**SoButtonEvent**类中包含有事件发生时，按键是处于按下状态还是抬起状态的信息。**SoMouseButtonEvent**类包含有那个按键被按下的信息（按键 1、2、或 3）。

*Open Inventor C++ Reference Manual*书中列有**SoEvent**类的所有可用方法。另外，我们还可以使用下面的宏定义：

- SO_MOUSE_PRESS_EVENT() — 传入一个**SoEvent**实例和按键序号，如果按键是按下

的状态，这个宏返回TRUE.

- SO_MOUSE_RELEASE_EVENT()－传入一个**SoEvent**实例和按键序号，如果按键是抬起的状态，这个宏返回TRUE.

SoKeyboardEvent类包含有那个按键被按下的信息。（但并没有指明是大写字母键还是小写字母键）

提示：对一个X Window按键事件来说，使用**SoKeyboardEvent::getKey()**方法与使用XLookupKeysym()，两者的功能是相同的。

无论何时，只要移动光标，系统就会产生一个**SoLocation2Event**事件。在这个事件中包含有光标在窗口坐标系下的绝对位置（窗口坐标系以窗口的左下角为（0,0）坐标原点）。当一个 3D输入设备，例如空间轨迹球（spaceball），移动时，将会产生一个**SoMotion3Event**事件。在这个事件中包含有相对设备前一次位置的旋转和平移变换矩阵。

提示：Inventor的事件模型是可扩展的。如果我们的设备没有对应的**SoEvent**类存在，我们可用创建自己的事件类。（见*The Inventor Toolmaker* 第 11 章）。

场景管理器

如图 10-1 所示，**SoSceneManager**是一个公共类，它将依赖于特定窗口系统的渲染区（如**SoXtRenderArea**）和Inventor联系在一起。渲染区利用场景管理器来操作图形场景。场景管理器将同时管理渲染和事件处理这两种任务，而且场景管理器是不依赖于任何特定的窗口系统。

Inventor 事件处理

在 Inventor 内部，事件将被分配到场景数据库内的 3D 事件处理对象上。像操作器（Manipulator）和拖拽器（Dragger）对象（第 15 章中将做详细讨论）就是 Inventor 场景中可以处理事件的 3D 对象。形体对象（像球体、圆柱体、四边形网格）、属性对象（像材质和绘制风格）、几何变换对象、灯光对象、照相机对象等这些对象都不会处理事件。在“节点如何处理事件：SoHandleEventAction”的章节中将会讨论所有可以处理事件的节点。

使用 Inventor，我们可以从下列的四种事件处理机制中选择其一：

1. 我们可以使用 Inventor 场景管理器提供的自动事件处理机制。在这种机制中，有一些节点可以处理事件（见“节点如何处理事件：SoHandleEventAction”）。这种

机制也许是使用起来最简单的机制。（注意：我们也可以创建自己的节点来处理事件。对于某些 Inventor 不支持的事件来说，如果在不同的程序中需要处理这些事件，或者其它程序员也需要处理这些事件时，最好的方式就是我们创建一个自己的节点来专门处理这些事件。我们还可以创建新的操作器来处理事件。在 *The Inventor Toolmaker* 书中有更多有关创建新节点和操作器的内容）。

2. 我们可以使用 Inventor 的事件回调机制。在这种机制中，可以使用用户自定义的回调节点来处理事件（见“使用事件回调节点”章节）。这种方法是以每个对象为基础来处理事件的，相对来说还算比较容易实现。它的缺点是：虽然回调节点可以被输出到一个文件中，但因为回调节点中没有域值来描述正在监控那条路径或对那种事件感兴趣，所以输出到文件中的回调节点是没有作用的。

3. 我们可以“绕过”Inventor 的事件处理机制，直接将事件传递到应用程序中（见“直接发送事件到应用程序中”）。如果我们更喜欢自己直接处理 X Window 事件，并且不需要以每个对象为基础来处理事件时，可以使用这种方法。这种方法会绕过场景遍历过程，且只能处理窗口事件。

4. 我们可以使用Inventor的通用回调机制。在这种机制中，我们编写的回调节点可以处理所有的动作（见第 17 章 使用**SoCallback**节点的例子）。当我们在处理事件时，还需要执行其它的动作时（例如渲染），这时就可以使用这种机制。如果我们只是处理事件的话，最好使用第 2 种方法（事件回调节点），因为它可以为我们做更多的事情。

我们建议用户使用第 1、2 和 4 种方法，因为这些方法都是与窗口系统无关，具有跨平台的能力，因此会更容易移植和扩展。相对来说，方法 1 和 2 比较容易实现。

节点如何处理事件：SoHandleEventAction

（*高级内容*）

Inventor 通过使用一些“智能”节点，提供了一种自动处理事件的机制。这种机制概述如下：

1. 渲染区对其窗口系统里感兴趣的特定事件进行注册登记。
2. 渲染区从它的窗口系统中接收到一个事件（**SoXtRenderArea**接收X事件）
3. 渲染区将事件转换成一个**SoEvent**。
4. **SoEvent**被发送到场景管理器中，场景管理器将创建一个**SoHandleEventAction**实例。
5. （*高级内容*）
   处理事件的动作将被应用到场景的顶层节点上。这个动作将遍历整个场景。如前面章节中所描述的那样，被遍历到的节点将执行自己相对应的动作行为。当遍历到一个可以处理这个事件的节点时（通常是操作器），**SoHandleEventAction**就会停止遍历，并把事件交给那个节点来处理。

接下面将讨论不同类型的节点将如何执行**SoHandleEventAction**.

SoNode

SoNode是所有节点的基类。它对**SoHandleEventAction**不做任何事情。因此，所有的属性节点、变换节点、几何体节点、照相机节点和灯光节点也都不对**SoHandleEventAction**做任何事情。

SoGroup

当处理事件动作应用在SoGroup上时，组节点将按照从左到右的顺序遍历它的子节点，请求每个子节点来处理事件。在每个子节点处理完事件后，组节点都要查看事件是否已经被处理过了。如果事件已被处理，那么处理事件动作就会停止遍历场景。如果事件还没有被处理，那么**SoGroup**就会继续请求下个子节点来处理这个事件。

SoGroup使用**SoHandleEventAction**类的**isHandled()**方法来查询事件是否已经被处理过了。当一个节点对事件做了处理，那么这个节点就会调用**setHandled()**方法设置上事件已被处理的标志。

SoManipulator

操作器是场景中一种可以处理某些事件的“智能”对象（第 15 章将做更详细地讨论）。通常，一个操作器会将场景中的某个节点替换成相应的可编辑版本的节点（a manipulator replaces a node in the scene graph with an editable version of that node）。通过利用**SoHandleBoxManip**将场景中能对物体产生影响的几何变换节点替换成手柄盒操作器，我们就可以修改物体的大小和位置。通过利用**SoTrackballManip**将场景中适当的几何变换节点替换成轨迹球操作器，我们就可以围绕一个中心点来旋转物体。**SoTrackballManip**操作器通过响应用户移动鼠标来修改自身的旋转域值。

像手柄盒和轨迹球这样的操作器，除了需要事件类型外，还需要一些拾取信息。这些操作器通过调用**SoHandleEventAction**类的**getPickedPoint()**函数，查看那个物体被拾取了。如果一个操作器被拾取了，这个操作器就会处理这个事件。我们也可以创建一种另类的操作器，这种操作器不需要在它的几何形体上点击就可以激活它（you can also create manipulators that do not require a hit on their geometry to be activated.）（见*The Inventor Toolmaker* 书中，第 8 章 有关创建用户自定义的操作器）。

拦截事件（Grabbing）

节点可以请求所有后来的事件都直接发送到它那里，直到节点以后再另行通知取消这种请求。这种请求称为*拦截事件（grabbing）*。例如，当操作器接收到一个鼠标左键按下的事件后，操作器就会拦截所有后来的事件，直到接收到一个鼠标左键抬起的事件为止。操作器使用一个指向自身的指针（this）作为参数，调用事件处理动作的**setGrabber()**方法。

```
handleEA->setGrabber(this);
```

在执行完上面的代码后，事件处理动作会将动作直接应用到拦截事件的节点上，而不再像往常那样将动作应用到场景根节点上。操作器可以使用**releaseGrabber()**方法停止拦截事

件。

```
handleEA->releaseGrabber();
```

如果节点对一个事件调用了**releaseGrabber()**，但这个节点却没有处理那个事件的话，事件处理动作将会针对根节点重新初始化一次遍历，将那个事件传递给场景中所有的节点上。例如，操作器在一个鼠标按下事件发生之后，开始拦截事件。然而，如果在鼠标还没有移动前，用户就释放了鼠标按键，这时，操作器会停止拦截事件，并且操作器也没有对鼠标按下事件进行处理。在这种情况下，鼠标按下事件将被传递到场景中再做相应的处理。

getGrabber()方法将返回当前正在拦截事件的节点。（见*Open Inventor C++ Reference Manual*书中**SoHandleEventAction**）。

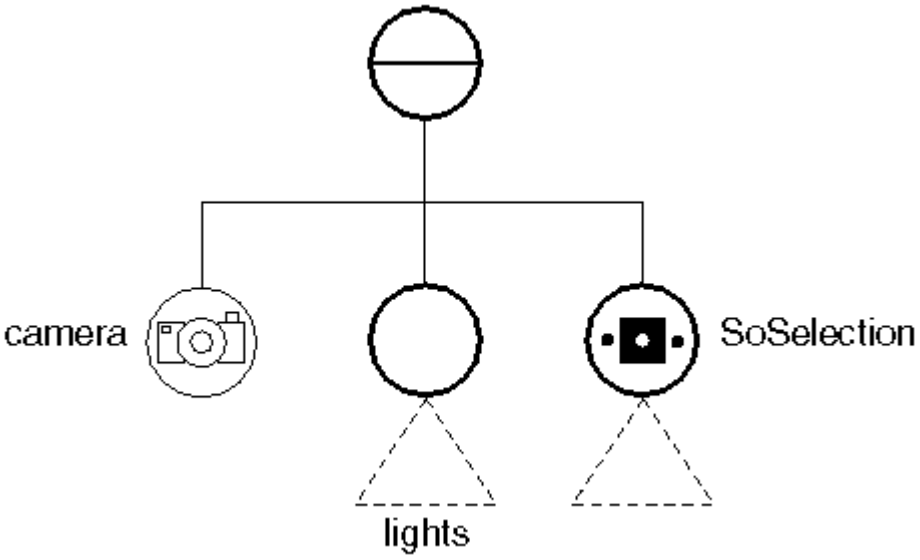
注意：在场景中拦截事件不能拦截到 X Window 服务器端的事件。（Grabbing events in the scene graph does not perform an X server grab.）

（setGrabber 和 releaseGrabber 这两个函数的功能与 Microsoft Windows API 的 SetCapture 和 ReleaseCapture 的功能非常类似，我们可以对比理解它们。译者注）

SoSelection

SoSelection节点是从**SoGroup**派生出来的，通常都是插到接近场景顶层结构的位置处（如图 10-3 所示）。当处理事件动作应用到**SoSelection**节点上时，**SoSelection**节点按照**SoGroup**同样的方式遍历自己的子节点。然而唯一不同的地方是，如果其子节点没有一个对事件进行了处理，那么选择器（selection）节点将自己处理这个事件。

图 10-3 插入一个 **SoSelection** 节点



拾取（Picking）

当发生一个鼠标左键按下的事件后，**SoSelection**对象需要知道用户到底点击了那个节点。**SoSelection**对象会通过调用处理事件动作的**getPickedPoint()**方法来查询鼠标点击的位置信息（**SoSelection**会在鼠标按下与鼠标抬起这两个事件中检查被选取的物体，这样做的目的是为了保证这两个鼠标事件是发生在同一个物体上的。调用**setPickMatching()**方法并且设置输入参数为FALSE，可以禁止这个特性。这时**SoSelection**将只在鼠标抬起事件中查询拾取点）。

场景中的节点只有在第一次被拾取的时候，处理事件动作才会去查询点击物体的相关信息（也就是说只有在必要的时候，才会执行拾取动作）。**SoHandleEventAction**会将所查询到的拾取信息放入自己的缓存中，以便以后遍历节点时，如遇到像操作器和选择器这些节点，就可以很快地查询到这些信息。在**SoHandleEventAction**遍历的过程中，通常只执行一次拾取动作。

选择规则（**Selection Policy**）

当接收到拾取信息后，**SoSelection**类会执行适当的选择规则。当前我们对**SoSelection**的**policy**域有三种规则可供选择。**SoSelection**会为我们记录一个选择列表。被选择的物体可以被渲染区高亮显示（见“高亮显示被选物体”）。缺省的选择规则是**SoSelection::SHIFT**。规则如下：

- 如果用户在一个物体节点上点击鼠标左键，**SoSelection**会首先清空选择列表，然后将这个物体增加到选择列表中。
- 如果用户在一个物体节点上点击鼠标左键的同时还按下了Shift键，**SoSelection**会切换这个物体的选择状态(即如果物体当前位于选择列表中，则将它从列表中删除。如果物体当前没有被选中，则将它增加到选择列表中）
- 如果用户没有点击到任何物体上时，**SoSelection**会清空选择列表。

SoSelection::SINGLE规则表示在同一时间内只能选择一个物体。规则如下：

- 如果用户在一个物体节点上点击鼠标左键，**SoSelection**会首先清空选择列表，然后将这个物体增加到选择列表中。
- 如果用户没有点击到任何物体上时，**SoSelection**会清空选择列表。

SoSelection::TOGGLE规则表示在同一时间内可以选择多个物体。规则如下：

- 如果用户在一个物体节点上点击鼠标左键，**SoSelection**会切换那个物体的选择状态（如果物体以前没有被选中，则将它增加到选择列表中，而如果物体以前被选中了，则将它从选择列表中删除）。
- 如果用户没有点击到任何物体上时，**SoSelection**不做任何事情，选择列表不发生改变。

提示：当 Shift 键没有按下时，Single 选择规则与 Shift 选择规则是相同的。当 Shift 键按下时，Toggle 选择规则与 Shift 选择规则是相同的。

如果上面这些规则没有一个能符合我们要求，我们还可以实现我们自己定制的选择规则。首先我们要创建一个事件回调节点，并向这个节点传递一个指向**SoSelection**节点的指针。然后我们可以（在事件回调函数中）调用**SoSelection**节点的select()、deselect()，toggle()、deselectAll()函数来实现我们自己新的选择规则。还有一种定制选择规则的办法，就是从**SoSelection**类派生出我们自己的类，在派生类中修改基类的选择规则。关于选择列表的更多内容请见“选择器”。

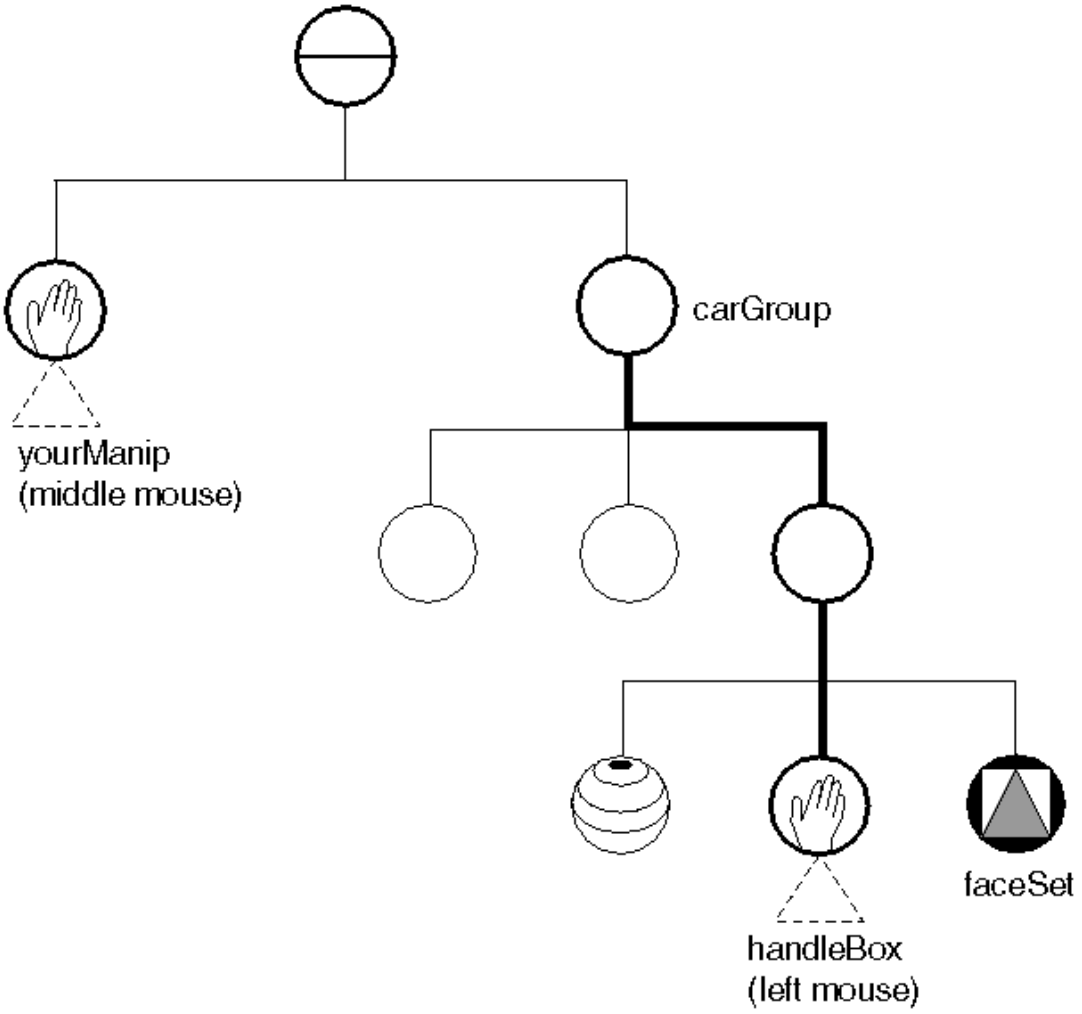
确定事件处理者（Finding the Event Handler）

下面的例子将举例说明对一个给定的事件寻找其处理者的过程。假设我们建立了一个场景，场景中包含有若干个物体，其中有一个手柄盒操作器，这个操作器可以影响一个索引面集物体（indexed face set）的几何变换矩阵。场景中还包含有一个 *yourManip*，这是我们自己开发的用于处理鼠标中键事件的操作器。场景图见图 10-4

如果我们在手柄盒所包围的面集物体（face-set object）上点击了鼠标左键，场景管理器将首先接收到这个事件，然后它调用 **setEvent()** 方法将事件发送给 **SoHandleEventAction**。下面是图 10-4 中的节点在处理事件动作遍历的过程中，如何做出相应处理的过程：

1. **SoSeparator**节点按照从左到右的顺序，请求它的子节点来处理鼠标左键按下事件。
2. *yourManip* 节点不会处理这个鼠标左键按下的事件，因为它只处理鼠标中键事件。

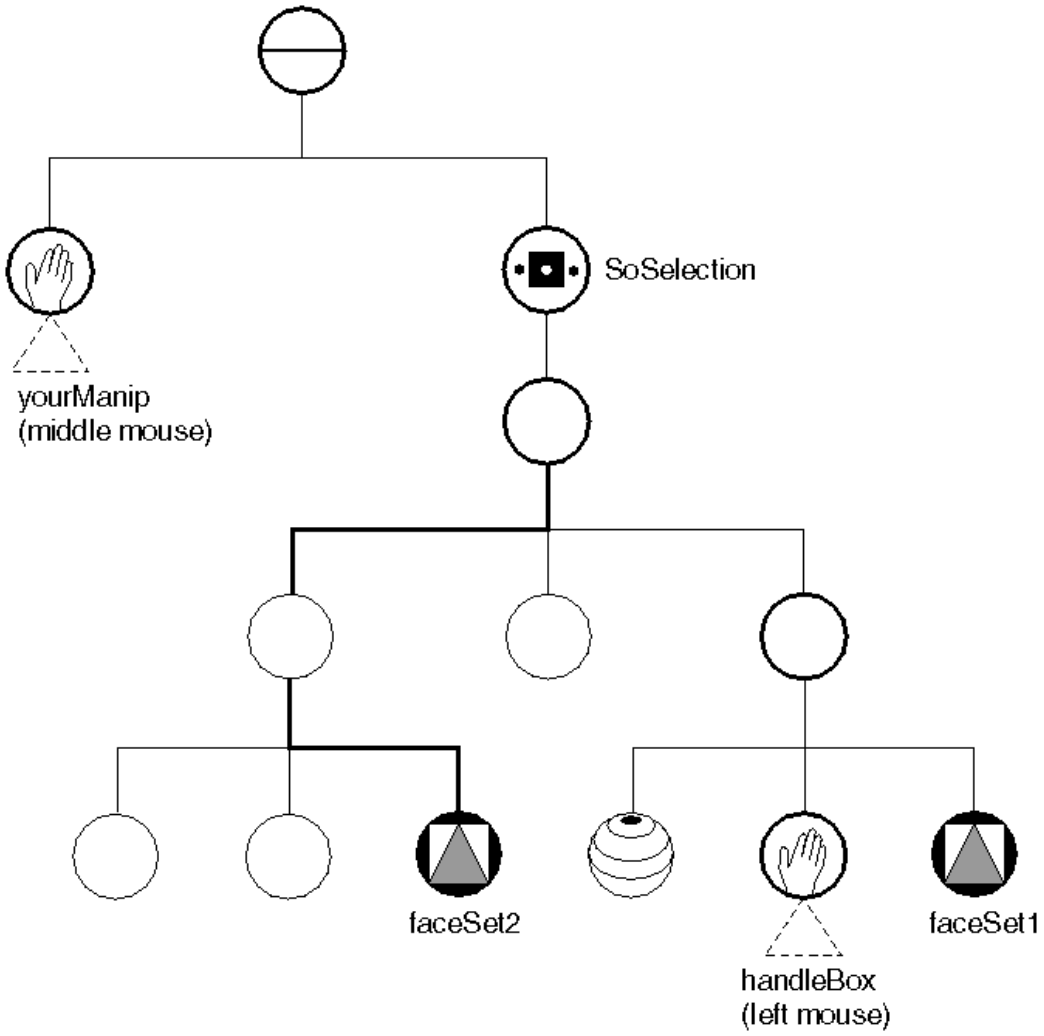
图 10-4 带有操作器的场景



3. *carGroup* 组节点按照从左到右的顺序遍历它的子节点，并请求这些子节点来处理鼠标左键按下事件。第三个子节点也是一个组节点，所以这个子节点将会继续按照从左到右的顺序遍历它的子节点。
4. 当遍历到手柄盒操作器节点时，手柄盒操作器可以处理鼠标左键事件，它首先需要知道自己是否位于拾取路径中（即需要先确定鼠标是否点击到自己，译者注）。手柄盒操作器可以通过调用**SoHandleEventAction**的**getPickedPoint()**方法来确定自己是否被点击了。因为它当前的确是被点击了，所以手柄盒操作器就会调用**setHandled()**，然后处理这个事件。

在图 10-5 中，场景新增加了一个面集物体（*faceSet2*），假设用户点击了这个物体（而不是像上面那样点击了手柄盒包围的*faceSet1*），因为手柄盒操作器没有包含在拾取路径中，并且不会对拾取路径产生影响，所以手柄盒操作器不会处理这个事件。在这种情况，**SoSelection**对象将处理这个事件。

图 10-5 拾取一个物体节点



使用事件回调节点（*高级内容*）

如果我们需要一个 Inventor 没有提供的事件处理方法，那么我们就需要创建自己的操作器，或利用事件回调节点来开发自己的事件处理器。在 *The Inventor Toolmaker* 书的第 8 章中讨论了如何创建新类型的操作器。本节将讨论使用事件回调节点的方法。

一个事件回调节点中包含有一个用户自定义的函数，只要发生了指定类型的事件、拾取了指定的路径、或处理事件动作遍历到了这个事件回调节点，自定义函数都会被调用。如果没有指定路径（也就是说输入参数为NULL），Inventor会在发生特定类型的事件或处理事件动作遍历到事件回调节点的时候自动调用事件回调函数。我们可以定义多个事件回调函数，并将它们添加到**SoEventCallback**节点维护的回调函数列表中。

调用**addEventCallback()**方法，可以设置感兴趣的事件类型和回调函数。

```
SoEventCallback *eventCB = new SoEventCallback;
eventCB->addEventCallback(SoKeyboardEvent::getClassTypeId(),
```

```
        myCallbackFunc, userData);
```

使用**setPath()**方法，可以设置需要监视那条路径。

当回调函数被调用时，Inventor会将用户自定义数据和一个指向**SoEventCallback**实例的指针作为参数传入到回调函数中。使用**removeEventCallback()**方法，可以将一个回调函数从事件回调函数列表中删除掉。

提示：如果我们希望在发生任何类型的事件的时候，回调函数都会被调用。应该使用**SoEvent::getClassTypeId()**作为事件类型参数传到**addEventCallback()**方法中。

当我们使用事件回调函数时，在前面章节中讨论的**SoHandleEventAction**类会在幕后完成它的工作。如果有必要，**SoHandleEventAction**会执行一次拾取操作，并且会缓存拾取信息。事件回调函数自己有责任设置当前的事件是否已经处理过了（使用**setHandled()**方法）。如果在一个事件回调节点中包含有多个事件回调函数，这些回调函数都会被调用，不管它们是否某个回调函数已经处理过了这个事件。

在事件回调函数中，可以使用下列**SoEventCallback**中的方法，这些方法同样使用在标准的Inventor事件处理过程中。

getAction()	返回所应用的处理事件动作。
getEvent()	返回当前处理的 Inventor 事件。
getPickedPoint()	返回物体的拾取点击位置。SoHandleEventAction 会自动执行这个拾取操作。
grabEvents()	通知事件回调节点要拦截事件。然而，即使调用了此函数，仍然是只有在指定类型的事件发生时，事件回调函数才会被调用。
releaseEvents()	通知事件回调节点要停止拦截事件
setHandled()	通知动作，当前事件已经被处理。
isHandled()	返回当前事件是否已经被处理过。

例 10-1 将演示带有**SoEventCallback**节点的事件回调函数的用法。例子创建了一个对按键事件感兴趣的事件回调节点，然后调用**addEventCallback()**方法，注册了*myKeyPressCB*回调函数。场景中有四个物体可供选择（使用Shift键可以同时选中多个物体）。当一个按键按下时，回调函数会检查是那个按键被按下了，如果是上或下箭头键被按下，程序将会缩放那些被选择的物体。

例 10-1 使用事件回调

```
// An event callback node so we can receive key press events
SoEventCallback *myEventCB = new SoEventCallback;
myEventCB->addEventCallback(
         SoKeyboardEvent::getClassTypeId(),
         myKeyPressCB, selectionRoot);
selectionRoot->addChild(myEventCB);

...

// userData is the selectionRoot from main().
void
myKeyPressCB(void *userData, SoEventCallback *eventCB)
{
   SoSelection *selection = (SoSelection *) userData;
   const SoEvent *event = eventCB->getEvent();

   // Check for the Up and Down arrow keys being pressed.
   if (SO_KEY_PRESS_EVENT(event, UP_ARROW)) {
      myScaleSelection(selection, 1.1);
      eventCB->setHandled();
   } else if (SO_KEY_PRESS_EVENT(event, DOWN_ARROW)) {
      myScaleSelection(selection, 1.0/1.1);
      eventCB->setHandled();
   }
}
```

直接向应用程序发送事件（*高级内容*）

有时，我们可能希望绕过Inventor处理事件的过程，直接将所有事件都发送给应用程序。**SoXtRenderArea**类中包含有一个方法，这个方法可以将所有的事件直接发送给应用程序定义的事件处理函数中。例如：

```
SoXtRenderArea *myRenderArea;
myRenderArea->setEventCallback(myEventCallback, userData);
```

当使用一个非 NULL 的用户函数指针作为参数来调用 setEventCallback 函数后，所有进入到渲染区的事件都将被传递到用户定义的函数中。用户定义的回调函数应该返回一个布尔值。如果布尔值是 TRUE，则表示用户回调函数已经处理过了这个事件，渲染区不必再将这个事件发送给场景管理器处理了。如果布尔值是 FALSE，则这个事件将被发送到场景中，以便 Invnetor 继续处理。

注意：直接发送到用户回调函数中的事件不再是Inventor类型的事件了。如果使用了**SoXtRenderArea**渲染区，那么传递进来的事件是X Window事件。应用程序会接收到所有的事件，即使有些事件并不需要转换成Inventor事件。

例 10-2 将演示**setEventCallback()**函数的用法，这个函数会将使所有事件都直接发送到应用程序中，不再发送到场景中。

例 10-2 直接向应用程序发送事件

```
// Clicking the left mouse button and dragging will draw
// points in the xy plane beneath the mouse cursor.
// Clicking middle mouse and holding causes the point set
// to rotate about the Y axis.
// Clicking right mouse clears all points drawn so far out
// of the point set.
...
// Have render area send events to us instead of the scene
// graph.  We pass the render area as user data.
myRenderArea->setEventCallback(
            myAppEventHandler, myRenderArea);

SbBool
myAppEventHandler(void *userData, XAnyEvent *anyevent)
{
   SoXtRenderArea *myRenderArea = (SoXtRenderArea *) userData;
   XButtonEvent *myButtonEvent;
   XMotionEvent *myMotionEvent;
   SbVec3f vec;
   SbBool handled = TRUE;

   switch (anyevent->type) {

   case ButtonPress:
      myButtonEvent = (XButtonEvent *) anyevent;
      if (myButtonEvent->button == Button1) {
         myProjectPoint(myRenderArea,
                  myButtonEvent->x, myButtonEvent->y, vec);
         myAddPoint(myRenderArea, vec);
      } else if (myButtonEvent->button == Button2) {
         myTicker->schedule();  // start spinning the camera
      } else if (myButtonEvent->button == Button3) {
         myClearPoints(myRenderArea);  // clear the point set
      }
      break;
```

```
   case ButtonRelease:
      myButtonEvent = (XButtonEvent *) anyevent;
      if (myButtonEvent->button == Button2) {
         myTicker->unschedule();  // stop spinning the camera
      }
      break;

   case MotionNotify:
      myMotionEvent = (XMotionEvent *) anyevent;
      if (myMotionEvent->state & Button1Mask) {
         myProjectPoint(myRenderArea,
                  myMotionEvent->x, myMotionEvent->y, vec);
         myAddPoint(myRenderArea, vec);
      }
      break;

   default:
      handled = FALSE;
      break;
   }

   return handled;
}
```

（上面的例子代码是针对 X Window 系统的。在我们提供的本书例子代码中，有专门针对 Microsoft Windows 系统的代码，读者如果感兴趣的话可以下载查看。译者注）

选择器（Selection）

SoSelection节点提供了一些关于用户交互性操作的额外特性。这些特性包括：可管理的选择列表（在“选择规则”章节中做了介绍），高亮选择物体，当选择列表发生变化时，可以调用用户自定义的函数。下面的章节将分别描述这些特性。

可管理的选择列表

SoSelection节点会维护一个被选物体的路径列表。这个列表叫做*选择列表*。通常，用户可以选择一个或多个物体，然后对这些被选物体执行一个操作，例如像拷贝它们，删除它们，修改它们的颜色等。选择列表中的每个路径都起始于**SoSelection**节点，结束于被选物体。有两种方法可以将物体增加到列表中或从列表中删除掉。

- 通过选择器本身的事件处理机制。这种机制是基于当前选择规则的。
- 通过调用**SoSelection**类的方法，我们可以从选择列表中选择、取消、切换或清除物

体。只有在希望直接管理选择列表的时候，我们才需要调用这些方法。

下面是用于直接管理选择列表的**SoSelection**类方法：

select(*path*)	将一个路径增加到选择列表中
deselect(*path*)	从选择列表中删除一个路径
toggle(*path*)	切换一个路径在选择列表中的存在状态（即如果路径不在选择列表中，则将路径增加到列表中。如果路径已经在列表中了，则从列表中删除这个路径）
deselectAll()	从选择列表中删除所有的路径
isSelected()	如果传入的路径参数已经位于选择列表中，函数返回 TRUE.
getNumSelected()	返回选择列表的长度。
getList()	返回选择列表
getPath(*index*)	返回选择列表中某个路径

有时如果我们可以很容易获得节点数据，而不大容易得到节点的路径数据，这时也可以对上面的函数使用节点作为输入参数。如果一个节点在场景中有多个实例，那么 Inventor 将使用指向节点第一个实例的路径作为参数。

例如，假设在场景中的包含有一辆汽车模型，组成汽车模型的每个子模型都带有一个名字。用户可以通过在一个名字列表框中选择一个名字（例如“轮毂罩”）来选中场景中对应的物体。程序首先使用用户选择的名字，找到被选物体的路径，然后将这个路径增加到选择列表中。例 10-3 将演示使用一个 Motif 风格（**X Window 系统的一种窗口显示风格。译者注**）的文本列表来选择物体。这个例子将演示如何在一个包含有四个物体名称（立方体，球体，锥体，柱体）的列表窗口中，通过选择名字来选择和取消选择场景中的物体。

例 10-3 使用一个 Motif 风格的文本列表选择物体

```
//  The scene graph has 4 objects which may be
//  selected by picking with the left mouse button
//  (use shift key to extend the selection to more
//  than one object).
//
//  Hitting the up arrow key will increase the size of
//  each selected object; hitting down arrow will decrease
//  the size of each selected object.
//
//  This also demonstrates selecting objects from a Motif-style
//  list, and calling select/deselect functions on the
//  SoSelection node to change the selection. Use the Shift
//  key to extend the selection (i.e. pick more than one
//  item in the list.)
...
```

```
enum objects {
        CUBE,
        SPHERE,
        CONE,
        CYL,
        NUM_OBJECTS
};

static char *objectNames[] = {
        "Cube",
        "Sphere",
        "Cone",
        "Cylinder"
};
...
   cube->setName(objectNames[CUBE]);
   sphere->setName(objectNames[SPHERE]);
   cone->setName(objectNames[CONE]);
   cyl->setName(objectNames[CYL]);
...
   // Create a table of object names
   XmString *table = new XmString[NUM_OBJECTS];
   for (i=0; i<NUM_OBJECTS; i++) {
       table[i] = XmStringCreate(objectNames[i],
                                 XmSTRING_DEFAULT_CHARSET);
   }

   // Create the list widget
   n = 0;
   XtSetArg(args[n], XmNitems, table);
   n++;
   XtSetArg(args[n], XmNitemCount, NUM_OBJECTS);
   n++;
   XtSetArg(args[n], XmNselectionPolicy, XmEXTENDED_SELECT);
   n++;

   motifList = XmCreateScrolledList(shell, "funcList", args, n);
   XtAddCallback(motifList, XmNextendedSelectionCallback,
      (XtCallbackProc) myListPickCB, (XtPointer) selection);
...
   // Clear the selection node, then loop through the list
   // and reselect
   selection->deselectAll();
```

```
   // Update the SoSelection based on what is selected in
   // the list.  We do this by extracting the string
   // from the selected XmString, and searching for the
   // object of that name.
   for (int i = 0; i < listData->selected_item_count; i++) {
      mySearchAction.setName(
               SoXt::decodeString(listData->selected_items[i]));
      mySearchAction.apply(selection);
      selection->select(mySearchAction.getPath());
   }
```

例 10-4 是另一个演示如何使用选择列表的例子。这个例子可以让用户同时选择若干个物体，然后将它们放大两倍。在例子中，通过调用**getList()**和**getPath()**方法，得到了每个被选物体的路径，然后在路径中查找到适当的**SoTransform**节点，最后修改**scaleFactor**域值来改变物体的大小。

例 10-4 使用选择列表

```
// Scale each object in the selection list
void
myScaleSelection(SoSelection *selection, float sf)
{
   SoPath *selectedPath;
   SoTransform *xform;
   SbVec3f scaleFactor;
   int i,j;

   // Scale each object in the selection list

   for (i = 0; i < selection->getNumSelected(); i++) {
      selectedPath = selection->getPath(i);
      xform = NULL;

      // Look for the shape node, starting from the tail of the
      // path.  Once we know the type of shape, we know which
      // transform to modify
      for (j=0; j < selectedPath->getLength() &&
                (xform == NULL); j++) {
         SoNode *n = (SoNode *)selectedPath->getNodeFromTail(j);
         if (n->isOfType(SoCube::getClassTypeId())) {
            xform = cubeTransform;
```

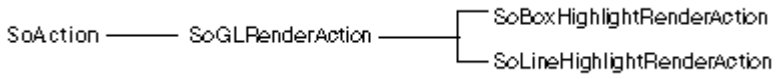
```
        } else if (n->isOfType(SoCone::getClassTypeId())) {
           xform = coneTransform;
        } else if (n->isOfType(SoSphere::getClassTypeId())) {
           xform = sphereTransform;
        } else if (n->isOfType(SoCylinder::getClassTypeId())) {
           xform = cylTransform;
        }
     }
     // Apply the scale
     scaleFactor = xform->scaleFactor.getValue();
     scaleFactor *= sf;
     xform->scaleFactor.setValue(scaleFactor);
   }
}
```

高亮选择物体

通常，当我们选中场景中某些物体时，我们希望以高亮方式显示这些物体，或使用其它可以将选中的物体与未选中的物体区别出来的方式。Inventor的**SoXtRenderArea**类提供了两种高亮显示方式。我们可以选择按照以某种特定颜色、宽度、和样式的框线风格来高亮绘制物体。也可以选择在选中物体的周围绘制一个包围盒。高亮类型可以在**SoXtRenderArea**中设置。缺省情况下，不使用任何高亮显示方式。

图 10-6 显示了高亮方式类的类树图。因为高亮显示物体不过是另外一种渲染场景的方式，所以Inventor的高亮方式类都是从**SoGLRenderAction**派生出来的。如果我们希望创建自己定制的高亮形式，请参考*The Inventor Toolmaker*，第 11 章。

图 10-6 高亮方式类的类树图



调用**SoXtRenderArea**类的**setGLRenderAction()**方法，传入一个某种高亮方式类的指针，就可以指定使用那种方式的高亮显示。动作将以高亮方式渲染场景中被选中的物体。注意：只要我们创建了一个新的高亮方式类，将这个类传递给渲染区，那么我们必须负责在渲染区销毁后删除掉那个高亮方式类。渲染区不会替我们删除它的。

提示：当场景的选择状态发生了变化，渲染区使用的重绘传感器并不会触发场景重绘。只有调用了**SoXtRenderArea**类的**redrawOnSelectionChanges()**方法，并使用一个需要监视其状态改变的**SoSelection**节点作为参数，才能保证在改变选择状态后重新绘制场景。

高亮显示是怎样发生的

首先，高亮渲染动作要渲染一次整个的场景。然后它再渲染一次选择列表中的每个节点。如果使用的是**SoLineHighlightRenderAction**，那么这个动作将使用指定的颜色，线段样式，和线宽等参数，以框线风格形式把被选物体绘制出来。下面是**SoLineHighlightRenderAction**可用的方法：

setColor（*color*）	指定高亮线框的颜色
setLinePattern（*pattern*）	指定高亮线框的线段样式
setLineWidth（*width*）	指定高亮线框的线宽

如果使用的是**SoBoxHighlightRenderAction**，那么这个动作将使用指定的颜色，线段样式，和线宽等参数，在被选物体的周围绘制出一个包围盒。**SoBoxHighlightRenderAction**的可用方法和**SoLineHighlightRenderAction**方法是相同的。

定制高亮显示风格

如果我们希望使用一种不同于 Inventor 提供的线型和包围盒型的高亮风格，可以尝试下列两种方法：

- 创建一个新的**SoGLRenderAction**子类，然后将这个子类作为参数传到**renderArea->setGLRenderAction()**函数中。
- 使用NULL参数，调用**addSelectionHighlight()**函数。然后使用下面将要讨论的“选择回调函数”功能。向场景中增加上用于形体节点、绘制风格节点、和其它用于高亮选择物体所需的节点。一种比较通用的高亮显示技术是通过使用选择回调函数，向被选物体上增加一个操作器。

在 *The Inventor Toolmaker*，第 10 章节中，列有更多有关创建自定义高亮方式的详细解释。

用于选择变化的回调函数

SoSelection类中有多种类型的回调函数：选中回调函数、取消选中回调函数、拾取过滤回调函数、选择开始回调函数、选择结束回调函数。例如，如果我们想开发一个回调函数，其功能是每次物体被选中时，对物体增加上一个包围它的轨迹球操作器。那么我们应该使用“选中回调函数”的类型。同时，我们还应该编写另外一个回调函数，用于当物体被取消选中后，删除轨迹球操作器。很显然，这个函数应该是“取消选中回调函数”的类型。“拾取过滤回调函数”的作用是，当一个物体被拾取了，并且这个物体或者将被选中、或者将被取消选中时，Inventor就会调用拾取过滤回调函数。这个函数允许我们在选择路径中某个特定类型的节点（像NodeKit节点 见例 10-8）处截断选择路径。

当选择列表将要发生改变的时候，Inventor 就会调用“选择开始回调函数”。而当选择列表已经完成了变化，Inventor 就会调用“选择结束回调函数”。这两种函数可以用来实现撤销（undo）和重做（redo）功能。也就是说，当将要发生选择改变时，把当前的状态保存

到一个撤销（undo）缓存中，当需要撤销一个修改时，就可以从撤销（undo）缓存中恢复以前保存的状态了。

为选择物体使用回调函数的另外一个例子是，有时，我们可能需要这样的一个功能，当用户选中了场景中的某个物体后，程序会显示出一个材质编辑器，之后用户就通过它来修改物体的材质参数了。要想实现此功能，必须要在用户选中了一个物体后，使用一个回调函数来检查当前是否已经显示了一个材质编辑器，如果没有显示，则首先需要创建显示出一个材质编辑器。否则如果当前已经有一个材质编辑器，那么回调函数就需要找到可用影响到被选物体的材质节点，然后将此材质节点与材质编辑器相关联，这样用户就可以修改物体的材质了。

下列函数允许我们设置当物体被选或取消选择后的处理方式。

addSelectionCallback(*functionName*, *userData*) removeSelectionCallback(*functionName*, *userData*)	物体被选中后，调用此回调函数
addDeselectionCallback(*functionName*, *userData*) removeDeselectionCallback(*functionName*, *userData*)	物体取消选中后，调用此回调函数
setPickFilterCallback(*functionName*, *userData*)	物体被拾取，并且或者将要被选中或者将要被取消选中时，就会调用此回调函数。这个函数允许我们在选择路径中某个特定类型的节点处截断选择路径。
addStartCallback(*functionName*, *userData*) removeStartCallback(*functionName*, *userData*)	当选择列表将要发生改变的时候，就会调用此回调函数。
addFinishCallback(*functionName*, *userData*) removeFinishCallback(*functionName*, *userData*)	选择列表已经完成了变化，就会调用此回调函数

这些方法都允许我们使用一个回调函数和一个用户自定义数据作为参数。如果我们对过滤回调函数设置为 NULL,那么无论我们拾取什么物体，这些物体都将被选中或者被取消选中（If you specify NULL for the pick filter callback function, whatever is picked will be selected and deselected）。

例 10-5 将举例说明选择回调函数的用法。例子中的场景包含有一个球体和一个 3D 文本对象。在场景结构的顶层放置了一个选择器节点。当用户选中了一个物体时，将调用选择回调函数，函数将修改物体的材质颜色。

例 10-5 使用选择回调函数

```
#include <X11/Intrinsic.h>
#include <Inventor/Sb.h>
#include <Inventor/SoInput.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoPickStyle.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoText3.h>
#include <Inventor/nodes/SoTransform.h>

// global data
SoMaterial *textMaterial, *sphereMaterial;
static float reddish[] = {1.0, 0.2, 0.2};
static float white[] = {0.8, 0.8, 0.8};

// This routine is called when an object gets selected.
// We determine which object was selected, and change
// that object's material color.
void
mySelectionCB(void *, SoPath *selectionPath)
{
   if (selectionPath->getTail()->
            isOfType(SoText3::getClassTypeId())) {
      textMaterial->diffuseColor.setValue(reddish);
   } else if (selectionPath->getTail()->
            isOfType(SoSphere::getClassTypeId())) {
      sphereMaterial->diffuseColor.setValue(reddish);
   }
}

// This routine is called whenever an object gets deselected.
// We determine which object was deselected, and reset
// that object's material color.
void
myDeselectionCB(void *, SoPath *deselectionPath)
{
   if (deselectionPath->getTail()->
            isOfType(SoText3::getClassTypeId())) {
      textMaterial->diffuseColor.setValue(white);
```

```
   } else if (deselectionPath->getTail()->
            isOfType(SoSphere::getClassTypeId())) {
      sphereMaterial->diffuseColor.setValue(white);
   }
}

void
main(int argc, char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   // Create and set up the selection node
   SoSelection *selectionRoot = new SoSelection;
   selectionRoot->ref();
   selectionRoot->policy = SoSelection::SINGLE;
   selectionRoot-> addSelectionCallback(mySelectionCB);
   selectionRoot-> addDeselectionCallback(myDeselectionCB);

   // Create the scene graph
   SoSeparator *root = new SoSeparator;
   selectionRoot->addChild(root);

   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);

   // Add a sphere node
   SoSeparator *sphereRoot = new SoSeparator;
   SoTransform *sphereTransform = new SoTransform;
   sphereTransform->translation.setValue(17., 17., 0.);
   sphereTransform->scaleFactor.setValue(8., 8., 8.);
   sphereRoot->addChild(sphereTransform);

   sphereMaterial = new SoMaterial;
   sphereMaterial->diffuseColor.setValue(.8, .8, .8);
   sphereRoot->addChild(sphereMaterial);
   sphereRoot->addChild(new SoSphere);
   root->addChild(sphereRoot);

   // Add a text node
   SoSeparator *textRoot = new SoSeparator;
```

```
   SoTransform *textTransform = new SoTransform;
   textTransform->translation.setValue(0., -1., 0.);
   textRoot->addChild(textTransform);

   textMaterial = new SoMaterial;
   textMaterial->diffuseColor.setValue(.8, .8, .8);
   textRoot->addChild(textMaterial);
   SoPickStyle *textPickStyle = new SoPickStyle;
   textPickStyle->style.setValue(SoPickStyle::BOUNDING_BOX);
   textRoot->addChild(textPickStyle);
   SoText3 *myText = new SoText3;
   myText->string = "rhubarb";
   textRoot->addChild(myText);
   root->addChild(textRoot);

   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
   myRenderArea->setSceneGraph(selectionRoot);
   myRenderArea->setTitle("My Selection Callback");
   myRenderArea->show();

   // Make the camera see the whole scene
   const SbViewportRegion myViewport =
            myRenderArea->getViewportRegion();
   myCamera->viewAll(root, myViewport, 2.0);

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

拾取过滤回调函数（*高级内容*）

拾取过滤回调函数将返回一个表示被选中的、被取消选中的、或被切换选中的新物体节点的路径。

```
typedef SoPath *SoSelectionPickCB( void *userData, SoDetail *d);
void setPickFilterCallback(SoSelectionPickCB *f,
         void *userData = NULL,
         SbBool callOnlyIfSelectable = TRUE);
```

这个回调函数可以通过拾取点找到当前被拾取的物体，返回一条可被应用到任何选择规则上的路径。回调函数可以截断拾取路径，使路径截至于某个特殊类型的节点位置。如果拾取了一个不可选择的物体，那么拾取过滤回调函数将决定如何使用那个信息（If an

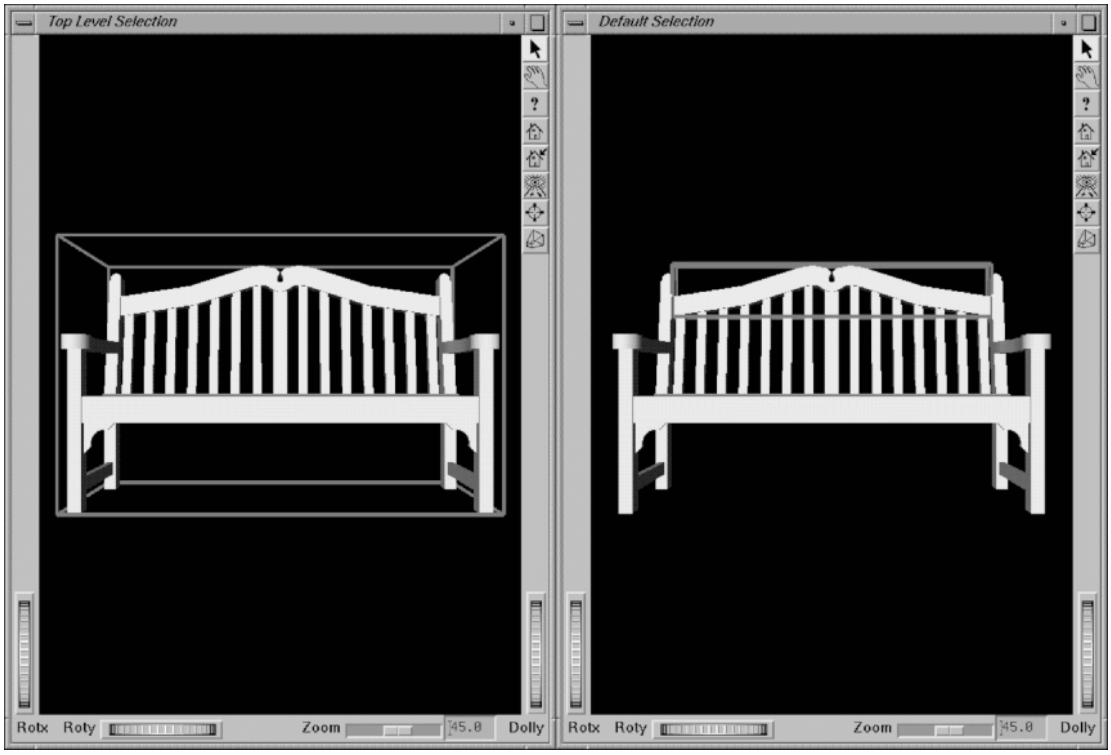
unselectable object is picked, the pick filter callback determines how that information is used)。当程序设置使用了拾取过滤回调函数后，程序可以将一个布尔值作为用户自定义参数，传入到回调函数中。通过这个布尔值参数来指定：是只有可选择物体才允许调用此函数，还是对于所有的物体（包括那些不可选择的物体）都允许调用此函数。然后，拾取过滤回调函数将可以返回下列路径值：

- NULL－表示不选择任何物体。(对于 SINGLE 和 SHIFT 选择规则来说，这种返回值相当于清空选择列表)。
- 路径（Path）－根据当前选择规则，选中或取消选中这个路径表示的物体。路径中必须包含有选择器节点。
- 路径（Path)，但路径中不包含选择器节点－选择器将忽略这次拾取事件并且选择列表不会发生任何变化。
- 只包含一个选择器节点的路径－表示尽管没有拾取到任何物体，但仍然需要执行选择规则，并继续遍历场景。

例 10-6 将使用图 10-7 演示应用程序所使用的简单拾取过滤回调函数。

例 10-6 演示了拾取过滤回调函数来完成一个“顶层（top-level)”选择规则的用法。使用这种“顶层（top-level)”选择规则，可以总是选择位于选择器节点之下的最顶端的物体，而不在选择实际点取的物体。 图 10-7 显示了这个例子创建的两个窗口。

图 10-7 顶层选择规则（左）和缺省选择规则（右）



例 10-6 创建顶层选择规则

```
#include <X11/StringDefs.h>
#include <X11/Intrinsic.h>

#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/SoPath.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/actions/SoBoxHighlightRenderAction.h>
#include <Inventor/misc/SoPickedPoint.h>
#include <Inventor/nodes/SoSelection.h>

// Pick the topmost node beneath the selection node
SoPath *
pickFilterCB(void *, const SoPickedPoint *pick)
{
   // See which child of selection got picked
   SoPath *p = pick->getPath();
   int i;
   for (i = 0; i < p->getLength() - 1; i++) {
      SoNode *n = p->getNode(i);
      if (n->isOfType(SoSelection::getClassTypeId()))
         break;
   }
   // Copy 2 nodes from the path:
   // selection and the picked child
   return p->copy(i, 2);
}


void
main(int argc, char *argv[])
{
   // Initialization
   Widget mainWindow = SoXt::init(argv[0]);

   // Open the data file
   SoInput in;
   char *datafile = "parkbench.iv";
   if (! in.openFile(datafile)) {
      fprintf(stderr, "Cannot open %s for reading.\n",
              datafile);
      return;
```

```
   }

   // Read the input file
   SoNode *n;
   SoSeparator *sep = new SoSeparator;
   while ((SoDB::read(&in, n) != FALSE) && (n != NULL))
      sep->addChild(n);

   // Create two selection roots - one will use the pick filter.
   SoSelection *topLevelSel = new SoSelection;
   topLevelSel->addChild(sep);
   topLevelSel->setPickFilterCallback(pickFilterCB);

   SoSelection *defaultSel = new SoSelection;
   defaultSel->addChild(sep);

   // Create two viewers, one to show the pick filter for top
   // level selection, the other to show default selection.
   SoXtExaminerViewer *viewer1 = new
            SoXtExaminerViewer(mainWindow);
   viewer1->setSceneGraph(topLevelSel);
   viewer1->setGLRenderAction(new SoBoxHighlightRenderAction());
   viewer1->redrawOnSelectionChange(topLevelSel);
   viewer1->setTitle("Top Level Selection");

   SoXtExaminerViewer *viewer2 = new SoXtExaminerViewer();
   viewer2->setSceneGraph(defaultSel);
   viewer2->setGLRenderAction(new SoBoxHighlightRenderAction());
   viewer2->redrawOnSelectionChange(defaultSel);
   viewer2->setTitle("Default Selection");

   viewer1->show();
   viewer2->show();

   SoXt::show(mainWindow);
   SoXt::mainLoop();
}
```

例 10-7 将演示怎样使用拾取过滤回调函数，让拾取“透过”操作器。在例子中，场景包含了一组“Open Inventor”字符，使用鼠标左键点击某个字符后，将选中这个字符，然后在这个字符上增加一个操作器。如果再次点击这个字符，将取消选中字符，并且删除掉操作器。拾取过滤回调函数用于取消选择字符而不是导致选择了操作器。

例 10-7 “透过”操作器拾取物体

```
SoPath *
pickFilterCB(void *, const SoPickedPoint *pick)
{
   SoPath *filteredPath = NULL;

   // See if the picked object is a manipulator.
   // If so, change the path so it points to the object the
   // manip is attached to.
   SoPath *p = pick->getPath();
   SoNode *n = p->getTail();
   if (n->isOfType(SoTransformManip::getClassTypeId())) {
      // Manip picked! We know the manip is attached
      // to its next sibling. Set up and return that path.
      int manipIndex = p->getIndex(p->getLength() - 1);
      filteredPath = p->copy(0, p->getLength() - 1);
      filteredPath->append(manipIndex + 1); // get next sibling
   }
   else filteredPath = p;

   return filteredPath;
}
```

例 10-8 将演示使用拾取过滤回调函数在 SoShapeKit 节点处截断拾取路径，这种过滤使编辑物体材质属性变得更容易，因为只有节点工具包才关心材质节点创建的细节。

例 10-8 选择节点工具包

```
// Truncate the pick path so a nodekit is selected
SoPath *
pickFilterCB(void *, const SoPickedPoint *pick)
{
   // See which child of selection got picked
   SoPath *p = pick->getPath();
   int i;
   for (i = p->getLength() - 1; i >= 0; i--) {
      SoNode *n = p->getNode(i);
      if (n->isOfType(SoShapeKit::getClassTypeId()))
         break;
   }

   // Copy the path down to the nodekit
```

```
    return p->copy(0, i+1);
}
```

第十一章 文件格式

本章目标

在阅读完本章后，我们将有能力做下列事情：

- 以 ASCII 或二进制格式将场景数据输出到文件中。
- 读取场景文件，将数据输入到 Inventor 数据库中。
- 利用 Inventor 场景文件创建图形场景。
- 从内存缓冲区中读入图形场景。

本章将描述 Inventor 的 ASCII 文件格式。无论何时，只要我们将输出动作（write action）应用到一个节点、路径、或路径列表上，那么场景数据就会以 ASCII 格式输出的文件中。使用数据库的读文件的方法，我们可以按照 ASCII 文件格式读取文件，将数据输入到 Inventor 数据库中。这种文件格式也可以用于在两个程序进程之间传递 3D 拷贝和粘贴数据。

输出图形场景

如在第 9 章中所描述的那样，我们可以将输出动作应用到一个节点、路径、或路径列表上。当输出动作应用到一个节点上时，它会以这个节点为根节点，输出整个子场景。

```
SoWriteAction    writeAction;
writeAction.apply(root); //writes the entire scene graph to stdout
```

从文件中读入场景

使用Inventor数据库的**readAll()**方法，我们可以将图形场景数据从一个文件中读到场景数据库中。下面的代码是给定一个文件名称，读入此文件，然后返回一个包含此文件场景数据的SoSeparator节点。如果在读文件的过程中发生了错误，函数返回NULL。

```
SoSeparator *
readFile(const char *filename)
{
   // Open the input file
   SoInput mySceneInput;
   if (!mySceneInput.openFile(filename)) {
      fprintf(stderr, "Cannot open file %s\n", filename);
      return NULL;
   }
   // Read the whole file into the database
   SoSeparator *myGraph = SoDB::readAll(&mySceneInput);
```

```
   if (myGraph == NULL) {
      fprintf(stderr, "Problem reading file\n");
      return NULL;
   }
   mySceneInput.closeFile();
   return myGraph;
}
```

Inventor有两种**read()**方法。一种方法是读取以一个节点为根节点的场景文件，这种方法返回一个指向根节点的指针。另一种方法是读取一个由路径定义的场景。我们必须要根据输入内容的不同，选择正确的读入方法。当读入一个模型文件时，我们通常使用读入节点的方法。如果是使用路径来剪切和粘贴场景的话，我们必须使用读路径的方法。

当要读入Inventor数据文件时，需要使用**SoDB**类。这个类也可以从内存缓冲区中读入数据。缺省情况下，**SoInput**会在程序当前目录下寻找指定的文件（除非文件名称以“/”开始）。我们可以使用**addDirectoryFirst()**和**addDirectoryLast()**方法来增加搜索文件的路径（见*Open Inventor C++ Reference Manual*书中**SoInput**类）。使用**clearDirectories()**方法可以清除搜索文件路径列表。

我们还可以使用**SoInput**的下列方法，将搜索路径列表作为环境变量增加到系统中。

addEnvDirectoriesFirst()
addEnvDirectoriesLast()

文件语法格式

在接下来的章节中，我们将概要描述 Inventor ASCII 文件格式的语法形式。这种文件格式将忽略掉由空格、制表符、和回车符所产生的多余空白字符。注释是以“#”字符为开始，直到这行字符串结束为止。“#”字符可以位于本行字符串的任意位置上。

```
# this is a comment in the Inventor file format
```

为了简单起见，本节讨论的内容将主要放在图形场景如何输出到文件中。同样的格式也可以应用在我们创建的可被读入到 Inventor 数据库中的文件上（This same format applies to files you create that will be read into the Inventor database.）。

在 *Open Inventor C++ Reference Manual* 书中将描述每个 Inventor 类的文件格式。

文件头

每个 Inventor 数据文件都必须带有一个标准的文件头，以供鉴别文件是否是 Inventor 数据文件。文件头必须位于文件的第一行，使用下列形式：

```
#Inventor V2.0 ascii
```

或

```
#Inventor V2.0 binary
```

我们可以使用**SoDB::isValidHeader()**方法，将当前需要验证的文件的开始部分作为参数传入函数中，来确定那个文件是否是一个Inventor文件。尽管文件头可能会因为版本的不同而发生改变（当前版本是 2.0）,但所有版本的Inventor文件都可以保证文件头是以“#”字符作为开始，不超过 80 个字符，并且以换行符作为结束。所以，也可以使用C语言的文件函数来验证Inventor文件（Therefore, the C () routine can be used)。如果文件包含有一个Inventor文件头，那么**isValidHeader()**方法返回TURE。Inventor也可以读取老版本（V1.0）的Inventor文件，并且会自动转换它们。

输出节点

输出节点时，会输出下列元素：

- 节点名称（节点的类名称去掉 So 前缀，就是输出的节点名称）
- 左括号（{）
- 节点中的域值（如果有的话），然后接着是节点的子节点（如果有的话）
- 右括号（}）

例如：

```
DrawStyle {
   style         LINES
   lineWidth         3
   linePattern     255
}
```

输出域值

在输出节点中的域时，首先输出域的名称，然后是输出域值或域值组。如果域值没有发生改变，仍然使用它的缺省值，那么可以不必输出这个域。节点中的域可以按照任意的顺序输出到文件中。下面是一个输出域值的例子：

```
Transform {
            translation        0 -4 0.2
}
LightModel {
   model            BASE_COLOR
}
```

```
Material {
   ambientColor   .3 .1 .1
   diffuseColor  [.8 .7 .2,
                   1 .2 .2,
                  .2  1 .2,
                  .2 .2  1]
   specularColor  .4 .3 .1
   emissiveColor  .1  0 .1
}
```

在上面的例子中，**diffuseColor**域是一个多值域，输出多值域的语法是：使用方括号包围多值域，使用逗号来分隔每个域值。在最后一个域值的后面最好也有一个逗号（**没有也可以。译者注**）。

[*value1, value2, value3,*]

单值域不包括任何的方括号和逗号。多值域通常都带有方括号，但如果只存在一个域值的话，也可以不输出方括号。

```
specularColor    .4 .3 .1
```

或

```
specularColor   [.4 .3 .1]
```

域值的输出格式依赖于域类型。

域类型	**允许的输出格式**
longs, shorts, unsigned shorts	以十进制，十六进制，或八进制表示的整型数。例如： 255 0xff 0177
floats	整型或浮点数。例如 13 13.0 13.123 1.3e-2
names,	如果名字中包含有多个单词，则需要用双引号括住名字，如果名字只是一个

strings　　单词，则双引号是可选的。例如：

```
label   " front left leg "
label   car
```

字符串中可以包含任意的 ASCII 字符，包括换行符和反斜杠符。但如要包含一个双引号字符，必须在双引号字符前加上反斜杠符（\"）。

enums　　即可以是枚举名称格式，也可以是枚举整数格式（为了方便移植和代码的可读性，建议使用枚举名称格式）例如：

```
MaterialBinding {
   value   PER_FACE
         }
```

bit mask　　一个或多个名称标志位，如果是多标志位，应该使用竖线符进行分隔。而且当使用多个标志位时，还需要使用圆括号：

```
         Cylinder {
            parts   SIDES
         }

         Cylinder {
            parts   (SIDES | TOP)
         }
```

vectors　　使用空格符分隔的 *n* 个浮点数：(SbVec*n*f, 其中 n 表示浮点数的个数)

```
PerspectiveCamera {
components of the   position   0 0 9.5
vector)  }
```

colors　　使用空格符分隔的 3 个浮点数（RGB）：

```
         BaseColor {
            rgb   0.3 0.2 0.6
         }
```

rotation　　一个代表轴方向的矢量，接着是一个表示旋转角度的浮点数（弧度单位），这些都是使用空格符分隔的：

```
Transform {
   rotation   0 1 0 1.5708
   # y axis ... pi\xb9 /2 radians
}
```

matrix　　使用空格符分隔的 16 个浮点数

path　　SFPath 中有一个指向节点的指针数据。要输出这个指针数据，必须要输出路径数据（见输出路径章节）。MFPath 带有多个指向路径的指针数据。要输出这些指针数据，就要使用方括号将路径列表括起来，并且使用逗号分隔每个路径：
[*first_path, second_path, ... nth_path*]

node　　SFNode 中有一个指向节点的指针数据。要输出这个指针数据，必须要输出节点数据。MFNode 带有多个指向节点的指针数据。要输出这些指针数据，就要使用方括号将节点列表括起来，并且使用逗号分隔每个节点：
[*node1, node2, ... noden*]

Boolean　　TRUE, FALSE 或 0, 1:

```
SoFile {
   isWriteBack   FALSE
}
```

忽略标志

域忽略标志（见第 3 章）的输出形式是在域值或域名称的后面输出一个“～”字符。例如：

```
transparency [ .9, .1 ] ~
```

或

```
transparency ~
```

第一种情况，即使这个域已经被忽略了，但文件中仍然保存其域值。第二种情况是使用缺省域值，但会忽略这个域。

提示：忽略标志只用于属性节点，不能用于照相机、灯光、和形体节点。

域连接

连接是作为对象被输出的，输出包括：域或连接到域上的输出，接着是一个句号（.），最后域或输出的名称（Connections are written as the object containing the field or output connected to the field, followed by a period (.) and then the name of the field or output）。例如：

```
Separator {
            DEF Trans Translation { translation  1 2 3 }
            Cube {}
        }
        Separator {
            Translation { translation 0 0 0 = USE Trans.translation }
            Cone {}
        }
```

被连接域的数值（0 0 0）是可选项，也可以不输出。所以第二个**Translation**节点也可以输出成：

```
Translation { translation = USE Trans.translation }
```

如果连接一个被忽略的域，那么应该在忽略标志之后输出域连接。

```
translation 000 ~ = USE Trans.translation
#or
translation ~ = USE Trans.translation
```

如果一个给定值也是一个连接的话，那么域可以直接使用这个值。如果该值是连接之后才传递过来的，那么这个值将覆盖掉原来的域值(If a value is given as well as a connection, the value is used for the field. If a value is sent along the connection later, it will override the value)。

全局域

全局域需要至少一个以上的连接才能被输出。全局域的输出格式如下：

GlobalField {
type
value
}

在大括号中包含有全局域的类型和域值。全局域的名称就是*value* 域的名称。例如一个**Text3**节点可以连接一个全局域（**currentFile**），这个全局域保存着应用程序当前使用的文件名称。所以，即使应用程序更换了当前使用的文件，**Text3**节点仍然可以总是正确地显示出当前使用的文件名称。

```
Text3 {
```

```
   string "" = GlobalField {
      type SFString
      currentFile "aircar.iv"
   } . currentFile
}
```

输出引擎

引擎与非组（nongroup）节点的输出语法格式是相同的。

EngineType {
input_fields
}

不能单独输出引擎。引擎必须要和场景中的某个节点相连接后才能被输出。下面是域-引擎连接的语法结构：

fieldname value = engine . outputname

下面的例子将演示使用**SoOneShot**引擎来改变球体的半径：

```
Sphere {
   radius 0.5 = OneShot { duration 3.0 } . ramp
}
```

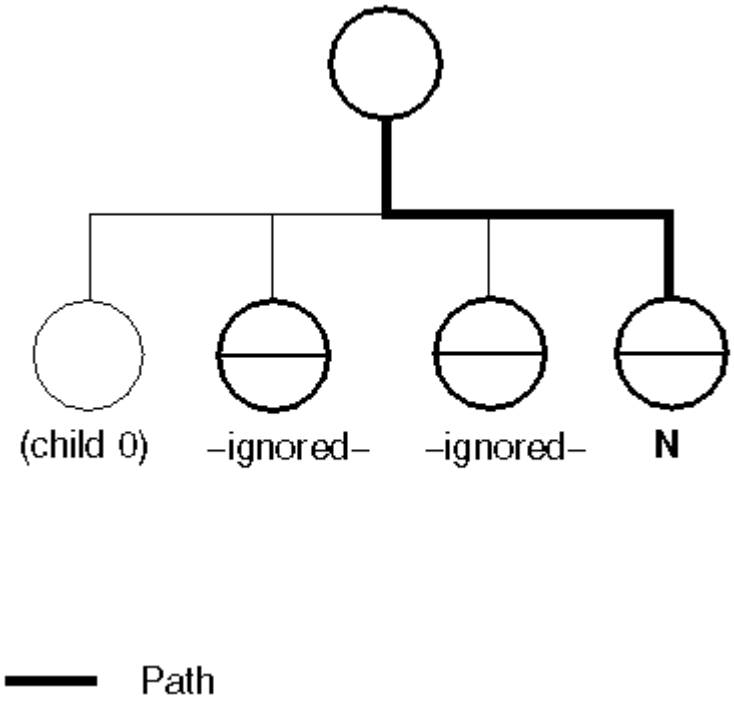
输出路径

输出路径（见第 3 章）时，需要输出下列内容：

- 关键字**Path**
- 左括号
- 以路径的起始节点为根的整个子场景
- 其它路径链中索引的个数（Number of indices in the rest of the path chain）
- 路径中每个子节点的索引值
- 右括号

当 Inventor 在子场景中遍历到了 SoSeparator 组节点的时候，如果这些组节点不会对路径中的节点产生影响的话，Inventor 就会忽略这些组节点。为在组中的子节点输出索引值是为了对那些因忽略而跳过的组节点做适当的调整(Written indices for the children within a group are adjusted to account for the skipped separator groups)。例如，在图 11-1 中，节点 N 的输出索引值是 1，这是因为左边两个子 SoSeparator 组节点不会对节点 N 产生任何影响。(注意：路径自身保存的节点索引值没有发生改变，只是输出到文件中的时候索引值发生了改变)

图 11-1 为 SoSeparator 组节点调整路径索引值



注意：如果一个路径中包含有一些连接，而与这些连接相关联的节点却不包含在这个路径中，那么这些节点不会被输出。例如：一个路径中包含有一个引擎，引擎连接到路径外的一个节点上，输出路径时，将只输出引擎，而不会输出那个被连接的节点。

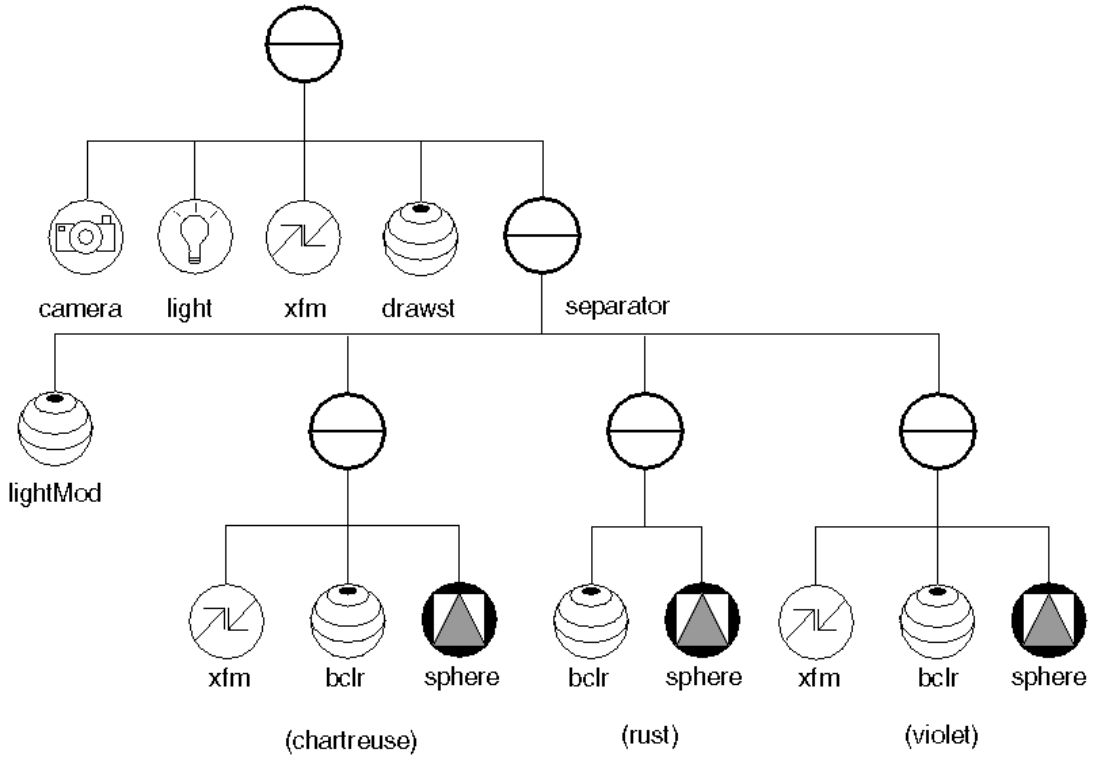
例 11-1 将演示输出路径的过程，下面是一个图形场景文件，场景中包含有黄绿色、红褐色球体和紫罗兰色的三个球体。

```
Separator {
   PerspectiveCamera {
      position 0 0 9.53374
      aspectRatio 1.09446
      nearDistance 0.0953375
      farDistance 19.0675
      focalDistance 9.53374
   }

   DirectionalLight {
   }

   Transform {
      rotation -0.189479 0.981839 -0.00950093 0.102051
      center 0 0 0
```

```
   }

   DrawStyle {
   }

   Separator {
      LightModel {
         model   BASE_COLOR
      }
      Separator {
         Transform {
            translation -2.2 0 0
         }
         BaseColor {
            rgb .2 .6 .3   # chartreuse
         }
         Sphere { }
      }
      Separator {
         BaseColor {
            rgb .6 .3 .2   # rust
         }
         Sphere { }
      }
      Separator {
         Transform {
            translation 2.2 0 0
         }
         BaseColor {
            rgb .3 .2 .6   # violet
         }
         Sphere { }
      }
}
```

图 11-2 展示了这个文件的场景图

图 11-2 包含有三个球体的场景图



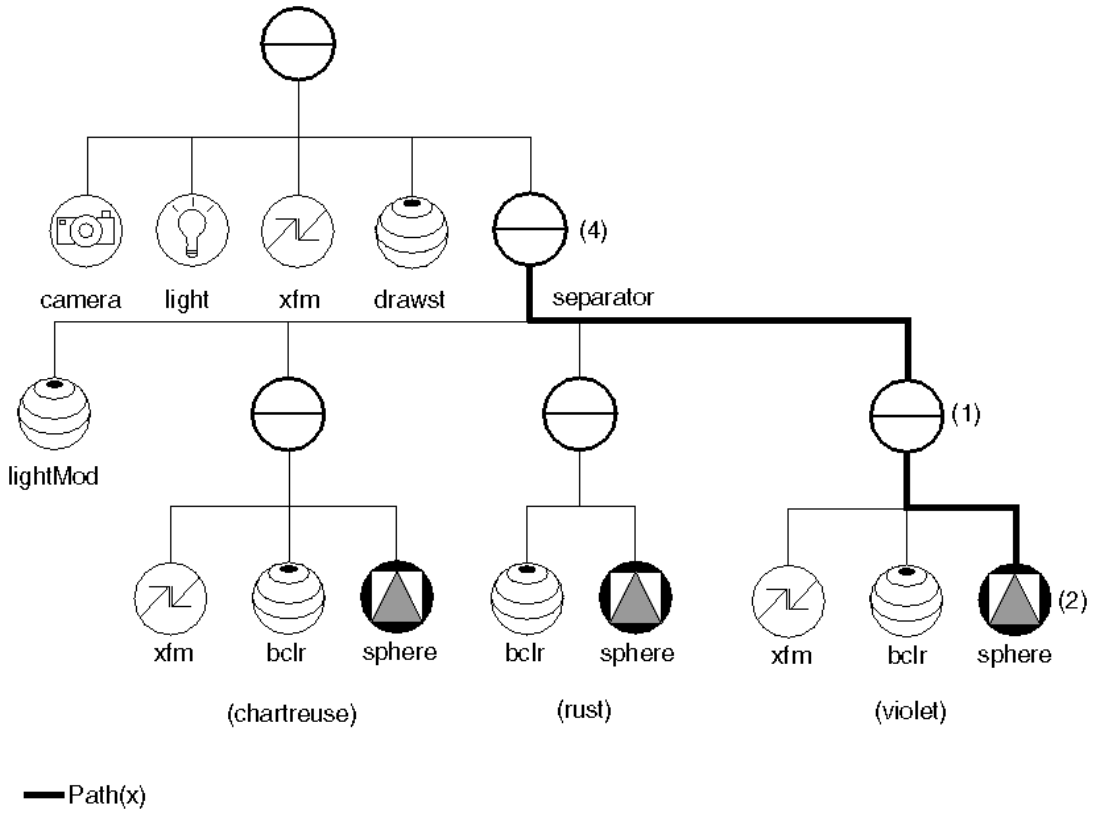
如果我们点击拾取了第三个球（紫罗兰色的），那么例 11-1 将是这个拾取路径的输出。首先输出的是根节点下的子场景。接下来输出路径中索引的个数（3）。最后输出路径中每个子节点的索引值。

例 11-1 输出路径

```
Path {
   Separator {
      PerspectiveCamera {
         position 0 0 9.53374
         aspectRatio 1.09446
         nearDistance 0.0953375
         farDistance 19.0675
         focalDistance 9.53374
      }
      DirectionalLight {
      }
      Transform {
         rotation -0.189479 0.981839 -0.00950093 0.102051
      }
      DrawStyle {
      }
      Separator {
         LightModel {
```

```
            model BASE_COLOR
         }
         Separator {
            Transform {
               translation 2.2 0 0
            }
            BaseColor {
               rgb 0.3 0.2 0.6
            }
            Sphere {
            }
         }
      }
   }
   3
   4
   1
   2
}
```

图 11-3 拾取紫罗兰色球体的路径



定义并使用共享的节点实例

在文件格式中，关键字 DEF 可以用来定义一个带名称的节点、路径或引擎的实例。这些实例可以在以后的 USE 关键字中被引用。例如：

```
// This example shows keeping a cone between two cubes using an
// InterpolateVec3f engine.

Separator {
   DEF A Translation { translation -4 0 0 }
   Cube { }
}
Separator {
   DEF B Translation { translation 4 5 6 }
   Cube { }
}
Separator {
   Translation { translation 0 0 0 =
                 InterpolateVec3f {
                    input0 0 0 0 = USE A.translation
                    input1 0 0 0 = USE B.translation
                    alpha 0.5
                 } . output
               }
   Cone { }
}
```

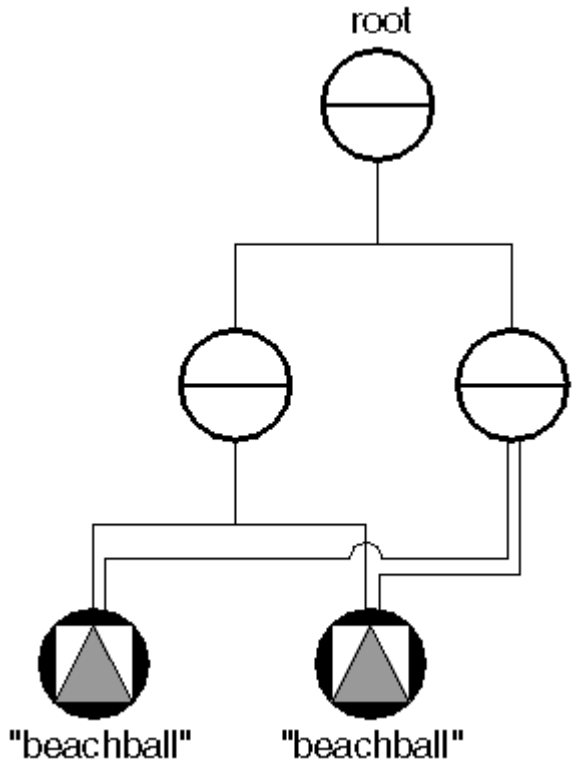
定义的名称可以是任意有效的**SbName**类型名称。在某种情况下，Inventor在输出节点名称时，会为名称增加上一些扩展字符。例如，在图 11-4 所示的一个稍微有些特殊的场景中，为了标识B节点到底使用了那个“气球”节点的实例，Inventor将以下列形式输出场景：

```
Separator{
   Separator{
      DEF beachball+0
      DEF beachball+1
   }
   Separator{
      USE beachball+0
      USE beachball+1
   }
}
```

当场景被重新读回到 Inventor 中后，这些原始的名称还会被保留，不过那么+n 的重复

符号将被丢弃。

图 11-4 节点的共享实例



输出节点工具包

当输出节点工具包时，Inventor 会同时输出工具包中每个部件的域值。例如：

```
AppearanceKit {
   lightModel                          LightModel { model PHONG }
   drawStyle                            DrawStyle  { style LINES }
   material                              Material
{ diffuseColor .5 .5 .5 }
   complexity        Complexity { value .5 }
}
```

在这种格式中，先输出域名（**lightModel**），接着输出节点名（**LightModel**），然后是节点所包含的域名与域值(每个部件都包含在一个**SoSFNode**域中)。如果某个部件没有被创建，或者被设置成NULL，那么就不会输出这个部件。然而，如果某个部件是缺省创建的（例如在**SoShapeKit**中的形体部件），即使它被显式地设置成NULL，它也会被输出。

下面的例子是一个嵌套的节点工具包。在这里，**appearance**的域值为外观工具包（appearance kit)。外观工具包中有一个**material**域。

```
SeparatorKit {
   appearance                          AppearanceKit {
      material                              Material { diffuseColor
1 1 1 }
   }
}
```

当Inventor输出一个节点工具包时，它会输出中间嵌套的部件。当我们自己手工编写工具包的输出内容时，我们可以使用一个简单易记的方法，忽略掉中间嵌套的部件。例如，如果我们忽略了**AppearanceKit**，**SeparatorKit**仍然会增加上一个**AppearanceKit**，并将那个**Material**节点插入其中。所以，我们可以简单地这样输出：

```
SeparatorKit {
   material                              Material { diffuseColor
1 1 1 }
}
```

在节点工具包中列表部件（list parts）的文件格式有一点特殊。每个列表部件都有三个标准的域：

containerTypeName 例如，separator 或 switch；字符串格式

childTypeName 是一个 SoMFString 类型的域。这个域中列有这个部件所允许包含的所有子节点的类型。

containerNode 部件所包含的子节点

例如，下面就是一个**SoSeparatorKit**实例的**childList**部件。

```
SeparatorKit {
   childList                           NodeKitListPart {
      containerTypeName       "Separator"
      childTypeNames             "SeparatorKit"
      containerNode                                  Separator {
         SeparatorKit {
            transform            Transform { translation      1 0 0 }
         }
         SeparatorKit {
            transform            Transform { translation      0 1 0 }
         }
         SeparatorKit {
            transform            Transform { translation      0 0 1 }
         }
      }
```

```
   }
}
```

缺省情况下，Inventor 不会输出像 SoSeparator 和 SoGroup，或带有缺省值的域等这些节点工具包的内部部件。但是，如果一个节点工具包位于一个路径中，那么节点工具包中所有的部件都将被输出，如下面的例子所示。通常，Inventor 会按照部件所定义的倒序方式输出每个部件，即先输出叶子节点。

```
#Inventor V2.0 ascii

SeparatorKit {
   appearance        DEF +0 AppearanceKit {
      material            DEF +1 Material {
         diffuseColor        1 0 1
      }
   }
   childList         DEF +2 NodeKitListPart {
      containerTypeName   "Separator"
      childTypeNames             "SeparatorKit"
      containerNode                     DEF +3 Separator {
         ShapeKit {
            appearance                        DEF +4 AppearanceKit {
               material                         DEF +5 Material {}
                              }
            transform                         DEF +6 Transform {}
            shape                     DEF +7 Cube {}
                              topSeparator
Separator {
               USE +4
               USE +6
               DEF +8 Separator {
                  USE +7
               }
            }

            shapeSeparator                          USE +8
         }
   }
}
topSeparator         Separator {
   USE +0
   USE +2
```

```
   }
}
```

包含其它文件

如果要将一个文件包含在另外一个文件中，需要使用一个**SoFile**节点。当我们要建立一个包含很多物体对象的场景时，这个节点是非常有用的。物体对象可以放在自己专门的文件中，然后使用**SoFile**节点来引用它们，不需要将这些物体对象拷贝倒新文件中了。**SoFile**节点按照下列格式输出：

```
File {
   name "myFile.iv"
}
```

这里，**name**的域值是被包含文件的名称。当Inventor读入**SoFile**节点时，会将myFile.iv文件的内容作为一个隐含的子节点增加到**SoFile**节点中。当Inventor输出**SoFile**节点时，将只输出文件名称（不输出隐含的子节点）。

在**SoFile**节点中的对象是不可编辑的。我们可以使用下面的方法拷贝**SoFile**节点的内容：

```
SoFile::copyChildren()
```

我们可以修改**SoFile**节点的**name**域。只要**name**域值发生了改变，Inventor都会立即读入新文件。如果名称不是一个绝对文件路径，Inventor将在**SoInput**节点中设置的目录列表中搜索这个文件。Inventor会自动将被读入的文件所在目录增加到搜索目录中。例如，假设我们有一个myFile.iv的文件，在这个文件中还引用了windmill.iv文件。

/usr/tmp/myFile.iv 中的内容：

```
#Inventor V2.0 ascii
File { name "myObjects/windmill.iv" }
```

usr/tmp/myObjects/windmill.iv 中的内容：

```
#Inventor V2.0 ascii
//format to make the windmill
```

当Inventor读入了/usr/tmp/myFile.iv文件后，/usr/tmp/目录就会自动增加到搜索目录列表中。当在myFile.iv文件中的**SoFile**节点调用**SoDB::read**时，**SoInput**将会查找usr/tmp/myObjects/windmill.iv。这时Inventor会成功读入windmill.iv文件（这时usr/tmp/myObjects目录也将会增加到搜索目录列表中）。当**SoFile**节点中的所有文件都读入

后，/usr/tmp/和usr/tmp/myObjects这两个目录将会从搜索目录列表中删除掉。

ASCII 和二进制版本

在**SoWriteAction**的**SoOutput**对象中有一个**setBinary()**方法。这个方法可以设置输出的文件格式是ASCII格式的（缺省格式）还是二进制格式的（见第 9 章）。**getOutput()**方法可以返回一个的**SoOutput**指针。当输出文件时，Inventor会在文件的起始部分插入一段字符串，以此来标识文件是ASCII格式还是二进制格式，这与设置Inventor版本号的方式很类似。

读取扩展的节点与引擎

正如 *The Inventor Toolmaker* 书中所描述的那样，开发者可以创建自己的节点和引擎类，并在新的应用程序中使用它们。本节将讨论如果我们读入的文件中引用了扩展的节点和引擎，而当前的程序或许可以，或许不可以获得开发这些扩展节点的代码时，Inventor 将如何处理。（在大多数情况下，节点和引擎的概念是可以互相交换的。所以本节将只简单地讨论节点的情况。当然节点和引擎还是稍微有一点差别的，如果遇到这些差别，我们将明确地提出来）。

当一个 Inventor 节点从一个文件中读入时，Inventor 会首先检查这个节点的类型是否已经注册过了。如果找到了相应的类型名称，那么 Inventor 就会创建此节点的一个实例，然后读入节点的域值。这种情况发生在创建扩展节点的代码本身就存在于我们的程序中。

然而，如果我们的程序没有连接那些创建扩展新节点的代码时，事情就变得稍微有趣了。如果我们所使用的系统支持编译对象（compiled objects）动态装载(在 Microsoft Windows 系统下，就是指动态链接库 DLL。译者注)，Inventor 可以找到这些编译对象并注册新节点。在这种情况下，开发新节点的作者应该提供新节点的编译对象文件，并将文件放在系统可以搜索到的目录下。（请读者自己检查所用系统是否支持编译对象动态装载，以及系统是如何指定搜索文件目录的）

未知节点和引擎的文件格式

如果Inventor不能定位到新节点或引擎的代码。它将会创建一个**SoUnknownNode**或**SoUnknownEngine**的实例。在所有新节点的文件格式中，第一个关键字命名为**fields**。然后是定义在节点中的所有域的类型和名称。例如：

```
WeirdNode {
   fields [ SFFloat length, SFLong data ]
   length 5.3
   Material {}
   Cube {}
}
```

未知节点有两个域：**length**和**data**。因为**data**域使用自己的缺省值，所以不必输出它。未知节点还包含有两个子节点，一个是**SoMaterial**，另一个是**SoCube**。这两个子节点位于**WeirdNode**节点域的后面。它们被当作隐藏的子节点来处理，渲染、拾取、搜索这些动作都不会使用它们。它们只用于输出动作。

对于新引擎的文件格式中，还包含有引擎输入输出的描述内容。例如：

```
WeirdEngine {
   inputs [ SFFloat factor, SFFloat seed ]
   factor 100
   seed 0.5
   outputs [ SFFloat result ]
}
```

因为没有相应的代码与未知节点相关联，所以对于未知节点的大多数操作都是没有任何作用的。然而，对这些节点仍然可以执行读、写、和搜索动作（但不会执行它们的子节点）。

备用表示（Alternate Representation）

新节点的作者可以为新节点提供一种备用的表示，这种表示可以用于节点被当作是未知节点的情况。节点中**alternateRep**域指定节点的备用表示。这个域可以包含一个完整的场景。这个场景可代替新节点执行拾取、渲染、和包围盒等动作。

下面是节点工具包提供的一个备用表示

```
Airplane {
   fields [ SFNode wing, SFNode fuselage, SFNode alternateRep ]
   wing Separator { ... the wing scene graph ... }
   fuselage Separator { ... the fuselage scene graph ... }
   alternateRep Separator {
      Cube {}
      Transform { translation 10 0 0 }
      Cone {}
   }
}
```

从字符串读入场景

在“从文件中读入场景”章节中，我们已经讨论了如何从文件中读入场景。例 11-2 将演示如何从一个存储在内存中的字符串读入场景。注意：当从内存缓冲区读取场景时，我们不再需要读取这个文件的文件头字符串了。例 11-2 使用索引面集节点创建了一个十二面体。

例 11-2 从字符串读入场景

```
// Reads a dodecahedron from the following string:
// (Note: ANSI compilers automatically concatenate
// adjacent string literals together, so the compiler sees
// this as one big string)

static char *dodecahedron =
   "Separator { "
      "   Material { "
   "      diffuseColor [ "
   "         1  0  0,   0 1  0,   0  0 1,   0  1  1, "
   "         1  0  1,  .5 1  0,  .5  0 1,  .5  1  1, "
   "         1 .3 .7,  .3 1 .7,  .3 .7 1,  .5 .5 .8 "
   "      ] "
   "   } "
   "   MaterialBinding { value PER_FACE } "
   "   Coordinate3 { "
   "      point [ "
   "         1.7265 0 0.618,    1 1 1, "
   "         0 0.618 1.7265,    0 -0.618 1.7265, "
   "         1 -1 1,    -1 -1 1, "
   "         -0.618 -1.7265 0,    0.618 -1.7265 0, "
   "         1 -1 -1,    1.7265 0 -0.618, "
   "         1 1 -1,    0.618 1.7265 0, "
   "         -0.618 1.7265 0,    -1 1 1, "
   "         -1.7265 0 0.618,    -1.7265 0 -0.618, "
   "         -1 -1 -1,    0 -0.618 -1.7265, "
   "         0 0.618 -1.7265,    -1 1 -1 "
   "      ] "
   "   } "
   "   IndexedFaceSet { "
   "      coordIndex [ "
   "         1, 2, 3, 4, 0, -1,  0, 9, 10, 11, 1, -1, "
   "         4, 7, 8, 9, 0, -1,  3, 5, 6, 7, 4, -1, "
   "         2, 13, 14, 5, 3, -1,  1, 11, 12, 13, 2, -1, "
   "         10, 18, 19, 12, 11, -1,  19, 15, 14, 13, 12, -1, "
   "         15, 16, 6, 5, 14, -1,  8, 7, 6, 16, 17, -1, "
   "         9, 8, 17, 18, 10, -1,  18, 17, 16, 15, 19, -1, "
   "      ] "
   "   } "
   "}";

// Routine to create a scene graph representing a dodecahedron
```

```
SoNode *
makeDodecahedron()
{
   // Read from the string.
   SoInput in;
   in.setBuffer(dodecahedron, strlen(dodecahedron));

   SoNode *result;
   SoDB::read(&in, result);

   return result;
}
```

第十二章 传感器

本章目标

在阅读完本章节后，我们将有能力做下列事情：

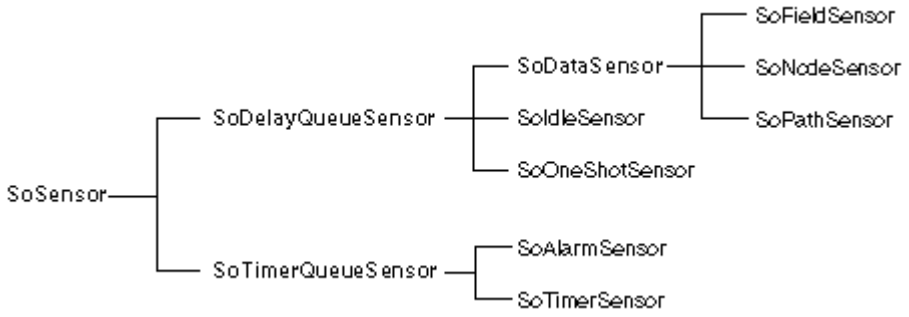
- 描述在场景中使用的不同类型的传感器，以及每种传感器的用法。
- 理解传感器在延迟队列和定时队列中是如何被调度的，以及何时被处理的。
- 可以为数据和定时传感器编写回调函数。
- 设置延迟队列传感器（delay-queue sensor）的优先级。

本章将描述怎样向场景增加传感器节点。传感器是一种 Inventor 对象，它可以监视多种类型的事件，在发生这些事件的时候，它可以调用用户提供的回调函数。传感器分成两大类：*数据传感器*，可以响应节点域、子节点、或路径中的数据变化；*定时传感器*，可以对某些调度条件做出响应。

传感器简介

传感器是一种比较特殊的对象类。它可以与Inventor的场景数据库相关联。它通过调用用户提供的回调函数来响应数据库的变化或某些时间事件。数据传感器（从**SoDataSensor**派生）会监视数据库中某些部分数据。当这些数据发生变化时，它会通知应用程序。定时传感器（像**SoAlarmSensor**和**SoTimerSensor**）会在某些类型的时间事件发生时通知应用程序。注意：这里的时间“事件”只发生在Inventor内部。它们不属于第 10 章所描述的事件模型。图 12-1 是传感器的部分类树图。

图 12-1 传感器类



传感器队列

如图 12-1 类树所暗示的那样，传感器可以放置在两种队列中：

- *定时队列*，当产生了一个定时警报或调度了定时传感器，将会调用传感器。
- *延迟队列*，只要 CPU 处于空闲状态（即 Inventor 没有事件和定时传感器需要处理），或者在用户定义的一个时间超时后，将会调用传感器。

当开始处理这两种队列时，队列中的传感器都将会按照顺序被处理一次（见“使用域传感器”）。

关键术语

下面将讨论数据和定时传感器所使用的一些新的术语：

- *触发*： 表示调用了用户自定义的回调函数，并将传感器从队列中移除。
- *调度*： 表示将传感器添加进队列中，以便在将来的某个时刻触发它。如果一个传感器已经被调度过了，那么再次调度它将不会产生任何效果。*取消调度* 表示将不会再处理这个传感器，并将它从队列中移除。
- *通告*： 表示让传感器知道它所关联的节点（域、或路径）发生了改变。数据传感器可以在接收到变化通告后自动调度自己。

数据传感器

Inventor 共有三种类型的数据传感器：

- **SoFieldSensor**， 与域相关联
- **SoNodeSensor**， 与节点相关联
- **SoPathSensor**， 与路径相关联

只要在某个特定域中的数据发生了变化，**SoFieldSensor**就会得到通告。当某一个节点内的数据发生了变化，或者位于这个节点下面的任意子节点内的数据发生了变化，或者当这个节点下的图形拓扑关系发生了变化时，**SoNodeSensor**都会得到通告。只要一个路径内的任意节点里的数据发生了变化，或者对路径增加或删除节点时，**SoPathSensor**都会得到通告。注意：这里的节点“*位于路径内*”的意思是指，只要遍历路径时，可以遍历到某个节点，那么这个节点就位于路径内。

提示：当为一个域设置的数据与当前域值相同时（例如 field.setValue(field.getValue()），这种设置也会被认为是修改了域值。还有，调用域和节点的**touch()**方法，也会被当作是做了一次修改。

渲染区对场景根节点关联了一个节点传感器，这样渲染区就可以探测出我们何时修改了场景。之后，它就会重新自动渲染一遍场景。

如果我们只是希望监视场景中某部分的数据变化，并且使这些部分与场景的其它部分互相“沟通”时，数据传感器还是非常有用的。例如，假设我们在场景中有一个材质正在与一个材质编辑器相关联，如果材质属性发生了变化，那么材质编辑器就需要更改其滚动条的位

置以反映新的材质属性。**SoNodeSensor**可以为材质编辑器支持这种反馈。

提示：域－域连接是另外一种场景中不同部分的数据保持同步的方法。见第 13 章

数据传感器的使用步骤：

下面列出的序列是创建数据传感器的必要步骤：

1. 构造传感器。
2. 设置回调函数。（见下节）
3. 设置传感器的优先级（见“优先级”）
4. 传感器关联到域、节点、或路径上。
5. 使用完传感器后，删除传感器。

回调函数

回调函数，如它的名字所暗示的那样，允许Inventor在某些预定义的事件发生时，对应用程序进行回调操作。回调函数通常都带有一个**void***类型的参数，这个参数用于将用户定义的数据传到回调函数中。传感器所使用的回调函数还带有另外一个**SoSensor***类型的参数。这个参数对于多个传感器使用同一个回调函数的情况是非常有用的。它将被设置成导致这次回调发生的传感器的指针（This argument is useful if the same callback function is used by more than one sensor. The argument is filled with a pointer to the sensor that caused the callback）。

在C++语言中，传感器的回调函数应该声明为类的静态函数。因为静态函数没有**this**的概念，所以我们需要显式地将希望修改的类实例作为用户自定义数据传入到静态函数中：

```
colorSensor->setData(this);
```

非静态 C++函数不适合做回调函数。

优先级

所有从**SoDelayQueueSensor**派生出来的传感器都使用优先级属性来维护其在延迟队列中的次序。下面的方法用于存取传感器的优先级属性：

setPriority(*priority*)	设置优先级。所有延迟队列传感器都有一个缺省值为 100 的优先级。传感器会根据优先级，在队列中按照从小到大的顺序排列。（**即优先级越小，传感器越先执行。译者注**）
getPriority()	获取传感器的优先级。
getDefaultPriority()	获取传感器的缺省优先级（100）

优先级为 0 的传感器具有最高的优先级。只要场景变化完，它马上就会被触发。如果两个传感器有相同的优先级，Inventor 不会保证某个传感器一定会先被触发。

SoXtRenderArea内部带有一个用来监控场景“重绘”的数据传感器，这个传感器的优先级为 10000。我们可以通过适当调整优先级的大小，在场景“重绘”之前或之后调度其它的传感器。例如，如果希望一个传感器就在场景“重绘”之前的那一刻被触发，可以按照下面的代码设置优先级：

```
SoNodeSensor   *s;
SoRenderArea   *renderArea;
s->setPriority(renderArea->getRedrawPriority() - 1);
```

触发数据传感器

当位于域、节点、或路径中的数据发生了变化，则会发生下列事件：

1. 传感器将得到一个数据已变化的通知。
2. 传感器被调度 — 根据传感器的优先级，将传感器添加进延迟队列中。
3. 在未来的某个时刻，Inventor 处理了这个队列，在队列中的所有传感器都被触发。
4. 当触发完毕后，传感器将从队列中被移除，同时传感器会调用它的回调函数。
5. 执行回调函数.回调函数可以访问到引起本次通告的触发域、节点和路径（见“使用触发节点和触发域”）。

使用域传感器

例 12-1 将演示把一个域传感器关联到场景照相机的**position**域上。每当照相机的位置发生改变时，回调函数便会将照相机的新位置打印出来。

例 12-1 关联域传感器

```
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/sensors/SoFieldSensor.h>

// Callback that reports whenever the viewer's position changes.
static void
cameraChangedCB(void *data, SoSensor *)
{
   SoCamera *viewerCamera = (SoCamera *)data;
```

```
   SbVec3f cameraPosition = viewerCamera->position.getValue();
   printf("Camera position: (%g,%g,%g)\n",
            cameraPosition[0], cameraPosition[1],
            cameraPosition[2]);
}

main(int argc, char **argv)
{
   if (argc != 2) {
      fprintf(stderr, "Usage: %s filename.iv\n", argv[0]);
      exit(1);
   }

   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoInput inputFile;
   if (inputFile.openFile(argv[1]) == FALSE) {
      fprintf(stderr, "Could not open file %s\n", argv[1]);
      exit(1);
   }

   SoSeparator *root = SoDB::readAll(&inputFile);
   root->ref();

   SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Camera Sensor");
   myViewer->show();

   // Get the camera from the viewer, and attach a
   // field sensor to its position field:
   SoCamera *camera = myViewer->getCamera();
   SoFieldSensor *mySensor =
            new SoFieldSensor(cameraChangedCB, camera);
   mySensor->attach(&camera->position);

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

使用触发节点和触发域（*高级内容*）

我们可以使用下列方法来获取到底是那个域、节点、或路径触发了数据传感器：

- getTriggerField()
- getTriggerNode()
- getTriggerPath()

只有优先级为 0 的传感器才能使用这些方法。

触发路径（*trigger path*）是一个节点链，它起始于最后被通告的那个节点，向下直到产生通告的那个节点为止（The *trigger path* is the chain of nodes from the last node notified down to the node that initiated notification）。因为保存触发路径信息需要很多的资源，所以要想获得触发路径，我们必须首先调用**setTriggerPathFlag()**，将触发路径标志位设置为TRUE。而且，我们还必须要在传感器被通告之前调用此方法。否则，触发路径的信息将不会被保存，**getTriggerPath()**永远都返回NULL。（缺省情况下，触发路径标志位为FALSE）。与此形成对照的是，触发节点和触发域总是有效的。注意：如果变化不是因为修改域数据而产生的（例如，调用了**addChild()**或**touch()**），那么**getTriggerField()**将返回NULL。

例 12-2 将演示在回调函数中使用**getTriggerNode()**和**getTriggerField()**的方法。只要场景发生了变化，回调函数都会向屏幕输出一条关于引起变化的节点的信息。

例 12-2 使用触发节点和触发域

```
#include <Inventor/SoDB.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/sensors/SoNodeSensor.h>

// Sensor callback function:
static void
rootChangedCB(void *, SoSensor *s)
{
   // We know the sensor is really a data sensor:
   SoDataSensor *mySensor = (SoDataSensor *)s;

   SoNode *changedNode = mySensor->getTriggerNode();
   SoField *changedField = mySensor->getTriggerField();

   printf("The node named '%s' changed\n",
            changedNode->getName().getString());

   if (changedField != NULL) {
```

```
      SbName fieldName;
      changedNode->getFieldName(changedField, fieldName);
      printf(" (field %s)\n", fieldName.getString());
   }
   else
      printf(" (no fields changed)\n");
}

main(int, char **)
{
   SoDB::init();

   SoSeparator *root = new SoSeparator;
   root->ref();
   root->setName("Root");

   SoCube *myCube = new SoCube;
   root->addChild(myCube);
   myCube->setName("MyCube");

   SoSphere *mySphere = new SoSphere;
   root->addChild(mySphere);
   mySphere->setName("MySphere");

   SoNodeSensor *mySensor = new SoNodeSensor;

   mySensor->setPriority(0);
   mySensor->setFunction(rootChangedCB);
   mySensor->attach(root);

   // Now, make a few changes to the scene graph; the sensor's
   // callback function will be called immediately after each
   // change.
   myCube->width = 1.0;
   myCube->height = 2.0;
   mySphere->radius = 3.0;
   root->removeChild(mySphere);
}
```

其它的延迟队列传感器

除了数据传感器外，还有两类传感器可以添加进延迟队列中：**SoOneShotSensor**和**SoIdleSensor**。

单触发传感器和空闲传感器的使用步骤

下面是创建单触发传感器和空闲传感器的必要步骤：

1. 构造传感器。
2. 设置回调函数（见“回调函数”）。
3. 设置优先级（见“优先级”）。
4. 使用**schedule()**方法调度传感器。
5. 使用完毕后，删除传感器。

注意：这两种传感器必须要通过程序代码显式地调度它们（**其它传感器是Inventor自己触发调度的，而这两种传感器必须要在程序代码里通过调用schedule()方法来实现调度。译者注**）。使用**unschedule()**方法可用将这两种传感器从队列中移除。

SoOneShotSensor

无论何时，只要Inventor开始处理延迟队列，**SoOneShotSensor**都会调用一次它的回调函数。这种传感器对于那些不需要立即执行的或不必立即执行（可能这些任务非常耗时）的任务来说是非常有用的。例如，当要处理一个快速设备（例如像鼠标设备）所产生的事件时，我们希望可以快速地处理每个事件，这样可以避免事件堵塞队列。但是，如果我们知道处理某种事件是非常耗时的，那么我们就可以使用单触发传感器来调度这个事件。例如：

```
handleEvent(SoHandleEventAction *action)
{
   //Check for correct event type ...
   .
   .
   .
   // Remember information from event for later processing
   currentMousePosition = event->getPosition();

   // Schedule a one-shot sensor to do hard work later
   SoOneShotSensor oneShot = new SoOneShotSensor(
         OneShotTriggerCallback, NULL);
   oneShot->schedule();
}
void OneShotTriggerCallback(void *userData, SoSensor *)
{
   // Do lengthy operation based on current mouse position;
}
```

注意：像**SoOneShotSensor**、**SoIdleSensor**、和**SoAlarmSensor**这些传感器只会对它们的

回调函数调用一次，在执行完回调函数后，它们仍然可以继续存在，但只有在下次它们被调度后，它们才会再次触发回调函数。当这些传感器被调度时，可以使用**unschedule()**方法来终止传感器调用它们回调函数。

下面的代码使用了**SoOneShotSensor**传感器来延迟渲染场景，直到CPU处于空闲状态的时候才开始渲染。

```
SoOneShotSensor *renderTask;

main() {
   ...
   renderTask = new SoOneShotSensor(doRenderCallback, NULL);
   // ... set up events, UI, which will call changeScene()
   // routine.
}

void
changeScene()
{
   // ... change scene graph ...
   renderTask->schedule();
}

void
doRenderCallback(void *userData, SoSensor *)
{
   // ... does rendering ...
}
```

SoIdleSensor

只有应用程序处于空闲状态的时候（没有事件和定时器等待处理），**SoIdleSensor**才会会调用一次回调函数。可以为低优先级的任务使用空闲传感器，只有在无其它事情需要做的时候才会处理这个任务。如果我们希望它可以重复触发（但请注意，这会使CPU总是处于“忙”状态），我们就需要在传感器的回调函数中调用**schedule()**方法。注意，如果其它事件和定时器发生的太频繁，以致于程序根本就没有空闲时间，那么可能会导致空闲传感器永远不会被处理。详情见“传感器队列的处理过程”。

定时队列传感器（Timer－Queue Sensors）

与数据传感器一样，定时队列传感器也可以调用用户定义的回调函数。不同的是，我们不用将定时队列传感器关联到场景中的节点或路径上，我们只需要简单地调度它，Inventor就会在指定的时刻调用其回调函数。(定时队列传感器在定时队列中是按照时间的先后而不

是优先级的大小来排序的)，Inventor包含有两种定时队列传感器：**SoAlarmSensor**和**SoTimerSensor**。

定时队列传感器的使用步骤

1. 构造传感器
2. 设置回调函数（见“回调函数”）
3. 设置传感器的时间参数
4. 使用**schedule()**方法调度传感器
5. 使用完毕后，删除传感器

当一个定时队列传感器被调度后，就不能再修改其时间参数了（时间参数可以指定传感器何时被触发，以及以种频率被触发）。如果需要重新设置传感器的时间参数，只有先使用**unschedule()**方法将传感器从队列中移除，然后再修改时间参数，最后再次调度传感器。

SoAlarmSensor

SoAlarmSensor传感器就像一个闹钟，它也可以设置一个指定的时间。当到达或超过了指定的时间后，它的回调函数就会被调用（**出现传感器在超过了指定的时间后才被触发的原因是：如果在到达指定时间的时候，CPU可能正在忙着处理其它事情。当CPU处理完其它事情后再处理定时传感器时，可能处理的时间已经超过了指定的时间。因此定时队列传感器不能在时间上保证会精确触发。译者注**）。可以在处理日程安排方面的程序中使用**SoAlarmSensor**，例如，可以在程序中某个时刻上放置一个标记，以此来提醒我们参加下午 2 点的会议。

使用下列方法设置传感器的时间参数：

setTime(*time*)　　设置在 *time* 时刻调度触发传感器

setTimeFromNow(*time*) 设置从现在开始，经过了 *time* 时间后调度触发传感器

可以使用**SbTime**类来定义*time*参数，**SbTime**类可以提供多种时间格式。使用**SoAlarmSensor**的**getTime()**方法可以得到传感器的时间参数。

例 12-3 将演示如何在经过一分钟后，使用**SoAlarmSensor**升起屏幕中的一面旗帜（**using an SoAlarmSensor to raise a flag on the screen when one minute has passed。英文原版中说例子会升起一面旗帜，但实际运行 12-3 的例子时，屏幕上只显示了一个圆锥体。读者可以自己运行例子来查看效果。译者注**）。

例 12-3 使用警报传感器

```
static void
raiseFlagCallback(void *data, SoSensor *)
{
   // We know data is really a SoTransform node:
   SoTransform *flagAngleXform = (SoTransform *)data;
```

```
   // Rotate flag by 90 degrees about the z axis:
   flagAngleXform->rotation.setValue(SbVec3f(0,0,1), M_PI/2);
}

{
   ...

   SoTransform *flagXform = new SoTransform;

   // Create an alarm that will call the flag-raising callback:
   SoAlarmSensor *myAlarm =
       new SoAlarmSensor(raiseFlagCallback, flagXform);
   myAlarm->setTimeFromNow(60.0);
   myAlarm->schedule();
}
```

SoTimerSensor

SoTimerSensor与SoAlarmSensor是非常相似的，不同之处是SoTimerSensor需要设置的是时间间隔参数－就像闹钟上的定时唤醒开关。我们可以使用**SoTimerSensor**传感器来产生动画效果。例如，转动屏幕上闹钟的秒针。使用下面的方法可以设置**SoTimerSensor**的间隔时间和基准时间。

setInterval(*interval*)	在给定的 *interval* 时间间隔内调度一次传感器，例如，每分钟调度一次。缺省的时间间隔为 1/30 秒
setBaseTime(*time*)	以给定的 *time* 时间为基准时间来调度传感器。缺省的基准时间是现在－即当传感器第一次被调度时（schedules a sensor to occur starting at a given *time*. The default base time is right now—that is, when the sensor is first scheduled.）

在修改间隔时间和基准时间之前，我们必须首先取消调度传感器。如例 12-4 中所示。

例 12-4 创建了两个定时传感器。第一个传感器用于旋转物体。第二个传感器会每 5 秒触发一次，来修改旋转传感器的定时间隔时间。旋转传感器轮流使用每秒 1 次和每秒 10 次两种时间间隔。（这个例子主要用于演示目的。如果使用两个引擎（见第 13 章）会更好也更容易实现相同的效果）

例 12-4 使用定时传感器

```
// This function is called either 10 times/second or once every
// second; the scheduling changes every 5 seconds (see below):
static void
```

```
rotatingSensorCallback(void *data, SoSensor *)
{
   // Rotate an object...
   SoRotation *myRotation = (SoRotation *)data;
   SbRotation currentRotation = myRotation->rotation.getValue();
   currentRotation = SbRotation(SbVec3f(0,0,1), M_PI/90.0) *
            currentRotation;
   myRotation->rotation.setValue(currentRotation);
}

// This function is called once every 5 seconds, and
// reschedules the other sensor.
static void
schedulingSensorCallback(void *data, SoSensor *)
{
   SoTimerSensor *rotatingSensor = (SoTimerSensor *)data;
   rotatingSensor->unschedule();
   if (rotatingSensor->getInterval() == 1.0)
      rotatingSensor->setInterval(1.0/10.0);
   else
      rotatingSensor->setInterval(1.0);
      rotatingSensor->schedule();
}

{
   ...

   SoRotation *myRotation = new SoRotation;
   root->addChild(myRotation);

   SoTimerSensor *rotatingSensor =
      new SoTimerSensor(rotatingSensorCallback, myRotation);
   rotatingSensor->setInterval(1.0); //scheduled once per second
   rotatingSensor->schedule();

   SoTimerSensor *schedulingSensor =
   new SoTimerSensor(schedulingSensorCallback, rotatingSensor);
   schedulingSensor->setInterval(5.0); // once per 5 seconds
   schedulingSensor->schedule();
}
```

传感器队列的处理过程（*高级内容*）

下面的讨论内容只针对于使用了 Inventor Xt 组件库的应用程序。其它窗口系统工具包在处理不同队列的问题上也许有不同的关系（Other window system toolkits may have a different relationship between processing of the different queues）。读者如果对定时器是如何被调度的技术细节不感兴趣的话，可以跳过本节内容。

Inventor 处理队列的顺序是：事件队列，定时器队列、延迟队列。有时可能会发生稍微不同于上面处理顺序的情况，因为无论当前是否存在有未处理的定时器和事件，延迟队列都可能在处理等待时间超时的时候，先被强制处理掉。处理顺序如下：

SoXt的消息主循环调用XtAppMainLoop：

```
BEGIN:
If there's an event waiting:
Process all pending timers.
Process the delay queue if the delay queue time-out is
         reached.
Process the event.
Go back to BEGIN.
else (no event waiting)
if there are timers,
   Process timers.
   Go back to BEGIN.
else (no timers or events pending)
Process delay queue.
Go back to BEGIN.
```

当处理定时器队列时，下面的情况可以得到保证：

- 所有被调度的定时传感器和警报传感器在启动（go off）之前，或者在队列处理结束后，Inventor 会按顺序触发它们（All timer or alarm sensors that are scheduled to go off before or at the time processing of the queue ends are triggered, in order）。
- 当定时传感器被重新调度时，它们将设置相同的重新调度时间，这个时间就是所有的定时传感器被触发完之后的时间（When timer sensors are rescheduled, they are all rescheduled at the same time, after they have all been triggered）。

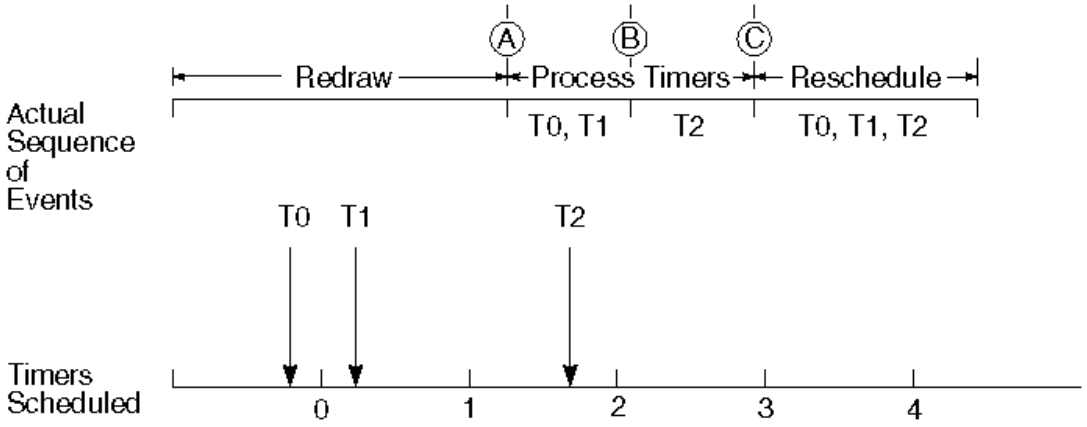
例如，在图 12-2 中，在场景重绘之后的 *A* 时刻，Inventor 开始处理定时器队列。在队列中有三个被调度的定时器（timer0、1、2）。其中 Time0 和 Time1 已经准备启动了（go off）（即当前时刻已经超过了它们的触发时间），Timer2 将在以后的某个时刻启动。那么处理的顺序如下：

1. Timer0 被触发
2. Time1 被触发
3. Inventor 检查定时器队列（此时是时刻 *B*），注意到此时 Timer2 的触发时间也超时

了，所以 Inventor 会触发 Timer2

4. Timer1、2、和 3 将在时刻 *C* 处被重新调度。
5. Inventor 返回主循环，继续检查未处理事件。

图 12-2 触发和重调度定时器



当没有事件和定时器传感器需要处理或者当延迟队列内部的time-out时间已经超时，Inventor就要开始处理延迟队列。缺省情况下，延迟队列每秒超时 30 次。我们可以使用**SoDB::setDelaySensorTimeout**方法来修改这个时间间隔。当是因为延迟传感器内部处理时间超时而导致了延迟队列被处理的时候，空闲传感器将被忽略掉。(因为在这种情况下，CPU 并没有处于空闲状态）.(**延迟传感器只要是程序处于空闲状态时，就可以被触发。但假设程序要连续处理多个事件，从而导致了很长一段时间内程序没有处于空闲状态，那么延迟传感器就不会及时地被触发，setDelaySensorTimeout的作用是设置延迟传感器一个内部的处理超时时间，当程序如果超过了这个内部处理时间后仍然没有处于空闲状态时，Inventor将会强制处理一次延迟传感器。这样就可以保证程序的某个事件可以及时被处理。译者注**)

当处理延迟队列时，下面的情况可以得到保证：

- 所有位于延迟队列中的传感器，将按照优先级的顺序，依次被触发。
- 每个传感器只被触发一次，不管传感器是否又重新调度了自己.

第十三章 引擎

本章目标

在阅读完本章节后，我们将有能力做下列事情：

- 实现场景中域与域之间的连接
- 在数据库中创建并使用全局域。
- 实现引擎与域以及引擎与引擎之间的连接。
- 临时禁用引擎连接。
- 创建一个简单的引擎网络（engine network）
- 使用引擎驱动场景
- 使用转体节点（rotor）、频闪节点（blinker）、开关节点（shutter）、摇摆节点（pendulum）来驱动场景。

本章所要描述的引擎对象是一种 Inventor 对象，它可以和其它域进行连接，用来驱动场景运动或强迫场景中的某些物体按照设定的规则运动。本章还将描述域与引擎，以及域和域之间是如何连接的。以及怎样在数据库中创建并使用全局域。

引擎简介

在以前的章节中，我们已经可以使用能对用户事件产生反应的 3D 对象来创建图形场景。这些 3D 对象本身是固定不变的，它们只能通过响应用户的交互式操作或受到传感器的驱动才能进行运动。在本章，我们将学习使用一种叫做“*引擎*”的新对象，引擎可以将运动和几何物体压缩进一个单一的场景中。就像在现实世界中，我们可以将发动机连接到其它设备上来驱动转轮或风扇那样，我们也可以将引擎对象“连接”进场景数据库中，来驱动场景运动或产生可以体现物体之间复杂关系的行为。引擎也可以和其它引擎相互连接，这样它们之间可以相互驱动或相互反应，最终使场景数据库发生改变。

举一个简单的例子，假设有一个用来描述风车模型的场景，我们可以对场景附加上一个引擎对象，这个引擎用来描述风车叶片的旋转并定时增加叶片的旋转角度（You can attach an engine object that describes the rotation of the windmill blades and performs an incremental rotation of the blades in response to time）。这个场景，包括引擎节点在内，都可以保存到一个 Inventor 文件中。无论何时，只要读入了此场景文件，场景中就会出现一个风车模型，并且风车的叶片还是不停旋转的。几何模型和运动行为都可以使用场景中的节点和引擎来描述。

另外一个比较复杂的例子是让两个对象相互缠绕（wiring）在一起。例如，我们可以创建一个场景，场景中有一个卡车。一个引擎对象用来驱动卡车的运动。另一个引擎对象可以根据卡车运动轨迹的信息来驱动照相机运动，这样照相机将可以跟随运动的卡车漫游场景。或者我们也可以将两个引擎连接起来，这样一个引擎就可以影响另外一个引擎的行为。在上面风车的例子中，我们还可以将另外一个引擎连接到旋转引擎的前面，使第二个引擎能过过滤第一个引擎的时间，让风车只能在早上 9 点至晚上 5 点这段时间内旋转叶片。

在某些情况下，我们即可以使用传感器，也可以使用引擎来实现同一种效果。表 13-1 比较了传感器与引擎之间的差别，可以帮助我们更好地选择适合自己的方法。

表 13-1 传感器和引擎的对照表

传感器	引擎
是程序代码的一部分（不能输出到文件中）	是场景的一部分（可以读入或输出到文件中）
使用用户定义的回调函数	使用内建的函数
可以明确地控制触发的顺序	自动评估，不能手工控制
可以关联到任意类型的域上（域数据传感器）	只有固定类型的输入和输出
可以影响场景外的对象	只能影响到场景内的其它节点和引擎

引擎的一般用途

引擎通常用于两个方面：

- 驱动一部分场景（To animate parts of a scene）
- 用来维持场景中的某部分物体与另外物体之间的关系（To constrain one part of a scene in relation to some other part of the scene）

图 13-1 和 13-2 展示了两个使用引擎的应用程序。在图 13-1 中，有四种不同类型的连接件－拉杆、铰链、曲柄、和双拉杆。引擎用于将这些连接件首尾相连。在图 13-2 中的物体使用引擎来编辑修改几何变换节点，来驱动场景中的动物和其它物体。

图 13-1 使用一组连接件制造的机器装置

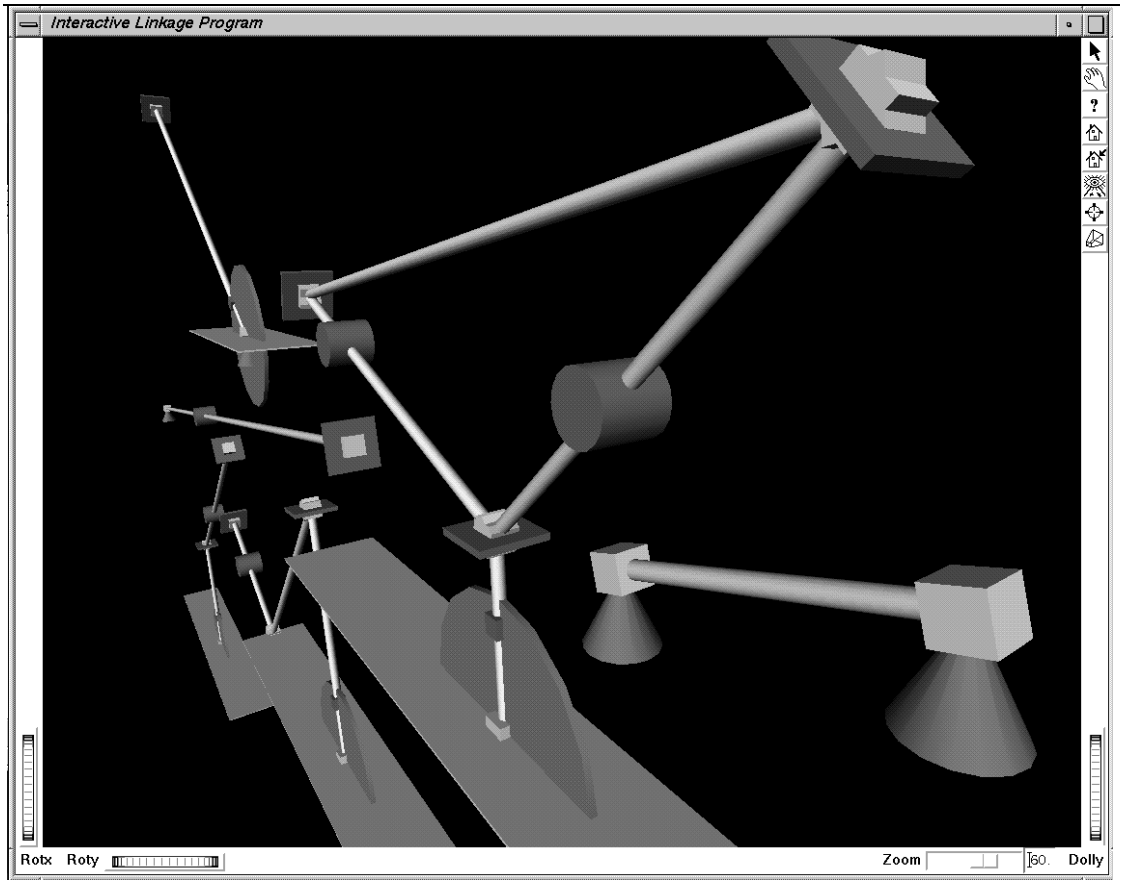
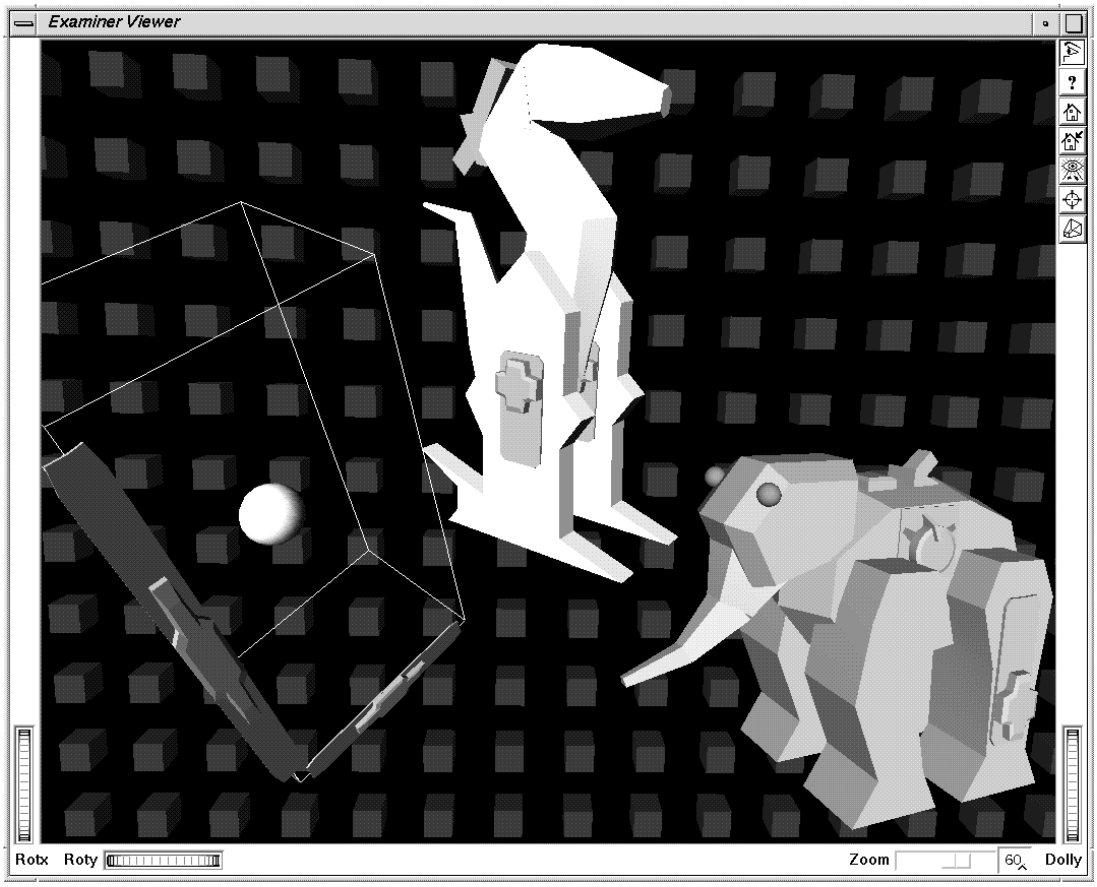


图 13-2 使用引擎来驱动和定位物体

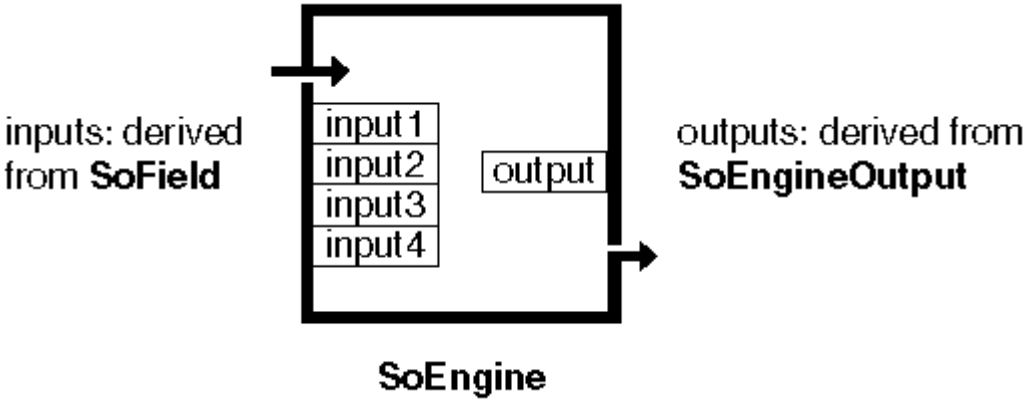


如在本章后面给出的例 13-1 中所示那样，我们可以将部分场景与一个定时器相连接，这样当定时器的时间发生了改变后，引擎就可以驱动场景运动起来。例 13-6 是一个引擎约束的例子，这个例子利用引擎将物体的运动轨迹限制一个设定的路径上。

我们可以将引擎想象成一个黑盒子，它可以接收输入数据，对这些数据执行一些操作，然后将结果拷贝到它的一个或多个输出中。引擎的输入和输出接口都可以连接到场景中其它的域或引擎上。当引擎的输出值发生了改变，这些新的输出值将会被发送到与之相连的任何域或引擎中。

在图 13-3 中，引擎带有从**SoField**派生出来的输入接口和从**SoEngineOutput**派生出来的输出接口。当引擎的输入值发生了变化，引擎就会使用一个内建的函数来计算新的输出值。结果输出值将被发送到与之相连的所有域和引擎中。因为**SoEngine**是从**SoBase**派生出来的，所以引擎包含有读写文件的方法。

图 13-3 引擎解剖图



例如，图 13-3 中的引擎代表的是**SoComposeVec4f**节点，**SoComposeVec4f**节点是一种内部创建了**SoMFVec4f**对象的引擎节点。它有 4 个类型为**SoMFFloat**的输入和 1 个类型为**SoMFVec4f**的输出。这种引擎可以将 4 个输入排列到一个**SoMFVec4f**输出中。

引擎类型

图 13-4 显示了引擎的类树图。类树图中按照引擎执行的操作类型的不同而分成了若干个组。

下面是算法类引擎：

- SoCalculator
- SoBoolOperation
- SoInterpolateFloat, SoInterpolateRotation, SoInterpolateVec2f, SoInterpolateVec3f, SoInterpolateVec4f
- SoTransformVec3f
- SoComposeVec2f, SoDecomposeVec2f
  SoComposeVec3f, SoDecomposeVec3f
  SoComposeVec4f, SoDecomposeVec4f
  SoComposeRotation, SoDecomposeRotation
  SoComposeMatrix, SoDecomposeMatrix
- SoComputeBoundingBox

动画驱动类引擎：

- SoElapsedTime
- SoOneShot
- SoTimeCounter

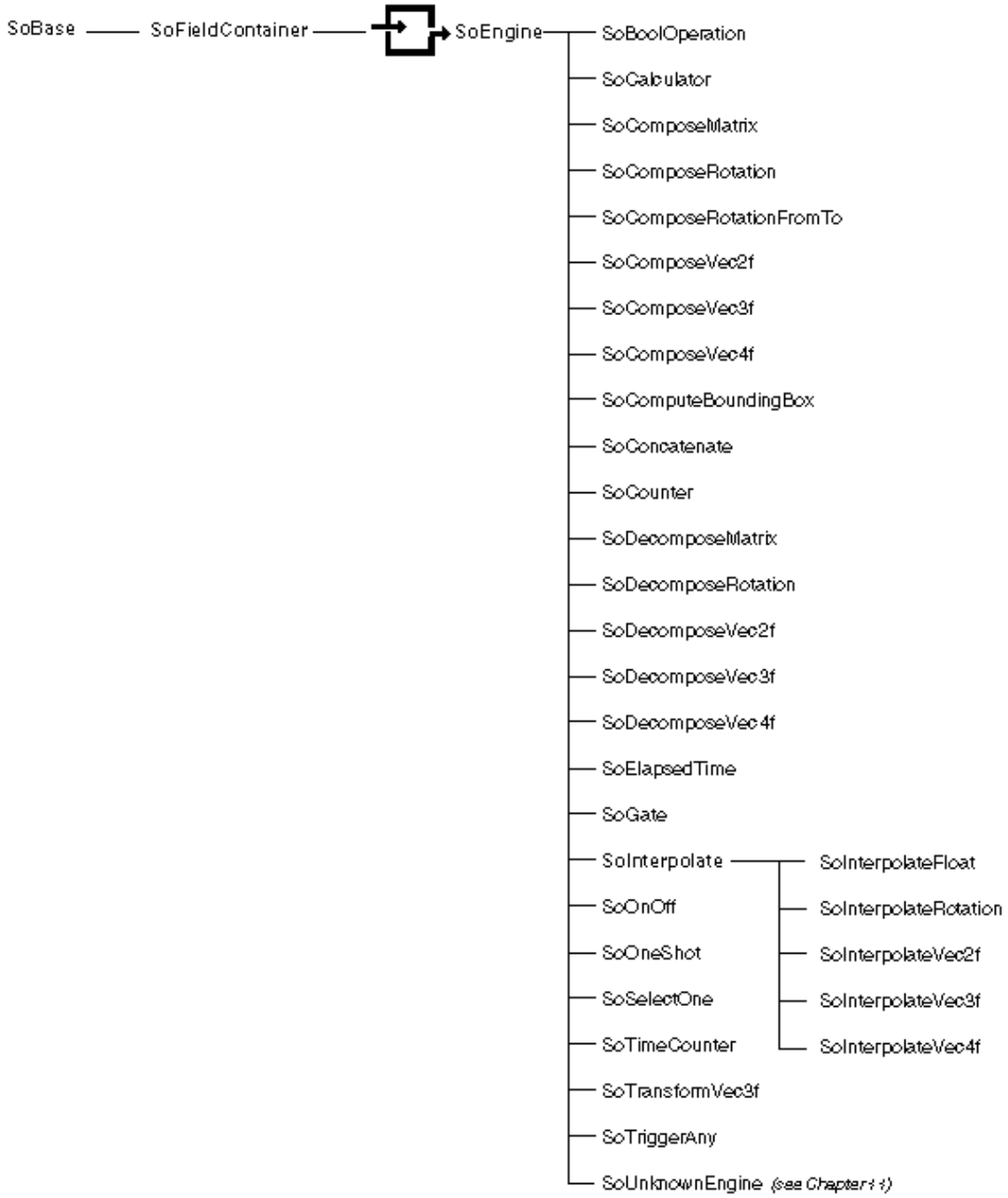
触发类引擎：

- SoCounter
- SoOnOff
- SoTriggerAny
- SoGate

数组处理类引擎：

- SoSelectOne
- SoConcatenate

图 13-4 引擎的类树



域连接

使用**SoField**的**connectFrom()**方法，可以将一个域连接到其它的域或引擎上。当我们连接不同类型的域时，输入域的值会自动转换成新的输出域的类型。下面是连接一个域的语法：

void connectFrom(SoField *field);

void connectFrom(SoEngineOutput *engineOutput);

例如，将**SoPerspective**照相机的**orientation**域连接到**SoTransform**的**rotation**域上：

```
xform->rotation.connectFrom(&pCamera->orientation);
```

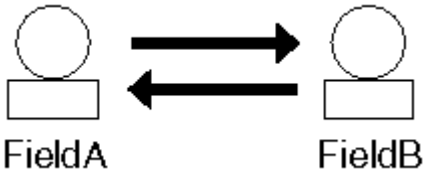
（这里照相机的**orientation**域是源域，**SoTransform**的**rotation**域是目的域，当源域发生了变化，目的域的也做相应的变化。译者注）

将**SoElapsedTime**引擎连接到**SoText3**节点的**string**域上：

```
yourText->string.connectFrom(&elapsedTime->timeOut);
```

假设我们按照图 13-5 的样式连接了两个域。在这个例子中，上面的箭头指出*域 A* 是源域，*域 B* 是目的域。下面的箭头指出*域 B* 是源域，*域 A* 是目的域。当我们建立了这种连接关系后，只要我们修改了*域 A*，那么*域 B* 也会相应做修改。同样如果我们修改了*域 B*，*域 A* 也会做修改。我们不必担心建立了一个“死循环”，两个域在互相不间断地更新对方。事实上，当*域 A* 发生变化，*域 B* 也会发生变化，这时，*域 A* 知道自己已经发生过变化了，所以它不会再此修改自己（即 Inventor 会避免出现这种“死循环”现象的发生。译者注）。

图 13-5 域－域连接

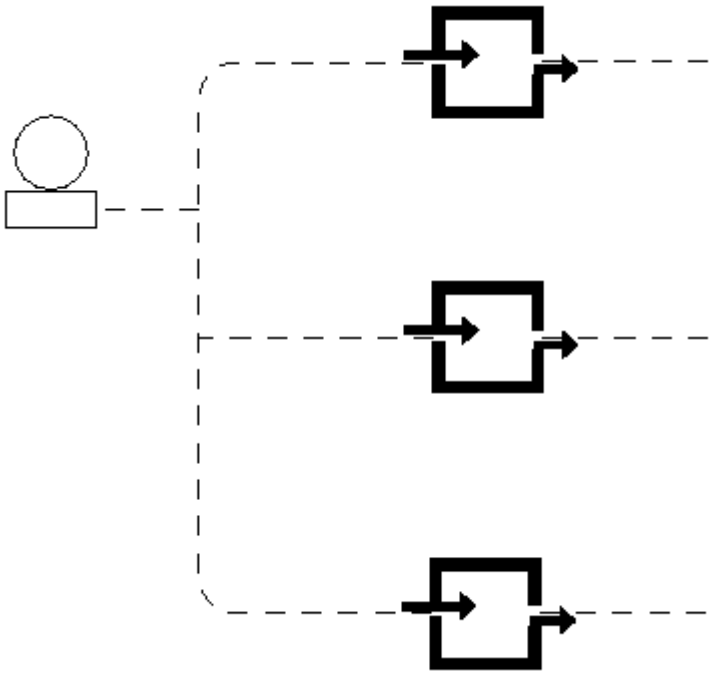


使用**disconnect()**方法可以断开一个目标域上的连接。使用**isConnected()**方法可以查询当前域上是否存在着一个连接。连接到其它域或引擎上的域可以调用像**setValue()**这样的方法。无论是谁最后设置了域值，谁就可以调用**setValue()**。（Methods such as **setValue()** can also be called on a field that is connected from another field or engine. Whoever sets the field value last, wins）

多重连接

“*引擎网络（engine network）*”这个术语是指场景中相互缠绕连接在一起的引擎和域的集合。当要计划设计一个关系复杂的引擎网络时，有时我们需要使用具有多重连接的域或引擎设计。下面给出一个多重连接的规则：一个给定的域或引擎，只能有一个输入连接，但可以有多个输出连接。图 13-6 举例说明了这个规则。

图 13-6 允许多个输出连接



如果在我们调用一个域或引擎的**connectFrom()**方法时，先前这个域或引擎已经被连接到另一个不同的源域上，那么Inventor会打断原来的连接，而建立起新的连接。

域的转换

当我们连接不同类型的域时，Inventor 会自动地将域值从一种类型转换成另外一种类型。它会在必要的时候执行下列形式的类型转换：

- 任意域类型转换到**String**字符串类型
- **String**字符串类型转换到任意类型
- 在任意两个**Bool**, **Float**, **Long**, **Short**, **ULong**, **UShort**之间相互转换
- 在**Color**和**Vec3f**之间相互转换
- 在**Float**和**Time**之间相互转换
- 在**Matrix**和**Rotation**之间相互转换
- 在**Name**和**Enum**之间相互转换
- 在**Rotation**和**Vec4f**（四元数）之间相互转换
- 在域的**MF**形式与相应的**SF**形式上相互转换。

Inventor不支持多步（Multiple-step）转换, 也就是说，虽然我们可以直接将**Vec4f**转换成**Rotatio**n，然后再将**Rotatio**n转换成**Matrix**。但是我们不能直接将**Vec4f**转换成**Matrix**。

如果我们的应用程序试图将一个域连接到另外一个输出不同类型的域或引擎上，而此时不存在这两种类型的转换，那么我们将无法建立这个连接，并且会产生一个调试错误。在 *The Inventor Toolmaker* 书中详细讨论了如何创建用户自己的域转换器。

引用计数

引擎的引用计数和前面章节中所讨论的节点的引用计数非常相似。域－域之间的连接，包括引擎的输入和其它域之间的连接，这些类型的连接都不会增加引用计数。但引擎的输出域和其它域之间的连接将会使引擎的引用计数增加 1。同样，打断引擎的输出域连接则会使引擎的引用计数减 1。如果引擎的最后一个连接也被打断了，引擎的引用计数将变成 0，这时 Inventor 就会删除掉这个引擎节点。如果我们希望在打断连接的时候，防止引擎不会被意外删除，可以在创建引擎后立即引用一次引擎。同时需要注意的是，对于一个域－域连接来说，当删除了包含连接域的节点或引擎时，这个域－域连接也会被打断。进而可能会导致连接的引擎被删除（**当一个节点的域和一个引擎相连接时，节点和引擎的引用计数假设为 1，那么当删除了节点后，这个连接也会自动被打断，而连接被打断就会导致引擎的引用计数减 1，最终导致引擎被删除。译者注**）。

禁用连接

对目标域调用**enableConnection**(FALSE)，或对引擎的输出域调用**enable**(FALSE)，可以使域连接暂时失效。

当我们希望临时禁用使用某个引擎网络的时候，这种方法是很有效的。如果不使用这种方法，而是通过打断域和引擎之间的连接来禁用引擎网络，那么引擎有可能会因解除引用而导致引用计数值变为 0，最终可能会被错误地删除。禁用域连接的方法不会影响到引擎的引用计数值。可以使用**isConnectionEnabled()**方法查询一个连接是否处于可用状态。

更新数据

当我们修改引擎网络中的一个域数据时，我们可以假定所有依赖于这个数据的其它域数据都会立即得到更新。但事实上，考虑到程序运行的效率原因，Inventor 只是将其它域数据标记为过期数据，只有当开始使用这些域数据的时候，Inventor 才会去更新它们。例如，一个非常复杂的引擎网络可能会连接到一个在 SoSwitch 组中未被选择的子节点上，而且假设场景从来就不使用这个引擎网络。这时如果修改了子节点的数值，那么引擎网络的数据将被标记为需要更新，但事实上这些数据从未被更新过，这是因为引擎网络从来就没有被遍历使用过。

有些引擎，例如像闸门引擎（gate engine）和动画引擎（animation engine），当它们的数值更新后，它们可以做一些选择性的控制操作。这些引擎中，多数引擎都带有一个**SoSFTrigger**类型的域。这种域只有在域被“触动”（touched）时，才会将输出数据更新一次。详见“闸门引擎”章节

全局域

全局域也是Inventor数据库中的一种域。我们可以通过名称来访问它。它不属于任何特

定的节点或引擎。Inventor提供了一个内建的全局域：**realTime**，这个域的类型为**SoSFTime**。**realTime**域记录着系统的当前时间，通过将它连接到其它的域或引擎上，我们可以创建出一个基于时间驱动的动画。我们可以根据需要创建额外的全局域。例如，假设我们要开发一个关键帧动画编辑器的应用程序，我们也许会希望创建一个“当前帧”域，这个域可以连接到多个引擎上。一旦创建了这个域，我们就可以使用标准的域的方法，将它与场景中的其它部分连接起来。

使用**SoDB**的**createGlobalField()**方法可以创建一个全局域：

static SoField ***SoDB::createGlobalField**(const SbName &*name*, SoType *type*);

在 Inventor 数据库中，一个的名字只能对应一个全局域。如果已经存在了一个给定名字和类型的全局域，那么函数就直接返回那个已经存在的全局域。如果已经存在了一个给定名字，但类型不兼容的全局域时，函数将返回 NULL。

getGlobalField()方法将根据给定的名字参数，返回一个全局域。

static SoField ***SoDB::getGlobalField**(const SbName &*name*);

可以使用域的**getTypeId()**方法来检查返回域的类型。例如：

```
if (globalField->isOfType(SoSFFloat::getClassTypeId()) ...
```

下面是使用**realTime**全局域的一个例子：

```
engineA->input1.connectFrom(SoDB::getGlobalField("realTime"));
```

例 13-1 创建了一个电子表，例子是将一个**SoText3**文本节点连接到**realTime**全局域上。图 13-7 显示了这个例子的场景图。图 13-8 显示了例子运行的结果。

例 13-1 使用实时全局域

```
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoText3.h>
```

图 13-7 电子表例子的场景图

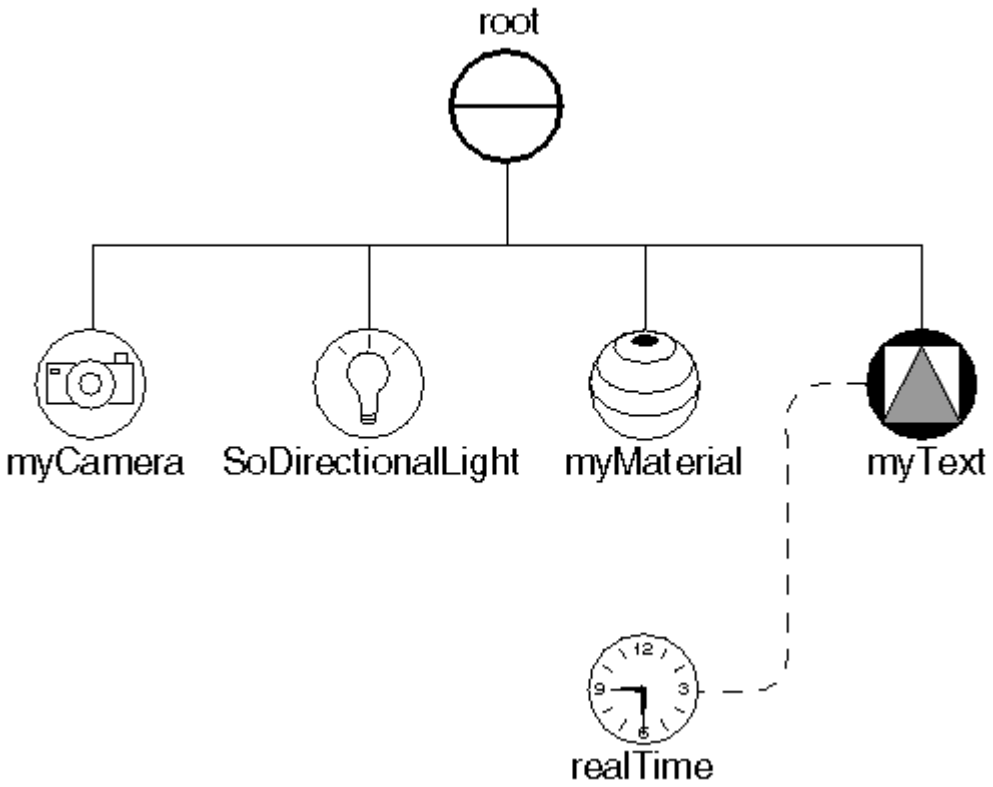
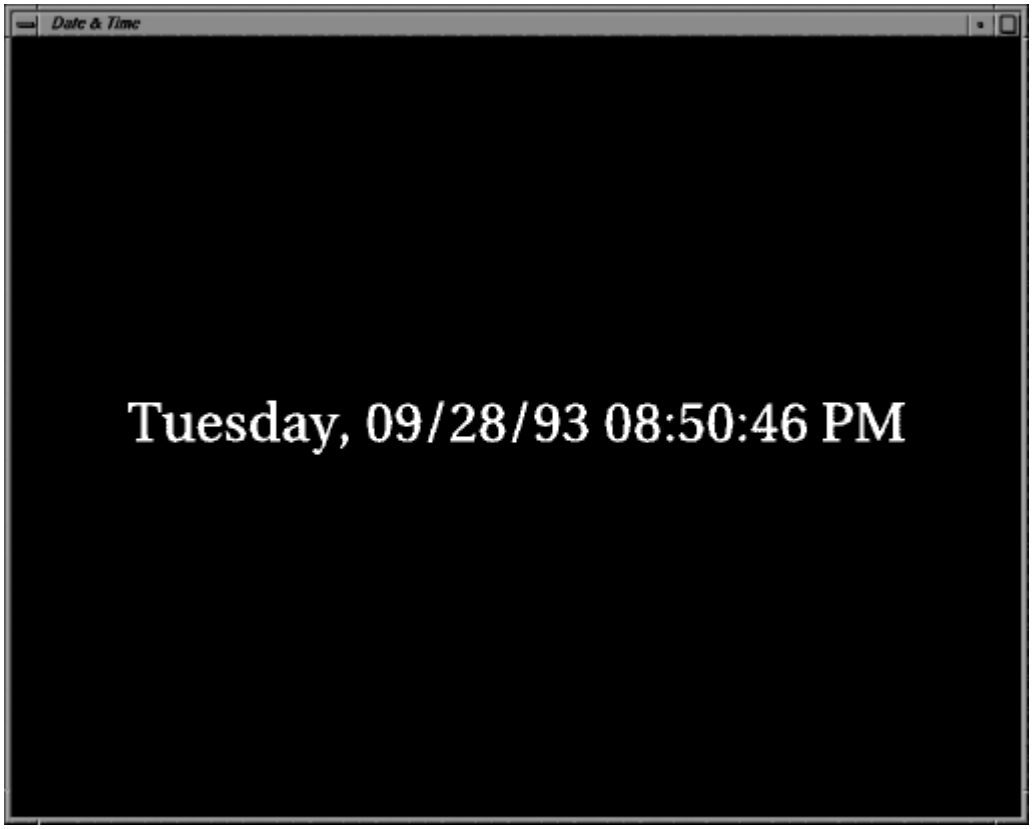


图 13-8 使用实时全局域的电子表



```
main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL)
      exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   // Add a camera, light, and material
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   SoMaterial *myMaterial = new SoMaterial;
   myMaterial->diffuseColor.setValue(1.0, 0.0, 0.0);
   root->addChild(myMaterial);

   // Create a Text3 object, and connect to the realTime field
   SoText3 *myText = new SoText3;
   root->addChild(myText);
   myText->string.connectFrom(SoDB::getGlobalField("realTime"));

   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
   myCamera->viewAll(root, myRenderArea->getSize());
   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Date & Time");
   myRenderArea->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

动画引擎

下面讨论的引擎可被用于驱动场景中的物体。这些引擎都包含有一个**timeIn**域，这个域在构造引擎对象的时候，就已经自动和**realTime**全局域连接上了。当然，这个域也可以连接到任何其它的时间源上。

- **SoElapsedTime** — 功能类似秒表；输出从开始运行时到现在已经过去的时间。
- **SoOneShot** — 运行了一段指定的时间后，引擎停止运行。
- **SoTimeCounter** — 以给定的频率，在最小与最大计数之间循环运行。

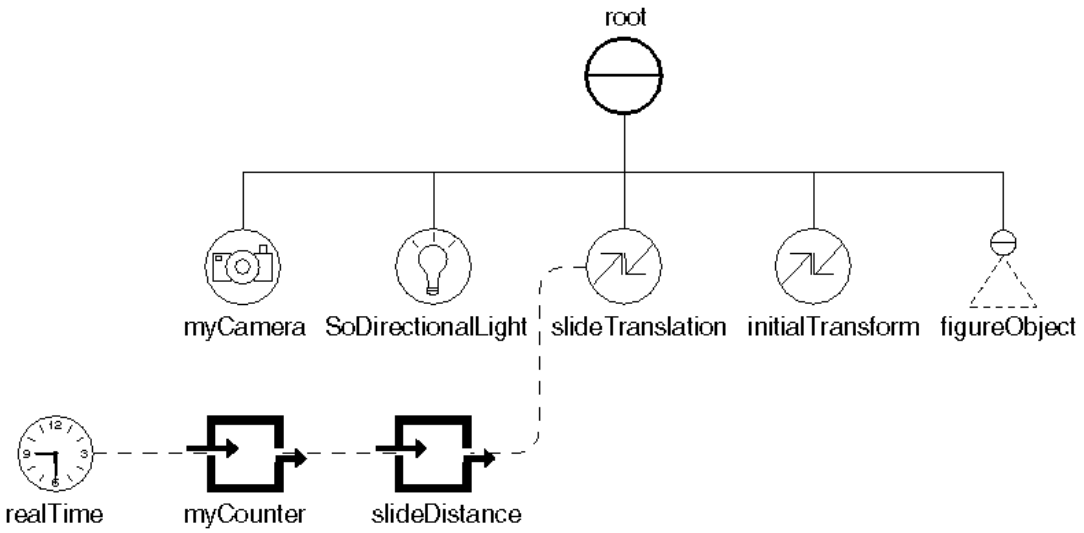
计时引擎（Elapsed-Time Engine）

计时引擎是一种基本的可控的时间源。我们可以控制引擎的开始、停止、重置、暂停和速度。如果我们暂停了引擎（将**pause**设置成TRUE），引擎将停止更新它的**timeOut**域，不过，引擎仍然会在内部继续计时。当我们关闭暂停功能后，引擎的输出会直接跳到当前时间位置上，这样就不会发生时间错位的现象了。

例 13-2 使用计时引擎的输出数据来控制一个物体的位置。程序运行的效果是物体会滑过屏幕窗口。图 13-9 给出了这个例子的场景图。计时引擎（*myCounter*）的**timeOut**输出连接到一个**SoComposeVec3f**引擎上（*slideDistance*）。第二个引擎将**timeOut**的数值赋值到一个向量的x方向上。当**timeOut**的数值变成了向量格式后，这个向量就可以连接到*slideTranslation*节点的**translation**域上了。

注意，**timeOut**数值是一个**SoSFTime**类型的数值，但是**SoComposeVec3f**引擎却需要**SoSFFloat**类型的输入。Inventor会自动执行这两种数据类型的转换，即将时间数据转换成从当前运行的秒数。

图 13-9 使用计时引擎例子的场景图



例 13-2 使用计时引擎

```
// Set up transformations
SoTranslation *slideTranslation = new SoTranslation;
root->addChild(slideTranslation);
SoTransform *initialTransform = new SoTransform;
initialTransform->translation.setValue(-5., 0., 0.);
initialTransform->scaleFactor.setValue(10., 10., 10.);
initialTransform->rotation.setValue(SbVec3f(1,0,0), M_PI/2.);
root->addChild(initialTransform);
```

```
// Read the figure object from a file and add to the scene
SoInput myInput;
if (!myInput.openFile("jumpyMan.iv"))
   return (1);
SoSeparator *figureObject = SoDB::readAll(&myInput);
if (figureObject == NULL)
   return (1);
root->addChild(figureObject);

// Make the X translation value change over time.
SoElapsedTime *myCounter = new SoElapsedTime;
SoComposeVec3f *slideDistance = new SoComposeVec3f;
slideDistance->x.connectFrom(&myCounter->timeOut);
slideTranslation->translation.connectFrom(
         &slideDistance->vector);
```

单触发引擎

SoOneShot引擎会在它的**trigger**输入域受到“触动”（touched）（即通过调用**touch()**或**setValue()**函数来修改引擎的内部数据时）后开始计时。它会运行**duration**域所指定一段时间，期间它会更新**timeOut**输出域数值，直到运行完指定的持续时间为止。**SoOneShot**引擎的**ramp**输出域将输出一个从 0.0(当触发器开始的时候)到 1.0（当运行完指定的持续时间）的浮点值。**ramp**输出域可以为某些应用提供一种使用上的便利。例如，场景中有一扇窗户，我们可以将**ramp**输出域连接到一个差值旋转节点的**alpha**输入域上，这样就可以模拟打开窗户的过程了。（**插值旋转节点是指SoInterpolateRotation节点，它本身也是一个引擎。SoInterpolateRotation可以在两个方向之间进行插值运算，计算出从一个方向转到另一个方向的过程中，中间的某个临时方向。它有一个alpha输入域，当alpha输入域的值为 0 时，SoInterpolateRotation输出第一个方向数值，当alpha输入域的值为 1 时，输出第二个方向数值。当alpha输入域的值在 0 到 1 之间的时候，SoInterpolateRotation会输出中间插值计算的方向。因此可以使用这个节点来模拟窗户的打开过程。译者注**）

SoOneShot引擎在**SoSFBitMask**域中有两个标志位。**Retriggerable**标志位指示如果在循环的中间时刻发生了一次触发，引擎是否需要重新开始计数循环。如果没有设置这个标志位（缺省情况），那么这个在循环中间时刻发生的触发将会被引擎忽略掉，并且会完成整个循环。如果设置了这个标志位，那么只要发生了触发事件，引擎会立即重新开始循环。

Hold_Final标志表示当循环结束后，引擎该如何处理输出数据。如果没有设置这个标志位（缺省情况），当循环结束时，引擎的所有输出都会重置为 0。如果设置了这个标志位，则在循环结束时，**isActive**输出会变成 0，但**ramp**和**timeOut**这两个域会继续保持原来的数值。

计数引擎（Time-Counter Engine）

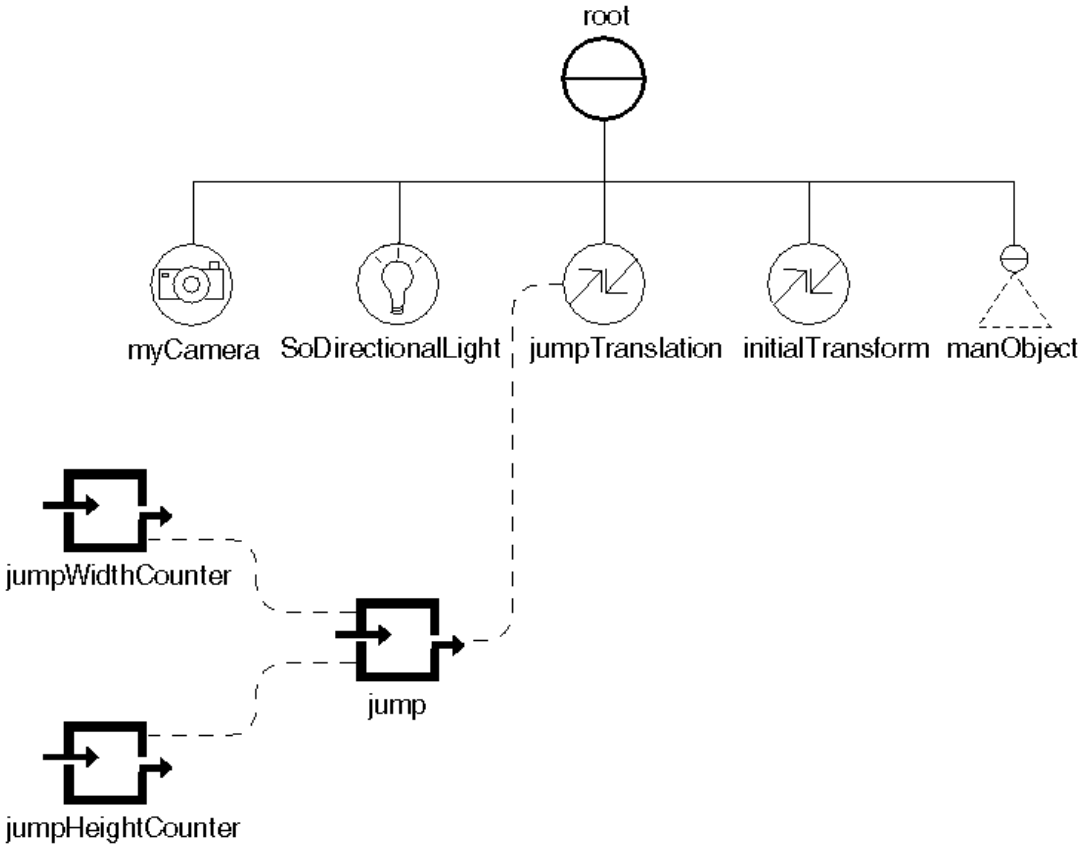
SoTimeCounter引擎会在最小数（min）与最大数（max）这两个数之间进行计数操作。**step**的域值表示定时器的计时步长（缺省情况下是每次增加 1），**frequency**域表示每秒中从小到大的循环次数。

不像单触发引擎和计时引擎那样，计数引擎不输出时间数据；它输出的是当前计数值。每次当时间计数器开始循环的时候，计数引擎就会触发它的**syncOut**输出。这种输出可以用来和某个事件激发引擎保持同步状态（Each time the time counter starts a cycle, it triggers its **syncOut** output. This output can be used to synchronize one of the triggered engines with some other event）。

例 13-3 使用两个计数引擎的输出来控制物体的水平和垂直运动。程序运行的效果是物体跳跃着滑过屏幕窗口。

如图 13-10 中所示，例 13-3 创建了 3 个引擎。*jumpWidthCounter*（计数引擎）的输出连接到*jump*引擎（**SoComposeVec3f**引擎）的x输入上。*jumpHeightCounter*（另外一个计数引擎）的输出连接到*jump*引擎的y输入上。*jump*引擎使用x和y的输入数据组成了一个向量方向数据。然后将这个向量赋值到*jumpTranslation*节点的**translation**域中。图 13-11 显示了例子的运行效果。

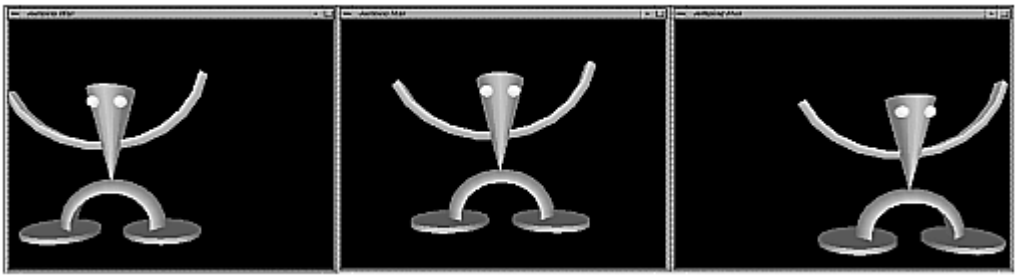
如图 13-10 计数引擎例子的场景图



例 13-3 使用计数引擎

```
// Set up transformations
SoTranslation *jumpTranslation = new SoTranslation;
root->addChild(jumpTranslation);
SoTransform *initialTransform = new SoTransform;
initialTransform->translation.setValue(-20., 0., 0.);
initialTransform->scaleFactor.setValue(40., 40., 40.);
initialTransform->rotation.setValue(SbVec3f(1,0,0), M_PI/2.);
root->addChild(initialTransform);

// Read the man object from a file and add to the scene
SoInput myInput;
if (!myInput.openFile("jumpyMan.iv"))
   return (1);
SoSeparator *manObject = SoDB::readAll(&myInput);
if (manObject == NULL)
   return (1);
root->addChild(manObject);
```

图 13-11 使用计数引擎控制物体的运动



```
// Create two counters, and connect to X and Y translations.
// The Y counter is small and high frequency.
// The X counter is large and low frequency.
// This results in small jumps across the screen,
// left to right, again and again and again.
SoTimeCounter *jumpHeightCounter = new SoTimeCounter;
SoTimeCounter *jumpWidthCounter = new SoTimeCounter;
SoComposeVec3f *jump = new SoComposeVec3f;

jumpHeightCounter->max = 4;
jumpHeightCounter->frequency = 1.5;
```

```
jumpWidthCounter->max = 40;
jumpWidthCounter->frequency = 0.15;

jump->x.connectFrom(&jumpWidthCounter->output);
jump->y.connectFrom(&jumpHeightCounter->output);
jumpTranslation->translation.connectFrom(&jump->vector);
```

闸门引擎

本节将讨论*闸门引擎*,闸门引擎提供了一种可以有选择地将数据从输入域拷贝到输出域中的简便机制。它包含有两个可被其它引擎所使用的 *enable* 域和 *trigger* 域。

缺省情况下,每次当引擎网络中的一个数据发生了变化,这个新数据就会传遍整个网络。然而，如果一个数据是持续频繁地发生变化，我们也许不希望这个变化会持续不断地传遍整个场景（**主要是出于对系统性能的考虑。译者注**）。在这种情况下，我们也许希望可以按照一个固定的时间间隔来采样这个数据，或仅当某个特定事件发生后才去更新这个数据。这时，我们可以使用闸门引擎来控制某些数据何时应该被发送到场景中的其它部分中。

当我们构造闸门引擎的时候，我们需要向构造函数传递输入和输出域的参数类型。域类型必须是多值域的类型（如果想对一个单值域使用闸门引擎，只需要传入其对应的多值域类型即可。Inventor将会自动进行类型转换）。**SoSelectOne**和**SoConcatenate**引擎也使用类似的构造方法。

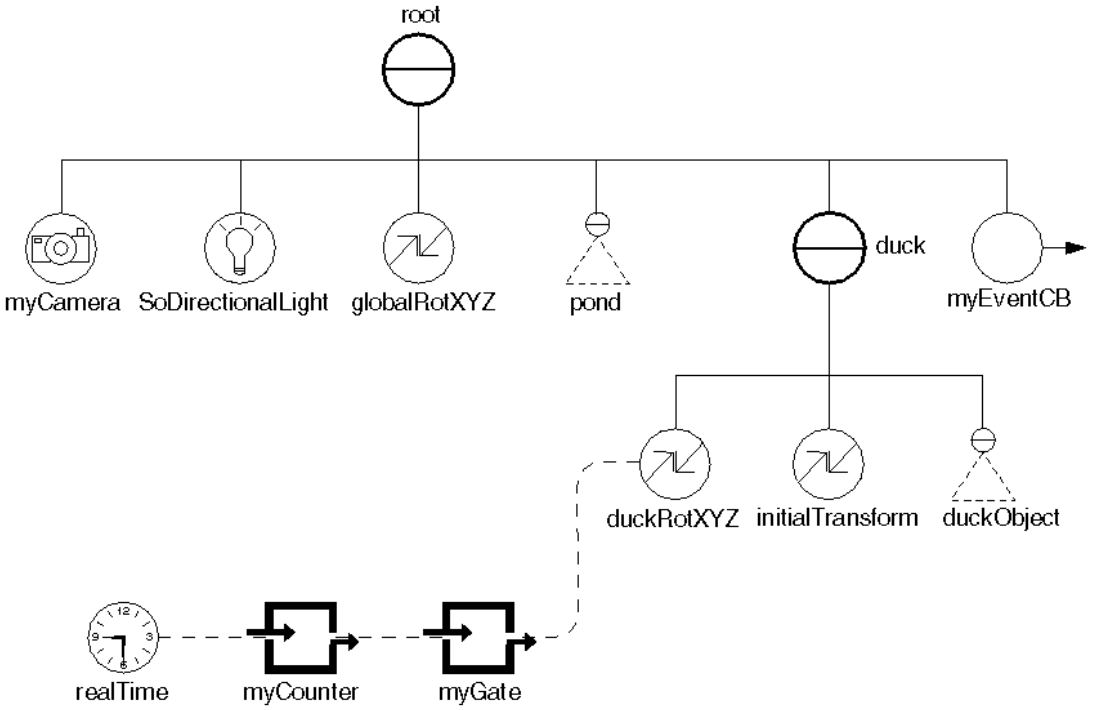
SoGate引擎有两个输入：

enable（SoSFBool）　　允许更新数据流通过

trigger（SoSFTrigger）只拷贝传递一个数据

当**enable**域为TURE时，每次引擎从输入域中接收到的新数据都将立即被拷贝到输出域中。当设置**enable**域为FALSE,并且使用**trigger**域发送数据时，将只发送一个数据到引擎的输出域中。当“触动”一次**trigger**域，引擎就会发送一个数据。调用引擎的**touch()**或**setValue()**方法可以“触动”**trigger**域。例 13-4 将一个计时引擎（*myCounter*）连接到一个闸门引擎（*myGate*）上。通过点击鼠标键，可以允许和禁止闸门引擎，而闸门引擎可以控制场景中鸭子的动作。图 13-2 是这个例子的场景图。

图 13-2 闸门引擎例子的场景图



例 13-4 使用闸门引擎

```
// Duck group
SoSeparator *duck = new SoSeparator;
root->addChild(duck);

// Read the duck object from a file and add to the group
SoInput myInput;
if (!myInput.openFile("duck.iv"))
   return (1);
SoSeparator *duckObject = SoDB::readAll(&myInput);
if (duckObject == NULL)
   return (1);

// Set up the duck transformations
SoRotationXYZ *duckRotXYZ = new SoRotationXYZ;
duck->addChild(duckRotXYZ);
SoTransform *initialTransform = new SoTransform;
initialTransform->translation.setValue(0., 0., 3.);
initialTransform->scaleFactor.setValue(6., 6., 6.);
duck->addChild(initialTransform);

duck->addChild(duckObject);

// Update the rotation value if the gate is enabled.
```

```
SoGate *myGate = new SoGate(SoMFFloat::getClassTypeId());
SoElapsedTime *myCounter = new SoElapsedTime;
myGate->input->connectFrom(&myCounter->timeOut);
duckRotXYZ->axis = SoRotationXYZ::Y;  // rotate about Y axis
duckRotXYZ->angle.connectFrom(myGate->output);

// Add an event callback to catch mouse button presses.
// Each button press will enable or disable the duck motion.
SoEventCallback *myEventCB = new SoEventCallback;
myEventCB->addEventCallback(
         SoMouseButtonEvent::getClassTypeId(),
         myMousePressCB, myGate);
root->addChild(myEventCB);

...

// This routine is called for every mouse button event.
void
myMousePressCB(void *userData, SoEventCallback *eventCB)
{
   SoGate *gate = (SoGate *) userData;
   const SoEvent *event = eventCB->getEvent();
   // Check for mouse button being pressed
   if (SO_MOUSE_PRESS_EVENT(event, ANY)) {

      // Toggle the gate that controls the duck motion
      if (gate->enable.getValue())
         gate->enable.setValue(FALSE);
      else
         gate->enable.setValue(TRUE);

      eventCB->setHandled();
   }
}
```

算法引擎

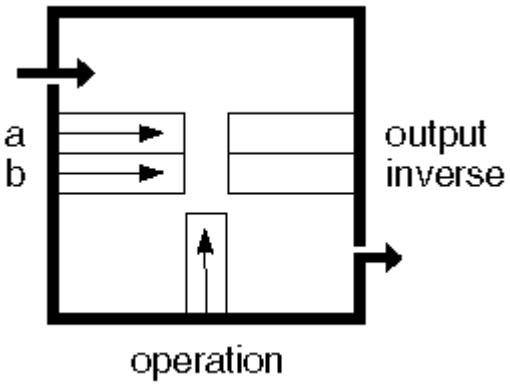
按照惯例，算法引擎的所有输入和输出域都应该是多值域（MF）。如果我们提供的是一个**SoSF**类型的数值，Inventor会自动将它转换成一个**MF**类型的数值。另一个重要的特性是，如果我们传递给输入域的参数是一个数值数组的话，那么引擎的输出也将是一个数组（MF类型）。如果引擎有多个输入域，也许传入到某些输入域的数据要多于其它输入域的数据。例如，input1 也许有 5 个输入值，而input2 可能只有 3 个输入值。在这种情况下，那些只

有少数数值的域将会重复域中最后一个数据，以此来填满数组的空余位置（例如，input2 的第 3 个数值将会被重复 2 次）。

布尔引擎

如图 13-13 所示，布尔引擎，**SoBoolOperation**,有两个布尔输入（a和b），和一个用来描述执行何种操作的**SoSFEnum**输入（**operation**）。

图 13-13 **SoBoolOperation**引擎



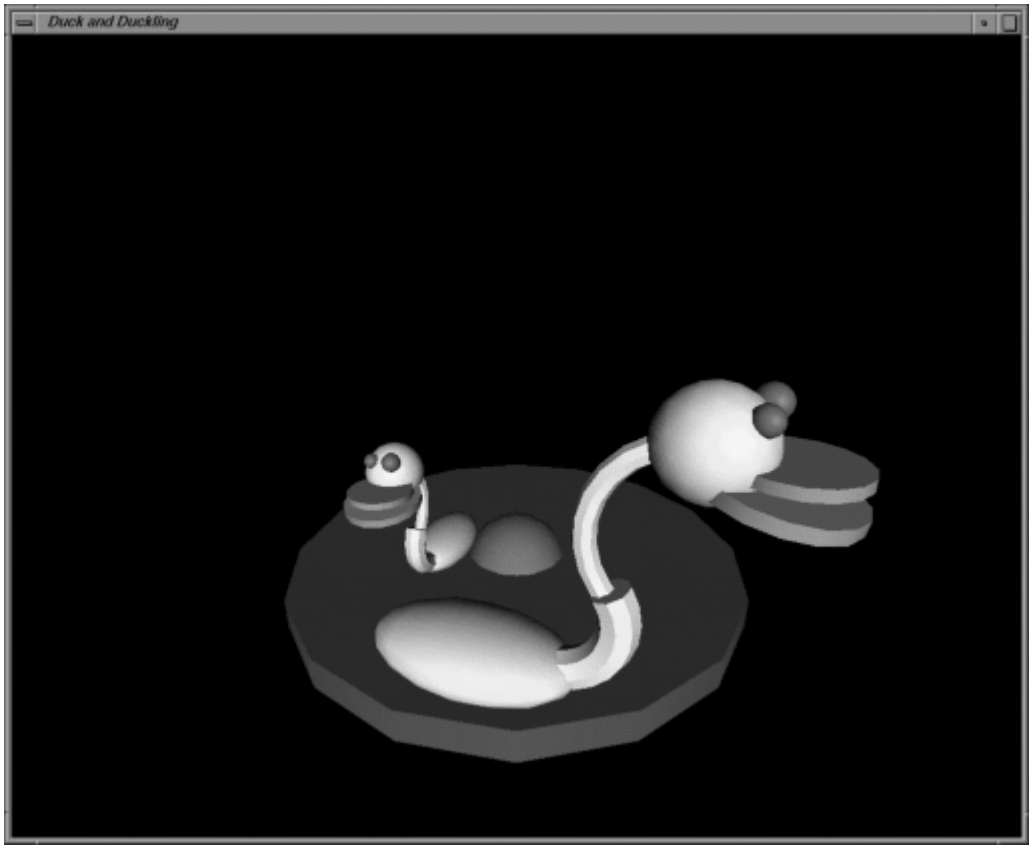
下面是**operation**输入域的有效值：

操作	输出为 TRUE，如果
CLEAR	输出永远是 FALSE
SET	输出永远是 TRUE
A	A 等于 TRUE
NOT_A	A 等于 FALSE
B	B 等于 TRUE
NOT_B	B 等于 FALSE
A_OR_B	A 等于 TRUE 或 B 等于 TRUE
NOT_A_OR_B	A 等于 FALSE 或 B 等于 TRUE
A_OR_NOT_B	A 等于 TRUE 或 B 等于 FALSE
NOT_A_OR_NOT_B	A 等于 FALSE 或 B 等于 FALSE
A_AND_B	A 和 B 等于 TRUE
NOT_A_AND_B	A 等于 FALSE 和 B 等于 TRUE
A_AND_NOT_B	A 等于 TRUE 和 B 等于 FALSE
NOT_A_AND_NOT_B	A 和 B 等于 FALSE
A_EQUALS_B	A 等于 B
A_NOT_EQUALS_B	A 不等于 B

布尔引擎有两个输出域：**output**和**inverse**。如果**output**是FALSE,则**inverse**是TRUE，反之亦然。如果传入某个输入域的是一个数组数值的话（即**SoMFBool**类型的数值），那么输出域也会输出一个数组数值。

例 13-5 修改了例 13-4 的代码，增加上一个布尔引擎，根据大鸭子的行为来控制小鸭子的行为。当大鸭子静止不同的时候，小鸭子将会绕中心运动。图 13-14 给出了这个例子所产生的场景图像。

图 13-14 被布尔引擎控制的游泳的鸭子



例 13-5 使用布尔引擎

```
// Bigger duck group
SoSeparator *bigDuck = new SoSeparator;
root->addChild(bigDuck);
SoRotationXYZ *bigDuckRotXYZ = new SoRotationXYZ;
bigDuck->addChild(bigDuckRotXYZ);
SoTransform *bigInitialTransform = new SoTransform;
bigInitialTransform->translation.setValue(0., 0., 3.5);
bigInitialTransform->scaleFactor.setValue(6., 6., 6.);
bigDuck->addChild(bigInitialTransform);
bigDuck->addChild(duckObject);
```

```
// Smaller duck group
SoSeparator *smallDuck = new SoSeparator;
root->addChild(smallDuck);
SoRotationXYZ *smallDuckRotXYZ = new SoRotationXYZ;
smallDuck->addChild(smallDuckRotXYZ);
SoTransform *smallInitialTransform = new SoTransform;
smallInitialTransform->translation.setValue(0., -2.24, 1.5);
smallInitialTransform->scaleFactor.setValue(4., 4., 4.);
smallDuck->addChild(smallInitialTransform);
smallDuck->addChild(duckObject);

// Use a gate engine to start/stop the rotation of
// the bigger duck.
SoGate *bigDuckGate =
         new SoGate(SoMFFloat::getClassTypeId());
SoElapsedTime *bigDuckTime = new SoElapsedTime;
bigDuckGate->input->connectFrom(&bigDuckTime->timeOut);
bigDuckRotXYZ->axis = SoRotationXYZ::Y;
bigDuckRotXYZ->angle.connectFrom(bigDuckGate->output);

// Each mouse button press will enable/disable the gate
// controlling the bigger duck.
SoEventCallback *myEventCB = new SoEventCallback;
myEventCB->addEventCallback(
         SoMouseButtonEvent::getClassTypeId(),
         myMousePressCB, bigDuckGate);
root->addChild(myEventCB);

// Use a Boolean engine to make the rotation of the smaller
// duck depend on the bigger duck.  The smaller duck moves
// only when the bigger duck is still.
SoBoolOperation *myBoolean = new SoBoolOperation;
myBoolean->a.connectFrom(&bigDuckGate->enable);
myBoolean->operation = SoBoolOperation::NOT_A;

SoGate *smallDuckGate = new
         SoGate(SoMFFloat::getClassTypeId());
SoElapsedTime *smallDuckTime = new SoElapsedTime;
smallDuckGate->input->connectFrom(&smallDuckTime->timeOut);
smallDuckGate->enable.connectFrom(&myBoolean->output);
smallDuckRotXYZ->axis = SoRotationXYZ::Y;
smallDuckRotXYZ->angle.connectFrom(smallDuckGate->output);
```

计算引擎（Calculator Engine）

计算引擎**SoCalculator**，类似于布尔引擎。不过计算引擎可以处理更广范围的操作，并且具有更多的输入和输出域。计算引擎具有下列的输入输出域：

Inputs	SoMFFloat	a, b, c, d, e, f, g, h
	SoMFVec3f	A, B, C, D, E, F, G, H
	SoMFString	表达式
Outputs	SoEngineOutput	oa, ob, oc, od (SoMFFloat)
	SoEngineOutput	oA, oB, oC, oD (SoMFVec3f)

expression是一个**SoMFString**类型的输入域，它一般表达成下面的形式：

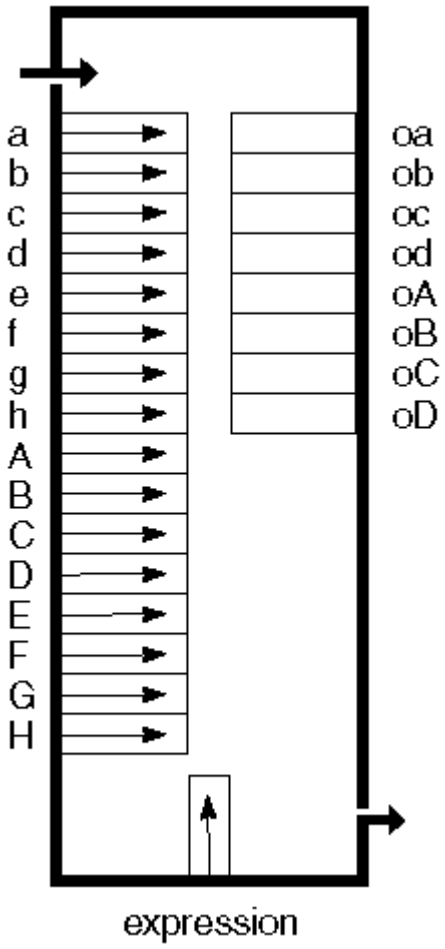
"*lhs* = *rhs*"

lhs（左侧）可以是任何一个输出域符号或一个中间临时变量。计算引擎提供有 8 个临时的浮点变量（使用 ta～th 来表示），还提供 8 个临时的向量变量（使用 tA～tH 来表示）。

rhs（右侧）支持下列操作：

操作类型	例子
二元操作符	+ - * / < > >= <= == != && ||
一元操作符	- !
条件操作符	*cond* ? *trueexpr* : *falseexpr*
括号操作符	(*expr*)
向量下标操作符	*vec* [*int*]
函数	func(*expr*, ...)
其它项目	命名为 MAXFLOAT, MINFLOAT, M_LOG2E, M_PI 的整形或浮点型常量； 引擎的输入、输出、以及临时变量名称（a, b, A, B, oa, ob, ta, tb, tA, tB, and so on）

图 13-15 **SoCalculator** 引擎



在 *Open Inventor C++ Reference Manual* 书中列有更多关于如何使用这些操作的内容。

下面是一个使用计算引擎的简单例子。这个例子使用下列输入和输出域：

输入	输出
2 个向量(A, B)	oA (A 与 B 叉乘结果的负值乘 f)
2 个标量值(a, f)	oa (将 a 从弧度单位转成角度单位)

下面的代码是为一个叫做 *calc* 的计算引擎指定计算表达式：

```
calc->expression.set1Value(0, "oa = a * M_PI / 180");
calc->expression.set1Value(1, "oA = -f * cross(A, B)");
```

多重表达式需要按照计算顺序分别进行计算。一个被前面的表达式赋值的变量可以用在后面的右侧表达式种。可以在一个字符串中指定多个表达式，这些表达式使用分号相互隔离。

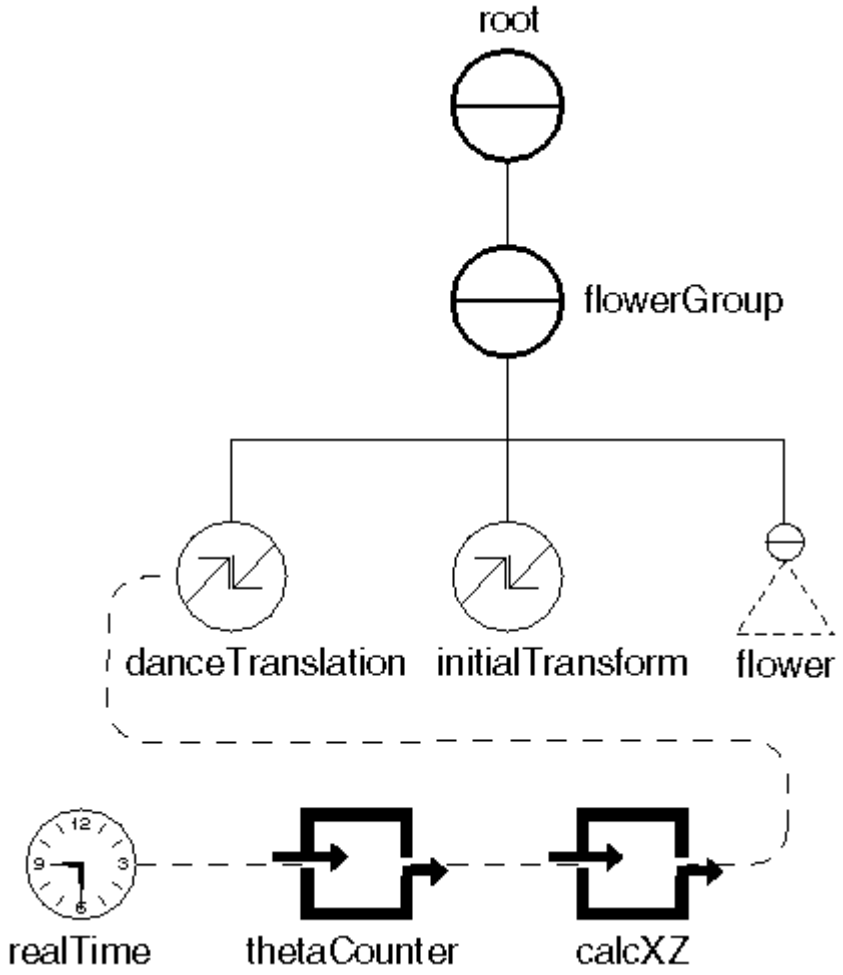
表达式也可以对数组进行操作。如果一个输入域包含的数据少于其它输入域的数据，那么这个输入域的最后一个数值将被重复复制，直到填满数组剩余的部分为止。所有的表达式都会分别被应用到数组的每个元素上。例如，如果输入域 a 包含有多个值，输入 b 只包含一

个值 1.0，那么表达式“oa = a + b”将会使 a 中的所有元素都加 1。

使用计算引擎来约束控制物体的行为

例 13-6 将演示使用计算引擎让一束鲜花沿着一个路径运动。计算引擎计算出一个 2 维闭合的曲线。计算引擎的输出连接到鲜花的平移节点上，之后鲜花就会沿着曲线的路径进行运动。图 13-16 给出了这个例子的场景图。图 13-17 中绘制了这支“跳舞”的鲜花。

图 13-16 计算引擎例子的场景图

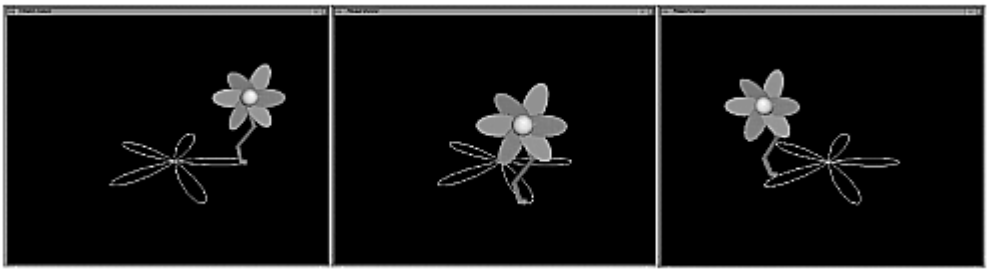


例 13-6 使用计算引擎

```
// Flower group
SoSeparator *flowerGroup = new SoSeparator;
root->addChild(flowerGroup);

// Read the flower object from a file and add to the group
if (!myInput.openFile("flower.iv"))
   exit(1);
SoSeparator *flower= SoDB::readAll(&myInput);
```

```
if (flower == NULL)
   exit(1);

// Set up the flower transformations
SoTranslation *danceTranslation = new SoTranslation;
SoTransform *initialTransform = new SoTransform;
flowerGroup->addChild(danceTranslation);
initialTransform->scaleFactor.setValue(10., 10., 10.);
initialTransform->translation.setValue(0., 0., 5.);
flowerGroup->addChild(initialTransform);
flowerGroup->addChild(flower);
```

图 13-17 使用计算引擎来约束控制物体的行为



```
// Set up an engine to calculate the motion path:
// r = 5*cos(5*theta); x = r*cos(theta); z = r*sin(theta)
// Theta is incremented using a time counter engine,
// and converted to radians using an expression in
// the calculator engine.
SoCalculator *calcXZ = new SoCalculator;
SoTimeCounter *thetaCounter = new SoTimeCounter;

thetaCounter->max = 360;
thetaCounter->step = 4;
thetaCounter->frequency = 0.075;

calcXZ->a.connectFrom(&thetaCounter->output);
calcXZ->expression.set1Value(0, "ta=a*M_PI/180");   // theta
calcXZ->expression.set1Value(1, "tb=5*cos(5*ta)");  // r
calcXZ->expression.set1Value(2, "td=tb*cos(ta)");   // x
calcXZ->expression.set1Value(3, "te=tb*sin(ta)");   // z
```

```
calcXZ->expression.set1Value(4, "oA=vec3f(td,0,te)");
danceTranslation->translation.connectFrom(&calcXZ->oA);
```

产生动画效果的节点

引擎通常都是连接到节点上的。尽管我们可以创建一个带有内建引擎，并且可以自动与引擎相连接的节点类，但其实上，Inventor 已经提供了一些具有上述功能的引擎节点。这些引擎节点提供了一种可以向场景增加上动画效果的简单机制，。

- **SoRotor**是一个几何变换节点，它可以保持旋转轴不变，绕轴旋转一个角度。
- **SoPendulum**是一个几何变换节点，可以在两个旋转角度之间振荡。
- **SoShuttle**是一个几何变换节点，可以在两个平移量之间振荡。
- **SoBlinker**是一个开关节点，可以循环遍历它的子节点。

下面是 rotor 和 blinker 节点的使用例子。

转体节点（Rotor）

SoRotor节点派生于**SoRotation**，它可以按照一定的速率来修改旋转角度。我们可以在任何使用**SoRotation**的地方同样使用**SoRotor**节点。它下面这些域：

rotation (SoSFRotation)	指定旋转量（轴方向和初始角度）。当 rotor 旋转时修改角度
speed (SoSFFloat)	指定每秒循环旋转的次数
on (SoSFBool)	TRUE 表示运行，FALSE 表示停止。缺省值为 TRUE。

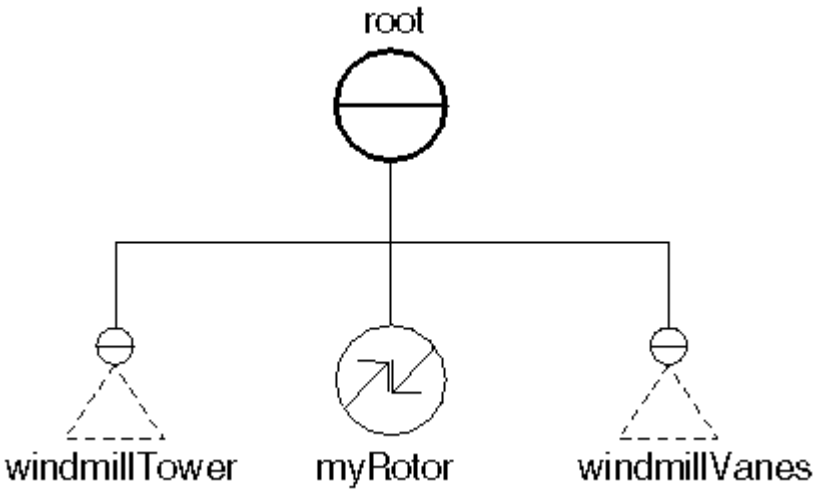
转体节点的每秒更新次数取决于应用程序系统本身。这个节点内建有一个引擎，这个引擎连接到实时（Real-Time）全局域上。例 13-7 将演示如何使用这个节点来旋转风车的叶片。例子为转体节点指定了旋转量和旋转速度。并且在向场景读入风车图形之前先增加上这个节点。风车的旋转轴为（0.0,0.0,1.0），初始旋转角度为 0.0。转体节点将自动更新旋转这个旋转角度。

例 13-7 使用转体节点来驱动的旋转风车

```
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoRotor.h>

SoSeparator *
```

```
readFile(const char *filename)
{
   // Open the input file
   SoInput mySceneInput;
   if (!mySceneInput.openFile(filename)) {
      fprintf(stderr, "Cannot open file %s\n", filename);
      return NULL;
   }
```

图 13-18 转体节点例子的场景图




```
   // Read the whole file into the database
   SoSeparator *myGraph = SoDB::readAll(&mySceneInput);
   if (myGraph == NULL) {
      fprintf(stderr, "Problem reading file\n");
      return NULL;
   }

   mySceneInput.closeFile();
   return myGraph;
}

main(int, char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
```

```
   SoSeparator *root = new SoSeparator;
   root->ref();

   // Read in the data for the windmill tower
   SoSeparator *windmillTower =
            readFile("windmillTower.iv");
   root->addChild(windmillTower);

   // Add a rotor node to spin the vanes
   SoRotor *myRotor = new SoRotor;
   myRotor->rotation.setValue(SbVec3f(0, 0, 1), 0); // z axis
   myRotor->speed = 0.2;
   root->addChild(myRotor);

   // Read in the data for the windmill vanes
   SoSeparator *windmillVanes =
            readFile("windmillVanes.iv");
   root->addChild(windmillVanes);

   // Create a viewer
   SoXtExaminerViewer *myViewer =
            new SoXtExaminerViewer(myWindow);

   // Attach and show viewer
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Windmill");
   myViewer->show();

   // Loop forever
   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

频闪节点（Blinker）

SoBlinker节点派生于**SoSwitch**，可以通过修改其**whichChild**域的数值来循环引用它的子节点。**SoBlinker**节点带有下列域：

whichChild (SoSFLong) 被遍历引用的子节点索引值

speed (SoSFFloat) 每秒循环的次数

on (SoSFBool) TRUE 表示运行，FALSE 表示停止。缺省值为 TRUE

当**SoBlinker**节点只有一个子节点时，它将在第 0 个子节点和SO_SWITCH_NONE之间进行

循环引用。例 13-8 将演示如何让“Eat at Josie’s”这个字符串不停地闪烁。

图 13-19 使用频闪节点控制的闪烁的招牌

例 13-8 使用频闪节点创建一个闪烁的招牌

```
// Add the non-blinking part of the sign to the root
root->addChild(eatAt);

// Add the fast-blinking part to a blinker node
SoBlinker *fastBlinker = new SoBlinker;
root->addChild(fastBlinker);
fastBlinker->speed = 2;  // blinks 2 times a second
fastBlinker->addChild(josie);

// Add the slow-blinking part to another blinker node
SoBlinker *slowBlinker = new SoBlinker;
root->addChild(slowBlinker);
slowBlinker->speed = 0.5;  // 2 secs per cycle; 1 on, 1 off
slowBlinker->addChild(frame);
```

第十四章 节点工具包

本章目标

在阅读完本章节后，我们将有能力做下列事情：

- 在场景中使用节点工具包，选择工具包中所需要的部件，并可以设置这些部件。
- 解释路径、全路径、以及节点工具包路径之间的差异。
- 使用节点工具包来实现一个简单多级运动场景。

本章所要描述的节点工具包将提供一种创建Inventor节点分组的便捷机制。当我们要创建一个像索引三角面集（indexed triangle strip set）那样的形体节点时，通常我们至少需要坐标系节点、材质节点、几何变换节点等这些属性节点。如果我们希望产生更好的效果，也许还需要指定绘制风格节点和材质绑定节点。如果使用**SoShapeKit**节点工具包，我们就不必再单独创建这些属性节点，设置它们的域值，排列它们在子场景中的顺序了。**SoShapeKit**节点工具包本身已经包含了这些节点该如何排列在子场景中的必要信息。我们可以通过一组简单方便的方法来指定希望使用那个节点，以及设置和查询这些节点的域值。本章将主要介绍*节点工具包*、*工具包目录（node-kit catalogs）*、*目录条目（catalog entries）*、以及*隐藏子节点（hidden children）*等概念

为什么使用节点工具包

节点工具包提供了一种简便的方法来创建“单一的”或“复合的”节点图形场景。节点工具包还可以包含其它的节点工具包，利用这个特性，我们可以建立起一种工具包之间相互的层次关系。下面是使用节点工具包的一些好处：

- 节点工具包可以将许多Inventor节点组成一个对我们来说具有更高层次意义的子场景。例如，**SoShapeKit**可以用来描述一个能运动和有特定外观的物体。物体本身和它的属性都打包进一个节点工具包内。我们不必关心这些节点是如何放置在场景中的，因为节点工具包会为我们管理这些节点。
- 节点工具包具有更好的灵活性。即允许我们使用多种 Inventor 特性来创建结构复杂的子场景，也允许使用少量特性来创建一个简单的子场景。
- 节点工具包可以更高效地创建节点。对于某个特定的节点工具包，只创建必要的节点。
- 节点工具包提供了快捷的方法来创建节点和为节点赋值。我们的程序代码可能会变的更小和更容易阅读。
- 通过子类化，我们可以设计用于特定应用程序，用户自己定制的节点工具包。（见 *Inventor Toolmaker*，第 7 章）

隐藏的子节点与 SoNodeKitPath

在节点工具包中包含有一个多个节点组成的集合。节点工具包管理这些节点，负责安排节点在子场景中的排列顺序。我们可以创建和删除节点工具包中的节点、或*部件*（*parts*）。不过，实际上因为是节点工具包在全权管理着这些部件，所以我们没有权限直接使用它们。这些部件被称为节点工具包的*隐藏子节点*（*hidden children*）。尽管节点工具包是一个节点组，但它不是从**SoGroup**派生出来的；所以节点工具包中不存在像**addChild()**这样的函数。

无论何时，只要我们执行一次拾取或搜索动作时，动作都会返回一条路径数据。所返回的缺省路径是**SoPath**类型的，这种路径将会以第一个带有隐藏子节点的节点（通常是一个节点工具包）作为结束。如果我们需要知道有关路径中节点工具包下面是什么节点的更多详细信息时，我们可以将**SoPath**强制转换成**SoFullPath**。**SoFullPath**可以像包含公共子节点那样包含隐藏的子节点（**这样我们就可以访问节点工具包中的隐藏子节点了。译者注**）。例如，如果我们在一个给定的场景中搜索一个球体节点，我们也许会得到一条指向带有隐藏子节点的节点工具包路径，而这些隐藏的子节点中就包含有一个球体节点。搜索动作所返回的**SoPath**结束于节点工具包处。在通常的情况下，我们或许可以忽略这个隐藏的子节点。但是如果需要知道这些隐藏子节点信息，我们可以将这个路径转换成**SoFullPath**。

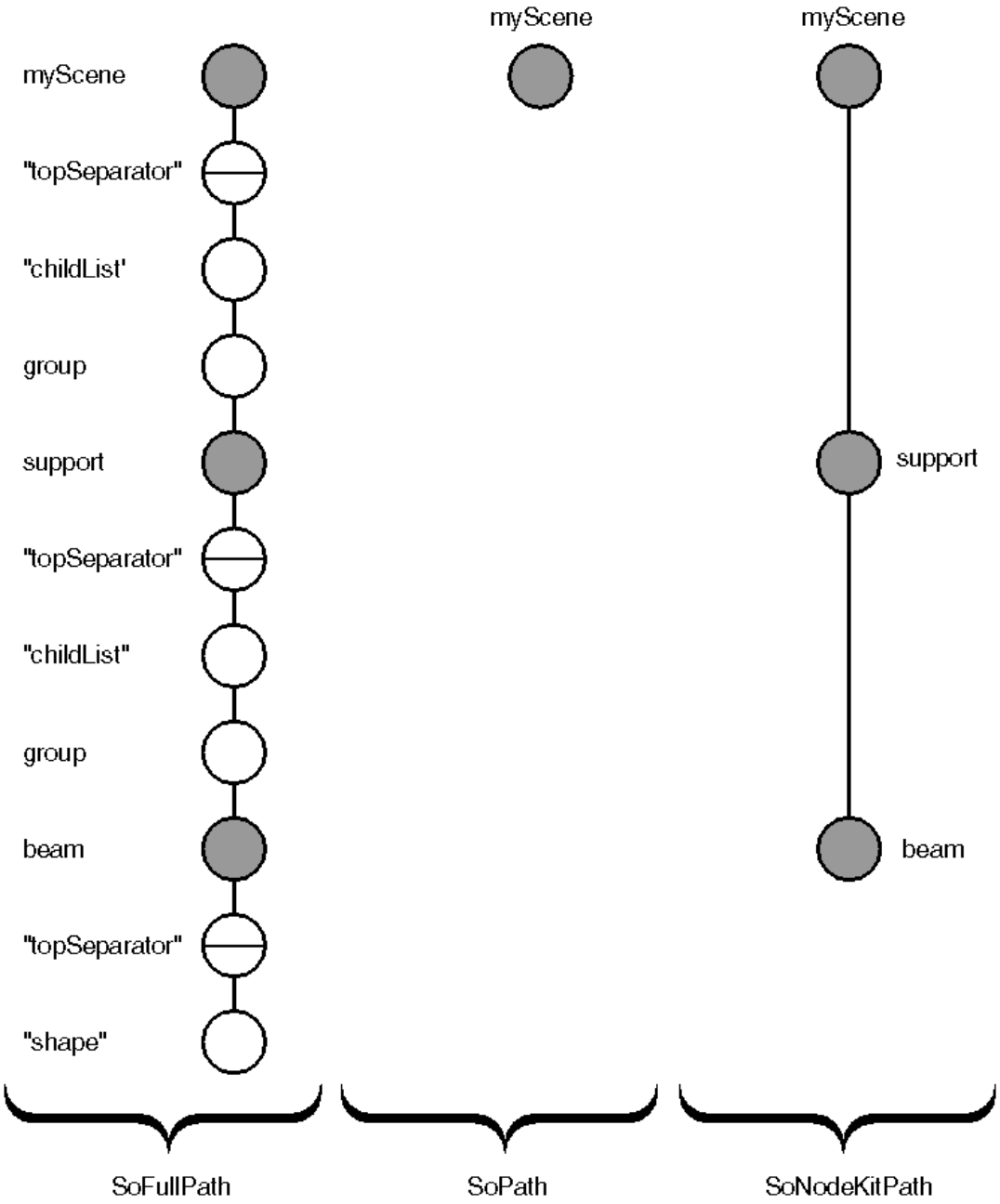
正常情况下，我们使用节点工具包路径的机会要远超过使用全路径的机会。如果我们使用一个带有节点工具包的全路径，一定要注意不要修改节点工具包的结构。

注意：当我们将一个路径（不是一个指针变量）强制转换成**SoFullPath**时，一定要确保转换的是一个指针；否则Inventor将会创建这个路径的一个新实例。例如：我们可以这样做：

```
SoPath &pathRef;
((SoFullPath *) &pathRef)->getLength();

But don't do this:
length = ((SoFullPath) pathRef).getLength();
```

另外一种类型的路径是**SoNodeKitPath**，这种路径只会保存节点工具包，路径中的其它节点都会被剔除掉。如果我们要查看一个运动场景的内部层次关系（见例 14-3），并且希望将每个节点工具包都作为一个物体来看待的话，这时可以使用节点工具包路径（**节点工具包路径可以让我们更有效和直观地观察场景的层次关系。译者注**）。图 14-1 为同一个场景给出了路径，全路径，节点工具包路径的示意图。带有阴影的圆环表示节点工具包。没有阴影的圆环表示其它节点。

图 14-1 不同类型的路径

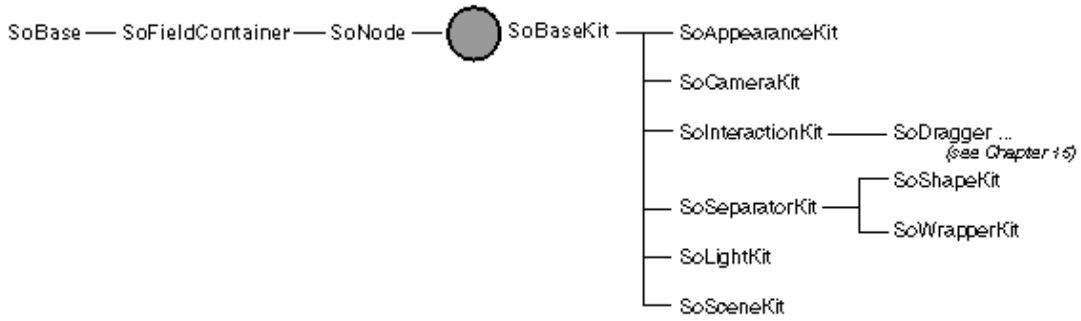


节点工具包类

图 14-2 给出了节点工具包的类树图，节点工具包类都是从**SoBaseKit**派生出来的。

请查看在*Open Inventor C++ Reference Manual*书中的**SoBaseKit**条目，在那里列有查询和设置节点工具包部件的方法列表。

图 14-2 节点工具包类



节点工具包目录

图 14-2 给出的每种节点工具包类都带有一个与之相关联的*目录列表*。在目录中列有节点工具包中所有可用的部件（或节点），这点有些像在电子设备或软件产品目录中列出所有可销售的项目。就像我们可以在软件产品目录中选择定购产品那样，我们也可以从节点工具包目录中选择一些节点。除了简单地列出了所有可用部件之外，节点工具包还将记录所选节点在子场景中是如何排列的。

例如，在图 14-3 中给出了**SoShapeKit**的目录列表。当我们首次创建**SoShapeKit**实例时，我们得到的是一个“基本模型（base model)”，如图 14-4 所示。缺省情况下，“shape”部件是一个立方体。我们可以修改这个“shape”部件，并可以任意增加所需的其它选项。

图 14-3 **SoShapeKit**的目录列表

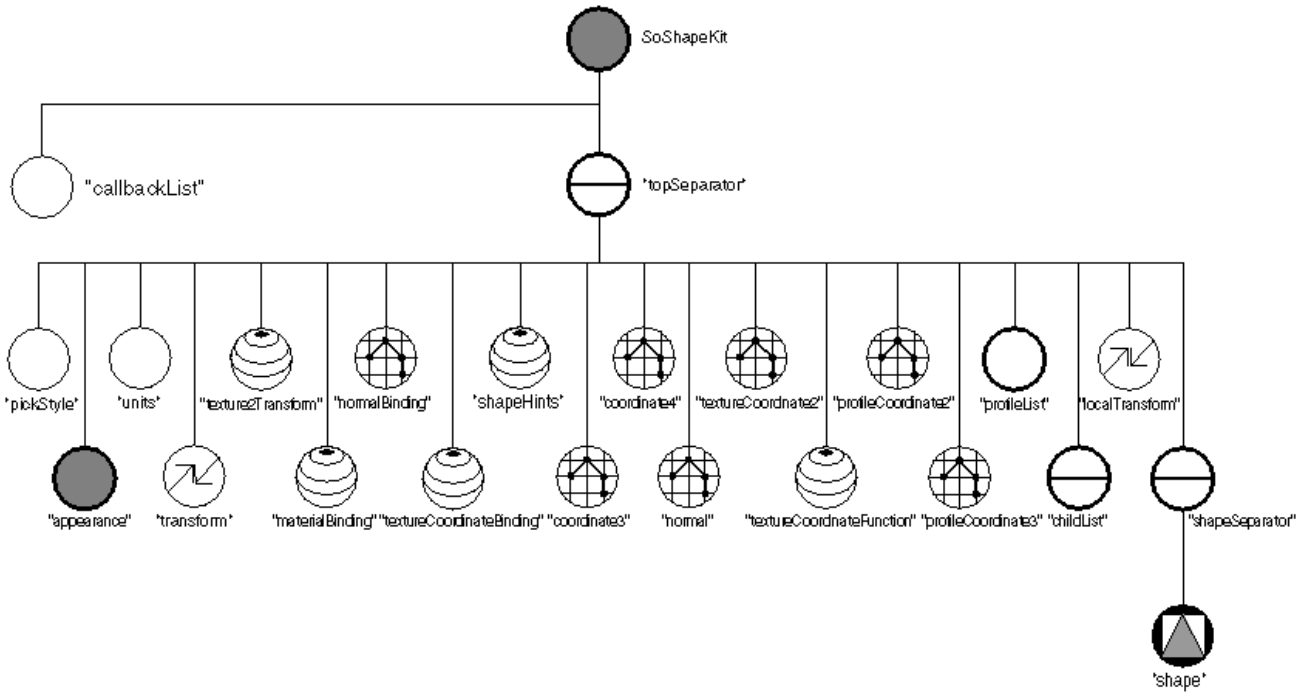
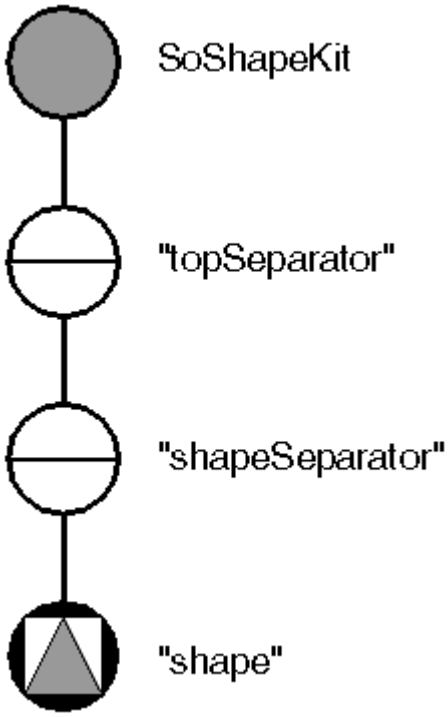


图 14-4 **SoShapeKit** 的基础版本



节点工具包目录中的一个条目可以描述工具包的一个部件。在图 14-3 所示的**SoShapeKit**目录中一共有 24 个条目。每个目录条目都包含有下列的信息：

- 本部件的名称
- 节点的类型
- 缺省类型（在类型是抽象类时使用）
- 本部件是否是缺省创建的
- 父部件的名称
- 本部件的右侧兄弟（*right sibling*）部件的名称
- 本部件是否是列表
- 如果部件是列表，那么组节点类型（*type of group node*）将用来保存列表项目
- 如果部件是列表，这里保存列表中目录的可允许节点类型（If the part is a list, the permissible *node types* for entries in this list）
- 部件是否是可以公开（*public*）访问的

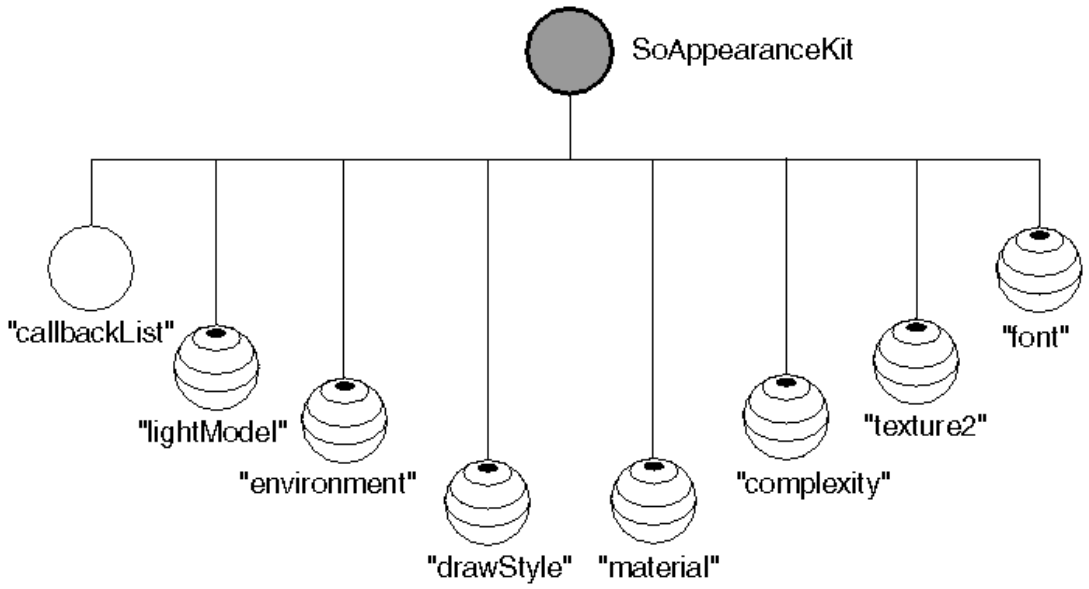
下面列出了一些**SoShapeKit**的目录条目例子：

Information	Sample Entry 1	Sample Entry 2
部件的名称	“callbackList”	“transform”
节点的类型	SoNodeKitListPart	SoTransform
缺省类型	(不可用)	(不可用)
否是缺省创建的 ？	FALSE	FALSE
父部件的名称	“this”	“topSeparator”
同层右侧部件	“topSeparator”	“texture2Transform”

Information	Sample Entry 1	Sample Entry 2
是否是列表 ?	TRUE	FALSE
列表容器的类型	SoSeparator	(不可用)
列表元素的类型	SoCallback	(不可用)
	SoEventCallback	
是否可以公开访问 ?	TRUE	TRUE

SoShapeKit节点工具包还包含有另外一个叫做“appearance”的节点工具包。“appearance”是一种类型为**SoAppearanceKit**的节点工具包。图 14-5 给出了**SoAppearanceKit**的目录列表。

图 14-5 **SoAppearanceKit的目录**



缺省创建的部件

下面的代码创建了一个**SoShapeKit**的实例：

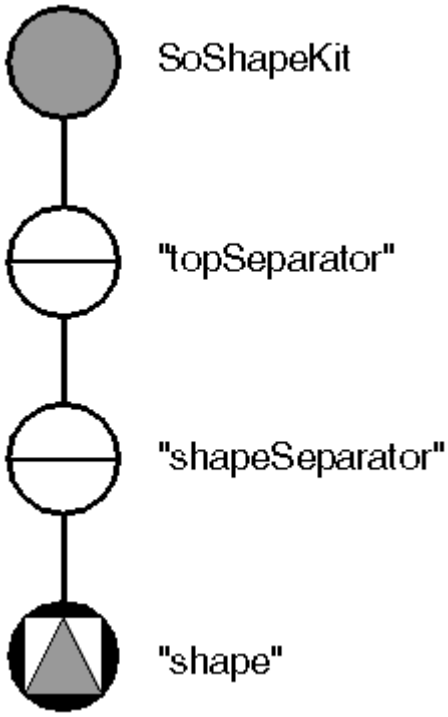
```
SoShapeKit *myShapeKit = new SoShapeKit();
```

当创建完一个节点工具包的实例后，某些节点也就同时被缺省创建了。在Inventor所提供的工具包中，**SoShapeKit**、**SoLightKit**、和**SoCameraKit**在构造的时候分别缺省地创建了“shape”,“light”，和“camera”部件。这些部件的缺省类型是**SoCube**、**SoDirectionalLight**、和**SoPerspectiveCamera**。

当构造完**SoShapeKit**后，**SoShapeKit**会自动创建“TopSeparator”、“ShapeSeparator”、和“shape”点。(像这种separator节点，是一种工具包内部使用的节点。当我们向节点工具包增加一个新节点时，这些内部节点将会被自动创建)。这时，工具包的场景图应该像图

14-4 所示的那样。**SoShapeKit**现在由 4 个节点组成：**SoShapeKit**节点本身、TopSeparator节点、ShapeSeparator节点（主要用于缓存Shape节点的形状信息，即使当外边的几何变换节点和材质节点发生了改变，仍然可以使用这个节点的缓存信息）、立方体节点。对于在**SoShapeKit**目录中的其它节点，只有在我们明确需要它们的时候才会被创建。

图 14-6　创建一个**SoShapeKit**实例



选择与设置部件数据

接下来，我们可以使用从**SoBaseKit**继承来的**set()**方法，创建工具包的部件和对新节点的域进行赋值。这种方法有两种调用格式：

set(*nameValuePairListString*); //使用大括号来分离部件名称和部件数值
或
set(*partNameString*, *parameterString*); // 不使用大括号

下面是**set()**第一种格式的例子代码，代码创建了一个材质节点，并将材质的漫射光颜色设置成紫色。

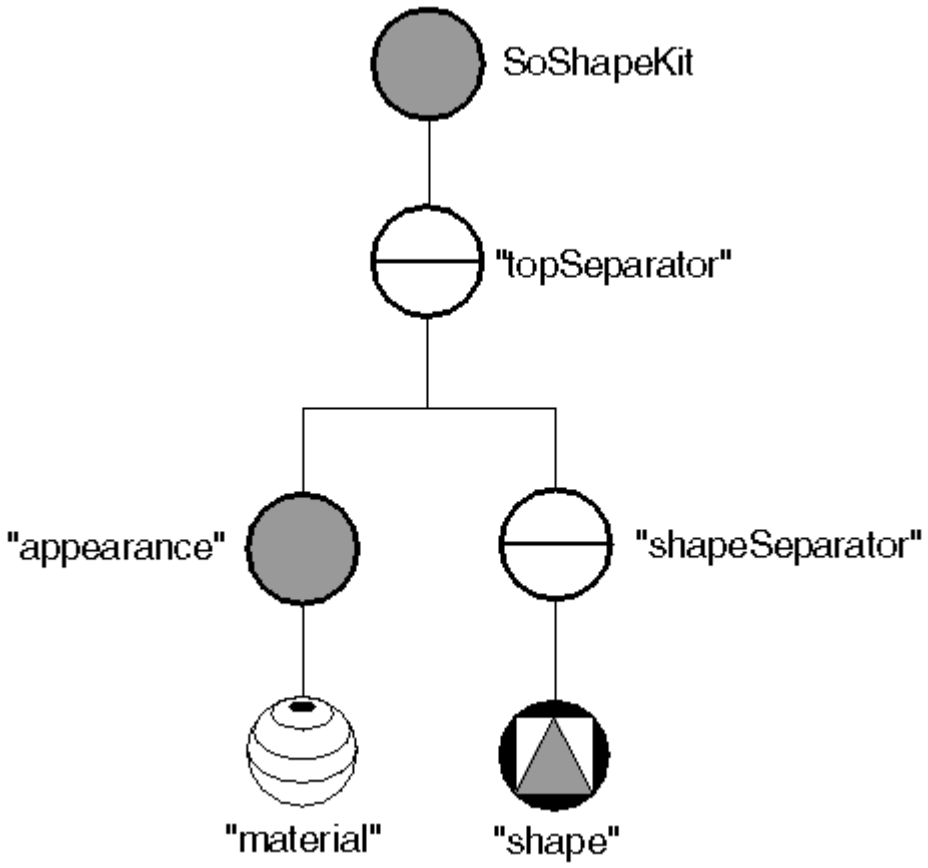
```
myShape->set("material { diffuseColor 1 0 1 }");
```

下面是**set()**第二种格式的例子代码，这个代码可以做和上面代码同样的事情：

```
myShape->set("material", "diffuseColor 1 0 1");
```

现在**SoShapeKit**实例的场景图应该如图 14-7 所示的那样。需要注意的是，当我们请求材质节点时，Inventor会同时自动创建**SoAppearanceKit**节点。还有，当调用**set()**方法设置材质节点的域值时，只有当材质节点不存在的时候，才会创建材质节点。而不是每调用一次**set()**方法就创建一遍材质节点。

图 14-7 增加材质节点

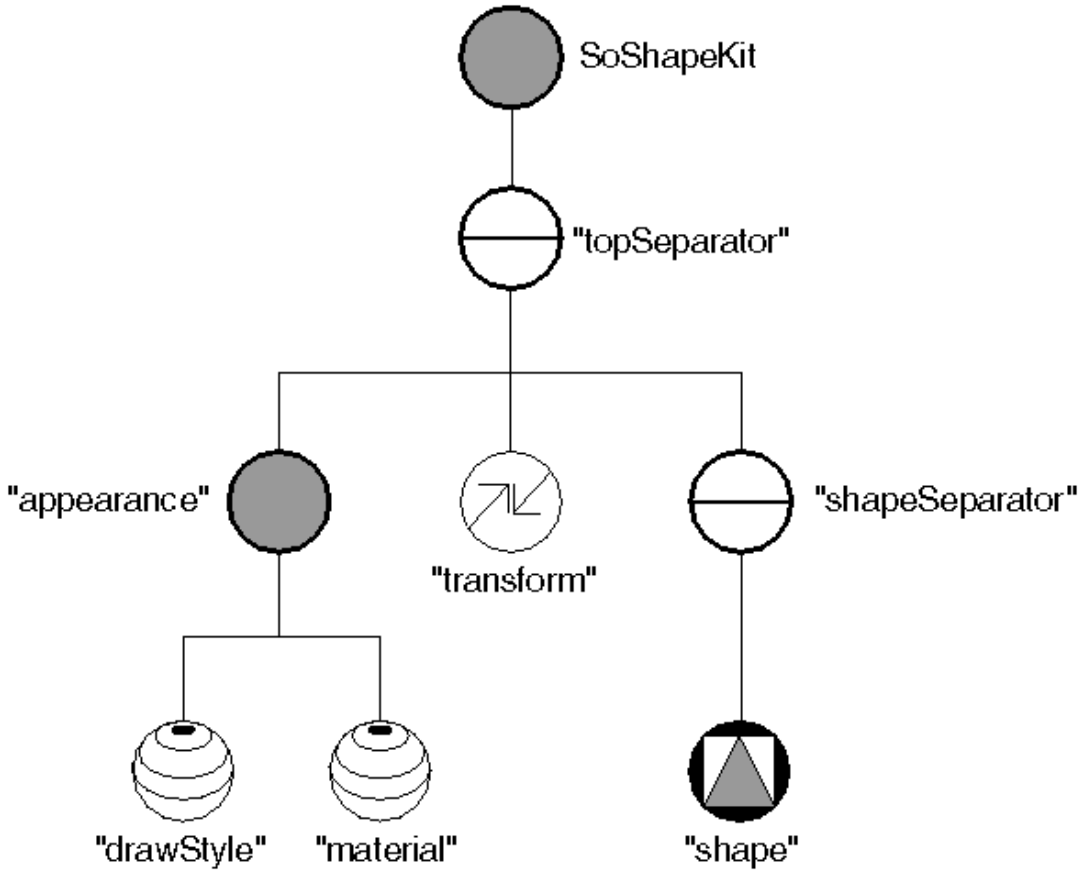


现在假设我们希望不使用实体填充风格，而使用框线风格并且体积要放大两倍来显示立方体：

```
myShape->set("drawStyle { style LINES }
             transform { scaleFactor 2.0 2.0 2.0 } ");
```

场景图如图 14-8 所示：

图 14-8 增加绘制风格和几何变换节点



请注意，我们可以按照任意顺序调用**set()**方法来创建节点。节点工具包会自动将创建的节点放置在场景中正确的位置上，所谓正确的位置，就是在节点工具包目录中所指定的位置。

现在**SoShapeKit**实例应该包含有 8 个节点，如图 14-8 所示。

其它方法：getPart（）和 setPart（）

SoBaseKit()另外两个非常有用的方法是**getPart()**和**setPart()**。

getPart() 方法

getPart()方法可以返回所需的节点（部件）：

getPart(*partName*, *makeIfNeeded*)；

如果 *makeIfNeeded* 参数为 TRUE,并且工具包中不存在 *partName* 所指定的节点，那么 Inventor 将会自动创建那个节点。另外，如果存在其它的节点需要连接到工具包的顶层节点（“this”）上，Inventor 也会自动创建它们（if any extra nodes are needed to connect the node to the top node（“this”）of the node kit, those nodes are created as well）。

例如：

```
xf = (SoTransform *) myKit->getPart("transform", TRUE);
```

上面的代码将会在节点工具包中查找一个名为“transform”的节点，如果当前节点工具包中存在“transform”节点，getPart（）将返回一个指向所找节点的指针。否则 getPart（）会自动新创建一个，并返回其节点指针。 如果我们将 *makeIfNeeded* 参数设置为 FALSE，并且节点工具包当前还没有“transform”节点，那么 getPart（）将会立即返回 NULL，不再自动创建新节点了。如果我们当前所用的节点工具包目录中不存在一个名为“transform”的条目，那么无论 *makeIfNeeded* 参数设置成什么，getPart（）都只会返回 NULL。

setPart() 方法

setPart()方法可以将一个给定的节点作为新部件插入到节点工具包中。

setPart(*partName*, *node*);

如果需要另外一些节点来连接部件到节点工具包结构中，这些节点也将会被创建（If extra nodes are required to connect the part into the node-kit structure, those nodes are created as well）。例如，假设我们希望另一个节点工具包可以分享使用前面例子创建的几何变换节点（*xf*）：

```
myOtherKit->setPart("transform", xf);
```

如果给定的节点类型不是从部件类型派生出来的(部件类型定义在节点工具包目录中)，那么将不能设置此部件。如果我们使用的是一个调试版本的 Inventor 库，则 Inventor 将会弹出一个错误消息。

要想删除几何变换条目，可以将节点指针设置成 NULL：

```
myOtherKit->setPart("transform", NULL);
```

将**SoShapeKit**中“shape”从缺省的立方体修改成圆锥体：

```
myShape->setPart("shape", new SoCone);
```

当然，如果在目录中并不存在一个给定名称的部件，**setPart()**将不会做任何事情。

用于获取部件的宏

除了使用**getPart()**方法，我们还可以使用宏SO_GET_PART()和SO_CHECK_PART()来获取工具包的部件。如果我们使用调试版本的Inventor库来编译程序，这两个宏还会为我们执行类型转换和类型检查的工作（如果我们使用的是优化版本的Inventor库，则不会执行类型检查）。

SO_GET_PART() 宏

SO_GET_PART()的语法如下所示：

SO_GET_PART(*kitContainingPart*, *partName*, *partClassName*);

这个宏将会为我们自动进行类型转换，相当于下面的代码：

(*partClassName* *) *kitContainingPart*->getPart(*partName*, TRUE);

因为在宏中，*makeIfNeeded* 参数设为 TRUE，所以工具包中如果不存在所需的部件，则部件会自动被创建。例如：

```
xf = SO_GET_PART(myKit, "transform", SoTransform);
```

SO_CHECK_PART()宏

SO_CHECK_PART()的语法如下所示：

SO_CHECK_PART(*kitContainingPart*, *partName*, *partClassName*);

这个宏将会为我们自动进行类型转换，相当于下面的代码：

(*partClassName* *) *kitContainingPart*->getPart(*partName*, FALSE);

因为在宏中，*makeIfNeeded* 参数设为 FALSE，所以工具包中如果不存在所需的部件，getPart（）会直接返回 NULL，而不会去创建那个部件。例如：

```
xf = SO_CHECK_PART(myKit, "transform", SoTransform);
if (xf == NULL)
   printf("Transform does not exist in myKit.");
else
   printf("Got it!");
```

指定部件名称

假设我们已经创建了如图 14-9 中所示的 3 个节点工具包类(在 *The Inventor Toolmaker* 第 7 章中讨论了如何子类化节点工具包)

- **SoGoonKit**，定义了一个完整的人。这个人由一个**SoAppearanceKit**、两个**SoLegKit**实例（分别表示*leg1* 和*leg2*）、和一个代表身体的**SoCone**组成。
- **SoLegKit**，定义了人的一条腿。这个类包含有一个**SoAppearanceKit**、一个**SoFootKit**、和一个代表大腿的**SoCylinder**。
- **SoFootKit**，定义了人的一只脚。这个类包含有一个**SoAppearanceKit**、一个代表*toe1*的**SoCube**、一个代表*toe2*的**SoCube**。

当创建完一个**SoGoonKit**（*myGoon*）实例后，我们可以这样请求部件。

```
myCube = SO_GET_PART(myGoon, "toe1", SoCube);
```

首先，Inventor 会在 *myGoon* 目录中查询 *toe1*。如果没有找到 *toe1*，同时在目录中有些子节点也是节点工具包，那么 Inventor 会在这些子节点工具包中继续寻找 *toe1*，直到最终返回找到的第一次匹配的节点。在本例中，第一次匹配应该发生在 *leg1* 的脚部件中。但是，如果我们想得到在 *leg2* 中的 *toe1*，我们就需要这样指定：

```
myCube = SO_GET_PART(myGoon, "leg2.toe1", SoCube);
```

上面的代码将返回在 *leg2* 中的 *toe1*，等同于 *leg2.foot.toe1*。

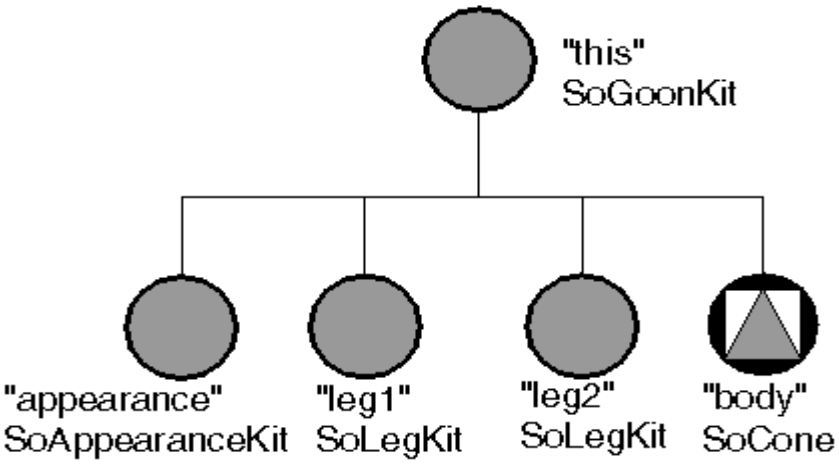
对于在目录中以列表方式定义的任意部件，我们可以通过索引方式获取它们－例如，“childList[0]”或“callbackList[2]”。

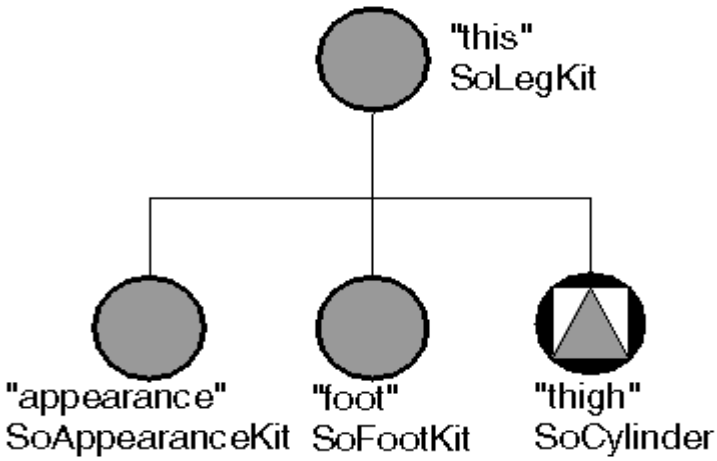
下面将举例说明使用三种不同的方法来创建节点工具包的部件以及设置它们的域值。下面的代码假设我们已经通过子类化的办法创建了一个从**SoBaseKit**派生出来的**SoGoonKit**工具包类（见*The Inventor Toolmaker* 第 7 章）。这个人具有身体、腿、和脚等器官。

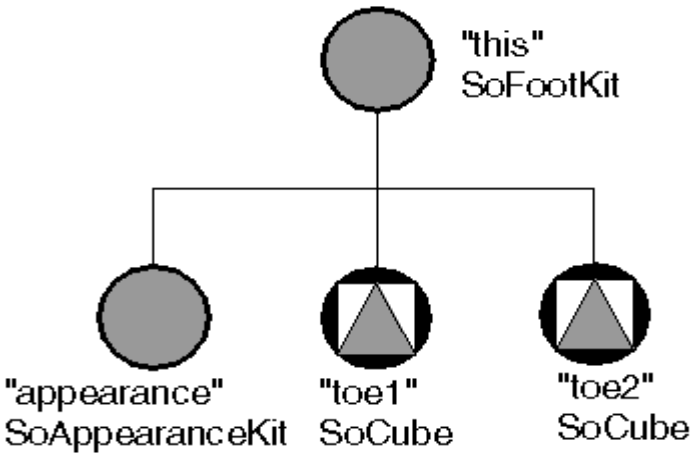
下面的代码段用来分别设置每个部件：

```
SoGoonKit *myGoon = new SoGoonKit();
myGoon->set("body.material", "diffuseColor [1 0 0 ]");
                                    // makes body red
myGoon->set("leg2.toe1", "width 2 height 3 depth 1");
                                    // creates toe with proper dimensions
```

图 14-9 组成人的三个节点工具包类


"this"
SoGoonKit
"appearance"
SoAppearanceKit
"leg1"
SoLegKit
"leg2"
SoLegKit
"body"
SoCone


"this"
SoLegKit
"appearance"
SoAppearanceKit
"foot"
SoFootKit
"thigh"
SoCylinder


"this"
SoFootKit
"appearance"
SoAppearanceKit
"toe1"
SoCube
"toe2"
SoCube

下面的代码将可以获取部件并编辑它们：

```
SoGoonKit *myGoon = new SoGoonKit();
SoMaterial *bodyMtl;
SoCube     *toe;
bodyMtl = SO_GET_PART(myGoon, "body.material", SoMaterial);
         bodyMtl->diffuseColor.setValue(1, 0, 0);
toe = SO_GET_PART(myGoon, "leg2.toe1", SoCube);
         toe->width.setValue(2);
         toe->height.setValue(3);
         toe->depth.setValue(1);
```

下面的代码将可以在一个命令中设置所有的部件：

```
SoGoonKit *myGoon = new SoGoonKit();
myGoon->set("body.material { diffuseColor [ 1 0 0 ] }
             leg2.toe1     { width 2
                             height 3
                             depth 1 }");
```

创建指向部件的路径

有时我们可能需要一条向下指向某个节点工具包部件的路径－例如，使用一个操作器（将在第 15 章中讨论）来取代工具包中的一个部件。可以使用**createPathToPart()**方法来获取指向操作器所要节点的路径。

```
createPathToPart(partName, makeIfNeeded, pathToExtend);
```

例如，在拾取了某个节点工具包后，使用跟踪球操作器来取代工具包中的几何变换部件：

```
SoPath *pickPath = myPickAction->getPath();

if((pickPath != NULL) &&
(pickPath->getTail()->isOfType(SoBaseKit::getClassTypeId())){
SoTrackballManip *tb = new SoTrackball;
SoBaseKit *kit = (SoBaseKit *) pickPath->getTail();
   // extends the pick path all the way down
                    // to the transform node
SoPath *attachPath = kit->createPathToPart("transform",
                                            TRUE, pickPath);
                 tb->replaceNode(attachPath);
```

值得注意的是，相对于简单地调用**setPart**方法，使用**replaceNode()**可以做更多的工作。

```
setPart("transform", tb)
```

域值将从已存在的“transform”部件中拷贝到跟踪球操作器的域中（Field values are copied from the existing “transform” part into the trackball manipulator’s fields）。

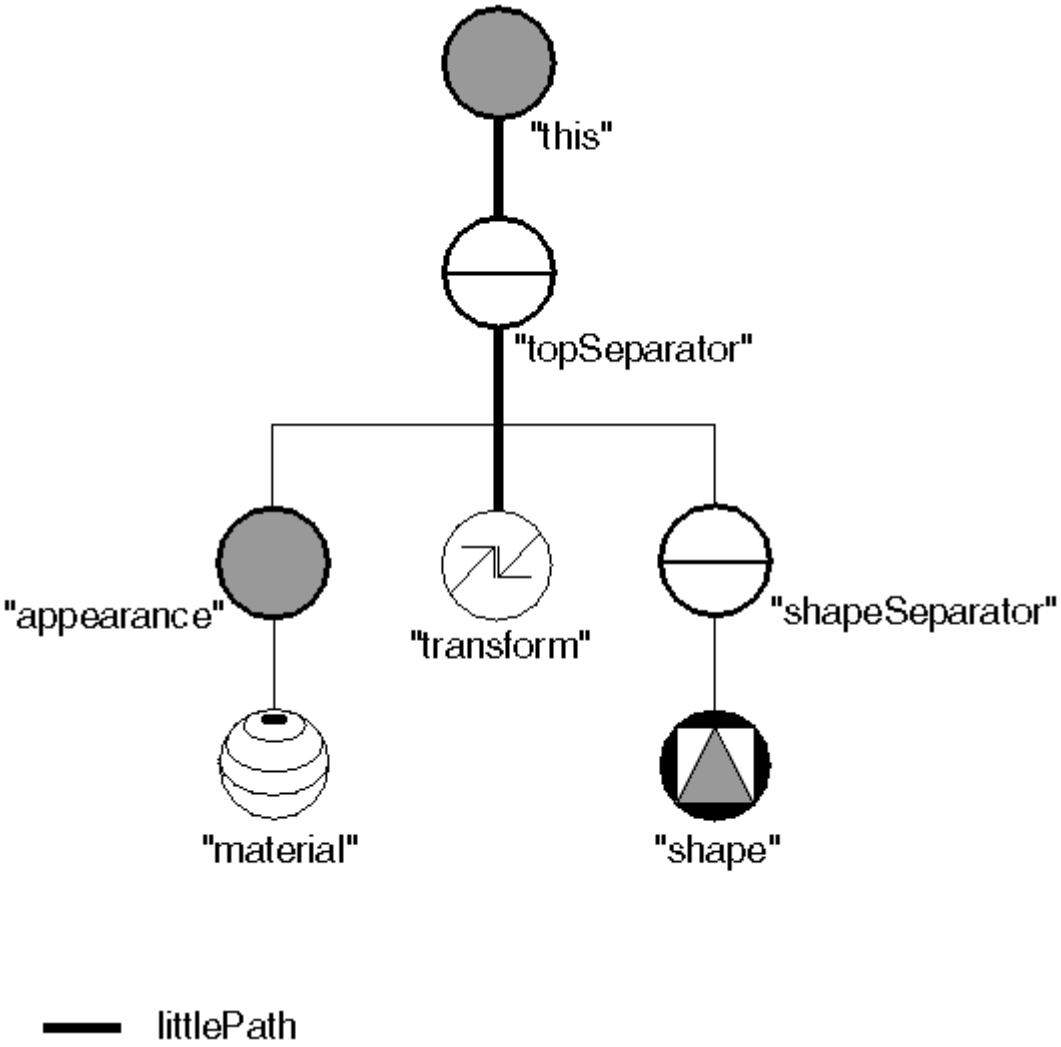
如果*pathToExtend*参数为NULL或不提供，那么**createPathToPart()**将只返回一条从节点工具包顶层节点到指定部件的路径（见图 14-10）。

```
SoPath *littlePath;
littlePath = myKit->createPathToPart("transform", TRUE);
```

在上面的代码中，因为 *makeIfNeeded* 参数为 TRUE，所以如果工具包中不存在“transform”部件，那么 Inventor 将会自动创建它。然而如果 *makeIfNeeded* 参数为 FALSE 并且不存在“transform”部件，那么 *createPathToPart* 将立即返回 NULL。

注意：如果希望查看包含隐藏子节点的全路径，一定要将**SoPath**转换成**SoFullPath**。

图 14-10 获取指向给定部件的路径



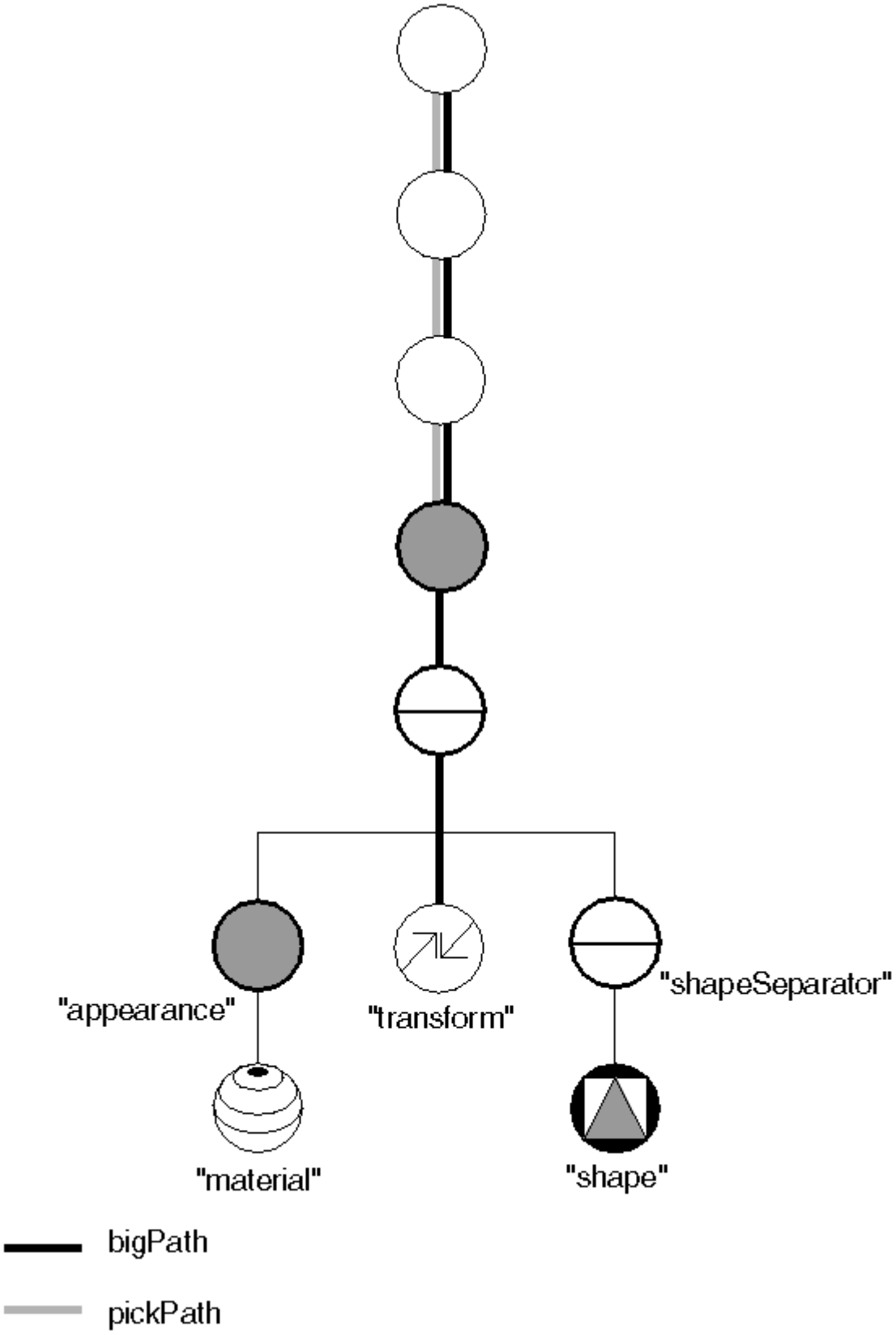
如果设置了*pathToExtend*参数，那么**createPathToPart()**将会沿着提供的路径向下扩展到节点工具包中指定的部件（这里指“transform”节点）。（见图 14-11）。如果作为输入参数提供的路径（这里指*pickPath*）没有包含指定的节点工具包，那么*bigPath*将等于NULL. 如果作为输入给定的路径（这里指*pickPath*）已经超过了指定的节点工具包，那么在将路径扩展到指定部件之前，需要首先截断剔除掉路径中节点工具包位置以后的部分。

```
bigPath = myKit->createPathToPart("transform", TRUE, pickPath);
```

如果要创建一条指向部件列表中某个子项目的路径，可以像我们在**setPart()**或**getPart()**那样，使用索引符号来指定到底使用那个子项。

```
pathToListElement = createPathToPart("callbackList[0]", TRUE);
```

图 14-11 将一条给定的路径扩展到所希望的部件上


"appearance"
"transform"
"shapeSeparator"
"material"
"shape"
bigPath
pickPath

使用部件列表

有些节点工具包的部件其实是一种*部件列表*（*lists* of parts）。列表的类型为**SoNodeKitListPart**，这是一种特殊的组类型（type of group）。节点工具包使用这些部件将它的子节点限制在某些节点类中。例如，像“childList”部件（可在**SoSeparatorKit**和**SoSceneKit**中找到）、“cameraList”、和“lightList”（都可以在**SoSceneKit**中找到）。无论何时，我们只要向节点工具包列表增加一个子节点时，这些列表组将会查看子节点是否是合法的。如果子节点不合法，那么就会拒绝将子节点增加到工具包中（如果我们使用的是调试版本的Inventor库，那么程序将会弹出一个错误提示）。

使用**getPart()**可获得所需要的列表，然后就可以使用任何标准的组节点方法向部件列表增加、删除、替换、和插入节点。（不过要记住，这些方法已经被重新定义了，这样就可以在增加子节点之前，先检查子节点的类型是否合法）。例如：

```
SoPointLight *myLight = new SoPointLight;
ls = (SoNodeKitListPart *) k->getPart("lightList", TRUE);

ls->addChild(myLight);
```

使用SoSeparatorKit创建多级运动场景（Using Separator Kits to Create Motion Hierarchies）

SoSeparatorKit是一种节点工具包类。所有从**SoSeparatorKit**派生出来的类都会继承了一个名为“childList”，类型为**SoNodeKitListPart**的部件。通过使用“childList”，**SoSeparatorKit**可以允许我们根据物体各个部分之间的相互运动关系来考虑场景结构（Through use of the “childList,” separator kits allow you to think in terms of how parts of an object move relative to each other）。“childList”的每个子项都可以是一个**SoSeparatorKit**，并且都可以包含自己的几何变换节点（Each element of the child list is, in turn, an **SoSeparatorKit** and may contain its own transform node）。通过嵌套使用**SoSeparatorKit**，我们就可以建立起一个多级相互运动的场景。

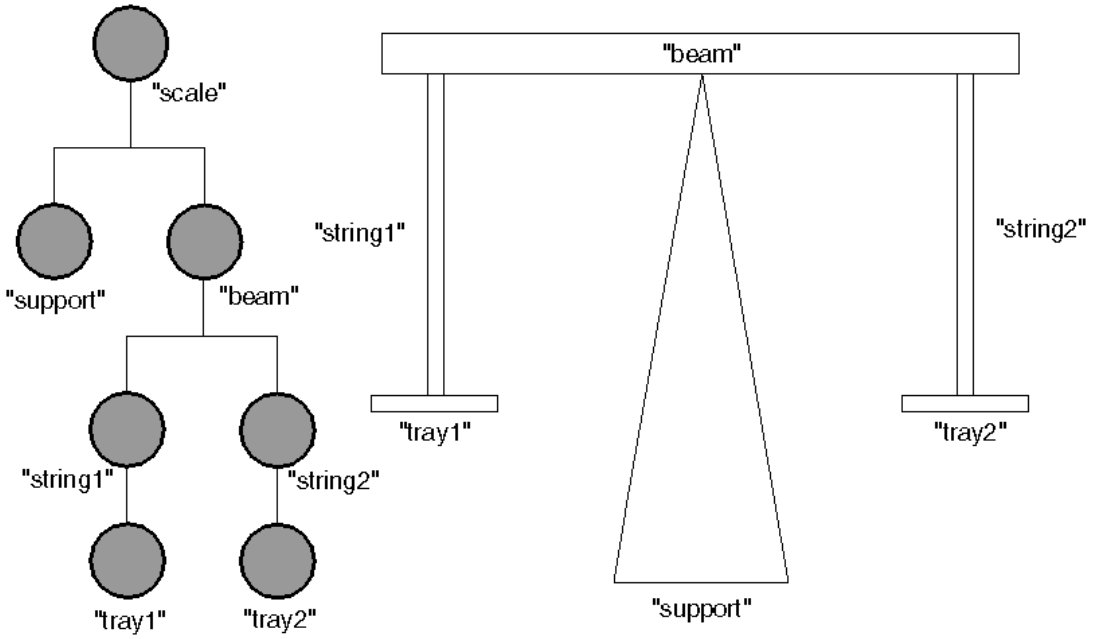
图 14-12 给出了如何将共同运动的各个部件聚合起来的场景图。假设我们使用一个**SoSeparatorKit**来定义天平中的各个部件。我们希望将*tray1* 和*string1* 作为一个整体来运动，将*tray2* 和*string2* 作为另一个整体来运动。但当天平横梁运动后，所有的托盘和天平臂都会随着天平横梁一起运动。

当将这些组工具包按照分层结构组织排列后，我们就不需要再考虑每个组工具包所包含的内容（“材质”、“复杂度”等内容）。我们只需要考虑这些物体本身（横梁、托盘、天平臂），以及它们之间的相对运动关系。**SoSeparatorKit**的**childList**可以包含任意多个从**SoSeparatorKit**派生出来的节点。所以任何**SoSeparatorKit**的类型都可以作为列表的一个条目。

下面的代码构造出显示在图 14-12 中的分层场景。在本章结尾部分的例 14-3 提供了这个天平模型可工作的版本。

```
scale->setPart("childList[0]", support);
scale->setPart("childList[1]", beam);
beam->setPart("childList[0]", string1);
beam->setPart("childList[1]", string2);
string1->setPart("childList[0]", tray1);
string2->setPart("childList[0]", tray2);
```

图 14-12 分层运动的相互关系



例子程序

本节包括节点工具包的 3 个例子程序。第一个例子将使用两个**SoShapeKits**。第二个例子将详细注释如何使用**SoWrapperKit**和一个包含了**SoLightKit**和**SoCameraKit**的**SoSceneKit**。第三个例子将使用多种节点工具包，并且利用一个带有回调函数的**SoEventCallback**节点来驱动天平。

节点工具包简单的使用方法

例 14-1 将使用节点工具包创建两个 3D 单词，然后使用节点工具包提供的方法，存取“material”和“transform”这两个域值。这个例子还将使用一个计算引擎和一个记时引擎，通过它们来修改 3D 单词的颜色，并驱使单词在屏幕上飞来飞去。图 14-13 给出了截取自这个例子的两幅屏幕拷贝。

例 14-1 节点工具包简单的使用方法

```
#include <Inventor/engines/SoCalculator.h>
#include <Inventor/engines/SoElapsedTime.h>
#include <Inventor/nodekits/SoShapeKit.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoText3.h>
#include <Inventor/nodes/SoTransform.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>

main(int , char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSeparator *root = new SoSeparator;
   root->ref();

   // Create shape kits with the words "HAPPY" and "NICE"
   SoShapeKit *happyKit = new SoShapeKit;
   root->addChild(happyKit);
   happyKit->setPart("shape", new SoText3);
   happyKit->set("shape { parts ALL string \"HAPPY\"}");
   happyKit->set("font { size 2}");

   SoShapeKit *niceKit = new SoShapeKit;
   root->addChild(niceKit);
   niceKit->setPart("shape", new SoText3);
   niceKit->set("shape { parts ALL string \"NICE\"}");
   niceKit->set("font { size 2}");

   // Create the Elapsed Time engine
   SoElapsedTime *myTimer = new SoElapsedTime;
   myTimer->ref();
```

图 14-3 使用一个带有引擎的 **SoShapeKit**

```
// Create two calculators - one for HAPPY, one for NICE.
SoCalculator *happyCalc = new SoCalculator;
happyCalc->ref();
happyCalc->a.connectFrom(&myTimer->timeOut);
happyCalc->expression = "ta=cos(2*a); tb=sin(2*a);\
   oA = vec3f(3*pow(ta,3),3*pow(tb,3),1);          \
   oB = vec3f(fabs(ta)+.1,fabs(.5*fabs(tb))+.1,1);\
   oC = vec3f(fabs(ta),fabs(tb),.5)";

// The second calculator uses different arguments to
// sin() and cos(), so it moves out of phase.
SoCalculator *niceCalc = new SoCalculator;
niceCalc->ref();
niceCalc->a.connectFrom(&myTimer->timeOut);
niceCalc->expression = "ta=cos(2*a+2); tb=sin(2*a+2);\
   oA = vec3f(3*pow(ta,3),3*pow(tb,3),1);            \
   oB = vec3f(fabs(ta)+.1,fabs(.5*fabs(tb))+.1,1);   \
   oC = vec3f(fabs(ta),fabs(tb),.5)";

// Connect the transforms from the calculators...
SoTransform *happyXf
   = (SoTransform *) happyKit->getPart("transform",TRUE);
happyXf->translation.connectFrom(&happyCalc->oA);
happyXf->scaleFactor.connectFrom(&happyCalc->oB);
SoTransform *niceXf
   = (SoTransform *) niceKit->getPart("transform",TRUE);
niceXf->translation.connectFrom(&niceCalc->oA);
niceXf->scaleFactor.connectFrom(&niceCalc->oB);

// Connect the materials from the calculators...
```

```
   SoMaterial *happyMtl
      = (SoMaterial *) happyKit->getPart("material",TRUE);
   happyMtl->diffuseColor.connectFrom(&happyCalc->oC);
   SoMaterial *niceMtl
      = (SoMaterial *) niceKit->getPart("material",TRUE);
   niceMtl->diffuseColor.connectFrom(&niceCalc->oC);

   SoXtExaminerViewer *myViewer = new
            SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(root);
   myViewer->setTitle("Frolicking Words");
   myViewer->viewAll();
   myViewer->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

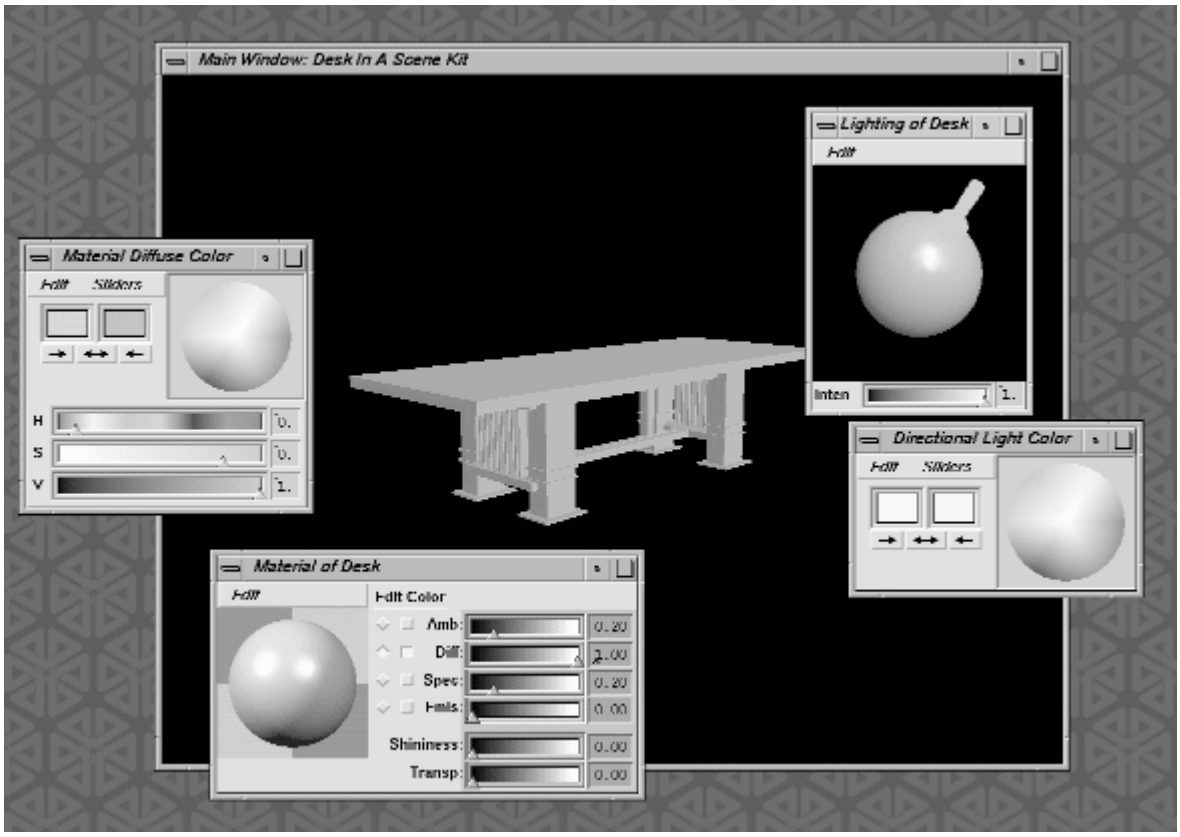
使用带有编辑器的节点工具包

例 14-2 将从一个文件中读入一个桌子模型，然后将桌子模型设置在 SoWrapperKit 的“contents”部件中。例子还有一个与场景光源相关联的射灯编辑器，一个与桌子模型相关联的材质编辑器。如图 14-14 所示。使用 SoSceneKit 来组织管理场景结构，SoSceneKit 包含有分组的光源列表（“lightList”）、照相机列表（“cameras”）、和物体列表（“childList”）。

例 14-2 使用带有编辑器的节点工具包

```
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/nodekits/SoCameraKit.h>
#include <Inventor/nodekits/SoLightKit.h>
#include <Inventor/nodekits/SoSceneKit.h>
#include <Inventor/nodekits/SoWrapperKit.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtDirectionalLightEditor.h>
#include <Inventor/Xt/SoXtMaterialEditor.h>
#include <Inventor/Xt/SoXtRenderArea.h>
```

图 14-14 使用带有射灯编辑器和材质编辑器的 **SoSceneKit**




```
main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   // SCENE!
   SoSceneKit *myScene = new SoSceneKit;
   myScene->ref();

   // LIGHTS! Add an SoLightKit to the "lightList." The
   // SoLightKit creates an SoDirectionalLight by default.
   myScene->setPart("lightList[0]", new SoLightKit);

   // CAMERA!! Add an SoCameraKit to the "cameraList." The
   // SoCameraKit creates an SoPerspectiveCamera by default.
   myScene->setPart("cameraList[0]", new SoCameraKit);
   myScene->setCameraNumber(0);

   // Read an object from file.
```

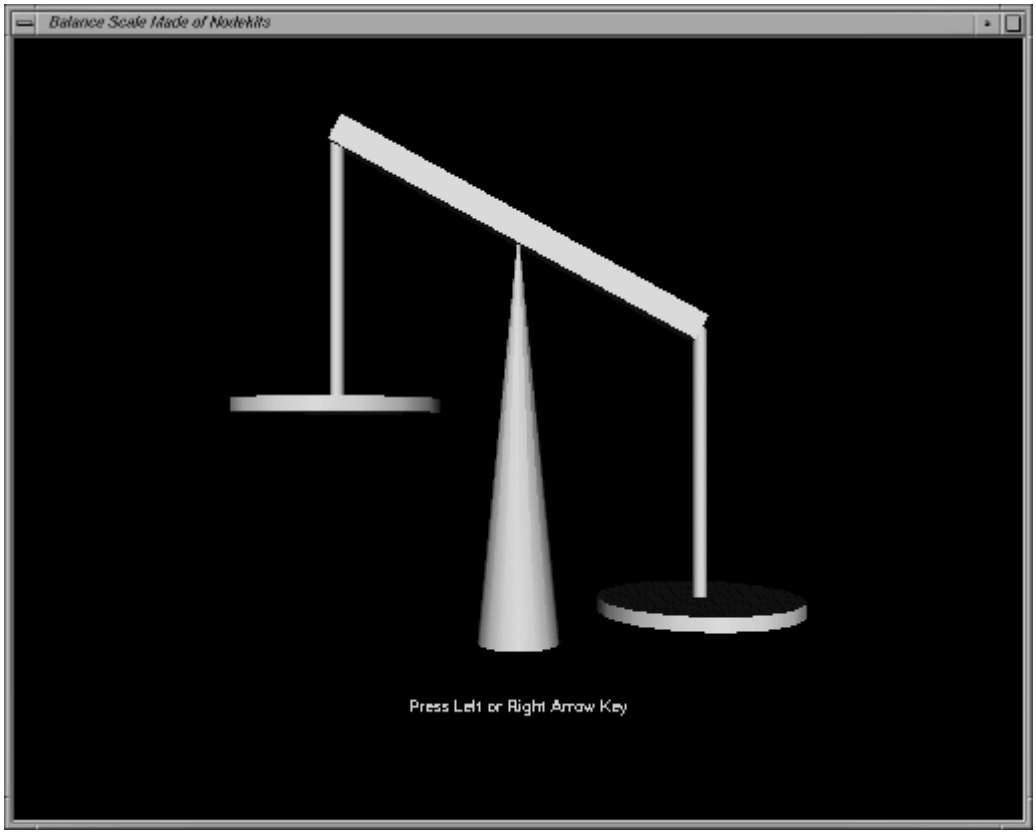
```
SoInput myInput;
if (!myInput.openFile("desk.iv"))
   return (1);
SoSeparator *fileContents = SoDB::readAll(&myInput);
if (fileContents == NULL) return (1);
// OBJECT!! Create an SoWrapperKit and set its contents to
// be what you read from file.
SoWrapperKit *myDesk = new SoWrapperKit();
myDesk->setPart("contents", fileContents);
myScene->setPart("childList[0]", myDesk);
// Give the desk a good starting color
myDesk->set("material { diffuseColor .8 .3 .1 }");

// MATERIAL EDITOR!!  Attach it to myDesk's material node.
// Use the SO_GET_PART macro to get this part from myDesk.
SoXtMaterialEditor *mtlEditor = new SoXtMaterialEditor();
SoMaterial *mtl = SO_GET_PART(myDesk,"material",SoMaterial);
mtlEditor->attach(mtl);
mtlEditor->setTitle("Material of Desk");
mtlEditor->show();

// DIRECTIONAL LIGHT EDITOR!! Attach it to the
// SoDirectionalLight node within the SoLightKit we made.
SoXtDirectionalLightEditor *ltEditor =
              new SoXtDirectionalLightEditor();
SoPath *ltPath = myScene->createPathToPart(
   "lightList[0].light", TRUE);
ltEditor->attach(ltPath);
ltEditor->setTitle("Lighting of Desk");
ltEditor->show();

SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

// Set up Camera with ViewAll...
// -- use the SO_GET_PART macro to get the camera node.
// -- viewall is a method on the 'camera' part of
//    the cameraKit, not on the cameraKit itself.  So the part
//    we ask for is not 'cameraList[0]' (which is of type
//    SoPerspectiveCameraKit), but
//    'cameraList[0].camera' (which is of type
//    SoPerspectiveCamera).
SoPerspectiveCamera *myCamera = SO_GET_PART(myScene,
   "cameraList[0].camera", SoPerspectiveCamera);
```

```
    SbViewportRegion myRegion(myRenderArea->getSize());
    myCamera->viewAll(mySCene, myRegion);
    myRenderArea->setSceneGraph(mySCene);
    myRenderArea->setTitle("Main Window: Desk In A Scene Kit");
    myRenderArea->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

创建多级运动场景

例 14-3 使用节点工具包创建了一个天平场景，并构造了它们的相互运动关系。图 14-15 给出了这个例子所产生的场景图。

图 14-15 使用节点工具包创建的天平模型



例 14-3 使用节点工具包创建多级运动场景

```
// This example illustrates the creation of motion hierarchies
// using nodekits by creating a model of a balance-style scale.
```

```
// It adds an SoEventCallback to the "callback" list in the
// nodekit called 'support.'
// The callback will have the following response to events:
// Pressing right arrow key == lower the right pan
// Pressing left arrow key  == lower the left pan
// The pans are lowered by animating three rotations in the
// motion hierarchy.
// Use an SoText2Kit to print instructions to the user as part
// of the scene.

#include <Inventor/events/SoKeyboardEvent.h>
#include <Inventor/nodekits/SoCameraKit.h>
#include <Inventor/nodekits/SoLightKit.h>
#include <Inventor/nodekits/SoSceneKit.h>
#include <Inventor/nodekits/SoShapeKit.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoCylinder.h>
#include <Inventor/nodes/SoEventCallback.h>
#include <Inventor/nodes/SoText2.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>

// Callback Function for Animating the Balance Scale.
// --used to make the balance tip back and forth
// --Note: this routine is only called in response to KeyPress
//   events since the call 'setEventInterest(KeyPressMask)' is
//   made on the SoEventCallback node that uses it.
// --The routine checks if the key pressed was left arrow (which
//   is XK_Left in X-windows talk), or right arrow (which is
//   XK_Right)
// --The balance is made to tip by rotating the beam part of the
//   scale (to tip it) and then compensating (making the strings
//   vertical again) by rotating the string parts in the opposite
//   direction.
void
tipTheBalance(
   void *userData, // The nodekit representing 'support', the
                   // fulcrum of the balance. Passed in during
                   // main routine, below.
```

```
   SoEventCallback *eventCB)
{
   const SoEvent *ev = eventCB->getEvent();

   // Which Key was pressed?
   // If Right or Left Arrow key, then continue...
   if (SO_KEY_PRESS_EVENT(ev, RIGHT_ARROW) ||
        SO_KEY_PRESS_EVENT(ev, LEFT_ARROW)) {
      SoShapeKit  *support, *beam1, *string1, *string2;
      SbRotation  startRot, beamIncrement, stringIncrement;

      // Get the different nodekits from the userData.
      support = (SoShapeKit *) userData;

    // These three parts are extracted based on knowledge of
    // the motion hierarchy (see the diagram in the main
    // routine.
      beam1   = (SoShapeKit *)support->getPart("childList[0]",TRUE);
      string1 = (SoShapeKit *)  beam1->getPart("childList[0]",TRUE);
      string2 = (SoShapeKit *)  beam1->getPart("childList[1]",TRUE);

      //Set angular increments to be .1 Radians about the Z-Axis
      //The strings rotate opposite the beam, and the two types
      //of key press produce opposite effects.
      if (SO_KEY_PRESS_EVENT(ev, RIGHT_ARROW)) {
         beamIncrement.setValue(SbVec3f(0, 0, 1), -.1);
         stringIncrement.setValue(SbVec3f(0, 0, 1), .1);
      }
      else {
         beamIncrement.setValue(SbVec3f(0, 0, 1), .1);
         stringIncrement.setValue(SbVec3f(0, 0, 1), -.1);
      }

      // Use SO_GET_PART to find the transform for each of the
      // rotating parts and modify their rotations.

      SoTransform *xf;
      xf = SO_GET_PART(beam1, "transform", SoTransform);
      startRot = xf->rotation.getValue();
      xf->rotation.setValue(startRot *  beamIncrement);

      xf = SO_GET_PART(string1, "transform", SoTransform);
      startRot = xf->rotation.getValue();
```

```
      xf->rotation.setValue(startRot *  stringIncrement);

      xf = SO_GET_PART(string2, "transform", SoTransform);
      startRot = xf->rotation.getValue();
      xf->rotation.setValue(startRot *  stringIncrement);

      eventCB->setHandled();
   }
}

main(int , char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   SoSceneKit *myScene = new SoSceneKit;
   myScene->ref();

   myScene->setPart("lightList[0]", new SoLightKit);
   myScene->setPart("cameraList[0]", new SoCameraKit);
   myScene->setCameraNumber(0);

   // Create the Balance Scale -- put each part in the
   // childList of its parent, to build up this hierarchy:
   //
   //                    myScene
   //                       |
   //                     support
   //                       |
   //                     beam
   //                       |
   //                   --------
   //                   |       |
   //                string1  string2
   //                   |       |
   //                tray1     tray2

   SoShapeKit *support = new SoShapeKit();
   support->setPart("shape", new SoCone);
   support->set("shape { height 3 bottomRadius .3 }");
   myScene->setPart("childList[0]", support);

   SoShapeKit *beam = new SoShapeKit();
```

```
beam->setPart("shape", new SoCube);
beam->set("shape { width 3 height .2 depth .2 }");
beam->set("transform { translation 0 1.5 0 } ");
support->setPart("childList[0]", beam);

SoShapeKit *string1 = new SoShapeKit;
string1->setPart("shape", new SoCylinder);
string1->set("shape { radius .05 height 2}");
string1->set("transform { translation -1.5 -1 0 }");
string1->set("transform { center 0 1 0 }");
beam->setPart("childList[0]", string1);

SoShapeKit *string2 = new SoShapeKit;
string2->setPart("shape", new SoCylinder);
string2->set("shape { radius .05 height 2}");
string2->set("transform { translation 1.5 -1 0 } ");
string2->set("transform { center 0 1 0 } ");
beam->setPart("childList[1]", string2);

SoShapeKit *tray1 = new SoShapeKit;
tray1->setPart("shape", new SoCylinder);
tray1->set("shape { radius .75 height .1 }");
tray1->set("transform { translation 0 -1 0 } ");
string1->setPart("childList[0]", tray1);

SoShapeKit *tray2 = new SoShapeKit;
tray2->setPart("shape", new SoCylinder);
tray2->set("shape { radius .75 height .1 }");
tray2->set("transform { translation 0 -1 0 } ");
string2->setPart("childList[0]", tray2);

// Add EventCallback so Balance Responds to Events
SoEventCallback *myCallbackNode = new SoEventCallback;
myCallbackNode->addEventCallback(
     SoKeyboardEvent::getClassTypeId(),
               tipTheBalance, support);
support->setPart("callbackList[0]", myCallbackNode);

// Add Instructions as Text in the Scene...
SoShapeKit *myText = new SoShapeKit;
myText->setPart("shape", new SoText2);
myText->set("shape { string \"Press Left or Right Arrow Key\" }");
myText->set("shape { justification CENTER }");
```

```
   myText->set("font { name \"Helvetica-Bold\" }");
   myText->set("font { size 16.0 }");
   myText->set("transform { translation 0 -2 0 }");
   myScene->setPart("childList[1]", myText);

   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);

   // Get camera from scene and tell it to viewAll...
   SbViewportRegion myRegion(myRenderArea->getSize());
   SoPerspectiveCamera *myCamera = SO_GET_PART(myScene,
      "cameraList[0].camera", SoPerspectiveCamera);
   myCamera->viewAll(myScene, myRegion);

   myRenderArea->setSceneGraph(myScene);
   myRenderArea->setTitle("Balance Scale Made of Nodekits");
   myRenderArea->show();

   SoXt::show(myWindow);
   SoXt::mainLoop();
}
```

第十五章 拖拽器和操作器

本章目标

在阅读完本章节后，我们将有能力做下列事情：

- 将拖拽器与场景中的域或引擎相连接。
- 解释拖拽器与操作器之间的区别。
- 能够编写当交互操作开始或结束时、鼠标移动时、或拖拽器的某个域值发生变化时被调用的回调函数。
- 在应用程序中允许用户利用操作器直接编辑场景中的节点。
- 定制拖拽器的外观。

本章将讨论如何使用 *拖拽器* 和 *操作器* 对象，这两种对象是一种带有用户界面且可以响应事件的特殊对象。像手柄盒操作器（handle box）、轨迹球操作器（trackball）、方向光源操作器（directional light）这些对象，都是利用拖拽器来和用户进行交互操作以及编辑处理。有关拖拽器是如何接收与响应事件的内容，请阅读本书第 10 章内容。

什么是拖拽器？

拖拽器是场景中具有响应用户输入事件能力的一种节点。所有的 Inventor 拖拽器都内建有一个用户界面。拖拽器会在场景中插入一个用于拾取和用户反馈的几何物体（**这个几何物体就是拖拽器的用户界面。译者注**）。图 15-1 给出了拖拽器的类树图。

拖拽器的类型

所有的拖拽器类都是从**SoDragger**派生出来的，用户可以利用鼠标的点击-拖动-释放动作来操作拖拽器。表 15-1 列出了所有从**SoDragger**派生出来的拖拽器的用途。例如，SoDragPointDragger是通过在三维空间中做平移运动的方式来响应用户的拖拽。

SoDragger的子类分成两大类：简单拖拽器（*simple* draggers）和复合拖拽器（*compound* draggers）。通常，简单拖拽器只能执行一种操作，例如像放大、平移操作等。复合拖拽器是由多个简单拖拽器组合而成的，因此可以执行多种操作。简单拖拽器通常可用于下面的三个方面：

- 我们可将简单拖拽器的域和场景中的其它域或引擎连接起来。这是场景中物体之间建立相互依赖关系的一种简单方式。
- 我们可以编写当交互操作开始或结束时、鼠标移动时、或拖拽器的某个域值发生改变时被调用的回调函数。
- 我们可以利用简单拖拽器作为创建更复杂拖拽器的一种基础模块。

复合拖拽器除了具有更多的部件之外，其实与简单拖拽器是非常类似的。复合拖拽器是

由多个简单拖拽器组合而成的。例如，**SoTransformBoxDragger**拖拽器就是由一个缩放拖拽器、三个旋转拖拽器、和六个平移拖拽器组合而成的。

图 15-1 拖拽器类

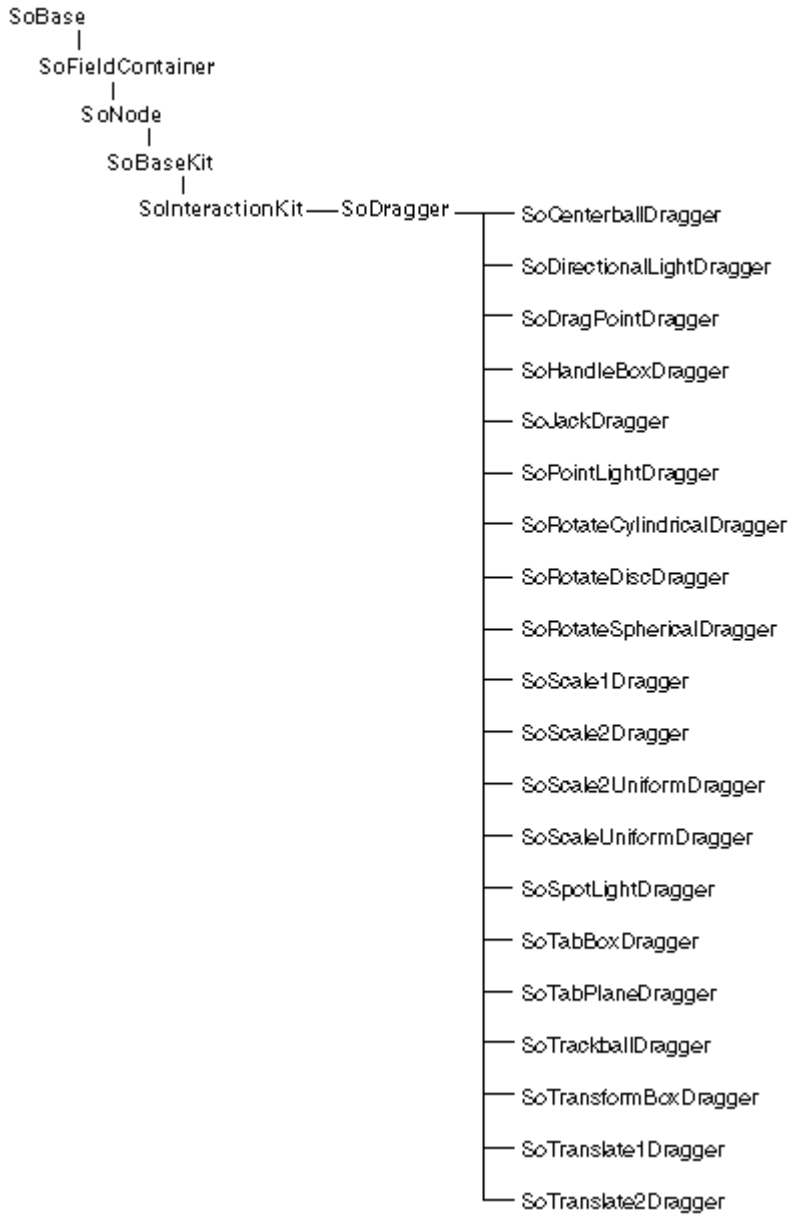


表 15-1 拖拽器的用途

拖拽器	用途
SoCenterballDragger	旋转，定位中心位置
SoDirectionalLightDragger	旋转
SoDragPointDragger	平移
SoHandleBoxDragger	平移，缩放
SoJackDragger	旋转，平移，等比缩放(三维)

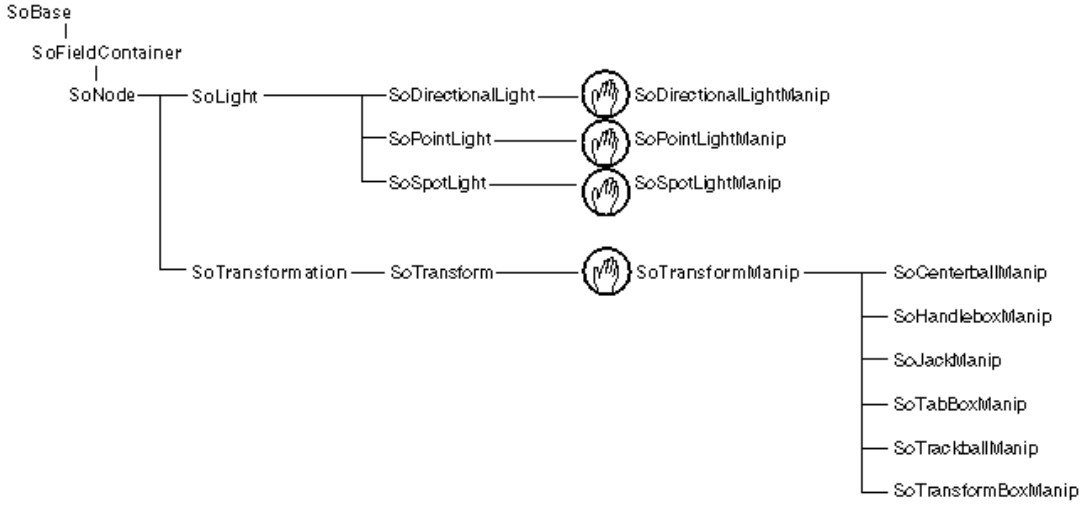
拖拽器	用途
SoPointLightDragger	平移
SoRotateCylindricalDragger	旋转
SoRotateDiscDragger	旋转
SoRotateSphericalDragger	旋转
SoScale1Dragger	缩放(一维)
SoScale2Dragger	缩放(二维)
SoScaleUniformDragger	等比缩放(三维)
SoScale2UniformDragger	等比缩放(二维)
SoSpotLightDragger	平移, 旋转, 修改锥角
SoTabBoxDragger	缩放, 平移
SoTabPlaneDragger	缩放(二维), 平移(二维)
SoTrackballDragger	旋转, 缩放
SoTransformBoxDragger	旋转, 平移, 缩放
SoTranslate1Dragger	平移(一维)
SoTranslate2Dragger	平移(二维)

操作器 VS 拖拽器

操作器是某些节点（SoTransform或SoDirectionalLight节点）的子类，它利用拖拽器（作为隐藏子节点）来响应用户事件以及修改自身的数据。图 15-2 给出了部分操作器类树图。每个操作器都含有可以直接响应用户事件以及修改操作器域值的拖拽器。操作器会在场景中插入一个几何形体，以此为用户提供反馈界面；这个几何形体是由操作器中的拖拽器提供的。例如，**SoHandleBoxManip**操作器会在场景中添加一些立方体和线段，用户通过在这些几何形体上点击移动鼠标来修改**SoTransform**节点的缩放和平移域值（见图 15-3）。这些几何形体是**SoHandleBoxManip**操作器所包含的**SoHandleBoxDragger**拖拽器中的一部分。**SoTrackballManip**操作器则是在场景中添加上一个由三个带状条组成的球体，用户可以通过操作这个球体来编辑修改**SoTransform**节点的旋转数据域（见图 15-4）。由此用户就可以旋转和缩放位于轨迹球中的物体。

拖拽器在响应用户事件的时候只是移动其自身。操作器不仅移动自身，而且还会影响场景中其它的物体。这是因为操作器是**SoTransform**或**SoLight**的子类，所以它也可以作为一个几何变换或光源节点来看待，也可以修改遍历状态数据。拖拽器为操作器提供了几何形体和用户界面。操作器使用从拖拽器接收过来的数据，并将这些数据拷贝到自己域中。当交互操作结束后，操作器就可以从场景中删除掉，这时操作器会将自己的域值拷贝到被它替换的几何变换或光源节点的域中。

图 15-2 操作器类树



操作器会将场景中的节点替换成另外一种可编辑的节点。当交互操作结束后，将恢复使用原来的节点（即不可编辑的节点）。每个操作器都包含有一个允许用户编辑其域值的拖拽器。下面是从**SoTransform**派生出来的操作器：

SoCenterBallManip
SoHandleBoxManip
SoJackManip
SoTabBoxManip
SoTrackballManip
SoTransformBoxManip

其它的操作器还包括从**SoPointLight**派生出来的**SoPointLightManip**，和从**SoDirectionalLight**派生出来的**SoDirectionalLightManip**。

图 15-3 手柄盒操作器

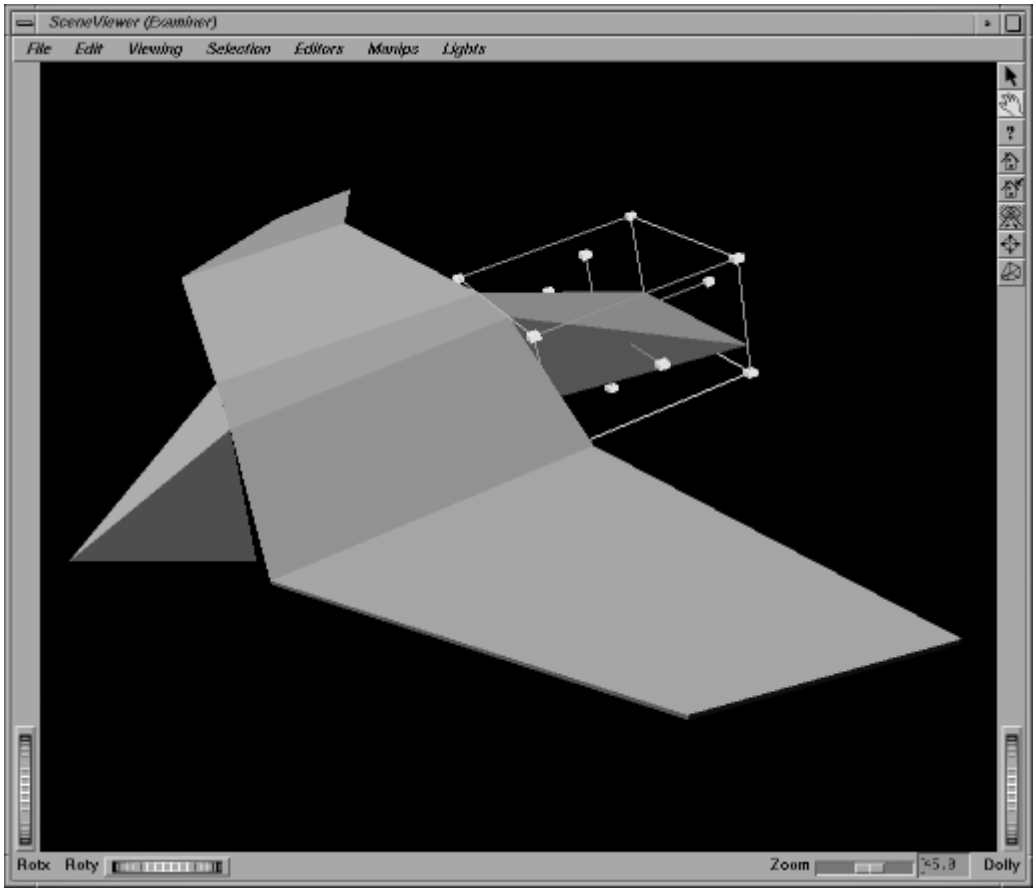
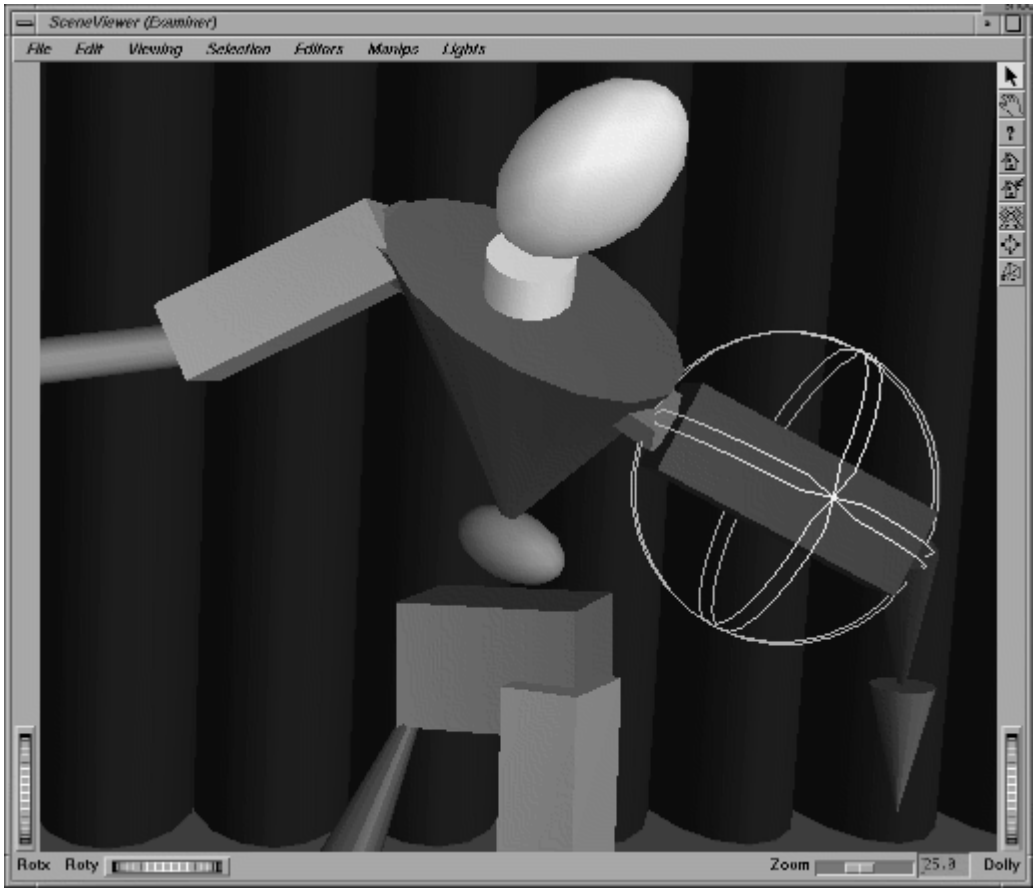

SceneViewer (Examiner)
File
Edit
Viewing
Selection
Editors
Manips
Lights
Rotx
Roty
Zoom
Dolly

图 15-4 轨迹球操作器



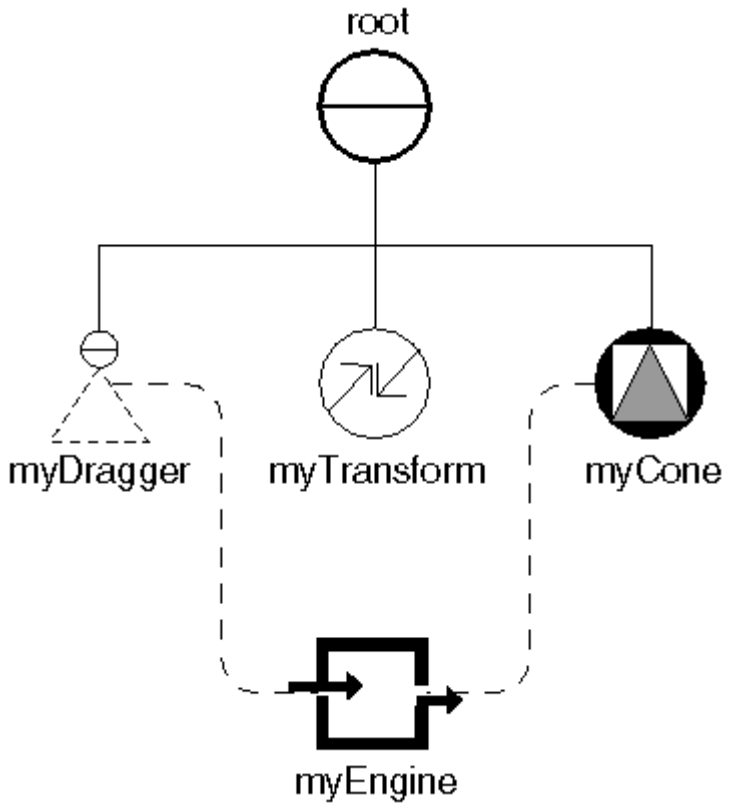
简单拖拽器

简单拖拽器可以根据鼠标的点击-拖动-释放事件在 3D 空间中做相应的运动。它在场景层次结构中的位置决定了它在 3D 空间中的位置。每个简单拖拽器都带有一个可以反映其当前状态的域。例如，SoRotateDiscDragger 拖拽器带有一个用于指示当前旋转量的旋转域。这个域可以连接到场景中其它的域上，或者也可以连接到一个引擎的输入域上（见下节内容）。简单拖拽器也可以使用回调函数，见“回调函数”章节。

域连接

使用拖拽器的一个简便方法是将它的域与场景中其它的域或引擎相连接。例如，**SoTranslate1Dragger**拖拽器带有一个具有多种用途的**translation**域。图 15-5 显示了如何使用这个域来编辑修改圆锥节点的半径。因为拖拽器的**translation**域是一个类型为**SoSFVec3f**的数据，所以我们必须使用**SoDecomposeVec3f**引擎将拖拽器平移数据中的x轴平移数据提取出来。然后这个x轴平移数据设置到圆锥节点的**bottomRadius**域上。这样，只要一移动拖拽器，圆锥的半径就会做相应的变化。

图 15-5 一个拖拽器的域盒场景中的其它域相连接

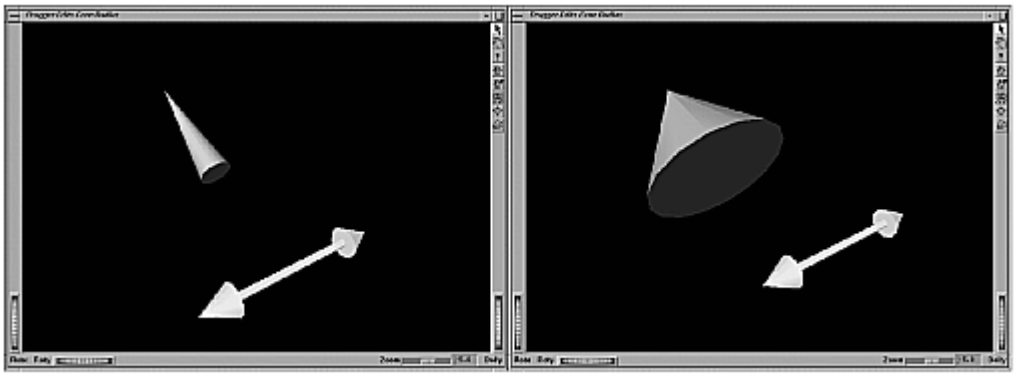


例 15-1 将演示连接这些域和引擎的代码。图 15-6 显示了这个例子创建的场景。

例 15-1 使用简单拖拽器

```
// Create myDragger with an initial translation of (1,0,0)
SoTranslate1Dragger *myDragger = new SoTranslate1Dragger;
root->addChild(myDragger);
myDragger->translation.setValue(1,0,0);

// Place an SoCone below myDragger
SoTransform *myTransform = new SoTransform;
SoCone *myCone = new SoCone;
root->addChild(myTransform);
root->addChild(myCone);
myTransform->translation.setValue(0,3,0);

// SoDecomposeVec3f engine extracts myDragger's x-component
// The result is connected to myCone's bottomRadius.
SoDecomposeVec3f *myEngine = new SoDecomposeVec3f;
myEngine->vector.connectFrom(&myDragger->translation);
myCone->bottomRadius.connectFrom(&myEngine->x);
```

图 15-6 使用拖拽器和引擎来修改一个圆锥的半径



回调函数

任何拖拽器和操作器都可以利用回调函数将其数据回传给应用程序。这种回调机制可用于增强拖拽器或操作器的缺省功能。当构造拖拽器对象时，构造函数会自动创建若干个类型为**SoCallbackList**的列表变量，这些列表变量用来保存回调函数和与回调函数相关联的数据。我们可以对这些列表增加或删除回调函数，可以将一个指向用户回调数据的指针作为参数传递进回调函数中。拖拽器使用下列回调函数的列表：

- 起始响应回调函数－当操作开始时被调用。
- 运动响应回调函数－在操作过程中，每次鼠标移动时被调用。
- 数据变化回调函数－当拖拽器的域值发生变化后被调用。
- 结束响应回调函数－当操作结束后被调用。

下列方法可以向回调函数列表增加和删除回调函数：

addStartCallback(*functionName*, *userData*)
removeStartCallback(*functionName*, *userData*)

addMotionCallback(*functionName*, *userData*)
removeMotionCallback(*functionName*, *userData*)

addValueChangedCallback(*functionName*, *userData*)
removeValueChangedCallback(*functionName*, *userData*)

addFinishCallback(*functionName*, *userData*)
removeFinishCallback(*functionName*, *userData*)

上面这些函数都是**SoDragger**的成员函数。如果想对操作器也调用这些函数，必须首先调用操作器的**getDragger()**函数，然后在调用这些回调列表函数。

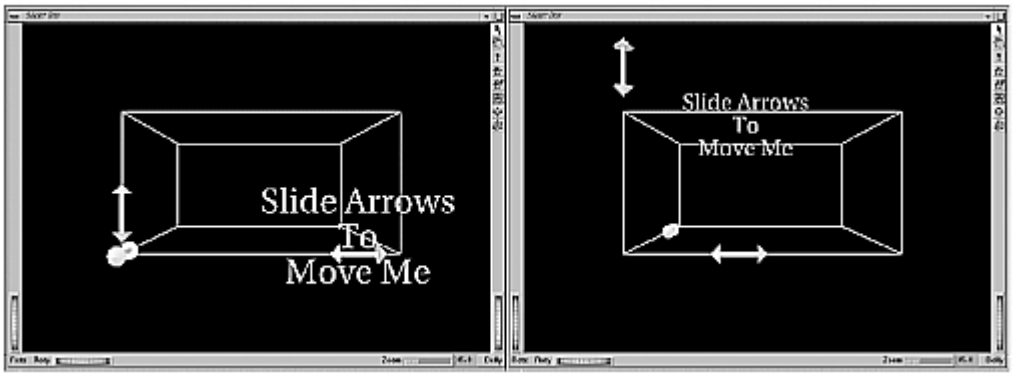
例如，我们可以编写一个起始响应回调函数，当操作开始的时候，利用这个函数将物体的绘制风格变成框线风格的。我们可以再编写一个结束响应回调函数，当操作结束后，将物

体的绘制风格再变回原来的填充风格。我们还可以编写数据变化回调函数，当数据发生变化时可以得到通知，然后使用**getValue()**方法来获得域的新值。

使用多个拖拽器

例 15-2 将使用三个**translate1Draggers**拖拽器修改一个几何平移节点的x,y,z数据，这个几何平移节点可以影响到场景中 3D文字的位置。图 15-7 显示了这个例子所产生的场景。

图 15-7 使用一个由拖拽器和引擎组成滑动盒来移动文字



例 15-2 使用多个拖拽器

```
//  Uses 3 translate1Draggers to change the x, y, and z
//  components of a translation. A calculator engine assembles
//  the components.
//  Arranges these draggers along edges of a box containing the
//  3D text to be moved.
//  The 3D text and the box are made with SoShapeKits

#include <Inventor/engines/SoCalculator.h>
#include <Inventor/nodekits/SoShapeKit.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoText3.h>
#include <Inventor/nodes/SoTransform.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>

#include <Inventor/draggers/SoTranslate1Dragger.h>

main(int , char **argv)
{
   Widget myWindow = SoXt::init(argv[0]);
```

```
if (myWindow == NULL) exit(1);

SoSeparator *root = new SoSeparator;
root->ref();

// Create 3 translate1Draggers and place them in space.
SoSeparator *xDragSep = new SoSeparator;
SoSeparator *yDragSep = new SoSeparator;
SoSeparator *zDragSep = new SoSeparator;
root->addChild(xDragSep);
root->addChild(yDragSep);
root->addChild(zDragSep);
// Separators will each hold a different transform
SoTransform *xDragXf = new SoTransform;
SoTransform *yDragXf = new SoTransform;
SoTransform *zDragXf = new SoTransform;
xDragXf->set("translation 0 -4 8");
yDragXf->set("translation -8 0 8 rotation 0 0 1 1.57");
zDragXf->set("translation -8 -4 0 rotation 0 1 0 -1.57");
xDragSep->addChild(xDragXf);
yDragSep->addChild(yDragXf);
zDragSep->addChild(zDragXf);

// Add the draggers under the separators, after transforms
SoTranslate1Dragger *xDragger = new SoTranslate1Dragger;
SoTranslate1Dragger *yDragger = new SoTranslate1Dragger;
SoTranslate1Dragger *zDragger = new SoTranslate1Dragger;
xDragSep->addChild(xDragger);
yDragSep->addChild(yDragger);
zDragSep->addChild(zDragger);

// Create shape kit for the 3D text
// The text says 'Slide Arrows To Move Me'
SoShapeKit *textKit = new SoShapeKit;
root->addChild(textKit);
SoText3 *myText3 = new SoText3;
textKit->setPart("shape", myText3);
myText3->justification = SoText3::CENTER;
myText3->string.set1Value(0,"Slide Arrows");
myText3->string.set1Value(1,"To");
myText3->string.set1Value(2,"Move Me");
textKit->set("font { size 2}");
textKit->set("material { diffuseColor 1 1 0}");
```

```
    // Create shape kit for surrounding box.
    // It's an unpickable cube, sized as (16,8,16)
    SoShapeKit *boxKit = new SoShapeKit;
    root->addChild(boxKit);
    boxKit->setPart("shape", new SoCube);
    boxKit->set("drawStyle { style LINES }");
    boxKit->set("pickStyle { style UNPICKABLE }");
    boxKit->set("material { emissiveColor 1 0 1 }");
    boxKit->set("shape { width 16 height 8 depth 16 }");

    // Create the calculator to make a translation
    // for the text. The x component of a translate1Dragger's
    // translation field shows how far it moved in that
    // direction. So our text's translation is:
    // (xDragTranslate[0],yDragTranslate[0],zDragTranslate[0])
    SoCalculator *myCalc = new SoCalculator;
    myCalc->ref();
    myCalc->A.connectFrom(&xDragger->translation);
    myCalc->B.connectFrom(&yDragger->translation);
    myCalc->C.connectFrom(&zDragger->translation);
    myCalc->expression = "oA = vec3f(A[0],B[0],C[0])";

    // Connect the the translation in textKit from myCalc
    SoTransform *textXf = (SoTransform *)
             textKit->getPart("transform",TRUE);
    textXf->translation.connectFrom(&myCalc->oA);

    SoXtExaminerViewer *myViewer = new
             SoXtExaminerViewer(myWindow);
    myViewer->setSceneGraph(root);
    myViewer->setTitle("Slider Box");
    myViewer->viewAll();
    myViewer->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

操作器

我们可以在应用程序中以不同的方式来使用操作器：

- 我们可以使用**replaceNode()**方法，将场景中某些类型的节点替换成对应的可编辑版本的操作器。当用户完成了对节点的操作后，再使用**replaceManip()**方法将原来的节点恢复到场景中。
- 我们可以编写自定义回调函数来使用操作器的域值。这些回调函数（将在后面的“回调函数”章节中讨论）可被用于所有类型的操作器（请回想前面讨论的内容，这些回调函数都是属于拖拽器的。所以在使用它们之前，需要调用getDragger()方法）。

我们可以在应用程序中同时使用这两种技术。例如，我们可以使用**replaceNode()**方法将一个**SoTransform**节点替换成操作器节点。然后当操作器的域值发生变化时，再利用“数据变化回调函数”来通知应用程序，这样应用程序就可以按照自己的意愿来使用新域值了。

下面的章节将会更加详细地讨论这两种技术。

节点替换成操作器

要在应用程序中使用操作器，必须遵照下面的必要步骤：

1. 构造操作器的实例
2. 如果计划重用这个操作器实例，则需要引用它一次。
3. 将场景中的某个节点替换为操作器替。像“手柄盒”和“轨迹球”这些从**SoTransform**派生出来的操作器可以替换**SoTransform**节点。同理，**SoDirectionalLightManip**可以替换**SoDirectionalLight**节点，**SoPointLightManip**可以替换**SoPointLight**节点，等等。

替换节点

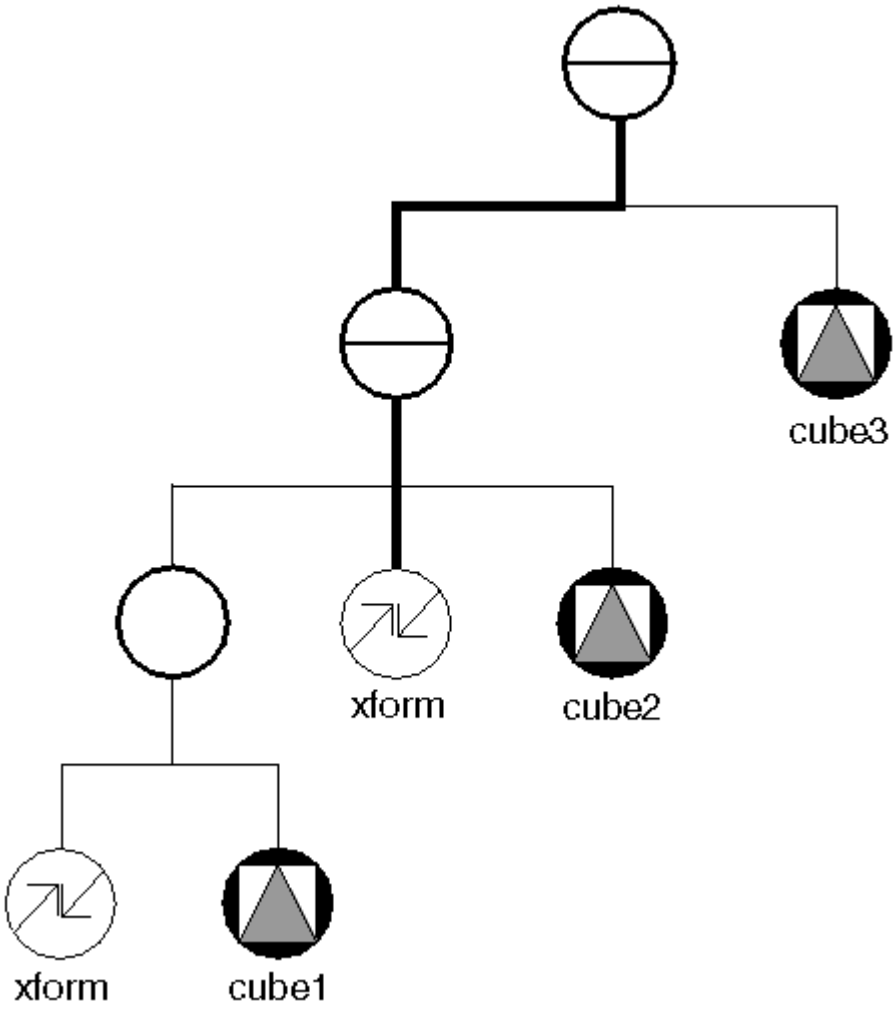
replaceNode()方法需要一个路径作为参数：

```
replaceNode(SoPath *p)
```

路径参数需要应用程序提供。例如，在图 15-8 中显示的一条指向**SoTransform**节点的路径，当一个变换类操作器替换了此节点后，编辑修改操作器将会影响到场景中*cube2* 节点的外观或位置。

派生于**SoTransformManip**的操作器使用一些特定的节点来构建其外观形体（例如“轨迹球”的外观总是一个球体），并且操作器可以确保自身的形体总是可以将被影响到的物体包围住。具体的这些特定节点在*The Inventor Toolmaker*有描述。

图 15-8 指定目标节点的路径



移除操作器

从场景中移除一个操作器：

1. 利用**replaceManip()**方法将原始节点恢复到场景中，在上面那个例子中，这个方法同时还会将操作器的域值拷贝到几何变换节点的域中
2. 调用操作器的**unref()**方法，删除操作器。

因为操作器的**replaceManip()**和**replaceNode()**这两种方法会将新节点替换到给定路径的尾节点上，所以我们可以在以后调用这些方法时重用那个路径。

例如：如果我们这样开始替换节点：

```
myManip = new SoTrackballManip;
myPathToTransform = createPathtoTransform(pickPath);
```

然后，我们调用：

```
myManip->replaceNode(myPathToTransform);
```

将操作器替换到路径的结尾处。

最后，我们调用：

```
myManip->replaceManip(myPathToTransform, new SoTransform);
```

来移走操作器，并使用一个新的几何变换节点替换它。

使用 replaceNode()方法

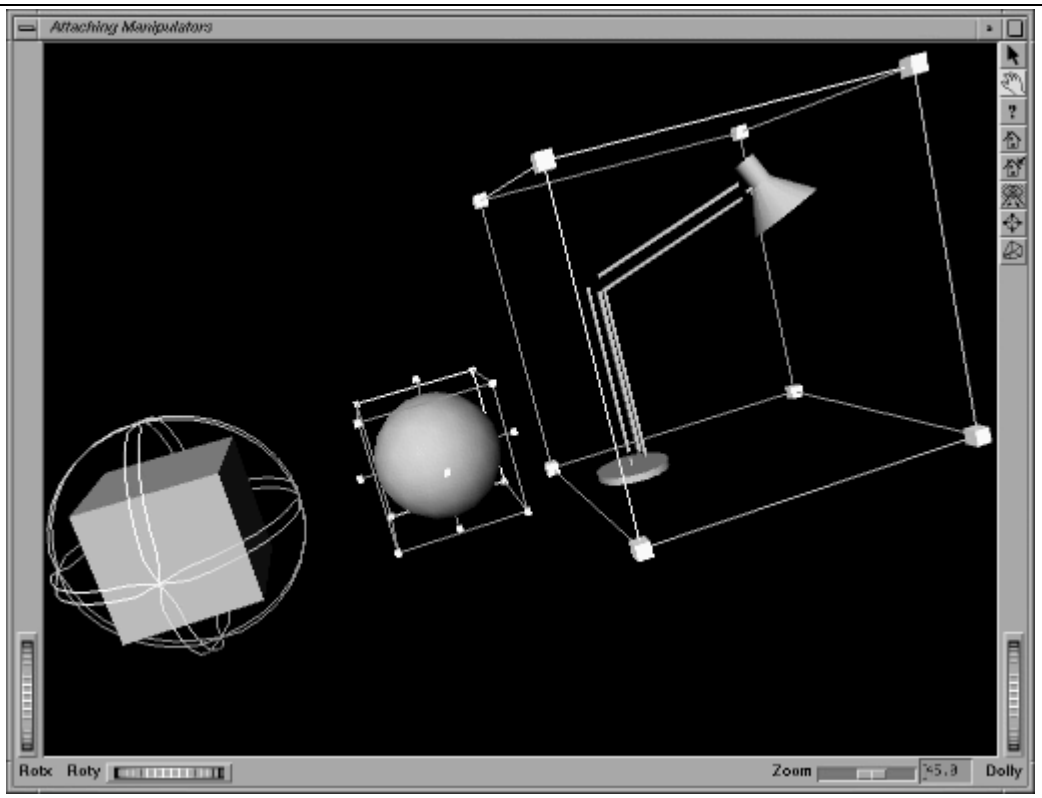
例 15-3 将显示一个立方体，一个球体，以及一盏台灯。台灯是从文件中读入的，并作为**SoWrapperKit**的“contents”部件插入到场景中。当用户点击了立方体时，轨迹球操作器会替换掉那个可以影响到立方体的几何变换节点。当用户点击了球体时，手柄盒操作器会替换掉那个可以影响到球体的几何变换节点。当用户点击了台灯时，变换盒操作器会替换掉台灯的“transform”部件。图 15-9 显示了这个程序创建的场景图。这个例子程序使用了下面的技术：

- 使用**replaceNode()**方法将场景中某个节点变成可编辑修改的节点，当操作结束后，使用**replaceManip()**方法恢复原来的节点。
- 使用了选择器回调函数（见第 10 章）

例 15-3 使用操作器来变换对象

```
// Note that for illustration purposes, the
// cube and SoWrapperKit already have transform nodes
// associated with them; the sphere does not. In all cases,
// the routine createTransformPath() is used to find the
// transform node that affects the picked object.

#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/manips/SoHandleBoxManip.h>
#include <Inventor/manips/SoTrackballManip.h>
#include <Inventor/manips/SoTransformBoxManip.h>
#include <Inventor/nodekits/SoWrapperKit.h>
#include <Inventor/nodes/SoCamera.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoGroup.h>
#include <Inventor/nodes/SoLight.h>
```

图 15-9 向场景中增加操作器




```
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoSelection.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoTransform.h>

#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>

// function prototypes
void selectionCallback(void *, SoPath *);
void deselectionCallback(void *, SoPath *);
void dragStartCallback(void *, SoDragger *);
void dragFinishCallback(void *, SoDragger *);

// global data
SoSeparator *root;
SoHandleBoxManip    *myHandleBox;
SoTrackballManip    *myTrackball;
SoTransformBoxManip *myTransformBox;
SoPath *handleBoxPath    = NULL;
SoPath *trackballPath    = NULL;
SoPath *transformBoxPath = NULL;
```

```
main(int, char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   if (myWindow == NULL) exit(1);

   // Create and set up the selection node
   SoSelection *selectionRoot = new SoSelection;
   selectionRoot->ref();
   selectionRoot->
      addSelectionCallback(selectionCallback, NULL);
   selectionRoot->
      addDeselectionCallback(deselectionCallback, NULL);

   // Create the scene graph
   root = new SoSeparator;
   selectionRoot->addChild(root);

   // Read a file into contents of SoWrapperKit
   // Translate it to the right.
   SoWrapperKit *myWrapperKit = new SoWrapperKit;
   root->addChild(myWrapperKit);
   SoInput myInput;
   if (!myInput.openFile("luxo.iv"))
      return (1);
   SoSeparator *objectFromFile = SoDB::readAll(&myInput);
   if (objectFromFile == NULL) return (1);
   myWrapperKit->setPart("contents",objectFromFile);
   myWrapperKit->set("transform { translation 3 -1 0 }");
   SoMaterial *wrapperMat
      = (SoMaterial *) myWrapperKit->getPart("material",TRUE);
   wrapperMat->diffuseColor.setValue(.8, .8, .8);

   // Create a cube with its own transform.
   SoSeparator *cubeRoot = new SoSeparator;
   SoTransform *cubeXform = new SoTransform;
   cubeXform->translation.setValue(-4, 0, 0);
   root->addChild(cubeRoot);
   cubeRoot->addChild(cubeXform);

   SoMaterial *cubeMat = new SoMaterial;
   cubeMat->diffuseColor.setValue(.8, .8, .8);
   cubeRoot->addChild(cubeMat);
```

```
cubeRoot->addChild(new SoCube);

// Add a sphere node without a transform
// (one will be added when we attach the manipulator)
SoSeparator *sphereRoot = new SoSeparator;
SoMaterial *sphereMat = new SoMaterial;
root->addChild(sphereRoot);
sphereRoot->addChild(sphereMat);
sphereRoot->addChild(new SoSphere);
sphereMat->diffuseColor.setValue(.8, .8, .8);

// Create the manipulators
myHandleBox = new SoHandleBoxManip;
myHandleBox->ref();
myTrackball = new SoTrackballManip;
myTrackball->ref();
myTransformBox = new SoTransformBoxManip;
myTransformBox->ref();

// Get the draggers and add callbacks to them. Note
// that you don't put callbacks on manipulators. You put
// them on the draggers which handle events for them.
SoDragger *myDragger;
myDragger = myTrackball->getDragger();
myDragger->addStartCallback(dragStartCallback,cubeMat);
myDragger->addFinishCallback(dragFinishCallback,cubeMat);

myDragger = myHandleBox->getDragger();
myDragger->addStartCallback(dragStartCallback,sphereMat);
myDragger->addFinishCallback(dragFinishCallback,sphereMat);

myDragger = myTransformBox->getDragger();
myDragger->addStartCallback(dragStartCallback,wrapperMat);
myDragger->addFinishCallback(dragFinishCallback,wrapperMat);

SoXtExaminerViewer *myViewer
   = new SoXtExaminerViewer(myWindow);
myViewer->setSceneGraph(selectionRoot);
myViewer->setTitle("Attaching Manipulators");
myViewer->show();
myViewer->viewAll();

SoXt::show(myWindow);
```

```
   SoXt::mainLoop();
}

// Is this node of a type that is influenced by transforms?
SbBool
isTransformable(SoNode *myNode)
{
   if (myNode->isOfType(SoGroup::getClassTypeId())
      || myNode->isOfType(SoShape::getClassTypeId())
      || myNode->isOfType(SoCamera::getClassTypeId())
      || myNode->isOfType(SoLight::getClassTypeId()))
      return TRUE;
   else
      return FALSE;
}

//  Create a path to the transform node that affects the tail
//  of the input path.  Three possible cases:
//   [1] The path-tail is a node kit. Just ask the node kit for
//       a path to the part called "transform"
//   [2] The path-tail is NOT a group.  Search siblings of path
//       tail from right to left until you find a transform. If
//       none is found, or if another transformable object is
//       found (shape,group,light,or camera), then insert a
//       transform just to the left of the tail. This way, the
//       manipulator only affects the selected object.
//   [3] The path-tail IS a group.  Search its children left to
//       right until a transform is found. If a transformable
//       node is found first, insert a transform just left of
//       that node.  This way the manip will affect all nodes
//       in the group.
SoPath *
createTransformPath(SoPath *inputPath)
{
   int pathLength = inputPath->getLength();
   if (pathLength < 2) // Won't be able to get parent of tail
      return NULL;

   SoNode *tail = inputPath->getTail();

   // CASE 1: The tail is a node kit.
   // Nodekits have built in policy for creating parts.
   // The kit copies inputPath, then extends it past the
```

```
// kit all the way down to the transform. It creates the
// transform if necessary.
if (tail->isOfType(SoBaseKit::getClassTypeId())) {
   SoBaseKit *kit = (SoBaseKit *) tail;
   return kit->createPathToPart("transform",TRUE,inputPath);
}

SoTransform *editXf = NULL;
SoGroup     *parent;
SbBool      existedBefore = FALSE;

// CASE 2: The tail is not a group.
SbBool isTailGroup;
isTailGroup = tail->isOfType(SoGroup::getClassTypeId());
if (!isTailGroup) {
   // 'parent' is node above tail. Search under parent right
   // to left for a transform. If we find a 'movable' node
   // insert a transform just left of tail.
   parent = (SoGroup *) inputPath->getNode(pathLength - 2);
   int tailIndx = parent->findChild(tail);

   for (int i = tailIndx; (i >= 0) && (editXf == NULL);i--){
      SoNode *myNode = parent->getChild(i);
      if (myNode->isOfType(SoTransform::getClassTypeId()))
         editXf = (SoTransform *) myNode;
      else if (i != tailIndx && (isTransformable(myNode)))
         break;
   }
   if (editXf == NULL) {
      existedBefore = FALSE;
      editXf = new SoTransform;
      parent->insertChild(editXf, tailIndx);
   }
   else
      existedBefore = TRUE;
}
// CASE 3: The tail is a group.
else {
   // Search the children from left to right for transform
   // nodes. Stop the search if we come to a movable node
   // and insert a transform before it.
   parent = (SoGroup *) tail;
   for (int i = 0;
```

```
         (i < parent->getNumChildren()) && (editXf == NULL);
         i++) {
         SoNode *myNode = parent->getChild(i);
         if (myNode->isOfType(SoTransform::getClassTypeId()))
            editXf = (SoTransform *) myNode;
         else if (isTransformable(myNode))
            break;
      }
      if (editXf == NULL) {
         existedBefore = FALSE;
         editXf = new SoTransform;
         parent->insertChild(editXf, i);
      }
      else
         existedBefore = TRUE;
   }

   // Create 'pathToXform.' Copy inputPath, then make last
   // node be editXf.
   SoPath *pathToXform = NULL;
   pathToXform = inputPath->copy();
   pathToXform->ref();
   if (!isTailGroup) // pop off the last entry.
      pathToXform->pop();
   // add editXf to the end
   int xfIndex   = parent->findChild(editXf);
   pathToXform->append(xfIndex);
   pathToXform->unrefNoDelete();

   return(pathToXform);
}

// This routine is called when an object
// gets selected. We determine which object
// was selected, then call replaceNode()
// to replace the object's transform with
// a manipulator.
void
selectionCallback(
   void *, // user data is not used
   SoPath *selectionPath)
{
   // Attach the manipulator.
```

```
   // Use the convenience routine to get a path to
   // the transform that affects the selected object.
   SoPath *xformPath = createTransformPath(selectionPath);
   if (xformPath == NULL) return;
   xformPath->ref();

   // Attach the handle box to the sphere,
   // the trackball to the cube
   // or the transformBox to the wrapperKit
   if (selectionPath->getTail()->isOfType(
        SoSphere::getClassTypeId())) {
      handleBoxPath = xformPath;
      myHandleBox->replaceNode(xformPath);
   }
   else if (selectionPath->getTail()->
        isOfType(SoCube::getClassTypeId())) {
      trackballPath = xformPath;
      myTrackball->replaceNode(xformPath);
   }
   else if (selectionPath->getTail()->
        isOfType(SoWrapperKit::getClassTypeId())) {
      transformBoxPath = xformPath;
      myTransformBox->replaceNode(xformPath);
   }
}

// This routine is called whenever an object gets
// deselected. It detaches the manipulator from
// the transform node, and removes it from the
// scene graph that will not be visible.
void
deselectionCallback(
   void *, // user data is not used
   SoPath *deselectionPath)
{
   if (deselectionPath->getTail()->
        isOfType(SoSphere::getClassTypeId())) {
      myHandleBox->replaceManip(handleBoxPath,NULL);
      handleBoxPath->unref();
   }
   else if (deselectionPath->getTail()->
        isOfType(SoCube::getClassTypeId())) {
      myTrackball->replaceManip(trackballPath,NULL);
```

```
      trackballPath->unref();
   }
   else if (deselectionPath->getTail()->
        isOfType(SoWrapperKit::getClassTypeId())) {
      myTransformBox->replaceManip(transformBoxPath,NULL);
      transformBoxPath->unref();
   }
}

// This is called when a manipulator is
// about to begin manipulation.
void
dragStartCallback(
   void *myMaterial, // user data
   SoDragger *)         // callback data not used
{
   ((SoMaterial *) myMaterial)->diffuseColor=SbColor(1,.2,.2);
}

// This is called when a manipulator is
// done manipulating.
void
dragFinishCallback(
   void *myMaterial, // user data
   SoDragger *)    // callback data not used
{
   ((SoMaterial *) myMaterial)->diffuseColor=SbColor(.8,.8,.8);
}
```

定制拖拽器（高级内容）

本节将讨论如何修改拖拽器的外观。可以通过修改拖拽器部件的缺省模型数据或通过修改一个已被建立的拖拽器实例的部件来完成定制化的工作。虽然可以通过这两种方式来修改拖拽器的外观，但却不能以此来增加新的功能。

本节将使用**SoTrackballDragger**作为一个例子，讨论拖拽器的组成部件以及它们是如何结合在一起而成为一个完整的拖拽器。本节还将解释如何修改和删除每个部件的模型，以及这些模型是如何影响到拖拽器功能的。例 15-4 将演示当创建完一个**SoTranslate1Dragger**实例后如何修改其部件的外观。

拖拽器的部件

每种拖拽器都是一种由部件构成的节点工具包。“部件”只不过是拖拽器模型的一小块部分，一般部件都与某些任务相关联。通常两个部件为一组，其中一个部件在被使用（或激活）时显示，另外一个在不被使用（或非激活）时显示。例如，轨迹球的“XRotator”非激活部件是一个白色的条带，而其“XRotatorActive”激活部件则是一个黄色的条带。

每种拖拽器都有一个相关联的资源文件。这些资源文件包含在 Inventor 的图形场景数据库中，主要用来定义拖拽器的每个部件的缺省模型。通过创建新的资源文件，我们可以覆盖掉部件缺省的模型，为部件定义新的模型和新的颜色或绘制风格等属性。在资源文件中，每个部件的场景数据都应该使用它们唯一的资源名称来标注。

许多拖拽器类都使用相同的部件名称。例如，轨迹球、转盘（rotate-disk）、旋转柱（rotate-cylindrical）、旋转球（rotate-spherical）等这些拖拽器都带有一个叫做“rotator”的部件。因为缺省的部件是存放在全局字典（global dictionary）中，所以每个拖拽器类的部件必须有一个唯一的资源名称。在所有这类的情况下，资源名等于类名（不带“So”和“Dragger”单词）加上部件名。表 15-2 展示了资源名称与部件名称的关系。

例如，**SoTrackballDragger**一共有 12 个部件。表 15-2 列出了其中的 8 个部件的资源名称和部件名称（为简单起见，忽略了“userRotator”和“userAxis”部件）。当我们使用**SoTrackballDragger**拖拽器时，实际上我们是在操作拖拽器的部件。例如，如果鼠标点击并拖动轨迹球的“XRotator”部件时，那么轨迹球拖拽器将会开始以x轴为旋转轴做旋转运动。

表 15-2 **SoTrackballDragger**部分挑选的部件

资源名称	部件名称	作用
trackballRotator trackball RotatorActive	rotator rotatorActive	自由旋转
trackballXRotator trackballXRotatorActive	XRotator XRotatorActive	以 x 轴为旋转轴做旋转
trackballYRotator trackballYRotatorActive	YRotator YRotatorActive	以 y 轴为旋转轴做旋转
trackballZRotator trackballZRotatorActive	ZRotator ZRotatorActive	以 z 轴为旋转轴做旋转

在创建完拖拽器后修改部件

如果想修改一个已被创建的拖拽器的某个部件，可以使用**SoInteractionKit**基类提供的**setPart()**或**setPartAsPath()**方法。**setPart()**方法使用部件场景根节点作为一个输入参数，而**setPartAsPath()**则使用路径作为输入参数。

例如，修改**myDragger**的“rotator”部件：

```
myDragger->setPart("rotator", myNewRotatorSceneGraph);
```

修改一个位于操作器中的拖拽器“rotator”部件：

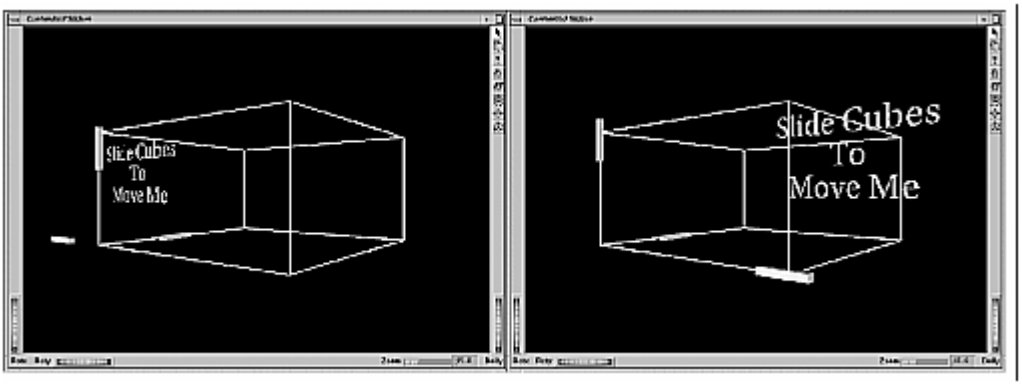
```
myManip->getDragger()->setPart("rotator",myNewRotatorSceneGraph);
```

我们也可以通过向**setPartAsPath()**方法提供一个指向场景中某个特定对象实例的路径来完成上面同样的工作。之后，拖拽器将使用那个特定对象来表示相应的部件（The dragger then uses that object for the part）。例如，如果我们现在有一个箭头模型，用来表示一个安装在柱子上的风向标。我们可以通过向**setPartAsPath()**方法提供一条指向这个箭头的路径，来修改拖拽器的某个部件。（请注意指定一个节点与指定一条路径之间的不同之处。如果我们使用**setPart()**方法来指定一个箭头节点，那么将会创建一个新的箭头节点，屏幕上将会出来两个完全相同的箭头节点。如果我们使用**setPartAsPath()**方法来指定一个指向箭头的路径，拖拽器将只使用已有的箭头，等待用户使用鼠标点击位于谷仓上的相同风向标模型（Note the difference here between specifying a node and specifying a path. If you specify the arrow node using setPart(), a new instance of that node is created and two copies of the same geometry appear on the screen. If you specify the path to the arrow using setPartAsPath(), the dragger actually uses the existing arrow node and waits for the user to press the mouse on the same weather vane that is sitting on the barn））

```
myRotateManip->getDragger()->setPartAsPath("rotator",
                                pathToMyWeatherVaneArrow);
```

例 15-4 将演示如何修改在例 15-2 中创建的拖拽器的模型。“translator”和“translatorActive”部件现在已经变成了立方体模型，而不再是原来的箭头模型了。这里使用 **setPart()**方法将新的图形场景替换掉缺省的部件场景。图 15-10 显示了新的拖拽器模型。

图 15-10 将拖拽器的某些部件修改成立方体模型



例 15-4 在创建完拖拽器后修改部件

```
// Create myTranslator and myTranslatorActive.
// These are custom geometry for the draggers.
SoSeparator *myTranslator = new SoSeparator;
SoSeparator *myTranslatorActive = new SoSeparator;
myTranslator->ref();
myTranslatorActive->ref();

// Materials for the dragger in regular and active states
SoMaterial *myMtl = new SoMaterial;
SoMaterial *myActiveMtl = new SoMaterial;
myMtl->diffuseColor.setValue(1,1,1);
myActiveMtl->diffuseColor.setValue(1,1,0);
myTranslator->addChild(myMtl);
myTranslatorActive->addChild(myActiveMtl);

// Same shape for both versions.
SoCube *myCube = new SoCube;
myCube->set("width 3 height .4 depth .4");
myTranslator->addChild(myCube);
myTranslatorActive->addChild(myCube);

// Now, customize the draggers with the pieces we created.
xDragger->setPart("translator",myTranslator);
xDragger->setPart("translatorActive",myTranslatorActive);
yDragger->setPart("translator",myTranslator);
yDragger->setPart("translatorActive",myTranslatorActive);
zDragger->setPart("translator",myTranslator);
zDragger->setPart("translatorActive",myTranslatorActive);
```

修改部件的缺省模型

每种拖拽器类都一个与其相关联的资源文件，这些资源文件包含有定义拖拽器缺省模型的 Inventor 场景图形数据。这些缺省模型的场景数据同时被编译到 Inventor 发行库中，这样，即使这些资源文件丢失了，拖拽器仍能正常操作。（**读者可以在 Open Inventor 开发包安装目录下找到一个叫做“data\draggerDefaults”的目录，这个目录里面的文件就是用来定义拖拽器缺省模型的 Inventor 场景数据文件。这些文件也同时存在于 Inventor 二进制发行库（例如 DLL 库）中，这样用户的 Inventor 程序即使在其它计算机中不能找到“data\draggerDefaults”目录中的文件，也能保证拖拽器的正常使用。译者注**）

拖拽器到那里寻找缺省资源文件

当创建一个拖拽器时，拖拽器会检查系统中是否创建了一个覆盖缺省资源的新资源文件。当读入这个新文件时（如果发现了），如果拖拽器发现了一个特定模型的第二个定义，

那么新模型将替换掉以前定义的具有相同名称的旧模型。

只有系统中定义了 SO_DRAGGER_DIR 环境变量，Inventor 才会去查找缺省资源文件。如果定义了 SO_DRAGGER_DIR 环境变量，Inventor 将在环境变量所指定的路径中查找文件。

对于所有的拖拽器类来说，一个特定的拖拽器类只读取和它同名的资源文件：例如，**SoTranslate1Dragger**类只读取名为translate1Dragger.iv的资源文件。**SoTrackballDragger**类只读取名为trackballDragger.iv的文件。

修改拖拽器部件的缺省模型是一件比较简单的事情，只需要创建一个用来定义新资源的场景数据文件即可。当拖拽器读入新数据文件时，如果遇到了和缺省模型名称相同的新模型，那么新定义的模型将覆盖掉以前定义的模型。接下来的两个章节以一个**SoTrackballDragger**为基础来演示如何修改拖拽器部件的缺省模型。

如何编辑资源文件

要修改部件的缺省模型，我们只需要重新定义部件所使用的模型图形场景。当产生一个新资源文件时，只需要定义我们希望修改的模型。其它的模型可以继续使用原来缺省的模型。

需要记住的是，我们决不应该编辑“Inventor/resources”目录中的文件。如果我们希望应用程序使用新替换的资源文件，应该将这些新资源文件放置在一个合适的目录中，然后设置 SO_DRAGGER_DIR 环境变量指向这个目录。

作为一个例子，假设我们使用一个立方体模型来替换掉轨迹球的**trackballRotator**资源数据（更多关于Inventor文件格式的知识，见第 11 章）。查看轨迹球的缺省模型文件，我们可以看到**trackballRotator**资源定义成下面的图形场景：

```
# default geometry for SoTrackballDragger's "rotator" part (inactive)
DEF trackballRotator Separator {
   DrawStyle { style INVISIBLE }
   Sphere {}
}
# default geometry for SoTrackballDragger's "rotatorActive" part
DEF trackballRotatorActive Separator {
   DrawStyle { style INVISIBLE }
   Sphere {}
}
```

注意，在轨迹球的资源文件中，用来指定**rotator**和**rotatorActive**部件的资源都是一个不可见球体的模型。虽然这种情况很常见，但有些拖拽器对于激活和非激活状态有着完全不同的显示模型（绝大多数操作器都有一个比球体更复杂图形模型）。

要将**trackballRotator**和**trackballRotatorActive**资源从一个不可见球体变成一个可见的

立方体，我们只需要简单将场景中所有的球体替换成立方体即可。

```
# default geometry for the SoTrackballDragger's "rotator" part
DEF trackballRotator Separator {
   BaseColor {
      rgb 1. 1. 1.        #white
   }
   Cube {}
}
# default geometry for the SoTrackballDragger's "rotatorActive" part
DEF trackballRotatorActive Separator {
   BaseColor {
      rgb .5 .5  0.        #yellow
   }
   Cube {}
}
```

使用这种方法，我们不仅可以修改一个给定部件的模型，还可以完全屏蔽掉那个部件的功能。例如，禁止轨迹球“rotator”部件的功能，但同时仍然显示立方体模型，我们只需要使立方体处于不可拾取的状态。

```
# default geometry for the SoTrackballDragger's "rotator" part
DEF trackballRotator Separator {
   BaseColor {
      rgb 1. 1. 1.        #white
   }
   PickStyle {
      style UNPICKABLE
   }
   Cube {}
}
# default geometry for the SoTrackballDragger's "rotatorActive" part
DEF trackballRotatorActive Separator {
   BaseColor {
      rgb .5 .5  0.        #yellow
   }
   PickStyle {
      style UNPICKABLE
   }
   Cube {}
}
```

要将轨迹球的“rotator”部件完全去掉，只留下绕 x、y、z 轴旋转的功能，应该将“rotator”

部件重新定义为一个空的图形场景。

```
# default geometry for SoTrackballDragger's "rotator" part
DEF trackballRotator Separator {
}
# default geometry for SoTrackballDragger's "rotatorActive" part
DEF trackballRotatorActive Separator {
}
```

我们也可以从一个文件中读入模型数据，而不只是必须在文件中内联定义。

```
DEF trackballRotator Separator {
   File { name "myCustomRotator.iv" }
}
DEF trackballRotatorActive Separator {
   File { name "myCustomRotatorActive.iv" }
}
```

注意：绝对不要将一个拖拽器的部件设置成 NULL。拖拽器内部函数需要一个真实存在的节点，即使它是如前面例子中演示的一个空场景。（通常，在拖拽器内部函数中，拖拽器的部件是一个 **SoSwitch** 节点的子节点，将部件节点设置成 NULL，将导致 **SoSwitch** 节点的子节点处于一个不正确的排列位置上）

第十六章 Inventor 组件库

本章目标

在阅读完本章节后，我们将有能力做下列事情：

- 构造、创建和使用**SoXtRenderArea**
- 使用用于初始化和窗口管理的 Inventor 工具函数
- 以叠加平面（overlay planes）的方式渲染一个简单的图形场景。
- 构造和创建 Inventor 组件，并将这些组件作为 Xt widgets 来管理
- 将组件直接附着在图形场景，并把组件的数据传递到应用程序中。
- 利用回调函数将数据从组件传递到应用程序中。
- 为 Inventor 标准的观察器窗口增加上用户自定义的按钮控件
- 利用 Inventor 的剪贴板拷贝和粘贴场景数据

本章将讨论 Inventor 组件库，Inventor 组件库中包括工具函数、渲染区、和一组 Xt 组件。组件是一种可重用的模块，这种模块一般带有以交互方式修改图形场景的内置用户界面。为了将组件更容易整合进我们的应用程序中，每个组件都是建立在 Motif 风格的 Xt widgets 之上的，它们可以单独使用，也可以和其它 Xt widgets 结合使用。本章将介绍的最重要的概念包括：两种类型的组件－*编辑器*（*editors*）和*观察器*（*viewers*）、*构造* 和*创建* 组件的步骤、组件作为 Xt widgets 来管理的步骤。因为所有的组件都是交互式的，都可用于编辑 3D 场景中的物体，所以本章还将描述不同类型的组件如何将数据传递回应用程序中。

（**本章主要讨论UNIX的Xt窗口系统下的Inventor组件库，但因为Inventor具有平台无关性，所以本章讨论的所有内容对Microsoft Windows操作系统也是可以使用的，读者可以简单地将Xt变成Win就可以转换到Microsoft Windows操作系统下，例如：将SoXtRenderArea变成SoWinRenderArea。译者注**）

组件简介

Inventor 组件库主要由三个部分组成：

- 用于初始化和窗口管理的 Xt 工具函数。
- 用于图形场景静态显示(static display)的 Xt 渲染区(**所谓静态显示，就是指没有交互作用，用户不能使用鼠标来改变场景的观察位置。译者注**)。
- 一组 Xt 组件，这些组件带有自己的渲染区，以及一个用以修改场景显示的用户界面。

下面的章节将会更详细地讨论这些内容。本章假设读者已经阅读过本书第 10 章的内容，在第 10 章中讨论了 Xt 库和 Open Inventor 工具包（独立于具体的窗口系统）的关系。

Xt 工具函数

本节将简要描述使用 Xt Intrinsics(Intrinsics 是一个建立在 X Windows 系统库之上

的界面库)来初始化 Inventor 的基本步骤。一个 Xt widget 包含有一个 X 窗口，以及一些用来控制 widget 行为的额外功能。因为 widget 包含有窗口，所以它可以从 X Server 端接收事件。

SoXt::init()函数返回一个可以充当应用程序主外壳窗口（main shell window）的Xt widget。在下面的例子中，widget被命名为**myWindow**。之后将向这个窗口插入一个**SoXtRenderArea**。

下面是基本的步骤：

1. 使用Xt Intrinsics（**SoXt::init()**）来初始化Inventor。
2. 创建**SoXtRenderArea**渲染区。
3. 构造其它 Inventor 对象和 Xt widgets
4. 显示渲染区和Xt widgets(**myRenderArea->show(); SoXt::show()**)
5. 开始事件循环（**SoXt::mainLoop()**）

下面是基本步骤的例子代码：

```
#include <X11/Intrinsic.h>
#include <Inventor/So.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>

main(int argc, char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);

   SoXtRenderArea *myRenderArea =
         new SoXtRenderArea(myWindow);

   SoSeparator *root = new SoSeparator;
   // Build other Inventor objects and Xt widgets
   // and set up the root
   // ...

   myRenderArea->setSceneGraph(root);
   myRenderArea->setTitle("Simple Xt");
   myRenderArea->show(); // this calls XtManageChild
   SoXt::show(myWindow); // this calls XtRealizeWidget

   // Realize other Xt widgets
   // ...
```

```
   // Go into main event loop
   SoXt::mainLoop();
}
```

提示：应用程序要保证在调用shell widget的**show()**之前调用子widget的**show()**。如果我们试图先显示shell widget，那么我们的程序会收到一个“Shell widget x has zero width and/or height”的错误。

渲染区

SoXtRenderArea是一个用来执行OpenGL渲染任务的Xt widget。当它接收到一个X 事件时，它会将X事件转换成**SoEvents**，然后再把**SoEvents**传入到场景管理器中以供处理。

方法

可以使用**setSceneGraph()**方法将被渲染的图形场景设置进渲染区中（**setSceneGraph()**将会增加场景根节点的参考计数值）。**getSceneGraph()**方法可返回图形场景的根节点。

SoXtRenderArea中其它比较有用的方法包括：

setTransparencyType() 定义如何渲染透明物体（详见第 9 章的渲染 action 章节）

setAntialiasing() 定义反走样方法

setBorder() 显示或隐藏窗口的边框

setBackgroundColor() 定义窗口的背景色

渲染区附带有一个作用于场景根节点的节点传感器，只要场景发生了变化，这个传感器就会使场景自动重绘。下面的方法用于修改重绘传感器的优先级：

setRedrawPriority() 定义重绘传感器的优先级（缺省优先级为 10000）

如果我们希望禁止自动重绘功能，可以使用下面两个方法：

setAutoRedraw() 允许或禁止渲染区的重绘传感器

render() 立即重绘场景。如果 AutoRedraw 是 TRUE，则不需要调用此方法

这些方法的更多资料请查阅*Open Inventor C++ Reference Manual*中**SoXtRenderArea**。

Xt 设备

当我们创建一个**SoXtRenderArea**渲染区时，如果使用的是缺省参数，则渲染区将自动处理鼠标和键盘事件。**SoXtRenderArea**的构造函数是：

```
SoXtRenderArea( Widget parent = NULL,
                const char * name = NULL,
                SbBool buildInsideParent = TRUE,
                SbBool getMouseInput = TRUE,
                SbBool getKeyboardInput = TRUE);
```

如果要禁止从鼠标或键盘来的输入事件，可以将 *getMouseInput* 或 *getKeyboardInput* 参数设置为 FALSE 即可。例如，禁止鼠标输入：

```
SoXtRenderArea *renderArea = new SoXtRenderArea(parent,
        "myRenderArea", TRUE, FALSE, TRUE);
```

Inventor 定义了三种 Xt 设备：

- **SoXtKeyboard**
- **SoXtMouse**
- **SoXtSpaceball**

使用**registerDevice()**方法可以向渲染区注册其它另外的设备(像轨迹球设备)。当调用此方法后，渲染区将会注册自己感兴趣的事件（新设备产生的)。当渲染区接收到这些事件后，它会将这些事件转化成**SoEvents**，然后将这些**SoEvents**传递进场景管理器中以供处理。关于如何创建我们自己的设备，请查阅*The Inventor Toolmaker*。

使用叠加平面（overlay planes）（高级内容）

叠加平面是一组分离的用于 Inventor 特殊目的的位平面（bitplanes)。(请读者检查自己的 OpenInventor 开发库的发行声明，里面有开发库支持叠加平面个数的信息。叠加平面个数和具体的 OpenInventor 实现库有关)。叠加平面通常用于显示位于主画面之上的，且独立重绘的场景对象（The overlay planes are typically used for objects in the scene that appear on top of the main image and are redrawn independently)。尽管在叠加平面内的场景颜色和复杂度有一些限制，但使用叠加平面可以快速重绘一个简单的场景，而不必重新绘制“整个”场景。叠加平面为用户界面反馈提供了一种有用的机制。例如，可以快速绘制跟随鼠标移动的物体模型。

使用下面的方法可以将场景设置进叠加平面中：

setOverlaySceneGraph()	将需要渲染的场景设置进叠加平面中
setOverlayColorMap()	设置叠加位平面所使用的颜色表。叠加平面通常使用颜色－索引模式（color-index mode）
setOverlayBackgroundIndex()	设置叠加图象背景色的索引值（缺省为 0，清除色）

叠加场景有自己的重绘传感器，这点和“正规”的场景有些类似，不过叠加场景也有一些限制：

- 如果我们的系统只支持很少数量的叠加平面（例如，只支持两个)，只能对

SoLightModel的**model**域设置成BASE_COLOR（这是一种效果比较差的光照模型。译者注）。（如果我们的系统支持超过两个以上的叠加平面，那么我们就可以使用**SoMaterialIndex**节点（或使用**SoColorIndex**节点指定颜色索引值）获得更自然的灯光效果）

- 叠加场景要尽可能的简单。尽量使用线性绘制风格、矩形物体、和 2D 文本等这些可以快速绘制的节点。不能使用纹理。因为叠加平面是单缓冲的，所以如果场景太复杂的话，重绘场景很容易会发生闪烁现象。
- 确保要装载颜色表。Inventor 没有对叠加平面设置缺省的颜色表。

叠加平面的颜色表中包含有少量的颜色。Color 0 是背景清除色，不能被修改。如果有两个位平面，我们可以使用索引值 1~3 来指定颜色。下面是的**setOverlayColorMap()**语法：

setOverlayColorMap(int *startIndex*, int *num*, const SbColor **colors*);

如果要以一种特定的颜色来渲染某个物体时，需要使用**SoColorIndex**节点设置当前颜色索引值。当程序处于颜色－索引模式（color-index mode）下时，不要使用**SoMaterial**和**SoBaseColor**节点设置颜色(即使使用了，Inventor也会忽略掉这两种节点)

例 16-1 将演示叠加平面的用法。缺省情况下，Color 0 用于叠加平面的背景色（即清除色），所以本例将使用 Color 1 来显示物体。（请注意：Coin 2.44 还不支持叠加平面，所以例 16-1 在 Coin 中不能正确显示。TGS 支持叠加平面。译者注）

例 16-1 使用叠加平面

```
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/nodes/SoNode.h>
#include <Inventor/nodes/SoCone.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>

static char *overlayScene = "\
#Inventor V2.0 ascii\n\
\
Separator { \
   OrthographicCamera { \
      position 0 0 5 \
      nearDistance 1.0 \
      farDistance 10.0 \
      height 10 \
   } \
   LightModel { model BASE_COLOR } \
   ColorIndex { index 1 } \
   Coordinate3 { point [ -1 -1 0, -1 1 0, 1 1 0, 1 -1 0] } \
```

```
   FaceSet {} \
} ";

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);

   // Read the scene graph in
   SoInput in;
   SoNode *scene;
   in.setBuffer((void *)overlayScene, (size_t)
         strlen(overlayScene));
   if (! SoDB::read(&in, scene) || scene == NULL) {
      printf("Couldn't read scene\n");
      exit(1);
   }

   // Allocate the viewer, set the overlay scene and
   // load the overlay color map with the wanted color.
   SbColor color(.5, 1, .5);
   SoXtExaminerViewer *myViewer = new
         SoXtExaminerViewer(myWindow);
   myViewer->setSceneGraph(new SoCone);
   myViewer->setOverlaySceneGraph(scene);
   myViewer->setOverlayColorMap(1, 1, &color);
   myViewer->setTitle("Overlay Plane");

   // Show the viewer and loop forever
   myViewer->show();
   XtRealizeWidget(myWindow);
   SoXt::mainLoop();
}
```

Xt 组件

组件是一些提供了具有 3D编辑功能的widgets。位于Inventor组件库中的所有组件都会为标准Motif风格的布局器和控件返回一个Xt widget句柄。渲染区也是一个简单组件的实例。观察器组件是从**SoXtRenderArea**派生出来的。

每种组件都包含有一些像按钮、菜单、滚动轮等这些允许用户以交互方式修改场景的用户界面。材质编辑器就是其中之一，我们将在例 16-2、16-3、以及 16-4 中使用到它。通过

使用材质编辑器，用户可以定制位于 Inventor 窗口中物体的材质。用户可以修改材质的环境光、漫反射、高光、透明度、和表面发光度等域值，并且可以立即在场景中看到这些修改产生的效果。另外一个例子是 examiner 观察器，这种观察器允许用户在场景中自由移动照相机，提供实时观察场景变换的能力。图 16-1 给出了组件的类树图。

SoXtComponent是一种Inventor C++包装类，用于包装Motif-compliant风格的widget。这意味着我们可以使用像公告牌（bulletin boards）、form widgets、以及row/column widgets这些标准的布局方案（layout schemes）来布置窗口中的组件。材质编辑器本身也是一个**SoXtComponent**，它是由其它组件和Motif风格的widget组成的（它的颜色滚动轮是从**SoXtComponent**派生出来的，单选按钮、转换按钮（toggle buttons）、和菜单都是Motif风格的widget）。我们可以将一个widget的名字传递进每个组件中，之后名字就可用被用在资源文件中，作为widget的Motif名字。（You can pass in a widget name to each component, which can then be used in resource files as the Motif name of the widget）

组件分成观察器和编辑器这两大类，这主要是根据组件会影响到场景的那部分来划分的。“观察器”会影响到场景的照相机节点，“编辑器”会影响到场景中像**SoMaterial**和**SoDirectionalLight**这样的节点。

图 16-1 组件类

通用模式

下面介绍的是在程序中使用组件的一般步骤(对于某些特定组件额外需要注意的事项也将在下面章节中做介绍)。

1. 利用组件的构造函数创建组件。将父 widget、widget 名称、以及组件是否在父 widget 中创建等这些作为输入参数传到构造函数中。
2. 显示或隐藏组件。
3. 将数据从组件传到应用程序中。

构造组件

利用组件的构造函数创建组件。构造函数的形式为：

```
SoXtComponent(Widget parent = NULL,
              const char * name = NULL,
              SbBool buildInsideParent = TRUE,
              SbBool getMouseInput = TRUE,
              SbBool getKeyboardInput = TRUE);
```

例如：

```
SoXtMaterialEditor *editor = new SoXtMaterialEditor(parentWidget);
```

这步将会初始化组件的局部变量和数据结构，并创建组件。我们可以提供父 widget 作为参数，用于指定新创建的组件位于那个 widget 内。如果我们没有提供父 widget，或设置参数 *buildInsideParent* 为 FALSE，那么组件将被创建在自己的 Shell 窗口中。将组件创建在自己的窗口中会有一个“副作用”，即组件可以在必要时改变窗口的大小。如果组件创建在父 widget 中，则不能改变大小。如果没有提供 name 参数，在上面的例子中，名字将是类名称“SoXtMaterialEditor”。

如果我们设置参数 *buildInsideParent* 为 FALSE，组件将被创建在自己的 Shell 窗口，但组件会将传入的父 widget 作为 X 资源查找的 widget 层次的一部分。(If you specify FALSE for the buildInsideParent parameter, the component is built inside its own shell, but it uses the passed parent as part of the widget hierarchy for X resource lookup.)

显示和隐藏组件

show()和**hide()**是用来管理组件widget的方法。总之，**show()**方法用来显示组件，**hide()**方法则用来隐藏组件。然而在Motif-compliant风格的应用程序中，在组件的子窗口显示前，widget树的最顶层父widget必须要首先显示（*realized*）。(However, in Motif-compliant applications, the topmost parent of the widget tree must be *realized* before its children are displayed)。另外，只有被管理的子窗口才会被显示。

如果Inventor组件是一个顶层的Shell Widget（即没有父widget参数传到构造函数中），调用**show()**方法将会导致组件调用它自己的**XtRealizeWidget()**函数和子窗口的**XtManageChild()**函数。

如果组件不是一个顶层的Shell Widget，调用**show()**方法将会导致组件调用它自己和其所有子窗口的**XtManageChild()**函数。不过，在调用顶层widget的**XtRealizeWidget()**函数之前，这些widegt都是不可见的。

SoXtComponent的**show()**和**hide()**方法还会对组件做一些必要的辅助工作。当我们使用组件时，一定要保证调用组件的**show()**方法，不能调用**XtManage()**或**XtRealize()**方法。同时要确保调用**hide()**方法，不能调用**XtUnmanage()**或**XtUnrealize()**方法。例如：

```
SoXtRenderArea *ra = new SoXtRenderArea();
ra->show();
```

组件还有一些用于在程序运行期间改变其外观的特定方法，（具体请查阅*Open Inventor C++ Reference Manual*书中的**SoXtComponent**）。这些方法包括：

`setTitle()` 在顶层 shell widget 的标题栏中设置标题

`setSize()` 定义组件的大小(uses `XtSetValue()`)

`getSize()` 返回组件的大小(uses `XtGetValue()`)

`isVisible()` 如果组件当前是已被映射,并显示在显示器中，则返回 TRUE

将数据传到应用程序中

组件有两种方法可以将数据回传给应用程序：

- 当组件中发生了某些特定变化的事件时,可以利用回调函数列表来通知应用程序（见例 16-2）。在我们希望可以处理多个节点的时候，这种方法是非常有用的（通常，我们每次只能将组件关联到一个节点上）。
- 将组件关联到场景中一个节点（或域）上（见例 16-3）。对于“观察器”组件来说，这种方法是唯一的一种将数据回传给应用程序的方法。“观察器”组件是被关联到一个场景节点之上的。

使用回调函数

像“材质编辑器”这样的编辑器组件可以使用回调函数将数据回传给应用程序。例 16-2 将演示材质编辑器的回调函数的用法。

当构建组件的时候，在组件中会自动创建一个类型为**SoCallbackList**的回调函数以及相关联数据的列表。我们可以向列表中增加和删除回调函数，可以传递一个回调数据的指针。

某些像“观察器”这样的 widget，使用下列回调函数列表：

- *Start callbacks*－当交互操作开始时被调用（例如，在鼠标按键按下事件中）
- *Finish callbacks*－当交互操作结束时被调用（例如，在鼠标按键抬起事件中）

下面的方法可以向列表中增加和删除回调函数：

addStartCallback(*functionName, userData*)
removeStartCallback(*functionName, userData*)

addFinishCallback(*functionName, userData*)
removeFinishCallback(*functionName, userData*)

材质编辑器根据自身的一个可编程控制的更新频率,来调用它的回调函数或更新与其相关联的节点。使用**setUpdateFrequency()**方法可以设置更新频率。更新的方式有两种：

CONTINUOUS　当数据发生变化时，持续更新域值（缺省方式）

AFTER_ACCEPT 只有在用户点击了“接受”按钮后才更新域值

例 16-2 在应用程序的一个窗口中创建了一个渲染区组件，同时还创建了一个位于自身窗口内的材质编辑器组件。程序使用组件的回调函数来报告新的域值。

例 16-2 使用回调函数

```
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtMaterialEditor.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

// This is called by the Material Editor when a value changes
void
myMaterialEditorCB(void *userData, const SoMaterial *newMtl)
{
   SoMaterial *myMtl = (SoMaterial *) userData;

   myMtl->copyFieldValues(newMtl);
}

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);

   // Build the render area in the applications main window
   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
   myRenderArea->setSize(SbVec2s(200, 200));
   // Build the Material Editor in its own window
   SoXtMaterialEditor *myEditor = new SoXtMaterialEditor;

   // Create a scene graph
   SoSeparator *root = new SoSeparator;
   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   SoMaterial *myMaterial = new SoMaterial;
   root->ref();
   myCamera->position.setValue(0.212482, -0.881014, 2.5);
```

```
   myCamera->heightAngle = M_PI/4;
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   root->addChild(myMaterial);

   // Read the geometry from a file and add to the scene
   SoInput myInput;
   if (!myInput.openFile("dogDish.iv"))
      exit (1);
   SoSeparator *geomObject = SoDB::readAll(&myInput);
   if (geomObject == NULL)
      exit (1);
   root->addChild(geomObject);

   // Add a callback for when the material changes
   myEditor->addMaterialChangedCallback(
         myMaterialEditorCB, myMaterial);

   // Set the scene graph
   myRenderArea->setSceneGraph(root);

   // Show the main window and the Material Editor
   myRenderArea->setTitle("Editor Callback");
   myRenderArea->show();
   SoXt::show(myWindow);
   myEditor->show();

   // Loop forever
   SoXt::mainLoop();
}
```

将组件关联到场景节点上

一种可以直接修改场景的方法是将编辑器组件关联到场景中某个节点上。例 16-3 将演示使用**attach()**方法将材质编辑器关联到一个材质节点上：

```
myEditor->attach(myMaterial);
```

attach()的语法是：

attach(SoMaterial **material*, int *index* = 0);

material 被编辑的节点

Index 对于多值材质节点来说，Index 值用来定义在被编辑材质节点内的索引值

同样的，“观察器”也是被“关联到”场景中的照相机上。例如：

```
SoXtFlyViewer *spaceShip = new SoXtFlyViewer;
spaceShip->setSceneGraph(root);
```

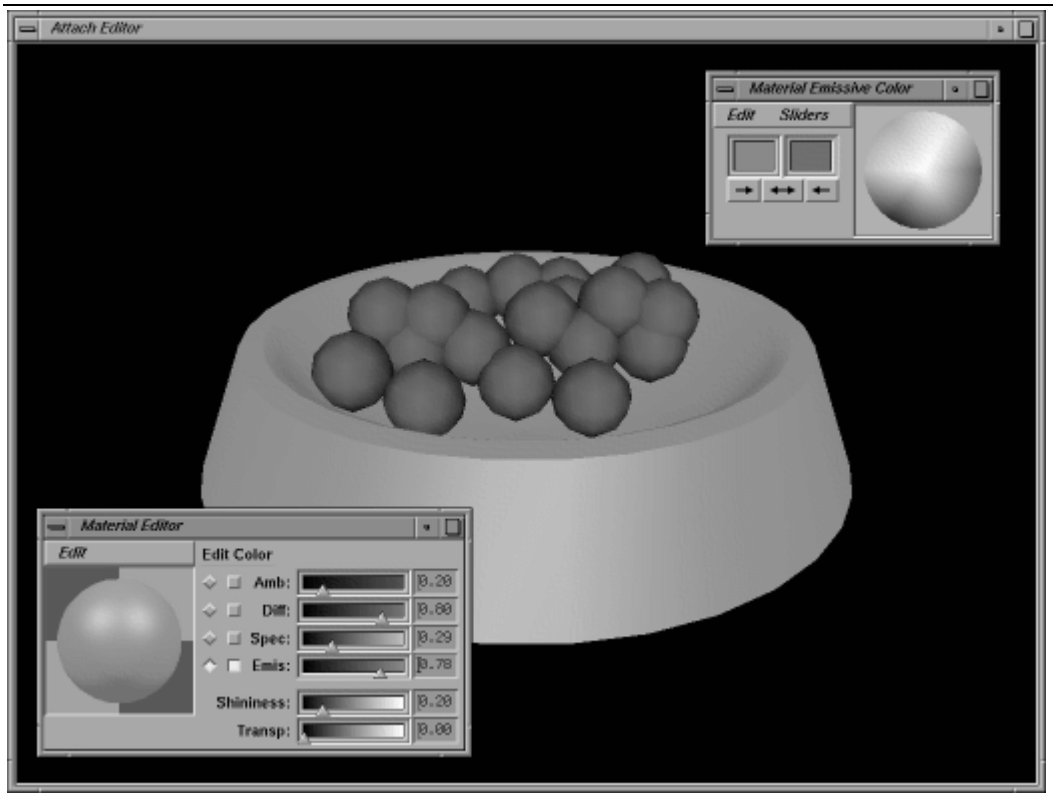
详见“观察器”章节中当观察器被关联到场景中时所做的讨论。

例 16-3 在应用程序的一个窗口中创建了一个渲染区组件，同时还创建了一个位于自身窗口内的材质编辑器组件。例子将材质编辑器关联到一个物体的材质节点上。图 16-2 显示了例子所产生的结果图像。

例 16-3 关联材质编辑器

```
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtMaterialEditor.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
```

图 16-2 创建在分离窗口中的材质编辑器和渲染区




```
// Build the render area in the applications main window
SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
myRenderArea->setSize(SbVec2s(200, 200));

// Build the material editor in its own window
SoXtMaterialEditor *myEditor = new SoXtMaterialEditor;

// Create a scene graph
SoSeparator *root = new SoSeparator;
SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
SoMaterial *myMaterial = new SoMaterial;

root->ref();
myCamera->position.setValue(0.212482, -0.881014, 2.5);
myCamera->heightAngle = M_PI/4;
root->addChild(myCamera);
root->addChild(new SoDirectionalLight);
root->addChild(myMaterial);

// Read the geometry from a file and add to the scene
SoInput myInput;
if (!myInput.openFile("dogDish.iv"))
   exit (1);
```

```
   SoSeparator *geomObject = SoDB::readAll(&myInput);
   if (geomObject == NULL)
      exit (1);
   root->addChild(geomObject);

   // Set the scene graph
   myRenderArea->setSceneGraph(root);

   // Attach material editor to the material
   myEditor->attach(myMaterial);

   // Show the application window and the material editor
   myRenderArea->setTitle("Attach Editor");
   myRenderArea->show();
   SoXt::show(myWindow);
   myEditor->show();

   // Loop forever
   SoXt::mainLoop();
}
```

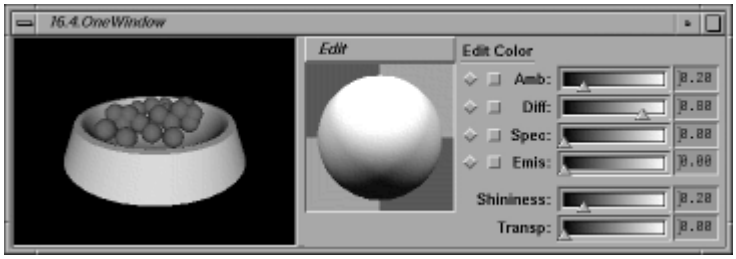
例 16-4 在应用程序的一个窗口中同时创建了一个渲染区组件和一个材质编辑器组件。它使用 Motif-compliant 形式的 widget，在同一个窗口中来安排这两个组件。编辑器被关联到一个物体的材质节点上。图 16-3 显示了这个例子初始的屏幕图像。

例 16-4 将两个组件放置在一个相同的窗口中

```
#include <Xm/Form.h>
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtMaterialEditor.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSeparator.h>

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);
   // Build the form to hold both components
```

```
Widget myForm = XtCreateWidget("Form",
         xmFormWidgetClass, myWindow, NULL, 0);

// Build the render area and Material Editor
SoXtRenderArea *myRenderArea = new SoXtRenderArea(myForm);
myRenderArea->setSize(SbVec2s(200, 200));
SoXtMaterialEditor *myEditor =
         new SoXtMaterialEditor(myForm);

// Lay out the components within the form
Arg args[8];
XtSetArg(args[0], XmNtopAttachment, XmATTACH_FORM);
XtSetArg(args[1], XmNbottomAttachment, XmATTACH_FORM);
XtSetArg(args[2], XmNleftAttachment, XmATTACH_FORM);
XtSetArg(args[3], XmNrightAttachment, XmATTACH_POSITION);
XtSetArg(args[4], XmNrightPosition, 40);
XtSetValues(myRenderArea->getWidget(), args, 5);
XtSetArg(args[2], XmNrightAttachment, XmATTACH_FORM);
XtSetArg(args[3], XmNleftAttachment, XmATTACH_POSITION);
XtSetArg(args[4], XmNleftPosition, 41);
XtSetValues(myEditor->getWidget(), args, 5);

// Create a scene graph
SoSeparator *root = new SoSeparator;
SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
SoMaterial *myMaterial = new SoMaterial;


root->ref();
myCamera->position.setValue(0.212482, -0.881014, 2.5);
myCamera->heightAngle = M_PI/4;
```

图 16-3 使用材质编辑器组件来编辑场景



```
   root->addChild(myCamera);
   root->addChild(new SoDirectionalLight);
   root->addChild(myMaterial);

   // Read the geometry from a file and add to the scene
   SoInput myInput;
   if (!myInput.openFile("dogDish.iv"))
      exit (1);
   SoSeparator *geomObject = SoDB::readAll(&myInput);
   if (geomObject == NULL)
      exit (1);
   root->addChild(geomObject);

   // Make the scene graph visible
   myRenderArea->setSceneGraph(root);

   // Attach the material editor to the material in the scene
   myEditor->attach(myMaterial);

   // Show the main window
   myRenderArea->show();
   myEditor->show();
   SoXt::show(myForm); // this calls XtManageChild
   SoXt::show(myWindow); // this calls XtRealizeWidget

   // Loop forever
   SoXt::mainLoop();
}
```

观察器

像 examiner viewer 和 fly viewer 这样的观察器组件，都可以通过修改照相机的位置，从而影响观察场景的效果。examiner viewer 使用一个虚拟不可见的轨迹球围绕一个用户感兴趣的位置来旋转场景。当使用 fly viewer 时，移动鼠标会产生使观察器向上、下、左、右倾斜的效果，也可以沿着面向观察器的方向进行移动。

所有观察器都内建以下组件：

- 用来显示场景的渲染区
- 在边框两侧的拇指滚轮（Thumbwheel）和滚动条，不同的观察器有不同的功能。
- 鼠标右键控制的弹出菜单

- 位于右侧边框上的，表示弹出菜单中某些操作的快捷按钮
- 位于窗口左上角，可选的应用程序图标。

图 16-4 展示了 examiner viewer 的一个例子。

构造观察器

当我们构造一个观察器时，我们可以指定观察器是一个浏览观察器（browser viewer）(BROWSER;缺省方式)，还是一个编辑观察器（EDITOR）。如果一个浏览观察器创建了一个照相机节点（见下面的章节），则当删除这个观察器时，那个照相机节点也会从场景中删除掉。如果一个编辑观察器创建了一个照相机节点，当删除这个观察器时，那个照相机节点仍将保留在场景中。

观察器的构造函数还带有另外一个参数，这个参数用来指定观察器内部将创建什么部件。缺省情况下，会创建 decoration（**表示窗口边框上的拇指滚轮、滚动条、快捷按钮等 UI 界面。译者注**）和弹出菜单。例如，examiner viwer 的构造函数：

```
SoXtExaminerViewer( Widget parent = NULL,
                    const char * name = NULL,
                    SbBool buildInsideParent = TRUE,
                    SoXtFullViewer::BuildFlag buildFlag = BUILD_ALL,
                    SoXtViewer::Type type = BROWSER);
```

参数 *buildFlag* 可以是下列数值：

BUILD_NONE	不创建 decoration 和弹出菜单
BUILD_DECORATION	只创建 decoration
BUILD_POPUP	只创建弹出菜单
BUILD_ALL	同时创建 decoration 和弹出菜单

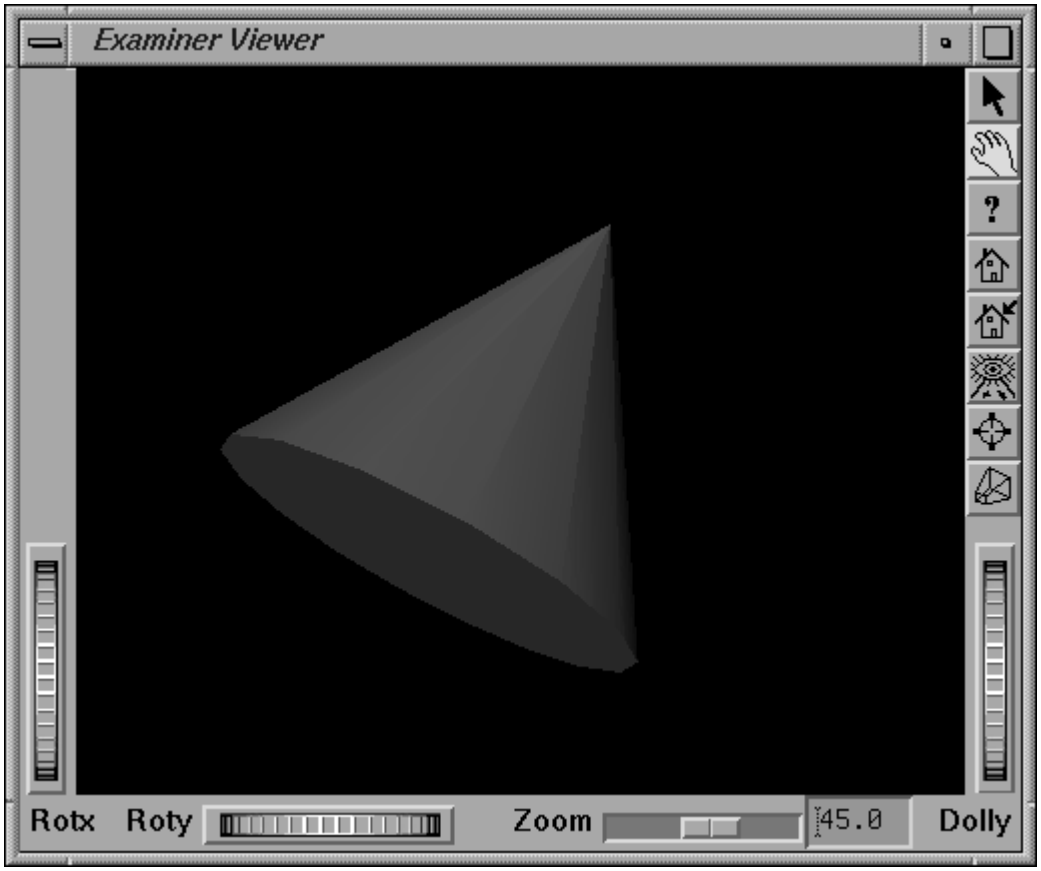
提示：如果用户不需要观察器的 decoration，可以在构造的时候使用 *buildFlag* 参数禁止创建 decoration，这样做可以改善性能。

为观察器指定场景

当我们调用观察器的**setSceneGraph()**方法后，将会自动发生若干个事件。首先，观察器将会搜索整个场景，寻找一个照相机节点。如果它在场景中发现了照相机节点，那么观察器就会使用这个照相机。否则如果没有找到任何照相机节点，观察器则会自己增加上一个。然后，观察器还会向场景增加上顶灯、绘制风格、光照模型等节点（下面的段落将会更详细地描述这些步骤）。

调用**setSceneGraph(NULL)**将会使场景与观察器组件分离。如果观察器自己创建了一个照相机，并且观察器是一个浏览类型的，那么观察器将会从场景中删除掉那个照相机。如果观察器是一个编辑类型的，那么观察器会保留下照相机，因为这时照相机是连同场景一起被保存的（**If the viewer is an editor, it leaves the camera, since the view is saved along with the scene graph，英文的原意是：因为这时观察器是连同场景一起被保存的。联系到上下文的意思，以及后面章节的内容，这里应该是英文作者的一个错误，这里不应该是观察器，而应该是照相机。即应该是：照相机是连同场景一起被保存的。译者注**）。当场景从观察器中删除后，不论是那种类型的观察器，顶灯节点都将会同时被删除。

图 16-4 Examiner Viewer



照相机

所有的观察器都会从场景的根节点向下搜索第一个照相机节点。如果观察器找到了一个照相机，它就会使用这个照相机。如果它没有找到，观察器就会自己创建一个（缺省情况下创建的是**SoPerspectiveCamera**类型的照相机）。如果观察器是一个编辑类型的，它会将照相机插到场景根节点之下的位置，如图 16-5 所示。当我们要保存场景时，这个新创建的照相机将随着场景一起被保存到文件中。如果观察器是一个浏览类型的，则它会在场景的上面插入一个照相机，如图 16-6 所示。当我们要保存场景时，这个照相机将不会和场景保存在一起，并且当场景和观察器分离后，照相机还会被删除掉。

图 16-5 为编辑观察器插入一个照相机

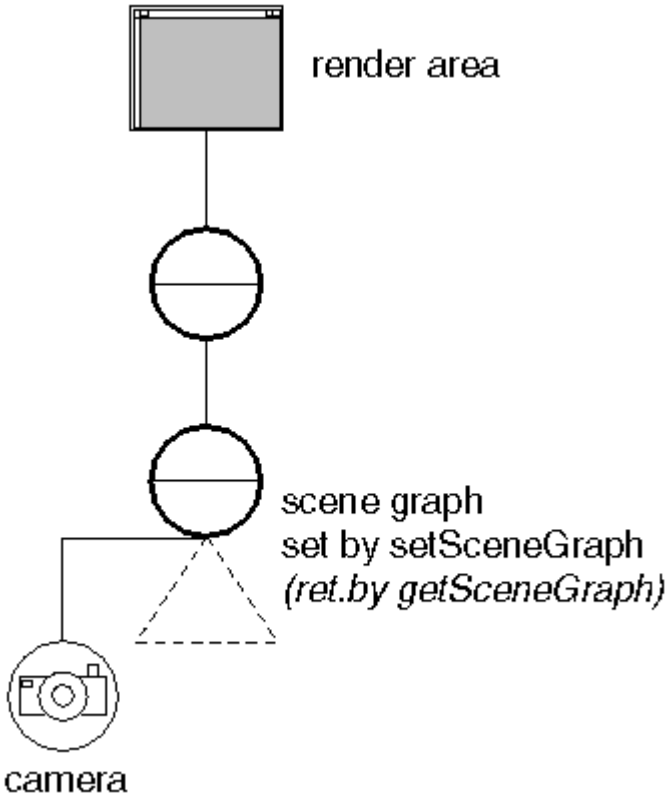
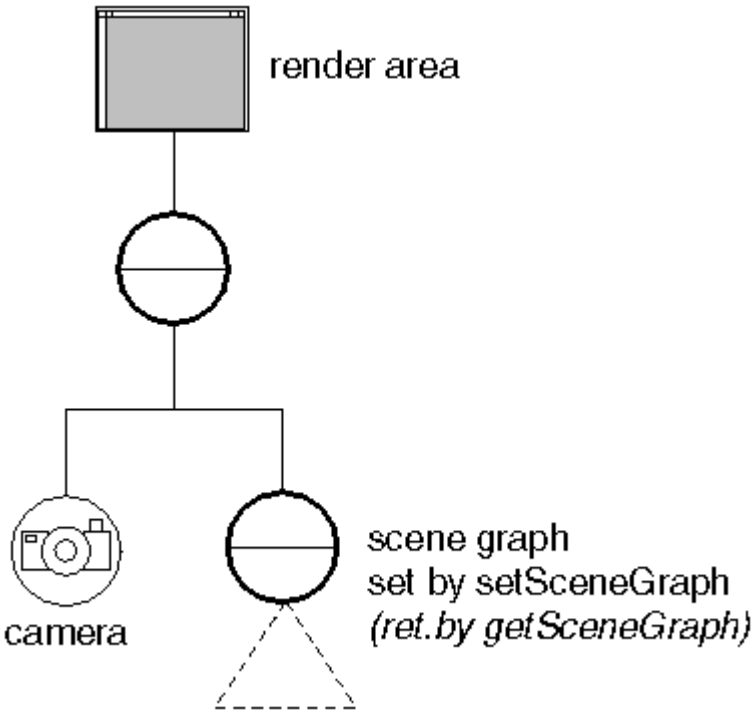


图 16-6 为浏览观察器插入一个照相机



灯光

观察器组件缺省情况下还会向场景增加上一个直射灯光源。观察器会不断地改变光源的

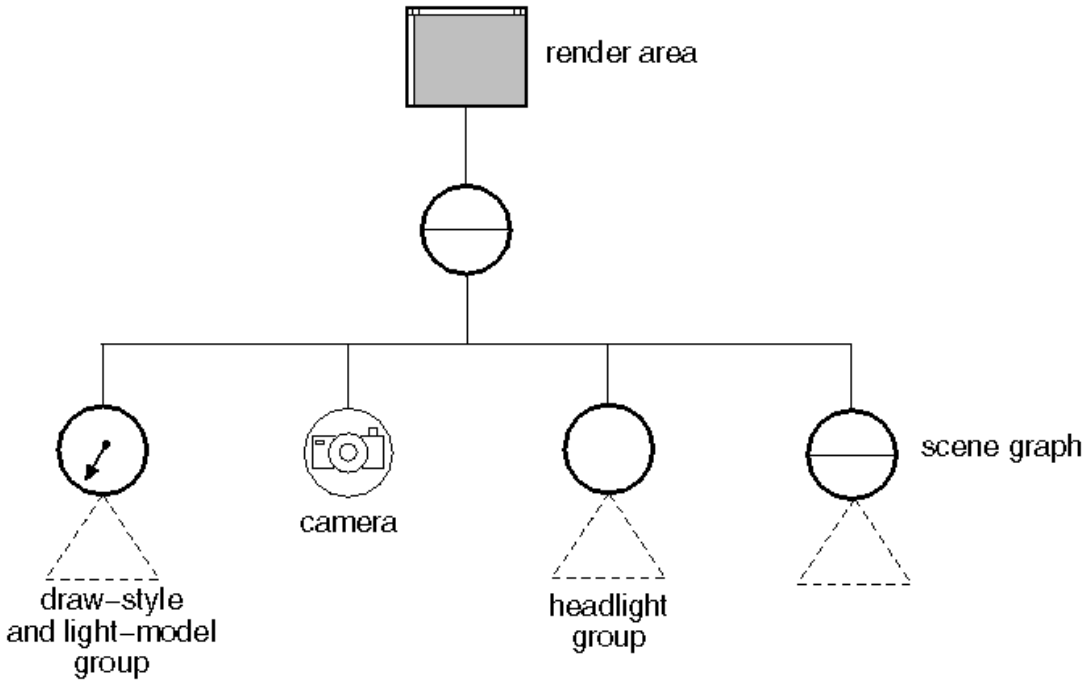
位置，以便让光源跟踪照相机，起到照亮照相机视野的作用。顶灯组节点会增加到场景中照相机节点之后位置上。如果在将场景保存到文件中时，不希望保存顶灯节点，我们可以让场景与观察器分离，或者关闭顶灯（详见*Open Inventor C++ Reference Manual*书中**SoXtViewer**的**setHeadlight()**方法）。

观察器的绘制风格

所有的观察器都带有一个弹出菜单，通过它可以允许我们修改整个场景的绘制风格。有时，当观察器修改完绘制风格后，还会同时修改光照模型（例如，框线绘制风格将使用BASE_COLOR 光照模型）。当场景和观察器相关联时，观察器会在场景的上面插入绘制风格和光照模型节点，如图 16-7 所示。下面的列表将介绍绘制风格的选择以及相对应的光照模型的变化：

VIEW_AS_IS	忽略观察器的绘制风格和灯光模型（缺省方式）
VIEW_HIDDEN_LINE	强制所有的物体以框线方式显示，修改灯光模型为 BASE_COLOR.这种类型只显示多边形物体前面的边线（隐藏后面的线）（**即以消隐方式显示物体，译者注**）
VIEW_NO_TEXTURE	强制所有物体渲染时不使用材质
VIEW_LOW_COMPLEXITY	强制所有物体以低复杂度和无材质方式渲染
VIEW_LINE	强制所有物体以框线方式显示，修改灯光模型为 BASE_COLOR
VIEW_POINT	强制所有物体以离散点方式显示，修改灯光模型为 BASE_COLOR。点的大小为 3.0
VIEW_BBOX	强制所有物体以包围盒方式显示

图 16-7 插入绘制风格和灯光模型



观察器的绘制类型

上面讨论的绘制风格可以在照相机静止不动时或在用户以交互方式移动照相机时影响到场景里物体的显示。当设置完绘制风格后，我们可以在STILL和INTERACTIVE这两种绘制类型中做选择，以此来决定绘制风格应该在什么状态下影响场景的显示。调用**SoXtViewer**的**setDrawStyle()**方法可以设置绘制风格和绘制类型。

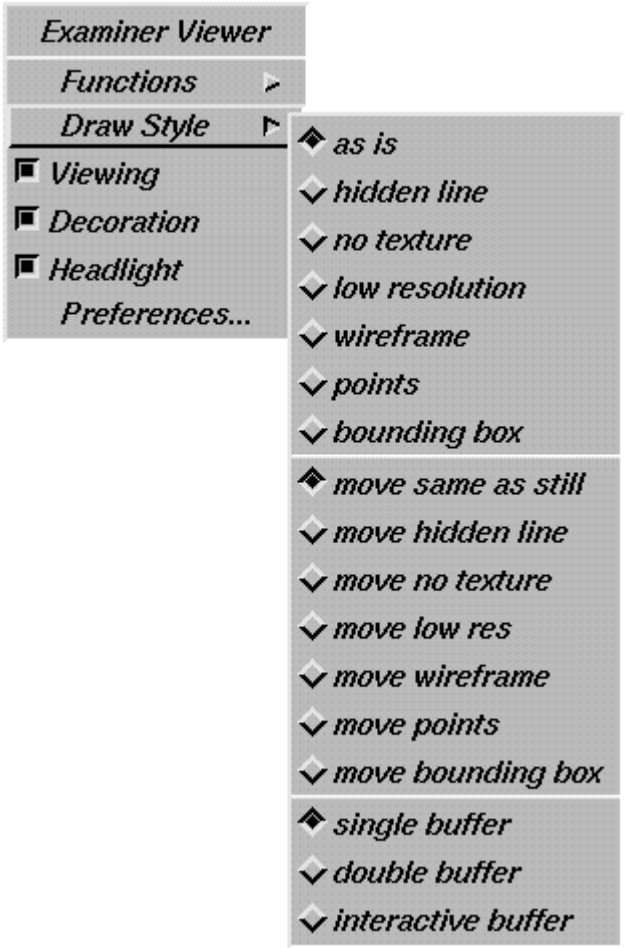
```
setDrawStyle(SoXtViewer::DrawType type,SoXtViewer::DrawStyle style)
```

例如：

```
setDrawStyle(SoXtViewer::INTERACTIVE, SoXtViewer::VIEW_LINE);
```

如图 16-8 所示，观察器的弹出菜单中列有 STILL 类型下绘制风格选项、INTERACTIVE 类型下绘制风格选项、以及缓冲类型选项。

图 16-8 观察器的弹出菜单



SoXtViewer 包含的方法

调用**SoXtViewer**的**setBufferingType()**方法可以指定观察器是使用单缓冲、双缓冲、或是二者相结合。缺省的缓冲类型是双缓冲。双缓冲可以使屏幕刷新更流畅，不过这种类型提供的颜色数量比较少。Inventor支持的缓冲类型如下：

SoXtViewer::BUFFER_SINGLE	只使用单缓冲；在两次刷新屏幕的时候，屏幕会发生闪烁
SoXtViewer::BUFFER_DOUBLE	在后台缓冲（back buffer）中重绘场景，然后交换缓冲
SoXtViewer::BUFFER_INTERACTIVE	只当用户正在做交互式工作时使用双缓冲，其它时候使用单缓冲

在**SoXtViewer**中其它比较有用的方法：

setHeadlight()	打开或关闭顶灯。缺省是打开顶灯
setViewing()	打开或关闭观察器。当关闭观察器时，所有在渲染区上发生的事件都将被发送到场景中做处理。
viewAll()	观察整个场景
setAutoClipping()	打开或关闭自动剪裁功能。当打开时，观察器会不断地调整照相机的远近两个剪裁平面来包围住场景包围盒。缺省情况下是打开自动剪裁功能。
saveHomePosition()	保存当前的照相机参数，以便在以后可以快速地重新回到这个位置上
resetToHomePosition()	将照相机的位置设置成前面保存过的 home 位置上
setStereoViewing()	渲染两遍场景，在两次渲染之间要偏移一下照相机的位置。当以这种方式观察场景时，必须要使用立体眼镜才能看到效果。（这个特性是硬件相关的。请读者查看自己的 Inventor 发行注释中是否有支持这种特性的信息）
setSteroOffset()	当以立体方式观察场景时，设置两个眼睛之间的间隔

更多细节请查看*Open Inventor C++ Reference Manual*中的**SoXtViewer**条目。

SoXtFullViewer 包含的方法

SoXtFullViewer类是从**SoXtViewer**派生出来的，它是所有在渲染区周围包含有decoration的观察器类的抽象基类。decoration是由拇指滚轮、滚动条、和按钮组成的。**setDecoration()**方法可以设置显示或隐藏decoration。**setPopupMenuEnabled()**方法可以设置使用或禁用观察器的弹出菜单。

我们可以向观察器的左上角增加上一些应用程序的图标。使用下面的方法增加图标：

addAppPushButton()	将应用程序的按钮增加到按钮列表的尾部。
insertAppPushButton()	按照给定的按钮列表索引值，将按钮插入到相应的位置上
removeAppPushButton()	从按钮列表中删除一个按钮

更多细节请查看*Open Inventor C++ Reference Manual*中的**SoXtFullViewer**条目。

例 16-5 将创建一个简单的场景，场景中带有一个材质节点和一个托盘物体。然后又创建了一个浏览类型的 examiner viewer，并将它与场景相关联。场景中的照相机和灯光节点是观察器自动创建的。

例 16-5 使用浏览类型的 examiner viewer

```
#include <Inventor/SoDB.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <Inventor/nodes/SoSeparator.h>

main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);

   // Build the viewer in the application's main window
   SoXtExaminerViewer *myViewer =
      new SoXtExaminerViewer(myWindow);

   // Read the geometry from a file and add to the scene
   SoInput myInput;
   if (!myInput.openFile("dogDish.iv"))
      exit (1);
   SoSeparator *geomObject = SoDB::readAll(&myInput);
   if (geomObject == NULL)
      exit (1);

   // Attach the viewer to the scene graph
   myViewer->setSceneGraph(geomObject);

   // Show the main window
   myViewer->show();
   SoXt::show(myWindow);

   // Loop forever
   SoXt::mainLoop();
}
```

使用 3D 剪贴板

本节将讨论一种 Inventor 所提供的用来在应用程序间交换 Inventor 数据的简便方法。Inventor 拷贝和粘贴数据的方法遵循 X 联盟的 *Inter-Client Communication Conventions Manual* (ICCCM)规范，这种规范发表于 1989 年 7 月，只要用于指导在相互交换数据的时候，应用程序如何处理它们之间的通信问题。

Inventor 当前支持两种数据类型：Inventor 数据和字符串。如果我们需要拷贝粘贴额外的数据类型，或者我们希望对 Inventor 提供的拷贝粘贴功能做更多的控制，可以直接使用 Motif 或 Xt 数据交换方法。更多的信息请查看 *X Toolkit Intrinsics Programming Manual* by Adrian Nye and Tim O'Reilly (Sebastopol, Ca.: O'Reilly & Associates, 1990)。

SoXtClipboard类根据ICCCM规范来处理数据交换的细节。这个类包含一个构造方法、一个**copy()**方法、以及一个**paste()**方法。

创建 SoXtClipboard 实例

SoXtClipboard的构造函数语法如下：

SoXtClipboard(Widget *w*, Atom *selectionAtom* = _XA_CLIPBOARD_);

剪贴板要和一个像“渲染区”或“顶层 widget”这样特定的 widget 相关联，例如，我们可以传入：

```
renderArea->getWidget()
```

作为构造函数的第一个参数。

X工具包支持一系列的选择类型（主要、次要、以及剪贴板；这些都是以*selection atoms*来使用的）。缺省情况下，Inventor支持剪贴板选择（_XA_CLIPBOARD_）。如果我们需要执行从主要到次要选择的数据传输，我们必须要在构造函数中为**SoXtClipboard**指定选择的类型。然而在绝大多数情况下，我们只需要使用缺省的选择类型即可。

将数据拷贝到剪贴板上

使用Inventor的三个**copy()**方法可以将数据拷贝到**SoXtClipboard**上。我们可以拷贝节点、路径、或路径列表：

copy(SoNode **node*, Time *eventTime*);
copy(SoPath **path*, Time *eventTime*);
copy(SoPathList *&pathList*, Time *eventTime*);

copy()和**paste()**方法都需要设置一个事件时间的参数。事件时间是一个当触发拷贝或粘贴的用户事件发生时的时间戳（The **copy()** and **paste()** methods require an event time, which is the time stamp from the user event that triggered the copy or paste request）。例如，事件可以是一个键盘按下事件或一个菜单选择事件，事件主要用于同步X 服务器拷贝粘贴的请求。在后台，数据被拷贝到一个字节流中，对任何请求得到它的X客户端来说都是有效的。

从剪贴板拷贝数据

paste()方法需要一个回调函数，这个回调函数将在接收剪贴数据时被调用。不管原来拷贝的是什么，剪贴数据的类型总是一个路径列表。

paste(Time *eventTime*, SoXtClipboardPasteCB *pasteDoneFunc*, void *userData* = NULL);

paste()方法将从X 服务器中请求数据，当数据准备好后，X 服务器会调用*pasteDoneFunc*回调函数。粘贴是异步的。它只是简单地向X服务器发送一个请求剪贴数据的命令，然后就直接返回。当数据准备好后，X 服务器会调用*pasteDoneFunc*回调函数，并将用户数据以及被粘贴的路径列表作为参数传到*pasteDoneFunc*函数中。如果没有准备好的数据，则*pasteDoneFunc*将永远不会被调用。当应用程序结束后，应用程序要负责删除这些剪贴数据的路径列表。

提示：**SoXtClipboard**可以简单和**SoSelection**一起使用。可以从selection节点中获得一个路径列表，然后告诉剪贴板来拷贝这个路径。

第十七章在 Inventor 中使用 OpenGL

本章目标

在阅读完本章节后，我们将有能力做下列事情：

- 创建 Inventor 回调节点，在回调函数中调用 OpenGL 库函数。
- 了解 Inventor 是如何使用和影响 OpenGL 的状态变量
- 编写一个同时使用 Inventor 与 OpenGL 的程序，并在程序中使用 **SoGLRenderAction**来渲染场景。
- 使用颜色索引模式（color index mode）。

（高级内容）

本章将讨论如何在同一个窗口中同时调用 Inventor 和 OpenGL 库。本章包括几个以不同方式同时使用 Inventor 与 OpenGL 的例子程序。从表 17-1 到表 17-9 展示了 Inventor 与 OpenGL 状态是如何相互影响的。本章的所有内容都是比较高级的内容。

简介

本章只适合于有 OpenGL 编程经验的开发人员阅读，所以不会刻意地去介绍 OpenGL。在阅读本章内容之前，我们应该至少要先阅读 1 至 5 章和第 9 章的内容。在开始将 OpenGL 与 Inventor 同时使用前，我们需要对 Inventor 数据库（第 1 至 4 章）、Inventor 动作（第 9 章）、和 Inventor 事件处理（第 10 章）有一个基本的了解。

同时使用 OpenGL 与 Inventor 的首选方式是子类化。通过子类化，我们可以创建一个能调用OpenGL的新节点。这种处理方式已经超过了本章内容所涉及的范围，它将在 *The Inventor Toolmaker* 书中做详细的介绍。子类化的另外一个优点是新类可以读出或写入到 Inventor 的文件中（而本章所要讨论的回调节点不能被读写到文件中）。

需要注意的是，Inventor 只是从 OpenGL 中继承了用于渲染操作的状态变量。其它像拾取、计算包围盒、写文件等这些 Inventor 的特性都不会使用 OpenGL，也不会直接修改 OpenGL 的状态变量。例如，我们可以直接将观察矩阵（viewing matrix）传给 OpenGL，然后使用 Inventor 绘制一个没有照相机的场景。然而，如果我们此时试图去拾取一个物体，Inventor 将无法得知用于拾取操作的观察矩阵，因为 Inventor 不会使用 OpenGL 中的拾取操作。

我们可以有多种方法同时使用Inventor与OpenGL。一种比较简单的方法是通过增加一个回调节点（**SoCallback**，见例 17-2）向场景数据库增加上OpenGL定制渲染功能。回调节点允许我们定义一个回调函数，当节点执行各种不同的动作（渲染、拾取、包围盒计算）时都会调用此回调函数。**SoCallback**节点和事件回调节点之间的不同之处在于，**SoCallback**节点对所有的场景操作都提供回调功能，而事件回调节点只对事件进行处理。

同时使用Inventor与OpenGL的第二种方法是创建一个GLX窗口，编写OpenGL和Inventor调用代码，最后执行**SoGLRenderAction**。具体见例 17-3。例如，我们可以先创建一个GLX

窗口，清空背景，在窗口中做一些渲染初始化的工作，建立观察矩阵，然后通过对场景应用**SoGLRenderAction**，使用Inventor绘制场景，。或者我们也可以使用Inventor来创建照相机、灯光、和材质等节点，然后使用OpenGL的代码绘制场景。只要我们遵照下面章节中将要介绍的OpenGL状态的使用规则来编写代码，我们就可以随心所欲地同时使用OpenGL与Inventor了（注意，这是一种很“刺激”用法，不大适合胆小的人（Note that this is an advanced feature, not for the faint of heart））。

OpenGL 状态变量与 Inventor

如果我们需要同时调用 Inventor 和 OpenGL 库，首先需要了解表 17-1 至表 17-9 中列出的 OpenGL 状态变量，以及那些 Inventor 节点（或动作）将修改这些变量。如果 Inventor 使用了某些 OpenGL 状态变量的当前值，但却从来没有修改过它们，则下面表中不会列出这些变量。如须了解完整的 OpenGL 状态变量及其缺省值的话，请查阅 *OpenGL Programming Guide*。对于下面表中列出的这些变量，推荐它们使用在 OpenGL 中的缺省值，但有两个例外：打开 Z 缓冲和使用 RGB 颜色模式。

请注意，**SoGLRenderAction**的构造函数需要一个参数，用来指定是否需要继承当前的OpenGL状态变量值。如果我们将这个参数设置为TRUE，Inventor将会继承OpenGL中的数值，如果我们将这个参数设置为FALSE（缺省方式），Inventor将会设置适合自己的缺省数值（见第 9 章）。

要保存和恢复OpenGL的状态，可以调用OpenGL的**pushAttributes()**和**popAttributes()**命令。例如，如果我们在回调节点中修改了一些OpenGL的状态变量，当回调节点结束时我们就需要恢复这些状态的原来数值。注意，如果我们的回调节点以**pushAttributes()**作为开始，以**popAttributes()**作为结束，但如果一个渲染操作在中间代码处发生异常中断，这时**popAttributes()**将永远不会被调用了，那些状态变量也就不能恢复了。

表 17-1 OpenGL 状态变量：当前值与关联的数据

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_CURRENT_COLOR	形体节点，材质节点，颜色节点
GL_CURRENT_INDEX	颜色索引节点，形体节点
GL_CURRENT_TEXTURE_COORDS	形体节点，TextureCoordinate2
GL_CURRENT_NORMAL	形体节点，法线节点
GL_CURRENT_RASTER_POSITION	2D 文本
GL_CURRENT_RASTER_COLOR	2D 文本
GL_CURRENT_RASTER_INDEX	2D 文本
GL_CURRENT_RASTER_POSITION_-VALID	2D 文本

表 17-2 OpenGL 状态变量：几何变换状态

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_MODELVIEW_MATRIX	几何变换节点，照相机节点
GL_PROJECTION_MATRIX	照相机节点
GL_TEXTURE_MATRIX	2D 纹理变换节点
GL_VIEWPORT	照相机节点
GL_DEPTH_RANGE	照相机节点
GL_MODELVIEW_STACK_DEPTH	几何变换节点
GL_TEXTURE_STACK_DEPTH	2D 纹理变换节点
GL_MATRIX_MODE	照相机节点，2D 纹理变换节点

表 17-3 OpenGL 状态变量：颜色

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_FOG_COLOR	环境节点
GL_FOG_INDEX	环境节点
GL_FOG_DENSITY	环境节点
GL_FOG_START	环境节点
GL_FOG_END	环境节点
GL_FOG_MODE	环境节点
GL_FOG	环境节点
GL_SHADE_MODEL	光照模型（如果在颜色索引模式下）

表 17-4 OpenGL 状态变量：灯光

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_LIGHTING	光照模型
GL_COLOR_MATERIAL	形体节点
GL_MATERIAL_PARAMETER	形体节点
GL_MATERIAL_FACE	形体节点
GL_AMBIENT	形体节点，材质节点
GL_DIFFUSE	形体节点，材质节点
GL_SPECULAR	形体节点，材质节点
GL_EMISSION	形体节点，材质节点
GL_SHININESS	形体节点，材质节点
GL_LIGHT_MODEL_AMBIENT	形体节点，材质节点
GL_LIGHT_MODEL_LOCAL_VIEWER	形体节点，材质节点
GL_LIGHT_MODEL_TWO_SIDE	形体提示节点（Shape Hints）

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_AMBIENT	灯光节点
GL_DIFFUSE	灯光节点
GL_SPECULAR	灯光节点
GL_POSITION	灯光节点
GL_CONSTANT_ATTENUATION	环境节点
GL_LINEAR_ATTENUATION	环境节点
GL_QUADRATIC_ATTENUATION	环境节点
GL_SPOT_DIRECTION	灯光节点
GL_SPOT_EXPONENT	灯光节点
GL_SPOT_CUTOFF	灯光节点
GL_LIGHT*i*	灯光节点
GL_COLOR_INDEXES	灯光节点

表 17-5 OpenGL 状态变量：光栅化

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_POINT_SIZE	绘制风格节点
GL_POINT_SMOOTH	渲染动作
GL_LINE_WIDTH	绘制风格节点
GL_LINE_SMOOTH	渲染动作
GL_LINE_STIPPLE_PATTERN	绘制风格节点
GL_LINE_STIPPLE	绘制风格节点
GL_CULL_FACE	形体提示节点
GL_CULL_FACE_MODE	形体提示节点
GL_POLYGON_MODE	绘制风格节点
GL_POLYGON_STIPPLE	形体节点（如果透明类型是 SCREEN_DOOR）

表 17-6 OpenGL 状态变量：纹理

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_TEXTURE_*x*	2D 纹理节点
GL_TEXTURE	2D 纹理节点
GL_TEXTURE_WIDTH	2D 纹理节点
GL_TEXTURE_HEIGHT	2D 纹理节点
GL_TEXTURE_COMPONENTS	2D 纹理节点
GL_TEXTURE_MIN_FILTER	复杂度节点

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_TEXTURE_MAG_FILTER	复杂度节点
GL_TEXTURE_WRAP_*x*	2D 纹理节点
GL_TEXTURE_ENV_MODE	2D 纹理节点
GL_TEXTURE_ENV_COLOR	2D 纹理节点
GL_TEXTURE_GEN_*x*	纹理坐标函数节点
GL_EYE_LINEAR	纹理坐标函数节点
GL_OBJECT_LINEAR	纹理坐标函数节点
GL_TEXTURE_GEN_MODE	纹理坐标函数节点

表 17-7 OpenGL 状态变量：像素操作

GL_BLEND	渲染动作，2D 纹理节点
GL_BLEND_SRC	渲染动作，2D 纹理节点
GL_BLEND_DST	渲染动作，2D 纹理节点

表 17-8 OpenGL 状态变量：像素

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_UNPACK_ALIGNMENT	2D 纹理节点
GL_*_SCALE (* = RED; GREEN; BLUE; ALPHA)	2D 纹理节点
GL_*_BIAS (* = RED; GREEN; BLUE; ALPHA)	2D 纹理节点

表 17-9 OpenGL 状态变量：杂项

OpenGL 状态变量	修改这些变量的 Inventor 节点
GL_LIST_BASE	2D，3D 文本节点
GL_LIST_INDEX	Separator，2D，3D 文本节点
GL_LIST_MODE	Separator，2D，3D 文本节点

颜色索引模式

我们可以创建一个支持在RGB模式下或颜色索引模式下（也称颜色映射模式）OpenGL渲染功能的X 窗口。如果我们使用颜色索引模式，则一定要保证首先装载颜色表。例 17-1 将演示如何为**SoXtRenderArea**设置颜色表。也可见*Open Inventor C++ Reference Manual*书中的**SoXtRenderArea::setColorMap()**描述。

如果我们当前使用的光照模型是BASE_COLOR，可以使用**SoColorIndex**节点来指定颜

色表中的索引值。

如果我们当前使用的光照模型是PHONG，就需要使用**SoMaterialIndex**节点为环境光、漫反射、高光颜色指定在颜色表中的索引值。这个节点还包含指定表面发光度和透明度（但不能指定自发光数据）的域变量。可以这样认为，颜色表中包含有一个从环境光颜色渐变到高光颜色的所有颜色的平台（It expects the color map to contain a ramp from ambient to diffuse to specular colors）。

提示：如果我们向场景中同时插入**SoMaterialIndex**和**SoMaterial**节点，那么这个场景将既可以用在RGB模式窗口下，也可以用在颜色索引模式窗口下。

例 17-1 使用颜色索引模式

```
#include <Inventor/SoDB.h>
#include <Inventor/SoInput.h>
#include <Inventor/nodes/SoNode.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/viewers/SoXtExaminerViewer.h>
#include <GL/glx.h>

// Window attribute list to create a color index visual.
// This will create a double buffered color index window
// with the maximum number of bits and a zbuffer.
int attribList[] = {
   GLX_DOUBLEBUFFER,
   GLX_BUFFER_SIZE, 1,
   GLX_DEPTH_SIZE, 1,
   None };

// List of colors to load in the color map
static float colors[3][3] = {{.2, .2, .2}, {.5, 1, .5},
         {.5, .5, 1}};

static char *sceneBuffer = "\
#Inventor V2.0 ascii\n\
\
Separator { \
   LightModel { model BASE_COLOR } \
   ColorIndex { index 1 } \
   Coordinate3 { point [ -1 -1 -1, -1 1 -1, 1 1 1, 1 -1 1] } \
   FaceSet {} \
```

```
   ColorIndex { index 2 } \
   Coordinate3 { point [ -1 -1 1, -1 1 1, 1 1 -1, 1 -1 -1] } \
   FaceSet {} \
} ";

void
main(int , char **argv)
{
   // Initialize Inventor and Xt
   Widget myWindow = SoXt::init(argv[0]);

   // Read the scene graph in
   SoInput in;
   SoNode *scene;
   in.setBuffer((void *)sceneBuffer, (size_t)
            strlen(sceneBuffer));
   if (! SoDB::read(&in, scene) || scene == NULL) {
      printf("Couldn't read scene\n");
      exit(1);
   }

   // Create the color index visual
   XVisualInfo *vis = glXChooseVisual(XtDisplay(myWindow),
      XScreenNumberOfScreen(XtScreen(myWindow)), attribList);
   if (! vis) {
      printf("Couldn't create visual\n");
      exit(1);
   }

   // Allocate the viewer, set the scene, the visual and
   // load the color map with the wanted colors.
   //
   // Color 0 will be used for the background (default) while
   // color 1 and 2 are used by the objects.
   //
   SoXtExaminerViewer *myViewer = new
            SoXtExaminerViewer(myWindow);
   myViewer->setNormalVisual(vis);
   myViewer->setColorMap(0, 3, (SbColor *) colors);
   myViewer->setSceneGraph(scene);
   myViewer->setTitle("Color Index Mode");

   // Show the viewer and loop forever...
```

```
   myViewer->show();
   XtRealizeWidget(myWindow);
   SoXt::mainLoop();
}
```

使用 SoCallback 节点

一个**SoCallback**节点的典型用途就是调用OpenGL。在回调函数的开始处，我们需要首先检查动作的类型，然后根据应用到节点上的动作类型做相应的处理。通常情况下，我们一般只对渲染动作感兴趣。

```
if(action->isOfType(SoGLRenderAction::getClassTypeId())){

   ...execute rendering code ..

}
```

缓冲

回调节点处理的结果是否可以被缓冲，取决于它当前处理的内容。例如，如果回调节点包含有一些物体，而这些物体的形状已经发生了改变，那么回调节点处理的结果就不能被缓冲。在例 17-2 中，回调节点创建了一个棋盘样式的背景图案，这个图案就可以被缓冲，因为它从来就没有发生过改变。

如果回调节点依赖一些位于Inventor之外的信息，而这些信息也许会被修改（例如一个全局变量等），那么这个回调节点也不能被缓冲。为了防止Inventor自动创建缓冲，可以在回调函数中调用**SoCacheElement::-invalidate()** 方法。例如：

```
void
myCallback(void *myData, SoAction *action)
{
   if (action->isOfType(SoGLRenderAction::getClassTypeId())){
      SoCacheElement::invalidate(action->getState());
         //makes sure this isn't cached
      //...make OpenGL calls that depend on a global variable...//
   }
}
```

在**SoCallback**节点中打开一个OpenGL显示列表时一定需要注意。请回忆在第 9 章中曾经描述的，Inventor渲染缓冲中包含有一个OpenGL显示列表。每次只能同时打开一个OpenGL的显示列表，位于回调节点之上的某个SoSeparator节点也许已经为缓冲打开了一个显示列

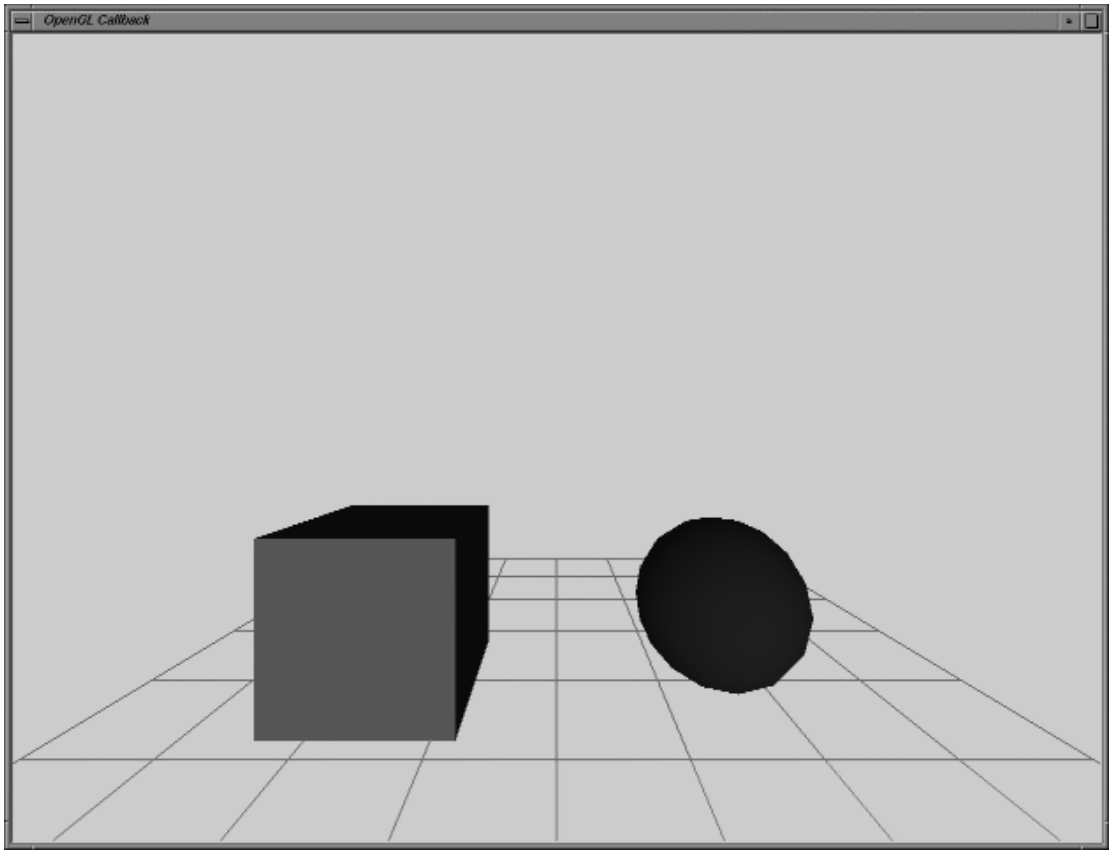
表。如果此时回调节点要打开第二个显示列表的话，就会产生一个错误。调用**SoCacheElement::anyOpen()**方法可以检查是否存在已经打开过的缓冲。

使用回调节点

例 17-2 将创建一个Inventor渲染区。它使用Inventor创建了一个红色立方体和一个蓝色球体，然后使用一个包含有OpenGL调用的**SoCallback**节点来绘制一个棋盘样式的“地板”。这个地板会被Inventor自动缓冲。注意，当窗口大小发生变化后，**SoXtRenderArea**会自动重绘场景图像。例 17-3 则是使用了一个GLX窗口，它不会自动重绘。

例 17-2 和例 17-3 所产生的场景图像完全一样，显示在下面的图 17-1 中。

图 17-1 同时使用 Inventor 和 OpenGL



例 17-2 使用回调节点

```
#include <GL/gl.h>
#include <Inventor/SbLinear.h>
#include <Inventor/Xt/SoXt.h>
#include <Inventor/Xt/SoXtRenderArea.h>
#include <Inventor/nodes/SoCallback.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoDirectionalLight.h>
```

```
#include <Inventor/nodes/SoLightModel.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoPerspectiveCamera.h>
#include <Inventor/nodes/SoSphere.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoTransform.h>

float   floorObj[81][3];

// Build a scene with two objects and some light
void
buildScene(SoGroup *root)
{
   // Some light
   root->addChild(new SoLightModel);
   root->addChild(new SoDirectionalLight);

   // A red cube translated to the left and down
   SoTransform *myTrans = new SoTransform;
   myTrans->translation.setValue(-2.0, -2.0, 0.0);
   root->addChild(myTrans);

   SoMaterial *myMtl = new SoMaterial;
   myMtl->diffuseColor.setValue(1.0, 0.0, 0.0);
   root->addChild(myMtl);

   root->addChild(new SoCube);

   // A blue sphere translated right
   myTrans = new SoTransform;
   myTrans->translation.setValue(4.0, 0.0, 0.0);
   root->addChild(myTrans);

   myMtl = new SoMaterial;
   myMtl->diffuseColor.setValue(0.0, 0.0, 1.0);
   root->addChild(myMtl);

   root->addChild(new SoSphere);
}

// Build the floor that will be rendered using OpenGL.
void
buildFloor()
```

```
{
   int a = 0;

   for (float i = -5.0; i <= 5.0; i += 1.25) {
      for (float j = -5.0; j <= 5.0; j += 1.25, a++) {
         floorObj[a][0] = j;
         floorObj[a][1] = 0.0;
         floorObj[a][2] = i;
      }
   }
}

// Draw the lines that make up the floor, using OpenGL
void
drawFloor()
{
   int i;

   glBegin(GL_LINES);
   for (i=0; i<4; i++) {
      glVertex3fv(floorObj[i*18]);
      glVertex3fv(floorObj[(i*18)+8]);
      glVertex3fv(floorObj[(i*18)+17]);
      glVertex3fv(floorObj[(i*18)+9]);
   }

   glVertex3fv(floorObj[i*18]);
   glVertex3fv(floorObj[(i*18)+8]);
   glEnd();

   glBegin(GL_LINES);
   for (i=0; i<4; i++) {
      glVertex3fv(floorObj[i*2]);
      glVertex3fv(floorObj[(i*2)+72]);
      glVertex3fv(floorObj[(i*2)+73]);
      glVertex3fv(floorObj[(i*2)+1]);
   }
   glVertex3fv(floorObj[i*2]);
   glVertex3fv(floorObj[(i*2)+72]);
   glEnd();
}

// Callback routine to render the floor using OpenGL
```

```
void
myCallbackRoutine(void *, SoAction *)
{
   glPushMatrix();
   glTranslatef(0.0, -3.0, 0.0);
   glColor3f(0.0, 0.7, 0.0);
   glLineWidth(2);
   glDisable(GL_LIGHTING);  // so we don't have to set normals
   drawFloor();
   glEnable(GL_LIGHTING);
   glLineWidth(1);
   glPopMatrix();
}


main(int, char **)
{
   // Initialize Inventor utilities
   Widget myWindow = SoXt::init("Example 17.1");

   buildFloor();

   // Build a simple scene graph, including a camera and
   // a SoCallback node for performing some GL rendering.
   SoSeparator *root = new SoSeparator;
   root->ref();

   SoPerspectiveCamera *myCamera = new SoPerspectiveCamera;
   myCamera->position.setValue(0.0, 0.0, 5.0);
   myCamera->heightAngle  = M_PI/2.0;  // 90 degrees
   myCamera->nearDistance = 2.0;
   myCamera->farDistance  = 12.0;
   root->addChild(myCamera);

   SoCallback *myCallback = new SoCallback;
   myCallback->setCallback(myCallbackRoutine);
   root->addChild(myCallback);

   buildScene(root);

   // Initialize an Inventor Xt RenderArea and draw the scene.
   SoXtRenderArea *myRenderArea = new SoXtRenderArea(myWindow);
   myRenderArea->setSceneGraph(root);
```

```
    myRenderArea->setTitle("OpenGL Callback");
    myRenderArea->setBackgroundColor(SbColor(.8, .8, .8));
    myRenderArea->show();

    SoXt::show(myWindow);
    SoXt::mainLoop();
}
```

在一个 GLX 窗中使用渲染动作

例 17-3 将创建一个GLX窗口，编写OpenGL和Inventor调用代码，最后执行GL渲染动作（**SoGLRenderAction**）。它在同一个窗口内，使用OpenGL来渲染棋盘样式的“地板”，使用Inventor来渲染红色立方体和蓝色球体。

例 17-3 使用 GLX 窗口

```
#include <GL/glx.h>
#include <GL/gl.h>
#include <GL/glu.h>
#include <stdio.h>
#include <unistd.h>

#include <Inventor/SoDB.h>
#include <Inventor/actions/SoGLRenderAction.h>
#include <Inventor/nodes/SoCube.h>
#include <Inventor/nodes/SoDirectionalLight.h>
#include <Inventor/nodes/SoLightModel.h>
#include <Inventor/nodes/SoMaterial.h>
#include <Inventor/nodes/SoTransform.h>
#include <Inventor/nodes/SoSeparator.h>
#include <Inventor/nodes/SoSphere.h>

#define    WINWIDTH       400
#define    WINHEIGHT      400

float   floorObj[81][3];

// Build an Inventor scene with two objects and some light
void
buildScene(SoGroup *root)
{
```

```
   // Some light
   root->addChild(new SoLightModel);
   root->addChild(new SoDirectionalLight);

   // A red cube translated to the left and down
   SoTransform *myTrans = new SoTransform;
   myTrans->translation.setValue(-2.0, -2.0, 0.0);
   root->addChild(myTrans);

   SoMaterial *myMtl = new SoMaterial;
   myMtl->diffuseColor.setValue(1.0, 0.0, 0.0);
   root->addChild(myMtl);
   root->addChild(new SoCube);

   // A blue sphere translated right
   myTrans = new SoTransform;
   myTrans->translation.setValue(4.0, 0.0, 0.0);
   root->addChild(myTrans);

   myMtl = new SoMaterial;
   myMtl->diffuseColor.setValue(0.0, 0.0, 1.0);
   root->addChild(myMtl);

   root->addChild(new SoSphere);
}


// Build a floor that will be rendered using OpenGL.
void
buildFloor()
{
   int a = 0;

   for (float i = -5.0; i <= 5.0; i += 1.25) {
      for (float j = -5.0; j <= 5.0; j += 1.25, a++) {
         floorObj[a][0] = j;
         floorObj[a][1] = 0.0;
         floorObj[a][2] = i;
      }
   }
}

// Callback used by GLX window
```

```
static Bool
waitForNotify(Display *, XEvent *e, char *arg)
{
   return (e->type == MapNotify) &&
            (e->xmap.window == (Window) arg);
}

// Create and initialize GLX window.
void
openWindow(Display *&display, Window &window)
{
   XVisualInfo   *vi;
   Colormap cmap;
   XSetWindowAttributes swa;
   GLXContext cx;
   XEvent event;
   static int attributeList[] = {
            GLX_RGBA,
            GLX_RED_SIZE, 1,
            GLX_GREEN_SIZE, 1,
            GLX_BLUE_SIZE, 1,
            GLX_DEPTH_SIZE, 1,
            GLX_DOUBLEBUFFER,
            None,
   };

   display = XOpenDisplay(0);
   vi = glXChooseVisual(display,
            DefaultScreen(display), attributeList);
   cx = glXCreateContext(display, vi, 0, GL_TRUE);
   cmap = XCreateColormap(display,
            RootWindow(display, vi->screen),
            vi->visual, AllocNone);
   swa.colormap = cmap;
   swa.border_pixel = 0;
   swa.event_mask = StructureNotifyMask;
   window = XCreateWindow(display,
            RootWindow(display, vi->screen), 100, 100,
            WINWIDTH, WINHEIGHT, 0, vi->depth, InputOutput,
            vi->visual,
            (CWBorderPixel | CWColormap | CWEventMask), &swa);

   XMapWindow(display, window);
```

```
   XIfEvent(display, &event, waitForNotify, (char *) window);
   glXMakeCurrent(display, window, cx);
}


// Draw the lines that make up the floor, using OpenGL
void
drawFloor()
{
   int i;

   glBegin(GL_LINES);
   for (i=0; i<4; i++) {
      glVertex3fv(floorObj[i*18]);
      glVertex3fv(floorObj[(i*18)+8]);
      glVertex3fv(floorObj[(i*18)+17]);
      glVertex3fv(floorObj[(i*18)+9]);
   }

   glVertex3fv(floorObj[i*18]);
   glVertex3fv(floorObj[(i*18)+8]);
   glEnd();

   glBegin(GL_LINES);
   for (i=0; i<4; i++) {
      glVertex3fv(floorObj[i*2]);
      glVertex3fv(floorObj[(i*2)+72]);
      glVertex3fv(floorObj[(i*2)+73]);
      glVertex3fv(floorObj[(i*2)+1]);
   }
   glVertex3fv(floorObj[i*2]);
   glVertex3fv(floorObj[(i*2)+72]);
   glEnd();
}

main(int, char **)
{
   // Initialize Inventor
   SoDB::init();

   // Build a simple scene graph
   SoSeparator *root = new SoSeparator;
   root->ref();
```

```
    buildScene(root);

    // Build the floor geometry
    buildFloor();

    // Create and initialize window
    Display *display;
    Window window;
    openWindow(display, window);
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.8, 0.8, 0.8, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Set up the camera using OpenGL.
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(90.0, 1.0, 2.0, 12.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -5.0);

    // Render the floor using OpenGL
    glPushMatrix();
    glTranslatef(0.0, -3.0, 0.0);
    glColor3f(0.0, 0.7, 0.0);
    glLineWidth(2.0);
    glDisable(GL_LIGHTING);
    drawFloor();
    glEnable(GL_LIGHTING);
    glPopMatrix();

    // Render the scene
    SbViewportRegion myViewport(WINWIDTH, WINHEIGHT);
    SoGLRenderAction myRenderAction(myViewport);
    myRenderAction.apply(root);
    glXSwapBuffers(display, window);

    sleep (10);
    root->unref();
    return 0;
}
```

附录 A：C 程序员面向对象编程入门

Open Inventor 是一个用于开发 3D 程序的面向对象工具包。它虽然是用 C++开发的，但它同时还包含有 C 语言编程接口。这本书提到了许多像类、子类、和其它面向对象编程的概念。本书的所有例子都是用 C++编写的。如果在阅读本书之前，读者已经对“类”和面向对象的编程方法有了一定的了解，那么您将可以从本书中学到尽可能多的东西（You will get the most from this book if you have a reasonable understanding of classes and objected-oriented programming before you begin reading it）。

本附录向 C 程序员介绍了一些面向对象编程的入门知识，以及后面将要讨论的 C 语言接口概念的一些简介。如果读者已经完全熟悉了面向对象的编程方法，可以跳过本附录的第一节内容，不必理会附录中的例子代码。对于 Inventor C 语言编程接口的详细说明，请查看附录 B。

本章包括下列小节：

- “什么是面向对象的编程方法？”介绍两个面向对象编程的基础性概念：数据抽象和继承派生
- “一个类的例子：球”，通过开发一个真实的 C++类例子，举例说明面向对象编程的概念。
- “阅读建议”建议读者阅读更多有关 C++方面的知识。

什么是面向对象的编程方法？

许多成功的程序员在还没有知道面向对象技术之前就已经在使用这种技术了。我们可能听说过程序员将一块执行功能描述为“*模块*”。模块化的代码具有一个定义明确的接口，这个接口可以在不需要用户了解它是如何编写的情况下仍能被用户正常使用。模块化代码可以被许多程序重用，降低了程序员的学习时间，它允许修改模块内部的代码而不会影响到使用它的其它程序。它可以防止程序员纠缠于程序的细节。编程接口只定义功能，不关心具体的实现。

数据抽象

举例来说，我们分析一下标准C运行库提供的文件I/O函数：**create()**、**open()**、**read()**、**write()**、和**close()**。这些例程函数明确定义了文件的I/O功能，没有暴露任何文件系统的细节或执行代码。每个函数使用一个文件描述符来标识数据所代表的文件、设备、或网络Socket。对每种文件类型来说，代表这些对象的数据结构都是不同的，但是这些对程序员都是完全隐藏。open/close/read/write语法对于每种对象都是一样的。

这种隐藏内部数据结构的技术称作：*数据抽象* — 这是面向对象编程的第一个基础性概念。限制代码修改数据结构的做法是一种良好的编程习惯（programming practice）。如果暴露私有数据，允许程序员使用这些数据结构来修改他们不应该修改的东西，那么程序员编写的代码将会依赖内部系统代码的细节，这将导致系统的实现者不能对内部代码做出修改。

对象代表着一个由程序构造的功能模块（building blocks）。它是由数据结构和一些相关的用来操作这些数据结构的功能函数组成的。对象执行它们自己的函数，而不再允许应用程序使用它们的内部执行代码。在我们所举的例子中，C 文件 I/O 例程定义了一个通用文件（对象），这个通用文件只能通过 open/close/read/write 函数来使用。

继承

目前为止，我们已经讨论了表现良好的模块化代码，不过这不是面向对象编程思想的全部。继承是一种将面向对象代码与已经写好（well-written）的模块化代码相隔离的概念（*Inheritance* is the concept that sets object-oriented code apart from well-written modular code）。继承允许我们从已存在的对象上创建新的对象。这使得我们在程序中实现定制化和特例化变得非常容易。继承是面向对象编程的第二个基础性概念。

我们可能经常希望重新使用某些已经存在的代码，但在有些情况下却不能使用，因为我们需要还需要对这些代码做一些小的修改。于是我们不得不将修改后的代码拷贝到另外独立的执行模块中。这种重用方式是非常单调的，易错的，而且还会浪费我们的时间。继承为重用已存在的代码并做一些小修改的这种情况提供了一种简便机制。

C 文件 I/O 例程的例子事实上定义了三种对象类型：文件、设备和网络 sockets.这些对象是由通用文件对象创建的，它们定义了 open/close/read/write 函数的语法。每种文件对象的 I/O 例程是由具体实现其功能的代码来完成的（Writing the I/O routines is just a matter of implementing those functions for each type of file object）。功能具体实现的差别对程序员来说是隐藏的。

实现数据抽象和继承：类

面向对象编程语言使用的技术就是我们前面正式描述过的技术。C++在 C 语言基础上提供了一些扩展结构来加强这些技术。最基本的结构是*类*。我们既可以将类看作是一个带有相关函数的数据结构，也可以将其看作是一个带有某些数据的相关函数组。喜欢那种定义方式都没有关系。需要理解的最重要的概念是：对象将相关数据和函数压缩到一个单一包内，这个包称作*类*。

在类中的函数通常称作*成员函数*，或简称为*方法*。在类中的数据结构称作*成员变量*。所以，类是由成员函数和成员变量组成的。

注意，我们现在使用的术语：*“类”*，是代表着一个对象的抽象概念，这有些类似于在 C 语言中的“结构”数据类型。术语 *“对象”* 通常是指类的一个实例。当我们实例某个给定类时，也就是从这个类中创建出一个对象。C 语言也可以分配内存，产生一个结构的拷贝（The C parallel would be allocating memory to make a copy of a structure）。我们可以将结构的拷贝称作是那个结构的一个*实例*，或者称作是和结构有相同类型的一个*对象*。

当定义一些新类时，这些类可以从已经存在的类中派生出来。这些已存在的类称作*基类*或*父类*，新定义的类称作*派生类*或*子类*。按照这种方式创建的新类，通常都会继承定义在基

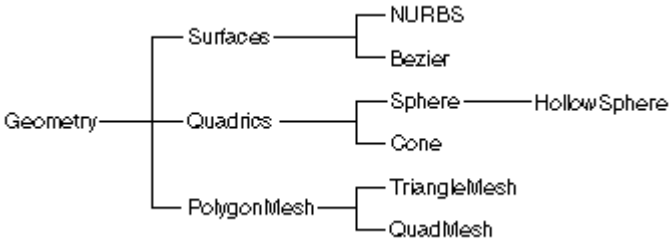
类中的所有方法和变量。

类的层次结构

Open Inventor 是由大量用于实现 3D 编程方面相关的类组成的。这些类是使用 C++来实现的。Open Inventor C 语言编程接口允许我们在 C 程序中使用这些类。因此我们可以不必需要先学习 C++语言，就可以在编写 C 程序的时候，获得 C++语言带来的好处。但是 C 编程接口并没有隐藏它本身就是类的事实。如果我们能够理解 Inventor 所包含的类，以及它们之间的相互关系，那么我们的编程任务也许会变得更容易些。例如，我们需要了解某个类是从那些类中派生出来的，这样就可以知道有那些函数可用使用了。

在面向对象系统中，类之间的相互关系经常使用*类层次结构图表*或*类树*来表示。图 A-1 就是一个例子。图 A-1 举例说明了一个假想的类层次结构。请注意，这个例子不是基于 Inventor 的。它只是以一种简单的方式来传达一些重要的概念。例子中的球体、锥体、和四边形网格等这些对象只是作为例子演示使用，和 Inventor 中带有相似名称的类不是同一回事。

图 A-1 简单的类层次结构图表



定义在类**Geometry**中函数和变量同样存在于每个子类中。即如果**Geometry**有一个叫做**Bbox**变量和一个叫做**getBbox()**函数，那么**Geometry**的所有子类也都有**Bbox**和**getBbox()**。

本书第 1 章中摘要列出了 Open Inventor 类树。

一个类的例子：球体

本节将讨论一个C++类的例子，以及这个类的所有成员函数。我们可以认为这个类就是图A-1 中虚构的类树中的一个类：**Sphere**，这个类代表和操作一个球体。**Sphere**类定义有若干个成员函数和成员变量：

```
class Sphere {
public:
            Sphere();                                    //
creates a sphere with default values
            ~Sphere();                                  //
destructor, which deletes a sphere
```

```
            void render()                                    //
renders the sphere
            Boolean pick(int x, int y); // picks the sphere

            float radius;
// radius of the sphere
            float center[3];
         // center of the sphere
};
```

Sphere是一个用来创建、管理、以及操作一个几何球体对象的类。球体的内部实现细节并没有向我们透漏。

注意，**Sphere**的成员函数并不带有一个球体参数。当我们在C++中调用这些函数时，我们会使用类本身作为参数来调用它们（When you invoke these functions from C++, you invoke them from the class itself.）。每个函数都有一个隐含的球体参数。例如，下面是如何创建一个球体，设置球体半径，然后在C++中渲染它。

```
Sphere *mySphere;                // pointer to a sphere object
mySphere = new Sphere();         // creates and initializes sphere
mySphere->radius = 3.0;          // sets the radius
mySphere->render();              // renders it
```

和C语言中访问结构成员数据一样，**sphere->**语法用来访问成员变量和调用成员函数。例如，**mySphere->render()**会对**mySphere**调用**render()**函数。**new Sphere()**语法是用来创建一个球体对象、为这个对象分配内存并初始化这个对象。

下面是与球体类相对应的 C 语言接口：

```
Sphere *SphereCreate();
void SphereDelete(Sphere *sphere);
void SphereRender(Sphere *sphere);
Boolean SpherePick(Sphere *sphere, int x, int y);
```

(这个例子遵循 Inventor C 函数命名规范。至于这些规范的详细知识，请参看附录 B)

C 语言接口还需要为球体定义一个结构变量。

```
struct Sphere {
            char   pad[48];  /* padding generated by Inventor */
            float  radius;
            float  center[3];
};
```

pad[48]是从C++代码中自动产生的。这些**pad**声明是C++类转成C语言接口所产生的副产品。它们主要用于保护那些程序员不应该访问的私有数据。

我们可以按照下面的代码来创建一个球体，设置球体半径，然后在 C 语言中渲染：

```
Sphere *mySphere;       /* my sphere object */

mySphere = SphereCreate();
mySphere->radius = 3.0;
SphereRender(mySphere);
```

请注意这些代码与 C++例子代码的相似之处。它们之间主要的差别只是语法。（请再次注意，这只是一个假设的例子；这不是在 Inventor 程序中设置球体半径的正确方式）

一个继承的例子：HollowSphere

回想一下，继承是一种可以从现有类中构建一个特殊类的能力。在 C++中，我们可以创建一个类的子类，除了一些我们可以选择的例外，子类和父类是相同的（In C++, you can create *subclasses* of a class, which are identical to the parent class with exceptions you can select.）。这些例外可能是与父类的函数实现上不同，或者仅仅只是额外增加了一些函数。子类可以被认为是从父类派生出来的，或从父类子类化来的（Subclasses are said to be *derived from* or *subclassed from* the parent class）。

例如，我们可以创建一个叫做**HollowSphere**的**Sphere**子类。**HollowSphere**和**Sphere**是相同的，除了**HollowSphere**带有一个厚度数值和一个用来指示是否需要透明渲染的新函数。**HollowSphere**是从**Sphere**派生出来的。因此它是**Sphere**的子类，**Sphere**的所有成员函数都适用于**HollowSphere**。我们的**HollowSphere**定义中并没有定义**delete()**、**render()**、或**pick()**函数。**HollowSphere**将从**Sphere**类中继承这些函数。这种方式同样适合**Sphere**'s的成员变量：**radius**和**center**：**HollowSphere**同样也会继承使用它们。

下面是**HollowSphere**的C++类定义：

```
class HollowSphere : public Sphere {  // subclass of Sphere
            void     showEquator();   // show equator during render
            float    thickness;       // stores thickness value
}
```

接下来的代码片段创建了一个空心球（hollow sphere），并设置球体的半径和厚度，打开 equator 选项，然后渲染它：

```
HollowSphere *mySphere;
mySphere = new HollowSphere();
```

```
mySphere->radius = 3.0;
mySphere->thickness = 0.25;
mySphere->showEquator();
mySphere->render();
```

使用 C 语言接口做同样的事情：

```
HollowSphere *mySphere;
mySphere = HollowSphereCreate();
mySphere->radius = 3.0;
mySphere->thickness = 0.25;
HollowSphereShowEquator(mySphere);
HollowSphereRender(mySphere);  /* inherited from parent class */
```

注意，当我们调用一个从父类继承来的方法时，这个方法的名字是使用子类的名字作为前缀的。关于 Inventor C 语言接口命名继承规则的全部说明请查看附录 B。

阅读建议

如果希望更深入地学习 C++和面向对象的编程方法，下面列出的书籍可以作为一个好的起点：

- Dewhurst, Stephen C., and Kathy T. Stark, *Programming in C++*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1989.
- Ellis, Margaret A., and Bjarne Stroustrup, *The Annotated C++ Reference Manual*. Reading, Mass.: Addison-Wesley, 1990.
- Lippman, Stanley B., *A C++ Primer*, 2e. Reading, Mass.: Addison-Wesley, 1991.
- Pohl, Ira, *C++ for C Programmers*. Redwood City, Ca.: The Benjamin/Cummings Publishing Co., 1989.
- Shapiro, Jonathan, *A C++ Toolkit*. Englewood Cliffs, N.J.: Prentice-Hall, Inc., 1991.
- Weiskamp, Keith, and Bryan Flamig, *The Complete C++ Primer*, 2e. San Diego, Ca.: Academic Press, 1992.

附录 B：C API 入门

如果读者是使用 C 语言进行开发工作的，那么就有必要阅读本章节！如果读者不熟悉面向对象的编程方法，请首先阅读附录 A，在附录 A 中介绍了一些 C++特性，以及一些 Open Inventor C 语言接口的基础性概念。

尽管 C 与 C++的语法是不同的，但在本书先前章节中所描述的概念仍然适用于 Open Inventor 的 C 实现库。如果读者已经读完本书的其它章节，那么读者将可以参照本章节，将本书的例子代码由 C++转换成 C 语言。如果读者已经购买了 Open Inventor 开发库，那么也可以将本书的 C++版本的例子编译成 C 版本的。请查看 Open Inventor 发行公告中如何使用这些在线例子程序，以及如何编译连接一个用 C 语言编写的 Inventor 程序。

在下面的章节中，将解释 Open Inventor 的 C 和 C++接口之间的不同之处。将会讨论下列主题：

- “命名 C 函数”
- “缩写 C 函数名称”
- “在 C 语言中创建和删除对象”
- “调用函数”
- “C 类和参考手册”
- “一个用 C 语言编写的简单的 Open Inventor 程序”

Open Inventor C语言编程接口是由一个自动转换程序产生的。C语言接口为每个C++类都定义了一个C结构数据，这个结构数据事实上是一个C++类的直接映射。它定义在类的C的头文件中。在C结构中，域变量不是隐藏的，就是公开的。公开域（成员变量）公布在每个类的参考手册页中。隐藏域（私有成员变量）被命名为**pad[]**。我们不应该修改隐藏域。

本章节的例子代码段是从本书的第 2 章中挑选出来的。

命名 C 函数

每个C函数的名字是由类名作为前缀，再加上C++成员函数名组成的。假设现在有一个叫做**SoGenericType**的类，这个类带有一个叫做**doOp()**的成员函数。这个函数的C语言名称应该是**SoGenericTypeDoOp()**。被上面的成员函数操作的对象作为这个C版本函数的第一个参数（The object to be operated upon by the member function appears as the first argument to the C version of the function）。我们上面虚构的函数的C语言原型应该是这样的：

```
SoGenericTypeDoOp(SoGenericType *);
```

再举一个更详细的例子，将一个节点增加到组中的函数，在 C++和 C 语言中应该是这样的：

```
C++:
root->addChild(material);
C:
SoGroupAddChild(root, (SoNode *) material);
```

C函数名字中带有一个前缀**SoGroup**。这是因为**addChild()** 是**SoGroup**的一个成员函数。C函数的第一个参数必须是一个指向**SoGroup**类型对象的指针。因为**addChild()**需要一个指向**SoNode**类型对象的指针作为它的第二个参数，所以我们需要将指针转换成**SoNode ***。

现在，我们假设对一个**SoGenericType**子类（像**SoGenericSubType**）调用**doOp()**函数。C子类以宏函数的形式继承父类的所有成员函数（The C subclasses inherit all the member functions of their parent classes in the form of macro functions）。这些宏只是简单地调用父类的函数，并将对象转换成父类的对象类型。这些继承下来的宏函数名称将用子类名称作为前缀，而不是使用父类名称。这样，**doOp()**的C函数名称应该是**SoGenericSubTypeDoOp()**。如果我们先将对象转换成**SoGenericType**的类型，那么我们也可以使用父类函数**SoGenericTypeDoOp()**。

在真正的Inventor例子中，我们也许更多的是向**SoGroup**的派生类增加子节点。例如，将节点增加到separator节点中的C 宏函数应该是：

```
SoSepAddChild(separator, (SoNode *) material);
```

缩写 C 函数名称

下面是另外一个使用 Open Inventor C 函数的例子：

```
C++:
viewer = new SoXtExaminerViewer(myWindow);
viewer->setSceneGraph(root);
C:
viewer = SoXtExamVwrCreateStd(myWindow, NULL);
SoXtExamVwrSetScene(viewer, (SoNode *) root);
```

C++函数**setSceneGraph()**是**SoXtRenderArea**类的一个成员函数，但是viewer是一个指向**SoXtExamVwr**类型的指针。所以，C宏函数的名字是以**SoXtExamVwr**作为前缀，并使用一个指向这个类型的指针作为第一个参数。

但是，如我们所见，这个C函数名称是被缩写了。这个函数在前面的例子中，如果它是未被缩写的，它应该是**SoXtExaminerViewerSetScene()**，这种名称太长了而且也很难使用。表B-1 列出了用于C函数名称的缩写词。只要C++函数中包含有位于表中的字符串，那么相应的C函数就可以使用第二列中缩写词。

表 B-1 C 函数名称缩写

C++ 函数名称	缩写
Action	Act
Attenuation	Atten
Binding	Bind
BoundingBox	BBox
Button	Btn
Callback	CB
Camera	Cam
Catalog	Cat
Character	Char
Color	Col
Component	Comp
Cylinder	Cyl
Decoration	Decor
Detail	Dtl
Direction	Dir
Directory	Dir
Editor	Ed
Environment	Env
Event	Ev
Examiner	Exam
FaceSet	FSet
Function	Func
Highlight	HL
Index	Ind (Indexed->Index)
Input	In
Integer	Int
Keyboard	Key
Light	Lgt
LineSet	LSet
Location	Loc
Locator	Loc

C++ 函数名称	缩写
Manager	Mgr
Manipulator	Mnp
Material	Mtl
Matrix	Mx
Normal	Norm
Orthographic`	Ortho
Output	Out
Perspective	Persp
Plane	Pln
Point	Pt
PointSet	PSet
Pointer	Ptr
Position	Pos
Primitive	Prim
Profile	Prof
Projector	Proj
QuadMesh	QMesh
RenderArea	RA
Rotate	Rot
Searching	Search
Section	Sect
Slider	Sldr
String	Str
Sphere	Sph
Texture	Tex
Tolerance	Tol
Transform	Xf
Translate	Xlate
TriangleStripSet	TSet
Vec	V(Vec3f-> V3f)
Vertex	Vtx
Viewer	Vwr
Visibility	Vis

C++ 函数名称	缩写
Wheel	Whl
Window	Win

在 C 语言中创建和删除对象

构造函数和析构函数的C语言版本是按照一种稍微不同于大多数成员函数的命名规范来命名的。我们假设的**SoGenericType**类型对象的C语言构造函数应该是**SoGenericTypeCreate()**。下面的例子演示了**SoSeparator**类的C++和C语言的构造函数：

```
C++:
SoSeparator *root = new SoSeparator();
C:
SoSep *root;
root = SoSepCreate();
```

析构函数以类似的方式命名。类型为**SoGenericType**的对象的析构函数应该是**SoGenericTypeDelete()**。例如，如果想删除一个viewer，我们可以这样调用**SoXtExamVwrDelete(viewer)**。当要删除节点和组的时候，我们不需调用析构函数，只要调用函数对节点和组取消引用就可以了。详见第 3 章中删除节点章节。

重载 C++方法

C++语言中允许一个类中有多个方法使用相同的名称，只要方法的参数列表不同就可以了。C语言没有提供此能力，所以对于每个C++中方法名称相同的函数在C语言中必须是名称不同的函数。下面是一个**SoAction**类的例子：

C++

```
void apply(soNode * node);
void apply(SoPath *path);
```

C：

```
void SoActApply(SoAction * act, SoNode * node);
void SoActApplyPath(SoAction * act, SoPath * path);
```

调用函数

我们应该知道，以C++方式调用函数与以C方式调用是有一些不同的。Open Inventor的

C++版本的方法对有些参数使用缺省数值。如果我们没有指定这些参数，C++会简单地使用缺省值来调用这些函数。C语言不会为我们自动填写这些缺省值,所以我们必须为每个参数都提供数值。下面是一个C++调用创建一个**appWindow**对象的例子，以及相对应的C调用：

```
C++
Widget appWindow = SoXt::init(argv[0]);
C:
widget appWindow;
appwindow = SoXtInit(argv[0], "Inventor");
```

C 类和参考手册

有一个针对每个 Inventor 类的在线 C 参考手册。对于一个类的详细信息，可以查看它的参考手册，在手册中列出了所有可用的函数和数据结构。在“继承于（Inherits From）”节中将告诉我们当前类是从那个类继承来的。许多类的继承层次结构可能有很多层。如果我们希望了解一个给定类的所有可用函数，我们必须首先要了解父类的可用函数。它们是以参考手册大纲的一部分列在参考手册中的。每个类都带有一个以这个类命名的单独 Include 文件。

参考手册中还列有函数参数的缺省值，如下所示：

```
void setSomeValue(x, y, z = 1.0)
```

z = 1.0表示 1.0 是参数z的建议缺省值。

参考手册同时使用 C 类名称的缩写的与未缩写的版本。我们的程序中也可以同时使用这两种版本。

一个用 C 语言编写的简单的 Open Inventor 程序

现在我们应该已经充分了解如何将例 2-4 代码以 C 语言的方式来编写。下面是例子 2-4 的 C 语言代码：

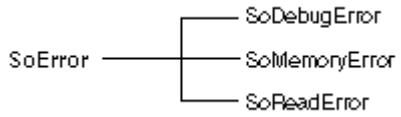
```
main(int argc, char **argv)
{
   Widget      myWindow;
   SoSep       *root;
   SoMtl       *myMaterial;
   SoCone      *myCone;
   SoXtExamVwr *myViewer;

   myWindow = SoXtInit(argv[0], "Inventor");
```

```
    if (myWindow == NULL) exit(1);

    root = SoSepCreate();
    SoSepRef(root);
    myMaterial = SoMtlCreate();
    myCone     = SoConeCreate();
    SoMColSetR_G_B(&(myMaterial->diffuseColor), 1.0, 0.0, 0.0);
    SoSepAddChild(root, (SoNode *)myMaterial);
    SoSepAddChild(root, (SoNode *)myCone);

    /* Set up viewer: */
    myViewer = SoXtExamVwrCreateStd(myWindow, NULL);
    SoXtExamVwrSetScene(myViewer, (SoNode *)root);
    SoXtExamVwrSetTitle(myViewer, "Examiner Viewer");
    SoXtExamVwrShow(myViewer);
    SoXtShow(myWindow);
    SoXtMainLoop();
}
```

附录 C：错误处理

Inventor 的调试版本和发行版本都提供了基本的错误处理机制。如图 C-1 所示，有三个类是从基类 SoError 派生出来的。在读 Inventor 格式文件时有可能发生*读* 错误。如果应用程序运行时缺少内存资源，就有可能发生*内存访问* 错误。*调试* 错误是由一个编程错误而产生。绝大部分的调试错误只在 Inventor 的调试版本中产生，发行版本一般不会做检查。一个运行正确的应用程序不会产生任何调试错误。在开发工作的初期，使用 Inventor 的调试版本是一个好主意，这样可以保证我们的程序运行是正确的。之后，当不再有错误产生时，我们就可以将程序切换成发行版本。

图 C-1 **SoError** 类树



例如，假设 Inventor 在读取一个文件的过程中，遇到了一个未知的域名。那么将发生下列事情：

1. Inventor调用静态方法**SoReadError::post()**，传入一个正确的参数。
2. **SoReadError**类创建了自身的一个实例。在这个实例中，保存有调试字符串，这个字符串用来详细描述错误消息（使用**getDebugString()**可以获得调试字符串）。
3. **SoReadError**类将这个错误实例传递到它的错误处理器中。所有错误类的缺省错误处理器只是简单地将调试字符串输出到**stderr**设备上。如需要对错误进行特殊处理，我们也可以注册自己错误处理器。例如，以上面的情况来说，我们也许希望弹出一个窗口，显示一个消息，通知用户在读文件的过程中发生了一个错误。使用方法**setHandlerCallback()**可以为一个错误类注册我们自己的错误处理器。

运行时刻类型检查

标准 Inventor 错误类的运行时刻类型检查机制是由基类 SoError 提供的。每个错误类都包含有一个运行时刻类的类型 ID，此 ID 可被用来确定实例的类型。

发送错误

每个**SoError**的子类都有自己的**post()**方法。发送主要是由Inventor来执行的，不过程序开发人员也可以发送他们自己的错误。

处理错误

我们可以通过指定一个回调函数来覆盖掉缺省的错误处理器，这样当发生错误时就会调用此回调函数。每个**SoError**的子类都支持一个静态方法**setHandlerCallback()**。我们可以为**SoError**注册一个回调函数，这样一来，所有的错误都将会调用我们注册的回调函数。我们也可为**SoError**派生出来的某个错误类注册一个回调函数。为某个特定错误实例的派生类而注册的回调函数将被用来处理这个错误（or you can register it on one of the classes derived from **SoError**. The callback for the most derived class of a specific error instance is used to handle the error）。注意，传递到回调函数中的错误实例在调用完回调函数后会立即被删除。应用程序如需要在以后使用错误信息的话，应该提前拷贝存储这些数据。

调试

如果我们使用的是一个交互式的调试器（**例如，微软公司推出的Visual C++集成开发环境。译者注**），那么我们可以在**SoError::handleError()**中设置一个断点，只要一个错误需要处理时，都会调用**SoError::handleError()**。我们也可以创建自己的错误处理器，然后注册它，通过使用回调函数来研究错误发生的条件。