

VRML 虚拟现实应用技术

张德丰 周 灵 编 著

電子工業出版社

Publishing House of Electronics Industry

北京 • BEIJING

内 容 简 介

本书系统地介绍了虚拟现实建模语言（VRML）的基础知识，各节点详细的使用方法和应用，并给出了大量实例。全书共分为 8 章，分别介绍了虚拟现实概述，虚拟现实软件及三维立体造型，造型的其他相关操作，虚拟现实环境设计，动画效果与交互节点，虚拟现实的高级应用，虚拟现实与 MATLAB 接口应用，VRML 脚本语言与编程等内容。

本书可作为大学本科和高职高专计算机相关专业的虚拟现实技术课程教材，也可供成人教育和在职人员的培训使用。同时适合于欲制作虚拟空间的 VRML 初学者，也可作为运用 VRML 技术进行系统仿真、虚拟现实程序设计的研究人员和程序开发人员的参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

VRML 虚拟现实应用技术 / 张德丰，周灵编著. —北京：电子工业出版社，2010.8

ISBN 978-7-121-11679-7

I. ①V… II. ①张… ②周… III. ①VRML 语言—程序设计 IV. ①TP312

中国版本图书馆 CIP 数据核字（2010）第 164396 号

责任编辑：陈韦凯

特约编辑：李玉昌

印 刷：

装 订：

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编 100036

开 本：787×1 092 1/16 印张：23.75 字数：608 千字

印 次：2010 年 8 月第 1 次印刷

印 数：4 000 册 定价：45.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888。

质量投诉请发邮件至 zltz@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

前 言

计算机技术发展迅猛，已由初级计算、实时控制、事务处理迅速朝着巨型化、微型化、网络化、智能化及多媒体化等方面发展，计算机将人类社会带入了崭新的信息时代。尤其是计算机网络的飞速发展，使我们的地球变成了一个“地球村”。早期的网络系统主要传送文字、数字等信息，随着多媒体技术在网络上的应用，使得目前的计算机网络无法承受如此巨大的信息量。为此，人们开发出信息高速公路，即宽带网络系统，而在信息高速公路上驰骋的“高速跑车”就是虚拟现实建模语言（Virtual Reality Modeling Language, VRML）系统。

VRML 是一门最近才兴起的新型的三维造型语言，它可以通过创建一个虚拟场景达到现实中的效果。VRML 支持三维动画，其实时交互功能大大改变了原来互联网上单调、交互性较差的弱点，从而创建一个全新的可进入、可参与的三维立体虚拟现实世界。短短数年，VRML 已经应用到很多领域，如航天、军事、建筑、医疗、教育……人们相信它会有广阔的发展前景。由于 VRML 本身自带脚本语言，而且也有 Java、JavaScript 的支持，使三维造型的控制交互比其他同类造型语言具有更强的优势。用此语言设计的造型和动画文件小、交互性强、控制灵活，适合嵌入网页在网上播放。此语言易学，有系列开发工具支持，有自身的特色。

VRML 提供了三维（3D）应用中大多数常见功能，也提供了足够的创造空间。

（1）建模能力强。VRML 定义了类型丰富的几何、编组、定位等节点，建模能力较强。

（2）有真实感及渲染能力。通过提供丰富的渲染相关节点，可以很精细地实现光照、着色、纹理贴图、三维立体声源。

（3）观察及交互手段，传感器类型丰富。可以感知用户交互，视点可以控制对三维世界的观察方式。

（4）动画可控制。VRML 提供了方便的动画控制方式。

HTML 和 VRML 的差别与建筑物的蓝本和它的模型的差别是同一个道理。VRML 是 Internet 上基于万维网的虚拟现实建模语言，用来描述三维物体及其运动行为，构建虚拟世界。它的基本特征包括三维性、交互式、分布式集成性和逼真性等。随着国际互联网的普及，网络技术和硬件设施飞速发展，虚拟现实技术将成为今后网络多媒体发展方向的主流。总之，VRML 将创建一种融多媒体、三维图形、网络通信、虚拟现实为一体的新型媒体，具有先进性和普及性。

全书共分为 8 章：第 1 章介绍虚拟现实概述，包括虚拟现实技术的发展史、虚拟现实的特点、虚拟软件技术等内容；第 2 章介绍虚拟现实软件及三维立体造型，包括软件开发模型、几何造型、造型外观设计等内容；第 3 章介绍造型的其他相关操作，包括造型空间变换、造型群节点、其他组节点使用等内容；第 4 章介绍虚拟现实环境设计，包括背景设计、光源创建、阴影效果创建等内容；第 5 章介绍动画效果与交互节点，包括路由和事件、动画效果、交互造型功能等内容；第 6 章介绍虚拟现实的高级应用，包括虚拟现实在三维立体场景中的设计、虚拟实现自然景观设计等内容；第 7 章介绍虚拟现实与 MATLAB 接口应用，包括 MATLAB 的简单介绍、虚拟实现工具箱的 MATLAB 函数、MATLAB 接口中虚拟现实的应用等内容；第 8 章介绍 VRML 脚本语言与编程，包括 Script 脚本、对象处理等内容。

本书旨在帮助读者通过学习掌握虚拟现实三维立体网络程序的开发和应用技能，了解在计算机软件开发方面如何利用目前国际上最先进的开发工具，以及如何运用软件工程的思想来开发和设计三维立体网络程序。

为便于学习，本书免费提供书中示例程序的源代码，读者可通过登录华信教育资源网（www.hxedu.com.cn）查找本书下载，所有源程序都在计算机上经过严格的调试并通过。

本书中作者的研究成果得到了广东省科技计划项目（项目编号：2009B010800053）的大力支持，正是这些支持为本书的形成奠定了坚实的基础，在此表示衷心的感谢。

本书由张德丰、周灵、周燕和马莉编写。参与图书编写及源程序校对、调试等工作的还有雷小平、崔如春、李娅、栾颖、刘志为和周品等。

由于编著者水平有限，加之时间仓促，书中难免会有错误和不足之处，恳请广大读者给予批评和指正。

编著者
2010年5月

目 录

第 1 章 虚拟现实概述	(1)
1.1 人机交互技术的发展史	(1)
1.2 VR 技术的发展史	(3)
1.3 虚拟现实的特点	(5)
1.3.1 交互性	(5)
1.3.2 沉浸感	(6)
1.3.3 构想性	(6)
1.4 虚拟现实的发展前景	(7)
1.5 虚拟现实的系统分类	(7)
1.5.1 沉浸式虚拟现实系统	(8)
1.5.2 桌面式虚拟现实系统	(9)
1.5.3 增强式虚拟现实系统	(10)
1.5.4 分布式虚拟现实系统	(11)
1.6 虚拟软件技术	(12)
1.6.1 VRML 的基本概念	(12)
1.6.2 VRML 的立体空间计量单位	(13)
1.6.3 VRML 语法	(14)
1.6.4 VRML 的节点和域	(17)
1.7 虚拟现实主要应用领域	(21)
1.7.1 城市规划	(21)
1.7.2 室内设计	(22)
1.7.3 文物保护	(22)
1.7.4 军事与航空航天	(22)
1.7.5 虚拟教育与培训	(25)
1.7.6 商业领域	(28)
1.7.7 娱乐	(28)
1.7.8 工业应用	(29)
1.7.9 医学领域	(30)
1.7.10 虚拟现实在 Web3D/产品/静物展示中的应用	(31)
第 2 章 虚拟现实软件及三维立体造型	(33)
2.1 软件开发模型	(33)
2.1.1 软件瀑布开发模型	(33)
2.1.2 软件原型开发模型	(34)
2.1.3 软件渐进式开发模型	(35)
2.2 虚拟现实软件开发工具	(39)
2.3 几何造型	(41)
2.3.1 造型节点	(41)

2.3.2	简单几何造型节点	(42)
2.3.3	复杂几何造型节点	(47)
2.4	造型外观设计	(68)
2.4.1	外观节点	(69)
2.4.2	材质节点	(70)
2.4.3	图片型的表面材质节点	(85)
2.4.4	表面材质转换节点	(87)
2.4.5	影像纹理节点	(89)
2.4.6	纹理坐标节点	(92)
2.4.7	造型的材质设计	(94)
第3章	造型的其他相关操作	(100)
3.1	造型空间变换	(100)
3.1.1	坐标变换节点	(100)
3.1.2	空间坐标的平移	(101)
3.1.3	空间坐标的旋转	(102)
3.1.4	空间坐标的缩放	(106)
3.2	造型群节点	(110)
3.2.1	编组节点	(110)
3.2.2	素材调用基本方法	(113)
3.2.3	节点的定义及引用	(113)
3.2.4	内联节点	(120)
3.3	其他组节点使用	(123)
3.3.1	布告牌	(123)
3.3.2	开关节点	(125)
3.3.3	细节层次节点	(127)
3.3.4	视点	(131)
3.3.5	锚节点	(134)
3.3.6	导航节点	(136)
第4章	虚拟现实环境设计	(139)
4.1	背景设计	(139)
4.2	光源创建	(145)
4.2.1	法线	(145)
4.2.2	点光源	(147)
4.2.3	平行光源	(150)
4.2.4	锥光源	(153)
4.3	创建阴影效果	(158)
4.4	创建雾化效果	(161)
4.5	创建声音效果	(164)
4.5.1	环境中声音的基本概念	(164)
4.5.2	音频剪辑节点	(165)
4.5.3	声音节点	(166)

第 5 章	动画效果与交互节点	(170)
5.1	事件和路由	(170)
5.1.1	事件	(170)
5.1.2	路由	(170)
5.2	动画效果	(172)
5.2.1	时间传感器节点	(172)
5.2.2	位置插补器	(173)
5.2.3	颜色插补器	(178)
5.2.4	朝向插补器	(181)
5.2.5	标量插补器	(186)
5.2.6	坐标插补器	(190)
5.2.7	法向量插补器	(192)
5.3	交互造型功能	(196)
5.3.1	交互的基本概念	(196)
5.3.2	触摸传感器	(196)
5.3.3	平面传感器	(199)
5.3.4	圆柱体传感器	(204)
5.3.5	球体传感器	(211)
5.3.6	接近传感器	(213)
5.3.7	可视传感器	(219)
5.3.8	碰撞传感器	(222)
第 6 章	虚拟现实的高级应用	(226)
6.1	虚拟现实在三维立体场景中的设计	(226)
6.1.1	虚拟现实在生日场景中的设计	(226)
6.1.2	虚拟现实在奥运五环场景中的设计	(228)
6.1.3	虚拟现实月亮绕地球转动场景设计	(232)
6.2	虚拟实现自然景观设计	(236)
6.2.1	虚拟现实雪山设计	(236)
6.2.2	虚拟现实海上日出设计	(243)
6.2.3	虚拟现实雪山树林设计	(248)
6.3	虚拟现实建筑设计	(252)
6.3.1	虚拟现实办公楼	(252)
6.3.2	虚拟现实医院设计	(257)
6.3.3	虚拟现实公路设计	(262)
6.3.4	虚拟现实客厅设计	(271)
6.4	虚拟三维人体骨骼设计	(282)
第 7 章	虚拟现实与 MATLAB 接口应用	(293)
7.1	MATLAB 的简单介绍	(293)
7.1.1	MATLAB 的概述	(293)
7.1.2	MATLAB 的启动、退出及工作界面	(293)
7.1.3	Simulink	(294)

7.2	虚拟实现工具箱的 MATLAB 函数	(296)
7.2.1	MATLAB 的接口函数	(296)
7.2.2	Vrworld 对象	(301)
7.2.3	Vrnode 对象	(308)
7.3	Simulink 的应用接口	(311)
7.3.1	使用 Simulink 连接虚拟世界	(311)
7.3.2	一个虚拟世界例子	(315)
7.4	MATLAB 接口中虚拟现实的应用	(317)
7.4.1	热传送的虚拟现实	(317)
7.4.2	汽车在山中运动的模拟	(322)
7.5	Simulink 接口虚拟现实示例	(332)
7.5.1	灯光的模拟	(332)
7.5.2	磁悬浮模型	(340)
第 8 章	VRML 脚本语言与编程	(351)
8.1	脚本	(351)
8.2	VRMLScript 语言	(355)
8.2.1	表达式	(355)
8.2.2	语法	(356)
8.2.3	函数	(360)
8.3	对象处理	(360)
8.3.1	对象	(360)
8.3.2	数学运算对象	(361)
8.3.3	Browser 对象	(361)
8.4	VRML 与网络	(370)
参考文献		(372)

第 1 章 虚拟现实概述

虚拟现实（Virtual Reality, VR）技术，是 20 世纪末才兴起的一门崭新的综合性信息技术。它融合了图像处理、计算机图形学、人工智能、多媒体技术、传感器、网络以及并行处理技术等多个信息技术分支的最新发展成果，为创建和体验虚拟世界提供了有力的支持，从而大大推进了计算机技术的发展。VR 技术的特点在于，由计算机产生一种人为虚拟的环境，这种虚拟的环境是通过计算机构成的三维空间，或是把其他现实环境编制到计算机中去产生逼真的“虚拟环境”，从而使得用户在多种感官上产生一种沉浸于虚拟环境的感觉。

VR 技术实时的三维空间的表现能力、人机交互式的操作环境以及给人带来的身临其境感受，将一改人与计算机之间枯燥、生硬和被动的现状。它不但为人机交互界面开创了新的研究领域，为智能工程的应用提供了新的界面工具，为各类工程大规模的数据可视化提供了新的描述方法，同时，它还能为人们探索宏观世界和微观世界以及由于种种原因不便于直接观察的事物的运动变化规律，提供极大的便利。

VR 技术一经问世，人们就对它身临其境的真实感和超越现实的虚拟性的追求，以及建立的个人能够沉浸其中、进出自如并具有交互作用的虚拟世界，产生了浓厚的兴趣。近几年，VR 技术不但已开始军事、医学、设计、房地产、考古、艺术、娱乐等诸多领域得到越来越广泛的应用，而且还给社会带来了巨大的经济效益。因此，有关人士认为：20 世纪 80 年代是个人计算机的时代；90 年代是网络、多媒体的时代；而 21 世纪初则将是 VR 技术的时代。

1.1 人机交互技术的发展史

目前，计算机已经成为现代科学技术的支柱，而计算机的使用却离不开人与计算机之间的交互。因此，计算机的发展史不仅是计算机本身处理速度和存储容量飞速提高的历史，而且也是人机交互界面（Human-Computer Interface, HCI）不断改进的历史。它的重要性就在于它极大地影响了最终用户的使用，影响了计算机的推广应用，甚至影响了人们的工作和生活。

从计算机诞生至今，人机交互界面经历了以下几个发展阶段。

1. 命令行界面（Command Line Interface, CLI）

作为第一代人机交互界面，交互终端使用了文本编辑程序，可以把各种输入/输出信息直接显示在屏幕上，并通过问答式对话、文本菜单或命令语言等方式进行人机交互。但在这种界面中，人只能使用手这一种交互通道，通过键盘输入信息，输出也只能是静态的单一字符，同时界面和应用还没有分开。因此，这一时期的人机交互界面的自然性和效率性都很差。

2. 图形用户界面（Graphical User Interface, GUI）

20 世纪 80 年代初，GUI 的广泛流行将人机交互推向图形用户界面的新阶段。人们不再需要死记硬背大量的命令，而可以通过窗口（Windows）、图标（Icon）、菜单（Menu）、指点装置（Pointing Device）直接对屏幕上的对象进行操作，即形成了 WIMP 的第二代人机界面。与命令行界面相比，GUI 采用视、点（鼠标）代替了记、击（键盘），使得人机交互的自然性和效率性都有较大的提高，从而极大地方便了非专门用户的使用。

3. 多媒体界面（Multimedia Interface）

目前流行的多媒体界面可以看作是 WIMP 界面的另一种风格，它在界面信息的表现方式上进行了改进，采用了多种媒体。同时界面输出也开始转为静态/动态、二维图形/图像及其他多媒体信息的方式，从而有效地提高了计算机到用户的通信带宽。

图形交互技术的飞速发展充分说明了对于应用来说，使处理的数据易于操作并直观是十分重要的问题。人们的生活空间是三维的，虽然 GUI 已提供了一些仿三维按钮等界面元素，但界面仍难进行三维操作；另一方面，人们习惯于日常生活中的人与人、人与环境之间的交互方式，其特点是形象、直观、自然，人通过多种感官来接收信息，如可见、可听、可说、可摸、可拿等，而且这种交互方式是人类所共有的，对于时间和地点的变化是相对不变的。但无论是命令行界面，还是图形用户界面，都不具有以上所述的进行自然、直接、三维操作的交互能力。因为在实质上它们都属于一种静态、单通道的人机界面，而用户只能使用精确的信息在一维和二维空间中完成人机交互。因此，更加自然的交互方式将逐渐为人们所重视，并成为今后人机交互界面的发展趋向。为适应目前和未来的计算机系统要求，人机交互界面应能支持时变媒体（Time-Varing Media）实现三维、非精确及隐含的人机交互，而 VR 技术正是实现这一目的的重要途径，它为建立起方便、自然、直观的人与计算机的交互方式创造了极好的条件。

超脱不同的应用背景看，VR 技术是把抽象、复杂的计算机数据空间表示为直观的、用户熟悉的事物，它的技术实质在于提供了一种高级的人与计算机交互的接口，使用户与计算机产生数据空间进行直观的、感性的、自然的交互。它是多媒体技术发展的更高境界（图 1-1）。

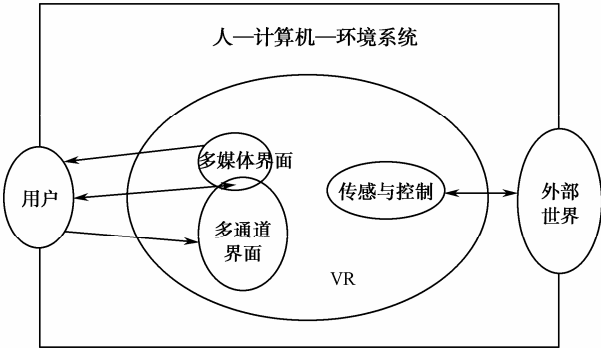


图 1-1 VR 与多通道-多媒体用户界面的关系图

作为新一代的人机交互系统，VR 技术与传统交互技术的区别可以从下列几方面说明。

1) 自然交互

人们研究“虚拟现实”的初衷就是“计算机应该适应人，而不是人适应计算机”，认为人机接口的改进应该基于相对不变的人类特性。在 VR 技术中，人机交互可以不再借助键盘、鼠标、菜单，而是使用头盔、手套甚至向“无障碍”的方向发展，从而使最终的计算机能对人体有感觉，聆听人的声音，通过人的所有感官传递反应。

2) 多通道（Multimodality）

多通道界面旨在充分利用一个以上的感觉和运动通道的互补特性来捕捉用户的意向，从而增进人机交互中的自然性。现在，计算机操作时，人的眼和手十分累，效率也不高。VR 技术可以将听、说和手、眼等协同动作，实现高效人机通信，还可以由人或机器选择最佳反应通道，从而不会使某一通道负担过重。

3) 高“带宽”

目前,计算机输出的内容已经可以快速、连续地显示彩色图像,其信息量非常大。而人们的输入却还是使用键盘一个又一个地敲击,VR技术则可以利用语音、图像及姿势等的输入和理解进行快速大批量地信息输入。

4) 非精确

精确交互技术是指能用一种技术来完全说明用户交互目的的交互方式,键盘和鼠标器均需要用户的精确输入。但是,人们的动作或思想往往并不很精确,而计算机应该理解人的要求,甚至于纠正人的错误,因此,虚拟现实系统中智能化的界面将是一个重要的发展方向。

5) 通过交互作用表示事物的现实性

传统的计算机应用方式中,人机交互的媒介是将真实事物用符号表示,是对现实的抽象替代,而VR技术则可以使这种媒介成为真实事物的复现、模拟甚至想象和虚构。它能使用户感到并非是在使用计算机,而是在直接与应用对象打交道。

1.2 VR技术的发展史

虚拟现实的渊源可以追溯到40多年前。1962年,美国专利号#3,050,870发布了Morton Heilig的一项题为“全传感仿真器”,它是第一个虚拟现实视频设备。在Heilig完成其发明的时候,没有人能意识到它所代表的技术发展方向。从政府、RCA、MGM到好莱坞,到私营企业,没有人想对他的发明投资。你能想象当今大家都想在这一技术上花钱时,他的感受如何。

Heilig“全传感仿真器”的主要组成部分有三维视频(由一对并排的35mm摄像机拍摄)、运动、颜色、立体声音、香气、风和一个可以振动的座位。它可以仿真骑车穿越纽约市的过程,“骑车人”能感受到风,感受到路面的颠簸,当经过饭店时“骑车人”甚至闻到食物的香味。

Heilig也设计了一个头戴式电视。这一设计被Ivan Sutherland继承和发展。1965年,Ivan Sutherland提出“The Ultimate Display”的概念。1966年,Sutherland在使用者眼睛前绑上两个CRT显示器,让使用者能看到立体的图像。大约40年后,现代的HMD(Head-Mounted Display,头盔式显示器)使用缩小的两个CRT或液晶显示器(LCD)安装在头部,原理上与Sutherland没有很大的差别,只是那时的CRT显示器比今天的要重得多,所以,Sutherland使用了一副机械臂来负担显示器的重量。这一机械臂还具备检测用户视角的功能。大多数今天的HMD使用非接触式位置跟踪器(如磁跟踪器或超声波跟踪器),但这项技术在20世纪60年代还没有被发明。

Sutherland与Heilig相比最大的进步在于,Sutherland意识到可以用计算机生成HMD中的场景,代替摄像机拍摄的模拟图像,并开始设计这样的“场景生成器”。这便是图形加速器的先驱。早期的图形场景生成器大约于1973年由Evans和Sutherland研制成功,但在1/20s内只能计算与显示200~400个简单的多边形。

这方面的研究引起了美国军方的兴趣。由于飞机模拟器可能花费数百万美元,而且是为特定的飞机型号设计的,当那个飞机型号过时后,该飞机的仿真器也便被淘汰。如果仿真器能做出通用平台上,飞机型号的改变可能只需要进行少量软件上的更新。看来,使用新型技术的优点是明显的。20世纪70年代和80年代早期,美国军方投入大量的经费开展了大量有关“飞行头盔”和军用现代仿真器的研究,但这方面的研究成果大部分被定为机密,未能正式出版。随着国防资金的削减,这方面的研究资助大为减少,一些研究开始由军用转为民用,促使了

VR 技术在更广泛的领域发展。

NASA 是对现代仿真器感兴趣的另一个美国政府部分，他们需要仿真器帮助训练宇航员。由于实际生成一个与外层空间或遥远星球一致的环境很难或根本不可能，所以在 1981 年，在一笔很小经费的支撑下，他们生成了一个基于 LCD 的 HMD 原型，并命名为虚拟显示环境显示器（VIVED）。NASA 科学家把 Sony 公司生产的产品“Watchman”TV 进行简单的改装，在 LCD 前安装了特殊的光学镜头，用于把图像聚焦到眼睛相近的位置。NASA 科学家集成了 DEC PDP 11-40 主计算机、由 Evans 和 Sutherland 设计的 Picture System 2 图形计算机和 Polhemus 非接触类跟踪器（图 1-2），研制出第一个虚拟现实系统。跟踪器被用于测量用户的头部运动，并把测量数据传输给 PDP。PDP 主计算机再把这些数据编排后传给图形计算机，由图形计算机计算出新的图像，并立体显示在 VIVED 上。



图 1-2 Polhemus 非接触类跟踪器

1985 年后，Scott Fisher 加入该项目，他把新型的传感手套集成到仿真器中。手套最先是 由 Thomas Zimmerman 和 Jaron Lanier 作为一个非程序员使用的虚拟编程接口开发的。到 1988 年，Fisher 和 Elizabeth Wenzel 生成了第一个能操纵 4 个三维虚拟声源的硬件系统。它们能生成定位到任意空间位置的声源。这给仿真器提供了重要的扩展，原始的 VIVED 系统演变为 VIEW（Virtual Interface Environment Workstation）。以前的软件被移植到 HP9000 型功能更强的计算机上。

进入 20 世纪 90 年代，虚拟现实研究热潮转移到民间的高科企业。研究出第一套数据手套的 Jaron Lanier 成为销售虚拟现实产品的第一家商业公司 VPL 公司的总裁，这家公司卖的第一套传感手套叫“DataGloves”，第一套 HMD 叫“EyePhone”。

在其他方面，标准化的虚拟现实建模语言（Virtual Reality Modeling Language, VRML）在 Internet 上构建可共享、可互换的一个虚拟环境奠定了很好的基础，VRML 本身也由早期的 VRML1.0 发展到 VRML2.0，再发展到 VRML97，现在还在进一步发展中。分布式交互仿真标准化工作也由早期的 SIMNET 发展到分布式交互仿真 DIS，再发展到如今的高层体系结构 HLA。

虚拟现实系统已由单机系统发展到分布式虚拟现实系统，现在人们研究的比较热门的是支持协同工作的分布式虚拟现实系统，即协同虚拟现实系统。即由过去只支持多人共享，发展到需要支持多人相互感知、协同操纵等为目的的协同感知。

我们知道虚拟现实引擎是虚拟现实系统中最为关键的部件之一。过去负责实时渲染虚拟环

境的虚拟现实引擎主要是基于计算机图形,即首先对真实世界进行抽象,从而建立起三维几何模型,一般用多边形表示。在给定观察和观察方向以后,利用计算机由模型实现多边形绘制、着色、消隐、光照以及投影等一系列绘制过程,产生虚拟环境。这种基于图形渲染技术所面临的主要问题是:一方面需要高性能的图形工作站,系统成本非常昂贵;另一方面需要三维建模,工作非常烦琐,而且往往需要专业人员,这部分的费用也相当高。为此,基于图像渲染技术被提出,并在近几年引起了人们的广泛关注。基于图像渲染与基于图形渲染不同,它是直接由照片图像来构造虚拟环境。这方面的研究起步较晚,还有大量的工作要做,但对虚拟现实系统的大面积推广应用起到较大的作用。

虚拟现实开发工具发展也迅速,1992年,英国 Sense8 公司开发出“WorldToolKit”,该软件包提供了一系列函数用于支持从更高的层次上开发虚拟实现应用,而且为用户屏蔽底层硬件上的差异,使用户只要分析与实现虚拟现实应用本身,而不需要考虑不同机型、不同操作系统、不同接口的硬件差异。WorldToolKit 使开发虚拟现实应用变得更加科学化,测试时间也大大减少。随后,类似的软件相继推出,比较著名的有 Vega、MR、dVS 等。

1.3 虚拟现实的特点

从本质上说,虚拟现实系统就是一种先进的计算机用户接口,它通过给用户同时提供诸如视、听、触等各种直观而又自然的实时感知交互手段、最大限度地方使用户的操作,从而减轻用户的负担、提高整个系统的工作效率。美国科学家 Burdea G 和 Philippe Coiffet 曾在 1993 年世界电子年会上发表的“Virtual Reality Systems and Applications”一文中,提出一个“VR 技术的三角形”,它简明地表示了虚拟现实具有的 3 个最突出的特征:交互性(Interactivity)、沉浸感(Immersion)和构想性(Imagination),也就是人们熟称的 3 个“I”特性(图 1-3)。

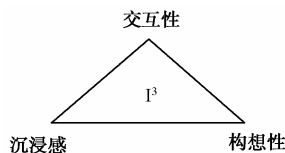


图 1-3 3 个“I”特性

下面分别对 3 个特性介绍如下。

1.3.1 交互性

交互性是指参与者对虚拟环境中物体的可操作程度和从环境中得到反馈的自然程度(包括实时性)。这种交互的产生,主要借助于各种专用的三维交互设备(如头盔显示器、数据手套等),它们使人类能够利用自然技能,如同在真实的环境中一样与虚拟环境中的对象发生交互关系。例如,在原杭州大学开发的虚拟故宫游玩系统中,用户可以检验到虚拟现实系统的交互性带来的全新感受:参观者佩戴着头盔显示器,由图像发生器把立体图像送到用户的视觉中,并随着用户头部的运动,不断将更新后的新视点场景实时地显示给参观者。用户不但可以在虚拟故宫中任意行走,还可以用手(或虚拟手)去直接抓取虚拟环境中的物体。例如,当你“拿起”一件珍宝玉器时,手会有握着东西的感觉,并能感觉到物体的重量,而被抓取的物体也将随着手的移动、旋转等动作而产生实时地、相应地运动和改变,以便于用户从任意角度欣赏它们。另外,在系统中用户还可以直接控制对象的各种参数,如运动方向、速度等,而系统也可以向用户反馈信息。

虚拟现实系统的最大特点就在于它与用户的直接交互性。这种交互性粗看只是一个技术上的变化,但它出现以后,计算机界的一些观念却起了翻天覆地的变化:“以计算机为主体”的

看法逐渐被人们所抛弃，大多数人开始接受“人是信息环境的主体”这一思想：过去，人只能通过键盘、鼠标与计算机环境中的一维、二维数字信息发生交互作用。现在，VR 技术可以让我们通过使用专用的交互设备和人类的自然技能，与多维化信息的环境发生交互，从而使人机交互达到一个新的境界。

1.3.2 沉浸感

沉浸感又称临场感，它是指用户感到作为主角存在于模拟环境中的真实程度。VR 技术最主要的技术特征就是使用户具备一种在计算机环境中的沉浸感，即让用户觉得自己是计算机系统所创建的虚拟环境中的一部分，使人由观察者变为参与者，能全身心地投入计算机实践，并沉浸于其中。想象一下，当我们戴上头盔，骑上特制的自行车时，可以感到被虚拟景物所包围，看到沿途不断变化的风景，感受到迎面吹来的徐徐清风，甚至闻到路边的花香；当我们骑到崎岖不平的路面时，可以感到车身的颠簸和摇晃；而遇到上坡时，特制自行车脚踏板中的力反馈装置又将使我们感受到额外的阻力……一切都如同在已有经验的现实世界中一样。但是，当你摘下头盔时，就会发觉自己仍然是骑在自行车上，一步也没有动。这就是 VR 技术的沉浸感给我们生活带来的乐趣。

沉浸感被认为是虚拟现实系统的性能尺度，导致“沉浸感”的原因是用户对计算机环境中的虚拟物体产生了类似于对现实物体的存在意识或幻觉，它需要以下诸多方面的特性。

1) 多感知性 (Multi-Sensory)

多感知性是指除了一般计算机所具有的视觉感知外，还有听觉感知、力觉感知、触觉感知、运动感知、甚至包括味觉感知、嗅觉感知等。理想的虚拟现实就是应该具有人所具有的多种感知功能。例如，虚拟场景应能随着人的视点作全方位的运动，有比现实更逼真的纹理、灯光、照明以及声音和视频等效果，用户在操纵虚拟物体时能感受到虚拟物体的反作用力等。

2) 自主性 (Autonomy)

虚拟物体在独立活动、相互作用、或与用户的交互作用中，其动态都要有一定的表现，这些表现应服从于自然规律或是设计者想像的规律。例如，当受到力的推动时，物体会向力的方向移动、翻倒、或从桌面落到地面等。自主性就是指虚拟环境中物体依据物理定律动作的程度。

除了上面两点特性以外，影响临场感的主要因素还有体现图像中的深度信息（是否与用户的生活经验一致），画面的视野（是否足够大），实现跟踪的时间或空间响应（是否滞后或不准确），以及交互设备的约束程度（能否用户适应）等。

1.3.3 构想性

人类在许多领域面临着越来越多前所未有或前所难为而又必须解决和突破的问题，例如，载人航天、核试验、核反应堆维护、包括新武器系统在内的大型产品的设计研究、气象及自然灾害预报、医疗手术的模拟与训练以及多兵种军事联合训练与演练等。如果按传统的方法解决这些问题，必须要花费巨额资金，投入巨大的人力，消耗过长的时间，甚至要承担人员伤亡的风险。而 VR 技术的产生和发展，为解决和处理这些问题提供了新方法和新途径。所以说，VR 技术并不只是一种媒介或一个高层终端用户界面，它的应用能解决人类在工程、医学、军事等方面的很多需求。而所要考虑的关键问题就是如何开发针对虚拟现实的应用并寻找合适的场合和对象，即如何发挥人类的创造力和构想性（又称想象性）。适当的应用对象加上充分的想象力可以大幅度地提高生产效率、减轻劳动强度、提高产品开发质量。

虚拟现实系统的应用是虚拟现实与设计者并行操作,为发挥它们的创建性而设计的。过去,人只能从定量计算为主的结果中得到启发而加深对事物的认识。现在,借助于 VR 技术,人们有可能从定性和定量综合集成的虚拟环境中得到感性和理性的认识进而使人能深化概念、产生新意和构想,主动地寻求、探索信息,而不是被动地接收,这就更进一步地依赖和体现了虚拟现实的创意和构想性。

VR 技术具有的“沉浸感”、“交互性”、“构想性”,使得参与者能在虚拟环境中做到沉浸其中、超越其上、进出自如和交互自由。它强调了人在虚拟现实系统中的主导作用,即人的感受在整个系统中是最重要的。因此,“沉浸感”和“交互性”这两个特征,可以说是 VR 技术区别于任何一种其他相关技术(如三维动画、仿真、遥感与遥作、科学可视化以及传统的多媒体图形图像技术等)的本质区别。

1.4 虚拟现实的发展前景

虚拟现实发展前景十分诱人,而与网络通信特性的结合,更是人们所梦寐以求的。在某种意义上说它将改变人们的思维方式,甚至会改变人们对世界、自己、空间和时间的看法。它是一项发展中的、具有深远的潜在应用方向的新技术。利用它,可以建立真正的远程教室,在这间教室中可以和来自五湖四海的朋友们一同学习、讨论、游戏,就像在现实生活中一样。使用网络计算机及其相关的三维设备,我们的工作、生活、娱乐将更加有情趣。因为数字地球带给我们的是一个绚丽多彩的三维世界!

计算机硬件技术、网络技术及多媒体技术的融合与高速发展使得 VR 技术获得长足的发展,使 VR 技术能在 Internet 上得以实现和发展。目前,网站使用的均为二维图像与动画网页,而在网站上采用 VRML,则可以设计出虚拟现实三维立体网页场景和立体景物。利用 VR 技术可以制造出一个逼真的“虚拟人”,为医学实习、治疗、手术及科研做出贡献,也可应用于军事领域而设计一个“模拟战场”来进行大规模的高科技军事演习,既可以节省大量费用,又使部队得到了锻炼。在航空航天发射中,也可以制造一个“模拟航天器”,模拟整个航天器生产、发射、运行和回收的全过程。VR 技术还可以应用于工业、农业、商业、教学和科研等方面,其应用前景非常广阔。总之,VRML 是 21 世纪融合计算机网络、多媒体及人工智能为一体的最为优秀的开发工具和手段。

1.5 虚拟现实的系统分类

近 10 年来,随着计算机技术、网络技术、人工智能等新技术的高速发展及应用,VR 技术也发展迅速,并呈现多种化的发展趋势,其内涵也已经大大扩展。VR 技术不仅指那些采用高档可视化工作站、高档头盔式显示器等一系列昂贵设备的技术,而且包括一切与其有关的具有自然交互、逼真检验的技术与方法。VR 技术的目的在于达到真实的体验和基于自然的交互,而一般的单位或个人不可能承受昂贵的硬件设备及相应软件的价格,因此说只要达到上述部分目的的系统就可以称为虚拟现实系统。

在实际应用中,根据 VR 技术对“沉浸感”程度的高低和交互程度的不同,划分了 4 种典型类型:沉浸式虚拟现实系统、桌面式虚拟现实系统、增强式虚拟现实系统、分布式虚拟现实系统。其中桌面式虚拟现实系统因其技术非常简单、实际性强、需投入的成本也不高,在实际

应用中较为广泛。

下面对 4 个分类进行介绍。

1.5.1 沉浸式虚拟现实系统

沉浸式虚拟现实系统（Immersive VR）是一种高级的、较理想的虚拟现实系统，它提供一个完全沉浸的体验，使用户有一种仿佛置身于真实世界之中的感觉。它通常采用洞穴式立体显示装置或头盔式显示器设备，首先把用户的视觉、听觉和其他感觉封闭起来，并提供一个新的、虚拟的感觉空间，利用空间位置跟踪器、数据手套、三维鼠标等输入设备和视觉、听觉等设备，使用户产生一种身临其境、完全投入和沉浸于其中的感觉，如图 1-4 所示。



图 1-4 沉浸式虚拟现实系统

沉浸式虚拟现实系统具有以下 5 个特点。

1) 高度实时性能

沉浸式虚拟现实系统中，要达到与真实世界相同的感受，必须具有高度实时性能。如当人头部转动改变观察点时，空间位置跟踪设备须及时检测到，并且由计算机进行运算，改变输出的相应场景，要求必须有足够小的延迟，而且变化要连续平滑。

2) 高度的沉浸感

沉浸式虚拟现实系统采用多种输入/输出设备来营造一个虚拟的世界，并使用户沉浸于其中，营造一个“看起来像真的、听起来像真的、摸起来像真的、闻起来像真的、尝起来像真的”多感官的三维虚拟世界，同时使用户与真实世界完全隔离，不受外面真实世界的影响，可产生高度的沉浸感。

3) 良好的系统集成度与整合性能

为了实现用户产生全方位的沉浸，就必须多种设备与多种相关软件相互作用，且相互之间不能有影响，所以系统必须有良好的整合性能。

4) 良好的开放性

VR 技术之所以发展迅速是因为它采用其他先进技术的成果。在沉浸式虚拟现实系统中要尽可能利用最先进的硬件设备、软件技术及软件，这就要求虚拟现实系统能方便地改进硬件设备及软件技术，因此，必须用比以往更灵活的方式构造虚拟现实系统的软、硬件结构体系。

5) 能同时支持多种输入与输出设备并行工作

为了实现沉浸性，可能需要多个设备综合应用，如用手拿一个物体，就必须要有数据手套、空间位置跟踪器等设备同步工作。所以说同时支持多种输入/输出设备的并行处理是实现虚拟现实系统的一项必备技术。

常见的沉浸式虚拟现实系统有基于头盔式显示器的系统、投影式虚拟现实系统、远程存在系统。

基于头盔式虚拟现实系统是采用头盔显示器来实现单用户的立体视觉输出、立体声音输入的环境，可使用户完全投入。它把现实世界与之隔离，使用户从听觉到视觉都能投入到虚拟环境中去。

投影式虚拟现实系统是采用一个或多个大屏幕投影来实现大画面的立体视觉效果和立体声音效果，使多个用户具有完全投入的感觉。

远程存在系统是一种远程控制形式，也称遥控操作系统。它由人、人机接口、遥控操作机器人组成。实际上是遥控操作机器人代替了计算机，这里的环境是机器人工作的真实环境，这个环境是远离用户的，可能是人类无法进入的工作环境，如核环境、深海工作环境等，这时通过虚拟现实系统可使人自然地感受这种环境，完成此环境下的工作。

1.5.2 桌面式虚拟现实系统

桌面式虚拟现实（Desktop VR）系统也称窗口虚拟现实系统，是利用个人计算机或初级图形工作站等设备，以计算机屏幕作为用户观察虚拟世界的一个窗口，采用立体图形、自然交互等技术，产生三维立体空间的交互场景，通过包括键盘、鼠标和力矩球等各种输入设备操纵虚拟现实世界，实现与虚拟世界的交互，如图 1-5 所示。

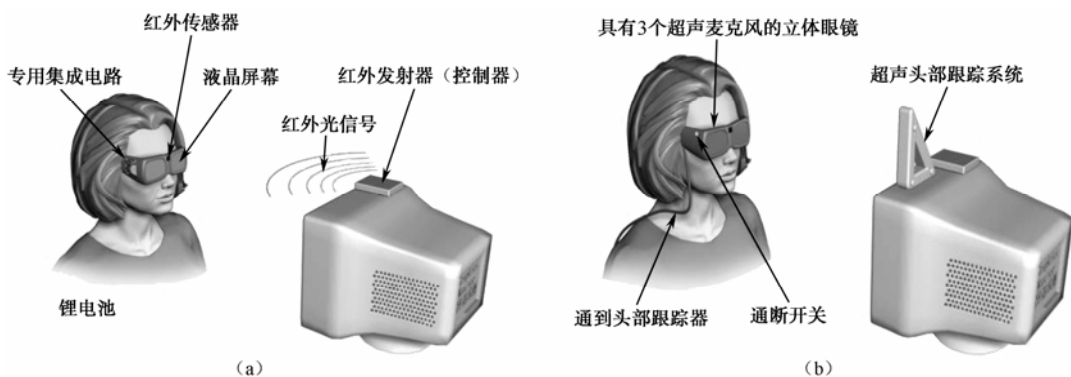


图 1-5 桌面式虚拟现实系统

桌面式虚拟现实系统一般要求参与者使用空间位置跟踪器和其他输入设备（如数据手套和 6 个自由度的三维空间鼠标），使用户虽然坐在显示器前，但可以通过计算机屏幕观察 360° 范围内的虚拟世界。

在桌面式虚拟现实系统中，计算机的屏幕是用户观察虚拟世界的一个窗口，在一些虚拟现实工具软件的帮助下，参与者可以在仿真过程中进行各种设计。使用的硬件设备主要是立体眼镜和一些交互设备（如数据手套和空间跟踪设备等）。立体眼镜用来观看计算机屏幕中的虚拟三维场景的立体效果，它所带来的立体视觉能使用户产生一定程度的沉浸感。有时为了增强桌面虚拟现实系统的效果，在桌面虚拟现实系统中还可以借助于专业的投影设备，达到增大的屏幕范围及多人观看的目的。

桌面式虚拟现实系统主要有以下 3 个特点。

(1) 用户处于完全沉浸的环境，缺少完全沉浸、身临其境的感觉，即使戴上立体眼镜，他

仍然会受到周围现实世界的干扰。

(2) 对硬件设备要求极低, 有的简单型甚至只需要计算机, 或是增加数据手套、空间跟踪设置等。

(3) 由于桌面式虚拟现实系统实现成本相对较低, 应用相对比较普遍, 而且它也具备了沉浸式虚拟现实系统的一些技术要求。

桌面式虚拟现实系统采用设备较少, 实现成本低, 对于开发者及应用者来说, 应用桌面式虚拟实现技术是从事虚拟现实研究工作的初始阶段。

1.5.3 增强式虚拟现实系统

在沉浸式虚拟现实系统中强调人的沉浸感, 即沉浸在虚拟世界中, 人所处的虚拟世界与现实世界相隔离, 看不到真实的世界也听不到真实的世界。而增强式虚拟现实 (Augmented VR) 系统既可以允许用户看到真实世界, 同时也可以看到叠加在真实世界上的虚拟对象, 它是把真实环境和虚拟环境组合在一起的一种系统, 既可减少构成复杂真实环境的开销 (因为部分真实环境由虚拟环境取代), 又可对实际物体进行操作 (因为部分物体是真实环境), 真正达到了亦真亦幻的境界。在增强式虚拟现实系统中, 虚拟对象所提供的信息往往是用户无法凭借其自身感觉器官直接感知的深层信息, 用户可以利用虚拟对象所提供的信息来加强现实世界中的认识, 如图 1-6 所示。



图 1-6 增强式虚拟现实系统

增强式虚拟现实系统主要具有以下 3 个特点。

- (1) 真实世界和虚拟世界融为一体。
- (2) 具有实时人机交互功能。
- (3) 真实世界和虚拟世界是在三维空间中整合的。

增强式虚拟现实系统可以在真实的环境中增加虚拟物体, 如在室内设计中, 可以在门、窗上增加装饰材料、改变各种式样、颜色等来审视最后的效果以达到增强现实的目的。

增强式虚拟现实系统常见的有基于台式图形显示器的系统、基于单眼显示器的系统 (一个眼睛看到显示屏上虚拟世界; 另一个眼睛看到的是真实世界)、基于光学透视式头盔显示器、基于视频透视式头盔显示器的系统。

目前, 增强现实系统常用于医学可视化、军用飞机导航、设备维护与修理、娱乐、文物古迹的复原等。典型的实例是医生在进行虚拟手术中, 戴上可透视性头盔式显示器, 既可看到做手术现场的情况, 也可以看到手术中所需的各种资料。

1.5.4 分布式虚拟现实系统

目前,计算机、通信技术的同步发展和相互促进成为全世界信息技术与产业飞速发展的主要特征。特别是网络技术的迅速崛起,使得信息应用系统在深度和广度上发生了本质性的变化,分布式虚拟现实(Distributed VR)系统是一个较为典型的实例。分布式虚拟现实系统是VR技术和网络技术发展和结合的产物,是一个在网络的虚拟世界中,位于不同地理位置的多个用户或多个虚拟世界通过网络相连接共享信息的系统,分布式虚拟现实系统的目标是在沉浸式虚拟现实系统的基础上,将分布在不同的地理位置上的多个用户或多个虚拟世界通过网络连接在一起,使每个用户同时参与到一个虚拟空间,计算机通过网络与其他用户进行交互,共同体验虚拟经历,以达到协同工作的目的,它将虚拟现实的应用提升到一个更高的境界。

虚拟现实系统运行在分布式系统下有两方面的原因:一方面是充分利用分布式计算机系统提供的强大计算功能;另一方面是有些应用本身具有分布特性,如多人通过网络进行游戏和虚拟战争模拟等。

分布式虚拟现实系统有以下特点。

- (1) 各用户具有共享的虚拟工作空间。
- (2) 伪实体的行为真实感。
- (3) 支持实时交互,共享时钟。
- (4) 多个用户可以各自不同的方式相互通信。
- (5) 资源信息共享以及允许用户自然操作虚拟世界中的对象。

根据分布式虚拟现实系统所运行的共享应用系统的个数,可以把分布式虚拟现实系统分为集中式结构和复制式结构两种。

集中式结构是指在中心服务器上运行一份共享应用系统,该系统可以是会议代理或对话管理进程,中心服务器是多个参加者的输入/输出操作进行管理,允许多个参加者信息共享。集中式结构的优点是结构简单,同时,由于同步操作只在中心服务器上完成,因而比较容易实现。缺点:由于输入和输出都要对其他所有的工作站广播,因此,对网络通信带宽有较高的要求,而且所有的活动都要通过中心服务器来协调,当参加者人数较多时,中心服务器往往会成为整个系统的瓶颈。另外,由于整个系统对网络延迟十分敏感,并且高度依赖于中心服务器,所以,这种结构的系统固性不如复制式结构。

复制式结构是指在每个参加者所在的机器上复制中心服务器,这样每个参加者进程都有一份共享应用系统。服务器接收来自于其他工作站的输入信息,并把信息传送到运行在本地机上的应用系统中,由应用系统进行所需的计算并产生必要的输出。复制式结构的优点是所需网络带宽较小。由于每个参加者只与应用系统的局部备份进行交互,所以,交互式响应效果好,而且在局部主机上生成输出,简化了异种机环境下的操作,复制应用系统依然是单线程,必要时把自己的状态多点广播到其他用户。该结构的缺点是:它比集中式结构复杂,在维护共享应用系统中的多个备份的信息或状态一致性方面比较困难,需要有控制机制来保证每个用户得到相同的输入事件序列,以实现共享应用系统所有备份必须同步,并且用户接收到的输出具有一致性。

目前,最典型的应用是SIMNET系统,SIMNET由坦克仿真器通过网络连接而成,用于部队的联合训练。通过SIMNET,位于德国的仿真器可以和位于美国的仿真器运行在同一个虚拟世界,参与同一场作战演习,如图1-7所示。

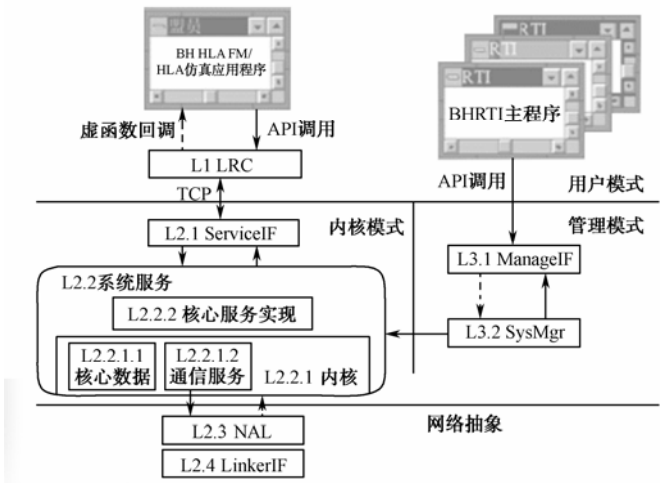


图 1-7 分布式虚拟现实系统

1.6 虚拟软件技术

虚拟现实硬件系统集成了高性能的计算机软件系统、硬件及先进的传感器设备，因此虚拟现实硬件系统设计复杂、价格昂贵，不利于 VR 技术的发展、推广和普及。这样，VR 技术软件平台的出现成为 VR 技术发展的必然。

虚拟现实软件技术（平台）以传统计算机为依托，以虚拟现实软件为基础，构造出大众化的虚拟现实三维立体场景，从而实现虚拟现实硬件零投入，只需要投入软件产品，同样可以达到虚拟现实的动态交互效果。

虚拟现实软件的典型代表有 VRML、Java3D、X3D、OpenGL 及 Vega 等软件产品。

1.6.1 VRML 的基本概念

虚拟现实三维立体网络程序设计语言 VRML 涉及有关的基本概念和名词，它们是编写 VRML 的基础。虚拟现实 VRML 的基本术语包括各种节点、事件、原型、物体造型、脚本及路由等。

1. 节点

这是 VRML 文件最基本的组成要素，是 VRML 文件基本的组成部分。节点是对客观世界中各种事物、对象、概念的抽象描述。VRML 文件就是由许多节点之间并列或层层嵌套而构成的。

2. 事件

每一个节点一般都有两种事件：一个“入事件”和一个“出事件”。在多数情况下，事件只是一个要改变域值的请求：“入事件”请求节点改变自己某个域的值，而“出事件”则是请求别的节点改变它的某个域值。

3. 原型

这是用户建立的一种新的节点类型，而不是一种“节点”。进行了原型定义就相当于扩充了 VRML 的标准节点类型集（节点的原型是节点对其中的域、入事件和出事件的声明，可以

通过原型扩充 VRML 的节点类型集)。原型的定义可以包含在使用该原型的文件中,也可以在外部定义;原型可以根据其他的 VRML 节点来定义,也可以利用特定于浏览器的扩展机制来定义。

4. 物体造型

由描述对象及其属性的节点组成。在场景图中:一类由节点构成的层次体系组成;另一类由节点事件和路由构成。

5. 脚本

这是一套程序,是与其他高级语言或数据库的接口。在 VRML 中可以用 Script 节点来利用 Java 或 JavaScript 语言编写的程序脚本来扩充 VRML 的功能。脚本通常作为一个事件级联的一部分而执行,脚本可以接受事件,处理事件中的信息,还可以产生基于处理结果的输出事件。

6. 路由

这是产生事件和接受事件的节点之间的连接通道。路由不是节点,路由说明是为了确立被指定的域的事件之间的路径而人为设定的框架。路由说明可以在 VRML 文件的顶部,也可以在文件节点的某一域中。在 VRML 文件中,路由说明与路径无关,它既可以在源节点之前,也可以在目标节点之后。在一个节点中进行说明,与该节点无任何联系。路由的作用是将各个不同的节点联系在一起,使虚拟空间具有更好的交互性、立体感、动感性和灵活性。

在场景图中,除了节点构成的层次体系结构外,还有一个“事件体系”。事件体系由相互通信的节点组成。在大多数的 VRML 节点中,每一个事件都有两个接口:输入接口——能够接收事件的节点称为 eventIn,即入事件(也称事件入口);输出接口——发送事件的节点称为 eventOut,即出事件(也称事件出口)。

入事件和出事件通过路径相连,这就是 VRML 文件除节点外的另一基本组成部分——路由(ROUTE)。ROUTE 语句把事件出口和事件入口联系起来,从而构成了“事件体系”。

1.6.2 VRML的立体空间计量单位

VRML 要创建立体背景空间和立体空间造型就需要定位,就需要立体空间坐标系、相应的长度、角度单位及空间立体着色等。

1. VRML立体空间坐标系

VRML 立体空间物体造型定位依靠 VRML 立体空间坐标系来实现,它包括 X 轴、Y 轴、Z 轴,这些坐标轴为每个空间物体的造型定义了一个坐标系。在三维立体空间中,X 轴、Y 轴和 Z 轴相交的点构成该坐标系的原点,X 轴的正方向指向右边,Y 轴的正方向指向正上方,Z 轴的正方向指向前方(即浏览者所在的方向),如图 1-8 所示。空间物体的造型在该坐标中的位置由相对于该坐标原点的三维坐标来确定。

2. VRML长度单位

VRML 三维立体空间长度单位是统一的,只有一个单位标量,即 VRML 单位。VRML 单位并不是一个绝对的尺寸大小,因此 VRML 尺寸和现实中的长度单位不具有任何可比性,只有在 VRML 单位之间

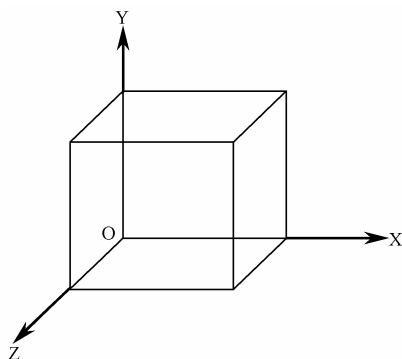


图 1-8 VRML 三维立体空间坐标系

才具有可比性，VRML 单位一般比实际的长度单位要小。在实际开发工作中，考虑到多个开发者在开发工作中协调统一的问题，通常都要指定一个统一的参考标准，即以实际中的长度单位为准。

3. VRML角度单位

在 VRML 立体空间中使用的角度单位不是普通的角度，而是通常所使用的弧度。当在 VRML 立体空间中使用角度单位时，先将角度单位换算成弧度单位后，再将其编写入 VRML 源程序中。常用到的角度与弧度换算见表 1-1。

表 1-1 角度弧度对照表

角度	0°	30°	45°	60°	90°	120°	135°	150°	180°
弧度	0	0.524	0.785	1.047	1.571	2.094	2.356	2.618	3.141

4. VRML空间立体着色

在 VRML 三维立体空间中着色，无论是立体空间背景、光线，还是立体空间中的各种物体，它们的颜色都是由 3 种基本颜色——红、绿、蓝（RGB）组合而成。红、绿、蓝 3 种基本颜色对应 3 个浮点数，它们的域值分别为 0.0~1.0。

红、绿、蓝 3 种颜色可以组成各种各样姹紫嫣红的“颜色”。常见的主要颜色表 1-2。

表 1-2 常用三种颜色组合

红（Red）	绿（Green）	蓝（Blue）	合成颜色
0.0	0.0	0.0	黑色
0.0	0.0	1.0	蓝色
0.0	1.0	0.0	绿色
1.0	0.0	0.0	红色
1.0	1.0	1.0	白色
1.0	1.0	0.0	黄色
0.0	1.0	1.0	青蓝色
1.0	0.0	1.0	紫红色
0.75	0.75	0.75	浅灰色
0.25	0.25	0.25	暗灰色
0.5	0.5	0.5	中灰色
0.5	0.0	0.5	暗红色
0.0	0.5	0.0	暗绿色
0.0	0.0	0.5	暗蓝色

1.6.3 VRML语法

VRML 文件的语法主要包括 VRML 文件头、节点、原型、造型、脚本和路由等。不是每一个 VRML 文件都必须包括这几个部分，只有 VRML 文件头是必需的，而其他项不一定是必需的。VRML 立体空间的场景和造型是由节点构成的，再通过路由实现动态交互和感知或者使用脚本文件与外部接口进行动态交互。在软件开发中，根据不同的实际情况编写不同 VRML

程序，它所包含的成分也可能不同。在 VRML 文件中，“节点”是核心、灵魂，如果没有节点，VRML 也就不存在了。

1. VRML文件

VRML 文件可由文本编辑器编写或由专用编辑器 VmlPad 编写。VRML 文件的扩展名为 .wrl 或 .wrz，使用较多的是 .wrl 文件。任何扩展名为 .wrl 文件都是 VRML 文件。通过 VRML 浏览器可以直接运行 VRML 文件。

VRML 文件名的全称为：*.wrl 或*.wrz。

2. VRML文件的结构

VRML 文件的语法结构由 VRML 文件头、节点、事件、脚本和路由等组成。“节点”可以由一个或多个组成，也可以创建新的节点，即原型节点。

VRML 文件的语法结构如下：

#VRML V2.0 utf8	#VRML 文件在第一行必须有头文件
节点名{	#VRML 中各种类型的"节点"
域 域值	#相应"节点"的"域"和"域值"
...	
}	
Script {	#脚本 Script 节点
}	
ROUTE	#路由:把入事件与出事件联系起来

在 VRML 文件的结构中，要突出软件工程的设计思想，使 VRML 文件的结构设计层次清晰、结构严谨、条理清楚，使读者、用户易于理解和掌握。

3. VRML文件头

在每一个 VRML 文件中，文件头是必须，且位于 VRML 文件的第一行。

VRML 文件头的语法结构是：#VRML V2.0 utf8。

VRML 文件头的作用相当于高级 C 语言程序中的主函数的作用，每一个 C 语言程序必须从 main() 开始，否则编译程序时就会出错。

VRML 文件头包含 4 个部分，每一个部分代表一定的含义。

(1) 以 # 开头，这里的 # 不是注释含义，而是 VRML 文件头的一部分。

(2) “VRML”告诉浏览器该文件是一个 VRML 文件。

(3) “V2.0”告诉浏览器该 VRML 文件遵循 VRML 规范的 2.0 版本。

(4) “utf8”表示该 VRML 文件是一个使用国际 UTF-8 字符集的文件。UTF-8 的英文全称是“UCS Transform Fromat”，而 UCS 是“Universal Character Set”的缩写。国际 UTF-8 字符集包含任何计算机键盘上能够找到的字符，而多数计算机使用的 ASCII 字符集则是 UTF-8 字符集的子集，因此，使用 UTF-8 书写和阅读 VRML 文件很方便。UTF-8 支持多种语言字符集，由国际标准化组织的“ISO 10646-1: 1993”标准定义。

VRML 文件头必须按上述语法格式书写，必须位于 VRML 文件的第一行，不得随意更改。

4. VRML文件注释

在编写 VRML 源程序时，为了使源程序结构更合理、更清晰及层次更强，经常在源程序

中添加注释信息。在 VRML 文件中允许编程人员在文件的任何地方进行注释说明,以进一步增加源程序的可读性,从而使层次清晰、结构合理,形成好的文档资料,并符合软件开发的要求。

VRML 文件注释部分以一个符号“#”开头,结束于该行的末尾。当注释信息多于一行时,将产生语法错误,因此 VRML 不支持多行注释信息。浏览器在浏览 VRML 文件时将跳过“#”之后本行的所有内容。另外,浏览器在浏览 VRML 文件时将自动忽略 VRML 文件中的所有空格和空行。开发人员可根据软件开发的结构和思想编写 VRML 源程序。

5. 事件、路由和脚本

1) 事件

在 VRML 中,每一个节点一般都有两种事件:一个“入事件”和一个“出事件”,每一个节点通过这些“入事件”和“出事件”来改变节点自身的某个域的值。在多数情况下,每个节点事件只是一个要改变域值的请求:“入事件”请求节点改变自己某个域的值,而“出事件”则是请求别的节点改变它的某个赋值。

“事件”相当于高级程序语言中的函数调用:“入事件”相当于高级程序语言中函数调用时的入口参数,“出事件”相当于高级程序语言中函数调用返回时的参数。

在 VRML 中的每一个节点内部都有一些定义为“暴露域”(exposedField)的内容。一个“暴露域”能够接收“入事件”,也能产生“出事件”。事件的调用是短暂的,事件的值是不会被写入 VRML 文件中的。“暴露域”就像变量一样被放置在 VRML 文件中。例如,如果一个节点中的“暴露域”叫做 translation,那么它的“入事件”为 set_translation,而“出事件”则为 translation_changed。

每一个节点的“入事件”和“出事件”的命名规则如下。

(1) 大部分“入事件”都是以“set_”为开头的事件,除了 addChildren 和 removeChildren 这两个“入事件”。

(2) 大部分“出事件”都以“_changed”为结尾的事件,除了形式为 SFBool 的“出事件”,但布尔值的“出事件”是以 is 作为开头的,如 isBound、isActive 等。

(3) “入事件”和“出事件”的形式为 SFTime 时,就不再使用“set_”和“_changed”作为连接,如 bindTime 和 touchTime 等。

2) 路由

路由是连接一个节点的“入事件”和“出事件”的节点之间的通道。路由不是节点,它只是简单的语法结构,只说明为了确立被指定的域的一个事件如何从某个节点到达另一个节点。路由说明可以在 VRML 文件的顶部,也可以在 VRML 文件节点的某一个域中。

路由的作用是将各个不同的节点联系在一起,使虚拟空间具有更好的交互性、立体感、感知性和灵活性。在 VRML 中,每一个节点都有输入和输出接口,它们分别是“入事件”和“出事件”,但有一些节点不同时具有这两种事件。一个节点一般具有多个不同的“入事件”和“出事件”。

路由绑定不同节点时应注意以下 3 点。

(1) 两个节点的路由在没有被触发之前一直都处于休眠的状态,只有在被触发时,才有事件从输出接口的节点输出,通过路由传送到输入接口的节点,从而引起相应虚拟现实世界中的某种变化。

(2) 在 VRML 中,节点的“入事件”和“出事件”同样具有一定的数值类型。

(3) 在 VRML 中, 可以将多个节点绑定在一起, 从而创建复杂的路线 (联系), 以便在虚拟现实世界中实现更真实的交互。

3) 脚本

脚本是一套程序, 是与各种高级语言或数据库的接口。在 VRML 文件的两个节点之间存在着路由, 事件可以通过相应的路径从一个节点传送到另一个节点。此外, 还可以通过添加脚本对这些事件与路由进行编程设计, 这能产生更加鲜明的交互性和生动逼真的动画效果。

在 VRML 中可以用 Script 节点利用 Java 或 JavaScript 语言编写的程序脚本来扩充 VRML 的功能。脚本通常作为一个事件级联的一部分而执行, 脚本可以接受事件, 处理事件中的信息, 还可以产生基于处理结果的输出事件。

例如, Transform 空间变换节点的语法结构定义如下:

```
Transform
{
  children[]           #暴露域 MF 节点
  translation 0.0 0.0 0.0 #暴露域 SFVec3f
  rotation 0.0 0.0 1.0 0.0 #暴露域 SFRotation
  ...
}
```

VRML 源文件如下:

```
#VRML V2.0 utf8           #VRML 文件在第一行必须有的头文件
节点名{                   #VRML 中各种类型的"节点"
域 域值                   #相应"节点"的"域"和"域值"
... ...
}
DEF zjz Transform {       #重新定义节点为 zjz
children [
...
DEF sensor PlaneSensor{} #PlaneSensor 平面检测器节点
]
Script{                  #脚本 Script 节点
}
ROUTE sensor.translation_changed TO zjz.set_translation
#ROUTE 路由:把 Transform 节点的域 translation 的"入事件"与"出事件"联系起来
}
```

注意: 上例只举出一种路由与事件之间关系的应用, 在实际编程设计中要学会举一反三。

1.6.4 VRML的节点和域

节点是 VRML 文件中最基本的单位, 节点是 VRML 的精髓与核心。每个节点还包含子节点和描述节点属性的“域名”或“域值”, 在高级语言中称为变量、数组等, 在数据库中常称为字段。

单一节点的语法定义如下:

```
节点名{
    域名    域值    #域值类型说明
    ...    ....
}
```

节点由节点名、一对花括号组成，花括号内又包括节点的域名、域值及域值类型说明等。下面以锥体节点 **Cone** 为例来加以说明。

节点名	域	域值	域值类型
↓	↓	↓	↓
Cone{	外		# Cone 节点（注释）
	bottomRadius	1.0	# field SFFloat 单值浮点域
	height	2.0	#field SFFloat 单值浮点域
	side	TRUE	#field SFFloat 单值浮点域
	bottom	TRUE	#field SFBool 单值布尔域
}			

其中，“#”为注释语句。

节点的作用是描述空间造型（对象）及其属性。“节点”可以理解为高级语言中的函数、子程序、结构体及可视化编程语言中的类和对象等。

VRML 节点的层次关系分为父节点和子节点。父节点即根节点。在一个嵌套节点中，最顶层的节点就是父节点，由它派生的节点称为子节点。例如，Group 编组节点中，有一个域 children[]，children 就是子节点，而 Group 就是父节点，如图 1-9 所示。

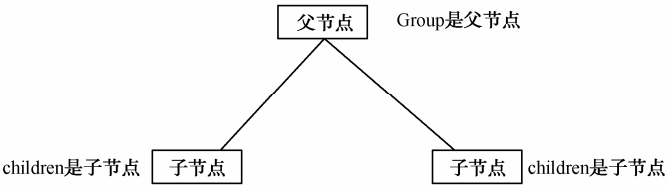


图 1-9 节点层次关系图

1. 节点的域名和域值

节点中包括“域”、“域值”或子节点。根据语法结构的要求，节点之间可以并列或层层嵌套使用。不同作用的节点有着不同的语法结构，父节点包括“域”、“域值”或子节点，子节点中也包含“域”、“域值”等。

域分为两种：一种称为“域”；另一种称为“暴露域”。“域”相当于高级语言中的普通变量，而“暴露域”相当于 C 语言中的外部变量，相当于可视编程语言 VC++中的公共变量或公共函数。

域包含有各种类型的数据，可以是单值的或多值的。“域”可理解为高级语言中的“变量”、“数组”及 C 语言中的“结构体变量”或数据库中的“字段”等。

每一个节点都是由一个域多个域组成的，域即“属性”，描述空间造型尺寸。每个域有相应的“域值”，这些域值指定了一个场景的尺寸、大小和颜色等特征。“域值”相当于高级语言中的变量取值或取值范围，也相当于数据库中的字段取值。

同一节点中的域遵循如下规则。

- 同一节点中的各个域之间无先后次序之分。
- 同一节点中的各个域均有自己相应的默认值。
- 不同的域对应不同的域值类型。

域名：域的标识符叫域名。在节点作用域的范围內，域名是唯一的，也是 VRML 的保留字或关键字。

常见的两类域名的域值类型前缀如下。

- 单值类型的域名，以“SF”开始，只包含单值。单值可以是一个独立的数，也可以是一个向量、颜色的几个数等，甚至可以是图像数据。
- 还有一种多值域名以“MF”开始，它包含多个值，所以称为多值域名。

在 VRML 文件中，表示多域值的方法是：整个用方括号括起来的一系列用逗号和空格隔开的单值。如果一个多值域不包含任何值，则只标出方括号“[]”。如果一个多值域恰好只包含一个数，可以不写括号，直接写该值。

例如，以“MF”开始的多域值的表示如下：

```
skyColor[
0.2 1.0 1.0
0.6 0.8 1.0
1.0 0.2 0.5
]
```

在 VRML 文件中，各个节点语法的各种“域值类型”详细说明为：单值域类型名用“SF”表示，多值域名称用“MF”表示。

1) SFBool

SFBool 域是一个单值布尔量，常用于开启或关闭一个节点的性质特征。SFBool 域输出事件的默认值为 TRUE（真），否则为 FALSE（假）。

2) SFFlot 和 MFflot

SFFlot 域是单值单精度浮点数，含有一个 ANSI C 格式的单精度浮点数。SFFlot 域输出事件的初始值是 0.0。

MFflot 域是多值单精度浮点数，含有零个或多个 ANSI C 格式的单精度浮点数。MFflot 域输出事件的初始值是 []。

3) SFInt32 和 MFInt32

SFInt32 域是一个单值含有 32 位的整数。一个 SFInt32 值由一个十进制或十六进制格式（以 0X 开头）的整数构成。SFInt32 域输出事件的初始值为 0。

MFInt32 域是一个多值域，由任意数量的以逗号或空格分隔的整数组成。MFInt32 域输出事件的初始值为 []。

4) SFString 和 MFString

SFString 域包含一个字符串。SFString 的域值含有双引号括起来的字符串。任何字符都可在引号中出现。为了在字符串中使用双引号，在它之前加一个反斜杠“\”，为了在字符串中使用反斜杠，可连续使用两个反斜杠“\\”。SFString 域输出事件的初始值为 “”。

MFString 域是一个含有零个或多个单值的多值域，每个单值都和 SFString 值的格式相同。MFString 域输出事件的初始值为 []。

5) SFVect2f 和 MFVec2f

SFVect2f 域定义了一个二维向量。SFVect2f 的域值由两个分隔的浮点数组成。SFVect2f 域输出事件的初始值为 (0 0)。

MFVec2f 域是一个包含任意数量的二维向量的多值域。MFVec2f 域输出事件的初始值为 []。

6) SFVec3f 和 MFVec3f

SFVec3f 域定义了一个三维向量空间。一个 SFVec3f 域值包含有 3 个浮点数，数与数之间用空格分隔。该值表示从原点到给定点的向量。SFVec3f 域输出事件的初始值为 (0 0 0)。

MFVec3f 域是一个包含任意数量的三维向量的多值域。MFVec3f 域输出事件的初始值为 []。

7) SFTIME 和 MFTime

SFTIME 域含有一个单独的时间值。每个时间值是一个 ANSI C 格式的双精度浮点数，表示从 1970 年 1 月 1 日 (GMT, 格林尼治平均时) 子夜开始计时，延续当前时间的秒数。SFTIME 域输出事件的初始值为 -1。

MFTime 域包含任意数量的时间值。

8) SFRotation 和 MFRotation

SFRotation 域规定了一个绕任意轴的任意角度的旋转。SFRotation 的域值含有 4 个浮点数，各数之间用空格分隔，前 3 个数表示旋转轴，第 4 个数表示围绕该轴旋转多少弧度。SFRotation 域输出事件的初始值为 (0 0 1 0)。

MFRotation 域可包含任意数量的这类旋转值。MFRotation 域输出事件的初始值为 []。

9) SFImage

SFImage 域含有非压缩的二维彩色图像或灰度图像。SFImage 域首先列出 3 个整数值，前两个整数表示图像的宽度和高度，第三个整数表示构成图像格式的元素个数，随后按“宽度×高度”的格式列出一组十六进制数，数与数之间用空格或逗号分隔，每一个十六进制数表示图像中一个单独的像素，像素的排列按从左到右、从下到上的顺序。第一个十六进制数描述一个图像左下角的像素，最后一个则描述右上角的像素。SFImage 域输出事件的初始值为 (0 0 0)。

10) SFCOLOR 和 MFColor

SFCOLOR 域是只有一个颜色的单值域。SFCOLOR 的域值由一组 3 个浮点数组成，每个数都在 0.1~1.0 范围内，分别表示构成颜色的红、绿、蓝 3 个分量。SFCOLOR 域输出事件的初始值是 (0 0 0)。

MFColor 域是一个多值域，包含任意数量的 RGB 颜色值。MFColor 域输出事件的初始值是 []。

11) SFNode 和 MFNode

SFNode 域含有一个单节点，必须按标准节点语法写成。一个 SFNode 域允许包含一个关键字，表示不包含任何节点。SFNode 域输出事件的初始值为 NULL。

MFNode 域包含任意数量的节点。MFNode 域输出事件的初始值为 []。在编写 VRML 文件、节点和域时，特别要注意大小写，因为 VRML 源文件对大小写很敏感，即 VRML 源文件要严格区分大小写。

2. 节点名的重定义和重用

在 VRML 虚拟空间的场景中，为了减少 VRML 源代码的编写量，提高 VRML 编程效率，对重复的造型或多个同样的造型在场景中多次出现时，可通过定义这个造型节点，然后再重复调用该定义的节点。

1) 重定义节点：DEF

在 VRML 中，使用 DEF 命令可为空间的造型重新定义节点，以便在以后的节点重用或调用时引用，从而使程序编写简练，减少程序代码的重复和冗余，而且具有层次清晰、结构合理的软件工程设计的风格。

重定义节点的语法定义如下：

```
DEF 节点名 节点类型 { }
```

其中，DEF 是重定义节点的 VRML 关键字，节点名可以由任何字母和数字及下划线组成，但不要使用与 VRML 保留字或关键字相重复的节点名。

在 VRML 中，对节点名的规定是：节点名区分大小写，命名不能以数字开头，不能带有非印刷的 ASCII 字符，不能包括单引号、双引号、数字运算或英镑符号等。

2) 重用节点：USE

当重定义一个节点名之后，便可在 VRML 源程序中重复使用该节点名，即可以一次或多次调用该节点名。

重用节点的语法定义如下：

```
USE 节点名
```

其中，USE 是 VRML 的保留字，节点名就是前面用 DEF 重定义的节点名。

节点重新定义和重复引用的文件结构形式如下：

VRML V2.0 utf8	#VRML 文件在第一行必须有的头文件
节点名{	#VRML 中各种类型的"节点"
域 域值	
... ..	
}	
DEF 节点 1 节点类型{	#定义节点
域 域值	#相应"节点"的"域"和"域值"
... ..	
}	
USE 节点名 1	#重用节点语法

1.7 虚拟现实主要应用领域

VR 技术已经发展很多年，虚拟现实的应用领域也越来越广泛，最初是用于军事仿真，近年来在城市规划、室内设计、文物保护、交通模拟、虚拟现实游戏、工业设计、远程教育等方面都取得了巨大的发展，并且将来运用会越来越广泛。下面简要介绍目前虚拟现实的应用领域。

1.7.1 城市规划

在城市规划中经常会用到 VR 技术。用 VR 技术不仅能十分直观地表现虚拟的城市环境，而且能很好地模拟各种天气情况下的城市，可以一目了然地了解排水系统、供电系统、道路交通、沟渠湖泊等，并且能模拟飓风、火灾、水灾、地震等自然灾害的突发情况。VR 技术对于政府在城市规划工作中起到了举足轻重的作用，图 1-10 所示为虚拟城市效果图。



图 1-10 虚拟城市效果图

1.7.2 室内设计



图 1-11 虚拟室内设计

在室内设计应用方面,用 VR 技术不仅能十分完美地表现室内的环境,而且能在三维的室内空间中自由行走。目前业内常用 VR 技术做室内 360° 全景展示和室内漫游,受到一致好评,而且不仅能在室内漫游,还能用 VR 技术做预装修系统,可以实现即时动态地对墙壁的颜色进行更换或贴上不同材质的墙纸,还可以更换地面的颜色或贴上不同的木地板、瓷砖等,更能移动家具的摆放位置、更换不同的装饰物。这一切都在 VR 技术下将被完美地实现,如图 1-11 所示。

1.7.3 文物保护

VR 技术在文物保护方面也是应用相当广泛的,埃及的金字塔就做过网上的体验中心,运用了全景虚拟技术和三维虚拟技术,而且 IBM 目前正在运用 VR 技术对北京故宫进行整个故宫的数字虚拟。届时大家也许可以在网上直接看到数字三维化的故宫,如图 1-12 所示。



图 1-12 虚拟故宫

1.7.4 军事与航空航天

VR 技术根源可以追溯到军事领域,军事应用是推动 VR 技术发展的源动力,直到现在依然是虚拟现实系统的最大应用领域,在军事和航天领域早已理解仿真和训练的重要性。当前趋势是减少经费开支、提高演习效果和改善军用硬件的生命周期等。

1. 军事上的应用

军事上的应用中,采用虚拟现实系统不仅提高了作战能力和指挥效能,而且大大减少了军费开支,节省了大量人力、物力,同时保障了人员的生命安全。与VR技术最为相关的应用有军事训练和武器的设计制造等。

1) 军事训练方面

现在各个国家都习惯于采用举行实战演习来训练军事人员和士兵,但是这种实战演练,特别是大规模的军事演习,将耗费大量资金和军用物资,安全性差,而且还很难在实战演习条件下改变状态,来反复进行各种战场态势下的战术和决策研究。近年来,随着VR技术在军事上的应用,使演习与训练在概念和方法上有了一个飞跃,如图1-13所示。目前,军事训练领域主要用于以下4个方面。



图1-13 军事训练

(1) 虚拟战场环境。利用虚拟现实系统生成相应的三维战场环境图形图像数据库,包括作战背景、战地场景、各种武器装备和作战人员等,并通过网络等手段为使用者创造一种逼真的立体战场世界,以增强其临场感觉,提高训练的效率。

在20世纪80年代初,美国国防先进研究课题局(ADRPA)开始研究第一个真正的虚拟战场SIMNET,这是为了在联合演习中训练坦克队伍。这个尝试的最初动机是减少训练代价,同时也增加了安全性和减少了环境影响(爆炸和坦克轨迹会大大破坏训练场地)。北大西洋公约组织同盟国将逐步把各国军事力量集成放进一个虚拟战场,这有利于联合军力作战。

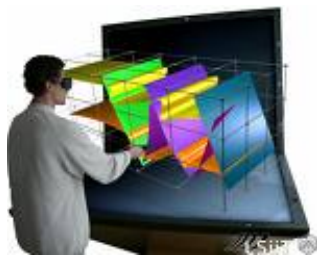


图1-14 DVENET平台

北京航空航天大学虚拟现实与可视化新技术教育部重点实验室在国家“863计划”支持下,作为集成单位,与国防科技大学、海军潜艇学院、装甲兵工程学院等单位一起建立了一个用于VR技术研究和应用的分布式虚拟世界基础信息平台DVENET。DVENET由一个专用广域计算机网络以及支持分布式虚拟世界研究与应用的各種标准、开发工具和基础信息数据(如3D逼真地形)组成,如图1-14所示。

(2) 近战战术训练。近战战术训练系统把在地理上分散的各个单位、战术分队的多个训练模拟器和仿真器连接起来,以目前的武器系统、配置等为基础,把陆军的近战战术训练系统、空军的合成战术训练系统、防空合成战术训练系统、野战炮兵合成战术训练系统、工程兵合成战术训练系统,通过局部网和广域网连接起来。这样的虚拟作战世界,可以使众多军事单位参与到作战模拟之中,而不受地域、空间的限制,具有动态的、分布交互作用;进行战役理论和作战计划的检验,并预测军事行动和作战计划的效果;可以评估武器系统的总体性能,启发新的作战思想。

(3) 单兵模拟训练。让士兵穿上数据衣服,戴上头盔显示器和数据手套,通过操作传感装置选择不同的战场场景,练习不同的处置方案,体验不同的作战效果,进而像参加实战一样,锻炼和提高技术、战术水平,快速反应能力和心理承受力。美国空军用VR技术研制的飞行训练模拟器,能进行视觉控制,能处理三维实时交互图形,且有图形以外的声音和触感,不但能以正常方式操纵和控制飞行器,还能处理系统中飞机以外的各种情况,如气球的威胁、导弹的发射轨迹等。

还有一个基于单兵训练的课题是 TNO Physics Electronics Laboratory (物理电子实验室) 在荷兰开发的“虚拟 Stinger 训练器”。Stinger 是为防御低空飞机设计的紧凑的士兵发射的火箭, 它用于全世界很多军队。荷兰军队使用的标准的 Stinger 训练器包括 20m 直径的投影拱顶。背景景色由安装在拱顶上的一台有鱼镜头的投影机投影。指导者能确定攻击场景, 并用工作站跟踪训练过程。

(4) 诸军兵种联合战略战术演习。建立一个“虚拟战场”, 使陆、海、空多军种现处一个战场, 根据虚拟世界中的各种情况及其变化, 实施联合演习。利用 VR 技术, 根据侦察的资料合成出战场全景图, 让受训指挥员通过传感器装置观察各军种兵力部署和战场情况, 以便模拟相互配合, 共同作战。

2) 在武器装备研究与新武器展示中的应用

(1) 在武器设计研制过程中, 采用 VR 技术提供先期演示, 检验设计方案, 把先进设计思想融入武器装备研制的全过程, 从而保证总体质量和效能, 实现武器装备投资的最佳选择, 对于有些无法进行试验或试验成本太高的武器研制工作, 也可由虚拟现实系统来完成, 所以尽管不进行武器试验, 也能不断改进武器性能。

(2) 研制者和用户 VR 技术, 可以很方便地介入系统建模和仿真实验的全过程, 既能加快武器系统的研制周期, 又能合理评估其作战效能及其操作的合理性, 使之更接近实战的要求。

(3) 采用 VR 技术对未来高技术战争的战场环境、武器装备的技术性能和使用效率等方面进行仿真, 有利于选择重点发展的武器装备体系, 改善其整体质量和作战效果。

(4) 很多武器供应商借助于网络, 采用虚拟现实系统来展示武器的各种性能。

2. 航空航天方面的应用

众所周知, 航天飞行是一项耗资巨大、变量参数很多、非常复杂的系统工程, 其安全性、可靠性是航天器设计时必须考虑的重要问题。因此, 可利用将 VR 技术与仿真实验相结合的方法来进行飞行任务或操作的模拟, 以代替某些费时、费力、费钱的真实试验或者真实试验无法开展的场合, 利用 VR 技术的经济、安全及可重复性等特点, 从而获得提高航天员工作效率、航天器系统可靠性等的设计对策。

美国政府把虚拟现实看成保持美国技术优越的战略努力的一部分, 并开始了“高性能计算和计算机通信”计划 (HPCC)。在这个计划中, 资助开发先进的计算机硬件、软件和应用, 给虚拟现实的研究与开发产生了很大的推动。

在航空航天方面, 美国国家航空航天局于 20 世纪 80 年代初就开始研究 VR 技术。在 1984 年, 美国艾姆斯航天研究中心利用流行的液晶显示电视和其他设备开始研究低成本的虚拟现实系统, 这对于 VR 技术的软、硬件研制发展推动很大。90 年代以来, 虚拟现实的研究与应用范围不断扩大。例如, 美国马歇尔空间飞行中心研制载人航天器的虚拟现实座舱, 指导座舱布局设计并训练航天员熟悉航天器的舱内布局、界面和位置关系, 演练飞行程序。目前, 美国各大航天中心已广泛地应用 VR 技术开展相关领域内的研究工作, 宇航员利用虚拟现实系统进行了失重心理等各种训练。美国航空航天局计划将虚拟现实系统用于国际空间站组装、训练等工作。

1) NASA 的虚拟现实训练

1993 年 12 月, 人类在太空成功地更换了哈勃太空望远镜上有缺陷的仪器板。在这之前的工作中, 美国约翰逊航天中心启用了一套虚拟现实系统来训练航天员熟悉太空环境, 为修复哈勃望远镜作准备。航天员通过操作虚拟设备, 大大提高了操作水平, 使修复工作取得了圆满成功。

2) EVA 的虚拟现实训练

EVA (欧洲航天局) 近些年来在探索把 VR 技术用于提高宇航员训练、空间机器人遥控和航天器设计水平等方面的可能性, 而近期内的计划重点是开发用于宇航员舱外活动训练、月球与火星探测模拟, 以及把地球遥感卫星的探测数据转化为三维可视图像用的虚拟现实系统。

3) 英国空军的虚拟座舱

虚拟座舱方向早期的工作于 1991 年在巴黎国际航空展览上发布。演示的是英国空军“虚拟环境布局训练辅助 (VECTA)”课题的研究结果。在这个早期研究阶段, 系统包括一对 SGI210 显示生成器, 具有 Polhemus 跟踪器的低分辨率 VPL EyePhone 等, 但存在着 HMD 图形分辨率低、缺乏纹理映射等问题。

在“神舟”七号飞船发射任务的准备和实施过程中, 航天发射一体化仿真训练系统起到了重要作用。航天发射一体化仿真训练系统是采用半实物仿真技术、虚拟仪器技术、VR 技术, 形成一套融虚拟装备、测试发射、测量控制、指挥通信、地勤支持于一体的大型系统, 可以实现发射场全系统、全流程、全人员的综合训练, 从而有效提高参加航天发射人员的技术水平。具体地说, 在没有火箭、飞船目标的情况下, 系统可以把火箭、飞船的信息虚拟出来, 组织模拟发射场全系统参加的火箭测试发射, 从而大大缩短产品研制开发的周期, 节省研发成本。

1.7.5 虚拟教育与培训

1. 虚拟校园

虚拟校园是指从 Internet、VR 技术、网上虚拟社区和 3S 技术的发展角度, 对现实大学三维景观和教学环境的虚拟化和数字化, 是基于现实大学的一个三维虚拟环境, 用于支持现实大学的资源管理、环境规划和学校发展。

虚拟校园在现在很多高校都有成功的例子, 先后有浙江大学、清华大学、上海交通大学、北京大学、中国人民大学、山东大学、西北大学、杭州电子工业大学、西南交通大学、中国海洋大学等高校, 都采用 VR 技术构建了虚拟校园。

大学对每个人来说都有着特殊的感情, 大学校园的学习氛围、校园文化对我们具有巨大影响, 教师、同学、教室、实验室等, 以及校园的一草一木无不潜移默化地影响我们每一个人, 大学校园赋予我们的教益从某种程序来说, 远远超出书本所给予我们的。网络 VR 技术的应用, 使我们仿真校园环境, 因此, 虚拟校园成了 VR 技术与网络教育早期的具体应用, 图 1-15 为湖南大学的虚拟校园。



图 1-15 虚拟校园

2. 虚拟演示教学与试验

在学校教育中, VR 技术在教学中应用较多, 特别是对于理工科类课程的教学, 尤其在建筑、机械、物理、生物、化学等学科有着质的突破。它不仅适用于课堂教学, 使其更形象生动, 也适用于互动性试验中。很多大学都有 VR 技术研究中心或实验室, 如杭州电子工业大学虚拟现实与多媒体研究所, 研究人员把虚拟现实应用于教学, 开发了虚拟教育环境。

浙江大学 CAD&CG 国家重点实验室虚拟现实与多媒体研究室在承担的欧盟科技项目中(与英国 Salford 大学、葡萄牙里斯本大学合作), 开发了基于虚拟人物的电子学习环境(ELVIS), 用来辅助 9~12 岁的小学生进行故事创作, 研究人员设计了一组虚拟人物, 并支持不同情绪变化。

VR 技术在仿真领域方面, 特别是交互性仿真方面尤为突出, 如西南交通大学致力于工程漫游方面的虚拟现实应用, 开发出了一系列有国际水平的计算机仿真和虚拟现实应用产品, 在此基础上, 还开发了虚拟现实模拟培训系统、交互式仿真系统。

中国科技大学运用 VR 技术, 开发了几何光学设计试验平台, 它是全国第一套基于虚拟现实的教学软件。它用计算机制作的虚拟智能仪器代替价格昂贵、操作复杂、容易损坏、维修困难的试验仪器, 具有操作简便、效果真实、物理图像清晰、着重突出物理实验设计思想的特点。

利用 VR 技术在课堂上可对办公自动化设备进行三维展示, 并模拟通电运行, 特别对于有些学校缺乏相应的试验设备的情况下, 一方面可大大提高教学效果, 另一方面可节省大量的实验投资。

3. 远程教育系统

随着 Internet 技术的发展、网络教育的深入, 远程教育有了新的发展, 真实、互动、情节化、突破了物理时空的限制并有效地利用了共享资源, 这些特点可以弥补远程教学条件的不足, 彻底打破空间、时间的限制, 它可以虚拟历史人物、伟人、名人、教师、学生、医生等各种人物形象, 创设一个人性化的学习环境, 使接受远程教育的学生能够在自然、亲切的气氛中进行学习。也可以利用虚拟现实系统来虚拟试验设备, 使学生足不出户便可以做各种各样的试验, 获得与真实试验一样的体会, 从而丰富感性认识, 加深对教学内容的理解, 同时可避免真实试验或操作所带来的各种危险。

正是 VR 技术独特的魅力所在, 基于国际 Internet 的远程教育系统具有巨大的发展前景, 也必将引起教育方式的革命。如中央广播电视大学, 投入较大的人力和物力, 采用基于互联网的类游戏图形引擎, 将网络学院具体的实际功能整合在图形引擎中, 突破了目前大多数 VR 技术的应用仅仅停留在一般性浏览应用上的限制。

4. 特殊教育

由于 VR 技术是一种基于自然的交互形式, 这个特点对于一些特殊的教育有其特殊的作用。在国家“863 计划”、国家自然科学基金委、北京科委等的支持下, 中国科学院计算技术研究所多年来积极从事多功能感知技术的研究, 开发集语音、体态、情感于一体的“中国手语合成系统”, 2004 年 4 月 16 日, 研发的新版中国手语合成系统 V2.0(中国手语电子词典 V2.0)完成系统开发和内部测试。中国手语合成系统是利用计算机技术, 将正常的语言或词语转变成计算机三维手语。通过虚拟人合成和手语合成技术, 用三维虚拟人来展示合成的手语。该软件还可以合成任意的句子和文章, 帮助用户从多角度多方位学习标准的中国手语, 帮助听力障碍者与正常人进行交流。

新版中国手语合成系统重点对原来的手语库进行了升级, 提高了手语的标准性, 加入了人

脸表情和“眨眼”等个性化脸部动作、增加了 3 个更有亲和力的女性主持人形象的虚拟人模型。运用这套系统还可以在任意显示设备上用手语发布信息，实现手语导游。这有助于聋人使用电视、网络、电话、计算机、电影等高科技产品，提高生活质量，并有助于手语的标准化，改善聋人受教育的环境，给聋人朋友的学习和生活带来极大方便，如图 1-16 所示。

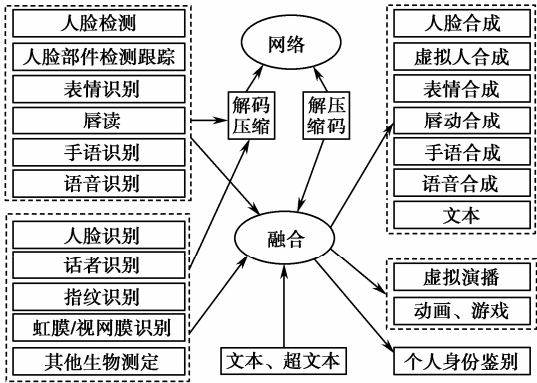


图 1-16 手语合成系统

日本京都的先进电子通信研究所（ATR）系统研究实验室的开发者们开发的一套系统，能用图像处理来识别手势和面部表情，并把它们作为系统输入。该系统提供了一个更加自然的接口，而不需要操作者带上任何特殊的设备。

5. 技能培训

将 VR 技术应用于技能培训可以使培训工作更加安全，并节约了成本。比较典型的应用是训练飞行员的模拟器及用于汽车驾驶的培训系统。交互式飞机模拟驾驶器是一种小型的动感模拟设备，舱体内前面是显示屏幕，配备飞行手柄和战斗手柄，在虚拟的飞机驾驶训练系统中，学员可以反复操作控制设备，学习在各种天气情况下驾驶飞机起飞、降落，通过反复训练，达到熟练掌握驾驶技术的目的，如图 1-17 所示。

交互式汽车模拟驾驶器采用 VR 技术构造一个模拟真车的环境，通过视景仿真、声音仿真、驾驶系统仿真，给驾驶人员以真车般的感觉，让驾驶学员在轻松、安全、舒适的环境中掌握汽车的常识，学会汽车驾驶，又可以体验疯狂飞车的乐趣，集科普、学车以及娱乐于一体，如图 1-18 所示。



图 1-17 飞机模拟驾驶



图 1-18 汽车模拟驾驶

在我国“神舟”五号载人飞船发射项目中，也采用模拟训练器来辅助发射训练工作，“神

舟”五号模拟训练器系统包括有飞船系统、运载系统、监控系统、着陆系统等，对驾驶的“神舟”载人飞船发射升空、白天和黑夜在空中运行状态以及返回着陆等进行模拟。

1.7.6 商业领域

在商业方面，近年来，VR 技术常被用于产品的展示与推销。采用 VR 技术来进行展示，全方位地对商品进行展览，展示商品的多种功能，另外还能模拟工作时的情景，包括声音、图像等效果，比单纯使用文字或图片宣传更加有吸引力。并且这种展示可用于 Internet 之中，可实现网络上的三维互动，为电子商务服务，同时顾客在选购商品时可根据自己的意愿自由组合，并实时看到它的效果。全国第一家 3D 全景购物网开通，它采用全景展示技术来对所出售的商品进行展示，创建网络贸易新亮点，如图 1-19 所示。



图 1-19 网上三维商城

1.7.7 娱乐

娱乐上的应用是 VR 技术应用最广阔的领域，从早期的立体电影到现代高级的沉浸式游戏，是 VR 技术应用较多的领域之一。丰富的感觉能力与三维显示世界使得虚拟现实成为理想的视频游戏工具。

作为传输显示信息的媒体，虚拟现实在未来艺术领域方面所具有的潜在应用能力也不可低估。虚拟现实所具有的临场参与感与交互能力可以将静态的艺术（如油画、雕刻等）转化为动态的，可以使观赏者更好地欣赏作者的思想艺术。如虚拟博物馆，利用网络或光盘等其他载体实现远程访问。另外，虚拟现实提高了艺术表现能力，如一个虚拟的音乐家可以演奏各种各样的乐器，即使远在外地，也可以在居室中去虚拟的音乐厅欣赏音乐会等。

世界第一个较大的虚拟现实娱乐系统是“BattleTech Center”，是 1990 年 8 月在芝加哥开放的。主题是 3025 年的未来战争，由称“BattleMech”的人控制的强大的机器人作战。

世界第一个基于 HMD 的娱乐系统是 1991 年英国 W-Industries Ltd.开发的。这个沉浸的虚拟现实电子游戏系统称为“Virtuality”，以后在美国由 Horizon Entertainment 销售。专门设计的 Virtuality 的立体 HMD 基于 LCD，它使用 4 路声音和游戏者之间进行语音通信。

浙江大学 CAD&CG 国家重点实验室开发了虚拟乒乓球、虚拟网络马拉松和“轻松保龄球艺健身器”等项目，并与国家体育总局合作进行体育训练仿真，开发了“大型团体操作演练仿

真系统”、“帆板帆船仿真系统”等项目。

1.7.8 工业应用

随着 VR 技术的发展,其应用已大幅进入民用市场。如在工业设计中,虚拟样机就是利用 VR 技术和科学计算可视化技术,根据产品的计算机辅助设计(CAD)模型和数据以及计算机辅助工程(CAE)仿真和分析的结果,所生成的一种具有沉浸感和真实感、并可进行直观交互的产品样机。

虚拟制造技术于 20 世纪 80 年代提出来,在 90 年代随着计算机技术的迅速发展,得到人们的极大重视而获得迅速发展。虚拟制造是采用计算机仿真和 VR 技术在分布技术环境中开展群组协同工作,支持企业实现产品的异地设计、制造和装配,是 CAD/CAM 等技术的高级阶段。利用 VR 技术、仿真技术等计算机上建立起的虚拟制造环境是一种接近人们自然活动的一种“自然”环境,人们的视觉、触觉和听觉都与实际环境接近。人们在这样的环境中进行产品的开发,可以充分发挥技术人员的想象力和创建能力,相互协作发挥集体智慧,大大提高产品开发的质量和缩短开发周期。目前应用主要在以下几个方面。

1. 产品的外形设计

汽车工业是采用 VR 技术的先驱。一般情况下开发或设计一辆新式汽车,从初始设想到汽车出厂大约需要两年或更多的时间,当图纸设计好后,以前多采用泡沫塑料或黏土制作外形模型,然后通过多次的评测和修改,还需要许多后续的工序去研究基本外形、检测空气动力学性能、调整乘客的人机工程学特性等。而采用 VR 技术可随时修改、评测,可以大大地缩短这一周期,因为采用 VR 技术设计与制造汽车不需要建造实体模型,可以简化很多工序,并根据 CAD 和 CAE 程序所收集的有关汽车设计的数据库进行仿真。在其他产品(如飞机、建筑、家用电器、物品包装设计等)外形设计中,均表现出极大的优势。

2. 产品的布局设计

在复杂产品的布局设计中,通过 VR 技术可以直观地进行设计,甚至走入到产品中去,这样可避免出现很多不合理问题。例如,工厂和车间设计的机器布置、管道铺设、物流系统等,都需要该技术的支持。在复杂的管道系统、液压集流块设计中,设计者可以“进入”其中进行管道布置,检查可能的干涉等错误。在汽车、飞机的内部设计中,“直观”是最有效的工具,VR 技术可发挥不可替代的积极作用。

3. 机械产品的运动仿真

在产品阶段中必须解决运动构件在运动过程中的运动协调关系、运动范围设计、可能的运动干涉检查等。

4. 虚拟装配

机械产品中有成千上万的零件要装配在一起,其配合设计、可装配性是设计人员常常出现的错误,往往要到产品最后装配时才能发现,造成零件的报废和工期的延误,不能及时交货造成巨大的经济损失和信誉损失。采用 VR 技术可以在设计阶段就进行验证,保证设计的正确。

汽车工业所用的虚拟现实应用程序中,技术人员可以在仿真过程中尝试装配汽车零部件,因此,在花费时间和金钱去制造实际的零件之前,就可以将各个零部件虚拟地装配在一起。

5. 产品加工过程仿真

产品加工是个复杂的过程。产品设计的合理性、可加工性、加工方法和加工设备的选用、

加工过程中可能出现的加工缺陷等,有时在设计时是不容易发现和确定的,必须经过仿真和分析。通过仿真,可以预先发现问题,采取修改设计或其他措施,保证工期和产品质量。

6. 虚拟样机

在产品的设计、重新制造等一系列的反复试制过程,许多不合理设计和错误设计只能等到制造、装配过程中,甚至到样机试验才能发现。产品的质量和工作性能也只能当产品生产出来后,通过试运转才能判定。这时,多数问题是无法更改的,修改设计就意味着部分或全部的报废和重新试制。因此常常要进行多次试制才能达到要求,试制周期长,费用高。而采用虚拟制造技术,可以在设计阶段就对设计的方案、结构等进行仿真,解决大多数问题,提高一次试制成功率。用虚拟样机技术取代传统的硬件样机,可以大大节约新产品开发的周期和费用,很容易地发现许多以前难以发现的设计问题,如图 1-20 所示。



图 1-20 汽车虚拟制造

同时,虚拟样机技术还可明显地改善开发团体成员之间的意见交流方式。VR 技术还允许公司的主管人员、技术人员等对汽车的外形等做出评价,研究各个零部件如何装配在一起及审核最终产品,这一切都不需要建造零部件或整车的模型。

德国汽车业应用 VR 技术最快也最广泛。目前,德国所有的汽车制造企业都建成了自己的虚拟现实开发中心,建立了面向汽车虚拟设计、虚拟装配、维护和维修和虚拟现实系统。奔驰、宝马、大众等大公司的报告显示,应用 VR 技术,以“数字汽车”模型来代替木制或铁皮制的汽车模型,可将新车型开发时间从一年以上缩短到 2 个月左右,开发成本最多可降低到原先的 1/10。

美国伊利诺斯州立大学的研制车辆设计中,研究人员提出采用支持远程协作的分布式虚拟现实系统进行设计,不同国家、不同地区的工程师们可以通过计算机网络实时协作进行设计。在设计车辆的过程中,各种部分都可以共享一个虚拟世界,并且可以查看对方任何一个位置的图像,通过视频传递和相应的定位方向。在系统中采用了虚拟快速成型技术,从而减少了设计图像和新产品进入市场的时间,这样,产品在生产之前就可以估算和测试,并且大大提高了产品质量。

日本东京大学的高级科学研究中心将他们的研究重点放在远程控制方面,其开发的系统可以使用户控制远程摄像系统和一个模拟人手的随动机械人手臂。

1.7.9 医学领域

VR 技术在医学方面的应用具有十分重要的现实意义。在虚拟环境中,可以建立虚拟的人体模型,如图 1-21 所示。借助于跟踪球、HMD、感觉手套,学生可以很容易了解人体内部各

器官结构, 这比现有的采用教科书的方式要有效得多。Pieper 及 Satara 等研究者在 20 世纪 90 年代初基于两个 SGI 工作站建立了一个虚拟外科手术训练器, 用于腿部及腹部外科手术模拟。这个虚拟的环境包括虚拟的手术台与手术灯, 虚拟的外科工具(如手术刀、注射器、手术钳等), 虚拟的人体模型与器官等。借助于 HMD 及感觉手套, 使用者可以对虚拟的人体模型进行手术。但该系统有待进一步改进, 如需提高环境的真实感, 增加网络功能, 使其能同时培训多个使用者, 或可在地专家的指导下工作等。

在医学院校, 学生可在虚拟实验室中, 进行“尸体”解剖和各种手术练习, 如图 1-22 所示。用这项技术, 由于不受标本、场地等的限制, 培训费用大大降低。一些用于医学培训、实习和研究的虚拟现实系统, 仿真程度非常高, 其优越性和效果是不可估量和不可比拟的。例如, 导管插入动脉的模拟器, 可以使学生反复实践导管插入动脉时的操作; 眼睛手术模拟器, 根据人眼的前眼结构创造出三维立体图像, 并带有实时的触觉反馈, 学生利用它可以观察模拟移去晶状体的全过程, 并观察到眼睛前部结构的血管、虹膜和巩膜组织及角膜的透明度等。还有麻醉虚拟现实系统、口腔手术模拟器等。

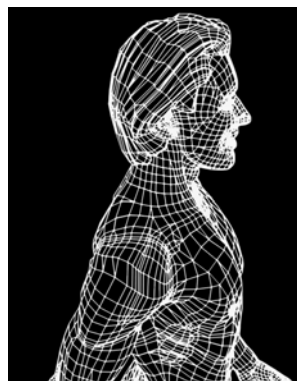


图 1-21 虚拟的人体模型



图 1-22 虚拟解剖图

外科医生在真正动手术之前, 通过 VR 技术的帮助, 能在显示器上重复地模拟手术, 移动人体内的器官, 寻找最佳手术方案并提高熟练度。另外, 在远距离遥控外科手术、复杂手术的计划安排、手术过程的信息指导、手术后果预测及改善残疾人生活状况、乃至新药研制等方面, VR 技术都能发挥十分重要的作用。

1.7.10 虚拟现实在Web3D/产品/静物展示中的应用

Web3D主要有 4 类运用方向: 商业、教育、娱乐和 虚拟社区。对企业和电子商务三维的表现形式, 能够全方位地展现一个物体, 具有二维平面图像不可比拟的优势。企业将他们的产品发布成网上三维的形式, 能够展现出产品外形的方方面面, 加上互动操作, 演示产品的功能和使用操作, 充分利用互联网高速迅捷的传播优势来推广公司的产品。对于网上电子商务, 将销售产品展示做成在线三维的形式, 顾客通过观察和操作能够对产品有更加全面的认识了解, 决定购买的概率必将大幅增加, 为销售者带来更多的利润。

对教育业现今的教学方式, 不再是单纯的依靠书本、教师授课的形式。计算机辅助教学

(CAI) 的引入, 弥补了传统教学所不能达到的许多方面。在表现一些空间立体化的知识, 如原子、分子的结构、分子的结合过程、机械的运动时, 三维的展现形式必然使学习过程形象化, 学生更容易接受和掌握。许多实际经验告诉我们, 做比听和说更能接受更多的信息。使用具有交互功能的三维课件, 学生可以在实际的动手操作中得到更深的体会。

对计算机远程教育系统而言, 引入 Web3D 内容必将达到很好的在线教育效果。娱乐游戏业永远是一个不衰的市场。现今, 互联网上已不是单一静止的世界, 动态 HTML、flash 动画、流式音视频, 使整个互联网呈现生机盎然。动感的页面较之静态页面更能吸引更多的浏览者。三维的引入, 必将造成新一轮的视觉冲击, 使网页的访问量提升。娱乐站点可以在页面上建立三维虚拟主持这样的角色来吸引浏览者。游戏公司除了在光盘上发布三维游戏外, 现在可以在网络环境中运行在线三维游戏。利用互联网的优势, 受众和覆盖面得到迅速扩张。

对虚拟现实展示与虚拟社区使用 Web3D 实现网络上的虚拟现实展示, 只需构建一个三维场景, 人以第一视角在其中穿行。场景和控制者之间能产生交互, 加之高质量的生成画面使人产生身临其境的感觉。对于像虚拟展厅、建筑房地产 虚拟漫游展示, 提供了解决方案。如果是建立一个多用户而且可以互相传递信息的环境, 也就形成了所谓的虚拟社区。

第2章 虚拟现实软件及三维立体造型

2.1 软件开发模型

软件项目开发包括传统的瀑布开发模型、原型开发模型、渐进式开发模型等几种模式。下面分别作介绍。

2.1.1 软件瀑布开发模型

软件工程实践是利用工程的概念、原理、技术和方法对软件项目进行开发、设计和维护的实践活动，是运用先进的软件项目管理技术和开发工具来完成软件项目开发任务。

软件工程项目的开发分为软件项目的定义时期、软件项目的开发时期和软件项目的维护时期3个阶段。

1) 软件项目的定义时期

软件项目的定义时期是确立软件项目开发所要完成的总体目标，其主要任务是工程项目的立项、工程项目的可行性论证及工程项目的需求分析，并估算该项目需要的资源（计算机硬件、软件和人力）、成本和预算，制定项目开发计划等。

2) 软件项目的开发时期

软件项目的开发时期是指软件项目具体设计、开发和实施的过程，即根据在软件项目定义时期产生的需求分析结果，进行系统的概要设计、详细设计、编码和集成测试工作。

3) 软件项目的维护时期

软件项目的维护时期是指软件项目开发完成后，产生的软件产品（版本）在投入市场前需要经历一段时间的试运行。在试运行过程中，还会暴露一些在软件开发过程中没有遇到的新问题或故障，需要在运行维护阶段解决这些问题和故障。在试运行之后，认为正确的软件产品才能正式投放市场。软件产品正式投放市场后，还要进一步进行维护，也就是软件产品的版本升级过程，所以也把运行和维护称为二次开发。

软件项目开发的传统瀑布模型就是采用自上而下的开发手段，包括工程立项（提出问题）、可行性分析、需求分析、总体设计、详细设计、编码、集成测试（单元测试和综合测试）、运行维护等几个阶段。

软件项目开发的传统瀑布模型如图 2-1 所示。

软件工程立项是确立一个有意义的软件开发项目，可以是学术型的、科学研究型的或实际工程型的。要明确该项目的总体目标、模型、性质等，形成文档并研究讨论其可行性。

软件项目可行性分析是在较短的时间内确定该软件工程项目是否有解或值得去解。如果有解，就要导出系统高层逻辑模型，即数据流图。可行性分析包括技术可行性、经济可行性和操作可行性几个方面，并形成相关文档。如果无解，就停止该项目的开发工作。

软件项目需求分析是为该项目开发提出完整、准确、清晰、具体的工作要求，主要是软件开发人员与项目有关的业务人员进行协商来共同完成软件项目的需求。确定对系统的综合要求主要包括系统功能要求、系统性能要求、运行要求和未来可能提出的要求。要分析系统的数据

要求，导出系统的逻辑模型（逻辑模型=综合要求+数据要求），形成层次方框图。

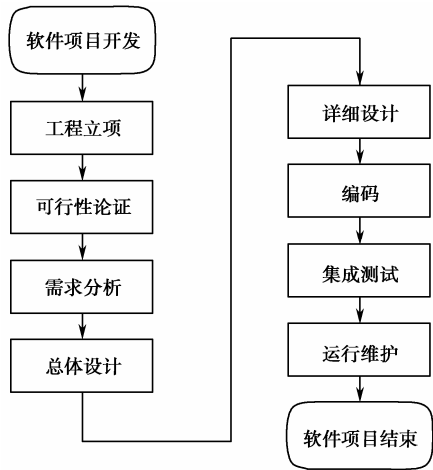


图 2-1 软件项目开发的传统瀑布模型

软件项目总体设计也称为概要设计，是指在可行性论证和需求分析阶段产生的各种不同方案中，通过数据流图和层次方框图得到一份系统层次图或结构图，并分析比较每一个合理的方案，以便从中选出一个最佳方案。总体设计主要完成软件项目系统设计，确定系统的具体实施方案及软件项目的总体结构设计。软件项目总体设计过程包括设想供选择的方案、选取一个合理方案、推荐一个最佳方案、软件项目功能分解、设计软件项目总体结构、数据设计、制定测试计划、形成文档资料及审查和复审。

软件项目详细设计是确定每一个具体模块的功能要求，设计的处理过程要尽可能简单、明了、清晰，并产生程序流程图和形成文档。

软件项目编码阶段是根据项目要求，运用计算机语言和各种开发工具进行设计编码。对于工业控制，可以采用汇编语言；对于操作系统软件，可采用 C 语言和汇编语言；对于应用软件，可以采用各种高级语言和数据库技术等；对于可视化编码，可采用 C#.net、J++及虚拟现实语言等。

软件项目集成测试主要包括单元测试和综合测试两大部分。单元测试是在详细设计和编码阶段进行，综合测试是在完成单元测试的基础上对全部软件项目进行调试。综合测试工作量大，难度较大，需要成熟且有经验的高级编程人员进行。软件测试的方法有黑盒测试和白盒测试。软件的测试步骤为模块测试、子系统测试、集成测试、试运行测试等。

软件项目的运行维护是指在试运行和交付使用期间为更正软件项目开发中的错误或满足新的需要（功能）而开发和修改软件的过程，包括改正性维护、适应性维护、完善性维护、预防性维护 and 安全性维护等。

2.1.2 软件原型开发模型

软件项目的原型开发模型就是在软件工程项目开发之前，首先开发一套样板软件产品模型或 demo 提供给用户。利用样板软件产品模型（原型）可以尽早获得更准确、更完善的需求，为软件项目的真正开发消除障碍、铺平道路，这种方法称为原型法。

在软件项目开发中使用原型系统的目的是通过原型样板软件产品展示未来系统将怎样为

用户服务，更直接、更具体、更方便，从而可以使用户提出更准确、更全面、更完善的需求。用户通过试用原型系统能够提出系统的哪些功能和特性是他们需要和喜欢的，哪些功能和特性是不需要的，以及他们还需要增加哪些新的功能。通过试用原型系统，可与用户进行现场演示交流，产生真正满足用户需要的“需求分析结果”，为软件项目的成功开发奠定坚实的基础。

原型法软件项目开发模型的特点是开发出原型样板软件产品演示给用户，这样会增加一部分开发成本费用，但是带来的回报却是更准确、更全面、更完善的需求分析结果，从而为软件项目的正式开发（后续开发）奠定坚实的基础。

原型法软件项目开发模型如图 2-2 所示。

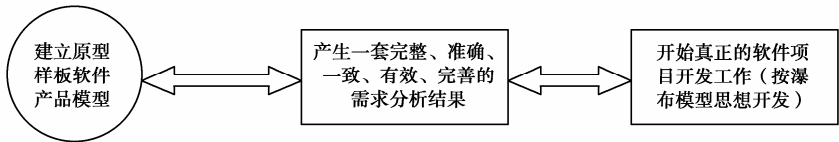


图 2-2 原型法软件项目开发模型

支持原型法的依据有以下几个方面。

(1) 并非所有需求分析都能预先定义，在很多情况下，用户对其目标和需求最初是笼统的、模糊不清的，许多细节也是含糊不清。要求一个对计算机系统不太熟悉的用户提出准确无误的全部需求分析是不切实际的，所以要使用原型系统。

(2) 存在着项目参与者之间的交流和沟通障碍。大、中型软件项目的开发需要系统分析员、软件工程师、程序员、管理者和用户等各类人员的共同参与，在工作中要协同一致，沟通和交流至关重要。为方便参与者之间的沟通、交流，需要使用原型系统。

(3) 可以利用快速建造原型工具。通用的超高级语言是基本的建造原型工具。超高级语言的语句功能很强，可以用少数语句实现一个系统，但超高级语言运行时需要很大的支持系统，这就增加了需要的存储容量，降低了该语言的执行速度，因此，超高级语言不适合于开发较大的实际软件系统，但对原型系统却是非常适合。通用的软件工具能将需求说明转换成可执行程序等。

实现原型系统的途径：首先，用于验证软件需求的原型，即用适当的工具快速构造出可运行的原型系统，由用户来测试（试用）和评估；其次，用于验证设计方案的原型，即为保证软件项目开发的质量，在总体设计和详细设计中用原型法来验证总体结构及关键算法的正确性；最后，用于演示目标系统的原型，即通过演示过程确定需求策略，使开发人员和用户对系统的理解逐渐加深，不断对原型进行修改、改进、扩充和完善，直到用户满意为止。

2.1.3 软件渐进式开发模型

渐进式软件项目开发模型吸收了瀑布开发模型和原型开发模型的优点，但剔除了瀑布和原型开发模型的不足。瀑布开发模型开发到中后期时关键技术和需求缺陷才暴露出来，从而造成软件项目开发延期、重做或失败。在软件项目开发的测试阶段发现错误或重大错误而导致开发项目失败的事例屡见不鲜。在 3 万个失败项目中，统计结果表明该软件项目有 45%的需求分析是无用的。

原型开发模型虽然能够解决瀑布开发模型中的不足，但它本身也存在着缺陷，那就是创建原型样板产品时要抽入一定的开发成本。当原型样本产品使用完后，该原型样板产品就作废，

不能直接应用到软件项目的开发中，从而造成了原型样板产品开发的浪费。

渐进式软件项目开发模型的特点是，在每一个开发阶段的软件产品，既是与用户交流和沟通的演示产品，又是软件开发项目的目标产品，因此，可在交流中不断完善目标软件产品，直到用户满意为止。按照这种策略，像“滚雪球”一样，在分析、沟通和交流中开发软件项目，循序渐进，循环开发，一直到完成整个软件项目的全部开发工作。渐进式软件项目开发模型如图 2-3 所示。

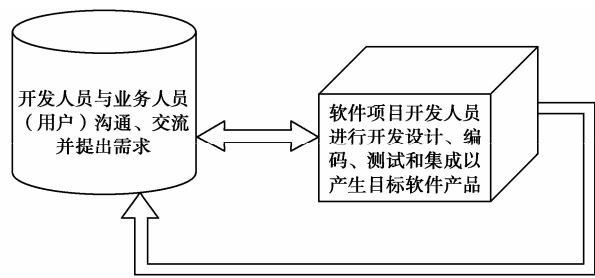


图 2-3 渐进式软件项目开发模型

渐进式软件项目开发模型是目前最先进、最优秀的软件工程项目开发模型。它的开发分为两个阶段：第一阶段是开发者与用户（客户）初期合作，随时与用户沟通和交流，需用 4~5 个周期（一个周期为 2~3 周）的时间来进行从需求分析到编码，再到测试的全过程，以完成初步项目开发任务；第二阶段是在第一阶段的基础上，如果双方不能继续合作，则可以取消该项目的开发工作，如果双方能够继续合作，则可以把该项目按此开发方式循序渐进地开发完成。

在渐进式软件项目开发模型开发过程的第一阶段中，软件开发人员根据用户初步需求对一个或多个模块进行设计、编码和测试，产生目标软件产品（初始版本），将该产品交与用户演示并反复交流和沟通，以提出新的补充需求，并对初级软件产品进行修改、补充和完善，直到用户满意为止，形成最终软件产品（一个或多个模块）。

在第一阶段规定的时间内，解决软件项目开发的关键技术并使用户满意而双方达成一致，签好软件项目开发合同，再进行第二阶段的真正实质性的软件项目开发工作。这样，第一阶段的软件开发工作投入就没有白做，可在第一阶段工作的基础上继续进行该项目的软件开发工作。随着软件项目开发的不断进展及软件产品的不断生产、测试和集成，就会像“滚雪球”一样，将软件产品逐渐生产积累起来，最终完成全部软件项目的开发工作。

软件项目渐进式开发模型的策略如下所述。

1) 完成软件开发的范围（Scope）

在每次提出新的需求过程中，要确定在每个周期内完成多少任务，根据实际开发进度完成情况进一步调整所要完成任务的多少。

2) 软件开发的质量（Quality）

每个周期内根据需求所完成的软件开发一定是正确的。

3) 软件开发的成本及资源（Cost/Resource）

软件项目开发中使用的硬件、软件、人力资源的合理分配。

4) 软件项目开发时间（Time）

如何在有效的 4~5 个周期的时间内完成 90%的任务，而且完成的任务一定是正确的。

5) 软件项目自动测试 (Automatic Test)

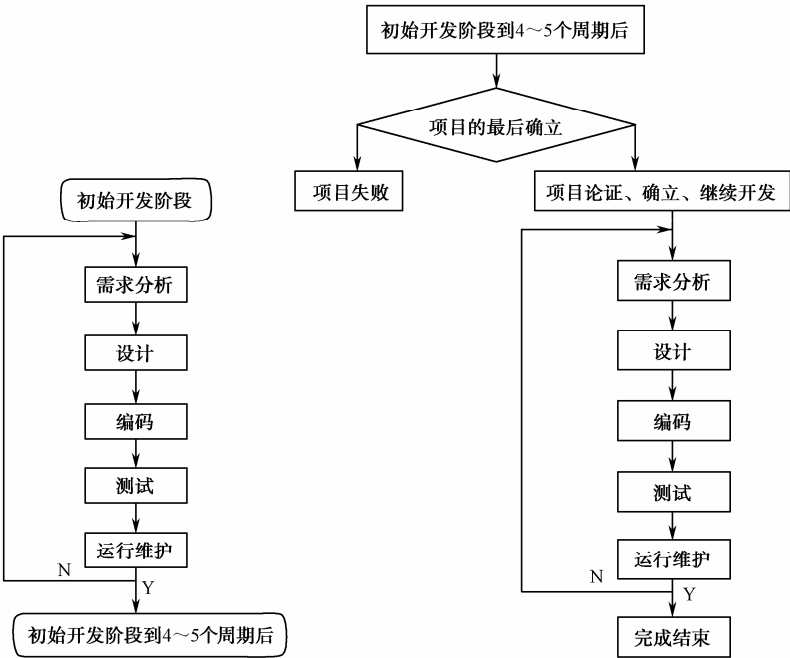
这是最终完成项目开发的技术保证，利用测试工具软件或将用户需求写在测试程序中，来检验所开发的每一个程序中的每一条语句代码的正确性，以确保开发出来的全部程序的正确性。

在软件项目开发初期，双方在完成软件项目开发任务的同时，还要保证软件的开发质量、成本及交货期。在软件项目开发初期，既不能延误软件项目的开发时间，也不能降低软件开发质量，更不能增加软件项目开发成本，只能减少在软件开发范围中的任务数量，以保证软件项目的开发质量、成本和开发时间。

在运用渐进式开发模型对大、中型软件项目进行开发时，开发者与用户要签证合同。开发过程一般分为两个阶段。第一阶段为 4~5 个周期的（一个周期为 2~3 周）时间，开发时间和费用是固定不变，开发的时间较短，投资费用较少，并且开发的产品不是演示 demo 产品，而是实际产品。第二阶段双方再提出该软件项目下一步开发的造价、费用、成本及时间，如果双方不能达成共识，则该项目的开发到此结束；如果双方能够达成共识，该项目可以签署第二阶段合同，继续开发，一直到完成整个项目的全部开发任务。这样做的目的是减少双方的投资风险，使双方在第一阶段结束时，对该项目的成败又多提供了一次决策的机会。因为第一阶段用户投入较少费用和时间，所以，如果不成功，双方损失都不大，如果成功可以继续合作，前期工作又不浪费。

渐进式（“滚雪球”式）软件项目开发模型如图 2-4 所示。

在渐进式软件项目开发的全过程中，为保证软件项目开发质量，常采用以下技术手段。



(a) 第一阶段：磨合阶段 (b) 第二阶段：项目确立、继续开发至完成全部工作

图 2-4 渐进式软件项目开发模型

1) 绝对编程技术

利用两个一组进行编程的开发方法，一个人编程，另一个人监督检查，以提高编程质量，减少差错和故障率。每一个程序员在编写自己的软件程序时，对自己编写的软件程序是有“感情”的，总是认为自己编写的程序是“正确”的。因此，要对源程序进行换人校对、检查，才能更有效地查出程序错误和故障点。

2) 自动测试技术

利用测试工具软件将用户需求写在测试程序中，来检验所开发的每一个程序中的每一条语句代码的正确性，以确保软件项目开发出来的全部程序的正确性。

3) 先测试后设计

在软件项目开发前，先编写测试程序，后开发应用程序。要把用户的需求写到测试程序中，测试程序要求简单、明了、不出错。测试程序与实际程序一一对应，形成完善的自动调试机制，从而使测试编程自动化。

4) 简单编程

在软件项目开发过程中，少用或不用编程技巧，即用最简单、最直接的方法编程和设计，以减少编程的复杂度，提高编程的质量和效率。

5) 组件技术

这是指将软件“硬件”化，把软件模块（组件）看成各自独立“集成模块”，不管内部设计、结构如何，只关心“组件”的功能和接口。这样在使用软件组件时，先考虑软件组件的功能及接口，然后再将其集成到所开发的系统中。

软件开发团队可以自己开发一些通用的软件组件，并通过实践验证其是否为正确的软件组件；也可以购买一些商用的通用软件组件，以减少开发量，提高软件项目开发速度。但在购买商用的通用软件组件时，要注意选择一家资质比较好的公司来购买通用软件组件。购买商用的通用软件的优点是：商用的通用软件组件维护、升级都由卖方公司负责，从而减少商用的通用软件的维护和管理费用。在购买商用的通用软件组件时，不要贪图便宜而到多家软件公司购买不同的商用通用软件组件，一旦购买的商用通用软件组件出现故障或错误，会出现各家公司“扯皮”现象，互相推诿，推卸责任，而造成不必要的麻烦。

6) 软件重用技术

这是提高软件开发质量、效率的有效手段。软件重用主要指它的易用性、有用性（可用性）和广泛性。可重用技术产品包括可执行代码重用、源代码重用、设计重用、需求规范重用、测试数据重用、文档重用、软件架构重用等。

在现在软件项目开发中，软件组件、特定领域组件及编写代码的比例为：通用成分组件占 20%，特定领域组件占 65%，应用特定代码占 15%。通过分析可知，在软件项目开发中利用软件重用技术可以减少软件开发的工作量，降低成本和提高软件项目的开发效益。

保证软件项目开发质量需着重考虑的因素包括功能性、可靠性、易用性、效率、可维护性及可移植性等。

1) 功能性（Functionality）

在软件项目开发中，首先要保证用户提出的功能需求，其次要按功能需求划分为各种功能模块、组件进行分析、设计和编程，最后将各种模块和组件进行集成测试。

2) 可靠性（Reliability）

这是指软件产品使用中的故障率，故障率低说明开发的软件产品质量优异；反之，软件质

量次之。

3) 易用性 (Usability)

这是指为了方便用户使用,在软件项目开发过程中为用户设计良好的用户界面,采用可视化、三维立体场景进行编程设计,以便于用户使用和操作。

4) 效率 (Efficiency)

这是指计算机资源利用率,包括人力、硬件、软件及网络资源利用效率。在软件设计方面,效率的提高可能会影响其他一些性能指标,例如,影响到可移植性、可靠性和维护性等。

5) 可维护性 (Maintainability)

这是指软件在运行和使用期间的可修改性,以及对运行环境的适应能力,主要表现为该软件系统可以方便地维护、修改和扩充,有很强的适应性。可维护性能力的增强会影响到效率。

6) 可移植性 (Portability)

这是指软件产品在不同环境平台上所具有的适应能力,对不同计算机硬件的适应能力及软件操作系统的适应能力。软件产品不加任何修改就可以移植到不同机器和操作系统中,便是移植性最好的软件项目系统。

2.2 虚拟现实软件开发工具

1. 使用记事本来编写VRML源程序

VRML 编辑器是用来编写 VRML 源程序所用的编辑工具,它包括 VRML 文本编辑器和专用编辑器等。

VRML 源文件是一种 ASCII 码的描述语言,可用一般计算机中提供的文本编辑器编写 VRML 源程序,也可以使用 VRML 专用编辑器来编写源程序。

在 Windows 操作系统中,选择【开始】菜单的【程序】子菜单下【附件】子菜单的【记事本】选项,然后在记事本编辑状态下创建一个新文件,便可开始编写 VRML 源程序。但要注意所编写的 VRML 源程序的文件名。大家都知道,程序名(文件名)由“文件名.扩展名”组成,并且在 VRML 文件中要求文件的扩展名必须以.wrl 或.wrz 结尾,否则 VRML 浏览器就无法识别。用文本编辑器编辑 VRML 源程序来完成软件项目开发的过程如图 2-5 所示。

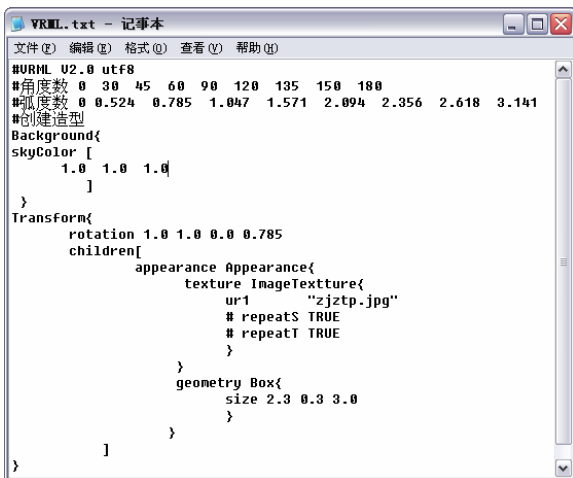


图 2-5 用文本编辑器编辑 VRML 源程序

2. 使用专用编辑器来编写VRML源程序

VrmlPad 编辑器是由 ParallelGraphics 公司开发的 VRML 开发工具。此外, VRML 开发工具还有 Cosmo World、Internet3D Space Buider、3Ds max 等。这里主要介绍 VrmlPad 编辑器。



图 2-6 VrmlPad 编辑器

VrmlPad 2.0 版专用编辑器可以从网上下载。下载完成后, 把 VrmlPad 专用编辑器软件复制到某个盘的根目录下(此软件是费安装的)。VrmlPad 2.0 版专用编辑器的图标如图 2-6 所示。

双击图 2-6 所示的图标, 可以启动 VrmlPad 专用编辑器。启动后的 VrmlPad 编辑器主要界面如图 2-7 所示。

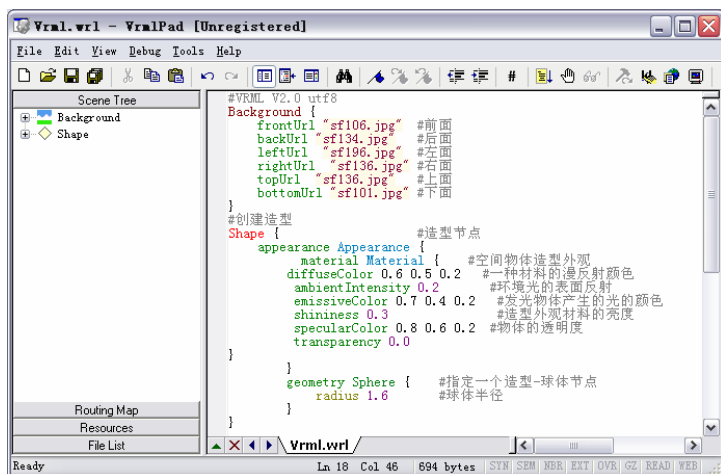


图 2-7 VrmlPad 编辑器的主界面

VrmlPad 编辑器的工作环境由标题栏、菜单栏、工具栏、功能窗口和编辑窗口等组成。

标题栏位于整个 VrmlPad 编辑器主界面的顶端。

菜单栏位于 VrmlPad 编辑器标题栏下方, 包括文件 (File)、编辑 (Edit)、视图 (View)、工具 (Tools) 和帮助 (Help) 菜单。

常用工具栏位于 VrmlPad 编辑器菜单栏的下方, 包括文件的打开 (Open)、保存 (Save)、剪切 (Cut)、粘贴 (Paste)、运行 (Run) 等常用快捷工具。

功能窗口位于 VrmlPad 编辑器的左边, 包括 File List (文件列表)、Resources (资源)、Scene Tree (场景树): 文件列表显示所有目录文件; 资源显示编辑代码.class 文件; 场景树显示 VRML 源程序中的节点树。

编辑窗口位于 VrmlPad 编辑器右部的空白处, 是编写 VRML 源程序的场所。每当创建一个新的 VRML 源程序时, 在编辑窗口的第一行自动显示 “#VRML V2.0 utf8”, 然后在此基础上可以编写 VRML 源程序。

VrmlPad 编辑器具有自动检错功能。在编写 VRML 源程序的过程中, 如果出现编写错误或语法错误等, 系统会自动进行检查, 对出现错误的地方, 用红色点下划线加以标注。

使用者可以根据自己的需要设置 VrmlPad 编辑器的各种功能。单击【Tools】(工具) 菜单下的【Options】(设置) 选项, 可弹出其选项设置界面, 如图 2-8 所示。

选项设置界面包含 General、Editor、Format、Node Folds、Tree、File List、Shortcut Keys 和 Preview 选项卡, 用以对编辑器进行有关的设置。

- (1) 在 General 选项卡中, 可对系统、保存选项、下载选项、VRML 扩展等进行设置。
- (2) 在 Editor、Format 选项卡中, 可对编辑器和文本显示进行设置, 包括字体、字号、前景、背景、颜色等设置。
- (3) 在 Node Folds 选项卡中, 可以对不敏感节点进行设置。

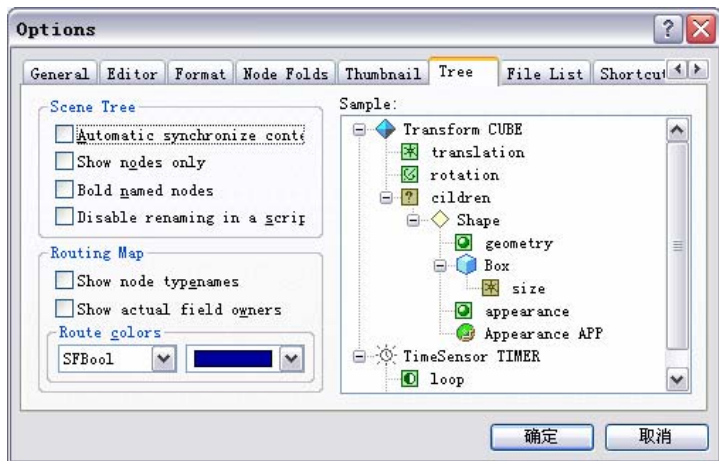


图 2-8 VrmIpad 编辑器的设置

- (4) 在 Tree 选项卡中, 可对 Scene Tree、Resources 进行设置。
- (5) 在 File List 选项卡中, 可对文件进行有关设置, 包括常规、鼠标选项、是否在新窗口中打开文件夹和起始目录等的设置。
- (6) 在 Shortcut Keys 选项卡中, 可对快捷键进行管理, 修改原始定义的快捷键及定义新的快捷键等。若想恢复对系统快捷键的默认值, 单击【Reset All】按钮。
- (7) 在 Preview 选项卡中, 一个区用来设置当从外部浏览器打开文件进行预览时浏览器的尺寸, 另一个区用来设置当前打开文件夹时浏览器的尺寸。

VrmIpad 编辑器支持在浏览器的预览。VrmIpad 编辑器采用树型结构显示场景, 具有高级查找、分色显示、自动纠错、取消操作、重复操作和使用书签等功能。VrmIpad 编辑器可以编辑本地和远程文件, 可以处理和执行其他语言编写的外部应用程序。VrmIpad 编辑器有强大的网络发布向导功能, 并提供文件列表功能, 便于用户编程和文件的目录管理。

把 VrmIpad 程序复制到当前目录下, 在系统中安装 Cortona3d 软件。此软件安装完成后, 在执行 VrmIpad 程序后, 便可在系统中生成三维虚拟现实画面。

2.3 几何造型

2.3.1 造型节点

在 VRML 中创建三维立体造型是最重要的、最基本性的工作, Shape 节点定义了立体造型的几何形状、尺寸、色彩、材质、纹理等外观特征, 学会使用 Shape 节点就掌握了 VRML 设计的基本内容。

Shape 节点的语法格式如下:

节点名称	域名称	域值	#域及域值类型
Shape{ }			
	appearance	NULL	#exposedField SFNode
	geometry	NULL	#exposedField SFNode

其中：

（1）appearance 域定义立体造型的外观特征，域值可以选择 NULL 为空域 Appearance{} 节点。如果选择 Appearance{} 节点，可以通过域值和子节点进一步设定包括材质、色彩、纹理等造型的外观属性；appearance 域的默认值为 NULL，表示立体造型外观为白色无光照效果。

（2）geometry 域定义立体造型的几何形状，域值可以选择 NULL 为空或 10 种造型节点，其中包括 4 种简单几何造型节点、5 种复杂几何造型节点、一种文本造型节点。geometry 域的默认值为 NULL，表示没有创建任何立体几何造型。

Shape 节点的层次结构见表 2-1。

表 2-1 Shape 节点的层次结构表

父节点	域	域值或子节点	域值类型
节点 Shape	外观特征域 appearance	NULL 空值	SFNode 单一节点
		Appearance 节点	
	几何形状域 geometry	NULL 空值	
		Box 节点	
		Shpere 节点	
		Cone 节点	
		Cylinder 节点	
		PointSet 节点	
		IndexedLineSet 节点	
		IndexedFaceSet 节点	
		Extrusion 节点	
		ElevationGrid 节点	
		Text 节点	

2.3.2 简单几何造型节点

1. Box节点

Box 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Box { }	size	2.0 2.0 2.0	#SFVec3f

其中，size 域的域值设定了三维立方体几何造型的尺寸大小。该域值为三维数组，第一个数值为立方体在 X 轴方向的边长尺寸，第二个数值为立方体在 Y 轴方向的边长尺寸，第三个数值

为立方体在 Z 轴方向的边长尺寸，每个数值之间用空格分隔，这 3 个数值必须大于零。size 域的默认值为 2.0 2.0 2.0，是一个以坐标原点为中心，长、宽、高都为 2.0 单位的立方体几何造型。

【例 2-1】 使用 Box 节点创建一个边长为 3×2×3 的长方体几何造型。在默认设置下，背景颜色为黑色。示例中，appearance 域值没有选择默认值，因为白色无光照效果无法显示长方体的棱线，其详细比较将在后面的外观设计中介绍到。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Box{
    size 3.0 2.0 3.0
  }
}
```

运行程序，效果如图 2-9 所示。

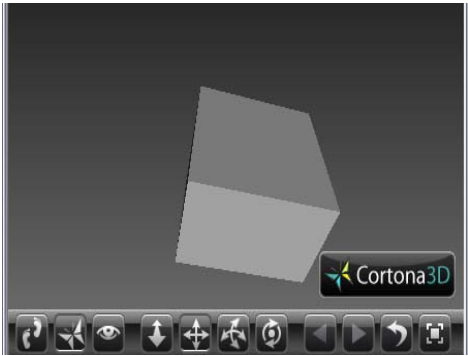


图 2-9 使用 Box 节点创建长方体

2. Cone节点

Cone 节点用来创建以坐标原点为中心、以 Y 轴为对称轴的圆锥体几何造型。

Cone 节点的语法格式如下：

节点名称	域名称	域值	#域及域值类型
Cone {			
	bottomRadius	1.0	#SFFloat
	height	2.0	#SFFloat
	side	TRUE	#SFBool
	bottom	TRUE	#SFBool
}			

其中：

(1) bottomRadius 域的域值设定圆锥体底面半径尺寸。该域值必须大于零，其默认值为 1.0

单位。

(2) **height** 域的域值设定圆锥体的高度。该域值必须大于零，其默认值为 2.0 单位，圆锥体的底面中心位于 Y 轴-1.0 处，顶端位于 Y 轴 1.0 处。

(3) **side** 域的域值设定是否创建圆锥体的锥面。如果该域值取 **TRUE** 则创建锥面，如果取 **FALSE** 则不创建锥面，即不显示圆锥体的锥面。默认值为 **TRUE**。

(4) **bottom** 域的域值设定是否创建圆锥体的底面。如果该域值取 **TRUE** 则创建底面，如果取 **FALSE** 则不创建底面，即不显示圆锥体的底面。默认值为 **TRUE**。

【例 2-2】 使用 **Cone** 节点创建一个锥体底面半径和锥体高度均为默认值的完整圆锥体几何造型。

其实现源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Cone {}
}
```

运行程序，效果如图 2-10 所示。

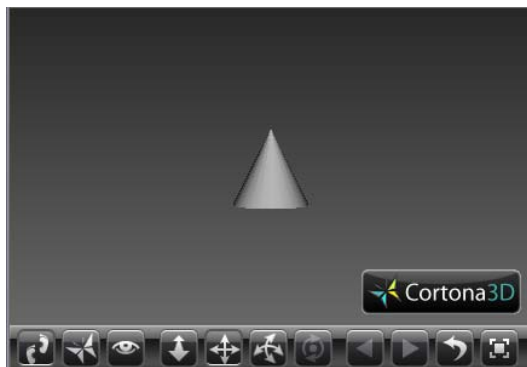


图 2-10 设置默认的完整圆锥体几何造型

【例 2-3】 创建一个底面半径为 3.0、锥体高度为 5.0 的无底几何造型。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Cone {
    bottomRadius 3.0
    height 5.0
  }
}
```

```
side TRUE
bottom FALSE
}
}
```

运行程序，效果如图 2-11 所示。

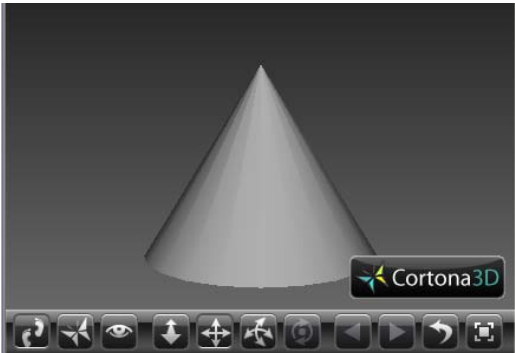


图 2-11 无底的圆锥体几何造型

3. Sphere节点

Sphere 节点用来创建以坐标原点为中心的球体几何造型。

Sphere 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Sphere {			
	radius	1.0	#SFFloat
}			

其中，radius 域的域值设定了以坐标原点为中心的三维球体的半径尺寸。该域值必须大于零，其默认值为 1.0 单位。

【例 2-4】 使用 Sphere 节点创建一个半径为 3.0 的球体几何造型。
其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Sphere {
    radius 3.0
  }
}
```

运行程序，效果如图 2-12 所示。

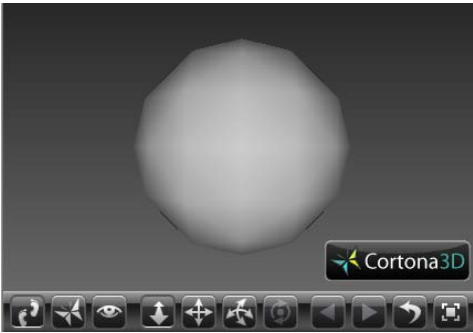


图 2-12 使用 Sphere 节点创建几何造型

4. Cylinder节点

Cylinder 节点用来创建以坐标原点为中心、以 Y 轴为对称轴的圆柱体几何造型。

Cylinder 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Cylinder {			
	radius	1.0	#SFFloat
	height	2.0	#SFFloat
	side	TRUE	#SFBool
	bottom	TRUE	#SFBool
	top	TRUE	#SFBool
}			

其中：

- (1) radius 域的域值设定圆柱体底面半径尺寸。该域值必须大于零，其默认值为 1.0 单位。
- (2) height 域的域值设定圆柱体的高度。该域值必须大于零，其默认值为 2.0 单位，圆柱体的底面中心位于 Y 轴-1.0 处，顶端位于 Y 轴 1.0 处。
- (3) side 域的域值设定是否创建圆柱体的曲面。如果该域值取 TRUE 则创建曲面，如果取 FALSE 则不创建曲面。默认值为 TRUE。
- (4) bottom 域的域值设定是否创建圆柱体的底面。如果该域值取 TRUE 则创建底面，如果取 FALSE 则不创建底面。默认值为 TRUE。
- (5) top 域的域值设定是否创建圆柱体的顶面。如果该域值取 TRUE 则创建顶面，如果取 FALSE 则不创建顶面。默认值为 TRUE。

圆柱体节点和圆锥体节点一样，可以通过对域值设定不显示底面、顶面或曲面。

【例 2-5】 使用 Cylinder 节点创建一个底面半径为 3.0、柱体高度为 5.0 的圆柱体几何造型。其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Cylinder {
```

```
radius 3.0
height 5.0
}
}
```

运行程序，效果如图 2-13 所示。

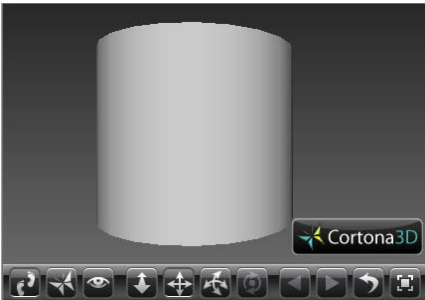


图 2-13 圆柱体几何造型

2.3.3 复杂几何造型节点

前面介绍了创建 4 种简单几何造型的方法，然而虚拟空间的内容是极其丰富的，VRML 造型设计的目的是实现对所有能够看见的、能够想象出来的物体和场景进行模拟，仅靠简单几何造型立方体、球体、圆锥体、圆柱和它们的组合是无法完成的。因此，下面将介绍几种创建复杂几何造型的节点。

1. 地表网格节点（ElevationGrid）

ElevationGrid 节点是在 VRML 中可以用来描述地表特征的节点，高山、丘陵和平地的地表不规则变化都可以用 ElevationGrid 节点来构建。ElevationGrid 节点先将某一个要描述的地表区域分割成很多网格，定义网格的个数，再定义网格的长和宽，最后定义网格的高度，就可以描述出想要表达的地表形状，所以，ElevationGrid 节点很适合应用于立体地形图的构建。

ElevationGrid 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Cylinder {			
	xDimension	0	#SFInt32
	xSpacing	0.0	#SFFloat
	zDimension	0.0	#SFInt32
	zSpacing	0	#SFFloat
	height	[]	#MFFloat
	color	NULL	#exposedField SFNode
	colorPerVertex	TRUE	#SFBool
	normal	NULL	#exposedField SFNode
	normalPerVertex	TRUE	#SFBool
	texCoord	NULL	#exposedField SFNode
	ccw	TRUE	#SFBool
	solid	TRUE	#SFBool
	creaseAngle	0.0	#SFFloat

```
set_height                                     #eventIn MFFloat
}
```

其中，网格分布示意图如图 2-14 所示。

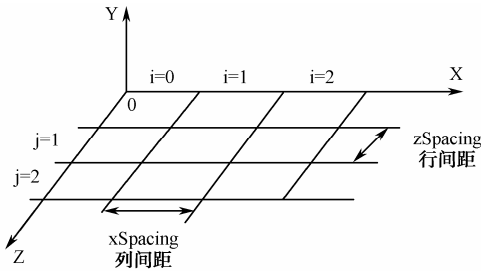


图 2-14 网格分布示意图

行与列的交点为网格点，作为设定 Y 方向网格高度的顶点。

i-X 方向列序号，取值范围由 0 至 (xDimension-1)。

j-Z 方向行序号，取值范围由 0 至 (zDimension-1)。

xSpacing-X 方向列间距；

zSpacing-Z 方向行间距。

(1) xDimension 和 zDimension 域的域值用于设定 X 和 Z 方向网格点的数量，而不是网格的数量。通常 X 和 Z 方向上网格的数量比网格点的数量都少一个。ElevationGrid 网格标高造型节点中，网格点的总数为 (xDimension×zDimension)，网格的总数为 [(xDimension-1) × (zDimension-1)]。这两个域值的默认值均为 0，表示没有网格。

(2) xSpacing 和 zSpacing 域的域值用于设定 X 方向的列间距、Z 方向的行间距。xSpacing 域值设定 X 方向的列间距，zSpacing 域值设定 Z 方向的行间距。域值为单值浮点数，默认值为 0.0，不能为负值。

(3) height 域的域值用于设定每个网格点 Y 方向上的高度。该域值为两维数列，按行排列，每一个高度值 height(yij)赋予对应的网格点。例如，第一行第一个高度值 height(yij)赋予位于坐标原点上的网格点，第一行第二个高度值赋予 X 轴 i=1 处的网格点，第一行第三个高度值赋予 X 轴上 i=2 处的网格点，以此类推。该域值的默认值为 height[]空数组，表示不创建网格。

(4) color、colorPerVertex 域的域值均用于为所创建的网格表面着色，将在后面色彩设计中介绍。

(5) normal 域用于网格设定法向量，此域包含一个 Normal 节点。

(6) normalPerVertex 域用于设定法向量是作用于每个网格点，还是作用于每个网格表面。域值为 TRUE 表示法向量作用于每个网格点，域值为 FALSE 时表示法向量作用于每个网格表面。

(7) texCoord 域的域值用于为所创建的网格表面粘贴纹理，将在后面介绍。

(8) ccw 域用于设定每个平面的边界线是按逆时针方向索引，还是按顺时针方向索引。域值为 TRUE 时，按逆时针方向索引，所画平面的正面面向屏幕。域值为 FALSE 时，按顺时针方向索引，所画平面的反面面向屏幕。该域的默认值为 TRUE。

(9) solid 域用于设定是否构建面集平面的背面。域值 TRUE 或 FALSE 表示平面组成的立体几何造型是否为实体。域值为 TRUE 时，表示组成的是实体造型，不需要构建面集平面的背

面，浏览器将跳过对背面的绘画，从而节省时间。域值为 **FALSE** 时，造型不实体，面集平面的表面和背面都能看出。该域的默认值为 **TRUE**。

(10) **creaseAngle** 域用于设定以弧度表示的转折角，可以使相邻两个平面间的边界看上去有平滑过滤。该域的默认值为 **0**，表示不进行平滑处理。

(11) **set_height** 为事件入口，用来接收发送过来的高度值，并改变原有高度列表。

【例 2-6】 在 X 方向创建了 6 个顶点，间距为 2；在 Z 方向创建 4 个顶点，间距为 2。构建的地表网格点，并进行平滑处理。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {material  Material {}}
  geometry ElevationGrid {
    xDimension 6
    zDimension 4
    xSpacing 2.0
    zSpacing 2.0
    ccw TRUE
    solid TRUE
    creaseAngle1.2    #对地表面进行平滑处理
    height [
      0 0.8 0.6 0.3 0.7 1.2      #j=0
      0 1.1 0.9 0.88 0.65 0      #j=1
      0 0.2 0.8 0.16 0.45 1.1    #j=2
      0.1 0.6 0.4 0.9 1.0 0.8    #j=3
    ]
  }
}
```

运行程序，效果如图 2-15 所示。



图 2-15 利用网格标高绘制地表面

【例 2-7】 利用 ElevationGrid 节点画出一条彩色。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
  }
  geometry ElevationGrid {
    xDimension 2
    zDimension 20
    xSpacing 1.0
    zSpacing 2.0
    solid FALSE
    creaseAngle0.7
    height [
      0.0 0.5 1.0 1.0 0.5
      0.0 -0.5 -1.0 -1.0 -0.5
      0.0 0.5 1.0 1.0 0.5
      0.0 -0.5 -1.0 -1.0 -0.5
      0.0 0.5 1.0 1.0 0.5
      0.0 -0.5 -1.0 -1.0 -0.5
      0.0 0.5 1.0 1.0 0.5
      0.0 -0.5 -1.0 -1.0 -0.5
    ]
  }
}
```

运行程序，效果如图 2-16 所示。

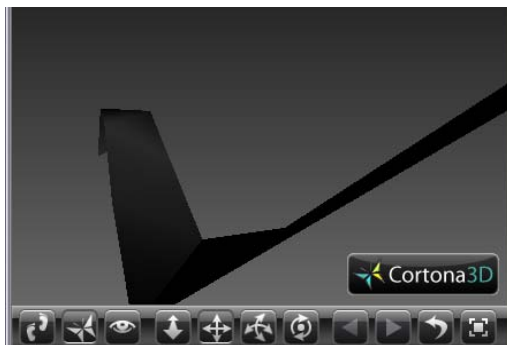


图 2-16 利用 ElevationGrid 节点画出一条彩色效果

2. 点集节点 (PointSet)

VRML 中点的造型是用 X、Y、Z 三维坐标定位，大小为 1 个像素的空间点。点集就是不同位置空间的集合。

PointSet 节点用于三维立体空间中，创建一系列位置不同的空间点的造型。它常常被用来

模拟天上闪烁的繁星和远处点点灯光。空间点的造型不受光源影响，不能粘贴纹理，不进行碰撞检测。

PointSet 节点的层次结构见表 2-2。

表 2-2 PointSet 节点的层次结构

父节点	域	域值或子节点	域值	域值类型
PointSet 节点	coord	NULL 空值		
	坐标域	Coordinate 节点	point[]点坐标列表	多个三维向量
	color	NULL 空值		
	颜色域	Color 节点	color[]颜色列表	多个三维向量

PointSet 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
PontSet {			
	coord	NULL	#exposedField SFNode
	color	NULL	#exposedField SFNode
}			

其中：

(1) coord 域的域值设定离散点的三维坐标位置。该域值有两种选择：NULL 或 Coordinate 节点。该域值默认值为 NULL，表示不创建任何点造型。如果创建离散点造型，该域值必须选择 Coordinate 节点，该节点包含了用来进行点定位的坐标。

Corrdinate 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Corrdinate {			
	point	[]	#exposedField SFNode
}			

point 域的域值提供了一张三维坐标列表，用来设定一个或一组空间点的 X、Y、Z 坐标。每个点的坐标都包含 3 个浮点数，分别表示在 X、Y、Z 方向上与坐标原点的距离，3 个数值均不允许缺省，没有距离用 0 填充，3 个数值之间用空格分开，如 point[2 0 0, 5 3 1]不能表示为 point[2, 5 3 1]，否则浏览器将会报错。各个点的坐标之间用“,”号分开。该域值的默认值为空，即 point[]，表示不创建任何点造型。

(2) color 域的域值通常选择 Color 颜色节点，用来设定每个空间点的颜色。该域值的默认值为 NULL，表示不对空间点着色。Color 节点的设定方法将在后面色彩设计中详细介绍。

【例 2-8】 利用 PointSet 节点绘制点集。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  geometry PointSet {
    color Color {
```

```

        color [
            1.0 0.0 0.0, #RED
            0.0 1.0 0.0, #GREEN
            0.0 0.0 1.0, #BLUE
            0.0 0.0 0.0, #BLACK
            1.0 1.0 1.0, #WHITE
            1.0 1.0 0.0, #YELLOW
            0.0 1.0 1.0, #GREENBLUE
            1.0 0.0 1.0 #PURPLE
        ]
    }

    coord Coordinate{
        point[
            1.0 1.0 1.0,
            1.0 -1.0 1.0,
            -1.0 -1.0 1.0,
            -1.0 1.0 1.0,
            1.0 1.0 -1.0,
            1.0 -1.0 -1.0,
            -1.0 1.0 -1.0
        ]
    }
}
}
}

```

运行程序，效果如图 2-17 所示。

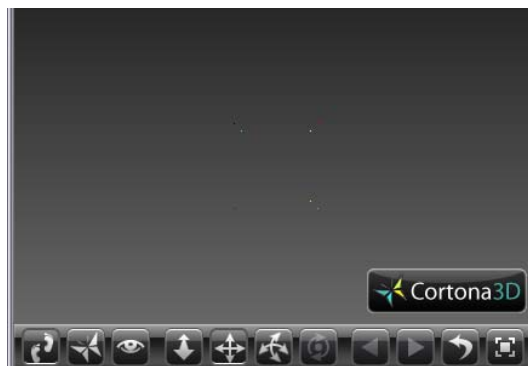


图 2-17 绘制的点集

3. 线集合节点 (IndexedLineSet)

VRML 中直线造型是由两个端点连接而成的，创建一条直线必须设定线的起点和终点。VRML 中折线造型由多个点连接而成，除了起点、终点外还有中间转折点。因此，创建直线、折线造型需要做两项工作：一是设定连接点的坐标；二是设定连接点的顺序。为了方便设定连接顺序，每一个连接点的坐标都有一个默认的索引号，只要指定索引号的连接次序，就可以完成直线或折线造型的创建工作。线集就是不同线段的集合。

IndexedLineSet 节点用于在三维空间中创建由线段组合而成的各种开放或封闭的立体几何造型。线集节点创建的造型不受光源影响，不能粘贴纹理，不进行碰撞检测。

IndexedLineSet 节点的层次结构见表 2-3。

表 2-3 IndexedLineSet 节点的层次结构表

父节点	域	域值或子节点	域值	域值类型
IndexedLineSet 节点	coord	NULL 空值		
	坐标域	Coordinate 节点	point[]点坐标列表	多个三维向量
	coordIndex	NULL 空值		多个 32 位整数
	坐标索引域	[]折线路径的坐标索引列表		
	color	NULL 空值		
	颜色域	Color 节点	color[]颜色列表	多个三维向量
	colorIndex	[]颜色索引列表		多个 32 位整数
	颜色索引域			
	colorPerVertex	TRUE 或 FALSE		
	着色方式域			

IndexedLineSet 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
IndexedLineSet {			
	coord	NULL	#exposedField SFNode
	coordIndex	[]	#MFIInt32
	color	NULL	#exposedField SFNode
	colorIndex	[]	#MFIInt32
	colorPerVertex	TRUE	#SFBool
	set_coordIndex		#eventIn MFIInt32
	set_colordIndex		#eventIn MFIInt32
	}		

其中：

(1) coord 域的域值设定线段连接点的三维坐标位置。与 PointSet 节点完全一样，该域值有两种选择：NULL 或 Coordinate 节点。该域值默认值为 NULL，表示为空的坐标列表，不创建任何线的造型。如果创建线的几何造型，该域值必须选择 Coordinate 节点，该节点包含了用来构建线段的全部连接点的定位坐标。Coordinate 节点语法格式与点集节点中出现的 Coordinate 节点完全相同，故不再重复。值得强调的是：point 域的域值提供了一张连接点的三维坐标列表，同时为每一个连接点按照出现的顺序建立了默认的索引号。也就是说，point 列表中第一组坐标的索引号为 0，第二组坐标的索引号为 1，第三组坐标的索引号为 2，以此类推。

(2) coordIndex 域的域值设定一条或多条线段连接路径的索引列表，其中的索引号与 point 域值相对应。每一个索引号对应一个连接点的坐标位置，索引号之间用逗号分隔，线的创建方式是按照索引号依次连接对应的点。索引号为-1 表示当前连接的线段已经结束，下一条线段的连接即将开始，列表中最后一条线段结束时可以不标索引号-1。该域值的默认值为空，表示不创建任何线造型。

(3) color、colorIndex、colorPerVertex 域的域值均用于为所创建的线段着色，关于这一点将在后面颜色设计中介绍。

(4) set_coordIndex 为入事件用以设置新的坐标索引列表；set_colorIndex 为入事件用于设置新的颜色列表。详细将在后面章节中介绍。

【例 2-9】 使用 IndexedLineSet 节点，将例 2-8 中 5 个顶点连接成的四棱锥的白色轮廓线造型。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {
      emissiveColor 1.0 1.0 1.0
    }
  }
  geometry IndexedLineSet {
    coord Coordinate {
      point [
        0.0 1.0 0.0,      #索引号为 0
        1.0 0.0 1.0,      #索引号为 1
        1.0 0.0 -1.0,     #索引号为 2
        -1.0 0.0 -1.0,    #索引号为 3
        -1.0 0.0 1.0      #索引号为 4
      ]
    }
    coordIndex [
      #side
      0 1 -1,
      0 2 -1,
      0 3 -1,
      0 4 -1,
      #bottom
      1 2 3 4 1 -1
    ]
  }
}
```

运行程序，效果如图 2-18 所示。

4. 面集节点 (IndexedFaceSet)

VRML 中的平面造型是由封闭折线勾画出平面的边界然后填充而成。平面边界线的建立过程与线集节点创建封闭折线造型完全相同，首先设定连接点的坐标（至少 3 个点），然后按照点坐标的索引号设定点连接的顺序，形成平面封闭的边界线。浏览器自动将边界包围的部分填充并显示出来，构建完成一个平面的造型。面集就是不同平面的集合。

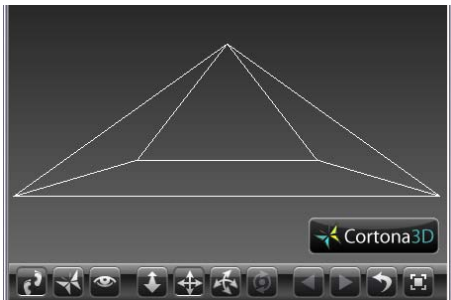


图 2-18 白色四棱锥轮廓线造型

IndexedFaceSet 节点用于在三维空间中创建各种由平面组合而成的不规则的立体几何造型。**IndexedFaceSet** 节点不仅可以创建平面几何造型，也可以创建实体几何造型。**IndexedFaceSet** 节点是一个应用非常广泛的造型节点，几乎可以用来创建任意立体几何造型。

IndexedFaceSet 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
IndexedLineSet {	coord	NULL	#exposedField SFNode
	coordIndex	[]	#MFINt32
	texCoord	NULL	#exposedField SFNode
	texCoordIndex	[]	#MFINt32
	color	NULL	#exposedField SFNode
	colorIndex	[]	#MFINt32
	colorPerVertex	TRUE	#SFBool
	normal	NULL	#exposedField SFNode
	normalIndex	[]	#MFINt32
	normalPerVertex	TRUE	#SFBool
	ccw	TRUE	#SFBool
	convex	TRUE	#SFBool
	solid	TRUE	#SFBool
	creaseAngle	0.0	#SFFloat
	set_coordIndex		#eventIn MFINt32
	set_texCoordIndex		#eventIn MFINt32
	set_colorIndex		#eventIn MFINt32
	set_normalIndex		
	}		

其中：

- (1) **coord** 域的域值设定平面边界线连接点的三维坐标位置。与 **PointSet** 节点完全一样，该域值有两种选择：**NULL** 或 **Coordinate** 节点。该域值默认值为 **NULL**，表示为空的坐标列表，不创建任何平面的造型。如果创建平面的几何造型，该域值必须选择 **Coordinate** 节点，该节点包含了用来构建平面边界线全部连接点的定位坐标。**Coordinate** 节点语法格式及下一层 **point** 域的域值说明与线集节点中出现的情况完全相同，故不再重复。
- (2) **coordIndex** 域的域值设定一条或多条平面边界线连接路径的索引列表，其中的索引号与 **point** 域值相对应。每一个索引号对应一个连接点的坐标位置，索引号之间用逗号分隔。平

面边界线的创建方式是按照索引号依次连接对应的点，当索引号为-1 时，表示当前连接的平面已经结束，下一个平面的连接即将开始，浏览器会自动将最后一个坐标和第一个坐标连接起来，形成封闭的平面边界线。因此，在坐标索引列表中不需要再最后重复第一个索引号。列表中最后一个平面结束时可以不标索引号-1。该域值的默认值为空，表示不创建任何平面。

(3) **texCoord**、**texCoordIndex** 域的域值均用于为所创建的平面粘贴纹理，将在后面纹理包装介绍。

(4) **color**、**colorIndex**、**colorPerVertex** 域的域值均用于为所创建的平面着色，将在后面色彩设计中介绍。

(5) **normal** 域用于为平面设定法向量，此域包含一个 **Normal** 节点。设定法向量，不是为确定平面的位置，而是用来控制平面的光线明暗效果。因为设定法向量会使文件增大，所以除非需要特殊明暗效果时才会使用 **Normal** 节点，通常情况都使用浏览器自动产生的默认法向量。

(6) **normalIndex** 域的域值为一个法向量索引列表，指向 **Normal** 节点，用于设定所用的法向量。

(7) **normalPerVertex** 域用于设定法向量是作用于每个平面的连接点（顶点），还是作用于每个平面。域值为 **TRUE** 时表示法向量作用于每个平面的连接点（顶点），域值为 **FALSE** 时表示法向量作用于每个平面。

(8) **ccw** 域用于设定每个平面的边界线是按逆时针方向索引，还是按顺时针方向索引。域值为 **TRUE** 时，按逆时针方向索引，所画平面的正面面向屏幕。域值为 **FALSE** 时，按顺时针方向索引，所画平面的反面面向屏幕。该域的默认值为 **TRUE**。

(9) **convex** 域的域值设定挤出造型中的面是否为凸面。当域值为 **TRUE** 时，说明几何体为一个完全的凸面体；当域值为 **FALSE** 时，浏览器将对凹面进行优化处理。该域的默认值为 **TRUE**。

(10) **solid** 域用于设定是否构建面集平面的背面。域值 **TRUE** 或 **FALSE** 表示平面组成的立体几何造型是否为实体。域值为 **TRUE** 时，表示组成的是实体造型，不需要构建面集平面的背面，浏览器将跳过对背面的绘画，从而节省时间。域值为 **FALSE** 时，造型不实体，面集平面的表面和背面都能看出。该域的默认值为 **TRUE**。

(11) **creaseAngle** 域用于设定以弧度表示的转折角，可以使相邻两个平面间的边界看上去有平滑过滤。该域的默认值为 0，表示不进行平滑处理。

(12) **creaseAngle** 域用于设定以弧度表示的转折角，可以使相邻两个平面间的边界看上去有平滑过渡。该域的默认值为 0，表示不进行平滑处理。

(13) **set_coordIndex**、**set_texCoordIndex**、**set_colorIndex**、**set_normalIndex** 是面集节点的 4 个入事件，分别用于域值的动态设置。将在后面章节介绍。

【例 2-10】 使用 **IndexedFaceSet** 节点，将例 2-9 中线集节点创建的轮廓线改造成由 5 个平面组合的四棱锥体造型。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
```

```
Shape {  
  appearance Appearance {  
    material Material {}  
  }
```

```
}
geometry IndexedFaceSet {
  solid FALSE
  coord Coordinate {
    point [
      0.0 1.0 0.0,      #索引 0
      1.0 0.0 1.0,      #索引 1
      1.0 0.0 -1.0,     #索引 2
      -1.0 0.0 -1.0,    #索引 3
      -1.0 0.0 1.0      #索引 4
    ]
  }
  coordIndex [
    #side
    0 1 2 -1,
    0 2 3 -1,
    0 3 4 -1,
    0 4 1 -1,
    #bottom
    1 2 3 4 -1
  ]
}
}
```

运行程序，效果如图 2-19 所示。

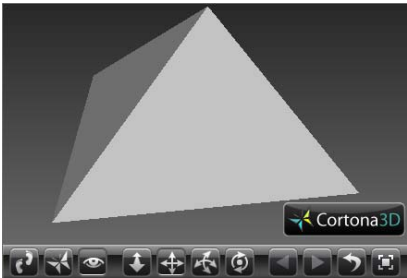


图 2-19 四棱锥体造型

5. 文件造型（Text）

在虚拟场景中，除了基本几何造型以外，文本也是不可少的。在 VRML 中，文本也是一种造型，通常用节点 Text{} 创建它。节点 Text 也是 geometry 域的一个域值，用来创建文本造型。Text 文本造型有 4 个域，分别是 string、fontStyle、maxExtent、length。其中 fontStyle 是节点型文本外观域。

Text 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Text {	string	[]	#MFString

	fontStyle	NULL	#SFNode
	maxExtent	0.0	#SFFloat
	length	[]	#MFFloat
	}		

其中：

(1) string 指定要显示的文本字符串。它是多域值字符串型。意味着可以添加多个字符串，每个字符串用双引号，并且单独占一行。字符串采用 UTF-8 编码，默认值是空字符串。注意：若需显示中文内容，则必须用专门工具转换编码格式为 UTF-8。

(2) fontStyle 确定文本字符串的相关特性。这个域值是一个同名的节点 fontStyle，该节点说明了如何绘制文本。fontStyle 域的默认值是 NULL，即不设置文本特征。

(3) maxExtent 确定文本的任意一行在主要方向上的最大范围。其值必须大于等于 0。主要方向由 fontStyle 节点的 horizontal 域来确定，如果该域值是 TRUE，则主要方向是水平方向，否则是垂直方向。maxExtent 域值的默认值为 0,表示字符串可为任意长度。

(4) length 设置单个文本串的长度，0 表示可以任意长度。

【例 2-11】 创建一个木板，并在上面写几个字母。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

#创建一个木板
Shape {                                     #Shape 模型节点
  appearance Appearance {
    material Material {                   #空间物体造型外观
      diffuseColor 1.0 1.0 1.0          #一种材料的漫反射颜色
      transparency 0.8                  #物体的透明度
    }
  }
  geometry Box {
    size 20 8 0.8                        #透明度
  }
}

#创建文本造型
Shape {                                     #Shape 模型节点
  appearance Appearance {
    material Material {                   #空间物体造型外观
      diffuseColor 1.0 0.0 0.0          #一种材料的漫反射颜色
    }
  }
  geometry Text {
    string [
      "Hellow",                          #不同的行用逗号隔开
      "It is VMRL"
    ]
    fontStyle FontStyle {
```

```
family "typewriter"      #字符集（是一种等宽字体）
size 2.0
style "BOLDITALIC"
justify [
    "MIDDLE"              #中间
]
}
}
}
```

运行程序，效果如图 2-20 所示。



图 2-20 利用 Text 节点与 Box 节点创建的字牌

6. 挤出造型节点（Extrusion）

挤出造型节点的设计思想源于生活实例。试想用金属材料制作一个三角形的模型边框，把橡皮泥从模型边框的一端向下挤压，边框的另一端将产生三棱柱的立体造型。如果将挤压出来的三棱柱沿着折线轨迹索引，就会产生一段弯折的三棱柱造型；如果将挤压出的三棱柱沿着方形轨迹牵引，就可以创建封闭的三棱柱边框。Extrusion 节点，首先利用 crossSection 和 XOZ 平面设定挤出造型的二维截面轮廓线，相当于生活中制作模型边框，然后用 spine 域在三维空间设定挤出造型的牵引轨迹，设定轨迹可以是直线、曲线、折线、开放的或封闭的线段。因为 Extrusion 节点能够很方便地创建各种各样的柱体、环状柱体、曲面、环状曲面等立体几何造型，所以 Extrusion 节点是使用率很高、在造型中占有重要位置的节点。

几种常见的挤出造型截面轮廓线如图 2-21 所示。

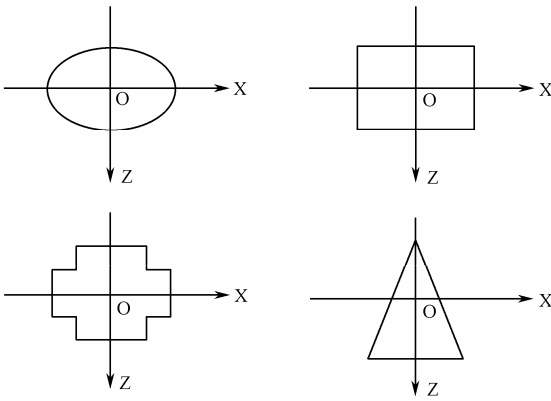


图 2-21 常见的挤出造型截面轮廓线

Extrusion 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Extrusion {	crossSection	[1 1,1 -1,-1 -1,1 1 1]	#MFVec2f
	spine	[0 0 0,0 1 0]	#MFVec2f
	scale	1 1	#MFVec2f
	orientation	0 0 1 0	#MFRotation
	beginCap	TRUE	#SFBool
	endCap	TRUE	#SFBool
	ccw	TRUE	#SFBool
	convex	TRUE	#SFBool
	solid	TRUE	#SFBool
	creaseAngle	0.0	#SFFloat
	set_crossSection		#eventIn MFVec2f
	set_spine		#eventIn MFVec2f
	set_scale		#eventIn MFVec2f
	}		

其中：

- (1) crossSection 域的域值在 XOZ 平面设定一系列的二维坐标点，依次连接这些坐标点得到一条封闭或开放的折线，这就是挤出造型的截面轮廓线。该域值中坐标的第一个值是 X 坐标、第二值是 Z 坐标。crossSection 域的默认值是一个边长为 2 的正方形轮廓线。
- (2) spine 域的域值设定一系列的三维坐标点，依次连接这些坐标点在立体空间中得到一条封闭或开放的折线，这就是挤出造型在 VRML 空间中被牵引的运动轨迹，称作龙骨线。挤出造型就是二维截面沿龙骨线牵引成型的。spine 域的默认值是一条沿 Y 轴指向上方的直线轨迹。
- (3) scale 域的域值设定一系列截面轮廓线的缩放比较系数对。每对比例系数的第一个值指定 X 方向的缩放比例，第二个值指定 Z 方向的缩放比例。缩放比例系数对与 spine 域设定的龙骨线坐标点一一顺序对应，缩放比例系数对的数目与龙骨线坐标点的数目相同。如果该值只设定一对比例系数，则每个龙骨线坐标点对应的截面轮廓线均使用相同的比例系数缩放。该域值必须大于等于 0.0。scale 域的默认值为 (1.0 1.0)，表示在 X 和 Z 方向上不进行任何缩放。
- (4) orientation 域的域值设定沿龙骨线坐标点的挤出造型的旋转参数。每一组参数的前三个数值指定一个旋转轴，第四个数值指定旋转轴旋转的角度，角度为弧度值。该域值旋转参数的数目与龙骨坐标点的数目相同，每组参数与龙骨线坐标点一一顺序对应。如果该域只有一组旋转参数，则每个龙骨线坐标点处对应的截面轮廓均使用相同的旋转参数变换。注意，这个旋转变换不在节点的局部坐标系上定义，而是对应于每个龙骨线坐标点处的特殊点坐标系上进行。orientation 域的默认为 (0.0 0.0 1.0 0.0)，表示没有任何旋转变换。
- (5) beginCap 和 endCap 域的域值用于设定挤出造型是否创建起始端和终止端面。起始端面对应于龙骨线的第一个坐标点，终止端面对于龙骨线的最后一个坐标点。域值为 TRUE 时，表示创建端面，域值为 FALSE 时表示不创建端面。beginCap 和 endCap 域的默认值均为 TRUE，即创建端面。
- (6) ccw 域的域值设定 Extrusion 节点自动生成的几何面法线与截面轮廓线走向关系。当域值为 TRUE 时，轮廓线走向与法线方向成右手关系；当域值为 FALSE 时，轮廓线走向与法

线方向成左手关系。该域的默认值为 **TRUE**，说明轮廓线逆时针连接的表面为正面。只有逆着法线观察，才能看到几何面。

(7) **convex** 域的域值设定挤出造型中的面是否为凸面。当域值为 **TRUE** 时，说明几何体为一个完全的凸面体；当域值为 **FALSE** 时，浏览器将对凹面进行优化处理。该域的默认值为 **TRUE**。

(8) **solid** 域的域值设定挤出造型是否构成实体。当域值为 **TRUE** 时，挤出造型构成实体，只有逆着法线方向观察才能看到造型，也就是只能看到外表面；当域值为 **FALSE** 时，挤出造型不构成实体，无论逆着还是顺着法线方向观察都能看到造型，也就是内外表面都能看到。该域的默认值为 **TRUE**。

(9) **creaseAngle** 域的域值设定一个弧度表示的褶痕角的阈值。如果挤出造型两个相邻面的法线夹角小于所设定的阈值，浏览器将对相邻表面做平滑处理，使相接的棱线变得模糊，相接处过渡圆滑。如果法线夹角大于所设定的阈值，则不做平滑处理，相接的棱线保持原状。该域值大于等于 0.0，默认值为 0.0 表示不进行平滑处理。

(10) **set_crossSection**、**set_spine**、**set_scale** 均为入事件，用于接收和改变挤出造型的参数，应用的详细将在后面章节说明。

使用 **Extrusion** 挤出造型节点创建的造型，主要基于 **crossSection** 域设定的截面轮廓线和 **spine** 域设定的龙骨线。截面轮廓线由 **crossSection** 域设定的坐标点顺序连接而成，可以是封闭的也可以是开放的，首先尾坐标点相同时为封闭的轮廓，首尾坐标点不同时为开放的轮廓线。当轮廓线闭合时，沿龙骨线挤出的造型为柱体；当轮廓线开口时，沿龙骨线挤出的造型为曲面。同理，龙骨线由 **spine** 域设定的三维坐标点顺序连接而成，可以是直线、曲线、折线，可以是封闭的也可以是开放的，龙骨线是挤出造型的中心线。

【例 2-12】 利用 **Extrusion** 函数绘制挤压龙骨线。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Extrusion {
    creaseAngle 0.2
    endCap FALSE
    beginCap FALSE
    solid FALSE
    crossSection [
      1.00 0.00, 0.92 0.38,
      0.71 0.71, 0.38 0.92,
      0.00 1.00, -0.38 0.92,
      -0.71 0.71, -0.38 0.92,
      -1.00 0.00
    ]
  }
  spine [
```

```

1.00 4.00 0.00,0.922 3.75 0.38,
0.71 3.50 0.71,0.38 3.25 0.92,
0.00 3.00 1.00,-0.38 2.75 0.92,
-0.71 2.50 0.71,-0.92 2.25 0.38,
-1.00 2.00 0.00,-0.92 1.75 -0.38,
-0.71 1.50 -0.71,-0.38 1.25 -0.92,
0.00 1.00 -1.00,0.38 0.75 -0.92,
0.71 0.50 -0.71,0.92 0.25 -0.38,
1.00 0.00 0.00,0.92 -0.25 0.38,
0.71 -0.50 0.71,0.38 -0.75 0.92,
0.00 -1.00 1.00,-0.38 -1.25 0.92,
-0.71 -1.50 0.71,-0.92 -1.75 0.38,
-1.00 -2.00 0.00,-0.92 -2.25 -0.38,
-0.71 -1.50 0.75,-0.92 -1.75 0.38,
-1.00 -2.00 0.00,-0.92 -2.25 -0.38,
-0.71 -2.50 -0.71,-0.38 -2.75 -0.92,
0.00 -3.00 -1.00,0.38 -3.25 -0.92,
0.71 -3.50 -0.71,0.92 03.75 -0.38,
1.00 -4.00 0.00
]
}
}

```

运行程序，效果如图 2-22 所示。



图 2-22 挤出造型效果

【例 2-13】 在蓝绿色背景下，创建一个红色多面锥体的挤出造型。背景颜色及造型外观设计方法将在后面相关章节中介绍。本例采用十字形截面轮廓线，挤压轨迹为沿 X 轴方向牵引的直线。利用 scale 域的域值，将例中龙骨线起点对应的截面轮廓按照（0 0）的比例缩小。龙骨线终点对应的截面轮廓按照（2 2）的比例放大。

其实现的源程序代码如下：

```

#VRML V2.0 utf8

Background {                               #背景节点
  skyColor [
    0.0 0.6 0.8
  ]
}

```

```
}
Shape {
  appearance Appearance {      #外观节点
    material Material {        #材质节点
      diffuseColor 0.8 0.2 0.2
      ambientIntensity 0.1
      specularColor 0.8 0.8 0.8
      shininess 0.15
    }
  }
}
```

运行程序，效果如图 2-23 所示。

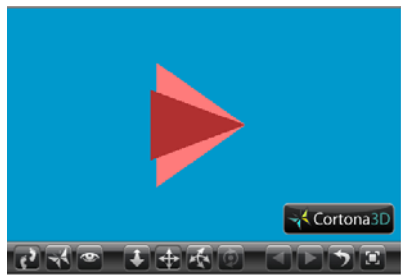


图 2-23 设定旋转比例的挤出造型

【例 2-14】 通过设定 **orientation** 域的域值，将上例中多面锥体造型旋转成抽象的花朵造型。为了方便设置旋转参数，本例的龙骨线与上例相比作了两点改动，一是缩短了 1/3，二是在起点和终点之间增加了 3 个坐标点。**orientation** 域设定了 5 个旋转角度与龙骨线上的 5 个坐标点相对应。

其实现的 MATLAB 程序代码如下：

```
#VRML V2.0 utf8

Background {                                #背景节点
  skyColor [
    0.0 0.6 0.8
  ]
}

Shape {
  appearance Appearance {                  #外观节点
    material Material {                   #材质节点
      diffuseColor 0.8 0.2 0.2
      ambientIntensity 0.1
      specularColor 0.8 0.8 0.8
      shininess 0.15
    }
  }
}

geometry Extrusion {                      #挤压造型节点
  crossSection [                          #十字形截面轮廓线
```

```

0.5 1.0,0.5 0.5,1.0 0.5,
1.0 -0.5,0.5 -0.5,0.5 -1.0,
-0.5 -1.0,-0.5 -0.5,-1.0 -0.5,
-1.0 0.5,-0.5 0.5,-0.5 1.0,
0.5 1.0
]
spine [                                #直线轨迹的龙骨线
    1.0 0.0 0.0
    0.5 0.0 0.0
    0.0 0.0 0.0
    -0.5 0.0 0.0
    -1.0 0.0 0.0
]
scale [                                #缩放比例系数
    0.0 0.0
    0.5 0.5
    1.0 1.0
    1.5 1.5
    2.0 2.0
]
orientation [                          #挤压造型旋转参数
    0 1 0 0                            #旋转 0°
    0 1 0 1.57                         #旋转 90°
    0 1 0 3.14                         #旋转 180°
    0 1 0 4.71                         #旋转 270°
    0 1 0 6.28                         #旋转 360°
]
solid FALSE
creaseAngle 1.57
}
}

```

运行程序，效果如图 2-24 所示。

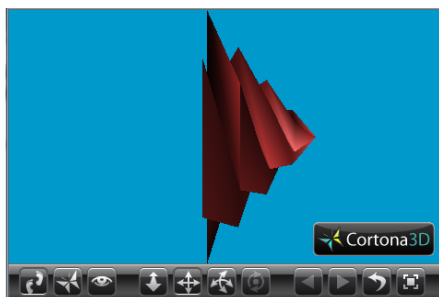


图 2-24 设定旋转参数的挤出造型

7. 文本外观节点 (FontStyle)

FontStyle 节点是 Text 文本节点中 fontStyle 域的域值，用于设定文本造型的外观特征、文

本的语种、字符的大小和形状、字体的风格以及文本的对齐方式、排列方式、行间距或列间距等。

FontStyle 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Text {	family	"SERIF"	#MFString
	size	1.0	#SFFloat
	spacing	1.0	#SFFloat
	style	"PLAIN"	#MFString
	horizontal	TRUE	#SFBool
	justify	["BEGIN"]	#SFString
	leftToRight	TRUE	#SFBool
	topToBottom	TRUE	#SFBool
	language	""	#SFString
}			

其中：

(1) family 域的域值设定 Text 节点创建的文本造型使用字符集，字符集包括 3 种：“SERIF”、“SANS”和“TYPEWRITER”。浏览器将在设定的字符集中选择特定的字体进行渲染。serif 类是一种细线变宽字体，浏览器选择 Times New Roman 字体；sans 类是一种简单变宽字体，浏览器选择 Helvetica 字体；typewriter 类是一种等宽字体，浏览器选择 Coutier 字体。该域值的默认值为“SERIF”的字符集。

(2) size 域的域值设定所创建的文本字符的尺寸，水平排列的文本设定其高度，垂直排列的文本设定其宽度。该域值的默认值为 1.0。

(3) spacing 域的域值设定所创建的文本字符的行间距。当文本水平排列时，该域值为行间距；当文本垂直排列时，该域值为列间距。该域值的默认为 1.0。

(4) style 域的域值设定所创建的文本的字体风格。该域值包括 4 种：“PLAIN”通常体、“BOLD”粗体、“ITALIC”斜体和“BOLDITALIC”加粗的斜体。该域值的默认值为“PLAIN”，即通常体风格。

(5) horizontal 域的域值设定文本造型是水平方向排列还是垂直方向排列。该域值的默认值为 TRUE，表示文本造型水平方向排列。

(6) justify 域的域值设定文本造型中字符块的对齐方式，这是相对 X 轴或 Y 轴来说的。该域为含有一个或两个域值的列表，当含有两个域值时，需使用空格或逗号分隔，也可分写两行。其中，第一个域值设置主对齐方式，当 horizontal 域的域值为 TRUE 时，即相对于 Y 轴字符块的水平方向对齐方式；第二个域值设置次对齐式，即相对于 X 轴字符块的垂直方向对齐方式。域值类型有 4 种：“BEGIN”左对齐、“MIDDLE”居中对齐、“END”右对齐和“FIRST”。当 horizontal 域的域值为 FALSE 时，主对齐方式为垂直方向，次对齐方式为水平方向。该域值的默认值为“BEGIN”，指的是主对齐方式，如果第二个域值未明确指定，则次对齐方式按“FIRST”处理。

(7) leftToRight 域的域值设定文本造型中的文本块或字符是从左到右排列还是从右到左排列。对于水平排列的文本造型，它表示前一个字符是否在后一个字符的左边；对于垂直排列的文本造型，它表示前一列字符是否在后一列字符的左边。该域值的默认值为 TRUE，表示文本

块或字符沿 X 方向从左到右排列。

（8）**topToBottom** 域的域值设定文本造型中的文本块或字符是从上到下排列还是从下到上排列。对于垂直排列的文本造型，它表示前一个字符是否在后一个字符的上边；对于水平排列的文本造型，它表示前一行字符是否在后一行字符的上边。该域值的默认值为 **TRUE**，表示文本块或字符沿 Y 轴方向从上到下排列。

（9）**language** 域的域值设定文本造型使用的语言，如英语、德语等。该域值以两字符语言代码表示，对于有不同语言编码的地区则在语言代码后加连字符和地区编码。由于很多语言暂时还不能在 VRML 浏览器中显示，一般情况下最好使用英语。该域值的默认值为英语，所以，通常该域值不再另外设置。表 2-4 列出 **language** 的可选域值。

表 2-4 language 可选域值列表

language 域值	选用语言	language 域值	选用语言
ar	阿拉伯语	hi	印度语
de	德语	jp	日语
de_DE	德语（德国）	ru	俄语
de_CH	德语（瑞士）	sa	梵语
en	英语	zh	中文
en_US	英语（英国）	zh_WT	中文（繁体）
en_GB	英语（英国）	zh_CN	中文（简体）

【例 2-15】 文本造型字体设定，**family** 域的域值设定，可以将文本造型的字体设置成“**SERIF**”、“**SANS**”和“**TYPEWRITER**”。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {
    }
  }
  geometry Text {
    string [
      "serif"
    ]
    fontStyle FontStyle {
      family "SERIF"
    }
  }
}

Transform {
  translation 0 -1 0
  children [
    Shape {
```

```

        appearance Appearance {
            material Material {
            }
        }
    }
    geometry Text {
        string [
            "sans"
        ]
        fontStyle FontStyle {
            family "SANS"
        }
    }
}
]
}
Transform {
translation 0 -2 0
children [
    Shape {
        appearance Appearance {
            material Material {
            }
        }
        geometry Text {
            string [
                "typewriter"
            ]
            fontStyle FontStyle {
                family "TYPEWRITER"
            }
        }
    }
]
}
}

```

运行程序，效果如图 2-25 所示。



图 2-25 文本造型字体设定

【例 2-16】 FontStyle 节点域值的综合设置。本例中，size 域值设定为 2，表示字符的尺寸比默认值放大 2 倍；style 域值设定为“BOLDITALIC”，表示字体风格为加粗斜体；justify 域值设定为[“MIDDLE” “MIDDLE”]，表示文本造型水平和垂直方向均居中对齐。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
Shape {
  appearance Appearance {
    material Material {
    }
  }
  geometry Text {
    string [
      "Welcome" "Color" "Red Clock"
    ]
    fontStyle FontStyle {
      size 2
      style "BOLDITALIC"
      justify [
        "MIDDLE" "MIDDLE"
      ]
    }
  }
}
```

运行程序，效果如图 2-26 所示。



图 2-26 加粗放大居中的文本造型

2.4 造型外观设计

立体几何造型的外观设计是指，为造型添加颜色；设置有关材质的属性，如造型的透明性、反光度、发光度等，以便创建出诸如金属、玻璃、木材、石头的质地；为造型添加纹理贴图，增强造型的真实感等。造型的外观设计一般利用 Appearance 节点来完成。

2.4.1 外观节点

外观节点（Appearance）用来设置立体造型的外观属性，通常作为 Shape 节点的 appearance 外观特征域的域值。

Appearance 节点的层次结构见表 2-5。

表 2-5 Appearance 节点的层次结构表

父节点	域	域值或子节点	域值类型
Appearance 节点	material 材质域	NULL 空值	SFNode
		Material 节点	单一节点
	texture 纹理映射像域	NULL 空值	SFNode 单一节点
		ImageTexture 节点	
		PixelTexture 节点	
		MovieTexture 节点	
	textureTransform 纹理坐标变换域	NULL 空值	SFNode
		TextureTransform 节点	单一节点

Appearance 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Appearance {			
	material	NULL	#exposedField SFNode
	texture	NULL	#exposedField SFNode
	textureTransform	NULL	#exposedField SFNode
	}		

其中：

（1）material 域的域值用于设定立体造型外观的材质属性，包括颜色、透明度、反光度、发光度等。域值包含一个 Material 节点或者取 NULL 空值，造型的外观材质属性是通过 Material 节点来设置的。该域值的默认值是 NULL，即 appearance Appearance{} 或 appearance Appearance{material NULL}，表示与 Appearance 相关的几何体本身为发光白光的材质，忽略周围所有光照，造型为纯白色，无任何亮度对比，无立体感。如果设置为 appearance Appearance{material Material{}}，则表示包含一个默认材质节点，几何体采取默认材质的光照，呈现浅灰色造型，有亮度对比和立体感。

（2）texture 域的域值用于设定立体造型表面静态或动态的纹理贴图。该域包括 3 种不同类型的纹理节点：ImageTexture 节点、MoiveTexture 节点和 PixelTexture 节点。该域的默认值为 NULL，表示造型表面无纹理贴图。

（3）textureTransform 域的域值用于对使用的纹理贴图进行坐标变换。该域包含一个 TextureTransform 节点，对 texture 域中纹理节点指定的纹理贴图进行裁剪、旋转设置，使相同的纹理贴图产生多种不同的纹理效果。

【例 2-17】 对例 2-1 中外观特征域的设置做了一下修改，将原有的 appearance Appearance {material Material{}}，改为 appearance Apperance{}，可以发现长方体造型为纯白色，没有明暗

层次，看不到面与面相接的棱线，没有立体感。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
  }
  geometry Box{
    size 3.0 2.0 3.0
  }
}
```

运行程序，效果如图 2-27 所示。

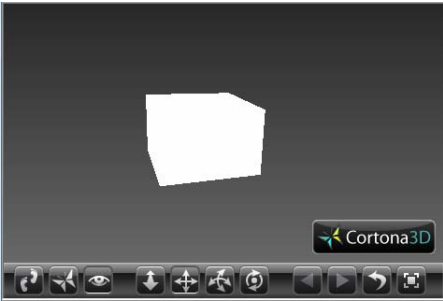


图 2-27 无材质设置的长方体造型

2.4.2 材质节点

材质节点（Material）设定造型外观的颜色、透明度、发光度、反光度等基本材质特征，因此在造型外观设计中，重点介绍 Material 节点的使用方法。

Material 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Material {	diffuseColor	0.8 0.8 0.8	#exposedField SFCOLOR
	ambientIntensify	0.2	#exposedField SFCOLOR
	specularColor	0 0 0	#exposedField SFCOLOR
	emissiveColor	0 0 0	#exposedField SFCOLOR
	shininess	0.2	#exposedField SFCOLOR
	transparency	0	#exposedField SFCOLOR
	}		

其中：

（1）diffuseColor 域的域值设定物体的漫反射颜色。漫反射颜色指的是：当光照射在物体表面时，物体表面向各个方向反射的基本色彩。造型的颜色主要由该域值来设定，该域值由 3 个浮点数组成。3 个浮点数分别代表 Red、Green、Blue（红、绿、蓝）三原色的含量，每个浮点数的取值范围都为（0.0~1.0）。例如：（1 0 0）为红色，（0 1 0）为绿色，（0 0 1）为蓝色，（0 0 0）为黑色，（1 1 1）为白色。该域的默认值为（0.8 0.8 0.8）为浅灰色。

(2) `ambientIntensity` 域的域值设定有多少环境光线被物体表面反射。环境光各向同性，环境光颜色以 $(\text{ambientIntensity}) \times (\text{diffuseColor})$ 计算。该域值的取值范围也为 $(0.0 \sim 1.0)$ ，该域值的默认值为 (0.2) ，表示对环境光产生较低的反射效果。

(3) `specularColor` 域的域值设定物体镜面反射光的颜色。入射角等于出射角是镜面反射的基本原理。设置了镜面反射，会因观察角度变化感受到强弱不同的反射效果。该域值也是由代表三原色含量的浮点值组成。该域的默认值为 $(0\ 0\ 0)$ ，表示镜面反射光为黑色。

(4) `emissiveColor` 域的域值设定发光物体产生光的颜色。该域值的默认值为 $(0\ 0\ 0)$ ，黑色表示不发光。

(5) `shininess` 域的域值设定物体表面的亮度。该域值的取值范围从漫反射表面的 0 到高度抛光表面的 1 ，较小的取值使物体看上去像纤维材质，较大的取值使物体看上去像金属材质。该域的默认值为 (0.2) ，表示选择适当的材质亮度。

(6) `transparency` 域的域值设定物体的透明度。该域值的取值范围从完全不透明的 0 到完全透明的 1 。其默认值为 0 ，表示不透明。

【例 2-18】 绘制一个发光的灯泡。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1.0 1.0 1.0          #颜色
      transparency 0.5
    }
  }
  geometry Cylinder {
    radius 0.05
    height 3.0
  }
}

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1.0 0.0 0.0          #颜色
    }
  }
  geometry Cylinder {
    radius 0.01
    height 3.0
  }
}

Transform {
  translation 0.0 -2.0 0.0
  children [
```

```
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0.5 0.5 0.5      #颜色
        }
      }
      geometry Sphere {
        radius 1.0
      }
    }
  ]
}
Transform {
translation 0.0 -2.0 0.0
children [
  Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0.5 0.5 0.5      #颜色
      }
    }
    geometry Cylinder {
      radius 1.0
      height 1.0
    }
  }
]
}
Transform {
translation 0.0 -4.2 0.0
children [
  Shape {
    appearance Appearance {
      material Material {
        emissiveColor 1.0 1.0 0.0
      }
    }
    geometry Sphere {
      radius 1.5
    }
  }
]
}
```

运行程序，效果如图 2-28 所示。

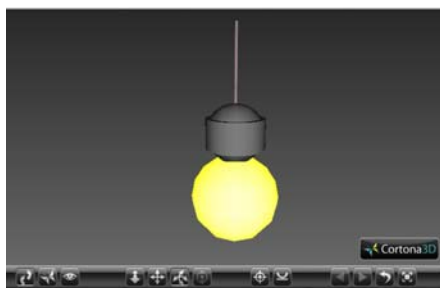


图 2-28 绘制发光的灯泡效果

【例 2-19】 下面以一组红、绿、蓝、蓝绿色的正方体造型。在造型构建中运用了 Transform 节点。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Transform {                                     #坐标变换节点
translation -5 0 0
children [
  Shape {
    appearance Appearance {
      material Material {
        diffuseColor 1 0 0                     #红色
      }
    }
    geometry Box {}                             #正方体
  }
]
}

Transform {
translation -2.5 0 0
children [
  Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0 1 0                     #绿色
      }
    }
    geometry Box {}
  }
]
}

Transform {
translation 0 0 0
children [
  Shape {
```

```

        appearance Appearance {
            material Material {
                diffuseColor 0 0 1          #蓝色
            }
        }
        geometry Box {}
    }
}
}
Transform {
translation 2.5 0 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 1 1 0          #黄色
            }
        }
        geometry Box {}
    }
}
}
Transform {
translation 5 0 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 1 1          #蓝绿色
            }
        }
        geometry Box {}
    }
}
}
}
}

```

运行程序，效果如图 2-29 所示。

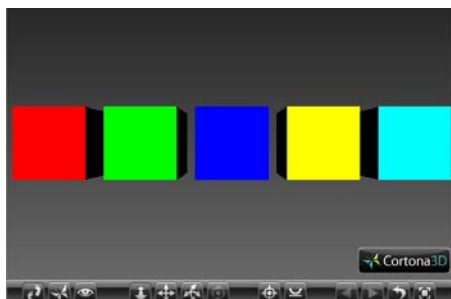


图 2-29 一组不同颜色的正方体

1. 几何造型的简单色彩设计

VRML 的色彩设计分成两种类型：一种为整体造型添加同样的颜色；另一种为造型添加变化的颜色。第一种类型设计方法比较简单，只要为 `diffuseColor` 域的域值设定一组合适 RGB 分量值，就完成了对造型基本颜色的设计，浏览器会自动为整体造型的各个表面添加相同的漫反射颜色。简单几何造型的色彩设计通常采用这种方法，这种方法也适用于为复杂造型添加单一颜色的情况。

【例 2-20】 四棱锥体设定为紫红色。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 1.0 0.0 1.0          #紫红色
    }
  }
  geometry IndexedFaceSet {
    coord Coordinate {
      point [
        0 3 0
        3 0 0
        0 -3 0
        -3 0 0
        0 0 3
      ]
    }
    coordIndex [
      0 1 4 -1
      1 2 4 -1
      2 3 4 -1
      3 0 4 -1
      0 1 2 3
    ]
    solid FALSE
  }
}
```

运行程序，效果如图 2-30 所示。

2. 几何造型的复杂色彩设计

第二种类型的色彩设计为造型添加不同的颜色或变化的颜色，着色的方式分为对面添加颜色和对顶点添加颜色。对面添加颜色，可以使构建造型的各个面具有不同的色彩；对顶点添加颜色，可以使顶点之间的面产生渐变过渡的色彩效果。同理，对线集节点造型着色的方式也可分为对线添加颜色和对端点添加颜色。下面用示例说明复杂几何造型的色彩设计方法。

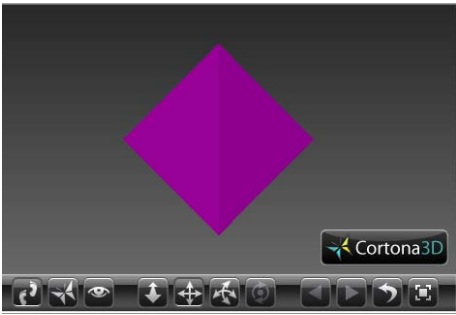


图 2-30 紫红色四棱锥体造型

1) 点集和线集造型的色彩设计

点集和线集造型的色彩着色有两种。

一是通过 Material 节点的 emissiveColor 域的域值，为造型设定单一的颜色。

注意：不能使用 diffuseColor 域的域值设定点和线造型的颜色，因为点集和线集节点创建的造型不受光照的影响，没有光照的情况下，无论 diffuColor 域的域值设定成什么颜色，浏览器均显示为黑色。

二是通过 PointSet 节点和 IndexedLineSet 节点的 color 域的域值 Color 颜色节点，为造型设定不同的颜色。

Color 颜色的节点格式如下：

节点名称	域名称	域值	#域及域值类型
Color { }	color	[]	#exposedField MFColor

其中，Color 节点的 color 域的域值设定一系列的颜色列表，用于造型添加不同的颜色。

【例 2-21】 使用 Color 节点为添加的 5 个顶点添加了红、绿、黄、紫红、白 5 种不同的颜色。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  geometry PointSet {                                #点集节点
    coord Coordinate {                               #坐标节点
      point [
        0 3 0
        3 0 0
        0 -3 0
        -3 0 0
        0 0 3
      ]
    }
  }
  color Color {                                       #颜色节点
    color [
      1 0 0                                           #红色
```

```

        0 1 0          #绿色
        1 1 0          #黄色
        1 0 1          #紫红色
        1 1 1          #白色
    ]
}
}
}
```

运行程序，效果如图 2-31 所示。



图 2-31 为顶点添加不同的颜色效果

在图 2-31 中可以看得很清楚。color 域中的颜色列表与 point 域中的顶点坐标列表一一对应，例如：color 域中的第一种颜色（1 0 0）红色赋予 point 域中第一个坐标点（0 3 0），第二种颜色（0 1 0）绿色赋予第二个坐标点（3 0 0），以此类推。所以，颜色列表中的颜色数至少应该和坐标列表中的坐标点数相等，若多于坐标点的数目则会自动舍去多余的颜色设置，若少于坐标点的数目则需要增加 colorIndex 域，在颜色索引列表中重复使用 color 域中的颜色。需要强调的是，Color 颜色节点的设置优于 Material 节点的颜色设置，使用 Color 颜色节点后，Material 节点的 emissiveColor 域设定的颜色失效，所以，可以省掉 appearance 外观域的全部设置。

【例 2-22】 使用 Material 节点的 emissiveColor 域的域值为创建的四棱锥轮廓线添加单一紫红色。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {
      emissiveColor 1.0 0.0 1.0
    }
  }
  geometry IndexedLineSet {
    coord Coordinate {
      point [
        0 3 0
        3 0 0
```

```

        0 -3 0
        -3 0 0
        0 0 3
    ]
}
coordIndex [
    0 3 4 -1
    3 2 4 -1
    2 1 4 -1
    1 0 4
]
}
}

```

运行程序，效果如图 2-32 所示。

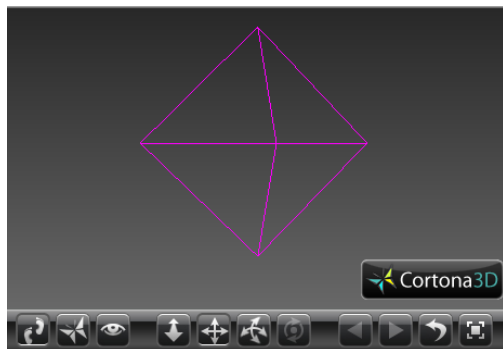


图 2-32 紫红色轮廓线效果

IndexedLineSet 节点有 **color**、**colorIndex**、**colorPerVertex** 3 个域的域值用于为所创建的线段着色，**color** 域的域值设定在例 2-21 中已经作了介绍。**colorIndex** 域的域值用于设定颜色索引列表，把 **Color** 节点中 **color** 域提供的颜色从 0 开始以整型数值顺序编号，即 0 代表第一种颜色，1 代表第二种颜色，以此类推。为复杂造型添加多种颜色的时候，往往使用颜色索引列表指定颜色比直接使用颜色列表简单方便，因为颜色列表是三位浮点数，而颜色索引列表是一位整数型，但是颜色索引列表以颜色列表为基础。**colorPerVertex** 域的域值用于设定对于造型的着色是基于端点还是基于线段，该域值为 **TRUE** 说明着色方式基于线段造型的端点，域值为 **FALSE** 说明着色方式基于整条线段。当 **colorIndex** 域的域值为空列表或默认值时，基于端点的着色方式是将 **color** 域中的颜色按照顺序一一赋予 **point** 域中的坐标点（线造型的端点），两个端点之间的线段将产生颜色渐变等特殊效果；基于线段的着色方式就是将 **color** 域中的颜色按照顺序一一赋予 **coordIndex** 域中指定的线段，每条线段上的颜色不发生变化。

【例 2-23】 用红、天蓝、黄 3 种颜色为两条折线和 5 条直线组合而成的线造型着色。**colorIndex** 域的颜色索引号与 **color** 域的颜色一一对应，0 代表（1 0 0）红色，1 代表（0 0.5 0.8）天蓝色，2 代表（1 1 0）黄色。依照 **colorIndex** 域中颜色索引号排列的顺序为 **coordIndex** 域坐标索引列表指定的线段逐一着色，第一个索引号代表的颜色为第一条线段着色，第二个索引号代表的颜色为第二条线段着色，以此类推。索引号相同则说明重复使用同一种颜色，本例中颜

色索引号为 2 的黄色重复为 5 条线段着色。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  geometry IndexedLineSet {
    coord Coordinate {
      point[
        0.0 0.0 0.0      #0 号端点
        1.0 -1.0 0.0     #1 号端点
        2.0 0.0 0.0      #2 号端点
        3.0 1.0 0.0      #3 号端点
        4.0 2.0 0.0      #4 号端点
        0.0 0.0 1.0      #5 号端点
        1.0 -1.0 1.0     #6 号端点
        2.0 0.0 1.0      #7 号端点
        3.0 1.0 1.0      #8 号端点
        4.0 2.0 1.0      #9 号端点
      ]
    }
    coordIndex [
      0 1 2 3 4 -1      #0 号线段
      5 6 7 8 9 -1      #1 号线段
      0 5 -1             #2 号线段
      1 6 -1             #3 号线段
      2 7 -1             #4 号线段
      3 8 -1             #5 号线段
      4 9                #6 号线段
    ]
    colorPerVertex FALSE      #着色方式基于线段
    color Color {
      color [
        1 0 0      #0 号颜色
        0 0.5 0.8  #1 号颜色
        1 1 0      #2 号颜色
      ]
    }
    colorIndex [          #颜色索引列表
  ]
}
}
```

运行程序，效果如图 2-33 所示。



图 2-33 重复使用颜色的线造型效果

2) 面集造型的色彩设计

`IndexedFaceSet` 节点和 `IndexedLineSet` 节点一样，有 `color`、`colorIndex`、`colorPerVertex` 三个域的域值用于为所创建的平面造型着色。`color` 域中的 `Color` 节点的 `color` 域用于设定一系列为造型添加的颜色列表。`colorIndex` 域的域值用于设定颜色索引列表，把 `color` 域提供的颜色从 0 开始以整型数值顺序编号，并赋予着色的平面或构成平面的顶点。`colorPerVertex` 域的域值用于设定对于造型的着色是基于构成平面的顶点还是基于平面，该域值为 `TRUE` 说明着色方式基于平面造型的顶点，域值为 `FALSE` 说明着色方式基于整个平面。当 `colorIndex` 域的域值为空列表或默认值时，基于顶点的着色方式就是将 `color` 域中的颜色，按照顺序一一赋予 `point` 域中的坐标点（平面造型的顶点），顶点之间的平面将产生颜色渐变等特殊效果；基于平面的着色方式就是将 `color` 域中的颜色，按照顺序一一赋予 `coordIndex` 域中指定的平面，每个平面上的颜色不发生变化。

【例 2-24】 基于顶点着色，可以使用同一个顶点相邻的平面具有不同的颜色。利用 `colorIndex` 域的域值为每一个平面的顶点设定不同的颜色索引列表。注意：每一个平面的颜色索引列表用 -1 结束，`colorIndex` 域的颜色索引列表与 `coordIndex` 域的坐标索引列表所对应的平面顺序相同，颜色索引号代表的颜色按照排列顺序赋予坐标索引号代表的点。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  geometry IndexedFaceSet {
    coord Coordinate {
      point[
        0.0 3.0 0.0      #0 号端点
        3.0 0.0 0.0      #1 号端点
        0.0 -3.0 0.0     #2 号端点
        -3.0 0.0 0.0     #3 号端点
        0.0 0.0 3.0      #4 号端点
      ]
    }
    coordIndex [
      0 3 4 -1           #(0)号平面
```

```
3 2 4 -1          #(1)号平面
2 1 4 -1          #(2)号平面
1 0 4 -1          #(3)号平面
0 1 2 3           #(4)号平面
]
colorPerVertex TRUE      #着色方式基于顶点
color Color {
    color [
        1.0 0.0 0.0      #0 号颜色-红色
        0.0 1.0 0.0      #1 号颜色-绿色
        1.0 1.0 0.0      #2 号颜色-黄色
        0.0 0.0 1.0      #3 号颜色-蓝色
        1.0 1.0 1.0      #4 号颜色-白色
    ]
}
colorIndex [
    0 1 2 -1          #红绿黄-(0)平面 3 个顶点
    4 1 0 -1          #白绿红-(1)平面 3 个顶点
    3 2 1 -1          #蓝黄绿-(2)平面 3 个顶点
    1 2 3 -1          #绿黄蓝-(3)平面 3 个顶点
    4 3 2 1           #白蓝黄绿-(4)平面 4 个顶点
]
}
}
```

运行程序，效果如图 2-34 所示。

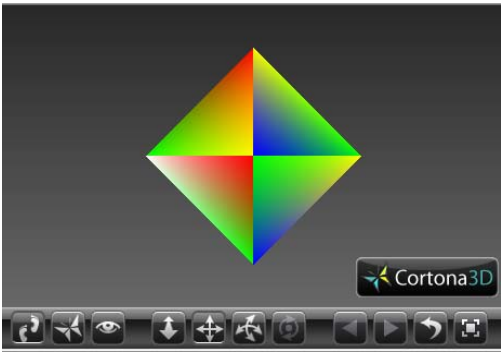


图 2-34 基于顶点着色的四棱锥体效果

【例 2-25】 基于平面着色，如果 color 域中颜色的数目少于 coordIndex 域设定的平面的数目，则需利用 colorIndex 域，在颜色索引域中重复使用相同的颜色。有时这种方法要比在 color 域中增加重复的颜色列表简单清楚。colorIndex 域中第一个颜色索引号代表的颜色赋予 coordIndex 域设定的第一个平面，第二种颜色赋予第二个平面，以此类推。相同的颜色索引号说明重复使用同一种颜色。本例中的索引号为 1 的绿色和索引号为 2 的黄色都被使用了两次。其实现的源程序代码如下：

```

#VRML V2.0 utf8

Shape {
  geometry IndexedFaceSet {
    coord Coordinate {
      point[
        0.0 3.0 0.0      #0 号端点
        3.0 0.0 0.0      #1 号端点
        0.0 -3.0 0.0     #2 号端点
        -3.0 0.0 0.0     #3 号端点
        0.0 0.0 3.0      #4 号端点
      ]
    }
    coordIndex [
      0 3 4 -1            # (0)号平面
      3 2 4 -1            # (1)号平面
      2 1 4 -1            # (2)号平面
      1 0 4 -1            # (3)号平面
      0 1 2 3             # (4)号平面
    ]
    colorPerVertex FALSE  #着色方式基于平面
    color Color {
      color [
        1.0 0.0 0.0        #0 号颜色-红色
        0.0 1.0 0.0        #1 号颜色-绿色
        1.0 1.0 0.0        #2 号颜色-黄色
      ]
    }
    colorIndex [
      2 1 2 1 0            #黄-(0)(2)平面
                          #绿-(1)(3)平面
      ]                    #白-(4)平面
    }
  }
}

```

运行程序，效果如图 2-35 所示。

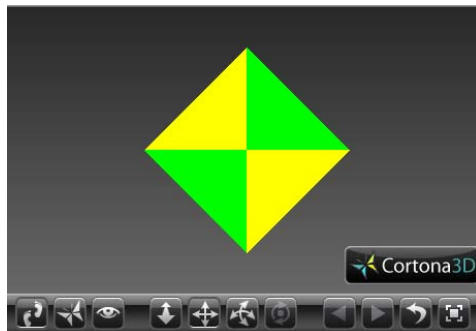


图 2-35 重复使用颜色索引的四棱锥体效果

3) 网格标高造型的色彩设计

ElevationGrid 节点有 color、colorPerVertex 两个域的域值用于为网格标高造型添加颜色。color 域中 Color 节点的 color 域用于设定一系列为造型添加的颜色列表。colorPerVertex 域的域值用于设定对于造型的着色是基于标高网格的顶点还是基于网格的平面。该域值为 TRUE 说明着色方式基于网格的顶点，域值为 FALSE 说明着色方式基于网格平面。基于顶点的着色方式就是 color 域中的颜色，按照顺序逐行一一赋予网格顶点，顶点之间的平面将产生颜色的渐变的特殊效果；基于平面的着色方式就是将 color 域中的颜色，按照顺序逐行一一赋予网格的平面，每个平面上的颜色不发生变化，无渐变效果。

【例 2-26】 采用基于网格顶点的着色方式，创建桌子造型渲染成桌腿及周边为黄色、桌面为蓝白相间条状的色彩。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
```

```
Shape {
```

```
  geometry ElevationGrid {
```

```
    xDimension 11
```

```
    xSpacing 0.4
```

```
    zDimension 9
```

```
    zSpacing 0.6
```

```
    height [
```

```
      -2 0 0.3 0 0.3 0 0.3 0 0.3 0 -2
```

```
      0 0 0 0 0 0 0 0 0 0
```

```
      0.3 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3
```

```
      0 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0
```

```
      0.3 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3
```

```
      0 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0
```

```
      0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3
```

```
      0 0 0 0 0 0 0 0 0 0
```

```
      -2 0 0.3 0 0.3 0 0.3 0 0.3 0 -2
```

```
  ]
```

```
  colorPerVertex TRUE
```

```
  color Color {
```

```
    color [
```

```
      1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0
```

```
      1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0
```

```
      1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0
```

```
      1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0
```

```
      1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0
```

```
      1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0
```

```
      1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0
```

```
      1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0,0 1,1 1 0
```

```
      1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0
```

```
    ]
```

```

    }
    creaseAngle 1.571
    solid FALSE
  }
}

```

运行程序，效果如图 2-36 所示。

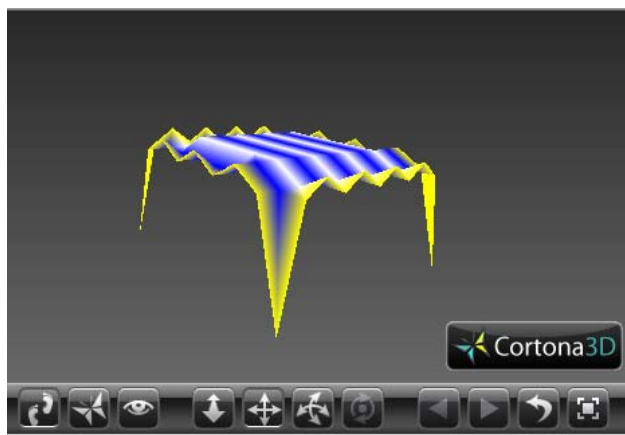


图 2-36 基于网格顶点着色的桌子效果

【例 2-27】 采用基于网格平面的着色方式，将创建的桌子造型渲染成桌腿及周边为黄色、桌面为蓝白相间的条状色彩，没有颜色渐变的效果，注意观察与例 2-26 的差异。因为网格平面的数目为 $(xDimension-1) \times (zDimension-1)$ ，所以，color 域中排列的颜色比例 2-26 中少一行一列。

其实现的源程序代码如下：

```

#VRML V2.0 utf8

Shape {
  geometry ElevationGrid {
    xDimension 11
    xSpacing 0.4
    zDimension 9
    zSpacing 0.6
    height [
      -2 0 0.3 0 0.3 0 0.3 0 0.3 0 -2
      0 0 0 0 0 0 0 0 0 0
      0.3 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0 0.3
      0 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0 0
      0.3 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0 0.3
      0 0 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0 0
      0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3 0.3
      0 0 0 0 0 0 0 0 0 0
      -2 0 0.3 0 0.3 0 0.3 0 0.3 0 -2
    ]
  }
}

```

```
]
colorPerVertex TRUE
color Color {
  color [
    1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0
    1 1 0,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,1 1 0
    1 1 0,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,1 1 0
    1 1 0,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,1 1 0
    1 1 0,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,1 1 0
    1 1 0,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,1 1 0
    1 1 0,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,1 1 0
    1 1 0,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,0 0 1,1 1 1,1 1 0
    1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0,1 1 0
  ]
}
creaseAngle 1.571
solid FALSE
}
}
```

运行程序，效果如图 2-37 所示。



图 2-37 基于网格平面着色的桌子效果

2.4.3 图片型的表面材质节点

图片型的表面材质节点（ImageTexture）是 Appearance 节点中 texture 字段的节点之一，用于在几何对象的表面粘贴上图片作为这个对象的外观，例如，用 Box 节点画出一面墙，如果想使这面墙具真实性，可以用 ImageTexture 节点，将砖墙的图片粘贴在这个对象的表面。贴上的图片是一个二维图片，它的扩张方向是 0~1，不管是在水平（S）方向还是垂直（T）方向。

ImageTexture 节点格式如下：

节点名称	域名称	域值	#域及域值类型
ImageTexture {			

	ulr	[]	#exposedField MFString
	repeatS	TRUE	#SFBool
	repeatT	TRUE	#SFBool
	}		

其中：

- （1）ulr 域的域值用于粘贴图片的文件及来源位置，图片的格式可以是 JPEG、PNG 或 GIF 等。
- （2）repeatS 域的域值是布尔值。如果是 TRUE，粘贴图片会重复填满这个几何对象的表面到[0.0 1.0]的范围外，在水平（S）方向，此为默认值；如果是 FALSE，粘贴图片只会被限制在[0.0 1.0]的范围内，重复填满几何对象表面，在水平方向。
- （3）repeatT 域的域值是布尔值。如果值为 TRUE，粘贴图片会重复填满这个几何对象的表面到[0.0 1.0]的范围外，在垂直（T）方向；如果值为 FALSE，粘贴图片只会被限制在[0.0 1.0]的范围内，重复填满几何对象表面，在垂直方向。

【例 2-28】 利用 ImageTexture 绘制易拉罐图片。
其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    Shape {
      appearance Appearance {
        material Material {}
        texture ImageTexture {
          url "top.jpg"
        }
      }
      geometry Cylinder {
        bottom FALSE
        side FALSE
        height 2.7
      }
    }
    Shape {
      appearance Appearance {
        material Material {}
        texture ImageTexture {
          url "bottom.jpg"
        }
      }
      geometry Cylinder {
        top FALSE
        side FALSE
        height 2.7
      }
    }
  ]
}
```

```
    }
  }
  Shape {
    appearance Appearance {
      material Material {}
      texture ImageTexture {
        url "side.jpg"
      }
    }
    geometry Cylinder {
      height 2.7
      top FALSE
      bottom FALSE
    }
  }
}
```

运行程序，效果如图 2-38 所示。

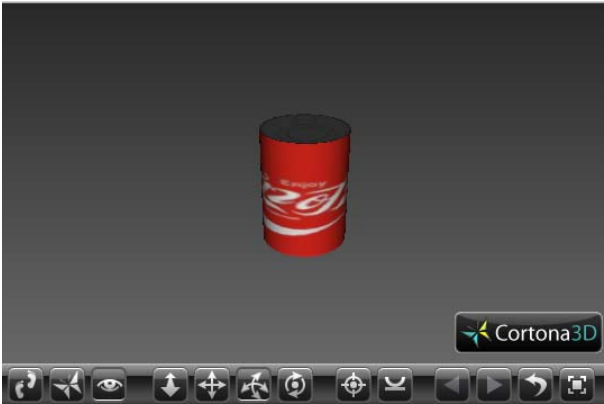


图 2-38 绘制易拉罐效果

2.4.4 表面材质转换节点

表面材质转换节点(TextureTransform)是 Appearance 节点中 textureTransform 字段的字段值。Transform 意为转换，而 TextureTransform 节点的功能就是改变粘贴在几何对象表面的图片或影片的位置，可以使其转动、平移或改变图片的尺寸。

TextureTransform 节点的格式如下：

节点名称	域名称	域值	#域及域值类型
TextureTransform {	center	0 0	#exposedField SFVec2f
	rotation	0	#exposedField SFFloat
	scale	1 1	#exposedField SFVec2
	translation	0 0	#exposedField SFVec2f

```
}
```

其中：

(1) **center** 域的域值是一个二维的浮点数向量值。它的功能是定义一个粘贴图片的任意几何中心点，作为旋转 (**rotation**) 或缩放尺寸 (**scale**) 的中心位置，默认位置是 (0 0)，以粘贴图片的 (S,T) 坐标为坐标。

(2) **rotation** 域的域值是一个浮点数的字段值。它的功能是依照所定义的几何中心点旋转，旋转的角度是弧度 (**radians**)，以粘贴图片的 (S,T) 坐标为坐标。默认值为 0。

(3) **scale** 域的域值是一个二维的浮点数向量。它的功能是依照所定义的几何中心点，做旋转，旋转的角度是弧度，以粘贴图片的 (S,T) 坐标为坐标。默认值为 0。

(4) **translation** 域的域值是一个二维的浮点数向量值。它可以重新定义欲粘贴图片的位置，以粘贴图的 (S,T) 坐标为坐标。默认值为 0。

【例 2-29】 利用 **TextureTransform** 函数绘制创建的立方体粘贴图片。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
    texture ImageTexture {
      url "eden.jpg"
    }
  }
  geometry Box {
    size 1 1 1
  }
}

Transform {
  translation 2 0 0
  children [
    Shape {
      appearance Appearance {
        material Material {}
        texture ImageTexture {
          url "eden.jpg"
        }
        textureTransform TextureTransform {
          center 0.5 0.5
          rotation 0.785
        }
      }
      geometry Box {
        size 1 1 1
      }
    }
  ]
}
```

```
    }  
  }  
}
```

运行程序，效果如图 2-39 所示。



图 2-39 材质表面粘贴图片效果

2.4.5 影像纹理节点

对于用于纹理映射的 MPEG 电影纹理文件，则需用影像纹理节点（MovieTexture）。利用这个节点，提供 MPEG 文件的 URL 和域值来控制电影的播放时间和速度。

MovieTexture 节点说明了纹理映射属性，并可用作 Appearance 节点使 texture 域的值。如果一个电影文件包括伴音，当播放时 MovieTexture 节点会为 Sound 节点 source 域指定所需要的声音文件，这将在后面章节介绍到。

MovieTexture 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
TextureTransform {	url	[]	#exposedField MFString
	speed	1	#exposedField MFFloat
	loop	FALSE	#exposedField SFBBool
	startTime	0	#exposedField SFTTime
	stopTime	0	#exposedField SFTTime
	repeatS	TRUE	#SFBBool
	repeatT	TRUE	#SFBBool
	duration_changed		#SFTTime
	isActive		#SFBBool
	}		

其中：

（1）url 域值指定纹理文件的 URL。如果指定了多个 URL，则浏览器按排列的顺序装载第一个能找到的文件。纹理电影文件必须为 MPEG 图形文件格式。MPEG 文件格式的变体 MPEG-System（声音和图形）和 MPEG-1Video（只有图形）均被支持。每一个 MPEG 电影都存储了一系列电影画面，其中的每一幅都可以作为造型的纹理贴图。电影画面的数目决定了电

影的持续时间。

(2) **startTime** 公共域的值指定电影纹理开始播放时间。**startTime** 域值是从格林威治时间 1970 年 1 月 1 日 00: 00 开始的以秒计算的绝对时间。其默认值是 0.0s。

(3) **stopTime** 公共域的值指定电影何时停止播放。**stopTime** 域值为绝对时间，默认值是 0.0s。

(4) **speed** 公共域的值指定电影纹理播放速度的乘法因子。值 1.0 表示电影以正常速度播放。0.0~1.0 之间的值会放慢播放速度，1.0 以上的值加快电影的播放速度。负值为电影倒放。其默认值为 1.0 使电影以正常速度向前播放。指定电影纹理播放速度的乘数因子。值 1 表示以正常速度播放；值 2 表示以双倍的速度播放；负值表示倒放。当电影正在播放时，**set_speed** 事件将被忽略。如果 **speed** 为 0，**MovieTexture** 将总显示第 0 帧。当一个电影纹理变为非激活状态时，相应变化的那一刻的帧将保持为物体上的当前纹理。

(5) **loop** 公共域的值指定电影是否循环播放。如果此域的为 TRUE，电影将重复播放；如果 **loop** 域值为 FALSE，电影播放一次后停止。**loop** 域的默认值为 FALSE。

(6) **duration_changed** 说明电影的持续时间。随着电影纹理文件的装载，该值被设置。-1 表明电影纹理文件还没被装载。当电影文件被 VRML 浏览器读出时，以秒为单位的电影持续时间取决于 **duration_changed** 域（**eventOut** 出事件）并由其输出。输出持续时间与 **speed** 域所选择的播放速度无关。如果电影文件出了问题，或者由于某些原因持续时间无法确定，**duration_changed** 域（**eventOut** 出事件）将输出 -1 取代电影持续时间。

(7) **isActive** 为 SFFBool 值类型的 **eventOut** 事件，指明电影当前是否正在播放。当开始播放时该事件以 TRUE 值送出；当播放结束时以 FALSE 值送出。**startTime**、**stopTime**、**speed** 和 **loop** 域的值共同作用控制 **MoiveTexture** 节点的纹理。在开始时间 **MovieTexture** 节点被激活用 **isActive** 域（**eventOut** 出事件）输出 TRUE，并开始正向播放第一画面（如果 **speed** 域的为负，则为反向）。在每一幅新的电影画面中，造型纹理被变为新的电影画面。如果 **loop** 域的为 FALSE，那么直接停止时间，或 **startTime+duration/speed** 的播放时间到达之后，**MovieTexture** 节点才开始产生纹理。如果 **loop** 域的为 TRUE，那么 **MovieTexture** 节点持续产生纹理，直至停止时间。在停止时间早于或造型开始时间的特殊情况下，停止时间将被忽略。这可以用来不停地产生电影纹理。在任何情况下，当电影停止时，**isActive** 域（**eventOut** 出事件）将输出 FALSE，电影中的最后一幅画面保留作为造型纹理。

(8) **repeatS** 和 **repeatT** 的含义与 2.4.3 节中 **repeatS** 和 **repeatT** 的含义相同，此处不再赘述。**startTime**、**stopTime**、**speed** 和 **loop** 域的值可以一起使用生成几个标准效果，见表 2-6。

表 2-6 基于 **startTime**、**stopTime**、**speed** 和 **loop** 域值的标准效果

Loop 域值	startTime、stopTime、speed 和 loop 域值的关系	效果
TRUE	$stopTime \leq startTime$	一直执行
TRUE	$startTime < stopTime$	执行到 stopTime
FALSE	$stopTime \leq startTime$	执行一个循环，然后停止在 $(stopTime + duration / speed)$
FALSE	$startTime < (stopTime + duration / speed) \leq stopTime$	执行一个循环，然后停止在 $(stopTime + duration / speed)$
FALSE	$startTime \leq startTime < (startTime + duration / speed)$	执行到 stopTime

发送一个事件到公共域的隐含 **set_speed** 域（**eventIn** 入事件），可以改变 **speed** 公共域的值。

当新值被收到时，如果节点是活动的，新值将被忽略。否则，新值将设置公共域，并由此公共域的隐含 `speed_changed` 域（`eventOut` 出事件）输出。

发送一个 `TRUE` 或 `FALSE` 事件到公共域的隐含 `set_loop` 域（`eventIn` 入事件），可以改变 `loop` 域的值。如果 `loop` 域的值由 `TRUE` 变为 `FALSE`，并且 `MovieTexture` 节点是活动的，正处于播放状态，那么将会继续播放到一周结束，或停止时间；如果 `loop` 域的值由 `FALSE` 变为 `TRUE`，并且节点是活动的，那么将播放到停止时间。`loop` 域的值被改变后，新值由此公共域的隐含 `loop_changed` 域（`eventOut` 出事件）输出。

发送一个事件到公共域的隐含 `set_startTime` 域（`eventIn` 入事件），可以改变 `startTime` 公共域的值。当新值被收到时如果节点是活动的，新值将被忽略。否则，新值将设置公共域，并由此公共域隐含 `startTime_changed` 域（`eventOut` 出事件）输出。

当一个新的开始时间被接收到时如果 `MovieTexture` 节点是非活动的，节点的开始时间将被改变。如果新的开始时间为当前时间或当前时间小于（`startTime+duration/speed`），这会导致节点变成活动的。

发送一个事件到公共域的隐含 `set_stopTime` 域（`eventIn` 入事件），可以改变 `stopTime` 公共域的值。如果新的停止时间早于开始时间，新的停止时间将被忽略。否则，新的停止时间将改变 `stopTime` 域的值，并由此公共域的时间的基础上，新的停止时间用来重新估计电影是否到了停止播放的时间。

颜色、灰度和 `alpha` 值与造型 `MovieTexture` 和 `Color` 节点的域一起作用，改变造型表面的颜色和透明度等级。

由 `PointSet` 和 `IndexedLineSet` 节点构造的造型、纹理将被忽略。

`MovieTexture` 节点指定了一幅电影纹理图和控制播放及纹理映射的参数。纹理图是在水平轴（S）和垂直轴（T）均从 0~1 延展的二维图像。

【例 2-30】 利用 `MovieTexture` 节点的绘制效果。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {}
    texture MovieTexture {
      url "run.jpg"
      loop TRUE
      startTime 5.0
      stopTime 10.0
    }
  }
  geometry Cylinder {
    radius 0.6
    height 1.0
  }
}
```

运行程序，效果如图 2-40 所示。

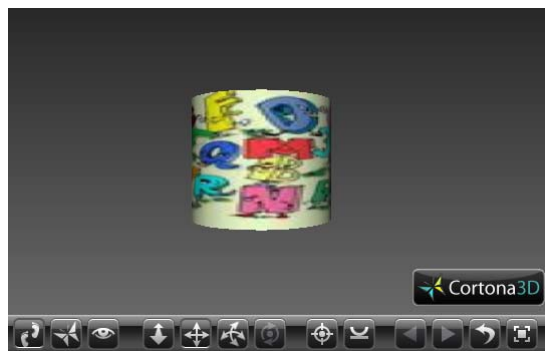


图 2-40 MovieTexture 节点效果

2.4.6 纹理坐标节点

改变纹理应用的办法是使用纹理坐标节点（TextureCoordinate）。TextureCoordinate 通过对它们实施一种变换，修改这些坐标值，而 TextureCoordinate 节点则允许在一个形状的每一个顶点控制 S 和 T 的坐标值。只有 IndexedFaceSet 和 ElevationGrid 节点允许使用 TextureCoordinate，这些形状中的每一个都有一个 texCoord 域，其中可以放入 TextureCoordinate 节点。

IndexedFaceSet 和 ElevationGrid 节点造型的边界立方体由 VRML 的浏览器计算生成其长、宽和高。纹理贴图投影到面集的造型上。纹理贴图的水平轴沿边界立方体的最长边延伸，而纹理贴图投影的垂直轴沿边界立方体的第二个长边方向延伸。为了避免纹理贴图回绕，在第二个长边小于最长边时，纹理贴图的顶部可以被截去。

IndexedFaceSet 和 ElevationGrid 节点的 texCoordIndex 域中定义了一组描述一个或多个纹理切割器形状的纹理坐标索引，每一个坐标索引对应面集中的一个表面。序列中的每一个值是一个整数，该值与坐标索引域中所列的纹理坐标之一对应。texCoord 域的默认值是一个空序列。如果 texCoordIndex 为空，而 texCoord 中给定了一个纹理坐标序列，则在 CoordIndex 域中给定的坐标索引被用作纹理坐标序号。

如果提供了纹理坐标索引，每一个索引要在 CoordIndex 域中给出的坐标索引一一对应。第一个纹理坐标序号与将被匹配的第一个坐标序号对应，第二个纹理坐标序号与将匹配的第二个坐标序号对应，并以此类推。对每一个坐标序号要有一个纹理坐标序号对应，包括用作面分隔符的坐标序号-1。

纹理坐标节点的值可以通过向各造型节点的公共域 texCoord 的隐含 set_texCoord 域（eventIn 入事件）发送值来改变。当该输入收到一个值时，纹理坐标节点被改变，新的纹理坐标节点由该公共域的隐含 texCoord_changed 域（eventOut 出事件）。纹理坐标索引序列可以通过向 set_texCoord 域（eventIn 入事件）发送一系列纹理坐标索引改变。

TextureCoordinate 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
TextureCoordinate{			
	point	[]	#exposedField MFVec2f
}			

其中, point 域有一系列的 SFVec2f 值, 是以二维 (S,T) 形式给出的一组纹理坐标点, 它们一般与 IndexFaceSet 或 ElevationGrid 中的顶点构成对应关系。对于 IndexedFaceSet, 有两种在 point 列表中使用纹理坐标的方法: 如果 IndexedFaceSet 节点的 texCoordIndex 域为空, 则纹理坐标与 Coordinate 节点中的点一一对应; 如果 texCoordIndex 域非空, 将用它选择纹理坐标, 其方法与用 coordIndex 域从 Coordinate 节点中选择点的方向相同。

【例 2-31】 使用 TextureCoordinate 节点的绘制效果。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

Group {
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1.0 1.0 0.0
        }
        texture ImageTexture {
          url "puke.jpg"
        }
      }
      geometry IndexedFaceSet {
        coord Coordinate {
          point [
            0.0 0.0 0.0, 1.0 0.0 0.0
            1.0 2.0 0.0, 0.0 2.0 0.0
          ]
        }
        coordIndex [0 1 2 3]
        texCoord TextureCoordinate {
          point [
            0.0 0.5 0.5 0.5
            0.5 1.0 0.0 1.0
          ]
        }
        texCoordIndex [0 1 2 3]
      }
    ]
  ]
}
```

运行程序, 效果如图 2-41 所示。

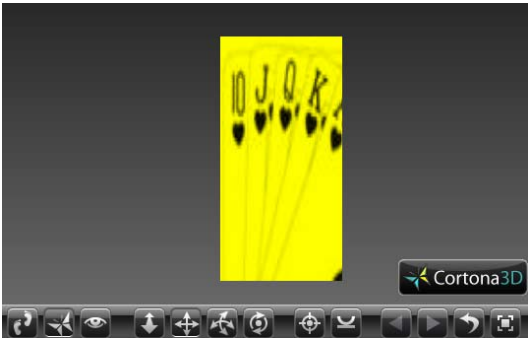


图 2-41 TextureCoordinate 节点效果

2.4.7 造型的材质设计

造型的材质设计是一项复杂的工作，现实世界中常用的材料，如金属、木材、砖瓦、石料、玻璃、塑料等不仅仅是外观颜色上的差别，材质的属性更多反映在透明度、发光颜色、反光度、外观亮度的不同。因此，材质设计必须依赖于 **Material** 节点中 6 个域值的综合设定来完成。表 2-7 是常用材质域值设定的参考表。

表 2-7 常用材质 Material 节点的域值

视觉效果	材料漫反射颜色	环境光反射	物体镜面反射颜色	材质外观亮度
	diffuseColor	ambientIntensity	specularColor	Shininess
黄金	0.5 0.3 0.0	0.4	0.7 0.7 0.6	0.2
白银	0.5 0.5 0.7	0.4	0.8 0.8 0.9	0.2
铜	0.4 0.2 0.0	0.28	0.8 0.4 0.0	0.1
铝	0.3 0.3 0.5	0.3	0.7 0.7 0.8	0.1
红塑料	0.8 0.2 0.2	0.1	0.8 0.8 0.8	0.15
绿塑料	0.2 0.2 0.8	0.1	0.8 0.8 0.8	0.15
蓝塑料	0.2 0.8 0.2	0.1	0.8 0.8 0.8	0.15

【例 2-32】 使用黄金、白银、铜、蓝塑料等不同材质创建的一组圆柱体造型。
其实现的源程序代码如下：

```
#VRML V2.0 utf8

Transform {
translation -5 0 0
children [
  Shape {          #第一个黄金圆柱体
    appearance Appearance {
      material Material {
        diffuseColor 0.5 0.3 0.0
        ambientIntensity 0.4
        specularColor 0.7 0.7 0.6
```

```

        shininess 0.2
    }
}
geometry Cylinder {}
}
]
}
Transform {
translation -2.5 0 0
children [
    Shape {                                     #第二个白银圆柱体
        appearance Appearance {
            material Material {
                diffuseColor 0.5 0.5 0.7
                ambientIntensity 0.4
                specularColor 0.8 0.9 0.9
                shininess 0.2
            }
        }
        geometry Cylinder {}
    }
]
}
Transform {
translation 0 0 0
children [
    Shape {                                     #第三个铜圆柱体
        appearance Appearance {
            material Material {
                diffuseColor 0.4 0.2 0.0
                ambientIntensity 0.28
                specularColor 0.8 0.4 0.0
                shininess 0.1
            }
        }
        geometry Cylinder {}
    }
]
}
Transform {
translation 2.5 0 0
children [
    Shape {                                     #第四个蓝塑料圆柱体
        appearance Appearance {
            material Material {
                diffuseColor 0.2 0.8 0.2

```

```

        ambientIntensity 0.3
        specularColor 0.8 0.8 0.8
        shininess 0.15
    }
}
geometry Cylinder {}
}
]
}
Transform {
translation 5 0 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.8 0.2 0.2
                ambientIntensity 0.1
                specularColor 0.8 0.8 0.8
                shininess 0.15
            }
        }
        geometry Cylinder {}
    }
]
}
}

```

#第五个红塑料圆柱体

运行程序，效果如图 2-42 所示。

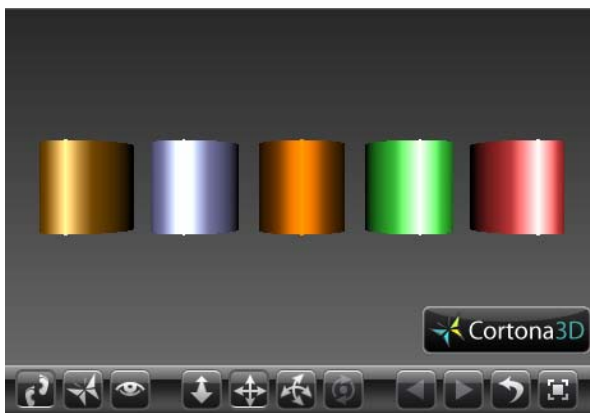


图 2-42 不同材质的圆柱体造型效果

【例 2-33】 浅灰色背景下，一组 5 个透明度不同的绿色球体，其透明度 `transparency` 分别设定为 0、0.25、0.5、0.75、1。球中间穿过一根黄色的棍，观察一下不同设置的可见度。当 `transparency` 设置为 0 时，绿色球体完全不透明，看不到中间穿过的黄色棍。当 `transparency` 设置为 1 时，绿色球体完全透明就看不到轮廓了，只有黄色的棍全部显示出来。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
skyColor 0.8 0.8 0.8
}
Shape {
appearance Appearance {
material Material {
diffuseColor 1 1 0
}
}
geometry Box{
size 14 0.2 0.2
}
}
Transform {
translation -5 0 0
children [
Shape {
appearance Appearance {
material Material {
diffuseColor 0 1 0
transparency 0
}
}
geometry Sphere {}
}
]
}
Transform {
translation -2.5 0 0
children [
Shape {
appearance Appearance {
material Material {
diffuseColor 0 1 0
transparency 0.25
}
}
geometry Sphere {}
}
]
}
Transform {
```

```
translation 0 0 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 1 0
                transparency 0.5
            }
        }
        geometry Sphere {}
    }
]
}
Transform {
translation 2.5 0 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 1 0
                transparency 0.75
            }
        }
        geometry Sphere {}
    }
]
}
Transform {
translation 5 0 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 1 0
                transparency 1
            }
        }
        geometry Sphere {}
    }
]
}
```

运行程序，效果如图 2-43 所示。

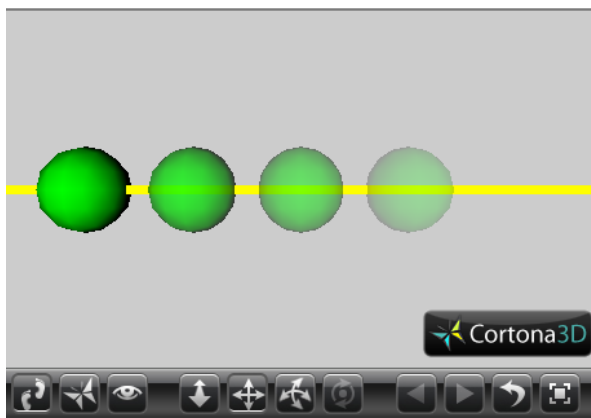


图 2-43 一组透明度不同的球体效果

第 3 章 造型的其他相关操作

3.1 造型空间变换

在 VRML 中创建的立体造型都是以默认坐标系的原点为中心进行定位的，如果需要将多个立体造型组合成更加复杂的整体造型，使用单一的坐标系，所有造型基于相同的定位中心点，就会出现造型重叠在一起无法区分、无法按照需要正确组合的情况。坐标变换节点（Transform）可以创建一个新的坐标系，通过对坐标系的平移、缩放、旋转等操作，实现对立体造型位置、角度、缩放比例的改变。

3.1.1 坐标变换节点

坐标变换节点用于创建一个或多个不同于默认（原始）坐标系的新坐标系。Transform 节点是一个组节点，在此节点下面可以包含一个或多个子节点，所有子节点的定位中心都基于同一个坐标系的原点。这些子节点可以是 Shape 节点、Group 节点，也可以是下一层的 Transform 节点。

Transform 节点的语法格式如下：

节点名称	域名称	域值	#域及域值类型
Inline{	children	[]	#exposedField MFNode
	translation	0 0 0	#exposedField SFVec3f
	rotation	0 0 1 0	#exposedField SFRotation
	scale	1 1 1	#exposedField SFVec3f
	scaleOrientation	0 0 1 0	#exposedField SFRotation
	bboxCenter	0 0 0	#SFVec3f
	bboxSize	-1 -1 -1	#SFVec3f
	center	0 0 0	#exposedField SFVec3f
	addChildren		#eventIn MFNode
	removeChildren		#eventIn MFNode
}			

其中：

- （1）children 域的域值用于设定受该节点变换影响的所有子节点，浏览器将逐个变换并创建该域值所包含的每个造型和造型编组。该域值的默认值为空列表，即不包含任何子节点。
- （2）translation 域的域值用于设定变换后新坐标的原点与原始坐标系原点在 X、Y、Z 方向上的距离。该域值的第一个数值为 X 方向上的距离，第二个数值为 Y 方向上的距离，第三个数值为 Z 方向上的距离，域值可正可负，只是方向相反。该域值的默认值为（0.0 0.0 1.0 0.0），表示新坐标系与原始坐标系重合，没有位移。
- （3）rotation 域的域值用于设定变换后新坐标系相对原始坐标系进行旋转的旋转轴和旋转角度。该域值的前 3 个值是新坐标系的原点在原始坐标系中的 X、Y、Z 坐标分量，两个坐标系原点的连线就是旋转轴，旋转轴的长度可以延伸，不受连线长度的限制；该域值的第四个值

是以弧度为单位的新坐标系的旋转角度。该域值的默认值为 (0.0 0.0 1.0 0.0)。坐标系以 Z 轴为旋转轴，但不发生旋转。

这里需要强调的是旋转方向的确定方法：

① 旋转轴的方向：由原始坐标系的原点指向新坐标系的原点。

② 新坐标系的旋转方向符合右手规则：用右手握住旋转轴，拇指指向轴的正方向。如果旋转角度为正，则旋转方向为其余 4 个手指所指方向；如果旋转方向为负，则旋转方向与其余 4 个手指所指方向相反。

(4) **scale** 域的域值用于设定新坐标系在 X、Y、Z 方向上的缩放系数。该域值的默认值为 (1.0 1.0 1.0)，表示在 X、Y、Z 方向上均无缩放。

(5) **scaleOrientation** 域的域值也是用于设定旋转和旋转角度。与 **rotation** 域的作用不同之处在于：新坐标系的旋转是为了方便缩放操作，先按照旋转后的新坐标系进行缩放操作，然后再将坐标系旋转回原方位。该域值的默认值为 (0.0 0.0 1.0 0.0)，表示旋转轴为 Z 轴，缩放操作前坐标系无旋转。

(6) **bboxCenter** 域的域值设定包围该变换全部子节点的区域中心坐标点。该域值的默认值为 (0.0 0.0 0.0)，表示默认包围区域的中心点位于新坐标系的原点。

(7) **bboxSize** 域的域值设定包围该变换全部子节点的区域在 X、Y、Z 方向上的尺寸。如果设置包围区域，其尺寸必须大到足以涵盖所有子节点。该域值的默认值为 (-1.0 -1.0 -1.0)，表示不人为设置包围区域，由浏览器自动设置。

(8) **center** 域的域值设定了一个三维坐标点。该坐标点位于新坐标系的原点上，坐标系的旋转与缩放均围绕该中心点进行。该域值的默认值为 (0.0 0.0 1.0 0.0)，表示变换的中心点为原始坐标系的原点。

(9) **addChildren** 为入事件，其将要被指定到组节点的列表中。若该节点已在子节点的列表中，则事件将被忽略。

(10) **removeChildren** 入事件将指定的节点从组节点的子节点列表中删除。若该节点不在子节点列表中，则事件将被忽略。

值得强调的一点是：当对造型进行平移、缩放、旋转多项操作时，无论编辑程序中各个域值的排列顺序如何，浏览器的默认执行顺序都为先缩放、再旋转、后平移。

3.1.2 空间坐标的平移

空间坐标的平移是通过对 **Transform** 节点的 **transform** 域的域值的设定，建立新的坐标原点，作为 **children** 域中所有子节点创建的造型的定位点，从而实现原始坐标系和新坐标系造型之间相对位置的移动和变化的。

【例 3-1】 使用 **Transform** 节点的坐标平移显示同位置的两个 **Box**。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
```

```
Transform {
  children [
    Shape {
      appearance Appearance {
        material Material {
```

```

        diffuseColor 0.0 0.6 0.6
    }
}
geometry Box {}
}
Transform {
    translation 4.0 0.0 0.0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.0 0.3 0.3
                }
            }
            geometry Box {}
        }
    ]
}
}
}
}

```

运行程序，效果如图 3-1 所示。

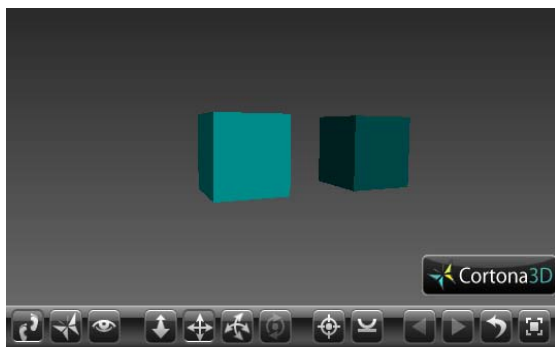


图 3-1 节点的坐标平移效果

3.1.3 空间坐标的旋转

通过造型的三维坐标可以确定造型的位置，但要显示一个有一定倾斜角度的造型，还必须借助 Transform 节点的 rotation 域或 center 域。这些域实际上是将 Transform 节点坐标系围绕一个旋转轴旋转，从而能够实现旋转一个或一组造型。

一个旋转轴是坐标系旋转所围绕的一条虚拟的线。一条旋转轴可指向任何方向。在 VRML 中旋转一坐标系，需要给定一个旋转轴和旋转的角度。旋转角度在 rotation 域的第四位浮点数指定，以弧度计算，旋转角度可正可负。VRML 使用一个向量指定旋转轴。这个向量的起点在 center 域给出，通常默认为旋转坐标系原点。这个向量的终点是 rotation 域的前三位浮点数指定的一个三维点。在这两点间的虚线就是旋转轴，起点到终点的方向即为旋转轴方向。实际上，这个矢量的长度与旋转轴并无关系，如定义一条沿 Y 轴方向上的旋转轴，(0.0 2.0 0.0) 到 (0.0 2.357 0.0) 和 (0.0 0.0 0.0) 到 (0.0 1.0 0.0) 都是一样的，因为它们定义的旋转轴都是笔直地

指向 Y 轴正方向。

坐标系的下移和旋转能够用在同一个 Transform 节点之中。

【例 3-2】 金色十字棒造型，添加 Transform 节点，设定域值 rotation 0 0 1 1.571，将第二根金色圆柱体沿 Z 轴旋转 90°，使两根圆柱体形成十字交叉造型。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {           #第一个圆柱体造型
  appearance Appearance {
    material Material {
      diffuseColor 0.5 0.3 0.4
      ambientIntensity 0.4
      specularColor 0.7 0.7 0.6
      shininess 0.2
    }
  }
  geometry Cylinder {
    height 5
    radius 0.2
  }
}

Transform {           #添加坐标变换节点
  rotation 0 0 1 1.571  #沿 Z 轴旋转 90°
  children [
    Shape {           #第二个圆柱体造型
      appearance Appearance {
        material Material {
          diffuseColor 0.5 0.3 0.4
          ambientIntensity 0.4
          specularColor 0.7 0.7 0.6
          shininess 0.2
        }
      }
      geometry Cylinder {
        height 5
        radius 0.2
      }
    }
  ]
}
```

运行程序，效果如图 3-2 所示。

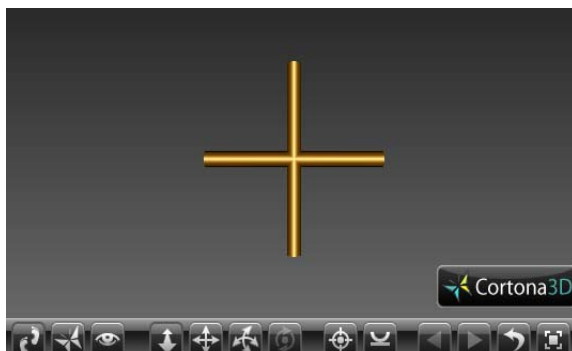


图 3-2 绘制金色十字棒效果

【例 3-3】 简单的钟表造型。如果需要在前面变换的基础上连续进行空间变换，可以使用 Transform 节点进行逐级的嵌套变换，嵌套级数没有限制。

本例中，表轴、时针、分针必须位于表壳表面才能看到，因此设置 Transform 节点的域值为 transform 0 0 1.1，在 Z 方向平移 1.1 个单位；钟表指针显示 3 点钟，分针一端要位于表盘中心，与表轴相接，所以，在前面位移基础上还要在 Y 方向向上平移 0.85 个单位，嵌套 Transform 子节点设定域值为 transform 0 0.85 0；同理，时针在第一级平移基础上，在 X 方向平移 0.6 个单位并顺时针旋转 90°，嵌套 Transform 子节点设定域为 transform 0.6 0 0，rotation 0 0 -1 1.57。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
    #表壳造型
    appearance Appearance {
        material Material {
            diffuseColor 0.6 0.4 0.3
        }
    }
    geometry Box {
        size 4 4 2
    }
}

Transform {
    #表轴造型
    translation 0 0 1.1
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.7 0.3 0.1
                    ambientIntensity 0.4
                    specularColor 0.7 0.7 0.6
                    shininess 0.2
                }
            }
        }
    ]
}
```

```

        geometry Sphere {
            radius 0.1
        }
    }
    Transform {                #分针造型
        translation 0 0.85 0
        children [
            Shape {
                appearance Appearance {
                    material Material {
                        diffuseColor 0.5 0.5 0.7
                        ambientIntensity 0.4
                        specularColor 0.8 0.8 0.9
                        shininess 0.2
                    }
                }
                geometry Cylinder {
                    height 1.5
                    radius 0.05
                }
            }
        ]
    }
    Transform {                #时针造型
        translation 0.6 0 0
        rotation 0 0 -1 1.57
        children [
            Shape {
                appearance Appearance {
                    material Material {
                        diffuseColor 0.3 0.4 0.5
                        ambientIntensity 0.4
                        specularColor 0.8 0.7 0.8
                        shininess 0.2
                    }
                }
                geometry Cylinder {
                    height 1.0
                    radius 0.05
                }
            }
        ]
    }
}
}

```

运行程序，效果如图 3-3 所示。

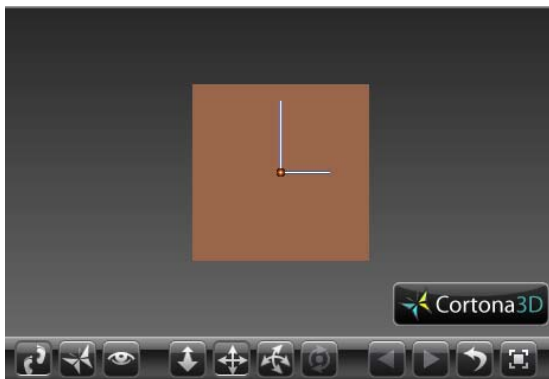


图 3-3 简单钟表造型效果

3.1.4 空间坐标的缩放

现实世界中的物体是千姿百态的，很难用几种标准的造型来概括。例如，前面学习的简单造型节点可以很容易地创建出圆球造型，但是要做出一个椭圆形的气球就不容易了，其实这样的造型可以通过对简单造型圆球进行不同方向上的缩放拉伸来完成。和造型的平移、旋转一样，造型的缩放必须依靠空间坐标度量单位比例的缩放来实现。通常使用的空间坐标，在 X、Y、Z 轴上采用相同比例的度量单位，利用 Transform 节点中 scale 域、scaleOrientation 域和 center 域的域值设定不同方向坐标轴的缩放比例，形成一个新的度量单位缩放比例不同的空间坐标系，然后在新坐标系中创建一个或一组经过缩放处理的造型。

【例 3-4】 在例 3-1 的基础上，使用 rotation 域进行坐标旋转。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Transform {
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0.0 0.6 0.6
        }
      }
      geometry Box {}
    }
  ]
}

Transform {
  translation 4.0 0.0 0.0
  rotation 1.0 0.0 0.0 1.0
  children [
    Shape {
      appearance Appearance {
        material Material {
```

```
diffuseColor 0.0 0.3 0.3
    }
}
geometry Box {}
}
]
}
]
```

运行程序，效果如图 3-4 所示。

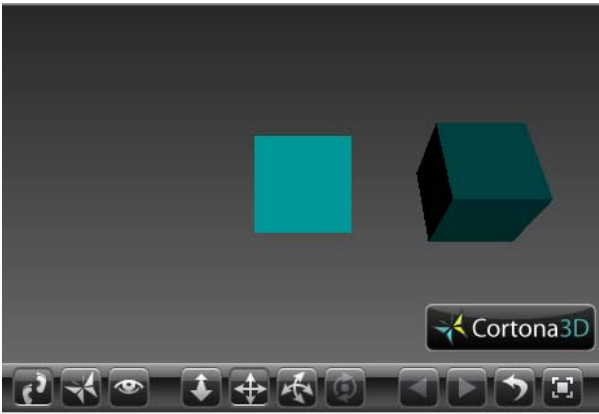


图 3-4 rotation 域坐标旋转效果

【例 3-5】 中心缩放。

本例首先在原始坐标下建立一个橘红色的基准参考平面和一个绿色的标准圆柱体，然后利用 center 域的域值分别设定 4 个不同的缩放中心点，在新的缩放中心点上，设定域值 scale 1 2 1，将标准圆柱体在 Y 方向上放大一倍，建立了 4 个缩放比例相同位置不同的圆柱体。左边第一个紫色圆柱体，缩放中心点设定为 center 0 -1 0，说明原始坐标原点在 Y 方向向上距离新的缩放中心点向下一个单位，所以这圆柱体的中心与标准圆柱体的中心相比上移了一个单位，3/4 的圆柱体都位于基准参考平面的上方。同理，左边第二个金黄色圆柱体，缩放中心点设定为 center 0 -2 0，原始坐标原点在 Y 方向上距离新的缩放中心点向下两个单位，所以这个圆柱体在 Y 方向上放大后全部移动基准参考平面上方。

注意：center 域的域值是以新的缩放中心点作为参考原点的坐标值，即原始坐标原点在坐标系中的坐标值，和通常将原始坐标系作为参照物的习惯相反。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
  skyColor 0.9 0.9 0.9
}

Shape {
  #基准参考平面
  appearance Appearance {
```

```

        material Material {
            diffuseColor 1 0.5 0.4
            specularColor 0.7 0.4 0.4
            ambientIntensity 0.15
            shininess 0.9
        }
    }
    geometry Box {
        size 14 0.1 3
    }
}
Shape {                                #标准圆柱体
    appearance Appearance {
        material Material {
            diffuseColor 0.32 0.54 0.26
            specularColor 0.46 0.46 0.46
            ambientIntensity 0.0933
            shininess 0.51
        }
    }
    geometry Cylinder {
        height 2
        radius 1
    }
}
Transform {                            #左边第一个圆柱体
    translation -6 0 0
    scale 1 2 1
    center 0 -1 0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.3 0.09 0.21
                    specularColor 0.67 0.58 0.83
                    ambientIntensity 0.0467
                    shininess 0.07
                }
            }
            geometry Cylinder {
                height 2
                radius 1
            }
        }
    ]
}

```

```

Transform {                                #左边第二个圆柱体
translation -3 0 0
scale 1 2 1
center 0 -2 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.8 0.51 0.09
                specularColor 0.92 0.43 0.01
                ambientIntensity 0.117
                shininess 0.4
            }
        }
        geometry Cylinder {
            height 2
            radius 1
        }
    }
]
}

Transform {                                #右边第一个圆柱体
translation 3 0 0
scale 1 2 1
center 0 1 0
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.3 0.09 0.21
                specularColor 0.67 0.58 0.83
                ambientIntensity 0.0467
                shininess 0.07
            }
        }
        geometry Cylinder {
            height 2
            radius 1
        }
    }
]
}

Transform {                                #右边第二个圆柱
translation 6 0 0
scale 1 2 1
center 0 0 0

```

```
children [  
  Shape {  
    appearance Appearance {  
      material Material {  
        diffuseColor 0.8 0.51 0.09  
        specularColor 0.92 0.43 0.01  
        ambientIntensity 0.117  
        shininess 0.4  
      }  
    }  
    geometry Cylinder {  
      height 2  
      radius 1  
    }  
  }  
]  
}
```

运行程序，效果如图 3-5 所示。

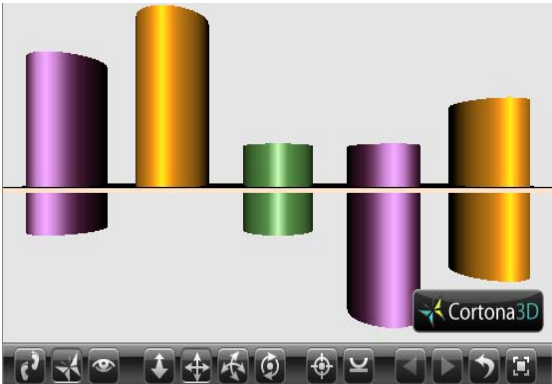


图 3-5 中心点缩放造型效果

3.2 造型群节点

一个虚拟的世界是由很多物体组成的，在前面已创建了很多造型，但这些造型节点是默认位于坐标原点的。如果是多个物体，就会产生重复叠加在一起的情况，这时就需要对各个造型节点定义不同的坐标系。同时还需要对物体进行放大与旋转等变换。

3.2.1 编组节点

编组节点（Group）可以包含任意数目的子节点，将多个简单造型组合成一组复杂造型，作为一个整体进行设计、命名和调用，给 VRML 程序设计带来很多方便之处。

Group 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
------	-----	----	---------

Group{			
	children	[]	#exposedField MFNode
	bboxCenter	0.0 0.0 0.0	#SFVec3f
	bboxSize	-1.0 -1.0 -1.0	#SFVec3f
	addChildren		#eventIn MFNode
	removeChildren		#eventIn MFNode
}			

其中：

(1) children 域的域值用于设定包含在该编组节点中所有子节点，通常为 Shape 节点、Transform 节点及下一层的 Group 节点，组节点的数目无限制。浏览器将逐个创建该域值所包含的所有造型和造型编组。该域值的默认值为空列表，即不包含任何子节点。

(2) bboxCenter 域的域值用于设定包围该编组节点所有造型的包围盒的中心点坐标。其默认值为 (0.0 0.0 0.0)，即中心点位于当前坐标原点。

(3) bboxSize 域的域值用于设定包围盒在当前坐标系中 X、Y、Z 方向上的尺寸。包围盒是一个立方体。该域值的默认值为 (-1.0 -1.0 -1.0)，即不人为设置包围盒尺寸，由浏览器自动设置。

(4) addChildren 入事件用于将指定的节点增加到该编组节点的子节点列表中，若该节点已在子节点列表中，则事件被忽略。

(5) removeChildren 入事件用于将指定的节点从该编组节点的子节点列表中删除，若该节点不在子节点列表中，则事件被忽略。

【例 3-6】 在虚拟空间中，建立一个简易的徽章。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    # 1. 方形外框
    Shape {
      appearance DEF o_white Appearance {
        material Material {
          emissiveColor 1 1 1    #对象发光颜色——白色
        }
      }
      geometry Box {
        size 5.0 0.03 1.6
      }
    }
    # 2.方形内框
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0.0 0.3 0.8    #对象漫反射颜色
        }
      }
    }
  ]
}
```

```
    }  
    geometry Box {  
        size 4.8 0.06 1.4  
    }  
}  
# 3. 圆形外框  
Shape {  
    appearance USE o_white  
    geometry Cylinder {  
        radius 1.6  
        height 0.07  
    }  
}  
#4.圆形内框  
Shape {  
    appearance Appearance {  
        material Material {  
            diffuseColor 0.8 0 0  
        }  
    }  
    geometry Cylinder {  
        radius 1.5  
        height 0.09  
    }  
}  
# 5.VRML 标志  
Shape {  
    appearance Appearance {  
        texture ImageTexture {  
            url "tuan.gif"  
        }  
    }  
    geometry Box {  
        size 0.6 0.1 0.6  
    }  
}  
]  
}
```

运行程序，效果如图 3-6 所示。

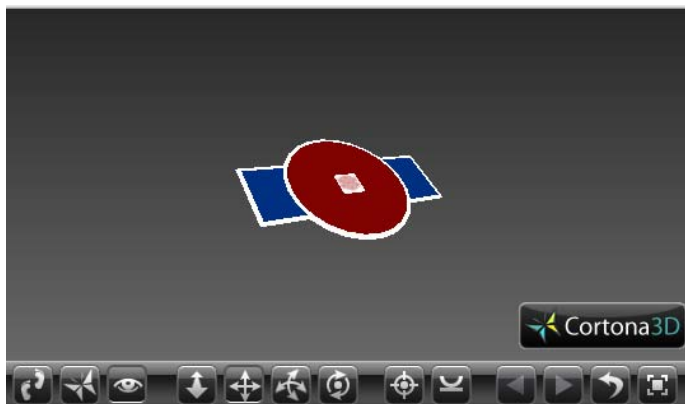


图 3-6 创建的 VRML 徽章效果

注意：在所有对象在 Y 方向最好不要设置相同的高度，否则在浏览时会出现重叠的面相互穿插的情况。

3.2.2 素材调用基本方法

素材调用的基本方法大致可分 4 类。

(1) 采用 **DEF** 节点定义及 **USE** 节点引用方法，在本文件范围内定义和调用不需要进行改进的素材，素材成为原场景的一部分。

(2) 利用 **Inline** 节点，从本文件范围之外的其他文件或互联网上，调用不需要进行修改加工的素材，素材成为原场景的一部分。

(3) 利用带有链接功能的 **Anchor** 节点，把存储在本机硬盘或互联网上的素材作为一个新的完整的场景进行调用，取代原有的场景。

(4) 利用 **PROTO** 语句和 **EXTERPROTO** 语句，在本文件范围内或本文件范围外及互联网上定义原型节点和外部原型节点，调用素材时可以根据需要对其原有的属性进行更改或赋予更多的功能。

3.2.3 节点的定义及引用

VRML 可以采用节点定义及引用的方法，在本文件范围内反复调用相同的素材。在虚拟现实的场景设计中，常常需要将相同的造型经过不同的排列、旋转组合成新的复杂造型。采用节点定义及引用的方法，只需使用 **DEF** 语句为节点定义一个节点名，然后在需要调用此节点的位置，使用 **USE** 语句写出节点名即可。因此，编程中可以很方便地反复引用相同的节点，避免出现大量的重复代码。需要强调的是：定义的节点只在本文件范围内调用有效，一旦对被定义的源节点的域值进行修改，所有引用的节点的域值都随之改变。

节点定义及引用的语法格式如下：

```
节点定义: DEF 节点名 节点{}
节点定义: USE 节点名
```

在定义节点名称和调用时要遵循以下规则。

(1) 节点名可以由字母、数字及下划线组成，但是不能以数字开头，不能包括无法印刷的 ASCII 字符，字母区分大小写。

(2) 节点名不能使用单引号、数字运算符、各种括号及英镑符号。

(3) 节点名不能使用 **VRML** 中已经定义了的节点类型名（包括标准节点类型名与自定义

节点类型名) 以及在 VRML 中有特定意义的字符, 如: Shape, Sphere, Transform 等; NULL, TRUE, FALSE, IS, TO, DEF, USE, PROTO, EXTERPROTO, ROUTE, eventIn, eventOut, exposedField, field 等。

【例 3-7】 对例 3-2 金色十字棒造型的程序进行修改, 使用节点定义及引用方法, 可以看出改进后的程序得到了明显的简化。

其实现的源程序代码如下:

```
#VRML V2.0 utf8
DEF yyz Shape {           #定义金色圆柱造型节点
  appearance Appearance {
    material Material {
      diffuseColor 0.5 0.3 0.0
      ambientIntensity 0.4
      specularColor 0.7 0.7 0.6
      shininess 0.2
    }
  }
  geometry Cylinder {
    height 5
    radius 0.2
  }
}
Transform {
  rotation 0 0 1 1.571      #旋转 90°
  children [
    USE yyz                 #引用金色圆柱造型节点
  ]
}
```

运行程序, 效果与图 3-2 一致。

【例 3-8】 使用旋转嵌套, 围绕 Z 轴, 每旋转 0.524rad (30°) 引用 5 次金色圆柱体造型节点, 连续引用 4 次, 构成了 6 根金色圆柱交叉的造型。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

DEF yyz Shape {           #定义金色圆柱造型节点
  appearance Appearance {
    material Material {
      diffuseColor 0.5 0.3 0.0
      ambientIntensity 0.4
      specularColor 0.7 0.7 0.6
      shininess 0.2
    }
  }
  geometry Cylinder {
```

```

    height 5
    radius 0.2
  }
}
Transform {
rotation 0 0 1 0.524
children [
  USE yyz
  Transform {
    rotation 0 0 1 0.524
    children [
      USE yyz
      Transform {
        rotation 0 0 1 0.524
        children [
          USE yyz
          Transform {
            rotation 0 0 1 0.524
            children [
              USE yyz
              Transform {
                rotation 0 0 1 0.524
                children [
                  USE yyz
                ]
              }
            ]
          }
        ]
      }
    ]
  }
}
] } } ] } } ]
}

```

运行程序，效果如图 3-7 所示。



图 3-7 重复引用造型节点效果

【例 3-9】 重复引用编组节点。首先将例 3-8 创建的金色图形用 Group 节点进行编组定义，然后作为整体造型连续 3 次用 Transform 节点进行位移调用。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
```

```

DEF yyz Group {
children [
    DEF yyy Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.5 0.3 0.0
                ambientIntensity 0.4
                specularColor 0.7 0.7 0.6
                shininess 0.2
            }
        }
        geometry Cylinder {
            height 5
            radius 0.2
        }
    }
}
Transform {
    rotation 0 0 1 0.524
    children [
        USE yyy
        Transform {
            rotation 0 0 1 0.524
            children [
                USE yyy
                Transform {
                    rotation 0 0 1 0.524
                    children [
                        USE yyy
                        Transform {
                            rotation 0 0 1 0.524
                            children [
                                USE yyy
                            ]
                        }
                    ]
                }
            ]
        }
    ]
}
}}}]]]]]]}}
Transform {
translation 2 0 -8
children [
    USE yyz
    Transform {
        translation 2 0 -8
        children [
            USE yyz
            Transform {

```

```

translation 2 0 -8
children [
    USE yyz
]
}
]
}
]
}
}

```

运行程序，效果如图 3-8 所示。

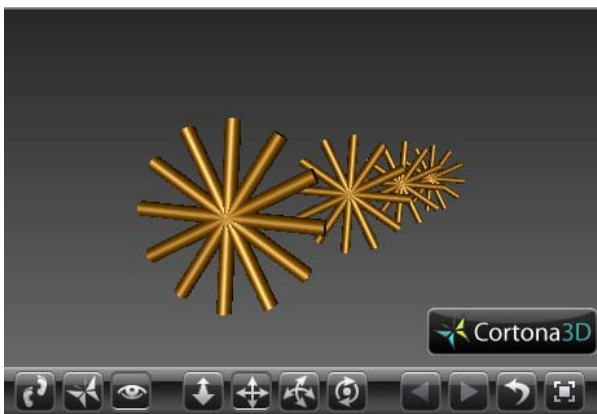


图 3-8 重复引用编组节点效果

【例 3-10】 使用重复引用节点的方法，为例 3-3 钟表造型的表盘配上 12 个刻度点。用金色小球作为小时刻度标准，每隔 0.524rad (30°) 引用一次，共 11 次。

其实现的源程序代码如下：

```

#VRML V2.0 utf8

Shape {
    #表壳造型
    appearance Appearance {
        material Material {
            diffuseColor 0.6 0.4 0.3
        }
    }
    geometry Box {
        size 4 4 2
    }
}

Transform {
    #表轴造型
    translation 0 0 1.1
    children [
        Shape {
            appearance Appearance {

```

```

        material Material {
            diffuseColor 0.7 0.3 0.1
            ambientIntensity 0.4
            specularColor 0.7 0.7 0.6
            shininess 0.2
        }
    }
    geometry Sphere {
        radius 0.1
    }
}
Transform {           #分针造型
    translation 0 0.85 0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.5 0.5 0.7
                    ambientIntensity 0.4
                    specularColor 0.8 0.8 0.9
                    shininess 0.2
                }
            }
            geometry Cylinder {
                height 1.5
                radius 0.05
            }
        }
    ]
}
Transform {           #时针造型
    translation 0.6 0 0
    rotation 0 0 -1 1.57
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.3 0.4 0.5
                    ambientIntensity 0.4
                    specularColor 0.8 0.7 0.8
                    shininess 0.2
                }
            }
            geometry Cylinder {
                height 1.0
                radius 0.05
            }
        }
    ]
}

```

```

    }
  }
]
}
DEF bbc Transform {      #表盘刻度造型
translation 0 1.8 0
children [
  Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0.5 0.4 0.3
        ambientIntensity 0.4
        specularColor 0.8 0.7 0.6
        shininess 0.2
      }
    }
    geometry Sphere {
      radius 0.07
    }
  }
]
}
Transform {
rotation 0 0 1 0.524
children [
  USE bbc
  Transform {
rotation 0 0 1 0.524
children [
  USE bbc
  Transform {
rotation 0 0 1 0.524
children [
  USE bbc
  Transform {
rotation 0 0 1 0.524
children [
  USE bbc
  Transform {
rotation 0 0 1 0.524
children [
  USE bbc

```

```

Transform {
rotation 0 0 1 0.524
children [
USE bbc
Transform {
rotation 0 0 1 0.524
children [
USEbbc
Transform {
rotation 0 0 1 0.524
children [
USEbbc
Transform {
rotation 0 0 1 0.524
children [
USE bbc  ]
}}} }}}}

```

运行程序，效果如图 3-9 所示。

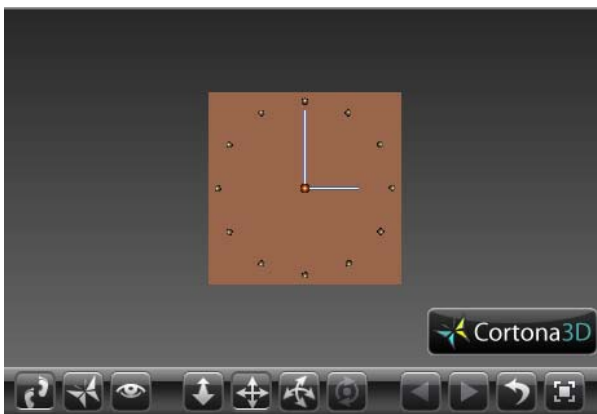


图 3-9 带刻度的钟表造型效果

3.2.4 内联节点

内联节点 (Inline) 的作用是引入外观 VRML 场景。有时由于创建的场景很复杂，使 VRML 源程序过大或过长，给程序编写和调试带来诸多不便，这时可将整个 VRML 源程序进行拆分。这就是软件工程的设计思想，采用结构化、模块化、层次化，提高软件设计质量，设计出层次清晰、结构合理的软件项目。

Inline 节点可以使 VRML 程序设计实现模块化。由基本 VRML 程序 (模块) 组成复杂和庞大的 VRML 立体静态或动态场景。Inline 是一个群节点，可以包含在其他群节点之下。同时，此节点还可以从网络中引入 VRML 文件 (程序)，可以实现分工协作，完成三维立体虚拟现实

空间场景的创建。

Inline 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Inline{	url	""	#exposesfield MFString
	bboxCenter	0.0 0.0 0.0	#SFVec3f
	bboxSize	-1.0 -1.0 -1.0	#SFVec3f
	}		

其中：

- (1) url 输入所要嵌入的 VRML 文件的路径和文件名，可以是本地计算机或网络中远程计算机文件名和位置。
- (2) bboxCenter 定义包围盒体几何中心的坐标，其含义与 Group 节点的该值相同。
- (3) bboxSize 定义包围的尺寸，其含义与 Group 节点的该域值相同。

Inline 节点调用示意图如图 3-10 所示。

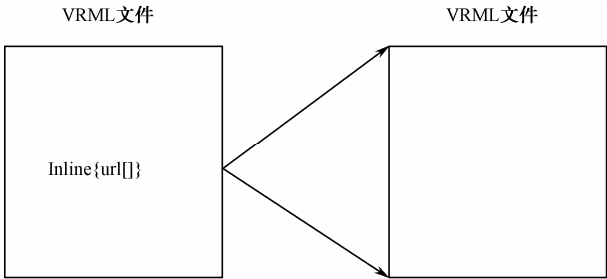


图 3-10 Inline 节点调用示意图

【例 3-11】 使用两次内联节点，将钟表造型和 Return 文本造型组成一个场景。利用内联节点调用素材（下面将进行介绍），使主程序变得非常简洁、清晰。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
Inline {
url  "li3_10.wrl"          #内联钟表造型
}
Transform {
translation 0 -3 0
children [
    Inline {
        url  "return.wrl"   #内联文本造型
    }
]
}
```

运行程序，效果如图 3-11 所示。

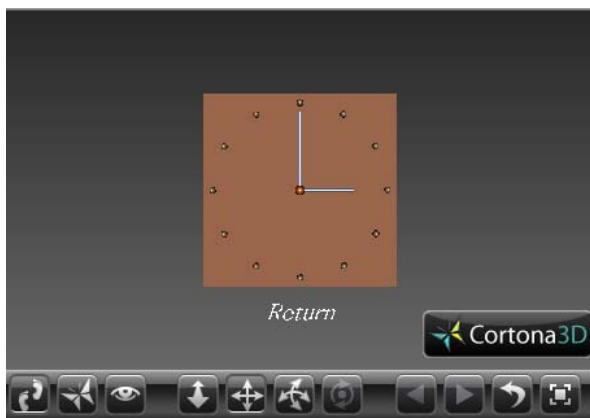


图 3-11 内联构建组场景效果

在以上程序中调用到用户编写的 `return` 文件造型，其源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0.5 0.6 0.8
          ambientIntensity 0.3
          specularColor 0.7 0.6 0.5
          shininess 0.2
        }
      }
      geometry Text {
        string "Return"
        fontStyle FontStyle {
          style "BOLDITALIC"
          justify "MIDDLE"
          size 0.5
        }
      }
    }
  ]
}
```

3.3 其他组节点使用

3.3.1 布告牌

布告牌（Billboard）是一个组节点，它可以在用户浏览时动态地改变自己的坐标系，以其坐标系的 Z 轴绕 `axisOfRotation` 轴旋转从而使其所包含的物体面向浏览者。Billboard 有着很广

泛的作用，比如产提示符、帮助信息、广告、注释、控制面板和状态显示。

可以指定布告牌组面向观察者而转动的轴，那么布告牌组的限制于围绕此轴转动。通常情况下，这是一个垂直的轴，也可以选取其他样式的轴。因为它只能围绕这个旋转轴旋转，所以假如观察者视点为正对旋转轴方向，布告牌组就没有可以跟随的转动角度了。

Billboard 节点语法格式如下：

节点名称	域名称	域值	#域及域值类型
Billboard{			
	addChildren		#eventIn SFNode
	removeChildren		#eventIn SFNode
	children		#exposedField MFNode
	axisOfRotation	0 1 0	#exposedField SFVec3f
	bboxCenter	0 0 0	#SFVec3f
	bboxSize	-1 -1 -1	#SFVec3f
}			

其中：

（1）addChildren 入事件用于将指定的节点增加到该编组节点的子节点列表中。若该节点已在子节点列表中，则该事件将被忽略。

（2）removeChildren 入事件用于将指定的节点从该编组节点的子节点列表中删除。若该节点不在子节点列表中，则该事件将被忽略。

（3）children 域的域值用于设定场景中的布告牌造型列表。这些子节点创建的造型，都受 Billboard 节点局部坐标系的影响，始终面向浏览者。该域值的默认值为空列表，即没有布告牌造型。

（4）axisOfRotation 域的域值用于设定一个旋转轴。当用户移动的时候， Billboard 节点自动地以其局部坐标系的 Z 轴围绕该轴旋转，从而保证布告牌造型始终面向浏览者。该域值的默认值为 0 1 0，表示绕 Y 轴旋转。

（5）bboxCenter 域的域值用于设定包围该编组节点所有造型的包围盒的中心点坐标。其默认值为 (0.0 0.0 0.0)，即中心点位于当前坐标原点。

（6）bboxSize 域的域值用于设定包围盒城当前坐标系 X、Y、Z 方向上的尺寸。包围盒是一个立方体。该域值的默认值为 (-1.0 -1.0 -1.0)，即不人为设置包围盒尺寸，由浏览器自动设置。

【例 3-12】 利用 Billboard 节点来做一个广告牌。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    Billboard {
      axisOfRotation 0.0 1.0 0.0
      children [
        Shape {
```

```

        appearance DEF bred Appearance {
            material Material {
                diffuseColor 0.3 0.4 0.5
            }
        }
        geometry Box {
            size 10.0 3.0 2.0
        }
    }
    Transform {
        translation 0.0 0.0 1.03
        children Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.0 1.0 1.0
                }
            }
            geometry Text {
                string "It is Billboard"
                fontStyle FontStyle {
                    justify "MIDDLE"
                    family "TYPEWRITER"
                    size 1.0
                }
            }
        }
    }
}

```

运行程序，效果如图 3-12 所示。

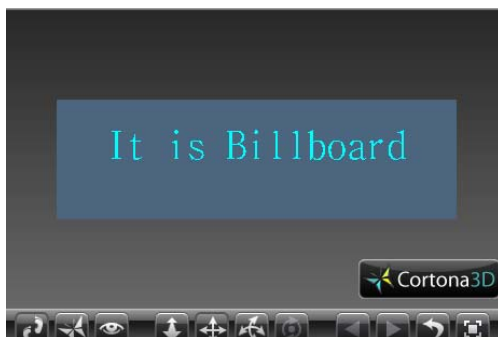


图 3-12 广告牌效果

3.3.2 开关节点

开关节点（switch）能够将一个造型不同版本组织在一起，例如在车间中机床设备的不同样式设计。通过改变 whichChoice 域的值，而不需要对 VRML 文件作太多修改，就可以在各种机床样式间迅速地变化。

switch 组长节点激活 choice 域中指定的零个或一个子项。非活动子项被忽略。但是，无论是否被激活，switch 节点的所有子项都对事件作出反应。

switch 节点语法如下：

节点名称	域名称	域值	#域及域值类型
Switch{			
	whichChoice	-1	#exposedField SFInt32
	choice	[]	#exposedField MFNode
}			

其中：

(1) whichChoice 域是活动子项的标号。choice 域中的第一个活动子项的标号是 0。如果 whichChoice 的值小于 0 或者大于 choice 域中的项目数，那么不选中任何子项。公共域的值决定创建组中的哪一个子节点。子节点的编号是 choice 域值序列中的第一个子节点，从 0 开始。这个域的默认值是-1，它指定不创建组中的任何造型。

(2) choice 域是含该组节点的各个子项。每个子项有一个隐含的序号。第一个子项的隐含序号为 0。choice 公共域的值指定组中的一系列子节点。典型的 choice 域值包括 Shape 节点和其他编组节点。VRML 浏览器通过创建一个组中包含的造型和组来创建组。这个域的默认值是一个子节点空序列。

当前子节点的选择通过向 whichChoice 公共域的隐含 set_whichChoice 或 (eventIn 入事件) 传送一个值来改变。当这个输入收到一个值后，则相应的域值就改变。同时，新值用公共域的隐含 whichChoice_changed 域 (eventOut 出事件) 输出。

组中子节点序列可用 choice 公共域的隐含 set_choice 域 (eventIn 入事件) 设置。当一系列节点传送到隐含 set_choice 域 (eventIn 入事件) 时，choice 域的子节点序列值将由输入节点序列代替。子节点序列变化后，新的子节点序列用 choice 公共域的隐含 choice_changed 域 (eventOut 出事件) 输出。

switch 节点的所有子节点将继续接收和输出事件，不管 whichChoice 域中的指定选择。

【例 3-13】 本例要求：当 whichChoice 域的域值为 0 时，显示第一个文本造型“Hellow”；当 whichChoice 域的域值为 1 时，显示第二个文本造型“Switch”；当 whichChoice 域的域值为 2 时，显示第三个文本造型“Welcome”。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
  skyColor [
    0.1 0.7 0.6
  ]
}

Switch {
  #选择开关节点
  choice [
    Shape {
      #选择序号为 0 的造型
      appearance DEF cco Appearance {
        material Material {
          diffuseColor 0.4 0.4 0.8
          ambientIntensity 0.4
```

```

        specularColor 0.7 0.7 0.6
        shininess 0.2
    }
}
geometry Text {
    string [
        "Hellow"
    ]
    fontStyle DEF ssy FontStyle {
        size 2
        style "BOLDITALIC"
        justify [
            "MIDDLE" "MIDDLE"
        ]
    }
}
}
Shape {          #选择序号 1 的造型
    appearance USE cco
    geometry Text {
        string [
            "Switch 示例"
        ]
        fontStyle USE ssy
    }
}
Shape {          #选择序号 2 的造型
    appearance USE cco
    geometry Text {
        string [
            "Welcome"
        ]
        fontStyle USE ssy
    }
}
]
whichChoice 1      #可以更改的选择序号
}

```

运行程序，效果如图 3-13 所示。



图 3-13 选择开关造型 1 的效果

3.3.3 细节层次节点

空间的细节层次节点(LOD)控制原理是通过空间距离的远近来展现空间造型的各个细节。细节层次控制和现实世界中的感观是极为相似的。在现实世界中人们都体验过，在离一个建筑很远时，只能隐约地看到一座建筑物的轮廓、形状和大小等，但是当你走近时，就会看清楚整个建筑物的窗户、门等，再走近些会看到更加清晰的内容。

在 VRML 世界里，根据不同的细节层次节点创建出不同的造型，然后再根据在 VRML 世界中视觉与立体空间造型的远近在浏览器中调用不同细节的空间造型。LOD 节点是分级型群节点，用于对相同景物做出不同精细度的表述。通过 VRML 所提供的 LOD 细节层次节点，可以将各个不同的细节穿插起来，在不同的距离调用不同的细节空间造型。因此，在创建 VRML 虚拟现实空间造型时，要平衡浏览器速度和造型的真实性两者之间的关系。造型越真实，相应的 VRML 文件就越大，就要影响浏览器的浏览速度，耗费大量的 GPU 时间。而 LOD 节点能够解决这一问题，可以对不同的景物做出不同细致程度的刻画，比较近的景物用比较精细的描述，比较远的景物用比较粗糙的描述，分级程度完全根据浏览者与景物的相对距离而定，从而提高浏览速度。

LOD 节点语法定义如下：

节点名称	域名称	域值	#域及域值类型
LOD{	level	[]	#exposedField MFNode
	center	0 0 0	#SFVec3f
	range	[]	#MFFloat
	}		

其中：

(1) **level** 域的域值是一个包含在组内的子节点的列表。该域值可以包含 **Shape** 节点和其他编组节点。在该域值中所列出的每一个子节点都分别描述了不同细节的造型，通常第一个子节点提供最高细节层次的空间造型，后面的子节点的细节层次依次降低。至于不同细节层次造型的切换，则由空间造型距离的远近来决定。其默认值为一空的子节点。

(2) **center** 域的域值定义了 LOD 节点的子节点的几何中心位置坐标。该域值与原点的距离可以用来作为不同细节层次选取的依据。其默认值为 (0,0,0)。

(3) **range** 域的域值定义了观察者与对象（造型）之间距离范围的大小。它是用来描述与空间造型的距离远近的列表，根据此列表，浏览器会从一个细节切换到另一个细节。此值必须为正，并且是顺序增长。

【例 3-14】 本例依据浏览者与造型的距离，将钟表造型分为近景、中景和远景细节层次造型。当距离小于 20 时，显示“li3_10.wrl”带刻度的钟表造型；当距离小于 40 而大于 20 时，显示“clock2.wrl”文件描述的中景；当距离大于 40 时，显示“clock3.wrl”文件描述的远景。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
Background {
    skyColor 0.0 0.6 0.4
}
LOD {
    level [
        Inline {
            url "li3_10.wrl"
        }
        Inline {
            url "clock2.wrl"
        }
        Inline {
            url "clock3.wrl"
        }
    ]
    range [20,40]      #可以改变此值来进行远近景观察
}
```

运行程序，效果如图 3-14～图 3-16 所示。

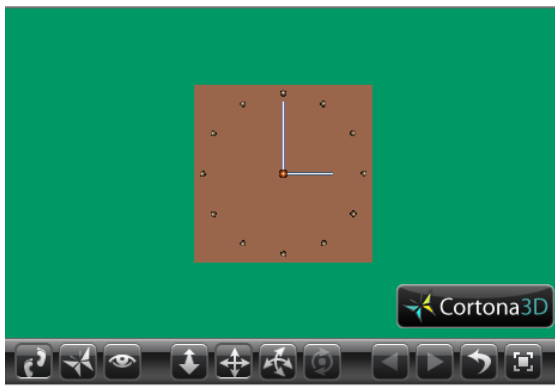


图 3-14 近景细节层次造型效果

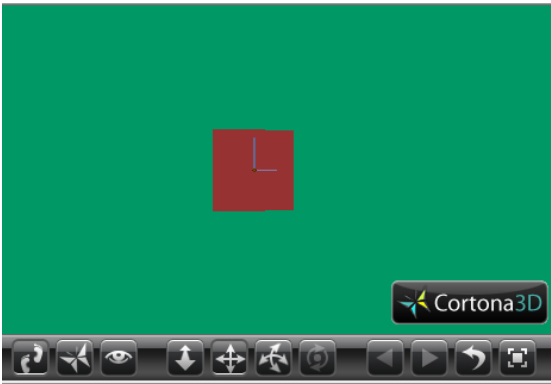


图 3-15 中景细节层次造型效果

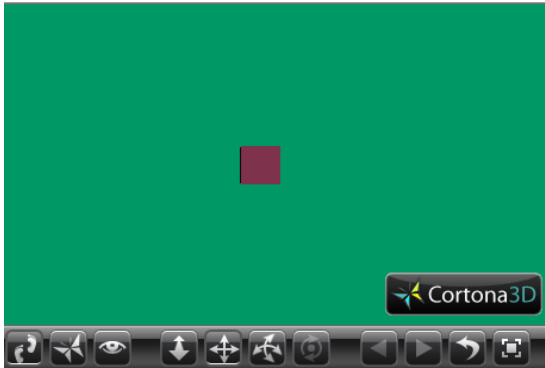


图 3-16 远景细节层次造型效果

远景细节造型文件 li3_10.wrl 程序的代码见例 3-10。
中景细节造型文件 clock2.wrl 的源程序代码如下：

```
#VRML V2.0 utf8

Shape {
    #表壳造型
    appearance Appearance {
        material Material {
            diffuseColor 0.6 0.2 0.2
        }
    }
    geometry Box {
        size 4 4 2
    }
}

Transform {
    #表轴造型
    translation 0 0 1.1
    children [
        Shape {
            appearance Appearance {
```

```

        material Material {
            diffuseColor 0.5 0.3 0
        }
    }
    geometry    Sphere {
        radius 0.1
    }
}
Transform {
    translation 0 0.85 0
    children    [
        Shape {
            appearance Appearance {
                material    Material {
                    diffuseColor 0.5 0.5 0.7
                }
            }
            geometry    Cylinder {
                height 1.5
                radius 0.05
            }
        }
    ]
}
Transform {
    #时针造型
    translation 0.6 0 0
    rotation 0 0 -1 1.57
    children    [
        Shape {
            appearance Appearance {
                material    Material {
                    diffuseColor 0.5 0.5 0.7
                }
            }
            geometry    Cylinder {
                height 1.0
                radius 0.05
            }
        }
    ]
}
]
}
}

```

远景细节造型文件 clock3.wrl 的源程序代码如下：

```
#VRML V2.0 utf8

Shape {           #表壳造型
  appearance Appearance {
    material Material {
      diffuseColor 0.5 0.2 0.3
    }
  }
  geometry Box {
    size 4 4 2
  }
}
```

对比这 3 个文件的程序代码，很容易发现一个比一个简单。可以通过激活浏览器的 plan 控制按钮，拖动鼠标改变视点与造型的距离，观察场景造型细节的变化。

3.3.4 视点

VRML 提供的视点（Viewpoint）就是在所浏览的场景中，为用户预告设定观察的位置和朝向，在这个观测点上，用户就像通过照相机的取景框一样，观察到虚拟世界的相应场景。在 VRML 中允许创建多个不同的视点供用户选择，但是这些视点不能同时使用，每个时刻只能选择其中的一个视点，随着时间的推移，视点之间可以切换。从一个视点切换到另一个视点有两种方式：跳跃式和非跳跃式。跳跃式切换的视点是一般用来定义那些能够在虚拟世界中观察到用户感兴趣的重要的精彩的场景造型的观测点，使用户不必观看每一个场景，创建一种快捷方便的观察方式，而非跳跃式切换的视点用来定义那些随坐标系平滑转换连续变化的观测点。

Viewpoint 节点用来创建观察视点，设置观察的位置、视线方向、视野范围以及视点切换方式等参数。

Viewpoint 节点的语法格式如下：

节点名称	域名称	域值	#域及域值类型
Viewpoint{	position	0.0 0.0 10.0	#exposedField SFVec3f
	orientation	0.0 0.0 1.0 0.0	#exposedField SFRotation
	fieldOfView	0.785398	#exposedField SFFloat
	jump	TRUE	#eventIn SFBool
	description	""	#SFString
	set_bind		#eventIn SFBool
	isBound		#eventOut SFBool
	bindTime		#eventOut SFTime
}			

其中：

(1) position 域的域值用于设定视点在 VRML 场景中的空间位置，由一个 X、Y、Z 的三维坐标确定。该域值的默认值为 (0.0 0.0 10.0)，即系统默认将初始视点放在 Z 轴正方向距离原点 10 个 VRML 单位长度的位置上，视线方向沿 Z 轴负方向。

(2) **orientation** 域的域值用于设定视点在 VRML 场景中的空间朝向,也就是观察者面对的方向。该域值的前 3 个值为一个 X、Y、Z 坐标分量,由原点指向该坐标点的连线为旋转轴;该域值的第四个分量是以弧度为单位的视点发生旋转的角度值。该域值的默认值为 (0.0 0.0 1.0 0.0),即视点不发生旋转。

(3) **fieldOfView** 域的域值用于设定视点视角的大小,以弧度为单位。大的视角可以产生类似广角镜头的效果,用以将造型推远;小的视角则产生类似远焦镜头的效果,用以将造型拉近。该域值的取值范围是 0.0~0.3141rad,视角超过 1.047rad (即 60°) 时观察到的物体会产生显著的变形。该域值的默认值为 0.785rad (即 45°),这与现实中人们的视野范围接近。

(4) **jump** 域的域值用于设定视点的切换类型是否为跳跃性。该域值的默认值为 **TRUE**,即为跳跃性视点,在视点发生变化时,浏览器立即将观察者的视点移动到新的位置,而不经前后两个观察点之间的任何空间。该域值若为 **FALSE**,则为非跳跃性视点,在视点发生变化时,浏览器将观察者的视点逐渐移到新的位置。

(5) **description** 域的域值用于设定一个描述视点的字符串,即该视点的名称。这些文字描述会自动出现在浏览器的视点列表中,通过该域值,人们可以很方便地找到自己感兴趣的视点。该域值的默认值为空字符。

(6) **set_bind** 入事件与 **isBound** 出事件用于进行 VRML 空间视点切换。只要向指定的 Viewpoint 节点的 **set_bind** 入事件发送 **TRUE** 值,该节点就将被设置为当前空间视点;同时原来的 Viewpoint 节点的 **isBound** 出事件将向外发送一个 **FALSE** 值,表示不再作为当前空间视点。**isBound** 出事件在浏览器到达该指定视点时,输出事件值为 **TRUE**。

(7) **bindTime** 出事件用于发出视点被切换的时间,该事件可以用来在一个给定的视点被激活时,开始运行一段动画或者执行一段脚本程序。

【例 3-15】 使用不同的视角对物体进行观察。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

Group {
  children [
    Viewpoint {
      position 0.0 0.0 3.0
      description "front"
    }
    Viewpoint {
      position 0.0 3.0 3.0
      orientation 1.0 0.0 0.0 -0.76
      description "above"
    }
    Viewpoint {
      position 0.0 0.0 -3.0
      orientation 0.0 1.0 1.0 3.14
      description "back"
      fieldOfView 1.0
    }
  ]
}
```

```
Viewpoint {
    position -3.0 -3.0 -3.0
    orientation 0.0 1.0 0.0 -2.5
    description "beside"
    fieldOfView 1.57
}
Transform {
    translation 0.0 0.0 1.0
    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.6 0.7 0.8
            }
        }
        geometry Box {
            size 0.1 0.1 0.1
        }
    }
}
Transform {
    translation -1.0 0.0 0.0
    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.8 0.5 0.2
            }
        }
        geometry Cone {
            bottomRadius 0.2
            height 0.5
        }
    }
}
Transform {
    translation 1.0 0.0 0.0
    children Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.4 0.6 0.6
            }
        }
        geometry Cylinder{
            radius 0.1
            height 0.4
        }
    }
}
```

```
    }  
  ]  
}
```

运行程序，效果如图 3-17 所示。

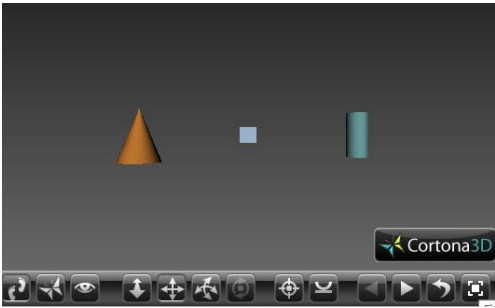


图 3-17 不同视点观察物体效果

3.3.5 锚节点

锚节点（Anchor）即超级链接群节点，它的作用是链接 VRML 三维立体空间中各个不同场景，使 VRML 世界变得更加生动有趣。还可以利用 Anchor 节点直接上网，实现真实意义上的网络世界。

Anchor 节点能够实现 VRML 场景之间的调用和互动。它是 VRML 的外部接口，可实现与 VRML 网页之间的调用及与三维之间的调用等。使用超级链接功能实现网络上任何地域或文件之间的互联、互动及感知。

Anchor 节点语法定义如下：

节点名称	域名称	域值	#域及域值类型
Anchor{	url	""	
	children	[]	#exposedField MFNode
	description	""	#exposedField MFString
	parameter	[]	#exposedField MFString
	bboxCenter	0 0 0	#exposedField SFVec3f
	bboxSize	-1 -1 -1	#SFVec3f
	addChildren		#eventIn MFNode
	removeChildren		#eventIn MFNode
	}		

其中：

（1）url 域指定需装入文件的路径或 url（统一资源定位器）。如果指定多个 url，按优先顺序进行排列，浏览器装入 url 序列中发现的第一个文件。

（2）children 域指定场景中锚节点对象。它包含指向其他文件（在 url 域中指定）的超级链接。当观察者单击其中一个对象时，浏览器便装入在 url 域中指定的文件。

（3）description 域指定一个文本字符串提示，当移动光标到锚节点对象而不单击它时，浏览器显示该提示字符串文件。

（4）parameter 为 VRML 和 THML 浏览器附加的信息，这些信息是一连串的字符串，格式

为“关键词=值”的字符串。

(5) `bboxCenter` 域指定包围 `Anchor` 节点中子节点对象的包围盒的中心。

(6) `bboxSize` 域指定包围 `Anchor` 节点中子节点对象的包围盒在 X、Y、Z 方向上的尺寸。

(7) `addChildren` 入事件用来在 `Anchor` 节点的子节点序列中加入指定的节点。如果指定子节点已经在子节点序列中，则忽略。

(8) `removeChildren` 入事件用来在 `Anchor` 节点的子节点序列中删除指定的节点。如果指定子节点已经不在子节点序列中，则忽略。

当用户选择 `Anchor` 节点中的任何一个子节点对象时，可将 `Anchor` 节点中 `url` 域中指定的文件从网上取来。如果此文件是 `*.wrl` 文件格式，则直接装入并显示它，以取代包含本 `Anchor` 节点的世界。如果取得的文件是其他类型的文档，由浏览器决定如何处理这些数据。

【例 3-16】 创建一个圆锥体，单击它跳到另一个 VRML 程序。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyColor [
        0.3 0.3 0.3]
    }
# 创建锚节点造型
Group {
    children [
        Anchor {                                     #锚节点
            children [
                Shape {
                    appearance Appearance {
                        material Material {           #空间物体造型外观
                            diffuseColor 1.0 0.0 0.0 #一种材料的漫反射颜色
                        }
                    }
                    geometry Cone {                  #锥体
                        bottomRadius 3.0
                        height 4
                    }
                }
            ]
            description "call sphere"
            url "li3_16A"
        }
    ]
}
```

运行程序，效果如图 3-18 所示。

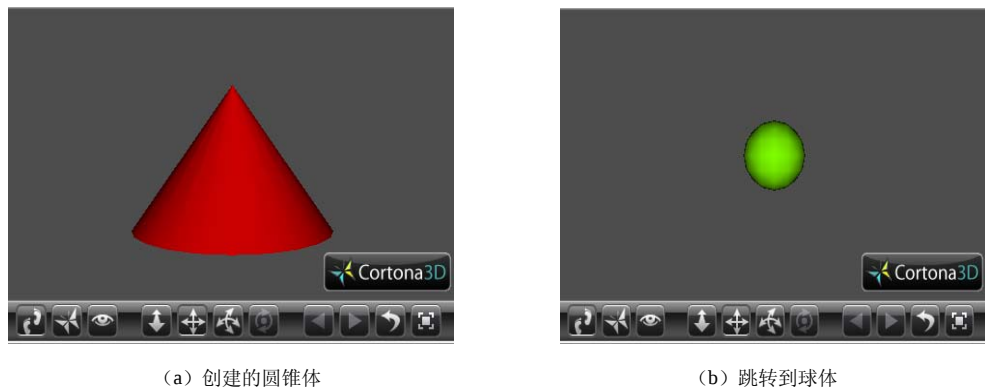


图 3-18 创建的圆锥体效果

在以上的程序中，跳转到 li3_16A 的程序的源代码如下：

```
#VRML V2.0 utf8

Background {
  skyColor [
    0.3 0.3 0.3]
}
# 创建锚节点造型
Group {
  children [
    Anchor {                                     #锚节点
      children [
        Shape {
          appearance Appearance {
            material Material {                  #空间物体造型外观
              diffuseColor 0.5 1 0.0            #一种材料的漫反射颜色
            }
          }
          geometry Sphere {                      #球体
            radius 1.0
          }
        }
      ]
      description "Goto Cone"
      url "li3_16.wrl"
    }
  ]
}
```

3.3.6 导航节点

场景导航信息（NavigationInfo）节点用来提供有关浏览者如何在 VRML 虚拟世界里导航的信息，如以移动、行走、飞行等类型进行浏览，而且该节点可提供一个虚拟现实的替身（avatar），使用该替身可在虚拟现实世界空间里遨游驰骋。

NavigationInfo 节点语法定义如下：

节点名称	域名称	域值	#域及域值类型
NavigationInfo{	avatarSize	[0.25 1.6 0 75]	#exposedField MFFloat
	headlight	TRUE	#exposedField SFFBool
	type	["WALK","ANY"]	#exposedField MFString
	spead	1.0	#exposedField SFFloat
	visibilityLinmit	0.0	#exposedField SFFloat
	set_bind		#eventIn SFFBool
	isBound		#SFFBool
	}		

其中：

(1) avatarSize 域的域值定义了三维空间中浏览者替身的尺寸。在运行 VRML 程序时，可以假设三维空间中有一个不可见的浏览者替身，通常利用该替身来进行碰撞检查。

Size 域值有以下 3 个参数：

- ① width 参数指定了替身与其他几何物体发生碰撞的最小距离。
- ② height 参数定义了替身距离地面的高度。
- ③ stepheight 参数指定替身能够跨越的最大高度，即步幅高度。

(2) headlight 域定义了替身的头顶灯开关的打开或关闭。若该域值为 TRUE，表示打开替身的头顶灯；若为 FALSE，则关闭替身的头顶灯。替身的头顶灯是由 DirectionalLight 节点创建，它相当于强度值为 1.0 的方向（平行）光。

(3) type 域的域值定义了浏览者替身的漫游（浏览）类型，该域值可在 ANY、WALK、FLY、EXAMINE、NONE 这 5 种类型中进行转换。其默认值为 WALK。

- ① WALK 表示替身以行走方式浏览虚拟世界，替身会受到重力影响。
- ② FLY 表示替身以飞行方式浏览虚拟世界，替身不会受到重力影响，可在虚拟空间飞翔、遨游。
- ③ EXAMINE 表示替身不能移动，要改变替身与物体之间的距离，只能移动物体去靠近或远离它，甚至可以围绕它旋转。
- ④ NONE 表示不提供替身导航方式。
- ⑤ ANY 表示浏览器支持以上 4 种浏览方式。

(4) speed 域的域值指定了浏览者在虚拟场景中替身行进的速度，单位为每秒多少单位长度（units/s），其默认值为 1.0（units/s）。漫游的速度也会受到浏览器设置的影响，大多数浏览器（如 BS Contact、Cosmol player）都可以通过浏览器本身的设置来改变漫游速度。当采用 EXAMINE 导航方式时，speed 域不会影响观察旋转的速度。如果 type 域设置的是 none，漫游速度将变为 0，浏览者的位置将被固定，但浏览者改变视角将不受影响。

(5) VisibilityLimit 域的域值指定了用户能够观察到的最大距离。该域值必须大于 0，其默认值为 0.0，表示最远可以观察到了无穷远处。如果浏览者在最大观察距离之内没有观察到任何对象，则将显示背景图。在构造一个大的三维立体空间场景时，其运算量是很大的，如虚拟城市，当远景看不到或可忽略时，就可以利用这个域来定义用户能够观察到的最大距离。

(6) set_bind 入事件与 isBound 出事件用于进行 VRML 空间视点切换。只要向指定的

Viewpoint 节点的 `set_bind` 入事件发送 `TRUE` 值，该节点就将被设置为当前空间视点；同时原来的 Viewpoint 节点的 `isBound` 出事件将向外发送一个 `FALSE` 值，表示不再作为当前空间视点。`isBound` 出事件在浏览器到达该指定视点时，输出事件值为 `TRUE`。

【例 3-17】 利用导航节点观察所创建的物体。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

DirectionalLight {
    direction -1 0 -1
}
NavigationInfo {
    type "WALK"
    speed 1.5
    headlight FALSE
    avatarSize [0.25 3.2 3.0]
}
Shape {
    appearance Appearance {
        material Material {
            diffuseColor 0.1 0.2 0.3
        }
    }
    geometry Box {
    }
}
```

此例中，头灯设置为 `FALSE`，但因为给了定向光，立方体仍然可见。

第 4 章 虚拟现实环境设计

虚拟现实的场景环境设计, 就是为造型创建一个仿真的外部环境, 使虚拟空间与多姿多彩的真实世界更加接近。在 VRML 中常常通过场景环境设计, 为虚拟空间创建室内外背景 (后续章节将展开介绍)、各种不同的照射光源、大气雾效果及声音效果等。

4.1 背景设计

在 VRML 中添加背景是最简单、最基本的场景环境设计方法。虚拟空间背景分为全景空间背景和天体空间背景两种: 全景空间背景是一个将造型包围在里面的空间立方体, 具有前、后、左、右、上、下 6 个平面, 可以根据需要设置不同的材质和图片; 天体空间背景是一个无穷大的空间球体, 分为天空背景和地面背景。天空和地面的划分以平线为界。地平线位于原始坐标系 XOZ 平面向后延伸的无穷远处, 上面的球体是天空, 下面的球体是地面。背景和造型的嵌套关系是: 造型在最里面, 全景空间背景的立方体为中间包围层, 天体空间背景的无穷大球体是最外面的包围层。如果同时增加了全景空间和天体空间两种背景, 可以通过设置全景空间背景的透明值决定能否看到天空和地面的颜色。默认情况下, 没有全景空间背景, 天空和地面均为黑色。

背景设计节点 (Background) 用于生成 VRML 空间的背景。利用该节点可以控制天空和地面的颜色, 设置远方山脉和城市的景象。在不必生成三维天空和远景造型的情况下, 为 VRML 世界提供了一个外部环境。它可以控制 VRML 世界中天空和地面的颜色, 指定一组全景图像旋转在 VRML 世界的上方、下方或四周。可以对天空的颜色采用梯度着色, 从顶部的蓝色到水平线处的深色或浅色。当生成天空颜色梯度时, 每一个梯度的颜色要在 Background 节点中的 skyColor 域中定义。从正上方开始, 直到正下方为止。除第一个颜色以外, 对每一个颜色要在 skyAngle 域中提供一个角度。每一个角度是从垂直向上的轴开始度量。在角度的位置对应一种颜色。角度之间的部分的颜色从一种到另一种平滑过渡, 形成颜色梯度。采用天空角, 可以精确地指定从头顶到地平线乃至再向下颜色的变化。第一个颜色总是用来生成天空球体正上方部分的颜色。第二个颜色放置在由第一个天空角指定的位置。在正上方和第一个天空角位置之间的颜色由第一个颜色到第二个颜色平滑过渡形成。同样第三个颜色放置在第二个天空角位置之间的颜色, 依此类推。如果有 n 种颜色, 将要提供 $n-1$ 个天空角。可以指定正下方 (180°) 的天空颜色。如果指定角度小于 180° , 将采用最后的颜色对天空球体接近正下方其他部分着色。VRML 的地面也是一个环绕世界的无限大的球体。采用与天空球体类似的特征, 可以指定地面球体的颜色梯度。天空和地面为 VRML 世界提供了基本的背景颜色。在该背景上可以添加背景图像以在地平线上生成山脉或城市的图像。

VRML 文件中的第一个 Background 节点可作为初始背景, 而其他的 Background 节点在接收到 set_bind 事件后才显示出来。

Background 节点可以是任意组节点的子节点, 它在当前坐标系中构造背景。但是背景仅受坐标系旋转变化的影响。而不受平移或缩放变换的影响。浏览者永远不可能靠近背景, 但是可以转到不同角度去观察全景图像的不同侧面。

从概念上来说地面和天空背景由一个半径无限大的球面构成，光滑梯度进行变化的地面颜色和天空颜色涂画在球面上。

全景图像的透明度值可以决定透过全景图像是否可以看见地面颜色和天空颜色。一般并不指定 topUrl 和 bottomUrl 图像，从而可使地面和天空透过全景凸现。即它的 4 个图像可以描绘山脉或者其他远景。默认的情况下，没有全景图像。

VRML 浏览器保存一个 Background 节点栈。栈顶点是处于激活状态的节点。给 Background 节点的 set_bind 输入事件赋 TRUE 值，就可将此 Background 节点压至栈顶，新被激活的 Background 节点的背景颜色和全景图像将取代老的背景图像。若给背景节点的 set_bind 事件赋 FALSE 值，就可以将删除此背景节点处，在栈中紧接其下的背景节点成为激活节点。如果堆栈已空，则使用默认的 Background 节点。VRML 世界只有一个背景。如果要改变背景，如从明亮的中午到橘黄的落日，可以生成多个 Background 节点，并利用背景连接，在 Background 节点之间进行切换。在一个时间只有一个 Background 节点在使用。

Background 节点语法如下：

节点名称	域名称	域值	#域及域值类型
Background{	set_bind		#eventIn SFBool
	groundColor	[]	#exposedField MFColor
	groundAngle	[]	#exposedField MFFloat
	skyColor	[0 0 0]	#exposedField MFColor
	skyAngle	[]	#exposedField MFFloat
	frontUrl	""	#exposedField MFString
	backUrl	""	#exposedField MFString
	rightUrl	""	#exposedField MFString
	leftUrl	""	#exposedField MFString
	topUrl	""	#exposedField MFString
	bottomUrl	""	#exposedField MFString
	isBound		#eventOut SFBool
	}		

其中：

(1) 公共域 skyColor 的值指定了一系列用于天空着色的 RGB 颜色。该域的默认值为单一的黑色。

(2) 公共域 skyAngle 的值指定了着色所需的球体角。该角从正上方开始为 0.0°(0.0rad)，到球体赤道部分为 90°(1.571rad)，到正下方为 180°(3.141rad)。天空角必须以升序排列。如果最后的天空角小于 180°，将用最后的天空颜色为其余部分到正下方着色。该域的默认值为空。用于指定天空球体着色的 RGB 值。所列出的第一个值是当竖直向上看时该点的颜色。skyColor 域所含数据的个数应该比 skyAngle 的多一个。对应于天空颜色值，给出一组转角值（浮点数，取值单位为 rad）。在本域内隐含的规定起始为 0rad（处于垂直向上的位置），而将终点隐含的规定为 π （处于垂直向下的位置）。如果最后一个值小于 180°，则最后的 skyColor 被用来填充天空球体的剩余部分（从最后一个 skyAngle 到 π 的范围）。在任意两个天空角之间的部分，将形成颜色梯度。颜色从一个天空所对应的颜色平滑过渡到另一个天空角所对应的颜色，如果有 n 种颜色，则有 n-1 个天空角。skyColor 和 skyAngle 的默认值生成一个黑色的天空球体。

(3) `groundColor`、`groundAngle` 公共域与 `skyColor`、`skyAngle` 域的作用方式类似。如果有 n 种地面颜色,那么就必须有 $n-1$ 个地面角。`GroundColor` 和 `groundAngle` 域的默认值生成一个透明的地面球体,用于指定地面球体着色的 RGB 值。可以设置一系列颜色以产生按水平分层渐变的效果,其中所列出的第一个值是指竖直向下看到的那个点的颜色。

(4) `frontUrl`、`backUrl`、`rightUrl`、`leftUrl`、`topUrl` 和 `bottomUrl` 域为背景立方体 6 个面指定不同的图像。通过没有背景图像的任意面,可以看到外部的地面和天空。通常在顶部和底部不设置背景图像,可以看到天空和地面。公共域 `frontUrl`、`backUrl`、`rightUrl`、`leftUrl`、`topUrl` 和 `bottomUrl` 分别指定在当前坐标系中,围绕观察者的背景立方体的各个面所对应的图像。该立方体的前面在 Z 轴的负半轴,背面在 Z 轴的正半轴。同样,左面沿 X 轴的负半轴,右面沿 X 轴的正半轴,上面沿 Y 轴的正半轴,底面在 Y 轴的负半轴。

每一个 `url` 域指定一组有优先级的 URL,优先级从高到低。VRML 浏览器首先试图打开由该组中第一个 URL 指定的文件。如果没有找到该文件,VMRL 浏览器将尝试组中第二个 URL,依此类推,如果找到一个可以被打开的 URL。文件中的图像被读出,并映射到背景立方体上对应的一面。如果没有 URL 可以被打开,就没有图像映射发生。这些域的默认值为 URL 的空列表。分别指定将被映射到空间立方体各个面上的图像。该空间立方体的六个面形成了一组包围世界中所有几何体的全景图像。映射的图像多为一些山脉、摩天大楼或云彩等远景。

背景图像一定要是 JPEG、GIF 或 PNG 图像格式。这些图像格式对每一个像素点提供了色彩或灰度值,还包含每一个像素点的像素透明值。

如果一个背景图像包括像素透明值,背景图像的透明部分可以使外部地面球体和天空球体显示出来。如果对背景立方体的一面或多面没有提供背景图像,就可以通过这些部分看到外部的地面和天空。一般上面和底面没有图像。在前、后、左、右各面的图像均使用像素透明值使山脉或城市之后的天空显示出来。

天空球体、地面球体以及背景立方体不被 VRML 世界中的光源所影响。所有这三者的明暗处理均认为其采用了辐射材料。

`Background` 节点可以是任意组节点的子节点。它在当前坐标系中构造背景。但是背景仅受坐标系旋转变化的影响,而不受缩放变换的影响。这可以保证观察者可以看到球体和背景立方体的不同部分,而不接受它们。

VRML 浏览器中维护一个含 `Background` 节点的栈。在该堆栈顶部的节点被连接。VRML 浏览器使用该节点控制当前世界的背景着色。通过向 `set_bind` 或 (`eventIn` 入事件)发送 `TRUE` 或 `FALSE`,可使 `Background` 节点放置到堆栈顶部或从堆栈中移出。当一个 `Background` 节点的接收到一个值时,相应的处理依赖于所接收到的值和该节点在堆栈中的位置。

当向一个特定的 `Background` 节点的 `set_bind` 或 (`eventIn` 入事件)发送一个 `TRUE`,而这个 `Background` 节点就在堆栈的顶部时,不作任何动作。否则将产生如下动作:

- ① 在顶部的 `Background` 节点使用其 `isBound` 域 (`eventIn` 入事件)发送 `FALSE`。
- ② 新的 `Background` 节点移动堆栈的顶部并且成为当前的背景。
- ③ 新的当前 `Background` 节点利用其 `isBound` 或 (`eventOut` 出事件)发送 `TRUE`。

当向一个特定的 `Background` 节点的 `set_bind` 或 (`eventIn` 入事件)发送一个 `FALSE`,并且该指定的 `Background` 节点不在背景堆栈中,没有动作发生。如果该 `Background` 节点在堆栈中,而不在堆栈顶部时,该节点被从堆栈中移出。如果该节点在堆栈的顶部时,产生以下动作:

- ① 指定的 Background 节点利用 isBound 域（eventOut 出事件）发送 FALSE。
- ② 指定的 Background 节点被从背景堆栈中移出。
- ③ 在堆栈中的下一个节点成为当前新的背景。
- ④ 新的当前 Background 节点利用其 isBound 域（eventOut 出事件）发送 TRUE。

在上述情况下，当在堆栈顶部的节点和曾在堆栈顶部的节点都要使用各自的 isBound 域（eventOut 出事件）发送值。这些值同时被发出。如果由于从堆栈中移出所有的 Background 节点而造成堆栈为空时，当前没有 Background 节点连接。在这种情况下 VRML 浏览器使用默认的 Background 节点值控制 VRML 世界的背景着色。

当一个 VRML 空间由浏览器读入时，在最顶层文件中，不包括在任意的嵌入文件中，遇到的第一个 Background 节点自动移到堆栈的顶部并连接。如果在最高层文件中没有 Background 节点，背景堆栈一直为空，直到通过线路的动作将一个 Background 节点连接。

(5) set_bind 是 eventIn 入事件，值类型是 SFBool，指定此事件为 TRUE 可使该 Background 节点处于激活状态。

(6) isBound 是 eventOut 出事件，值类型是 SFBool，指明该背景节点处于激活状态 (TRUE) 或非激活状态 (FALSE)。

通过向隐含 set_groundColor、set_groundAngle、set_skyColor、set_skyAngle、set_frontUrl、set_backUrl、set_leftUrl、set_topUrl 和 set_bottomUrl 域（eventIn 入事件）发送值可以改变背景图像和颜色。当这些输入接收到值时，对应域的值产生变化，新的值由公共域的隐含 groundColor_changed、groundAngle_changed、skyColor_changed、skyAngle_changed、frontUrl_changed、backUrl_changed、leftUrl_changed、topUrl_changed 和 bottomUrl_changed 域（eventOut 出事件）输出。

【例 4-1】 使用 Background 节点绘制天空与地面背景图。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    # 环境背景
    Background {
      skyColor [0.0 0.2 0.7,0.0 0.5 1.0,1.0 1.0 1.0]
      skyAngle [1.309,1.571]
      groundColor [0.1 0.0 0.0,0.4 0.25 0.2,0.6 0.6 0.6]
      groundAngle [1.309 1.571]
    }
    Transform {
      translation -2.0 -2.0 0.0
      children [
        Shape {
          appearance Appearance {
            material Material {
            }
          }
          geometry Text {
```

```

                                string "VRML Ground"
                                fontStyle FontStyle    {
                                    size 1
                                }
                            }
                        }
                    ]
                }
            Transform {
                translation -2.0 -1.0 0.0
                children [
                    Shape {
                        appearance DEF green Appearance    {
                            material Material {
                                diffuseColor 0.0 1.0 0.0
                            }
                        }
                    }
                    Transform {
                        translation 0.0 3.0 0.0
                        children Shape {
                            appearance USE green
                            geometry Text {
                                string "VRML Sky"
                                fontStyle FontStyle    {
                                    size 1
                                }
                            }
                        }
                    }
                ]
            }
        }
    }
}

```

运行程序，效果如图 4-1 所示。



图 4-1 绘制的天空与地面背景图效果

【例 4-2】 本例创建了一个高尔夫球场全景空间。在设置天空地面外围背景的基础上，增设了左、右、前、后四幅侧面背景图片，上、下两个没有设置，构成了一个可以看到天空和地面的立方体内层空间背景。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyAngle [                #设置天空角
        1.2 1.57
    ]
    skyColor [                #设置天空角对应的颜色
        0 0 1
        0 0.5 0.8
        1 1 1
    ]
    groundAngle [            #设置地面角
        1.2 1.571
    ]
    groundColor [            #设置地面角对应的颜色
        0.1 0.1 0.1
        0.4 0.3 0.2
        0.8 0.8 0.8
    ]
    frontUrl "GG01.jpg"      #设置前景图片
    rightUrl "GG02.jpg"      #设置右景图片
    leftUrl "GG03.jpg"       #设置左景图片
    backUrl "GG04.jpg"       #设置后景图片
}
```

运行程序，效果如图 4-2～图 4-5 所示。



图 4-2 全景空间背景 1 效果



图 4-3 全景空间背景 2 效果



图 4-4 露出天空的背景效果

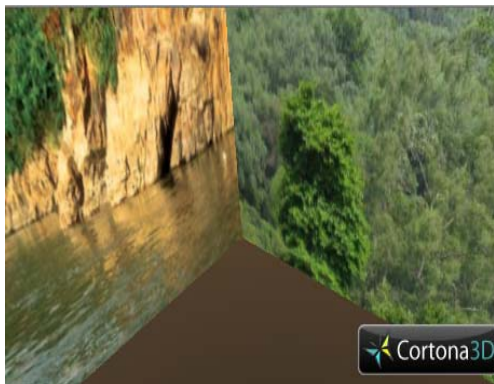


图 4-5 露出地面的背景效果

4.2 光源创建

现实世界中的光源来自于各种能发光的物体，如太阳、灯泡、激光器等。在 VRML 中创建光源，使造型在场景环境中产生光照效果，让虚拟世界更逼真、更生动。在默认情况下，VRML 浏览器自动生成一个白色灯光源 `headlight`。此光源为平行光束，与浏览者的视点同步运动，始终照亮浏览者的前方，和井下工作的矿工帽子上的头灯很相似。头灯光源可以通过设置 `NavigationInfo` 节点的 `headlight` 域的域值，控制光源打开或关闭 (`FALSE`)，默认情况下，头灯为打开状态，光源的颜色无法改变。本章之前所有的实例均采用这种默认的白色头灯光源。除此之外，VRML 中还可以人工设置 3 种类型的光源：点光源 (`PointLight`)、平行光源 (`DirectionalLight`) 和锥光源 (`SpotLight`)。点光源与常见的灯泡类似，由一个发光点向整个空间发射光线。平行光源与激光或太阳光类似，始终朝一个特定方向发射光线，在场景中产生一组完全平行的光照效果。锥光源类似于带罩的台灯，由一个发光点向一个特定方向照亮圆锥体。光源的颜色可以在 `color` 域中，由一个 RGB 颜色域值进行设置。VRML 创建的光源与现实世界中的光源之间最大的差异是无法自动产生阴影，必须通过人为设置阴影造型模拟阴影效果。

4.2.1 法线

面的法向量被看作是垂直于一个面并指向面之外的一个箭头。如果一个面朝向右侧，其法向量也指向右侧。如果一个面朝向上，其法向量也指向上方。由点坐标定义面的法向量，VRML 通常将坐标次序显示为逆时针方向的一侧作为该面的前面，即逆着面的法向量看去该面的坐标点次序是逆时针方向。

VRML 根据面的法向量与面与光源连线的夹角确定面的亮度。角度越大，面的亮度越弱；角度越小，面的亮度越大。当表面法向量离观察者越来越远时，该表面也将变得越来越暗，该表面法向量与光线之间的角度越来越大。上述法向量角度和明暗程度之间的联系是 VRML 浏览器用来计算该世界中明暗的依据。面的法向量通常是由 VRML 浏览器自动完成。

当 `IndexedFaceSet` 和 `ElevationGrid` 节点对造型进行明暗控制时，可以为它们的各个面指定法向量来控制各面的明暗效果。此时需要一个法向量列表（即为节点的 `normal` 域指定一个法线节点 (`Normal`)），将列表中的法向量与造型的坐标顶点或构造面相联系，也就是用法向量列表的索引将法向量指定给各坐标顶点或构造面。

法向量列表与某些几何节点用来指定造型坐标的坐标列表相似。在法向量列表中的每个法

向量是由 X, Y, Z 分量构成的单位向量。以 0 开始的法向量索引表示第一个法向量, 1 表示第二个, 依次类推。

如为 ElevationGrid 节点提供法向量, 法向量列表中的列表自动分配给海拔栅格中的每一部分。可以控制这种自动分配是应用于整个栅格方块还是单独的栅格点。当法向量用于整个栅格方块时, 列表中的第一个法向量控制第一个栅格方块的明暗, 第二个法向量对第二个方块, 依此类推; 当法向量用于单独的栅格点时, 列表中的第一个法向量控制第一个栅格点的明暗, 第二个法向量对第二个栅格点的明暗, 依此类推。ElevationGrid 节点自动使各角落之间的亮度平滑变化。

Normal 节点的语法如下:

节点名称	域名称	域值	#域及域值类型
Normal{			
	vector	[]	#exposedField MFVec3f
}			

其中, vector 公共域的值指定一个可用作造型法向量的单位法向量矢量列表。每个法向量被说明为三个浮点值, 分别为向量的 X、Y、Z 分量。vector 域的省略值为一空法向量列表。当法向量索引被几何节点使用时, Normal 节点列表中的第一个法向量索引为 0, 第二个索引为 1, 依次类推。

发送一个值到 vector 公共域的隐含 set_vector 域 (eventIn 入事件), 可以改变向量列表的值, 当输入收到此值时, 法向量矢量列表被改变。新的法向量矢量列表由此公共域的隐含 vector_changed 域 (eventOut 出事件) 输出。法向量应为单位长。若未指定法向量, 则大多数浏览器能为指定几何体产生默认法向量。

【例 4-3】 下面为一个正三角形面片指定法向量。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.0 1.0 1.0
    }
  }
  geometry IndexedFaceSet {
    coord Coordinate {
      point [
        -1.0 -1.0 0.0,1.0 -1.0 0.0
        -1.0 1.0 0.0,-1.0 1.0 0.0
      ]
    }
    coordIndex [0 1 2 3]
    normal Normal {
      vector [
        1.0 0.0 0.0,0.0 0.0 1.0
```

```
0.0 0.0 1.0,0.0 0.0 1.0
    ]
    }
    normalIndex    [0 1 2 3]
  }
}
```

运行程序，效果如图 4-6 所示。



图 4-6 指定法向量的正方形面片效果

4.2.2 点光源

点光源由发光点向四面八方发射亮度相同的光线，各个方向具有相同性。点光源节点既可作为独立的节点，也可作为其他组节点的子节点使用。

PointLight 节点语法如下：

节点名称	域名称	域值	#域及域值类型
PointLight{			
	on	TRUE	#exposedField SFBool
	location	0.0 0.0 0.0	#exposedField SFVec3f
	radius	100.0	#exposedField SFFloat
	intensity	1.0	#exposedField SFFloat
	ambientIntensity	0.0	#exposedField SFFloat
	color	1.0 1.0 1.0	#exposedField SFColor
	attenuation	1.0 0.0 0.0	#exposedField SFVec3f
}			

其中：

- (1) on 域的域值用于设定点光源处于打开状态还是关闭状态。该域值设置为 FALSE 点光源关闭。该域值的默认值为 TRUE，即点光源打开，对场景中的造型产生光照效果。
- (2) location 域的域值用于设定点光源在当前坐标中的位置，由一个 X、Y、Z 三维空间坐标值来确定。其默认值为 (0.0 0.0 0.0)，即点光源位于当前坐标的原点。
- (3) radius 域的域值用于设定点光源的辐射半径值。点光源的照亮范围是以光源的位置为中心、以此域值为半径构成的圆球体空间，圆球体以外的区域无法被点光源照亮，该域值的默

认为 100.0，即照亮区域的半径为 100 个 VRML 单位。

(4) **intensity** 域的域值用于设定点光源的明亮程度。该域值的取值范围为 0.0~1.0，0.0 表示最暗，1.0 表示最亮。该域值的默认值为 0.0。

(5) **ambientIntensity** 域的域值用于设定点光源对照明球体内的造型所产生的环境光线的影响。该域值的取值范围为 0.0~1.0，0.0 表示该光源对环境光线没有影响，1.0 表示该光源对环境光线的影响最大。该域值的默认值为 0.0。

(6) **color** 域的域值用于设定点光源的 RGB 颜色。该域值的默认值为 (1.0 1.0 1.0)，表示生成一个白色的光源。

(7) **attenuation** 域的域值用于设定在光照范围内光线的衰减方式。该域值是由 3 个控制参数组成的集合。第一个数值用于控制该光源在照明球体中是否光线恒定；第二个数值用于控制光线按线性方式衰减，即随着距离的增加，光线的亮度逐渐减弱；第三个数值用于控制光线亮度衰减和距离平方之间的关系。该域的所有值要大于或等于 0.0。该域值的默认值为 (1.0 0.0 0.0)，表示在照明球体中光线亮度保持恒定。

【例 4-4】 创建一个球体的点阵，在横向与纵向方向共有 5 个球。
其实现的源程序代码如下：

```
#VRML V2.0 utf8

DEF spp
Transform {
  translation -3.0 0.0 0.0
  children [
    DEF sp Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1.0 1.0 0.0
        }
      }
      geometry Sphere {
        radius 0.4
      }
    }
  ]
  Transform {
    translation 1.5 0.0 0.0
    children [
      USE sp
      Transform {
        translation 1.5 0.0 0.0
        children [
          USE sp
          Transform {
            translation 1.5 0.0 0.0
            children [
              USE sp
```

```

                                Transform {
                                    translation 1.5 0.0 0.0
                                    children [
                                        USE spp    ]  }
                                ]}
                            ] }
                    ]}
    ]}
    Transform {
        translation 0.0 3.0 0.0
        children [
            USE spp
            Transform {
                translation 0.0  -1.5 0.0
                children [
                    USE spp
                    Transform {
                        translation 0.0 3.0 0.0
                        children [
                            USE spp
                            Transform {
                                translation 0.0 -1.5 0.0
                                children [
                                    USE spp    ]}
                            ]}
                        ]}
                    ]}
                ]}
            ]}
        ]}
    ]}

```

运行程序，效果如图 4-7 所示。

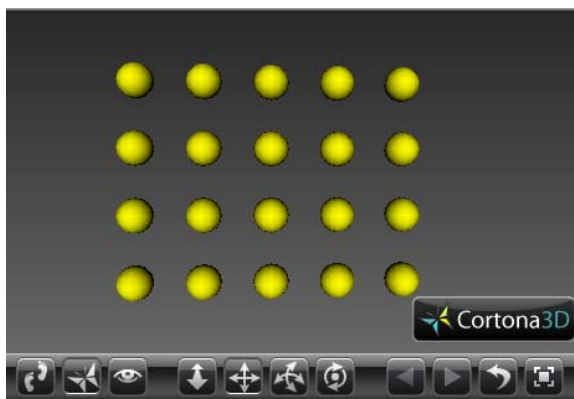


图 4-7 创建的球阵效果

【例 4-5】 在球体的点阵加入一个点光源。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
```

```
Transform {
    scale 0.2 0.2 0.2
    children [
        Inline {
            url "LI4_4.wrl"
        }
    ]
}

PointLight {
    ambientIntensity 1.0      #点光源节点
    attenuation 1 0 0        #点光源向四周照射强度
    location 0 0 0            #光线衰减方式
    color 1.0 0.0 0.0        #光源的颜色
    radius 8.0                #光源的颜色
    on TRUE                   #点光源的开关
}
```

运行程序，效果如图 4-8 所示。

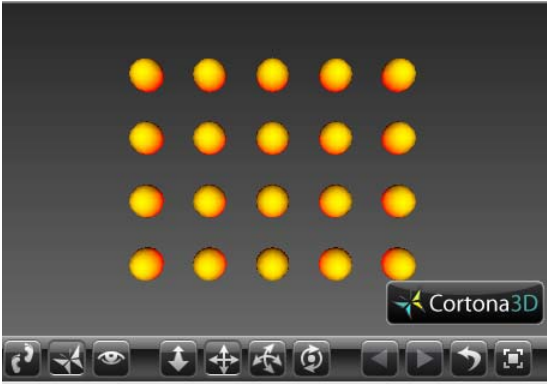


图 4-8 球阵中加入点光源的效果

4.2.3 平行光源

现实世界中的平行光源是从无穷远处照射，因此其光线是平行指向同一个方向的。例如，太阳基本是一个平行光源，太阳离地球相当远，当太阳的光线达到地球时基本是平行的。在 VRML 中，利用 DirectionalLight 节点，创建一种光线沿某一个特定方向平行传播的光源。

DirectionalLight 节点语法如下：

节点名称	域名称	域值	#域及域值类型
DirectionalLight{	on	TRUE	#exposedField SFBool
	direction	0.0 0.0 -1.0	#exposedField SFVec3f
	intensity	1.0	#exposedField SFFloat
	color	1.0 1.0 1.0	#exposedField SFCOLOR
}			

其中:

(1) **on** 域的域值用于设定该光源处于打开状态还是关闭状态。该域值设置为 **FALSE**, 光源关闭。该域值的默认为 **TRUE**, 即光源打开, 对场景中的造型产生光照效果。

(2) **direction** 域的域值用于设定平行光源的映射方向。该域值设置一个 **X、Y、Z** 三维空间坐标, 在当前坐标中确定一个点的位置, 光源映射的方向由坐标原点指向这个特定的点。若 **direction 1 0 0**, 则表示平行光线由坐标原点指向 **X** 轴正方向。其默认值为 **(0.0 0.0 -1.0)**, 即平行光线由坐标原点指向 **Z** 轴负方向。

(3) **intensity** 域的域值用于设定光源的明亮程度。该域值的取值范围为 **0.0~1.0**, **0.0** 表示最暗, **1.0** 表示最亮。该域值的默认值为 **1.0**。

(4) **color** 域的域值用于设定光源的 **RGB** 颜色。该域值的默认值为 **(1.0 1.0 1.0)**, 表示生成一个白色的光源。

【例 4-6】 点光源定义在一个固定的三维位置上。点光源在各方向上的亮度相等, 即点光源是各向同性的。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

Group {
  children [
    PointLight {
      location 2.0 2.0 0.0
      radius 50.0
    }
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0.5 0.8 0.8
        }
      }
    }
    DEF qiiy Transform {
      translation 0.0 0.0 0.0
      children [
        DEF qii Transform {
          translation 0.0 0.0 0.0
          children [
            DEF qi Shape {
              appearance DEF bred Appearance {
                material Material {
                  diffuseColor 0.5 0.5 0.5
                }
              }
            }
            geometry Sphere {
              radius 0.35
            }
          ]
        }
      ]
    }
  ]
}
```

```

        }
    }
    Transform {
        translation 1.0 0.0 0.0
        children USE qi
    }
    Transform {
        translation 2.0 0.0 0.0
        children USE qi
    }
    Transform {
        translation 3.0 0.0 0.0
        children USE qi
    }
    ]
}
Transform {
    translation 0.0 1 0.0
    children USE qii
}
Transform {
    translation 0.0 2.0 0.0
    children USE qii
}
Transform {
    translation 0.0 3.0 0.0
    children USE qii
}
]
}
Transform {
    translation 0.0 0.0 1.0
    children USE qiiy
}
Transform {
    translation 0.0 0.0 2.0
    children USE qiiy
}
Transform {
    translation 0.0 0.0 3.0
    children USE qiiy
}
]
}

```

运行程序，效果如图 4-9 所示。

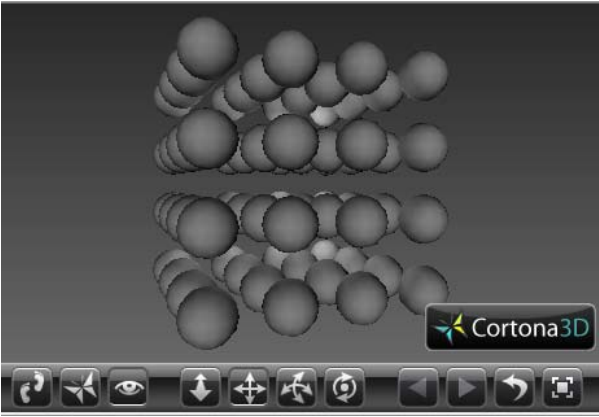


图 4-9 在球阵中分别加入平行光源的效果

4.2.4 锥光源

锥光源即光线照射呈光锥状，光源位于圆锥体的顶点，光线被限制在一个圆锥体的照射空间里，造型只有位于圆锥体空间中才会被照亮。在 VRML 中，常常利用 SpotLight 节点创建锥光源，模拟一些特殊光照效果的场景，如舞台灯光、探照灯、艺术摄影等。

SpotLight 节点语法如下：

节点名称	域名称	域值	#域及域值类型
DirectionalLight{	on	TRUE	#exposedField SFBool
	direction	0.0 0.0 -1.0	#exposedField SFVec3f
	intensity	1.0	#exposedField SFFloat
	color	1.0 1.0 1.0	#exposedField SFColor
	location	0.0 0.0 0.0	#exposedField SFVec3f
	radius	100.0	#exposedField SFFloat
	beamWidth	1.570796	#exposedField SFFloat
	cutOffAngle	0.785398	#exposedField SFloat
	ambientIntensity	0.0	#exposedField SFFloat
	attenuation	1.0 0.0 0.0	#exposedField SFVec3f
	}		

其中：

- (1) on 域的域值用于设定该光源处于打开状态还是关闭状态。该域值设置为 FALSE，光源关闭。该域值的默认值为 TRUE，即光源打开，对场景中的造型产生光照效果。
- (2) direction 域的域值用于设定锥光源的照射方向。该域值设置一个 X、Y、Z 三维空间坐标，在当前坐标中确定一个点的位置，光源照射的方向由坐标原点指向这个特定的点。若 direction 1 0 0，则表示光线由坐标原点指向 X 轴正方向。其默认值为 (0.0 0.0 -1.0)，即光线由坐标原点指向 Z 轴负方向。
- (3) intensity 域的域值用于设定光源的明亮程度。该域值的取值范围为 0.0~1.0，0.0 表示最暗，1.0 表示最亮。该域值的默认值为 1.0。
- (4) color 域的域值用于设定光源的 RGB 颜色。该域值的默认值为 (1.0 1.0 1.0)，表示生成一个白色的光源。

(5) **location** 域的域值用于设定光源在当前坐标中的位置，由一个 X、Y、Z 三维空间坐标值来确定。其默认值为 (0.0 0.0 0.0)，即光源位于当前坐标的原点。

(6) **radius** 域的域值用于设定锥光源的照射半径值。该域值设置光线照射的距离。该域值的默认值为 100.0，即照亮距离为 100 个 VRML 单位。

(7) **beamWidth** 域的域值用于设定在照明锥体中所包含的一个等强度光内的锥体的扩散角，即锥体中心轴到内部锥体表面所形成的夹角。内部锥体中的强度处处相等，保持不变，而内部锥体到照明锥体之间的区域内，光照强度逐渐衰减直到为零。该域值以弧度为计量单位，在 0.0~1.571 范围内变化，其默认值为 1.571 (90°)。

(8) **cutOffAngle** 域的域值用于设定整个照明锥体的扩散角，即锥体中心轴到照明锥体表面所形成的夹角。增大该域值可以扩大照明锥体的空间，位于锥体中的造型将被此锥光源照亮。该域值以弧度为计量单位，在 0.0~1.571 范围内变化，其默认值为 0.785 (45°)。

(9) **ambientIntensity** 域的域值用于设定锥光源对照明球体内的造型所产生的环境光线的影响。该域值的取值范围为 0.0~1.0，0.0 表示该光源对环境光线没有影响，1.0 表示该光源对环境光线的影响最大。该域值的默认值为 0.0。

(10) **attenuation** 域的域值用于设定在光照范围内光线的衰减方式。该域值是由三个控制参数组成的集合。第一个数值用于控制该光源在照明锥体中是否光线恒定；第二个数值用于控制光线按线性方式衰减，即随着距离的增加光线的亮度逐渐减弱；第三个数值用于控制光线亮度衰减和距离平方之间的关系。该域值的所有值要大于或等于 0.0。该域值的默认值为 (1.0 0.0 0.0)，表示在照明锥体中光线亮度保持恒定。

锥体光源照射示意图如图 4-10 所示。

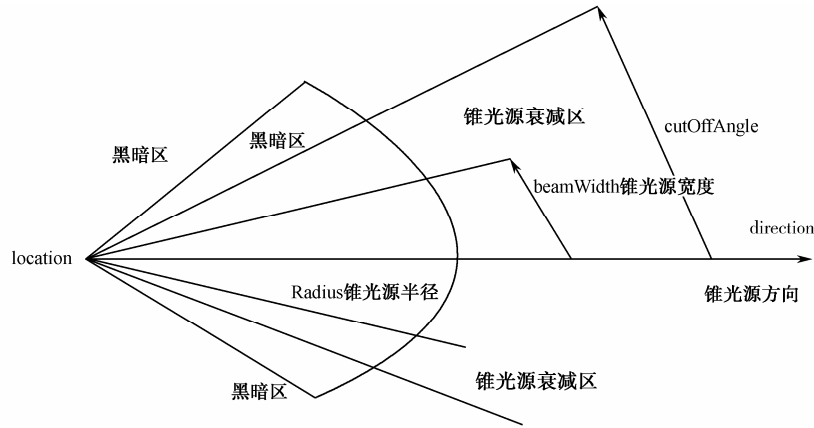


图 4-10 锥体光源照射示意图

【例 4-7】 为了更好地显示锥体光源的照射效果，将被照造型组按 5×5×5 个小球组成正方体球阵，并添加聚光源。
其实现的源程序代码如下：

```
#VRML V2.0 utf8

NavigationInfo {           #导航节点
    headlight FALSE         #关闭头灯
}
```

```

SpotLight {                                #锥光源节点
    location 0 0 5                          #位置
    beamWidth 0.2                          #锥光源的等强度内部锥体扩散角
}
DEF a Group {
    children [
        DEF g Group {
            children [
                DEF ball Shape {
                    appearance Appearance {
                        material Material {
                        }
                    }
                    geometry Sphere {
                        radius 0.5
                    }
                }
                Transform {
                    translation 1.5 0 0
                    children [
                        USE ball
                    ]
                }
                Transform {
                    translation 3 0 0
                    children [
                        USE ball
                    ]
                }
                Transform {
                    translation -1.5 0 0
                    children [
                        USE ball
                    ]
                }
                Transform {
                    translation -3 0 0
                    children [
                        USE ball
                    ]
                }
            ]
        }
    ]
}
Transform {
    translation 0 3 0
    children [

```

```
        USE g
    ]
}
Transform {
    translation 0 1.5 0
    children [
        USE g ]
    }
    Transform {
        translation 0 -1.5 0
        children [
            USE g ]
        }
        Transform {
            translation 0 -3 0
            children [
                USE g
            ]
        }
    ]
}
Transform {
    translation 0 0 1.5
    children [
        USE a
    ]
}
Transform {
    translation 0 0 -1.5
    children [
        USE a
    ]
}
Transform {
    translation 0 0 -3
    children [
        USE a ]
    }
    Transform {
        translation 0 0 -4.5
        children [
            USE a
        ]
    }
}
```

运行程序，效果如图 4-11 所示。

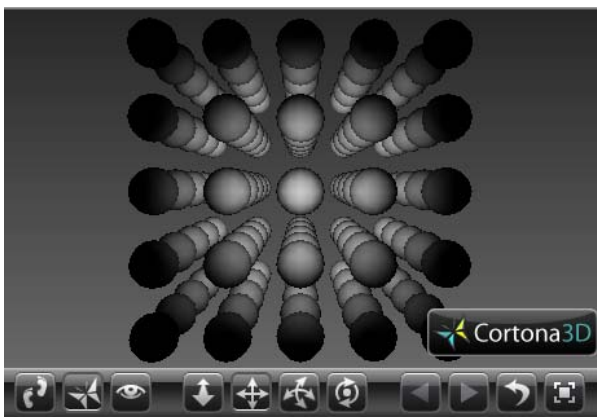


图 4-11 锥光源效果

【例 4-8】 发光效果不同的一组球体，同样是黄色，设置为 `diffuseColor` 域的域值，球体颜色受环境光线的影响；设置为 `emissiveColor` 域的域值，球体自身向外放射黄色光芒，不受环境光线的影响，有很强的感光；同时设置为 `diffuseColor` 域和 `emissiveColor` 域的域值，球体的光感减弱。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Transform {
  translation -2.5 0 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 1 1 0
        }
      }
      geometry Sphere {
      }
    }
  ]
}

Transform {
  translation 0 0 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          emissiveColor 1 1 0
        }
      }
      geometry Sphere {
```

```
    }  
  }  
]  
}  
Transform {  
  translation 2.5 0 0  
  children [  
    Shape {  
      appearance Appearance {  
        material Material {  
          diffuseColor 1 1 0  
          emissiveColor 1 1 0  
        }  
      }  
      geometry Sphere {  
      }  
    }  
  ]  
}
```

运行程序，效果如图 4-12 所示。

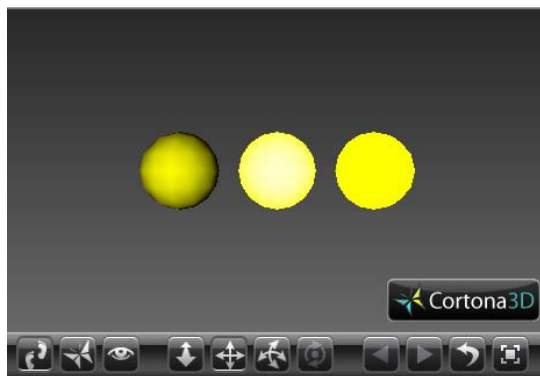


图 4-12 发光效果不同的球体

4.3 创建阴影效果

VRML 光源与自然光源的主要差别在于不能自动产生阴影。自然界的光线无法穿过不透明的物体，当光线照射这些物体时就会出现阴影。由于 VRML 中的光照只是对造型表面明暗程度的一种模拟，造型对光线没有遮挡作用，光线可以透过造型完全照射到后面的物体上，所以 VRML 中的光照只能改变造型表面的明暗分布，不能自动产生阴影效果。如果想在虚拟世界中模拟自然界的光照效果，则必须人工设置阴影造型。

阴影的形状常常是造型沿光线映射方向形状的压缩和变形，一般利用 Transform 节点的 scale 域模拟阴影的形状。阴影的颜色通常利用 M 模拟。

【例 4-9】 创建烟囱造型，设置垂直照射的平行光源，增加一块地板造型作显示阴影的地板，然后用一个镶嵌在平板造型里的灰色圆柱体模拟阴影造型。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyColor 1 1 1
}

NavigationInfo {          #关闭头灯光源
    headlight FALSE
}

DirectionalLight {        #垂直照射的平行光源
    direction 0 -1 0
}

Shape {                   #绘制烟囱造型
    appearance Appearance {
        material Material {
            diffuseColor 0 0 1
        }
    }
    geometry Cylinder {
        radius 1
        height 2
    }
}

Transform {
    translation 0 1.5 0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0 1 0
                }
            }
            geometry Cone {
                bottomRadius 2
                height 1
            }
        }
    ]
}

Transform {               #地板的造型
    translation 0 -2 0
    children [
```

```

    Shape {
      appearance Appearance {
        material Material {
        }
      }
      geometry Box {
        size 15 0.1 15
      }
    }
  ]
}
Transform {                                #阴影造型
  translation 0 -1.99 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0.5 0.5 0.5
        }
      }
      geometry Cylinder {
        height 0.1
        radius 2
        side FALSE
      }
    }
  ]
}
}

```

运行程序，效果如图 4-13 所示。

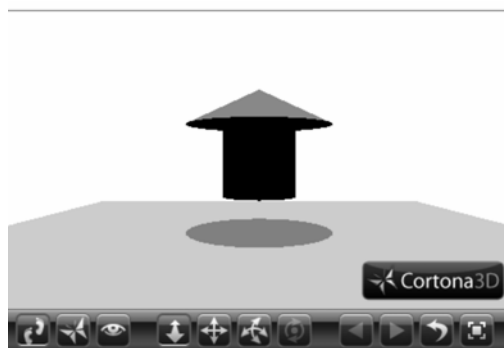


图 4-13 垂直照射的阴影效果

4.4 创建雾化效果

在 VRML 中使用 Fog 节点创建雾化效果，模拟自然界的大气现象，使虚拟现实的场景更

加逼真、更加具有层次感，创建一种朦胧的美。雾化效果主要控制两个因素：一是雾的颜色；二是雾的浓度。雾的颜色可以按照实际情况任意选择，但通常使用白色，因为白色更接近自然界雾的本色。雾的浓度与人在雾中的能见度相反，雾越浓，人的能见度越差，因此可以通过设定观察者能见范围的大小来调整雾的浓淡程度。

在 VRML 中，由 Background 节点设定的背景图像不会受雾的遮蔽效果的影响，因此设计者强调雾化效果时，最好不使用背景图像，或者使雾的颜色与背景颜色协调相融。

Fog 节点的语法如下：

节点名称	域名称	域值	#域及域值类型
DirectionalLight{	color	1.0 1.0 1.0	#exposedField SFCOLOR
	fogType	"LINEAR"	#exposedField SFString
	visibilityRange	0.0	#exposedField SFFloat
	set_bind		#eventIn SFBool
	isBound		#eventOut SFBool
	}		

其中：

(1) color 域的域值用于设定雾的颜色。设计者可以按照预期的雾化效果选择不同的颜色，如果场景中有背景图像，雾的颜色最好与背景颜色相一致。该域值的默认值为（1.0 1.0 1.0），即系统默认雾的颜色为白色。

(2) fogType 域的域值用于设定雾的渲染类型。该域值的默认值为“LINEAR”，此时雾的浓度随观察距离的增大而线性增大；当域值为“EXPONENTIAL”时，雾的浓度将随观察距离的增大而指数增大，变化程序远远大于线性变化。

(3) visibilityRange 域的域值用于设定浏览者在当前 VRML 场景中的可见范围。该域值若较大，则雾逐渐变浓，产生薄雾的效果；该域值若较小，则雾突然变浓，产生浓雾的效果。当距离超过此域值时，造型将完全被雾遮挡而无法看见。该域值的默认值为 0.0，即不产生雾化效果。

(4) set_bind 入事件与 isBound 出事件用于进行当前 Fog 节点的切换。系统可以生成多个不同设置的 Fog 节点，用来模拟多种雾化效果。一般情况下，VRML 文件中的第一个 Fog 节点作为系统的初始雾化设置，其他 Fog 节点只有在 set_bind 入事件为 TRUE 时，才能切换成当前雾化节点；同时系统原来指定的 Fog 节点的 isBound 出事件为 FALSE，表示不再作为当前雾节点。

【例 4-10】 下面是在 Fog 节点的应用示例以及在 visibilityRange 域值不同时，两个粮仓的雾化效果显示。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    Fog {
      color 1.0 1.0 1.0
      fogType "LINEAR"
      visibilityRange 15.0
```

```

    }
    # 创建造型
    Transform {
        translation -2.0 0.0 0.0
        children [
            Transform {
                translation 0 0 -2
                children [
                    DEF aa Transform {
                        translation 0 -0.75 0
                        children [
                            Shape {
                                appearance Appearance {
                                    material Material {
                                        diffuseColor 1.0 0.0 0.0
                                    }
                                }
                                geometry Box {
                                    size 1.1 0.5 0.3
                                }
                            }
                        ]
                    }
                ]
            }
            DEF ab Shape {
                appearance DEF breen Appearance {
                    material Material {
                        diffuseColor 0.0 1.0 0.0
                    }
                }
                geometry Box {
                    size 1.0 2.0 2.0
                }
            }
            DEF ac Transform {
                translation 0.0 1.5 0.0
                children Shape {
                    appearance USE breen
                    geometry Cone {
                        height 1.2
                        bottomRadius 1.2
                    }
                }
            }
        ]
    }
    Transform {

```

```
translation 5 0 2
children [
    USE aa
    USE ab
    USE ac
]
}
}
}
Background {
    skyColor [0.0 0.5 1.0,1.0 1.0 1.0]
    skyAngle [1.309,1.571]
    groundAngle [1.309,1.571]
    groundColor [0.1 0.0 0.0,0.4 0.25 0.2,0.6 0.6 0.6]
}
]
}
```

运行程序，效果如图 4-14～图 4-16 所示。

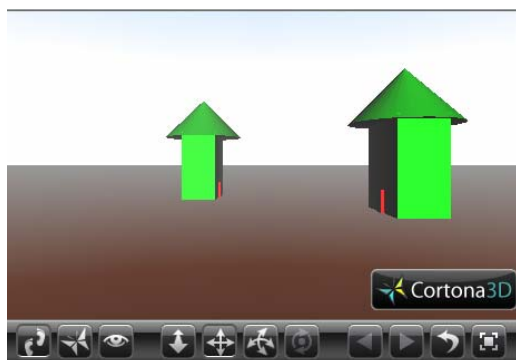


图 4-14 可见范围为 40.0 的显示效果

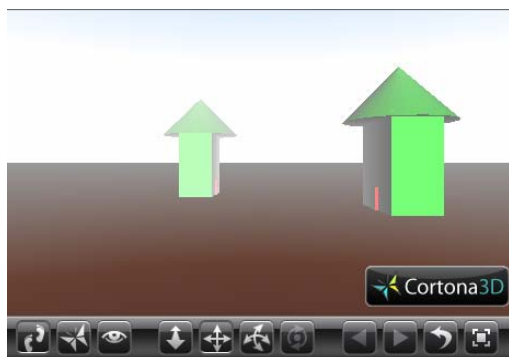


图 4-15 可见范围为 15.0 的显示效果

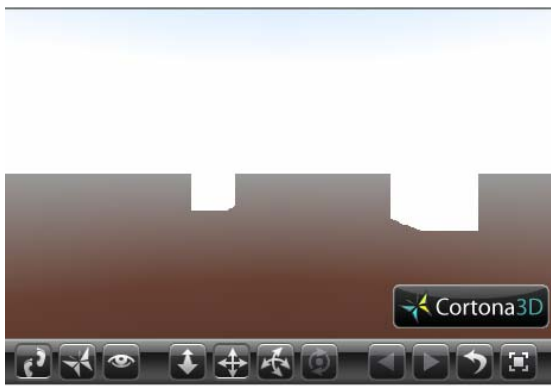


图 4-16 可见范围为 1.0 的显示效果

4.5 创建声音效果

现实世界中除了五彩缤纷的物体外,还有各种各样的声音,如鸟鸣、蛙叫、流水声、汽笛声、讲话声、音乐声等。在 VRML 虚拟现实的场景环境设计中,添加声音创建音响效果是通过声音节点来实现的。Sound 节点用于创建声场并设定声音播放方式, AudioClip 和 MovieTexture 节点用于创建声源。

4.5.1 环境中声音的基本概念

有 3 种声音文件可以用作 VRML 虚拟世界的声源: WAV 文件、通用 MIDI 文件、MPEG-1 文件。

(1) WAV 是“waveform”(波形)的缩写,是微软 Windows 中的标准声音文件格式,可存储低质量、一般质量和高质量的数字音响波形信息。为了准确地描述声音, WAV 文件使用 44100 双字节(用于双声道), 1min 的数字声音要占用 10MB 左右的磁盘空间。WAV 文件的扩展名为“.wav”。

(2) MIDI 是“Musical Instrument Digital Interface”(乐器数字接口)的缩写,是一个用于音乐演奏时声音合成器之间通信的标准。该标准指定了声音合成器应当支持的声音,包括长号、钢琴、长笛、架子鼓等。MIDI 格式是专门用于电子音乐合成的统一接口。MIDI 指令存储的是演奏的信息,如开始和停止时间、演奏的节拍以及用多大的强度演奏一个节拍等。因为 MIDI 文件中存储的只是如何演奏的指令,而不是声音的数字化信息。1min 的中等复杂程度的钢琴音乐采用 MIDI 文件记录可能只占几千字节的磁盘空间。MIDI 文件的扩展名为“.mid”。

(3) MPEG 是“Moving Picture Experts Group”(运动图像专家组)的缩写,是一种压缩比率较大的活动图像和声音的压缩比率标准,其压缩率为 0.8 位/像素到 0.4 位/像素之间。MPEG 文件的扩展名为“.mpg”。

这 3 种文件都可以为 VRML 虚拟世界提供丰富的声源。在一个 VRML 场景中,可以在其中选择任何一种类型的文件作为声源,也可以同时使用 3 种不同类型的声音文件。通常情况下, WAV 文件适于播放短的声音效果或讲话, MIDI 文件适于演奏背景音乐, MPEG 文件适于播放声音和图像同步的影片。

在 VRML 场景中,利用 Sound 节点创建声场:设置声源的位置及声音播放的方式。AudioGlip 和 MovieTexture 节点是 Sound 节点 source 域的域值,用于创建声源:具体引入声音

文件。

4.5.2 音频剪辑节点

音频剪辑 (AudioClip) 节点描述了一个声音发生源, 指定了其他需要声源的节点可以引用的音频文件的位置及播放的各种参数, 就像生成一台播放音乐的机器。

VRML 所支持音频文件有以下几种: WAV、MIDID 文件格式是通过 AudioClip 节点引用的, 而 MPEG-1 文件格式是通过 MovieTexture 影像文件节点来引用的。

AudioClip 节点语法如下:

节点名称	域名称	域值	#域及域值类型
AudioClip {	url	[]	#exposedField MFString
	description	""	#exposedField SFString
	loop	FALSE	#exposedField SFBool
	pitch	1.0	#exposedField SFFloat
	startTime	0.0	#exposedField SFTIME
	stopTime	0.0	#exposedField SFTIME
	duration_change		#eventOut SFTIME
	isActive		#eventOut SFBool
	}		

其中:

(1) url 域的值定义了一个或一组引入的音频文件的 url 地址。该域值提供了在该 VRML 场景中所要播放的声音文件的具体位置, 其排列顺序为从高优先级到低优先级。通常浏览器从地址列表中第 1 个 url 指定地址试起, 如果音频文件没有被找到或不能被打开, 浏览器就尝试读入下一个作为声源。其默认值为一张空的 url 列表, 表示没有任何文件被打开, 不播放任何声音。

(2) loop 域的值指定是否循环播放所引用的音频。它是一个布尔运算量; 当域值为 TRUE, 只要 startTime 大于 stopTime, 音频便循环播放; 当域值为 FALSE, 则音频只播放一次。其默认值为 FALSE。

(3) description 域的值定义了一组描述所引用音频文件的文本串。在浏览器播放该音频文件的同时显示这些文本串, 或在不能播放时显示该文本串, 以说明该音频文件。该域值的默认值为空。

(4) pitch 域的值指定了播放音频的相乘因子 (频率的倍数), 用来加快或减慢播放速度。将 pitch 域的域值同音频文件的固有播放时间相乘就是该音频文件在 VRML 场景中的播放时间。当该域值为 1.0 时, 音频按正常速度播放; 当该域值在 0.0~1.0 之间时, 将减慢音频的播放速度, 并降低音调; 当该域值大于 1.0 时, 将加快音频的播放速度, 并提高音调。该域值的默认值为 1.0。

(5) startTime 域的值定义了音频文件开始播放的时间, 该域值的默认值为 0.0s。

(6) stopTime 域的值定义了音频文件停止播放的时间。该域值的默认值为 0.0s。

(7) startTime、stopTime、pitch 和 loop 域共同控制着 AudioClip 节点的音频播放。此节点在其 startTime 到达之前保持休眠状态, 即不播放音频文件。在 stopTime 时刻, 开始播放音频; 如果 loop 域值为 FALSE, 当 startTime 到达或播放完一遍音频后, 停止播放; 如果 loop 域值为 TRUE, 将连续反复播放音频, 直到 stopTime 为止。在 stopTime 早于 startTime 的情况下, 系统将忽略停止时间, 这可用来生成永远循环的音频。

(8) duration_change 为 SFTIME 值类型的 eventOut 事件，指明播放的周期。

(9) isActive 为 SFFBool 值类型的 eventOut 事件，指明电影当前是否正在播放。当开始播放时该事件以 TRUE 值送出；当播放结束时以 FALSE 值送出。startTime, stopTime, speed 和 loop 域的值共同作用控制 MovieTexture 节点的纹理。在开始时间 MovieTexture 节点被激活用 isActive 域 (eventOut 出事件) 输出 TRUE，在开始正向播放第一画面 (如果 speed 域的值 为负，则为反向)。在每一幅新的电影画面中，造型纹理被变为新的电影画面。如果 loop 域 的值为 FALSE，那么直接停止时间，或 startTime+duration/speed 的播放时间到达之后，MovieTexture 节点才开始产生纹理。如果 loop 域 的值为 TRUE，那么 MovieTexture 节点持续产生纹理，直至停止时间。在停止时间早于造型开始时间的特殊情况下，停止时间将被忽略。这可以用来不停地产 生电影纹理。在任何情况下，当电影停止时，isActive 域(eventOut 出事件)将输出 FALSE，电影中的最后一幅画面保留作为造型纹理。

4.5.3 声音节点

AudioClip 在 VRML 文件中，不能作为单独的节点来使用，它只是创建了声源、而要想播 放就需要声音节点来实现。

Sound 节点在 VRML 世界中生成了一个声音发射器，可以用来指定声源的各种参数，即指 定了 VRML 场景中的声源的位置和声音的立体化表现。声音可以位于局部坐标系中的任何一 个点，并以球面或椭球的模式发射出声音。Sound 节点也可以使声音环绕，即不通过立体化处 理，这种声音离它所指定的距离逐渐变为 0。Sound 节点可以出现在 VRML 文本文件的顶层， 也可以作为组节点的子节点，如图 4-17 所示。

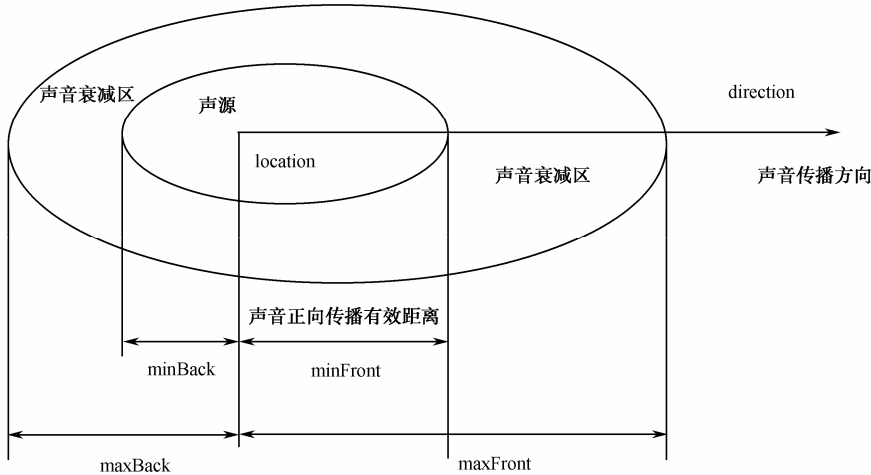


图 4-17 声音传播原型示意图

Sound 节点语法如下：

节点名称	域名称	域值	#域及域值类型
Sound {			
	direction	0.0 0.0 1.0	#exposedField SFFloat
	intensity	1.0	#exposedField SFVec3f
	location	0.0 0.0 0.0	#exposedField SFFloat
	maxBack	10.0	#exposedField SFFloat
	maxFront	10.0	#exposedField SFFloat

minBack	10.0	#exposedField SFFloat
minFront	1.0	#exposedField SFFloat
priority	0.0	#exposedField SFFloat
source	NULL	#exposedField SFNode
spatialize	TRUE	#exposedField SFBool
}		

其中：

(1) **direction** 域的值指定了声音发射器的空间朝向，即规定 VRML 界中声音发射器所方向的矢量，声音发射器将以这个矢量的方向发射声音。该矢量由 3 个浮点数表示，分别表示一个三维向量的 X、Y、Z 部分。该域的默认值为 (0.0 0.0 1.0)，即指向空间坐标系的 Z 轴正方向的向量。

(2) **intensity** 域的值指定了声音发射器发射声音的强度，即音量。该域值在 0.0~1.0 范围内变化。1.0 表示音量最大，为了声音文件建立时的全音量；0.0 表示静音，在 0.0~1.0 之间的值则表示不同声音发射器的音量。但当 **intensity** 的域值大于 1.0 时，会使该声音失真。其默认值为 1.0。

(3) **location** 域的值指定了当前局部坐标系中一个用来表示声音发射器位置的三维坐标。该域的默认值为 (0.0 0.0 0.0)，即坐标系的原点。

(4) **priority** 域的值为 VRML 场景中播放的声音提供了声源。其域值就是用来创建声源的 **audioClip** 节点或 **MovieTexture** 节点。该域的默认值为 NULL，表示没有声源。

(5) **source** 域的值为 VRML 场景中播放的声音提供了声源。其域值就是用来创建声源的 **audioClip** 节点或 **MovieTexture** 节点。该域的默认值为 NULL，表示没有声源。

(6) **spatialize** 域的值指定是否实现声音立体化，即是否将声音经过数字处理，使浏览者听到声音的同时可感觉出声音发射器在三维空间的具体位置，从而达到立体效果。该域值为布尔值运算。当域值为 TRUE，声音信号被转换为一个单音信号，经过立体化处理，然后由扬声器或耳机输出；当域值为 FALSE 时，声音信号将不经处理，直接由扬声器或耳机输出。

(7) **maxBack** 域的值指定了当前坐标系中从声音发射器所在位置沿 **direction** 域所指定方向的相反方向假定的直线距离，超过此距离则听不到声音。该域值的设定要求大于或等于 0.0，其默认值为 10.0。

(8) **maxFront** 域的值指定了在当前坐标系中从声音发射器所在位置沿 **direction** 域所指定方向假定的直线距离，超过此距离则听不到声音。该域值要求大于或等于 0.0，其默认值为 10.0。

(9) **minBack** 域的值指定了在当前坐标系中从声音发射器所在位置沿 **direction** 域所指定方向的相反方向假定的直线距离，超过此距离则声音开始衰减，直到 **maxBack** 域所指定的距离处，音量为零。该域值要求大于或等于 0.0，其默认值为 1.0。

(10) **minFront** 域的值指定了在当前坐标系统中从声音发射器所在位置沿 **direction** 域所指定方向假定的直线距离，超过此距离声音开始衰减，直到 **maxFront** 域所指定的距离处，音量为零。该域值要求大于或等于 0.0，其默认值为 1.0。

direction、**location**、**maxBack**、**maxFront**、**minBack**、**minFront** 这几个域在 VRML 空间中共同指定了两个不可见的表示声音范围的椭圆型半球。这两个椭圆球表示浏览者在 VRML 中移动时由一个声音发射器发出的声音在音量上的变化和声音传播的范围。当观察者位于最小范围椭圆球中时，因为观察者离声音发射器较近，所听到的该声音发射器发出的声音强度量大，

且处处相等；当观察者位于最小范围椭圆球和最大范围椭圆球之间时，观察者离声音发射器越远，所听到该发射器发出的声音强度越小，这与现实中的声音减弱规律一样，直到最大范围椭圆球边缘处时，声音强度变为 0。当观察者位于最大范围椭圆球之外时，观察者离声音发射器已经相当远了，听不到该发射器发出的声音。声音强度变化规律如图 4-17 所示。正常声音范围在最小范围椭圆球内，声音衰减范围在最小范围椭圆球到最大范围椭圆球之间，声音盲区最大范围椭圆球之外的空间中。

【例 4-11】 虚拟声音的播放。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    Background {
      skyAngle [
        0.2 0.3 0.2
      ]
    }
    Sound {                                     #声音节点
      source AudioClip {                       #声源节点
        url "music2.mid"                      #声源文件所在位置或路径
        description "sound"                   #文字描述声音文件
        loop TRUE                             #循环播放
        pitch 1.0                             #声音播放的速率
      }
      direction 0 0 1                         #声音的传播方向
      intensity 1                             #声音的强度最大
      location 0 0 0                           #声源来源的坐标位置
      maxBack 30                              #正向发射的最大距离
      maxFront 30
      minBack 30
      minFront 10
      spatialize FALSE
    }
  ]
}
```

【例 4-12】 创建一个液晶电视造型，并播放视频。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
  skyColor 0.1 0.5 0.6
}

Transform {
```

```

translation 0 0 -0.1
children [
    Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0.3 0.3 0.3
            }
        }
        geometry Box {
            size 5.1 3.4 0.2          #电视造型
        }
    }
]
}
Shape {
    appearance Appearance {
        texture    DEF hg MovieTexture{          #电影纹理
            url     "aa.mpg"
            loop TRUE
        }
    }
    geometry Box {
        size 4.5 3.0 0.01
    }
}
Sound {
    maxBack 10
    maxFront 80
    minBack 5
    minFront 30
}

```

运行程序，效果如图 4-18 所示。

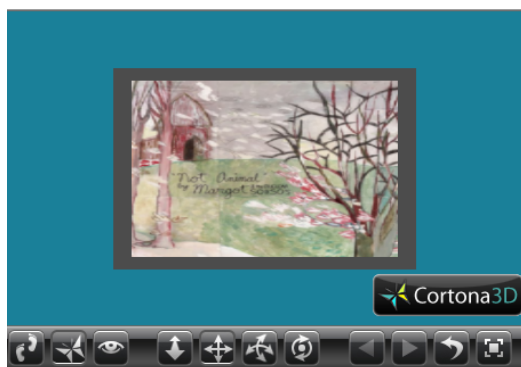


图 4-18 播放影像文件的效果

第 5 章 动画效果与交互节点

动画效果和传感节点功能是 VRML 最具有魅力的特色。现实世界中的万物是多姿多彩、瞬息万变、具有强大生命力的，彼此之间相互影响、相互作用、相互联系。VRML 使用 6 种插补器节点和 8 种传感节点，通过路由在各个造型节点和场景节点之间传递事件，为浏览者创建一个身临其境、动态、逼真、可以交互沟通的虚拟环境。

5.1 事件和路由

5.1.1 事件

节点可以通过接收事件而改变。大部分的节点有能力处理入事件（eventIns）。通过接收 eventIns 的指示，节点可以改变目前的状态，例如节点的颜色可以改变，set_color。当一个节点被改变了，它会对改变的状态有所反应，送出一些发生事件（eventOuts），例如 Color_changed。

一个 exposedField 能够接收事件，如 eventIn；能够产生事件，例如 eventOut。事件是短暂的，事件值是不会被写入 VRML 的文件中。exposedField 就如字段一样被放置在 VRML 文件中。如果一个 exposedField 叫做 zzz，那么它的 eventIn 事件为 set_zzz。eventOut 事件则为 zzz_changed。

下面列出一些 eventIns 和 eventOuts 的规则。

（1）大部分的 eventIns 都是以 set_为开头的事件，除了 addChildren 和 removeChildren 这两个 eventIns 以外。

（2）大部分的 eventOuts 都是以 _changed 为结尾的事件，除了形态为 SFBool 的 evntOut。布尔值的 eventOuts 是以 is 作为开头，如 isBound、isActive 等。

（3）eventIns 和 eventOuts 的形态为 SFTime 时，就不再使用 set_和 changed_作为连接，如 bindTime 和 touchTime 等。

5.1.2 路由

路由（Route）的功能是连接一个节点的 eventOuts 事件和一个节点 eventIns 事件。Route 并不是节点，它只是简单的语法结构，告诉一个事件该如何从某个节点到达另一个节点。Route 用于建立两个节点间的路径。

Route 可以放置在 VRML 的最顶层、在 PROTO 的定义中或在会出现字段的节点内。

Route 的语法如下：

```
Route NodeName.eventOutName changed To NodeName.set eventName
```

【例 5-1】 路由与事件示例。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
```

```
Group {  
  children [  
    DEF ball Transform {
```

```
children Shape {
  appearance Appearance {
    material Material {}
  }
  geometry Sphere {}
}
DEF clock TimeSensor {
  cycleInterval 2.0
  loop TRUE
}
DEF ballpath PositionInterpolator {
  key [
    0.0 0.2 0.65 1.0
  ]
  keyValue [
    1.0 1.0 1.0
    1.5 1.5 1.5
    1.1 1.1 1.1
    1.0 1.0 1.0
  ]
}
ROUTE clock.fraction_changed TO ballpath.set_fraction
ROUTE ballpath.value_changed TO ball.scale
```

运行程序，效果如图 5-1 所示。

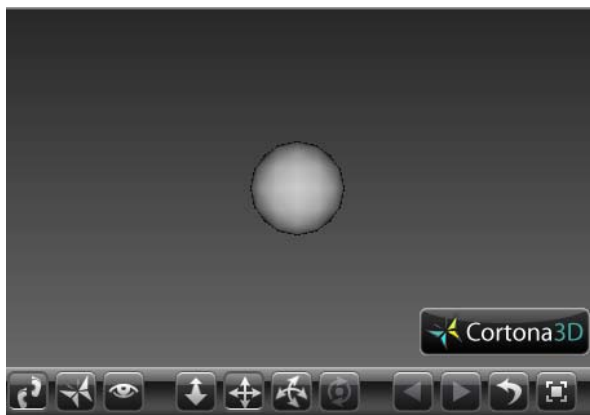


图 5-1 球体改变尺寸效果

通过以上示例说明了 Route 如何建立节点间的路径，并传送事件。图 5-1 的画面中的球体会不断地改变尺寸。

5.2 动画效果

在现实世界中，很多东西是随时间发生变化的，如太阳的升起和降落、花蕾的绽开和凋谢、树叶由绿变黄、鱼儿戏水、小鸟飞翔、汽车行驶、表针转动等。这些就是 VRML 要在虚拟环境中，通过人为设置实现的动画效果。

动画就是物体随时间发生变化的动态效果。VRML 创建动画效果的基本方法是：由时间传感器控制动画的时钟，包括动画开始的时间、停止的时间、循环的周期等；然后将时间控制参数作为事件传送给各种插补器节点，插补器依据事先设计好的时间关键点和动画关键值，在浏览器渲染时形成连续变化的动画效果。

插补器节点是为线性关键帧动画而设计的。其采用一组相对时间关键点，且每个关键点对应一种关键状态（关键值），该状态允许以各种数据形式表示，如 SFVec3f 或 SFCOLOR。浏览器将根据这些关键点所对应的关键状态，在场景中通过线性插值自动生成连续的动画。一般来说，浏览器在两个相邻关键帧之间生成的连续帧是线性变化的。

根据所插值的数据类型不同插补器节点可分为 6 种：颜色差补器（ColorInterpolator 节点）、位置插补器（PositionInterpolator 节点）、朝向插补器（OrientationInterpolator 节点）、标量插补器（ScalarInterpolator 节点）、坐标插补器（CoordinateInterpolator 节点）、法向量插补器（NormalInterpolator 节点）。

5.2.1 时间传感器节点

时间传感器节点（TimeSensor）用于创建一个系统虚拟时钟，并对动画插补器节点实施时间控制。随着时间的推移。TimeSensor 节点可以在一个指定的时间引发动作或以固定的时间间隔输出事件。TimeSensor 节点在 VRML 场景中不产生任何造型和可视效果，只是向各种插补器节点输出时间事件，进行时间控制，以使插补器节点产生随时间变化的动画效果。该节点通常放置在最外层节点的后面，也可以作为任何组节点的子节点使用，但 TimeSensor 节点总是独立于所选用的坐标系之处。

TimeSensor 节点语法如下：

节点名称	域名称	域值	#域及域值类型
TimeSensor	{		
	enabled	TRUE	#exposedField SFBool
	loop	FALSE	#exposedField SSFBool
	cycleInterval	1.0	#exposedField SFTIME
	startTime	0	#exposedField SFTIME
	stopTime	0	#exposedField SFTIME
	isActive	0	#eventOut SFBool
	time		#exposedField SFTIME
	cycleTime		#exposedField SFTIME
	fraction_changed		#exposedField SFFloat
	}		

其中：

（1）enabled 域的域值用于设定时间传感器的使用状态。该域值的默认值为 TRUE，表示时间传感器打开；若域值为 FALSE，则表示时间传感器关闭。

(2) **loop** 域的域值用于设定时间传感器是否循环输出事件。该域值默认值为 **FALSE**，表示时间传感器不循环输出事件，经过一个时间周期后，就会自动停止；若域值为 **TRUE**，则表示传感器按时间周期自动循环输出事件，直到 **stopTime** 设定的停止时间为止。

(3) **cycleInterval** 域的域值用于设定时间传感器的时间周期。也就是设定时间传感器从 0.0 到 1.0 一个周期的具体时间长度，单位为 s。该域值必须大于零，默认值为 1s。

(4) **startTime** 域的域值用于设定时间传感器开始输出事件的时间，单位为 s。该域值的默认值为 0s。

(5) **stopTime** 域的域值用于设定时间传感器停止输出事件的时间，单位为 s。该域值的默认值为 0s。该域值若小于等于 **startTime** 域值，则将被系统忽略。

(6) **isActive** 是出事件，表明时间传感器当前的运行状态。若正在运行，则向外输出 **TRUE** 事件；若处于停止状态，则向外输出 **FALSE** 事件。

(7) **time** 是出事件，在时间传感器开始运行后，不断向外输出当前时间。该域值为从格林威治时间 1970 年 1 月 1 日 00: 00 时（午夜 12 点）至今所经过的秒数。

(8) **cycleTime** 是出事件，在时间传感器每次循环开始时输出一个当前时间。

(9) **fraction_changed** 是出事件，在时间传感器开始运行后，输出 0.0 到 1.0 之间的时刻比例数值。0.0 表示时间周期开始，1.0 表示时间周期结束。

同 **AudioClip** 节点中的域类似，**startTime**、**stopTime**、**cycleInterval** 和 **loop** 域共同控制时间传感器的运行状态，见表 5-1。

表 5-1 时间传感器运行状态表

loop 值	startTime、stopTime、cycleInterval 域值之间的关系	运 动 状 态
TRUE	$stopTime \leqslant startTime$	无限循环
TRUE	$startTime < stopTime$	循环到 stopTime 为止
FALSE	$stopTime \leqslant startTime$	只运行一个周期
FALSE	$startTime < (startTime + cycleInterval) \leqslant stopTime$	只运行一个周期
FALSE	$startTime < stopTime < (startTime + cycleInterval)$	不到一个周期，在 stopTime 处停止

5.2.2 位置插补器

PositionInterpolator 节点的计算输出值是一个 **SFVec3f** 类型，并被设计为使用平移值或三维坐标值的节点输入。例如，**Transform** 节点使用一个 **SFVec3f** 值作为 **translation** 域的值。**PositionInterpolator** 节点适用于激活该域的值，从而导致坐标系的平移。任何适于 **SFVec3f** 类型值作为域值的角度均能够使用 **PositionInterpolator** 节点来发生动画，被激活的域并不一定必须是一个描述位置的域。例如，**Transform** 节点的出 **scale** 域使用 **SFVec3f** 类型的值描述坐标系缩放的 X、Y、Z 缩放因子。可以通过将插补器的输出路由至节点的 **scale** 域来实现这些缩放因子。这样，**Transform** 节点坐标系将在 **PositionInterpolator** 节点的动画关键值控制下进行放大和缩小。

可利用 **PositionInterpolator** 动态改变观察位置，或者改变形体的位置。如果用来动态改变 **Viewpoint** 节点的位置，或用来动态改变形体的位置，使形体动态移动，或可用来改变形体所在局部坐标系的 **scale** 字段。**PositionInterpolator** 节点允许对三维空间的一个坐标点进行动画关键帧的插值操作。建立插补器时，为动画的不同完成比率设置相应的坐标值（最好包括开始值

和结束值)。通常坐标插补器从时间传感器接收 `set_fraction` 事件, 经处理后, 将输出值发送给 `Transform` 节点的 `translation` 域。使用步骤如下:

- (1) 给需要变化的具有同样值类型的 `eventIn` 事件的节点 (如 `Viewpoint` 或 `Transform` 等节点) 命名。
- (2) 定义一个 `TimeSensor`, 确定变化周期及循环方式。
- (3) 定义一个 `PositionInterpolator` 节点, 给出变化的具体方式。
- (4) 给出两个 `Route` 语句, 一个将 `TimeSensor` 的变化传给 `PositionInterpolator`, 一个将 `PositionInterpolator` 的变化传给 `Transform` 或 `Viewpoint`。

`PositionInterpolator` 节点语法如下:

节点名称	域名称	域值	#域及域值类型
<code>PositionInterpolator {</code>	<code>set_fraction</code>		<code>#eventIn MFFloat</code>
	<code>key</code>	<code>[]</code>	<code>#exposedField MFFloat</code>
	<code>keyvalue</code>	<code>[]</code>	<code>#exposedField MFVec3f</code>
	<code>value_changed</code>		<code>#eventOut MFVec3f</code>
	<code>}</code>		

其中:

- (1) `key` 域的域值为关键帧时间比率列表。通常介于 `0.0~1.0` 之间的浮点数, 包括 `0.0` 和 `1.0`。它指定了一个浮点时刻关键值列表。时刻在 `0.0~1.0` 之间。但是, 关键时刻可能是一个任意小的正的或负的浮点值。关键时刻必须按递增列出。默认的 `key` 值是一个空的列表。
- (2) `keyvalue` 域的域值指定一个关键位置的坐标列表。每一个关键位置都是一组由 `X`、`Y` 和 `Z` 浮点值组成的三维坐标或平移距离。在一些运用中, 关键值也可以是 `X`、`Y` 和 `Z` 的比例要素或其他三个浮点值的组。`KeyValue` 域值的默认值是一个空的列表。关键时刻和位置在一起使用, 其目的是第一个时刻指定第一个关键位置的时间, 第二个关键时刻指定第二个关键位置的时间, 如此下去。列表可以提供任意数目的时刻和位置, 但是两者必须包括相同数目的列表值。
- (3) `set_fraction` 域的域值为 `SFFloat` 值类型的 `eventIn` 事件, 控制动画的完成比率。
- (4) `value_changed` 域的域值为 `SFVec3f` 值类型的 `eventOut` 事件, 控制与比率相对应的坐标值。

当一个 `PositionInterpolator` 节点接收到一个时刻时, 它将计算基于关键位置列表和相关的关键时刻的一个位置。新计算出的位置由 `Value_changed` 或 (`eventOut` 出事件) 输出。

一个输入时刻由 `PositionInterpolator` 节点计算:

- (1) 扫描关键时刻值, 找到一个时间领域 `t1` 和 `t2`, $t1 \leq t2$ 。
- (2) 查出一对相应的位置关键值。
- (3) 由在关键位置中的线性内插来计算一个中间位置。

两个相邻的关键值 `t1` 和 `t2` 可以用相同的值。其目的是在一个动画中建立一个不连续的跳跃过程。在这种情况下, 当线性内插时间小于 `t1` (两个指定的关键时刻中的时), 与 `t1` 相应的关键位置被 `PositionInterpolator` 节点使用, 当线性内插时间大于 `t2` (两个指定的关键时刻中的第二个) 时, 与之相关的关键位置就被 `PositionInterpolator` 节点使用。

通常情况下，时刻被送入 `set_fraction` 域（`eventIn` 入事件），只需将 `TimeSensor` 节点的 `fraction_changed` 域（`eventOut` 出事件）捆绑到一个路由上。时刻也可以由其他方式产生。例如节点的输出产生一个一般的浮点值。

关键时刻和关键位置的列表值通过将新的值传送到公共域 `key` 和 `keyValue` 的隐含 `set_key` 和 `set_key` 域（`eventIn` 入事件）来改变。当接收到新值时，相应的域值被改变，并且新的值通过公共域隐含的 `keychanged` 和 `keyValue_changed` 域（`eventOut` 出事件）来输出。

`PositionInterpolator` 节点不创建物体并且在虚拟世界中也没有可视效果。`PositionInterpolator` 可作为子节点包含于任何编组节点中，但是它独立于所使用的坐标系。插补器可置于一个 VRML 文档的最外层组的最后。

【例 5-2】 创建一棵移动的树，但树是被限定在 X 轴上移动，共有 5 个 `key`，每个 `key` 分别有一个对应的坐标值 `keyValue`，循环周期是 10s。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    DEF tree Transform {
      children Inline {
        url"tree.wrl"
      }
    }
    DEF clock TimeSensor {
      cycleInterval 5.0
      loop TRUE
    }
  ]
}
```

运行程序，效果如图 5-2 和图 5-3 所示。

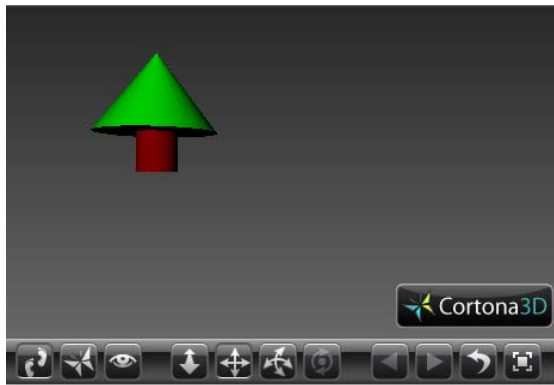


图 5-2 绘制的树效果

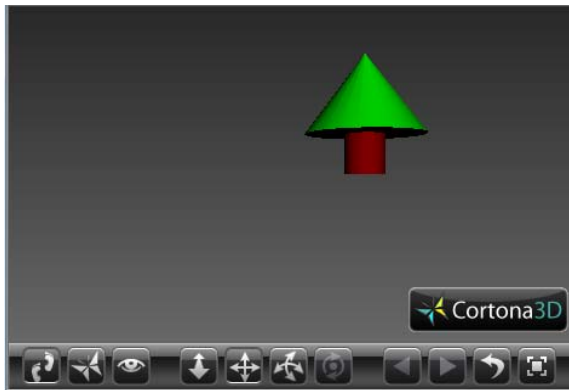


图 5-3 移动位置的树效果

```
DEF treepath PositionInterpolator {
  key [
    0.00
```

```

        0.25
        0.75
        1.00
    ]
    keyValue [
        -4.0 0.0 0.0
        -2.0 0.0 0.0
        2.0 0.0 0.0
        4.0 0.0 0.0
    ]
}
]
}
ROUTE clock.fraction_changed TO    treepath.set_fraction
ROUTE treepath.value_changed TO    tree.translation

```

而在程序中调用绘制树“tree.wrl”的源程序代码如下：

```

#VRML V2.0 utf8

DEF tree Group {
    children [
        Transform {
            translation 0 2 0
            children Shape {
                appearance Appearance {
                    material DEF green Material {
                        diffuseColor 0.0 1.0 0.0
                    }
                }
                geometry Cone {
                    height 2
                    bottomRadius 1.5
                }
            }
        }
        Transform {
            translation 0 0.5 0
            children Shape {
                appearance Appearance {
                    material DEF brown Material {
                        diffuseColor 0.5 0.0 0.0
                    }
                }
                geometry Cylinder {
                    height 1.0
                    radius 0.5
                }
            }
        }
    ]
}

```

```

    }
  }
}
]
}

```

【例 5-3】 利用 PositionInterpolator 对虚拟对象进行放大或缩小。
其实现的源程序代码如下：

```

#VRML V2.0 utf8

Group {
  children [
    DEF tree Transform {
      children Inline {
        url "tree.wrl"
      }
    }
    DEF clock TimeSensor {
      cycleInterval 3.0
      loop TRUE
    }
  ]
  DEF treepath PositionInterpolator {
    key [
      0.0
      0.25
      0.75
      1.00
    ]
    keyValue [
      0.5 0.5 0.5
      0.75 0.75 0.75
      1.0 1.0 1.0
      1.25 1.25 1.25
      1.5 1.5 1.5
    ]
  }
}

ROUTE clock.fraction_changed TO treepath.set_fraction
ROUTE treepath.value_changed TO tree.scale

```

运行程序，效果如图 5-4 和图 5-5 所示。

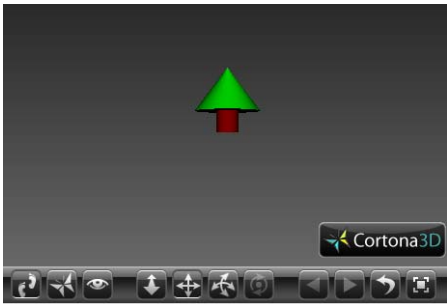


图 5-4 绘制的树

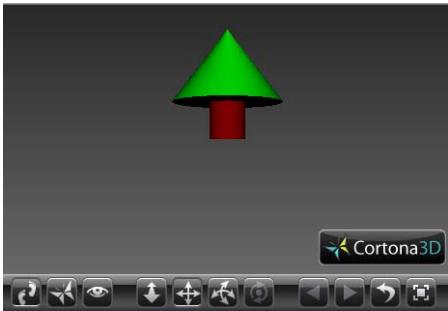


图 5-5 树的尺寸改变

5.2.3 颜色插补器

ColorInterpolator 节点用于产生场景造型变色的动画效果。该节点并不创建造型，可以作为任何组节点的子节点。

ColorInterpolator 节点语法如下：

节点名称	域名称	域值	#域及域值类型
ColorInterpolator {	key	[]	#exposedField MFFloat
	keyValue	[]	#exposedField MFColor
	set_fraction		#eventOut SFFloat
	value_changed		#eventOut SFColor
	}		

其中：

(1) key 域的域值用于设定一组时间关键点的列表。每一个关键点是一个浮点时刻值，与接收到的入事件 set_fraction 相对应。时刻值一般在 0.0~1.0 范围内取值，代表一个时间周期中的相对时刻，对应于时间传感器发出的 fraction_changed 出事件值，并要求依次递增排列。该域值的默认值为空列表。

(2) keyValue 域的域值用于设定一系列 RGB 颜色的关键色彩值。关键色彩值与 key 域的时间关键点一一对应。该域值的默认值为空列表。

(3) set_fraction 为入事件，用于不断接收来自时间传感器发出的比例数值。每收到一个时刻值，ColorInterpolator 节点就在时间关键点和其相应的关键色彩值的基础上计算出一个 RGB 色彩值，并通过 value_changed 出事件发送出去。

(4) value_changed 为出事件，用于输出计算后的 RGB 色彩值。

【例 5-4】 创建一个颜色变化的吊灯。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
  skyColor [
    0.5 0.6 0.8
    0.6 0.7 0.9
    1.0 1.0 1.0
  ]
}
```

```

skyAngle [1.309 1.571]
#利用群节点将灯变成一个整体，以便其他地方调用
}
Group {
  children [
    #灯的吊杆
    Shape {
      appearance Appearance {
        material Material {
          diffuseColor 0 0 0
          specularColor 1 0.6 0.5
        }
      }
      geometry Cylinder {
        radius 0.04
        height 2.0
      }
    }
    Transform {
      #在灯内部的局部坐标系中，将灯与灯罩向下移动的距离，以得到正确的效果
      translation 0.0 -1.48 0.0
      children [
        #灯罩
        Shape {
          appearance Appearance {
            material Material {
              diffuseColor 0 0 0
              specularColor 1 1 1
              emissiveColor 0.62 0.65 0
              ambientIntensity 0
              shininess 0.32
              transparency 0.5 #设置一定的透明度
            }
          }
          geometry Cone {
            bottom TRUE #可以尝试将其设为 FALSE，看看有什么变化
            height 1.0
            bottomRadius 1.5
          }
        }
        Transform {
          #在局部坐标系中，向下移动 0.5 个单位
          translation 0.0 -0.5 0.0
          children [
            # 灯泡
            Shape {

```

```

                                appearance Appearance {
                                    material DEF yzs Material {
                                        diffuseColor 1 1 1
                                    }
                                }
                                geometry Sphere {
                                    radius 0.6
                                }
                            }
                        ]
                    }
                ]
            }
        }
    DEF clock TimeSensor {
        cycleInterval 8.0
        loop TRUE
    }
    DEF path ColorInterpolator {
        key [0 0.15 0.3 0.45 0.6 0.75 0.90 1.0]
        keyValue [
            1.0 0.0 0.0,
            0.5 0.0 0.0,
            1.0 1.0 0.0,
            0.0 1.0 0.0,
            0.0 1.0 1.0,
            0.0 0.0 1.0,
            1.0 0.0 1.0,
            1.0 1.0 1.0,
        ]
    }
    ROUTE clock.fraction_changed TO path.set_fraction
    ROUTE path.value_changed TO yzs.diffuseColor

```

运行程序，效果如图 5-6 和图 5-7 所示。

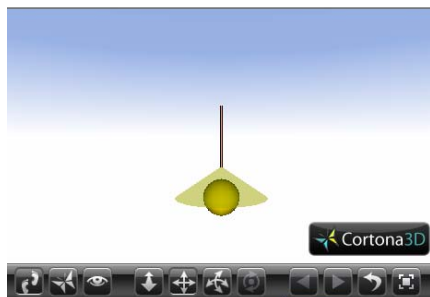


图 5-6 改变颜色的动画效果 1

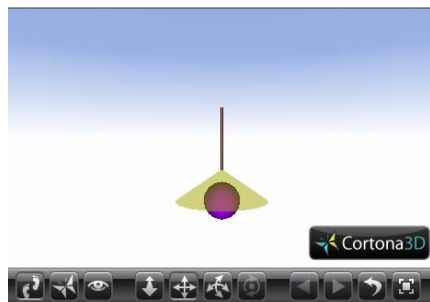


图 5-7 改变颜色的动画效果 2

5.2.4 朝向插补器

利用 `OrientationInterpolator` 动态可改变观察方向，或者改变形体的方向。`OrientationInterpolator` 节点的输出是 `SFRotation` 类型，被设计成适于作为旋转节点的输入。例如 `Transform` 节点的 `rotation` 域，使用一个 `SFRotation` 值来指定该节点围绕旋转的旋转轴和角度。使用步骤如下：

- (1) 给需要变化的方向的 `Viewpoint` 节点命名，或者给需要旋转形体所在的 `Transform` 节点命名。
- (2) 定义一个 `TimeSensor`，确定变化周期及循环方式。
- (3) 定义一个 `OrientationInterpolator` 节点，给出变化的具体方式。
- (4) 给出两个 `ROUTE` 语句：一个将 `TimeSensor` 的变化给 `OrientationInterpolator`，一个将 `OrientationInterpolator` 的变化传给 `Transform` 或 `Viewpoint`。

`OrientationInterpolator` 节点语法如下：

节点名称	域名称	域值	#域及域值类型
<code>OrientationInterpolator{</code>			
	<code>set_fraction</code>		<code>#eventIn MFFloat</code>
	<code>key</code>	<code>[]</code>	<code>#exposedField MFFloat</code>
	<code>keyvalue</code>	<code>[]</code>	<code>#exposedField MFRotation</code>
	<code>value_changed</code>		<code>#eventOut MFRotation</code>
<code>}</code>			

其中：

- (1) `key` 域的域值关键帧时间列表，分别代表所占总动画的时间的比率（即介于 0.0~1.0 之间的浮点数，0.0 和 1.0）。
- (2) `keyvalue` 域的域值列表，每个值对应一个关键帧，用于在其间插值。它指定了一个旋转关键值的列表。每一个旋转关键值是一个 4 个值的组。前 3 个值指定了一个旋转轴的 X、Y 和 Z 分量。第 4 个值指定了旋转轴的一个旋转角度，如在此前讨论的旋转轴和旋转角度。`keyValue` 域的默认值是一个空的列表。

关键时刻和关键旋转值在一起使用，目的是第一个关键时刻指定第一个关键旋转值，第二个关键时刻指定第二个关键旋转值，如此继续。这个列表可以提供任意数量的时刻和旋转，但是它们的列表必须包含相同数量的值。

当时刻由 `OrientationInterpolator` 节点接收到时，它将计算基于关键旋转和相应时刻列表中的旋转值。新计算出的旋转值由 `value_changed` 域（`eventOut` 出事件）输出。

一个时刻 `t` 的输出由 `OrientationInterpolator` 节点来计算：

- ① 扫描关键时刻值，找到一个时间邻域 `t1` 和 `t2`， $t1 \leq t \leq t2$ 。
- ② 查出一对相应的旋转关键值。
- ③ 在关键旋转值中线性内插来计算一个中间位置。

两个相邻的关键值 `t1` 和 `t2` 可以用相同的值，其目的是在一个动画中建立一个不连续跳跃过程。在这种情况下，当线性内插时间小于 `t1`（两个指定的关键时刻中的第一个）时，与 `t1` 相应的关键位置被 `PositionInterpolator` 节点使用。当线性内插时间大于 `t2`（两个指定的关键时刻的第二个）时，与之相关的关键位置就被 `PositionInterpolator` 节点使用。关键时刻和关键旋转的列表可以通过将新值传送到公共域 `key` 和 `keyValue` 的隐含 `set_key` 和 `set_keyValue` 域


```

DEF clock TimeSensor {          #定义时间传感器节点
    cycleInterval 9
    loop TRUE
}
DEF path1 OrientationInterpolator #定义第一根棒的朝向插补器
    key [
        0 0.5 1                #时间关键点
    ]
    keyValue [                  #朝向关键值
        0 0 1 0
        0 0 1 3.141
        0 0 1 6.281
    ]
}
DEF path2 OrientationInterpolator { #定义第二根棒的朝向插补器
    key [
        0 0.5 1                #时间关键点
    ]
    keyValue [                  #朝向关键值
        0 0 1 1.571
        0 0 1 4.712
        0 0 1 7.851
    ]
}
ROUTE clock.fraction_changed TO    path1.set_fraction
ROUTE clock.fraction_changed TO    path2.set_fraction
ROUTE path1.value_changed TO    bar1.rotation
ROUTE path2.value_changed TO bar2.rotation

```

运行程序，效果如图 5-8 所示。

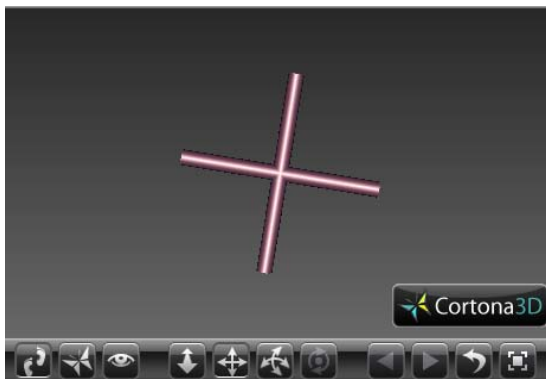


图 5-8 旋转动画

【例 5-6】 使用 OrientationInterpolator 朝向插补器和 ColorInterpolator 颜色插补器，让十字架在旋转的同时颜色也发生变化。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

DEF bar1 Transform {                                #定义第一根棒坐标变换节点
    children [
        Shape {
            appearance Appearance {
                material DEF cbar1 Material {
                    diffuseColor 0.5 0.3 0.4
                    ambientIntensity 0.4
                    specularColor 0.7 0.7 0.6
                    shininess 0.2
                }
            }
            geometry Cylinder {
                height 5
                radius 0.2
            }
        }
    ]
}

DEF bar2 Transform {                                #定义第二根棒坐标变换节点
    children [
        Shape {
            appearance Appearance {
                material DEF cbar2 Material {
                    diffuseColor 0.5 0.3 0.4
                    ambientIntensity 0.4
                    specularColor 0.7 0.7 0.6
                    shininess 0.2
                }
            }
            geometry Cylinder {
                height 5
                radius 0.2
            }
        }
    ]
}

DEF clock TimeSensor {                               #定义时间传感器节点
    cycleInterval 9
    loop TRUE
}

DEF path OrientationInterpolator {                   #定义第一根棒的朝向插补器
    key [
```

```

        0 0.5 1                                #时间关键点
    ]
    keyValue [                                #朝向关键值
        0 0 1 0
        0 0 1 3.141
        0 0 1 6.281
    ]
}
DEF path2 OrientationInterpolator {           #定义第二根棒的朝向插补器
    key [
        0 0.5 1                                #时间关系点
    ]
    keyValue [                                #朝向关键点
        0 0 1 1.571
        0 0 1 4.712
        0 0 1 7.851
    ]
}
DEF cpath1 ColorInterpolator {               #定义第一根棒的颜色插补器
    key [
        0 0.5 1                                #时间关键点
    ]
    keyValue [                                #颜色关键值
        1 0 0
        0 1 0
        0 0 1
    ]
}
DEF cpath2 ColorInterpolator {               #定义第二根棒的颜色插补器
    key [                                     #时间关键点
        0 0.5 1
    ]
    keyValue [                                #颜色关键值
        0.8 0.6 0.2
        0.6 0.4 0.9
        0.1 0.8 0.4
    ]
}
ROUTE clock.fraction_changed TO path.set_fraction
ROUTE clock.fraction_changed TO path2.set_fraction
ROUTE clock.fraction_changed TO cpath1.set_fraction
ROUTE clock.fraction_changed TO cpath2.set_fraction
ROUTE path.value_changed TO bar1.rotation
ROUTE path2.value_changed TO bar2.rotation
ROUTE cpath1.value_changed TO cbar1.diffuseColor
ROUTE cpath2.value_changed TO cbar2.diffuseColor

```

运行程序，效果如图 5-9 和图 5-10 所示。

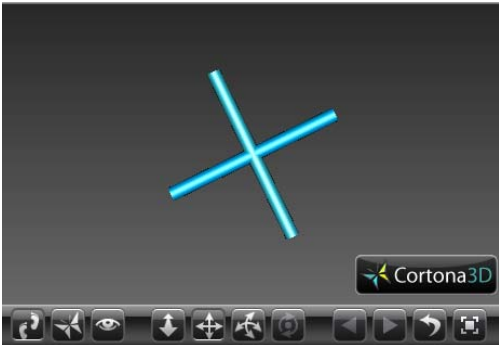


图 5-9 旋转变色动画 1

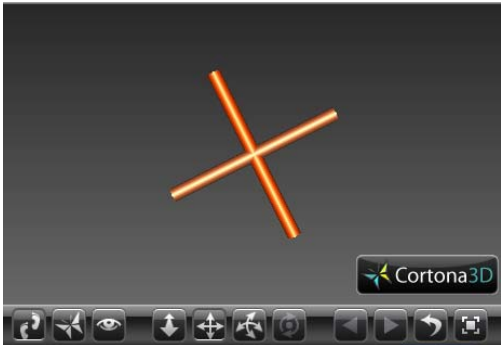


图 5-10 旋转变色动画 2

5.2.5 标量插补器

ScalarInterpolator 节点输出一个常用的浮点值，用于描述浮点数变量变化。如可以改变透明程度、零的影响范围，改变 Material 节点的 transparency 字段，从而改变形体的透明效果；或用来动态改变 Fog 节点 visibilityRange 字段，从而改变雾的影响响应。

ScalarInterpolator 节点在它的 key 和 keyValue 域中使用一系列的关键时刻值和关键浮点值。在一个 TimeSensor 节点的时刻输出驱动下，ScalarInterpolator 节点使用线性内插计算出中间的浮点值，使用 Value_changed 域（eventOut 出事件）输出。可以将该输出路由到 Material 节点的 transparency 域来动态改变造型的透明度。使用步骤如下：

- （1）给 Material 或 Fog 节点命名，Material 通常是 Appearance 节点 material 字段的具体内容。
- （2）定义一个 TimeSensor 节点，确定变化周期及循环方式。
- （3）定义一个 ScalarInterpolator 节点，给出变化的具体方式。
- （4）给出两个 ROUTE 语句：一个将 TimeSensor 的变化传给 ScalarInterpolator，一个将 ScalarInterpolator 的变化传给 Material 的 transparency 字段或 Fog 的 visibilityRange 字段。

ScalarInterpolator 节点语法如下：

节点名称	域名称	域值	#域及域值类型
ScalarInterpolator{	set_fraction		#eventIn MFFloat
	key	[]	#exposedField MFFloat
	keyvalue	[]	#exposedField MFFloat
	value_changed		#eventOut MFFloat
	}		

其中：

- （1）key 公共域的值表示一系列关键的浮点时刻值。通常情况下，时刻值在 0.0~1.0 区间内，像 TimeSensor 节点的 fraction_changed 域（eventOut 出事件）输出。关键时刻值可以是任意形式的正或负浮点值。关键时刻必须以非递减的方式表示。key 域的默认值是一个空格。
- （2）keyvalue 公共域的值表示一系列的域是浮点数值 keyValue 域的默认值是一个空的列表。关键时刻和关键值在一起使用，第一个关键时刻值表示第一个关键值对应的时刻。第二个

关键时刻值表示第二个关键值对应的时刻，以此类推。表中可以提供任意数目的时刻和值，但是它们的数目必须相等。

当一个时刻值被 `set_fraction` 域（`eventIn` 入事件）收到时，`ScalarInterpolator` 节点就在关键值和它们相应的关键时间值的基础上计算出一个值。新计算的值用 `value_changed` 域（`eventOut` 出事件）输出。

在时刻 t ，一个输出值由 `ScalarInterpolator` 节点计算：

- ① 扫描关键时刻值，找到一对相近的值 t_1 和 t_2 ，其中 $t_1 \leq t \leq t_2$ 。
- ② 检索对应的关键值。
- ③ 使用线性内插，在关键值之间计算一个中间值。

两个相近的关键时刻段 t_1 和 t_2 仅可以取相等的值，从而在动画过程中，创建一个中断或跳跃。如此，当线性内插的时刻值小于 t_1 时，由 `ScalarInterpolator` 节点使用与 t_1 对应的关键值，当线性插入的时间值大于 t_2 时，使用与之对应的关键值。

通常情况下，时刻值由 `TimeSensor` 节点的 `fraction_changed` 域（`eventOut` 出事件）的输出路由到 `set_fraction` 域（`eventIn` 入事件）。时刻值被这些输入接收时，则相对域的值就发生变化，新的值使用公共域的隐含 `key_changed` 域（`eventOut` 出事件）和 `keyValue_changed` 域（`eventOut` 出事件）输出。

`ScalarInterpolator` 节点不创建造型，在空间中没有视觉效果。`ScalarInterpolator` 节点可以作为任何编组节点的子节点，但是独立于所使用的坐标系。通常情况下，插补器被安置在 VRML 文件的最外层组节点的末端。

(3) `value_changed` 为 `MFFloat` 值类型的 `eventOut` 事件，输出与比率相对应的浮点数值。

(4) `set_fraction` 为 `SFFloat` 值类型的 `eventIn` 事件，控制动画的完成比率。

`ScalarInterpolator` 节点允许对单精度浮点数进行动画关键帧的插值操作。建立插补器时，为动画的不同完成比率设置相应的浮点数（最好包括开始值和结束值）。通常坐标插补器从时间传感器接收 `set_fraction` 事件，经处理后，将输出值发送给类型为单精度浮点数的域，如 `intensity` 和 `radius`。

【例 5-7】 创建一条绿色大道，并利用了 `ScalarInterpolator` 节点的功能改变 `DirectionalLight` 节点中 `ambientIntensity` 字段值，使绿色大道四周的光线强度有所变化，会产生由暗到亮的动态效果。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Group {
  children [
    DEF sunlight DirectionalLight {
      direction 0.8 -0.2 -0.2
      intensity 1.0
      ambientIntensity 0.5
      color 1.0 0.0 0.0
    }
    DEF clock TimeSensor {
      cycleInterval 5.0
```

```

        loop TRUE
    }
    DEF lightambient ScalarInterpolator {
        key [
            0.0 0.5 1.0
        ]
        keyValue [
            0.2 0.5 0.8
        ]
    }
    Group {
        children [
            Shape {
                appearance Appearance {
                    material Material {}
                }
                geometry Box {
                    size 6 0.1 20
                }
            }
            Transform {
                translation -3 0 10
                scale 0.5 1 0.5
                children Inline {
                    url "tree.wrl"
                }
            }
            Transform {
                translation -3 0 5
                scale 0.5 1 0.5
                children Inline {
                    url "tree.wrl"
                }
            }
            Transform {
                translation -3 0 0
                scale 0.5 1 0.5
                children Inline {
                    url "tree.wrl"
                }
            }
            Transform {
                translation -3 0 -5
                scale 0.5 1 0.5
                children Inline {
                    url "tree.wrl"
                }
            }
        ]
    }

```

```

    }
}
Transform {
    translation -3 0 -10
    scale 0.5 1 0.5
    children Inline {
        url "tree.wrl"
    }
}
Transform {
    translation 3 0 10
    scale 0.5 1 0.5
    children Inline {
        url "tree.wrl"
    }
}
Transform {
    translation 3 0 5
    scale 0.5 1 0.5
    children Inline {
        url "tree.wrl"
    }
}
Transform {
    translation 3 0 0
    scale 0.5 1 0.5
    children Inline {
        url "tree.wrl"
    }
}
Transform {
    translation 3 0 -5
    scale 0.5 1 0.5
    children Inline {
        url "tree.wrl"
    }
}
Transform {
    translation 3 0 -10
    scale 0.5 1 0.5
    children Inline {
        url "tree.wrl"
    }
}
}
]
}

```

```
    ]  
  }  
  ROUTE clock.fraction_changed TO    lightambient.set_fraction  
  ROUTE lightambient.value_changed TO    sunlight.ambientIntensity
```

运行程序，效果如图 5-11 所示。

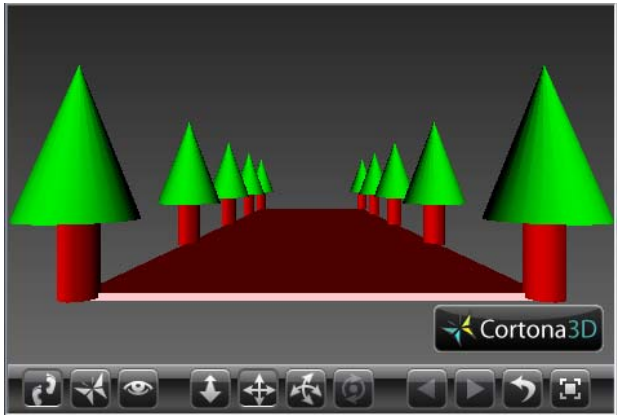


图 5-11 不断由暗变亮的绿色大道效果

5.2.6 坐标插补器

CoordinateInterpolator 节点用于产生基于坐标点的复杂造型（如 Indexed LineSet 线集节点造型、InexdeFactSet 面集节点造型等）的变形动画。该节点并不创建造型，可以作为任何组节点的子节点。

CoordinateInterpolator 节点语法如下：

节点名称	域名称	域值	#域及域值类型
CoordinateInterpolator{			
	set_fraction		#eventIn MFFloat
	key	[]	#exposedField MFFloat
	keyvalue	[]	#exposedField MFVec3f
	value_changed		#eventOut MFVec3f
	}		

其中：

- (1) value_changed 为出事件，用于输出计算后新的坐标子列表。
- (2) set_fraction 为入事件，用于不断接收来自时间传感器发出的时刻比例数值。每收到一个时刻值，CoordinateInterpolator 节点就在时间关键点和其相应的关键坐标点的子列表的基础上计算出一个新的坐标子列表，并通过 value_changed 出事件发送出去。
- (3) key 域的域值用于设定一组时间关键点的列表。每一个关键点是一个浮点时刻值，与接收到的入事件 set_fraction 相对应。时刻值一般在 0.0~1.0 范围内取值，代表一个时间周期中的相对时刻，对应于时间传感器发出的 fraction_changed 出事件值，并要求依次递增排列。该域值的默认值为空列表。
- (4) keyvalue 域的域值用于设定一组关键坐标点的列表。每一个坐标点都包含 X、Y、Z 三个分量，关键坐标点与 key 域的时间关键点一一对应。该域值的默认值为空列表。

【例 5-8】 创建一个正方形变化例子。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyColor 0.0 0.0 0.2
}

Shape {
    appearance Appearance {
        material Material {
            emissiveColor 1.0 1.0 1.0
        }
    }
    geometry IndexedLineSet {
        coord DEF square Coordinate {
            point [
                0.0 0.0 0.0
                1.0 0.0 0.0
                2.0 0.0 0.0
                2.0 1.0 0.0
                2.0 2.0 0.0
                1.0 2.0 0.0
                0.0 2.0 0.0
                0.0 1.0 0.0
            ]
        }
        coordIndex [
            0 1 2 3 4 5 6 7 0
        ]
    }
}

DEF clock TimeSensor {
    cycleInterval 6.0
    loop TRUE
}

DEF start CoordinateInterpolator {
    key [
        0.0 1.00
    ]
    keyValue [
        0.0 0.0 0.0
        1.0 0.0 0.0
        2.0 0.0 0.0
        2.0 1.0 0.0
        2.0 2.0 0.0
    ]
}
```

```
1.0 2.0 0.0
0.0 2.0 0.0
0.0 1.0 0.0
0.0 0.0 0.0
1.0 0.5 0.0
2.0 0.0 0.0
1.5 1.0 0.0
2.0 2.0 0.0
1.0 1.5 0.0
0.0 2.0 0.0
0.5 1.0 0.0
]
}
ROUTE clock.fraction_changed TO start.set_fraction
ROUTE start.value_changed TO squire.point
```

运行程序，效果如图 5-12 所示。

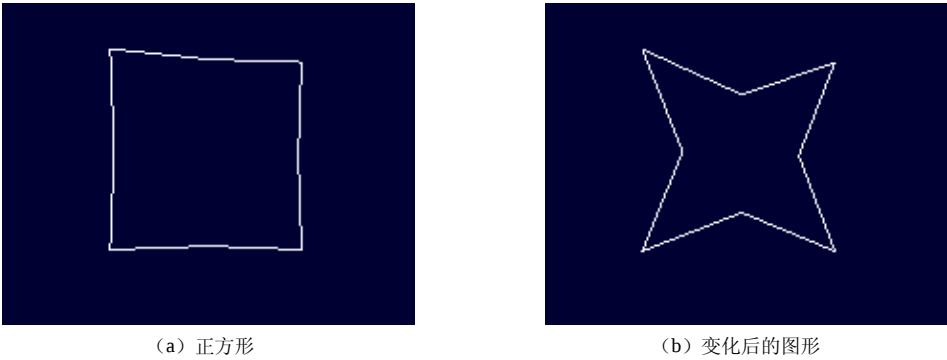


图 5-12 变化效果

5.2.7 法向量插补器

此节点允许对一组法向量进行关键帧动画插值操作。建立插补器时，为动画中得到每个指定时刻选择一组法向量（最好包括动画的开始和结束时刻），并且在 keyValue 域中列出这些法向量。在 key 域中列出相应比率（一组法向量对应一个值）。NormalInterpolator 节点使用 key 域和 keyValue 域中的关键时刻列表和关键法向量矢量列表。当 ImeSensor 节点的时刻输出驱动时，NormalInterpolator 节点用线性插入算法法向量矢量列表的中间值。每一个新的计算出的法向量矢量列表输出到节点 value_changed 域（eventOut 出事件）。可以发送一个 Normal 节点使造型产生动画效果。

NormalInterpolator 节点语法如下：

节点名称	域名称	域值	#域及域值类型
NormalInterpolator{	set_fraction		#eventIn MFFloat
	key	[]	#exposedField MFFloat
	keyvalue	[]	#exposedField MFVec3f
	value_changed		#eventOut MFVec3f
	}		

其中:

(1) **key** 为公共域指定了一个关键浮点时刻列表。一般来说,时刻在 0.0~1.0 之间,即如由 **TimeSensor** 节点的 **fraction_changed** 域(**eventOut** 出事件)输出的那样。虽然如此,时刻可为任意大小的正、负浮点数。关键时刻必须以非减少的顺序排列。**key** 域的默认值为一空列表。

(2) **keyValue** 域是法向量组的列表,每一组值对应一个关键帧,用于在其间插值。公共域的值指定了一个关键法向量矢量列表。每个法向量矢量被说明为 3 个浮点值,分别为此向量的 X、Y、Z 分量点。**keyValue** 域的默认值为一空列表。**set_fraction** 为 **SFFloat** 值类型的 **eventIn** 事件,控制动画的完成比。

(3) **value_changed** 为 **MFVec3f** 值类型的 **eventOut** 事件,输出与动画中给定点相对应的一组标准化向量值。

通常法向量插补器接收来自时间传感器的 **set_fraction** 事件。插值处理后,将输出值送给标号平面几何中 **Noraml** 节点的 **vector** 域中。由于时间传感器发生 0~1 之间的分数值,插补器通常使所有 **key** 值均在此范围内,但并不限于此特定范围。

关键时刻和关键法向量矢量一起使用来说明法向量矢量的子列表,在其中插入一个节点产生一个输出法向量矢量列表。在节点最简单的形式中,一个子列表只包含每个时刻的一个法向量矢量。例如:如果有 N 个时刻值,那么在 **keyValue** 域中恰好有 N 个关键法向量矢量:第一个关键时刻给出第一个关键法向量矢量的时间,第二个关键时刻给出第二个关键法向量矢量的时间,依此类推。在这种简单的插入节点形式中,每个输出法向量矢量列表只包含一个法向量矢量。

在复杂一些的插入节点形式中。**keyValue** 域为每个关键时刻提供了两个域更多的关键法向量矢量。例如:有 N 个关键时刻,每个时刻有 M 个关键法向量矢量,那么 **keyValue** 域包含 $N \times M$ 个法向量矢量,并且节点的输出包含 M 个法向量矢量。

(4) 当 **set_fraction** 域(**eventIn** 入事件)收到一个时刻时,节点根据关键时刻列表和与其对应的关键法向量矢量子列表,计算出一个法向量矢量子列表。新算出的法向量矢量列表由 **value_changed** (**eventOut** 出事件)输出。输出法向量矢量为单位向量。

用 **value_changed** (**eventOut** 出事件)输出的法向量矢量被逐点插入,就像在半径为 1.0 的球体表面上。单位向量对 P 和 Q 用三维坐标,以表示单位球体表面上的位置。输入时,P 和 Q 的中间值沿 P 和 Q 所在球体表面的最短圆弧路径计算。

一个输入时刻 t 的输出法向量矢量列表由 **NormalInterpolator** 节点用以下方法计算:

- ① 扫描关键时刻找出一对相邻时间 t_1 和 t_2 , 要求 $t_1 \leq t \leq t_2$ 。
- ② 取得相应的关键法向量矢量列表。
- ③ 在关键法向量矢量子列表中线性插入,计算出中间法向量矢量子列表。

两个相邻的关键时刻 t_1 和仅可能有相同的值,以便在动画轨迹中产生一个中断或跳跃,在这种情况下,当对比较小的时间进行线性插入时,对应 t_1 (两个相同的关键时刻中的第一个)的关键法向量矢量子列表用于 **NormalInterpolator** 节点。当对比较大的时间进行线性插入时,对应于 t_2 (两个时间中的第二个)的关键法向量矢量子列表被使用。

一般情况下,通过一个路由绑定到 **TimeSensor** 节点的 **fraction_changed** 域(**eventOut** 出事件),时刻被发送到 **set_fraction** (**eventIn** 入事件)。时刻也可以用其他方法产生,例如用产生一般浮点值的 **ScalarInterpolator** 节点。

发送一个值到公共域 `key` 和公共域 `keyValue` 的隐含 `set_key` 和 `set_keyValue` 域（`eventIn` 入事件）可以改变时刻列表或法向量矢量列表。当输入收到此值，对应域的值被改变。新值由此公共域的隐含 `key_changed` 和 `keyValue_changed` 域（`eventOut` 出事件）输出。

`NormalInterpolator` 节点不生成造型，在虚拟世界中没有视觉效果。`NormalInterpolator` 节点可以为任何成组节点的子节点被包含进去，但它独立于使用中的坐标系统。一般来说，插入节点被放置在 VRML 文件最外层组的尾部。

【例 5-9】 使用 `IndexedFaceSet` 节点画出一个平面的正方形，再利用 `NormalInterpolator` 节点配合 `TimeSensor` 节点改变 `IndexedFaceSet` 节点中 `Normal` 节点的 `vector` 字段，`Normal` 向量的变化使此平面正方形会产生对象线的阴影，而这阴影会持续的移动。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

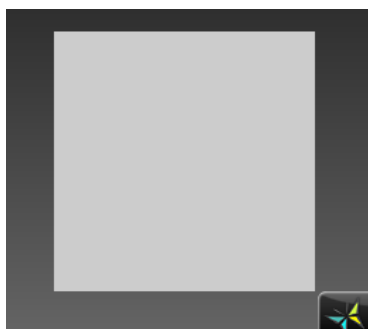
Group {
  children [
    Shape {
      appearance Appearance {
        material Material {
        }
      }
      geometry IndexedFaceSet {
        coord Coordinate {
          point [
            -1.0 -1.0 0.0
            1.0 -1.0 0.0
            1.0 1.0 0.0
            -1.0 1.0 0.0
          ]
        }
        normal DEF shade Normal {
          vector [
            0.0 0.0 1.0
            0.0 0.0 1.0
            0.0 0.0 1.0
            0.0 0.0 1.0
          ]
        }
        solid FALSE
        coordIndex [
          0 1 2 3
        ]
        normalIndex [
          0 1 2 3
        ]
        normalPerVertex TRUE
      }
    ]
  }
}
```

```

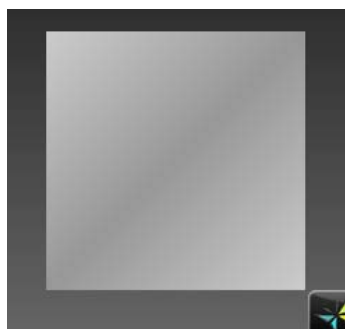
    }
  }
  DEF clock TimeSensor {
    cycleInterval 4.0
    loop TRUE
  }
  DEF normal_path NormalInterpolator {
    key [
      0.0 0.2 0.4 0.6 0.8 1.0
    ]
    keyValue [
      0.0 0.0 1.0,0.0 0.0 1.0
      0.0 0.0 1.0,0.0 0.0 1.0
      1.0 1.0 1.0,0.0 0.0 1.0
      1.0 0.0 1.0,0.0 0.0 1.0
      0.0 0.0 1.0,0.0 0.0 1.0
      0.0 0.0 1.0,0.0 0.0 1.0
      1.0 0.0 1.0,0.0 0.0 1.0
      1.0 0.0 1.0,0.0 0.0 1.0
      0.0 0.0 1.0,0.0 0.0 1.0
      0.0 0.0 1.0,0.0 0.0 1.0
      1.0 0.0 1.0,0.0 0.0 1.0
      1.0 0.0 1.0,0.0 0.0 1.0
      0.0 0.0 1.0,0.0 0.0 1.0
      0.0 0.0 1.0,0.0 0.0 1.0
    ]
  }
}
ROUTE clock.fraction_changed TO normal_path.set_fraction
ROUTE normal_path.value_changed TO shade.vector

```

运行程序，效果如图 5-13 所示。



(a) 绘制的正方形



(b) 阴影移动效果

图 5-13 正方形表面的阴影连续变化图

5.3 交互造型功能

为了方便与同观察者的交互操作，以及实现随观察者观察位置变化显示相应的场景画面，VRML 提供了一些感觉观察者行为动作的节点，在 VRML 中称为交互功能。VR 技术的魅力在于凭借强大的交互功能，使浏览者在虚拟世界中如同身临其境。例如，浏览者点击电视开关，可以播放影视片段；走近宾馆，大门可以自动打开；可以控制造型的位移或旋转等。

5.3.1 交互的基本概念

大多数节点只能创建静态造型，时间传感器和插补器虽然可以使造型产生连续变化的动画效果，但是动画的起止时间和变化规律都是设计者事先设置的，浏览者只是观察而无法参与控制。要想实现人机交互，首先要感知浏览者的操作或者在场景中的观察位置，然后对此操作或者位置移动做出反应。VRML 使用 7 种传感器节点完成感知和反应的交互功能（时间传感器除外，前面已作介绍），其中包括触摸传感器（TouchSensor）、平面传感器（PlaneSensor）、圆柱体传感器（CylinderSensor）、球体传感器（SphereSensor）、接近传感器（ProximitySensor）、可视传感器（VisibilitySensor）、碰撞传感器（Collision）。

第一类传感器：通过感知浏览者对于鼠标的操作行为（移动鼠标、点击鼠标、拖动鼠标），触发并输出事件，实现人机交互的功能。这类传感器有触摸传感器、平面传感器、圆柱体传感器、球体传感器。

第二类传感器：通过感知浏览者在虚拟场景中的观察位置，也就是当浏览者的视点与造型的接近程度达到一定范围的时候，触发并输出事件，实现人机交互功能。这类传感器有、接近传感器、可视传感器、碰撞传感器。

5.3.2 触摸传感器

触摸传感器节点产生基于定点输入设备（通常是鼠标）的事件。这些事件表明用户是否正在点选某个几何体和用户在什么地方及在什么时候按了定点输入设备的按键。

TouchSensor 节点可以添加到任何组中，在这样一个组中，TouchSensor 节点感知观察者对组中造型所做的移动、点击、拖动等动作。TouchSensor 节点能够感知组中所有造型，使可以创建更加复杂的可感知造型。例如，可以在组中建立一个完整的飞机，然后给这个组附加一个 TouchSensor 节点。当观察者点击到飞机的任何部位时，VRML 都会检测到接触，并且将其输出。

TouchSensor 节点可以感知观察者的移动。当观察者的光标移到一组可感知造型上时，TouchSensor 节点就使用它的 isOver 域（eventOut 出事件）输出一个 TRUE。当观察者移到光标离开这个造型时。节点就使用它的 isOver（eventOut 出事件）输出一个 FALSE。使用这些输出，可以将 isOver（eventOut 出事件）输出路由到 TouchSensor 节点的 set_enable 域（eventIn 入事件），从而使时间传感器随着观察者把光标移到造型上或从造型上移开，而开始或停止一个动画。

TouchSensor 节点同样可以感知点击和拖动。当观察者移动光标到一组可感知造型上并且按下鼠标键时，TouchSensor 节点就使用它的 isActive 域（eventOut 出事件）来输出一个 TRUE。当观察者放开鼠标键时，TouchSensor 节点就使用它的 isActive 域（eventOut 出事件）来输出一个 FALSE，并且用 touchTime 域（eventOut 出事件）输出当前的绝对时间。可以使用这些输出来开始和停止一个动画。例如，当一个造型被接触到时，可以把 touchTime 域（eventOut 出事

件)路由到 TouchSensor 节点的 set_startTime 事件,从而引起一个动画开始播放。在移动、点击和拖动的过程中,TouchSensor 节点也输出关于光标在造型什么位置上的信息。这些造型位置输出也可以用来控制动画。

TouchSensor 节点语法如下:

节点名称	域名称	域值	#域及域值类型
TouchSensor{	enable	TRUE	#exposedField SFBool
	hitNormal_changed		#eventOut SFVec3f
	hitPoint_changed		#eventOut SFVec3f
	hitTexcoord_changed		#eventOut SFVce2f
	isActive		#eventOut SFBool
	isOver		#eventOut SFBool
	touchTime		eventOut SFTIME
}			

其中:

(1) enable 为公共域,用来指定传感器是否打开。如果域值为 TRUE,那么传感器打开并且产生输出。如果域值是 FALSE,那么传感器是关闭的,并且没有输出产生,除非它们产生对改变公共域的响应(例如,enable_changed)。这个域的默认值是 TRUE。

当观察者用 TouchSensor 节点在一个可感知造型上移动光标时,传感器节点就通过 isOver 域(eventOut 出事件)输出 TRUE 值。当观察者将光标离开可感知造型时,传感器就使用 isOver 域输出 FALSE 值。

(2) point_changed (eventOut 出事件)为输出击点位置的三维坐标。随着击点的变化,可感知造型上的击点的法向量和纹理坐标被输出,这个输出使用了 hitTexCoord_changed 和 hitNormal_changed 域(eventOut 出事件)。

当光标在一个可感知造型上时,观察者按下鼠标的按键。传感器节点使用 isActive 域(eventOut 出事件)输出 TRUE。当观察者放开鼠标的按键时,传感器节点就使用 isActive 域(eventOut 出事件)输出 FALSE。如果观察者放开这个鼠标的按键时,光标在一个可感知造型上,就使用触 isActive 域(eventOut 出事件)输出 FALSE。使用 touchTime 域(eventOut 出事件)输出当前的绝对时间。

(3) 当鼠标的按键按在一个造型上时,isActive 域(eventOut 出事件)就会输出 TRUE,传感器获得对鼠标的专有使用,直到观察者放开鼠标,isActive 域(eventOut 出事件)输出 FALSE。在这个专有使用过程中,虚拟空间中的其他鼠标传感器不能被激活。

(4) 可以通过把事件路由到隐含的 set_enabled 域(eventIn 入事件)来改变公共域 enabled 的值。如果将这个值由 TRUE 改为 FALSE,并且如果传感器当前是激活的,那么传感器被关闭,使用 isActive 域(eventOut 出事件)输出 FALSE。无论如何,当改变这个域值时,新的域值通过 enable_changed 域(eventOut 出事件)输出。

(5) 接触传感器监视的几何体是传感器的兄弟几何体传感器父组节点的所有子节点。若鼠标指针未指向传感器的兄弟几何体,而用户开始将鼠标指针移到传感器的兄弟几何体时,传感器将产生一个 isOver 事件,并将其值设置为 TRUE;相反,若鼠标指针已经指向传感器的兄弟几何体,此时用户将鼠标指针移出传感器的兄弟几何体,传感器将产生一个 isOver 事件,并将

其值设置为 FALSE。

(6) 当用户将鼠标指针从几何体的一点移动到另一点时, 传感器将发送一系列事件: hitPoint_changed、hitNormal_changed、hitTexCoord_changed, 分别表明用户所指的位置、该点的法向量和纹理坐标。

(7) 当用户点击到被 TouchSensor 监视的对象时, 传感器将产生值为 TRUE 的 isActive 事件; 而当用户释放鼠标的按键时, 传感器将产生 isActive 为 FALSE 的事件。

若用户在指向几何体时按下鼠标键, 然后在仍然指向这个几何体或又回到这个几何体时释放鼠标键, 传感器将要发送一个 touchTime 事件, 表明按键被释放的时间。可以使用这一事件来模拟许多常用的用户接口(如只有在用户点击和释放鼠标按键时才产生的动作)。

当一个接触传感器正处理事件时, 其他鼠标传感器不会产生事件。

接触传感器仅在观察者移动光标时更新它的输出。如果光标不移动, 就不产生更新。如果可感知造型从静止光标下移动出去, 或其他造型以这种方式移动出去, 直到观察者移动光标时, 传感器才会有输出。

如果传感器使用 DEF 和 USE 引用, 传感器感知观察者在其父组中任一造型上的动作。使用步骤如下:

① 在一个局部坐标系中, 给出一个形体及用 DEF 命名的 TouchSensor, 程序运行时, 当鼠标移到形体上时, TouchSensor 传感器准备接受鼠标点击信号, 同时发出 isOver 信号。

② 利用 isOver 或 touchTime 信号, 使程序发生相应变化, 如动画显示、发出声音、改变颜色。

【例 5-10】 利用 TouchSensor 节点, 对椭圆球位移画进行控制。当光标移动到椭圆球造型上时, 就会出现一个小手的图标, 表示这是一个被感应被控制的造型。此时, 按下鼠标左键不放, 椭圆球位移动画开始, 直到松开鼠标左键为止。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

Background {
    skyColor 0.6 0.6 0.6
}

Transform {
    translation 0 0 -30
    children [
        DEF movement Transform {
            scale 1.5 1.5 1.5
            children Inline {
                url "li3_6.wrl"
            }
            #引入创建的徽章
        DEF touch TimeSensor {
            enabled TRUE
        }
        DEF time TimeSensor {
            cycleInterval 30.0
            loop TRUE
            #时间传感器
```

```

        enabled    FALSE
    }
    DEF movementinter PositionInterpolator {          #移动位置节点
        key [
            0.0 0.2
            0.4 0.5
            0.6 0.8
            0.9 1.0
        ]
        keyValue [
            10 6 -30
            10 -6 -20
            -10 -6 -20
            -10 6 -20
            10 6 -20
            -10 -6 -80
            0 0 0
            10 6 -20
        ]
    }
}
ROUTE touch.isActive TO time.enabled
ROUTE time.fraction_changed TO movementinter.set_fraction
ROUTE movementinter.value_changed TO movement.translation
```

在这个例子中可以看到，场景中设置了动画，但是没有给出时钟即时间传感器设定动画开始的时间，起始造型是不动的，当鼠标位于对象上单击的时候，则造型会运动起来。

TouchSensor 检测到鼠标单击的动作，输出事件的时间值作为时间传感器的入事件，也就是传递给时钟一个开始时间，则造型运动起来。

TouchSensor 检测的是鼠标位于对象上单击的情况，常见的鼠标动作还有拖动，在这种情况下，需要用与之对应的检测以及响应方式。

5.3.3 平面传感器

PlaneSensor 节点用于感知观察者拖动动作，并且计算出平移距离，通过使用 translation_changed 域（eventOut 出事件）将这个距离输出。可将鼠标的动作转换成适于操作造型的输出，转变成形体沿自身 Z=0 的平面移动。通过坐标变换，可使形体绕任意轴旋转。可以用鼠标（按住鼠标左键）拖动一个形体，来使自身移动，也可以用鼠标（按住鼠标左键）拖动一个形体，来使另一个形体移动 PlaneSensor 节点可以是任何一个组的子节点，它可以感知观察者对组及其子节点中任何造型的动作。传感器节点的输出通常被路由到 Transform 节点，并引起造型的平移。定点设备的运动，例如，鼠标的运动，当按下鼠标的按键时，就产生一个平移输出，对于观察者来说。这就好像是造型在其父组的坐标系中沿着 XOY 平面滑动。

PlaneSensor 节点语法如下：

节点名称	域名称	域值	#域及域值类型
PlaneSensor {			

minPosition	0 0	#exposedField SFVce2f
maxPosition	-1-1	#exposedField SFVce2f
enabled	TRUE	#exposedField SFBool
offset	0 0 0	#exposedField SFVce3f
autoOffset	TRUE	#exposedField SFBool
isActive		#eventOut SFBool
trackPoint_changed		#eventOut SFVce3f
translation_changed		#eventOut SFVce3f
}		

当观察者按下鼠标的按键把光标移动到可感知造型上时，传感器节点就使用 `isActive` 域（`eventOut` 出事件）输出 `TRUE`。当观察者放开鼠标的按键时，传感器节点就使用 `isActive` 域输出 `FALSE`。

当观察者把光标移动到一个可感知造型上时，在击点位置就会建立一个虚拟的平坦的 `trackplane`（轨迹面），它与传感器父组坐标系中的 `XOY` 平面平行，并且通过击点位置，此点是这个轨迹面的原点。当观察者在鼠标的按键被按下状态移动光标时，`trackpoint`（轨迹点）就从点击的一个初始位置开始沿着轨迹面滑动。水平的光标移动使轨迹点水平滑动，垂直的光标移动使轨迹点垂直滑动。当观察者改变轨迹点的位置时，新的轨迹点的三维轨迹面坐标就由 `trackpoint_changed` 域（`eventOut` 出事件）输出。

（1）每一次观察者改变轨迹点时，`autoOffset`、`offset`、`maxPosition` 和 `minPosition` 域值一起作用将轨迹点转化成平移值，并通过 `translation_changed` 域（`eventOut` 出事件）输出。这个过程包括以下几个部分：

- ① 计算轨迹点和轨迹面之间的距离并产生一个三维坐标值。
- ② 通过增加 `offset` 域值，有选择地使平移值发生偏移。
- ③ 可选择地建立 `minPosition` 和 `maxPosition` 域值，把平移值的范围限制在 `X` 和 `Y` 值之间。
- ④ 使用 `translation_changed` 域（`eventOut` 出事件）输出偏移量和限制的平移值。

输出的平移值路由到一个 `Transform` 节点，当传感器移动一个鼠标时，引起造型在平面里的平移。使用对轨迹点的限制，可以限制可平移造型的移动范围。

这些域值控制的限制和偏移特性并不影响使用 `trackpoint_changed` 域（`eventOut` 出事件）输出轨迹点位置，仅影响在计算 `trackpoint_changed` 域（`eventOut` 出事件）输出的平移值时如何使用轨迹点。

（2）浏览器将使用轨迹点和轨迹面原点之间的距离来计算一个三维平移值。域 `autoOffset` 的值指定平移值是否偏移。`autoOffset` 域指示是否在拖动结束时将当前位置保存在 `offset` 中来实现两次拖动之间跟踪当前位置（`TRUE` 表示跟踪）。若 `autoOffset` 值为 `FALSE`，则用户每次开始新一轮拖动时，被拖动的几何体都自动先复位到初始位置。当 `autoOffset` 域值是 `FALSE` 时，`offset` 域中存储的三维平移值将添加到平移中。`autoOffset` 域值的默认值是 `TRUE`。`offset` 域值的默认值是 `0.0`。

（3）`offset` 域值指出相关几何体被移动后相对于初始点的位置。

（4）`minPosition` 和 `maxPosition` 域值指定 `X` 和 `Y` 平移量限制。`minPosition` 域值指定所允许的最小 `X` 和 `Y` 平移值，而 `maxPosition` 域值，则指定所允许的 `X` 和 `Y` 平移值。平移值就在这个限制之间，任何超出该限制的平移值将被限制到限定值。`minPosition` 域在 `XOY` 平面内限制当前点向上和向右的 `translation` 事件。`maxPosition` 域在地平面内限制当前点向下和向左的

translation 事件, 表示的二维点位于 XOY 平面中此点的左下方。

(5) enabled 域指示传感器当前是否响应鼠标事件。欲关闭此传感器, 向其发送值为 FALSE 的 set_enabled 事件。

可以用下面 3 个方式使用 minPosition 和 maxPosition 域值的 X 分量:

- ① 如果最小 X 值小于最大 X 值, 那么平移就被限制在最小和最大的 X 分量之间。
- ② 如果最小 X 值等于最大 X 值, 那么平移在 X 分量就是这个值。
- ③ 如果最小 X 值大于最大 X 值, 那么平移在 X 方向没有限制。

可以用和 X 分量相同的方式, 来使用 minPosition 和 maxPosition 域值的 Y 分量。通过使用这 3 种方式, 平移可以被限制在 X 或 Y 方向或同时在这两个方向, 这个域值的默认值是建立一个在 X 和 Y 方向没有限制的轨迹点。

在偏移和限制计算的平移值之后, 就使用 translation_changed 域 (eventOut 出事件) 输出这个值。只要用户点击、拖动该传感器的兄弟几何体, 此节点均将拖动 (如用户使用鼠标进行的拖动) 解释为在传感器的局部 XOY 平面上的变换。

若仅对特定范围内的变换值感兴趣, 可通过设置 minPosition 和 maxPosition 将 translation_changed 事件锁定在平面内以这些点为顶点的长方形区域。若 minPosition 的 X 分量大于 maxPosition 的 X 分量, 或者 minPosition 的 Y 分量大于 maxPosition 的 Y 分量, 则转换在该方向不会被锁定。若最小分量中的某一个等于所对应的最大分量, 则转换被限制在另一个方向, 还导致对拖动的解释变为一维的变换, 即将平面传感器变成了直线传感器。

要使几何体被用户移动后仍停留在该处, 置 autoOffset 为 TRUE; 欲使几何体的位置在每一次新的拖动前复位, 置 autoOffset 为 FALSE。

平面传感器被激活, 即用户正在拖动传感器的相关几何体时, 其他鼠标传感器不产生事件。

当观察者放开鼠标的按键时, 传感器停止平移输出。如果 autoOffset 域值是 TRUE, 最后输出的平移值就被存储于节点的 offset 公共域中, 并且使用公共域值的 offset_eventOut 事件输出。使用 VRML 的偏移特性确保了每一个造型的后续拖动开始于前一个拖动的结尾。

(6) isActive 的 eventOut 出事件是 SFBool 类型, 指示鼠标在当前按钮是否按下。此事件仅当按钮被按下或释放时才发出, 拖动期间则不生成。

(7) trackpoint_changed 是 eventOut 出事件, 值 SFVec3f 类型, 拖动期间任何给定时刻用户的鼠标在 XY 平面上的实际点 (忽略 minPosition 和 maxPosition)。

(8) translation_changed 是 eventOut 出事件, 值 SFVec3f 类型, 拖动期间任何给定时刻用户的鼠标在 XY 平面上的锁定点 (受限于 minPosition 和 maxPosition)。

公共域 set_autoOffset、set_offset、set_minPosition 和 set_maxPosition 可以被改变, 这是通过分别路由一个事件到公共域隐含的 set_autoOffset、set_offset、set_minPosition 和 set_maxPosition 域 (eventIn 入事件) 来完成的。当任何一个值被改变, 新的值都由公共域的隐含 autoOffset_changed、offset_changed、minPosition_changed 和 maxPosition_changed 域 (eventOut 出事件) 输出。

【例 5-11】 利用 PlaneSensor 节点, 使用户可以在 XOY 平面上用鼠标任意移动正方体造型, 且造型的移动范围不受限制。

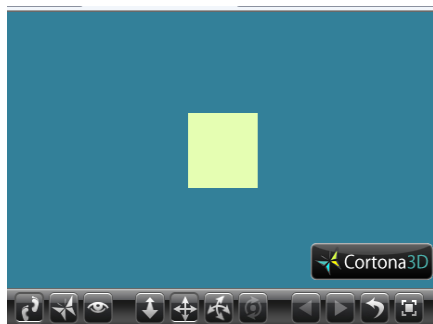
其实现的源程序代码如下:

```

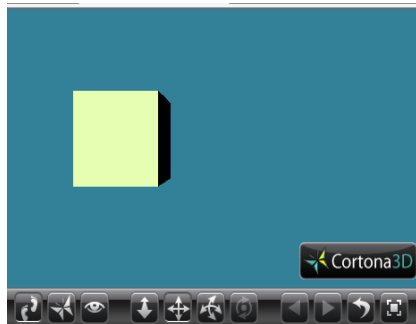
#VRML V2.0 utf8
Background {
    skyColor 0.2 0.5 0.6
}
DEF box Transform {      #定义正方体造型坐标变换节点
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.2 0.3 0.1
                    specularColor 0.7 0.7 0.6
                    ambientIntensity 0.4
                    shininess 0.2
                }
            }
            geometry Box {
            }
        }
    ]
}
DEF sensor PlaneSensor {      #定义平面传感器节点
}
ROUTE sensor.trackPoint_changed TO box.translation

```

运行程序，效果如图 5-14 所示。



(a) 平面传感器造型 1



(b) 平面传感器移动造型 2

图 5-14 平面传感器移动效果

说明：

(1) 首先将被感应的造型节点定义为 **box**，将平面传感器节点定义为 **sensor**。

(2) 路由语句 **ROUTE sensor.trackPoint_changed TO box.translation**，将平面传感器感知用户拖动鼠标的位置，作为出事件发送给造型 **box** 的 **translation** 域，使造型的位置跟随鼠标移动。

【例 5-12】 为 **PlaneSensor** 节点增加两个域值：**offset 2 2 0**，当用户首先用鼠标单击正方体造型时，造型自动从坐标原点位移到 **(2 2 0)** 坐标点；**maxPosition 4 3**，设定了 **X、Y** 方向的最大移动位置，而最小移到位置的默认值为 **minPosition 0 0**，所以用鼠标拖动造型时受到限

制，造型的移动范围是一个矩形；X 方向 0~4，Y 方向 0~3。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyColor 0.2 0.5 0.6
}

DEF box Transform {                                #定义正方体造型坐标变换节点
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.2 0.3 0.1
                    specularColor 0.7 0.7 0.6
                    ambientIntensity 0.4
                    shininess 0.2
                }
            }
            geometry Box {
            }
        }
    ]
}

DEF sensor PlaneSensor {                            #定义平面传感器节点
    offset 2 2 0                                     #设定造型的初始移动位置
    maxPosition 4 3                                  #设定造型的最大移动位置
}

ROUTE sensor.trackPoint_changed TO box.translation
```

运行程序，效果如图 5-15 所示。

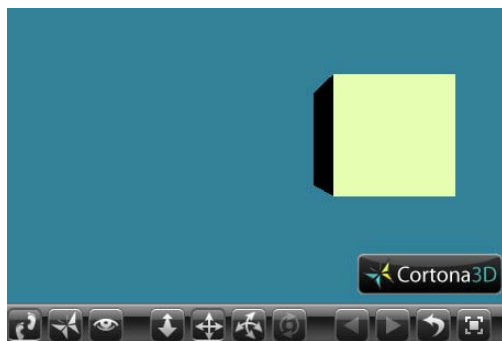


图 5-15 造型首次移动效果

5.3.4 圆柱体传感器

CylinderSensor 节点也可以感知一个观察者的拖动动作，并且计算旋转轴和角度，且通过它的 rotation_changed 域（eventOut 出事件）输出。将鼠标的动作转换成适于操作造型的输出。

CylinderSensor 节点可以是任何组节点的子节点，它可以感知观察者在组及其子节点的任何造型上的动作。通过将传感器节点的输出路由到 Transform 节点来引起造型物体的旋转。如鼠标的运动，当鼠标的按键被按下时就产生旋转输出，对于观察者来说这就好像它们使造型围绕着父节点坐标系的 Y 轴旋转。

CylinderSensor 节点语法如下：

节点名称	域名称	域值	#域及域值类型
CylinderSensor{	minAngle	0	#exposedField SFFloat
	maxAngle	-1	#exposedField SFFloat
	enable	TRUE	#exposedField SFBool
	offset	0	#exposedField SFFloat
	autoOffset	TRUE	#exposedField SFBool
	diskAngle	0.262	#exposedField SFFloat
	isActive		#eventOut SFBool
	trackPoint_changed		#eventOut SFVec3f
	rotation_changed		#eventOut SFRoation
}			

其中：

- (1) enable 域指明传感器是否监控鼠标事件。若想关闭监控，可向它发送一个值为 FALSE 的 set_value 事件。
- (2) isActive 是 eventIn 入事件，值类型为 SFBool，指出用户是否正在拖动被监控的几何体。
- (3) trackPoint_changed 是 eventOut 出事件，值类型为 SFVec3f，输出用户任意时刻所指向的在圆柱或圆盘轨迹面上的轨迹的点。
- (4) rotation_changed 是 eventOut 出事件，值类型为 SFRotation，输出旋转角度。
- (5) 当观察者移动光标在一个柱体可感知造型上时，一个虚拟直线与传感父群组坐标系 Y 轴之间就有一个角度。这个角度和 diskAngle 域值进行比较，根据该角度的大小，CylinderSensor 节点将采取两个相关行为之一。

当观察者高于可感知的造型时，沿 Y 方向向下看，虚拟直线和 Y 轴之间的角度较小。如果这个角度小于 diskAngle，那么 CylinderSensor 节点的动作对于观察者来说就像一个盘子绕中心轴旋转。鼠标指针的圆周运动将使盘子运动，当鼠标指针沿逆时针方向转动时，盘子也逆时针转动；反之亦然。观察者拖动鼠标时，鼠标的位置将通过 trackPoint_changedfak 域（eventOut 出事件）输出。

当观察者在可感知造型旁边或前方、接近垂直于 Y 轴观看时，这个角度是比较大的。如果虚拟直线和 Y 轴之间的角度大于 diskAngle 域值时，CylinderSensor 节点的动作就好像一个围绕 Y 轴旋转的旋转门。鼠标指针的水平动作将使之旋转，当鼠标指针移动到左端时，就使它顺时针旋转；当鼠标指针移动到右边时，就使它逆时针旋转。观察者拖动鼠标时，鼠标的位置将通过 trackPoint_changed 域（eventOut 出事件）输出。

(6) diskAngle 域的默认值是 15°（0.2632rad）。观察者每次改变轨迹点。autoOffset、offset、maxAngle 和 minAngle 域值在一起作用把轨迹点转换成旋转值，并通过 rotation_changed 或（eventOut 出事件）输出。这个过程包括：

- ① 根据在轨迹盘或轨迹柱上的轨迹点位置和中心轴之间的不同，来计算旋转轴和旋转角。

- ② 有选择地加入 `offset` 域值，来使旋转角发生偏移。
- ③ 限制旋转角的范围在 `maxAngle` 和 `minAngle` 域值之间。
- ④ 用 `rotation_changed` 域（`eventOut` 出事件）输出旋转角。

通常情况下，输出的旋转值被路由到 `Transform` 节点，使得观察者运动鼠标指针时，引起造型围绕 Y 轴的旋转。角度范围可以是控制造型旋转的转角范围。

这些域值控制的偏移和范围并不影响由 `trackPoint_changed` 域（`eventOut` 出事件）输出的轨迹点的位置。这些特性仅影响轨迹点如何被用于计算由 `rotation_changed` 域（`eventOut` 出事件）输出的旋转值。

（7）`autoOffset` 域值指定旋转是否应该偏移。如果希望几何体在已旋转的基础上继续旋转，则设置 `autoOffset` 为 `TRUE`，否则设置为 `FALSE`。当 `autoOffset` 域值是 `TRUE` 时，存放于 `offset` 域的旋转角就加入到计算所得的旋转角中。当 `autoOffset` 域值是 `FALSE` 时，不使用 `offset` 域值，旋转值不偏移。默认的 `autoOffset` 域值是 `TRUE`，`offset` 域值是 `0.0`，指出从初始方向转换被监控对象的角度值。`autoOffset` 指出是否在每次拖动完成后将当前的方向角存入 `offset` 域。以达到跟踪当前方位的目的。如果 `autoOffset` 是 `FALSE`，每当用户开始一个新的拖动时，被拖动几何体恢复到最初的方位。

如果用户点中圆柱的顶或底面，拖动动作会被解释为旋转一个平面圆盘一样。如果用户点中圆柱的侧面部分，拖动动作就会被解释为旋转这个圆柱。如果仅仅对特殊的旋转范围有兴趣，可以通过设置 `minAngle` 和 `maxAngle` 的值把旋转限制在某一范围内。如果 `minAngle` 大于 `maxAngle`，就没有任何限制了。当计算的旋转值被偏移和限制后，这个值用 `rotation_changed` 域（`eventOut` 出事件）输出。

（8）公共域 `autoOffset`、`offset`、`minAngle` 和 `maxAngle` 的值也可以通过将一个事件分别路由到这些公共域的隐含 `set_autoOffset`、`set_offset`、`set_minAngle` 和 `set_maxAngle` `eventIn` 事件来改变，当这些域值中的任何一个改变时，新的值就用公共域值分别通过隐含的 `autoOffset_changed`、`offset_changed`、`maxAngle_changed` 和 `minAngle_changed` 域（`eventOut` 出事件）来输出。

【例 5-13】 利用 `CylinderSensor` 节点，创建一个旋转门的场景。旋转门造型的玻璃门在鼠标拖动下可以旋转任意角度，因为设置了初始旋转角度 `offset -0.781`，所以当鼠标首先点击时，玻璃门扇自动顺时针旋转 45° 。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyColor 0.6 0 0
}

Transform {                                #门框造型
    translation 0 2.1 0
    children [
        DEF up Shape {                    #定义上门框造型
            appearance Appearance {
                material Material {
                    diffuseColor 0.3 0.2 0.0
                    ambientIntensity 0.4
```

```

                                specularColor 0.7 0.7 0.6
                                shininess 0.2
                            }
                        }
                    geometry Cylinder {
                        radius 3.8
                        height 0.2
                    }
                }
            ]
        }
    Transform {
        translation 0 -2.1 0
        children [
            USE up
        ]
    }
    Transform {
        translation 3.4 0 0
        children [
            DEF right Shape {                                #定义右门框造型
                appearance Appearance {
                    material Material {
                        diffuseColor 0.3 0.2 0.0
                        ambientIntensity 0.4
                        specularColor 0.7 0.7 0.6
                        shininess 0.2
                    }
                }
                geometry Cylinder {
                    radius 0.4
                    height 4
                }
            }
        ]
    }
}
Transform {
    translation -3.4 0 0
    children [
        USE right
    ]
}
DEF door Transform {                                #定义玻璃旋转门坐标变换节点
    children [
        DEF door1 Shape {                                #定义玻璃门扇造型节点
            appearance Appearance {

```

```

        material Material {
            diffuseColor 0.05 0.46 0.73
            shininess 0.31
            specularColor 1 1 1
            emissiveColor 0.03 0.04 0.2
            transparency 0.45
        }
    }
    geometry Box {
        size 6 4 0.1
    }
}
Transform {
    rotation 0 1 0 1.571
    children [
        USE door1
    ]
}
Shape {                                #门的旋转轴造型
    appearance Appearance {
        material Material {
            diffuseColor 0.5 0.5 0.7
            ambientIntensity 0.4
            shininess 0.2
            specularColor 0.8 0.8 0.9
        }
    }
    geometry Cylinder {
        radius 0.15
        height 4
    }
}
]
}
DEF clock TimeSensor {                  #定义时间传感器
    cycleInterval 5
}
DEF path CylinderSensor {               #定义圆柱体传感器
    offset -0.781                        #设定初始旋转角度
}
ROUTE path.rotation_changed TO door.rotation

```

运行程序，效果如图 5-16 及图 5-17 所示。

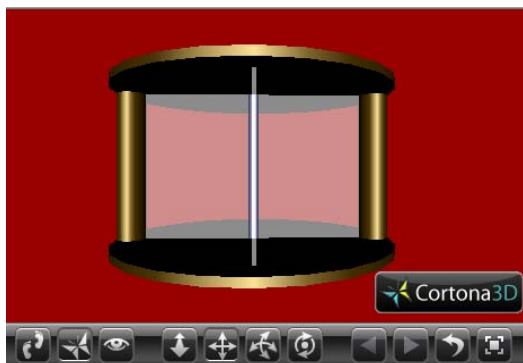


图 5-16 旋转门造型

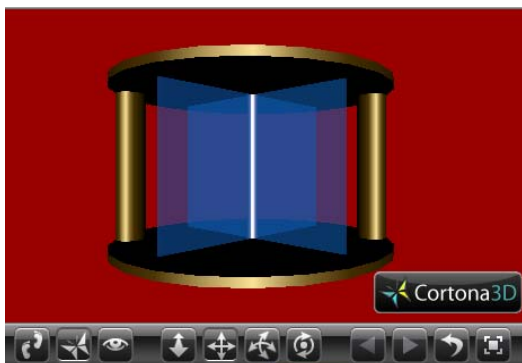


图 5-17 鼠标首先单击后旋转门效果

【例 5-14】 创建一个“蜻蜓”玩具的旋转造型。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
Background {
    skyColor [
        0.2 0.5 0.6
    ]
}
Group {
    children [
        DEF fan Transform {
            translation 0.0 -1.0 -5.0
            scale 0.5 0.5 0.5
            children Inline {
                url "qiting.wrl"
            }
        }
    ]
}
DEF touch TouchSensor {}
DEF time TimeSensor {
    cycleInterval 1.5
}
DEF xy OrientationInterpolator {
    key [
        0.0 0.2 0.4 0.6 0.8 1.0
    ]
    keyValue [
        0.0 1.0 0.0 0.0
        0.0 1.0 0.0 -1.256
        0.0 1.0 0.0 -2.512
        0.0 1.0 0.0 -3.768
        0.0 1.0 0.0 -5.024
        0.0 1.0 0.0 -7.280
    ]
}
```

```

    }
    DEF fangs CylinderSensor {
        autoOffset TRUE
        diskAngle 0.262
        enabled TRUE
        autoOffset TRUE
        maxAngle -1.0
        minAngle 0.0
        offset 2.618
    }
}

ROUTE fangs.rotation_changed TO fan.scaleOrientation
ROUTE touch.touchTime TO time.startTime
ROUTE time.fraction_changed TO xy.set_fraction
ROUTE xy.value_changed TO fan.rotation

```

运行程序，效果如图 5-18 和图 5-19 所示。



图 5-18 创建的“蜻蜓”玩具造型



图 5-19 旋转的“蜻蜓”玩具

在程序中调用到“qiting.wrl”程序创建“蜻蜓”玩具造型。其源程序代码如下：

```
#VRML V2.0 utf8
```

```

Transform {
    #创蜻蜓玩具造型
    translation 0.0 4.9 0.0
    scale 0.8 0.5 0.8
    children [
        Shape {
            appearance Appearance {
                material Material {
                    #空间物体造型外观
                    diffuseColor 0.3 0.2 0.0 #一种材料的漫反射颜色
                    ambientIntensity 0.4
                    specularColor 0.7 0.7 0.6
                    shininess 0.2
                }
            }
        }
    ]
}

```

```
        }
        geometry Sphere {
            radius 0.5
        }
    ]
}
Transform {
    translation 0.0 1.0 0.0
    scale 0.5 1.0 0.5
    children [
        DEF leaf Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.3 0.2 0.0
                    ambientIntensity 0.4
                    specularColor 0.7 0.7 0.6
                    shininess 0.2
                }
            }
            geometry Cylinder {
                radius 0.2
                height 8.0
                top TRUE
                bottom TRUE
                side TRUE
            }
        }
        Transform {
            translation 3.8 4.0 0.0
            scale 20.0 0.03 1.5
            rotation 1.0 0.0 0.0 0.524
            children [
                USE leaf]
            ]
        }
        Transform {
            translation -3.8 4.0 0.0
            scale 20.0 0.03 1.5
            rotation 1.0 0.0 0.0 -0.524
            children [
                USE leaf ]
            ]
        }
    ]
}
```

5.3.5 球体传感器

SphereSensor 节点也用于感知观察者的拖动动作，并且计算旋转轴和角度。通过使用它的

rotation_changed 域（eventOut 出事件）输出。可将鼠标的动作转换成适于操作造型的输出。SphereSensor 节点可以是任何组节点的子节点，它可以感知观察者在组及其子节点的任何造型上的动作。传感器节点的输出被路由到 Transform 节点，这样就可以使造型旋转。鼠标的移动，如鼠标的移动，在鼠标的按键被按下时就产生一个旋转输出，这对于观察者来说就好像是在滚动一个球体。可使鼠标的移动转变成形体绕自身原点的转动，可以用鼠标（按住鼠标左键）拖动一个形体，来使自身转动，也可以鼠标（按住鼠标左键）拖动一个形体，来使另一个形体转动。

SphereSensor 节点语法如下：

节点名称	域名称	域值	#域及域值类型
SphereSensor{	enabled	TRUE	#exposedField SFBool
	offset	0 1 0	#exposedField SFRotation
	autoOffset	TRUE	#exposedField SFBool
	isActive		#eventOut SFBool
	trackPoint_changed		#eventOut SFVec3f
	rotation_changed		#eventOut SFRoation
	}		

当观察移动光标到一个可感知造型上时，击点的位置就建立了一个虚拟轨迹球体表面，这个轨迹球体的中心在传感器父组的坐标系中心。坐标系原点和击点距离就是轨迹球体的半径。

在鼠标的按键被按下时，观察者每次移动光标。一个轨迹点就在轨迹球的表面滑动，它在击点以一个初始位置开始。水平移动光标轨迹点就在轨迹球上东西方向滑动。当光标垂直移动时，轨迹点就在轨迹球上南北方向移动。当观察者改变轨迹点的位置时，新的轨迹点的三维轨迹球坐标就由 trackPoint_changed 域（eventOut 出事件）输出。

每一次观察者改变轨迹点时，autoOffset 和 offset 域值就一起作用将轨迹点转化成为一个旋转值，并由 rotation_changed 域（eventOut 出事件）输出。这个过程包括：

- ① 使用轨迹点和击点位置角度的不同计算一个旋转轴和旋转角。
- ② 通过添加 offset 域值，来有选择地偏移旋转值。
- ③ 使用 rotation_changed 域输出偏移旋转。

由这些域值来控制的偏移特性不影响由 trackPoint_changed 域（eventOut 出事件）输出的轨迹点的位置。这些特性仅影响在计算旋转值时如何使用轨迹点，该旋转值将通过 rotation_changed 域输出。

（1）enabled 域表示传感器当前是否检测鼠标的事件。可向该传感器发送一个值为 FALSE 的 set_enabled 时间来关闭它。

（2）公共域 autoOffset 和 offset 的值可以被改变，这是通过将一个事件路由到公共域隐含的 set_autoOffset 和 set_offset 域（eventIn 入事件）中来实现的。当这些域值中的任何一个被改变时，新的值就通过该公共域隐含的 autoOffset_changed 域（eventOut 出事件）和 offset_changed 域（eventOut 出事件）来输出。

（3）autoOffset 域值来指定旋转值是否发生偏移。当 autoOffset 域值是 TRUE 时，旋转值在通过 rotation_changed 域输出前，将 offset 域中存储的三维的旋转值加入原旋转值中。当

autoOffset 域值是 FALSE 时, offset 域值将不被使用, 原来的旋转值将通过 offset_changed 域输出。autoOffset 域的默认值为 TRUE, offset 域值的默认值为 0。autoOffset 域设置为 TRUE, 表示在拖动结束时, 将当前的方位值存储到 offset 中, 若 autoOffset 设置为 FALSE, 在用户每次开始一个新的拖动时, 几何体复位到初始值。offset 表示在一次拖动后, 相关几何体相对于初始位置的旋转角度。

当观察者放开鼠标按键时, 传感器就停止输出。如果 autoOffset 域值是 TRUE, 那么最后一个输出的旋转值就被存入到该节点的 offset 公共域中, 并且通过该公共域隐含的 offset_changed 域来输出。使用 VRML 的偏移特性可以确保每一个后续拖动在上一个拖动的结尾开始。

(4) isActive 是 eventOut 出事件, 值是 SFBool 类型, 指出用户是否在拖动与传感器相关的几何体。

(5) trackPoint_changed 是 eventOut 出事件, 值是 SFVec3f 类型, 用户在拖动过程中任意时刻鼠标在假想的球体表面上的实际位置。

(6) rotation_changed 是 eventOut 出事件, 值是 SFRoation 类型, 用户在拖动过程中的任意时刻假想球体的当前朝向, 即旋转角度。

SphereSensor 节点将二维的拖动解释为三维空间中绕局部原点的旋转。若在该节点的兄弟几何体处按下鼠标, 浏览会以鼠标点击的点到原点的距离为半径作一个想象中的球。随后的拖动将解释为旋转球体。

【例 5-15】 SphereSensor 节点应用示例。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

Group {
  children [
    DEF sphere Transform {
      children Shape {
        appearance Appearance {
          material Material {
            diffuseColor 0.0 0.6 0.5
          }
        }
        geometry Sphere {}
      }
    }
    DEF senor SphereSensor {}
  ]
}

ROUTE senor.rotation_changed TO sphere.rotation
```

运行程序, 效果如图 5-20 所示。

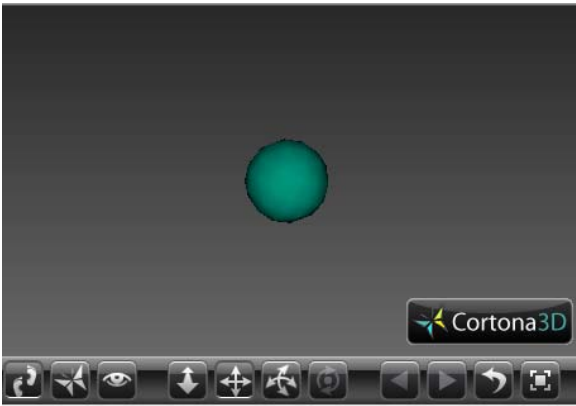


图 5-20 SphereSensor 节点应用效果

5.3.6 接近传感器

接近传感器感知观察者进入并在一个空间的长方体区域中移动的时间。当观察者接近区域时，能使用这些传感器启动一个动画，当观察者离开时，停止这个动画。**ProximitySensor** 节点能够作为任何组的子节点，并且它可以感知观察者何时进入、退出和移动于当前坐标系内一个长方体区域。可以检测到观察点接近的信号，利用它可控制其他操作，如发出声音，让开体运动、让形体颜色变化等。

可以通过给出中心和尺寸指定一个由 **ProximitySensor** 节点感知的空间区域。每一次观察者进入这个区域，传感器就使用 **enterTimer** 事件输出当前的绝对时间。每一次观察者离开这个区域，传感器使用 **exitTime** 事件输出时间。每一次观察者在感知区域改变位置和方向后，**ProximitySensor** 节点使用 **Position_changed** 和 **Orientation** 域（eventOut 出事件）来输出新的位置和方向。可以使用这些输出来跟踪观察者在一个感兴趣的区域内的移动。

ProximitySensor 节点语法如下：

节点名称	域名称	域值	#域及域值类型
ProximitySensor{	center	0 0 0	#exposedField SFVec3f
	size	0 0 0	#exposedField SFVec3f
	enabled	TRUE	#exposedField SFBool
	isActive		#eventOut SFBool
	trackPoint_changed		#eventOut SFVec3f
	rotation_changed		#eventOut SFVec3f
	}		

其中：

- （1）size 域值为 0.0 0.0 0.0 的接近传感器，表示它已被关闭了。
- （2）center 域表示区域的中心，在该区域内传感器检测用户动作。size 域以 center 域为中心，检测区沿每个坐标轴方向的坐标值。
- （3）enabled 域表示传感器是否正在检测用户的动作。可向传感器发送一个值为 FALSE 的 set_enabled 事件来关闭它。
- （4）isActive 是 eventOut 出事件，值类型为 SFBool，表示用户是否已进入（TRUE）或已离开（FALSE）此区域。

(5) `trackPoint_changed` 是 `eventOut` 出事件, 值是 `SFVec3f` 类型, 用户在拖动过程中任意时刻鼠标在假想的球体表面上的实际位置。

(6) `rotation_changed` 是 `eventOut` 出事件, 值是 `SFRoation` 类型, 用户在拖动过程中的任意时刻假想球体的当前朝向, 即旋转角度。

当观察者进入感知区域时, 传感器节点通过 `isActive` 域(`eventOut` 出事件)输出一个 `TRUE`, 并使用 `enterTime` 域(`eventOut` 出事件)输出当前的绝对时间。当观察者退出感知区域时, 由 `isActive` 域输出 `FALSE` 值, 使用 `exitTime` 域(`eventOut` 出事件)输出当前的绝对时间。

在感知区域中, 每次观察者的位置和方向发生改变时, 观察者新的位置和方向分别由 `position_changed` 和 `orientation_changed` 域(`eventOut` 出事件)来输出。两个值在 `ProximitySensor` 坐标系中测量。

传感器属性可通过向公共域的隐含 `set_enabled`、`set_center` 和 `set_size` 域(`eventIn` 入事件)传送值来改变。当这些值被接收时。相应的域值被改变。并且新的值通过公共域的隐含 `enabled_changed`、`center_changed` 和 `size_changed` 域(`eventOut` 出事件)来输出。

如果一个传感器使用 `DEF` 和 `USE` 来引用, 可视性区域就会在每一个传感器实例中被观测。多个传感器的区域可以重叠, 每个传感器产生的输出事件都是独立于其他传感器的。一个单独的传感器可被用在多个位置来检测用户的移动。当用户接近其中任何一个区域时都产生事件。包围整个世界的 `ProximitySensor` 所设置的 `enterTime` 值是指用户进入该世界的时候, 一旦世界被载入, 传感器立即激活动画。一个 `size` 域为 `(0 0 0)` 的传感器不产生任何事件, 相当于将 `enabled` 域的值设为 `FALSE`。

【例 5-16】 利用 `ProximitySensor` 节点, 创建一个常见于宾馆、商店的自动开关感应门的场景。感应门的造型分为三部分: 金属门框、左右两扇固定的玻璃门及中间两扇活动玻璃门。当用户使用 `Cortona` 浏览器控制面板提供 `plan` 按钮, 按下鼠标左键拖动与感应门造型逐渐接近时, 一旦进入设定的感知区域, 活动门便自动打开; 当用户反向拖动鼠标左键逐渐远离感应门造型的时候, 一旦退出设定的感知区域, 活动门便自动关闭。

其实现的源程序代码如下:

```
#VRML V2.0 utf8

Background {
    skyColor 0.2 0.5 0.6
}

Viewpoint {
    position 0 0 14
}

Transform {                                #创建门框造型
    translation 0 3.2 0
    rotation 0 0 1 1.571
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.4 0.2 0
                    ambientIntensity 0.28
```

```

        shininess 0.2
        specularColor 0.7 0.6 0.1
    }
}
geometry Cylinder {
    radius 0.4
    height 11.6
}
]
}
Transform {
    translation 0 -3.1 0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.4 0.2 0
                    ambientIntensity 0.28
                    shininess 0.2
                    specularColor 0.7 0.6 0.1
                }
            }
            geometry Box {
                size 11.6 0.2 0.8
            }
        }
    ]
}
Transform {
    translation 5.4 0 0
    children [
        DEF right Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.4 0.2 0
                    ambientIntensity 0.28
                    shininess 0.2
                    specularColor 0.7 0.6 0.1
                }
            }
            geometry Cylinder {
                radius 0.4
                height 6
            }
        }
    ]
}

```

```

    ]
}
Transform {
    translation -5.4 0 0
    children [
        USE right
    ]
}
DEF mdoor1 Transform {                                #右侧玻璃活动门坐标变换节点
    translation 1.3 0 -0.05
    children [
        DEF door1 Shape {                                #一扇玻璃活动门造型
            appearance Appearance {
                material Material {
                    diffuseColor 0.05 0.46 0.73
                    shininess 0.30
                    specularColor 1 1 0
                    emissiveColor 0.03 0.038 0.2
                    transparency 0.45
                }
            }
            geometry Box {
                size 2.55 6 0.1
            }
        }
    ]
}
DEF mdoor2 Transform {                                #左侧玻璃活动门坐标变换节点
    translation -1.3 0 -0.05
    children [
        USE door1
    ]
}
Transform {                                            #固定玻璃门造型
    translation 3.8 0 0.05
    children [
        DEF door2 Shape {                                #一扇固定玻璃门造型
            appearance Appearance {
                material Material {
                    diffuseColor 0.05 0.46 0.73
                    shininess 0.30
                    specularColor 1 1 0
                    emissiveColor 0.03 0.038 0.2
                    transparency 0.45
                }
            }
        }
    ]
}

```

```

        geometry Box {
            size 2.6 6 0.1
        }
    }
}
Transform {
    translation -3.8 0 0.05
    children [
        USE door2
    ]
}
DEF clock1 TimeSensor {                                #控制开门动作的时间
    cycleInterval 3
}
DEF clock2 TimeSensor {                                #控制关门动作的时间
    cycleInterval 3
}
DEF sensor ProximitySensor {                            #接近传感器设定感知区域
    size 10 6 24
}
DEF path1 PositionInterpolator {                         #控制开门动作的位置
    key [
        0 1
    ]
    keyValue [
        1.3 0 -0.05
        3.8 0 -0.05
    ]
}
DEF path2 PositionInterpolator {
    key [
        0 1
    ]
    keyValue [
        -1.3 0 -0.05
        -3.8 0 -0.05
    ]
}
DEF path3 PositionInterpolator {                         #控制关门动作的位置
    key [
        0 1
    ]
    keyValue [
        3.8 0 -0.05
        1.3 0 -0.05
    ]
}

```

```

    ]
}
DEF path4 PositionInterpolator {
    key [
        0 1
    ]
    keyValue [
        -3.8 0 -0.05
        -1.3 0 -0.05
    ]
}
ROUTE sensor.enterTime TO clock1.startTime
ROUTE clock1.fraction_changed TO path1.set_fraction
ROUTE path1.value_changed TO mdoor1.translation
ROUTE clock1.fraction_changed TO path2.set_fraction
ROUTE path2.value_changed TO mdoor2.translation
ROUTE sensor.exitTime TO clock2.startTime
ROUTE clock2.fraction_changed TO path3.set_fraction
ROUTE path3.value_changed TO mdoor1.translation
ROUTE clock2.fraction_changed TO path4.set_fraction
ROUTE path4.value_changed TO mdoor2.translation

```

运行程序，效果如图 5-21 及图 5-22 所示。

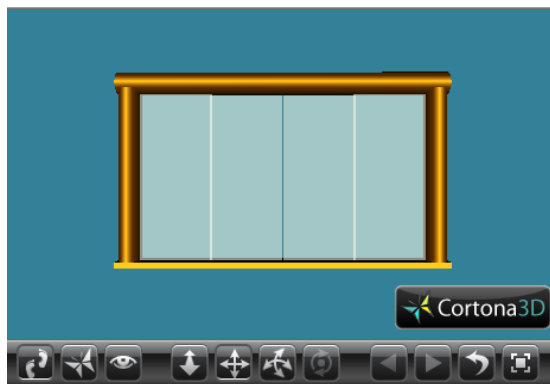


图 5-21 感应门造型效果

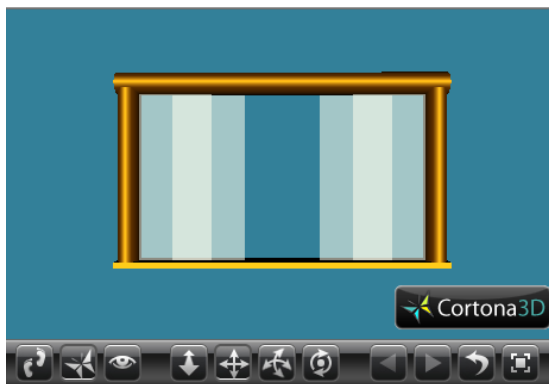


图 5-22 浏览者接近自动感应门开门效果

说明：

(1) 首先利用接近传感器，在感应门造型的周围设置一个尺寸为 $10 \times 6 \times 24$ 的长方体感知区域。

(2) 由 clock1 时间传感器和 path1、path2 两个位置插补器，控制 mdoor1、mdoor2 节点创建的两扇玻璃活动门的自动开门动作；由 clock2 时间传感器和 path3、path4 两个位置插补器，控制两扇玻璃活动门的自动关门动作。

(3) 路由语句 ROUTE sensor.enterTime TO clock1.startTime，一旦浏览者进入感知区域，接近传感器输出 enterTime 事件至时间传感器 clock1 的 startTime 域，启动 clock1 开始工作。路由语句 ROUTE clock1.fraction_changed TO path1.set_fraction 和 ROUTE path1.value_changed TO

`mdoor1.translation`，时间传感器 `clock1` 输出事件至位置插补器 `path1`，位置插补器 `path1` 控制右侧的活动门产生动画。同理，可以分析左侧活动门产生开门动画的过程以及关门动画的路由语句。

5.3.7 可视传感器

可视传感器从观察者的位置和方向来感知在空间中的一个长方体区域是否可视。可以使用这些传感器来启动和停止动画或者控制其他的动作。`VisibilitySensor` 节点能作为某个组的子节点，并且能从观察者的方向和位置感知一个长方体区域在当前的坐标系统中何时是可视的。

可以通过给出中心和尺寸指定一个由 `VisibilitySensor` 节点感知的空间区域。每一次任何区域部分进入立方体，传感器就使用 `enterTimer` 域（`eventOut` 出事件）来输出当前的绝对时间。同样地，每当区域离开立方体时，传感器使用 `exitTime` 域（`eventOut` 出事件）输出时间。使用这两个 `eventOut`，当一个区域变得可视时，可以绑定到一个路线来开始一个动作，或者当一个区域变为不可视时，就停止它。

`VisibilitySensor` 节点的语法如下：

节点名称	域名称	域值	#域及域值类型
<code>VisibilitySensor{</code>	<code>center</code>	<code>0 0 0</code>	<code>#exposedField SFVec3f</code>
	<code>size</code>	<code>0 0 0</code>	<code>#exposedField SFVec3f</code>
	<code>enabled</code>	<code>TRUE</code>	<code>#exposedField SFBool</code>
	<code>isActive</code>		<code>#eventOut SFBool</code>
	<code>enterTime</code>		<code>#eventOut SFTIME</code>
	<code>exitTime</code>		<code>#eventOut SFTIME</code>
<code>}</code>			

其中：

(1) `enabled` 为公共域的值指定传感器的开或关。如果域值为 `TRUE`，传感器处于打开状态，并产生输出。如果域值为 `FALSE`，传感器是关闭的，没有输出产生，除非是响应公共域值（如 `enabled_changed`）的改变。这个域的默认值为 `TRUE`。

(2) `center` 为公共域的值指定在当前三维坐标系中一个可感知区域中心的三维坐标。其默认区域中心为原点。

(3) `size` 为公共域的值指定一个传感器长方体区域的尺寸。包括宽度、高度和深度，在当前的坐标系中沿 `X`、`Y`、`Z` 方向测量。默认值为区域在原点的一个点。

(4) 当传感器任何部分在立方体内部时，传感器节点会使用 `isActive` 域（`eventOut` 出事件）来输出 `TRUE`，并使用 `enterTime` 域来输出当前的绝对时间。当全部传感器都在立方体外部时。传感器就会使用 `isActive` 域输出 `FALSE`，并且通过 `exitTime` 域输出当前的绝对时间。

传感器属性可被改变，这个改变是通过向可见域的隐含 `set_enabled`、`set_center` 和 `set_size` 域（`eventIn` 入事件）传送值来实现的。当这些值被接收时。相应域值被改变。并且新的值通过公共域的隐含 `enabled_changed`、`center_changed` 和 `size_changed` 域（`eventOut` 出事件）来输出。

(5) `isActive` 是 `eventOut` 出事件，值类型是 `SFBool`、`TRUE`，表明用户已进入包围盒；`FALSE` 表明用户不再包围盒内。当用户浏览时，`VisibilitySensor` 检查用户是否进入包围盒。当用户进入包围盒的任意一部分时，它输出一个值为 `TRUE` 的 `isActive` 事件；当用户退出包围盒时，他输出一个值为 `FALSE` 的 `isActive` 事件。

如果一个传感器使用 DEF 和 USE 来引用。可视性区域就会在每一个传感器实例中被观测。

【例 5-17】 利用 VisibilitySensor 节点，创建一个用紫红色小球是否可见控制白色正方体造型旋转的场景。紫红色小球的位置与 VisibilitySensor 节点设定的感知区域基本相同，小球可作为感知区域的标志。当用户使用 Cortona 浏览器控制面板提供的 plan 或者 plan 按钮，拖动鼠标改变浏览者观察点的位置时，一旦能够看到小球，正方体开始循环旋转动画；当再次改变浏览者观察点的位置，看不到小球的时候，正方体旋转一个周期后自动停止运行。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyColor 0.2 0.3 0.4
}
DEF rot Transform {                                #定义白色正方体坐标变换节点
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 1 1 1
                }
            }
            geometry Box {
                size 2.5 2.5 2.5
            }
        }
    ]
}

Transform {                                        #紫红色小球造型
    translation 0 5 0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 1 0 1
                }
            }
            geometry Sphere {
                radius 0.3
            }
        }
    ]
}

DEF clock TimeSensor {                            #定义时间传感器
    cycleInterval 5                                #设定循环周期
}
```

```

DEF path OrientationInterpolator {           #朝向插补器控制正方体旋转动画
    key [
        0 0.125 0.25
        0.375 0.5 0.625
        0.75 0.875 1.0
    ]
    keyValue [
        1 1 0 0
        1 1 0 0.785
        1 1 0 1.571
        1 1 0 2.356
        1 1 0 3.141
        1 1 0 3.926
        1 1 0 4.711
        1 1 0 5.496
        1 1 0 6.281
    ]
}
DEF sesor VisibilitySensor {
    center 0 5 0                             #定义可视化传感器
    size 0.6 0.6 0.6                         #设定感知区域中心
}
ROUTE sesor.isActive TO clock.loop
ROUTE clock.fraction_changed TO path.set_fraction
ROUTE path.value_changed TO rot.rotation

```

运行程序，效果如图 5-23 及图 5-24 所示。

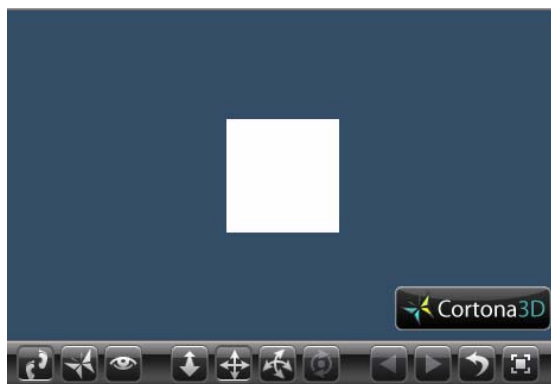


图 5-23 初始场景

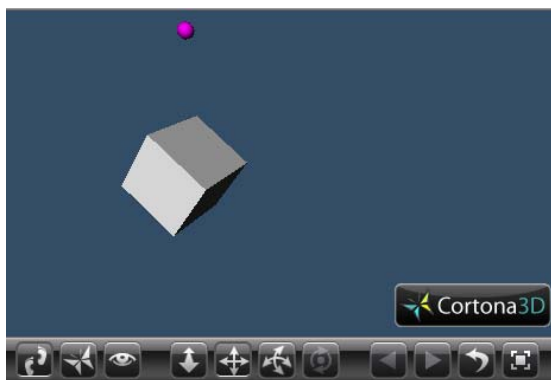


图 5-24 感知区域可视后的动画效果

说明：

(1) 利用 VisibilitySensor 节点设置一个感知区域。

(2) 路由语句 ROUTE sesor.isActive TO clock.loop，一旦浏览器看到小球（感知区域），可视传感器便输出 isActive 为 TRUE 的事件到时间传感器 clock 的 loop 域，启动 clock 开始循环工作。路由语句 ROUTE clock.fraction_changed TO path.set_fraction 和 ROUTE path.value_changed TO

rot.rotation，时间传感器 clock 输出事件至朝向插补器 path，朝向插补器 path 控制 rot 白色正方体造型产生旋转动画。同理，一旦浏览者看不到小球（感知区域），可视传感器便输出 isActive 为 FALSE 的事件至时间传感器 clock 的 loop 域，使时间传感器 clock 停止循环工作。朝向插补器 path 控制 rot 白色正方体造型继续完成一个周期的旋转动画后自动停止运行。

5.3.8 碰撞传感器

碰撞节点检测空间中观察者与造型接近和碰撞的时间，其发生在观察者碰撞到造型时。Collision 节点在检测观察者的碰撞时做两件事：通过 CollideTime 域（eventOut 出事件）输出当前的绝对时间；提示浏览者。

Collision 节点是组节点，类似于 Group 节点，碰撞节点有一个在它的 children 域中的子节点，目的是当观察者碰到一个特定造型时（如墙）能感知。也可用节点的 collide 域来关闭组中所有造型的碰撞检测，这样不会改变子节点的外形，但观察者可自由地穿越它们，这就可以创建观察者可穿越的造型。

检测观察者是否已经与一个造型发生碰撞，对于 VRML 浏览器来说将是非常费时的的工作。提供一个代理造型在 Collision 节点的 proxy 域中能帮助加速浏览器对碰撞的检测。代理造型是一个由 Collision 节点的子节点所创建的一种简单代替复杂的造型。例如，如果组的子节点描述一架钢琴和它的键。那么一个代理造型就可以用一个简单的长方体来模拟钢琴的形体和尺寸。当 VRML 浏览器检测到观察者与代理物碰撞时，长方体就代替了钢琴来检测碰撞。

当 Collision 节点提示浏览器一个碰撞发生时，浏览器检测当前往 Navigation 节点指定的导航类型。如果当前的导航类型是“EXAMINE”或“NONE”，那么观察者在与造型发生碰撞时。浏览器不作任何操作。如果当前导航类型是“WALK”和“FLY”，通常情况下，浏览器就停止观察者的动作。这样就可以防止观察者走进或飞入一个空间中的实心造型，以增加真实感。

Collision 节点语法如下：

节点名称	域名称	域值	#域及域值类型
Collision{			
	addChildren		#eventIn SFNode
	removeChildren		#eventIn SFNode
	children	[]	#exposedField SFBool
	collide	TRUE	#exposedField SFBool
	proxy	NULL	#SFVec3f
	bboxCenter	0 0 0	#SFVec3f
	bboxSize	-1 -1 -1	#SFVec3f
	collideTime		#eventOut SFTIME
	}		

其中：

（1）children 为公共域的值指定了一个包含在组中的子节点列表。典型的 children 域值包括 Shape 节点和其他组节点。VRML 浏览器由创建每一个造型和组，从而来创建该组。这个域的缺默认是空的子节点列表。

（2）bboxSize 域的值指定了一个约束长方体的尺寸，这个尺寸的大小要足以包容组中的所有造型。第一个给出的值是长方体 X 方向的宽度；第二个给出的值是长方体 Y 方向的高度；第三个给出值是长方体 Z 方向的深度。这个域的默认值是一个具有-1.0 单位宽度、高度和深度

的长方体，表示由 VRML 浏览器自动计算的约束长方体的尺寸。

(3) `bboxCenter` 域的值指定了约束长方体的中心，这个域值是组坐标系中的一个三维坐标。其默认值是原点。如果 VRML 浏览器自动计算约束长方体的尺寸，那么它也将自动计算约束长方体的中心。

(4) `collide` 为公共域值指定一个 TRUE 或一个 FALSE 值，它确定对于组中子节点的碰撞检测变为有效或无效。如果域值为 TRUE，当观察者使用由 `NavigationInfo` 节点指定的“WALK”或“FLY”导航类型移动时，组中的造型将进行碰撞检测。如果观察者碰撞到了造型，浏览器就被通知，并且观察者的移动停止。当域值 `collide` 是 FALSE 时，在组中的碰撞检测就为无效，观察者可以穿过组中的造型。域值 `collide` 的默认值为 TRUE。

`Collision` 节点的子节点可以是 `Collision` 节点。这样就创建了一个碰撞控制的家族树。在碰撞检测中，VRML 浏览器检测碰撞家族树，它从最高的父节点开始，直到每一个子节点。如果 `Collision` 节点父节点允许对子节点进行碰撞检测，那么 VRML 浏览器就可以检测父节点每一子节点的碰撞情况，并且检测子节点的子节点，如此一直到树端。如果 VRML 浏览器遇到一个不允许对其子节点进行碰撞检测的 `Collision` 父节点，那么 VRML 浏览器就跳过对该 `Collision` 节点的子节点及其后代的碰撞检测。

默认的情况下，认为场景中的所有物体都有可能被碰撞。如果在一直场景中没有指定 `Collision` 节点，在进行浏览时浏览器将检测用户是否要与某个物体发生碰撞。VRML 浏览器创建一个隐含的 `Collision` 节点，它围绕于整个空间，并且 `collide` 域值为 FALSE，这就确保了在使用“WALK”和“FLY”导航类型时，碰撞检测特性被打开。

一个造型和观察者之间的距离小于或等于由 `NavigationInfo` 节点的 `avatarSize` 域值指定的 `avatar` 碰撞宽度时，碰撞就发生。当碰撞在一个造型上发生时，包含该造型的每个 `Collision` 节点使用 `collide eventOut` 出事件输出当前绝对时间。

(5) 一个单个的造型也许是多个 `Collision` 节点的后代，例如，当一个 `Collision` 节点包括一个 `Collision` 节点作为其子节点，而这一子节点又包含另一个 `Collision` 子节点，如此一直延伸到一个造型。如果所有的 `Collision` 节点一直到这个造型都允许对子节点进行碰撞检测，当观察者碰撞这个造型时，那么所有的 `Collision` 节点都由它们的 `collideTime` 域（`eventOut` 出事件）输出碰撞时间。

(6) `proxy` 域值指出一个可以选择的造型，它用来作子节点造型的简单替代。由于对任意复杂的几何形体进行碰撞检测的计算量相当大，一个提高运行性能的方法是另外定义一个碰撞代理体（`proxy`）完成碰撞检测。这个 `proxy` 域中定义的代理体可以是任意合法的 VRML 组或子节点。在碰撞检测时，仅用代理体参与检测，所有 `Collision` 节点的子节点完全的被忽略掉。一个代理体可以简单到是一个包围盒或包围球。代理造型仅用于碰撞检测时代替子节点造型。子节点造型被实际创建，代理造型并不创建。`proxy` 域值的默认值是一个 NULL 造型。如果 `children` 域为空，`collide` 域为 TRUE，且已指定一个代理体，那么尽管什么也没有显示，碰撞检测仍将针对代理体进行。这是一种针对不可见几何形体进行的碰撞检测。

除了 `PointSet` 和 `IndexedLineSet` 节点外，所有几何节点都可以用来创建可碰撞的造型。

(7) `collide` 域值可被改变。这个改变是通过向公共域的隐含 `set_collide` 域（`eventIn` 入事件）传送值来实现的。当这些值被接收时，相应域值被改变，并且新的值通过公共域的隐含 `collide_changed` 域（`eventOut` 入事件）来输出。

`proxy` 和 `children` 可以包含嵌套的 `Collision` 节点，如果一个子 `Collision` 节点检测到一次碰

撞并且发生一个 `collideTime` 输出事件，它的所有的父 `Collision` 节点也会发生一个与它的 `collideTime` 值相同的输出事件。

(8)可以通过使用公共域 `children` 公共域的隐含 `set_children`、`addChildren` 和 `removeChildren` 域 `eventIn` 入事件，来设置、添加和删除组中子节点的列表。当一个节点值的列表传送到 `set_children` 或 `eventIn` 入事件时，`Children` 域的子节点列表值由输入的节点列表代替。当列表节点值被送到 `addChildren` 域 `eventIn` 入事件时，新子节点列表值被加入节点列表中，如果这个节点不在列表中。最后当列表节点值被送入 `removeChildren` 域 `eventIn` 入事件时，被选中的子节点列表被从节点列表中删除，如果这个节点在列表中。在所有 3 种情况下，如果子节点列表被更改，新的子节点列表可以用公共域 `children` 的隐含 `children_changed` 域 `eventOut` 出事件传出去。

【例 5-18】 感受碰撞检测器作用。

其实现的源程序代码如下：

```
#VRML V2.0 utf8

Background {
    skyColor 0.6 0.2 1
}

DEF sensor Collision {                                #定义碰撞传感器节点
    children [
        DEF ball Transform {                          #定义小球坐标变换节点
            children [
                Shape {
                    appearance Appearance {
                        material Material {
                            diffuseColor 1 1 0
                        }
                    }
                }
                geometry Sphere {
                    radius 1
                }
            ]
        }
    ]
}

DEF clock TimeSensor {                                #定义时间传感器节点
    cycleInterval 4
}

DEF path PositionInterpolator {                       #定义位置插补节点
    key [
        0 0.2 0.4 0.6 0.8 1.0
    ]
    keyValue [
        0 0 0
    ]
}
```

```
0 0 -30
10 0 -20
-10 0 20
0 0 -30
0 0 -10
]
}
ROUTE sensor.collideTime TO clock.startTime
ROUTE clock.fraction_changed TO path.set_fraction
ROUTE path.value_changed TO ball.translation
```

运行程序，效果如图 5-25 所示。

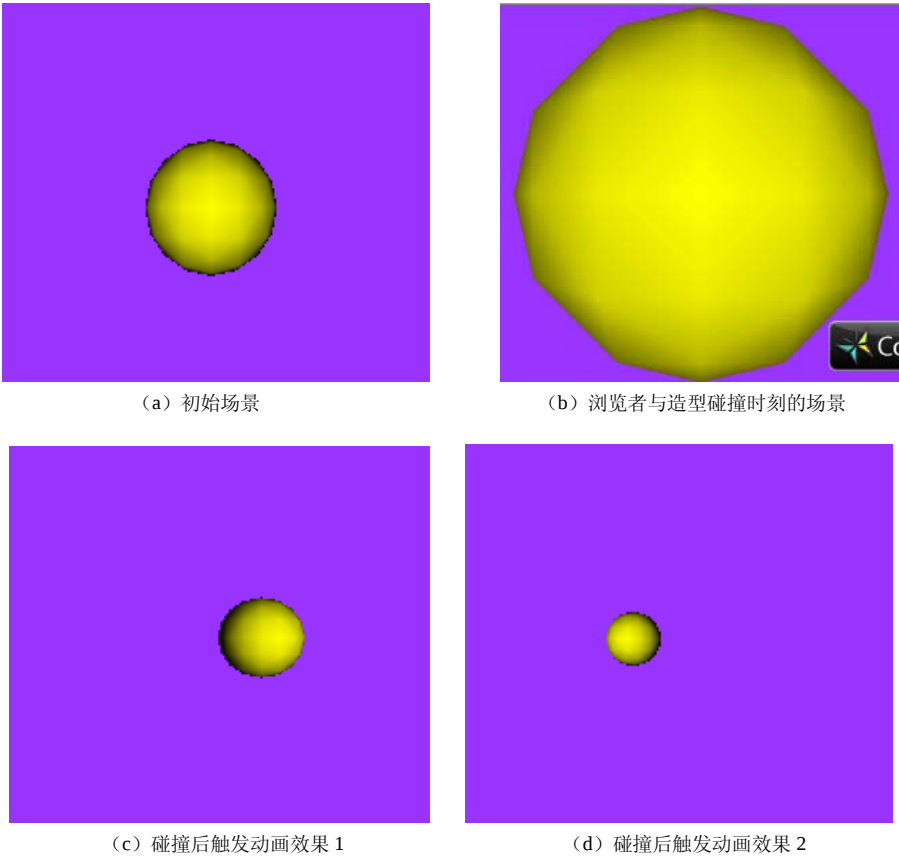


图 5-25 球体的碰撞效果

第 6 章 虚拟现实的高级应用

6.1 虚拟现实在三维立体场景中的设计

虚拟现实三维立体场景设计是以传统计算机系统和虚拟现实软件为平台来构建虚拟现实环境的。可以利用虚拟现实语言中的各种基本节点、群节点及场景效果节点、动态感知节点等进行三维立体造型和场景设计，可以利用软件项目开发方法和技术来提高虚拟现实软件产品开发的质量、速度和效率，使用户亲身感受虚拟现实技术的魅力，在虚拟现实场景中直接感受与虚拟造型和景物的动态交互，产生身临其境的感受。在虚拟现实环境中，真实感受可达到视觉、听觉、触觉和智能感知的自然效果。

6.1.1 虚拟现实在生日场景中的设计

虚拟现实生日烛光场景设计利用软件项目开发中的渐进式现代软件开发模型进行开发、设计、编程。该设计利用虚拟现实程序设计语言 VRML 作为软件项目开发工具进行编码、测试和运动，并采用结构化、组件化、模块化和面向对象思想及渐进式现代软件模型进行开发设计，遵循需求分析、总体设计、详细设计、编码和测试循环的开发过程。

1. 虚拟现实生日场景软件设计

虚拟现实生日烛光场景软件设计是利用软件工程的思想对虚拟现实生日烛光三维立体场景进行开发与设计。虚拟现实生日烛光三维立体场景由“祝您生日快乐!!!”三维立体文字、彩色蜡烛、烛光及生日蛋糕底盘组成。虚拟现实生日烛光场景设计的层次结构如图 6-1 所示。

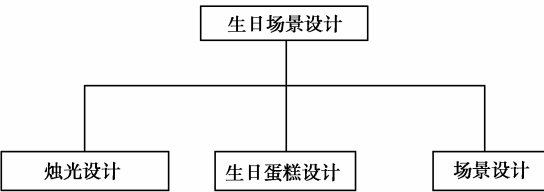


图 6-1 虚拟现实生日场景设计的层次结构图

2. 虚拟现实生日场景中的示例

虚拟现实生日烛光场景设计利用虚拟现实程序设计语言 VRML，对虚拟现实生日烛光三维立体场景中的生日烛光主程序、生日快乐子程序、蛋糕底盘子程序、红烛光子程序、黄烛光子程序、蓝烛光子程序、绿烛光子程序等进行编程、设计和调试。该设计采用模块化和组件化思想设计，在程序设计中使⤵用背景节点、组节点、模型节点、坐标变换节点、内联节点及面节点等虚拟现实节点进行编程、设计和调试。

【例 6-1】 虚拟现实生日三维立体场景中的设计。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
#创建多个坐标系
```

```
#角度      0      30      45      60      90      120      135      150      180
#弧度      0      0.524  0.785  1.047  1.571  2.094  2.356  2.618  3.141

Background{
    skyColor[
        0.98 0.98 0.98
    ]
}

#创建蜡烛烛火造型
Group {
    children [
        Transform {                                #利用嵌入节点调入烛火
            translation -2.0 0.6 0.0
            scale 0.2 0.2 0.2
            children Inline {url "redlight.WRL"}
        }
        Transform {                                #利用嵌入节点调入烛火
            translation -1.6 0.45 1.6
            scale 0.2 0.2 0.2
            children Inline {url "bluelight.WRL"}
        }
        Transform {                                #利用嵌入节点调入烛火
            translation 0 0.4 2.3
            scale 0.2 0.2 0.2
            children Inline {url "greenlight.WRL"}
        }
        Transform {                                #利用嵌入节点调入烛火
            translation 1.6 0.45 1.6
            scale 0.2 0.2 0.2
            children Inline {url "yellowlight.WRL"}
        }
        Transform {                                #利用嵌入节点调入烛火
            translation 2.0 0.6 0.0
            scale 0.2 0.2 0.2
            children Inline {url "redlight.WRL"}
        }
        Transform {                                #利用嵌入节点调入烛火
            translation 1.3 0.8 -1.3
            scale 0.2 0.2 0.2
            children Inline {url "bluelight.WRL"}
        }
        Transform {                                #利用嵌入节点调入烛火
            translation 0 0.8 -1.8
            scale 0.2 0.2 0.2
            children Inline {url "greenlight.WRL"}
        }
    ]
}
```

```

Transform {                                     #利用嵌入节点调入烛火
    translation -1.3 0.8 -1.3
    scale 0.2 0.2 0.2
    children Inline {url "yellowlight.WRL"}
}
Transform {                                     #利用嵌入节点调入蛋糕底盘
    translation 0.0 -0.8 0
    scale 9.0 8.0 9.0
    children Inline {url "cakebottomdisk.wrl"}
}
Transform {                                     #利用嵌入节点调入“生日快乐”
    translation 0.0 3.0 0
    scale 0.3 0.3 0.3
    children Inline {url "生日快乐.wrl"}
}
}

```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击主程序，生日烛光效果如图 6-2 所示。



图 6-2 虚拟现实生日烛光场景的运行效果

6.1.2 虚拟现实在奥运五环场景中的设计

虚拟现实奥运五环场景设计按照软件开发思想，利用瀑布模型、原型法模型及现代渐进式模型进行开发、设计和编码，并利用虚拟现实程序设计语言对三维立体场景的独立设计来完成奥运五环（五色）、北京天安门场景及三维立体汉字造型的设计。通过对三维立体造型进行动态交互控制，可以使虚拟现实场景的开发设计具有更加逼真的三维立体效果。

1. 虚拟现实奥运五环场景软件设计

为迎接 2008 年北京奥运会的胜利召开，可以利用虚拟现实软件技术开发一个虚拟现实奥运五环三维立体场景。该场景由北京天安门、奥运五环及“迎接 2008 年奥运会”文字等组成。该设计利用虚拟现实语言中的各种基本节点和复杂节点等对三维立体场景进行设计。虚拟现实奥运五环场景设计的层次结构如图 6-3 所示。

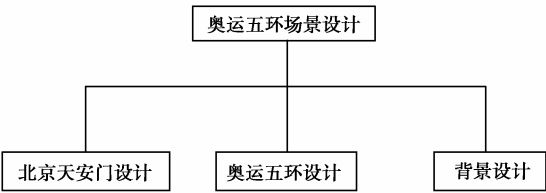


图 6-3 虚拟现实奥运五环场景设计的层次结构图

2. 虚拟现实奥运五环场景设计实例

虚拟现实奥运五环场景设计就是利用虚拟现实程序设计语言对三维立体场景进行设计和编码。该设计利用背景节点、内联节点、组节点及面节点进行设计开发，并利用内联节点实现子程序调用，实现模块化和组件化设计。

【例 6-2】 虚拟现实奥运五环三维立体场景设计。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8

#创建多个坐标系
#角度      0      30      45      60      90      120      135      150      180
#弧度      0      0.524  0.785  1.047  1.571  2.094  2.356  2.618  3.141
Background{
    skyColor[
        0.98 0.98 0.98
    ]
}

#创建五环造型场景
Group {
    children [
        Transform {                                     #利用嵌入节点调入 blueround
            translation -3.0 2.0 0
            scale 4.0 4.0 4.0
            children Inline {url "blueround.wrl"}
        }
        Transform {                                     #利用嵌入节点调入 blackround
            translation 0.5 2.0 0
            scale 4.0 4.0 4.0
            children Inline {url "blackround.wrl"}
        }
        Transform {                                     #利用嵌入节点调入 redround
            translation 4 2.0 0
            scale 4.0 4.0 4.0
            children Inline {url "redround.wrl"}
        }
        Transform {                                     #利用嵌入节点调入 yellowround
            translation -1.2 0.2 0
            scale 4.0 4.0 4.0
```



```

    }
}
geometry IndexedFaceSet {                                #面节点
    coord Coordinate {
        point [
            6.0 4.0 0.0,
            6.0 -4.0 0.0,
            -6.0 -4.0 0.0,
            -6.0 4.0 0.0
        ]
    }
    coordIndex[
        0,1,2,3
    ]
    texCoord TextureCoordinate {
        point [

            1.0 1.0,
            1.0 0.0,
            0.0 0.0,
            0.0 1.0,

        ]
    }
    texCoordIndex [
        0,1,2,3
    ]
    solid FALSE
}
}

```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。

双击“VRMLmain.wrl”的程序代码，便可运行虚拟现实奥运五环三维立体场景设计程序，其效果如图 6-4 所示。



图 6-4 虚拟现实奥运五环场景的运行效果

6.1.3 虚拟现实月亮绕地球转动场景设计

虚拟现实月亮绕地球转动场景设计是利用原型法模型及现代渐进式模型进行开发、设计和编码的。利用虚拟现实程序设计语言 VRML 可以对虚拟现实月亮绕地球转动三维立体场景进行设计和开发。在设计中，地球造型外部包含一个地球大气层，地球自转时大气层随之旋转，而且月亮绕地球旋转，从而使虚拟现实软件产品更加具有独特、逼真的三维立体效果。

1. 虚拟现实月亮绕地球转动场景软件设计

虚拟现实月亮绕地球转动场景软件设计是利用滚雪球式的软件开发模式对虚拟现实月亮绕地球转动的三维立体场景进行开发与设计。虚拟现实月亮绕地球转动场景由三维立体地球、三维立体月球和三维立体文字等构成。该设计利用层次化、结构化、模块化、组件化思想进行设计和开发。虚拟现实月亮绕地球转动场景设计的层次结构如图 6-5 所示。



图 6-5 虚拟现实月亮绕地球转动场景设计层次结构图

2. 虚拟现实月亮绕地球转动场景示例

虚拟现实月亮绕地球转动场景设计使用虚拟现实程序设计语言中的视角节点、背景节点、坐标变换节点、内联节点、球节点、柱节点、重定义节点、重用节点、组节点、时间传感器节点、朝向插补器节点等进行设计和开发，并利用内联节点实现了子程序调用，实现模块化和组件化设计。在虚拟现实程序设计中，为了提高软件运行效果，可以通过修改源程序中的背景节点的颜色域值，调整红、绿、蓝 3 种颜色的比例来改变背景颜色，建议采用天蓝色或者自己喜欢的背景颜色。

【例 6-3】 虚拟现实月亮绕地球转动场景设计。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
#月亮围绕地球转动
Group {
  children [
    Background {
      skyColor [
        0.2 0.3 0.6
        1.0 1.0 1.0
        0.2 0.6 0.2
      ]
    }
    skyAngle [1.571 2.012 2.857]
  ]
}
DEF waylight DirectionalLight {
  direction 1.0 0.0 0.0
  intensity 1.0
  ambientIntensity 1.0
  #方向光源
  #光线的朝向
  #光线强度
  #光线的影响度
```

```

        color 1.0 1.0 1.0                                #光的颜色
    }
    DEF fly Transform {                                    #引入地球造型
        translation 0 0 -20
        scale 1.2 1.2 1.2
        children Inline {url "earth.wrl"}
    }
    DEF Time TimeSensor {                                  #时间传感器
        cycleInterval 6.0
        loop TRUE
    }
    DEF flyinter OrientationInterpolator {                 #朝向移动位置节点
        key [                                              #相对时间的逻辑值
            0.0,0.2,0.3,0.4,0.5
            0.6,0.8,0.9,1.0
        ]
        keyValue [                                         #空间坐标旋转轴的位置值与相对时间的逻辑值
            0.0 1.0 0.0 0.0,
            0.0 1.0 0.0 0.524,
            0.0 1.0 0.0 1.571,
            0.0 1.0 0.0 1.982,
            0.0 1.0 0.0 2.618,
            0.0 1.0 0.0 3.141,
            0.0 1.0 0.0 4.782,
            0.0 1.0 0.0 5.681,
            0.0 1.0 0.0 6.280,
        ]
    }
    Transform {                                            #引入汉字造型
        translation 0 -6 -10
        scale 0.8 0.8 0.8
        children Inline {url "moonround.wrl"}
    }
    DEF fly1 Transform {                                    #引入月亮造型
        children Inline {url "moon.wrl"}
    }
    DEF Time1 TimeSensor {                                  #时间传感器
        cycleInterval 8.0
        loop TRUE
    }
    DEF flyinter1 PositionInterpolator {                   #移动位置节点
        key [                                              #相对时间的逻辑值
            0.0,0.2,
            0.4,0.6,

```

```

    ]
    keyValue [                                     #空间坐标的位置值与相对时间的逻辑值

        0 0 -35,
        -15 0 -20,
        0 0 0,
        15 0 -20,
        0 0 -35,
    ]
}
DEF fly2 Transform {                               #大气层
    translation 0 0 -20
    scale 1.2 1.2 1.2
    children [
Shape {                                             #银白颜色
appearance Appearance{
    material Material {                            #空间物体造型外观
        diffuseColor 0.5 0.5 0.7                  #一种材料的漫反射颜色
        ambientIntensity 0.4                      #多少环境光被该表面反射
        specularColor 0.8 0.8 0.9                 #物体镜面反射光线的颜色
        shininess 0.20                           #造型外观材料的亮度
        transparency 0.8
    }
}
geometry Sphere {                                 #球体
    radius 3.5
}
}
]
}
DEF Time2 TimeSensor {                             #时间传感器
    cycleInterval 6.0
    loop TRUE
}
DEF flyinter2 OrientationInterpolator {            #朝向移动位置节点
    key [                                           #相对时间的逻辑值
        0.0,0.2,0.3,0.4,0.5
        0.6,0.8,0.9,1.0
    ]
    keyValue [                                     #空间坐标旋转轴的位置值与相对时间的逻辑值
        0.0 1.0 0.0 0.0,
        0.0 1.0 0.0 0.524,
        0.0 1.0 0.0 1.571,
        0.0 1.0 0.0 1.982,
        0.0 1.0 0.0 2.618,
        0.0 1.0 0.0 3.141,
    ]
}

```

```

0.0 1.0 0.0 4.782,
0.0 1.0 0.0 5.681,
0.0 1.0 0.0 6.280,

    ]
    }
]
}

ROUTE Time.fraction_changed TO flyinter.set_fraction
ROUTE flyinter.value_changed TO fly.set_rotation
ROUTE Time1.fraction_changed TO flyinter1.set_fraction
ROUTE flyinter1.value_changed TO fly1.set_translation
ROUTE Time2.fraction_changed TO flyinter2.set_fraction
ROUTE flyinter2.value_changed TO fly2.set_rotation

```

在主程序中调用的虚拟现实月亮绕地球转动场景设计中“earth.wrl”子程序的源程序代码如下：

```

#VRML V2.0 utf8
#角度数 0 30 45 60 90 120 135 150 180
#弧度数 0 0.524 0.785 1.047 1.571 2.094 2.356 2.618 3.141
#创建造型
Background {
    skyColor [
        0.1 0.3 0.6
    ]
}
Transform{
    rotation 1.0 1.0 0.0 0.785
    children[
        Shape {
            appearance Appearance{
                texture ImageTexture {
                    url "earth.gif"
                }
            }
            geometry Sphere {

                radius 1.8

            }
        }
    ]
}

```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。

双击主程序 VRMLmain.wrl，效果如图 6-6 所示。

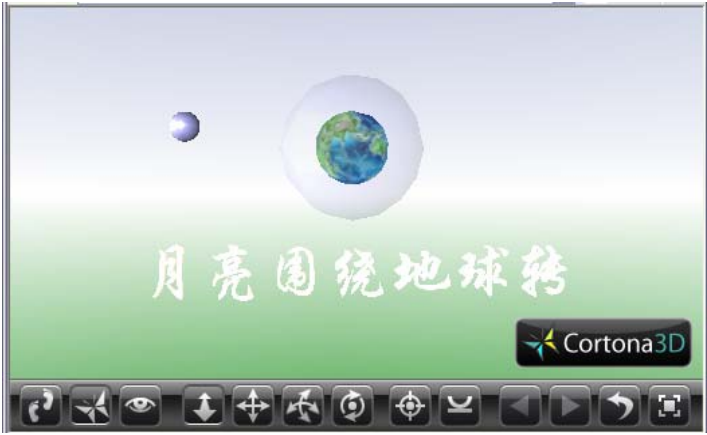


图 6-6 虚拟现实月亮绕地球转动场景的设计效果

6.2 虚拟实现自然景观设计

虚拟现实自然景观设计就是对自然界中的自然现象，如一年四季中春、夏、秋、冬的变化及雨、雪、阴、晴、冷、暖等进行虚拟现实仿真。

6.2.1 虚拟现实雪山设计

虚拟现实雪山场景设计是根据自然界的四季轮回，并利用虚拟现实程序设计语言开发设计逼真的雪山三维立体场景。虚拟现实雪山场景设计引入公路场景和彩灯场景，利用不断变化的彩色路灯渲染整体的动态智能感知动画场景效果，实现人与自然和谐共处的、生动的虚拟现实场景。

1. 虚拟现实雪山软件设计

虚拟现实雪山场景软件利用冬季雪山场景和公路造型，运用软件工程思想，并采用滚雪球式的软件开发模式对虚拟现实雪山三维立体场景进行需求分析、设计和编码。虚拟现实雪山场景由三维立体雪山、公路、彩色路灯和三维立体文字等组成，并采用模块化、组件化设置思想，使开发的软件产品层次清晰、结构合理。虚拟现实雪山设计的层次结构如图 6-7 所示。

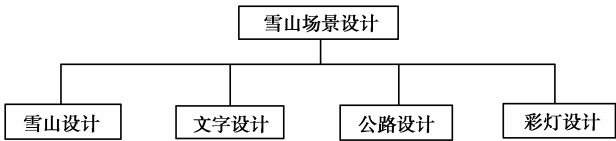


图 6-7 虚拟现实雪山设计层次结构图

2. 虚拟现实雪山设计示例

虚拟现实雪山场景设计使用虚拟现实程序设计语言中的视角节点、组节点、背景节点、坐标变换节点、内联节点、柱节点、立方体节点、球节点、重定义节点、重用节点、面节点、时间传感器节点、动态颜色插补器节点及事件和路由等进行设计和开发，并利用内联节点实现子程序调用，实现模块化和组件化设计。该设计利用动态智能感知节点设计自然景观中太阳东升

西落的动态场景。

【例 6-4】 虚拟现实雪山场景设计。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
#雪山公路和场景

DEF View1 Viewpoint {
  position 0 90 80
  orientation 1 0 0 0
  fieldOfView 0.6024
  description "View1"
}
Group {
  children [
    Background {
      skyColor 0.98 0.98 0.98
    }

    Transform {
      translation -5 -0.1 0
      scale 1.0 1.0 1.0
      children [
        Inline {url "caideng.wrl"}
      ]
    }
    Transform {
      translation 15 -0.1 0
      scale 1.0 1.0 1.0
      children [
        Inline {url "caideng.wrl"}
      ]
    }
    Transform {
      translation 20 1.0 0
      scale 1.5 1.5 1.5
      rotation 0 1 0 1.571
      children [
        Inline {url "xueshanjingguan.wrl"}
      ]
    }
  ]
}

Transform {                                     #公路地面
  translation 4.0 -3.5 0
  children[
```

```
DEF way1 Shape {
  appearance Appearance{
    material Material {
      diffuseColor 1.0 1.0 1.0
      ambientIntensity 0.2
      specularColor 1.0 1.0 1.0
      shininess 0.2
      transparency 0.0
    }
  }
  geometry Box {
    size 25 0.5 160
  }
}
]
}
Transform {
  translation 4.0 -3.2 0
  children[
    DEF way1 Shape {
      appearance Appearance{
        material Material {
          diffuseColor 0.8 0.2 0.8
          ambientIntensity 0.2
          specularColor 1.0 1.0 1.0
          shininess 0.2
          transparency 0.0
        }
      }
      geometry Box {
        size 0.6 0.5 160
      }
    }
  ]
}
Transform {
  translation 20.0 0.5 0.0
  scale 1.0 1.0 1.0
  rotation 0 1 0 1.571
  children Inline {url "wenzi.WRL"}
}
```

在主程序调用的虚拟现实雪山场景设计中“xueshanjingguan.wrl”子程序的源程序代码如下：

```
#VRML V2.0 utf8
#创建多个坐标系
```

```
#角度      0      30      45      60      90      120      135      150      180
#弧度      0      0.524  0.785  1.047  1.571  2.094  2.356  2.618  3.141

Background{
    skyColor[
        0.0 0.0 0.98
    ]
}

#创建雪山场景造型
Group {
    children [

        Transform {                                     #利用嵌入节点调入雪山模型
            translation -40.0 -3.0 -30.0
            scale 3.0 3.5 3.0
            children Inline {url "xueshan.WRL"}
        }
        Transform {                                     #利用嵌入节点调入雪山模型
            translation -20 -3.0 -34.0
            scale 4.0 5.0 4.0
            children Inline {url "xueshan.WRL"}
        }
        Transform {                                     #利用嵌入节点调入雪山模型
            translation 0.0 -3.0 -34.0
            scale 4.5 8.6 4.0
            children Inline {url "xueshan.WRL"}
        }
        Transform {                                     #利用嵌入节点调入雪山模型
            translation 18.0 -3.0 -30.0
            scale 3.0 5.8 3.0
            rotation 0 1 0 1.571
            children Inline {url "xueshan.WRL"}
        }
        Transform {                                     #利用嵌入节点调入雪山模型
            translation 40.0 -3.0 -32.0
            scale 3.5 3.5 3.5
            rotation 0 1 0 1.571
            children Inline {url "xueshan.WRL"}
        }
    ]
}
```

在主程序中调用的虚拟现实雪山场景设计中“caideng.wrl”子程序的源程序代码如下：

```
#VRML V2.0 utf8
#使装饰灯变色
Background {
```

```

        skyColor 0.2 0.8 1.0
    }
DEF waylamp Group {
    children[
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.5 0.5 0.7
                    ambientIntensity 0.4
                    specularColor 0.8 0.8 0.9
                    shininess 0.2
                }
            }
            geometry Cylinder {                #灯柱
                radius 0.3
                height 7.2
            }
        }
        Transform {
            translation 0.0 2.5 0.0
            children [
                Shape {
                    appearance Appearance {
                        material DEF lamp Material {
                            diffuseColor 1.0 1.0 1.0
                        }
                    }
                    geometry Sphere {                #灯头
                        radius 1.0
                    }
                }
            ]
        }
    ]
}

Transform {
    translation 0 0 70
    children USE waylamp
}

Transform {
    translation 0 0 60
    children USE waylamp
}

Transform {
    translation 0 0 50
    children USE waylamp
}

```

```
    }  
    Transform {  
        translation 0 0 40  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 30  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 20  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 10  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 0  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 -10  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 -20  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 -30  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 -40  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 -50  
        children USE waylamp  
    }  
    Transform {  
        translation 0 0 -60  
        children USE waylamp  
    }  
    Transform {
```

```
translation 0 0 -70
children USE waylamp
}
DEF Time TimeSensor      {                                #时间传感器
    cycleInterval 6.0
    loop TRUE
}
DEF lightcolor ColorInterpolator {                        #颜色插补器
    key [                                                  #相对时间间隔值
        0.1
        0.2
        0.5
        0.6
        0.8
        1.0
    ]
    keyValue [                                              #颜色变换值
        0.0 0.0 1.0
        0.0 1.0 0.0
        1.0 0.0 0.0
        1.0 1.0 0.0
        0.0 1.0 1.0
        1.0 0.0 1.0
    ]
}
ROUTE Time.fraction_changed TO lightcolor.set_fraction
ROUTE lightcolor.value_changed TO lamp.set_diffuseColor
```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击“VRMLamin.wrl”主程序，得到如图 6-8 所示的效果。

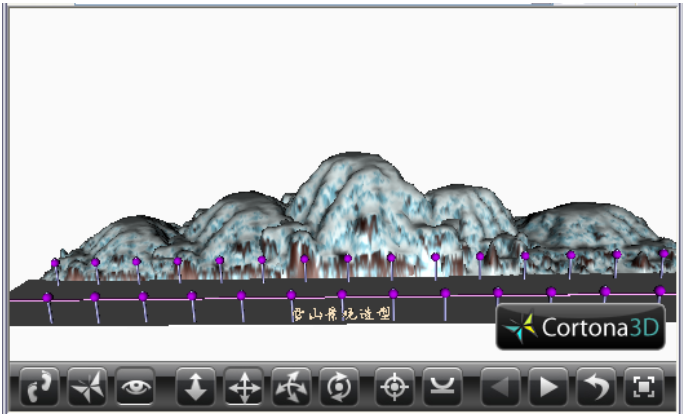


图 6-8 虚拟现实雪山设计效果

6.2.2 虚拟现实海上日出设计

虚拟现实海上日出场景设计是利用虚拟现实世界中的自然景观来展示自然界中的山脉、海

洋、河流等自然景观。利用虚拟现实程序设计语言、软件工程设计思想及传统软件开发模式和现代软件开发模式，可以创建逼真的虚拟现实海上日出三维立体场景。

1. 虚拟现实海上日出软件设计

虚拟现实海上日出场景软件设计是利用自然界山川、河流及海洋等自然景观开发设计虚拟现实场景。该设计运用软件工程思想，并采用滚雪球式的软件开发模式对虚拟现实海上日出三维立体动画场景进行需求分析、设计和编码。虚拟现实海上日出三维立体场景由山脉、三维立体旭日及动态三维立体海浪等组成，设计采用层次化、模块化、组件化设计思想，使开发的软件产品层次清晰、结构合理。虚拟现实海上日出场景设计的层次结构如图 6-9 所示。



图 6-9 虚拟现实海上日出场景设计的层次结构图

2. 虚拟现实海上日出设计示例

虚拟现实海上日出场景设计利用虚拟现实程序设计语言中的背景节点、组节点、坐标变换节点、内联节点、重定义节点、重用节点、面节点及三维动画节点等进行设计和开发，并利用内联节点实现子程序调用及模块化和组件化设计。

在软件设计中，为了输出较好的虚拟现实海上日出三维立体场景设计效果，可以修改源程序中的背景节点的颜色域值，调整红、绿、蓝 3 种颜色的比例。

【例 6-5】 虚拟现实海上日出设计示例。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
#创建多个坐标系
#角度      0      30      45      60      90      120      135      150      180
#弧度      0      0.524  0.785  1.047  1.571  2.094  2.356  2.618  3.141
Background{
    skyColor[
        0.3 0.4 0.5                                #打印背景颜色
    ]
}
#创建山脉河流景观造型
Group {
    children [
        Transform {                                #利用嵌入节点调入山模型
            translation -8.0 -2.6 0.0
            scale 1.0 1.5 1.0
            children Inline {url "shanmoxing.wrl"}
        }
        Transform {                                #利用嵌入节点调入山模型
            translation -8.5 -2.6 -10.0
            scale 1.0 1.0 1.0
```

```

        children Inline {url "shanmoxing.wrl"}
    }
    Transform {                                #利用嵌入节点调入山模型
        translation -8.0 -2.6 -20.0
        scale 1.0 1.3 1.0
        children Inline {url "shanmoxing.wrl"}
    }
    Transform {                                #利用嵌入节点调入河流模型
        translation -2.0 -4.0 0.0
        scale 1.0 1.0 1.0
        children Inline {url "heliu.wrl"}
    }
    Transform {                                #利用嵌入节点调入河流模型
        translation -2.0 -4.1 -25.0
        scale 1.0 1.0 1.0
        children Inline {url "heliu.wrl"}
    }
    Transform {                                #利用嵌入节点调入河流模型
        translation 25.0 -4.0 0.0
        scale 1.0 1.0 1.0
        children Inline {url "heliu.wrl"}
    }
    Transform {                                #利用嵌入节点调入河流模型
        translation 25.0 -4.1 -25.0
        scale 1.0 1.0 1.0
        children Inline {url "heliu.wrl"}
    }
    Transform {                                #利用嵌入节点调入山模型
        translation 10.0 -2.6 0.0
        scale 1.5 2.0 1.5
        children Inline {url "shanmoxing.wrl"}
    }
    Transform {                                #利用嵌入节点调入山模型
        translation 10.0 -2.6 -15.0
        scale 1.5 1.0 1.5
        children Inline {url "shanmoxing.wrl"}
    }
    Transform {                                #利用嵌入节点调入山模型
        translation 10.0 -2.6 -28.0
        scale 1.5 2.5 1.5
        children Inline {url "shanmoxing.wrl"}
    }
    Transform {                                #利用嵌入节点调入文字
        translation 0.0 -2 1.0
        scale 0.2 0.2 0.2
        children Inline {url "haishangrichu.WRL"}
    }

```

```

    }
  ]
}
#创建太阳造型
DEF sun Transform {
  translation 0 0 0
  scale 1.0 1.0 1.0
  children [
    Shape {
      appearance Appearance{
        material Material {
          diffuseColor 1.0 0.1 0.1      #空间物体造型外观
          ambientIntensity 0.1          #一种材料的漫反射颜色
          specularColor 0.7 0.7 0.6     #多少环境光被该表面反射
          shininess 0.4                 #物体镜面反射光线的颜色
          transparency 0.2             #造型外观材料的亮度
        }
      }
      geometry Sphere {
        radius 5.0                      #球体
      }
    ]
  ]
}
DEF Time TimeSensor {
  cycleInterval 16.0                  #时间传感器
  loop TRUE
}
DEF run PositionInterpolator {
  key [
    0.0,0.2,
    0.4,0.6,
    0.8,1.0
  ]
  keyValue [
    0 -2.0 -35,
    0 3.0 -35,
    0 5.0 -35,
    0 8 -35,
    0 10 -35,
    0 15 -35,
  ]
}
ROUTE Time.fraction_changed TO run.set_fraction
ROUTE run.value_changed TO sun.set_translation

```

在主程序中调用的虚拟现实雪山场景设计中“heliu.wrl”子程序的源程序代码如下：

```
#VRML V2.0 utf8
Background {
    skyColor [0.0 0.8 1.0]
}
DEF Nav0 NavigationInfo{
    type["EXAMINE","ANY"]
}
DEF View0 Viewpoint{
    position 0 5 10 orientation 1 0 0 -0.2
}
DEF Time0 TimeSensor{
    cycleInterval 3 loop TRUE
}
Transform{
    translation -15 0 -10
    children[
        Shape{
            geometry DEF Grid0 ElevationGrid{
                xDimension 15
                zDimension 15
                xSpacing 2
                zSpacing 2
                creaseAngle 0.01
                height[
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
                    0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

                ]
            }
        }
    ]
}
color DEF Color0 Color{
    color[
```

```

0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,

0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,

0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
0 1 1, 0 0 1, 0 0 1, 1 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1, 0 0 1,
    ]
    }
    }
    }
    ]
}

```

```

DEF Script0 Script{
    eventIn SFFloat   XXX
    eventIn SFTIME    YYY
    field   SFNode    grid0 USE Grid0
    field   SFNode    color0 USE Color0
    field   MFColor    randColors0[]
    field   MFColor    randColors1[]
    field   MFColor    randColors2[]
    field   MFFloat    randHeights0[]
    field   MFFloat    randHeights1[]
    field   MFFloat    randHeights2[]
    field SFCOLOR bb 1.0 1.0 1.0
    field SFCOLOR ll 0.0 0.0 1.0
    field SFCOLOR ss 0.0 0.0 0.0
    url "javascript:
    function initialize(){
        for(i=0;i<225;i++){
            for(j=0;j<15;j=j+1){
                randColors1[i][j] = Math.random();
            }
            randHeights1[i] = 2-Math.random()*0.8;
        }
    }
}

```

```

}
function YYY(v){
  for(i=0;i<225;i++){
    randColors0[i] = randColors1[i];
    randHeights0[i] = randHeights1[i];
  }
  initialize();
}
function XXX(v){
  for(i=0;i<225;i++){
    for(j=0;j<15;j++){
      randColors2[i][j] = randColors0[i][j]-(randColors0[i][j]-randColors1[i][j])*v;
    }
    randHeights2[i] = randHeights0[i]-(randHeights0[i]-randHeights1[i])*v;
  }
  color0.set_color = randColors2;
  grid0.set_height = randHeights2;
}
}
ROUTE Time0.fraction_changed TO Script0.XXX
ROUTE Time0.cycleTime TO Script0.YYY

```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击“VRMLamin.wrl”主程序，得到如图 6-10 所示的效果。

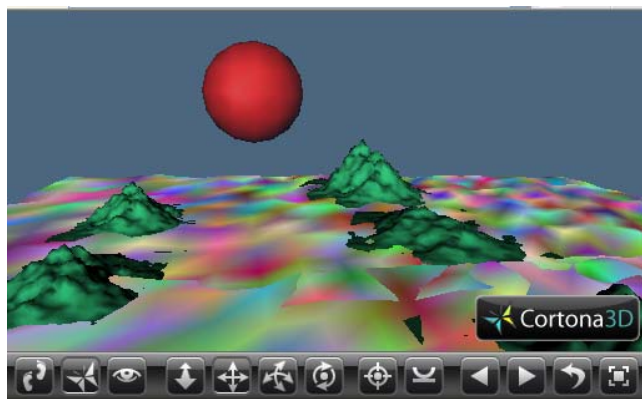


图 6-10 虚拟现实海上日出场景设计效果

6.2.3 虚拟现实雪山树林设计

虚拟现实雪山树林自然界中秋、冬季场景的一个掠影。利用虚拟现实程序设计语言及传统软件开发模式和现代软件开发模式，可以创建逼真的虚拟现实雪山树林三维立体场景。虚拟现实雪山树林三维立体场景设计引入了雪山场景、各种植物和树木造型，从而使虚拟现实雪山树林场景更加逼真。

1. 虚拟现实雪山树林软件设计

虚拟现实雪山树林场景软件设计是利用各种植物造型、雪山场景、运用软件工程思想，并

采用滚雪球式的软件开发模式对虚拟现实雪山树林三维立体场景进行需求分析、设计和编码。虚拟现实雪山树林三维立体场景由各种三维立体植物、雪山场景等组成，并采用模块化、组件化设计思想，使开发的软件产品层次清晰、结构合理。虚拟现实雪山树林场景设计的层次结构如图 6-11 所示。



图 6-11 虚拟现实雪山树林场景设计的层次结构图

2. 虚拟现实雪山树林示例

虚拟现实树林场景设计利用虚拟现实程序设计语言中的各种简单节点和复杂节点，如背景节点、组节点、坐标变换节点、内联节点、重定义节点、重用节点及面节点等进行设计和开发，并利用内联节点实现子程序调试及模块化和组件化设计。

【例 6-6】 虚拟现实雪山树林场景设计示例。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
#创建多个坐标系
#角度    0      30      45      60      90      120      135      150      180
#弧度    0      0.524  0.785    1.047    1.571    2.094    2.356    2.618    3.141

Background{
    skyColor[
        0.98 0.98 0.98
    ]
}

#创建树木场景造型
Group {
    children [
        DEF tree1  Transform {                                #利用嵌入节点调入树模型
            translation -10.0 8.0 -10.0
            scale 1.0 1.0 1.0
            children Inline {url "shu1.wrl"}
        }

        DEF tree2 Transform {                                #利用嵌入节点调入树模型
            translation 0.0 -5.0 -10.0
            scale 0.03 0.03 0.03
            children Inline {url "shu2.wrl"}
        }

        DEF tree3  Transform {                                #利用嵌入节点调入树模型
            translation 10.0 -8.0 -10.0
            scale 0.06 0.08 0.06
```

```
        rotation 0 1 0 1.571
        children Inline {url "shu3.wrl"}
    }
}
}
Transform {                                #利用嵌入节点调入树模型
    translation 10.0 -8.0 -250.0
    scale 30.0 30.0 30.0
    rotation 0 1 0 1.571
    children Inline {url "xueshan.wrl"}
}
Transform {                                #利用重用节点模型
    translation 0.0 0 -10.0
    children USE    tree1
}
Transform {                                #利用重用节点模型
    translation 0.0 0 -30.0
    children USE    tree1
}
Transform {                                #利用重用节点模型
    translation 0.0 0 -50.0
    children USE    tree1
}
Transform {                                #利用重用节点模型
    translation 0.0 0 -70.0
    children USE    tree1
}
Transform {                                #利用重用节点模型
    translation 0.0 0 -90.0
    children USE    tree1
}
Transform {                                #利用嵌入节点调入树模型
    translation 0.0 0.0 -10.0
    children USE    tree2
}
Transform {                                #利用嵌入节点调入树模型
    translation 0.0 0.0 -40.0
    children USE    tree2
}
Transform {                                #利用嵌入节点调入树模型
    translation 0.0 0.0 -50.0
    children USE    tree2
}
Transform {                                #利用嵌入节点调入树模型
    translation 0.0 0.0 -60.0
    children USE    tree2
}
```

```
    }
    Transform {
        translation 0.0 0.0 -70.0
        children USE tree2
    }
    Transform {
        translation 0.0 0.0 -20.0
        children USE tree3
    }
    Transform {
        translation 0.0 0.0 -40.0
        children USE tree3
    }
    Transform {
        translation 0.0 0.0 -60.0
        children USE tree3
    }
    Transform {
        translation 0.0 0.0 -80.0
        children USE tree3
    }
    Transform {
        translation 0.0 0.0 -100.0
        children USE tree3
    }
}
```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击主程序，效果如图 6-12 所示。

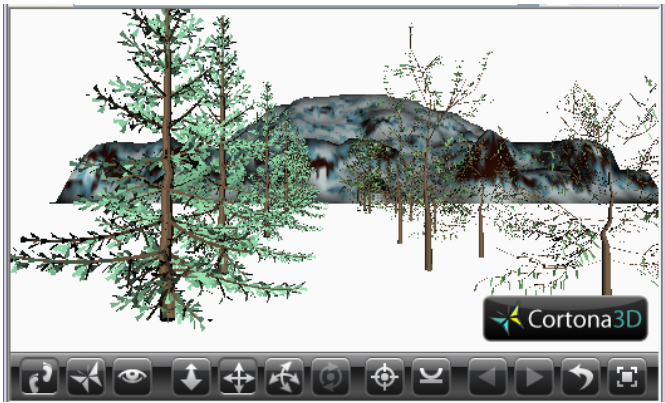


图 6-12 虚拟现实雪山树林场景设计效果

6.3 虚拟现实建筑设计

虚拟现实建筑场景可视化设计包括建筑物设计、室内装饰设计、城市景观设计、施工过程

设计、物理环境设计、历史性建筑模拟设计等。虚拟现实建筑场景设计是利用建筑景观设计来实现人和自然景观的完美结合，如商场、超市、法院、公司和住宅小区的设计，从而为人们学习、生活和工作提供方便、快捷、便利的服务。虚拟现实建筑场景设计针对建筑设计、城市规划及周边环境进行设计、编码和布局，可以创建出逼真的三维立体城市建筑场景，让人们真正感受虚拟现实三维立体建筑场景所蕴涵的无穷魅力。

在建筑设计中既要进行空间形象思维，又要考虑以用户的感受为核心，是一种创新过程，包括规划、设计、建设施工、维护等。建筑物巨大的成本和不可逆的执行程序要求不能出现设计差错。虚拟现实技术是一种可以创造和体现虚拟现实世界的计算机系统，而虚拟世界则是整体虚拟环境或给定仿真的对象的全体。充分利用计算机系统和虚拟现实技术，可减轻设计人员的劳动强度，缩短设计周期，提高设计质量，节省投资。因此，虚拟现实技术在建筑设计中得到了广泛的应用。

6.3.1 虚拟现实办公楼

虚拟现实办公楼建筑场景包括办公楼建筑、街道及绿化带等。在虚拟现实办公楼建筑场景中可以体验虚拟世界给人们带来的真实感受，并在虚拟世界中体验办公、生活和娱乐。虚拟现实办公楼建筑场景与现实世界中的场景相融合，可以创造出逼真的三维立体办公楼建筑场景。虚拟现实办公楼建筑场景设计是利用虚拟现实语言开发设计办公楼建筑场景，所用开发程序占用较少的存储空间，程序运行效率高，利于计算机网络传输和应用。

1. 虚拟现实办公楼软件设计

虚拟现实办公楼建筑场景的软件设计采用渐进式软件开发模式对虚拟现实办公楼建筑场景进行需求分析、设计和编码。虚拟现实办公楼建筑场景设计由办公楼建筑设计、布局和周围环境设计组成，并采用模块化、组件化设计思想创建逼真的虚拟现实三维立体办公楼建筑场景。虚拟现实办公楼建筑场景设计的层次结构如图 6-13 所示。

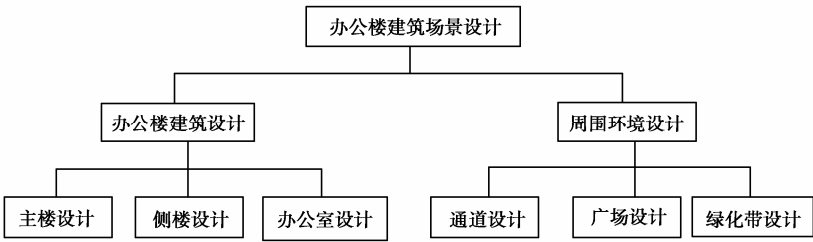


图 6-13 虚拟现实办公楼建筑场景设计的层次结构图

2. 虚拟现实办公楼建筑场景设计示例

虚拟现实办公楼建筑场景设计利用虚拟现实语言的各种简单节点和复杂节点创建生动、逼真的三维立体办公楼建筑场景，并利用内联节点实现子程序调用，实现模块化和组件化设计。该设计主要使用虚拟现实程序设计语言中的背景节点、视角节点、聚光灯节点、重定义节点、重用节点、面节点、坐标变换节点、内联节点、组节点、时间传感器节点、动态插补器节点、事件和路由等进行设计和开发，从而创建出更加真实的三维立体办公楼建筑场景。

【例 6-7】 虚拟现实办公楼建筑场景设计。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
```

```
Background {
    skyColor [
        1.0 1.0 1.0,
        0.1 0.5 1.0,
        1.0 1.0 0.8,
    ]
    skyAngle [1.32 1.571]
    groundColor [
        0.2 0.20 0.0,
        0.1 0.25 0.2,
        0.1 0.60 0.6,
    ]
    groundAngle [1.32 1.571]
}

DEF front Viewpoint {
    position 8.96606 13 62
    orientation -0.997096
                -0.0760361
                -0.0042219
                0.111409
    fieldOfView 0.785398
    description "front"
}

DEF left-front Viewpoint {
    position -49 8 42
    orientation -1.71948e-007
                -1
                -4.10955e-008
                0.968761
    fieldOfView 0.785398
    description "left-front"
}

DEF left-back Viewpoint {
    position -42.149 15 -31.9995
    orientation 0.0720334
                0.996234
                0.0482695
                4.05226
    fieldOfView 0.785398
    description "left-back"
}

DEF back Viewpoint {
    position 10.8394 21 -49
    orientation -0.0133604
                0.987994
```



```

}
Sound {
    source AudioClip
        {
            url "file/M001.mid"          #声音节点
            loop TRUE
        }
    minFront 100
    minBack 100
    maxBack 100
    maxFront 100
    spatialize FALSE
}

```

在主程序中调用的虚拟现实雪山场景设计中“MID-ZONG.wrl”子程序的源程序代码如下：

```

#VRML V2.0 utf8
DEF floor-1 Transform {
    children Inline {
        url "mid-zong-211.wrl" }
    }
DEF floor-2 Transform {
    children [
DEF floor-2-2 Transform {
    children Inline {
        url "office31.wrl"
    }
    translation 4 2.07492 0
    }
Transform {
    children [
        USE floor-2-2
    ]
    translation 4 0 0
    }
Transform {
    children [
        USE floor-2-2
    ]
    translation 8 0 0
    }
Transform {
    children [
        USE floor-2-2
    ]
    translation 12 0 0
    }
}

```

```
    ]
}
DEF floor-3 Transform {
    children [
DEF floor-3-3 Transform {
    children Inline {
        url    "LAB1.wrl"
    }
    translation    0 4.17366 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 8 0 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 16 0 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 24 0 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 0 2.07429 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 8 2.07429 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 16 2.07429 0
}
}
```

```
Transform {
    children [
        USE floor-3-3
    ]
    translation 24 2.07429 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 0 -2.07429 0
}
Transform {
    children [
        USE floor-3-3
    ]
    translation 24 -2.07429 0
}
}
```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击主程序，效果如图 6-14 所示。



图 6-14 虚拟现实办公楼场景的设计效果

6.3.2 虚拟现实医院设计

在虚拟现实医院场景设计中，可以创建医院大楼场景及医疗设施等，并运用树木、街道和建筑物等三维立体造型使虚拟现实场景与现实场景之间产生交流、互动，从而使虚拟现实医院场景设计得更加逼真、生动。虚拟现实医院建筑场景设计包括房屋建筑设计、街道设计及树木设计等。虚拟现实医院建筑场景设计是利用人工建筑和自然景观设计出人与自然相融合的三维立体场景。

1. 虚拟现实医院软件设计

虚拟现实医院建筑场景由医院主楼、两个侧楼及周围环境（包括街道、树木和草坪等）组成。虚拟现实医院建筑场景软件设计利用软件工程思想及渐进式软件开发模式进行需求分析、

设计和编码，并采用模块化、组件化设计思想进行开发设计。虚拟现实医院建筑场景设计的层次结构如图 6-15 所示。

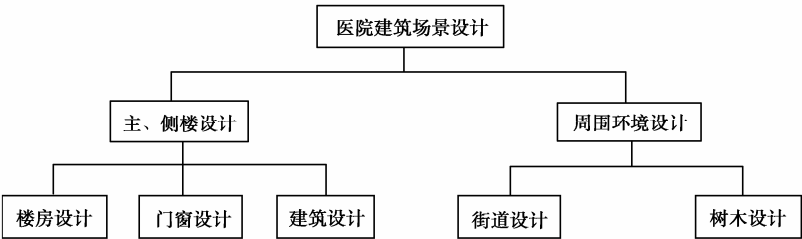


图 6-15 虚拟现实医院建筑场景设计层次结构图

2. 虚拟现实医院示例

虚拟现实医院建筑场景设计利用虚拟现实语言的各种节点创建生动、逼真的虚拟现实医院建筑三维立体场景，例如背景节点、视点导航节点、坐标变换节点、视角节点、内联节点、重定义节点、重用节点、面节点、时间传感器节点、动态插补器节点、事件和路由等，并利用内联节点实现子程序调用，实现模块化和组件化设计，从而使虚拟现实医院建筑场景更加完美。

【例 6-8】 虚拟现实医院建筑场景设计。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
Background {
  groundAngle [0.7,1.5,1.57]
  groundColor [0 0.333 0,
               0 0.4 0,
               0 0.5 0,
               0.62 0.67 0.60]
  skyAngle    [ 0.9,1.5,1.57 ]
  skyColor    [ 0.21 0.18 0.66,
               0.2 0.44 0.85 ,
               0.51 0.81 0.95,
               0.77 0.8 0.82]  }

NavigationInfo {
  avatarSize    [ 0.01, 0.2, 0.5]
  headlight     TRUE
  speed         1
  # type        "WALK"
  visibilityLimit 0
}

DEF Camera01 Viewpoint {
  position 20 0 0
  orientation 0 1 0 1.57
  description "Camera01"  }

DEF Camera02 Viewpoint {
  position -12 0 4
```

```

orientation    0 -1 0  1
description    "Camera02"
}
DEF Camera03 Viewpoint {
  position 3 0 12
  description    "Camera03"
}
DEF Camera04 Viewpoint {
  position 3 0 -12
  orientation    0 1 0  3.14
  description    "Camera04"
}
DEF bigdoor Viewpoint {
  position 1 -2.7 4
  description    "Camera05"
}
DEF tree1      Transform {                                     #树
  translation 12.5 -3.412 -10.5
  scale       1 1 1
  children [
    Billboard {
      axisOfRotation 0.0 1.0 0.0
      children [
        Shape {
          appearance Appearance {

            texture ImageTexture { url "tree1.png" }
          }
          geometry DEF TreeFace1 IndexedFaceSet {
            coord Coordinate {
              point [
                -1.51 0.05 0.0,   1.51 0.05 0.0,
                1.51 3.05 0.0,   -1.51 3.05 0.0,]}
            coordIndex [ 0, 1, 2, 3 ]
            texCoord TextureCoordinate {
              point [
                0.0 0.0, 1.0 0.0,
                1.0 1.0, 0.0 1.0,]}
            texCoordIndex [ 0, 1, 2, 3 ]
            solid FALSE }
          }
        ]
      }
    ]
  }
}
Transform {                                               #树

```

```

        translation 0 0 2.5
        scale 1 1 1
        children [ USE tree1]
    }
Transform {                                     #树
    translation 0 0 5
    scale 1 1 1
    children [ USE tree1]
}
Transform {                                     #树
    translation 0 0 7.5
    scale 1 1 1
    children [ USE tree1]
}
DEF tree2 Transform {                           #树
    translation 8.5 -3.412 6.5
    scale 1 1 1
    children [
        Billboard {
            axisOfRotation 0.0 1.0 0.0
            children [
                Shape {
                    appearance Appearance {

                        texture ImageTexture { url "tree1.png" }
                    }
                    geometry DEF TreeFace2 IndexedFaceSet {
                        coord Coordinate {
                            point [
                                -1.51 0.05 0.0, 1.51 0.05 0.0,
                                1.51 3.05 0.0, -1.51 3.05 0.0,]
                        coordIndex [ 0, 1, 2, 3 ]
                        texCoord TextureCoordinate {
                            point [
                                0.0 0.0, 1.0 0.0,
                                1.0 1.0, 0.0 1.0,]
                        texCoordIndex [ 0, 1, 2, 3 ]
                        solid FALSE }
                    }
                ]
            ]
        }
    ]
}
Transform {                                     #树
    translation -2.5 0 0
    scale 1 1 1

```

```

        children [ USE tree2]
    }
    Transform {                                     #树
        translation  -5 0 0
        scale  1 1 1
        children [ USE tree2]
    }
    Transform {                                     #树
        translation  -7.5 0 0
        scale  1 1 1
        children [ USE tree2]
    }
    Transform {                                     #树
        translation  -10 0 0
        scale  1 1 1
        children [ USE tree2]
    }
    Transform {                                     #树
        translation  -12.5 0 0
        scale  1 1 1
        children [ USE tree2]
    }
    Transform {
        children Transform {
            children  Inline {
                url  "build\MAINLEFT.wrl"
            }
        }
    }
    Transform {
        children Transform {
            children  Inline {
                url  "build\MAINOUT.wrl"
            }
        }
        translation  1.7 0 2
    }
    Transform {
        children Transform {
            children  Inline {
                url  "build\MAINRIGHT.wrl"
            }
        }
        translation  9 0 0
    }
    Transform {

```

```
children Shape {
  appearance Appearance {
    material Material {
      ambientIntensity 0
      diffuseColor      0.544676 0.7 0.0404636
    }
  }
  geometry Box {
    size20 0.2 20
  }
}
translation 3 -3.5 -3
}
Transform {
  translation 9.9 -1.512 0.0
  scale 0.1 0.1 0.1
  rotation 0 1 0 0.524
  children [ Inline {url "Oct.wrl"} ]
}
```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击主程序，效果如图 6-16 所示。



图 6-16 虚拟现实医院建筑场景设计效果

6.3.3 虚拟现实公路设计

虚拟现实公路场景设计是利用虚拟现实程序设计语言进行软件的设计开发，使虚拟公路场景与现实公路场景相融合，从而创建出逼真的三维立体公路场景。虚拟现实公路场景包括公路、人行道、绿化带及汽车等。在虚拟现实公路场景中可以体验虚拟世界给人们带来的现实所无法比拟的感受。

1. 虚拟现实公路软件设计

虚拟现实公路场景软件设计是利用先进的渐进式软件开发模式对虚拟现实公路场景进行需求分析、设计和编码。虚拟现实公路场景设计包括公路设计、人行道设计、绿化带设计及交通工具设计等。该设计采用模块化、组件化设计思想，开发设计层次清晰、结构合理的虚拟现实公路场景。虚拟现实公路场景设计的层次结构如图 6-17 所示。

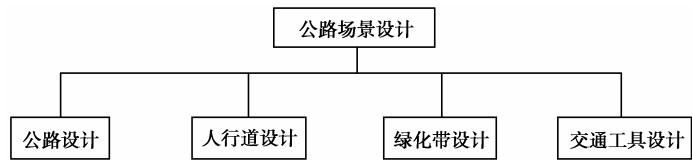


图 6-17 虚拟现实公路场景设计的层次结构图

2. 虚拟现实公路设计示例

虚拟现实公路场景设计利用虚拟程序设计语言中的简单几何节点、复杂节点和动态感知节点进行开发设计，包括背景节点、视角节点、坐标变换节点、内联节点、组节点、重定义节点、重用节点、面节点、时间传感器节点、动态插补器节点、事件和路由等，并利用内联节点实现子程序调用，实现模块化和组件化设计。该设计利用动态插补器节点设计行驶的汽车，使三维立体公路场景更加逼真、生动。

【例 6-9】 虚拟现实公路场景设计。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
Background {
    skyColor [0.98 0.98 0.98]
}
Transform {
    translation 0 -10 -60
    rotation 0 1 0 1.571
    scale 4.0 3.0 2.0
    children Inline {url "rode.wrl"}
}
DEF tree1 Transform {                                     #左侧树
    translation -18.5 -10 -35
    scale 1.0 2.0 1.0
    children [
        Billboard {
            axisOfRotation 0.0 1.0 0.0
            children [
                Shape {
                    appearance Appearance {

                        texture ImageTexture { url "tree1.png" }
                    }
                    geometry DEF TreeFace IndexedFaceSet {
                        coord Coordinate {
                            point [
                                -1.51 0.05 0.0, 1.51 0.05 0.0,
                                1.51 3.05 0.0, -1.51 3.05 0.0,]
                        coordIndex [ 0, 1, 2, 3 ]
                        texCoord TextureCoordinate {
                            point [
```

```
                                0.0 0.0, 1.0 0.0,  
                                1.0 1.0, 0.0 1.0,}}  
                                texCoordIndex [ 0, 1, 2, 3 ]  
                                solid FALSE }  
                                }  
                                ]  
                                }  
                                ]  
                                }  
Transform {  
    translation 0 0 30  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 25  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 20  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 15  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 10  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 5  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 0  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 -5  
    children [ USE tree1]  
}  
Transform {  
    translation 0 0 -10  
    children [ USE tree1]  
}  
Transform {
```

```
        translation 0 0 -15
        children [ USE tree1]
    }
Transform {
    translation 0 0 -20
    children [ USE tree1]
}
Transform {
    translation 0 0 -25
    children [ USE tree1]
}
Transform {
    translation 0 0 -30
    children [ USE tree1]
}
Transform {
    translation 0 0 -35
    children [ USE tree1]
}
Transform {
    translation 0 0 -40
    children [ USE tree1]
}
Transform {
    translation 0 0 -45
    children [ USE tree1]
}
Transform {
    translation 0 0 -50
    children [ USE tree1]
}
Transform {
    translation 0 0 -55
    children [ USE tree1]
}
Transform {
    translation 0 0 -60
    children [ USE tree1]
}
Transform {
    translation 0 0 -65
    children [ USE tree1]
}
Transform {
    translation 0 0 -70
    children [ USE tree1]
```

```

    }
Transform {
    translation 0 0 -75
    children [ USE tree1]
}
Transform {
    translation 0 0 -80
    children [ USE tree1]
}
DEF tree2 Transform { #右侧树
    translation 18.5 -10 -35
    scale 1.0 2.0 1.0
    children [
        Billboard {
            axisOfRotation 0.0 1.0 0.0
            children [
                Shape {
                    appearance Appearance {

                        texture ImageTexture { url "tree1.png" }
                    }
                    geometry DEF TreeFace1 IndexedFaceSet {
                        coord Coordinate {
                            point [
                                -1.51 0.05 0.0, 1.51 0.05 0.0,
                                1.51 3.05 0.0, -1.51 3.05 0.0,]
                        coordIndex [ 0, 1, 2, 3 ]
                        texCoord TextureCoordinate {
                            point [
                                0.0 0.0, 1.0 0.0,
                                1.0 1.0, 0.0 1.0,]
                        texCoordIndex [ 0, 1, 2, 3 ]
                        solid FALSE }
                    }
                ]
            ]
        }
    ]
}
Transform {
    translation 0 0 30
    children [ USE tree2]
}
Transform {
    translation 0 0 25
    children [ USE tree2]
}

```

```
Transform {  
    translation 0 0 20  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 15  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 10  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 5  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 0  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 -5  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 -10  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 -15  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 -20  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 -25  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 -30  
    children [ USE tree2]  
}  
Transform {  
    translation 0 0 -35
```

```

        children [ USE tree2]
    }
    Transform {
        translation 0 0 -40
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -45
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -50
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -55
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -60
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -65
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -70
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -75
        children [ USE tree2]
    }
    Transform {
        translation 0 0 -80
        children [ USE tree2]
    }
    #创建汽车造型
    DEF car Transform {
        translation 0 0 0
        scale 1.0 1.0 1.0
        children [ Inline {url "bluecar.wrl"}
    ]
    }
    DEF car1 Transform {

```

```

translation 0 0 0
scale 1.0 1.0 1.0
children [ Inline {url "yellowcar.wrl"}
]
}
}
DEF car2 Transform {
translation 0 0 0
scale 1.0 1.0 1.0
rotation 0 1 0 3.141
children [ Inline {url "greencar.wrl"}
]
}
}
DEF car3 Transform {
translation 0 0 0
scale 1.0 1.0 1.0
rotation 0 1 0 3.141
children [ Inline {url "redcar.wrl"}
]
}
}
DEF Time TimeSensor { #时间传感器
cycleInterval 10.0
loop TRUE
}
DEF run PositionInterpolator { #移动位置节点
key [ #相对时间的逻辑值
0.0,0.2,
0.4,0.6,
0.8,1.0
]
keyValue [ #空间坐标的位置值与相对时间的逻辑值
6 -10 -10,
6 -10 -30,
6 -10 -50,
6 -10 -70,
6 -10 -90,
6 -10 -120,
]
}
}
DEF Time1 TimeSensor { #时间传感器
cycleInterval 5.0
loop TRUE
}
DEF run1 PositionInterpolator { #移动位置节点
key [ #相对时间的逻辑值
0.0,0.2,

```

```
        0.4,0.6,
        0.8,1.0
    ]
    keyValue [                                     #空间坐标的位置值与相对时间的逻辑值
        12 -10 -10,
        12 -10 -30,
        12 -10 -50,
        12 -10 -70,
        12 -10 -90,
        12 -10 -120,

    ]
}
DEF Time2 TimeSensor {                           #时间传感器
    cycleInterval 5.0
    loop TRUE
}
DEF run2 PositionInterpolator {                  #移动位置节点
    key [                                         #相对时间的逻辑值
        0.0,0.2,
        0.4,0.6,
        0.8,1.0
    ]
    keyValue [                                     #空间坐标的位置值与相对时间的逻辑值
        -6 -10 -120,
        -6 -10 -90,
        -6 -10 -70,
        -6 -10 -50,
        -6 -10 -30,
        -6 -10 -10,

    ]
}
DEF Time3 TimeSensor {                           #时间传感器
    cycleInterval 3.0
    loop TRUE
}
DEF run3 PositionInterpolator {                  #移动位置节点
    key [                                         #相对时间的逻辑值
        0.0,0.2,
        0.4,0.6,
        0.8,1.0
    ]
    keyValue [                                     #空间坐标的位置值与相对时间的逻辑值
        -12 -10 -120,
        -12 -10 -90,
```

```

-12 -10 -70,
-12 -10 -50,
-12 -10 -30,
-12 -10 -10,

    ]
}

ROUTE Time.fraction_changed TO run.set_fraction
ROUTE run.value_changed TO car.set_translation
ROUTE Time1.fraction_changed TO run1.set_fraction
ROUTE run1.value_changed TO car1.set_translation
ROUTE Time2.fraction_changed TO run2.set_fraction
ROUTE run2.value_changed TO car2.set_translation
ROUTE Time3.fraction_changed TO run3.set_fraction
ROUTE run3.value_changed TO car3.set_translation

```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击主程序，效果如图 6-18 所示。

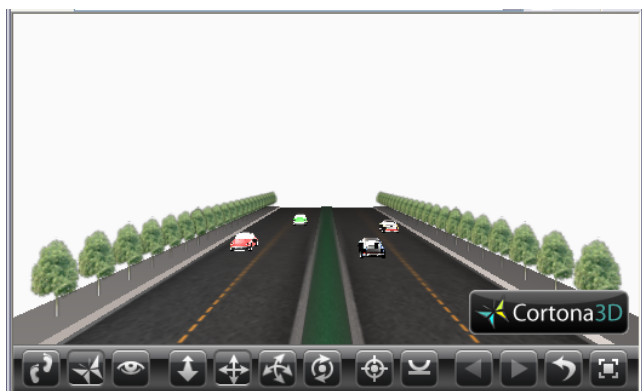


图 6-18 虚拟现实公路场景设计效果

6.3.4 虚拟现实客厅设计

虚拟现实客厅场景设计涵盖了室内设计、装饰和布局，创建房屋建筑、客厅内的各种三维立体造型，如沙发、茶几、窗帘、电视机、空调等，并利用动态交互智能感知节点实现三维动画场景，播放电视节目，使人感受虚拟现实场景中家庭温暖及和谐的气氛。

1. 虚拟客厅软件设计

虚拟客厅软件设计采用渐进式软件开发模式对虚拟现实多功能客厅进行设计和编码。虚拟现实多功能客厅场景设计由房屋设计、装饰和布局等组成，并采用模块化、组件化进行设计。虚拟现实客厅场景设计的层次结构如图 6-19 所示。

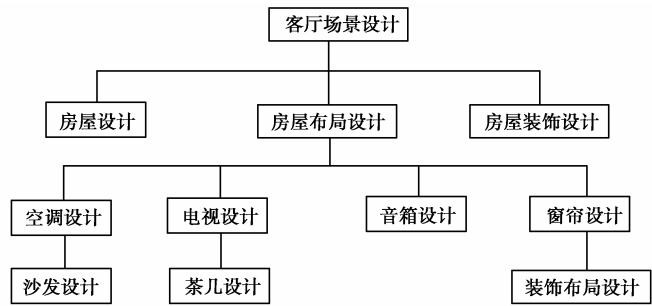


图 6-19 虚拟现实客厅场景设计的层次结构图

2. 虚拟现实客厅设计示例

虚拟现实客厅场景设计使用虚拟现实程序设计语言中的背景节点、视角节点、坐标变换节点、内联节点、组节点、重定义节点、重用节点、面节点、时间传感节点、动态插补器节点、声音节点和影视播放节点等进行设计和开发，并利用内联节点实现子程序调用，实现模块化和组件化设计。该设计利用动态智能感知节点设计自动门，利用影视和声音节点实现播放电视节目等动态智能感知场景。

【例 6-10】 虚拟现实客厅设计示例。

其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
Background
{
    groundColor [ 0.2 0.5 0.2,          # 背景颜色
                  0.3 0.5 0.3,
                  0.4 0.5 0.4 ]
    groundAngle [ 1.05, 1.6 ]
    skyColor [0.2 0.2 1
              1.0 1.0 1.0
              #   0.0 0.0 0.0
              ]
    skyAngle 1.571
}
Viewpoint                                # 视角 1
{
    position 10 -37.5
    orientation 0 1 0 -1.571
    description "View1"
    jump FALSE
}
Viewpoint                                # 视角 2
{
    position 0 0 -5
    orientation 0 1 0 -.2
    description "View2"
```

```

        jump FALSE
    }
Viewpoint                                # 视角 3
{
    position 1 0 -52.5
    orientation 0 1 0 -1.571
    description "view3"
    jump FALSE
}
Transform                                # 房屋建筑 后墙
{
    translation 11 0 -60
    #rotation 0 0 0 1.571
    children
    [
        Shape
        {
            appearance Appearance
            {
                material DEF blue Material
                {
                    specularColor 0.8 0.8 0.9

                    diffuseColor 1.0 1.0 1.0
                    shininess 0.1
                    ambientIntensity 0.2
                }
            }
            geometry Box { size 38.5 16.5 .5 }
        }
    ]
}
Transform                                # 右墙
{
    translation 30 0 -45
    rotation 0 1 0 1.571
    children
    [
        DEF wall Shape
        {
            appearance Appearance
            {
                material USE blue
            }
            geometry Box
            {

```

```

        size 30 16.5 .5
    }
}
]
}
Transform
{
    # 左墙
    translation -8 0 -45
    rotation 0 1 0 1.571
    children
        USE wall
}
Transform
    # 前面(左)的墙
{
    translation -0.8 0 -30
    #rotation 0 0 0 1.571
    children
    [
        DEF wallfront Shape
        {
            appearance Appearance
            {
                material USE blue
            }
            geometry Box { size 15 16.5 .5 }
        }
    ]
}
Transform
    # 前面(右)墙
{
    translation 22.8 0 -30
    #rotation 0 0 0 1.571
    children  USE  wallfront
}
Transform
    # 门桥
{
    # translation 11 4 -30
    translation 11 5.5 -30
    rotation 0 1 0 1.571
    children
    [
        Shape
        {
            appearance Appearance
            {
                material USE blue
            }

```

```

    }
    # geometry Box { size .5 2.5 9 }
    geometry Box { size .5 5.5 9 }
  }
]
}
Transform                                # 大沙发
{
  translation -2.5 -8.0 -52
  rotation 0 1 0 1.571
  scale 0.05 0.05 0.05
  children
  [
    Inline
    {
      url "shafa.wrl"
    }
  ]
}
Transform                                # 茶几
{
  translation 3.5 -5.0 -52
  rotation 0 1 0 -1.57
  scale 0.03 0.03 0.03
  children
  [
    Inline
    {
      url "chaji.wrl"
    }
  ]
}
Transform                                # 窗帘
{
  translation 12.5 5.0 -57.3
  rotation 0 0 0 -1.57
  scale 0.0014 0.0014 0.0014
  children
  [
    Inline
    {
      url "chuanglian.wrl"
    }
  ]
}
Transform{                                #窗户贴图

```

```

#rotation 1.0 0.0 0.0 0.785
translation 6.5 0.0 -59.6
children[
  Shape {
    appearance Appearance{
      material Material {
      }
      texture ImageTexture {
        url "haijing.jpg"
      }
    }
    geometry IndexedFaceSet {
      coord Coordinate {
        point [
          5.0 5.0 0.0,
          5.0 -5.0 0.0,
          -5.0 -5.0 0.0,
          -5.0 5.0 0.0
        ]
      }
      coordIndex[0,1,2,3 ]
      texCoord TextureCoordinate {
        point [1.0 1.0,
          1.0 0.0,
          0.0 0.0,
          0.0 1.0]
      }
      texCoordIndex [
        0,1,2,3
      ]
      solid FALSE
    }
  }
]
}
Transform{                                     #壁画
  #rotation 1.0 0.0 0.0 0.785
  translation 20.5 1.0 -59.6
  children[
    Shape {
      appearance Appearance{
        material Material {
        }
        texture ImageTexture {
          url "baishe1.jpg"
        }
      }
    }
  ]
}

```

```

    }
    geometry IndexedFaceSet {
        coord Coordinate {
            point [
                4.0 2.0 0.0,
                4.0 -4.0 0.0,
                -4.0 -4.0 0.0,
                -4.0 2.0 0.0
            ]
        }
        coordIndex[0,1,2,3    ]
        texCoord TextureCoordinate {
            point [1.0 1.0,
                1.0 0.0,
                0.0 0.0,
                0.0 1.0,
            ]
        }
        texCoordIndex [
            0,1,2,3
        ]
        solid FALSE
    }
}

]
}

Transform{                                     #壁画
    rotation 0.0 1.0 0.0 1.571
    translation -7.5 1.0 -50.6
    children[
        Shape {
            appearance Appearance{
                material Material {
                }
                texture    ImageTexture {
                    url    "baishe2.jpg"
                }
            }
        }
        geometry IndexedFaceSet {
            coord Coordinate {
                point [
                    4.0 2.0 0.0,
                    4.0 -4.0 0.0,
                    -4.0 -4.0 0.0,
                    -4.0 2.0 0.0
                ]
            }
        }
    ]
}

```

```

    }
    coordIndex[
        0,1,2,3
    ]
    texCoord TextureCoordinate {
        point [

            1.0 1.0,
            1.0 0.0,
            0.0 0.0,
            0.0 1.0,

        ]
    }
    texCoordIndex [
        0,1,2,3
    ]
    solid FALSE
}
}
]
}
Transform #电视
{
    translation 25.5 -8 -42.5
    rotation 0 1 0 -1.57
    scale 0.007 0.007 0.007
    children
    [
        Inline
        {
            url "dianshi.wrl"
        }
    ]
}
Transform { #声音、影视播放
    translation 25.5 -1.1 -42.6
    rotation 0 1 0 1.571
    children [
        Background {
            skyColor 0.1 0.5 0.3
        }
        Sound { #声音节点
            source DEF movie MovieTexture #影视文件节点
            {
                url "hw.mpg" #影视文件
            }
        }
    ]
}

```

```

        loop TRUE
        repeatS FALSE
        repeatT FALSE
    }
    location 0.0 0.0 0.0
    direction 0 0 1
    spatialize FALSE
    maxFront 1000.0
    maxBack 100.0
    minBack 10.0
    minFront 10.0
}
Shape {
    appearance Appearance{
        material Material {
            diffuseColor 0.0 0.0 0.0
        }
        texture USE movie
    }
    geometry Box {
        size 7.5 5.8 0.01
    }
}
]
}
Transform
{
    translation 25.5 -9 -52.5
    rotation 0 1 0 -1.047
    scale 0.016 0.016 0.016
    children
    [ Inline {url "yinxiang1.wrl"}
    ]
}
Transform
{
    translation 26.5 -7.5 -35.5
    rotation 0 1 0 -1.57
    scale 0.015 0.014 0.015
    children
    [ Inline {url "yinxiang2.wrl"}
    ]
}
Transform
{
    translation 20.5 5 -31.5

```

#循环播放

#S 代表水平

#T 代表垂直

#声音的位置坐标

#声音的传播方向（沿 Z 轴）面向观众

#环绕立体声

#向前传播最大距离

#向后传播最大距离

#使用影视节点

#音箱 1

#音箱 2

#空调

```

        rotation 0 1 0 3.141
        scale 0.0005 0.0005 0.0005
        children
        [
            Inline
            {
                url "kongdiao.wrl"
            }
        ]
    }
Transform                                     #客厅
{
    translation 11 5.0 -29.5
    children
    [
        Inline
        {
            url "keting.WRL"
        }
    ]
}
Group {
    children [
        DEF ldoor Transform                     #左门
        {
            translation 8.8 -2.8 -30
            rotation 0 1 0 1.571
            children
            [
                DEF door Shape
                {
                    appearance Appearance
                    {
                        material Material
                        {
                        }
                        texture ImageTexture
                        {
                            url "men.jpg"
                        }
                    }
                    geometry Box { size .5 11 4.3 }
                }
            ]
        }
    ]
}
DEF rdoor Transform                           #右门

```

```

    {
        translation 13.15 -2.8 -30
        rotation 0 1 0 1.571
        children USE door
    }
    DEF Touch TouchSensor {
        enabled TRUE
    }
]
}
DEF Time TimeSensor { #时间传感器
    cycleInterval 8.0
    loop FALSE
}
DEF flyinter1 PositionInterpolator { #移动位置节点
    key [ #相对时间的逻辑值
        0.0,0.2,0.3,0.4,0.5,
        0.6,0.7,0.8,1.0 ,
    ]
    keyValue [ #空间坐标的位置值与相对时间的逻辑值
        13.15 -2.8 -29.8
        14.15 -2.8 -29.8
        15.15 -2.8 -29.8
        16.15 -2.8 -29.8
        17.15 -2.8 -29.8
        16.15 -2.8 -29.8
        15.15 -2.8 -29.8
        14.15 -2.8 -29.8
        13.15 -2.8 -29.8
    ]
}
DEF flyinter2 PositionInterpolator { #移动位置节点
    key [ #相对时间的逻辑值
        0.0,0.1,0.2,0.4,0.5,
        0.6,0.7,0.8,1.0
    ]
    keyValue [ #空间坐标的位置值与相对时间的逻辑值
        8.8 -2.8 -29.8
        7.8 -2.8 -29.8
        6.8 -2.8 -29.8
        5.8 -2.8 -29.8
        4.8 -2.8 -29.8
        5.8 -2.8 -29.8
        6.8 -2.8 -29.8
        7.8 -2.8 -29.8
        8.8 -2.8 -29.8
    ]
}

```

```
    ]  
  }  
  
  ROUTE Touch.touchTime TO Time.startTime  
  ROUTE Time.fraction_changed TO flyinter1.set_fraction  
  ROUTE Time.fraction_changed TO flyinter2.set_fraction  
  ROUTE flyinter1.value_changed TO ldoor.set_translation  
  ROUTE flyinter2.value_changed TO rdoor.set_translation
```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击主程序，效果如图 6-20 所示。



图 6-20 虚拟现实客厅场景的设计效果

6.4 虚拟三维人体骨骼设计

1. 虚拟现实三维人体骨骼软件设计

虚拟现实三维人体骨骼由颅骨、躯干骨、上肢骨和下肢骨组成，其中颅骨由脑颅骨、面颅骨组成，躯干骨由椎骨、肋骨和胸骨组成，上肢骨由上肢带骨、自由上肢骨组成，下肢骨由下肢带骨、自由下肢骨组成。人体骨骼的具体构造如下。

- (1) 脑颅骨：额骨、顶骨、枕骨、蝶骨、筛骨、颞骨。
- (2) 面颅骨：上颌骨、鼻骨、泪骨、颧骨、下鼻骨、腭骨、犁骨、下颌骨、舌骨。
- (3) 椎骨：颈椎、胸椎、腰椎、骶骨、尾骨。
- (4) 肋骨：12 对。
- (5) 胸骨。
- (6) 上肢带骨：锁骨、肩胛骨。
- (7) 自由上肢带骨：肱骨、桡骨、尺骨、手骨（腕骨、掌骨、指骨）。
- (8) 下肢带骨：髌骨。
- (9) 自由下肢骨：股骨、髌骨、胫骨、腓骨、足骨（跗骨、跖骨、趾骨）。

虚拟现实三维人体骨骼软件设计是利用软件工程思想及渐进式软件开发模式对虚拟现实三维人体骨骼进行开发、设计、编码、调试和运行。该设计采用模块化、组件化、层次化设计思想，开发设计层次清晰、结构合理的虚拟现实三维人体骨骼。虚拟现实三维人体骨骼设计的层次结构如图 6-21 所示。

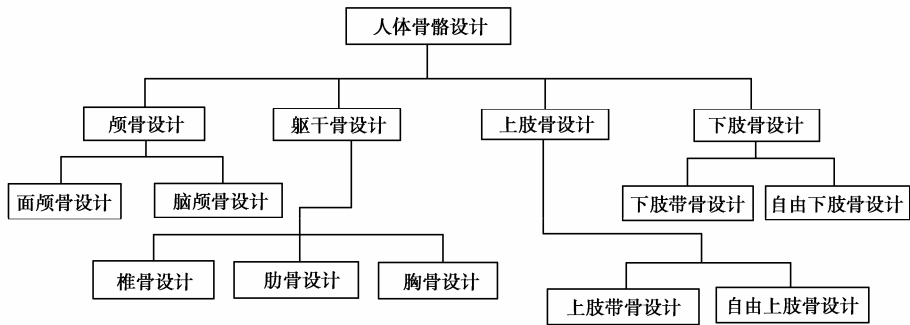


图 6-21 虚拟现实三维人体骨骼设计的层次结构图

2. 虚拟现实三维人体骨骼示例

虚拟现实三维人体骨骼设计使用虚拟现实程序设计语言中的背景节点、视角节点、坐标变换节点、内联节点等进行设计和开发，并利用内联节点实现子程序调用，实现模块化和组件化设计。

【例 6-11】 虚拟现实三维人体骨骼设计示例。
其实现的主程序（VRMLmain.wrl）代码如下：

```
#VRML V2.0 utf8
Background
{
    skyColor
        [0.2 0.2 1.0          # 背景颜色
         9.8 9.8 9.8]
    skyAngle 1.571
}
Viewpoint                                # 正面视角 1
{
    position 0 0 10
    description "View1"
    jump FALSE
}
Viewpoint                                # 背面视角 2
{
    position 15 0 10
    description "View2"
    jump FALSE
}
Transform                                # 人体骨骼正面
{
    translation 0 -3.5 0
    rotation 0 0 0 1.571
    scale 10 10 10
    children
    [
```

```

        Inline
        {
            url "rentiguge.wrl"

        }
    ]
}

Transform # 颅骨
{
    translation -2 3.2 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   { url "lugu.wrl" }
    ]
}

Transform # 颅骨 22 块
{
    translation -2.05 3.0 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   { url "ershier.wrl" }
    ]
}

Transform # 脑颅骨
{
    translation -1 3.3 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   { url "naolugu.wrl" }
    ]
}

Transform # 脑颅骨 8 块
{
    translation -1.05 3.2 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   { url "ba.wrl" }
    ]
}

Transform # 面颅骨
{
    translation -1 3.0 0

```

```

        rotation 0 0 0 1.571
        scale 0.05 0.05 0.05
        children
        [   Inline   {   url   "mianlugu.wrl"   }
          ]
    }
Transform                                     # 面颅骨 15 块
{
    translation -1.05 2.9 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "shiwu.wrl"           }
      ]
}
Transform                                     # 额骨
{
    translation 1 3.3 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "egu.wrl"   }
      ]
}
Transform                                     # 颧骨
{
    translation 1 3.15 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "quangu.wrl"         }
      ]
}
Transform                                     # 上颌骨
{
    translation 1 3.0 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "shangegu.wrl"       }
      ]
}
Transform                                     # 下颌骨
{
    translation 1 2.85 0
    rotation 0 0 0 1.571

```

```

        scale 0.05 0.05 0.05
        children
        [   Inline   {   url   "xiaoegu.wrl"   }
        ]
    }
Transform                                     # 肩关节
{
    translation -0.6 2.4 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "jiangujie.wrl"   }
    ]
}
Transform                                     # 肘关节
{
    translation -1.6 2.4 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "zouguanjie.WRL"   }
    ]
}
Transform                                     # 腕关节
{
    translation -2.6 2.4 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "wanguanjie.WRL"   }
    ]
}
Transform                                     # 脊柱
{
    translation -0.6 0.9 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "jigu.WRL"   }
    ]
}
Transform                                     # 骨盆
{
    translation -1.0 0.5 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08

```

```

        children
        [   Inline   {   url   "gupan.WRL"   }
        ]
    }
Transform                                     # 髌关节
{
    translation -1.0 0.2 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "kuanguanjie.WRL"   }
    ]
}
Transform                                     # 膝关节
{
    translation -1.0 -1.5 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "xiguanjie.WRL"   }
    ]
}
Transform                                     # 距小腿关节
{
    translation -1.0 -3.3 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "juxiaotui.WRL"   }
    ]
}
Transform                                     # 锁骨
{
    translation 0.6 2.4 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "suogu.WRL"   }
    ]
}
Transform                                     # 肩胛骨
{
    translation 0.8 1.9 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children

```

```
[   Inline   {   url   "jianguanjie.WRL"   }
]
}
Transform                                     # 胸骨
{
    translation 0.0 1.6 0.4
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "xionggu.WRL"   }
    ]
}
Transform                                     # 肋骨
{
    translation 0.8 1.5 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "leigu.WRL"   }
    ]
}
Transform                                     # 股骨
{
    translation 0.8 -0.8 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "gugu.wrl"   }
    ]
}
Transform                                     # 髌骨
{
    translation 0.8 -1.55 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "bingu.wrl"   }
    ]
}
Transform                                     # 胫骨
{
    translation 0.0 -2.55 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "jinggu.wrl"   }
    ]
}
```

```

    ]
}
Transform                                # 腓骨
{
    translation 0.8 -2.55 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "paigu.wrl"   }
    ]
}
Transform                                # 跗骨
{
    translation 0.6 -3.2 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "fugu.wrl"   }
    ]
}
Transform                                # 跖骨
{
    translation 0.7 -3.35 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "shigu.wrl"   }
    ]
}
Transform                                # 趾骨
{
    translation 0.8 -3.5 0
    rotation 0 0 0 1.571
    scale 0.05 0.05 0.05
    children
    [   Inline   {   url   "zhigu.wrl"   }
    ]
}
Transform                                # 顶骨
{
    translation 14 3.2 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "dinggu.wrl"   }
    ]
}

```

```

    }
Transform                                     # 颞骨
{
    translation 16 3.2 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "neigu.WRL"   }
      ]
}
Transform                                     # 枕骨
{
    translation 16 3.0 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "zhenggu.WRL"   }
      ]
}
Transform                                     # 肱骨
{
    translation 14.0 2.4 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "honggu.WRL"   }
      ]
}
Transform                                     # 桡骨(外桡)
{
    translation 12.8 2.4 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "qiaogu.WRL"   }
      ]
}
Transform                                     # 尺骨(里尺)
{
    translation 12.8 1.8 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "chigu.wrl"   }
      ]
}

```

```
Transform                                     # 胸椎
{
    translation 15.0 1.8 0.6
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "xiongzui.WRL"   }
      ]
}

Transform                                     # 胸椎 12 块
{
    translation 15.0 1.6 0.6
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "shier.wrl"   }
      ]
}

Transform                                     # 腰椎
{
    translation 15.0 1.05 0.6
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "yaozui.wrl"   }
      ]
}

Transform                                     # 腰椎 5 块
{
    translation 15.0 0.85 0.6
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "wu.wrl"   }
      ]
}

Transform                                     # 髌骨
{
    translation 16.0 0.5 0
    rotation 0 0 0 1.571
    scale 0.08 0.08 0.08
    children
    [   Inline   {   url   "kuangu.WRL"   }
      ]
}

Transform                                     # 骺骨
```

```
{
  translation 15.0 0.45 0.6
  rotation 0 0 0 1.571
  scale 0.05 0.05 0.05
  children
  [   Inline   {   url   "digu.WRL"   }
  ]
}
Transform                                     # 尾骨
{
  translation 15.0 0.25 0.6
  rotation 0 0 0 1.571
  scale 0.05 0.05 0.05
  children
  [   Inline   {   url   "weigu.wrl"   }
  ]
}
```

在主程序中调用的其他子程序请参见本书附带的源代码（登录 www.hxedu.com.cn 下载）。双击主程序，效果如图 6-22 所示。

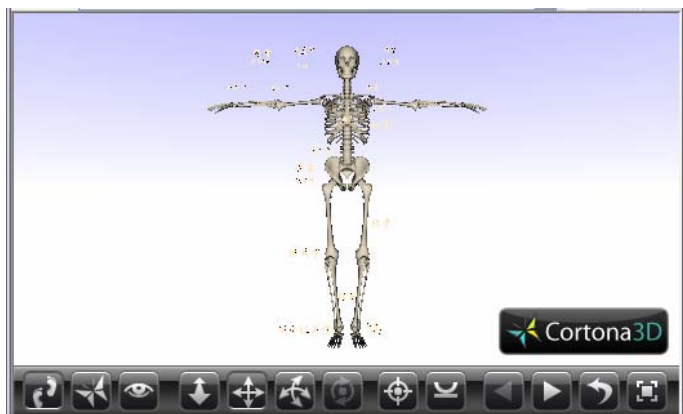


图 6-22 虚拟现实三维人体骨骼设计效果

第 7 章 虚拟现实与MATLAB接口应用

7.1 MATLAB的简单介绍

7.1.1 MATLAB的概述

MATLAB 是一种高效的科学计算软件，它将强大的计算功能、可视化、程序设计整合在一个极易使用的开发环境中，在该环境下，各种问题和计算都以一种数学的方式来表达。MATLAB 广泛地使用在从个人计算机到超级计算机范围内的各种计算机上，其应用领域相当广泛，如：

- (1) 数学和计算。
- (2) 算法开发。
- (3) 获取数据。
- (4) 建模、仿真。
- (5) 数据分析和可视化编程。
- (6) 科学和工程作图。

MATLAB 是一个交互操作系统，其基本数据元素为阵列，且阵列的维数没有限制。这就使得用户可以解决工程上的许多计算问题，尤其是那些带有矩阵和向量的公式，而且使用起来相当简洁。这些问题若使用 C 或 FORTRAN 语言编程来实现，则需要花费较长的时间。

MATLAB 名称的本来含义是矩阵实验室 (Matrix Laboratory)，其创建的最初目的就是为了使烦琐的矩阵处理和运算变得容易。最初的 MATLAB 是用 FORTRAN 语言编写的，并且采用了当时极为流行的线性代数软件包 LINPACK 和基于特征值计算的软件包 EISPACK 中大量可靠的子程序。

随着 MATLAB 的应用范围越来越广，MATLAB 的最初开发者 Morle 博士等数学家和一些软件专家成立了现在的 MathWorks 公司，如今的 MATLAB 已用 C 语言进行了全面的改写，增添了丰富多彩的图形处理功能，并且形成了一个规模庞大的工具箱 (Toolbox)。

工具箱中包含了大量 MATLAB 函数 (M 文件)，从而大大延伸了 MATLAB 处理特定问题的能力，如信号处理 (Signal Processing)、控制系统 (Control Systems)、神经网络 (Neural Networks)、模糊逻辑 (Fuzzy Logic)、小波分析 (Wavelet Analysis) 和鲁棒控制 (Robust Control) 等，其包含了大量的现代工程技术学科的内容，极为实用。

现在，MathWorks 公司推出的最新产品 MATLAB R2009，适用于各种硬件平台和操作系统，功能进一步增强，用户界面更为人性化。

7.1.2 MATLAB的启动、退出及工作界面

1. 启动MATLAB

MATLAB 可以在 Windows 平台或 UNIX 平台下启动。

在 Window 平台下启动 MATLAB 只需要双击 Windows 桌面上的  快捷图标即可，这个快捷方式是在安装时由用户创建的。如果用户在 DOS 窗口下启动 MATLAB，则在 DOS 状态

下直接输入“MATLAB”命令，按回车键即可。在 UNIX 操作系统下直接输入“MATLAB”命令就可以启动 MATLAB 了。

2. 退出 MATLAB

单击【File】菜单下的【Exit MATLAB】选项，或者在命令窗口输入“quit”命令就可以退出 MATLAB。另外还可以编写脚本程序达到退出 MATLAB 的目的。

3. MATLAB 界面

当用户启动 MATLAB 后，将看到 MATLAB 界面，其中包括文件管理工具、变量以及 MATLAB 相关的一些应用软件。

在第一次启动 MATLAB 时，将出现图 7-1 所示的界面，在以后的应用中用户可以通过单击【File】菜单下的【Preference】选项设置自己所喜欢的界面。例如，用户可以设置命令窗口的字体等。

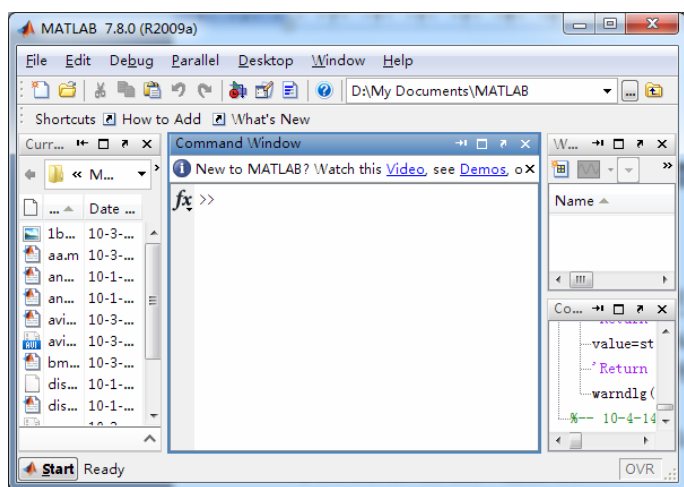


图 7-1 MATLAB 界面

如果需要执行命令，即可在命令窗口的“>>”右边输入命令语句，然后按“Enter”键即可。

7.1.3 Simulink

1. Simulink 简介

1990 年，MathWorks 软件公司为 MATLAB 提供了新的控制系统模型化图形输入与仿真工具，并命名为 SIMULAB，该工具很快就在控制工程界获得了广泛的认可，使得仿真软件进入了模型化图形组态阶段。1992 年正式将该软件更名为 Simulink。

Simulink 的出现给控制系统分析与设计带来了福音。它有两个主要功能：Simu（仿真）和 Link（连接），即该软件可以利用鼠标在模型窗口上绘制出所需要的控制系统模型，然后利用 Simulink 提供的功能来对系统进行仿真和分析。

在实际工程中，控制系统的结构往往很复杂，如果不借助专用的系统建模软件，则很难准确地把一个控制系统的复杂模型输入计算机，对其进行进一步的分析与仿真。因此，熟悉掌握 Simulink 对于一个当代从事自动控制方面工作的人来说是非常必要的。

2. Simulink应用

Simulink 是 MATLAB 软件的扩展，它是实现动态系统建模和仿真的一个软件包，它与 MATLAB 语言的主要区别在于：它与用户交互接口是基于 Windows 的模型化图形输入的，从而使得用户可以把更多的精力投入到系统模型的构建而非语言的编程上。

所谓模型化图形输入是指 Simulink 提供了一些按功能分类的基本系统模块，用户只需要知道这些模型的输入、输出及模块的功能，而不必考察模块内部是如何实现的，通过对这些基本模块的调用，再将它们连接起来就可以构成所需要的系统模型（以.mdl 文件进行存取），进而进行仿真与分析。

Simulink 的最新版本是 Simulink6.6，MATLAB6.5 里的版本为 5.0，它们的基本功能相差不多。

3. Simulink的运行方法及界面

Simulink 的启动有 3 种方式。

(1) 启动 MATLAB 后，单击 MATLAB 主窗口的快捷按钮来打开 Simulink Library Browser 窗口，如图 7-2 所示。

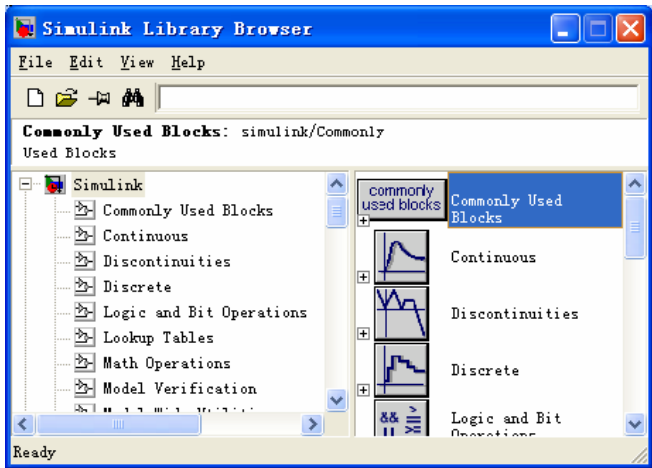


图 7-2 Simulink Library Browser 窗口

(2) 在 MATLAB 命令行窗口中输入“Simulink”，结果也是在桌面上出现一个 Simulink Library Browser 的窗口，在这个窗口中列出了按功能分类的各种模块的名称。

(3) 单击 MATLAB 的【File】菜单下【New】选项的【Model】命令。

在 MATLAB 命令窗口中输入“simulink3”，结果是在桌面上出现一个用图标形式显示的 Library: simulink3 的 Simulink 模块库窗口，如图 7-3 所示。这两种模块库窗口界面只是不同的显示形式，用户可以根据个人喜欢进行选用，一般第 2 种窗口更直观、更形象，适合初学者，但使用时会打开太多的子窗口。

Simulink 启动后，便可打开图 7-4 所示的 Simulink 的仿真编辑窗口，用户此时就可以开始编辑自己的仿真程序了。

打开已经存在的模型文件也有以下 3 种方法。

(1) 直接在 MATLAB 命令窗口输入模型文件名（不要加扩展名“.mdl”），这要求文件在当前的路径范围内。

- (2) 单击 MATLAB 的【File】菜单下的【Open】选项，在弹出的浏览窗口中选择所需的模型文件名。
- (3) 单击图 7-2 中的图标。

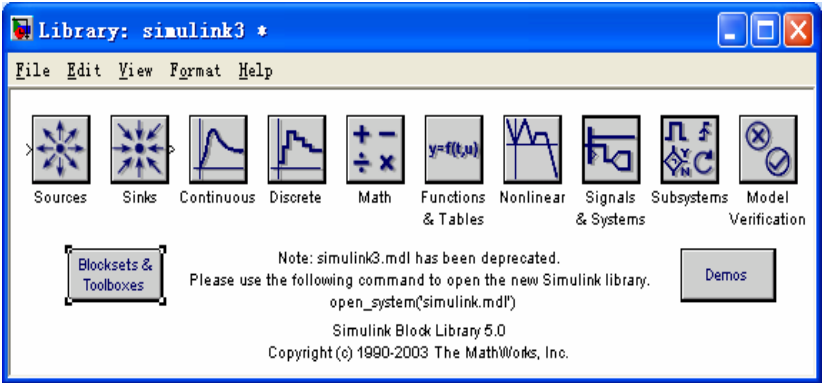


图 7-3 Simulink 模块库窗口

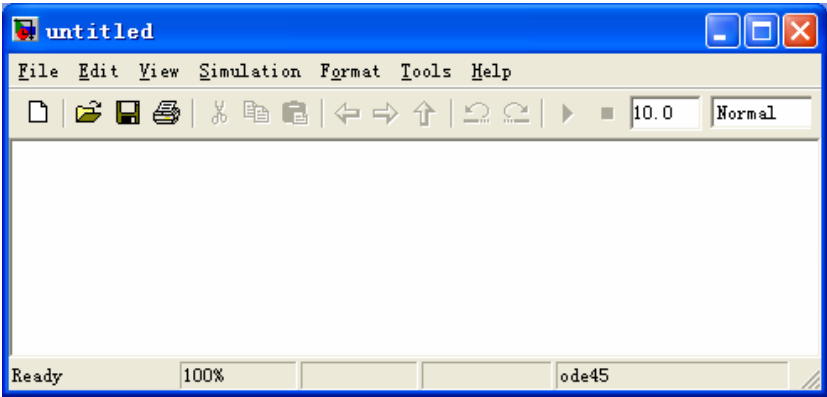


图 7-4 Simulink 仿真编辑窗口

7.2 虚拟实现工具箱的MATLAB函数

7.2.1 MATLAB的接口函数

作为一种新推出的工具箱，虚拟实现工具箱包含的 MATLAB 接口函数较其他成熟的工具箱而言少得多，表 7-1 列出了这些函数及其功能描述。

表 7-1 MATLAB 的接口函数的功能

函 数	功 能 描 述
vrinstall	安装和检测虚拟现实工具箱的组成部分
vrclear	清除所有的虚拟世界对象
vrgetpref	获得虚拟现实工具箱的参数值
vrsetpref	设置虚拟现实工具箱的参数值

续表

函 数	功 能 描 述
vrlib	为虚拟现实工具箱打开 Simulink 模块库
vrview	用浏览器打开的世界对象名称
vrwho	列出内存中的虚拟世界对象名称
vrwhos	列出内存中虚拟世界对象的详细清单

1. vrinstall函数

vrinstall 函数为安装和检测虚拟现实工具箱的组成部分。其调用格式如下：

```
vrinstall('action')
vrinstall('action','component')
x = vrinstall('action', 'component')
```

其中，action 为确定此函数的操作行为，它的值是 ‘-interactive’、‘-selftest’、‘-check’、‘-install’ 和 ‘uninstall’。其值含义如下：

（1）‘-selftest’：检查虚拟现实工具箱的完整性。如果此函数报告一个错误，则表示应当重新安装虚拟现实工具箱。

（2）‘-interactive’：对安装的虚拟现实工具箱的组成部分进行检查，并且列出没有安装（也可以选择安装）的组成部分的清单，这是系统默认的行为。

（3）‘-check’ 检查指定部分是否已经安装。如果指定的部分已安装，返回“1”；如果指定的部分没有安装，则返回“0”。如果没有指定一个部分，则列出组成部分和它们的状态。

（4）‘-install’：对指定的部分进行安装，此行为需要指定组成部分的名字，它们用于安装或重新安装虚拟现实工具箱的组成部分。所有的组成部分可以使用这个命令安装，但是对于通过此方法安装的 VRML 浏览器需要使用系统标准卸载过程来卸载。

（5）‘uninstall’：对指定的组成部分进行卸载。可以使用此方法卸载 VRML 编辑器，但并没有从内存中清除文件。对于卸载 VRML 浏览器，需要退出 MATLAB，在控制面板中使用“添加/删除程序”命令来完成卸载。

component 函数为行为部分的名字，其值为 ‘viewer’ 和 ‘editor’。

【例 7-1】 安装 VRML 编辑器的 MATLAB 命令。

在命令窗口中输入如下代码：

```
>> vrinstall('-install','editor')
Starting editor installation ...
Done.
```

下面再安装 VRML 浏览器，代码如下：

```
>> vrinstall('-install','viewer')
Installing blaxxun Contact viewer ...
Do you want to use OpenGL or Direct3D acceleration? (o/d) o
Starting viewer installation ...
Done.
```

这时用 MATLAB 函数检查一下是否已安装了 VRML 浏览器和编辑器：

```
>> vrinstall('-check')
```

检查结果如下：

```
External VRML viewer:    installed
VRML editor:             installed
```

表明 VRML 浏览器和编辑器均已安装。

2. vrclear函数

vrclear 函数从内存中清除所有已经关闭的虚拟世界。其调用格式如下：

```
vrclear
vrclear('-force')
```

此函数为从内存中清除所有处于关闭状态的虚拟世界，并且使所有与之相关的 Vrworld 对象无效。在清除的过程中，打开的虚拟世界不受影响。此命令通常用于确保内存最大限度地被释放出来，或者将其作为一个普通的清除命令来使用。

3. vrgetpref函数

vrgetpref 函数用于读取虚拟现实工具箱的参数值。其调用格式如下：

```
x = vrgetpref
x = vrgetpref('preference_name')
x = vrgetpref('preference_name','factory')
x = vrgetpref('factory')
```

此函数返回给定的虚拟现实工具箱参数的值。当使用 ‘-default’ 开关时，返回给定属性的默认值，而不是实际的值。

例如想知道 VRML 编辑器的主程序 vrbuid2.exe 在哪个路径下，则调用函数 vrgetpref 如下：

```
>> vrgetpref('Editor')
ans =
%matlabroot\toolbox\sl3d\vrealm\program\vrbuild2.exe %file
```

4. vrsetpref函数

vrsetpref 函数为设置虚拟现实工具箱的参数值。其调用格式如下：

```
vrsetpref('preference_name','preference_value')
vrsetpref('factory')
```

其中，preference_name 为参数的名字；preference_value 为参数的新值；设置的参数需要重新启动 MATLAB 之后才能生效。

当 factory 被用作一个单一参数时，所有的参数都恢复为默认值。如果 factory 不被用作单独的参数值，那么指定的另一个参数将被恢复成默认值。

5. vrlib函数

该函数为虚拟现实工具箱打开 Simulink 功能模块库。其调用格式如下：

```
vrlib
```

虚拟现实工具箱的 Simulink 库有 9 个模块：VR Sink、VR To Video、VR Text Output、VR Traor、VR Placeholder、Joystick Input、Space Mouse Input、Utilities、Demos。作为选择，用

户可以从 Simulink 图表中访问这些模块，其操作方法是在 Simulink 窗口中，或者在“View”菜单中单击“Show Library Browser”。

例如在 MATLAB 命令窗口中输入：

```
>> vrlib
```

则出现虚拟现实工具箱的 Simulink 库模块，效果如图 7-5 所示。

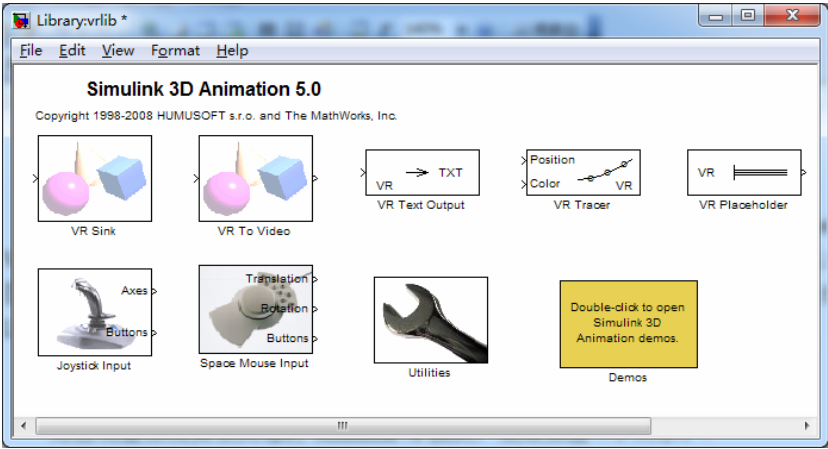


图 7-5 虚拟现实工具箱的 Simulink 库模块

可以对其中的某个模块进行修改，比如双击“VR Text Output”，则弹出对话框，可以对其中的参数进行修改，如图 7-6 所示。

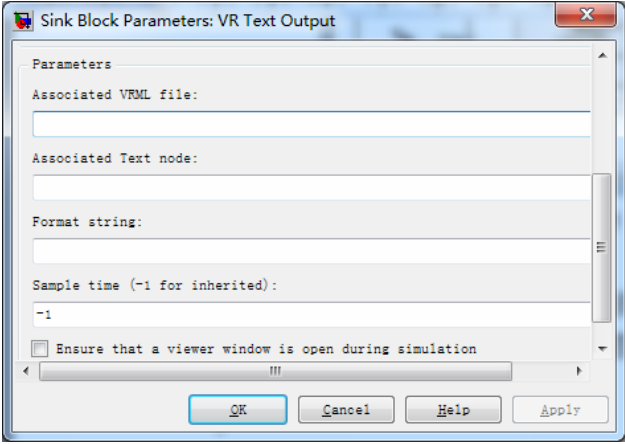


图 7-6 “VR Text Output” 参数设置对话框

6. vrview函数

此函数为在浏览器中观察网页中列出的虚拟世界。其调用格式如下：

```
vrview
x = vrview('filename')
x = vrview('filename','-internal')
x = vrview('filename','-web')
```

其中, filename 为浏览的*.wrl 虚拟文件; -internal 为返回的 Simulink 3D 虚拟世界; -web 为网页中列出的虚拟世界。

例如, 在 MATLAB 命令窗口中输入:

```
>> vrview
```

则打开浏览器, 效果如图 7-7 所示。

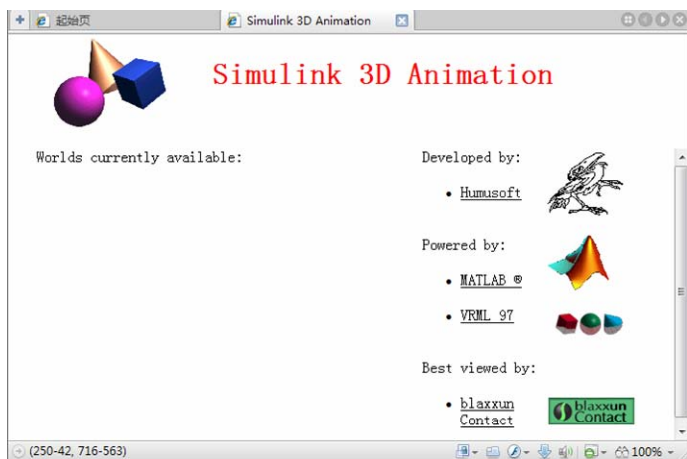


图 7-7 运行 vrview 函数

在浏览器中看到虚拟世界列表是空的, 当前没有虚拟世界可供浏览, 这是因为还没有打开虚拟世界, 还需调用 open 函数将虚拟世界打开。

7. vrwho 函数

此函数为列出内存中的虚拟世界, 并且为每一个虚拟世界生成 Vrworld 对象。其调用格式如下:

```
vrwho  
x=vrwho
```

如果不指定一个输出参数, 那么 vrwho 将在 MATLAB 命令窗口中输出内存中的虚拟世界清单; 如果指定一个输出参数, vrwho 在内存中为每一个虚拟世界产生一个 Vrworld 对象, 然后返回一个 Vrworld 对象矢量。Vrworld 对象表示当前内存中的虚拟世界。

例如, 指定一个虚拟世界, 并打开它, 其代码如下:

```
>> world=vrworld('vrmount.wrl','Car in the Mountains');  
>> open(world)
```

并查看结果:

```
>> vrwho  
VR Car in the Mountains (E:/MATLAB2009a/toolbox/sl3d/sl3ddemos/vrmount.wrl)
```

8. vrwhos 函数

此函数为列出内存中虚拟世界对象的有关信息的详细清单。其调用格式如下:

```
vrwhos
```

vrwhos 与 vrwho 函数之间的区别请参看以下示例。

紧接上列，在命令窗口中输入：

```
>> vrwhos
```

输出如下：

```
VR Car in the Mountains
Loaded from 'E:/MATLAB2009a/toolbox/sl3d/sl3ddemos/vrmount.wrl' (open once).
Visible for local viewers.
No clients are logged on.
```

7.2.2 Vrworld对象

MATLAB 虚拟现实工具箱关于虚拟世界的 Vrworld 对象的操作方法有 12 种，表 7-2 列出了这些对象的 12 种操作方法。

表 7-2 Vrworld 对象的操作方法

方 法	描 述
vrworld	产生一个与给定虚拟世界相关的新的 Vrworld 对象
close(w)	关闭一个虚拟世界对象
delete(w)	从内存中删除一个关闭的虚拟世界对象
edit(w)	在 VRML 编辑器中编辑关于虚拟世界对象的 VRML 文件
get(w)	获得现有的虚拟世界对象的属性
isvalid(w)	对于一个合法的虚拟世界对象返回 TRUE
nodes(w)	列出虚拟世界中的可用节点
open(w)	打开一个虚拟世界对象
reload(w)	重新加载改变之前的虚拟世界
save(w)	将一个虚拟世界存储到一个 VRML 文件中
set(w)	设置虚拟世界对象的属性
view(w)	在 VRML 浏览器中观察一个虚拟世界对象

1. vrworld函数

此函数为产生一个与给定虚拟世界相关的新的 Vrworld 对象。其调用格式如下：

```
myworld = vrworld('filename')
myworld = vrworld('filename', 'new')
myworld = vrworld
myworld = vrworld([])
```

其中，filename 为虚拟世界的 VRML 文件名字；new 为创建虚拟世界新的对象。用 Vrworld 对象标记一个虚拟世界，影响虚拟世界的所有功能都集中反映在 Vrworld 对象上。如果指定的虚拟世界已经存在于内存中，那么新的 Vrworld 对象将与已经存在的虚拟世界建立起联系。如果虚拟世界不存在，将会产生一个新的 Vrworld 对象，最后产生的虚拟世界处于关闭状态，在使用之前必须将其打开。

2. open函数

此函数为打开指定的虚拟世界对象。调用格式如下：

```
open('name')
```

其中，**name** 为虚拟世界的 **Vrworld** 对象。此方法使得虚拟对象从关闭状态变为打开状态。虚拟世界在它使用之前必须是打开的。虚拟世界可以通过 **close** 函数进行关闭。

open 可以多次调用，并且相同次数的 **close** 调用才能使得虚拟世界恢复到关闭的状态。

例如，首先建立一个与虚拟世界 **vrplanets.wrl** 相关联的 **Vrworld** 对象，并打开它。

```
>> world=vrworld('vrplanets.wrl')
world =
vrworld object: 1-by-1
(E:/MATLAB2009a/toolbox/sl3d/sl3ddemos/vrplanets.wrl)
>> open(world)
```

3. iev函数

此函数为在外部的网络浏览器中观察虚拟世界对象。其调用格式如下：

```
view(w)
```

其中，**w** 为虚拟世界的 **Vroworld** 对象。此语法为打开一个主机默认配置的网络浏览器，并且将其指向包含虚拟世界的网页。如果虚拟世界不能观察，那么 **view** 命令什么也不做。

例如，接着以上例子，对打开的虚拟世界 **vrplanets.wrl** 进行观察，代码如下：

```
>> view(world)
ans =
vrfigure object: 1-by-1
Planets
```

运行以上代码，可以看到图 7-8 所示的虚拟场景。



图 7-8 通过 **view** 看到的虚拟场景

4. edit函数

此函数为在外部 VRML 编辑器中打开和编辑一个给定的虚拟世界对象。其调用格式如下：

```
edit(w)
```

其中, w 为一个虚拟世界对象。此语法为打开在外部编辑器中与给定的虚拟世界相对应的 VRML 文件。

如果在外部编辑器中修改虚拟世界, 那么将会把这些改变直接写到虚拟现实文件中。所以, 当用户重新加载该虚拟世界时, 将会显示改变后的结果。另一方面, 如果打开的虚拟世界在 MATLAB 中存储, 那么它将取代修改的 VRML 文件, 在编辑器中的修改将会丢失。

例如, 接上上面的例子, 在虚拟世界对象 `world` 已打开的前提下调用 `edit` 命令:

```
>> edit(world)
```

这时出现 VRML 编辑器, 如图 7-9 所示, 可以对此虚拟世界进行编辑、修改。

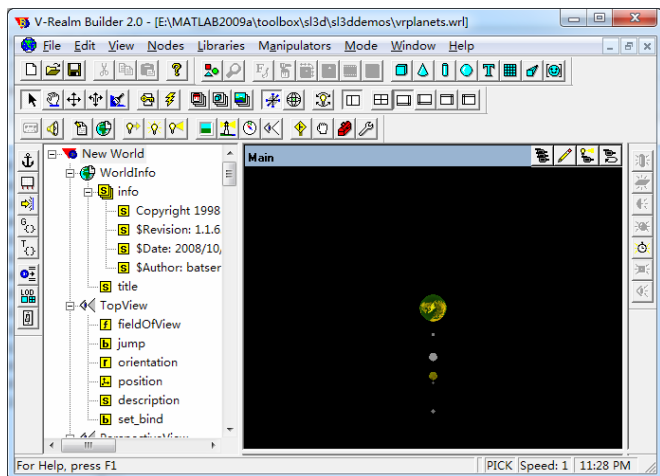


图 7-9 VRML 编辑界面

5. get函数

此函数为读取虚拟世界对象的属性。其调用格式如下:

```
get(w)
x=get(w, propname)
```

其中, w 为一个代表虚拟世界的 `Vrworld` 对象; `propname` 为虚拟世界属性的名字。虚拟世界有下列属性:

- (1) `FileName`: 装载虚拟世界的文件名字。
- (2) `Description`: 虚拟世界的描述, 它出现在主 Web 页上。
- (3) `Open`: 如果虚拟世界打开, 则为 “on”, 否则为 “off”。
- (4) `OpenCount`: 虚拟世界打开的次数, 如果是关闭的, 则为 0。
- (5) `View`: 如果虚拟世界可以被观察, 则其值为 “on”, 否则为 “off”。
- (6) `RemoteView`: 如果允许远程观察, 则其值为 “on”, 否则为 “off”。
- (7) `Id`: 一个自动产生的、唯一的标志符。
- (8) `Clients`: 观察此虚拟世界的顾客的数量。
- (9) `Nodes`: 虚拟世界所有节点的 `vnode` 对象的矢量。
- (10) `ClientUpdates`: 如果客户允许更新, 则其值为 “on”, 否则为 “off”。

接着上例, 对打开的虚拟世界运行如下代码:

```
>> get(world)
```

运行程序，输出如下：

```
Canvases = vr.canvas object: 0-by-0
Clients = 1
ClientUpdates = 'on'
Description = 'Planets'
Figures = vrfigure object: 1-by-1
FileName = 'E:/MATLAB2009a/toolbox/sl3d/sl3ddemos/vrplanets.wrl'
Nodes = vrnode object: 10-by-1
Open = 'on'
Record3D = 'off'
Record3DFileName = '%f_anim_%n.wrl'
Recording = 'off'
RecordMode = 'manual'
RecordInterval = [0 0]
RemoteView = 'off'
Time = 0
TimeSource = 'external'
View = 'on'
```

下面的代码表示一直等到所有的客户关闭与虚拟世界连接时才结束。

```
>> while get(world,'Clients')>0
    pause(0.1);
end
```

6. set函数

此函数为设置虚拟世界对象的属性。其调用格式如下：

```
set(w,propname,propvalue)
```

其中，**w** 表示虚拟世界的 **Vrworld** 对象；**propname** 为属性的名字；**propvalue** 为属性的新值。

7. isvalid函数

如果 **Vrworld** 对象是合法的，此函数则返回“1”；否则返回“0”。其调用格式如下：

```
x=isvalid(w)
```

其中，**w** 为虚拟世界的 **Vrworld** 对象。本函数为检测指定虚拟世界是否为合法的，返回一个向量。对于合法虚拟世界，此向量包含 1,对于不合法虚拟世界，则包含 0。

例如，对上例打开的虚拟世界 **vrplanets.wrl** 运行 **isvalid**：

```
>> isvalid(world)
ans =
    1
```

注意：如果结果为 1，则证明虚拟世界是打开的；如果为 0，则表明虚拟世界是关闭的。

8. nodes函数

此函数为列出在指定虚拟世界中的可用节点。其调用格式如下：

```
nodes(w,)  
nodes(w, '-full')
```

其中，`w` 为虚拟世界的 `Vrworld` 对象；`-full` 开关可以用来获得一个更详细的清单。此调用格式不仅可以列出节点，而且还可以通过列表方式显示它们的域。注意，此开关只影响显示输出，而对节点本身没有影响。

例如，在以上打开 `vrplanets.wrl` 虚拟世界前提下，输入如下代码：

```
>> nodes(world)
```

则列出此虚拟世界所有可用的节点如下：

```
SunTexture (ImageTexture) [Planets]  
Sun (Transform) [Planets]  
Mars (Transform) [Planets]  
EarthView (Viewpoint) [Planets]  
Earth (Transform) [Planets]  
Venus (Transform) [Planets]  
Mercury (Transform) [Planets]  
Moon (Transform) [Planets]  
PerspectiveView (Viewpoint) [Planets]  
TopView (Viewpoint) [Planets]
```

而在 `nodes` 函数中加上一个开关参数 ‘`-full`’，则输出如下：

```
>> nodes(world,'-full')  
Field                Access                Type                Sync  
-----  
SunTexture (ImageTexture) [Planets]  
  url                exposedField        MFString            off  
  repeatS            field              SFBool              off  
  repeatT            field              SFBool              off  
  
Sun (Transform) [Planets]  
  addChildren        eventIn            MFNode              off  
  removeChildren     eventIn            MFNode              off  
  children            exposedField        MFNode              off  
  center              exposedField        SFVec3f             off  
  rotation            exposedField        SFRotation          off  
  scale              exposedField        SFVec3f             off  
  scaleOrientation    exposedField        SFRotation          off  
  translation         exposedField        SFVec3f             off  
  bboxCenter          field              SFVec3f             off  
  bboxSize            field              SFVec3f             off  
  
Mars (Transform) [Planets]  
  addChildren        eventIn            MFNode              ff  
  removeChildren     eventIn            MFNode              off
```

children	exposedField	MFNode	off
center	exposedField	SFVec3f	off
rotation	exposedField	SFRotation	off
scale	exposedField	SFVec3f	off
scaleOrientation	exposedField	SFRotation	off
translation	exposedField	SFVec3f	off
bboxCenter	field	SFVec3f	off
bboxSize	field	SFVec3f	off

EarthView (Viewpoint) [Planets]

set_bind	eventIn	SFBool	off
fieldOfView	exposedField	SFFloat	off
jump	exposedField	SFBool	off
orientation	exposedField	SFRotation	off
position	exposedField	SFVec3f	off
description	field	SFString	off
bindTime	eventOut	SFTime	off
isBound	eventOut	SFBool	off

Earth (Transform) [Planets]

addChildren	eventIn	MFNode	off
removeChildren	eventIn	MFNode	off
children	exposedField	MFNode	off
center	exposedField	SFVec3f	off
rotation	exposedField	SFRotation	off
scale	exposedField	SFVec3f	off
scaleOrientation	exposedField	SFRotation	off
translation	exposedField	SFVec3f	off
bboxCenter	field	SFVec3f	off
bboxSize	field	SFVec3f	off

Venus (Transform) [Planets]

addChildren	eventIn	MFNode	off
removeChildren	eventIn	MFNode	off
children	exposedField	MFNode	off
center	exposedField	SFVec3f	off
rotation	exposedField	SFRotation	off
scale	exposedField	SFVec3f	off
scaleOrientation	exposedField	SFRotation	off
translation	exposedField	SFVec3f	off
bboxCenter	field	SFVec3f	off
bboxSize	field	SFVec3f	off

Mercury (Transform) [Planets]

addChildren	eventIn	MFNode	off
removeChildren	eventIn	MFNode	off

children	exposedField	MFNode	off
center	exposedField	SFVec3f	off
rotation	exposedField	SFRotation	off
scale	exposedField	SFVec3f	off
scaleOrientation	exposedField	SFRotation	off
translation	exposedField	SFVec3f	off
bboxCenter	field	SFVec3f	off
bboxSize	field	SFVec3f	off
Moon (Transform) [Planets]			
addChildren	eventIn	MFNode	off
removeChildren	eventIn	MFNode	off
children	exposedField	MFNode	off
center	exposedField	SFVec3f	off
rotation	exposedField	SFRotation	off
scale	exposedField	SFVec3f	off
scaleOrientation	exposedField	SFRotation	off
translation	exposedField	SFVec3f	off
bboxCenter	field	SFVec3f	off
bboxSize	field	SFVec3f	off
PerspectiveView (Viewpoint) [Planets]			
set_bind	eventIn	SFBool	off
fieldOfView	exposedField	SFFloat	off
jump	exposedField	SFBool	off
orientation	exposedField	SFRotation	off
position	exposedField	SFVec3f	off
description	field	SFString	off
bindTime	eventOut	SFTime	off
isBound	eventOut	SFBool	off
TopView (Viewpoint) [Planets]			
set_bind	eventIn	SFBool	off
fieldOfView	exposedField	SFFloat	off
jump	exposedField	SFBool	off
orientation	exposedField	SFRotation	off
position	exposedField	SFVec3f	off
description	field	SFString	off
bindTime	eventOut	SFTime	off
isBound	eventOut	SFBool	off

9. reload函数

此函数为从相关文件重新加载指定的虚拟世界。其调用格式如下：

```
reload(w)
```

其中，w 为虚拟世界的一个 Vrworld 对象。此语句在虚拟世界的相关文件中重装虚拟世界

的内容，同时也使得所有客户当前观察的虚拟世界为重新加载的虚拟世界。这个方法对于虚拟世界的显示内容是非常有用的。

10. save函数

此函数将指定的虚拟世界存储到一个 VRML 文件中。其调用格式如下：

```
save(w)
save(w, filename)
```

其中，w 为虚拟世界的 Vfworld 对象；filename 为要存储的虚拟的文件名。如果没有指定文件名，将使用当前虚拟对象对应的 VRML 文件。

11. close函数

此函数为关闭指定的虚拟世界对象。其调用格式如下：

```
close(w)
```

其中，w 为一个虚拟世界对象。此方法将虚拟世界从打开状态变为关闭状态。如果虚拟世界打开的次数多于一次，那么将调用相应次数的 close，从而确保虚拟世界是关闭的。

打开和关闭虚拟世界是内存管理的机制。当虚拟世界关闭时，系统可以得到更多的内存，相应的 open 命令将在硬盘加载虚拟世界。

一般地，当打开的虚拟世界不需要用时应当关闭，以便释放它占用的内存空间。当关闭虚拟世界时，它就成为不合法的了，必要时用户还可以再次将其打开。

12. delete函数

此函数为从内存中删除指定的虚拟世界对象。其调用格式如下：

```
delete(w)
```

其中，w 为一个虚拟世界对象。此语法为从内存中删除世界对象，使得所有与虚拟世界对象相关存在的虚拟世界对象无效。虚拟世界对象在删除之前必须被关闭。

例如，接着上面的例子，使用完 world 虚拟世界时，可以将它关闭并且删除。

```
>> close(world)
>>delete(world)
```

7.2.3 Vrnode对象

MATLAB 虚拟现实工具箱的 Vrnode 对象的操作方法共有 6 种，见表 7-3。

表 7-3 Vrnode 对象的操作方法

方 法	描 述
vrnode	产生一个新的虚拟节点对象
fields	列出属于一个节点的 VRML 域
get	得到虚拟节点对象的属性
isvalid	对于合法的虚拟节点对象返回“TRUE”
sync	与远程计算机同步一个 VRML 域
set	设置虚拟节点对象的属性

1. vrnode函数

此函数为产生一个新的 Vnode 对象。其调用格式如下：

```
n=vrnode(w,nodename)
```

其中，w 为虚拟世界的 Vrworld 对象；nodename 为节点的名字。这个函数建立一个虚拟节点 Vnode 对象，这有点类似于程序设计中的句柄（handle）操作，所有与虚拟节点相关的函数都需要先建立一个 Vnode 对象。产生一个 Vnode 对象就得到了访问这个节点的指针。当更多的 Vnode 对象在同一虚拟世界和同样的节点名字中产生时，它们都指向同一个节点。此节点不能复制。

例如，首先打开虚拟世界 vrplanets.wrl：

```
>> world=vrworld('vrplanets.wrl')
world =
vrworld object: 1-by-1
(E:/MATLAB2009a/toolbox/sl3d/sl3ddemos/vrplanets.wrl)
>> open(world)
```

然后列出此虚拟世界所有可用的节点：

```
>> nodes(world)
SunTexture (ImageTexture) [Planets]
Sun (Transform) [Planets]
Mars (Transform) [Planets]
EarthView (Viewpoint) [Planets]
Earth (Transform) [Planets]
Venus (Transform) [Planets]
Mercury (Transform) [Planets]
Moon (Transform) [Planets]
PerspectiveView (Viewpoint) [Planets]
TopView (Viewpoint) [Planets]
```

接着建立一个 node 对象，其节点名为 Earth：

```
>> node=vrnode(world,'Earth')
node =
vrnode object: 1-by-1
Earth (Transform) [Planets]
```

2. fields函数

此方法返回一个节点的 VRML 域的信息。其调用格式如下：

```
fields(N)
x=fields(N)
```

其中，N 为一个虚拟节点 Vnode 对象。如果给定输出参数，函数将返回一个结构，此结构包含每个节点的 VRML 域。参数包括如下成员：

- (1) Type: VRML 域类型的名字，如 ‘MFString’、‘SFColor’。
- (2) Access: 可达性的描述，如 ‘eventIn’、‘exposedField’。
- (3) Sync: 同步状态，它的值是 on 或 off。

如果没有指定输出参数，屏幕上也会输出同样的格式化信息。
例如，接着上面例子，列出节点 Earth 的 VRML 域结构：

>> fields(node)

Field	Access	Type	Sync

addChildren	eventIn	MFNode	off
removeChildren	eventIn	MFNode	off
children	exposedField	MFNode	off
center	exposedField	SFVec3f	off
rotation	exposedField	SFRotation	off
scale	exposedField	SFVec3f	off
scaleOrientation	exposedField	SFRotation	off
translation	exposedField	SFVec3f	off
bboxCenter	field	SFVec3f	off
bboxSize	field	SFVec3f	off

3. get函数

读取一个虚拟节点属性值或一个 VRML 域的值。其调用格式如下：

```
x=get(N, propname)
x=get(N, fieldname)
```

其中，N 为一个 Vnode 节点对象；propname 为读取属性的名字；fieldname 为读取 VRML 域的名字。如果只给出唯一的一个 VRML 对象参数，那么此函数将列出所有属性和 VRML 域；如果给出一个属性的名字，则只返回此属性的值。

虚拟节点具有以下属性：

- (1) Name: 节点的名字。
- (2) Type: 节点类型（如 ‘Transform’、‘Shape’）。
- (3) Fields: 节点的 VRML 域名字的一个单元数组。

例如，接着上面已打开 Earth 节点示例，列出它的所有属性和 VRML 域：

```
>> get(node)
Fields = cell array: 10-by-1
Name = 'Earth'
Type = 'Transform'
World = vrworld object: 1-by-1
```

4. isvalid函数

此函数为如果 Vnode 对象是合法的，则返回 “1”；否则返回 “0”。其调用格式如下：

```
x=isvalid(nodes)
```

其中，nodes 为节点的 Vnode 对象的一维数组。本语句检测指定节点是否合法的，并返回一个向量。对于合法节点此向量包含 1，对于不合法节点包含 0。

当出现下面 3 种情况时，节点将被视为合法。

- (1) 节点的父虚拟世界仍然存在。
- (2) 节点的父虚拟世界是打开的。

(3) 节点还存在于父虚拟世界中。

5. sync函数

此函数为标识 VRML 域与客户端是否同步。其调用格式如下：

```
sync(N, fieldname, action)
```

其中, N 为 Vnode 节点对象; fieldname 为将要改变的 VRML 域的名字; action 为其属性。当标记为同步时, 客户端将向服务器端通报域改变的值 (如通过一个用户的行为); 如果域不同步, 客户端信息的改变将被忽略。域同步给网络增加了更多的通信量, 这是因为域的值必须等客户端提交多次后才能确定。因此, 只有真正需要用户浏览的域才应该标记为同步。一般地, 所有域的默认值是不同步的。

6. set函数

此函数为设置虚拟节点属性或 VRML 域的值。其调用格式如下：

```
set(N)
x=set(N, propname)
x=set(N, fieldname)
```

其中, N 为 Vnode 节点对象; propname 为将要读取的属性名字; fieldname 为将要读取的 VRML 域的名字。如果只给出 Vmod 对象这样唯一的参数, 则屏幕上将输出所有设置的 VRML 域和其他值类型。因为所有的节点属性是只读的, 所以此方法只用于写值到 VRML 域中。当设置一个 double 向量值时, 所有行向量和列向量都可接受该值。

7.3 Simulink的应用接口

虚拟现实工具箱可以在 MATLAB 接口和 Simulink 接口两种环境中运行, 而 Simulink 接口可能是更合适的工作方式, 因为它更直接、更容易使用, 并且很容易通过图形用户界面来进行交互。7.2 节学习了基于 MATLAB 接口的虚拟现实工具箱的函数和方法, 这一节将通过使用 Simulink 连接虚拟世界和一个利用 Simulink 演示虚拟世界例子来学习 Simulink 在虚拟世界中的应用。

7.3.1 使用Simulink连接虚拟世界

使用虚拟现实工具箱可以建立起与虚拟世界有关的一个 Simulink 模型。下面的例子将解释如何在本地计算机上通过虚拟现实工具箱使用 Simulink 连接虚拟世界, 并对该虚拟世界进行模拟。

1. 建立虚拟现实工具箱的连接

模拟一个 Simulink 模型产生动态系统的信号数据, 通过连接 Simulink 模型到一个虚拟世界, 就可以利用此数据控制和操纵相应的虚拟世界。

当产生一个虚拟世界和一个 Simulink 模型后, 就可以通过虚拟现实工具箱将它们二者连接起来。现在来模拟虚拟世界中的一架飞机。

(1) 在 MATLAB 命令窗口, 输入:

```
>> vrtut2
```

就可以看到 Simulink 模型被打开了, 如图 7-10 所示。

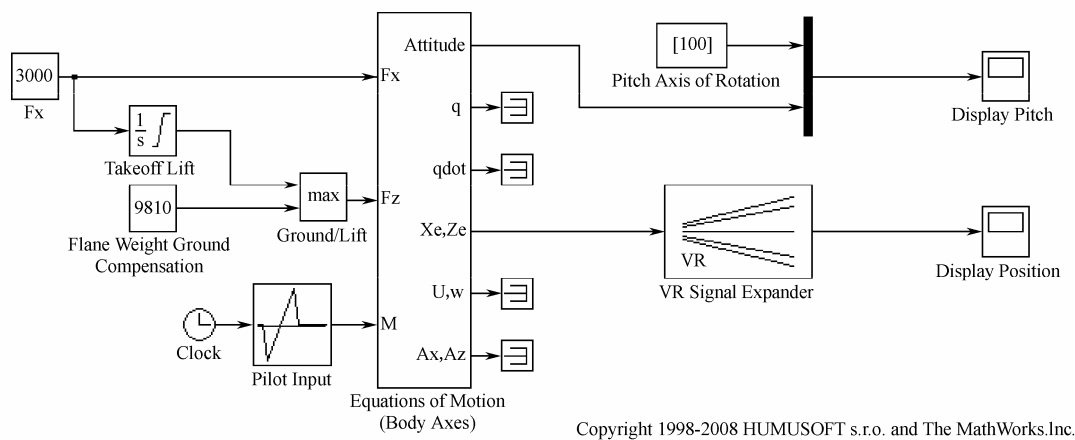


图 7-10 Vrtut2 的 Simulink 模型窗口

(2) 单击【Simulation】菜单下的【Normal】选项，并且用鼠标左键单击“Start”。在窗口范围内观察模拟的结果如图 7-11 所示。

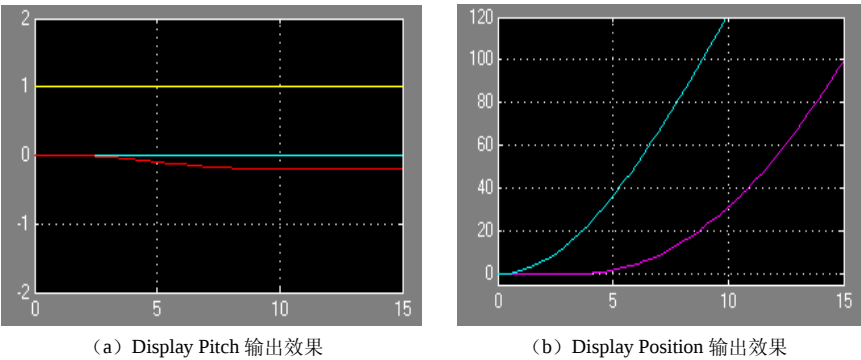


图 7-11 Vrtut2 输出效果

(3) 在 MATLAB 命令窗口输入：

```
>> vrlib
```

打开虚拟现实工具箱，如图 7-5 所示。

(4) 从库的窗口中用鼠标拖动 VR Sink 模块到 Simulink 图表上，这时关闭 vrlib 窗口。现在选择一个虚拟世界用于模拟，VRML 文件 vrtkoff.wrl 是一个简单的具有跑道和飞机的虚拟世界。

(5) 在 Simulink 模块内，用鼠标左键双击标记为 VR Sink 的模块。“Parameters:VR Sink”对话框被打开，如图 7-12 所示。

(6) 在描述窗口中，输入模型的一个简短描述。在网页浏览器内此描述内容将出现在可用的虚拟世界列表中。例如，输入“VR Plane take-off”。

(7) 用鼠标左键单击“Browse”按钮，这时可选择的虚拟世界对话框打开，找到目录\ MATLAB2009a\toolbox\sl3d\sl3ddemos\，选择文件 vrtkoff.wrl，并且用鼠标左键单击“Open”按钮。

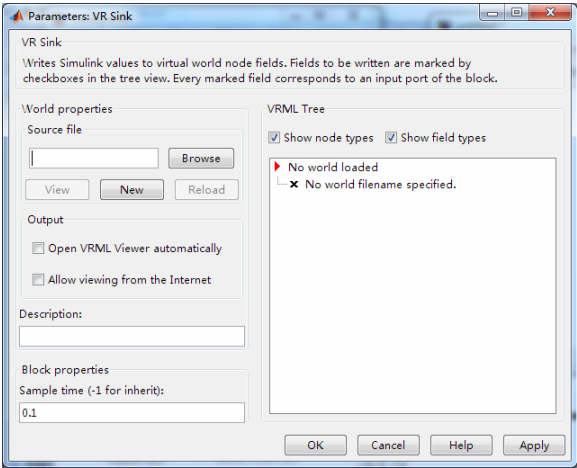


图 7-12 “Parameters:VR Sink” 对话框

(8) 在模块参数 “Parameters:VR Sink” 对话框内，单击 “Apply” 按钮。这时一个 VRML 目录树出现在对话框的右边，如图 7-13 所示，目录树包含有相关的虚拟现实景的结构。

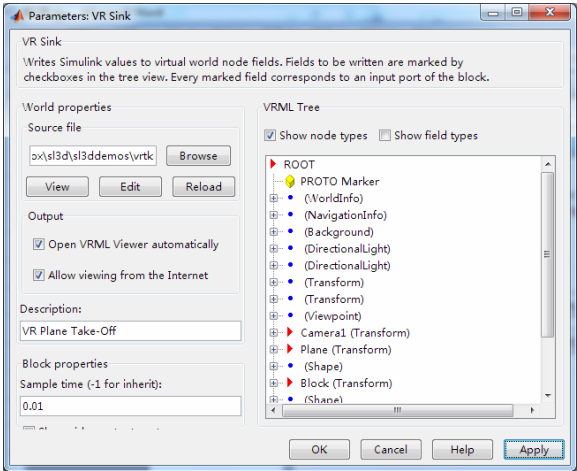


图 7-13 VRML 目录树

(9) 在飞机转换节点的左边，单击 “+” 标记，那么飞机转换树目录展开。现在可以看到飞机的特征，并从 Simulink 得到控制，在模型中计算飞机的位置和仰角。

(10) 在飞机的转变树中，单击平移和旋转区域，则这些选择的区域被标记为控制标记点，这些区域代表飞机的位置（平移）和仰角（旋转）。

(11) 单击 “OK” 按钮。在 Simulink 图表中，VR Sink 模块通过两个输入参数来更新，如图 7-14 所示。

第一个输入是飞机的旋转参数，它通过一个四元素矢量来定义，其中，前三个元素定义旋转的坐标轴，如[1 0 0]。飞机的倾斜度通过相对 X 轴旋转的角度来表达，另一个元素是围绕 X 轴旋转的角度，用弧度表示。

在 Simulink 模块中，旋转的倾斜轴矢量与飞机的旋转输入联系起来。第二个输入是飞机的平移，它表达了虚拟世界中飞机的位置，此位置由三个坐标 X、Y、Z 来

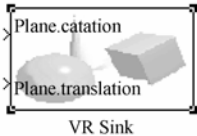


图 7-14 VR Sink 模块

标识，相关的矢量必须有三个元素。在这个例子中，飞机的跑道在 X-Z 平面，Z 轴和 Y 轴定义了飞机的海拔高度。

(12) 在 Simulink 模块中，把 VR 信号放大器同飞机的平移输入联系起来。连接信号之后，整个模型图效果如图 7-15 所示。

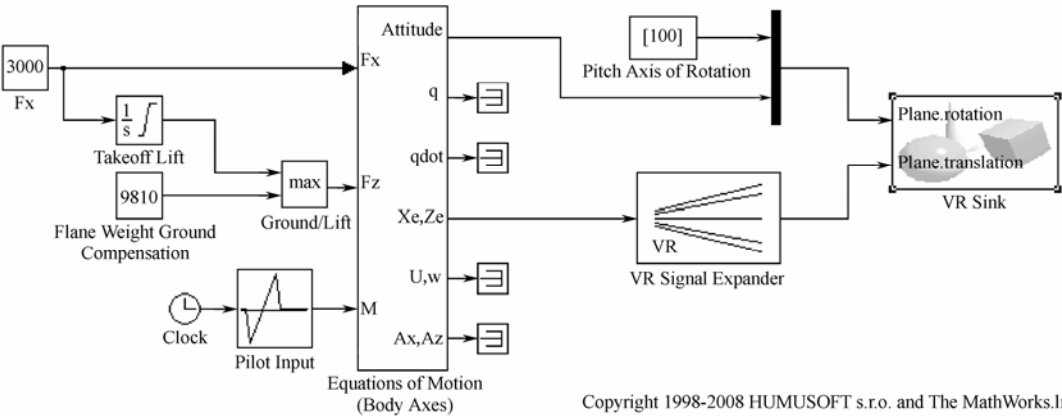


图 7-15 整个模型图效果

虚拟世界的自由度，依赖于与之相关的 VRML 域格式和不同的请求输入矢量大小，如果相关的信号矢量大小与相连接的 VRML 域大小不匹配，那么当开始模块时系统就会提示一个出错信息。

“Incorrect input vector size”

(13) 单击【Simulation】菜单下的【Start】选项，飞机就开始从跑道的右边飞入左边的空中，如图 7-16 所示。

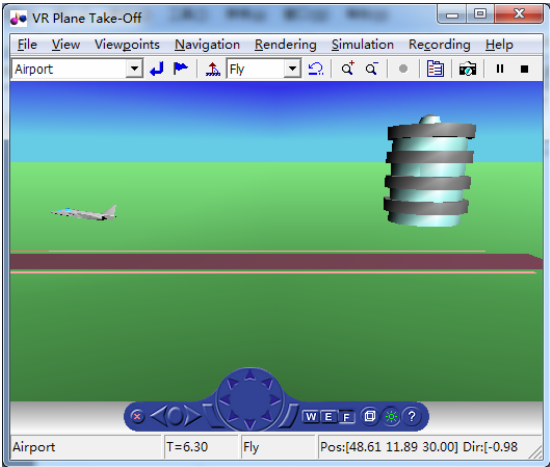


图 7-16 飞机跑道起飞的模拟结果

2. 改变虚拟现实模块的连接

有时候可能想利用 Simulink 来模拟连接一个不同的虚拟世界或连接不同的信号。当使用 Simulink 模拟连接了一个虚拟世界后，还可以选择另一个虚拟世界或改变连接到虚拟世界的信号。下面的例子模拟了飞机的飞行和在虚拟环境中交互性控制的效果。

(1) 在模拟环境中,用鼠标左键双击标记为 VR Sink 的模块。“Parameters:VR Sink”对话框被打开,如图 7-12 所示。

(2) 用鼠标左键双击飞机节点。

(3) 用鼠标左键单击“Browse”按钮,这时可选择的虚拟世界对话框打开,找到目录\MATLAB2009a\toolbox\sl3d\sl3ddemos/,选择文件 vrtkoff2.wrl,并且用鼠标左键单击“Open”按钮。

(4) 在模块参数“Parameters:VR Sink”对话框内,单击“Apply”按钮。这时一个 VRML 目录树出现在对话框的右边,Simulink 连接一个新的虚拟世界用于模拟。

(5) 在飞机转换节点的左边,单击“+”标记,那么飞机转换树目录展开。现在可以看到飞机的特征,并从 Simulink 得到控制,在模型中计算飞机的位置。

(6) 在飞机的转变树中,单击“转换区域”检查框,再按“OK”按钮,VR Sink 块进行更新,并且改变成只有一个输入,即飞机的平移运动。虚拟现实模块开始准备使用新定义的属性进行模拟。

诸如像改变特征、改变虚拟世界、删除虚拟现实模块或关闭模型的行为,将关闭在 Simulink 和浏览器间建立连接或使之无效,浏览器中将显示出主虚拟现实工具箱的 HTML 页,上面带有当前可用的虚拟世界的列表。如果想再观察一遍虚拟世界,可在列表中单击它的描述。

7.3.2 一个虚拟世界例子

可以观察一个与 Simulink 模块图表相连接的虚拟世界,也可以观察一个通过 Simulink 改变参数的虚拟世界。

1. 演示一个虚拟世界并且进行模拟

这个例子说明了如何在本地计算机上演示模拟虚拟世界的方法。在 Simulink 窗口模拟一个简单的汽车运动,在虚拟世界中可以看到汽车的位置和角度的变化。

(1) 在 MATLAB 命令窗口中输入:

```
>> vrtut1
```

这时,Simulink 窗口打开,里面有一个汽车的模型,如图 7-17 所示。

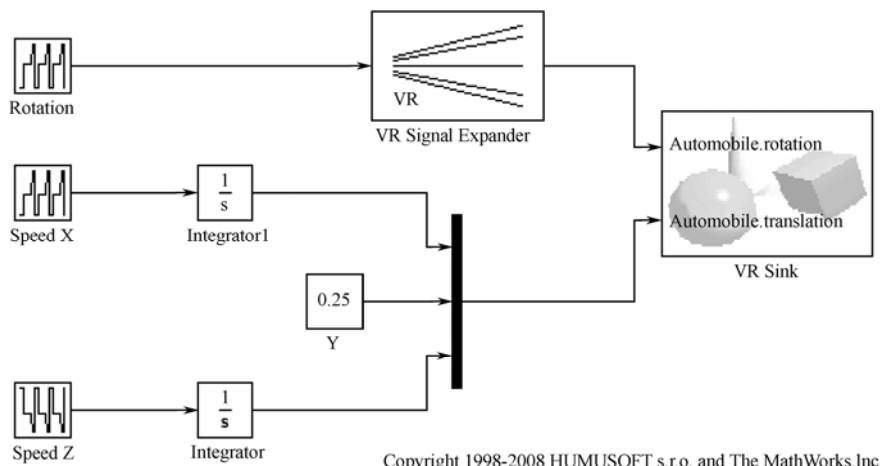


图 7-17 汽车模型

(2) 双击标记为 VR Sink 的模块，于是“Parameters:VR Sink”对话框打开，如图 7-18 所示。

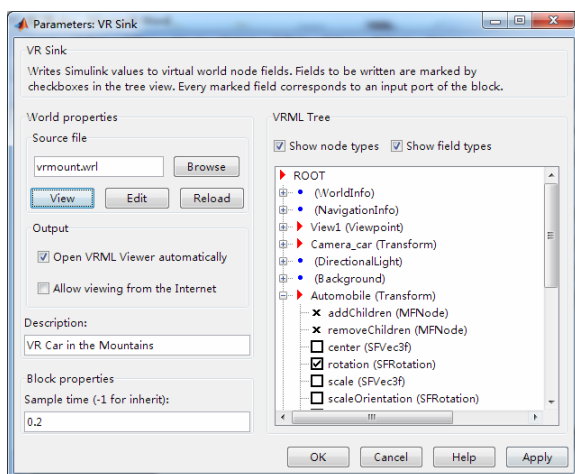


图 7-18 “Parameters:VR Sink”对话框

(3) 单击“View”按钮。这时带有三维模型的虚拟世界的浏览器打开。

(4) 在屏幕上合理安排浏览器和 Simulink 窗口，使它们能够同时被看到。

(5) 在 Simulink 窗口中，单击【Simulation】菜单下的【Start】选项，这时模拟过程开始，在浏览器中可以看到一辆汽车行驶在山中的路上，如图 7-19 所示。

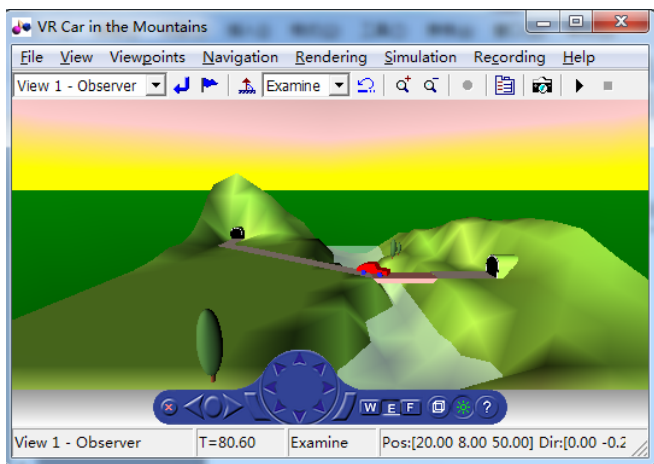


图 7-19 汽车在山路运动的模拟场景

2. 使用浏览器进行本地观察

在 Web 浏览器中演示一个虚拟世界，允许在本地计算机上进行人-机交互。可以通过“Parameters:VR Sink”对话框中的“View”按钮访问一个虚拟世界。但有时也可以在本地配置中使用虚拟现实工具箱 HTTP 服务器访问一个虚拟世界，此服务器一般给远程计算机提供虚拟世界。使用这种方法在本地机上观察虚拟世界的理由是：

- 在计算机上如果有多于一个的浏览器，则可能使用其中的一个浏览器而不一定是默认的浏览器。

- 这种访问虚拟世界的方法对于使用 Simulink 或 MATLAB 产生虚拟对象的效果是一样的。
- 当虚拟世界存在的情况下，如果在使用标准观察方法不可能或不方便时，仍可以访问虚拟世界对象。

下面的例子解释了如何在本地主计算机上演示一个模拟的虚拟世界。Simulink 窗口中有一辆运动的汽车。汽车的位置和角度可以通过人-机交互方式来进行控制。

(1) 在 MATLAB 命令窗口中输入：

```
>> vrtut1
```

这时，Simulink 窗口打开，里面有一个汽车的模型，如图 7-17 所示。

(2) 改变 HTTP 端口号的方法是用鼠标双击 VR Sink 块，在参数对话框内选择“show video output port”，单击“Set up and preview video output”按钮，弹出如图 7-20 所示的窗口。

(3) 在屏幕上安排好浏览器和 Simulink 窗口，以便能同时看到它们。

(4) 在 Simulink 窗口中，单击【Simulation】菜单下的【Start】选项，这时模拟开始，在浏览器窗口中一辆汽车行驶在山中的路上。

(5) 虚拟世界中使用浏览器控制汽车的运动。

在 Simulink 窗口中，单击【Simulation】菜单下的【Stop】选项，关闭 Web 浏览器窗口。

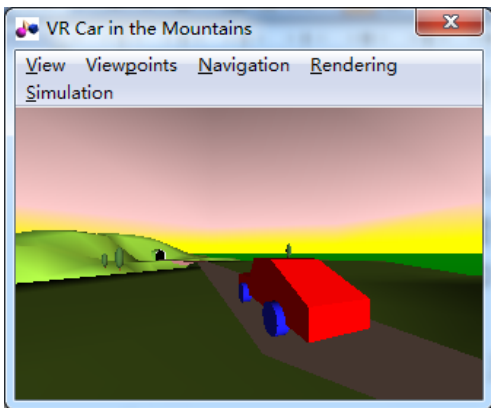


图 7-20 汽车在山中运动的模拟场景

7.4 MATLAB接口中虚拟现实的应用

7.4.1 热传送的虚拟现实

在这个例子中演示了在 MATLAB 和虚拟现实之间矩阵数据的传递，从中可以看到颜色的改变和形状的变化，这对于观察不同物理过程的虚拟化是非常有用的。

在一个 L 形的金属导体中使用预先计算的基于时间的温度数据，将这些数据一步步地送到虚拟现实中，这就导致了金属体在每一动画帧的颜色的改变。

下面按步骤地完成这一模拟过程，每一个命令将显示在 MATLAB 的命令行中，当按下“Enter”键后命令将执行。

(1) 首先装入预先计算好的数据文件 `vrheat.mat`:

```
>> load 'vrheat.mat'
```

(2) L 形的几何体块存储在 `lblock` 结构中, 为了可视化, 此块被分解成三角形的小平面。小平面的顶点坐标存储在 `lblock.mesh.p` 域中, 三角形的边通过索引描述成顶点数组。

在命令窗口输入如下代码:

```
>> lblock.mesh
```

运行程序, 输出如下:

```
ans =  
    e: [3x400 double]  
    p: [3x262 double]
```

(3) 由于坐标是以矩阵形式存放的, 而 VRML 的顶点数组是一个简单的矢量形式, 所以需要将顶点数组转化成坐标矩阵的形式。

```
>> vert='lblock.mesh.p';  
>> v=vert;  
>> v=v(:);
```

(4) VRML 中的一系列小平面被定义成一个简单的顶点矢量, 这里顶点乘以-1, 所以需要适当地转换顶点数组。代码如下:

```
>> facets=lblock.mesh.e(1:3,:)-1;  
>> facets(4,:)= -1;  
>> f=facets;  
>> f=f(:);
```

(5) 准备代表不同温度级别的颜色图。MATLAB 内嵌的 `jet` 颜色图就是为此目的设计的。

```
>> cmap=jet(192);
```

(6) 因为 VRML 需要的颜色图是以单矢量形式存在的, 所以需要转换颜色图矩形。

```
>> cm=cmap;  
>> cm=cm(:);
```

`lblock.sol.u` 域包含了一个描述温度随着时间变化的矩阵。对于 262 个顶点有 41 个 (从 1 开始) 预先算好的相位。

```
>> lblock.sol
```

输出如下:

```
ans =  
    u: [262x41 double]  
    tlist: [1x41 double]
```

(7) 现在需要依比例设定温度, 使得这些温度值可以映射到颜色图中。

```
>> u=lblock.sol.u;
```

```
>> ucolor=(u-repmat(min(u),size(u,1),1)).*(size(cmap,1)-1);
>> urange=max(u)-min(u);
>> urange(find(urange==0))=1;
>> ucolor=round(ucolor./repmat(urange,size(u,1),1));
```

(8) 下面计算第一个动画帧, 准备开始模拟。

```
>> colind=zeros(size(facets));
>> colind(:,4)=-1;
>> ci=colind';
>> ci=ci(:);
```

(9) 将准备好的三维数据文件 `vrheat.wrl` (其源代码将在本小节后面列出) 装入这个虚拟世界, 并且打开它。

```
>> world=vrworld('vrheat.wrl');
>> open(world);
```

(10) 为所有将要改变的 VRML 节点得到 `Vrworld` 对象。由于它们是普通的 MATLAB 对象, 所以可以把它们安排到一个结构中。

```
>> nh.IFS=vrnode(world,'IFS');
>> nh.IFS_Coords=vrnode(world,'IFS_Coords');
>> nh.IFS_Colormap=vrnode(world,'IFS_Colormap');
>> nh.TEXT=vrnode(world,'TEXT');
```

(11) 开始观察虚拟世界。

```
>> view(world);
while get(world,'Clients')==0;
    pause(0.1);
end
```

在浏览器中观察到图 7-21 所示的效果。

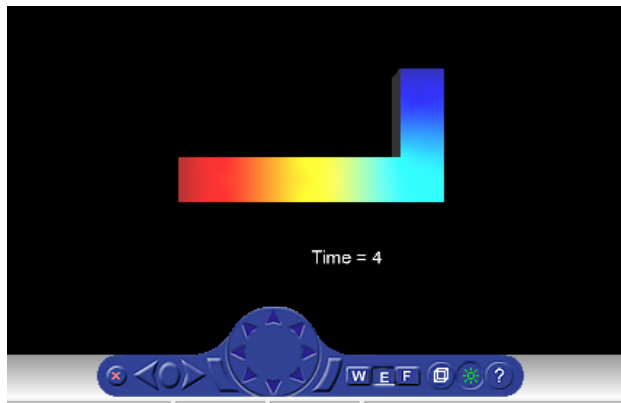


图 7-21 热传递的虚拟现实情况 1

可以通过控制面板上的按钮, 控制物体的运动。如控制物体运动到某种状态, 如图 7-22 所示。



图 7-22 热传递的虚拟现实情况 2

热传送的 VRML 文件 vrheat.wrl 源程序代码如下：

```
#VRML V2.0 utf8

WorldInfo {
  title "VR Heat Transfer"
  info [
    "Copyright 1998-2008 HUMUSOFT s.r.o. and The MathWorks, Inc.",
    "$Revision: 1.1.6.1 $",
    "$Date: 2008/10/31 07:12:29 $",
    "$Author: batserve $"
  ]
}

NavigationInfo {
  headlight TRUE
  type "EXAMINE"
}

Viewpoint {
  position 0 0 10
  description "View 1"
}

Transform {
  children Shape {
    appearance Appearance {
      material Material {
        diffuseColor 0.3 0.3 0.3
        emissiveColor 0.2 0.2 0.2
      }
    }
  }
  geometry DEF IFS IndexedFaceSet {
    ccw FALSE
    colorIndex [
```

```
    0 1 2 3 -1
    4 5 1 0 -1
    6 7 5 4 -1
    3 2 7 6 -1
    3 6 4 0 -1
    2 1 5 7 -1
]
colorPerVertex TRUE
convex TRUE
coordIndex [
    0 1 2 3 -1
    4 5 1 0 -1
    6 7 5 4 -1
    3 2 7 6 -1
    3 6 4 0 -1
    2 1 5 7 -1
]
creaseAngle 0
normalPerVertex TRUE
solid FALSE
coord DEF IFS_Coords Coordinate {
    point [
        1 1 1
        1 -1 1
        -1 -1 1
        -1 1 1
        1 1 -1
        1 -1 -1
        -1 1 -1
        -1 -1 -1
    ]
}
color DEF IFS_Colormap Color {
    color [
        1 0 0
        1 0 1
        1 1 0
        1 1 1
        0 1 0
        0 0 1
        0 1 1
        0 0 0
    ]
}
normal NULL
texCoord NULL
```

```

    }
  }
}
Transform {
  translation 0 -2 0
  children Billboard {
    axisOfRotation 0 0 0
    children Shape {
      geometry DEF TEXT Text {
        string "This is a cube"
        fontStyle FontStyle {
          family "SANS"
          size 0.5
        }
      }
    }
  }
}
}
}
}

```

7.4.2 汽车在山中运动的模拟

在这个例子中将学习如何在虚拟世界中使用基于对象接口的 **MATLAB** 来控制一个物体，其操作步骤如下：

(1) 首先生成一个表示虚拟世界的 **Vrworld** 对象，它表示一个虚拟世界。建立虚拟世界的 **VRML** 文件预先通过包含在虚拟现实工具箱中的 **V-Realm** 的 **VRML** 构造器来生成，文件的名字是 **vrmount.wrl**，其源代码见本节后面。还可以设置对该对象的描述，这将有助于观察者确定此虚拟世界。打开该对象：

```

>> world=vrworld('vrmount.wrl','Car in the Mountains')
world =
vrworld object: 1-by-1
VR Car in the Mountains (E:/MATLAB2009a/toolbox/sl3d/sl3ddemos/vrmount.wrl)

```

(2) 虚拟世界在使用前必须打开，可以通过 **open** 函数来实现：

```

>> open(world)

```

(3) 虚拟世界可以在 **VRML** 浏览器（已经集成在网络浏览器中）中观察。通用的 **VRML** 浏览器是 **Blaxxun Contact**，在安装虚拟现实工具时它已经被安装到网络浏览器中了。浏览虚拟世界：

```

>> view(world)
ans =
vrfigure object: 1-by-1
VR Car in the Mountains

```

在浏览器看到图 7-23 所示的虚拟场景。

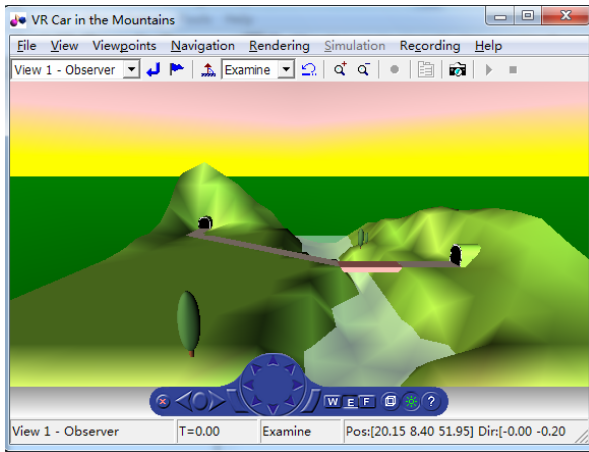


图 7-23 浏览器观察到的汽车在山中运动的场景

(4) 通过 `get` 函数来检查虚拟世界的属性。注意, ‘FileName’ 属性包含文件的名称, ‘Description’ 属性包含所设置的文件的描述。关于其他属性的相关描述, 读者可以在虚拟现实工具箱使用手册或在线帮助文档中找到。

```
>> get(world)
Canvases = vr.canvas object: 0-by-0
Clients = 1
ClientUpdates = 'on'
Description = 'VR Car in the Mountains'
Figures = vrfigure object: 1-by-1
FileName = 'E:/MATLAB2009a/toolbox/sl3d/sl3ddemos/vrmount.wrl'
Nodes = vrnode object: 13-by-1
Open = 'on'
Record3D = 'off'
Record3DFileName = '%f_anim_%n.wrl'
Recording = 'off'
RecordMode = 'manual'
RecordInterval = [0 0]
RemoteView = 'off'
Time = 0
TimeSource = 'external'
View = 'on'
```

(5) 虚拟世界中的所有元素都可以通过 VRML 节点来表示, 任何元素的行为也都可以通过改变节点的域来控制。

`nodes` 函数显示出在虚拟世界中可用节点的列表。

```
>> nodes(world)
Tunnel (Transform) [VR Car in the Mountains]
Road (Shape) [VR Car in the Mountains]
Bridge (Shape) [VR Car in the Mountains]
River (Shape) [VR Car in the Mountains]
```

```

ElevApp (Appearance) [VR Car in the Mountains]
Canal (Shape) [VR Car in the Mountains]
Wood (Group) [VR Car in the Mountains]
Tree1 (Group) [VR Car in the Mountains]
Wheel (Shape) [VR Car in the Mountains]
Automobile (Transform) [VR Car in the Mountains]
VPfollow (Viewpoint) [VR Car in the Mountains]
Camera_car (Transform) [VR Car in the Mountains]
View1 (Viewpoint) [VR Car in the Mountains]

```

(6) 在访问 VRML 节点之前, 必须产生一个 Vnode 对象, 节点通过它的名字和归属的虚拟世界类别来对它进行识别。将生成一个与 VRML 节点 ‘Automobile’ 相关联的 Vnode 对象, 它表示在路上行驶汽车的模型。如果在场景中看不到, 不要着急, 它隐藏在左边的隧道里。

```

>> car=vrnode(world,'Automobile')
car =
vrnode object: 1-by-1
Automobile (Transform) [VR Car in the Mountains]

```

(7) 给定节点的 VRML 域可以通过 get 函数来询问, 将看到在节点列表中有名为 ‘translation’ 和 ‘rotation’ 的域, 可以通过改变这些域来移动汽车。

```

>> get(car)
Fields = cell array: 10-by-1
Name = 'Automobile'
Type = 'Transform'
World = vrworld object: 1-by-1

```

(8) 现在准备设置汽车运动的坐标矢量, 通过在循环中设置坐标矢量值可以生成一个运动场景。

对于汽车运动的 3 个位置有对应的 3 组数据:

```

>> z1=0:12;
x1=3+zeros(size(z1));
y1=0.25+zeros(size(z1));

z2=12:26;
x2=3:1.4285:23;
y2=0.25+zeros(size(z2));

z3=23:43;
x3=26+zeros(size(z3));
y3=0.25+zeros(size(x3));

```

① 首先, 让汽车沿着第一次运动轨迹移动。汽车是通过设置 ‘Automobile’ 节点的 ‘translation’ 域来移动的。

```

>> for i=1:length(x1)
    set(car,'translation',[x1(i) y1(i) z1(i)]);

```

```
    pause(0.1);
end
```

② 接着，将稍微旋转一下汽车，使它驶上第二条运动轨迹。这可通过设置 ‘Automobile’ 节点的 ‘rotation’ 属性来完成。

```
>> set(car,'rotation',[0 1 0 -0.7]);
>> for i=1:length(x2)
    set(car,'translation',[x2(i) y2(i) z2(i)]);
    pause(0.1);
end
```

③ 最后，再次让汽车转向左边。

```
>> for i=1:length(x3)
    set(car,'translation',[x3(i) y3(i) z3(i)]);
    pause(0.1);
end
```

(9) 当使用完 Vrworld 对象后，有必要关闭并且删除它。通过 close 和 delete 函数完成这一任务。

```
>> close(world);
>> delete(world);
```

通过这一实例，让我们学会了虚拟现实工具箱的交互操作。如果你还意犹未尽，可以让虚拟世界处于打开状态，并且按照前面学过的相似命令移动汽车，或者访问其他节点和它们的域来进一步进行操作。

汽车在山中运动的 VRML 文件 vrmount.wrl 源程序代码如下：

```
#VRML V2.0 utf8

WorldInfo {
  title "VR Car in the Mountains"
  info [
    "Copyright 1998-2008 HUMUSOFT s.r.o. and The MathWorks, Inc.",
    "$Revision: 1.1.6.1 $",
    "$Date: 2008/10/31 07:12:48 $",
    "$Author: batserve $"
  ]
}

NavigationInfo {
  headlight TRUE
  type "EXAMINE"
}

DEF View1 Viewpoint {
  orientation 1 0 0 -0.2
  position 20, 8, 50
  fieldOfView 1
```

```

        description "View 1 - Observer"
    }
    DEF Camera_car Transform {
        translation 3 0.25 0
        rotation 0 1 0 -1.57
        children [
            DEF VPfollow Viewpoint {
                fieldOfView 1.57
                orientation 0 1 0 -1.9
                position -2 1 -3
                description "View 2 - Driver"
            }
        ]
    }
}
DirectionalLight {
    color 1 0.8 0.7
    direction 0.9 -0.3 -0.35
    intensity 1
}
Background {
    skyAngle [
        1.39626
        1.51844
    ]
    skyColor [
        0.0 0.0 0.0
        1.0 0.8 0.8
        1.0 1.0 0.0
    ]
    groundAngle [
        1.5708
    ]
    groundColor [
        0.0 0.0 0.0
        0.0 0.50196 0.0
    ]
}
DEF Automobile Transform
{
    translation 3 0.25 0
    rotation 0 1 0 -1.57
    center 1.5 0 -0.5
    children [
        Shape {
            appearance Appearance {
                material Material {

```

```

        diffuseColor 1 0 0
    }
}
geometry IndexedFaceSet {
coord Coordinate {
    point [ 0 0 0, 3 0 0, 2.94 0.4 0, 2 0.43 0, 1.7 0.9 0, 0.9 0.9 0, 0 0.4 0
           0 0 -1, 3 0 -1, 2.94 0.4 -1, 2 0.43 -1, 1.7 0.9 -1, 0.9 0.9 -1, 0 0.4 -1]
}
coordIndex [ 0, 3, 4, 5, 6, -1,
             0, 1, 2, 3, -1,
             13, 12, 11, 10, 7, -1
             10, 9, 8, 7, -1,
             0, 7, 8, 1, -1,
             1, 8, 9, 2, -1,
             2, 9, 10, 3, -1,
             3, 10, 11, 4, -1,
             4, 11, 12, 5, -1,
             5, 12, 13, 6, -1,
             6, 13, 7, 0 ]
}
}
Transform {
    translation 0.7 0 0.03
    rotation 1 0 0 1.57
    children DEF Wheel Shape{
        appearance Appearance {
            material Material {
                diffuseColor 0.1 0.1 0.9
            }
        }
        geometry Cylinder {
            height 0.2
            radius 0.25
        }
    }
}
Transform {
    translation 2.3 0 0.03
    rotation 1 0 0 1.57
    children USE Wheel
}
Transform {
    translation 2.3 0 -1.03
    rotation 1 0 0 1.57
    children USE Wheel
}

```

```

    Transform {
        translation 0.7 0 -1.03
        rotation 1 0 0 1.57
        children USE Wheel
    },
]
}
DEF Tree1 Group
{
    children [
        Transform {
            translation 0 1.5 0
            scale 0.5 1.5 0.5
            children Shape {
appearance Appearance {
    material Material {
        diffuseColor 0.23 0.4 0.2
    }
}
geometry Sphere { }
    },
    Shape {
        appearance Appearance {
material Material {
    diffuseColor 0.4 0.23 0.13
}
}
        geometry Cylinder {
height 0.8
radius 0.1
        }
    },
]
}
DEF Wood Group {
    children [
        Transform {
            translation 15 0.3 40
            children USE Tree1
        },
        Transform {
            translation 28 -0.4 17
            scale 0.8 0.8 0.8
            children USE Tree1
        },
    ],
}

```

```

    Transform {
      translation 29 -0.2 16
      scale 0.5 0.5 0.5
      children USE Tree1
    },
  ]
}

DEF Canal Shape {
  appearance DEF ElevApp Appearance {
    material Material {
      ambientIntensity 0.25
      diffuseColor 0.551217 0.904409 0.269294
      specularColor 0.0955906 0.0955906 0.0955906
      shininess 0.078125
    }
  }
  geometry ElevationGrid {
    height [ 9, 11, 9, 7, 4, 2, -2, -3, -4, -4, -2, 0, 0, 0.6, 0.9,
      9, 7, 7, 5, 3, 0, -2.5, -3.5, -4, -4, -1.9, 0, 0.5, 0.8, 0.8,
      7, 0, 0, 1, 1, 0, 0, -3, -4, -4, -1.5, 1, 1.5, 1.9, 1.9,
      4, 0, 0, 0.3, 0, 0, 0, -3, -4, -4, -1.5, 2, 2.2, 2.9, 2.4,
      2, 0, 0, 0, 0, 0, 0, -3, -4, -4, -1, 2, 3, 4.1, 3.5,
      1, 0, 0, 0, 0, 0, 0, -3.9, -4, -1, 0, 1, 4, 4.6, 4.0,
      1, 0.2, 0, 0, 0, 0, 0, -3.9, -4, -1, 0, 0, 3.8, 4.3, 4.6,
      0.3, 0, 0, 0, 0, 0, 0, 0, 0, -4, -4, 0, 0, 0, 4.1, 5,
      0, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0, 0, 0, 3.9, 5,
      0, 0, 0, 0, 0, 0, 0, 0, 0, -4, 0, 0, 0, 3.5, 4.7,
      0, 0, 0, 0, 0, 0, 0, 0, 0, -4, -4, 0, 0, 1, 2.9, 3.1,
      0, 0, 0, 0, 0, 0, 0, 0, 0, -4, -4, -1, 0, 0.5, 1.7, 1.7,
      0, 0, 0, 0, 0, 0, 0, -3.1, -3.5, -4, -3, 0, 0.3, 1, 1,
      0, 0, 0, 0, 0, 0, 0, -3, -4, -1, 0, 0, 0, 0.3, 1,
      0, 0, 0, 0, 0, 0, 0, -3, -4, -1, 0, 0, 0, 0, 1,
      -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10, -10]
    xDimension 15
    xSpacing 3
    zDimension 16
    zSpacing 3
    creaseAngle 3.14
  }
}

DEF River Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.6 0.8 1
      emissiveColor 0.6 0.8 1
    }
  }
}

```

```

        transparency 0.5
    }
}
geometry IndexedFaceSet {
    coord Coordinate {
        point [ 18 -2.5 0, 30 -2.5 0, 30 -2.5 42, 18 -2.5 42,
            18 -10 45, 30 -10 45 ]
    }
    coordIndex [ 3, 2, 1, 0, -1,
        3, 4, 5, 2, -1 ]
    }
}
DEF Bridge Shape {
    appearance Appearance {
        material Material {
            diffuseColor 0.9 0.6 0.6
        }
    }
    geometry IndexedFaceSet {
        coord Coordinate {
            point [ 24 0 24.5, 24 0 26.5, 30 0 26.5, 30 0 24.5,
                24 -0.5 24.5, 24 -0.5 26.5, 30 -0.5 26.5, 30 -0.5 24.5 ]
        }
        coordIndex [ 0, 1, 2, 3, -1,
            7, 6, 5, 4, -1,
            1, 5, 6, 2, -1,
            3, 7, 4, 0, -1]
    }
}
DEF Road Shape {
    appearance Appearance {
        material Material {
            diffuseColor 0.9 0.9 0.9
        }
    }
    geometry IndexedFaceSet {
        coord Coordinate {
            point [ 3.5 0.01 6, 24 0.01 26.5, 24 0.01 24.5, 5.5 0.01 6,
                3.5 0.01 12, 5.5 0.01 10.5,
                30 0.01 24.5, 30 0.01 26.5, 36 0.01 26.5, 36 0.01 24.5 ]
        }
        coordIndex [ 0, 4, 5, 3, -1,
            4, 1, 2, 5, -1,
            6, 7, 8, 9, -1]
    }
}

```

```

DEF Tunnel Transform {
  translation 4.5 1 6
  children [
    Transform {
      translation 0 0 -0.5
      rotation 1 0 0 1.57
      children [
Shape {
  appearance Appearance {
    texture ImageTexture { url "vrmount1.gif" }
  }
  geometry Cylinder {
    height 2
    side FALSE
    bottom FALSE
  }
},
Shape {
  appearance USE ElevApp
  geometry Cylinder {
    height 2
    top FALSE
    bottom FALSE
  }
},
    ]
  },
  Shape {
    appearance Appearance {
texture ImageTexture { url "vrmount2.gif" }
    }
    geometry IndexedFaceSet {
coord Coordinate {
  point [ -1 0 0.51, -1 -2 0.51, 1 -2 0.51, 1 0 0.51 ]
}
coordIndex [ 0, 1, 2, 3, -1]
    }
  }
  Shape {
    appearance USE ElevApp
    geometry IndexedFaceSet {
coord Coordinate {
  point [ -1 0 0.51, -1 -2 0.51, 1 -2 0.51, 1 0 0.51
    -1 0 -0.5, -1 -2 -0.5, 1 -2 -0.5, 1 0 -0.5 ]
}
coordIndex [ 4, 5, 1, 0, -1,

```

```
        6, 7, 3, 2, -1]
      }
    },
  ]
}
Transform {
  rotation 0 1 0 -1.57
  translation 42 0 21
  children USE Tunnel
}
ROUTE Automobile.translation_changed TO Camera_car.translation
ROUTE Automobile.rotation_changed TO Camera_car.rotation
```

7.5 Simulink接口虚拟现实示例

7.5.1 灯光的模拟

Simulink 允许用户在子系统的基础上构造更为复杂的模型，其中每一个子系统都是相对完整地 完成一定功能的模块框图。通过对子系统的封装，用户还可以实现带触发或使能功能的特殊子系统。子系统的概念体现了分层建模的思想，是 Simulink 的重要特征之一。

下面用 Simulink 建立对灯光的模拟，其具体步骤如下：

（1）在 Simulink 窗口中建立灯光的总体仿真模型，如图 7-24 所示。

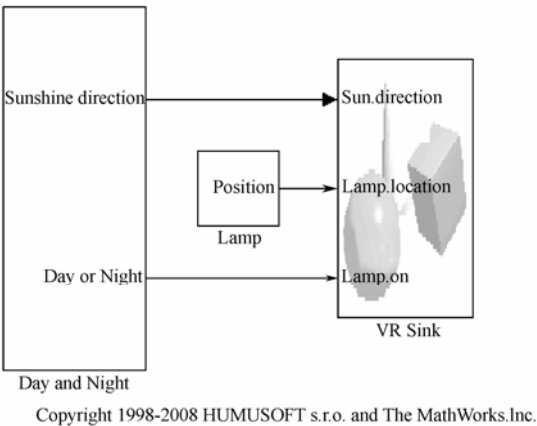


图 7-24 Simulink 中的灯光仿真模型

（2）这一总体仿真模型由三大模块组成：Day and Night、Lamp 和 VR Sink。先来看一下 Lamp 模块的仿真模型，如图 7-25 所示。

（3）Day and Light 模块的仿真模型如图 7-26 所示。

（4）在此可以对其其中的一些参数进行修改，如双击 Night 模块，可以修改它的常数值，如图 7-27 所示。

（5）现在回到总体的仿真模型中，然后双击 VR Sink 模块，将这一仿真模型的信号送到虚拟世界中，这时出现图 7-28 所示的对话框。

(6) 在图 7-28 中可以看到这一模型的 VRML 文件是 `vrlights.wrl`, 其源程序代码在本小节后面列出。如果想让别的客户机也能同时浏览这一三维场景, 就在“Output”中勾选“Allow viewing from the Internet”选项。单击“Edit”按钮, 可以对三维模型进行编辑、修改, 如图 7-29 所示。

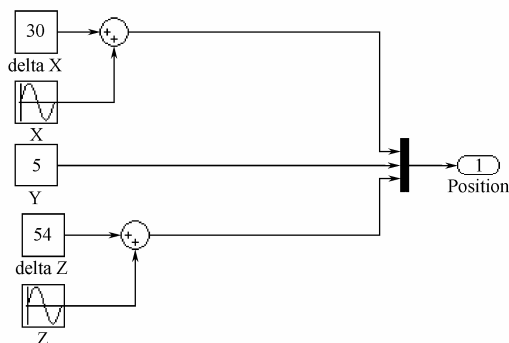


图 7-25 Lamp 模块的仿真模型

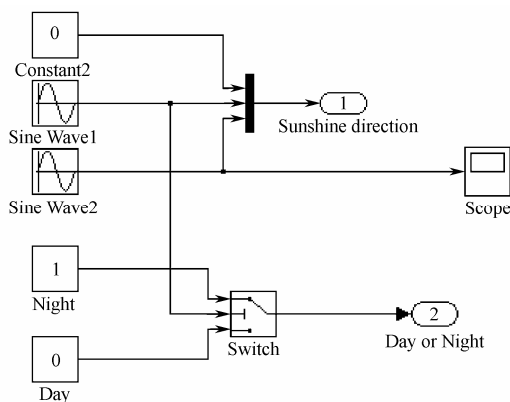


图 7-26 Day and Light 模块的仿真模型

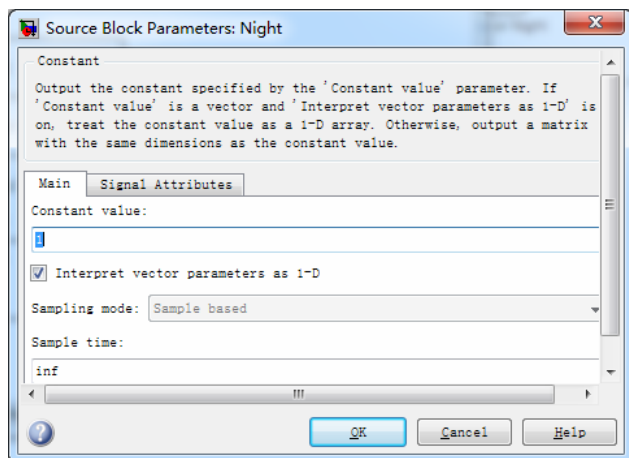


图 7-27 修改 Night 模块的参数值

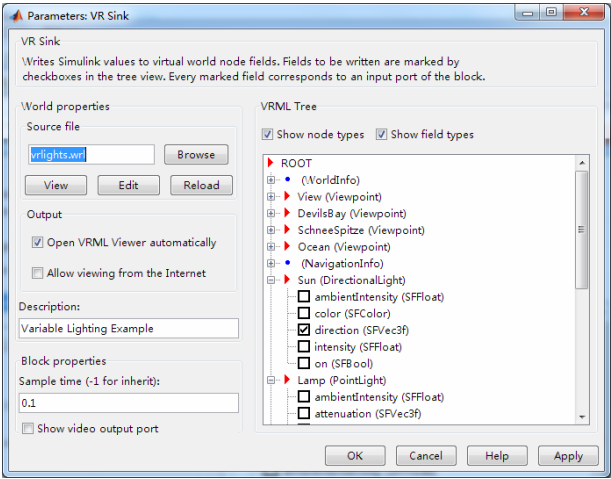


图 7-28 VR Sink 的参数对话框

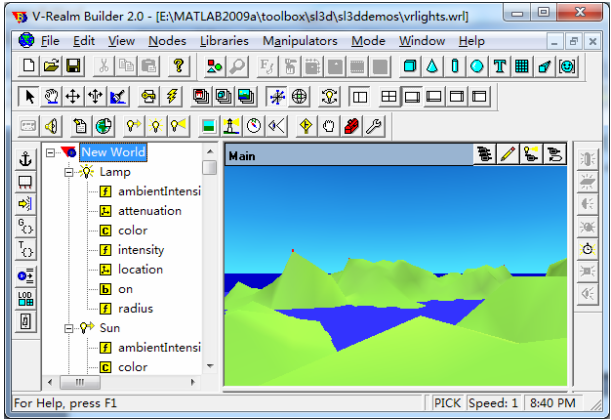


图 7-29 V-Realm Builder 2.0 中的灯光三维场景

(7) 关于如何编辑三维场景请参看前面第 2~5 章的相关内容。当完成对灯光的三维场景编辑后关闭 V-Realm Builder 2.0 窗口，回到图 7-28 所示的界面，单击“View”按钮，在浏览器看到不同角度的景象，如图 7-30~图 7-32 所示。

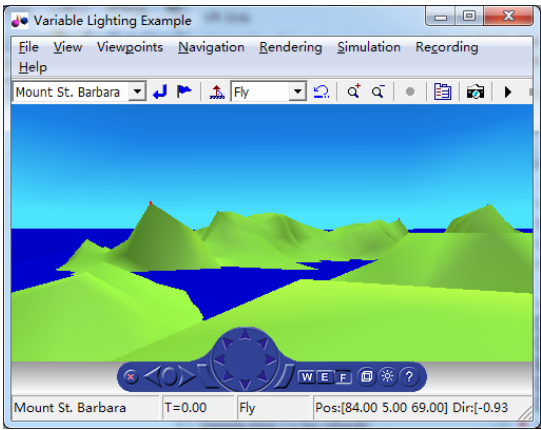


图 7-30 浏览器中的灯光虚拟现实之一

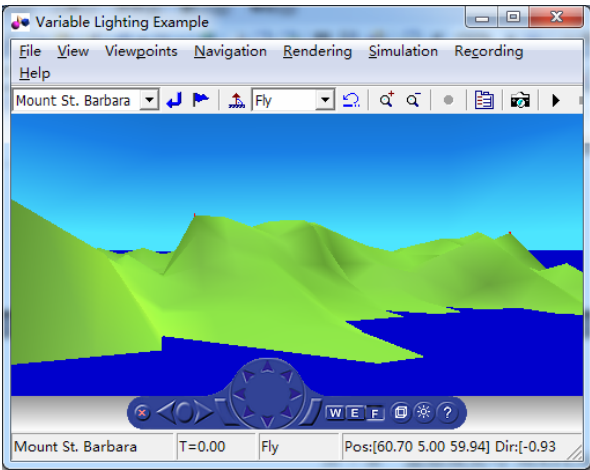


图 7-31 浏览器中的灯光虚拟现实之二

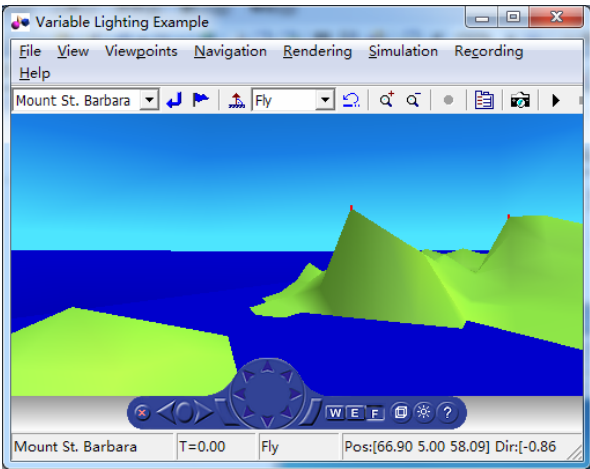


图 7-32 浏览器中的灯光虚拟现实之三

Simulink 加速器能够提高模型仿真的速度，它的基本工作原理是利用 Real-Time Workshop 工具将模型方框图转换成 C 语言代码，然后采用编辑器将 C 代码编译成可执行代码。由于用可执行代码取代了原有的 MATLAB 解释器，所以其仿真速度有了本质的提高。

Simulink 既可以工作在正常模式（Normal）下，也可以工作在加速模式（Accelerator）下。工作在加速模式下时，Simulink 将 C 代码编译成.mex 文件，在编译过程中，对原有的 C 代码进行优化重组，可以极大地提高模型仿真的速度，并且模型越复杂，提高的程度越明显，一般来说，可以将仿真的速度提高 2~6 倍。

为了使 Simulink 工作在加速模式下，可以单击【Simulation】菜单下的【Accelerator】选项，如果要恢复到正常模式，即可单击【Simulation】菜单下的【Normal】选项。

灯光的 VRML 文件 vrlights.wrl 源程序代码如下：

```
#VRML V2.0 utf8

WorldInfo {
  title "Variable Lighting Example"
```

```
info [  
  "Copyright 1998-2008 HUMUSOFT s.r.o. and The MathWorks, Inc.",  
  "$Revision: 1.1.6.1 $",  
  "$Date: 2008/10/31 07:12:33 $",  
  "$Author: batserve $"  
]  
}  
DEF View Viewpoint {  
  position 84, 5, 69  
  orientation 0 1 0 1.2  
  description "Mount St. Barbara"  
}  
DEF DevilsBay Viewpoint {  
  position 14, 1, 75  
  description "Devil's Bay"  
}  
DEF SchneeSpitze Viewpoint {  
  position 15, 9, 15  
  orientation 0 -1 0 2.0  
  description "Schnee Spitze"  
}  
DEF Ocean Viewpoint {  
  position 48, 0, 150  
  description "Ocean"  
}  
NavigationInfo {  
  type "FLY"  
  headlight FALSE  
}  
DEF Sun DirectionalLight {  
  color 1 1 1  
  direction 0 -1 0  
  intensity 1  
}  
DEF Lamp PointLight {  
  color 1 0.8 0.5  
  attenuation 0 0 0  
  on TRUE  
  radius 30  
}  
DEF Bckg Background {  
  skyAngle [  
    1.3  
    1.51844  
  ]  
  skyColor [  
    0.5 0.5 0.5  
    0.5 0.5 0.5  
    0.5 0.5 0.5  
  ]  
}
```

[illegible]

```
0 24225 46576 75310 06832 00000 01222,0
003524 33645 62000 00310 00000 00012,0
002423 33453 51000 00000 00000 00001,0
003231 46643 30000 00000 00000 00000,0
014342 57754 21000 00000 00000 00000,0
025674 79842 10000 00000 00000 00000,0
036747 98984 21000 00000 05550 00000,0
058968 79764 21000 00000 55050 00000,0
086746 68875 43211 00000 05550 00000,0
077523 47965 32110 00000 03000 00000,0
065410 25974 21100 00000 11100 00000,0
053300 13333 11000 00000 00000 00000,0
063200 02454 33500 00000 00000 24444,0
054100 00336 66700 00000 00000 44334,0
063200 00145 67900 00011 21100 33214,0
042100 01233 34200 00123 23301 23223,0
011000 00122 22110 00011 22200 12021,0
001200 00034 21000 00001 11000 00000 0,
000000 00000 00000 00000 00000 00000 0]
xDimension 32
xSpacing 3
zDimension 28
zSpacing 3
    creaseAngle 3.14
}
}
Transform {
    translation 24 8.8 57
    children DEF LightHouse Shape {
        appearance Appearance {
            material Material {
                diffuseColor 0 0 0
                emissiveColor 1 0 0
            }
        }
        geometry Cylinder
        {
            height 0.8
            radius 0.1
            bottom FALSE
        }
    }
}
Transform {
    translation 39 8.8 69
    children USE LightHouse
```

```

}
Transform {
    translation 52.5 8.8 22.5
    children USE LightHouse
}
Transform {
    translation 27 7.8 15
    children USE LightHouse
}
DEF Sc1 Script {                                #背景颜色直接取决于灯光
    eventIn  SFVec3f direction
    eventOut MFColor sky
    eventOut MFColor ground
    field    SFNode  atollMaterial USE AtollMaterial
    url "vrmlscript:
        function direction(value) {
            y = value[1];
            if (y > 0 || y < -1)
            {
                s = new SFColor(0, 0, 0);
                sky[0] = s;
                sky[1] = s;
                sky[2] = s;
                ground[0] = s;
                ground[1] = s;
            }
            else
            {
                s = new SFColor(0, 0, 0.1);
                sky[0] = s;
                sky[1] = new SFColor(-0.1*y, -0.5*y, -0.9*y);
                sky[2] = new SFColor(-0.3*y, -0.9*y, -y);
                ground[0] = s;
                ground[1] = new SFColor(0, 0, -0.8*y);
                atollMaterial.diffuseColor = new SFColor(0, 0, -0.8*y);
            }
            s = sky[2];
            if (s[0] < 0.05) s[0] = 0.05;
            if (s[1] < 0.05) s[1] = 0.05;
            if (s[2] < 0.2) s[2] = 0.2;
            sky[2] = s;
        }"
    }
ROUTE Sun.direction_changed TO Sc1.direction
ROUTE Sc1.sky TO Bckg.set_skyColor
ROUTE Sc1.ground TO Bckg.set_groundColor

```


源代码见本小节后面), VRML 节点域的名字是 Grab_Sensor.translation_changed, 如图 7-35 所示。

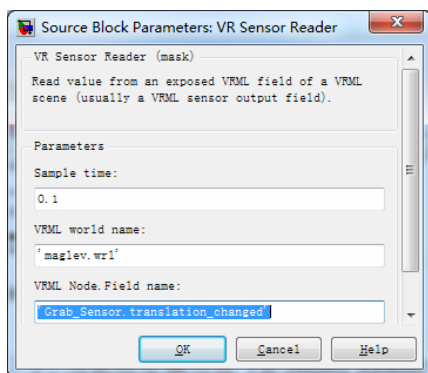


图 7-35 VR Sensor Reader 模块

(4) 在磁悬浮的仿真模型中, 双击 VR Sink 模块, 将模型的信号送到虚拟世界中, 这时出现图 7-36 所示的模块参数对话框。

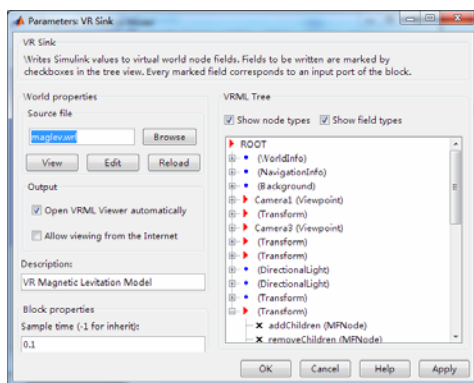


图 7-36 VR Sink 模块参数对话框

(5) 单击“Edit”按钮, 可以对三维的磁悬浮场景进行编辑和修改。编辑后的图像如图 7-37 所示。

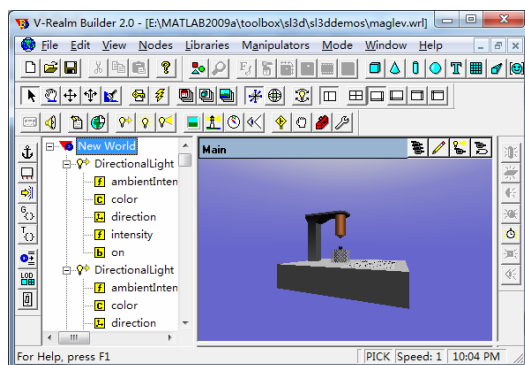


图 7-37 三维的磁悬浮场景

(6) 编辑完后关闭 V-Realm Buider, 在前一画面(图 7-36)中选取“View”按钮, 则可在

浏览器中观察到不同角度的磁悬浮场景，如图 7-38~图 7-40 所示。这里的技巧是使用浏览器中出现的操纵面板。

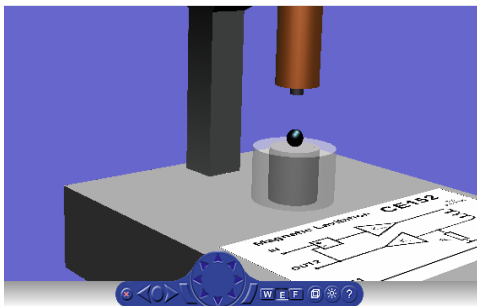


图 7-38 浏览器中观察到的不同角度的磁悬浮场景之一

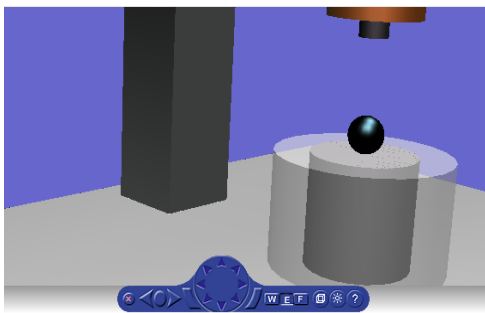


图 7-39 浏览器中观察到的不同角度的磁悬浮场景之二



图 7-40 浏览器中观察到的不同角度的磁悬浮场景之三

如果用户调试的模型很大，可以采用加速器加快调试的进程。例如，如果仿真执行到某个断点处的时间很长，就可以采用加速器来尽快达到所要调试的断点处。在调试状态下运行加速器的方法是单击【Simulation】菜单下的【Accelerator】选项。

磁悬浮模型的 VRML 文件 maglev.wrl 源程序代码如下：

```
#VRML V2.0 utf8WorldInfo {  
  title "VR Magnetic Levitation Model"  
  info [  
    "Copyright 1998-2008 HUMUSOFT s.r.o. and The MathWorks, Inc.",  
    "$Revision: 1.1.6.1 $",
```

```

"$Date: 2008/10/31 07:11:48 $",
"$Author: batserve $"
]
}
NavigationInfo {
type "EXAMINE"
headlight FALSE
}
Background {
groundColor 0 0.8 0
skyAngle [1 1.2 1.5708]
skyColor [
    0.8 0.8 1
    0.7 0.7 0.8
    0.5 0.5 0.8
    0.4 0.4 0.8
]
}
DEF Camera1 Viewpoint {
orientation 0 1 0 0.4
position    7 4 15
fieldOfView 1.2
description "Camera 1"
}
Transform {
center 70 40 150
translation 70 40 150
rotation 0 1 0 0.22
children
DEF Camera2 Viewpoint {
    orientation 1 0 0 -0.21
    position    0 0 0
    fieldOfView 0.07
    description "Pseudo ortho view"
}
}
DEF Camera3 Viewpoint {
orientation 0 1 0 0.45
position    7 4 15
fieldOfView 0.3
description "Camera 3"
}
Transform {
center 0 0 0
translation 100 50 0
rotation 0 1 0 1.57

```

```

children
DEF Camera4 Viewpoint {
    orientation 1 0 0 -0.45
    position    0 0 0
    fieldOfView 0.15
    description "Camera 4"
}
}
Transform {
center 1 2.5 3
translation 1 2.5 3
rotation 0 1 0 0.2
children
DEF Camera5 Viewpoint {
    orientation 1 0 0 0.7
    position    0 0 0
    fieldOfView 0.7
    description "Camera 5"
}
}
DirectionalLight {
direction -1 -2 -1
color 1 1 1
intensity 1
}
DirectionalLight {
direction 0 -2 1
color 1 1 1
intensity 1
}
Transform {
scale 0.5 0.5 0.5
children [
    #Base
    Shape {
        appearance Appearance {
            material    Material {
                ambientIntensity .25
                diffuseColor    .4 .4 .4
                specularColor .77 .77 .77
                shininess .6
            }
        }
        geometry IndexedFaceSet {
            colorPerVertex FALSE
            color NULL

```

```

        creaseAngle 1
        solid FALSE
        coord Coordinate {
            point [
                15 5 10, 15 0 10, 15 5 -10, 15 0 -10, -10 5 10, -10 0 10,
                -10 5 -10, -10 0 -10
            ]
        }
        coordIndex [
            4 0 1 5 -1 7 3 2 6 -1 6 2 0 4 -1 2 3 1 0 -1 3 7 5 1 -1 7 6 4 5 -1
        ]
    }
}
#Diagram
Shape {
    geometry IndexedFaceSet {
        colorPerVertex FALSE
        color NULL
        creaseAngle 1
        solid FALSE
        coord Coordinate {
            point [
                13 5.1 9, 13 5.1 -9, 4 5.1 -9, 4 5.1 9,
            ]
        }
        coordIndex [
            0 1 2 3 -1
        ]
        texCoord TextureCoordinate {
            point [0 0, 1 0, 1 1, 0 1]
        }
    }
    appearance Appearance {
        texture ImageTexture {
            url ["texture/maglev.gif"]
        }
    }
}
#Humusoft Logo
Shape {
    appearance Appearance {
        texture ImageTexture {
            url ["texture/humusoft.png"]
        }
    }
    geometry IndexedFaceSet {

```

```

        colorPerVertex FALSE
        color NULL
        creaseAngle 1
        solid FALSE
        coord Coordinate {
            point [
                15.1 4 7, 15.1 1 7, 15.1 1 -7, 15.1 4 -7
            ]
        }
        coordIndex [
            0 1 2 3 -1
        ]
        texCoord TextureCoordinate {
            point [0 1, 0 0, 1 0, 1 1]
        }
    }
}
Shape {
    appearance Appearance {
        material Material {
            ambientIntensity .137
            diffuseColor .57 .58 .58
            specularColor .67 .46 .44
            shininess .17
        }
    }
    geometry IndexedFaceSet {
        colorPerVertex FALSE
        color NULL
        creaseAngle 1
        solid FALSE
        coord Coordinate {
            point [
                -6.5 15 1.3,
                -6.5 5 1.3,
                -6.5 15 -1.3,
                -6.5 5 -1.3,
                -9 15 1.3,
                -9 5 1.3,
                -9 15 -1.3,
                -9 5 -1.3,
                -9 16.5 2.5,
                -9 16.5 -2.5,
                -4 16.5 2.5,
                -4 16.5 -2.5,
            ]
        }
    }
}

```

```

    }
    coordIndex [
        4  0  1  5 -1
        7  3  2  6 -1
        2  3  1  0 -1
        3  7  5  1 -1
        7  6  4  5 -1
        11 9  10 8 -1
        4  8  10 0 -1
        0  10 11 2 -1
        2  11 9  6 -1
        8  4  6  9
    ]
}
}
Shape {
    appearance Appearance {
        material Material {
            ambientIntensity .0933
            diffuseColor .38 .38 .42
            specularColor .53 .53 .53
            shininess .93
        }
    }
    geometry IndexedFaceSet {
        colorPerVertex FALSE
        color NULL
        creaseAngle 1.4
        solid FALSE
        coord Coordinate {
            point [
                -9  16.5  2.5,
                1  16.5  2.5,
                2  16.5  1.5,
                2  16.5 -1.5,
                1  16.5 -2.5,
                -9  16.5 -2.5,
                -9  17  2.5,
                1  17  2.5,
                2  17  1.5,
                2  17 -1.5,
                1  17 -2.5,
                -9  17 -2.5
            ]
        }
    }
    coordIndex [

```

```

        0  1  2  3  4  5  -1
        0  6  7  1  -1
        1  7  8  2  -1
        2  8  9  3  -1
        3  9  10 4  -1
        4  10 11 5  -1
        5   0  6  11 -1
        11 10  9  8  7  6  -1
    ]
}
}
Transform {
translation 0 14 0
children [
    Shape {
        appearance Appearance {
            material Material {
                ambientIntensity .0833
                diffuseColor .44 .1 0
                specularColor 1 .68 .51
                shininess .07
                emissiveColor .15 .13 .06
            }
        }
        geometry Cylinder {
            radius 1.3
            height 5
        }
    }
]
}
Transform {
translation 0 11.25 0
children [
    Shape {
        appearance Appearance {
            material Material {
                ambientIntensity .06
                diffuseColor .24 .24 .24
                specularColor .3 .25 .3
                shininess .05
            }
        }
        geometry Cylinder {
            radius 0.4
            height 0.5

```

```

    }
  }
]
}
Transform {
  translation 0 6.5 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          ambientIntensity .06
          diffuseColor .24 .24 .24
          specularColor .3 .25 .3
          shininess .05
        }
      }
      geometry Cylinder {
        radius 1.5
        height 3
      }
    }
  ]
}
Transform {
  translation 0 6.5 0
  children [
    Shape {
      appearance Appearance {
        material Material {
          ambientIntensity 0
          diffuseColor .29 .29 .29
          specularColor 1 .94 .54
          shininess .83
          emissiveColor .4 .4 .4
          transparency .4
        }
      }
      geometry Cylinder {
        radius 2.5
        height 3
      }
    }
  ]
}
]
}

```

```
Transform {
translation 0 4.25 0
children [
    DEF Grab_Sensor PlaneSensor {
        enabled TRUE
        autoOffset TRUE
        maxPosition      0 1
        minPosition      0 0
        offset 0 0 0
    }
    DEF Ball Transform {
        translation 0 0 0
        children Shape {
            appearance Appearance {
                material DEF Ball_material Material {
                    diffuseColor    0 0 0
                    specularColor .63 .92 1
                    ambientIntensity 0
                }
            }
            geometry Sphere {
                radius 0.25
            }
        }
    }
]
}
# ROUTE Grab_Sensor.translation_changed TO Ball.translation
```

第 8 章 VRML脚本语言与编程

脚本节点指包含语言程序设计的节点，而且这个程序设计应该能被浏览器解释并运行。前面讲到的插补器节点、传感器节点和路由的结合可以实现虚拟现实的动画设计，但是如果引入程序设计，将会充分发挥编程灵活、控制手段多样、实现范围更广的特点，设计出更加生动、复杂的动画。

8.1 脚 本

Script 节点可以描述一些由用户自定义制作的传感器和插补器，能接受事件，对其处理，并输出处理后的结果。这些检测器和插补器需要一些有关域、事件出口和事件入口的列表及处理这些操作时所须做的事件。因此，该节点又定义了一个包含程序脚本节点的域（注意不能定义 exposedField）、事件出口、事件入口及描述用户自定义制作的传感器、插补器所做的事情。Script 节点可以出现在文件的顶层或作为成组节点的子节点。

脚本节点语法格式如下：

```
DEF <节点名>Script{
    域、输入事件、输出事件定义
    url"<脚本语言声明>:
    脚本语言程序
"
```

其中，DEF<节点名>从语法上讲并不是必需的，但是实际应用中一般总要用到，读者可以从下面的示例中看到。

在 Script 节点中，可由用户定义一些域、入事件和出事件等，所以 Script 节点的结构与前面介绍的 VRML 节点有所不同。

Script 节点定义如下：

节点名称	域名称	域值	#域及域值类型
Script {	url	[]	#exposedField MSFtring
	direction	FALSE	# SFBool
	mustEvent	FALSE	# SFBool
	eventIn		#eventTypeName eventName
	eventOut		#eventTypeName eventName
}			

其中：

(1) url 域的域值定义为一个 url 列表。其域值指定的程序脚本可以由任何 VRML 浏览器支持的语言写成。通常情况下，VRML 浏览器支持的语言有 Java 语言和 JavaScript/VRMLScript 脚本语言。而且 JavaScript 的函数和指令可以直接包含在 url 域中。其值默认为空列表。

JavaScript 是由 Netscape 公司创建的一种脚本语言。虽然名字跟 Java 差不多，但它与 Java 并没有直接的联系。它的设计目的是为了在 Web 页中编写脚本，它比较简单。它由内核和一些外延的对象组成，这些对象一般都能提供一种特殊的功能，如实现与当前通用文件的通信。在 VRML 2.0 创建的时候，创始者们就在寻求一种简单的语言来作 VRML 的脚本成分，VRMLScript 是 JavaScript 的一个子集，它支持 VRML 的数据类型。VRMLScript 与其他脚本语言相比有以下特性。

① 脚本可以用源代码或单独的 URL 形式引入。

② 直接支持 VRML 2.0 的所有数据类型。

③ 使用单独的函数接收外部事件，可以简化开发过程，提高运算速度。

④ 使用简单的赋值向外部发送事件。

⑤ 在表达式中可直接用标量数据（SFTIME、SFInt32、SFFloat、SFBool）。JavaScript 数据对象可直接转换为此 4 种数据类型的任何一种。

⑥ 可使用构造器轻松创建与转换大多数的数据类型。

⑦ 数据与字符串对象与全部 JavaScript 函数完全兼容。

⑧ 全部 JavaScript string 方法与属性可用，标量可自动转化为字符变量。

(2) **mustEvent** 域的域值指定程序脚本如何进行求值。当该域值为 **TRUE** 时，每当由节点的 **eventIn** 事件接收到一个新值时，浏览器就立即对该程序脚本进行计算。当该域值为 **FALSE** 时，则浏览器在此脚本不影响事件的任何可视部分的情况下，可以推迟对脚本的计算，直到合适的时间到来。这样就会出现事件到节点的传送和计算节点处理该事件动作的延迟，此时如果多个事件被传送，待处理的事件时间就会变长。为了尽量使其性能达到最佳值，**mustEvent** 域值应设定为 **FALSE**，因为浏览器的性能取决于一个事件被发出后必须进行计算的程序脚本数目，如果将其域值设为 **TRUE**，就会增加浏览器的工作量，从而降低画面的刷新速度和交互性。该域值的默认值为 **FALSE**。

(3) **direction** 域的域值用来说明程序脚本的输出是否受到限制。当该域值为 **TRUE** 时，程序脚本可以直接对它能访问的任何节点的可见域进行写操作或对任何节点的 **eventIn** 事件送值，另外还可以在 VRML 场景中增加或删除一条通路。当该域值为 **FALSE** 时，程序脚本不能直接发送事件，不过可以访问。通常情况下，**direction** 域值设为 **FALSE**。该域值的默认值为 **FALSE**。

Script 节点可定义多个域和事件（入事件和出事件）。接口域、接口 **eventIn** 和接口 **eventOut** 都可以任意取名字，但必须遵循 **DEF** 的命名规则。按照 VRML 的约定，要区分大小写，名字必须以小写字母开头，而且名字的每个子序列单词都必须大写，允许在名字中使用下划线和阿拉伯数字。

eventIn（入事件）通常以“**set_**”开始，除非它们具有对组内进行添加或删除子元素的操作权限。**eventOut**（出事件）通常以“**_changed**”结尾，除非它们是一个布尔值或时间值。布尔类型的 **eventOut** 事件名以“**is**”开头，时间类型的 **eventOut** 事件以“**Time**”结尾。

【例 8-1】 在虚拟世界中，当来到这个房间门前，用 **Mouse** 单击门，门会自动打开，当再次单击时，门自动关闭。

其实现的源程序代码如下：

```
#VRML V2.0 utf8
```

#定义一个比较好的初始视点

```
Viewpoint {
    position 1.5 0 10
}
```

#定义两扇门，注意重用

```
Transform {
    children [
        DEF tchsensor TouchSensor {}
        DEF idoor Transform {
            children [
                DEF door Transform {
                    translation 1 0 0
                    children [
                        Shape {
                            appearance Appearance {
                                material Material {
                                    diffuseColor 0.6 0.1 0.1
                                }
                            }
                            geometry Box {
                                size 2 4 0.15
                            }
                        }
                    ]
                }
            ]
        }
        DEF rdoor Transform {
            translation 2 0 0
            center 2 0 0
            children [
                USE door
            ]
        }
    ]
}

DEF tmsensor TimeSensor {
    cycleInterval 2
}

#由于两扇门的转动方向相反，需要定义两条转动路径
DEF ontInp OrientationInterpolator {
    key [0 1]
    keyValue [
        0 1 0 0,
        0 1 0 -1.571]
}
```

```

DEF ontInp2 OrientationInterpolator {
  key [0 1]
  keyValue [
    0 1 0 0,
    0 1 0 1.571]
}
DEF controller Script {
  field SFBool isopen FALSE      #域，用于记录门当前开关状态
  eventIn SFTIME doorclicked    #入事件，门被单击时的时间
  #入事件，用于接收时段，构成连续性动画，而不是突变
  eventIn SFFloat fracIn
  eventOut SFTIME actionstart    #出事件，开关门动作开始的时间
  eventOut SFFloat fracOut       #出事件，发出时段
  url "javascript:
function doorclicked(value){
  isopen=!isopen;
  actionstart=value;
}
function fractIn(value,timeStamp)
{
  if (!isopen)
    fracOut=1-value;
  else
    fracOut=value;
}
"
}
ROUTE tchsensor.touchTime TO controller.doorclicked
ROUTE controller.actionstart TO tmsensor.startTime
ROUTE tmsensor.fraction_changed TO controller.fracIn
ROUTE controller.fracOut TO ontInp.set_fraction
ROUTE controller.fracOut TO ontInp2.set_fraction
ROUTE ontInp.value_changed TO idoor.rotation
ROUTE ontInp2.value_changed TO rdoor.rotation

```

此外，虽然 VRML 中提供了丰富的标准节点类型，利用这些标准节点，可创建出虚拟场景，但在实际工作中，仅仅靠这些标准节点还是不够的。VRML 为开发者提供了更为灵活的自定义原型方式。原型 **PROTO** 节点是一种用户可以创建新节点的类型。它可以定义新节点的名字、域、域值及节点体，一旦定义了用户的新节点，就可以像 VRML 标准节点一样去使用它，通常放在 VRML 文件的前面部分。

同样，还有一个创建外部定义的新节点 **EXTERNPROTO** 节点，它是在 VRML 主程序中定义外部新节点的名字、域值事件、域值类型、域值事件名字及 url，并且在一个外部 VRML 文件中定义一个或多个 **PROTO** 新节点的名字、域、域值、事件与节点主体。

8.2 VRMLScript语言

8.2.1 表达式

表达式是变量、值和运算符的组合。VRMLScript 和 JavaScript 中的变量、操作符、表达式用法一致。

1. 变量

变量名以英文字母或下划线“_”开头，后面可以跟任意字母、数字，变量名句区分大小写。变量的数据类型和第1章讲的一样。节点的域相当于变量。数组可以看成变量（名称有规律）的集合，数组下标从0开始。`sec.rotation[0]`表示对象 `sec` 的一维数组 `rotation` 的第一个元素。

VRML2.0 有 14 个关键字属于保留字，定义变量名时应予以回避。

2. 运算符

1) 算术运算符：+，-，*，/，++，--，%

其中++、--可以作为前缀或后缀运算，结果不一样。如 `i=1`，`x=++i`，则 `i` 先加 1 再赋值给 `x`，于是 `x` 等于 2；而 `j=1`，`x=j++` 先赋值给 `x`，于是 `x` 等于 1，然后 `j` 再自增 1 为 2。%是求余运算符，如 `10%3` 得余数 1。

2) 赋值运算符：=，+=，-=，*=，/=，%=

例如，`x=100`，`y=200`，`x+=y` 相当于 `x=x+y`，结果 `x` 被赋值 300。

3) 关系运算符：==（等于），!=（不等于），>（大于），<（小于），>=（大于等于），<=（小于等于）

关系运算符一般用来比较两个变量或值是否相等，或比较大小。关系运算符是比较值，而赋值运算符是分配值。关系运算的结果是逻辑值 TRUE 或 FALSE。如 `x=8`，`y=9`，`x==y` 的结果是 FALSE。

4) 逻辑运算符：&&（逻辑与），||（逻辑或），！（逻辑非）

逻辑运算一般用来为两个变量或值作运算，字符在 VRML 中是区分大小写的。作逻辑运算的结果为真时，返回 1，为假时返回 0。如 `A&&A` 的结果为 0。

5) 字符串运算符：+，+=

VRML 中的字符串用双引号或单引号作为定界符，比如‘虚拟现实’。+、+=用于字符串连接。如‘仿真’+‘技术’，结果为：‘仿真技术’；`ss=‘虚拟现实’`，`ss+=‘语言’`，`ss` 的运算结果为‘虚拟现实语言’。

一般在 VRML 文本造型节点中，在 Script 节点前面域定义时用双引号作为字符串的定界符，在 Script 的脚本程序中字符串用单引号作为定界符。

6) 条件赋值运算符：?:

例如，`x=(7>5?9:10)`，这里先计算表达式 `7>5` 是否为真，如果真则 `x=9`，否则 `x=10`。表达式可以是变量形式。

7) 运算符优先级

运算符的优先级运算顺序：括号→求反/逻辑非/递增减（!、++、--）→乘/除/求余→加/减→关系运算→相等（==）→逻辑与→逻辑或→赋值。

3. 特殊字符

VRML 中有一些特殊字符在引用时前面要加 “\”，见表 8-1。

表 8-1 特殊字符的引用形式

符号	\b	\f	\n	\r	\t	\'	\"	\\
含义	退格	换页	换行	回车	水平制表符	单引号	双引号	反斜杠

例如，如果要表达“设计一个“模拟战场””，这里在双引号定界的字符串内又使用双引号，应该写成“设计一个\“模拟战场””。

8.2.2 语法

VRMLScript 和 JavaScript 的基本语法是一致的。多条语句应放在“{”与“}”内，语句后用“:”结尾。

1. 条件语句

格式 1:

```
if (条件表达式)
{
    语句体;
}
```

格式 2:

```
if (条件表达式)
{
    语句体;
else {
    语句体;
}
}
```

格式 3:

```
if (条件表达式)
{
    语句体;
}
else {
    语句体;
}
```

2. 选择语句

其格式如下:

```
switch (表达式)
{
    //开始花括号
case 标签 1:
    语句体;
case 标签 2:
    语句体;
```

```
        break;
        .....
default:
    语句体;
}    //结束花括号
```

switch 语句主要包含在一对花括号内。根据表达式的计算结果，选择和其匹配的标签执行此标签下的语句，由 **break** 语句终止 **switch** 语句的执行，转到结束花括号后。如果没有匹配的标签，则执行 **default** 后的语句。

例如：

```
num=2;
switch (num)
{
case 1:
    print('不匹配的数值条件!');
    break;
case 2:
    print('匹配的数值条件! ');
    break;
default;
    print('没有匹配的数值条件! ');
}
```

由于 `num=2`，故在控制台中显示：匹配的数值条件！

3. 循环语句

1) while 语句

其调用格式如下：

```
while (条件表达式)
{
    语句体;
}
例如：
i=0;
while (i<10)
{
    print('i='+i);
    ++i
}
```

2) do-while 语句

其调用格式如下：

```
do
{
    语句体;
}
```

while (条件表达式);

while 语句是先判断条件再决定是否进入循环，而 **do-while** 语句是先进入循环一次，再判断条件决定是否继续循环。

例如：

```
i=10;
do
{
    print('i='+i);
    i++
}
while (i<10);
```

这时，控制台上显示了 **i=10**，而用 **while** 语句就不会显示了。

3) for 语句

其调用格式如下：

```
for (初始条件;循环条件;步长 )
{
    语句体;
}
例如：
for (i=0;i<10 ;++i )
{
    print('i='+i);
}
```

此例的结果和 **while** 语句的一样，但语句看起来简练多了。

4) for-in 语句

对象操作语句，用于对象属性或数组（看成对象）的所有元素进行循环操作。这种循环无须知道循环的次数，能自动将所有属性或元素操作一次。

其调用格式如下：

```
for (变量名 in 对象名)
```

例如，用于对象属性，新建一个对象 **sy**，**sy=new SFCOLOR(196 178 120)**（见下面的 **new** 语句），则语句：**for(vars in sy){print('vars='+vars);}** 分别显示 **sy** 的 3 个属性名 **r**、**g**、**b**（注意不是 196、178、120 这 3 个值）。

又如用于数组，如果已经定义：**field SFVec3f a1 2 3 4**，可以用下面的语句显示 **a1** 的 3 个数值 2、3、4：**for(ary in a1){print('a1='+a1[ary]);}**。

上面的例子中的变量名 **vars** 和 **ary** 由程序员自由命名，无须先定义。

4. 跳转语句

1) break 语句

用于中止当前的循环，强制退出循环外。

2) continue 语句

用于中止当前的循环，退出循环内的条件判断部分，判断是否继续循环。

3) return 语句

从函数调用中立即返回，可以携带返回参数，如 `return as`，这里 `as` 即为返回参数。

5. with 语句

对象操作语句，对象的引用方法之一。

其调用格式如下：

```
with (对象名)
{
    语句体;
}
```

例如，脚本程序中有下面一段语句：

```
sec.rotation[0]=0;
sec.rotation[1]=0;
sec.rotation[2]=1;
sec.rotation[3]=sz*i;
sec.translation[0]=x;
sec.translation[1]=y;
sec.translation[2]=z;
```

这里 `sec` 是对象名称，“.” 符号是对象的一种引用方法，也可以用 `with` 语句：

```
with(sec)
{
    rotation[0]=0;
    rotation[1]=0;
    rotation[2]=1;
    rotation[3]=sz*i;
    translation[0]=x;
    translation[1]=y;
    translation[2]=z;
}
```

这两种引用的效果是一样的。

6. new 语句

对象操作语句，用于新建一个对象实例。

其调用格式如下：

```
新对象名=new 对象类型 (参数 1[, 参数 2]...[, 参数 n])
```

例如：

```
sy=new SFCOLOR(196 178 120)
```

8.2.3 函数

1. 脚本程序的几个默认函数

`initialize`、`print`、`eventsProcessed` 和 `shutdown` 函数，函数名是不能变更的，它们在脚本运

行中影响代码的运行流程。

`initialize` 是自动首先运行的函数代码。

`print` 函数可以在浏览器的控制台中显示其参数值。

`eventsProcessed` 脚本处理入事件后会自动调用此函数。

`shutdown` 函数仅仅在有节点被删除、卸载或替换时才调用。

2. 字符和数值转换函数

`parseInt(<字符表达式>,[radix])`函数

能将数值字符转换为整型数值，如果`[radix]`忽略，则字符表达式中的数值代表十进制数；如果`[radix]`是 16 或 8，则为十六进制或八进制数。如，`parseInt('123.45')`、`parseInt('123.45', 16)`、`parseInt('123.45', 8)`转换结果分别是 123、291、83。如果开头字符是非数值字符，则停止转换。

`parseFloat(<字符表达式>)`函数

能将数值字符串转换为浮点型数值，如果开头字符是非数值字符，则停止转换。

`parseFloat('123.45')`转换结果为 123.45。

变量`.toString` 和变量`.valueOf`

分别能将变量的数值转换为字符串，变量的数值字符转换为数值。如 `sz=123.45`，则 `sz.toString` 结果为 '123.45'。 `zf='123.45'`，则 `zf.valueOf` 结果为 123.45。

实际应用时，VRMLScript 像 JavaScript 一样有数据类型的自动转换能力，例如，`print('条件运算结果='+ (7>5?9:10))`，在控制台上显示的是：条件运算结果=9，这里 9 已经自动被转换为字符了。

8.3 对象处理

8.3.1 对象

JavaScript 与 VRMLScript 将 VRML 中的节点视为对象，节点的域和事件就是对象的属性。Script 节点自带的域和出事件可以类似相同名称的变量在脚本程序中使用，而它的入事件不能当作变量使用，因为每一个入事件都会对应一个事件响应函数。如果需要随时取用入事件的值，应当在响应函数中将值赋给一个 `field` 域，然后把这个域作为一个变量使用。要访问除了所在的 Script 节点以外的其他节点的域和事件时，该 Script 节点的 `direction` 域值必须为 `TRUE`。因为外部节点都被视为对象，访问这些节点的域和事件时可使用“.”操作符。但对于这些外部对象需要注意：`exposedfield` 域是可读可写的；`eventIn` 域是可写不可读的；`eventOut` 域是可读不可写的；`eventIn` 域（`eventOut` 出事件）保留最后一次所发送事件的值，如果未发送事件，则值为其值类型的默认值；当向外部对象 `eventIn` 和 `exposedField` 域发送事件时，将会引起与发送事件函数时间相同的外部对象的响应事件。

8.3.2 数学运算对象

尽管 JavaScript 与 VRMLScript 并不擅长数值计算，但也是不可避免的，所以这两种语言都定义了数学运算的对象。JavaScript 如何进行数学运算这里就不作介绍了，来看一下 VRMLScript 中定义的专用于数学运算的全局对象 `Math`。

数学运算中常用的一些常数，如自然对数、圆周率等都是作为 `Math` 对象的属性定义：

`e`：自然对数，约为 2.718。

`ln10`：对 10 取自然对数的结果，约为 2.302。

ln2: 对 2 取自然对数的结果, 约为 0.693。

pi: 圆周率, 约为 3.1416。

sqrt1/2: 0.5 的平方根, 约为 0.707。

sqrt2: 2 的平方根, 约为 1.414。

Math 对象的方法:

abs(number): 返回 number 的绝对值。

acos(number): 返回 number 的反余弦值, 以弧度表示。

asin(number): 返回 number 的反正弦值, 以弧度表示。

atan(number): 返回 number 的反正切值, 以弧度表示。

cos(number): 返回 number 的余弦值, 以弧度表示。

sin(number): 返回 number 的正弦值, 以弧度表示。

tan(number): 返回 number 的正切值, 以弧度表示。

ceil(number): 返回最小的大于或等于 number 的整数。

exp(number): 返回自然对数的 number 次幂。

floor(number): 返回最大的小于或等于 number 的整数。

log(number): 返回 number 的对数值。

max(number1, number2): 返回较大的数。

min(number1, number2): 返回较小的数。

pow(base, exponent): 返回 base 的 exponent 次幂。

random: 返回 0.0~1.0 之间的随机数。

round(number): 返回最接近 number 的整数。

sqrt(number): 返回 number 的平方根。

8.3.3 Browser对象

为了在脚本程序中可以获取 VRML 浏览器的信息, 以及在某些情况下改变浏览器中的场景 (并非只改变某个节点), VRMLScript 的对象中增加了 Browser 对象。它包含了以下对象方法:

getName: 获取该浏览器名称的字符串。

getVersion: 获取该浏览器版本的字符串。

getCurrentSpeed: 获取目前浏览器的航行速度。

getCurrentFrameRate: 获取该浏览器在 1s 内刷新屏幕的帧数。

getWorldURL: 获取当前场景的资源 URL 地址, 即 VRML 文件位置。

repaceWorld(nodes): 以 MFNode 对象 nodes 的内容取代当前场景。

createVrmlFromString(VRMLSyntax): 将字符串 VRMLSyntax (以 VRML 节点格式排列) 翻译为场景, 可用该方法向场景中增加节点。

createVRMLFormURL(url, node, event): 将 url 指定的资源 (即其他 VRML 文件) 翻译为场景, 可将其他 VRML 文件中的节点添加到场景中。

addRouteFromNode(fromevent, tonode, toevent): 生成事件消息传递路由, 添加到场景中。

deleteRouteFromNode(fromevent, tonode, toevent): 删除场景中的事件消息传递路由。

loadURL(url, parameter): 调入 url 指定的资源, 即链接到其他场景中。

setDescription(description): 设置场景的描述字符串, 该字符串出现在场景显示的下边。

【例 8-2】 下面演示一个电子钟, 其造型在前面已经介绍过, 在此引入脚本, 使电子钟按当前时间开始启动。

其实现的源程序代码如下:

```
#VRML V2.0 utf8
Background {
    skyColor 0 0.5 0.7
}
Shape {
    appearance Appearance {
        material Material {
            diffuseColor 0.6 0.7 0.7
        }
    }
    geometry Box {
        size 4 4 2
    }
}
DEF time1 TimeSensor {
    cycleInterval 1
    loop TRUE
    enabled TRUE
}
DEF an Transform {
    translation 0 0 1.1
    children [
        DEF ts0 TouchSensor {}
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.5 0.3 0.2
                    ambientIntensity 0.4
                    specularColor 0.7 0.7 0.5
                    shininess 0.2
                }
            }
            geometry Sphere {
                radius 0.11          #中心钮
            }
        }
    ]
}
Transform {
    translation 0 0 1.1
    children [
```

```

DEF p1 Transform {           #秒针
    translation 0 0.9 0.05
    children [
        DEF ts1  CylinderSensor {}
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.4 0.4 0.3
                    ambientIntensity 0.4
                    specularColor 0.7 0.7 0.6
                    shininess 0.2
                }
            }
            geometry Cylinder {
                height 1.62
                radius 0.03
            }
        }
    ]
}

DEF p2 Transform {           #分针
    translation 0 0.8 0
    children [
        DEF ts2  CylinderSensor {}
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.5 0.4 0.3
                    ambientIntensity 0.4
                    specularColor 0.8 0.8 0.9
                    shininess 0.1
                }
            }
            geometry Cylinder {
                height 1.5
                radius 0.05
            }
        }
    ]
}

DEF p3 Transform {           #时针
    translation 0 0.6 0
    children [
        DEF ts3  CylinderSensor {}
        Shape {
            appearance Appearance {

```

```

        material Material {
            diffuseColor 0.5 0.5 0.7
            ambientIntensity 0.2
            specularColor 0.7 0.6 0.5
            shininess 0.1
        }
    }
    geometry Cylinder {
        height 1.2
        radius 0.05
    }
}
]
}
DEF bkd Transform {
    translation 0 1.8 0
    children [
        Shape {
            appearance Appearance {
                material Material {
                    diffuseColor 0.5 0.5 0.7
                    ambientIntensity 0.4
                    specularColor 0.8 0.8 0.9
                    shininess 0.2
                }
            }
            geometry Sphere {
                radius 0.07
            }
        }
    ]
}
]
}
Transform {
    rotation 0 0 1 0.524
    children [
        USE bkd ]
    }
    Transform {
        rotation 0 0 1 1.048
        children [
            USE bkd ]
        }
        Transform {
            rotation 0 0 1 1.572
            children [
                USE bkd ]
            }
        ]
    }
}

```

```

    }
  Transform {
    rotation 0 0 1 2.096
    children [
      USE bkd ]
    }
  Transform {
    rotation 0 0 1 2.620
    children [
      USE bkd ]
    }
  Transform {
    rotation 0 0 1 3.144
    children [
      USE bkd ]
    }
  Transform {
    rotation 0 0 1 3.668
    children [
      USE bkd ]
    }
  Transform {
    rotation 0 0 1 4.192
    children [
      USE bkd ]
    }
  Transform {
    rotation 0 0 1 4.716
    children [
      USE bkd ]
    }
  Transform {
    rotation 0 0 1 5.240
    children [
      USE bkd ]
    }
  Transform {
    rotation 0 0 1 5.764
    children [
      USE bkd ]
    }
  ]
}

DEF controller Script {
  eventIn    SFTime clicked
  eventOut   SFBooleanabledt

```

```

eventOut SFFloat setenabled
field SFInt32 i 1
field SFInt32 j 1
field SFInt32 k 1
field SFInt32 on 0
field SFFloat sz -0.10472
field SFFloat mz -0.10472
field SFFloat hz -0.10472
eventOut SFTIME      miao
eventOut SFTIME      fen
eventOut SFTIME      xshi
eventIn  SFTIME sec_xz
eventIn  SFRotation xuanzhuan_sec
eventIn  SFRotation xuanzhuan_min
eventIn  SFRotation xuanzhuan_hou
field SFNode sec USE p1
field SFNode min USE p2
field SFNode hou USE p3
url      "VRMLScript:
function initialize(){
    print('点中心钮可以调用秒分时针! ');
    enabledt=TRUE;
    setenabled=TRUE;
    mydate=new Date();    #获取系统日期时间
    year=mydate.getFullYear();
    mont=mydate.getMonth();
    date=mydate.getDate();
    hour=mydate.getHours();
    minu=mydate.getMinutes();
    seco=mydate.getSeconds();
    deff1=seco;
    deff2=minu*60+seco;
print('现在日期时间:='+year+'年' +mont+'月'+date+'日'+hour+'点'+minu+'分'+seco+'秒');
    if (deff2>720){    #时针转动 1 次 720 秒
        deff2=deff2-720*Math.floor(deff2/720);
    }
    hour1=hour+minu/60;
    loop=2*3.14159*hour1/12/0.10472;
    while(k<=loop){
        xuanzhuan_hou();
        loop=2*3.14159*minu/60/0.10472+1;
    }
    minu1=minu+seco/60;    #分针初始位置确定
    while(j<=loop){
        xuanzhuan_min();
    }
}

```

```
loop=2*3.14159*seco/60/0.10472+1;
while(i<=loop){ #秒针初始位置确定
    xuanzhuan_sec();
}
setenabled=FALSE;
}
function clicked (value){ #时间调整控制开关
    enabledt=!enabledt;
    setenabled=!setenabled;
}
function sec_xz(a)
{
    a=a+hour*3600+minu*60+seco;
    if(on==0){
        on=1;
        miao=a;
        fen=a;
        xshi=a;
    }
    if(a-miao>=1){ #秒针转动 1 次 1 秒
        hd=1.57+sz * i;
        y=0.9* Math.sin(hd);
        x=0.9*Math.cos(hd);
        z=0.05;
        with (sec){
            rotation[0]=0;
            rotation[1]=0;
            rotation[2]=1;
            rotation[3]=sz*i;
            translation[0]=x;
            translation[1]=y;
            translation[2]=z;
        }
        i=i+1;
        miao=a;
    }
    if(a+deff1-fen>=60){
        hd=1.57+sz*j;
        y=0.8*Math.sin(hd);
        x=0.8*Math.cos(hd);
        z=0;
        min.rotation[0]=0;
        min.rotation[1]=0;
        min.rotation[2]=1;
        min.rotation[3]=sz*j;
        min.translation[0]=x;
```

```
        min.translation[1]=y;
        min.translation[2]=z;
        j=j+1;
        fen=a;
        deff1=0;
    }
    if (a+deff2-xshi>=720){#时针转动 1 次 720 秒
        hd=1.57+sz*k;
        y=0.6*Math.sin(hd);
        x=0.6*Math.cos(hd);
        z=0;
        hou.rotation[0]=0;
        hou.rotation[1]=0;
        hou.rotation[2]=1;
        hou.rotation[3]=sz*k;
        hou.translation[0]=x;
        hou.translation[1]=y;
        hou.translation[2]=z;
        k=k+1;
        xshi=a;
        deff2=0;
    }
}

function xuanzhuan_sec() {    #秒针位置调整
    if(setenabled){
        hd=1.57+sz*i;
        y=0.9*Math.sin(hd);
        x=0.9*Math.cos(hd);
        z=0.05;
        sec.rotation[0]=0;
        sec.rotation[1]=0;
        sec.rotation[2]=1;
        sec.rotation[3]=sz*i;
        sec.translation[0]=x;
        sec.translation[1]=y;
        sec.translation[2]=z;
        i=i+1;
    }
}

function xuanzhuan_min(){    #分针位置调整
    if(setenabled){
        hd=1.57+sz*j;
        y=0.8*Math.sin(hd);
        x=0.8*Math.cos(hd);
        z=0;
        min.rotation[0]=0;
```

```

min.rotation[1]=0;
min.rotation[2]=1;
min.rotation[3]=sz*j;
min.translation[0]=x;
min.translation[1]=y;
min.translation[2]=z;
j=j+1;
    }
}
function xuanzhuan_hou(){    #时针位置调整
    if(setenabled){
        hd=1.57+sz*k;
        y=0.6*Math.sin(hd);
        x=0.6*Math.cos(hd);
        z=0;
        hou.rotation[0]=0;
        hou.rotation[1]=0;
        hou.rotation[2]=1;
        hou.rotation[3]=sz*k;
        hou.translation[0]=x;
        hou.translation[1]=y;
        hou.translation[2]=z;
        k=k+1;
    }
}
"}
ROUTE  time1.time      TO  controller.sec_xz      #秒针转动
ROUTE  ts0.touchTime   TO  controller.clicked     #时间调整控制
ROUTE  controller.enabledt TO time1.enabled       #时间传感器控制
ROUTE  ts1.rotation_changed TO controller.xuanzhuan_sec #秒调传感器控制
ROUTE  ts2.rotation_changed TO controller.xuanzhuan_min #分调整控制
ROUTE  ts3.rotation_changed TO controller.xuanzhuan_hou #时调整控制

```

8.4 VRML与网络

VRML 是一种专门为网络而设置的虚拟现实建模语言。采用 VRML 不仅可以直接访问网络（网页），而且还可以采用 VRML 来进行建立网站，实现立体化网站的意图。

在 VRML 程序中，可以使用 Anchor 节点来实现直接打开网络资源的目的。

【例 8-3】 在 VRML 程序中实现链接到相关网站（图 8-1）。

其实现的源程序代码如下：

```

#VRML V2.0 utf8
Background {
    skyColor 0.2 0.3 0.4
}

```

```

Anchor {
    url "http://www.hao123.com/" #链接的目标网站
    children [
        Transform {
            translation 0.0 -5.0 -7.0
            children [
                Shape {
                    appearance Appearance {
                        material Material {
                            diffuseColor 1.0 0.0 0.0
                        }
                    }
                    geometry Text {
                        string ["Go to Internet www.hao123.com"] #显示的提示
                        fontStyle FontStyle {
                            family "TYPEWRITER"
                            style "BOLD"
                            size 10.0
                            horizontal TRUE
                            leftToRight TRUE
                            justify ["MIDDLE"]
                        }
                        maxExtent 60.0}}}]
            ]
        }
        Anchor { #链接的目标网络
            url "http://www.baidu.com/index.php?tn=sitehao123" #链接的目标网址
            children [
                Transform {
                    translation 0.0 10.0 -7.0
                    children [
                        Shape {
                            appearance Appearance {
                                material Material {
                                    diffuseColor 1.0 1.0 0.0 }
                                }
                            geometry Text {
                                string ["Go local net"] #显示的提示
                                fontStyle FontStyle {
                                    family "TYPEWRITER"
                                    style "BOLD"
                                    size 10.0
                                    horizontal TRUE
                                    leftToRight TRUE
                                    justify ["MIDDLE"]
                                }
                            }
                        }
                    ]
                }
            ]
        }
    ]
}

```

```
maxExtent 60.0}}}}  
}  
}
```

在这个示例中，可实现直接上 Internet 网络，也可以链接到本地网页，实现本地网页与 VRML 程序的跳转。



图 8-1 在 VRML 程序中访问网站

参考文献

- [1] 黄文丽, 卢碧红, 杨志刚, 等. VRML 语言入门与应用. 北京: 中国铁道出版社, 2003.
- [2] 张金钊, 张金锐, 张金镛, 等. VRML 编程实训教程. 北京: 清华大学出版社/北京交通大学出版社, 2007.
- [3] 张金钊, 张金锐, 张金镛. 虚拟现实三维立体网络程序设计语言 VRML: 第二代网络程序设计语言. 北京: 北京交通大学出版社, 2004.
- [4] 刘怡, 张洪定, 崔欣. 虚拟现实 VRML 程序设计. 天津: 南开大学出版社, 2007.
- [5] 胡小强. 虚拟现实技术基础与应用. 北京: 北京邮电大学出版社, 2009.
- [6] 魏迎梅, 栾悉道. 虚拟现实技术 (第 2 版). 北京: 电子工业出版社, 2005.
- [7] 洪炳镕, 蔡则芬, 唐好选. 虚拟现实及其应用. 长沙: 国防科技大学出版社, 2005.
- [8] 石教英. 虚拟现实基础及实用算法. 北京: 科学出版社, 2002.
- [9] 周正平. 使用 VRML 与 JAVA 创建网络虚拟环境. 北京: 北京大学出版社, 2003.
- [10] 韦有双, 杨湘龙, 王飞. 虚拟现实与系统仿真. 北京: 国防工业出版社, 2004.
- [11] 汤跃明. 虚拟现实技术在教育中应用. 北京: 科学出版社, 2007.
- [12] 张金钊. 虚拟现实三维立体网络程序设计 VRML. 北京: 清华大学出版社/北京交通大学出版社, 2004.
- [13] 严子翔. VRML 虚拟现实网页语言. 北京: 清华大学出版社, 2001.
- [14] 张家祥, 方凌江, 毛全胜. 基于 MATLAB6. X 的系统分析与设计——虚拟现实. 西安: 西安电子科技大学出版社, 2002.
- [15] 张茂军. 虚拟现实系统. 北京: 科学出版社, 2001.
- [16] 申蔚, 夏立文. 虚拟现实技术. 北京: 北京希望电子出版社, 2002.