信息科学与技术丛书

Visual C++ CAD 应用程序开发技术

王清辉　王　彪　编著

机械工业出版社

本书系统地阐述了在 MFC 与 OpenGL API 的集成开发环境下，用面向对象的技术开发三维 CAD 软件的有关知识与方法。内容包括：总体程序框架的分析与设计、功能模块的划分、相关 DLL 库的开发与使用、CAD 基础几何类库的开发、在 MFC 环境下使用 OpenGL 进行图形绘制、开发面向 CAD 应用程序的 OpenGL 通用绘图类、使用面向对象技术设计 CAD 软件的几何内核、CAD 软件的图形交互、软件界面设计等。在介绍过程中，以一个完整的三维 CAD 软件（STLViewer）的开发实例贯穿于全书各章节，并附有完整的 Visual C++实现代码。全书面向开发实例进行分析与介绍，讲解透彻，易于理解。读者通过对本书的阅读和学习，能够掌握使用 Visual C++进行具有一定复杂程度的软件的设计与实现方法。

本书适合于从事图形及 CAD 软件开发的技术人员和具有一定 Visual C++基础的大专院校相关专业学生阅读。

本书提供的开发实例可在 Visual C++ 6.0 版本上实现。

图书在版编目（CIP）数据

Visual C++ CAD 应用程序开发技术/王清辉，王彪编著. －北京：机械工业出版社，2003.7
（信息科学与技术丛书）
ISBN 7-111-12383-2

Ⅰ. V… Ⅱ. ①王…②王… Ⅲ. ①C 语言－程序设计②计算机辅助设计
Ⅳ. ①TP312②TP391.72

中国版本图书馆 CIP 数据核字（2003）第 047133 号

机械工业出版社（北京市百万庄大街 22 号　邮政编码 100037）
策　　划：胡毓坚
责任编辑：刘　青
责任印制：

·新华书店北京发行所发行

2003 年 7 月第 1 版·第 1 次印刷
787mm×1092mm 1/16·17.5 印张·432 千字
0 001－5 000 册
定价：32.00 元（含 1CD）

凡购本图书，如有缺页、倒页、脱页，由本社发行部调换
本社购书热线电话（010）68993821、88379646
封面无防伪标均为盗版

出 版 说 明

随着信息科学与技术的迅速发展，人类每时每刻都会面对层出不穷的新技术、新概念。毫无疑问，在节奏越来越快的工作和生活中，人们需要通过阅读和学习大量信息丰富、具备实践指导意义的图书，来获取新知识和新技能，从而不断提高自身素质，紧跟信息化时代发展的步伐。

众所周知，在计算机硬件方面，高性价比的解决方案和新型技术的应用一直备受青睐；在软件技术方面，随着计算机软件的规模和复杂性与日俱增，软件技术受到不断挑战，人们一直在为寻求更先进的软件技术而奋斗不止。目前，计算机在社会生活中日益普及，随着因特网延伸到人类世界的层层面面，掌握计算机网络技术和理论已成为大众的文化需求。正是这种在社会各领域的全方位渗透，信息科学与技术正在电工、电子、通信、工业控制、智能建筑、工业产品设计与制造等专业领域中得到充分、广泛的应用。相应地，这些专业领域中的研究人员和工程技术人员将越来越迫切需要汲取自身领域信息化所带来的新理念和新方法。

针对人们对了解和掌握新知识、新技能的热切期待，以及由此促成的人们对语言简洁、内容充实、融合实践经验的图书迫切需要的现状，机械工业出版社适时推出了“信息科学与技术丛书”。这套丛书涉及计算机软件、硬件、网络、工程应用等内容，注重理论与实践相结合，内容实用，层次分明，语言流畅，是信息科学与技术领域专业人员学习和参考不可或缺的图书。

现今，信息科学与技术的发展可谓一日千里，机械工业出版社欢迎从事信息技术方面工作的科研人员、工程技术人员积极参与我们的工作，为推进我国的信息化建设作出贡献。

机械工业出版社

前　言

CAD 软件的开发是软件开发中的一个重要领域。开发一个三维 CAD 软件所涉及的知识面较多，如何规划和展开软件的开发是系统开发成功与否的关键问题之一。本书的特点在于，从完整的、模块化结构的软件设计与实现的角度逐步讲述一个三维 CAD 软件从系统设计到使用 Visual C++编程实现的各个主要环节。着重于讲述使用 Visual C++、OpenGL 进行图形及 CAD 软件开发的有关技术问题，将所涉及到的许多技术难点融入到具体的开发实例中，使读者易于理解和掌握。在讲述中，以一个三维 CAD 原型系统 STLViewer 的开发为主线，贯穿全书，并提供详细的代码注解。

Visual C++是 Microsoft 公司迄今开发的功能最为强大的软件开发工具，是新一代 CAD 软件的主要开发平台。随着面向对象程序设计技术的广泛应用，Visual C++优秀的开发环境、Microsoft 推出的 Microsoft Foundation Class （MFC）以及 MFC 程序框架、Windows 操作系统对 OpenGL 的支持等，为在 Windows 系统上开发三维 CAD 软件提供了极大的方便。读者通过本书的学习，将能够循序渐进地了解和掌握使用 Visual C++开发三维图形及 CAD 应用软件的相关技术。通过对应用实例的具体剖析，希望能使读者从软件开发的思想方法上对面向对象的程序设计技术有更深入的了解。

本书内容共分九章，分别介绍了基于 MFC 的总体程序框架的分析与设计；功能模块的划分以及相关 DLL 库的开发与使用；CAD 的基础几何类库的开发；在 MFC 环境下使用 OpenGL 进行图形绘制；开发面向 CAD 应用程序的 OpenGL 通用绘图类；使用面向对象技术设计 CAD 软件的几何内核；CAD 软件的图形交互；软件界面设计等。

本书所附带的光盘中，根据每章讲述的内容提供了全部的实现代码。本书的代码适用于 Visual C++ 6.0 以上版本。

本书的内容是作者长期从事 CAD 系统设计和开发的一些经验的提炼和总结。在内容或方法上若有疏漏和不妥之处，恳请各位读者给予指正。作者的电子邮件地址：wangqh15@yahoo.com.sg。

作　者

目　录

出版说明
前言
第 1 章　基于 MFC 的三维 CAD 应用程序框架结构分析 …… 1
1.1　MFC 应用程序的文档/视图结构 …… 2
1.1.1　文档/视图结构概述 …… 2
1.1.2　文档与多个视图的关联 …… 2
1.1.3　文档模板及主要组成类 …… 3
1.2　实例分析——三维 CAD 示例软件 STLViewer …… 6
1.3　STLViewer 的程序框架 …… 6
1.4　STLViewer 中类的层次设计及软件模块结构划分 …… 11
1.5　建立和使用动态链接库 …… 12
1.5.1　动态链接库的基本概念 …… 13
1.5.2　基于 MFC 的动态链接库 …… 13
本章相关程序 …… 14
第 2 章　几何基本工具库的开发 …… 15
2.1　点、矢量和齐次变换矩阵 …… 15
2.1.1　点 …… 15
2.1.2　矢量 …… 16
2.1.3　齐次坐标与齐次变换矩阵 …… 17
2.2　设计点、矢量和齐次变换矩阵类 …… 18
2.2.1　点类 CPoint3D …… 19
2.2.2　矢量类 CVector3D …… 22
2.2.3　变换矩阵类 CMatrix3D …… 25
2.3　三维图形的几何变换 …… 28
2.3.1　三维齐次变换矩阵 …… 28
2.3.2　平移变换 …… 29
2.3.3　旋转变换/绕空间任意轴的旋转变换函数的实现 …… 29
2.3.4　几何缩放 …… 32
2.3.5　对称变换/沿空间任意平面的对称变换函数的实现 …… 33
2.4　设计几何基本工具库 GeomCalc.dll …… 34
2.4.1　GeomCalc.dll 中的输出类与输出函数 …… 34
2.4.2　创建几何基本工具库 GeomCalc.dll 的步骤 …… 35
2.4.3　使用 GeomCalc.dll …… 37
2.5　有关源程序代码 …… 37
2.5.1　文件 CadBase.h …… 37
2.5.2　文件 CadBase.cpp …… 40
2.5.3　文件 CadBase1.cpp …… 49
本章相关程序 …… 50
第 3 章　基于 MFC 的 OpenGL Windows 程序的创建 …… 51
3.1　OpenGL 介绍 …… 51
3.2　在 Windows 环境下使用 OpenGL …… 52
3.2.1　OpenGL 的函数库 …… 52
3.2.2　OpenGL 与 GDI …… 53
3.2.3　渲染场境 …… 53
3.2.4　像素格式 …… 56
3.3　OpenGL MFC 应用程序创建实例 …… 60
3.3.1　创建一个应用程序框架 …… 61
3.3.2　修改视图类 CGLView …… 63
3.3.3　使用 OpenGL 的双缓存技术为应用程序增加动画效果 …… 69
3.4　程序清单 …… 71
3.4.1　文件 GLView.h …… 71
3.4.2　文件 GLView.cpp …… 72
本章相关程序 …… 78
第 4 章　封装 OpenGL 功能的 C++类的设计 …… 79
4.1　封装 OpenGL 的 C++类的设计 …… 79
4.2　照相机类 GCamera 的设计 …… 80

4.2.1 视点坐标系和视图变换 …… 80
4.2.2 投影变换与视景体 …… 82
4.2.3 视口变换 …… 83
4.2.4 设计照相机类 GCamera …… 85
4.3 类 COpenGLDC …… 88
4.4 修改类 CGLView …… 93
4.5 运行应用程序 …… 96
4.6 源程序清单 …… 96
4.6.1 类 GCamera 的声明代码 …… 96
4.6.2 类 GCamera 的实现代码 …… 97
4.6.3 类 COpenGLDC 的声明代码 …… 99
4.6.4 类 COpenGLDC 的实现代码 …… 100
4.6.5 类 CGLView 的声明代码 …… 103
4.6.6 类 CGLView 的实现代码 …… 104
本章相关程序 …… 106
第 5 章 基于 OpenGL 的 CAD 图形工具库的设计 …… 107
5.1 创建动态链接库 glContext.dll …… 107
5.2 类 GCamera 的功能增强 …… 108
5.2.1 选择典型的观察视图 …… 110
5.2.2 景物平移 …… 113
5.2.3 景物缩放 …… 114
5.2.4 使用 OpenGL 的选择模式 …… 115
5.3 类 COpenGLDC 功能的增强 …… 116
5.3.1 实现和 Windows 窗口的关联 …… 120
5.3.2 定义光源 …… 120
5.3.3 定义颜色 …… 126
5.3.4 图形绘制函数 …… 131
5.3.5 选择模式 …… 136
5.4 增加类 CGLView 中的功能 …… 137
5.5 glContext 类的输出和调用 …… 140
5.6 源程序清单 …… 141
5.6.1 文件 Camera.h（类 GCamera） …… 141
5.6.2 文件 Camera.cpp（类 GCamera） …… 142
5.6.3 文件 OpenGLDC.h（类 COpenGLDC、CGLView） …… 147
5.6.4 文件 OpenGLDC.cpp（类 OpenGLDC） …… 151
5.6.5 文件 GLView.cpp（类 CGLView） …… 160
本章相关程序 …… 163
第 6 章 CAD 应用程序的几何内核模块的设计 …… 164
6.1 几何对象类的设计 …… 164
6.1.1 类的层次设计 …… 164
6.1.2 几何对象基本类 CEntity …… 167
6.1.3 三角面片对象类 CTriChip …… 170
6.1.4 STL 几何模型类 CSTLModel …… 171
6.1.5 高级几何模型类 CPart …… 176
6.2 串行化(Serialize)实现文档存取功能 …… 179
6.2.1 为什么要使用串行化 …… 179
6.2.2 CArchive 类 …… 179
6.2.3 串行化类的设计步骤 …… 180
6.2.4 CObArray 的 Serialize()函数 …… 181
6.2.5 应用程序的文档串行化实例剖析 …… 181
6.3 虚拟函数 …… 184
6.3.1 虚拟函数与多态性 …… 184
6.3.2 纯虚拟函数 …… 186
6.3.3 实现 CPart 模型的 OpenGL 显示 …… 186
6.4 建立几何内核库 GeomKernel.dll …… 188
6.5 程序清单 …… 189
6.5.1 文件 Entity.h …… 189
6.5.2 文件 Entity.cpp …… 192
本章相关程序 …… 200

第 7 章　CAD 应用程序 STLViewer 的模块化实现 …… 201
7.1　STLViewer 的模块结构 …… 201
7.2　创建应用程序框架 …… 202
7.3　修改应用程序框架 …… 205
7.3.1　增加界面资源 …… 205
7.3.2　修改框架类 CMainFrame …… 206
7.3.3　修改文档类 CSTLViewerDoc …… 207
7.3.4　修改视图类 CSTLViewerView …… 209
7.4　运行 STLViewer.exe …… 215
7.5　源程序清单 …… 216
7.5.1　文件 MainFrm.h …… 216
7.5.2　文件 MainFrm.cpp …… 217
7.5.3　文件 STLViewerDoc.h …… 219
7.5.4　文件 STLViewerDoc.cpp …… 220
7.5.5　文件 STLViewerView.h …… 222
7.5.6　文件 STLViewerView.cpp …… 224
本章相关程序 …… 228
第 8 章　增强 CAD 应用程序的界面功能 …… 229
8.1　STLViewer 的界面增强 …… 229
8.2　工具栏的排列 …… 230
8.3　使用快捷菜单 …… 232
8.4　创建类似 Visual Studio 风格的浮动窗口 …… 232
8.4.1　控制条与停靠栏 …… 233
8.4.2　开发具有 Visual Studio 风格的浮动窗口 …… 235
8.4.3　CTabCtrl 控件的功能增强 …… 237
8.4.4　建立界面工具库 DockTool.dll …… 243
8.5　使用树型视图 CtreeView 显示和管理文档数据 …… 243
8.5.1　树型视图与树型控件 …… 244
8.5.2　在 STLViewer 中创建 CPartTreeView …… 244
8.5.3　树型视图 CPartTreeView 与文档的关联/文档多视图 …… 246
8.5.4　在树型控件中使用图标 …… 248
8.5.5　使用树型视图控件显示文档中几何模型的结构和属性 …… 249
8.5.6　通过树型视图控件对文档数据进行操作 …… 251
本章相关程序 …… 253
第 9 章　基于 OpenGL 的 CAD 软件拾取功能的实现 …… 255
9.1　使用 OpenGL 选择模式 …… 256
9.1.1　OpenGL 的三种操作模式 …… 256
9.1.2　使用选择模式 …… 256
9.2　一个 OpenGL 选择模式的应用程序 …… 262
9.3　OpenGL 的选择功能与 CAD 应用程序的集成 …… 266
9.3.1　定义选择视景体/修改类 CCamera …… 266
9.3.2　对选择过程的操作/修改类 COpenGLDC …… 267
9.3.3　自动给对象命名/对类 CSTLModel 的修改 …… 269
9.3.4　在 STLViewer 中调用拾取功能 …… 270
9.3.5　运行程序 …… 271
本章相关程序 …… 271

第 1 章　基于 MFC 的三维 CAD 应用程序框架结构分析

本章要点：

- MFC 应用程序的文档/视图结构概述。
- 三维 CAD 应用程序的模块化结构分析。
- 动态链接库的创建与使用。

在 Visual C++ 2.0 以后的版本中，Microsoft 公司推出了 MFC（Microsoft Foundation Class）类库。MFC 类库是用来编写 Windows 程序的 C++类集。使用 MFC 类库，可以简化应用程序的开发，从而缩短开发周期，而且代码的可靠性和可重用性也大大提高。很多读者可能已经有了使用 MFC 的经验，并对 MFC 编程有了浓厚的兴趣。这里，我们首先从分析 MFC 的应用程序框架开始，分析如何采用面向对象的技术来设计 CAD 软件的程序框架和软件模块结构。

什么是应用程序框架？一种定义是“提供一个一般应用程序所需要的全部面向对象的软件组件的集合”。应用程序框架设计得好坏与否，直接决定了程序功能的实现难易程度和软件开发与维护的代价高低。任何一个应用程序从本质上来说都是对数据的操作，因此，一个好的程序框架结构就意味着对应用数据管理上的友好安全和处理上的简便通用。应用程序框架的实例很多，工程应用的计算机辅助设计/制造/分析（CAD/CAM/CAE：Computer Aided Design/ Manufacturing/ Engineering）软件多采用文档与视图相结合的程序框架。本章将结合 MFC 提供的应用程序的文档/视图结构这一工程软件的常见框架进行阐述和分析。为了便于阐述，在本书中，作者将一个应用于快速原型制造系统（RPM：Rapid Prototyping Manufacturing）的 CAD/CAM 软件作了大量简化，去除了主要的专业功能，使之成为一个浅显易懂的三维 CAD 示例软件，并贯穿全书，整个软件系统和模块设计均采用了面向对象的编程技术。在本章中，将具体分析这一三维 CAD 软件的模块化结构设计，使读者对工程 CAD 软件的应用程序框架设计有一定的理解和认识，同时，本章也在示例中讲解了基于 MFC 的有关动态链接库的具体创建和使用过程，以期加强读者的理解和实际开发能力。

以下的各章中，通过循序渐进的介绍，读者将可以对以下主要技术有所掌握：

（1）面向对象的编程技术。
（2）软件的结构及模块化设计。
（3）动态链接库的开发与应用。
（4）几何计算基础类库的开发。
（5）CAD 软件的界面设计和交互操作。
（6）CAD 系统的几何内核设计。
（7）基于 OpenGL 的 CAD 模型显示。
（8）开发面向 Windows 应用程序的 OpenGL C++类库。

1.1 MFC 应用程序的文档/视图结构

1.1.1 文档/视图结构概述

MFC 提供了一个典型且实用的基于文档与视图的应用程序框架模板，按照其应用程序生成向导的导引步骤（MFC AppWizard）就可以创建一个基于文档/视图结构的 MFC 应用程序框架。在此框架的基础上，设计和插入相关的对象，就可以实现交互式的用户界面、几何模型的管理和操作、图形图像的显示，以及其他各种专业功能。

在 MFC 的文档/视图结构的应用程序框架中，文档类和视图类是成对出现的。文档用于管理应用程序的数据；而视图用于显示文档中的数据，并处理与用户的交互信息。MFC 通过文档类和视图类的划分，使数据的存储和显示既相对独立又相互关联。

在 MFC 所提供的框架结构中，文档与视图的关系可以由图 1-1 简要表示。如图 1-1 所示，MFC 中的视图和文档是由视图类（CView Class）和文档类（CDocument Class）分别表示的。视图类可以调用其本身的成员函数 GetDocument()，获得一个指向文档类的指针，从而能够访问文档类中的数据。例如，在视图类的 OnDraw()函数中，视图类通过调用 GetDocument()函数获得一个指向文档类的指针。然后，通过这个指针获取文档类中的数据，并使用 CDC 类（负责处理应用程序显示设备接口的 MFC 类）中的函数将这些数据绘制在视图窗口中。视图可以通过图形、图像、文字、表格等多种方式以视图对象（CView Object）来显示实际文档（文档对象 CDocument Object）中的数据。同时，视图对象也负责接收鼠标、键盘等用户输入信息，并通过这些与用户之间的交互信息来操作和修改文档中的数据。

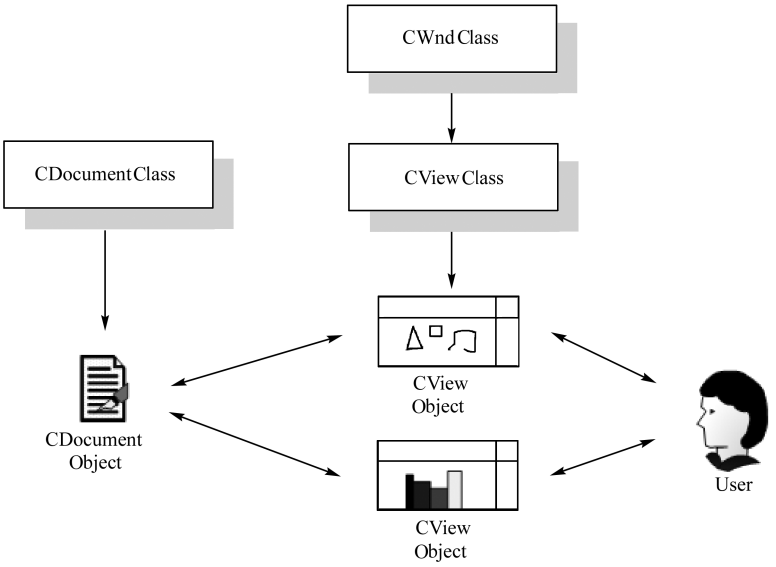


图 1-1 文档与视图的关系

1.1.2 文档与多个视图的关联

使用 MFC AppWizard 自动创建的 MFC 应用程序中，为每个文档类的对象创建并关联了

一个视图类的对象。实际上，文档和视图的关系可以是一对一或一对多，即一个文档可以关联一个或一个以上的视图，也就是可以用多个视图来显示和操作文档中的数据。通常，设计不同的视图是为了以不同的方式来显示文档内容，如图形、图表、结构、文字等。如图 1-1 所示，一个文档对象就同时关联了多个视图对象。当在一个视图类对象中对文档数据作了修改后，可以调用文档类的 UpdateAllViews()函数来更新所有与文档相关联的视图类对象的显示，以此来保持所有视图对同一文档数据变化的同步显示。一个文档关联多个视图为应用程序提供了很多的方便。例如，程序中可以使用两个视图，分别以表格和图形的方式来显示文档中的数据，图形显示给用户直观的感觉，而用户可以通过表格来访问和操作文档中的具体数据。这种方式在 CAD 应用程序中也较常用。在 CAD 程序中，使用一个大的视图窗口用于显示几何模型的同时，通常还会使用一些其他的视图以不同的方式来显示文档数据，如使用一个树型结构来显示模型的组成结构和属性信息等。这样的多视图显示的例子可以参见图 1-2。该图显示的是本书中使用的 CAD 示例软件——STLViewer 的主界面。在图形用户界面（GUI：Graphic User Interface）中，左侧浮动窗口是一个树型结构的视图（CTreeView）类的对象，用于显示几何模型的结构与属性；右侧的视图窗口用于模型的 OpenGL 图形绘制和交互操作。这两个视图对象都与同一文档对象（存储几何模型的信息）相关联。在本书第 8 章还将进一步介绍单文档多视图关系的实现及有关内容。

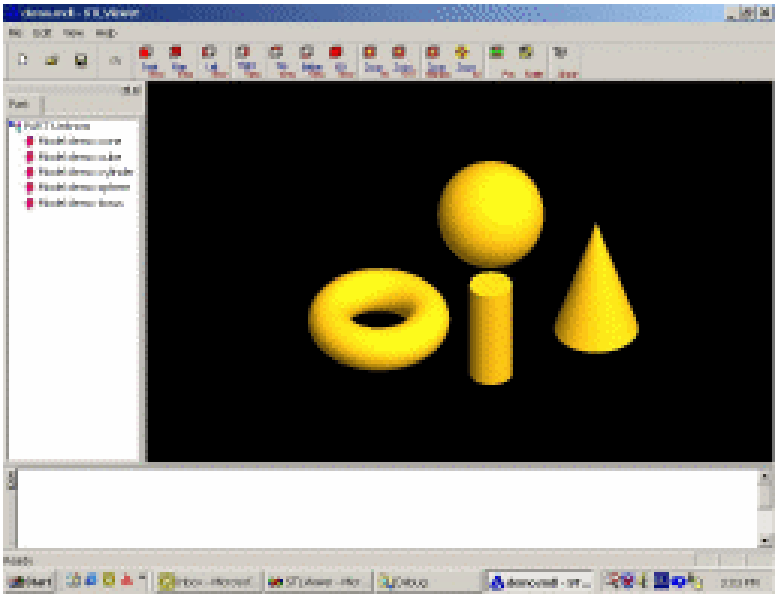


图 1-2　STLViewer 的主界面

1.1.3　文档模板及主要组成类

MFC 中，文档/视图的结构关系是由文档模板（Document Template）定义的。文档模板用于存放与应用程序文档、视图和框架窗口有关的信息。MFC 类库提供有两种文档模板类，即用于单文档（SDI）应用程序的 CSingleDocTemplate 和用于多文档（MDI）应用程序的 CMultiDocTemplate 。 CSingleDocTemplate 每次只能创建并管理一个文档，而

CMultiDocTemplate **可以创建并管理多个文档。图** 1-3**、图** 1-4 **分别显示了基于** MFC **单文档和多文档的模板结构。**

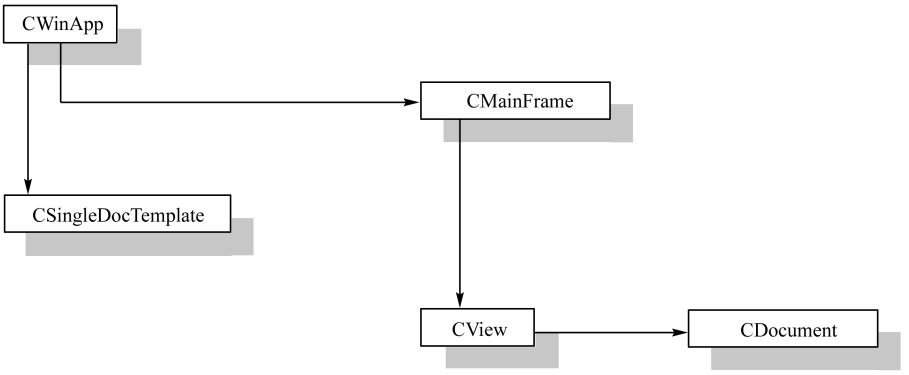


图 1-3　MFC **的单文档应用程序结构**

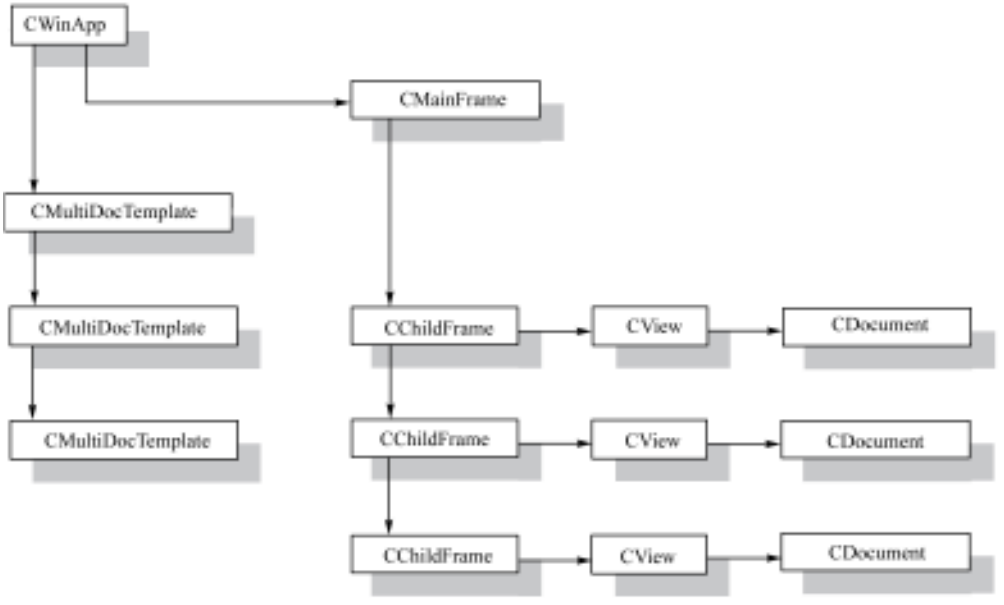


图 1-4　MFC **的多文档应用程序结构**

文档模板的创建和维护是在 MFC **的应用程序类** CWinApp **的对象中实现的。在类** CWinApp **调用成员函数** InitInstance()**进行初始化时，必须为应用程序创建一个文档模板对象，并使用函数** AddDocTemplate()**向新创建的应用程序加载这个模板。在应用程序对象中，也可以创建并加载多个文档模板。下面列出的代码是在应用程序类** CSTLViewerApp（CWinApp **的派生类）的成员函数** InitInstance()**中创建并加载一个单文档模板的例子。**

```
BOOL CSTLViewerApp::InitInstance()
{
    ……
    CSingleDocTemplate* pDocTemplate;
    pDocTemplate = new CsingleDocTemplate(
```

```
        IDR_MAINFRAME,
        RUNTIME_CLASS(CSTLViewerDoc),
        RUNTIME_CLASS(CmainFrame),          // main SDI frame window
        RUNTIME_CLASS(CSTLViewerView));
    AddDocTemplate(pDocTemplate);
    ……
}
```

由文档模板定义的 MFC **文档和视图程序框架包括四个主要的类：**

- **文档类** CSTLViewerDoc。
- **视图类** CSTLViewerView。
- **主框架类** CMainFrame。
- **应用程序类** CSTLViewerApp。

应用程序的主要功能实现分配在这四个主要的类中。MFC AppWizard **为每个类自动生成了源文件。以此为基础，开发人员进一步在这些类中插入需要的代码和对象，就可以实现软件的专业功能。下面将对这四个类的主要功能分别加以介绍。**

（1）应用程序类。应用程序类由 MFC **提供的程序基类** CWinApp **所派生，负责从总体上实现对程序的管理。例如初始化程序以及进行最后的程序清除工作。应用程序类管理从程序开始到结束的全过程。**MFC **的文档/视图结构、程序的主窗口都在应用程序类中定义和创建。每个** MFC **应用程序都必须生成一个** CWinApp **派生类的对象。**

（2）文档类。文档类是由 MFC **提供的文档基类** CDocument **所派生，负责存放应用程序的数据，并管理程序和磁盘文档文件之间的存储与读取。在建立一个** CAD **系统时，几何模型的数据通常是文档中最主要的数据，应该存放在文档类中。文档类把数据处理从界面中分离出来，同时提供一个与其他对象交互的接口。**

（3）视图类。视图类是由 MFC **基类** CView **所派生，负责显示文档类中的数据。显示的设备可以是计算机屏幕，也可以是打印机或其他设备。视图类还负责处理用户的交互输入，它可以处理多种类型的输入命令，如键盘输入、鼠标输入、菜单以及工具栏命令等。在设计一个** CAD **系统时，屏幕上的图形绘制、打印机的绘图等功能都需要在视图类中开发完成。**

不仅如此，视图类还可通过 GetDocument()**函数来获取文档指针，用于读取和操作文档中的数据，从而实现对文档数据的修改。如前面所提及的，当文档中数据发生变化时，可通过调用** CDocument::UpdateAllViews()**函数来更新视图的显示内容。**

需要特别说明的是，CView **是个虚拟类，虚拟类中由于包含了纯虚函数（例如** CView::OnDraw()**就是一个纯虚函数），它本身不能直接用于声明对象，但在** CView **中封装了许多对视图进行操作的成员函数，可以被其派生类使用。**

（4）主框架类。主框架窗口也就是应用程序的主窗口，是由主框架类管理。如图 1-3**、图** 1-4 **所示，单文档应用程序和多文档应用程序的主框架类的** MFC **基类是不同的。对单文档应用程序（**SDI**），主框架窗口的基类为** CFrameWnd **类；而在多文档应用程序（**MDI**）的情况下，其框架窗口所继承的类为** CMDIFrameWnd **类，文档框架窗口所继承的类则为** CMDIChildWnd **类，每个文档都有一个文档框架窗口。在用户界面上，一个文档框架窗口至少含有一个视图以显示该文档数据。**

主框架窗口还负责管理用户界面对象，如菜单、工具条、状态栏等。每个框架窗口对应

管理一个可选的加速键表，改变加速键表的内容就可以自动转换键盘的加速键，这样可以方便地定义调用菜单命令的加速键。

主框架窗口同时管理所有的视图窗口，并跟踪当前活动的视图。当框架窗口含有多个视图窗口时，当前视图就是最近使用的视图。当视图活动改变时，边框窗口通过调用 CView 的成员函数 OnActiveView()来通知当前视图。例如，在如图 1-2 所示的应用程序界面中，主框架窗口就同时管理了三个视图，分别是左边的树型视图、右边的 OpenGL 图形显示视图和下边的信息输出视图。

1.2 实例分析——三维 CAD 示例软件 STLViewer

本书将结合一个三维 CAD 软件 STLViewer 的设计与开发，讲述在 MFC 环境下使用面向对象的方法设计三维 CAD 软件的一些技术。包括总体程序框架的分析与设计、功能模块的划分以及相关 DLL 库的开发、如何在 MFC 环境下使用 OpenGL 进行图形绘制、为 CAD 应用程序开发 OpenGL 的通用绘图类、使用面向对象技术设计 CAD 软件的几何内核、CAD 软件的图形交互、软件的界面设计等。

这里，首先介绍一下 STLViewer 的主要功能。图 1-2 是 STLViewer 的主界面。作为一个简化了的三维 CAD 软件，STLViewer 可以接受输入的 STL 几何模型，并将 STL 文件转化为系统自定义的几何模型，修改并增加属性，还可以采用串行化（Serialize）方法存储和装载系统自定义的几何模型（*.mdl 文件）。在 STLViewer 中，使用 OpenGL 对几何模型进行了三维真实感渲染，并对模型进行视角变换、显示缩放、光照设置以及鼠标拾取等。在界面中，设计了具有 Visual Studio 界面风格的浮动窗口和信息输出框。浮动窗口采用树型控件，用于显示几何模型结构、修改模型的附加信息（颜色、名称等）和操作选取几何模型。信息输出框用于输出系统提示信息，类似于 Visual Studio 开发环境中的 Output 窗口。

STL 文件，即立体光造型文件（STL File, Stereo Lithographic File），是描述三维几何形状的标准文件格式之一，它采用一系列离散的三角片来描述三维曲面形状。目前，大多数 CAD 软件都具有了 STL 文件输出接口，可以将三维几何模型输出成 STL 文件。本书所附带的 STL 文件（ASCII 格式）均是从 AutoCAD R14 中输出的。STL 文件在工业领域有较多的应用，如快速原型制造（RPM，Rapid Prototyping Manufacturing）就使用 STL 文件作为加工模型的几何输入接口。贯穿本书的示例程序 STLViewer 就是作者对参与开发的一个快速原型制造软件作了大量简化后的结果。

1.3 STLViewer 的程序框架

图 1-5 显示了 STLViewer 的 MFC 文档/视图结构框架。STLViewer 采用 MFC 提供的单文档模板，其中包含四个主要的应用程序类：

- 文档类 CSTLViewerDoc。
- 视图类 CSTLViewerView。
- 主框架类 CMainFrame。
- 应用程序类 CSTLViewerApp。

它们所对应的源文件分别是：STLViewerDoc.h，STLViewerDoc.cpp，STLViewerView.h，STLViewerView.cpp，MainFrm.h，MainFrm.cpp，STLViewerApp.h，STLViewerApp.cpp。**所有本书提及的源程序** Visual C++**代码都在随书附带的光盘内。**

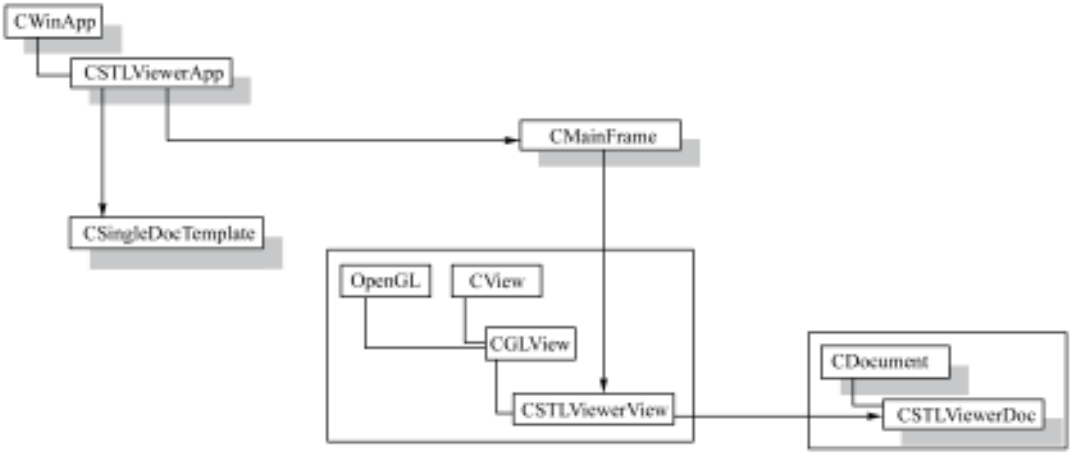


图 1-5　STLViewer **的文档/视图结构**

下面将对这四个应用程序类分别加以分析。

（1**）文档类** CSTLViewerDoc**。文档类负责管理应用程序的数据。在** STLViewer **中，**CSTLViewerDoc **由** MFC **的文档基类** CDocument **派生，并插入了存放几何模型的对象** m_Part**。**m_Part **由一个自定义的几何类** CPart **定义，用于存储和管理全部的几何模型。**

CSTLViewerDoc **的定义如下（详见本书附带光盘内的** STLViewerDoc.h **文件）：**

```
class CSTLViewerDoc : public CDocument
{
protected: // create from serialization only
        CSTLViewerDoc();
        DECLARE_DYNCREATE(CSTLViewerDoc)

// Attributes
public:
        CPartm_Part;              //几何模型对象

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CSTLViewerDoc)
        public:
        virtual BOOL OnNewDocument();
        virtual void Serialize(CArchive& ar);
```

```
        //}}AFX_VIRTUAL

    // Implementation
    public:
        virtual ~CSTLViewerDoc();
    #ifdef _DEBUG
        virtual void AssertValid()const;
        virtual void Dump(CDumpContext& dc)const;
    #endif

    protected:
    // Generated message map functions
    protected:
        //{{AFX_MSG(CSTLViewerDoc)
        afx_msg void OnStlFilein();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
    };
```

对 m_Part **进行串行化存储和读取操作由** CSTLViewerDoc **的串行化函数** Serialize()**实现。用** m_Part **代表的几何模型可以以二进制文件的形式存储在磁盘上，或从一个磁盘文件中读取出来并创建该对象。**

CPart **由一个自行开发的动态链接库** GeomKernel.dll **输出。关于** CPart **类的描述，将在本书第** 6 **章开发几何内核库时详细介绍。**

（2**）视图类** CSTLViewerView**。视图类负责文档类中数据的显示，以及负责处理用户与图形窗口之间的交互操作。**CSTLViewerView **在** CGLView **基础上派生，用于在** OpenGL **的三维环境下绘制和操作几何模型对象** m_Part**。通常，应用程序的视图类都是由** MFC **提供的基类** CView **派生得来。但由于** CView **作为一个视图窗口的基础类，没有直接调用** OpenGL **绘制三维几何图形的功能。在本书开发的** CAD **图形显示动态链接库** glContext.dll **中，专门开发了支持** OpenGL **的视图类** CGLView**，它由** CView **类派生，并采用面向对象技术封装了** OpenGL **的图形绘制功能。**CGLView **的设计将在以后的章节中详细论述。由于在类** CGLView **中已经实现了主要的模型显示和操作功能，类** CSTLViewerView **所承担的任务更多的是接受用户输入信息，并调用** CGLView **中的相关函数处理用户输入。**

CSTLViewerView **的定义如下（详见本书附带光盘内的** STLViewerView.h **文件）：**

```
class CSTLViewerView : public CGLView
{
protected: // create from serialization only
    CSTLViewerView();
    DECLARE_DYNCREATE(CSTLViewerView)

// Attributes
public:
```

```
        CSTLViewerDoc* GetDocument();

        virtual void RenderScene(COpenGLDC* pDC); //  用 OpenGL 显示几何体

    // Operations
    public:

    // Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CSTLViewerView)
        public:
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

    // Implementation
    public:
        virtual ~CSTLViewerView();
    #ifdef _DEBUG
        virtual void AssertValid()const;
        virtual void Dump(CDumpContext& dc)const;
    #endif

    protected:

    // Generated message map functions
    protected:
        //{{AFX_MSG(CSTLViewerView)
    //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

        virtual BOOL    GetBox(double& x0,double& y0,double& z0,
                    double& x1,double& y1,double& z1); //  获取视图中显示模型的最大包围盒

    };
```

视图类中的虚拟函数 CSTLViewerView::RenderScene()**在基类** CGLView::OnDraw()**中被调用，用于** OpenGL **的场景绘制。**

（3）主框架类 CMainFrame**。主框架类提供了文档界面的主窗口，并创建和管理系统菜单、浮动工具条和状态条等界面对象。程序** STLViewer **中，在** MFC AppWizard **生成的界面基础上进一步增强了用户界面的功能，即在主框架类** CMainFrame **中插入了一个具有** Visual Studio **界面风格的浮动窗口（图** 1-2 **所示界面中左边的包含树型视图的窗口）对象和一个信息输出框对象（图** 1-2 **所示界面中底部的信息输出窗口），分别用于几何模型的浏览操作和提示信息的输出。程序示例中给出了具体的程序实现。浮动窗口和信息输出框对象是分别由对象** m_LeftDockBar **和** m_OutputDockBar **对应表示的。在本书第** 8 **章将介绍一个用于界面增强的**

动态库 DockTool.dll 的开发，该动态库中开发并输出了几个专门用于界面增强的类。对象 m_LeftDockBar、m_OutputDockBar 分别由 DockTool.dll 的输出类或输出类的派生类所定义。对象 m_LeftDockBar 是个具有浮动特性的窗口。它的功能类似于 Visual C++开发环境中的 Workspace 窗口，可以在主框架内任意浮动，并在其中可以嵌套多个视图。这些视图又可以与文档相关联，即实现文档多视图。m_OutputDockBar 也是一个具有浮动特性的窗口，但只可以停靠在主窗口内的上下侧，在其中还嵌套了一个用于文本输出的滚动视图，用于系统信息的输出显示。

CMainFrame 的主要任务集中在创建并管理上述这些界面对象。

类 CMainFrame 的定义如下（详见本书附带光盘内的 Mainfrm.h 文件）：

```
class CMainFrame : public CFrameWnd
{
        protected: // create from serialization only
        CMainFrame();
        DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:
        CMyLeftDockBar        m_LeftDockBar;              //可嵌套多个视图的浮动窗口
        CMessageViewDockBar       m_OutputDockBar;    //提示信息输出浮动窗口

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMainFrame)
        public:
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        //}}AFX_VIRTUAL

// Implementation
public:
                virtual ~CMainFrame();
#ifdef _DEBUG
        virtual void AssertValid()const;
                virtual void Dump(CDumpContext& dc)const;
#endif

protected:    // control bar embedded members
        CStatusBar        m_wndStatusBar;
        CToolBar          m_wndToolBar;
        CToolBar          m_wndDisplayBar;

// Generated message map functions
```

```
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        // NOTE - the ClassWizard will add and remove member functions here.
        //DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

    void DockControlBarLeftOf(CToolBar* Bar,CToolBar* LeftOf);
};
```

（4）应用程序类 CSTLViewerApp**。应用程序类管理程序的总体，从程序的初始化直到最后的清除任务。程序** STLViewer **中，应用类** CSTLViewerApp **由** CWinApp **直接派生。只需做少量修改，如给程序增加一个封面、设置主窗口的初始状态等，在此不做专门的介绍。**

1.4 STLViewer 中类的层次设计及软件模块结构划分

学习过 C++**的读者对于面向对象的编程技术（**OOP: Object-Oriented Programming**）都该不陌生，**OOP **技术是当前程序设计的主流方法学。详细讲述** OOP **技术的著作和文章很多。由于本书的重点不在于此，所以不在此花篇幅详细介绍** OOP **技术。概括地说，**OOP **技术的主要特征在于三个方面，即函数的重载、数据的封装和类的继承。能否在程序设计中较好地运用** OOP **技术的关键就在于对这三个主要特征的理解和体会。对一个较复杂的功能软件而言，在初期最重要的一部分工作是设计软件的整体结构。主要的考虑包括：要设计哪些主要的类；这些主要类之间的关系怎样，例如类之间的继承和派生关系；类之间的消息传递和数据交换等。如果类的数量较多，系统较复杂的话，还应该考虑软件功能模块的划分问题。读者或许已注意到，很多软件都是由一个执行程序（***.exe **文件）附加多个动态链接库（**DLL: Dynamic Link Library**）组成。可以这样说，类是软件设计时的模块，而** DLL **库是软件运行时的模块，一个** DLL **库可以输出实现类似功能的一组类、函数以及资源。通常的做法是把一些功能相对集中、可重复利用率高的类和函数集中于一个动态链接库中，执行程序在运行时根据需要动态地链接并调用这些** DLL **库中输出的类和函数。这种动态的链接和调用关系也可以存在于动态链接库之间。**

以 STLViewer **为例，整个应用程序由可执行程序** STLViewer.exe **和四个动态链接库组成，即** GeomCalc.DLL**、**glContext.DLL**、**GeomKernel.DLL **和** DockTool.DLL**。图** 1-6 **显示了这些软件模块相互之间以及它们与** MFC **基本类库之间的层次关系。**

STLViewer.exe **在运行时调用所需要的动态库，四个动态链接库均是** MFC **扩展方式构造的** DLL **库，这是因为只有采用扩展方式的** DLL **库才能输出** C++**的类。具体的创建和使用** DLL **库将在下节介绍。四个模块或动态链接库的功能介绍如下：**

（1）几何基本工具模块 GeomCalc.dll**。该模块输出基本几何对象类与几何计算函数，如描述点、矢量、矩阵的类以及相关的计算函数。它是** CAD **系统中必不可少的模块，**CAD **系统中与几何造型、操作以及显示等相关的功能都需要这些基本的几何对象和计算功能。系统中另外两个模块，即几何内核模块和图形显示模块就建立在** GeomCalc.dll **基础之上。**

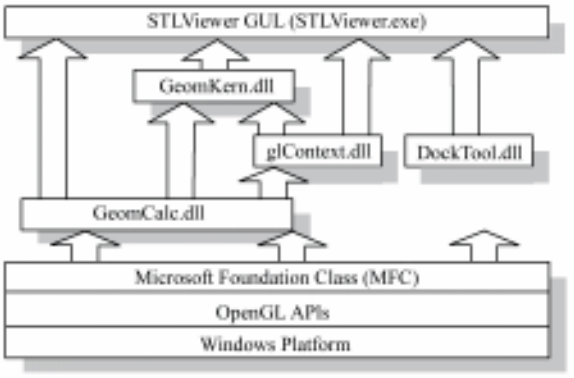


图 1-6　STLViewer 的层次结构

（2）CAD 图形工具模块 glContext.dll。该模块输出一系列用于 OpenGL 三维图形绘制的 C++类。在这些类中，基于 MFC 的机制对 OpenGL 的有关功能进行了封装。在 MFC 下可方便地调用这些类，完成 OpenGL 的初始化设置、三维几何体的光照显示，以及对显示的操作，如视角变换、显示缩放、光照或颜色的设置等。glContext.dll 建立在 MFC 及 GeomCalc.dll 的输出类基础之上。

（3）CAD 几何内核模块 GeomKernel.dll。该模块输出一系列用于描述三维几何对象的类。几何对象是 CAD 软件操作和显示的主体，这些对象之间又存在各种关系，如层次关系、拓扑关系。在本书中将描述和管理这些几何对象以及它们之间关系的类集称为几何内核模块。对这些几何体的描述建立在 GeomCalc.dll 输出类的基础之上。GeomKernel.dll 调用 glContext.dll 用于几何体的 OpenGL 绘制。建立一个真正的三维几何内核是一项复杂而专业的工作，将涉及到大量 CAD 领域的专业知识。在本书中，笔者构造 GeomKernel.dll 这样一个较简单的例子，主要目的在于讲述如何使用面向对象的编程技术设计和实现 CAD 几何内核。经过简化的几何内核库 GeomKernel 虽然只包括了为数不多的几何类，但构造了一个面向对象的几何模型结构。在它的基础上，可方便地对这个库进行扩充，增加更多功能的几何对象类。

（4）浮动界面工具模块 DockTool.dll。该模块输出一些增强界面效果的浮动窗口类。如图 1-2 所示，左侧的类似于 Visual C++集成开发环境中的 Workspace 浮动窗口，以及底部的用于提示信息输出的 Output 浮动窗口，都由 DockTool.dll 输出。

1.5　建立和使用动态链接库

以上我们分析了应用程序实例 STLViewer 的模块化的层次结构。这是一个由多个动态链接库（DLL）组成的、模块化的 CAD 软件。本节将介绍动态链接库的一些基本概念、使用方法，以及利用 Visual C++开发动态链接库的有关知识。在第 2 章中将尝试创建第一个 DLL——几何工具类库 GeomCalc.dll。GeomCalc.dll 将输出开发 CAD 系统需要的基础几何表示类，如点、矢量和矩阵，以及一些常用的计算函数。通过对 GeomCalc.dll 的开发，相信读者能够掌握开发 DLL 的有关技术。同时，通过几何基础类的设计和开发，读者也会对使用 Visual C++设计与开发中的一些技术细节有更深刻的认识。

1.5.1 动态链接库的基本概念

在软件结构中，库（library）是指一个或多个目标文件（.obj 文件）经过组合而形成的一个代码群。目标文件经过链接后生成一个可执行文件。与库的链接方式有两种：静态与动态链接。在静态链接中，链接程序将所需要的目标代码从库文件拷贝到可执行文件中。之所以被称为“静态”，是因为对库中的所有函数调用在链接生成可执行文件时已经完成，运行可执行文件时，不需要再从库中调用目标代码。这样链接生成的可执行文件的尺寸会比较大。而动态链接时，链接程序并没有把所需要的目标代码从库文件拷贝到可执行文件中，而是可执行文件在运行的开始或运行的过程中，根据需要从库文件中装载并使用相应目标代码。之所以被称为动态链接，是因为对目标代码的调用不是在链接时完成的，而是在执行过程中从库文件中动态装载使用的。

动态链接库 DLL（Dynamic Link Library 的英文缩写）即是这样一种动态链接模块，是包含输出类和共享函数库的二进制文件，可以被多个程序同时使用。建立应用程序的可执行文件时，不必将 DLL 链接到程序中，而是在运行时动态装载 DLL，装载时 DLL 被映射到调用进程的地址空间中。因而，DLL 是“运行时”的模块。

利用动态链接库技术，有利于应用程序的模块化，可将一个大的应用程序分成多个单独的功能模块。这样也有利于开发队伍之间的分工协作。对大型的 CAD 软件，如果不采用动态链接技术，将所有的执行代码都要加入到执行文件中，会导致程序的主执行文件异常庞大，执行时占用大量的内存而严重影响程序的运行效率。事实上，如果不使用动态链接技术，微机的内存恐怕连 Window 操作系统本身都启动不了。

除了节省程序运行时的资源消耗，DLL 的另一个主要特点是一个 DLL 可以同时被多个应用程序共享。在开发软件时，如果能够将一些功能集中、可重复利用率高的类和函数组合成一个独立的模块，这样不仅一个系统的其他模块可以调用这些类和函数，它也可以供别的应用程序共享。这也就是说，在 STLViewer 中开发的四个 DLL 库，还可用于其他的应用程序。世界著名的几何内核软件 ACIS 就是应用 DLL 技术的典型例子，它所提供给用户的就是一系列功能相对集中的 DLL 库，由用户在这些库的基础上开发自己的 CAD 应用程序。

另外，如果不改变 DLL 的接口，即使 DLL 库发生改变，也不需要重新编译所有调用它的应用程序。这也在一定程度上减轻了软件开发和维护的工作量。

1.5.2 基于 MFC 的动态链接库

MFC 提供了三种不同的方式支持 DLL 的开发：

- 建立静态链接 MFC 的常规 DLL（Regular DLL）。
- 建立动态链接 MFC 的常规 DLL。
- 建立动态链接 MFC 的扩展 DLL（Extension DLL）。

扩展 DLL 和常规 DLL 的区别在于：

（1）MFC 的扩展 DLL 支持 C++接口，即扩展 DLL 能够导出整个 C++类。这就是说，我们可以从已有的 MFC 类派生新的可再用类。扩展 DLL 在建立时使用的是 MFC 的动态链接，因而扩展 DLL 要求客户程序动态地链接到 MFC 动态库。扩展 DLL 的一个特点是尺寸很小，能够很快加载。

（2）常规 DLL 可被任意 Win32 编程环境加载。它的局限性在于常规 DLL 只能导出标准 C 接口，不能导出 C++类。但在常规 DLL 内部，仍然可以使用 C++类及 MFC 类。

（3）常规 DLL 能够采用显式链接或隐式链接，而扩展 DLL 只能采用显式链接。

静态链接 MFC 的 DLL 与动态链接 MFC 的 DLL 的区别在于：

（1）静态链接 MFC 的 DLL 将拷贝所有需要的 MFC 的代码，成为自我包含的模块，可以独立于 MFC 的 DLL 库而运行，但 DLL 的代码尺寸会较大。

（2）动态链接 MFC 的 DLL 在运行时动态链接 MFC 类库，因而 DLL 的代码尺寸会大大减小，但必须确保在运行的系统上有合适版本的 MFC DLL 库。

由以上分析可知，因为需要导出一系列 C++类，且很多类还需要从 MFC 类派生，因而，在本书的实例中要创建的是 MFC 的扩展 DLL。

本章相关程序

在本书附带光盘的“ CH1 ”子目录下，给出了运行 STLViewer 的执行程序和 DLL 库，它们是：

CH1\STLViewer.exe，执行程序；

CH1\GeomCalc.dll，几何基本工具库；

CH1\glContext.dll，CAD 图形工具库；

CH1\GeomKernel.dll，CAD 几何内核库；

CH1\DockTool.dll，浮动界面工具库。

运行 STLViewer.exe，程序出现如图 1-2 所示界面，但窗口中没有几何模型。

在目录“ CH1\Models\ ”下，给出了一些 STL 模型文件和 STLViewer 自己的模型文件（*.mdl），可以分别通过读入一个或多个 STL 文件来构造几何模型。在 ToolBar 中，使用“ STL File In ”按钮来输入 STL 模型，每读入一个文件，STL 模型将显示在窗口上。

STL 作为一个标准的三维几何形状文件接口，很多 CAD 软件都可以输出 STL 模型。本例中的几个 STL 模型由 AutoCAD R14 输出（ASCII 格式）。读者自己也可以创建一些 STL 模型，用 STLViewer.exe 来读入并观察模型。图 1-2 显示的就是读入了“ CH1\Models ”子目录下的几个 STL 文件后构造的模型。

mdl 文件是 STLViewer 自己定义的二进制格式文件。使用 File 菜单下的 Save 命令，可将当前模型存储成系统自己的.mdl 类型文件。也可使用 Open 命令，可以直接读入.mdl 文件。在目录“ CH1\Models\ ”下提供了几个已经生成的.mdl 文件。

第 2 章　几何基本工具库的开发

本章要点：

- **介绍几何计算中的基本对象：点、矢量和齐次变换矩阵。**
- **设计实现点、矢量和齐次变换矩阵的** C++**类。**
- **齐次变换矩阵与三维图形变换。**
- **开发几何基本工具库** GeomCalc.dll**。**

点、矢量和齐次变换矩阵是 CAD **中构造几何元素及几何变换运算时最常用的基本对象。本章将分析点、矢量和齐次变换矩阵之间的几何运算，并为它们的数据结构设计实现相应的** C++**类。最后，以这些类以及相关的计算函数为主要内容开发一个** DLL **库——**GeomCalc.dll**，即一个几何基本工具类库。这些基本的几何对象以及其相关计算在** CAD **中有着非常广泛的应用，是进一步开发其他高级功能的基础。本章所开发的** DLL **库，作为一个基本模块可供系统中其他模块开发时调用。**

2.1　点、矢量和齐次变换矩阵

2.1.1　点

点是构成所有几何对象的最基本的元素。在三维几何系统中，点用于表示空间的一个位置。几乎所有对几何对象的操作都可以归结为对点的操作，如几何体的平移、旋转、缩放等，实际上都是对构成几何对象的点进行平移、旋转、缩放。对点的操作是通过点与矢量、矩阵之间的运算来实现的。

在三维 CAD **系统中，点的空间位置由三个标量所表示的坐标值**(x,y,z)**定义，它是点的惟一属性。二维平面上的点是三维点的特殊情况。一个描述点的数据结构定义如下：**

```
typedef struct tagPoint3D{
        double x;
        double y;
        double z;
} POINT3D, *PPOINT3D;
```

C **语言中提供的数据结构是一种复合数据类型，用于将一些相关的数据类型组织成一个新的数据类型。数据结构的声明以关键字** struct **开始。注意，在结构** struct **之前使用了** typedef, typedef **用于将某个标识符定义成数据类型，然后将这个标识符当作数据类型使用。例如，用** typedef **将标识符** POINT3D **定义为点的数据结构** struct tagPoint3D **；将标识符** PPOINT3D **定义为指向点的数据结构** struct tagPoint3D **的指针。用** typedef **来重定义数据类型的目的在于提高程序的可移植性。**

例如，以下两行代码的作用是完全相同的，都声明了一个 struct tagPoint3D 的变量：

```
struct tagPoint3D    Pt;
POINT3D    Pt;
```

以下两行代码都声明了一个指向 struct tagPoint3D 的指针 pPt：

```
struct tagPoint3D * pPt;
PPOINT3D pPt;
```

2.1.2 矢量

矢量是带方向的长度单位，是几何计算中最重要的工具之一，几乎所有的几何计算都会涉及到矢量的运算。如物体的移动、旋转、坐标系的变换等等。空间中的一个矢量，由 dx、dy、dz 三个分别沿坐标轴方向的标量组成，二维平面上的矢量是三维矢量的特殊情况。矢量没有一个固定的空间位置。两个矢量，只要它们的长度和方向都一致，就认为这两个矢量是相等的。

我们定义空间矢量的数据结构如下：

```
typedef struct tagVector3D{
    double dx;
    double dy;
    double dz;
} VECTOR3D,*PVECTOR3D;
```

矢量与矢量之间、矢量与标量、矢量与点之间的常用的运算关系有：

1．矢量的和与差

两个矢量 V1(dx1,dy1,dz1)、V2(dx2,dy2,dz2)相加或相减，生成一个新的矢量 V^*。

$$V^* = V1 + V2 = (dx1 + dx2, dy1 + dy2, dz1 + dz2)$$

$$V^* = V1 - V2 = (dx1 - dx2, dy1 - dy2, dz1 - dz2)$$

2．矢量点乘

两个矢量 V1(dx1,dy1,dz1)、V2(dx2,dy2,dz2)点乘，得出一个标量积 s。

$$s = V1 \cdot V2 = dx1 \cdot dx2 + dy1 \cdot dy2 + dz1 \cdot dz2$$

求两矢量点乘的另一个公式如下，其中 θ 是两个矢量的夹角。

$$s = V1 \cdot V2 = |V1| \cdot |V2| \cdot \cos\theta$$

3．矢量叉乘

两个矢量 V1(dx1,dy1,dz1)、V2(dx2,dy2,dz2)叉乘，得出一个与这两个矢量都垂直的矢量 V^*。

$$V^* = V1 \times V2 = \begin{bmatrix} i & j & k \\ dx1 & dy1 & dz1 \\ dx2 & dy2 & dz2 \end{bmatrix}$$

$$= (dy1\ dz2 - dy2\ dz1, dz1\ dx2 - dz2\ dx1, dx1\ dy2 - dx2\ dy1)$$

4．矢量的模长

求矢量 V(dx,dy,dz)的模长 M 的计算公式如下：

$$M = |V| = (V \cdot V)^{1/2} = (dx\ dx + dy\ dy + dz\ dz)^{1/2}$$

5．矢量与标量相乘

矢量 V(dx,dy,dz)**与标量** s **相乘，可以放大或缩小该矢量的长度。**

$$V = V \cdot s = (dx \cdot s, dy \cdot s, dz \cdot s)$$

6．矢量与点的和与差

点 P(x, y, z)**与矢量** V(dx,dy,dz)**相加或相减，生成一个新的点** P^***，即沿矢量方向平移该点。**

$$P^* = P + V = (x + dx, y + dy, z + dz)$$

7．表示两空间点之间位置关系的矢量

点 P1(x1, y1, z1)**与点** P2(x2, y2, z2)**相减，返回一个矢量** V^***，表示两点之间的位置关系。**

$$V^* = P1 - P2 = (x1 - x2, y1 - y2, z1 - z2)$$

矢量还可以与一个变换矩阵相乘，返回一个变换了的矢量。例如，矢量的旋转就是用一个旋转矩阵来乘以该矢量而实现的。

矢量的运算是实现几何运算和变换操作的基础，因此，在点类、矢量类和齐次变换矩阵类中，设计和实现这些常用的运算功能，将会大大方便对这些类的使用。在 2.2 **节中，将分别介绍这三个类的实现，并针对这些类的对象之间存在的运算关系，设计一系列操作符重载函数来实现这些计算。**

2.1.3 齐次坐标与齐次变换矩阵

图形变换使得三维世界变得更加生动，物体的旋转、平移、缩放等都要靠图形变换来实现。图形变换也是 CAD **系统中的最基本内容之一。对几何对象的变换操作实际上是以点的变换为基础，在对图形中几何对象的一系列顶点进行变换后，用新的顶点来生成变换后的图形。点的齐次坐标表示方法提供了通过矩阵运算进行坐标变换，即把一个点从一个坐标系变换到另一个坐标系的有效方法，因而** CAD **系统中，通常使用齐次坐标的表示方法。所谓齐次坐标表示法就是由** n+1 **维矢量表示一个** n **维矢量。例如三维空间点的位置使用非齐次坐标表示时，具有三个坐标分量**(x, y, z)**，且是惟一的。若用齐次坐标表示时，需要使用四个分量**(hx,hy,hz,h)**来表示，且不惟一。例如** (15,30,25,1)**、**(7.5,15,12.5,0.5)**都是点**(15,30,25)**的齐次坐标。**

三维齐次坐标的变换矩阵的是一个 4×4 **的矩阵，它的形式如下：**

$$T_{3D} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

具体来说，如果三维图形变换前的某一点坐标是[X, Y, Z,1]**，经过由矩阵** T_{3D} **表示的几何变换后，其新坐标为**[X^*,Y^*,Z^*,1]**。**

齐次坐标还可以表示无穷远的点。在三维空间中，(x,y,z,0)**就表示了**(x,y,z)**方向上一个无穷远的点。例如，在** OpenGL **中使用了齐次坐标设置光源位置，**(x,y,z,0)**就表示一个在**(x,y,z)**方向上无穷远的光源位置。从无穷远处的光源发出的光，可被看作是平行光，它的几何意义非常明显。有关内容本书将在讲述** OpenGL **的光源设置时还会予以详细介绍。**

$$\begin{bmatrix} X^* & Y^* & Z^* & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \cdot \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

对如上所示的三维齐次坐标变换矩阵，其数据结构实际上就是一个 4×4 的双精度型数组。具体代码如下：

```
typedef struct tagCal3DMatrix{
    double A[4][4];
} MATRIX3D, *PMATRIX3D;
```

矩阵之间存在许多运算关系，如矩阵的加法运算、数乘矩阵、矩阵之间的乘法运算、逆矩阵、矩阵转置等。对于齐次变换矩阵，最关键的运算是矩阵之间的乘法运算。另外，如何构造变换矩阵也是齐次变换矩阵的一个重要概念。这些将在下面介绍矩阵类和矩阵变换的设计时进一步予以介绍。

2.2 设计点、矢量和齐次变换矩阵类

在定义了点、矢量和齐次变换矩阵的数据结构之后，下面将在这些数据结构的基础上进一步设计和实现封装与操作这些数据的 C++类。

在面向对象的程序设计中，类是最基本的程序单元。因此，面向对象的程序设计首先要以类的方式设计所要解决问题的模型，即设计类中应包含的数据，以及对这些数据进行操作的成员函数，并根据需要将这些数据和成员函数分别定义为私有类型（Private）、公有类型（Public）或是保护类型（Protected）。

MFC 中定义了数据结构 POINT，用于描述 Windows 窗口中的二维点坐标，并在数据结构 POINT 基础上派生了类 CPoint。但 CPoint 中的成员变量 x、y 都是整型数据，主要是用于对应窗口的像素坐标。在 MFC 中没有提供表示三维空间位置的数据结构和类，所以，在进行三维 CAD 系统的开发时，用户需要设计自己的结构和类来表示三维的点、矢量以及变换矩阵。

在上一节中提及的数据结构 POINT3D 的定义已经包括了描述一个点所需要的三个 double 型分量：x、y、z。结构和类的区别在于，结构只存有对象的数据，而类不仅封装了对象的数据，还增加了对数据进行操作的成员函数。类是一个独立的操作单元，是编程层次上的模块。但是，从结构 POINT3D 基础上可以派生出类 CPoint3D，派生的类 CPoint3D 不仅继承了结构 POINT3D 中的数据，而且可在类中进一步添加其他数据（如果需要的话）和对数据进行操作的成员函数，如各种构造函数、点与矢量、矩阵之间的运算函数等。

分别用来定义点、矢量和齐次变换矩阵的类的名称如下所列：

点类：CPoint3D

矢量类：CVector3D

齐次变换矩阵类：CMatrix3D

2.2.1 点类 CPoint3D

1．类 CPoint3D 的定义

类 CPoint3D 在头文件（CadBase.h）中的定义和相关说明如下：

```
class CPoint3D :public POINT3D  //从结构 POINT3D 基础上派生类 CPoint3D
{
public:
//构造函数与析构函数
        CPoint3D();
        CPoint3D(double ix,double iy,double iz=0.0);
        CPoint3D(const double*);
        CPoint3D(POINT3D p);
        ~CPoint3D();

public:
        //操作符重载，用于点与齐次变换矩阵之间的运算
        CPoint3D operator*(const MATRIX3D& matrix) const;
        void    operator*=(const MATRIX3D& matrix);

        //操作符重载，用于点与矢量之间的运算
        CPoint3D   operator+(VECTOR3D v) const;
        void       operator+=(VECTOR3D v);
        CPoint3D   operator-(VECTOR3D v) const;
        void       operator-=(VECTOR3D v);
        CVector3D       operator-(POINT3D sp) const;

        //操作符重载，用于判别空间两点是否位置相同
        BOOL            operator==(POINT3D pos) const;
        BOOL            operator!=(POINT3D pos) const;
} ;
```

2．声明类 CPoint3D 的对象

在类 CPoint3D 的构造函数中，由于设计了多个重载的构造函数，所以可以使用多种方式来方便地构造一个 CPoint3D 的对象。例如：

```
CPoint3D pt0；
CPoint3D pt1(10.0,20.0,30.0);

double v[3];
v[0] = 100;       v[1] = 100;       v[2] = 100;
CPoint3D pt2(v);

POINT3D pt3;
pt3.x = 10; pt3.y = 10; pt3.z = 30;
CPoint3D pt4(pt3);
```

由上面的多种三维点的定义可见，构造函数的重载为程序设计提供了很大方便。

3．设计点和矢量之间的运算的运算符重载

如前提及，点和矢量进行加、减运算的意义在于对点进行沿矢量的平移操作。例如在下列代码中，将点 pt0 (5,5,0)沿矢量 vec (20,20,0) 移动，点在完成加法操作之后的新位置将是 pt1(25,25,0)。

```
CPoint3D pt0(5,5,0);
CVector3D vec(20,20,0);
CPoint3D pt1 = pt0 + vec;            //(25, 25, 0)
```

点 pt 可以完成与 vec 直接相加的操作，并返回一个点，是因为在类 CPoint3D 中已经定义相应的操作符重载。这也正是操作符重载的优点。同样地，在类 CPoint3D 中还定义了下列的操作符重载函数，用于点与矢量之间的运算。

```
CPoint3D   operator+(VECTOR3D v) const;
void       operator+=(VECTOR3D v);
CPoint3D   operator-(VECTOR3D v) const;
void       operator-=(VECTOR3D v);
CVector3D        operator-(POINT3D sp) const;
```

由于设计了以上的操作符重载函数，在编程时就可以很方便地直接运用这些操作符实现点与矢量之间的操作。例如：

```
//点 pt0 沿矢量 vec 平移
pt0= pt0 + vec;
pt0 += vec;

//点 pt1 沿矢量 vec 的相反方向平移
pt1= pt1 - vec;
pt1 -= vec;

//计算由点 pt0 指向点 pt1 的矢量 vec
vec = pt1 – pt0;
```

读者也许注意到，在有些操作符重载函数的结尾定义了一个 const 限定词，这是限定此函数将不改变对象中的数据。下面还会进一步介绍有关 const 的使用。

4．设计点与齐次变换矩阵之间的运算的运算符重载

在类 CPoint3D 中定义了下列操作符重载函数，用于点与齐次变换矩阵的相乘：

```
CPoint3D operator*(const MATRIX3D& matrix) const;
void    operator*=(const MATRIX3D& matrix);
```

要对一个点进行有关变换，首先需要创建相应的变换矩阵，如旋转变换矩阵、镜像变换矩阵和平移变换矩阵等。有了以上操作符重载，在编写程序时可以就方便地实现点与齐次变换矩阵之间的操作。例如：

```
CMatrix3D mat;
CPoint3D    pt1,pt2;
```

```
…… //创建变换矩阵 mat，如旋转、镜像等

pt2 = pt1*mat;
pt1 *= mat;
```

5．设计判断空间两点位置关系的操作符重载

C 的程序员习惯于使用“ == ”、“ != ”来判断两个变量是否相等。在 CAD 系统开发时，也经常需要判断两个点是否相同(在一定误差范围内，判断两点是否重合)。

在类 CPoint3D 中定义了操作符“ == ”、“ != ”的重载函数，用于判断两点是否重合。

```
BOOL        operator==(POINT3D pos) const;      //是否重合
BOOL        operator!=(POINT3D pos) const;      //是否不重合
```

具体应用示例代码如下：

```
CPoint3D pt0,pt1;
if(pt0 == pt1){ //如果两点重合
……
}
```

6．使用 const 限定词

在类 CPoint3D 的成员函数以及函数接口中，均使用了限定词 const。这样定义的作用在于使代码更易于理解，且执行起来更加安全。当一个类的成员函数不改变类中的数据（成员变量）时，可以在说明函数时定义 const 限定词，这样做的优点是：

（1）加上 const 限定词的对象只能使用带 const 限定词的成员函数。

例如：

```
CVector3D vec(10,10);
const CPoint3D& pt = CPoint3D(0,0,30);
Cpoint3D pt1 = pt+vec;      //使用正确，因为 pt 的数据不被改变
pt += vec;                  //使用错误，因为将改变 pt 的数据
```

pt 由于被加了 const 限定词，所以只能调用 Cpoint3D 中定义了 const 限定词的成员函数。

（2）用户可以立即知道一个成员函数是否是只读的（read-only），即调用这个函数不会改变对象中的数据。

具体的示例如下：

```
void CPoint3D::operator+=(VECTOR3D v)
{
   x+=v.dx;
   y+=v.dy;
   z+=v.dz;
}
```

这个点和矢量的加法操作符重载函数将导致类中的成员变量值的改变，所以不能给函数定义 const 限定词。

```
CPoint3D CPoint3D::operator-(VECTOR3D v) const
```

```
{
  return CPoint3D(x-v.dx,y-v.dy,z-v.dz);
}
```

这个操作符重载函数只是读取类中的数据，而不改变它的值，所以可以给函数定义 const 限定词。

（3）当函数接口中的一个参数不允许在函数执行过程中被改变时，通常也给它加上 const 限定符。这样，编译器在编译该代码时可以保证这个参数的值不允许被改变。

例如，类 CPoint3D 中与矩阵相乘的操作符重载函数接口中的参数 matrix 是使用引用类型（类似于使用指针类型），即引用了一个外部的变量。对于这个外部变量，我们仅仅希望在函数中读取它的值，而不允许改变它的内部数据，所以需要在函数接口中对这个变量加上 const 限定词。

```
void    operator*=(const MATRIX3D& matrix);
```

2.2.2 矢量类 CVector3D

1．类 CVector3D 的定义

类 CVector3D 在头文件（CadBase.h）中的定义如下：

```
class CVector3D: public VECTOR3D
//从结构 VECTOR3D 基础上派生类 CVector3D
{
public:
    //构造函数与析构函数
    CVector3D();
    CVector3D(double dx,double dy,double dz=0);
    CVector3D(const double*);
    CVector3D(VECTOR3D v);
    virtual ~CVector3D();

    //操作符重载，用于矢量之间的加减运算
    CVector3D operator+(VECTOR3D v) const;
    void operator+=(VECTOR3D v);
    CVector3D operator-(VECTOR3D v) const;
    void operator-=(VECTOR3D v);

    //操作符重载，用于矢量与标量之间的运算
    //矢量与标量相乘
    CVector3D operator*(double d) const;
    void operator*=(double d);
    //矢量与标量相除
    CVector3D operator/(double d) const;
    void operator/=(double d);
    //操作符重载，用于矢量的叉乘
    CVector3D operator*(VECTOR3D v) const;
    //操作符重载，用于矢量的点乘
```

```
    double operator|(VECTOR3D v) const;

    //操作符重载，用于矢量与齐次变换矩阵相乘
    CVector3D operator*(const MATRIX3D& matrix) const;
    void    operator*=(const MATRIX3D& matrix);

    //计算矢量的模长
    double GetLength() const;
    double GetLengthXY() const;//在 XY 平面投影上的长度
    double GetLengthYZ() const; //在 YZ 平面投影上的长度
    double GetLengthZX() const; //在 ZX 平面投影上的长度

    //计算该矢量的单位矢量，即该矢量的方向
    CVector3D GetNormal() const;

    //单位化该矢量
    void        Normalize();

    //判断是否是个模长为 0 的矢量
    BOOL            IsZeroLength() const;
};
```

2．声明类 CVector3D 的对象

在如下的代码中，CVector3D 定义了四个构造函数重载，提供了多种方式来构造 CVector3D 的对象。

```
CVector3D();
CVector3D(double dx,double dy,double dz=0);
CVector3D(const double*);
CVector3D(VECTOR3D v);
```

根据以上的定义，可以使用如下构造方式来声明矢量对象：

```
CVector3D vec0;
CVector3D vec1(10,20,15);

double v[3];
v[0] = 100;         v[1] = 100;         v[2] = 100;
CVector3D vec2(v);

VECTOR3D tagVec;
tagVec.dx = 100; tagVec.dy = 100; tagVec.dy = 100;
CVector3D vec3(tagVec);
```

3．使用类 CVector3D 进行矢量运算

（1）矢量间的加减运算。在类 CVector3D 中定义了以下四个操作符重载函数用于矢量之间的加减运算：

```
CVector3D operator+(VECTOR3D v) const;
void operator+=(VECTOR3D v);
CVector3D operator-(VECTOR3D v) const;
void operator-=(VECTOR3D v);
```

有了以上操作符重载函数的定义，在编写有关几何运算的程序时就可以直接使用符号“+”、“-”来进行矢量间的加减运算。具体的运用代码如下：

```
CVector3D vec0(30,30,0);
CVector3D vec1(10,20,0);

CVector3D    vec = vec0 + vec1;    // vec(40,50,0)
vec += vec0;                       // vec(70,80,0)
vec -= vec1;                       // vec(60,60,0)
```

（2）矢量与标量间的乘除运算。类 CVector3D **中定义了以下四个操作符重载函数用于矢量与标量之间的乘除运算，即改变一个矢量的模长：**

```
CVector3D operator*(double d) const;
void operator*=(double d);
CVector3D operator/(double d) const;
void operator/=(double d);
```

应用示例如下：

```
CVector3D vec0(30,30,0);

CVector3D    vec = vec0*1.5;          // vec(45,45,0)
vec *= 0.2;                           // vec(9,9,0)
vec = vec0/2.0;                       // vec(15,15,0)
vec /= 2.0;                           // vec(7.5,7.5,0)
```

（3）矢量的点乘与叉乘。在类 CVector3D **中，使用符号“|”与“*”分别作为点乘、叉乘的操作符。操作符重载函数如下：**

```
double operator|(VECTOR3D v) const;
CVector3D operator*(VECTOR3D v) const;
```

应用示例如下：

```
CVector3D vec0(1,0,0);
CVector3D vec1(0,1,0);

double m = vec0|vec1;              // m = 0
CVector3D vec = vec0*vec1;         // vec(0,0,1)
```

4．使用齐次变换矩阵对矢量进行变换

使用齐次变换矩阵来实现矢量变换是 CAD **中最常用的计算方法之一，如进行矢量的旋转、坐标系的变换操作等。**

类 CVector3D **中与齐次变换矩阵相乘的操作符重载函数如下：**

```
CVector3D operator*(const MATRIX3D& matrix) const;
void    operator*=(const MATRIX3D& matrix);
```

应用示例：

```
CMatrix3D mat;
CVector3D    vec1,vec2;

……. //创建变换矩阵 mat ，如旋转、镜像……

vec1 = vec2*mat;
vec2 *= mat;
```

5．定义单位化矢量函数

单位矢量是指模长为 1 **的矢量，它可以被认为是一个方向。单位矢量的概念在** CAD **中经常被应用于计算有关的方向。例如，在网格曲面的表示中，每个三角曲面片都附带一个单位矢量，用于表示这个曲面片的方向。在生成曲面光照模型时，需要计算曲面片方向和光源之间的夹角，以确定曲面片的反光度。曲面片方向和光源之间的夹角越小，它的反光度就越高。**

将一个矢量单位化，就是将这个矢量的模长置为 1**。类** CVector3D **中，定义的矢量单位化函数如下：**

```
CVector3D   GetNormal() const;
void        Normalize();
```

这两个函数分别用于返回当前矢量的单位矢量，以及将该矢量单位化。

2.2.3 变换矩阵类 CMatrix3D

1．类 CMatrix3D **的定义**

变换矩阵类（CMatrix3D**）是用来进行有关的矩阵操作和生成相应的几何运算所需的变换矩阵。在头文件（**CadBase.h**）中，其具体的定义代码如下：**

```
class CMatrix3D : public MATRIX3D
//从结构 MATRIX3D 基础上派生类 CMatrix3D
{
public:
    //构造函数与析构函数
    CMatrix3D();
    CMatrix3D(const MATRIX3D&);
    CMatrix3D(const double *);
    virtual ~CMatrix3D();
public:
    //操作符重载，用于矩阵的乘法运算
    CMatrix3D operator*(const MATRIX3D& matrix)const;
    void operator*=(const MATRIX3D& matrix);
    //单位化矩阵
    void      IdenticalMatrix();
```

```
        //矩阵的值
        double GetValue() const;

    public:
        //静态函数
        //矩阵求值
        static double GetValue(double a00, double a01, double a02,
                               double a10, double a11, double a12,
                               double a20, double a21, double a22);

        //创建镜像矩阵
        static CMatrix3D CreateMirrorMatrix(VECTOR3D plnNorm);

        //创建旋转矩阵
        static CMatrix3D CreateRotateMatrix(double da,VECTOR3D bv);

        //创建缩放矩阵
        static CMatrix3D CreateScaleMatrix(double);

        //创建平移矩阵
        static CMatrix3D CreateTransfMatrix(VECTOR3D vec);
    };
```

2．声明类 CMatrix3D **的对象**

类 CMatrix3D **中定义了三种构造函数重载：**

```
CMatrix3D();                    //默认构造方式
CMatrix3D(const MATRIX3D&);     //使用 MATRIX3D 数据结构构造
CMatrix3D(const double *);      //使用数组构造
```

对应于以上的构造函数重载，可分别使用下列方式来构造 CMatrix3D **的对象**:

```
CMatrix3D matrix0;              //使用默认方式定义
……                              //填写 matrix0
CMatrix3D matrix1(matrix0);     //使用 MATRIX3D 数据结构定义

double mat[16];
……                              //填写数组 mat16
CMatrix3D matrix2(mat);         //使用数组方式定义
```

3．定义齐次矩阵的乘法运算符重载函数

两个三维齐次矩阵相乘的结果是返回一个新的三维齐次矩阵，它的意义在于将两次相应的变换集成在一个变换矩阵中。图形变换中，经常将多个齐次矩阵连乘，以对几何对象进行复合变换，从而使几何对象完成一次以上的图形变换。例如，把对一个点的平移、旋转变换矩阵连乘，即将这两个变换复合在一起，它的作用相当于先将该点平移，然后在新的位置上再进行旋转。

在类 CMatrix3D **中，为矩阵相乘定义了以下操作符重载函数：**

```
CMatrix3D operator*(const MATRIX3D& matrix)const;
void operator*=(const MATRIX3D& matrix);
```

应用示例如下：

```
CPoint3D pt(10,10,0);
CMatrix3D mat1,mat2;

//创建旋转变换矩阵 mat1，沿矢量 CVector3D(0,0,1)旋转 90°
mat1 = CMatrix3D:: CreateRotateMatrix(PI/2,CVector3D(0,0,1));

//创建平移变换矩阵 mat2，沿矢量 CVector3D(-10,0,0)平移
mat2 = CMatrix3D::CreateTransfMatrix(CVector3D(-10,0,0));

CMatrix3D mat;
mat = mat1*mat2;              //矩阵连乘
pt *= mat;                    //pt(-20,10,0)
```

mat 矩阵中包含了两个变换，即先沿 z **轴旋转** 90° **，进而沿** x **轴负方向平移** 10**。点** pt(10,10,0) **被最终变换到** pt(-20,10,0)**。当然，也可以使用该操作符重载实现多个矩阵的连乘。例如：**

```
mat = mat1*mat1*mat1;
```

将三个绕 z **轴旋转** 90° **的矩阵连乘，所生成的将是一个绕** z **轴旋转** 270° **的旋转矩阵。**

4．定义矩阵单位化函数

对于一个齐次矩阵，若其主对角线各元素 m_{ij} **都等于** 1**，且其余各元素均为零，这个矩阵可称作单位矩阵。例如，**4 **阶的齐次矩阵的单位矩阵形式如下：**

$$I=\begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}$$

单位矩阵的特点在于它不对物体作任何变换，所以在构造一般齐次变换矩阵时，默认形式下均初始化为单位矩阵。例如，在类 CMatrix3D **的构造函数中，默认状态下将矩阵构造为单位矩阵（函数** CMatrix3D()**）。在** CMatrix3D **中，相应的矩阵单位化函数是** IdenticalMatrix()**，用于将矩阵对象重新设为单位矩阵。函数定义如下：**

```
void      IdenticalMatrix();
```

5．创建变换矩阵

如前提及，齐次矩阵的主要用途在于它的图形变换功能，所以，如何创建这些特定的变换矩阵是类 CMatrix3D **的关键内容。在类** CMatrix3D **中，就定义了几个静态函数，用于创建所需要的变换矩阵。这些函数定义如下：**

（1**）创建镜像矩阵，将几何对象变换到沿所定义的镜面对称的位置。函数的说明如下：**

```
static CMatrix3D CreateMirrorMatrix(VECTOR3D plnNorm);
```

参数：

plnNorm, **镜子平面的法矢。**

返回值：

返回生成的镜像变换矩阵。

（2）创建旋转矩阵，将几何对象绕规定的矢量旋转一个角度。函数的说明如下：

```
static CMatrix3D CreateRotateMatrix(double da,VECTOR3D bv);
```

参数：

da, **旋转的弧度。**

bv, **转轴的矢量方向。**

返回值：

返回生成的旋转变换矩阵。

（3）创建缩放矩阵，将几何对象的尺寸放大或缩小一个比例。函数的说明如下：

```
static CMatrix3D CreateScaleMatrix(double sc);
```

参数：

sc, **缩放比例。**

返回值：

返回生成的缩放变换矩阵。

（4）创建平移矩阵，将几何对象沿一个矢量平移。函数的说明如下：

```
static CMatrix3D CreateTransfMatrix(VECTOR3D vec);
```

参数：

vec, **平移的矢量。**

返回值：

返回生成的平移变换矩阵。

由于这些函数被定义为静态函数，因而它们可以被直接调用，而不需要事先声明一个类的对象。例如：

```
CMatrix3D mat = CMatrix3D:: CreateRotateMatrix(PI/2,CVector(0,0,1));
```

创建的矩阵就是一个绕 z **轴旋转** 90 **° 的旋转变换矩阵。**

下面一节将介绍创建齐次变换矩阵的有关知识。

2.3 三维图形的几何变换

2.3.1 三维齐次变换矩阵

三维齐次变换矩阵的形式如下：

$$T_{3D}=\begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix}$$

从功能上划分，T_{3D} 可分为四个子矩阵，分别负责不同的变换任务。其中，$\begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{bmatrix}$用于产生比例、旋转、剪切等几何变换；$[m_{30} \quad m_{31} \quad m_{32}]$用于产生平移变换；$\begin{bmatrix} m_{03} \\ m_{13} \\ m_{23} \end{bmatrix}$用于产生投影变换；$[m_{33}]$用于产生整体比例变换。

2.3.2 平移变换

$$\begin{bmatrix} X^* & Y^* & Z^* & 1 \end{bmatrix}=\begin{bmatrix} X & Y & Z & 1 \end{bmatrix}\cdot\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

$$=\begin{bmatrix} X+T_x & Y+T_y & Z+T_z & 1 \end{bmatrix}$$

上式中，平移变换矩阵将点(X,Y,Z)沿矢量(T_X,T_Y,T_Z)平移至点(X^*,Y^*,Z^*)，它等同于下式：

$$X^*=X+T_x$$
$$Y^*=Y+T_y$$
$$Z^*=Z+T_z$$

2.3.3 旋转变换/绕空间任意轴的旋转变换函数的实现

在三维右手坐标系中，相对于坐标系原点绕坐标轴旋转θ角的变换矩阵形式是：

（1）绕 x 轴旋转

$$\begin{bmatrix} X^* & Y^* & Z^* & 1 \end{bmatrix}=\begin{bmatrix} X & Y & Z & 1 \end{bmatrix}\cdot\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

（2）绕 y 轴旋转

$$\begin{bmatrix} X^* & Y^* & Z^* & 1 \end{bmatrix}=\begin{bmatrix} X & Y & Z & 1 \end{bmatrix}\cdot\begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

（3）绕 z 轴旋转

$$\begin{bmatrix} X^* & Y^* & Z^* & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

但在实际情况下，使用得更多的还是绕任意坐标轴旋转。将几何对象绕任意轴旋转实际上是分别绕 x、y、z **旋转后复合变换的结果。例如，在类** CMatrix3D **中定义的创建旋转变换矩阵的函数：**

```
static CMatrix3D CreateRotateMatrix(double da,VECTOR3D bv);
```

该函数用于创建一个相对于坐标原点，绕任意 bv **旋转弧度** da **的齐次变换矩阵。**

创建这个矩阵的基本思想是：先后绕 z **轴、**x **轴旋转矢量** bv**，使矢量** bv **与** y **轴重合，再绕** z **轴旋转**θ**角。然后做上述变换的逆变换，使之回到原来的位置。具体过程如下：**

（1**）绕** z **轴旋转矢量** bv**，至平面** yox**。**

具体实现：计算出矢量 bv **至平面** yoz **的夹角**θ**，得出这个变换的矩阵** Rz**。**

（2**）绕** x **轴旋转矢量** bv**，至平面** xoy**。**

具体实现：计算出矢量 bv **至平面** xoy **的夹角**θ**，得出这个变换的矩阵** Rx**。**

（3**）绕** z **轴将矢量** bv **旋转弧度** da**。**

具体实现：计算出这个变换的矩阵 R**。**

（4**）分别求出矩阵** Rz **和** Rx **的逆矩阵，用** Rzn **和** Rxn **表示。**

具体实现：连乘这些变换矩阵 Rz*Rx*R*Rxn*Rzn**，即为所求得的变换矩阵。**

有必要指出的是：这个变换的过程不惟一，即旋转的先后顺序不是惟一的。但不同旋转顺序返回的结果是相同的。

下面我们将结合创建这个矩阵的源代码来分析这个过程。函数 CreateRotateMatrix()**的源代码如下：**

```
CMatrix3D CMatrix3D::CreateRotateMatrix(double da,VECTOR3D v)
{
    CMatrix3D R;    //初始化一个单位矩阵
    CVector3D bv(v);

    //如果旋转 0 弧度，则返回初始化的单位矩阵
    if(IS_ZERO(da)) return R;

    //确认矢量不是零矢量
    ASSERT(!bv.IsZeroLength());

    //求矢量在 xoy 平面上的投影长度 lxy
    double lxy = bv.GetLengthXY();

    //若 lxy 为 0，则矢量 bv 和 z 轴平行，创建并返回绕 z 轴旋转弧度 da 的变换矩阵
    if(IS_ZERO(lxy))
    {
        if(bv.dz < 0.0) da *= -1.0;
```

```
        R.A[0][0]=R.A[1][1]=cos(da);
        R.A[0][1]=sin(da);R.A[1][0]=-sin(da);
        return R;
}

//求矢量在 yoz 平面上的投影长度 lyz
double lyz=bv.GetLengthYZ();

//若 lyz 为 0，则矢量 bv 和 x 轴平行，创建并返回绕 x 轴旋转弧度 da 的变换矩阵
if(IS_ZERO(lyz))
{
        if(bv.dx < 0.0) da *= -1.0;
        R.A[2][2]=R.A[1][1]=cos(da);
        R.A[1][2]=sin(da);R.A[2][1]= -sin(da);
        return R;
}


//求矢量在 zox 平面上的投影长度 lxz
double lxz=bv.GetLengthZX();

//若 lxz 为 0，则矢量 bv 和 y 轴平行，创建并返回绕 y 轴旋转弧度 da 的变换矩阵
if(IS_ZERO(lxz))
{
        if(bv.dy < 0.0) da *= -1.0;
        R.A[0][0]=R.A[2][2]=cos(da);
        R.A[0][2]=-sin(da);R.A[2][0]=sin(da);
        return R;
}
//创建矩阵 Rz，将矢量 bv 绕 z 轴旋转至 yoz 平面
//旋转角度θ = acos(by.dy/lxy)
CMatrix3D Rz;
Rz.A[0][0]=Rz.A[1][1]=bv.dy/lxy;
Rz.A[0][1]=bv.dx/lxy;Rz.A[1][0]= -bv.dx/lxy;

//创建矩阵 Rx，将矢量 bv 绕 x 轴旋转至 xy 平面
//旋转角度θ = acos(by.dz/len)
double len=bv.GetLength();
CMatrix3D Rx;
Rx.A[2][2]=Rx.A[1][1]=bv.dz/len;
Rx.A[1][2]=lxy/len;Rx.A[2][1]= -lxy/len;

//创建矩阵 R，将矢量 bv 绕 z 轴旋转弧度 da
R.A[0][0]=R.A[1][1]=cos(da);
R.A[0][1]=sin(da);R.A[1][0]= -sin(da);

//创建矩阵 Rx 的逆矩阵 Rxn
CMatrix3D Rxn;
```

```
        Rxn.A[2][2]=Rxn.A[1][1]=bv.dz/len;
        Rxn.A[2][1]=lxy/len;Rxn.A[1][2]= -lxy/len;

        //创建矩阵 Rz 的逆矩阵 Rzn
        CMatrix3D Rzn;
        Rzn.A[0][0]=Rzn.A[1][1]=bv.dy/lxy;
        Rzn.A[1][0]=bv.dx/lxy;Rzn.A[0][1]= - bv.dx/lxy;

        //返回创建的旋转变换矩阵，即一系列变换的连乘后的矩阵
        return Rz*Rx*R*Rxn*Rzn;
    }
```

以上讨论的旋转变换的情况都是基于坐标系原点的几何变换，但大多数情况下，用户会要求绕空间任意一根轴线 AB 作旋转变换，即旋转轴的起点 A 是空间任意位置，方向是矢量 AB。其实，这个变换过程可看作首先把 A 点平移到坐标系的原点(0,0,0)，旋转变换以后再平移回相反的距离。

例如，创建一个绕轴线 AB 旋转弧度 a 的齐次变换矩阵 R。其中 A 的坐标是 A(Xa,Ya,Za)，方向矢量为 bv。这个旋转变换矩阵的创建过程如下：

（1）创建平移变换矩阵 T。

$$T=\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ -Xa & -Ya & -Za & 1 \end{bmatrix}$$

（2）计算 T 的逆矩阵 T^{-1}。

$$T^{-1}=\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ Xa & Ya & Za & 1 \end{bmatrix}$$

（3）创建基于坐标系原点，绕矢量 bv 旋转弧度 a 的旋转变换矩阵 R1。

（4）连乘以上变换矩阵，可获得旋转所要求的变换矩阵 R。

$$R = T \times R1 \times T^{-1}$$

2.3.4 几何缩放

几何缩放也称为比例变换。若变换是相对于坐标系的原点，比例变换矩阵的形式是：

$$Ts=\begin{bmatrix} Sx & & & \\ & Sy & & \\ & & Sz & \\ & & & 1 \end{bmatrix}$$

对一个齐次坐标[X,Y,Z,1]的比例变换计算如下：

$$\begin{bmatrix} X^* & Y^* & Z^* & 1 \end{bmatrix} = \begin{bmatrix} X & Y & Z & 1 \end{bmatrix} \cdot \begin{bmatrix} Sx & & & \\ & Sy & & \\ & & Sz & \\ & & & 1 \end{bmatrix}$$

$$= \begin{bmatrix} Sx \cdot X_x & Sy \cdot Y_y & Sz \cdot Z & 1 \end{bmatrix}$$

（1）当 Sx、Sy、Sz 相等，且等于 1 时，为恒等比例变换，即图形不变。

（2）当 Sx、Sy、Sz 相等，且大于 1 时，图形放大。

（3）当 Sx、Sy、Sz 相等，且小于 1 时，图形缩小。

（4）当 Sx、Sy、Sz 不相等时，图形在三个坐标轴上的变化是不均匀的。

但在 CAD 软件中，通常使用等比例缩放，即三个缩放参数是相等的。类 CMatrix3D 中的静态函数 CreateScaleMatrix(double sc)即是创建一个等比例的缩放矩阵。这个变换矩阵也是以坐标系原点为中心。若对于任意点缩放变换的情况，可使用上面介绍的方法，即先创建平移矩阵将该点移动至坐标系原点，缩放变换后，再平移回相反的方向。

2.3.5 对称变换/沿空间任意平面的对称变换函数的实现

对称变换也称为镜像变换（mirroring）。对称变换如同照镜子，将物体变换到沿镜面对称的位置上。对于三维空间中的对称变换，首先需要定义一个平面作为镜面。

下面列出了几个对称变换矩阵，分别对应于以平面 yoz、zox、xoy 为镜面的对称变换矩阵。

$$T_{yz} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ //**沿 yoz 平面(平面 x=0)的对称变换矩阵**

$$T_{zx} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ //**沿 zox 平面(平面 y=0)的对称变换矩阵**

$$T_{xy} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$ //**沿 xoy 平面(平面 z=0)的对称变换矩阵**

以上列出的是三个特殊情况的对称变换矩阵。但在大多数场合，要求创建的是一个沿空间任意平面的对称变换矩阵。这就需要将平移、旋转和镜像几个变换复合起来，最终实现沿任意平面的对称。下面，通过一个例子来掌握这个复合变换过程。

在一个右手坐标系中，空间有三个点 A(0,0,0)、B(0,3,0)和 C(3,1,1)，创建由这三个点所定义的镜面的对称变换矩阵 Tm。步骤如下：

（1）沿 y 轴将平面旋转角度θ，使该平面与 xy 平面重合。

$$\sin\theta = 1/\sqrt{10} \qquad \cos\theta = 3/\sqrt{10}$$

（2）沿 xy 平面作镜像变换。

（3）用该旋转矩阵的逆矩阵作旋转变换。

这个复合矩阵变换的过程是：

$$Tm = \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 3/\sqrt{10} & 0 & -1/\sqrt{10} & 0 \\ 0 & 1 & 0 & 0 \\ 1/\sqrt{10} & 0 & 3/\sqrt{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 3/\sqrt{10} & 0 & 1/\sqrt{10} & 0 \\ 0 & 1 & 0 & 0 \\ -1/\sqrt{10} & 0 & 3/\sqrt{10} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$= \begin{bmatrix} 0.8 & 0 & 0.6 & 0 \\ 0 & 1 & 0 & 0 \\ 0.6 & 0 & -0.8 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

类 CMatrix3D 中的静态函数 CreateMirrorMatrix(VECTOR3D plnNorm)就是用来创建一个沿坐标系原点和平面法矢量 plnNorm 定义的镜面的对称变换矩阵。

2.4 设计几何基本工具库 GeomCalc.dll

2.4.1 GeomCalc.dll 中的输出类与输出函数

在以上内容中，介绍了三个主要的几何基本工具类——点、矢量和齐次变换矩阵的设计。这些基本类是创建 CAD 软件的基础，其他 CAD 高级功能的开发都需要这些基础几何元素。基于这样的考虑，在这一节中，将它们集成为一个单独的运行模块，即一个动态链接库，会更方便程序的其他功能模块的调用，同时也有利于系统的开发与维护。

在本书提供的基本几何库 GeomCalc.dll 中（如图 2-1 所示），设计了包括上述三个类在内的一共四个几何基本工具类作为输出类，它们是：

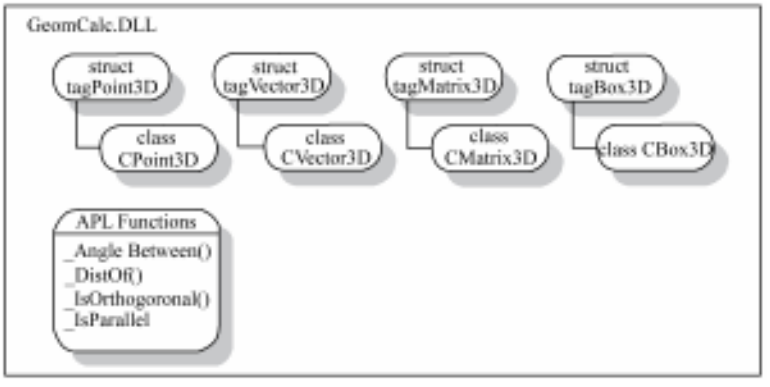


图 2-1 GeomCalc.dll 的输出类与输出函数

- CPoint3D　　三维点类
- CVector3D　　三维矢量类
- CMatrix3D　　三阶齐次变换矩阵类
- CBox3D　　三维空间包容盒类

其中，包容盒类的作用在于描述平面或空间的一个立方形的体积。在 CAD 算法设计中，经常会用到包容盒，例如粗略估计一个物体在空间所占的位置。一个几何体的对象通常附带一个包容盒的信息，用于迅速获取这个几何体所占空间的大致范围。这在几何求交、全局显示等计算时经常用到。

在 GeomCalc.dll 中，还设计并输出以下常用的全局计算函数，以方便在不同的场合调用。对于要输出的全局函数，可使用定义符 AFX_EXT_API 来定义它为 DLL 库的输出函数。如下面列出的输出函数。

```
double    AFX_EXT_API _AngleBetween(VECTOR3D v1,VECTOR3D v2);
double    AFX_EXT_API _DistOf(POINT3D pt0, POINT3D pt1);
BOOL      AFX_EXT_API _IsParallel(VECTOR3D v0,VECTOR3D v1);
BOOL      AFX_EXT_API _IsOrthogonal(VECTOR3D v0,VECTOR3D v1);
```

特别值得注意的是，GeomCalc 是一个扩展类型的 DLL，扩展型 DLL 的最大特点就是可以输出 C++的类。对于上述的几个输出类，如下代码所示，需要在前头加上输出类的定义 AFX_EXT_CLASS。在使用 DLL 的应用程序中可以利用这些输出的类来声明对象。需要在输出类的头上加上定义符号 AFX_EXT_CLASS，以表明这个类是 DLL 库的输出类。因为在 DLL 库中，用户可以定义一些中间类或函数，只供内部调用，若不用来输出，则不必加此定义符号。

```
class AFX_EXT_CLASS CPoint3D :public POINT3D
{
……
};

class AFX_EXT_CLASS CVector3D :public VECTOR3D
{
……
};

class AFX_EXT_CLASS CMatrix3D :public MATRIX3D
{
……
};
```

2.4.2　创建几何基本工具库 GeomCalc.dll 的步骤

由于需要输出 C++类，要创建的 GeomCalc 应该是 MFC 扩展类型的 DLL，创建该 DLL 库的步骤如下：

（1）从 File 菜单选择 New 命令，弹出 New 对话框。

（2）如图 2-2 所示，切换到 Project 选项卡，选择项目类型为 MFC AppWizard (dll)。

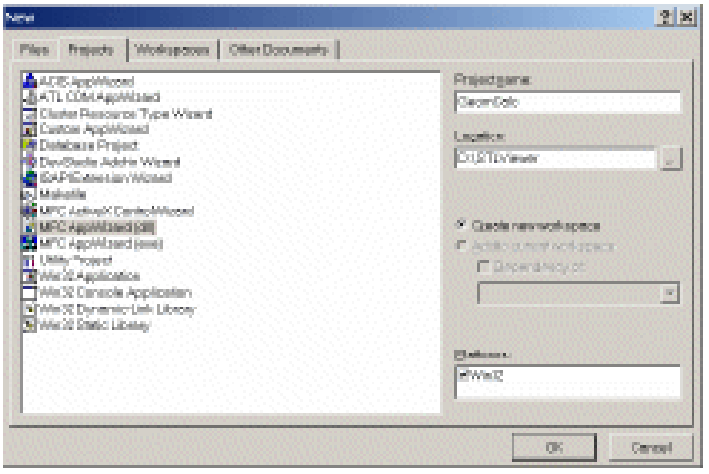


图 2-2　创建 DLL 项目

（3）在 Project name 文本框中输入名字 GeomCalc，单击 OK 按钮，弹出如图 2-3 所示的对话框。创建 MFC 扩展 DLL，即选择 MFC Extension DLL (using shared MFC DLL)。

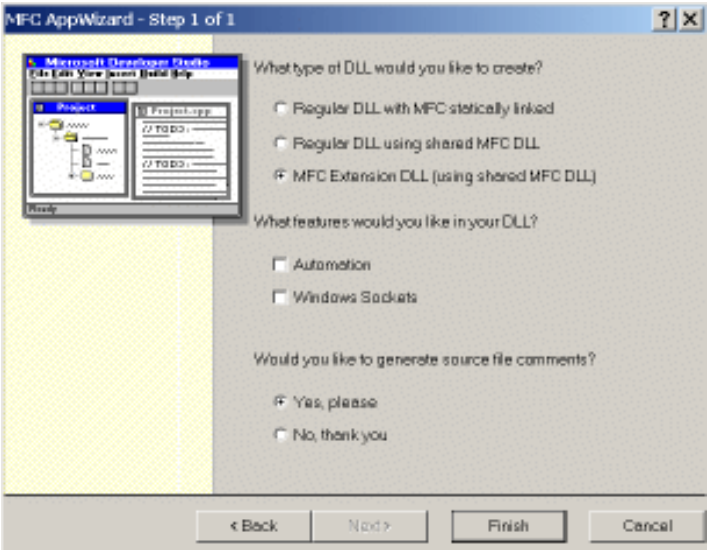


图 2-3　选择 MFC 扩展类型

（4）选择后，单击 Finish 按钮。

（5）在随后的对话框中单击 OK 按钮，MFC 将自动生成 DLL 的框架文件。

查看 MFC AppWizard 创建生成的 MFC 扩展类型 DLL 库项目中的文件，在文件 GeomCalc.cpp 中包含了 DllMain()主函数。编译连接项目，一切正常后，就可以在项目中增加类和函数了。在项目中插入类的方法和在普通的 MFC 项目中插入类的方法一样，只需将类的头文件（*.h）和实现文件(*.cpp)加入到项目中即可。需要注意的是，对于要输出的类，需要在声明类时添加 AFX_EXT_CLASS 的标识符。对于要输出的全局函数，需要在声明类时添加

AFX_EXT_API 的标识符。读者可具体参照 GeomCalc 项目中的头文件 CadBase.h。

完成了类和函数的添加后，就可以编译连接这个 DLL 项目，以生成所需要的动态链接库。如果没有编译或连接错误的话，将会生成一个 DLL 文件和 LIB 文件，即 GeomCalc.dll 和 GeomCalc.lib。当 GeomCalc.dll 供其他项目调用时，除了需要上述两个文件外，还需要说明类和函数的头文件 CadBase.h。通常情况下，可以将这些 DLL、LIB 文件和有关的头文件拷贝到调用它的项目下，以方便使用。当然，也可以通过设置路径的方法来实现。

2.4.3 使用 GeomCalc.dll

如前所述，将几何基本工具类集成在一个 DLL 中的好处在于，相应的计算功能可供其他应用程序调用，并可在多个应用程序中共享。对库 GeomCalc.dll 的使用，应完成以下步骤：

（1）为库和头文件设置路径或拷贝这些文件到当前目录。

要使应用程序在运行时能够找到 GeomCalc.dll，这个库必须放在和应用程序相同的目录下，或者是系统设置的默认路径下。最好 GeomCalc.lib 文件和头文件 CadBase.h 也都分别拷贝到当前项目的一个子目录下，以便于调用。

（2）在应用程序项目中连接库文件 GeomCalc.lib。

（3）在需要使用 GeomCalc.dll 的输出类或输出函数的文件头部插入头文件 CadBase.h。

```
#include “CadBase.h”
```

2.5 有关源程序代码

2.5.1 文件 CadBase.h

```
////////////////////////////////////////////////////////CadBase.h/////////////////////////////////////////////

#ifndef _CAD_DEF_H_
#define _CAD_DEF_H_

#include <math.h>
#include <afxtempl.h>

#define CAD_ZERO          1.0E-6
#define NC_ZERO           1.0E-3
#define IS_ZERO(x)        (fabs(x)<=CAD_ZERO)
#define IS_NCZERO(x)      (fabs(x)<=NC_ZERO)
#define IS_BETWEEN(x,min,max) (x<=max && x>=min)
#define PI  3.1415926535

typedef struct tagPoint3D{
        double x;
        double y;
        double z;
} POINT3D, *PPOINT3D;
```

```
typedef struct tagVector3D{
	double dx;
	double dy;
	double dz;
} VECTOR3D,*PVECTOR3D;

typedef struct    tagMatrix3D{
	double A[4][4];
} MATRIX3D, *PMATRIX3D;

class CPoint3D;
class CVector3D;
class CMatrix3D;

class AFX_EXT_CLASS CPoint3D :public POINT3D
{
public:
	CPoint3D();
	CPoint3D(double ix,double iy,double iz=0.0);
	CPoint3D(const double*);
	CPoint3D(POINT3D p);
	~CPoint3D();

public:
	//operators
	CPoint3D operator*(const MATRIX3D& matrix) const;
	void    operator*=(const MATRIX3D& matrix);

	//offsetting with vector
	CPoint3D operator+(VECTOR3D v) const;
	void operator+=(VECTOR3D v);
	CPoint3D operator-(VECTOR3D v) const;
	void operator-=(VECTOR3D v);

	BOOL operator==(POINT3D pos) const;
	BOOL operator!=(POINT3D pos) const;

	//derived vector = this point - sp
	CVector3D operator-(POINT3D sp) const;
} ;

class AFX_EXT_CLASS CVector3D: public VECTOR3D
{
public:
	CVector3D();
	CVector3D(double dx,double dy,double dz=0);
	CVector3D(const double*);
```

```
	CVector3D(VECTOR3D v);
	virtual ~CVector3D();

	//operator
	CVector3D operator+(VECTOR3D v) const;
	void operator+=(VECTOR3D v);
	CVector3D operator-(VECTOR3D v) const;
	void operator-=(VECTOR3D v);

	CVector3D operator*(double d) const;
	void operator*=(double d);
	CVector3D operator/(double d) const;
	void operator/=(double d);

	//cross product
	CVector3D operator*(VECTOR3D v) const;

	//dot product
	double operator|(VECTOR3D v) const;

	CVector3D operator*(const MATRIX3D& matrix) const;
	void    operator*=(const MATRIX3D& matrix);

	//length
	double GetLength() const;
	double GetLengthXY() const;
	double GetLengthYZ() const;
	double GetLengthZX() const;

	CVector3D GetNormal() const;
	void		Normalize();
	BOOL		IsZeroLength() const;
};

class AFX_EXT_CLASS CMatrix3D : public MATRIX3D
{
public:
	CMatrix3D();
	CMatrix3D(const MATRIX3D&);
	CMatrix3D(const double *);
	virtual ~CMatrix3D();
public:
	//operators
	CMatrix3D operator*(const MATRIX3D& matrix)const;
	void operator*=(const MATRIX3D& matrix);

	//methods
```

```
        void      IdenticalMatrix();
        double GetValue() const;

public:
        // static member functions
        static double GetValue(double a00, double a01, double a02,
                                            double a10, double a11, double a12,
                                            double a20, double a21, double a22);
        static CMatrix3D CreateMirrorMatrix(VECTOR3D plnNorm);
        static CMatrix3D CreateRotateMatrix(double da,VECTOR3D bv);
        static CMatrix3D CreateScaleMatrix(double);
        static CMatrix3D CreateTransfMatrix(VECTOR3D vec);
} ;

// exported API functions
double      AFX_EXT_API _AngleBetween(VECTOR3D v1,VECTOR3D v2);
double      AFX_EXT_API _DistOf(POINT3D pt0, POINT3D pt1);
BOOL        AFX_EXT_API _IsParallel(VECTOR3D v0,VECTOR3D v1);
BOOL        AFX_EXT_API _IsOrthogonal(VECTOR3D v0,VECTOR3D v1);
#endif
```

2.5.2 文件 CadBase.cpp

```
////////////////////////////////////////////////////CadBase.cpp/////////////////////////////////////////

#include "stdafx.h"
#include "cadbase.h"
#include "math.h"

/*****************************************************************************/
/     CLASS NAME: CPoint3D
/     CLASS DESCRIPATION: Designed for 2 dimensional point
/     CREATED BY: Olive Wang in Apr.28,2000
/     MODIFIED BY:
*****************************************************************************/
// constructor && destructor
CPoint3D::CPoint3D()
{
        x=0.0;
        y=0.0;
        z=0.0;
}

CPoint3D::CPoint3D(double ix,double iy,double iz)
{
        x = ix;
        y = iy;
```

```
	z = iz;
}

CPoint3D::CPoint3D(const double*p)
{
	x=p[0];
	y=p[1];
	z=p[2];
}

CPoint3D::CPoint3D(POINT3D p)
{
	x=p.x;
	y=p.y;
	z=p.z;
}

CPoint3D::~CPoint3D()
{
}

//operators
CPoint3D CPoint3D::operator*(const MATRIX3D& matrix) const
{
	double rx,ry,rz,sc;
	rx = x * matrix.A[0][0] + y * matrix.A[1][0] + z * matrix.A[2][0] + matrix.A[3][0];
	ry = x * matrix.A[0][1] + y * matrix.A[1][1] + z * matrix.A[2][1] + matrix.A[3][1];
	rz = x * matrix.A[0][2] + y * matrix.A[1][2] + z * matrix.A[2][2] + matrix.A[3][2];
	sc = x * matrix.A[0][3] + y * matrix.A[1][3] + z * matrix.A[2][3] + matrix.A[3][3];
	rx /= sc;
	ry /= sc;
	rz /= sc;
	return CPoint3D(rx,ry,rz);
}

void    CPoint3D::operator*=(const MATRIX3D& matrix)
{
	(*this) = (*this)*matrix;
}

// offsetting with vector
CPoint3D CPoint3D::operator+(VECTOR3D v) const
{
	return CPoint3D(x+v.dx,y+v.dy,z+v.dz);
}

void CPoint3D::operator+=(VECTOR3D v)
```

```
{
	x+=v.dx;
	y+=v.dy;
	z+=v.dz;
}

CPoint3D CPoint3D::operator-(VECTOR3D v) const
{
	return CPoint3D(x-v.dx,y-v.dy,z-v.dz);
}

void CPoint3D::operator-=(VECTOR3D v)
{
	x-=v.dx;
	y-=v.dy;
	z-=v.dz;
}

// derive vector = this point - sp
CVector3D CPoint3D::operator-(POINT3D sp) const
{
	return CVector3D(x-sp.x,y-sp.y,z-sp.z);
}

BOOL CPoint3D::operator==(POINT3D pos) const
{
	CVector3D	vect(x-pos.x,y-pos.y,z-pos.z);

	if( IS_ZERO( vect.GetLength( ) ) ) return TRUE;
	else return FALSE;

}
BOOL CPoint3D::operator!=(POINT3D pos) const
{
	CVector3D	vect(x-pos.x,y-pos.y,z-pos.z);

	if( IS_ZERO( vect.GetLength( ) ) ) return FALSE;
	else return TRUE;
}


/*****************************************************************************
/	CLASS NAME:	CVector3D
/	DESCRIPTION: designed for 3 dimensional vector
*****************************************************************************/
// constructor&&destructor
CVector3D::CVector3D()
```

```
{
	dx=0.0;
	dy=0.0;
	dz=0.0;
}

CVector3D::CVector3D(double ix,double iy,double iz)
{
	dx=ix;
	dy=iy;
	dz=iz;
}

CVector3D::CVector3D(const double* pv)
{
	dx=pv[0];
	dy=pv[1];
	dz=pv[2];
}

CVector3D::CVector3D(VECTOR3D v)
{
	dx=v.dx;
	dy=v.dy;
	dz=v.dz;
}

CVector3D::~CVector3D()
{
}

CVector3D CVector3D::operator+(VECTOR3D v) const
{
	return CVector3D(dx+v.dx,dy+v.dy,dz+v.dz);
}

CVector3D CVector3D::operator-(VECTOR3D v) const
{
	return CVector3D(dx-v.dx,dy-v.dy,dz-v.dz);
}

void CVector3D::operator+=(VECTOR3D v)
{
	dx += v.dx;
	dy += v.dy;
	dz += v.dz;
}
```

```
void CVector3D::operator-=(VECTOR3D v)
{
	dx -= v.dx;
	dy -= v.dy;
	dz -= v.dz;
}

CVector3D CVector3D::operator*(double d) const
{
	return CVector3D(dx*d,dy*d,dz*d);
}

void CVector3D::operator*=(double d)
{
	dx *= d;
	dy *= d;
	dz *= d;
}

CVector3D CVector3D::operator/(double d) const
{
	return CVector3D(dx/d,dy/d,dz/d);
}

void CVector3D::operator/=(double d)
{
	dx /= d;
	dy /= d;
	dz /= d;
}

// cross product
CVector3D CVector3D::operator*(VECTOR3D v) const
{
	return CVector3D(dy*v.dz-v.dy*dz,v.dx*dz-dx*v.dz,dx*v.dy-v.dx*dy);
}
// dot product
double CVector3D::operator|(VECTOR3D v) const
{
	return dx*v.dx+dy*v.dy+dz*v.dz;
}

//methods implementation
double CVector3D::GetLength() const
{
	return sqrt(dx*dx+dy*dy+dz*dz);
}
```

```
double CVector3D::GetLengthXY() const
{
	return sqrt(dx*dx+dy*dy);
}

double CVector3D::GetLengthYZ() const
{
	return sqrt(dy*dy+dz*dz);
}

double CVector3D::GetLengthZX() const
{
	return sqrt(dx*dx+dz*dz);
}

CVector3D CVector3D::GetNormal() const
{
	double len = GetLength();
	return CVector3D(dx/len,dy/len,dz/len);
}

void CVector3D::Normalize()
{
	double len = GetLength();
	dx /= len;
	dy /= len;
	dz /= len;
}

BOOL CVector3D::IsZeroLength() const
{
	return IS_ZERO(GetLength());
}

CVector3D CVector3D::operator*(const MATRIX3D& matrix) const
{
	double rx,ry,rz,sc;
	rx = dx * matrix.A[0][0] + dy * matrix.A[1][0] + dz * matrix.A[2][0] + matrix.A[3][0];
	ry = dx * matrix.A[0][1] + dy * matrix.A[1][1] + dz * matrix.A[2][1] + matrix.A[3][1];
	rz = dx * matrix.A[0][2] + dy * matrix.A[1][2] + dz * matrix.A[2][2] + matrix.A[3][2];
	sc = dx * matrix.A[0][3] + dy * matrix.A[1][3] + dz * matrix.A[2][3] + matrix.A[3][3];
	rx /= sc;
	ry /= sc;
	rz /= sc;
	return CVector3D(rx,ry,rz);
}
```

```
void    CVector3D::operator*=(const MATRIX3D& matrix)
{
        (*this) = (*this)*matrix;
}

/*****************************************************************************
/     CLASS NAME: CMatrix3D
/     DESCRIPTION: designed for 3 dimensional matrix
*****************************************************************************/
//construction&&destruction
CMatrix3D::CMatrix3D()
{
        for(int i=0;i<4;i++)
                for(int j=0;j<4;j++)
                {
                        A[i][j] = (i==j)?1.0:0.0;
                }
}

CMatrix3D::CMatrix3D(const MATRIX3D& matrix)
{
        for(int i=0;i<4;i++)
                for(int j=0;j<4;j++)
                {
                        A[i][j] = matrix.A[i][j];
                }
}

CMatrix3D::CMatrix3D(const double *matrix)
{
        for(int i=0;i<4;i++)
                for(int j=0;j<4;j++)
                {
                        A[i][j] = matrix[i*4+j];
                }
}

CMatrix3D::~CMatrix3D()
{
}

//operators
CMatrix3D CMatrix3D::operator*(const MATRIX3D& matrix2) const
{
        CMatrix3D matrix;
        for(int i=0;i<4;i++)
        for(int j=0;j<4;j++){
```

```
            matrix.A[i][j] = A[i][0]*matrix2.A[0][j]
                                + A[i][1]*matrix2.A[1][j]
                                + A[i][2]*matrix2.A[2][j]
                                + A[i][3]*matrix2.A[3][j];
        }
        return matrix;
}

void CMatrix3D::operator*=(const MATRIX3D& matrix)
{
        (*this) = (*this)*matrix;
}

//methods
void CMatrix3D::IdenticalMatrix()
{
        for(int i=0;i<4;i++)
        for(int j=0;j<4;j++)
        {
                A[i][j] = (i==j)?1.0:0.0;
        }
}

double CMatrix3D::GetValue() const
{
        return    A[0][0]*A[1][1]*A[2][2] +
                  A[0][1]*A[1][2]*A[2][0] +
                  A[0][2]*A[1][0]*A[2][1] -
                  A[0][2]*A[1][1]*A[2][0] -
                  A[0][1]*A[1][0]*A[2][2] -
                  A[0][0]*A[1][2]*A[2][1];
}

// static member functions
double CMatrix3D::GetValue(double a00, double a01, double a02,
                                        double a10, double a11, double a12,
                                        double a20, double a21, double a22)
{
        return    a00*a11*a22 +
                  a01*a12*a20 +
                  a02*a10*a21 -
                  a02*a11*a20 -
                  a01*a10*a22 -
                  a00*a12*a21;
}

CMatrix3D CMatrix3D::CreateMirrorMatrix(VECTOR3D v)
```

```
{
	double len=((CVector3D)v).GetLength();
	CMatrix3D matrix;
	matrix.A[0][0]= (v.dx*v.dx -1.0)*2.0/len/len;
	matrix.A[1][1]= (v.dy*v.dy -1.0)*2.0/len/len;
	matrix.A[2][2]= (v.dz*v.dz -1.0)*2.0/len/len;
	matrix.A[0][1]=matrix.A[1][0]= v.dx*v.dy*2.0/len/len;
	matrix.A[0][2]=matrix.A[2][0]= v.dx*v.dz*2.0/len/len;
	matrix.A[1][2]=matrix.A[2][1]= v.dz*v.dy*2.0/len/len;
	return matrix;
}

CMatrix3D CMatrix3D::CreateRotateMatrix(double da,VECTOR3D v)
{
	CMatrix3D R;
	CVector3D bv(v);

	if(IS_ZERO(da)) return R;

	ASSERT(!bv.IsZeroLength());

	double lxy=bv.GetLengthXY();
	if(IS_ZERO(lxy))
	{
		if(bv.dz < 0.0) da *= -1.0;
		R.A[0][0]=R.A[1][1]=cos(da);
		R.A[0][1]=sin(da);R.A[1][0]=-sin(da);
		return R;
	}
	double lyz=bv.GetLengthYZ();
	if(IS_ZERO(lyz))
	{
		if(bv.dx < 0.0) da *= -1.0;
		R.A[2][2]=R.A[1][1]=cos(da);
		R.A[1][2]=sin(da);R.A[2][1]= -sin(da);
		return R;
	}
	double lxz=bv.GetLengthZX();
	if(IS_ZERO(lxz))
	{
		if(bv.dy < 0.0) da *= -1.0;
		R.A[0][0]=R.A[2][2]=cos(da);
		R.A[0][2]=-sin(da);R.A[2][0]=sin(da);
		return R;
	}

	CMatrix3D Rz;
```

```
	Rz.A[0][0]=Rz.A[1][1]=bv.dy/lxy;
	Rz.A[0][1]=bv.dx/lxy;Rz.A[1][0]= -bv.dx/lxy;

	double len=bv.GetLength();
	CMatrix3D Rx;
	Rx.A[2][2]=Rx.A[1][1]=bv.dz/len;
	Rx.A[1][2]=lxy/len;Rx.A[2][1]= -lxy/len;

	R.A[0][0]=R.A[1][1]=cos(da);
	R.A[0][1]=sin(da);R.A[1][0]= -sin(da);

	CMatrix3D Rxn;
	Rxn.A[2][2]=Rxn.A[1][1]=bv.dz/len;
	Rxn.A[2][1]=lxy/len;Rxn.A[1][2]= -lxy/len;

	CMatrix3D Rzn;
	Rzn.A[0][0]=Rzn.A[1][1]=bv.dy/lxy;
	Rzn.A[1][0]=bv.dx/lxy;Rzn.A[0][1]= - bv.dx/lxy;

	return Rz*Rx*R*Rxn*Rzn;
}

CMatrix3D CMatrix3D::CreateScaleMatrix(double d)
{
	CMatrix3D m;
	m.A[0][0]=m.A[1][1]=m.A[2][2]=d;
	return m;
}

CMatrix3D CMatrix3D::CreateTransfMatrix(VECTOR3D vec)
{
	CMatrix3D m;
	m.A[3][0]=vec.dx;
	m.A[3][1]=vec.dy;
	m.A[3][2]=vec.dz;
	return m;
}
```

2.5.3 文件 CadBase1.cpp

```
#include "stdafx.h"
#include "cadbase.h"
#include "math.h"


double _AngleBetween(VECTOR3D v1,VECTOR3D v2)
{
```

```
    if(_IsParallel(v1,v2))    return 0;
    CVector3D cv1(v1),cv2(v2);
    return acos((cv1|cv2.GetNormal())/cv1.GetLength());
}


double _DistOf(POINT3D pt0,POINT3D pt1)
{
    CVector3D vec(pt1.x-pt0.x,pt1.y-pt0.y);
    return vec.GetLength();
}

BOOL _IsParallel(VECTOR3D v0,VECTOR3D v1)
{
    CVector3D cv0(v0),cv1(v1);
    return IS_ZERO((cv0*cv1).GetLength());
}

BOOL _IsOrthogonal(VECTOR3D v0,VECTOR3D v1)
{
    CVector3D cv0(v0),cv1(v1);
    return IS_ZERO(cv0|cv1);
}
```

本章相关程序

- ch2\GeomCalc：**几何基本工具库** GeomCalc **的工程。**
- ch2\GeomCalc.dll：**动态链接库。**
- ch2\GeomCalc.lib：**连接库。**
- ch2\cadbase.h：**输出类与输出函数的头文件。**

第 3 章　基于 MFC 的 OpenGL Windows 程序的创建

本章要点：

- OpenGL 概述。
- OpenGL 与 GDI。
- 渲染场境（Rendering Context）。
- Wiggle 函数的使用。
- 为 OpenGL Window 程序设置像素格式。
- 在 Windows 环境下使用 OpenGL。

3.1　OpenGL 介绍

OpenGL（Open Graphics Library）是一个优秀的三维图形硬件的软件接口，实际上是一个三维图形和模型库。使用 OpenGL 可以绘制出真实感很强的三维图形，且由于越来越多的高档图形加速卡支持 OpenGL，所以使用 OpenGL 绘图可以获得很快的执行速度。OpenGL 是一个与硬件无关的图形编程接口，可以在不同的硬件平台上实现。OpenGL 最初是 SGI 公司为其图形工作站开发的可独立于窗口操作系统和硬件设备的图形开发环境，其目的是将用户从具体的硬件环境和操作系统中解放出来，而可以完全不去理解这些系统的结构和指令系统。由于它在三维真实感图形制作中具有优秀的性能，目前 OpenGL 已被广泛接受，已有几十家大公司采用 OpenGL 作为标准图形的软件接口。OpenGL 事实上已成为高性能的图形和交互视景处理的工业标准，能够在 Windows 95/98、Windows NT/2000、Macos、Beos、OS/2、及 Unix 上应用。三维 CAD 软件广泛使用 OpenGL 进行图形绘制与仿真。

作为图形软件接口，OpenGL 由几百个函数组成，用于访问和操作图形硬件所提供的各种功能。OpenGL 本身不提供高级的造型命令，而是通过基本的几何图元——点、线和多边形来建立几何模型。用户则是通过这些基本图元来建立高级的几何模型。归纳起来，OpenGL 提供的功能包括以下几个方面：

（1）图形绘制。OpenGL 使用基本图元——点、线、多边形来绘制三维图形和场景。用户定义的高级模型和场景都可以通过这些基本图元组成。

（2）变换操作。用户往往希望能够控制观察物体的角度、方向和距离。如在 CAD 软件中，用户可以从不同的观察角度和距离来观察几何模型，可以对模型的显示进行放大、缩小、剪切等。在 OpenGL 中，这些功能是通过一系列的变换操作来实现的。OpenGL 提供了一系列基本的变换操作：如使用投影变换定义一个视景体；几何变换可以使物体在三维场景中平移、旋转和缩放；视点变换可以使得用户从不同角度去观察物体；裁剪变换可以定义裁剪平面；视口变换决定怎样把图形映射到屏幕上。

（3）颜色模式。OpenGL 提供两种可供选择的颜色模式：RGBA 模式和颜色索引模式。在 RGBA 模式中，颜色由红、绿、蓝三种颜色分量来描述；在颜色索引模式中，颜色值由颜色索引表中的索引值来指定。

（4）光照。光照是生成真实感图形的重要环节。OpenGL 中可以通过配置如下四种光，即环境光、漫反射光、辐射光和镜面光，来定义一个光源。用户可以在应用程序中定义多个光源，以及它们的位置和属性，以达到需要的光照效果。

（5）图像效果增强。OpenGL 提供了一系列的增强图像效果的函数，它们通过反走样、融合、雾化等来增强图像效果。反走样改善图像中两个面交接处的锯齿效果；融合可以产生半透明效果；雾化则可以模拟场景，使场景更逼真。

（6）位图和图像。OpenGL 提供了一系列的函数来实现位图显示与操作。

（7）纹理映射。纹理映射是增强三维真实感图形效果的重要工具。当显示一个三维对象时，在模型表面贴上模拟自然界的纹理，可以使三维图形显得更逼真、更自然。

（8）交互与动画。OpenGL 提供的选择模式（SELECT）和反馈模式（FEEDBACK）方便了图形和用户之间交互功能的实现。OpenGL 的双缓存技术能够使图像帧与帧之间平滑地过渡，为实现图形的动画效果提供了可能。

3.2 在 Windows 环境下使用 OpenGL

本章的重点在于介绍在 Windows 环境下使用 OpenGL 进行三维图形绘制的有关知识。

3.2.1 OpenGL 的函数库

OpenGL 本身不包括任何 Windows 窗口管理和用户交互的函数。不同的操作系统会为 OpenGL 提供不同的支持。在 Windows NT3.5 以及 Windows 95 以上的 Windows 操作系统中增加了对 OpenGL 的支持，提供了一套 OpenGL 的动态库。Windows 下的应用程序通过调用这些 OpenGL 动态库输出的 API 函数来使用 OpenGL。

在微机版本中，OpenGL 提供了三个函数库，它们是基本库（OpenGL library）、实用库（Utility library）和辅助库（Auxiliary or Toolkit）。对这三个库的介绍如下：

（1）OpenGL 基本库是 OpenGL 的核心函数库，在这个函数库中，提供了一百多个函数。这些函数在 opengl32.dll 中实现，并在头文件 gl.h 中声明。这个基本库的导出函数都以 gl 为前缀。OpenGL 提供的所有操作都由这些函数来实现。

（2）OpenGL 实用库中包括了大约四十多个输出函数。它们的作用在于提供一些更方便于使用的函数给用户，如绘制球（sphere）、圆柱（cylinder）、圆环（tours）等这些复杂形状。这些函数都由 OpenGL 的基本库中的函数写成，所以和基本库一样，能够在所有支持 OpenGL 规范的平台上使用。实用库函数在 glu32.dll 中输出，在头文件 glu.h 中声明，所有函数以 glu 为前缀。

（3）OpenGL 的辅助库实际上不能算 OpenGL 规范的一部分。它更是一个工具包，提供一些独立于平台的框架供调用 OpenGL 的函数。如在辅助库中，提供了一些基本的窗口管理函数、事件处理函数和简单的模型制作函数。辅助库的函数在 glaux.dll 中输出，在头文件 glaux.h 中声明，函数以 aux 为前缀。

3.2.2 OpenGL 与 GDI

很多开发人员在使用 OpenGL 之前，习惯于使用 GDI（Graphical Device Interface）函数，即图形设备接口函数来绘制图形。GDI 是 Windows 体系结构中的重要组件，它为 Windows 提供了所有基本的绘图函数。GDI 与 Windows 系统中的图形设备驱动程序连接，通过调用驱动程序中相应的功能，以响应 Windows 图形函数的调用。这样，程序员只需要将绘图代码实现到 GDI 函数这个层次上，而不必去关心具体的硬件设备。

GDI 与 Windows 窗口的关联靠一个设备场境（Device Context）来实现。在 Windows 窗口中使用 GDI 时，需要创建一个设备场境对象。设备场境指定了许多如何在窗口上显示图形的信息，例如指定画笔和画刷的颜色、设置绘图模式、调色板、映射模式和其他图形模式。该场境作为一个参数被发送到 GDI 函数中。每个 Windows 窗口都有一个设备场境用于图形输出，而每个 GDI 函数都需要一个设备场境，用来标识这个函数将向哪一个窗口输出。一个程序中可以存在多个设备场境，但每个窗口只有一个它自己的设备场境。任何时候，图形绘制到显示器或者打印机时，它必须使用 GDI 函数。GDI 函数可以绘制点、直线、矩形、多边形、椭圆、位图和文本。

OpenGL Windows 应用程序不使用 GDI 函数，而是直接使用 OpenGL API 函数进行图形绘制。OpenGL 与 GDI 的关系如图 3-1 所示。

Microsoft 还提供了一套“Wiggle”函数，用于 OpenGL API 和 Microsoft Windows GDI 的连接。与使用 GDI 在 Windows 下绘图不同，使用 OpenGL 绘图后，GDI 中的画笔（Pens）、画刷（Brushes）、字体（Fonts）以及其他 GDI 对象将不再起作用。正如 GDI 使用设备场境（Device Context）用于控制 Windows 中的图形绘制，OpenGL 使用了一个渲染场境（Rendering Context）。这个渲染场境通过与设备场境相关联，从而也实现了与窗口的关联，下面还将进一步论述这个问题。

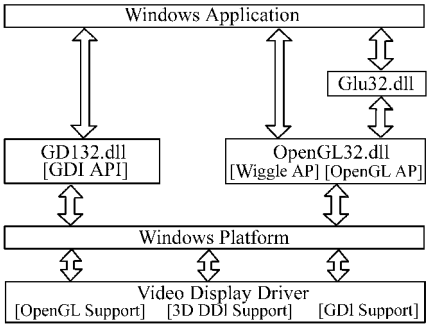


图 3-1 OpenGL 工作结构

3.2.3 渲染场境

1．渲染场境和设备场境

正如使用 GDI 绘图时，在每个 Windows 程序的窗口中都拥有一个设备场境（Device Context），用于窗口中的图形绘制。使用 OpenGL 在 Windows 窗口中绘制图形时，也需要在每个窗口中嵌入一个渲染场境（Rendering Context）。渲染场境也被称为绘图描述表或着色描述表，它保存了与 OpenGL 发生联系的信息，由这个渲染场境负责处理应用程序发出的 OpenGL 命令。所以在发出 OpenGL 命令之前，必须首先为该窗口创建一个渲染场境。

一个应用程序中可以具有一个或多个渲染场境，但同时只能有一个渲染场境被设为当前的渲染场境（Current Rendering Context）。程序中发出的 OpenGL 命令只输出给当前的渲染场境。例如，在一个应用程序中同时开了两个窗口，当程序中发出一个 OpenGL 的命令时，系统如何知道这个命令是传给哪一个窗口的呢？为了使系统知道当前命令将发给哪一个窗口，

所以在一个应用程序中（严格地说是在一个线程中）只能同时存在一个当前化的渲染场境。当一个渲染场境被设为当前场境后，由于该渲染场境关联了一个 Windows 的设备场境，从而能够找到它所关联的窗口，于是 OpenGL 向该窗口绘制图像。这个过程如图 3-2 所示。

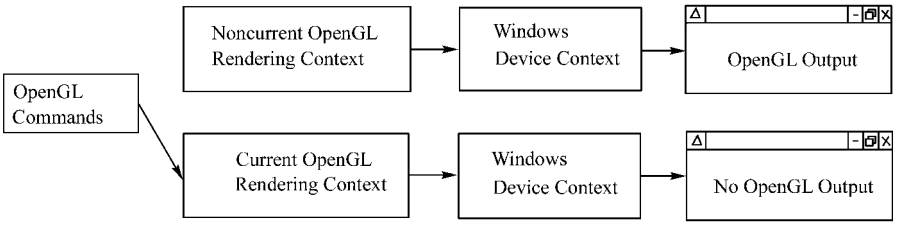


图 3-2　OpenGL 命令与当前渲染场境

2．Wiggle 函数/对渲染场境的操作

严格地说，渲染场境不属于 OpenGL 的概念，而更是 Windows 窗口中一个附加的概念，以支持窗口与 OpenGL 的连接。Windows API 中增加 Wiggle 函数的目的在于增加 Windows 窗口对 OpenGL 的支持。这些函数都以 wgl 为其前缀，其中最常用的几个与渲染场境相关的 Wiggle 函数是：

```
HGLRC wglCreateContext( HDC    hdc );
BOOL wglDeleteContext( HGLRC hglrc );
BOOL wglMakeCurrent( HDC    hdc, HGLRC    hglrc );
HGLRC wglGetCurrentContext();
HDC wglGetCurrentDC();
```

下面是对这些 Wiggle 函数的介绍。

（1）函数 wglCreateContext()。该函数输入一个 GDI 的设备场境句柄 hdc，创建与这个设备场境相关联的 OpenGL 渲染场境。函数的原型如下：

```
HGLRC wglCreateContext( HDC    hdc );
```

返回值：

如果渲染场境创建成功，返回所创建的渲染场境的句柄；否则函数返回 NULL。

参数：

hdc——窗口的设备场境句柄。

在调用这个函数创建渲染场境之前，必须先调用函数 SetPixelFormat()设置与设备场境相匹配的像素格式。

（2）函数 wglDeleteContext()。所创建的渲染场境在用完之后需要将它删除。函数 wglDeleteContext()用于删除一个渲染场境。函数的原型如下：

```
BOOL wglDeleteContext( HGLRC hglrc );
```

返回值：

若删除成功，返回 TRUE；否则返回 FALSE。

参数：

hglrc——要被删除的渲染场境的句柄。

（3）函数 wglMakeCurrent()。函数 wglMakeCurrent()用于当前化一个渲染场境，即将一个渲染场境设置为线程的当前渲染场境。线程中一旦当前化了一个渲染场境，所有随后的 OpenGL 命令都发往这个当前的渲染场境中处理，并通过与该渲染场境关联的设备场境实现窗口的场景绘制。

可以用该函数改变线程中的当前渲染场境。如将另一个渲染场境设为当前的，或者将当前渲染场景设为 NULL，即没有当前的渲染场境。

该函数的原型如下：

```
BOOL wglMakeCurrent( HDC    hdc, HGLRC    hglrc );
```

返回值：

若设置成功，返回 TRUE；否则返回 FALSE。

参数：

hdc——与渲染场境相关联的设备场境的句柄。线程中随后调用的 OpenGL 命令将绘制在这个句柄标识的设备上。

hglrc——将要被设置为当前化的设备场境的句柄。当 hglrc 被设置为 NULL 时，执行该函数将线程的当前渲染场境非当前化。

（4）函数 wglGetCurrentContext()。使用该函数获得线程中当前的渲染场境。该函数的原型如下：

```
HGLRC wglGetCurrentContext();
```

返回值：

返回当前的渲染场境句柄。若线程中没有当前的渲染场境，返回值为 NULL。

（5）函数 wglGetCurrentDC()。使用该函数获得与当前渲染场境相关联的设备场境。该函数的原型如下：

```
HDC wglGetCurrentDC();
```

返回值：

返回与当前渲染场境相关联的设备场境句柄。若线程中没有当前的渲染场境，返回值为 NULL。

下面给出一个操作渲染场境的主要过程的程序清单。

```
//////////创建和使用 OpenGL 渲染场境的主要步骤//////////////////////
HDC       hdc;
HGLRC    hglrc;

//  创建一个渲染场境
hglrc = wglCreateContext (hdc);

//当前化该渲染场境
wglMakeCurrent (hdc, hglrc);

//  调用 OpenGL 函数
……..
```

```
//当 OpenGL 渲染场境使用完毕
//非当前化该渲染场境
wglMakeCurrent (NULL, NULL) ;

// 删除所创建的渲染场境
wglDeleteContext (hglrc);
```

3.2.4 像素格式

要在一个窗口中使用 OpenGL，**还需要为它选择一个合适的像素格式（**Pixel Format**）。和渲染场境的概念类似，像素格式并不属于** OpenGL **的概念。它作为** Windows API **的一个扩展，用于支持** Windows **下的** OpenGL **功能。在** OpenGL **的图形操作步骤中，最终的环节是将图形光栅化，即把图形转化成可在计算机屏幕上显示的像素信息，并将这些像素写到帧存中以实现屏幕显示。**

像素格式为设备场境设置一些与 OpenGL **相关的属性。在像素格式中规定了和像素操作有关的信息，诸如：像素缓存是单缓存还是双缓存；像素的颜色模式是** RGBA **模式，还是索引模式；颜色的位数；深度缓存的位数等。因而，在创建一个渲染场境之前，需要首先为窗口的设备场境设置一个像素格式。**

Windows **中，使用结构** PIXELFORMATDESCRIPTOR **来设置像素格式，并提供了四个函数用于操作像素格式。设置像素格式的步骤如下：**

第一步，根据需要，填写结构 PIXELFORMATDESCRIPTOR**，因为像素格式是用这个结构来描述的；**

第二步，调用函数 ChoosePixelFormat()**，将填写好的像素格式传递给该函数，函数** ChoosePixelFormat()**返回一个整形的序号。这个序号标识一个当前设备场境中所能提供的，且与所要求的像素格式最为匹配的像素格式；**

第三步，将这个返回的像素格式序号传递给函数 SetPixelFormat()**，使之成为设备场境的像素格式。**

对像素格式的数据结构以及操作像素格式的几个常用函数介绍如下。

（1）结构 PIXELFORMATDESCRIPTOR

该结构的定义如下：

```
typedef struct tagPIXELFORMATDESCRIPTOR {
    WORD    nSize;
    WORD    nVersion;
    DWORD dwFlags;
    BYTE    iPixelType;
    BYTE    cColorBits;
    BYTE    cRedBits;
    BYTE    cRedShift;
    BYTE    cGreenBits;
    BYTE    cGreenShift;
    BYTE    cBlueBits;
    BYTE    cBlueShift;
    BYTE    cAlphaBits;
```

```
    BYTE    cAlphaShift;
    BYTE    cAccumBits;
    BYTE    cAccumRedBits;
    BYTE    cAccumGreenBits;
    BYTE    cAccumBlueBits;
    BYTE    cAccumAlphaBits;
    BYTE    cDepthBits;
    BYTE    cStencilBits;
    BYTE    cAuxBuffers;
    BYTE    iLayerType;
    BYTE    bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
    DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

对结构中成员的解释如下：

- nSize：**该结构的尺寸（所占用的字节数）。**
- nVersion：**该结构的版本号，现在应设为** 1。
- dwFlags：**像素缓存的属性，告诉** OpenGL **怎样处理像素，可以是下列标识常量的逻辑或。**

PFD_DOUBLEBUFFER	PFD_STEREO
PFD_DRAW_TO_WINDOW	PFD_DRAW_TO_BITMAP
PFD_SUPPORT_GDI	PFD_SUPPORT_OPENGL
PFD_GENERIC_FORMAT	PFD_NEED_PALETTE
PFD_NEED_SYSTEM_PALETTE	PFD_SWAP_LAYER_BUFFERS

- IPixelType：**说明像素的颜色模式。存在两个选项** PFD_TYPE_RGBA **和** PFD_TYPE_COLORINDEX，**分别指** RGBA **模式和颜色索引模式。**
- cColorBits：**颜色的位数，如** 8 **位代表** 256 **色，**24 **位代表真彩色。**
- cRedBits，cRedShift，cGreenBits，cGreenShift，cBlueBits，cBlueShift，cAlphaBits, cAlphaShift：**通常不使用，一般情况下均置为** 0，**它们的意义分别如下：**

cRedBits：**使用** RGBA **模式时，红色组件所使用的位数；**
cRedShift：**使用** RGBA **模式时，红色组件可调节的位数；**
cGreenBits：**使用** RGBA **模式时，绿色组件所使用的位数；**
cGreenShift：**使用** RGBA **模式时，绿色组件可调节的位数；**
cBlueBits：**使用** RGBA **模式时，蓝色组件所使用的位数；**
cBlueShift：**使用** RGBA **模式时，蓝色组件可调节的位数；**
cAlphaBits：**使用** RGBA **模式时，**Alpha **组件所使用的位数；**
cAlphaShift：**使用** RGBA **模式时，**Alpha **组件可调节的位数。**

- cAccumBits: **累计缓存的位数。**
- cAccumRedBits，cAccumGreenBits，cAccumBlueBits，cAccumAlphaBits: **一般情况下均置为** 0，**它们的意义介绍如下：**

cAccumRedBits：累计缓存中红色组件的位数；

cAccumGreenBits：累计缓存中绿色组件的位数；

cAccumBlueBits：累计缓存中蓝色组件的位数；

cAccumAlphaBits：累计缓存中 Alpha 组件的位数。

- cDepthBits：深度缓存的位数。
- cStencilBits：模板缓存的位数。
- cAuxBuffers： 辅助缓存的位数，一般情况下置为 0。
- iLayerType：OpenGL 的早期属性，现已不用，可忽略，一般情况下置为 0。
- bReserved：必须设为 0。
- dwLayerMask：NT 下不支持，设为 0。
- dwVisibleMask：NT 下不支持，设为 0。
- dwDamageMask：NT 下不支持，设为 0。

（2）函数 ChoosePixelFormat()

该函数返回与所定义的像素结构最匹配的线程中当前的渲染场境。该函数的原型如下：

```
HGLRC wglGetCurrentContext();
```

返回值：

返回当前的渲染场境句柄。若线程中没有当前的渲染场境，返回值为 NULL。

（3）函数 SetPixelFormat()

这个函数为指定的设备场境设置像素格式。该函数的原型如下：

```
BOOL SetPixelFormat(  HDC  hdc,  int  iPixelFormat,
          CONST PIXELFORMATDESCRIPTOR * ppfd );
```

返回值：

若像素格式设置成功，返回 TRUE；否则返回 FALSE。

参数：

hdc——指定将被设置像素格式的设备场境。

iPixelFormat——指定被设置像素格式的索引号，这个索引号从 1 开始。

ppfd——指向一个 PIXELFORMATDESCRIPTOR 的指针。

（4）函数 DescribePixelFormat()

这个函数获得被 PixelFormat 指定的像素格式的信息，PixelFormat 是和设备场境相关联的。该函数的原型如下：

```
int DescribePixelFormat(HDC  hdc,  int  iPixelFormat, UINT  nBytes, LPPIXELFORMATDESCR-
IPTOR * ppfd );
```

返回值：

若函数调用成功，返回值是设备场境的像素格式的最大索引；否则返回 0。

参数：

hdc——指定一个设备场境。

iPixelFormat——指定像素格式的索引。

nBytes——结构 ppfd 所使用内存空间的大小。

ppfd——指向一个像素格式描述结构的指针，这个结构存放返回的像素格式的信息。

（5）函数 GetPixelFormat()

这个函数返回设备场境中被指定的像素格式的索引号。函数的原型如下：

```
int GetPixelFormat( HDC    hdc )；
```

返回值：

若函数调用成功，返回被指定像素格式的索引号，这个索引号应大于等于 1；否则，返回 0。

参数：

hdc——设备场境的句柄。

下面给出一个设置像素格式的主要操作过程的程序清单。

```
///////////////设置像素格式的主要步骤//////////////////
int     pixelformat;
HDC hDC;

//填写像素格式描述结构
static PIXELFORMATDESCRIPTOR pfdWnd =
      {
            sizeof(PIXELFORMATDESCRIPTOR),
            1,
            PFD_DRAW_TO_WINDOW |
            PFD_SUPPORT_OPENGL |
            PFD_DOUBLEBUFFER,
            PFD_TYPE_RGBA,
            24,                                    // 24-bit color
            0, 0, 0, 0, 0, 0,
            0, 0, 0, 0, 0, 0, 0,
            32,                                    // 32-bit depth buffer
            0, 0,
            PFD_MAIN_PLANE,
            0, 0, 0, 0
      };

//选择一个最匹配的像素格式
pixelformat = ChoosePixelFormat(m_hDC, &pfdWnd);

//为设备场境设置像素格式
SetPixelFormat(m_hDC, pixelformat, &pfdWnd);
```

在开始使用 OpenGL 绘制之前，程序中必须先完成像素格式的设置和渲染场境的创建。由于设置像素格式需要设备场境，所以这个过程必须在窗口创建完成之后进行。通常在 Windows 消息 WM_CREATE 的映射函数 OnCreate()中进行像素格式的设置和渲染场境的创建。对于创建的渲染场境，最后还应该在 Windows 消息 WM_DESTROY 的映射函数 OnDestroy()

中予以删除。图 3-3 显示了在 Windows 窗口的有关环节中创建和使用渲染场境。

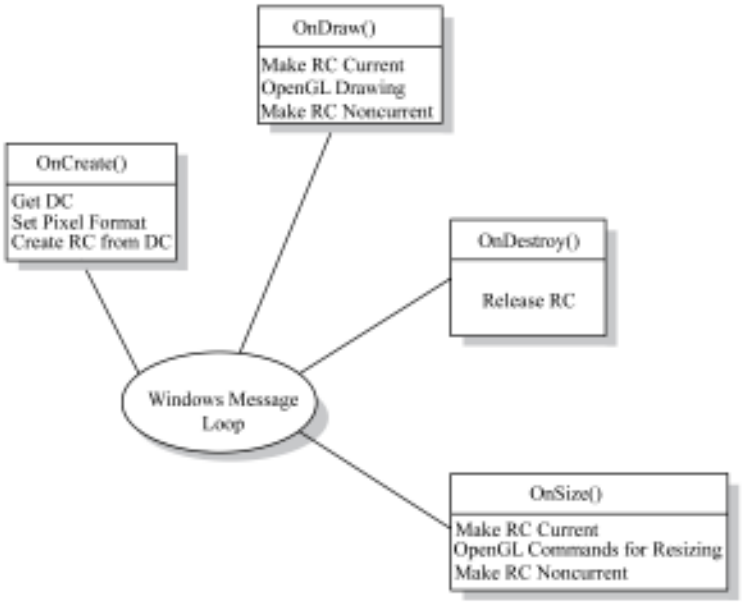


图 3-3　创建和使用渲染场境的环节

3.3　OpenGL MFC 应用程序创建实例

微软基础类库 MFC 为编写 OpenGL Windows 程序提供了良好的环境。本章将介绍如何在 MFC 环境下创建一个 OpenGL Windows 程序，即在 MFC 视图类 CView 的基础上派生一个封装了 OpenGL 的类 CGLView。类 CGLView 中封装并自动设置了渲染场境，使 Windows 窗口与 OpenGL 相关联,在 CGLView 的绘图函数 CGLView::RenderScene()中可以直接使用 OpenGL 命令进行图形绘制。

在 MFC 中，类 CView 是所有视图类的基类。类 CView 和文档基类 CDocument 相关联，提供了实现文档数据的屏幕显示和交互操作的机制。对于 CAD 系统来说，文档的主要数据是几何模型。在 CView 中，可直接使用设备场境类 CDC 来绘制简单的二维图形。在开发三维 CAD 程序时，需要这样一个视图类，在该视图类中能够直接使用 OpenGL 进行图形绘制，就像可直接调用 CDC 绘制二维图形一样方便。遗憾的是，Microsoft Visual C++中没有提供这样一个现成的类可直接用 OpenGL 进行图形绘制。因而，用户需要自己创建一个这样的视图类。下面的内容将介绍怎样在 CView 的基础上派生一个封装 OpenGL 的类 CGLView。用户还可以根据需要在 CGLView 的基础上再派生新的类，以方便在自己开发的 MFC 应用程序中使用 OpenGL。由于在创建 CGLView 过程中完成了 Windows 窗口与 OpenGL 的关联，因而在 CGLView 及 CGLView 的派生类中可直接调用 OpenGL 的绘图命令，就如同在 CView 的派生类中可直接使用 CDC 进行文本和图形的绘制。

在进行有一定规模的系统开发时，开发者更希望所开发的代码具有较强的独立性和模块化。在本章中，设计的 CGLView 将具备最基本的 OpenGL Windows 绘图功能。若进一步对它

的功能予以完善，可以在这个基础上创建一个专门用于 OpenGL Windows 绘图的动态链接库。在接下来的两章，将介绍一个这样的 DLL 的开发技术。在该 DLL 中输出一些支持 MFC 的 C++类，用于在 MFC 环境下的 OpenGL 图形绘制和交互。

在了解了创建渲染场境和设置像素格式的有关概念以后，可以按照下面的步骤开始建立一个 OpenGL Windows 程序。

3.3.1 创建一个应用程序框架

首先需要创建一个 MFC 文档/视图结构的应用程序框架，即一个 Visual C++的工程，然后再循序渐进完成整个 CGLView 类的设计。在这个工程中，因为要使用到 OpenGL，所以还必须连接 OpenGL 的几个连接库。项目的创建步骤如下：

（1）建立一个 MFC 文档/视图结构的工程。使用 Developer Studio 中的 AppWizard 建立一个新的 MFC 程序框架。执行如下步骤：

1）从 File 菜单选择 New 命令，弹出 New 对话框。

2）切换到 Project 选项卡，选择项目类型为 MFC AppWizard (exe)。

3）在 Project name 文本框中输入名字 GL，单击 OK 按钮。

4）选择单文档结构 (SDI)。

5）关闭所有的其他选项，如 OLE、ODBC、3D Controls 等等。

完成以上步骤后，将看到如图 3-4 所示的窗口。选择 Finish 按钮，Developer Studio 将产生一个单文本的文档/视图框架结构。这时，可以编译和试运行整个工程。如果编译运行正常，在工程中会生成一个派生的视类 CGLView，它分别在文件 GLView.h 和 GLView.cpp 中声明和实现。

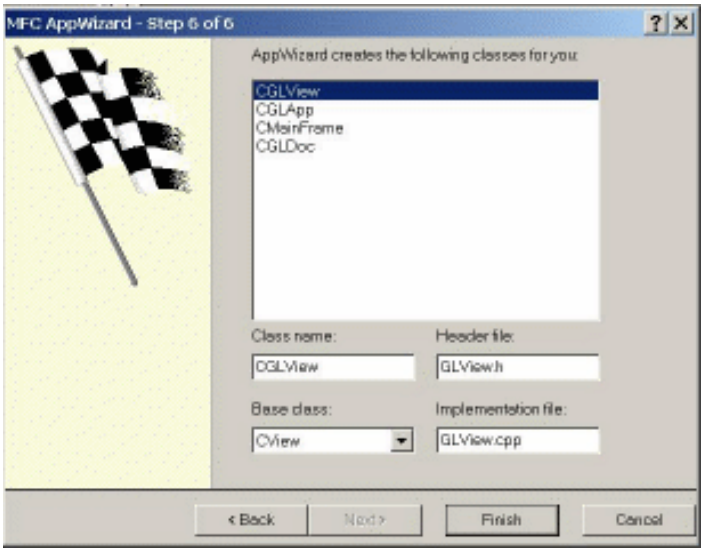


图 3-4　创建应用程序框架 GL

（2）在工程中加入 OpenGL 的连接库。为在工程中使用 OpenGL 的函数库，需要在工程

中加入 OpenGL 的三个连接库，它们是 openGL32.lib、glu32.lib 和 glaux.lib。步骤如下：

打开主菜单中的 Project，选择 Settings……，出现 Project Settings 对话框。选择 Link 页面。在如图 3-5 所示对话框中的 Object/library modules 编辑栏中填入“ opengl32.lib glu32.lib glaux.lib ”。

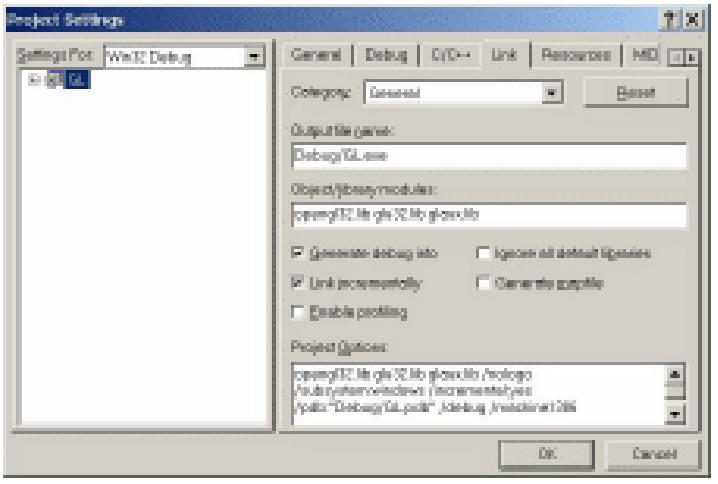


图 3-5　Project Settings 对话框

另一种加入 OpenGL 连接库的方法是直接在工程中加入以上连接文件。

在左边的浮动窗口 Workspace Bar 中，切换到 File View 一栏。鼠标点在 GL files 上，按鼠标右键后选择 Add Files to Project…，出现如图 3-6 所示的对话框 Insert Files into Project。选择 Visual C++的安装目录，进入 lib 子目录，分别选择文件 opengl32.lib、glu32.lib 和 glaux.lib ，然后按下 Add 按钮，将连接文件加到应用程序中去。

（3）添加头文件到 GLView.h 中。在文件 GLView.h 首部包括以下头文件：

```
#include <gl\gl.h>                    //OpenGL 基本库
#include <gl\glu.h>                   //OpenGL 实用库
```

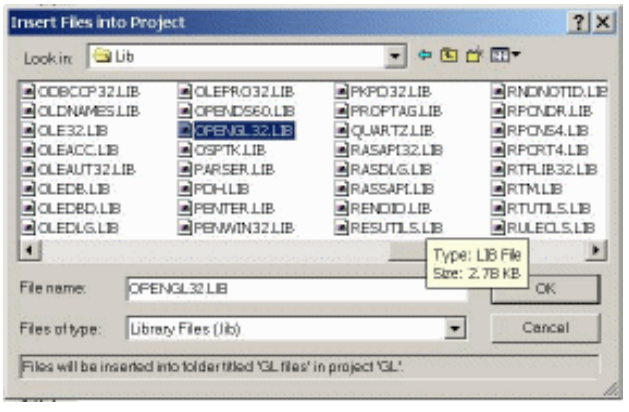


图 3-6　在工程中插入 OpenGL 连接库

若需要在程序中调用 OpenGL 的辅助库，则还需要包括辅助库的头文件：

```
#include <gl\glaux.h>                //OpenGL 辅助库
```

3.3.2 修改视图类 CGLView

现在对类 CGLView 进行修改。步骤如下：

（1）在 CGLView 中增加以下成员变量：

```
public:
    HGLRC    m_hRC;              //渲染场境句柄
    HDC      m_hDC;              //设备场境句柄
```

（2）在 CGLView 中增加以下成员函数：

```
public:
    virtual void RenderScene();          //用 OpenGL 绘制图形
private:
    void     GLInit();           //OpenGL 的初始化设置
    void     GLRelease();        //清除渲染场境

    void  GLResize(int cx,int cy);
    //根据视图窗口尺寸的变化，调整 OpenGL 的显示空间

    void     GLSetupRC(void *pData);
    //设置渲染场境
```

这些函数的作用在于对 OpenGL 进行必要的初始化设置，在 CGLView 的相关函数中需要调用这些函数。CGLView 中的函数调用关系如图 3-7 所示。这些函数的具体功能以及实现代码将在下面的篇幅中结合对它们的调用予以介绍。

（3）修改窗口风格设置。需要在函数 CGLView::PreCreateWindow(CREATESTRUCT& cs) 中增加对 Windows 窗口风格的设置。函数 PreCreateWindow()由 AppWizard 自动创建，允许用户在窗口创建之前修改 CREATESTRUCT 中的信息，以定义窗口类型。结构 CREATESTRUCT 中包含了所要创建的窗口的类型设置。

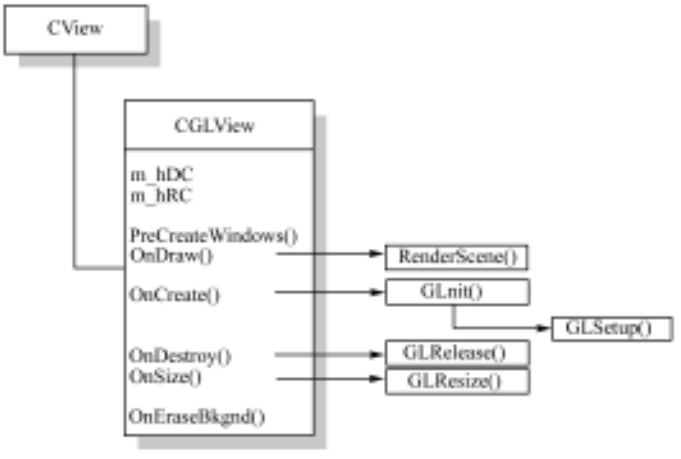


图 3-7　CGLView 中的函数调用关系

一般需要为窗口增加两个属性：WS_CLIPCHILDREN **和** WS_CLIPSIBLINGS**。对函数** PreCreateWindow()**修改如下：**

```
BOOL CGLView::PreCreateWindow(CREATESTRUCT& cs)
{
    //为 OpenGL 增加窗口风格设置
    cs.style |= WS_CLIPSIBLINGS|WS_CLIPCHILDREN;
    return CView::PreCreateWindow(cs);
}
```

（4**）设置像素格式与创建渲染场境。设置像素格式和创建渲染场境过程如下：**

使用 Class Wiszard**，在** CGLView **中增加** Windows **消息** WM_CREATE **的映射函数** CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)**，并在** OnCreate **中增加如下代码：**

```
int CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    // 视图窗口创建完成后，进行 OpenGL  Windows 的初始化设置
    GLInit();
return 0;
}
```

初始化设置在函数 GLInit()**中实现。这样，将这些代码从** Windows **的消息映射函数中分离出来，有利于软件的修改和维护。函数** GLInit()**的作用是在窗口创建完成之后，对其进行必要的初始化设置。**

函数 GLInit()**的实现代码如下：**

```
void CGLView::GLInit()
{
    m_hDC = ::GetDC(m_hWnd); // 获取设备场境句柄
    ASSERT(m_hDC);

    // 定义像素格式 Pixel Format
    static PIXELFORMATDESCRIPTOR pfdWnd =
    {
        sizeof(PIXELFORMATDESCRIPTOR),          // Structure size
        1,                                      // Structure version number
        PFD_DRAW_TO_WINDOW |                    // Property flags
        PFD_SUPPORT_OPENGL |
        PFD_DOUBLEBUFFER,
        PFD_TYPE_RGBA,
        24,                                     // 24-bit color
        0, 0, 0, 0, 0, 0,                       // Not concerned with these
        0, 0, 0, 0, 0, 0, 0,                    // No alpha or accum buffer
        32,                                     // 32-bit depth buffer
        0, 0,                                   // No stencil or aux buffer
```

```
        PFD_MAIN_PLANE,                     // Main layer type
        0,                                  // Reserved
        0, 0, 0                             // Unsupported
    };

    //在设备场境中选取和所定义最匹配的像素格式
    int pixelformat;
    pixelformat = ChoosePixelFormat(m_hDC, &pfdWnd);

    //为设备场境设置像素格式
     ASSERT(SetPixelFormat(m_hDC, pixelformat, &pfdWnd));

    //创建渲染场境
    m_hRC=wglCreateContext(m_hDC);

    //选择渲染场境 m_hRC 为当前场境
    VERIFY(wglMakeCurrent(m_hDC,m_hRC));

    //初始化渲染场境
    GLSetupRC(m_hDC);

    //关闭渲染场境
    VERIFY(wglMakeCurrent(NULL,NULL));
}
```

在函数 GLInit()**中，分别将获得的设备场境和渲染场境的句柄存储于两个变量** m_hDC **和** m_hRC **中。一旦创建了渲染场境，便将其设置为当前场境，并调用函数** GLSetupRC()**对渲染场境作必要的设置。设置完成后，再将当前场境设置为空。这样，只有在每次使用** OpenGL **进行绘制时才立即将渲染场境当前化，并在绘制任务结束后立即非当前化。在应用程序同时使用多个** OpenGL **显示窗口时，这样做是必要的，使应用程序能够使用多个渲染场境。**

另一种方法是：当应用程序只使用一个 OpenGL **显示窗口时，一旦渲染场境被创建，就一直保持其当前化，直到在函数** OnDestroy()**中被非当前化并被删除为止。这样整个程序只当前化一次渲染场境，有利于加快程序的运行。**

在函数 GLSetupRC()**中，主要工作是设置渲染场境的光照、颜色和物体的材质等。在下一章里，将对这些设置过程做具体的介绍。函数** GLSetupRC()**的实现代码如下：**

```
void CGLView::GLSetupRC()
{
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_COLOR_MATERIAL);

    //  光源设置
    GLfloat    ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat    diffuseLight[] = { 0.6f, 0.6f, 0.6f, 1.0f };
    GLfloat    specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat      lightPos[] = { -1000.0f, 1000.0f, 1000.0f, 1.0f };
```

```
    glEnable(GL_LIGHTING);
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glEnable(GL_LIGHT0);

    //材料设置
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glMaterialfv(GL_FRONT, GL_SPECULAR,specular);
    glMateriali(GL_FRONT,GL_SHININESS,100);

    //背景颜色设置（黑色）
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );
    //默认的前景颜色设置（蓝色）
    glColor3ub(0, 0, 255);
}
```

（5）清除渲染场境（Rendering Context）。在视图窗口释放时，不能忘记去释放创建的渲染场境。用Class Wizard在CGLView中添加窗口消息WM_DESTROY的映射函数OnDestroy()，并增加如下代码：

```
void CGLView::OnDestroy()
{
    //在释放窗口之前释放渲染场境
    GLRelease();
    CView::OnDestroy();
}
```

函数GLRelease()专门用于清除渲染场境，并释放设备场境，其实现代码如下：

```
void CGLView::GLRelease()
{
    wglDeleteContext(m_hRC);
    ::ReleaseDC(m_hWnd,m_hDC);
}
```

（6）处理视图窗口尺寸变化。当视图窗口的尺寸变化时，在OpenGL中应做相应的设置。在CGLView中添加WM_SIZE消息的映射函数OnSize()。并在OnSize()中添加如下代码：

```
void CGLView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);
    VERIFY(wglMakeCurrent(m_hDC,m_hRC));
    GLResize(cx,cy);
    VERIFY(wglMakeCurrent(NULL,NULL));
}
```

函数GLResize用于处理窗口尺寸的变化，其实现代码如下：

```
void CGLView::GLResize(int cx,int cy)
{
	double nRange = 1200.0;

	// 防止被 0 除
	if(cy == 0)
		cy = 1;

	//设置视口
	glViewport(0, 0, cx, cy);

	glMatrixMode(GL_PROJECTION);

	// 初始化矩阵
	glLoadIdentity();

	// 定义视景体(left, right, bottom, top, near, far)
	if(cx <= cy)
		glOrtho(-nRange,nRange,-nRange*cy/cx,nRange*cy/cx,1,2*nRange);
	else
		glOrtho(-nRange*cx/cy,nRange*cx/cy,-nRange,nRange,1,2*nRange);

	glMatrixMode( GL_MODELVIEW );
	glLoadIdentity( );

	//设置视点和方向
	double eye[3],ref[3],up_dir[3];
	eye[0] = 0; eye[1] = 0; eye[2] = nRange;
	ref[0] = 0;  ref[1] = 0; ref[2] = 0;
	up_dir[0]=0; up_dir[1]=1;up_dir[2]=0;
	gluLookAt(eye[0],eye[1],eye[2],ref[0],ref[1],ref[2],up_dir[0],up_dir[1],up_dir[2]);
}
```

（7）处理屏幕显示。完成以上的工作后，接下来将使用 OpenGL 进行图形绘制。需要对函数 OnDraw()做如下修改：

```
void CGLView::OnDraw(CDC* pDC)
{
	//当前化渲染场境
	wglMakeCurrent(m_hDC,m_hRC);

	//调用虚拟函数，具体绘制 OpenGL 图形
	RenderScene();

	//交换帧存
	SwapBuffers(m_hDC);
```

```
        //非当前化渲染场境
        wglMakeCurrent(m_hDC,NULL);
    }
```

RenderScene()是一个虚拟函数，被设计专门用来具体绘制 OpenGL 图形。要注意的是，在这里我们将 RenderScene()设为虚拟函数。在以后的应用中，只需在 CGLView 基础上派生一个视图类，即可用于 OpenGL 显示。然后所要做的只是：①重载虚拟函数 RenderScene()，并写入需要的显示命令；②注释掉派生的视图类中的虚拟函数 OnDraw()，这样程序会直接调用 CGLView 的 OnDraw()。

在 CView 中，窗口消息 WM_ERASEBACKGROUND 对应处理函数 OnEraseBkgnd()，用于窗口背景的擦除。当窗口尺寸变化或者调用函数 Invalidate()刷新时，MFC 会发出消息 WM_ERASEBACKGROUND 以擦除屏幕背景。用于在 OpenGL 中已设置了不同颜色的背景，这样会造成两次背景擦除，并有可能出现屏幕闪动的效果。所以在 CGLView 中，要屏蔽掉 MFC 的窗口擦除。

在 CGLView 中添加消息 WM_ERASEBACKGROUND 的处理函数 OnEraseBkgnd()，修改如下：

```
BOOL CGLView::OnEraseBkgnd(CDC* pDC)
{
    //return CView::OnEraseBkgnd(pDC);
    return FALSE;
}
```

（8）填写函数 RenderScene()。以上代码已完成了基本的 OpenGL Windows 程序的设置。接下来，用户可以轻松地在函数 RenderScene()中填写代码，用 OpenGL 绘制自己的图形。以下代码用于绘制三个不同颜色和尺寸的圆球。

```
void CGLView::RenderScene()
{
    //清除颜色缓存和深度缓存
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // 旋转
    glRotatef(3.0f,0.0f, 0.0f, 1.0f); // Rock Z
    glRotatef(3.0f,1.0f, 0.0f, 0.0f); // Roll X

    glPushMatrix();
    glColor3ub(255,0,0);//red
    auxSolidSphere(360); //0,0,0

    glColor3ub(0,0,255);//blue
    glTranslatef(600,0,0);//600,0,0
    auxSolidSphere(120);
    glColor3ub(0,255,0);//green
    glTranslatef(-1200,600,0);//-600,0,0
    auxSolidSphere(60);
```

```
        glPopMatrix();
        glFlush();
    }
```

这时用户编译并连接整个程序，运行程序会显示如图 3-8 所示的窗口。

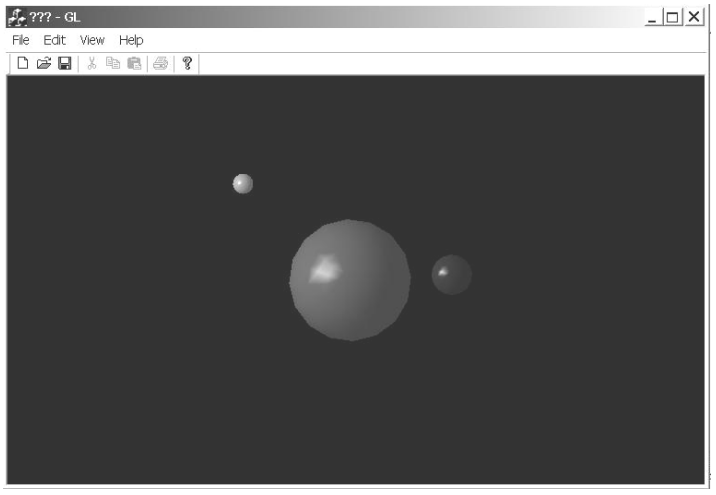


图 3-8　运行 GL.exe

我们看到的是红、黄、蓝的三个静止的球。

以上设计完成了一个封装了 OpenGL 基本功能的窗口类 CGLView。在具体应用时，还可在 CGLView 的基础上派生新的视图类，最简单的方法是仅仅覆盖虚拟函数 RenderScene()，在该函数中填入需要的显示命令。

3.3.3　使用 OpenGL 的双缓存技术为应用程序增加动画效果

OpenGL 支持双缓存（Double Buffer）技术。使用双缓存技术可以实现平滑的图形动画效果。在双缓存模式下，帧存被分为两个视频缓存：前台视频缓存和后台视频缓存。只有前台视频缓存的内容被显示，而绘制函数所绘制的场景首先被送往后台视频缓存。当绘制函数调用结束，并完成了后台视频缓存的内容时，OpenGL 便将它拷贝至前台视频缓存。由于这个视频交换的时间极短，肉眼感觉不出来，因此可以实现平滑的图形动画效果。

为使用双缓存，需要在像素格式定义时予以设置，即给属性标志变量增加 PFD_DOUBLE-BUFFER 属性。

```
static    PIXELFORMATDESCRIPTOR pfdWnd =
    {
        sizeof(PIXELFORMATDESCRIPTOR),
        1,
        PFD_DRAW_TO_WINDOW |                    // Property flags
        PFD_SUPPORT_OPENGL |
        PFD_DOUBLEBUFFER,
        PFD_TYPE_RGBA,
        24,
```

```
        0, 0, 0, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0,
        32,
        0, 0,
        PFD_MAIN_PLANE,
        0,0, 0, 0
    };
```

函数 SwapBuffers()**用于将后台视频缓存拷贝到前台。函数** OnDraw()**中，在调用函数** RenderScene()**完成** OpenGL **的场景绘制之后，需要调用** SwapBuffers()**以更新屏幕显示。**

```
void CGLView::OnDraw(CDC* pDC)
{
    wglMakeCurrent(m_hDC,m_hRC);
    RenderScene();
    SwapBuffers(m_hDC);
    wglMakeCurrent(m_hDC,NULL);
}
```

现在，可以修改上面的程序，以实现动画效果。只需要增加几行语句，就可以实现窗口中景物的旋转。步骤如下：

（1**）在函数** OnCreate()**中设置一个时钟。**

```
int CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CView::OnCreate(lpCreateStruct) == -1)
        return -1;

    GLInit();
    SetTimer(0,50,NULL); //每隔 50ms 发出消息 WM_TIMER
    return 0;
}
```

SetTimer()**函数为系统设置了一个序号为** 0 **的定时器，每隔** 50ms **发出消息** WM_TIMER**。**

（2**）在** CGLView **中添加消息** WM_TIMER **的映射函数** OnTimer()**。**

```
void CGLView::OnTimer(UINT nIDEvent)
{
    Invalidate(FALSE);      //刷新屏幕
    CView::OnTimer(nIDEvent);
}
```

这样，视图每接到一个消息 WM_TIMER**，窗口就刷新一次。**

（3**）对象的旋转。**

在 RenderScene()**中，已经包括了旋转的命令：**

```
// Rotate
glRotatef(3.0f,0.0f, 0.0f, 1.0f); //  绕 z 轴旋转 3°
glRotatef(3.0f,1.0f, 0.0f, 0.0f); //  绕 x 轴旋转 3°
```

每调用一次，图形分别绕 z 轴和 x 轴旋转 3°。每秒钟屏幕刷新 20 次，这样就能看到连续的旋转效果。

3.4 程序清单

这里给出类 CGLView 完整的源程序清单。

3.4.1 文件 GLView.h

```
////////////////////////////////////////////////////////////////////////////////
//      GLView.h : interface of the CGLView class
////////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_GLVIEW_H__B2996431_1A48_492F_87B8_D8E922845B95__INCLUDED_)
#define AFX_GLVIEW_H__B2996431_1A48_492F_87B8_D8E922845B95__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CGLView : public CView
{
protected: // create from serialization only
        CGLView();
        DECLARE_DYNCREATE(CGLView)

// Attributes
public:
        CGLDoc* GetDocument();

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CGLView)
        public:
        virtual void OnDraw(CDC* pDC);    // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual void RenderScene();
```

```
        virtual ~CGLView();

protected:

// Generated message map functions
protected:
        //{{AFX_MSG(CGLView)
        afx_msg int  OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
        afx_msg BOOL OnEraseBkgnd(CDC* pDC);
        afx_msg void OnSize(UINT nType, int cx, int cy);
        afx_msg void OnTimer(UINT nIDEvent);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
private:
        void        GLInit();
        void        GLRelease();
        void        GLResize(int cx,int cy);
        void        GLSetupRC();

        HGLRC           m_hRC;          //rendering context
        HDC             m_hDC;          //device context
};
#ifndef _DEBUG  // debug version in GLView.cpp
inline CGLDoc* CGLView::GetDocument()
   { return (CGLDoc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_GLVIEW_H__B2996431_1A48_492F_87B8_D8E922845B95__INCLUDED_)
```

3.4.2 文件 GLView.cpp

```
////////////////////////////////////////////////////////////////////////////////////////
// GLView.cpp : implementation of the CGLView class
////////////////////////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include "GL.h"

#include "GLDoc.h"
#include "GLView.h"
```

```
#include "gl\gl.h"
#include "gl\glu.h"
#include "gl\glaux.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CGLView

IMPLEMENT_DYNCREATE(CGLView, CView)

BEGIN_MESSAGE_MAP(CGLView, CView)
	//{{AFX_MSG_MAP(CGLView)
	ON_WM_CREATE()
	ON_WM_DESTROY()
	ON_WM_ERASEBKGND()
	ON_WM_SIZE()
	ON_WM_TIMER()
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CGLView construction/destruction

CGLView::CGLView()
{
	m_hDC = NULL;
	m_hRC = NULL;
}

CGLView::~CGLView()
{
}

BOOL CGLView::PreCreateWindow(CREATESTRUCT& cs)
{

	//    Add Window style required for OpenGL before window is created
	cs.style |= WS_CLIPSIBLINGS|WS_CLIPCHILDREN|CS_OWNDC;

	return CView::PreCreateWindow(cs);
}
```

```
/////////////////////////////////////////////////////////////////////////////
// CGLView drawing

void CGLView::OnDraw(CDC* pDC)
{
	wglMakeCurrent(m_hDC,m_hRC);

	RenderScene();

	SwapBuffers(m_hDC);

	wglMakeCurrent(m_hDC,NULL);
}

/////////////////////////////////////////////////////////////////////////////
// CGLView diagnostics

#ifdef _DEBUG

CGLDoc* CGLView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CGLDoc)));
	return (CGLDoc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CGLView message handlers

int CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CView::OnCreate(lpCreateStruct) == -1)
		return -1;

	// TODO: Add your specialized creation code here
	GLInit();
	SetTimer(0,50,NULL);
	return 0;
}

void CGLView::OnDestroy()
{
	GLRelease();
	CView::OnDestroy();
}

BOOL CGLView::OnEraseBkgnd(CDC* pDC)
```

```
{
	//return CView::OnEraseBkgnd(pDC);
	return FALSE;
}

void CGLView::OnSize(UINT nType, int cx, int cy)
{
	CView::OnSize(nType, cx, cy);

	VERIFY(wglMakeCurrent(m_hDC,m_hRC));
	GLResize(cx,cy);
	VERIFY(wglMakeCurrent(NULL,NULL));
}

void CGLView::GLInit()
{
	m_hDC = ::GetDC(m_hWnd); // Get the Device context

	ASSERT(m_hDC);

	static PIXELFORMATDESCRIPTOR pfdWnd =
	{
		sizeof(PIXELFORMATDESCRIPTOR), // Structure size
		1,                             // Structure version number
		PFD_DRAW_TO_WINDOW |           // Property flags
		PFD_SUPPORT_OPENGL |
		PFD_DOUBLEBUFFER,
		PFD_TYPE_RGBA,
		24,                            // 24-bit color
		0, 0, 0, 0, 0, 0,              // Not concerned with these
		0, 0, 0, 0, 0, 0, 0,           // No alpha or accum buffer
		32,                            // 32-bit depth buffer
		0, 0,                          // No stencil or aux buffer
		PFD_MAIN_PLANE,                // Main layer type
		0,                             // Reserved
		0, 0, 0                        // Unsupported
	};

	int pixelformat;
	pixelformat = ChoosePixelFormat(m_hDC, &pfdWnd);

	ASSERT(SetPixelFormat(m_hDC, pixelformat, &pfdWnd));

	m_hRC=wglCreateContext(m_hDC);

	//make the rending context current, perform initialization
	//deselect it
```

```
	VERIFY(wglMakeCurrent(m_hDC,m_hRC));
	GLSetupRC();
	VERIFY(wglMakeCurrent(NULL,NULL));
}

//clear up rendering context
void CGLView::GLRelease()
{
	wglDeleteContext(m_hRC);
	::ReleaseDC(m_hWnd,m_hDC);
}

void CGLView::GLResize(int cx,int cy)
{
	double nRange = 1200.0;

	// Prevent a divide by zero
	if(cy == 0)
		cy = 1;

	// Set Viewport to window dimensions
	glViewport(0, 0, cx, cy);
	glMatrixMode(GL_PROJECTION);

	// Reset coordinate system
	glLoadIdentity();

	// Establish clipping volume (left, right, bottom, top, near, far)
	if(cx <= cy)
		glOrtho(-nRange,nRange,-nRange*cy/cx,nRange*cy/cx,1,2*nRange);
	else
		glOrtho(-nRange*cx/cy,nRange*cx/cy,-nRange,nRange,1,2*nRange);


	glMatrixMode( GL_MODELVIEW );
	glLoadIdentity( );

	double eye[3],ref[3],up_dir[3];
	eye[0] = 0; eye[1] = 0; eye[2] = nRange;
	ref[0] = 0;  ref[1] = 0; ref[2] = 0;
	up_dir[0]=0; up_dir[1]=1;up_dir[2]=0;
	gluLookAt(eye[0],eye[1],eye[2],ref[0],ref[1],ref[2],up_dir[0],up_dir[1],up_dir[2]);
}

void CGLView::GLSetupRC()
{
	glEnable(GL_DEPTH_TEST);	// Hidden surface removal
```

```
	glEnable(GL_COLOR_MATERIAL);

	// Lighting components
	GLfloat	ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
	GLfloat	diffuseLight[] = { 0.6f, 0.6f, 0.6f, 1.0f };
	GLfloat	specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
	GLfloat		lightPos[] = { 1000.0f, 1000.0f, 1000.0f, 1.0f };

	glEnable(GL_LIGHTING);
	glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
	glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
	glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
	glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
	glEnable(GL_LIGHT0);

	glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
	glMaterialfv(GL_FRONT, GL_SPECULAR,specular);
	glMateriali(GL_FRONT,GL_SHININESS,100);

	glClearColor(0.0f, 0.0f, 0.0f, 1.0f ); //background color
	// default color
	glColor3ub(0, 0, 255);
}

void CGLView::RenderScene()
{
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

	// Rotate
	glRotatef(3.0f,0.0f, 0.0f, 1.0f); // Rock Z
	glRotatef(3.0f,1.0f, 0.0f, 0.0f); // Roll X

	glPushMatrix();
	glColor3ub(255,0,0);//red
	auxSolidSphere(360); //0,0,0

	glColor3ub(0,0,255);//blue
	glTranslatef(600,0,0);//600,0,0
	auxSolidSphere(120);

	glColor3ub(0,255,0);//green
	glTranslatef(-1200,600,0);//-600,0,0
	auxSolidSphere(60);
	glPopMatrix();

	glFlush();
```

```
}

void CGLView::OnTimer(UINT nIDEvent)
{
	Invalidate(FALSE);
	CView::OnTimer(nIDEvent);
}
```

本章相关程序

- ch3\GL：OpenGL Windows **应用程序** GL **的工程。**
- ch3\GL.exe：**执行程序。**

第 4 章　封装 OpenGL 功能的 C++类的设计

本章要点：

- **照相机取景与** OpenGL **的取景原理。**
- OpenGL **的视图变换和模型变换。**
- OpenGL **的投影变换。**
- OpenGL **的视口变换。**
- **封装** OpenGL **取景功能的照相机类** GCamera **的设计。**
- **封装** OpenGL **场景绘制类** COpenGLDC **的设计。**
- **支持** OpenGL **的通用视图类** CGLView **的设计。**

上一章讨论了怎样在 MFC **应用程序中使用** OpenGL **进行图形绘制。在创建的视图类** CGLView **中，具备了** OpenGL **绘制图形的基本环境设置。这一章将讨论使用面向对象的编程技术，根据** OpenGL **的功能设计一些** C++**类。在这些类中封装** OpenGL **的相关功能，并通过对这些类的操作，使得在** Windows **程序中操作** OpenGL **的相关功能更加方便。并在此基础上，开发一个类库** glContext.dll**，输出这些封装** OpenGL **的** C++**类，供其他应用程序调用。**

4.1　封装 OpenGL 的 C++类的设计

在上一章中，创建了在 CView **基础上派生的视图类** CGLView**。类** CGLView **中实现了在** Windows **环境下使用** OpenGL **绘制图形的基本设置，包括设置像素格式、创建和清除渲染场境、设置视点（**eye**）和投影方式/视景体（**viewing volume**）、光源和着色方式的设置、场景绘制等功能。实际上，在** CGLView **中只是完成了一些最基本的环境设置和场景绘制，并且所有相关代码都集中在类** CGLView **中实现。这样就存在一个问题，即随着程序功能的不断增加，** CGLView **中的代码会越来越长。在实际软件开发中，我们不应该也不可能将有关** OpenGL **的所有功能都写在** CGLView **类中，这样会使** CGLView **的实现文件很大，不利于修改和维护，同时也不利于软件的重复使用。开发人员更希望的是所设计的类的功能具有较强的功能性和相对的独立性。以** CGLView **为例，在这一章我们对它作进一步修改，将它所包括的功能划分成几个不同的** C++**类予以实现，如图** 4-1 **所示。**

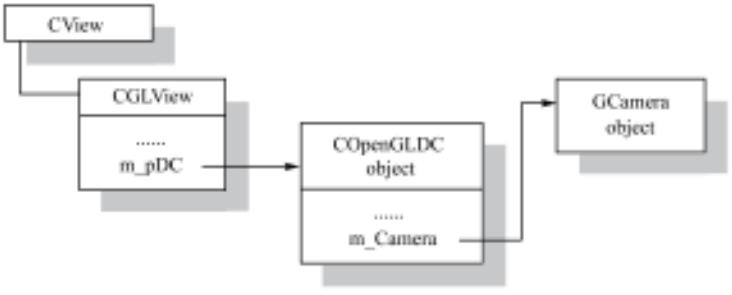


图 4-1　**类** CGLView **的结构**

GCamera 的功能类似于一个照相机，可称之为照相机类。GCamera 中定义了视口大小、投影变换和一个视点的位置与观察方向。这如同使用一架相机取景，使用者需要确定相机的位置和角度、取景范围，最后才将景物投影于胶片上的过程。这些操作实际上定义了 OpenGL 中一系列的变换。在三维空间中，变换是图形制作和显示环节中最关键的问题之一，物体摆放的位置、方向以及动画的实现都依靠变换来实现。通过变换，OpenGL 将三维对象投影到二维屏幕上。同时，OpenGL 中的变换还使得用户可以对图形进行平移、旋转和缩放。这些变换分别通过对视点变换、模型变换、投影变换和视口变换等操作来最终实现。GCamera 就是这样一个定义和操作变换的类。具体到 CAD 应用中，视图的放大、缩小、旋转、平移以及在三维空间中导航（Navigaition）的实现，都归结于对变换的操作。

COpenGLDC 作为一个封装 OpenGL 功能的 C++类，封装了在 MFC 下 OpenGL 的环境设置，即一个渲染场境（Rendering Context），以及图形绘制的相关函数。它的内容包括 OpenGL 和 Windows 窗口的关联、光照和颜色、取景操作、场景绘制等几方面。当然，可以根据不同的应用需要在类 COpenGLDC 中增加新的内容，这一点在本书第 5 章中还将具体论述。COpenGLDC 中包含了一个 GCamera 的对象，用于取景操作。对 OpenGL 的操作可在类 COpenGLDC 中实现。因而在 MFC 的窗口类插入一个 COpenGLDC 的对象，并使它与窗口关联，就可以使用 COpenGLDC 进行对 OpenGL 的操作和图形绘制。

和上一章中创建的 CGLView 不同的是，这里的 CGLView 把与 OpenGL 相关的代码分离了出来，在类中包括了一个 COpenGLDC 的对象，与 OpenGL 相关的操作通过对 COpenGLDC 的调用予以实现。作为一个视类，CGLView 本身的代码将集中于处理和分发用户和视图窗口的交互信息，如对视图的放大、缩小、视角的变换、旋转、平移以及鼠标的拖动和物体的拾取等操作。

这样划分的优点在于代码更具有独立性，能够提高代码的重复利用率，也便于对类进行维护和功能扩充。下面将具体介绍这几个类的设计与实现，通过这几个类的设计，也可以对 OpenGL 的一些相关概念有一个深入的了解。

4.2 照相机类 GCamera 的设计

在 OpenGL 程序中，屏幕所显示的景物范围是由 OpenGL 中的取景设置决定的。在绘制景物之前，必须首先进行取景设置。在第 3 章中创建的 CGLView 中，最终显示在窗口中的是三个不同颜色的圆球。我们看到的景物在窗口中所占的比例大小较为适中，之所以有这样的效果，是因为在函数 CGLView::GLSize()中进行了取景设置。OpenGL 中，取景操作是通过视点变换、投影变换来实现的，即定义出要观察的景物空间，并将三维景物投影成二维图像，最后再经过视口变换将图像输出于相应的窗口区域中。取景是 OpenGL 图形绘制和显示操作的一个关键技术，设计类 GCamera 的目的就是为了能够更方便地在应用程序中实现取景操作。下面需要介绍实现类 GCamera 的相关概念。

4.2.1 视点坐标系和视图变换

在 OpenGL 中，绘制图形的顶点（Vertex）使用的是世界坐标系（World Coordinate）。世界坐标系也被称为用户坐标系，它是一个右手三维直角坐标系，用于用户定义几何形体和图

素。如使用 OpenGL 的库函数 glVertex()来定义顶点，输入的坐标是相对于世界坐标系的，而与观察者的位置或观察方向无关。

视点坐标系（Eye Coordinate）是 OpenGL 中的一个重要概念。视点坐标系由视点的位置和观察的方向决定，正如我们从一架相机中观察到的空间，这个空间的坐标系是相对于相机的，即只由相机的位置和观察方向决定。如图 4-2 所示，视点坐标系是个右手坐标系，视线方向垂直于 xy 平面，+x 和+y 分别指向视点的右边和上边。+z 方向和视线方向相反，-z 方向表示从视点指向屏幕的方向。在 OpenGL 默认状态下，即没有任何视点变换的境况下，视点坐标系与世界坐标系一致。

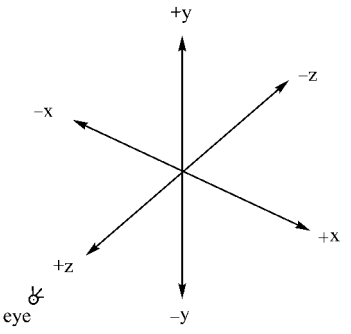


图 4-2　视点坐标系

视图变换的本质是改变视点坐标系与世界坐标系之间的位置关系。视图变换又可以根据实际操作中的不同概念而分为视点变换（Viewing Transformation）和模型变换（Modeling Transformation）。

默认状态下，视点的位置位于世界坐标系的原点，即（0,0,0），视线方向沿-z 轴方向。视点变换就是移动视点，相当于反向移动物体。可以通过移动视点的位置和改变视点的观察方向，以从不同位置、不同角度对世界坐标系中的场景进行观察。这个概念如同在空间移动一架相机进行取景。可以用下面所描述的方法来实现视点变换。

（1）使用一个或一组 OpenGL 变换命令，例如 glTranslate()、glRotate()。可以把这些变换的效果看成是反向移动相机。

（2）使用 OpenGL 实用库函数 gluLookAt()。正如该它的名字一样，该函数可以很直观且很方便地定义一个观察位置和观察方向。

函数 glLookAt()的原型定义如下：

```
void gluLookAt( GLdouble eyex, GLdouble eyey, GLdouble eyez,
                GLdouble centerx, GLdouble centery, GLdouble centerz,
                GLdouble upx, GLdouble upy, GLdouble upz );
```

该函数的参数包括三个部分，即：

（1）视点的位置 (eyex, eyey, eyez)。

（2）相机所瞄准的一个参考点的位置 (centerx, centery, centerz)。

（3）视线向上的方向 (upx, upy, upz)。

通常来说，参考点位于画面中间部分的某个位置。例如，如果所绘制的景物都在坐标系原点的附近，就可以将这个参考点取为坐标系的原点。视线向上的方向的定义非常重要，通常可以取 y 轴正方向作为向上的方向，这时情况是观察者正对着 xy 平面。更形象的使用场合是，如果我们在设计一个飞行模拟器，向上的方向就是与机翼垂直的方向，我们看到的景物随向上方向的变化而旋转。

函数 gluLookAt()是 OpenGL 实用库而非基本库的一部分，在它内部综合了一组平移（glTranslate()）和旋转（glRotate()）的变换命令，以实现所定义的视点变换。

模型变换的概念是视点不动而移动模型，如将一个物体移动到一个指定位置，并进行旋转和缩放。在图形绘制时，常使用这样的操作。例如，第 3 章中的函数 CGLView::RenderScene()

绘制三个圆球时，使用了函数 glTranslatef()分别将蓝色和绿色圆球的中心定于(600,0,200)、(-600,600,200)。

模型变换和视点变换的本质是一样的，即都是改变视点坐标系与世界坐标系之间的位置关系。它们都只对一个 OpenGL 变换矩阵 MODELVIEW 进行操作。对它们进行区分的目的在于便于理解和方便程序编写。对绘制效果来说，将一个物体向后移动和将一个视点向前移动并没有本质区别，绘制的效果是完全一样的。使用 MODELVIEW 来命名这个变换矩阵也是说明这个矩阵既可用来进行模型（Model）变换，又可用于视点（View）变换。其实，所有的视图变换的作用就是在于改变视点坐标系和世界坐标系之间的位置关系，以达到需要的观察效果。

4.2.2 投影变换与视景体

投影变换（Projection Transformation）定义的是一个视景体（Viewing Volume），即所要观察的物体空间。显示在屏幕上的内容就是视景体中的内容。只有把三维物体绘制在视景体中，才能显示在屏幕上。在 OpenGL 中，投影变换分为两种：正交投影和透视投影。它们的区别也就在于定义的视景体的形状不一样。

正交投影（Orthographic Projections）定义的是一个长方形的视景体，如图 4-3 所示。物体在屏幕上的显示尺寸不受所处距离远近的影响，这也称作平行投影或直角投影。大多数 CAD 软件都采用正交投影，以使得几何体距离的远近不影响其显示的大小。

OpenGL 使用函数 glOrtho()来定义正交投影，该函数原型如下：

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

该函数定义了如图 4-3 所示的长方形视景体。其中，left、right 为垂直剪切面的坐标；bottom、top 为水平剪切面的坐标；near、far 分别为近视点剪切面和远视点剪切面。通常近视点剪切面和远视点剪切面都位于视线的前方，即值都为正，但也允许分别位于视点的前后，在第 3 章 CGLView::GLResize()中定义正交投影时，使用的就是近视点剪切面位于视点后方的情况。

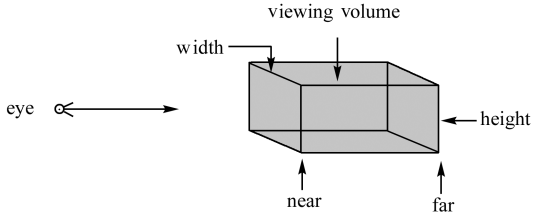


图 4-3 正交投影变换

透视投影(Perspective Projections)定义了一个金字塔形的视景体，也可称之为视锥体，如图 4-4 所示。它符合人们日常观察物体的视觉习惯，靠近视点的物体显得大一些，远离视点的物体显得小一些。这种投影方式多用于游戏、动画和虚拟现实（Virtual Reality）软件中，以产生更强的真实感效果。OpenGL 中提供了两个函数可以定义透视投影。

```
void glFrustum(GLdouble left, GLdouble right, GLdouble bottom,
```

GLdouble top, GLdouble near, GLdouble far);

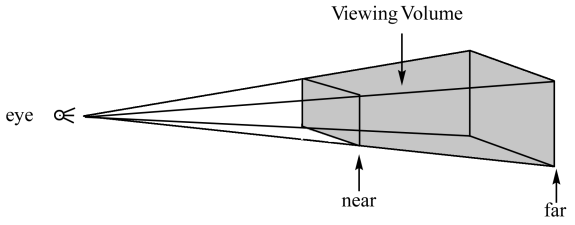


图 4-4　透视投影变换

其中，left、right、bottom、top **的含义与** glOrtho()**中的参数含义相同。**near、far **分别表示近剪切面和远剪切面，和正交投影不同的是，**near **和** far **的值都必须为正。**

但是，使用 glFrustum()**定义透视投影的几何意义不够直观。**OpneGL **提供的另外一个定义透视投影的函数是** gluPerspective()**。它更容易使用，几何意义也更加直观。**

```
void gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar );
```

其中，fovy **定义了垂直方向上的视角；**aspect **定义了视景体高度与宽度的比率；**zNear、zFar **分别是近剪切面和远剪切面。例如，下面代码定义了一个视角为** 45° **，深度从** 1 **到** 1000 **的正方视锥体。**

```
glMatrixMode(GL_PROJECTION);        //切换当前矩阵为投影矩阵
glLoadIdentity();                   //矩阵作初始化
gluPerspective(45.0, 1.0, 1.0, 1000.0);  //定义透视投影
```

4.2.3　视口变换

视口（Viewport**）是指** Windows **窗口的客户区（**Client Area**）中绘制** OpenGL **图像的区域。要把物体最终显示在窗口中，还需要进行视口变换。视口变换（**Viewport Transformation**）就是指把视景体的投影面映射到视口的过程。视口变换是** OpenGL **变换中的最后一个环节，它决定** OpenGL **绘制的景物将输出在窗口中的哪一部分。通常我们会将整个** Windows **窗口的客户区定义为视口，但用户也可以只定义窗口的一部分作为视口。**

OpenGL **中，定义视口的函数如下：**

```
void glViewport(GLint x, GLint y, GLint width, GLint height);
```

其中，x **和** y **分别是视口左下角在窗口中的位置；**width **和** height **分别是视口的宽度和高度。视口的大小以像素为单位。在默认情况下，**OpenGL **将视口设为窗口的整个客户区。**

当 Windows **窗口大小改变时，**Windows **会在发出消息** WM_PAINT**（通知窗口进行重新绘制）之前先发出消息** WM_SIZE**（通知窗口的尺寸已经发生了变化）。所以，消息** WM_SIZE **的处理函数是设置视口变换并定义投影变换的理想位置。因而，在** CGLView∷OnSize()**中需要调用函数** GLResize()**重新设置视口，并定义投影变换。**

在第 3 **章创建类** CGLView **时，关于视口变换和投影变换的操作是在函数** CGLView::GLResize()**中进行的，当时没有对相关代码作解释。下面我们来分析这段代码。**

```
void CGLView::GLResize(int cx,int cy)
{
    double nRange = 1200.0;

    if(cy == 0)            //防止被 0 除
        cy = 1;

    gViewport(0, 0, cx, cy);                    //设置视口
    glMatrixMode(GL_PROJECTION);        //切换当前矩阵为投影矩阵
    glLoadIdentity();                           //单位化矩阵

    //设置视景体大小
    if(cx <= cy)
        glOrtho(-nRange,nRange,-nRange*cy/cx,nRange*cy/cx, nRange,-nRange);
    else
        glOrtho(-nRange*cx/cy,nRange*cx/cy,-nRange,nRange, nRange,-nRange);

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
}
```

GLResize()**中代码的主要功能在于分别设置了视口（**Viewport**）、视景体（**Viewing Volume**）。函数** gViewport()**用于设置视口；函数** glOrtho()**用于设置正交投影，即一个长方体形状的视景体。这些设置决定了取景的范围、取景的方向和景物在** Windows **窗口中所对应的显示区域（视口）。假设窗口尺寸为** 640×320 **像素，即** cx=640、cy=320**，则在函数** GLResize() **中对视口和视景体设置分别如下：**

视口设置：

```
gViewport(0, 0, 640, 320);
```

这个视口设置函数将当前整个窗口的客户区设置为视口。

视景体设置：

```
glOrtho(-2400,2400,-1200,1200,-1200,1200);
```

这个正交投影函数创建了一个长、宽、高分别为 2400、4800、2400 **的一个视景体，如图** 4-5 **所示。**

函数 GLResize()**中还没有设置视点坐标系，即进行视点变换，这时视点坐标系处于默认的状态下，即仍然处于世界坐标系的原点，坐标轴的方向也与世界坐标系一致。**glOrtho()**定义的视景体是相对于视点坐标系的，所以这时视景体空间的中心位于世界坐标系的原点**(0,0,0)**。**

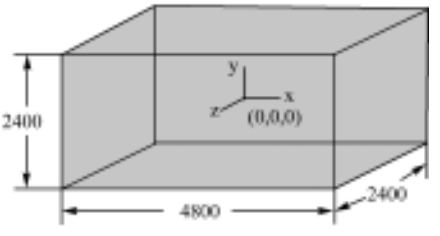


图 4-5　GLResize()**中创建的视景体**

在图形绘制函数 CGLView::RenderScene()**中，通过模型变换（使用函数** glTranslate()**）绘制了三个中心分别位于**(0,0,0)**、**(600,0,200)**、**(-600,600,200)**的圆球。这三个圆球的位置都落在所定义的视景体**

(如图 4-5 所示)中，这就是在屏幕上能够观察到这三个圆球的原因。如果它们的位置落在这个视景体之外，则我们将无法观察到这些物体。

函数 CGLView::RenderScene()的代码：

```
void CGLView::RenderScene()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Rotate
    glRotatef(3.0f,0.0f, 0.0f, 1.0f); // Roll Z
    glRotatef(3.0f,1.0f, 0.0f, 0.0f); // Roll X

    glPushMatrix();
    //绘制一个半径 360，中心位于(0,0,0)的球，颜色为红色
    glColor3ub(255,0,0);   //red
    auxSolidSphere(360);  //0,0,0

    //绘制一个半径 120，中心位于(600,0,200)的球，颜色为蓝色
    glColor3ub(0,0,255);          //blue
    glTranslatef(600,0,200);
    auxSolidSphere(120);          //600,0,200

    //绘制一个半径 60，中心位于(-600,600,200)的球，颜色为绿色
    glColor3ub(0,255,0);          //green
    glTranslatef(-1200,600,0);
    auxSolidSphere(60);
    glPopMatrix();

    glFlush();
}
```

4.2.4 设计照相机类 GCamera

以上分析了视图变换、投影变换和视口变换在 OpenGL 三维图形绘制中的作用，这些操作类似于一个照相机的取景过程，如果将这些功能集中在一个类 GCamera 中予以实现，将会使操作更加方便。就像我们使用相机取景时需要改变位置和角度、伸缩镜头以改变观察空间一样，在 GCamera 中也需要增加相应的操作函数来操作取景。

类 GCamera 定义如下：

```
class GCamera    {
protected:
    //视点位置和观察方向
    double          m_eye[3];
    double          m_ref[3];
    double          m_vecUp[3];
```

```
    //视景体参数
    double          m_far, m_near;
    double          m_width,m_height;

    //视口参数
    double          m_screen[2];

public:
    //构造与析构函数
    GCamera();
    ~GCamera();

    //初始化函数
    void init();

    //取景函数
    void projection();

    //设置视口尺寸
    void set_screen( int x, int y);

    //设置视点位置和方向
    void set_eye(double eye_x,double eye_y,double eye_z);
    void set_ref(double ref_x,double ref_y,double ref_z);
    void set_vecUp(double up_dx,double up_dy,double up_dz);

    //设置视景体
    void set_view_rect(double width,double height);
    void get_view_rect(double& width,double& height);
};
```

类 GCamera **中包括以下三部分数据：**

（1**）视点位置和视线方向。点** m_eye **表示视点所在的位置；点** m_ref **作为视线前方的一个参照点，它的位置对应于显示窗口的中心；矢量** m_vecUp **表示视线正上方的矢量方向。这三个数据定义了视点的位置和观察的方向。**

（2**）视景体定义。因为要观察的是** CAD **图形，所以这里使用正交投影。变量** m_far、m_near、m_width、m_height **定义了一个长方体，作为正交投影的视景体。这个视景体沿视线两侧对称。**

（3**）视口尺寸。参数** m_screen **定义的是视口的大小，通常情况下是用整个窗口作为视口，这时** m_screen **应定义为窗口客户区的像素尺寸。**

对于类 GCamera **的具体的实现函数，分析如下：**

（1**）用于初始化的成员函数** init()**。在使用** GCamera **的功能函数之前，必须首先对其初始化。在初始化函数** init()**中，分别为视点和观察方向、视景体定义、视口尺寸等赋初始值。**

```
void GCamera::init()
```

```
{
    m_eye[0] = 0;                    //视点位置(0,0,1000)
    m_eye[1] = 0;
    m_eye[2] = 1000.0;

    m_ref[0] = 0.0;                  //参考点位置(0,0,0)
    m_ref[1] = 0.0;
    m_ref[2] = 0.0;

    m_far = 10000;
    m_near= 1;

    m_width = 2400.0;
    m_height = 2400.0;

    m_vecUp[0] = 0.0;                //视线向上的方向(0,1,0)
    m_vecUp[1] = 1.0;
    m_vecUp[2] = 0.0;

    m_screen[0] = 400;
    m_screen[1] = 400;
}
```

（2）用于视口设置的成员函数 set_screen()**。当** Windows **窗口的尺寸变化时，需要对视口的尺寸进行重新设置。同时，为了使图形的显示比例不随窗口尺寸的变化而变化，还需要根据新的窗口尺寸重新计算视景体的宽和高。函数** set_screen()**在** Windows **窗口尺寸变化时被调用，用于重新设置视口和视景体。**

```
void GCamera::set_screen( int x, int y)
{
    glViewport(0,0,x,y);    //设置视口
    if(y==0) y=1;
    double ratio = (double)x/(double)y;
    m_width *= (double)x/m_screen[0];      //根据窗口尺寸变化更新视景体的宽与高
    m_height *= (double)y/m_screen[1];
    m_width =   m_height*ratio;   //保持视景体宽与高的比率与窗口一致
    m_screen[0] = x;
    m_screen[1] = y;
}
```

（3）用于投影变换的成员函数 projection()**。**projection()**是类中最关键的函数，它根据类中的数据来创建投影变换矩阵，并通过** OpenGL **实用库函数** gluLookAt()**来定义视点变换。它的实现如下：**

```
void GCamera::projection()
{
    //切换到投影变换矩阵设置
    glMatrixMode(GL_PROJECTION);
```

```
//初始化投影矩阵
glLoadIdentity();

//渲染模式
glRenderMode(GL_RENDER);

//创建投影矩阵
double left     =  - m_width/2.0;
double right    =  m_width/2.0;
double bottom   =  - m_height/2.0;
double top      =  m_height/2.0;

//直角投影方式
glOrtho(left,right,bottom,top,m_near,m_far);

//切换到视图变换矩阵设置
glMatrixMode( GL_MODELVIEW );

//初始化视图变换矩阵
glLoadIdentity( );

//设置视点位置和观察方向
gluLookAt(m_eye[0],m_eye[1],m_eye[2],m_ref[0],m_ref[1],m_ref[2],
m_vecUp[0], m_vecUp[1], m_vecUp[2]);
}
```

（4）用于设置视点和视景体的成员函数。GCamera **中还分别定义了设置视点和视景体的函数。**

设置视点位置和方向：

```
void set_eye(double eye_x,double eye_y,double eye_z);
void set_ref(double ref_x,double ref_y,double ref_z);
void set_vecUp(double up_dx,double up_dy,double up_dz);
```

设置视景体：

```
void set_view_rect(double width,double height);
void get_view_rect(double& width,double& height);
```

在本章 4.6 **节给出了源程序清单，这里不再具体分析。**

以上创建了一个简单的照相机类 GCamera**，它具备了** OpenGL **取景操作的基本功能。在** COpenGLDC **中插入一个** GCamera **的对象，用来实现** OpenGL **的取景操作。对于** GCamera **的使用，将在下面结合类** COpenGLDC **的创建予以介绍。**

4.3 类 COpenGLDC

创建类 COpenGLDC **的目的在于，就如同我们在** CView **类中调用** CDC **类绘制图形一样，**

调用 COpenGLDC 可直接进行 OpenGL 图形绘制。在第 3 章中，我们创建了一个支持 OpenGL 的视类 CGLView。定义了类 COpenGLDC 以后，只需要在 CGLView 中创建一个 COpenGLDC 的对象，从而将操作 OpenGL 的代码从 CGLView 类本身的实现函数中剥离出来。这样，有益于实现类的功能的独立性，也方便类的维护和功能扩充。我们不应该也不可能将有关 OpenGL 的所有功能都写在 CGLView 类中，这样会使 CGLView 的实现文件很大，不利于维护和修改，同时也不利于软件的重复使用。类 COpenGLDC 还可以用于其他形式的窗口类中，以实现 OpenGL 与这些窗口的关联。作为一个视类，CGLView 本身的代码将集中于处理和分发用户和视类本身的交互信息，如对视图的放大、缩小、视角的变换、旋转、平移以及鼠标的拖动和物体的拾取等等。

COpenGLDC 中封装了在 MFC 环境下设置 OpenGL 环境和调用 OpenGL 绘制图形的功能。如同使用类 CDC 进行 GDI 图形绘制一样，一个功能完善的 OpenGL 封装类将会大大方便在 Windows 窗口下的三维图形绘制。我们希望创建的类 COpenGLDC 将包括以下几个主要方面的功能：

- 实现和 Windows 窗口的关联。
- 取景操作。通过 GCamera 的对象实现。
- 绘图操作。
- 光源。
- 颜色。
- 选择操作。

这里，我们希望首先实现一个简单的 COpenGLDC 的框架，看它如何对 OpenGL 代码进行封装，并实现 Windows 窗口与 OpenGL 的关联。在下一章，还将进一步为它实现更多的功能。

对 OpenGL 的设置和操作都在类 COpenGLDC 的代码中实现。因而在 MFC 的窗口类插入一个 COpenGLDC 的对象，就可轻松地使用 OpenGL 的相关功能。

类 COpenGLDC 定义如下：

```
class COpenGLDC
{
public:
    //构造与析构函数
    COpenGLDC(CWnd* pWnd);
    virtual ~COpenGLDC();

private:
    //关联窗口的句柄
    HWnd    m_hWnd;


    //渲染场境句柄
    HGLRC   m_hRC;

    //设备场境句柄
```

```
    HDC         m_hDC;

public:
    //照相机，用于取景操作
    GCamera          m_Camera;

public:
    //初始化
    BOOL InitDC();

    //对应窗口尺寸变化
    void GLResize(int cx,int cy);

    //设置渲染场境
    void GLSetupRC();

    //绘图前准备函数
    void Ready();

    //结束绘图函数
    void Finish();
};
```

对于类 COpenGLDC **的主要函数，分析如下：**

（1**）用于初始化的成员函数** InitDC()**。在使用** COpenGLDC **之前，必须调用** InitDC()**对其初始化，包括像素格式设置、创建和设置渲染场境。由于需要在窗口创建完成之后才能创建渲染场景，所以通常在视图窗口的** OnCreate()**函数中调用** InitDC**。**

```
BOOL COpenGLDC::InitDC()
{
    if (m_pWnd == NULL) return FALSE;

    m_Camera.init();        //初始化照相机设置

    m_hDC = ::GetDC(m_pWnd->GetSafeHwnd());        //获取设备场境

    //设置像素格式
    int pixelformat;
    if ( (pixelformat = ChoosePixelFormat(m_hDC, &pfdWnd)) == 0 )
    {
        AfxMessageBox("ChoosePixelFormat to wnd failed");
        return FALSE;
    }
    if (SetPixelFormat(m_hDC, pixelformat, &pfdWnd) == FALSE)
            AfxMessageBox("SetPixelFormat failed");

    //创建渲染场境
```

```
    m_hRC=wglCreateContext(m_hDC);

    VERIFY(wglMakeCurrent(m_hDC,m_hRC));    //当前化渲染场境
    GLSetupRC();            //初始化渲染场境
    wglMakeCurrent(NULL,NULL);   //非当前化渲染场境
    return m_hRC!=0;
}
```

（2）用于窗口尺寸设置的成员函数 GLResize()**。函数** GLResize()**对应于** Windows **窗口的尺寸变化。当窗口的尺寸变化后，调用** GLResize()**重新设置视口和视景体。通常在视图窗口的** OnSize()**函数中调用** GLResize()**。**

```
void COpenGLDC::GLResize(int w,int h)
{
    wglMakeCurrent(m_hDC,m_hRC);

    if(h == 0) h = 1;  //防止被 0 除
    if(w == 0) w = 1;
    m_Camera.set_screen(w,h);
    wglMakeCurrent(NULL,NULL);
}
```

（3）用于渲染场境设置的成员函数 GLSetupRC()**。函数** GLSetupRC **用于设置渲染场境。包括设置光源、材料属性、当前色和背景色等。**

```
void COpenGLDC::GLSetupRC()
{
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_COLOR_MATERIAL);

    // 设置光源
    GLfloat     ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat     diffuseLight[] = { 0.6f, 0.6f, 0.6f, 1.0f };
    GLfloat     specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat     lightPos[] = { 1000.0f, 1000.0f, 1000.0f, 1.0f };

    glEnable(GL_LIGHTING);
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
    glEnable(GL_LIGHT0);

    //设置材料性质
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
    glMaterialfv(GL_FRONT, GL_SPECULAR,specular);
    glMateriali(GL_FRONT,GL_SHININESS,100);
```

```
    //设置背景色（黑色）
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f );

    //设置材料的默认颜色
    glColor3ub(0, 0, 255);
}
```

（4）用于绘图前准备的成员函数 Ready()。在每次绘制一帧图形之前，调用函数 Ready()以做开始图形绘制之前的准备工作。函数中，调用了 GCamera::projection()在每次绘图之前调整取景，这使得每次图形绘制都与取景设置相匹配。在程序的其他地方改变了取景设置后，只要重新刷新图形，所看到的景物范围就会及时改变。同时，如果使用 OpenGL 的拾取操作时，只需要将这个 projection()切换成一个选择投影变换函数 selection()即可。在本书后面讲述 OpenGL 拾取操作时会进一步对此讲述。

在每次绘图之前，还需要使用背景色清除原先的图形，OpenGL 的库函数 glClear()执行这个清除任务。glClear()使用函数 glClearColor()所定义的背景颜色。在函数 GLSetupRC()中，定义了背景色为黑色。

函数 COpenGLDC::Ready()实现如下：

```
void COpenGLDC::Ready()
{
    //当前化渲染场境
    wglMakeCurrent(m_hDC,m_hRC);

    //绘制图形之前的取景
    m_Camera.projection();

    //清除颜色缓存与深度缓存
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

（5）用于结束绘图的成员函数 Finish()。在一帧图形绘制结束之后，调用函数 Finish()以结束绘制并显示图形。当 OpenGL 发出一系列绘制命令之后，由于采用了双缓存技术，生成的图像被先送到后台视频缓存区中。当完整的图像在后台视频缓存中绘制完成以后，就调用 SwapBuffers()函数，将它设为前台视频缓存，以在屏幕上输出。

在交换视频缓存之前，还必须进行同步操作。在内部，OpenGL 使用一个队列来排列所要执行的命令，并将它们累积起来批处理。这样的处理方式有利于提高它的性能，尤其是绘制一个复杂对象的情况。函数 glFlush()告诉 OpenGL 必须立即处理队列中的命令（而不是继续等待更多的绘图命令），并在有限的时间内完成。在执行 SwapBuffers()之前，必须保证先前发出的绘制命令都已经完成，这时可调用 glFlush()以确保图形绘制已完成。

```
void COpenGLDC::Finish()
{
    glFlush();
```

```
        //交换帧存
        SwapBuffers(m_hDC);

        //非当前化渲染场境
        wglMakeCurrent(NULL,NULL);
}
```

4.4 修改类 CGLView

在创建了 GCamera **和** COpenGLDC **之后，可将大量与** OpenGL **相关的代码从** CGLView **中分离出来。而** CGLView **作为一个视类，本身的代码将集中于处理与视图有关的操作。**

修改后的类 CGLView **定义如下：**

```
class CGLView : public CView
{
protected:
        //插入一个 COpenGLDC 的指针
        COpenGLDC*    m_pGLDC;

protected:
        CGLView();
        DECLARE_DYNCREATE(CGLView)

// Attributes
public:
        //场景绘制函数
        virtual void RenderScene(COpenGLDC* pDC);

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CGLView)
        public:
        virtual void OnDraw(CDC* pDC);    // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL
// Implementation
public:
        virtual ~CGLView();

protected:
```

```
// Generated message map functions
protected:
    //{{AFX_MSG(CGLView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnDestroy();
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    afx_msg void OnSize(UINT nType, int cx, int cy);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

CGLView **中对** COpenGLDC **对象的调用如图** 4-6 **所示。**

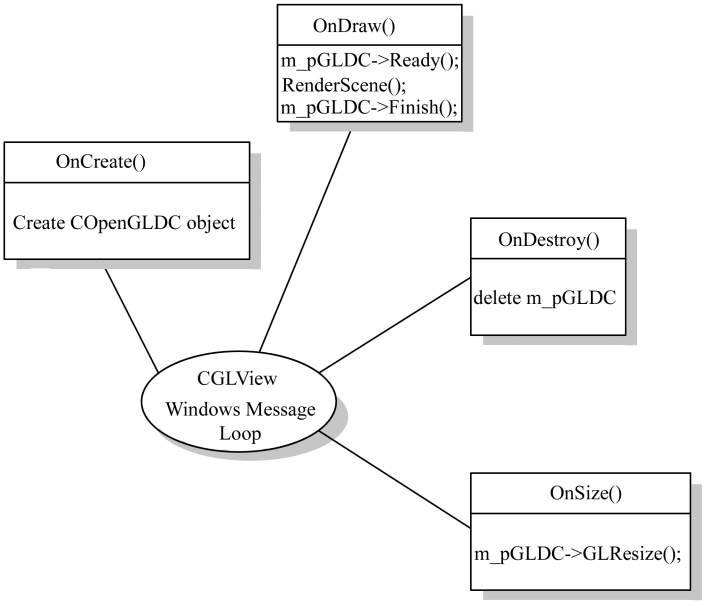


图 4-6　CGLView **的消息处理**

对于类 CGLView **的主要函数，修改如下：**

（1**）修改函数** OnDraw()

```
void CGLView::OnDraw(CDC* pDC)
{
    if(m_pGLDC){
        m_pGLDC->Ready();
        RenderScene(m_pGLDC);
        m_pGLDC->Finish();
    }
}
```

注意，在 CGLView **基础上派生的类中，必须删除掉自己的** OnDraw()**函数。**

由于 CGLView::RenderScene()**是一个虚拟函数，这样，在** CGLView **的派生类中只需要定**

义一个自己的 RenderScene()函数，就可直接调用 OpenGL 图形绘制命令进行图形绘制。

（2）修改函数 OnCreate()

COpenGLDC 的对象必须等到 CGLView 窗口创建完成之后才能创建。因而，消息 WM_CREATE 的映射函数 OnCreate()是创建并初始化这个对象的理想位置。

```
int CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CView::OnCreate(lpCreateStruct) == -1)
		return -1;

	//创建 COpenGLDC 的对象
	m_pGLDC = new COpenGLDC(this);

	//初始化 COpenGLDC 的对象
	m_pGLDC->InitDC();

	return 0;
}
```

（3）修改函数 OnDestroy()

在窗口释放时，必须释放创建的 COpenGLDC 对象。

```
void CGLView::OnDestroy()
{
	CView::OnDestroy();

	//释放 COpenGLDC 的对象
	if(m_pGLDC)	delete m_pGLDC;
}
```

（4）修改函数 OnSize()

```
void CGLView::OnSize(UINT nType, int cx, int cy)
{
	CView::OnSize(nType, cx, cy);

	if(m_pGLDC)
		m_pGLDC->GLResize(cx,cy);
}
```

（5）修改消息映射函数 OnEraseBkgnd()

```
BOOL CGLView::OnEraseBkgnd(CDC* pDC)
{
	//return CView::OnEraseBkgnd(pDC);
	return FALSE;
}
```

（6）用 RenderScene()绘制图形

在函数 CGLView::RenderScene()**中，我们将绘制三个彩色圆球作为** CGLView **绘图函数的默认绘制。也就是说，当用户在** CGLView **基础上派生一个新类时，如果不重新定义自己的** RenderScene()**函数，则默认状态下屏幕绘制的是三个彩色的圆球。**

```
void CGLView::RenderScene(COpenGLDC* pDC)
{
    // Rotate
    glRotatef(3.0f,0.0f, 0.0f, 1.0f);      // Roll Z
    glRotatef(3.0f,1.0f, 0.0f, 0.0f);      // Roll X

    glPushMatrix();
    glColor3ub(255,0,0);                   //绘制红色球
    auxSolidSphere(360);                   //球心位于(0,0,0)

    glColor3ub(0,0,255);                   //绘制蓝色球
    glTranslatef(600,0,200);               //球心位于(600,0,0)
    auxSolidSphere(120);

    glColor3ub(0,255,0);                   //绘制绿色球
    glTranslatef(-1200,600,0);             //球心位于(-600,600,200)
    auxSolidSphere(60);

    glPopMatrix();
}
```

4.5 运行应用程序

完成以上工作后，可编译连接工程。运行结果与第 3 **章中的程序一样，如图** 3-8 **所示，显示的是三个不同颜色的圆球。**

4.6 源程序清单

下面给出 GCamera、COpenGLDC **和** CGLView **三个类的完整的程序清单，供参考。**

4.6.1 类 GCamera 的声明代码

```
#ifndef _CAMERA_H__
#define _CAMERA_H__

class GCamera    {
protected:
    //eye position
    double          m_eye[3];
    double          m_ref[3];
    double          m_vecUp[3];
```

```
        //viewing volume
        double          m_far, m_near;
        double          m_width,m_height;

        //viewport
        double          m_screen[2];

public:
        GCamera();
        ~GCamera();

        void init();
        void projection();

        //set viewport acoording to window
        void set_screen( int x, int y);

        //set eye coordinate
        void set_eye(double eye_x,double eye_y,double eye_z);
        void set_ref(double ref_x,double ref_y,double ref_z);
        void set_vecUp(double up_dx,double up_dy,double up_dz);

        //set viewing volume
        void set_view_rect(double width,double height);
        void get_view_rect(double& width,double& height);
};
#endif
```

4.6.2 类 GCamera 的实现代码

```
////////////////////////////////////////////////////// Camera.cpp//////////////////////////////////////////////////////
#include "stdafx.h"
#include <gl/gl.h>
#include <gl/glu.h>

#include "camera.h"

GCamera::GCamera(void)
{
}

GCamera::~GCamera()
{
}

void GCamera::projection()
{
```

```
        //switch to projection
        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glRenderMode(GL_RENDER);

        //apply projective matrix
        double left     =   - m_width/2.0;
        double right    =   m_width/2.0;
        double bottom   =   - m_height/2.0;
        double top      =   m_height/2.0;

        glOrtho(left,right,bottom,top,m_near,m_far);

        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity( );
        gluLookAt(m_eye[0],m_eye[1],m_eye[2],m_ref[0],m_ref[1],m_ref[2],m_vecUp[0],   m_vecUp[1],
m_vecUp[2]);
  }

  void GCamera::init()
  {
        m_eye[0] = 0;
        m_eye[1] = 0;
        m_eye[2] = 1000.0;

        m_ref[0] = 0.0;
        m_ref[1] = 0.0;
        m_ref[2] = 0.0;

        m_far = 10000;
        m_near= 1;

        m_width = 2400.0;
        m_height = 2400.0;

        m_vecUp[0] = 0.0;
        m_vecUp[1] = 1.0;
        m_vecUp[2] = 0.0;

        m_screen[0] = 400;
        m_screen[1] = 400;
  }

  void GCamera::set_screen( int x, int y)
  {
        glViewport(0,0,x,y);
```

```
        if(y==0) y=1;
        double ratio = (double)x/(double)y;
        m_width *= (double)x/m_screen[0];
        m_height *= (double)y/m_screen[1];
        m_width =    m_height*ratio;
        m_screen[0] = x;
        m_screen[1] = y;
}

void GCamera::set_eye(double eye_x,double eye_y,double eye_z)
{
        m_eye[0] = eye_x;
        m_eye[1] = eye_y;
        m_eye[2] = eye_z;
}

void GCamera::set_ref(double ref_x,double ref_y,double ref_z)
{
        m_ref[0] = ref_x;
        m_ref[1] = ref_y;
        m_ref[2] = ref_z;
}

void GCamera::set_vecUp(double up_dx,double up_dy,double up_dz)
{
        m_vecUp[0] = up_dx;
        m_vecUp[1] = up_dy;
        m_vecUp[2] = up_dz;
}

void GCamera::set_view_rect(double width,double height)
{
        m_width = width;
        m_height = height;
        double aspect = m_screen[0]/m_screen[1];
        m_width =    m_height*aspect;
}

void GCamera::get_view_rect(double& width,double& height)
{
        width = m_width;
        height = m_height;
}
```

4.6.3 类 COpenGLDC 的声明代码

```
#ifndef _OPENGLDC_H
```

```
#define _OPNEGLDC_H

#include "gl/gl.h"
#include "gl/glu.h"
#include "gl/glaux.h"

#include "camera.h"

class COpenGLDC : public CObject
{
public:
	COpenGLDC(HWND hWnd);
	virtual ~COpenGLDC();

private:
	HWND	m_hWnd;
	HGLRC	m_hRC;
	HDC		m_hDC;

public:
	GCamera		m_Camera;

public:
	//initialize
	BOOL InitDC();
	void GLResize(int cx,int cy);
	void GLSetupRC();

	void Ready();
	void Finish();
};
#endif
```

4.6.4 类 COpenGLDC 的实现代码

```
////////////////////////////////////////////////// OpenGLDC.cpp//////////////////////////////////////////////////
```

文件 GLView.h

```
#include "stdafx.h"
#include "OpenGLDC.h"

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

COpenGLDC::COpenGLDC(HWND hWnd):m_hWnd(hWnd)
{
}
```

```
COpenGLDC::~COpenGLDC()
{
}

BOOL COpenGLDC::InitDC()
{
	if (m_hWnd == NULL) return FALSE;

	m_Camera.init();

	m_hDC = ::GetDC(m_hWnd);			// Get the device context

	PIXELFORMATDESCRIPTOR pfdWnd =
	{
		sizeof(PIXELFORMATDESCRIPTOR), // Structure size
		1,					// Structure version number
		PFD_DRAW_TO_WINDOW |		// Property flags
		PFD_SUPPORT_OPENGL |
		PFD_DOUBLEBUFFER,
		PFD_TYPE_RGBA,
		24,					// 24-bit color
		0, 0, 0, 0, 0, 0,			// Not concerned with these
		0, 0, 0, 0, 0, 0, 0,			// No alpha or accum buffer
		32,					// 32-bit depth buffer
		0, 0,					// No stencil or aux buffer
		PFD_MAIN_PLANE,			// Main layer type
		0,					// Reserved
		0, 0, 0				// Unsupported
	};

	int pixelformat;
	if ( (pixelformat = ChoosePixelFormat(m_hDC, &pfdWnd)) == 0 )
	{
		AfxMessageBox("ChoosePixelFormat to wnd failed");
		return FALSE;
	}

	if (SetPixelFormat(m_hDC, pixelformat, &pfdWnd) == FALSE)
		AfxMessageBox("SetPixelFormat failed");

	m_hRC=wglCreateContext(m_hDC);
	VERIFY(wglMakeCurrent(m_hDC,m_hRC));
	GLSetupRC();
	wglMakeCurrent(NULL,NULL);
	return m_hRC!=0;
}
```

```
void COpenGLDC::GLResize(int w,int h)
{
	wglMakeCurrent(m_hDC,m_hRC);

	// Prevent a divide by zero
	if(h == 0) h = 1;
	if(w == 0) w = 1;
	m_Camera.set_screen(w,h);
	wglMakeCurrent(NULL,NULL);
}

void COpenGLDC::GLSetupRC()
{
	glEnable(GL_DEPTH_TEST);	// Hidden surface removal
	glEnable(GL_COLOR_MATERIAL);

	// Lighting components
	GLfloat	ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
	GLfloat	diffuseLight[] = { 0.6f, 0.6f, 0.6f, 1.0f };
	GLfloat	specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
	GLfloat	lightPos[] = { 1.0f, 1.0f, 1.0f, 0.0f };

	glEnable(GL_LIGHTING);
	glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);
	glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);
	glLightfv(GL_LIGHT0,GL_SPECULAR,specular);
	glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
	glEnable(GL_LIGHT0);

	glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
	glMaterialfv(GL_FRONT, GL_SPECULAR,specular);
	glMateriali(GL_FRONT,GL_SHININESS,100);

	glClearColor(0.0f, 0.0f, 0.0f, 1.0f ); //background color
	// default color
	glColor3ub(0, 0, 255);
}

void COpenGLDC::Ready()
{
	wglMakeCurrent(m_hDC,m_hRC);
	m_Camera.projection();
	glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}

void COpenGLDC::Finish()
{
```

```
        glFlush();
        SwapBuffers(m_hDC);
        wglMakeCurrent(NULL,NULL);
}
```

4.6.5 类 CGLView 的声明代码

```
////////////////////////////////////////////////////// GLView.h//////////////////////////////////////////////////////
#if !defined(AFX_GLVIEW_H__B2996431_1A48_492F_87B8_D8E922845B95__INCLUDED_)
#define AFX_GLVIEW_H__B2996431_1A48_492F_87B8_D8E922845B95__INCLUDED_

class COpenGLDC;
class CGLView : public CView
{
protected:
        COpenGLDC*    m_pGLDC;

protected: // create from serialization only
        CGLView();
        DECLARE_DYNCREATE(CGLView)

// Attributes
public:
        virtual void RenderScene(COpenGLDC* pDC);

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CGLView)
        public:
        virtual void OnDraw(CDC* pDC);    // overridden to draw this view
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CGLView();
protected:

// Generated message map functions
protected:
        //{{AFX_MSG(CGLView)
        afx_msg int    OnCreate(LPCREATESTRUCT lpCreateStruct);
        afx_msg void OnDestroy();
```

```
        afx_msg BOOL OnEraseBkgnd(CDC* pDC);
        afx_msg void OnSize(UINT nType, int cx, int cy);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
/////////////////////////////////////////////////////////////////////////////
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.
#endif // !defined(AFX_GLVIEW_H__B2996431_1A48_492F_87B8_D8E922845B95__INCLUDED_)
```

4.6.6 类 CGLView 的实现代码

```
////////////////////////////////////////////////// GLView.CPP/////////////////////////////////////////////////
#include "stdafx.h"
#include "GL.h"
#include "GLDoc.h"
#include "GLView.h"
#include "OpenGLDC.h"

/////////////////////////////////////////////////////////////////////////////
// CGLView
IMPLEMENT_DYNCREATE(CGLView, CView)

BEGIN_MESSAGE_MAP(CGLView, CView)
        //{{AFX_MSG_MAP(CGLView)
        ON_WM_CREATE()
        ON_WM_DESTROY()
        ON_WM_ERASEBKGND()
        ON_WM_SIZE()
        //}}AFX_MSG_MAP
END_MESSAGE_MAP()

CGLView::CGLView()
{
        m_pGLDC = NULL;
}

CGLView::~CGLView()
{
}
BOOL CGLView::PreCreateWindow(CREATESTRUCT& cs)
{
        cs.style |= WS_CLIPSIBLINGS|WS_CLIPCHILDREN;
        return CView::PreCreateWindow(cs);
}

void CGLView::OnDraw(CDC* pDC)
```

```
{
	if(m_pGLDC){
		m_pGLDC->Ready();
		RenderScene(m_pGLDC);
		m_pGLDC->Finish();
	}
}

int CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CView::OnCreate(lpCreateStruct) == -1)
		return -1;

	m_pGLDC = new COpenGLDC(GetSafeHwnd());
	m_pGLDC->InitDC();
	return 0;
}

void CGLView::OnDestroy()
{
	CView::OnDestroy();
	if(m_pGLDC)	delete m_pGLDC;
}

BOOL CGLView::OnEraseBkgnd(CDC* pDC)
{
	//return CView::OnEraseBkgnd(pDC);
	return FALSE;
}

void CGLView::OnSize(UINT nType, int cx, int cy)
{
	CView::OnSize(nType, cx, cy);

	if(m_pGLDC)
		m_pGLDC->GLResize(cx,cy);
}
void CGLView::RenderScene(COpenGLDC* pDC)
{
	// Rotate
	glRotatef(3.0f,0.0f, 0.0f, 1.0f); // Roll Z
	glRotatef(3.0f,1.0f, 0.0f, 0.0f); // Roll X

	glPushMatrix();
	glColor3ub(255,0,0);//red
	auxSolidSphere(360); //0,0,0
```

```
        glColor3ub(0,0,255);//blue
        glTranslatef(600,0,0);//600,0,0
        auxSolidSphere(120);

        glColor3ub(0,255,0);//green
        glTranslatef(-1200,600,0);//-600,0,0
        auxSolidSphere(60);

        glPopMatrix();
}
```

本章相关程序

- ch4\NewGL：**修改后的** OpenGL Windows **程序** NewGL **的工程。**
- ch4\NewGL.exe: **执行程序。**

第 5 章　基于 OpenGL 的 CAD 图形工具库的设计

本章要点：

- **基于** OpenGL **的** CAD **图形工具库** glContext.dll **的创建。**
- **为照相机类** GCamera **增加取景操作函数。**
- **对** OpenGL **的光照的操作/在类** COpenGLDC **中增加光照的操作函数。**
- **对** OpenGL **的颜色的操作/在类** COpenGLDC **中增加颜色的操作函数。**
- **在类** COpenGLDC **中增加图形绘制函数。**
- OpenGL **的通用视图类** CGLView **的功能增强。**

在第 4 **章中，我们创建了几个** C++**类，用于** OpenGL **功能的封装。使用这些类，能够以面向对象的方式实现** OpenGL **与** Windows **窗口的关联，以及对** OpenGL **的一些相关功能的操作。这一章，我们将把这些类从应用程序中独立出来，创建一个独立的** CAD **图形工具模块——**glContext.dll**，用于** CAD **模型的绘制与操作。**glContext.dll **可供不同的应用程序在运行时调用。上一章中所设计的几个类还只是完成了一个简单的功能框架，为了方便使用，在这一章中还将进一步完善和增强这些类的功能。**

5.1　创建动态链接库 glContext.dll

glContext.dll **的功能将主要集中于对** OpenGL **功能的封装与操作。设计这个库的目的在于：能够在应用程序中，通过对这个库的调用而更加方便地使用** OpenGL **进行** CAD **模型的显示与操作。**

在上一章设计的几个类的基础上，在 glContext.dll **中对这些类增加了一些新的功能，如照相机类的取景操作函数。而这些功能的实现都离不开几何计算，如点、矢量和变换矩阵等。在本书的第** 2 **章中介绍了一个几何工具库** GeomCalc.dll **的开发，在开发** glContext.dll **时可以直接调用** GeomCalc.dll **库中输出的几何元素类和相关计算，而不必再重复进行这些类的开发。图** 5-1 **显示了** glContext.dll **的构造基础。读者也可以从中体会到一点模块化开发的特点。**

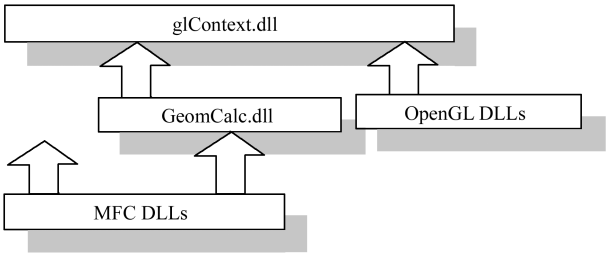


图 5-1　**动态链接库** glContext.dll **的构造基础**

在头文件 glContext.h 中说明了以下输出类。和第 4 章中所设计的这些类不同的是，在 glContext.dll 中，对它们的功能作了增强，并在每个类的定义中增加了 AFX_EXT_CLASS 说明，将它们定义为该 DLL 库的输出类。

- GCamera　　照相机类
- COpenGLDC　　OpenGL 绘图类
- CGLView　　OpenGL 视图基类

在本书附带的光盘目录“ch05\glContext”下给出了 glContext 项目完整的源代码，项目编译连接后，可将以下文件拷贝出来供其他应用程序调用。作为一个动态链接库，glContext 提供如下三个文件供其他应用程序（也包括动态链接库）调用：

- 动态链接库 glContext.dll
- 静态链接库 glContext.lib
- 输出类的头文件 glContext.h

5.2 类 GCamera 的功能增强

第 4 章中设计了一个照相机类 GCamera，它封装了 OpenGL 取景操作的最基本的功能。实际上，在 CAD 应用程序中，对景物显示的操作是软件中使用频率最高的操作之一，如观察三维几何模型时的视图选择操作（例如，定义左视图、俯视图、轴侧图），视图的缩放和移动操作(例如,景物的放大、缩小、平移、旋转等)。另外,由于 OpenGL 中提供的选择机制(Selection Mode）也是通过改变投影变换来实现的，因此也可以作为 GCamera 的一个功能封装在类中。在 glContext 中，功能增强后的类 GCamera 的主要功能划分如下：

- 定义视点的位置与方向。
- 定义投影变换。
- 定义选择区域。
- 用于观察模型的典型视图选择。
- 景物显示尺寸的缩放。
- 景物显示的移动。

在文件 camera.h 中，类 GCamera 的定义如下：

```
//宏定义，用于选择典型的观察视图
#define    VIEW_FRONT          0
#define    VIEW_BACK           1
#define    VIEW_TOP            2
#define    VIEW_BOTTOM         3
#define    VIEW_RIGHT          4
#define    VIEW_LEFT           5
#define    VIEW_SW_ISOMETRIC   6
#define    VIEW_SE_ISOMETRIC   7
#define    VIEW_NE_ISOMETRIC   8
#define    VIEW_NW_ISOMETRIC   9

//用于显示缩放的宏定义
```

```
#define ZOOM_ALL        10
#define ZOOM_IN         11
#define ZOOM_OUT        12

class AFX_EXT_CLASS GCamera
{
protected:
    //视点位置和方向
    CPoint3D   m_eye;
    CPoint3D   m_ref;
    CVector3D m_vecUp;

    //定义视景体的参数
    double          m_far, m_near;
    double          m_width,m_height;

    //视口参数
    double          m_screen[2];
public:
    //构造与析构函数
    GCamera();
    ~GCamera();

    //初始化函数
    void init();

    //取景定义，设置绘图模式下的投影变换
    void projection();

    //选择定义，设置选择模式下的投影变换
    void selection(int xPos,int yPos);

    //景物缩放
    void zoom(double scale);
    void zoom_all(double x0,double y0,double z0,double x1,double y1,double z1);

    //景物平移
    void move_view(double dpx, double dpy);

    //选择典型视图
    void set_view_type(int type);

    //设置视口尺寸
    void set_screen( int x, int y);

    //设置视点位置和方向
    void set_eye(double eye_x,double eye_y,double eye_z);
```

```
        void set_ref(double ref_x,double ref_y,double ref_z);
        void set_vecUp(double up_dx,double up_dy,double up_dz);

        //设置视景体
        void set_view_rect(double width,double height);
        void get_view_rect(double& width,double& height);
protected:
        //计算视线上方向的矢量
    void update_upVec();
};
#endif
```

和第 4 章中设计的类相比，GCamera 增加了以下几个方面的功能：选择典型的观察视图、景物平移、景物缩放、定义选择区域（选择模式下的投影变换）。下面将对这些功能的开发予以介绍。由于对选择功能的操作涉及到使用 OpenGL 选择模式的较多知识，对选择功能的开发将在本书第 9 章介绍使用 OpenGL 实现 CAD 软件的拾取功能时再作专门介绍。

5.2.1 选择典型的观察视图

在 CAD 软件中，观察视图的选择，即定义从什么角度观察模型，是最常用的操作之一。对于一个模型，用户需要从不同的位置和角度对它进行观察。我们常使用一些典型的观察角度，这就是机械制图中常说的视图。常用的视图有：左视图（Left View）、右视图（Right View）、顶视图（Bottom View）、俯视图（Top View）、前视图（Front View）、后视图（Back View）、轴侧图（Isometric View）。轴侧图又根据不同的观察方向予以定义，如 AutoCAD 中定义了如下四个观察方向的轴侧图：SW Isometric View、SE Isometric View、NE Isometric View、NW Isometric View。

在 OpenGL 中，对观察模型的位置和角度的定义是通过取景变换来实现的。在上一章中已经分析了 OpenGL 的取景变换的有关技术。我们使用三维 CAD 软件时，为了观察的需要，经常要进行典型观察视图的选择。为了操作方便，在类 GCamera 中增加对典型观察视图的定义是有必要的。如图 5-2、图 5-3 所示，GCamera 中定义了 10 个典型的观察视图：

- 前视图：Front View
- 后视图：Back View
- 俯视图：Top View
- 顶视图：Bottom View
- 右视图：Right View
- 左视图：Left View
- 西南方向轴侧图：SW Isometric View
- 东南方向轴侧图：SE Isometric View
- 东北方向轴侧图：NE Isometric View
- 西北方向轴侧图：NW Isometric View

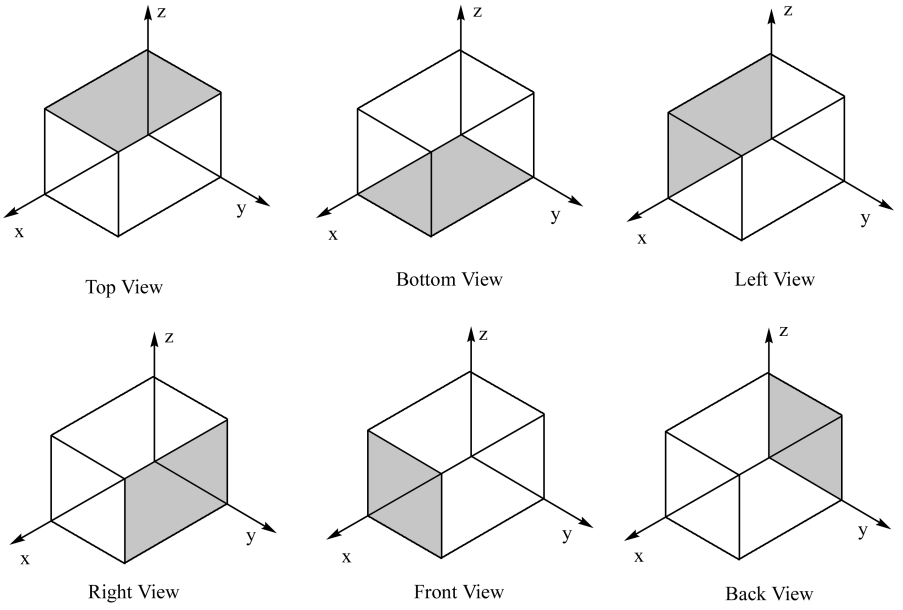


图 5-2　GCamera **中的典型视图定义**(1)

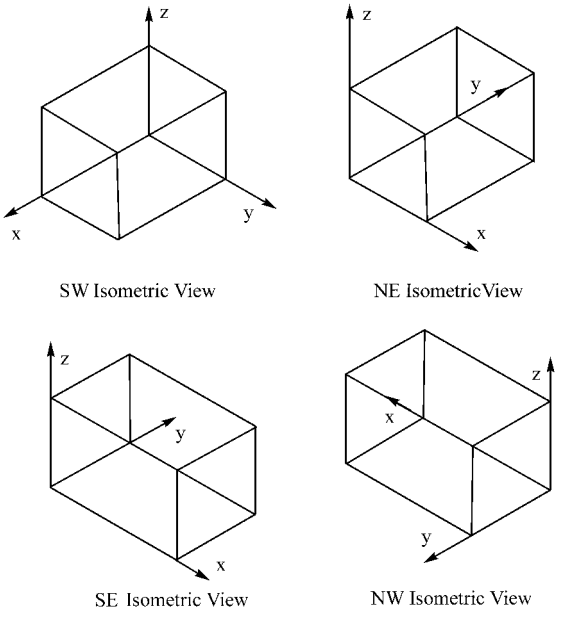


图 5-3　GCamera **中的典型视图定义**(2)——**轴侧图定义**

定义观察视图实际上就是定义观察者的位置和视线的方向。在 GCamera **中使用函数** gluLookAt()**来定义视点的位置和方向，**gluLookAt()**使用视点位置** m_eye**、参照点（视线正前方的一个位置）**m_ref**、视线的上方向** m_vecUp **来定义视图变换，这样类似于用户用肉眼观察景物，几何意义非常直观。定义一个典型观察视图，只需要定义合适的视点、参照点和视线的上方向三个参数。由于参照点** m_ref **通常取在被观察的模型接近中心的位置，所以定义不同的观察视图时，尽量保持参照点的位置不变，再通过参照点位置以及视线方向推算出视**

点 m_eye 的位置。计算出新的视点位置以后，还需要重新计算视线的上方向 m_vecUp。函数 update_upVec()的功能是根据新的视点位置，更新视线上方向 m_vecUp。

类 GCamera 中典型观察视图选择函数 GCamera::set_view_type( )实现代码如下，其中输入的参数 type 是文件 camera.h 中对典型观察视图的宏定义。

```
void GCamera::set_view_type( int type ) //典型观察视图选择
{
    double r;
    CVector3D vec;

    vec = m_ref - m_eye;          //矢量 vec 表示视线方向
    r = vec.GetLength();          //视点与参照点的距离

    if(IS_ZERO(r)) r = 50.0;      //防止视点与参照点重合
    if( r > 10000)   r = 10000;   //防止视点距离参照点太远

    switch(type){
    case VIEW_FRONT:          //前视图，
        m_eye = m_ref + CVector3D(0,-r,0);      //移动视点位置
        m_vecUp = CVector3D(0,0,1);
        break;
    case VIEW_BACK:           //后视图
        m_eye = m_ref + CVector3D(0,r,0);       //移动视点位置
        m_vecUp = CVector3D(0,0,1);
        break;
    case VIEW_TOP:            //俯视图
        m_eye = m_ref + CVector3D(0,0,r);       //移动视点位置
        m_vecUp = CVector3D(0,1,0);
        break;
    case VIEW_BOTTOM:         //顶视图
        m_eye = m_ref + CVector3D(0,0,-r);      //移动视点位置
        m_vecUp = CVector3D(0,1,0);
        break;
    case VIEW_RIGHT:          //右视图
        m_eye = m_ref + CVector3D(r,0,0);       //移动视点位置
        m_vecUp = CVector3D(0,0,1);
        break;
    case VIEW_LEFT:           //左视图
        m_eye = m_ref + CVector3D(-r,0,0);      //移动视点位置
        m_vecUp = CVector3D(0,0,1);
        break;
    case VIEW_SW_ISOMETRIC:    //SW 轴侧图
        m_eye = m_ref + CVector3D(-1,-1,1).GetNormal()*r;//移动视点位置
        update_upVec();       //更新视线上方向矢量
        break;
    case VIEW_SE_ISOMETRIC:    //SE 轴侧图
        m_eye = m_ref + CVector3D(1,-1,1).GetNormal()*r;//移动视点位置
```

```
            update_upVec();          //更新视线上方向矢量
            break;
        case VIEW_NE_ISOMETRIC:      //NE 轴侧图
            m_eye = m_ref + CVector3D(1,1,1).GetNormal()*r;  //移动视点位置
            update_upVec();          //更新视线上方向矢量
            break;
        case VIEW_NW_ISOMETRIC:      //NW 轴侧图
            m_eye = m_ref + CVector3D(-1,1,1).GetNormal()*r;
            update_upVec();          //更新视线上方向矢量
            break;
        }
}

void GCamera::update_upVec()
{
        CVector3D vec = m_ref - m_eye;   //视线方向矢量
        CVector3D zVec(0,0,1);
        CVector3D vec0;

        vec.Normalize();          //矢量单位化
        vec0 = vec*zVec;
        m_vecUp = vec0*vec;  //矢量 m_vecUp 与视线方向垂直
}
```

5.2.2 景物平移

使用 CAD 软件时，常常会使用景物平移功能来观察当前窗口以外的景物。实际上，景物平移的过程就是视点平移的过程。如图 5-4 所示，观察者位于视点 eye 的位置，x 轴、y 轴分别对应于屏幕上景物窗口的坐标轴，xy 轴构成的平面与视线垂直。景物平移时，保持视线方向不变，在与视线垂直的平面内同时移动视点 eye 和参照点 ref 的位置。在类 GCamera 中，设计了函数 GCamera::move_view()用于景物平移，参数 dpx、dpy 分别是沿景物窗口的 x 和 y 轴方向上移动的百分比，范围是[0，1]。例如，dpx=0.05 表示沿屏幕向右移动视景宽度的 5%，dpy=−0.05 表示沿屏幕向下移动视景宽度的 5%。下面是函数 GCamera::move_view()的实现代码。代码中，需要首先计算出景物窗口的 x 轴、y 轴对应于 OpenGL 用户坐标系的矢量方向，然后再对 m_eye、m_ref 进行平移。

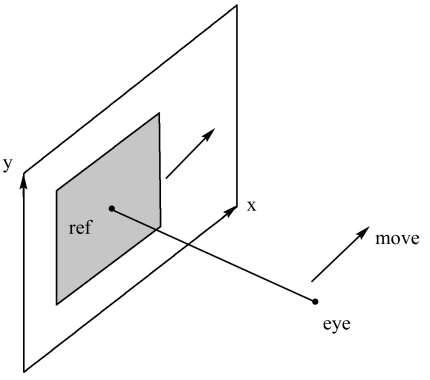


图 5-4　景物平移

函数 GCamera::move_view()的实现代码如下：

```
void GCamera::move_view(double dpx, double dpy)
{
        CVector3D vec;
        CVector3D xUp, yUp;
```

```
    vec = m_ref - m_eye;          //视线方向矢量
    vec.Normalize();              //单位化视线方向矢量
    xUp = vec*m_vecUp;            // xUp:  景物窗口的 x 轴对应于 OpenGL 用户坐标系的矢量
    yUp = xUp*vec;                // yUp:  景物窗口的 y 轴对应于 OpenGL 用户坐标系的矢量

    m_eye -= xUp*m_width*dpx + yUp*m_height*dpy; //移动视点位置
    m_ref -= xUp*m_width*dpx + yUp*m_height*dpy;  //移动参照点位置
}
```

5.2.3 景物缩放

在观察景物时，经常需要对景物进行放大、缩小，或者缩放景物使得景物空间中所有的模型都显示在窗口中。如 AutoCAD 中的 Zoom In、Zoom Out、Zoom All 等功能，都是改变景物显示大小的操作。反过来考虑，在一个窗口中，景物的显示尺寸被缩小一倍，实际上是将视野（视景体的宽和高）扩大了一倍。对景物显示尺寸的缩小和放大，实际上是对视野的放大和缩小。CAD 软件大多使用正交投影方式，定义了一个长方体形状的视景体。可以通过改变视景体的宽和高来实现景物的缩放。

1．景物放大（Zoom In）和缩小（Zoom Out）

GCamera 中设计了函数 GCamera::zoom()来等比例地缩放视景体的宽（m_width）和高（m_height），以实现对景物显示尺寸的缩放。函数实现如下：

```
void GCamera::zoom(double scale)
{
    ASSERT(scale > 0.0);          //缩放尺寸必须大于 0
    m_width *= scale;             //缩放视景体的宽
    m_height *= scale;            //缩放视景体的高
}
```

当缩放比例 scale<1.0 时，景物空间被缩小，则场景空间中的物体显示尺寸被放大；当缩放比例 scale>1.0 时，景物空间被放大，场景空间中的物体显示尺寸则被缩小。应用时，在应用程序的视图类中适当的地方调用函数 GCamera::zoom()改变视景体的大小，并接着调用函数 CView::Invalidate()刷新屏幕。当屏幕刷新重新绘制场景时，由于会首先调用函数 GCamera::projection()来根据当前的 m_width、m_height 等参数设置投影空间（视景体），所以所绘制的景物的尺寸被相应地改变。

2．显示全部模型（Zoom All）

很多情况下，在 CAD 应用程序中会要求将景物空间中所有的模型都显示在屏幕上，以便观察模型的整体，如 AutoCAD 软件中的 Zoom All 功能。要实现这个功能，实际上需要根据模型的空间尺寸，计算出一个合适的视景体空间以包容整个模型。由于 CAD 模型的几何形状有可能非常复杂，通常使用一个长方体形状的包容盒来描述这个模型的所占空间。下列函数 GCamera::zoom_all()的 6 个参数分别是这个包容盒的两个角点(x0, y0, z0)、(x1, y1, z1)的坐标，且 x0<x1、y0<y1、z0<z1。函数根据这个包容盒的尺寸，首先算出一个能够包含这个包容盒的视景体，并据此重新定义视景体的尺寸。最后，还需要调整视点 m_eye 和参照点 m_ref 的位置，即将参照点调整到模型的中央，然后相应地移动视点以保持视线方向不变。

函数 GCamera::zoom_all()**的实现如下：**

```
void GCamera::zoom_all(double x0,double y0,double z0,double x1,double y1,double z1)
{
    double width,height;
    double    xl, yl, zl;
    //模型包容盒的长宽高
    xl = x1-x0;
    yl = y1-y0;
    zl = z1-z0;

    //计算能够包含模型包容盒的视景体的宽和高
    width = max(max(xl,yl),zl);
    height= max(max(xl,yl),zl);

    //重新设置视景体的宽和高
    set_view_rect(width,height);

    //移动视点和参照点
    CVector3D vec = m_eye - m_ref;
    m_ref.x = (x0+x1)/2;
    m_ref.y = (y0+y1)/2;
    m_ref.z = (z0+z1)/2;
    m_eye = m_ref + vec;
}
```

5.2.4 使用 OpenGL 的选择模式

在 CAD **应用程序中，用户常常使用鼠标来拾取几何对象。拾取功能是实现** CAD **图形交互的重要工具。拾取的过程，实际上就是要计算出鼠标当前位置所覆盖的几何对象。有时在鼠标位置下有可能同时覆盖多个几何对象，因而还有必要返回每个对象的深度信息，以找出所需要的几何对象。用户可以自己设计算法来进行拾取的计算，**OpenGL **提供了一个选择模式（**Selection Mode**），专门用于实现鼠标的拾取。使用** OpenGL **的选择模式，会节省大量的开发时间。但值得研究的是，如何将** OpenGL **的选择模式和所开发的** CAD **应用程序很好地集成起来。**

OpenGL **的选择模式实际上是根据输入的鼠标的位置来定义一个有如射线般的细长形视景体。和正常的渲染模式（**Rendering Mode**）不同，在选择模式下图形不被绘制到屏幕上，而是记录下在这个细长形视景体空间中所绘制的所有对象的名称（也就是在鼠标位置之下的几何对象的名称），并将它们返回给应用程序。所以在选择模式下，需要根据鼠标位置定义一个选择视景体。在** GCamera **中，设计了函数** GCamera::selection()**用于定义选择的投影变换，即定义选择视景体，它的两个输入参数分别是当前的鼠标位置。**

定义选择视景体只是实现整个选择过程的一个环节，在本书的第 9 **章将详细介绍** OpenGL **的选择模式，以及如何将选择模式与** CAD **应用程序中的鼠标拾取功能有机地结合起来。**

函数 GCamera::selection()**的实现如下：**

```
void GCamera::selection(int xPos,int yPos)     //参数 x、y 分别是景物窗口中鼠标点的位置
{
    GLintvp[4];
    glGetIntegerv(GL_VIEWPORT,vp);      //获取当前视口坐标

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();                   //初始化投影变换矩阵

    //切换到选择模式
    glRenderMode(GL_SELECT);

    //定义选择投影矩阵
    gluPickMatrix(xPos,vp[3]-yPos, 1, 1, vp );
    double left     =   - m_width/2.0;
    double right    =   m_width/2.0;
    double bottom   =   - m_height/2.0;
    double top      =   m_height/2.0;
    glOrtho(left,right,bottom,top,m_near,m_far);

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
    gluLookAt(m_eye.x,m_eye.y,m_eye.z,m_ref.x,m_ref.y,m_ref.z,
            m_vecUp.dx, m_vecUp.dy, m_vecUp.dz);
}
```

5.3 类 COpenGLDC 功能的增强

在第 4 章中开发的类 COpenGLDC 实现了 OpenGL 绘图类的基本机制，即管理一个渲染场境（Rendering Context），以实现 OpenGL 与绘制窗口的关联，并通过这个类完成 OpenGL 在 Windows 窗口中图形绘制前后的一些主要环节的设置。

但作为一个绘图类，为了方便使用，有必要进一步增加更多的相关功能。在 glContext.dll 中，为类 COpenGLDC 设计了更多的相关功能，以便于绘图操作。它的主要功能划分如下：

- 实现和管理与 Windows 窗口的关联。
- 取景操作（通过 GCamera 的对象实现）。
- 绘制图形（在基本图元的基础上开发高一级的图形绘制命令）。
- 对光源的操作。
- 对颜色的操作。
- 对 OpenGL 选择功能的封装与操作。

类 COpenGLDC 的定义如下：

```
class AFX_EXT_CLASS COpenGLDC
{
public:
    //构造与析构函数
```

```
    COpenGLDC(HWND hWnd);
    virtual ~COpenGLDC();

private:
    //关联窗口的句柄
    HWND    m_hWnd;

    //渲染场境句柄
    HGLRC   m_hRC;

    //设备场境句柄
    HDC     m_hDC;

    //窗口背景色
    COLORREF    m_clrBk;

    //非光照模式下的模型颜色
    COLORREF    m_clr;

    //用于高亮度显示时的模型颜色，如拾取到一个物体时需要高亮度显示
    COLORREF    m_clrHighlight;

    //材料的颜色
    COLORREF    m_clrMaterial;

    //是否采用着色显示
    BOOL        m_bShading;

    //光源的方向
    GLfloat     m_vecLight[3];

    //当前是否是选择模式
    BOOL    m_bSelectionMode;

    //选择缓存区
    GLuint  m_selectBuff[BUFFER_LENGTH];
public:
    //照相机，用于取景操作
    GCamera     m_Camera;

protected:
    //清除背景颜色
    void ClearBkground();

    //光照/非光照模式设置
    void OnShading();
public:
```

```
//初始化
BOOL InitDC();

//对应窗口尺寸变化
void GLResize(int cx,int cy);

//设置渲染场境
void GLSetupRC();

//绘图前准备函数
void Ready();

//结束绘图函数
void Finish();

////////////////////// 对光照与颜色的操作函数 //////////////////////
//光照/非光照模式切换
void Shading(BOOL bShading);

//当前是否是着色模式
BOOL IsShading();

//是否使用光源
void Lighting(BOOL bLighting);
BOOL IsLighting();

//设置与获取光源方向
void SetLightDirection(float dx,float dy,float dz);
void GetLightDirection(float& dx,float& dy,float& dz);

//设置与获取材料颜色
void SetMaterialColor(COLORREF clr);
void GetMaterialColor(COLORREF& clr);

//设置与获取背景颜色
void SetBkColor(COLORREF rgb);
void GetBkColor(COLORREF& rgb);

//设置与获取非光照模式下的绘制颜色
void SetColor(COLORREF rgb);
void GetColor(COLORREF& rgb);

//设置与获取高亮度显示的颜色
void SetHighlightColor(COLORREF clr);
void GetHighlightColor(COLORREF& clr);

//高亮度/正常显示切换
```

```
void Highlight(BOOL bLight = TRUE);

////////////////////  基于 OpenGL 基本图元的高级绘图函数/////////////////////
//绘制一个空间点
void DrawPoint(const CPoint3D&);

//绘制用户坐标系
void DrawCoord();

//绘制一条直线
void DrawLine(const CPoint3D& sp,const CPoint3D& ep);

//绘制连续折线
void DrawPolyline(const CPoint3D* pt,int size);

//绘制一个三角面片
void DrawTriChip(double n0,double n1,double n2,double v00,double v01,double v02,
                 double v10,double v11,double v12,double v20,double v21,double v22);

//drawing solid entities
//绘制一个圆球
void DrawSphere(const CPoint3D& cen,double r,const CVector3D& vec);

//绘制一个圆柱
void DrawCylinder(const CPoint3D& cen,double r,const CVector3D& h);

//绘制一个圆锥
void DrawCone(const CPoint3D& cen,double r,const CVector3D& h);

//绘制一个圆环
void DrawTorus(const CPoint3D& cen,const CVector3D& ax,double r_in,double r_out);

//////////////////////////////////////  选择过程的操作函数/////////////////////////////////////////////////////////
//开始选择
void    BeginSelection(int xPos,int yPos);

//结束选择，返回选择记录
int     EndSelection(UINT* items);

//当前是否是选择模式
BOOL    IsSelectionMode();

//用于选择模式下的对象名称操作
void    InitNames();
void    LoadName(UINT name);
void    PushName(UINT name);
void    PopName();
```

};

5.3.1 实现和 Windows 窗口的关联

与第 4 章中设计的类 COpenGLDC 一样，以下函数用于实现 OpenGL 和 Windows 窗口的关联，并对渲染场境进行初始化。这些函数的功能、使用以及具体实现都已经介绍过（参见图 4-5），这里不再赘述。

- BOOL InitDC() 初始化
- void GLResize(int cx,int cy) 对应窗口尺寸变化
- void GLSetupRC() 设置渲染场境
- void Ready() 绘图前准备函数
- void Finish() 结束绘图函数

5.3.2 定义光源

OpenGL 的光照功能对于三维真实感图形的效果是非常重要的。事实上，如果没有光照，所绘制的三维图形就缺乏立体感，难以感觉到它与二维显示的区别。

自然界中，由光源（太阳、灯等）发出的光照射到物体表面时，可以被物体吸收、反射和透射。物体对光的吸收、反射和透射的程度是由物体的表面材料属性决定的。反射和透射的光进入到人的眼睛，这就是我们能够观察到的物体。如黑色物体吸收了所有的入射光，没有反射光进入肉眼，所以呈现黑色。以 OpenGL 模拟自然界的光照模式，提供了光源定义和材质定义的方式来实现光照效果。

1．光照的组成

如果一个物体本身不发光，我们能够观察到这个物体，是因为它被三种成分的光照亮。这三种光分别是：环境光（Ambient Light）、漫反射光（Diffuse Light）、镜面光（Specula Light）。

环境光：环境光是一种均匀散布的光，它在空间各个方向均匀散布。它由光源发出，但经过在空间中的多次散射而无法确定其方向。环境光均匀地照亮一个物体，即物体的所有表面、所有方向上被照亮的效果是相同的。

漫反射光：漫反射光来自一个特定的方向，但是它被物体均匀地向空间反射。虽然是均匀的反射，但反射的光强与入射光之间夹角的余弦成正比，即当入射光线垂直于物体表面时，物体最亮。

镜面光：镜面光具有方向性，并沿一个具体的方向反射入射光。一个强的镜面光将在材料表面造成一个亮点。

还有一种是辐射光，用于表示物体本身发出的光，例如物体是一个灯泡的情况。

OpenGL 独立计算所有四个部分，然后相加在一起，最终成为颜色缓存区中的每个像素的最终颜色。

2．定义光源

光源是本身可以发出光的对象，如太阳、灯泡等。没有光源就没有光照，光源的性质和被照射物体的材料属性共同决定了产生的光照效果。

OpenGL 提供了两种类型的光源：定位光源和平行光源。定位光源明确指定一个具体的光源的位置，它与被照射的场景之间存在有限的距离。使用定位光源，光线的角度随光源照

射到场景的不同位置而不同。如果我们定义一个距离场景无穷远的光源，光线照射到场景时是平行光，这如同太阳光照射到地球上是平行光一样。使用平行光源对计算量的耗费比使用定位光源要小，因为不需要考虑光线角度的变化。

这里要介绍几个 OpenGL 关于定义光源的函数。

（1）使用光照——glEnable(GL_LIGHTING)

glEnable(GL_LIGHTING)告诉 OpenGL 使用光照计算，即 OpenGL 将使用物体的材料属性和光照设置来计算场景中每个像素的颜色。

关闭光照计算，使用 glDisable(GL_LIGHTING)。

（2）设置光源函数——glLight<f,i>v()

函数原型：

```
void glLightfv(GLenum light, GLenum pname, const GLfloat *params);
void glLightiv(GLenum light, GLenum pname, const GLint *params);
```

功能：

设置光源的参数，即光源的组成。

参数说明：

light——指定一个光源，即光源的序号。OpenGL 最多可以定义 8 个光源，序号分别从 GL_LIGHT0～GL_LIGHT7。

pname——所要设置的光源参数。

params——所要设置的光源参数的参数值。

pname 的可选项及功能见表 5-1。

表 5-1　pname 的可选项及功能

pname 参数	表示的功能
GL_AMBIENT	环境光分量
GL_DIFFUSE	漫反射光分量
GL_SPECULAR	镜面光分量
GL_POSITION	光源位置
GL_SPOT_DIRECTION	光源聚光方向
GL_SPOT_EXPONENT	光源聚光指数
GL_SPOT_CUTOFF	聚光的截止角
GL_CONSTANT_ATTENUATION	光的常数衰减因子
GL_LINEAR_ATTENUATION	光的线性衰减因子
GL_QUADRATIC_ATTENUATION	光的二次衰减因子

定义光源是由函数 glLightfv()实现的。glLightfv()有三个参数，第一个参数指定光源的序号，它是 GL_LIGHT0～GL_LIGHT7

（3）使用光源——glEnable(GL_LIGHTi)

OpenGL 中可以开启至多 8 个光源，分别是 GL_LIGHT0～GL_LIGHT7。使用 glEnable(GL_LIGHTi)，告诉 OpenGL 开始启用 i 号光源。要关闭一个已启用的光源，使用 glDisable(GL_LIGHTi)。

下面列出的是第 4 章中创建的类 COpenGLDC 光源定义的有关代码。在函数 COpenGLDC::GLSetupRC()中定义了一个光源 GL_LIGHT0，它由环境光、漫反射光、镜面光三个分量组成。光源位置用齐次坐标的方式定义为无穷远，因而照射到物体上时是平行光。

```
void COpenGLDC::GLSetupRC()
{
    ……
    //定义一个光源
    //设置环境光的颜色组成
    GLfloat    ambientLight[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    //设置漫反射光的颜色组成
    GLfloat    diffuseLight[] = { 0.6f, 0.6f, 0.6f, 1.0f };
    //设置镜面光的颜色组成
    GLfloat    specular[] = { 1.0f, 1.0f, 1.0f, 1.0f};
    //光源位置，沿矢量(1,1,1)方向无穷远
    GLfloat    lightPos[] = { 1.0f, 1.0f, 1.0f, 0.0f };

    //使用光照
    glEnable(GL_LIGHTING);

    //设置光源 GL_LIGHT0
    glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);     //为光源 0 设置环境光
    glLightfv(GL_LIGHT0,GL_DIFFUSE,diffuseLight);          //为光源 0 设置漫反射光
    glLightfv(GL_LIGHT0,GL_SPECULAR,specular);        //为光源 0 设置镜面光
    glLightfv(GL_LIGHT0,GL_POSITION,lightPos);        //设置光源 0 的位置

    //打开光源 GL_LIGHT0
    glEnable(GL_LIGHT0);
    ……
}
```

在这段代码中，首先定义了 0 号光源 GL_LIGHT0。GL_LIGHT0 由三个光源分量组成，即环境光、漫反射光和镜面光。GL_LIGHT0 的光源位置定义为：

```
GLfloat    lightPos[] = { 1.0f, 1.0f, 1.0f, 0.0f };
```

这里 lightPos 采用齐次坐标表示方法。由于第四个分量为 0，实际上它表示一个在矢量（1,1,1）方向上无穷远的一个位置，这时的光线可以被认为是从无穷远处射来的平行光。

待光源 GL_LIGHT0 的参数设置完毕后，调用 glEnable(GL_LIGHT0)打开这个光源。

3．物体材质

OpenGL 中材质的概念是指物体表面对光的反射特性。这个反射特性包括对环境光、漫反射光、镜面光的反射率。材质影响材料的颜色、反光度和透明度等。例如在白色的阳光下，我们看到一个非光源的红色的物体，这是因为这个物体只反射太阳光中的红色分量，而吸收其他颜色分量。类似地，OpenGL 使用材料对光的三个分量红、绿、蓝的反射率来定义材质。一般来说，物体对环境光和漫反射光的反射方式是相同的，因而对环境光和漫反射光的反射率基本上是一致的。环境光和漫反射光决定物体的主要部分的颜色，而物体的高亮色基本上

由对镜面光的反射率决定。在设计光照和材质时，往往需要具体分析所需要的材质特性，将这些特性组合起来以达到需要的效果。

这里介绍 OpenGL 中与设置材质有关的函数 glMaterial<f,i>v()。

函数原型：

```
void glMaterialf( GLenum face, GLenum pname, GLfloat param );
void glMateriali( GLenum face, GLenum pname, GLint param );
```

功能：

设置光照计算中被照物体的材质参数。

参数说明：

face——OpenGL 常量，指要被设置材质属性的物体的面，取值必须是 GL_FRONT、GL_BACK、 GL_FRONT_AND_BACK 之一。这三个参数分别说明多边形的前面、后面和双面。在实体模型中，由于只需要显示模型的外表面，后面的面是不可见的，因而可以只设置朝外的表面（前面）的材质属性。正如在函数 COpenGLDC::SetMaterialColor()中，只设置三角片朝外的面 GL_FRONT(默认状态下，三角片的顶点逆时针旋转的面为正面)。

pname——OpenGL 常量，指要被设置的物体的材质参数。它告诉 OpenGL 现在定义材质的哪一个属性。其可选的常量和默认值如表 5-2 所示。

表 5-2　参数 pname 的选项和对应的默认值

pname 常量	含　义	默 认 值
GL_AMBIENT	材料对环境光的反射率	(0.2,0.2,0.2,1.0)
GL_DIFFUSE	材料对漫反射光的反射率	(0.8,0.8,0.8,1.0)
GL_AMBIENT_AND_DIFFUSE	材料对环境光和漫反射光的反射率	
GL_SPECULAR	材料对镜面光的反射率	(0.0,0.0,0.0,1.0)
GL_EMISSION	材料的辐射光	(0.0,0.0,0.0,1.0)
GL_SHININESS	镜面指数（光亮度）	0.0
GL_COLOR_INDEXES	用于颜色索引模式	

param——要被设置的材质参数的具体数值。

4．库 glConext.dll 中有关光源的操作

在库 glContext.dll 的类 COpenGLDC 中增加了对光源的操作函数，这些函数封装了 OpenGL 光源操作的相关命令。COpenGLDC 中只设置了一个平行光源 GL_LIGHT0，所设计的函数都是针对这个光源进行操作。对这些函数的功能与实现介绍如下：

（1）操作光源方向。由于 COpenGLDC 中使用的是平行光，光源的位置位于距离场景无穷远处，所以我们关心的是如何操作光源的方向，而不是位置。使用 COpenGLDC 中设计的光源操作函数，可以在程序中随时改变光源的方向。函数 SetLightDirection() 与 GetLightDirection()分别用于设置和获取光源方向。

1）函数 SetLightDirection()。函数 SetLightDirection()用于设置光源的方向，输入参数 dx、dy、dz 是光源的方向矢量。函数实现如下：

```
void COpenGLDC::SetLightDirection(float dx,float dy,float dz)
{
```

```
        m_vecLight[0] = dx;    //m_vecLight 是类中保存当前光源方向的成员变量
        m_vecLight[1] = dy;
        m_vecLight[2] = dz;

        //用齐次坐标将光源位置设置在沿矢量（dx, dy, dz）方向的无穷远处
        GLfloat lightPos[] = { dx, dy, dz, 0.0f };
        glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
}
```

2）函数 GetLightDirection()。函数 GetLightDirection()用于获取当前光源的方向，实现如下：

```
void COpenGLDC::GetLightDirection(float& dx,float& dy,float& dz)
{
        dx = m_vecLight[0];
        dy = m_vecLight[1];
        dz = m_vecLight[2];
}
```

（2）光源开关。在 COpenGLDC 中，设计了函数来开关所定义的光照。如果光照被关闭，则不使用 OpenGL 的光照模式计算场景中物体的颜色，物体将缺乏立体感。但有的场合下，需要关闭光照，例如在绘制坐标系框架时，如果希望每根坐标轴的颜色均匀一致，可以暂时关闭光照，在绘制完坐标系后再打开光照。有关操作函数介绍如下：

1）函数 Lighting()。函数 Lighting()用于打开或关闭光照，如：

Lighting(TRUE)　使用光照

Lighting(FALSE)　关闭光照

函数实现如下：

```
void COpenGLDC::Lighting(BOOL bLighting)
{
        if(bLighting)
                glEnable( GL_LIGHTING );
        else
                glDisable( GL_LIGHTING );
}
```

2）函数 IsLighting()。函数 IsLighting()判断当前是否使用光照，实现如下：

```
BOOL COpenGLDC::IsLighting()
{
        GLboolean bLighting;
        glGetBooleanv(GL_LIGHTING,&bLighting);
        return bLighting;
}
```

（3）设置材质。函数 SetMaterialColor()用于设置当前物体的材质。这里材质被设置成对环境光和漫反射光的反射。因为在我们熟悉的 CAD 软件中，几何模型的显示通常采用的是较柔和的环境光源，以较均匀的亮度来显示物体。所以，这里不设置对镜面光的反射属性。

函数 SetMaterialColor()**的实现如下：**

```
void COpenGLDC::SetMaterialColor(COLORREF clr)
{
    m_clrMaterial = clr;    // m_clrMaterial 是类中保存擦材料颜色的成员变量
    BYTE r,g,b;
    r = GetRValue(clr);     //获取颜色 clr 的红色分量
    g = GetGValue(clr);     //获取颜色 clr 的绿色分量
    b = GetBValue(clr);     //获取颜色 clr 的蓝色分量

    //将以上颜色设置为当前的材质属性
    GLfloat mat_amb_diff[] = {(GLfloat)r/255,(GLfloat)g/255,(GLfloat)b/255,1.0};
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_amb_diff);
}
```

（4）在函数 GLSetupRC()**中初始化设置光源。函数** GLSetupRC()**执行渲染场境的初始化。在函数中创建了一个光源和它的默认方向。如果在程序中不调用有关函数去重新设置光源方向，则光源一直使用这个默认的方向。函数** COpenGLDC::GLSetupRC()**的实现如下：**

```
void COpenGLDC::GLSetupRC()
{
    m_bShading = TRUE;              //使用着色模式

    glEnable(GL_DEPTH_TEST);    //使用消隐
    glEnable(GL_CULL_FACE);     //不计算对象内部
    glFrontFace(GL_CCW);            //三角片顶点逆时针方向（CCW）旋转表示模型的外表面

    //环境光分量
    GLfloat lightAmbient[] = {0.75f,0.75f,0.75f,1.0f};
    //漫反射光分量
    GLfloat lightDiffuse[] = {1.0f,1.0f,1.0f,1.0f};
    //使用光照模式
    glEnable(GL_LIGHTING)

    //设置 0 号光源的组成分量
    glLightfv(GL_LIGHT0,GL_AMBIENT,lightAmbient);
    glLightfv(GL_LIGHT0,GL_DIFFUSE,lightDiffuse);

    //设置光源的方向
    SetLightDirection(1,1,1);

    //打开 0 号光源
    glEnable(GL_LIGHT0);

    //设置默认的颜色属性
    SetBkColor(RGB(0,0,0));             //设置背景的默认颜色（黑色）
    SetMaterialColor(RGB(225,175,22));  //设置材料的默认颜色
    SetColor(RGB(255,255,255));         //设置框架显示的默认颜色（白色）
```

```
        SetHighlightColor(RGB(255,0,0));        //设置高亮度显示的颜色（红色）

        glPointSize(3.0);        //设置点的绘制尺寸
}
```

函数 GLSetupRC()中有关颜色操作的函数将在下面予以介绍。

5.3.3 定义颜色

1．背景颜色

背景颜色用于刷新窗口的背景。在每次进行场景绘制之前，几乎所有的应用程序都需要首先进行清屏，即使用设定的清除颜色刷新整个缓冲区。这就是我们看到的背景颜色。在第 4 章介绍的应用程序 GL 中，我们看到的是一个黑色背景的应用程序窗口。这是因为在类 COpenGLDC:: GLSetupRC()初始化渲染场境时设置了清除颜色为黑色。

```
void COpenGLDC::GLSetupRC()
{
        …….
        glClearColor(0.0f, 0.0f, 0.0f, 1.0f ); //设置清除颜色为黑色
        ……..
}
```

OpenGL 函数 glClearColor()用于设定清除颜色，在调用窗口清除函数 glClear()时，使用这个设定的颜色刷新整个窗口。在第 4 章设计的类 COpenGLDC 中，在每次绘制之前，即在绘图准备函数 COpenGLDC::Ready()中调用了这个背景色刷新绘图窗口。

```
void COpenGLDC::Ready()
{
        wglMakeCurrent(m_hDC,m_hRC);
        m_Camera.projection();

        //用清除颜色刷新整个窗口

        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
}
```

这样，使得每次绘制的场景都是以黑色为背景。

但是，用户或许希望能够在程序中在任意需要的时候设置或改变背景颜色，而不是使用一个固定的背景颜色。在增强了的 COpenGLDC 类中，增加了对背景颜色的设置，使得用户能够设置并在任意的时候修改背景色。所定义的相关的变量和成员函数如下：

（1）成员变量 COpenGLDC::m_clrBk

类 COpenGLDC 中定义了一个成员变量 m_clrBk，用于保存设置的背景颜色，定义如下：

```
COLORREF      m_clrBk；
```

m_clrBk 是 COLORREF 类型的变量。COLORREF 是 Windows 中定义的用于描述 RGB 颜色的 32 位数，它的结构如下：

0x00bbggrr

其中 bb、gg、rr **分别是** RGB **颜色模式下的蓝、绿、红三个颜色分量，取值范围是** 0～255。Windows **函数中与** COLORREF **操作相关的宏定义如下。**

通过 Red、Green、Blue **三个颜色分量合成** COLORREF **变量：**

```
COLORREF RGB(
    BYTE bRed,      // red component of color
    BYTE bGreen,    // green component of color
    BYTE bBlue      // blue component of color
);
```

获取 COLORREF **中红色颜色分量：**

```
BYTE GetRValue(   DWORD rgb   // 32-bit RGB value);
```

获取 COLORREF **中绿色颜色分量：**

```
BYTE GetGValue(   DWORD rgb   // 32-bit RGB value);
```

获取 COLORREF **中蓝色颜色分量：**

```
BYTE GetBValue(   DWORD rgb   // 32-bit RGB value);
```

（2）成员函数 COpenGLDC::SetBkColor()

函数 COpenGLDC::SetBkColor()**用于设置背景颜色。**

```
void COpenGLDC::SetBkColor(COLORREF clr)
{
    m_clrBk = clr;
}
```

（3）成员函数 COpenGLDC::GetBkColor()

函数 COpenGLDC::GetBkColor()**用于返回背景颜色。**

```
void COpenGLDC::GetBkColor(COLORREF& clr)
{
    clr = m_clrBk;
}
```

（4）成员函数 ClearBkground()

函数 ClearBkground()**是一个内部调用函数，在** COpenGLDC **的绘图准备函数** Ready()**中调用它进行背景刷新。在函数** SetBkColor()**中设置的背景色变量** m_clrBk **在函数** ClearBkground() **中被调用并被设置成清除颜色，接着调用** glClear()**清除窗口。在函数中，首先需要取出** m_clrBk **的颜色分量，并将颜色分量的取值范围从** 0～255 **转换到** 0～1 **之间。函数实现如下：**

```
void COpenGLDC::ClearBkground()
{
    GLclampf   r,g,b;
    //获取背景色变亮 m_clrBk 的颜色 RGB 分量
    r = (GLclampf)GetRValue(m_clrBk)/255.0;
```

```
        g = (GLclampf)GetGValue(m_clrBk)/255.0;
        b = (GLclampf)GetBValue(m_clrBk)/255.0;

        //设置清屏的 RGBA 颜色
        glClearColor(r,g,b,0.0f);

        //清屏
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
    }
```

下面介绍使用到的 OpenGL **用于背景清除的库函数：**glClearColor()、glClear()。

（1**）**OpenGL **库函数** glClearColor()

函数原型：

```
void glClearColor( GLclampf red,  GLclampf green,  GLclampf blue,  GLclampf alpha );
```

功能：

设定清除颜色。

参数说明：

red, green, blue, alpha**——指定清除颜色的** RGBA **分量，取值范围** 0**～**1**。**

（2**）函数** glClear()

函数原型：

```
void glClear( GLbitfield mask );
```

功能：

用设定的清除颜色 mask **刷新指定的缓冲区。**

参数说明：

指定被刷新的缓冲区可以是 GL_COLOR_BUFFER_BIT、GL_DEPTH_BUFFER_BIT、GL_ACCUM_BUFFER_BIT **和** GL_STENCIL_BUFFER_BIT**。对于这些缓存区，读者可能不全部明白，但不必为此担心，随着** OpenGL **的深入应用会逐步接触到这些内容。在这儿，我们需要知道的是颜色缓存区，也就是显示缓存区，显示在屏幕上的像素信息存储在颜色缓存区中。**

2．绘图颜色

在 OpenGL **绘制图形之前，必须指定用于绘图的颜色模式。**OpenGL **使用两种颜色模式，即** RGBA **模式和颜色索引模式。颜色模式一旦确定下来，就不能再进行更改。在** OpenGL **中，每一个几何图元的颜色值都是根据各个顶点要素来计算的。如果使用了光照，顶点的颜色则由光源特性、物体表面法矢和物体的表面材质特性共同决定。在执行完光照计算后，**OpenGL **还将执行阴影计算，用户可以选择没有明暗差别的阴影计算和平滑的阴影计算，不同的阴影计算将影响最终的像素颜色生成。**

至于是选用 RGBA **模式还是使用颜色索引模式，应该根据系统硬件以及应用程序的需要来决定。对于大多数系统，**RGBA **模式可以描述更多的颜色，而颜色索引模式描述的颜色较少。在** OpenGL **的光照模型、纹理映射模型中和生成雾化效果时，使用** RGBA **模式效果更佳。在** COpenGLDC **中，由于需要使用光照模式渲染三维场景，所以使用** RGBA **颜色模式。**

以下介绍在 RGBA 模式下指定颜色的有关 OpenGL 库函数。在 RGBA 模式下，可以使用 OpenGL 库函数 glColor*()来设置当前绘图颜色，函数 glColor*()有如下定义：

```
void glColor3<b s i f ub us ui>(TYPE red, TYPE green, TYPE blue );
void glColor4<b s i f ub us ui>(TYPE red, TYPE green, TYPE blue, TYPE alpha );
void glColor3<b s i f ub us ui>v(const TYPE* v);
void glColor4<b s i f ub us ui>v(const TYPE* v);
```

red、green、blue 以及 alpha 分别是要被设置的当前颜色 RGBA 的四个分量值。对于浮点型的参数，参数的取值在 0.0～1.0 之间，用于表示这个颜色分量的最弱（0.0）到最强（1.0）的范围。由于目前的计算机图像大多使用 24 位色或 32 位色来描述像素，即对每个颜色分量使用 8 位（256 个梯度，即 0~255）来描述。Windows 中也定义了表示颜色的类型 COLORREF，用一个 32 位整型数来描述一个 RGB 颜色，即使用 0～255 来表示颜色分量的最弱（0.0）到最强（1.0）的范围。有鉴于此，开发 Window 环境下的 OpenGL 应用程序，使用函数 glColor3ub()较为方便。

```
void glColor3ub( GLubyte red, GLubyte green, GLubyte blue);
```

其中，变量类型 GLubyte 是一个无符号的 8 位整型数，它和 BYTE 是一个类型，取值范围是 0～255。它们的类型定义如下：

```
typedef unsigned char    BYTE;
typedef unsigned char   GLubyte;
```

（1）成员变量 m_clr。COpenGLDC 的成员变量 m_clr 用于保存默认的绘图颜色。在成员函数 GLSetupRC()中被初始化为白色：RGB(255,255,255)。

（2）成员函数 SetColor()。成员函数 SetColor()用于设置 OpenGL 的绘图颜色。函数的实现如下：

```
void COpenGLDC::SetColor(COLORREF clr)
{
    m_clr = clr;
    BYTE r,g,b;
    //获取 m_clr 中的 RGB 成员分量
    r = GetRValue(clr);
    g = GetGValue(clr);
    b = GetBValue(clr);
    glColor3ub(r,g,b);      //设置绘图颜色
}
```

（3）成员函数 GetColor()。成员函数 GetColor()用于返回当前设置的绘图颜色。函数实现如下：

```
void COpenGLDC::GetColor(COLORREF& clr)
{
    clr =   m_clr;
}
```

有时，我们会使用非光照模式来绘制图形，这在一些场合（如绘制线框模型、绘制坐标

系）经常使用。使用非光照模式，物体表面的远近、角度等对颜色显示均无影响，这时模型看起来缺乏立体感。在非光照模式下，OpenGL 将使用函数 glColor()设定的颜色绘图。使用以上设计的 COpenGLDC 成员函数可用于这种情况下的颜色设置。

例如，COpenGLDC 成员函数 DrawCoord()用于绘制一个坐标系框架，使用非光照模式绘制，实现代码如下：

```
void COpenGLDC::DrawCoord()  //在用户坐标系原点绘制一个坐标系框架
{
    BOOL bLighting = IsLighting();    //保存当前光照模式
    Lighting(FALSE);        //关闭光照模式

    //获取当前场景的宽与高
    double width,height;
    m_Camera.get_view_rect(width,height);
    double len = min(width,height);
    len *= 0.2;          //取当前屏幕宽或高的 20%作为坐标轴的绘制长度
                         //这样无论场景空间放大或缩小，坐标轴的显示尺寸保持不变

    CPoint3D cPt,xPt,yPt,zPt;
    xPt.x = yPt.y = zPt.z = len;

    COLORREF old_clr;
    GetColor(old_clr);       //保存当前绘图颜色

    //用红色绘制 x 轴
    SetColor(RGB(255,0,0));
    DrawLine(cPt,xPt);      //DrawLine 是 COpenGLDC 中绘制直线的成员函数
    //用绿色绘制 y 轴
    SetColor(RGB(0,255,0));
    DrawLine(cPt,yPt);
    //用蓝色绘制 z 轴
    SetColor(RGB(0,0,255));
    DrawLine(cPt,zPt);

    Lighting(bLighting);    //恢复原先光照模式
    SetColor(old_clr);       //恢复原先绘图颜色
}
```

3．高亮度显示颜色

在 CAD 软件中，当用户选中一个物体时，常是用一个高亮度的颜色来显示当前选中的物体。在 COpenGLDC 中，我们需要定义一个颜色作为这个高亮度颜色。

（1）成员变量 m_clrHighlight

```
COLORREF      m_clrHighlight;
```

成员变量 m_clrHighlight 用于描述所设置的高亮度颜色。在渲染场境初始化函数中，设置了默认的高亮度显示颜色 RGB(255,0,0)。

（2）设置高亮度显示的颜色

```
void COpenGLDC::SetHighlightColor(COLORREF clr)
{
    m_clrHighlight = clr;
}
```

（3）获取高亮度显示的颜色

```
void COpenGLDC::GetHighlightColor(COLORREF& clr)
{
    clr = m_clrHighlight;
}
```

（4）使用/关闭高亮度显示

函数 Highlight()**用于使用或关闭高亮度显示。高亮度显示在光照模式下进行。高亮度显示关闭之后，**OpenGL **使用变量** m_clrMaterial **所定义的材质计算颜色。函数的实现如下：**

```
void COpenGLDC::Highlight(BOOL bHighlight)
{
    BYTE r,g,b;
    if(bHighlight){ //使用高亮度色 m_clrHighlight 的 RGB 分量
        r = GetRValue(m_clrHighlight);
        g = GetGValue(m_clrHighlight);
        b = GetBValue(m_clrHighlight);
    }
    else{       //使用材质色 m_clrMaterial 的 RGB 分量
        r = GetRValue(m_clrMaterial);
        g = GetGValue(m_clrMaterial);
        b = GetBValue(m_clrMaterial);
    }
    GLfloat mat_amb_diff[] = {(GLfloat)r/255,(GLfloat)g/255,(GLfloat)b/255,1.0};
    glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_amb_diff);
}
```

5.3.4 图形绘制函数

在 OpenGL **程序中，用户可以使用** OpenGL **提供的基本图元命令组合并绘制出各种图形。类** COpenGLDC **中设计了一些高一级的图形绘制函数，在这些函数中封装** OpenGL **的基本图元绘制命令，这样用户可以直接使用** COpenGLDC **进行图形绘制，而不必理会封装在函数中的具体的** OpenGL **命令。这样设计使** COpenGLDC **的功能更类似于** GDI **的绘图类** CDC**。在库** glContext.dll **中，为类** COpenGLDC **设计了如下常用的绘图函数，并可以根据需要进一步增加绘图函数。**

1．绘制用户坐标系的函数——DrawCoord()

在 CAD **应用程序的视图窗口中，经常需要绘制空间坐标系。在类** COpenGLDC **中设计了用于在原点绘制坐标系的函数** DrawCoord()**。在函数** DrawCoord()**中，使用不同颜色绘制了** x**、**y**、**z **三个坐标轴。由于我们希望坐标轴的绘制颜色是均匀的，即不受光照的影响，所以在绘**

制坐标系之前关掉了光照设置，并在绘制完成后予以恢复。和绘制几何对象不一样的是，我们希望坐标系的显示大小不随场景空间大小的变化而变化，否则，场景空间大小变化时，坐标系的显示可能变得太大或者太小。因而，在函数中需要根据场景的尺寸来决定坐标轴的绘制长度。函数的具体实现如下：

```
void COpenGLDC::DrawCoord()
{
    //关闭光照模式
    BOOL bLighting = IsLighting();
    Lighting(FALSE);

    //使坐标轴的显示长度为视景体宽或高的 20%
    double width,height;
    m_Camera.get_view_rect(width,height);
    double len = min(width,height);
    len *= 0.2;

    CPoint3D cPt,xPt,yPt,zPt;
    xPt.x = yPt.y = zPt.z = len;

    COLORREF old_clr;
    GetColor(old_clr);

    //x 轴（红色）
    SetColor(RGB(255,0,0));
    DrawLine(cPt,xPt);

    //y 轴（绿色）
    SetColor(RGB(0,255,0));
    DrawLine(cPt,yPt);

    //z 轴（蓝色）
    SetColor(RGB(0,0,255));
    DrawLine(cPt,zPt);

    //恢复光照模式
    Lighting(bLighting);

    //恢复绘图颜色
    SetColor(old_clr);
}
```

2．绘制一个空间点的函数——DrawPoint()

函数DrawPoint()在三维空间中绘制一个点。在COpenGLDC的场境设置函数GLSetupRC()中已经设置了点的绘制尺寸（调用函数glPointSize()）。函数的实现如下：

```
void COpenGLDC::DrawPoint(const CPoint3D& pt)
```

```
{
    glBegin(GL_POINTS);
        glVertex3f(pt.x,pt.y,pt.z);
    glEnd();
}
```

3．绘制直线的函数——DrawLine()

函数 DrawLine()在三维空间中绘制一条由点 sp、ep 所定义的直线。该函数实现如下：

```
void COpenGLDC::DrawLine(const CPoint3D& sp,const CPoint3D& ep)
{
    glBegin(GL_LINES);
        glVertex3f(sp.x,sp.y,sp.z);
        glVertex3f(ep.x,ep.y,ep.z);
    glEnd();
}
```

4．绘制连续折线的函数——DrawPolyline()

函数 DrawPolyline ()在三维空间中绘制由数组 pt[]定义的连续折线，参数 size 是数组中的元素个数。函数的实现如下：

```
void COpenGLDC::DrawPolyline(const CPoint3D* pt,int size)
{
    glBegin(GL_LINE_STRIP);
    for(int i=0;i<size;i++)
        glVertex3f(pt[i].x,pt[i].y,pt[i].z);
    glEnd();
}
```

5．绘制一个三角面片的函数——DrawTriChip()

函数 DrawTriChip ()绘制一个三角面片。函数的实现如下：

```
void COpenGLDC::DrawTriChip(double n0,double n1,double n2,

                            double v00,double v01,double v02,
                            double v10,double v11,double v12,
                            double v20,double v21,double v22)
{
    glBegin(GL_TRIANGLES);
        glNormal3d(n0,n1,n2);
        glVertex3d(v00,v01,v02);
        glVertex3d(v10,v11,v12);
        glVertex3d(v20,v21,v22);
    glEnd();
}
```

6．绘制圆球的函数——DrawSphere()

函数 DrawSphere()绘制一个圆心位于 cen，半径为 r，中心轴为 vec 的圆球。函数中调用了 OpenGL 辅助库函数 auxSolidSphere()用于圆球绘制。由于函数 auxSolidSphere()是在坐标系

原点绘制一个以 z 轴为轴心的圆球，所以在绘制之前需要进行模型变换，将绘制原点平移至点 cen，并旋转轴心使之与矢量 vec 重合。函数的实现如下：

```
void COpenGLDC::DrawSphere(const CPoint3D& cen,double r,const CVector3D& vec)
{
    glPushMatrix();
    //平移变换至点 cen
    glTranslatef(cen.x,cen.y,cen.z);

    CVector3D vecNY(0,-1,0);
    CVector3D axis = vecNY*vec;
    //计算矢量间夹角，函数_AngleBetween 是库 GeomCalc.dll 的输出函数
    double ang = _AngleBetween(vecNY,vec);
    ang = ang*180/GL_PI;
    //旋转变换
    glRotatef(ang,axis.dx,axis.dy,axis.dz);

    auxSolidSphere(r);      //绘制圆球
    glPopMatrix();
}
```

7．绘制圆柱的函数——DrawCylinder()

函数 DrawCylinder()绘制一个底面圆心位于点 cen，半径为 r，中心轴为 vec 的圆柱。圆柱的高度为 vec 的模长。函数中调用 OpenGL 辅助库函数 auxSolidCylinder()用于圆柱绘制，在绘制之前也需要进行模型变换。函数的实现如下：

```
void COpenGLDC::DrawCylinder(const CPoint3D& cen,double r,const CVector3D& h)
{
    glPushMatrix();
    //平移变换至点 cen
    glTranslatef(cen.x,cen.y,cen.z);
    CVector3D vecNY(0,-1,0);
    CVector3D axis = vecNY*h;
    double ang = _AngleBetween(vecNY,h);
    ang = ang*180/GL_PI;
    //旋转变换
    glRotatef(ang,axis.dx,axis.dy,axis.dz);

    //绘制圆柱
    auxSolidCylinder(r,h.GetLength());
    glPopMatrix();
}
```

8．绘制圆环的函数——DrawTorus()

函数 DrawTorus ()绘制一个中心位于 cen，中心轴为 ax 的圆环。圆环半径和截面圆的半径分别为 r_out、r_in。函数中调用 OpenGL 辅助库函数 auxSolidTorus()用于圆环绘制，在绘制

之前需要进行模型变换。函数的实现如下：

```
void COpenGLDC::DrawTorus(const CPoint3D& cen,const CVector3D& ax,double r_in,
                          double r_out)
{
    glPushMatrix();
    //平移变换至点 cen
    glTranslatef(cen.x,cen.y,cen.z);

    CVector3D vecNY(0,-1,0);
    CVector3D axis = vecNY*ax;
    double ang = _AngleBetween(vecNY,ax);
    ang = ang*180/GL_PI;
    //旋转变换
    glRotatef(ang,axis.dx,axis.dy,axis.dz);

    auxSolidTorus(r_in,r_out);    //绘制圆环
    glPopMatrix();
}
```

9．绘制圆锥的函数——DrawCone()

函数 DrawCone ()绘制一个底面中心位于点 cen，底面圆半径为 r，中心轴为矢量 h 的圆锥。圆锥的高度为矢量 h 的模长。

```
void COpenGLDC::DrawCone(const CPoint3D& cen,double r,const CVector3D& h)
{

    glPushMatrix();
    //平移变换至点 cen
    glTranslatef(cen.x,cen.y,cen.z);

    CVector3D vecNY(0,-1,0);
    CVector3D axis = vecNY*h;
    double ang = _AngleBetween(vecNY,h);
    ang = ang*180/GL_PI;
    //旋转变换
    glRotatef(ang,axis.dx,axis.dy,axis.dz);

    //绘制圆锥面
    GLfloat angle,x,y;
    glBegin(GL_TRIANGLE_FAN);
        glVertex3f(0,0,h.GetLength());
        for(angle =0.0f;angle<(2.0f*GL_PI);angle += (GL_PI/8.0f))
        {
            x = r*sin(angle);
            y = r*cos(angle);
            glVertex2f(x,y);
        }
```

```
        glEnd();

        //绘制圆锥底面
        glBegin(GL_TRIANGLE_FAN);
            glVertex2f(0.0f,0.0f);
            for(angle =0.0f;angle<(2.0f*GL_PI);angle += (GL_PI/8.0f))
            {
                x = r*sin(angle);
                y = r*cos(angle);
                glVertex2f(x,y);
            }
        glEnd();
        glPopMatrix();
    }
```

5.3.5 选择模式

在 OpenGL 应用程序开发中，很多情况下需要用户使用鼠标或其他交互设备来拾取屏幕上显示的对象，并对选中的对象进行操作，如移动、修改、删除或改变属性等。在本书附带的示例程序 STLViewer 中，允许用户使用鼠标选择一个几何对象，并高亮度显示这个对象。由于绘制在屏幕上的对象是一系列几何变换后的结果，要在三维场景中由用户自己编写算法来实现拾取功能将会比较困难。OpenGL 提供了一种选择机制，可计算出在某个特定区域（通常是鼠标拾取的区域）内绘制的是哪个对象，并将有关信息返回给应用程序。用户使用选择机制，并结合鼠标或其他交互设备，可以实现用户与图形之间的交互。

和 OpenGL 的绘图模式一样，选择模式不是专门支持 Windows 的。要在 Windows 应用程序中使用 OpenGL 的选择模式，需要对选择模式下的操作进行包装。选择是一个过程，从开始到返回要涉及到多个环节。在 COpenGLDC 中，为了支持这个选择过程，开发了以下函数。

（1）开始选择过程：

```
void        BeginSelection(int xPos,int yPos);
```

（2）结束选择过程，返回选择记录：

```
int         EndSelection(UINT* items);
```

（3）当前是否是选择模式：

```
BOOL        IsSelectionMode();
```

（4）选择模式下的对象名称操作：

```
void        InitNames();
void        LoadName(UINT name);
void        PushName(UINT name);
void        PopName();
```

由于选择模式的使用以及 Windows 环境下整个选择过程的实现较为复杂，在本书的第 9 章将专门讲述 OpenGL 选择模式和它在 Windows 应用程序中的实现，并将对以上函数的设计

与使用作详细介绍。

5.4 增加类 CGLView 中的功能

鉴于在类 COpenGLDC 和 GCamera 中对相关功能做了增强,特别是增加了对场景变换(场景的大小和观察角度）的操作。在三维 CAD 软件中，对模型显示的放大、缩小，以及对观察视图（观察角度的定义）的切换等是软件最常用的操作功能之一。可以在类 CGLView 中调用 COpenGLDC、GCamera 的相关功能，以实现对场景变换的操作。但由于这些功能的常用性，也可以将这些调用直接写成 CGLView 的成员函数封装在 CGLView 类中。在 CGLView 上派生的视图类可以直接调用这些函数，而不必去关心它是如何通过对 COpenGLDC、GCamera 的调用来实现的。

在库 glContext.dll 中，对 CGLView 的修改如下：

```
class AFX_EXT_CLASS CGLView : public CView
{
protected:
	COpenGLDC*	m_pGLDC;

protected: // create from serialization only
	CGLView();
	DECLARE_DYNCREATE(CGLView)

// Attributes
public:
	virtual void RenderScene(COpenGLDC* pDC);

	// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CGLView)
	public:
	virtual void OnDraw(CDC* pDC);	// overridden to draw this view
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CGLView();

protected:

// Generated message map functions
protected:
```

```
    //{{AFX_MSG(CGLView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnDestroy();
    afx_msg void OnSize(UINT nType, int cx, int cy);
    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()

    //获取当前模型的最大包容盒
    virtual BOOL GetBox(double& x0, double& y0, double& z0, double& x1, double& y1, double& z1);

    //缩放场景
    void  Zoom(double dScale);

    //计算一个合适的缩放比，以将模型全部显示在场景中
    void  ZoomAll();

    //使用典型视角来观察模型
    void  OnViewType(UINT type);

    //按当前场景尺寸的百分比移动场景，参数 dpx、dpy 的范围是 0～1
    void  MoveView(double dpx,double dpy);
};
```

下面对这几个函数进行介绍。

1．成员函数 GetBox()

新增加的成员函数 GetBox()**用于获取当前场景中所显示模型的包容盒信息。**

获取模型最大包容盒的目的在于粗略估计模型所占空间的位置和大小，以便于场景操作。例如，当需要在场景中将全部 CAD **模型都显示出来时，需要知道该模型的位置和大小，即它的最大包容盒，进而才能根据这个包容盒的数据计算出相应的视景体空间，以将整个模型容纳在其中。**

通常，CAD **的模型数据存放在文档类中。不同的应用程序获取模型的最大包容盒的方法也不一样。**CGLView **作为一个可重用的视图基类，不可能在函数** GetBox()**中具体实现这个功能，而是将函数** GetBox()**设计为一个虚拟函数，作为一个统一的接口。在派生类中若需要实现这个功能，必须实现自己的** GetBox()**函数。**

函数 GetBox()**实现如下：**

```
BOOL CGLView::GetBox(double& x0, double& y0, double& z0,
        double& x1, double& y1, double& z1)
{
    return FALSE;
}
```

若在派生类中没有实现自己的 GetBox()**函数，则调用** GetBox()**时返回** FALSE**，表示无法获得最大包容盒。**

2．成员函数 Zoom()

成员函数 Zoom()**用于实现对场景空间的放大和缩小。它对应于照相机类的函数** GCamera::zoom()**，并在改变了场景空间的尺寸后重新刷新窗口。**

```
void CGLView::Zoom(double dScale)
{
    m_pGLDC->m_Camera.zoom(dScale);
    Invalidate();
}
```

缩小场景空间,对应的效果是场景中模型的显示尺寸变大,这类似于 CAD **软件中的** Zoom In **功能。相反，放大场景空间，对应的效果是场景中模型的显示尺寸变小，这类似于** CAD **软件中的** Zoom Out **功能。**

例如：

Zoom(0.9)　**缩小场景空间，放大模型显示**

Zoom(1.1)　**放大场景空间，缩小模型显示**

3．**成员函数** ZoomAll()

函数 ZoomAll()**用于将模型全部显示在场景空间中，这样在屏幕上可以观察整个模型，它对应于照相机类的函数** GCamera::zoom_all()**，以调整场景空间大小，使之能够包容整个模型，并在改变了场景尺寸后重新刷新窗口。**

函数 ZoomAll()**首先需要获得模型的包容盒尺寸。所以，要实现这个全局缩放功能，必须在** CGLView **派生的类中实现自己** GetBox()**函数，并返回** TRUE**。**

```
void CGLView::ZoomAll()
{
    double x0,y0,z0,x1,y1,z1;
    if(GetBox(x0,y0,z0,x1,y1,z1)){    //获取模型的包容盒
        m_pGLDC->m_Camera.zoom_all(x0,y0,z0,x1,y1,z1);    //全局显示模型
        Invalidate();    //刷新视图
    }
}
```

4．**成员函数** OnViewType()

函数 OnViewType()**用于选择一个典型的观察视图（观察角度）来观察当前模型，它对应于照相机类中的函数** GCamera::set_view_type()**，并在调用函数** GCamera::set_view_type()**改变了观察角度之后，重新刷新窗口。**

```
void CGLView::OnViewType(UINT type)
{
    ASSERT(type >= VIEW_FRONT && type <= VIEW_NW_ISOMETRIC);
    m_pGLDC->m_Camera.set_view_type(type);  //切换对场景的观察角度
    Invalidate();    //刷新视图
}
```

输入变量 type **的类型在** camera.h **中作了定义，它必须是** VIEW_FRONT～VIEW_NW_ISOMETRIC **之间的一个。**

```
#define    VIEW_FRONT              0
```

```
#define    VIEW_BACK              1
#define    VIEW_TOP               2
#define    VIEW_BOTTOM            3
#define    VIEW_RIGHT             4
#define    VIEW_LEFT              5
#define    VIEW_SW_ISOMETRIC      6
#define    VIEW_SE_ISOMETRIC      7
#define    VIEW_NE_ISOMETRIC      8
#define    VIEW_NW_ISOMETRIC      9
```

5．**成员函数** MoveView()

函数 MoveView()**用于平移当前场景，如上下左右移动场景以观察模型。它对应于照相机类的函数** GCamera::move_view()**，实际上是平移视点的位置，并在改变了视点位置后重新刷新窗口。函数按照当前场景的百分比来移动场景，**dpx、dpy **分别是当前场景在** x、y **方向上的百分比。在应用程序中，可将按下键盘的上下左右键对应于此函数，以实现通过键盘平移场景。**

```
void CGLView::MoveView(double dpx,double dpy)
{
    m_pGLDC->m_Camera.move_view(dpx,dpy);
    Invalidate();        //刷新视图
}
```

5.5 glContext 类的输出和调用

以上，我们创建了用于支持 OpenGL Windows **程序的动态库工程** glContext**。编译连接整个工程后，将生成以下库文件（默认设置时，文件位于** Debug **目录下）：**

glContext.lib glContext **的连接库**

glContext.dll glContext **的动态链接库**

库 glContext.dll **中输出了三个类，供其他应用程序调用，即：**

GCamera

COpenGLDC

CGLView

要使用这些输出类，需要在相关文件中插入说明这些类的头文件，分别是：

Camera.h **说明类** GCamera

OpenGLDC.h **说明类** COpenGLDC, CGLView

5.6 源程序清单

下面给出实现类 GCamera、COpenGLDC、CGLView **的源程序清单。**

5.6.1 文件 Camera.h（类 GCamera）

```
/*************************************************************************
```

```
* This head file is for 3D scene script. We will define a scene class      *
* with some light and camera and models.                                    *
*****************************************************************************/
#ifndef _CAMERA_H__
#define _CAMERA_H__

#include "..\inc\GeomCalc\cadbase.h"

#define     VIEW_FRONT          0
#define     VIEW_BACK           1
#define     VIEW_TOP            2
#define     VIEW_BOTTOM         3
#define     VIEW_RIGHT          4
#define     VIEW_LEFT           5
#define     VIEW_SW_ISOMETRIC   6
#define     VIEW_SE_ISOMETRIC   7
#define     VIEW_NE_ISOMETRIC   8
#define     VIEW_NW_ISOMETRIC   9

#define     ZOOM_ALL            9
#define     ZOOM_IN         10
#define     ZOOM_OUT            11

class AFX_EXT_CLASS GCamera
{
protected:
    //eye coordinator
    CPoint3D        m_eye;
    CPoint3D        m_ref;
    CVector3D       m_vecUp;
    //viewing volume
    double          m_far, m_near;
    double          m_width,m_height;

    //viewport
    double          m_screen[2];

public:
    GCamera();
    ~GCamera();

    //initailizing
    void init();

    void projection();
    void selection(int xPos,int yPos);
```

```
        //zooming
        void zoom(double scale);
        void zoom_all(double x0,double y0,double z0,double x1,double y1,double z1);

        //switch into a classical view
        void set_view_type(int type);

        void move_view(double dpx, double dpy);

        //set viewport acoording to window
        void set_screen( int x, int y);

        //set eye coordinate
        void set_eye(double eye_x,double eye_y,double eye_z);
        void set_ref(double ref_x,double ref_y,double ref_z);
        void set_vecUp(double up_dx,double up_dy,double up_dz);

        //set viewing volume
        void set_view_rect(double width,double height);
        void get_view_rect(double& width,double& height);
protected:
      void update_upVec();
};
#endif
```

5.6.2 文件 Camera.cpp（类 GCamera）

```
#include "stdafx.h"
#include <gl/gl.h>
#include <gl/glu.h>
#include "camera.h"

//-------------------------------------------------------------
// For Camera class
//-------------------------------------------------------------
GCamera::GCamera(void)
{
}

GCamera::~GCamera()
{
}

void GCamera::projection()
{
        //switch to projection
        glMatrixMode(GL_PROJECTION);
```

```
        glLoadIdentity();

        glRenderMode(GL_RENDER);

        //apply projective matrix
        double left     =   - m_width/2.0;
        double right    =   m_width/2.0;
        double bottom   =   - m_height/2.0;
        double top      =   m_height/2.0;

        glOrtho(left,right,bottom,top,m_near,m_far);

        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity( );
        gluLookAt(m_eye.x,m_eye.y,m_eye.z,m_ref.x,m_ref.y,m_ref.z,m_vecUp.dx,m_vecUp.dy,
        m_vecUp.dz);
}

void GCamera::selection(int xPos,int yPos)
{
        GLintvp[4];
        glGetIntegerv(GL_VIEWPORT,vp);

        glMatrixMode(GL_PROJECTION);
        glLoadIdentity();

        glRenderMode(GL_SELECT);
        gluPickMatrix(xPos,vp[3]-yPos, 1, 1, vp );

        //apply projective matrix
        double left     =   - m_width/2.0;
        double right    =   m_width/2.0;
        double bottom   =   - m_height/2.0;
        double top      =   m_height/2.0;

        glOrtho(left,right,bottom,top,m_near,m_far);

        glMatrixMode( GL_MODELVIEW );
        glLoadIdentity( );
        gluLookAt(m_eye.x,m_eye.y,m_eye.z,m_ref.x,m_ref.y,m_ref.z,
                m_vecUp.dx, m_vecUp.dy, m_vecUp.dz);
}

void GCamera::init()
{
        m_eye = CPoint3D(0,0,1000);
        m_ref = CPoint3D(0,0,0);
```

```
	m_far = 10000;
	m_near= 1;

	m_width = 2400.0;
	m_height = 2400.0;

	m_vecUp = CVector3D(0,1,0);

	m_screen[0] = 400;
	m_screen[1] = 400;
}

void GCamera::set_screen( int x, int y)
{
	glViewport(0,0,x,y);
	if(y==0) y=1;
	double ratio = (double)x/(double)y;
	m_width *= (double)x/m_screen[0];
	m_height *= (double)y/m_screen[1];
	m_width =    m_height*ratio;
	m_screen[0] = x;
	m_screen[1] = y;
}
void GCamera::set_eye(double eye_x,double eye_y,double eye_z)
{
	m_eye.x = eye_x;
	m_eye.y = eye_y;
	m_eye.z = eye_z;
}

void GCamera::set_ref(double ref_x,double ref_y,double ref_z)
{
	m_ref.x = ref_x;
	m_ref.y = ref_y;
	m_ref.z = ref_z;
}

void GCamera::set_vecUp(double up_dx,double up_dy,double up_dz)
{
	m_vecUp.dx = up_dx;
	m_vecUp.dy = up_dy;
	m_vecUp.dz = up_dz;
}

void GCamera::set_view_rect(double width,double height)
{
	m_width = width;
```

```
	m_height = height;
	double aspect = m_screen[0]/m_screen[1];
	m_width =    m_height*aspect;
}

void GCamera::get_view_rect(double& width,double& height)
{
	width = m_width;
	height = m_height;
}

void GCamera::zoom(double scale)
{
	ASSERT(scale > 0.0);
	m_width *= scale;
	m_height *= scale;
}

void GCamera::zoom_all(double x0,double y0,double z0,double x1,double y1,double z1)
{
	double width,height;
	double	xl, yl, zl;
	xl = x1-x0;
	yl = y1-y0;
	zl = z1-z0;

	width = max(max(xl,yl),zl);
	height= max(max(xl,yl),zl);

	set_view_rect(width,height);
	CVector3D vec = m_eye - m_ref;
	m_ref.x = (x0+x1)/2;
	m_ref.y = (y0+y1)/2;
	m_ref.z = (z0+z1)/2;
	m_eye = m_ref + vec;
}

void GCamera::set_view_type( int type )
{
	double r;
	CVector3D vec;

	vec = m_ref - m_eye;
	r = vec.GetLength();

	if(IS_ZERO(r)) r = 50.0;
	if( r > 10000)    r = 10000;
```

```
	switch(type){
	case VIEW_FRONT:
		m_eye = m_ref + CVector3D(0,-r,0);
		m_vecUp = CVector3D(0,0,1);
		break;
	case VIEW_BACK:
		m_eye = m_ref + CVector3D(0,r,0);
		m_vecUp = CVector3D(0,0,1);
		break;
	case VIEW_TOP:
		m_eye = m_ref + CVector3D(0,0,r);
		m_vecUp = CVector3D(0,1,0);
		break;
	case VIEW_BOTTOM:
		m_eye = m_ref + CVector3D(0,0,-r);
		m_vecUp = CVector3D(0,1,0);
		break;
	case VIEW_RIGHT:
		m_eye = m_ref + CVector3D(r,0,0);
		m_vecUp = CVector3D(0,0,1);
		break;
	case VIEW_LEFT:
		m_eye = m_ref + CVector3D(-r,0,0);
		m_vecUp = CVector3D(0,0,1);
		break;
	case VIEW_SW_ISOMETRIC:
		m_eye = m_ref + CVector3D(-1,-1,1).GetNormal()*r;
		update_upVec();
		break;
	case VIEW_SE_ISOMETRIC:
		m_eye = m_ref + CVector3D(1,-1,1).GetNormal()*r;
		update_upVec();
		break;
	case VIEW_NE_ISOMETRIC:
		m_eye = m_ref + CVector3D(1,1,1).GetNormal()*r;
		update_upVec();
		break;
	case VIEW_NW_ISOMETRIC:
		m_eye = m_ref + CVector3D(-1,1,1).GetNormal()*r;
		update_upVec();
		break;
	}
}

void GCamera::move_view(double dpx, double dpy)
{
```

```
	CVector3D vec;
	CVector3D xUp, yUp;

	vec = m_ref - m_eye;
	vec.Normalize();
	xUp = vec*m_vecUp;
	yUp = xUp*vec;

	m_eye -= xUp*m_width*dpx + yUp*m_height*dpy;
	m_ref -= xUp*m_width*dpx + yUp*m_height*dpy;
}

void GCamera::update_upVec()
{
	CVector3D vec = m_ref - m_eye;
	CVector3D zVec(0,0,1);
	CVector3D vec0;

	vec.Normalize();
	vec0 = vec*zVec;
	m_vecUp = vec0*vec;
}

//////////////////
```

5.6.3 文件 OpenGLDC.h（类 COpenGLDC、CGLView）

```
#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
#include "gl/gl.h"
#include "gl/glu.h"
#include "gl/glaux.h"

#include "camera.h"
#include "..\inc\GeomCalc\cadbase.h"

#define BUFFER_LENGTH 64

class AFX_EXT_CLASS COpenGLDC
{
public:
	COpenGLDC(HWND hWnd);
	virtual ~COpenGLDC();

private:
	HWND	m_hWnd;
```

```
    HGLRC    m_hRC;
    HDC      m_hDC;

    COLORREF    m_clrBk;            //Background Color
    COLORREF    m_clr;              //Polygon Color for unshading
    COLORREF    m_clrHighlight;     //for highlight using
    COLORREF    m_clrMaterial;      //for normal rendering
    BOOL        m_bShading;         //use material property
    GLfloat     m_vecLight[3];      //lighting direction
    //selection
    BOOL    m_bSelectionMode;
    GLuint  m_selectBuff[BUFFER_LENGTH];
public:
    GCamera     m_Camera;

protected:
    void ClearBkground();
    void OnShading();

public:
    //initialize
    BOOL InitDC();
    void GLResize(int cx,int cy);
    void GLSetupRC();

    //uMode :zero for normal rendering. non-zero for selection
    void Ready();
    void Finish();

    ////////////////////// light & color //////////////////////
    void Shading(BOOL bShading);
    BOOL IsShading();

    void Lighting(BOOL bLighting);
    BOOL IsLighting();

    //Light direction
    void SetLightDirection(float dx,float dy,float dz);
    void GetLightDirection(float& dx,float& dy,float& dz);
    //material
    void SetMaterialColor(COLORREF clr);
    void GetMaterialColor(COLORREF& clr);

    //back ground
    void SetBkColor(COLORREF rgb);
    void GetBkColor(COLORREF& rgb);
```

```
        //frame material
        void SetColor(COLORREF rgb);
        void GetColor(COLORREF& rgb);

        //high light setting
        void SetHighlightColor(COLORREF clr);
        void GetHighlightColor(COLORREF& clr);
        void Highlight(BOOL bLight = TRUE);

        ////////////////// Drawing ////////////////////
        void DrawPoint(const CPoint3D&);

        //drawing curves
        void DrawCoord();

        void DrawLine(const CPoint3D& sp,const CPoint3D& ep);
        void DrawPolyline(const CPoint3D* pt,int size);

        //drawing surface
        void DrawTriChip(double n0,double n1,double n2,double v00,double v01,double v02,
                         double v10,double v11,double v12,double v20,double v21,double v22);

        //drawing solid entities
        void DrawSphere(const CPoint3D& cen,double r,const CVector3D& vec);
        void DrawCylinder(const CPoint3D& cen,double r,const CVector3D& h);
        void DrawCone(const CPoint3D& cen,double r,const CVector3D& h);
        void DrawTorus(const CPoint3D& cen,const CVector3D& ax,double r_in,double r_out);

        //selection Mode
        void  BeginSelection(int xPos,int yPos);
        int           EndSelection(UINT* items);
        BOOL      IsSelectionMode();

        void  InitNames();
        void  LoadName(UINT name);
        void  PushName(UINT name);
        void  PopName();

};
class AFX_EXT_CLASS CGLView : public CView
{
protected:
        COpenGLDC*    m_pGLDC;

protected: // create from serialization only
        CGLView();
        DECLARE_DYNCREATE(CGLView)
```

```
// Attributes
public:
	virtual void RenderScene(COpenGLDC* pDC);

	// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CGLView)
	public:
	virtual void OnDraw(CDC* pDC);    // overridden to draw this view
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
	//}}AFX_VIRTUAL
// Implementation
public:
	virtual ~CGLView();

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CGLView)
	afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
	afx_msg void OnDestroy();
	afx_msg void OnSize(UINT nType, int cx, int cy);
	afx_msg BOOL OnEraseBkgnd(CDC* pDC);
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()

	virtual BOOL GetBox(double& x0,double& y0,double& z0,double& x1,double& y1,
		double& z1);
	void  Zoom(double dScale);
	void  ZoomAll();
	void  OnViewType(UINT type);
	void  MoveView(double dpx,double dpy);
};
#endif
// !defined(AFX_OPENGLDC_H__30E692A3_4129_11D4_B1EE_0010B539EBC0__INCLUDED_)
```

5.6.4 文件 OpenGLDC.cpp（类 OpenGLDC）

```
// OpenGLDC.cpp: implementation of the COpenGLDC class.
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
```

```
#include "OpenGLDC.h"
#define GL_PI 3.1415f

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

COpenGLDC::COpenGLDC(HWND hWnd):m_hWnd(hWnd)
{
	m_bSelectionMode = FALSE;
}

COpenGLDC::~COpenGLDC()
{
}
BOOL COpenGLDC::InitDC()
{
	if (m_hWnd == NULL) return FALSE;

	m_Camera.init();

	m_hDC = ::GetDC(m_hWnd);		// Get the Device context

	int pixelformat;
	PIXELFORMATDESCRIPTOR pfdWnd =
	{
		sizeof(PIXELFORMATDESCRIPTOR), // Structure size
		1,				// Structure version number
		PFD_DRAW_TO_WINDOW |		// Property flags
		PFD_SUPPORT_OPENGL |
		PFD_DOUBLEBUFFER,
		PFD_TYPE_RGBA,
		24,				// 24-bit color
		0, 0, 0, 0, 0, 0,		// Not concerned with these
		0, 0, 0, 0, 0, 0, 0,		// No alpha or accum buffer
		32,				// 32-bit depth buffer
		0, 0,				// No stencil or aux buffer
		PFD_MAIN_PLANE,			// Main layer type
		0,				// Reserved
		0, 0, 0				// Unsupported
	};

	if ( (pixelformat = ChoosePixelFormat(m_hDC, &pfdWnd)) == 0 )
	{
		AfxMessageBox("ChoosePixelFormat to wnd failed");
		return FALSE;
	}
```

```
        if (SetPixelFormat(m_hDC, pixelformat, &pfdWnd) == FALSE)
            AfxMessageBox("SetPixelFormat failed");

        m_hRC=wglCreateContext(m_hDC);

        VERIFY(wglMakeCurrent(m_hDC,m_hRC));
        GLSetupRC();
        wglMakeCurrent(NULL,NULL);
        return m_hRC!=0;
}

void COpenGLDC::GLResize(int w,int h)
{
        wglMakeCurrent(m_hDC,m_hRC);

        // Prevent a divide by zero
        if(h == 0) h = 1;
        if(w == 0) w = 1;
        m_Camera.set_screen(w,h);
}

void COpenGLDC::GLSetupRC()
{
        //initialize color and rendering
        m_bShading = TRUE;

        //bright white light - full intensity RGB values
        GLfloat lightAmbient[] = {0.75f,0.75f,0.75f,1.0f};
        GLfloat lightDiffuse[] = {1.0f,1.0f,1.0f,1.0f};

        glEnable(GL_DEPTH_TEST);            //Hidden surface removal
        glEnable(GL_CULL_FACE);             //Do not calculate inside of object
        glFrontFace(GL_CCW);                //counter clock-wise polygons face out

        glEnable(GL_LIGHTING);              //enable lighting

        //setup and enable light 0
        glLightfv(GL_LIGHT0,GL_AMBIENT,lightAmbient);
        glLightfv(GL_LIGHT0,GL_DIFFUSE,lightDiffuse);
        SetLightDirection(1,1,1);
        glEnable(GL_LIGHT0);

        //Initialize Material Color to Gray
        SetBkColor(RGB(0,0,0));             //black background
        SetMaterialColor(RGB(225,175,22));  //golden material color
        SetColor(RGB(255,255,255));         //white frame color
```

```
	SetHighlightColor(RGB(255,0,0));	//red highlight color

	//Point Size
	glPointSize(3.0);
}

void COpenGLDC::Ready()
{
	wglMakeCurrent(m_hDC,m_hRC);
	ClearBkground();
	OnShading();
	m_Camera.projection();
}

void COpenGLDC::Finish()
{
	glFlush();
	SwapBuffers(m_hDC);
	wglMakeCurrent(m_hDC,NULL);
}

//////////////////LIGHT && MATERIALS SETTING//////////
void COpenGLDC::ClearBkground()
{
	GLclampf   r,g,b;
	r = (GLclampf)GetRValue(m_clrBk)/255.0;
	g = (GLclampf)GetGValue(m_clrBk)/255.0;
	b = (GLclampf)GetBValue(m_clrBk)/255.0;
	glClearColor(r,g,b,0.0f);
	glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
}

//setting model
void COpenGLDC::OnShading()
{
	if(m_bShading){
		glEnable( GL_LIGHTING );
		glEnable(GL_LIGHT0);
		glPolygonMode(GL_FRONT_AND_BACK,GL_FILL);
	}
	else{
		glDisable( GL_LIGHTING );
		glPolygonMode(GL_FRONT_AND_BACK,GL_LINE);
	}
}

void COpenGLDC::Shading(BOOL bShading)
```

```
{
	m_bShading = bShading;
}

BOOL COpenGLDC::IsShading()
{
	return m_bShading;
}

void COpenGLDC::Lighting(BOOL bLighting)
{
	if(bLighting)
		glEnable( GL_LIGHTING );
	else
		glDisable( GL_LIGHTING );
}

BOOL COpenGLDC::IsLighting()
{
	GLboolean bLighting;
	glGetBooleanv(GL_LIGHTING,&bLighting);
	return bLighting;
}

void COpenGLDC::SetLightDirection(float dx,float dy,float dz)
{
	m_vecLight[0] = dx;
	m_vecLight[1] = dy;
	m_vecLight[2] = dz;
	GLfloat lightPos[] = { dx, dy, dz, 0.0f };
	glLightfv(GL_LIGHT0,GL_POSITION,lightPos);
}

void COpenGLDC::GetLightDirection(float& dx,float& dy,float& dz)
{
	dx = m_vecLight[0];
	dy = m_vecLight[1];
	dz = m_vecLight[2];
}
//rendering color
void COpenGLDC::SetMaterialColor(COLORREF clr)
{
	m_clrMaterial = clr;
	BYTE r,g,b;
	r = GetRValue(clr);
	g = GetGValue(clr);
	b = GetBValue(clr);
```

```
	GLfloat mat_amb_diff[] = {(GLfloat)r/255,(GLfloat)g/255,(GLfloat)b/255,1.0};
	glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_amb_diff);
}

void COpenGLDC::GetMaterialColor(COLORREF& clr)
{
	clr = m_clrMaterial;
}

void COpenGLDC::SetBkColor(COLORREF clr)
{
	m_clrBk = clr;
}

void COpenGLDC::GetBkColor(COLORREF& clr)
{
	clr = m_clrBk;
}

void COpenGLDC::SetColor(COLORREF clr)
{
	m_clr = clr;
	BYTE r,g,b;
	r = GetRValue(clr);
	g = GetGValue(clr);
	b = GetBValue(clr);
	glColor3ub(r,g,b);
}

void COpenGLDC::GetColor(COLORREF& clr)
{
	clr =    m_clr;
}

void COpenGLDC::SetHighlightColor(COLORREF clr)
{
	m_clrHighlight = clr;
}

void COpenGLDC::GetHighlightColor(COLORREF& clr)
{
	clr = m_clrHighlight;
}

void COpenGLDC::Highlight(BOOL bHighlight)
{
	BYTE r,g,b;
```

```
	if(bHighlight){
		r = GetRValue(m_clrHighlight);
		g = GetGValue(m_clrHighlight);
		b = GetBValue(m_clrHighlight);
	}
	else{
		r = GetRValue(m_clrMaterial);
		g = GetGValue(m_clrMaterial);
		b = GetBValue(m_clrMaterial);
	}
	GLfloat mat_amb_diff[] = {(GLfloat)r/255,(GLfloat)g/255,(GLfloat)b/255,1.0};
	glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,mat_amb_diff);
}

////////////////////// drawing functions //////////////////////
//draw point
void COpenGLDC::DrawPoint(const CPoint3D& pt)
{
	glBegin(GL_POINTS);
		glVertex3f(pt.x,pt.y,pt.z);
	glEnd();
}

void COpenGLDC::DrawLine(const CPoint3D& sp,const CPoint3D& ep)
{
	glBegin(GL_LINES);
		glVertex3f(sp.x,sp.y,sp.z);
		glVertex3f(ep.x,ep.y,ep.z);
	glEnd();
}

void COpenGLDC::DrawPolyline(const CPoint3D* pt,int size)
{
	glBegin(GL_LINE_STRIP);
	for(int i=0;i<size;i++)
		glVertex3f(pt[i].x,pt[i].y,pt[i].z);
	glEnd();
}

void COpenGLDC::DrawTriChip(double n0,double n1,double n2,
                            double v00,double v01,double v02,
                            double v10,double v11,double v12,
                            double v20,double v21,double v22)
{
	glBegin(GL_TRIANGLES);
		glNormal3d(n0,n1,n2);
		glVertex3d(v00,v01,v02);
```

```
            glVertex3d(v10,v11,v12);
            glVertex3d(v20,v21,v22);
        glEnd();
}

//Draw 3D Solid
void COpenGLDC::DrawSphere(const CPoint3D& cen,double r,const CVector3D& vec)
{
        glPushMatrix();
        glTranslatef(cen.x,cen.y,cen.z);

        CVector3D vecNY(0,-1,0);
        CVector3D axis = vecNY*vec;
        double ang = _AngleBetween(vecNY,vec);
        ang = ang*180/GL_PI;
        glRotatef(ang,axis.dx,axis.dy,axis.dz);

        auxSolidSphere(r);
        glPopMatrix();
}

void COpenGLDC::DrawCylinder(const CPoint3D& cen,double r,const CVector3D& h)
{
        glPushMatrix();

        glTranslatef(cen.x,cen.y,cen.z);

        CVector3D vecNY(0,-1,0);
        CVector3D axis = vecNY*h;
        double ang = _AngleBetween(vecNY,h);
        ang = ang*180/GL_PI;
        glRotatef(ang,axis.dx,axis.dy,axis.dz);

        auxSolidCylinder(r,h.GetLength());
        glPopMatrix();
}

void COpenGLDC::DrawTorus(const CPoint3D& cen,const CVector3D& ax,double r_in,
double r_out)
{
        glPushMatrix();

        glTranslatef(cen.x,cen.y,cen.z);

        CVector3D vecNY(0,-1,0);
        CVector3D axis = vecNY*ax;
        double ang = _AngleBetween(vecNY,ax);
```

```
	ang = ang*180/GL_PI;
	glRotatef(ang,axis.dx,axis.dy,axis.dz);

	auxSolidTorus(r_in,r_out);
	glPopMatrix();
}

void COpenGLDC::DrawCone(const CPoint3D& cen,double r,const CVector3D& h)
{
	glPushMatrix();

	glTranslatef(cen.x,cen.y,cen.z);

	CVector3D vecNY(0,-1,0);
	CVector3D axis = vecNY*h;
	double ang = _AngleBetween(vecNY,h);
	ang = ang*180/GL_PI;

	glRotatef(ang,axis.dx,axis.dy,axis.dz);

	GLfloat angle,x,y;
	glBegin(GL_TRIANGLE_FAN);
		glVertex3f(0,0,h.GetLength());
		for(angle =0.0f;angle<(2.0f*GL_PI);angle += (GL_PI/8.0f))
		{
			x = r*sin(angle);
			y = r*cos(angle);
			glVertex2f(x,y);
		}
	glEnd();

	// Begin a new triangle fan to cover the bottom
	glBegin(GL_TRIANGLE_FAN);
		glVertex2f(0.0f,0.0f);
		for(angle =0.0f;angle<(2.0f*GL_PI);angle += (GL_PI/8.0f))
		{
			x = r*sin(angle);
			y = r*cos(angle);
			glVertex2f(x,y);
		}
	glEnd();
	glPopMatrix();
}

////////////////////////////////////////
void COpenGLDC::DrawCoord()
{
```

```
	BOOL bLighting = IsLighting();
	Lighting(FALSE);

	double width,height;
	m_Camera.get_view_rect(width,height);
	double len = min(width,height);
	len *= 0.2;
	CPoint3D cPt,xPt,yPt,zPt;
	xPt.x = yPt.y = zPt.z = len;

	COLORREF old_clr;
	GetColor(old_clr);

	//axis-x: red
	SetColor(RGB(255,0,0));
	DrawLine(cPt,xPt);

	//axis-y: green
	SetColor(RGB(0,255,0));
	DrawLine(cPt,yPt);

	//axis-z: blue
	SetColor(RGB(0,0,255));
	DrawLine(cPt,zPt);

	Lighting(bLighting);
	SetColor(old_clr);
}

//////////////////////////////////////////////////
void COpenGLDC::BeginSelection(int xPos,int yPos)
{
	m_bSelectionMode = TRUE;
	wglMakeCurrent(m_hDC,m_hRC);

	GLintviewport[4];

	//set up selection buffer
	glSelectBuffer(BUFFER_LENGTH,m_selectBuff);
	//switch to projection and save the matrix
	m_Camera.selection(xPos,yPos);

	glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
	InitNames();
}

int	COpenGLDC::EndSelection(UINT* items)
```

```
{
	m_bSelectionMode = FALSE;
	int hits = glRenderMode(GL_RENDER);
	for(int i=0;i<hits;i++){
		items[i] = m_selectBuff[i*4+3];
	}
	wglMakeCurrent(m_hDC,NULL);
	return hits;
}

BOOL COpenGLDC::IsSelectionMode()
{
	return m_bSelectionMode;
}

void COpenGLDC::InitNames()
{
	glInitNames();
	glPushName(0);
}

void COpenGLDC::LoadName(UINT name)
{
	glLoadName(name);
}

void COpenGLDC::PushName(UINT name)
{
	glPushName(name);
}

void COpenGLDC::PopName()
{
	glPopName();
}
```

5.6.5 文件 GLView.cpp（类 CGLView）

```
// GLView.cpp : implementation of the CGLView class
#include "stdafx.h"
#include "OpenGLDC.h"
#include "resource.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
```

```
#endif

/////////////////////////////////////////////////////////////////////////////
// CGLView

IMPLEMENT_DYNCREATE(CGLView, CView)

BEGIN_MESSAGE_MAP(CGLView, CView)
	//{{AFX_MSG_MAP(CGLView)
	ON_WM_CREATE()
	ON_WM_DESTROY()
	ON_WM_SIZE()
	ON_WM_ERASEBKGND()
	//}}AFX_MSG_MAP
	// Standard printing commands
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CGLView construction/destruction

CGLView::CGLView()
{
	m_pGLDC = NULL;
}
CGLView::~CGLView()
{
}

/////////////////////////////////////////////////////////////////////////////
// CGLView drawing

void CGLView::OnDraw(CDC* pDC)
{
	if(m_pGLDC){
		m_pGLDC->Ready();
		RenderScene(m_pGLDC);
		m_pGLDC->Finish();
	}
}
BOOL CGLView::PreCreateWindow(CREATESTRUCT& cs)
{

	//	Add Window style required for OpenGL before window is created
	cs.style |= WS_CLIPSIBLINGS|WS_CLIPCHILDREN;

	return CView::PreCreateWindow(cs);
}
```

```
////////////////////////////////////////////////////////////////////////////
// CGLView message handlers
void CGLView::RenderScene(COpenGLDC* pDC)
{
	pDC->DrawCoord();
}

int CGLView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
	if (CView::OnCreate(lpCreateStruct) == -1)
		return -1;

	m_pGLDC = new COpenGLDC(this->GetSafeHwnd());
	m_pGLDC->InitDC();
	return 0;
}

void CGLView::OnDestroy()
{
	CView::OnDestroy();
	if(m_pGLDC)	delete m_pGLDC;
}
void CGLView::OnSize(UINT nType, int cx, int cy)
{
	CView::OnSize(nType, cx, cy);
	if(m_pGLDC)
		m_pGLDC->GLResize(cx,cy);
}

BOOL CGLView::OnEraseBkgnd(CDC* pDC)
{
	//return CView::OnEraseBkgnd(pDC);
	return TRUE;
}

void CGLView::OnViewType(UINT type)
{
	ASSERT(type >= VIEW_FRONT && type <= VIEW_NW_ISOMETRIC);
	m_pGLDC->m_Camera.set_view_type(type);
	Invalidate();
}

BOOL CGLView::GetBox(double& x0,double& y0,double& z0,double& x1,double& y1,double& z1)
{
	return FALSE;
}
```

```
void CGLView::ZoomAll()
{
	double x0,y0,z0,x1,y1,z1;
	if(GetBox(x0,y0,z0,x1,y1,z1)){
		m_pGLDC->m_Camera.zoom_all(x0,y0,z0,x1,y1,z1);
		Invalidate();
	}
}

void CGLView::Zoom(double dScale)
{
	m_pGLDC->m_Camera.zoom(dScale);
	Invalidate();
}

void CGLView::MoveView(double dpx,double dpy)
{
	m_pGLDC->m_Camera.move_view(dpx,dpy);
	Invalidate();
}
```

本章相关程序

- ch5\glContext：**开发的基于** OpenGL **的** CAD **图形工具库的工程。**
- ch5\inc：glContext **需要调用的头文件（需要使用** GeomCalc.dll **的输出类）。**
- ch5\lib：glContext **需要连接的静态库。**
- ch5\dll：**运行** glContext.dll **需要链接的动态库。**

第 6 章　CAD 应用程序的几何内核模块的设计

本章要点：

- **利用面向对象的技术设计几何对象类。**
- **串行化存储/读取文档。**
- **虚拟函数的应用。**
- **纯虚拟函数。**
- **设计与使用 CAD 几何内核库。**
- **用 OpenGL 绘制 CAD 模型。**

CAD 系统中的核心问题是对几何模型的管理和操作。CAD 几何模型可以具体到点、线、面、实体和部件等几何对象。几何对象之间又存在各种关系，如层次关系、拓扑关系等。例如，直线段是由两个端点定义的，一个部件对象是由一个或多个实体对象组合而成的，而一个几何模型又是由一系列零部件组合而成，这些都构成层次关系。在面向对象的软件开发中，对 CAD 几何对象的管理和操作表现在设计和开发一系列的类来描述、管理和操作这些几何对象和它们之间的关系。这些类的集合在本书中被称之为 CAD 系统的几何内核。几何内核的开发是 CAD 系统开发中艰巨的任务之一，它不仅仅是纯软件开发技术的问题，更涉及到许多 CAD 的专业技术知识。本章主要从使用面向对象的编程技术这个角度，来介绍 CAD 系统几何内核开发中涉及的一些问题，并通过一个几何内核库 GeomKernel.dll 的开发，予以具体分析。为阐述方便和易于理解，本书中的几何内核库 GeomKernel.dll 经过了大量简化后只包括几个基本的几何对象类。但其已经构成一个面向对象的几何模型结构框架，在这个框架的基础上，读者可根据应用程序开发的实际需要，进一步修改与扩充，添加更多的专门的几何对象类。

结合 GeomKernel.dll 的实现过程，本章也将对 C++技术中的一些重要概念予以讲解。通过本章的阅读和实践，相信读者会对 C++中的虚函数和多态性、对象的序列化这些较抽象的概念以及它们的应用特点有一个更清楚的认识。

6.1　几何对象类的设计

6.1.1　类的层次设计

在 GeomKernel.dll 中，设计了如下的几何对象类。

- Centity：几何对象基本类，描述几何对象的共有属性。
- Cpart：高级几何模型类，描述应用程序中整个几何模型。
- CSTLModel：几何模型类，描述由离散三角面片（STL 格式）表示的实体。
- CtriChip：三角面片对象类，描述三角面片。

这四个基本几何对象类之间的层次关系如图 6-1 所示。

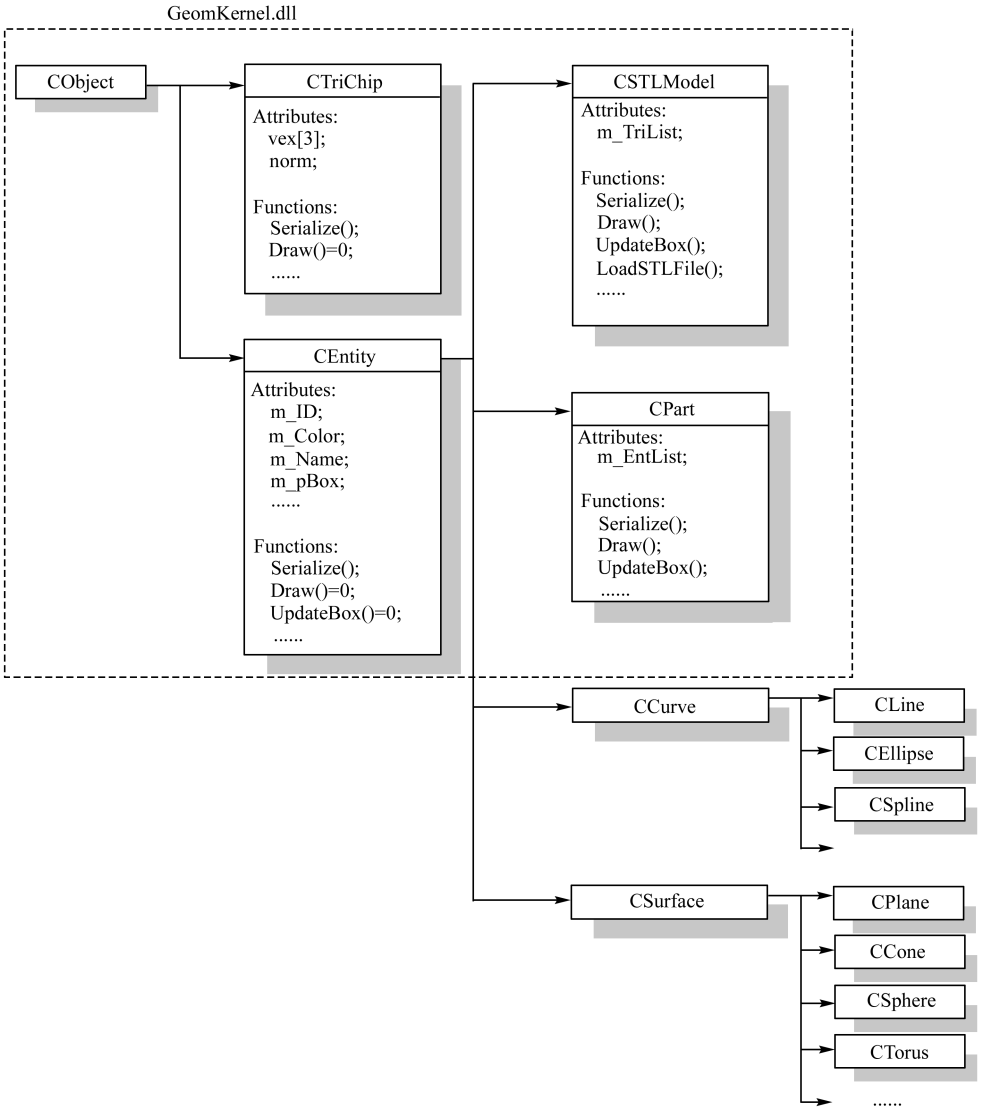


图 6-1　几何内核库 GeomKernel 中的几何类结构

CEntity 是几何对象基类，它定义了几何对象的许多公共属性，如 ID 号、颜色、名称、包容盒（描述该对象所占空间的大致范围）等，也包括了几何对象的一些公共的操作函数，如对象的图形绘制、串行化存储和读取等。CEntity 在 MFC 类 CObject 的基础上派生，因而可以继承 CObject 类的属性和操作。设计 CEntity 的目的是用于抽象描述所有几何对象的共有属性和操作。因此，和 MFC 的视图类基类 CView 一样，CEntity 也是一个抽象类，其中定义了纯虚函数，不能直接用于声明具体的几何对象。如图 6-1 所示，CEntity 作为基类，派生出了其他的几何对象类，例如 CSTLModel、Cpart、CCurve 和 CSurface。这样的设计充分利用了 C++面向对象技术的继承概念，不仅减少了代码重复，使得程序的结构清晰，而且方便管理和扩充。类 CSTLModel 是一个具体的几何对象类，用于描述一个由 STL 文件创建的三角

面片表示的实体模型。类 CTriChip 用于描述组成 STL 模型的基本三角面片对象。

GeomKernel.dll 中所设计的四个基本几何对象类构成了一个简单的层次结构。这也进一步说明了在软件结构设计的过程中，对应用问题的抽象化和层次分析是软件设计的前提。有了清晰明了的几何模型的层次结构，面向对象的编程技术中的许多概念，如类的派生继承关系、串行化技术、虚函数与多态性等，为设计 CAD 系统中几何对象的层次结构关系提供了很大方便。在这个基本结构的基础上，可以进一步地修改和扩充，如增加曲线、曲面的基本类（图 6-1 中的 CCurve 和 CSurface 类），以描述曲线、曲面的公共属性和操作。进一步地，在这些类的基础上，再派生一系列更具体的几何对象类，如直线、椭圆，以及球面、圆锥面等。图 6-2 是著名的几何造型内核系统 ACIS 中的几何类的层次设计，供读者参考。

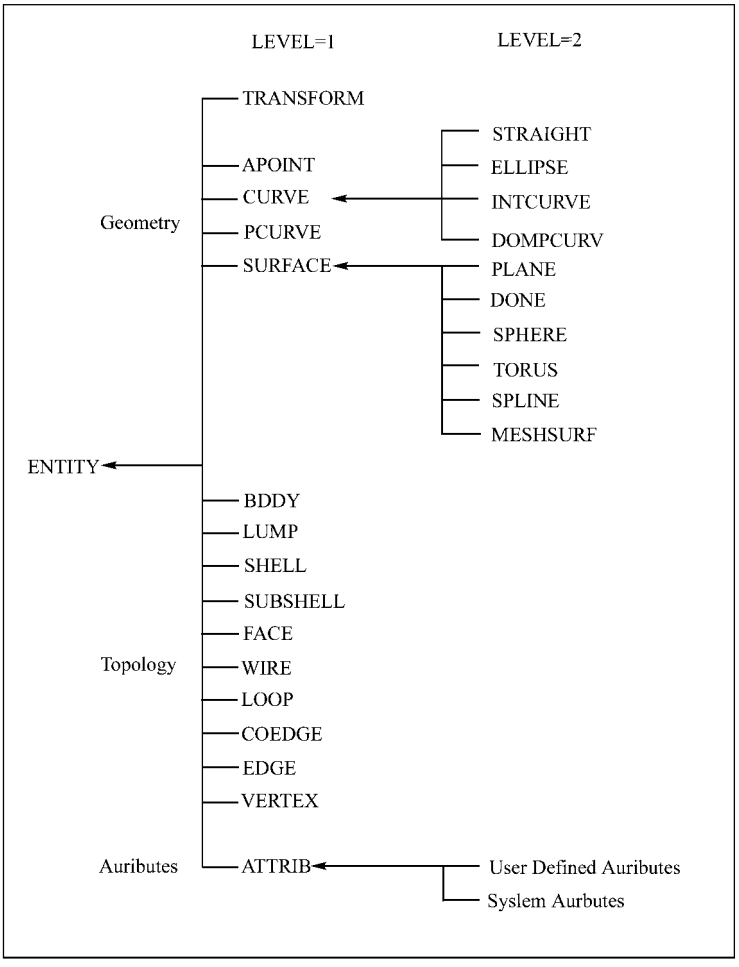


图 6-2　ACIS 中几何类的层次结构

类 CPart 是一个几何模型的集合类，它可以包含一系列的子模型。以本书中的应用程序实例 STLViewer 为例，其文档中的具体应用数据是一系列由 CSTLModel 对象组成的模型，CPart 就是设计来描述这样一个模型的集合。例如，在本书第 1 章图 1-2 中的 STLViewer 用户界面示例中的圆柱体、圆锥体等都分别是一个 CSTLModel 对象，而这些几何模型之间的相互

关系是 CPart 类描述和定义的。作为几何模型的集合，CPart 又可以被看作是一个几何体，它也具有一个几何对象的公共属性和操作函数，因而它也是在 CEntity 基础上派生得到的。

具体来说，在应用程序 STLViewer 中，对所有 STL 几何模型的数据管理是在其文档类 CSTLViewerDoc 中定义了一个 CPart 类的对象，如下面程序代码所示。

```
class CSTLViewerDoc : public CDocument
{
    ……
// Attributes
public:
    CPart          m_Part;
    …...
};
```

6.1.2 几何对象基本类 CEntity

如前提及，各种几何对象都具有一些共同的属性和操作，如对象的颜色、ID 号、注释信息、包容盒尺寸等，对象都需要图形显示和串行化读写。作为几何元素的基本类，CEntity 描述的就是这些几何对象的共同属性和操作功能或接口。由这个基本类派生得到的具体几何对象类将能够继承这些属性和基本操作。

CEntity 类的定义如下：

```
class AFX_EXT_CLASS CEntity : public CObject
{
//几何对象的基本属性变量
protected:
    BOOL        m_bModified;    //是否被修改过
    CBox3D*     m_pBox;         //指向一个最小包容盒
    UINT        m_id;           //几何体的 ID 号
    CString     m_name;         //几何体的字符标识信息
    COLORREF    m_color;        //几何体的颜色

//几何体的基本操作
public:
    CEntity();                          //构造函数
    virtual ~CEntity();                 //析构函数

    virtual void Draw(COpenGLDC* pDC)=0;    //显示几何体（纯虚拟函数）
    virtual void Serialize(CArchive& ar);   //串行化存取

    //对属性的访问和设置
    BOOL    GetBox(CBox3D& box);        //获取包容盒
    void    SetID(UINT nID);            //设置 ID 号
    UINT    GetID();                    //获取 ID 号
    void    SetName(LPCTSTR name);      //设置字符标识信息
    CString GetName();                  //获取字符标识信息
    void    SetColor(COLORREF color);   //设置颜色
```

```
    COLORREF      GetColor();                    //获取颜色

protected:
    virtual       void UpdateBox()=0;            //重新计算包容盒尺寸（纯虚拟函数）
};
```

对 CEntity 类的设计，以下几个概念有必要说明：

（1）在 CObject 类上派生 CEntity 类。CObject 是所有 MFC 类的基类。它设计了主要四组功能：内存管理、串行化存储和读取、RUN-TIME 信息、调试功能。在 CObject 上派生，CEntity 就可以继承基类提供的这些功能。例如，每个几何对象都需要串行化以实现对文档的读和写。有了 RUN-TIME 信息，对 CEntity 以及 CEntity 派生类的对象，可以使用 CObject 的函数 IsKindOf()来判断对象的类型。

（2）使用虚拟函数。在 CEntity 中定义了几个虚拟函数（Virtual Function），其中还包括两个纯虚拟函数用于统一接口。

```
virtual void Draw(COpenGLDC* pDC)=0;    //显示几何体（纯虚拟函数）
virtual void Serialize(CArchive& ar);   //串行化存取
virtual void UpdateBox()=0;             //重新计算包容盒尺寸（纯虚拟函数）
```

虚拟函数是指某个函数在基类中被声明为 virtual，而在派生类中又重新定义了此函数。类的虚拟函数和由此实现的多态性是面向对象技术的一个重要内容。使用虚拟函数的主要目的在于让程序在运行时能够实现多态性，大多数 MFC 类的设计中使用了虚拟函数。虽然几乎所有 C++的入门书籍都介绍了虚拟函数的概念，但对虚拟函数的真正理解和灵活运用还有赖于在实践中对虚拟函数概念及其特点的深入体会。本章 6.3 节将对虚拟函数和多态性进行深入的讨论。CEntity 作为几何元素基类，其他几何模型类，如 CPart 和 CSTLModel 类都由 CEntity 派生。我们将结合这些派生类的设计和应用来介绍多态性的特点，以及如何用虚拟函数实现多态性。

下面给出的程序代码是 CEntity 的实现函数，请注意函数 GetBox()对纯虚拟函数 UpdateBox()的调用。

```
/////////////////////////////////////// CEntity 的实现文件/////////////////////////////////////////////
CEntity::CEntity()
{
    //初始化公共属性
    //在构造一个由 CEntity 派生的类的对象时，这个构造函数被首先调用
    m_bModified = FALSE;
    m_id = -1;
    m_pBox = NULL;
    m_color = 0;
    m_name = _T("Unknow");
}

CEntity::~CEntity()
{
}
```

```
void CEntity::Serialize(CArchive& ar)
{
        If(ar.IsStoring()){//串行化存储
            ar << m_id;
            ar << m_name;
            ar << m_color;
        }
        else{       //串行化读取
            ar >> m_id;
            ar >> m_name;
            ar >> m_color;
        }
    }

BOOL    CEntity::GetBox(CBox3D& box)
{
    //如果实体被修改过，则需要重新计算它的包容盒
    if(m_bModified)
        UpdateBox();            // UpdateBox()是在基类 CEntity 中定义的纯虚拟函数
    if( m_pBox){
        box = *m_pBox;
        return TRUE;
    }
    else
        return FALSE;
    }
}
//操作 CEntity 中几何对象属性的有关函数

void  CEntity::SetID(UINT nID)
{
    m_id = nID;
}

UINTCEntity::GetID()
{
    return m_id;
}

void  CEntity::SetName(LPCTSTR name)
{
    m_name = name;
}
CString    CEntity::GetName()
{
    return m_name;
```

```
}

void  CEntity::SetColor(COLORREF color)
{
        m_color = color;
}

COLORREF       CEntity::GetColor()
{
        return m_color;
}
```

6.1.3 三角面片对象类 CTriChip

类 CTriChip **用于描述一个三角面片。**CTriChip **由** MFC **基类** CObject **直接派生，目的在于实现三角面片数据的串行化功能。一系列的** CTriChip **对象就构成一个** CSTLModel **的对象。实际上，**CTriChip **也可以被当作是一个几何对象类而从** CEntity **上派生。在本书中，考虑到它主要是用于描述具体的三角面片数据，如顶点和平面法矢量的坐标等，并不需要** CEntity **中的一些模型的几何属性和操作，因此，选择直接从** CObject **派生得到。这样做的另一好处是还会占用较少存储空间。类** CTriChip **的定义和实现代码如下，请读者注意类中关于串行化的声明。**

```
////////////////// CTriChip 的定义文件//////////////////////////////
class AFX_EXT_CLASS CTriChip:public CObject
{
        DECLARE_SERIAL(CTriChip)            //串行化声明
public:
        CPoint3D    vex[3];                 //三角面片顶点数据
        CVector3D normal;                   //三角面片法矢数据

public:
        CTriChip();         //构造函数
        CTriChip(const CPoint3D& v0,const CPoint3D& v1,const CPoint3D& v2,
                const CVector3D& norm);
        virtual ~CTriChip();      //析构函数

        virtual void Draw(COpenGLDC* pDC);          //显示函数
        virtual void Serialize(CArchive& ar);       //串行化存取
        const CTriChip& operator=(const CTriChip&);//操作符重载
};

////////////////// CTriChip 的实现文件//////////////////////////////
IMPLEMENT_SERIAL(CTriChip,CObject,0)
CTriChip::CTriChip()
{}
CTriChip::CTriChip(const CPoint3D& v0,const CPoint3D& v1,const CPoint3D& v2,
                const CVector3D& nor)
```

```
{
	vex[0] = v0;
	vex[1] = v1;
	vex[2] = v2;
	normal = nor;
}

CTriChip::~CTriChip()
{}

const CTriChip& CTriChip::operator=(const CTriChip& tri)
{
	normal = tri.normal;
		for(int i=0;i<3;i++)
			vex[i] = tri.vex[i];
	return *this;
}

void CTriChip::Draw(COpenGLDC* pDC)
{
	pDC->DrawTriChip(normal.dx,normal.dy,normal.dz,
		vex[0].x,vex[0].y,vex[0].z,
		vex[1].x,vex[1].y,vex[1].z,
		vex[2].x,vex[2].y,vex[2].z);
}

void CTriChip::Serialize(CArchive& ar)
{
	if(ar.IsStoring()){
		ar << normal.dx << normal.dy << normal.dz;
		for(int i=0;i<3;i++)
			ar << vex[i].x << vex[i].y << vex[i].z;
	}
	else{
		ar >> normal.dx >> normal.dy >> normal.dz;
		for(int i=0;i<3;i++)
			ar >> vex[i].x >> vex[i].y >> vex[i].z;
	}
}
```

6.1.4 STL 几何模型类 CSTLModel

类 CSTLModel **是** CEntity **类的派生类，用于描述由一系列离散的三角面片组成的单一实体模型，**CSTLModel **几何对象的创建是通过读入** STL **格式文件，获取有关的三角面片信息来组成单一的** STL **几何模型而实现的。如图** 6-3 **所示，模型中的实体对象——圆球、圆锥、圆柱、圆环及立方体等（已消隐）都是通过数据** STL **文件创建得到的** CSTLModel **的对象，根**

据 STL 文件输出的精度不同，它们各自都由一系列三角面片所构造。类 CSTLModel 在继承了基类 CEntity 基本属性的基础上，增加了离散三角面片的信息 m_TriList，以及对 m_TriList 的操作作为它的特殊属性。

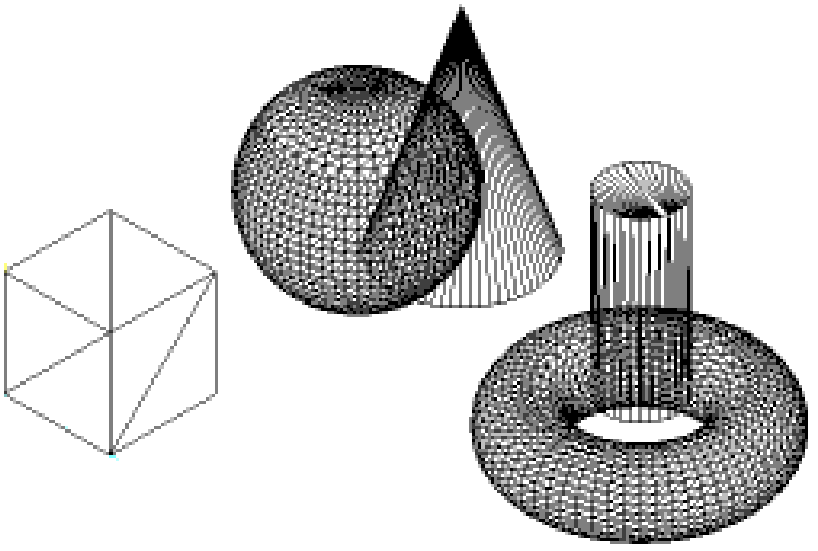


图 6-3　由 CSTLModel 类表示的一组实体模型

STL（StereoLithographic）文件又称为立体光造型文件。大多数 CAD 系统都能将几何模型的数据以 ASCII 或二进制形式输出其 STL 文件。为了便于讲述，本例中使用由 AutoCAD R14 输出的 ASCII 形式的 STL 文件。在 STL 文件中的三角面片信息单元 facet 是一个带矢量方向的三角面片，STL 模型就是由一组这样的三角面片构成。如下列 STL 文件所示，在一个 STL 文件中，每一个 facet 由 7 行数据组成，第 1 行是三角面片指向实体外部的法向矢量数据，第 2 行说明随后的第 3、4、5 行数据分别是三角面片的三个顶点的信息，沿指向实体外部的法向矢量（第 1 行数据）方向逆时针排列。

```
////////////////////// STL 文件格式 ////////////////////////
solid AutoCAD
  facet normal -9.9619470e-001 0.0000000e+000 8.7155743e-002
     outer loop
       vertex 0.0000000e+000 2.0000000e+002 0.0000000e+000
       vertex 0.0000000e+000 0.0000000e+000 0.0000000e+000
       vertex 3.4995465e+000 3.4995465e+000 4.0000000e+001
     endloop
endfacet
     ……
……
……
     facet normal 0.0000000e+000 0.0000000e+000 1.0000000e+000
        outer loop
          vertex 1.9650045e+002 3.4995465e+000 4.0000000e+001
```

```
      vertex 1.9650045e+002 1.9650045e+002 4.0000000e+001
      vertex 3.4995465e+000 3.4995465e+000 4.0000000e+001
    endloop
  endfacet
endsolid AutoCAD
```

类 CSTLModel **提供了读入** STL **格式文件的接口函数，在程序中对一个** STL **模型对象的创建需要先声明一个** CSTLModel **的对象，然后再读入相应的** STL **文件。一个** CSTLModel **对象的创建代码如下：**

```
CSTLModel    stlModel;
CString    stlFileName;
……
ASSERT(stlModel.LoadSTLFile(stlFileName));
```

类 CSTLModel **的定义如下：**

```
class AFX_EXT_CLASS CSTLModel: public CEntity
{
    DECLARE_SERIAL(CSTLModel)        //串行化声明
public:
    CTypedPtrArray<CObArray,CTriChip*> m_TriList;    //三角面片数据链

public:
    CSTLModel();                              //构造函数
    virtual ~CSTLModel();                     //析构函数

    virtual void        Draw(COpenGLDC* pDC);              //显示函数
    virtual void        Serialize(CArchive& ar);           //串行化存取
    const      CSTLModel& operator=(const CSTLModel&);     //操作符重载

    void  Add(CTriChip* tri);                              //增加三角面片
    BOOL LoadSTLFile(LPCTSTR file);                        //读入 STL 文件
    void  Clear();                                         //清除几何数据

    BOOL       IsEmpty();                                  //判断模型是否为空

protected:
    virtual void UpdateBox();     //重新计算包容盒尺寸
};
```

在以上的定义程序代码中需要注意的是，类中重载了虚拟函数 UpdateBox()**和** OnDraw()**。在** CAD **系统中，计算几何模型的包容盒尺寸和几何模型的显示是对一个具体的几何模型类而言必须实现的功能。由于** CEntity **类中包括了纯虚拟函数，它不能用来直接声明对象，这样的类也被称为抽象类或虚类。**CSTLModel **必须重载这些纯虚拟函数，为它们设计实现相应的代码。类** CSTLModel **的实现代码如下所示：**

```
IMPLEMENT_SERIAL(CSTLModel,CObject,0)
```

```
//类的构造和析构函数
CSTLModel::CSTLModel()
{}

CSTLModel::~CSTLModel()
{
	Clear();
}
///no explanantion?
void CSTLModel::Add(CTriChip* tri)
{
	m_TriList.Add(tri);
}

//根据当前所包含三角面片的几何信息，更新 STLModel 对象的最大包容盒
void CSTLModel::UpdateBox()
{
	if(m_pBox){
			delete m_pBox;
			m_pBox = NULL;
	}

	if(m_TriList.GetSize()==0)
			return;

	double x0,y0,z0,x1,y1,z1;
	x0=y0=z0=10000;
	x1=y1=z1=-10000;

	CTriChip* tri;
	for(int i=0;i<m_TriList.GetSize();i++){
			tri = m_TriList[i];
			for(int n=0;n<3;n++){
				if(tri->vex[n].x<x0)	x0 = tri->vex[n].x;
			if(tri->vex[n].x>x1)	x1 = tri->vex[n].x;
				if(tri->vex[n].y<y0)	y0 = tri->vex[n].y;
				if(tri->vex[n].y>y1)	y1 = tri->vex[n].y;
				if(tri->vex[n].z<z0)	z0 = tri->vex[n].z;
			if(tri->vex[n].z>z1)	z1 = tri->vex[n].z;
			}
	}

	m_pBox = new CBox3D(x0,y0,z0,x1,y1,z1);
	m_bModified = FALSE;
}

//读入 ASCII 格式的 STL 文件
```

```
BOOL CSTLModel::LoadSTLFile(LPCTSTR stlfile)
{
	FILE* file;
	int type=0;
	if((file = fopen(stlfile, "r")) == NULL)
		return FALSE;

	char str[80];
	CTriChip* tri = NULL;
	while(fscanf(file,"%s",str)==1){
		if(strncmp(str,"normal",6)==0){
			tri = new CTriChip();
			fscanf(file,"%lf %lf %lf", &(tri->normal.dx), &(tri->normal.dy), &(tri->normal.dz));
			fscanf(file,"%*s %*s");
			fscanf(file,"%*s %lf %lf %lf",&(tri->vex[0].x),&(tri->vex[0].y),&(tri->vex[0].z));
			fscanf(file,"%*s %lf %lf %lf",&(tri->vex[1].x),&(tri->vex[1].y),&(tri->vex[1].z));
			fscanf(file,"%*s %lf %lf %lf",&(tri->vex[2].x),&(tri->vex[2].y),&(tri->vex[2].z));
			Add(tri);
		}
	}
	char title[80];
	if(GetFileTitle(stlfile,title,80)==0){
		SetName(title);
	}

	m_bModified = TRUE;
		return TRUE;
}

//串行化读写文档中的几何模型数据
void  CSTLModel::Serialize(CArchive& ar)
{
	CEntity::Serialize(ar);
		m_TriList.Serialize(ar);

	if(ar.IsLoading())
	m_bModified = TRUE;
}

//使用 OpenGL 绘制模型，即绘制模型中的所有三角面片
void  CSTLModel::Draw(COpenGLDC* pDC)
{
	for(int i=0;i<m_TriList.GetSize();i++)
		m_TriList[i]->Draw(pDC);
}

//清除模型中的所有三角面片，即清空模型数据的存储空间
```

```
void  CSTLModel::Clear()
{
      for(int i=0;i<m_TriList.GetSize();i++)
            delete m_TriList[i];
      m_TriList.RemoveAll();
      m_bModified = TRUE;
}

BOOL      CSTLModel::IsEmpty()
{
      return m_TriList.GetSize() == 0;
}
```

6.1.5 高级几何模型类 CPart

在 STLViewer **中，描述和管理整个几何模型的类是** CPart**，本书称之为几何内核库中的高级几何模型。如下所示，在文档类** CSTLViewerDoc **中插入了一个** CPart **的对象** m_Part**，以实现对文档中几何模型的管理。**

```
class CSTLViewerDoc : public CDocument
{
……..
// Attributes
public:
      CPart        m_Part;
…….
};
```

几何模型的集合同样可以作为一个几何对象，所以 CPart **具有几何对象的公共属性和操作，在类的层次结构上，它也由几何模型基本类** CEntity **派生。同时作为一个高级几何模型类，一个** CPart **类的对象中可包括一系列** CEntity **派生的几何对象。在** CPart **类的定义中，以** MFC **基类** CObArray **的对象** m_EntList **来管理这些几何对象。具体的定义代码如下：**

```
CTypedPtrArray<CObArray,CEntity*> m_EntList;
```

如本书第 1 **章图** 1-2 **所示，应用程序中的几何模型集合（变量** m_Part **表示）包含圆锥、圆柱、立方体及圆环等多个实体模型。需要强调的是，**m_EntList **管理的是一个指向** CEntity **对象的指针数组，所以可载入任何** CEntity **的派生类，而不仅仅限于** CSTLModel **的对象， 这为** CAD **系统中几何模型的扩充创造了方便。**

类 CPart **的定义代码如下：**

```
class AFX_EXT_CLASS CPart : public CEntity    //由 CEntity 派生
{
protected:
      CTypedPtrArray<CObArray,CEntity*> m_EntList;   //几何元素对象指针链

public:
      CPart();                          //构造函数
```

```
    virtual ~CPart();                    //析构函数

    virtual void Draw(COpenGLDC* pDC);   //显示函数
    virtual void Serialize(CArchive& ar); //串行化存取函数
    void  AddEntity(CEntity* ent);       //添加几何体函数
    void  RemoveEntity(CEntity* ent);    //释放单一几何体的存储空间函数
    void  RemoveAllEntity();             //释放所有几何体的存储空间函数

    BOOL      IsEmpty();                 //判断模型是否为空的函数

    UINTGetEntitySize();                 //获取模型内几何体具体数量的函数
    CEntity*    GetEntity(UINT i);       //获取模型集合内序号为 i 的几何体的函数

protected:
    virtual voidUpdateBox();             //重新计算所有几何模型整体包容盒的函数
};
```

类 CPart **的实现代码如下：**

```
CPart::CPart()
{}

CPart::~CPart()
{
    RemoveAllEntity();
}

void CPart::Draw(COpenGLDC* pDC)
{
    for(int i=0;i<m_EntList.GetSize();i++)
        m_EntList[i]->Draw(pDC);
}

void CPart::Serialize(CArchive& ar)
{
    CEntity::Serialize(ar);
        m_EntList.Serialize(ar);
    if(ar.IsLoading()) m_bModified = TRUE;
}

void CPart::AddEntity(CEntity* ent)
{
    m_EntList.Add(ent);
        m_bModified = TRUE;
}

void CPart::RemoveEntity(CEntity* ent)
{
```

```
	for(int i=0;i<m_EntList.GetSize();i++){
		if(ent == m_EntList[i]){
m_EntList.RemoveAt(i);
			m_bModified = TRUE;
			break;
		}
	}
}

void CPart::RemoveAllEntity()
{
	for(int i=0;i<m_EntList.GetSize();i++)
delete m_EntList[i];
m_EntList.RemoveAll();
m_bModified = TRUE;
}

BOOL CPart::IsEmpty()
{
	return m_EntList.GetSize() == 0 ;
}

UINTCPart::GetEntitySize()
{
	return m_EntList.GetSize();
}

CEntity* CPart::GetEntity(UINT i)
{
	ASSERT(i<m_EntList.GetSize());
		return m_EntList[i];
}

void CPart::UpdateBox()
{
	if(m_pBox){
		delete m_pBox;
		m_pBox = NULL;
	}
	CBox3D box;
	for(int i=0;i<m_EntList.GetSize();i++){
		if(m_EntList[i]->GetBox(box)){

			if(m_pBox)
				*m_pBox += box;
			else{
				m_pBox = new CBox3D();
```

```
                    *m_pBox = box;
                }
            }
        }
        m_bModified = FALSE;
    }
```

6.2 串行化(Serialize)实现文档存取功能

6.2.1 为什么要使用串行化

MFC 中的串行化函数是为了在面向对象的程序设计中方便地存储对象和从文档中读取并构造对象而设计的。我们注意到以上提及的几何内核库中的每个类都是从 MFC 基类 CObject 派生，并且都重载了一个虚拟函数 Serialize()。MFC 就是通过这个函数来提供串行化存取文档功能的。

对于一个 CAD 系统，几何模型信息需要以文件的形式保存起来，并能够从文件中重新读取这些信息以恢复 CAD 模型。MFC 提供了两种常用的文档存取方法：一是直接利用 CFile 对象存取文档，二是利用串行化方法（Serialize）存取文档。直接利用 CFile 对象存取文档会比较繁琐，尤其对于文档结构复杂的情况。例如，对于各种不同类型和层次的数据对象，编程人员首先需要判断数据的类型，构造一个定义其结构的相应的抽象数据结构对象，然后才能调用 CFile 的读取函数来读取该对象的成员变量以获取对应的数据。因此，编写不同数据的存取函数将是件复杂而繁琐的工作。MFC 提供的串行化技术专门用于解决面向对象的程序设计中各种对象数据的存储和创建。使用串行化技术，用户可以很方便地把对象的不同数据信息存储到指定的二进制文件中去，也可方便地用同样的顺序从该二进制文件中读出并创建对象。因为 MFC 中的很多类都支持串行化功能，所以在 MFC 程序中使用串行化技术存取文档更加方便。例如，在类 CSTLModel 和 CPart 中用于管理数据的模板类 CObArray 本身就支持 Serialize 功能，这样在编写 CSTLModel 和 CPart 的串行化函数时，对 CObArray 中几何模型数据的存取只用一条语句就轻松完成了。

6.2.2 CArchive 类

文档的串行化功能是通过一个 MFC 基类 CAarchive 实现的，而 CArchive 类是对文件类 CFile 进行包装而建立起来的。也就是说，通过串行化管理的文档(一般为磁盘文件)从本质上说是一个用 CFile 类的对象管理的文件。在构造一个 CArchive 对象之前，必须首先建立一个 CFile 的对象。另外，还必须保证文档的读/写状态与文件打开的模式兼容。不过，串行化时不再通过 CFile 的成员函数来操作这个文件，而是利用直接由 CArchive 类提供的相应功能来管理文档。

CArchive 类的构造函数如下：

```
CArchive( CFile* pFile, UINT nMode, int nBufSize = 4096, void* lpBuf = NULL );
```

在调用以上构造函数构造一个 CArchive 对象时，必须输入一个指向 CFile 对象的指针

(pFile)。应用程序文档和磁盘文件之间的存储和读取就是通过这个 CFile 对象实现的。

下面两段代码是用来分别创建文档存储和读取的 CArchive 对象。

```
//////创建一个用于存储文档数据的 CArchive 对象
CFile file;
//以写模式打开文件
file.Open(strFileName,CFile::modeCreate|CFile::modeWrite);
//以存储模式构造 CFile 对象
CArchive ar(&file,CArchive::store);
……
//串行化存储
……
ar.Close();  //关闭 CFile 对象之前应该首先关闭 CArchive 对象
file.Close();

//////创建一个用于从文档中读取数据的 CArchive 对象
CFile file;
//以读取模式打开文件
file.Open(strFileName,CFile::modeRead);
//以读取模式构造 CFile 对象
CArchive ar(&file,CArchive::load);
……
//串行化读取
……
ar.Close();  //关闭 CFile 对象之前应该首先关闭 CArchive 对象
file.Close();
```

6.2.3 串行化类的设计步骤

以上完成的四个类都是串行化类。换句话说，由于是 MFC 基类 CObject 的派生类，它们都具有串行化存储和读取文档数据的功能。从它们各自的串行化函数 Serialize()中，可以体会到利用串行化存取不同文档数据的优势。

设计一个串行化类，需要以下几个步骤：

（1）由 CObject 直接或间接派生串行化类。例如，在以上四个串行化类中，CEntity、CTriChip 是直接派生，CSTLModel 和 CPart 则是间接派生自 CObject 类。

（2）在类的定义和实现代码中包含宏声明 DECLARE_SERIAL 和宏调用 IMPLEMENT_SERIAL。例如，在 CSTLModel 类的定义和实现文件中，宏声明和宏调用如下：

宏声明：DECLARE_SERIAL(CSTLModel)

宏调用：IMPLEMENT_SERIAL(CSTLModel,CObject,0)

在 MFC 的宏调用中，第一个参数（CSTLModel）是声明串行化的类，第二个参数（CObject）是基类的名字，第三个参数（0）用于标识应用程序的版本号。在声明的类中，必须有一个默认的构造器（不带参数的构造函数），在这里，构造函数 CSTLModel::CSTLMode()就是这样的构造器。在读取文档操作时，程序自动创建对象，宏则调用这个默认的构造函数来创建和初始化一个对象。

细心的读者会发现在 CEntity 中没有包括 SERIAL 的宏声明和宏调用。这是因为对 CEntity 的设计是将其作为几何对象的基类，它本身并不实例化对象，所以不必也不能在 CEntity 中加入宏。但为了存取类中的数据成员，在类中也需要一个串行化函数 Serialzie()，在派生类的 Serialize()函数需要调用它，以存取 CEntity 中的数据。

（3）串行化成员函数 Serialize()。针对不同的数据类型，必须给类定义和实现一个自己的成员函数 Serialize()。例如，类 CSTLModel 中定义的成员函数 Serialize()如下：

```
void  Serialize(CArchive& ar);
void  CSTLModel::Serialize(CArchive& ar)
{
      CEntity::Serialize(ar);          //首先存取基类数据
      m_TriList.Serialize(ar);         //串行化自身数据
      if(ar.IsLoading())
            m_bModified = TRUE;//从文档中读取数据后，CSTLModel 对象本身数据发生了改变
                               //需要设置标志，以更新包容盒
}
```

这里需要特别注意的是，在 CSTLModel::Serialize（CArchive& ar）中，首先要调用基类 CEntity 的串行化函数，以存取基类中的信息。

6.2.4 CObArray 的 Serialize()函数

如前提及，利用串行化存取文档数据的方便之处还在于，很多 MFC 类都支持串行化功能。读者可能注意到了，在 CPart::Serialize（CArchive& ar）中，对 CPart 中数据的串行化存取只需要一行语句：

```
void CPart::Serialize(CArchive& ar)
{
      CEntity::Serialize(ar);
      m_EntList.Serialize(ar);              //利用模板类 CObArray 的 Serialize()函数
      if(ar.IsLoading()) m_bModified = TRUE;
}
```

变量 m_EntList 是一个 CObArray 类支持的 CEntity 的指针对象数组（定义见 6.1.5 节）。这个指针对象数组的串行化存储和从文档串行化读取并创建 CEntity 对象都由 CObArray 本身的 Serialize()完成。因而，用户只需调用 CObArray 的串行化函数，而不必再专门为这个数组类编写串行化代码。

6.2.5 应用程序的文档串行化实例剖析

在这一节，我们将对应用程序实例 STLViewer 中实现模型串行化存储和读取的各步骤进行详细分析。整个过程如图 6-4 所示。

1．在文档中插入 m_Part 的串行化

在使用 MFC 的文档视图结构创建的应用程序中，当从 File 菜单中选择 Save 或 Open 时，应用程序框架将打开一个文件存取对话框，让用户选择或创建相应的文件，并创建了一个 CFile 对象管理的文件。系统同时在此基础上自动创建一个 CArchive 类的对象，然后程序调

用文档类的 Serialize()函数，并将创建的 CArchive 对象作为参数传到文档类的 Serialize()函数中。接着在文档类的 Serialize()函数中对其成员数据进行串行化操作。在 CSTLViewer 中，m_Part 是文档类 CSTLViewerDoc 中需要被串行化的对象，因此，需要在函数 CSTLViewerDoc::Serialize()中加入 m_Part 的串行化代码，具体实现如下：

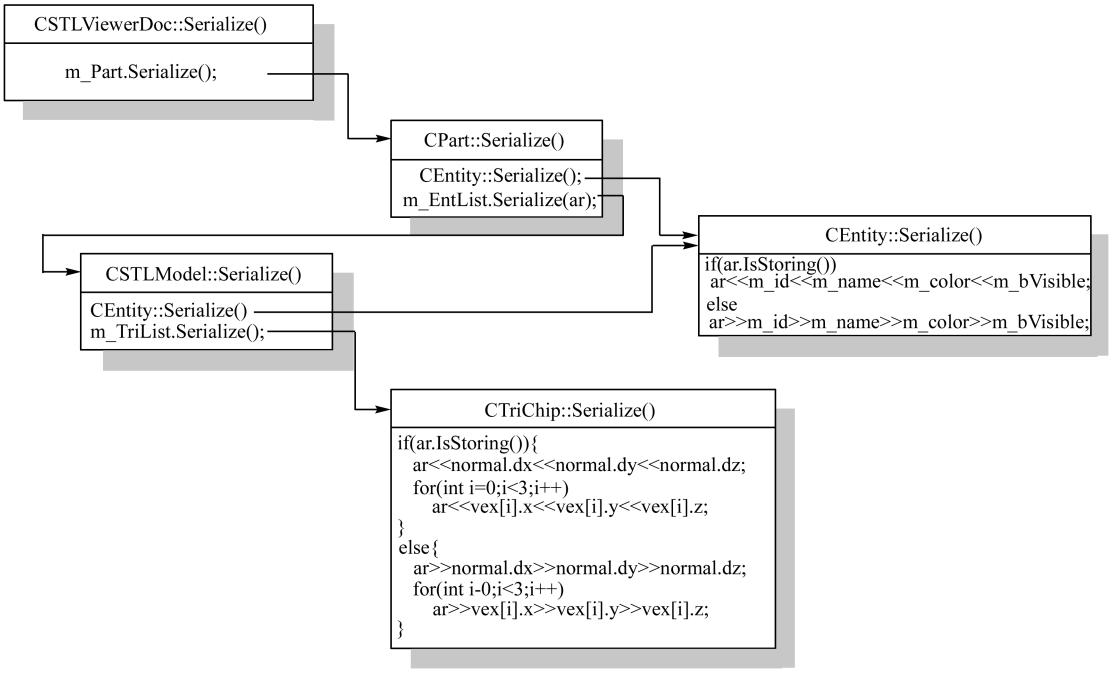


图 6-4　在应用程序 STLViewer 中实现文档串行化的各步骤

```
void CSTLViewerDoc::Serialize(CArchive& ar)
{
    m_Part.Serialize(ar);
}
```

2．执行 CPart 的串行化；

如下列代码所示，由于 CPart 是 CEntity 的派生类，所以必须先调用基类的串行化函数，然后再对自身的数据 m_EntList 进行串行化。如前提及，由于 m_EntList 是 CObArray 类的一个对象，本身支持串行化，所以可以直接使用其基类 CObArray 的 Serialize()成员函数。由于 m_EntList 是由一系列指向 CEntity 类的对象的指针所构成的指针数组，因此，对 m_EntList 进行串行化操作的实际结果是，在对 m_EntList 串行化的同时，也调用了 m_EntList 中成员对象 CEntity 的串行化函数，即实现了对指针数组 m_EntList 所指向的所有几何模型数据(STL 模型）的构造或存储。同样地，为了串行化存取 CEntity 类的对象，必须在它的串行化函数中添加相应的数据存取代码。

```
void CPart::Serialize(CArchive& ar)
{
    CEntity::Serialize(ar);          //首先调用基类的串行化
    if(ar.IsStoring()){
```

```
        m_EntList.Serialize(ar);
    }
    else{
        m_EntList.Serialize(ar);
        m_bModified = TRUE;
    }
}
```

3．执行 CEntity 的串行化

具体的程序代码如下，存取的几何对象的公用属性有：模型标识号（m_id）、模型名（m_name）、模型颜色（m_color）以及模型的可见性标识（m_bVisible）等。

```
void CEntity::Serialize(CArchive& ar)
{
    if(ar.IsStoring()){
        ar << m_id << m_name << m_color << m_bVisible;
    }
    else{ //loading
        ar >> m_id >> m_name >> m_color >> m_bVisible;
    }
}
```

4．执行 CSTLModel 的串行化

如前提及，调用 CObArray::Serialize()的结果将串行化存取一系列其所指向的 CSTLModel 对象。在 CSTLViewer 中，CEntity 指针所指向的对象的实际类型为 CSTLModel，由于 Serialize() 是虚拟函数，所以调用的实际代码是 CSTLModel 类的函数 CSTLModel::Serialize()。

如下列代码所示，由于 CSTLModel 是 CEntity 的派生类，同样地，函数 CSTLModel::Serialize() 首先需要调用基类的串行化函数来存取基类中的数据，然后进一步对本身数据进行串行化操作，即对组成该 STL 几何模型的一系列三角面片对象(m_TriList)进行串行化。

```
void  CSTLModel::Serialize(CArchive& ar)
{
    CEntity::Serialize(ar);
    if(ar.IsStoring()){
        m_TriList.Serialize(ar);
    }
    else{ //IsLoading()
        m_TriList.Serialize(ar);
        m_bModified = TRUE;
    }
}
```

5．执行 CTriChip 对象的串行化

函数 CTriChip::Serialize()对三角面片对象中的数据，即三角面片法向矢量和顶点坐标进行存储或读取。具体代码如下：

```
void CTriChip::Serialize(CArchive& ar)
{
```

```
    if(ar.IsStoring()){
        ar << normal.dx << normal.dy << normal.dz;
        for(int i=0;i<3;i++)
            ar << vex[i].x << vex[i].y << vex[i].z;
    }
    else{
        ar >> normal.dx >> normal.dy >> normal.dz;
        for(int i=0;i<3;i++)
            ar >> vex[i].x >> vex[i].y >> vex[i].z;
    }
}
```

6.3 虚拟函数

6.3.1 虚拟函数与多态性

在面向对象程序设计中，多态性（polymorphism**）是一个非常重要的概念。函数重载或操作符重载实现了名称相同而由于接口不同的功能不同，是在程序编译阶段完成的多态性。而虚拟函数实现的多态性是在程序运行阶段中的多态性，它与函数重载有较大的区别。在函数重载中，尽管函数的名称相同，但各函数的返回值类型或所传递的参数类型可以是不一样的。在虚拟函数中，各函数的返回值类型和所传递的参数类型一定要相同，否则就不能称作虚拟函数，而只能叫重载函数。事实上，虚拟函数就是通过这种接口和函数类型的完全一致来实现运行时的多态性。**

指针是支持虚拟函数多态性功能的重要工具。虚拟函数的特性是：一个指针无论是指向基类还是指向基类的派生类，在它调用虚拟函数时，都会执行和该对象相应的成员函数。读者可以通过下面的例子来体会虚拟函数的特性。

在本章提及的四个类中，使用了较多的虚拟函数。如在基类 CEntity **中，将显示函数** Draw() **定义为纯虚拟函数。它不执行任何任务，只用作统一接口。**

```
class AFX_EXT_CLASS CEntity : public CObject
{
public:
    virtual void Draw(COpenGLDC* pDC)=0;      //纯虚拟函数，用作显示几何体的统一接口
};
```

对应于基类中的纯虚拟函数 OnDraw()**，在** CEntity **的两个派生类中，也分别具体定义了各自的虚拟函数** Draw()**。**

```
class AFX_EXT_CLASS CSTLModel: public CEntity
{
protected:
    CTypedPtrArray<CObArray,CTriChip*> m_TriList;
public:
    ……
    //由于 Draw()在基类中已经被定义成虚拟函数
```

```
    //无论有没有 virtual，Draw()都是虚拟函数
    virtual void Draw(COpenGLDC* pDC);
    ……
};

class AFX_EXT_CLASS CPart : public CEntity    //由 CEntity 派生
{
protected:
    CTypedPtrArray<CObArray,CEntity*> m_EntList;
public:
    //由于 Draw()在基类中已经被定义成虚拟函数
    //无论有没有 virtual，Draw 都是虚拟函数
    virtual void Draw(COpenGLDC* pDC);
};

void  CSTLModel::Draw(COpenGLDC* pDC)
{
    for(int i=0;i<m_TriList.GetSize();i++){
        m_TriList[i]->Draw(pDC);
    }
}
```

函数 CPart::Draw()**的目的是将其成员变量** m_EntList**（一组指向** CEntity **对象的指针）所指向的对象（一组具体的几何对象）用** OpenGL **在屏幕上绘制出来。在** CPart **中，我们声明的是一组指向** CEntity **的指针（**m_EntList**）。这是因为在** C++**中当声明一个指向基类的指针时，该指针也可指向该基类的派生类。在** Cpart **中的** m_EntList **是一个指向** CEntity **对象的指针数组，也可以指向** CEntity **的派生类，如** CSTLModel **或其他的几何体（完全可以在** CEntity **基础上派生更多类型的派生类）。**CPart::Draw()**中，我们希望** pEnt->Draw(pDC) **执行的是相应的具体对象的成员函数，即** CSTLModel::Draw()**，而非函数** CEntity::Draw()**。由于** Draw()**是虚拟函数，这个问题就得到了解决。**pEnt->Draw(pDC)**调用的将是实际所指向的具体对象自己的** Draw()**，即函数** CSTLModel::Draw()**。如果不把** Draw()**设为虚拟函数，**pEnt->Draw(pDC)**调用的将是** CEntity::Draw()**，这就无法绘制出相应的对象模型，因为在** CEntity::Draw()**中什么都不执行。**

```
void CPart::Draw(COpenGLDC* pDC)
{
    CEntity* pEnt;
    for(int i=0;i<m_EntList.GetSize();i++){
        pEnt = m_EntList[I];
        pEnt->Draw(pDC);      //调用 CSTLModel 中的成员函数 Draw()
    }
}
```

函数被定义为虚拟函数后，在它的派生类中，只要函数的接口和返回值类型完全一致，无论是否在派生类中的相应函数上也加上标识符 virtual**，这种虚拟特性都会一直继承下去。由于** CSTLModel**、**CPart **是** CEntity **的派生类，所以在** CSTLModel **和** CPart **中，函数** Draw()

继承了虚拟的特性，不加 virtual 标识符也同样被系统定义为虚拟函数。

6.3.2 纯虚拟函数

我们注意到 CEntity 类作为描述几何对象的基类，本身不会用来声明任何对象。它的虚拟函数 Draw()和 UpdateBox()不执行任何语句，而只是作为单一接口供派生类调用。这是 C++中常用的一种方法，即在基类中通过定义虚拟函数来定义某事件的接口，而在派生类中定义处理该事件的方法。通过这样的方式来实现单一接口，多种功能。如果定义的虚拟函数不需要执行具体功能，为了减少程序的复杂程度并节省内存空间，可以将这些函数设为纯虚拟函数，即让此函数等于零。在 CEntity 中，定义了以下两个纯虚拟函数。

```
virtual void    Draw(COpenGLDC* pDC)=0;
virtual         void UpdateBox()=0;
```

一个定义了纯虚拟函数的类只有抽象的作用，也称为抽象类。抽象类不能用来声明对象，但可以用来声明指向该对象的指针，因为指针是支持多态性的主要工具。本章中的 CEntity 类即被设计成这样的一个抽象类，若用它来声明对象，系统编译时将无法通过。以下关于抽象类的不同使用方法供读者比较区分，以加深理解：

```
CEntity          ent;       //非法创建对象
CEntity*         pEnt;      //可以声明指向抽象类的指针
pEnt = new CEntity();       //非法创建对象
CSTLModel        stlModel;  //声明一个 CSTLModel 的对象
pEnt = &stlMode;            //将 pEnt 指向 CSTLModel 对象
……
pEnt->Draw(pGLDC);          //调用 CSTLModel::Draw()，而不是 CEntity::Draw()
……
```

6.3.3 实现 CPart 模型的 OpenGL 显示

应用程序实例 CSTLViewer 中的几何模型 m_Part 是 CPart 类的对象。实现对三维几何模型的显示，实际上就是用 OpenGL 来实现 m_Part 的显示。CPart 模型 OpenGL 显示的各环节是建立在使用虚拟函数 Draw()的基础上。在 CPart 中实现模型 OpenGL 显示的具体环节（图 6-5）如下：

（1）调用 m_Part.Draw()并将 COpenGLDC 的指针传递给它，以用 OpenGL 绘制该模型。因为 m_Part 是 STLViewer 文档类（STLViewerDoc)中的一个对象，在函数 CSTLViewerView::RenderScene()中可以被访问。程序代码如下：

```
void CSTLViewerView::RenderScene(COpenGLDC* pDC)
{

    CSTLViewerDoc* pDoc = GetDocument();
    ASSERT(pDoc);

    pDC->DrawCoord();                   //绘制坐标系框架
    if(!pDoc->m_Part.IsEmpty())         //如果模型不为空
            pDoc->m_Part.Draw(pDC);
```

```
}
```

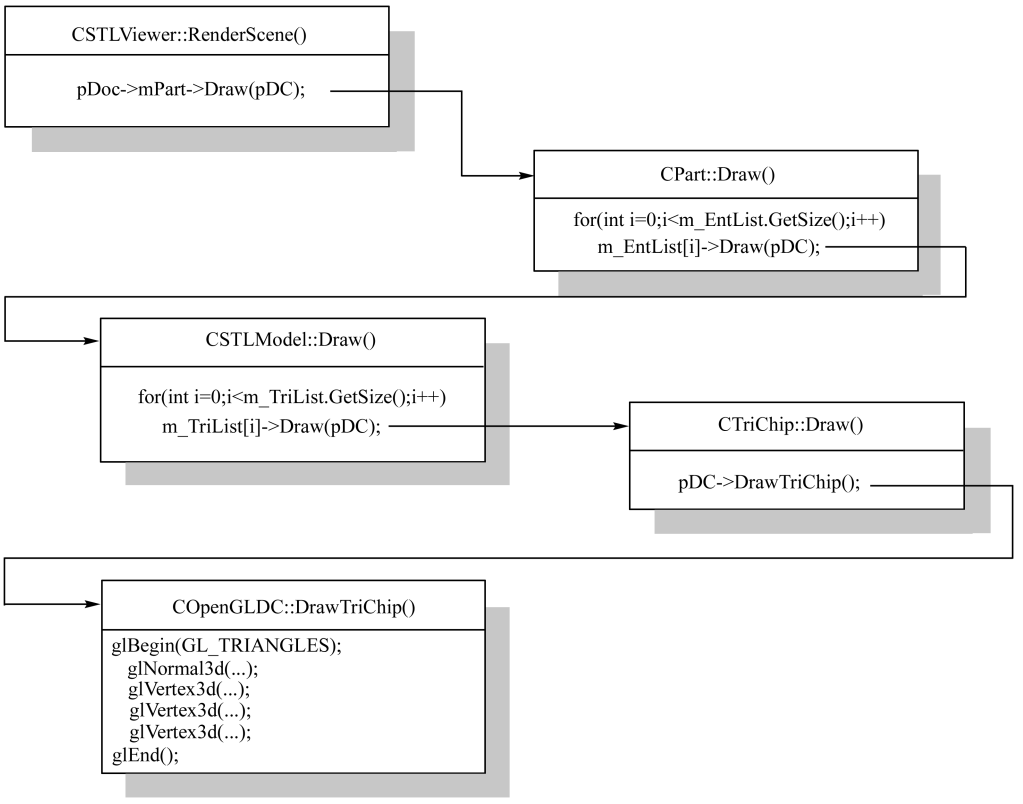


图 6-5　实现 CPart 模型 OpenGL 绘制的各环节

（2）在 CPart::Draw()**中，对指针数组** m_EntList **的每个成员，调用** CEntity **的显示函数** Draw()**，以绘制它所指向的几何对象。**

具体的程序代码如下：

```
void CPart::Draw(COpenGLDC* pDC)
{
    for(int i=0;i<m_EntList.GetSize();i++)
        m_EntList[i]->Draw(pDC);
}
```

（3）由于 CEntity::Draw()**是个虚拟函数，且是个纯虚拟函数，对** CEntity::Draw()**的调用实际上是执行了它的派生类中对应的函数** CSTLModel::Draw()**。在函数** CSTLModel::Draw() **中，调用** CTriChip::Draw()**以绘制所包含的每一个三角面片。**

```
void  CSTLModel::Draw(COpenGLDC* pDC)
{
    if(!m_bVisible)    return;
    if(m_bHighlight)
        pDC->Highlight(TRUE);
    else
```

```
        pDC->SetMaterialColor(m_color);

    if(pDC->IsSelectionMode())
        pDC->LoadName((UINT)this);

    for(int i=0;i<m_TriList.GetSize();i++)
        m_TriList[i]->Draw(pDC);
}
```

（4）在函数 CTriChip::Draw()**中，使用** COpenGLDC::DrawTriChip()**来绘制当前的三角面片对象。**

```
void CTriChip::Draw(COpenGLDC* pDC)
{
    pDC->DrawTriChip(normal.dx,normal.dy,normal.dz,
        vex[0].x,vex[0].y,vex[0].z,
        vex[1].x,vex[1].y,vex[1].z,
        vex[2].x,vex[2].y,vex[2].z);
}
```

（5）在函数 COpenGLDC::DrawTriChip()**中，使用** OpenGL **命令来绘制一个带法向矢量的三角面片。**

```
void COpenGLDC::DrawTriChip(double n0,double n1,double n2,
        double v00,double v01,double v02,
        double v10,double v11,double v12,
        double v20,double v21,double v22)
{
    glBegin(GL_TRIANGLES);
        glNormal3d(n0,n1,n2);
        glVertex3d(v00,v01,v02);
        glVertex3d(v10,v11,v12);
        glVertex3d(v20,v21,v22);
    glEnd( );
}
```

6.4 建立几何内核库 GeomKernel.dll

把以上介绍的几何模型类集成在一个独立的动态链接库 GeomKernel.dll **中，就成为一个基本的几何内核库（**GeomKernel.dll**）。这个** DLL **库将输出以下四个类供应用程序使用**:

- CEntity
- CPart
- CSTLModel
- CTriChip

一个几何内核库的开发往往是 CAD 开发中最复杂的部分，对真正的 CAD 系统而言，所包括的类的内容以及类与类之间的层次和结构都会复杂得多。这里的 GeomKernel.dll 只是一个简化过的几何层次结构，在这个结构基础上还可以根据需要进一步修改与扩充。

建立这个 DLL 库时要注意的是，由于 GeomKernel.dll 中的类使用了动态库 GeomCalc.dll 和 glContext.dll 的输出类，因而，需要在所创建的 MFC 项目（Project）中加入这两个动态库的静态连接库 GeomCalc.lib 和 glContext.lib。加入操作完成后的 GeomKernel 的文件列表窗口如图 6-6 所示。

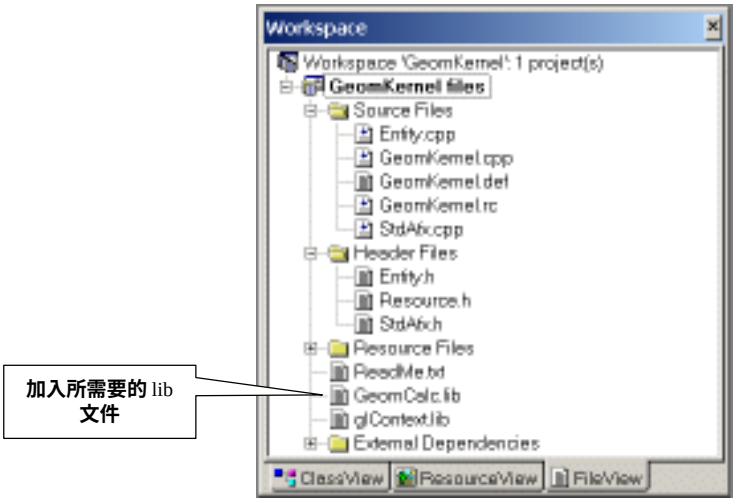


图 6-6　项目 GeomKernel 中的文件列表

项目 GeomKernel 的具体代码可参见本书附带光盘中的 ch6\GeomKernel 子目录。读者可以将内核库 GeomKernel 的以下相关文件拷贝出来，供其他程序调用。

- 类的说明文件：Entity.h
- 动态链接库文件：GeomKernel.dll
- 静态连接库文件：GeomKernel.lib

6.5　程序清单

6.5.1　文件 Entity.h

```
////////////////////////////// ENTITY.H ////////////////////////////////

#if !defined(AFX_ENTITY_H__BE769D85_6C72_43F4_88FB_5521CA66FC8E__INCLUDED_)
#define AFX_ENTITY_H__BE769D85_6C72_43F4_88FB_5521CA66FC8E__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

```
#include "..\inc\cadbase\cadbase.h"

class COpenGLDC;
class AFX_EXT_CLASS CEntity : public CObject
{
protected:
	BOOL		m_bModified;
	CBox3D*		m_pBox;
	UINT		m_id;
	CString		m_name;
	COLORREF	m_color;
	BOOL		m_bVisible;
	BOOL		m_bHighlight;

public:
	//constructor and destructor
	CEntity();
	virtual ~CEntity();

	//display
	virtual void Draw(COpenGLDC* pDC)=0;

	//serialize
	virtual	void Serialize(CArchive& ar);

	//attrib accessing
	BOOL	GetBox(CBox3D& box);

	void	SetID(UINT nID);
	UINT	GetID();

	void	SetName(LPCTSTR name);
	CString	GetName();

	void		SetColor(COLORREF color);
	COLORREF	GetColor();

	void	SetVisible(BOOL bVisible);
	BOOL	IsVisible();

	void	SetHighlight(BOOL bHighlight);
	BOOL	IsHighlight();

protected:
	virtual		void UpdateBox()=0;
};
```

```
//triangle chip
class AFX_EXT_CLASS CTriChip:public CObject
{
	DECLARE_SERIAL(CTriChip)
public:
	//attribs
	CPoint3D    vex[3];
	CVector3D normal;
public:
	//constructor && destructor
	CTriChip();
	CTriChip(const CPoint3D& v0,const CPoint3D& v1,const CPoint3D& v2,
const CVector3D& norm);
	virtual ~CTriChip();

	//display
	virtual void Draw(COpenGLDC* pDC);

	//serialize
	virtual void Serialize(CArchive& ar);

	//operator
	const CTriChip& operator=(const CTriChip&);
};

//CSTLModel
class AFX_EXT_CLASS CSTLModel: public CEntity
{
	DECLARE_SERIAL(CSTLModel)
//attribs
public:
	CTypedPtrArray<CObArray,CTriChip*> m_TriList;
public:
	//constructor && destructor
	CSTLModel();
	virtual ~CSTLModel();

	//display
	void  Draw(COpenGLDC* pDC);

	//serialize
	virtual voidSerialize(CArchive& ar);

	//operation
	const     CSTLModel& operator=(const CSTLModel&);

	void          Add(CTriChip* tri);
```

```
        BOOL        LoadSTLFile(LPCTSTR file);
        void  Clear();

        //attrib accessing
        BOOL        IsEmpty();

protected:
        virtual void UpdateBox();
};

class AFX_EXT_CLASS CPart : public CEntity
{
//attribs
protected:
        CTypedPtrArray<CObArray,CEntity*> m_EntList;
public:
        //constructor && destructor
        CPart();
        virtual ~CPart();

        //draw
        virtual void Draw(COpenGLDC* pDC);
        //serialize
        virtual void Serialize(CArchive& ar);

        //operation
        void  AddEntity(CEntity* ent);
        void  RemoveEntity(CEntity* ent);
        void  RemoveAllEntity();

        //attrib accessing
        BOOL        IsEmpty();
        UINTGetEntitySize();
        CEntity*    GetEntity(UINT i);

protected:
        virtual voidUpdateBox();
};

#endif // !defined(AFX_ENTITY_H__BE769D85_6C72_43F4_88FB_5521CA66FC8E__INCLUDED_)
```

6.5.2 文件 Entity.cpp

```
////////////////////////////// ENTITY.CPP ////////////////////////////////

#include "stdafx.h"
#include "..\inc\GeomKernel\Entity.h"
#include "..\inc\glContext\openGLDC.h"
```

```
#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[]=__FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CEntity::CEntity()
{
	m_bModified = FALSE;
	m_id = -1;
	m_pBox = NULL;
	m_color = RGB(128,128,128);
	m_name = _T("Unknow");
	m_bVisible = TRUE;
	m_bHighlight = FALSE;
}

CEntity::~CEntity()
{
}

void CEntity::Serialize(CArchive& ar)
{
	if(ar.IsStoring()){
		ar << m_id;
		ar << m_name;
		ar << m_color;
		ar << m_bVisible;
	}
	else{ //loading
		ar >> m_id;
		ar >> m_name;
		ar >> m_color;
		ar >> m_bVisible;
	}
}

BOOL	CEntity::GetBox(CBox3D& box)
{
	if(m_bModified)
		UpdateBox();
	if( m_pBox){
		box = *m_pBox;
		return TRUE;
	}
```

```
        else
                return FALSE;
}

void  CEntity::SetID(UINT nID)
{
        m_id = nID;
}

UINTCEntity::GetID()
{
        return m_id;
}

void  CEntity::SetName(LPCTSTR name)
{
        m_name = name;
}

CString      CEntity::GetName()
{
        return m_name;
}

void  CEntity::SetColor(COLORREF color)
{
        m_color = color;
}

COLORREF CEntity::GetColor()
{
        return m_color;
}

void CEntity::SetVisible(BOOL bVisible)
{
        m_bVisible = bVisible;
}

BOOL CEntity::IsVisible()
{
        return m_bVisible;
}
void CEntity::SetHighlight(BOOL bHighlight)
{
        m_bHighlight = bHighlight;
}

BOOL CEntity::IsHighlight()
```

```
{
	return m_bHighlight;
}

//////////////////////////
//class tri chip
IMPLEMENT_SERIAL(CTriChip,CObject,0)
CTriChip::CTriChip()
{
}

CTriChip::CTriChip(const CPoint3D& v0,const CPoint3D& v1,const CPoint3D& v2,
			const CVector3D& nor)
{
	vex[0] = v0;
	vex[1] = v1;
	vex[2] = v2;
	normal = nor;
}

CTriChip::~CTriChip()
{
}

const CTriChip& CTriChip::operator=(const CTriChip& tri)
{
	normal = tri.normal;
	for(int i=0;i<3;i++)
		vex[i] = tri.vex[i];
	return *this;
}

void CTriChip::Draw(COpenGLDC* pDC)
{
	pDC->DrawTriChip(normal.dx,normal.dy,normal.dz,
		vex[0].x,vex[0].y,vex[0].z,
		vex[1].x,vex[1].y,vex[1].z,
		vex[2].x,vex[2].y,vex[2].z);
}

void CTriChip::Serialize(CArchive& ar)
{
	if(ar.IsStoring()){
		ar << normal.dx << normal.dy << normal.dz;
		for(int i=0;i<3;i++)
			ar << vex[i].x << vex[i].y << vex[i].z;
	}
	else{
		ar >> normal.dx >> normal.dy >> normal.dz;
```

```
            for(int i=0;i<3;i++)
                ar >> vex[i].x >> vex[i].y >> vex[i].z;
        }
}

//////////////////////////////////////////
//class    CSTLModel

IMPLEMENT_SERIAL(CSTLModel,CObject,0)

CSTLModel::CSTLModel()
{
}

CSTLModel::~CSTLModel()
{
        Clear();
}

void CSTLModel::Add(CTriChip* tri)
{
        m_TriList.Add(tri);
}

void CSTLModel::UpdateBox()
{
        if(m_pBox){
                delete m_pBox;
                m_pBox = NULL;
        }
        if(m_TriList.GetSize()==0)
                return;

        double x0,y0,z0,x1,y1,z1;
        x0=y0=z0=10000;
        x1=y1=z1=-10000;
        CTriChip* tri;
        for(int i=0;i<m_TriList.GetSize();i++){
                tri = m_TriList[i];
                for(int n=0;n<3;n++){
                        if(tri->vex[n].x<x0)    x0 = tri->vex[n].x;
                        if(tri->vex[n].x>x1)    x1 = tri->vex[n].x;
                        if(tri->vex[n].y<y0)    y0 = tri->vex[n].y;
                        if(tri->vex[n].y>y1)    y1 = tri->vex[n].y;
                        if(tri->vex[n].z<z0)    z0 = tri->vex[n].z;
                        if(tri->vex[n].z>z1)    z1 = tri->vex[n].z;
                }
        }
```

```
	m_pBox = new CBox3D(x0,y0,z0,x1,y1,z1);
	m_bModified = FALSE;
}

//load with STL File
BOOL CSTLModel::LoadSTLFile(LPCTSTR stlfile)
{
	FILE* file;
	int type=0;
	if((file = fopen(stlfile, "r")) == NULL)
		return FALSE;

	char str[80];
	CTriChip* tri = NULL;
	while(fscanf(file,"%s",str)==1){
		if(strncmp(str,"normal",6)==0){
			tri = new CTriChip();
			fscanf(file,"%lf %lf %lf",&(tri->normal.dx),&(tri->normal.dy),&(tri->normal.dz));
			fscanf(file,"%*s %*s");
			fscanf(file,"%*s %lf %lf %lf",&(tri->vex[0].x),&(tri->vex[0].y),&(tri->vex[0].z));
			fscanf(file,"%*s %lf %lf %lf",&(tri->vex[1].x),&(tri->vex[1].y),&(tri->vex[1].z));
			fscanf(file,"%*s %lf %lf %lf",&(tri->vex[2].x),&(tri->vex[2].y),&(tri->vex[2].z));
			Add(tri);
		}
	}

	char title[80];
	if(GetFileTitle(stlfile,title,80)==0){
		SetName(title);
	}

	m_bModified = TRUE;
	return TRUE;
}
//Serialize
void  CSTLModel::Serialize(CArchive& ar)
{
	CEntity::Serialize(ar);

	if(ar.IsStoring()){
		m_TriList.Serialize(ar);
	}
	else{ //IsLoading()
		m_TriList.Serialize(ar);
		m_bModified = TRUE;
	}
}

void  CSTLModel::Draw(COpenGLDC* pDC)
```

```
{
	if(!m_bVisible)	return;

	if(m_bHighlight)
		pDC->Highlight(TRUE);
	else
		pDC->SetMaterialColor(m_color);

	if(pDC->IsSelectionMode()){
		pDC->LoadName((UINT)this);
	}

	for(int i=0;i<m_TriList.GetSize();i++){
		m_TriList[i]->Draw(pDC);
	}
}

void  CSTLModel::Clear()
{
	for(int i=0;i<m_TriList.GetSize();i++)
		delete m_TriList[i];
	m_TriList.RemoveAll();

	m_bModified = TRUE;
}

BOOL	CSTLModel::IsEmpty()
{
	return m_TriList.GetSize() == 0;
}

//////////////////////////////////////////
CPart::CPart()
{
}
CPart::~CPart()
{
	RemoveAllEntity();
}

//draw
void CPart::Draw(COpenGLDC* pDC)
{
	for(int i=0;i<m_EntList.GetSize();i++)
		m_EntList[i]->Draw(pDC);
}

//serialize
void CPart::Serialize(CArchive& ar)
```

```
{
	CEntity::Serialize(ar);
	if(ar.IsStoring()){
		m_EntList.Serialize(ar);
	}
	else{
		m_EntList.Serialize(ar);
		m_bModified = TRUE;
	}
}

//operation
void CPart::AddEntity(CEntity* ent)
{
	m_EntList.Add(ent);
	m_bModified = TRUE;
}

void CPart::RemoveEntity(CEntity* ent)
{
	for(int i=0;i<m_EntList.GetSize();i++){
		if(ent == m_EntList[i]){
			m_EntList.RemoveAt(i);
			m_bModified = TRUE;
			break;
		}
	}
}

void CPart::RemoveAllEntity()
{
	for(int i=0;i<m_EntList.GetSize();i++)
		delete m_EntList[i];
	m_EntList.RemoveAll();
	m_bModified = TRUE;
}

//attrib accessing
BOOL CPart::IsEmpty()
{
	return m_EntList.GetSize() == 0 ;
}

UINTCPart::GetEntitySize()
{
	return m_EntList.GetSize();
}

CEntity* CPart::GetEntity(UINT i)
```

```
{
	ASSERT(i<m_EntList.GetSize());
	return m_EntList[i];
}

void CPart::UpdateBox()
{
	if(m_pBox){
		delete m_pBox;
		m_pBox = NULL;
	}

	CBox3D box;
	for(int i=0;i<m_EntList.GetSize();i++){
		if(m_EntList[i]->GetBox(box)){
			if(m_pBox)
				*m_pBox += box;
			else{
				m_pBox = new CBox3D();
				*m_pBox = box;
			}
		}
	}
	m_bModified = FALSE;
}
```

本章相关程序

- ch6\GeomKernel：CAD **几何内核库的工程。**
- ch6\inc：GeomKernel **需要调用的头文件。**
- ch6\lib：GeomKernel **需要连接的静态库。**
- ch6\lib：GeomKernel **需要链接的动态库。**

第 7 章　CAD 应用程序 STLViewer 的模块化实现

本章要点：

- **在已开发模块的基础上设计** STLViewer **的应用程序框架。**
- **使用几何内核库** GeomKernel.dll **实现模型的创建、存储与读取。**
- **使用图形工具库** glContext.dll **实现模型的绘制与显示操作。**

在本书第 2、5、6 **章中，分别介绍了几何工具库（**GeomCalc.dll**）、图形工具库（**glContext.dll**）、几何内核库（**GeomKernel.dll**）三个模块的开发。这些模块开发完成之后，在它们基础上搭建主执行程序** STLViewer **就是一件不太复杂的工作了。实现主程序** STLViewer **本身不需要编写太多的代码，它主要是负责界面消息的处理、对动态库的链接与相关功能的调用。这一章中，在以上开发的三个动态库的基础上，我们将建立这个可执行的应用程序** STLViewer.exe**。在** STLViewer.exe **中，可通过输入** STL **文件构造几何模型，并将这些模型存储成系统自己的** mdl **文档文件；对窗口中的几何模型，可以使用放大、缩小、视角变换、着色模式切换等方式进行操作。图** 7-1 **显示的** STLViewer **中的模型是分别使用** STL **文件读入接口调入的** 5 **个几何零件，并使用侧视图来观察几何模型。**

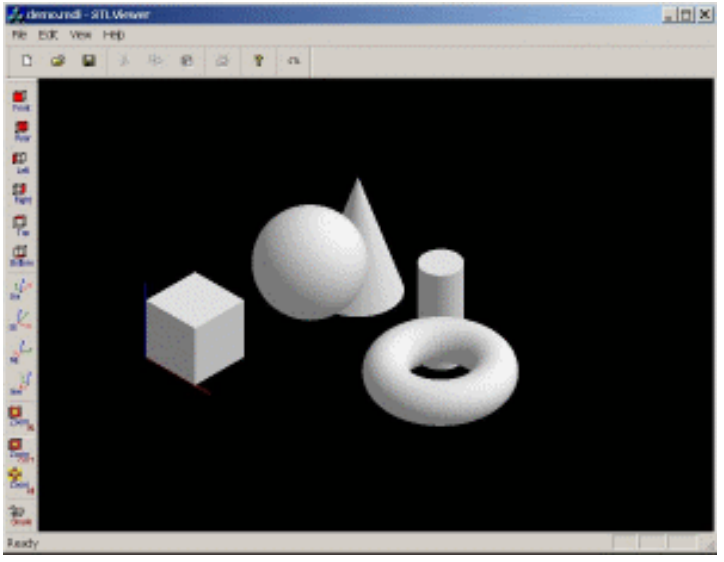


图 7-1　**用侧视图观察几何模型**

7.1　STLViewer 的模块结构

图 7-2 **显示的是** STLViewer **的模块层次结构。**STLViewer.exe **运行时，需要动态链接**

GeomKernel.dll、glContext.dll、GeomCalc.dll 三个动态库以及 MFC 和 OpenGL 的动态库。在 GeomKernel.dll、glContext.dll、GeomCalc.dll 三个动态库之间也存在动态链接关系。由于这些库和执行程序都是在 MFC 基础上开发的，因而都需要 MFC 的动态链接库。STLViewer.exe 在运行时才动态调用这些 DLL 库，因为绝大多数功能代码都在这些库中实现，所以 STLViewer.exe 本身文件尺寸不大。

被链接的 DLL 库必须和 STLViewer.exe 位于相同的目录下，或位于 Windows 系统已设定的搜索路径下。在 Window 系统安装 Visual C++时，已经将 MFC 的有关 DLL 库拷贝到了 Windows 系统的 System32 子目录下（如 C:\WINNT\System32），且默认状态下该目录已经被系统设置为搜索路径。

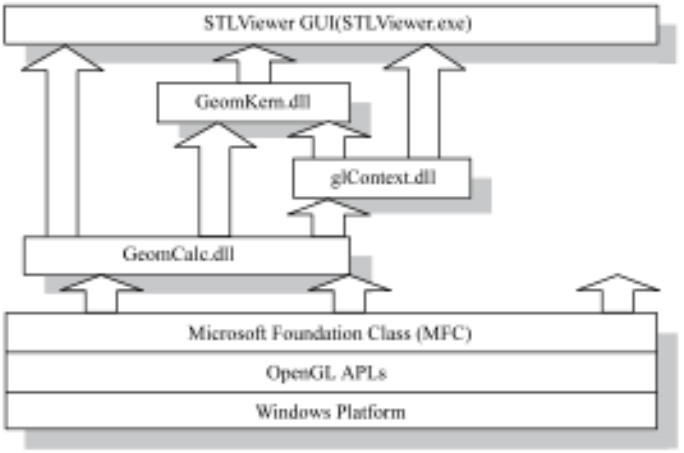


图 7-2　STLViewer 的模块结构图

7.2　创建应用程序框架

由于 STLViewer 应用程序的创建除了应用程序工程本身的文件之外，还需要其他组成模块的文件，因此有必要创建一个目录，将这些文件目录组织起来。在附带的光盘上，这个目录名为 ch7。

启动 Visual C++，使用 AppWizard 建立一个 MFC Wizard 工程，该工程的名称为 STLViewer，使用单文档模式，建立在目录 ch7 下。

需要注意的是，AppWizard 的第 4 步中，需要在 Advanced Option 选项中给 STLViewer.exe 的文档起个后缀名。如图 7-3 所示，给文档定义后缀名为 mdl，STLViewer.exe 使用串行化存储/读取的文档，将以 mdl 为后缀。

AppWizard 自动创建的 STLViewer 程序框架如图 7-4 所示。

STLViewer 程序框架创建之后，需要完成如下工作：

（1）建立文件目录结构。虽然 STLViewer.exe 执行时只需要相关的几个 DLL 库，但工程 STLViewer 在编译、连接时需要这些 DLL 的静态连接库（*.lib）和说明它们的头文件（*.h）。我们需要把这些相关的文件及其目录拷贝到工程 STLViewer 的同一个目录下，以方便使用。

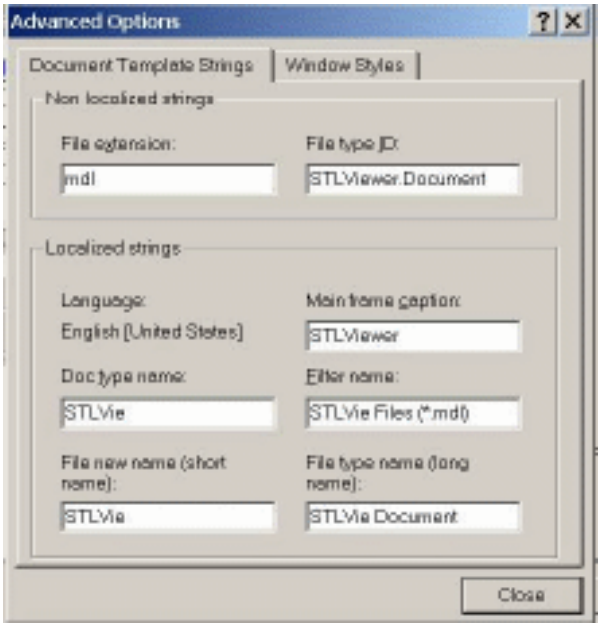


图 7-3　给 STLViewer 的文档定义后缀名

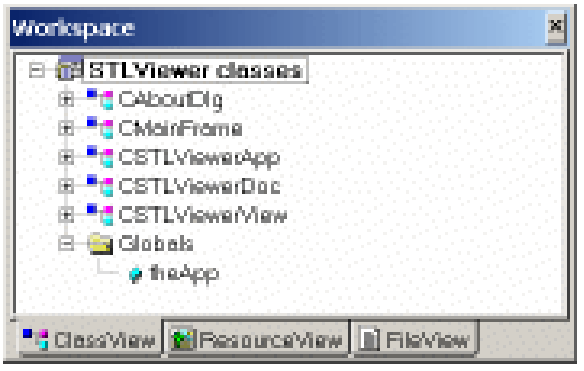


图 7-4　工程 STLViewer 下的类结构

如图 7-5 所示，是为创建应用程序 STLViewer 建立的文件目录结构。在该结构中，将 STLViewer.exe 执行时需要链接的 DLL 库与执行文件 STLViewer.exe 存在同一个目录 bin 下，使 STLViewer.exe 在运行时能够直接找到 DLL 库。在默认状态下，工程 STLViewer 所生成的执行文件存于 STLViewer\Debug 子目录下，需要修改这个默认设置。

单击 Project 菜单中的 Setting 命令，打开 Project Settings 对话框，单击 Link 文件属性页，如图 7-6 所示，在 Output file name 栏中，将路径由“Debug\STLViewer.exe”改为“..\bin\STLViewer.exe”。

（2）载入静态连接文件。在 Workspace 的 FileView 中，用鼠标右键单击 STLViewer files，如图 7-7 所示，在弹出的菜单中使用 Add Files to Project 将 lib 目录下的三个静态连接文件 GeomCalc.Lib、GeomKernel.Lib 和 glContext.Lib 加入工程。

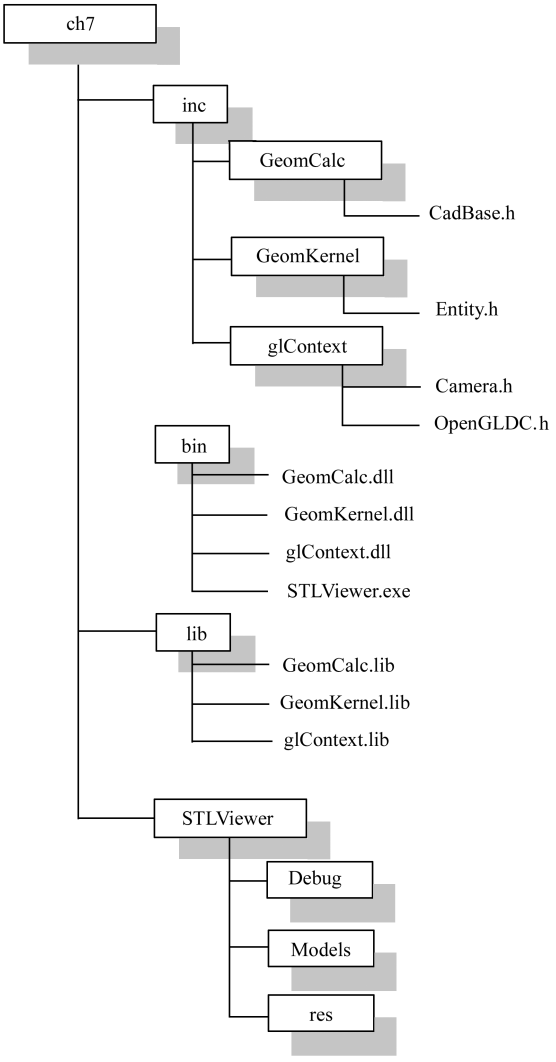


图 7-5　为工程 STLViewer 建立的目录结构

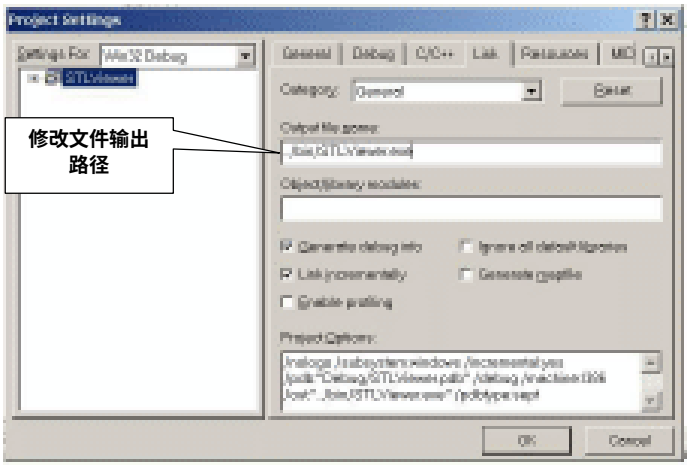


图 7-6　修改工程 STLViewer 的输出文件路径

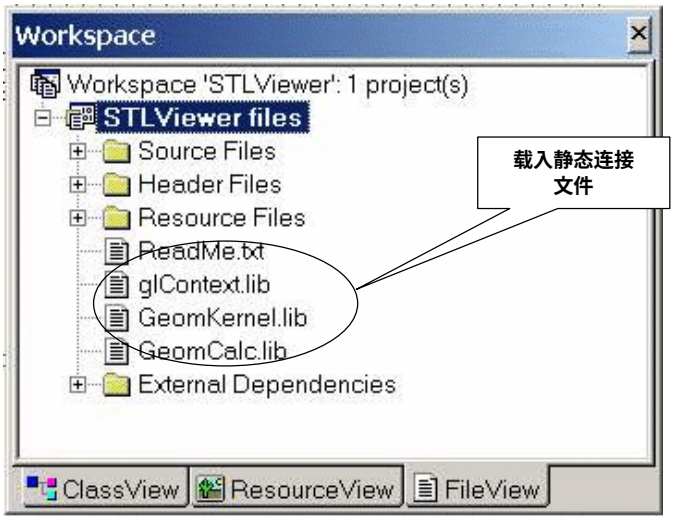


图 7-7　在工程 STLViewer 中载入静态连接文件

（3）编译并连接整个工程。完成以上工作以后，编译连接整个工程将生成可执行文件 STLViewer.exe**，并输出于** bin **子目录下。这时的可执行文件在功能上还是一片空白，需要对它进行修改：在主框架类** CMainFrame **中增加界面对象；在文档类** CSTLViewerDoc **中增加几何模型对象和对文档的操作；在视图类** CSTLViewerView **中增加对几何模型的显示与操作功能。**

7.3　修改应用程序框架

7.3.1　增加界面资源

在工程 STLViewer **的资源文件中，分别增加和修改工具条资源和菜单资源。**

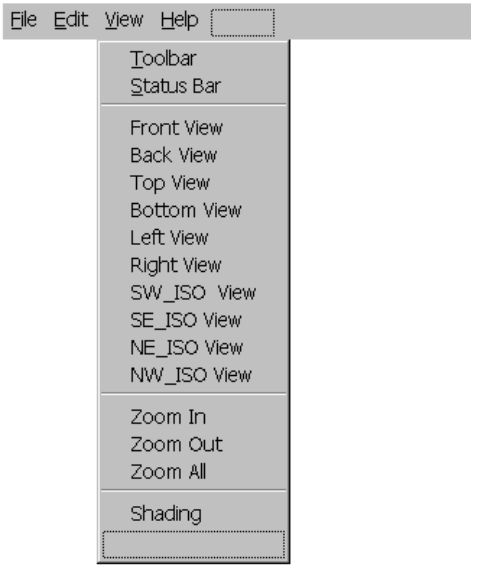
（1）修改工具条资源 IDR_MAINFRAME**，在其中增加一个按钮** STL**，用于** STL **文件读入，并设置** ID **号为** ID_STL_FILEIN**。**

（2）增加工具条资源 IDR_TOOLBAR_DISPLAY**。**

（3）修改菜单资源 IDR_MAINFRAME**，并对菜单/按钮事件设置** ID **号。**

Front View:	ID_VIEW_FRONT
Back View:	ID_VIEW_BACK
Top View:	ID_VIEW_TOP
Bottom View:	ID_VIEW_BOTTOM
Left View:	ID_VIEW_LEFT

Right View:	ID_VIEW_RIGHT
SW_ISO View:	ID_VIEW_SW_ISOMETRIC
SE_ISO View:	ID_VIEW_SE_ISOMETRIC
NE_ISO View:	ID_VIEW_NE_ISOMETRIC
NW_ISO View:	ID_VIEW_NW_ISOMETRIC
Zoom In:	ID_VIEW_ZOOMIN
Zoom Out:	ID_VIEW_ZOOMOUT
Zoom All:	ID_VIEW_ZOOMALL
Shading:	ID_VIEW_SHADE



7.3.2 修改框架类 CMainFrame

在主框架类中，AppWizard **已经自动创建了一个系统工具条和状态条。我们需要在其中加入一个用于模型显示与操作的工具条** m_wndDisplayBar**。**

修改类 CMainFrame **的定义如下：**

```
class CMainFrame : public CFrameWnd
{
        ……
protected:    // control bar embedded members
        CStatusBar        m_wndStatusBar;
        CToolBar          m_wndToolBar;
        CToolBar          m_wndDisplayBar;      //插入显示工具条
        ……
};
```

需要在 CMainFrame::OnCreate()**中创建这个工具条，并修改代码如下：**

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;          // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;          // fail to create
    }

    //创建显示工具条 m_wndDisplayBar
    if(!m_wndDisplayBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE
    |CBRS_TOP|CBRS_GRIPPER|CBRS_TOOLTIPS|CBRS_FLYBY | CBRS_SIZE_DYNAMIC)
    ||!m_wndDisplayBar.LoadToolBar(IDR_TOOLBAR_DISPLAY))
    {
        TRACE0("Failed to create display toolbar\n");
        return -1;          // fail to create
    }

    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);

    //设置 m_wndDisplayBar 的停靠属性
    m_wndDisplayBar.EnableDocking(CBRS_ALIGN_ANY);

    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    //将 m_wndDisplayBar 停靠在框架窗口的左边
    DockControlBar(&m_wndDisplayBar,AFX_IDW_DOCKBAR_LEFT);
    return 0;
}
```

7.3.3 修改文档类 CSTLViewerDoc

文档类用于存放和管理几何模型对象，在 STLViewer **中，这个模型是类** CPart **的一个实例。需要在文档类中插入一个** CPart **的对象，用于模型管理。类** CPart **由动态库** GeomKernel.dll **输出，在头文件** GeomKernel\Entity.h **中说明。一个** CPart **模型中可以包含多个** CEntity **派生类**

的对象。对文档类 CSTLViewerDoc 的修改如下：

（1）在 STLViewerDoc.h 文件中加入说明 CPart 的头文件。

```
#include "..\inc\GeomKernel\Entity.h"
```

（2）在 CSTLViewerDoc 中插入一个 CPart 的对象 m_Part。

```
public:
    CPart       m_Part;
```

（3）增加事件 ID_STL_FILEIN 的映射函数。

```
……
//{{AFX_MSG(CSTLViewerDoc)
afx_msg void OnStlFilein();
//}}AFX_MSG
……
```

在函数 OnStlFilein()中，读入一个 STL 文件，并用它创建一个 CSTLModel 的对象。对象创建成功后，将这个对象插入模型对象 m_Part 中，作为 m_Part 中的一个子模型。函数实现如下：

```
void CSTLViewerDoc::OnStlFilein()
{
    //用文件对话框打开一个以 STL 为后缀的文件
    CFileDialog dlg(TRUE,"stl",NULL,
        OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
        "Stereo Lithograpic File(*.stl)|*.stl", NULL );

    if(dlg.DoModal()==IDOK){
        //创建一个空的 CSTLModel 的对象
        CSTLModel* pSTLModel = new CSTLModel();

        //在对象 pSTLModel 中载入文件内的 STL 模型
        CString strName = dlg.GetPathName();
        pSTLModel->LoadSTLFile(strName);

        if(pSTLModel->IsEmpty()) //如果 STL 模型为空，释放 pSTLModel 对象
            delete pSTLModel;
        else   //如果 STL 模型不为空，加入 pSTLModel 对象到 m_Part 中
            m_Part.AddEntity(pSTLModel);

        UpdateAllViews(NULL); //更新所有与文档关联的视图，以显示新载入的模型
    }
}
```

（4）修改函数 CSTLViewerDoc::OnNewDocument()。

当用户使用菜单命令 File→New 创建一个新文档时，函数 OnNewDocument()被应用程序框架自动调用。在 OnNewDocument()中可以对新建的文档做必要的初始化。尤其在 SDI 应用

程序中，当用户使用 File→New 创建一个新文档时，与 MDI 应用程序不同的是，应用程序框架并不重新创建一个新的文档对象，而是使用 OnNewDocument()对已存在的文档对象重新初始化。所以，需要在函数 CSTLViewerDoc::OnNewDocument() 中对 m_Part 做初始化，即将对象中的内容清空。

```
BOOL CSTLViewerDoc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    m_Part.RemoveAllEntity();    //清空 m_Part 中的所有子模型
    return TRUE;
}
```

（5）修改文档串行化函数 CSTLViewerDoc::Serialize()，在串行化操作中插入对 m_Part 的串行化。

```
void CSTLViewerDoc::Serialize(CArchive& ar)
{
        ……
    m_Part.Serialize(ar);
}
```

7.3.4 修改视图类 CSTLViewerView

类 CGLView 是一个可完全重用的支持 OpneGL 的视图类。直接将类 CGLView 放到一个应用程序中并不是设计这个类的目标。在开发应用程序时，应该根据需要在 CGLView 的基础上派生定制一个自己的视图类。CSTLViewerView 就是这样一个派生的视图类。AppWizard 已经自动生成了类 CSTLViewerView，它使用 CView 作为基类。为了在该视图中使用 OpenGL 绘制图形，需要将 CSTLViewerView 的基类改为 CGLView，并在几处做相应的修改；并且为了响应界面的相关操作，还需要在 CSTLViewerView 中增加一些响应函数。修改步骤如下：

（1）在 STLViewerView.h 文件中加入说明 CGLView 的头文件。

```
#include "..\inc\glContext\openGLDC.h"
```

（2）在文件 STLViewerView.h、STLViewerView.cpp 中使用类 CGLView 替换 CView。

（3）注释掉 CSTLViewerView 中的虚拟函数 OnDraw()，这样 CSTLViewerView 将使用基类 CGLView 中的虚拟函数 OnDraw()。在 CGLView::OnDraw()中会调用相应的虚拟函数 RenderScene()进行场景绘制。

（4）实现类 CSTLViewerView 自己的虚拟函数 RenderScene()，在该函数中定义应用程序所要绘制的场景，即 m_Part 对象的 OpenGL 绘制。如果不在 CSTLViewerView 中实现自己的 RenderScene()函数，CSTLViewerView 将使用基类 CGLView 中的 RenderScene()函数，不绘制

任何景物。

```
void CSTLViewerView::RenderScene(COpenGLDC* pDC)
{
    CSTLViewerDoc* pDoc = GetDocument();
    ASSERT(pDoc);
    pDC->DrawCoord();     //绘制一个坐标系框架

    if(!pDoc->m_Part.IsEmpty())   //如果模型 m_Part 不为空
        pDoc->m_Part.Draw(pDC);  //用 OpenGL 绘制模型 m_Part
}
```

（5）实现类 CSTLViewerView 自己的虚拟函数 GetBox()，在 GetBox()中将访问文档中的模型对象 m_Part，获取并返回 m_Part 的最大包容盒信息。如果模型为空，函数返回 FALSE。

```
BOOL CSTLViewerView::GetBox(double& x0,double& y0,double& z0,
                            double& x1,double& y1,double& z1)
{
    CSTLViewerDoc* pDoc = GetDocument();
        ASSERT(pDoc);

    if(!pDoc->m_Part.IsEmpty()){ //如果模型不为空，提取包容盒信息
        CBox3D box;
        if(pDoc->m_Part.GetBox(box)){
            x0 = box.x0; y0 = box.y0;    z0 = box.z0;
            x1 = box.x1; y1 = box.y1;    z1 = box.z1;
            return TRUE;
        }
    }
    return FALSE; //模型为空
}
```

（6）实现菜单、工具条按钮的命令处理函数。由于在基类 CGLView 中已经定义了如典型视图切换、显示缩放、着色模式切换等函数，类 CSTLViewerView 的主要任务是处理用户的命令输入，将命令处理函数与基类中的相关功能对应起来。这些命令处理函数如下：

1）命令 ID_VIEW_BACK 的处理函数

```
afx_msg void OnViewBack();
void CSTLViewerView::OnViewBack()
{
    OnViewType(VIEW_BACK);      //模型的后视图
}
```

2）命令 ID_VIEW_BOTTOM 的处理函数

```
afx_msg void OnViewBottom();
void CSTLViewerView::OnViewBottom()
{
    OnViewType(VIEW_BOTTOM);        //模型的底视图
}
```

3）命令 ID_VIEW_FRONT 的处理函数

```
afx_msg void OnViewFront();
void CSTLViewerView::OnViewFront()
{
    OnViewType(VIEW_FRONT);          //模型的前视图
}
```

4）命令 ID_VIEW_LEFT 的处理函数

```
afx_msg void OnViewLeft();
void CSTLViewerView::OnViewLeft()
{
    OnViewType(VIEW_LEFT);       //模型的左视图
}
```

5）命令 ID_VIEW_RIGHT 的处理函数

```
afx_msg void OnViewRight();
void CSTLViewerView::OnViewRight()
{
    OnViewType(VIEW_RIGHT);      //模型的右视图
}
```

6）命令 ID_VIEW_TOP 的处理函数

```
afx_msg void OnViewTop();
void CSTLViewerView::OnViewTop()
{
    OnViewType(VIEW_TOP);        //模型的顶视图
}
```

7）命令 ID_VIEW_SW_ISOMETRIC 的处理函数

```
afx_msg void OnViewSWIsometric();
void CSTLViewerView::OnViewSWIsometric()
{
    OnViewType(VIEW_SW_ISOMETRIC);//模型的 SW 侧视图
}
```

8）命令 ID_VIEW_SE_ISOMETRIC 的处理函数

```
afx_msg void OnViewSEIsometric();
void CSTLViewerView::OnViewSEIsometric()
{
    OnViewType(VIEW_SE_ISOMETRIC);//模型的 SE 侧视图
}
```

9）命令 ID_VIEW_NE_ISOMETRIC 的处理函数

```
afx_msg void OnViewNEIsometric();
void CSTLViewerView::OnViewNEIsometric()
```

```
{
    OnViewType(VIEW_NE_ISOMETRIC);//模型的 NE 侧视图
}
```

10）命令 ID_VIEW_NW_ISOMETRIC 的处理函数

```
afx_msg void OnViewNWIsometric();
void CSTLViewerView::OnViewNWIsometric()
{
    OnViewType(VIEW_NW_ISOMETRIC);   //模型的 NW 侧视图
}
```

11）命令 ID_VIEW_ZOOMALL 的处理函数

```
afx_msg void OnViewZoomall();
void CSTLViewerView::OnViewZoomall()
{
    ZoomAll();        //将模型显示在整个窗口内
}
```

基类 CGLView 中调用了虚拟函数 GetBox()以获取当前模型的包容盒信息，并根据包容盒的尺寸修改场景尺寸，以将整个模型包括在场景中。实现这个功能的关键是，在 CGLView 派生的视图类中要实现一个自己的 GetBox()虚拟函数。

12）命令 ID_VIEW_ZOOMIN 的处理函数

```
afx_msg void OnViewZoomin();
void CSTLViewerView::OnViewZoomin()
{
    Zoom(0.9);   //将视景区域缩小为原来的 0.9，模型在屏幕上的实际显示尺寸放大
}
```

13）命令 ID_VIEW_ZOOMOUT 的处理函数

```
afx_msg void OnViewZoomout();
void CSTLViewerView::OnViewZoomout()
{
    Zoom(1.1);   //将视景区域放大至 1.1 倍，模型在屏幕上的实际显示尺寸缩小
}
```

14）命令 ID_VIEW_SHADE 的处理函数

```
afx_msg void OnViewShade();
void CSTLViewerView::OnViewShade()
{
    m_pGLDC->Shading(!m_pGLDC->IsShading()); //打开或关闭 OpenGL 的着色处理
    Invalidate();
}
```

（7）增加键盘消息 WM_KEYDOWN 的处理函数 OnKeyDown()。可以使用键盘的光标健上下左右平移场景。在基类 CGLView 中已经设计了平移场景的成员函数 MoveView()，类 CSTLViewerView 的主要任务是处理用户的键盘消息，将光标键的输入与函数

CGLView::MoveView()**对应起来，以实现光标键控制的场景平移。函数** OnKeyDown()**的实现如下：**

```
void CSTLViewerView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    switch(nChar){
    case VK_UP:
        MoveView(0.0,0.02);          //向上移动场景 2%
        break;
    case VK_DOWN:
        MoveView(0.0,-0.02);         //向下移动场景 2%
        break;
    case VK_RIGHT:
        MoveView(0.02,0);            //向右移动场景 2%
        break;
    case VK_LEFT:
        MoveView(-0.02,0);           //向左移动场景 2%
        break;
    }
    CGLView::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

修改后 CSTLViewerView **的定义如下：**

```
#include "..\inc\glContext\openGLDC.h"          //说明基类 CGLView
class CSTLViewerView : public CGLView           //在基类 CGLView 上派生
{
protected: // create from serialization only
    CSTLViewerView();
    DECLARE_DYNCREATE(CSTLViewerView)

// Attributes
public:
    CSTLViewerDoc* GetDocument();
    //重新定义虚拟函数，实现多态性
    virtual void RenderScene(COpenGLDC* pDC);

// Operations
public:
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CSTLViewerView)
    public:
    //virtual void OnDraw(CDC* pDC);   //注释掉此函数，使用基类的 OnDraw()函数
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

```
        protected:
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CSTLViewerView();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:
// Generated message map functions
protected:
        //{{AFX_MSG(CSTLViewerView)
        //视图选择的命令处理函数
        afx_msg void OnViewBack();
        afx_msg void OnViewBottom();
        afx_msg void OnViewFront();
        afx_msg void OnViewLeft();
        afx_msg void OnViewRight();
        afx_msg void OnViewTop();
        afx_msg void OnViewSWIsometric();
        afx_msg void OnViewSEIsometric();
        afx_msg void OnViewNEIsometric();
        afx_msg void OnViewNWIsometric();

        //全局缩放函数
        afx_msg void OnViewZoomall();
        //模型显示放大
        afx_msg void OnViewZoomin();
        //模型显示缩小
        afx_msg void OnViewZoomout();
        //渲染/非渲染切换
        afx_msg void OnViewShade();
        //响应键盘消息
        afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

        //实现类自己的虚拟函数 GetBox()
        virtual BOOL      GetBox(double& x0,double& y0,double& z0,
                double& x1,double& y1,double& z1);
```

};

7.4 运行 STLViewer.exe

完成以上步骤后，对工程 STLViewer 进行编译连接，将在目录 bin 下生成执行文件 STLViewer.exe。运行这个文件，界面如图 7-1 所示。所不同的是，由于还没有调入模型，所以窗口中没有模型显示。

1．输入 STL 模型

在本书附带光盘的 ch7\Models 子目录下给出了几个现成的 STL 文件，可以通过分别读入这些文件来构造一个模型。在 ToolBar 中，使用 STL File In 按钮来输入 STL 模型。每读入一个文件，STL 模型将显示在视图窗口上。

STL 作为一个三维形状文件接口，很多 CAD 软件都可以将实体转化成 STL 模型输出。本例中的几个 STL 模型由 AutoCAD R14 输出，ASCII 格式。读者自己也可以在商品化的 CAD 软件中输出一些 STL 模型，并尝试使用 STLViewer.exe 来读入并观察模型。图 7-1 显示的是读入了 ch7\Models 子目录下的几个 STL 文件后构造的模型。

2．存储 STLViewer 自己的文档(*.mdl)

在类 CSTLViewerDoc 中，设计了文档的串行化功能，即可以把文档中的数据用串行化方法存储成二进制文件，并还可以用同样的串行化顺序再读入文件中的数据和恢复模型。

在已经输入了若干个 STL 子模型之后，文档中的模型对象 m_Part 已经不再为空，而是包括了几个由 CSTLModel 对象组成的子模型。这时可以使用 File 菜单下的 Save 或 Save As 命令，将文档内容存储成一个以 mdl 为后缀的文件。在本书附带光盘的 ch7\Models 子目录下的 demo.mdl 文件就是这样一个文件。

使用 File 菜单下的 Open 命令，可以直接读入 mdl 文件，恢复模型。

3．模型显示缩放

使用工具条上的 Zoom All 可以在窗口中显示全部模型。使用 Zoom In、Zoom Out 可以分别放大和缩小模型的显示。

4．使用键盘平移场景

使用键盘的上下左右光标键，可以在窗口内平移场景。视图窗口中，模型和坐标系框架将随着键盘光标键而上下左右移动。

5．模型视图切换

使用工具条上的视图切换按钮，可以从不同的典型角度来观察模型。图 7-1 所显示的是使用 SE Isometric View 观察到的模型。

6．模型着色模式切换

使用着色模式切换按钮，可以切换着色显示方式，即选择使用着色方式来显示模型或只用网格来显示模型。图 7-8 所示的是使用网格来显示模型的情况。

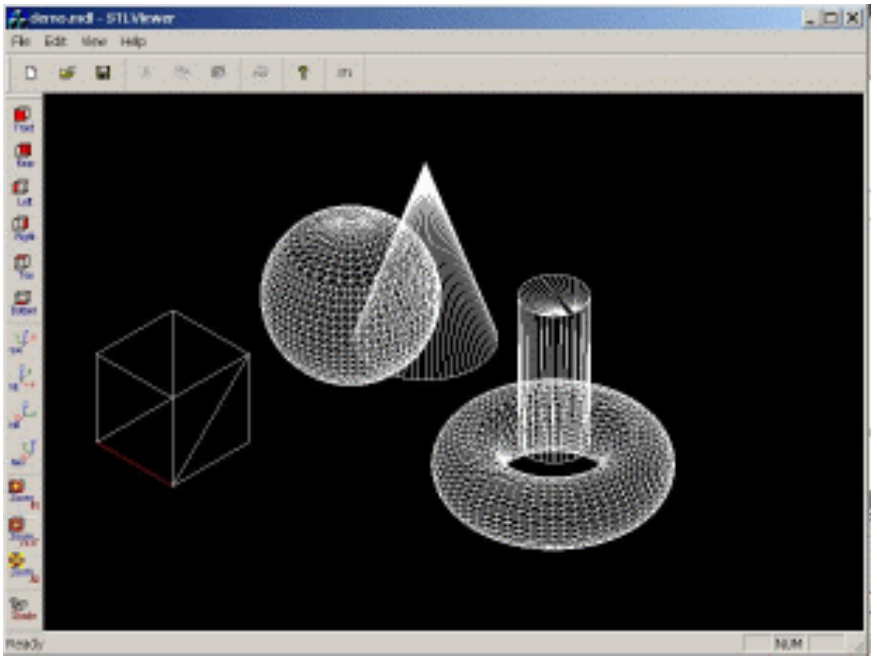


图 7-8　使用网格来显示模型

7.5　源程序清单

下面给出 CMainFrame、CSTLViewerDoc、CSTLViewerView **三个类的源程序清单，供参考。**

7.5.1　文件 MainFrm.h

```
class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
        CMainFrame();
        DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMainFrame)
        virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
```

```
        //}}AFX_VIRTUAL

// Implementation
public:
        virtual ~CMainFrame();
#ifdef _DEBUG
        virtual void AssertValid() const;
        virtual void Dump(CDumpContext& dc) const;
#endif

protected:    // control bar embedded members
        CStatusBar    m_wndStatusBar;
        CToolBar        m_wndToolBar;
        CToolBar   m_wndDisplayBar;

// Generated message map functions
protected:
        //{{AFX_MSG(CMainFrame)
        afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
                // NOTE - the ClassWizard will add and remove member functions here.
                // DO NOT EDIT what you see in these blocks of generated code!
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

7.5.2 文件 MainFrm.cpp

```
// MainFrm.cpp : implementation of the CMainFrame class

#include "stdafx.h"
#include "STLViewer.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNCREATE(CMainFrame, CFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CFrameWnd)
        //{{AFX_MSG_MAP(CMainFrame)
```

```
        // NOTE - the ClassWizard will add and remove mapping macros here.
        // DO NOT EDIT what you see in these blocks of generated code !
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{
    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here

}

CMainFrame::~CMainFrame()
{
}

int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;      // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;      // fail to create
    }
```

```
    if(!m_wndDisplayBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndDisplayBar.LoadToolBar(IDR_TOOLBAR_DISPLAY))
    {
        TRACE0("Failed to create display toolbar\n");
        return -1;          // fail to create
    }

    // TODO: Delete these three lines if you don't want the toolbar to
    //    be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    m_wndDisplayBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);
    DockControlBar(&m_wndDisplayBar,AFX_IDW_DOCKBAR_LEFT);
    return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // TODO: Modify the Window class or styles here by modifying
    //    the CREATESTRUCT cs

    return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
    CFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
    CFrameWnd::Dump(dc);
}

#endif //_DEBUG
```

7.5.3 文件 STLViewerDoc.h

```
#include "..\inc\GeomKernel\Entity.h"
```

```
class CSTLViewerDoc : public CDocument
{
protected: // create from serialization only
	CSTLViewerDoc();
	DECLARE_DYNCREATE(CSTLViewerDoc)

// Attributes
public:
	CPartm_Part;

// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CSTLViewerDoc)
	public:
	virtual BOOL OnNewDocument();
	virtual void Serialize(CArchive& ar);
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CSTLViewerDoc();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CSTLViewerDoc)
	afx_msg void OnStlFilein();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};
```

7.5.4 文件 STLViewerDoc.cpp

```
// STLViewerDoc.cpp : implementation of the CSTLViewerDoc class

#include "stdafx.h"
#include "STLViewer.h"

#include "STLViewerDoc.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerDoc

IMPLEMENT_DYNCREATE(CSTLViewerDoc, CDocument)

BEGIN_MESSAGE_MAP(CSTLViewerDoc, CDocument)
	//{{AFX_MSG_MAP(CSTLViewerDoc)
	ON_COMMAND(ID_STL_FILEIN, OnStlFilein)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerDoc construction/destruction

CSTLViewerDoc::CSTLViewerDoc()
{
	// TODO: add one-time construction code here
}

CSTLViewerDoc::~CSTLViewerDoc()
{
}

BOOL CSTLViewerDoc::OnNewDocument()
{
	if (!CDocument::OnNewDocument())
		return FALSE;

	m_Part.RemoveAllEntity();
	return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerDoc serialization

void CSTLViewerDoc::Serialize(CArchive& ar)
{
	if (ar.IsStoring())
	{
	}
	else
```

```
	{
	}
	m_Part.Serialize(ar);
}

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerDoc diagnostics

#ifdef _DEBUG
void CSTLViewerDoc::AssertValid() const
{
	CDocument::AssertValid();
}

void CSTLViewerDoc::Dump(CDumpContext& dc) const
{
	CDocument::Dump(dc);
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerDoc commands

void CSTLViewerDoc::OnStlFilein()
{
	CFileDialog dlg(TRUE,"stl",NULL,
		OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
		"Stereo Lithograpic File(*.stl)|*.stl", NULL );

	if(dlg.DoModal()==IDOK){
		CSTLModel* pSTLModel = new CSTLModel();

		CString strName = dlg.GetPathName();
		pSTLModel->LoadSTLFile(strName);

		if(pSTLModel->IsEmpty())
			delete pSTLModel;
		else
			m_Part.AddEntity(pSTLModel);
		UpdateAllViews(NULL);
	}
}
```

7.5.5 文件 STLViewerView.h

```
#include "..\inc\glContext\openGLDC.h"
```

```
class CSTLViewerView : public CGLView
{
protected: // create from serialization only
	CSTLViewerView();
	DECLARE_DYNCREATE(CSTLViewerView)

// Attributes
public:
	CSTLViewerDoc* GetDocument();

	virtual void RenderScene(COpenGLDC* pDC);
// Operations
public:

// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CSTLViewerView)
	public:
	virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
	protected:
	//}}AFX_VIRTUAL

// Implementation
public:
	virtual ~CSTLViewerView();
#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CSTLViewerView)
	afx_msg void OnViewBack();
	afx_msg void OnViewBottom();
	afx_msg void OnViewFront();
	afx_msg void OnViewLeft();
	afx_msg void OnViewRight();
	afx_msg void OnViewTop();
	afx_msg void OnViewSWIsometric();
	afx_msg void OnViewSEIsometric();
	afx_msg void OnViewNEIsometric();
	afx_msg void OnViewNWIsometric();
	afx_msg void OnViewZoomall();
	afx_msg void OnViewZoomin();
```

```
        afx_msg void OnViewZoomout();
        afx_msg void OnViewShade();
afx_msg void OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()

        virtual BOOL      GetBox(double& x0,double& y0,double& z0,
                                 double& x1,double& y1,double& z1);
};

#ifndef _DEBUG    // debug version in STLViewerView.cpp
inline CSTLViewerDoc* CSTLViewerView::GetDocument()
    { return (CSTLViewerDoc*)m_pDocument; }
#endif
```

7.5.6 文件 STLViewerView.cpp

```
#include "stdafx.h"
#include "STLViewer.h"

#include "STLViewerDoc.h"
#include "STLViewerView.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerView

IMPLEMENT_DYNCREATE(CSTLViewerView, CGLView)

BEGIN_MESSAGE_MAP(CSTLViewerView, CGLView)
        //{{AFX_MSG_MAP(CSTLViewerView)
        ON_COMMAND(ID_VIEW_BACK, OnViewBack)
        ON_COMMAND(ID_VIEW_BOTTOM, OnViewBottom)
        ON_COMMAND(ID_VIEW_FRONT, OnViewFront)
        ON_COMMAND(ID_VIEW_LEFT, OnViewLeft)
        ON_COMMAND(ID_VIEW_RIGHT, OnViewRight)
        ON_COMMAND(ID_VIEW_TOP, OnViewTop)
        ON_COMMAND(ID_VIEW_SW_ISOMETRIC, OnViewSWIsometric)
        ON_COMMAND(ID_VIEW_SE_ISOMETRIC, OnViewSEIsometric)
        ON_COMMAND(ID_VIEW_NE_ISOMETRIC, OnViewNEIsometric)
        ON_COMMAND(ID_VIEW_NW_ISOMETRIC, OnViewNWIsometric)
        ON_COMMAND(ID_VIEW_ZOOMALL, OnViewZoomall)
```

```
	ON_COMMAND(ID_VIEW_ZOOMIN, OnViewZoomin)
	ON_COMMAND(ID_VIEW_ZOOMOUT, OnViewZoomout)
	ON_COMMAND(ID_VIEW_SHADE, OnViewShade)
	//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerView construction/destruction

CSTLViewerView::CSTLViewerView()
{
	// TODO: add construction code here

}

CSTLViewerView::~CSTLViewerView()
{
}

BOOL CSTLViewerView::PreCreateWindow(CREATESTRUCT& cs)
{
	// TODO: Modify the Window class or styles here by modifying
	//  the CREATESTRUCT cs

	return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerView drawing
/////////////////////////////////////////////////////////////////////////////
// CSTLViewerView diagnostics

#ifdef _DEBUG
void CSTLViewerView::AssertValid() const
{
	CView::AssertValid();
}

void CSTLViewerView::Dump(CDumpContext& dc) const
{
	CView::Dump(dc);
}
CSTLViewerDoc* CSTLViewerView::GetDocument() // non-debug version is inline
{
	ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CSTLViewerDoc)));
	return (CSTLViewerDoc*)m_pDocument;
}
```

```
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CSTLViewerView message handlers
void CSTLViewerView::RenderScene(COpenGLDC* pDC)
{
	CSTLViewerDoc* pDoc = GetDocument();
	ASSERT(pDoc);

	pDC->DrawCoord();

	if(!pDoc->m_Part.IsEmpty())
		pDoc->m_Part.Draw(pDC);

}

BOOL CSTLViewerView::GetBox(double& x0,double& y0,double& z0,
double& x1,double& y1,double& z1)
{
	CSTLViewerDoc* pDoc = GetDocument();
	ASSERT(pDoc);

	if(!pDoc->m_Part.IsEmpty()){
		CBox3D box;
		if(pDoc->m_Part.GetBox(box)){
			x0 = box.x0; y0 = box.y0;	z0 = box.z0;
			x1 = box.x1; y1 = box.y1;	z1 = box.z1;
			return TRUE;
		}
	}
	return FALSE;
}

void CSTLViewerView::OnViewBack()
{
	OnViewType(VIEW_BACK);
}

void CSTLViewerView::OnViewBottom()
{
	OnViewType(VIEW_BOTTOM);
}
void CSTLViewerView::OnViewFront()
{
	OnViewType(VIEW_FRONT);
}
```

```
void CSTLViewerView::OnViewLeft()
{
	OnViewType(VIEW_LEFT);
}

void CSTLViewerView::OnViewRight()
{
	OnViewType(VIEW_RIGHT);
}

void CSTLViewerView::OnViewTop()
{
	OnViewType(VIEW_TOP);
}

void CSTLViewerView::OnViewSWIsometric()
{
	OnViewType(VIEW_SW_ISOMETRIC);
}

void CSTLViewerView::OnViewSEIsometric()
{
	OnViewType(VIEW_SE_ISOMETRIC);
}

void CSTLViewerView::OnViewNEIsometric()
{
	OnViewType(VIEW_NE_ISOMETRIC);
}

void CSTLViewerView::OnViewNWIsometric()
{
	OnViewType(VIEW_NW_ISOMETRIC);
}

void CSTLViewerView::OnViewZoomall()
{
	ZoomAll();
}

void CSTLViewerView::OnViewZoomin()
{
	Zoom(0.9);
}

void CSTLViewerView::OnViewZoomout()
{
```

```
        Zoom(1.1);
}

void CSTLViewerView::OnViewShade()
{
        m_pGLDC->Shading(!m_pGLDC->IsShading());
        Invalidate();
}

void CSTLViewerView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
        switch(nChar){
        case VK_UP:
                MoveView(0.0,0.02);
                break;
        case VK_DOWN:
                MoveView(0.0,-0.02);
                break;
        case VK_RIGHT:
                MoveView(0.02,0);
                break;
        case VK_LEFT:
                MoveView(-0.02,0);
                break;
        }

        CGLView::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

本章相关程序

- ch7\STLViewer：**应用程序** STLViewer **的整个工程。**
- ch7\inc：STLViewer **需要调用的头文件。**
- ch7\lib：STLViewer **需要的连接库文件。**
- ch7\bin：**执行程序** STLViewer.exe **和它需要的动态库。**
- ch7\Models：**供** STLViewer.exe **调用的** STL **模型和** mdl **文件。**

第 8 章　增强 CAD 应用程序的界面功能

本章要点：

- 界面工具库 DockTool.dll 的创建。
- 工具栏的排列。
- 使用快捷菜单。
- 设计类似 Visual Studio 风格的浮动窗口。
- CTabCtrl 控件的增强。
- 文档与多个视图的关联。
- 使用树型视图 CTreeView 显示与操作 CSTLViewerDoc 中的文档数据。

在开发应用程序时，许多开发人员都很关注应用程序的界面设计。我们设计的程序是否能够吸引用户，是否能够充分展示程序的精彩内涵，很大程度上取决于是否有一个好的用户界面。软件的功能再丰富，也需要通过好的界面得以反映出来。界面开发是软件开发中非常重要的一部分。在 CAD 软件中，除图形显示之外，对几何模型的交互操作、模型的结构以及相关信息的显示等都需要通过良好的软件界面予以实现。

使用 MFC 的应用程序向导，可帮助用户生成一些最基本的软件界面。对于大多数应用程序来说，这只是一个系统界面的基本框架。通常，开发人员还需要在这个框架基础之上对它进行修改和增强，以增加更多的界面功能。在 MFC 中提供了较多的可用于界面开发的基础类，开发人员可直接调用这些 MFC 类或在它的基础上派生新的自定义类，用于界面制作。程序的界面风格随不同类型的软件而差别很大。大多数 CAD 应用程序采用文档/视图的框架结构，用户界面的主要功能是显示几何模型，并对图形及数据（包括模型的结构、几何信息等）进行交互操作。

本章提供的开发实例中，对应用程序 STLViewer 作了界面增强，开发了类似 Visual Studio 风格的程序界面，如图 8-1 所示。Visual C++界面开发技术不是本书介绍的重点，在这一章里也无法系统讲述有关的界面开发技术。本章通过对如图 8-1 所示的软件界面开发过程的讲述，将开发中涉及到的技术作一些介绍。目的在于，使读者通过对这个开发实例的分析，对 Windows 界面开发技术有一定的了解。

8.1　STLViewer 的界面增强

如图 8-1 所示，本章提供的开发实例对 STLViewer 界面做了增强。在主界面中，主框架窗口中的客户区包括了以下对象：位于顶部的两个浮动工具栏、位于左边的浮动窗口、位于下部的信息输出窗口以及中间的 OpenGL 绘图窗口。

浮动窗口采用了类似于 Visual Studio 开发环境中浮动窗口的风格，它不仅可以在主框架内任意停靠和浮动，而且窗口的尺寸可以由鼠标拖动而变化。左边的浮动窗口中，嵌入了一

个功能增强了的 Tab 控件。在该 Tab 控件中又嵌入了一个树型视图（一个页面），以显示和操作文档。利用这个控件可进一步嵌入多个页面（窗口），以强化界面功能。在 OpenGL 绘图窗口中，还激活了一个浮动的快捷菜单。在这一节，将分析实现该界面设计而涉及到的一些具体问题，包括：

- 工具栏的排列。
- 使用快捷菜单。
- 创建类似 Visual Studio 风格的浮动窗口。
- 使用树型控件显示与操作文档。

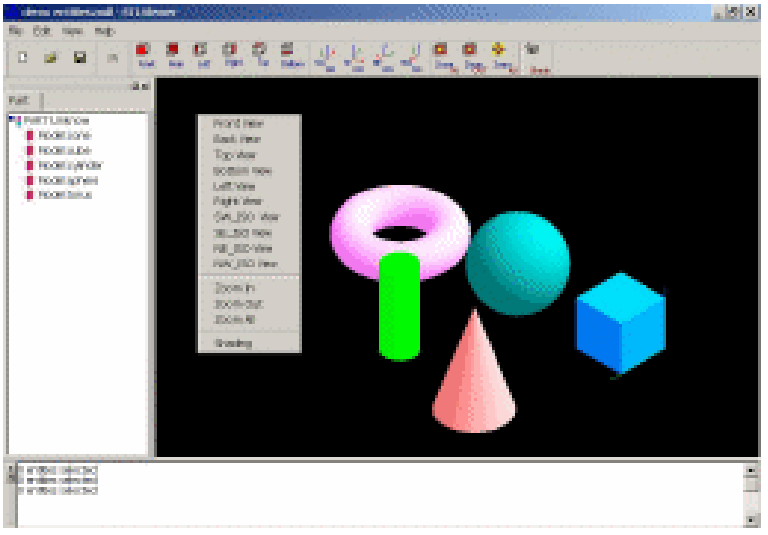


图 8-1　程序 STLViewer 增强后的用户界面

8.2　工具栏的排列

工具栏包含若干个位图按钮，单击位图按钮可以发出相应的命令消息。这个过程如同选中应用程序中的菜单项，以执行相关的命令。与菜单相比，工具栏提供了一种更快捷、直接的命令方式。工具栏的主要特点在于能够停靠和浮动，既可以停靠在主窗口边框的顶部、底部以及左右两边，也可以浮动于窗口中的任意位置。

在应用程序中，有时会使用多个工具栏。在默认的情况下，这些工具栏会按照创建的顺序，由上至下地排列。更多情况下，我们希望能够将一些长度不长的工具栏排列在一行，即左右或上下排列。这样能够节省工具栏占用的空间，也使得界面更整齐和美观，如图 8-2 所示。在 STLViewer 的工具栏排列中，由于 m_wndToolBar 只有四个按钮，将它与工具栏 m_wndDisplayBar 排在一行可以节省屏幕空间。

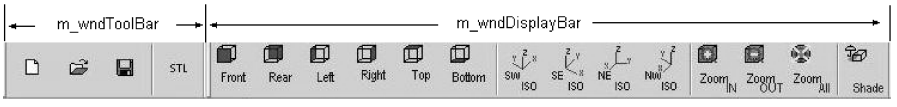


图 8-2　两个工具栏在一行中左右排列

为实现工具栏的左右或上下排列，需要在 CMainFrame 中设计停靠函数 DockControlBar-LeftOf()。该函数利用工具栏控制函数 DockControlBar()的 lpRect 参数，通过控制工具栏的停靠矩形区域的方法来实现这种停靠方式。

函数 DockControlBarLeftOf()实现如下，其中指针 pBar 指向右侧的工具栏，LeftOf 指向左侧的工具栏。

```
void CMainFrame::DockControlBarLeftOf(CToolBar* pBar,CToolBar* LeftOf)
{
    CRect rect;        //矩形区域定义
    DWORD dw;
    UINT n;

    // 使用 MFC 来重新计算所有工具栏的尺寸，确保 GetWindowRect()准确
    RecalcLayout();
    LeftOf->GetWindowRect(&rect);

    //设置偏移值以停靠在同一条边上
    rect.OffsetRect(1,0);
    dw=LeftOf->GetBarStyle();
    n = 0;
    n = (dw&CBRS_ALIGN_TOP) ? AFX_IDW_DOCKBAR_TOP : n;
    n = (dw&CBRS_ALIGN_BOTTOM && n==0) ? AFX_IDW_DOCKBAR_BOTTOM : n;
    n = (dw&CBRS_ALIGN_LEFT && n==0) ? AFX_IDW_DOCKBAR_LEFT : n;
    n = (dw&CBRS_ALIGN_RIGHT && n==0) ? AFX_IDW_DOCKBAR_RIGHT : n;

    //将工具栏停靠在 rect 规定的矩形区域内
    DockControlBar(pBar,n,&rect);
}
```

工具栏通常在 CMainFrame::OnCreate()函数中创建，并设置停靠风格和位置。在 OnCreate()中要增加创建和显示工具栏 m_wndDisplayBar 的代码，并设置它的停靠位置。需要使用函数 DockControlBarLeftOf()来替代函数 DockControlBar()，使两个工具栏停靠在一行。相关代码如下所示：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    ……
    //设置 m_wndToolBar 可以停靠在窗口内的任意位置上
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);

    //设置 m_wndDisplayBar 可以停靠在窗口内的任意位置上
    m_wndDisplayBar.EnableDocking(CBRS_ALIGN_ANY);

    //给工具栏窗口命名
    m_wndToolBar.SetWindowText("Standard Tool");
    m_wndDisplayBar.SetWindowText("Display Tool");
```

```
    //允许工具栏在窗口内的任意位置停靠
    EnableDocking(CBRS_ALIGN_ANY);

    //停靠工具栏 m_wndToolBar 于默认位置
    DockControlBar(&m_wndToolBar);
    //将工具栏 m_wndDisplayBar 与 m_wndToolBar 停靠于一行
    DockControlBarLeftOf(&m_wndDisplayBar,&m_wndToolBar);
    ……
    return 0;
}
```

8.3 使用快捷菜单

用户单击鼠标右键时弹出的一个浮动菜单称为快捷菜单(Context Menu)**。快捷菜单为用户提供常用的命令列表以方便操作，这些命令通常是和当前窗口相关的一些常用操作。如图** 8-1 **所示，在** STLViewer **的视图窗口** CSTLViewerView **中单击鼠标右键，会弹出一个快捷菜单，其中包括了诸如视图缩放、视角变换等常用的显示操作。**

Visual C++**专门为使用快捷菜单提供了** WM_CONTEXTMENU **消息。用户只需要在应用程序的窗口中添加该消息的处理函数即可。在系统的资源文件中，我们为快捷菜单创建了一个菜单资源** IDR_CONTEXT_MENU**，在消息处理函数中将装载并弹出这个菜单。下面是** WM_CONTEXTMENU **对应的消息处理函数。**

```
void CSTLViewerView::OnContextMenu(CWnd* pWnd, CPoint point)
{
    CMenu menu;
    //载入快捷菜单资源
    VERIFY(menu.LoadMenu(IDR_CONTEXT_MENU));

    CMenu* pPopup = menu.GetSubMenu(0);
    ASSERT(pPopup);

    //弹出快捷菜单
    pPopup->TrackPopupMenu(TPM_LEFTALIGN|TPM_RIGHTBUTTON,point.x,
            point.y, this);
}
```

完成以上工作后，当鼠标位于 CSTLViewerView **的窗口中，且按下鼠标右键时，在鼠标所在位置将弹出快捷菜单。**

8.4 创建类似 Visual Studio 风格的浮动窗口

很多 Visual C++**的用户很欣赏** Visual Studio **所提供的界面风格，尤其是主框架下的两个浮动的窗口** WorkSpace **和** Output**。类似于可浮动的工具栏，这种可浮动的窗口可以停靠在主窗口的任意一边，也可以浮动在主窗口内的任何位置，窗口的大小可以随鼠标的拖动而改变，**

还可以关闭和重新打开。在 CAD 应用程序中，这种界面风格较为常用。在 STLViewer 的界面增强中，我们创建了两个这种风格的窗口，即左侧的 Work Bar 和底部的 Output Bar。下面将介绍创建这种类型窗口的一些相关技术和过程。

8.4.1 控制条与停靠栏

1．控制条类 CControlBar

在 MFC 中，具有浮动性能的窗口都由一个共同的基类——控制条类 CControlBar 派生。如工具栏类 CToolBar、状态栏类 CStatusBar 和对话条类 CDialogBar 都是 CControlBar 的派生类，在这些派生类中，根据需要对 CControlBar 的默认属性作了修改，并增加了自己的特性。如在 CStatusBar 中限定了状态栏只能水平排列，而不能垂直排列。可以将这些在 CControlBar 基础上派生的窗口统称为控制条窗口。控制条窗口通常是一个与视图窗口处于同一级的框架类的子窗口。它通过获得父框架窗口的客户区（Client Area）的位置信息来计算自己的尺寸和位置，并通知父框架窗口剩下的客户区的位置和尺寸。

CControlBar 控制条之间免于互相覆盖，且不会遮盖视图窗口。但控制条能够沿框架窗口停靠取决于框架窗口沿边框四周创建的 4 个统一的控制条，即停靠栏。控制条与停靠栏如图 8-3 所示。

2．停靠栏（Dock Bar）

用户往往忽略了停靠栏的存在。因为没有控制条来停靠时，这些停靠栏沿应用程序主窗口缩小到不可见。当控制条停靠在边框时，停靠栏就展开并包围这些所停靠的控制条。

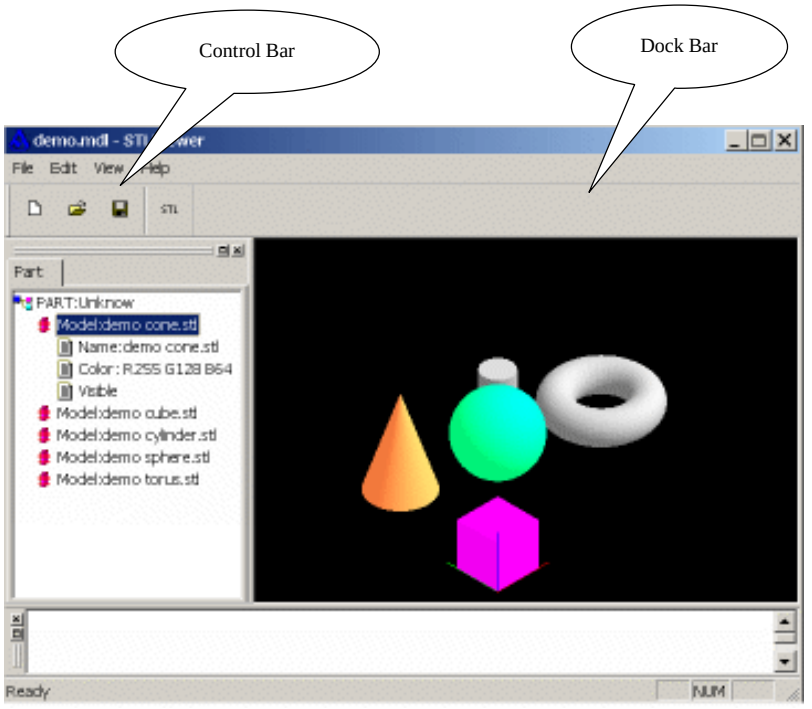


图 8-3　控制条与停靠栏

CFrameWnd 的成员函数 EnableDocking()沿框架窗口创建这 4 个停靠栏。我们注意到，停

靠功能不是自动地被 MFC 所启动的。在函数 CMainFrame::OnCreate()中，需要调用 EnableDocking()才能进行工具栏的停靠操作。几个与停靠操作相关的函数和功能分别是：

- CFrameWnd::EnableDocking()：沿框架四周创建停靠栏。
- CControlBar::EnableDocking()：设置控制条的停靠属性。
- CFrameWnd::DockControlBar()：将一个控制条停靠到一个停靠栏中。

在 CMainFrame::OnCreate()中，有关控制条停靠的代码解释如下：

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
……
    //允许工具栏停靠
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    m_wndDisplayBar.EnableDocking(CBRS_ALIGN_ANY);

    m_wndToolBar.SetWindowText("Standard Tool");
    m_wndDisplayBar.SetWindowText("Display Tool");

    //在主框架窗口四周创建停靠栏
    EnableDocking(CBRS_ALIGN_ANY);

    //将控制条停靠到停靠栏，默认状态下将停靠到上边的停靠栏
    DockControlBar(&m_wndToolBar);
    //将 m_wndDisplayBar 停靠到 m_wndToolBar 的右边
    DockControlBarLeftOf(&m_wndDisplayBar,&m_wndToolBar);

    //------------------------------------------------------------
    //创建左边的浮动窗口
    if (!m_LeftDockBar.Create(this,"Work Bar"))
    {
        TRACE0("Failed to create LeftDockBar\n");
        return -1;          // fail to create
    }

    //创建位于底部的信息输出浮动窗口
    m_OutputDockBar.SetDockSize(CSize(200,100));
    if(!m_OutputDockBar.Create(this,"Output Bar"))
    {
        return -1;
    }

    //只允许控制条 m_LeftDockBar 使用左右两个停靠栏
    m_LeftDockBar.EnableDocking(CBRS_ALIGN_LEFT | CBRS_ALIGN_RIGHT);
    //使用左边的停靠栏来停靠 m_LeftDockBar
    DockControlBar(&m_LeftDockBar,AFX_IDW_DOCKBAR_LEFT);

    //只允许将控制条 m_OutputDockBar 停靠在上下两个停靠栏
```

```
    m_OutputDockBar.EnableDocking(CBRS_ALIGN_TOP | CBRS_ALIGN_BOTTOM);
    //使用下面的停靠栏来停靠 m_OutputDockBar
    DockControlBar(&m_OutputDockBar,AFX_IDW_DOCKBAR_BOTTOM);
    //----------------------------------------------------------

    return 0;
}
```

在 CControlBar 基础上派生的窗口类，能够继承 CControlBar 提供的窗口停靠和浮动的属性。

8.4.2 开发具有 Visual Studio 风格的浮动窗口

图 8-1 所示界面中两个浮动窗口的实现类分别是自行开发的。

- CtabSheetDockBar：左边嵌套 Tab 控件的浮动窗口类。
- CmessageViewDockBar：底部嵌套信息提示窗口的浮动窗口类。

这两个类都由控制条类 CControlBar 派生，不过我们不需要直接从 CControlBar 派生，而使用 Mr. Oliver Smith 编写的类 CCoolDialogBar[①㊀]。类 CCoolDialogBar 由 CControlBar 派生，是一个优秀的浮动窗口类，可以在框架窗口内任意停靠，并可由鼠标拖动改变窗口的大小。CCoolDialogBar 的实现代码较长，在本书附带光盘 DockTool 工程的源代码中提供了 CCoolDialogBar 的源代码（CoolDialogBar.h、CoolDialogBar.cpp），有兴趣的读者可以参考这些源代码了解它的设计与实现。

类 CCoolDialogBar 提供了很好的窗口浮动特性，但窗口中不实现任何具体的内容。在 CCoolDialogBar 基础上派生的类可以保留这些窗口特性，并在窗口中增加自己的特殊内容。CTabSheetDockBar 和 CMessageViewDockBar 就是这样两个自定义的派生类。在 CTabSheetDockBar 中，嵌入了一个 CTabSheet（自行开发的 CTabCtrl 的派生类）的对象作为窗口中的内容。在 CTabSheet 中可以嵌入一个或多个页面，这就是我们看到的如图 8-1 所示界面中左边的浮动窗口。类 CMessageViewDockBar 的风格类似于 Visual C++中的 Output 窗口，由于嵌入了一个视图对象，因而可以在这个嵌入的视图中输出提示信息。

下面讲述类 CTabSheetDockBar 的实现与使用方法。

类 CTabSheetDockBar 的定义如下：

```
class AFX_EXT_CLASS CTabSheetDockBar : public CCoolDialogBar
{
    DECLARE_DYNCREATE(CTabSheetDockBar)
public:
    CTabSheetDockBar();
    virtual ~CTabSheetDockBar();
    virtual BOOL Create( CWnd* pParentWnd,LPCTSTR pTitle);

    // Dialog Data
    //{{AFX_DATA(CTabSheetDockBar)
```

㊀ CCoolDialogBar 的源代码作者是 Mr. Oliver Smith，作者同意在本书及附带的光盘中使用与公开 CCoolDialogBar 的源代码。

```
        //}}AFX_DATA
        CTabSheet m_TabCtrl; //增强了的 Tab 控件

        // Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CTabSheetDockBar)
        //}}AFX_VIRTUAL

        //设置窗口浮动时的尺寸
        void  SetFloatSize(const CSize& size);
        //设置窗口停靠时的尺寸
        void  SetDockSize(const CSize& size);
protected:

        // Generated message map functions
        //{{AFX_MSG(CTabSheetDockBar)
        afx_msg int    OnCreate(LPCREATESTRUCT lpCreateStruct);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

在类 CTabSheetDockBar **中，我们嵌入了一个功能增强了的** Tab **控件** m_TabCtrl**，这是该浮动窗口中的内容。在** CTabSheetDockBar::OnCreate()**函数中创建这个控件。**

```
int CTabSheetDockBar::OnCreate( LPCREATESTRUCT lpCreateStruct )
{
        if (CCoolDialogBar::OnCreate(lpCreateStruct) == -1)
                return -1;

        RECT rect;
        GetClientRect(&rect);
        //设置 Tab 控件的页边距
        rect.left += 15;
        rect.top += 15;
        rect.bottom -= 15;
        rect.right -= 15;

        //创建 Tab 控件
        m_TabCtrl.Create(WS_CHILD|WS_VISIBLE,rect,this,10071);
        m_pCtrlWnd = &m_TabCtrl;
        return 0;
}
```

类 CTabSheetDockBar **的使用方法如下：**

（1**）在** CMainFrame **中声明一个** CTabSheetDockBar **的对象。**

```
#include "..\inc\DockTool\DockTool.h"
class CMainFrame : public CFrameWnd
```

```
{
……
public:
    CTabSheetDockBar        m_LeftDockBar;
……
}
```

（2）在 CMainFrame::OnCreate()**中创建对象，并向** m_LeftDockBar **中的** m_TabCtrl **控件增加页面（窗口）。**

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
……
    //设置 m_LeftDockBar 窗口停靠时的尺寸
    m_LeftDockBar.SetDockSize(CSize(200,200));

    //设置 m_LeftDockBar 窗口浮动时的尺寸
    m_LeftDockBar.SetFloatSize(CSize(200,200));

    //创建 m_LeftDockBar 对象
    if (!m_LeftDockBar.Create(this,"Work Bar"))
    {
        TRACE0("Failed to create LeftDockBar\n");
        return -1;          // fail to create
    }

    //向 m_LeftDockBar 中的 Tab 控件添加页面
    m_LeftDockBar.m_TabCtrl.AddView("Part",RUNTIME_CLASS(CPartTreeView));

    //将所设置的页面设为 Tab 控件的当前页
    m_LeftDockBar.m_TabCtrl.SetActiveView(0);
……
}
```

类 CMessageViewDockBar **的实现与使用和** CTabSheetDockBar **类似，这里不再讲述。在本书附带的光盘中给出了全部源程序（参见文件** DockTool.h、LeftDockBar.cpp**），供读者参考。**

8.4.3 CTabCtrl 控件的功能增强

在类 CTabSheetDockBar **中嵌套了一个功能增强了的** Tab **控件，用于多页面的管理。类** CTabSheet **是在** MFC **类** CTabCtrl **基础上派生的。如图** 8-1 **所示，在** STLViewer **的界面中，使用了这个功能增强了的** Tab **控件来实现多个页面之间的切换。**Tab **控件可以在它的窗口中管理多个页面，可以把几个不同的页面（窗口、视图或对话框等）都安排在** Tab **控件中，使用时可根据需要在控件中动态添加、删除页面，并选择页面的标签来实现不同页面之间的切换，而且在页面选择发生变化时得到通知。**

MFC 通过使用 CTabCtrl 来封装 Tab 控件中的各种操作，CTabCtrl 的对象创建后必须向其中添加页面才可以使用。但使用 CTabCtrl 控件时，只能将添加的页面拖到第一个 Tab 页中显示，因此要实现页面之间的切换需要通过用户自己编写代码来实现，以在不同的 Tab 页中显示不同的内容。

在设计 STLViewer 界面时，根据以上的需要对 CTabCtrl 作了改进。即在 CTabCtrl 的基础上派生了一个较通用的类 CtabSheet，在类 CTabSheet 中增加对页面的操作函数(添加、删除、选择、设置边距等)。类 CTabSheet 的设计如图 8-4 所示。在 Tab 控件管理与操作多个页面的情况下，使用 CTabSheet 更加方便。

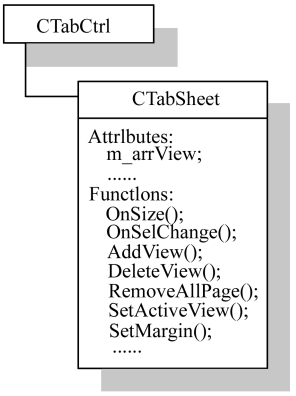


图 8-4　类 CTabSheet 的设计

以下是类 CTabSheet 的实现代码和解释：

```
//CTabSheet 的声明
class AFX_EXT_CLASS CTabSheet : public CTabCtrl
{
// Construction
public:
        CTabSheet();
// Attributes
public:
        CArray<CWnd*,CWnd*> m_arrView;      //指向控件中页面的指针数组
// Operations
public:
// Overrides
        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CTabSheet)
        public:
        virtual BOOL Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID );
        //}}AFX_VIRTUAL

// Implementation
public:
        void RemoveAllPage();               //删除所有页面
        void SetMargin(int margin);         //设置页边距
        void SetActiveView(int nView);      //设置当前页

        //使用动态创建方式创建一个页面，并增加到控件中
        // lpszLabel 是该窗口的标签名称
        BOOL AddView(LPTSTR lpszLabel, CRuntimeClass *pViewClass,
                CCreateContext *pContext=NULL);

        //将一个已创建的页面增加到对话框中，lpszLabel 是该窗口的标签名称
        BOOL AddView(LPTSTR lpszLable, CWnd* pView);
```

```
    virtual ~CTabSheet();

    // Generated message map functions
protected:
    int         m_iMargin;          //页边距
    CWnd*       m_curView;          //当前页的指针
    CRect       m_rcView;           //页窗口的大小
    CFont       m_TabFont;          //标签的字体

    virtual void InitFont();        //初始化标签的字体
    //{{AFX_MSG(CTabSheet)

    //响应页面的选择变化
    afx_msg void OnSelchange(NMHDR* pNMHDR, LRESULT* pResult);

    //响应页面的尺寸变化，使得页面的尺寸随控件尺寸的变化而变化
    afx_msg void OnSize(UINT nType,int cx,int cy);
    //}}AFX_MSG

    DECLARE_MESSAGE_MAP()
};

CTabSheet::CTabSheet()
{
    m_curView = NULL;
    m_iMargin = 5;
}

CTabSheet::~CTabSheet()
{
    m_arrView.RemoveAll();
}

BEGIN_MESSAGE_MAP(CTabSheet, CTabCtrl)
//{{AFX_MSG_MAP(CTabSheet)
    ON_NOTIFY_REFLECT(TCN_SELCHANGE, OnSelchange)
    ON_WM_SIZE()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CTabSheet message handlers
void CTabSheet::OnSelchange(NMHDR* pNMHDR, LRESULT* pResult)
{
    int nView = GetCurSel();
    //将所选页面放置到最前面，即 Tab 控件中能看到的当前页面
    SetActiveView(nView);
```

```
    *pResult = 0;
}

void CTabSheet::RemoveAllPage()
{
    m_arrView.RemoveAll();
}

//////////////////////////////////////////////////////////////////
//  virtual functions
//////////////////////////////////////////////////////////////////
void CTabSheet::OnSize(UINT nType,int cx,int cy)
{
    CTabCtrl::OnSize(nType, cx, cy);
    //根据 Tab 控件变化后的尺寸重新计算所嵌入页面的尺寸
    m_rcView.left   = m_iMargin;
    m_rcView.right = cx-m_iMargin;
    m_rcView.top    = m_iMargin+23;
    m_rcView.bottom= cy-m_iMargin;
    if(m_curView){
        //在改变了尺寸的窗口中重新显示页面
        m_curView->SetWindowPos(NULL,m_rcView.left,m_rcView.top, m_rcView.Width(),
        m_rcView.Height(),SWP_NOMOVE|SWP_NOZORDER|SWP_SHOWWINDOW);
    }
}

BOOL CTabSheet::Create(DWORD dwStyle, const RECT& rect, CWnd* pParentWnd, UINT nID )
{
    //创建 Tab 控件，并设置 Tab 控件标签上的字体
    if(CTabCtrl::Create(dwStyle,rect,pParentWnd,nID)){
        InitFont();
    return TRUE;
}
else
        return FALSE;
}

//用动态创建方式创建一个页面，并增加到控件中
BOOL CTabSheet::AddView(LPTSTR lpszLabel, CRuntimeClass *pViewClass,
         CCreateContext *pContext)
{
    CCreateContext context;
    if (pContext == NULL)
    {
    context.m_pCurrentDoc = NULL;
        context.m_pCurrentFrame = GetParentFrame();
        context.m_pLastView = NULL;
```

```
        context.m_pNewDocTemplate = NULL;
        context.m_pNewViewClass = pViewClass;
        pContext = &context;
    }

    CWnd* pWnd;
    TRY
    {
        pWnd = (CWnd*)pViewClass->CreateObject();
        if (pWnd == NULL)
            AfxThrowMemoryException();
    }
    CATCH_ALL(e)
    {
        TRACE0(_T("Out of memory creating a view.\n"));
        // Note: DELETE_EXCEPTION(e) not required
        return FALSE;
    }
    END_CATCH_ALL

    ASSERT_KINDOF(CWnd, pWnd);
    ASSERT(pWnd->m_hWnd == NULL);

    DWORD dwStyle = AFX_WS_DEFAULT_VIEW;

    // Create with the right size and position
    if (!pWnd->Create(NULL, NULL, dwStyle, m_rcView, this, 0, pContext))
    {
        TRACE0(_T("Warning: couldn't create client pane for view.\n"));
        // pWnd will be cleaned up by PostNcDestroy
        return FALSE;
    }
    CView* pView = (CView*) pWnd;

    TC_ITEM ti;
    ti.mask=TCIF_TEXT;

    ti.pszText= lpszLabel;
    InsertItem(m_arrView.GetSize(),&ti);
    m_arrView.Add(pView);

    SetActiveView(m_arrView.GetSize()-1);
    return TRUE;
}
```

//将一个已创建的页面增加到对话框中

```
BOOL CTabSheet::AddView(LPTSTR lpszLabel, CWnd* pView)
```

```
{
    TC_ITEM ti;
    ti.mask=TCIF_TEXT;

    ti.pszText= lpszLabel;
    InsertItem(m_arrView.GetSize(),&ti);
    m_arrView.Add(pView);

    //将新载入的页面设为当前活动页面
    SetActiveView(m_arrView.GetSize()-1);

    CRect rect;
    this->GetClientRect(&rect);
    int cx = rect.Width();
    int cy = rect.Height();

    m_rcView.left    = m_iMargin;
    m_rcView.right = cx-m_iMargin;
    m_rcView.top     = m_iMargin+23;
    m_rcView.bottom= cy-m_iMargin;
    if(m_curView){   //将页面移动到合适的位置上显示
        m_curView->SetWindowPos(NULL,m_rcView.left,m_rcView.top, m_rcView.Width(),
        m_rcView.Height(),SWP_NOZORDER|SWP_SHOWWINDOW);
    }
    return TRUE;
}

//设置页边距
void CTabSheet::SetMargin(int margin)
{
    m_iMargin = margin;
}

//在 Tab 控件管理的窗口中设置一个当前的页面，即 Tab 控件中所显示的页面
void CTabSheet::SetActiveView(int nView)
{
    SetCurSel(nView);
    ASSERT(nView>=0 && nView<m_arrView.GetSize());
    CWnd* pView = m_arrView[nView];
    if(m_curView && m_curView != pView){
        m_curView->EnableWindow(FALSE);
        m_curView->ShowWindow(SW_HIDE);//隐藏当前页面
    }
    pView->EnableWindow(TRUE);          //激活新的页面
    pView->ShowWindow(SW_SHOW);     //显示新的页面
    pView->SetFocus();
    pView->SetWindowPos(NULL,m_rcView.left,m_rcView.top, m_rcView.Width(),
```

```
m_rcView.Height(),SWP_NOMOVE|SWP_NOZORDER|SWP_SHOWWINDOW);
m_curView = pView;
}

//初始化字体
void CTabSheet::InitFont()
{
    // Get the log font.
    NONCLIENTMETRICS ncm;
    ncm.cbSize = sizeof(NONCLIENTMETRICS);
    VERIFY(::SystemParametersInfo(SPI_GETNONCLIENTMETRICS,
        sizeof(NONCLIENTMETRICS), &ncm, 0));

    m_TabFont.CreateFontIndirect(&ncm.lfMessageFont);
    this->SetFont(&m_TabFont);
}
```

8.4.4 建立界面工具库 DockTool.dll

考虑到上述类型的控件具有较强的通用性，在一个 DLL **库中实现这些类将有利于不同的应用程序共享这些类。我们建立一个界面工具库** DockTool.dll **来输出这些类，**DockTool.dll **中的输出类及类之间的关系如图** 8-5 **所示。**

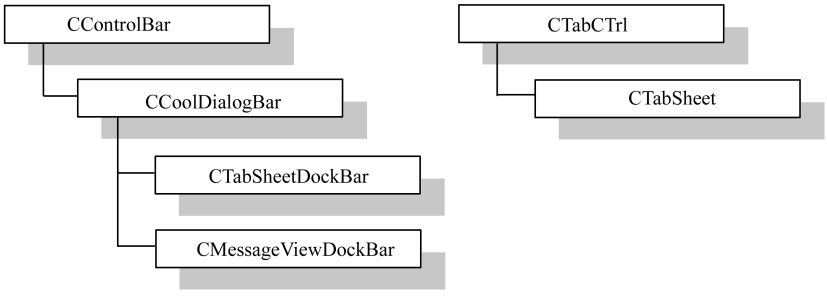


图 8-5　DockTool.dll **的输出类**

在本书附带光盘 ch8\DockTool**子目录下提供了整个库的源代码，供读者参考。**

在 STLViewer **中，对以上类的调用是通过调用动态库** DockTool.dll **来实现的。使用库** DockTool.dll **需要以下三个文件：**

- DockTool.dll**：动态库**
- DockTool.lib**：连接库**
- DockTool.h**：输出类的说明文件**

8.5 使用树型视图 CTreeView 显示和管理文档数据

STLViewer **的浮动窗口中添加了一个树型视图** CTreeView **来显示几何模型的结构，并通过其中的树型控件** CTreeCtrl **的对象对模型中的数据进行相关操作。树型控件的功能较丰富，**

CTreeCtrl 的成员函数也很多。在这里无法对树型控件的所有操作和具体函数作一一介绍，而是根据应用程序 STLViewer 开发中对树型控件和树型视图的使用，来介绍 CTreeCtrl 和 CTreeView 的主要功能和使用方法，以及如何实现它们与文档的关联。读者对 CTreeCtrl、CTreeView 的使用有了一定了解后，可查阅 MSDN 进一步了解它的其余功能。

8.5.1 树型视图与树型控件

树型控件类 CTreeCtrl 以及封装了树型控件的树型视图类 CTreeView 主要用于显示具有层次结构的条目，是 Windows 软件中最常用的控件之一。最典型的应用是文件系统目录结构的显示。使用 Visual C++的开发人员会注意到在 Visual Studio 的浮动窗口 WorkSpace 中，通过 Tab 控件嵌套了多个树型视图，分别用于类和函数、资源以及文件的结构化显示。在 CAD 软件中，这种树型控件和树型视图也常用于几何模型结构的显示以及对文档数据的操作。在界面增强了的 STLViewer 中，也使用了树型视图来显示几何模型的结构与属性，如图 8-6 所示，并通过其中的树型控件来实现与几何模型及其属性的关联和交互操作。

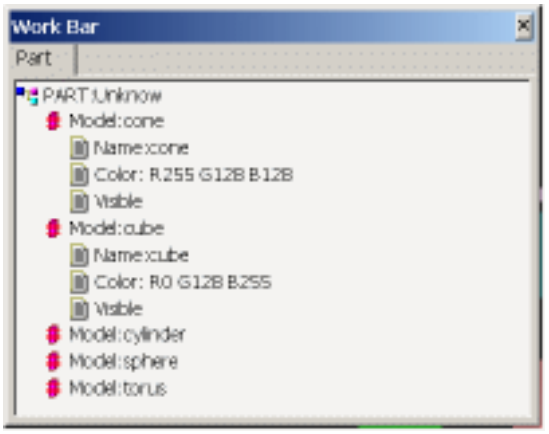


图 8-6 使用 CTreeView 显示 STLViewer 中模型的结构和属性

8.5.2 在 STLViewer 中创建 CPartTreeView

STLViewer 中，在类 CTreeView 的基础上派生了新类 CPartTreeView，用于表示 CSTLViewerDoc 中的几何模型对象 m_Part 的结构与属性。

CPartTreeView 的定义如下：

```
#include <afxcview.h>

class CSTLViewerDoc;
class CEntity;
class CPartTreeView : public CTreeView        //在 CTreeView 上派生
{
protected:
    CPartTreeView();                  // protected constructor used by dynamic creation
    DECLARE_DYNCREATE(CPartTreeView)
```

```
// Attributes
public:

// Operations
public:
    CSTLViewerDoc* GetDocument();        //实现与 CSTLViewerDoc 文档的关联

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CPartTreeView)
    public:
    virtual void OnInitialUpdate();
    protected:
    virtual void OnDraw(CDC* pDC);          // overridden to draw this view

    //对应于文档类的函数 UpdateAllViews();
    virtual void OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint);
    //}}AFX_VIRTUAL

// Implementation
protected:
    virtual ~CPartTreeView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
    //操作文档模型属性
    void  ChangeEntityName(CEntity* ent);        //改变模型名称
    void  ChangeEntityColor(CEntity* ent);       //改变模型颜色
    void  ChangeEntityVisible(CEntity* ent);     //改变模型显示属性

    // Generated message map functions
protected:
    //{{AFX_MSG(CPartTreeView)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
    afx_msg void OnLButtonDblClk(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()

protected:
    CImageListm_ImageList;                  //用于属性控件中条目的图标
};
```

在函数 CMainFrame::OnCreate()**中，动态创建了一个** CPartTreeView **的对象，并将它嵌入了浮动窗口的** Tab **控件中，代码如下：**

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
```

```
{
    ……
    //创建 CPartTreeView 对象，并将该控件嵌入 Tab 控件
    m_LeftDockBar.m_TabCtrl.AddView("Part",RUNTIME_CLASS(CPartTreeView));

    //将该页面设为当前活动页面
    m_LeftDockBar.m_TabCtrl.SetActiveView(0);
    ……
}
```

下面，将分别讲述设计类 CPartTreeView **的几个问题。**

- **使** CPartTreeView **与文档** CSTLViewerDoc **关联。**
- **在** CPartTreeView **中使用图标。**
- **使用** CPartTreeView **显示文档中模型结构及其属性。**
- **使用** CPartTreeView **对文档数据进行操作。**

8.5.3 树型视图 CPartTreeView 与文档的关联/文档多视图

一个文档可以对应多个视图，但一个视图却只能关联一个文档。在视图中，调用 CView::GetDocument **可以获得一个指向视图的文档的指针。函数原型如下：**

```
CDocument *GetDocument ( ) const;
```

如果该视图不与任何文档相关联，则函数返回 NULL**。**

类 CPartTreeView **是** CView **的派生类，也可关联一个文档。在** STLViewer **中，**CSTLViewerView **视图与** CSTLViewerDoc **文档之间互相关联。我们希望将** CPartTreeView **视图也关联到** CSTLViewerDoc **文档上。如图** 8-7 **所示，这样的** CSTLViewerDoc **文档可使用两个相关联的视图来分别显示文档中模型的图形和结构信息，这就是要实现文档的多视图，即一个文档关联多个视图。当需要从不同的角度来看单文档的内容时，就需要实现文档多视图。**

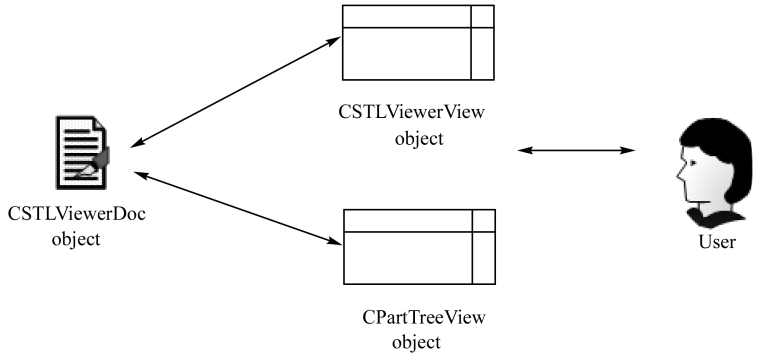


图 8-7　**文档与多个视图的关联**

在 STLViewer **中，视图类** CSTLViewerView **与文档类** CSTLViewerDoc **的关联是在模板类中自动实现的。在函数** CMainFrame::OnCreate()**中，我们创建了** CPartViewTree **视图，但它没**

有被自动关联到文档上，所以还需要将它与文档相关联。

文档类 CDocument 保存并维护一个指向所有相关联视图的指针列表 m_viewList，并设计了以下函数用于维护和操作相关联的视图。

（1）函数 AddView()。函数 CDocument::AddView()用于将一个视图连接到文档上，即将该视图的指针加入到列表 m_viewList 中，并将所添加的视图对象的文档指针 m_pDocument（指针 m_pDocument 在 CView 中定义）指向该文档。当使用 File→New、File→Open、Window→Split 等命令将一个新创建的视图对象连接到文档上时，MFC 会自动调用该函数，实现文档和视图的关联。但在创建视图时，若 MFC 没有自动调用该函数将视图关联到文档上，则需要程序员根据自己的需要调用该函数。

（2）函数 RemoveView()。函数 RemoveView()将一个与文档关联的视图从列表中去除，并将所去除的视图对象的文档指针 m_pDocument 设为 NULL。

（3）函数 UpdateAllViews()。函数 UpdateAllViews()是保持视图与文档同步的一个重要函数。当文档内容改变了后，需要调用此函数，通知与文档关联的视图更新视图内容。在 CView 中，与之对应的函数是 CView::OnUpdate()，在 OnUpdate()中需要根据文档内容的变化重新绘制视图。

（4）其他相关函数。以下两个函数用于访问文档中的视图列表：

```
virtual POSITION GetFirstViewPosition() const;
virtual CView* GetNextView(POSITION& rPosition) const;
```

另外，CDocument 还提供了一个虚拟函数 OnChangedViewList()。当在文档中增加或删除一个视图时，MFC 会调用 OnChangeViewList()函数。如果被删除的视图是该文档的最后一个视图，则删除该文档。

在 STLViewer 的函数 CMainFrame::OnCreate()中，只是创建了 CPartViewTree 视图，但还没有实现它与文档的关联。在函数 CPartViewTree::GetDocument()中，设计了有关代码以实现与文档的关联。

```
CSTLViewerDoc* CPartTreeView::GetDocument()
{
    //若该视图还未与文档关联
    if(m_pDocument==NULL){
        //获取应用程序的文档指针

        CFrameWnd* frm = (CFrameWnd*) ::AfxGetMainWnd();
        ASSERT(frm);
        CDocument* pDoc = frm->GetActiveDocument();
        ASSERT(pDoc);

        //确认获取的文档是所需要的类型
        ASSERT(pDoc->IsKindOf(RUNTIME_CLASS(CSTLViewerDoc)));

        //在文档中加入此视图的指针
        pDoc->AddView(this);
    }
```

```
        //若已经与文档关联，则返回文档指针
        return (CSTLViewerDoc*)m_pDocument;
    }
```

这样，当 CPartTreeView 第一次调用 GetDocument()时，将建立该视图与文档的关联。当文档内容发生变化后，如增加了一个子模型，在文档类中应该调用 UpdateAllViews()来更新这些关联的视图。

8.5.4 在树型控件中使用图标

树型控件的每个条目都可以与一对位图图标相关联。图标显示于条目标签的左边，其中一个图标用于标识条目未被选中时的状态，而另一个图标则用于标识条目被选中时的状态。例如，当文件夹条目被选中时，它的图标是一个打开的文件夹；而未被选中时，图标则是一个关闭的文件夹。如图 8-6 所示，在 STLViewer 的树型视图中，使用不同的三种图标来分别表示总模型、子模型、子模型的属性（名称、颜色、显示/隐藏）。

为 CPartTreeView 创建树型控件图标的步骤如下：

（1）创建位图资源。在 STLViewer 的系统资源文件中为 CPartTreeView 创建一个位图资源 IDB_PART_TREE，作为控件的图标资源。资源 IDB_PART_TREE 的像素为 48×16，即每个图标是一个 16×16 的像素点阵。用户可以设计其他的尺寸，但必须为正方形。

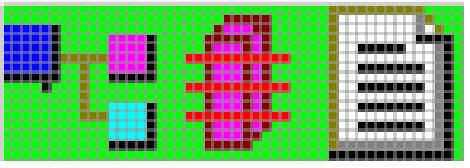


图 8-8　为 CPartTreeView 创建的控件图标资源

（2）在类中添加用于存放图像列表的成员变量。在类 CPartTreeView 中声明以下变量，MFC 类 CImageList 用于存放一组像素尺寸相同的图像。

```
protected:
    CImageListm_ImageList;
```

（3）为树型控件创建并指定图像列表。这个工作通常是在 OnCreate()函数中完成。以下是 CPartTreeView::OnCreate()函数中为树型控件创建并设置图像列表的代码。

```
int CPartTreeView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CTreeView::OnCreate(lpCreateStruct) == -1)
        return -1;

    //创建图像列表 m_ImageList
    m_ImageList.Create(IDB_PART_TREE, 16, 1, RGB(0,255,0));

    //将图象列表 m_ImageList 指定给树型控件
    GetTreeCtrl().SetImageList(&m_ImageList, TVSIL_NORMAL);
    GetTreeCtrl().CancelToolTips(TRUE);
```

```
    CancelToolTips(TRUE);
    return 0;
}
```

（4）为树型控件条目指定图标。为树型控件设置了图像后，就可以在插入条目时使用相应的图标了。当然，也可以在以后的任何时候为条目指定一个图标。使用函数 InsertItem()**可以在创建条目的同时为条目指定两个图标；函数** SetItemImage()**可对一个已经创建的条目指定两个图标。被指定的两个图标中，第一个用于被选中时显示，另一个用于条目未被选中时显示，这两个图标可以相同。**

CPartTreeView::OnUpdate()**中，在创建条目的同时为每个条目指定图标。例如，下列代码创建的一个条目，它的两个图标都使用图像列表中的第一个图标。**

```
HTREEITEM hti;
hti = treeCtrl.InsertItem("root",0,0,NULL,TVI_LAST);
```

8.5.5 使用树型视图控件显示文档中几何模型的结构和属性

在 STLViewer **中使用了树型控件来显示文档中几何模型的结构和属性，如图** 8-6 **所示。树型控件表示了整个模型的层次结构，每个子模型还包括了一些它的属性条目，如字符型名称、颜色、显示或隐藏。双击这些属性条目，可以对这些属性作修改。**

这个树型控件的条目应该随所载入的几何模型而及时更新。由于已经将这个树型视图和 CSTLViewerDoc **文档相关联，所以每次文档中数据变化时，一旦调用** UpdateAllViews()**函数，这个树型视图也会相应地通过函数** OnUpdate()**而作出更新。因而，需要在函数** OnUpdate()**中根据当前文档中的模型数据来重新绘制树型控件的条目。**

要注意的是在创建树型控件的条目之前，应将树型控件中原先的所有条目清空之后再增加新条目，否则会造成条目的重复。

在 CPartTreeView **中，创建树型控件条目的实现代码如下：**

```
void CPartTreeView::OnUpdate(CView* pSender, LPARAM lHint, CObject* pHint)
{
    //获取所关联的文档对象的指针
    CSTLViewerDoc* pDoc = GetDocument();
    ASSERT(pDoc);
    CTreeCtrl& treeCtrl = GetTreeCtrl();

    //清空树型控件的所有条目
    treeCtrl.DeleteAllItems();

    HTREEITEM hti,hti0,hti1;
    CString str;

    //插入根条目
    str.Format("PART:%s",pDoc->m_Part.GetName());
    hti = treeCtrl.InsertItem(str,0,0,NULL,TVI_LAST);
    CEntity* ent;
```

```
        //根据文档中 m_Part 的内容创建树型控件的条目
        for(int i=0; i<pDoc->m_Part.GetEntitySize();i++){
            ent = pDoc->m_Part.GetEntity(i);
            ASSERT(ent);

            //在根条目下插入 STL 模型的条目
            str.Format("Model:%s",ent->GetName());
            hti0 = treeCtrl.InsertItem(str,1,1,hti,TVI_LAST);

            //将 STL 模型对象的指针值设为这个条目的数据
            //通过这个数据使该条目和模型对象关联
            treeCtrl.SetItemData(hti0,(DWORD)ent);

            //在 STL 模型条目下插入名称属性条目
            str.Format("Name:%s",ent->GetName());
            hti1 = treeCtrl.InsertItem(str,2,2,hti0,TVI_LAST);

            //为名称属性条目设置数据 ITEM_OF_NAME
            treeCtrl.SetItemData(hti1,ITEM_OF_NAME);

            //在 STL 模型条目下插入颜色属性条目
            COLORREF clr = ent->GetColor();
            BYTE r = GetRValue(clr);
            BYTE g = GetGValue(clr);
            BYTE b = GetBValue(clr);
            str.Format("Color: R%d G%d B%d",r,g,b);
            hti1 = treeCtrl.InsertItem(str,2,2,hti0,TVI_LAST);

            //为颜色属性条目设置数据 ITEM_OF_COLOR
            treeCtrl.SetItemData(hti1,ITEM_OF_COLOR);
            //在 STL 模型条目下插入显示属性条目
            if(ent->IsVisible())
                str = "Visible";
            else
                str = "Unvisible";
            hti1 = treeCtrl.InsertItem(str,2,2,hti0,TVI_LAST);

            //为显示属性条目设置数据 ITEM_OF_ VISIBLE
            treeCtrl.SetItemData(hti1,ITEM_OF_VISIBLE);
        }

        //将根目录下的条目展开
        GetTreeCtrl().Expand(hti,TVE_EXPAND);
    }
```

8.5.6 通过树型视图控件对文档数据进行操作

在应用程序 STLViewer 中，在诸如名称、颜色和显示等条目上双击鼠标左键，可激活相应的属性对话框，从而修改模型的属性。

为每个条目都设置一个数值是实现条目和模型保持关联的关键。在函数 OnUpdate()中创建每个条目时，都使用 SetItemData()对条目设置一个 32 位的数据。条目通过这个数据与模型以及模型的具体属性保持关联。在函数 CPartTreeView::OnUpdate()中，给每个子模型（CSTLModel 的对象）的条目设置的数据是所代表的零件对象的指针值。对于子模型的属性条目，分别设置的数据是 ITEM_OF_NAME、ITEM_OF_COLOR 和 ITEM_OF_VISIBLE，分别关联子模型的名称、颜色、显示属性。CTreeCtrl 还提供了函数 GetItemData()来获取条目中的数据。

这样，当鼠标左健双击到一个属性条目时，可以通过这个条目的数据来判别这个条目关联文档中模型的什么属性（数据 ITEM_OF_NAME、ITEM_OF_COLOR、ITEM_OF_VISIBLE 中的一个），进一步可以取出它的父条目中的数据（指向 CSTLModel 对象的指针），从而可以确定这个属性隶属于哪一个子模型对象。

函数 HitTest()用于根据鼠标当前位置确定相关的条目句柄。它输入鼠标当前位置，返回处于鼠标位置下的条目句柄。其原型为：

```
HTREEITEM    HitTest(CPoint pt,UINT* pFlags);
HTREEITEM    HitTest(TVHITTESTINFO* pHitTestInfo);
```

返回值：如果函数调用成功，则返回指定位置处的条目句柄。如果指定位置处没有条目存在，则返回 NULL。

函数 CPartTreeView::OnLButtonDblClk()响应鼠标左键双击事件，根据鼠标选中的条目来弹出一个属性对话框，用于修改与条目相关联的 CSTLModel 子模型的属性，如名称、颜色、显示/隐藏。它的实现代码如下：

```
//鼠标消息 WM_LBUTTONDBLCLK 的响应函数
void CPartTreeView::OnLButtonDblClk(UINT nFlags, CPoint point)
{
    UINT uFlags = 0;
    CTreeCtrl& treeCtrl = GetTreeCtrl();

    //得到鼠标当前位置处的条目句柄
    HTREEITEM hItem = treeCtrl.HitTest(point,&uFlags);

    //确定该条目是属性条目
    if(   hItem &&                                  //必须有条目返回
          hItem != treeCtrl.GetRootItem() &&        //返回的条目不为根条目
          !treeCtrl.ItemHasChildren(hItem)){        //返回的条目没有子条目

          //获取该条目的数据，即属性类别
          DWORD itemData = treeCtrl.GetItemData(hItem);
```

```
            //获取该条目的父条目的句柄
            HTREEITEM hParentItem = treeCtrl.GetParentItem(hItem);
            ASSERT(hParentItem);

            //获取父条目的数据，即指向 CSTLModel 子模型的指针
            DWORD parentData = treeCtrl.GetItemData(hParentItem);
            ASSERT(parentData);

            //将父条目的数据转化为指向对象的指针，即将要被修改属性的对象的指针
            CEntity* ent = (CEntity*) parentData;

            //根据所选条目的数据来判断修改什么属性
            switch(itemData){
                case ITEM_OF_NAME:          //修改名称属性
                    ChangeEntityName(ent);
                    break;
                case ITEM_OF_COLOR:         //修改颜色属性
                    ChangeEntityColor(ent);
                    break;
                case ITEM_OF_VISIBLE:       //修改显示属性
                    ChangeEntityVisible(ent);
                    break;
                }
                }
        CTreeView::OnLButtonDblClk(nFlags, point);
}

//修改名称属性
void CPartTreeView::ChangeEntityName(CEntity* ent)
{
        CSTLViewerDoc* pDoc = GetDocument();
        ASSERT(pDoc);
        CDlgEntityName dlg;
        dlg.m_EntityName = ent->GetName();
        if(dlg.DoModal() == IDOK){                  //弹出修改名称对话框（见图 8-9）
            ent->SetName(dlg.m_EntityName);         //更新名称
            pDoc->UpdateAllViews(NULL);             //更新视图显示
        }
}
//修改颜色属性
void CPartTreeView::ChangeEntityColor(CEntity* ent)
{
        CSTLViewerDoc* pDoc = GetDocument();
        ASSERT(pDoc);
        CColorDialog dlg;                           //定义颜色编辑对话框
        if(dlg.DoModal() == IDOK){                  //弹出修改颜色对话框（见图 8-10）
            COLORREF color = dlg.GetColor();
```

```
            ent->SetColor(color);                //更新名称
            pDoc->UpdateAllViews(NULL);          //更新视图
        }
    }
```

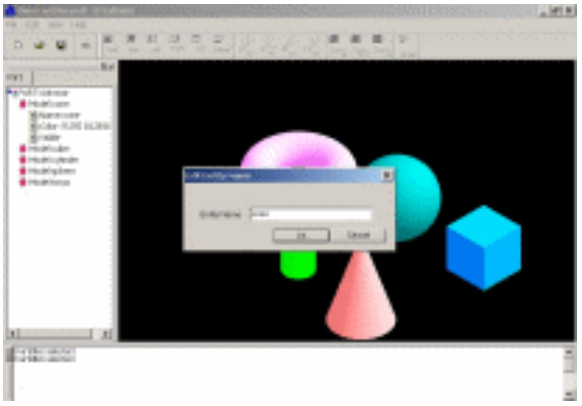


图 8-9　修改名称对话框

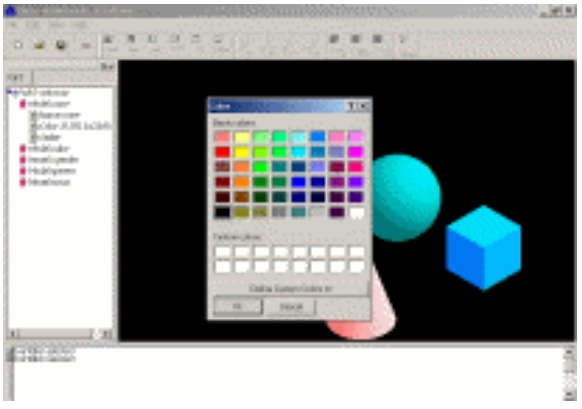


图 8-10　修改颜色对话框

```
//修改显示属性
void CPartTreeView::ChangeEntityVisible(CEntity* ent)
{
    CSTLViewerDoc* pDoc = GetDocument();
    ASSERT(pDoc);
    ent->SetVisible(!ent->IsVisible());          //修改显示属性
    pDoc->UpdateAllViews(NULL);                  //更新视图
}
```

本章相关程序

- ch8\DockTool：**具有浮动风格的界面工具库** DockTool.DLL **的工程。**

- ch8\Enhanced STLViewer：界面增强后的应用程序 STLViewer 的工程。
- ch8\inc：工程 STLViewer 所需要的头文件。
- ch8\lib：工程 STLViewer 所需要的静态库。
- ch8\bin：执行文件 Enhanced STLViewer.exe 和它所需要的动态库。
- ch8\Models：供执行程序 Enhanced STLViewer.exe 调用的 STL 模型和 mdl 文件。

第 9 章　基于 OpenGL 的 CAD 软件拾取功能的实现

本章要点：

- **使用** OpenGL **选择模式的有关技术。**
- **在** C++**类中对** OpenGL **选择功能的封装。**
- OpenGL **的选择功能与** CAD **应用程序的集成。**

在大多数 CAD **软件中，除了需要绘制出二维或三维物体的静态场景以外，还需要实现用户和图形之间的交互操作。用户可以通过菜单、工具条、对话框等界面对象来实现与图形的交互，而一种更直接的方法是用户和图形对象之间的直接交互操作。如在** Windows **程序中，用鼠标选择物体，并对所选取的物体进行移动、修改、删除等操作，就是一个最常用的用户与图形之间的交互操作。**

用鼠标选择物体的关键是要确定场景中哪些物体在鼠标位置之下，这个在场景中用鼠标选择物体的过程也称为拾取。在一些应用程序中，由用户自己设计拾取算法可能并不麻烦，例如对二维的绘图程序，很容易计算二维线框是否与鼠标的拾取框相交，并设计一些优化的搜索方法来减少计算量。但对于基于 OpenGL **的三维应用程序，由于绘制在场景中的几何对象都经过了在空间中的多次旋转、平移和投影变换等，由用户直接设计算法来算出所拾取的对象会比较困难。**OpenGL **的选择模式（**Selection Mode **）为用户提供了一种拾取物体的机制，它要求在所绘制的对象之前命名，并返回在窗口中某个特定区域里绘出的所有对象的名称。使用** OpenGL **的选择模式是实现** CAD **图形交互操作的重要方法之一。**

在本章中，将使用 OpenGL **的选择模式，为应用程序** STLViewer **增加使用鼠标拾取物体的功能，如图** 9-1 **所示。由于在** STLViewer **应用程序中增加了鼠标拾取功能，单击鼠标左键，可以拾取到相应对象，并将它高亮度（**highlight**）显示。**

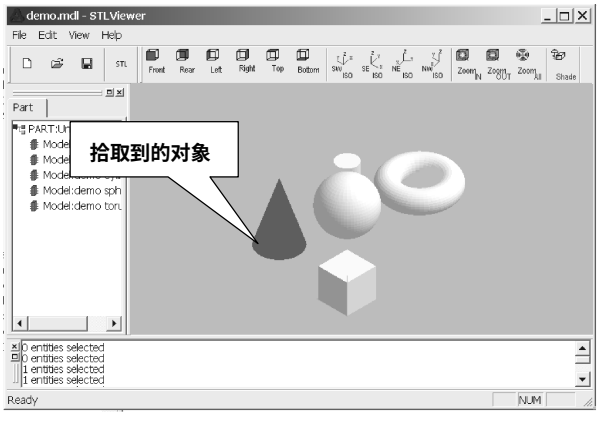


图 9-1　**在** STLViewer **中用鼠标拾取物体**

9.1 使用 OpenGL 选择模式

9.1.1 OpenGL 的三种操作模式

OpenGL 提供了三种操作模式：渲染模式（Render Mode）、选择模式（Selection Mode）和反馈模式（Feedback Mode）。

渲染模式是 OpenGL 的正常模式，也是默认模式。在这个模式下，所要绘制的几何图元被光栅化，并在计算成位图像素后被送到帧缓存区中。正常的图形绘制是在渲染模式下实现的。

在选择模式下，OpenGL 用户根据鼠标的位置在 OpenGL 中定义一个视景体，这个视景体的形状有如一条细长的射线。在绘制一个对象之前需要对它进行命名，OpenGL 将落在这个视景体之内的物体的名称存储在一个堆栈中。通过返回这个堆栈中的物体名称，可以确定拾取到的物体。选择模式和渲染模式的不同在于：在选择模式下，OpenGL 生成的场景（像素）并不真正被复制到帧缓存区中以供显示，而是根据在这个特殊的视景体中所绘制了的物体，生成拾取记录，并把这些生成的拾取记录存储于选择缓存区中。

反馈模式下，OpenGL 不生成光栅化的像素，也不往帧缓存区中写内容，而是将所要绘制的几何图元信息反馈到一个预先创建好的反馈内存区中（预先定义的浮点型数组）。这些信息包括图元类型（点、线、多边形等）的标志、图元的顶点、颜色或其他数据。

选择模式和反馈模式的一个共同点是：OpenGL 将不发送光栅化的图像到帧缓存区中，而是将有关信息返回给应用程序中分配的相关缓存区中。要使用 OpenGL 的选择模式或反馈模式，必须通过调用模式切换函数，将当前模式切换至选择模式或反馈模式。OpenGL 使用函数 glRenderMode()进行模式切换，对该函数的介绍如下。

原型：GLint glRenderMode(GLenum mode)

说明：设置 OpenGL 的操作模式，操作模式是渲染、选择、反馈中的一种，OpenGL 的默认操作模式是渲染模式。要注意的是，在将模式切换到选择模式之前，必须先使用函数 glSelectBuffer()创建选择缓存区；在将模式切换到反馈模式之前，必须先使用函数 glFeedbackBuffer()创建反馈缓存区。OpenGL 将有关信息返回到这些缓存区中。

参数：mode，定义 OpenGL 的操作模式。参数 mode 的取值可以为下面之一。

GL_SELECTION——切换至选择模式；

GL_RENDER——切换至渲染模式；

GL_FEEDBACK——切换至反馈模式。

返回值：函数的返回值取决于该函数被调用时 OpenGL 的当前操作模式，而不是取决于参数 mode 的值。当前操作模式为 GL_SELECTION 时，函数的返回值为选择缓存区中名称的数目；为 GL_RENDER 时，函数的返回值为 0；为 GL_FEEDBACK 时，函数的返回值为反馈缓存区中名称的数目。

9.1.2 使用选择模式

实际上，OpenGL 的选择模式和渲染模式下的场景绘制计算是完全相同的，只是两者所定义的视景体的大小不同。在选择模式下，用户需要使用函数 gluPickMatrix()定义一个与鼠标

位置相关的狭长形的视景体（即选择视景体），场景绘制只在这个选择视景体中进行。两者的区别在于：在渲染模式下，场景计算结果将输送到屏幕的显示帧缓存中；而在选择模式下，不往显示帧缓存中送计算结果，而是向应用程序所定义的选择缓存区（Selection Buffer）中输出一个名称列表。列表中的名称也称为命中记录（Hit Records），它包括所有空间位置位于该选择视景体中或与它相交的几何对象的名称。

选择缓存区和名称堆栈是应用 OpenGL 选择模式时的两个重要概念。使用选择模式时，必须首先给 OpenGL 分配一个选择缓存区，即一个内存中的数组。绘制场景的过程中，需要创建一个名称堆栈。在绘制每一个物体之前，需要给它分配一个合适的名称（一个 32 位整型数），并将这个名称压入（Push）或填写（Load）在名称堆栈中。最后在退出选择模式时，OpenGL 返回一列与选择视景体发生相交的对象的名称（命中对象）到选择缓存区中，用户由此来判断哪些物体与选择视景体发生相交。

使用选择模式的基本操作过程及相关操作函数如图 9-2 所示。

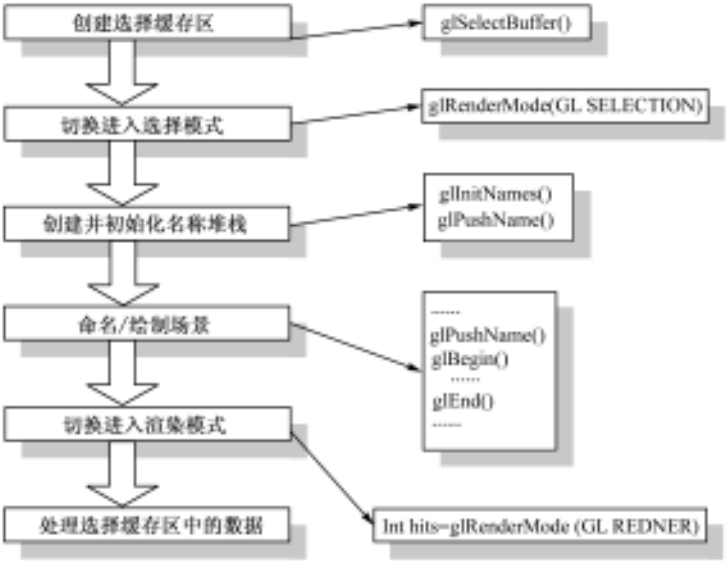


图 9-2　使用选择模式的主要步骤及相关操作函数

1．名称堆栈

OpenGL 通过名称堆栈（Name Stack）对对象进行命名。对于具有层次结构的名称（如对一个总模型命名后，又对构成这个总模型的子模型或组成元素命名），在名称堆栈中，还可建立对象的名称层次关系，以标记模型的组成结构。物体的名称由 32 位的无符号整型数(unsigned int) 来定义。在绘制物体前，需要使用函数 glInitNames()创建并初始化名称堆栈，将堆栈清空。然后，在绘制每个需要命名的物体之前，可以使用函数 glPushName()将物体的名称压入堆栈，用完后使用函数 glPopName()将名称弹出，或是使用函数 glLoadName()将名称填写在栈顶元素中。

在选择模式下，可以给 OpenGL 中的每一个图形元素命名，如一个三角面片；也可以对组成一个对象的一组图元命名。通常情况下，应用程序中只需要对由一组基本图形图元组成、用户感兴趣的对象命名。如在图 9-1 所示的应用中，用户只需要返回鼠标所拾取的是哪一个

几何对象(一个 CSTLModel 的对象),这时只需要对一个由一系列三角面片组成的 CSTLModel 的对象命名。

在选择模式下，当一个拾取记录产生时，所有在名称堆栈中的名称被复制到选择缓存区中。这样，对于层次结构的名称堆栈，一个拾取记录中可能包含多个物体名称。对于不具有层次结构的名称堆栈，由于堆栈中始终只有一个名称，所以一个拾取记录只可能包含一个名称。下面是一个创建名称堆栈的简单例子，这个例子用于对 5 个小球的命名。每个小球都由许多三角面片构成，但程序以小球为命名对象，无论鼠标拾取到小球的任何部位，都返回小球的名称。由于在名称堆栈中没有采用层次结构，即名称堆栈中始终都只有一个名称，所以一个拾取记录只会有一个物体名称。如果在鼠标位置下覆盖了两个小球，则在选择模式退出时，在选择缓存区中有两个拾取记录，每个记录中有一个小球的名称。

```
#define RED_SPHERE1
#define GREEN_SPHERE   2
#define BLUE_SPHERE    3
#define WHITE_SPHERE   4
#define GRAY_SPHERE    5

glInitNames();                  //初始化名称堆栈
glPushName(0);                  //往堆栈中压入一个元素，使堆栈不为空

glLoadName(RED_SPHERE);         //往栈顶元素中填入名称 RED_SPHERE
drawRedSphere();                //绘制红色球

glLoadName(BLUE_SPHERE);        //往栈顶元素中填入名称 BLUE_SPHERE
drawBlueSphere();               //绘制蓝色球

glLoadName(GREEN_SPHERE);       //往栈顶元素中填入名称 GREEN_SPHERE
drawGreenSphere();              //绘制绿色球

glLoadName(WHITE_SPHERE);       //往栈顶元素中填入名称 WHITE_SPHERE
drawWhiteSphere();              //绘制白色球

glLoadName(GRAY_SPHERE);        //往栈顶元素中填入名称 GRAY_SPHERE
drawGraySphere();               //绘制灰色球
```

下面介绍 OpenGL 名称堆栈管理的几个相关函数：

（1）函数 void glInitNames(void)。函数将名称堆栈初始化为清空的状态。

（2）函数 void glLoadName(GLuint name)。函数向名称堆栈的栈顶元素装载一个名称，即使用参数 name 替换栈顶元素。使用该操作的前提是堆栈不为空。

（3）函数 void glPushName(GLuint name)。函数向名称堆栈的栈顶元素压入一个名称。使用该操作时堆栈可以为空。

（4）函数 void glPopName(void)。函数将栈顶元素弹出堆栈。

2．选择缓存区

在 OpenGL 选择模式中，通过建立一个选择缓存区（Selection Buffer），以返回 OpenGL 的拾取记录。选择缓存区由一个无符号整型（unsigned int）的数组构成。

在选择模式下，一个被命名了的对象如果和所定义的选择视景体有重合，这个对象便成为一个拾取记录（Hit Record）。OpenGL 在处理场景命令的过程中，将发生的拾取记录填写在选择缓存区中。拾取记录可能不止一个，因为和鼠标重叠的物体可能会有多个。

一个拾取记录包含以下信息：名称的个数、深度值和拾取到的名称。这样，每个拾取记录由至少四个元素组成。第一个元素标识了当拾取发生时名称堆栈中名称的数目，即这个记录中包含的名称的数目。当命名具有层次关系时，会存在不止一个的名称数目。第二和第三个元素分别标识了当拾取发生时，在视景体中对象的最小和最大的 z 坐标值相对于视景体深度的比值。这个比值的范围是 0～1，将元素中的值(unsigned int) 除以无符号整型数的最大值(2^{32}-1)，就是这个深度比值。当在选择缓存区中存在多个拾取记录时，可以用这个深度比值来判断所拾取到物体的前后位置关系。从第四个元素开始，是拾取到的对象的名称，名称的顺序根据进入名称堆栈中的顺序排列。

图 9-3 是两个选择缓存区的例子。图 9-3a 中一共存在两个拾取记录，其中第二个拾取记录存在三个名称。一个记录中存在多个名称只会在具有层次结构的名称堆栈中发生。在前面创建名称堆栈的例子中，如果鼠标同时命中红色球和蓝色球，则在选择缓存区中的情况如图 9-3b 所示，每个记录中只有一个名称。

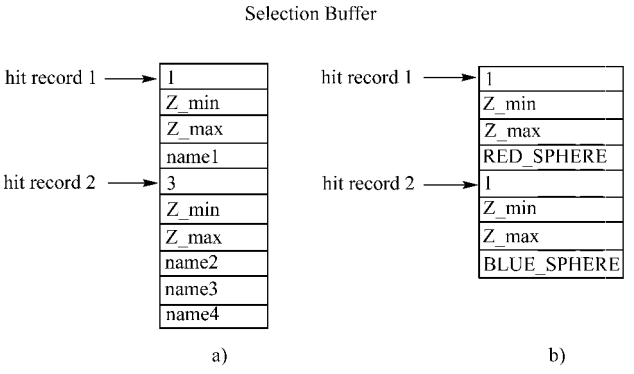


图 9-3　选择缓存区中的拾取记录

用户无法直接从选择缓存区中确定其中所包含的拾取记录的数目，因为该缓存区中没有一个元素用于标识它包含记录的个数。选择缓存区直到从选择模式切换回绘图模式时才被填满。当调用函数 glRenderMode（GL_RENDER）切换回绘图模式时，函数返回该选择缓存区中拾取记录的数目。下列代码返回所获得的拾取记录数目：

```
int hits = glRenderMode(GL_RENDER);
```

OpenGL 函数 glSelectBuffer()用于设置选择缓存区。

原型：void glSelectBuffer(GLsizei size, GLuint *buffer)

说明：函数为 OpenGL 的选择模式指定选择缓存区，即一个 32 位的整型数组。

参数：size——**缓存区数组的尺寸；**

buffer——**指向缓存区数组的指针，返回的数据就存放在该数组中。**

3．**定义拾取范围**

当鼠标按下时，需要根据鼠标按下的位置定义一个有效的拾取区域，在这个区域内的物体都被选中，这个区域是三维的。正如上文提到的，需要根据鼠标的位置信息定义一个特殊的视景体，凡在这个视景体之内绘制的物体都被选中。

OpenGL **中，这个有效拾取区域是利用拾取矩阵和投影变换共同定义的。函数** gluPickMatrix()**根据鼠标位置定义一个特殊的拾取矩阵，这个拾取矩阵与在渲染模式下的投影矩阵相乘，将产生一个以鼠标位置为中心的狭长形的选择视景体。建立了这个特殊的视景体后，在鼠标位置之下绘制的物体就会产生拾取记录。这时，虽然改变了视景体的大小，但由于** OpenGL **的场景计算结果并不往显示缓存区中送，所以对屏幕显示的场景没有影响。**

对 OpenGL **函数** gluPickMatrix ()**介绍如下：**

原型：void gluPickMatrix(GLdouble x, GLdouble y,
GLdouble width, GLdouble height,
GLint viewport[4]);

说明：该函数定义一个拾取区域用于处理用户选择。使用 gluPickMatrix()**之前，必须首先保存当前投影矩阵的状态。然后将投影矩阵初始化为单位矩阵，调用** gluPickMatrix()**创建拾取矩阵。最后，还需要乘上一个投影矩阵，这个投影矩阵应该和绘制场景时的投影矩阵一致。**

参数：

x, y——**矩形拾取区域的中心坐标（**Windows **窗口坐标系），通常就是鼠标相对于窗口的坐标；**

width, height——**矩形拾取区域的宽和高（**Windows **窗口坐标系）；**

viewport[4]——**当前的视口坐标。可以使用函数** glGetIntegerv()**及参数** GL_VIEWPORT **来获得当前的视口坐标。**

4．**具有层次结构的名称堆栈**

在上述五个小球的例子中，对于每个对象的命名，采用的是覆盖栈顶元素的方法，即在绘制每个对象之前，将栈顶元素的值替换掉。这样在名称堆栈中永远只有一个元素，即当前要绘制的对象名称。在这种方式下产生的一个拾取记录只包含一个名称。也可以使用 glPushName()**的方式在名称堆栈中压入多个名称，这样产生的拾取记录中会包含多个名称。这样的名称堆栈方式是非常有用的，尤其是所绘制的对象之间具有隶属关系或层次关系时。例如，一个** STL **模型由一系列三角面片组成，拾取到一个三角面片的同时，还需要知道这个三角面片是隶属于哪一个** STL **模型。利用** glPushName()**，将** STL **模型、三角面片名称都保存在堆栈中，这样一个拾取记录生成时，就会包含这两个名称。由此可以知道鼠标拾取到的三角面片，以及这个三角面片所隶属的** STL **模型。这就是层次结构的名称堆栈。**

下面，我们改写上述五个圆球的例子。如图 9-4a **所示，将它们分为两组，即** Left_Group **和** Right_Group**。** Left_Group **包括** RED_SPHERE**、** BLUE_SPHERE**；**Right_Group **包括** GREEN_SPHERE**、**WHITE_SPHERE**、**GRAY_SPHERE**。创建这样具有层次结构的名称堆栈的代码如下：**

```
#define LEFT_GROUP      1
#define RIGHT_GROUP     2

#define RED_SPHERE      1
#define GREEN_SPHERE    2
#define BLUE_SPHERE     3
#define WHITE_SPHERE    4
#define GRAY_SPHERE     5

glInitNames();                          //初始化名称堆栈
glPushName(0);                          //往堆栈中压入一个元素，使堆栈不为空

glLoadName(LEFT_GROUP);                 //LEFT_GROUP
    glPushName(RED_SPHERE);             //往栈顶元素中填入名称 RED_SPHERE
    drawRedSphere();                    //绘制红色球
    glPopName();

    glPushName(BLUE_SPHERE);            //往栈顶元素中填入名称 BLUE_SPHERE
    drawBlueSphere();                   //绘制蓝色球
    glPopName();

glLoadName(RIGHT_GROUP);                //RIGHT_GROUP
    glPushName(GREEN_SPHERE);           //往栈顶元素中填入名称 GREEN_SPHERE
    drawGreenSphere();                  //绘制绿色球
    glPopName();

    glPushName(WHITE_SPHERE);           //往栈顶元素中填入名称 WHITE_SPHERE
    drawWhiteSphere();                  //绘制白色球
    glPopName();

    glPushName(GRAY_SPHERE);            //往栈顶元素中填入名称 GRAY_SPHERE
    drawGraySphere();                   //绘制灰色球
    glPopName();
```

如果鼠标拾取到左边的红球时，获得的拾取记录如图 9-4b **所示，它包括两个名称：**LEFT_GROUP **和** RED_SPHERE**，表示所拾取到的是红色球，且它属于左边的一组。**

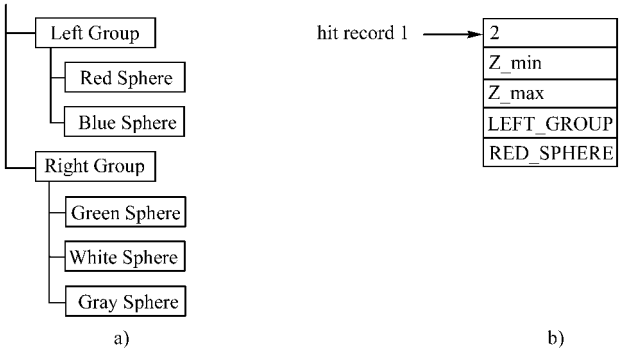


图 9-4　**使用具有层次结构的名称堆栈**

9.2 一个 OpenGL 选择模式的应用程序

下面将通过一个简单的例子来介绍 OpenGL 选择模式的基本操作。在这个例子的视图窗口中绘制了五个不同颜色的小球。单击鼠标左键来选择一个小球，若有小球被选中，则会弹出相应的信息窗来提示选中的小球的名称。在本书的第 3 章中创建了一个最基本的 OpenGL Windows 程序 GL，可以在 GL 的基础上进行修改，来创建一个具有鼠标拾取功能的 OpenGL 程序。修改过程如下：

（1）在类 CGLView 中增加处理鼠标拾取的函数 ProcessSelection()。

```
protected:
    void ProcessSelection(int xPos,int yPos);
```

其中，参数 xPos 和 yPos 是鼠标位置的窗口像素坐标。

（2）在类 CGLView 中增加鼠标消息 WM_LBUTTONDOWN 的响应函数 OnLButtonDown()。

```
protected:
    //{{AFX_MSG(CGLView)
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

（3）修改虚拟函数 RenderScene()。在 RenderScene()中，我们绘制五个不同颜色的小球。首先需要初始化名称堆栈，然后在绘制每个小球之前，需要分别给它们命名。名字应该是一个无符号整型数 (unsigned int)。

```
#define RED_SPHERE      1
#define GREEN_SPHERE    2
#define BLUE_SPHERE     3
#define WHITE_SPHERE    4
#define GRAY_SPHERE     5

void CGLView::RenderScene()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    //初始化名称堆栈
    glInitNames();
    //往堆栈中压入一个元素，使堆栈不为空
    glPushName(0);

    //往栈顶元素中填入名称 RED_SPHERE
    glLoadName(RED_SPHERE);
    glPushMatrix();
    glColor3ub(255,0,0);          //red
    glTranslatef(-800,0,0);       //-800,0,0
    auxSolidSphere(360);
    glPopMatrix();
```

```
    //往栈顶元素中填入名称 BLUE_SPHERE
    glLoadName(BLUE_SPHERE);
    glPushMatrix();
    glColor3ub(0,0,255);          //blue
    glTranslatef(-1300,0,0);      //-1300,0,0
    auxSolidSphere(120);
    glPopMatrix();

    //往栈顶元素中填入名称 GREEN_SPHERE
    glLoadName(GREEN_SPHERE);
    glPushMatrix();
    glColor3ub(0,255,0);          //green
    glTranslatef(600,0,0);        //600,0,0
    auxSolidSphere(60);
    glPopMatrix();

    //往栈顶元素中填入名称 WHITE_SPHERE
    glLoadName(WHITE_SPHERE);
    glPushMatrix();
    glColor3ub(255,255,255);    //white
    glTranslatef(800,300,0);      //800,300,0
    auxSolidSphere(60);
    glPopMatrix();

    //往栈顶元素中填入名称 GRAY_SPHERE
    glLoadName(GRAY_SPHERE);
    glPushMatrix();
    glColor3ub(128,128,128);    //gray
    glTranslatef(800,-300,0);     //800,-300,0
    auxSolidSphere(160);
    glPopMatrix();

    glFlush();
}
```

（4）填写拾取处理函数 ProcessSelection()。**函数** ProcessSelection()**在鼠标事件响应函数中被调用，用来生成并处理拾取记录。由于名称堆栈中没有使用层次结构的命名，所以拾取记录中的第** 3 **个元素就是所选中对象的名称。根据这个名称（**32 **位整型值），找到对应的文字名称，并使用** Windows Message **窗口提示拾取到的物体。**

```
void CGLView::ProcessSelection(int xPos,int yPos)
{
    GLuint      selectBuff[64];
    Glint       hits,viewport[4];

    //建立选择缓存区
    glSelectBuffer(64,selectBuff);
```

```
//获取当前视口信息
glGetIntegerv(GL_VIEWPORT,viewport);
yPos = viewport[3]-yPos;

//设置投影变换
glMatrixMode(GL_PROJECTION);
glPushMatrix();

//切换到选择模式
glRenderMode(GL_SELECT);

//根据鼠标位置，定义选择矩阵
//拾取区域的宽度和高度分别为 1
glLoadIdentity();
gluPickMatrix(xPos,yPos,1,1,viewport);

//乘以投影矩阵，这个投影矩阵应该和绘制场景时的投影矩阵相一致
double nRange = 1200.0;
int cx = viewport[2]-viewport[0];
int cy = viewport[3]-viewport[1];

if(cx <= cy)
    glOrtho(-nRange,nRange,-nRange*cy/cx,nRange*cy/cx,-nRange,nRange);
else
    glOrtho(-nRange*cx/cy,nRange*cx/cy,-nRange,nRange,-nRange,nRange);

//在选择模式下将景物绘制一遍
RenderScene();

//切换到绘图模式，并返回选择缓存区中拾取记录的数目
hits = glRenderMode(GL_RENDER);
//如果存在拾取记录，则从选择缓存区中获取拾取到的物体的名称，并显示该名称
if(hits > 0){
    switch(selectBuff[3]){          // selectBuff[3]是选中对象的名称
    case RED_SPHERE:
        AfxMessageBox("Red Sphere");
        break;
    case GREEN_SPHERE:
        AfxMessageBox("Green Sphere");
        break;
    case BLUE_SPHERE:
        AfxMessageBox("Blue Sphere");
        break;
    case WHITE_SPHERE:
        AfxMessageBox("White Sphere");
```

```
                break;
            case GRAY_SPHERE:
                AfxMessageBox("Gray Sphere");
                break;
            }
        }

        //恢复投影矩阵
        glMatrixMode(GL_PROJECTION);
        glPopMatrix();

        //返回视图矩阵
        glMatrixMode(GL_MODELVIEW);
    }
```

（5）填写鼠标消息处理函数 OnLButtonDown()**。在函数** OnLButtonDown()**中，调用函数** ProcessSelection()**来处理物体的拾取。在处理拾取之前，需要先将渲染场境当前化。**

```
void CGLView::OnLButtonDown(UINT nFlags, CPoint point)
{
    wglMakeCurrent(m_hDC,m_hRC);
    ProcessSelection(point.x,point.y);
    wglMakeCurrent(m_hDC,NULL);
    CView::OnLButtonDown(nFlags, point);
}
```

（6）运行程序。编译连接并运行程序。如图 9-5 **所示，单击鼠标左键，若有球被选中，屏幕上则弹出相应的信息窗口，以提示所选中小球的名称。**

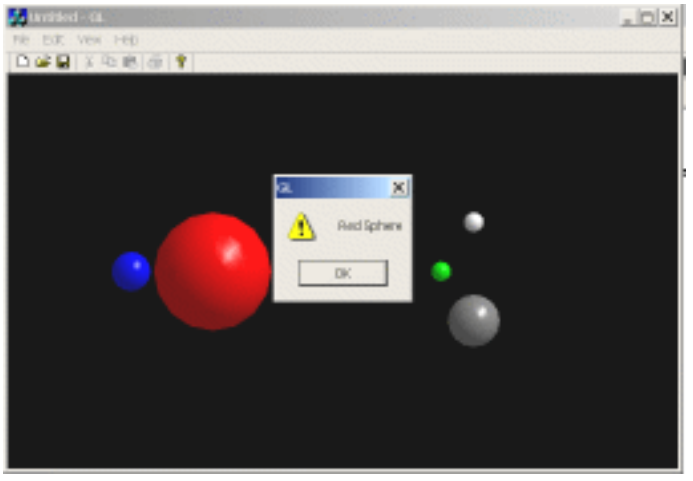


图 9-5　**使用鼠标选择**

9.3 OpenGL 的选择功能与 CAD 应用程序的集成

以上对 OpenGL 的选择模式及其使用作了介绍。但对于具有一定复杂程度的 CAD 系统，不太可能以上述的编程方式来实现拾取功能，而是应该将拾取的操作过程从视图类中分离出来。这样有利于程序的模块化，也便于实现在 CAD 应用程序中集成 OpenGL 的选择功能。要实现这样的集成将会涉及到更多一些的问题，例如：

（1）渲染模式和选择模式下的场景绘制。根据 OpenGL 选择模式的要求，在这两个模式下的场景绘制的内容和顺序、投影矩阵都应该完全一致。只是在选择模式下的场景绘制中，需要增加对绘制对象的命名。

（2）选择对象的自动命名。CAD 应用程序中几何模型的管理是动态的，不可能像上述例子的程序代码中为每个拾取对象人工起个名字，而是需要一个自动的命名机制。

（3）对 OpenGL 选择功能的封装。我们已经开发了一个 OpenGL 的封装类，将与选择过程有关的代码从视图类中分离了出来，并将它们封装在类 COpenGLDC 中，这将有利于程序的模块化，也使在应用程序中对 OpenGL 选择功能的操作更加方便。

（4）选择缓存区返回信息的处理。在一个选择过程结束后，需要对获得的拾取记录进行处理，例如通过名称找到相对应的物体。存在多个拾取记录时，有时还要通过记录中的深度信息来判断哪一个是位于最上面的。

在上述的章节中，已经设计了几个动态库，其中 glConext.dll 和 geomKernel.dll 分别支持对 OpenGL 的封装和对几何模型的管理。为使这两个库支持 OpenGL 的选择功能，下面将分别对它们做必要的修改。

9.3.1 定义选择视景体/修改类 CCamera

OpenGL 选择功能的一个重要环节就是根据鼠标位置定义了一个狭长型的选择视景体作为选择区域。对视景体的定义和操作是在照相机类中完成的，需要对照相机类 CCamera 做一些修改，以增加由用户定义选择区域的功能。

（1）在 CCamera 中增加成员函数。

```
public:
void selection(int xPos,int yPos);
```

函数 selection()将当前模式切换到选择模式，并根据鼠标位置和渲染模式下的视景体设置来创建选择视景体。函数 selection()和函数 project()的功能相对应。project()用于定义渲染模式下的视景体；selection()用于定义选择模式下的选择视景体，输入参数 xPos、yPos 是鼠标拾取位置的窗口像素坐标。

（2）填写函数 selection()。

```
void GCamera::selection(int xPos,int yPos)
{
    GLintvp[4];

    //获取当前视口信息
```

```
    glGetIntegerv(GL_VIEWPORT,vp);

    //设置投影变换
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    //切换到选择模式
    glRenderMode(GL_SELECT);

    //根据鼠标位置，定义选择矩阵。这里拾取范围的宽度和高度分别设为固定值 1，也可以
      在类中将这个拾取范围设为参数可调节的
    gluPickMatrix(xPos,vp[3]-yPos, 1, 1, vp );

    //乘以投影矩阵，这个投影矩阵应该和函数 projection()中的投影矩阵相一致
    double left     =   - m_width/2.0;
    double right    =   m_width/2.0;
    double bottom   =   - m_height/2.0;
    double top      =   m_height/2.0;

    glOrtho(left,right,bottom,top,m_near,m_far);

    glMatrixMode( GL_MODELVIEW );
    glLoadIdentity( );
    gluLookAt(m_eye.x,m_eye.y,m_eye.z,m_ref.x,m_ref.y,m_ref.z,
        m_vecUp.dx, m_vecUp.dy, m_vecUp.dz);
}
```

9.3.2 对选择过程的操作/修改类 COpenGLDC

对类 COpenGLDC 的修改如下：

（1）在 COpenGLDC 中增加成员变量。

```
#define BUFFER_LENGTH 64                    //定义选择缓存区的长度

protected:
BOOL    m_bSelectionMode;                   //是否处于选择模式
GLuint  m_selectBuff[BUFFER_LENGTH];        //选择缓存区
```

其中，数组 m_selectBuff 用作选择缓存区，需要在有关函数中使用 glSelectBuff()将它设置选择缓存区后才能生效。变量 m_bSelectionMode 用于标识当前 OpenGL 是否处于选择模式下。

（2）在 COpenGLDC 中增加成员函数。

```
void    BeginSelection(int xPos,int yPos); //拾取操作开始
int     EndSelection(UINT* items);         //拾取操作结束，返回拾取记录
BOOL    IsSelectionMode();                 //判断是否处于选择模式

//以下函数封装了 OpenGL 对名称堆栈的操作，这样在应用程序中可以不用直接调用 OpenGL 函数
```

```
void    InitNames();
void    LoadName(UINT name);
void    PushName(UINT name);
void    PopName();
```

使用时，首先调用函数 BeginSelection()，**然后开始绘制图形。图形绘制结束后，调用** EndSelection()**以生成拾取记录，返回其中的物体数目和选择缓存区中的数据。**

（3）填写函数 BeginSelection()**。**

```
void COpenGLDC::BeginSelection(int xPos,int yPos)
{
    //标识拾取过程开始
    m_bSelectionMode = TRUE;

    //当前化渲染场境
    wglMakeCurrent(m_hDC,m_hRC);

    GLintviewport[4];

    //设置选择缓存区
    glSelectBuffer(BUFFER_LENGTH,m_selectBuff);

    //切换到选择模式下，并根据鼠标位置创建选择视景体
    m_Camera.selection(xPos,yPos);

    glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );

    //初始化名称堆栈
    InitNames();
}
```

（4）填写函数 EndSelection()**。**

```
int    COpenGLDC::EndSelection(UINT* items)
{
    //标识拾取过程结束
    m_bSelectionMode = FALSE;

    //切换到绘图模式，并返回选择缓存区中拾取记录的数目
    int hits = glRenderMode(GL_RENDER);

    //返回拾取记录
    for(int i=0;i<hits;i++){
    items[i] = m_selectBuff[I*4+3];
    }

    //非当前化渲染场境
    wglMakeCurrent(m_hDC,NULL);
```

```
    return hits;
}
```

（5）填写函数 IsSelectionMode()。

```
BOOL COpenGLDC::IsSelectionMode()
{
    return m_bSelectionMode;
}
```

（6）填写名称堆栈管理函数。

```
void COpenGLDC::InitNames()
{
    glInitNames();
}

void COpenGLDC::LoadName(UINT name)
{
    glLoadName(name);
}

void COpenGLDC::PushName(UINT name)
{
    glPushName(name);
}

void COpenGLDC::PopName()
{
    glPopName();
}
```

9.3.3 自动给对象命名/对类 CSTLModel 的修改

在选择模式下，绘制每个可被拾取的对象之前需要使用名称堆栈对这些对象命名。在库 GemoKernel **中已经设计了描述** STL **模型的类** CSTLModel**。在** STLViewer **中，我们将一个** CSTLModel **的对象作为一个拾取单元。**CSTLModel **中提供了使用** COpenGLDC **绘制对象本身的函数** Draw()**。为了使它支持命名，需要对函数** CSTLModel::Draw()**做一些修改，即判断若当前是在选择模式下，则在绘制自己之前需要给自己命名，并将名称载入名称堆栈中。**

问题的关键是要为这个对象自动起一个名字。这个名字应该具有惟一性，且通过这个名字还能方便地获得它所关联的对象。基于以上考虑，使用指向对象的指针作为这个对象的名称应该是一个理想的解决方法。首先，任何对象的指针都是 32 **位的无符号整型数，这和** OpenGL **的命名要求相一致。其次，指针的值代表所指对象在实际存储空间的地址。一个应用程序中，不同的对象在内存中的存储地址是不同的，因而指针的值具有惟一性。另外，使用指针值作为对象名称的方便之处还在于，通过指针可以直接找到所指向的对象。**

在一个 C++**类中，关键字** this **表示指向自己的指针。函数** Draw()**中，在绘制对象本身之前，可以通过** this **来获得指针值，将** this **强制转换为无符号整型数（**UINT**）后，就可用于对**

象的命名。因为只有在选择模式下才需要在绘制对象之前给对象命名，所以在命名之前首先要判断当前是否处于选择模式。

```
void  CSTLModel::Draw(COpenGLDC* pDC)
{
    if(!m_bVisible)   return;      //如果处于隐藏模式，则不做任何绘制

    if(m_bHighlight)
        pDC->Highlight(TRUE); //如果处于高亮度模式，使用高亮度颜色绘制
    else
        pDC->SetMaterialColor(m_color); //正常模式下，使用对象本身的材料颜色

    //当前是否处于选择模式
    if(pDC->IsSelectionMode()){
    //用 this 指针值作为当前对象的名称，向名称堆栈中填写名称
        pDC->LoadName((UINT)this);
    }

    //绘制对象本身
    for(int i=0;i<m_TriList.GetSize();i++){
        m_TriList[i]->Draw(pDC);
    }
}
```

这样，无论是在正常的渲染模式下还是在选择模式下，场景绘制都调用该函数。这保证了绘制顺序与内容的完全一致。函数中通过 pDC->IsSelectionMode()判断当前的 OpenGL 模式，以决定是否给对象命名。

9.3.4 在 STLViewer 中调用拾取功能

在库 glContext.dll 和 GeomKernel.dll 中为适应 OpenGL 的选择功能增加了相应的函数后，在应用程序中调用相关函数，就可以方便地实现图形拾取操作。在应用程序 STLViewer 中，我们使用单击鼠标左键的方式来选择几何对象。若有选中，则高亮度显示该几何对象，并在信息输出浮动框中输出相关信息。要实现这个功能，只需要在类 CSTLViewerView 中增加鼠标事件 WM_LBUTTONDOWN 的消息处理函数，并进行修改。

（1）在类 CSTLViewerView 中增加鼠标消息 WM_LBUTTONDOWN 的响应函数 OnLButtonDown()。

```
protected:
//{{AFX_MSG(CSTLViewerView)
afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
```

（2）填写函数 OnLButtonDown()。

```
void CSTLViewerView::OnLButtonDown(UINT nFlags, CPoint point)
{
    CSTLViewerDoc* pDoc = GetDocument();
    ASSERT(pDoc);
```

```
        //判断模型是否为空。若为空，则不进行拾取处理
        if(!pDoc->m_Part.IsEmpty()){
            int hits;
            UINT items[64];

            //拾取处理开始
            m_pGLDC->BeginSelection(point.x,point.y);

            //在拾取模式下绘制模型
            RenderScene(m_pGLDC);

            //拾取结束，返回拾取记录和命中数
            hits = m_pGLDC->EndSelection(items);
            CEntity* ent;

            //若有命中，则高亮度显示这个对象。若该对象已经是高亮度显示，则关闭高亮度
            if(hits){
                for(int i=0;i<hits;i++){
                    ent = (CEntity*) items[I];
                    ent->SetHighlight(!ent->IsHighlight());
                }
                Invalidate(FALSE);      //刷新视图
            }

            //在屏幕底部的信息输出框中输出有关拾取的提示信息
            CString str;
            str.Format("%d entities selected",hits);
            CMainFrame* frm = (CMainFrame*) AfxGetMainWnd();
            ASSERT(frm);
            frm->m_OutputDockBar.AddMessageLine(str);
        }
        CGLView::OnLButtonDown(nFlags, point);
    }
```

9.3.5 运行程序

编译连接并运行程序。如图 9-1 **所示，单击鼠标左键，若有模型被选中，则该模型被高亮度显示，并在信息输出框中输出拾取的相关信息。图中，圆锥被选中并高亮显示。**

本章相关程序

- ch9\GLSelection：**一个** OpenGL **选择模式的应用程序。**
- ch9\glContext：**增加了** OpenGL **选择功能后的几何图形显示库。**
- ch9\GeomKernel：**增加了** OpenGL **选择功能后的几何内核库。**
- ch9\Enhanced STLViewer：**增加了** OpenGL **拾取功能后的** STLViewer **应用程序。**

- ch9\inc：工程所需要的头文件。
- ch9\lib：工程所需要的静态库。
- ch9\bin：执行程序 Enhanced STLViewer.exe 和它所需要的动态库。
- ch9\Models：供执行程序 Enhanced STLViewer.exe 调用的 STL 模型和 mdl 文件。
- STLViewer：整个 STLViewer 工程，包括所有 DLL 库的工程。