

用 GDI + 和面向对象设计方法构建交互式图形平台

周方晓^{1,2}, 李昌华¹, 丁有军¹

(1 西安建筑科技大学 信息与控制工程学院, 陕西 西安 710055; 2 西安建筑科技大学 土木工程学院, 陕西 西安 710055)

摘 要: 提出了一个 GDI + 和设计模式相结合的设计方案. 首先介绍一种分层式文档 - 视图框架. 在此框架基础上, 分析几种重要交互功能如图元组合、undo / redo、图面缩放及橡皮筋线的设计过程. 使用结果表明, 构建的平台满足交互功能需求, 并具有良好的可复用性.

关键词: GDI + ; 交互式图形; 面向对象设计; 设计模式; 软件可复用性

中图分类号: TP391

文献标识码: A

文章编号: 1000 - 7180(2010)10 - 0165 - 05

Using GDI + and Object - Oriented Design to Architecture Interactive Graphics Platform

ZHOU Fang-xiao^{1,2}, LI Chang-hua¹, DING You-jun¹

(1 School of Information and Control engineering, Xi'an University of Architecture and Technology, Xi'an 710055, China;

2 School of Civil Engineering, Xi'an University of Architecture and Technology, Xi'an 710055, China)

Abstract: We propose a platform built by GDI + and design patterns. Firstly, a hierarchical document - view framework is introduced. Then, we analyze the design process of important interactive features which are composite of graphics element, undo/redo, zoomin/out and rubber line. The results show that platform is meet the needs of interactive features and has good reusability.

Key words: GDI + ; interactive graphics; object - oriented design; design patterns; software reuse

1 引言

图形设备接口 (Graphics Device Interface plus, GDI +) 是微软为 Windows XP 版本提供的图形引擎^[1]. 作为 .NET Framework 的组成部分, 它是微软为用户提供的与设备无关的图形开发库. GDI + 是基于对象的, 它把一些细节封装到了对象内部, 这使得用户不必再去了解复杂的细节, 这个特点非常有利于用面向对象方法设计系统^[2].

按软件开发过程的不同阶段, 面向对象方法可以分为面向对象分析、面向对象设计、面向对象编程和面向对象测试. 面向对象设计在面向对象分析的基础上, 细化重组类及类之间的关系, 并利用设计模式对一组类进行局部设计, 使之成为粒度上比框架

(framework) 小但比类大的单元.

文中基于 GDI +, 采用面向对象设计方法, 特别是设计模式^[3], 介绍一个交互式图形平台系统的设计方案.

2 图形平台简介

2.1 平台框架

目前, 使用最多的框架是 MVC, 它把程序分成三个部分^[4]: 模型 (Model)、视图 (view) 和控制器 (Controller). 其中, 模型主要负责数据管理. 视图负责为用户显示模型数据. 而控制器则负责处理系统与用户的交互以及模型数据的操纵.

MVC 有一些变形, 在微软的开发环境中, 最常用的文档 - 视图架构就是一种简化的 MVC 框

架^[5]. 它把 MVC 中的视图、控制器合二为一, 成为文档视图框架中的视图部分, 而模型部分等同于文档视图框架中的文档部分. 由于在消息处理、显示控件的生成等方面, .Net 开发环境为编程人员提供了强大的支持, 使得框架中的视图、控制器部分的代码量大大减少. 文档-视图框架并不会因为合并而引起职责划分上的混乱, 相反可以将设计简化.

本图形平台采用分层式的文档-视图框架, 即把与用户交互的操作放到视图层, 而把数据处理的操作放到文档层, 二者是分离的. 模块的部署采用的是动态链接库的方式, 即把具体的操作、数据结构等封装到动态链接库中, 供框架调用. 数据表达和数据处理的分离以及基于动态链接库的部署方式提高了平台的可复用性. 如图 1 所示.

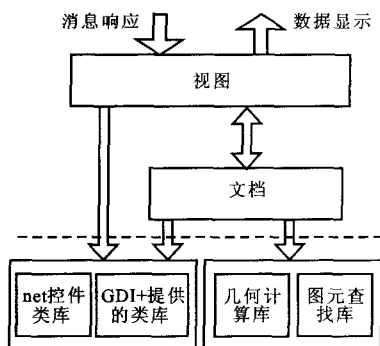


图 1 图形平台框架

2.2 平台的功能结构

平台的功能结构, 如图 2 所示.

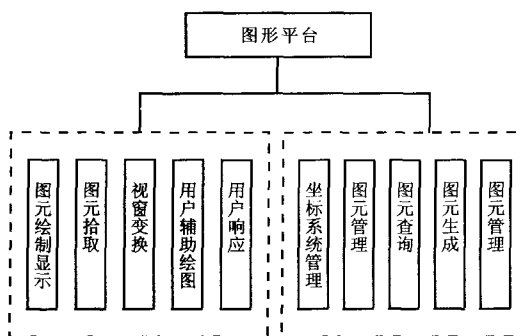


图 2 图形平台功能结构图

平台框架的实现主要采用了两个包, Canvas 包和 Model 包. 前者对应框架的视图部分, 后者对应框架的文档部分. 从功能上划分, Canvas 包主要实现图 2 中左虚线框内所列的功能, Model 包负责右虚线框内所列功能. 具体说, Canvas 包负责: ①鼠标、键盘等的消息处理, 即响应用户请求; ②图元的绘制和显示; ③在屏幕中选择拾取图元; ④视窗的放大缩小与漫游; ⑤方便用户设置的辅助绘图功能, 如正交

线、吸附、橡皮筋线等. Model 包负责: ①坐标系统的变换与计算; ②图元的管理, 包括图元的组合、增加、删除、修改、保存; ③查询指定图元; ④生成创建图元; ⑤管理平台的命令等. 这些功能及其子功能大多具有交互特性, 文中针对其中的组合图元、命令管理、视窗变换和橡皮筋线, 阐述这些交互功能的实现方案.

3 交互功能的实现

3.1 图元组合

图形系统中的图元可分成两大类:

(1) 基本图元, 如点、线、圆和多边形等;

(2) 组合图元, 它由基本图元组成的复杂的图元. 在 AutoCAD 中, 有一种称作块 (Block) 实体, 就是组合图元. 它将数个基本图元组合成一个实体, 作为一个整体实现选中、平移、旋转等操作.

第一类图元绘制可以使用 GDI+ 提供的类库. GDI+ 的 System.Drawing 命名空间中有一个 Graphics 类, 此类提供了绘制 Arc (弧)、Ellipse (椭圆)、Line (直线)、Polygon (多边形) 等十余种基本图元的方法^[2].

第二类图元则使用 Composite 模式, 如图 3 所示. Line 和 Polygon 是基本图元, Group 是组合图元, 它们都是 Entity 的子类. 不同的是 Group 内部有一个容器, 存储了多个图元; 这些内部图元本身又可以是组合图元, 形成递归包含. 无论是否是组合图元, 子类图元必须实现基类 Entity 的 draw() 虚方法. 不同的是, 组合图元还需要实现 add() 和 remove() 方法, 这两个方法是组合图元中的内部容器增加和删除图元用的. 基本图元可引用 GDI+ 提供的绘制方法, 实现其 draw() 方法. 组合图元通过遍历其内部容器中的基本图元, 完成其自身绘制. 同理, Entity 如增加移动、旋转等虚操作, 组合也可以实现这些操作, 从而实现整体移动、旋转.

3.2 Undo/Redo

undo/redo 功能是指: 在用户的交互式操作过程中, 已做的一步或多步操作可以撤销或恢复. 使用 Command 设计模式可以实现 undo/redo 功能. 此模式把操作命令封装成类, 用一个抽象类 Command 类, 定义所有操作命令的接口; 用具体的 Command 子类, 执行具体操作命令. 这样做的好处: 一个是解耦 (decouple) 了命令的发起方和接收方, 双方可以互不知道, 减少程序的耦合性. 另一个是把具体命令当作对象, 便于管理.

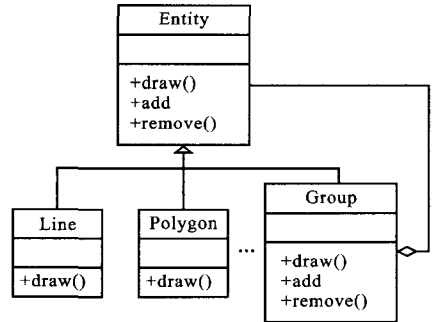


图 3 组合图元类图

用户的一组操作,对应一组具体 Command 对象。Command 对象按执行先后排列在一起,存放在 Command 堆栈中^[6]。每个具体 Command 对象中,需保存对图元的修改情况,以便执行命令的撤销操作时,恢复图元的原状。如图 4(a)。

图中 Command 类中的 state 成员变量用于保存图元的状态。客户端 Client 通过 CommandStack 类执行具体操作命令,调用 Receiver 类为具体 Command 对象设置图元的状态。

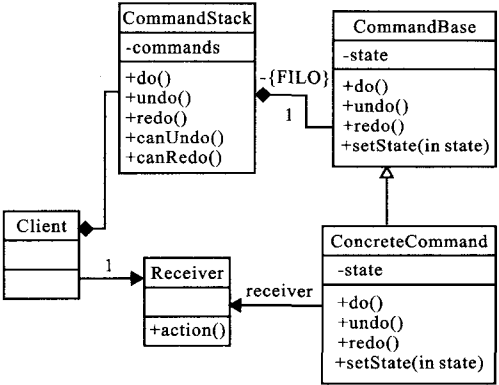
CommandStack 类是管理具体 Command 对象的类,设有两个成员变量:undo 堆栈和 redo 堆栈,前者负责 undo 命令,后者负责 redo 命令。如图 4(b)。

整个 undo/redo 操作分成三个过程:

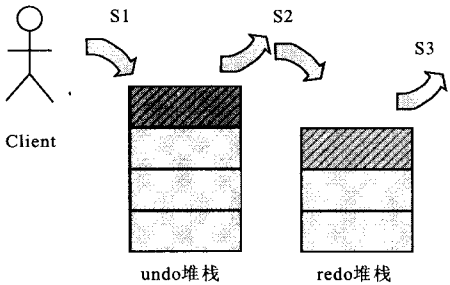
- (1) 用户调用具体 command 对象,用其 do() 方法,执行具体命令。然后,将保存了修改前状态的 command 对象,压入 undo 堆栈。
- (2) 用户需要撤销操作,点击 undo 按钮。调用 CommandStack 类的 undo() 方法,将刚才压入 undo 堆栈的 command 对象弹出栈,并调用 command 对象的 undo() 方法,恢复成命令前状态。同时,将此 command 对象压入 redo 堆栈中。
- (3) 用户需要恢复操作,点击 redo 按钮。调用 CommandStack 类的 redo() 方法,将刚才压入 redo 堆栈的 command 对象弹出栈,并将将此 command 对象压入 undo 堆栈中。

现以“移动”命令 (moveCommand) 为例,描述 Undo/Redo 命令的具体实现。moveCommand 从 CommandBase 类继承,因此需实现基类的虚方法。do() 方法中用 {entity.move(offset);} 执行“移动”命令。setState() 方法将参数 offset 存入成员变量 state 中。undo() 方法则用 {entity.move(-offset);} 将图元移回原位。redo() 则用 {entity.move(offset);} 执行 undo() 的反操作。moveCommand 对象存放在 CommandStack 堆栈中。通过 {用户 → stack.undo() →

movecommand.undo()} 过程,实现“移动”命令的撤销。



(a) 功能类图



(b) 堆栈操作过程

图 4 undo/redo 实现类图及操作过程

3.3 图面的放大缩小和漫游

图面的放大缩小 (zoom in/out) 的实现基础是计算机图形学中的窗口视区变换技术。用户观察一幅图形时,首先需要在整幅图中选取所需观察的区域,这个区域称为窗口。然后,此区域内的图形要投影到屏幕上的一个区域,为用户所见。屏幕上的区域称为视区。一般地说,视区的大小基本不变,而窗口的大小是随着用户的选择时常变化的。将大小变化的窗口,投影到大小不变的视区,同时还要保证二者的量纲匹配,就可以实现图形的放大缩小功能。窗口到视区的变换称为视窗变换。视窗变换具体算法涉及视窗的平移、比例缩放等计算,详见文献^[7]。

图面漫游 (roam) 也是一种视窗变换,与图面放大缩小不同的是:图面漫游中窗口的大小不变而位置改变,而图面放大缩小中窗口大小要改变。图面漫游每改变一个位置,即实现一次视窗变换。这种连续的视窗变换,就会产生在图面上“漫游”的效果。

GDI+ 之前,图面的放大缩小和漫游需要设计者使用公式自行计算^[8],效率及精度都不高。为了解决这一问题,GDI+ 在命名空间 System.Drawing.

Drawing2D 中提供了 Matrix 类,它封装表示几何变换的 3×3 仿射矩阵^[1]. Matrix 类公开了十余个矩阵运算的方法,如 Invert(矩阵逆)、Multiply(矩阵乘)、Reset(置单位阵)、Translate(平移矩阵)以及 Scale(缩放矩阵)等.这样,与视窗变换有关的计算都可以用 Matrix 类提供的方法实现.

图 5 中,客户端 ZoomTool 对象通过绘图类 Canvas 调用 Matrix 类,对其进行平移、缩放等计算.核心类是 Canvas,它从 .net 类库提供的 UserControl 类继承.Canvas 类中有两个成员变量:一个是变换矩阵——Matrix 的实例 m-transform,一个是用以表示变换窗口的 m-viewport. Canvas 类中的 OnPaint() 函数负责所有图元的绘制和显示,它执行 transform() 用传来的 m_transform 矩阵对每个图元做图形变换,从而显示视窗变换效果.Zoom() 则改变窗口 m_viewport 的值,并用 Invalidate() 函数通知 OnPaint() 重画.需要指出,用户观察到的图元,仅限于当前窗口中所包含的图元.因此,在实际应用时每做一次视窗变换之前,需要加一个图元预处理过程.筛出不在窗口内的图元,然后再通知 OnPaint 函数仅对包含在窗口内的图元做变换运算.当图元数量很大时,预处理过程犹显必要,它能大大提高视窗变换的显示速度.

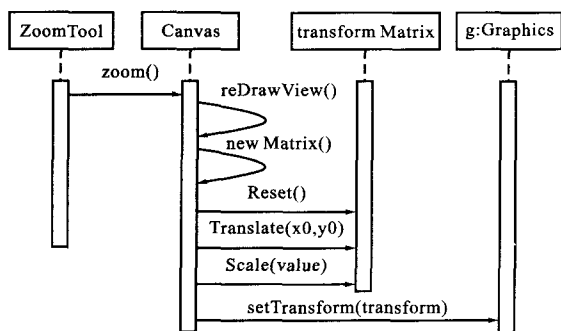


图 5 视窗变换顺序图

3.4 橡皮筋线

橡皮筋线(Rubber Line)是一项常用的交互式技术.它追踪光标位置,在操作未结束前,连续、动态地显示所绘制或所编辑的图形,操作结束则自行擦除.由于它可以随意拖拉,就像附在光标位置的橡皮筋,所以称为橡皮线.它使得用户能够看到操作过程中的动态效果,给用户带来很大方便.

实现橡皮线技术主要有两种方法:基于异或操作和基于缓冲区的橡皮线技术.本图形平台采用前一种方法.它的原理是每条线画两次,光标移动以前画一次,移动后画一次.后一次对前一次所画线上的

所有像素做异或操作.由于两次画线过程所生成的像素的 RGB 值完全一样,经异或操作后,就可以完成对第一次所画直线的擦除.具体实现时,首先用 GDI 中库函数 SetROP2() 将绘图模式设置为 R2-XORPEN,之后绘制的线就为异或线.

GDI+ 并不像 GDI 那样直接支持异或模式(这是它的一个不足之处).但它提供了直接引用 GDI 库的功能.利用这一点,用 facade 模式,设计了一个 GDIFacade 类,使得在 GDI+ 中也能画异或线,如图 6 所示.

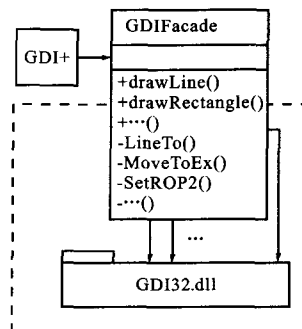


图 6 GDIFacade 类图

图中,GDI+ 通过 GDIFacade 类调用 GDI 的动态链接库(GDI32.dll)中的库函数.GDIFacade 类作为 GDI 的门面(facade)为外部提供接口,并封装内部细节.其私有函数,如 LineTo()、MoveToEx()等是从 GDI32.dll 中导出的库函数..net 提供了 DllImportAttribute 类以支持动态链接库的导入:

```
[DllImportAttribute("gdi32.dll");]
```

```
private static extern bool LineTo(IntPtr hdc, int x, int y);
```

GDIFacade 类中的公共方法 dsawLine() 等则通过调用从 GDI32.dll 中导入的库函数实现异或线功能.GDIFacade 类的实现如图 6 所示.

这样,GDI+ 就可以通过调用 GDIFacade 类中的公共方法完成异或线的绘制.从图 6 看到 facade 模式将一个复杂的系统隐藏起来(图中虚线框所示),客户端只看到 GDIFacade 类提供的简洁接口,从而实现了客户端 GDI+ 与 GDI 库之间的解耦.

4 结束语

本平台已成功应用到 GIS 建模图形平台中,如图 7 所示.从平台的开发过程看,GDI+ 提供的强大的类库支持,为开发节省了大量的工作量.从平台的使用过程看,使用诸如动态链接库部署、设计模式等面向对象方法,提高了程序的可复用性,易于平台的

升级改版.

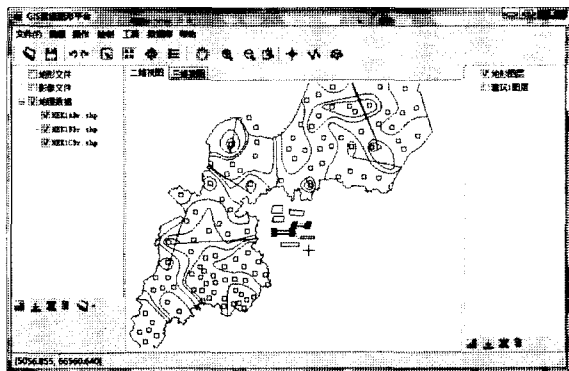


图 7 GIS 建模图形平台

作为一个图形平台,还有很多需要考虑的,像图元查找的效率、绘制的精确度、海量数据的读取等等,这些都是本图形平台需要不断优化的课题.

参考文献:

- [1] 张玉林,张树粹. 基于 VB.NET 的图像二值化与旋转处理[J]. 微电子学与计算机,2005,22(9):144-149.
- [2] Mahesh Chard. GDI+ 图形程序设计[M]. 北京:电子工

业出版社,2005.

- [3] Erich Gamma. Design patterns elements of reusable object-oriented software[M]. 北京:机械工业出版社,2002.
- [4] Peter Shirley. 计算机图形学[M]. 2版. 高春晓,译. 北京:人民邮电出版社,2007.
- [5] Yang-Ming Zhu, Steven M Cochoff. An object-oriented framework for medical image registration, fusion, and visualization[J]. Computer methods and programs in biomedicine,2006,82(3):258-267.
- [6] Eelke Folmer, Martijn van Welie, Jan Bosch. Bridging patterns: an approach to bridge gaps between SE and HCI[J]. Information and Software Technology, 2006,48(2):69-89.
- [7] 孙家广. 计算机图形学[M]. 3版. 北京:清华大学出版社,1998.
- [8] 陈建春. Visual C++ 开发 GIS 系统[M]. 北京:电子工业出版社,2004.

作者简介:

周方晓 男,(1971-),博士研究生. 研究方向为计算机图形学、软件工程.

(上接第 164 页)

对于所有的任务,当完成了每一个任务的一定百分比数量的任务时就会发生通信. 在任务执行时,不必知道确切的百分比,但需要知道发生了多少次通信还有通信量. 尽管这是一个约束,但在大部分的科学计算问题中都满足这个假设.

利用本调度算法后,完成所有任务的总花费是 134,而未使用本调度算法,使用相同资源完成相同任务的总花费是 547,使用本算法后,达到了时间和费用的平衡,使总花费达到最优.

4 结束语

文中通过对时间和费用的平衡优化,合理根据资源情况执行给定任务,为协同软件的计算任务合理调度资源,同时考虑了任务相关的调度情况,满足协同计算的要求. 实践证明了算法是有效的.

参考文献:

- [1] Raman R, Livny M, Solomon. Matchmaking: distributed

resource management for high throughput computing[C]// Proceedings of the 7th IEEE International Symposium on Performance Distributed Computing. Chicago, Illinois, 1995:25-30.

- [2] 史美林,向勇,杨光信. 计算机支持的协同工作理论与应用[M]. 北京:电子工业出版社,2000.
- [3] 张学敏,陈建新. 网格环境下一种改进的 Gossip 资源聚集算[J]. 微电子学与计算机,2009,26(1):55-58.
- [4] 李雅红. 基于 J-sim 的主动结点资源访问控制策略实现[J]. 微电子学与计算机,2009,26(4):230-233.
- [5] Rajasekar A K, Moore R W. Data and metadata collections for scientific applications[C]// European High Performance Computing Conference. Holland, Amsterdam, 2001.
- [6] 陈洪娜,黄洪钟,赵宏. 协同开发过程任务粒度设计的度量模型[J]. 西安交通大学学报,2006,40(1):57-62.

作者简介:

王世安 男,(1971-),副教授,高级工程师. 研究方向为算法设计与分析、计算机仿真.